UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Department of Electronics and Computer Science

A Joint Algorithm and Architecture Design Approach to Joint Source and Channel Coding Schemes

by

Matthew Brejza

Thesis for the degree of Doctor of Philosophy

Supervisors:
Professor Lajos Hanzo,
Dr Robert G. Maunder,
Professor Bashir M. Al-Hashimi

December 2016

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
Department of Electronics and Computer Science

Doctor of Philosophy

A JOINT ALGORITHM AND ARCHITECTURE DESIGN APPROACH TO JOINT SOURCE AND CHANNEL CODING SCHEMES

by Matthew Brejza

Shannon's separate source and channel coding theorem suggests that communication of throughputs closely approaching the capacity of the channel can be achieved using an near-entropy source code to compress a source by removing all redundancy, combined with a near-capacity channel code which adds redundancy to increase the resilience to transmission errors. However, in practice, Separate Source and Channel Coding (SSCC) can impose excessive delay and complexity, or cannot tolerate any transmission errors without causing an endless cascade of errors. This motives Joint Source and Channel Coding (JSCC), where the residual redundancy from a non-optimal source code is used by the channel code in order to increase the error correction capability. In particular, the recently proposed Unary Error Correction (UEC) code is an example of a JSCC scheme, which is well suited to encoding symbols generated during multimedia transmission, such as a H.264 or H.265 video encoder. Despite this, the UEC is only suitable for encoding symbols that are generated according to a limited range of probability distributions. Furthermore, due to their computational complexity, iterative decoder components such as source and channel decoders are usually implemented using specialized dedicated hardware. Despite this, there is little work in the open literature on the hardware implementation of JSCC schemes. Against this background, this thesis jointly considers the algorithm and architecture design of joint source and channel codes for the first time, in order to achieve an increased error correction performance and an improved hardware efficiency.

This thesis begins by proposing improvements to the UEC JSCC scheme. Firstly, an adaptive activation order algorithm is extended for use with a more complex UEC scheme, which comprises four iterative components, including a demodulator. This adaptive activation order algorithm facilitates an improved error correction performance, using a reduced number of iterations. Following this, in order to increase the applicability of the UEC code, it is extended and generalized to obtain the novel RiceEC and ExpGEC codes. These codes can be applied to any arbitrary unbound monotonic symbol distribution, including the symbols produced by the H.265 video codec and the letters of the English alphabet. Furthermore, the practicality of the proposed codes is enhanced to allow a continuous stream of symbol values to be encoded and decoded using only fixed-length system components.

This thesis also provides the first hardware implementations of UEC schemes. Owing to their relatively high complexity, many capacity-approaching techniques proposed in the literature have not yet been invoked in Wireless Sensor Network (WSN) applications, despite their potential benefits of facilitating a reduced transmission power or extended communication range. Against this background, this thesis proposes an energy-efficient architecture comprised of multiple Calculation Units (CUs), which is sufficiently flexible for accommodating different iterative decoder components of a UEC-based JSCC scheme, using the same hardware. This architecture achieves a throughput suitable for low-speed video applications, while achieving high hardware utilisation, which is important in cost- and energy-sensitive applications.

Following this, a UEC scheme is implemented for very high throughput applications, by extending the philosophy of the Fully Parallel Turbo Decoder (FPTD). More specifically, in the wireless transmission of multimedia information, the achievable *transmission* throughput and latency may be limited by the *processing* throughput and latency associated with source and channel coding. For example, ultra-high throughput and ultra-low latency processing of source and channel coding is required by the emerging new video transmission applications, such as the first-person remote control of unmanned vehicles. Here, a new architecture is developed by jointly considering the algorithm and hardware implementation, in order to achieve an improved hardware efficiency, high throughput, and low latency. This thesis will demonstrate the application of these improvements to both the LTE turbo code and the UEC code, where the proposed design achieves a throughput of 450 Mbps on a mid-range FPGA, as well as a factor of 2.4 hardware efficiency improvement over previous implementations of the FPTD.

List of publications

M. F. Brejza, L. Li, R. G. Maunder, B. Al-Hashimi, C. Berrou and L. Hanzo (2016). 20 Years of Turbo Coding and Energy-Aware Design Guidelines for Energy-Constrained Wireless Applications. *IEEE Communications Surveys & Tutorials, 18*(1), 8–28.

M. F. Brejza, T. Wang, W. Zhang, R. G. Maunder, B. Al-Hashimi and L. Hanzo (2016). Exponential Golomb and Rice Error Correction Codes for Generalized Near-Capacity Joint Source and Channel Coding. *IEEE Access, 4*, 7154–7175.

M. F. Brejza, R. G. Maunder, B. Al-Hashimi and L. Hanzo. A High-Throughput FPGA Architecture for Fully-Parallel Joint Source and Channel Decoding. *IEEE Access, PP*(99), 1–1.

M. F. Brejza, R. G. Maunder, B. Al-Hashimi and L. Hanzo. Flexible Iterative Receiver Architecture for Wireless Sensor Networks: A Joint Source and Channel Coding Design Example. Accepted by *IET Wireless Sensor Systems.*

M. F. Brejza, W. Zhang, R. G. Maunder, B. Al-Hashimi and L. Hanzo (2015). Adaptive iterative detection for expediting the convergence of a serially concatenated unary error correction decoder, turbo decoder and an iterative demodulator. In *IEEE International Conference on Communications (ICC), 2015* (pp. 2603–2608).

M. F. Brejza, J. Hooker, J. Sowman, D. Oakley and R. G. Maunder (2016). Design of Digital Testbeds for Undergraduate Microelectronics Teaching. In *11th European Workshop on Microelectronics Education.*

W. Zhang, **M. F. Brejza**, T. Wang, R. G. Maunder, and L. Hanzo (2015). Irregular Trellis for the Near-Capacity Unary Error Correction Coding of Symbol Values From an Infinite Set. *IEEE Transactions on Communications, 63*(12), 5073–5088.

W. Zhang, Y. Jia, X, Meng, **M. F. Brejza**, R. G. Maunder and L. Hanzo (2015). Adaptive iterative decoding for expediting the convergence of unary error correction codes. *IEEE Transactions on Vehicular Technology, 64*(2), 621–635.

W. Zhang, Z. Song, **M. F. Brejza**, T. Wang, R. G. Maunder and L. Hanzo (2016). Learning-aided unary error correction codes for non-stationary and unknown sources. *IEEE Access, 4*, 2408–2428.

T. Wang, **M. F. Brejza**, W. Zhang, R. G. Maunder and L. Hanzo (2016), Reordered Elias Gamma Error Correction Codes for the Near-Capacity Transmission of Multimedia Information. *IEEE Access, 4*, 5948–5970.

Contents

Declaration of Authorship

I, Matthew Brejza , declare that the thesis entitled *A Joint Algorithm and Architecture Design Approach to Joint Source and Channel Coding Schemes* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published, as seen in the list of publications

Signed:...

Date:..

Acknowledgements

I would like to thank my supervisors Dr Rob Maunder, Professor Lajos Hanzo and Professor Bashir Al-Hashimi for their outstanding help and support, as well as sharing their knowledge and experience during the course of my studies. In particular, this thesis has benefited immensely from their quick feedback, useful insight and guidance. I would also like to extend my gratitude to my colleagues for their valuable input during our discussions over the past four years.

Nomenclature

A	Area beneath the inverted EXIT function.
A_{Amp}	Power amplifier efficiency.
a, b, c	Bit or LLR vector lengths.
B	Bandwidth.
C	DCMC capacity.
E_b	Energy per bit.
E_s	Energy per symbol.
$\mathbf{d}$, $\mathbf{x}$, $\mathbf{t}$	Stream of symbols or sub-symbols at the transmitter.
$\mathbf{b}$, $\mathbf{y}$, $\mathbf{z}$, $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$	Bit vectors at the transmitter.
$\hat{\mathbf{d}}$, $\hat{\mathbf{x}}$, $\hat{\mathbf{t}}$	Stream of decoded symbols or sub-symbols.
$\tilde{\mathbf{y}}$	Vector of *a posteriori* LLRs at the receiver.
$\tilde{\mathbf{b}}^{\mathrm{a}}$, $\tilde{\mathbf{z}}^{\mathrm{a}}$, $\tilde{\mathbf{u}}^{\mathrm{a}}$, $\tilde{\mathbf{v}}^{\mathrm{a}}$, $\tilde{\mathbf{w}}^{\mathrm{a}}$	Vector of *a priori* LLRs at the receiver.
$\tilde{\mathbf{b}}^{\mathrm{e}}$, $\tilde{\mathbf{z}}^{\mathrm{e}}$, $\tilde{\mathbf{u}}^{\mathrm{e}}$, $\tilde{\mathbf{v}}^{\mathrm{e}}$, $\tilde{\mathbf{w}}^{\mathrm{e}}$	Vector of extrinsic LLRs at the receiver.
H	Symbol entropy.
I	Number of decoding iterations.
i,j,k	Bit or symbol indices.
k_{ExpG}	Parameter of the ExpG code.
l	Average codeword length.
L	Symbol alphabet cardinality.
M_{Rice}	Parameter of the Rice code.
M_{Mod}	Number of constellation points.
m	Trellis state.
$\mathbb{N}_1$	The set of natural numbers, $\mathbb{N}_1 = \{1, 2, 3, ...\}$.
N	Frame length.
n	Codeword length.
P	Number of hardware processing elements.
p_1	Probability of the value 1 in a zeta distribution.
R	Coding, puncturing or doping rate.
r	Number of states in a trellis.
s	Parameter of the zeta function.
T	Trellis transition.
$x_{\max}$, $d_{\max}$	Maximum value of sub-symbol considered by the FLC decoder.

α, β, γ, δ	α, β, γ and δ calculations of the BCJR algorithm.
η	Effective throughput.
ω	Window length.
3GPP	3rd Generation Partnership Project
ACS	Add-Compare-Select
ALM	Adaptive Logic Module
ALUT	Adaptive Look-Up Table
APTD	Arbitrarily Parallel Turbo Decoder
ASIC	Application-Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BCJR	Bahl-Cocke-Jelinek-Raviv
BER	Bit Error Ratio
BPSK	Binary Phase-Shift Keying
CC	Convolutional Code
CRC	Cyclic Redundancy Check
CU	Calculation Unit
DCMC	Discrete-Input Continuous-Output Memoryless Channel
EC	Energy Consumption
EG	Elias Gamma
EGEC	Elias Gamma Error Correction
EXIT	EXtrinsic Information Transfer
ExpG	Exponential Golomb
ExpGEC	Exponential Golomb Error Correction
FLC	Fixed Length Code
FLC-CC	Fixed Length Code-Convolutional Code
FPGA	Field Programmable Gate Array
FPTD	Fully Parallel Turbo Decoder
IID	Independent and Identically Distributed
IoT	‘Internet of Things’
JSCC	Joint Source and Channel Coding
LDPC	Low-Density Parity-Check
LLR	Logarithmic Likelihood Ratio
Log-BCJR	Logarithmic Bahl-Cocke-Jelinek-Raviv
LTE	Long Term Evolution
LUT	Look-Up Table
LUT-Log-BCJR	Look-Up-Table based Log-BCJR
Max-Log-BCJR	Maximum Log-BCJR
Max-SE-Log-BCJR	Maximum with Scaled Extrinsic Log-BCJR
MCTC	Multiple-Component Turbo Code
MI	Mutual Information
ML	Maximum Likelihood

PA	Power Amplifier
PIVI	Previous Iteration Value Initialization
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
RAM	Random Access Memory
RiceEC	Rice Error Correction
RNF	Receiver Noise Figure
RS	Reed-Solomon
RV	Random Variable
SBSD	Soft Bit Source Decoding
SER	Symbol Error Rate
SIMD	Single Instruction, Multiple Data
SISO	Soft-In Soft-Out
SMP	State-Metric Propagation
SNR	Signal-to-Noise Ratio
SSCC	Separate Source and Channel Coding
TC	Turbo Code
TCTC	Twin-Component Turbo Code
TSMC	Taiwan Semiconductor Manufacturing Company
UEC	Unary Error Correction
URC	Unity Rate Convolutional
VLEC	Variable Length Error-Correction
WSN	Wireless Sensor Network

Chapter 1

Introduction

In modern telecommunication systems, there is a desire to achieve ever increasing transmission throughputs, while maintaining reasonable transmit power and receiver complexity. Over the previous two decades, the high performance turbo [1] and Low-Density Parity-Check (LDPC) [2] error correction codes have been adopted extensively, since they have the ability to reduce the transmit power required to achieve reliable communications, and/or increase the transmission throughput. In addition to this, high efficiency source codes have been developed to reduce the number of bits required to convey a message, which increases the efficiency of the system. To enable the practical use of these high complexity techniques, Application-Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) implementation is generally required to achieve high processing throughputs, which match the high transmission throughputs that they afford.

Improvements to the overall system performance can be achieved by jointly designing the various parts of the system together. In particular, Joint Source and Channel Coding (JSCC) [3] takes advantage of inefficiencies in the source code in order to increase the performance of the error correction code. Likewise, by considering the algorithm and hardware implementation jointly, a reduction in the architecture's power consumption or hardware resource usage can be achieved [4].

Figure 1.1 and Table 1.1 characterises a selection of work relating to the topics of source coding and channel coding. These are categorised into algorithm design or hardware implementation, where an overlap between topics identifies papers which consider these topics jointly. Against this background, this thesis considers the algorithmic design and hardware implementation of joint source and channel coding schemes. By considering these topics together, this thesis aims to achieve an increased performance with a reduction in hardware resources, compared to an approach where the source coding and channel coding algorithms and their implementation are developed in isolation. To build upon this further, the following sections briefly outline the key topics which constitute this thesis. Following this, Section 1.5 summarises the novel contributions and structure of this thesis.

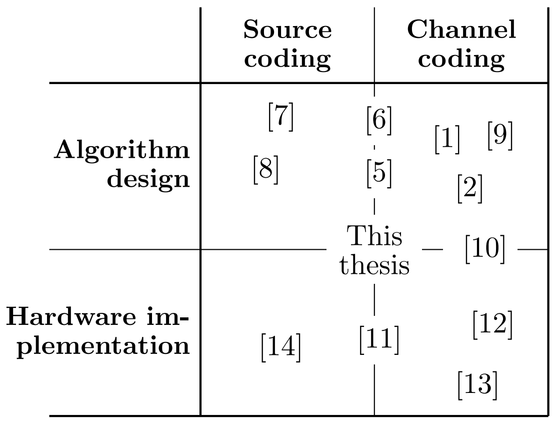


Figure 1.1: Topic map for some existing work in the topics considered by this thesis

[7]	Arithmetic code
[8]	Lempel-Ziv code
[1]	Turbo codes
[9]	Fully parallel turbo decoder
[2]	LDPC
[6]	Unary Error Correction (UEC) code
[5]	Variable Length Error-Correction (VLEC) code
[11]	A JSCC ASIC
[12]	Long Term Evolution (LTE) ASIC
[13]	2.15 Gbps turbo decoder
[14]	Arithmetic code ASIC
[10]	Turbo decoder memory optimisation

Table 1.1: Some existing work in the topics considered by this thesis

1.1 Source coding

Source coding is used for the conversion of symbols provided by a source such as a video encoder into a series of bits, which can then be transmitted over a digital wireless link. Figure 1.1 shows some examples of sophisticated source codes from the literature. Each symbol source emits symbols from a particular set of values, where each symbol value has a particular probability of occurring. For example, a particular source may emit symbols having values selected from the set $\{1, 2, 3, 4, 5, 6\}$, with the corresponding probabilities $\{0.30, 0.25, 0.15, 0.15, 0.10, 0.05\}$.

Table 1.2: Example codewords for different source codes

Symbol	1	2	3	4	5	6
Probability	0.30	0.25	0.15	0.15	0.10	0.05
Binary codewords	000	001	010	011	100	101
Huffman codewords	11	01	101	100	001	000
Unary codewords	1	01	001	0001	00001	000001

With this example, the most simple source code would represent each symbol using its binary representation, where $\lceil \log_2(6) \rceil = 3$ bits are required for the 6 different symbols, as shown in Table 1.2. This is an inefficient source code, since it does not take into account the probabilities of each symbol. Since each codeword in the binary scheme has the same length of 3 bits, the average codeword length is $l_{\text{Bin}} = 3$ bits. An improvement is offered by the Huffman code [15], which produces variable length binary representations of each possible symbol value, as shown in Table 1.2. More specifically, the Huffman code allocates shorter codewords to more frequently occurring symbols and longer codewords to the rarer codewords. In this way the average codeword length of the Huffman code is reduced to $l_{\text{Huff}} = 2.45$ bits. The entropy of a symbol source defines the minimum theoretical number of bits needed to represent a set of source symbols [16]. In the example of Table 1.2, the entropy of the source symbols is $H = 2.39$ bits, showing the Huffman code achieves close to the theoretical limit.

In the example of Table 1.2, the number of symbols in the set is limited to 6. However, much larger alphabets may be encountered in practical schemes. In this thesis, the infinite symbol set will be explored, where an arbitrarily high symbol value could be produced, but with successively lower probabilities for increasing symbol value. When the size of symbol set is large, or when the probabilities of occurrence change over time, source codes such as the Huffman code become impractical. In this case, a code such as the unary code of Table 1.2 may be preferred, owing to its simpler structure, which does not require a codeword to be specifically designed to match the probability of occurrence of each symbol. However, this simplicity is typically at the cost of an increased average codeword length, where $l_{\text{Unary}} = 2.65$ results for the example of Table 1.2.

1.2 Channel coding

In contrast to source coding which aims to remove redundancy from a message, the channel code intentionally adds redundant bits to a message before its transmission over a noisy channel. Since the value of the bits received may be changed by errors induced by the channel, a channel decoder uses these redundant bits to attempt to recover the original message. This avoids the requirement for the transmitter to re-transmit the message, or increase the transmit power for subsequent messages to avoid errors occurring. However, the redundant bits transmitted by the error correction code will also consume additional energy at the transmitter. Owing to this, different schemes are compared according to how much energy per message bit (E_b) is required for reliable communication.

The performance of an error correction code can be characterised by how close it performs compared to the theoretical limit. More specifically, the Shannon limit [16] specifies the maximum achievable spectral efficiency, given the Signal-to-Noise Ratio (SNR) at the receiver $E_{\text{b}}^{\text{rx}}/N_0$, where N_0 is the noise power spectral density. Figure 1.2 graphically illustrates the theoretical limit, in the case where BPSK modulation is used to transmit bits over an uncorrelated narrowband Rayleigh fading channel. Figure 1.2 characterises a 1/3-rate repetition code which is a simple error correction code that duplicates the transmission of each bit three times. The error correction capability relies on at least two copies of each bit being received without corruption, allowing the third corrupted

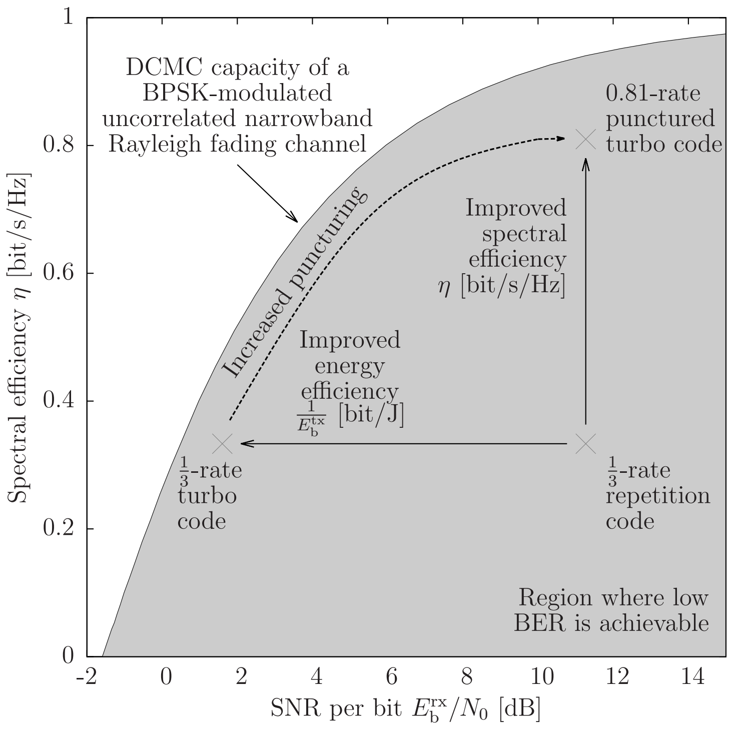


Figure 1.2: Performance of error correction codes compared to the theoretical limit for BPSK communication over an uncorrelated narrowband Rayleigh fading channel. From ©IEEE [17].

bit to be out voted. Here, the coding 'rate' refers to the fraction of the transmitted bits that are message bits. Figure 1.2 shows that the 1/3-rate repetition code has a spectral efficiency of 1/3 bit/s/Hz when employing BPSK, and requires an SNR of about 11 dB in order to achieve reliable communication. By contrast, turbo codes [1] are sophisticated error correction codes, used by modern communication standards such as LTE [18]. Figure 1.2 characterises a 1/3-rate turbo code, which has the same coding rate as the repetition code, but owing to the turbo code's increased error correction capability, the required SNR at the receiver is now only about 1 dB. The turbo code may be considered to be a near-capacity code, since it requires an SNR that is only about 1 dB greater than the theoretical limit. Alternatively, the turbo code may operate by transmitting only a reduced selection of its redundant bits, using a technique known as puncturing. Figure 1.2 shows how a punctured 0.81-rate turbo code requires the same receive SNR as the 1/3-rate repetition code, but achieves a significantly improved spectral efficiency of 0.81 bit/s/Hz.

As shown on Figure 1.1, turbo codes and LDPC codes [2] are examples of near-capacity iterative error correction codes. At the receiver, the turbo or LDPC decoder relies on two or more component decoders, which iteratively exchange information between themselves, such that each decoder aids the other to achieve successful decoding. This iterative approach leads to an attractive tradeoff between performance and complexity.

However, despite the complexity reduction offered by the iterative decoding methods relative to powerful non-iterative codes, turbo and LDPC decoders still have a high decoding complexity. For example, the LDPC code was originally proposed by Gallager in 1962 [2], however it was not until 1997 [19] when they were first considered suitable for use. More specifically, the authors of [20] considered the power consumption of the various components of a transceiver, finding that for the range of LTE base-stations which were considered, the turbo code consumes approximately the same power as all the other baseband radio components combined. Additionally, it was found for the smallest 'femto' base-stations that the turbo code also consumes approximately the same power as the Power Amplifier (PA) components. This demonstrates that the energy consumption required for error correction decoding is significant, especially for battery powered devices. Since error correction codes also have high complexity, the associated hardware cost required for high speed implementation is also a concern for wireless devices.

1.3 Joint source and coding

In Separate Source and Channel Coding (SSCC), a source code is combined with a channel code, as discussed in Sections 1.1 and 1.2, respectively. Here, the source code aims to remove as much redundancy as possible, while the channel code adds redundancy to allow error correction to be performed at the receiver. In contrast, JSCC [3] considers the source coding and channel coding jointly, which enables the channel code to take advantage of any residual redundancy that remains after source encoding, in order to enhance the error correction. Typically, a simple source code will be used, which may be intentionally chosen to give more excess redundancy, so that it can be used by the channel code. For example, Table 1.2 shows the unary source code [21] alongside the Huffman code. In this example, the unary source code has an average length of $l_{\text{Unary}} = 2.65$, which compares to $l_{\text{Huff}} = 2.45$ for the Huffman code, as discussed in Section 1.1. Of these two options, the Huffman code would be the best choice for SSCC, since it removes most of the redundancy. However, the unary code may be the best choice for JSCC since the channel code can exploit the extra redundancy, in order to improve error correction.

Figure 1.1 shows a selection of papers on JSCC, which consider the source code and channel code together, in order to offer performance enhancements over the equivalent SSCC schemes. In particular, the UEC code [6] is an example of JSCC which can encode source symbols from a large or infinite set, such as those generated by a H.265 video encoder. This code achieves near capacity operation, without requiring knowledge of all source symbol probabilities, and without requiring careful synchronisation between the encoder and decoder. By contrast, near-entropy source codes such as the arithmetic code and Lempl-Ziv [8] code require both the transmitter and receiver to have accurate knowledge of the probability of each symbol, which can impose large memory requirements when using large symbol sets. In these schemes, if the transmitter and receiver learn the symbol probabilities to avoid the transmitter having to convey them to the receiver, then any transmission errors will lead to the receiver becoming de-synchronised from the transmitter, which may introduce a large number of errors thereon [22].

However, the UEC code is not a universal code, which means that it cannot be applied for all symbol set probabilities. Motivated by this, this thesis proposes the Exponential Golomb Error Correction (ExpGEC) and Rice Error Correction (RiceEC) codes, which extend and improve the applicability of the UEC code. In particular, this thesis aims to offer performance gains without increasing the complexity required to achieve these gains. In contrast, the VLEC [23] code is an example of JSCC which offers a performance improvement compared to SSCC schemes, but requires an order of magnitude more complexity than SSCC schemes. This thesis shows that the ExpGEC and RiceEC schemes can offer the same performance improvements as the VLEC code, while using the same complexity as the SSCC schemes.

1.4 Joint algorithm and hardware design

This introduction has so far described the algorithmic methods which can be employed for reduced transmit power and increased throughput applications. However, in order to enable their practical use, the hardware implementation of these techniques must be considered.

When implementing an algorithm in hardware, there are tradeoffs which have to be made. For example, the algorithm may have originally been designed using floating point and high complexity mathematical operations. In order to reduce the complexity, the hardware implementation may use fixed point numbers with a more limited precision, and approximations of the mathematical operations, which typically lead to a performance degradation. In the case of an iterative decoder, such as those used by channel codes discussed in Section 1.2, these approximations may still approach the performance of the original algorithm, but requiring an increased number of iterations. Therefore, tradeoffs must be made between the complexity reduction achieved and the performance impairments associated with techniques such as using fixed point numbers.

In order to improve the overall performance of an implemented design, techniques must be developed which consider both the algorithm and hardware, some of which are shown in Figure 1.1. For example, techniques such as extrinsic scaling [24, 25] reduce the performance impact of the complexity reducing approximations made within an error correction decoder. Understanding how the algorithm works, and considering algorithmic aspects is also important when attempting to increase the throughput of an implementation. In particular, each iterative component of the turbo decoder is comprised of long serial operations, which imposes restrictions upon its parallelism and hence its throughput. In the case of a turbo decoder, the highly serial processing also increases the Random Access Memory (RAM) requirement of a decoder. Motivated by these limitations, various techniques have been proposed to modify the turbo decoding algorithm, in order to achieve a considerably greater degree of parallelism, and therefore throughput. However, these techniques may also impact upon the performance of the algorithm, which must also be considered during the joint algorithm and hardware design. The various algorithmic and architectural factors are shown in Figure 1.3, characterising the design trade-offs that must be considered.

As shown in Figure 1.1, the majority of previous work on the joint design of algorithms and hardware implementations has considered only the channel code, or only the source code, with no papers jointly considering the algorithm and hardware implementation of a JSCC scheme. Upon this background, this thesis jointly considers the algorithm and hardware design of JSCC schemes, for the first time. Furthermore, this thesis will show how the high performance implementation techniques that have been previously developed for channel codes can also be applied to JSCC schemes.

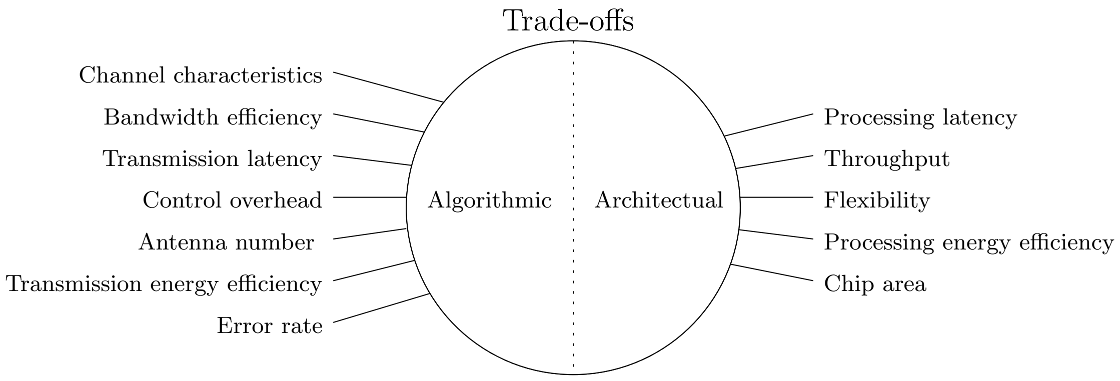


Figure 1.3: Design trade-offs in a communication system.

1.5 Outline and novel contributions

This section details the structure of the thesis, and lists the novel contributions of each chapter. Figure 1.4 shows which of the topics discussed in this introduction are considered by each of the chapters.

This thesis will continue in Chapter 2 with a background review. This will introduce and discuss the Log-BCJR error correction algorithm [17] and its implementation, which forms the basis of the iterative component decoders within the turbo code. An example of holistic design is provided, which applies the joint hardware and algorithm design philosophy of Section 1.4 to characterise the overall energy consumption in various example scenarios. This is followed by a discussion of EXtrinsic Information Transfer (EXIT)

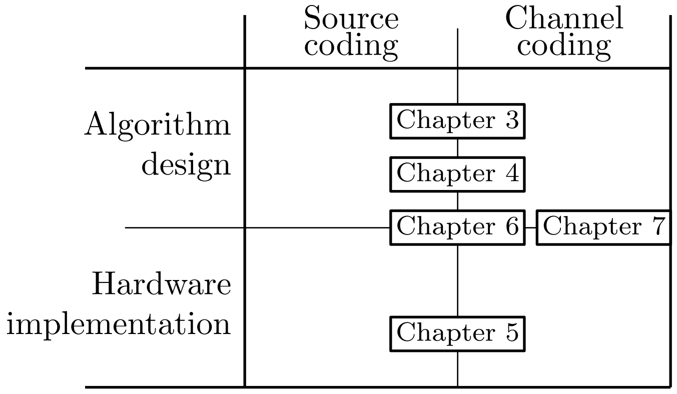


Figure 1.4: The topics covered by the chapters in this thesis.

functions [26], which may be used to characterise iterative decoders. Finally, the operation of the UEC code and its encoder and decoder will be discussed.

Chapter 3 will detail the adaptive activation algorithm for a UEC scheme. Here, EXIT charts are used in an on-line fashion for deciding in which order to activate each of the iterative decoding blocks, allowing decoding to be completed with fewer iterations. This chapter shows that for a fixed complexity, the adaptive UEC scheme offers a performance improvement over the non-adaptive schemes, including SSCC benchmarkers. As shown in Figure 1.4, this chapter considers primarily the algorithmic aspects of a joint source and channel code, however some consideration is given to the ease of implementation, specifically by limiting the complexity, and by applying approximations to reduce the adaptive algorithm's complexity. The novel contributions of this chapter are as follows.

- The adaptive algorithm of [27] is extended to a UEC scheme having four iterative components, including an iterative demodulator. This requires the adaptive algorithm to be modified in order to operate on a scheme featuring both parallel and serial concatenations.
- We reduce the complexity of the Mutual Information (MI) measuring operation, that is used to decide which decoding block to activate next, by removing the need for exponential and logarithmic functions.
- The inclusion of the demodulator into the iterative decoding process is shown to reduce the overall complexity of the scheme, since it provides a more complementary match between lower complexity decoder components.

In Chapter 4, the applicability of the UEC code is increased. As discussed in Section 1.3, the UEC code cannot be applied to all symbol set probabilities, and performs poorly in some others. The work of [28] proposed the Elias Gamma Error Correction (EGEC) code, which splits each input symbol into two sub-symbols, one of which exploits the UEC code. This chapter extends the work of [28], where the ExpGEC and RiceEC codes are proposed. These family of codes are shown to have greater applicability than the UEC or EGEC codes. Figure 1.4 shows that the work of this chapter mainly concerns the algorithmic aspects of the proposed JSCC schemes, however the implementational aspects are also considered. A complexity limit is imposed in order to demonstrate that the offered improvements do not come at the cost of extra complexity, while the practicality of the interleavers is also considered. The novel contributions of this chapter are as follows.

- The RiceEC and ExpGEC codes are proposed, which extend the EGEC of [28]. These codes use the UEC for near-capacity operation.
- The performance of the RiceEC, ExpGEC, UEC and EGEC schemes are characterised for a wide range of symbol sets, including the English alphabet and H.265 video data. The results show that the proposed family of codes can offer near-capacity performance for a wide range of symbol sets, where symbol sets with both large and small cardinality are considered.
- The wide range of considered symbol sets are analysed for each of the proposed codes, and the key aspects which impact upon the performance are discussed.
- A novel method is proposed to allow symbols and their variable-length codeword representations to be streamed through the encoder with fixed-length interleavers. This method is shown to be immune to synchronisation issues that might otherwise occur with transmission errors.

As shown in Figure 1.4, Chapter 5 details the implementation of a decoder for a JSCC scheme. This decoder is designed to be flexible, such that the same hardware can be used for the different constituent decoders of the JSCC scheme. This scheme is similar to that of Chapter 3, where a UEC scheme is used in conjunction with an iterative demodulator. The implementation is targetted towards Wireless Sensor Network (WSN) applications, hence the architecture has a low power consumption and small chip area. The novel contributions of this chapter are as follows.

- The same hardware is used for the different parts of the receiver, namely the UEC decoder, the two Unity Rate Convolutional (URC) decoders, and the demodulator.
- A controller was developed to handle the scheduling of the three different types of decoder, and to ensure that the hardware is kept as busy as possible, in order to achieve a high utility and hardware efficiency.

Chapter 6 also details the implementation of a UEC scheme. In contrast to Chapter 5, this implementation targets high throughputs and low latencies by vastly increasing the parallelism of the decoder, while offering a hardware efficiency improvement over similar implementations. As shown in Figure 1.4, this chapter also considers the algorithm alongside the implementation. More specifically, the decoding algorithm is modified while considering how it will be implemented, allowing the hardware to achieve higher throughputs and lower latencies than would otherwise be achievable by simply using the Log-BCJR. The novel contributions of this chapter are as follows.

- A novel 'paired scheduling' decoder, derived from the fully parallel decoder of [9], is used to achieve high throughputs and low latencies. This paired scheduling was designed while jointly taking into consideration the algorithm and the hardware implementation, in order to ensure improved error correction performance, and an increased hardware efficiency.
- The implementation of this chapter offers several improvements over previous fully parallel implementations. Increased pipelining allows a higher clock frequency, and the modified paired scheduling requires fewer iterations to achieve the same performance as the original fully parallel decoder of [9].
- Due to the increased hardware efficiency, the paired scheduling increases the frame length that can be decoded within a given hardware size, compared to the previous fully parallel implementations.

Chapter 7 extends the work of Chapter 6, where a potential future architecture is discussed. The paired scheduling is extended such that each of the parallel hardware processing elements can vary how many bits they each decode, depending on the frame length. Figure 1.4 shows that this chapter also considers the algorithm design and hardware implementation jointly, in order to offer increased performance. While a JSCC scheme is not considered by this chapter, the work of Chapter 6 shows how the techniques of this chapter may be extended to a UEC scheme. The novel contributions of this chapter are as follows.

- The proposed architecture is compatible with all LTE frame lengths within any given hardware resource limit, where the decoder is flexible in its operation and offers a higher degree of parallelism compared to conventional Log-BCJR decoders.
- Since practical interleavers impose a few clock cycles of delay, the corresponding performance degradation is investigated.

This thesis will conclude in Chapter 8 by evaluating the completed work, and suggest future work and improvements.

Chapter 2

Background

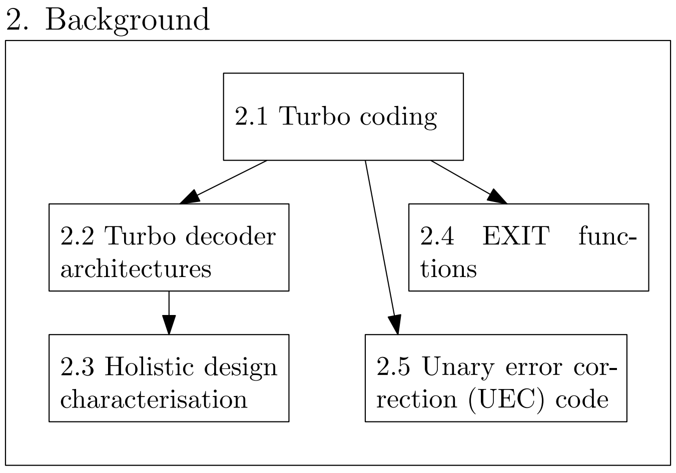

2. Background
2.1 Turbo coding
2.2 Turbo decoder architectures
2.3 Holistic design characterisation
2.4 EXIT functions
2.5 Unary error correction (UEC) code

Figure 2.1: The outline for the background chapter

The outline for this chapter is shown in Figure 2.1. This chapter commences in Section 2.1 by describing the operation of turbo codes [1] in a typical communication system, which provides key a foundation to the rest of this thesis, as well as the other background sections. Following this, Section 2.2 investigates the challenges and solutions when implementing the turbo decoder of Section 2.1 in hardware. Section 2.3 provides an example of joint algorithm and hardware design, by jointly optimizing the energy consumption of the wireless system. Here the transmitter's energy consumption is considered alongside the receiver's energy consumption, to allow the most energy efficient option to be chosen. In Section 2.4, EXIT functions are discussed, which are used to characterise and predict the operation of individual blocks of iterative decoders. Finally, Section 2.5 discusses Unary Error Correction (UEC) codes, which can be combined with turbo codes of Section 2.1 to form Joint Source and Channel Coding (JSCC)

This chapter is partially based on the following publication.
M. F. Brejza, L. Li, R. G. Maunder, B. Al-Hashimi, C. Berrou and L. Hanzo (2016). 20 Years of Turbo Coding and Energy-Aware Design Guidelines for Energy-Constrained Wireless Applications. *IEEE Communications Surveys & Tutorials, 18*(1), 8–28.

schemes, which yields improved performance compared to conventional Separate Source and Channel Coding (SSCC) schemes.

2.1 Turbo coding

In this section, the Turbo Code (TC) scheme of Figure 2.2 is introduced. More specifically, the topic of Figure 1.1 covered by this section is mostly the algorithm design of the channel code. Section 2.1.1 begins by describing the convolutional encoders, which

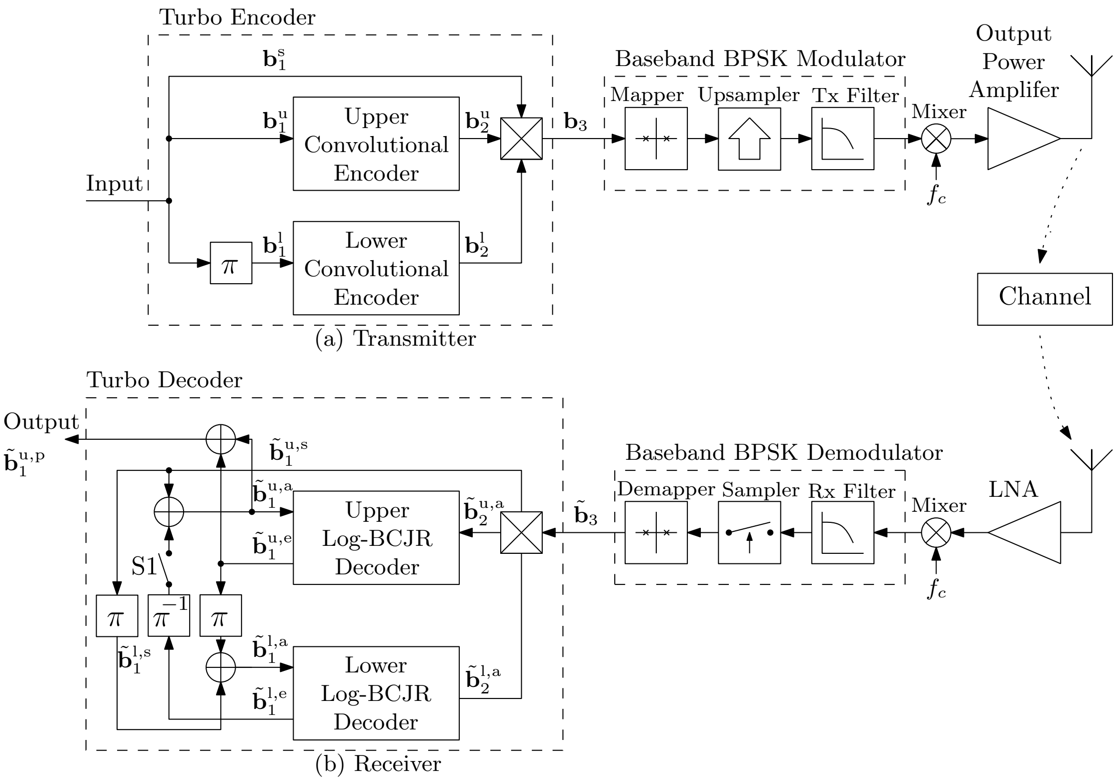

Turbo Encoder
$\mathbf{b}_1^{\mathrm{s}}$
$\mathbf{b}_1^{\mathrm{u}}$
Upper Convolutional Encoder
$\mathbf{b}_2^{\mathrm{u}}$
$\mathbf{b}_3$
Input
π
$\mathbf{b}_1^{\mathrm{l}}$
Lower Convolutional Encoder
$\mathbf{b}_2^{\mathrm{l}}$
(a) Transmitter
Baseband BPSK Modulator
Mapper
Upsampler
Tx Filter
Mixer
f_c
Output Power Amplifer
Channel
Turbo Decoder
Output
$\tilde{\mathbf{b}}_1^{\mathrm{u,p}}$
$\tilde{\mathbf{b}}_1^{\mathrm{u,s}}$
$\tilde{\mathbf{b}}_1^{\mathrm{u,a}}$
$\tilde{\mathbf{b}}_1^{\mathrm{u,e}}$
Upper Log-BCJR Decoder
$\tilde{\mathbf{b}}_2^{\mathrm{u,a}}$
$\tilde{\mathbf{b}}_3$
S1
π
π^{-1}
π
$\tilde{\mathbf{b}}_1^{\mathrm{l,s}}$
$\tilde{\mathbf{b}}_1^{\mathrm{l,a}}$
$\tilde{\mathbf{b}}_1^{\mathrm{l,e}}$
Lower Log-BCJR Decoder
$\tilde{\mathbf{b}}_2^{\mathrm{l,a}}$
(b) Receiver
Baseband BPSK Demodulator
Demapper
Sampler
Rx Filter
Mixer
f_c
LNA

Figure 2.2: A BPSK-modulated $R = 1/3$ TC scheme.

are concatenated in parallel in order to form the turbo encoder of Figure 2.2. The integration of the turbo encoder into a BPSK transmitter is discussed in Section 2.1.2. Following this, Section 2.1.3 describes the modelling of transmission over an Additive White Gaussian Noise (AWGN) channel, subject to a certain path loss. Section 2.1.4 discusses the operation of the turbo-coded BPSK receiver of Figure 2.2. This operates on the basis of the most frequently used variant of the BCJR decoder, namely the Logarithmic Bahl-Cocke-Jelinek-Raviv (Log-BCJR) decoder, which is detailed in Section 2.1.5. Modifications of the Log-BCJR algorithm are conceived for the practical implementations, which are discussed in Section 2.1.7, before the TC's error correction performance is characterized in Section 2.1.6.

2.1.1 Convolutional encoder

The convolutional encoder [29] is a widely adopted component in sophisticated error correcting schemes, forming the basis of the turbo encoder, as shown in Figure 2.2. In this application, the input of the convolutional encoder is a message frame $\mathbf{b}_1 = [b_{1,k}]_{k=1}^{N}$ comprising N bits, while the output is an N-bit encoded frame $\mathbf{b}_2 = [b_{2,k}]_{k=1}^{N}$. The parametrization of a convolutional encoder may be specified by a trellis, which graphically illustrates the relationship between the frames $\mathbf{b}_1$ and $\mathbf{b}_2$. The example trellis of Figure 2.3 corresponds to a simple convolutional encoder, which may be used for encoding a message frame $\mathbf{b}_1$ comprising $N = 5$ bits. This encoder adopts one of two possible states following the encoding of each bit $b_{1,k}$ in the frame, as represented by the dots in Figure 2.3. Depending on the value of this bit, the encoder state is selected by following one of two possible transitions from the previous state, as represented by the lines in Figure 2.3. As shown in Figure 2.3, the convolutional encoder is initialized in state 1 before encoding the first bit in the message frame $\mathbf{b}_1$. Each selected transition identifies a bit value for the encoded frame $\mathbf{b}_2$. For example, the message frame $\mathbf{b}_1 = [1, 1, 0, 0, 1]$ corresponds to the sequence of transitions that is highlighted in bold in Figure 2.3. In turn, this sequence identifies the encoded frame $\mathbf{b}_2 = [1, 0, 0, 0, 1]$.

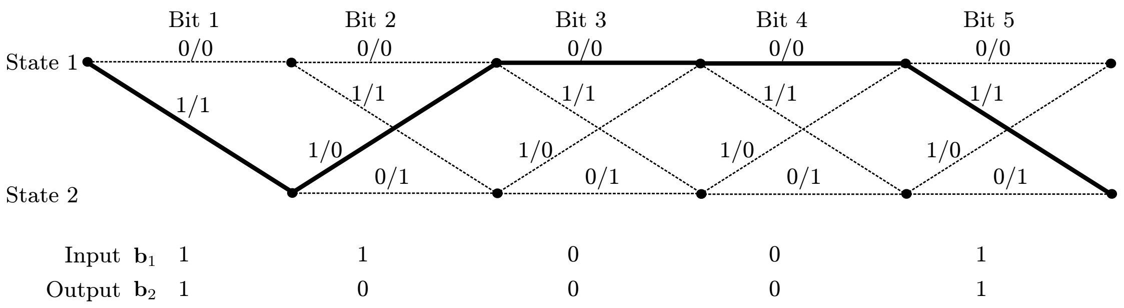

Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
State 1
State 2
0/0
1/1
1/0
0/1
Input $\mathbf{b}_1$ 1 1 0 0 1
Output $\mathbf{b}_2$ 1 0 0 0 1

Figure 2.3: An example convolutional code trellis having two possible states. Each transition T is labelled with the notation $a_1(T)/a_2(T)$. A particular transition T from the current state will be selected if the corresponding bit in the message frame $\mathbf{b}_1$ has the value $a_1(T)$, while $a_2(T)$ is the value that will be output for the corresponding bit in the encoded frame $\mathbf{b}_2$.

Note that the convolutional code's trellis of Figure 2.3 has $r = 2$ states, which corresponds to a shift register having one memory element. Furthermore, each transition between states is selected based on the value of one message bit, resulting in the generation of $n = 1$ encoded bit. This results in a coding rate for this convolutional encoder of $R = 1$, and an overall coding rate for the turbo code of Figure 2.2 of $R = 1/3$. However, the convolutional codes of generalized TCs may employ a shift register having any number of memory elements. The TC of the Long Term Evolution (LTE) standard in cellular telephony [18] employs a trellis having $r_{\text{LTE}} = 8$ states and $n = 1$ encoded bits, where the mapping of message and encoded bit values to each transition in the LTE TC trellis is specified by its *generator polynomials*. Furthermore, the LTE TC appends three additional termination bits to each message frame $\mathbf{b}_1$, in order to guarantee that the convolutional encoder always reaches the same particular state at the end of the encoding process.

2.1.2 Turbo coded transmitter

As shown in Figure 2.2, the turbo encoder comprises a parallel concatenation of two convolutional encoders, which are referred to as the upper and lower encoders. The upper encoder processes the frame of message bits $\mathbf{b}_1^{\mathrm{u}} = [b_{1,k}^{\mathrm{u}}]_{k=1}^{N}$ in order of increasing increasing k, while the lower encoder processes the same bits, but in a different order. This reordering is performed by the interleaver π of Figure 2.2, which outputs the interleaved message frame $\mathbf{b}_1^{\mathrm{l}} = [b_{1,k}^{\mathrm{l}}]_{k=1}^{N}$. The upper and lower convolutional encoders produce the N-bit encoded frames $\mathbf{b}_2^{\mathrm{u}} = [b_{2,k}^{\mathrm{u}}]_{k=1}^{N}$ and $\mathbf{b}_2^{\mathrm{l}} = [b_{2,k}^{\mathrm{l}}]_{k=1}^{N}$, respectively. These encoded frames provide $2N$ parity bits, which are multiplexed in the crossed block of Figure 2.2 with N systematic bits, which are provided by the N-bit message frame $\mathbf{b}_1^{\mathrm{u}}$. The resultant transmission frame $\mathbf{b}_3$ comprises $3N$ bits, corresponding to a coding rate of $R = N/(3N) = 1/3$.

Following turbo encoding, the transmitter of Figure 2.2 employs BPSK modulation, upsampling, pulse shaping, Radio Frequency (RF) mixing and power amplification. These are employed in order to transmit the frame $\mathbf{b}_3$ using the desired carrier frequency f_c at a desired transmission energy per bit $E_{\mathrm{b}}^{\mathrm{tx}}$. Here, $E_{\mathrm{b}}^{\mathrm{tx}}$ is related to the energy $E_{\mathrm{s}}^{\mathrm{tx}}$ dissipated per modulated symbol according to $E_{\mathrm{b}}^{\mathrm{tx}}[\mathrm{dBJ}] = E_{\mathrm{s}}^{\mathrm{tx}} - 10\log_{10}(\eta)$, where $\eta = R\log_2(M_{\mathrm{mod}})$, R is the coding rate and M_{mod} is the modulation order of the modulation scheme, with $M_{\mathrm{mod}} = 2$ in the case of BPSK. Note that the employment of $E_{\mathrm{b}}^{\mathrm{tx}}$ is typically preferable to $E_{\mathrm{s}}^{\mathrm{tx}}$, since this allows a fair comparison amongst schemes having different coding rates R and modulation orders M in terms of their transmission energy consumption.

2.1.3 Channel

The wireless channel of Figure 2.2 conveys the BPSK-modulated signal between the transmitter and receiver antennas, but imposes degradation. These antennas can be characterized by their gain (G_{tx} and G_{rx}) for the intended direction of propagation. In the scenario where there is a dominant line-of-sight (LOS) path between these antennas, the degradation may be modelled by the inverse-second-power free space path loss and AWGN. Here, the path loss is imposed by the attenuation of the BPSK-modulated signal as it propagates through free space. This depends on the distance between the transmit and receive antennas d (in m) and the carrier frequency f_c (in Hz) [30], according to

$$P_{\mathrm{l}}(d)[\mathrm{dB}] = 20\log_{10}(d) + 20\log_{10}(f_c) + 20\log_{10}\left(\frac{4\pi}{c}\right), \tag{2.1}$$

where $c = 2.998 \times 10^8$ m/s is the speed of light, resulting in the last term of (2.1) having a constant value of -147.55 dB. However, the free space path loss model may be optimistic, since often there are multiple paths between the transmitter and receiver but the LOS path might be absent. In order to account for this, the path loss equation can be generalized by parameterising the path loss exponent p [31,32], according to

$$P_l(d)[\text{dB}] = 10p\log_{10}(d) + 20\log_{10}(f) - 147.55. \tag{2.2}$$

Path loss exponents between $p = 2$ and $p = 4$ can be expected in the diverse environments encountered. The AWGN is imposed by the Brownian motion of electrons, resulting in thermal noise at the receiver, which has the power spectral density of $N_0[\text{dBJ}] = 10 \times \log(k \cdot T)$, where $k = 1.3806503 \times 10^{-23} JK^{-1}$ is the Boltzmann constant. For the case of the room temperature $T = 300K$, we obtain $N_0 = -203.8$ dBJ. Note that depending on the operating conditions, co-user interference is often more significant than the thermal noise. To model this, N_0 can instead be replaced with the noise power spectral density that is expected in the operating conditions of the wireless link [33].

Considering the above channel effects, we can therefore relate the energy per bit at the receiver E_b in terms of the energy dissipated at the transmitter E_b^tx and the channel conditions, according to

$$E_\text{b}[\text{dBJ}] = 10\log_{10}(E_\text{b}^\text{tx}) - A_\text{Amp} - P_l(d) + G_\text{tx} + G_\text{rx}, \tag{2.3}$$

where A_Amp is the power amplifier efficiency, and where all quantities are expressed in dB, except E_b^tx which is expressed in Joules. Note that if shadowing or fading is prevalent in the particular wireless environment considered, then (2.3) can be modified to model this by additionally subtracting corresponding fading margins [34]. In particular, narrowband uncorrelated Rayleigh fading [35] channels are explored in this thesis.

2.1.4 Turbo coded receiver

In the receiver of Figure 2.2, the BPSK-modulated signal provided by the receive antenna is passed to a Low Noise Amplifier (LNA). This is employed to boost the weak received signals, while introducing only a minimal amount of additional noise, which is quantified by its Receiver Noise Figure (RNF). The amplified signal is mixed down from the RF range to the baseband, where it is filtered to remove the out-of-band noise, down-sampled and provided to the BPSK demodulator.

The role of the BPSK demodulator is to extract information pertaining to the turbo-encoded bits from the received signal. However, the BPSK demodulator can never be certain of the correct value for each bit, owing to the unpredictable nature of the degradation imposed by the channel. Rather than making a binary *hard decision* of '1' or '0' for each bit, superior error correction performance can be obtained if the demodulator makes a *soft decision.* Here, a soft decision expresses not only *what* the most likely value of the bit is, but also *how likely* this value is. More specifically, the demodulator, which is also often referred to as a demapper, can express the soft information pertaining to a particular bit using a Logarithmic Likelihood Ratio (LLR), which represents the probabilities associated with the value of the bit b according to $\tilde{b} = \ln[\Pr(b = 1)/\Pr(b = 0)]$. Here, the sign of an LLR expresses whether a value of '1' or '0' is more likely for the corresponding bit, while the magnitude of the LLR is commensurate with how likely this value is. When employing BPSK modulation, it can be shown that each LLR is directly proportional to the corresponding sample provided by the down-sampler [36]. As shown in Figure 2.2, the BPSK demodulator generates the LLR sequences $\tilde{\mathbf{b}}_1^{\mathrm{u,s}}$, $\tilde{\mathbf{b}}_2^{\mathrm{u,a}}$ and $\tilde{\mathbf{b}}_2^{\mathrm{l,a}}$, which pertain to the bit sequences $\mathbf{b}_1^{\mathrm{u}}$, $\mathbf{b}_2^{\mathrm{u}}$ and $\mathbf{b}_2^{\mathrm{l}}$ respectively. Furthermore, an interleaver π is employed for converting $\tilde{\mathbf{b}}_1^{\mathrm{u,s}}$ into the LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{l,s}}$, which pertains to the bit sequence $\mathbf{b}_1^{\mathrm{l}}$. These LLR sequences are then provided to the turbo decoder, which is invoked for mitigating the corresponding uncertainty and for eliminating transmission errors. As shown in Figure 2.2, the turbo decoder comprises two Log-BCJR decoders, which correspond to the two convolutional encoders of the turbo encoder.

The turbo decoder is operated in an iterative manner, with the switch labelled 'S1' in Figure 2.2 being left open during the first decoding iteration. This enters the LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{u,s}}$ provided by the BPSK demodulator directly into the upper Log-BCJR decoder $\tilde{\mathbf{b}}_1^{\mathrm{u,a}}$. As shown in Figure 2.2, the upper Log-BCJR decoder's other input $\tilde{\mathbf{b}}_2^{\mathrm{u,a}}$ is supplied by the BPSK demodulator. The upper Log-BCJR decoder combines the old (or "*a priori*") information provided by its two input LLR sequences, in order to extract new (or "extrinsic") information for the output LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{u,e}}$. Since this LLR sequence pertains to the uncoded bit sequence $\mathbf{b}_1^{\mathrm{u}}$, the interleaver π may be used for converting it into information pertaining to the bit sequence $\mathbf{b}_1^{\mathrm{l}}$. Following this, the resultant interleaved LLR sequence may be added on a bit-by-bit basis to the values in the LLR sequence $\tilde{\mathbf{b}}_{1,s}^{\mathrm{l}}$ provided by the BPSK demodulator, which also pertains to $\mathbf{b}_1^{\mathrm{l}}$. The resultant LLR sequence is then forwarded to the lower Log-BCJR decoder's input $\tilde{\mathbf{b}}_1^{\mathrm{l,a}}$, as shown in Figure 2.2. Meanwhile, the lower Log-BCJR decoder's other input $\tilde{\mathbf{b}}_2^{\mathrm{l,a}}$ is supplied by the BPSK demodulator. In turn, the lower Log-BCJR decoder combines these *a priori* LLR sequences, in order to obtain the extrinsic LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{l,e}}$, completing the first decoding iteration.

In the second and in all subsequent decoding iterations, the switch labelled 'S1' in Figure 2.2 is closed. This allows the extrinsic LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{l,e}}$ to be deinterleaved π^{-1} and added on a bit-by-bit basis to the values in the LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{u,s}}$, in order to generate an improved *a priori* LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{u,a}}$ for the upper Log-BCJR decoder. This motivates the repeated operation of the upper Log-BCJR decoder, in order to produce an improved extrinsic LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{u,e}}$. In turn, this may be interleaved and added on a bit-by-bit basis to the values in the LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{l,s}}$, in order to generate an improved *a priori* LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{l,a}}$ for the lower Log-BCJR decoder. Likewise, the operation of the lower Log-BCJR decoder may be repeated for obtaining an improved extrinsic LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{l,e}}$. This process may be repeated during the third iteration and during all further iterations, in order to gradually improve the quality of the iteratively exchanged LLR sequences. However, as will be show in Section 2.1.7, each additional iteration yields a diminishing return, until convergence is eventually achieved, whereupon additional iterations provide no further improvement. Once a sufficient number I of iterations has been performed, a final output may be obtained by adding the LLR sequences $\tilde{\mathbf{b}}_1^{\mathrm{u,a}}$ and $\tilde{\mathbf{b}}_1^{\mathrm{u,e}}$ on a bit-by-bit basis. The resultant LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{u,p}}$ contains all (or "*a posteriori*") information pertaining to the turbo encoder's input bit sequence $\mathbf{b}_1^{\mathrm{u}}$. Finally, these soft-valued LLRs may be converted into hard-valued bits by considering the sign of each LLR, where a positive value corresponds to a '1' and a negative value corresponds to a '0'.

2.1.5 Log-BCJR decoder

In this section, this thesis provides an overview of the Log-BCJR algorithm [37], which is employed both by the upper and lower Log-BCJR decoders of Figure 2.2. Note that the Log-BCJR algorithm is a reduced-complexity version of the BCJR algorithm, as will be discussed in greater detail in Section 2.1.6, together with a discussion of other variants of the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm. Here, as an example, the trellis of Figure 2.3 is imposed for combining the example *a priori* LLR sequences $\tilde{\mathbf{b}}_1^{\mathrm{a}} = [-5, 4, 1, 6, -2]$ and $\tilde{\mathbf{b}}_2^{\mathrm{a}} = [3, 5, -4, -2, -1]$, in order to obtain the extrinsic LLR sequence $\tilde{\mathbf{b}}_1^{\mathrm{e}}$. Note that these example LLRs have been rounded to the nearest integer, for the sake of simplicity. The Log-BCJR algorithm comprises four intermediate steps, in which four sets of metrics are calculated, namely the $\tilde{\gamma}(T)$, $\tilde{\alpha}(m)$, $\tilde{\beta}(m)$ and $\tilde{\delta}(T)$ values, where T refers to a particular transition in the trellis and m refers to a particular state, as detailed in the following discussion. This thesis will show that the calculations of each step can be approximately decomposed into simple Add-Compare-Select (ACS) operations. Further detailed discussions are available in [38, 39].

In the first step of the Log-BCJR algorithm, a $\tilde{\gamma}(T)$ value is calculated for each transition in the trellis of Figure 2.3. This $\tilde{\gamma}(T)$ value represents the *a priori* probability that the transition T was selected during the convolutional encoding process. The $\tilde{\gamma}(T)$ value for a particular transition T in the trellis of Figure 2.3 is calculated according to

$$\tilde{\gamma}(T) = a_1(T) \cdot \tilde{b}^{\mathrm{a}}_{1,i(T)} + a_2(T) \cdot \tilde{b}^{\mathrm{a}}_{2,i(T)}, \tag{2.4}$$

where $a_1(T)$ and $a_2(T)$ are described in Figure 2.3. Here, $i(T)$ is the index of the bits that are represented by the transition T, while $\tilde{b}^{\mathrm{a}}_{1,i(T)}$ is the LLR having that specific index $i(T)$ in the sequence $\tilde{\mathbf{b}}_1^{\mathrm{a}}$. Likewise, $\tilde{b}^{\mathrm{a}}_{2,i(T)}$ is the corresponding LLR in the sequence $\tilde{\mathbf{b}}_2^{\mathrm{a}}$. In Figure 2.4 each transition is labelled with the particular $\tilde{\gamma}(T)$ value that results for our example. Note that relatively high $\tilde{\gamma}(T)$ values result for transitions where the *a priori* LLRs match with that transition's $a_1(T)$ and $a_2(T)$ combination. Since $a_1(T)$ and $a_2(T)$ have binary values, each $\tilde{\gamma}(T)$ value is given by 0, $\tilde{b}^{\mathrm{a}}_{1,i(T)}$, $\tilde{b}^{\mathrm{a}}_{2,i(T)}$ or $\tilde{b}^{\mathrm{a}}_{1,i(T)} + \tilde{b}^{\mathrm{a}}_{2,i(T)}$. Therefore, the entire set of $\tilde{\gamma}(T)$ values can be calculated using only addition and selection operations.

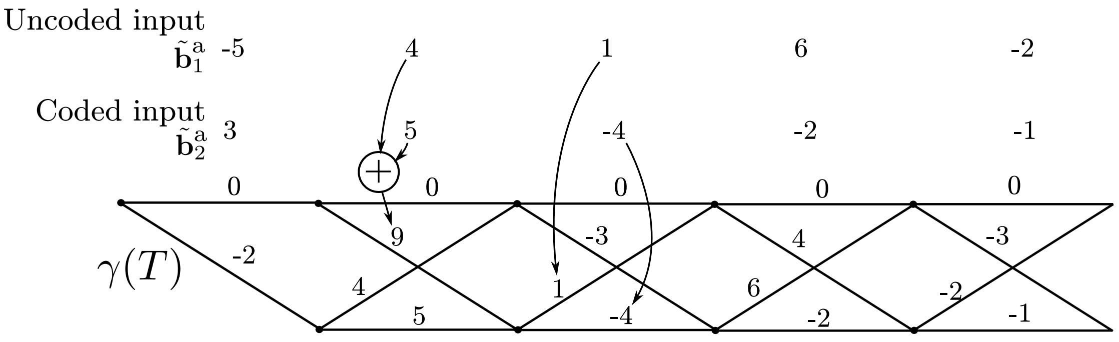

Uncoded input
$\tilde{\mathbf{b}}^{\mathrm{a}}_1$ -5 4 1 6 -2
Coded input
$\tilde{\mathbf{b}}^{\mathrm{a}}_2$ 3 5 -4 -2 -1
0 0 0 0 0
-2 9 -3 4 -3
4 1 6 -2
5 -4 -2 -1
$\gamma(T)$

Figure 2.4: Calculating the $\tilde{\gamma}(T)$ values for some example *a priori* LLRs.

The second step of the Log-BCJR algorithm is to calculate an $\tilde{\alpha}(m)$ value for each state m in the trellis. These $\tilde{\alpha}(m)$ values represent the probability that a particular state was entered into during the encoding process. This is obtained by considering the probabilities of the previous states having been entered into during encoding, as well as the probabilities that the transitions between these pairs of states have been taken. Owing to these dependencies between the probabilities associated with consecutive states, a forward recursion is required in order to calculate the $\tilde{\alpha}(m)$ values for the states of the trellis in a specific order, evolving from left to right. The calculation for an $\tilde{\alpha}(m)$ for a particular state m is given by

$$\tilde{\alpha}(m) = \max_{T\in to[m]}{}^{*} \left[\tilde{\gamma}(T) + \tilde{\alpha}(fr[T])\right], \tag{2.5}$$

where $to[m]$ returns the set of all transitions merging into the state m, while $fr[T]$ returns the particular state that the transition T emerges from. The operation $\max^*$ for two inputs A and B is defined as $\max^*(A, B) = \max(A, B) + \ln(1 + e^{-|A-B|})$. Since this operation is associative, it can be readily extended to more inputs. In the example of Figure 2.5, each state in the trellis is labeled with its $\tilde{\alpha}(m)$ value, where the $\max^*$ operator has been approximated using the max operation for simplicity. As shown in Figure 2.5, the forward recursion is initialized by setting the $\tilde{\alpha}(m)$ value of the state at the far left of the trellis to zero. Note that the $\tilde{\alpha}(m)$ values are calculated using only addition and $\max^*$ operations, which can be further decomposed into only ACS operations, as will shall show in Section 2.1.6.

In the third step of the Log-BCJR algorithm, a $\tilde{\beta}(m)$ value is calculated for each state in the trellis, using a similar process to that of the $\tilde{\alpha}(m)$ values. While the $\tilde{\alpha}(m)$ values depend on the previous $\tilde{\alpha}(m)$ values in the trellis, the $\tilde{\beta}(m)$ value of a particular state

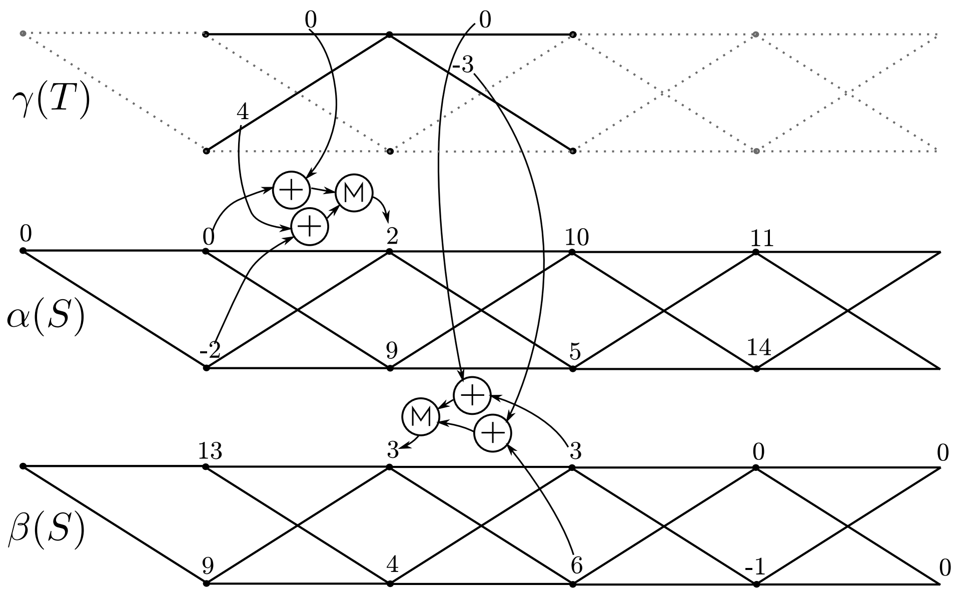


Figure 2.5: Calculating the $\tilde{\alpha}(m)$ and $\tilde{\beta}(m)$ values. The previous $\tilde{\gamma}(T)$ trellis is shown for reference. The circled 'M' represents the max* operation, which has been approximated using the max operation in the presented calculations, to maintain integer values for simplicity.

depends on those of the next states in the trellis. Therefore the $\tilde{\beta}(m)$ values must be calculated in order, using a backward recursion order, evolving from the right end of the trellis to the left end. This is achieved according to

$$\tilde{\beta}(m) = \max_{T \in fr[m]}{}^{*} \left[\tilde{\gamma}(T) + \tilde{\beta}(to[T])\right], \tag{2.6}$$

where $fr[m]$ returns the set of all transitions that emerge from the state m, while $to[T]$ returns the particular state that the transition T merges into. Once again, the $\tilde{\beta}(m)$ values for our example are shown on the states of Figure 2.5, where the max* operator has been approximated using the max operation for simplicity. As shown in Figure 2.5, the backward recursion is initialized by setting the $\tilde{\beta}(m)$ values of the states at the far right of the trellis to zero. Like the $\tilde{\alpha}(m)$ values, the $\tilde{\beta}(m)$ values can be calculated using only ACS calculations.

The fourth set of metrics required for the Log-BCJR algorithm are the $\tilde{\delta}(T)$ values, which combine the results from previous metrics in order to represent the *a posteriori* probabilities that the transitions were followed in the encoder. The $\tilde{\delta}(T)$ value of a particular transition T is calculated by adding its $\tilde{\gamma}(T)$ value to the $\tilde{\alpha}(m)$ value of the state it emerges from and the $\tilde{\beta}(m)$ value of the state it merges into, according to

$$\tilde{\delta}(T) = \tilde{\alpha}(fr[T]) + \tilde{\gamma}(T) + \tilde{\beta}(to[T]). \tag{2.7}$$

The $\tilde{\delta}(T)$ calculations detailed for our example can be seen in Figure 2.6. Since the $\tilde{\delta}(T)$ values are calculated using only additions, they can be decomposed into ACS operations.

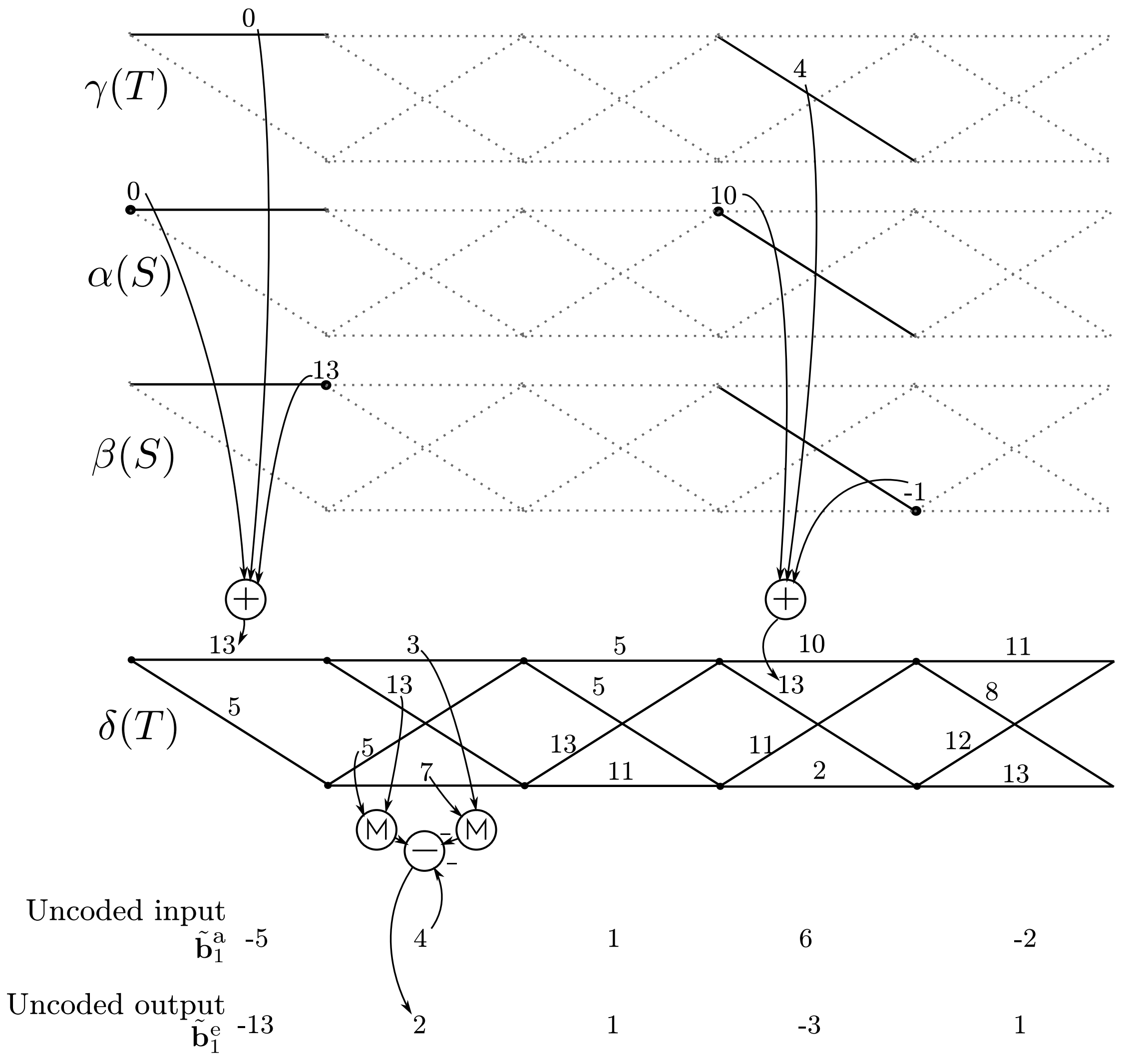


Figure 2.6: Calculating the $\tilde{\delta}(T)$ values and the extrinsic LLRs. The circled 'M' represents the max* operation, which has been approximated using the max operation in the presented calculations, to maintain integer values for simplicity.

Finally, the Log-BCJR algorithm can combine the $\tilde{\delta}(T)$ values in order to calculate the output extrinsic LLRs. This is achieved according to

$$\tilde{b}^{\mathrm{e}}_{1,i} = \max^*_{T\left|\substack{a_1(T)=1\\ i(T)=i}\right.} [\tilde{\delta}(T)] - \max^*_{T\left|\substack{a_1(T)=0\\ i(T)=i}\right.} [\tilde{\delta}(T)] - \tilde{b}^{\mathrm{a}}_{1,i}, \tag{2.8}$$

where $T\Big|_{i(T)=i}^{a_1(T)=0}$ is the set of all transitions for which the represented uncoded bit value $a_1(T)$ is zero and the index $i(T)$ of that uncoded bit is i. As shown in the example of Figure 2.6, this corresponds to the grouping of the $\tilde{\delta}(T)$ values into two sets, which are then combined using max* operations. Following this, the *a priori* LLR $\tilde{b}^{\mathrm{a}}_{1,i}$ is subtracted from the difference between these two max* calculations. Note that the extrinsic LLRs are calculated using only subtraction and max* operations, which can be further decomposed into only ACS operations, as will be shown in Section 2.1.6. This completes the Log-BCJR decoding process.

2.1.6 Algorithmic modifications to the Log-BCJR decoder

The Log-BCJR algorithm is universally preferred for implementation over the BCJR algorithm owing to its reduced computational complexity. More specially, the BCJR algorithm operates in the normal domain, requiring addition and multiplication operations for calculating the bit probabilities. Since these probabilities have a high dynamic range, a large number of bits are required for their digital representation. By converting the equations of the BCJR algorithm into the logarithmic domain, the Log-BCJR algorithm replaces multiplications with additions, and replaces additions with the max* operation. These operations have a lower computational complexity, and representing the probabilities in the logarithmic domain requires fewer bits.

As shown in Section 2.1.5, the max* operation of the Log-BCJR algorithm is defined by $\max^*(A, B) = \max(A, B) + f(|A-B|)$, where the correction term is given by $f(|A-B|) = \ln(1+e^{-|A-B|})$. Since the logarithmic and exponential functions of $f(|A-B|)$ are costly to implement in hardware, they are often approximated in practical applications of TCs. In the Maximum Log-BCJR (Max-Log-BCJR) approximation [40] of the Log-BCJR algorithm, $\max^*(A, B)$ is approximated using $\max(A, B)$. As shown in Figure 2.7, the value of $f(|A - B|)$ is always in the range $[0, 0.69]$, which is typically small compared to $\max(A, B)$, justifying this approximation. The Max-Log-BCJR approximation imposes a low computational complexity, but its error correction capability is lower than that of the original Log-BCJR algorithm [37]. This motivates the conception of a Look-Up-Table based Log-BCJR (LUT-Log-BCJR) algorithm [41], which uses a Look-Up Table (LUT) for approximating $f(|A - B|)$. As shown in Figure 2.7, the range of $|A - B|$ values for which $f(|A - B|)$ has a significant value is limited, meaning the LUT size can be small. Figure 2.7 shows how as few as four values given by $\{0, 0.25, 0.5, 0.75\}$ can be used for approximating $f(|A - B|)$, hence offering an error correction capability for the LUT-Log-BCJR which approaches that of the Log-BCJR algorithm [41], as shown in Figure 2.11.

Both of the Max-Log-BCJR and the LUT-Log-BCJR algorithms can be implemented using only ACS operations. Firstly, the $\max(A, B)$ operation is performed by comparing A and B, and selecting the largest value. Based on the knowledge of $\max(A, B)$, the subtraction $|A-B|$ of the LUT-Log-BCJR can be carried out so that a positive number is returned. By comparing this result to the boundary points of the LUT, the approximate value for $f(|A - B|)$ can be selected, and then added to the value of $\max(A, B)$.

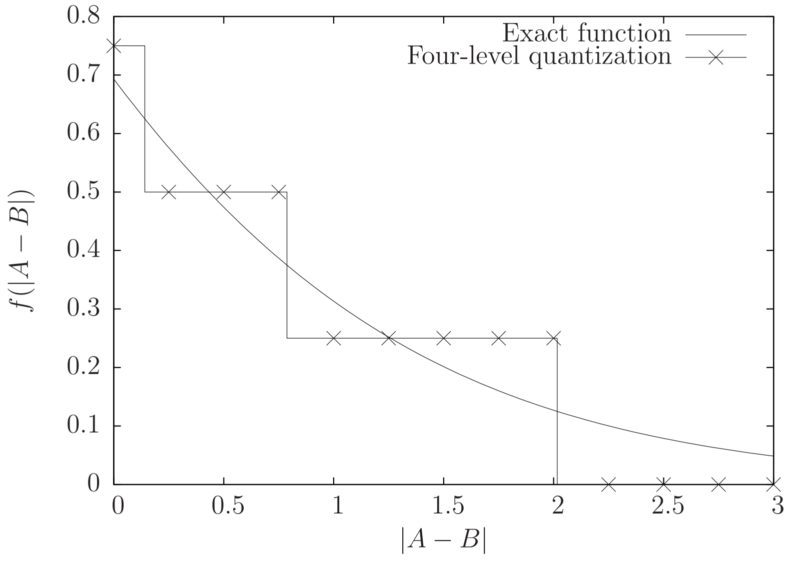


Figure 2.7: Plot of $f(|A-B|) = \ln(1+e^{-|A-B|})$, showing the quantization steps and points when a four-level quantization is employed.

As shown in Section 2.1.5, the $\tilde{\alpha}(m)$ and $\tilde{\beta}(m)$ calculations require forward and backward recursions respectively due to their data dependencies between consecutive states. Owing to this, the Log-BCJR and its variants are not naturally suited to parallel processing. Furthermore, a large amount of memory is required, since the $\tilde{\alpha}(m)$ and $\tilde{\beta}(m)$ values are calculated in different directions along the trellis. More specifically, in order to generate the first output extrinsic LLR $\tilde{b}^{\mathrm{e}}_{1,1}$, it is necessary to have first calculated the $\tilde{\beta}(m)$ values for every state in the trellis and then to store them for the calculation of the subsequent output extrinsic LLRs.

An appealing technique for overcoming the data dependency issue is to decompose the trellis into N/ω number of smaller windows [10], each having the length ω. The Log-BCJR algorithm (or one of its approximations) can be applied to each window independently, significantly reducing the memory required for storing metrics. However, with this approach, it is necessary to initialize the $\tilde{\alpha}(m)$ values of the states at the left end of each window, as well as the $\tilde{\beta}(m)$ values of the states at the right end. If the windows are processed sequentially in a left to right ordering, the boundary $\tilde{\alpha}(m)$ values can be passed from the right end of each window to the left end of the subsequent window. However, this approach cannot supply boundary $\tilde{\beta}(m)$ values for the right end of each window, requiring a pre-backward recursion to generate these boundary conditions [42]. This technique generates boundary conditions by starting to calculate the $\tilde{\beta}(m)$ values ahead of the window, then carrying out a backwards recursion towards the edge of the window. The first $\tilde{\beta}(m)$ values used by the pre-backward stage are initialized to zero, then the pre-backwards length w_p is chosen for ensuring that the beta values generated at the boundary of the window converge to those values in the non-windowed Log-BCJR algorithm. Further detailed reading on the pre-backward technique is available in [43].

Other windowing techniques include the Previous Iteration Value Initialization (PIVI) technique of [42,44], which is also known as State-Metric Propagation (SMP) [45]. This avoids the extra computation associated with the pre-backwards step by initializing the windows during the current turbo decoding iteration using the boundary conditions 'inherited' from the previous iteration.

2.1.7 Turbo code performance

When analysing the performance of error correcting codes, typically the Bit Error Ratio (BER) of the code is plotted against the *SNR per bit* E_b/N_0, where E_b is the energy received per message bit. A TC's BER plot can be used for determining the minimum E_b/N_0 required for reliable communication.

Figure 2.8 provides a BER plot for a $R = 1/3$ LTE turbo code, which uses the schematic shown in Figure 2.2. Figure 2.8 shows that the error correction performance improves with successive iterations of the decoder, until about 8 iterations have been completed. Beyond this convergence point however, there are diminishing returns, resulting in very little further improvement.

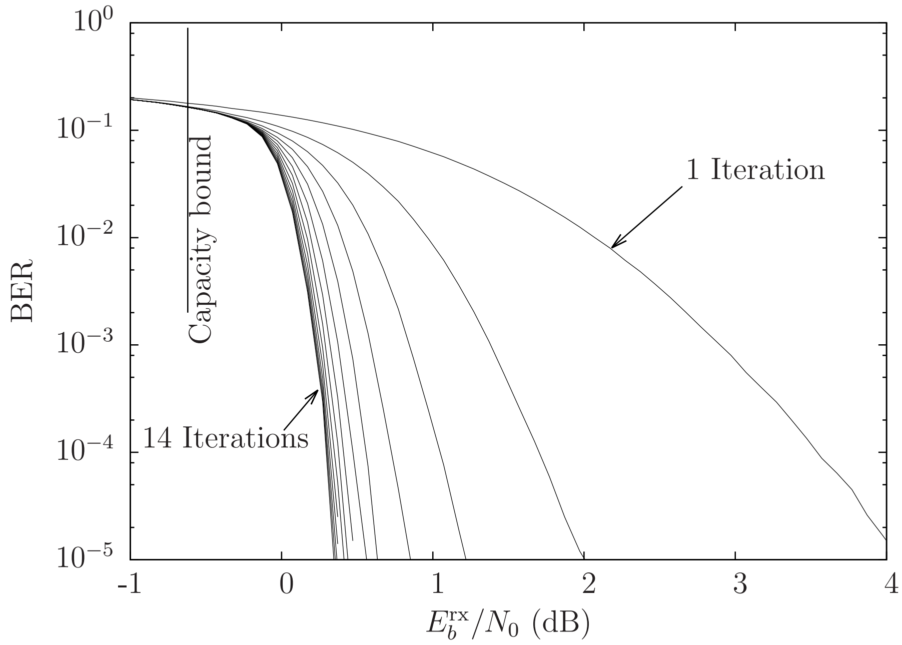


Figure 2.8: BER performance of a 6144-bit $R = 1/3$ LTE turbo code employing varying number of iterations $I = 1, 2, 3, ..., 14$, when using BPSK modulation for communication over an AWGN channel.

A specific feature of turbo codes is that they perform better with the aid of longer interleavers. Figure 2.9 shows the attainable BER performance for the message lengths of $N = 40$, 440 and 6144 bits, as well as for the uncoded BPSK case. While all of the turbo coded schemes offer an improved BER for E_b/N_0 values above 0 dB, the longer

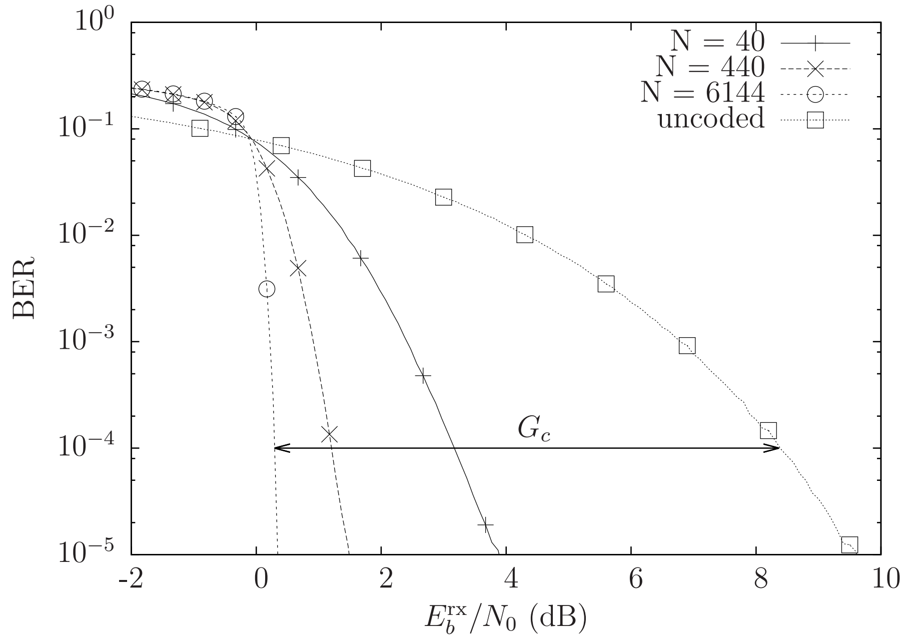


Figure 2.9: BER performance of $R = 1/3$ LTE turbo codes employing 10 iterations and having different frame lengths when using BPSK modulation for communication over an AWGN channel. The coding gain G_c is achieved by the turbo code relative to the uncoded case, when achieving a target BER of 10^{-4}.

frame lengths have a much steeper cliff than shorter ones. Owing to this, shorter frame lengths N correspond to higher E_b/N_0 requirements for achieving reliable communication. Figure 2.9 shows that the LTE TC provides a coding gain G_c of around 8 dB over the uncoded scheme, which equates to a corresponding transmission energy saving at the transmitter.

Using (2.3), the transmission energy per message bit E_b^{tx} required to achieve a particular target BER can be expressed as

$$E_b^{tx}[\text{dBJ}] = S_t + N_0 + \text{RNF} + P_l + A - G_{tx} - G_{rx}, \qquad (2.9)$$

where all quantities are expressed in dB, and S_t is the minimum SNR per bit E_b/N_0 that is required to achieve the target BER.

2.2 Turbo decoder architectures

Referring to Figure 1.1, this section will discuss the architecture design for channel codes, and some techniques which consider both the architecture and algorithm are also discussed. This section commences by reviewing the existing approaches for turbo decoder implementations. Then, our focus will be narrowed to focus to three major areas for formulating design considerations. Firstly, Section 2.2.1 considers the most significant challenges in energy-efficient and area-efficient datapath design, as well as in architectural solutions to these. Secondly, Section 2.2.2 considers the issues of algorithm control, where the scheduling of the decoder by the controller will be investigated, under the consideration of beneficial modifications to the algorithm that achieve a lower energy consumption and reduced area, at a minimal loss to error correction performance. This performance loss can then be considered during the holistic design stage, as shown in Figure 8.2. Finally the various aspects of energy-efficient memory usage is discussed in Section 2.2.3.

Table 2.1: Summary of $R = 1/3$ LTE turbo decoding architectures for a variety of applications

Work	Algorithm	E_b/N_0 [dB] at BER = 10^{-3}	Technology [nm]	Voltage [V]	Area [mm^2]	Area at 90nm [mm^2]	Iterations	Throughput [Mb/s]	Energy consumption [nJ/bit/iter.]	Energy consumption @ 90nm, 1 V [nJ/bit/iter.]
Bickerstaff M. et al. 2002 [46]	LUT-Log	0.32	180	1.8	9	2.25	10	2	14.6	2.3
Bickerstaff M. et al. 2003 [47]	LUT-Log	0.37	180	1.8	14.5	3.63	8	24	11.1	2.0
Li F. et al. 2008 [48]	LUT-Log	-	180	1.8	8.2	2.05	6.5	4.17	12.7	1.59
Benkeser C. et al. 2009 [25]	Max-Log	0.46	130	1.2	1.2	0.58	5.5	20.2	0.54	0.26
May M. et al. 2010 [49]	Max-Log	-	65	1.1	2.1	4	6	150	0.31	0.35
Sun Y. et al. 2010 [50]	-	-	65	0.9	8.3	15.9	6	1280	0.11	0.19
Studer C. et al. 2011 [12]	Max-Log	0.63	120	1.2	3.57	2	5.5	390.6	0.37	0.19
Li L. et al. 2011 [51]	LUT-Log	0.32	90	1.0	0.35	0.35	5	1.03	0.4	0.4
Studer C. et al. 2012 [52]	Max-Log	-	180	1.8	0.45	0.11		542	0.84	0.13
Ilnseher T. et al. 2012 [13]	Max-Log	-	65	1.1	7.7	14.8	6	2150	-	-
Belfanti S. et al. 2013 [53]; Roth C. et al. 2014 [45]	Max-Log	0.66	65	1.2	2.49	4.8	5.5	1013	0.17	0.16

Table 2.1 shows a range of Application-Specific Integrated Circuit (ASIC) turbo decoder architectures disseminated in the literature, which have been designed for meeting a variety of design goals. Some architectures, such as [51] and [47] employ the LUT-Log-BCJR of [41], which provides a superior error correction performance and a reduced transmission energy compared to the faster, less complex Max-Log-BCJR [40]

approximation. These decoders which use the LUT-Log-BCJR are low throughput turbo decoders, which also tend to have a reduced chip area, which results in a reduced static energy consumption and a reduced cost, which is often a concern in these applications. This is in contrast to conventional turbo decoder architectures [12, 50, 54], which are typically designed for bandwidth-constrained applications, such as cellular telephony, Wireless Local Area Network (WLAN) and broadcast systems. More specifically, these architectures are designed to have a high processing throughput, in order to match the high transmission throughputs that are sought in these applications. As a trade-off, these applications use the Max-Log-BCJR, which allows for a simpler approximation of the $\max^*$ calculation to support higher throughputs, but comes at the expense of both a degraded BER performance and an increased transmission energy requirement. Section 2.2.2 below discusses this tradeoff, as well as methods aimed at mitigating their performance loss.

This section will concentrate on conventional decoders, however alternate approaches have also been proposed for implementing the BCJR algorithm, which will be briefly discussed here. Firstly, stochastic decoders [55] represent each LLR as a series of bits, where the value of the sequence is represented by how many '1's or '0's there are in the sequence. In contrast to the conventional fixed-point binary representation, each bit in a stochastic sequence has the same significance. During decoding, each bit in these LLR sequences is processed sequentially by the stochastic decoder. The decoder only processes one bit of each LLR in each clock cycle, which results in a significant reduction of the number of gates required in the decoder. However, since long LLR sequences are required for a high error correction performance, stochastic decoders typically require many more clock cycles compared to a conventional decoder, hence resulting in lower throughputs.

Another alternative architecture is constituted by the family of analogue turbo decoders [56]. In these architectures, soft information is represented with the aid of analogue currents, while the various operations of the decoder are performed using analogue arithmetic circuits. Analogue decoders have shown a promising decoding energy consumption $E_\mathrm{b}^\mathrm{dec}$, outperforming the comparable digital turbo decoders. However, they also impose additional challenges which have limited their potential. For example, the difficulties in matching analogue circuits on a large scale leads to a potential performance degradation [57]. Furthermore, accurately simulating the BER performance of the circuit before its fabrication is not feasible or accurate. In [58] an analogue architecture, which supports long frames is described, although this is associated with other challenges. In particular, a sampling circuit is required at each input of the decoder, which holds the analogue value constant during decoding. However, these analogue values cannot be readily maintained for extended periods of time, hence affecting the achievable error correction performance.

In the following subsections, three salient aspects of conventional digital decoders are considered, namely the design issues of the data path, of the controller and of the memory.

2.2.1 Datapath considerations

Some of the designs listed in Table 2.1 rely on architectures that were designed for meeting the requirements of the latest telephony standards, resulting in optimizations for very high throughputs. These conventional architectures typically employ dedicated modules for each of the different steps in the Log-BCJR decoding algorithm. More specifically, they use separate hardware for calculating each of the $\tilde{\alpha}$, $\tilde{\beta}$, $\tilde{\delta}$ values and the extrinsic LLRs. However, a naive implementation can result in a long critical path, which precludes having a high processing energy efficiency, hardware area efficiency and throughput. More specifically, since a long critical path results in a low clock frequency, this results in a reduced throughput. By employing techniques to reduce the critical path and increase the throughput, the area efficiency may also be improved, providing the clock speed increase is greater than any added circuitry to reduce the critical path.

On this basis, a pair of techniques will now be discussed, which can be employed for mitigating the energy inefficiencies inherent in designs having a long critical path.

The first method this thesis will discuss is pipelining, which is employed extensively within the architectures of [13, 50, 52] and others. Pipelining reduces the critical path between two registers by adding additional registers to the middle of this path. This has the result of shortening the paths so that a higher clock frequency can be employed, but also adds latency to the circuit, since the number of clock cycles required before a result is available is increased for every pipeline stage that is added. This can therefore result in a slow down of a circuit's operation, if one part has to wait for a pipelined calculation to become available.

Figure 2.10 shows an example of pipelining in the turbo decoder of [52], which uses a similar decoder core to that proposed by the authors of [12, 25]. High-throughput turbo decoders, such as those proposed by [12, 13, 53], typically employ a multitude of these cores in parallel. The architecture of Figure 2.10 employs separate hardware units for calculating the $\tilde{\alpha}$ (forward state-metrics) and $\tilde{\beta}$ (reverse state-metrics), each having dedicated hardware for generating the $\tilde{\gamma}$ values. Since this architecture utilizes windowing, a separate dummy state-metric-recursion unit is used for generating the boundary conditions of the windows, as described in Section 2.1.6. This parallelisation within each decoder core facilitates higher throughputs than the alternative approaches. To perform the pipelining, registers are placed between the branch metric computation units that are used for calculating the $\tilde{\gamma}$ values, as well as between the ACS Units that are used for calculating the $\tilde{\alpha}$ or $\tilde{\beta}$ values. Note that due to their recursive nature, no pipelining can take place within the ACS Units. This is because the values for one bit depend on that of an adjacent bit, which is calculated in the preceding clock cycle. Adding pipelining to the ACS Unit then increases the number of cycles it takes for a new value to be calculated, hence slowing down the operation of the decoder, rather than speeding it up.

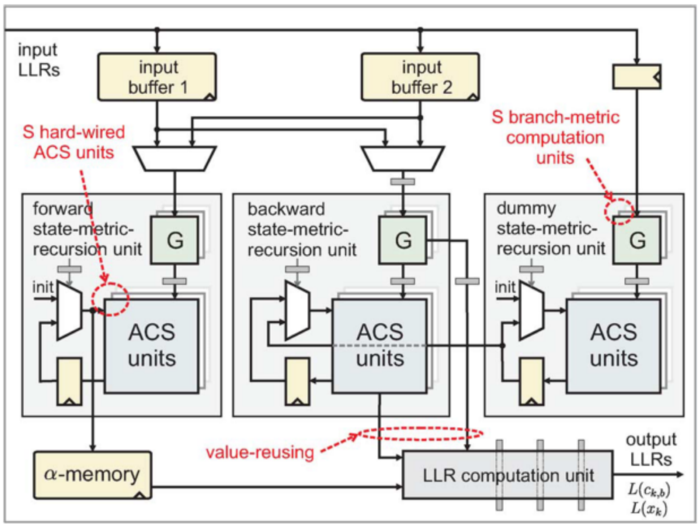


Figure 2.10: Pipelining in a high throughput Max-Log-BCJR decoder. From ©IEEE [52]. The pipelining registers are shown by the grey rectangles.

With careful pipelining, the critical paths in a design can be kept low and the path length can be kept more similar, therefore mitigating the previously mentioned impediments. However, as mentioned above, pipelining cannot be used in the recursive parts of the BCJR algorithm and the additional chip area as well as the Energy Consumption (EC) associated with the pipeline registers must also be considered.

2.2.2 Algorithm control

In this section, the control of the architecture is considered, where the controller instructs both the datapath and the memory to carry out a particular sequence of operations, in order to implement the algorithm.

In the Log-BCJR algorithm, the basic operation that imposes the highest computational overhead is the max* operation [41]. This is of particular concern in high-throughput decoders, where the max* calculation is used within the forward- and backward-recursive loops, preventing its pipelining for speeding up the decoder, as described in Section 2.2.1.

It is therefore desirable to favour the Max-Log BCJR over the LUT-Log-BCJR in applications, requiring a higher throughput. However, the naive employment of the Max-Log-BCJR results in a performance loss, when compared to the LUT-Log-BCJR. This

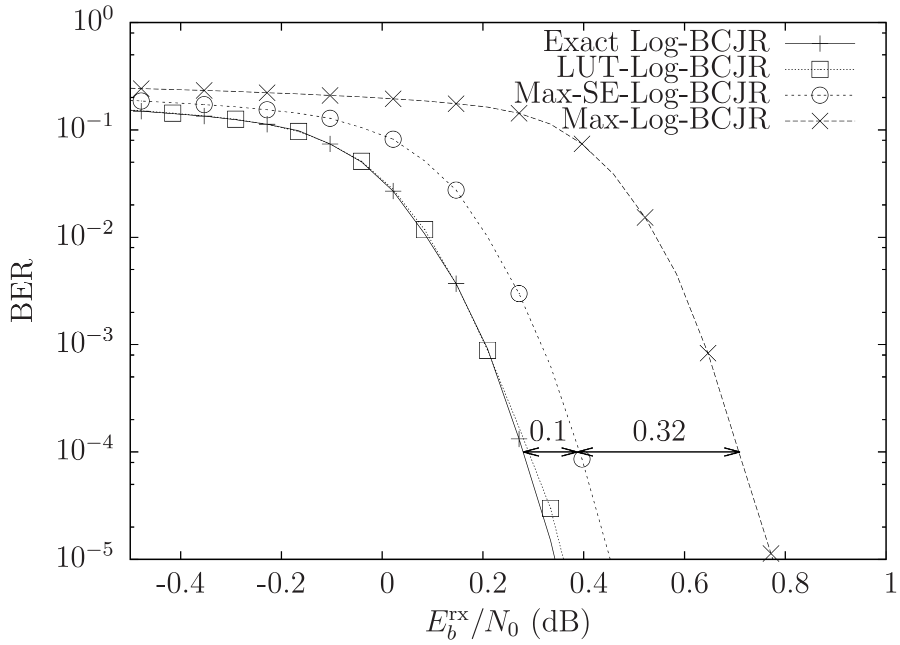


Figure 2.11: Error correction performance of 6144-bit turbo decoders employing extrinsic scaling (Max-SE-Log-BCJR), the Max-Log-BCJR and the LUT-Log-BCJR, in relation to that offered by the exact Log-BCJR. A LUT comprising 8 entries was used for the LUT-Log-BCJR, and a scaling factor of 0.7 was employed for the Max-SE-Log-BCJR.

motivates the employment of a technique known as extrinsic LLR scaling, which is capable of mitigating some of this performance loss [24, 25]. Figure 2.11 compares the error correction performance of the Log-BCJR, LUT-Log-BCJR, Max-Log-BCJR and Maximum with Scaled Extrinsic Log-BCJR (Max-SE-Log-BCJR) decoders. It can be seen that the extrinsic scaling technique improves the performance, which will be within a small margin of 0.1 dB of that offered by the Log-BCJR algorithm. This is a typical margin that may be observed for other turbo code parameterisations designed for communicating over AWGN and Rayleigh fading [59] channels.

The Max-SE-Log-BCJR decoder relies on multiplying the extrinsic LLR output of the decoder blocks in the receiver by a constant value of less than 1. This represents a reduction of confidence in the extrinsic LLRs, which is due to the non-optimal implementation of the $\max^*$ calculation. The author of [60] discuss the optimal selection of this constant, which is found to be between 0.6 and 0.8, depending on the SNR at the receiver. However, practical implementations tend to use a fixed scaling value [59]. A typical choice for the extrinsic scaling factor is one that leads to a simple hardware implementation using just adders. For example, a scaling factor of 0.75 can be achieved using fixed point arithmetic by simply adding the extrinsic output right-shifted once, to the extrinsic output right-shifted twice.

Extrinsic LLR scaling is also used in the Max-Log-BCJR architecture of [25], resulting in a 45% reduction in area and a 50% improvement in throughput, when compared to a similar architecture, which uses the LUT-Log-BCJR algorithm instead. The reduction in the number of logic gates required for the max* calculation also results in a reduced EC.

As described above, the use of extrinsic LLR scaling in conjunction with the Max-Log-BCJR results in an error correction performance loss relative to the LUT-Log-BCJR decoder. This equates to more *transmit* energy being required, but offers the advantage of requiring lower *decoding* energy. Note that the holistic design method discussed in Section 2.3 will address these conflicting design choices. This conflict demonstrates the importance of considering both the architecture and the algorithm jointly, since a holistic design approach facilitates striking the right balance between the algorithm and the architecture, resulting in the lowest overall EC and the best overall performance for the system.

Another beneficial technique for the implementation of turbo decoders is the Radix-4 transformation of [13, 47], which combines two trellis stages into a single one. Owing to this, the decoder considers twice the number of *a priori* LLRs at once and the number of transitions emerging from each state of the Radix-4 trellis is squared. However, this technique halves the number of state metrics that have to be calculated and stored, since it halves the number of stages in the trellis. In the most common case, where only two transitions emerge from each state, the total number of transitions per frame will remain constant. This leads to a moderate area increase for radix-4 decoders over radix-2 decoders [12], partly because more ACS operations per transition are required, when considering several transitions at once. The main advantage of radix-4 decoders is that by transversing two states at once, the degree of parallelism can be doubled, hence facilitating higher throughputs.

There are a number of other techniques that may be employed in turbo decoder implementations, as follows.

- Early Stopping [25, 53], which terminates the turbo decoding process early, if the correct bit-stream is unlikely to be found, thus saving energy. This technique considers the values of the LLRs, and detects if their quality no longer improves in successive decoding iterations, indicating that the remaining errors in the message will not be corrected. Furthermore, early stopping can also stop the iterative decoding process once the correct message is found, as verified using a Cyclic Redundancy Check (CRC).
- Modulo normalization [25, 52], which allows the state metrics to overflow, relying on the nature of the two's complement arithmetic to correct this overflow, instead of requiring a larger number of bits to represent these metrics. An additional logic gate is required for the max logic, in order to allow it to correctly process numbers, which have experienced an overflow.

- Voltage scaling [12, 25], which reduces the supply voltage when the throughput requirements are lower, or when less iterations are required, because the SNR is higher, resulting in a reduced energy consumption.

2.2.3 Memory considerations

Turbo decoder architectures require a large amount of memory for their operation. This memory is required for storing the *a priori* LLRs, the extrinsic LLRs generated by each of the Log-BCJR decoders and the intermediate $\tilde{\alpha}$ or $\tilde{\beta}$ values of the Log-BCJR decoder, as discussed in Section 2.1.5. While Section 2.1.6 discussed beneficial techniques, such as windowing for reducing the required memory, frequent access will still be required of this memory. It is desirable to minimize the number of memory accesses in architectures that are designed to minimise EC, since memory accesses have energy dissipations that are comparable to those of the datapath [51].

As described in Section 2.1.6, the Log-BCJR algorithm's data dependencies require an entire forward-recursion or backward-recursion to be carried out, before any extrinsic LLRs can be generated. This gives rise to the memory requirement for storing the $\tilde{\alpha}$ or $\tilde{\beta}$ values calculated during this recursion. The authors of [13] have proposed an additional method for reducing the storage requirement of state metrics during this initial forwards- or backwards-recursion. This 're-computation' method reduces the number of values stored in the memory during on the initial recursion, which is achieved by storing only every nth set of state metrics. However, this requires the missing state metrics to be recalculated as and when needed, during the subsequent pass through the trellis. The implementation advocated in [13] opted for storing every 6th set of state metrics, since it was found that the extra hardware required for the re-computation circuit occupied a smaller area than the memory, which would otherwise have been required.

For any design, the required amount of memory storage and the number of memory accesses can be traded-off against the requirement of repeating the computation of un-stored values in the decoder. However, as a minimum, the *a priori* LLRs have to be fetched from memory into the Log-BCJR decoder, while the extrinsic LLRs have to be stored from the Log-BCJR decoders into memory. The values, which require minimal computation may be readily recomputed as and when required, such as the $\tilde{\gamma}$ values, which typically necessitate no more than a single addition per transition. Conversely, due to the data dependencies, memory will be required for at least some of the forwards- or backwards- recursion values, so that they can be stored until they are needed for the duration of a window.

In high-throughput decoders, that employ parallelisation by concurrently operating multiple decoder cores, accessing the shared LLR memories may cause contention. As described in Section 2.1.2, the interleavers within the turbo decoder dictate the memory accesses of the decoder cores. In particular, the interleavers enforce the requirement for the *a priori* and extrinsic LLR memories to be shared between each of the decoder cores, rather than having independent LLR memories for each of the decoding cores. In the case where there are P decoder cores, it is desirable for the LLR memories to be split into P separate memory blocks, with the interleaver designed for ensuring that only one decoder core requires access to each memory block at a time. An interleaver that meets this criterion for some values of P is said to be contention-free [61]. However, an interleaver which is not contention-free will cause inefficiencies in the decoder, since some of the decoding blocks will have to stall their operation, while they wait to individually access the memory.

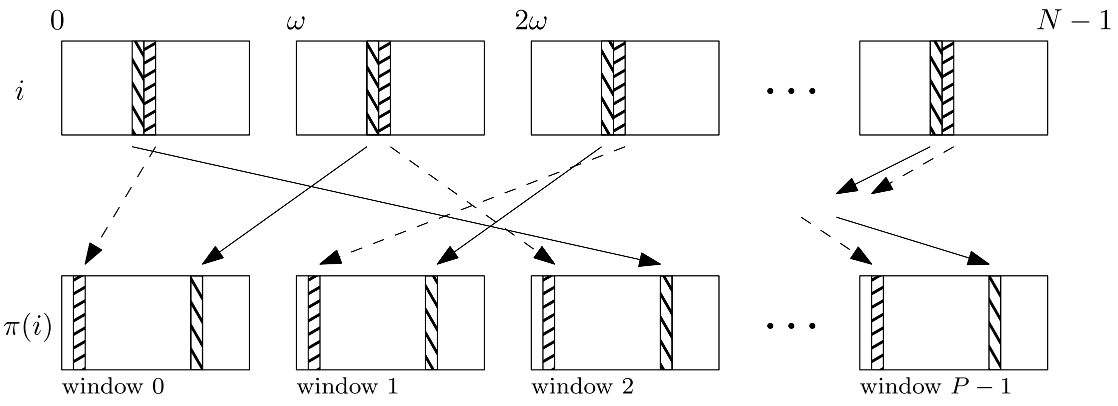


Figure 2.12: Contention-free interleaver using the same indices within each window, where address i is interleaved to yield the address $\pi(i)$. The interleaving pattern is shown for two sets of addresses.

While contention-free interleavers allow the LLR memories to be broken into separate memory blocks, the address decoding logic has to be duplicated for each of these memory blocks, hence increasing both the chip area and the associated EC. It is therefore also desirable for each decoder core to fetch or store the LLRs using the same addresses for their corresponding one from the set of these P blocks of memory. This design of the interleaver will allow contention-free memory accesses to be implemented using a single address decoding circuit, since each decoder core uses the same address. As shown in Figure 2.12, each decoder core carries out its fetching or storing action using a different memory window, but the index used within each window is the same. A pair of specific interleaver designs which meet both of these criteria are constituted by the so-called ARP and QPP interleavers [61]. The QPP design was chosen for the LTE standard [18]. The specific interleaver design has a significant affect on the BER performance of a turbo code, hence requiring a careful design of the interleaver for meeting the contention-free implementation requirement, as well as the BER performance requirement [62]. The authors of [12,63] demonstrated how to facilitate contention-free memory accesses, where a permutation network is employed for routing the LLRs between the memory and the decoder cores.

The QPP interleaver has the advantage that memory addresses can be generated readily. The QPP interleaver can be described by [61]

$$\pi(i) = (f_1 i + f_2 i^2) \mod N, \tag{2.10}$$

where the address i is interleaved to the address $\pi(i)$. Here, i is the range $0 \leq i \leq N-1$, and f_1 and f_2 are two parameters which are specific to the interleaver length N, and the interleaver design. Note that not all values of f_1 and f_2 are valid for a given length N, since certain conditions have to be met, described in detail in [64]. When N is even, which is the case for the LTE interleavers, f_1 is odd, and all prime factors of f_2 are also factors of N [61].

When incremental interleaver addresses are required, Equation (2.10) can be rearranged into a recursive formula as follows [65]

$$\begin{align} \pi(i+1) &= [(f_1 i + f_2 i^2) + (2f_2 i + f_1 + f_2)] \mod N \tag{2.11} \\ &= [\pi(i) + \Gamma(i)] \mod N \tag{2.12} \end{align}$$

where $\Gamma(i) = (2f_2 i + f_1 + f_2)$, which can be calculated itself recursively as $\Gamma(i+1) = [\Gamma(i) + 2f_2] \mod N$. This allows for simple implementation in hardware since only addition blocks are needed, as shown in [65].

2.3 Holistic design characterization

In this section, a range of methods capable of characterizing and holistically parameterising an overall wireless communications system is explored, while investigating the energy efficiency of different TCs and the effect their parameters. Referring to Figure 1.1, this section details a method which can be used to consider both the algorithm and architecture design jointly. This section explores the techniques outlined by the authors of [32] and [66], showing how these techniques can be applied to a specific scenario and architecture, in order to demonstrate the holistic design approach and to show the effect of the various system parameters on the overall EC. By considering the energy consumption in both the transmitter and the receiver, the candidate TCs may be evaluated holistically for employment in energy-constrained applications. More specifically, the transmitter's energy consumption is comprised of the turbo encoder's processing energy consumption $E_\mathrm{b}^\mathrm{enc}$, the modulator's energy consumption $E_\mathrm{b}^\mathrm{mod}$ and the Power Amplifier (PA) energy consumption E_b^tx. Likewise, the receiver energy consumption is comprised of the demodulator's energy consumption $E_\mathrm{b}^\mathrm{dem}$ and the turbo decoder's processing energy consumption $E_\mathrm{b}^\mathrm{dec}$. The techniques discussed in this section are similar to various other examples of holistic characterization that are available in the literature [67–69]. For example, the authors of [69] considered the holistic optimization of cellular networks, while the authors of [67, 68] investigated whether Multiple-Input Multiple-Output (MIMO)-based sensor networks can provide energy savings over conventional networks.

The conventional design method optimizes the algorithm and architecture separately, without considering the processing EC at the receiver alongside the transmission EC. By contrast, the methods explored in this section allow the TC to be used for reducing the overall EC of a wireless communication system. As highlighted in Figure 8.2, the holistic design characterization bridges the algorithm design and the implementation design, allowing the parameters of each to be combined, when considering the performance of the eligible schemes for a particular design scenario.

The objective of the design methods described is to determine the particular parametrization of the TC design that optimizes the overall EC of the system over the range of operating conditions expected in a particular scenario. The component encoder of the design is specified by the parameters k, m and n, as well as by the generator polynomial. Furthermore, different turbo coding schemes may be employed, which may use different arrangements of the component encoders. For example, Multiple-Component Turbo Codes (MCTCs) [70] employ multiple parallel component encoders, where the number of encoders employed also becomes a parameter of the scheme. Further parameters to be considered are those, which relate to the hardware implementation, such as which $\max^*$ approximation to utilize, as well as the number of bits used for representing the LLRs and other internal variables. Additionally, the number of decoding iterations performed also affects both the decoding EC $E_{\mathrm{b}}^{\mathrm{dec}}$ and the minimum required transmission EC $E_{\mathrm{b}}^{\mathrm{tx}}$ quite significantly.

The holistic design approaches of [32] and [66] go beyond the approaches proposed by the authors of [70, 71]. In these contributions, the decoder complexity is quantified by the number of operations undertaken in the decoder, which is related both to the number of Log-BCJR decoder activations and to the number of states in the trellis. This measure of complexity is used for representing the relative energy consumption of different codes. However, as shown in Section 2.2, the absolute energy consumption heavily depends on the architecture, as well as on factors such as the amount of memory in the design. As an example, [66] shows that two different schemes having the same operations-based complexity have a 45% difference in their processing EC $E_{\mathrm{b}}^{\mathrm{dec}}$. Furthermore, schemes having a lower coding rate also result in the modulation of more bits on to the channel, resulting in a higher $E_{\mathrm{b}}^{\mathrm{mod}}$ and $E_{\mathrm{b}}^{\mathrm{dem}}$. This illustrates that while the complexity-based comparison of [70] is useful for comparing the relative processing EC of schemes where the Log-BCJR decoders are similar, it does not allow the overall EC to be optimized, since it does not facilitate a fair comparison between different architectural parameterisations. This is because it has no knowledge of how the architecture performs the decoding, wherein different parameterisations of the architecture will cause different activation of blocks in the decoder. During the holistic optimization, the designer may also wish to compare the performance of different architectural parameterisations, which is not provided by the approaches disseminated in [70, 71].

In order to demonstrate the holistic design techniques, this tutorial considers a scenario, which is representative of a low-power, relatively low-throughput receiver, as is typical in Wireless Sensor Networks (WSNs) and in the 'Internet of Things' (IoT). This section uses the energy estimation techniques discussed in [72], to obtain estimations of the processing EC for different turbo code parameters. This thesis will consider a Twin-Component Turbo Code (TCTC) as discussed in Section 2.1.4, as well as two MCTCs [70] having three and four constituent codes, which are referred to as 3MCTC and 4MCTC, respectively. The TCTC and 3MCTC schemes are both $R = 1/3$-rate codes, while the 4MCTC is a $R = 1/4$-rate code. These TCs employ the generator polynomials $(17, 15)_o$, $(4, 7)_o$ and $(2, 3)_o$, respectively. A fourth scheme considered is provided by the uncoded case, which is associated with no TC processing energy $E_{\mathrm{b}}^{\mathrm{dec}}$, allowing us to explore the specific situations, where using TCs is the most energy efficient. A number of different parameters will also be considered for each of these codes. Furthermore, this thesis will investigate the effect of employing various approximations of the Log-BCJR algorithm, namely the LUT-Log-BCJR, the Max-Log-BCJR and the Max-SE-Log-BCJR [25]. In particular, this thesis will explore which approximations are most appropriate, when attempting to reduce the overall EC. The number of iterations in the receiver will also be considered in the holistic characterization.

Table 2.2: Environment assumptions and system specification for the two considered WSNs.

Parameter	Scenario 1	Scenario 2
Transmission frequency (f_c)	5.8 GHz	433 MHz
Power amplifier efficiency reduction (A_{Amp})	4.8 dB	5 dB
Transmit antenna gain (G_{tx})	2 dBi	-2 dBi
Receive antenna gain (G_{rx})	7 dBi	3 dBi
Receiver noise figure (RNF)	6 dB	15 dB
Path loss exponent (p)	4	2
BER target	10^{-5}	10^{-5}
Temperature	300 K	300 K
Thermal noise (N_0)	-203.8 dBJ	-203.8 dBJ
Transmitter modulator power consumption (P^{mod})	4.6 mW	1.7 mW
Receiver demodulator power consumption (P^{dem})	6.5 mW	3.3 mW
Symbol period (t_{s})	$1\mu s$	$4\mu s$

Table 2.2 shows two different operating scenarios, which will be considered in this tutorial, representing a range of environmental factors faced by energy-constrained systems. The power consumption figures P^{mod} and P^{dem} are representative of those achieved by a

particular low-power state-of-the-art transceiver [73]. These figures quantify the power consumption of the various analogue components in the transmitter and receiver. In the case of the transmitter, the power consumption P^{mod} does not include the power consumption of the PA, which instead is characterised by A_{Amp} and $E_{\mathrm{b}}^{\mathrm{tx}}$. While naturally only a limited number of parameterisations are considered in this tutorial, the designer of a real communications system may wish to consider a wider range of candidate schemes. For example, error correction codes such as Low-Density Parity-Check (LDPC) [2], Repeat Accumulate (RA) [74], or Reed-Solomon (RS) [75] codes may provide a lower overall energy consumption, depending on the scenario. For example, TCs out perform LDPC codes at lower coding rates [76], while LDPC and RA codes lend themselves to be conveniently implemented in parallel, albeit at the expense of a large chip area.

Table 2.3: Three turbo code schemes with their decoding energy consumption per bit ($E_{\mathrm{b}}^{\mathrm{dec}}$), and the E_{b}/N_0 for which a BER of 10^{-5} is achieved.

	Max				Max-SE (Scaled Extrinsic)				LUT-Max*			
	$E_{\mathrm{b}}^{\mathrm{dec}}$ (nJ)		E_{b}/N_0 (dB)		$E_{\mathrm{b}}^{\mathrm{dec}}$ (nJ)		E_{b}/N_0 (dB)		$E_{\mathrm{b}}^{\mathrm{dec}}$ (nJ)		E_{b}/N_0 (dB)	
Iterations	4	8	4	8	4	8	4	8	4	8	4	8
Uncoded	0	0	9.5	9.5	0	0	9.5	9.5	0	0	9.5	9.5
TCTC	2.31	4.63	1.4	1.0	5.00	10.01	1.0	0.6	5.00	10.01	0.9	0.5
3MCTC	2.52	5.04	1.6	0.6	5.45	10.90	1.2	0.2	5.45	10.90	1.1	0.1
4MCTC	3.36	6.72	1.6	0.6	7.27	14.53	1.1	0.2	7.27	14.53	1.0	0.1

2.3.1 Methodology

To evaluate the transmission energy required for each candidate scheme, first the BER requirement must be specified based on the target application. Here, a BER of 10^{-5} is assumed as the maximum tolerable BER. Table 2.3 shows the E_{b}/N_0 required for achieving a BER of 10^{-5} for each of the candidates considered in this tutorial. The corresponding BER simulation results of these candidate schemes are provided in [77], while the relative performance of different approximations of the Log-BCJR algorithm are taken from [59]. Next, the path loss model given in Section 2.1.3 is used for calculating the transmission EC per information bit $E_{\mathrm{b}}^{\mathrm{tx}}$, by invoking Equation (2.9). This path loss model may be substituted by alternative channel models, such as a Rayleigh fading [35] channel, if this is more appropriate for the design scenario. The assumptions and specifications for the target scenario of Table 2.2 are applied to the specific path loss model, having the parameters defined in Section 2.1. Furthermore, the decoding EC $E_{\mathrm{b}}^{\mathrm{dec}}$ of the candidate schemes can be estimated using the techniques discussed in [72]. In this way, we estimated the decoding EC $E_{\mathrm{b}}^{\mathrm{dec}}$ of a small, lower power architecture, similar to the one proposed in [51], as an example of the holistic design methodology. This decoding EC $E_{\mathrm{b}}^{\mathrm{dec}}$ is shown alongside the required E_{b}/N_0 in Table 2.3. Using the coding rates of the candidate schemes, the modulation and demodulation energy consumption can be calculated according to

$$E_{\mathrm{b}}^{\mathrm{mod}} = P^{\mathrm{mod}} \times t_{\mathrm{s}}/\eta, \tag{2.13}$$

$$E_{\mathrm{b}}^{\mathrm{dem}} = P^{\mathrm{dem}} \times t_{\mathrm{s}}/\eta. \tag{2.14}$$

The encoding energy $E_{\mathrm{b}}^{\mathrm{enc}}$ is typically considerably lower than the decoding EC [78], and therefore in this design example it is assumed to be negligible. Finally, the overall EC of the candidates can be calculated by summing these figures according to ($E_{\mathrm{b}}^{\mathrm{tx}} + E_{\mathrm{b}}^{\mathrm{mod}} + E_{\mathrm{b}}^{\mathrm{enc}} + E_{\mathrm{b}}^{\mathrm{dec}} + E_{\mathrm{b}}^{\mathrm{dem}}$), in order to obtain the combined energy consumption per bit.

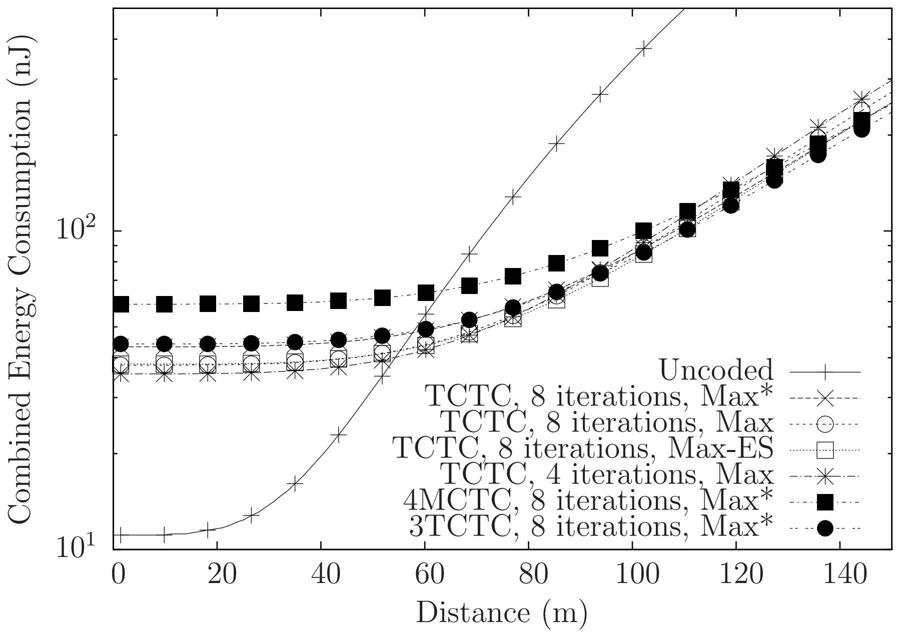


Figure 2.13: Combined EC for the four candidate schemes, when operating in scenario 1 of Table 2.2

2.3.2 Results

Figures 2.13 and 2.14 show the combined transmission and processing EC for the four candidate schemes, when operating in Scenarios 1 and 2, respectively. These graphs show how the combined EC increases with the transmission distance, allowing the designer to make decisions based on the range of required distances. It can be seen that for very short transmission distances, the uncoded candidate scheme has the lowest EC due to the processing overhead of the turbo coded schemes, as well as the additional modulator $E_{\mathrm{b}}^{\mathrm{mod}}$ and demodulator $E_{\mathrm{b}}^{\mathrm{dem}}$ energy required for transmitting the additional parity bits. As the distance increases, the turbo coded schemes overtake the uncoded one, since they facilitate a lower transmit energy. The low-complexity TC schemes have an advantage for shorter transmission distances, while the transmit power dominates the EC over

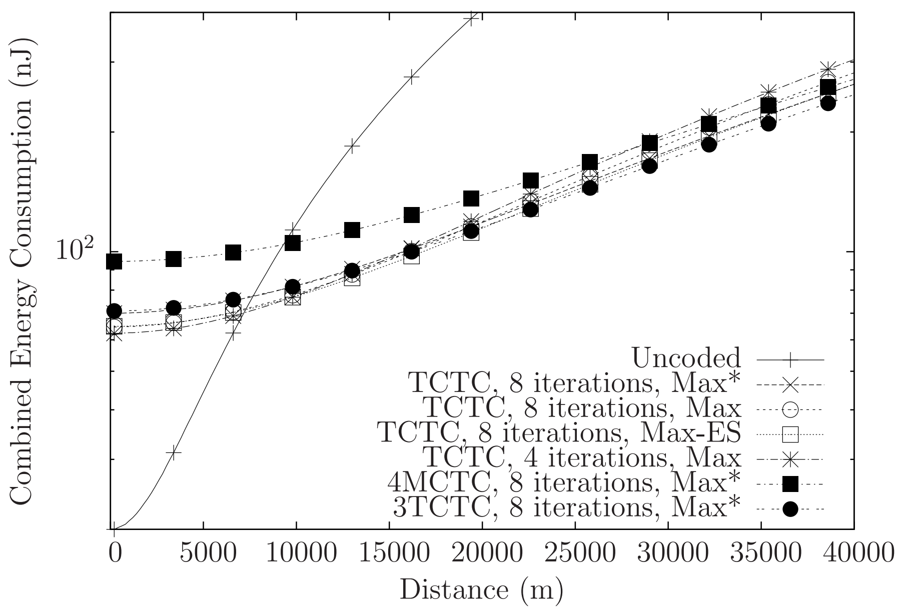


Figure 2.14: Combined EC for the four candidate schemes, when operating in scenario 2 of Table 2.2

longer distances, where the best performing scheme becomes the one having the best BER performance.

Table 2.4: The combined transmit, receive, and decoding energy consumption (E_b^tx + $E_\mathrm{b}^\mathrm{dec}$+ $E_\mathrm{b}^\mathrm{mod}$+ $E_\mathrm{b}^\mathrm{dem}$) [nJ] of the three schemes under the different conditions at a range of distances d. Brackets show number of iterations performed and approximation used. Bold font indicates the lowest energy consumption obtained for each transmission distance.

				Scenario 1				Scenario 2			
Scheme	$E_\mathrm{b}^\mathrm{dec}$	E_b/N_0	η	d [m] 20	50	100	250	5000	10000	20000	50000
uncoded	0.0	9.5	1.0	**11.6**	**31.9**	343.6	13001.3	**44.3**	117.3	409.1	2451.7
TCTC (4,Max)	2.3	1.4	0.3	35.7	38.8	87.4	2056.8	66.1	77.4	122.9	440.7
TCTC (4,Max-SE)	2.4	1.0	0.3	35.8	38.7	82.7	1870.6	65.9	**76.2**	117.4	405.9
TCTC (8,Max)	4.6	1.0	0.3	38.0	40.8	84.4	1851.8	68.0	78.2	119.0	404.2
TCTC (8,Max-SE)	4.9	0.6	0.3	38.2	40.8	**80.7**	1700.1	68.0	77.3	**114.6**	376.0
TCTC (8,Max-LUT)	10.0	0.5	0.3	43.4	45.9	84.7	1660.0	73.0	82.1	118.4	372.6
3MCTC (4,Max-SE)	2.6	1.2	0.3	36.0	39.0	85.4	1966.7	66.3	77.1	120.5	424.1
3MCTC (8,Max-LUT)	10.9	0.1	0.3	44.3	46.6	82.1	**1525.8**	73.7	82.0	115.3	**348.3**
4MCTC (4,Max-SE)	3.5	1.1	0.3	48.0	51.0	96.4	1942.8	87.1	97.7	140.3	438.2
4MCTC (8,Max-LUT)	14.5	0.1	0.3	59.0	61.3	96.8	1536.7	97.3	105.6	138.8	371.2

Table 2.4 summarizes the combined EC for a selection of the candidate schemes over a range of distances. It can be seen that the Max-SE-BCJR decoder offers an attractive trade-off. At short distances, it offers an energy saving due to its lower processing energy $E_\mathrm{b}^\mathrm{dec}$ compared to the LUT-Max-BCJR decoder, while at higher distances its slight BER performance degradation results in only a small increase of the overall EC. Compared to the Max-BCJR decoder, the Max-SE-BCJR decoder offers an improvement for the

majority of distances considered. Indeed, the only distances for which the Max-BCJR decoder offers a lower combined EC than the Max-SE-BCJR decoder is at distances, where the uncoded scheme provides the lowest EC.

The schemes offering the best BER performance are 3MCTC and 4MCTC of Table 2.3, however they also have the highest decoding EC $E_{\mathrm{b}}^{\mathrm{dec}}$. As a result, these schemes provide the lowest overall EC at longer distances, especially, when compared to the TCTC schemes, which have a slightly worse BER performance.

The authors of [32] refer to the point at which using an error correcting code becomes beneficial over uncoded transmission as the critical distance d_{cr}. This can be expressed as follows

$$\begin{aligned} &E_{\mathrm{b}}^{\mathrm{tx,e}}(d_{\mathrm{cr}}) + E_{\mathrm{b}}^{\mathrm{mod,e}} + E_{\mathrm{b}}^{\mathrm{dem,e}} + E_{\mathrm{b}}^{\mathrm{dec}} + E_{\mathrm{b}}^{\mathrm{enc}} \\ =&E_{\mathrm{b}}^{\mathrm{tx,u}}(d_{\mathrm{cr}}) + E_{\mathrm{b}}^{\mathrm{mod,u}} + E_{\mathrm{b}}^{\mathrm{dem,u}} \end{aligned} \tag{2.15}$$

where $E_{\mathrm{b}}^{\mathrm{tx,u}}(d_{\mathrm{cr}})$ is the transmission EC of the uncoded scheme at the critical distance, $E_{\mathrm{b}}^{\mathrm{tx,e}}(d_{\mathrm{cr}})$ is the transmission EC of the coded scheme at the critical distance, and $E_{\mathrm{b}}^{\mathrm{mod,e}}$, $E_{\mathrm{b}}^{\mathrm{dem,e}}$ and $E_{\mathrm{b}}^{\mathrm{mod,u}}$, $E_{\mathrm{b}}^{\mathrm{dem,u}}$ are the energy consumptions for the respective encoded and uncoded modulator and demodulator components. The critical distance depends on the particular error correction code used, as well as on all of the other factors shown in Table 2.2. Figure 2.15 shows how the critical distance varies both with the carrier frequency and with the path loss coefficient p for a variety of schemes.

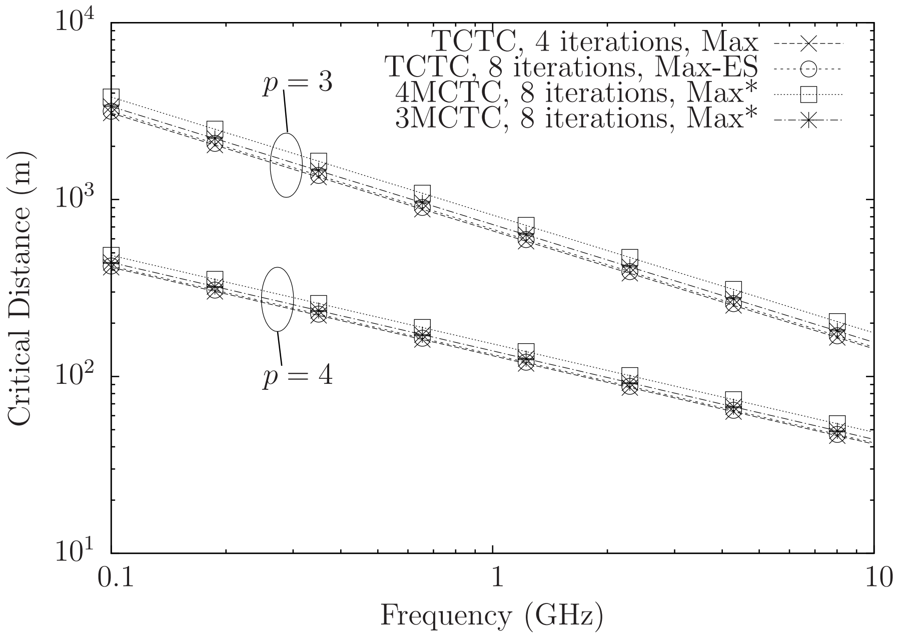


Figure 2.15: Critical distances for path loss exponents of p=3 and p=4

The case study of [70] offers a simple example for demonstrating the philosophy of the proposed holistic design method. Naturally, numerous idealized simplifying assumptions of the environment and of the WSN specifications had to be stipulated here for avoiding distraction from the holistic design methodology. As a benefit, the design methodologies discussed here are capable of assisting the designer in holistically optimizing a TC design by considering numerous different design aspects. For example, apart from the basic parameters of TC schemes that were considered in our example, the longest block length N of a TC determines both the memory requirement of the hardware implementation. The number of decoding iterations performed has a significant effect on both the BER performance and on the decoding EC. Additionally, the number of hops employed in a multi-hop network determines the average transmission range and the sensor densities. All of these aspects directly affect both the transmission EC and the decoding EC. As a result, these design methods can be used for optimizing a wide variety of related specifications for improving the system's energy efficiency.

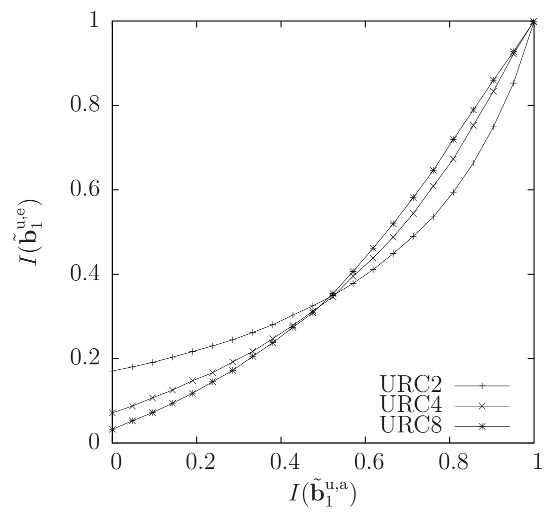
2.4 EXIT functions

In this section, this thesis discusses EXtrinsic Information Transfer (EXIT) functions [26], which can be used to characterize the performance of an iterative decoder. Firstly, we describe the EXIT function theory in Section 2.4.1, including how they can be used to estimate the SNR at which a decoder can achieve good error correction performance. Following this, Section 2.4.2 describes the relationship between the EXIT functions of a decoder and the channel capacity.

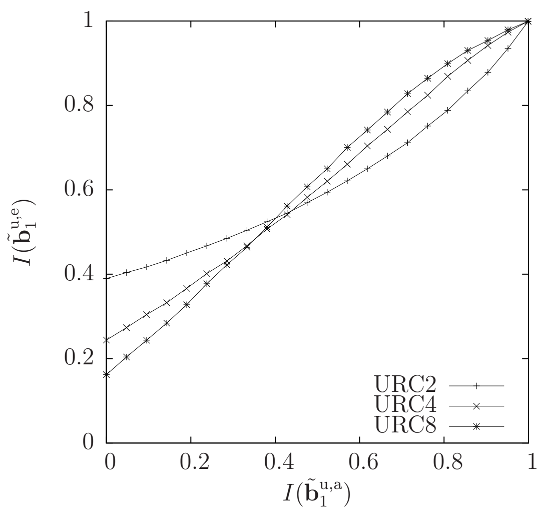
2.4.1 Theory

EXIT functions [26] may be used to quantify the performance of a particular decoding block in terms of the quality of extrinsic information output by that block as a function of the quality of the *a priori* information input to the block, and the channel SNR if the block receives LLRs from the channel. Mutual Information (MI) is used to measure the quality of information of a vector of LLRs, where an MI value of 0 represents an LLR vector where there is no confidence in the values of the corresponding bits, while an MI value of 1 represents an LLR vector with full confidence in the bit values.

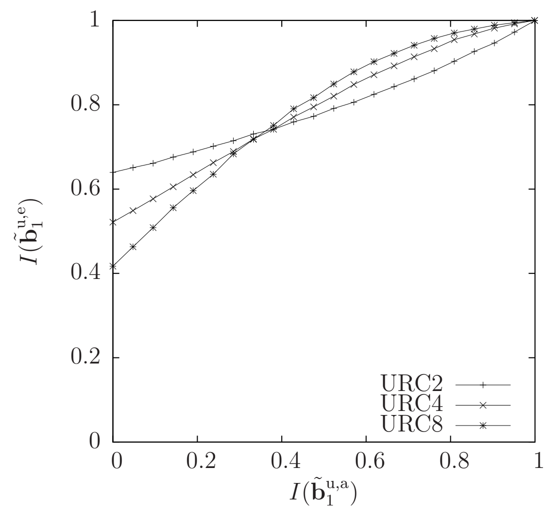
As an example, Figure 2.16 shows a series of EXIT functions for the Unity Rate Convolutional (URC) code designs URC2, URC4, URC8 of [79], which have trellises of $r =$ 2, 4 and 8 states, respectively. A URC is similar to the convolutional code discussed in Section 2.1.1, where the rate R of a URC is equal to 1. A URC can be described by a trellis in the same way as a convolutional code, as shown in Figure 2.17. Figure 2.16 shows the EXIT functions for these URC codes when the channel SNR is 0 dB, 4 dB and 8 dB. The quality of the extrinsic information $I(\tilde{\mathbf{b}}_1^{\mathrm{e}})$ is plotted as a function of the input *a priori* information $I(\tilde{\mathbf{b}}_1^{\mathrm{a}})$. This figure shows that each of the different URC designs



(a) SNR = 0 dB



(b) SNR = 4 dB



(c) SNR = 8 dB

Figure 2.16: Sample EXIT charts for three URC encoders from [79] at different channel SNRs

have a different characteristic. As a result, the URC EXIT chart which provides the best performance will depend on which URC EXIT chart provides the best match to the other codes in the scheme.

Figure 2.19 shows the method used to generate the EXIT functions. To measure the EXIT function for a given input MI and channel SNR, a random bit vector $\mathbf{b}_1$ is URC encoded to provide $\mathbf{b}_2$. LLR vectors $\tilde{\mathbf{b}}_1^{\mathrm{a}}$ and $\tilde{\mathbf{b}}_2^{\mathrm{a}}$ are generated based on the required input MI $I(\mathbf{b}_1)$ and channel SNR being investigated. After URC decoding, the MI of the extrinsic output $\tilde{\mathbf{b}}_1^{\mathrm{e}}$ is measured. To obtain the full EXIT function, this process is repeated for a range of $I(\mathbf{b}_1)$ and channel SNR values. The EXIT functions shown in Figure 2.18 are generated using a long frame length to get a smooth curve.

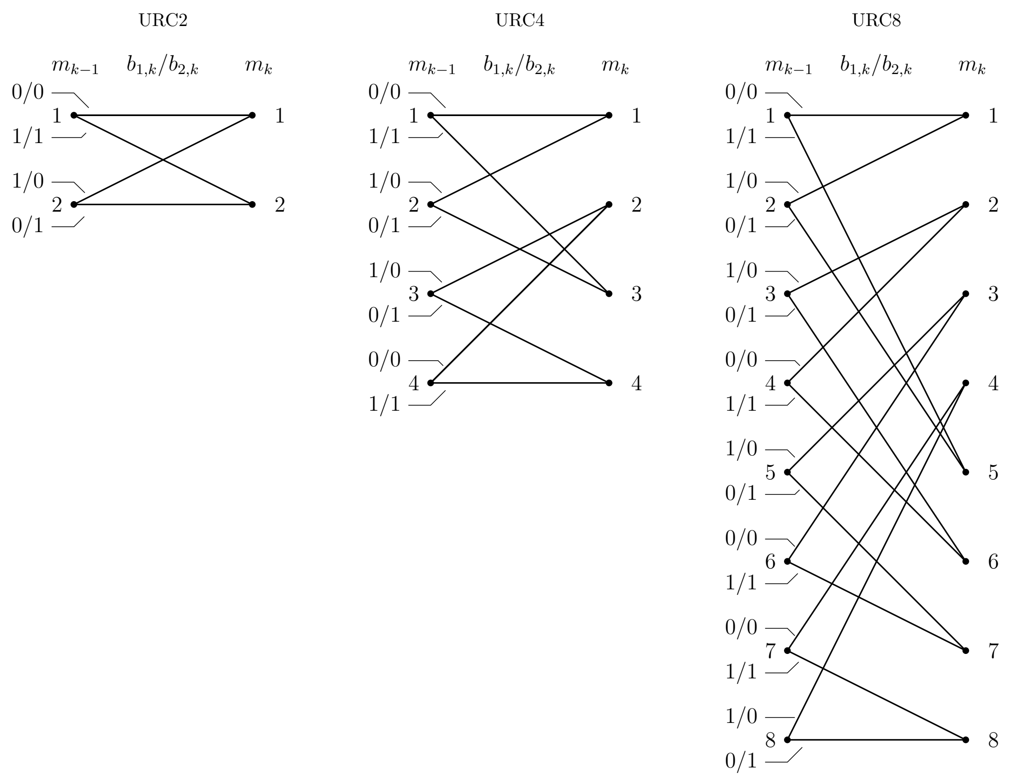


Figure 2.17: Trellises for three URC encoders URC2, URC4 and URC8 from [79]

We can use EXIT charts to predict the performance of iterative decoders. Graphically this can be achieved by plotting the EXIT trajectory, an example of which is shown in Figure 2.18. This example has two identical URC decoders in the configuration shown in Figure 2.18(d). The extrinsic outputs of the URC decoders are connected together through an interleaver and de-interleaver. This can be modelled graphically by plotting both EXIT functions on the same graph, but inverting one of them, such that the input $I(\tilde{\mathbf{b}}_1^{l,a})$ is on the y axis and the output $I(\tilde{\mathbf{b}}_1^{l,e})$ on the x axis. This means that the input of the upper Log-BCJR decoder and the output of the lower Log-BCJR decoder are on the same axis. As a result when the lower decoder outputs a frame of LLRs with a particular MI, these LLRs will be input to the upper decoder. We can then represent this passing of information as a vertical line on the EXIT trajectory. Likewise, a horizontal line is used to represent LLRs of a particular MI being output from the upper decoder and input to the lower decoder. The EXIT trajectory graph is therefore a series of alternating horizontal and vertical lines representing the passing of information between decoders, and can be used to estimate the required SNR threshold for reliable decoding.

Figure 2.18 has a series of EXIT trajectories for three different SNR values. It can be seen how reducing the SNR changes the shape of the EXIT functions. When the SNR is high enough, the trajectory can easily pass in-between the EXIT functions, reaching the (1,1) point of the graph with few activations of each decoder. This (1,1) point of the graph indicates good quality LLRs have been recovered, and the original message has also been

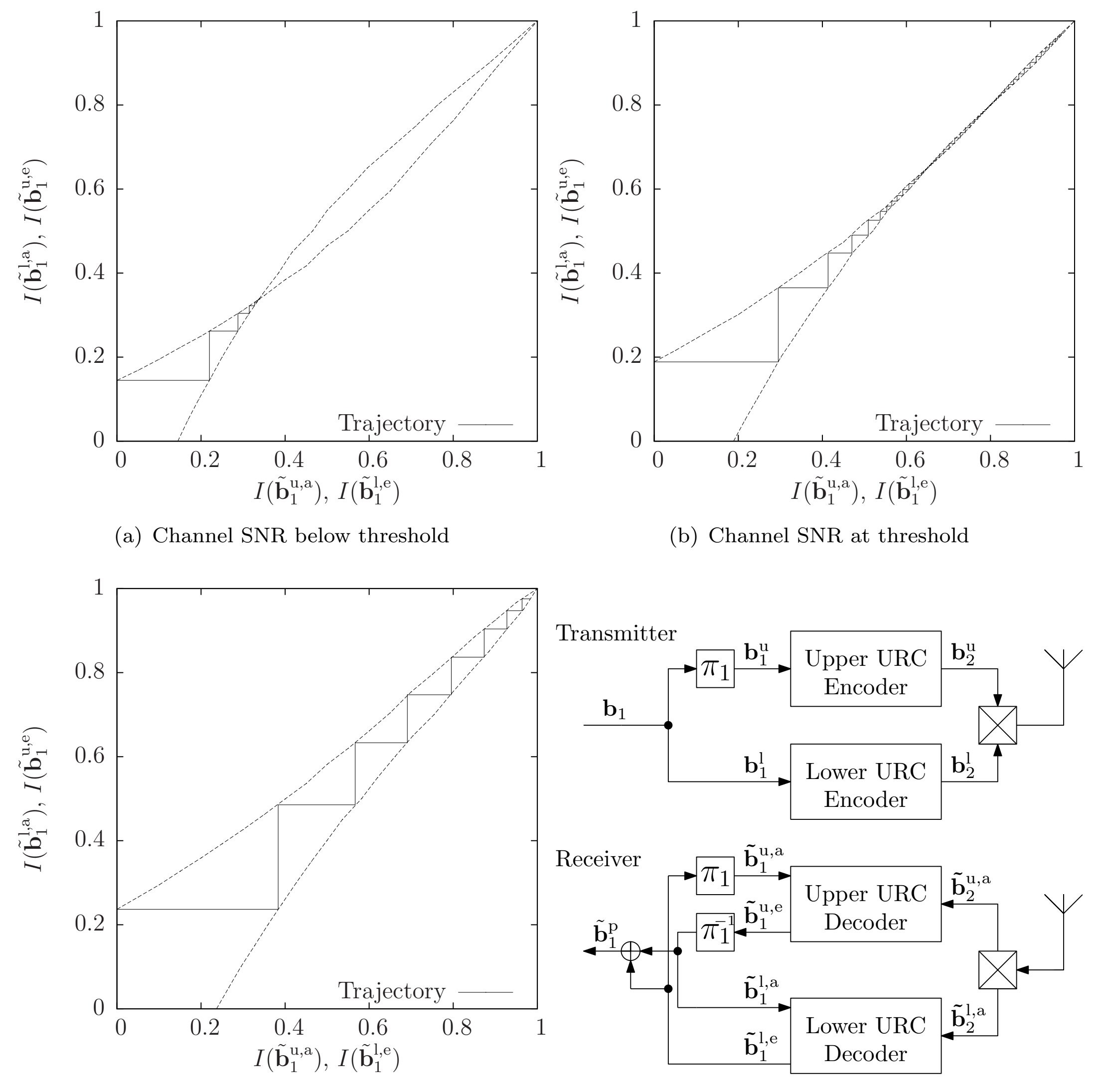


Figure 2.18: EXIT charts and trajectories for the scheme of Figure 2.18(d) at different channel SNRs

recovered. As the SNR reduces, the tunnel between the two EXIT functions narrows, as a result more iterations are required to recover the original message. When the SNR is lowered further, there is no open tunnel to the (1,1) point, therefore the message cannot be successfully recovered. The SNR at which the tunnel becomes open to allow the trajectory to pass to the (1,1) point is known as the *tunnel bound* of the scheme. Since this *tunnel bound* is the threshold at which reliable decoding becomes possible, EXIT charts can be used to characterise the performance of an iterative decoder quickly, without performing extensive BER plots. As shown in Figure 2.16, since different URC designs have different EXIT functions, we can use EXIT chart analysis to choose the URC design which gives the lowest threshold SNR when combined with other elements of the decoder.

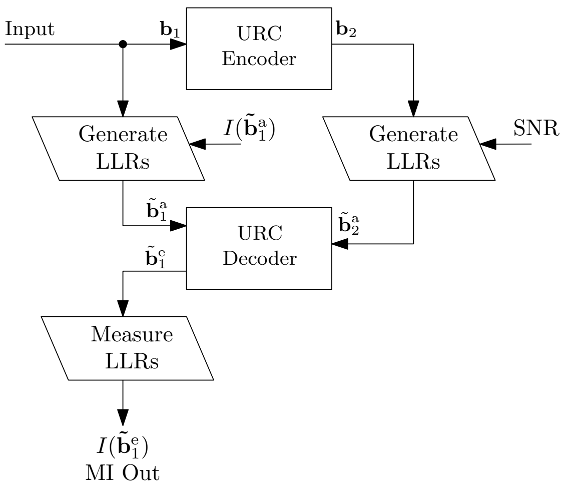

Input
$\mathbf{b}_1$
URC Encoder
$\mathbf{b}_2$
Generate LLRs
$I(\tilde{\mathbf{b}}_1^{\mathrm{a}})$
Generate LLRs
SNR
$\tilde{\mathbf{b}}_1^{\mathrm{a}}$
URC Decoder
$\tilde{\mathbf{b}}_2^{\mathrm{a}}$
$\tilde{\mathbf{b}}_1^{\mathrm{e}}$
Measure LLRs
$I(\tilde{\mathbf{b}}_1^{\mathrm{e}})$
MI Out

Figure 2.19: Method for generating the EXIT function of a decoder block

2.4.2 Near-capacity operation

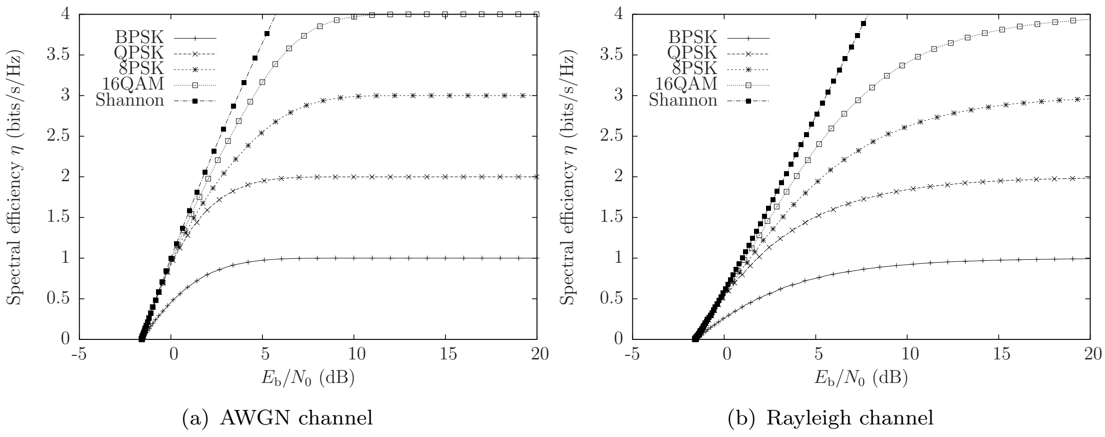


Figure 2.20: The DCMC capacity for an AWGN channel and the ergodic DCMC capacity for an uncorrelated narrowband Rayleigh channel, when the modulation scheme employed is BPSK, QPSK, 8-PSK or 16-QAM.

A communications channel has a capacity limit C, which is the maximum spectral efficiency which can be achieved with negligible errors at a given Signal-to-Noise Ratio (SNR). For a AWGN channel, this throughput is given by Shannon's capacity equation [16] $C = \log_2[1 + E_s/N_0]$, where E_s/N_0 is the channel SNR expressed as a linear ratio, and the capacity C is in bits/s/Hz. Figure 2.20 shows the DCMC capacity for an AWGN channel and the ergodic DCMC capacity for an uncorrelated narrowband Rayleigh fading channel. This figure shows the ultimate limit given by Shannon's equation, alongside the spectral efficiencies theoretically possible when different modulation schemes are employed. Note that these graphs show the capacity as a function of the channel E_b/N_0, where $E_b/N_0[\mathrm{dB}] = E_s/N_0[\mathrm{dB}] - 10\log_{10}(\eta)$, as discussed in Section 2.1.2.

The shape of EXIT functions is also closely related to the capacity of the channel. This can be seen in Figure 2.16, where the shape of the EXIT function for each decoder depends on the channel SNR. Furthermore, as shown by each sub-figure of Figure 2.16, for a given channel SNR, the area beneath the curve for each URC decoder is the same. More specifically, since these URC designs can facilitate near-capacity operation, the area under the URC EXIT function approximates $A = C/[R\log_2(M)]$ [80], where C is the channel capacity, R is the puncturing rate, and M is the number of constellation points used by the modulation scheme.

EXIT charts can also be used to analyse the near capacity operation of a scheme. More specifically, a scheme with an EXIT chart that has a very narrow tunnel at the threshold point, shown by Figure 2.18(b), can operate nearer to capacity compared to a scheme which has a much wider tunnel at the threshold point.

2.5 Unary error correction

In this section we detail the operation of the UEC code of [6], which is used for joint source and channel coding of source symbols, where these symbols may represent multimedia information. More specifically, in order to facilitate the reliable and bandwidth-efficient transmission of multimedia information, both source coding and channel coding is required. Referring to Figure 1.1, this section considers both source and channel coding topics, where some of the important contributions to the state-of-the-art of source coding or channel coding are listed in Table 2.5.

Shannon [16] proved that near-capacity operation may be achieved using SSCC. Here, a near-capacity channel code, such as a turbo code [17] or LDPC code [2], may be combined with a separate near-entropy source code, such as an arithmetic code [7] or Lempel-Ziv code [8]. However, these near-entropy source codes are typically impractical, since they assume that infinite complexity and/or latency can be afforded. For example, both the arithmetic code and Lempl-Ziv code require accurate knowledge of the probability of occurrence of each symbol value at both the transmitter and receiver. This imposes both an excessive complexity and memory requirement when the symbols are selected from an alphabet having a large or infinite cardinality, as it is typical for the encoding of multimedia information. In adaptive schemes, the transmitter and receiver learn the symbol value probabilities from the transmitted symbols, but any transmission errors result in de-synchronization between the transmitter and receiver, introducing a large number of decoding errors from that point onwards [22].

On the other hand, universal codes can encode source symbol values selected from large and infinite alphabets, without incurring the issues associated with near-entropy source codes. More specifically, a source code is said to be universal if it represents the source symbol vector using a bit vector that is guaranteed to have a finite average length for

Table 2.5: Major contributions in source and channel coding.

Year	**Author**	**Contribution**
1948	Shannon, C. [16]	The foundations of information theory.
1952	Huffman, D.A. [15]	Huffman source code.
1960	Reed, I.S.; Solomon, G. [75]	Reed-Solomon channel code.
1962	Gallager, R. [2]	The original paper on LDPC channel code.
1967	Viterbi, A.J. [81]	The Viterbi decoding algorithm.
1975	Elias, P. [82]	Elias Gamma code, Elias Omega code.
1977	Massey, J.L. [83]	Non-iterative JSCC
1978	Ziv, J.; Lempel, A. [8]	Lempel-Ziv variable rate source code.
1979	Rissanen, J. et al. [7]	Arithmetic near-entropy source coding.
1980	Stout, Q.F. [84]	The Stout source code.
1988	Bernard, M.A. et al. [23]	The VLEC JSCC.
1993	Berrou, C. et al. [1]	The turbo code.
1996	Fraenkel, A.S. et al. [85]	The Fibonacci code.
2000	Bauer, R.; Hagenauer, J. [86]	Introduces iterative decoding of VLECs.
2000, 2001	Görtz, N. [87] [88]	The introduction of iterative JSCC decoding.
2013	Maunder, R.G. et al. [6]	The UEC JSCC.
2014	Wang, T. et al. [28]	The Elias Gamma Error Correction (EGEC) JSCC.

any monotonic source symbol probability distribution. These universal codes include the Elias Gamma code [82], Elias omega code [82], Stout code [84], Fibonacci code [85] and the Exponential Golomb (ExpG) [89] code. These codes are capable of operating without any knowledge of the source symbol value probabilities, granting them immunity to the deleterious synchronization problems that detrimentally affect both the arithmetic and Lempl-Ziv codes. However, typically non-negligible redundancy remains in the encoded bitstreams produced by these source codes, which leads to capacity loss, when combined with a separate channel code. Motivated by this, JSCC [3] may be employed for exploiting the residual redundancy that remains after source encoding for enhancing the attainable error correction capability. Against this background, the UEC code [6] of Figure 2.21 facilitates the JSCC of source symbol values selected from alphabets having large or infinite cardinalities, while maintaining near-capacity operation and imposing only a modest decoding complexity. More specifically, the complexity of the UEC code does not increase as the cardinality of the source alphabet increases. This is in contrast to the Variable Length Error-Correction (VLEC) code, which is an example of a JSCC scheme where the complexity significantly increases as the cardinality increases. Furthermore, the UEC code can operate with short message sequences. and is resilient to de-synchronization, in contrast to many near-capacity SSCC schemes.

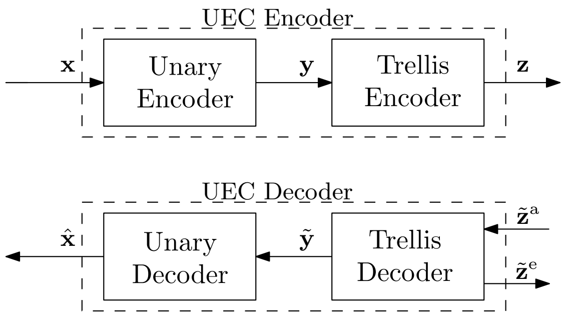


Figure 2.21: The UEC encoder and decoder

Section 2.5.1 commences by first describing the zeta distribution, which may be used to model the distribution of symbols encoded by the UEC scheme. Following this, Section 2.5.2 describes how unary symbols are encoded into bits, followed in Section 2.5.3 by a description of the complete UEC encoder. Finally, Section 2.5.4 discusses the UEC decoder.

2.5.1 Zeta distribution

Here, we detail the zeta distribution which can be used to model the distribution of many source symbols, including a H.264 or H.265 encoder, as well as many other distributions of physical phenomena which follow Zipf's law [90]. The probability distribution of symbols output from a H.264 encoder can be seen in Figure 2.22. Here, the zeta distribution is also plotted with a range of parameterisations p_1, which is the probability of a symbol adopting the value 1. Note that the zeta distribution will be used in this analysis, however the UEC code could be applied to any monotonic symbol source, such as the geometric distribution, as discussed in [6]. The UEC code has the advantage that it can be applied to symbols from a very large, or even infinite set, without requiring knowledge of the probability of occurrence of these symbols. However, as will be explained in Section 2.5.4, the UEC decoder may benefit from knowing the probabilities of the first few symbols.

The UEC encodes a vector of source symbols $\mathbf{x} = [x_i]_{i=1}^{a}$ having length a, which is obtained as a realization of a vector of Independent and Identically Distributed (IID) Random Variable (RV) $\mathbf{X} = [X_i]_{i=1}^{a}$. More specifically, each RV X_i adopts a value in the range $\{1, 2, 3, ..., \infty\}$, according to a particular source symbol distribution. In the case when the source symbols obey a zeta source symbol distribution, the probability distribution $\Pr(X_i = x) = P(x)$ is given by

$$P(x) = \frac{x^{-s}}{\sum_{\hat{x}\in\mathbb{N}_1} \hat{x}^{-s}} = \frac{x^{-s}}{\zeta(s)}, \tag{2.16}$$

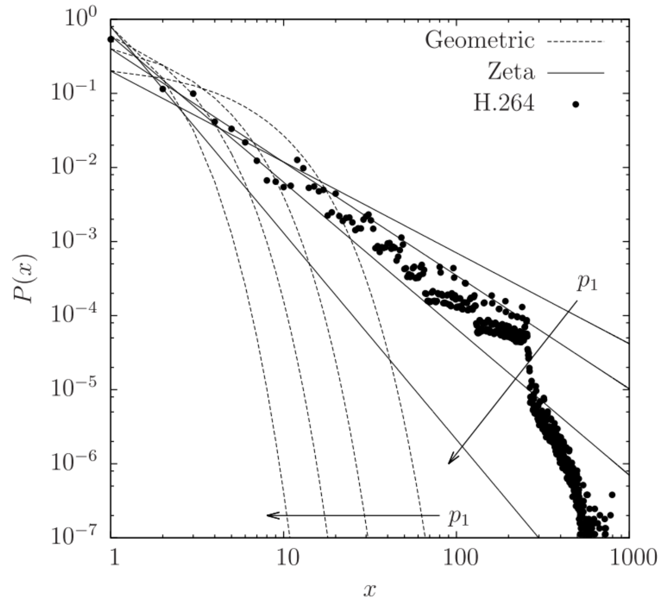


Figure 2.22: The probability distribution of symbols output from a H.264 decoder, shown alongside the zeta and geometric distributions where $p_1 \in \{0.2, 0.4, 0.6, 0.8\}$. From ©IEEE [6].

Table 2.6: The mapping of symbols x_i to unary encoded bit vectors $\mathbf{y}_i$

x_i	$\mathbf{y}_i$ =Unary(x_i)
1	1
2	01
3	001
4	0001
5	00001
6	000001
7	0000001
$\vdots$	$\vdots$

where $\zeta(s) = \sum_{x\in\mathbb{N}_1} x^{-s}$ is the Riemann zeta function. Here, the variable $s > 1$ is related to the parameter p_1 according to $p_1 = \Pr(X_i = 1) = 1/\zeta(s)$. The entropy of the source symbols is given by [6]

$$H_X = \sum_{x\in\mathbb{N}_1} P(x)\log_2\frac{1}{P(x)} = \frac{\ln(\zeta(s))}{\ln(2)} - \frac{s\zeta'(s)}{\ln(2)\zeta(s)} \tag{2.17}$$

where $\zeta'(s) = -\sum_{x\in\mathbb{N}_1}\ln(x)x^{-s}$.

2.5.2 Unary encoder

As shown in Figure 2.21, the unary encoder accepts an input vector of symbols $\mathbf{x}$. For each symbol x_i, the unary encoder outputs the corresponding x_i-bit unary codeword $\mathbf{y}_i$, according to Table 2.6. Note that each unary codeword has the last bit as 1-valued, while the remaining $(x_i - 1)$ bits are 0-valued. Since the length of each unary encoded symbol is equal to its value, we can express the average unary codeword length l_{Unary} as [6]

$$l_{\text{Unary}} = \sum_{x \in \mathbb{N}_1} P(x)x = \frac{\zeta(s-1)}{\zeta(s)}. \tag{2.18}$$

After unary encoding, each codeword $\mathbf{y}_i$ is concatenated to form the N-bit long vector $\mathbf{y} = [y_i]_{i=1}^{N}$. For example, given the symbol vector $\mathbf{x} = [x_i]_{i=1}^{a} = [1, 3, 6, 1, 4, 1, 1]$ containing a symbols, the unary encoder yields the $N = 17$-bit long vector $\mathbf{y} = 10010000011000111$.

2.5.3 UEC trellis encoder

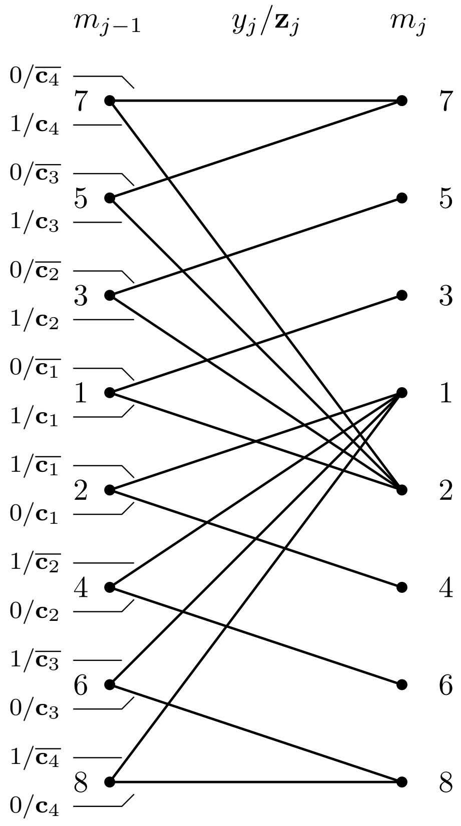

m_{j-1}
$y_j/\mathbf{z}_j$
m_j
$0/\overline{\mathbf{c}_4}$
$1/\mathbf{c}_4$
$0/\overline{\mathbf{c}_3}$
$1/\mathbf{c}_3$
$0/\overline{\mathbf{c}_2}$
$1/\mathbf{c}_2$
$0/\overline{\mathbf{c}_1}$
$1/\mathbf{c}_1$
$1/\overline{\mathbf{c}_1}$
$0/\mathbf{c}_1$
$1/\overline{\mathbf{c}_2}$
$0/\mathbf{c}_2$
$1/\overline{\mathbf{c}_3}$
$0/\mathbf{c}_3$
$1/\overline{\mathbf{c}_4}$
$0/\mathbf{c}_4$
7
5
3
1
2
4
6
8
7
5
3
1
2
4
6
8

Figure 2.23: An 8-state UEC trellis having codewords $\mathbb{C} = \{\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_4\}$, designed according to [6].

As shown in Figure 2.21, the unary encoder is combined with a trellis encoder to form the UEC encoder. Likewise at the receiver, a unary decoder is combined with a trellis decoder to form the UEC decoder. The trellis encoder and decoder is designed to exploit the redundancy of the unary encoder to provide near capacity operation.

Figure 2.23 shows a trellis employed by the trellis encoder. Here, $r = 8$ states are shown, however it can readily be extended to more states to provide operation nearer capacity, or it can operate with a reduced state count to reduce the required complexity. The bit vector $\mathbf{y}$ from the unary encoder is forwarded to the trellis encoder. For each successive input bit y_j in $\mathbf{y} = [y_j]_{j=1}^{N}$, the trellis transitions from a previous state $m_{j-1} \in \{1, 2, 3, ..., r\}$ to a next state $m_j \in \{1, 2, 3, ..., r\}$. The path taken through the UEC trellis when encoding the bit-vector $\mathbf{y}$ may be represented as $\mathbf{m} = [m]_{j=0}^{N}$, comprising $N + 1$ state values, where m_0 and m_N are the trellis start and end states. Each zero-valued bit y_i in the unary-encoded bit-vector causes transitions to states towards the outside of the trellis. By contrast, one-valued bit causes the trellis to transition back to one of the central states, according to

$$m_j = \begin{cases} 1 + \text{odd}(m_{j-1}) & \text{if } y_j = 1 \\ \min[m_{j-1} + 2, r_1 - \text{odd}(m_{j-1})] & \text{if } y_j = 0 \end{cases}. \tag{2.19}$$

Here, the function odd$(\cdot)$ returns zero if its operand is even, and one if it is odd. Note that each unary codeword comprises a sequence of zero-valued bits which is terminated with a single one-valued bit. The trellis structure of Figure 2.23 exploits this to maintain synchronization between the unary-encoded symbols and the path through trellis. In particular, the final one-valued bit N_j in each unary codeword will cause a transition to either state $m_j = 1$ or $m_j = 2$. For example, given the unary-encoded bit-vector comprising $N = 17$ bits $\mathbf{y} = 10010000011000111$, the path through the trellis of Figure 2.23 can be expressed as a vector $\mathbf{m} = [1, 2, 4, 6, 1, 3, 5, 7, 7, 7, 2, 1, 3, 5, 7, 2, 1, 2]$ of $N+1 = 18$-states. The path $\mathbf{m}$ may be modelled as a realization of a vector of RVs $\mathbf{M} = [M_j]_{j=0}^{N}$, where the probability of each state being selected $\Pr(M_j = m|M_{j-1} = m') = P(m|m')$, is given by [6, Eqn. (9)]. Knowledge of these conditional transition probabilities $P(m|m')$ may be exploited to aid the receiver, as described in Section 2.5.4.

Depending on the path selected through the UEC trellis, each of the bits in the unary-encoded bit-vector $\mathbf{y}$ is encoded using a n-bit codeword $\mathbf{z}_j$, which are concatenated to form the Nn-bit UEC-encoded vector $\mathbf{z} = [z_k]_{k=1}^{Nn}$. Each codeword $\mathbf{z}_j$ is selected from the set of $r/2$ codewords $\mathbb{C} = [\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, ..., \mathbf{c}_{r/2}]$ or from the complementary set $\overline{\mathbb{C}} = [\overline{\mathbf{c}_1}, \overline{\mathbf{c}_2}, \overline{\mathbf{c}_3}, ..., \overline{\mathbf{c}_{r/2}}]$, according to

$$\mathbf{z}_j = \begin{cases} \overline{\mathbf{c}_{\lceil m_{j-1}/2}} & \text{if } y_j \neq \text{odd}(m_{j-1}) \\ \mathbf{c}_{\lceil m_{j-1}/2\rceil} & \text{if } y_j = \text{odd}(m_{j-1}) \end{cases}. \tag{2.20}$$

After trellis encoding, the bit vector $\mathbf{y}$ is provided to other encoders as required by the specific scheme.

The coding rate R^{o} of the trellis encoder is given by [6]

$$R^{\mathrm{o}} = \frac{H_X}{l_{\mathrm{Unary}} n}. \tag{2.21}$$

Here R^{o} is used to denote the rate of the outer code, since the trellis encoder is usually concatenated with other encoders. Since the average unary codeword length l_{Unary} is infinite for $p_1 < 0.608$, the coding rate is only non-zero when $p_1 > 0.608$.

2.5.4 UEC trellis decoder

In this section, this thesis will describe the operation of the UEC symbol decoder of Figure 2.21. Here, the trellis decoder applies the Log-BCJR algorithm discussed in Section 2.1.5 to the trellis of Figure 2.23 in order to convert the vector of *a priori* LLRs $\tilde{\mathbf{z}}^{\mathrm{a}}$ into the vector of *a posteriori* LLRs $\tilde{\mathbf{y}}$, as well as the vector of extrinsic LLRs $\tilde{\mathbf{z}}^{\mathrm{e}}$. This extrinsic LLR vector $\tilde{\mathbf{z}}^{\mathrm{e}}$ is exchanged with other decoders in the system in order to facilitate iterative decoding. The performance of the UEC trellis decoder may be enhanced by exploiting knowledge of the conditional transition probabilities $P(m_j|m_{j-1})$ of the trellis, as discussed in Section 2.5.3. More specifically, the decoder requires knowledge of the average unary codeword length l_{Unary} and the first $r/2 - 1$ values of $P(x)$ in order to compute the transition probabilities $P(m_j|m_{j-1})$ according to [6, Eqn. (9)]. The logarithm of these conditional transition probabilities $ln[P(m_j|m_{j-1})]$ may be exploited during the computation of the *a priori* transition probabilities $\tilde{\gamma}$ of (2.4), which are employed by the Log-BCJR algorithm, as described in Section 2.1.5.

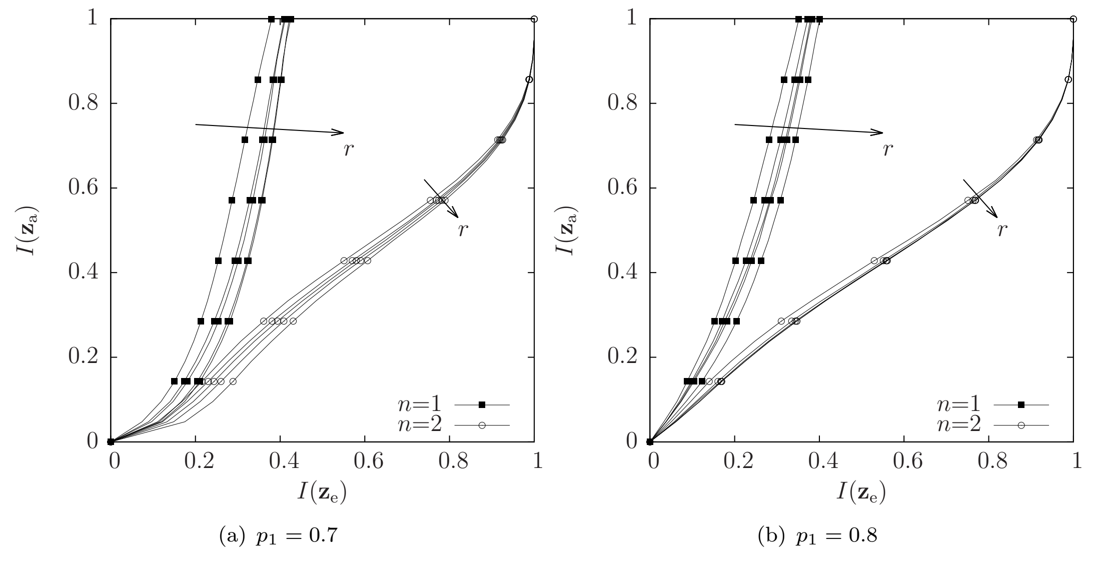


Figure 2.24: Inverted EXIT functions for the UEC decoder with a $n = 1$- and $n = 2$-bit codewords and $r \in \{4, 6, 8, 10, 12\}$ states.

The transformation of $\tilde{\mathbf{z}}^{\mathrm{a}}$ into $\tilde{\mathbf{z}}^{\mathbf{e}}$ may be characterized by the UEC trellis decoder's inverted EXIT function [26], as shown in Figure 2.24. Here, as a convention, the inverted

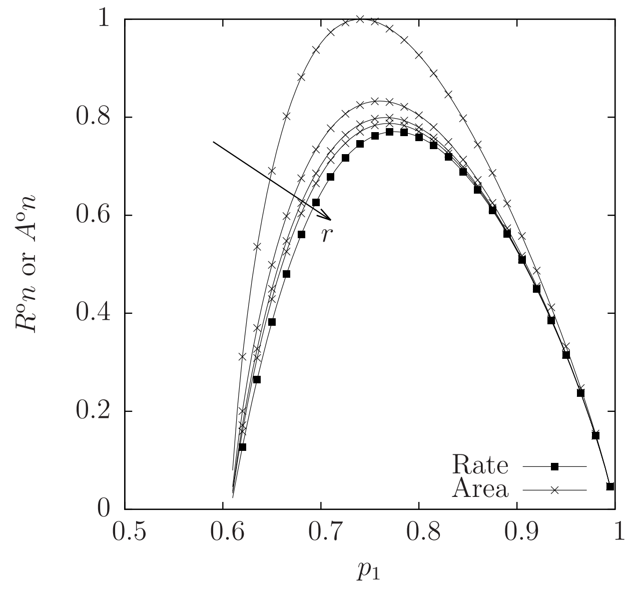


Figure 2.25: The rate and area for a UEC decoder having $r \in \{2, 4, 6, 8\}$ states, with different p_1 values.

EXIT function is drawn (as opposed to the non-inverted EXIT function) to allow the *tunnel bound* to be found when combined with a non-inverted EXIT function of an inner code, such as a URC EXIT function, as described in Section 2.4. This figure shows the EXIT functions for the scenario where the input symbols follow a zeta distribution having a $p_1 = 0.7$ and $p_1 = 0.8$. This figure shows the impact of increasing the number of states in the UEC trellis decoder, when a $n = 1$ or $n = 2$-bit codeword is used. More specifically the $n = 1$ codewords are $\mathbb{C} = [1, 1, 1, ...]$, and the $n = 2$ codewords are $\mathbb{C} = [01, 11, 11, ...]$. Note that the EXIT functions for the $n = 1$-bit codewords do not reach the (1,1) point where a low error rate is achieved, as a result, in this case the UEC decoder would have to be combined with multiple iterative decoders to achieve a low error rate, as explored further in Chapters 3 and 5. In contrast, the EXIT functions for the $n = 2$-bit codewords do reach the (1,1) point, therefore they only require one additional iterative component to achieve a low error rate.

Figure 2.25 shows the product of the coding rates R^o and inverted EXIT chart areas A^o with the corresponding codeword lengths n as a function of p_1, when the UEC decoder employs $r \in \{2, 4, 6, 8\}$ states. Here the products $A^o n$ and $R^o n$ have been plotted to eliminate the effect of different codeword lengths n on the analysis. This shows that as the number of states in the UEC decoder increases, the area $A^o n$ approaches the rate $R^o n$, and $A^o n - R^o n$ approaches zero. Since the term $A^o n - R^o n$ represents the capacity loss of the code, an increased number of states of the UEC decoder results in operation nearer capacity.

Once a sufficiently high number of iterations between the UEC trellis decoder and the rest of the system have been completed, the trellis decoder outputs the a posteriori LLR vector $\tilde{\mathbf{y}} = [\tilde{y}_j]_{j=1}^{N}$, which is provided to the unary decoder, as shown in Figure 2.21. Note that since each unary codeword contains only a single 1-valued bit, there are guaranteed to be a number of 1-valued bits in the unary-encoded bit vector $\mathbf{y}$. The unary decoder exploits this observation by selecting 1-valued hard decisions for the a highest LLR values in the a posteriori LLR vector $\tilde{\mathbf{y}}$, since high LLR values indicate that the corresponding bit in $\mathbf{y}$ is likely to have the value 1. Following this, 0-valued hard decisions are selected for the remaining $(N - a)$ LLRs in $\tilde{\mathbf{y}}$. Note that in practice, the value of a may be reliably conveyed to the receiver using a small amount of side information. Alternatively, a 1-valued hard decision may be selected for all positive LLRs in $\tilde{\mathbf{y}}$, while 0 may be selected for all of the other LLRs, avoiding the requirement for side information, but degrading the error correction performance. Finally, the unary decoder then turns the hard decision bit-vector $\tilde{\mathbf{y}}$ into symbols $\tilde{\mathbf{x}}$ according to Table 2.6.

2.6 Background summary

This chapter has introduced the main topics which form the basis for this thesis. In Section 2.1, turbo coding and the Log-BCJR algorithm was discussed, which facilitates wireless systems to operate with reduced transmit power, and/or over greater distances. Throughout this thesis, variants of the turbo coding scheme will be investigated, which will rely on the Log-BCJR algorithm, or its variants, to undertake decoding in the receiver.

Following this, in Section 2.2, state of the art turbo decoder architectures were investigated. The Log-BCJR turbo decoding algorithm has a high decoding complexity, therefore it is desirable to reduce the hardware cost as much as possible. The complexity of the Log-BCJR algorithm will be considered in all the chapters of this thesis, while Chapters 5, 6 and 7 will propose architectures which offer an improved hardware efficiency.

Section 2.3 completed the discussion of hardware implementation by describing a method to holistically optimise a wireless system, by jointly considering the transmitter's transmit power, as well as the receiver's decoding energy.

Section 2.4 discusses EXIT functions, which provide insight to the operation of iterative decoding schemes. EXIT functions will be extensively used by Chapters 3 and 4 for characterising and predicting the performance of turbo decoder schemes.

Finally, in Section 2.5, the UEC joint source and channel code was introduced, where the operation of this near-capacity scheme was described. Different variants of UEC schemes will be considered throughout this thesis. Chapter 3 reduces the complexity of the UEC scheme, and Chapter 4 increases the applicability of the UEC based scheme, while Chapter 5 focuses on the implementation of a UEC based scheme. Finally, Chapter 6 also focuses on the implementation of a UEC based scheme, but also achieves a significantly improved hardware implementation by jointly considering the decoding algorithm.

Chapter 3

Adaptive unary error correction decoder

3.1 Introduction

As discussed in Section 2.5. wireless multimedia transmission has to be both bandwidth-efficient and resilient to transmission errors, motivating both source and channel coding. Traditionally, Separate Source and Channel Codings (SSCCs) have been employed for satisfying these requirements. For example, Elias Gamma (EG) codes [82] represent a typical source code, which may be concatenated with channel codes such as turbo codes [1]. However, SSCCs lead to capacity loss due to the residual redundancy that typically remains following source encoding, hence potentially limiting the error correction performance of the scheme [6]. In general, SSCC can be replaced by Joint Source and Channel Codings (JSCCs), which exploit the residual redundancy for the sake of achieving an improved error correction capability [91]. However, the source symbols produced by multimedia codecs such as the H.264 video encoder are approximately zeta distributed [6], with most symbols having a low value, but with infrequent symbols having high values of around 1000. The resultant high cardinality of the source alphabet prevents the employment of classic JSCCs, such as the Variable Length Error-Correction (VLEC) [92], owing to the associated computational complexity. This potentially excessive complexity motivated the design of the Unary Error Correction (UEC) code which was discussed in Section 2.5.3. This JSCC scheme maintains a modest complexity, even when the cardinality of the source alphabet is infinite.

This chapter is partially based on the following publication.
M. F. Brejza, W. Zhang, R. G. Maunder, B. Al-Hashimi and L. Hanzo (2015). Adaptive iterative detection for expediting the convergence of a serially concatenated unary error correction decoder, turbo decoder and an iterative demodulator. In *IEEE International Conference on Communications (ICC), 2015* (pp. 2603–2608).

In [27] the UEC encoder was concatenated with a turbo encoder, corresponding to an iterative decoder comprising three decoding blocks. Following the activation of each decoding block, the next block to be activated must be chosen from the other two decoding options. Furthermore, the UEC decoder offers a number of decoding options [27], allowing its complexity versus error correction capability tradeoff to be dynamically adjusted. An adaptive iterative decoding algorithm was proposed in [27] for making a dynamic on-line selection of the decoding option to activate at each stage of the iterative decoding process. It was demonstrated that iterative decoding convergence can be expedited by activating the specific decoding option that offers the highest Mutual Information (MI) improvement to computational complexity ratio. This enables a frame to be decoded either at a reduced computational complexity or at a reduced Symbol Error Rate (SER), when compared to a static activation order of the decoders.

In this chapter, the previous contribution of [27] is extended by additionally concatenating the UEC code and turbo code with an iterative demodulator, producing an iterative receiver comprising four decoding blocks. EXtrinsic Information Transfer (EXIT) chart analysis is used for demonstrating that this iterative demodulator complements conveniently the turbo code, with the aid of reduced-complexity components, without increasing the E_b/N_0 at which iterative decoding convergence to a low SER becomes possible. This chapter demonstrates that the introduction of iterative demodulation actually reduces the overall complexity of the iterative receiver. However, since the iterative demodulator is invoked for recovering different bits for forwarding to the UEC decoder and to the turbo component decoders, the extension of the adaptive iterative decoding algorithm of [27] to this scheme is not trivial. More specifically, the MI improvements associated with the iterative demodulator cannot be compared on a like-for-like basis with those offered by the three other decoding blocks. This chapter proposes a solution to this problem, which may also be applied in other multi-component iterative receivers. More specifically, the operation of the iterative demodulator is jointly considered with one of the turbo component decoders, allowing all decoding options or activation orders to be considered on the basis of MI improvements pertaining to the same bits. In this chapter, the practical implementation of the adaptive algorithm is also explored, where a simple approximation of the MI measurement function is proposed. This reduces the associated implementational complexity by avoiding floating point logarithmic and exponential function evaluations, without significantly degrading the SER performance.

Section 3.2 commences by introducing the UEC code and our proposed schematic. Section 3.3 provides the EXIT chart [26] analysis of the decoding blocks, which form the basis of the adaptive algorithm. This section concludes by analysing the implementational complexity of the proposed scheme, firstly quantifying its storage requirements, followed by detailing the proposed MI approximation. Error correction results are provided in Section 3.4, where the proposed scheme is found to offer up to 2 dB performance gain without increasing the required transmission energy, bandwidth, delay or decoder complexity, while Section 3.5 offers the concluding remarks.

3.2 Transmitter and receiver scheme

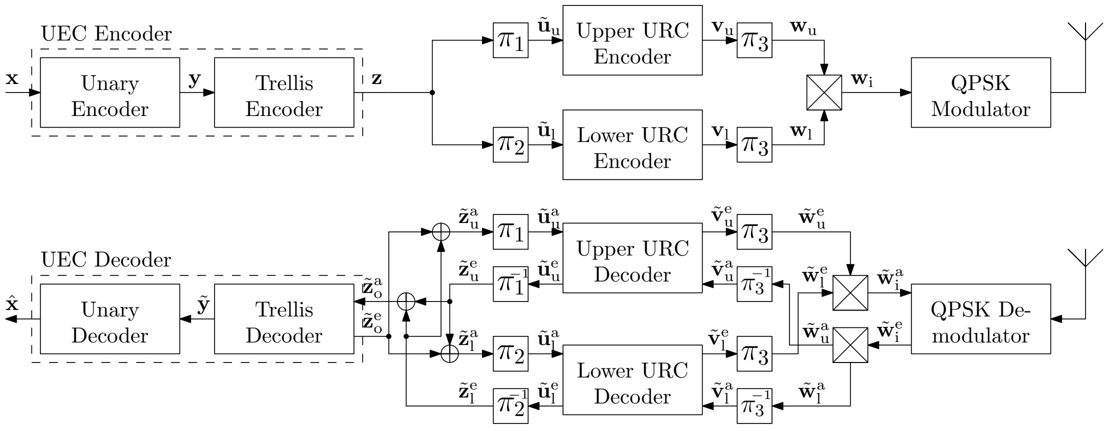


Figure 3.1: Schematic of the proposed transmitter (top), as well as the proposed receiver (bottom)

This section details the operation of the UEC-Turbo-QPSK scheme of Figure 3.1. This scheme is an extension of the scheme proposed in [27], which was the first to employ the adaptive activation order algorithm. The transmitter's operation is described in Section 3.2.1, while the receiver is considered in Section 3.2.2.

3.2.1 Transmitter

The transmitter design of the proposed joint source coding, channel coding and modulation scheme is shown in Figure 3.1. This scheme is comprised of a UEC code, as previously discussed in Section 2.5, a turbo code, as previously discussed in Section 2.1, and a Quadrature Phase Shift Keying (QPSK) modulator and demodulator. Note that this scheme can be readily extended to employ high-order modulation if required. This scheme is designed for conveying a vector $\mathbf{x} = [x_i]_{i=1}^a$ of a source symbols, each of which has a zeta-distributed integer value x_i in the range spanning from one to infinity $x_i \in N_1$, and parameterised by $p_1 = P(x_i = 1)$ [6]. This distribution models the symbols generated by a H.264 or H.265 video encoder [6], for example. Each vector of symbols $\mathbf{x}$ is input to the UEC source encoder, which comprises a unary encoder and a trellis encoder. The unary encoder produces a binary codeword $\mathbf{y}_i$ for each symbol x_i, according to Table 2.6, where the length of the unary codeword is equal to the value of the symbol. The codewords are then concatenated together, forming the bit vector $\mathbf{y} = [y_j]_{j=1}^N$. Here, $N = \sum_{i=1}^a x_i$ is the length of the bit vectors $\mathbf{y}$ and $\mathbf{z}$, which can vary from one frame to the next, owing to the varying lengths of the unary codewords produced by the UEC encoder.

As described in Section 2.5.3, the trellis encoder operates on the basis of a trellis, transforming the bit vector $\mathbf{y}$ into the bit vector $\mathbf{z} = [z_k]_{k=1}^{N}$. To elaborate further, Figure 2.23 illustrates an $r = 8$-state UEC trellis, which may be employed in the transmitter of Figure 3.1. Here, the $n = 1$ bit codewords are $\mathbb{C} = [0, 1, 1, 1]$. When encoding each bit vector $\mathbf{y}$, the trellis encoder emerges from state 1. Each bit of y_i causes the encoder to traverse from the previous state m_{j-1} to a new state m_j and to output a bit z_j. These bits are concatenated to form the output bit vector $\mathbf{z}$. In the case of a source having a zeta distribution, the UEC coding rate R^{o} is given by $R^{\mathrm{o}} = H_X/l_{\mathrm{Unary}}$, since $n = 1$. In this chapter, a zeta distribution where $p_1 = 0.797$ is used, which results in a coding rate of $R^{\mathrm{o}} = 0.762$. Here the superscript ^{o} indicates relevance to the outer code.

Following UEC encoding, two replicas of the vector $\mathbf{z}$ are interleaved by the interleavers π_1 and π_2 to obtain the bit vectors $\mathbf{u}_{\mathrm{u}} = [u_{\mathrm{u},k}]_{k=1}^{N}$ and $\mathbf{u}_{\mathrm{l}} = [u_{\mathrm{l},k}]_{k=1}^{N}$, as shown in Figure 3.1. This scheme uses a variant of the turbo code discussed in Section 2.1 for channel coding. More specifically, turbo encoding is performed upon $\mathbf{u}_{\mathrm{u}}$ and $\mathbf{u}_{\mathrm{l}}$ by a parallel concatenation of Unity Rate Convolutional (URC) encoders, which generate the resultant bit vectors $\mathbf{v}_{\mathrm{u}} = [v_{\mathrm{u},k}]_{k=1}^{N}$ and $\mathbf{v}_{\mathrm{l}} = [v_{\mathrm{l},k}]_{k=1}^{N}$. More specifically, for each input bit $u_{\mathrm{u},k}$ and $u_{\mathrm{l},k}$ in $\mathbf{u}_{\mathrm{u}}$ and $\mathbf{u}_{\mathrm{l}}$, each URC encoder traverses from the current state to the next state, outputting a corresponding bit $v_{\mathrm{u},k}$ or $v_{\mathrm{l},k}$. This chapter considers URC encoders having two, four and eight states, as shown in Figure 2.17. Following URC encoding, the bits $v_{\mathrm{u},k}$ and $v_{\mathrm{l},k}$ are concatenated to form the bit vectors $\mathbf{v}_{\mathrm{u}}$ and $\mathbf{v}_{\mathrm{l}}$, respectively. The bit vectors $\mathbf{v}_{\mathrm{u}}$ and $\mathbf{v}_{\mathrm{l}}$ are interleaved by a pair of interleavers, both having the same design π_3. The bits in the resultant vectors are paired up by a multiplexer to form the vector $\mathbf{w}$, which is QPSK modulated and transmitted over the channel using a natural mapping [93]. As we will show in Section 3.4, natural mapping facilitates improved error correction capability and/or reduced iterative decoding complexity, compared to Gray mapping which does not facilitate iterative operation. This results in an effective throughput given by $\eta = R^{\mathrm{o}}R^{\mathrm{i}}\log_2(M)$, where the use of the turbo code leads to $R^{\mathrm{i}} = 0.5$ and we have $M = 4$ for QPSK modulation.

3.2.2 Receiver

The receiver of the proposed scheme is shown in Figure 3.1. This figure shows that the URC decoders convert the *a priori* Logarithmic Likelihood Ratio (LLR) vectors $\tilde{\mathbf{v}}_{\mathrm{u}}^{\mathrm{a}}$,$\tilde{\mathbf{u}}_{\mathrm{u}}^{\mathrm{a}}$, $\tilde{\mathbf{v}}_{\mathrm{l}}^{\mathrm{a}}$, $\tilde{\mathbf{u}}_{\mathrm{l}}^{\mathrm{a}}$ into the extrinsic LLR vectors $\tilde{\mathbf{v}}_{\mathrm{u}}^{\mathrm{e}}$,$\tilde{\mathbf{u}}_{\mathrm{u}}^{\mathrm{e}}$, $\tilde{\mathbf{v}}_{\mathrm{l}}^{\mathrm{e}}$, $\tilde{\mathbf{u}}_{\mathrm{l}}^{\mathrm{e}}$. Likewise the trellis decoder and the QPSK demodulator convert the LLR vectors $\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{a}}$, $\tilde{\mathbf{w}}_{\mathrm{i}}^{\mathrm{a}}$ into the extrinsic LLR vectors $\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{e}}$, $\tilde{\mathbf{w}}_{\mathrm{i}}^{\mathrm{e}}$, respectively. The UEC and URC decoders all operate on the basis of the Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm, as detailed in Section 2.1.5.

The QPSK demodulator [94] receives symbols from the channel. It bi-directionally exchanges LLRs with the URC decoders, where the extrinsic LLR vector $\tilde{\mathbf{w}}_{\mathrm{i}}^{\mathrm{e}}$ is demultiplexed and deinterleaved through π_3^{-1}, before being provided as *a priori* LLR vectors $\tilde{\mathbf{v}}_{\mathrm{u}}^{\mathrm{a}}$ and $\tilde{\mathbf{v}}_{\mathrm{l}}^{\mathrm{a}}$ to the upper and lower URC decoders, respectively. Since the demodulator is iterative, it relies on natural mapping [93] to enhance error correction performance. The demodulator iterates with the upper and lower URC decoders, which exchange their extrinsic LLR vectors $\tilde{\mathbf{v}}_{\mathrm{u}}^{\mathrm{e}}$ and $\tilde{\mathbf{v}}_{\mathrm{l}}^{\mathrm{e}}$ with the demodulator through π_3 and a multiplexer, providing the demodulator with the *a priori* LLR vector $\tilde{\mathbf{w}}_{\mathrm{i}}^{\mathrm{a}}$.

Figure 3.1 shows that the *a priori* LLR vectors $\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{a}}$, $\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}}$ and $\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{a}}$ provided for each of the three decoders are obtained as the sum of the extrinsic LLR vectors most recently generated by the other two decoders, namely we have $\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{a}} = \tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}} + \tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{e}}$, $\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}} = \tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{e}} + \tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{e}}$ and $\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{a}} = \tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{e}} + \tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}}$. These LLR vectors comprise b number of LLRs, which pertain to the corresponding bits of $\mathbf{z}$. At the start of the iterative decoding process, the extrinsic LLR vectors $\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{e}}$, $\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}}$ and $\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{e}}$ are initialized with zero-valued LLRs. The interleavers π_1, π_2 and the deinterleavers π_1^{-1}, π_2^{-1} match with the interleavers in the transmitter, and are used for translating the LLR vectors $\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}}$, $\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{a}}$, $\tilde{\mathbf{u}}_{\mathrm{u}}^{\mathrm{e}}$, $\tilde{\mathbf{u}}_{\mathrm{l}}^{\mathrm{e}}$ into the vectors $\tilde{\mathbf{u}}_{\mathrm{u}}^{\mathrm{a}}$, $\tilde{\mathbf{u}}_{\mathrm{l}}^{\mathrm{a}}$, $\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}}$, $\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{e}}$, respectively.

As shown in [27] and Section 2.5.4, the number of states employed by the UEC trellis decoder can be selected independently of the number of states employed in the UEC trellis encoder. Increasing the number of states r employed by the UEC trellis decoder in this way has the benefit of improving its error correction capability, albeit this is achieved at the cost of increasing its complexity [6]. In Section 3.3.2, we will exploit this for dynamically adjusting r for the sake of striking an attractive trade-off between the UEC trellis decoder's complexity and its error correction capability.

During the iterative decoding process, the iterative operation of the URC1, URC2, UEC decoders and of the demodulator seen in Figure 3.1 may be performed using a wide variety of different decoder activation orderings. For the sake of conceptional simplicity, a fixed decoder activation order may be employed, in which the URC1, URC2, UEC decoders and the demodulator of Figure 3.1 are activated using a regular activation order repeated periodically, namely [demod, URC1, URC2, UEC, demod, URC1,...]. Alternatively, we may employ a dynamic decoder activation order, in which an on-line decision is made at each stage of the iterative decoding process, in order to adaptively and dynamically select which of the URC1, URC2, UEC decoders or whether the demodulator of Figure 3.1 should be activated next. This is explored in Section 3.3 using a novel 3D EXIT chart aided technique, in order to expedite iterative decoding convergence towards the Maximum Likelihood (ML) decoder's performance.

Following the achievement of iterative decoding convergence, the UEC trellis decoder of Figure 3.1 generates the vector of *a posteriori* LLRs $\tilde{\mathbf{y}}^{\mathrm{p}}$, which pertain to the corresponding unary-encoded bits in the vector $\mathbf{y}$. Finally, $\tilde{\mathbf{y}}^{\mathrm{p}}$ can be unary decoded in order to obtain the symbol vector $\hat{\mathbf{x}}$. Since each unary codeword contains only a single logical 1-valued bit, the number of 1-valued bits in the vector $\mathbf{y}$ is guaranteed to be equal to a, which is the number of symbols in each vector $\mathbf{x}$. As a result, the unary decoder exploits this property by setting the a highest LLR values in $\tilde{\mathbf{y}}$ to bit values of one, since these represent the bits most likely to have the value 1 in the vector $\mathbf{y}$. The remainder of the bits are set to zero. Finally, the unary decoder converts the bit vector to a symbol vector $\hat{\mathbf{x}}$, according to Table 2.6.

3.3 Adaptive decoder activation order algorithm

This section details the proposed extension to the adaptive iterative decoding algorithm of [27] for the sake of facilitating its contribution with schemes relying on additional serially concatenated blocks, such as the proposed concatenation of an iterative source and channel code with an iterative demodulator. This algorithm is best suited for scenarios which are complexity-limited, since it allows the receiver to adapt the decoder activation order to make best use of the limited resources. Section 3.3.1 begins by considering the EXIT functions of each decoder, which are used by the proposed algorithm for quantifying the benefit of activating the corresponding decoding block at a particular point in the iterative decoding process. Following this, Section 3.3.2 details the decision process used by the proposed adaptive algorithm. Section 3.3.3 quantifies the storage requirements of the algorithm, while Section 3.3.4 conceives a technique for eliminating the algorithm's complex logarithmic and exponential operations without a reducing its performance.

3.3.1 EXIT function analysis

EXIT functions [26], which were described in Section 2.4, rely on the MI of LLR vectors to quantify the quality of the extrinsic information output by a decoding block, as a function of the quality of its input *a priori* information, as well as the the channel Signal-to-Noise Ratio (SNR) if the decoding block performs demodulation, equalization or multi-user detection, for example.

The following sections describe the EXIT functions for the different components of the scheme shown in Figure 3.1, with the UEC EXIT function described in Section 3.3.1.1, the URC described in Section 3.3.1.2, and the combined demodulator and URC EXIT function described in Section 3.3.1.3. In common with previous work on the subject [27], all of the EXIT functions are defined in terms of $I(\tilde{\mathbf{z}})$ and $I(\tilde{\mathbf{w}})$, rather than $I(\tilde{\mathbf{u}})$ and $I(\tilde{\mathbf{v}})$.

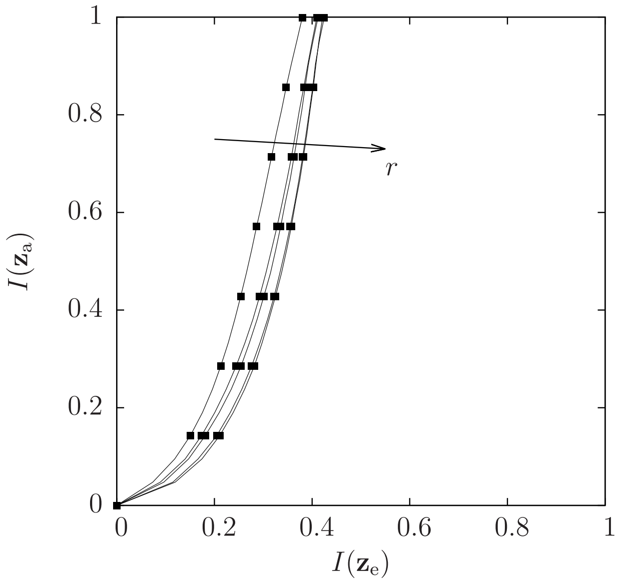


Figure 3.2: The inverted EXIT function $I(\tilde{\mathbf{z}}^\mathrm{e}_\mathrm{o}) = f_\mathrm{UEC}[I(\tilde{\mathbf{z}}^\mathrm{a}_\mathrm{o}), r]$ for the UEC decoder, with $r \in \{4, 6, 8, 10, 12\}$ states and $p_1 = 0.7$.

3.3.1.1 UEC decoder

The operation of the UEC decoder is only dependent on a single input vector of *a priori* LLRs $\tilde{\mathbf{z}}^\mathrm{a}_\mathrm{o}$, which is used for producing both the output extrinsic LLR vector $\tilde{\mathbf{z}}^\mathrm{e}_\mathrm{o}$ and the *a posteriori* LLR vector $\tilde{\mathbf{y}}$. The UEC EXIT functions may be expressed as $I(\tilde{\mathbf{z}}^\mathrm{e}_\mathrm{o}) = f_\mathrm{UEC}[I(\tilde{\mathbf{z}}^\mathrm{a}_\mathrm{o}), r]$, where $I(\tilde{\mathbf{z}})$ is the MI of the LLR vector $\tilde{\mathbf{z}}$, and r is the number of states in the UEC trellis decoder. The operation of the UEC decoder may be described by a series of 2D EXIT functions, with a separate EXIT function for each considered value of r. The 2D EXIT functions for the UEC decoder are shown in Figure 3.2, for $p_1 = 0.7$ and $r \in \{4, 6, 8, 10, 12\}$.

3.3.1.2 URC decoder

The URC decoder is dependent on the *a priori* LLR vector inputs $\tilde{\mathbf{w}}^\mathrm{a}$ and $\tilde{\mathbf{z}}^\mathrm{a}$, which it used for providing two extrinsic output LLR vectors $\tilde{\mathbf{w}}^\mathrm{e}$ and $\tilde{\mathbf{z}}^\mathrm{e}$, where the $_\mathrm{u}$ and $_\mathrm{l}$ subscripts refer specifically to the upper and lower URC decoders, respectively. Note however that it is only necessary for the MI of the $\tilde{\mathbf{z}}^\mathrm{e}$ output to be considered by the adaptive algorithm, as it will be described in Section 3.3.2. The EXIT function for the $\tilde{\mathbf{z}}^\mathrm{e}$ output of a URC decoder can therefore be expressed as a function depending on two input MI values, leading to a 3D EXIT function. More specifically, we have $I(\tilde{\mathbf{z}}^\mathrm{e}) = f_\mathrm{URC}[I(\tilde{\mathbf{z}}^\mathrm{a}), I(\tilde{\mathbf{w}}^\mathrm{a})]$, where $I(\tilde{\mathbf{z}}^\mathrm{a})$ and $I(\tilde{\mathbf{w}}^\mathrm{a})$ are the mutual information of $\tilde{\mathbf{z}}^\mathrm{a}$ and $\tilde{\mathbf{w}}^\mathrm{a}$, respectively.

3.3.1.3 Demodulator and URC Decoder

The iterative QPSK demodulator and one of the URC decoders can be considered as a single block for the purposes of the proposed algorithm. This is because the operation of the demodulator will always be followed by activating one of the URC decoders, so that the resultant information $\tilde{\mathbf{w}}_{\mathrm{i}}^{\mathrm{e}}$ gleaned from the demodulator can be propagated to the rest of the iterative decoder. This approach takes the demodulator into account, but allows an EXIT function to be produced that characterizes the expected MI improvement of the signal $\mathbf{z}^{\mathrm{e}}$. Since both the URC and UEC EXIT functions also characterize the improvement of $\mathbf{z}^{\mathrm{e}}$, this allows like-for-like comparisons to be made between the three different types of decoding blocks.

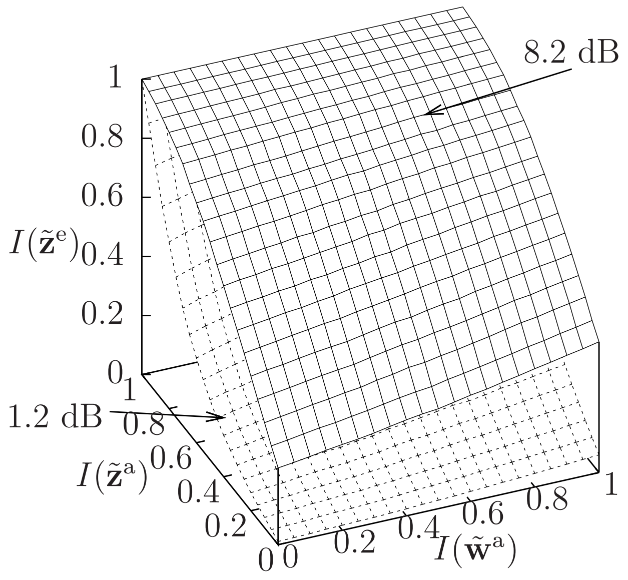


Figure 3.3: The EXIT function $I(\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}}) = f_{\mathrm{D-URC}}[I(\tilde{\mathbf{z}}^{\mathrm{a}}), I(\tilde{\mathbf{w}}^{\mathrm{a}}), E_{\mathrm{b}}/N_0]$ for E_{b}/N_0 = 1.2 dB and E_{b}/N_0 = 8.2 dB.

The joint demodulator and URC decoder EXIT function depends on the channel's E_{b}/N_0, as well as on the *a priori* LLR vector $\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}}$ or $\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{a}}$ provided for the URC decoder and on the *a priori* LLR vector $\tilde{\mathbf{w}}_{\mathrm{i}}^{\mathrm{a}}$ fed back to the demodulator. This can be expressed as $I(\tilde{\mathbf{z}}^{\mathrm{e}}) = f_{\mathrm{D-URC}}[I(\tilde{\mathbf{z}}^{\mathrm{a}}), I(\tilde{\mathbf{w}}_{\mathrm{i}}^{\mathrm{a}}), E_{\mathrm{b}}/N_0]$, which allows a series of 3D EXIT functions to be produced, with a specific 3D EXIT function for each E_{b}/N_0 value considered. The EXIT functions recorded for two E_{b}/N_0 values are shown in Figure 3.3 for the combination of natural mapping aided QPSK and a URC decoder having 8 states.

3.3.2 Adaptive activation algorithm

Upon starting to detect a new frame, the receiver first has to activate the iterative demodulator, followed by one of the URC blocks in order to disseminate the information received from the channel. Following this, it may decide from several options as to which decoding block to operate next. In general, the decoding activation options that can be selected based on Fig 3.1 are the upper URC decoder (URC1), the lower URC decoder (URC2), the demodulator followed by the upper URC decoder (URC1+demod), the demodulator followed by the lower URC decoder (URC2+demod), or the UEC decoder (UEC), where selecting the UEC presents the option of operating with different numbers of states r.

During the iterative decoding process, the benefit of each decoding option is quantified using the expected improvement to the quality of the extrinsic LLR vector that it provides. This is achieved by first measuring the MI of the most-recently generated values of the *a priori* LLR vectors $\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{a}}$, $\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}}$, $\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{a}}$, $\tilde{\mathbf{w}}_{\mathrm{u}}^{\mathrm{a}}$, and $\tilde{\mathbf{w}}_{\mathrm{l}}^{\mathrm{a}}$, using the technique described in Section 3.3.4. These MIs may be used as inputs to the EXIT functions of Section 3.3.1, which may be generated off-line and stored in Look-Up Tables (LUTs), as will be described in Section 3.3.3. These predict the MI of the extrinsic LLR vector that would be generated by each decoding option This can be compared to the MI of the currently available extrinsic LLR vector that would be replaced to make an estimation of the potential MI improvement offered by selecting this decoding option. Note that additions are used for obtaining the LLR vectors $\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{a}} = \tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}} + \tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{e}}$, $\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}} = \tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{e}} + \tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{e}}$ and $\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{a}} = \tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}} + \tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{e}}$. The MI of the *a priori* LLR vectors $\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{a}}$, $\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}}$ and $\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{a}}$ can be estimated without tentatively adding the extrinsic LLR vectors $\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{e}}$, $\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}}$ and $\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{e}}$. Instead, we can use the J-function proposed in [95], which takes as its input two MI values, and outputs the joint MI of the two inputs. This allows us to use $I(\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{a}}) = J[I(\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}}), I(\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{e}})]$ as an input to $I(\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{e}}) = f_{\mathrm{UEC}}[I(\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{a}}), r]$ in order to quantify the potential benefit of activating the UEC decoder. Likewise, $I(\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}}) = J[I(\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{e}}), I(\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{e}})]$ and $I(\tilde{\mathbf{w}}_{\mathrm{u}}^{\mathrm{a}})$ can be used as inputs to $I(\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}}) = f_{\mathrm{URC}}[I(\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}}), I(\tilde{\mathbf{w}}_{\mathrm{u}}^{\mathrm{a}})]$, for predicting the potential benefit of operating the URC1 decoder. Similarly, $I(\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}}) = J[I(\tilde{\mathbf{z}}_{\mathrm{l}}^{\mathrm{e}}), I(\tilde{\mathbf{z}}_{\mathrm{o}}^{\mathrm{e}})]$, $I(\tilde{\mathbf{w}}_{\mathrm{i}}^{\mathrm{e}})$ and the channel SNR can be used as inputs to $I(\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{e}}) = f_{\mathrm{D-URC}}[I(\tilde{\mathbf{z}}_{\mathrm{u}}^{\mathrm{a}}), I(\tilde{\mathbf{w}}_{\mathrm{i}}^{\mathrm{e}}), E_{\mathrm{b}}/N_0]$, for predicting the expected benefit of operating the demodulator followed by URC1. By subtracting the MI of the currently avaliable extrinsic LLR vector from the expected MI of the vector that would replace it, we obtain the expected MI improvement for each decoding option. By selecting the next decoding option with consideration of its expected MI improvement, it allows us to give priority to those blocks which have not been activated recently.

The algorithm for selecting which of the available decoding options to perform next (UEC, URC1, URC2, URC1+demod, URC2+demod) also quantifies of the cost of each the options. More specifically, we consider using the number of Add-Compare-Select (ACS) [96] operations needed per bit to perform each option. The number of ACS operations for the URC and UEC decoders are approximately proportional to the number of states in the trellis. A 4-state and an 8-state URC require 112 and 223 ACS operations per bit of $\mathbf{z}$, respectively. Meanwhile an $r = \{2, 4, 6, 8, 10\}$ state UEC requires $\{50, 108, 166, 224, 282\}$ ACS operations per bit of $\mathbf{z}$, respectively. The demodulator has a much lower complexity, requiring only 11 ACS operations per bit. This results in 123 and 234 ACS operations per bit of $\mathbf{z}$, when the demodulator is combined with a 4-state and 8-state URC, respectively. Upon dividing the expected MI improvement by the number of ACS operations needed per bit for each of the decoding options, we arrive at a score for each decoding option. The algorithm then picks the specific option with the highest score as the block which should be operated next.

3.3.3 Storage requirements

In this section, we consider the memory required for the storage of the EXIT function LUTs, which are required for the adaptive iterative decoding algorithm of Section 3.3.2 to function. As described in Section 3.3.2, three sets of EXIT function LUTs are required: one set for the UEC code $I(\tilde{\mathbf{z}}_{\text{o}}^{\text{e}}) = f_{\text{UEC}}[I(\tilde{\mathbf{z}}_{\text{o}}^{\text{a}}), r]$, with one 2D EXIT function for each of the $G_r = 5$ number of states $r \in \{2,4,6,8,10\}$ considered; one 3D EXIT function LUT for the URC codes $I(\tilde{\mathbf{z}}^{\text{e}}) = f_{\text{URC}}[I(\tilde{\mathbf{z}}^{\text{a}}), I(\tilde{\mathbf{w}}^{\text{a}})]$; and one set of 3D EXIT functions $I(\tilde{\mathbf{z}}^{\text{e}}) = f_{\text{D-URC}}[I(\tilde{\mathbf{z}}^{\text{a}}), I(\tilde{\mathbf{w}}^{\text{a}}), E_{\text{b}}/N_0]$ for the combined demodulator and URC decoders, where there is one 3D EXIT function for each of the 24 E_{b}/N_0 values considered. These G_{S}=24 different E_{b}/N_0 values were selected in the range 1.2 dB to 8.2 dB, where EXIT functions for lower E_{b}/N_0 values are not stored, since successful decoding is not possible in this case, while EXIT functions for higher E_{b}/N_0 values are not needed, since the decoding complexity is low in this case, hence reducing the benefit of the adaptive algorithm.

For all the EXIT functions, G_{I}=21 different *a priori* MI values $\{0, 0.05, 0.1, ..., 1\}$ were selected for each input. The results of Section 3.4.2 compare the iterative decoding performance, when quantization levels of $B \in \{4,6,8\}$ bits are used for representing each value in the EXIT function LUT, resulting in overall memory requirements of 5.5 kB, 8.25 kB and 11 kB, respectively. This storage requirement S can be generalized to $S[\text{bits}] = B(G_{\text{I}}^2(G_{\text{S}}+1)+G_{\text{I}}G_r)$. As a comparison, the combined memory requirement for three different 6144 bit random interleavers is 30 kB, demonstrating that the memory requirement of the proposed algorithm is not excessive, when compared to the original memory requirement of the scheme in Figure 3.1.

3.3.4 MI measurement

The adaptive iterative decoding algorithm of Section 3.3.2 relies on measuring the MI of the LLR vectors provided by the various blocks of the decoder. Using the averaging method [26], the MI of an LLR vector may be is estimated without any knowledge of the true bit values. The MI is calculated as a function of the entropies that are implied by the N LLRs in the vector [26], according to $I(\tilde{\mathbf{z}}) = \frac{1}{N}\sum_{k=1}^{N} 1 - H_{\text{b}}(\tilde{z}_k)$, where $H_{\text{b}}(\tilde{z}_k)$ is the binary entropy function.

The binary entropy function $H_\mathrm{b}(\tilde{z}_k)$ is comprised of several log and exp functions, which imposes a high implementation complexity. As a result, it is desirable to approximate this function. Figure 3.4 shows that the function $1 - H_\mathrm{b}(\tilde{z}_k)$ resembles an inverted Gaussian distribution. Motivated by this, we propose to approximate it using a linear approximation. As shown in Figure 3.4, this approximation is characterized by a straight line, which is mirrored for positive and negative LLR values, and that is clipped such that it has a maximum MI value of 1, and the minimum MI value of 0. This results in a linear piece-wise approximation of 5 segments, and can be written as $I(\tilde{z}_k) = \max(\min(m|\tilde{z}_k| + c, 1), 0)$, where m and c represent the gradient and intercept point of the line. To choose the optimum values for m and c, LLRs were generated having a range of target MI values in the range of 0 to 1. Following this, the approximate MI measurement function was fitted to these LLRs, producing an estimate for the MI of each group of LLRs. The slope m and intercept c values were chosen for the approximation as those, which produced the minimum aggregate error between the approximate MI values and the exact MI values. In this case, $m = 0.24$ and $c = -0.06$ were found to be optimum. To reduce the number of calculations needed, instead of scaling and shifting the MI of each LLR before averaging, we can transform the operation, so that the clipped LLRs are averaged, before this average is scaled by m and shifted by c to get the final MI of the vector. These techniques result in negligible complexity for the MI measurement compared to the operation of the decoding blocks.

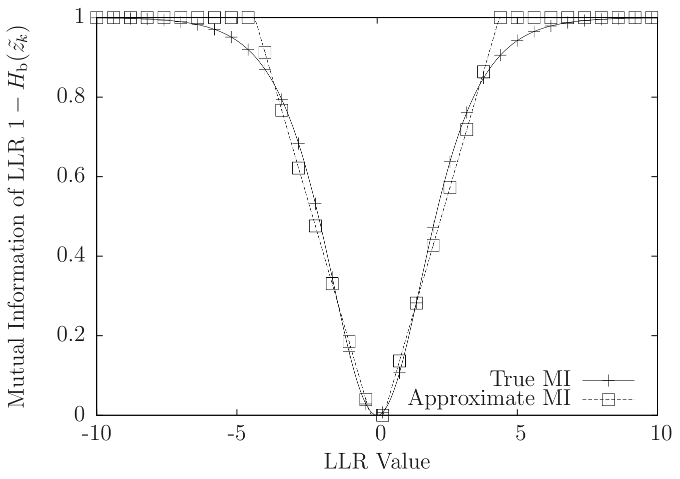


Figure 3.4: The relationship between the true MI and the approximate MI

3.4 Results

In this section, the error correcting performance of the proposed scheme using the adaptive iterative decoding algorithm is compared to that of various benchmarkers. This section will first describe the nature of these benchmarkers in Section 3.4.1, before presenting the results in Section 3.4.2.

3.4.1 Considered schemes

Table 3.1: Comparison between the characteristics of the different benchmarkers

Scheme	r	R_o	A_o	R_i	η	E_b/N_0 (dB) capacity bound	E_b/N_0 (dB) area bound	E_b/N_0 (dB) tunnel bound			
								Turbo4	Turbo4-QPSK	Turbo8	Turbo8-QPSK
UEC-Turbo	2	0.762	0.934	0.500	0.762	0.84	2.48	3.1	2.8	2.7	3.1
	4		0.808				1.27	2.3	1.6	1.8	2.0
	6		0.783				1.04	1.8	1.3	1.3	1.8
	8		0.774				0.95	1.8	1.3	1.2	1.7

In this section, the proposed schemes of Section 3.2 are contrasted to a number of benchmarkers. A summary of the benchmarkers is shown in Table 3.1. For each scheme, we will consider both a 4-state and an 8-state URC design for the upper and lower URC decoder producing turbo decoders, which are referred to as Turbo4 and Turbo8 from here on. Each of the schemes can also be operated with or without the iterative demodulator, where activation of the iterative demodulator is denoted with the -QPSK suffix. In the case when the QPSK demodulator is not operated iteratively, Gray mapping is employed since it provides superior performance to natural mapping in this case [93]. Table 3.1 shows the E_b/N_0 capacity bound, which is the theoretical limit for reliable communication of the given effective throughput η, the E_b/N_0 area bound which is the limit at which the area beneath the EXIT functions match each other, and finally the E_b/N_0 tunnel bound, which is the threshold where the EXIT charts exhibit an open tunnel. These bounds indicate successively stricter limits for the E_b/N_0, where a low SER becomes possible for each scheme. Since all of the schemes have the same effective throughput of $\eta = 0.762$, our comparisons between them are fair in terms of transmission energy, delay and bandwidth.

3.4.1.1 UEC-Turbo(-QPSK)

This is the proposed scheme of Figure 3.1, which may be operated with or without the aid of the adaptive iterative decoding algorithm advocated. The tunnel bound of Table 3.1 suggests that the UEC-Turbo8 scheme of our previous work [27] would marginally outperform UEC-Turbo4-QPSK by 0.1 dB, but only in the absence of an iterative decoding complexity limit. Although the iterative demodulator of the proposed scheme imposes only a modest complexity, it actually facilitates a significant overall complexity reduction since it allows the less complex 4-state URC to be used, suggesting that it will outperform the UEC-Turbo8 scheme in the presence of an iterative decoding complexity limit.

3.4.1.2 EG-URC-Turbo(-QPSK)

This scheme is a classic SSCC benchmarker, where the unary encoder of Figure 3.1 is replaced with an Elias Gamma (EG) code for source coding. Furthermore, the UEC trellis code of Figure 3.1 is replaced with an accumulator, which is an $r = 2$ state URC encoder. This is required for converting the vector of non-equiprobable bits $\mathbf{y}$ into a vector of equiprobable bits $\mathbf{z}$ [27]. During decoding, each of the three URC decoders are operated in turn. In this scheme the outer URC decoder is provided with *a priori* information pertaining to the bit value probabilities in $\mathbf{y}$, which allows some of the residual redundancy to be exploited for error correction.

3.4.1.3 EG-Turbo(-QPSK)

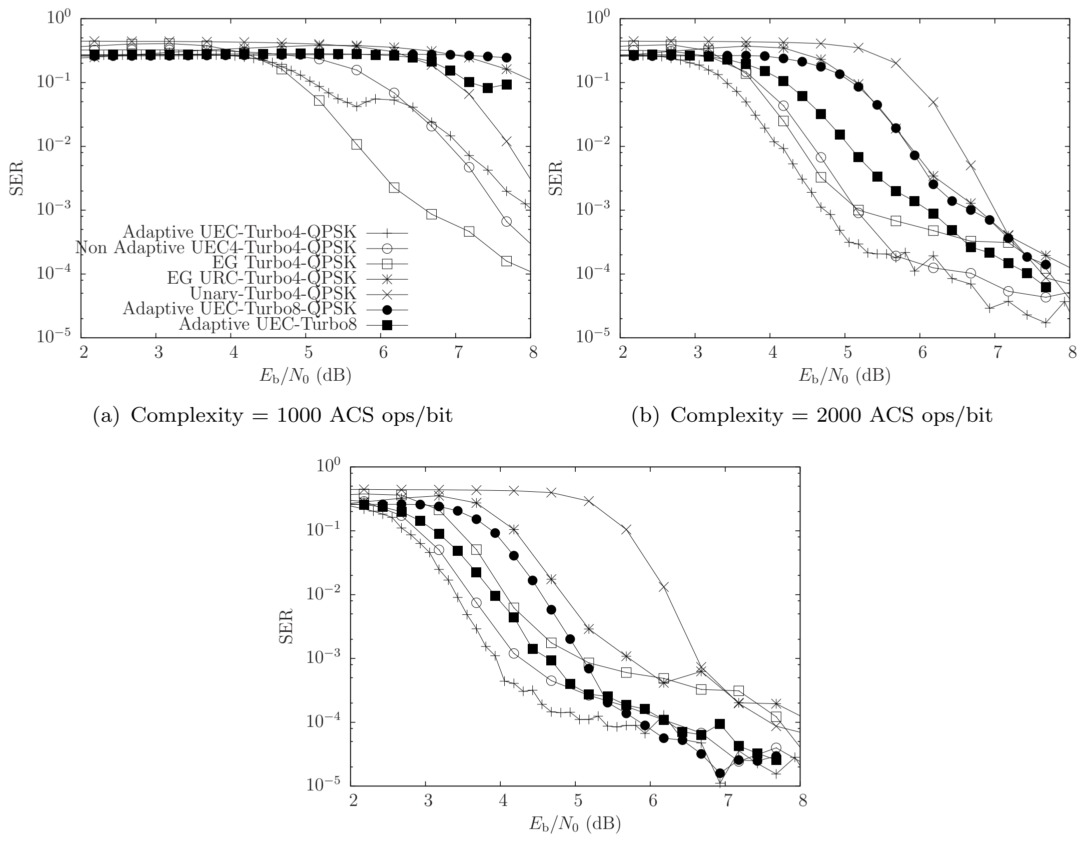
This SSCC benchmarker replaces the UEC code of Figure 3.1 with an EG code. This results in a variation of the EG-URC-Turbo scheme, which omits the URC code. This scheme suffers from a significant capacity loss due to the non-equiprobable bits in $\mathbf{z}$ [27]. However, this scheme has a low complexity due to the reduced number of iterative blocks, which are operated in the decoder.

3.4.1.4 Unary-Turbo(-QPSK)

This is another SSCC benchmarker, which has non-equiprobable bits at $\mathbf{y}$, and therefore also suffers a from capacity loss. Compared to the proposed scheme of Figure 3.1, the UEC code has been replaced by a unary code. It is therefore similar to the EG-Turbo scheme, but uses a unary encoder instead of a EG encoder.

3.4.2 SER performance

In this section, a limited complexity-receiver is considered, where there is a limited amount of computational resource. Figure 3.5 provides SER plots for each of the schemes chosen and for a range of complexity limits, where complexity is quantified in terms of ACS operations per bit of $\mathbf{z}$. Results are provided for the transmission of frames comprising $a = 650$ symbols over an uncorrelated narrowband Rayleigh fading channel. The complexity limits imposed are 1000, 2000 and 3000 operations per bit of $\mathbf{z}$, which allow the relative performance of the various schemes to be compared for a range of available decoder resources. If the iterative scheme of Figure 3.1 operates each of the blocks successively in turn, these complexity limits allow approximately 3, 6 and 9 iterations to be completed, when 4 states are used for the UEC and URC decoders. Note that the results of Figure 3.5 show a reasonably high error floor, although this is exacerbated by the use of SER plots rather than Bit Error Ratio (BER). The high error floor may be attributed to the poor distance properties of the non-systematic turbo code, and may be improved by using a systematic design, such as that of the Long Term Evolution (LTE) standard.



(a) Complexity = 1000 ACS ops/bit

(b) Complexity = 2000 ACS ops/bit

(c) Complexity = 3000 ACS ops/bit

Figure 3.5: SER performance for the adaptive scheme and several benchmarkers under different complexity limits. Frame length = 650 symbols

3.4.2.1 Effect of the iterative demodulator

As shown in Figure 3.5, when the scheme of Figure 3.1 uses 4-state URCs and the iterative demodulator (UEC-Turbo4-QPSK), it performs better than the UEC-Turbo8 scheme of [27], which uses 8-state URCs without iterative demodulation. Although the EXIT chart analysis of Section 3.4.1 suggests that the ultimate performance of the UEC-Turbo8 scheme would be 0.1 dB better, the UEC-Turbo4-QPSK arrangement is seen to be 2 dB superior, when a maximum complexity of $C = 2000$ ops/bit of $\mathbf{z}$ is imposed. This may be attributed to the fact that the 4-state URC decoder imposes about half the complexity of the 8-state URC decoder, while the added complexity of the iterative demodulator is minimal in comparison.

The proposed UEC-Turbo4-QPSK scheme performs particularly well under low complexity limits, since its advantage of reduced complexity is diminished, as the complexity

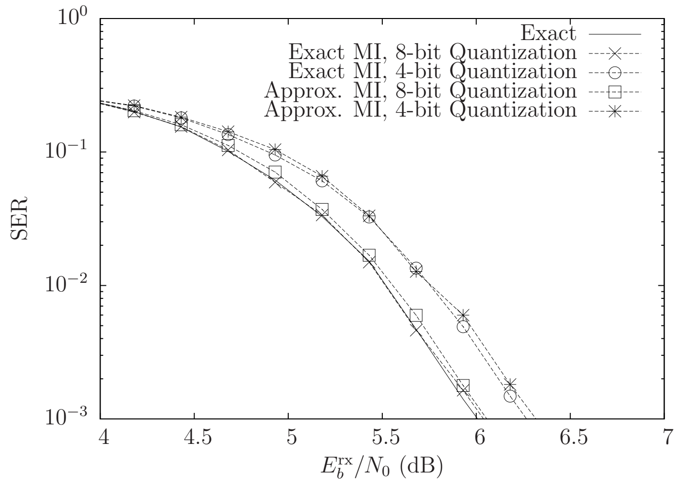


Figure 3.6: SER performance of the adaptive algorithm for different approximations of UEC-Turbo8-QPSK and $C = 2000$ ACS ops/bit. Frame length = 650 symbols

limit is increased. Owing to this, the gap between the UEC-Turbo8 and UEC-Turbo4-QPSK schemes reduces to 1.2 dB for a complexity limit of $C = 3000$ ACS ops/bit of $\mathbf{z}$.

3.4.2.2 Low complexity schemes vs. high performance schemes

Figure 3.5 also compares the proposed adaptive UEC-Turbo4-QPSK scheme to other benchmarkers. Since the proposed adaptive algorithm allows for a reduced decoding complexity, it has a performance advantage over the set of high-performance benchmarkers. However, under the low complexity limit of $C = 1000$ ACS per bit of $\mathbf{z}$, the less complex benchmarkers, such as the EG-Turbo scheme, exhibit a 1.5 dB gain over the more complex near-capacity schemes. However, when the complexity limit is increased to $C = 2000$ ACS per bit of $\mathbf{z}$, these low complexity schemes suffer a performance loss, while the proposed adaptive UEC-Turbo4-QPSK scheme offers a modest, but non-negligible performance gain of 0.25 dB, without any increase in transmission energy, delay, bandwidth or decoder complexity.

3.4.2.3 Effects of approximation

Figure 3.6 characterizes the performance erosion imposed upon the proposed adaptive UEC-Turbo4-QPSK scheme, when EXIT chart LUT quantization is used, as well as when approximate MI measurements are used. It can be seen in Figure 3.6 that 6-bit and 8-bit-quantization imposes only a negligible degradation on the SER performance, while 4-bit-quantization reduces the gain of the proposed scheme by 0.2 dB. The approximation invoked for measuring the MI also has a negligible impact on the proposed scheme, hence this is recommended for hardware implementation.

3.5 Conclusions

This chapter has demonstrated that the adaptive iterative decoding algorithm of [27] can be further developed for reducing the complexity of an iterative decoder with a four stage concatenation, including an iterative demodulator. This iterative demodulator also allows the employment of a URC code with fewer states, resulting in an overall complexity, whilst beneficially increasing the performance of the scheme in complexity-limited scenarios. The proposed adaptive algorithm allows low SERs to be achieved over a range of complexity limits, while the benchmarkers either favour low complexity limits or high complexity limits exclusively. This chapter also proposes an approximation to the MI measurement used in the adaptive algorithm, facilitating its hardware implementation with a low overhead.

Chapter 4

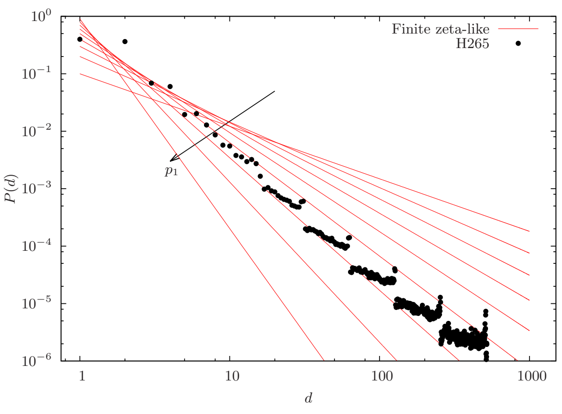
Improving the applicability of UEC using RiceEC and ExpGEC coding

4.1 Introduction

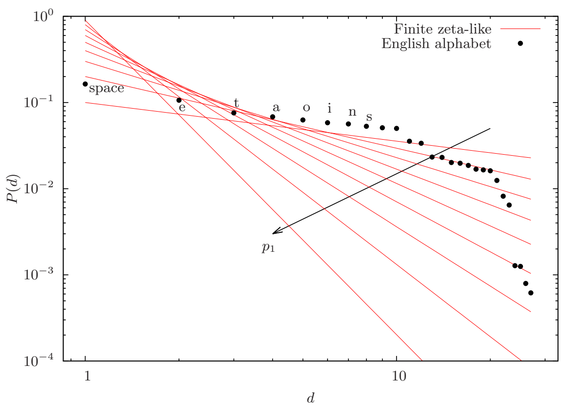
As discussed in Chapter 3, the encoding of multimedia information such as video and audio typically results in symbol values that are selected from large or infinite alphabets. For example, the H.264 and H.265 video encoders represent source video information using transform coefficients and motion vectors [97], which correspond to a large alphabet of symbol values in the range spanning from 1 to around $L = 1000$, as shown in Figure 4.1a[1]. It may be observed that these symbols obey Zipf's law [90], with low-valued symbols occurring frequently and high valued symbols occurring infrequently, as shown in Figure 4.1a. Owing to this, the occurrence of these symbol values may be modelled by a zeta probability distribution, as previously shown in Section 2.5.

As described in Section 2.5, both source coding and channel coding is required, in order to facilitate the reliable and bandwidth-efficient transmission of multimedia information. Near-capacity operation may be achieved using Separate Source and Channel Coding (SSCC), where a near-capacity channel code is combined with a separate near-entropy source code. Unfortunately, these source codes are often impractical since they have excessive complexity, or require knowledge of the probability of occurrence of each symbol value at both the transmitter and receiver. The alternative approach is to use universal codes, such as the Elias Gamma code [82] and Exponential Golomb (ExpG) [89]

[1]With thanks to Tao Wang for providing the video symbol data.
This chapter is partially based on the following publication.
M. F. Brejza, T. Wang, W. Zhang, R. G. Maunder, B. Al-Hashimi and L. Hanzo (2016). Exponential Golomb and Rice Error Correction Codes for Generalized Near-Capacity Joint Source and Channel Coding. *IEEE Access, 4*, 7154–7175.



(a) Distribution of H.265 symbols



(b) Distribution of English letters and the space character

Figure 4.1: The probability distribution of (a) letters of the English alphabet and the space character, ordered according to descending probability of occurrence, and (b) symbols output from a H.265 encoder. Also shown is the finite zeta-like distribution of (4.1) for $p_1 = \{0.1, ..., 0.9\}$ and with $L = 27$ and $L = 1000$, respectively.

code, which can encode source symbols from large or infinite sets. However, typically non-negligible redundancy remains in the encoded bitstreams produced by these source codes, which leads to capacity loss. This motivated the use of the Unary Error Correction (UEC) code in Chapter 3, which is a Joint Source and Channel Coding (JSCC) scheme [3], designed for exploiting the residual redundancy that remains after source encoding for enhancing the attainable error correction capability.

However, the UEC code is only practical for a subset of zeta distributions, and this subset does not include those which best model the H.264 and H.265 symbol distributions. More specifically, since a UEC is not a universal code [22], for some zeta distributions the UEC code results in a bit vector having an infinite average length. Motivated by this deficiency of the UEC code, this chapter extends the previously proposed class of Elias Gamma Error Correction (EGEC) codes [28]. Since the EGEC code is a universal code created from the class of Elias Gamma (EG) source codes, it produces bit vectors having a finite length for any monotonic probability distribution, including all zeta distributions. Despite its finite-length nature, it still produces long bit vectors for some zeta distributions, hence potentially resulting in low coding rates. To circumvent this problem, excessive puncturing may be required, if a high coding rate is desired, which potentially leads to increased complexity relative to codes having higher original coding rates, as well as to a degraded error correction performance [98]. As a result, the EGEC code family was only characterized for a limited set of zeta distributions in [28]. These limitations of the UEC and EGEC codes imply that neither of them exhibits general applicability, hence indicating that further generalization is required.

Against this background, this chapter extends and generalizes the UEC code family

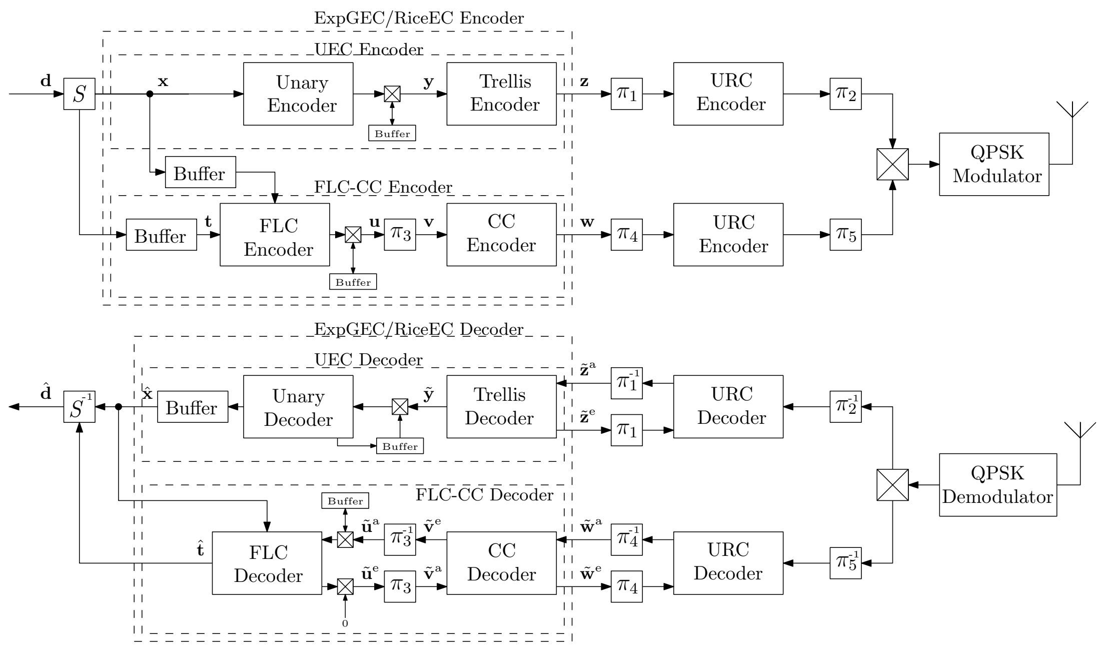


Figure 4.2: The proposed ExpGEC/RiceEC schemes, which comprise a Unary Error Correction (UEC) part and a Fixed Length Code-Convolutional Code (FLC-CC) part. Here, the buffers facilitate operation on the basis of a stream of source symbols, while maintaining fixed-length designs for the interleavers π_1–π_5.

of [6] and the EGEC code of [28], so that they become attractive for encoding symbols produced by *all* zeta distributions, hence granting them general applicability. More specifically, this chapter generalises the EGEC code family by extending its schematic to produce the class of Exponential Golomb Error Correction (ExpGEC) codes, as shown in Figure 4.2. The EGEC code is also extended to produce the Rice Error Correction (RiceEC) code family, which represents a generalization of the UEC code. This chapter shows that the proposed ExpGEC and RiceEC codes offer a better error correction performance over a wider range of source symbol distributions than that of its benchmarkers. In particular, the full range of zeta distributions are considered for the first time, allowing better represention over a wider range of source symbol distributions. This includes the distribution of the English alphabet, which has a cardinality of $L = 27$ when including the space character, as characterized in Figure 4.1b. Furthermore, the specific distribution of symbols produced by the H.265 video encoder is also considered, where previous work only considered the H.264 video encoder. Motivated by this, this chapter critically appraises the infinite-cardinality zeta distribution of previous work in the specific scenario of finite-cardinality zeta-like distributions, where symbols can assume values in the range $\{1, ..., L\}$. This chapter investigates how a specific choice of the parameter L affects the source symbol distribution, as well as how the parameters of the proposed codes may be best selected for optimizing their error correction performance. For the first time, the proposed codes are compared to the classic Variable

Length Error-Correction (VLEC) code [5, 92, 99, 100], although this benchmarker only has a practical complexity for low values of L. Additionally, this chapter enhances the previous EGEC code of [28] and the UEC used in Section 2.5 by improving the practicality of the proposed codes. More specifically, the EGEC scheme of [28] requires five different interleavers, each having different lengths and therefore designs, which change from frame to frame. Similarly, the UEC scheme discussed in previous chapters requires at least one interleaver, which may also change length from frame to frame. Not only is this arrangement challenging to implement, but the overall error correction performance is dominated by the worse-case performance, associated with the shortest inverleaver lengths. By contrast, the proposed ExpGEC and RiceEC schemes are designed for encoding continuous streams of symbols, while maintaining constant interleaver-lengths in every frame, hence significantly improving the associated practicality and performance. Furthermore, this chapter proves that these modifications guarantee synchronization between the transmitter and receiver.

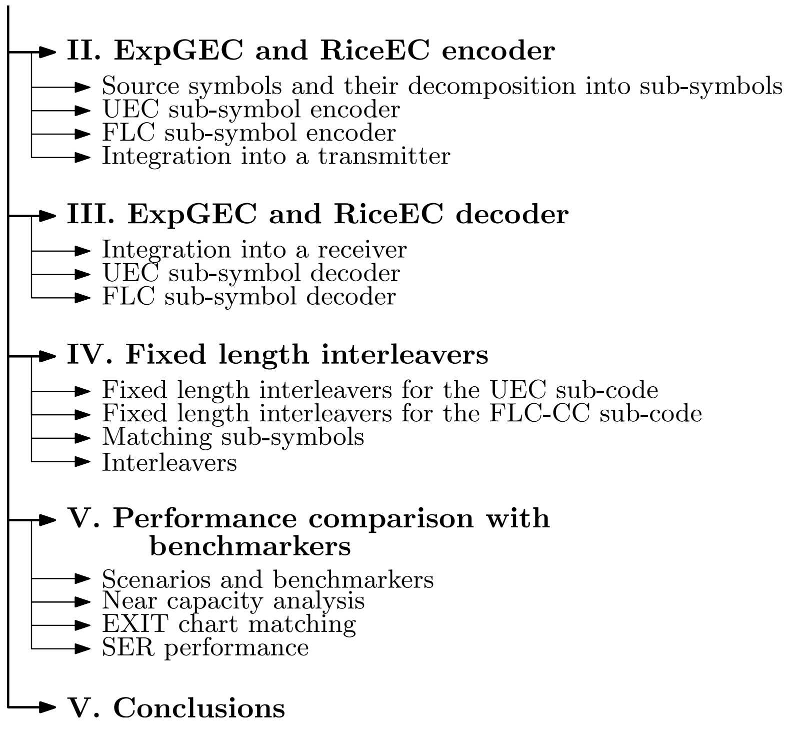

II. ExpGEC and RiceEC encoder
Source symbols and their decomposition into sub-symbols
UEC sub-symbol encoder
FLC sub-symbol encoder
Integration into a transmitter
III. ExpGEC and RiceEC decoder
Integration into a receiver
UEC sub-symbol decoder
FLC sub-symbol decoder
IV. Fixed length interleavers
Fixed length interleavers for the UEC sub-code
Fixed length interleavers for the FLC-CC sub-code
Matching sub-symbols
Interleavers
V. Performance comparison with benchmarkers
Scenarios and benchmarkers
Near capacity analysis
EXIT chart matching
SER performance
V. Conclusions

Figure 4.3: The structure of this chapter

As shown in Figure 4.3, this chapter commences by detailing the operation and characteristics of the proposed ExpGEC and RiceEC codes, where Sections 4.2 and 4.3 consider the operation of the encoders and decoders, respectively. Section 4.4 describes the arrangement proposed for processing streams of symbols, while maintaining fixed interleaver lengths. Section 4.5 characterizes the error correction performance of the proposed codes, demonstrating that they are superior to the best of several benchmakers for a wide variety of source distributions, including the full range of zeta distributions, as well as the H.265 and English alphabet distributions. Finally, this chapter concludes in Section 4.6.

4.2 ExpGEC and RiceEC encoder

This section introduces the ExpGEC and RiceEC encoders, which are shown in Figure 4.2. Firstly, Section 4.2.1 commences by describing how the proposed ExpGEC and RiceEC encoders decompose the source symbols into sub-symbols. Following this, Sections 4.2.2 and 4.2.3 describe how the sub-symbols are encoded by the UEC encoder and the FLC-CC encoder, respectively. Finally, Section 4.2.4 describes how the ExpGEC and RiceEC encoders may be integrated into a transmitter, as shown in Figure 4.2.

4.2.1 Source symbols and their decomposition into sub-symbols

As portrayed in Figure 4.2, the proposed ExpGEC and RiceEC encoders operate on the basis of a stream of source symbols $\mathbf{d} = [d_i]$, which may be modelled as a realization of a stream of Independent and Identically Distributed (IID) Random Variables (RVs) $\mathbf{D} = [D_i]$. In contrast to the infinite symbol alphabet of the previous work [28], here we consider source symbols which are randomly selected from a finite symbol set, as described in Section 4.1. This introduces an additional parameter, namely the cardinality L of the source symbol set. More specifically, each RV D_i adopts a value in the set $\{1, 2, 3, ..., L\}$, according to a particular source symbol distribution. Since Figure 4.1 shows that the H.265 symbols and the letters of the English alphabet obey Zipf's law, we consider a finite-cardinality zeta-like source symbol distribution, where the probability $\Pr(D_i = d) = P(d)$ is given by

$$P(d) = \frac{d^{-s}}{\sum_{\hat{d}=1}^{L} \hat{d}^{-s}} = \frac{d^{-s}}{\bar{\zeta}(s, L)}, \tag{4.1}$$

with $\bar{\zeta}(s, L) = \sum_{\hat{d}=1}^{L} \hat{d}^{-s}$ being the finite Riemann zeta-like function. Here, the variable $s > 1$ is related to the probability of an RV D_i adopting the value 1 according to $p_1 = \Pr(D_i = 1) = 1/\bar{\zeta}(s, L)$, which parameterises the finite zeta-like distribution. The entropy of the source symbols is given by

$$H_D = \sum_{d=1}^{L} P(d) \log_2 \frac{1}{P(d)}. \tag{4.2}$$

Table 4.1 shows the first 18 unary, Rice, EG and ExpG codewords, demonstrating how each of the corresponding source encoders maps each symbol d_i from the stream $\mathbf{d}$ to the codeword $\text{Unary}(d_i)$, $\text{Rice}(d_i)$, $\text{EG}(d_i)$ or $\text{ExpG}(d_i)$, respectively. Here, the Rice code is parametrized by $M_{\text{Rice}} \in \{1, 2, 4, 8, 16, ...\}$, where $M_{\text{Rice}} = 1$ gives a special case identical

Table 4.1: The decomposition of symbols d_i into sub-symbols x_i and t_i for the ExpG and Rice codes. The sub-codewords $\mathbf{y}_i$ and $\mathbf{u}_i$ relate to the sub-symbols x_i and t_i according to $\mathbf{y}_i = \text{Unary}(x_i)$ and $\mathbf{u}_i = \text{FLC}(t_i)$, respectively. The concatenation of $\mathbf{y}_i$ and $\mathbf{u}_i$ provides the codewords ExpG(d_i) or Rice(d_i), while the dashed line shows how these codewords are decomposed into the sub-codewords.

	Rice(d_i) M_{Rice}=1, Unary(d_i)				Rice(d_i) M_{Rice}=2				Rice(d_i) M_{Rice}=4				Rice(d_i) M_{Rice}=8				ExpG(d_i) k_{ExpG}=0, EG(d_i)				ExpG(d_i) k_{ExpG}=1				ExpG(d_i) k_{ExpG}=2			
d_i	x_i	$\mathbf{y}_i$	$\mathbf{u}_i$	t_i	x_i	$\mathbf{y}_i$	$\mathbf{u}_i$	t_i	x_i	$\mathbf{y}_i$	$\mathbf{u}_i$	t_i	x_i	$\mathbf{y}_i$	$\mathbf{u}_i$	t_i	x_i	$\mathbf{y}_i$	$\mathbf{u}_i$	t_i	x_i	$\mathbf{y}_i$	$\mathbf{u}_i$	t_i	x_i	$\mathbf{y}_i$	$\mathbf{u}_i$	t_i
1	1	1		0	1	1	0	0	1	1	00	0	1	1	000	0	1	1		0	1	1	0	0	1	1	00	0
2	2	01		0	1	1	1	1	1	1	01	1	1	1	001	1	2	01	0	0	1	1	1	1	1	1	01	1
3	3	001		0	2	01	0	0	1	1	10	2	1	1	010	2	2	01	1	1	2	01	00	0	1	1	10	2
4	4	0001		0	2	01	1	1	1	1	11	3	1	1	011	3	3	001	00	0	2	01	01	1	1	1	11	3
5	5	00001		0	3	001	0	0	2	01	00	0	1	1	100	4	3	001	01	1	2	01	10	2	2	01	000	0
6	6	000001		0	3	001	1	1	2	01	01	1	1	1	101	5	3	001	10	2	2	01	11	3	2	01	001	1
7	⋮	⋮		⋮	4	0001	0	0	2	01	10	2	1	1	110	6	3	001	11	3	3	001	000	0	2	01	010	2
8					4	0001	1	1	2	01	11	3	1	1	111	7	4	0001	000	0	3	001	001	1	2	01	011	3
9					5	00001	0	0	3	001	00	0	2	01	000	0	4	0001	001	1	3	001	010	2	2	01	100	4
10					5	00001	1	1	3	001	01	1	2	01	001	1	4	0001	010	2	3	001	011	3	2	01	101	5
11					6	000001	0	0	3	001	10	2	2	01	010	2	4	0001	011	3	3	001	100	4	2	01	110	6
12					6	000001	1	1	3	001	11	3	2	01	011	3	4	0001	100	4	3	001	101	5	2	01	111	7
13					⋮	⋮	⋮	⋮	4	0001	00	0	2	01	100	4	4	0001	101	5	3	001	110	6	3	001	0000	0
14									4	0001	01	1	2	01	101	5	4	0001	110	6	3	001	111	7	3	001	0001	1
15									4	0001	10	2	2	01	110	6	4	0001	111	7	4	0001	0000	0	3	001	0010	2
16									4	0001	11	3	2	01	111	7	5	00001	0000	0	4	0001	0001	1	3	001	0011	3
17									5	00001	00	0	3	001	000	0	5	00001	0001	1	4	0001	0010	2	3	001	0100	4
18									5	00001	01	1	3	001	001	1	5	00001	0010	2	4	0001	0011	3	3	001	0101	5

to the unary code. Likewise, the ExpG code is parametrized by $k_{\text{ExpG}} \in \{0, 1, 2, 3, 4, ...\}$, where $k_{\text{ExpG}} = 0$ gives a special case identical to the EG code.

The length of an ExpG codeword $\text{ExpG}(d_i)$ may be expressed as $l_{\text{ExpG}}(d_i) = 2\lfloor \log_2(d_i + 2^{k_{\text{ExpG}}} - 1)\rfloor + 1 - k_{\text{ExpG}}$, while the length of a Rice codeword $\text{Rice}(d_i)$ is $l_{\text{Rice}}(d_i) = \lceil d_i/M_{\text{Rice}}\rceil + \log_2(M_{\text{Rice}})$. Note that in the special case of the unary code where $M_{\text{Rice}} = 1$, the length of each corresponding codeword becomes equal to the value of the encoded symbol $l_{\text{Unary}}(d_i) = d_i$. When the source symbols obey the finite zeta-like distribution, the average length $l = \sum_{d=1}^{L} P(d) \cdot l(d)$ of the encoded ExpG, Rice, and unary codewords is given by

$$l_{\text{ExpG}} = 1 - k_{\text{ExpG}} + \frac{1}{\bar{\zeta}(s, L)} \sum_{d=1}^{L} d^{-s} \lfloor \log_2(d + 2^{k_{\text{ExpG}}} - 1)\rfloor, \tag{4.3}$$

$$l_{\text{Rice}} = \log_2(M_{\text{Rice}}) + \frac{1}{\bar{\zeta}(s, L)} \sum_{d=1}^{L} d^{-s} \lceil d/M_{\text{Rice}}\rceil, \tag{4.4}$$

$$l_{\text{Unary}} = \frac{\bar{\zeta}(s-1, L)}{\bar{\zeta}(s, L)}. \tag{4.5}$$

As shown using the dashed lines in Table 4.1, each ExpG and Rice codeword can be viewed as a concatenation of a unary prefix $\mathbf{y}_i = \text{Unary}(x_i)$ and Fixed Length Code (FLC) suffix $\mathbf{u}_i = \text{FLC}(t_i)$, where x_i and t_i are the sub-symbols derived from a particular symbol d_i. For each symbol d_i, the value of the sub-symbol x_i is given by

$$\text{ExpG: } x(d) = \lfloor \log_2(d + 2^{k_{\text{ExpG}}} - 1)\rfloor + 1 - k_{\text{ExpG}}, \tag{4.6}$$

$$\text{Rice: } x(d) = \left\lceil \frac{d}{M_{\text{Rice}}} \right\rceil. \tag{4.7}$$

Here, the sub-codeword $\mathbf{y}_i = \text{Unary}(x_i)$ comprises $(x_i - 1)$ zeros, followed by a single logical one-valued bit. Likewise, the value of the sub-symbol t_i is given by

$$\text{ExpG: } t(d) = d - 2^{\lfloor \log_2(d + 2^{k_{\text{ExpG}}} - 1)\rfloor} + 2^{k_{\text{ExpG}}} - 1, \tag{4.8}$$

$$\text{Rice: } t(d) = \text{mod}((d-1), M_{\text{Rice}}), \tag{4.9}$$

where $\text{mod}(a, b)$ provides the modulo of a when divided by b. Here, the sub-codeword $\mathbf{u}_i = \text{FLC}(t_i)$ is obtained by representing the sub-symbol t_i in a binary form having a particular length. In the case of the ExpG code, the length of this suffix $\mathbf{u}_i = \text{FLC}(t_i)$ depends on the corresponding value of x_i, where $l_{\text{FLC}}(t) = x + k_{\text{ExpG}} - 1$. For the Rice code, this suffix $\mathbf{u}_i = \text{FLC}(t_i)$ has a fixed length $l_{\text{FLC}}(t) = \log_2(M_{\text{Rice}})$, which is independent of the value of x_i or t_i. Given the expressions (4.6) – (4.9), we may express

d_i as functions of x_i and t_i, according to

$$\text{ExpG: } d(x,t) = 2^{(x+k_{\text{ExpG}}-1)} - 2^{k_{\text{ExpG}}} + 1 + t, \tag{4.10}$$

$$\text{Rice: } d(x,t) = M_{\text{Rice}}(x-1) + t + 1. \tag{4.11}$$

Motivated by the observation that each ExpG and Rice codeword comprises a unary prefix and an FLC suffix, the ExpGEC/RiceEC encoder of Figure 4.2 employs a splitter S. This uses (4.6)-(4.9) to decompose each symbol d_i in the stream $\mathbf{d} = [d_i]$ into the sub-symbols x_i and t_i, which are concatenated to form the streams $\mathbf{x} = [x_i]$ and $\mathbf{t} = [t_i]$. For example, given the symbol vector $\mathbf{d} = [6, 1, 9, 3, 12, 4, 1, 2]$, the $k_{\text{ExpG}} = 1$ ExpGEC splitter yields the sub-symbol vectors $\mathbf{x} = [2, 1, 3, 2, 3, 2, 1, 1]$ and $\mathbf{t} = [3, 0, 2, 0, 5, 1, 0, 1]$, while the $M_{\text{Rice}} = 4$ RiceEC splitter would yield the sub-symbol vectors $\mathbf{x} = [2, 1, 3, 1, 3, 1, 1, 1]$ and $\mathbf{t} = [1, 0, 0, 2, 3, 3, 0, 1]$. Following this, each sub-symbol x_i in the stream $\mathbf{x}$ is encoded by the UEC encoder of Figure 4.2, which is based on the unary code, as described in Section 4.2.2. Meanwhile, each sub-symbol t_i in the stream $\mathbf{t}$ is encoded by the FLC-CC encoder, which is based on the FLC code, as described in Section 4.2.3.

4.2.2 UEC sub-symbol encoder

As shown in Figure 4.2, the input of the UEC encoder, described in Section 2.5, is provided by the stream of sub-symbols $\mathbf{x} = [x_i]$. This stream may be modelled as a realization of a stream of IID RVs $\mathbf{X} = [X_i]$. In the scenario where the RV D_i obeys the finite zeta-like distribution of (4.1), the probability $\Pr(X_i = x) = P(x)$ is given by

$$\text{ExpG: } P(x) = \frac{1}{\bar{\zeta}(s,L)} \sum_{d=2^{x+k_{\text{ExpG}}-1}-2^{k_{\text{ExpG}}}+1}^{\min(2^{x+k_{\text{ExpG}}}-2^{k_{\text{ExpG}}},L)} d^{-s}, \tag{4.12}$$

$$\text{Rice: } P(x) = \frac{1}{\bar{\zeta}(s,L)} \sum_{d=M_{\text{Rice}}(x-1)+1}^{\min(Mx_i,L)} d^{-s}, \tag{4.13}$$

while the entropy of each RV X_i is given by

$$\text{ExpG: } H_X = \sum_{x=1}^{\substack{\lfloor \log_2(L+2^{k_{\text{ExpG}}}-1)\rfloor \\ +1-k_{\text{ExpG}}}} P(x)\log_2\left(\frac{1}{P(x)}\right), \tag{4.14}$$

$$\text{Rice: } H_X = \sum_{x=1}^{\left\lceil \frac{L}{M_{\text{Rice}}} \right\rceil} P(x)\log_2\left(\frac{1}{P(x)}\right). \tag{4.15}$$

Each sub-symbol x_i in the stream $\mathbf{x}$ is encoded by the unary encoder, which outputs the corresponding x_i-bit unary sub-codeword $\mathbf{y}_i = \text{Unary}(x_i)$, according to Table 4.1. The average length l_1 of these unary sub-codewords is given by

$$l_1 = \sum_{d=1}^{L} P(d) \cdot x(d), \tag{4.16}$$

$$\text{ExpG: } l_1 = 1 - k_{\text{ExpG}} + \frac{1}{\bar{\zeta}(s,L)} \sum_{d=1}^{L} \lfloor \log_2(d + 2^{k_{\text{ExpG}}} - 1) \rfloor \cdot d^{-s}, \tag{4.17}$$

$$\text{Rice: } l_1 = \frac{1}{\bar{\zeta}(s,L)} \sum_{d=1}^{L} \left\lceil \frac{d}{M_{\text{Rice}}} \right\rceil \cdot d^{-s}. \tag{4.18}$$

In Section 2.5 and Chapter 3, the unary code was employed for encoding the source symbols d_i directly, but this produces long average codeword lengths when p_1 is low, according to (4.5). However, in the scheme of Figure 4.2, the unary encoder is used for encoding the sub-symbols x_i instead. Since the sub-symbol probability distributions $P(x)$ of (4.12) and (4.13) are skewed towards the most likely symbol value $x = 1$, the UEC code may be used for their encoding without suffering from an excessive average codeword lengths, when p_1 is low. As shown in Figure 4.2, the codewords in the stream produced by the unary encoder are concatenated and partitioned into a succession of bit-vectors $\mathbf{y} = [y_j]_{j=1}^{b}$, having a fixed length of b bits, as it will be described in Section 4.4.1. For example, given the sequence $\mathbf{x} = [2, 1, 3, 2, 3, 2, 1, 1]$ comprising 8 sub-symbols, the unary encoder produces the sequence $\mathbf{y} = 011001010010111$ comprising $b = 15$ bits.

The bit vector $\mathbf{y}$ is entered into the trellis encoder of Figure 4.2, which operates on the basis of the UEC trellis of Figure 4.4. Here, UEC trellises comprising only $r_1 = 4$ states are adopted, since the previous work [28] showed that this is sufficient for avoiding any significant capacity loss despite its low complexity, as it will be characterized for the codes proposed in Section 4.5.2. However, the option to extend the trellis to more states remains open [6], facilitating the elimination of even more capacity loss, at the cost of increasing the associated complexity. For each successive input bit y_j in $\mathbf{y} = [y_j]_{j=1}^{b}$, the trellis transition emerges from a previous state $m_{j-1} \in \{1, 2, 3, ..., r_1\}$ to a next state $m_j \in \{1, 2, 3, ..., r_1\}$, according to the trellis of Figure 4.4. The path pursued through the UEC trellis when encoding the bit vector $\mathbf{y}$ may be represented as $\mathbf{m} = [m_j]_{j=0}^{b}$, comprising $b + 1$ state values, where m_0 and m_b are the start and end states of the trellis. Each zero-valued bit y_i in the unary-encoded bit vector triggers transitions to states towards the outside of the trellis. By contrast, each one-valued bit causes the trellis to transition back to one of the central states. Note that each unary codeword comprises a sequence of zero-valued bits which is terminated by a single one-valued bit. The trellis structure of Figure 4.4 exploits this for maintaining synchronization between the unary-encoded symbols and the path through trellis. For example, given the unary-encoded bit vector comprising the $b = 15$ bits of $\mathbf{y} = 011001010010111$, the path through the trellis

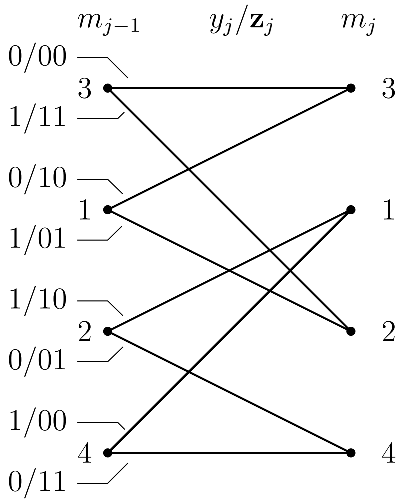


Figure 4.4: The UEC trellis utilizing $r_1 = 4$ states, $n_1 = 2$ output bits and the codewords $\mathbb{C} = [01; 11]$

can be formulated as the vector $\mathbf{m} = [1, 3, 2, 1, 3, 3, 2, 4, 1, 3, 3, 2, 4, 1, 2, 1]$ of $b + 1 = 16$-states. The path $\mathbf{m}$ may be modelled as a realization of a vector of RVs $\mathbf{M} = [M_j]_{j=0}^{b}$, where the probability of each state being selected $\Pr(M_j = m|M_{j-1} = m') = P(m|m')$. The knowledge of these conditional transition probabilities $P(m|m')$ may be exploited to aid the receiver, as described in Section 4.3.2.

Depending on the path selected through the UEC trellis, each of the bits in the unary-encoded bit vector $\mathbf{y}$ is encoded using a n_1-bit codeword $\mathbf{z}_j$, which are concatenated to form the bn_1-bit UEC-encoded vector $\mathbf{z} = [z_k]_{k=1}^{bn_1}$. The specific codeword selected for each transition is shown in Figure 4.4 for the case where $r_1 = 4$-states are used. In the example-path traversing through the trellis $\mathbf{m}$ provided above and the trellis of Figure 4.4, the UEC-encoded bit vector $\mathbf{z} = 101110100011010010001101000110$ comprising $bn_1 = 30$ bits is produced, when using the $r_1 = 4$-state, $n_1 = 2$-bit trellis of Figure 4.4. Since the top and bottom halves of the UEC trellis use complementary codewords, the UEC-encoded bits of $\mathbf{z}$ are guaranteed to have equiprobable binary values. Due to this equiprobablity, the average coding rate of the UEC encoder is given by

$$R_1^{\text{o}} = \frac{H_X}{l_1 n_1}. \tag{4.19}$$

Here the superscript ‘o’ is used to indicate this coding rate relates to the outer code of a serial concatenation, namely the UEC code of Figure 4.2.

4.2.3 FLC-CC sub-symbol encoder

As shown in Figure 4.2, the FLC-CC sub-symbol encoder is used for encoding the sub-symbol stream $\mathbf{t} = [t_i]$. Here, the FLC encoder of Figure 4.2 represents each sub-symbol

t_i using a codeword $\mathbf{u}_i$, which is given by the fixed-point binary representation of t_i, having a particular length that may depend on the particular value of the corresponding sub-symbol x_i, as described in Section 4.2.2. Motivated by this, the sub-symbol stream $\mathbf{t}$ may be modelled as a realization of a stream of RVs $\mathbf{T} = [T_i]$, where each RV T_i is dependent on the corresponding RV X_i. More specifically, in the case where the RV D_i obeys the finite zeta-like distribution of (4.1), the joint probability $\Pr(T_i = t \cap X_i = x) = P(t \cap x)$ may be obtained according to

$$\text{ExpG: } P(t \cap x) = \frac{1}{\bar{\zeta}(s,L)} \left(2^{x+k_{\text{ExpG}}-1} - 2^{k_{\text{ExpG}}} + 1 + t\right)^{-s}, \tag{4.20}$$

$$\text{Rice: } P(t \cap x) = \frac{1}{\bar{\zeta}(s,L)} \left(M_{\text{Rice}}(x-1) + t + 1\right)^{-s}, \tag{4.21}$$

where $0 \leq t < 2^{x-1+k_{\text{ExpG}}}$ for the case of the ExpG and $0 \leq t < M_{\text{Rice}}$ for the case of the RiceEC code.

These joint probabilities may be used for obtaining expressions for the corresponding conditional probabilities, which may be exploited in the receiver to aid FLC-CC decoding. In the case of the finite zeta-like distribution, the conditional probability $\Pr(T_i = t | X_i = x) = P(t|x)$ is given by

$$P(t|x) = \frac{P(t \cap x)}{P(x)}, \tag{4.22}$$

$$\text{ExpG: } P(t|x) = \frac{\left(2^{x+k_{\text{ExpG}}-1} - 2^{k_{\text{ExpG}}} + 1 + t\right)^{-s}}{\sum_{d=2^{x+k_{\text{ExpG}}-1}-2^{k_{\text{ExpG}}}+1}^{\min(2^{x+k_{\text{ExpG}}}-2^{k_{\text{ExpG}}},L)} d^{-s}}, \tag{4.23}$$

$$\text{Rice: } P(t|x) = \frac{\left(M_{\text{Rice}}(x-1) + t + 1\right)^{-s}}{\sum_{d=M_{\text{Rice}}(x-1)+1}^{\min(Mx,L)} d^{-s}}, \tag{4.24}$$

where $0 \leq t < 2^{x-1+k_{\text{ExpG}}}$ for the case of the ExpGEC code and $0 \leq t < M_{\text{Rice}}$ for the RiceEC code. Finally, the conditional entropy of the RV T_i is given by

$$H_{T|X} = \sum_{d=1}^{L} P(t \cap x) \cdot \log_2 \left(\frac{1}{P(t|x)}\right). \tag{4.25}$$

As described in Section 4.2.2, each FLC codeword $\mathbf{u}_i$ has the length $x_i + k_{\text{ExpG}} - 1$ in the case of the ExpGEC code, where x_i is the corresponding sub-symbol in the stream $\mathbf{x}$. Owing to this, the FLC-CC encoder requires knowledge of the symbol stream $\mathbf{x}$, as shown in Figure 4.2. In the case of the RiceEC code, the length of the FLC codewords is fixed at $\log_2(M_{\text{Rice}})$ and so the knowledge of $\mathbf{x}$ is not required during FLC-CC encoding in this case. When the sub-symbols of $\mathbf{d}$ obey the finite zeta-like distribution of (4.1), the average length of the FLC codewords l_2 is given by

$$\text{ExpG: } l_2 = \frac{1}{\bar{\zeta}(s,L)} \sum_{d=1}^{L} \lfloor \log_2(d + 2^{k_{\text{ExpG}}} - 1) \rfloor \cdot d^{-s}, \tag{4.26}$$

$$\text{Rice: } l_2 = \log_2(M_{\text{Rice}}). \tag{4.27}$$

Following FLC encoding, the resultant FLC codewords are then concatenated together to form a bit-stream, which is then partitioned into bit-vectors $\mathbf{u} = [u_j]_{j=1}^{c}$, comprising c number of bits, as it will be detailed in Section 4.4.2. In the example of using the $k_{\text{ExpG}} = 1$ ExpGEC code to encode the sub-symbol vectors $\mathbf{x} = [2, 1, 3, 2, 3, 2, 1, 1]$ and $\mathbf{t} = [3, 0, 2, 0, 5, 1, 0, 1]$, Table 4.1 may be used for producing the $c = 15$-bit FLC-encoded vector $\mathbf{u} = 110010001010101$. In the case of the $M_{\text{Rice}} = 4$ RiceEC code, the FLC encoding of the sub-symbol vector $\mathbf{t} = [1, 0, 0, 2, 3, 3, 0, 1]$ produces the $c = 16$-bit vector $\mathbf{u} = 0100001011110001$.

As shown in Figure 4.2, the FLC-encoded bit vector $\mathbf{u}$ is interleaved in the block π_3, in order to provide the bit vector $\mathbf{v} = [v_j]_{j=1}^{c}$. This is then encoded by a r_2-state, n_2-bit non-systematic recursive CC encoder having a design selected from [6, Table II], in order to obtain the bit vector $\mathbf{w} = [w_k]_{k=1}^{cn_2}$. The CC codes of [6, Table II] were shown in the previous work of [28] to mitigate capacity loss. More explicitly, while the bits of $\mathbf{u}$ are not guaranteed to have equiprobable values, the non-systematic recursive nature of these CCs guarantees equiprobable values for the bits of $\mathbf{w}$, which mitigates capacity loss [6]. For example, the interleaver $\pi_3 = [6, 14, 11, 10, 3, 1, 5, 8, 13, 16, 15, 2, 4, 7, 12, 9]$ may be employed for transforming the example $c = 16$-bit vector $\mathbf{u}$ provided above into the $c = 16$-bit vector $\mathbf{v} = 0011000001010111$, where $v_j = u_{\pi_3(j)}$. When the $r_2 = 4$-state, $n_2 = 2$-bit Convolutional Code (CC) encoder of [6, Table II] is employed, this bit vector $\mathbf{v}$ is encoded into the $cn_2 = 32$-bit FLC-CC-encoded bit vector $\mathbf{w} = 00001101010000000011100001110110$. The octally represented generator polynomials invoked for this CC encoder are [4,7], while the octal feedback polynomial is 6. Finally, the average coding rate of the FLC-CC encoder is given by

$$R_2^{\text{o}} = \frac{H_{T|X}}{l_2 n_2}. \tag{4.28}$$

4.2.4 Integration into a transmitter

The RiceEC and ExpGEC encoders may be integrated into a transmitter by serially concatenating them with an inner code. In the scheme of Figure 4.2, the bit vectors $\mathbf{z}$ and $\mathbf{w}$ provided by the UEC and FLC-CC encoders are interleaved in the blocks π_1 and π_4, before being encoded by 2-state Unity Rate Convolutional (URC) encoders, which behave as accumulators. As discussed in Section 4.3.1, these URC codes will facilitate iterative decoding in the receiver, enabling near capacity operation [101].

Following URC encoding, the pair of resultant bit vectors are interleaved and optionally punctured or doped in the blocks π_2 and π_5 of Figure 4.2, before they are multiplexed and QPSK modulated onto the channel using Gray mapping. Here, puncturing or doping may be employed for achieving a particular effective target throughput η for the scheme. As we will discuss further in Section 4.5.3, puncturing and doping may also be employed to provide unequal error protection for the UEC and FLC-CC parts of the schemes, in order to facilitate operation at lower channel SNRs. Puncturing is achieved in the blocks π_2 and π_5 of Figure 4.2 and by removing the appropriate number of bits from the end of the interleaved bit vector. By contrast, doping is achieved in the blocks π_2 or π_5 by duplicating the appropriate number of bits from the end of the interleaved bit vector and concatenating them. The puncturing and doping rates of π_2 and π_5 are represented by R_1^{i} and R_2^{i} respectively, which quantifies their ratio of input to output bits. Here, a value of $R^{\text{i}} > 1$ represents puncturing, where $R^{\text{i}} < 1$ represents doping. For example, $R^{\text{i}} = 0.75$ implies that one third of the original bits have been duplicated, while $R^{\text{i}} = 1.25$ implies that one fifth of the original bits have been discarded. Here, the superscript i indicates relevance to the inner code. The transmitter's overall effective throughput in bits per source symbol is given by

$$\eta = \frac{H_D \log_2(M_{\text{mod}})}{l_1 n_1 / R_1^{\text{i}} + l_2 n_2 / R_2^{\text{i}}}, \tag{4.29}$$

where M_{mod} is the number of constellation points used by the modulator, which is $M_{\text{mod}} = 4$ in the case of QPSK.

4.3 ExpGEC and RiceEC decoder

In this section, the operation of the ExpGEC and RiceEC decoders of Figure 4.2 is detailed. Section 4.3.1 discusses the integration of the UEC and FLC-CC sub-symbol decoders of the ExpGEC and RiceEC decoders into the receiver of Figure 4.2. Following this, the operation of the UEC sub-symbol decoder is described in Section 4.3.2, while the operation of the FLC-CC sub-symbol decoder is described in Section 4.3.3.

4.3.1 Integration into a receiver

In the receiver of Figure 4.2, the QPSK demodulator converts each received QPSK symbol into a pair of Logarithmic Likelihood Ratios (LLRs), which pertain to the bits at the transmitter. These LLRs convey the likelihood of the corresponding bits being 1-valued or 0-valued. More specifically, each LLR is defined by LLR $= \ln \frac{Pr(\text{bit}=1)}{Pr(\text{bit}=0)}$, where a large positive valued LLR represents a high confidence that the bit has the value of logical one, while a large negative-valued LLR represents a high confidence that the bit is logical zero-valued. The LLRs are then demultiplexed and entered into the blocks π_2^{-1} and π_5^{-1}

of Figure 4.2. These blocks undertake de-puncturing or de-doping as appropriate. De-puncturing is achieved by replacing each of the bits removed during puncturing with a zero-valued LLR, which reflects the absence of any knowledge about the value of the corresponding bit. By contrast, de-doping is achieved by removing the LLRs pertaining to the replicas of the duplicated bits and summing them into the LLRs pertaining to the corresponding duplicated bits. Following de-puncturing or de-doping the resultant LLRs are deinterleaved and forwarded to the URC decoders, which iteratively exchange their vectors of LLRs with the UEC decoder or the FLC-CC decoder, as appropriate. More specifically, the UEC trellis decoder and first URC decoder perform iterative decoding by exchanging the LLR vectors $\tilde{\mathbf{z}}^{\mathrm{a}}$ and $\tilde{\mathbf{z}}^{\mathrm{e}}$, which pertain to the bits in $\mathbf{z}$, as shown in Figure 4.2. Likewise, the CC decoder and second URC decoder perform iterative decoding by exchanging the LLR vectors $\tilde{\mathbf{w}}^{\mathrm{a}}$ and $\tilde{\mathbf{w}}^{\mathrm{e}}$, which pertain to the bits of $\mathbf{w}$. The UEC trellis decoder, CC decoder and URC decoder of Figure 4.2 invoke the Logarithmic Bahl-Cocke-Jelinek-Raviv (Log-BCJR) algorithm [17] for converting the input *a priori* LLR vector into the extrinsic LLR output vector, as may be characterized by their EXtrinsic Information Transfer (EXIT) functions, exemplified in Figure 4.5.

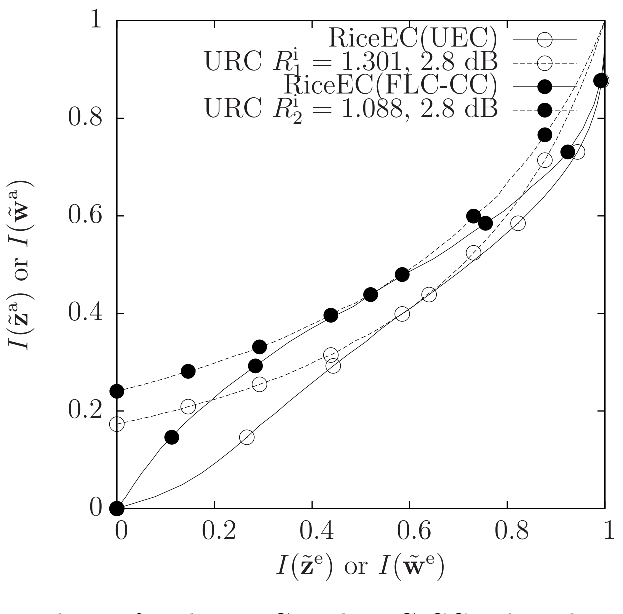


Figure 4.5: EXIT charts for the UEC and FLC-CC sub-codes of the proposed RiceEC scheme, when the symbols of $\mathbf{d}$ obey a finite zeta-like distribution having the parameters $p_1 = 0.2$ and $L = 1000$. The RiceEC scheme adopts the parameters $M_{\mathrm{Rice}} = 32$, $n_1 = 2$, $n_2 = 2$ and $\eta = 0.9$, for the case of using QPSK modulation over an uncorrelated narrowband Rayleigh fading channel. Here, the unequal error protection of the UEC and FLC-CC sub-codes relies on different puncturing rates R_1^{i} and R_2^{i}, in order to ensure that the corresponding EXIT charts open at the same E_{b}/N_0 value of 2.8 dB.

Before initiating the first decoding iteration, the LLR vectors $\tilde{\mathbf{w}}^{\mathrm{e}}$ and $\tilde{\mathbf{z}}^{\mathrm{e}}$ are populated with zero-valued LLRs. The URC decoders then invoke the Log-BCJR algorithm for converting the LLR vectors provided by the demodulator into extrinsic LLR vectors that can be used for iterative decoding. The deinterleavers π_1^{-1} and π_4^{-1} of Figure 4.2 convert these extrinsic LLR vectors into the *a priori* LLR vectors $\tilde{\mathbf{z}}^{\mathrm{a}} = [\tilde{z}_k^{\mathrm{a}}]_{k=1}^{bn_1}$ and $\tilde{\mathbf{w}}^{\mathrm{a}} = [\tilde{w}_k^{\mathrm{a}}]_{k=1}^{cn_2}$, respectively. The UEC decoder and FLC-CC decoder then operate as described in Sections 4.3.2 and 4.3.3, in order to generate the extrinsic LLR vectors $\tilde{\mathbf{z}}^{\mathrm{e}} = [\tilde{z}_k^{\mathrm{e}}]_{k=1}^{bn_1}$ and $\tilde{\mathbf{w}}^{\mathrm{e}} = [\tilde{w}_k^{\mathrm{e}}]_{k=1}^{cn_2}$. The interleavers π_1 and π_4 of Figure 4.2 convert these extrinsic LLR vectors into *a priori* LLR vectors that can be provided to the URC decoders. The iterations between the decoders continue until a fixed complexity limit has been reached or until the error-free decoding of $\hat{\mathbf{x}}$ and $\hat{\mathbf{t}}$ have been achieved, which may be detected using a Cyclic Redundancy Check (CRC) in practice. Note that the iterative UEC decoding must be completed before the iterative FLC-CC decoding can begin, since the latter takes the output $\hat{\mathbf{x}}$ of the former as its input, as will be discussed in Section 4.3.3. Finally, the symbols $\hat{\mathbf{d}}$ are recovered using the de-splitter S^{-1}, which operates according to Equations (4.10) and (4.11).

4.3.2 UEC sub-symbol decoder

In this section, we briefly describe the operation of the UEC sub-symbol decoder of Figure 4.2, which was previously described in Section 2.5.4. Here, the trellis decoder invokes the Log-BCJR algorithm for the trellis of Figure 4.4 in order to convert the vector of *a priori* LLRs $\tilde{\mathbf{z}}^{\mathrm{a}}$ into the vector of *a posteriori* LLRs $\tilde{\mathbf{y}}$, as well as the vector of extrinsic LLRs $\tilde{\mathbf{z}}^{\mathrm{e}}$. This extrinsic LLR vector $\tilde{\mathbf{z}}^{\mathrm{e}}$ is exchanged with the URC decoder in order to facilitate iterative decoding, as described in Section 4.3.1. The performance of the UEC trellis decoder may be enhanced by exploiting the knowledge of the conditional transition probabilities $P(m_j|m_{j-1})$ of the trellis, as discussed in Section 2.5.4. More specifically, the decoder requires the knowledge of the average unary codeword length l_1 and of the first $r_1/2-1$ values of $P(x)$ in order to compute the transition probabilities $P(m_j|m_{j-1})$ according to [6, Eqn. (9)]. These conditional transition probabilities $P(m_j|m_{j-1})$ may be exploited during the computation of the *a priori* transition probabilities γ of the Log-BCJR algorithm, as described in Section 2.1.5.

The transformation of $\tilde{\mathbf{z}}^{\mathrm{a}}$ into $\tilde{\mathbf{z}}^{\mathbf{e}}$ may be characterized by the UEC trellis decoder's inverted EXIT function [26], as shown in Figure 4.5. Note that for both the ExpGEC and RiceEC codes, the area A_1^{o} beneath the inverted UEC EXIT function may be closely approximated by [6, 102]

$$\begin{aligned}A_1^{\mathrm{o}} = &\frac{1}{l_1 n_1} \sum_{x=1}^{r_1/2-1} \big(H[P(x)]\big) + \frac{2}{l_1 n_1} H \left[1 - \sum_{x=1}^{r_1/2-1} P(x)\right] \\ &+ \frac{1}{l_1 n_1} H \left[l_1 - R_1/2 + \sum_{x=1}^{r_1/2-1} P(x)(r_1/2 - x)\right] \\ &- \frac{1}{l_1 n_1} H \left[l_1 + 1 - r_1/2 + \sum_{x=1}^{r_1/2-1} P(x)(r_1/2 - 1 - x)\right]. \end{aligned} \tag{4.30}$$

Once a sufficiently high number of iterations have been completed between the UEC trellis decoder and the URC decoder, the trellis decoder outputs the *a posteriori* LLR vector $\tilde{\mathbf{y}} = [\tilde{y}_j]_{j=1}^{b}$, ready for unary decoding, as shown in Figure 4.2. Note that since each unary codeword contains only a single logical one-valued bit, there are guaranteed to be a number of one-valued bits in the unary-encoded bit vector $\mathbf{y}$. As a result, the unary decoder sets the a highest LLR values in $\tilde{\mathbf{y}}$ to be one-valued bits, and the remaining $(b - a)$ LLRs are set to be zero-valued bits. Note that in practice the value of a may be reliably conveyed to the receiver using a small amount of side information. Finally, the unary decoder then converts the hard decision bit vector $\tilde{\mathbf{y}}$ into sub-symbols $\tilde{\mathbf{x}}$ according to Table 4.1.

4.3.3 FLC sub-symbol decoder

This section describes the FLC-CC decoder, which iteratively exchanges LLRs with the corresponding URC decoder of Figure 4.2. More specifically, the CC decoder invokes the Log-BCJR algorithm for processing the *a priori* LLR vectors $\tilde{\mathbf{w}}^{\mathrm{a}} = [\tilde{w}_k^{\mathrm{a}}]_{k=1}^{cn_2}$ and $\tilde{\mathbf{v}}^{\mathrm{a}} = [\tilde{v}_j^{\mathrm{a}}]_{j=1}^{c}$, where the latter adopts all zero values at the start of the iterative decoding process. In response, the CC decoder generates the extrinsic LLR vectors $\tilde{\mathbf{w}}^{\mathrm{e}} = [\tilde{w}_k^{\mathrm{e}}]_{k=1}^{cn_2}$ and $\tilde{\mathbf{v}}^{\mathrm{e}} = [\tilde{v}_j^{\mathrm{e}}]_{j=1}^{c}$, where the latter is iteratively exchanged with the FLC decoder of Figure 4.2. More specifically, $\tilde{\mathbf{v}}^{\mathrm{e}}$ is de-interleaved in the block π_3^{-1} for providing the *a priori* LLR vector $\tilde{\mathbf{u}}^{\mathrm{a}} = [\tilde{u}_j^{\mathrm{a}}]_{j=1}^{c}$ for the FLC decoder. The FLC decoder employs the Soft Bit Source Decoding (SBSD) algorithm of [103] for converting the *a priori* LLR vector $\tilde{\mathbf{u}}^{\mathrm{a}}$ into the decoded sub-symbol vector $\hat{\mathbf{t}} = [t_i]_{i=1}^{a}$ and the extrinsic LLR vector $\tilde{\mathbf{u}}^{\mathrm{e}} = [\tilde{u}_j^{\mathrm{e}}]_{j=1}^{c}$, which is interleaved in the block π_3 to provide the *a priori* LLR vector $\tilde{\mathbf{v}}^{\mathrm{a}} = [\tilde{v}_j^{\mathrm{a}}]_{j=1}^{c}$ for the next iteration of the CC decoder. The SBSD algorithm generates the extrinsic output pertaining to each bit by considering the probability of each of the potential sub-symbol values. More specifically, the *a priori* LLRs $\tilde{u}_j^{\mathrm{a}}$ which constitute a sub-symbol t_i are combined with the conditional probabilities $P(t|x)$, yielding a probability for each of the potential values of that sub-symbol t_i. Each of the extrinsic LLRs $\tilde{u}_j^{\mathrm{e}}$ is obtained by combining the maximum probability of the corresponding LLR being a '0' with the maximum probability of that bit being '1', before subtracting the *a priori* $\tilde{u}_j^{\mathrm{a}}$ LLR from this *a posteriori* value.

As shown in Figure 4.2, the FLC decoder requires the knowledge of the decoded sub-symbol vector $\hat{\mathbf{x}} = [\hat{x}_i]_{i=1}^{a}$, which is provided by the UEC decoder, as described in Section 4.3.2. This allows the SBSD algorithm employed by the FLC decoder to exploit the knowledge of the conditional sub-symbol probabilities $P(t|x)$ [103], where the corresponding decoded sub-symbol $\hat{x}_i$ is employed as a substitute for x_i, when decoding the sub-symbol t_i. The SBSD algorithm generates the hard output $\hat{t}_i$ for each sub-symbol by selecting the most probable value, given the *a priori* LLRs which pertain to that sub-symbol and the conditional probabilities $P(t|x)$. Note that if the source distribution is unknown at the receiver, the SBSD algorithm may be initially operated without the conditional sub-symbol probabilities $P(t|x)$, at the cost of a reduced error correction performance. Once a sufficient number of frames have been decoded, these probabilities may be heuristically estimated and exploited to restore the error correction performance. In the case of the ExpGEC code, the SBSD algorithm also requires the decoded sub-symbols of $\hat{\mathbf{x}}$ in order to determine the number of LLRs of $\tilde{\mathbf{u}}^{\mathrm{a}}$ that should be used to recover each of the sub-symbols of $\hat{\mathbf{t}}$. More specifically, the number of LLRs corresponding to the sub-symbol t_i is given by $l_{\mathrm{FLC}(t_i)} = x_i - 1 + k_{\mathrm{ExpG}}$, as previously described in Section 4.2.3. For the RiceEC code, by contrast, this length is fixed at $l_{\mathrm{FLC}}(t_i) = \log_2(M_{\mathrm{Rice}})$. Note that the complexity of the SBSD algorithm increases exponentially with the length $l_{\mathrm{FLC}}(t_i)$ of the codeword it is tasked to decode, since it considers each of the codeword's $2^{l_{\mathrm{FLC}}(t_i)}$ possible values. Since $l_{\mathrm{FLC}}(t_i)$ grows with x_i in the case of the ExpGEC code, we may limit the FLC decoding complexity by only invoking the SBSD algorithm for calculating the extrinsic LLRs of $\tilde{\mathbf{u}}^{\mathrm{e}}$ for the specific symbols satisfying $\hat{x} \leq x_{\max} = \lfloor \log_2(d_{\max} + 2^{k_{\mathrm{ExpG}}}) \rfloor + 1 - k_{\mathrm{ExpG}}$, where a parameter value of $d_{\max} = 18$ is recommended for striking an attractive trade-off between the decoding complexity imposed and the error correction capability attained. This approach also has the benefit of limiting the number of conditional sub-symbol probabilities $P(t_i|x_i)$ that are exploited by the SBSD algorithm and that must therefore be known and stored by the receiver. For this reason, the SBSD algorithm is only applied in the case of the RiceEC code for the particular symbols satisfying $\hat{x}_i \leq x_{\max} = \lceil (d_{\max} + 1)/M_{\mathrm{Rice}} \rceil$, where $d_{\max} = 18$ is also recommended. The exception to this is found in the cases where we have $M_{\mathrm{Rice}} = 32$ or $M_{\mathrm{Rice}} = 64$, when the choice of $d_{\max} = M_{\mathrm{Rice}}$ is recommended, for the sake of offering an attractive error correction capability. Note that the complexity limit is expressed in terms of d, since the number of conditional sub-symbol probabilities $P(t_i|x_i)$ that are required is equal to $d_{\max}$. The SBSD algorithm is not applied for all other symbols where $\hat{x}_i > x_{\max}$. Instead, zero-values are used for the corresponding extrinsic LLRs of $\tilde{\mathbf{u}}^{\mathrm{e}}$, while the corresponding decoded sub-symbols of $\hat{\mathbf{t}}$ are obtained by applying hard decisions to the corresponding LLRs of $\tilde{\mathbf{u}}^{\mathrm{a}}$.

During each decoding iteration, the URC, CC, and FLC decoders are each activated in

$$\text{ExpGEC: } A_2^{\text{o}} = \frac{1}{n_2 l_2} \sum_{d=1}^{\min(L, 2^{(x_{\max}+k_{\text{ExpG}})}-2^{k_{\text{ExpG}}})} H\Big[P\big(t(d)|x(d)\big)\Big] P\big(x(d)\big)$$
$$+ \frac{1}{n_2} \left(1 - \sum_{d=1}^{\min(L, 2^{(x_{\max}+k_{\text{ExpG}})}-2^{k_{\text{ExpG}}})} \frac{(x - 1 + k_{\text{ExpG}}) P\big(t(d) \cap x(d)\big)}{l_2}\right) \tag{4.31}$$

$$\text{RiceEC: } A_2^{\text{o}} = \frac{1}{n_2 l_2} \sum_{d=1}^{\min(L, M x_{\max})} H\Big[P\big(t(d)|x(d)\big)\Big] P\big(x(d)\big)$$
$$+ \frac{1}{n_2} \left(1 - \sum_{d=1}^{\min(L, M x_{\max})} \frac{\log_2(M_{\text{Rice}}) P\big(t(d) \cap x(d)\big)}{l_2}\right) \tag{4.32}$$

turn, according to the schedule {URC, CC, FLC, URC, CC, FLC, ... }. For symbols satisfying $\hat{x}_i \leq x_{\max}$, this schedule represents a three-stage iterative decoding process, but for symbols where $\hat{x}_i > x_{\max}$, this is effectively a two-stage iterative decoding process between the CC and URC decoders.

The transformation of $\tilde{\mathbf{w}}^{\text{a}}$ into $\tilde{\mathbf{w}}^{\text{e}}$ may be characterized by the FLC-CC decoder's inverted EXIT function, as shown in Figure 4.5. Note that the area beneath the inverted FLC-CC EXIT function may be closely approximated by (4.31) and (4.32) [28].

4.4 Fixed-length interleavers

As described in Section 4.2.4, the EGEC scheme of [28] employs five interleavers having specific lengths and therefore particular designs that change from frame-to-frame, hence limiting the practicality of the EGEC scheme. This may be attributed to the EGEC scheme's partitioning of the sub-symbol streams $\mathbf{x}$ and $\mathbf{t}$ into fixed length vectors, which are encoded using variable length codewords to produce bit vectors having lengths that change from frame-to-frame, Motivated by this, this section describes a novel approach that allows our proposed ExpGEC and RiceEC schemes of Figure 4.2 to use single fixed length designs for the interleavers π_1 to π_5 for each frame, significantly improving its practicality and error correction capability, as discussed in Section 4.1. More specifically, the proposed approach encodes the sub-symbol streams $\mathbf{x}$ and $\mathbf{t}$ into bit-streams, which are partitioned into bit-vectors $\mathbf{y} = [y_j]_{j=1}^{b}$ and $\mathbf{u} = [u_j]_{j=1}^{c}$ having fixed lengths of b and c for the UEC and FLC-CC sub-codes, respectively. The solution proposed in this chapter is designed for accommodating unary and FLC codewords that span across frames, as well as for maintaining synchronization between the UEC and FLC-CC sub-codes of the decoder. This section commences in Section 4.4.1 by detailing how the UEC sub-code partitions the corresponding bit stream into the fixed length bit vector $\mathbf{y}$, in order to ensure that π_1 and π_2 of Figure 4.2 have fixed lengths. Likewise, Section 4.4.2 describes how the FLC-CC sub-code partitions the corresponding bit stream into the fixed length bit vector $\mathbf{u}$, in order to ensure that π_3, π_4 and π_5 of Figure 4.2 have fixed lengths. Section 4.4.3 proves how synchronization is maintained between the UEC and FLC-CC sub-codes, while Section 4.4.4 discusses the specific design of the interleavers π_1 to π_5.

4.4.1 Fixed-length interleavers for the UEC sub-code

As described in Section 4.2.2, the unary encoder of Figure 4.2 converts each sub-symbol x_i in the stream $\mathbf{x}$ into the corresponding unary codeword $\mathbf{y}_i$. These codewords are concatenated to form a bit stream, which is partitioned into fixed length vectors $\mathbf{y} = [y_j]_{j=1}^{b}$, which may cause some unary codewords to be split between consecutive frames. In order to address this, the scheme of Figure 4.2 employs a buffer to store the last bits in the codeword that do not fit into the current frame, so that they can be concatenated onto the start of the next frame. For example, when unary encoding the sub-symbol vector $\mathbf{x} = [1, 2, 1, 1, 4, 1, 3, 1, 2]$ to form bit vectors comprising $b = 8$ bits, we obtain $\mathbf{y} = [1, 0, 1, 1, 1, 0, 0, 0]$ and $\mathbf{y} = [1, 1, 0, 0, 1, 1, 0, 1]$ for two consecutive frames. Note that the unary codeword corresponding to the fifth sub-symbol of $\mathbf{x}$ is split between the two bit-vectors of $\mathbf{y}$. Owing to this, the first and last bits of the bit vector $\mathbf{y}$ are not guaranteed to be the first and last bits of a unary codeword,which is in contrast to the usual UEC operation. The UEC trellis encoder is capable of accommodating this change in two ways.

In a **first method** for accommodating codewords split between two consecutive frames, the UEC trellis encoder may carry over its state between successive frames. The transmitter then uses a small amount of additional side information for reliably conveying this state to the receiver. The UEC trellis decoder may use this side information to initialize the end state of one frame, as well as the initial state of the next. For example, when employing this approach for the two successive bit vectors $\mathbf{y} = [1, 0, 1, 1, 1, 0, 0, 0]$ and $\mathbf{y} = [1, 1, 0, 0, 1, 1, 0, 1]$ from our previous example, the paths transversed through the UEC trellis are given by $\mathbf{m} = [1, 2, 4, 1, 2, 1, 3, 3, 3]$ and $\mathbf{m} = [3, 2, 1, 3, 3, 2, 1, 3, 2]$, where the state $m = 3$ is sent as side information between the transmitter and reciever. This method maintains all of the UEC code's near-capacity capability, although this is achieved at the cost of requiring additional side information to be transmitted.

In a **second method** conceived for accommodating codewords split between two consecutive frames, the trellis encoder may be forced to restart the trellis path m from state 1, when encoding each bit vector $\mathbf{y}$. This does not compromise the near-capacity capability of the UEC code since synchronization is still maintained between the trellis path and the unary codewords despite the trellis encoder potentially starting from the middle of a codeword. More specifically, owing to the particular design of the UEC trellis, the trellis path returns to one of the two central states, whenever the final bit in a unary codeword is encountered, as described in Section 4.2.2. For example, when employing this approach for the successive bit-vectors of $\mathbf{y} = [1, 0, 1, 1, 1, 0, 0, 0]$ and $\mathbf{y} = [1, 1, 0, 0, 1, 1, 0, 1]$ from the previous example, the paths traversed through the UEC trellis are given by $\mathbf{m} = [1, 2, 4, 1, 2, 1, 3, 3, 3]$ and $\mathbf{m} = [1, 2, 1, 3, 3, 2, 1, 3, 2]$, respectively. Observe that these paths are identical to those that result from the first method, with the only exception of the states corresponding to the end of the split codeword, demonstrating that synchronization has been maintained. Note that this approach causes the end state to be unknown to the receiver, since it may correspond to the middle of a unary codeword. This may be accommodated during the Log-BCJR algorithm, by not terminating the UEC trellis at its right-most end, like usual. Furthermore since the first state of each frame is forced to 1, the transition probabilities $P(m_j|m_{j-1})$ employed by the Log-BCJR algorithm for the first codeword may be slightly inaccurate. These two factors impose some error correction performance loss upon the UEC trellis decoder, although this effect is negligible in practice. Owing to this, the results of Section 4.5 will adopt this second method, since it does not require any side information to be sent between the transmitter and receiver.

In the receiver of Figure 4.2, the UEC trellis decoder and URC decoder perform iterative decoding, in order to obtain the *a posteriori* LLR vector $\tilde{\mathbf{y}}$, which pertains to the bit vector $\mathbf{y}$ of the transmitter, as described in Section 4.3.2. Mirroring the buffer employed in the transmitter, the receiver employs a buffer to temporarily store LLRs from the end of the vector $\tilde{\mathbf{y}}$ which correspond to a unary codeword that was split between consecutive frames. More specifically, when the UEC trellis decoder generates the LLR-vector $\tilde{\mathbf{y}}$ for the next frame, it is concatenated on to the end of any LLRs stored in the buffer, forming part of the first codeword in $\tilde{\mathbf{y}}$. The concatenated LLR-vector is then provided to the unary decoder, which generates the vector $\hat{\mathbf{x}}$ comprising a sub-symbols, as described in Section 4.3.2. Following this, any trailing LLRs of $\tilde{\mathbf{y}}$ that did not contribute to a complete unary codeword are placed into the buffer, ready to be concatenated to the beginning of the next LLR-vector $\tilde{\mathbf{y}}$. Note that since the number of sub-symbols a in the vector $\hat{\mathbf{x}}$ varies from frame to frame, it must be conveyed to the receiver using a small amount of side information, as previously discussed in Section 4.3.2. More specifically, this side information conveys the number of logical one-valued bits in $\mathbf{y}$, as exploited by the unary decoder. While the proposed scheme of Figure 4.2 is capable of functioning without this side information, its inclusion guarantees synchronization between the UEC and FLC-CC parts, as it will be described in Section 4.4.3. Using the example given above, if the unary decoder is successful in decoding the LLR vector $\tilde{\mathbf{y}}$ of the first frame, it will output the sub-symbol vector $\hat{\mathbf{x}} = [1, 2, 1, 1]$, and will place the last three LLRs of $\tilde{\mathbf{y}}$ into the buffer. Following this, the unary decoder will output the sub-symbol vector $\hat{\mathbf{x}} = [4, 1, 3, 1, 2]$ if it is successful in decoding the LLR-vector $\tilde{\mathbf{y}}$ of the second frame, with no LLRs needed to be placed into the buffer. The decoded sub-symbols of $\hat{\mathbf{x}}$ are buffered until the corresponding sub-symbols of $\hat{\mathbf{t}}$ have also been decoded by the FLC decoder, as will be discussed in Section 4.4.3.

4.4.2 Fixed-length interleavers for the FLC-CC sub-code

In a similar fashion to the UEC encoder, the FLC encoder of Figure 4.2 converts each sub-symbol t_i in the stream $\mathbf{t}$ into the corresponding FLC codeword $\mathbf{u}_i$, as described in Section 4.2.3. These codewords are concatenated to form a stream of bits, which is then partitioned into fixed length bit-vectors $\mathbf{u} = [u_j]_{j=1}^{c}$, which may result in some codewords becoming split between consecutive frames. In order to address this, the scheme of Figure 4.2 employs a buffer for storing the last bits in the codeword that do not fit into the current frame, so that they can be concatenated onto the start of the next frame. For example, in the case of a RiceEC code associated with $M_{\text{Rice}} = 2$, when FLC encoding the sub-symbol vector $\mathbf{t} = [1, 3, 2, 0, 2]$ to form bit vectors comprising $c = 5$ bits, we obtain $\mathbf{u} = [0, 1, 1, 1, 1]$ and $\mathbf{u} = [0, 0, 0, 1, 0]$ for two consecutive frames. Note that the FLC codeword corresponding to the third sub-symbol of $\mathbf{t}$ is split across the two bit-vectors of $\mathbf{u}$. In the receiver of Figure 4.2, the CC decoder iteratively exchanges the LLR-vectors $\tilde{\mathbf{w}}^{\text{a}}$ and $\tilde{\mathbf{w}}^{\text{e}}$ with the URC decoder, as well as the LLR-vectors $\tilde{\mathbf{v}}^{\text{a}}$ and $\tilde{\mathbf{v}}^{\text{e}}$ with the FLC decoder, as described in Section 4.3.3. More specifically, $\tilde{\mathbf{v}}^{\text{e}}$ is de-interleaved in the block π_3^{-1} to obtain $\tilde{\mathbf{u}}^{\text{a}}$, while $\tilde{\mathbf{u}}^{\text{e}}$ is interleaved to obtain $\tilde{\mathbf{v}}^{\text{a}}$. However, the FLC decoder can only generate a sub-symbol of $\hat{\mathbf{t}}$ and a codeword of extrinsic LLRs for $\tilde{\mathbf{v}}^{\text{e}}$ when the *a priori* LLR vector $\tilde{\mathbf{v}}^{\text{a}}$ contains all of the *a priori* LLRs for that codeword. Owing to this, a buffer is required for storing *a priori* LLRs for incomplete codewords that have been split between frames, as shown in Figure 4.2. When the CC decoder forwards an *a priori* LLR vector $\tilde{\mathbf{u}}^{\text{a}}$ to the FLC decoder, it is concatenated onto the end of any LLRs stored in the buffer from the previous frame. In the case of the ExpGEC scheme, the decoded sub-symbols of $\hat{\mathbf{x}}$ are used for determining the length of each of the codewords in $\tilde{\mathbf{u}}^{\text{a}}$, hence allowing the FLC decoder to determine how many excess trailing LLRs of $\tilde{\mathbf{u}}^{\text{a}}$ must be stored in the buffer for each frame. In the case of the RiceEC scheme, since the length of the FLC codewords is fixed at $\log_2(M_{\text{Rice}})$, the FLC decoder does not need any additional information for determining how many excess trailing LLRs to place in the buffer for each frame. The FLC decoder processes the *a priori* LLRs of $\tilde{\mathbf{u}}^{\text{a}}$ comprising complete codewords, in order to generate corresponding extrinsic LLRs for the vector $\tilde{\mathbf{u}}^{\text{e}}$, which is provided to the CC decoder. The extrinsic LLRs corresponding to the *a priori* LLRs that were concatenated from the buffer must be removed before $\tilde{\mathbf{u}}^{\text{e}}$ is provided to the CC decoder, since they are not relevant to the current frame. Likewise, zero-valued LLRs must be inserted at the end of $\tilde{\mathbf{u}}^{\text{e}}$ to pad the vector, in correspondence to the *a priori* LLRs of $\tilde{\mathbf{u}}^{\text{a}}$ that were placed into the buffer, rather than being decoded by the FLC decoder. As a result of this, the FLC decoder and CC decoder may not iteratively exchange LLRs pertaining to every bit in the vector $\mathbf{u}$. This may therefore result in a slight degradation of the FLC decoder's near-capacity capability, in a similar manner to the effect of limiting the FLC decoding complexity using the parameter $x_{\max}$. However, since only a small number of LLRs in the vector $\tilde{\mathbf{u}}^{\mathrm{e}}$ are impacted in this way, this effect is negligible in practice.

4.4.3 Matching sub-symbols

As described in Section 4.2, the ExpGEC and RiceEC codes decompose the symbol stream $\mathbf{d}$ into two sub-symbol streams $\mathbf{x}$ and $\mathbf{t}$. It is important to ensure that the two sub-symbol streams remain synchronized at the receiver, even in the presence of transmission errors. More specifically, if errors cause deleted or inserted sub-symbols to appear in $\hat{\mathbf{x}}$ or $\hat{\mathbf{t}}$, then the incorrect pairings of sub-symbols will be recombined for generating the reconstructed symbol stream $\hat{\mathbf{d}}$, hence resulting in a high SER for the remainder of the stream. This motivates the approach described in this section for avoiding the two sub-symbol streams becoming de-synchronized. This section commences by describing how the transmitter multiplexes the UEC part and FLC-CC part onto the channel, then discuss how the receiver maintains synchronization.

As explained in Section 4.3.3, the FLC decoder requires knowledge of the sub-symbol $\hat{x}_i$ in order to generate the corresponding sub-symbol $\hat{t}_i$. Motivated by this, the transmitter ensures that each sub-symbol x_i is encoded and transmitted in advance of its corresponding sub-symbol t_i, so that the reconstructed sub-symbol $\hat{x}_i$ is available for use by the FLC decoder at the appropriate time. Since the number of sub-symbols of $\mathbf{x}$ and $\mathbf{t}$ that are represented by each bit vector of $\mathbf{z}$ and $\mathbf{w}$ may vary from frame to frame, the transmitter has to ensure that a sufficiently high number of sub-symbols of $\mathbf{x}$ have been transmitted before transmitting a bit vector representing a selection of sub-symbols of $\mathbf{t}$. Since the transmitter is capable of maintaining a count of how many sub-symbols of $\mathbf{x}$ it has transmitted at any given time, it can infer how many sub-symbols of $\hat{\mathbf{x}}$ are buffered in the receiver, ready to be used by the FLC decoder. Likewise, the transmitter employs a buffer for storing the sub-symbols of $\mathbf{t}$ until they are ready for transmission, as shown in Figure 4.2. When the transmitter identifies that the number of sub-symbols of $\hat{\mathbf{x}}$ buffered in the receiver exceeds the number of sub-symbols of $\mathbf{t}$ required to generate the next bit vector $\mathbf{w}$, it is transmitted to the receiver.

The proposed ExpGEC and RiceEC schemes are designed to ensure that synchronization is maintained between the UEC and FLC-CC decoders. As described above, de-synchronization can occur if the unary decoder or FLC decoder output too many or too few sub-symbols for $\hat{\mathbf{x}}$ and $\hat{\mathbf{t}}$, when decoding the LLR vectors $\tilde{\mathbf{y}}$ and $\tilde{\mathbf{u}}^{\mathrm{a}}$, respectively. However, owing to the side information used to indicate to the UEC decoder how many one-valued bits there are in each bit vector $\mathbf{y}$, the unary decoder is guaranteed to output the correct number of sub-symbols for $\hat{\mathbf{x}}$, since each unary codeword contains only a single logical one-valued bit, as described in Section 4.4.1. For the RiceEC code, the correct number of sub-symbols for $\hat{\mathbf{t}}$ is also guaranteed, since each FLC codeword comprises the same number of $\log_2(M_{\text{Rice}})$ bits. In the case of the ExpGEC code, the properties of the UEC code may be exploited for ensuring synchronization. More specifically, the number of FLC encoded bits of $\mathbf{u}$ associated with each decoded vector $\hat{\mathbf{x}} = [\hat{x}_i]_{i=1}^{a}$ is given by $\sum_{i=1}^{a}(x_i + k_{\text{ExpG}} - 1)$, where a is the length of a specific decoded vector of $\hat{\mathbf{x}}$, which can be correctly inferred from the side information, which conveys the number of logical one-valued bits in each vector $\mathbf{y}$. Since the length of each unary encoded codeword is given by $l_{\text{Unary}(x_i)} = x_i$, the previous sum may be expressed as $\sum_{i=1}^{a}(l_{\text{Unary}(x_i)} + k_{\text{ExpG}} - 1) = a(k_{\text{ExpG}} - 1) + \sum_{i=1}^{a} l_{\text{Unary}(x_i)}$, where $\sum_{i=1}^{a} l_{\text{Unary}(x_i)}$ is the number of LLRs in $\tilde{\mathbf{y}}$ constituting the sub-symbol vector $\hat{\mathbf{x}}$. Since the values of a and the number of LLRs in $\tilde{\mathbf{y}}$ are known to the receiver, the number of LLRs of $\tilde{\mathbf{u}}^{\text{a}}$ associated with a decoded sub-symbol vector of $\hat{\mathbf{x}}$ does not depend on the received value of those sub-symbols. Owing to this, any errors in $\hat{\mathbf{x}}$ cannot cause the sub-symbols $\hat{\mathbf{x}}$ and $\hat{\mathbf{t}}$ to become permanently de-synchronized.

4.4.4 Interleavers

In contrast to the JSCC schemes of our previous work [28], the proposed ExpGEC and RiceEC schemes employ interleavers π_1 to π_5 of Figure 4.2 having fixed lengths, which do not change from frame to frame. Owing to this, these lengths and the corresponding interleaver designs may be hard-coded into the transmitter and receiver. This has the added benefit of avoiding the large memory requirement for storing a large number of interleaver designs, as well as the additional design effort required for ensuring that all the interleavers have desirable distance properties. It also avoids the requirement for using side information to signal which interleaver lengths are used for each frame.

The interleavers π_2 and π_5 of Figure 4.2 are required for evenly distributing the punctured or doped bits throughout the corresponding bit vector, as opposed to the interleavers π_1, π_3 and π_4 of Figure 4.2, which are required for maintaining desirable distance properties. As a result, the LTE sub-block interleaver [104] for the interleavers π_2 and π_5 is recommended in order to evenly distribute the doped or interleaved bits. Meanwhile, S-Random interleavers [105] are recommended for the interleavers π_1, π_3 and π_4.

4.5 Performance comparison with benchmarkers

In this section, the performance of our proposed RiceEC and ExpGEC schemes are compared with that of several appropriate benchmarkers, which are introduced in Section 4.5.1. Section 4.5.2, analyses the near-capacity potential of both the proposed codes and of the benchmarkers. Following this, Section 4.5.3 discusses the EXIT chart analysis which is used for selecting the doping or puncturing rates R_1^{i} and R_2^{i}, which control the unequal error correction of the proposed schemes. Finally, Section 4.5 compares the performance of the proposed schemes and the benchmarkers.

4.5.1 Scenarios and benchmarkers

Table 4.2 lists the considered parameterisations of the proposed schemes and of the benchmarkers, namely of the Rice-CC, ExpG-CC and VLEC schemes [99]. In analogy with the EG-CC scheme of [28], the Rice-CC and ExpG-CC benchmarkers are both SSCCs, which replace the EG source code of [28] by the Rice and ExpG codes, respectively. Note that the EG-CC scheme of [28] may be viewed as the special case of the ExpG-CC benchmarker, where $k_{\text{ExpG}} = 0$. In these benchmarkers, separate channel coding is provided by a serial concatenation of an $r = 4$-state non-systematic recursive CC of [6, Table II] with an $r = 2$-state URC and Grey-coded QPSK modulation. Note that this combination was shown to minimize (but not eliminate) the capacity loss of SSCC schemes like the Rice-CC and ExpG-CC benchmarkers [6, 28]. A further benchmarker is provided by the JSCC VLEC scheme of [5,99] in which a VLEC code is serially concatenated with a $r = 2$ state URC and Gray-coded QPSK modulator. This scheme offers a near capacity operation, but has a rapidly increasing complexity as L increases, as described in Section 4.1.

As shown in Table 4.2, our comparisons consider several scenarios, including the case where the source symbols represent the 26 letters of the English alphabet and the space character, which have a particular probability distribution, as shown in Figure 4.1b. Here, we map the letters and the space character to the symbol values of 1 to 27 according to descending probability of occurrence. Figure 4.1b compares this probability distribution to the finite zeta-like distribution of (4.1) having a parameter value of $L = 27$ and various values of p_1. The entropy of the letters probability distribution is $H_D = 4.1$ bits per symbol, which is equal to that of the finite zeta-like distribution having $p_1 = 0.45$. This section also considers the scenario where the source symbols obey the same probability distribution as the transform coefficients and motion vectors produced by a H.265 encoder, as shown in Figure 4.1b. Note that while the H.265 encoder produces some symbols having values higher than 1000, these have a combined probability of less than 2×10^{-5}. Motivated by this, Figure 4.1a also shows the finite zeta-like distribution having $L = 1000$ and various values of p_1. The entropy of the H.265 probability distribution is $H_D = 2.4$ bits per symbol, which is equal to that of the finite zeta-like distribution having $p_1 = 0.6$.

As shown in Table 4.2, the scenarios where the source symbols obey the finite zeta-like probability distribution of (4.1) are also considered. Here, the alphabet cardinality of $L = 27$ is used in order to match that of the English letter source set, as well as the cardinality $L = 1000$, since this is approximately equal to the highest symbol value produced by H.264 [6, Fig. 7] and H.265. This approach allows the performance of the

Table 4.2: Table of the selected parameters for the proposed schemes and the benchmarkers. All schemes use $r_1 = 4$ and $r_2 = 4$ states. Likewise all schemes use $d_{\max} = 18$, except for the RiceEC schemes having $M_{\text{Rice}} = 32$ and $M_{\text{Rice}} = 64$, where $d_{\max} = M_{\text{Rice}}$ is employed.

L	p_1 or distribution	η	Scheme, parameter	k_{ExpG}/ M_{Rice}	n_1	n_2	R_1^{o}	R_2^{o}	A_1^{o}	A_2^{o}	R_1^{i}	R_2^{i}	Capacity bound E_{b}/N_0	Area bound E_{b}/N_0	Tunnel bound E_{b}/N_0	Complexity
27	Letters	0.95	ExpGEC	0	2	2	0.374	0.466	0.434	0.490	1.186	1.116	1.60	2.29	3.50	304
				3	2	2	0.354	0.473	0.354	0.474	1.277	1.021		1.68	2.63	311
			RiceEC	4	2	2	0.484	0.489	0.494	0.492	0.972	0.987		1.69	2.67	252
				8	2	2	0.432	0.480	0.442	0.479	1.067	1.004		1.65	2.58	281
			ExpG-CC	1	2		0.443		0.491		1.072			2.07	3.42	270
			Rice-CC	4	2		0.485		0.500		0.979			1.73	2.70	245
			VLEC-URC		2		0.377		0.375		1.289			1.67	3.01	9641
	0.2	0.90	ExpGEC	0	2	2	0.386	0.476	0.444	0.497	1.093	1.028	1.39	1.94	3.08	278
				2	2	2	0.427	0.481	0.477	0.481	0.981	0.979		1.55	2.64	264
			RiceEC	1	2		0.260		0.266		1.732			1.49	3.71	435
				8	2	2	0.460	0.463	0.475	0.463	0.965	0.980		1.46	2.62	265
			ExpG-CC	0	2		0.422		0.475		1.066			1.89	3.52	267
			Rice-CC	4	2		0.472		0.500		0.954			1.63	2.70	240
			VLEC-URC		2		0.360		0.366		1.249			1.44	2.74	9717
	0.4	0.83	ExpGEC	0	2	2	0.471	0.474	0.493	0.488	0.872	0.892	1.10	1.26	2.20	228
				2	2	2	0.413	0.437	0.432	0.436	0.980	0.965		1.17	2.33	259
			RiceEC	1	2		0.350		0.358		1.184			1.17	3.01	251
				4	2	2	0.455	0.430	0.462	0.432	0.917	0.960		1.17	2.23	243
			ExpG-CC	0	3		0.315		0.329		1.318			1.26	3.10	409
			Rice-CC	4	2		0.442		0.469		0.940			1.31	2.67	237
			VLEC-URC		2		0.325		0.335		1.276			1.20	2.43	10693
	0.6	0.85	ExpGEC	0	2	2	0.479	0.461	0.485	0.469	0.886	0.916	1.19	1.22	2.18	229
				1	2	2	0.407	0.410	0.411	0.409	1.046	1.038		1.23	2.53	268
			RiceEC	1	2		0.426		0.438		0.997			1.27	2.65	297
				2	2	2	0.424	0.397	0.429	0.397	1.006	1.068		1.24	2.42	260
			ExpG-CC	0	3		0.316		0.332		1.344			1.36	3.45	417
			Rice-CC	2	2		0.413		0.465		1.028			1.63	3.27	260
			VLEC-URC		2		0.272		0.280		1.565			1.29	3.23	14007
1000	H.265	0.85	ExpGEC	0	2	2	0.464	0.386	0.465	0.375	0.926	1.071	1.19	1.25	2.35	248
				1	2	2	0.408	0.481	0.409	0.472	1.008	0.909		1.21	2.26	246
			RiceEC	1	2	2	0.348		0.427		1.222			2.03	4.66	307
				2	2	2	0.351	0.499	0.397	0.500	1.121	0.956		1.65	3.05	269
			ExpG-CC	1	2		0.445		0.489		0.956			1.56	2.78	241
			Rice-CC	4	2		0.351		0.446		1.210			2.20	4.70	305
	0.2	0.90	ExpGEC	0	2	2	0.368	0.495	0.388	0.498	1.161	0.959	1.39	1.63	2.85	273
				3	2	2	0.456	0.470	0.478	0.471	0.968	0.968		1.48	2.42	256
			RiceEC	64	2	2	0.390	0.405	0.395	0.405	1.148	1.114		1.42	2.65	494
			ExpG-CC	1	2		0.447		0.471		1.007			1.60	2.81	254
			Rice-CC	64	2		0.401		0.475		1.122			2.14	4.07	283
	0.4	0.83	ExpGEC	0	2	3	0.473	0.325	0.477	0.327	0.914	1.200	1.10	1.27	2.45	285
				3	2	2	0.425	0.405	0.429	0.405	0.969	1.029		1.12	2.31	277
			RiceEC	32	2	2	0.274	0.327	0.285	0.327	1.441	1.281		1.15	2.78	478
			ExpG-CC	0	3		0.319		0.323		1.300			1.15	2.95	403
			Rice-CC	16	2		0.334		0.466		1.244			2.63	5.67	313
	0.6	0.85	ExpGEC	0	2	2	0.491	0.470	0.491	0.473	0.872	0.892	1.19	1.21	2.09	224
				1	2	2	0.430	0.414	0.430	0.414	0.993	1.024		1.20	2.40	259
			RiceEC	1	2		0.246		0.291		1.712			1.83	5.70	430
				8	2	2	0.224	0.300	0.240	0.300	1.771	1.453		1.26	3.50	425
			ExpG-CC	0	3		0.282		0.333		1.316			1.29	2.21	408
			Rice-CC	4	2		0.315		0.454		1.347			2.98	6.69	339

various schemes considered to be compared for both small and large symbol alphabet cardinalities L. Furthermore, this section parametrises the finite zeta-like probability distributions using $p_1 \in \{0.2, 0.4, 0.6\}$, which spans the range of p_1 values that best approximate the English alphabet and the H.265 probability distributions, as discussed above. Note that the previous work of [28] focused only on the complementary parameter value range of $p_1 \geq 0.6$.

As shown in Table 4.2, the puncturing and doping of the proposed schemes and benchmarkers are specifically parameterised in each scenario for achieving the same effective throughput η, for the sake of facilitating fair comparisons. More specifically, the effective throughput of $\eta = 0.95$ was selected for the English letters distribution, while the effective throughput of $\eta = 0.85$ was selected for the H.265 distribution. Meanwhile, the target throughputs of $\eta \in \{0.90, 0.83, 0.85\}$ were selected for the scenarios using the finite zeta-like distribution having $p_1 \in \{0.2, 0.4, 0.6\}$, respectively. These target effective throughputs were selected since they are close to the native effective throughputs of all schemes considered, ensuring that none of them were excessively impacted by puncturing or doping.

As described in Section 4.2, the proposed RiceEC and ExpGEC schemes are parameterised by M_{Rice} and k_{ExpG}, respectively. Table 4.2 shows the specific values of M_{Rice} and k_{ExpG} that were selected in each scenario considered. In each case, we selected the particular parameter values that give the best SER performance, as will be characterized in Section 4.5.4. We also selected $M_{\text{Rice}} = 1$ and $k_{\text{ExpG}} = 0$, which reduce the RiceEC and ExpGEC schemes to the special cases of the UEC and EGEC schemes of our previous work [6, 28]. The latter arrangements served as additional benchmarkers. The only exception to this however is the omission of the UEC benchmarker in scenarios where excessive puncturing would be required for achieving the effective target throughput η. For example, the UEC benchmarker has very small outer coding rates of $R_1^{\text{o}} = 0.0004$ for the case of a finite zeta-like distribution having $p_1 = 0.2$ and $L = 1000$, as well as $R_1^{\text{o}} = 0.007$ for $p_1 = 0.4$ and $L = 1000$. This may be expected, since the UEC benchmarker has an average codeword length l_{Unary} which approaches infinity, as the source alphabet cardinality L tends to infinity, for the case of $p_1 < 0.608$. Despite this, the UEC benchmarker has only a moderately small outer coding rate of $R_1^{\text{o}} = 0.246$ for the case of the finite zeta-like distribution having $p_1 = 0.6$ and $L = 1000$, as well as of $R_1^{\text{o}} = 0.348$ for the case of the H.265 distribution, as shown in Table 4.2. Since the Rice-CC and ExpG-CC benchmarkers employ the Rice and ExpG source codes respectively, they are also parameterised by M_{Rice} and k_{ExpG}. Table 4.2 shows the values of M_{Rice} and k_{ExpG} that were selected for the Rice-CC and ExpG-CC benchmarkers in each of the scenarios considered. In each case, this selection was made by finding the parameterisation of M_{Rice} or k_{ExpG} that gives the best SER performance, as will be characterized in Section 4.5.4.

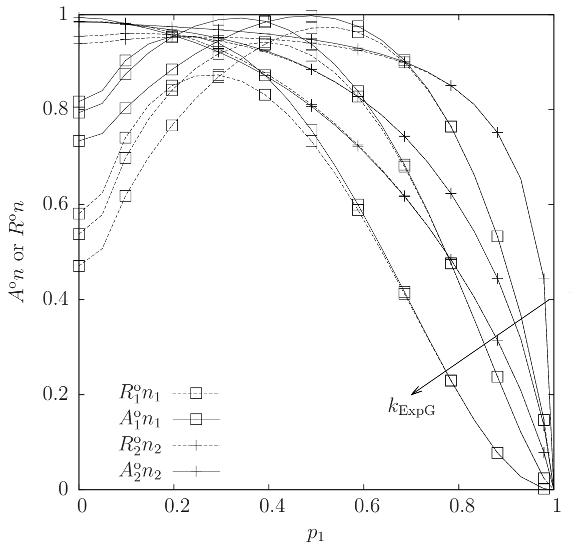
Note that the VLEC benchmarker is only considered for scenarios having source symbol alphabets associated with the cardinality of $L = 27$, since it suffers from an excessive trellis complexity for larger values of L. The VLEC codewords were designed using the approach of [106], which was parameterised using a block distance $d_{\mathrm{b}} = 2$, divergence distance $d_{\mathrm{d}} = 2$ and convergence distance $d_{\mathrm{c}} = 1$. Here, the block distance d_b specifies the minimum Hamming distance required between all pairs of equal length VLEC codewords. Meanwhile, the divergence distance d_d and convergence distance d_c specify the minimum Hamming distances required between all pairs of un-equal length codewords, when they are left- and right-aligned, respectively [5,102]. After a VLEC codebook was designed in this way, each codeword was then doubled in length by concatenating it with its inverse. For example, the codebook $\{101, 1001, 10001\}$ becomes $\{101010, 10010110, 1000101110\}$ using this approach. This enables a fair comparison by reducing the VLEC coding rate R^{o} to become comparable to those of the other schemes, while also ensuring that the VLEC-encoded bits have equiprobable values, which is a necessary condition for avoiding capacity loss [6].

The final column of Table 4.2 quantifies the complexity of each scheme, which was quantified in terms of the number of Add-Compare-Select (ACS) operations performed per bit entered into the QPSK modulator, per decoding iteration. This method of comparing complexity is motivated since logarithmic implementations of the algorithms used within each of the iterative decoding blocks can be decomposed into only the basic ACS operations. In particular, each max* operation employed by the Log-BCJR algorithm is assumed to be computed using several lookup table operations, requiring a total of 5 ACS operations [51].

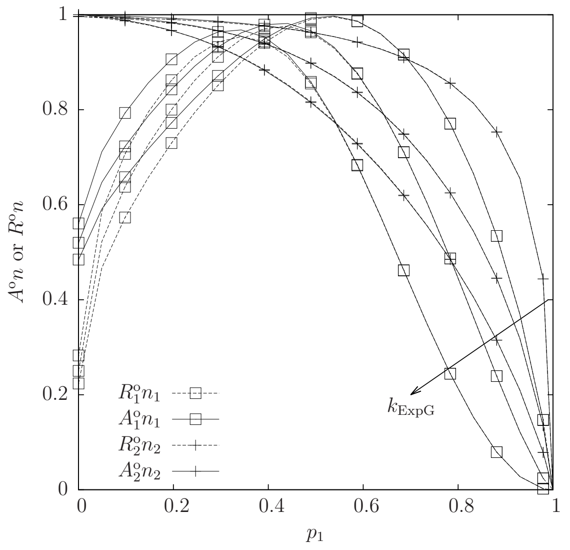
4.5.2 Near capacity analysis

There are two requirements for an iterative decoding scheme to facilitate near-capacity operation, where reliable communication is achieved using an effective throughput η that approaches the Discrete-Input Continuous-Output Memoryless Channel (DCMC) capacity C [107]. Firstly, the inner coding rate should obey $A^{\mathrm{i}} = C/[R^{\mathrm{i}} \log_2(M_{\mathrm{mod}})]$, where C is the DCMC capacity. As discussed in [26], this condition is satisfied by the URC, regardless of the puncturing or doping rate R^{i} used for meeting the required effective throughput η. Secondly, the areas beneath the inverted EXIT outer functions A^{o} should approach the corresponding outer coding rates R^{o} [80]. Motivated by this, this section compares the areas A^{o}_1 and A^{o}_2 beneath the inverted EXIT functions of the UEC and FLC-CC sub-codes of the proposed RiceEC and ExpGEC schemes with the corresponding coding rates R^{o}_1 and R^{o}_2, in order to characterize their near-capacity operation.

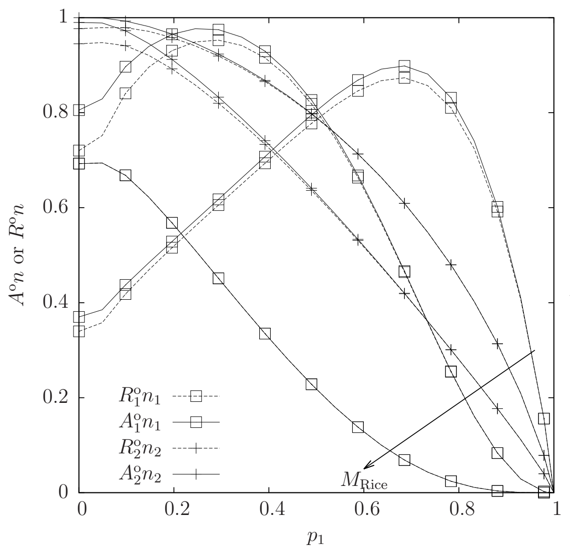
Figure 4.6 shows the product of the coding rates R^{o} and inverted EXIT chart areas A^{o} with the corresponding codeword lengths n for the proposed schemes as functions



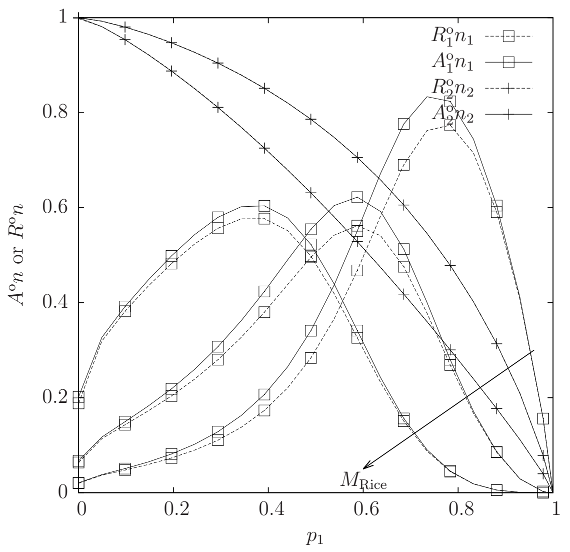
(a) ExpGEC, $L = 27$



(b) ExpGEC, $L = 1000$

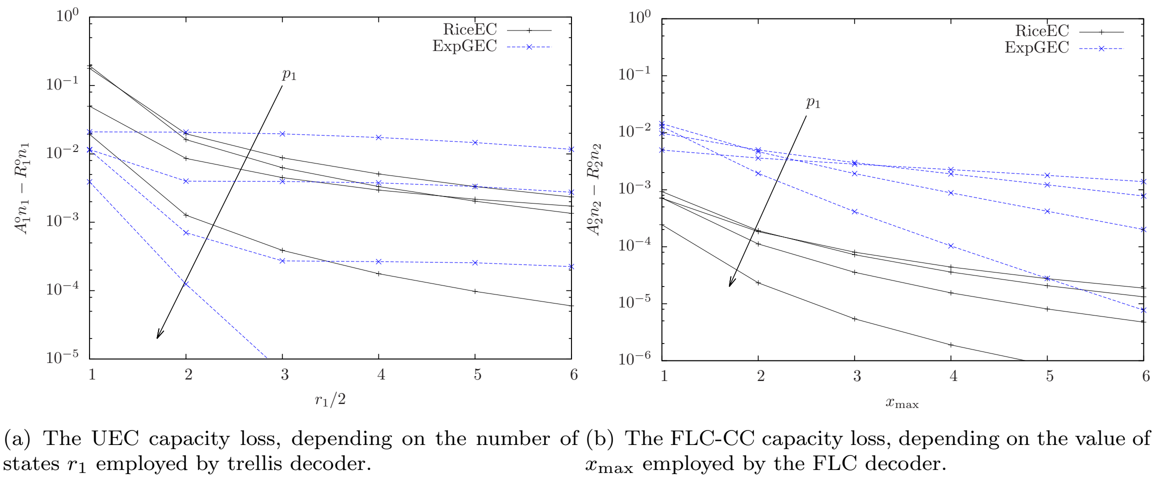


(c) RiceEC, $L = 27$



(d) RiceEC, $L = 1000$

Figure 4.6: Product of the coding rate R^{o} and inverted EXIT chart area A^{o} with the codeword length n for the UEC and FLC-CC sub-codes of the proposed ExpGEC and RiceEC schemes, for the case of the finite zeta-like source distribution with various p_1 values and $L \in \{27, 1000\}$. A parameter value of $r_1 = 4$ is used to obtain the inverted UEC EXIT chart area A_1^{o}, while $d_{\text{max}} = 18$ is used to obtain the inverted FLC-CC EXIT chart area A_2^{o}. (a) and (b) characterize the proposed ExpGEC scheme for different values of the parameter $k_{\text{ExpG}} \in \{0, 1, 2\}$, while (c) and (d) characterize the proposed RiceEC scheme for different values of the parameter $M_{\text{Rice}} \in \{1, 4, 16\}$. Note that there are no plots of A_2^{o} and R_2^{o} for the RiceEG scheme having $M_{\text{Rice}} = 1$, since there is no FLC-CC sub-code in this case.



(a) The UEC capacity loss, depending on the number of states r_1 employed by trellis decoder.

(b) The FLC-CC capacity loss, depending on the value of $x_{\max}$ employed by the FLC decoder.

Figure 4.7: The capacity loss $A^o n - R^o n$ for the UEC and FLC-CC sub-codes with $L = 1000$ and $p_1 \in \{0.2, 0.4, 0.6, 0.8\}$, where $k_{\text{ExpG}} = 1$ in the case of the ExpGEC code and $M_{\text{Rice}} = 8$ in the case of the RiceEC code.

of the finite zeta-like distribution parameter p_1. Here the products $A^o n$ and $R^o n$ have been plotted to eliminate the effect of different codeword lengths n on the analysis. Furthermore, the values of R^o and A^o for each of the considered schemes are listed in Table 4.2 for each scenario considered. For both the RiceEC and ExpGEC schemes, the coding rate R_1^o was obtained using (4.19), while the FLC-CC coding rate R_2^o was obtained using (4.28). Likewise, (4.30) is used to obtain the area beneath the inverted UEC EXIT function A_1^o, for the case of employing $r_1 = 4$ states. Meanwhile, (4.31) and (4.32) are used for obtaining the area beneath the inverted FLC-CC EXIT function A_2^o, where we use $d_{\max} = 18$. These values of r_1 and $d_{\max}$ were found to strike an attractive tradeoff between the complexity and near-capacity operation, as discussed in Sections 4.3.2 and 4.3.3, respectively.

The discrepancy between $A^o n$ and $R^o n$ represents capacity loss, as discussed in [6]. Figure 4.6 shows that the capacity loss $A^o n - R^o n$ depends on the particular scheme, scenario and parameterisation considered. The capacity loss $A_1^o n_1 - R_1^o n_1$ of the UEC sub-code of the proposed RiceEG and ExpGEC schemes depends on the number of states r_1 employed by the UEC trellis decoder, as shown in Figure 4.7(a). If more than $r_1 = 4$ states are employed, then the capacity loss shown in Figure 4.6 will be reduced accordingly, as it may also be seen in (4.19) and (4.30), which show that A_1^o approaches R_1^o as r_1 approaches $2L$. Figure 4.7(a) also shows that this capacity loss reduces as p_1 is increased, since this results in the less frequent occurrence of higher sub-symbol values in the stream $\mathbf{x}$, which would benefit from using more than $r_1 = 4$ states in the UEC trellis decoder to exploit knowledge of the corresponding occurrence probabilities. As characterized in Figure 4.7(b), the capacity loss $A_2^o n_2 - R_2^o n_2$ of the FLC-CC sub-code is caused by symbols in the stream $\mathbf{d}$ having values that exceed the limit $d_{\max}$, which occur more frequently as p_1 is reduced. This explains why employing a value higher than $d_{\max} = 18$ reduces the capacity loss shown in Figure 4.7(b). This can also be seen in (4.31) and (4.32), which show that A_2^{o} approaches R_2^{o} as $d_{\max}$ approaches L.

Table 4.2 also quantifies the E_{b}/N_0 values, where the performance of each scheme considered is limited by the *capacity bound*, *area bound* and *tunnel bound*. More specifically, the *capacity bound* is the E_{b}/N_0 value where the DCMC capacity C becomes equal to the effective throughput η of the scheme, representing theoretical minimum E_{b}/N_0 at which reliable communication is possible [26]. The *area bound* is the E_{b}/N_0 value at which $A^{\mathrm{i}} = A^{\mathrm{o}}$, implying that it is theoretically possible to create an open EXIT tunnel, providing that there is a good match between the shapes of the EXIT curves of the schemes' inner and outer decoders [28]. The discrepancy between the capacity bound and the area bound represents the capacity loss of the particular scheme. Table 4.2 shows that our best performing ExpGEC and RiceEC schemes have an area bound that is within 0.1 dB of the capacity bound, demonstrating their capability of near-capacity operation. By contrast, the discrepancies between the area and capacity bounds are significantly higher for the SSCC ExpG-CC and Rice-CC benchmakers, owing to the corresponding capacity loss. Finally the *tunnel bound* is the E_{b}/N_0 value at which an open tunnel is actually created between the EXIT curves of the scheme's inner and outer decoders [108]. Note that for all proposed schemes and for all benchmarkers, the 2-state URC was found to produce lower tunnel bounds than any 4-state or 8-state URCs. Furthermore, the 2-state URC exhibits the additional benefit of having the lowest complexity of these design options.

4.5.3 EXIT chart matching

This section discusses the employment of EXIT charts for designing the parameterisation of the proposed ExpGEC and RiceEC schemes, as well as for characterizing their iterative decoding convergence and that of the benchmarkers. This facilitates the rapid characterization of these schemes, considering a wide range of values for the scheme parameters such as k_{ExpG}, M_{Rice}, R_1^{i}, R_2^{i}, n_1 and n_2, as well as for various combinations of the scenario parameters p_1 and L, without requiring time-consuming SER simulations. Separate EXIT charts may be used for characterizing the pair of iterative decoding processes corresponding to the UEC and FLC-CC sub-codes of our proposed RiceEC and ExpGEC schemes. As discussed in [26], codeword lengths of $n_1 \geq 2$ and $n_2 \geq 2$ are required in the proposed schemes in order to facilitate iterative decoding convergence to the (1,1) point in the EXIT chart, associated with a vanishingly low SER [26]. If both sub-codes correspond to equal inner coding rates of R_1^{i} and R_2^{i} and therefore equal amounts of puncturing or doping, then they may be said to adopt equal error protection. However, in this case, the EXIT charts corresponding to the UEC and FLC sub-codes may not form marginally open tunnels at the same channel E_{b}/N_0 value. This results

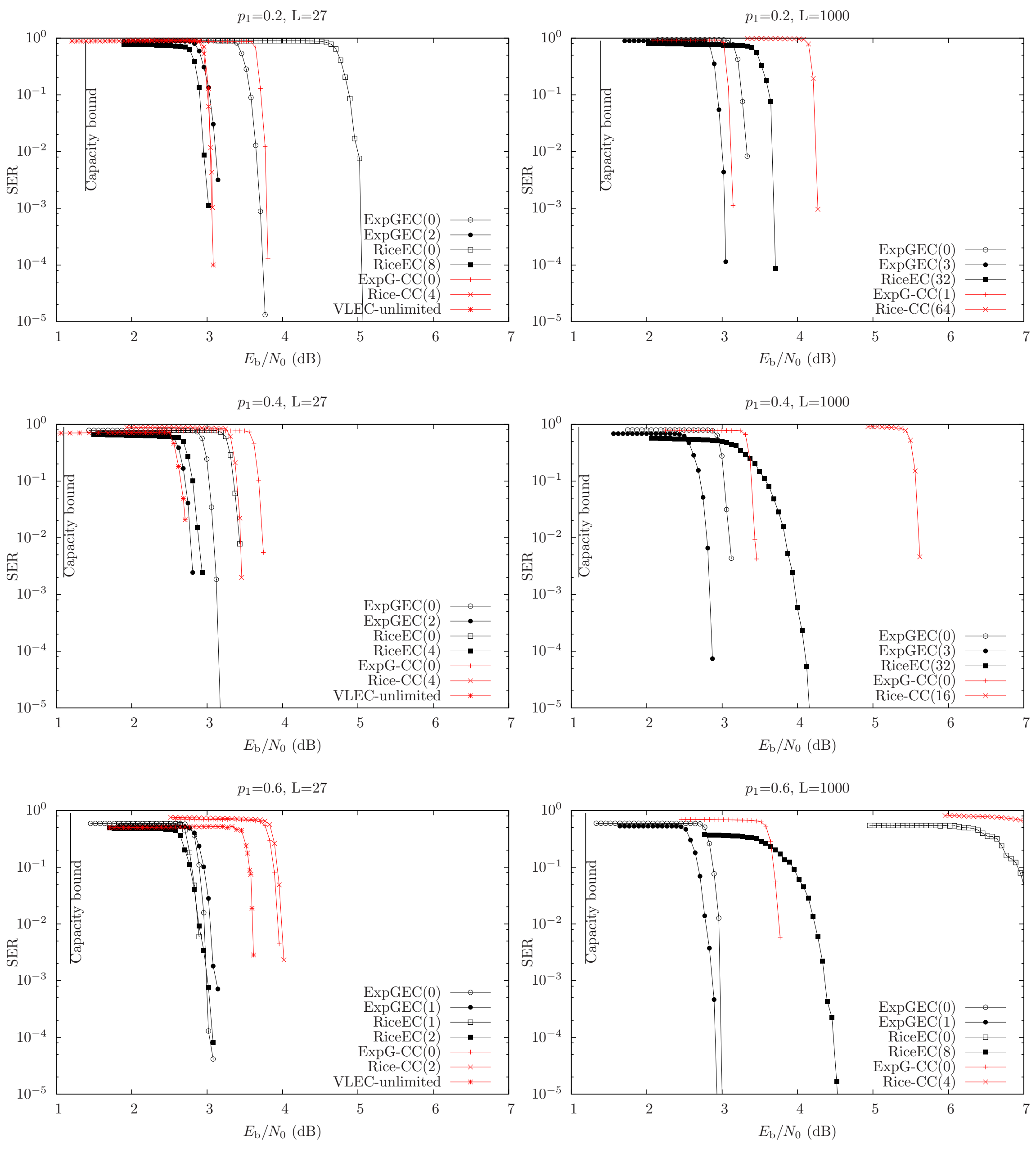


Figure 4.8: SER performance of the proposed schemes and benchmarkers listed in Table 4.2. Each scheme encodes an average of $a = 20000$ symbols per frame, which are generated using finite zeta-like probability distributions having different combinations of the parameters $L \in \{27, 1000\}$ and $p_1 \in \{0.2, 0.4, 0.6\}$. Each scheme uses QPSK modulation for communication over an uncorrelated narrowband Rayleigh fading channel. For each scheme, the adopted value of the parameter k_{ExpG} or M_{Rice} is listed in the legend within brackets. A complexity limit of 5000 ACS operations per bit input into the QPSK modulator is imposed on each scheme, except in the case of the VLEC benchmarker where the ultimate unlimited complexity performance is shown.

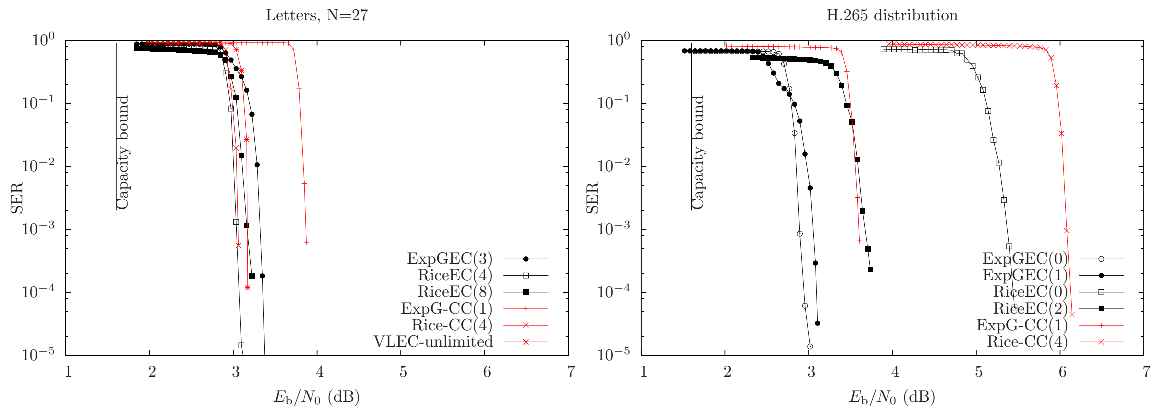


Figure 4.9: SER performance of the proposed schemes and benchmarkers listed in Table 4.2 when the source symbols obey the probability distribution of letters in the English alphabet, as well as when the symbols obey the probability distribution of a H.265 encoder. Each scheme encodes an average $a = 20000$ symbols per frame and uses QPSK modulation for communication over an uncorrelated narrowband Rayleigh fading channel. For each scheme, the parameter k_{ExpG} or M_{Rice} is listed in the legend within brackets. A complexity limit of 5000 ACS operations per bit input into the QPSK modulator is imposed on each scheme, except in the case of the VLEC benchmarker where the ultimate unlimited complexity performance is shown.

in a range of E_b/N_0 values where the EXIT chart of one sub-code has an open tunnel allowing iterative decoding convergence to the (1,1) point and a low SER for the corresponding sub-symbols, while the other sub-code has a closed tunnel leading to a high SER for the corresponding sub-symbols and resulting in a high SER overall. To combat this, unequal error protection [28] may be employed to increase the E_b/N_0 value where the first EXIT chart tunnel becomes open, allowing the E_b/N_0 value where the second EXIT chart tunnel opens to be reduced to the same value. This enables a low overall SER to be achieved at this lower E_b/N_0 value. For example, Figure 4.5 provides the EXIT charts for the unequal error protection of a particular parameterisation of the RiceEC scheme. In this case where $E_b/N_0 = 2.8$ dB, both the UEC and FLC-CC sub-codes are associated with marginally open EXIT chart tunnels, facilitating iterative decoding convergence to the (1,1) point and a low overall SER.

To be specific, unequal error protection is achieved by carefully selecting the inner puncturing or doping rates R_1^{i} and R_2^{i} that are associated with the UEC or FLC-CC sub-codes. Note that when R_1^{i} or R_2^{i} is reduced in order to increase the error protection for the corresponding sub-code, the other one of R_1^{i} or R_2^{i} must be increased, in order to maintain the same overall throughput η, according to (4.29). As an alternative to excessive doping, the error protection of the FLC-CC sub-code may be increased by increasing the codeword length of the $r_2 = 4$-state CC from $n_2 = 2$ to $n_2 = 3$ bits, according to the design of [6, Table II].

Most of the schemes characterized in Table 4.2 have puncturing or doping rates R^{i} that are close to 1, avoiding the performance degradation that is associated with excessive puncturing or doping. However, schemes such as the UEC benchmarker that results for the $M_{\mathrm{Rice}} = 1$ special case of the RiceEC scheme, have puncturing rates that are as high as $R^{\mathrm{i}} = 1.7$, which partly contributes to a high E_{b}/N_0 tunnel bound. More specifically, excessive puncturing results in EXIT charts with narrower open tunnels, resulting in a gradual SER improvement with E_{b}/N_0, rather than a steep turbo cliff. Secondly, excessive puncturing negatively impacts the threshold E_{b}/N_0 value where a marginally open EXIT chart tunnel is created, leading to more capacity loss. Furthermore, a punctured code will also have a decoding complexity disadvantage, since the punctured bits must be decoded alongside the transmitted bits.

4.5.4 SER performance

This section compares the SER performance of the proposed RiceEC and ExpGEC schemes to that of the benchmarkers for each of the scenarios listed in Table 4.2. The SER performance of these schemes is characterized in Figures 4.8 and 4.9, where a complexity limit of 5000 ACS operations per QPSK input bit is imposed, as it was characterized per decoding iteration in Table 4.2. This complexity limit was chosen in all scenarios considered, since we found that it only marginally impacts the SER performance of whichever scheme converges to its ultimate unlimited performance with the lowest complexity in each scenario. The employment of this criterion for selecting the complexity limit ensures that excessively high-complexity schemes are not favored over those which have a marginally worse SER performance but lower complexity. Due to the considerable complexity of the trellis employed in the VLEC benchmarker, it performs poorly when this complexity limit is imposed, even for the case of $L = 27$. Owing to this, Figures 4.8 and 4.9 characterize the SER of the VLEC benchmarker when the complexity limit is removed, in order to characterize its ultimate performance, although this is only achieved at the cost of potentially excessive complexity.

Figures 4.8 and 4.9 show that our family of schemes offer the best SER performance in all of the considered scenarios. In the finite zeta-like distribution scenarios having $p_1 = 0.6$, the best of the proposed schemes offers around 1 dB of gain compared to the best SSCC benchmarker for both considered values of L. Likewise, the proposed schemes offer around 0.5 dB of gain for $p_1 = 0.4$, as well as around 0.75 dB of gain for the H.265 distribution. For $p_1 = 0.2$ and for the English letters distribution however, the proposed schemes offer only a marginal SER performance gain over the Rice-CC benchmarker. Note however that this benchmarker suffers from poor performance in other scenarios, which prevents its general applicability. Note that the gains offered by the proposed schemes are achieved ‘for free,’ since they are achieved without increasing the required decoding complexity, or transmission-energy, -bandwidth, or -duration. The unlimited complexity VLEC benchmarker offers an SER performance very close to the proposed schemes for $p_1 \in \{0.2, 0.4\}$ and $L = 27$, however it should be noted that its complexity is more than an order of magnitude greater than that of the proposed schemes. Furthermore, the complexity of the VLEC benchmarker becomes impractical for values of L significantly greater than 27.

At higher values of p_1, lower values of the parameters of k_{ExpG} and M_{Rice} give higher coding rates R_1^{o} and R_2^{o}, enabling higher effective throughputs η without requiring excessive puncturing. This facilitates better SER performance than may be achieved using higher values of k_{ExpG} and M_{Rice} at these p_1 values. For example, in the scenario where we have $L = 27$ and $p_1 = 0.2$, the $k_{\text{ExpG}} = 2$ parameterisation of the ExpGEC scheme outperforms the $k_{\text{ExpG}} = 0$ parameterisation by 0.5 dB. By contrast, when $p_1 = 0.6$, the $k_{\text{ExpG}} = 0$ parameterisation offers a marginal improvement over the $k_{\text{ExpG}} = 1$ parameterisation. This is also the case in the RiceEC, where the $M_{\text{Rice}} = 8$ parameterisation offers the best performance at $p_1 = 0.2$ and $L = 27$, while the $M_{\text{Rice}} = 1$ parameterisation offers the best performance at $p_1 = 0.6$. In the case where $L = 1000$, the coding rates R_1^{o} and R_2^{o} of the RiceEC scheme are lower than those of the ExpGEC, requiring more puncturing to meet the effective target throughput η. Owing to this, the ExpGEC schemes are superior to the RiceEC scheme in these cases. By contrast, in the case where we have $L = 27$, the coding rates R_1^{o} and R_2^{o} of the RiceEC scheme are similar to those of the ExpGEC, with neither schemes requiring any significant puncturing or doping. This allows the RiceEC to provide similar performance to the ExpGEC when $L = 1000$.

4.6 Conclusions

This chapter extended and generalized the EGEC code of [28] to give the proposed ExpGEC code. Similarly, the UEC code of discussed in Section 2.5 and Chapter 3 has been extended to design the proposed RiceEC code. Both these novel codes facilitate operation over a significantly wider range of source symbol distributions. This chapter has focused on the scenario where the cardinality L of the source symbol set is finite, allowing comparison to a VLEC benchmarker. This chapter has shown that the proposed schemes achieve the same SER performance as the VLEC benchmarker, but at an order of magnitude lower complexity when we have $L = 27$. Furthermore, our proposed schemes maintain a moderate complexity for significantly higher L values, while that of the VLEC benchmarker may become excessive. Furthermore, we have proposed a technique for increasing the practicality of the proposed schemes, which allow fixed-length designs to be employed for all interleavers. We have shown across a wide range of application scenarios that our family of proposed schemes outperforms several SSCC benchmarkers by as much as 1 dB with no cost in terms of decoding complexity, transmission-energy, -bandwidth or -duration.

4.6.1 Design guidelines

When designing ExpGEC, RiceEC and similar JSCC schemes, the following design guidelines are recommended.

1. Characterise the probability distribution of the source symbols, in order to aid the selection of the most appropriate source code to comprise the JSCC scheme.
2. Using coding rate plots of the candidate source codes, identify schemes which maintain a suitably high coding rate over the source symbol probability range.
3. Likewise, use the inverted EXIT chart area plots of the candidates to ensure that there is not significant capacity loss over the source symbol probability range.
4. Use EXIT chart matching to determine the best inner code design to use with the trellis decoder and CC decoder.
5. Since the RiceEC and ExpGEC is comprised of two separate parts, use EXIT chart analysis to derive the puncturing rate for each part, so that the two parts achieve a turbo cliff at the same SNR.
6. For a given effective throughput η, select schemes which do not have too much puncturing or doping, which will negatively affect performance.
7. Finally, run SER simulations to confirm the candidate codes match the predictions made by the EXIT matching. Here, a complexity limit can be applied if desired.

Chapter 5

A turbo decoder architecture for wireless sensor networks

5.1 Introduction

The authors of [109–111] have predicted the ubiquity of Wireless Sensor Networks (WSNs) and the 'Internet of Things' (IoT) for the monitoring and control of residential and commercial environments. These systems are typically required to maintain reliable wireless communications over extended periods of time, while relying only on scarce energy resources. The life-span of these energy-constrained wireless communication applications can be significantly extended by employing the sophisticated iterative receiver techniques which have been explored throughout this thesis, in order to reduce the required transmission energy. More specifically, it was argued in [112] that error correction codes can be used for redistributing the energy consumption from the energy-constrained sensor nodes to the central data fusion node, which typically has access to more plentiful energy resources. More specifically, the employment of error correction encoding reduces the transmit energy required by the sensor nodes in order to maintain reliable communication. While the use of error correction decoding increases the processing energy consumption at the receiver, this is acceptable since the receiver will often be unconstrained by the available energy, especially at the uplink receiver fed by the mains. This energy redistribution concept was extended in [17] to include multi-hop networks, where jointly considering the transmit energy and processing energy at each hop can lead to a significant overall reduction in energy consumption. As a benefit of this, life-spans of the order of years are potentially facilitated in applications having low

This chapter is partially based on the following publication.
M. F. Brejza, R. G. Maunder, B. Al-Hashimi and L. Hanzo. Flexible Iterative Receiver Architecture for Wireless Sensor Networks: A Joint Source and Channel Coding Design Example. Accepted by *IET Wireless Sensor Systems.*

duty-cycles, low throughputs and multiple short-range hops. However, in these scenarios, the iterative receiver processing techniques are associated with a significant portion of the overall energy dissipation. Furthermore, the available processing resources of WSN nodes remain limited at the time of writing, hence preventing the application of iterative decoding techniques. As a result, most previous publications on WSNs have considered non-iterative Reed-Solomon (RS) [113] and convolutional codes [114]. Furthermore, the IEEE 802.15.4 standard [115] does not include any channel coding in its PHY layer [114], exemplifying the absence of error correction codes in existing WSNs. Therefore, further significant life-span extensions would be facilitated, if this processing energy dissipation could be reduced.

Previous work has proposed Application-Specific Integrated Circuit (ASIC) designs for a variety of iterative receiver techniques, including synchronization [116], channel estimation [117], equalization [118], bit-to-symbol demapping [94], turbo decoding [1], Low-Density Parity-Check (LDPC) decoding [2] and source decoding [91]. Despite this, there are only a handful of papers that propose iterative decoding architectures specifically designed for wireless sensor networks. In order to facilitate the lowest possible transmission energies, an iterative receiver can benefit from combining several of these techniques. While it would be possible to combine several of these different ASIC decoders into a single System on Chip (SoC), this would not produce the most energy and chip-area efficient design. This is because only one of the various decoders in an iterative receiver is operated at a time, with all the others remaining idle until their turn is reached. Furthermore, the iterative decoding complexity can be minimized by performing more iterations of the less complex techniques and fewer iterations of the complex ones [27]. Idle hardware may be deemed to waste chip area, whilst its static energy consumption will waste energy, particularly for smaller technology scales [119]. This motivated the energy-efficient turbo decoder ASIC of [51], and the LDPC ASIC of [120], both of which efficiently exploit their hardware resources, in order to minimize energy dissipation. However, neither of these architectures supported any of the other iterative receiver techniques that are listed above.

Against this background, this chapter proposes a programmable ASIC, which can be used for implementing a wide variety of iterative receiver techniques, while maintaining a low decoding energy dissipation. More specifically, this ASIC is designed for the efficient exploitation of its computational resources, regardless of its particular application, in order to minimize both the energy dissipation and area wastage. Hence this ASIC is particularly suitable for WSN applications, where having a low energy consumption and a low chip area are more important than achieving a high throughput. As shown in Figure 1.4, this chapter considers the hardware implementation of a Joint Source and Channel Coding (JSCC) scheme. In particular, the architecture proposed in this chapter is characterised using a JSCC scheme similar to that used in Chapter 3, that iteratively activates a Quadrature Phase Shift Keying (QPSK) demodulator [94], a turbo decoder and a Unary Error Correction (UEC) based joint source and channel decoder [121]. In contrast to the family of source codes [122, 123] previously designed for WSN applications, the UEC code, described in Section 2.5, improves the error correction capability and it is designed to have a low complexity. In this chapter, a turbo decoder is employed to target applications requiring high throughputs, such as image or video transmission [124]. Furthermore, the UEC code is employed since it is well suited for encoding the symbols produced by a H.264 or H.265 [97] video encoder, as may be employed in camera-based sensor networks. Furthermore, a high throughput WSN architecture may also be invoked for applications where a receiver may have to decode frames generated by hundreds of low-throughput sensor nodes. The novel contributions of this chapter are as follows.

- This chapter presents the first hardware implementation of a UEC scheme [6], where the proposed architecture is applied to a scheme comprising a UEC decoder, a parallel concatenation of two URC decoders and an iterative demodulator, similar to that of Chapter 3.
- The proposed architecture is comprised of a set of general-purpose Calculation Units (CUs). This chapter extends the CUs of the previous work [51], so that they can perform the basic operations common to a wider range of iterative decoders, under the instruction of the controller.
- In this way, the same hardware is used for processing the above-mentioned four different decoders in the receiver.
- A novel controller has been developed for handling the scheduling of the four different decoders, and for ensuring that the hardware is kept as busy as possible, in order to achieve a high utility and hardware efficiency. Since the controller can be programmed to implement any particular combination of iterative receiver techniques, this allows the operation of the decoder to be changed, without requiring the ASIC to be redesigned.

This chapter commences by introducing the proposed programmable architecture in Section 5.2. Section 5.3 uses the proposed architecture for implementing the UEC application example. Section 5.4 characterizes the utility, energy consumption, throughput and chip area of the proposed architecture, while comparing these with previous architectures. Finally, Section 5.4 offers the concluding remarks.

5.2 Programmable ASIC architecture

This section details the proposed programmable architecture. Firstly, in Section 5.2.1, a brief outline of the Logarithmic Bahl-Cocke-Jelinek-Raviv (Log-BCJR) algorithm is provided, which was previously described on Section 2.1.5. Here, we will focus on the characteristics of the algorithm which impact on the choices made for the hardware architecture. The architecture proposed for implementing iterative receiver processing and the Log-BCJR algorithm will then be described in Section 5.2.2, which also discusses the specific features of the architecture that allow it to be flexible in its operation.

5.2.1 The BCJR algorithm

Iterative receivers operate on the basis of an iterative exchange of soft information between the various receiver components, which may perform synchronization [116], channel estimation [117], equalization [118], demapping [94], turbo decoding [1], LDPC decoding [2] and source decoding [91] for example. This soft information expresses not only *what* the most likely value of each transmitted bit is, but also *how* likely that bit value is. Therefore, Soft-In Soft-Out (SISO) versions of the various receiver components are required for converting the soft inputs into soft outputs. The Log-BCJR is a flexible SISO algorithm, which can be employed as the basis of the above-mentioned iterative receiver components.

As described in Section 2.1.5, the Log-BCJR algorithm [17] is a reduced-complexity variant of the Bahl-Cocke-Jelinek-Raviv (BCJR) that is particularly well-suited for implementation. Rather than operating on the basis of bit likelihoods having a high dynamic range, the Log-BCJR algorithm processes the Logarithm of bit Likelihood Ratios (LLRs), which have a low dynamic range and can be readily represented using fixed-point arithmetic. Furthermore, the additions and multiplications of the BCJR algorithm are converted to the reduced-complexity max* and addition operations in the Log-BCJR algorithm.

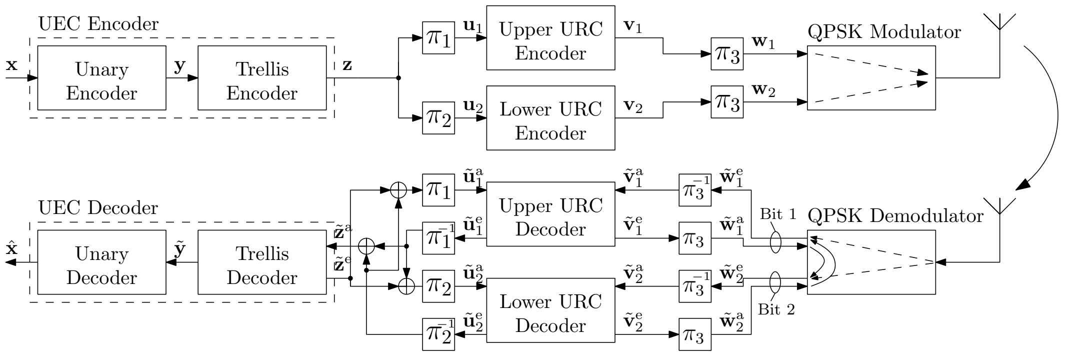


Figure 5.1: The UEC-Turbo-QPSK scheme schematic.

The Log-BCJR accepts vectors of *a priori* LLRs as its input and generates vectors of higher-quality extrinsic LLRs as its output. For example, the upper URC decoder of Figure 5.1 operates on the basis of the Log-BCJR algorithm, having the *a priori* LLR vectors $\tilde{\mathbf{u}}_1^{\mathrm{a}}$ and $\tilde{\mathbf{v}}_1^{\mathrm{a}}$ as its inputs, while generating the extrinsic LLR vectors $\tilde{\mathbf{u}}_1^{\mathrm{e}}$ and $\tilde{\mathbf{v}}_1^{\mathrm{e}}$ as its outputs. The Log-BCJR algorithm operates on the basis of a trellis, which comprises r number of states for each bit. The states corresponding to a particular bit are connected to the states of the next bit using T number of legitimate trellis-transitions. These states and transitions describe the logical constraints and relationships between the consecutive transmitted bits, that are imposed by the corresponding iterative receiver component. The Log-BCJR algorithm converts the *a priori* LLRs into the higher-quality extrinsic LLRs using a sequence of four intermediate steps, as detailed in Section 2.1.5.

Firstly, a set of α values is calculated for each state. Each α value depends on the value of *a priori* LLRs of the corresponding bit, as well as on some α values from the previous bit, depending on the specific transitions in the trellis. Owing to the dependences between the α values of consecutive bits, they must therefore be calculated in a forward-recursive manner along the trellis. This is achieved using low-complexity addition and max* operations.

Secondly, the β values are calculated in a similar manner to the α values. However, instead of depending on the previous bit, each β value depends on some β values from the next bit. Hence the latter values are calculated using a backward recursion along the trellis.

The penultimate step before generating the output is to calculate a δ value for each transition between each bit in the trellis. These are calculated using additions of the α, β and *a priori* LLR values related to the corresponding bit. In the fourth step, extrinsic Logarithmic Likelihood Ratios (LLRs) can then be generated from the set of δ values, using low-complexity subtraction and max operations.

The optimal Log-BCJR algorithm uses the max* operation, where $\max^*(A,B) = \max(A,B) + \ln(1+e^{-|A-B|})$. As discussed in Section 2.1.6, the Log-BCJR can be transformed into the reduced complexity Maximum Log-BCJR (Max-Log-BCJR) by approximating $\max *(A,B)$ as simply $\max(A,B)$. The Log-BCJR algorithm reduces the transmission energies required for maintaining an adequate SNR at the receiver for near-error-free communication by about 0.5 dB–1 dB, at the cost of a higher complexity. The exact savings depend on the design of the iterative decoding scheme [25]. However, extrinsic scaling [25] may be used in conjunction with the Max-Log-BCJR algorithm, yielding the Maximum with Scaled Extrinsic Log-BCJR (Max-SE-Log-BCJR), in order to close this performance gap with respect to the Log-BCJR algorithm. The elaborate a little further, extrinsic scaling multiplies the extrinsic LLRs output from the Max-Log-BCJR decoders by a constant in the range of 0.6–0.8. This scaling represents the reduced confidence in the extrinsic LLRs, which is imposed by the sub-optimal use of the Max-Log-BCJR algorithm. The Max-SE-Log-BCJR algorithm is adopted in the decoder of Section 5.2.2, which results in a capable yet low-complexity design, having both a low energy consumption as well as a small chip area, as required for WSN applications.

5.2.2 The decoder top level

The decoder described in this and the following sections has been designed based on the nature of the Max-SE-Log-BCJR algorithm. As described in Section 5.2.1, the only operations required by the Max-SE-Log-BCJR algorithm are addition, subtraction and max, where the max operation can be broken down to a subtraction, compare and select operation. Note that the multiplication required by extrinsic scaling can be implemented using shifting and addition operations, as described below. The fundamental Add-Compare-Select (ACS) operations are also the fundamental operations of the Max-Log-BCJR required for other iterative decoding algorithms, such as the min-sum algorithm of LDPC decoders [2].

The grade of parallelism facilitated for the Max-SE-Log-BCJR algorithm is limited by the forward and backward recursions, which impose data dependencies between the α and β values of neighbouring trellis stages. However, there are no data dependencies between the r number of α and β values *within* each trellis stage, hence facilitating a parallel processing opportunity as determined by the number of trellis states r. This motivates the conception of a processing architecture comprised of r parallel CUs, where each CU is designed for performing the ACS operations employed by the Max-Log-BCJR algorithm. Note that this architecture could be readily adapted to the scenario, where fewer parallel CUs are used. This would support applications that have a lower throughput requirement in order to benefit from a reduced chip area. More specifically, when operating with fewer CUs, each CU would have to process multiple states for each trellis stage. This leads to a small overhead of including extra registers in each CU to store intermediate values for multiple states, rather than just one state. Furthermore, some additional multiplexers would be required to choose between these registers, depending on which state is being decoded. While the logic area will scale with the number of CUs, the memory requirement will remain the same.

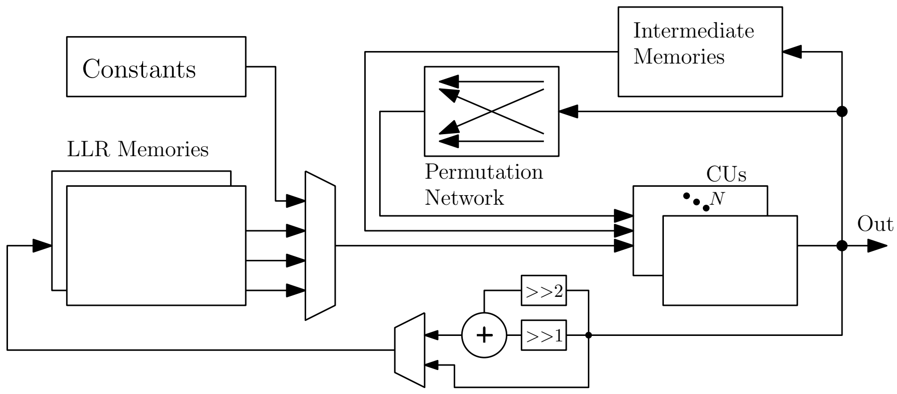


Figure 5.2: Top-level schematic of the proposed iterative receiver processing architecture.

This novel approach of designing a decoder processor architecture facilitates a Single Instruction, Multiple Data (SIMD) like approach for exploiting the parallelism of the flexible decoder. Figure 5.2 shows the top level schematic of the proposed iterative receiver processing architecture, which is centred around the group of r parallel CUs. The output generated by each CU is input to a permutation network, if it is to be used as an input to another CU in the next clock cycle. This permutation network allows a CU to pass its calculated values to any other CU, according to the specific requirements imposed by the transitions between states in the trellis. By contrast, if the output generated by a CU is to be used in a later clock cycle, then it is stored in intermediate memory until needed, whereupon the permutation network may be activated to deliver it to the correct CU. For example, the α values generated during the forward recursion are stored in the intermediate memory until used during the backward recursion as part of the δ calculations. During the backward recursion, the extrinsic LLRs are generated and stored in the LLR memories. If extrinsic scaling is used, the LLRs can be multiplied by 0.75 before being stored. This may be achieved at a low hardware cost by using a single adder and two fixed bit shifters, as shown in Figure 5.2.

Note that storing all α values generated during a forward recursion of the entire trellis would require a large amount of intermediate memory. One technique for reducing this memory requirement is Previous Iteration Value Initialization (PIVI) [44], which decomposes the trellis into a number of shorter processing windows, which are processed separately. This reduces the intermediate memory requirement to a value that is proportional to the window length, rather than to the trellis length. The unknown α and β values at the ends of each window are initialized using the α and β values that were obtained at the adjacent ends of the neighbouring windows during the previous decoding iteration. This requires additional PIVI memory for storing the boundary values between iterations. In the proposed architecture, only the boundary β values for the three separate decoding blocks have to be stored. By contrast, the boundary α values do not need storing, since each window is calculated sequentially in increasing order, so that boundary α values can be passed between windows as they are calculated. However, in addition to the PIVI memory, an α memory is required for storing all of the α values calculated during the forwards recursion of each window. Therefore, in the proposed PIVI windowing architecture, the total number of values which need storing is given by $3rN/\omega$ for PIVI, together with $r\omega$ values for the forward α recursion. Here, r is the number of states in the trellis, ω is the window length and N is the frame length. By comparison, when dispensing with windowing as well as with PIVI and the entire α forwards recursion has to be stored in memory, the total number of state metrics that would have to be stored is rN, which is many times higher. Note that Ilnseher and Kienle [13] describes a 'Re-Computation' method, which stores only every sixth α value during the forwards recursion. This reduces the amount of memory required for storing state metrics, albeit at the cost of requiring extra hardware for recomputing the values, which were not stored when they are later required.

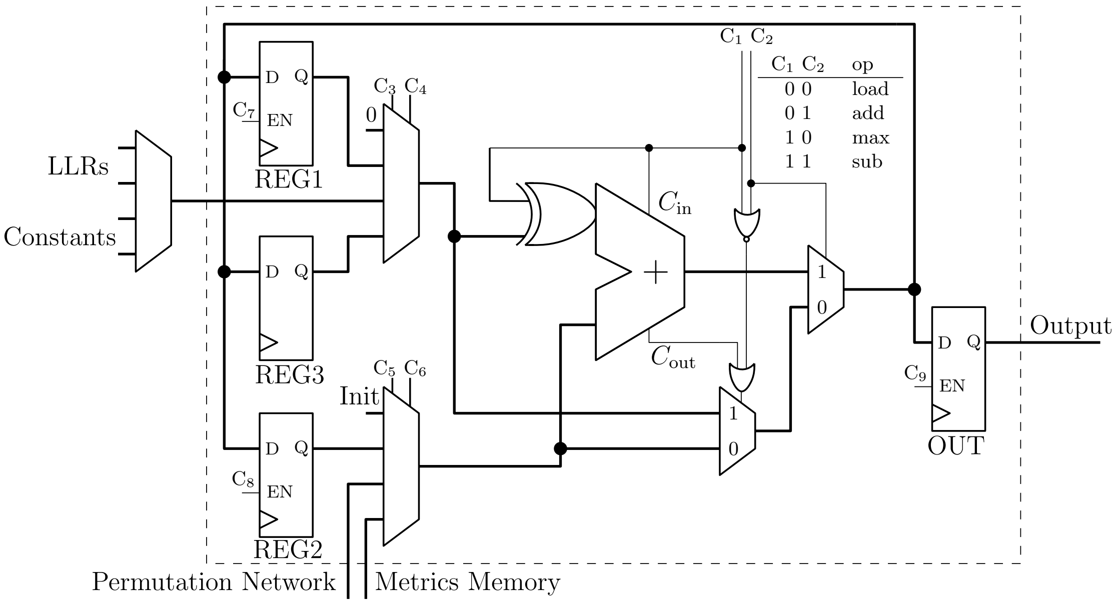


Figure 5.3: The block diagram of a CU.

5.2.3 The Computation Unit

Figure 5.3 shows the internal schematic of the proposed CUs, which performs the ACS operations required by the Max-Log-BCJR algorithm. In each clock cycle, a CU can undertake either an addition, a subtraction or a max operation. Alternatively, it can perform a load instruction, which does not perform any calculation operations, but allows the data to be loaded into the CU's registers for use in subsequent clock cycles. The architecture owes its flexibility partly to the ability for all the parts of the algorithm to be undertaken in a single low-complexity block, which is in contrast to the majority of previous work [12, 25], where each part of the architecture is dedicated to a single function. This characteristic of WSN applications, which requires a low energy consumption and a small chip area, rather than targeting high throughputs, like the suite of existing architectures. The core of the CU is the adder of Figure 5.3, which is reused by all the addition, subtraction and max operations. Here a subtraction is carried out by using an XOR gate to invert one of the adder inputs and then using the carry in for completing the subtraction. The max operation is carried out by performing a subtraction and using the adder's carry out C_{out} for selecting the appropriate input, as shown in Figure 5.3.

The CU of Figure 5.3 includes three data storage registers provided for storing intermediate results between clock cycles, namely REG1, REG2 and REG3. This number of intermediate data storage registers offers an attractive tradeoff between the chip area, processing throughput and flexibility. Furthermore, the register OUT is used for holding the output, while it is being loaded into the memories or permutation network of Figure 5.2. In total, nine control signals, namely C_1-C_9 are used for controlling the registers, multiplexers and the operational mode of the CU, as shown in Figure 5.3.

Note that the proposed CU can be readily extended to perform other operations for different algorithms. For example, the proposed CU could implement other max* approximations [125], such as the Look-Up Table (LUT) based max* operation by replacing the max operation in the existing datapath by the circuitry required for the LUT max* operation. If this operation is implemented such that it can be completed within a single clock cycle, then no modification is required for the controller. However, if a multi-cycle LUT max* circuit was employed, such as the LUT max* circuit used in our previous work [51], then the controller would also have to be modified for scheduling these four steps. Furthermore, the datapath of the flexible CUs could be modified to perform the min* (a.k.a. boxplus) operations [126], which would allow these CUs to form the basis of a flexible LDPC decoder.

5.2.4 Controller

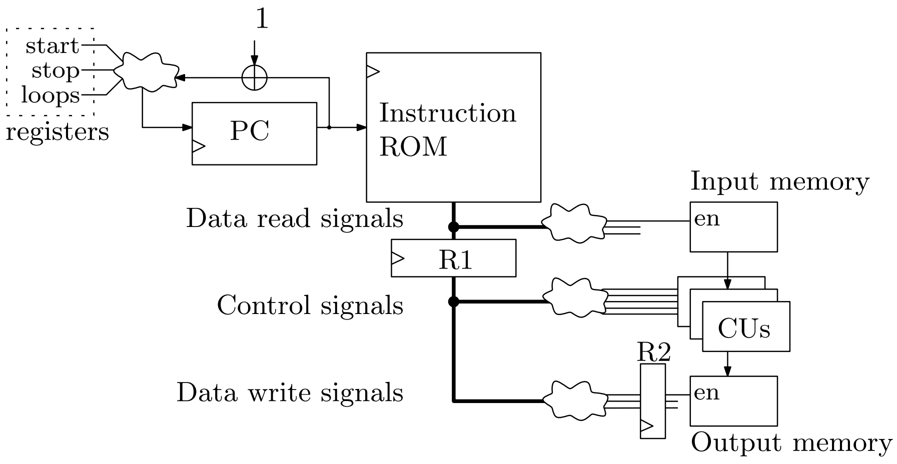


Figure 5.4: Controller Block Diagram

The controller's block diagram is shown in Figure 5.4, which has to schedule the operations undertaken by the CUs, in order to successfully implement the Max-Log-BCJR algorithm. In contrast to the approach of most other decoder architectures, the controller of the proposed architecture is designed to be programmable, so that it can flexibly carry out the iterative operations of the decoders of different types, as will be demonstrated in Section 5.3. In order to maximize the processing throughput and to minimize the energy consumption of the proposed architecture, it is necessary for the controller to maximize the utility of the CUs. Since the Max-Log-BCJR algorithm is comprised of many identical calculations that are performed using different data for different states of the trellis, a SIMD approach to the control of the CUs is motivated. This has the advantage of minimizing the complexity overhead of the controller, since it is not required to control the CUs individually. The proposed controller can either issue a single instruction to all CUs, or it can decompose the CUs into two groups of any size and simultaneously issue two different instructions to these.

As shown in Figure 5.4, the proposed controller is based around an instruction memory, which can be loaded with sequences of instructions that control the operation of each stage of the Max-Log-BCJR algorithm, for each of the concatenated decoders. A program counter (PC) is used for addressing the successive instructions of the current sequence within the instruction memory, resembling the instruction fetching behaviour of a simple processor. As shown in Figure 5.4, looping around and jumping between the sequences of instructions stored in the instruction memory is achieved by loading the PC with values loaded from the looping registers. These store the address of the start and end of each instruction sequence, as well as the number of times that each sequence should be looped over, before jumping to the next sequence.

The control signals of the CUs in the decoder are derived directly from the bits in the current instruction, with some added pipeline delays, labelled as R1 and R2 in Figure 5.4. These are required for matching the delays imposed by reading and writing to both the LLR and to the metrics memories of Figure 5.2. This results in a low-complexity, yet flexible controller, which can perform one of the operations described in Section 5.2.3 in every clock cycle, leading to a high utilization of the hardware. This avoids any processing throughput penalty as compared to the approach of [51], where the controller was implemented as a fixed state machine.

5.3 Application to a joint source coding, channel coding and modulation scheme

This section demonstrates the employment of the proposed architecture for implementing the receiver of a joint source coding, channel coding and modulation scheme. This application includes a variety of different receiver components, which employ different versions of the Max-SE-Log-BCJR algorithm, exemplifying the flexibility of the proposed programmable ASIC architecture. This example employs the UEC code for joint source and channel coding. As described in Section 2.5, the UEC code is particularly suited to zeta-distributed integer values in the range of one to infinity, which are produced by sources obeying Zipf's law [127]. This models the symbols generated by a wide variety of physical processes, such a H.265 video encoder [6], for example. However, the flexibility of the proposed architecture allows for other iterative decoder blocks to be used, depending on the specific requirements of the WSN system. In particular, the flexibility of the architecture allows it to implement the classic systematic turbo code, as used in mobile telephony. Section 5.3.1 commences by detailing the transmitter design, in order to aid the description of the receiver design in Section 5.3.2. The mapping of this receiver design onto the proposed programmable ASIC architecture is detailed in Section 5.3.3.

5.3.1 Transmitter design

The transmitter of the joint source coding, channel coding and modulation scheme is shown in Figure 5.1. This scheme is similar to the scheme used in Chapter 3, and is designed for conveying a stream $\mathbf{x} = [x_i]$ of zeta-distributed source symbols. The stream of symbols $\mathbf{x}$ are input to the unary encoder, which outputs a stream of bits. More specifically, for each input symbol x_i, the unary encoder outputs the bit vector Unary(x_i) according to Table 2.6. These unary encoded bits are then partitioned into bit vectors $\mathbf{y} = [y_k]_{k=1}^{N}$ having length N, using the technique described in Section 4.4. Following this, an $r = 8$-state UEC trellis encoder is employed for transforming the bit vector $\mathbf{y}$ into the bit vector $\mathbf{z} = [z_k]_{k=1}^{N}$, as described in Section 2.5. When encoding each bit vector, the UEC encoder is reset to its default initial state $m_0 = 1$. The UEC encoder sequentially processes each bit in the vector $\mathbf{y}$, traversing to a new state m_k, and outputting a bit z_k. Here, the $r = 8$-state trellis of Figure 2.23 is used by the UEC encoder, with the codewords $\mathbb{C} = \{0, 1, 1, 1\}$ Following UEC encoding, two copies of $\mathbf{z}$ are interleaved by the interleavers π_1 and π_2 to obtain the bit vectors $\mathbf{u}_1 = [u_{1,k}]_{k=1}^{N}$ and $\mathbf{u}_2 = [u_{2,k}]_{k=1}^{N}$. Turbo channel encoding is performed by a pair of parallel concatenated $r = 8$-state Unity Rate Convolutional (URC) encoders [27] and the resultant bit vectors $\mathbf{v}_1 = [v_{1,k}]_{k=1}^{N}$ and $\mathbf{v}_2[v_{2,k}]_{k=1}^{N}$ are interleaved by a pair of interleavers having the same design π_3. The bits in the resultant vectors $\mathbf{w}_1 = [w_{1,k}]_{k=1}^{N}$ and $\mathbf{w}_2 = [w_{2,k}]_{k=1}^{N}$ are paired up and QPSK modulated onto the channel using a natural mapping, which motivates iterative demodulation in the receiver and improves the attainable error correction performance [94].

5.3.2 Receiver design

The receiver designed for this scheme is shown in Figure 5.1, which closely mirrors the transmitter design, except that it processes vectors of LLRs, rather than vectors of bits. Furthermore, these vectors of LLRs are iteratively exchanged bidirectionally between the demodulator, the URC decoders and the UEC decoder, via the interleavers $\pi_1 - \pi_3$ and the deinterleavers $\pi_1^{-1} - \pi_3^{-1}$.

Upon reception of a transmission, the QPSK demodulator outputs the pair of extrinsic LLR vectors $\tilde{\mathbf{w}}_1^{\mathrm{e}} = [\tilde{w}_{1,k}^{\mathrm{e}}]_{k=1}^{N}$ and $\tilde{\mathbf{w}}_2^{\mathrm{e}} = [\tilde{w}_{2,k}^{\mathrm{e}}]_{k=1}^{N}$, which pertain to the corresponding bit vectors $\mathbf{w}_1$ and $\mathbf{w}_2$ in the transmitter. These extrinsic LLRs are obtained by combining the information received from the channel with any information available within the *a priori* LLR vectors $\tilde{\mathbf{w}}_1^{\mathrm{a}} = [\tilde{w}_{1,k}^{\mathrm{a}}]_{k=1}^{N}$ and $\tilde{\mathbf{w}}_2^{\mathrm{a}} = [\tilde{w}_{2,k}^{\mathrm{a}}]_{k=1}^{N}$ provided by the URC decoders, as shown in Figure 5.1. More specifically, the demodulator algorithm [94] generates the first extrinsic LLR vector $\tilde{\mathbf{w}}_1^{\mathrm{e}}$ by combining the information received from the channel with the second *a priori* LLR vector $\tilde{\mathbf{w}}_2^{\mathrm{a}}$. Likewise, the demodulator generates the second extrinsic LLR vector $\tilde{\mathbf{w}}_2^{\mathrm{e}}$ by combining the information received from the channel with the first *a priori* LLR vector $\tilde{\mathbf{w}}_1^{\mathrm{a}}$. This relationship is shown in the circuit of Figure 5.1 and it is exploited by the hardware mapping of Section 5.3.3 for reducing the number of memory accesses required.

The UEC trellis decoder and the URC decoders apply the Max-Log-BCJR algorithm to their $r = 8$-state trellises, as previously described in Section 5.2.1. However, the URC decoders operate on the basis of a different trellis design from that of the UEC trellis decoder. More specifically, trellis-transitions connect each state in the URC trellis to two other states, whereas different states are connected to different numbers of other states in the UEC trellis, as shown in Figure 2.23. A further difference is that the knowledge of the symbols' zeta probability distribution can be incorporated into the α and β calculation of the UEC trellis decoder, in order to enhance its error correction capability [6].

As shown in Figure 5.1, the URC decoders convert the *a priori* LLR vectors $\tilde{\mathbf{u}}_1^{\mathrm{a}}$, $\tilde{\mathbf{v}}_1^{\mathrm{a}}$, $\tilde{\mathbf{u}}_2^{\mathrm{a}}$ and $\tilde{\mathbf{v}}_2^{\mathrm{a}}$ into the extrinsic LLR vectors $\tilde{\mathbf{u}}_1^{\mathrm{e}}$, $\tilde{\mathbf{v}}_1^{\mathrm{e}}$, $\tilde{\mathbf{u}}_2^{\mathrm{e}}$ and $\tilde{\mathbf{v}}_2^{\mathrm{e}}$. Similarly, the UEC trellis decoder converts the *a priori* LLR vector $\tilde{\mathbf{z}}^{\mathrm{a}}$ into the extrinsic LLR vector $\tilde{\mathbf{z}}^{\mathrm{e}}$. The *a priori* LLR vector input to each of the URC decoders and to the UEC trellis decoder are obtained as the sum of the extrinsic LLR vectors provided by the other two decoders, as shown in Figure 5.1. At the start of the iterative decoding process, all *a priori* LLR vectors are initialized with zero-valued LLRs and iterative decoding continues until all of these vectors have been updated a certain number of times. At this point, the UEC trellis decoder inputs $\tilde{\mathbf{y}}$ to the unary decoder, which carries out the final hard decision and outputs the final received symbols $\hat{\mathbf{x}}$, as described in Section 2.5.4. Since the bit representation of each symbol varies in length, there may be trailing bits in the vector owing to the partitioning of the bit-vector $\mathbf{y}$ into fixed length parts. Here, these trailing bits represent an incomplete symbol, which may be stored in the unary decoder, until the next vector is received. The total number of symbols in each vector is sent as side information to assist the decoder, which also ensures that the decoded symbols in the receiver stay synchronized to the transmitter.

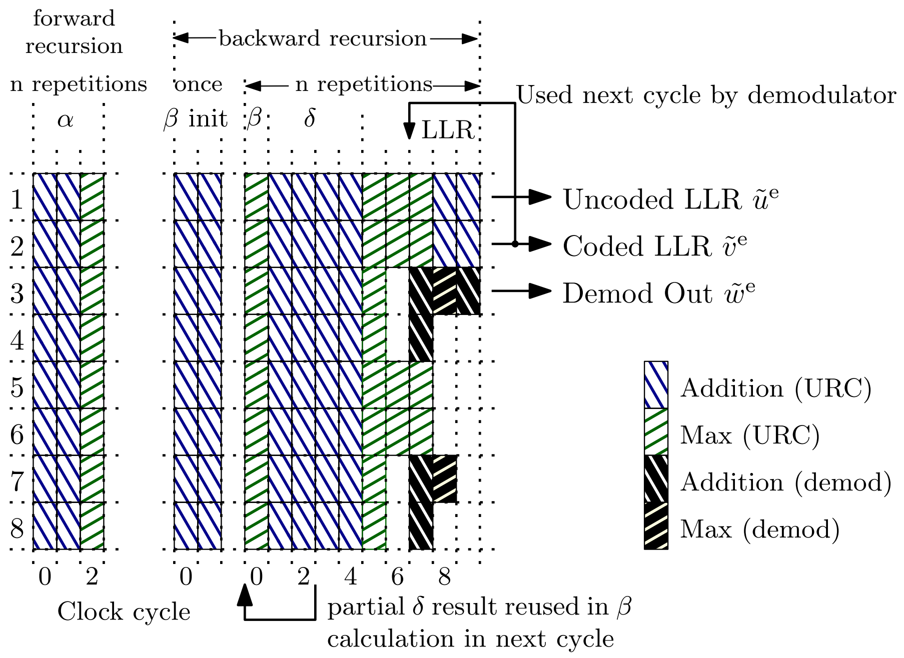
5.3.3 Mapping of the receiver design to the proposed architecture

This section describes how the decoder architecture described in Section 5.2.2 can be employed for implementing the receiver described in Section 5.3.2. The proposed implementation supports interleaver lengths of up to $N = 6144$ bits, in conjunction with UEC and URC window lengths of $\omega = 64$ trellis stages. These long frame lengths are well suited to the WSN applications such as video transmission, which have relatively high throughputs, and high numbers of bits per video frame. A 9-bit fixed point 2's complement format is used for representing the internal values, in accordance with the findings of [128]. Furthermore, demodulator symbol probabilities are represented by unsigned 4-bit fixed point values, which were found to be the minimum number of bits imposing a negligible performance degradation. Finally, the extrinsic LLRs of $\tilde{\mathbf{z}}^{\mathrm{e}}$, $\tilde{\mathbf{u}}_1^{\mathrm{e}}$ and $\tilde{\mathbf{u}}_2^{\mathrm{e}}$ are scaled by 0.75 in order to mitigate the performance loss imposed by using the Max-Log-BCJR over the Log-BCJR, as described in Section 5.2.1. The following sections further detail how the individual parts of the scheme seen in Figure 5.1 are mapped to the proposed architecture.

5.3.3.1 URC Decoder

As described in Section 2.1.6, in order to reduce the memory requirement of the Max-Log-BCJR algorithm, the URC trellis [44, Figure 1] is decomposed into groups of trellis stages referred to as windows, which are processed separately and in order, as described in Section 5.2.2. The decoding process invoked for each window of the URC decoder proceeds using PIVI, as follows. Firstly, the decoder carries out a forwards recursion for calculating the α values for the window, storing them in the intermediate memories of Figure 5.2. This forward recursion is initialized using the α values calculated at the end of the previous window. As shown in Figure 5.5, each CU is used for calculating the α value for a different one of the $r = 8$ URC states in each trellis stage, using three clock cycles. This is followed by the window's backwards recursion, which is initialized using PIVI, where the β values are gleaned from the beginning of the next window required from the previous iteration of the decoder. Then, on a trellis-stage by trellis-stage basis, the β values are calculated, followed by the δ values and the output extrinsic LLR using a total of 10 clock cycles, as shown in Figure 5.5. Initially for the first bit of each window the β value calculations require three clock cycles. However, for the subsequent bits a partial result gleaned from the δ calculation is saved in registers, which is used by the β calculation, hence resulting in this calculation requiring only a single clock cycle, as shown in Fig 5.5. Note that the extrinsic LLR calculation requires two 8-input max operations and a subtraction to be undertaken. This is achieved using eight 2-input max operations in a first clock cycle, four 2-input max operations in a second clock cycle, two 2-input max operations in a third clock cycle, and a subtraction in a fourth clock cycle, as shown in Figure 5.5 However, this process does not require all $r = 8$ CUs in all four clock cycles, hence resulting in a slight under-utilization of the CUs for this part of the decoding process.

Owing to the flexible nature of the decoder, these idle cycles of the CU can be used for performing the iterative demodulator operations. Figure 5.1 shows that each of the demodulator's extrinsic LLRs $\tilde{\mathbf{w}}_1^{\mathrm{e}}$ and $\tilde{\mathbf{w}}_2^{\mathrm{e}}$ depends only on one *a priori* LLR from $\tilde{\mathbf{w}}_2^{\mathrm{a}}$ or $\tilde{\mathbf{w}}_1^{\mathrm{a}}$ respectively. As a benefit, the demodulator can be operated immediately after the *a priori* LLR has been obtained by interleaving the corresponding extrinsic LLR of $\tilde{\mathbf{v}}_1^{\mathrm{e}}$ or $\tilde{\mathbf{v}}_2^{\mathrm{e}}$. Figure 5.5 shows how each of these extrinsic LLRs is stored in the registers of a CU, until the next period in which there are idle CUs. The demodulator can be operated using four additions in a first clock cycle, plus two max operations in a second clock



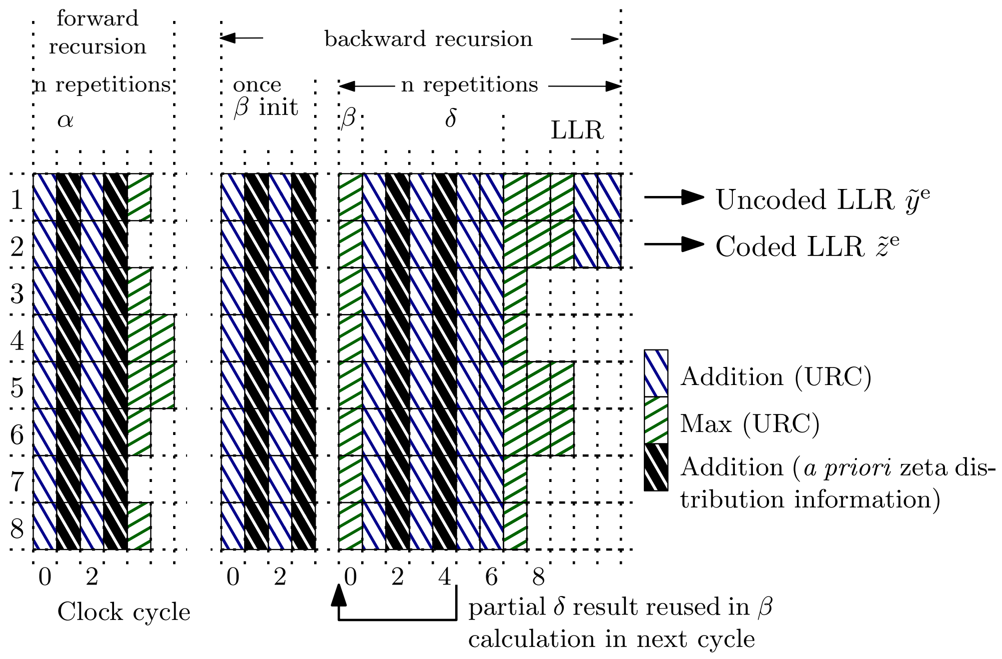


Figure 5.5: Timing diagram for the CUs when performing URC and UEC decoding. Here, the operations for one trellis stage are shown for both the forwards and backwards recursion. Before performing the first backwards recursion for each window, the operations 'β init' are activated. Since the δ calculation and the β calculation in the next trellis stage share an identical operation, the result of this operation is saved in the CU's registers during the δ calculation for use in the subsequent β calculation, as indicated by the arrows on the two timing diagrams.

cycle and a final addition in a third clock cycle, as shown in Figure 5.5. The extrinsic LLRs generated by the URC decoder and demodulator are then both saved into the LLR memories.

Finally, after the decoding of a window has been completed, the last set of β values are saved in the intermediate memories, in order to initialize the decoding of the adjacent window during the next iteration of that decoder, in accordance with PIVI.

5.3.3.2 UEC trellis decoder

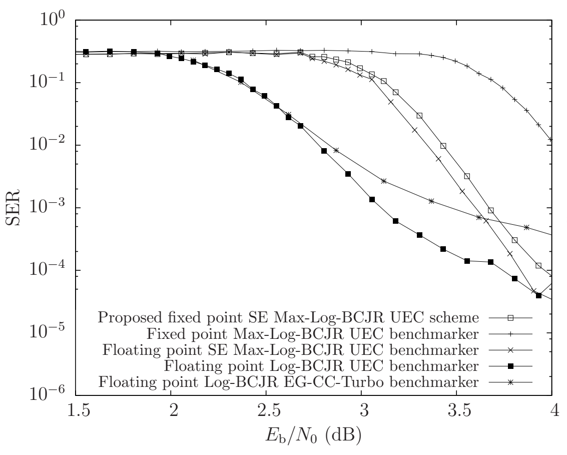
The UEC trellis decoder operates in a similar fashion to the URC decoder, employing windowing and PIVI. Figure 5.5 shows that more clock cycles are needed for the UEC trellis decoder, owing to its different trellis structure. More specifically, while each state of the URC trellis has only two merging transitions, an $r = 8$-state UEC trellis has two central states with four merging transitions, which results in a max operation that requires two clock cycles, using two CUs, as shown in Figure 5.5.

As described in Section 5.3.2, the other key difference associated with the UEC trellis decoder is that the *a priori* knowledge of the source symbols' probability distribution can be incorporated into the α and β calculation, in order to enhance the attainable performance. These values are constants, which can be loaded into the CUs by the input multiplexer, as shown in Figures 5.2 and 5.3. This extra addition can be seen in Figure 5.5, and results in two extra clock cycles being required per bit.

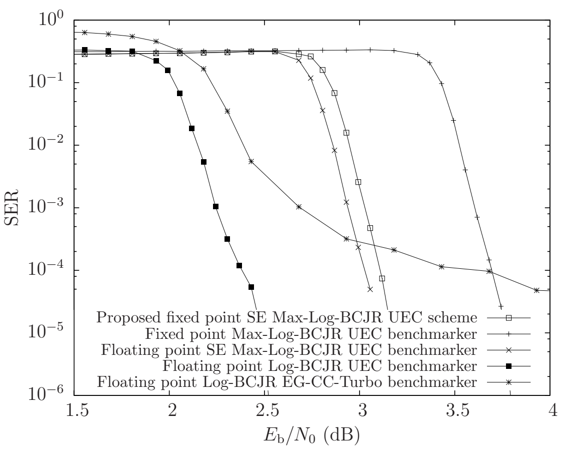
The $r = 8$-state UEC trellis [27, Figure 3] was chosen to match the number of URC trellis states, which was found to provide the best exploitation of the proposed architecture's hardware resources. Note that the number of states that are used in a UEC trellis can be selected independently of the number used in the encoder and can be adjusted at run-time to strike a trade-off between error correction capability and complexity [6]. Motivated by this, the proposed architecture may also be configured to operate in a mode, where only $r = 4$ states are used in the trellis decoder. This doubles the throughput of the UEC decoder, since two bits are scheduled to be decoded at the same time.

5.4 Results

This section will characterize the performance of the proposed architecture and the implementation of the iterative decoder shown in Figure 5.1. Since this is the first implementation of a UEC-based iterative decoder, there is no previous work that provides a direct comparison. However, in order to confirm that the design proposed by this chapter operates within reasonable boundaries, we will compare it – wherever possible



(a) $N = 624$



(b) $N = 6144$

Figure 5.6: SER plot for the proposed scheme after 8 iterations, for frames comprising $N \in \{624, 6144\}$ bits. The implementation is compared to benchmarkers which implement the exact Log-BCJR to show the implementation loss, as well as a SSCC benchmarker to show the improvement offered by the UEC.

– to the architecture of [51]. Figure 5.6 shows the SER performance of the implemented iterative decoder after $I = 8$ decoding iterations, when receiving $N = 624$-bit or $N = 6144$-bit frames of zeta distributed symbols associated with $p_1 = 0.797$ transmitted over a Additive White Gaussian Noise (AWGN) channel. The SER curves for the proposed decoder were generated using a MATLAB-based software simulation, that matches the operation of the implemented decoder. The implementation in this chapter is compared against three benchmarkers, in order to explicitly quantify the impact of the design decisions. The first of these represents the lower-bound performance of the scheme, which is obtained under the idealized conditions of using floating point numbers and the exact Log-BCJR. The second benchmarker employs the floating point Max-SE-Log-BCJR, in order to show the impact of fixed point numbers, as well as to demonstrate how the scaled extrinsic LLRs close the performance gap between the Max-Log-BCJR and the exact Log-BCJR. In order to quantify the advantage of using JSCC, the third benchmarker is a SSCC scheme. This benchmarker is referred to as the EG-CC-Turbo scheme, since it replaces the UEC code of our proposed scheme with the Elias Gamma (EG) source code and a separate Convolutional Code (CC), while retaining the turbo code comprising the two URCs. More specifically, the EG source code encodes input symbols, yielding a bit-vector which does not have equiprobable bit values, which would normally lead to capacity loss [27]. In order to mitigate this capacity loss, the 8-state CC encoder is invoked for encoding the EG encoder's output into a bit-vector which has equiprobable bit values. A parallel concatentation of URC encoders is then used as the channel code, before QPSK modulation. In the EG-CC-Turbo receiver, iterations occur between the two URC decoders and the CC decoder. The benchmarker has a similar complexity to the proposed scheme, since the three decoders of the proposed scheme and the three decoders of the benchmarker all employ 8 states. Note that the

Table 5.1: The key characteristics of the proposed architecture compared to other WSN channel decoders.

	This work	Li 2013 [51]	Biroli 2012 [120]
Decoder type	Max-Log-BCJR	Log-BCJR	Serial LDPC
Frame length N	6144	6144	576
Technology scale (nm)	90	90	90
Voltage (V)	1.0	1.0	1.0
Frequency (MHz)	333	333	20
Iterations I	8	5	10
Memory requirement (kbit)	289	188	
Area (mm^2)	0.46	0.35	0.13
Throughput (Mbit/s)	0.93	1.03	0.25
Area efficiency (Area/Throughput)	0.49	0.34	0.52
Energy efficiency (nJ/bit/iteration)	0.80	0.81	0.27

EG-CC-Turbo benchmarker employs floating point numbers and the exact Log-BCJR. The parameter value $p_1 = 0.797$ is employed for the zeta distribution that is used to generate the random source symbols, since this results in the same effective throughput of $\eta = 0.762$ bits per symbol for both the UEC schemes and the EG-CC-Turbo benchmarker of [27]. Compared to the corresponding JSCC UEC benchmarker, the SSCC EG-CC-Turbo benchmarker requires an SNR of about 0.7 dB higher, in order to reach the SER turbo cliff at frame lengths of both 624 and 6144 bits. This benchmarker also suffers from a pronounced error floor, which potentially causes a high SER at higher SNRs.

Table 5.1 shows the key performance characteristics of the proposed architecture, compared to similar decoders for WSN applications, namely the turbo decoder of [51] and the LDPC decoder of [120]. The proposed implementation has a total memory requirement of 289 kbits, which is significantly higher than the 188 kbit of [51]. However, the memory requirement of the demodulator's symbol probabilities constitutes the majority of the memory in the proposed implementation. Motivated by the fact that [51] does not consider the implementation of the demodulator, ignoring the demodulator's memory requirement in this implementation leads to a total memory requirement of 190 kbits, which is comparable to [51]. This chapter has demonstrated support for frame lengths of 6144 bits, however, further memory deduction can be achieved if the ASIC is not required to support long frame lengths. Note that further throughput improvements could be achieved by employing early stopping [25] to halt the iterative decoding process, as soon as a Cyclic Redundancy Check (CRC) is satisfied. In this case, the results provided in Table 5.1 would represent the worst-case performance of the proposed scheme. More specifically, the addition of a CRC would allow the throughput to be increased and the energy efficiency to be improved, in the case where fewer iterations are required, for example, when the channel SNR is high.

The timing diagrams of Figure 5.5 can be used for obtaining the number of clock cycles required for processing each interleaved bit. It can be seen that 13 clock cycles per bit are required for the URC decoder, while 18 clock cycles per bit are necessitated for the UEC trellis decoder [27, Figure 3]. Furthermore, the exploitation of the CUs is 88% for the URC decoder and demodulator, while it is 81% for the UEC trellis decoder. These exploitation ratios constitute significant improvements over the 68% recorded in [51].

The proposed ASIC design was synthesized by Synopsis Design Compiler using the ST 90nm 1.00V design kit. The design can achieve a clock frequency of 333 MHz after design compilation and uses approximately 0.46 mm^2 of die area, most of which is for the memories. This is comparable to the solution of [51], which has an area of 0.35 mm^2 at the same clock frequency and technology scale. With the aid of $I = 8$ decoding iterations, our design achieves a decoded throughput of 0.93 Mbps, while the previous solution of [51] achieved 1.03 Mbps at the same clock frequency, albeit without the overhead of UEC decoding and demodulation. Meanwhile the LDPC decoder of [120] achieved a decoded throughput of 0.25 Mbps, but has a smaller chip area of 0.13 mm^2, at the same technology scale. The worst-case post-synthesis energy consumption of our design is 0.80 nJ/bit/iteration, which is higher than that of the comparable architectures due to the inclusion of the UEC trellis decoder. While the architecture of [51] achieved a comparable energy efficiency of 0.81 nJ/bit/iteration, the LDPC design of [120] achieved an energy efficiency of 0.27 nJ/bit/iteration. As shown in Table 5.1, the three decoders achieve a similar area efficiency, despite the proposed design having the overhead of including UEC decoding and demodulation.

5.5 Conclusions

This chapter has proposed a flexible and programmable architecture suitable for WSNs that can be used to implement iterative receivers employing a wide variety of different components. The application of the proposed architecture has been demonstrated for iteration between a UEC decoder, a turbo decoder and an iterative demodulator. This chapter has shown that the schemes discussed in Chapters 3 and 4 can lend themselves to efficient hardware implementation The flexibility of the architecture allows for the iterative demodulator to use otherwise idle hardware, ultimately achieving a hardware exploitation of up to 88%, while restricting the area of the design to 0.46 mm^2 with a

0.93 Mbps throughput. This compares favourably to a significantly less-capable benchmarker, which achieved a hardware exploitation ratio of 68%, while having an area of 0.35mm^2 and a throughput of 1.03 Mbps.

Chapter 6

A high-throughput FPGA architecture for UEC and LTE

6.1 Introduction

With the increasing application of high and ultra-high definition video, the demand for high throughput wireless communication systems is also increasing. Furthermore, low latency wireless communication is also required by many of these video applications such as the first-person remote control of unmanned vehicles, or mobile access to cloud-computing based video games. A high transmission throughput, and hence a low transmission latency, can be achieved by using near-capacity transmission techniques to maximize the bandwidth efficiency. As was discussed in Section 2.5, near-capacity operation may be achieved using Shannon's Separate Source and Channel Coding (SSCC) concept [16]. Here, a near-entropy source code such as the arithmetic code [7] is used to remove redundancy from the source, in order to achieve a high degree of compression. Meanwhile, a separate near-capacity channel code, such as a turbo code [1], is used for introducing specifically selected redundancy to achieve a high degree of error correction. However, Shannon's SSCC concept assumes that infinite computational complexity and latency can be afforded. Indeed, arithmetic codes are only capable of removing all redundancy for the sake of achieving near-entropy compression when they operate on long sequences of source symbols, imposing a latency bottleneck. Meanwhile, the conventional approach to turbo decoding employs the serial Logarithmic Bahl-Cocke-Jelinek-Raviv (Log-BCJR) algorithm, discussed in Section 2.1.5, which imposes a throughput limitation due to its limited grade of parallelism and iterative nature, even when using hardware acceleration. These factors limit the overall throughput and latency, precluding the high-throughput and low-latency applications described above.

This chapter is partially based on the following publication.
M. F. Brejza, R. G. Maunder, B. Al-Hashimi and L. Hanzo. A High-Throughput FPGA Architecture for Fully-Parallel Joint Source and Channel Decoding. *IEEE Access, PP*(99), 1–1.

As we have seen with the previous chapters, this motivates Joint Source and Channel Coding (JSCC) [3], which can offer performance gains compared to conventional SSCC. In contrast to SSCC, a JSCC scheme does not attempt to remove all of the redundancy from the encoded source symbols using sophisticated compression techniques. Instead, a JSCC scheme uses the residual redundancy that remains after compression for the purpose of error correction. As a result, a JSCC scheme can encode frames of any length, while maintaining near-capacity operation. Upon this background, this chapter will employ the Unary Error Correction (UEC) code, which has been used throughput this thesis to offer a near-capacity operation with a low complexity, even for large source alphabets such as those of video sources.

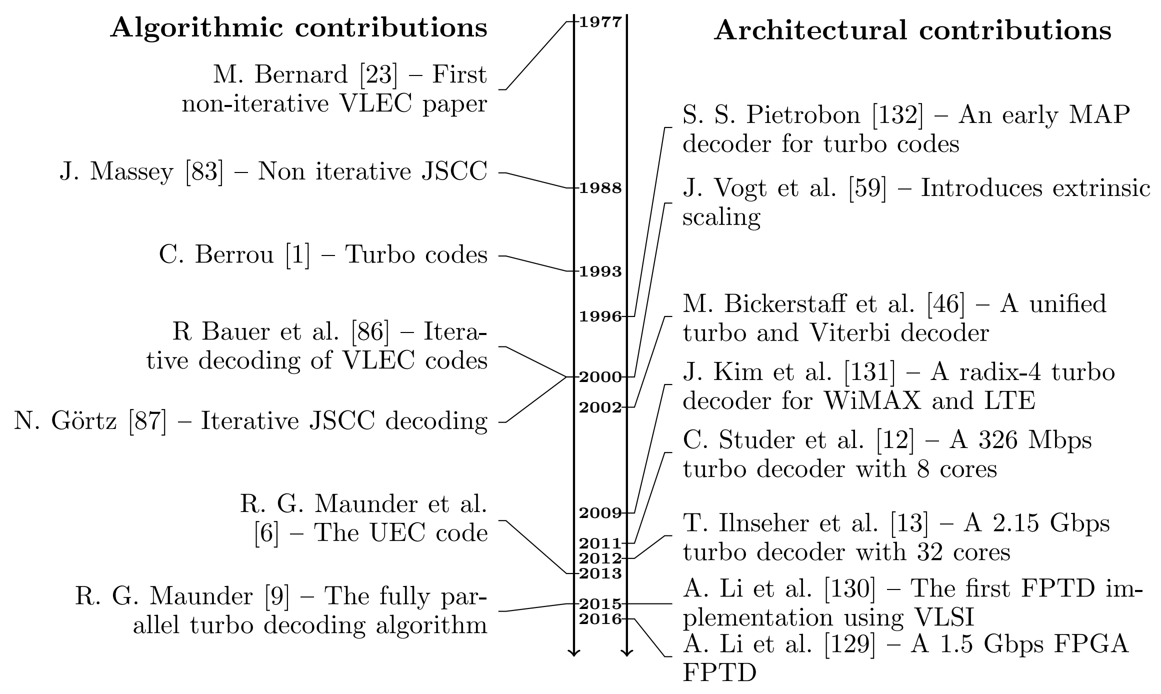

Algorithmic contributions
M. Bernard [23] – First non-iterative VLEC paper
J. Massey [83] – Non iterative JSCC
C. Berrou [1] – Turbo codes
R Bauer et al. [86] – Iterative decoding of VLEC codes
N. Görtz [87] – Iterative JSCC decoding
R. G. Maunder et al. [6] – The UEC code
R. G. Maunder [9] – The fully parallel turbo decoding algorithm
1977
1988
1993
1996
2000
2002
2009
2011
2012
2013
2015
2016
Architectural contributions
S. S. Pietrobon [132] – An early MAP decoder for turbo codes
J. Vogt et al. [59] – Introduces extrinsic scaling
M. Bickerstaff et al. [46] – A unified turbo and Viterbi decoder
J. Kim et al. [131] – A radix-4 turbo decoder for WiMAX and LTE
C. Studer et al. [12] – A 326 Mbps turbo decoder with 8 cores
T. Ilnseher et al. [13] – A 2.15 Gbps turbo decoder with 32 cores
A. Li et al. [130] – The first FPTD implementation using VLSI
A. Li et al. [129] – A 1.5 Gbps FPGA FPTD

Figure 6.1: The key algorithmic and architectural contributions in source and channel coding.

As shown in Figure 1.4, this chapter considers the hardware implementation of a decoder for a JSCC scheme, namely the UEC-URC code. This UEC-URC decoder is one of two key parts of the Rice Error Correction (RiceEC) and Exponential Golomb Error Correction (ExpGEC) decoders of Chapter 4 In contrast to Chapter 5 which was designed for low throughputs where the processing latency was not a concern, this chapter requires an architecture having a high processing throughput and a low processing latency, in order to avoid imposing a bottleneck upon the achievable transmission throughput and latency. This is a particular challenge, since UEC-URC decoding has been previously based on the Log-BCJR algorithm, which suffers from serial data dependencies that are not conducive to high throughput and low latency processing. In order to address the bottleneck imposed by the serial data dependencies of conventional Log-BCJR turbo decoders, the previous work of [130] has considered the hardware implementation of turbo decoders having high throughput and low latency. More specifically, the Fully Parallel Turbo Decoder (FPTD) [9] achieves a high processing throughput and a low processing latency by eliminating the serial data dependencies of the conventional Log-BCJR approach, allowing the decoding of all bits to be completed at the same time, in a fully parallel manner, albeit at the cost of requiring a large Application-Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) area. These key contributions on JSCC algorithms and turbo decoder architectures are shown on Figure 6.1.

The contribution of this chapter is that the principle of the FPTD is extended to the implementation of a UEC-URC scheme, which achieves for the first time near-capacity JSCC at a high processing throughout and a low processing latency, hence meeting the requirements described above. Furthermore, as shown in Figure 1.4, this chapter considers the architecture and algorithm jointly, allowing this chapter to propose several novel techniques for improving the chip-area efficiency of the FPTD approach. This is achieved by improving the fully parallel algorithm to allow a pair of bits to be decoded using the same hardware processing element. This also facilitates improved pipelining for the sake of increasing the clock frequency, as well as for reducing the number of decoding iterations required. Since the UEC-URC scheme comprises both UEC and URC decoding components, the novel hardware processing elements of this chapter are designed to be capable of processing both of these different decoders, hence facilitating an efficient hardware design.

The remainder of this chapter is structured as shown in Figure 6.2. In Section 6.2, the LTE turbo code and the UEC-URC scheme are described. Both the LTE turbo decoder and UEC-URC decoder will be considered throughout the chapter. By considering the turbo code alongside the UEC-URC scheme, the architecture proposed on this chapter can be compared with previous implementations of the turbo code. Section 6.2 concludes by describing the fully parallel decoding algorithm introduced in [9]. Following this, Section 6.3 proposes a new paired activation order for the blocks of the fully parallel decoding algorithm, using Bit Error Ratio (BER) and Symbol Error Rate (SER) simulations to quantify the number of iterations required to match the error correction capability of the existing Log-BCJR and FPTD algorithms. Following this, Section 6.4 uses this paired activation order as the basis of an FPGA implementation of the UEC-URC scheme. Section 6.5 details the implementation results, where a throughput of 450 Mbps is achieved on a mid-range FPGA, demonstrating applicability to video applications. The proposed LTE implementation is compared to previous fully parallel LTE turbo decoder implementations, demonstrating a 2.4-fold hardware efficiency improvement. Finally, Section 6.6 offers the conclusions.

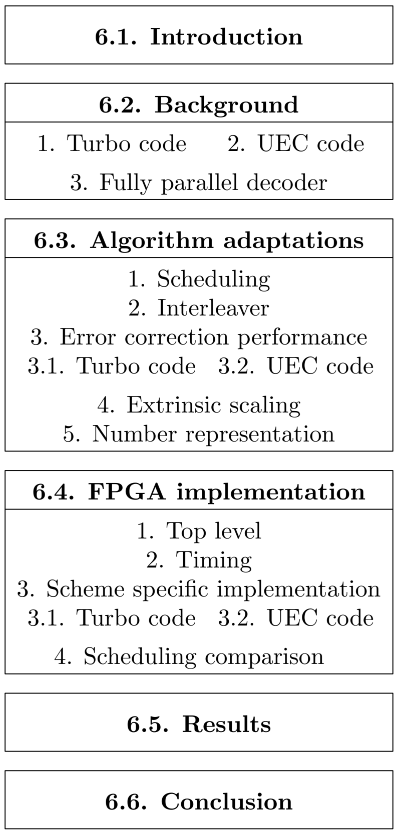


Figure 6.2: The chapter outline.

6.2 Background

This section commences by describing the operation of the LTE turbo code of Figure 6.3 in Section 6.2.1. Following this, Section 6.2.2 highlights the operation of the UEC-URC scheme of Figure 6.5. Finally, Section 6.2.3 describes the fully parallel decoding algorithm, which will be applied to both the LTE turbo code and to the UEC-URC scheme.

6.2.1 LTE turbo code scheme

In this section, the operation of the turbo code used in LTE is described. We commence by describing the encoder in Section 6.2.1.1, followed by the decoder in Section 6.2.1.2.

6.2.1.1 Encoder

As shown in Figure 6.3a, the LTE turbo encoder [18] is comprised of two convolutional encoders [29], namely the upper and lower encoders. This encoder works in the same way as the encoder of Figure 2.2, however here the systematic LLR vectors have been

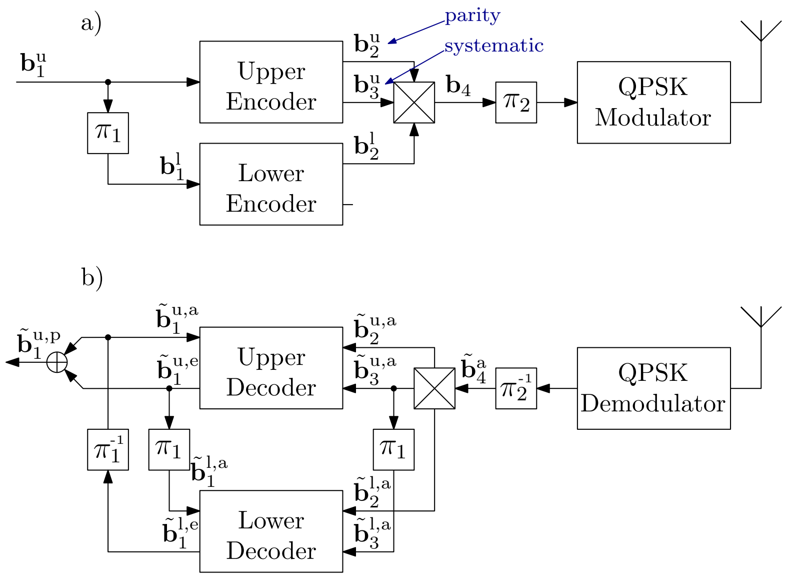

a)
$\mathbf{b}_1^{\mathrm{u}}$
Upper Encoder
Lower Encoder
π_1
$\mathbf{b}_1^{\mathrm{l}}$
$\mathbf{b}_2^{\mathrm{u}}$
$\mathbf{b}_3^{\mathrm{u}}$
$\mathbf{b}_2^{\mathrm{l}}$
parity
systematic
$\mathbf{b}_4$
π_2
QPSK Modulator
b)
$\tilde{\mathbf{b}}_1^{\mathrm{u,p}}$
$\tilde{\mathbf{b}}_1^{\mathrm{u,a}}$
$\tilde{\mathbf{b}}_1^{\mathrm{u,e}}$
Upper Decoder
Lower Decoder
π_1^{-1}
π_1
$\tilde{\mathbf{b}}_1^{\mathrm{l,a}}$
$\tilde{\mathbf{b}}_1^{\mathrm{l,e}}$
$\tilde{\mathbf{b}}_2^{\mathrm{u,a}}$
$\tilde{\mathbf{b}}_3^{\mathrm{u,a}}$
$\tilde{\mathbf{b}}_2^{\mathrm{l,a}}$
$\tilde{\mathbf{b}}_3^{\mathrm{l,a}}$
$\tilde{\mathbf{b}}_4^{\mathrm{a}}$
π_2^{-1}
QPSK Demodulator

Figure 6.3: The LTE turbo encoder and decoder. The interleaver π_2 is beneficial in the case where QPSK modulation is used for communication over a Rayleigh fading channel.

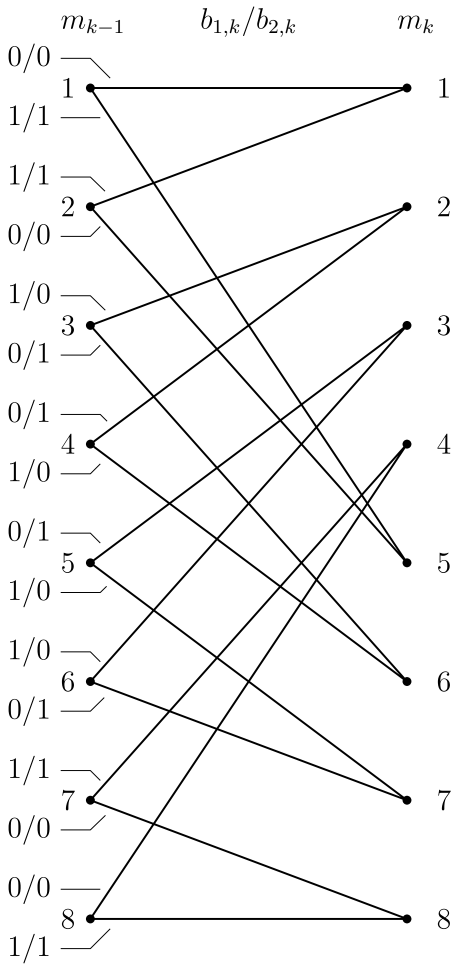


Figure 6.4: Trellis for the LTE turbo code

shown in an alternate manner, in order to more closely mirror the UEC-URC scheme which is considered in this chapter. The input bit-vector $\mathbf{b}_1^{\mathrm{u}} = [b_{1,k}^{\mathrm{u}}]_{k=1}^{N}$ comprises N bits and typically has equiprobable bit values. This bit-vector is encoded by the upper decoder to give the encoded bit-vector $\mathbf{b}_2^{\mathrm{u}} = [b_{2,k}^{\mathrm{u}}]_{k=1}^{N}$, as well as the systematic bit-vector $\mathbf{b}_3^{\mathrm{u}} = [b_{3,k}^{\mathrm{u}}]_{k=1}^{N}$, which is identical to $\mathbf{b}_1^{\mathrm{u}}$. Meanwhile, an interleaver π_1 is used to reorder the input bits of $\mathbf{b}_1^{\mathrm{u}}$ to give the bit-vector $\mathbf{b}_1^{\mathrm{l}} = [b_{1,k}^{\mathrm{l}}]_{k=1}^{N}$, which is encoded by the lower encoder to give the bit-vector $\mathbf{b}_2^{\mathrm{l}} = [b_{2,k}^{\mathrm{l}}]_{k=1}^{N}$. In this way, the LTE turbo code has a coding rate of $R = 1/3$, since each input bit is encoded using three output bits, as shown in figure 6.3. Note that in the following discussion, the superscripts $^{\mathrm{u}}$ and $^{\mathrm{l}}$ will be omitted from the notation when the discussion applies equally to both the upper and lower encoders.

Figure 6.4 shows the trellis of the LTE convolutional encoders. The trellis characterizes the transition between the $r_{\mathrm{LTE}} = 8$ possible states of the encoder, based on its input bit-vector $\mathbf{b}_1$. At the beginning of the encoding process, each encoder starts from state $m_0 = 1$. The bits of $\mathbf{b}_1$ are considered in the order of increasing bit index k. Given any particular previous state m_{k-1}, the value of the input bit $b_{1,k}$ will trigger a state-transition to a state m_k selected from one of two potential options, as shown by the transition labels in Figure 6.4. The transitions between the states m_k corresponding to the successive input bits $b_{1,k}$ form a path $\mathbf{m} = [m_k]_{k=0}^{N}$ through the trellis. Since each input bit $b_{1,k}$ has equiprobable values, the transitions are also equiprobable. For each transition selected, an output bit $b_{2,k}$ is also identified. These parity bits are concatenated together to form the parity bit-vector $\mathbf{b}_2 = [b_{2,k}]_{k=1}^{N}$, mentioned above. For example, given the input bit-vector $\mathbf{b}_1 = [1000111011100100]$ comprising $N = 16$ bits, the convolutional encoding selects the $N + 1 = 17$ states $\mathbf{m} = [1, 5, 3, 6, 7, 4, 6, 3, 6, 3, 2, 1, 1, 1, 5, 3, 6]$, which yields the parity bit-vector $\mathbf{b}_2 = [1111100100100111]$.

Note that the LTE turbo encoder also appends three trellis-termination bits to the end of each of the bit-vectors $\mathbf{b}_1^{\mathrm{u}}$, $\mathbf{b}_1^{\mathrm{l}}$, $\mathbf{b}_2^{\mathrm{u}}$ and $\mathbf{b}_2^{\mathrm{l}}$, in order to guarantee that the end state of each convolutional encoder is $m_{N+3} = 1$ [18], which avoids an error floor [133]. These 12 termination bits are also output by the turbo encoder, but are not shown in Figure 6.3 for the sake of simplicity. Following turbo encoding, the resultant output bits are multiplexed and then modulated as well as transmitted over the channel. Since in this work we are assuming QPSK modulation and an uncorrelated narrowband Rayleigh fading channel [35], no channel-interleaving is required, but the interleaver π_2 is beneficial before modulation to nonetheless ensure that neighbouring bits are not transmitted as pairs within the QPSK symbols.

6.2.1.2 Decoder

The LTE turbo decoder is shown in Figure 6.3b, where the upper and lower decoders correspond to the upper and lower encoders of the LTE turbo encoder. Likewise, the demodulator of Figure 6.3b mirrors the modulator of Figure 6.3a. While the encoder works on the basis of hard bits, having the values either 0 or 1, the demodulator and decoder uses soft bits called Logarithmic Likelihood Ratios (LLRs) [134], which express the decoder's uncertainty in the bit value owing to the noise in the channel. The use of LLRs allows the two decoders in the receiver to iteratively exchange information about their confidence in the various bit values, yielding improved decoding performance. Following their reception, the LLRs gleaned from the demodulator are de-interleaved by π_2^{-1}, then de-multiplexed, yielding the *a priori* LLR-vectors $\tilde{\mathbf{b}}_2^{\mathrm{u,a}} = [\tilde{b}_{2,k}^{\mathrm{u,a}}]_{k=1}^{N}$, $\tilde{\mathbf{b}}_2^{\mathrm{l,a}} = [\tilde{b}_{2,k}^{\mathrm{l,a}}]_{k=1}^{N}$ and $\tilde{\mathbf{b}}_3^{\mathrm{u,a}} = [\tilde{b}_{3,k}^{\mathrm{u,a}}]_{k=1}^{N}$. The systematic LLR-vector $\tilde{\mathbf{b}}_3^{\mathrm{u,a}}$ is also interleaved through π_1 to yield $\tilde{\mathbf{b}}_3^{\mathrm{l,a}} = [\tilde{b}_{3,k}^{\mathrm{l,a}}]_{k=1}^{N}$. Furthermore, the lower decoder provides the upper decoder with the *a priori* LLR-vector $\tilde{\mathbf{b}}_1^{\mathrm{u,a}} = [\tilde{b}_{1,k}^{\mathrm{u,a}}]_{k=1}^{N}$, which is populated with zero-valued LLRs at the start of the decoding process. Likewise, the upper decoder provides the lower decoder with $\tilde{\mathbf{b}}_1^{\mathrm{u,a}} = [\tilde{b}_{1,k}^{\mathrm{u,a}}]_{k=1}^{N}$, as shown in Figure 6.3a.

In a conventional turbo decoder, the upper and lower decoders employ the Log-BCJR algorithm, described in Section 2.1.5, for converting the input *a priori* LLR-vectors into the extrinsic output LLR-vectors $\tilde{\mathbf{b}}_1^{\mathrm{u,e}} = [\tilde{b}_{1,k}^{\mathrm{u,e}}]_{k=1}^{N}$ and $\tilde{\mathbf{b}}_1^{\mathrm{l,e}} = [\tilde{b}_{1,k}^{\mathrm{l,e}}]_{k=1}^{N}$. Note that the Log-BCJR algorithm can also beneficially exploit the 12 LLRs provided by the demodulator to correspond to the 12 termination bits. In these conventional turbo decoders, the upper and lower decoders are operated alternately, in an iterative manner. More specifically, the upper decoder outputs $\tilde{\mathbf{b}}_1^{\mathrm{u,e}}$, which is interleaved through π_1 to become the *a priori* LLR-vector $\tilde{\mathbf{b}}_1^{\mathrm{l,a}}$, which is input to the lower decoder. Likewise, the lower decoder outputs the extrinsic LLR-vector $\tilde{\mathbf{b}}_1^{\mathrm{l,e}}$, which is de-interleaved through π_1^{-1} and input as the *a priori* LLR-vector $\tilde{\mathbf{b}}_1^{\mathrm{u,a}}$ into the upper decoder. At the same time, the turbo decoder outputs the vector of *a posteriori* LLRs $\tilde{\mathbf{b}}_1^{\mathrm{u,p}} = [\tilde{\mathbf{b}}_{1,k}^{\mathrm{u,p}}]_{k=1}^{N}$, which is obtained by the addition of $\tilde{\mathbf{b}}_1^{\mathrm{u,a}}$ and $\tilde{\mathbf{b}}_1^{\mathrm{u,e}}$, and so represents the combined knowledge of the bit-vector $\mathbf{b}_1^{\mathrm{u}}$.

If the channel Signal-to-Noise Ratio (SNR) is sufficiently high, the quality of LLRs may be expected to increase in each successive iteration, as the decoder recovers the original encoded message. The iterations exchanging extrinsic information between the decoders continue until a fixed number of iterations is completed, or until a hard decision based on the *a posteriori* LLR-vectors satisfies the classic Cyclic Redundancy Check (CRC).

6.2.2 UEC-URC JSCC scheme

In this section, the operation of the UEC-URC JSCC scheme is detailed. First, Section 6.2.2.1 describes how the scheme encodes and transmits a stream of symbols. Following this, Section 6.2.2.2 describes the operation of the UEC-URC decoder, which attempts to recover the original symbols. In this section, notation is adopted that is consistent with the turbo scheme of Section 6.2.1.

6.2.2.1 Encoder

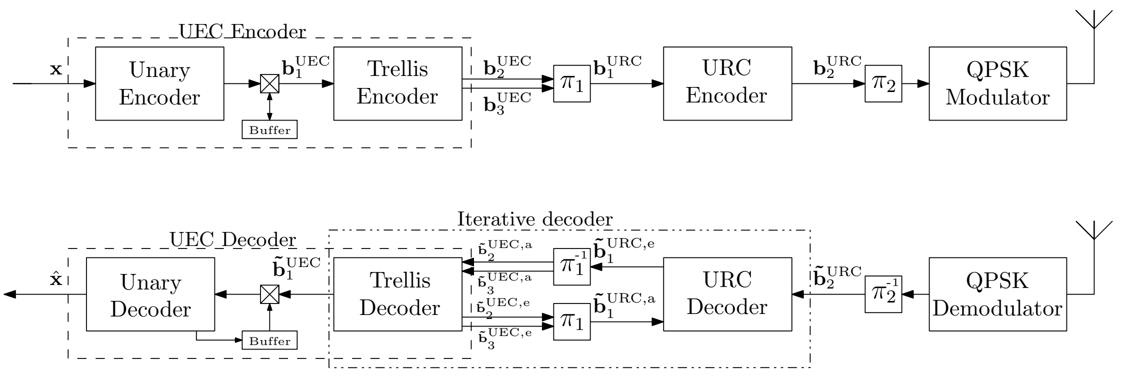

UEC Encoder
$\mathbf{x}$
Unary Encoder
$\mathbf{b}_1^{\mathrm{UEC}}$
Buffer
Trellis Encoder
$\mathbf{b}_2^{\mathrm{UEC}}$
$\mathbf{b}_3^{\mathrm{UEC}}$
π_1
$\mathbf{b}_1^{\mathrm{URC}}$
URC Encoder
$\mathbf{b}_2^{\mathrm{URC}}$
π_2
QPSK Modulator
Iterative decoder
UEC Decoder
$\hat{\mathbf{x}}$
Unary Decoder
$\tilde{\mathbf{b}}_1^{\mathrm{UEC}}$
Buffer
Trellis Decoder
$\tilde{\mathbf{b}}_2^{\mathrm{UEC,a}}$
$\tilde{\mathbf{b}}_3^{\mathrm{UEC,a}}$
$\tilde{\mathbf{b}}_2^{\mathrm{UEC,e}}$
$\tilde{\mathbf{b}}_3^{\mathrm{UEC,e}}$
π_1^{-1}
π_1
$\tilde{\mathbf{b}}_1^{\mathrm{URC,e}}$
$\tilde{\mathbf{b}}_1^{\mathrm{URC,a}}$
URC Decoder
$\tilde{\mathbf{b}}_2^{\mathrm{URC}}$
π_2^{-1}
QPSK Demodulator

Figure 6.5: The UEC-URC scheme.

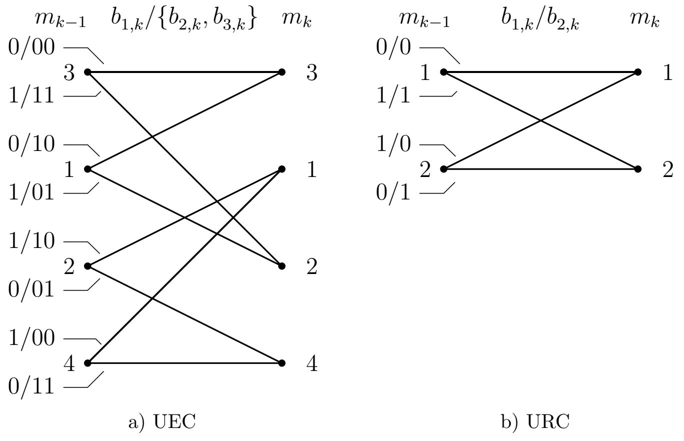


Figure 6.6: Trellis for UEC and URC codes

The UEC-URC encoder is shown in Figure 6.5. This is a JSCC scheme, which operates on the basis of a stream of symbols $\mathbf{x} = [x_i]$, rather than bits. The UEC encoder operates in the same way as described in Section 2.5 and in previous chapters. To briefly summarize, the UEC encodes a stream of source symbols, which are assumed to be a realization of a stream of Independent and Identically Distributed (IID) Random Variables (RVs) $\mathbf{X} = [X_i]$, where each RV is selected from the set $\mathbb{N}_1 = \{1, 2, 3, 4, ..., \infty\}$, according to the zeta probability distribution which is given by Equation (2.16). In the zeta distribution, symbols having the value 1 are the most probable, where this probability is given by $p_1 = Pr(X_i = 1)$, and the probability of a symbol value decreases as the symbol value increases.

The unary encoder of Figure 6.5 converts each symbol x_i to a codeword denoted by Unary(x_i), as shown in Table 2.6. The length of each unary codeword is equal to the value of the corresponding symbol x_i, where the first $(x_i - 1)$ bits in the codeword have the value 0, while the last bit has the value of 1.

In contrast to turbo coding, the bit values output by a unary encoder are not equiprobable. A further difference with respect to turbo coding is that for a fixed number of input symbols, unary source coding yields a variable number of bits. This necessitates a mechanism for partitioning the bits output from the unary encoder into fixed length bit-vectors. More specifically, the stream of codewords produced by the unary encoder are concatenated together, then using the technique developed in Section 4.4,the concatenated codewords are partitioned into a succession of bit-vectors $\mathbf{b}_1^{\mathrm{UEC}} = [b_{1,k}^{\mathrm{UEC}}]_{k=1}^{N}$, having a fixed length of N bits. Owing to the partitioning, some unary codewords may be split between successive bit-vectors, hence a buffer is used for storing these bits so they can be removed from the end of one bit-vector and concatenated onto the start of the next, as shown in Figure 6.5. For example, the unary encoding of the symbol vector $\mathbf{x} = [1, 2, 1, 1, 4, 1, 3, 1, 2]$ associated with $N = 8$ produces the successive bit-vectors $\mathbf{b}_1^{\mathrm{UEC}} = [1, 0, 1, 1, 1, 0, 0, 0]$ and $\mathbf{b}_1^{\mathrm{UEC}} = [1, 1, 0, 0, 1, 1, 0, 1]$. Note that the fifth element of $\mathbf{x}$ is split between the two bit-vectors of $\mathbf{b}_1^{\mathrm{UEC}}$.

As shown in Figure 6.5, the bit-vector $\mathbf{b}_1^{\mathrm{UEC}}$ is encoded by the trellis encoder, yielding the bit-vectors $\mathbf{b}_2^{\mathrm{UEC}} = [b_{2,k}^{\mathrm{UEC}}]_{k=1}^{N}$ and $\mathbf{b}_3^{\mathrm{UEC}} = [b_{3,k}^{\mathrm{UEC}}]_{k=1}^{N}$. This encoding is performed according to the $r_{\mathrm{UEC}} = 4$-state trellis of Figure 6.6a, in a manner similar to the convolutional encoding described in Section 6.2.1.1. Figure 6.6a shows how the trellis transitions from state m_{k-1} to m_k, depending on the input bit $b_{1,k}^{\mathrm{UEC}}$. The transitions are also labelled with the corresponding codeword $\{b_{2,k}^{\mathrm{UEC}}, b_{3,k}^{\mathrm{UEC}}\}$ which is output when each transition is selected. The trellis encoder starts at state $m_0 = 1$ at the beginning of each bit-vector $\mathbf{b}_1^{\mathrm{UEC}}$. In contrast to the turbo code, since the bit values of $\mathbf{b}_1^{\mathrm{UEC}}$ are not equiprobable, the transitions $P(m|m')$ are not equiprobable either. As a result, knowledge of the conditional transition probabilities $P(m|m')$ can be used to aid the receiver, as it will be described in Section 6.2.2.2. Here an $r_{\mathrm{UEC}} = 4$-state UEC trellis has been chosen, since this provides reasonable performance, while allowing an efficient hardware implementation, as will be demonstrated in Section 6.4. Since the UEC trellis is symmetric and employs complementary codewords in the top and bottom halves, the UEC-encoded output bits $b_{2,k}^{\mathrm{UEC}}$ and $b_{3,k}^{\mathrm{UEC}}$ generated by the trellis encoder are guaranteed to have equiprobable values.

The bits $b_{2,k}^{\mathrm{UEC}}$ and $b_{3,k}^{\mathrm{UEC}}$ output by the trellis encoder are concatenated for forming the vectors $\mathbf{b}_2^{\mathrm{UEC}} = [b_{2,k}^{\mathrm{UEC}}]_{k=1}^{N}$ and $\mathbf{b}_3^{\mathrm{UEC}} = [b_{3,k}^{\mathrm{UEC}}]_{k=1}^{N}$. For example, given the vector comprising $N = 15$ bits $\mathbf{b}_1^{\mathrm{UEC}} = 011001010010111$, the path through the trellis can be expressed as a vector $\mathbf{m} = [1, 3, 2, 1, 3, 3, 2, 4, 1, 3, 3, 2, 4, 1, 2, 1]$ of $N+1 = 16$-states. This path through the trellis encoder produces the output vectors $\mathbf{b}_2^{\mathrm{UEC}} = [111101001010001]$ and $\mathbf{b}_3^{\mathrm{UEC}} = [010001100011010]$, each comprising $N = 15$ bits.

Following trellis encoding, the bit-vectors $\mathbf{b}_2^{\mathrm{UEC}}$ and $\mathbf{b}_3^{\mathrm{UEC}}$ are concatenated and interleaved through an interleaver π_1, similar to the one used in the turbo code, producing the bit-vector $\mathbf{b}_1^{\mathrm{URC}} = [b_{1,k}^{\mathrm{URC}}]_{k=1}^{2N}$. An $r_{\mathrm{URC}} = 2$-state Unity Rate Convolutional (URC) encoder is employed to accumulate the bits of $\mathbf{b}_1^{\mathrm{URC}}$, in order to generate the bit-vector

$\mathbf{b}_2^{\mathrm{URC}} = [b_{2,k}^{\mathrm{URC}}]_{k=1}^{2N}$. This accumulation is equivalent to performing encoding using the trellis shown in Figure 6.6b. This URC code will facilitate iterative decoding exchanging extrinsic information with the UEC trellis code in receiver, for the sake of allowing near-capacity operation, as discussed in Section 6.2.2.2. It was shown in [92] that a 2-state URC code has more complementary EXIT characteristics to the UEC trellis encoder, compared to a 4- or 8-state URC encoder. The 2-state URC encoder also has a lower complexity, which will be exploited in Section 6.4. The URC encoder operates in a similar manner to the convolutional encoder described in Section 6.2.1.1. The input bits $b_{1,k}^{\mathrm{URC}}$ are processed in order of increasing index k, where each bit causes the trellis of Figure 6.6b to transition from the previous state m_{k-1} to the next state m_k, while outputting the associated bit $b_{2,k}^{\mathrm{URC}}$. The path comprising $2N+1$ states taken by the encoder may be represented as $\mathbf{m} = [m_k]_{k=0}^{2N}$. Since the bits input to the URC encoder have equiprobable values, the bits output from the URC encoder also have equiprobable values. In contrast to the LTE turbo code, no termination is used by the URC trellis. Following URC encoding, the bit-vector $\mathbf{b}_2^{\mathrm{URC}}$ is interleaved by π_2, before being Quadrature Phase Shift Keying (QPSK) modulated and transmitted over the Rayleigh fading channel.

6.2.2.2 Decoder

As shown in Figure 6.5b, the LLR-vector $\tilde{\mathbf{b}}_2^{\mathrm{a,URC}} = [\tilde{b}_{2,k}^{\mathrm{a,URC}}]_{k=1}^{2N}$ is obtained after QPSK demodulation and de-interleaving by π_2^{-1}. This is entered into the iterative decoder, which is comprised of a URC decoder and a UEC trellis decoder, in correspondence to the URC encoder and UEC trellis encoder in the transmitter. At the start of the decoding process, all other LLR-vectors are populated with zero values. More specifically, the URC decoder is provided with the encoded *a priori* LLR-vector $\tilde{\mathbf{b}}_2^{\mathrm{a,URC}}$ from the demodulator, as well as the uncoded *a priori* LLR-vector $\tilde{\mathbf{b}}_1^{\mathrm{a,URC}} = [\tilde{b}_{1,k}^{\mathrm{a,URC}}]_{k=1}^{2N}$ provided by the trellis decoder. Likewise, the UEC trellis decoder is provided with the *a priori* LLR-vectors $\tilde{\mathbf{b}}_2^{\mathrm{a,UEC}} = [\tilde{b}_{2,k}^{\mathrm{a,UEC}}]_{k=1}^{N}$ and $\tilde{\mathbf{b}}_3^{\mathrm{a,UEC}} = [\tilde{b}_{3,k}^{\mathrm{a,UEC}}]_{k=1}^{N}$ by the URC decoder. In a conventional receiver, the URC decoder may employ the Log-BCJR decoder to transform the *a priori* input LLR-vectors into the extrinsic output LLR-vector $\tilde{\mathbf{b}}_1^{\mathrm{e,URC}} = [\tilde{b}_{1,k}^{\mathrm{e,URC}}]_{k=1}^{2b}$, according to the trellis of Figure 6.6b. Likewise, the UEC trellis decoder may employ the Log-BCJR algorithm to transform the input *a priori* LLR-vectors into the extrinsic LLR-vectors $\tilde{\mathbf{b}}_2^{\mathrm{e,UEC}} = [\tilde{b}_{2,k}^{\mathrm{e,UEC}}]_{k=1}^{N}$ and $\tilde{\mathbf{b}}_3^{\mathrm{e,UEC}} = [\tilde{b}_{3,k}^{\mathrm{e,UEC}}]_{k=1}^{N}$, according to the trellis of Figure 6.6a. Note that the trellis decoder's encoded *a priori* LLR vectors $\tilde{\mathbf{b}}_2^{\mathrm{a,UEC}}$ and $\tilde{\mathbf{b}}_3^{\mathrm{a,UEC}}$, as well as the encoded extrinsic LLR vectors $\tilde{\mathbf{b}}_2^{\mathrm{e,UEC}}$ and $\tilde{\mathbf{b}}_3^{\mathrm{e,UEC}}$ jointly comprise two LLRs corresponding to each LLRs in the *a posteriori* LLR-vector $\tilde{\mathbf{b}}_1^{\mathrm{UEC}}$.

In a conventional decoder, the URC decoder and UEC trellis decoder are activated alternately, in a similar manner to the action of the turbo decoder of Section 6.2.1.2.

After the activation of one of the two decoders, they exchange their LLR-vectors through the interleaver π_1 and deinterleaver π_1^{-1}. More specifically, the extrinsic LLR-vector $\tilde{\mathbf{b}}_1^{\mathrm{e,URC}}$ gleaned from the URC decoder is passed through π^{-1}, yielding the *a priori* encoded UEC LLR-vectors $\tilde{\mathbf{b}}_2^{\mathrm{a,UEC}}$ and $\tilde{\mathbf{b}}_3^{\mathrm{a,UEC}}$. Furthermore, the extrinsic encoded LLR-vectors $\tilde{\mathbf{b}}_2^{\mathrm{e,UEC}}$ and $\tilde{\mathbf{b}}_3^{\mathrm{e,UEC}}$ generated by the UEC decoder are passed through π, yielding the *a priori* URC input LLR vector $\tilde{\mathbf{b}}_1^{\mathrm{a,URC}}$.

The performance of the UEC decoder can be improved by exploiting the fact that the conditional probabilities associated with different transitions $P(m|m')$ are not equal. The logarithm of these conditional transition probabilities $\ln[P(m|m')]$ may be added to the *a priori* transition probabilities $\tilde{\gamma}$ during the Log-BCJR [6]. Once a sufficient number of decoding iterations have been performed, the trellis decoder can output the vector of N *a posteriori* uncoded LLRs $\tilde{\mathbf{b}}_1^{\mathrm{UEC}} = [\tilde{b}_{1,k}^{\mathrm{UEC}}]_{k=1}^{N}$, which is passed to the unary decoder for recovering the symbol vector $\hat{\mathbf{x}}$. In analogy with the partitioning of the unary-encoded bits at the encoder into fixed length vectors $\mathbf{b}_1^{\mathrm{UEC}}$, there is also a buffer at the receiver for temporarily storing these LLRs, which pertain to symbols that are split between successive symbol-vectors, as described in Section 6.2.2.1. As described in Section 2.5.4, the unary decoder makes a hard decision for the LLRs in the vector $\mathbf{b}_1^{\mathrm{UEC}}$. The unary decoder then converts this resultant bit-vector into symbols $\tilde{\mathbf{x}}$, according to Table 2.6.

6.2.3 Fully parallel decoding algorithm

This section will describe the operation of the fully parallel iterative decoder, which was proposed in [9]. In Sections 6.2.1 and 6.2.2, we described how the turbo decoder and UEC-URC decoder may employ the Log-BCJR algorithm for alternately processing each component decoder, which iteratively exchange extrinsic information. By contrast, this section will show that the Log-BCJR algorithm can be replaced by the fully parallel iterative decoder, which carries out the tasks of both component decoders simultaneously.

Figure 6.7 depicts a fully parallel decoder for the LTE turbo code of Section 6.2.1.2. The fully parallel decoder is comprised of two component decoders, namely the upper decoder and the lower decoder, which are connected through an interleaver. In the fully parallel decoder, each component decoder is comprised of N decoding blocks, each of which correspond to a different trellis stage. The upper and lower decoder of Figure 6.7 are provided with the *a priori* LLR-vectors $\tilde{\mathbf{b}}_2^{\mathrm{u,a}}$, $\tilde{\mathbf{b}}_2^{\mathrm{l,a}}$ and $\tilde{\mathbf{b}}_3^{\mathrm{u,a}}$ from the channel. During the iterative decoding process, the upper and lower decoders exchange the extrinsic LLR-vectors $\tilde{\mathbf{b}}_1^{\mathrm{u,e}}$ and $\tilde{\mathbf{b}}_1^{\mathrm{l,e}}$ through the interleaver and deinterleaver, in the same manner as described in Section 6.2.1.2. Likewise, Figure 6.8 shows the fully parallel decoder applied to the UEC-URC decoder of Section 6.2.2.2. Here, in contrast to the fully parallel turbo decoder of Figure 6.7, there are twice as many decoder blocks for the URC decoder as for the UEC trellis decoder, since there are twice as many URC trellis stages

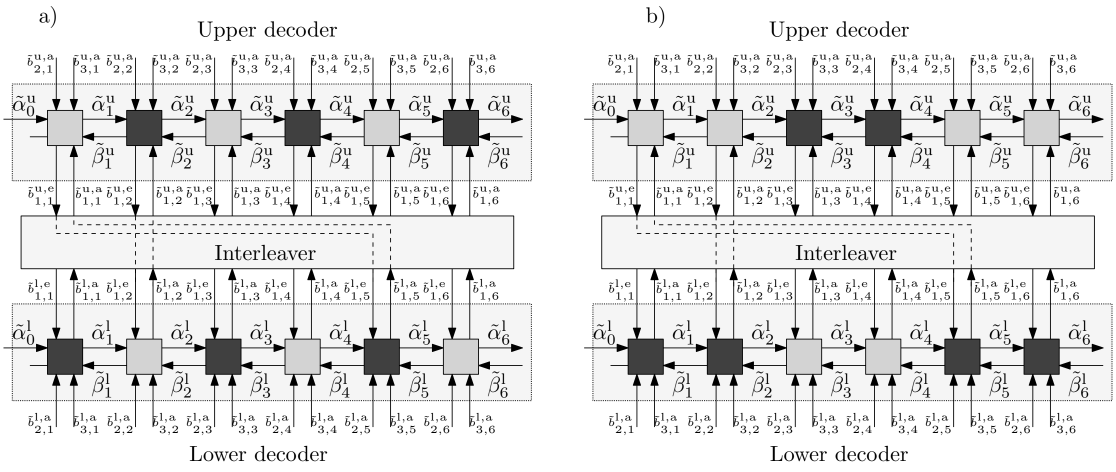


Figure 6.7: Fully parallel iterative decoders for the LTE turbo code. The shading of each decoding block indicates the odd-even grouping. a) State of the art fully parallel iterative decoder (adapted from [9]). b) Novel scheduling proposed in this chapter.

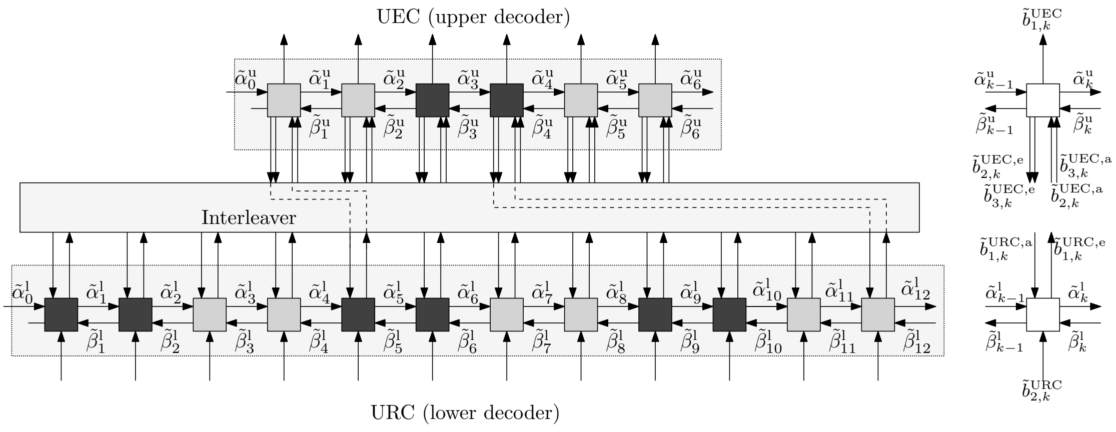


Figure 6.8: Novel scheduling proposed in this chapter for the UEC-URC scheme. The shading of each decoding block indicates the odd-even grouping.

as there are UEC trellis stages. This is because in the UEC encoder, each input bit $b_{1,k}^{\mathrm{UEC}}$, generates two output bits $b_{2,k}^{\mathrm{UEC}}$ and $b_{3,k}^{\mathrm{UEC}}$, which are encoded by the URC encoder, as shown by Figure 6.5. A further difference with respect to the turbo decoder is that for the fully parallel UEC-URC decoder, each UEC trellis decoder block is required to output the pair of extrinsic LLRs $\tilde{b}_{2,k}^{\mathrm{UEC,e}}$ and $\tilde{b}_{3,k}^{\mathrm{UEC,e}}$, instead of just one.

Since the fully parallel iterative decoder is comprised of a separate decoding block for each trellis stage, these trellis stages can be processed in parallel, facilitating high throughputs and low latency at the receiver. This is in contrast to the conventional Log-BCJR, where the trellis stages are processed in order, according to forward and backward recursions. The operation of the decoding blocks in Figures 6.7 and 6.8 is described in (6.1)-(6.5), where the time indicies t and $(t-1)$ are used to show in which time period the various values are used. Note that these equations have been re-arranged from those in [9], so that they match with the hardware schematic that will be used in Section 6.4. In (6.1)-(6.5), the max* operator is defined for two operands as $\max^*(a,b) = \max(a,b) + \ln(1+e^{-|a-b|})$, but it may be readily extended to more operands by exploiting the associative property.

$$\tilde{\gamma}_k^t(m_{k-1},m_k) = \sum_{j=1}^{L}\left[b_j(m_{k-1},m_k)\cdot \tilde{b}_{j,k}^{\mathrm{a},t-1}\right] + \log[P(m_k|m_{k-1})] \tag{6.1}$$

$$\tilde{\alpha}_k^t(m_k) = \max^*_{\{s_{k-1}|c(m_{k-1},m_k)=1\}}[\tilde{\gamma}_k^t(m_{k-1},m_k) + \tilde{\alpha}_{k-1}^{t-1}(m_{k-1})] \tag{6.2}$$

$$\tilde{\beta}_{k-1}^t(m_{k-1}) = \max^*_{\{s_k|c(m_{k-1},m_k)=1\}}[\tilde{\gamma}_k^t(m_{k-1},m_k) + \tilde{\beta}_k^{t-1}(m_k)] \tag{6.3}$$

$$\tilde{\delta}_k^t(m_{k-1},m_k) = \tilde{\gamma}_k^t(m_{k-1},m_k) + \tilde{\alpha}_{k-1}^{t-1}(m_{k-1}) + \tilde{\beta}_k^{t-1}(m_k) \tag{6.4}$$

$$\tilde{b}_{j,k}^{\mathrm{e},t} = \left[\max^*_{\{(s_{k-1},m_k)|b_j(m_{k-1},m_k)=1\}}[\tilde{\delta}_k^t(m_{k-1},m_k)]\right] - \left[\max^*_{\{(s_{k-1},m_k)|b_j(m_{k-1},m_k)=0\}}[\tilde{\delta}_k^t(m_{k-1},m_k)]\right] - \tilde{b}_{j,k}^{\mathrm{a},t-1} \tag{6.5}$$

Like the Log-BCJR algorithm [17], the equations of the fully parallel decoder comprise four sets of metrics. The $\tilde{\gamma}_k^t$ values of (6.1) represent the *a priori* probabilities of the transitions. These are calculated based on the *a priori* LLRs provided for each decoder block, either by the demodulator or by the interleaver in the previous time period, as well as based on the specific trellis structure $b_k(m_{k-1},m_k)$ employed by the scheme. The $\tilde{\alpha}_k^t$ and $\tilde{\beta}_k^t$ values of (6.2) and (6.3) are forwards and backwards state metrics, respectively. These are provided based on the $\tilde{\alpha}_k^{t-1}$ and $\tilde{\beta}_{k-1}^{t-1}$ values calculated by a neighbouring decoding block in the previous time period, as well as the $\tilde{\gamma}^t$ values from the current time period. Each decoder block outputs its $\tilde{\alpha}_k^t$ values to the next decoder block and the $\tilde{\beta}_k^t$ values to the previous decoder block, which are used in the subsequent time period. The $\tilde{\delta}_k^t$ values of (6.4) represent the *a posteriori* transition probabilities. These are calculated based on the *a priori* $\tilde{\alpha}_k^{t-1}$ and $\tilde{\beta}_{k-1}^{t-1}$ values provided by the neighbouring decoding blocks in the previous time period, as well as the $\tilde{\gamma}_k^t$ values from the current time period. These $\tilde{\delta}_k^t$ values are used to generate the extrinsic outputs $\tilde{b}_{j,k}^{\mathrm{e},t}$ of (6.5), which may be passed through the interleaver in order to assist the other constituent decoder in the iterative decoding process.

While all of the decoding blocks of both component decoders in Figures 6.7 and 6.8 can be operated at the same time, there are also other attractive activation orders. The first example of this is shown in Figure 6.7a, where two groups of blocks are shown, one with dark shading and one with light shading. This is known as odd-even activation, where each group of shaded blocks are operated in alternate time periods. More specifically, the dark shaded blocks are activated in odd indexed time periods, followed by the lightly shaded blocks in the even indexed time periods. Note that the t and $(t-1)$ notation of Equations (6.1)-(6.5) naturally support this activation order, since all inputs provided for a particular block in the previous time period are generated by the blocks having the opposite shading. Note that odd-even activation requires an odd-even interleaver, which connects dark shaded blocks to light shaded ones and vice versa. The LTE interleaver meets this requirement, as will be shown in Section 6.3.2. This odd-even scheduling reduces the complexity of the turbo decoder shown in Figure 6.7a by 50%, without compromising either its throughput or its error correction capability, as described in [9]. Note that the novel schedules shown in Figures 6.7b and 6.8 will be introduced in Section 6.3.

6.3 Algorithm adaptations

This section details the operation of the proposed enhancements to the fully parallel decoding algorithm of Section 6.2.3. The modifications proposed in this section are motivated by the constraints imposed by the associated hardware implementation. More specifically, to increase the clock frequency of the hardware and hence improve the throughput and latency, more pipelining [135] is required, although this delays the exchange of information through the decoder. A naive approach would be to increase the degree of pipelining without considering the negative impact on the algorithm's error correction performance. By contrast, this section describes a novel scheduling, which considers the effect of pipelining in hardware implementation upon its error correction performance. This novel scheduling will be shown in Section 6.4 to significantly improve the hardware efficiency attained, since it allows state metrics to propagate through the decoder faster. In this way, these improvements have been achieved by jointly considering the design of the iterative decoding algorithm and hardware. Indeed, we show that these enhancements improve its error correction performance, whilst increasing the achievable throughput of the entire system.

Section 6.3.1 commences by justifying the proposed modifications of Equations (6.1)-(6.5), in order to improve the scheduling of the fully parallel algorithm and aid its hardware implementation. Following this, Section 6.3.2 describes how the properties of the interleaver affect the operation of this modified fully parallel decoding algorithm. Section 6.3.3 characterizes the error correction performance of the proposed fully parallel algorithm, by comparing the number of decoding iterations required to those of the conventional Log-BCJR algorithm and the fully parallel decoding algorithm of [6]. Section 6.3.4 discusses the impact of adopting the reduced complexity approach of the Max-Log-BCJR algorithm [59] upon the error correction performance, showing that this can be mitigated by using extrinsic scaling [25]. Finally, Section 6.3.5 discusses the bit-widths required for the fixed point two's-complement numbers used inside the decoder.

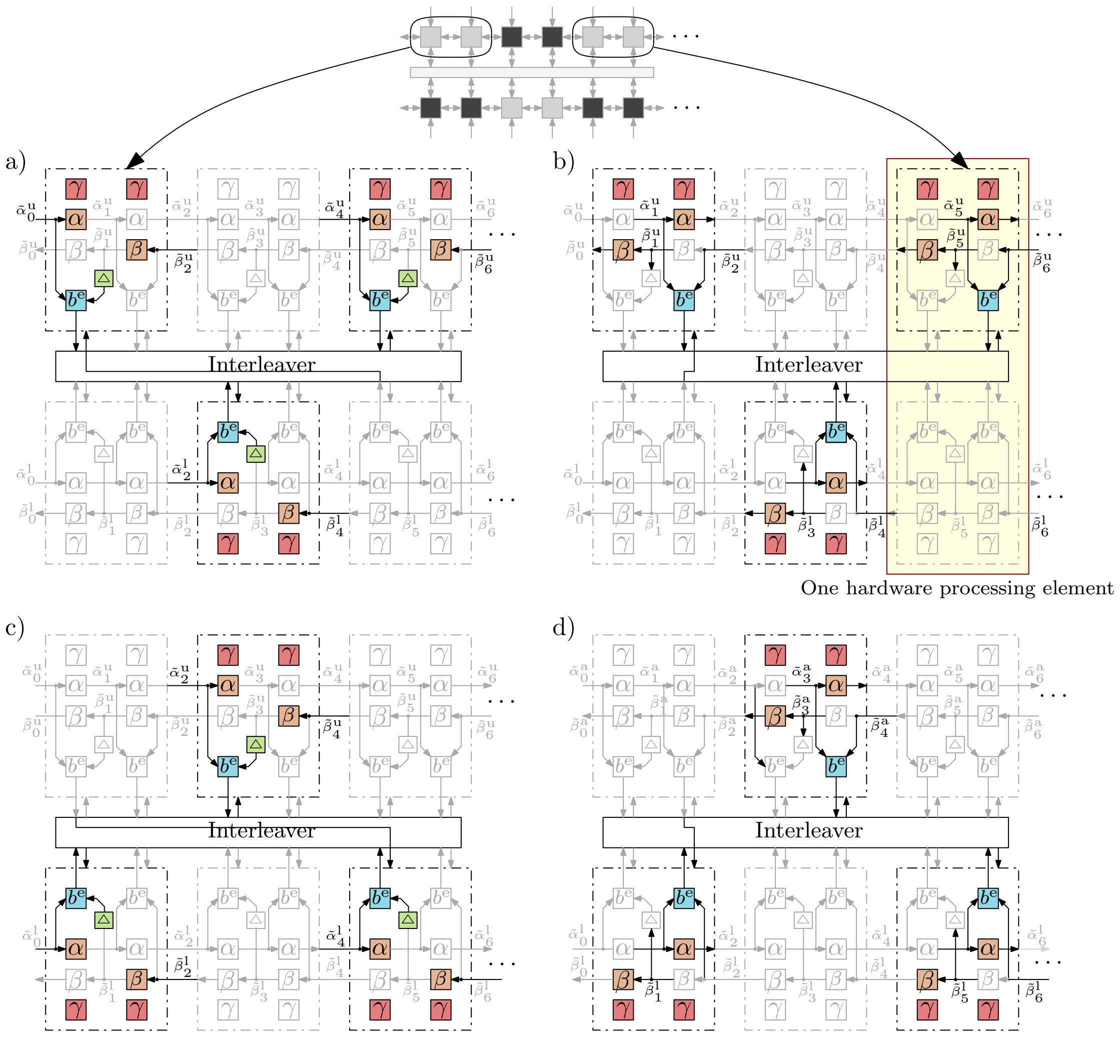


Figure 6.9: Proposed enhanced fully parallel decoding algorithm. The shaded portion shows the components that are implemented by each hardware decoder of Figure 6.16. Sub-figures a, b, c and d show the four successive steps of each decoding iteration.

6.3.1 Scheduling

This section will detail the novel approach to scheduling the operations of the fully parallel decoding algorithm, where the Equations (6.1)-(6.5) are reordered and transformed into (6.6)-(6.17), in order to facilitate its improved hardware implementation and to improve its error correction performance. Figure 6.9 graphically represents Equations (6.6)-(6.17), detailing which of the different operations of each algorithmic block are performed in which of the four successive time periods that constitute each complete

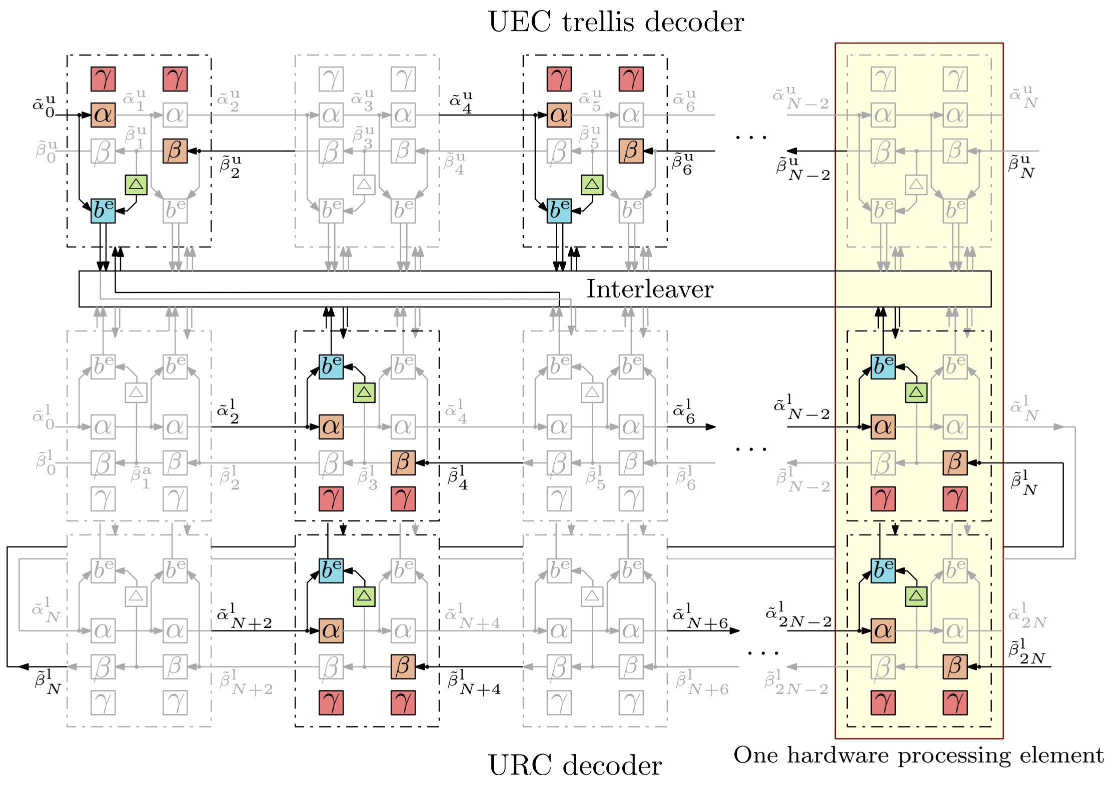


Figure 6.10: Proposed enhanced fully parallel decoding algorithm for the UEC-URC scheme. Here, only the first of four steps is shown, in analogy to that shown in Figure 6.9a.

decoding iteration. More specifically, Figure 6.9 shows the four time periods of the proposed scheduling, when applied to the LTE turbo code, while Figure 6.10 illustrates one of the four time periods in the schedule for the UEC-URC scheme. Note that since the UEC-URC scheme comprises twice as many URC trellis stages as that of the UEC trellis stages, the former are arranged into two rows of equal length in Figure 6.10.

Equations (6.1)-(6.5) of the fully parallel decoding algorithm of [9] have been transformed into (6.6)-(6.17) of the proposed modification by altering the order of operation. When applying an odd-even interleaver, each decoding iteration of the fully parallel decoding algorithm of [9] requires two time periods t, in which each algorithmic block is operated once, as described in Section 6.2.3. By contrast, Equations (6.6)-(6.17) result in each decoding iteration of the proposed modification requiring four time periods t, during which each algorithmic block is also operated once. Note that the changes relative to Equations (6.1)-(6.5) have been highlighted in (6.6)-(6.17). Some of the equations appear unchanged, but are shown in (6.6)-(6.17) because the time slot in which they are activated has changed.

In Figures 6.9 and 6.10, the dashed lines group the processing, which is required for each *pair* of algorithmic blocks. Specifically, the pairs of algorithmic blocks, which are being processed in the current time period t are surrounded by black dashed lines, while

if $t \mod 4 = 0, k \in 1, 5, 9, \ldots$ (upper), $k \in 3, 7, 11, \ldots$ (lower) or
if $t \mod 4 = 2, k \in 3, 7, 11, \ldots$ (upper), $k \in 1, 5, 9, \ldots$ (lower)

$$\tilde{\gamma}_k^t(m_{k-1}, m_k) = \sum_{j=1}^{L} \left[b_j(m_{k-1}, m_k) \cdot \tilde{b}_{j,k}^{\mathrm{a},t-1} \right] + \log[P(m_k|m_{k-1})] \tag{6.6}$$

$$\tilde{\alpha}_k^t(m_k) = \max{}^* \{s_{k-1}|c(m_{k-1}, m_k) = 1\}[\tilde{\gamma}_k^t(m_{k-1}, m_k) + \tilde{\alpha}_{k-1}^{t-1}(m_{k-1})] \tag{6.7}$$

$$\tilde{\beta}_{k}^t(m_k) = \max_{\{s_{k+1}|c(m_k, m_{k+1})=1\}}{}^* [\tilde{\gamma}_{k+1}^t(m_k, m_{k+1}) + \tilde{\beta}_{k+1}^{t-1}(m_{k+1})] \tag{6.8}$$

$$\tilde{\delta}_k^t(m_{k-1}, m_k) = \tilde{\gamma}_k^t(m_{k-1}, m_k) + \tilde{\alpha}_{k-1}^{t-1}(m_{k-1}) + \tilde{\beta}^{t-4}{}_k(m_k) \tag{6.9}$$

$$\tilde{b}_{j,k}^{\mathrm{e},t} = \left[\max_{\{(s_{k-1}, m_k)|b_j(m_{k-1}, m_k)=1\}}{}^* [\tilde{\delta}_k^t(m_{k-1}, m_k)] \right] \tag{6.10}$$

$$- \left[\max_{\{(s_{k-1}, m_k)|b_j(m_{k-1}, m_k)=0\}}{}^* [\tilde{\delta}_k^t(m_{k-1}, m_k)] \right] - \tilde{b}_{j,k}^{\mathrm{a},t-1} \tag{6.11}$$

if $t \mod 4 = 1, k \in 2, 6, 10, \ldots$ (upper), $k \in 4, 8, 12, \ldots$ (lower) or
if $t \mod 4 = 3, k \in 4, 8, 12, \ldots$ (upper), $k \in 2, 6, 10, \ldots$ (lower)

$$\tilde{\gamma}_k^t(m_{k-1}, m_k) = \sum_{j=1}^{L} \left[b_j(m_{k-1}, m_k) \cdot \tilde{b}_{j,k}^{\mathrm{a},t-1} \right] + \log[P(m_k|m_{k-1})] \tag{6.12}$$

$$\tilde{\alpha}_k^t(m_k) = \max_{\{s_{k-1}|c(m_{k-1}, m_k)=1\}}{}^* [\tilde{\gamma}_k^t(m_{k-1}, m_k) + \tilde{\alpha}_{k-1}^{t-1}(m_{k-1})] \tag{6.13}$$

$$\tilde{\beta}_{k-2}^t(m_{k-2}) = \max_{\{s_{k-1}|c(m_{k-2}, m_{k-1})=1\}}{}^* [\tilde{\gamma}_{k-1}^t(m_{k-2}, m_{k-1}) + \tilde{\beta}_{k-1}^{t-1}(m_{k-1})] \tag{6.14}$$

$$\tilde{\delta}_k^t(m_{k-1}, m_k) = \tilde{\gamma}_k^t(m_{k-1}, m_k) + \tilde{\alpha}_{k-1}^{t-1}(m_{k-1}) + \tilde{\beta}^{t-2}{}_k(m_k) \tag{6.15}$$

$$\tilde{b}_{j,k}^{\mathrm{e},t} = \left[\max_{\{(s_{k-1}, m_k)|b_j(m_{k-1}, m_k)=1\}}{}^* [\tilde{\delta}_k^t(m_{k-1}, m_k)] \right] \tag{6.16}$$

$$- \left[\max_{\{(s_{k-1}, m_k)|b_j(m_{k-1}, m_k)=0\}}{}^* [\tilde{\delta}_k^t(m_{k-1}, m_k)] \right] - \tilde{b}_{j,k}^{\mathrm{a},t-1} \tag{6.17}$$

---

the pairs of algorithmic blocks that are not being processed are surrounded by greyed dashed lines. Within the pairs of algorithmic blocks, individual operations are also shown, which are active when shaded, while the inactive ones are printed in grey. More specifically, the orange blocks marked α and β implement Equations (6.7), (6.13) and (6.8), (6.14), respectively. The blue blocks marked b_e represent (6.9), (6.15) and (6.11), (6.17), while the red blocks marked γ correspond to Equations (6.6) and (6.12). The green Δ block represents a memory element, which is used to store values that are not used immediately. These colours are consistently used throughout the remainder of this chapter to show the mapping between the algorithm and hardware implementation.

The proposed scheduling of this section preserves the odd-even activation order that was initially proposed in [9]. However, in the proposed algorithm, the odd-even scheduling applies to pairs of algorithmic blocks rather than to individual blocks. As will be described in Section 6.3.2, the LTE interleaver supports this scheduling, while the UEC-URC interleaver can be readily designed to support this scheduling.

The shaded region of Figures 6.9 and 6.10 shows the specific portion of the algorithm that is decoded by one hardware processing element, as will be discussed in Section 6.4. More specifically, Figure 6.9 illustrates the processing performed by three hardware processing elements for 12 algorithmic blocks of Figure 6.7, in four time steps.

Note that steps (a) and (b) of Figure 6.9 correspond to the light shaded pairs of blocks shown in Figures 6.7b and 6.8. These are processed in the first two of the four time periods. Meanwhile, steps (c) and (d) show how the dark shaded pairs of blocks of Figures 6.7b and 6.8 are processed in the remaining two time periods. The novel scheduling of this work allows information to pass through the two decoders at a higher rate than in the conventional fully parallel decoding algorithm. More specifically, after one iteration of the proposed algorithm, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ state metrics have propagated from $\tilde{\boldsymbol{\alpha}}_k$ to $\tilde{\boldsymbol{\alpha}}_{k+4}$ and $\tilde{\boldsymbol{\beta}}_{k+4}$ to $\tilde{\boldsymbol{\beta}}_k$, respectively. By contrast, for the odd-even fully parallel decoding algorithm of [9] requires two iterations for state metrics to propagate this distance.

The $\tilde{b}_k^{\mathrm{e},t}$ calculation in steps (a) and (c) uses the $\tilde{\boldsymbol{\alpha}}_{k-1}^{t-1}$ values, which have been calculated in the previous time period, but require the $\tilde{\boldsymbol{\beta}}_k^{t-4}$ values that were calculated in the previous iteration. This is because the $\tilde{b}_k^{\mathrm{e},t}$ and $\tilde{\boldsymbol{\beta}}_k^t$ calculations are performed in the same time period, and so the result of the $\tilde{\boldsymbol{\beta}}_k^t$ calculation is not ready in time for use in the $\tilde{b}_k^{\mathrm{e},t}$ calculation. Owing to this, the memory elements Δ are loaded with the output of the $\tilde{\boldsymbol{\beta}}$ calculation at the start of step (b). These $\tilde{\boldsymbol{\beta}}$ values are then stored until the start of step (a) in the next iteration, where they are used as inputs to the $\tilde{b}_k^{\mathrm{e},t}$ calculation. Note that the $\tilde{b}_k^{\mathrm{e},t}$ calculation of steps (b) and (d) do not suffer from this issue, since the $\tilde{b}_k^{\mathrm{e},t}$ calculation requires $\tilde{\boldsymbol{\beta}}_k^{t-2}$, which is also used in the previous time period.

It is worth noting that the paired operation proposed here is different to the radix-4 technique [13, 47], which is often used with conventional Log-BCJR decoders. Radix-4 decoders traverse two trellis stages in the same time period, by combining two trellis stages together and processing the combined trellis as one operation. By contrast, the paired operation proposed here requires two time periods for processing two trellis stages, but with the overlapping of their processing.

6.3.2 Interleaver

As previously discussed in Section 6.2.3, the fully parallel turbo decoder of Figure 6.7a benefits from odd-even operation, which enables a 50% reduction in computational complexity. More specifically, the blocks of different shading are operated alternately in successive time periods, so that outputs generated in one time period by blocks of one shading are consumed in the next time period by blocks of the other shading. In order to facilitate this, the interleaver of Figure 6.7a should be designed such that the lightly shaded blocks are only connected to the darkly shaded blocks. More specifically, the interleaver π_1 only connects blocks in the upper row having even indices to blocks in the lower row having even indices $\pi(i)$. Likewise, blocks having odd indices are only connected to blocks in the other row having odd indices. We may express this property as $\mod(\pi(i), 2) = \mod(i, 2)$, where $\mod(\cdot)$ is the modulo operator. This property is held by all of the 188 LTE interleavers, which have different frame lengths $N \in \{40, 48, ..., 6144\}$.

In the proposed algorithm of Section 6.3.1, the interleaver is still required to connect all lightly shared blocks to darkly shaded blocks, in order to maintain correct odd-even operation. Since the blocks of Figures 6.7b and 6.8 are arranged as pairs of the same colour, a different interleaver property is required relative to the algorithm of Figure 6.7a. More specifically, the first block in each pair of a specific shading must be interleaved to the first block of another pair of the other shading. Likewise, the second block in each pair of one shading must be interleaved to the second block of another pair of the other shading. This can be seen in Figure 6.7b, where the extrinsic LLR $\tilde{b}^{\mathrm{e}}_{1,1}$, which is output from the first block of a lightly shaded pair, is interleaved to $\tilde{b}^{\mathrm{a}}_{1,5}$ and input to the first block of a darkly shaded pair. This maximizes the number of time periods available for extrinsic LLRs to be generated, interleaved and be used as *a priori* LLRs in the connected algorithmic block. We may express the interleaver property required by the proposed scheduling as $\mod(\pi(i), 4) = \mod(i, 4)$, which we refer to as the *mod4 type A* property. This expression is also demonstrated by Figure 6.11, where the solid lines show the *mod4 type A* property, which will be exploited by the implementation of Section 6.4. Approximately half of the 188 LTE interleavers for different frame lengths $N \in \{40, 48, ..., 6144\}$ meet the *mod4 type A* property. The other half have a similar property, which we refer to as the *mod4 type B* property and which is shown by the dashed lines of Figure 6.11. When a *mod4 type B* interleaver is used, the odd-even scheduling property shown in Figure 6.7b is not entirely satisfied, since half of the extrinsic LLR connections through the interleaver will connect to blocks of the same shading. While a scheme using this interleaver can still be decoded using the scheduling of Figure 6.9 and the architecture of Section 6.4, there is a slight performance disadvantage, which will be explored in Section 6.3.3. This performance degradation is imposed by the extrinsic LLRs that suffer from an additional delay before they are used as *a priori* LLRs.

For the UEC-URC scheme of Figure 6.5, a mod4 interleaver is also required. Since we are free to design an interleaver for this proprietary scheme, a mod4 type A interleaver may be designed for all supported frame lengths. To maintain the mod4 properties of the interleaver, each extrinsic output $\tilde{b}^{\mathrm{UEC},e}_{2,k}$ and $\tilde{b}^{\mathrm{UEC,e}}_{3,k}$ from each UEC algorithmic block of a particular shading must be interleaved to a URC algorithmic block of the other shading. Likewise, the deinterleaver must route the extrinsic LLR $\tilde{b}^{\mathrm{URC,e}}_{1,k}$ produced by a URC algorithmic block of a particular shading to become either the *a priori* LLR $\tilde{b}^{\mathrm{UEC,a}}_{2,k}$ or $\tilde{b}^{\mathrm{UEC,a}}_{3,k}$ of a UEC algorithmic block having the opposite shading, using the inverse interleaving pattern.

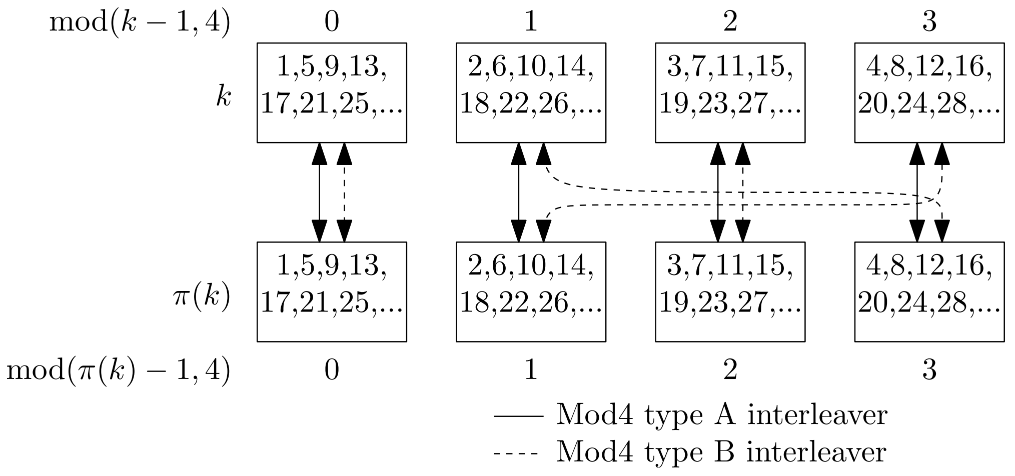


Figure 6.11: The mod4 property of the LTE interleaver, which is of either type A or type B for all 188 designs for the different frame lengths $N \in \{40, 48, ..., 6144\}$.

6.3.3 Error correction performance

Figure 6.12 shows the BER performance of the LTE turbo code of Figure 6.3 using the proposed decoding schedule of Figure 6.9, when using QPSK modulation for communication over an uncorrelated narrowband Rayleigh fading channel. Here, the proposed scheduling is compared with the Log-BCJR algorithm and the odd-even scheduling of [9], for both types of interleavers characterized in Figure 6.11. More specifically, the $N = 4864$-bit LTE interleaver is of *mod4 type A*, while the $N = 4800$-bit LTE interleaver is of *mod4 type B*. Figure 6.12 plots the BER performance for the Log-BCJR algorithm when employing 6 decoding iterations. Figure 6.12 shows that the same BER performance may be achieved using the proposed turbo decoding schedule of Figure 6.9 and a *mod 4 type A* interleaver, when performing 28 decoding iterations. However, when employing 28 iterations for a *mod4 type B* interleaver, the proposed schedule of Figure 6.9 can be seen to impose a modest 0.07 dB performance degradation at a BER of 10^{-5}. Likewise, when performing 28 decoding iterations, the odd-even scheduling of [9] suffers a more significant 0.3 dB performance degradation at a BER of 10^{-5}, compared to the proposed scheduling. Indeed, this schedule requires 42 decoding iterations to achieve the BER performance offered by the proposed schedule. Note that since each decoding iteration constitutes one activation of each algorithmic block, the comparison of the fully parallel schemes is fair in terms of decoding complexity. While requiring a more than 4-fold increase in the number of iterations required to match the BER performance of the conventional Log-BCJR may seem excessive, this hardware efficiency and throughput trade-off is required in order to achieve the highest throughputs and lowest latencies. Note that this trade-off is also exists with other high-throughput error correction decoders, such as LDPC decoders using the flooding scheduling [136], rather than layered belief propagation scheduling.

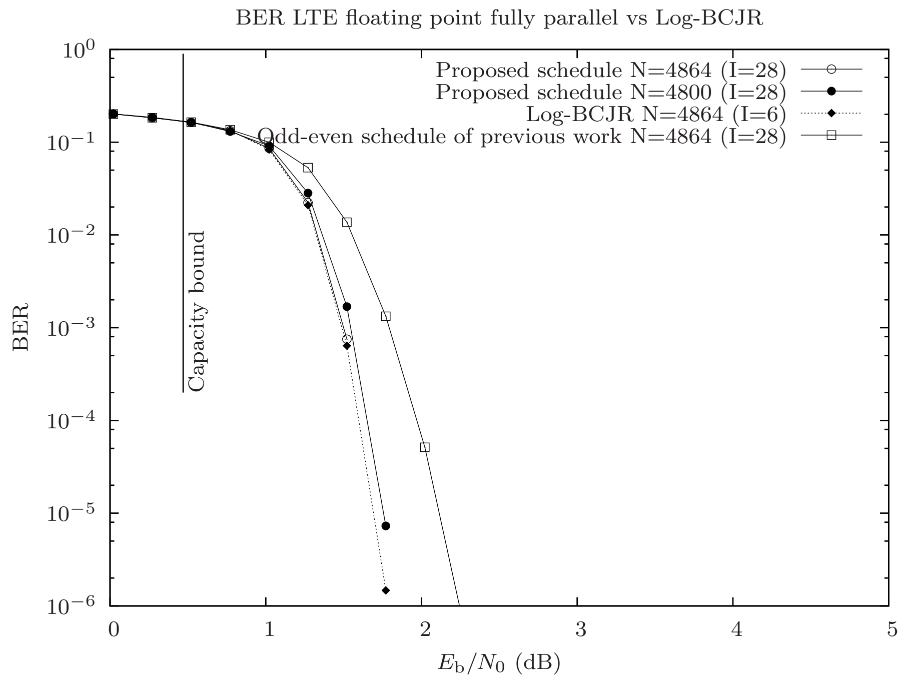


Figure 6.12: BER performance of the LTE turbo scheme of Figure 6.3, when employing various decoding algorithms and QPSK modulation for communication over an uncorrelated narrowband Rayleigh fading channel.

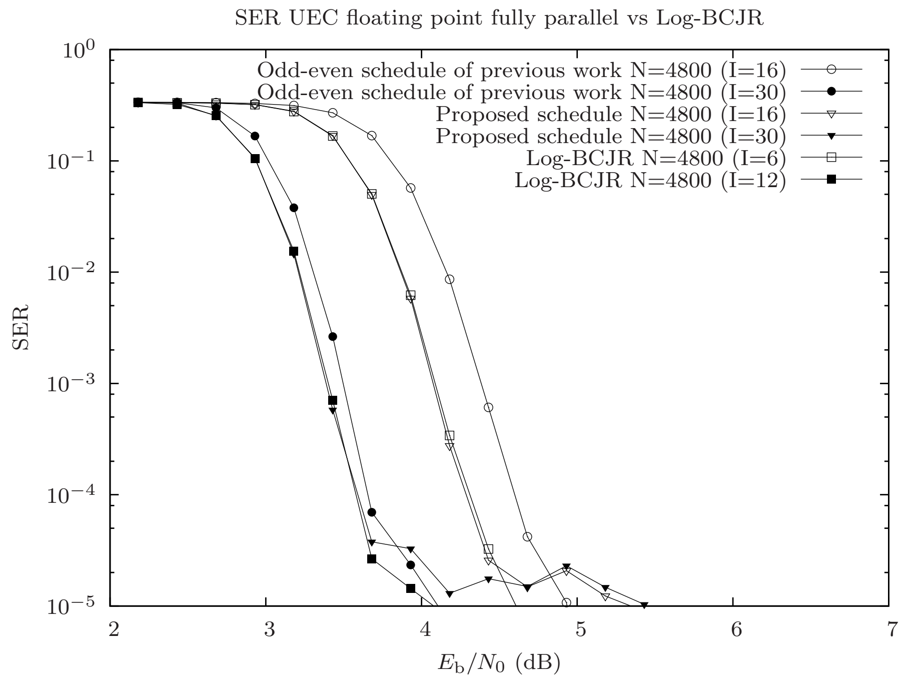


Figure 6.13: SER performance of the UEC-URC scheme of Figure 6.5, when employing various decoding algorithms and QPSK modulation for communication over an uncorrelated narrowband Rayleigh fading channel.

Figure 6.13 plots the SER performance of the UEC-URC scheme of Figure 6.5 using the proposed decoding schedule of Figure 6.10, when employing QPSK modulation for communication over an uncorrelated narrowband Rayleigh fading channel. Here, the generated symbols $\mathbf{x}$ obey a zeta distribution, having $p_1 = 0.8$, while the unary-encoded bits are partitioned into frames $\mathbf{b}_1^{\mathrm{UEC}}$ comprising $N = 4800$ bits, which corresponds to an average of 3134 symbols per frame. These schemes use random interleavers that obey the *mod 4 type A* constraint of Section 6.3.2. Figure 6.13 also compares the SER performance achieved, when employing the Log-BCJR algorithm and the fully parallel algorithm using the odd-even scheduling of [9]. The number of decoding iterations was chosen to match the performance achieved by the Log-BCJR after 6 and 12 iterations, which requires 16 and 30 iterations of the proposed schedule, respectively. Note that for the UEC-URC scheme, the proposed schedule requires only 16 decoding iterations to match the performance of the Log-BCJR after 6 iterations, which is significantly lower than the 28 required for the LTE turbo decoder. Similarly to the LTE scheme, our proposed scheduling improves the UEC-URC scheme by about 0.3 dB compared to the odd-even schedule after 16 iterations, as well as offering a marginal improvement after 30 iterations, since further iterations give marginal performance gains. As described in Section 3.4.2, Figure 6.13 shows that the UEC-URC scheme of this chapter also has an error floor, owing to its distance properties.

6.3.4 Extrinsic scaling

As described in Section 6.2.3, the algorithmic blocks of Figures 6.7 and 6.8 employ the max* operator, where $\max^*(a, b) = \max(a, b) + \ln(1 + e^{-|a-b|})$. However, in order to reduce the associated computational complexity, both the natural logarithm and the exponential operations are often omitted by using the approximation $\max^*(a, b) \approx \max(a, b)$. As discussed in Section 2.2.2, this approximation typically imposes an error correction performance penalty of about 1 dB depending on the scheme, although some of this loss can be mitigated by using extrinsic scaling [24,25]. This method multiplies the iteratively exchanged extrinsic LLRs by a constant value, which represents the decoder's reduced confidence in the values of the bits, due to the employment of sub-optimal decoding. Typically a value of 0.75 is chosen, since it can be implemented in a simple manner, when using the two's-complement fixed point number representation. In this case, a multiplication by 0.75 can be approximated by adding the extrinsic LLR right-shifted once, to itself but right-shifted twice. This yields the floor($\cdot$) of the multiplication by 0.75, owing to the limited precision of the fixed point numbers [130].

Figures 6.14 and 6.15 show the resultant BER and SER performance of the LTE turbo code scheme and UEC-URC schemes, respectively. Here, frame lengths of $N = 440$ bits are used, which is representative of the frame lengths presented in Section 6.5. Figures 6.14 and 6.15 compare the error correction performance obtained using the ideal max* operator, the max operator with an extrinsic LLR scaling factor of 0.75 (max-SE) and the max operator without scaling. For the LTE scheme, the extrinsic scaling reduces the performance gap between the ideal max* and the max operations to 0.2 dB, while the non-scaled max operator suffers from a 0.5 dB loss. The UEC-URC scheme has similar performance gaps, where the extrinsic scaling reduces the loss to 0.4 dB, while a loss of 0.8 dB is imposed without extrinsic scaling.

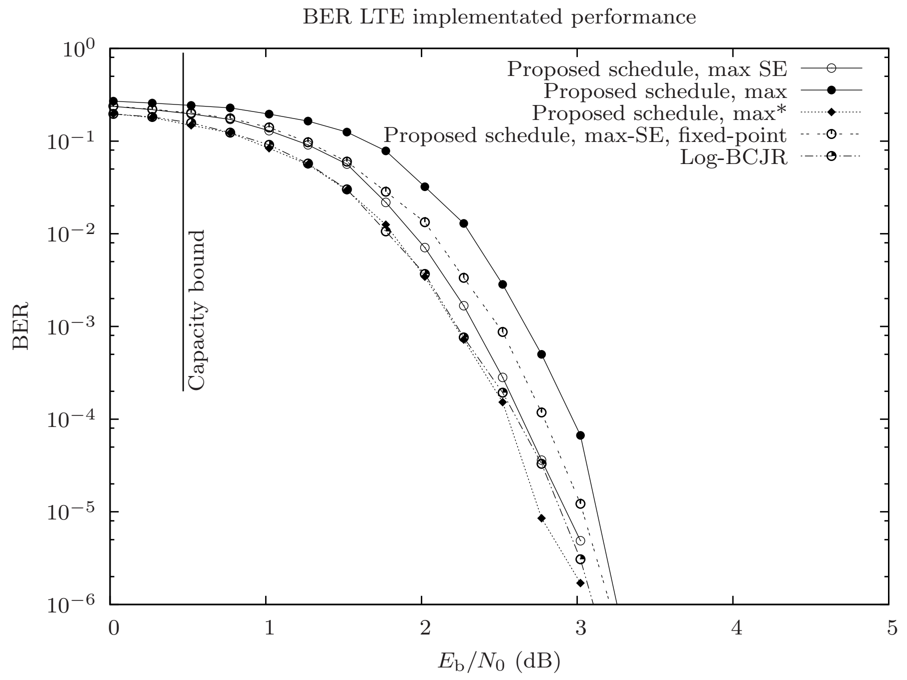


Figure 6.14: BER results for the LTE turbo scheme of Figure 6.3, using different extrinsic scaling and numerical representation techniques. Here, an uncorrelated narrowband Rayleigh fading channel and QPSK modulation is used. All schemes use a frame length of $N = 440$ bits, as well as 28 decoding iterations.

6.3.5 Number representation

In this section, the specific number representations used in the proposed decoders are discussed. While floating point numbers have been assumed throughout the simulations discussed in the previous sections, fixed point twos-complement number representations are preferred in hardware implementations, since this dramatically reduces the complexity. In particular, this section discusses a method conceived for reducing the dynamic range of the $\tilde{\boldsymbol{\alpha}}$ and $\tilde{\boldsymbol{\beta}}$ state metrics. This section also quantifies the fixed-point bit widths that are required, in order to approach the upper-bound performance of a floating point decoder.

When using two's-complement fixed-point numbers, the LLRs provided by the demodulator may be represented by fixed-point numbers having a bit width of w_d, the extrinsic LLRs may use a bit width of w_e, while the state metric $\tilde{\boldsymbol{\alpha}}$ and $\tilde{\boldsymbol{\beta}}$ values may be conveyed

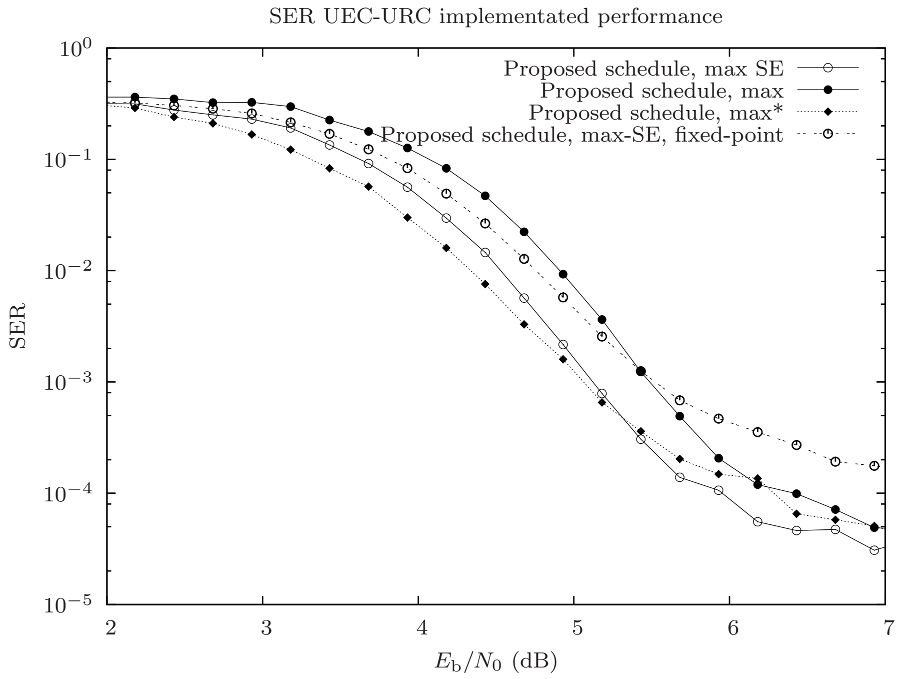


Figure 6.15: SER results for UEC-URC scheme of Figure 6.5, using different extrinsic scaling and numerical representation techniques. An uncorrelated narrowband Rayleigh fading channel and QPSK modulation is used. All schemes use partitioning to guarantee $N = 440$ bits in each frame, as well as 20 decoding iterations.

between adjacent algorithmic blocks using w_m bits. However, as shown in Equations (6.2) and (6.3), the values of the state metrics $\tilde{\boldsymbol{\alpha}}$ and $\tilde{\boldsymbol{\beta}}$ tend to grow without bound in successive iterations of the proposed decoding algorithm, owing to the accumulation of values that are typically positive, due to the action of the max operator. If however the values of the state metrics $\tilde{\boldsymbol{\alpha}}$ and $\tilde{\boldsymbol{\beta}}$ become excessively large, they may cause twos-complement overflow, where a small positive integer added to large positive integer erroneously results in a large negative integer. These errors can severely degrade the operation of the decoder, resulting in a very poor error correction performance. Since the state metrics tend to grow without bound, this overflow problem is inevitable unless an excessively high bit width w_m is employed or unless a technique is used for reducing the dynamic range of the $\tilde{\boldsymbol{\alpha}}$ and $\tilde{\boldsymbol{\beta}}$ values. In the proposed algorithm, the dynamic range of the state metrics are reduced to maintain a modest bit width w_m by using the clipping technique [130]. This technique relies on the observation that the absolute value of $\tilde{\boldsymbol{\alpha}}$ or $\tilde{\boldsymbol{\beta}}$ is not important, but rather it is the difference between the $\tilde{\boldsymbol{\alpha}}$ or $\tilde{\boldsymbol{\beta}}$ values produced for each trellis stage that conveys the relative probability of each state. Note that the number r of $\tilde{\boldsymbol{\alpha}}$ and $\tilde{\boldsymbol{\beta}}$ values produced for each trellis stage is given by $r_{\text{LTE}} = 8$ for the LTE code, $r_{\text{UEC}} = 4$ for the UEC trellis code and $r_{\text{URC}} = 2$ for the URC code of the UEC-URC scheme. An implication of this observation is that adding or subtracting the same value to all state metrics in a set of r number of $\tilde{\boldsymbol{\alpha}}$ or $\tilde{\boldsymbol{\beta}}$ values makes no difference as to the decoding algorithm's operation.

In the clipping technique, each processing element avoids overflow by using more than w_m bits for its internal calculations, but the state metrics $\tilde{\boldsymbol{\alpha}}$ and $\tilde{\boldsymbol{\beta}}$ are clipped to the bit widths of w_m. More specifically, the clipping method subtracts the $\tilde{\alpha}$ and $\tilde{\beta}$ value for the first trellis state from the rest of the values for each trellis stage, according to $\tilde{\alpha}_k(m_k) = \tilde{\alpha}'_k(m_k) - \tilde{\alpha}'_k(1)$ and $\tilde{\beta}_k(m_k) = \tilde{\beta}'_k(m_k) - \tilde{\beta}'_k(1)$. This guarantees an output where $\alpha_k(1) = 0$ and $\beta_k(1) = 0$, and where the remaining state metrics have lower values than they would otherwise. As a further step to ensure that the α and β values do not overflow, each α and β value is clipped so that it does not exceed the bit width w_m of the fixed-point number representation. Note that since clipping guarantees that we have $\tilde{\alpha}_k(1) = 0$ and $\tilde{\beta}_k(1) = 0$, it has the additional advantage that these values can be readily removed from subsequent calculations.

Figures 6.14 and 6.15 characterize the impact of using the fixed-point number representation on the BER and SER performance of the LTE scheme and UEC-URC scheme, respectively. Each figure compares the idealized floating point performance against the performance obtained when fixed-point numbers having particular bit-widths are employed.

More specifically, in the case of the LTE turbo decoder employing clipping, this chapter recommends the use of a bit width of $w_m = 6$ for the state metrics, $w_e = 6$ bits for the extrinsic LLRs, and $w_d = 6$ bits for the LLRs provided by the demodulator. Meanwhile, in the case of the UEC-URC decoder employing clipping, this chapter recommend the use of $w_m = 7$ bits for the state metrics, $w_e = 6$ bits for the extrinsic output, and $w_d = 6$ bits for the demodulator input. Note that wider bit widths are required for the state metrics of the UEC-URC scheme owing to the extended dynamic range that is caused by the non-equiprobable transitions and states in the UEC trellis, as described in Section 6.2.2.1. As shown in Figures 6.14 and 6.15, the bit widths recommended above offer a similar error correction performance to the floating point algorithm using the max approximation and extrinsic scaling. Note that if shorter bit widths are chosen, the decoder can exhibit a high error floor or a turbo cliff at a higher SNR.

6.4 FPGA implementation

This section details the FPGA implementation of the algorithm described in Section 6.3. By designing the algorithm and architecture jointly, an efficient exploitation of the decoder hardware can be achieved, as well as a powerful error correction performance. Furthermore, the proposed FPGA implementation is designed for facilitating the decoding of longer frame lengths than that which can be achieved with the aid of the previous design of [130] within the limited amount of hardware resources on an FPGA. This is achieved by sharing hardware processing elements between pairs of trellis stages, which also supports efficient pipelining and an increased clock frequency.

Section 6.4.1 commences by detailing the operation of each hardware processing element in the decoder. A generic hardware processing element is discussed, which could be applied to either the LTE or UEC-URC schemes of Figures 6.3 and 6.5 respectively. Following this, Section 6.4.2 describes the timing of the hardware processing elements, as well as how the algorithmic schedule of Section 6.3.1 is implemented on the hardware. Section 6.4.3 discusses the specific modifications required by the UEC-URC scheme of Figure 6.5 compared to the LTE scheme of Figure 6.3, detailing each of the computation units within each hardware processing element. Finally, Section 6.4.4 compares our jointly designed algorithm and hardware FPGA implementation with the previous implementations of the fully parallel decoder.

6.4.1 Decoding block top level

As described in Section 6.3.1, the $2N$ algorithmic blocks of the proposed LTE decoder are implemented using $N/2$ hardware processing elements, as indicated by the shaded area of Figure 6.9b Likewise, the $3N$ algorithmic blocks of the proposed UEC-URC decoder are implemented using $N/2$ hardware processing elements, as indicated by the shaded area of Figure 6.10. The schematic of each of the $N/2$ hardware processing block is shown in Figure 6.16. Each hardware processing element of Figure 6.16 is used for implementing the algorithmic blocks in both the top and bottom rows of the scheme. More specifically, when the hardware processing elements implement the scheduling of Figure 6.9, half of the hardware processing elements process the top decoder, while half the processing elements process the bottom decoder during steps (a) and (b). For steps (c) and (d), each hardware processing element switches to carrying out the other decoder's actions.

In this work, each hardware processing block is comprised of an $\tilde{\alpha}/\tilde{\beta}$ unit, a $\tilde{b}^{\mathrm{e}}$ unit, a $\tilde{\gamma}$ unit, as well as other multiplexers and registers. More specifically, in the $\tilde{\alpha}/\tilde{\beta}$ unit, a single piece of hardware is used to undertake the $\tilde{\alpha}$ and $\tilde{\beta}$ calculations of (6.7), (6.8), (6.13), (6.14) and Figure 6.9. This is in contrast to [130], where separate hardware was used for the $\tilde{\alpha}$ and $\tilde{\beta}$ calculations. This also allows a more heavily pipelined design, which increases the clock frequency, as will be detailed in Section 6.4.2. Furthermore, the $\tilde{\gamma}$ unit of Figure 6.16 is used to calculate equation (6.6) and (6.12), while the $\tilde{b}^{\mathrm{e}}$ unit is used to calculate the final extrinsic output of equation (6.11) and (6.17). This $\tilde{b}^{\mathrm{e}}$ unit is pipelined, enabling the hardware processing element to achieve very similar path lengths for the $\tilde{\alpha}/\tilde{\beta}$ unit and the two $\tilde{b}^{\mathrm{e}}$ unit stages, facilitating high clock frequencies and hence high throughputs and low latencies. The LLRs output from each $\tilde{b}^{\mathrm{e}}$ unit must be interleaved and input to the appropriate hardware processing element. This work employs a hardwired interleaver pattern, although our future work will develop a more

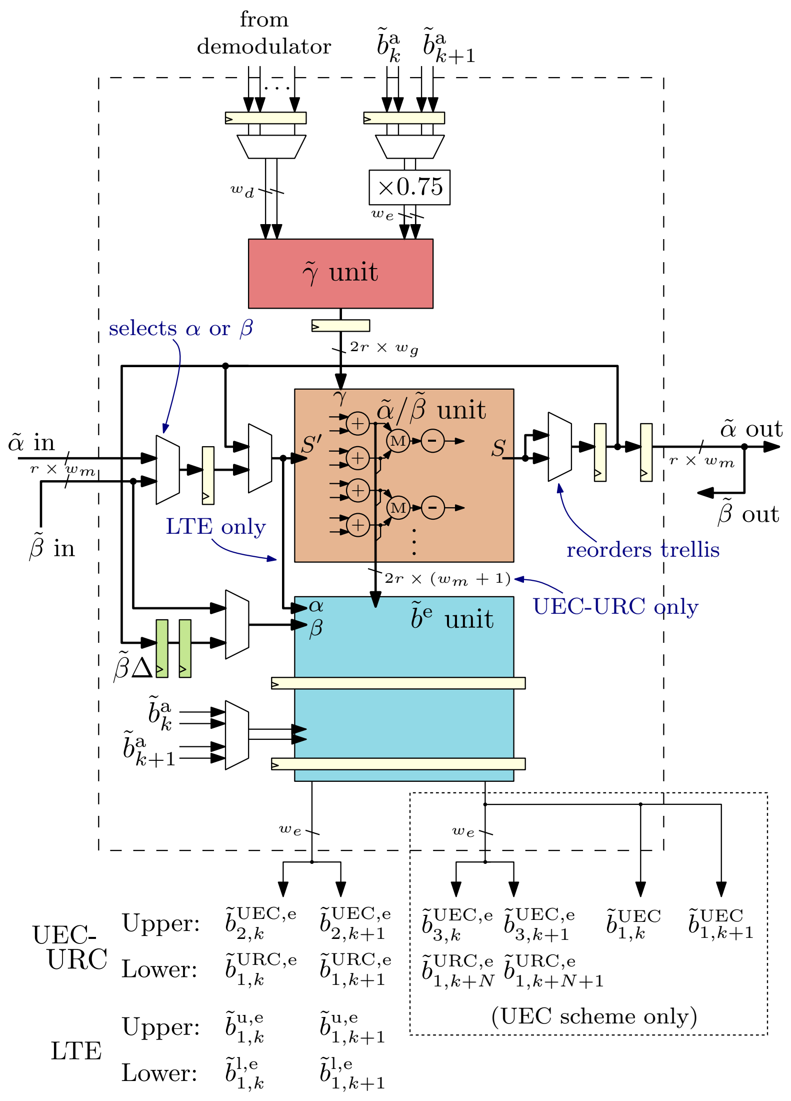

from demodulator
$\tilde{b}^{a}_{k}$
$\tilde{b}^{a}_{k+1}$
w_d
$\times 0.75$
w_e
$\tilde{\gamma}$ unit
selects α or β
$2r \times w_g$
$\tilde{\alpha}/\tilde{\beta}$ unit
$\tilde{\alpha}$ in
$r \times w_m$
$\tilde{\beta}$ in
S'
S
$\tilde{\alpha}$ out
$r \times w_m$
$\tilde{\beta}$ out
LTE only
reorders trellis
$2r \times (w_m + 1)$
UEC-URC only
$\tilde{b}^{e}$ unit
α
β
$\tilde{\beta}\Delta$
$\tilde{b}^{a}_{k}$
$\tilde{b}^{a}_{k+1}$
w_e
w_e
UEC-URC
Upper: $\tilde{b}^{UEC,e}_{2,k}$ $\tilde{b}^{UEC,e}_{2,k+1}$ $\tilde{b}^{UEC,e}_{3,k}$ $\tilde{b}^{UEC,e}_{3,k+1}$ $\tilde{b}^{UEC}_{1,k}$ $\tilde{b}^{UEC}_{1,k+1}$
Lower: $\tilde{b}^{URC,e}_{1,k}$ $\tilde{b}^{URC,e}_{1,k+1}$ $\tilde{b}^{URC,e}_{1,k+N}$ $\tilde{b}^{URC,e}_{1,k+N+1}$
(UEC scheme only)
LTE
Upper: $\tilde{b}^{u,e}_{1,k}$ $\tilde{b}^{u,e}_{1,k+1}$
Lower: $\tilde{b}^{l,e}_{1,k}$ $\tilde{b}^{l,e}_{1,k+1}$

Figure 6.16: Schematic of a hardware processing element. Here, $k \in \{1, 3, 5, ..., N-1\}$ for each of the $N/2$ hardware processing elements. The colour shading of the blocks matches with the colours of previous figures.

flexible interleaver that will enable the fully parallel turbo decoder to support different frame lengths and interleaver patterns at runtime. Since each hardware processing element can undertake decoding for both the upper and lower decoder, the interleaver also employs multiplexers for selecting between the hardwired interleaver connections of the upper and lower decoder.

Each hardware processing element is connected to its two neighbours, as well as to the interleaver. More specifically, the '*$\tilde{\alpha}$ in*' and '*$\tilde{\beta}$ out*' signals of Figure 6.16 connect to the neighbouring hardware processing element that processes trellis stages with lower indexes k. Meanwhile, the '*$\tilde{\alpha}$ out*' and '*$\tilde{\beta}$ in*' signals connect to the neighbouring hardware processing element that processes trellis stages with higher indexes k. These connections correspond to the $\tilde{\alpha}$ and $\tilde{\beta}$ connections between neighbouring pairs of algorithmic blocks in Figures 6.9b and 6.10. Since each hardware processing element undertakes decoding for the upper and lower decoder, these signals alternate between conveying the $\tilde{\alpha}^{\mathrm{u}}$ and $\tilde{\alpha}^{\mathrm{l}}$ values, or the $\tilde{\beta}^{\mathrm{u}}$ and $\tilde{\beta}^{\mathrm{l}}$ values in successive half iterations. In the case of the UEC-URC scheme, when a hardware processing element is undertaking the decoding of two URC trellis stages, the signals '$\tilde{\alpha}$ *in*' and '$\tilde{\alpha}$ *out*' are comprised of $\{\tilde{\boldsymbol{\alpha}}_{k-1}, \tilde{\boldsymbol{\alpha}}_{k+N-1}\}$ and $\{\tilde{\boldsymbol{\alpha}}_{k+1}, \tilde{\boldsymbol{\alpha}}_{k+N+1}\}$, respectively. Meanwhile, the signals '$\tilde{\beta}$ *in*' and '$\tilde{\beta}$ *out*' are respectively comprised of $\{\tilde{\boldsymbol{\beta}}_{k+1}, \tilde{\boldsymbol{\beta}}_{k+N+1}\}$ and $\{\tilde{\boldsymbol{\beta}}_{k-1}, \tilde{\boldsymbol{\beta}}_{k+N-1}\}$, in correspondence to the notation used within the URC decoder of Figure 6.10.

As shown in Equations (6.2) and (6.5), the $(\tilde{\alpha} + \tilde{\gamma})$ term is common to both the $\tilde{\alpha}$ calculation of (6.2) and the $\tilde{b}^{\mathrm{e}}$ calculation of (6.5). Owing to this, the adders that perform this operation can be efficiently shared between the $\tilde{\alpha}/\tilde{\beta}$ unit and the $\tilde{b}^{\mathrm{e}}$ unit, as shown in Figure 6.16 and as it will be detailed in Section 6.4.3.

Since the same hardware performs both the $\tilde{\alpha}$ and $\tilde{\beta}$ calculations in different clock cycles, multiplexers are required for selecting between the inputs $\tilde{\alpha}_n$ and $\tilde{\beta}_n$, as shown in Figure 6.16. Furthermore, a feedback path is employed across the $\tilde{\alpha}/\tilde{\beta}$ unit for allowing it to calculate two successive $\tilde{\boldsymbol{\alpha}}$ or $\tilde{\boldsymbol{\beta}}$ values in two successive clock cycles. More specifically, this feedback path allows the $\tilde{\alpha}/\tilde{\beta}$ unit to calculate $\tilde{\boldsymbol{\alpha}}_{k+1}$ and $\tilde{\boldsymbol{\beta}}_{k-1}$ based on the results of $\tilde{\boldsymbol{\alpha}}_k$ and $\tilde{\boldsymbol{\beta}}_k$, respectively. At the output of the $\tilde{\alpha}/\tilde{\beta}$ unit, a multiplexer reorders the output state metrics. In the case of the UEC-URC scheme, this is required since the UEC and URC trellises differ from each other. In the case of the LTE turbo scheme, this reordering is required, since the trellis connections for calculating the $\tilde{\boldsymbol{\alpha}}$ values are different from the trellis connections for calculating the $\tilde{\boldsymbol{\beta}}$ values. This reordering allows the $\tilde{\alpha}/\tilde{\beta}$ unit to have fixed connections for both the $\tilde{\alpha}$ and $\tilde{\beta}$ calculations.

Two register stages placed in series are used to implement the $\tilde{\beta}\Delta$ memory of Figure 6.9. Two registers are required, since the memory must hold the $\tilde{\boldsymbol{\beta}}$ values for both the upper and lower decoder during each algorithmic iteration. Another multiplexer is employed at the input of the $\tilde{b}^{\mathrm{e}}$ unit in order to select which $\tilde{\boldsymbol{\beta}}$ values it is provided with. For steps (b) and (d) of Figure 6.9, this multiplexer selects the $\tilde{\beta}_{k+1}$ values provided by the neighbouring processing element. Meanwhile, in steps (a) and (c) of Figure 6.9, the $\tilde{\beta}_k$ values are selected from the $\tilde{\beta}\Delta$ memory. Finally, in the case of the URC-UEC decoder, another multiplexer is used to provide the appropriate *a priori* LLRs into the second pipeline stage of the $\tilde{b}^{\mathrm{e}}$ unit, as required by Equation (6.5). More specifically, when the hardware processing element is undertaking URC decoding, the multiplexer selects either $\tilde{b}^{\mathrm{URC,a}}_{1,k}$ and $\tilde{b}^{\mathrm{URC,a}}_{1,k+N}$ or $\tilde{b}^{\mathrm{URC,a}}_{1,k+1}$ and $\tilde{b}^{\mathrm{URC,a}}_{1,k+N+1}$. Meanwhile, when the hardware processing element is undertaking UEC decoding, the multiplexer selects either $\tilde{b}^{\mathrm{UEC,a}}_{2,k}$ and $\tilde{b}^{\mathrm{URC,a}}_{3,k}$ or $\tilde{b}^{\mathrm{URC,a}}_{2,k+1}$ and $\tilde{b}^{\mathrm{URC,a}}_{3,k+1}$.

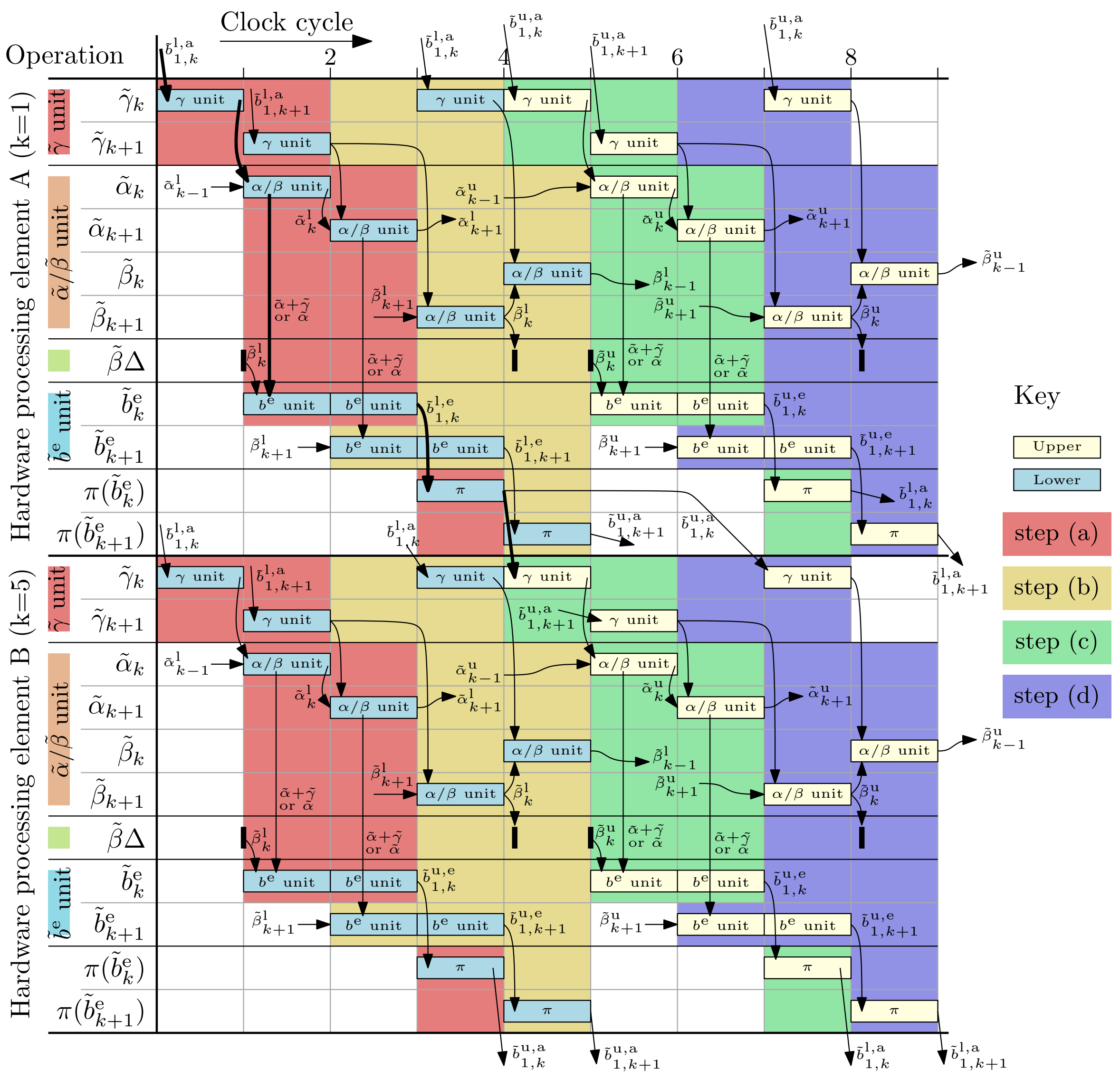


Figure 6.17: Timing diagram for two decoder blocks during one iteration of the LTE decoding algorithm. These decoding blocks perform the operations associated with different bit indexes of the upper or lower decoder in the same time periods. Owing to this, the output $\tilde{b}^{\mathrm{e}}_{1,k}$ provided by hardware processing element A is shown being interleaved to the input $\tilde{b}^{\mathrm{a}}_{1,k}$ of hardware processing element B. In the example of Figure 6.9, hardware processing element A processes the bits having the indices $k \in \{1, 2\}$, while hardware processing element B processes bits $k \in \{5, 6\}$. For each decoder, the diagram shows when each of the different tasks are undertaken, as well as the transfer of data between the tasks according to the dependencies between them. The background shading identifies which step of Figure 6.9 each operation corresponds to.

6.4.2 Scheduling

This section proposes a novel pipelining technique, in which the hardware processing elements use two clock cycles for processing each of the time periods (a)-(d) of Figure 6.9. Owing to this, each decoding iteration requires 8 clock cycles in the proposed decoder.

Figure 6.17 characterizes the timing of the various operations performed by the proposed decoder, illustrating the operation of two hardware processing elements, which are referred to as A and B. More specifically, Figure 6.17 shows when each hardware processing element performs each of the different operations of Figure 6.9. Note that the flow of data is the same in both Figures 6.9 and 6.17, but Figure 6.17 shows how the hardware implements this data flow on a clock cycle by clock cycle basis, allowing the pipelining techniques to be shown. Note that while Figure 6.17 adopts the notation of the LTE scheme, the operation of the UEC-URC scheme is identical.

Figure 6.17 shows how the hardware processing elements alternate between performing processing for the upper and lower decoders in successive half-iterations. More specifically, $N/4$ hardware processing elements, including blocks A and B, use four clock cycles for performing steps (a) and (b) of Figure 6.9, where each hardware processing element undertakes the processing tasks for two trellis stages. Following this, these hardware processing elements use four clock cycles to perform steps (c) and (d) of Figure 6.9, where each hardware processing element undertakes the processing for two trellis stages of the lower decoder. At the same time, the other $N/4$ hardware processing elements each perform steps (a) and (b) of Figure 6.9 for two trellis stages of the lower decoder, then perform steps (c) and (d) of Figure 6.9 for two trellis stages of the upper decoder.

As shown in Figure 6.7b, the upper decoder's extrinsic LLRs are interleaved to become the lower decoder's *a priori* LLR inputs. Accordingly, Figure 6.17 provides an example of how the extrinsic LLRs gleaned from processing element A may be passed to processing element B through the interleaver. More specifically, Figure 6.17 shows that an extrinsic LLR arriving from the $k = 1^{\text{st}}$ block in the upper decoder is interleaved and entered into the $k = 5^{\text{th}}$ block of the lower decoder. This example illustrates the critical path in the flow of information through the different operations, where the 4-stage pipeline transversing through the decoder is shown for one bit by the bold arrows. This example also shows that the *mod4* interleaver property of Figure 6.11 is key to facilitating the 4-stage pipelining technique proposed in this section. More specifically, if the extrinsic LLR produced by one algorithmic block in group 1 of Figure 6.11 was interleaved to an algorithmic block in group 0 of Figure 6.11, then the *a priori* LLRs output from the interleaver would arrive one clock cycle too late to be used. This can be seen in Figure 6.17, where the extrinsic LLR $\tilde{b}^{\text{l,e}}_{1,k+1}$ output from hardware processing element A is not interleaved to $\tilde{b}^{\text{u,a}}_{1,k+1}$ in time to be used at the start of step (c) by hardware processing element B, as may be required by an interleaver that does not have the *mod4* property. Instead of being consumed immediately, these *a priori* LLRs would need to be stored in an additional memory, until the next opportunity to use them arose. In addition to this additional hardware requirement, this delay would degrade the decoders error correction capability. By contrast, the *mod4 type A* interleaver ensures that these *a priori* LLRs do not need storing, since they are consumed immediately.

6.4.3 Scheme-specific implementation

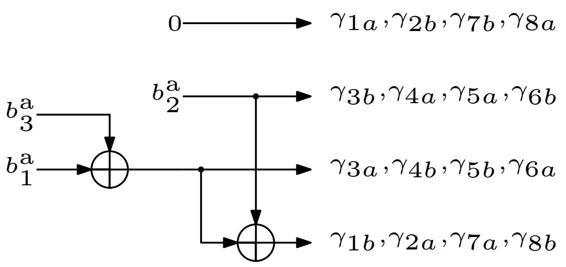
This section describes the specific features of the FPGA implementation that are used when implementing the LTE turbo decoder scheme of Figure 6.3, as well as the UEC-URC decoder scheme of Figure 6.5. In particular, this section details the components of each hardware processing block that behave differently for the pair of schemes considered.

In contrast to the LTE turbo code of Figure 6.3, Figure 6.6 shows that the UEC and URC decoders employ two different trellises, both of which must be implemented using the same hardware in order to maintain a high hardware efficiency. As described in Section 6.2.2.2, the UEC decoder comprises N trellis stages, where each trellis stage has $r_{\text{UEC}} = 4$ states. During the processing of each trellis stage, two extrinsic LLRs $\tilde{b}_{2,j}^{\text{e,UEC}}$ are generated. By contrast, the URC decoder comprises $2N$ trellis stages, where each trellis stage has $r_{\text{URC}} = 2$ states. Here, each trellis stage generates only one extrinsic LLR $\tilde{b}_{1,k}^{\text{e,URC}}$. In order to maintain a high hardware efficiency, each hardware processing unit is designed for processing one UEC trellis stage at a time, or process two URC trellis stages at a time. Since each UEC trellis stage corresponds to the same number of states, transitions and extrinsic LLRs as two URC trellis stages, both the UEC and URC trellises can be efficiency processed by the same hardware, as will be described in the following sections.

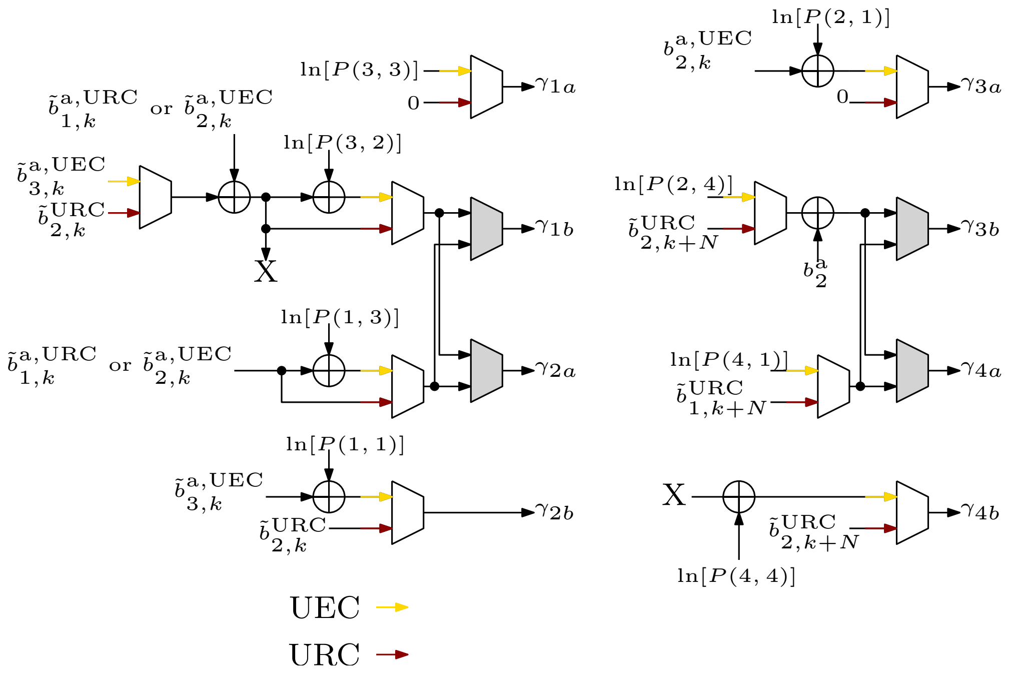
6.4.3.1 $\tilde{\gamma}$ unit

Figure 6.18 illustrates the $\tilde{\gamma}$-calculation unit both for the LTE and for the UEC-URC implementation, which produces the *a priori* transition probabilities $\tilde{\gamma}$ according to (6.6) or (6.12). In the case of the UEC-URC scheme, Figure 6.18(b) employs the four multiplexers with grey shading on the $\tilde{\gamma}_{1b}$, $\tilde{\gamma}_{2a}$, $\tilde{\gamma}_{3b}$ and $\tilde{\gamma}_{4a}$ outputs, in order to switch the pairs of $\tilde{\gamma}$ values that are output depending on whether the $\tilde{\alpha}/\tilde{\beta}$ unit is currently decoding $\tilde{\alpha}$ values or $\tilde{\beta}$ values. For example, in the URC trellis of Figure 6.6b, the $\alpha_k(1)$ calculation requires $\tilde{\gamma}(1,1)$ and $\tilde{\gamma}(2,1)$, while the $\beta_k(1)$ calculation requires $\tilde{\gamma}(1,1)$ and $\tilde{\gamma}(1,2)$, necessitating the $\tilde{\gamma}$ unit to swap some of the $\tilde{\gamma}$ values. Note that the corresponding multiplexers are not required for the LTE turbo decoder scheme, since the different connections for the $\tilde{\boldsymbol{\alpha}}$ values and $\tilde{\boldsymbol{\beta}}$ values can be handled by the reordering scheme of Figure 6.16.

In the UEC-URC scheme, each hardware processing element has to carry out both UEC and URC decoding, requiring the unshaded multiplexers of Figure 6.18(b) to switch


0
$\gamma_{1a}, \gamma_{2b}, \gamma_{7b}, \gamma_{8a}$
b_2^{a}
$\gamma_{3b}, \gamma_{4a}, \gamma_{5a}, \gamma_{6b}$
b_3^{a}
b_1^{a}
$\gamma_{3a}, \gamma_{4b}, \gamma_{5b}, \gamma_{6a}$
$\gamma_{1b}, \gamma_{2a}, \gamma_{7a}, \gamma_{8b}$

(a) LTE


$\ln[P(3,3)]$
0
γ_{1a}
$\tilde{b}_{1,k}^{\mathrm{a,URC}}$ or $\tilde{b}_{2,k}^{\mathrm{a,UEC}}$
$\ln[P(3,2)]$
$\tilde{b}_{3,k}^{\mathrm{a,UEC}}$
$\tilde{b}_{2,k}^{\mathrm{URC}}$
γ_{1b}
X
$\ln[P(1,3)]$
$\tilde{b}_{1,k}^{\mathrm{a,URC}}$ or $\tilde{b}_{2,k}^{\mathrm{a,UEC}}$
γ_{2a}
$\ln[P(1,1)]$
$\tilde{b}_{3,k}^{\mathrm{a,UEC}}$
$\tilde{b}_{2,k}^{\mathrm{URC}}$
γ_{2b}
UEC
URC
$\ln[P(2,1)]$
$b_{2,k}^{\mathrm{a,UEC}}$
0
γ_{3a}
$\ln[P(2,4)]$
$\tilde{b}_{2,k+N}^{\mathrm{URC}}$
b_2^{a}
γ_{3b}
$\ln[P(4,1)]$
$\tilde{b}_{1,k+N}^{\mathrm{URC}}$
γ_{4a}
X
$\tilde{b}_{2,k+N}^{\mathrm{URC}}$
γ_{4b}
$\ln[P(4,4)]$

(b) UEC-URC

Figure 6.18: Schematic of the $\tilde{\gamma}$-calculation unit of Figure 6.16 for the (a) LTE and (b) UEC-URC scheme.

between the different inputs that are necessary for UEC and URC decoding. This is necessary since the transitions of the UEC and URC trellises to not share the same inputs and outputs. Furthermore, the UEC trellis decoder also requires the addition of the conditional transition probabilities $\ln[P(m|m')]$, as described in Section 6.2.2.2. In the proposed implementation, these conditional transition probabilities are stored using 6-bit fixed point numbers.

6.4.3.2 $\tilde{\alpha}/\tilde{\beta}$ unit

The $\tilde{\alpha}/\tilde{\beta}$ units are characterized by the orange boxes in the top half of Figure 6.19, which detail the internal operation of the orange block of Figure 6.16. In the case of the UEC-URC scheme, each hardware processing element switches between the UEC trellis decoder and URC decoder every 4 clock cycles. Accordingly, the $\tilde{\alpha}/\tilde{\beta}$ unit is designed to switch between carrying out all of the operations of either one UEC trellis stage

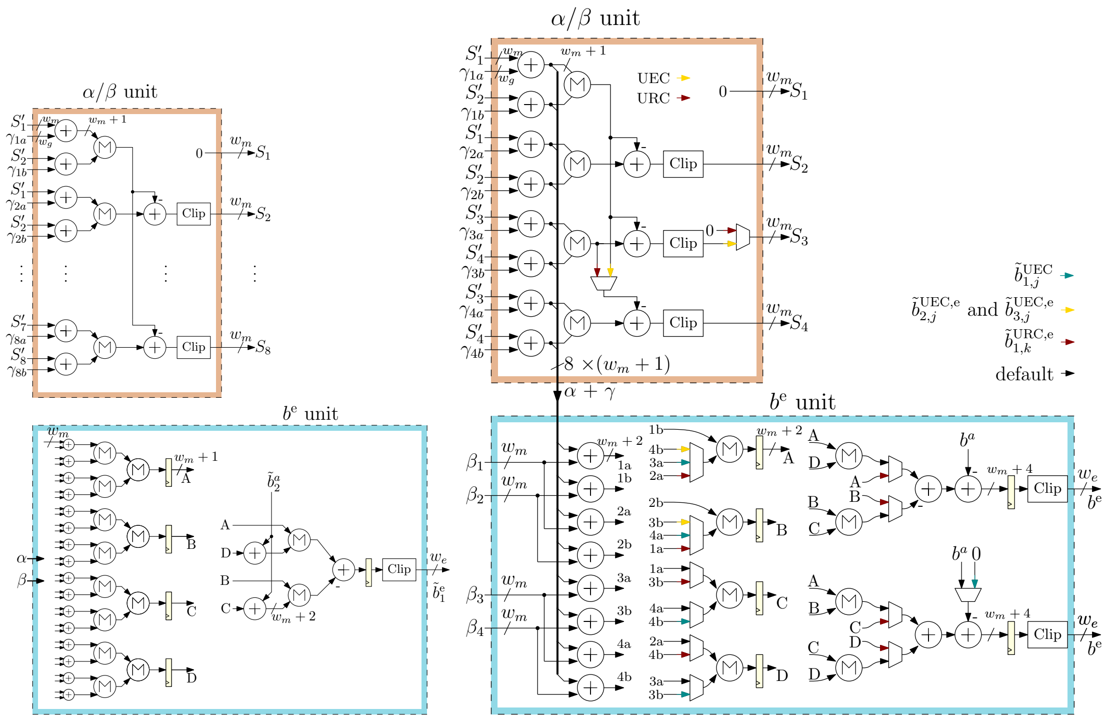

(a) LTE (b) UEC-URC

Figure 6.19: The $\tilde{\alpha}/\tilde{\beta}$ unit and $\tilde{b}^{\mathrm{e}}$ unit for (a) the LTE turbo scheme, and (b) the UEC-URC scheme. Here, the coloured outlines correspond to the similarly coloured blocks of Figures 6.10, 6.16 and 6.17.

or of two URC trellis stages. As shown in Figure 6.6, two URC trellis stages can be processed at the same time to give a similar structure to one UEC trellis stage, but with some different transition connections. More specifically, the trellis of Figure 6.6a can be converted into two copies of the trellis of Figure 6.6b by simply switching the transitions associated with the central two states. Furthermore, by switching these two central states, the trellis may be mirrored from left to right, allowing the same connections to be used for both $\tilde{\alpha}$ and $\tilde{\beta}$ calculations. This switching of trellis states is implemented using the multiplexers of Figure 6.16. In the case of the LTE scheme, state switching is also undertaken by multiplexers shown in Figure 6.16, in order to produce a mirrored trellis, allowing the same connections to be used for both $\tilde{\alpha}$ and $\tilde{\beta}$ calculations.

The $\tilde{\alpha}/\tilde{\beta}$ unit's inputs S' all have bit widths of w_m, as investigated in Section 6.3.5. However, the bit-widths within the $\tilde{\alpha}/\tilde{\beta}$ unit grow following successive additions, in order to ensure that they do not overflow. Following this, clipping is employed to restore the bit widths of the outputs S to w_m. Note that the two multiplexers in the UEC-URC $\tilde{\alpha}/\tilde{\beta}$ unit of Figure 6.19(b) change the operation of the clipping circuit, depending on whether outputs 3 and 4 are part of the same trellis as outputs 1 and 2, as in the case of the UEC, but not for the URC.

6.4.3.3 $\tilde{b}^{\mathrm{e}}$ unit

Figure 6.19 illustrates the $\tilde{b}^{\mathrm{e}}$-calculation unit of both the LTE and UEC-URC schemes, which is used for generating the extrinsic LLRs. Here, the $\tilde{b}^{\mathrm{e}}$ unit is pipelined into two stages, in order to facilitate a high clock frequency and hence a high hardware efficiency. More specifically, this pipelining ensures that the propagation delay in each of the two $\tilde{b}^{\mathrm{e}}$ stages is of a similar length to those of the other parts of the decoder. This pipelining scheme was detailed in Section 6.4.2, which considered the clock cycle by clock cycle scheduling of each hardware processing element. The connections required within the $\tilde{b}^{\mathrm{e}}$ unit for generating extrinsic LLRs are dictated by the specifics of the LTE, URC or UEC trellis. More particularly, the input and output bits associated with each transition identify, which specific state and transition metrics have to be combined, according to (6.9) and (6.15). In the case of the UEC-URC scheme, the required combinations of state and transition metrics are different, depending on whether the $\tilde{b}^{\mathrm{e}}$ unit is generating $\tilde{b}_{1,k}^{\mathrm{UEC}}$, $\tilde{b}_{2,k}^{\mathrm{e,UEC}}$ or $\tilde{b}_{1,k}^{\mathrm{e,URC}}$. The required flexibility is provided by multiplexers in the first pipeline stage of Figure 6.19, which are used for selecting which particular pairs of metrics are input to the max calculation, as well as by multiplexers in the second stage, which bypass the second max calculation, since it is not required in the case of the URC. The final subtraction in (6.11) and (6.17) is undertaken before the register in the second pipeline stage. Since the second $\tilde{b}^{\mathrm{e}}$ output can be used for generating either the *a posteriori* LLR b_1^{UEC} or the extrinsic LLR $\tilde{b}_3^{\mathrm{e,UEC}}$, a multiplexer is required for selection between ‘0’ or the *a priori* LLR $\tilde{b}_3^{\mathrm{a,UEC}}$, respectively.

In the case of the LTE scheme, the operations of Figure 6.19(a) have been reordered [130] compared to those of the UEC-URC scheme, in order to reduce the critical path-length, but at the expense of a slightly increased hardware resource requirement. This technique dispenses with adding $\tilde{\gamma}$ values during the $\tilde{\delta}$ calculation of (6.9) and (6.15) as well as with subtracting the *a priori* LLR in the $\tilde{b}^{\mathrm{e}}$ calculation of (6.11) and (6.17). Instead, the LLR $\tilde{b}_2^{\mathrm{a}}$ is added during the $\tilde{b}^{\mathrm{e}}$ calculation in Figure 6.19. Note that this technique cannot be used for the UEC-URC scheme, owing to the conditional probabilities $\ln[P(m_{k-1}|m_k)]$, which arise from the non-equiprobable transitions of the UEC code. Instead, the UEC-URC scheme passes the $(\tilde{\alpha}+\tilde{\gamma})$ values from the $\tilde{\alpha}/\tilde{\beta}$ unit to the $\tilde{b}^{\mathrm{e}}$ unit, in order to reduce the required hardware resources.

6.4.4 Comparison to scheduling in the existing state-of-the-art implementations

Table 6.1 characterizes the performance of the proposed algorithmic and hardware scheduling, and compares this both to the state-of-the-art Log-BCJR [13] implementation and to the fully parallel decoder of [9, 130], when implementing the LTE turbo decoder. Compared to the fully parallel turbo decoder of [9, 130], our proposed implementation requires 4x as many clock cycles per iteration, owing to two design decisions.

Table 6.1: Comparison of the proposed approach with the state-of-the-art Log-BCJR decoder and the fully parallel decoder of [9], when used to decode the $N = 6144$-bit LTE turbo code.

	Estimation in [130]		Estimation in this work
	State-of-the-art LTE algorithm of [13]	LTE fully parallel of [9, 129, 130]	Proposed paired schedule
Number of parallel hardware processing elements	64	$N^{\star}$	$N/2$
Clock cycles per iteration T	$N/32$	2	8
Clock cycle duration D	3 stages	6 stages	3 stages
Complexity per decoding iteration C	$320N$	$155N$	$155N$
Decoding iterations I	6	39	28
Combinational requirement X	14144	$80N$	$30N$
Register requirement Y	1792	$21N$	$26N$
RAM requirement Z	$14N/3$+8320	0	0
Overall throughput $N/(T \cdot D \cdot I)$	16/9 (1x)	$N/468$ (7.38x)	N/672 (5.14x)
Overall latency ($T \cdot D \cdot I$)	$9N/16$ (7.38x)	468 (1x)	672 (1.44x)
Overall complexity ($C \cdot I$)	$1920N$ (1x)	$6045N$ (3.15x)	$4340N$ (2.26x)
Overall resource $w \max(\frac{X}{2}, \frac{Y}{2}, \frac{Z}{160})$	$7072w$ (1x)	$40Nw$ (34.8x)	$15Nw$ (13.0x)
Hardware efficiency (throughput/resource)	2.51×10^{-4} (1x)	5.34×10^{-5} (0.21x)	9.92×10^{-5} (0.40x)

*The work of [130] uses $2N$ processing elements, which offers a reduced power consumption but increased area.

Firstly, each processing element in the proposed approach decodes two trellis stages, while the processing elements of [9, 130] only decode a single trellis stage. This allows our proposed decoder to support longer frame lengths N than the design of [9, 130] using the same amount of hardware. Secondly, the hardware scheduling of the proposed approach contains more pipelining, which results in a clock cycle duration D that is half that of [130]. Furthermore, our additional pipelining improves the hardware reuse, therefore reducing the required hardware resources whilst increasing hardware efficiency, without reducing the throughput.

The complexity C per decoding iteration of the proposed approach is the same as that of [9, 130], as shown in Table 6.1. This is because both implementations perform the same operations of (6.1)-(6.5), although here a different activation order for these equations is proposed. As explored in Section 6.3.3, our novel scheduling means that the proposed approach achieves the same BER performance as the benchmarkers using only

28 decoding iterations I, in contrast to the 39 required by the fully-parallel turbo decoder of [130].

Table 6.1 also characterizes the breakdown of the hardware resource requirements into combinational X, registers Y and RAM Z requirements. As detailed in [9], the combinational requirement X is obtained by quantifying the number of adder and max circuits employed, while, the register requirement Y is quantified in terms of the number of values that must be held between successive clock cycles. Finally, Z is determined by quantifying the number of values that must be stored in RAM. The ASIC implementation of [130] comprises $2N$ hardware processing elements, each dedicated to one trellis stage of either the upper or lower decoder. This decoder operates on the basis of the odd-even activation order and so each hardware processing element is inactive in alternate clock cycles. The reduced switching of this approach reduces energy consumption, but leads to a larger hardware requirement. Meanwhile, the FPGA implementation of [129] comprises N hardware processing elements, each of which alternate between performing decoding for both the upper and lower decoder. By contrast, the proposed decoder comprises $N/2$ hardware processing elements, each or which processes two trellis stages of the upper decoder and two trellis stages of the lower decoder. This difference leads to a combinational requirement X that is half of that required by the fully parallel turbo decoder of [129], for a given frame length N. The combinational requirement X is further reduced in the proposed decoder by reusing the same hardware for the $\tilde{\alpha}$ and $\tilde{\beta}$ calculations, as discussed in Section 6.4.2. Note that owing to its increased use of pipelining, the proposed design requires more registers per hardware processing element than the fully-parallel turbo decoder of [9, 130]. However, Section 6.5 will demonstrate that the combinational requirement is the limiting factor in the FPGA implementation of both the proposed and benchmarker designs. Owing to this, the use of more registers to achieve superior performance represents a more desirable utility of the FPGA resources.

The combinational requirement X, register requirement Y and RAM requirement Z may be combined to predict the overall resource requirement of each design considered. Here, the Altera Stratix IV FPGA is targeted, which is comprised of a large number of Adaptive Logic Modules (ALMs). Each ALM comprises two single-bit registers and two single-bit adders with a corresponding Look-Up Table (LUT). Since the combinational requirement X represents the number of additions and max operations required, we may estimate that $wX/2$ ALMs are required to fulfil the combinational requirement, where w is the average bit-width used in the decoder. Likewise, the number of ALMs required to fulfil the register requirement may be estimated by $wY/2$. In the Stratix IV device targeted by this work, there are 160 bits of RAM available for each ALM. Therefore, the total resource requirement may be approximated by $w\max(\frac{X}{2}, \frac{Y}{2}, \frac{Z}{160})$. Note that we combine the three elements using the maximum, since the FPGA has a limited amount of each hardware resource type and one of these will inevitably impose the ultimate limitation, as the degree of parallelism is increased [137]. Note that this analysis can only provide an approximation, but it may be applied equally to the three designs considered, allowing a fair comparison. This analysis provides a lower bound on the required hardware, since it does not consider the resources required by multiplexers or the restrictions imposed by routing, which may lead to the inefficient exploitation of resources. As shown in Table 6.1, this analysis reveals that the overall resource requirement of the proposed design is just 25% of that of the fully-parallel turbo decoder of [9, 130].

Table 6.1 shows that the overall throughput of our proposed design is 5 times greater than that of the state of the art Log-BCJR decoder of [13], but 0.70 times that of the fully parallel turbo decoder of [130] for a given frame length N. Although the proposed design requires fewer decoding iterations than that of [130], this reduced throughput may be expected since each decoding iteration of the proposed design has a duration TD which is double that of [130]. Note that this increased decoding iteration duration TD also results in a slightly longer latency than that of [130]. However, owing to the significantly reduced hardware requirement of the proposed design, its overall hardware efficiency is almost double that of the fully parallel turbo decoder of [9, 129, 130]. While the hardware efficiency of the proposed design is half that of the state-of-the-art Log-BCJR decoder of [13], this work has achieved a significantly higher throughput and a much lower latency.

6.5 Results

This section characterizes the FPGA implementation of the proposed LTE turbo decoder and UEC-URC decoder of Figures 6.3 and 6.5, respectively. Table 6.2 shows the key performance criteria of the proposed architecture, when implemented using SystemVerilog, and targeting a midrange Altera Stratix IV EP4SE230 FPGA with 91k ALMs [138]. This table characterizes the proposed FPGA implementation of the LTE turbo decoder of Figure 6.3 using 28 decoding iterations, which was found in Section 6.3.3 to offer the same error correction capability as a Log-BCJR decoder performing 6 decoding iterations. Table 6.2 also characterizes the proposed FPGA implementation of the UEC-URC scheme of Figure 6.5 using 20 decoding iterations.

As shown in Table 6.2, the proposed LTE turbo decoder implementation achieves a maximum throughput of 306 Mbps with a latency of 1.44 μs. Table 6.2 also shows that the proposed UEC-URC decoder implementation achieves a maximum throughput of 450 Mbps at a latency of 1 μs, which readily fulfils the requirements of low latency, high throughput video applications. Table 6.2 characterizes the combinational and register

Table 6.2: FPGA implementation of the proposed LTE turbo decoder when performing 28 decoding iterations as well as of the proposed UEC-URC decoder when performing 20 iterations using the Altera Stratix IV EP4SE230 FPGA. Here, the combinational and register utilization quantifies the percentage of the 182,400 ALUTs and registers used, respectively.

	LTE decoder							UEC-URC decoder							
Frame length N	48	160	200	320	360	400	440	48	160	240	320	400	440	460	500
Interleaver type	B	A	B	A	B	B	A	A	A	A	A	A	A	A	A
Clock frequency (MHz)	195	189	182	179	170	171	156	202.5	194	177.6	173	165.7	161.9	156.5	140.9
Total Utilization (%)	11	35	43	70	78	84	98	10	34	52	65	81	88	89	100
Combinational Utilization (%)	9	30	37	60	68	75	83	9	29	44	58	72	79	82	89
Register Utilization (%)	5	17	22	35	39	44	48	6	18	27	37	45	50	52	56
Throughput (Mbps)	42	135	163	256	273	305	306	61	194	266	346	414	445	450	440
Latency (μs)	1.15	1.19	1.23	1.25	1.32	1.31	1.44	0.79	0.82	0.9	0.92	0.97	0.99	1.02	1.14

utilization, which quantify the percentage of the FPGA's combinational Adaptive Look-Up Tables (ALUTs) and registers used, respectively. Table 6.2 also characterizes the total hardware utilization of the proposed LTE and UEC-URC decoders, where the percentage of ALMs used by the designs are quantified. Note that each ALM comprises two ALUTs and two registers. However, due to routing constraints, the percentage of ALMs used is higher than the maximum of the combinational and register usage, since the FPGA tool cannot produce the most area efficient design without severely degrading the clock frequency. Note that the throughput drops slightly as the design reaches 100% hardware utilization, when longer frames are targeted. This may be explained by congestion within the FPGA, which also degrades the achievable clock frequency. Note that the Stratix IV EP4SE230 FPGA used for implementing the design is a mid-range device, whilst more powerful FPGAs containing up to 325k ALMs are also available in the Stratix IV series. For the LTE turbo decoder implementation, a maximum frame length of $N = 440$ is supported. In contrast, the FPGA fully parallel turbo decoder of [129] achieved a frame length of $N = 720$, using a FPGA with 3.5 times more resources available. An FPGA was used to confirm the SER performance results of Figure 6.15, which were obtained in simulation. Note that there is some discrepancy when comparing the combinational requirement X and register requirement Y, quantified in Table 4.2, to the actual implementation results of Table 6.2. More specifically, since the analysis of Table 4.2 only considers the combinational contribution of the addition and max operations in the datapath, it underestimates the overall combinational requirement, which also includes the multiplexers in the datapath, the multiplexers in the interleaver and the logic required by the controller. Table 4.2 predicts the register contribution more closely, since the only registers not considered in the analysis of Table 4.2 are those required by the controller. Note that these discrepancies exist for both the proposed architecture and the estimation of [129], however the analysis of Table 4.2 is useful for comparing the different algorithms before implementation.

Since this chapter presents the first high-speed FPGA implementation of a UEC-URC decoder, it cannot be compared with any previous work. However, we may compare the proposed FPGA implementation of the LTE turbo decoder with those of [129] and [139]. More specifically, Table 6.3 characterizes the proposed FPGA implementation of the LTE decoder and compares this work with the FPGA-based fully parallel turbo decoder of [129], as well as the FPGA-based implementation of the conventional Log-BCJR turbo decoder of [139]. Table 6.3 characterizes the performance of the proposed turbo decoder when performing $I = 20$ iterations, which is the number required to offer the same error correction capability as $I = 28$ iterations of the decoder of [129] and $I = 5$ iterations of the decoder of [139]. Here, the FPGA hardware utilization is compared using only the combinational utilization, which is quantified by the number of ALUTs used. While this disregards the register and RAM usage, it is the combinational usage that imposes the greatest resource requirement upon all three designs, therefore constituting the limiting factor of both the design size and the grade of parallelism.

Table 6.3: Comparison between the proposed FPGA implementation of the LTE turbo decoder, an FPGA implementation of the fully parallel turbo decoder of [129], as well as the state of the art FPGA implementation of the conventional Log-BCJR turbo decoder.

	This work (N=400)	Fully parallel of [129]	Log-BCJR of [139]
FPGA	EP4SE230 (182 kALUT)	EP4SE820 (650 kALUT)	EP4SE820 (650 kALUT)
Iterations I	20	28	5
Clock frequency (MHz)	171	65	102
Resource usage (kA-LUT)	137	644	254
Throughput after I iterations (Mbps)	425	835	524
Hardware efficiency (kALUT/Mbps)	0.32	0.77	0.48

Furthermore, [139] does not quantify the overall device utilization, preventing any other comparison. In order to compare the relative performance of the three considered FPGA implementations, Table 6.3 characterizes the throughput when the decoders perform a fixed number of iterations I, without early stopping. Note however that the fully parallel turbo decoder implementation of [129] is capable of using a CRC check to detect when the iterative decoding process has been successful and can be stopped early. Furthermore, the implementation of [139] is fully flexible and can support all the frame lengths specified by the LTE standard, while the proposed implementation and the work of [129] can only support a single frame length.

Of the three FPGA implementations compared, it is the proposed design that achieves the best hardware efficiency of 0.32 kALUT/Mbps, compared to 0.77 kALUT/Mbps for the fully parallel turbo decoder of [129] and 0.48 kALUT/Mbps for the Log-BCJR turbo decoder of [139]. Indeed, the turbo decoding algorithm and architecture proposed here achieves a 2.4-fold improvement in hardware efficiency over those of the fully parallel turbo decoder of [129], which is similar to the expected gain predicted in Section 6.4.4. Our algorithm and architecture achieves this gain by requiring fewer decoding iterations and by employing more efficient pipelining, which results in a much higher clock frequency. Furthermore, compared to the fully-parallel FPGA LTE turbo decoder implementation of [129], the register utilization of the proposed implementation is much closer to its combinational utilization, demonstrating better usage of the FPGA's resources.

Figure 6.20 shows a comparison of the performance characteristics of the proposed architecture and the architectures of the benchmarkers. This diagram considers the throughput achievable for a given frame length; the area efficiency; the flexibility of the architecture to support different interleaver designs; the maximum frame length supported given

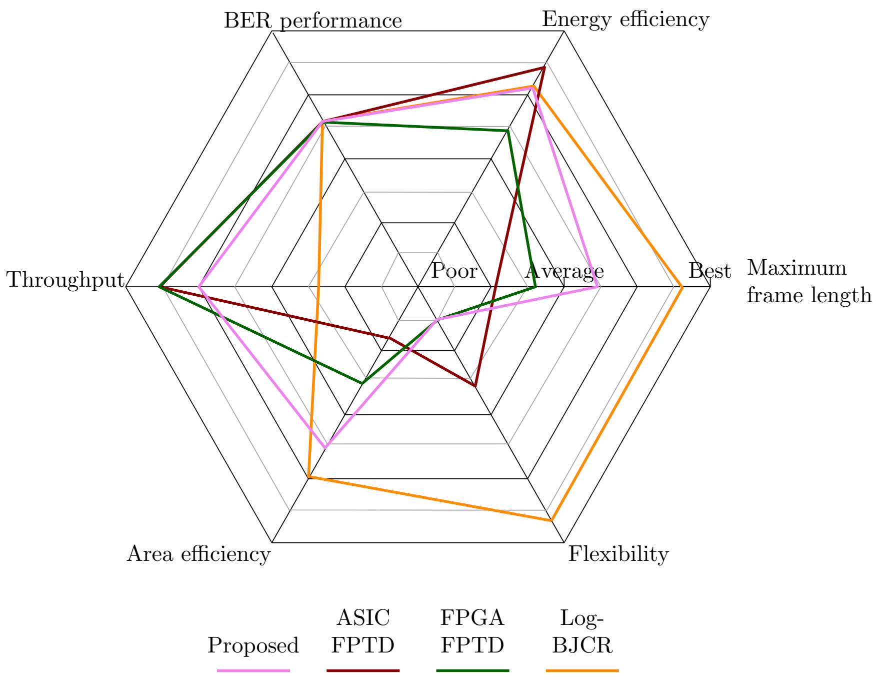


Figure 6.20: A comparison of the key performance characteristics of the proposed paired scheduling decoder, the ASIC [130] and FPGA [129] implementations of the FPTD, and the state-of-the-art BCJR decoder [139].

a limited amount of hardware resource; the energy efficiency; and the BER performance. Here, all schemes can achieve the same BER performance, albeit after performing a differing number of iterations, which impacts upon the other characteristics. This diagram shows that the ASIC FPTD architecture of [130] has traded-off reduced area efficiency and maximum frame length, in order to achieve better energy efficiency compared to the FPGA FPTD architecture of [129]. Likewise, the proposed paired scheduling architecture offers a slightly reduced throughput for a given frame length, but with the advantage of being able to support a larger maximum frame length, and greater area efficiency, compared to the FPTD architectures of [129] and [130].

6.6 Conclusions

This chapter has shown that in addition to its application in the LTE turbo decoder, the proposed highly parallel iterative decoder may be extended to the decoding of a UEC-URC code, where this UEC-URC code is found within the RiceEC and ExpGEC codes of Chapter 4. This work has shown that for both the LTE turbo code and UEC-URC code that the fully parallel scheduling can be modified to allow each processing block to operate two trellis stages. This enables improved pipelining and reduces the number of decoding iterations required for achieving strong error correction, leading to a 2.4-fold hardware efficiency improvement compared to the implementation of the original fully parallel decoding approach. In particular, this chapter has jointly considered the algorithm and its implementation. By reusing the same hardware to process both the forward state metrics $\tilde{\alpha}$ and $\tilde{\beta}$, the hardware resource requirement is significantly reduced and pipelining can be used without impacting upon the error correction performance. This novel pipelining technique also enables a better utilization of the FPGA's resources, by making a more equal use of register and combinational resources, compared to previous designs which under-used the available register hardware. This technique also allows significantly longer frame lengths to be supported within a given FPGA. Our implementation achieves a throughput of 306 Mbps and a latency of 1.44 μs for the LTE turbo decoder, as well as a throughput of 450 Mbps with 1.1 μs latency for the UEC-URC code, when targeting a midrange FPGA.

Chapter 7

Further development of a high throughput decoder

7.1 Introduction

The Fully Parallel Turbo Decoder (FPTD) algorithm of [9] is capable of achieving both a significantly increased throughput and a reduced latency compared to the state-of-the-art decoders, albeit at the cost of having an increased hardware resource requirement. Since each hardware processing element of a FPTD undertakes the decoding required for only a single bit, the available hardware resources imposes a limitation on the maximum frame length that can be supported by a FPTD implementation. Specifically, the solution of [129] was only capable of accommodating a FPTD supporting frame lengths of up to $N = 720$ bits on a large Field Programmable Gate Array (FPGA). However, for a decoder to be usable for the LTE standard [18], it would have to support frame lengths up to $N = 6144$ bits. This demanding hardware requirement was addressed by the paired scheduling decoder proposed in Chapter 6. This architecture benefits from a more intense hardware reuse, which allows the paired scheduling decoder to approximately double the frame length that can be accommodated on an FPGA, compared to the previous FPTD implementation of [129]. The FPTD also suffers from lower hardware efficiency, when decoding frames shorter than the maximum supported. More specifically, each hardware processing element of the FPTD carries out the decoding required for only a single bit. When decoding frames with length N shorter than the maximum supported, some of the hardware processing elements will become unused.

Against this background, this chapter extends the FPTD and paired scheduling decoder of Chapter 6 to conceive the Arbitrarily Parallel Turbo Decoder (APTD). For a given frame length N, the APTD attempts to achieve the highest throughput and lowest latency possible using as much of the given hardware as possible, therefore achieving a high hardware efficiency. More specifically, while the paired scheduling of Chapter 6 simultaneously carries out the decoding for a pair of trellis stages on each hardware processing unit, the APTD takes this approach a step further by processing an arbitrary number of trellis stages on each hardware processing element. This allows the APTD to vary the number of trellis stages decoded by each hardware processing element depending on the frame length N being decoded, hence allowing a wide range of frame lengths N to be decoded efficiently. This chapter implements the LTE scheme of Figure 6.3, in order to facilitate direct comparisons to the previous implementations of [129], as well as that of Chapter 6. The techniques described in this chapter may also be applied to implementations of other schemes, such as the UEC-URC scheme of Figure 6.5.

The APTD architecture proposed in this chapter follows the philosophy of Chapter 6, which managed to achieve a 2.4x hardware efficiency improvement over the conventional FPTD. More specifically, by increasing pipelining, an increased clock frequency, throughput and hardware efficiency can be achieved. In particular, the APTD uses the same level of pipelining and same design for the α, β, γ and b^{e} blocks as were used in Section 6.4, in order to maintain a high clock frequency. The APTD algorithm also benefits by decoding multiple trellis stages on a single hardware processing element, allowing a reduction in the number of algorithmic iterations required.

This chapter commences in Section 7.2 with a discussion of the proposed APTD algorithm, which describes the evolution of the APTD from the paired scheduling of Chapter 6. Following this, Section 7.3 discusses the hardware implementation of an APTD, including the scheduling of each part of the APTD and the operation of the interleaver. Section 7.4 quantifies the error correction performance of the APTD, where the implementation results show that the APTD achieves a 4.7x hardware efficiency improvement over the FPTD, when applied to the LTE turbo code. In these sections, the algorithmic and architectural aspects are considered jointly, in order to trade off the architectural requirements against the algorithm's performance. Finally, Section 7.5 offers the concluding remarks.

7.2 Algorithm modifications

This section details how the paired scheduling decoder of Chapter 6 can be generalised to the APTD. Section 7.2.1 commences with a discussion of the top level architecture of the APTD and its operation is contrasted to that of the FPTD of [9] and the paired scheduling of Chapter 6. Following this, Section 7.2.2 discusses the scheduling of the operations required by the APTD, demonstrating how the benefits offered by an odd-even activation order can be applied to the APTD.

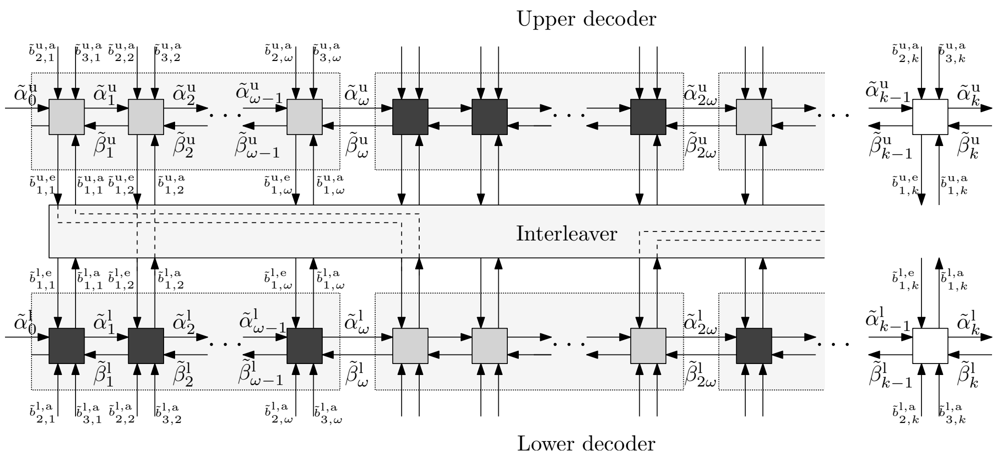

Upper decoder
Interleaver
Lower decoder
$\tilde{\alpha}_0^{\mathrm{u}}$
$\tilde{\alpha}_1^{\mathrm{u}}$
$\tilde{\alpha}_2^{\mathrm{u}}$
$\tilde{\alpha}_{\omega-1}^{\mathrm{u}}$
$\tilde{\alpha}_{\omega}^{\mathrm{u}}$
$\tilde{\alpha}_{2\omega}^{\mathrm{u}}$
$\tilde{\alpha}_{k-1}^{\mathrm{u}}$
$\tilde{\alpha}_{k}^{\mathrm{u}}$
$\tilde{\beta}_1^{\mathrm{u}}$
$\tilde{\beta}_2^{\mathrm{u}}$
$\tilde{\beta}_{\omega-1}^{\mathrm{u}}$
$\tilde{\beta}_{\omega}^{\mathrm{u}}$
$\tilde{\beta}_{2\omega}^{\mathrm{u}}$
$\tilde{\beta}_{k-1}^{\mathrm{u}}$
$\tilde{\beta}_{k}^{\mathrm{u}}$
$\tilde{\alpha}_0^{\mathrm{l}}$
$\tilde{\alpha}_1^{\mathrm{l}}$
$\tilde{\alpha}_2^{\mathrm{l}}$
$\tilde{\alpha}_{\omega-1}^{\mathrm{l}}$
$\tilde{\alpha}_{\omega}^{\mathrm{l}}$
$\tilde{\alpha}_{2\omega}^{\mathrm{l}}$
$\tilde{\alpha}_{k-1}^{\mathrm{l}}$
$\tilde{\alpha}_{k}^{\mathrm{l}}$
$\tilde{\beta}_1^{\mathrm{l}}$
$\tilde{\beta}_2^{\mathrm{l}}$
$\tilde{\beta}_{\omega-1}^{\mathrm{l}}$
$\tilde{\beta}_{\omega}^{\mathrm{l}}$
$\tilde{\beta}_{2\omega}^{\mathrm{l}}$
$\tilde{\beta}_{k-1}^{\mathrm{l}}$
$\tilde{\beta}_{k}^{\mathrm{l}}$

Figure 7.1: The top algorithmic level of the APTD, when used for decoding the LTE turbo code.

7.2.1 APTD top level

Figure 7.1 shows the top level algorithmic view of the APTD. This figure highlights the difference between the FPTD of Figure 6.7a and the paired scheduling decoder of Figure 6.7b. More specifically, Figure 7.1 shows that in the APTD, each of the $2N$ decoder blocks are placed into windows of size ω. The window size ω can be appropriately selected based on the frame length N and how much hardware resources are available.

The shading of each block indicates which half-iteration it is activated in, where the lightly-shaded blocks are activated within the first half of each iteration and the darkly-shaded blocks are activated in the second half. As shown in Figure 7.1, each block in each group has the same shading, indicating that the blocks in each group are processed simultaneously within the same half-iteration. In this way, the APTD of Figure 7.1 still maintains an odd-even activation order [9]. While each window of the same shading is operated in parallel, the decoding blocks within each window are processed in a serial manner using one hardware processing element. Note that the paired scheduling shown in Figure 6.7b may be viewed as a special case of the APTD, where $\omega = 2$. Likewise, the FPTD is a special case where $\omega = 1$. Section 7.2.2 will justify why the odd-even activation order of the algorithmic blocks provides superior performance.

7.2.2 APTD scheduling

Figure 7.2 illustrates the operation of each of the $\tilde{\boldsymbol{\alpha}}$, $\tilde{\boldsymbol{\beta}}$ and $\tilde{b}^{\mathrm{e}}$ calculations for the APTD, where processing for the first two windows of decoding blocks are shown over the period of 1.5 algorithmic iterations. In the APTD, the $\tilde{\boldsymbol{\alpha}}$, $\tilde{\boldsymbol{\beta}}$ and $\tilde{\boldsymbol{b}}^{\mathrm{e}}$ calculations of Equations (6.2) – (6.5), are carried out according to the schedule of Figure 7.2. More specifically, during the processing of each window of Figure 7.1, the $\tilde{\boldsymbol{\alpha}}$ values are processed in a

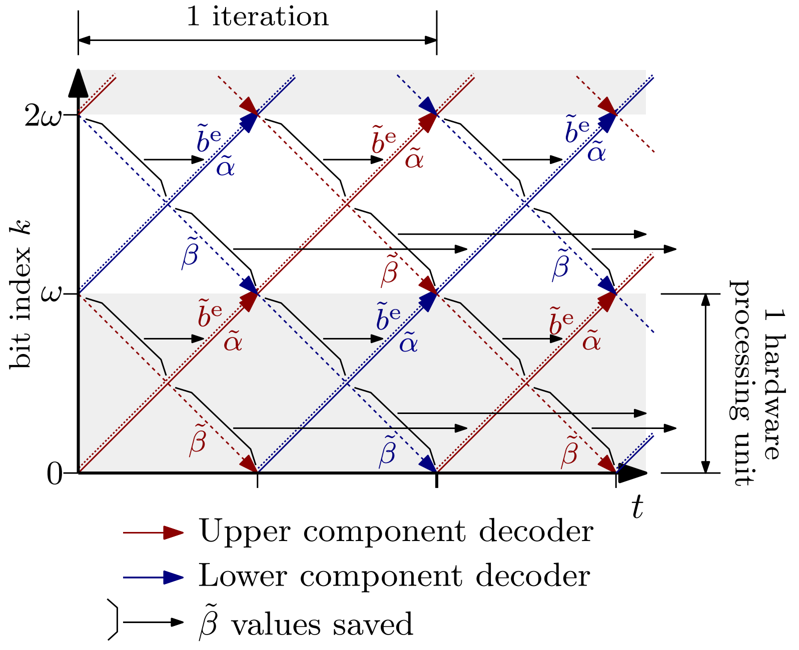


Figure 7.2: The scheduling for the $\tilde{\alpha}$, $\tilde{\beta}$ and $\tilde{b}^{\mathrm{e}}$ calculations for the APTD. The amount of processing undertaken by one hardware processing element is indicated.

forwards recursion, since (6.2) shows that each value $\tilde{\alpha}_k$ relies on the value of $\tilde{\alpha}_{k-1}$ for the previous trellis stage, while the $\tilde{\boldsymbol{\beta}}$ values are processed in a backwards recursion, since (6.3) shows that each value $\tilde{\beta}_k$ relies on the value of $\tilde{\beta}_{k+1}$ for the next trellis stage. Each extrinsic output $\tilde{b}^{\mathrm{e}}$ having index k is generated at the same time as the $\tilde{\alpha}$ metric having index k. While the $\tilde{b}^{\mathrm{e}}_k$ calculation of Equation (6.5) requires both $\tilde{\alpha}_{k-1}$ and $\tilde{\beta}_k$ values, Figure 7.2 shows that the $\tilde{\beta}_k$ values are not calculated at the same time as the $\tilde{\alpha}_{k-1}$, which is owing to the constraints imposed on the $\tilde{\boldsymbol{\alpha}}$ and $\tilde{\boldsymbol{\beta}}$ calculations. As a result, Figure 7.2 shows how the $\tilde{\boldsymbol{\beta}}$ values that are calculated in the first part of each half iteration are saved for use during the second part of each half iteration. Likewise, the $\tilde{\boldsymbol{\beta}}$ values calculated during the second part of each half iteration are saved for use during the next iteration. This delayed exploitation of the $\tilde{\boldsymbol{\beta}}$ values was also used by the paired scheduling decoder, as described in Section 6.3. This figure shows how the odd-even activation order of groups of blocks improves the performance by allowing the state metrics to be passed along the decoder more promptly. More specifically, after an algorithmic iteration, the state metrics propagate further by a distance of 2ω trellis stages for an odd-even decoder. By contrast, for a 'top-bottom' decoder, which activates all of the upper groups followed by all of the lower groups, after an algorithmic iteration the state metrics will have propagated further a distance of only ω trellis stages. Note that in the case of the paired FPTD, where $\omega = 2$, the scheduling of Figure 7.2 is identical to that of Figure 6.9.

7.3 Hardware implementation

This section details the potential hardware implementation for the APTD described in Section 7.2. Firstly, Section 7.3.1 commences by describing the scheduling of each operation undertaken by each hardware processing element, including the pipelining and data dependencies. Following this, Section 7.3.2 describes the proposed APTD architecture, which implements the scheduling of Section 7.3.1. Finally, Section 7.3.3 proposes a method for reducing the number of interleaver connections, as well as how a reconfigurable interleaver may be constructed.

7.3.1 Hardware scheduling

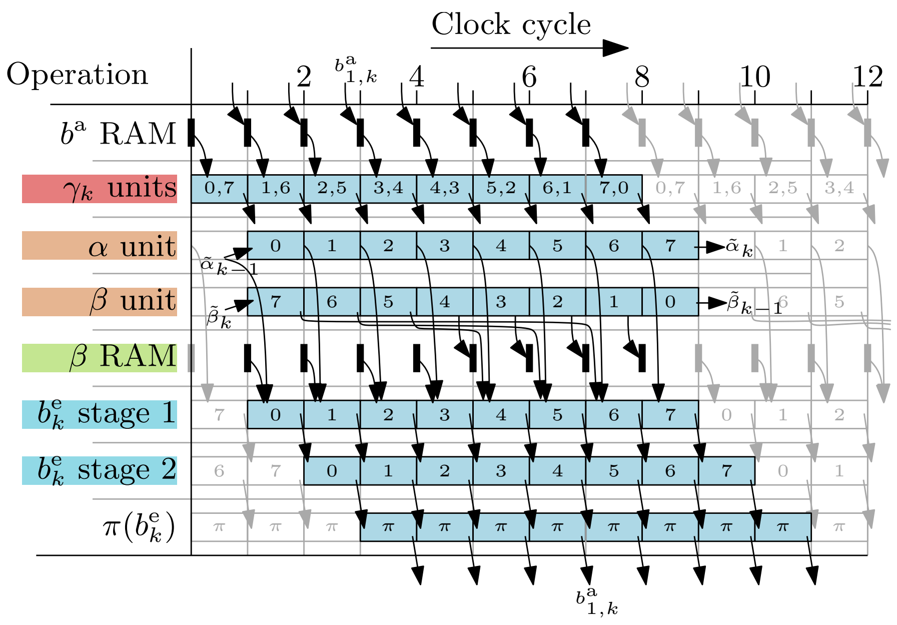


Figure 7.3: Timing diagram for one hardware processing element for one half-iteration of the algorithm, for the example of windows comprising $\omega = 8$ trellis stages. The diagram shows when each of the different tasks are undertaken, as well as the transfer of data between tasks to show to dependency between them. The number written in each box specifies which of the $\omega = 8$ trellis stages that box is processing.

This section details the beneficial scheduling of a hardware architecture, which implements the APTD algorithmic schedule of Figure 7.2. The hardware architecture is comprised of multiple hardware processing elements, which each alternately undertake the decoding of one window of decoding blocks in one half-iteration, followed by decoding a different window of decoding blocks in the following half-iteration. More specifically, each “row” of Figure 7.2 represents the decoding undertaken by one hardware processing unit. Each hardware processing element of the proposed hardware architecture is comprised of hardware dedicated to each of the $\tilde{\alpha}$, $\tilde{\beta}$, $\tilde{\gamma}$ and $\tilde{b}^{\mathrm{e}}$ calculations. More specifically, each hardware processing element is comprised of two $\tilde{\alpha}/\tilde{\beta}$ units of Section 6.4.3.2, which complete their operation within one clock cycle; one $\tilde{b}^{\mathrm{e}}$ unit of Section 6.4.3.3, which comprises two pipeline stages that are operated across two consecutive clock cycles, and two $\tilde{\gamma}$ units of Section 6.4.3.1, which require one clock cycle. Here, two $\tilde{\gamma}$ units are required since the $\tilde{\alpha}$ and $\tilde{\beta}$ values are calculated at the same time. As a result, the architecture proposed in this section has less hardware reuse within each hardware processing element than the architecture of Chapter 6, which employs only a single $\tilde{\alpha}/\tilde{\beta}$ unit in each hardware processing element.

Figure 7.3 shows the timing proposed for the APTD architecture. The scheduling advocated in this section attempts to make the best possible use of the hardware. As shown in Figure 7.3, each of the parts of each hardware processing element undertakes a unique operation in each clock cycle, leading to a high hardware efficiency. Figure 7.3 illustrates the operation of one window of $\omega = 8$ trellis stages by one hardware processing element, during one half-iteration. For each of the ω trellis stages in the window, Figure 7.3 shows that each hardware processing element processes the $\tilde{\alpha}$ and $\tilde{\beta}$ values in a sequential manner, with the $\tilde{\alpha}$ values calculated in a forwards recursion and the $\tilde{\beta}$ values calculated in a backwards recursion, as described in Section 7.2.2. As shown in Figure 7.3, in each clock cycle, the first pipeline stage of the $\tilde{b}^{\mathrm{e}}$ unit processes the extrinsic output for the same trellis stage having the same index k as the $\tilde{\alpha}$ unit, mirroring the method used in the paired scheduling decoder, shown in Figure 6.9. In analogy with the $\Delta\tilde{\beta}$ memory of Figure 6.9 that is required by the paired scheduling decoder, some RAM is required by the APTD architecture to temporarily store $\tilde{\beta}$ values, since most $\tilde{\beta}$ values are not used by the $\tilde{b}^{\mathrm{e}}$ unit immediately. More specifically, in the example of $\omega = 8$, the β values 5, 6 and 7 can be used in the same iteration in which they are calculated. However, the β values 1, 2, 3 and 4 must be stored in the $\tilde{\beta}$ RAM until the next iteration. Meanwhile the output of $\tilde{\beta}$ block 0 is passed to the neighbouring processing element. Note that the $\tilde{\boldsymbol{\alpha}}$ and $\tilde{\boldsymbol{\beta}}$ values forwarded to the neighbouring processing element can be used immediately by the hardware processing elements, owing to the odd-even activation order. As a result, these $\tilde{\boldsymbol{\alpha}}$ and $\tilde{\boldsymbol{\beta}}$ values do not require storing for the next half-iteration, as demonstrated by Figure 7.2.

After the $\tilde{b}^{\mathrm{e}}$ unit has output an LLR, it is passed through the interleaver. Since the LLR may not be used immediately, a $\tilde{b}^{\mathrm{a}}$ RAM is required for storing the $\tilde{b}^{\mathrm{a}}$ values input to each hardware processing element from the interleaver until they are required. This RAM also stores the LLR vectors $\tilde{\mathbf{b}}_2^{\mathrm{a}}$ and $\tilde{\mathbf{b}}_3^{\mathrm{a}}$ from the demodulator for the duration of the decoding process.

Due to the pipelined nature of the proposed APTD, each hardware processing element slightly overlaps the decoding of successive windows, as shown in Figure 7.3. As a result, the extrinsic LLR $\tilde{b}^{\mathrm{e}}$ output from one hardware processing element may not be interleaved to the input of the next hardware processing element before it is required, hence that particular hardware processing element will use older $\tilde{b}^{\mathrm{a}}$ values from the $\tilde{b}^{\mathrm{a}}$ RAM. The number of older values of $\tilde{b}^{\mathrm{a}}$ that will be used instead of the newest one depends on the particular design and implementation of the interleaver, as well as upon the parameters N and ω. Note that the scheme of Chapter 6, which is similar to the APTD with $\omega = 2$, exploited the *mod4* property of the LTE interleaver for ensuring that the $\tilde{b}^{\mathrm{a}}$ values arrived at the input of each processing hardware element as and when they were needed. However, this is not possible for the APTD, since it requires fewer clock cycles for decoding each trellis stage and because it may use any arbitrary value of ω. The performance impact of the interleaver and pipelining delay will be explored in Section 7.4.

7.3.2 Proposed APTD hardware architecture

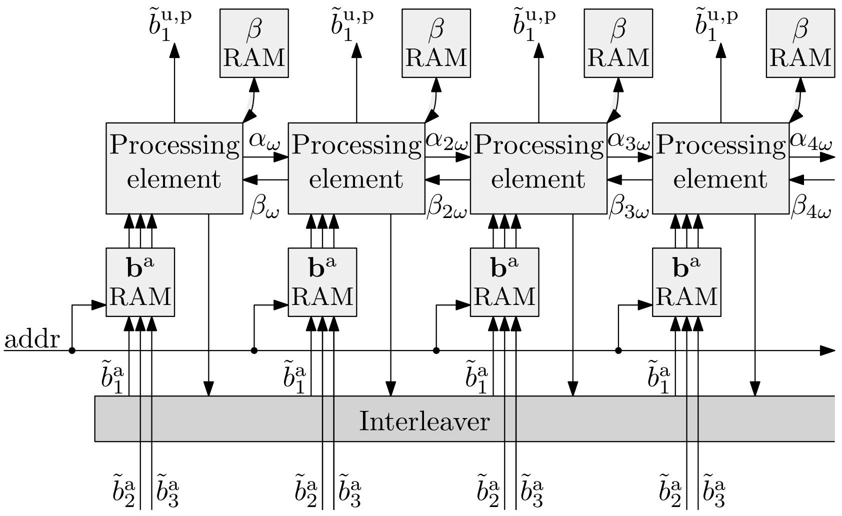


Figure 7.4: Hardware architecture showing the first four hardware processing elements of the APTD.

Figure 7.4 shows the hardware architecture proposed for implementing the scheduling of Section 7.3.1. As shown in Figures 7.2 and 7.3, each hardware processing element has to pass $\tilde{\alpha}$ and $\tilde{\beta}$ values between its neighbours, and each hardware processing element outputs extrinsic LLRs directly to the interleaver. These extrinsic LLRs $\tilde{b}^{\mathrm{e}}_{1,k}$ are interleaved into *a priori* LLRs $\tilde{b}^{\mathrm{a}}_{1,k}$, and then loaded into a small $\tilde{\mathbf{b}}^{\mathrm{a}}_1$ RAM associated with each hardware processing element, so that they can be used when required. Each hardware processing unit also exchanges $\tilde{\beta}$ values with a small $\tilde{\beta}$ RAM, as required by the scheduling of Figure 7.3. Since this design requires small blocks of RAM for each of the hardware processing elements, it lends itself to convenient FPGA implementation, since FPGAs have many blocks of RAM distributed across the device. Furthermore, since the FPGA device utilization is dominated by the resource which is used the most, the extra RAM resources required by the APTD will not significantly increase the FPGA utilization, since these RAM resources would have otherwise remained unused. Each of the hardware processing elements is connected to an interleaver, which allows extrinsic outputs $\tilde{b}^{\mathrm{e}}_{1,k}$ from one hardware processing element to be passed to another in the form of *a priori* inputs $\tilde{b}^{\mathrm{a}}_{1,k}$, as required by the interleaver design.

7.3.3 Interleaver requirements

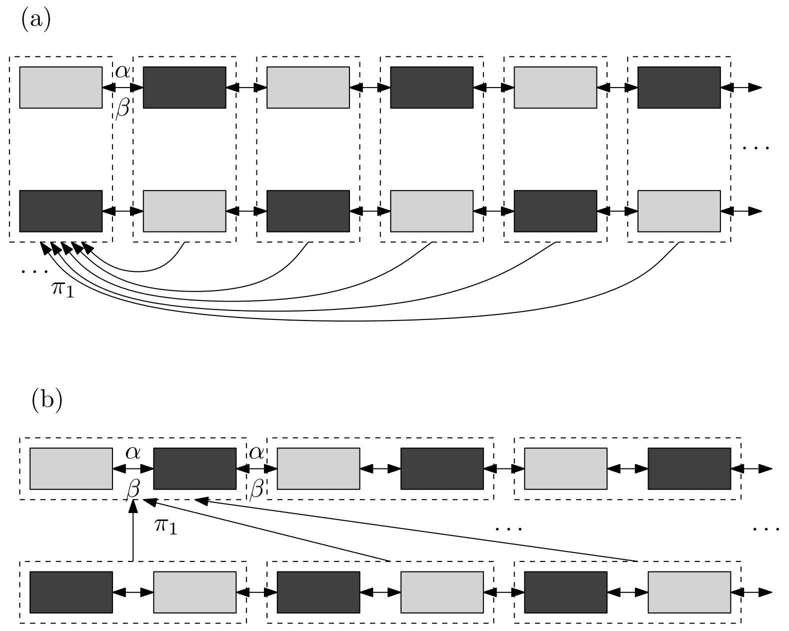


Figure 7.5: The required interleaver connections per hardware processing element can be reduced by half. Each hardware processing element is shown by the dashed lines. The shaded blocks may each represent a window of multiple trellis stages, as proposed here, or single trellis stages as proposed by [9]

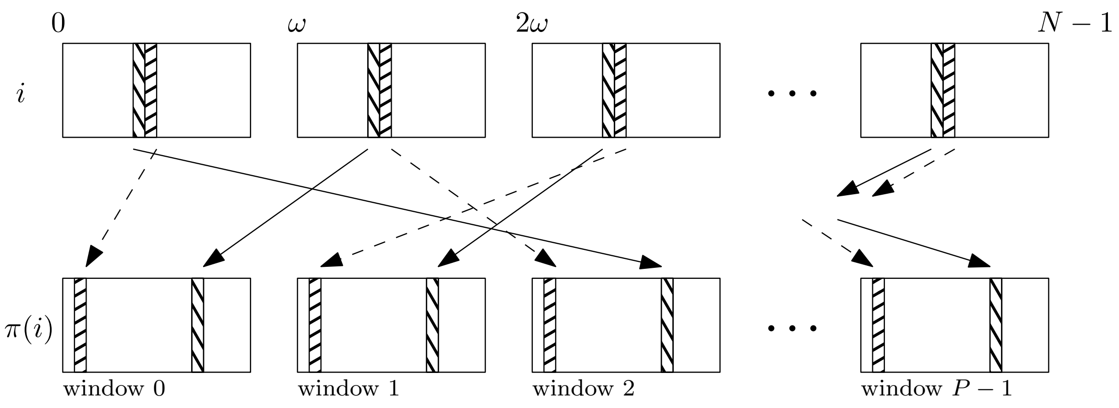


Figure 7.6: Contention-free interleaver using the same indices within each window, where address i is interleaved to yield the address $\pi(i)$. The interleaving pattern is shown for two sets of addresses.

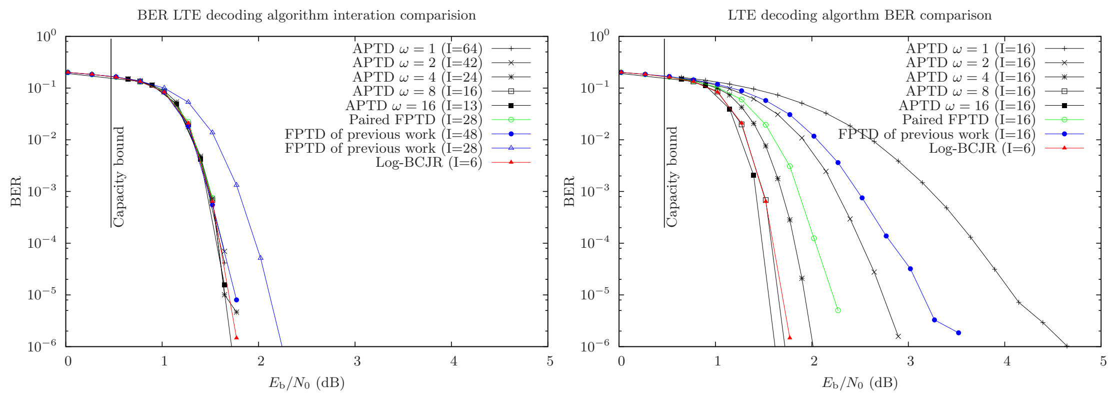
While the implementation results presented in Chapter 6 and the previous implementations of the FPTD [129,130] have all used a single, specific hard-wired interleaver design, this section explores the design requirements of an interleaver conceived for the APTD architecture, which supports the full set of 188 different designs that are specified in the LTE standard. The approach taken by Chapter 6 and by the previous fully parallel implementations of [9,130] has arranged for each hardware processing element to alternate between the decoding windows within both the upper decoder and the lower decoder. The interleaver requirement for this arrangement is shown in Figure 7.5(a). In order for the interleaver to achieve any arbitrary connection between the upper and lower decoders, each of the P hardware processing elements would require $(P-1)$ connections to each of the other hardware processing elements. Figure 7.5(b) shows an alternative option, where each hardware processing element only processes a pair of neighbouring windows within either the upper or lower decoder, without changing the algorithm. Instead, it imposes different requirements on the connections between hardware processing elements. Since the upper and lower decoders do not interleave to themselves, the arrangement of Figure 7.5(b) only requires each of the P hardware processing elements to have $P/2$ connections to other processing elements, instead of $(P-1)$ connections per processing element in the case of Figure 7.5(a). This halves the number of interleaver connections, hence reducing the hardware resource requirement of the interleaver.

An interleaver may be said to be contention-free if all LLRs generated at the same time by the source processing elements are interleaved to different destination processing elements, therefore eliminating the requirement for extra hardware to route multiple LLRs to the same destination. This property is demonstrated graphically by Figure 2.12, and reproduced in Figure 7.6 for convenience. Note that all LTE interleaver designs are contention-free, if ω is an integer factor of the frame length N, provided that each hardware processing element processes the same trellis stage associated with the same index within each window at the same time. Note that conventional Log-BCJR decoders [17] typically adopt a maximum parallelism of $P = 64$, since this is the highest common multiple for all the LTE interleavers having lengths $N \geq 2048$. However, since the APTD may be used for supporting high throughputs with a high degree of parallelism over the full range of LTE frame lengths N, the issue of contention must be considered during the hardware design of the interleaver. More specifically, in order to benefit from the contention-free interleaver property, ω must be an integer factor of the frame length N, which will result in an under-utilisation of the P hardware processing elements for some values of N. Alternatively, best use of all the available P hardware processing elements for all interleaver designs could be achieved by designing an interleaver that can readily handle contention. For example, rather than ensuring that all extrinsic LLRs are interleaved as required by the algorithm, a hardware implementation may accept some performance loss from an interleaver design that may not interleave all extrinsic LLRs for particular values of N and ω. Another option would be having an interleaver that ensures that all extrinsic LLRs are interleaved to their destination, but imposes a large area by allowing more than one LLR to be interleaved to the same destination at the same time. Alternatively, a long delay could be imposed by rescheduling the interleaving operations. However, this latter option will lead to both idle hardware, as well as a reduced throughput and to a lower hardware efficiency. Alternatively, the hardware processing elements may use old *a priori* LLRs, if newer extrinsic LLRs are still traversing the interleaver, at the cost of a reduced error correction performance, or an increased number of iterations required. Motivated by this, Section 7.4.1 will consider different interleaver delays, in order to investigate the associated performance degradation.

7.4 Performance characteristics

This section commences in Section 7.4.1 by investigating the error correction performance of the APTD, when decoding the LTE turbo code. Following this, Section 7.4.2 discusses and compares the potential resource usage, throughput and latency to those of the FPTD of [129] and t o the paired scheduling of Chapter 6.

7.4.1 BER performance



(a) (b)

Figure 7.7: BER results for the LTE turbo decoder, when implemented using the proposed APTD architecture, the FPTD decoder of [9, 130], the paired scheduling architecture of Chapter 6, and the ideal Log-BCJR decoder. Here, the schemes use a frame length of $N = 4864$, and employ QPSK modulation for communication over an uncorrelated Rayleigh fading channel. The APTD scheduling using window sizes of $\omega \in \{1, 2, 4, 8, 16\}$ is compared. The top graph shows how many iterations of each scheme are required to match the BER performance of the Log-BCJR decoder after 6 iterations, while the lower graph shows the APTD and FPTD schemes' performance after $I = 16$ iterations.

Figure 7.7 shows the Bit Error Ratio (BER) performance of the proposed APTD architecture of Figure 7.3, compared to the classic Log-BCJR decoder, to the paired scheduling decoder of Chapter 6, and to the odd-even FPTD of [9]. The performance of the

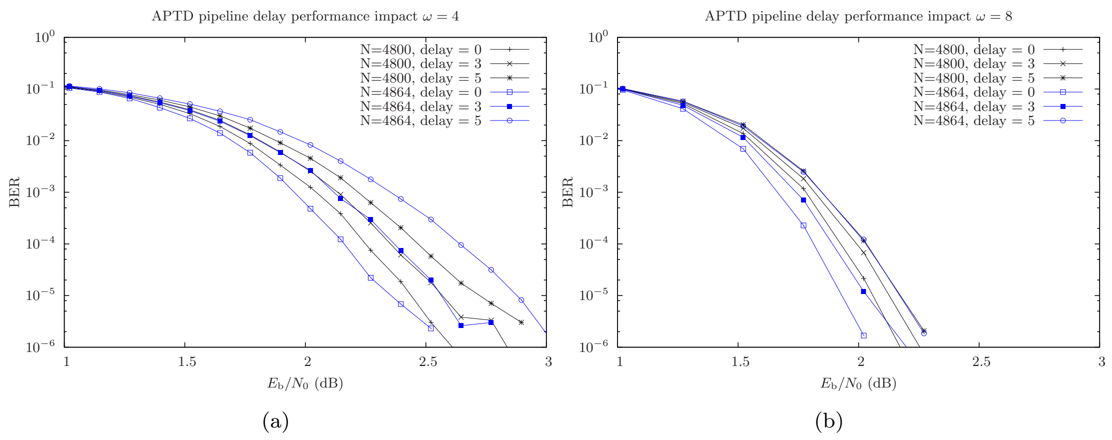


Figure 7.8: BER results for the proposed APTD scheduling for different pipeline delays, when employing QPSK modulation for communication over an uncorrelated Rayleigh fading channel. A pipeline delay of 3 corresponds to the scheduling proposed in Figure 7.3. This is compared to a pipeline delay of 0 where there is no pipelining and where the extrinsic LLRs arrive without delay, as well as a pipeline delay of 5 where the interleaver takes three clock cycles to complete rather than the one clock cycle afforded by Figure 7.3.

proposed APTD scheduling includes the pipeline delays shown in Figure 7.3, when employing a window-length of $\omega \in \{1, 2, 4, 8, 16\}$ trellis stages per hardware processing element. Here, QPSK modulation is used for transmission over an uncorrelated narrowband Rayleigh fading channel. In Figure 7.7, the number of iterations performed by each scheme has been chosen to give a similar BER performance as 6 iterations of the Log-BCJR algorithm. Figure 7.7(a) shows that in the case of $\omega = 1, 2, 4, 8, 16$, the number of iterations required to match the Log-BCJR performance is 64, 42, 28, 24, 16 and 13, respectively. Meanwhile, Figure 7.7(b) shows the BER performance of the Log-BCJR after 6 iterations and the remaining schemes after 16 iterations.

Note that even with a modest number of bits per hardware processing element, the number of iterations required by the APTD reduces substantially compared to the FPTD. Note that the APTD with $\omega = 1$ has the same activation order as the FPTD algorithm of [9] and as the ASIC FPTD architecture of [130]. However the proposed architecture features greater pipelining for the sake of achieving a higher clock frequency, which is not used by the FPTD architecture of [130]. As a result, the extra delay imposed by the pipelining in the proposed APTD architecture with $\omega = 1$ results in the APTD requiring 16 more iterations than the FPTD of [9, 130].

Likewise, the APTD using $\omega = 2$ is equivalent to the paired scheduling of Chapter 6. However, due to the differing schedules of Figure 7.3 and Figure 6.17, the paired scheduling outperforms the APTD using $\omega = 2$ by 0.5 dB at 10^{-5} after 16 iterations. This is because the architecture of the paired FPTD was designed jointly with the algorithm to ensure that the hardware accurately implemented the algorithm. By contrast, due to the range of frame lengths N and window lengths ω supported, the APTD cannot guarantee that all extrinsic LLRs are interleaved before they are required. These examples using $\omega = 1$ and 2 demonstrate that the APTD is better suited for longer frame lengths N, which result in higher values of ω.

Figure 7.8 shows the effect of the interleaver on the BER performance. Two interleavers are considered, the $N = 4864$ interleaver is a *mod type A* interleaver of Figure 6.11, while the $N = 4800$ interleaver is a *mod 4 type B* interleaver of Figure 6.11. Figure 7.8 explores the impact of pipelining on the system. The proposed scheduling shown in Figure 7.3 corresponds to a pipeline delay of three clock cycles, which is compared to the case where there is no pipeline delay. The results of Section 7.4.2 assume a fixed interleaver with a pipeline delay of one clock cycle. However Figure 7.8 also considers the case where the interleaver requires three additional clock cycles, giving a total delay of five clock cycles. This shows the performance degradation expected, should a flexible interleaver be used, which may impose further delay. This graph shows that there is some flexibility in designing the interleaver, and that adding pipeline stages to aid the hardware architecture will not impact significantly upon the number of iterations required for achieving a particular BER. The non-pipelined decoder achieves a marginal BER performance advantage over the implemented decoder. However an architecture with no pipelining would have a much longer critical path, leading to a considerably lower clock frequency, a reduced throughput and a reduced hardware efficiency.

7.4.2 Implementation results

This section considers the implementation of the APTD architecture of Figure 7.3, when applied to the LTE turbo code. This section commences by comparing the APTD architecture to the FPTD architecture of [129,130], and to the paired scheduling decoder of Chapter 6. Following this, the FPGA implementation results are discussed, including a discussion of the relative throughput and latency of the schemes.

Table 7.1 shows the comparison of the proposed APTD architecture to the paired scheduling decoder of Chapter 6, to the state-of-the-art Log-BCJR decoder [13] and to the FPGA and ASIC FPTD of [129, 130]. This table characterizes the proposed APTD architecture, when having different window lengths ω. Table 7.1 shows that when the APTD uses higher values of ω and fewer hardware processing elements P, fewer iterations are required and therefore the hardware efficiency (resource/throughput) improves. However, this results in a throughput reduction owing to the reduced parallelism. When compared to the FPTD of [130], the proposed APTD architecture offers an up to 5.7-fold improvement in hardware efficiency with $\omega = 16$, due to the reduced clock cycle duration from the pipelining and the reduced number of required iterations. Note that the analysis of the FPTD of [130] does not account for the increased clock cycle duration D due to the pipelining delay, which is significant even for a fixed

Table 7.1: Comparison between the state-of-the-art BCJR decoder [13] and the fully parallel decoder of [9, 130] when used to decode the $N = 6144$-bit LTE turbo code.

	Estimation in [9, 130]		Chapter 6	Proposed APTD		
	State-of-the-art LTE decoder of [13]	FPTD of [9, 129, 130]	Paired scheduling decoder of Section 6.3	$\omega = 2$	$\omega = 8$	$\omega = 16$
Number of parallel hardware processing elements	64	N*	N/2	N/ω		
Clock cycles per iteration T	$N/32$	2	8	$2 \cdot \omega$		
Clock cycle duration D	3 stages	6 stages + interleaver	3 stages	3 stages		
Complexity per decoding iteration C	320N	155N	$155N$	158N		
Decoding iterations I	6	39	28	42	16	13
Combinational requirement X	14144	$80N$	$30N$	$83N/\omega$		
Register requirement Y	1792	$21N$	$26N$	$27N/\omega$		
RAM requirement Z	$14N/3$ + 8320	0	0	$21N$		
Overall throughput $N/(T \cdot D \cdot I)$	16/9 (1x)	$N/468$ (7.38x)	$N/672$ (5.14x)	$N/504$ (6.86x)	$N/768$ (4.50x)	$N/1248$ (2.77x)
Overall latency ($T \cdot D \cdot I$)	$9N/16$ (7.38x)	468 (1x)	672 (1.44x)	504 (1.08x)	768 (1.64x)	1248 (2.67x)
Overall complexity ($C \cdot I$)	$1920N$ (1x)	$6045N$ (3.15x)	$4340N$ (2.26x)	$6636N$ (3.45x)	$2528N$ (1.32x)	$2054N$ (1.07x)
Overall resource $w \max(\frac{X}{2}, \frac{Y}{2}, \frac{Z}{160})$	$7072w$ (1x)	$40Nw$ (34.8x)	$15Nw$ (13.0x)	$20.8Nw$ (18.1x)	$5.2Nw$ (4.51x)	$2.6Nw$ (2.25x)
Hardware efficiency (resource/throughput)	2.51×10^{-4} (1x)	5.34×10^{-5} (0.21x)	9.92×10^{-5} (0.40x)	9.56×10^{-5} (0.38x)	2.50×10^{-4} (1.00x)	3.08×10^{-4} (1.22x)

Table 7.2: Comparison of FPTD and APTD FPGA implementations for the LTE turbo code.

	FPTD architecture of [129]	Paired scheduling decoder of Section 6.3	Proposed APTD architecture
ALUT per hardware processing element	793	680	892
Registers per hardware processing element	90	334	202
M9K RAM per hardware processing element	0	0	3
Estimated clock speed	65 MHz	171 MHz	150 MHz

interleaver, when employing an FPGA. Table 7.1 shows that when the proposed APTD uses a window size of $\omega = 2$ and therefore the same parallelism P as the paired scheduling of Chapter 6, the paired scheduling has a marginally better hardware efficiency. For a given frame length N, the paired scheduling also requires less hardware than the APTD architecture, which is an explicit benefit of the paired scheduling's greater hardware reuse and its reliance on fewer iterations. The proposed APTD scheduling maintains a high throughput and low latency throughout the range of ω values considered, achieving a 4.5-fold throughput improvement compared to the state-of-the-art Log-BCJR [13], and a 4.5-fold latency improvement, without increasing the hardware requirement. Furthermore, when using $\omega > 8$, the proposed APTD exceeds the hardware efficiency of the state-of-the-art Log-BCJR.

Table 7.2 shows the FPGA resource requirement for a single hardware processing element of the APTD, for the paired scheduling decoder and for the FPTD, when implementing an LTE decoder. The clock speed shown in Table 7.2 is representative of an FPGA associated with high utilization, since the placement and routing constraints imposed on a highly utilized FPGA leads to a reduced clock speed, as described in Section 6.5. Note that the achievable clock speed of the proposed APTD is based on the speed achieved by the paired fully parallel architecture, since a compete decoder, including the controller, has not yet been constructed.

The values of Table 7.2 are used by Figure 7.9 to show the throughput of the three schemes discussed as a function of the frame length N. In each case, the decoders have a fixed overall resource usage of 435 kALUT, as offered by the EP4SGX530. This yields 500 processing elements for the fully parallel solution of [130], 625 processing elements for the paired scheduling of Chapter 6, and 476 processing elements of the APTD proposed here. In Figure 7.9, the throughput increases with the frame length, since this allows a proportionately higher number of active hardware processing elements to be used. Note that this figure does not convey the hardware or energy efficiency of each of the

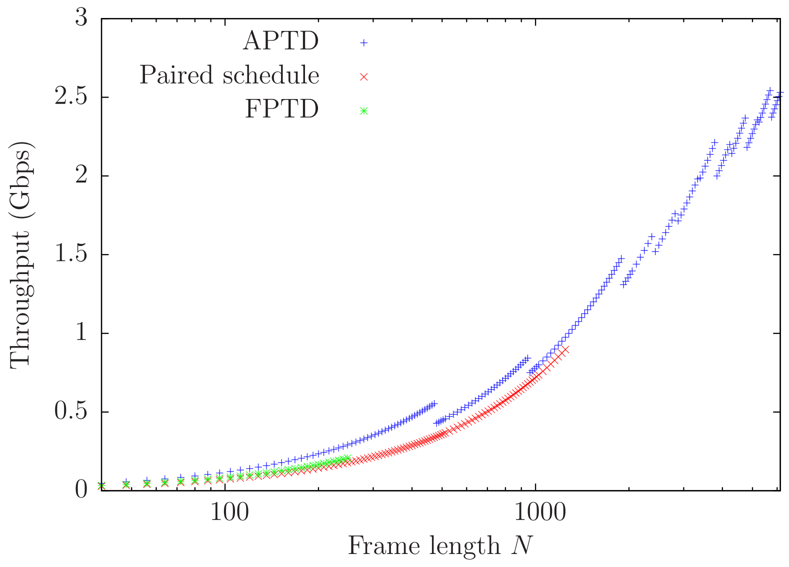


Figure 7.9: Estimated throughputs for the three turbo decoder schemes discussed, as a function of frame length N where each scheme has the same fixed overall resource usage as offered by the EP4SGX530 FPGA.

schemes. Furthermore, each of these schemes uses a fixed hard-wired interleaver, that supports only a single frame length N at run-time. The APTD may be designed in conjunction with a variable window length ω depending on the frame length N, so that as many hardware processing elements as possible are utilised. In the case where the frame length N is not a multiple of the number of processing elements P, some of the hardware processing elements will become unused. This is indicated by the zig-zag nature of the throughput graph for the proposed APTD. Figure 7.9 shows that for a given frame length N, the FPTD of [129] has a marginally lower throughput than the proposed APTD, since the FPTD has a longer critical path and low clock frequency, despite requiring fewer iterations. The FPTD also has the lowest limit on the maximum frame length it can decode, due to its poor hardware efficiency. Note that the maximum throughput of the paired scheduling of Chapter 6 is higher than that of the FPTD of [129] since it has higher hardware efficiency, as discussed in Section 6.4.4. The APTD proposed in this section can be designed with a varying window size ω, in order to achieve the best hardware efficiency, and therefore the highest maximum throughput.

Figure 7.10 shows the latency for the turbo decoders considered. The latency rises for the proposed APTD as the frame length N increases, which is owing to the reduced parallelism P as the window length ω increases. The FPTD of [130] has a low latency, since it has the highest parallelism, but this scheme still suffers from having a long critical path, which increases its latency. As a result, the proposed APTD has the lowest latency when the frame length is short and $\omega = 1$, despite requiring more iterations than the solution of [130].

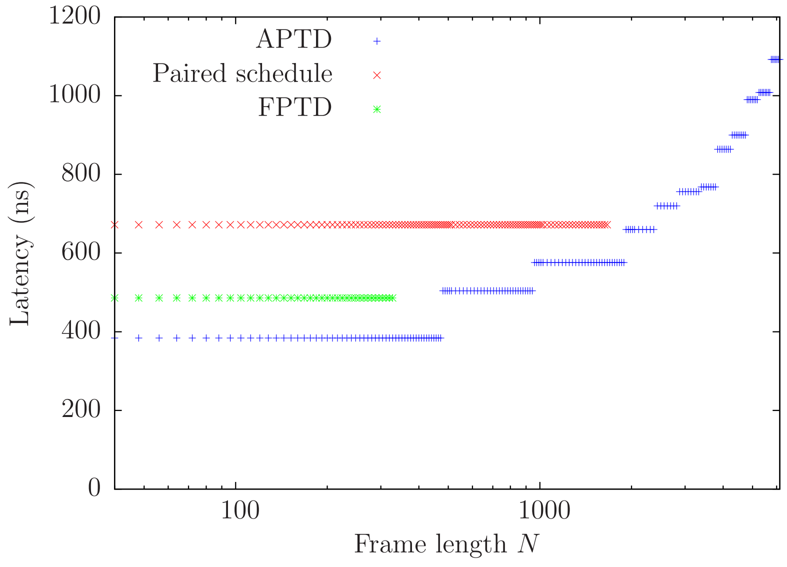


Figure 7.10: Estimated latency for the three turbo decoder schemes discussed, as a function of frame length where each scheme has the same fixed overall resource usage, as offered by the EP4SGX530 FPGA.

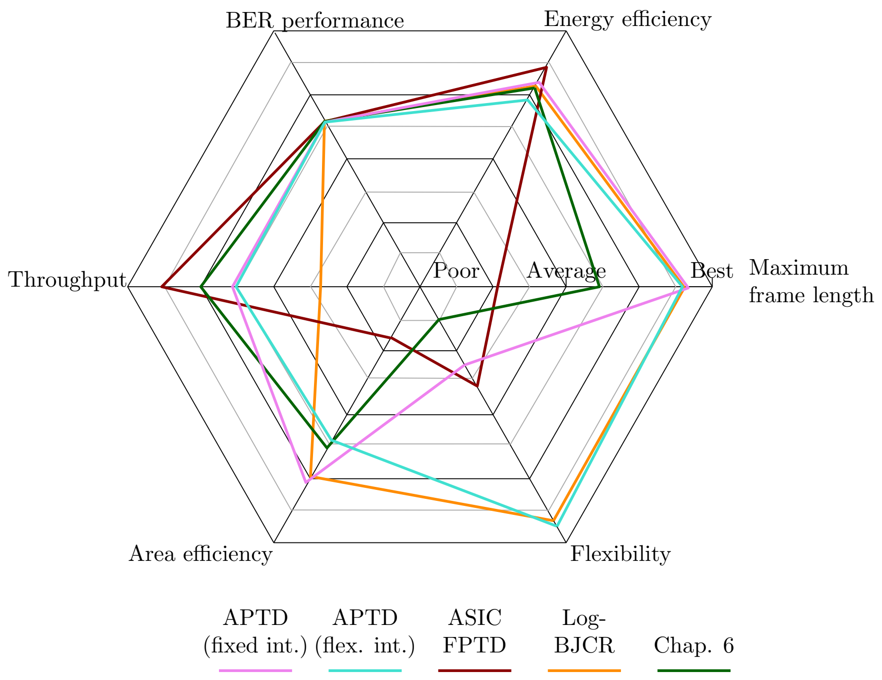


Figure 7.11: A comparison of the key performance characteristics of the proposed APTD (both with and without a flexible interleaver), the ASIC implementation of the FPTD [130] and the state-of-the-art BCJR decoder [139].

Figure 7.11 compares the performance characteristics of the proposed APTD architecture and the architectures of the benchmarkers. Here, we show the APTD with a fixed interleaver, as assumed for the results during this section, as well as an APTD with a flexible run-time configurable interleaver. This figure considers the throughput achievable for a given frame length; the area efficiency; the flexibility of the architecture to support different interleaver designs; the maximum frame length supported given a limited amount of hardware resource; the energy efficiency; and the BER performance. As demonstrated by Figure 6.20, the APTD extends the paired scheduling of Chapter 6, resulting in an increased area efficiency, and maximum supported frame length. If the fixed interleaver considered in this section was replaced with a flexible interleaver, the APTD could match the flexibility of the state-of-the-art Log-BCJR decoder, and be able to decode any of the LTE interleaver designs at run-time. Note that, as shown in Figure 6.20, this reconfigurable interleaver will trade-off some energy and area efficiency, in order to offer this considerably increased flexibility.

7.5 Conclusions

This chapter has shown that the paired scheduling of Chapter 6 can be generalised to yield the APTD, which offers high throughputs and low latency, regardless of the frame length, using all of the available hardware resources. This is achieved by varying the window length ω, while maintaining the high clock frequency offered by the pipelining methods developed in Chapter 6. This chapter has also shown that the wiring requirement of the interleaver can be reduced by carefully choosing which windows each hardware processing element operates on. The analysis of this chapter shows favourable results for the APTD compared both to the FPTD and to the paired scheduling of Chapter 6, when all three architectures assume a fixed interleaver. In particular, this chapter shows that the APTD is capable of achieving a throughput and 4.5-fold latency improvement compared to a conventional Log-BCJR benchmarker, while maintaining the same hardware efficiency. For a given frame length, the proposed APTD can match the throughput of the FPTD, while requiring about half of the hardware.

Chapter 8

Conclusions and future work

This chapter offers the concluding remarks for the thesis. Section 8.1 commences by summarising each of the chapters in turn, while Section 8.2 discusses potential future work that follows on from this thesis. Some closing remarks are offered in Section 8.3.

8.1 Conclusions

As shown in Figure 1.4, this thesis has considered both the algorithmic and implementational aspects of source codes and channel codes, with the aim of reducing the energy consumption both at the transmitter and in the decoder, while providing a more hardware efficient implementation. In particular, this thesis has explored topics surrounding the Unary Error Correction (UEC) code and its derivatives, which constitute Joint Source and Channel Coding (JSCC) schemes. More specifically, Chapters 3 and 4 have focused on the JSCC decoding algorithm, with the aim of reducing the transmit power required for reliable communications, without increasing the complexity. Following this, while Chapter 5 focused on an area-efficient JSCC decoder hardware implementation. Meanwhile, Chapter 6 has considered the joint design of both the algorithm and architecture of a JSCC decoder. This was accomplished by proposing a novel decoder algorithm, which was designed for efficient mapping to hardware. This architecture has been extended and generalised in Chapter 7. The summary of each of these chapters is as follows.

Chapter 2 began by describing the common background themes, which are used throughout this thesis. Firstly, the Logarithmic Bahl-Cocke-Jelinek-Raviv (Log-BCJR) algorithm [17] was described, which has been used in each chapter as the basis for decoding the various iterative decoder schemes. This was followed by a comparison of the techniques that are employed for overcoming the challenges associated with implementing the Log-BCJR algorithm. To demonstrate the combined algorithmic and architectural design philosophy advocated by this thesis, a technique was conceived for holistic characterisation with the aid of an application example. EXtrinsic Information Transfer (EXIT) functions [26] were also introduced, which form the basis of near-capacity analysis for the algorithmic design and characterisation of the potential codes. Finally, the UEC encoder and decoder [6] was described, which formed the basis of the schemes used throughout the thesis.

Chapter 3 introduced a UEC scheme, which included an iterative demodulator. This chapter used EXIT function analysis for demonstrating that the iterative demodulator facilitates nearer-capacity operation at a reduced complexity. The use of EXIT functions was extended from off-line code analysis to on-line adaptive activation, in order to select in which order to operate the four decoder blocks, yielding a modest but non-negligible complexity reduction. This on-line EXIT analysis also required the development of a novel reduced-complexity method to measure the Mutual Information (MI) within the receiver.

Since the UEC is a non-universal code with limited applicability, Chapter 4 proposed the Rice Error Correction (RiceEC) and Exponential Golomb Error Correction (ExpGEC) codes, which can be used for the transmission of any source symbol set derived from any monotonic source distribution. The RiceEC and ExpgEC codes constitute an extension of the EGEC code of [28], where all these codes are partly comprised of the UEC, which facilitates near-capacity operation. This chapter explores the operation of the proposed codes over a wide range of source symbols, showing that the proposed schemes can operate near to capacity without increasing the implementational complexity imposed. More specifically, the RiceEC and ExpGEC codes match or exceed the performance of the JSCC Variable Length Error-Correction (VLEC) code, while requiring an order of magnitude lower complexity.

While Chapters 3 and 4 have characterized the error correction performance of UEC based codes, Chapter 5 has demonstrated that the scheme of Chapter 3 lends itself to efficient hardware implementation. In the first implementation of the UEC code, this chapter shows how the advantages of near-capacity error correction codes can be exploited in Wireless Sensor Networks (WSNs). WSNs are typically cost sensitive, hence the architecture proposed in this section achieves a 1 Mbps throughput with a small hardware footprint, exceeding the hardware efficiency of previous similar implementations.

In contrast to Chapter 5, Chapter 6 presents an architecture capable of achieving high throughputs and low latencies, which is suitable for the next generation of wireless systems. This chapter considered the Fully Parallel Turbo Decoder (FPTD) [9], which has a high throughput but poor hardware efficiency, and proposed algorithmic modifications to increase its error correction performance, while also facilitating an implementation relying a high clock frequency. This method of jointly optimising the algorithm and architecture yields considerable hardware efficiency improvements, while maintaining a high throughput and low latency. These improvements were demonstrated for the UEC-URC code, which also constitutes part of the ExpGEC and RiceEC schemes of Chapter 4. The proposed architecture was also invoked for the LTE turbo code [18], for demonstrating the wide applicability of this architecture, and allowed comparison to existing work.

Chapter 7 extended and generalised the work of Chapter 6 to yield the Arbitrarily Parallel Turbo Decoder (APTD). More specifically, the architecture of Chapter 6 imposes a limit on the maximum frame length, due to by the size of the hardware available. However, the APTD proposed by this chapter flexibly allows any frame length to be decoded, regardless of the specific amount of available hardware, while still achieving high throughputs and low latencies. The APTD inherits the high clock frequencies of Chapter 6, whilst further improving on the hardware efficiency. This chapter only considered an LTE turbo decoder implementation, in order to allow comparison to existing architectures, whilst Chapter 6 has shown how the architecture may also be applied to a UEC scheme.

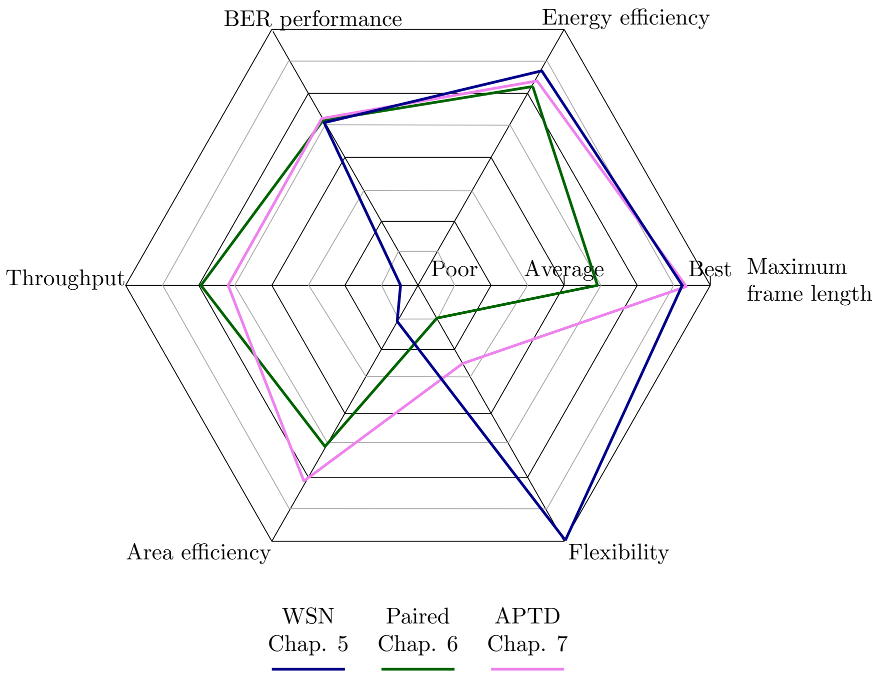


Figure 8.1: A comparison of the key performance characteristics of the architectures proposed in this thesis.

Figure 8.1 shows a comparison of the three architectures proposed in this thesis. This figure considers the throughput achievable for a given frame length; the area efficiency; the flexibility of the architecture to support different interleaver designs; the maximum frame length supported given a limited hardware resources; the energy efficiency; and the BER performance. This figure shows that since the paired scheduling decoder of Chapter 6 and the APTD of Chapter 7 were designed for different applications to the WSN turbo decoder of Chapter 5, the WSN turbo decoder has different strengths compared to the other two. More specifically, the WSN turbo decoder's operation is run-time programmable, and can also decode frames of different lengths. In comparison, the decoders of Chapters 6 and 7 have only a fixed interleaver. The WSN turbo decoder of Chapter 5 was also designed to have a small hardware footprint, and so also has much lower throughputs than the other decoders. Despite this, the area efficiency of the WSN turbo decoder is relatively poor, owing to the fact that the memory requirement is roughly the same for a high speed decoder as it is for a low speed decoder. In fact, the area of the WSN turbo decoder is dominated by the memory area. Figure 8.1 shows that the APTD of Chapter 7 extends and improves upon the paired scheduling decoder of Chapter 6, allowing a much larger range of frame lengths to be implemented using the given hardware. However, this improvement in area efficiency and maximum supported frame length comes at the cost of a reduced throughout, owing to the reduced parallelism.

8.1.1 Design guidelines

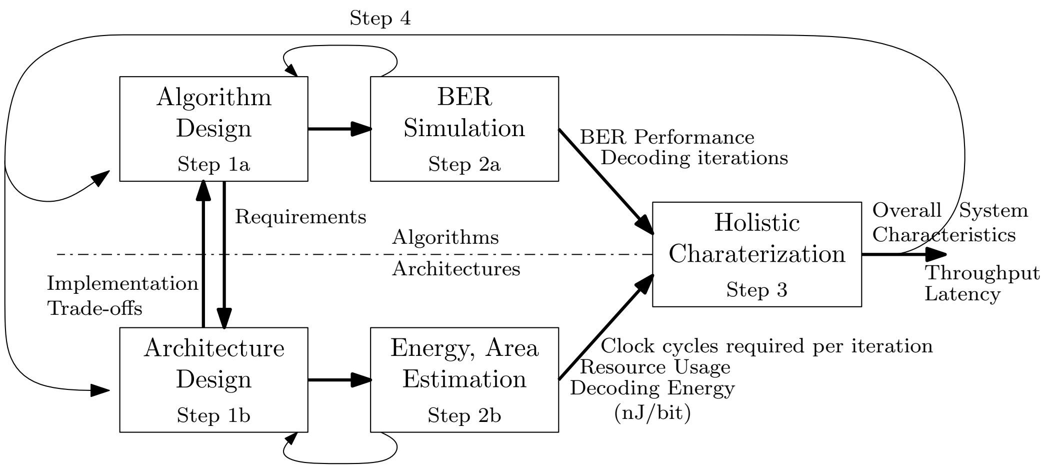


Figure 8.2: The design methodology embraced by this thesis.

Figure 8.2 shows the design flow for combined algorithm and architecture design, which is comprised of algorithmic elements and architectural elements. The last step, step 3 combines the output of the algorithmic and architectural steps, which characterizes the overall design and gives the overall system characteristics. The detailed design guidelines based on this work-flow are as follows.

- Step 1a – Given the requirements of the system, an algorithm needs to be designed which is capable of fulfilling these requirements. For example, given a high throughput video system, this thesis has opted to use the UEC code for JSCC, and the fully parallel algorithm to undertake the decoding.
- Step 1b – Given a chosen algorithm, the architecture will need to be designed to implement it. The architecture design will also feed back trade-offs which need

to be made to aid implementation back to step 1a, such as moving from floating point numbers to fixed point numbers.

- Step 2a – The performance of the algorithm will need to be analysed, in order to quantify its key characteristics, including the approximations introduced by the algorithm design.
- Step 2b – The architecture design will need to go through the design flow for the targeted platform. This will provide an estimation for the energy consumption, resource usage and clock frequency achieved by the design.
- Step 3 – The performance of the algorithm and the performance of the architecture are combined to give the overall system characteristics. For the first time, the overall performance can be determined, since changes to the algorithm will often have implications for the architecture, and vice versa.

  For example, increased pipelining may reduce the clock period and reduce the time each iteration takes, but increase the required number of iterations, which overall may increase or reduce the throughput and latency.
- Step 4 – Given the results of the characterisation, the architecture and algorithm may require further modification to meet the desired performance. The results from the BER simulation of step 2a will feedback ways to improve the algorithm in step 1a. Likewise, the results from the hardware compilation will highlight areas of improvement for the architecture design. As before, the updated architecture design and updated algorithm design steps may exchange requirements and trade-offs between each other. For example, if the hardware compilation results show a particular part of the design to be a bottleneck, we may see if the algorithm can tolerate an increased delay for that part, to allow for extra pipelining.

During the design flow of Figure 8.2, this thesis makes the following recommendations for joint architecture and algorithm design of turbo decoders.

- When considering the decoding complexity of an algorithm, use Bit Error Ratio (BER) or Symbol Error Rate (SER) simulations to investigate how the performance of the algorithm varies with different decoding complexities. With joint algorithm and architecture design, the specific architecture will need to be considered, since two schemes with the same number of Add-Compare-Select (ACS) operations per iteration may have different hardware requirements. Therefore, the specific hardware implementation will have to inform the simulations of the algorithm so that the simulations are representative of the underlying hardware.
- Identify areas of the algorithm that will lead to unfriendly implementation, and investigate how these issues can be solved while impacting on the algorithm as little as possible. For example, with the UEC, ExpGEC and RiceEC schemes,

Chapter 4 proposed a solution to the issue of frames having variable length. Since this may lead to difficulties in the interleaver design and implementation, the proposed solution allows all frames to have the same length, without impacting upon the algorithm.

- When designing an architecture with multiple different decoder blocks, identify opportunities for hardware to be shared between these different decoders. Where it is identified that different decoders may share the same hardware, aim for each to achieve a high hardware utilisation. For example, the UEC and Unity Rate Convolutional (URC) decoders of Chapter 5 both used 8 states. Investigate the algorithmic impact of these decisions.
- Identify the critical part of the datapath. This will be the part where adding more delay through pipelining will have a detrimental impact on the performance. Ensure that the critical path delays in each pipeline stage of the datapath have equal length.
- Take care to ensure that high fanout control signals are not in the critical path. In flexible designs these often drive multiplexers used to select different modes of operation. In the Log-BCJR or FPTD algorithms, these control signals can have high fanouts, due to the duplication between the r states. This issue can often be solved by ensuring that these multiplexers are positioned just before a register, rather than just after one.
- When designing hardware, a high clock speed is desirable. Since a high clock speed is achieved through pipelining, identify the parts of the algorithm that can tolerate delay without too much performance degradation, or modify the algorithm to incorporate this delay. For example, this was achieved by the paired scheduling decoder of Chapter 6, where the FPTD algorithm was modified to process trellis stages as pairs.

8.2 Future work

This section discusses potential future work, which arises from the work detailed in this thesis. While this thesis is the first one considering the hardware implementation of UEC schemes, neither the RiceEC and ExpGEC of Chapter 4, nor the adaptive scheduling of Chapter 3 have been implemented in hardware. More specifically, while the UEC-URC scheme implemented in Chapter 6 is a constituent of the RiceEC and ExpgEC of Chapter 4, more work is l required for conceiving a complete RiceEC or ExpgEC implementation. Furthermore, while a similar UEC-Turbo-QPSK scheme was considered by both Chapters 3 and 5, the implementation of Chapters 5 did not consider any of the adaptive aspects of Chapter 3. The challenges involved are as follows:

- The implementation of the Soft Bit Source Decoding (SBSD) algorithm required by the FLC decoder of Figure 4.2 must be considered, where a high-speed implementation is needed for complementing the high-speed implementation of the UEC-URC scheme.
- To maintain a high hardware efficiency, the Convolutional Code (CC) decoder of Figure 4.2 may have to be implemented on the same hardware as the UEC trellis decoder and URC decoder. Note that since all of these decoders conventionally employ the Log-BCJR algorithm but with the aid of different trellises, the solutions of Chapters 5 and 6 may be used as the basis of a decoder, which can decode these three codes.
- The novel buffering and synchronisation techniques of Chapter 4 would have to be implemented for a complete RiceEC or ExpgEC decoder.
- The adaptive algorithm of Chapter 3 will have to be translated to hardware to allow a full implementation of the scheme of Chapter 3.
- For short frame lengths, the UEC code suffers from an error floor, which has to be improved.

Throughout Chapter 6, the LTE scheme and UEC-URC scheme were jointly considered, for ensuring that the improvements of the architecture become comparable to the existing LTE implementations. While Chapter 7 extended and generalised the solutions of Chapter 6 to conceive the APTD, it only considered the LTE scheme, which allowed comparisons to be made to existing architectures. To realise the benefits of JSCC and the improvements offered by the APTD architecture of Chapter 7, the APTD architecture could be extended to support the UEC, RiceEC and ExpGEC codes. Chapter 6 already detailed the modifications required for supporting a UEC based scheme, and owing to the similarities between Chapters 6 and 7, only relatively minor modifications should be needed for the APTD to support the UEC-URC scheme.

In Chapter 6 and Chapter 7, an interleaver relying on a fixed design was used. While this may not be an issue for the UEC-URC scheme when applied to applications such as video streaming, the fixed interleaver may limit the applicability of the architectures of Chapter 6 and Chapter 7 to general purpose wireless telephony in LTE. The architecture proposed in Chapter 7 can be designed to employ a variable window length ω depending on the frame length N required, in order to ensure best exploitation of the hardware. To construct a hardware decoder that is capable of decoding any frame length, an interleaver which can vary its design depending on the frame length currently being decoded is required. Furthermore, the interleaver may have to solve the contention problem, which occurs if extrinsic LLRs from two different hardware processing elements have to be interleaved to the same destination hardware processing element. As discussed in Section 7.3.3, this may be solved either with a large hardware requirement, or by discarding some extrinsic LLRs. More investigations will be required to strike a trade-off between the extra hardware requirement and reduced error correction performance before implementing this flexible interleaver.

8.3 Closing remarks

This thesis has considered the joint design of algorithms and hardware architectures for joint source and channel codes. It has demonstrated that this holistic approach to the design and implementation of wireless communication schemes can offer significant advantages in terms of:

- transmission bandwidth and transmission throughput owing to the near-capacity spectral efficiency that is facilitated;
- transmission latency, owing to the near-entropy compression that is facilitated;
- transmission energy, owing to the near-capacity power efficiency that is facilitated;
- error correction capability, owing to the near-optimal iterative decoding;
- processing throughput and processing latency, owing to the highly parallel processing;
- processing energy and hardware resources, owing to the low complexities that are afforded.

The above mentioned holistic design approach allows attractive trade-offs between these conflicting characteristics to be struck. In particular, the transmission throughput, latency and energy can be adjusted so that they are not disproportionate to the processing throughput, latency and energy. In this way, the processing will not bottleneck the transmission and vice versa. Owing to these advantages of holistic design, it may be considered to be critical to the success of all future wireless communication systems, which are targeting ever greater performance.

References

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit error correcting coding and decoding: turbo codes,” in *Proceedings of the IEEE International Conference on Communications*, vol. 2, (Geneva, Switzerland), pp. 1064–1070, 1993.

[2] R. Gallager, “Low-density parity-check codes,” *IEEE Transactions on Information Theory*, vol. 8, pp. 21–28, jan 1962.

[3] M. Fresia, F. Perez-Cruz, H. V. Poor, and S. Verdu, “Joint source and channel coding,” *IEEE Signal Processing Magazine*, vol. 27, pp. 104–113, nov 2010.

[4] Y.-W. Yu-Wen Huang, T.-C. Tu-Chih Wang, B.-Y. Bing-Yu Hsieh, and L.-G. Liang-Gee Chen, “Hardware architecture design for variable block size motion estimation in MPEG-4 AVC/JVT/ITU-T H.264,” in *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS ’03.*, vol. 2, pp. II–796–II–799, IEEE, 2003.

[5] V. Buttigieg and P. Farrell, “Variable-length error-correcting codes,” *IEE Proceedings - Communications*, vol. 147, p. 211, aug 2000.

[6] R. G. Maunder, W. Zhang, T. Wang, and L. Hanzo, “A unary error correction code for the near-capacity joint source and channel coding of symbol values from an infinite set,” *IEEE Transactions on Communications*, vol. 61, pp. 1977–1987, 2013.

[7] J. Rissanen and G. G. Langdon, “Arithmetic coding,” *IBM Journal of Research and Development*, vol. 23, pp. 149–162, mar 1979.

[8] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” *IEEE Transactions on Information Theory*, vol. 24, pp. 530–536, sep 1978.

[9] R. G. Maunder, “A fully-parallel turbo decoding algorithm,” *IEEE Transactions on Communications*, vol. 63, pp. 2762–2775, aug 2015.

[10] C. Schurgers, F. Catthoor, and M. Engels, "Memory optimization of MAP turbo decoder algorithms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 2, pp. 305–312, 2001.

[11] S. Zezza, S. Nooshabadi, and G. Masera, "A 2.63 Mbit/s VLSI implementation of SISO arithmetic decoders for high performance joint source channel codes," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 4, pp. 951–964, 2013.

[12] C. Studer, C. Benkeser, S. Belfanti, and Q. Huang, "Design and implementation of a parallel turbo-decoder ASIC for 3GPP-LTE," *IEEE Jouranal of Solid-State Circuits*, vol. 46, no. 1, pp. 8–17, 2011.

[13] T. Ilnseher and F. Kienle, "A 2.15 GBit/s turbo code decoder for LTE advanced base station applications," in *Turbo Codes and Iterative Information Processing (ISTC), 2012 7th International Symposium on*, pp. 21–25, 2012.

[14] J. Nunez and V. Chouliaras, "High-performance arithmetic coding VLSI macro for the H264 video compression standard," *IEEE Transactions on Consumer Electronics*, vol. 51, pp. 144–151, feb 2005.

[15] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, pp. 1098–1101, 1952.

[16] C. E. Shannon, "A mathematical theory of communications," *The Bell System Technical Journal*, vol. 27, pp. 379–423,623–656, 1948.

[17] M. F. Brejza, L. Li, R. G. Maunder, B. Al-Hashimi, C. Berrou, and L. Hanzo, "20 years of turbo coding and energy-aware design guidelines for energy-constrained wireless applications," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 8–28, 2016.

[18] "ETSI TS 136 212 LTE; Evolved Universal Terrestrial Radio Access (E- UTRA); Multiplexing and channel coding," *V12.0.0 ed*, 2013.

[19] D. MacKay and R. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 33, no. 6, p. 457, 1997.

[20] G. Auer, V. Giannini, and C. Desset, "How much energy is needed to run a wireless network?," *IEEE Wireless Communications*, vol. 18, no. 5, pp. 40–49, 2011.

[21] R. G. Gallager and D. C. Van Voorhis, "Optimal source codes for geometrically distributed integer alphabets," *IEEE Transactions on Information Theory*, pp. 228–230, 1974.

[22] D. MacKay, *Information theory, inference and learning algorithms.* Cambridge University Press, 2003.

[23] M. Bernard and B. Sharma, "Some combinatorial results on variable length error correcting codes," *Ars Combinatoria*, pp. 181–194, 1988.

[24] D. Kim and T. Kwon, "A modified two-step SOVA-based turbo decoder with a fixed scaling factor," *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, 2000.

[25] C. Benkeser, A. Burg, T. Cupaiuolo, and Q. Huang, "Design and optimization of an HSDPA turbo decoder ASIC," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 1, pp. 98–106, 2009.

[26] J. Hagenauer, "The EXIT Chart – introduction to extrinsic information transfer in iterative processing," in *Proc. 12th European Signal Processing Conf.*, (Vienna), pp. 1541–1548, 2004.

[27] W. Zhang, Y. Jia, X. Meng, M. Brejza, R. G. Maunder, and L. Hanzo, "Adaptive iterative decoding for expediting the convergence of unary error correction codes," *IEEE Transactions on Vehicular Technology*, vol. 64, pp. 621–635, may 2014.

[28] T. Wang, W. Zhang, R. G. Maunder, and L. Hanzo, "Near-capacity joint source and channel coding of symbol values from an infinite source set using elias gamma error correction codes," *IEEE Transactions on Communications*, vol. 62, pp. 280–292, jan 2014.

[29] P. Elias, "Coding for noisy channels," in *IRE Convention Record Pt. 4*, vol. 3, pp. 37–46, 1955.

[30] D. Tse and P. Viswanath, *Fundamentals of wireless communication.* Cambridge: Cambridge University Press, 2005.

[31] N. Sadeghi, S. Howard, S. Kasnavi, K. I. V. C. Gaudet, and C. Schlegel, "Analysis of error control code use in ultra-low-power wireless sensor networks," in *Proceedings of International Symposium on Circuits and Systems*, (Island of Kos), pp. 3558–3561, 2006.

[32] S. L. Howard, C. Schlegel, and K. Iniewski, "Error control coding in low-power wireless sensor networks: when is ECC energy-efficient?," *EURASIP Journal of Wireless Communications and Networking, Special Issue: CMOS RF Circuits for Wireless Applications*, vol. 2006, Arti, pp. 1–14, 2006.

[33] V. S. Annapureddy and V. V. Veeravalli, "Gaussian interference networks: sum capacity in the low-interference regime and new outer bounds on the capacity region," *IEEE Transactions on Information Theory*, vol. 55, pp. 3032–3050, jul 2009.

[34] W. T. W. L. Hanzo, S. X. Ng, T. Keller, *Quadrature amplitude modulation: from basics to adaptive trellis-coded, turbo-equalised and space-time coded OFDM, CDMA and MC-CDMA systems.* Wiley-IEEE Press, 3rd ed., 2004.

[35] B. Sklar, “Rayleigh fading channels in mobile digital communication systems Part I: Characterization,” *IEEE Communications Magazine*, vol. 35, pp. 90–100, jul 1997.

[36] J. Hagenauer, “The turbo principle: Tutorial introduction and state of the art,” in *Proc. International Symposium on Turbo Codes*, (Brest, France), pp. 1–11, 1997.

[37] P. Robertson, E. Villebrun, and P. Hoeher, “A comparison of optimal and Sub-optimal MAP decoding algorithms operating in the log domain,” in *Proceedings of IEEE International Conference of Communication*, vol. 2, (Seattle, WA, USA), pp. 1009–1013, 1995.

[38] L. Hanzo, T. H. Liew, B. L. Yeap, R. Tee, and S. X. Ng, *Turbo Coding, Turbo Equalisation and Space-Time Coding.* John Wiley & Sons Inc, 2002.

[39] L. Hanzo, J. P. Woodard, and P. Robertson, “Turbo Decoding and Detection for Wireless Applications,” in *Proceedings of the IEEE*, vol. 95, pp. 1178–1200, jun 2007.

[40] J. Vogt, K. Koors, A. Finger, and G. Fettweis, “Comparison of different turbo decoder realizations for IMT-2000,” in *Seamless Interconnection for Universal Services. Global Telecommunications Conference. GLOBECOM’99. (Cat. No.99CH37042)*, vol. 5, pp. 2704–2708, IEEE, 1999.

[41] P. Robertson, P. Hoeher, and E. Villebrun, “Optimal and Sub-Optimal Maximum A Posteriori Algorithms Suitable for Turbo Decoding,” *European Transactions on Telecommunications*, vol. 8, no. 2, pp. 119–125, 1997.

[42] C.-H. Lin, C.-Y. Chen, E.-J. Chang, and A.-Y. Wu, “A 0.16nJ/bit/iteration 3.38mm2 turbo decoder chip for WiMAX/LTE standards,” in *Integrated Circuits (ISIC), 2011 13th International Symposium on*, pp. 168–171, 2011.

[43] Y. Sun and J. R. Cavallaro, “Efficient hardware implementation of a highly-parallel 3GPP LTE/LTE-advance turbo decoder,” *Integration, the VLSI Journal*, vol. 44, pp. 305–315, sep 2011.

[44] D. Yoge and N. Chandrachoodan, “GPU Implementation of a Programmable Turbo Decoder for Software Defined Radio Applications,” *VLSI Design (VLSID)*, pp. 149–154, 2012.

[45] C. Roth, S. Belfanti, C. Benkeser, and Q. Huang, “Efficient parallel turbo-decoding for high-throughput wireless systems,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, pp. 1824–1835, jun 2014.

[46] M. A. Bickerstaff, D. Garrett, T. Prokop, C. Thomas, B. Widdup, G. Zhou, L. M. Davis, G. Woodward, C. Nicol, and R.-H. Yan, “A unified turbo/viterbi channel

decoder for 3GPP mobile wireless in 0.18-μm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1555–1564, 2002.

[47] M. Bickerstaff, L. Davis, C. Thomas, D. Garrett, and C. Nicol, "A 24Mb/s radix-4 logMAP turbo decoder for 3GPP-HSDPA mobile wireless," in *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*, vol. 1, pp. 150–151,484, Ieee, 2003.

[48] F.-M. Li, C.-H. Lin, and A.-Y. Wu, "Unified Convolutional/Turbo Decoder Design Using Tile-Based Timing Analysis of VA/MAP Kernel," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1063–8210, 2008.

[49] M. May, T. Ilnseher, N. Wehn, and W. Raab, "A 150Mbit/s 3GPP LTE Turbo Code Decoder," in *Proceedings of the Conference on Design, Automation and Test in Europe*, (Dresden, Germany), pp. 1420–1425, European Design and Automation Association, mar 2010.

[50] Y. Sun and J. R. Cavallaro, "Efficient Hardware Implementation of A Highly-Parallel 3GPP LTE, LTE-Advance Turbo Decoder," *Integration, the VLSI Journal*, vol. 44, no. 1, pp. 1–11, 2010.

[51] L. Li, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "A low-complexity turbo decoder architecture for energy-efficient wireless sensor networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, pp. 14–22, jan 2013.

[52] C. Studer, S. Fateh, C. Benkeser, and Q. Huang, "Implementation Trade-Offs of Soft-Input Soft-Output MAP Decoders for Convolutional Codes," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 59, no. 11, pp. 2774–2783, 2012.

[53] S. Belfanti, C. Roth, M. Gautschi, C. Benkeser, and Qiuting Huang, "A 1Gbps LTE-advanced turbo-decoder ASIC in 65nm CMOS," in *VLSI Circuits (VLSIC), 2013 Symposium on*, pp. C284–C285, 2013.

[54] J. Chen and J. Hu, "High throughput stochastic turbo decoder based on low bits computation," *Signal Processing Letters, IEEE*, vol. 20, no. 11, pp. 1098–1101, 2013.

[55] Q. T. Dong, M. Arzel, C. Jego, and W. J. Gross, "Stochastic decoding of turbo codes," *IEEE Transactions on Signal Processing*, vol. 58, pp. 6421–6425, dec 2010.

[56] D. Vogrig, A. Gerosa, A. Neviani, A. Amat, G. Montorsi, and S. Benedetto, "A 0.35-/spl mu/m CMOS analog turbo decoder for the 40-bit rate 1/3 UMTS channel code," *IEEE Journal of Solid-State Circuits*, vol. 40, pp. 753–762, mar 2005.

[57] V. Gaudet and P. Gulak, "A 13.3-Mb/s 0.35-μm CMOS analog turbo decoder IC with a configurable interleaver," *IEEE Journal of Solid-State Circuits*, vol. 38, pp. 2010–2015, nov 2003.

[58] M. Arzel, C. Lahuec, F. Seguin, D. Gnaedig, and M. Jezequel, "Semi-iterative analog turbo decoding," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, pp. 1305–1316, jun 2007.

[59] J. Vogt and A. Finger, "Improving the Max-Log-MAP turbo decoder," *Electronics Letters*, vol. 36, no. 23, pp. 1937–1939, 2000.

[60] M. van Dijk, "Correcting systematic mismatches in computed log-likelihood ratios," *European Transactions on Telecommunications*, vol. 14, no. 3, pp. 227–244, 2003.

[61] A. Nimbalker, Y. Blankenship, B. Classon, and T. K. Blankenship, "ARP and QPP interleavers for LTE turbo coding," *2008 IEEE Wireless Communications and Networking Conference*, pp. 1032–1037, mar 2008.

[62] S. Crozier, P. Guinand, and A. Hunt, "Estimating the minimum distance of turbo-codes using double and triple impulse methods," *IEEE Communications Letters*, vol. 9, pp. 631–633, jul 2005.

[63] A. Ardakani, M. Mahdavi, and M. Shabany, "An efficient VLSI architecture of QPP interleaver/deinterleaver for LTE turbo coding," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, pp. 797–800, IEEE, may 2013.

[64] J. Sun and O. Takeshita, "Interleavers for turbo codes using permutation polynomials over integer rings," *Information Theory, IEEE Transactions on*, 2005.

[65] Y. Sun, Y. Zhu, M. Goel, and J. R. Cavallaro, "Configurable and Scalable High Throughput Turbo Decoder Architecture for Multiple 4G Wireless Standards," in *IEEE International Conference on Application-specific System, Architectures and Processors*, (Leuven, Belgium), pp. 209–214, 2008.

[66] L. Li, R. Maunder, B. Al-Hashimi, M. Zwolinski, and L. Hanzo, "Energy-conscious turbo decoder design: a joint signal processing and transmit energy reduction approach," *IEEE Transactions on Vehicular Technology*, oct 2013.

[67] S. Jayaweera, "Virtual MIMO-based cooperative communication for energy-constrained wireless sensor networks," *IEEE Transactions on Wireless Communications*, vol. 5, pp. 984–989, may 2006.

[68] S. Cui, A. Goldsmith, and A. Bahai, "Energy-efficiency of MIMO and cooperative MIMO techniques in sensor networks," *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 1089–1098, aug 2004.

[69] E. Calvanese Strinati and L. Hérault, "Holistic approach for future energy efficient cellular networks," *e & i Elektrotechnik und Informationstechnik*, vol. 127, pp. 314–320, nov 2010.

[70] H. Chen, R. G. Maunder, and L. Hanzo, "Low-Complexity Multiple-Component Turbo-Decoding-Aided Hybrid ARQ," *IEEE Transactions on Vehicular Technology*, vol. 60, pp. 1571–1577, may 2011.

[71] R. Souza, M. Pellenz, and T. Rodrigues, "Hybrid ARQ scheme based on recursive convolutional codes and turbo decoding," *IEEE Transactions on Communications*, vol. 57, pp. 315–318, 2009.

[72] L. Li, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "Energy-Conscious Turbo Decoder Design: A Joint Signal Processing and Transmit Energy Reduction Approach (Submitted)," *IEEE Transactions on Vehicular Technology*, 2013.

[73] A. Wong, M. Dawkins, G. Devita, N. Kasparidis, A. Katsiamis, O. King, F. Lauria, J. Schiff, and A. Burdett, "A 1V 5mA multimode IEEE 802.15.6/bluetooth low-energy WBAN transceiver for biotelemetry applications," in *2012 IEEE International Solid-State Circuits Conference*, pp. 300–302, IEEE, feb 2012.

[74] D. Divsalar, H. Jin, and R. J. McEliece, "Coding theorems for turbo-like' codes," in *Proceeding of 36th Allerton Conf. on Communication, Control and Computing*, (Allerton, Illinois), pp. 201–210, 1998.

[75] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *SIAM Journal of Applied Math*, vol. 8, pp. 300–304, 1960.

[76] K. S. Andrews, D. Divsalar, S. Dolinar, J. Hamkins, C. R. Jones, and F. Pollara, "The development of turbo and LDPC codes for deep-space applications," *Proceedings of the IEEE*, vol. 95, pp. 2142–2156, nov 2007.

[77] H. Chen, R. G. Maunder, and L. Hanzo, "A survey and tutorial on low-complexity turbo coding techniques and a holistic hybrid ARQ design example," jan 2013.

[78] L. Li, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "An energy-efficient error correction scheme for IEEE 802.15. 4 wireless sensor networks," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 57, no. 3, pp. 233–237, 2010.

[79] L. Hanzo, R. G. Maunder, J. Wang, and L. Yang, *Near-capacity variable-length coding: regular and EXIT-chart-aided irregular designs.* 2011.

[80] A. Ashikhmin, "Extrinsic information transfer functions: model and erasure channel properties," *Information Theory, IEEE Transactions on*, vol. 50, no. 11, pp. 2657–2673, 2004.

[81] A. J. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *IEEE Transactions on Information Theory*, vol. IT-13, pp. 493–497, 1967.

[82] P. Elias, “Universal codeword sets and representations of the integers,” *IEEE Transactions on Information Theory*, vol. 21, pp. 194–203, mar 1975.

[83] J. Massey, “Joint source and channel coding,” *Communication Systems and Random Process Theory*, 1977.

[84] Q. Stout, “Improved prefix encodings of the natural numbers,” *IEEE Trans. Inform. Theory*, 1980.

[85] A. Fraenkel and S. Kleinb, “Robust universal complete codes for transmission and compression,” *Discrete Applied Mathematics*, vol. 64, no. 1, pp. 31–55, 1996.

[86] R. Bauer and J. Hagenauer, “Symbol-by-symbol MAP decoding of variable length codes,” in *Proc. 3. ITG Conf. on Source and Channel Coding*, pp. 111–116, 2000.

[87] N. Gortz, “Iterative source-channel decoding using soft-in/soft-out decoders,” in *2000 IEEE Int. Symp. on Information Theory*, (Sorrento), p. 173, 2000.

[88] N. Gortz, “On the iterative approximation of optimal joint source-channel decoding,” *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 9, pp. 1662–1670, 2001.

[89] D. Salomon, *Data compression: the complete reference.* Springer Science & Business Media, 2004.

[90] N. Johnson, A. Kemp, and S. Kotz, *Univariate discrete distributions.* John Wiley & Sons, 2005.

[91] J. Hagenauer and N. Gortz, “The Turbo Principle in Joint Source-Channel Coding,” in *Proceedings 2003 IEEE Information Theory Workshop*, pp. 275–278, IEEE, 2003.

[92] L. Hanzo, R. G. Maunder, J. Wang, and L.-L. Yang, *Near-capacity variable-length coding.* Chichester, UK: John Wiley & Sons, Ltd, oct 2010.

[93] N. F. Kiyani and J. H. Weber, “Iterative demodulation and decoding for rotated MPSK constellations with convolutional coding and signal space diversity,” in *2007 IEEE 66th Vehicular Technology Conference*, pp. 1712–1716, IEEE, sep 2007.

[94] M. Valenti and S. Cheng, “Iterative demodulation and decoding of turbo-coded M-ary noncoherent orthogonal modulation,” *Selected Areas in Communications, IEEE Journal on*, 2005.

[95] S. ten Brink, G. Kramer, and A. Ashikhmin, "Design of low-density parity-check codes for modulation and detection," *IEEE Transactions on Communications*, vol. 52, pp. 670–678, apr 2004.

[96] W. Zhang, R. G. Maunder, and L. Hanzo, "On the complexity of unary error correction codes for the near-capacity transmission of symbol values from an infinite set," in *IEEE Wireless Communications and Networking Conference 2013*, no. IID, pp. 1–6, oct 2012.

[97] ITU-T, "Series H: audiovisual and multimedia systems, infrastructure of audiovisual services coding of moving video, high efficiency video coding," 2015.

[98] D. Rowitch and L. Milstein, "On the performance of hybrid FEC/ARQ systems using rate compatible punctured turbo (RCPT) codes," *IEEE Transactions on Communications*, vol. 48, pp. 948–959, jun 2000.

[99] M. Bernard and B. Sharma, "A lower bound on average codeword length of variable length error-correcting codes," *IEEE Transactions on Information Theory*, vol. 36, no. 6, pp. 1474–1475, 1990.

[100] R. G. Maunder and L. Hanzo, "Near-capacity irregular variable length coding and irregular unity rate coding," *IEEE Transactions on Wireless Communications*, vol. 8, no. 11, pp. 5500–5507, 2009.

[101] J. Kliewer, A. Huebner, and D. Costello, "On the achievable extrinsic information of inner decoders in serial concatenation," in *2006 IEEE Int. Symp. on Information Theory*, (Seattle), pp. 2680–2684, jul 2006.

[102] R. G. Maunder and L. Hanzo, "Genetic algorithm aided design of component codes for irregular variable length coding," *IEEE Transactions on Communications*, vol. 57, pp. 1290–1297, may 2009.

[103] M. Adrat, R. Vary, and J. Spittka, "Iterative source-channel decoder using extrinsic information from softbit-source decoding," in *2001 IEEE Int. Conf. on Acoustics, Speech, and Signal Processing. Proc.*, pp. 2653–2656 vol.4, 2001.

[104] T. generation partnership project, "Multiplexing and channel coding (Release 8). TS36. 212 V8.4.0," 2008.

[105] S. Dolinar and D. Divsalar, "Weight distributions for turbo codes using random and nonrandom permutations," *JPL Progress report 42-122*, pp. 56–65, 1995.

[106] J. Wang, L.-L. Yang, and L. Hanzo, "Iterative construction of reversible variable-length codes and variable-length error-correcting codes," *IEEE Communications Letters*, vol. 8, pp. 671–673, nov 2004.

[107] J. Proakis, *Digital communications.* McGraw-Hill, 1983.

[108] J. Lee and R. Blahut, “Generalized EXIT chart and BER analysis of finite-length turbo codes,” *Global Telecommunications Conf. 2003*, pp. 2067–2072 vol.4, 2003.

[109] M. Zorzi and A. Gluhak, “From today’s intranet of things to a future internet of things: a wireless-and mobility-related view,” *Wireless Communications, IEEE*, vol. 17, no. 6, pp. 44–51, 2010.

[110] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: a survey,” *Computer Networks*, vol. 54, pp. 2787–2805, oct 2010.

[111] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, “Internet of Things (IoT): A vision, architectural elements, and future directions,” *Future Generation Computer Systems*, vol. 29, pp. 1645–1660, sep 2013.

[112] R. G. Maunder, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, and L. Hanzo, “Iterative secoding for redistributing energy consumption in wireless sensor networks,” in *2008 Proceedings of 17th International Conference on Computer Communications and Networks*, pp. 1–6, IEEE, aug 2008.

[113] Y. Qassim and M. E. Magana, “Error-tolerant non-binary error correction code for low power wireless sensor networks,” in *The International Conference on Information Networking 2014 (ICOIN2014)*, pp. 23–27, IEEE, feb 2014.

[114] A. Abedi, “Power-efficient-coded architecture for distributed wireless sensing,” *IET Wireless Sensor Systems*, vol. 1, pp. 129–136, sep 2011.

[115] “Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs),” 2006.

[116] N. Noels, C. Herzet, A. Dejonghe, V. Lottici, H. Steendam, M. Moeneclaey, M. Luise, and L. Vandendorpe, “Turbo synchronization: an EM Algorithm Interpretation,” *IEEE International Conference on Communications, 2003. ICC ’03.*, vol. 4, pp. 2933–2937, 2003.

[117] S. Haene, A. Burg, N. Felber, and W. Fichtner, “OFDM channel estimation algorithm and ASIC implementation,” in *2008 4th European Conference on Circuits and Systems for Communications*, pp. 270–275, IEEE, jul 2008.

[118] C. Douillard, M. Jézéquel, and C. Berrou, “Iterative correction of intersymbol interference: Turbo-equalization,” in *European transactions on telecommunications*, vol. 6, pp. 507–511, sep 1995.

[119] T. Austin, D. Blaauw, T. Mudge, K. Flautner, M. Irwin, M. Kandemir, and V. Narayanan, “Leakage current: moore’s law meets static power,” *Computer*, vol. 36, pp. 68–75, dec 2003.

[120] A. D. G. Biroli, M. Martina, and G. Masera, “An LDPC decoder architecture for wireless sensor network applications,” *Sensors*, vol. 12, pp. 1529–1543, feb 2012.

[121] M. F. Brejza, W. Zhang, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, “Adaptive iterative detection for expediting the convergence of a serially concatenated unary error correction decoder, turbo decoder and an iterative demodulator,” in *Communications (ICC), 2015 IEEE International Conference on*, pp. 2603 – 2608, jun 2015.

[122] J. Kim, K. R. Vijayanagar, and W. Liu, “Low-complexity distributed multiple description coding for wireless video sensor networks,” *IET Wireless Sensor Systems*, vol. 3, pp. 205–215, sep 2013.

[123] I. Balasingham, H. Nguyen, and T. Ramstad, “Wireless sensor communication system based on direct-sum source coder,” *IET Wireless Sensor Systems*, vol. 1, pp. 96–104, jun 2011.

[124] W.-T. Chen, P.-Y. Chen, W.-S. Lee, and C.-F. Huang, “Design and implementation of a real time video surveillance system with wireless sensor networks,” in *VTC Spring 2008 - IEEE Vehicular Technology Conference*, pp. 218–222, IEEE, may 2008.

[125] S. Papaharalabos, P. Mathiopoulos, G. Masera, and M. Martina, “On optimal and near-optimal turbo decoding using generalized max operator,” *IEEE Communications Letters*, vol. 13, pp. 522–524, jul 2009.

[126] S. Papaharalabos, P. Sweeney, B. Evans, P. Mathiopoulos, G. Albertazzi, A. Vanelli-Coralli, and G. Corazza, “Modified sum-product algorithms for decoding low-density parity-check codes,” *IET Communications*, vol. 1, no. 3, p. 294, 2007.

[127] N. L. Johnson, A. W. Kemp, and S. Kotz, *Univariate discrete distributions.* John Wiley & Sons, 2005.

[128] L. Li, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, “Design of Fixed-Point Processing Based Turbo Codes Using Extrinsic Information Transfer Charts,” in *Proceeding of IEEE Vehicular Technology Conference*, (Ottawa, Canada), pp. 1–5, 2010.

[129] A. Li, P. Hailes, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, “1.5 Gbit/s FPGA implementation of a fully-parallel turbo decoder designed for mission-critical machine-type communication applications,” *IEEE Access*, vol. PP, no. 99, 2016.

[130] A. Li, L. Xiang, T. Chen, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, “VLSI implementation of fully-parallel LTE turbo decoders,” *IEEE Access*, vol. 4, pp. 323 – 346, jan 2016.

[131] J.-H. Kim and I.-C. Park, “A unified parallel radix-4 turbo decoder for mobile WiMAX and 3GPP-LTE,” in *IEEE Custom Integrated Circuits Conference*, (San Jose, CA), pp. 487–490, 2009.

[132] S. Pietrobon, “Efficient implementation of continuous MAP decoders and a synchronisation technique for turbo decoders,” *Proc. Int. Symp. Information Theory Applied*, pp. 586–589, 1996.

[133] R. Garello, F. Chiaraluce, P. Pierleoni, M. Scaloni, and S. Benedetto, “On error floor and free distance of turbo codes,” in *IEEE International Conference on Communications.*, vol. 1, pp. 45–49, IEEE, 2001.

[134] J. Hagenauer, E. Offer, and L. Papke, “Iterative decoding of binary block and convolutional codes,” *IEEE Transactions on Information Theory*, vol. 42, pp. 429–445, mar 1996.

[135] S. Hong, J. Yi, and W. Stark, “VLSI design and implementation of low-complexity adaptive turbo-code encoder and decoder for wireless mobile communication applications,” in *IEEE Workshop on Signal Processing Systems*, pp. 233–242, IEEE, 1998.

[136] A. I. V. Casado, M. Griot, and R. D. Wesel, “Informed dynamic scheduling for belief-propagation decoding of LDPC codes,” in *2007 IEEE International Conference on Communications*, pp. 932—-937, 2007.

[137] P. Hailes, L. Xu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, “A survey of FPGA-based LDPC decoders,” *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 1098–1122, jan 2016.

[138] Altera, “Chapter 1. Overview for the stratix IV device family,” 2012.

[139] L. F. Gonzalez-Perez, L. C. Yllescas-Calderon, and R. Parra-Michel, “Parallel and configurable turbo decoder implementation for 3GPP-LTE,” in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, (Cancun, Mexico), pp. 1–6, IEEE, dec 2013.

Index