Code Generation and Simulation of an Automatic, Flexible QC-LDPC Hardware Decoder

by

Mirko von Leipzig

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Electronic Engineering in the Faculty of Engineering at Stellenbosch University

Department of Electrical & Electronic Engineering,
Stellenbosch University,
Private Bag X1, Matieland 7602, South Africa

Supervisor: Dr G-J van Rooyen

March 2015

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature: ....
M. von Leipzig

Date: November 2014 ....

Abstract

Iterative error correcting codes such as LDPC codes have become prominent in modern forward error correction systems. A particular subclass of LDPC codes known as quasi-cyclic LDPC codes has been incorporated in numerous high speed wireless communication and video broadcasting standards. These standards feature multiple codes with varying codeword lengths and code rates and require a high throughput. Flexible hardware that is capable of decoding multiple quasi-cyclic LDPC codes is therefore desirable.

This thesis investigates binary quasi-cyclic LDPC codes and designs a generic, flexible VHDL decoder. The decoder is further enhanced to automatically select the most likely decoder based on the initial *a posterior* probability of the parity-check equation syndromes.

A software system is developed that generates hardware code for such a decoder based on a small user specification. The system is extended to provide performance simulations for this generated decoder.

Uitreksel

Iteratiewe foutkorreksiekodes soos LDPC-kodes word wyd gebruik in moderne voorwaartse foutkorreksiestelsels. ’n Subklas van LDPC-kodes, bekend as kwasisikliese LDPC-kodes, word in verskeie hoëspoed-kommunikasie- en video-uitsaaistelselstandaarde gebruik. Hierdie standaarde inkorporeer verskeie kodes van wisselende lengtes en kodetempos, en vereis hoë deurset. Buigsame apparatuur, wat die vermoë het om ’n verskeidenheid kwasisikliese LDPC-kodes te dekodeer, is gevolglik van belang.

Hierdie tesis ondersoek binêre kwasisikliese LDPC-kodes, en ontwerp ’n generiese, buigsame VHDL-dekodeerder. Die dekodeerder word verder verbeter om outomaties die mees waarskynlike dekodeerder te selekteer, gebaseer op die aanvanklike a posteriori-waarskynlikheid van die pariteitstoetsvergelykings se sindrome.

’n Programmatuurstelsel word ontwikkel wat die fermware-kode vir so ’n dekodeerder genereer, gebaseer op ’n beknopte gebruikerspesifikasie. Die stelsel word uitgebrei om werksverrigting te simuleer vir die gegenereerde dekodeerder.

Acknowledgements

I would like to express my sincere gratitude to the following people:

- my parents, for their support, concern and motivation,
- my sister, for her company and constant niggling,
- Gert-Jan van Rooyen, for being a valuable bouncing board for my ideas,
- my friends, for their critique and distractions.

Contents

List of Figures

List of Algorithms

List of Code Snippets

Nomenclature

Vectors and Matrices

$\mathbf{x}$	Row vector
$\mathbf{x}^T$	Column vector
x_i	Element i of vector $\mathbf{x}$
$\mathbf{X}$	Matrix
$X_{i,j}$	Element in row i, column j of matrix $\mathbf{X}$

Messages and Nodes

v_i	Variable node i
f_j	Function node j
$Q_{i\text{-}j}$	Message from some node i to node j
$Q^0_{i\text{-}j}$	Message containing the probability of a bit being zero
$Q^1_{i\text{-}j}$	Message containing the probability of a bit being one
$Q^{\mathrm{D}}_{i\text{-}j}$	Message containing the probabilities of a bit being zero or one as a tuple
$Q^{\mathrm{LR}}_{i\text{-}j}$	Message containing the likelihood ratio of a bit
$Q^{\mathrm{LLR}}_{i\text{-}j}$	Message containing the log-likelihood ratio of a bit

Variables

E_b	Energy per bit
η	Normalising constant
ε	Binary symmetric channel crossover probability
n	Codeword length i.e. number of bits
m	Number of parity equations i.e. number of parity bits
n^b	Quasi-cyclic LDPC parity matrix block columns
m^b	Quasi-cyclic LDPC parity matrix block rows
B	Block size
Π	Quasi-cyclic LDPC permutation value

$\mathbf{\Pi}$	Quasi-cyclic LDPC permutation matrix
G	Global function
G_i	Marginal function i of global function G
g	Local factor function of global function G
σ	Transmission medium
C	A code
γ	Characteristic function
α	Key parity-check node used in stopping-set encoding
β	Key parity bit node used in stopping-set encoding
Γ_θ	Average syndrome log-likelihood of code C_θ

Operators

$\underset{j}{\overset{\sim i}{\Box}}$	Perform operation $\Box$ over all valid values of j excluding i
$\oplus$	Logical XOR operation
$\ln$	Natural logarithm function
e	Natural exponent
sgn	Signum function
$L(x)$	Log-likelihood of x

List of Abbreviations

APP	*a posteriori* probability
BER	bit error rate
BPSK	binary phase-shift key
FEC	forward error correction
GPL	Go Programming Language
IP	intellectual property
LDPC	low-density parity-check
LLR	log-likelihood ratio
LSB	least significant bit
QC-LDPC	quasi-cyclic low-density parity-check
RAM	random-access memory
ROM	read-only memory
SPA	sum-product algorithm
SNR	signal-to-noise ratio
VHDL	VHSIC Hardware Description Language
XML	Extensible Markup Language

Chapter 1

Introduction

In recent years low-density parity-check (LDPC) codes have been included in multiple communications standards [1]. These standards usually include multiple LDPC codes with different code rates [1]. In this thesis a hardware decoding system capable of supporting an arbitrary set of binary, structured LDPC codes is developed. The decoder can support codes of different rates as well as different block sizes. The decoder is further developed to automatically select and decode the most likely code of the set. Applications of this technology include cognitive radio receivers and systems in which the encoder and decoder have no means to communicate a change in code.

1.1 Background

Modern digital communications systems need to communicate information across noisy mediums or channels [2]. This is usually achieved by encoding the information bits into codeword bits at the sender. The encoding process adds redundant data to the information data according to some deterministic algorithm [2], which in turn allows the receiver to correct errors in the received data. This error correction can be achieved according to one of two general methodologies. The automatic repeat request methodology detects errors in the received data and requests a retransmission of the data if any are found [2]. The forward error correction (FEC) methodology uses the redundant data to correct errors, if any, in the received data without requiring retransmission [2].

The Shannon capacity of a channel is the upper bound on the rate of information transfer for a given channel bandwidth and signal-to-noise (SNR) power ratio with an arbitrarily small error probability [3]. In the 1990s, a family of FEC codes known as Turbo codes were discovered [3]. Turbo codes were the first codes capable of nearing the Shannon capacity of a channel. Prior to their discovery, it was believed that gains in channel capacity required increasing decoding complexity [3]. Turbo codes disproved this belief and consequently became widely adopted [2]. This led to vigorous investigation of other, similar codes. One such code family are the LDPC codes [3]. These LDPC codes are named for their sparse parity-check matrix.

1.2 Motivation for Work

LDPC codes have shown similar error correcting performance to Turbo codes at high codeword lengths and rates [3]. They have been incorporated into numerous communication's standards, particularly in the high-speed video broadcast and wireless communications areas [1]. LDPC decoding complexity is linear with respect to codeword length while encoding is quadratic [4]. The long codeword length and high bit rate requirements of most standards incorporating LDPC codes cause both encoding and decoding speed to be issues in most designs [1]. LDPC codes are therefore often structured to help speed up encoding and decoding [1] such that both encoding and decoding of these structured codes is inherently parallel. Decoders for structured codes are therefore typically built on custom hardware [1] in order to meet the stringent speed requirements.

LDPC codes exist mainly as binary codes, however q-ary LDPC codes can and have been constructed. This thesis focuses exclusively on binary codes, however similar concepts could be used to incorporate q-ary codes.

Most standards specify multiple LDPC codes within a single standard [1], including codes of different codeword lengths and code rates. This calls for flexible hardware decoders capable of decoding any of the codes specified by a standard [1]. It would be beneficial if such a decoder design could easily be adapted to different standards without much effort. A system that generates code based on some user configuration solves this need. Compiling, synthesising and testing large hardware designs is time consuming [5]. It is therefore of use to have a means of simulating the performance of the hardware decoder prior to the implementation thereof, to allow the user to make configuration tweaks to better meet the expectations.

This work investigates binary LDPC codes and the design of an autonomous, flexible decoder capable of decoding a set of structured LDPC codes, as well as a code generation system that generates hardware code for such a decoder. A simulation package is also developed to simulate the performance of the decoder.

1.3 Objectives

The objectives of this thesis are to

- design a flexible hardware decoder for arbitrary, structured LDPC codes,
- automate the decoder in so far as possible,
- design software to generate the hardware code for such a decoder based on user configuration,
- simulate such a decoder's performance.

1.4 Contributions

This thesis makes the following contributions.

- We investigate and compile a wide range of existing LDPC decoding and encoding techniques.
- We investigate the proposed technique of Xia et al. [6] which allows us to find the most likely code of a set based on the received codeword. We show that this technique can be exploited at minimal extra cost to allow our decoder to autonomously select the correct code during the decoding process.
- We develop a software tool capable of generating code for a hardware decoder based on user configuration.
- We also develop a software model of the hardware decoder. This, coupled with software models of the communications system, allows us to simulate the performance of a decoder under certain channel and modulation conditions. This gives feedback to

the user without the user needing to test the hardware directly. Software simulation is much quicker and does not require the recompilation of a large hardware project when the configuration changes. These simulation models are abstract and the set of available models can be extended to add more simulation options e.g. more channel models or modulation schemes.

1.5 Thesis Overview

During the investigation of LDPC codes a number of topics were covered. An overview of the main points is covered here.

1.5.1 Existing Literature

FEC codes, prior to the emergence of Turbo codes in 1993, used non-iterative decoding algorithms i.e. the codeword bits obtained their final, correct values after a single execution of the relevant decoding algorithm [3]. Turbo codes and other modern codes, including LDPC codes, utilise an iterative decoding process. In the iterative decoding process, the codeword bits undergo multiple iterations of updating their values.

The sum-product algorithm (SPA) [7] generalises algorithms commonly used in the artificial intelligence, digital communications and signal processing communities such as the Viterbi algorithm, the forward/backward algorithm and the Kalman filter [7]. In sections 2.3.1 and 2.3.2 the SPA is covered and use it to establish a link between iterative and non-iterative codes. The SPA operates on a graph created by factorising a complicated global function into the product of simpler factor functions [7]. In non-iterative codes, this graph is cycle free and the SPA gives an exact result. In iterative codes, the graph contains cycles and the SPA gives only an approximate result. The SPA is therefore executed iteratively on graphs with cycles in order to achieve a better approximation.

In section 2.4 we cover the derivation of the iterative LDPC decoding algorithm from the SPA. We start with the general *a posteriori* probability (APP) equation and show how this leads to the general iterative LDPC decoding algorithm.

Various approaches to lessening the computational load of the decoding process are discussed in sections 2.4.1 and 2.4.2. This includes numerical approximations, probability and likelihood formats, as well as likelihood update schedules.

LDPC encoding has a quadratic encoding complexity with respect to codeword length [4]. This is a problem because LDPC codeword length needs to be large in order to achieve good performance [8]. Several general approaches exist to deal with this issue, some of which are covered in section 2.5. In general, these approaches require a specific code structure. Two of the more promising approaches which can be applied to arbitrary LDPC codes, and guarantee linear encoding performance, are also covered.

Quasi-cyclic LDPC (QC-LDPC) codes are a structured subset of LDPC codes. A QC-LDPC code's parity matrix can be divided into equally sized blocks. Each block is either the zero matrix, or a shifted identity matrix. This builds an inherent parallel capability into the code as each bit and parity equation will feature at most once in a block. Furthermore, each block can be represented using a simple rotation module. This is useful for simplifying the connection system, which is usually the largest consumer of hardware real estate in a hardware implementation [1]. QC-LDPC codes are covered in more detail in section 2.6.

Standards implementing QC-LDPC codes usually define multiple such codes, with multiple different code rates and block sizes. It therefore becomes important to have flexible decoders capable of supporting variable code rates and block sizes. It may also be beneficial to have autonomous decoders, capable of detecting a change of code at the encoder. Xia et al. [6]

suggest using the average syndrome APP to select the most likely LDPC code from a set of predefined codes. This technique is fully explained in section 2.7.

1.5.2 System Overview

The aim of this project is to create a software tool concerned with the code generation and performance simulation of a hardware QC-LDPC decoder.

The developed tool is split into two distinct systems: the hardware code generation system and the simulation system.

The code generation system takes a set of QC-LDPC code descriptions as input. These are used to generate various files containing the hardware decoder code. These files can then be compiled and synthesised into a working decoder and run on some target hardware system.

The simulation system likewise requires a set of QC-LDPC codes as input. It uses this code set to simulate various decoder performance properties such as the bit error rate and code misidentification rate across a range of SNR values. This is detailed fully in section 3

1.5.3 Hardware Decoder Design and Code Generation

A simple hardware decoder is developed in section 4. It is capable of supporting multiple codes, with different code rates and block sizes. In doing so, we investigate multiple techniques to reduce the area and increase the speed of the decoder, particularly in the inter-routing network of the decoder, which typically consumes the most resources [1].

We further extend the design by incorporating the technique of Xia et al. [6] at minimal speed loss. This allows the decoder to decide which code of the set is the most likely to be active currently. This lets our decoder design become fully autonomous.

The developed software tool takes a set of QC-LDPC codes, and other user options (e.g. maximum iterations), and outputs hardware code files which can be compiled into the autonomous, flexible decoder. This decoder is then specific to the set of QC-LDPC codes. This tool is elaborated on in section 5.1.

1.5.4 Simulation System

The simulation tool is discussed in section 5.2. It allows the user to specify channel and modulation conditions in addition to the decoder configuration. These are used to provide meaningful feedback to the user about the decoder's expected performance under these conditions. The feedback is provided by means of graphs and includes average bit error rate, average code misidentification rate, as well as bit error and code misidentification rates for each code.

The overall simulation system inputs are designed to be abstract. In our simulations, we implement only Gaussian noise channel models. The abstract nature of the system makes it very easy to add a new channel and other parameter models pertinent to the simulations.

Chapter 2

Literature Review

In this chapter we discuss existing literature pertaining to LDPC codes. We cover the early history of error correction in digital communications and how this led to the development and rise of iterative codes such as LDPC codes. The SPA is explained in section 2.3. We present a simple step-by-step example applying the SPA and show how the LDPC decoding algorithm can be derived from the general SPA. We link iterative and non-iterative codes using the SPA as a common starting point. In section 2.4 we derive common LDPC decoding algorithm approximations and discuss the implications of number formats on decoding complexity. Section 2.5 discusses various attempts at gaining linear encoding complexity, including two methods that manage to guarantee linear complexity for arbitrary LDPC codes. We then focus on structured LDPC codes, called QC-LDPC codes, which are commonly implemented [1]. Finally, we discuss a method proposed in [6], that allows for selecting the most likely code of a set for a received codeword.

2.1 Digital Communications and Error Correction History

A basic communication system requires a sender, a receiver and a means to transport the information from the former to the latter. This transportation medium is commonly known as a channel [3]. The information may become distorted during transport due to interference on the channel. This distortion is usually called noise [3]. Figure 2.1 shows this basic communications system.

Originally, telecommunications systems used analogue signals to convey information [3]. These analogue signals become distorted prior to arriving at the receiver. At the receiver, a signal estimator uses the received noisy signal to provide an approximation of the original signal [3]. Analogue signals, by definition, allow for an infinite variation of possible values. This means that the receiver is never certain of the accuracy of the approximation.

The rise of digital communications over analogue started with the work of Nyquist in 1928 [3]. Nyquist proved that a band-limited signal can be perfectly reconstructed from a finite

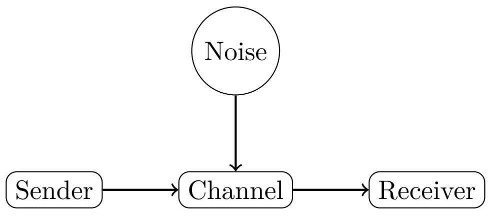


Figure 2.1 – A basic communications system.

set of discrete-time samples of the signal [3]. In 1948 Shannon built on this by proving that these discrete samples could be represented by a finite number of amplitudes [3], dependent on the noise level. Combined, Nyquist and Shannon's work imply that any band-limited signal can be completely described by a finite set of discrete-time digital values even in the presence of noise. This led to the development of digital communications systems.

In a digital system, information needs to be in digital format before it can be sent. In a binary digital system this may require converting the information to a series of 1's and 0's called bits. These bits then get mapped to waveforms which can be transmitted across the channel [2]. This process is called modulation. A simple example is binary phase-shift key (BPSK) modulation. In BPSK a 1 is mapped to $+E_b$ and a 0 to $-E_b$, where E_b is the bit energy [2]. Prior to the development of error correction codes, these modulation schemes were the only way of combating the errors caused by noise [3]. At the receiver, a demodulator takes the received, distorted waveform and makes a hard decision about the bit's value, based on whether a 1 or 0 is more likely. In the case of BPSK a demodulator would map any received waveform above zero to a 1, and below to a 0. An error would therefore occur if noise distorted the waveform to flip around the zero mark. The rate of these errors is directly correlated to the signal-to-noise power ratio (SNR) [2]. As the noise power cannot be controlled, it was believed that only by increasing waveform power, E_b, could one improve error performance [3].

Error correction coding started with another of Shannon's works, namely his famous channel capacity theorem [3]. Shannon proved that arbitrarily small error rates can be achieved for a noisy channel, as long as the bit transmission rate is lower than the channel's capacity [3]. This capacity is often called the Shannon capacity of a channel [2]. Shannon's work proved that smart encoding and decoding of digital signals could drastically improve a communications system's performance without requiring an increase in power [3]. Unfortunately, the proof does not include information on how these codes should be structured to achieve this [3].

The earliest codes could only perform error checking i.e. they could only detect whether or not an error had occurred. A simple example of such a code is the parity bit. In a parity bit code, a single bit is added to the information bits. This bit is called the parity bit and together with the information bits forms the codeword. The parity bit is used to ensure that the codeword as a whole has either an odd number of 1's (odd parity) or an even number of 1's (even parity) – the choice of which parity is arbitrary. This allows the receiver to detect if an odd number of errors occurred as the parity of the codeword would be incorrect.

Hamming and Golay were the first to develop error codes capable of detecting and correcting errors [3]. Hamming codes function by having multiple parity bits mixed in between the information bits. The parity bits are positioned such that, if the bits are indexed starting from 1, they cover every index number whose binary representation contains only a single 1 [2]. The parity bits values are further only calculated using a subset of the codeword bits. A parity bit covers all the bits whose index yield a non-zero result when logically AND'ed with the parity bit's index [2]. Error's in the parity bits are detected as usual – if the overall parity is incorrect, an error has occurred. When an error is detected, adding the positions of all the parity bits that indicated an error will result in the position of the erroneous bit [2]. Hamming codes therefore allow the detection and correction of a single bit error.

Early error correction decoders, such as for the Hamming and Golay codes, exclusively used the hard decision bit outputs of a demodulator. Modern codes often utilise soft-decision decoding which skips the demodulator completely [3]. Each bit is attributed a probability for being a 1 or a 0 based on the received waveform and the channel model. The entire codeword is then analysed using each bit's probabilities to find the most likely codeword [2]. Examples of such codes are the Viterbi codes, convolutional codes and includes iterative codes, such as the LDPC codes we are focussing on.

The driving goal for the field of error correction has always been to get as close as possible to the Shannon capacity of a channel [3]. This would allow for the highest efficiency in terms of bandwidth and power used for effective bit rate [3]. The discovery and adoption of iterative codes such as Turbo and LDPC codes have dramatically narrowed the gap to the Shannon capacity [3].

2.2 LDPC Code History

LDPC codes and their iterative decoding techniques were first proposed in Gallager's 1963 paper [22]. The computational requirements of the codes did not allow them to be implemented at the time (Gallager could only simulate low noise situations using small codeword lengths [22]) and LDPC codes were forgotten.

In 1993 Turbo codes were introduced by Berrou et al. [37]. Turbo codes utilise an iterative decoding method which relies heavily on APP obtained using the method proposed by Bahl et al. [38]. Turbo codes became widely adopted due to their ability to approach the Shannon channel capacity [2] which led to a surge in the research of iterative decoding techniques. In 1996 Gallager's paper [22] and LDPC codes were rediscovered by MacKay et al. [23].

Initially LDPC code performance lagged behind that of Turbo codes, but have recently surpassed them at higher code rates [3]. LDPC codes have since been adopted by many communications standards particularly in the video broadcast [41; 42; 43] and high speed Wi-Fi [44; 45; 47; 13; 46] domains.

2.3 Iterative Codes and the Sum-Product Algorithm

This section discusses the SPA and its application in iterative decoding as used in LDPC codes. It provides a link between iterative and non-iterative decoding and is a summary of the work of Kschischang et al. [7].

The sum-product algorithm[1] is a general theory that allows calculation of marginal values in complex systems [7]. It is a generalisation of many popular probability inference algorithms in the fields of artificial intelligence, statistical modelling and digital communications [7]. Examples include the BCJR forward-backward algorithm [38], the Viterbi algorithm, Kalman filters and, more specifically for this paper, iterative decoding algorithms such as those used by Turbo codes and LDPC codes. Each of these examples employs some version of the sum-product algorithm [7].

The notation used will be similar to that of Kschischang et al. [7] for simplicity. Given a set of variables $\mathbf{x} = \{x_1, ..., x_n\}$ which form part of some global function $G(\mathbf{x})$, then there exist n marginal functions $G_i(x_i)$. A marginal function is defined as

$$G_i(x_i) = \sum_{x_1} ... \sum_{x_{i-1}} \sum_{x_{i+1}} ... \sum_{x_n} G(\mathbf{x}) \tag{2.3.1}$$

where $\sum\limits_{x_j} G(\mathbf{x})$ indicates summation over $G(\mathbf{x})$ for all values of x_j. Note the absence of the variable x_i in the summations of (2.3.1). The marginal is computed by summing over all variations of the global function excluding the variable being marginalised. Such operations will be required often in this writing, and as such a short notation for it is presented as

$$\underset{j}{\overset{\sim i}{\square}}$$

[1]An alternative explanation using the distributive law is available in [26]. The sum-product approach is chosen here as it has closer ties to existing graphical models of LDPC codes.

which implies performing the operation $\square$ for all valid j values except i. (2.3.1) can then be simply rewritten as

$$G_i(x_i) = \sum_{x_j}^{\sim x_i} G(\mathbf{x}) \tag{2.3.2}$$

The purpose of the sum-product algorithm is to efficiently compute marginals, reusing partial sums where possible [7]. It does this by factorising the global function into smaller, local functions which can be represented accurately using a factor graph.

A factor graph is a bipartite graph in which one node set $\mathbf{v} = \{v_{x_1}, ..., v_{x_n}\}$ represents the variables $\{x_1, ..., x_n\}$ and the other node set $\mathbf{f} = \{f_{g_1}, ..., f_{g_m}\}$ represents the factorised local functions $\{g_1(\mathbf{x}_1), ..., g_m(\mathbf{x}_m)\}$ such that

$$G(\mathbf{x}) = \prod_i g_i(\mathbf{x}_i) \qquad \mathbf{x}_i \subseteq \mathbf{x}$$

The factor graph of $G(\mathbf{x})$ contains n variable nodes $\mathbf{v}$ and m function nodes $\mathbf{f}$. An edge is formed between nodes v_{x_i} and f_{g_j} if $x_i \in \mathbf{x}_j$ of the local factor function $g_j(\mathbf{x}_j)$.

The sum-product algorithm only gets exact marginals when applied on a cycle-free factor graph [7]. The next section discusses the execution of the sum-product algorithm on cycle-free graphs, followed by a discussion on graphs containing cycles.

2.3.1 Acyclic Factor Graphs

If a graph is cycle free, every node has at most one path to any other node. This makes it trivial to transform the graph into a tree with an arbitrary node as the root node. A marginal $G_i(x_i)$ is calculated by choosing variable node v_i to be the root node of the tree. Information is exchanged between nodes by passing messages along edges. A message from some node a to node b will be denoted by $Q_{a\text{-}b}$.

Computation starts in the leaf nodes where each variable leaf node passes an identity function message to its parent and each function leaf node passes a description of its function. Each internal node then waits for messages from all of its child nodes to arrive before computing the message to its parent. In such a manner, messages travel up the tree until they reach the root where the final marginal is computed. A variable node v_a computes the message to its parent function node f_b as the product of the messages received from its children i.e.

$$Q_{v_a\text{-}f_b} = \prod_i^{\sim b} Q_{f_i\text{-}v_a} \tag{2.3.3}$$

A function node f_b computes the message to its parent variable node v_a by executing its function on the messages received from its children and then marginalises out its parent variable using (2.3.2) i.e.

$$Q_{f_b\text{-}v_a} = \sum_i^{\sim a} g_b(Q_{v_i\text{-}f_b}) \tag{2.3.4}$$

As one can see in (2.3.3) and (2.3.4), the sum-product algorithm was aptly named after the only operations it requires, namely summation and multiplication. It is also possible to calculate (2.3.3) as

$$Q_{v_a\text{-}f_b} = Q_{f_k\text{-}v_a} \cdot \prod_i^{\sim b,k} Q_{f_i\text{-}v_a} \tag{2.3.5}$$

and (2.3.4) as

$$Q_{f_b\text{-}v_a} = Q_{v_k\text{-}f_b} + \sum_i^{\sim a,k} g_b(\mathbf{x}_b) \tag{2.3.6}$$

which shows the possibility of calculating (2.3.3) and (2.3.4) recursively as

$$\theta(Q_1, ..., Q_n) = \theta(Q_1, \theta(Q_2, ..., Q_n)) \tag{2.3.7}$$

where θ represents the relevant function.

In order to compute all n marginals it is possible to avoid repeating the full computations (and graph restructuring) by employing the full sum-product algorithm [7]. In this algorithm no node is chosen as the root but computation still begins at the leaf nodes. Each vertex now waits until it has received messages from all but one neighbour. It then forms a message in the same manner as before and sends it to this neighbour – essentially treating this neighbour as its parent. It then awaits a return message. Once received it can form messages to the rest of its neighbours, treating each as a parent node in turn. The algorithm terminates once messages have traversed an edge in both directions.

This algorithm works for any system in which multiplication and addition are well defined and the corresponding factor graph is cycle free. Here is an example taken from [27] to illustrate how the sum-product algorithm is expressed as the well known APP algorithm.

Given a sequence $\mathbf{y} = \{y_1, ..., y_n\}$ received from a memoryless channel σ, the APP distribution $G(\mathbf{x})$ for the original codeword symbols $\mathbf{x} = \{x_1, ..., x_n\}$ of some code C is proportional to

$$G(\mathbf{x}) = \sigma_{\mathbf{y}}(\mathbf{x})p(\mathbf{x})$$

where $\sigma_{\mathbf{y}}(\mathbf{x})$ is the channel conditional probability density function for a given $\mathbf{y}$ and $p(\mathbf{x})$ is the *a priori* probability distribution of $\mathbf{x}$. One can factorise $\sigma_{\mathbf{y}}(\mathbf{x})$ as follows because the channel is memoryless:

$$\sigma_{\mathbf{y}}(\mathbf{x}) = \prod_i \sigma_i(x_i)$$

resulting in

$$G(\mathbf{x}) = p(\mathbf{x}) \prod_i \sigma_i(x_i)$$

The factor graph for this general APP distribution function is show in figure 2.2.

If each codeword is equally likely then according to [7] the *a priori* probability distribution can be written as

$$p(\mathbf{x}) = \frac{1}{|C|}\gamma_C$$

where $|C|$ is the number of codewords in C and γ_C the characteristic function of C. This characteristic function can often also be factorised.

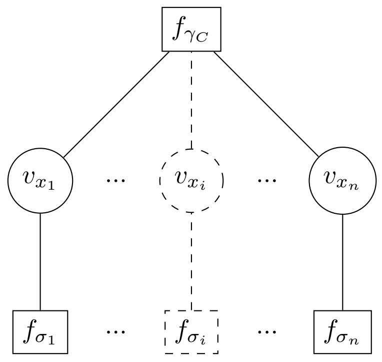


Figure 2.2 – General APP factor graph.

For a code, the characteristic function is simply an indicator function which indicates membership of the code set. For the following binary code

$$C = \{(0,0,0,0),(0,1,1,1),(1,0,1,1),(1,1,0,0)\}$$

the indicator function can be represented as

$$\gamma_C = [x_1 \oplus x_2 = x_3 = x_4]$$

where $\mathbf{x} = \{x_1, x_2, x_3, x_4\}$ are the code bits and $\oplus$ represents the logical XOR operation. This constraint can be split into two simpler functions by making use of an intermediary variable z such that

$$\gamma_1 = [x_1 \oplus x_2 \oplus z = 0]$$

$$\gamma_2 = [x_3 = x_4 = z]$$

and

$$\gamma_C = \gamma_1 \cdot \gamma_2$$

The APP distribution function then becomes

$$G(\mathbf{x}) = \gamma_1 \cdot \gamma_2 \cdot \prod_i \sigma_i(x_i)$$

where the constant $\frac{1}{|C|}$ has been dropped for simplicity. The factor graph representing this is shown in figure 2.3.

Messages in a soft decoding algorithm usually contain two pieces of information, namely the probability that some bit is a zero and the probability that it is a one. Let this information be represented as $Q^0_{a\text{-}b}$ and $Q^1_{a\text{-}b}$ respectively for some message $Q_{a\text{-}b}$.

To illustrate how the message passing works in practice, let the memoryless channel σ be a binary symmetric channel with crossover probability ε and the received codeword $\mathbf{y} = \{0,0,1,0\}$. All messages will be sent as 2-tuples i.e.

$$Q^{\mathrm{D}}_{a\text{-}b} = (Q^0_{a\text{-}b}\ ,\ Q^1_{a\text{-}b})$$

Important to note is that every internal variable node in this graph is of degree two. This implies that messages received on one edge can simply be sent out on the other edge with no computation required [7].

The characteristic sub-function nodes f_{γ_1} and f_{γ_2} operate as follows to marginalise out a variable q:

$$r = (a\ ,\ b) \qquad s = (c\ ,\ d)$$

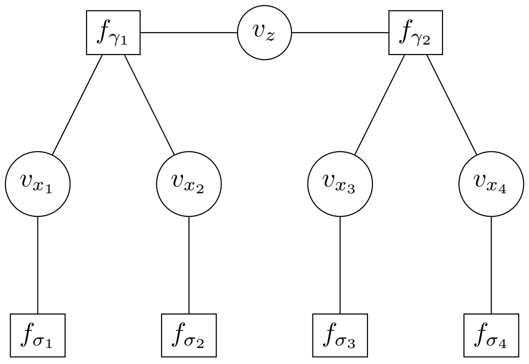


Figure 2.3 – APP example factor graph.

$$\sum_{i}^{\sim q} f_{\gamma_1}(q,r,s) = (a \cdot c + b \cdot d \, , \; a \cdot d + b \cdot c)$$

$$\sum_{i}^{\sim q} f_{\gamma_2}(q,r,s) = (a \cdot c \, , \; b \cdot d) \; \cdot \eta$$

where η is a normalising factor

$$\eta = \frac{1}{a \cdot c + b \cdot d}$$

The algorithm starts in the leaf nodes, namely function nodes f_{σ_i}, which send messages $Q_{f_{\sigma_i}\text{-}v_{x_i}}$ to the codeword symbol variable nodes v_{x_i}. For this example the messages would be

$$\begin{aligned}
Q^{\mathrm{D}}_{f_{\sigma_1}\text{-}v_{x_1}} &= (\Pr(x_1 = 0) \, , \; \Pr(x_1 = 1)) = (1 - \varepsilon \, , \; \varepsilon) \\
Q^{\mathrm{D}}_{f_{\sigma_2}\text{-}v_{x_2}} &= (\Pr(x_2 = 0) \, , \; \Pr(x_2 = 1)) = (1 - \varepsilon \, , \; \varepsilon) \\
Q^{\mathrm{D}}_{f_{\sigma_3}\text{-}v_{x_3}} &= (\Pr(x_3 = 0) \, , \; \Pr(x_3 = 1)) = (\varepsilon \, , \; 1 - \varepsilon) \\
Q^{\mathrm{D}}_{f_{\sigma_4}\text{-}v_{x_4}} &= (\Pr(x_4 = 0) \, , \; \Pr(x_4 = 1)) = (1 - \varepsilon \, , \; \varepsilon)
\end{aligned}$$

This allows the variable nodes v_{x_i} to compute their messages to their respective characteristic sub-function nodes f_{γ_1} and f_{γ_2} as

$$\begin{aligned}
Q^{\mathrm{D}}_{v_{x_1}\text{-}f_{\gamma_1}} &= (1 - \varepsilon \, , \; \varepsilon) \\
Q^{\mathrm{D}}_{v_{x_2}\text{-}f_{\gamma_1}} &= (1 - \varepsilon \, , \; \varepsilon) \\
Q^{\mathrm{D}}_{v_{x_3}\text{-}f_{\gamma_2}} &= (\varepsilon \, , \; 1 - \varepsilon) \\
Q^{\mathrm{D}}_{v_{x_4}\text{-}f_{\gamma_2}} &= (1 - \varepsilon \, , \; \varepsilon)
\end{aligned}$$

These characteristic sub-function nodes then calculate the marginal probabilities messages for variable node v_z as

$$Q^{\mathrm{D}}_{f_{\gamma_1}\text{-}v_z} = \left((1-\varepsilon)^2 + \varepsilon^2 \, , \; 2\varepsilon - 2\varepsilon^2\right)$$

$$Q^{\mathrm{D}}_{f_{\gamma_2}\text{-}v_z} = \left((1-\varepsilon)^2 + \varepsilon^2 \, , \; (1-\varepsilon)^2 + \varepsilon^2\right) \; \cdot \eta_A$$

$$\eta_A = \frac{1}{2(1-\varepsilon)^2 + 2\varepsilon^2}$$

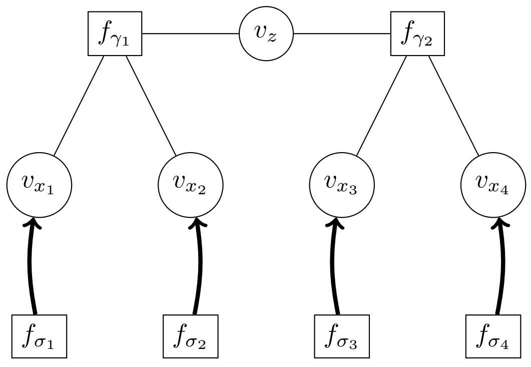


Figure 2.4 – APP example: step 1.

v_z can then immediately send these messages on to the other characteristic sub-function node f_{γ_2}

$$\begin{aligned} Q^{\mathrm{D}}_{v_z\text{-}f_{\gamma_2}} &= Q^{\mathrm{D}}_{f_{\gamma_1}\text{-}v_z} \\ &= \left((1-\varepsilon)^2+\varepsilon^2\ ,\ 2\varepsilon-2\varepsilon^2\right) \end{aligned}$$

and f_{γ_1}

$$\begin{aligned} Q^{\mathrm{D}}_{v_z\text{-}f_{\gamma_1}} &= Q^{\mathrm{D}}_{f_{\gamma_2}\text{-}v_z} \\ &= \left((1-\varepsilon)^2+\varepsilon^2\ ,\ (1-\varepsilon)^2+\varepsilon^2\right)\ \cdot\ \eta_A \\ &= (0.5\ ,\ 0.5) \end{aligned}$$

The characteristic sub-function nodes can now compute their marginals for the variable

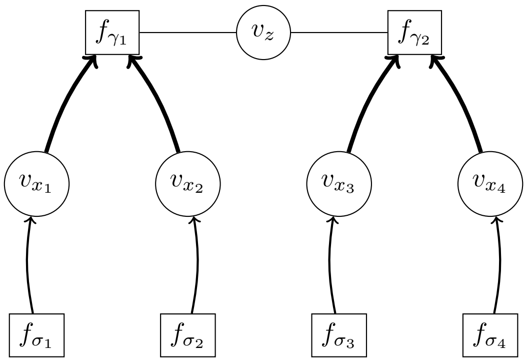


Figure 2.5 – APP example: step 2.

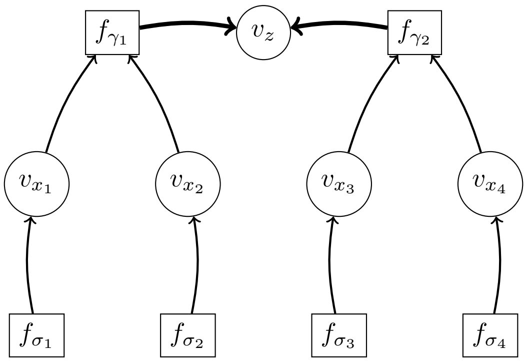


Figure 2.6 – APP example: step 3.

nodes v_{x_i}

$$\begin{aligned}
Q^{\mathrm{D}}_{f_{\gamma_1}\text{-}v_{x_1}} &= (0.5 \cdot (1-\varepsilon) + 0.5 \cdot \varepsilon \ , \ 0.5 \cdot \varepsilon + 0.5 \cdot (1-\varepsilon)) \\
&= (0.5 \ , \ 0.5) \\
Q^{\mathrm{D}}_{f_{\gamma_1}\text{-}v_{x_2}} &= (0.5 \cdot (1-\varepsilon) + 0.5 \cdot \varepsilon \ , \ 0.5 \cdot \varepsilon + 0.5 \cdot (1-\varepsilon)) \\
&= (0.5 \ , \ 0.5) \\
Q^{\mathrm{D}}_{f_{\gamma_2}\text{-}v_{x_3}} &= \big((1-\varepsilon) \cdot ((1-\varepsilon)^2 + \varepsilon^2) \ , \ \varepsilon \cdot (2\varepsilon - 2\varepsilon^2)\big) \ \cdot \eta_B \\
&= \big(1 - 3\varepsilon + 4\varepsilon^2 - 2\varepsilon^3 \ , \ 2\varepsilon^2 - 2\varepsilon^3\big) \ \cdot \eta_B \\
Q^{\mathrm{D}}_{f_{\gamma_2}\text{-}v_{x_4}} &= \big(\varepsilon \cdot ((1-\varepsilon)^2 + \varepsilon^2) \ , \ (1-\varepsilon) \cdot (2\varepsilon - 2\varepsilon^2)\big) \ \cdot \eta_C \\
&= \big(\varepsilon - 2\varepsilon^2 + 2\varepsilon^3 \ , \ 2\varepsilon - 4\varepsilon^2 + 2\varepsilon^3\big) \ \cdot \eta_C \\
\eta_B &= \frac{1}{1 - 3\varepsilon + 6\varepsilon^2 - 4\varepsilon^3} \\
\eta_C &= \frac{1}{2\varepsilon - 6\varepsilon^2 + 4\varepsilon^3}
\end{aligned}$$

Technically, the algorithm continues by passing messages down to the channel conditional probability function nodes f_{σ_i}. This is not necessary in this specific case as these are static functions and cannot change, and also cannot pass messages on further. The APP for each bit x_i can now be computed as the product of all the received messages of variable node v_{x_i}.

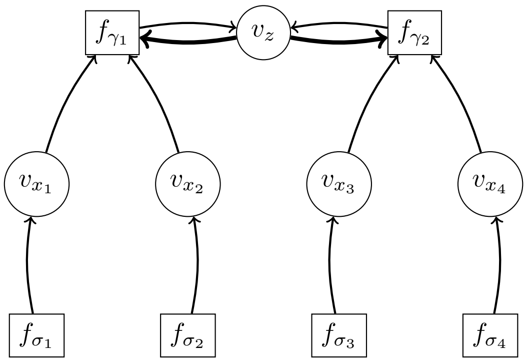


Figure 2.7 – APP example: step 4.

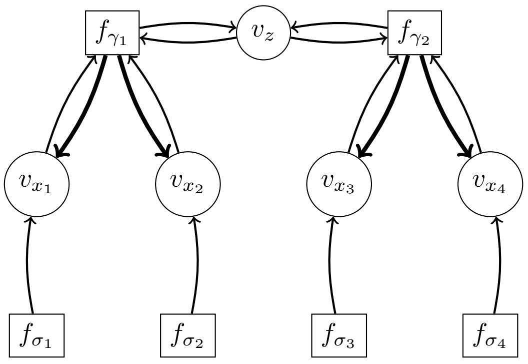


Figure 2.8 – APP example: step 5.

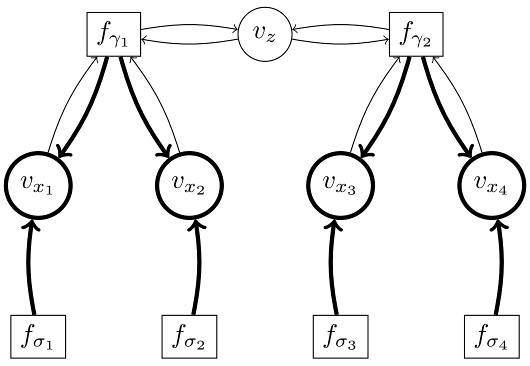


Figure 2.9 – APP example: final calculations at variable nodes.

This gives the following un-normalised probabilities:

$$\begin{aligned}
G_1(x_1) &= (0.5 \cdot (1-\varepsilon)\ ,\ 0.5 \cdot \varepsilon) \\
G_2(x_2) &= (0.5 \cdot (1-\varepsilon)\ ,\ 0.5 \cdot \varepsilon) \\
G_3(x_3) &= \big((1-3\varepsilon+4\varepsilon^2-2\varepsilon^3) \cdot \varepsilon\ ,\ 2\varepsilon-4\varepsilon^2+2\varepsilon^3 \cdot (1-\varepsilon)\big) \cdot \eta_B \\
G_4(x_4) &= \big((\varepsilon-2\varepsilon^2+2\varepsilon^3) \cdot (1-\varepsilon)\ ,\ (2\varepsilon-4\varepsilon^2+2\varepsilon^3) \cdot \varepsilon\big) \cdot \eta_C \\
\eta_B &= \frac{1}{1-3\varepsilon+6\varepsilon^2-4\varepsilon^3} \\
\eta_C &= \frac{1}{2\varepsilon-6\varepsilon^2+4\varepsilon^3}
\end{aligned}$$

For a crossover probability of $\varepsilon = 0.1$ the normalised APP are

$$\begin{aligned}
G_1(x_1) &= (0.9\ ,\ 0.1) \\
G_2(x_2) &= (0.9\ ,\ 0.1) \\
G_3(x_3) &= (0.82\ ,\ 0.18) \\
G_4(x_4) &= (0.82\ ,\ 0.18)
\end{aligned}$$

2.3.2 Cyclic Factor Graphs

The previous section explained the usage of the sum-product algorithm on graphs containing no cycles. The concept for graphs containing cycles and the message forming rules are the same, however the message passing changes to an iterative version. The issue with the cycle-free message passing algorithm is that nodes forming a cycle will never start passing messages. Each node in a cyclic sub-graph has two edges within the sub-graph. This means that every node in the sub-graph is waiting on messages from the other nodes resulting in a deadlock.

This issue is solved by assuming every node receives a unit message on each of its edges at the start of the algorithm [7], allowing every node to compute messages right from the start. If this is done in a cycle-free graph, the message passing will eventually come to a natural halt. In a graph with cycles the message passing never terminates naturally, as a message sent between two nodes in a cycle will propagate through the cycle until it reaches the original sender node, which prompts it to send a new message again, restarting the process. Message passing in cyclic graphs therefore needs some form of halt condition – usually until some maximum number of iterations has been reached or convergence has been determined.

As mentioned previously, the sum-product algorithm only calculates exact solutions when operating on cycle-free factor graphs. When executed iteratively, the sum-product algorithm produces approximate solutions [7]. Despite that, codes using the iterative algorithm such as LDPC and Turbo codes are capable of good error correcting performance. The exact reasons for this are the topic of much research currently [7]

2.4 LDPC Decoding

In this section we delve into the inner workings of LDPC decoding. We start with the definition of an LDPC code and move on to its representation as a factor graph using the SPA. From there we derive the full LDPC decoding algorithm using the SPA as a starting point. We end off by showing how various approximations, message structuring and message schedules can be used to simplify the decoding process.

An LDPC code C is entirely defined by its binary parity matrix $\mathbf{H}$. Such a matrix has dimensions $(m \times n)$ i.e. it has m rows and n columns. Each row represents a parity equation and each column a codeword bit. The LDPC code C therefore has a codeword length of n and contains m parity equations. A parity equation ensures that either an odd (odd parity) or even (even parity) number of ones is present in the codeword bits that are participating in the equation. The parity chosen is irrelevant so long as one is consistent throughout. A codeword bit j participates in parity equation i if

$$H_{i,j} = 1$$

A codeword is only valid if it satisfies every parity equation in the parity matrix i.e. for even parity a codeword $\mathbf{x}$ is valid if it satisfies

$$\mathbf{H} \cdot \boldsymbol{x}^T = \mathbf{0} \tag{2.4.1}$$

where the dot-product is binary i.e. using modulo 2. A code C is therefore made up of all codewords that satisfy (2.4.1).

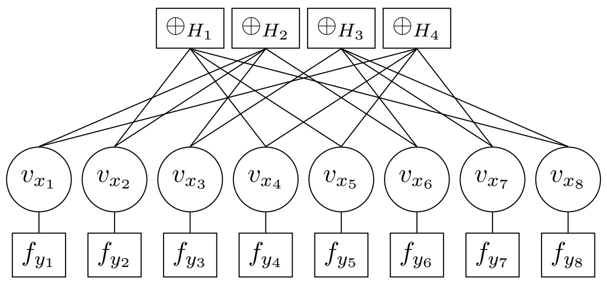
These parity equations equate to the characteristic function for C which allows the calculation of the APP (if the channel model is memoryless) as

$$G(\mathbf{x}) = \frac{1}{|C|} \prod_j f_{\sigma_j}(x_j) \prod_i H_i(\mathbf{x}_i)$$

where $\{H_1(\mathbf{x}_1), ..., H_m(\mathbf{x}_m)\}$ are the m parity equations of $\mathbf{H}$ and $\mathbf{x}_i$ the subset of bit variables participating in parity equation H_i. This leads to factor graphs as seen in figure 2.10a, commonly referred to as Tanner graphs after Tanner [24] first proposed their use in LDPC codes. In Tanner graphs the only variables nodes are nodes representing the codeword bits, $\mathbf{v_x} = \{v_{x_1}, ..., v_{x_n}\}$. Function nodes fall into exactly two types: conditional probability function nodes $f_\sigma = \{f_{\sigma_1}, ..., f_{\sigma_n}\}$ (whose messages are constant during a decoding) and parity-check function nodes $\mathbf{f_H} = \{f_{H_1}, ..., f_{H_m}\}$.

In the APP example of section 2.3.1, all variable nodes were of degree two. This allowed an incoming message on one edge to simply be sent out the other edge with no computation required. For variable nodes of a higher degree this is no longer possible and one needs to follow (2.3.3) to compute messages. For binary decoding this implies calculating the probability that a variable node is a zero or a one. A variable node v_z could compute the probability messages to function node f_j as

$$Q^0_{v_z\text{-}f_j} = \prod_i^{\sim j} Q^0_{f_i\text{-}v_z}$$



(a)

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

(b)

Figure 2.10 – A Tanner graph and its parity matrix.

and

$$Q^1_{v_z\text{-}f_j} = \prod_i^{\sim j} Q^1_{f_i\text{-}v_z}$$

Unfortunately, this allows for cases where $Q^0_{v_z\text{-}f_j} + Q^1_{v_z\text{-}f_j} \neq 1$, which would result in messages not being uniform. This is corrected by adding a normalising factor to ensure all messages have a probability magnitude of one i.e.

$$Q^0_{v_z\text{-}f_j} = \frac{\prod_i^{\sim j} Q^0_{f_i\text{-}v_z}}{\prod_i^{\sim j} Q^0_{f_i\text{-}v_z} + \prod_i^{\sim j} Q^1_{f_i\text{-}v_z}} \tag{2.4.2}$$

and

$$Q^1_{v_z\text{-}f_j} = \frac{\prod_i^{\sim j} Q^1_{f_i\text{-}v_z}}{\prod_i^{\sim j} Q^0_{f_i\text{-}v_z} + \prod_i^{\sim j} Q^1_{f_i\text{-}v_z}} \tag{2.4.3}$$

A parity-check function node was also dealt with in the APP example. It had a degree of three. The calculation of messages from a parity-check function node f_j of arbitrary degree to a variable node v_z can be generalised as

$$Q^0_{f_j\text{-}v_z} = \frac{1}{2} + \frac{1}{2}\prod_i^{\sim z}\left(1 - 2Q^1_{v_i\text{-}f_j}\right) \tag{2.4.4}$$

and

$$Q^1_{f_j\text{-}v_z} = \frac{1}{2} + \frac{1}{2}\prod_i^{\sim z}\left(1 - 2Q^0_{v_i\text{-}f_j}\right) \tag{2.4.5}$$

These equations already result in normalised messages and stem from Gallager's original paper [22] on LDPC codes. He derived them using [22, Lemma 4.1] whereby the probability that a sequence of bits $\mathbf{x}$ contains an even number of symbol $S \in \{0, 1\}$ is equal to

$$\frac{1}{2} + \frac{1}{2}\prod_i (1 - 2\Pr(x_i = S))$$

In practice, a message format such as a 2-tuple requires calculating two separate messages – one for each probability (even if only one is sent; the one requires the other for normalisation), which is inefficient.

2.4.1 LDPC Codes Message Formats

The necessity of sending two separate numbers per message can be avoided by using a likelihood ratio, defined as

$$Q^{\mathrm{LR}}_{a\text{-}b} = \frac{Q^0_{a\text{-}b}}{Q^1_{a\text{-}b}} \tag{2.4.6}$$

This allows the computation and sending of both message pieces as one number. Substituting (2.4.2) and (2.4.3) into (2.4.6), gives the following calculation for the messages from variable node z to function node j

$$\begin{aligned} Q^{\mathrm{LR}}_{v_z\text{-}f_j} &= \frac{Q^0_{v_z\text{-}f_j}}{Q^1_{v_z\text{-}f_j}} \\ &= \frac{\prod_i^{\sim j} Q^0_{f_i\text{-}v_z}}{\cancel{\prod_i^{\sim j} Q^0_{f_i\text{-}v_z} + \prod_i^{\sim j} Q^1_{f_i\text{-}v_z}}} \cdot \frac{\cancel{\prod_i^{\sim j} Q^0_{f_i\text{-}v_z} + \prod_i^{\sim j} Q^1_{f_i\text{-}v_z}}}{\prod_i^{\sim j} Q^1_{f_i\text{-}v_z}} \\ &= \frac{\prod_i^{\sim j} Q^0_{f_i\text{-}v_z}}{\prod_i^{\sim j} Q^1_{f_i\text{-}v_z}} \\ &= \prod_i^{\sim j} Q^{\mathrm{LR}}_{f_i\text{-}v_z} \end{aligned} \tag{2.4.7}$$

where one can see that the normalisation factor cancels out. The message probability pieces $Q^0_{a\text{-}b}$ and $Q^1_{a\text{-}b}$ can be written in terms of the likelihood message $Q^{\mathrm{LR}}_{a\text{-}b}$ by using (2.4.6) and the fact that $Q^0_{a\text{-}b} + Q^1_{a\text{-}b} = 1$. This results in

$$Q^0_{a\text{-}b} = \frac{Q^{\mathrm{LR}}_{a\text{-}b}}{Q^{\mathrm{LR}}_{a\text{-}b} + 1} \tag{2.4.8}$$

and

$$Q^1_{a\text{-}b} = \frac{1}{Q^{\mathrm{LR}}_{a\text{-}b} + 1} \tag{2.4.9}$$

Substituting the parity-check message computation equations (2.4.4) and (2.4.5) into (2.4.6) gives

$$Q^{\mathrm{LR}}_{f_j\text{-}v_z} = \frac{1 + \prod_i^{\sim z}\left(1 - 2Q^1_{v_i\text{-}f_j}\right)}{1 + \prod_i^{\sim z}\left(1 - 2Q^0_{v_i\text{-}f_j}\right)}$$

where $Q^{0}_{v_i\text{-}f_j}$ and $Q^{1}_{v_i\text{-}f_j}$ can be replaced using (2.4.8) and (2.4.9). This gives

$$\begin{aligned} Q^{\mathrm{LR}}_{f_j\text{-}v_z} &= \frac{1+\prod\limits_{i}^{\sim z}\left(1-\frac{2}{Q^{\mathrm{LR}}_{v_i\text{-}f_j}+1}\right)}{1+\prod\limits_{i}^{\sim z}\left(1-\frac{2Q^{\mathrm{LR}}_{v_i\text{-}f_j}}{Q^{\mathrm{LR}}_{v_i\text{-}f_j}+1}\right)} \\ &= \frac{1+\prod\limits_{i}^{\sim z}\frac{Q^{\mathrm{LR}}_{v_i\text{-}f_j}-1}{Q^{\mathrm{LR}}_{i\text{-}j}+1}}{1-\prod\limits_{i}^{\sim z}\frac{Q^{\mathrm{LR}}_{v_i\text{-}f_j}-1}{Q^{\mathrm{LR}}_{v_i\text{-}f_j}+1}} \end{aligned} \tag{2.4.10}$$

The likelihood ratio suffers from an underflow problem when represented using a limited resolution, as probabilities can reach very small numbers. This problem is fixed by using the log-likelihood ratio (LLR) to convey messages from node a to b as

$$Q^{\mathrm{LLR}}_{a\text{-}b} = \ln\frac{Q^{0}_{a\text{-}b}}{Q^{1}_{a\text{-}b}} = \ln Q^{\mathrm{LR}}_{a\text{-}b} \tag{2.4.11}$$

This message format has further advantages, one of which is the ability to determine the most likely symbol of the bit by looking at the sign of the LLR. Another bonus is the computationally friendly variable node message calculation. For some variable node v_z to function node f_j the message calculation becomes

$$\begin{aligned} Q^{\mathrm{LLR}}_{v_z\text{-}f_j} &= \ln\left(\prod_{i}^{\sim j} Q^{\mathrm{LR}}_{f_i\text{-}v_z}\right) \\ &= \sum_{i}^{\sim j}\log\left(Q^{\mathrm{LR}}_{f_i\text{-}v_z}\right) \\ &= \sum_{i}^{\sim j} Q^{\mathrm{LLR}}_{f_i\text{-}v_z} \end{aligned} \tag{2.4.12}$$

where multiplication has now become summation in the log domain. This is of great benefit in real world applications as summation is much cheaper to do computationally. In a similar fashion, the bit node marginal probability can be calculated as

$$Q^{\mathrm{LLR}}_{v_z\text{-}f_=} \sum_{i} Q^{\mathrm{LLR}}_{f_i\text{-}v_z} \tag{2.4.13}$$

On the parity-check node message calculation side, things become more complex. Rearranging (2.4.11) gives

$$Q^{\mathrm{LR}}_{a\text{-}b} = \mathrm{e}^{Q^{\mathrm{LLR}}_{a\text{-}b}}$$

Substituting this into (2.4.10) gives the following equation for a message from f_j to v_z

$$Q^{\mathrm{LLR}}_{f_j\text{-}v_z} = \ln\frac{1+\prod\limits_{i}^{\sim z}\frac{\mathrm{e}^{Q^{\mathrm{LLR}}_{v_i\text{-}f_j}}-1}{\mathrm{e}^{Q^{\mathrm{LLR}}_{v_i\text{-}f_j}}+1}}{1-\prod\limits_{i}^{\sim z}\frac{\mathrm{e}^{Q^{\mathrm{LLR}}_{v_i\text{-}f_j}}-1}{\mathrm{e}^{Q^{\mathrm{LLR}}_{v_i\text{-}f_j}}+1}}$$

This can be simplified using the definitions

$$\tanh\left(\frac{\zeta}{2}\right) = \frac{\mathrm{e}^{\zeta}-1}{\mathrm{e}^{\zeta}+1}$$

$$2\tanh^{-1}(\zeta) = \ln\frac{1+\zeta}{1-\zeta}$$

to get

$$\begin{aligned} Q^{\mathrm{LLR}}_{f_j\text{-}v_z} &= \ln\frac{1+\prod_i^{\sim z}\tanh\left(\frac{Q^{\mathrm{LLR}}_{v_i\text{-}f_j}}{2}\right)}{1-\prod_i^{\sim z}\tanh\left(\frac{Q^{\mathrm{LLR}}_{v_i\text{-}f_j}}{2}\right)} \\ &= 2\tanh^{-1}\left(\prod_i^{\sim z}\tanh\left(\frac{Q^{\mathrm{LLR}}_{v_i\text{-}f_j}}{2}\right)\right) \end{aligned} \tag{2.4.14}$$

Both tanh and $\tanh^{-1}$ are transcendental functions, requiring lookup tables when implemented in hardware, and cause significant computational delays in software implementations. This has led to a range of approximation alternatives being developed.

2.4.2 Parity-Check Message Approximations

The simplest of the approximations is called the min-sum algorithm and suffers a 2 dB performance loss when compared to the full sum-product decoding [1]. The min-sum message computation is defined as

$$Q^{\mathrm{LLR}}_{f_j\text{-}v_z} = \prod_i^{\sim z}\operatorname{sgn}(Q^{\mathrm{LLR}}_{v_i\text{-}f_j}) \cdot \min_i^{\sim z}|Q^{\mathrm{LLR}}_{v_i\text{-}f_j}| \tag{2.4.15}$$

which is equal to the min-sum algorithm used in the Viterbi algorithm [3]. This can be derived by splitting (2.4.14) into smaller recursive pieces (we show this is possible in (2.3.7)) until the smallest piece requires only two messages, Ψ and Ω:

$$\begin{aligned} &2\tanh^{-1}\left(\tanh\left(\frac{\Psi}{2}\right)\tanh\left(\frac{\Omega}{2}\right)\right) \\ &= \ln\frac{1+\tanh\left(\frac{\Psi}{2}\right)\tanh\left(\frac{\Omega}{2}\right)}{1-\tanh\left(\frac{\Psi}{2}\right)\tanh\left(\frac{\Omega}{2}\right)} \\ &= \ln\frac{\cosh\left(\frac{\Psi}{2}\right)\cosh\left(\frac{\Omega}{2}\right)+\sinh\left(\frac{\Psi}{2}\right)\sinh\left(\frac{\Omega}{2}\right)}{\cosh\left(\frac{\Psi}{2}\right)\cosh\left(\frac{\Omega}{2}\right)-\sinh\left(\frac{\Psi}{2}\right)\sinh\left(\frac{\Omega}{2}\right)} \end{aligned}$$

Applying the identities

$$\begin{aligned} \cosh(x+y) &= \cosh(x)\cosh(y)+\sinh(x)\sinh(y) \\ \cosh(x-y) &= \cosh(x)\cosh(y)-\sinh(x)\sinh(y) \end{aligned}$$

gives

$$\begin{aligned} &\ln\frac{\cosh\left(\frac{\Psi+\Omega}{2}\right)}{\cosh\left(\frac{\Psi-\Omega}{2}\right)} \\ &= \ln\left(\mathrm{e}^{\Psi+\Omega}+\mathrm{e}^{-\Psi-\Omega}\right) - \ln\left(\mathrm{e}^{\Psi-\Omega}+\mathrm{e}^{-\Psi+\Omega}\right) \end{aligned}$$

For $|x| \gg 1$

$$\mathrm{e}^{x}+\mathrm{e}^{-x} \approx \mathrm{e}^{|x|}$$

which results in the approximation

$$\begin{aligned} \ln\frac{\cosh\left(\frac{\Psi+\Omega}{2}\right)}{\cosh\left(\frac{\Psi-\Omega}{2}\right)} &\approx \ln\left(\mathrm{e}^{|\Psi+\Omega|}\right) - \ln\left(\mathrm{e}^{|\Psi-\Omega|}\right) \\ &= |\Psi+\Omega| - |\Psi-\Omega| \\ &= \operatorname{sgn}\Psi\operatorname{sgn}\Omega \cdot \min(|\Psi|,|\Omega|) \end{aligned}$$

Finally, by noting that

$$\begin{gathered}\operatorname{sgn}\Phi \cdot \operatorname{sgn}(\operatorname{sgn}\Psi \cdot \operatorname{sgn}\Omega) = \operatorname{sgn}\Phi \cdot \operatorname{sgn}\Psi \cdot \operatorname{sgn}\Omega \\ \min(|\Phi|, \min(|\Psi|, |\Omega|)) = \min(|\Phi|, |\Psi|, |\Omega|)\end{gathered}$$

allows the approximation of (2.4.14) as the min-sum computation by using the recursive nature shown in (2.3.7). The transcendental tanh function and its inverse have now been approximated using only the product of the signum and minimum functions, of which the former reduces to the XOR of the signum function.

A few enhancements to the min-sum algorithm exist. The performance loss of the min-sum algorithm can be largely negated by using a correction factor. The normalised min-sum algorithm does this by multiplying the min-sum result by a positive number smaller than one [1]. The offset min-sum algorithm replaces each incoming message magnitude $|\Psi|$ at a parity-check node by $\max(|\Psi| - \beta, 0)$ [1]. This effectively removes the influence of all messages whose magnitude is less than β. A review of the performance trade-offs is done in [28]. The two min-sum algorithm adaptions can also be combined. In their simplest form, both algorithms' correction factor is a constant, although ideally it would vary with both iteration and node. A few adaptive algorithms that do this have been proposed, an example of which is discussed in [29].

2.4.3 LDPC Message Passing Algorithms

A message passing schedule is often used to add some order to the message passing. General schedule types include:

Flooding Every node sends a message along every edge at the same time.

Serial Nodes send messages along edges one at a time.

Clumping A combination of flooding and serial. Nodes are grouped together, each group then takes turns to pass messages using the flood schedule.

Flooding and serial are trivial types and are not discussed further. LDPC codes have a few ways in which to take advantage of a clumping type schedule.

The standard LDPC message passing algorithm is called two phase message passing [1]. It splits the nodes into two clumps according to their type, namely variable nodes and function nodes. In one phase variable nodes receive messages and calculate their messages to the function nodes. This is known as the variable node update phase. In the other phase, function nodes receive messages and calculate their messages to the variable nodes. This is known as the check update phase, as the channel conditional function nodes don't require receiving messages or any calculation (their messages are a constant).

An extension of the two phase message passing is the layered decoding algorithm [1]. Messages are still passed during variable update and the check update phases. The parity-check function nodes are separated into groups called layers such that every variable node has at most one connection to each layer. During the check update phase only one layer updates its messages, the rest of the groups still pass the old messages. Variable nodes therefore receive at most one new message from parity-check nodes per variable update phase. This allows for simplification of the variable update phase's message calculation. It can also be used to unify the variable and check update phases as described in [1].

2.5 General LDPC Encoding

LDPC codes need long codewords to reach good error correcting performance [23]. This makes encoding complexity with respect to code length an important factor. Unfortunately the encoding of an LDPC code is, in general, quadratic with code length. This can be demonstrated by splitting a codeword $\mathbf{x}$ and the parity-check matrix $\mathbf{H}$ into two parts such that

$$\mathbf{H} \cdot \mathbf{x}^T = \mathbf{0}^T$$

becomes

$$[\mathbf{H}_i|\mathbf{H}_p] \cdot \left[\begin{smallmatrix}\mathbf{x}_i^T \\ \mathbf{x}_p^T\end{smallmatrix}\right] = \mathbf{0}^T \tag{2.5.1}$$

where $\mathbf{x}_i$ and $\mathbf{x}_p$ are vectors containing the information bits and parity bits respectively. As the information bits are already known, all that is required is to calculate the values for the parity bits. This can be done by rearranging (2.5.1) to get

$$\mathbf{x}_p = \mathbf{H}_p^{-1} \cdot \mathbf{H}_i \cdot \mathbf{x}_i$$

Both parts of the parity matrix, $\mathbf{H}_p$ and $\mathbf{H}_i$, are sparse because $\mathbf{H}$ is sparse. This means that the dot product $\mathbf{H}_i \cdot \mathbf{x}_i$ has linear complexity as the sparseness of $\mathbf{H}_i$ can be exploited. Although $\mathbf{H}_p$ is sparse, this does not mean $\mathbf{H}_p^{-1}$ is, which results in quadratic complexity overall.

This encoding complexity has a few solutions, some of which are now examined.

2.5.1 Lookup Table

All encoding can be relegated to performing a simple lookup in a table storing all possible information to codeword combinations. Due to the large code length requirements of LDPC codes, this implementation requires large volumes of memory. It is hardly ever used in practice but should be kept in mind as the cost of memory falls.

2.5.2 Triangular Parity-Check Matrix

If the parity-check matrix can be transformed into a triangular matrix using only row and column operations then it is linearly encodable [3]. For an $(m \times n)$ upper-triangular parity-check matrix, set the first $n - m$ bits as information bits. This allows the calculation of the m parity bits in order by using the parity equations from top to bottom as each equation relies only on information bits and calculated parity bits.

This encoding method can also be expressed using a binary erasure channel decoder as described in [3]. After setting the n information bits and the parity bits as erased bits, the decoder will find the codeword in m iterations. This allows encoder and decoder to share chip real estate which is useful for transceiver and half-duplex systems [3].

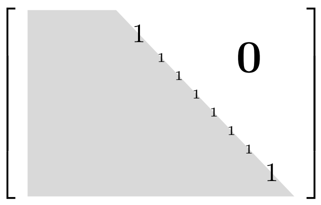


Figure 2.11 – Parity matrix in upper-triangular form.

2.5.3 Approximate Triangular Parity-Check Matrix

Richardson et al. [4] proposed transforming a parity-check matrix as close to a triangular matrix as possible. If a parity matrix can be transformed such that only m' rows do not fall into the triangular matrix form then $m-m'$ parity bits can be calculated using the triangular matrix approach and the other m' parity bit values need to be calculated by solving the remaining m' parity-check equations. This last part has exponential complexity with respect to m'. Richardson et al. [4] show that for randomly constructed LDPC matrices, $m' \ll m$ which allows encoding complexity to be linear with respect to overall code length in most cases.

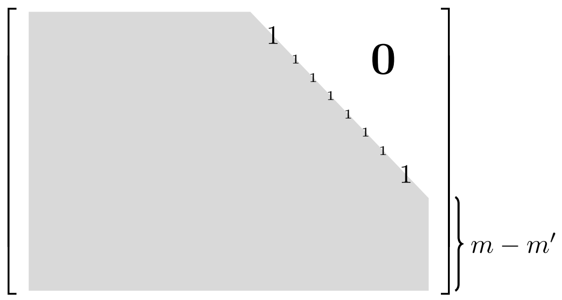


Figure 2.12 – Parity matrix in approximate upper-triangular form.

2.5.4 Block-Triangular parity-check Matrix

This method was recently (2011) proposed as a solution to linearly encode arbitrary p-ary LDPC codes [35]. Although this method is capable of encoding non-binary LDPC codes as well, the focus here is on the binary encoding case only.

The authors of [35] extend the approximate triangular parity-check matrix approach of section 2.5.3 proposed by Richardson et al. [4]. They show that the required parity-check matrix structure can be formed from any parity-check matrix of a linear block code. The authors further show that if the original parity-check matrix is sparse, then the encoding is linear with respect to code length. This means this method can be used to encode arbitrary LDPC codes.

The original parity-check matrix $\mathbf{H}$ is split up into information and parity parts so that

$$\mathbf{H}_p \cdot \mathbf{x}_p^T = \mathbf{H}_i \cdot \mathbf{x}_i^T = \mathbf{b}^T$$

The value of $\mathbf{b}$ can be linearly calculated from $\mathbf{H}_i \cdot \mathbf{x}_i^T$ so long as $\mathbf{H}_i$ is sparse. $\mathbf{H}_p$ is now transformed into block-triangular matrix $\mathbf{A}$ which [35] defines as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{0,0} & \cdots & \cdots & \mathbf{A}_{0,n-m} \\ \mathbf{0} & \mathbf{A}_{1,1} & \cdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \mathbf{0} & \cdots & \mathbf{0} & \mathbf{A}_{m,n-m} \end{bmatrix}$$

where $\mathbf{A}_{i,j}$ is sub-matrix i,j of $\mathbf{A}$ and $\mathbf{0}$ is a zero matrix. Furthermore, the diagonal sub-matrices $\mathbf{A}_{i,i}$ need to have approximate lower triangular structure

$$\mathbf{A}_{i,i} = \left[\begin{array}{cccc|c} \multicolumn{4}{c|}{\mathbf{B}} & \mathbf{C} \\ \hline * & & & & \\ & * & & & \\ \mathbf{0} & & \ddots & & \mathbf{D} \\ & & & * & \end{array}\right]$$

where $*$ represents any non-zero value. $\mathbf{x}'_p$ may now be determined from $\mathbf{A}$ and $\mathbf{b}'$, where $\mathbf{x}'_p$ and $\mathbf{b}'$ are permutations of their respective namesakes in order to match the column permutations in the transformation of $\mathbf{H}_p \rightarrow \mathbf{A}$. This encoding process is directly proportional to the number of non-zeroes in the original parity matrix [35]. If the original parity matrix is sparse as in LDPC codes, then the number of non-zeroes is directly proportional to the block length and therefore encoding is linear.

2.5.5 Generic Graph Based Algorithm

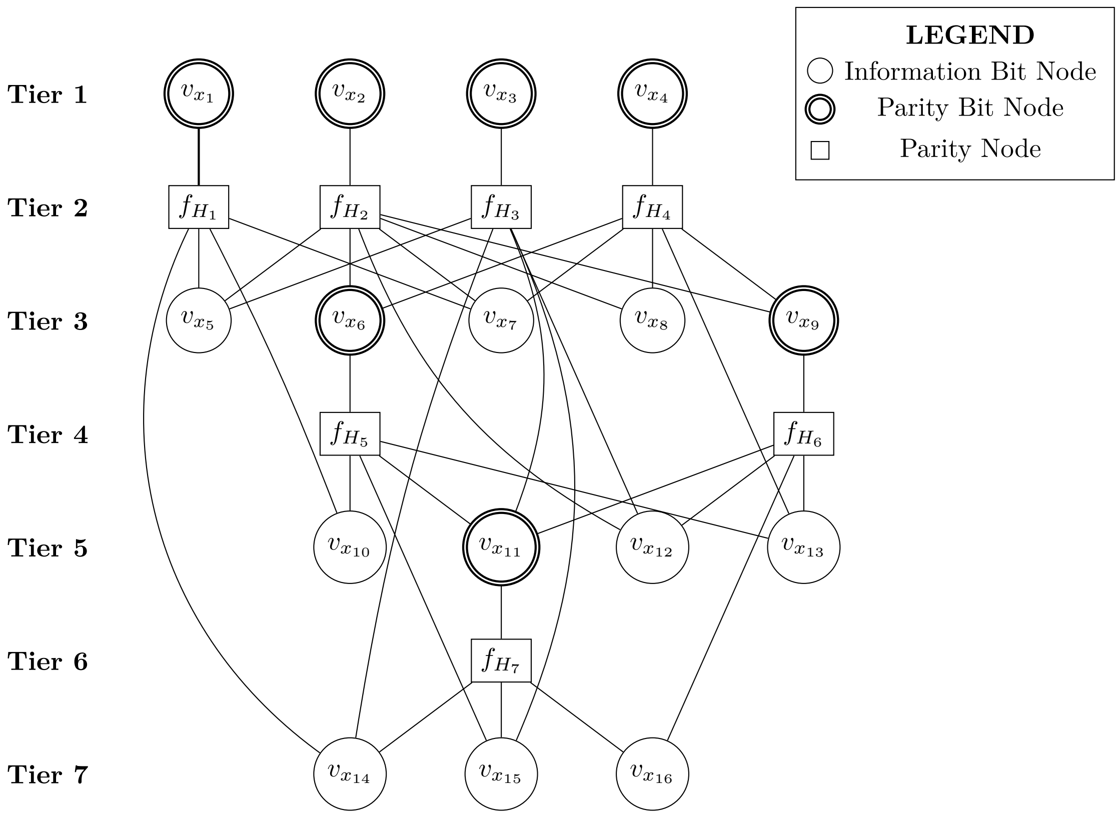
Lu et al. [36] suggests a graph based approach that is similar to that used in the block-triangular parity-check matrix method described in section 2.5.4. To be more exact, the methods are identical for parity-check matrices in which the maximum column weight is less than or equal to three [35] and diverge for higher weights. This graphical approach also only works for binary LDPC codes.

Lu et al. [36] use a structured version of a Tanner graph which they call a *pseudo-tree* [36]. A pseudo-tree has only variable nodes and parity-check nodes. The variable nodes are further subdivided into information bit nodes and parity bit nodes. The nodes are arranged into alternating tiers containing only variable nodes or only parity-check nodes. The structure is further constrained by forcing every parity-check node to have exactly one edge to a higher tier variable node, namely its parent parity bit node. Any node that is not the parent of a parity-check node becomes an information bit node. This structure guarantees linear encoding. Once the information nodes have had their values set, the lowest tier of parity-check nodes can calculate the value of their parity bit nodes. This in turn allows the following tier of parity-check nodes to calculate theirs and so on until the top of the tree is reached. This is the graphical equivalent to the triangular parity-check matrix method described in section 2.5.2. An example of a pseudo-tree is shown in figure 2.13b.

Not all parity-check matrices can be structured as a pseudo-tree. This should be apparent from the fact that not all parity-check matrices can be transformed into a triangular matrix using only row and column operations. Lu et al. [36] circumvent this by extending pseudo-trees into *stopping sets*. A k-fold stopping set is a pseudo-tree plus k extra parity-check nodes, called key check nodes, that cannot fit into the pseudo-tree structure. A possible reason for not fitting into the structure might be that all connected bit nodes are either in lower tiers or already are parity bit nodes of other parity-check nodes thus not letting the extra parity-check node find a suitable parent parity bit node. These key check nodes each need a unique bit node to become their parity bit node. Lu et al. [36] describe an algorithm for finding these k parity bit nodes $\{\beta_1, ..., \beta_k\}$ from the bit nodes in the pseudo-tree. An example of an encoding stopping set is shown in figure 2.14b.

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1
\end{bmatrix}$$

(a) Parity-check matrix.

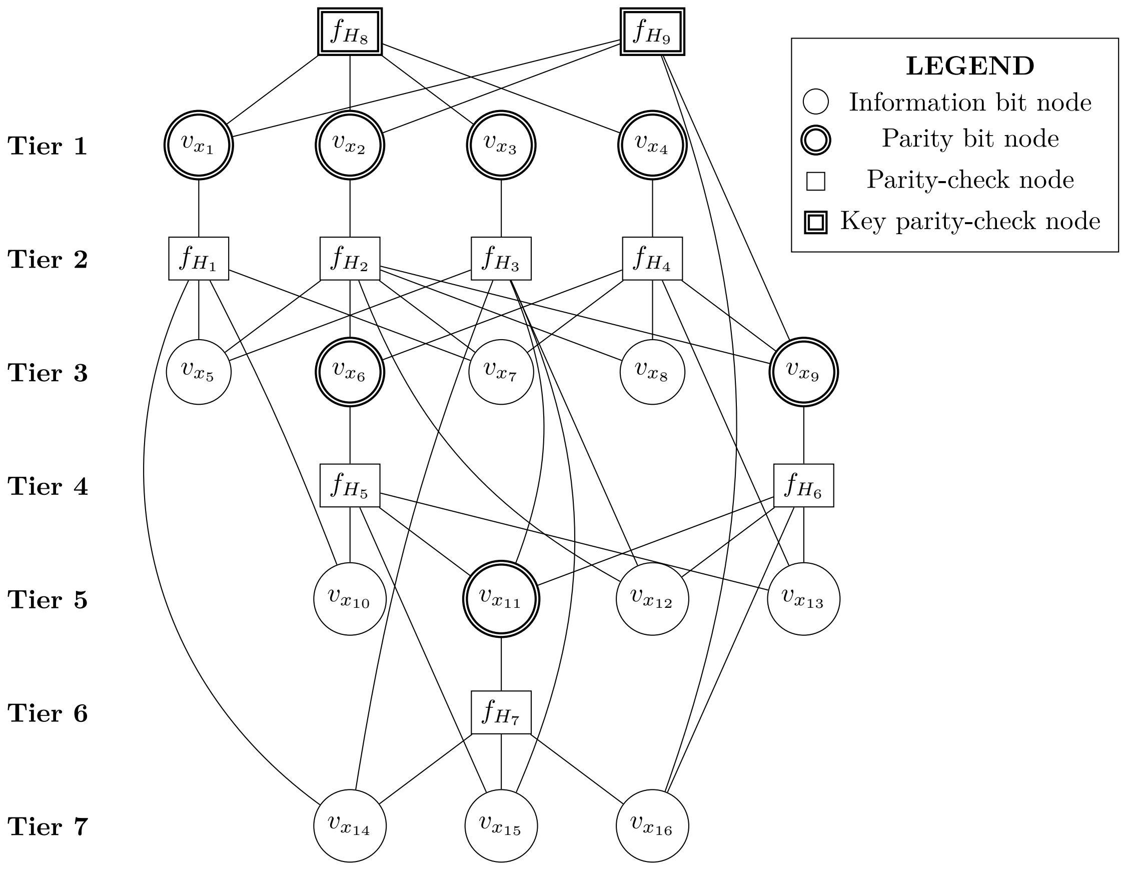


(b) Pseudo-tree.

Figure 2.13 – A parity-check matrix and its pseudo-tree.

$$\begin{bmatrix}
1&0&0&0&1&0&1&0&0&1&0&0&0&1&0&0\\
0&1&0&0&1&1&1&1&1&0&0&1&0&0&0&0\\
0&0&1&0&1&0&0&0&0&0&0&1&0&1&1&0\\
0&0&0&1&0&1&1&1&1&0&0&0&1&0&0&0\\
0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&0\\
0&0&0&0&0&0&0&0&1&1&1&1&1&0&0&1\\
0&0&0&0&0&0&0&0&0&0&1&0&0&1&1&1\\
1&1&1&1&0&0&0&0&0&0&1&0&0&1&1&1\\
1&1&0&0&0&0&0&0&1&0&0&0&0&0&0&1
\end{bmatrix}$$

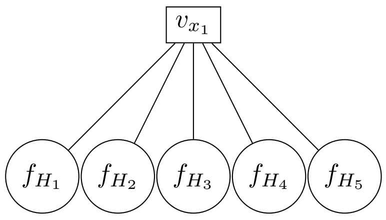
(a) Parity-check matrix.



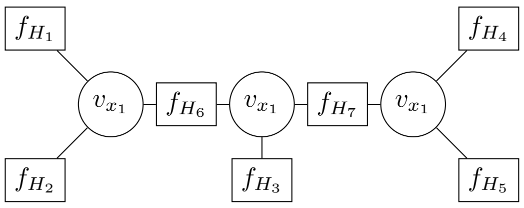
(b) Stopping set graph.

Figure 2.14 – A stopping set graph and its parity matrix.

Encoding starts by encoding the pseudo-tree sub-graph, where $\{\beta_1, ..., \beta_k\}$ are set to arbitrary values and the key check nodes are ignored. After the pseudo-tree has been encoded, some combination of the key check nodes $\{\alpha_1, ..., \alpha_i\}; i \leq k$ might be unsatisfied. This prompts a flip of the associated parity bit nodes $\{\beta_1, ..., \beta_i\}$ which in turn causes certain parity-check nodes in the pseudo-tree to become unsatisfied. This then necessitates the flipping of some parity bit nodes in the pseudo-tree. Lu et al. [36] provide an algorithm to determine which parity bits will be affected by which combination of incorrect key check nodes. This allows for the bulk of the work to be done during preprocessing. Unfortunately this means every parity bit node in the pseudo-tree needs to be aware of whether or not it needs to be flipped for 2^k possible combinations of incorrect key check nodes, which swiftly becomes untenable with the required large code length. Lu et al. [36] solve this by proving that it is possible to restrict $k \leq 2$ by ensuring the maximum bit node degree is three and that the latter is always possible by transforming all bit nodes of degree more than three as shown in figure 2.15.



(a) Bit node of degree 5.



(b) Equivalent graph with maximum bit node degree 3. The extra parity-check nodes ensure that the all bit nodes will have the same value.

Figure 2.15 – Transformation of a bit node of degree 5 into an equivalent graph with maximum bit node degree 3.

2.6 Quasi-cyclic LDPC codes

QC-LDPC codes is a term used to describe a subset of LDPC codes with a specific structure. They are important because of their implementation by many emerging standards, particularly in the wireless communications area [1].

A QC-LDPC code's parity-check matrix $\mathbf{H}$ can be split into $(m^b \times n^b)$ equally sized square submatrices or blocks $\mathbf{H}_{ij}$. Each block is either a cyclic shift of the identity matrix or a zero matrix. Subsequently, their parity-check matrices can be fully described by a block size B and a permutation matrix $\mathbf{\Pi}$ whose elements represent the cyclic shifts or the nil matrix. The nil matrix is usually indicated by a -1 or $-$. The shift direction chosen is arbitrary so long as one is consistent. An example is shown in figure 2.16b.

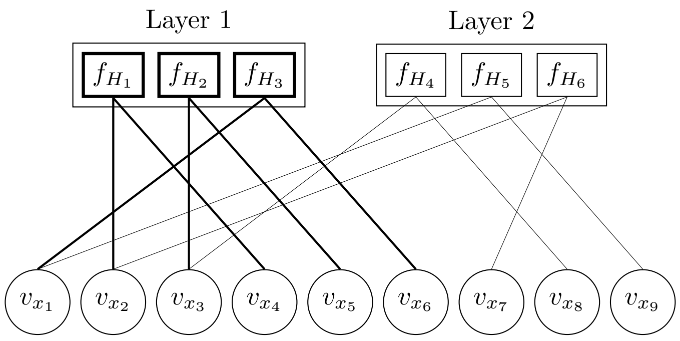
The permutation matrix greatly simplifies the interconnection structure between variable nodes and function nodes in hardware implementations [1]. Instead of remembering or

$$\begin{array}{c|ccc|ccc|ccc|}
 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 \\
H_1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
H_2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
H_3 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
H_4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
H_5 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
H_6 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}$$

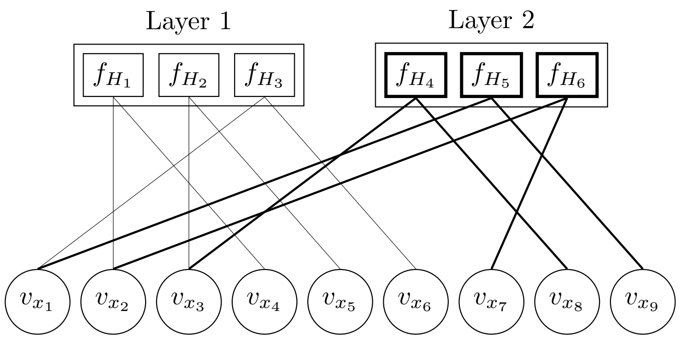
(a) QC-LDPC parity matrix.

$$\mathbf{\Pi} = \begin{bmatrix} 1 & 0 & - \\ 2 & - & 1 \end{bmatrix} \quad B = 3$$

(b) Permutation representation of a.



(c) Tanner graph highlighting layer 1's connections.



(d) Tanner graph highlighting layer 2's connections.

Figure 2.16 – QC-LDPC code example and its layered decoding Tanner graph.

hard-wiring every connection, one can simply rotate each block of incoming messages as determined by the appropriate cyclic shift.

QC-LDPC codes also lend themselves easily to the layered decoding message passing algorithm discussed in section 2.4.3. Each block row of the parity-check matrix naturally represents one layer already. This can easily be seen in the Tanner graphs in figures 2.16c and 2.16d.

2.7 LDPC Code Detection

This section covers the selection of the most likely LDPC code from a set of such codes based on the method proposed by Xia et al. in [6].

Xia et al. propose that the likelihood of a code being responsible for the received codeword **y** is equivalent to the average likelihood of the syndrome APP. That is, the likelihood is equal to the average of the parity-check equation's likelihood of being satisfied. The log-likelihood Γ_θ of some code C_θ is then

$$\Gamma_\theta = \frac{1}{B_\theta} \sum_i \Phi_i^\theta \tag{2.7.1}$$

where m_θ is the number of parity-check equations and Φ_i^θ is the log-likelihood of syndrome i i.e. log-likelihood of parity-check equation i being correct. Xia et al. prove that Φ_i^θ can be written as

$$\Phi_i^\theta = 2\tanh^{-1}\left[\prod_j \tanh\left(\frac{L(x_j|y_j)}{2}\right)\right] \tag{2.7.2}$$

where $L(x_j|y_j)$ represents the log-likelihood of bit x_j being zero based on the conditional channel probability. j iterates over every bit index that is involved in equation parity-check equation i. (2.7.2) should be reminiscent of (2.4.14) which is the message calculation for messages from a parity-check node to a variable node. This similarity will be leveraged in the design of the detection and decoding system.

If a code C_θ is the correct code, then (2.7.2) is expected to yield a positive log-likelihood for each parity equation in C_θ[6]. This leads to an overall high expected value for Γ_θ. If C_θ is not the correct code, then the result of (2.7.2) is random [6], leading to an overall expected log-likelihood of zero for Γ_θ.

The most likely code of a set can be selected by evaluating (2.7.1) for each code and then selecting the code with the largest average syndrome APP. Codes with different codeword lengths can be compared by either cutting off the extra bits (if the received sequence is too long) or by padding with zero bits as required.

The following chapter discusses the requirements and general design of the hardware and software components.

Chapter 3

System Design

In this chapter we broadly outline the design of the system and its subsystems. We go over the aims of the thesis and distil these into system and subsystem requirements. The general concept behind the hardware decoder functionality is analysed. We then discuss the various subsystems and show how these come together to achieve the aims of this thesis.

As mentioned before, the main goal of this thesis is to produce a tool that can generate hardware code for a flexible QC-LDPC decoder. Here, flexible implies that the decoder should be capable of supporting multiple different codes. No restrictions are placed on these codes, except that they need to be QC-LDPC codes. Secondary goals include automating the decoder and providing a software tool that can simulate the applicable performance aspects of the hardware decoder.

The overall system has two outputs:

1. hardware code files for the decoder,
2. simulation results.

A black box model of the overall system is shown in figure 3.1.

We split the overall system into three distinct subsystems, namely configuration parsing, code generation and simulation. The configuration parsing subsystem is somewhat trivial. As the name suggests it simply parses the user's configuration settings and splits these into those relevant to the code generation and simulation subsystems. The configuration parsing subsystem therefore acts as a front end that allows the configuration to remain compact while still keeping the code generation and simulation subsystems distinct from one another. This is visually represented in figure 3.2.

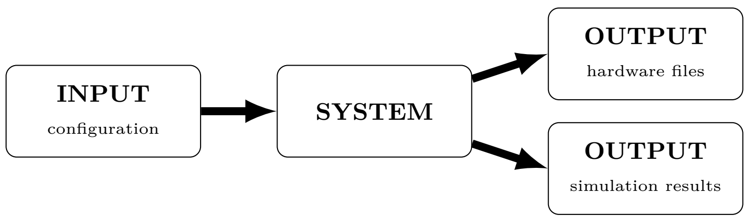


Figure 3.1 – Black box system overview.

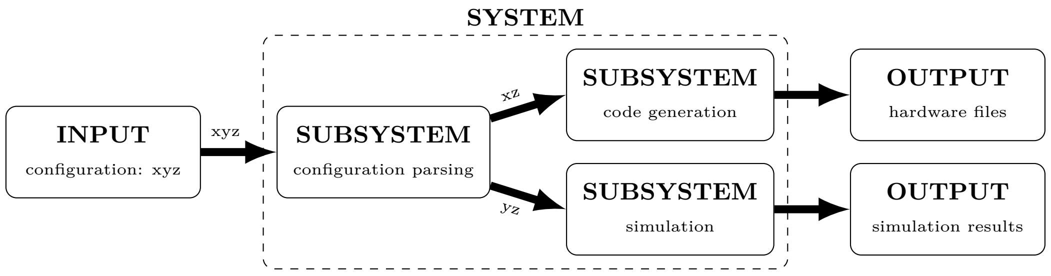


Figure 3.2 – Configuration parsing: x represents settings applicable to code generation only, y to simulation only and z common to both subsystems.

3.1 Decoder Concept

We limit the scope of our decoder by assuming that we have the channel conditional probabilities messages available as input. This means the user is required to deal with converting the received signal into likelihood messages. Computing these likelihoods requires knowledge of the modulation scheme used as well as an accurate model of the channel.

In the case that no prior knowledge of the channel is available, [9] offers a means to estimate the SNR of an additive white Gaussian noise (AWGN) channel when coupled with a BPSK modulation scheme. This in turn allows the user to calculate the necessary channel conditional likelihoods.

The decoder's job is to decode the likelihood messages into codeword bits according to the LDPC decoding algorithm. This is shown in figure 3.3.

In order to do this, the decoder needs to know which code of the code set is currently encoding the data. This is known as the active code. We use the average syndrome APP, as proposed in [6] and discussed in section 2.7, to select the most likely active code. It is unlikely that the active code will change very often and it is therefore inefficient to perform a full detection algorithm for every codeword received. Instead, the average syndrome APP is calculated only for the code currently selected by the decoder. If this APP drops below a user-defined threshold, then a full detection algorithm is performed to select the most likely active code.

The decoder can be viewed as a state machine with three major states:

1. idling,
2. decoding,
3. detecting.

The decoder is idling whenever it is not busy decoding or detecting. A flag is provided to indicate whether or not the decoder is busy. The user therefore knows when it is safe to push new input to the decoder. The decoder moves to the decoding state whenever it receives

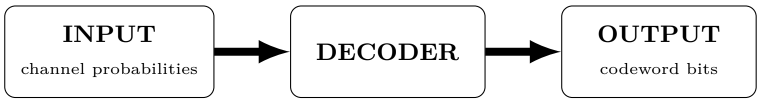


Figure 3.3 – Decoding process.

a new input. This means the decoder can be interrupted and forced to start decoding the new input even if it was not finished with the previous decoding.

The decoder exits the decoding state and returns to the idling state by successfully completing the decoding. If, during decoding, the average syndrome APP falls below the threshold, the decoder enters the detection state.

Once the decoder is in the detection state, it can only transition to the decoding state. This occurs either when the detection algorithm is completed, or when the decoder is forced to start decoding new input. These states and state transitions outline the full functionality of the decoder. The state diagram is shown in figure 3.4.

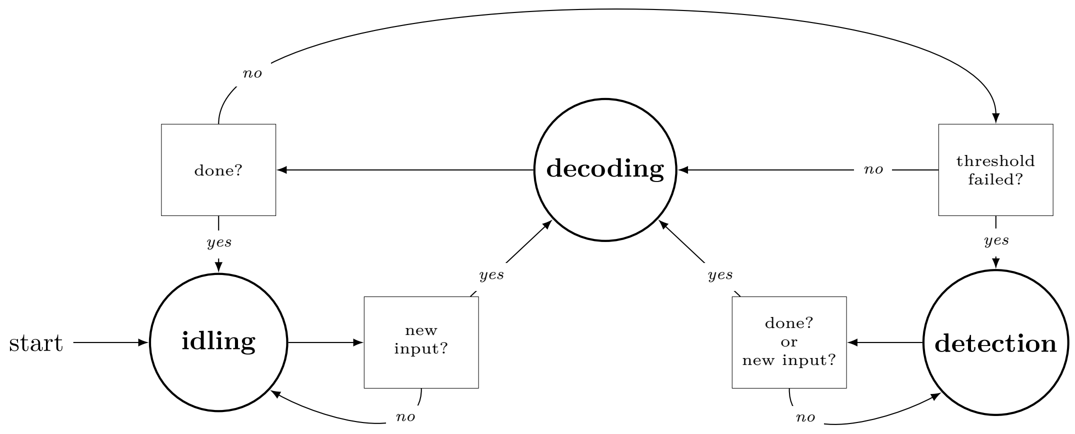


Figure 3.4 – Decoder state machine

3.2 Simulation Subsystem

This subsystem needs to accurately describe and replicate the performance of the hardware decoder using software. It achieves this by providing a software model of the decoder which follows the same decoding and code detection state rules as the hardware decoder. In order to simulate the decoder's performance, input channel conditional likelihoods need to be provided for the software decoder model to operate on. These likelihoods are generated by simulating the entire communications process up to the start of the decoding process. Generating a single codeword's channel conditional likelihoods for some code C requires the following steps.

1. Create a random information bit string of appropriate length for C.
2. Encode this bit string into a valid codeword for C.
3. Modulate the codeword.
4. Simulate transmitting the modulated codeword across a noisy channel.
5. Simulate receiving and demodulating the noisy signal into channel conditional likelihoods.

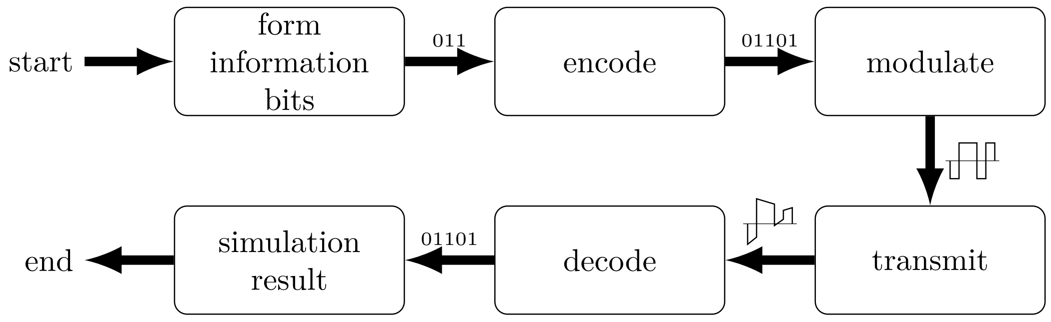


Figure 3.5 – Simulated communications process.

Steps 1 and 2 are achieved by building a software encoder for C. Steps 3, 4 and 5 require that a modulation scheme and channel model be specified. The entire decoder simulation process therefore requires the following to be successful:

- a decoder model for every code in codeset,
- an encoder model for every code in codeset,
- a modulation model or scheme,
- a channel model.

The decoder and encoder models can be created using the code definitions which are supplied by the user. An encoder model can use any of the encoding methods described in section 2.5. The modulation scheme and channel type will vary depending on the situation being simulated. The models for these are therefore kept abstract, allowing new modulation schemes and channel types to be added in as necessary. The simulation subsystem allows the user to select modulation schemes and channel type from among those implemented. This forms part of the configuration settings.

Specific simulations that are possible include: average bit error rate versus SNR, average code identification rate, individual code identification rate and average iterations needed for decoding.

3.3 Code Generation Subsystem

This subsystem is responsible for generating files containing the necessary code for the hardware decoder. The decoder's design pattern is chosen to make this process as easy as possible without sacrificing decoder performance or future extensions. The basic idea is that the decoder's structure is modularised into many highly independent, static parts. These can be easily generated by the subsystem as they are constant. A single dynamically-generated part is used to allow for flexibility in the setup of the decoder. This part is used to control user-defined aspects such as maximum iterations or message bit length as is shown in figure 3.6. The modularisation has a further benefit in that it allows for easy extensibility - parts can be switched out for new or different designs with ease. Each part is written to its own separate file with a logical name to allow for easy readability.

All together, the subsystems, in conjunction with the decoder design, allow us to generate code for a hardware decoder and simulate its performance for various channel and modulation combinations. The decoder is automated by using the average syndrome APPs to decide which code to use in decoding. Our overall system therefore meets all of our aims.

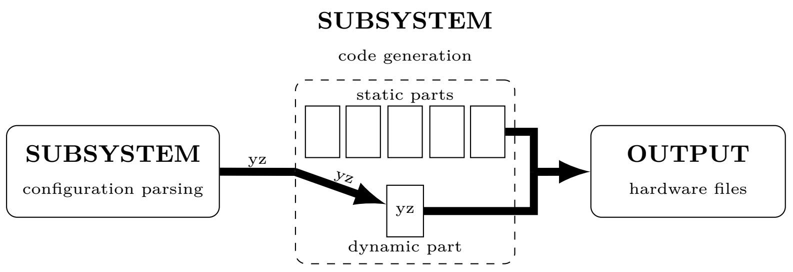


Figure 3.6 – Code generation subsystem.

The full design for both the hardware and software components if elaborated on in the next chapter.

Chapter 4

Hardware Design

In this chapter the design process and choices behind the hardware design for the decoder system is discussed. The main goal of this design is to be a proof-of-concept hardware implementation. As such, the design prioritises simplicity over speed and efficiency. This is the decoder that will be generated by the code generation subsystem and therefore needs to be designed beforehand. We initially discuss some general choices and how they impact our design. We then discuss the decoder modules in detail and show how these form the final decoder design.

The hardware decoder is designed in the VHSIC Hardware Description Language (VHDL). Other target hardware description languages could be implemented to extend the tool. This is left for future work.

4.1 Clock Synchronisation

A hardware design can be asynchronous or synchronous. Both are possible for LDPC decoders. Asynchronous can provide better performance [1] but was deemed unnecessarily complex for this proof-of-concept. The design is split into modules which execute sequentially. Synchronisation between modules is maintained through the use of a common clock. Each module performs its task in a single clock cycle. This allows information to flow from module to module with minimal downtime.

4.2 Message Format

As discussed in section 2.4.1, we have three possible message formats to choose from:

- 2-tuple messages,
- likelihood ratio messages,
- log-likelihood ratio messages.

The dual and likelihood ratio message formats are discarded as they require many multiplications which are expensive to calculate in hardware. We therefore chose the log-likelihood ratio format, which involves only summation and the transcendental functions tanh and $\tanh^{-1}$.

4.3 Transcendental Functions

It is possible to implement transcendental functions in hardware, but the cost of doing so is usually prohibitive [3]. The transcendental functions of interest to us are the tanh function and its inverse, which are used in the detection and parity update equations (2.7.2) and (2.4.14).

These functions can be implemented approximately using lookup tables. This is cheap computationally but quickly becomes expensive in terms of memory, board area and extra wiring in order to achieve higher resolutions and accuracy. Alternatively, they can be approximated using the Taylor series expansion which can save on board area but is slower. Both of these approximations are usually avoided in favour of the min-sum approximation as discussed in section 2.4.2. We use the min-sum approximation in our implementation – the more effective min-sum adaptions could also have been implemented, but this was deemed unnecessary for a proof-of-concept.

4.4 Fixed Point versus Floating Point

Performing arithmetic on floating point numbers is more expensive than for fixed point numbers [10]. Fixed point numbers have a smaller range, however research has shown that good performance can still be attained by fixed point implementations [1]. We therefore use fixed point numbers in our design.

4.5 Sign Magnitude versus Two's Complement Format

Upon examining the min-sum approximation equation (2.4.15) we note that the sign and the magnitude are needed separately. This leads us to examine the use of sign magnitude numbers in place of the more common fixed point two's complement representation.

It was found that the computational gains of the sign magnitude representation at the parity update equation were mostly nullified at the bit update equation as the two's complement representation is easier to add. We decided on the two's complement format as it is directly supported by the VHDL package 'numeric_std'.

We let the user determine the bit length of a message as part of the options available during code generation.

4.6 Message Passing Schedule

Most of the complexity of an LDPC decoder in hardware stems from the module connecting the bit nodes to the parity nodes [11]. In our implementation this module is called the interconnection network. The size and complexity of this network depends entirely on the parity check matrix of the code, as well as the message passing schedule (discussed in section 2.4.3) being used.

At the one extreme, we have the flooding schedule in which every edge in the factor graph is represented by a wired connection, which results in the fastest and most accurate results, but consumes a large portion of hardware real estate. It is further completely inflexible (connections are hard wired) and as such does not afford any benefits to our flexible decoder. At the opposite end, we have the serial schedule where a single message is passed at a time. This would be extremely slow, and not suitable for most applications. The only real option is to use the clumping schedule, where the layered decoding algorithm's schedule stands out for QC-LDPC codes in particular.

In layered decoding, parity nodes are grouped together such that each bit node has at most one connection to each parity node group. QC-LDPC codes are already naturally grouped up this way which means no pre-computation is required. Each bit node has at most one connection to the parity nodes in a block row as the edges are represented by either a shifted identity matrix or a zero matrix. QC-LDPC codes lend themselves so readily to the layered message passing schedule, therefore it is the one we adopted.

4.7 Decoder Design

In this section we go over the detailed design of a decoder such as the one discussed in section 3.1. We briefly outline the decoder modules and their purpose. We then delve into the inner workings of the individual modules and how they interlink. The decoder makes use of all the design choices we have made thus far:

- synchronous clock,
- log-likelihood ratio message format,
- min-sum approximation at the parity nodes,
- fixed point sign-magnitude number representation,
- layered message passing schedule.

Our LDPC decoder has the following modules:

bit module
The bit module is the hardware equivalent of the bit variable nodes. It is responsible for computing the current bit node values as well as the messages to the parity-check nodes.

parity-check module
Similar to the bit module, the parity-check module is equivalent to the parity-check function nodes. It is responsible for calculating the messages for the bit node messages as well as the parity-check APP syndromes used for code detection.

interconnection network module
The interconnection module connects the bit module to the parity-check module and ensures that messages get to the correct node.

detection module
The detection module uses the parity-check APP syndromes to determine which code is the most likely when the decoder is in detection mode. If the decoder is in decoding mode, the detection module uses the parity-check APP syndromes to determine whether or not to enter detection mode, as discussed in section 3.1.

read-only memory module
The read-only memory (ROM) module stores the permutation matrices of the codes, which are used by the interconnection network.

random-access memory module
The random-access memory (RAM) module has three components. Component I stores the channel conditional messages, component II the current bit node values and component III the most recent messages from the parity-check nodes to the bit nodes.

control module
This module controls the state of the decoder. It connects the different modules together and determines when a decoding starts and ends.

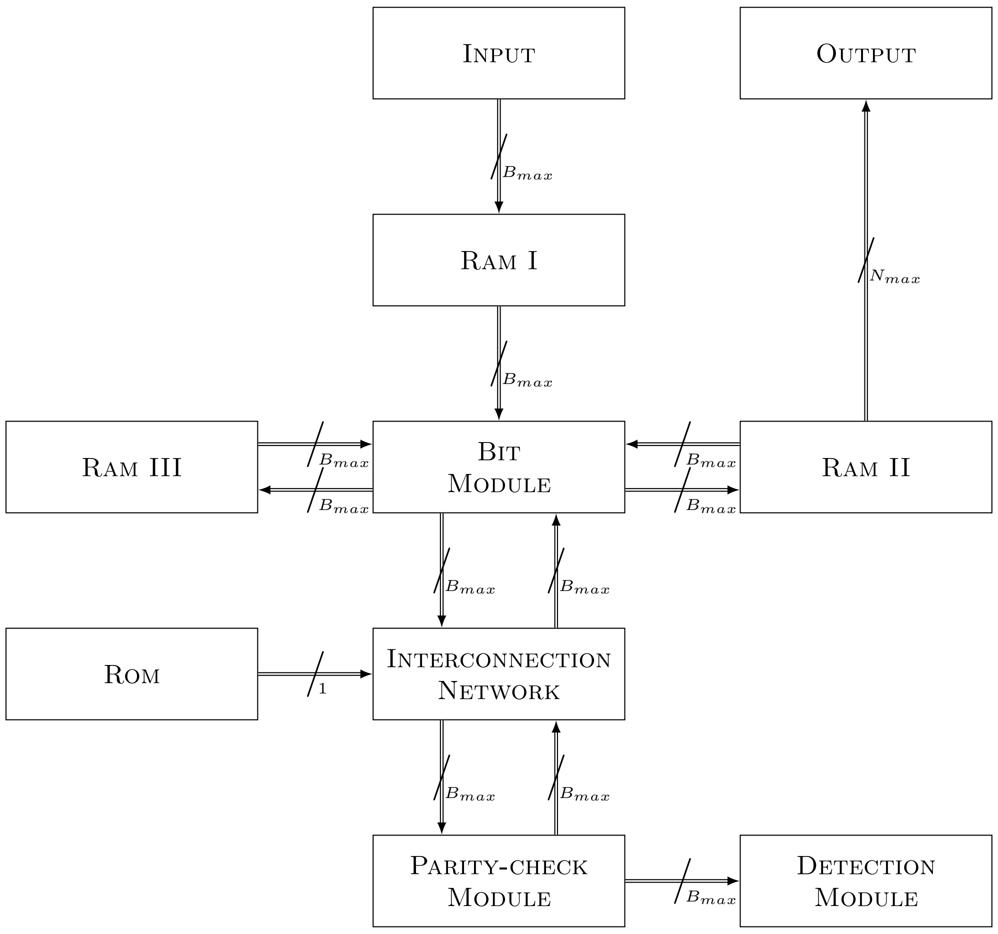


Figure 4.1 – Decoder module connections.

The main connections between the different modules are shown in figure 4.1.

The various decoder modules are now discussed, starting with the interconnection module, as it consumes the largest board area and its design will shape the rest of the modules.

4.7.1 Interconnection Network

The interconnection network is responsible for routing the messages in between the bit and parity check modules according to the permutation values supplied by the ROM module. As we are only supporting QC-LDPC codes we can perform this routing cheaply by rotating each block of messages as per the appropriate permutation. We choose that the rotations should be to the right for messages from the bit module to the parity-check module, and to the left for the returning messages.

For a single code decoder it suffices to implement a simple rotation unit capable of rotating B bits any of the B possible permutations. This can be achieved in hardware using a barrel rotator[1] which rotates an input the desired amount in a single clock cycle. A barrel rotator consists of $s = \lceil \log_2 B \rceil$ stages. Each stage $i \in 0 : s-1$ rotates the message block by 2^i if it

[1]The term barrel shifter is commonly used when referring to both rotation and shift operations. We use the term rotator to differentiate between the rotation and shift.

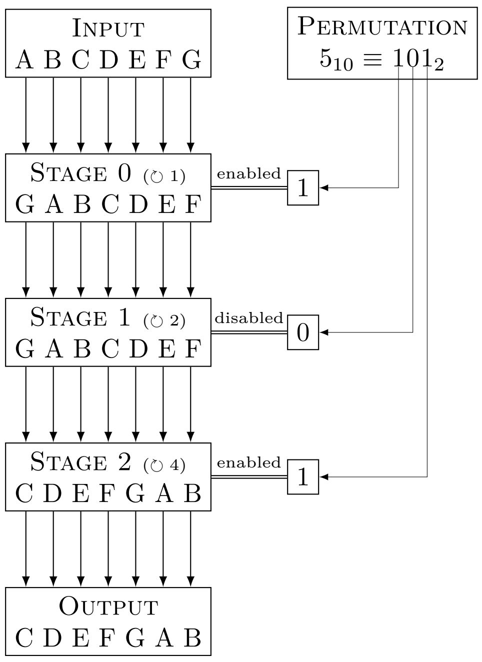


Figure 4.2 – Barrel rotation to the right.

is active. If it is inactive the message block is passed through as is. The control for stage i is simply bit i of the rotation amount. An example is shown in figure 4.2.

Unfortunately, a barrel rotator does not suffice for our uses. The barrel rotator rotates blocks of a fixed size. We need to support multiple different block sizes. One option is to have a barrel rotator for every block size. This is not sustainable, especially for our purpose, as we do not know how many different block sizes we might have.

We need a flexible rotation which has maximum block size B_{max}, input $\mathbf{x} = \{x_0, ..., x_{B_{max}-1}\}$ and output $\mathbf{y} = \{y_0, ..., y_{B_{max}-1}\}$. It is capable of rotating the first $1 \geq B_i \leq B_{max}$ inputs for any valid rotation value $0 \geq \Pi_i \wr B_i$. The flexible rotation therefore maps input to output according to

$$y_j = \begin{cases} x_{(j+\Pi_i) \bmod B_i} & : j \leq B_i \\ * & : \text{otherwise} \end{cases}$$

where $*$ stands for don't care. Two designs for flexible rotation units are proposed in [12] and [11].

Oh et al. [12] propose using a modified Benes network to rotate the first $B \leq B_{max}$ messages of a block of size B_{max}. A Benes network is a series of stages containing parallel 2-input, 2-output switches called crossbar switches [12]. Each switch is controlled to either cross or bar state. In the cross state, the two inputs are switched. In the bar state, the outputs are the inputs. This allows a Benes network to shuffle its input into any conceivable sequence, so long as each input is mapped to an output [12]. Benes networks were commonly used in telephone switchboards [12]. The Benes network is not very efficient with respect to rotating a sub-block as it is designed to be able to output any sequence, even non-rotations. An issue with using a typical Benes network to perform rotations is the large board area that becomes consumed by storing the necessary control signals as the number of rotations and possible block sizes increase. Oh et al. [12] note that one can break a Benes network into two parallel half-sized Benes networks. They utilise this fact to calculate the necessary control signals on the fly, which saves considerable board space.

Xia et al. [11] describe a simple shift network they call QSN[2]. They note that a rotation can be generated by combining the outputs of a left shift and a right shift of the input. This works for rotating any sub-block as well. Tests comparing the improved Benes network and the QSN were done in [11]. They show that QSN is a more efficient implementation for a variable size rotation unit. QSN uses almost a factor 8 smaller board area and its critical path is less than half of the improved Benes network [11]. The QSN is therefore chosen to be used in our interconnection network. We now show how a QSN functions in more detail.

A QSN has three components. A left shift component, a right shift component and merge component. The left and right shift components are simple barrel shifters which operate in parallel. A barrel shifter is identical to the barrel rotator shown in figure 4.2 except that the inputs do not wrap around as they are shifted. The new value that is shifted in is arbitrary which may allow for some small optimisations. Each shifter takes the input and shifts it either left or right. The merge component is then responsible for selecting from either the left or right shifts output to give the correct rotation output. The full functioning of a QSN is outlined in algorithm 1. A graphical example of a QSN is shown in figure 4.3.

The QSNs are used to rotate message blocks in between bit and parity check nodes. A single QSN can rotate a single block at a time. If a set of codes has a maximum number of block columns m_{b-max} then we can rotate an entire layer's worth of messages at a time by using $2 \times m^b_{max}$ QSN units in parallel. We need two QSNs per block, because we require messages to and from every node. The second QSN performs the inverse permutation of the

[2]likely for quasi-cyclic shift network

Algorithm 1 QSN rotate right algorithm.

```
const $B_{max}$                         ▷ maximum block size
in $B$                                  ▷ current block size
in $\Pi$                                ▷ rotation amount
in input                                ▷ input block
var $l, r$                              ▷ left, right shift amount
var $l_o, r_o$                          ▷ left, right shift output
out output                              ▷ output block

if $\Pi == 0$ then
    output $\leftarrow$ *input*
    return output
end if

$l \leftarrow B - \Pi$                  ▷ calculate left shift amount
$r \leftarrow \Pi$                      ▷ calculate right shift amount

$l_o \leftarrow$ shift_left(input, $l$)
$r_o \leftarrow$ shift_right(input, $r$)

for $i < B_{max}$ do
    if $i < \Pi$ then
        output[$i$] $\leftarrow l_o[i]$
    else
        output[$i$] $\leftarrow r_o[i]$
    end if
end for

return output
```

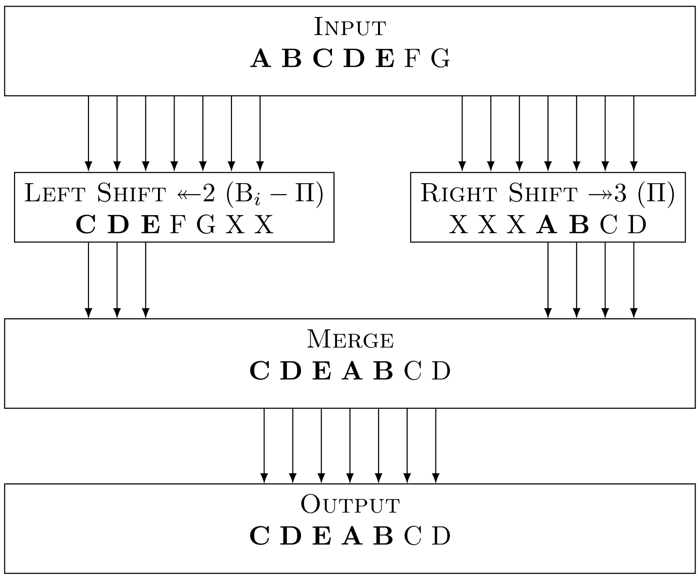


Figure 4.3 – A QSN rotating right with $B_{max} = 7$, $B_i = 5$ and $\Pi = 3$.

first. A fully parallel decoder requires enough bit units to handle a full layer, as well as a parity-check unit that can calculate return messages in a single pass. A decoder prototype following this was designed and tested. A fully parallel interconnection network is shown in figure 4.4. The design worked, however the maximum clock frequency f_{max} possible for the design was very low as the design required many pipelined components which lengthened the critical path. An implementation of this decoder design using the code set defined in the IEEE 802.11n wi-fi standard [13] was generated. This decoder could not fit onto the hardware development board available for testing. It was determined that the fully parallel interconnection network consumed too many resources to be useful in practical applications and a serial design was adopted instead.

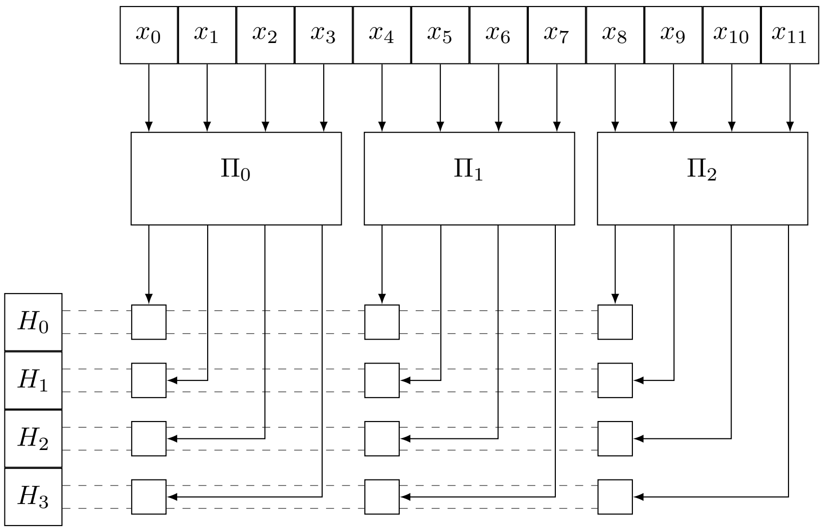


Figure 4.4 – A parallel interconnection network with block size of four.

In the serial design we employ only a single QSN. We are therefore capable of processing only a single block of messages at a time. A second QSN is not needed as one can be used for both message directions. A second QSN would not improve the speed of the decoder. The parity-check module requires all messages to be received prior to calculating return messages. The parity-check and bit modules are therefore never sending and receiving messages simultaneously, which means we require only a single QSN. This QSN is used to permute messages from the bit module to the parity-check module a block at a time. Once all blocks have arrived, the parity-check module calculates the return messages, which the same QSN then de-permutes en-route to the bit module. The QSN uses only a single clock cycle. This can be broken down into multiple clock cycles to increase the maximum clock frequency, however this was deemed unnecessary.

The QSN is controlled via a permutation value gotten from the ROM module. The permutation value indicates either the rotation amount or a value representing the nil matrix. In the case of the nil matrix, the interconnection network needs to pass messages that will not influence the receiver module at all. The interconnection network therefore passes messages with a zero value when a nil matrix occurs and the receiver is the bit module. It sends the maximum positive messages when the receiver is the parity-check module – these represent that the bit is definitely a zero and do not influence the parity equation.

A further advantage of using the serial approach is that the interconnection network does not waste any resources even while decoding codes with less block columns. When using the parallel approach, it is possible that a code has only $m_i^b \leq m_{max}^b$ block columns, which would leave the last $m_{max}^b - m_i^b$ QSNs lying idle. The QSN has been adapted to allow both left and right rotations. This is easily implemented by switching the barrel shifter controls and inverting the merge selection process.

4.7.2 Bit Module

The bit module is responsible for calculating the bit values and messages to the parity-check module using the channel conditional messages and the received parity-check messages. The interconnection network limits the amount of messages the bit module can send or receive at once. The bit module can therefore send or receive only a single block's worth of messages as this is the throughput allowed by a single QSN. We therefore limit the number of bit nodes in the bit module to the block size.

The bit node v_z value and message calculation for parity-check node f_j are provided here for clarity.

$$Q^{\text{LLR}}_{v_z\text{-}f_j} = \sum_{i}^{\sim j} Q^{\text{LLR}}_{f_i\text{-}v_z} \tag{2.4.12}$$

The bit node LLR can be calculated as the sum of its messages as

$$L(v_z) = \sum_{i} Q^{\text{LLR}}_{f_i\text{-}v_z}$$

These calculations can be substantially reduced by noting that only a single received message changes every layer. This allows us to carry a total and simply replace a received message by subtracting the old and adding the new message. A bit node has a maximum of m^b connections to parity-check nodes. We therefore exchange $m^b + 1$ (+1 is for the channel conditional message) additions for a single addition and subtraction. The running total is stored in RAM II and the parity-check messages in RAM III as the bit module can only house a single block's worth.

A bit node unit has two modes, send mode and receive mode. In send mode it calculates and sends a message to a parity-check node. In receive mode it receives a message and calculates the updated value for the bit. The bit module contains B_{max} bit node units, in order to match the interconnection network. During send mode, each block of bit nodes gets loaded in sequence and sends its messages. After the parity-check module has processed the entire layer of messages, the bit module changes to receive mode. The bit node blocks are loaded sequentially again and receive the messages. This completes a single iteration of decoding. If the decoder is in detection mode, the receive mode is skipped.

Send mode has two slightly different sub-modes, initial-pass and secondary-pass. Initial-pass is active during the first pass over every layer, while secondary-pass is active for every subsequent pass. During initial-pass, the previous parity-check messages are ignored when calculating the messages for the parity-check nodes and only the bit value is used. In second-pass, the parity-check messages are subtracted from the the bit value to form the new messages. Initial-pass is necessary when starting a new decoding – the previous messages should be zero however RAM III will still hold messages from the last decoding. A full reset of RAM III would be impractical with respect to time or board area or just impossible. The use of initial- and secondary-pass allows us to bypass this issue. The RAM III holds accurate message values during the subsequent passes because the correct values get written to memory during the previous pass's receive mode. At the start of a decoding the total value is reset to the channel conditional message value gotten from RAM I. This removes the need to add this value to the message and bit value calculations as it is always present in the total.

The working of a bit node during send and receive mode are shown as algorithms 2 and 3 respectively.

Each algorithm is executed in a single clock cycle. Each bit node unit therefore needs to be able to read RAM I and RAM II as well as write to RAM II and RAM III once per clock cycle. RAM has a finite response time, which would slow down the process. This is circumvented by synchronising the RAM to the same clock and requesting the required read data one cycle in advance. The writes can occur one cycle later without consequences to the algorithms as the send and receive modes will be separated by several clock cycles. The bit module consists of B_{max} bit node units which operate in parallel. This allows an entire block of bit nodes to be processed in a single clock cycle.

Algorithm 2 Bit node sending algorithm.

```
const id                                              ▷ bit node ID
in layer                                              ▷ current layer
var msg                                               ▷ message to send
var val                                               ▷ bit value

if initial-pass then
    if layer = 0 then
        val ← access_RAM_I(id)                        ▷ get channel value
    else
        val ← access_RAM_II(id)                       ▷ get bit value
    end if
    msg ← val                                         ▷ no previous message to subtract
else
    val ← access_RAM_II(id)                           ▷ get bit value
    msg ← val – access_RAM_III(layer-1)               ▷ subtract previous message
    set_RAM_II(id, msg)  ▷ update the bit value. new message will be added in receive
end if
send_message(layer, msg)
return
```

Algorithm 3 Bit node receiving algorithm.

```
const id                                    ▷ bit node ID
in layer                                    ▷ current layer
in msg                                      ▷ message received
var val                                     ▷ bit value

val ← access_RAM_II(id)                     ▷ get bit value
val ← val + msg                             ▷ add new message
set_RAM_II(id, msg)                         ▷ update bit value
set_RAM_III(id, layer, msg)                 ▷ update previous message
```

Important to note is that each message has a fixed bit length. We cannot afford to have overflow when calculating the messages as the message can then represent the incorrect bit symbol. A bit value is the sum of at most m^b_{max} messages. The bit value's bit length is made long enough to guarantee that no overflow will occur. This includes values in both the bit unit, as well as the storage in RAM II. When calculating the return message, the result is quantified to the bit length of a message. The message is saturated if overflow occurs.

4.7.3 RAM

RAM II is designed to be be capable of simultaneous read and write operations during the same clock cycle. This is necessary to pipeline multiple bit send algorithms in successive clock cycles. RAM II contains the current bit LLR values, separated into n^b_{max} blocks of B_{max} size. Each bit value has enough bits to store the summation of m^b_{max} messages in order to prevent overflow. This allows it to provide a block's worth of memory in a clock cycle as the memory is contiguous. This is required by the bit module. RAM III does not require simultaneous read and writes as these occur separately in the bit node send and receive modes. It does need to be compartmentalised into blocks similar to the RAM II. We need to store an LLR value for every connection between parity-check nodes and bit nodes. RAM III therefore contains $n^b_{max} \times m^b_{max}$ blocks of B_{max} size where each value is the size of a message. RAM I is very similar to RAM II. It also has simultaneous read and write capabilities but stores only message values. This is because we wish to start decoding while still writing the channel data to memory. RAM I also needs to provide a block of values at a time. Unfortunately, these blocks do not necessarily occur at the same offset for every code. Imagine that RAM I contains a contiguous set of values that is equivalent to the channel

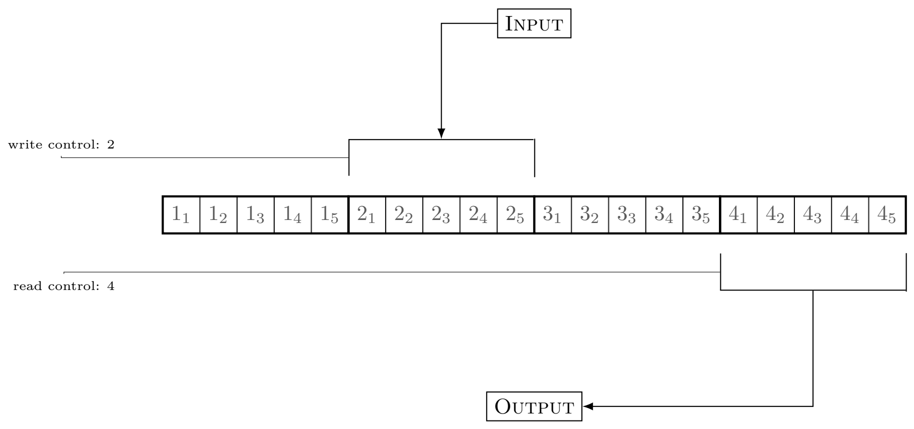


Figure 4.5 – RAM II model.

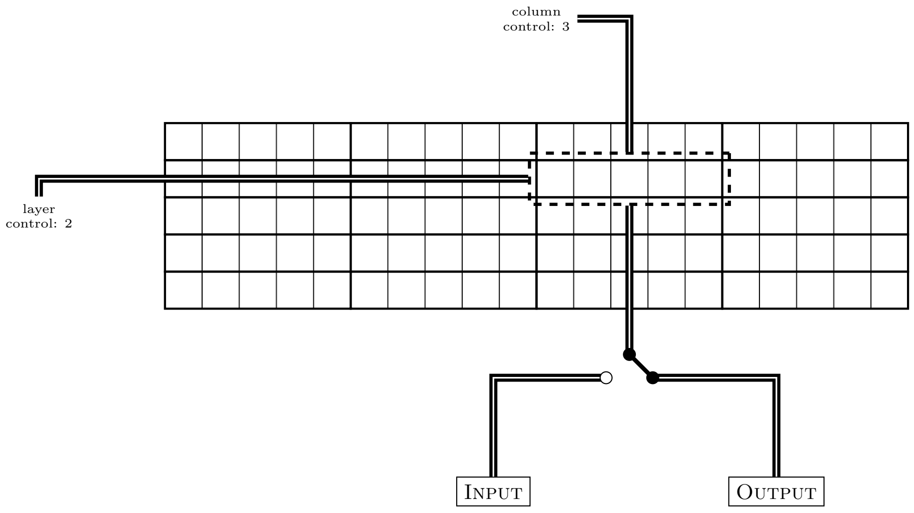


Figure 4.6 – RAM III model.

LLR values given as input. If a code with block size $B < B_{max}$ is active, then the second and successive block offsets will not fall on multiples of B_{max}. We need to access RAM I in offsets of B which can be variable for every code. We still need to provide a contiguous block of memory of size B_{max} (the bit module expects this size), however this means that RAM I needs extra space. Given that we have a maximum codeword length of n_{max} and a code C_θ such that

$$n_\theta^b \times B_{max} > n_{max}$$

then C_θ may exceed the bounds of RAM I when requesting a block of data. This can be visualised by imagining a window of size B_{max} sliding across RAM I's contiguous memory in jumps of size B_θ. When the window gets to its final block it covers the last B_θ values and overshoots the RAM I block by $B_{max} - B_\theta$. This is solved by pre-computing the maximum possible overshoot and appropriately increasing the size of RAM I. These extra pieces of memory are minimal with respect to the total and are never written to. They are only used to guarantee reading complete blocks.

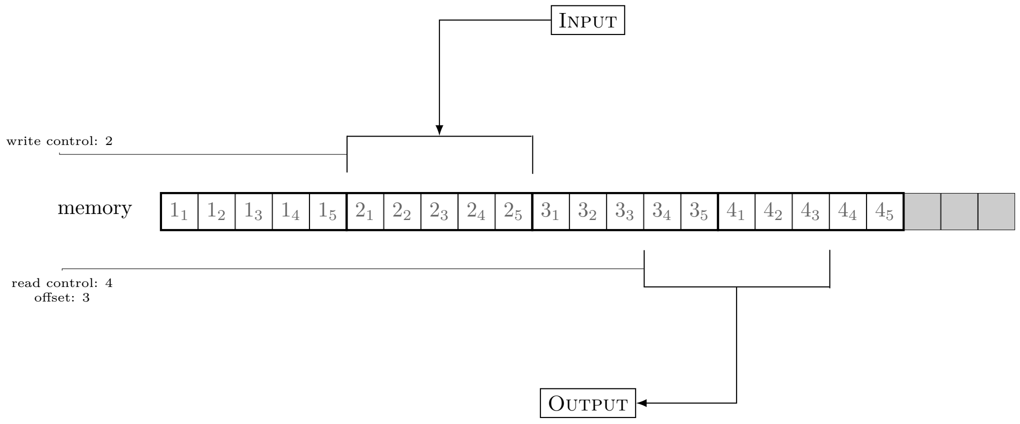


Figure 4.7 – RAM I model.

4.7.4 ROM

The ROM module stores the permutation matrix of every supported code. Each code C_θ has a matrix of permutation values of size $m^b_\theta \times n^b_\theta$. A simpler alternative is to use a cube of size $|\mathbf{C}| \times m^b_{max} \times n^b_{max}$ where $|\mathbf{C}|$ is the total number of codes. This allows for easier access code, but does waste a great deal of memory, as not all codes require $m^b_{max} \times n^b_{max}$ values.

The ROM module outputs a single permutation value as is needed by the interconnection network. The permutation value needs to be able to represent both a zero matrix as well as rotations $0 \leq \Pi < B_{max}$. The common approach is to use a signed integer to represent the permutation value where a negative value represents the zero matrix and a positive the rotation value.

An alternative solution is to find an unused rotation value in the range $\left[0 : 2^{\lceil \log_2 B_{max} \rceil}\right)$. The unused value can then be used to represent the zero matrix. This allows the permutation value to be represented as an unsigned integer which saves a single bit of storage per permutation over the integer solution. The downside is that such an unused value might not exist. This can be countered by simply adding another bit – removing the memory advantage. A further downside is that the test for the zero matrix becomes more complex. The signed integer solution requires checking only the sign bit. The unsigned integer solution requires comparing every bit of the integer – this does however only occur once in the design and might be acceptable. Finding such an unused integer is easy as we are using a software code generator which can pre-compute this for us.

We use the unsigned integer approach as the memory saved outweighs the cost of a more expensive test comparison. For example, in a decoder that supports the IEEE 802.11n wi-fi standard [13], the maximum block size is $B_{max} = 81$. The signed implementation therefore uses eight bits (seven magnitude, one sign) to represent each permutation value and compares a single bit. The unsigned implementation uses seven bits and can definitely use any of the values $[81 : 128)$ to represent the zero matrix. Other unused values may also exist in the $[0 : 81)$ range, but this would need to be checked algorithmically. The unsigned implementation therefore saves one out of eight bits or 12.5% of ROM memory and compares seven bits instead of one when compared to the signed implementation.

4.7.5 Parity-check Module

The parity-check module is responsible for calculating the parity-check node to bit node messages as well as the parity-check syndromes used for code detection. The module receives a single block of B_{max} messages from the interconnection network during the bit node to parity-check node message cycle. The module has B_{max} parity-check units, each representing a parity-check equation. If code C_θ is active then only the first B_θ units are enabled.

A parity-check unit receives a message per clock cycle while it is active, until it has received n^b_θ messages. During these cycles it keeps track of the current minimum, $\min_1$, and second minimum, $\min_2$, message magnitudes as well as the XOR of the message signs, s. It further stores the block index, i_{min}, of the minimum and the incoming message signs. After receiving the n^b_θ messages, it calculates a return message for each received message, one clock cycle per message. The return message Q_r's sign can be calculated by simply XORing the received message's sign with s. The magnitude is set equal to $\min_1$ if $i_r \neq i_{min}$. If $i_r = i_{min}$ then the magnitude is set to $\min_2$. In this way the min-sum parity-check approximation is calculated cheaply. The equation is repeated here for comparison.

$$Q^{\text{LLR}}_{f_j\text{-}v_z} = \prod_i^{\sim z} \operatorname{sgn}(Q^{\text{LLR}}_{v_i\text{-}f_j}) \cdot \min_i^{\sim z} |Q^{\text{LLR}}_{v_i\text{-}f_j}| \qquad (2.4.15)$$

The parity-check syndrome calculation given by [6] is repeated here.

$$\Phi_i^\theta = 2\tanh^{-1}\left[\prod_j \tanh\left(\frac{L(x_j|y_j)}{2}\right)\right] \tag{2.7.2}$$

If we apply the approximations used in the min-sum approximation then we get

$$\Phi_i^\theta = \prod_j \operatorname{sgn}(L(x_j|y_j)) \;\cdot\; \min|L(x_j|y_j)| \tag{4.7.1}$$

Recall that $L(x_j|y_j)$ is the channel conditional LLR. Our decoder gets these as its input. Taking a closer look at message progression through the decoder, we see that for the very first iteration of a decoding: the channel conditional LLRs are passed from RAM I into the bit module, and then used directly as the messages to the parity-check module via the interconnection network. This is verified in algorithm 2. We can therefore calculate (4.7.1) as the XOR of the received message signs and the minimum of the received message magnitudes during the very first iteration. We are already performing this calculation – we have the total sign XOR and first minimum. We pass these to the detection module once they are calculated. As only B_θ units are active at a time, the parity-check module can only guarantee actual values for B_{min} parity-check syndromes.

A parity-check unit requires a single magnitude comparison and XOR while it is receiving messages. During sending (and syndrome calculation), it requires only an index comparison and another XOR. Algorithm 4 describes the functioning of the parity-check module.

4.7.6 Detection Module

The detection module has two purposes. When the decoder is in detection mode, it determines which code is the most likely code and sets it to be the active code. If the decoder is in decoding mode, the detection module calculates a confidence value for the currently active code. If this confidence falls below a predetermined value then the decoder enters detection mode. The detection module uses the parity-check syndromes received from the parity-check module for both purposes.

The detection module receives a block of parity-check syndrome values sporadically. This is because different codes can have a different number of block columns, which determines the clock cycles it takes the parity-check module to calculate the syndromes. The code detection method proposed by [6] uses

$$\Gamma_\theta = \frac{1}{B_\theta}\sum_i \Phi_i^\theta \tag{2.7.1}$$

to calculate the confidence of a code C_θ. We receive all the syndromes Φ_i^θ during a single clock cycle. We can simplify this by removing the normalising factor $\frac{1}{B_\theta}$ which saves an expensive division. Instead of the normalising factor, (2.7.1) is limited to use only the first B_{min} syndrome values. This does result in a loss of performance. The inclusion or exclusion of the normalising factor is left to the user as an option.

During decoding mode, if Γ_θ falls below the threshold, the detection module triggers detection mode for the decoder. Note that Γ_θ is only calculated once for a decoding, during the very first iteration. Once in detection mode, the detection module keeps track of the most likely code. Each code is activated sequentially and performs a single, first iteration of decoding – this allows the detection module to receive each code's syndrome values. Upon calculating the final code's confidence, the most confident code is selected as the active code and decoding is resumed.

Algorithm 4 Parity-check unit execution for a single layer of a code.

```
in $C_\theta$                                   ▷ current active code
var msg                                         ▷ message to send
var $\text{min}_1$                              ▷ minimum magnitude
var $\text{min}_2$                              ▷ second minimum
var $\text{min}_i$                              ▷ minimum index
var sgns                                        ▷ message signs
var sgn                                         ▷ syndrome sign
var $i$                                         ▷ block index

min $\leftarrow$ $max$                          ▷ initialise
$\text{min}_2 \leftarrow max$                   ▷ initialise
sgn $\leftarrow 0$                              ▷ initialise
$i \leftarrow 0$

for $i < m_\theta^b$ do
    msg $\leftarrow$ receive_msg()              ▷ receive next message
    sgns[$i$] $\leftarrow$ msg.sgn              ▷ Store sign
    sgn $\leftarrow$ sgn $\oplus$ msg.sgn       ▷ XOR all signs

    if msg.mag $< \text{min}_1$ then
        $\text{min}_2 \leftarrow \text{min}_1$
        $\text{min}_1 \leftarrow$ msg.mag
        $\text{min}_i \leftarrow i$
    else if msg.mag $< \text{min}_2$ then
        $\text{min}_2 \leftarrow$ msg.mag
    end if

    $i \leftarrow i + 1$
end for

if detecting OR iteration $= 0$ then
    send_syndrome(sgn, min)                     ▷ send syndrome to detection module
end if

if detecting then
    return
end if

$i \leftarrow 0$
for $i < m_\theta^b$ do
    if $i \neq \text{min}_i$ then
        send_msg($\text{min}_1, \text{sgn} \oplus \text{sgns}[i]$)
    else
        send_msg($\text{min}_2, \text{sgn} \oplus \text{sgns}[i]$)
    end if
end for
```

The detection module needs to perform B_{min} summation calculations to calculate Γ_θ. This can lower f_{max} for the decoder if done within a single clock cycle. There is no guarantee that there will be B_{min} clock cycles available to calculate Γ_θ. The smallest interval between blocks of parity-check syndromes occurs during the detection mode and is equal to n^b_{min}. The necessary number of summations necessary per clock cycle to perform B_{min} summations over n^b_{min} cycles is therefore calculated. This is all done as part of preprocessing in the code generator. The detection module is represented as algorithm 5

An alternative detection algorithm is also available as an option during code generation. The alternative algorithm totals the number of satisfied parity equations. This is easily achieved by taking the sign of the parity equation's syndrome – a positive syndrome indicates the parity equation was more likely satisfied than not. This algorithm is identical in execution to the previous algorithm, except that this one uses only the sign instead of the entire syndrome. This solution uses fewer resources but also has less performance. A more thorough comparison is done using simulations in chapter 6.

4.7.7 Control Module

The control module connects the other modules together as needed. It stores the current decoder state and determines the next state based on a few internal variables (such as the

Algorithm 5 Detection module using the parity-check syndromes.

```
const threshold                                   ▷ confidence threshold
var $C_i$                                         ▷ active code
var syn                                           ▷ received syndromes
var conf                                          ▷ current confidence
var max                                           ▷ highest confidence
var $C_{max}$                                     ▷ best code

if decoding then
    syn ← receive_syndromes()
    conf ← $\sum_{i=0}^{B_{min}}$ syn[$i$]
    if conf < threshold then
        enter detection mode
    end if
else                                              ▷ detecting..
    $C_i \leftarrow C_0$
    max ← 0
    for $C_i < C_{max}$ do
        syn ← receive_syndromes()
        conf ← $\sum_{i=0}^{B_{min}}$ syn[$i$]
        if conf > max then
            max ← conf
            $C_{max} \leftarrow C_i$
        end if
        $C_i \leftarrow C_{i+1}$
    end for
    set_code($C_{max}$)
end if
```

current iteration) and the other module outputs. The module is furthermore responsible for starting and ending a decoding.

A new decoding is started once a new set of channel conditional LLRs is received. An external new data flag is made available to the channel conditional inputs. If set, the new data flag indicates that a new set of LLRs is available. The control module immediately begins a new decoding regardless of whether the previous one completed or not.

A decoding terminates once the current bit values correspond to a valid codeword or the maximum number of iterations has been reached. The latter is trivial to implement and we focus on the former decoding termination. A bit value's sign corresponds to the same bit symbol i.e. a positive bit value reflects a 0 and negative a 1. A valid codeword $\mathbf{x}$ satisfies (2.4.1). Put into words, a codeword is valid if it satisfies every parity-check equation, which implies that the parity-check syndrome should be positive. During every iteration, our decoder calculates the parity-check equation syndromes for a single block row. We therefore require N^b successive iterations in which every parity-check equation was satisfied. A further caveat is that the bit symbols may not change throughout these N^b iterations. This is because a bit symbol might change even when the parity-check equations are satisfied, which in turn might invalidate some of the previous successful iterations. This might occur if for example a bit value is barely positive, and the bit node receives a new message (to replace the previous message from that parity-check equation) that has the same sign as the previous message, but a lower magnitude. A valid codeword is therefore only detected once N^b successive iterations have both all parity-check equations satisfied, and no bit symbol changes.

4.7.8 Timing Overview

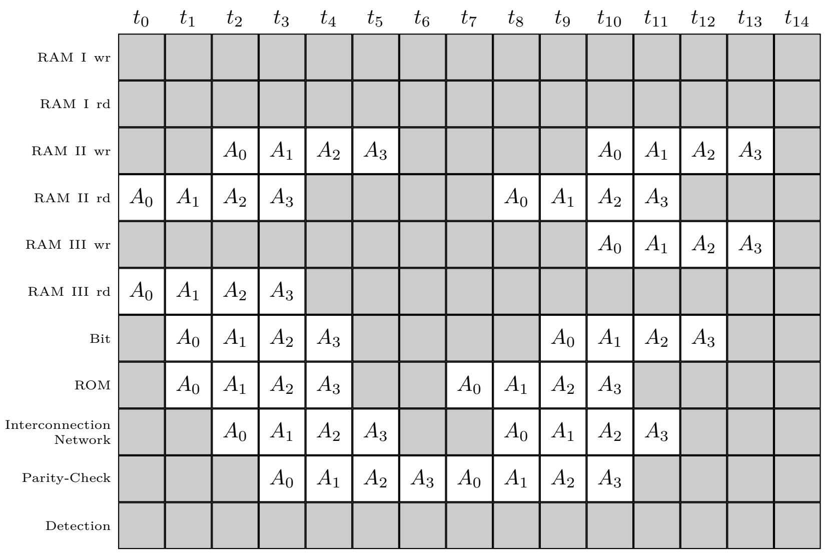
We show various example timing diagrams to showcase the flow of the various information blocks throughout the decoder modules. Figure 4.8 represents the stages for the very first iteration of a decoder, figure 4.9 that of a general decoding for iteration $0 < i < n^b$, figure 4.10 for decoding iterations $i \geq n^b$ and figure 4.11 that for a decoder in detection mode.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
RAM I wr	A_0	A_1	A_2	A_3											
RAM I rd		A_0	A_1	A_2	A_3										
RAM II wr				A_0	A_1	A_2	A_3					A_0	A_1	A_2	A_3
RAM II rd										A_0	A_1	A_2	A_3		
RAM III wr												A_0	A_1	A_2	A_3
RAM III rd															
Bit			A_0	A_1	A_2	A_3					A_0	A_1	A_2	A_3	
ROM			A_0	A_1	A_2	A_3			A_0	A_1	A_2	A_3			
Interconnection Network				A_0	A_1	A_2	A_3			A_0	A_1	A_2	A_3		
Parity-Check					A_0	A_1	A_2	A_3	A_0	A_1	A_2	A_3			
Detection									$A_{\frac{1}{3}}$	$A_{\frac{2}{3}}$	$A_{\frac{3}{3}}$				

Figure 4.8 – Timing diagram for a decoder in decode mode for the very first iteration. The active code C_A has the most block columns $n^b_{max} = 4$. The subscripts represent the block indices and indicate which block each module is busy processing. The detection module subscripts indicate the progress with calculating the code confidence. The detection module has a maximum of three clock cycles to calculate this due to another limiting code C_B with $n^b_B = 3$. The diagram represents the flow of information for the very first iteration of the decoding process.

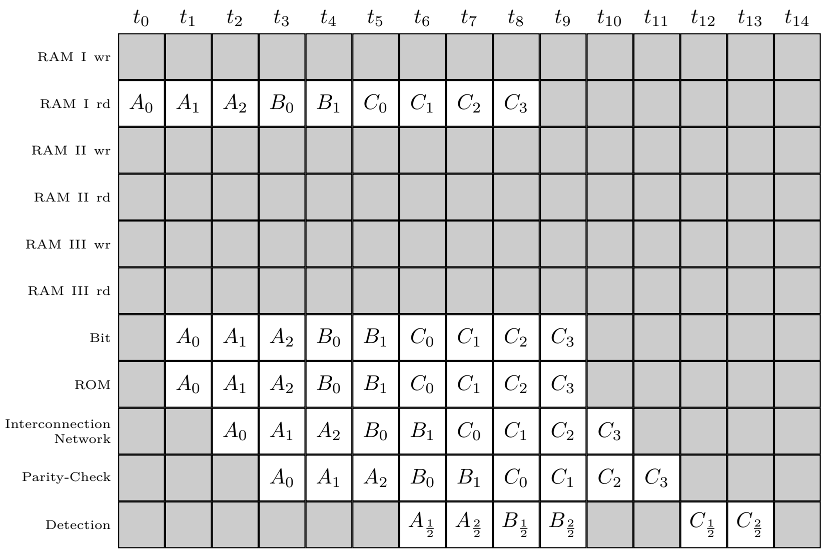
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
RAM I wr															
RAM I rd															
RAM II wr			A_0	A_1	A_2	A_3					A_0	A_1	A_2	A_3	
RAM II rd	A_0	A_1	A_2	A_3					A_0	A_1	A_2	A_3			
RAM III wr											A_0	A_1	A_2	A_3	
RAM III rd															
Bit		A_0	A_1	A_2	A_3					A_0	A_1	A_2	A_3		
ROM		A_0	A_1	A_2	A_3			A_0	A_1	A_2	A_3				
Interconnection Network			A_0	A_1	A_2	A_3			A_0	A_1	A_2	A_3			
Parity-Check				A_0	A_1	A_2	A_3	A_0	A_1	A_2	A_3				
Detection															

Figure 4.9 – Timing diagram for a decoder in decode mode for the iterations following the first iteration but while the bit module is still in the initial-pass phase. The active code C_A has $n^b_A = 4$. The subscripts represent the block indices and indicate which block each module is busy processing. The only change from figure 4.8 is the exclusion of the RAM I wr and rd cycle – it is replaced by the RAM II rd. This is because the channel conditional LLRs only need to be written to RAM I once and RAM II now stores the bit values.



	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
RAM I wr															
RAM I rd															
RAM II wr			A_0	A_1	A_2	A_3					A_0	A_1	A_2	A_3	
RAM II rd	A_0	A_1	A_2	A_3					A_0	A_1	A_2	A_3			
RAM III wr											A_0	A_1	A_2	A_3	
RAM III rd	A_0	A_1	A_2	A_3											
Bit		A_0	A_1	A_2	A_3					A_0	A_1	A_2	A_3		
ROM		A_0	A_1	A_2	A_3			A_0	A_1	A_2	A_3				
Interconnection Network			A_0	A_1	A_2	A_3			A_0	A_1	A_2	A_3			
Parity-Check				A_0	A_1	A_2	A_3	A_0	A_1	A_2	A_3				
Detection															

Figure 4.10 – Timing diagram for a decoder in decode mode for an iteration while the bit module is in the secondary-pass phase. The active code C_A has $n_A^b = 4$. The subscripts represent the block indices and indicate which block each module is busy processing. The change from figure 4.9 is the inclusion of the RAM III rd cycle – previous messages now need to be taken into account.



	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
RAM I wr															
RAM I rd	A_0	A_1	A_2	B_0	B_1	C_0	C_1	C_2	C_3						
RAM II wr															
RAM II rd															
RAM III wr															
RAM III rd															
Bit		A_0	A_1	A_2	B_0	B_1	C_0	C_1	C_2	C_3					
ROM		A_0	A_1	A_2	B_0	B_1	C_0	C_1	C_2	C_3					
Interconnection Network			A_0	A_1	A_2	B_0	B_1	C_0	C_1	C_2	C_3				
Parity-Check				A_0	A_1	A_2	B_0	B_1	C_0	C_1	C_2	C_3			
Detection							$A_{\frac{1}{2}}$	$A_{\frac{2}{2}}$	$B_{\frac{1}{2}}$	$B_{\frac{2}{2}}$			$C_{\frac{1}{2}}$	$C_{\frac{2}{2}}$	

Figure 4.11 – Timing diagram for a decoder in detection mode. The supported codes are C_A, C_B and C_C which have block columns $n_A^b = 3$, $n_B^b = 2$ and $n_C^b = 4$. The letters indicate which code the information is associated with and the subscript the block index. The detection module has at most two clock cycles to calculate a code's confidence because $n_{min}^b = 2$, the subscripts here indicate the completion of this calculation.

Chapter 5

Software Design

This chapter contains the design of the software system. We start with how a user can interact with the system. This is followed by a look at the code generation and simulation subsystems. Finally, the testing environment used to validate the simulation results is discussed.

The software is written in the Go Programming Language[1] (GPL). The software tool is accessible via the command line only. Available options can be set by appending '-option value' to the command line call. Most options have default values and may be omitted. A notable exception to this is the definition of the set of QC-LDPC codes to be used by the tool – these are always required. As discussed in section 2.6, a QC-LDPC code C_θ can be fully described by its permutation matrix $\mathbf{\Pi}_\theta$ and block size B_θ. The tool requires an Extensible Markup Language (XML) file containing the set of codes as input. The XML file should have the general format shown in code snippet 5.1. Each code description may describe the code as either a block size, permutation matrix combination (shown in code snippet 5.2) or as a block size, permutation matrix file name combination (shown in code snippet 5.3) or by using a valid name for a built-in code (shown in code snippet 5.4). A list of all built-in code names can be obtained by running the tool with the '-builtin' option. A permutation matrix file should contain the permutation values using space and new line as column and row separators. A nil permutation should be represented as a '-1'.

[1] http://golang.org/

Code snippet 5.1 – XML code set file.

```
<code>
        <!-- first code description -->
</code>
<code>
        <!-- second code description -->
</code>
...
<code>
        <!-- last code description -->
</code>
```

Code snippet 5.2 – XML block size, permutation matrix code description.

```
<code>
        <size>3</size>
        <matrix>
                1  0 -1
                2 -1  1
        </matrix>
</code>
```

Code snippet 5.3 – XML block size, matrix file code description.

```
<code>
        <size>3</size>
        <file>example-file-path</file>
</code>
```

Code snippet 5.4 – XML built-in code description.

```
<code>
        <name>IEEE 802.11 648 1/2</name>
</code>
```

The code generation subsystem is activated by specifying the '-generate' option. Similarly, the simulation subsystem is activated using the '-simulate' option. Both subsystems may be activated simultaneously. A list of general options common to both subsystems is presented here.

-codes string
: XML file path containing the QC-LDPC codes.

-bits int
: Bit length of the messages.

-iterations int
: Maximum iterations before halting decoding.

-threshold float
: Detection threshold.

5.1 Code Generation Subsystem

The code generation subsystem uses the user-provided set of QC-LDPC codes to generate VHDL files which contain the code for the hardware decoder described in section 3.1. This subsystem has the following extra options available:

-path string
: Target file path location for the generated files.

-detectionNorm bool
: Whether or not to use a normalising factor in the detection algorithm.

-detectionAlg string
: Type of detection algorithm to use. Currently limited to the summation and count algorithms discussed in section 4.7.6. If it is an invalid choice (including blank), a list of available options is displayed.

The code generation subsystem generates the following files:

bit_unit.vhd
: Acts as a single bit node. Calculates bit values and messages to the parity-check nodes.

bit_module.vhd
: Houses a single block's worth of the bit units.

control_module.vhd
: Connects and controls the various modules and their states. Determines when a decoding starts and ends.

decoder_pkg.vhd
: A package containing various constants and type declarations needed by multiple modules.

detection_module.vhd
: Uses the syndromes to determine either the most likely code (if in detection mode) or if the current code falls below the threshold (starting detection mode).

interconnection_network.vhd
: Rotates a block of messages left or right.

parity_unit.vhd
: Acts as a single parity-check node. Calculates the parity-check equation's syndrome as well as messages to the bit nodes.

parity_module.vhd
: Contains a block of parity-check units.

ram_I.vhd
: Stores the channel conditional messages.

ram_II.vhd
: Stores all current bit values.

ram_III.vhd
: Stores all messages received by the bit_units.

rom.vhd
: Stores the permutation matrix of every code.

Each file contains a single module, or entity as it is called in VHDL. A more detailed description of the various modules is available in chapter 4. All of the modules rely on the types and constants declared in the 'decoder_pkg' module. This allows all files to be statically[2] created, except for the 'decoder_pkg.vhd', 'detection_module.vhd' and the 'rom.vhd' files. The 'rom' module stores the permutation matrices and therefore cannot be completely static. The 'detection_module.vhd' contents are dependent on the '-detectionNorm' and '-detectionAlg' options.

[2]Static implying the file contents do not change – this allows them to be represented as a constant string that requires no input.

5.2 Simulation Subsystem

The simulation subsystem is meant to relay relevant information to the user about the hardware decoder's expected performance. This includes information to help the user choose the best options e.g. the ideal message bit length or which detection algorithm to use. The subsystem achieves this by providing performance graphs using variations of the relevant variables. For example, the bit error rate (BER) is shown graphically for a variety of message bit lengths. A list of available simulations is shown below.

BER vs. SNR
: Typical code performance graph. Shows the average BER as well as the individual codes' BER.

BER for message bit length vs. SNR
: Average BER for different message bit lengths.

correct detection rate vs. SNR
: The average correct detection rate when the decoder is in detection mode i.e. how often the correct code is chosen. Includes all versions of the detection type (summation, counting).

codeword detection delay vs. SNR
: Average number of codewords it takes to enter detection mode when the current code is incorrect.

average decode iterations vs. SNR
: Average number of iterations it took to successfully decode.

The subsystem is set up to be as flexible as possible in order to allow for future development. Two features of the GPL play a key role in achieving this, namely *interfaces* and *functions*. Interfaces are GPL's solution for abstraction. An interface is defined as a collection of function prototypes. Any type that implements every function in this collection automatically conforms to this interface and can be used in any place where this interface is used. For example, if an interface A is defined as the collection of methods $c(), d()$ and some function $F(A)$ which requires A as an input, then if some type B implements functions $B.c(), B.d()$ then $F(B)$ is a valid function call. A function is defined in the GPL as shown in code snippet 5.5. where in, T, out represent inputs, variable types and outputs and t refers to the optional variable implementing the function. The code in snippet 5.6 shows the interface example.

Code snippet 5.5 – Go Programming Language function definition.

```
func (t T) name(in₁ T, in₂ T...) (out₁ T, out₂ T...)
```

Code snippet 5.6 – Go Programming Language interface example

```go
// definition of interface A
type A interface {
   c()
   d()
}

// definition of function F(A)
func F(v A) {
   // execute function
}

// definition of type B (similar to C-style structures)
type B struct {
   // variables forming part of type B
}

// define functions c, d for B
func (v B) c() {
   // execute c()
}
func (v B) d() {
   // execute d()
}

// snippet using F(B)
...
   var varB B // declare a variable of type B
   F(varB)    // legal call
...
```

GPL supports the following function related features that enable flexibility:

first class functions
Allows using functions as variables.

higher order functions
Functions can have other functions as input or output.

functions as types
A function can be declared as a type. This is similar to the interface concept, but applies to only a single function. This allows for multiple versions of a single function to exist.

anonymous functions
An anonymous function may be declared within a limited scope, including within another function. This is mostly useful when using a function to return a secondary function. The secondary function can then be defined using inputs to the first function.

In order to simulate meaningful results, channel conditional LLRs are required. This means we need to encode an information bit sequence into a valid codeword for some code C_θ, transmit it across some medium (noisy or otherwise) and receive and convert it to LLR values. An example model of this was shown in figure 3.5. The information bit sequence is generated randomly. Many potential encoding algorithms are detailed in section 2.5.

We allow for any potential encoding method by defining an encoding function type – any encoding method that fulfils this prototype can be used during encoding. Similarly, the transmission and conversion to LLR values is combined into a single function type. This is done because although the transmission itself can usually be modelled as a modulator and a noisy channel, this is not always the case. Not all channel models allow for modulation e.g. a binary symmetric channel. We therefore decided to combine the two. This combination often allows for simplification of calculations and therefore performance gains. The encode and transmission prototypes are shown in code snippet 5.7.

The simulations are all related to either detection or decoding. The simulations are separated into these two types by defining a configuration interface for each. Each configuration needs to provide a set of methods that is required for the simulation. The two interfaces are shown in code snippet 5.8.

This allows a wide variety of different method combinations to be implemented easily. For example, in one simulation the correct detection rates of two nearly identical decoders is compared. The only difference is that one decoder uses the average of the syndromes to calculate a code's confidence (it uses every available syndrome), and the other uses only the

Code snippet 5.7 – Encoding and transmission function prototypes.

```go
// encode function prototype definition
// []bool - an array of boolean values
type Encode(infoBits []bool) (codeword []bool)

// transmission and llr conversion function
// []float64 - an array of 64-bit floating point values
type Transmit(codeword []bool) (llrs []float64)
```

Code snippet 5.8 – Simulation configurations.

```go
// detection simulation configuration
type DetectionSimulationConfig interface {
   // converts channel LLRs to messages
   ToMsgs(llrs []float64) (msgs []float64)

   // calculates parity-to-bit messages and syndromes
   ParityUpdate(msgsIn []float64) (msgsOut, syndromes []float64)

   // calculates a code's confidence from syndromes
   CodeConfidence(syndromes []float64) (confidence float64)
}

// decoder simulation configuration
type DecoderSimulationConfig interface {
   // converts channel LLRs to messages
   ToMsgs(llrs []float64) (msgs []float64)

   // calculates parity-to-bit messages and syndromes
   ParityUpdate(msgsIn []float64) (msgsOut, syndromes []float64)

   // calculates bit-to-parity messages and bit values
   BitUpdate(msgsIn []float64) (msgsOut, values []float64)
}
```

first B_min syndromes and therefore does not require averaging (we use this in the hardware decoder). Both of these 'CodeConfidence' functions are created in the same way. We utilise GPL's higher order functions to create a 'CodeConfidence' function that sum either the first B syndromes if $B > 0$ or uses all available syndromes. A code snippet of this is shown in snippet 5.9.

The simulation subsystem has the following additional options:

-encoder string
: Encoder method name to use for encoding codes. Currently only has one option.

-bitStart int
: A message's least significant bit's (LSB) index. This is mostly used when calculating message values to ensure that the message value falls within the bit range specified by -bitStart and -bits.

-snrStart float
: Start of SNR simulation range in decibels.

-snrEnd float
: End of SRN simulation range in decibels.

-ticks int
: Number of ticks between snrStart and snrEnd.

-simulations int
: Number of simulation trials to run per tick per configuration.

-simIn string
: Name of simulations to include. This option can be repeated to include multiple simulations. If excluded, all simulations are run. Invalid values prompt a list of options.

-simEx string
: Name of simulations to exclude. This option can be repeated to exclude multiple simulations. Invalid values prompt a list of options.

A general simulation run consists of iterating over every code in the provided set, encoding -*simulations* random codewords and using these to simulate the various simulation configurations as applicable.

Code snippet 5.9 – Higher order function example.

```
// higher order function - returns a function
func CreateCodeConfidenceFunc(B int) func([]float64) (float64) {
  if B > 0 {
     ...
     return CodeConfidence(syndromes []float64) (confidence float64) {
        ... // use only B syndromes
     }
  } else {
        ...
     return CodeConfidence(syndromes []float64) (confidence float64) {
        ... // use all syndromes
     }
  }
}
```

5.3 Test Environment

In this section we discuss the testing and validation of the hardware design. The design's performance and functionality are analysed to validate the decoder is operating as expected. The simulation software is also validated against the decoder, to ensure that the software model is sound. The code set used, consisted of all QC-LDPC codes defined in the IEEE 802.11n standard [13]. This set contains twelve codes in total. The standard defines four code rates, $\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{5}{6}$ and three block sizes $B \in \{27, 54, 81\}$. Each code rate is defined once per block size, giving a total of twelve codes. Each code has the same number of block columns, $N^b = N^b_{max} = 24$ which means that the three block sizes correspond to codeword lengths $N \in \{648, 1296, 1944\}$. More details can be found in appendix A. This particular code set was chosen as it is the code set used by [6] from which we got the code detection algorithm. This gives us results to compare against, to ensure the algorithms are correct. In addition, the codes outlined in IEEE 802.11n are trivially encodable using an algorithm described in [14]. The simplicity of this algorithm removes the encoding process as a potential source of programmatic errors.

A DSP Cyclone III (EP3C120F780) FPGA development board was available for hardware testing purposes. This board is manufactured by Altera and we therefore used Altera's Quartus II (v13.sp1 - 64bit) FPGA design software to compile and synthesise the hardware code. Hardware simulations were run using Altera's ModelSim (version 10.1d) software. The development board unfortunately did not have a serial port that we could access. A serial port does exist but it is solely accessible using Altera Intellectual Property (IP) cores. In order to communicate between a computer and the development board it was necessary to add a Altera Nios II processor to the hardware design. The Nios II functions as a limited processor and is capable of running a custom limited-C program. This processor can be augmented with various Altera IPs, including a JTAG serial communications core, which gives us access to the serial port. The Nios II is visible to our hardware decoder as a black box entity with a variety of STD_LOGIC_VECTOR ports, whose number, bit length and signal direction (IN, OUT, INOUT) can be set during the Nios II creation process. The C program running on the Nios II has access to these ports via two C functions:

```
int IORD_ALTERA_AVALON_PIO_DATA(PIO_BASE);
IOWR_ALTERA_AVALON_PIO_DATA(PIO_BASE, int);
```

The former function allows reading an integer from a port, and the latter writing an integer to it. 'PIO_BASE' is the base address of the target port and can be found in the 'system.h' header file which accompanies the Nios II processor. We used six such ports to enable communication between the C program running on the Nios II and our hardware decoder. This communication allows us to control the decoder's state and read it's status. The ports used were:

cmd
5 bit command identifier. Used to tell the decoder which command to execute.

exec
An interrupt flag. On interrupt, execute the current command.

rd
16 bit read data register. Contains the data for all read commands.

wr
16 bit write data register. Contains the data for all write commands.

index
8 bit index for indexing into various arrays during commands.

rdy

A ready flag indicating that the decoder is ready for more commands.

The C program sets the command type, and sets the index and write data if necessary. The C program then sets the exec flag. The decoder executes the command. Once complete the ready flag is set, and any relevant data may be read from the read data register. In this manner, the decoder is controlled by the C program. Key controllable aspects include:

- clock (setting high and low),
- RAM contents (writing channel values, reading bit values and messages),
- decoder mode,
- and iteration number.

A computer can communicate with the C program using the serial port. Altera provides a terminal interface for communicating with the Nios II processor. Interactions include starting, stopping and pausing the processor as well as sending and receiving data on its StdIn, StdOut and StdErr data streams. The Nios II processor is generally started by utilising an Eclipse plug-in that comes bundled with Quartus II. Unfortunately, doing so binds the Nios II terminal interface to Eclipse which does not allow our software access. We circumvent this by starting the Nios II processor from the command line, running the Eclipse plug-in's Nios II start executable. This leaves the Nios II terminal free to be accessed by our software. This enabled communication between our software and the hardware decoder via the C program running on the Nios II hardware processor. A downside to this is that the decoder will not run at full speed. The decoder clock needs to be controlled as an input in order to reliably be able to predict the decoder's state.

Initially, the various hardware modules and software algorithms were tested individually to ensure they perform as expected. For example, a test bench was written that tests each input and rotation permutation for the interconnection network in hardware. The full hardware decoder was then inspected by hand, using ModelSim, to ensure that data is flowing between modules as expected. Test benches were then written and executed in ModelSim, using values provided by the simulation software. For example, to test code detection, realistic channel conditional LLRs were computed and used as input to the hardware decoder. The decoder was then forced into detection mode and the code confidences were compared to those obtained using the simulation software. Similar test benches were run for other decoder states. Channel conditional LLRs were generated using BPSK modulation with AWGN.

The final tests are identical to the previous tests except that the decoder is running on the development board and not being simulated in ModelSim. These tests utilise the direct communication we developed between our software and the hardware decoder via the C program running on the Nios II processor. A test might proceed as follows:

1. The simulation software decoder is initialised to some state e.g. channel conditional LLRs are set, decoder is put into detect mode.
2. The hardware decoder is initialised to the same state via the C program.
3. The software simulated decoder performs some operation e.g. calculates bit messages.
4. The hardware decoder is commanded to perform the same operation via the C program.
5. The hardware decoder's state is read via the C program.

6. The software and hardware decoders' states are compared.

This process may be repeated many times for a single test. The next chapter showcases some simulation results obtained using the simulation tool. We show how these may be leveraged to optimise the hardware decoder's design.

Chapter 6

Simulation Results

In this chapter, we present some simulation results and demonstrate how these can be leveraged to make informed design choices for a hardware decoder. The set of codes used for these simulations are the codes defined in the IEEE 802.11n standard [13]. Details for these codes can be found in appendix A.

We shall be using simulation results to choose the following options for this decoder:

- message bit length,
- message LSB index,
- detection algorithm,
- confidence normalisation,
- maximum iterations.

The LSB index does not play a direct role in the decoder, but determines the resolution we should be using when calculating the channel conditional LLRs. We focus on the detection related simulations, as these are likely novel to the reader.

The first simulation results are shown in figure 6.1. It plots the average correct detection rates of two decoders. The first decoder calculates the parity-check equation syndromes optimally (2.7.2), whereas the second decoder uses the min-sum approximation (4.7.1). Apart from this, both decoders are set up optimally i.e. channel conditional messages are represented using floating point numbers, all parity-check equation syndromes are considered (using all layers), the detection algorithm is optimal (2.7.1) and uses a normalising factor. Somewhat surprisingly, the min-sum approximation outperforms the optimal syndrome calculation decoder by a small margin. The difference is minimal, and may be explained by numerical instability experienced by the transcendental functions found in the optimal calculation.

The next simulation, figure 6.2, shows the average correct detection rates of decoders with a variety of LSB indices. As mentioned at the start of the chapter, this determines the resolution of the channel conditional LLRs used as decoder input. The decoders require a finite message bit length in order to utilise the LSB index. Each decoder's message bit length is set to sixteen, uses the optimal detection algorithm with a normalising factor and considers all syndromes when calculating code confidence. The syndromes are calculated using the min-sum approximation in order to represent the hardware decoder. From the simulation, we see that performance converges from LSB index zero downwards. We decide upon using a LSB index of -1.

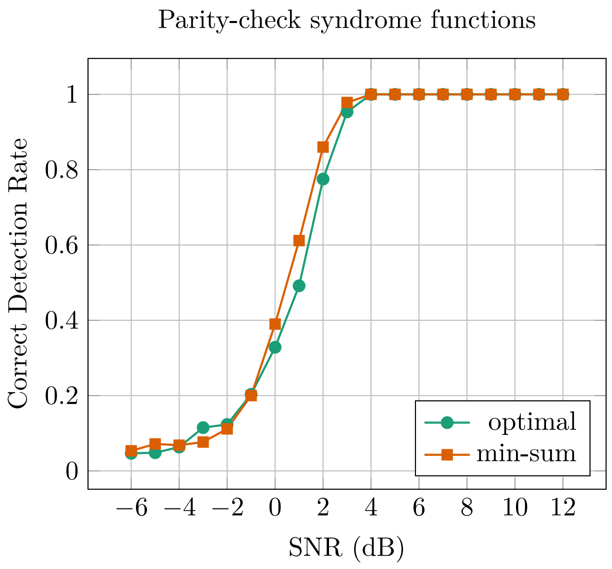


Figure 6.1 – Comparison of parity-check syndrome functions.

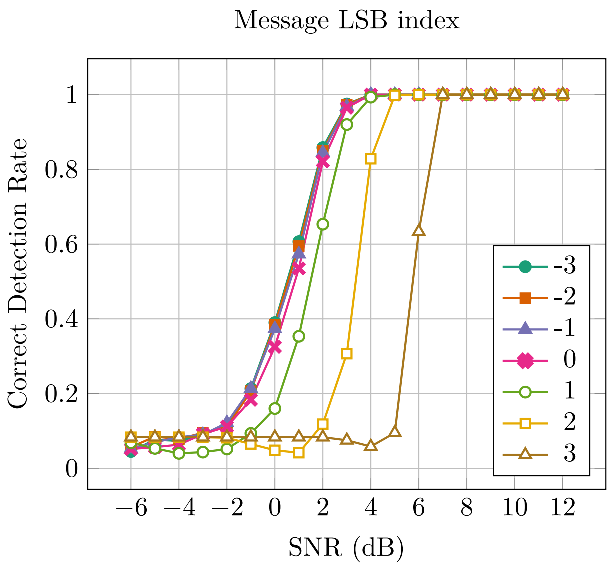


Figure 6.2 – Comparison of message LSB indices.

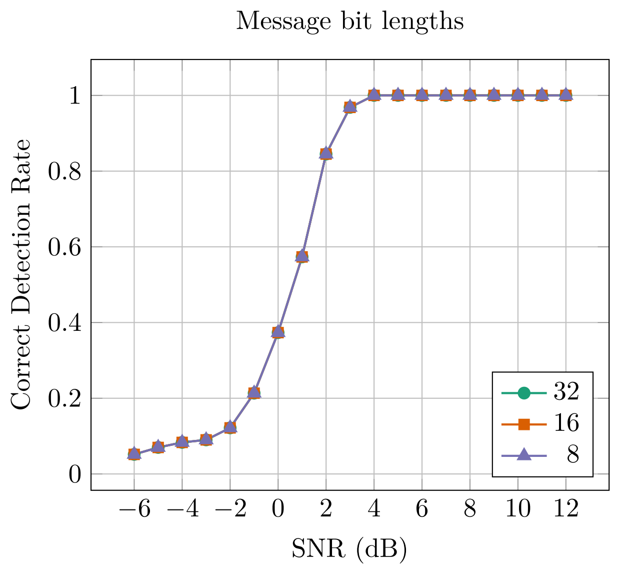


Figure 6.3 – Comparison of message bit lengths.

Figure 6.3 shows the average correct detection rates for decoders using a variety of common bit lengths. The decoders have a message LSB index of −1, calculate syndromes using the min-sum approximation, use the optimal detection algorithm with a normalising factor and consider all syndromes. The various message bit lengths show no performance difference at all. The smallest bit length, 8, is chosen.

The simulation in figure 6.4 focusses on the number of parity-check equation syndromes used to calculate code confidence. The syndromes used is somewhat tied into using a confidence normalisation factor. If we use only B_{min} syndromes, then the normalisation factor can be ignored. Up till now, simulations have considered all syndromes i.e. including all layers. The hardware decoder considers only the very first layer in order to save time. The simulation compares decoders which use all syndromes, a single layer of syndromes and $B_{min} = 27$ syndromes. Using only a single layer's syndromes, degrades the detection performance by $\sim$ 2 dB. Using B_{min} syndromes degrades it by a further $\sim$ 1 dB. It might be worth considering using all syndromes, however this is not currently implemented as an option for hardware. We decide to not use a normalising factor, and only use $B_{min} = 27$ syndromes. All previous choices were kept for these decoders.

The simulation in figure 6.5 shows the performance of the optimal (summing the syndromes) confidence calculation versus the count algorithm. The performance is compared using all syndromes, a single row's syndromes and B_{min} syndromes. The more syndromes are used, the more the optimal confidence algorithm gains over the count algorithm. This advantage is eroded when using only B_{min} syndromes; the count algorithm even outperforms the optimal algorithm at the higher SNR values. As no detection performance is gained from using the optimal confidence calculation algorithm (we are using B_{min} syndromes), we decide to use the count algorithm as it is cheaper computationally.

The final detection simulation, figure 6.6, compares the performance of the optimal decoder from figure 6.1 to our final decoder design. Our design degrades performance by just over 2 dB.

Figure 6.7 shows the average iterations taken to complete a decoding for our decoder. The maximum iterations was set to one hundred. In the cases where a decoding was unsuccessful the iterations used would therefore be one hundred. This simulation helps the user select

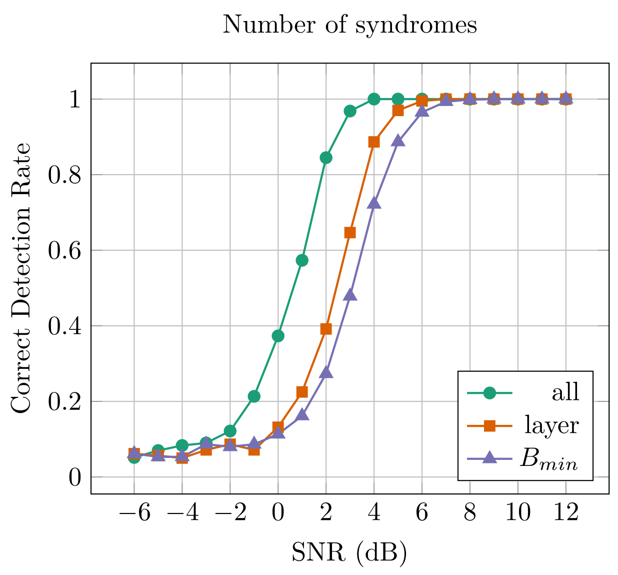


Figure 6.4 – Comparison of number of parity-check syndromes used for detection.

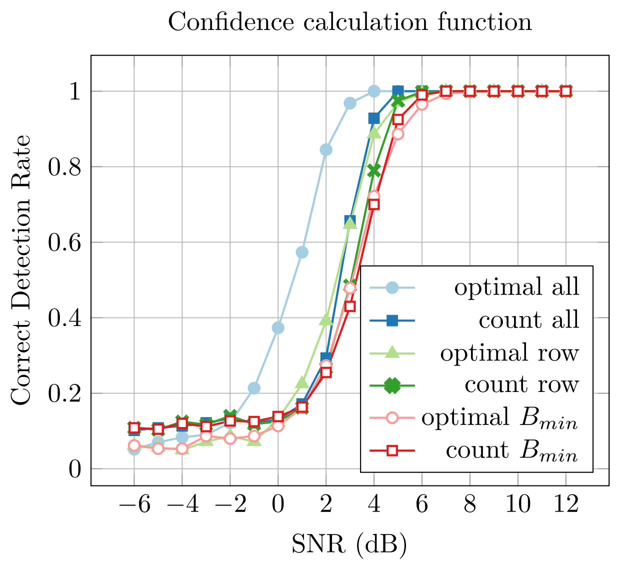


Figure 6.5 – Comparison of detection algorithms.

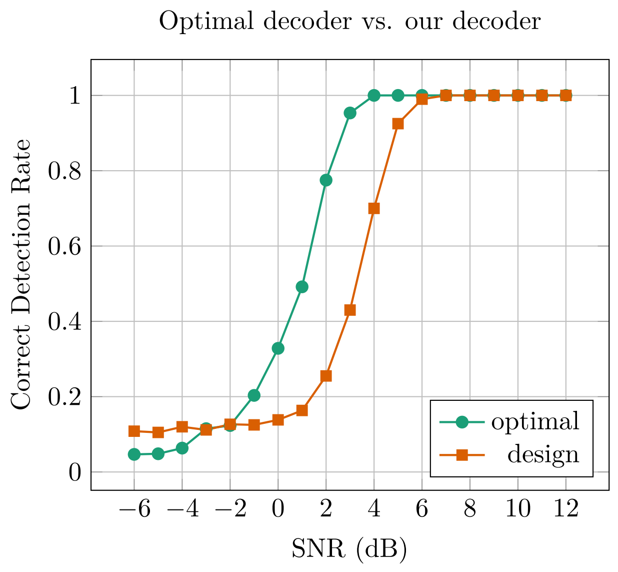


Figure 6.6 – Comparison of the optimal decoder and our design.

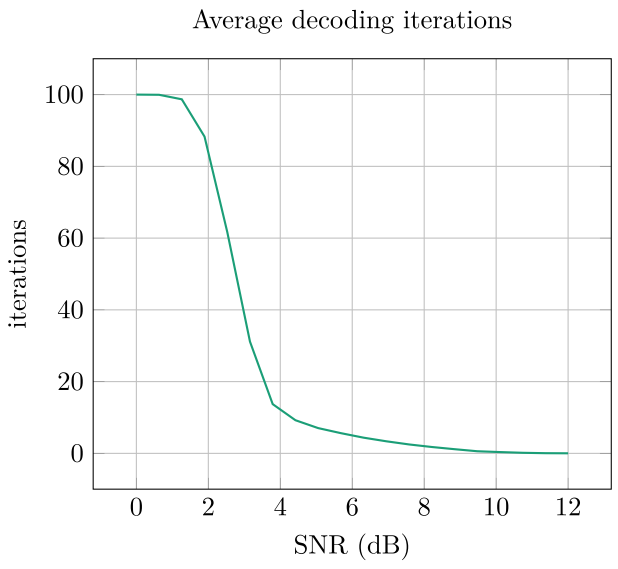


Figure 6.7 – Average decoding iterations used.

the maximum iterations to use in the decoder design. We decide that our decoder will be operating in SNR conditions > 4 dB, and settle on a maximum of thirty iterations.

To ensure that thirty iterations is enough, we use an iteration frequency simulation, shown in figure 6.8. In this simulation we fixed the SNR at 4 dB. It shows the frequency of how many iterations were needed to successfully decode. Note the collection at thirty iterations. This marks the codes which could not be completed in the required iterations. This could be minimised by increasing the maximum iterations. Note that this simulations checks for a correct codeword every iteration, and does not have a delay of m^b iterations as the hardware decoder would. This was done because this simulation uses the average across all codes in

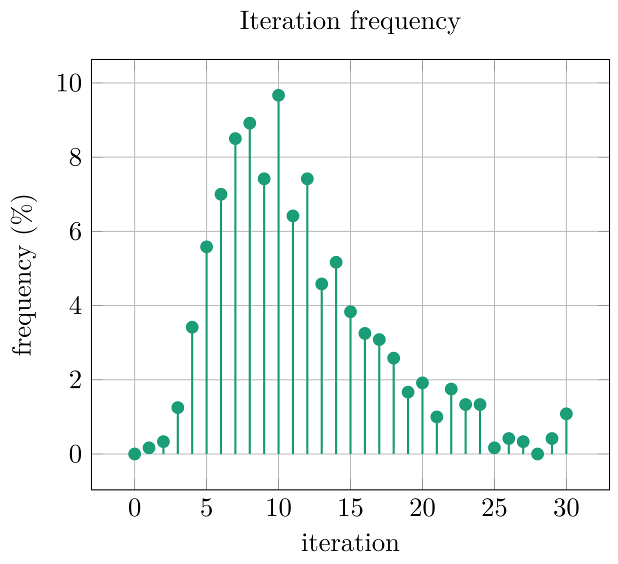


Figure 6.8 – Average decoding iterations used.

the set, and m^b changes per code. This would give skewed frequency results.

For our final simulation, figure 6.9, we simulate the expected transmission BER for each of the codes in the set.

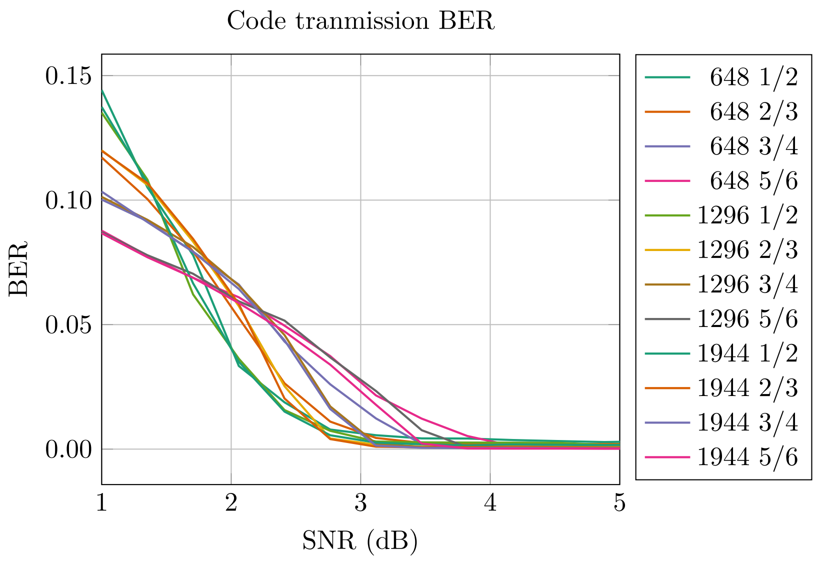


Figure 6.9 – IEEE 802.11n codes' BER.

Chapter 7

Conclusion

LDPC codes form part of a set of modern FEC codes that utilise iterative decoding algorithms. We showed how the roots of these algorithms can be traced to a common origin using the SPA, where links between iterative and non-iterative decoding can be established. As was shown in section 2.3, common decoding algorithms can be expressed using variations of the SPA, typically graphically represented as factor graphs. Non-iterative decoding algorithms have acyclic factor graphs, which implies that the algorithms are precise. Iterative algorithms' graphs contain cycles, which precludes the algorithm from being precise. This forces it to be an iterative process, whose results approximate the desired outcome. LDPC decoding was shown to be an iterative algorithm in section 2.4. The LDPC decoding algorithm was expressed using message passing, using a variety of message formats, culminating in the LLR format. The various message formats' effect on message calculation were examined, with the LLR format requiring the tanh and $\tanh^{-1}$ transcendental functions. We showed how these are commonly approximated using the min-sum approximation in section 2.4.2. In general, LDPC codes have quadratic encoding complexity with respect to codeword length. In section 2.5, we showed numerous solutions to achieve linear encoding complexity. We discussed a family of structured LDPC codes, called QC-LDPC codes. A QC-LDPC code's parity-check matrix can be sectioned into equally sized, square blocks, such that each block is either a zero matrix, or a cyclic permutation of the identity matrix.

In this thesis, we designed a hardware decoding solution for code sets, such as those defined in modern communications standards [1], containing multiple QC-LDPC codes. The design required flexibility, in that these codes may have non-uniform code rates, block sizes, block rows, block columns and codeword lengths. Such a decoder would further benefit from being autonomous, as this allows for systems in which the encoder and decoder have no means to communicate a change in code.

A hardware decoder design was summarised in section 3.1. This was followed by a in-depth, proof-of-concept design of a hardware decoder in chapter 4. The message routing network was identified as the main cause of hardware complexity in LDPC decoders, particularly for decoders allowing for multiple block sizes. The design overcame this by using a QSN – this allows it to rotate a sub-block of size $B < B_{max}$, taking care of non-uniform block sizes. Other non-uniformity issues were dealt with by using a block-serial design i.e. each hardware module processes a block's worth of data at a time. This allows the design to handle any code rate, block rows, block columns and codeword length differences amongst its supported codes. Autonomous behaviour was added to the design by implementing a detection algorithm proposed in [6]. This algorithm assigns each code in a set a confidence value based on the code's average syndrome APP. The code with the highest confidence is then the most likely code. This detection algorithm is engaged once the currently active code's initial confidence value falls below a user-set threshold. In this manner, the decoder potentially detects a change in code at the encoder. A few adaptions to the detection algorithm are available as options to allow the user a trade-off between hardware complexity and performance.

A software system was developed. This system has two subsystems – code generation and simulation. The code generation subsystem automatically generates VHDL code based on user settings. The simulation subsystem enables the user to make decisions about these settings. It provides decoder performance graphs (based on simulations) which allows the user to make informed trade-offs between performance and complexity. These subsystems were discussed broadly in sections 3.2 and 3.3, with a deeper look in chapter 5.

An example of how the simulation subsystem can be leveraged to design a decoder was presented in chapter 6. The design was aimed at incorporating the twelve codes specified by the IEEE 802.11n standard [13]. The simulation results led to a design which, although it had a ~ 2 dB performance loss, still managed to correctly detect the codes 100% at SNR values above ~ 5 dB.

The current state of the decoder design and software system allow for a great deal of future developments. The simulation subsystem can be significantly improved by allowing more fine tune control over individual simulations. This can be achieved by enabling configuration files for simulations, similar to the XML file used to define the code set. Care would need to be taken to ensure that the file layout is simple and the options straightforward to understand. The available transmission options can be expanded drastically, as it currently only caters for BPSK over an AWGN channel. The list of available simulations can also be increased. Simulations involving timing diagrams may be beneficial e.g. to approximate how many clock cycles will be needed, on average, to decode successfully. The code generation subsystem can be extended by adding other target hardware languages and platforms. A further option may be to target software platforms, such as highly parallel servers. This would require a new design, but the decoding principle can remain the same.

The hardware decoder design can be improved significantly, as this was a simple proof-of-concept design. The easiest way to do this would be to add a parallelism factor to the code generation options. This factor would reflect how many blocks can be processed in parallel by the system. A factor of 1 would result in the same design as is currently implemented, with each module processing a single block at a time. A factor q would have each module processing q blocks simultaneously. A further topic worth investigating is the reverse engineering of the permutation matrix. If possible, this might enable a decoder system in which a code's permutation matrix can be retrieved on the fly by analysing the received channel conditional LLRs. This can replace the current code set and code detection modules, and allow the decoder to support all QC-LDPC codes with $B < B_{max}$, where B_{max} would be arbitrarily chosen. Decoding would proceed using the reverse-engineered permutation matrix, until the syndrome APP falls below an arbitrary threshold. This would then trigger a new reverse engineering of the permutation matrix. A reverse engineering methodology that might serve as a starting point is proposed by Jing et al. [15].

Overall, the software system and decoder design fulfils our thesis aims. The decoder design is flexible, and can support multiple QC-LDPC codes with different block sizes and characteristics. The design was successfully automated by adding a detection module, which detects a change in code. This detection has 100% accuracy above an SNR of ~ 8 dB, using the IEEE 802.11n standard's code set [13]. The code generation subsystem successfully generates VHDL code for the decoder design. The simulation subsystem provides information pertinent to making design choices. The subsystem utilises GPL's functions to abstract the simulation software, which allows for an easy extension of its capabilities.

Appendix A

IEEE 802.11n

Code Name	Rate	B	n^b	m^b	n	m	k
IEEE 648 1/2	$\frac{1}{2}$	27	24	12	648	324	324
IEEE 648 2/3	$\frac{2}{3}$	27	24	8	648	216	432
IEEE 648 3/4	$\frac{3}{4}$	27	24	6	648	162	486
IEEE 648 5/6	$\frac{5}{6}$	27	24	4	648	108	540
IEEE 1296 1/2	$\frac{1}{2}$	54	24	12	1296	648	648
IEEE 1296 2/3	$\frac{2}{3}$	54	24	8	1296	432	864
IEEE 1296 3/4	$\frac{3}{4}$	54	24	6	1296	324	972
IEEE 1296 5/6	$\frac{5}{6}$	54	24	4	1296	216	216
IEEE 1944 1/2	$\frac{1}{2}$	81	24	12	1944	972	972
IEEE 1944 2/3	$\frac{2}{3}$	81	24	8	1944	648	1296
IEEE 1944 3/4	$\frac{3}{4}$	81	24	6	1944	486	1458
IEEE 1944 5/6	$\frac{5}{6}$	81	24	4	1944	324	1620

Figure A.1 – IEEE 802.11n code characteristics.

0	-	-	-	0	0	-	-	0	-	-	0	1	0	-	-	-	-	-	-	-	-	-	-
22	0	-	-	17	-	0	0	12	-	-	-	-	0	0	-	-	-	-	-	-	-	-	-
6	-	0	-	10	-	-	-	24	-	0	-	-	-	0	0	-	-	-	-	-	-	-	-
2	-	-	0	20	-	-	-	25	0	-	-	-	-	-	0	0	-	-	-	-	-	-	-
23	-	-	-	3	-	-	-	0	-	9	11	-	-	-	-	0	0	-	-	-	-	-	-
24	-	23	1	17	-	3	-	10	-	-	-	-	-	-	-	-	0	0	-	-	-	-	-
25	-	-	-	8	-	-	-	7	18	-	-	0	-	-	-	-	-	0	0	-	-	-	-
13	24	-	-	0	-	8	-	6	-	-	-	-	-	-	-	-	-	-	0	0	-	-	-
7	20	-	16	22	10	-	-	23	-	-	-	-	-	-	-	-	-	-	-	0	0	-	-
11	-	-	-	19	-	-	-	13	-	3	17	-	-	-	-	-	-	-	-	-	0	0	-
25	-	8	-	23	18	-	14	9	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
3	-	-	-	16	-	-	2	25	5	-	-	1	-	-	-	-	-	-	-	-	-	-	0

Figure A.2 – Permutation Matrix: IEEE 648 1/2

25	26	14	-	20	-	2	-	4	-	-	8	-	16	-	18	1	0	-	-	-	-	-	-
10	9	15	11	-	0	-	1	-	-	18	-	8	-	10	-	-	0	0	-	-	-	-	-
16	2	20	26	21	-	6	-	1	26	-	7	-	-	-	-	-	-	0	0	-	-	-	-
10	13	5	0	-	3	-	7	-	-	26	-	-	13	-	16	-	-	-	0	0	-	-	-
23	14	24	-	12	-	19	-	17	-	-	-	20	-	21	-	0	-	-	-	0	0	-	-
6	22	9	20	-	25	-	17	-	8	-	14	-	18	-	-	-	-	-	-	-	0	0	-
14	23	21	11	20	-	24	-	18	-	19	-	-	-	-	22	-	-	-	-	-	-	0	0
17	11	11	20	-	21	-	26	-	3	-	-	18	-	26	-	1	-	-	-	-	-	-	0

Figure A.3 – Permutation Matrix: IEEE 648 2/3

16	17	22	24	9	3	14	-	4	2	7	-	26	-	2	-	21	-	1	0	-	-	-	-
25	12	12	3	3	26	6	21	-	15	22	-	15	-	4	-	-	16	-	0	0	-	-	-
25	18	26	16	22	23	9	-	0	-	4	-	4	-	8	23	11	-	-	-	0	0	-	-
9	7	0	1	17	-	-	7	3	-	3	23	-	16	-	-	21	-	0	-	-	0	0	-
24	5	26	7	1	-	-	15	24	15	-	8	-	13	-	13	-	11	-	-	-	-	0	0
2	2	19	14	24	1	15	19	-	21	-	2	-	24	-	3	-	2	1	-	-	-	-	0

Figure A.4 – Permutation Matrix: IEEE 648 3/4

17	13	8	21	9	3	18	12	10	0	4	15	19	2	5	10	26	19	13	13	1	0	-	-
3	12	11	14	11	25	5	18	0	9	2	26	26	10	24	7	14	20	4	2	-	0	0	-
22	16	4	3	10	21	12	5	21	14	19	5	-	8	5	18	11	5	5	15	0	-	0	0
7	7	14	14	4	16	16	24	24	10	1	7	15	6	10	26	8	18	21	14	1	-	-	0

Figure A.5 – Permutation Matrix: IEEE 648 5/6

40	-	-	-	22	-	49	23	43	-	-	-	1	0	-	-	-	-	-	-	-	-	-	-
50	1	-	-	48	35	-	-	13	-	30	-	-	0	0	-	-	-	-	-	-	-	-	-
39	50	-	-	4	-	2	-	-	-	-	49	-	-	0	0	-	-	-	-	-	-	-	-
33	-	-	38	37	-	-	4	1	-	-	-	-	-	-	0	0	-	-	-	-	-	-	-
45	-	-	-	0	22	-	-	20	42	-	-	-	-	-	-	0	0	-	-	-	-	-	-
51	-	-	48	35	-	-	-	44	-	18	-	-	-	-	-	-	0	0	-	-	-	-	-
47	11	-	-	-	17	-	-	51	-	-	-	0	-	-	-	-	-	0	0	-	-	-	-
5	-	25	-	6	-	45	-	13	40	-	-	-	-	-	-	-	-	-	0	0	-	-	-
33	-	-	34	24	-	-	-	23	-	-	46	-	-	-	-	-	-	-	-	0	0	-	-
1	-	27	-	1	-	-	-	38	-	44	-	-	-	-	-	-	-	-	-	-	0	0	-
-	18	-	-	23	-	-	8	0	35	-	-	-	-	-	-	-	-	-	-	-	-	0	0
49	-	17	-	30	-	-	-	34	-	-	19	1	-	-	-	-	-	-	-	-	-	-	0

Figure A.6 – Permutation Matrix: IEEE 1296 1/2

39	31	22	43	-	40	4	-	11	-	-	50	-	-	-	6	1	0	-	-	-	-	-	-
25	52	41	2	6	-	14	-	34	-	-	-	24	-	37	-	-	0	0	-	-	-	-	-
43	31	29	0	21	-	28	-	-	2	-	-	7	-	17	-	-	-	0	0	-	-	-	-
20	33	48	-	4	13	-	26	-	-	22	-	-	46	42	-	-	-	-	0	0	-	-	-
45	7	18	51	12	25	-	-	-	50	-	-	5	-	-	-	0	-	-	-	0	0	-	-
35	40	32	16	5	-	-	18	-	-	43	51	-	32	-	-	-	-	-	-	-	0	0	-
9	24	13	22	28	-	-	37	-	-	25	-	-	52	-	13	-	-	-	-	-	-	0	0
32	22	4	21	16	-	-	-	27	28	-	38	-	-	-	8	1	-	-	-	-	-	-	0

Figure A.7 – Permutation Matrix: IEEE 1296 2/3

39	40	51	41	3	29	8	36	-	14	-	6	-	33	-	11	-	4	1	0	-	-	-	-
48	21	47	9	48	35	51	-	38	-	28	-	34	-	50	-	50	-	-	0	0	-	-	-
30	39	28	42	50	39	5	17	-	6	-	18	-	20	-	15	-	40	-	-	0	0	-	-
29	0	1	43	36	30	47	-	49	-	47	-	3	-	35	-	34	-	0	-	-	0	0	-
1	32	11	23	10	44	12	7	-	48	-	4	-	9	-	17	-	16	-	-	-	-	0	0
13	7	15	47	23	16	47	-	43	-	29	-	52	-	2	-	53	-	1	-	-	-	-	0

Figure A.8 – Permutation Matrix: IEEE 1296 3/4

48	29	37	52	2	16	6	14	53	31	34	5	18	42	53	31	45	-	46	52	1	0	-	-
17	4	30	7	43	11	24	6	14	21	6	39	17	40	47	7	15	41	19	-	-	0	0	-
7	2	51	31	46	23	16	11	53	40	10	7	46	53	33	35	-	25	35	38	0	-	0	0
19	48	41	1	10	7	36	47	5	29	52	52	31	10	26	6	3	2	-	51	1	-	-	0

Figure A.9 – Permutation Matrix: IEEE 1296 5/6

57	-	-	-	50	-	11	-	50	-	79	-	1	0	-	-	-	-	-	-	-	-	-	-
3	-	28	-	0	-	-	-	55	7	-	-	-	0	0	-	-	-	-	-	-	-	-	-
30	-	-	-	24	37	-	-	56	14	-	-	-	-	0	0	-	-	-	-	-	-	-	-
62	53	-	-	53	-	-	3	35	-	-	-	-	-	-	0	0	-	-	-	-	-	-	-
40	-	-	20	66	-	-	22	28	-	-	-	-	-	-	-	0	0	-	-	-	-	-	-
0	-	-	-	8	-	42	-	50	-	-	8	-	-	-	-	-	0	0	-	-	-	-	-
69	79	79	-	-	-	56	-	52	-	-	-	0	-	-	-	-	-	0	0	-	-	-	-
65	-	-	-	38	57	-	-	72	-	27	-	-	-	-	-	-	-	-	0	0	-	-	-
64	-	-	-	14	52	-	-	30	-	-	32	-	-	-	-	-	-	-	-	0	0	-	-
-	45	-	70	0	-	-	-	77	9	-	-	-	-	-	-	-	-	-	-	-	0	0	-
2	56	-	57	35	-	-	-	-	-	12	-	-	-	-	-	-	-	-	-	-	-	0	0
24	-	61	-	60	-	-	27	51	-	-	16	1	-	-	-	-	-	-	-	-	-	-	0

Figure A.10 – Permutation Matrix: IEEE 1944 1/2

61	75	4	63	56	-	-	-	-	-	-	8	-	2	17	25	1	0	-	-	-	-	-	-
56	74	77	20	-	-	-	64	24	4	67	-	7	-	-	-	-	0	0	-	-	-	-	-
28	21	68	10	7	14	65	-	-	-	23	-	-	-	75	-	-	-	0	0	-	-	-	-
48	38	43	78	76	-	-	-	-	5	36	-	15	72	-	-	-	-	-	0	0	-	-	-
40	2	53	25	-	52	62	-	20	-	-	44	-	-	-	-	0	-	-	-	0	0	-	-
69	23	64	10	22	-	21	-	-	-	-	-	68	23	29	-	-	-	-	-	-	0	0	-
12	0	68	20	55	61	-	40	-	-	-	52	-	-	-	44	-	-	-	-	-	-	0	0
58	8	34	64	78	-	-	11	78	24	-	-	-	-	-	58	1	-	-	-	-	-	-	0

Figure A.11 – Permutation Matrix: IEEE 1944 2/3

48	29	28	39	9	61	-	-	-	63	45	80	-	-	-	37	32	22	1	0	-	-	-	-
4	49	42	48	11	30	-	-	-	49	17	41	37	15	-	54	-	-	-	0	0	-	-	-
35	76	78	51	37	35	21	-	17	64	-	-	-	59	7	-	-	32	-	-	0	0	-	-
9	65	44	9	54	56	73	34	42	-	-	-	35	-	-	-	46	39	0	-	-	0	0	-
3	62	7	80	68	26	-	80	55	-	36	-	26	-	9	-	72	-	-	-	-	-	0	0
26	75	33	21	69	59	3	38	-	-	-	35	-	62	36	26	-	-	1	-	-	-	-	0

Figure A.12 – Permutation Matrix: IEEE 1944 3/4

13	48	80	66	4	74	7	30	76	52	37	60	-	49	73	31	74	73	23	-	1	0	-	-
69	63	74	56	64	77	57	65	6	16	51	-	64	-	68	9	48	62	54	27	-	0	0	-
51	15	0	80	24	25	42	54	44	71	71	9	67	35	-	58	-	29	-	53	0	-	0	0
16	29	36	41	44	56	59	37	50	24	-	65	4	65	52	-	4	-	73	52	1	-	-	0

Figure A.13 – Permutation Matrix: IEEE 1944 5/6

Bibliography

[1] Awais, M. and Condo, C.: Flexible ldpc decoder architectures. *VLSI Design*, vol. 2012, 2012.

[2] B.P. Lathi and Zhi Ding: *Modern Digital and Analog Communication Systems*. Oxford University Press, 2010.

[3] Christian B. Schlegel and Lance C. Pérez: *Trellis and Turbo Coding*. John Wiley & Sons Inc., 2004.

[4] Richardson, T.J. and Urbanke, R.L.: Efficient encoding of low-density parity-check codes. *Information Theory, IEEE Transactions on*, vol. 47, no. 2, pp. 638–656, 2001.

[5] Stephen Brown and Zvonko Vranesic: *Fundamentals of Digital Logic with VHDL Design*. McGraw-Hill, 2009.

[6] Tian Xia and Hsiao-Chun Wu: Novel Blind Identification of LDPC Codes Using the Average LLR of Syndrome a Posteriori Probability. 2012.

[7] Kschischang, F., Frey, B. and Loeliger, H.-A.: Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, vol. 47, no. 2, pp. 498–519, Feb 2001. ISSN 0018-9448.

[8] Fossorier, M.: Quasicyclic low-density parity-check codes from circulant permutation matrices. *Information Theory, IEEE Transactions on*, vol. 50, no. 8, pp. 1788–1793, Aug 2004. ISSN 0018-9448.

[9] Wiesel, A., Goldberg, J. and Messer, H.: Non-data-aided signal-to-noise-ratio estimation. In: *Communications, 2002. ICC 2002. IEEE International Conference on*, vol. 1, pp. 197–201. 2002.

[10] Peter J. Ashenden: *The Designer's Guide to VHDL*. Morgan Kaufman Publishers, 1996.

[11] Chen, X., Lin, S. and Akella, V.: Qsn - a simple circular-shift network for reconfigurable quasi-cyclic ldpc decoders. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 57, no. 10, pp. 782–786, Oct 2010. ISSN 1549-7747.

[12] Oh, D. and Parhi, K.K.: Area efficient controller design of barrel shifters for reconfigurable ldpc decoders. In: *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pp. 240–243. IEEE, 2008.

[13] IEEE Std 802.11-2012. March 2012.

[14] Cai, Z., Hao, J., Tan, P., Sun, S. and Chin, P.: Efficient encoding of ieee 802.11 n ldpc codes. *electronics letters*, vol. 42, no. 25, pp. 1471–1472, 2006.

[15] Zhou Jing, Huang Zhiping, Su Shaojing and Yang Shadowu: Blind Recognition of Binary Cyclic Codes. *EURASIP Journal on Wireless Communications and Networking*, vol. 218, no. 1, 2013.

[16] Proakis, J.G.: *Digital signal processing: principles algorithms and applications*. Pearson Education India, 2007.

[17] Peebles, P.Z., Read, J. and Read, P.: *Probability, random variables, and random signal principles*, vol. 3. McGraw-Hill New York, 1987.

[18] Martin Fowler with Rebecca Parsons: *Domain-Specific Languages*. Addison-Wesley, 2010.

[19] Altera: *Cyclone III 3C120 Development Board Reference Manual*. Altera Corporation, March 2009.

[20] Tian Xia and Hsiao-Chun Wu: Blind Identification of Nonbinary LDPC Codes Using Average LLR of Syndrome a Posteriori Probability. *IEEE Communications Letters*, vol. 17, no. 7, pp. 1301–1304, July 2013.

[21] Gappmair, W. and López-Valcarce, R. and Mosquera, C.: Joint nda estimation of carrier frequency/phase and snr for linearly modulated signals. *Signal Processing Letters, IEEE*, vol. 17, no. 5, pp. 517–520, May 2010. ISSN 1070-9908.

[22] Gallager, R.: Low-density parity-check codes. *Information Theory, IRE Transactions on*, vol. 8, no. 1, pp. 21–28, January 1962. ISSN 0096-1000.

[23] MacKay, D.J.C. and Neal, R.M.: Near shannon limit performance of low density parity check codes. *Electronics Letters*, vol. 33, no. 6, pp. 457–458, Mar 1997. ISSN 0013-5194.

[24] Tanner, R.: A recursive approach to low complexity codes. *Information Theory, IEEE Transactions on*, vol. 27, no. 5, pp. 533–547, Sep 1981. ISSN 0018-9448.

[25] Pearl, J.: Reverend bayes on inference engines: A distributed hierarchical approach. In: *AAAI*, pp. 133–136. 1982.

[26] Aji, S. and McEliece, R.: The generalized distributive law. *Information Theory, IEEE Transactions on*, vol. 46, no. 2, pp. 325–343, Mar 2000. ISSN 0018-9448.

[27] Loeliger, H.-A.: An introduction to factor graphs. *Signal Processing Magazine, IEEE*, vol. 21, no. 1, pp. 28–41, Jan 2004. ISSN 1053-5888.

[28] Chen, J., Dholakia, A., Eleftheriou, E., Fossorier, M. and Hu, X.-Y.: Reduced-complexity decoding of ldpc codes. *Communications, IEEE Transactions on*, vol. 53, no. 8, pp. 1288–1299, Aug 2005. ISSN 0090-6778.

[29] Jiang, M., Zhao, C., Zhang, L. and Xu, E.: Adaptive offset min-sum algorithm for low-density parity check codes. *Communications Letters, IEEE*, vol. 10, no. 6, pp. 483–485, June 2006. ISSN 1089-7798.

[30] Mansour, M.M. and Shanbhag, N.R.: High-throughput ldpc decoders. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 11, no. 6, pp. 976–996, 2003.

[31] Sun, Y., Karkooti, M. and Cavallaro, J.R.: Vlsi decoder architecture for high throughput, variable block-size and multi-rate ldpc codes. In: *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pp. 2104–2107. IEEE, 2007.

[32] Sun, Y. and Cavallaro, J.R.: Vlsi architecture for layered decoding of qc-ldpc codes with high circulant weight. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, no. 10, pp. 1960–1964, 2013.

[33] Li, Z., Chen, L., Zeng, L., Lin, S. and Fong, W.H.: Efficient encoding of quasi-cyclic low-density parity-check codes. *Communications, IEEE Transactions on*, vol. 54.

[34] Kobayashi, K. and Shibuya, T.: Generalization of lu's linear time encoding algorithm for ldpc codes. In: *Information Theory and its Applications (ISITA), 2012 International Symposium on*, pp. 16–20. IEEE, 2012.

[35] Shibuya, T.: Block-triangularization of parity check matrices for efficient encoding of linear codes. In: *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, pp. 533–537. IEEE, 2011.

[36] Lu, J. and Moura, J.M.: Linear time encoding of ldpc codes. *Information Theory, IEEE Transactions on*, vol. 56, no. 1, pp. 233–249, 2010.

[37] Berrou, C. and Glavieux, A.: Near optimum error correcting coding and decoding: Turbo-codes. *Communications, IEEE Transactions on*, vol. 44, no. 10, pp. 1261–1271, 1996.

[38] L. R. Bahl and J. Cocke, F. Jelinek and J. Raviv: Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate. *IEEE Transactions on Information Theory*, pp. 284–287, March 1974.

[39] Silvio A. Abrantes: From BCJR to turbo decoding:MAP algorithms made easier, April 2004.

[40] Glavieux, A.: *Channel coding in communication networks: from theory to turbocodes*, vol. 667. John Wiley & Sons, 2010.

[41] ETSI, E.: 302 769 v1. 2.1 (2011-04) digital video broadcasting (dvb); frame structure channel coding and modulation for a second generation digital transmission system for cable systems (dvb-c2). 2011.

[42] ETSI, T.: 302 755 v1. 3.1 (2012-04): Digital video broadcasting (dvb). *Frame structure channel coding and modulation for a second generation digital terrestrial television broadcasting system (DVB-T2)*, 2012.

[43] ETSI, E.: 302 307 v1. 3.1. *Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2).*

[44] IEEE Std 802.3an-2006. September 2006.

[45] IEEE Std 802.22-2011. July 2011.

[46] IEEE Std 802.15.3c-2009. October 2009.

[47] IEEE Std 802.16-2009. May 2009.