Field-Programmable Gate-Array (FPGA) Implementation of Low-Density Parity-Check (LDPC) Decoder in Digital Video Broadcasting – Second Generation Satellite (DVB-S2)

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Electrical and Computer Engineering
University of Saskatchewan
Saskatoon, Saskatchewan, Canada

By
Kung Chi Cinnati Loi

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Department of Electrical and Computer Engineering
University of Saskatchewan
57 Campus Drive
Saskatoon, Saskatchewan
Canada, S7N 5A9

Abstract

In recent years, LDPC codes are gaining a lot of attention among researchers. Its near-Shannon performance combined with its highly parallel architecture and lesser complexity compared to Turbo-codes has made LDPC codes one of the most popular forward error correction (FEC) codes in most of the recently ratified wireless communication standards. This thesis focuses on one of these standards, namely the DVB-S2 standard that was ratified in 2005.

In this thesis, the design and architecture of a FPGA implementation of an LDPC decoder for the DVB-S2 standard are presented. The decoder architecture is an improvement over others that are published in the current literature. Novel algorithms are devised to use a memory mapping scheme that allows for 360 functional units (FUs) used in decoding to be implemented using the Sum-Product Algorithm (SPA). The functional units (FU) are optimized for reduced hardware resource utilization on a FPGA with a large number of configurable logic blocks (CLBs) and memory blocks. A novel design of a parity-check module (PCM) is presented that verifies the parity-check equations of the LDPC codes. Furthermore, a special characteristic of five of the codes defined in the DVB-S2 standard and their influence on the decoder design is discussed.

Three versions of the LDPC decoder are implemented, namely the 360-FU decoder, the 180-FU decoder and the hybrid 360/180-FU decoder. The decoders are synthesized for two FPGAs. A Xilinx Virtex-II Pro family FPGA is used for comparison purposes and a Xilinx Virtex-6 family FPGA is used to demonstrate the portability of the design. The synthesis results show that the hardware resource utilization and minimum throughput of the decoders presented are competitive with a DVB-S2 LDPC decoder found in the current literature that also uses FPGA technology.

Acknowledgements

The project is supported by the Natural Sciences and Engineering Research Council of Canada and SED Systems Inc., a division of Calian Ltd., Saskatoon, SK, Canada. I would like to thank my supervisor, Dr. Seok-Bum Ko, for his support in this project. In the beginning of my M.Sc. program, he gave me a lot of freedom in the topic selection, and supported and guided me in every decision that I made along the way. Even when he was away he constantly touched base with me to make sure everything is going well. Without Dr. Ko's support, this project would not have been possible. I would also like to thank my supervisor at SED Systems, Mr. Dave Armstrong, and all other SED Systems employees for their help while I researched at SED Systems. Special thanks for Prof. Dave Dodds and Mr. Dennis Akins for setting up the project with SED Systems. Thanks to all the professors at the University of Saskatchewan who have supported me and their instruction, help and inspiration during my studies. Also, thanks to Dr. J. C. Lo for his input on the PCM design. Finally, a big thanks to my parents, friends and family for their moral support throughout the years.

Contents

List of Tables

List of Figures

List of Abbreviations

APSK - Amplitude and Phase-Shift Keying
ASIC - Application-Specific Integrated Circuit
AWGN - Additive White Gaussian Noise
BCH - Bose, Chaudhuri and Hocquenghem codes
BPSK - Binary Phase-Shift Keying
BS - Barrel Shifter
BSC - Binary Symmetric Channel
BRAM - Block Random-Access Memory
CLB - Configurable Logic Block
dB - decibel
DVB-C2 - Digital Video Broadcasting – Second Generation Cable
DVB-T2 - Digital Video Broadcasting – Second Generation Terrestrial
DVB-S2 - Digital Video Broadcasting – Second Generation Satellite
FEC - Forward Error Correction
FIFO - First-In First-Out
FPGA - Field-Programmable Gate-Array
FF - Flip-Flop
FU - Functional Unit
GPU - Graphics Processing Unit
HDL - Hardware Description Language
IC - Integrated Circuit
IP - Intellectual Property
IRA - Irregular Repeat-Accumulate
Kb - Kilobit
LDPC - Low-Density Parity-Check
LLR - Log-Likelihood Ratio
LOF - List of Figures
LOT - List of Tables
LUT - Look-Up Table
LRP - Least Reliable Position
LSB - Least Significant Bit
Mb - Megabit
MBWA - Mobile Broadband Wireless Access
MRIP - Most Reliable Independent Position
MRP - Most Reliable Position
MLD - Maximum Likelihood Decoding
MSB - Most Significant Bit
OC - Ones Counter
PCM - Parity Check Module
PER - Packet Error Rate
PSD - Power Spectral Density

PSK - Phase-Shift Keying
PWL - Piece-Wise Linear
QPSK - Quadrature Phase-Shift Keying
RAM - Random-Access Memory
RMSE - Root Mean Square Error
ROM - Read-Only Memory
SNR - Signal-to-Noise Ratio
VHDL - VHSIC Hardware Description Language
VHSIC - Very-High-Speed Integrated Circuit
WLAN - Wireless Local Area Network
WPAN - Wireless Personal Area Network
XOR - Exclusive-OR

Chapter 1

Introduction

In digital data transmission or storage systems, messages transmitted or stored often go through a channel or storage medium that introduces noise that may corrupt the original message. Forward error correction (FEC) codes were introduced in order to solve this problem. In the case of a data transmission communication system, an encoder is introduced at the transmitter to encode the message bits by adding redundancy to the message. This redundancy is transmitted to the receiver along with the message. At the receiver, the received message is decoded in hopes of correcting the errors that may have been introduced during the transmission through the channel and retrieving the original message. According to Shannon's theorem [1], no matter how noisy the communication channel is, there exists an error correction code that can make the probability of error arbitrarily small provided that the transmission rate is less than the Shannon limit. Over the years, researchers have been developing different kinds of codes to increase the transmission rate, in hopes to reach the channel capacity as described by Shannon. In recent years, one of the most successful types of codes in doing so has been LDPC codes.

LDPC codes were originally introduced by Gallanger [2] in the 1960s. However, due to the lack of an efficient decoding algorithm and subpar hardware capabilities, the codes were not widely used at the time and slowly faded away. In the 1990s, LDPC codes were rediscovered and were shown to have performance close to the Shannon limit [3]. In addition, the encoding and decoding process is much less complex in LDPC codes compared to Turbo codes [4], another code that has shown to perform close to the Shannon limit. Furthermore, LDPC codes have highly parallel code structures which are extremely suitable for FPGA implementation. Due to its advantages, it was adopted by many standards to be used for FEC such as Digital Video Broadcasting – Second Generation Satellite (DVB-S2),

Digital Video Broadcasting – Second Generation Cable (DVB-C2), and Digital Video Broadcasting – Second Generation Terrestrial (DVB-T2) [5], wireless local area network (WLAN) air interface (802.11), wireless personal area networks (WPAN) (802.15), broadband wireless metropolitan area network (802.16), and mobile broadband wireless access (MBWA) networks (802.20), among others.

Field-programmable gate-array (FPGA) is an integrated circuit (IC) consisting of logic circuit elements that can be configured by the user after the IC is fabricated, as opposed to application-specific integrated circuits (ASICs), where the users' logic circuits are configured prior to fabrication. FPGAs are usually programmed using hardware description languages (HDL), such as VHDL or Verilog. The advantages of the FPGAs compared to ASIC is in its flexibility to be re-programmed without the need to re-fabricate the IC, which allows for faster turn-around time for hardware designers. For example, design faults can be fixed by simply fixing the programming code, re-synthesizing the design, generating all the programming files and re-programing the FPGA, as opposed to submitting the updated design for fabrication. Furthermore, an existing FPGA design can be implemented for a different target FPGA by simply re-synthesizing the existing design for the new target FPGA, which makes FPGAs very portable. For these reasons among others, FPGA designs have gained much attention in recent years.

This thesis presents a FPGA implementation of a LDPC decoder in the DVB-S2 standard. The architecture of the LDPC decoder in this thesis is a combination and improvement of some of the designs published in the current literature. These designs are reviewed in the next section.

1.1 Literature Review of DVB-S2 LDPC Decoders

Since the adoption of LDPC codes into the DVB-S2 standard in 2005 [6], researchers have been working on designing efficient implementations of an LDPC decoder that is compliant with the standard.

The algorithm used for decoding LDPC codes is a message passing algorithm, where messages, which are real numbers, are passed between two sets of nodes, called bit nodes

and check nodes, through a code rate-specific interconnection network. These messages are updated at the nodes by performing a mathematical calculation. A more detailed description of the decoding algorithm is presented in Section 2.3.

One of the challenges in implementing the LDPC decoder for the DVB-S2 standard is its large block length, or frame size. In the DVB-S2 standard, the block length, N, of the LDPC codes is either 64800 bits, called normal frames, or 16200 bits, called short frames. For high throughput, which means a high number of message bits are decoded per unit of time, the LDPC decoders can be designed with a fully parallel architecture, such as the one by Blanksby and Howland [7]. In the fully parallel architecture of the LDPC decoder, N bit node functional units (FUs) are connected to $N-K$ check node FUs, where K is the number of bits in the transmitted message, through a network of interconnections. A more detailed discussion on FUs is presented in Section 3.3. However, even with a 1024-bit block length, such as the decoder by Blanksby and Howland [7], the routing of the interconnections between the FUs is already cumbersome, not to mention the even larger block lengths in the DVB-S2 standard. Furthermore, N bit node FUs and $N-K$ check node FUs need to be implemented. Thus, a fully parallel architecture of the time is not practical to be used by the LDPC decoders for the DVB-S2 standard. On the other hand, in a fully serial architecture, where only one FU is implemented to perform all $N+N-K$ calculations, a very large memory must be used to store all temporary values updated at the nodes and the throughput of the decoder becomes extremely low. Therefore, a partially parallel architecture is best suited for the implementation of the LDPC decoders for the DVB-S2 standard. In 2006, Eroz et al. [8] present a memory architecture that allows for the usage of 360 FUs in the decoder design. In the paper, the authors explain how the interconnection between the bit and check nodes is mapped to memory and how the memory is accessed for processing.

The first known hardware decoder design compliant with the DVB-S2 standard is published in 2005 by Kienle et al. [9]. In their decoder design, 360 FUs are implemented, but it decodes only the normal frame code rates and not the short frame code rates. Furthermore, the authors have optimized the message passing algorithm, such that instead of updating all the messages in the bit node before passing all the messages to the check nodes for processing, and vice versa, some bit nodes are processed as soon as some of the check nodes

messages are updated and vice versa, which reduces processing time. In addition, the design is implemented using ASIC technology.

Subsequently, many other ASIC decoder designs have been published based on the decoder by Kienle et al. [9]. In 2005, Urard et al. [10] present a decoder that uses 360 FUs and supports both normal frames and short frames, yet does not support all the short frame code rates in the DVB-S2 standard. In 2006, Dielissen et al. [11] propose a decoder that uses fewer FUs by further subdividing the calculations at the nodes. The authors also use a modified algorithm, in which the node update calculations are simplified, called the min-sum algorithm. However, their decoder also only handles normal frame code rates. Segard et al. [12] use a different decoding scheduling, called horizontal shuffle scheduling, where multiple bit and check nodes are updated in one step. In 2007, Masera et al. [13] present a decoder that supports the DVB-S2, 802.11n and 802.16 standards, but only 32 FUs are used, so the throughput of the decoder when used for the DVB-S2 standard is very low. Brack et al. [14] present a decoder that uses 90 FUs and only decodes normal frame code rates. In 2009, Zhang et al. [15] and Ying et al. [16] also use modified versions of the min-sum algorithm.

In addition to ASIC designs, some hardware designs have been implemented on FPGAs. In 2005, Yadav and Parhi [17] have proposed LDPC codes different from the ones that are used in the standard [6]. However, they were not able to implement all the code rates into one decoder due to memory limitations at the time and the paper only discusses normal frame code rates. In 2007, Gomes et al. [18] have presented a decoder that uses 180, 90 or 45 FUs. The paper presents a method that can reduce the number of FUs to factors of 360 without the need to increase the amount of memory utilization. The architecture of the decoder is the most similar to the one presented in Chapter 3, so its synthesis results and throughput are used for comparison in Chapter 4. In 2008, Beuschel and Pfleiderer [19] designed a LDPC decoder that supports any LDPC code up to the block length of 65536 bits. However, it only uses 16 FUs, which deteriorates the throughput to only about 75 Mbps.

In the implementation of the FUs, one of the main concerns is the approximation of the

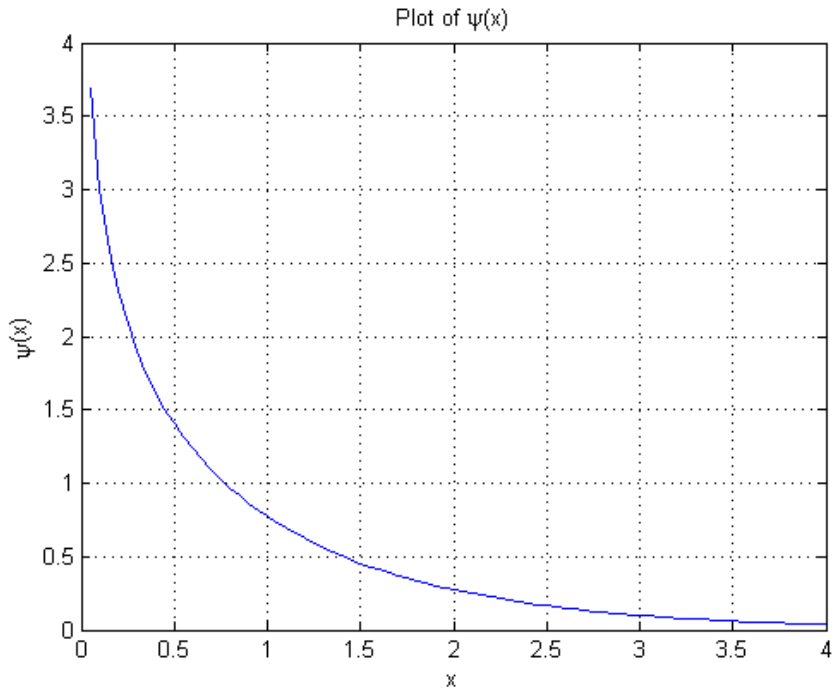

Figure 1.1: Graph of the ψ function.

ψ function defined as follows:

$$\psi(x) = -\ln\left(\tanh\left|\frac{x}{2}\right|\right) = \ln\left(\frac{(1+e^{-|x|})}{(1-e^{-|x|})}\right) \tag{1.1}$$

and the graph of the equation is shown in Figure 1.1. One of the approximation approaches is to use a look-up table (LUT) where an input value is mapped into an output value. However, using a uniform step size for the input values generates wasted storage in the LUT because at high input values the slope of the ψ function graph is close to zero. In 2001, Zhang et al. [20] propose the use of a variable precision quantization scheme for the inputs and outputs of the ψ function. In this scheme, if the value is less than 1, then its most significant bit (MSB) is 0, and the rest of the bits are fractional; if the value is greater than or equal to 1, then the MSB is 1, the decimal point is between the 3rd and 4th MSB, and the integer part is interpreted as (n is the number of bits):

$$v_{n-1} \cdot \overline{v}_{n-2} \cdot \overline{v}_{n-3} \cdot 2^2 + v_{n-2} \cdot 2 + v_{n-3} \tag{1.2}$$

The authors show that the use of 6 magnitude bits for the messages between the bit node and the check nodes provides reasonable trade-off between hardware complexity and

performance. Furthermore, the proposed variable precision quantization scheme improves error performance by about 0.1 dB compared to a uniform quantization scheme.

In 2006, Oh and Parhi [21] propose a different variable quantization scheme for the ψ function using LUTs. Their quantization scheme is based on the uniform $(q : f)$ quantization scheme, where q is the total number of bits, including the sign bit, and f is the number of bits in the fractional part of the value. For input values, x, below decimal number 1.0, the $(q : f)$ uniform quantization scheme is used for their outputs values. If the input values are 1.0 or above, then its output values use a quantization scheme given as follows:

$$\begin{array}{lll} (q, f) & \text{for} & 0 < x < 2^{(q-f-3)} \\ (q-1, f-1) & \text{for} & 2^{(q-f-3)} \leq x < 2^{(q-f-2)} \\ (q-2, f-2) & \text{for} & 2^{(q-f-2)} \leq x < 2^{(q-f-1)} \end{array} \tag{1.3}$$

The proposed quantization scheme reduces the LUT size by 50% compared to the uniform $(q : f)$ quantization scheme. Furthermore, the authors propose further reduction of the LUT size to 75% by using the following quantization scheme instead of (1.3):

$$\begin{array}{lll} (q-1, f-1) & \text{for} & 0 < x < 2^{(q-f-3)} \\ (q-2, f-2) & \text{for} & 2^{(q-f-3)} \leq x < 2^{(q-f-2)} \\ (q-3, f-3) & \text{for} & 2^{(q-f-2)} \leq x < 2^{(q-f-1)} \end{array} \tag{1.4}$$

Using (1.4) to approximate the ψ function reduces the LUT size to only have $2^{(q-3)}$ entries, which means the input of the LUT is $q-3$ bits. Thus, Oh and Parhi propose a compression function that reduces the messages sent between the bit nodes and the check nodes to only have $q-3$ bits. Furthermore, the authors show that the performance loss of using the LUT reduction schemes presented is less than 0.05 dB.

Aside from using LUTs to approximate the ψ function, Masera et al. [22] have proposed two other approximation techniques: *a*) piece-wise linear (PWL) approximation and *b*) a direct implementation of base 2 formulation (PSI2). The PWL approximation uses linear equations to approximate sections of the ψ function. The coefficients of the linear equations are selected to easily implemented with shift and add operations. The PSI2 approximation changes the base of the logarithmic and exponential functions in (1.1) to base 2 and uses state-of-the-art binary logarithmic arithmetic units to approximate the ψ function. Both of

the techniques show to have no more than 0.1 dB performance loss compared to the infinite precision case, but the implementation of these techniques in hardware is more complex than using the LUT approximations. Other approximations can also be found in the literature based on different approximations of the ψ function, different approximations of the node update equations and different scheduling techniques, i.e. the order that the bit node and check nodes are updated. A comparison and analysis of these algorithms is published by Papaharalabos et al. [23].

1.2 Motivation

In all the LDPC decoder designs discussed in Section 1.1, only the design of the FUs and the memory mapping schemes are shown. These units perform the updates at the bit and check nodes and the message passing between the nodes. However, no publication presents the architecture of a module that is used to verify the parity-check equations to perform hard-decision decoding. More information about parity-check equations is presented in Section 2.2. Furthermore, most of the decoders only support normal frames in the DVB-S2 standard. Some of the ones that support short frames omit some code rates defined in the standard.

Additionally, most of the decoder designs are implemented using ASICs, and only a few use FPGAs. One reason may be that ASIC implementations give the designers the freedom to use as many hardware resources as it is necessary to implement the decoder, as opposed to FPGAs that have a limited number of hardware resources fabricated with the device that is available to the users. However, FPGA designs gives the design increased portability and faster design turn-around time compared to ASIC designs, as discussed earlier in this chapter. Thus, in this thesis, the main design goal of the LDPC decoder is to reduce hardware resource utilization, such that the design can be implemented using FPGAs. One way to reduce hardware utilization is by reducing the number of FUs, such as the decoder by Beuschel and Pfleiderer [19], but using fewer FUs corresponds to lowering throughput. Therefore, the other design goal of the decoder in this thesis is not to reduce throughput drastically in the process of reducing hardware resource utilization.

This research project is done in collaboration with SED Systems[1], who have expressed a high interest in an FPGA implementation of the DVB-S2 LDPC decoder. SED Systems currently use commercial ASIC decoders in use in their DVB-S2 receivers. However, implementing the LDPC decoder section of the receiver for use on an FPGA can facilitate system debugging and increase the portability of the decoder to other systems. Some existing commercial decoders are sold as system-on-chip ASICs, which include the complete DVB-S2 receiver design [24], or as devices in a chassis [25, 26]. In some situations, the functionalities of these complete receiver solutions are not applicable, which makes these products very difficult, if not impossible, to integrate with other components.

Furthermore, the complete receiver solutions have a limited throughput. In some situations, if a higher throughput is required that is not supported by the complete receiver solutions, the receiver must be re-designed. However, using an FPGA solution, multiple decoders can be instantiated and executed in parallel to increase throughput. In other situations when lower throughput is sufficient, an FPGA implementation can be easily modified to reduce throughput by for example, reducing the number of FUs. Thus, the FPGA implementation provides the flexibility to implement "special" DVB-S2 receivers that the complete receiver solutions cannot accommodate.

There are also software solutions available for LDPC decoders, such as the MATLAB built-in functions *dvbs2ldpc* [27] and *fec.ldpcdec* [28], yet the large block lengths of the LDPC codes defined in the DVB-S2 standard makes the throughput of the software solutions very low. Furthermore, the low number of processors in current computer architectures does not allow software implementations to efficiently take advantage of the parallel structure of LDPC codes.

Moreover, currently, the two world leading FPGA suppliers, Xilinx [29] and Altera [30], only distribute FPGA intellectual property (IP) Core for DVB-S2 LDPC encoders and not for decoders. However, some FPGA IP designs can be purchased from smaller independent suppliers, such as Navtel Systems [31], SoftJin [32] and RAD3 Communications [33].

Even though the architecture of the decoders presented in this thesis targets the LDPC decoder in the DVB-S2 standard, the soon-to-be ratified DVB-C2 and DVB-T2 standards

[1]SED Systems, a Division of Calian Ltd., Saskatoon, SK, Canada

also adopt LDPC codes for FEC with almost the same structures as the ones in the DVB-S2 standard. Thus, the LDPC decoders described herein may be extended to include DVB-C2 and DVB-T2 standard LDPC codes in the future.

1.3 Description of the Problem and Major Contributions

Based on the discussion in Section 1.2, the problem with the existing DVB-S2 LDPC decoders is that the published designs are incomplete, as no module is presented to verify the parity-check equations and many of the designs only handle normal frames, and not short frames. The existing designs are also less flexible for the end users as they are implemented using ASIC technology. However, implementing the design using FPGAs requires optimizations for the decoder architecture to reduce hardware resource utilization as hardware resources are limited in FPGAs. Furthermore, there are FPGA decoder designs that reduce hardware resources by reducing the number of FUs used, but also reduce the throughput, which is not desired. The objective of this thesis is to improve the DVB-S2 LDPC decoder designs by combining the decoder architectures published in the current literature and is a proposed solution for the aforementioned problems. This is verified by comparisons with other implementations for correctness and performance.

The main issue that originally challenged SED Systems in implementing the DVB-S2 LDPC decoder in FPGA is the memory size. Originally, the idea is to implement two RAMs for message exchange, where one RAM would store the messages temporarily, while the other is used for computing, which might consume too many memory resources on the FPGA. However, using the decoder architecture and memory organization proposed by Eroz et al. [8], the memory required by the decoder to handle all code rates in the DVB-S2 standard is only approximately 2 Mb, which can be easily accommodated by most modern FPGAs. Nevertheless, the authors do not clearly indicate the algorithm that is used to map the interconnection network between the bit and check nodes to the RAM. Thus, a novel algorithm is devised and presented in Section 3.2.1 to perform such mapping.

Another architecture that is improved upon is the FU architecture by Gomes et al. [34]. The components of the FU are modified in order to reduce hardware resource utilization of the FPGA, while maintaining a competitive decoding throughput. The ψ function and adders are used instead of the boxplus and boxminus units. More details on these improvements are presented in Section 3.3. Furthermore, the decoder by Gomes et al. [18] is used for comparison in Chapter 4 as previously mentioned.

Section 1.2 has indicated that none of the published decoders in the current literature present a module that verifies the parity-check equations. Thus, a novel module is designed, called the parity-check module (PCM), and presented in this thesis to perform the verification of the parity-check equations. Section 3.4 shows that the operation of the PCM is very similar to the operation of the DVB-S2 LDPC encoder. Thus, the architecture of the PCM is based on the DVB-S2 LDPC encoder architecture by Gomes et al. [35].

Furthermore, as mentioned in Section 1.2, the short frame code rates are not implemented in many of the designs in current literature. Among the ones that do, not all short frame code rates are supported or the details of the implementation are not clear. One of the reasons may be that there are some code rates in the short frame, denoted as special short frame code rates in this thesis, have a characteristic that changes the memory organization and memory mapping of the decoder. These code rates are discussed in more detail in Section 3.2.4.

1.4 Organization of Thesis

The subsequent chapters of the thesis are organized as follows: Chapter 2 reviews some background information, including the architecture of the target FPGA, the encoding and decoding of linear block codes, and the encoding and decoding of LDPC codes in the DVB-S2 standard. Chapter 3 presents the architecture and implementation of the designed LDPC decoder, the details of each component of the decoder, and the modifications on the decoder architecture to create two other decoders designs. Chapter 4 presents the synthesis results and minimum throughput of the decoders on the target FPGAs and their comparison with the decoder designed by Gomes et al. [18], and the simulation results of the decoder are also presented. Chapter 5 concludes the thesis and suggests potential future work.

Chapter 2

Background Information

2.1 Architecture of Target FPGA

The DVB-S2 LDPC decoder design that is shown in Chapter 3 is implemented on two FPGAs: Xilinx Virtex-II Pro XC2VP100 and Xilinx Virtex 6 XC6VLX240T. In this section, a brief overview of the architecture of these two FPGAs is presented, in order to give some insight into some of the design decisions made in Chapter 3 and to help understand the synthesis results in Chapter 4. The information in this section and more information about the architecture of these two FPGAs can be obtained from Xilinx datasheets and user guides [36, 37, 38, 39].

The Xilinx XC2VP100 FPGA is a Virtex-II Pro device. In Virtex-II Pro FPGAs, configurable logic blocks (CLBs) are used to realize combinational and sequential logic designs. CLBs are arranged in arrays in an FPGA. A CLB is made up of four slices organized in two columns, with two slices on each column, with local feedback within the CLB. Each slice consists of two 4-input function generators, two storage elements, wide function multiplexers, carry logic and arithmetic logic gates.

Each of the two 4-input function generators can be used as a 4-input look-up table (LUT), among other functionalities. Each of the two function generators have four independent inputs and can be used to realize any 4-input Boolean function, and the propagation delay is independent of the function implemented. The output of each function generator can drive an output of the slice, the input of the XOR dedicated gate, the input of the carry-logic multiplexer, the D input of the storage element, or the input of a multiplexer. There is also logic within a slice that is capable of combining the 4-input functions generators to provide

functions of five, six, seven or eight inputs, or selected functions of nine inputs.

The storage elements in the slice can be configured as either edge-triggered D-type flip-flops or level-sensitive latches. The D input can be driven by either the output of the function generators or directly by the input of the slice. For control, other than the clock input, there are also the clock enable and the set and reset inputs, which can be configured to be synchronous or asynchronous.

The function generators and multiplexers in the Virtex-II Pro FPGAs can be configured to implement multiplexing functionalities and the resources utilized are as follows:

- 2:1 multiplexer in one LUT
- 4:1 multiplexer in one slice
- 8:1 multiplexer in two slices
- 16:1 multiplexer in one CLB (four slices)
- 32:1 multiplexer in two CLBs (eight slices)

There are other functions that the multiplexers can be used for, but the discussion of these functionalities are beyond the scope of this thesis.

The dedicated lookahead carry logic allows for faster arithmetic addition and subtraction calculations. Each CLB has two separate carry chains. The arithmetic logic has an XOR gate that allows for a 2-bit full adder to be implemented within a slice.

For large memory needs, the Virtex-II Pro FPGAs have a large amount of 18 Kb block SelectRAM+ (BRAM) resources. Each BRAM can be configured to be single-port RAM, single-port ROM, dual-port RAM or dual-port ROM, where the ROMs are essentially RAMs without write ports. Each BRAM can also be configured to have one of the following dimensions:

- 16 Kb configurations:
  - 16K $\times$ 1 bit
  - 8K $\times$ 2 bits
  - 4K $\times$ 4 bits
- 18 Kb configurations:

- 2K × 9 bits
- 1K × 18 bits
- 512 × 36 bits

where the first value is the depth of the memory and the second value is the width of the data. Multiple BRAMs can be combined to implement memory deeper or wider memories. In the single-port configuration, each BRAM memory has access to either 18 Kb or 16 Kb memory locations depending on the configuration, it is synchronous and the input and output data bus widths are identical.

In the dual-port configuration, each port of a BRAM that accesses a common 18 Kb memory location, is synchronous and has independent control signals. The data width of each port can be configured independently. The two ports have separate inputs and outputs and have independent clock inputs.

For the target Xilinx Virtex-II Pro XC2VP100 FPGA, the total available logic and memory resources are as follows:

Number of Slices: 44096
Number of 4-Input LUTs: 88192
Number of Slice Flip-Flops: 88192
Total Number of 18 Kb BRAMs: 444

The Xilinx XC6LX240T belongs to the Virtex-6 FPGA family, which is a group Xilinx's state-of-the-art FPGA devices. Virtex-6 FPGAs also implement combinational and sequential logic in CLBs, yet the architecture of the CLB differs from the Virtex-II Pro FPGAs.

In Virtex-6 devices, each CLB consists of two slices, with no direct connection between them. The two slices are organized in two columns, with a slice on each column. Each slice is made up with four function generators, eight storage elements, wide-function multiplexers and carry logic.

Each of the four function generators is implemented as a LUT. Each LUT can be used to realize one 6-input Boolean function with six independent inputs or two 5-input Boolean functions provided that at least one of the inputs is common. Thus, the function generators

have either one or two outputs, depending on the function. These outputs can drive the output of the slice, be used for fast lookahead carry logic, feed the D input of the storage elements or go to the multiplexers. Each slice has the ability to combine multiple function generators to implement Boolean functions with seven or eight independent inputs. For functions with more than eight inputs, multiple slices are necessary.

There are eight storage elements in a slice. Four out of the eight storage elements can be configured as edge-sensitive D-type flip-flops or level-sensitive latches, as in the Virtex-II Pro FPGAs. The other four storage elements can only be configured as edge-sensitive D-type flip-flops and they cannot be used if the former four are used as latches. The input to the storage elements can come directly from the input of the slice or from the output of the function generators. Similar to Virtex-II Pro FPGAs, the control inputs of the storage elements are clock, clock enable, and set and reset.

The function generators and multiplexers in a Virtex-6 CLB can be configured to be multiplexers using the following amount of resources:

- 4:1 multiplexer in one LUT
- 8:1 multiplexer in two LUTs
- 16:1 multiplexer in four LUTs

Similar to Virtex-II Pro devices, dedicated carry logic is available in the slices to provide fast lookahead carry logic to perform arithmetic addition and subtraction more efficiently.

Virtex-6 BRAMs differ from Virtex-II Pro BRAMs in that Virtex-6 BRAM stores up to 36 Kb of data. The Virtex-6 BRAM can be used as two independent 18 Kb BRAMs or one 36 Kb BRAM. Each 36 Kb BRAM can be configured to the following dimensions:

- 32 Kb configurations:
  - 64K $\times$ 1 bit (by cascading two 36 Kb BRAMs)
  - 32K $\times$ 1 bit
  - 16K $\times$ 2 bits
  - 8K $\times$ 4 bits
- 36 Kb configurations:

- 4K × 9 bits
- 2K × 18 bits
- 1K × 36 bits
- 512 × 72 bits

and each 18 Kb BRAM can be configured to the same dimensions as in Virtex-II Pro. Similar to the Virtex-II Pro BRAMs, they can be implemented as single- or dual-port RAMs or ROMs. The memory is synchronous and in dual-port configuration, the ports have independent read and write data buses and clocks that share a common memory data.

The total available logic and memory resources of the XC6VLX240T FPGA are as follows:

Number of Slices: 37680
Number of 6-Input LUTs: 150720
Number of Slice Flip-Flops: 301440
Total Number of 36 Kb BRAMs: 416 (or 832 18 Kb BRAMs)

2.2 Review of Linear Block Codes

In this section, linear block codes are reviewed. The encoding and hard-decision decoding of linear block codes using generator and parity check matrices are presented. Decoding with soft-decision decoding metrics is also discussed and a general reliability-based soft-decision decoding scheme is presented. The information in this section is based on chapters 3 and 10 from Lin and Costello's book [40]. The block codes discussed in this section are binary block codes and the information source is also binary digits.

Block codes are a type of error control codes where a message block of length K information bits, denoted by $\mathbf{u}$, is encoded into codewords of N bits, denoted by $\mathbf{v}$. Linear block codes are block codes that the modulo-2 sum of two codewords is also a codeword. In fact, in linear block codes, it is possible to find K linearly independent codewords, $\mathbf{g}_0$, $\mathbf{g}_1$, $\cdots$, $\mathbf{g}_{K-1}$, such that every codeword $\mathbf{v}$ is a linear combination of these K codewords. These K linearly independent codewords can be organized in a $K \times N$ matrix to form the generator

matrix, as follows:

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_{K-1} \end{bmatrix} = \begin{bmatrix} g_{00} & g_{01} & g_{02} & \cdots & g_{0,N-1} \\ g_{10} & g_{11} & g_{12} & \cdots & g_{1,N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ g_{K-1,0} & g_{K-1,1} & g_{K-1,2} & \cdots & g_{K-1,N-1} \end{bmatrix} \tag{2.1}$$

where $\mathbf{g}_i = (g_{i0}, g_{i1}, \cdots, g_{i,N-1})$ for $0 \leq i \leq K$ is one of the codewords. In order to encode the message $\mathbf{u} = (u_0, u_1, \cdots, u_{K-1})$, the following operation is performed:

$$\begin{aligned} \mathbf{v} &= \mathbf{u} \cdot \mathbf{G} \\ &= (u_0, u_1, \cdots, u_{K-1}) \cdot \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_{K-1} \end{bmatrix} \\ &= u_0\mathbf{g}_0 + u_1\mathbf{g}_1 + \cdots + u_{K-1}\mathbf{g}_{K-1} \end{aligned} \tag{2.2}$$

As shown in equation (2.2), the K linear independent codewords that form the generator matrix can be used to form all the codewords in the code. Thus the generator matrix completely specifies the linear block code and only the K rows of the generator matrix need to be stored in the encoder during implementation instead of all 2^K N-bit codewords.

In order to further simplify encoding, linear systematic block codes can be used. The systematic structure means that the codeword can be subdivided into two parts, the message part and the redundant checking part. The message part has the K information bits of the original message before encoding and the redundant checking part has $N - K$ bits, called parity-check bits, that are linear sums of the information bits. The generator matrix of a linear systematic code has the form:

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \mathbf{g}_2 \\ \vdots \\ \mathbf{g}_{K-1} \end{bmatrix} = \begin{bmatrix} p_{00} & p_{01} & \cdots & p_{0,N-K-1} & 1 & 0 & 0 & \cdots & 0 \\ p_{10} & p_{11} & \cdots & p_{1,N-K-1} & 0 & 1 & 0 & \cdots & 0 \\ p_{20} & p_{21} & \cdots & p_{2,N-K-1} & 0 & 0 & 1 & \cdots & 0 \\ & & & & & & & & \\ p_{K-1,0} & p_{K-1,1} & \cdots & p_{K-1,N-K-1} & 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \tag{2.3}$$

where $p_{ij} = 0$ or 1. Let $\mathbf{P}$ be the left side of $\mathbf{G}$ in equation (2.3) and $\mathbf{I}_k$ be the $K \times K$ identity matrix on the right, then $\mathbf{G} = [\mathbf{P}\mathbf{I}_k]$. Encoding message $\mathbf{u} = (u_0, u_1, \cdots, u_{K-1})$ with the generator matrix in equation (2.3) yields:

$$v_{N-K+i} = u_i \tag{2.4}$$

for $0 \leq i < K$, which is the message part, and

$$v_j = u_0 p_{0j} + u_1 p_{1j} + \cdots + u_{K-1} p_{K-1,j} \tag{2.5}$$

for $0 \leq j < N - K$, which is the redundant checking part. The equations in (2.5) are called parity-check equations. When encoding linear systematic block codes, the parity-check bits are generated from the parity-check equations and the codeword is formed by concatenating the parity-check bits and the information bits.

For decoding linear block codes, there is another matrix that is associated with linear block codes, called the parity-check matrix, denoted matrix $\mathbf{H}$. $\mathbf{H}$ is a $(N-K) \times N$ matrix with $N-K$ linearly independent rows that satisfies the equation $\mathbf{G} \cdot \mathbf{H}^T = \mathbf{0}$, where $\mathbf{H}^T$ is the transpose of $\mathbf{H}$. Matrix $\mathbf{H}$ also satisfies the equation $\mathbf{v} \cdot \mathbf{H}^T = \mathbf{0}$. If $\mathbf{G}$ is in the systematic form as shown in equation (2.3), the $\mathbf{H}$ matrix has the following form:

$$\mathbf{H} = [\mathbf{I}_{N-K}\mathbf{P}^T] = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & p_{00} & p_{10} & \cdots & p_{K-1,0} \\ 0 & 1 & 0 & \cdots & 0 & p_{01} & p_{11} & \cdots & p_{K-1,1} \\ 0 & 0 & 1 & \cdots & 0 & p_{02} & p_{12} & \cdots & p_{K-1,2} \\ \vdots & & & & & & & & \\ 0 & 0 & 0 & \cdots & 1 & p_{0,N-K-1} & p_{1,N-K-1} & \cdots & p_{K-1,N-K-1} \end{bmatrix} \tag{2.6}$$

The parity-check equations can also be generated from the parity-check matrix, and linear block codes are also completely specified by its parity-check matrix $\mathbf{H}$.

Let $\mathbf{v} = (v_0, v_1, \cdots, v_{N-1})$ be the transmitted codeword and $\mathbf{r} = (r_0, r_1, \cdots, r_{N-1})$ be the received vector from the channel. Subsequently, the error vector, denoted $\mathbf{e}$, is given by:

$$\begin{aligned} \mathbf{e} &= \mathbf{r} + \mathbf{v} \\ &= (e_0, e_1, \cdots, e_{N-1}) \end{aligned} \tag{2.7}$$

where $e_j = 1$ if and only if $r_j \neq v_j$ and $e_j = 0$ if and only if $r_j = v_j$. By rearranging the terms, the following equations are produced:

$$\mathbf{v} = \mathbf{r} + \mathbf{e} \tag{2.8}$$

$$\mathbf{r} = \mathbf{v} + \mathbf{e} \tag{2.9}$$

Since the decoder receives $\mathbf{r}$ from the channel, the goal of the decoder is to generate $\mathbf{e}$ in order to recover the original codeword $\mathbf{v}$. Subsequently, for linear systematic block codes, the original message can be obtained from the message part of $\mathbf{v}$.

When decoding, the decoder receives $\mathbf{r}$ and produces the syndrome of $\mathbf{r}$, denoted by $\mathbf{s}$, by performing the following calculation:

$$\mathbf{s} = \mathbf{r} \cdot \mathbf{H}^T \tag{2.10}$$

The syndrome vector has length $N - K$ and since $\mathbf{v} \cdot \mathbf{H}^T = \mathbf{0}$, $\mathbf{s} = \mathbf{0}$ if and only if $\mathbf{r}$ is a codeword, otherwise $\mathbf{s} \neq \mathbf{0}$. However, if $\mathbf{e}$ is itself a codeword, then $\mathbf{r} = \mathbf{v} + \mathbf{e}$ is also a codeword, from the definition of linear block codes. Thus, the syndrome, $\mathbf{s} = \mathbf{0}$, but $\mathbf{r}$ is not the original codeword sent through the channel, in which case, it is said that a decoding error has occurred. Furthermore, according to equation (2.9), $\mathbf{s}$ can also be written as:

$$\mathbf{s} = \mathbf{e} \cdot \mathbf{H}^T \tag{2.11}$$

Solving the set of linear equations resulting from the expansion of equation (2.11) would yield the error vector $\mathbf{e}$. However, the result of the linear equations does not have a unique solution. Thus, in order to minimize decoding error, the most probable solution is selected. In a binary symmetric channel (BSC), where the output of the channel is a binary digit, the most probable solution is the one with the fewest number of non-zero elements. Furthermore, for large values of N and K, solving the set of $N - K$ equations with N unknowns becomes impractical, so more efficient methods are required.

One of these methods is called syndrome decoding. In this decoding method, the first step is to build a standard array. First, *1*) place the 2^K codewords on the zeroth row of the standard array with the all-zero codeword, $\mathbf{v}_0 = (0, 0, \cdots, 0)$ as the leftmost element. Then, *2*) select $\mathbf{e}_1$ to be a N-bit vector with the smallest number of non-zero elements and place

it under the all-zero vector, $\mathbf{v}_0$. *3*) Complete the first row by adding each of the remaining $2^K - 1$ codewords in the zeroth row to $\mathbf{e}_1$ and place $\mathbf{e}_1 + \mathbf{v}_i$ under $\mathbf{v}_i$. Afterwards, *4*) select $\mathbf{e}_2$ to be another N-bit vector with the smallest number of non-zero elements that does not already exist in the standard array. *5*) Complete the second row using the same method as the first row. *6*) Complete the remaining rows in a similar fashion until all N-bit vectors are exhausted. The completed standard array has the following format:

$$\begin{array}{ccccccc}
\mathbf{v}_0 = \mathbf{0} & \mathbf{v}_1 & \cdots & \mathbf{v}_i & \cdots & \mathbf{v}_{2^K-1} \\
\mathbf{e}_1 & \mathbf{e}_1 + \mathbf{v}_1 & \cdots & \mathbf{e}_1 + \mathbf{v}_i & \cdots & \mathbf{e}_1 + \mathbf{v}_{2^K-1} \\
\vdots & & \cdots & & & \vdots \\
\mathbf{e}_l & \mathbf{e}_l + \mathbf{v}_1 & \cdots & \mathbf{e}_l + \mathbf{v}_i & \cdots & \mathbf{e}_l + \mathbf{v}_{2^K-1} \\
\vdots & & \cdots & & & \vdots \\
\mathbf{e}_{2^{N-K}-1} & \mathbf{e}_{2^{N-K}-1} + \mathbf{v}_1 & \cdots & \mathbf{e}_{2^{N-K}-1} + \mathbf{v}_i & \cdots & \mathbf{e}_{2^{N-K}-1} + \mathbf{v}_{2^K-1}
\end{array} \tag{2.12}$$

Each row in the standard array is called a coset, and the leftmost element, $\mathbf{e}_l$ is called the coset leader. Decoding can be performed using the standard array as a dictionary because all 2^N possible N-bit vectors are present. In order to use the standard array for decoding, find the received vector, $\mathbf{r}$, among the vectors in the standard array, and the decoded codeword is vector, $\mathbf{v}_i$, on the same column as $\mathbf{r}$. However, the decoded codeword $\mathbf{v}_i$ may or may not be the original codeword sent through the channel because using this method to decode $\mathbf{r}$ to $\mathbf{v}_i$ means that $\mathbf{r} = \mathbf{e}_l + \mathbf{v}_i$, where $\mathbf{e}_l$ is interpreted as the error vector. Thus, $\mathbf{v}_i$ is the original codeword sent through the channel if and only if the error vector is indeed $\mathbf{e}_l$. Therefore, assuming on a BSC, in order to minimize decoding error, the coset leaders, $\mathbf{e}_l$, are chosen to have the smallest number of non-zero elements.

One drawback in decoding using the standard array directly is that all 2^N vectors must be stored in the decoder, so for large N it becomes impractical. This drawback can be overcome with some observations about the standard array.

Firstly, the syndrome of every vector in a coset is the same. Consider the vector $\mathbf{e}_l + \mathbf{v}_i$,

then its syndrome is as follows:

$$\begin{aligned} \mathbf{s} &= (\mathbf{e}_l + \mathbf{v}_i) \cdot \mathbf{H}^T \\ &= \mathbf{e}_l \cdot \mathbf{H}^T + \mathbf{v}_i \cdot \mathbf{H}^T \\ &= \mathbf{e}_l \cdot \mathbf{H}^T + \mathbf{0} \\ &= \mathbf{e}_l \cdot \mathbf{H}^T \end{aligned} \tag{2.13}$$

Since $\mathbf{s}$ is independent of $\mathbf{v}_i$, the syndrome of any element of the coset is equal to the syndrome of the coset leader. In addition, the set of all non-zero coset leaders can generate the set of all non-zero syndromes using equation (2.13) and there is a one-to-one correspondence between them. Thus, the decoder only needs to store or to wire a look-up table that converts the syndrome to the coset leader and uses it to correct the received vector from the channel, $\mathbf{r}$.

In summary, syndrome decoding is performed using the following three steps:

1. Compute the syndrome of $\mathbf{r}$, $\mathbf{s} = \mathbf{r} \cdot \mathbf{H}^T$.
2. Using the look-up table to convert the syndrome, $\mathbf{s}$, into the error vector, $\mathbf{e}_l$.
3. Decode the received vector, $\mathbf{r}$, into the codeword, $\mathbf{v}^* = \mathbf{r} + \mathbf{e}_l$.

An example of the encoding and decoding using the syndrome decoding of a $N = 7$ and $K = 4$ linear systematic block code is attached in Appendix A.

As mentioned earlier in this section, decoding a received vector into a codeword does not guarantee that the decoded codeword is the original codeword sent through the channel. An important parameter that determines the random-error-detecting and random-error-correcting capabilities of a linear block code is the minimum distance, denoted d_{min}. The minimum distance can be defined using one of two parameters, the Hamming weight or the Hamming distance. The Hamming weight of a vector $\mathbf{v} = (v_0, v_1, \cdots, v_{N-1})$, denoted by $w(\mathbf{v})$, is defined as the number of non-zero elements in $\mathbf{v}$. The Hamming distance between two vectors, $\mathbf{v} = (v_0, v_1, \cdots, v_{N-1})$ and $\mathbf{w} = (w_0, w_1, \cdots, w_{N-1})$, denoted by $d(\mathbf{v}, \mathbf{w})$, is the number of places where $\mathbf{v}$ and $\mathbf{w}$ differ by. Subsequently, the minimum distance can be defined as the minimum Hamming distance among all the codewords in the linear block code. The minimum distance can also be defined as the minimum Hamming weight of all

the codewords in the linear block code. Based on the minimum distance, d_{min}, the random-error-detecting capability of a block code is $d_{min} - 1$, which means any error vector with $d_{min} - 1$ or less non-zero elements is guaranteed to be detected by the decoder. Additionally, the random-error-correcting capability of a block code is given by:

$$t = \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor \tag{2.14}$$

which means that any error vector with t or less non-zero elements is guaranteed to be corrected by the decoder.

The discussion presented so far only applies to hard-decision decoding, where the received values from the channel are only treated as binary digits, 0 or 1. By doing so, much of the information from the channel is lost, which degrades performance. If the received values are interpreted as more than two levels, then the decoding is called soft-decision decoding. In general, soft-decision decoding has better performance than hard-decision decoding from the usage of the channel information. However, the drawback is the increased complexity in the implementation of the decoder to handle the multi-level values.

In soft-decision decoding, the minimum distance, Hamming weight and Hamming distance metrics are not applicable, so other metrics must be used. The most commonly used metrics are likelihood functions, Euclidean distance, correlation and correlation discrepancy.

Assume that the codeword $\mathbf{v} = (v_0, v_1, \cdots, v_{N-1})$ is transmitted over an additive white Gaussian noise (AWGN) channel with two-sided power spectral density (PSD) $N_0/2$ using binary phase-shift keying (BPSK) modulation. The codeword is mapped into a bipolar signal sequence $\mathbf{c} = (c_0, c_1, \cdots, c_{N-1})$, as follows:

$$c_l = 2v_l - 1 = \begin{cases} -1 & \text{for } v_l = 0 \\ +1 & \text{for } v_l = 1 \end{cases} \tag{2.15}$$

where $l = 0, 1, \cdots, N$. In addition, assume at the output of the channel, the soft-decision vector, $\mathbf{r} = (r_0, r_1, \cdots, r_{N-1})$, is received. The log-likelihood function of $\mathbf{r}$ given a codeword $\mathbf{v}$ is as follows:

$$\log P(\mathbf{r}|\mathbf{v}) = \sum_{i=0}^{N-1} \log P(r_i|v_i) \tag{2.16}$$

Using the log-likelihood function as decoding metric to perform maximum likelihood decoding (MLD), the received vector $\mathbf{r}$ is decoded into codeword $\mathbf{v}$ for which the log-likelihood function in (2.16) is maximized.

The squared Euclidean distance between $\mathbf{r}$ and $\mathbf{c}$, denoted $d_E^2(\mathbf{r}, \mathbf{c})$ is defined as follows:

$$d_E^2(\mathbf{r}, \mathbf{c}) \triangleq \sum_{i=0}^{N-1} (r_i - c_i)^2 \tag{2.17}$$

Soft-decision MLD is carried out by decoding the received sequence $\mathbf{r}$ into codeword $\mathbf{v}$ for which the squared Euclidean distance, $d_E^2(\mathbf{r}, \mathbf{c})$, is minimized.

The correlation between the received sequence $\mathbf{r}$ and the transmitted code sequence $\mathbf{c}$ is defined as follows:

$$m(\mathbf{r}, \mathbf{c}) \triangleq \sum_{i=0}^{N-1} r_i \cdot c_i \tag{2.18}$$

Soft-decision MLD is achieved by decoding the received sequence $\mathbf{r}$ into the codeword $\mathbf{v}$ for which the correlation, $m(\mathbf{r}, \mathbf{c})$, is maximized.

Finally, the correlation discrepancy between $\mathbf{r}$ and $\mathbf{c}$ is defined as:

$$\lambda(\mathbf{r}, \mathbf{c}) \triangleq \sum_{i: r_i \cdot c_i < 0} |r_i| \tag{2.19}$$

Soft-decision MLD is also carried out by decoding $\mathbf{r}$ into $\mathbf{v}$ for which the correlation discrepancy, $\lambda(\mathbf{r}, \mathbf{c})$, is minimized.

Using these metrics, soft-decision MLD can be performed by taking the received signal sequence, $\mathbf{r}$, and computing one of the metrics for all 2^K codewords and selecting the codeword with the maximum, or minimum depending on the metric, as the decoded codeword. However, for large K values, this method becomes impractical. To overcome this challenge, several non-optimum or sub-optimum soft-decoding algorithms have been developed. These algorithms can be divided into two categories: structure-based and reliability-based. The following discussion is on a general reliability-based soft-decision decoding algorithm scheme because the decoding algorithm used for the decoder in this thesis, as described in Section 2.3, is a reliability-based soft-decision decoding algorithm. For more information on other decoding schemes, refer to Lin and Costello's book [40].

In reliability-based decoding, the each symbol, r_i, in the received signal sequence, $\mathbf{r} = (r_0, r_1, \cdots, r_{N-1})$, is separated into two parts. The sign part, which is used for hard-decision

decoded bit:

$$z_i = \begin{cases} 0 & \text{for } r_i < 0 \\ 1 & \text{for } r_i \geq 0 \end{cases} \tag{2.20}$$

and the magnitude part, $|r_i|$, which is used as a reliability measure of z_i because the magnitude of the log-likelihood ratio (LLR) given by:

$$\left| \log \left(\frac{P(r_i|v_i = 1)}{P(r_i|v_i = 0)} \right) \right| \tag{2.21}$$

is proportional to $|r_i|$. Thus, the larger the $|r_i|$, the more reliable the hard-decision decoded bit z_i is. Based on the reliability measure, the elements in the received signal sequence, $\mathbf{r}$, can be reordered in decreasing order of reliability. As a result, the left side of the reordered sequence has elements that are more reliable, so they are called the most reliable positions (MRPs). In contrast, the right side of the reordered sequence contains the less reliable elements, so they are called the least reliable positions (LRPs). Consequently, errors are more likely to occur in the LRPs and less likely to occur in the MRPs. Based on these positions there exists two sub-categories of reliability-based decoding algorithms: LRP-reprocessing algorithms and MRIP-reprocessing algorithms.

LRP-reprocessing algorithms take advantage of the property that most errors exist within the LRPs of $\mathbf{r}$. These algorithms generally follow these steps:

1. Construct a set of error patterns confined only in the LRPs of $\mathbf{r}$.
2. Add each error pattern, $\mathbf{e}$, in the set to the hard-decision decoded vector, $\mathbf{z}$.
3. The resultant vector, $\mathbf{z} + \mathbf{e}$, is decoded using a hard-decision decoding algorithm to generate a list of candidate codewords.
4. Apply the soft-decision decoding metrics, as presented above, to each candidate codeword and select the codeword from the list of candidates which maximizes or minimizes the metric, depending on the metric used, to be the decoded codeword.

MRIP-reprocessing algorithms are based on the MRPs of $\mathbf{r}$. Since there are K independent positions on $\mathbf{z}$ that uniquely determine a codeword in a linear block code, these algorithms first determine a set of K most reliable independent positions (MRIPs) in $\mathbf{r}$. Let $\mathbf{z}_k$ denote a vector that consists of these K MRIP elements of $\mathbf{z}$. The following steps are the general procedure of MRIP-reprocessing algorithms:

1. Construct a set of low-weight K-length error patterns based on the K MRIPs of $\mathbf{r}$.
2. Add each error pattern, $\mathbf{e}$, in the set to $\mathbf{z}_k$.
3. Encode each resultant vector, $\mathbf{z}_k + \mathbf{e}$, into a codeword to form a list of candidate codewords.
4. Apply the soft-decision decoding metrics, as presented above, to each candidate codeword and select the codeword from the list of candidates which maximizes or minimizes the metric, depending on the metric used, to be the decoded codeword.

LDPC codes are a subcategory of linear block codes. Thus, they are characterized by the parity check matrix, $\mathbf{H}$, except the structure of $\mathbf{H}$ has the following properties, according to Lin and Costello's book [40]: *a*) no two rows or columns have more than one non-zero element in common; *b*) the row and column weights of $\mathbf{H}$ are small compared to the length of the code. Row and column weights refer to the number of non-zero elements in a row and column of the $\mathbf{H}$ matrix, respectively. If the row and column weights are constant, then the $\mathbf{H}$ matrix describes a regular LDPC code; otherwise, it describes an irregular LDPC code, which is the case in the DVB-S2 standard. All of the properties discussed in this section are applicable to LDPC codes. However, the encoding and decoding techniques discussed in the Section 2.3 differ from the ones presented in this section since the structure of the LDPC codes in the DVB-S2 standard allows for more efficient encoder and decoder implementations.

2.3 LDPC Codes in DVB-S2 Standard

One of the improvements of the DVB-S2 standard, which was ratified in 2005, from the original DVB-S standard is the usage of LDPC codes concatenated with BCH codes for FEC encoding and decoding, replacing convolutional and Reed-Solomon codes. This thesis focuses solely on the LDPC codes in the DVB-S2 standard. The discussion of the BCH codes in the standard is beyond the scope of this thesis. In this section, an overview of the LDPC codes in the DVB-S2 standard is presented.

The LDPC codes in the DVB-S2 standard have two block lengths. Normal frames have block length $N = 64800$ and short frames have $N = 16200$. Eleven code rates are specified

in normal frames[1] and ten in short frames[2].

According to the standard, even though the parity check matrices, $\mathbf{H}$, chosen by the standard are sparse, their respective generator matrices are not. Thus, the DVB-S2 standard adopts a special structure of the $\mathbf{H}$ matrix in order to reduce the memory requirement and the complexity of the encoder. It is called Irregular Repeat-Accumulate (IRA) [41]. The $\mathbf{H}$ matrix consists of two matrices $\mathbf{A}$ and $\mathbf{B}$, as follows:

$$\mathbf{H}_{(N-K)\times N} = [\mathbf{A}_{(N-K)\times K} | \mathbf{B}_{(N-K)\times(N-K)}] \tag{2.22}$$

where $\mathbf{B}$ is a staircase lower triangular matrix as shown in equation (2.23). The matrix $\mathbf{A}$ is a sparse matrix, where the locations of the non-zero elements are specified in Annex B and C of the standard [6] and reproduced in Appendix B. Furthermore, the standard also introduces a periodicity of $M = 360$ to the submatrix $\mathbf{A}$ in order to further reduce storage requirements.

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 1 & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & 1 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & 1 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 1 & 0 & 0 \\ 0 & \cdots & \cdots & 0 & 1 & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & 0 & 1 & 1 \end{bmatrix} \tag{2.23}$$

The periodicity condition divides the $\mathbf{A}$ matrix into groups of $M = 360$ columns. For each group, the locations of the non-zero elements of the first column are given in Appendix B. Let the set of non-zero locations on first, or leftmost, column of a group be $c_0, c_1, \cdots, c_{d_b-1}$, where d_b is the number of non-zero elements in that first column. For each of the $M-1 = 359$ other columns, the locations of the non-zero elements of the i^{th} column of the group are given by $(c_0 + (i-1)p) \bmod (N-K)$, $(c_1 + (i-1)p) \bmod (N-K)$, $(c_2 + (i-1)p) \bmod (N-K)$, $\cdots$, $(c_l + (i-1)p) \bmod (N-K)$. $N-K$ is the number of parity-check bits and $p = \frac{N-K}{M}$ is a

[1] 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 8/9, and 9/10
[2] 1/5, 1/3, 2/5, 4/9, 3/5, 2/3, 11/15, 7/9, 37/49, and 8/9

Table 2.1: The values of p values in DVB-S2 LDPC codes

$N = 64800$		$N = 16200$	
Code Rate	p	Code Rate	p
1/4	135	1/5	36
1/3	120	1/3	30
2/5	108	2/5	27
1/2	90	4/9	25
3/5	72	3/5	18
2/3	60	2/3	15
3/4	45	11/15	12
4/5	36	7/9	10
5/6	30	37/49	8
8/9	20	8/9	5
9/10	18		

code dependent constant as shown in Table 2.1. The values in Table 2.1 are obtained from the user guidelines of the standard [42].

Since the LDPC codes in the DVB-S2 standard are systematic, the encoding of message bits simply involves finding the parity bits through the parity-check equations. Using the structure of the codes as mentioned above, the $\mathbf{A}$ submatrix with dimensions $(N-K)\times K$ can be generated. Let $a_{i,j}$ denote the elements in the $\mathbf{A}$ submatrix, where $i = 0, 1, \cdots, N-K-1$ and $j = 0, 1, \cdots, K-1$. In order to encode the message, $\mathbf{u} = u_0, u_1, \cdots, u_{K-1}$, the parity bits are computed using the following parity-check equations as shown in Gomes et al. [35]:

$$\begin{aligned}
p_0 &= a_{0,0}u_0 \oplus a_{0,1}u_1 \oplus \cdots \oplus a_{0,K-1}u_{K-1} \\
p_1 &= a_{1,0}u_0 \oplus a_{1,1}u_1 \oplus \cdots \oplus a_{1,K-1}u_{K-1} \oplus p_0 \\
p_2 &= a_{2,0}u_0 \oplus a_{2,1}u_1 \oplus \cdots \oplus a_{2,K-1}u_{K-1} \oplus p_1 \\
&\vdots \\
p_{N-K-1} &= a_{N-K-1,0}u_0 \oplus a_{N-K-1,1}u_1 \oplus a_{N-K-1,K-1}u_{K-1} \oplus p_{N-K-2}
\end{aligned} \tag{2.24}$$

The encoded codeword is the concatenation of the message bits and the parity bits. Thus,

the resultant N-bit codeword has the following form:

$$(u_0, u_1, \cdots, u_{K-1}, p_0, p_1, \cdots, p_{N-K-1}) \tag{2.25}$$

The decoding of LDPC codes in the DVB-S2 standard is a soft-decision decoding. The output of the decoder is the hard-decision information bit sequence. To simplify the calculations, the inputs of the system are log-likelihood ratio (LLR) values. Let the transmitted codeword be $\mathbf{v} = (v_0, v_1, \cdots, v_l, \cdots, v_{N-1})$ and the soft-decision received sequence be $\mathbf{y}$, then the LLR value, denoted λ_l, for each code bit is given by:

$$\lambda_l = \log\left(\frac{P(v_l = 0|\mathbf{y})}{P(v_l = 1|\mathbf{y})}\right) \tag{2.26}$$

The LLR value represents the probability that a given received signal is more likely to be a 1 or a 0. A larger positive value represents a higher probability of the received signal being a 0 and a larger negative value represents a higher probability of it being a 1. The output of the system is the decoded message in bits, along with an output to indicate whether the decoding was completed successfully or an error still exists in the decoded message.

The decoding process can be visualized using the parity-check matrix $\mathbf{H}$ or with the help of Tanner graphs. In 1981, Tanner [43] developed a method of representing LDPC codes in a graphical form, which enabled further research using an iterative method to decode LDPC codes. An example of a Tanner graph is shown in Figure 2.1, and its respective $\mathbf{H}$ matrix is shown in (2.27)[3]. In the Tanner graph, each bit node (BN) represents a column in the $\mathbf{H}$ matrix, each check node (CN) represents a row, and each edge represents a non-zero element in the $\mathbf{H}$ matrix. For example, consider the $\mathbf{H}$ matrix in (2.27), there is a non-zero element at row 1, column 4, so there is an edge connecting check node m_1 to bit node n_4 in Figure 2.1.

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{2.27}$$

[3]For simplicity, the $\mathbf{H}$ matrix in (2.27) does not have the structure of the $\mathbf{H}$ matrices in the DVB-S2 standard.

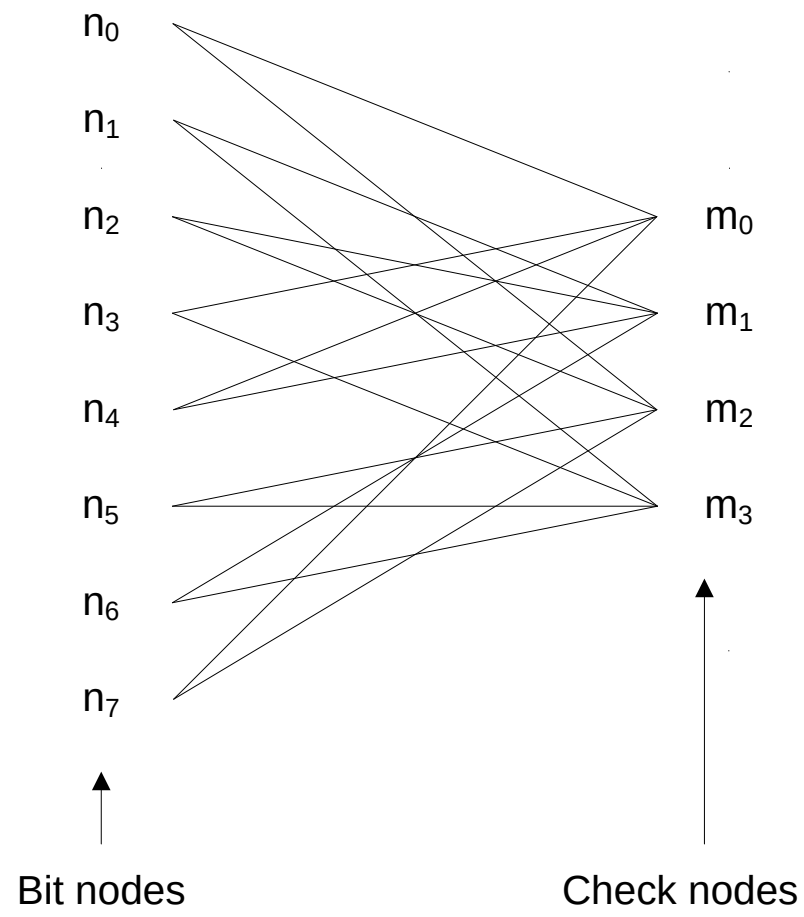

Figure 2.1: Example of a Tanner graph.

One of the commonly known algorithm to decode LDPC codes is the Sum-Product Algorithm (SPA) [40]. There are a few variations of this algorithm resulting from algebraic manipulation of mathematical expressions and approximations as presented by Papaharalabos et al. [23]. The SPA is a message-passing algorithm, where the messages that are real values are passed back-and-forth between the bit nodes and the check nodes. The message boxes of the nodes are the edges on the Tanner graph, which represent the non-zero elements in the parity-check matrix, **H**. The algorithm described below is similar to the one presented by Masera et al. [22], except the ψ^{-1} function in the Check Node Update step is replaced by the ψ function in the Bit Node Update step. This modification simplifies the control flow for the implementation of the equations in hardware, but does not affect the outcome of the equations because the ψ function is an involution, which means that the ψ function is its

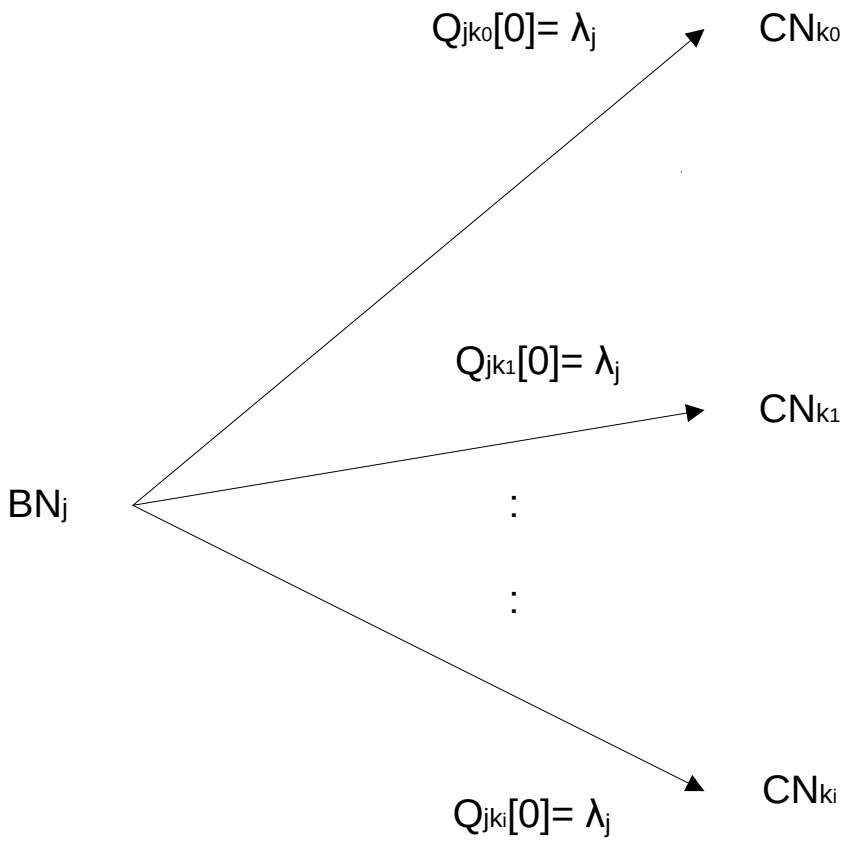

Figure 2.2: Initialization step of SPA.

own inverse. In addition, the steps are as laid out by Eroz et al. [44]. The algorithm consists of four steps:

1. (Initialization) Let the block length be N and the LLR of the received signals be λ_j where $j = 0, 1, \cdots, N-1$. Let $Q_{jk_i}[l]$ be the messages sent from BN_j to CN_{k_i} during the l^{th} iteration, where k_i is the index of a check node that has an edge connecting it to BN_j, $i = 0, 1, \cdots, d_b - 1$ and d_b is the bit node degree of BN_j. In the initialization step, perform the following operation:

$$Q_{jk_i}[0] = \lambda_j \tag{2.28}$$

The Tanner graph representation is shown in Figure 2.2. Using the **H** matrix, this step is

equivalent to assigning λ_j to every non-zero element on column j of the $\mathbf{H}$ matrix as follows:

$$\mathbf{H}[0]_b = \begin{bmatrix} h_{00} \cdot \lambda_0 & h_{01} \cdot \lambda_1 & \cdots & h_{0,N-1} \cdot \lambda_{N-1} \\ h_{10} \cdot \lambda_0 & h_{11} \cdot \lambda_1 & \cdots & h_{1,N-1} \cdot \lambda_{N-1} \\ \vdots & & & \vdots \\ h_{N-K-1,0} \cdot \lambda_0 & h_{N-K-1,1} \cdot \lambda_1 & \cdots & h_{N-K-1,N-1} \cdot \lambda_{N-1} \end{bmatrix} \tag{2.29}$$

where $h_{ij} = 0$ or 1 is the element on the i^{th} row and j^{th} column in $\mathbf{H}$ and $\mathbf{H}[0]_b$ is the initialized $\mathbf{H}$ matrix.

2. (Check Node Update) Let $R_{ik_j}[l]$ be the message sent from CN_i to BN_{k_j} during the l^{th} iteration, where k_j is the index of a bit node that has an edge connecting it to CN_i, $j = 0, 1, \cdots, d_c - 1$ and d_c is the check node degree of CN_i. Let $B[i]$ be the set of BN indices of all the messages incoming into CN_i from the BNs connected to it, i.e. the set of all k_j indices of CN_i. Perform the following calculation:

$$\begin{aligned} R_{ik_j}[l] &= [\sum_{m \in B[i]} \psi(Q_{mi}[l]) - \psi(Q_{k_j i}[l])] \cdot \\ &\quad [\prod_{m \in B[i]} \text{sgn}(Q_{mi}[l]) \times \text{sgn}(Q_{k_j i}[l])] \end{aligned} \tag{2.30}$$

where the ψ function is shown in (1.1) and $\text{sgn}(x)$ is the signum function defined as follows:

$$\text{sgn}(x) = \begin{cases} -1 & \text{for } x < 0 \\ 0 & \text{for } x = 0 \\ 1 & \text{for } x > 0 \end{cases} \tag{2.31}$$

In other words, check node update has 2 parts: magnitude and sign. In the magnitude part, for every CN, take all the incoming Q values, transform them using the ψ function and add up the results. Finally, for every outgoing message, subtract the sum by the $\psi(Q)$ value that corresponds to it. Similarly, the sign is computed the same way except, there is no ψ function and the product is taken instead of the sum. Refer to Figure 2.3 for a graphical representation of this step. Using the $\mathbf{H}$ matrix to visualize, this step takes in all non-zero elements on each row of $\mathbf{H}[l]_b$, processes them according to equation (2.30), and assigns $Rik_j[l]$ to the i^{th} row and k_j^{th} column of the $\mathbf{H}[l]_c$ matrix, where $\mathbf{H}[l]_c$ is the resultant $\mathbf{H}$ matrix after the Check Node Update step of the l^{th} iteration.

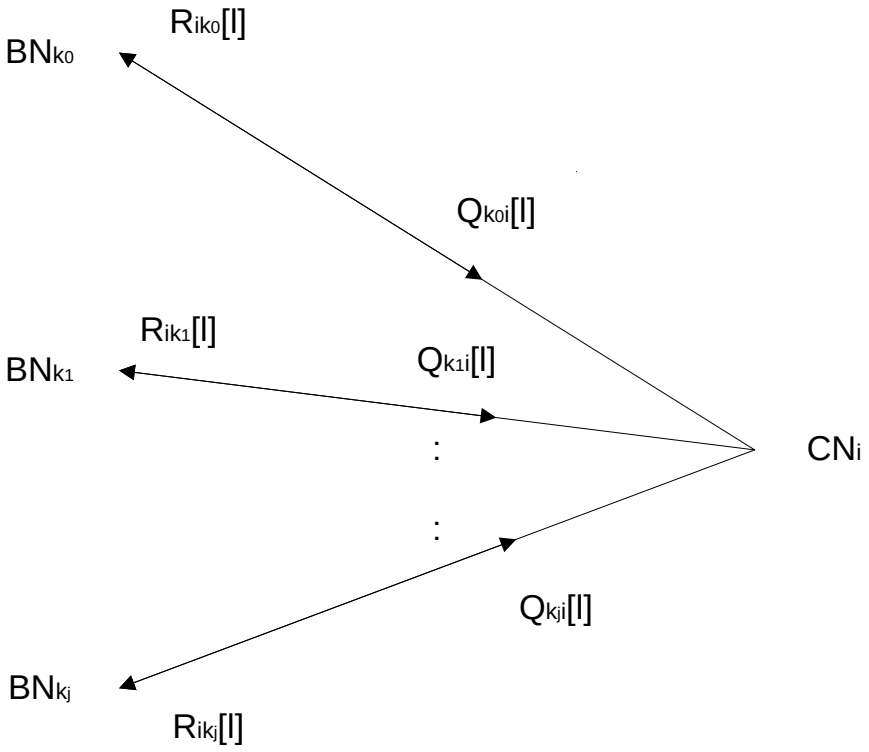

Figure 2.3: Check node update step of SPA.

3. (Bit Node Update) Let $C[j]$ be the set of CN indices of all messages incoming into BN_j from the CNs connected to it, i.e. the set of all k_i indices of BN_j. For bit node update, perform the following calculation:

$$\begin{aligned} Q_{jk_i}[l] &= \lambda_j + \sum_{m \in C[j]} \operatorname{sgn}(R_{mj}[l-1]) \cdot \psi(R_{mj}[l-1]) - \\ &\quad \operatorname{sgn}(R_{k_ij}[l-1]) \cdot \psi(R_{k_ij}[l-1]) \end{aligned} \tag{2.32}$$

This equation is similar to the check node update equation, except the sign calculations are included in the sum calculations, and the sum is added to the LLR value, λ_j. Figure 2.4 shows the graphical representation of this step. Using the $\mathbf{H}$ matrix, this step takes in all non-zero elements on each column of $\mathbf{H}[l]_c$, processes them using equation (2.32), and assigns $Qjk_i[l]$ to the k_i^{th} row and j^{th} column of the $\mathbf{H}[l]_b$ matrix., where $\mathbf{H}[l]_b$ is the resultant $\mathbf{H}$ matrix after the Bit Node Update step of the l^{th} iteration.

4. (Hard Decision Making) After bit node update, the soft-decision candidate codeword

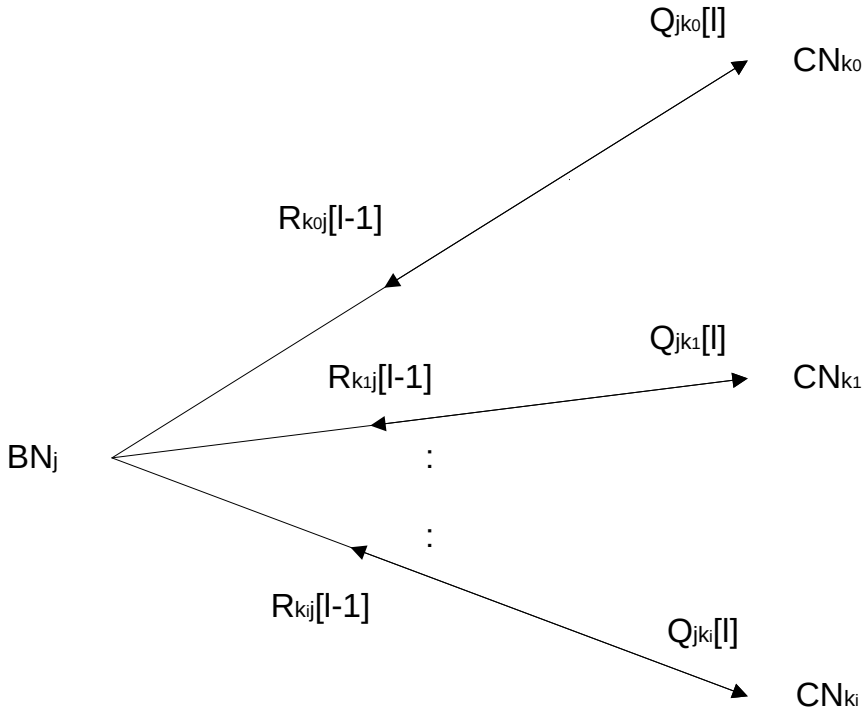

Figure 2.4: Bit node update step of SPA.

sequence, $\mathbf{S}[l] = (S_0, S_1, \cdots, S_j, \cdots, S_{N-1})$, is computed as follows:

$$S_j = \lambda_j + \sum_{m \in C[j]} \text{sgn}(R_{mj}[l-1]) \cdot \psi(R_{mj}[l-1]) \tag{2.33}$$

Equation (2.33) is equivalent to the first part of equation (2.32). Subsequently, the $\mathbf{S}$ sequence is decoded into hard-decision sequence, $\mathbf{z[l]} = (z_0, z_1, \cdots, z_j, \cdots, z_{N-1})$, with the following equation:

$$z_j = \begin{cases} 0 & \text{for } S_j \geq 0 \\ 1 & \text{for } S_j < 0 \end{cases} \tag{2.34}$$

The resultant sequence, $\mathbf{z}$, is a candidate codeword and is used to verify whether or not the parity-check equations in (2.24) are satisfied. If all parity-check equations are satisfied, decoding is complete, $\mathbf{z}$ is the decoded codeword, and its message part is the output. More specifically, the decoder outputs $(z_0, z_1, \cdots, z_{K-1})$. Otherwise, repeat steps 2, 3 and 4 until all parity-check equations are satisfied, or until a pre-determined number of iterations has elapsed without satisfying all parity-check equations, in which case a decoding error is declared.

Chapter 3

Architecture of DVB-S2 LDPC Decoder

3.1 Architecture of the Decoder

In this chapter, the details of the architecture of the implemented DVB-S2 LDPC decoder are presented. Figure 3.1 shows the inputs and outputs of the decoder. Table 3.1 describes each input and output of the decoder in more detail. In Table 3.1, the upstream side refers to the inputs and outputs of the decoder that interfaces an external module that inputs LLR values to the decoder. The downstream side refers to the inputs and outputs of the decoder that interfaces an external module that reads the output decoded message bits from the decoder.

The architecture of the LDPC decoder is based on the memory mapping scheme that is presented by Eroz et al. [8]. Figure 3.2 shows the block diagram of the LDPC decoder. The decoder consists of eight components: LLR Buffer, Functional Units (FUs), Shuffle Network, ROM, RAM, Parity Check Module (PCM), Decoded Message Buffer and Controller.

Referring back to the steps of SPA as laid out in Section 2.3, during the initialization step, the LLR values are input into the LLR Buffer as a serial stream of data. The LLR values are 6 bit values since Zhang et al. [20] demonstrates that 6-bit LLR values are sufficient to a small performance degradation. For every 360 LLR values collected, the values are copied into the RAM through the FUs, where they are compressed (described in more detail later in this chapter), and the Shuffle Network, where the values are shifted to the correct positions for the next step (also discussed in more detail later in this chapter). In the Check Node Update step, the values are read from the RAM and processed in the FUs according to equation (2.30). The results are written back to the RAM through the Shuffle Network,

Table 3.1: Description of the Inputs and Outputs of the Decoder

Input/ Output	Bit Width	Name	Description
Upstream Side:			
Input	6	llr	Serial 6-bit wide input LLR values
Input	1	nd	New data indicates that input LLR values are incoming
Input	1	fd_in	First data input marks the beginning of an input frame
Output	1	rfd	Ready for data indicates that the decoder is ready for more LLR values
Output	1	rffd	Ready for first data indicates that the decoder is ready for a new frame
Downstream Side:			
Output	1	decmsg	Serial hard decoded message output
Output	1	err	Indicates whether or not a decoding error has occurred
Output	1	rdy	Ready indicates the output data is ready to stream out
Output	1	fd_out	First data output marks the beginning of an output frame
Input	1	cts	Clear to send informs the decoder as to whether or not to output the decoded message
Others:			
Input	1	clk	Clock
Input	1	reset	Reset
Input	1	N	Selects between normal and short frames (0 - normal frame; 1 - short frame)
Input	4	rate	Selects the code rate ($0000_b \rightarrow 1010_b$ for normal frames; $0000_b \rightarrow 1001_b$ for short frames; in increasing order of code rate)
Input	8	max_iter	Sets the maximum number of iterations the decoder will perform
Input	1	fu_sel	Only used in the hybrid implementation, to select between the 360- or 180-Functional Unit mode

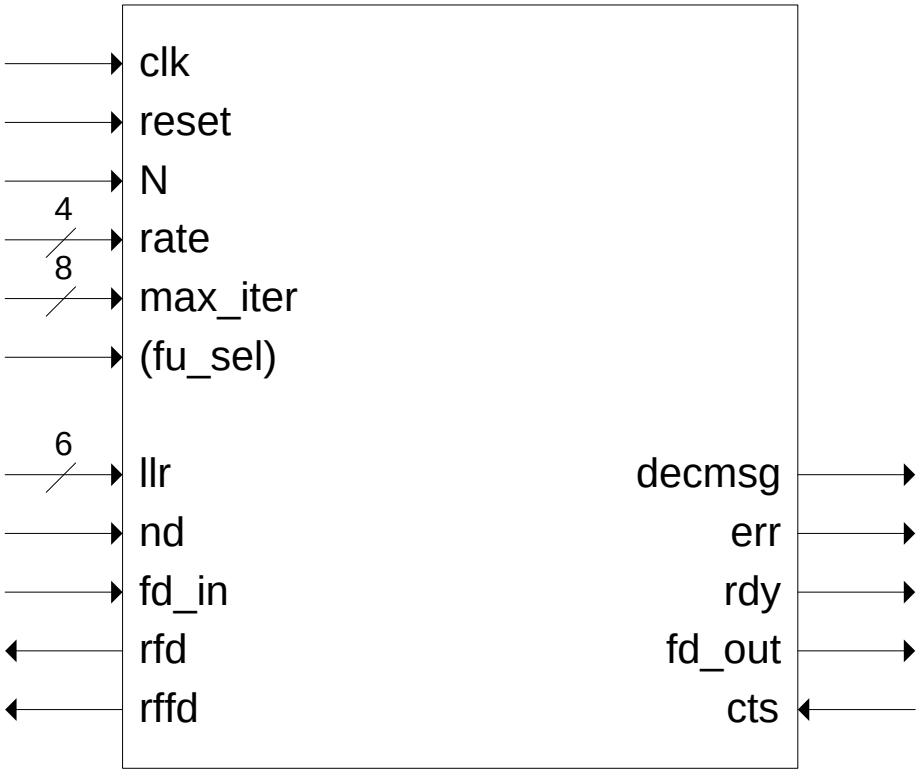

Figure 3.1: Inputs and Outputs of the LDPC decoder.

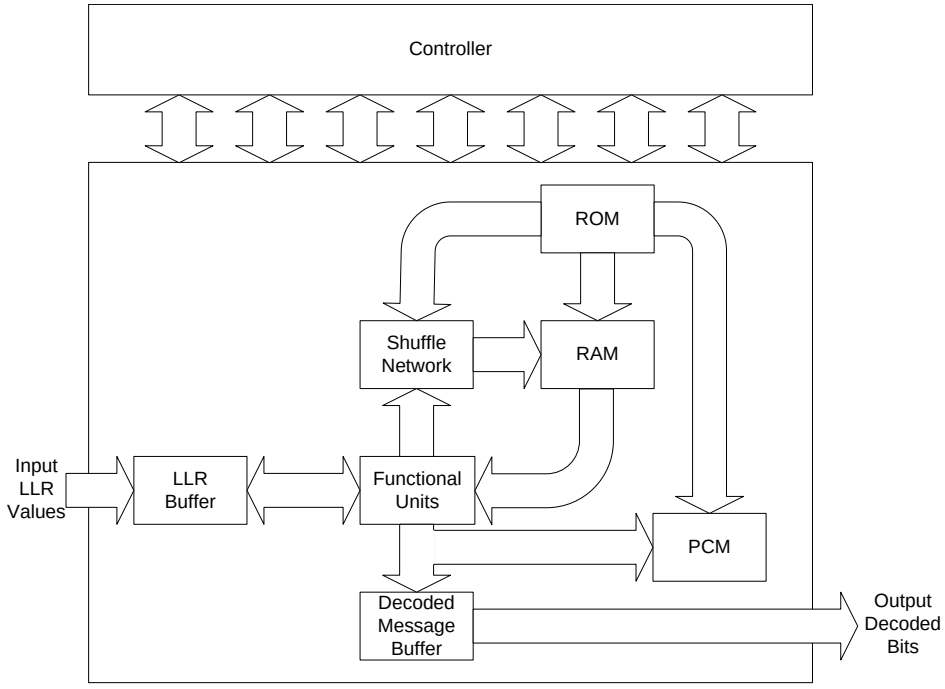

Figure 3.2: Top level block diagram of LDPC decoder.

where they are shifted into position for the next step. In the Bit Node Update step, the values are once again read from the RAM and processed in the FUs, but this time using equation (2.32). Since the LLR values are necessary in equation (2.32), the LLR Buffer is also read during this step. Once the resultant values are computed, the output is written back into the RAM through the Shuffle Network, where the values are shifted for the Check Node Update step if necessary. During the Bit Node Update step, the decoder is also performing the Hard Decision Making step because in order for the FUs to compute equation (2.32), they first compute the summation part, which is equation (2.33), as discussed in Section 2.3. Furthermore, from equation (2.34), the elements of the hard-decision candidate codeword sequence, $\mathbf{z}[l]$, are equivalent to the sign of the elements of the sequence $\mathbf{S}[l]$, so only the sign bits of $\mathbf{S}[l]$ are output from the FUs to the PCM. The FUs can generate 360- or 180-bit portions of the complete sign bit sequence, $\mathbf{z}[l]$, at a time and they are input into the PCM as they are generated. The portions that belong to the message part of the codeword are simultaneously stored in the Decoded Message Buffer as they are generated. The PCM verifies the parity-check equations, and its error output indicates whether or not the parity-check equations are satisfied. The error output of the PCM is input into the Controller to indicate whether or not to continue decoding. If the error output indicates that all the parity-check equations are satisfied, then the message part of the candidate codeword that has been stored in the Decoded Message Buffer is the decoded message and it is the output of the decoder. The decoded message is outputted from the Decoded Message Buffer and the decoder serially. Simultaneously, a new set of LLR values may be inputted into the decoder. If the error output of the PCM indicates that not all parity-check equations are satisfied, the decoder returns to the Check Node Update step and iterate until all parity-check equations are satisfied or a maximum number of iterations is reached.

The controller is a finite state machine that controls the above mentioned data flow in the decoder, so it has connections to all seven other components available in Figure 3.2, but these connections are not shown in the block diagram to avoid congestion in the figure. The state transition diagram of the controller is shown in Figure 3.3.

The decoder's control flow begins in the IDLE state. When both inputs nd and fd_in are active, the controller enters the INIT state. During the INIT state, the LLR values are being

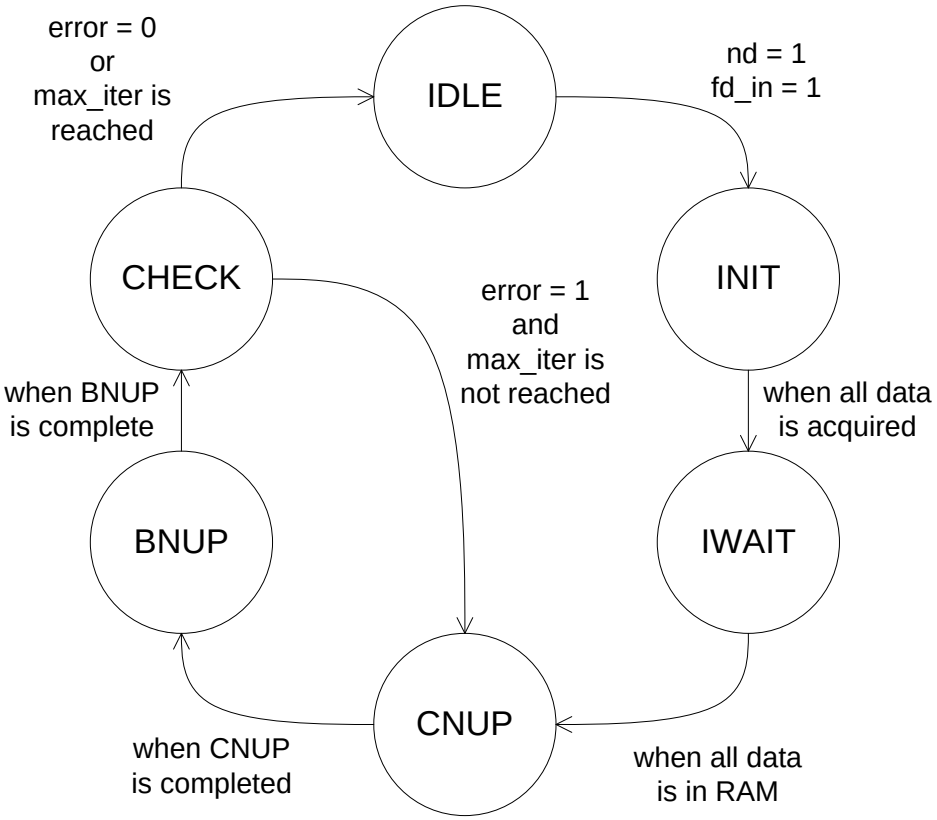

Figure 3.3: Controller FSM state diagram.

inputted into the decoder and the controller remains in the INIT state until all 64800 LLR values for normal frame, or 16200 LLR values for short frame, are inputted into the decoder, in which case the controller moves to the IWAIT state. The IWAIT state is a transitional state where the decoder has received all LLR values, but is not ready to perform calculations yet because some LLR values are still being written into the RAM through the FUs and the Shuffle Network. Once all the RAM values are ready, the controller goes into the CNUP state where the Check Node Update step is performed. Once the Check Node Update step is complete, the controller goes into the BNUP state where the Bit Node Update is performed. After all the Bit Node Update calculations are performed, the controller enters the CHECK state. During the CHECK state, the PCM verifies the parity-check equations. If all parity-check equations are satisfied, error = 0, then the controller enters the IDLE state and waits for the next frame of LLR values while outputting the decoded message. Otherwise, error = 1, and the controller returns to the CNUP state to repeat the CNUP, BNUP and CHECK states. If the maximum number of iterations is reached during the CHECK state, the

controller also moves to the IDLE state and outputs the decoded message with the output err set to 1.

There are three versions of the decoder, namely the 360-Functional-Unit (360-FU) version, the 180-Functional-Unit (180-FU) version and the hybrid 360/180-Functional-Unit (hybrid) version. Architecture of the 360-FU version is discussed first in the sections to follow. The design and architecture of each of the components of the decoder is discussed in detail. Subsequently, the modifications required to change from the 360-FU version to the 180-FU and the hybrid versions are presented.

The architecture of the decoder presented in the subsequent sections are designed for the Xilinx Virtex II-Pro XC2VP100 FPGA for comparison purpose with the decoder designed by Gomes et al. [18]. The discussion involving the use of the Xilinx Virtex-6 XC6VLX240T FPGA is presented in Chapter 4.

3.2 Architecture of the RAM and the ROM

From the control flow discussion above, for high throughput, the ideal decoder implementation would be to have one FU for every bit and check node, where each FU can perform either (2.30) or (2.32) calculations, and all FUs would run independently, which is the fully parallel decoder architecture. However, as mentioned above, the Check Node Update and Bit Node Update calculations are never performed at the same time. Thus, the FUs are designed to be able to handle both the (2.30) and (2.32) calculations provided that they are not performed at the same time, in which case, one FU is necessary per bit node because $N > N-K$. However, in the DVB-S2 standard, $N = 64800$ for normal frames and $N = 16200$ for short frames, so 64800 FUs are required, which is impractical for hardware implementation because there is a limited number of hardware resources on an FPGA as described in Section 2.1.

Eroz et al. [8] propose that taking advantage of the periodicity factor of $M = 360$ of the parity-check matrix $\mathbf{H}$, as discussed in Section 2.3, and appropriately organizing the RAM can result in a decoder architecture that can efficiently perform the decoding by only using 360 FUs. In the following subsections, the mentioned memory organization scheme is

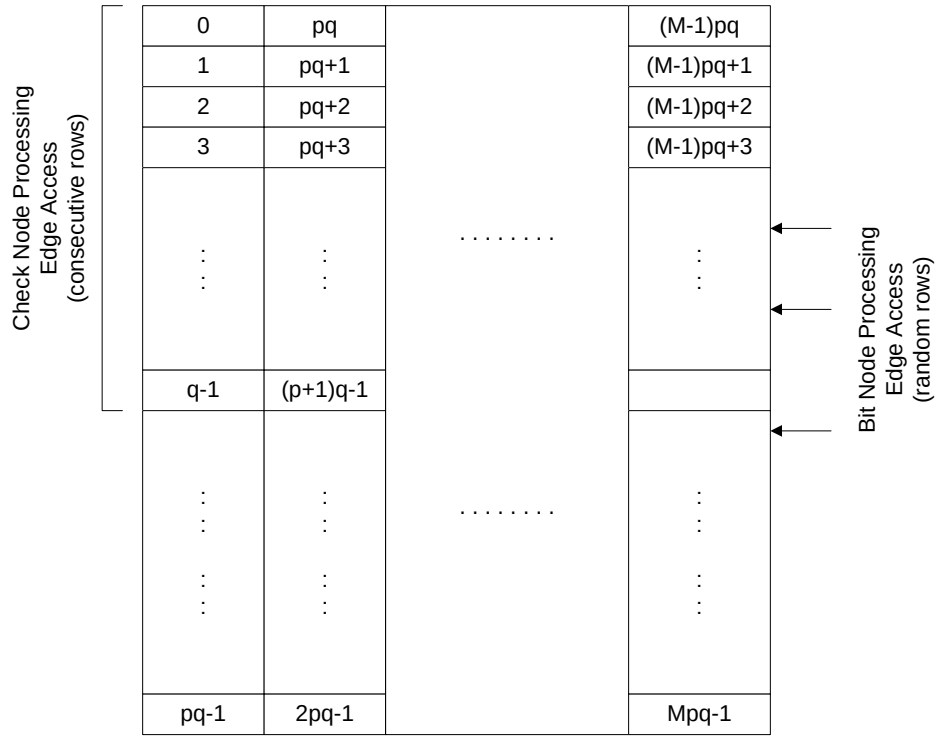

Figure 3.4: Edge placement and access of the Top RAM.

presented, followed by a discussion of the modules required for its implementation.

3.2.1 Memory Mapping Scheme

The memory mapping scheme of the LDPC decoder presented here is based on the scheme presented by Eroz et al. [8]. Each edge in the Tanner graph (or each non-zero element in the **H** matrix) is mapped to a location in the RAM, which acts as the message box for the R and Q values from (2.30) and (2.32), respectively, to be stored. The RAM is virtually divided into a top and a bottom RAM. The top RAM corresponds to the non-zero elements of the **A** submatrix and the bottom RAM corresponds to the non-zero elements of the **B** submatrix. During the Check Node Update and Bit Node Update steps, the FUs read the values from the RAM, process them, and write them back to the RAM. The locations from which the FUs read from the RAM depending on the **H** matrix.

Eroz et al. [8] suggest that if the RAM is organized as shown in Figure 3.4 and Figure 3.5 and each non-zero element of **H** is mapped correctly to each cell of the RAM, the RAM access during the Check Node Update step is a sequential access in the top RAM of q rows

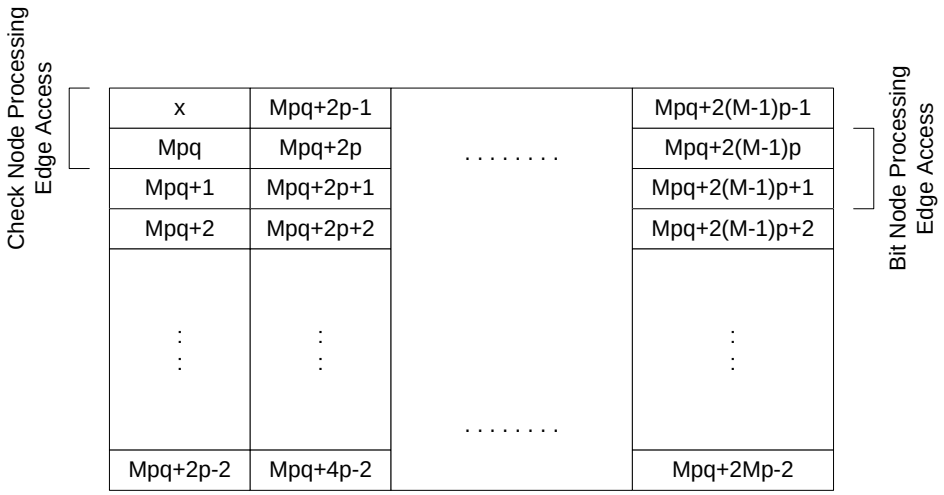

Figure 3.5: Edge placement and access of the Bottom RAM

followed by 2 rows of the bottom RAM. In Figure 3.4 and Figure 3.5, p and q are code-rate-specific values. p is shown in Table 2.1 and Table 3.2 and it represents the number of $M = 360$ check node groups and q is shown in Table 3.2 and it corresponds to the row weight of the **A** submatrix. Furthermore, by organizing the RAM as in Figure 3.4 and Figure 3.5, only $M = 360$ FUs need to be implemented and each FU only accesses and processes the values stored in one column of the top RAM and one column of the bottom RAM. During the Bit Node Update step, the rows that need to be processed are at random locations in the top RAM followed by a sequential access of rows in the bottom RAM. Thus, the address of the rows that are accessed during Bit Node Update need to be stored in the ROM for each code rate in the standard.

Relating back to the **H** matrix, during the Check Node Update step, each row of the top or bottom RAM corresponds to the set that consists of one non-zero element from each row of a check node group in **H**, where a check node group is the collection of the rows $i, i+p, i+2p, \cdots, i+(M-1)p$ of the **H** matrix, and $i = 0, 1, 2, \cdots, p-1$ is the check node group index. Thus, when sequentially accessing the top RAM rows, the decoder is accessing the message box values that corresponds to each non-zero element of the rows in a check node group in the **A** submatrix. When accessing the bottom RAM rows, the decoder is accessing the message box values that corresponds to each non-zero element of the rows in a check node group in the **B** submatrix. In other words, if cell 0 in Figure 3.4 corresponds to a non-

Table 3.2: RAM size of all the block length and code rates in DVB-S2

Block length	Code Rate	# of check node groups	check node degree	# of edges in top RAM	# of RAM rows
N		$p = \frac{N-K}{M}$	d_c	$q = d_c - 2$	$pq + 2p$
64800	1/4	135	4	2	540
	1/3	120	5	3	600
	2/5	108	6	4	648
	1/2	90	7	5	630
	3/5	72	11	9	792
	2/3	60	10	8	600
	3/4	45	14	12	630
	4/5	36	18	16	648
	5/6	30	22	20	660
	8/9	20	27	25	540
	9/10	18	30	28	540
16200	1/5	36	3.75*	1.75	135
	1/3	30	5	3	150
	2/5	27	6	4	162
	4/9	25	5.4*	3.4	135
	3/5	18	11	9	198
	2/3	15	10	8	150
	11/15	12	11*	9	132
	7/9	10	12.5*	10.5	125
	37/49	8	17.13*	15.13	137
	8/9	5	27	25	135

*Special Case Code Rates

zero element on row 0 of the **A** submatrix, then cells $1, 2, \cdots, q-1$ correspond to the other non-zero elements on row 0 of the **A** submatrix. Furthermore, cells $pq, 2pq, \cdots, (M-1)pq$ correspond to a non-zero element on rows $p, 2p, \cdots, (M-1)p$, respectively.

During the Bit Node Update step, each row in the top RAM corresponds to the set that consists of one non-zero element from each column of a bit node group, where a bit node group is a collection of the columns $i, i+1, i+2, \cdots, i+(M-1)$ of the **A** submatrix,

and $i = 0, M, 2M, \cdots, K - M$ is the row index of the bit node group leader. Recall from Section 2.3 that if the locations of the non-zero elements of the leftmost column in a bit node group in the **A** submatrix, which is the bit node group leader, are $c_0, c_1, \cdots, c_{d_b-1}$, where d_b is the bit node degree, which is the number of edges that are connected to that bit node, then the non-zero element locations of the other columns of the same bit node group are given by the downward cyclic shift of $c_0, c_1, \cdots, c_{d_b-1}$ by p. By mapping the RAM accordingly, the cells of a top RAM row correspond to the respective non-zero elements on each column of a bit node group. In other words, if cell 0 in Figure 3.4 corresponds to the non-zero element on row (or location) c_0 and column 0 of the **A** submatrix, then cell pq, which is in the same top RAM row as cell 0, corresponds to the non-zero element on row $(c_0 + p) \bmod (N - K)$ and column 1 of the **A** submatrix, cell $2pq$ corresponds to the non-zero element on row $(c_0 + 2p) \bmod (N - K)$ and column 2 of the **A** submatrix, and so on. Furthermore, since cell 0, pq and $2pq$ are in row 0 of the top RAM, the value 0 must be stored in the ROM. Similarly, the row indices of the cell in the top RAM, which corresponds to rows $c_1, c_2, \cdots, c_{d_b-1}$ and column 0 of the **A** submatrix also need to be stored in the ROM because these row indices are code rate dependent.

In the **B** submatrix, the bit node groups are organized differently. The bit node groups are columns $i, i + p, i + 2p, \cdots, i + (M - 1)p$ of the **B** submatrix, where $i = 0, 1, 2, \cdots, p - 1$. However, since the **B** submatrix always has two non-zero elements in sequence, except for the rightmost column, the bottom RAM to **B** submatrix correspondence during Bit Node Update is less complex. If cell Mpq in Figure 3.5 corresponds to the top non-zero element of column 0 of **B**, then cell $Mpq + 1$ corresponds to the other non-zero element of column 0 and cells $Mpq + 2p, Mpq + 4p, \cdots, Mpq + 2(M - 1)p$ correspond the top non-zero element of columns $p, 2p, \cdots, (M - 1)p$, respectively.

As can be seen in Figure 3.4 and Figure 3.5, the size of the top RAM is $pq \times M$ and the size of the bottom RAM is $2p \times M$. In order for the decoder to support all 21 block length and code rate combinations, the size of the RAM must be the maximum value of $(pq + 2p) \times 360$ for each of the code rates. According to Table 3.2, the largest RAM is necessary when the block length is 64800 and the code rate is 3/5, where the RAM size is 792×360. Note that in Table 3.2 there are some code rates where q is not an integer. These

code rates will be discussed later.

In Section 3.3, the values stored in RAM will be shown to be 5 bits wide. Thus, the total size of the RAM is $792 \times 360 \times 5 = 1425600$ bits or 1.36 Mb. However, recall from Section 2.1 that the RAM in the Virtex-II Pro FPGAs are organized as 18 Kb BRAMs with various configurations. In order to implement the RAM for the decoder, the FPGA utilizes 100 18Kb BRAMs using the 1K $\times$ 18 bits configuration.

Eroz et al. [8] make a one-to-one mapping of every non-zero element in the **H** matrix to every location in the RAM. The paper presents an example on how after the **H** matrix is successfully mapped to the RAM, the RAM access is sequential during the Check Node Update step and indexed during the Bit Node Update step as described above, but it only discusses the algorithm to map the bottom RAM. When mapping the **B** submatrix to the bottom RAM, the top left corner cell of the bottom RAM is unused. Subsequently, the non-zero elements in the submatrix **B** are mapped as follows:

$$\mathbf{B} = \begin{bmatrix}
1 & & & & & & & & & & \\
\downarrow & & & & & & & & & & \\
1 & \rightarrow & 1 & & & & & & & & \\
 & & \downarrow & & & & \mathbf{0} & & & & \\
 & & 1 & \rightarrow & 1 & & & & & & \\
 & & & & \downarrow & & & & & & \\
 & & & & 1 & \rightarrow & & & & & \\
 & & & & & & \ddots & & & & \\
 & & & \mathbf{0} & & & & \rightarrow & 1 & & \\
 & & & & & & & & \downarrow & & \\
 & & & & & & & & 1 & \rightarrow & 1
\end{bmatrix} \tag{3.1}$$

However, Eroz et al. [8] do not present the algorithm to map the **A** submatrix to the top RAM. Thus, a novel algorithm is devised that can systematically map the **A** submatrix of any LDPC code with the same structure as the ones defined in the DVB-S2 standard to the RAM architecture as described above.

Consider the **H** matrix of the example in Eroz et al. [8] as equation (3.2).

$$\mathbf{H} = \underbrace{\left[\begin{array}{cccccccccccc|cccccc}
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}\right]}_{\mathbf{A} \qquad\qquad\qquad\qquad \mathbf{B}}
\begin{array}{l}
\Rightarrow e_0 \rightarrow e_5, e_{36} \\
\Rightarrow e_6 \rightarrow e_{11}, e_{37} \rightarrow e_{38} \\
\Rightarrow e_{12} \rightarrow e_{17}, e_{39} \rightarrow e_{40} \\
\Rightarrow e_{18} \rightarrow e_{23}, e_{41} \rightarrow e_{42} \\
\Rightarrow e_{24} \rightarrow e_{29}, e_{43} \rightarrow e_{44} \\
\Rightarrow e_{30} \rightarrow e_{35}, e_{45} \rightarrow e_{46}
\end{array} \tag{3.2}$$

The parameters for this code are: $N = 18$, $M = 3$, $p = 2$, and $q = 6$. The top and bottom RAM are shown in (3.3).

$$topRAM = \begin{bmatrix}
e_0 & e_{12} & e_{24} \\
e_1 & e_{13} & e_{25} \\
e_2 & e_{14} & e_{26} \\
e_3 & e_{15} & e_{27} \\
e_4 & e_{16} & e_{28} \\
e_5 & e_{17} & e_{29} \\
e_6 & e_{18} & e_{30} \\
e_7 & e_{19} & e_{31} \\
e_8 & e_{20} & e_{32} \\
e_9 & e_{21} & e_{33} \\
e_{10} & e_{22} & e_{34} \\
e_{11} & e_{23} & e_{35}
\end{bmatrix} \quad bottomRAM = \begin{bmatrix}
x & e_{39} & e_{43} \\
e_{36} & e_{40} & e_{44} \\
e_{37} & e_{41} & e_{45} \\
e_{38} & e_{42} & e_{46}
\end{bmatrix} \tag{3.3}$$

The cells in the top and bottom RAMs are represented by e_i, which correspond to edges in the Tanner graph or non-zero elements in the **H** matrix. As shown in (3.2), the six non-zero elements in row 0 of submatrix **A** maps to $e_0 \rightarrow e_5$, the six non-zero elements in row 1 maps to $e_6 \rightarrow e_{11}$, and so on. However, in order to perform the Bit Node Update step, one needs to know exactly which non-zero element corresponds to which edge. Thus, Algorithm 1 is devised to systematically map the non-zero elements to the edges.

Algorithm 1 Memory Mapping Algorithm

1. Set up top RAM with size $pq \times M$.

2. Label all edges sequentially as in (3.3).

3. Identify the set of edges that corresponds to each row of submatrix **A** as in (3.2).

for Every bit node group of M columns in submatrix **A**, starting from the left **do**

for Every non-zero element in the bit node group leader, starting from the top **do**

4a. Identify the row where the current non-zero element is located.

4b. Assign the lowest numbered edge available of that row to the current non-zero element.

4c. From the top RAM, find the edge that was just assigned.

4d. Identify the remaining edges that are in the same top RAM row.

4e. Assign those edges from left to right and cyclically wrapped to the corresponding non-zero elements in the remaining columns of the bit node group, which are all downward-cyclic-shifted versions of the column to its left by p.

end for

end for

In the above example, start with the non-zero element in row 0, column 0 and assign it to e_0. The remaining edges in the same row as e_0 in the top RAM are e_{12} and e_{24} which are assigned to the non-zero elements in row 2, column 1 and row 4, column 2, respectively. Next, row 2, column 0 is assigned e_{13} because e_{12} is already used, and e_{25} and e_1 is assigned to row 4, column 1 and row 0, column 2, respectively. The process continues until all edges are assigned. Upon the completion of applying Algorithm 1 and using the mapping of the bottom RAM provided in (3.1), the following is the mapped version of matrix **H** in (3.2):

$$\mathbf{H} = \begin{bmatrix} e_0 & 0 & e_1 & 0 & e_2 & 0 & 0 & 0 & e_3 & e_4 & e_5 & 0 & e_{36} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & e_6 & e_7 & e_8 & 0 & e_9 & e_{10} & 0 & 0 & 0 & e_{11} & e_{37} & e_{38} & 0 & 0 & 0 & 0 \\ e_{13} & e_{12} & 0 & 0 & 0 & e_{14} & e_{15} & 0 & 0 & 0 & e_{16} & e_{17} & 0 & e_{39} & e_{40} & 0 & 0 & 0 \\ e_{18} & 0 & 0 & 0 & e_{19} & e_{20} & 0 & e_{21} & e_{22} & e_{23} & 0 & 0 & 0 & 0 & e_{41} & e_{42} & 0 & 0 \\ 0 & e_{25} & e_{24} & e_{26} & 0 & 0 & 0 & e_{27} & 0 & e_{29} & 0 & e_{28} & 0 & 0 & 0 & e_{43} & e_{44} & 0 \\ 0 & e_{30} & 0 & e_{32} & 0 & e_{31} & e_{34} & 0 & e_{33} & 0 & e_{35} & 0 & 0 & 0 & 0 & 0 & e_{45} & e_{46} \end{bmatrix} \tag{3.4}$$

From (3.4), the $\frac{row}{shift}$ coefficients can be generated. These coefficients are stored in the ROM for the Bit Node Update step and for the example above they are as follows:

$$\begin{array}{ccccc} \frac{0}{0} & , & \frac{1}{1} & , & \frac{6}{1} \\ \frac{7}{0} & , & \frac{2}{2} & , & \frac{8}{2} \\ \frac{9}{0} & , & \frac{3}{1} & , & \frac{10}{2} \\ \frac{4}{0} & , & \frac{11}{1} & , & \frac{5}{2} \end{array} \tag{3.5}$$

The *row* coefficient indicates the locations of the top RAM rows for a given bit node group. From the example, for the leftmost bit node group, which consists of columns 0, 1 and 2 of the **H** matrix, the *row* coefficients are 0, 1 and 6, from the top row in (3.5). These coefficients are generated from the **H** matrix in (3.4). The bit node group leader, which is column 0 of the **H** matrix in (3.4), has three non-zero elements and they are labelled e_0, e_{13} and e_{18}. From (3.3), these three labels are found in rows 0, 1 and 6 of the top RAM, which are the *row* coefficients of the bit node group. Notice that the non-zero elements on column 1 and 2 of (3.4) are also found on rows 0, 1 and 6 of the top RAM in (3.3) and the respective elements resulted from the downward cyclic shift of the non-zero locations of the bit node group leader are on the same row in the top RAM, i.e. e_0, e_{12} and e_{24} are on the same row because the locations of e_{12} and e_{24} are downward cyclic shifts of the location of e_0 by $p = 2$. The *shift* coefficients are generated by finding column in the top RAM in which the non-zero elements of the bit node group leader are located. For the leftmost bit node group, edges e_0, e_{13} and e_{18} from the bit node group leader in matrix **H** in (3.4) are in columns 0, 1 and 1 of the top RAM in (3.3), respectively. Thus, the *shift* coefficients in the first row of (3.5) are 0, 1 and 1.

The set of coefficients presented in (3.5) are different from the ones generated by Eroz et al. [8] for the same **H** matrix, which means that the memory mapping algorithm applied is different. Nevertheless, Algorithm 1 still produces a RAM in which the same RAM access mechanism can be applied. Furthermore, in the given example, the **A** submatrix only has 36 non-zero elements, and the number of non-zero elements in the **A** submatrices defined in the DVB-S2 standard are in the order of 10^5 for normal frames and 10^4 for short frames. In order to generate the ROM coefficients for the LDPC codes in the DVB-S2 standard using Algorithm 1, the complete **A** submatrix needs to be generated first using the values

given in Appendix B, which are related to the non-zero element locations of the bit node group leaders. Subsequently, one can apply Algorithm 1, which searches through every non-zero element in the **A** submatrix and assign them to a cell in the top RAM. However, the completely labelled **A** submatrix is then reduced to the $\frac{row}{shift}$ coefficients to store in ROM, which only depends on the non-zero element locations of the bit node group leader. The process is cumbersome and impractical because of the large number of non-zero elements in **A** that are assigned a label only to be reduced back to only the non-zero elements in the bit node group leader. Therefore, a more efficient method has been devised to generate the ROM coefficients as described next.

3.2.2 Generation of ROM Coefficient

Upon examining Algorithm 1 more closely, one can see that only the locations of the non-zero elements of the bit node group leaders, given in Appendix B, are necessary to map the cells of the top RAM to the **A** submatrix. Furthermore, only the labels and positions of the cells in the top RAM of the non-zero elements in the bit node group leaders are responsible for the *row* and *shift* values given in (3.5). Thus, it would be logical to conclude that it is possible to convert the values given in Appendix B directly into the ROM coefficients. Algorithm 2 has been devised to perform this conversion more efficiently than the process described in Section 3.2.1.

Referring back to the $N = 18$ example, if the LDPC code was part of the standard, the values in Appendix B would be of the form:

$$\begin{matrix} 0\ 2\ 3 \\ 1\ 4\ 5 \\ 1\ 2\ 5 \\ 0\ 3\ 4 \end{matrix} \tag{3.6}$$

These values are obtained from the **H** matrix in (3.2). The bit node group leader of the leftmost bit node group has non-zero elements in rows 0, 2 and 3, and the remaining bit node group columns are the downward cyclic shift version of the bit node group leader by $p = 2$. Thus, the first set of coefficients is 0, 2 and 3 as found in (3.6).

Algorithm 2 ROM Coefficient Generation Algorithm for any code rate (except for the special code rates discussed in Section 3.2.4)

1. Read the values from Appendix B for a particular code rate.

2. Initialize a vector $LUT = [0, q, 2q, 3q, \cdots, (p-1)q]$.

for Every value (g) read from Appendix B **do**

3a. $index = g \bmod p$

3b. Collect $row = LUT(index)$

3c. $LUT(index) = LUT(index) + 1$

3d. Collect $shift = g \div p$

end for

return Every *row* and *shift* value collected

By applying Algorithm 2, the $\frac{row}{shift}$ values obtained in (3.5) can be generated from (3.6) and the result is identical to the ones generated using Algorithm 1 and reading the ROM coefficients from the labelled **H** matrix as described in Section 3.2.1.

Even though the generation of the ROM coefficients are performed off-line, Algorithm 2 allows the generation of the ROM coefficients more efficiently without the need to expand the values given in the standard into a complete **H** matrix. Furthermore, Algorithm 2 can be used for any LDPC code that has the same structure as the codes defined in the DVB-S2 standard.

3.2.3 Function and Architecture of the Shuffle Network

The *shift* coefficients discussed in previous sections are also stored in the ROM. These coefficients are used by the Shuffle Network to perform cyclic shifts on the outputs of the FUs before they are stored in the RAM. The outputs of the FUs need to be shifted because from the memory mapping scheme discussed in Section 3.2.1, each FU accesses and processes one column in the top RAM and one column in the bottom RAM. For example, consider the mapped **H** matrix in (3.4) and its top and bottom RAM in (3.3). Since $M = 3$ in the example, 3 FUs are implemented, denoted FU_0, FU_1 and FU_2 that are responsible for columns 0, 1 and 2 respectively. Consider the top RAM access for FU_0. During the Check Node Update

step, FU_0 accesses top RAM cells e_0, e_1, e_2, e_3, e_4 and e_5 because they are all in row 0 of the **A** submatrix in (3.4). Since these cells are all in column 0, FU_0 simply needs to access contents of the cells sequentially from the RAM. Similarly, the RAM access of FU_1 and FU_2 are also sequential and the RAM access for all other check node groups are also performed in a similar fashion.

Assume that the new contents at the output of FU_0 are written back to the same top RAM locations as they are read from at the end of the Check Node Update. During the Bit Node Update step, FU_0 is responsible for processing the bit node group leader of the bit node groups. Thus, FU_0 needs to process the contents in cells e_0, e_{13} and e_{18} for the leftmost bit node group, according to (3.4). In the top RAM, the contents of these cells are in row 0, columns 0; row 1, column 1; and row 6, column 1 of the top RAM, respectively, which are the *row* and *shift* coefficients stored in ROM for the leftmost bit node group, as shown in (3.5). In order for FU_0 to access the appropriate contents, the outputs of rows 0, 1 and 6 of the top RAM must be cyclically left-shifted by 0, 1 and 1, respectively, before entering FU_0. Furthermore, at the end of the Bit Node Update step, the outputs of FU_0 need to be cyclically right-shifted back by 0, 1 and 1 before they are stored in the RAM in order for FU_0 to be able to access the appropriate cell contents during the Check Node Update of the next iteration. Thus, two shifting modules, called Shuffle Network, are necessary. One between the outputs of the RAM the inputs FUs and another one between the inputs of the RAM and the outputs of the FUs.

In order to reduce the hardware resource utilization of the FPGA, only one Shuffle Network is implemented between the outputs of the FUs and the inputs of the RAM, as shown in the top-level block diagram in Figure 3.2. This implementation is possible by implementing both left and right cyclic shift operations in the Shuffle Network. At the end of the Check Node Update step, instead of writing the outputs of the FUs back to the same locations as they were read, the outputs of the FUs are cyclically left-shifted by the *shift* coefficients to set up for the Bit Node Update. At the end of the Bit Node Update step, the output of the FUs are cyclically right-shifted before writing back to the RAM as previously discussed.

Furthermore, two sets of the *shift* coefficients are stored for more efficient ROM access. Consider the example, where the *row* and *shift* values are shown in (3.5). During the Bit

Table 3.3: *row*, *shift* and *ishift* coefficients in the ROM of the example

ROM Location	*row*	*shift*	*ishift*
0	0	0	3
1	1	1	2
2	6	1	1
3	7	0	2
4	2	2	3
5	8	2	1
6	9	0	2
7	3	1	3
8	10	2	1
9	4	0	3
10	11	1	1
11	5	2	2

Node Update step, the *shift* values stored in ROM are in the same order as in (3.5), as shown in Table 3.3. Thus, the ROM access for the *shift* coefficients is sequential during the Bit Node Update step because the top RAM row that is accessed is indexed by the *row* coefficients which are in the same ROM address location as the *shift* coefficients. However, during the Check Node Update step, the top RAM row access is sequential, so in order to avoid the need to search for the *shift* coefficient from the ROM, another set of *shift* coefficients are stored in the ROM, called *ishift*. Furthermore, since another set of *shift* coefficients is stored in the ROM, the Shuffle Network architecture can be further simplified by implementing only the cyclic right-shift operation and store the values $ishift = M - shift$ in the ROM. In summary, in order to obtain the *ishift* coefficients and their respective ROM address location, sort the list of *shift* coefficients in increasing order of their respective *row* coefficients and generate the *ishift* values using $ishift = M - shift$. The resultant *ishift* values and their ROM locations are shown in Table 3.3.

As shown later in Section 3.3, the output of each FU to be stored in the RAM is a 5-bit value. Since 360 FUs are used, the input and output of the Shuffle Network is $360 \times 5 = 1800$

bits wide. Furthermore, the Shuffle Network can cyclically right-shift the input of 360 5-bit values by 0 to 359 positions selected by an input. The Shuffle Network is implemented as a structure that consists of five barrel shifters. Each barrel shifter outputs a 360-bit sequence from cyclically right-shifting the 360-bit input by 0 to 359 positions selected by an input. More details on the architecture of the barrel shifter are presented in Section 3.4.

3.2.4 Special Case of Code Rates in Short Frames

In Table 3.2, some code rates are marked as special case code rates, yet these code rates in the short frame are not specially marked in the standard. The reason that these code rates are marked is that the memory mapping scheme in Algorithm 1 assumes that the row weight of submatrix $\mathbf{A}$ is always constant. However, the assumption is not true in the code rates that are marked as special case code rates in Table 3.2. In these code rates, the row weight of submatrix $\mathbf{A}$ is not always constant. Moreover, these code rates have row weights of anywhere between two and five different values, as shown in Table 3.4.

The various row weights of these code rates affects the Check Node Update step because the FUs no longer always access a constant number of top RAM rows, q, at a given time, especially since in some of these code rates, q is not an integer. This special characteristic of these code rates also affects the mapping of the edges to the non-zero elements in the $\mathbf{A}$ submatrix to the top RAM using Algorithm 1. Nevertheless, the periodicity of $M = 360$ still exists, which means if row i in $\mathbf{A}$, for $0 \leq i < p$, has a particular row weight, then the rows $i + p, i + 2p, i + 3p, \cdots, i + (M - 1)p$, which are from the same check node group, all have the same row weight. In Table 3.4, the check node group indices identify the check node groups in the particular code rate that has the particular row weight. Using the periodicity property of the row weights, during the Check Node Update step, the FUs read the number of rows from the top RAM according to the row weight values and check node group indices given in Table 3.4, instead of always reading q rows from the top RAM, which is the case for all other code rates.

Table 3.4: Row Weight of submatrix **A** of Problematic Code Rates

rate	row weight	check node group indices
1/5	1	6, 13, 14, 15, 20, 26, 27, 28, 30
	2	0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 16, 17, 18, 19, 21, 22, 23, 24, 25, 29, 31, 32, 33, 34, 35
4/9	2	4, 6, 16, 19
	3	0, 5, 7, 9, 10, 15, 18, 21, 23
	4	1, 3, 8, 11, 12, 13, 14, 17, 20, 22
	5	2, 24
11/15	7	3
	8	0, 4, 9
	9	2, 6, 8, 10
	10	1, 7, 11
	11	5
7/9	9	1
	10	0, 3, 4
	11	2, 5, 6, 7, 8, 9
37/45	14	0, 1, 2, 3
	15	7
	16	5
	17	4,6

For example, in code rate 37/45, where $p = 8$, the FUs reads the first 14 rows of the top RAM to process the check node group with index 0, which corresponds to processing rows $0, 8, 16, \cdots, 2872$ of submatrix **A**. Then, it reads the next 14 rows to process the check node group with index 1, which corresponds to rows $1, 9, 17, \cdots, 2873$ of submatrix **A**. Then, it reads the next 14 rows for check node group 2 and another 14 rows for check node group 3. Next, it reads 17 rows from the top RAM for the check node group 4, 16 rows for check

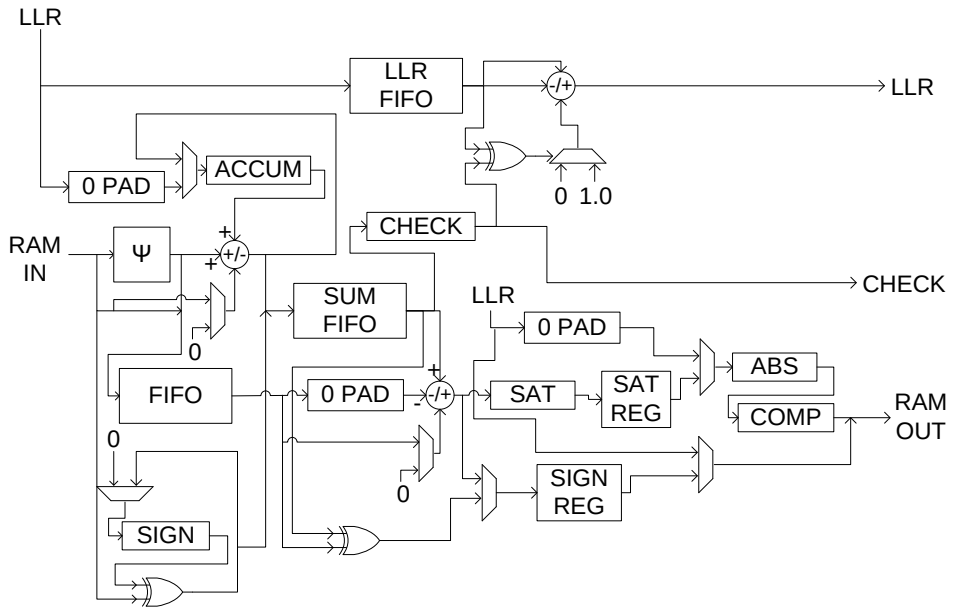

Figure 3.6: Block diagram of functional unit.

node group 5, 17 rows for check node group 6 and 15 rows for check node group 7.

For the generation of the ROM coefficients, the only modification to Algorithm 2 is the initialization of the vector LUT. Instead of initializing $LUT = [0, q, 2q, 3q, \cdots, (p-1)q]$, initialize $LUT = [0, rw_0, rw_0 + rw_1, rw_0 + rw_1 + rw_2, \cdots]$, where rw_i is the row weight of check node group index i. The rest of Algorithm 2 is executed the same way as with any other code rate.

3.3 Architecture of the Functional Units

The Functional Units (FUs) are used to compute the equations (2.30) and (2.32). The FU design is a modification and improvement of the serial FU architecture presented by Gomes et al. [34]. The modified FU design and architecture is presented in this section. The block diagram of a FU is shown in Figure 3.6. Each FU is a hybrid structure that is capable of performing either equation (2.30) or (2.32). Since the two calculations are not performed simultaneously in the SPA as described in Section 2.3, the two operations are combined into one module to reduce hardware resources. The operation of the FU module is described below using the block diagram in Figure 3.6 as reference.

The FU receives the values from the RAM through the RAM IN input. Recall from Section 3.2 that during the Check Node Update step, q rows from the top RAM and two rows from the bottom RAM are processed for each check node group. During the Bit Node Update step, for each bit node group the number of top RAM rows processed depends on the bit node degree, d_b, of the bit node group. Since q and the bit node degree vary for each code rate, the FU has adopted a serial input architecture where one row of the RAM is accessed per clock cycle, which means that the content of one cell in the row of the RAM is sent to the RAM IN input of one FU per clock cycle.

During the Check Node Update step, when each of the RAM values are inputted, its sign bit is sent to the XOR gate at the bottom left of Figure 3.6 where the sign bits of each RAM value is accumulated in the SIGN register. The SIGN register is initialized to the value 0 through the multiplexer at its input before every set of $q+2$ input RAM values. The magnitude of each RAM value goes through the ψ block, which performs the ψ function as shown in (1.1) and the implementation of the function in hardware is shown in Section 3.3.1. The output of the ψ function is added with the value in the ACCUM register and stored back into ACCUM to perform the summation in (2.30). The ACCUM register is also initialized to the value 0 through the multiplexer at its input by setting the LLR input to 0. The third input of the adder/subtractor selects whether the adder/subtractor performs an addition or subtraction. During the Check Node Update step, the multiplexer to the right of the ψ block in Figure 3.6 selects the value 0 for addition since the ψ outputs are to be added together. The outputs of the ψ function along with the sign bit of the RAM values are concatenated and stored in the FIFO, which is a first-in first-out queue. Once the ACCUM and SIGN registers have accumulated $q+2$ items, their values are concatenated and stored in the SUM FIFO. The output of the SUM FIFO is separated back into the ACCUM and SIGN register parts. The ACCUM register part is used by the subtractor/adder, where the ψ output, which is stored in the FIFO for each RAM value, is read out to subtract from the accumulated sum. The third input of the subtractor/adder is used to select between the subtraction or addition operations and is controlled by a multiplexer that outputs the value 0 for subtraction during the Check Node Update step. The output of the subtraction is saturated in the SAT block, temporarily stored in the SAT REG register for pipelining

purposes and compressed by the COMP block. These operations are discussed in more detail in the following subsections. The SIGN register part of the output of the SUM FIFO is used as input of the XOR gate before the SIGN REG register. The other input of the XOR gate is the sign bit of the RAM that is stored in the FIFO. The XOR gate performs the sign "subtraction" in the second part of (2.30). The output of the XOR gate is selected by the multiplexer to be temporarily stored in the SIGN REG register for pipelining purposes. The output of the SIGN REG register is selected by the next multiplexer to be concatenated with the output of the COMP block. The combined value is the output value of RAM OUT that is sent to the Shuffle Network and subsequently stored back to the RAM.

During the Bit Node Update step, d_b values are inputted to the FU from the RAM IN input. Since in (2.32) the sign calculations are included in the summation, as opposed to separate as in the Check Node Update step, the XOR gate and SIGN register are not used in the Bit Node Update step. The sign bit of the RAM value is simply separated from the magnitude part. The magnitude part is input into the ψ block and its output goes to the adder/subtractor where the summation is accumulated in the ACCUM register. During the Bit Node Update step, the multiplexer at the select input of the adder/subtractor outputs the sign bit of the RAM IN input. Thus, if the RAM IN value is positive, then the sign bit is 0 and the adder/subtractor performs addition, otherwise it performs subtraction. The ACCUM register is initialized to the value of the LLR input, which reads the values from the LLR Buffer, to add the λ_j term in (2.32) to the summation. Similar to the Check Node Update step, the output of the ψ function and the sign bit of the RAM IN input are stored in the FIFO. Once d_b values have been accumulated, the resultant sum in the ACCUM register is stored in the SUM FIFO. Similar to the Check Node Update step, the SUM FIFO output is input into the subtractor/adder, where each ψ output value stored in the FIFO are to be subtracted from the sum. Since the sign and magnitude values are used together in the calculation in the Bit Node Update step, the sign bit from the FIFO is selected by the multiplexer to be used as the select input of the subtractor/adder, such that if the sign of the value from the FIFO is positive the subtractor/adder performs subtraction, otherwise it performs addition. The value stored in the SUM FIFO is also the S_j value in equation (2.33) of the Hard Decision Making step. Thus, its sign bit, which is z_j in (2.34), is

temporarily stored in the CHECK register for pipelining purposes and used as the CHECK output of the FU that is sent to the Parity Check Module (PCM) and Decoded Message Buffer. The output of the subtractor/adder is separated into the magnitude and sign parts. The magnitude part is saturated through the SAT block, temporarily stored in the SAT REG register for pipelining purposes, and compressed by the COMP block. The sign part is selected by the multiplexer to be stored temporarily stored in the SIGN REG register and subsequently selected by the next multiplexer to be combined with the COMP block output to form the RAM OUT output of the FU.

According to the memory mapping scheme discussed in Section 3.2.1, 360 FUs are required for the implementation of the decoder. These 360 FUs operate in parallel and independent from each other, such that all rows or columns of the same check node group or bit node group, respectively, are processed simultaneously. Collectively, all 360 FUs begin all the calculations for a check node group or bit node group together, finish all the calculations together and begin the calculations of the next check node group or bit node group together. Furthermore, the 360 outputs of the 360 FUs to be written back to the RAM always belong to the same row in the RAM, as 360 values were read together from the RAM to be inputs of the 360 FUs.

Since 360 FUs are required in the implementation of the LDPC decoder, any reduction in the hardware resource utilization of one FU results in a 360 times reduction in the hardware resource utilization of the decoder. The following subsections discuss the modifications of the FU in Figure 3.6 from the original serial architecture FU design presented by Gomes et al. [34] to reduce hardware resource utilization. Furthermore, other modifications that result in better control flow and performance of the decoder are also presented.

3.3.1 Implementation of the ψ Function

The main modification of the FU design from the original serial FU architecture by Gomes et al. [34] is the usage of adders and the ψ function, instead of the boxplus and boxminus operations. The block diagram of the implementation of these two functions are shown in Figure 3.7 and Figure 3.8. In the FU design in Figure 3.6, the boxplus function is replaced by the adder/subtractor before the ACCUM register and the boxminus function is replaced

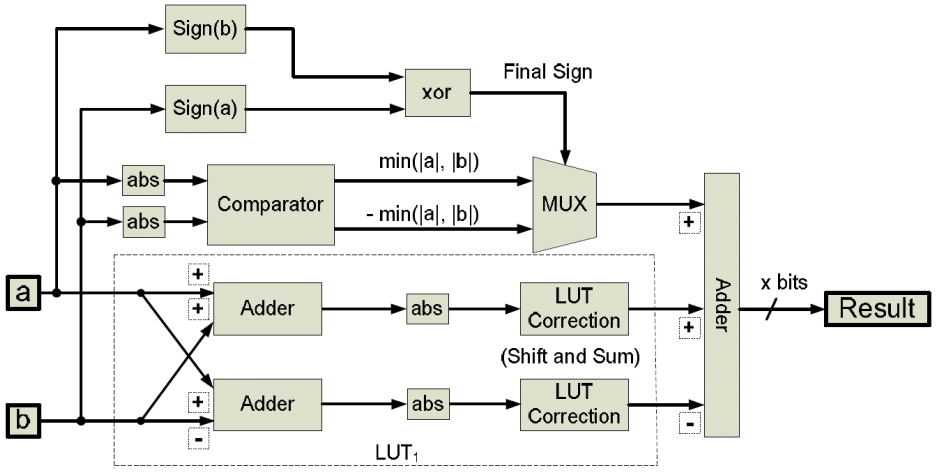

Figure 3.7: Block Diagram of the boxplus unit.

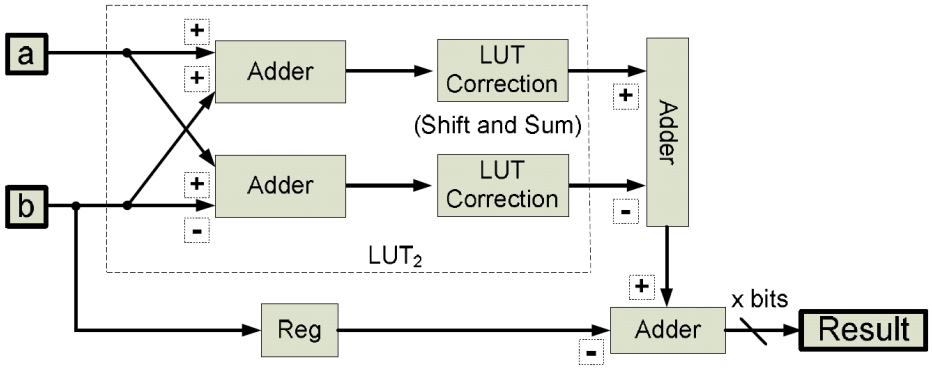

Figure 3.8: Block Diagram of the boxminus unit.

by the subtractor/adder at the output of the SUM FIFO. These modifications are possible with the usage of the ψ function to implement the equations (2.30) and (2.32) instead of the equations used by Gomes et al. [34]. Furthermore, these modifications reduce the hardware resource utilizations of the decoder as long as the implementation of the ψ function is less complex than the boxplus and boxminus functions in Figure 3.7 and Figure 3.8. Thus, the implementation of the ψ function is discussed in this section.

The ψ block in Figure 3.6 is the implementation of the ψ function in (1.1). The graph of the function is shown in Figure 1.1 and in Figure 3.9. Notice from Figure 3.9 that for large input values, the output becomes arbitrarily small. Due to its non-linearity and constantly increasing slope, Zhang et al. [20] have suggested a variable precision quantization scheme,

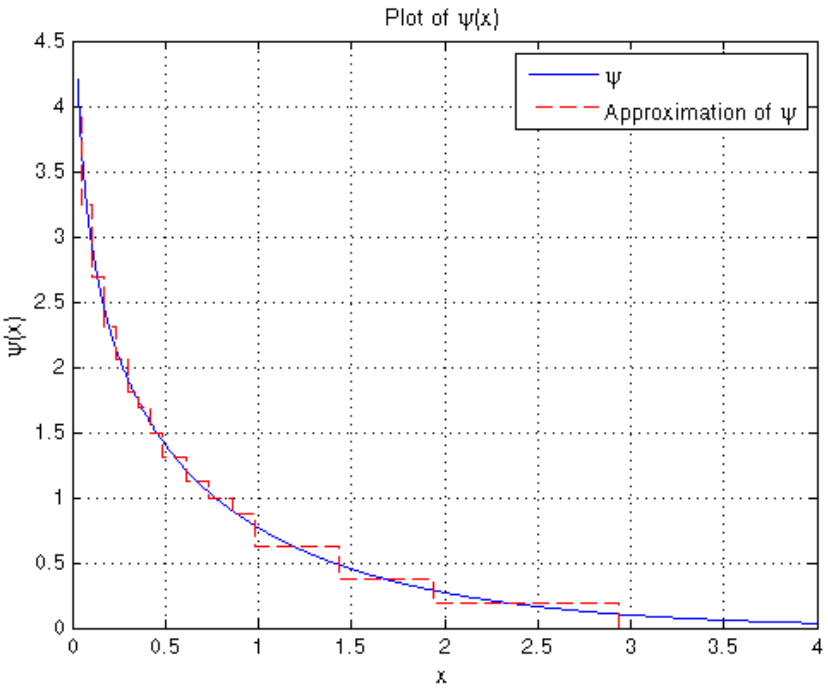

Figure 3.9: Graph of the ψ function and its approximation.

as presented in Section 1.1. However, using this quantization scheme for both the inputs and outputs of the ψ block, requires the outputs to be converted from the variably quantized format to two's complement format before they can be used by the adder/subtractor and accumulated in the ACCUM register in Figure 3.6, which increases hardware resource utilization. Thus, the FU utilizes the variable precision quantization scheme only for the inputs of the ψ function. The outputs of the ψ block are 6-bit fixed point value in 2.4 format, where the most significant two bits are the integer part and the least significant four bits are the fractional part of the value. Using the variable quantization scheme, Table 3.5 is generated. Table 3.5 is sorted in sequential order of the binary value of the input and the output decimal ψ values are not quantized, but the binary output values are generated by quantizing the decimal values in 2.4 format.

Direct implementation of Table 3.5 requires a 6-bit input, 6-bit output Boolean function, which is a $2^6 \times$ 6-bit LUT. Oh and Parhi [21] have suggested a method of reducing the LUT size as discussed in Section 1.1. Applying a similar method, the "Comp." column in Table 3.5 is generated. As a result, the ψ function can be implemented as a $2^4 \times$ 6-bit LUT

Table 3.5: ψ Function Quantization Scheme

Input		Output			Input		Output		
Dec.	Bin.	Dec.	Bin.	Comp.	Dec.	Bin.	Dec.	Bin.	Comp.
0.00000	0.00000	Inf	11.1111	11.1111	4.000	100.000	0.0366	00.0001	00.0000
0.03125	0.00001	4.1590	11.1111		4.125	100.001	0.0323	00.0001	
0.06250	0.00010	3.4661	11.0111	11.0100	4.250	100.010	0.0285	00.0000	
0.09375	0.00011	3.0610	11.0001		4.375	100.011	0.0252	00.0000	
0.12500	0.00100	2.7739	10.1100	10.1011	4.500	100.100	0.0222	00.0000	
0.15625	0.00101	2.5515	10.1001		4.625	100.101	0.0196	00.0000	
0.18750	0.00110	2.3701	10.0110	10.0101	4.750	100.110	0.0173	00.0000	
0.21875	0.00111	2.2170	10.0011		4.875	100.111	0.0153	00.0000	
0.25000	0.01000	2.0846	10.0001	10.0001	1.000	101.000	0.7719	00.1100	00.1010
0.28125	0.01001	1.9682	01.1111		1.125	101.001	0.6737	00.1011	
0.31250	0.01010	1.8644	01.1110	01.1101	1.250	101.010	0.5895	00.1001	
0.34375	0.01011	1.7708	01.1100		1.375	101.011	0.5169	00.1000	
0.37500	0.01100	1.6856	01.1011	01.1011	1.500	101.100	0.4539	00.0111	00.0110
0.40625	0.01101	1.6076	01.1010		1.625	101.101	0.3990	00.0110	
0.43750	0.01110	1.5356	01.1001	01.1000	1.750	101.110	0.3511	00.0110	
0.46875	0.01111	1.4689	01.1000		1.875	101.111	0.3092	00.0101	
0.50000	0.10000	1.4063	01.0111	01.0101	2.000	110.000	0.2723	00.0100	00.0011
0.53125	0.10001	1.3488	01.0110		2.125	110.001	0.2400	00.0100	
0.56250	0.10010	1.2944	01.0101		2.250	110.010	0.2116	00.0011	
0.59375	0.10011	1.2432	01.0100		2.375	110.011	0.1866	00.0011	
0.62500	0.10100	1.1950	01.0011	01.0010	2.500	110.100	0.1645	00.0011	
0.65625	0.10101	1.1494	01.0010		2.625	110.101	0.1451	00.0010	
0.68750	0.10110	1.1062	01.0010		2.750	110.110	0.1280	00.0010	
0.71875	0.10111	1.0652	01.0001		2.875	110.111	0.1130	00.0010	
0.75000	0.11000	1.0262	01.0000	01.0000	3.000	111.000	0.0997	00.0010	00.0000
0.78125	0.11001	0.9891	01.0000		3.125	111.001	0.0879	00.0001	
0.81250	0.11010	0.9538	00.1111		3.250	111.010	0.0776	00.0001	
0.84375	0.11011	0.9200	00.1111		3.375	111.011	0.0685	00.0001	
0.87500	0.11100	0.8878	00.1110	00.1110	3.500	111.100	0.0604	00.0001	
0.90625	0.11101	0.8569	00.1110		3.625	111.101	0.0533	00.0001	
0.93750	0.11110	0.8274	00.1101		3.750	111.110	0.0470	00.0001	
0.96875	0.11111	0.7991	00.1101		3.875	111.111	0.0415	00.0001	

Table 3.6: ψ function LUT

Input	Output	Input	Output
0000	11.1111	1000	01.0101
0001	11.0100	1001	01.0010
0010	10.1011	1010	01.0000
0011	10.0101	1011	00.1110
0100	10.0001	1100	00.0000
0101	01.1101	1101	00.1010
0110	01.1011	1110	00.0110
0111	01.1000	1111	00.0011

instead of a $2^6 \times 6$-bit LUT. The resultant LUT to approximate the ψ function is shown in Table 3.6 and its graph is shown with the dashed lines in Figure 3.9.

One parameter that can be used to measure the precision of the ψ function approximation in Table 3.6 compared to the actual value of ψ as given in (1.1) is the root mean square error (RMSE). The RMSE between two vectors, $\mathbf{y}_1 = (y_{1,0}, y_{1,1}, \cdots, y_{1,n-1})$ and $\mathbf{y}_2 = (y_{2,0}, y_{2,1}, \cdots, y_{2,n-1})$, of length n is given by the following equation:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=0}^{n-1}(y_{1,i} - y_{2,i})^2}{n}} \tag{3.7}$$

Since (3.7) uses a vector of discrete values for $\mathbf{y}_1$ and $\mathbf{y}_2$, rather than a continuous function, a vector $\mathbf{x}$ is selected to be a sequence of sequential value between 0.03 and 4 in steps of 0.000001 to compute $\psi(\mathbf{x})$ and the approximation of $\psi(\mathbf{x})$. Applying (3.7) on the two resultant vectors, the RMSE value is found to be 0.0719. Since the RMSE value of the approximation of the ψ function is not published for the decoders reviewed in Section 1.1, no comparisons are drawn from the result. However, the RMSE result presented can be used as a comparison benchmark for future improvements on the approximation of the ψ function.

Oh and Parhi [21] also propose a compression function (COMP) that compresses the values of the RAM OUT output. According to Figure 3.6, the magnitudes of the RAM IN inputs enter the ψ block immediately after they are inputted. Since the ψ function LUT

Table 3.7: Compression Function LUT

Input	Output	Input	Output
00.0000	0000	00.100x	1000
00.0001	0001	00.101x	1001
00.0010	0010	00.110x	1010
00.0011	0011	00.111x	1011
00.0100	0100	01.0xxx	1101
00.0101	0101	01.1xxx	1110
00.0110	0110	10.xxxx	1111
00.0111	0111	11.xxxx	1100

only has a 4-bit input value, even if the RAM IN input has more than 4 bits, it would need to be rounded off or truncated to be used by the ψ block. Thus, the result of the subtractor/adder is compressed through the COMP block to reduce the two's complement value into a 4-bit value. The generated 4-bit value corresponds to the 4-bit input of the ψ block, so the compression function is obtained by finding the set of input values that would generate the same ψ output, group them together and assign them the 4-bit value in Table 3.6 that generates the corresponding ψ output value. The resultant LUT is shown in Table 3.7. By doing so, the values in the RAM are 4 magnitude bits + 1 sign bit = 5 bits wide. Furthermore, from Table 3.7, the input values of the COMP block are 6 bits wide and they are interpreted with the 2.4 format. Any value larger than 11.1111 in binary, which is 3.9375 in decimal, is saturated to 6 bits because from Table 3.5, any input to the ψ function larger than the decimal value 3.0 outputs 000000. Thus, the SAT block is used before the COMP block to saturate the values to 6 bits wide.

3.3.2 Usage and Design of the SUM FIFO

The SUM FIFO is one of the blocks that is not in the serial FU architecture by Gomes et al. [34], where a sum register is used instead. The use of the SUM FIFO improves the control flow of the FU that is made difficult by the variation in the number of RAM IN values

that are inputted into the FU. Consider the normal frame code rate 1/4 LDPC code in the DVB-S2 standard. In the **A** submatrix of this code, there are columns with twelve, three, two and one non-zero elements, i.e. the bit node degrees, d_b, of the bit node groups can be any of the four values. During the Bit Node Update step, the FU begins with processing sets of twelve RAM IN values. When the last set of twelve RAM IN values is inputted, RAM IN inputs sets of three RAM IN values. However, when the first set of three RAM IN values have finished inputting, the FU is still producing the RAM OUT values using sum value from the last set of twelve RAM IN values and subtracting it from the values in the FIFO. If a sum register is used instead of the SUM FIFO, the inputting of the set of three RAM IN values must be suspended after the last set of twelve RAM IN values are inputted, until the sum register value is ready to be overwritten. However, using the SUM FIFO, the set of three RAM IN values does not need to be suspended and the resultant summation value is queued in the SUM FIFO, while the RAM OUT values from the last set is computed. By using the SUM FIFO instead of the sum register, the control flow of the FU is improved because the flow of RAM IN values does not need to be suspended in situations described above. However, the trade-off is an increase in hardware resource utilization. Nevertheless, the size of the SUM FIFO may be reduced by making an observation about the RAM IN values that are inputted.

The ACCUM register and adder/subtractor in Figure 3.6 perform the summation part of equations (2.30) and (2.32). Given the code specified in the DVB-S2 standard [6], the maximum number of RAM values that are added together in (2.30) and (2.32) are 30 and 13, respectively. Since the FU performs both the Check Node Update and Bit Node Update steps, the maximum number of values that would be accumulated in the FU is 30. Thus, the bit width of the ACCUM register and the adder/subtractor are equal to the number of bits of each value that is added, which is 1 sign bit + the 6-bit output of ψ, plus the ceiling of $\log_2$ of the number of items added together. Thus, the ACCUM register size is $7 + \lceil \log_2(30) \rceil = 7 + 5 = 12$. However, since the sum stored in the ACCUM register is subsequently stored in the SUM FIFO and subtracted by exactly one ψ output value, saturated and then compressed to generate the output values of the FU, the width of the SUM FIFO can be reduced.

According to Table 3.6, the maximum output of the ψ function is 11.1111 in binary, which is 011.1111 in two's complement binary and 3.9375 in decimal. Thus, the maximum possible sum stored in the ACCUM register is $3.9375 \times 30 = 118.125$, which is 01110110.0010 in a 12-bit two's complement binary. Subsequently, this sum must be subtracted by one value from the FIFO at a time, whose maximum is the maximum output of the ψ function, namely 3.9375 in decimal. The difference is then compressed through the LUT given in Table 3.7. However, according to Table 3.7, any value greater than 11.0000 in binary, or 3.0 in decimal, is compressed to 1100 in binary. Thus, if the sum stored in the ACCUM register is greater than $3.0 + 3.9375 = 6.9375$ in decimal or 0110.1111 in two's complement binary, regardless of the FIFO value that is subtracted from it, the result is always greater than or equal to 3.0 in decimal or 011.0000 in two's complement binary. This result is passed to the ABS block, which converts the value to a number greater than or equal to 11.0000 in binary. Finally, any value greater than or equal to 11.0000 in binary is compressed to 1100 through the COMP block and according to Table 3.7. Therefore, the value in the ACCUM register can be reduced from 12 bits to 8 bits before storing in the SUM FIFO and the SUM FIFO is only 8 bits wide because if the sum result stored in the SUM FIFO is greater than or equal to 0110.1111 in binary, the RAM OUT output is always 1100 regardless of the values stored in the FIFO.

3.3.3 LLR Value Update

In the Hard Decision Making step of the SPA algorithm, the LLR values (λ_j) always remain the same and in essence, the second part of (2.33) adjusts the value of the LLR in order to produce S_j and its sign bit, z_j, is used for parity checking. (2.33) has no problems as long as the LLR values and the summation have infinite or high precision. A performance degradation occurs when the decoder has finite precision and is restricted by the hardware resources. In infinite precision, assume that the magnitude of the LLR value in a certain position is close to 0 and the sign bit is positive, which means the LLR value indicates that its hard-decision bit is 0 and it has a lower reliability. In addition, assume that the correct transmitted hard-decision bit of the codeword is actually 1. Furthermore, assume that the summation part of the (2.33) is negative and the magnitude is also close to 0 but larger than

the magnitude of the LLR value. Applying (2.33), S_j is a negative value, which decodes to hard-decision bit 1 and decoding is correct. In finite precision, due to truncation of values and approximations of ψ function, after multiple iterations of the SPA, the small magnitude of the summation part may become smaller than the magnitude of the LLR value, in which case the result of (2.33) remains positive and the hard-decision bit is erroneously decoded to 0 causing the decoder to continue the SPA instead of exiting with the correct decoded bit.

To improve the performance of the decoder, the LLR values may be updated by adding or subtracting by 1.0 decimal. The sign bit of the LLR value and the sign bit of the SUM FIFO value stored in the CHECK register determine if any changes to the LLR value are necessary. If the sign of the LLR value is the same as value in the CHECK register, then the LLR value and S_j have the same sign, so no change is needed. If the sign of the LLR value and the value in the CHECK register are different, when the LLR value is updated by adding 1.0 if LLR is negative and subtracting 1.0 if LLR is positive. By doing so, the LLR value adjusts its reliability value towards what S_j suggests its sign is and reduces the number of iterations necessary.

3.3.4 The Initialization Step

During the Initialization step of the SPA, the LLR values are inputted into the decoder into the LLR Buffer. Subsequently, these values are stored in the RAM when assigning each LLR value to a bit node message box. Since the RAM values are compressed using the COMP block in the FU, the LLR values also need to be compressed before storing into the RAM in the Initialization step. In order to avoid implementing a set of COMP blocks external to the FU, the LLR values are input into the FUs during the Initialization step through the LLR input in the middle of the block diagram in Figure 3.6. The multiplexer selects the LLR value instead of the SAT REG register value to be used by the ABS and COMP blocks. The sign bit of LLR is selected by the multiplexer in the bottom right to be combined with the output of the COMP block to form the RAM OUT output value. By doing so, the hardware resources for the ABS and COMP blocks are efficiently utilized because the calculation of (2.30) and (2.32) for the Check Node Update and Bit Node Update steps are not being carried out during the Initialization step.

3.4 Architecture of the Parity Check Module

The Parity Check Module (PCM) is used to verify the parity-check equations during the Hard Decision Making step in the SPA. The parity check equations as shown in (2.24) that are used during the encoding of DVB-S2 LDPC codes are also used in the decoder, except they are expressed as follows:

$$\begin{aligned} 0 &= a_{0,0}u_0 \oplus a_{0,1}u_1 \oplus \cdots \oplus a_{0,K-1}u_{K-1} \oplus p_0 \\ 0 &= a_{1,0}u_0 \oplus a_{1,1}u_1 \oplus \cdots \oplus a_{1,K-1}u_{K-1} \oplus p_0 \oplus p_1 \\ 0 &= a_{2,0}u_0 \oplus a_{2,1}u_1 \oplus \cdots \oplus a_{2,K-1}u_{K-1} \oplus p_1 \oplus p_2 \\ &\vdots \\ 0 &= a_{N-K-1,0}u_0 \oplus a_{N-K-1,1}u_1 \oplus a_{N-K-1,K-1}u_{K-1} \oplus p_{N-K-2} \oplus p_{N-K-1} \end{aligned} \tag{3.8}$$

where the difference is that the parity bits are moved to the right hand side of the equations. To verify the parity-check equations, the N hard-decision decoded bits, z_j, generated at the CHECK output of the FUs are inserted into the parity-check equations and if all parity-check equations are true, i.e. the right hand side of the equations equals to 0, the parity-check equations are verified. However, since each of the twenty-one different frame and code rate combinations have different parity-check equations, a PCM that verifies each of them separately is required. Too many hardware resources are required if each set of parity check equations for each code rate is implemented separately on the FPGA. The PCM architecture presented in this section has the ability to verify the parity-check equations for any LDPC code in the DVB-S2 standard. The design of the PCM is similar to the design of a DVB-S2 LDPC encoder because in both cases, the parity-check equations are implemented, except the encoder uses the message bits to generate the parity bits and the PCM uses both the message bits and the parity bits to verify whether or not the parity-check equations are satisfied. Thus, the architecture of the PCM is based on the DVB-S2 LDPC encoder designed by Gomes et al. [35]. The algorithm of the PCM is shown in Algorithm 3.

Algorithm 3 Algorithm of the Parity Check Module

Initialize $\mathbf{S}$ to all 0

Get the first $\mathbf{v}$ from the FUs

for each of the rows in Appendix B **do**

  for r = each element in each row **do**

    $\mathbf{S}(r \bmod p, :) = \mathbf{S}(r \bmod p, :) \oplus \mathrm{rot}_{r \div p}(\mathbf{v})$

  end for

  $\mathbf{v}$ = next $\mathbf{v}$ from the FUs

end for

Get the first $\mathbf{p}$ from the FUs

for $i = 0$ to $p - 1$ (each row index of $\mathbf{S}$) **do**

  for $j = 0$ to 1 **do**

    if $i = p - 1$ (last row of $\mathbf{S}$) and $j = 1$ **then**

      $\mathbf{S}((i + j) \bmod p, :) = \mathbf{S}((i + j) \bmod p, :) \oplus \mathrm{rot}_1(\mathbf{p})$

    else

      $\mathbf{S}((i + j) \bmod p, :) = \mathbf{S}((i + j) \bmod p, :) \oplus \mathbf{p}$

    end if

  end for

  $\mathbf{p}$ = next $\mathbf{p}$ from the FUs

end for

if number of non-zero values in $\mathbf{S} \leq 0$ **then**

  PASS PARITY CHECK

else

  FAIL PARITY CHECK

end if

Consider the binary matrix $\mathbf{S}$ as follows:

$$\mathbf{S} = \begin{bmatrix} S_0 & S_p & S_{2p} & \cdots & S_{(M-1)p} \\ S_1 & S_{p+1} & S_{2p+1} & \cdots & S_{(M-1)p+1} \\ S_2 & S_{p+2} & S_{2p+2} & \cdots & S_{(M-1)p+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ S_{p-1} & S_{2p-1} & S_{3p-1} & \cdots & S_{N-K-1} \end{bmatrix} \tag{3.9}$$

where $M = 360$ and p is the code rate dependent parameter as shown in Table 2.1. During the operations of the PCM, each element of $\mathbf{S}$ is initialized to 0. Recall from Section 3.3 that the 360 FUs generate 360 CHECK output values of 1 bit each simultaneously, which are 360 of the z_j values in (2.34) and are inputted into the PCM. If these 360 bits of CHECK output values are corresponds to the message part of the codeword, i.e. they correspond to z_j where $0 \leq j \leq K-1$, then they form a vector, $\mathbf{v}$, that is used to update the $\mathbf{S}$ matrix as they get generated using the following equation:

$$\mathbf{S}(r \bmod p, :) = \mathbf{S}(r \bmod p, :) \oplus \mathrm{rot}_{r \div p}(\mathbf{v}) \tag{3.10}$$

where $\mathbf{S}(x, :)$ is the 360-bit vector formed by row x of matrix $\mathbf{S}$, $\mathrm{rot}_y(\mathbf{v})$ is the $\mathbf{v}$ vector cyclically right-shifted by y positions, r is a value obtained from Appendix B for a given code rate and $\oplus$ is the module-2 addition operator. Lastly, let $\mathbf{p}$ be the 360-bit vector formed by the 360 CHECK output values from the FUs that corresponds z_j where $K \leq j \leq N-1$. Using the aforementioned notations, the operation of Algorithm 3 is discussed below.

The algorithm initializes all the values in $\mathbf{S}$ to 0. As the 360 CHECK output bits are generated from the FUs and inputted into the PCM, the matrix $\mathbf{S}$ is updated according to equation (3.10). Since the FUs first generate the message part of the potential codeword, these bits form the $\mathbf{v}$ vector and the first **for** loop is executed. According to Algorithm 3, each vector $\mathbf{v}$ updates as many rows of $\mathbf{S}$ as there are elements in the rows in Appendix B, which is the bit node degree, d_b, of the bit node group. When all the values in Appendix B are exhausted, all the CHECK output bits of the message part of the potential codeword are also exhausted and the first **for** loop exits. The second **for** loop begins and the 360 CHECK output bits correspond to the redundant checking part of the potential codeword and forms the vector $\mathbf{p}$. During the second **for** loop, the two consecutive rows of the $\mathbf{S}$ are updated for every $\mathbf{p}$ vector using the following equation:

$$\mathbf{S}((i+j) \bmod p, :) = \mathbf{S}((i+j) \bmod p, :) \oplus \mathbf{p} \tag{3.11}$$

Notice that (3.11) is similar to (3.10), except the row indices of $\mathbf{S}$ are different and there is no cyclic right-shift operation on the $\mathbf{p}$ vector, except when the last $\mathbf{p}$ vector is inputted, in which case $\mathbf{p}$ is cyclically right-shifted by one. Upon the completion of the second **for** loop, all the

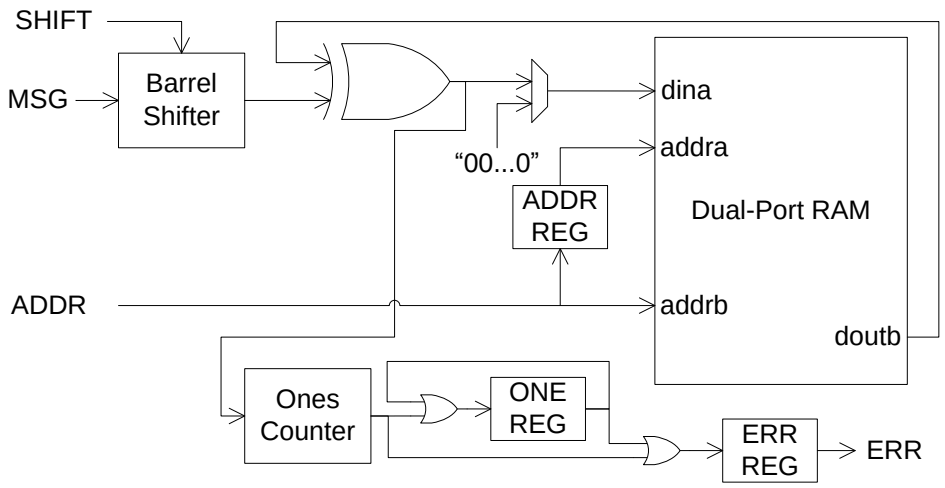

Figure 3.10: Block diagram of parity check module.

360 CHECK outputs from the FUs are also exhausted and the resultant matrix **S** contains the result of all the parity-check equations. Each element in the **S** matrix corresponds to the result of one parity-check equation in (3.8). Thus, if all the elements in the **S** matrix are equal to zero, then the potential codeword is said to pass parity-check verification and becomes the decoded codeword, otherwise, decoding is not complete and the decoder returns to the Check Node Update step and iterates.

It is important to note that Algorithm 3 only works with control flow of the decoder as described in this chapter due to the order that the **v** and **p** are generated. In the decoder presented when the FUs generate the message part of the potential codeword, the 360 CHECK output bits are generated in sequential order, i.e. bits $0 \rightarrow 359$ of the codeword are generated, followed by bits $360 \rightarrow 719$, and so on. When the redundant checking part of the potential codeword is generated, the sets of 360 CHECK output bits are generated as follows: bits $0, p, 2p, \cdots, 359p$ of the redundant check part are generated, followed by bits $1, p+1, 2p+1, \cdots, 359p+1$, and so on. According to this order of bit generation, after the first **for** loop in Algorithm 3, the **S** matrix contains the $\oplus$ sum of all $a_{x,y}u_x$ in equation (3.8). The second **for** loop in Algorithm 3 completes the parity check equations by applying $\oplus p_z$ to the values already in **S**.

The block diagram of the PCM is shown in Figure 3.10. The main components of the PCM are the barrel shifter (BS), dual-port RAM and the ones counter (OC). The 360

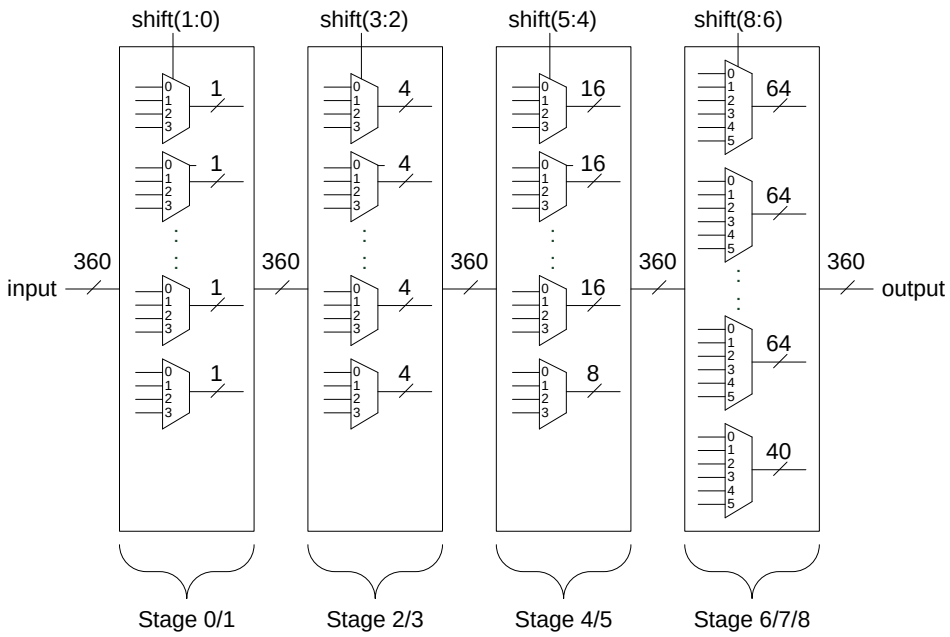

Figure 3.11: Block diagram of the barrel shifter.

CHECK output bits from the FUs enter the PCM through the MSG input. The MSG input is shifted by the BS by the number of positions as selected by the SHIFT input. The values inputted into the SHIFT input of the PCM are the same *shift* values generated by Algorithm 2. The BS performs the $\text{rot}_y(\mathbf{v})$ and $\text{rot}_1(\mathbf{p})$ operations in (3.10) and (3.11), respectively.

The BS is a cyclic right-shift operation that is able to rotate 360 bits of data by any number of positions. The BS used in this design is a multi-stage multi-level logarithmic shifter as discussed by Neto and Vestias [45]. The BS consists of 3 2-stage shifters (i.e. 4:1 multiplexers) and a final stage of 3-stage shifter (i.e. 8:1 multiplexers). The block diagram is shown in Figure 3.11. The reason for this selection is because recall from Section 2.1 that the Virtex-II Pro FPGA can realize a 4:1 multiplexer using one slice and 8:1 multiplexers using two slices. Thus, using 4:1 or 8:1 multiplexers, the BS can be mapped efficiently to the FPGAs.

The output of the BS goes into one of the inputs of the XOR gate, which performs the $\oplus$ operation. The other input of the XOR gate is the doutb output of the dual-port RAM, where the $\mathbf{S}$ matrix is stored. The doutb output of the RAM is selected by the addrb input of the RAM, which is controlled by the ADDR input. The values of the ADDR input are either

the values of $r \bmod p$, which is stored in the ROM along with the other values mentioned in Section 3.2.2, or a counter, which outputs $(i + j) \bmod p$. The output of the XOR gate is selected by the multiplexer to be stored back in the RAM through the dina input of the RAM to complete the assignment of the rows of **S** in (3.10) and (3.11). The dina input of the RAM writes into the RAM location selected by the addra input, which is the value of ADDR delayed by one clock cycle through the ADDR REG register. The delay is added because the doutb output has a one clock cycle delay, so the dina input value is not computed until one clock cycle after the ADDR input selects the row in the RAM to output in doutb.

Since the dual-port RAM stores the **S** matrix in (3.9), its dimensions are $p \times M$ bits. In order to support all the code rates in the DVB-S2 standard, where the maximum $p = 135$, the RAM is 135×360 bits. A dual port RAM is used since the RAM values are read out from one port, looped back and XORed with the incoming bits in order to evaluate equations (3.10) and (3.11). The dual-port RAM is implemented on the FPGA using BRAMs. Recall from Section 2.1 that the BRAM can have a 512×36 bit configuration. Thus, the dual-port RAM is implemented using 10 18 Kb BRAMs on the Virtex-II Pro FPGA.

Notice that during the second **for** loop, rows 0 and 1 of the **S** matrix are updated first, followed by rows 1 and 2. After the PCM updates rows 1 and 2 during the second **for** loop, row 1 is never updated again. Thus, after rows 1 and 2 update the contents of row 1 are the result of the parity-check equations in that row, so row 1 is reset to a vector of zeros through the multiplexer at the dina input and the contents of row 1, which is at the output of the XOR gate, are sent to the OC. Subsequently, after rows 2 and 3 are updated, the contents of row 2 are sent to the OC, after rows 3 and 4 are updated, the contents of row 3 are sent to the OC, and so on until finally rows p and 0 are updated, the contents of both rows are sent to the OC and the RAM is completely reset to zeros to prepare for the next iteration.

At the OC, the 360-bit vectors are counted for the number of non-zero elements. After every 360-bit vector is counted, the 1-bit result is accumulated in the ONE REG register. The OC does not need to count the exact number of non-zero elements in the 360-bit vector, instead it simply needs to determine whether or not there are any non-zero elements in matrix **S** because the decoder does not require the exact number of parity-check equations that are not satisfied. Thus, if there are no non-zero elements in the 360-bit vector the

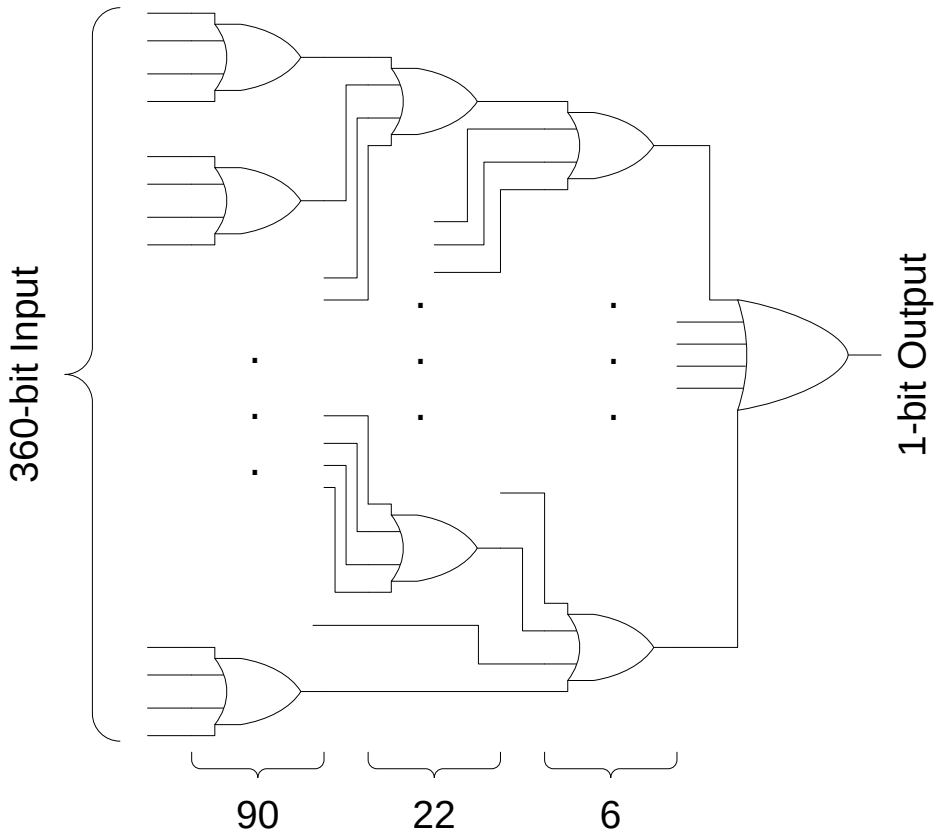

Figure 3.12: Block diagram of ones counter.

output of the OC is 0, otherwise it is 1. The OC is designed as a tree of 4-input OR gates, except for the last stage that has a 6-input OR gate, as shown in Figure 3.12. The use of 4-input OR gates is because recall from Section 2.1 that the Virtex-II Pro FPGA's slices have two 4-input LUTs, so the 4-input OR gate maps efficiently to the FPGA. In addition, the slices have internal circuits that can implement a 6-input function using only one slice, which means the 6-input OR gate at the final stage only requires one slice. If the final stage of the OC is implemented as two 3-input OR gates and one 2-input OR gate, then three LUTs, which is more than one slice, are required.

After the processing of rows p and 0 of the **S** matrix, the output of the OC bypasses the storage in the ONE REG register, combines with the value in the ONE REG register and is stored in the ERR REG register. Thus, when the contents of all the rows in the **S** matrix are processed by the OC, the ERR REG register stores the value that indicates whether or not there any non-zero results among all the parity-check equations and its value is sent out

of the PCM through the ERR output to the controller of the decoder. If the ERR output is 0, then all the parity-check equations are satisfied, otherwise the value is 1 and one or more parity-check equations do not equal to 0. Using the control flow of the PCM as described, the ERR output is generated 3 clock cycles after the set of 360 MSG input bits of the last bit node group is inputted.

3.5 Architecture of the LLR and Decoded Message Buffers

The LLR Buffer and the Decoded Message Buffer are a serial-to-parallel converter and a parallel-to-serial converter, respectively. Their architectures are based on the deserializer and serializer designs shown by Defossez and Sawyer [46]. The LLR Buffer is used to store the incoming LLR values. Since the possible block lengths are $N = 16200$ and 64800, the LLR Buffer must be able to store the maximum of the two, which is 64800 values. In addition, since the FUs access the LLR values in groups of 360, as the 360 FUs operate simultaneously, the RAM of the LLR Buffer is 360 values wide. Furthermore, as mentioned in Section 3.1, the LLR values are 6 bits wide, so the dimensions of the RAM of the LLR Buffer are $180 \times 360 \times 6$ bits. Since the Virtex-II Pro FPGA can use 18 Kb BRAMs for memory resources, the LLR Buffer uses sixty BRAMs in the 512×36 bits configuration.

The values stored in the LLR Buffer are 6-bit two's complement values to facilitate the summation that takes place inside the FUs. The 6-bit LLR values use the fixed point 3.3 format, where the MSB is the sign bit followed by two bits to form the integer portion and the least significant three bits form the fractional portion. There is one exception to the two's complement value, where the value 100.000 in binary is interpreted as -0.0_d in decimal instead of -4.0. This special consideration is used to improve the performance of the decoder. During the truncation or rounding of the LLR values to 6-bit two's complement values, there are cases where negative values with a small magnitude are quantized to 0, in which case it would be treated as a positive value. During the Initialization step of the SPA, the LLR values are stored in the RAM after being compressed in the FUs, yet the sign bit

remains intact. If the sign of one LLR value is inverted because of quantization, during the Check Node Update step, the product of the signs in (2.30) becomes inverted, which affects signs of the output of the FUs for every step thereafter. Thus, in order to avoid the effect of the sign inversion during quantization, the LLR values use 100.000 in binary to represent -0 in decimal during the Initialization step, which after the compression in the FU, stores 11100 instead of 01100 in the RAM.

The Decoded Message Buffer is used to store the potential outgoing decoded message during every iteration. If the PCM asserts that all parity-check equations are verified, then the Decoded Message Buffer outputs the decoded message serially. Furthermore, since the LDPC codes in the DVB-S2 standard are systematic and only the message part of the decoded codeword is outputted of the decoder, the RAM of the Decoded Message Buffer only needs to store the message part of the potential codeword for any code rate. The largest number of bits in the message part of the LDPC codes in the DVB-S2 standard is when the code rate is 9/10 in normal frames, where $K = 58320$. In addition, 360 decoded message bits are generated at a time by the CHECK outputs of the 360 FUs, so the RAM size of the Decoded Message Buffer is 162×360 bits, which is implemented on the FPGA using ten 18 Kb BRAMs using the 512×36 bits configuration.

The decoder design assumes that the LLR values are streamed in serially, where one LLR value is inputted per clock cycle. Thus, the LLR Buffer is designed to read one LLR value at a time. However, if the module external to the decoder that generates the LLR values has the ability to output more than one LLR value at a time, the LLR Buffer may be modified to accommodate the change. Similarly, the Decoded Message Buffer may also be modified to stream out more than one bit of decoded message bits at a time. These modifications are not discussed because they are beyond the scope of this thesis.

3.6 Architecture of the 180-FU and Hybrid 360/180-FU Decoders

In order to reduce hardware resource utilization, Gomes et al. [18] propose a method to systematically reducing the number of FUs from 360 to any factor, L, of 360, by essentially subsampling the RAM data. The method divides the columns of the RAM into L groups and process $\frac{M}{L}$ columns one after another. In other words, columns $0, L, 2L, 3L, \cdots$ are processed together by $\frac{M}{L}$ FUs, then columns $1, 1+L, 1+2L, 1+3L, \cdots$ are processed, then $2, 2+L, 2+2L, 2+3L, \cdots$, and so on.

By using a similar method, the LDPC decoder presented in the previous sections is reduced from 360 FUs to 180 FUs, which reduces the hardware resource utilization by almost half. The trade-off from the hardware resource reduction is that the throughput is also reduced by half. Since only 180 FUs are used, the RAM is changed to only have 180 values per row. Furthermore, the RAMs in the PCM, LLR Buffer, Decoded Message Buffer are also changed to have 180 values per row. These changes split the original RAMs in the 360-FU version of the decoder into two RAMs. The RAM generated by columns $0, 2, 4, 6, \cdots, 358$ of the original RAM is labelled even half and the RAM generated by columns $1, 3, 5, 7, \cdots, 359$ of the original RAM is labelled odd half. Figure 3.13 shows a diagram of how the RAM is divided.

When the RAM is accessed in the 180-FU decoder, the even and odd halves are accessed one after another. In the 360-FU decoder, each FU processes the values in one column of the RAM, so in the 180-FU decoder, each FU processes one column from the even half and one column from the odd half. Thus, it may seem like two sets of *row*, *shift* and *ishift* coefficients are required to be stored in the ROM, one set for the RAM access of each half. However, Gomes et al. [18] suggest that the ROM size does not have to be increased because all the *row* and *shift* coefficients used in the 180-FU decoder can be calculated from the ROM coefficients of the 360-FU decoder. Nevertheless, the formula presented by Gomes et al. [18], does not clearly explain the necessary calculations. Furthermore, the given formula is not completely applicable to the decoder described in the previous sections because the decoder

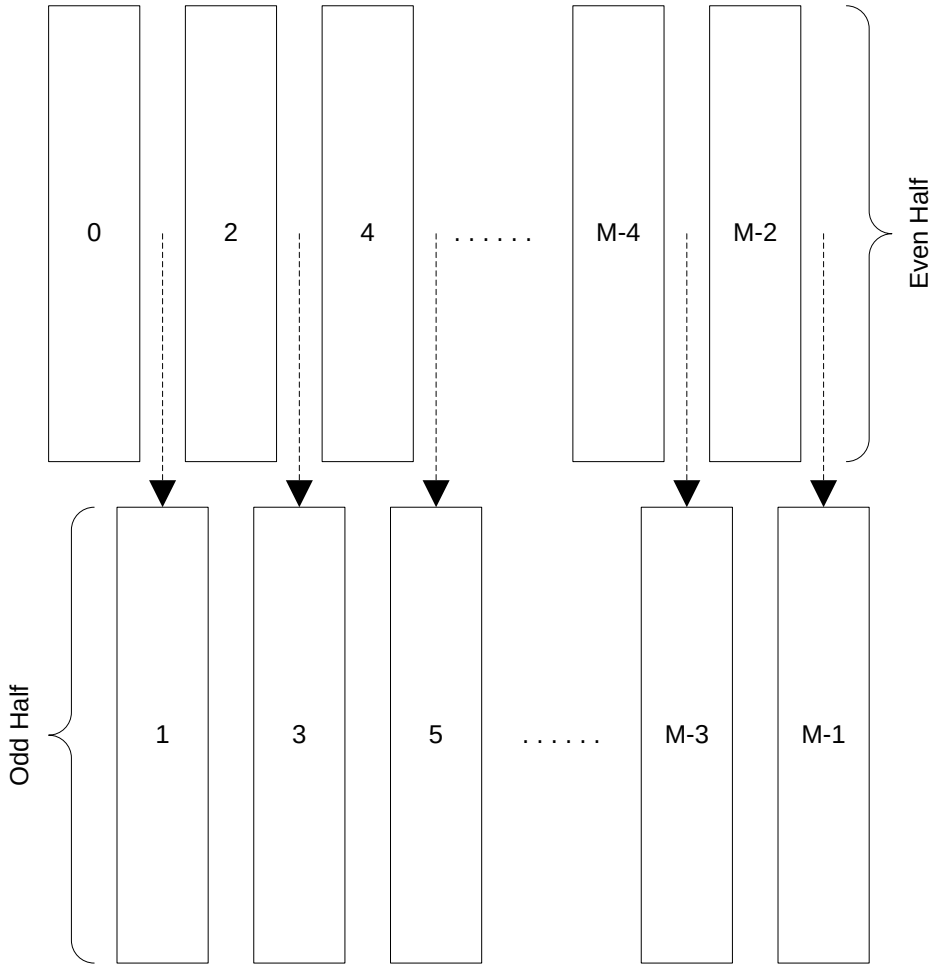

Figure 3.13: Diagram of splitting the RAM for 180 FU implementation.

by Gomes et al. [18] accesses the RAM sequentially during the Bit Node Update step, as opposed to accessing the RAM sequentially during the Check Node Update step. Thus, Algorithm 4 and Algorithm 5 have been devised to compute the *shift* and *ishift* coefficients for the 180-FU decoder using the same *shift* and *ishift* coefficients stored in the ROM. The new $shift_{180}$ and $ishift_{180}$ coefficients, which the coefficients used by the 180-FU decoder, are not stored in the ROM, instead they are computed in real-time from *shift* and *ishift* coefficients that are stored in the ROM, respectively. In Algorithm 4 and Algorithm 5, the First Pass refers to the FUs accessing half of the rows in the RAM and Second Pass refers to the FUs accessing the other half of the rows in the RAM.

Since the calculations to obtain the new $shift_{180}$ and $ishift_{180}$ coefficients only require division by 2 or ± 1 followed by a division by 2, these operations are implemented on the

Algorithm 4 $shift_{180}$ Coefficients for the 180-FU Decoder

Used during the Initialization and Bit Node Update steps:

if First Pass **then**

  if *shift* is even **then**

    $even\_odd = even$ // Read from and write to even half

    $shift_{180} = \frac{shift}{2}$

  else

    $even\_odd = even$ // Read from and write to odd half

    $shift_{180} = \frac{shift-1}{2}$

  end if

else // Second Pass

  if *shift* is even **then**

    $even\_odd = odd$ // Read from and write to odd half

    $shift_{180} = \frac{shift}{2}$

  else

    $even\_odd = even$ // Read from and write to even half

    $shift_{180} = \frac{shift+1}{2}$

  end if

end if

FPGA as a truncation of the least significant bit (LSB) or a sum of the truncated value and the LSB. In other words, if the *shift* coefficient is even, then $\frac{shift}{2}$ is implemented as the truncation of the LSB of *shift*. If the *shift* coefficient is odd, then $\frac{shift-1}{2}$ is also implemented as the truncation of the LSB of *shift* and $\frac{shift+1}{2}$ is implemented as the *shift* coefficient with the LSB truncated added to the LSB value, which is always 1.

The *row* coefficients for the 180-FU decoder do not need to be modified from the ones used for the 360-FU decoder because in the 180-FU decoder, the structure of the RAM is divided into an even and an odd half, so FUs access the same rows as they do in the 360-FU decoder. However, the controller needs to select which half of the RAM to be accessed depending on the *shift* and *ishift* coefficients and which Pass is occurring. In Algorithm 4 and Algorithm 5,

Algorithm 5 $ishift_{180}$ Coefficients for the 180-FU Decoder

Used during the Check Node Update step:

if First Pass **then**

$even\_odd = even$ // Read from and write to even half

if *ishift* is even **then**

$ishift_{180} = \frac{ishift}{2}$

else

$ishift_{180} = \frac{ishift-1}{2}$

end if

else // Second Pass

$even\_odd = odd$ // Read from and write to odd half

if *ishift* is even **then**

$ishift_{180} = \frac{ishift}{2}$

else

$ishift_{180} = \frac{ishift+1}{2}$

end if

end if

the *even_odd* parameter indicates whether the controller reads the rows from the even or odd half of the RAM for the FUs. Furthermore, in the 180-FU decoder, the architecture of the Shuffle Network, the BS, and the OC are also modified. The Shuffle Network and BS are modified to cyclically right-shift 0 to 179 positions and the OC is modified to only have a 180-bit input and counts the number of non-zero elements in the 180-bit vector.

From the architectures of the 360-FU and the 180-FU decoders of the DVB-S2 LDPC decoder, one can notice that the two have almost identical structures. The only difference is the RAM structure, which does not change in the number of cells but only the organization of the columns. The top RAM changes from $pq \times 360$ to $2pq \times 180$ and the bottom RAM changes from $2p \times 360$ to $4p \times 180$. Thus, the same number of BRAMs can be used for both the 360-FU and 180-FU decoders. The same can be said about the BRAMs in the LLR Buffer, Decoded Message Buffer and the PCM. Thus, a hybrid 360/180-FU LDPC decoder

is implemented based on those observation.

The difference between the 360-FU decoder and the hybrid decoder is in the controller. When the hybrid implementation is in 360-FU mode, all FUs are processing values from the RAM and the control flow of the decoder is as described in Section 3.1. In the 180-FU mode, the output of half of the FUs are discarded and the RAM inputs and outputs are controlled by multiplexers to select whether the even of odd half of the RAM are accessed for the FUs to process. Even though the hybrid decoder has a similar hardware resource utilization compared to the 360-FU decoder, depending on the application, the decoder might be required to reduce the throughput, for example, if the device connected to the downstream side of the decoder cannot handle a high throughput. Thus, the hybrid decoder has the ability to reduce the throughput in real-time without the need for a different decoder.

Chapter 4

Results and Discussion

4.1 Synthesis Results

The DVB-S2 LDPC decoder is synthesized using the Xilinx ISE 10.1 software package for use in the Xilinx Virtex-II Pro XC2VP100 FPGA, which is the same FPGA used by Gomes et al. [18]. The synthesis results for the 360-FU, 180-FU and hybrid decoders along with the synthesis results of the 180-FU decoder by Gomes et al. [18] are presented in Table 4.1. For simplicity, in the remainder of this chapter and the next, the decoders presented in Chapter 3 are referred to as the 360-FU decoder, 180-FU decoder and hybrid decoder, and the 180-FU decoder by Gomes et al. [18] is referred to as the Gomes decoder.

In Table 4.1, the resource utilizations of the slices, slices FFs, slice LUTs and BRAM are given as percentages of the maximum number of available hardware resources as shown in Section 2.1. Furthermore, notice that for the hybrid and 360-FU decoders, using the Xilinx

Table 4.1: Synthesis results and comparison

FPGA	XC2VP100				XC6VLX240T		
# of FUs	hybrid	360	180	180 [18]	hybrid	360	180
Slices (%)	154*	140*	79	88			
Slice FFs (%)	58	58	30	23	17	17	9
Slice LUTs (%)	150*	137*	76	80	67	60	34
BRAM (%)	43	43	43	50	31	31	31
f_{max} (MHz)	76.9	87.1	83.4	73.2	190.3	214.5	209.6
Estimated Power Consumption (W)					2.202	2.209	2.187

XC2VP100 FPGA, the slice and slice LUTs utilizations are above 100%. These results indicate that these two decoders utilizes more hardware resources than there are available in the FPGA, which means they cannot be implemented on the target FPGA. Thus, another set of synthesis results are shown, where the three versions of the LDPC decoder are synthesized for use on a XIlinx Virtex-6 XC6VLX240T FPGA using the Xilinx ISE Design Suite 11.5 software package. As shown in Table 4.1, none of the decoders exceed the amount of hardware resources available, which means that all three decoders can be implemented on the target FPGA. Furthermore, the synthesis results from the Xilinx ISE Design Suite 11.5 does not provide the slices utilization information. The f_{max} parameter is the estimated maximum frequency of the clock input that can be used with the decoder to ensure that the contents of the registers of the design are valid. The estimated power consumption of the decoders on the Virtex-6 FPGA using default settings of the Xilinx Power Analyzer software is also shown in Table 4.1. However, power optimization is not one of the design goals of the decoders, so these results are not discussed in detail. Nevertheless, the estimated power consumption information is included for reference.

Since the details of the Gomes decoder are not completely known, the explanation for the differences in the synthesis results cannot be stated with complete certainty. However, knowing the general architecture of its components, the comparison analysis in this section is performed to the best effort given the publications by Gomes et al. [18, 34].

According to the synthesis results on the XC2VP100 FPGA in Table 4.1, the 180-FU decoder outperforms the Gomes decoder as the 180-FU decoder uses fewer hardware resources, with the only exception in the slice FFs utilization, and has a higher maximum clock frequency. However, the Gomes decoder only includes the synthesis results of the core units (i.e., the FUs, the Shuffle Network, the RAM, the ROM and the Controller), yet the synthesis results of the 180-FU decoder includes the LLR Buffer, Decoded Message Buffer and the PCM. If the resource utilizations of these units are removed from the synthesis results of the decoder, the slice FFs utilization would be reduced. Furthermore, all three units not included in the synthesis results of the Gomes decoder are RAM based units, which are implemented using BRAM. If the BRAM usage of these three units are also removed from the synthesis results of the 180-FU decoder, its BRAM utilization would only be 26%, which

is almost a 50% reduction in the BRAM utilization from the Gomes decoder.

This reduction is due to the ROM coefficients that have been pre-computed and stored in the ROM. Gomes et al. [18] indicate that in order to obtain the *row* and *shift* coefficients, every non-zero element in the **A** submatrix of the parity-check matrix **H** is mapped in the ROM. Subsequently, the *row* and *shift* coefficients are searched from the ROM during the Check Node Update step (as mentioned previously in Section 3.6, the Gomes decoder accesses the RAM sequentially in the Bit Node Update step, and the access of the RAM is indexed during the Check Node Update step). Another possible explanation for the reduced BRAM utilization is because the RAM OUT outputs of the FUs are only 5 bits wide, as they are compressed by the COMP block, as discussed in Section 3.3.1.

Additionally, the 4% difference in the LUT utilization may be attributed to the optimization of the FUs. As discussed in Section 3.3.1, the Gomes decoder uses the boxplus and boxminus blocks in the FU design. These two units are made up of adders, comparators, multiplexers, LUTs, etc., as shown in Figure 3.7 and Figure 3.8. Compared to the ψ block, COMP block and adder/subtractors used in their places, the boxplus and boxminus blocks are more complex and utilize more hardware resources. This difference is also likely to be the reason for the 14% increase in the maximum clock frequency from the Gomes decoder.

4.2 Throughput Comparison

Gomes et al. [18] present that the minimum throughput of the Gomes decoder is given by:

$$throughput \geq \frac{frame\_length \times f_{op}}{(2 \times W + w_j - 3) \times max\_iter \times L} \tag{4.1}$$

where *frame_length* is equivalent to N, f_{op} is the maximum operating frequency which is set to f_{max} in the calculation, W is equivalent to pq, where p and q are given in Table 3.2, w_j is the bit node degree of bit node groups in the **A** submatrix that has a bit node degree not equal to 3, *max_iter* is the maximum number of iterations, and L is the reduction factor of FUs which is 2 in the case of the 180-FU implementation.

For the decoder of this project, the following equations demonstrate the derivation process

of the minimum throughput equations:

$$\begin{aligned}
throughput &\geq \frac{frame_length}{max_num_clock_cycle} \times f_{op} \\
&\geq \frac{frame_length \times f_{op}}{(max_clock_cycle_per_iter) \times max_iter} \\
&\geq \frac{frame_length \times f_{op}}{(CNUP + BNUP + FU_delay + PCM_delay) \times L \times max_iter} \\
&\geq \frac{frame_length \times f_{op}}{((pq + 2p) + (pq + 2p) + (w_j + 3) + 3) \times L \times max_iter} \\
throughput &\geq \frac{frame_length \times f_{op}}{(2 \times (pq + 2p) + w_j + 6) \times max_iter \times L} \qquad (4.2)
\end{aligned}$$

where the variables have the same meaning as described before and $L = 1$ for the 360-FU implementation and $L = 2$ for the 180-FU implementation. Note that the difference between equations (4.1) and (4.2) is that W, which is equivalent to pq, is replaced by $pq + 2p$, and $w_j - 3$ is replaced by $w_j + 6$, which means that the Gomes decoder requires fewer clock cycles per iteration. The derivation of equation (4.2) is explained in the next paragraph.

The throughput is given by the frame length, *frame_length* or N, divided by the maximum number of clock cycles spent to decode one frame multiplied by the maximum operating frequency. The maximum number of decoding clock cycles is given by the maximum number of iterations multiplied by the number of clock cycles per iteration. The number of clock cycles per iteration is equal to the number of clock cycles to process Check Node Update plus the number of clock cycles to process the Bit Node Update plus the delay of the FUs plus the delay of the PCM, all multiplied by the reduction factor of FUs. The number of clock cycles to process the Check Node Update and Bit Node Update steps are both equal to the number of rows in the top and the bottom RAMs, which is $pq + 2p$, the delay of the FUs is $w_j + 3$ and the delay of the PCM is 3 clock cycles as mentioned in Section 3.4. Notice that the number of clock cycles for inputting the LLR values and outputting the decoded message is not included in the calculation of the throughput. The throughput calculation of the Gomes decoder also leaves out these delays. As mentioned in Section 3.5, the LLR Buffer is designed for serial input of the LLR values, yet the architecture can be changed to accept more than one LLR value simultaneously. Thus, the throughput calculation presented here only includes the processing times of the core units of the decoder relative to the frame length.

Using equations (4.1) and (4.2), Table 4.2 is generated to show the minimum throughput comparison between the two decoder designs. The maximum number of iterations is chosen to be 15 as selected by Gomes et al. [18].

Even though the maximum number of clock cycles per iteration is higher for the decoder presented in Chapter 3 compared to the Gomes decoder, the maximum frequency, f_{max}, of the 180-FU decoder is higher than the f_{max} of the Gomes decoder, according to Table 4.1. Thus, the two variables offset each other in the minimum throughput equations (4.1) and (4.2). As a result, the minimum throughput of the Gomes decoder is higher for some code rates and lower in others compared to the 180-FU decoder, as shown in Table 4.2. In particular, higher code rates perform better in the 180-FU decoder and lower code rates perform better in the Gomes decoder because the higher the code rate, the smaller the value of $2p$, which makes the $pq + 2p$ part of equation (4.2) closer to W of equation (4.1). Consequently, the smaller maximum number of clock cycles per iteration and a higher maximum clock frequency results in a higher throughput in higher code rates. Furthermore, the Gomes decoder does not include the Hard Decision Making step, which requires 3 clock cycles per iteration in the 180-FU decoder. Thus, depending on the design of the PCM used by Gomes et al., which is not shown in the publications, the throughput of the Gomes decoder would be reduced.

The minimum throughput of the 360-FU and the hybrid decoders are not shown for the Virtex-II Pro FPGA because the two decoders cannot be implemented on the FPGA as discussed in Section 4.1. The minimum throughput values are also shown for the Virtex-6 FPGA in Table 4.2 for reference and they demonstrate the throughput improvement achieved by using a more modern FPGA. Even though the architecture of the decoders is optimized for the Virtex-II Pro FPGA, it is not limited to be used only by that family of FPGAs. By implementing the decoders designed on more modern FPGAs as technology advances, the decoders are further improved by the technology, without the need to redesign the components.

Table 4.2: Minimum Throughput of the Decoders

Block Length	Code Rate	Throughput (Mbps)					
		XC2VP100		XC6VLX240T			
		180-FU Decoder	Gomes Decoder	hybrid decoder		360-FU Decoder	180-FU Decoder
				360 mode	180 mode		
64800	1/4	164.1	288.0	748.7	374.4	843.9	412.3
	1/3	147.9	216.9	675.0	337.5	760.8	371.7
	2/5	137.1	181.1	625.6	312.8	705.2	344.5
	1/2	141.4	174.7	645.3	322.6	727.4	355.4
	3/5	112.5	121.2	513.2	256.6	578.4	282.6
	2/3	147.8	163.0	674.4	337.2	760.2	371.4
	3/4	141.0	145.2	643.3	321.6	725.1	354.3
	4/5	137.2	136.3	626.1	313.1	705.7	344.8
	5/6	134.5	130.7	614.0	307.0	692.0	338.1
	8/9	165.3	158.0	754.2	377.1	850.1	415.4
	9/10	165.3	156.7	754.2	377.1	850.1	415.4
16200	1/5	156.4	292.8	713.6	356.8	804.4	393.0
	1/3	141.6	209.1	646.3	323.2	728.5	355.9
	2/5	131.7	175.7	601.0	300.5	677.4	331.0
	4/9	158.6	255.9	723.7	361.8	815.7	398.5
	3/5	108.8	118.7	496.4	248.2	559.6	273.4
	2/3	141.2	158.1	644.3	322.1	726.2	354.8
	11/15	159.7	175.7	728.8	364.4	821.5	401.4
	7/9	178.0	188.2	812.4	406.2	915.7	447.4
	37/49	153.7	156.9	701.5	350.7	790.7	386.3
	8/9	160.9	157.5	734.0	367.0	827.4	404.2

4.3 Simulation Results

The 360-FU decoder presented in Chapter 3 is verified using a MATLAB testbench script. The testbench begins by generating a random sequence of bits. The length of the sequence depends on the number of frames needed and the code rate selected. Every frame of the sequence is encoded using the MATLAB built-in LDPC encoder to generate frames of N bits long and subsequently modulated using the BPSK modulation scheme. The BCH outer encoding specified in the DVB-S2 standard [6] is not used because only the performance of the LDPC decoder is tested. The DVB-S2 standard [6] also uses quadrature phase-shift keying (QPSK), 8 phase-shift keying (8PSK), 16 amplitude and phase shift keying (16APSK) and 32 amplitude and phase-shift keying (32APSK) modulation schemes, but the simulation testbench uses BPSK modulation scheme to modulate the encoded sequence for simplicity. Subsequently, the modulated signal passes through a transmission channel, which is simulated by adding AWGN. The receiving side of the testbench demodulates the transmitted signal using the MATLAB built-in demodulator, which produces LLR values. These LLR values are divided into frames of N values and each frame is inputted into the 360-FU decoder. Since the control flow of the three decoders presented in Chapter 3 are identical, with the only difference being the decoding delay from using 360 FUs or 180 FUs, the verification of only the 360-FU decoder is necessary. Furthermore, for reducing the need to interface MATLAB with the FPGA, the 360-FU decoder is also implemented on MATLAB using finite precision for its calculations and the same control flow as described in Section 3.1. Finally, the message part of the decoded codeword for each frame from the 360-FU decoder is compared to the frames of the original random sequence generated. If the two sequences are identical, then the decoding is correct, otherwise, decoding error has occurred for that particular frame.

For verification, the packet error rate (PER) versus the signal-to-noise ratio (SNR) of the channel is graphed. The PER is defined by the number of frames that are decoded erroneously divided by the number of frames processed by the decoder. The SNR is the characteristic of the AWGN channel in units of decibels (dB), which is defined by the power of the signal transmitted divided by the power of the noise in the channel. For normal

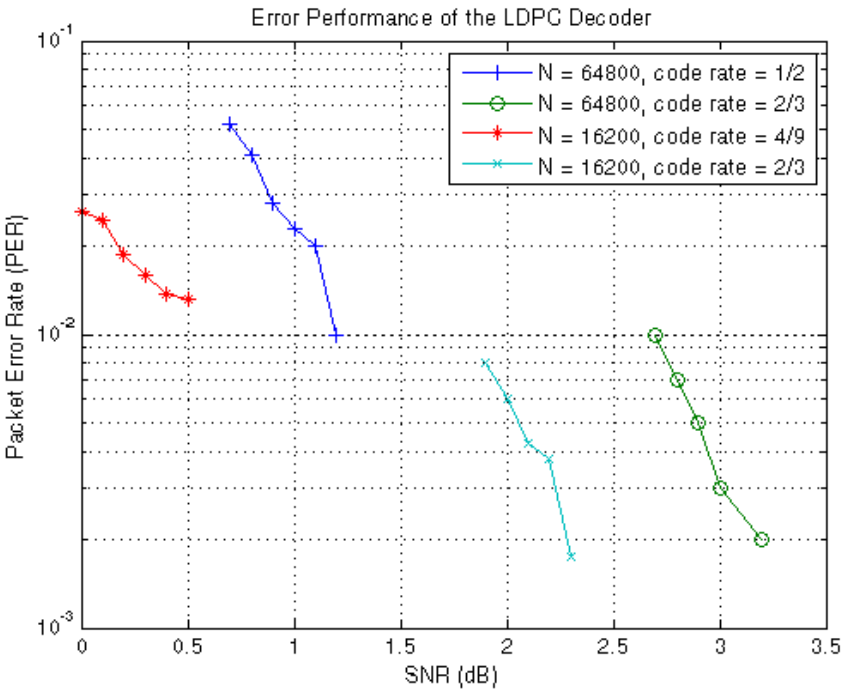

Figure 4.1: PER vs. SNR of the LDPC decoder.

frames, 1000 frames are used and for short frames 4000 frames are used, such that in both cases, the same number of bits are decoded. The graphs in Figure 4.1 are generated by using normal frame code rates 1/2 and 2/3 and short frame code rate 4/9 and 2/3.

Compared to the error performance graphs shown in Annex A of the user guidelines of the DVB-S2 standard [42], the error performance of the decoders does not seem to perform well. However, the error performance graphs in the standard include the BCH outer encoding and decoding and may be performed using decoders with higher precision in software. Nevertheless, Figure 4.1 cannot conclude that the error performance of the three decoders presented in this thesis is compliant with the DVB-S2 standard, but the graphs do indicate that the LDPC decoders work with the codes given in the DVB-S2 standard. Moreover, the error performance of the decoders can be improved with an improved ψ function approximation and by testing the usage of different number of bits for the LLR and RAM values in the decoding process. In addition, Figure 4.1 shows that the special case code rates can be handled using the method described in Section 3.2.4, as the graph shows that the short frame 4/9 code rate can be decoded using the 360-FU decoder.

Chapter 5

Conclusion

In this thesis, detailed FPGA designs of a LDPC decoder for the DVB-S2 standard are presented. The decoders presented are modifications and improvements of the DVB-S2 LDPC decoders published in current literature. The resultant decoder is able to decode all code rates in both normal frames and short frames of the DVB-S2 standard.

The memory mapping scheme proposed by Eroz et al. [8] does not include an algorithm that maps each of the non-zero element in the **A** submatrix to the RAM, so a novel algorithm is devised and given in Algorithm 1 to perform such mapping. Furthermore, another novel algorithm, Algorithm 2, is devised to eliminate the need to locate all the non-zero elements in the **A** submatrix in order to generate the coefficients that need to be stored in the ROM. Algorithm 2 uses the values given in the standard, which are also shown in Appendix B, and directly generates the coefficients stored in the ROM from them. By using this memory mapping scheme, the decoders presented in this thesis are implemented using 360 FUs or 180 FUs.

The details of the FUs are shown in Section 3.3. Each FU is implemented using the serial architecture as presented by Gomes et al. [34], but the architecture of the components used are less complex, which reduces the hardware resource utilization on the FPGA. The FUs presented in this thesis use the ψ function along with adder/subtractor modules instead of the boxplus and boxminus modules used by the Gomes decoder, as described in Section 3.3.1. The ψ function is approximated by combining the variable precision quantization scheme presented by Zhang et al. [20] and the LUT reduction technique presented by Oh and Parhi [21] to generate a LUT that only has $2^4 \times 6$-bit entries. Furthermore, the LUT reduction also results in fewer BRAM memory resources used by the decoder allowing for the decoder to be implemented in FPGA devices with fewer BRAMs available. The details

of the FU implementation, as presented in Section 3.3, have been published [47].

The architecture of the PCM is novel, as DVB-S2 LDPC decoder designs reviewed in Section 1.1 do not present any module that verifies the parity-check equations to perform the Hard Decision Making step in the SPA. The PCM architecture is based on the encoder architecture by Gomes et al. [35]. Nevertheless, a novel algorithm, Algorithm 3, is devised to verify the parity-check equations efficiently and using only a small amount of hardware resources.

Section 3.2.4 presents five short frame code rates that have a characteristic that differs from all the other codes in the DVB-S2 standard. The codes of these code rates do not have the same row weight on every row of the **A** submatrices. Some of the publications reviewed in Section 1.1 do not handle short frames. The ones that do handle short frames do not address this characteristic of these code rates and assumes that all the row weights of the **A** matrix are constant. In this thesis, a novel workaround for these special code rates is presented, which modifies Algorithm 2 to generate the coefficients stored in the ROM. These code rates also have a different RAM access order compared to the other code rates, which are also presented in Section 3.2.4.

In order to further reduce hardware resource utilization, the 360-FU decoder reduced to only use 180 FUs. The trade-off of the change is a reduced throughput. On the other hand, the hybrid decoder is implemented to increase the flexibility of the decoder, since the throughput of the decoder can be changed in real-time. The synthesis results in Section 4.1 show that only the 180-FU decoder can be implemented using the Virtex-II Pro XC2VP100 FPGA because the other two decoders requires more than the available hardware resources on the FPGA. Thus, the decoders are re-synthesized for the Virtex-6 XC6VLX240T FPGA, which can implement all three decoders. Comparing the 180-FU decoder to the Gomes decoder, the 180-FU decoder uses fewer slices, slice LUTs and BRAMs, and has a higher maximum frequency. In addition, the 180-FU decoder implements the PCM, LLR Buffer and Decoded Message Buffer that are not implemented in the Gomes decoder.

The throughput results in Table 4.2 show that the minimum throughput of the 180-FU decoder is better than minimum throughput of the Gomes decoder in higher code rates. Furthermore, the throughput of the three decoders implemented on the Virtex-6 FPGA are

even more superior than the decoders implemented on the Virtex-II Pro FPGA because of the higher maximum clock frequencies. The error performance graphs in Figure 4.1 indicate that the three decoders might not perform up to the error performance requirements desired by the DVB-S2 standard. However, the graphs do demonstrate that the decoders successfully handle the special code rates as previously mentioned and the decoder works properly.

For future work, different ψ function approximations can be tested to find the optimal trade-off between the hardware complexity of implementing the ψ function and performance. Furthermore, the different decoding algorithms presented by Papaharalabos et al. [23] can be implemented on the FUs to improve the error performance of the LDPC decoder implemented on the FPGA in hopes of satisfying the error performance required by the DVB-S2 standard. The comparison of these decoding algorithms can also be revisited from a FPGA design point of view by evaluating the FPGA hardware resource utilization and throughput of the decoder. The error performance can also be improved by using more bits to represent the LLR and RAM values.

Recall that the design of some of the components of the decoder are customized for Virtex-II Pro FPGAs, and not the Virtex-6 FPGA. Thus, in order to further reduce hardware resource utilization when using the Virtex-6 FPGA, the architecture of these components can be re-designed and be customized for the Virtex-6 FPGA CLB and slice architecture.

Another potential platform for the DVB-S2 LDPC decoder is to implement the decoder using a graphics processing unit (GPU). Modern GPUs are known for their high number of processors dedicated for arithmetic calculations [48]. Since the decoding of the DVB-S2 LDPC codes are also highly parallel and requires a large number of FUs, the GPU platform is very suitable for the implementation of the DVB-S2 LDPC decoders. Since the GPU is programmed in a software environment, it has a similar flexibility that FPGAs can provide because the GPU programs can be easily re-compiled for use in a different GPU. In addition, GPUs have built-in floating point arithmetic units in each processor, so they can perform the Check Node Update and Bit Node Update calculations with higher precision than the FPGA designs, which would improve the error performance of the decoder.

References

[1] C. E. Shannon, "A Mathematical Theory of Communication," *Bell system Technical Journal*, vol. 27, pp. 379–423 and 623–656, July and October 1948.

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

[3] D. J. C. MacKay and R. M. Neal, "Near Shannon Limit Performance of Low Density Parity Check Codes," *Electronics Letters*, vol. 32, no. 18, pp. 1645–1646, August 1996.

[4] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correction coding and decoding: Turbo-codes," in *IEEE International Conference on Communications, 1993*, vol. 2, May 1993, pp. 1064–1070.

[5] DVB. [Online]. Available: http://www.dvb.org

[6] ETSI, "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), EN 302 307 V 1.2.1." August 2009.

[7] A. Blanksby and C. 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.

[8] M. Eroz, F.-W. Sun, , and L.-N. Lee, "An Innovative Low-Density Parity-Check Code Design With near-Shannon-limit Performance and Simple Implementation," *IEEE Transactions on Communications*, vol. 54, no. 1, pp. 13–17, January 2006.

[9] F. Kienle, T. Brack, and N. Wehn, "A synthesizable IP Core for DVB-S2 LDPC Code Decoding," in *Proceedings of the Design, Automation and Test in Europe, 2005*, vol. 3, March 2005, pp. 100–105.

[10] P. Urard, E. Yeo, L. Paumier, P. Georgelin, T. Michel, V. Lebards, E. Lantreibecq, and G. Gupta, "A 135Mb/s DVB-S2 compliant codec based on 64800b LDPC and BCH codes," in *IEEE International Solid-State Circuits Conference, 2005*, vol. 1, February 2005, pp. 446–609.

[11] J. Dielissen, A. Hekstra, and V. Berg, "Low cost LDPC decoder for DVB-S2," in *Design, Automation and Test in Europe, 2006*, vol. 2, March 2006, pp. 1–6.

[12] A. Segard, F. Verdier, D. Declercq, and P. Urard, "A DVB-S2 compliant LDPC decoder integrating the Horizontal Shuffle Scheduling," in *International Symposium on Intelligent Signal Processing and Communications*, December 2006, pp. 1013–1016.

[13] G. Masera, F. Quaglio, and F. Vacca, "Implementation of a Flexible LDPC Decoder," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 6, pp. 542–546, June 2007.

[14] T. Brack, M. Alles, T. Lehnigk-Emden, F. Kienle, N. Wehn, N. L'Insalata, F. Rossi, M. Rovini, and L. Fanucci, "Low Complexity LDPC Code Decoders for Next Generation Standards," in *Design, Automation and Test in Europe Conference and Exhibition, 2007*, April 2007, pp. 1–6.

[15] B. Zhang, H. Liu, X. Chen, D. Liu, and X. Yi, "Low Complexity DVB-S2 LDPC Decoder," in *IEEE 69th Vehicular Technology Conference, 2009*, April 2009, pp. 1–5.

[16] Y. Ying, D. Bo, S. Huang, B. Xiang, Y. Chen, and X. Zeng, "A cost efficient LDPC decoder for DVB-S2," in *IEEE 8th International Conference on ASIC, 2009*, October 2009, pp. 1007–1010.

[17] M. K. Yadav and K. K. Parhi, "Design and implementation of LDPC Codes for DVB-S2," in *Conference Record of the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers, 2005*, November 2005, pp. 723–728.

[18] M. Gomes, G. Falcao, V. Silva, V. Ferreira, A. Sengo, and M. Falcao, "Flexible Parallel Architecture for DVB-S2 LDPC Decoders," in *IEEE Global Telecommunications Conference, 2007*, November 2007, pp. 3265–3269.

[19] C. Beuschel and H.-J. Pfleiderer, "FPGA implementation of a flexible decoder for long LDPC codes," in *International Conference on Field Programmable Logic and Applications, 2008*, September 2008, pp. 185–190.

[20] T. Zhang, Z. Wang, and K. K. Parhi, "On Finite Precision Implementation of Low Density Parity Check Codes Decoder," in *IEEE International Symposium on Circuits and Systems, 2001*, vol. 4, May 2001, pp. 202–205.

[21] D. Oh and K. K. Parhi, "Low Complexity Implementations of Sum-Product Algorithm for Decoding Low-Density Parity-Check Codes," in *IEEE Workshop on Signal Processing Systems Design and Implementation, 2006*, October 2006, pp. 262–267.

[22] G. Masera, F. Quaglio, and F. Vacca, "Finite precision implementation of LDPC decoders," *IEE Proceedings – Communications*, vol. 152, no. 6, pp. 1098–1102, December 2005.

[23] S. Papahralabos, M. Papaleo, P. T. Mathiopoulos, M. Neri, A. Vanelli-Coralli, and G. Corazza, "DVB-S2 LDPC Decoding Using Robust Check Node Update Approximations," *IEEE Transactions on Broadcasting*, vol. 54, no. 1, pp. 120–126, March 2008.

[24] ViaSat, "SkyPHY^{TM} DVB-S2 Receiver ASIC(ECC3100)." [Online]. Available: http://www.viasat.com/files/assets/web/datasheets/ecc3100_skyphy_receiver_asic_datasheet_104.pdf

[25] ——, "DVB-S2 Mini Receiver." [Online]. Available: http://www.viasat.com/files/assets/web/datasheets/DVB_S2_mini_receiver_military_110_lores.pdf

[26] ——, "MIRD-S2: DVB-S2 Receiver Decoder." [Online]. Available: http://www.viasat.com/files/assets/web/datasheets/MIRD-S2_104.pdf

[27] MATLAB, "dvbs2ldpc." [Online]. Available: http://www.mathworks.com/help/toolbox/comm/ref/dvbs2ldpc.html

[28] ——, "fec.ldpcdec." [Online]. Available: http://www.mathworks.com/help/toolbox/comm/ref/fec.ldpcdec.html

[29] Xilinx. [Online]. Available: http://www.xilinx.org

[30] Altera. [Online]. Available: http://www.altera.org

[31] N. Systems, "DVB-S2 LDPC Decoder." [Online]. Available: http://www.navtelsystems.com/pdf/DatasheetLDPCDVBS2.pdf

[32] SoftJin, "LDPC Decoder for DVB-S2." [Online]. Available: http://www.softjin.com/IP_Datasheet_PDF_version/LDPC_Decoder_datasheet.pdf

[33] R. Communications, "Product Brief: DVB-S2 LDPC Decoder." [Online]. Available: http://www.rad3comm.com/uploads/DVB-S2_LDPC.pdf

[34] M. Gomes, G. Falcao, J. Goncalves, V. Silva, M. Falcao, and P. Faia, "HDL Library of Processing Units for Generic and DVB-S2 LDPC Decoding," in *International Conference on Signal Processing and Multimedia Applications*, August 2006.

[35] M. Gomes, G. Falcao, A. Sengo, V. Ferreira, V. Silva, and M. Falcao, "High throughput encoder architecture for DVB-S2 LDPC-IRA codes," in *International Conference on Microelectronics, 2007. ICM 2007*, December 2007, pp. 271–274.

[36] Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet," November 2007. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf

[37] ——, "Virtex-6 FPGA Configurable Logic Block: User Guide," September 2009. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug364.pdf

[38] ——, "Virtex-6 Family Overview," January 2010. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[39] ——, "Virtex-6 FPGA Memory Resources: User Guide," August 2010. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug363.pdf

[40] S. Lin and D. J. C. Jr., *Error Control Coding*, 2nd ed. Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2004.

[41] H. Jin, A. Khandekar, and R. McEliece, "Irregular repeat-accumulate codes," in *2nd International Symposium on Turbo Codes and Related Topics*, September 2000.

[42] ETSI, "Digital Video Broadcasting (DVB): User guidelines for the second generation system for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2), TR 102 376 V 1.1.1." February 2005.

[43] R. M. Tanner, "Recursive Approach to Low Complexity Codes," *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, September 1981.

[44] M. Eroz, F.-W. Sun, and L.-N. Lee, "DVB-S2 Low Density Parity Check Codes with Near Shannon Limit Performance," *International Journal of Satellite Communications and Networking*, vol. 22, no. 3, pp. 269–279, June 2004.

[45] H. C. Neto and M. P. Vestias, "Architectural Tradeoffs in the Design of Barrel Shifters for Reconfigurable Computing," in *4th Southern Conference on Programmable Logic, 2008*, March 2008, pp. 31–36.

[46] M. Defossez and N. Sawyer, "Xilinx Application Note: Using Block SelectRAM Memories as Serializers or Deserializers, XAPP690 (v1.0)," October 2003. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp690.pdf

[47] K. C. C. Loi, S.-B. Ko, and D. Armstrong, "Optimizations and Improvements of the DVB-S2 LDPC Decoder Functional Unit," in *12th International Workshop on Multimedia Signal Processing and Transmission*, September 2009, pp. 114–118.

[48] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th ed. Burlington, MA, USA: Morgan Kaufmann, 2009.

Appendix A

The Encoding and Decoding of a Simple Linear Systematic Block Code

In this appendix, the encoding and decoding of a linear systematic block code is shown using a simple example. The code used for the remainder of this appendix has a codeword length of $N = 7$, and the number of information bits, $K = 4$. The code can also be expressed as a (7,4) linear systematic block code. The examples shown here are based on chapter 3 of Lin and Costello's book [40].

Consider a (7,4) linear systematic block code that has the following generator matrix:

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \mathbf{g}_2 \\ \mathbf{g}_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The generator matrix has the form $\mathbf{G} = [\mathbf{P}\mathbf{I}_K]$ as discussed in Section 2.2 so the code is a systematic code. Using the generator matrix, the 7-bit codewords can be generated from any 4-bit message by using equation (2.2) from Section 2.2. For example, consider the message $\mathbf{u} = (1\ 1\ 0\ 1)$ and the encoding is performed as follows:

$$\begin{aligned} \mathbf{v} &= \mathbf{u} \cdot \mathbf{G} \\ &= 1 \cdot \mathbf{g}_0 + 1 \cdot \mathbf{g}_1 + 0 \cdot \mathbf{g}_2 + 1 \cdot \mathbf{g}_3 \\ &= (1101000) + (0110100) + (1010001) \\ &= (0001101) \end{aligned}$$

Using the same technique, the other fifteen codewords can also be generated, and the result is shown in Table A.1. It can be seen that the code is systematic because the rightmost four bits of the codeword is identical to the original message, so they make up the message part of the codeword. The remaining three leftmost bits are the redundant checking part.

Encoding can be simplified from the systematic property of the code by using the parity-

Table A.1: Example of a (7,4) Linear Systematic Block Code

Messages	Codewords
(0 0 0 0)	(0 0 0 0 0 0 0)
(1 0 0 0)	(1 1 0 1 0 0 0)
(0 1 0 0)	(0 1 1 0 1 0 0)
(1 1 0 0)	(1 0 1 1 1 0 0)
(0 0 1 0)	(1 1 1 0 0 1 0)
(1 0 1 0)	(0 0 1 1 0 1 0)
(0 1 1 0)	(1 0 0 0 1 1 0)
(1 1 1 0)	(0 1 0 1 1 1 0)
(0 0 0 1)	(1 0 1 0 0 0 1)
(1 0 0 1)	(0 1 1 1 0 0 1)
(0 1 0 1)	(1 1 0 0 1 0 1)
(1 1 0 1)	(0 0 0 1 1 0 1)
(0 0 1 1)	(0 1 0 0 0 1 1)
(1 0 1 1)	(1 0 0 1 0 1 1)
(0 1 1 1)	(0 0 1 0 1 1 1)
(1 1 1 1)	(1 1 1 1 1 1 1)

check equations generated as follows:

$$\mathbf{v} = (u_0, u_1, u_2, u_3) \cdot \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} v_6 &= u_3 \\ v_5 &= u_2 \\ v_4 &= u_1 \\ v_3 &= u_0 \\ v_2 &= u_1 + u_2 + u_3 \\ v_1 &= u_0 + u_1 + u_2 \\ v_0 &= u_0 + u_2 + u_3 \end{aligned}$$

Using the parity-check equations, the encoder can be implemented as a set of shift registers for the message bits, three 3-input modulo-2 adders and a set of shift registers for the parity-check bits. An encoder block diagram for this (7,4) linear systematic block code is shown in Figure A.1.

The parity-check matrix, $\mathbf{H}$, of this code can be obtained as follows:

$$\mathbf{H} = [\mathbf{I}_{N-K}\mathbf{P}^T] = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

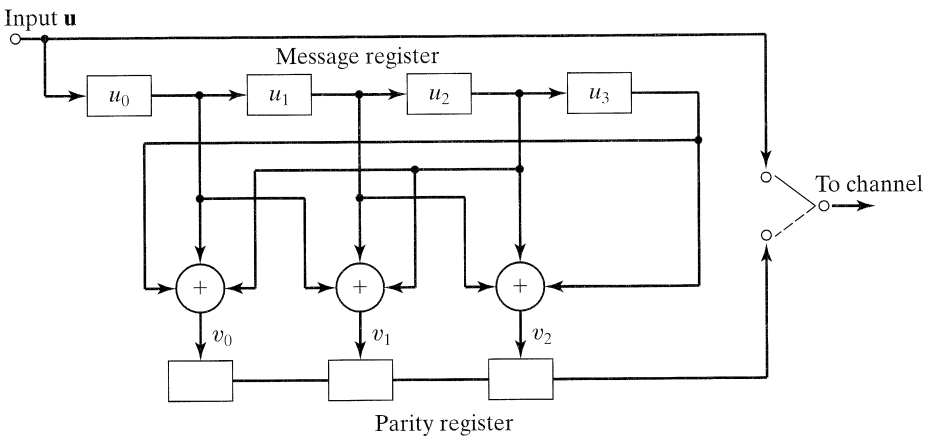

Figure A.1: An example encoder for the (7,4) linear systematic block code.

Let the received vector from the output of the channel be $\mathbf{r} = (r_0, r_1, r_2, r_3, r_4, r_5, r_6)$. Using the parity-check matrix, the equation for each bit in the syndrome, $\mathbf{s} = (s_0, s_1, s_2)$, can be computed as follows:

$$
\begin{aligned}
\mathbf{s} &= \mathbf{r} \cdot \mathbf{H}^T \\
&= (r_0, r_1, r_2, r_3, r_4, r_5, r_6) \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \\
s_0 &= r_0 + r_3 + r_5 + r_6 \\
s_1 &= r_1 + r_3 + r_4 + r_5 \\
s_2 &= r_2 + r_4 + r_5 + r_6
\end{aligned}
$$

These syndrome computations can be implemented using 4-input modulo-2 adders. In order to implement the decoder using syndrome decoding, the standard array needs to be

Table A.2: Decoding table for the (7,4) linear systematic block code

Syndrome	Coset leaders
(1 0 0)	(1 0 0 0 0 0 0)
(0 1 0)	(0 1 0 0 0 0 0)
(0 0 1)	(0 0 1 0 0 0 0)
(1 1 0)	(0 0 0 1 0 0 0)
(0 1 1)	(0 0 0 0 1 0 0)
(1 1 1)	(0 0 0 0 0 1 0)
(1 0 1)	(0 0 0 0 0 0 1)

built first. By using the steps in Section 2.2 the following standard array can be built:

0000000	1101000	0110100	1011100	1110010	0011010	1000110	0101110	1010001
1000000	0101000	1110100	0011100	0110010	1011010	0000110	1101110	0010001
0100000	1001000	0010100	1111100	1010010	0111010	1100110	0001110	1110001
0010000	1111000	0100100	1001100	1100010	0011010	1010110	0111110	1000001
0001000	1100000	0111100	1010100	1111010	0010010	1001110	0100110	1011001
0000100	1101100	0110000	1011000	1110110	0011110	1000010	0101010	1010101
0000010	1101010	0110110	1011110	1110000	0011000	1000100	0101100	1010011
0000001	1101001	0110101	1011101	1110011	0011011	1000111	0101111	1010000

0000000	0111001	1100101	0001101	0100011	1001011	0010111	1111111
1000000	1111001	0100101	1001101	1100011	0001011	1010111	0111111
0100000	0011001	1000101	0101101	0000011	1101011	0110111	1011111
0010000	0101001	1110101	0011101	0110011	1011011	0000111	1101111
0001000	0110001	1101101	0000101	0101011	1000011	0011111	1110111
0000100	0111101	1100001	0001001	0100111	1001111	0010011	1111011
0000010	0111011	1100111	0001111	0100001	1001001	0010101	1111101
0000001	0111000	1100100	0001100	0100010	1001010	0010110	1111110

From the standard array, the coset leaders can be found on the leftmost column. For each coset leader, using the equation $\mathbf{s} = \mathbf{e}_l \cdot \mathbf{H}^T$, the respective syndromes can be found for each coset leader. The resultant decoding table is shown in Table A.2.

Let the original message be $\mathbf{u} = (1\ 1\ 0\ 1)$. As shown above, the codeword generated by the encoder is $\mathbf{v} = (0\ 0\ 0\ 1\ 1\ 0\ 1)$. Assume the received vector from the output of the channel is $\mathbf{r} = (0\ 1\ 0\ 1\ 1\ 0\ 1)$. The decoding process begins with computing the syndrome, $\mathbf{s}$:

$$\mathbf{s} = (0101101) \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = (010)$$

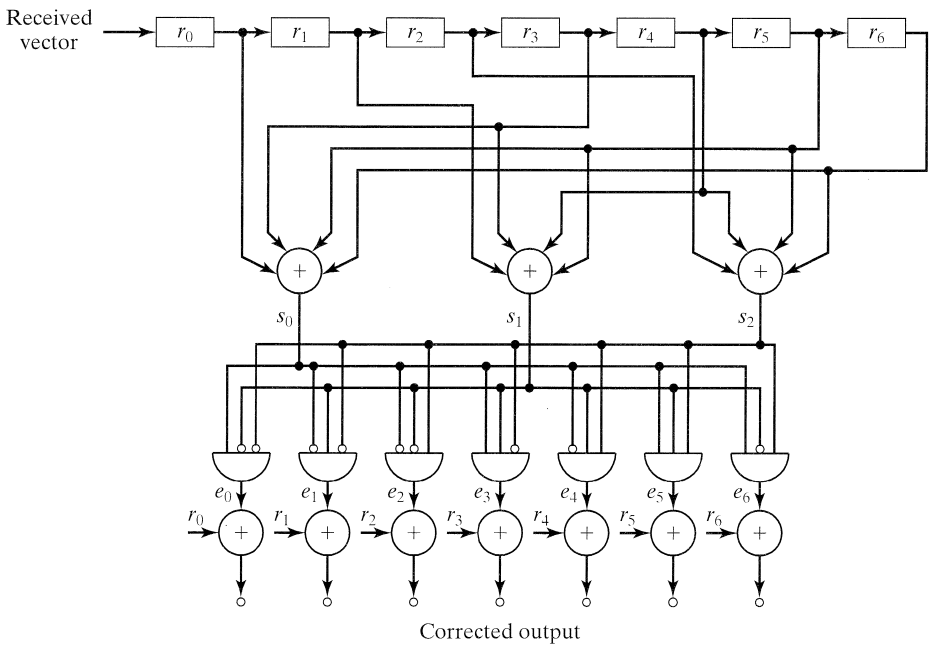

Figure A.2: An example decoder for the (7,4) linear systematic block code.

Subsequently, using Table A.2, find the syndrome and decode it into its respective coset leader, $\mathbf{e}_l = (0\ 1\ 0\ 0\ 0\ 0\ 0)$. Finally, add the coset leader to the received vector, $\mathbf{r}$, to obtain the decoded codeword:

$$\begin{aligned} \mathbf{v}^* &= \mathbf{e}_l + \mathbf{r} \\ &= (0100000) + (0101101) \\ &= (0001101) \end{aligned}$$

Since the code is systematic, the original message can be retrieved from the message part of the decoded codeword. Thus, the decoded message is $\mathbf{u}^* = (1\ 1\ 0\ 1)$, which is the original message that was encoded.

In terms of implementation, the decoder can be implemented using a set of shift registers for the received vector from the channel, a syndrome calculation circuit using modulo-2 adders, a combinational logic circuit that realizes the decoding table and a set of modulo-2 adders to combine the error vector with the received vector. An example of a decoder implemented for the (7,4) code discussed is shown in Figure A.2.

Appendix B

Values from Annex B and C of the DVB-S2 Standard

In this Appendix, the values from Annex B and C of the DVB-S2 standard [6] are reproduced. The values for the normal frames are shown first, followed by the values for short frames.

Table B.1: $N = 64800$, Code Rate = 1/4

23606 36098 1140 28859 18148 18510 6226 540 42014 20879 23802 47088
16419 24928 16609 17248 7693 24997 42587 16858 34921 21042 37024 20692
1874 40094 18704 14474 14004 11519 13106 28826 38669 22363 30255 31105
22254 40564 22645 22532 6134 9176 39998 23892 8937 15608 16854 31009
8037 40401 13550 19526 41902 28782 13304 32796 24679 27140 45980 10021
40540 44498 13911 22435 32701 18405 39929 25521 12497 9851 39223 34823
15233 45333 5041 44979 45710 42150 19416 1892 23121 15860 8832 10308
10468 44296 3611 1480 37581 32254 13817 6883 32892 40258 46538 11940
6705 21634 28150 43757 895 6547 20970 28914 30117 25736 41734 11392
22002 5739 27210 27828 34192 37992 10915 6998 3824 42130 4494 35739
8515 1191 13642 30950 25943 12673 16726 34261 31828 3340 8747 39225
18979 17058 43130 4246 4793 44030 19454 29511 47929 15174 24333 19354
16694 8381 29642 46516 32224 26344 9405 18292 12437 27316 35466 41992
15642 5871 46489 26723 23396 7257 8974 3156 37420 44823 35423 13541
42858 32008 41282 38773 26570 2702 27260 46974 1469 20887 27426 38553
22152 24261 8297
19347 9978 27802
34991 6354 33561
29782 30875 29523
9278 48512 14349
38061 4165 43878
8548 33172 34410
22535 28811 23950
20439 4027 24186
38618 8187 30947
35538 43880 21459
7091 45616 15063
5505 9315 21908
36046 32914 11836
7304 39782 33721
16905 29962 12980
11171 23709 22460
34541 9937 44500
14035 47316 8815
15057 45482 24461
30518 36877 879
7583 13364 24332
448 27056 4682
12083 31378 21670
1159 18031 2221
17028 38715 9350
17343 24530 29574
46128 31039 32818
20373 36967 18345
46685 20622 32806

Table B.2: $N = 64800$, Code Rate = 1/3

34903 20927 32093 1052 25611 16093 16454 5520 506 37399 18518 21120
11636 14594 22158 14763 15333 6838 22222 37856 14985 31041 18704 32910
17449 1665 35639 16624 12867 12449 10241 11650 25622 34372 19878 26894
29235 19780 36056 20129 20029 5457 8157 35554 21237 7943 13873 14980
9912 7143 35911 12043 17360 37253 25588 11827 29152 21936 24125 40870
40701 36035 39556 12366 19946 29072 16365 35495 22686 11106 8756 34863
19165 15702 13536 40238 4465 40034 40590 37540 17162 1712 20577 14138
31338 19342 9301 39375 3211 1316 33409 28670 12282 6118 29236 35787
11504 30506 19558 5100 24188 24738 30397 33775 9699 6215 3397 37451
34689 23126 7571 1058 12127 27518 23064 11265 14867 30451 28289 2966
11660 15334 16867 15160 38343 3778 4265 39139 17293 26229 42604 13486
31497 1365 14828 7453 26350 41346 28643 23421 8354 16255 11055 24279
15687 12467 13906 5215 41328 23755 20800 6447 7970 2803 33262 39843
5363 22469 38091 28457 36696 34471 23619 2404 24229 41754 1297 18563
3673 39070 14480 30279 37483 7580 29519 30519 39831 20252 18132 20010
34386 7252 27526 12950 6875 43020 31566 39069 18985 15541 40020 16715
1721 37332 39953 17430 32134 29162 10490 12971 28581 29331 6489 35383
736 7022 42349 8783 6767 11871 21675 10325 11548 25978 431 24085
1925 10602 28585 12170 15156 34404 8351 13273 20208 5800 15367 21764
16279 37832 34792 21250 34192 7406 41488 18346 29227 26127 25493 7048
39948 28229 24899
17408 14274 38993
38774 15968 28459
41404 27249 27425
41229 6082 43114
13957 4979 40654
3093 3438 34992
34082 6172 28760
42210 34141 41021
14705 17783 10134
41755 39884 22773
14615 15593 1642
29111 37061 39860
9579 33552 633
12951 21137 39608
38244 27361 29417
2939 10172 36479

29094 5357 19224
9562 24436 28637
40177 2326 13504
6834 21583 42516
40651 42810 25709
31557 32138 38142
18624 41867 39296
37560 14295 16245
6821 21679 31570
25339 25083 22081
8047 697 35268
9884 17073 19995
26848 35245 8390
18658 16134 14807
12201 32944 5035
25236 1216 38986
42994 24782 8681
28321 4932 34249
4107 29382 32124
22157 2624 14468
38788 27081 7936
4368 26148 10578
25353 4122 39751

Table B.3: $N = 64800$, Code Rate = 2/5

31413 18834 28884 947 23050 14484 14809 4968 455 33659 16666 19008
13172 19939 13354 13719 6132 20086 34040 13442 27958 16813 29619 16553
1499 32075 14962 11578 11204 9217 10485 23062 30936 17892 24204 24885
32490 18086 18007 4957 7285 32073 19038 7152 12486 13483 24808 21759
32321 10839 15620 33521 23030 10646 26236 19744 21713 36784 8016 12869
35597 11129 17948 26160 14729 31943 20416 10000 7882 31380 27858 33356
14125 12131 36199 4058 35992 36594 33698 15475 1566 18498 12725 7067
17406 8372 35437 2888 1184 30068 25802 11056 5507 26313 32205 37232
15254 5365 17308 22519 35009 718 5240 16778 23131 24092 20587 33385
27455 17602 4590 21767 22266 27357 30400 8732 5596 3060 33703 3596
6882 873 10997 24738 20770 10067 13379 27409 25463 2673 6998 31378
15181 13645 34501 3393 3840 35227 15562 23615 38342 12139 19471 15483
13350 6707 23709 37204 25778 21082 7511 14588 10010 21854 28375 33591
12514 4695 37190 21379 18723 5802 7182 2529 29936 35860 28338 10835
34283 25610 33026 31017 21259 2165 21807 37578 1175 16710 21939 30841
27292 33730 6836 26476 27539 35784 18245 16394 17939 23094 19216 17432
11655 6183 38708 28408 35157 17089 13998 36029 15052 16617 5638 36464
15693 28923 26245 9432 11675 25720 26405 5838 31851 26898 8090 37037
24418 27583 7959 35562 37771 17784 11382 11156 37855 7073 21685 34515
10977 13633 30969 7516 11943 18199 5231 13825 19589 23661 11150 35602
19124 30774 6670 37344 16510 26317 23518 22957 6348 34069 8845 20175
34985 14441 25668 4116 3019 21049 37308 24551 24727 20104 24850 12114
38187 28527 13108 13985 1425 21477 30807 8613 26241 33368 35913 32477
5903 34390 24641 26556 23007 27305 38247 2621 9122 32806 21554 18685
17287 27292 19033

25796 31795 12152
12184 35088 31226
38263 33386 24892
23114 37995 29796
34336 10551 36245
35407 175 7203
14654 38201 22605
28404 6595 1018
19932 3524 29305
31749 20247 8128
18026 36357 26735
7543 29767 13588
13333 25965 8463
14504 36796 19710
4528 25299 7318
35091 25550 14798
7824 215 1248
30848 5362 17291
28932 30249 27073
13062 2103 16206
7129 32062 19612
9512 21936 38833
35849 33754 23450
18705 28656 18111
22749 27456 32187

28229 31684 30160
15293 8483 28002
14880 13334 12584
28646 2558 19687
6259 4499 26336
11952 28386 8405
10609 961 7582
10423 13191 26818
15922 36654 21450
10492 1532 1205
30551 36482 22153
5156 11330 34243
28616 35369 13322
8962 1485 21186
23541 17445 35561
33133 11593 19895
33917 7863 33651
20063 28331 10702
13195 21107 21859
4364 31137 4804
5585 2037 4830
30672 16927 14800

Table B.4: $N = 64800$, Code Rate $= 1/2$

54 9318 14392 27561 26909 10219 2534 8597
55 7263 4635 2530 28130 3033 23830 3651
56 24731 23583 26036 17299 5750 792 9169
57 5811 26154 18653 11551 15447 13685 16264
58 12610 11347 28768 2792 3174 29371 12997
59 16789 16018 21449 6165 21202 15850 3186
60 31016 21449 17618 6213 12166 8334 18212
61 22836 14213 11327 5896 718 11727 9308
62 2091 24941 29966 23634 9013 15587 5444
63 22207 3983 16904 28534 21415 27524 25912
64 25687 4501 22193 14665 14798 16158 5491
65 4520 17094 23397 4264 22370 16941 21526
66 10490 6182 32370 9597 30841 25954 2762
67 22120 22865 29870 15147 13668 14955 19235
68 6689 18408 18346 9918 25746 5443 20645
69 29982 12529 13858 4746 30370 10023 24828
70 1262 28032 29888 13063 24033 21951 7863
71 6594 29642 31451 14831 9509 9335 31552
72 1358 6454 16633 20354 24598 624 5265
73 19529 295 18011 3080 13364 8032 15323
74 11981 1510 7960 21462 9129 11370 25741
75 9276 29656 4543 30699 20646 21921 28050
76 15975 25634 5520 31119 13715 21949 19605
77 18688 4608 31755 30165 13103 10706 29224
78 21514 23117 12245 26035 31656 25631 30699
79 9674 24966 31285 29908 17042 24588 31857
80 21856 27777 29919 27000 14897 11409 7122
81 29773 23310 263 4877 28622 20545 22092
82 15605 5651 21864 3967 14419 22757 15896
83 30145 1759 10139 29223 26086 10556 5098
84 18815 16575 2936 24457 26738 6030 505
85 30326 22298 27562 20131 26390 6247 24791
86 928 29246 21246 12400 15311 32309 18608
87 20314 6025 26689 16302 2296 3244 19613
88 6237 11943 22851 15642 23857 15112 20947
89 26403 25168 19038 18384 8882 12719 7093
0 14567 24965
1 3908 100
2 10279 240
3 24102 764
4 12383 4173
5 13861 15918
6 21327 1046
7 5288 14579
8 28158 8069
9 16583 11098
10 16681 28363
11 13980 24725
12 32169 17989
13 10907 2767
14 21557 3818
15 26676 12422
16 7676 8754
17 14905 20232
18 15719 24646
19 31942 8589
20 19978 27197
21 27060 15071
22 6071 26649
23 10393 11176
24 9597 13370
25 7081 17677
26 1433 19513
27 26925 9014
28 19202 8900
29 18152 30647
30 20803 1737
31 11804 25221
32 31683 17783
33 29694 9345
34 12280 26611
35 6526 26122
36 26165 11241
37 7666 26962
38 16290 8480
39 11774 10120
40 30051 30426
41 1335 15424
42 6865 17742
43 31779 12489
44 32120 21001
45 14508 6996
46 979 25024
47 4554 21896
48 7989 21777
49 4972 20661
50 6612 2730
51 12742 4418
52 29194 595
53 19267 20113

Table B.5: $N = 64800$, Code Rate = 3/5

22422 10282 11626 19997 11161 2922 3122 99 5625 17064 8270 179
25087 16218 17015 828 20041 25656 4186 11629 22599 17305 22515 6463
11049 22853 25706 14388 5500 19245 8732 2177 13555 11346 17265 3069
16581 22225 12563 19717 23577 11555 25496 6853 25403 5218 15925 21766
16529 14487 7643 10715 17442 11119 5679 14155 24213 21000 1116 15620
5340 8636 16693 1434 5635 6516 9482 20189 1066 15013 25361 14243
18506 22236 20912 8952 5421 15691 6126 21595 500 6904 13059 6802
8433 4694 5524 14216 3685 19721 25420 9937 23813 9047 25651 16826
21500 24814 6344 17382 7064 13929 4004 16552 12818 8720 5286 2206
22517 2429 19065 2921 21611 1873 7507 5661 23006 23128 20543 19777
1770 4636 20900 14931 9247 12340 11008 12966 4471 2731 16445 791
6635 14556 18865 22421 22124 12697 9803 25485 7744 18254 11313 9004
19982 23963 18912 7206 12500 4382 20067 6177 21007 1195 23547 24837
756 11158 14646 20534 3647 17728 11676 11843 12937 4402 8261 22944
9306 24009 10012 11081 3746 24325 8060 19826 842 8836 2898 5019
7575 7455 25244 4736 14400 22981 5543 8006 24203 13053 1120 5128
3482 9270 13059 15825 7453 23747 3656 24585 16542 17507 22462 14670
15627 15290 4198 22748 5842 13395 23918 16985 14929 3726 25350 24157
24896 16365 16423 13461 16615 8107 24741 3604 25904 8716 9604 20365
3729 17245 18448 9862 20831 25326 20517 24618 13282 5099 14183 8804
16455 17646 15376 18194 25528 1777 6066 21855 14372 12517 4488 17490
1400 8135 23375 20879 8476 4084 12936 25536 22309 16582 6402 24360
25119 23586 128 4761 10443 22536 8607 9752 25446 15053 1856 4040
377 21160 13474 5451 17170 5938 10256 11972 24210 17833 22047 16108
13075 9648 24546 13150 23867 7309 19798 2988 16858 4825 23950 15125
20526 3553 11525 23366 2452 17626 19265 20172 18060 24593 13255 1552
18839 21132 20119 15214 14705 7096 10174 5663 18651 19700 12524 14033
4127 2971 17499 16287 22368 21463 7943 18880 5567 8047 23363 6797
10651 24471 14325 4081 7258 4949 7044 1078 797 22910 20474 4318
21374 13231 22985 5056 3821 23718 14178 9978 19030 23594 8895 25358
6199 22056 7749 13310 3999 23697 16445 22636 5225 22437 24153 9442
7978 12177 2893 20778 3175 8645 11863 24623 10311 25767 17057 3691
20473 11294 9914 22815 2574 8439 3699 5431 24840 21908 16088 18244
8208 5755 19059 8541 24924 6454 11234 10492 16406 10831 11436 9649
16264 11275 24953 2347 12667 19190 7257 7174 24819 2938 2522 11749
3627 5969 13862 1538 23176 6353 2855 17720 2472 7428 573 15036
0 18539 18661
1 10502 3002
2 9368 10761
3 12299 7828
4 15048 13362
5 18444 24640
6 20775 19175
7 18970 10971
8 5329 19982
9 11296 18655
10 15046 20659
11 7300 22140
12 22029 14477
13 11129 742
14 13254 13813
15 19234 13273
16 6079 21122
17 22782 5828
18 19775 4247
19 1660 19413
20 4403 3649
21 13371 25851
22 22770 21784
23 10757 14131
24 16071 21617
25 6393 3725
26 597 19968
27 5743 8084
28 6770 9548
29 4285 17542
30 13568 22599
31 1786 4617
32 23238 11648
33 19627 2030
34 13601 13458
35 13740 17328
36 25012 13944
37 22513 6687
38 4934 12587
39 21197 5133
40 22705 6938
41 7534 24633
42 24400 12797
43 21911 25712
44 12039 1140
45 24306 1021
46 14012 20747
47 11265 15219
48 4670 15531
49 9417 14359
50 2415 6504
51 24964 24690
52 14443 8816
53 6926 1291
54 6209 20806
55 13915 4079
56 24410 13196
57 13505 6117
58 9869 8220
59 1570 6044
60 25780 17387
61 20671 24913
62 24558 20591
63 12402 3702
64 8314 1357
65 20071 14616
66 17014 3688
67 19837 946
68 15195 12136
69 7758 22808
70 3564 2925
71 3434 7769

Table B.6: $N = 64800$, Code Rate $= 2/3$

0 10491 16043 506 12826 8065 8226 2767 240 18673 9279 10579 20928
1 17819 8313 6433 6224 5120 5824 12812 17187 9940 13447 13825 18483
2 17957 6024 8681 18628 12794 5915 14576 10970 12064 20437 4455 7151
3 19777 6183 9972 14536 8182 17749 11341 5556 4379 17434 15477 18532
4 4651 19689 1608 659 16707 14335 6143 3058 14618 17894 20684 5306
5 9778 2552 12096 12369 15198 16890 4851 3109 1700 18725 1997 15882
6 486 6111 13743 11537 5591 7433 15227 14145 1483 3887 17431 12430
7 20647 14311 11734 4180 8110 5525 12141 15761 18661 18441 10569 8192
8 3791 14759 15264 19918 10132 9062 10010 12786 10675 9682 19246 5454
9 19525 9485 7777 19999 8378 9209 3163 20232 6690 16518 716 7353
10 4588 6709 20202 10905 915 4317 11073 13576 16433 368 3508 21171
11 14072 4033 19959 12608 631 19494 14160 8249 10223 21504 12395 4322
12 13800 14161
13 2948 9647
14 14693 16027
15 20506 11082
16 1143 9020
17 13501 4014
18 1548 2190
19 12216 21556
20 2095 19897
21 4189 7958
22 15940 10048
23 515 12614
24 8501 8450
25 17595 16784
26 5913 8495
27 16394 10423
28 7409 6981
29 6678 15939
30 20344 12987
31 2510 14588
32 17918 6655
33 6703 19451
34 496 4217
35 7290 5766
36 10521 8925
37 20379 11905
38 4090 5838
39 19082 17040
40 20233 12352
41 19365 19546
42 6249 19030
43 11037 19193
44 19760 11772
45 19644 7428
46 16076 3521
47 11779 21062
48 13062 9682
49 8934 5217
50 11087 3319
51 18892 4356
52 7894 3898
53 5963 4360
54 7346 11726
55 5182 5609
56 2412 17295
57 9845 20494
58 6687 1864
59 20564 5216
0 18226 17207
1 9380 8266
2 7073 3065
3 18252 13437
4 9161 15642
5 10714 10153
6 11585 9078
7 5359 9418
8 9024 9515
9 1206 16354
10 14994 1102
11 9375 20796
12 15964 6027
13 14789 6452
14 8002 18591
15 14742 14089
16 253 3045
17 1274 19286
18 14777 2044
19 13920 9900
20 452 7374
21 18206 9921
22 6131 5414
23 10077 9726
24 12045 5479
25 4322 7990
26 15616 5550
27 15561 10661
28 20718 7387
29 2518 18804
30 8984 2600
31 6516 17909
32 11148 98
33 20559 3704
34 7510 1569
35 16000 11692
36 9147 10303
37 16650 191
38 15577 18685
39 17167 20917
40 4256 3391
41 20092 17219
42 9218 5056
43 18429 8472
44 12093 20753
45 16345 12748
46 16023 11095
47 5048 17595
48 18995 4817
49 16483 3536
50 1439 16148
51 3661 3039
52 19010 18121
53 8968 11793
54 13427 18003
55 5303 3083
56 531 16668
57 4771 6722
58 5695 7960
59 3589 14630

Table B.7: $N = 64800$, Code Rate = 3/4

0 6385 7901 14611 13389 11200 3252 5243 2504 2722 821 7374
1 11359 2698 357 13824 12772 7244 6752 15310 852 2001 11417
2 7862 7977 6321 13612 12197 14449 15137 13860 1708 6399 13444
3 1560 11804 6975 13292 3646 3812 8772 7306 5795 14327 7866
4 7626 11407 14599 9689 1628 2113 10809 9283 1230 15241 4870
5 1610 5699 15876 9446 12515 1400 6303 5411 14181 13925 7358
6 4059 8836 3405 7853 7992 15336 5970 10368 10278 9675 4651
7 4441 3963 9153 2109 12683 7459 12030 12221 629 15212 406
8 6007 8411 5771 3497 543 14202 875 9186 6235 13908 3563
9 3232 6625 4795 546 9781 2071 7312 3399 7250 4932 12652
10 8820 10088 11090 7069 6585 13134 10158 7183 488 7455 9238
11 1903 10818 119 215 7558 11046 10615 11545 14784 7961 15619
12 3655 8736 4917 15874 5129 2134 15944 14768 7150 2692 1469
13 8316 3820 505 8923 6757 806 7957 4216 15589 13244 2622
14 14463 4852 15733 3041 11193 12860 13673 8152 6551 15108 8758
15 3149 11981
16 13416 6906
17 13098 13352
18 2009 14460
19 7207 4314
20 3312 3945
21 4418 6248
22 2669 13975
23 7571 9023
24 14172 2967
25 7271 7138
26 6135 13670
27 7490 14559
28 8657 2466
29 8599 12834
30 3470 3152
31 13917 4365
32 6024 13730
33 10973 14182
34 2464 13167
35 5281 15049
36 1103 1849
37 2058 1069
38 9654 6095
39 14311 7667
40 15617 8146
41 4588 11218
42 13660 6243
43 8578 7874
44 11741 2686
0 1022 1264
1 12604 9965
2 8217 2707
3 3156 11793
4 354 1514
5 6978 14058
6 7922 16079
7 15087 12138
8 5053 6470
9 12687 14932
10 15458 1763
11 8121 1721
12 12431 549
13 4129 7091
14 1426 8415
15 9783 7604
16 6295 11329
17 1409 12061
18 8065 9087
19 2918 8438
20 1293 14115
21 3922 13851
22 3851 4000
23 5865 1768
24 2655 14957
25 5565 6332
26 4303 12631
27 11653 12236
28 16025 7632
29 4655 14128
30 9584 13123
31 13987 9597
32 15409 12110
33 8754 15490
34 7416 15325
35 2909 15549
36 2995 8257
37 9406 4791
38 11111 4854
39 2812 8521
40 8476 14717
41 7820 15360
42 1179 7939
43 2357 8678
44 7703 6216
0 3477 7067
1 3931 13845
2 7675 12899
3 1754 8187
4 7785 1400
5 9213 5891
6 2494 7703
7 2576 7902
8 4821 15682
9 10426 11935
10 1810 904
11 11332 9264
12 11312 3570
13 14916 2650
14 7679 7842
15 6089 13084
16 3938 2751
17 8509 4648
18 12204 8917
19 5749 12443
20 12613 4431
21 1344 4014
22 8488 13850
23 1730 14896
24 14942 7126
25 14983 8863
26 6578 8564
27 4947 396
28 297 12805
29 13878 6692
30 11857 11186
31 14395 11493
32 16145 12251
33 13462 7428
34 14526 13119
35 2535 11243
36 6465 12690
37 6872 9334
38 15371 14023
39 8101 10187
40 11963 4848
41 15125 6119
42 8051 14465
43 11139 5167
44 2883 14521

Table B.8: $N = 64800$, Code Rate = 4/5

0 149 11212 5575 6360 12559 8108 8505 408 10026 12828
1 5237 490 10677 4998 3869 3734 3092 3509 7703 10305
2 8742 5553 2820 7085 12116 10485 564 7795 2972 2157
3 2699 4304 8350 712 2841 3250 4731 10105 517 7516
4 12067 1351 11992 12191 11267 5161 537 6166 4246 2363
5 6828 7107 2127 3724 5743 11040 10756 4073 1011 3422
6 11259 1216 9526 1466 10816 940 3744 2815 11506 11573
7 4549 11507 1118 1274 11751 5207 7854 12803 4047 6484
8 8430 4115 9440 413 4455 2262 7915 12402 8579 7052
9 3885 9126 5665 4505 2343 253 4707 3742 4166 1556
10 1704 8936 6775 8639 8179 7954 8234 7850 8883 8713
11 11716 4344 9087 11264 2274 8832 9147 11930 6054 5455
12 7323 3970 10329 2170 8262 3854 2087 12899 9497 11700
13 4418 1467 2490 5841 817 11453 533 11217 11962 5251
14 1541 4525 7976 3457 9536 7725 3788 2982 6307 5997
15 11484 2739 4023 12107 6516 551 2572 6628 8150 9852
16 6070 1761 4627 6534 7913 3730 11866 1813 12306 8249
17 12441 5489 8748 7837 7660 2102 11341 2936 6712 11977
18 10155 4210
19 1010 10483
20 8900 10250
21 10243 12278
22 7070 4397
23 12271 3887
24 11980 6836
25 9514 4356
26 7137 10281
27 11881 2526
28 1969 11477
29 3044 10921
30 2236 8724
31 9104 6340
32 7342 8582
33 11675 10405
34 6467 12775
35 3186 12198
0 9621 11445
1 7486 5611
2 4319 4879
3 2196 344
4 7527 6650
5 10693 2440
6 6755 2706
7 5144 5998
8 11043 8033
9 4846 4435
10 4157 9228
11 12270 6562
12 11954 7592
13 7420 2592
14 8810 9636
15 689 5430
16 920 1304
17 1253 11934
18 9559 6016
19 312 7589
20 4439 4197
21 4002 9555
22 12232 7779
23 1494 8782
24 10749 3969
25 4368 3479
26 6316 5342
27 2455 3493
28 12157 7405
29 6598 11495
30 11805 4455
31 9625 2090
32 4731 2321
33 3578 2608
34 8504 1849
35 4027 1151
0 5647 4935
1 4219 1870
2 10968 8054
3 6970 5447
4 3217 5638
5 8972 669
6 5618 12472
7 1457 1280
8 8868 3883
9 8866 1224
10 8371 5972
11 266 4405
12 3706 3244
13 6039 5844
14 7200 3283
15 1502 11282
16 12318 2202
17 4523 965
18 9587 7011
19 2552 2051
20 12045 10306
21 11070 5104
22 6627 6906
23 9889 2121
24 829 9701
25 2201 1819
26 6689 12925
27 2139 8757
28 12004 5948
29 8704 3191
30 8171 10933
31 6297 7116
32 616 7146
33 5142 9761
34 10377 8138
35 7616 5811
0 7285 9863
1 7764 10867
2 12343 9019
3 4414 8331
4 3464 642
5 6960 2039
6 786 3021
7 710 2086
8 7423 5601
9 8120 4885
10 12385 11990
11 9739 10034
12 424 10162
13 1347 7597
14 1450 112
15 7965 8478
16 8945 7397
17 6590 8316
18 6838 9011
19 6174 9410
20 255 113
21 6197 5835
22 12902 3844
23 4377 3505
24 5478 8672
25 4453 2132
26 9724 1380
27 12131 11526
28 12323 9511
29 8231 1752
30 497 9022
31 9288 3080
32 2481 7515
33 2696 268
34 4023 12341
35 7108 5553

Table B.9: $N = 64800$, Code Rate = 5/6

0 4362 416 8909 4156 3216 3112 2560 2912 6405 8593 4969 6723
1 2479 1786 8978 3011 4339 9313 6397 2957 7288 5484 6031 10217
2 10175 9009 9889 3091 4985 7267 4092 8874 5671 2777 2189 8716
3 9052 4795 3924 3370 10058 1128 9996 1016 5 9360 4297 434 5138
4 2379 7834 4835 2327 9843 804 329 8353 7167 3070 1528 7311
5 3435 7871 348 3693 1876 6585 10340 7144 5870 2084 4052 2780
6 3917 3111 3476 1304 10331 5939 5199 1611 1991 699 8316 9960
7 6883 3237 1717 10752 7891 9764 4745 3888 10009 4176 4614 1567
8 10587 2195 1689 2968 5420 2580 2883 6496 111 6023 1024 4449
9 3786 8593 2074 3321 5057 1450 3840 5444 6572 3094 9892 1512
10 8548 1848 10372 4585 7313 6536 6379 1766 9462 2456 5606 9975
11 8204 10593 7935 3636 3882 394 5968 8561 2395 7289 9267 9978
12 7795 74 1633 9542 6867 7352 6417 7568 10623 725 2531 9115
13 7151 2482 4260 5003 10105 7419 9203 6691 8798 2092 8263 3755
14 3600 570 4527 200 9718 6771 1995 8902 5446 768 1103 6520
15 6304 7621
16 6498 9209
17 7293 6786
18 5950 1708
19 8521 1793
20 6174 7854
21 9773 1190
22 9517 10268
23 2181 9349
24 1949 5560
25 1556 555
26 8600 3827
27 5072 1057
28 7928 3542
29 3226 3762
0 7045 2420
1 9645 2641
2 2774 2452
3 5331 2031
4 9400 7503
5 1850 2338
6 10456 9774
7 1692 9276
8 10037 4038
9 3964 338
10 2640 5087
11 858 3473
12 5582 5683
13 9523 916
14 4107 1559
15 4506 3491
16 8191 4182
17 10192 6157
18 5668 3305
19 3449 1540
20 4766 2697
21 4069 6675
22 1117 1016
23 5619 3085
24 8483 8400
25 8255 394
26 6338 5042
27 6174 5119
28 7203 1989
29 1781 5174
0 1464 3559
1 3376 4214
2 7238 67
3 10595 8831
4 1221 6513
5 5300 4652
6 1429 9749
7 7878 5131
8 4435 10284
9 6331 5507
10 6662 4941
11 9614 10238
12 8400 8025
13 9156 5630
14 7067 8878
15 9027 3415
16 1690 3866
17 2854 8469
18 6206 630
19 363 5453
20 4125 7008
21 1612 6702
22 9069 9226
23 5767 4060
24 3743 9237
25 7018 5572
26 8892 4536
27 853 6064
28 8069 5893
29 2051 2885
0 10691 3153
1 3602 4055
2 328 1717
3 2219 9299
4 1939 7898
5 617 206
6 8544 1374
7 10676 3240
8 6672 9489
9 3170 7457
10 7868 5731
11 6121 10732
12 4843 9132
13 580 9591
14 6267 9290
15 3009 2268
16 195 2419
17 8016 1557
18 1516 9195
19 8062 9064
20 2095 8968
21 753 7326
22 6291 3833
23 2614 7844
24 2303 646
25 2075 611
26 4687 362
27 8684 9940
28 4830 2065
29 7038 1363
0 1769 7837
1 3801 1689
2 10070 2359
3 3667 9918
4 1914 6920
5 4244 5669
6 10245 7821
7 7648 3944
8 3310 5488
9 6346 9666
10 7088 6122
11 1291 7827
12 10592 8945
13 3609 7120
14 9168 9112
15 6203 8052
16 3330 2895
17 4264 10563
18 10556 6496
19 8807 7645
20 1999 4530
21 9202 6818
22 3403 1734
23 2106 9023
24 6881 3883
25 3895 2171
26 4062 6424
27 3755 9536

Table B.10: $N = 64800$, Code Rate = 8/9

0 6235 2848 3222
1 5800 3492 5348
2 2757 927 90
3 6961 4516 4739
4 1172 3237 6264
5 1927 2425 3683
6 3714 6309 2495
7 3070 6342 7154
8 2428 613 3761
9 2906 264 5927
10 1716 1950 4273
11 4613 6179 3491
12 4865 3286 6005
13 1343 5923 3529
14 4589 4035 2132
15 1579 3920 6737
16 1644 1191 5998
17 1482 2381 4620
18 6791 6014 6596
19 2738 5918 3786
0 5156 6166
1 1504 4356
2 130 1904
3 6027 3187
4 6718 759
5 6240 2870
6 2343 1311
7 1039 5465
8 6617 2513
9 1588 5222
10 6561 535
11 4765 2054
12 5966 6892

13 1969 3869
14 3571 2420
15 4632 981
16 3215 4163
17 973 3117
18 3802 6198
19 3794 3948
0 3196 6126
1 573 1909
2 850 4034
3 5622 1601
4 6005 524
5 5251 5783
6 172 2032
7 1875 2475
8 497 1291
9 2566 3430
10 1249 740
11 2944 1948
12 6528 2899
13 2243 3616
14 867 3733
15 1374 4702
16 4698 2285
17 4760 3917
18 1859 4058
19 6141 3527
0 2148 5066
1 1306 145
2 2319 871
3 3463 1061
4 5554 6647
5 5837 339

6 5821 4932
7 6356 4756
8 3930 418
9 211 3094
10 1007 4928
11 3584 1235
12 6982 2869
13 1612 1013
14 953 4964
15 4555 4410
16 4925 4842
17 5778 600
18 6509 2417
19 1260 4903
0 3369 3031
1 3557 3224
2 3028 583
3 3258 440
4 6226 6655
5 4895 1094
6 1481 6847
7 4433 1932
8 2107 1649
9 2119 2065
10 4003 6388
11 6720 3622
12 3694 4521
13 1164 7050
14 1965 3613
15 4331 66
16 2970 1796
17 4652 3218
18 1762 4777

19 5736 1399
0 970 2572
1 2062 6599
2 4597 4870
3 1228 6913
4 4159 1037
5 2916 2362
6 395 1226
7 6911 4548
8 4618 2241
9 4120 4280
10 5825 474
11 2154 5558
12 3793 5471
13 5707 1595
14 1403 325
15 6601 5183
16 6369 4569
17 4846 896
18 7092 6184
19 6764 7127
0 6358 1951
1 3117 6960
2 2710 7062
3 1133 3604
4 3694 657
5 1355 110
6 3329 6736
7 2505 3407
8 2462 4806
9 4216 214
10 5348 5619
11 6627 6243

12 2644 5073
13 4212 5088
14 3463 3889
15 5306 478
16 4320 6121
17 3961 1125
18 5699 1195
19 6511 792
0 3934 2778
1 3238 6587
2 1111 6596
3 1457 6226
4 1446 3885
5 3907 4043
6 6839 2873
7 1733 5615
8 5202 4269
9 3024 4722
10 5445 6372
11 370 1828
12 4695 1600
13 680 2074
14 1801 6690
15 2669 1377
16 2463 1681
17 5972 5171
18 5728 4284
19 1696 1459

Table B.11: $N = 64800$, Code Rate = 9/10

0 5611 2563 2900
1 5220 3143 4813
2 2481 834 81
3 6265 4064 4265
4 1055 2914 5638
5 1734 2182 3315
6 3342 5678 2246
7 2185 552 3385
8 2615 236 5334
9 1546 1755 3846
10 4154 5561 3142
11 4382 2957 5400
12 1209 5329 3179
13 1421 3528 6063
14 1480 1072 5398
15 3843 1777 4369
16 1334 2145 4163
17 2368 5055 260
0 6118 5405
1 2994 4370
2 3405 1669
3 4640 5550
4 1354 3921
5 117 1713
6 5425 2866
7 6047 683
8 5616 2582
9 2108 1179
10 933 4921
11 5953 2261
12 1430 4699
13 5905 480
14 4289 1846
15 5374 6208
16 1775 3476

17 3216 2178
0 4165 884
1 2896 3744
2 874 2801
3 3423 5579
4 3404 3552
5 2876 5515
6 516 1719
7 765 3631
8 5059 1441
9 5629 598
10 5405 473
11 4724 5210
12 155 1832
13 1689 2229
14 449 1164
15 2308 3088
16 1122 669
17 2268 5758
0 5878 2609
1 782 3359
2 1231 4231
3 4225 2052
4 4286 3517
5 5531 3184
6 1935 4560
7 1174 131
8 3115 956
9 3129 1088
10 5238 4440
11 5722 4280
12 3540 375
13 191 2782
14 906 4432
15 3225 1111

16 6296 2583
17 1457 903
0 855 4475
1 4097 3970
2 4433 4361
3 5198 541
4 1146 4426
5 3202 2902
6 2724 525
7 1083 4124
8 2326 6003
9 5605 5990
10 4376 1579
11 4407 984
12 1332 6163
13 5359 3975
14 1907 1854
15 3601 5748
16 6056 3266
17 3322 4085
0 1768 3244
1 2149 144
2 1589 4291
3 5154 1252
4 1855 5939
5 4820 2706
6 1475 3360
7 4266 693
8 4156 2018
9 2103 752
10 3710 3853
11 5123 931
12 6146 3323
13 1939 5002
14 5140 1437

15 1263 293
16 5949 4665
17 4548 6380
0 3171 4690
1 5204 2114
2 6384 5565
3 5722 1757
4 2805 6264
5 1202 2616
6 1018 3244
7 4018 5289
8 2257 3067
9 2483 3073
10 1196 5329
11 649 3918
12 3791 4581
13 5028 3803
14 3119 3506
15 4779 431
16 3888 5510
17 4387 4084
0 5836 1692
1 5126 1078
2 5721 6165
3 3540 2499
4 2225 6348
5 1044 1484
6 6323 4042
7 1313 5603
8 1303 3496
9 3516 3639
10 5161 2293
11 4682 3845
12 3045 643
13 2818 2616

14 3267 649
15 6236 593
16 646 2948
17 4213 1442
0 5779 1596
1 2403 1237
2 2217 1514
3 5609 716
4 5155 3858
5 1517 1312
6 2554 3158
7 5280 2643
8 4990 1353
9 5648 1170
10 1152 4366
11 3561 5368
12 3581 1411
13 5647 4661
14 1542 5401
15 5078 2687
16 316 1755
17 3392 1991

Table B.12: $N = 16200$, Code Rate = 1/5

6295 9626 304 7695 4839 4936 1660 144 11203 5567 6347 12557
10691 4988 3859 3734 3071 3494 7687 10313 5964 8069 8296 11090
10774 3613 5208 11177 7676 3549 8746 6583 7239 12265 2674 4292
11869 3708 5981 8718 4908 10650 6805 3334 2627 10461 9285 11120
7844 3079 10773
3385 10854 5747
1360 12010 12202
6189 4241 2343
9840 12726 4977

Table B.13: $N = 16200$, Code Rate = 1/3

416 8909 4156 3216 3112 2560 2912 6405 8593 4969 6723 6912
8978 3011 4339 9312 6396 2957 7288 5485 6031 10218 2226 3575
3383 10059 1114 10008 10147 9384 4290 434 5139 3536 1965 2291
2797 3693 7615 7077 743 1941 8716 6215 3840 5140 4582 5420
6110 8551 1515 7404 4879 4946 5383 1831 3441 9569 10472 4306
1505 5682 7778
7172 6830 6623
7281 3941 3505
10270 8669 914
3622 7563 9388
9930 5058 4554
4844 9609 2707
6883 3237 1714
4768 3878 10017
10127 3334 8267

Table B.14: $N = 16200$, Code Rate = 2/5

5650 4143 8750 583 6720 8071 635 1767 1344 6922 738 6658
5696 1685 3207 415 7019 5023 5608 2605 857 6915 1770 8016
3992 771 2190 7258 8970 7792 1802 1866 6137 8841 886 1931
4108 3781 7577 6810 9322 8226 5396 5867 4428 8827 7766 2254
4247 888 4367 8821 9660 324 5864 4774 227 7889 6405 8963
9693 500 2520 2227 1811 9330 1928 5140 4030 4824 806 3134
1652 8171 1435
3366 6543 3745
9286 8509 4645
7397 5790 8972
6597 4422 1799
9276 4041 3847
8683 7378 4946
5348 1993 9186
6724 9015 5646
4502 4439 8474
5107 7342 9442
1387 8910 2660

Table B.15: $N = 16200$, Code Rate = 4/9

20 712 2386 6354 4061 1062 5045 5158
21 2543 5748 4822 2348 3089 6328 5876
22 926 5701 269 3693 2438 3190 3507
23 2802 4520 3577 5324 1091 4667 4449
24 5140 2003 1263 4742 6497 1185 6202
0 4046 6934
1 2855 66
2 6694 212
3 3439 1158
4 3850 4422
5 5924 290
6 1467 4049
7 7820 2242
8 4606 3080
9 4633 7877
10 3884 6868
11 8935 4996
12 3028 764
13 5988 1057
14 7411 3450

Table B.16: $N = 16200$, Code Rate = 3/5

2765 5713 6426 3596 1374 4811 2182 544 3394 2840 4310 771
4951 211 2208 723 1246 2928 398 5739 265 5601 5993 2615
210 4730 5777 3096 4282 6238 4939 1119 6463 5298 6320 4016
4167 2063 4757 3157 5664 3956 6045 563 4284 2441 3412 6334
4201 2428 4474 59 1721 736 2997 428 3807 1513 4732 6195
2670 3081 5139 3736 1999 5889 4362 3806 4534 5409 6384 5809
5516 1622 2906 3285 1257 5797 3816 817 875 2311 3543 1205
4244 2184 5415 1705 5642 4886 2333 287 1848 1121 3595 6022
2142 2830 4069 5654 1295 2951 3919 1356 884 1786 396 4738
0 2161 2653
1 1380 1461
2 2502 3707
3 3971 1057
4 5985 6062
5 1733 6028
6 3786 1936
7 4292 956
8 5692 3417
9 266 4878
10 4913 3247
11 4763 3937
12 3590 2903
13 2566 4215
14 5208 4707
15 3940 3388
16 5109 4556
17 4908 4177

Table B.17: $N = 16200$, Code Rate = 2/3

0 2084 1613 1548 1286 1460 3196 4297 2481 3369 3451 4620 2622
1 122 1516 3448 2880 1407 1847 3799 3529 373 971 4358 3108
2 259 3399 929 2650 864 3996 3833 107 5287 164 3125 2350
3 342 3529
4 4198 2147
5 1880 4836
6 3864 4910
7 243 1542
8 3011 1436
9 2167 2512
10 4606 1003
11 2835 705
12 3426 2365
13 3848 2474
14 1360 1743
0 163 2536

1 2583 1180
2 1542 509
3 4418 1005
4 5212 5117
5 2155 2922
6 347 2696
7 226 4296
8 1560 487
9 3926 1640
10 149 2928
11 2364 563
12 635 688
13 231 1684
14 1129 3894

Table B.18: $N = 16200$, Code Rate = 11/15

3 3198 478 4207 1481 1009 2616 1924 3437 554 683 1801
4 2681 2135
5 3107 4027
6 2637 3373
7 3830 3449
8 4129 2060
9 4184 2742
10 3946 1070
11 2239 984
0 1458 3031
1 3003 1328
2 1137 1716
3 132 3725
4 1817 638
5 1774 3447
6 3632 1257
7 542 3694

8 1015 1945
9 1948 412
10 995 2238
11 4141 1907
0 2480 3079
1 3021 1088
2 713 1379
3 997 3903
4 2323 3361
5 1110 986
6 2532 142
7 1690 2405
8 1298 1881
9 615 174
10 1648 3112
11 1415 2808

Table B.19: $N = 16200$, Code Rate = 7/9

5 896 1565
6 2493 184
7 212 3210
8 727 1339
9 3428 612
0 2663 1947
1 230 2695
2 2025 2794
3 3039 283
4 862 2889
5 376 2110
6 2034 2286

7 951 2068
8 3108 3542
9 307 1421
0 2272 1197
1 1800 3280
2 331 2308
3 465 2552
4 1038 2479
5 1383 343
6 94 236
7 2619 121
8 1497 277

9 2116 1855
0 722 1584
1 2767 1881
2 2701 1610
3 3283 1732
4 168 1099
5 3074 243
6 3460 945
7 2049 1746
8 566 1427
9 3545 1168

Table B.20: $N = 16200$, Code Rate = 37/49

3 2409 499 1481 908 559 716 1270 333 2508 2264 1702 2805
4 2447 1926
5 414 1224
6 2114 842
7 212 573
0 2383 2112
1 2286 2348
2 545 819
3 1264 143
4 1701 2258
5 964 166
6 114 2413
7 2243 81
0 1245 1581
1 775 169
2 1696 1104
3 1914 2831
4 532 1450
5 91 974
6 497 2228
7 2326 1579
0 2482 256
1 1117 1261
2 1257 1658
3 1478 1225
4 2511 980
5 2320 2675
6 435 1278
7 228 503
0 1885 2369
1 57 483
2 838 1050
3 1231 1990
4 1738 68
5 2392 951
6 163 645
7 2644 1704

Table B.21: $N = 16200$, Code Rate = 8/9

0 1558 712 805
1 1450 873 1337
2 1741 1129 1184
3 294 806 1566
4 482 605 923
0 926 1578
1 777 1374
2 608 151
3 1195 210
4 1484 692
0 427 488
1 828 1124
2 874 1366
3 1500 835
4 1496 502
0 1006 1701
1 1155 97
2 657 1403
3 1453 624
4 429 1495
0 809 385
1 367 151
2 1323 202
3 960 318
4 1451 1039
0 1098 1722
1 1015 1428
2 1261 1564
3 544 1190
4 1472 1246
0 508 630
1 421 1704
2 284 898
3 392 577
4 1155 556
0 631 1000
1 732 1368
2 1328 329
3 1515 506
4 1104 1172