HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Electronics, Communication and Automation
Department of Communications and Networking

Huageng Chi

HARDWARE DESIGN OF DECODER FOR LOW-DENSITY PARITY CHECK CODES

Thesis submitted in partial fulfilment of the requirement for the degree of Master of Science in Technology

Espoo, Finland, 3rd August 2009

Supervisor: Prof. Patric Östergård

Instructor: M.Sc. Mika Rautio

HELSINKI UNIVERSITY OF TECHNOLOGY
ABSTRACT of the Master's thesis

Author:	Huageng Chi
Name of the thesis:	Hardware Design of Decoder for Low-Density Parity Check Codes
Date:	3rd August 2009 Number of pages: x + 62
Faculty:	Faculty of Electronics, Communication and Automation
Professorship:	Communications Code: S-72
Supervisor:	Prof. Patric Östergård
Instructor:	M.Sc. Mika Rautio

A hardware decoder architecture is presented in this thesis for quasi-cyclic (QC) low-density parity check (LDPC) codes.

The decoder is real-time configurable and supports 15 codes which are combination of 3 rates and 5 lengths. The partly parallel architecture implements layered decoding. A check node decoder is serial and implements min-sum correction algorithm. The proposed design techniques include out-of-order memory-write, two-stage multi-size shifter, serial decoding termination.

The decoder consumes about half amount of logic resource on the Xilinx FPGA chip XC2VP50-5F1152. The worst case throughput at 20 iterations ranges from 5 Mbits to 60 Mbits (information bits) per second. Higher throughput can be obtained by the proposed optimisation. Reuse for similar codes is possible.

Keywords: low-density parity check (LDPC) decoder, FPGA, multi-rate, multi-length, layered decoding, out-of-order memory-write, multi-size shifter

Preface

This thesis work was part of a project[1] carried out at VTT Technical Research Center of Finland in 2007. The work started with the codes and certain directive information provided by project partners. The thesis was written in Department of Communications and Networking at TKK Helsinki University of Technology.

I would like to thank Professor Patric Östergård at TKK for being my supervisor and providing detailed guidelines, instructions, advices and comments.

I would also like to thank the team manager Mika Rautio and the project manager Jussi Roivainen, both at VTT, for preparing the topic for me, and also for their support to my work and study. Special thanks go to Mika Rautio for being my instructor.

I am most grateful to my girl friend Jing Zhang. This thesis would never be complete without her love and support. I owe so much to her.

Finally, I would like to express my gratitude to my mother, my father, and my elder brother.

Otaniemi, Espoo, 3rd August 2009

Huageng Chi

[1] It is a sub project of WINNER II, an EU funded project: http://www.ist-winner.org/index.html

Contents

List of symbols and abbreviations

C_m	check node $m \in \mathcal{M}$
$C_{m'}^i$	check node with local index m' within CNB i
$H_{M\times N}$	parity check matrix, size is $M \times N$
$H_{i,j}$	submatrix at block row i and block column j of parity check matrix for QC LDPC code
L_n	total information of bit n, generally in LLR form
M	number of rows of parity check matrix
$\mathcal{M}$	index set $\mathcal{M} = \{0, 1, \cdots, M-1\}$ for check nodes, or parity check constraints, or rows of parity check matrix
$\mathcal{M}(n)$	all check nodes that checks variable node n
$\mathcal{M}(n) \backslash m$	all check nodes that checks variable node n, excluding check node $m \in \mathcal{M}(n)$
N	number of columns of parity check matrix, equal to length of codeword
$\mathcal{N}$	index set $\mathcal{N} = \{0, 1, \cdots, N-1\}$ for variable nodes, or code bits, or columns of parity check matrix
$\mathcal{N}(m)$	all variable nodes checked by check node m

$\mathcal{N}(m)\backslash n$	all variable nodes checked by check node m, excluding variable node $n \in \mathcal{N}(m)$
$Q_{n,m}$	V2C message from variable node n to check node m, generally in LLR form
$Q^{j,i}_{n',m'}$	V2C message from variable node $V^j_{n'}$ to check node $C^i_{m'}$, generally in LLR form
$R_{m,n}$	C2V message from check node m to variable node n, generally in LLR form
$R^{i,j}_{m',n'}$	C2V message from check node $C^i_{m'}$ to variable node $V^j_{n'}$, generally in LLR form
V_n	check node $n \in \mathcal{N}$
$V^j_{n'}$	variable node with local index n' within VNB j
z	block size of block LDPC code
APP	*a posteriori* probability
C2V	check-to-variable
CNB	check node block
CND	check node decoder
FPGA	field-programmable gate array
LDPC	low-density parity check
LLR	log likelihood ratio
QC	Quasi-cyclic
TPMP	two-phase message passing
TDMP	turbo decoding message passing

V2C	variable-to-check
VHDL	very-high-speed-integrated-circuit hardware description language
VNB	variable node block

List of Figures

List of Tables

Chapter 1

Introduction

Channel noise causes transmission errors in communication systems. An effective means to improve reliability is to employ forward error correction (FEC) codes. After Shannon laid down the theoretical foundation of communications in his 1948 landmark paper [1], a "central objective was to find practical coding schemes that could approach channel capacity on well-understood channels such as the additive white Gaussian noise (AWGN) channel" [2]. Turbo codes invented in 1993 [3] is the first practical capacity-approaching code, which is widely deployed nowadays.

Following the success of turbo code, Low-density parity check (LDPC) code, which was invented by Gallager in early 1960s [4] but forgotten for three decades, was rediscovered by MacKay *et al* in mid-1990s [5, 6]. LDPC code has near Shanon limit performance [7, 8] as well. Unlike turbo code, the structure of LDPC code is inherently parallel, thus enables flexible implementations for a wide range of applications.

The rediscovery triggered research on LDPC code. LDPC code has been adopted in various areas, including new generation standards like DVB-S2, IEEE 802.22, 802.16e, 802.11n, 802.3an, and so on [9]. However, products of LDPC decoders are still rare in market, because of implementation chal-

lenges. Effective implementation of practical hardware decoder is of great interest.

1.1 LDPC decoder hardware implementation

LDPC code had been ignored for about 30 years, partly because it was "much too complex for the technology at that time [2]". Various decoding algorithms exist. In a straightforward implementation, a hardware decoder consists of a large number of small decoders of two kinds. Every small decoder of one kind is connected to multiple small decoders of the other kind. During decoding procedure, every small decoder receives information from its neighbours, processes the received information, and sends the processing result to each neighbour. Information exchanged between a pair of nodes is called message, exchange of information is called message passing, and processing of information is called message update. All small decoders perform message updates and message passing simultaneously and repeatedly. Stop criterion is evaluated from time to time.

Updating a message by hardware requires a data path. A data path consists of basic logical and mathematical operators. Operators are in turn built by digital circuits. Hardware resource includes logic resource, memory resource, routing resource, and so on. A small decoder consumes certain amount of logic resource and memory resource to build data paths. Passing messages between small decoders require routing resources. A typical LDPC decoder in the straightforward implementation consists of large number of small decoders. For example, a code in this thesis needs 4,608 variable node decoders and 2,304 check node decoders. 35,328 messages need to be computed, stored, and passed in every iteration, and the number of iterations can be as many as 20. Given the hardware technology in the past, such an implementation is too complex to be practical for high speed applications.

Hardware decoder implementation became realisable only in recent years due to improvements in code design, algorithm design, decoder architecture de-

sign, and microelectronics technology. But new challenges exist due to stringent system requirements such as low hardware cost, high error correction performance, high decoding throughput, flexibility to support multiple codes of different rates and codeword lengths. A fully parallel decoder enables high throughput but requires prohibitively large amount of hardware resource [10]; a completely serial decoder consumes least amount of hardware resource but its throughput is low. All practical decoders are made partly parallel to make trade-offs between hardware cost and throughput. The key task in architecture design is to parallelize message updates and message exchanges, and map those algorithmic functionality to limited amount of logic, memory, and routing resource provided by target hardware device. The task is harder when a decoder needs support multiple code rates and multiple codeword lengths. A decoder designed for one application may not fit to another, as architecture design of LDPC decoder is closely coupled with application-specific decisions such as code's characteristics, decoding algorithm, system requirements, and hardware resource constraints.

1.2 Scope of thesis

The codes to be implemented in the thesis are Quasi-cyclic (QC) LDPC codes. 3 base matrices are defined for code rates $\frac{1}{2}$, $\frac{2}{3}$, and $\frac{3}{4}$. Each base matrix consists of 48 columns and expands to 5 parity check matrices corresponding to block sizes 12, 24, 36, 48, and 96. Therefore 15 codes are defined. 5 codeword lengths of each rate are $[12, 24, 36, 48, 96] \times 48 = [576, 1152, 1728, 2304, 4608]$.

The objective of the thesis work was to design a hardware decoder architecture suitable for FPGA device. The implementation aims at small hardware cost and challenging system requirements, including high decoding throughput, real-time support to multiple code rates and multiple codeword lengths, and small degradation of error correction performance caused by hardware implementation. The decoder is part of a trail system for demonstration

purpose, therefore optimisation for hardware cost and decoding throughput is not prioritised, however, space for future optimisation is considered.

The main result of the thesis is a hardware architecture suitable for FPGA device. The decoder is real time configurable to decode any of the 15 specified LDPC codes. A partly parallel architecture implements layered decoding, all check node decoders in a check node block (CNB) operate in parallel, and each check node decoder is serial and pipelined.

The thesis also presents some design techniques. Out-of-order memory-read is used to improve throughput. A two-stage multi-size shifter is designed to perform cyclic shift on first z values of of 96-value vector, where $z \in \{12, 24, 36, 48, 96\}$ is the block size. Decoder checks consecutive m_b CNBs in serial manner to evaluate stop criterion, where $m_b \in \{12, 16, 24\}$ is number of block rows. Control signals are saved in memory modules, and updating those memory content adapts the design to other codes without hardware redesign, if the new codes are not far different from those used in this thesis.

1.3 Thesis organisation

The rest of the thesis is organised as follows. Decoding algorithms and techniques to reduce complexity in hardware implementation are discussed in Chapter 2. The proposed decoder architecture and techniques are presented in Chapter 3. Result of the solution is reported in Chapter 4, and discussed in Chapter 5. Chapter 6 concludes the thesis. References and appendices are in the end of the thesis.

Chapter 2

Algorithm and architecture

LDPC codes and decoding algorithms are introduced in this chapter. Various techniques that reduce hardware implementation complexity can be found in literature, those employed in this thesis work are briefly introduced. A partly parallel decoder architecture is illustrated.

2.1 System model

Figure 2.1 presents a basic model of a communications channel. An encoder maps a source message $\underline{s} \triangleq (s_0 s_1 \cdots s_{K-1})$ to a codeword $\underline{c} \triangleq (c_0 c_1 \cdots c_{N-1}) \in \mathcal{C}$, where $s_k \in \{0,1\}$, $c_n \in \{0,1\}$, $\mathcal{C}$ is a code, K is message length, N is codeword length. Codeword $\underline{c}$ is transmitted in form of signal $\underline{x} \triangleq (x_0 x_1 \cdots x_{N-1})$, $x_n \in \{-1,+1\}$, using mapping: $c_n = 0 \rightarrow x_n = +1$; $c_n = 1 \rightarrow x_n = -1$.

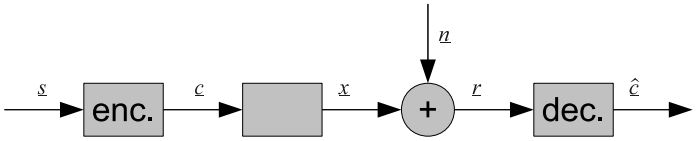

Figure 2.1: System model

row index ↓	0	1	2	3	4	5	6	7	8	9	←column index
0	0	0	0	0	1	1	0	1	0	1	
1	1	0	1	0	0	1	0	1	0	0	
2	0	1	1	0	1	0	1	0	1	1	
3	1	0	0	1	0	1	0	1	1	1	
4	0	1	1	1	1	0	0	0	0	0	

Table 2.1: Example: parity check matrix

The received signal through an AWGN channel is $\underline{r} = \underline{x} + \underline{n}$, where $\underline{n} \triangleq (n_0 n_1 \cdots n_{N-1})$ is zero mean white Gaussian noise with variance $\sigma^2 = N_0/2$. An decoder produces estimation $\hat{\underline{c}}$ of the transmitted codeword $\underline{c}$.

2.2 Parity check matrices and graphs

Every (N, K) binary linear block code is specified by its parity check matrix $H_{M\times N}$, which is assumed full ranked, and whose dimension is $M \times N$. A column of a matrix corresponds to a code bit. Columns and bits are indexed from left to right by $0, 1, \cdots, N-1$. A row corresponds to a parity check constraint, rows and parity check constraints are indexed from top to bottom by $0, 1, \cdots, M-1$. Let $h(m, n)$ be the parity check matrix entry at row m and column n, parity check m checks bit n if and only if $h(m, n) = 1$. Table 2.1 presents an example parity check matrix with $M = 5$ and $N = 10$.

The graph introduced by Tanner in [11] is referred as Tanner graph, which is commonly used to represent LDPC code. A graph is a set of nodes connected by a set of edges. A Tanner graph is a bipartite graph, which has two types of nodes, and each edge connects a node of one type to a node of the other type. One type of nodes represent symbols of a codeword, and the other type of nodes represent parity check constraints. Figure 2.2 presents the Tanner graph for the example code given in Table 2.1. Circles are variable nodes,

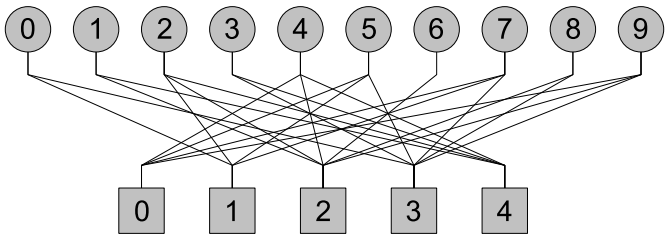

Figure 2.2: Example: Tanner graph

indexed from left to right by $0, 1, \cdots, N-1$, with node n corresponding to bit n. Boxes are check nodes representing parity checks, indexed from left to right by $0, 1, \cdots, M-1$, with node m corresponding to parity check m. An edge connects check node m and variable node n if and only if $h(m, n) = 1$. Degree of a node is the number of its neighbours.

A one-to-one correspondence links a row of a parity check matrix, a parity check constraint, and a check node. Another one-to-one correspondence links a column of a parity check matrix, a codeword bit, and a variable node.

A LDPC code is a linear block codes with sparse parity check matrix. A sparse matrix contains only a few ones in each row and each column. In general, parity check matrix of a LDPC code is made large and pseudo-random in order to obtain good error correction performance. Most recent practical codes are structured, showing much regularity, but randomness is still obvious.

Following notations are used in this thesis:

- $\mathcal{N} \triangleq \{0, 1, \cdots, N-1\}$ is index set for variable nodes, or code bits, or matrix columns; $\mathcal{M} \triangleq \{0, 1, \cdots, M-1\}$ is index set for check nodes, or parity check constraints, or matrix rows.
- $\mathcal{N}(m)$ denotes all variable nodes checked by m. $\mathcal{N}(m)$ with $n \in \mathcal{N}(m)$ excluded is denoted by $\mathcal{N}(m) \backslash n$; $\mathcal{M}(n)$ is index set for all check nodes that are neighbours of variable node n. $\mathcal{M}(n)$ with $m \in \mathcal{M}(n)$ excluded is denoted by $\mathcal{M}(n) \backslash m$.

- Check node m is denoted by C_m; variable node n is denoted by V_n.
- Check node m's degree is $d_m = |\mathcal{N}(m)|$; variable node n's degree is $d_n = |\mathcal{M}(n)|$.

2.3 Encoding and decoding

To design a LDPC code is to find a good parity check matrix. Given a parity check matrix H, the code $\mathcal{C}$ is the null space of H, i.e., $\underline{c} \in \mathcal{C}$ if and only if $H\underline{c}^T = 0$.

Expression $\underline{c} = \underline{m}G$ describes encoding message $\underline{m}$ to codeword $\underline{c}$, where G is the generator matrix that can be obtained from parity check matrix H. This direct method requires large number of multiplications and additions thus it is too complex for hardware implementation. For a class of widely used QC LDPC codes, encoding is efficiently realised by solving $H\underline{c}^T = 0$ for $\underline{c}$. The codes are systematic, and a codeword $\underline{c}$ consists of information part $\underline{m}$ and parity part $\underline{p}$, written as $\underline{c} = [\underline{m}\,\underline{p}]$. When $\underline{m}$ is given, corresponding $\underline{p}$ can be derived by solving $H[\underline{m}\,\underline{p}]^T = 0$. Further discussion of encoding can be found in literature such as [12], and it is out of the scope of this thesis.

Given received signal $\underline{r}$, a maximum likelihood (ML) decoding algorithm finds codeword estimation

$$\hat{\underline{c}} = \max_{\underline{c} \in \mathcal{C}} \Pr(\underline{r} \mid \underline{c})$$

A maximum *a posteriori* (MAP) algorithm finds bit-wise estimation for each bit, the estimation for bit n is

$$\hat{c_n} = \max_{c_n \in \{0,1\}} \Pr(c_n \mid \underline{r})$$

The message passing algorithm discussed in the following text is a MAP algorithm. The min-sum algorithm, which can be considered an approximation to message passing algorithm, is a ML algorithm.

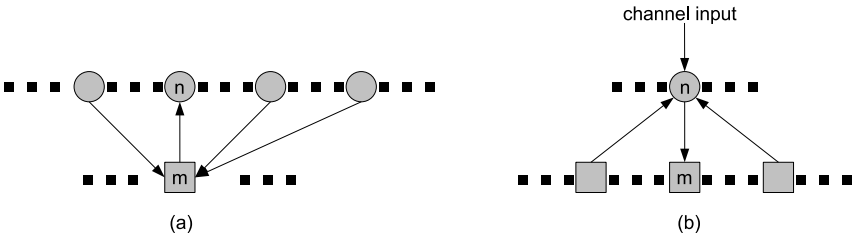

Figure 2.3: Messages and updates

2.4 Message passing algorithms

Among various decoding algorithms, a fundamental one and its variants are referred with different names like message passing (MP), belief propagation (BP), and sum product algorithm (SPA). Those terms are often interchangeably used in literature, so in this thesis. Message passing algorithm is briefly illustrated in this section. Detailed treatment of codes on graphs can be found in a number of references, for example, factor graph and SPA discussed in [13], Forney graph and SPA discussed in [14], BP discussed in [5] [6].

Along every graph edge flow two kinds of information, which are represented by variable-to-check (V2C) message and check-to-variable (C2V) message. C2V message from C_m to V_n carries conditional information on V_n's value being 0 or 1, as illustrated in Figure 2.3 (a). The information is obtained using all V2C messages from C_m's neighbouring variable nodes excluding V_n. V2C message from V_n to C_m carries conditional information on V_n's value being 0 or 1, as illustrated in Figure 2.3 (b). The information is obtained using channel input to V_n and all C2V messages from V_n's neighbouring check nodes excluding C_m. Procedures computing V2C messages and C2V messages are termed as variable update and check update, respectively.

A decoder needs perform large number of updates as fast as possible. Scheduling is needed to allocate update tasks to hardware resource and time resource. An iteration covers the least number of clock cycles for all messages to be

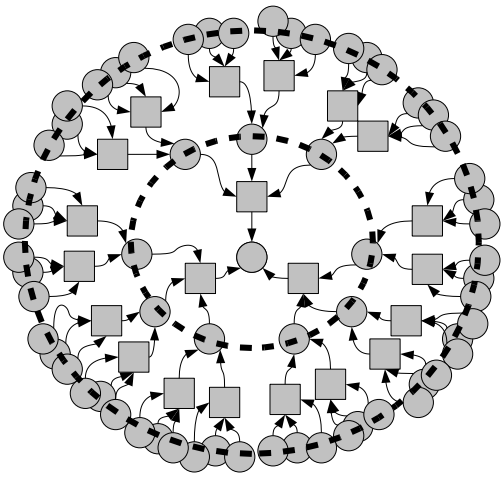

Figure 2.4: Message passing viewed from a particular variable node

updated. The algorithm runs for multiple iterations to converge. The conventional two-phase message passing (TPMP) is the most straightforward example to illustrate scheduling. In TPMP, all check nodes perform check node updates in one phase, all variable nodes perform variable node updates in the subsequent phase, and those two phases constitute one iteration. In order to start the iterative procedure, all V2C messages leaving a variable node are initialised to channel input to that node, then, check node updates can be performed. At the end of each iteration, every variable node makes hard decision using latest C2V messages as well as its channel input. Decoding terminates if decisions satisfy all parity check constraints, otherwise, new iteration starts, until a preset maximum number of iterations is reached.

The effectiveness of such algorithms is illustrated by Figure 2.4. The variable node of interest is drawn in the center of the figure. Tiers of variable nodes are indexed to outwards direction by $1, 2, \cdots$. Only two tiers of variable nodes are presented due to lack of space. Arrows represent graph edges and messages. Each variable node also accepts information from corresponding channel input, which is not shown in the figure. The key observation is that, after n iterations, the central variable node collects information from all variable nodes up to tier n, and the collected information is utilised to

decode the center variable node. Consequently and generally, the more the number of iterations, the more reliable is the decoding result.

2.4.1 Message definition

Messages being exchanged can be probabilities or other quantities derived from probabilities. For probability messages, let $r_{m,n}$ denote the C2V message from C_m to V_n, and $q_{n,m}$ denote the V2C message from V_n to C_m. $r_{m,n}$ and $q_{n,m}$ are two-element *a posteriori* probability (APP) vectors defined as

$$\begin{Bmatrix} r_{m,n} \triangleq \begin{Bmatrix} \Pr\left(n=0 \mid q_{j,m}, j \in \mathcal{N}\left(m\right) \backslash n\right) \\ \Pr\left(n=1 \mid q_{j,m}, j \in \mathcal{N}\left(m\right) \backslash n\right) \end{Bmatrix} \\ q_{n,m} \triangleq \begin{Bmatrix} \Pr\left(n=0 \mid r_n, r_{i,n}, i \in \mathcal{M}\left(n\right) \backslash m\right) \\ \Pr\left(n=1 \mid r_n, r_{i,n}, i \in \mathcal{M}\left(n\right) \backslash m\right) \end{Bmatrix} \end{Bmatrix}$$

where r_n is channel input for bit n.

Log likelihood ratios (LLRs) are widely used as messages for a couple of advantages over probabilities: instead of operating on two probabilities, only one ratio is maintained, multiplication and division in real valued domain become addition and subtraction in LLR domain, fixed-point algorithms using LLRs are more numerically stable and suffers less performance loss, expressions like (2.1) are neat in LLR domain, and easier to analyse and approximate, and so forth. In accordance to $r_{m,n}$ and $q_{n,m}$, messages in LLR domain are defined as

$$\begin{cases} R_{m,n} \triangleq \ln \frac{\Pr(n=0 \mid q_{j,m}, j \in \mathcal{N}(m) \backslash n)}{\Pr(n=1 \mid q_{j,m}, j \in \mathcal{N}(m) \backslash n)} \\ Q_{n,m} \triangleq \ln \frac{\Pr(n=0 \mid r_n, r_{i,n}, i \in \mathcal{M}(n) \backslash m)}{\Pr(n=1 \mid r_n, r_{i,n}, i \in \mathcal{M}(n) \backslash m)} \end{cases}$$

and the channel input LLR to bit n is defined as

$$L_{c,n} \triangleq \ln \frac{\Pr\left(c_n = 0 \mid r_n\right)}{\Pr\left(c_n = 1 \mid r_n\right)}$$

2.4.2 Equations

The check node update producing C2V message $R_{m,n}$ is described (e.g., in [15]) by

$$\tanh \frac{R_{m,n}}{2} = \prod_{j \in \mathcal{N}(m) \setminus n} \tanh \left(\frac{Q_{j,m}}{2} \right) \tag{2.1}$$

The variable node update producing V2C message $Q_{n,m}$ is described (e.g., in [15]) by

$$Q_{n,m} = \sum_{i \in \mathcal{M}(n) \setminus m} R_{i,n} + L_{c,n} \tag{2.2}$$

The LLR of APP for bit n is also referred in this thesis as column sum, or total information, as it is a sum of all LLR messages to n

$$L_n \triangleq \ln \left(\frac{p\left(c_n = 0 \mid \underline{\mathrm{r}}\right)}{p\left(c_n = 1 \mid \underline{\mathrm{r}}\right)} \right) = \sum_{m \in \mathcal{M}(n)} R_{m,n} + L_{c,n} \tag{2.3}$$

L_n is sliced to obtain hard decision:

$$\begin{cases} \hat{c}_n = 0 & L_n \geq 0 \\ \hat{c}_n = 1 & L_n < 0 \end{cases}$$

2.5 Quasi-cyclic (QC) LDPC code

Quasi-cyclic (QC) LDPC code [16] [17] is also referred as Block LDPC code [18]. Let$H_{M \times N}$ be an $M \times N$ parity check matrix of a QC LDPC code. $H_{M \times N}$ consists of m_b rows and n_b columns of submatrices of size $z \times z$, where z is block size, $M = m_b \times z$, and $N = n_b \times z$. A submatrix is either a null matrix, or a circulant matrix obtained by shifting cyclically each row of an

identity matrix to the right for p steps. Define

$$P_z \triangleq \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

then, a submatrix of size $z \times z$ with shift step p, $0 \leq p \leq z-1$, is $(P_z)^p$, the p-th power of P_z. Conventionally, $(P_z)^0$ is defined as identity matrix.

A row of submatrices is called a block row. Block rows are indexed from top to bottom by $0, 1, \cdots, m_{b-1}$. Let the symbol $\lfloor\ \rfloor$ denote floor function such that $\lfloor x \rfloor$ returns the largest integer no more than x. Row m locates in block row $\lfloor m/z \rfloor$, and is indexed locally in block row $\lfloor m/z \rfloor$ by $(m \mod z)$. Accordingly, parity check constraints and check nodes are also grouped to m_b blocks, and indexed globally and locally in the same manner. For instance, check node m is indexed locally by $(m \mod z)$ in check node block (CNB) $\lfloor m/z \rfloor$. Similarly, parity check matrix columns, code bits, and variable nodes, are grouped respectively to n_b block columns, code bit blocks, and variable node blocks (VNBs). Those blocks are indexed from left to right by $0, 1, \cdots, n_b - 1$. A global index n corresponds to local index $(n \mod z)$ in block $\lfloor n/z \rfloor$. For instance, code bit n is indexed locally by $(n \mod z)$ in code bit block $\lfloor n/z \rfloor$.

Following notations are used. Check node C_m is also written as $C^{\lfloor m/z \rfloor}_{(m \mod z)}$, meaning a check node in CNB $\lfloor m/z \rfloor$ with local index $(m \mod z)$. Likewise, variable node V_n can be denoted by $V^{\lfloor n/z \rfloor}_{(n \mod z)}$, and total information L_n can be written as $L^{\lfloor n/z \rfloor}_{(n \mod z)}$. V2C message $Q_{n,m}$ can be written as $Q^{\lfloor n/z \rfloor, \lfloor m/z \rfloor}_{(n \mod z),(m \mod z)}$, meaning the Q message from $V^{\lfloor n/z \rfloor}_{(n \mod z)}$ to $C^{\lfloor m/z \rfloor}_{(m \mod z)}$, also, C2V message $R_{m,n}$ can be written as $R^{\lfloor m/z \rfloor, \lfloor n/z \rfloor}_{(m \mod z),(n \mod z)}$.

Suppose submatrix at block row i and block column j is $H_{i,j} = (P_z)^p$. For $k = 0, 1, \cdots, z-1$, check node C^i_k is connected to variable node $V^j_{k+p \mod z}$, and conversely, variable node V^j_k is connected to check node $C^i_{k+z-p \mod z}$.

		0	1	2	3	4	5	6	7	8	9	10	11	←	global indices
		0	1	2	0	1	2	0	1	2	0	1	2	←	local indices
0	0	0	1	0	0	0	0	0	1	0	0	0	1		
1	1	0	0	1	0	0	0	0	0	1	1	0	0	←	block row 0
2	2	1	0	0	0	0	0	1	0	0	0	1	0		
3	0	1	0	0	0	1	0	0	0	1	0	0	0		
4	1	0	1	0	0	0	1	1	0	0	0	0	0	←	block row 1
5	2	0	0	1	1	0	0	0	1	0	0	0	0		
			↑			↑			↑			↑			
	↑		0			1			2			3		←	block column index

↑ local indices
global indices

Table 2.2: Block LDPC code parity check matrix (trivial example)

2.5.1 An example

Foregoing definitions are illustrated by a trivial example shown in Table 2.2. In this example, $M = 6$, $N = 12$, $z = 3$, $m_b = 2$, $n_b = 4$, and parity check matrix is partitioned to $2 \times 4 = 8$ submatrices. For example, the submatrix in block row 0 and block column 3 is

$$H_{0,3} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

therefore, check nodes 0, 1, and 2 in CNB 0 are connected respectively to variable nodes 2, 0, and 1 in VNB 3, and the permutation of index vector is described by

$$\begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} = H_{0,2} \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \text{ or } \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0+2 & \mod 2 \\ 1+2 & \mod 2 \\ 2+2 & \mod 2 \end{bmatrix}$$

1	-1	1	2
0	1	2	-1

Table 2.3: Block LDPC code base matrix (trivial example)

2.5.2 Base matrix and expansion

Parity check matrix of a block LDPC code is often described by a $m_b \times n_b$ base matrix $B_{m_b \times n_b}$. The (i, j)-th entry of $B_{m_b \times n_b}$, denoted by $b(i, j)$, is

$$\begin{cases} b(i,j) = -1 & \text{when } H_{i,j} = \mathbf{0}_{z \times z} \\ b(i,j) = p & \text{when } H_{i,j} = (P_z)^p \end{cases}$$

Table 2.3 presents the base matrix of the code defined in Table 2.2.

Expanding one base matrix to multiple parity check matrices with different block sizes lead to codes of multiple codeword lengths. Given a base matrix, a parity check matrix for a code of block size z is obtained by setting submatrix as

$$\begin{cases} H_{i,j} = \mathbf{0}_{z \times z} & \text{if } b(i,j) = -1 \\ H_{i,j} = (P_z)^{b(i,j) \mod z} & \text{otherwise} \end{cases}$$

This is the commonly used modulo expansion method.

2.6 Approaches to reduce complexity

The techniques employed in this thesis work to reduce computation complexity and hardware complexity are shortly presented in this section.

2.6.1 Min-sum algorithm and its correction

Comparing (2.1) and (2.2) shows that check nod update dominates computational complexity. Min-sum algorithm (MSA [19] [20]) provides simple approximation to (2.1) at the cost of small performance loss.

In MSA, the sign part and magnitude part of $R_{m,n}$ are computed separately. Let the sign function be $\operatorname{sign}(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases}$, the sign part is given by

$$\operatorname{sign}(R_{m,n}) = \prod_{j \in \mathcal{N}(m) \backslash n} \operatorname{sign}(Q_{j,m}) \tag{2.4}$$

and the magnitude part is related to input magnitudes by

$$\tanh \frac{|R_{m,n}|}{2} = \prod_{j \in \mathcal{N}(m) \backslash n} \tanh \left(\frac{|Q_{j,m}|}{2} \right) \tag{2.5}$$

Equation (2.5) is given in some literature in logarithm form:

$$|R_{m,n}| = \psi \left(\sum_{j \in \mathcal{N}(m) \backslash n} \psi \left(|Q_{j,m}| \right) \right)$$

where the *psi* function is

$$\psi(|x|) \triangleq -\ln \tanh \frac{|x|}{2} = \ln \frac{\left(e^{|x|}+1\right)}{\left(e^{|x|}-1\right)}$$

The key step leading to MSA is to approximate the magnitude (2.5) by

$$|R_{m,n}| = \min_{j \in \mathcal{N}(m) \backslash n} \left(\{ |Q_{j,m}| : j \in \mathcal{N}(m) \backslash n \} \right) \tag{2.6}$$

MSA is described by (2.2), (2.4) and (2.6). In the thesis work, correction with scaling factor [20] [21] is used to reduce performance loss.

2.6.2 Value reuse property

Direct implementation of (2.6) requires large number of adders to find minimum values. The value reuse property [22] [23] [24] [25] [26] [27] [28] [29] states that, when a check node performs update according to (2.6), among all the outgoing magnitudes, there is at most one magnitude which is different from the rest. The outgoing magnitudes can be obtained as follows. Let $\{|Q_{j,m}| : j \in \mathcal{N}(m)\}$ be the set of input magnitudes to check node m. Suppose $|Q_{k_1,m}|$ is a minimum of $\{|Q_{j,m}| : j \in \mathcal{N}(m)\}$, and $|Q_{k_2,m}|$ is a minimum of $\{|Q_{j,m}| : j \in \mathcal{N}(m) \setminus k_1\}$. Denote the two minima by first minimum $m_1 \triangleq |Q_{k_1,m}|$ and second minimum $m_2 \triangleq |Q_{k_2,m}|$, then, the output magnitude given by (2.6) becomes

$$|R_{m,n}| = \begin{cases} m_2 & n = k_1 \\ m_1 & n \neq k_1 \end{cases} \tag{2.7}$$

As an low cost hardware implementation, a check node decoder accepts input message and sends output messages in serial. When it accepts a stream of input messages, the check node decoder compares incoming magnitudes to find m_1, m_2, and k_1. These three values are obtained immediately once all messages are received, and then the outgoing magnitudes can be produced according to (2.7).

2.6.3 Layered decoding

Turbo-decoding message passing (TDMP) presented in [30, 31] is modified and referred as layered decoding in [32]. Alternatively, layered decoding can be considered as a variant of two-phase message passing (TPMP). When applied to QC LDPC codes, the algorithm is summarised as follows.

Iterations start after initialisation

$$\begin{cases} L_n = L_{c,n} & \forall n \in \mathcal{N} \\ R_{m,n} = 0 & \forall m \in \mathcal{M}, \forall n \in \mathcal{N}(m) \end{cases}$$

Each iteration consists of m_b subiterations, corresponding to m_b check node blocks (CNBs), and a CNB is regarded as a layer. Subiterations are performed sequentially from CNB 0 to $m_b - 1$. In i-th subiteration, for any check node m in CNB i, following steps are performed.

1. restore variable-to-check messages, $\forall n \in \mathcal{N}(m)$:

$$Q_{n,m} = L_n - R_{m,n} \tag{2.8}$$

2. check node update, $\forall n \in \mathcal{N}(m)$:

$$R_{m,n} = 2 \tanh^{-1} \left(\prod_{j \in \mathcal{N}(m) \backslash n} \tanh \left(\frac{Q_{j,m}}{2} \right) \right) \tag{2.9}$$

3. update total information, $\forall n \in \mathcal{N}(m)$:

$$L_n = Q_{n,m} + R_{m,n} \tag{2.10}$$

Given QC LDPC code, above computation for a check node in a CNB is independent on any other check node in the same CNB, therefore, all check nodes in a CNB can perform above computation in parallel to increase throughput.

At the end of each iteration, convergence of decoding is checked, and decoding terminates if stop criterion is met, otherwise, another iteration begins.

Compared with TPMP, layered decoding converges faster in terms of number of iterations, requires less amount of memory and wiring. In layered decoding, CNBs decode a codeword sequentially layer by layer. At the end of each subiteration, newly updated C2V messages are used to perform variable node update immediately, and the updated variable-to-check messages are used in

subsequent subiterations. For any variable node V_n, instead of storing all V2C messages $Q_{n,m}, \forall m \in \mathcal{M}(n)$, only the total information L_n is maintained as a running sum, which is updated in every subiteration.

2.7 A partly parallel architecture

Architectures proposed in [33], [34], [35], [36], [27], [37], and [38] are based on the TPMP or its variants. This kind of architectures are good candidates for application-specific integrated circuit (ASIC), but they were not adopted in the thesis work because it is complex to make them support multiple rates and lengths, and they consume more hardware resource. The solution to this thesis work is motivated by architectures presented in [32], [22], [23], [24], [25], [28], [26], and [29], those architectures are based on layered decoding.

A decoder architecture for the example code in Table 2.2 is illustrated in Figure 2.5. Sum memory stores total information L_j^i at data lane j in memory entry address i. The first row of the base matrix in Table 2.3 is $[b(0,0), b(0,1), b(0,2), b(0,3)] = [1, -1, 1, 2]$, therefore, for check node C_0^0, (2.8), (2.9), and (2.10) lead to following equations: variable-to-check messages to C_0^0 are restored as

$$\begin{bmatrix} Q_{1,0}^{0,0} \\ Q_{1,0}^{2,0} \\ Q_{2,0}^{3,0} \end{bmatrix} = \begin{bmatrix} L_1^0 \\ L_1^2 \\ L_2^3 \end{bmatrix} - \begin{bmatrix} R_{0,1}^{0,0} \\ R_{0,1}^{0,2} \\ R_{0,2}^{0,3} \end{bmatrix} \tag{2.11}$$

where check-to-variable messages $R_{0,1}^{0,0}$, $R_{0,1}^{0,2}$, and $R_{0,2}^{0,3}$ are updated in previous iteration. Check-to-variable messages are updated by

$$\begin{bmatrix} R_{0,1}^{0,0} \\ R_{0,1}^{0,2} \\ R_{0,2}^{0,3} \end{bmatrix} = \begin{bmatrix} f\left(Q_{1,0}^{2,0}, Q_{2,0}^{3,0}\right) \\ f\left(Q_{1,0}^{0,0}, Q_{2,0}^{3,0}\right) \\ f\left(Q_{1,0}^{0,0}, Q_{2,0}^{2,0}\right) \end{bmatrix} \tag{2.12}$$

where $f(x,y) \triangleq 2\tanh^{-1}\left(\tanh(x/2) \cdot \tanh(y/2)\right)$. Finally, total information

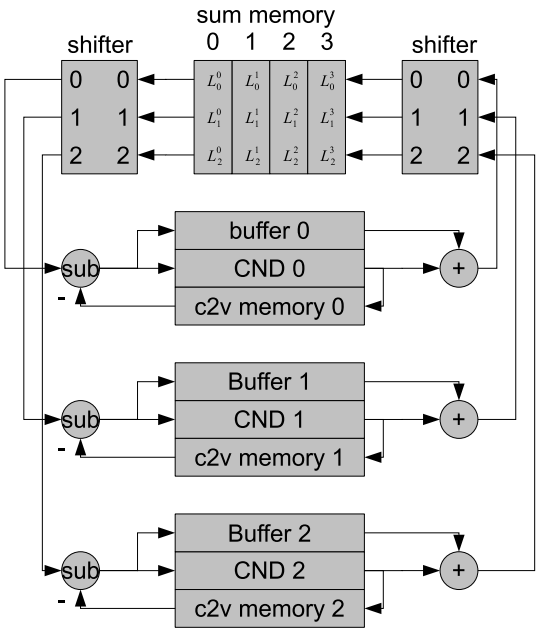

Figure 2.5: Layered decoder: an example architecture

is updated by

$$\begin{bmatrix} L_1^0 \\ L_1^2 \\ L_2^3 \end{bmatrix} = \begin{bmatrix} Q_{1,0}^{0,0} \\ Q_{1,0}^{2,0} \\ Q_{2,0}^{3,0} \end{bmatrix} + \begin{bmatrix} R_{0,1}^{0,0} \\ R_{0,1}^{0,2} \\ R_{0,2}^{0,3} \end{bmatrix} \tag{2.13}$$

Vectors

$$\begin{bmatrix} L_0^0 \\ L_1^0 \\ L_2^0 \end{bmatrix}, \begin{bmatrix} L_0^2 \\ L_1^2 \\ L_2^2 \end{bmatrix}, \text{ and } \begin{bmatrix} L_0^3 \\ L_1^3 \\ L_2^3 \end{bmatrix}$$

are read out serially from address 0, 2, and 3 of sum memory, one vector out of one memory entry per clock cycle. A shifter rotates vectors circularly and upwards, respectively for $b(0,0) = 1$ step, $b(0,2) = 1$ step and $b(0,3) = 2$ steps to obtain

$$\begin{bmatrix} L_1^0 \\ L_2^0 \\ L_0^0 \end{bmatrix}, \begin{bmatrix} L_1^2 \\ L_2^2 \\ L_0^2 \end{bmatrix}, \text{ and } \begin{bmatrix} L_2^3 \\ L_0^3 \\ L_1^3 \end{bmatrix}$$

The topmost elements of the 3 vectors, L_1^0, L_1^2, and L_2^3, are routed to the topmost subtractor in serial to form a data stream. Like the memory access to sum memory, c2v memory 0 is accessed for 3 times, so that old check-to-variable messages $R_{0,1}^{0,0}$, $R_{0,1}^{0,2}$, and $R_{0,2}^{0,3}$ are read out and routed in serial to the same subtractor to form another data stream. The two data streams are synchronised, so that (2.11) can be done correctly at the subtractor. The subtractor's output data stream $Q_{1,0}^{0,0}$, $Q_{1,0}^{2,0}$, and $Q_{2,0}^{3,0}$ enters check node decoder (CND) 0 as well as buffer 0. CND 0 updates check-to-variable messages according to (2.12). Value reuse property introduced in Section 2.6.2 is utilised, the updated messages $R_{0,1}^{0,0}$, $R_{0,1}^{0,2}$ and $R_{0,2}^{0,3}$ are produced at the CND 0's output immediately after $Q_{1,0}^{0,0}$, $Q_{1,0}^{2,0}$, and $Q_{2,0}^{3,0}$ enter into the decoder. $Q_{1,0}^{0,0}$, $Q_{1,0}^{2,0}$, and $Q_{2,0}^{3,0}$ are delayed by buffer 0 in order to get synchronised with CND 0's output stream $R_{0,1}^{0,0}$, $R_{0,1}^{0,2}$ and $R_{0,2}^{0,3}$, and the two streams enter into the adder which produces total information L_1^0, L_1^2, and L_1^3, according to (2.13).

As shown in Figure 2.5, when CND 0 decodes on behalf of check node C_0^0, CND 1 and 2 decode respectively for C_1^0 and C_2^0 in parallel. The parallel operations produce updated total information

$$\begin{bmatrix} L_1^0 \\ L_2^0 \\ L_0^0 \end{bmatrix}, \begin{bmatrix} L_1^2 \\ L_2^2 \\ L_0^2 \end{bmatrix}, \text{ and } \begin{bmatrix} L_2^3 \\ L_0^3 \\ L_1^3 \end{bmatrix}$$

The 3 vectors are shifted to obtain

$$\begin{bmatrix} L_0^0 \\ L_1^0 \\ L_2^0 \end{bmatrix}, \begin{bmatrix} L_0^2 \\ L_1^2 \\ L_2^2 \end{bmatrix}, \text{ and } \begin{bmatrix} L_0^3 \\ L_1^3 \\ L_2^3 \end{bmatrix}$$

before they are written back to sum memory's locations from which the old total information vectors are read out. Writing back updated total information completes one subiteration, and the subsequent subiteration can start immediately.

Because messages enter into, and leaves from, a check node decoder in serial

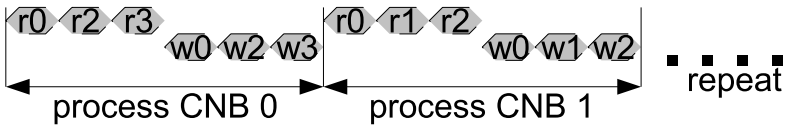

Figure 2.6: Example decoder wave form, non-pipelined

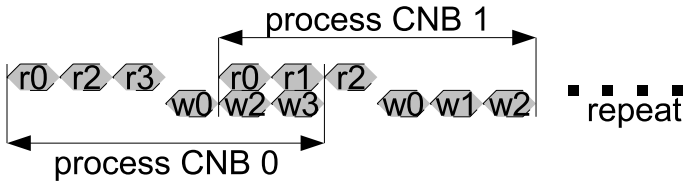

Figure 2.7: Example decoder wave form, pipelined

manner, such a CND is said serial. The overall architecture is said partly parallel due to multiple copies of CNDs. The parallelism factor is 3 in this example, and all 3 check nodes in a CNB are processed simultaneously.

2.7.1 Improve throughput of the example architecture

The decoding throughput of the example architecture can be improved by pipelining [22] and out-of-order memory-write [24].

The behaviour of the decoder in a subiteration can be abstracted as reading sum memory for a number of clock cycles, and then writing sum memory for a number of clock cycles, as shown in Figure 2.6. rn in the figure stands for "read memory address n", wn means "write memory address n", and possible latencies are not shown in the waveform. The throughput can be increased if memory read operation overlaps with preceding memory write operation, as in Figure 2.7. Overlapping is allowed by using dual port memory. In this example, complete overlap is not possible because write operation to address 0 (w0) must precede read operation to address 0 (r0). Check node decoder is made serial and two-stage pipelined, so that memory read stage and memory write stage can operate simultaneously on consecutive check node blocks.

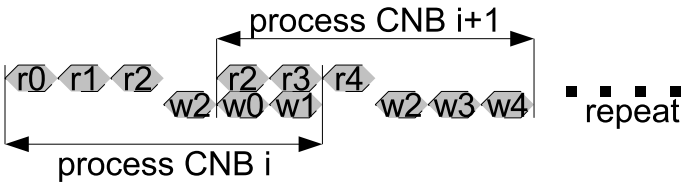

Figure 2.8: Example decoder wave form, pipelined, out-of-order memory-write

Out-of-order memory-write is illustrated by Figure 2.8. In this example, during subiteration i, instead of writing messages back to memory in the order of w0-w1-w2, writing sequence is w2-w0-w1. If the order w0-w1-w2 is in use, no overlap can be achieved because w2 must precede r2.

2.7.2 Multi-size shifter

A number of architectures include shifter. Given a nonzero submatrix $H_{i,j} = (P_z)^p$, where $0 \leq p < z$, z is block size, for $k = 0, 1, \cdots, z-1$, check node C_k^i is connected to variable node $V_{(k+p \mod z)}^j$, and corresponding messages on the edge are $Q_{(k+p \mod z),k}^{j,i}$ and $R_{k,k+p \mod z}^{i,j}$.

Suppose a partly parallel architecture has z check node decoders (CNDs) operating in parallel, like the architecture given in Figure 2.5. Let z total information messages L_k^j, $k = 0, 1, \cdots, z-1$, be stored together in one memory entry j. A memory entry is divided to z data lanes indexed by $0, 1, \cdots, z-1$. Data lane k in memory entry j stores L_k^j, in other words, memory entry with address j stores a column vector

$$\begin{bmatrix} L_0^j \\ L_1^j \\ \vdots \\ L_{z-1}^j \end{bmatrix}$$

The message required by CND k is stored at lane $(k+p \mod z)$, and a

column vector of the messages for

$$\begin{bmatrix} \mathrm{CND}_0 \\ \mathrm{CND}_1 \\ \vdots \\ \mathrm{CND}_{z-1} \end{bmatrix} \text{ is } \begin{bmatrix} L^j_{0+p \mod z} \\ L^j_{1+p \mod z} \\ \vdots \\ L^j_{z-1+p \mod z} \end{bmatrix}$$

which is obtained by shifting

$$\begin{bmatrix} L^j_0 \\ L^j_1 \\ \vdots \\ L^j_{z-1} \end{bmatrix}$$

upwards and circularly for p step. Therefore a shifter is needed.

To handle codes with multiple block sizes, the shifter is required to shift first z messages of a vector of Z messages, where z is the block size of a code, and Z is the maximum of all block sizes. It is common to design block sizes as multiples of the sizes of the smallest block. For instance, in this thesis work, the block sizes are 12, 24, 36, 48 and 96, consequently, a shifter needs to perform circular shift on as many as 96 messages for block size $z = 96$, but for block size $z = 48$, the shifter operates on message $0, 1, \cdots, 47$, while messages in lane $48, 49, \cdots, 95$ are ignored and can be processed in any manner.

In short, the difficulty of making such a shifter steps from the large number of messages to shift, as well as the multi-size requirement.

2.7.3 Remove one shifter

In Figure 2.5 are two shifters: the left one locates in sum memory's memory-read data path, the right one locates in sum memory's memory-write data path.

The shifter in memory-write data path can be removed, and the corresponding shift operation can be compensated by the left shifter in memory-read path. Removal of the shifter reduces latency of decoding data path and improves throughput. This technique is applied in some works such as [26].

Chapter 3

Proposed decoder architecture

In this chapter, the codes to be implemented are first specified, and then the decoding algorithm is formulated, and mapped to proposed hardware architecture using old techniques introduced in Chapter 2, as well as techniques proposed in this thesis work.

3.1 Definition of 15 LDPC codes

The codes to be implemented in this thesis are QC LDPC codes. 3 base matrices are specified respectively for code rate 1/2, 2/3, and 3/4, listed in appendix A. Each base matrix can be expanded to 5 parity check matrices, with block sizes of 12, 24, 36, 48 and 96. The modulo expansion method described in Section 2.5 is used. All parity check matrices have $n_b = 48$ block columns, i.e., every codeword has 48 code bit blocks, hence code lengths are 576, 1152, 1728, 2304, and 4608. Number of block rows, m_b, is 24 for rate 1/2 codes, 16 for rate 2/3, and 12 for rate 3/4. The 15 codes are enumerated in appendix A as code $1, 2, \cdots, 15$.

3.2 Decoding algorithm

Initialisation is regarded as (-1)-th iteration, described as

$$\begin{cases} L_n^{(-1)} = L_{c,n} & \forall n \in \mathcal{N} \\ R_{m,n}^{(-1)} = 0 & \forall m \in \mathcal{M}, \forall n \in \mathcal{N}(m) \end{cases}$$

Decoding proceeds iteration by iteration. An iteration consists of m_b subiterations, executed sequentially from check node block (CNB) 0 to $m_b - 1$. Let (it) be the iteration index, i be the subiteration index, $\mathcal{M}_i$ the set of check nodes in CNB i, and let quantities updated in the it-th iteration be labelled with superscript (it). A subiteration includes following sequential operations.

1. restore variable-to-check messages, $\forall m \in \mathcal{M}_i$, $n \in \mathcal{N}(m)$:

$$Q_{n,m}^{(it)} = L_n^{(it-1)} - R_{m,n}^{(it-1)} \tag{3.1}$$

2. check node update, sign processing and magnitude processing respectively, $\forall m \in \mathcal{M}_i$, $n \in \mathcal{N}(m)$:

$$\operatorname{sign}\left(R_{m,n}^{(it)}\right) = \prod_{j \in \mathcal{N}(m) \backslash n} \operatorname{sign}\left(Q_{j,m}^{(it)}\right) \tag{3.2}$$

$$\left|R_{m,n}^{(it)}\right| = \min_{j \in \mathcal{N}(m) \backslash n} \left(\left\{\left|Q_{j,m}^{(it)}\right| : j \in \mathcal{N}(m) \backslash n\right\}\right) \tag{3.3}$$

3. magnitude correction, down scale the magnitude by factor of $\beta = 0.8$, $\forall m \in \mathcal{M}_i$, $n \in \mathcal{N}(m)$:

$$\left|R_{m,n}^{(it)}\right| = \beta \cdot \left|R_{m,n}^{(it)}\right| \tag{3.4}$$

4. update total information, $\forall m \in \mathcal{M}_i$, $n \in \mathcal{N}(m)$:

$$L_n^{(it)} = Q_{n,m}^{(it)} + R_{m,n}^{(it)} \tag{3.5}$$

substitution of (3.1) into (3.5) gives

$$\begin{aligned} L_n^{(it)} &= L_n^{(it-1)} + \left(R_{m,n}^{(it)} - R_{m,n}^{(it-1)}\right) \\ &= L_n^{(it-1)} + \Delta R_{m,n}^{(it)} \end{aligned} \tag{3.6}$$

it implies that check node update improves total information of every codeword bit iteration by iteration.

Following stop condition is evaluated sequentially at the end of each iteration:

1. make hard decision, $\forall n \in \mathcal{N}$: $\hat{c}_n = \begin{cases} 0 & L_n^{(it)} \geq 0 \\ 1 & L_n^{(it)} < 0 \end{cases}$
2. decoding converges if $H\underline{\hat{c}}^T = 0$, where $\underline{\hat{c}} \triangleq (\hat{c}_0 \hat{c}_1 \cdots \hat{c}_{N-1})$, exit decoding
3. if $it = 20$, decoding fails, exit decoding
4. continue to next iteration: $it = it + 1$

3.3 Proposed design techniques

In addition to the techniques presented in literature and summarised in Chapter 2, some other techniques not found in literature are employed in the thesis work, explained in this section.

3.3.1 Out-of-order memory-read

Out-of-order memory-write introduced in Section 2.7.1 improves throughput. The example given in Figure 2.8 can be further improved by using also out-of-order memory-read, as shown in Figure 3.1. When processing CNB $i+1$, in stead of using read order r2-r3-r4, the order in use is r3-r4-r2, and in this arrangement, the idle cycle in Figure 2.8 can be removed.

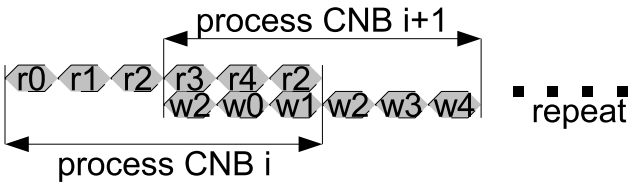

Figure 3.1: Example decoder wave form, pipelined, out-of-order memory-write and -read

If memory entry j is to be accessed in both subiteration i and its subsequent iteration $i+1$, it is written in the beginning of memory write operation in subiteration i, and is read in the end of memory read operation in subiteration $i+1$. This technique is useful because the data path generally has latency of some clock cycles, which is not illustrated in Figure 2.8 and Figure 3.1.

3.3.2 Two-stage shifter

A multi-size shifter is needed, as described in Section 2.7.2. A two-stage shifter is presented in this thesis, illustrated by the following example. Assume block sizes are 3, 6, and 9. The input and output ports of the shifter are 9-message wide, and messages are viewed as a column vector, indexed from top to bottom by $0, 1, \cdots, 8$.

Suppose current block size is $z = 6$, shifting step is $p = 4$, i.e., the shifter is required to shift elements 0 through 5 circularly upwards for 4 positions, and ignores the elements 6 through 8.

The process is shown in Table 3.1. In Table 3.1 (a), column vector $[0, 1, \cdots, 8]^T$ is reshaped to 3 columns as a 3×3 table, with column and row indexed by $0, 1$, and 2. In general, block sizes are multiples of the smallest block size z_{min}, and the number of rows of a table equals to the smallest block size z_{min}. In this example, shifting step is $p = 4$, it is seen that element 4 locates in row $i_p = 1$ and column $j_p = 1$ in Table 3.1 (a). Table 3.1 (b) is obtained by left-shifting all rows in Table 3.1 (a) circularly, however, only the leftmost

column 0
↓ column 1
↓

row 0 →	0	3	6
row 1 →	1	(4)	7
row 2 →	2	5	8

(a)

0	3	×
(4)	1	×
5	2	×

(b)

(4)	1	×
5	2	×
0	3	×

(c)

Table 3.1: Illustration: shifter

z/z_{min} elements in each row are subjected to shift operation when $z < Z$, where Z is the maximum block size, and other elements are ignored, shown as symbol $\times$ in the tables. Shift step for row 0 through $i_p - 1$ is $j_p + 1$, in this example, row 0 is left-shifted circularly for 2 steps. Shift step for the rest of rows is j_p, in this example, row 1 and 2 in Table 3.1 (a) are left-shifted circularly for 1 step. Table 3.1 (c) is obtained by up-shifting all columns in Table 3.1 (b) circularly for j_p steps, i.e., all columns in Table 3.1 (b) are up-shifted circularly for 1 step. Finally, Table 3.1 (c) is reshaped back to a column vector, and the shifter realises the required shifting as

$$[0, 1, 2, 3, 4, 5, \times, \times, \times]^T \rightarrow [4, 5, 0, 1, 2, 3, \times, \times, \times]^T$$

The shifting is done first row-wise and then column-wise, and it is a two-stage shifter.

3.3.3 Iteration termination

Evaluation of stop criterion given in Section 3.2 requires computing $H\hat{\underline{c}}^T$ and compare it with a long zero vector in the end of each iteration. This method leads to high computational complexity and latency.

The decoder implemented in this thesis work checks convergence in serial manner: at the end of subiteration i, new hard decisions for all bits decoded

by CNB i are compared with the old hard decisions produced in previous iteration. Decoder checks if all those decisions are equal, also checks if all decisions satisfy all the parity check constraints in this CNB. A one bit flag is set to 1 if and only if both conditions are true, otherwise cleared to 0. With one flag corresponding to one CNB, m_b flags make up a register, where m_b is the number of CNBs of a code. If all flags in the register are set to 1's in the end of a subiteration, then decoding is regarded converged.

3.4 Decoder core

3.4.1 Operation overview

Main part of decoder architecture is shown as decoder core in Figure 3.2. The implementation is similar to the example given in Section 2.7.

The algorithm described in Section 3.2 is mapped to hardware resources as follows. Total information $L_n, n \in \mathcal{N}$, stored in sum memory, is read out from the memory and rotated by a shifter to get aligned with check node decoders (CNDs). The subtractors implement (3.1) to restore variable-to-check (V2C) messages. A CND implements min-sum algorithm with scaling factor correction, which is described by (3.2), (3.3) and (3.4). The adders implement (3.5) to update total information. Suppose a CND performs computation on behalf of check node $m \in \mathcal{M}$, the V2C messages $Q_{n,m}$, $n \in \mathcal{N}(m)$, are saved in the buffer accompanying that CND for a short while, and then read out synchronously with the check-to-variable (C2V) messages updated by that CND, and the two message streams are routed to the adder next to that CND. Also saved in that buffer are hard decisions of bits $n \in \mathcal{N}(m)$ produced in previous iteration, which are compared with the new decisions based on total information updated in current iteration. The comparison is done by a terminator, which evaluates stop criteria and terminates the decoding process when necessary. The C2V messages updated in current iteration are saved into c2v memory for use in subsequent iterations. In the beginning

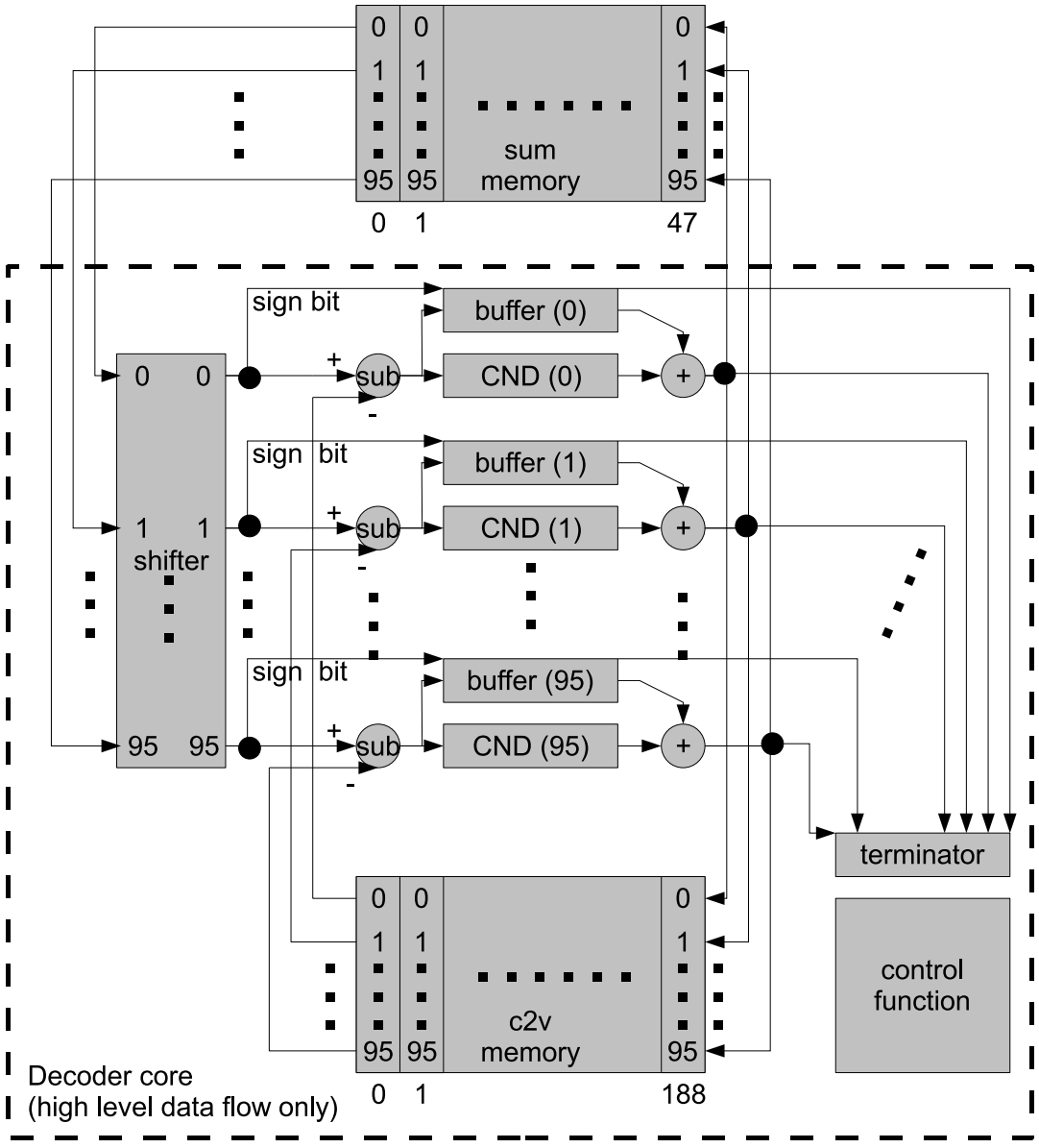

Figure 3.2: Decoder core

L_0^0	L_0^1	$\cdots$	L_0^{47}	row 0
L_1^0	L_1^1	$\cdots$	L_1^{47}	row 1
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
L_{z-1}^0	L_{z-1}^1	$\cdots$	L_{z-1}^{47}	row $z-1$
column 0	column 1	$\cdots$	column 47	

Table 3.2: Total information organised as columns

of current iteration, messages in c2v memory are C2V messages updated in previous iteration.

The maximum of block sizes $\{12, 24, 36, 48, 96\}$ is 96. Each section of the data path in Figure 3.2 operates in parallel on 96 messages. Those messages are viewed as elements of a column vector, indexed from top to bottom by $0, 1, \cdots, 95$. For example, the data ports of sum memory are 96-message wide, so that total information of nodes pertaining to a variable node block (VNB) can be stored in one memory entry, and accessed in parallel in one clock cycle. There are 96 identical subtractors, CNDs, buffers, and so on. If block size z is less than 96, only messages $0, 1, \cdots, z-1$ are under meaningful operation.

Next, each part of the decoder is briefly discussed.

3.4.2 Sum memory

A codeword is divided to 48 blocks. A vector of total information

$$\left[L_0^0, L_1^0, \cdots, L_{z-1}^0, L_0^1, L_1^1, \cdots, L_{z-1}^1, \cdots, L_0^{47}, L_1^{47}, \cdots, L_{z-1}^{47}\right]$$

is reshaped to 48 columns as shown in Table 3.2. The sum memory in Figure 3.2 is 48 entries deep and 96 lanes wide. The memory layout is shown by Table 3.3, in which $m_{i,j}$ denotes memory location at address j and lane i. Table 3.2 is loaded to sum memory by saving L_i^j to memory location $m_{i,j}$,

	address 0	address 1	$\cdots$	address 47
lane 0	$m_{0,0}$	$m_{0,1}$	$\cdots$	$m_{0,47}$
lane 1	$m_{1,0}$	$m_{1,1}$	$\cdots$	$m_{1,47}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
lane 95	$m_{95,0}$	$m_{95,1}$	$\cdots$	$m_{95,47}$

Table 3.3: Sum memory

$0 \leq i \leq z-1$, $0 \leq j \leq 47$. One memory entry stores a message block of 48 messages, when $z < 96$, only message 0 through $z-1$ are valid.

3.4.3 Dual port memory

The sum memory is a simple-dual port memory, with one port dedicated to write access and the other for write access. The two ports operate simultaneously, when read port sends out data for subiteration i, write port is able to receive data of previous subiteration.

Similarly, the c2v memory is also a dual port memory.

3.4.4 Multi-size shifter

In subiteration i, if submatrix $H_{i,j}$ is not a null matrix, the 96 CNDs access message block at address j in sum memory. The topmost z messages in the message vector out of the sum memory are shifted cyclically to get aligned with check node decoders, as described in Section 2.7.2. The two-stage shifter proposed in Section 3.3.2 is implemented as in Figure 3.3. The 96 messages are viewed as elements of Table 3.4. Firstly, each of the 12 rows of the table are shifted, and then, each of the 8 columns are shifted. A register separates row operation and column operation, so that the shifter is two-stage pipelined.

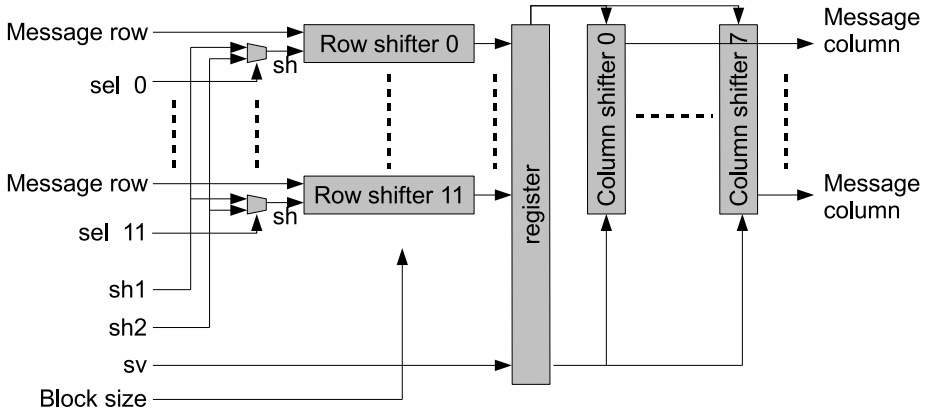

Figure 3.3: Shifter, top level

L_0^j	L_{12}^j	$\cdots$	L_{84}^j	row 0
L_1^j	L_{13}^j	$\cdots$	L_{85}^j	row 1
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
L_{11}^j	L_{23}^j	$\cdots$	L_{95}^j	row 11
column 0	column 1	$\cdots$	column 7	

Table 3.4: A message block viewed as table

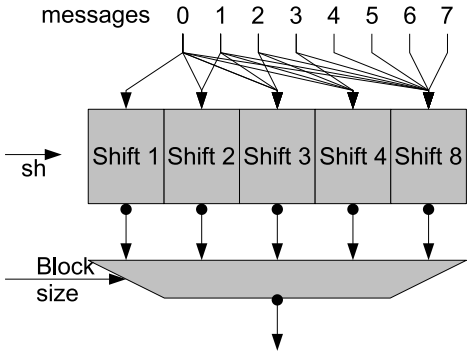

Figure 3.4: Shifter, horizontal shifter

Because the 5 block sizes $\{12, 24, 36, 48, 96\}$ are multiples of 12, only the $(z/12)$ leftmost columns in Table 3.4 are valid. Consequently, only the $(z/12)$ leftmost messages in a message row are subjected to circular shift operation. Figure 3.4 presents structure of a row shifter, which contains 5 subshifters. A subshifter labelled with number n performs leftwards circular shift on the n leftmost messages, ignoring the rest of the messages. Therefore, subshifter 1 is dummy, it lets the leftmost message pass and ignores others, subshifter 2 either lets the 2 leftmost messages pass or swaps them, ignoring the rest, and so forth. The step for shifting is given by signal *sh*, which stands for step for horizontal shift. As shown in Figure 3.3, *sh* is driven by a multiplexer selecting *sh1* or *sh2*, the usage of the two values is described in Section 3.3.2. 12 multiplexers are controlled by signals $sel0, \cdots, sel11$. The column shifters in Figure 3.3 are controlled by signal *sv*, which stands for step for vertical shift.

A column shifter and a subshifters in a horizontal shifter can be implemented as logarithm shifter. Figure 3.5 presents an example of a log shifter. $\log_2 8 = 3$ stages of 2-input multiplexers are needed to implement circular upwards shifter on 8 elements. The bold lines show the path for shift step of 6.

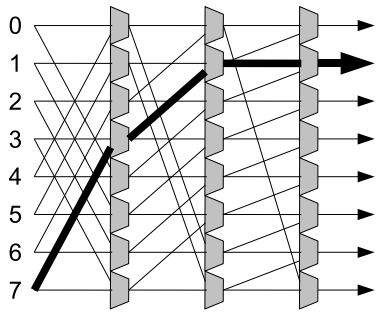

Figure 3.5: Example, log shifter

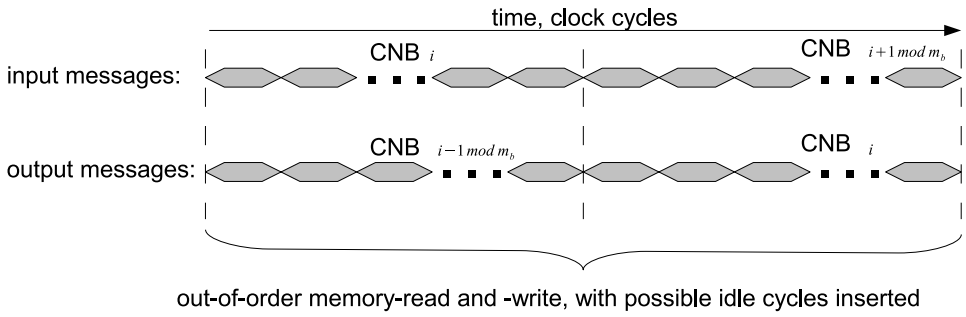

Figure 3.6: Check node decoder, pipelined, sample waveform

3.4.5 Check node decoder (CND)

The main part of the decoder core is a column of 96 CNDs, indexed from top to bottom by $k = 0, 1, \cdots, 95$ in Figure 3.2. During subiteration i, CND k performs check node update on behalf of check node C_k^i. A CND is serial (Section 2.7), pipelined (Section 2.7.1), using out-of-order memory-write (Section 2.7.1) and out-of-order memory-read (Section 3.3.1). Figure 3.6 presents sample waveforms to illustrate the behaviour of a CND, and Figure 3.7 presents its structure, which is explained next.

A counter in control block generates indices $0, 1, \cdots, \rho_i - 1$ for incoming messages, where ρ_i is degree of check nodes in CNB i. Messages are integers, taking two's complement format outside of CNDs, and sign-magnitude format inside a CND. A converter at input converts a message from two's

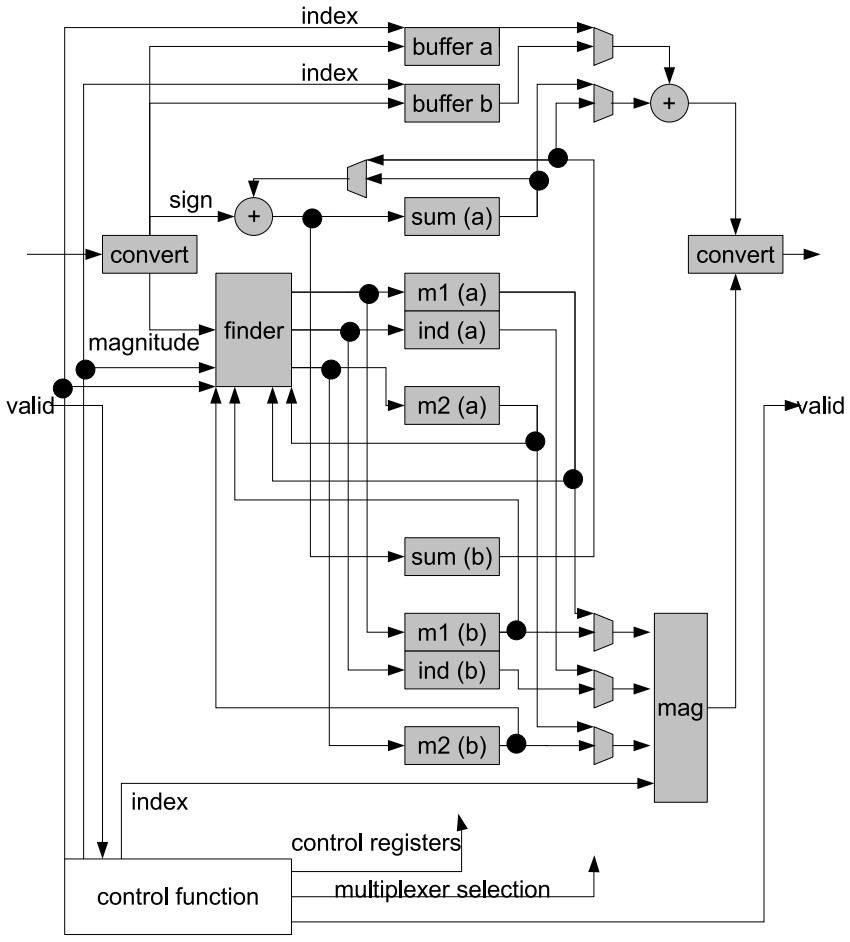

Figure 3.7: Check node decoder (CND) structure

complement format to sign-magnitude format, and another converter does the converse at output. A column of registers in the middle of Figure 3.7 are grouped to set (a) and set (b). When one set is used to process incoming messages, the other can be used to produce outgoing messages pertaining to previous subiteration. First minimum m_1, second minimum m_2, and index ind for the first minimum (refer to Section 2.6.2), are hold in registers named m1, m2 and ind, respectively.

Magnitude processing

At the beginning of each subiteration, register m1 and m2 are initialised to the largest values they can hold . As messages enters into a CND one by one, those registers are updated by the finder module. Let mag be the current magnitude at finder's input, ind be its index, m_1 and m_2 be the current values in the respective registers, and 3 cases are enumerated:

$$\begin{cases} \text{case A:} & mag \leq m_1 \\ \text{case B:} & m_1 < mag \leq m_2 \\ \text{case C:} & mag > m_2 \end{cases}$$

In case A, mag is saved to m1 register, m_1 is saved to m2 register, and ind is saved to ind register; in case 2, mag is saved to m2 register, and the other two registers are unchanged; in case 3, all registers are unchanged. The registers are used to produce C2V messages as described in (2.7), which is reproduced below

$$\text{output magnitude of message with index } j \text{ is: } \begin{cases} m_2 & j = ind \\ m_1 & j \neq ind \end{cases}$$

where the index $j = 0, 1, \cdots, \rho_i - 1$ is driven by the control block, as shown in Figure 3.7.

Sign processing

Equation (3.2) can be written as

$$\text{sign}\left(R_{m,n}^{(it)}\right) = \text{sign}\left(Q_{j,n}^{(it)}\right) \prod_{j \in \mathcal{N}(m)} \text{sign}\left(Q_{j,m}^{(it)}\right)$$

The sign function returns real value ± 1. In two's complement representation, a sign bit of 1 indicates negative number, and 0 indicates non-negative number. Product of ± 1 in the equation are translated to binary sum over binary field $\{0, 1\}$, implemented by XOR logic operator in hardware. When messages step into a CND one by one, their sign bits are saved in a buffer memory as shown in Figure 3.7, indices of messages serve as addresses to the buffer. The sum register in Figure 3.7 is a 1-bit flipflop, which is cleared to zero at the beginning of each subiteration, and then stores the running sum of sign bits. To produce the sign bit for output check-to-variable message with index j, the sign bit saved in buffer memory at address j is read out and added to the value in sum flip flop to produce the required sign bit. Addition is done by the adder in the upper-right corner in Figure 3.7.

Out-of-order memory access

Address sequences for all memory access in the decoder can be determined by inspecting base matrices in design time, as described in Section 2.7.1 and Section 3.3.1. Address sequences are saved in on-chip memories.

3.5 Overall architecture with dual buffer

It takes time to initialise the sum memory in Figure 3.2, and also takes time to output decoded result from the memory to decoder user. A dual buffer configuration can hide the time and improve throughput. Figure 3.8 presents decoder core equipped with dual buffers. When one memory is in decoding

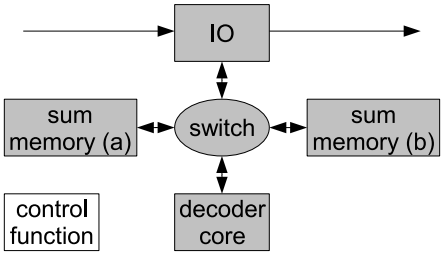

Figure 3.8: Decoder with dual buffer

mode, it is connected with decoder core, which reads out and writes back data to this memory. Meanwhile, the other memory can operate in input-output (IO) mode, it is connected with IO control block, which sends out decoded data to decoder user, or accepts next received codeword data to initialise the memory.

For the reason illustrated in Section 2.7.3, the decoder core presented in Figure 3.2 includes only one shifter. Consequently, the IO block shown in Figure 3.8 includes another shifter, which pre-shifts messages before iteration starts during input mode and post-shifts messages during output mode. Control bits supplied to this shifter is stored in memory.

The target Xilinx FPGA chip provides Block SelectRAM memory and Distributed SelectRAM memory [39]. Most memories in the decoder are built with the former memory type, and the buffers in Figure 3.2 are built with the latter type. The Xilinx block memory can be configured to dual port mode.

A number of memories in the decoder are used to store memory access addresses. Details of memory addressing is out of scope of thesis writing. The basic idea is that, the code structure is reflected by memory address sequences, and it is possible to modify those address sequences so that the implementation can adapt to other codes, as long as the new codes are not far different.

3.6 Fixed-point issues

Most parts of data path in Figure 3.2 are two's complement integers represented by 5 bits, and total information messages are 8 bits wide. A subtractor has two outputs, the message entering into a buffer is 8 bits wide, and the other going to a CND is 5 bits wide. The adder output is 8 bits wide. Subtractor and adder outputs are clipped.

The input data to the hardware decoder are LLR values which are also coded in format of two's complement integers. As shown in Figure 2.1, the received signal $\underline{r}$ is assumed as real valued signal, and consequently LLRs are also real valued. Quantisation is needed to convert real valued LLR values to two's complement integers with finite digits. An interval is determined, and real valued LLR values falling into this interval are quantised to 5 bits integers. Real valued LLRs outsides the interval are clipped to the boundary values of the interval. The selected interval in simulation is

$$\left[-2/\sigma^2 - 3 \times (2/\sigma), 2/\sigma^2 + 3 \times (2/\sigma)\right]$$

where σ^2 is noise power. Interval of this size covers over 99% LLR values.

Decoding performance is related to size of the quantisation interval, as well as number of bits of each section of data path in the hardware. MATLAB fixed-point simulation determines input quantisation interval and widths of sections of data path. Detailed discussion of fixed-point modelling and simulation is omitted due to lack of space.

Chapter 4

Results

The architecture proposed in Chapter 3 was implemented in the FPGA device. The implementation results are reported in this chapter.

4.1 Introduction to design flow

Implementing demanding algorithms to hardware is a complex procedure. This section presents a simplified design flow consisting of most important steps only. Detailed discussion is out of the scope. The hardware design work flow is generally iterative, one needs revert to earlier steps if result of current step does not meet requirement.

1. The thesis work starts with understanding design requirement. Codes are specified, system requirements are formulated, target hardware device is studied.

2. During literature review, various algorithms are studied, a variety of hardware architecture presented in literature are summarised.

3. Next, algorithms is chosen and draft hardware architecture is made. This is the step requiring creativity and experience. Decoding throughput and consumed hardware resource are roughly estimated to check if the architecture satisfies system requirement.

4. Fixed-point modelling is done for the drafted architecture. Number of bits for each variable is determined by fixed-point simulation. Simulation also validates the error correction performance, and the result should be checked against system requirement. One must revert to earlier steps to modify architecture, or even change to another algorithm, if current one does not meet the system requirement. This step is done with MATLAB.

5. Following MATLAB simulation, a detailed hardware implementation specification is made. The overall hardware entity is partitioned to sub blocks. Interfaces of blocks and their communications between blocks are formulated, and documented with diagrams and text.

6. VHDL coding is done according to the specification. VHDL stands for very-high-speed-integrated-circuit hardware description language, which is used to describe functionality of a digital hardware entity.

7. VHDL models are simulated and debugged with ModelSim software.

8. VHDL model is verified by simulation using reference data. Reference data are obtained from MATLAB simulation of the fixed-point MATLAB model. Output of VHDL simulation must agree with output of MATLAB simulation.

9. The task of generating hardware net list from VHDL model is called synthesis. Net list is a low level description of digital electronics hardware, specifying what elementary hardware resource are used and how they are connected. For example, a net list may specifies how logical gates are connected. The synthesis software utilised in this thesis work is Synplify Pro. Given synthesis result, more accurate estimation such

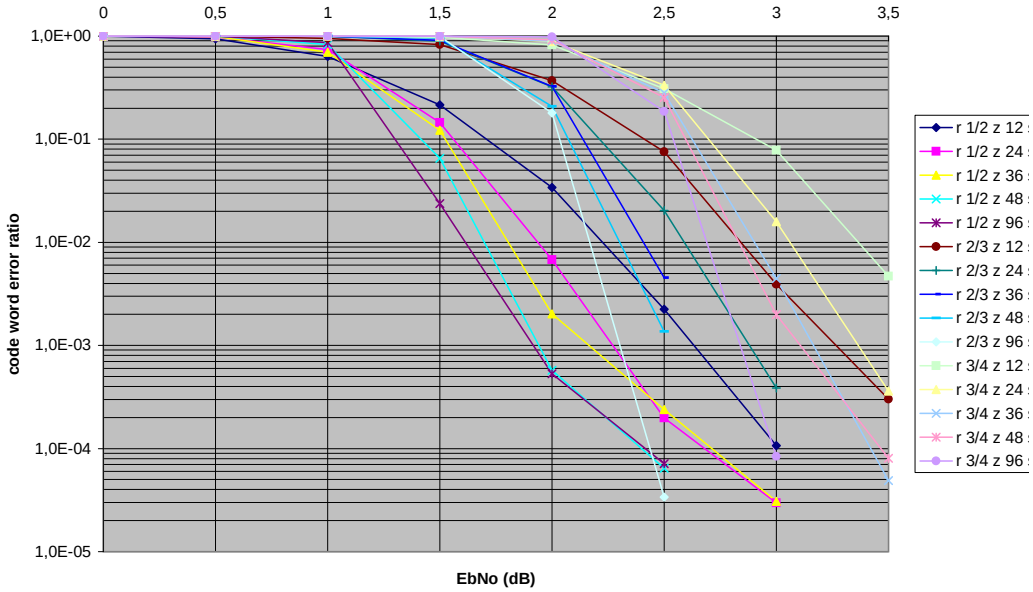

Figure 4.1: Error correction performance

as hardware size and decoding throughput can be obtained. Optimisation within synthesis step can improve the result to certain degree. One must revert to early steps if the result does not meet requirement.

10. There are other tasks left in a complete implementation flow. For example, the procedure of generating final physical layout from net list is called place and routing, and post-synthesis optimisation. The wide sense synthesis also includes this step. This step is not emphasised in the thesis work, as it is out of the scope of the thesis.

4.2 Error correction performance

Fixed-point model was developed and simulated using MATLAB. It is not presented in this thesis due to lack of space. Error correction performance is validated by MATLAB fixed-point simulation. The curves are shown in Figure 4.1. 15 codes are presented in the figure. For example, legend "r 1/2 z 12" refers to a code of rate 1/2 and block size 12.

The simulation collects 20 error codewords at each signal to noise ratio (SNR) point. SNR is measured in dB of E_b/N_0, where E_b is energy per information bit, and $N_0/2 = \sigma^2$. The performance is measured in codeword error ratio, rather not bit error ratio.

4.3 VHDL design, verification, and synthesis

The implementation is described by multiple files in very-high-speed-integrated-circuit hardware description language (VHDL [40]). VHDL codes are written in fully synthesizable register transfer level (RTL). VHDL files are not presented in this thesis due to lack of space.

The VHDL design is verified by ModelSim simulation using testing vectors captured from MATLAB fixed-point simulation. It is verified that output from VHDL model and MATLAB fixed-point simulation are exactly the same.

XC2VP50-5F1152 [39, 41], a Xilinx Virtex II Pro FPGA chip, was selected as target device. Trail synthesis of the VHDL design was performed using Synplify Pro with default setting. It is estimated that about half of logic resource on XC2VP50-5F1152 chip is consumed, and clock frequency is around 70 MHz.

4.4 Decoding throughput

For the proposed architecture in this thesis work, throughput varies depending on channel noise, code rate, and codeword length. When SNR is low, more iterations are needed and throughput is low. When the number of decoding iterations and codeword length are fixed, code with higher rate produces higher throughput. When the number of decoding iterations and code rate are fixed, code with larger block size produces higher throughput.

	block size (codeword length)				
	12 (576)	24 (1152)	36 (1728)	48 (2304)	96 (4608)
rate 1/2	0.0755	0.1511	0.2266	0.3021	0.6042
rate 2/3	0.1011	0.2023	0.3034	0.4045	0.8091
rate 3/4	0.1119	0.2237	0.3356	0.4474	0.8949

Table 4.1: Decoding throughput (decoded bits per clock cycle, 20 iterations)

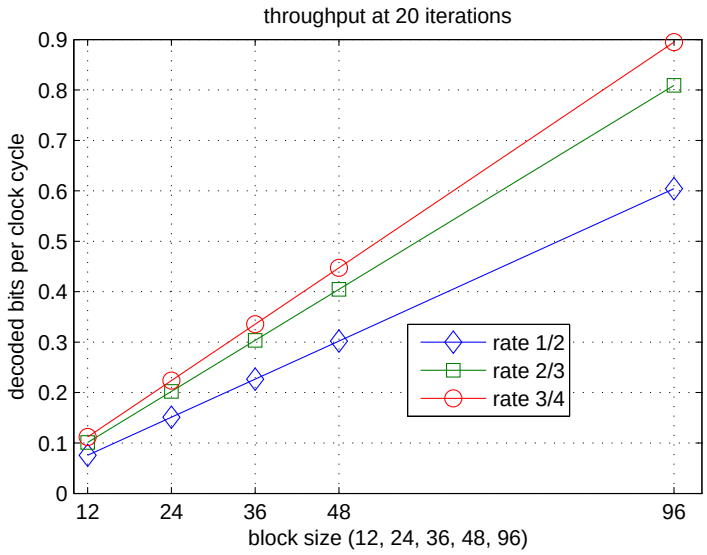

Figure 4.2: Decoding throughput

20 iterations are performed in the worst case. Number of clock cycles to run 20 iterations are counted in ModelSim simulation of the VHDL designs: codes of rate 1/2 require 3813 clock cycles, rate 2/3 codes require 3797 clock cycles, rate 3/4 codes require 3862 clock cycles. Throughput at 20 iterations is the ratio of number of information bits of a codeword over number of clock cycles for 20 iterations. Throughput for 15 codes at 20 iterations, measured in decoded information bits per clock cycle, are presented in Table 4.1, drawn in Figure 4.2. In this worst case, highest throughput is obtained by rate 3/4 code of block size 96, it is about 0.9 information bits per cycle; lowest throughput is about 0.075 information bits per clock cycle obtained by rate 1/2 code of block size 12. Other rates and block sizes lead to throughput in

between 0.075 bits/cycle and 0.9 bits/cycle.

The throughput increases when the number of iterations drops if channel is good. For instance, when iteration number drops from 20 to 10, the throughput will be doubled to reach 0.15 to 1.8 bits per cycle.

Assuming 70 MHz clock frequency, the worst case throughput at 20 iteration translates to throughput ranging from 5.2 Mbits to 63Mbits (information bits) per second. If the number of iterations drop to 10 from 20 due to good channel quality, the throughput will be doubled to reach 10 Mbits to 120 Mbits. Average throughput of a given code depends on the average number of iterations as well as the code parameters.

Chapter 5

Conclusion

The results presented in Chapter 4 are discussed in this chapter. The result are evaluated, limit and future work are pointed out.

The objective of the thesis work are met successfully. The consumed hardware resource is within the design constraint. The worst case throughput at 20 decoding iterations is 0.075 to 0.9 information bits per clock cycle. Corresponding throughput at 70 MHz clock frequency is 5.2 Mbits to 63 Mbits per second. The decoder is real-time configurable such at any of the 15 codes can be decoded. Comparing performance curves in Figure 4.1 and reference figures in appendix B shows that performance degradation is acceptably small. The design can be further improved for smaller hardware size and higher throughput.

The class of LDPC codes is large, and there has not been a universal hardware decoder suitable for many applications. The decoder architecture presented in this thesis is constrained by limited amount of FPGA resource, and throughput is sacrificed to trade for hardware resource. The architecture assumes codes like those defined in appendix A, and it is not likely to fit to other codes which are far different.

Optimisation for speed and hardware cost have not yet been highlighted in the thesis work. Improvements can be done in following aspects. re-

synthesise VHDL files with higher optimisation effort and apply synthesis techniques. Currently it is synthesised only with default settings. Insert registers into critical path can increase clock frequency. However, More effective approach is to modify part of the logic design. For example, handshaking is used in the decoding data path. In fact, the values of handshaking signals can be determined in design time, therefore, the handshaking can be removed. The removal shortens current critical path significantly, thus increases clock frequency and throughput significantly. A second example is to reduce amount of memory. For check node C_m of degree d_m, in current implementation, all d_m check-to-variable (C2V) messages updated by C_m are saved in c2v memory. Those messages can be compressed by applying the value reuse property (Section 2.6.2) as in [29], so that large amount of memory can be saved. Reduction in memory reduces hardware size and increases clock frequency and throughput. Thirdly, some inefficient logic design can be corrected to save hardware and increase clock speed. For example, the shifter in IO block performs post-shifting (refer to Section 3.5), and the needed control bits are saved in a memory block in decoding core, rather not locally in IO block. Each time decoding core writes messages to an entry of sum memory, it reads out control bits from its local memory and writes them together with messages to the target memory entry in sum memory. In a more efficient way, the storage of those control bits can be moved to IO block, so that extra memory and data movement can be avoided

In summary, a partly parallel LDPC decoder hardware architecture was designed and implemented successfully in this thesis work. Various design techniques are reviewed in the thesis, and some new techniques are proposed. Thesis objective are met, and further improvements have been pointed out. The architecture can be applied to similar codes. The implementation can be reused for other codes which are not far different, by merely updating contents in read-only-memories (ROMs). LDPC codes will be widely adopted, research and development on implementing decoders continue. This thesis provides useful information for practical design tasks.

Appendix A

Base matrices for 15 LDPC codes

The base matrices for codes of rate 1/2, 2/3, and 3/4 are given in Figure A.1, A.2, and A.3, respectively. All 15 codes, corresponding to block sizes of 12, 24, 36, 48, and 96, can be obtained by modulo expansion method described in Section 2.5.

The 15 codes are enumerated in the following list. For example, the first line reads as: code 1 is a $(n, k) = (576, 288)$ code of rate $r = 1/2$ with block size $z = 12$ and codeword length N=576.

1. (576, 288) code: rate 1/2, block size 12, word length 48x12= 576
2. (1152, 576) code: rate 1/2, block size 24, word length 48x24= 1152
3. (1728, 864) code: rate 1/2, block size 36, word length 48x36= 1728
4. (2304, 1152) code: rate 1/2, block size 48, word length 48x48= 2304
5. (4608, 2304) code: rate 1/2, block size 96, word length 48x96= 4608
6. (576, 384) code: rate 2/3, block size 12, word length 48x12= 576
7. (1152, 768) code: rate 2/3, block size 24, word length 48x24= 1152

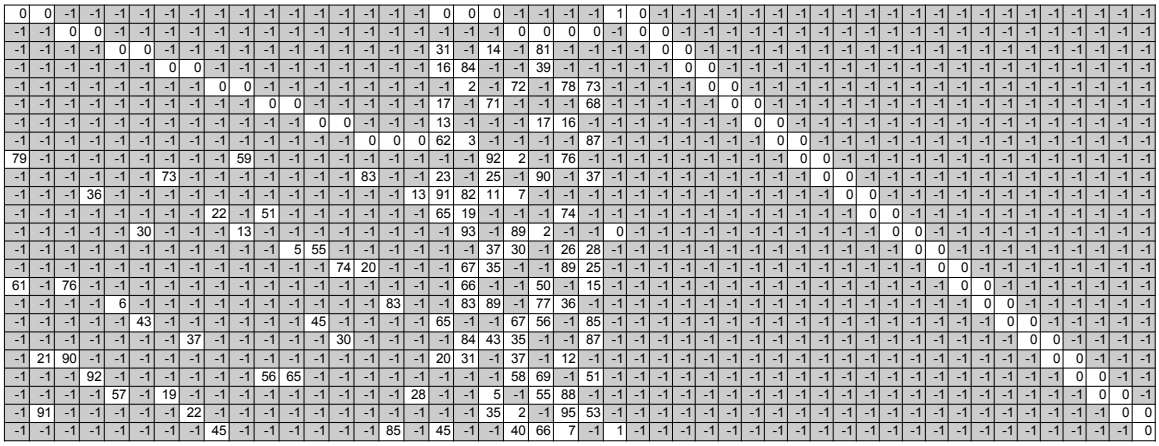

Figure A.1: Base matrix, rate 1/2

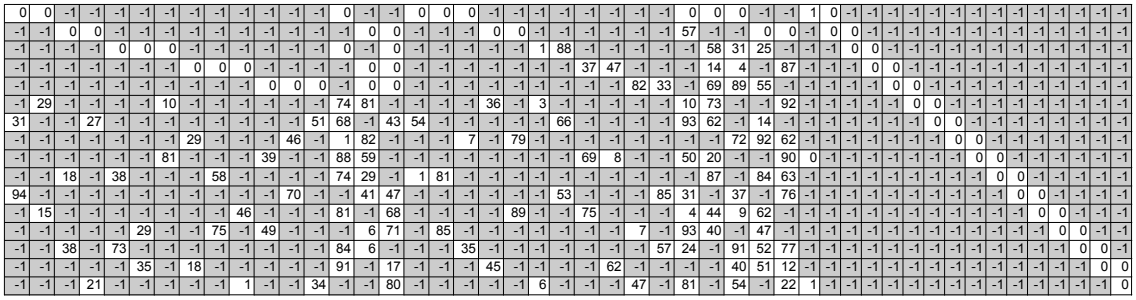

Figure A.2: Base matrix, rate 2/3

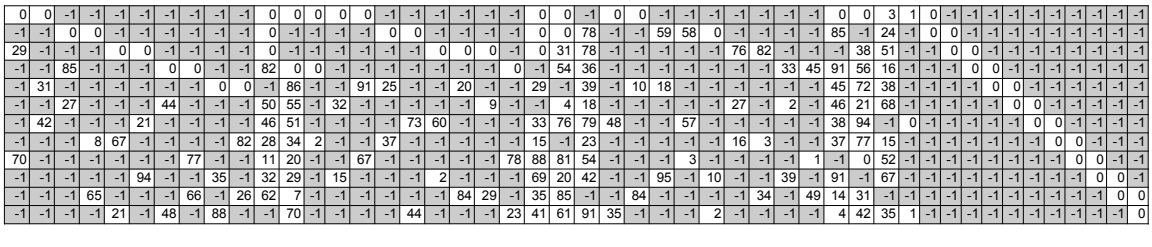

Figure A.3: Base matrix, rate 3/4

8. (1728, 152) code: rate 2/3, block size 36, word length 48x36= 1728
9. (2304, 1536) code: rate 2/3, block size 48, word length 48x48= 2304
10. (4608, 3072) code: rate 2/3, block size 96, word length 48x96= 4608
11. (576, 472) code: rate 3/4, block size 12, word length 48x12= 576
12. (1152, 864) code: rate 3/4, block size 24, word length 48x24= 1152
13. (1728, 1296) code: rate 3/4, block size 36, word length 48x36= 1728
14. (2304, 1728) code: rate 3/4, block size 48, word length 48x48= 2304
15. (4608, 3456) code: rate 3/4, block size 96, word length 48x96= 4608

Appendix B

Reference performance

Figure B.1, B.2, and B.3 are provided as decoding performance reference, they are given by project partner designing the codes. AWGN channel and BPSK modulation are used. Decoding is done by standard believe propagation algorithm with 50 iterations, and the data are obtained through floating point simulation. The hardware implementation uses less optimal algorithm and is subjected to fixed-point effect, hence the performance of hardware implementation is degraded. The figures can be found in page 65 and 66 of [42].

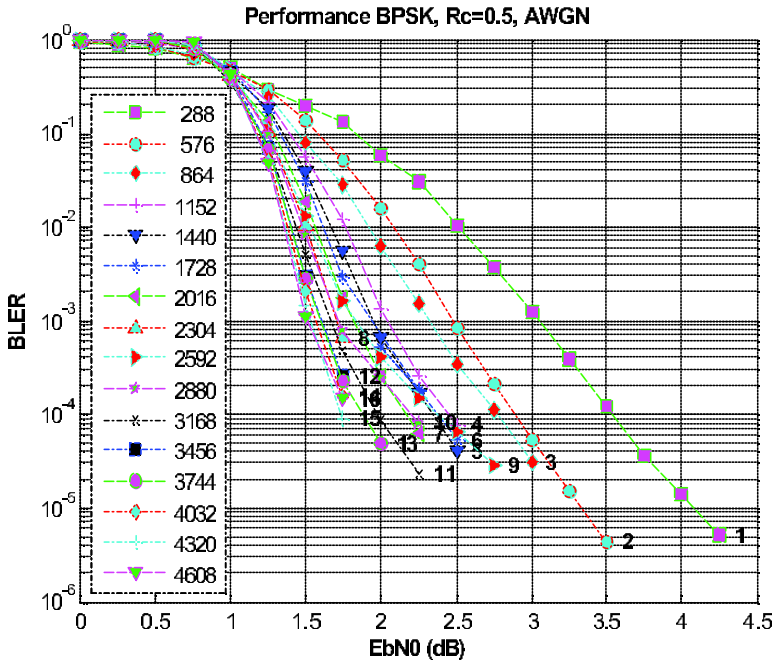

Figure B.1: Codeword error ratio, rate $\frac{1}{2}$, BPSK, AWGN

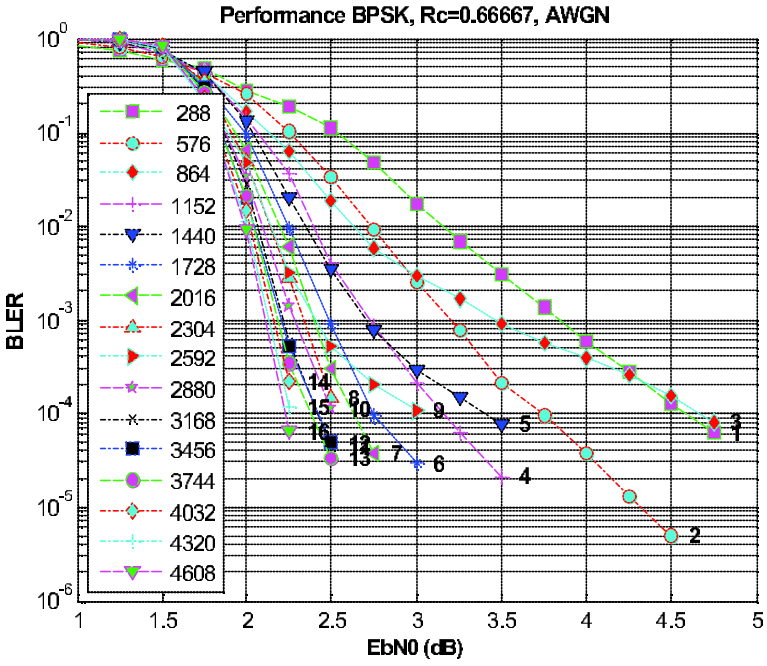

Figure B.2: Codeword error ratio, rate $\frac{2}{3}$, BPSK, AWGN

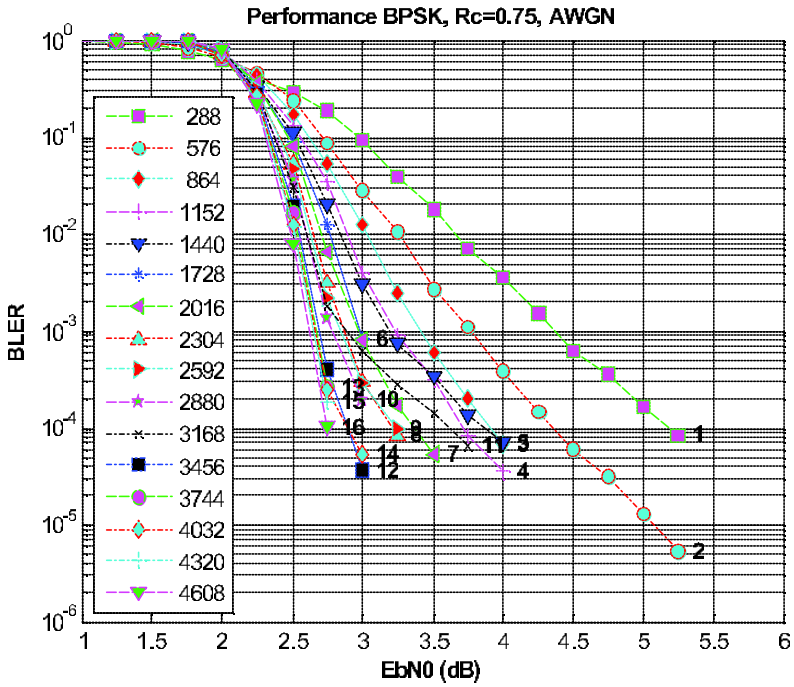

Figure B.3: Codeword error ratio, rate $\frac{3}{4}$, BPSK, AWGN

Bibliography

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

[2] G. D. Forney and D. J. Costello, "Channel Coding: The Road to Channel Capacity," *Proceedings of the IEEE*, vol. 95, no. 6, pp. 1150–1177, June 2007.

[3] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *IEEE International Conference on Communications ICC 93. Geneva. Technical Program, Conference Record*, vol. 2, 23–26 May 1993, pp. 1064–1070.

[4] R. G. Gallager, "Low-density parity-check codes," Ph.D. dissertation, MIT, Cambridge, MA, 1963.

[5] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 33, no. 6, pp. 457–458, 13 March 1997.

[6] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, March 1999.

[7] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 619–637, Feb 2001.

[8] S.-Y. Chung, J. Forney, G. D., T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Communications Letters*, vol. 5, no. 2, pp. 58–60, Feb 2001.

[9] K. Gracie and M. H. Hamon, "Turbo and Turbo-Like Codes: Principles and Applications in Telecommunications," *Proceedings of the IEEE*, vol. 95, no. 6, pp. 1228–1254, June 2007.

[10] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, March 2002.

[11] R. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, Sep 1981.

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

[13] F. R. Kschischang, B. J. Frey, and H. A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, Feb 2001.

[14] G. D. Forney, "Codes on graphs: normal realizations," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 520–548, Feb 2001.

[15] K. S. Andrews, D. Divsalar, S. Dolinar, J. Hamkins, C. R. Jones, and F. Pollara, "The Development of Turbo and LDPC Codes for Deep-Space Applications," *Proceedings of the IEEE*, vol. 95, no. 11, pp. 2142–2156, Nov. 2007.

[16] D. E. Hocevar, "LDPC code construction with flexible hardware implementation," in *Proc. IEEE International Conference on Communications ICC '03*, vol. 4, 11–15 May 2003, pp. 2708–2712.

[17] H. Zhong and T. Zhang, "Design of VLSI implementation-oriented LDPC codes," in *Proc. VTC 2003-Fall Vehicular Technology Conference 2003 IEEE 58th*, vol. 1, 6–9 Oct. 2003, pp. 670–673.

[18] ——, "Block-LDPC: a practical LDPC coding system design approach," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 52, no. 4, pp. 766–775, April 2005.

[19] M. P. C. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Transactions on Communications*, vol. 47, no. 5, pp. 673–680, May 1999.

[20] J. Chen, A. Dholakia, E. Eleftheriou, M. P. C. Fossorier, and X.-Y. Hu, "Reduced-Complexity Decoding of LDPC Codes," *IEEE Transactions on Communications*, vol. 53, no. 8, pp. 1288–1299, Aug. 2005.

[21] J. Chen and M. P. C. Fossorier, "Near optimum universal belief propagation based decoding of low-density parity check codes," *IEEE Transactions on Communications*, vol. 50, no. 3, pp. 406–414, March 2002.

[22] M. Karkooti, P. Radosavljevic, and J. R. Cavallaro, "Configurable, High Throughput, Irregular LDPC Decoder Architecture: Tradeoff Analysis and Implementation," in *Proc. International Conference on Application-specific Systems, Architectures and Processors ASAP '06*, Sept. 2006, pp. 360–367.

[23] Y. Sun, M. Karkooti, and J. R. Cavallaro, "High Throughput, Parallel, Scalable LDPC Encoder/Decoder Architecture for OFDM Systems," in *Proc. IEEE Dallas/CAS Workshop on Design, Applications, Integration and Software*, Oct. 2006, pp. 39–42.

[24] K. K. Gunnam, G. S. Choi, W. Wang, E. Kim, and M. B. Yeary, "Decoding of Quasi-cyclic LDPC Codes Using an On-the-Fly Computation," in *Proc. Fortieth Asilomar Conference on Signals, Systems and Computers ACSSC '06*, Oct. 29 2006–Nov. 1 2006, pp. 1192–1199.

[25] K. K. Gunnam, G. S. Choi, and M. B. Yeary, "A Parallel VLSI Architecture for Layered Decoding for Array LDPC Codes," in *Proc. th International Conference on VLSI Design Held jointly with 6th International Conference on Embedded Systems*, 6–10 Jan. 2007, pp. 738–743.

[26] K. Gunnam, W. Wang, G. Choi, and M. Yeary, "VLSI Architectures for Turbo Decoding Message Passing Using Min-Sum for Rate-Compatible Array LDPC Codes," in *Proc. 2nd International Symposium on Wireless Pervasive Computing ISWPC '07*, 5–7 Feb. 2007igital Object Identifier 10.1109/ISWPC.2007.34 2007.

[27] Z. Wang and Z. Cui, "A Memory Efficient Partially Parallel Decoder Architecture for Quasi-Cyclic LDPC Codes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 4, pp. 483–488, April 2007.

[28] K. Gunnam, G. Choi, W. Wang, and M. Yeary, "Multi-Rate Layered Decoder Architecture for Block LDPC Codes of the IEEE 802.11n Wireless Standard," in *Proc. IEEE International Symposium on Circuits and Systems ISCAS 2007*, 27–30 May 2007, pp. 1645–1648.

[29] K. K. Gunnam, G. S. Choi, M. B. Yeary, and M. Atiquzzaman, "VLSI Architectures for Layered Decoding for Irregular LDPC Codes of WiMax," in *Proc. IEEE International Conference on Communications ICC '07*, 24–28 June 2007, pp. 4542–4547.

[30] M. M. Mansour and N. R. Shanbhag, "Turbo decoder architectures for low-density parity-check codes," in *Proc. IEEE Global Telecommunications Conference GLOBECOM '02*, vol. 2, 17–21 Nov. 2002, pp. 1383–1388.

[31] ——, "High-throughput LDPC decoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 976–996, Dec. 2003.

[32] D. E. Hocevar, "A reduced complexity decoder architecture via layered decoding of LDPC codes," in *Proc. IEEE Workshop on Signal Processing Systems SIPS 2004*, 2004, pp. 107–112.

[33] Y. Chen and D. Hocevar, "A FPGA and ASIC implementation of rate 1/2, 8088-b irregular low density parity check decoder," in *Proc. IEEE Global Telecommunications Conference GLOBECOM '03*, vol. 1, 1–5 Dec. 2003, pp. 113–117.

[34] M. Karkooti and J. R. Cavallaro, "Semi-parallel reconfigurable architectures for real-time LDPC decoding," in *Proc. International Conference on Information Technology: Coding and Computing ITCC 2004*, vol. 1, 2004, pp. 579–585.

[35] Y. Chen and K. K. Parhi, "Overlapped message passing for quasi-cyclic low-density parity check codes," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 6, pp. 1106–1113, June 2004.

[36] Z. Wang and Z. Cui, "Low-Complexity High-Speed Decoder Design for Quasi-Cyclic LDPC Codes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 1, pp. 104–114, Jan. 2007.

[37] Z. Cui and Z. Wang, "Efficient Message Passing Architecture for High Throughput LDPC Decoder," in *Proc. IEEE International Symposium on Circuits and Systems ISCAS 2007*, 27–30 May 2007, pp. 917–920.

[38] X.-Y. Shih, C.-Z. Zhan, C.-H. Lin, and A.-Y. Wu, "An 8.29 mm2 52 mW Multi-Mode LDPC Decoder Design for Mobile WiMAX System in 0.13 um CMOS Process," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 3, pp. 672–683, March 2008.

[39] Xilinx, *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, March 2007.

[40] P. J. Ashenden, *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 1995.

[41] Xilinx, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 2007.

[42] S. Stiglmayr, "IST-4-027756 WINNER II D2.2.1-v1.0 Joint Modulation and Coding Procedures," Tech. Rep., 2006. [Online]. Available: http://www.ist-winner.org/index.html