

FLEXIBLE ENCODER AND DECODER DESIGNS FOR
LOW-DENSITY PARITY-CHECK CODES

by

SUNITHA KOPPARTHI

B.E., Andhra University, 2000
M.S., Louisiana State University, 2003

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2010

Abstract

Future technologies such as cognitive radio require flexible and reliable hardware architectures that can be easily configured and adapted to varying coding parameters. The objective of this work is to develop a flexible hardware encoder and decoder for low-density parity-check (LDPC) codes. The design methodologies used for the implementation of a LDPC encoder and decoder are flexible in terms of parity-check matrix, code rate and code length. All these designs are implemented on a programmable chip and tested.

Encoder implementations of LDPC codes are optimized for area due to their high complexity. Such designs usually have relatively low data rate. Two new encoder designs are developed that achieve much higher data rates of up to 844 Mbps while requiring more area for implementation. Using structured LDPC codes decreases the encoding complexity and provides design flexibility. The architecture for an encoder is presented that adheres to the structured LDPC codes defined in the IEEE 802.16e standard.

A single encoder design is also developed that accommodates different code lengths and code rates and does not require re-synthesis of the design in order to change the encoding parameters. The flexible encoder design for structured LDPC codes is also implemented on a custom chip. The maximum coded data rate of the structured encoder is up to 844 Mbps and for a given code rate its value is independent of the code length.

An LDPC decoder is designed and its design methodology is generic. It is applicable to both structured and any randomly generated LDPC codes. The coded data rate of the decoder increases with the increase in the code length. The number of decoding iterations used for the decoding process plays an important role in determining the decoder performance and latency. This design validates the estimated codeword after every iteration and stops the decoding process when the correct codeword is estimated which saves power consumption. For a given parity-check matrix and signal-to-noise ratio, a procedure to find an optimum value of the maximum number of decoding iterations is presented that considers the affects of power, delay, and error performance.

FLEXIBLE ENCODER AND DECODER DESIGNS FOR
LOW-DENSITY PARITY-CHECK CODES

by

SUNITHA KOPPARTHI

B.E., Andhra University, 2000
M.S., Louisiana State University, 2003

A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2010

Approved by:

Major Professor
Don M. Gruenbacher

Copyright

SUNITHA KOPPARTHI

2010

Abstract

Future technologies such as cognitive radio require flexible and reliable hardware architectures that can be easily configured and adapted to varying coding parameters. The objective of this work is to develop a flexible hardware encoder and decoder for low-density parity-check (LDPC) codes. The design methodologies used for the implementation of a LDPC encoder and decoder are flexible in terms of parity-check matrix, code rate and code length. All these designs are implemented on a programmable chip and tested.

Encoder implementations of LDPC codes are optimized for area due to their high complexity. Such designs usually have relatively low data rate. Two new encoder designs are developed that achieve much higher data rates of up to 844 Mbps while requiring more area for implementation. Using structured LDPC codes decreases the encoding complexity and provides design flexibility. The architecture for an encoder is presented that adheres to the structured LDPC codes defined in the IEEE 802.16e standard.

A single encoder design is also developed that accommodates different code lengths and code rates and does not require re-synthesis of the design in order to change the encoding parameters. The flexible encoder design for structured LDPC codes is also implemented on a custom chip. The maximum coded data rate of the structured encoder is up to 844 Mbps and for a given code rate its value is independent of the code length.

An LDPC decoder is designed and its design methodology is generic. It is applicable to both structured and any randomly generated LDPC codes. The coded data rate of the decoder increases with the increase in the code length. The number of decoding iterations used for the decoding process plays an important role in determining the decoder performance and latency. This design validates the estimated codeword after every iteration and stops the decoding process when the correct codeword is estimated which saves power consumption. For a given parity-check matrix and signal-to-noise ratio, a procedure to find an optimum value of the maximum number of decoding iterations is presented that considers the affects of power, delay, and error performance.

Table of Contents

List of Figures	x
List of Tables	xiv
Acknowledgements.....	xvi
Dedication	xvii
CHAPTER 1 - Introduction	1
1.1 Motivation.....	2
1.2 Literature Review	4
1.2.1 Encoder Implementation	4
1.2.2 Decoder Implementation.....	6
1.3 Accomplishments.....	8
1.4 Organization of Dissertation.....	9
CHAPTER 2 - Low-Density Parity-Check Codes.....	10
2.1 Encoding	10
2.1.1 Generic Encoding.....	11
2.1.2 Efficient Encoding	12
2.2 Decoding.....	14
2.2.1 Logarithmic Message Passing Algorithm	15
2.2.2 Minimum Sum Algorithm.....	17
2.2.3 Modified Minimum Sum Algorithm.....	18
2.2.4 Other Decoding Algorithms.....	18
CHAPTER 3 - Design Tools for FPGA and ASIC Implementation	19
3.1 Altera Quartus.....	19
3.1.1 Compilation.....	21
3.1.2 Simulations	21
3.1.3 Power Analysis	22
3.2 Cadence.....	23
3.2.1 RTL Compiler	25
3.2.2 Encounter	26

3.3 Matlab	27
CHAPTER 4 - Encoder Design for Randomly Generated Low-Density Parity-Check Codes	30
4.1 Encoder Design.....	30
4.1.1 Preprocessing	31
4.1.2 Hardware Implementation	34
4.1.2.1 Multi Clocked Inner Product	35
4.1.2.2 Single Clock Inner Product.....	36
4.2 Results.....	36
CHAPTER 5 - Encoder Design for Structured Low-Density Parity-Check Codes.....	41
5.1 Structured LDPC Codes.....	41
5.2 Design Methodology.....	43
5.3 Hardware Implementation	44
5.3.1 Computation of V	44
5.3.1.1 Vector-Vector Multiplication.....	45
5.3.1.2 Computation of e_p	45
5.3.2 Computation of Parity Bits	46
5.4 Results.....	46
CHAPTER 6 - Flexible Multi-Code Rate and Multi-Code Length Encoder for Structured Low-Density Parity-Check Codes.....	51
6.1 Design Methodology.....	51
6.2 Hardware Implementation	52
6.2.1 Storing Base Parity-Check Matrices for Different Rates of LDPC Codes	52
6.2.2. Parity-Check Matrix.....	55
6.2.2.1 Multiplication.....	57
6.2.2.2 Division.....	59
6.2.2.3 Computation of H_1	61
6.2.2.4 Latency for the Computation of H	62
6.2.3 Computation of e_p	64
6.2.4 Computation of V	66
6.2.5 Computation of Parity Bits	67
6.3 Results of the Flexible Structured Encoder Implemented on an FPGA	67

6.4 Implementation of a Flexible Multi-Code Rate and Multi-Code Length Structured Encoder on an ASIC.....	70
CHAPTER 7 - Decoder for Low-Density Parity-Check Codes.....	73
7.1 Study of LDPC Decoder Parameters	74
7.1.1 Decoding Algorithm	74
7.1.2 Decoder Architecture	75
7.1.3 Quantization.....	75
7.1.3.1 Quantization of φ	75
7.1.3.2 Quantization of Log-Likelihood Ratios	76
7.1.4 Maximum Number of Decoding Iterations	77
7.2 Design and Implementation of LDPC Decoder on FPGA.....	78
7.2.1 Quantization.....	78
7.2.1.1 Quantization of φ	78
7.2.1.2 Quantization of Log-Likelihood Ratios	79
7.2.1.3 Conversion of Log-Likelihood Ratios from One Form to Another Form of Representation.....	80
7.2.2 Initialization of Decoder Process	81
7.2.3 Check Node Processing	82
7.2.4 Variable Node Processing.....	85
7.2.5 End of Decoding Process	86
7.2.6 Decoder	87
7.3 Results.....	88
7.4 Optimization of Decoder Parameters.....	90
7.4.1 Erroneous Codewords	90
7.4.2 Decoder Delay	92
7.4.3 Decoder Energy	92
7.4.4 Optimization	95
CHAPTER 8 - Conclusion.....	101
8.1 Future Work	103
CHAPTER 9 - References	104
Appendix A - Design of a Convolutional Encoder in Verilog HDL	110

A.1 Convolutional Encoder	110
Appendix B - FPGA Implementation using Quartus	112
B.1 Creating a Project	112
B.2 Compilation of the Project	114
B.3 Timing Simulation	115
B.4 Power Analysis.....	119
Appendix C - ASIC Implementation using Cadence	122
C.1 Initial Setup	122
C.2 Synthesis of Verilog HDL Modules in RTL Compiler	123
C.3 Place and Route using Cadence Encounter	125
C.4 Verification of the Design	136
Appendix D - Quantization of Log-Likelihood Ratios in Decoder Implementation	141

List of Figures

Figure 1.1: Comparison of bit error probability of error correcting codes.	3
Figure 2.1: Parity-check matrix.	10
Figure 2.2: Tanner graph representation of parity-check matrix.	11
Figure 2.3: A parity-check matrix in equivalent lower triangular form.	12
Figure 2.4: Parity-check matrix in approximate lower triangular form, H_{pres} , and its division of sub-matrices.	12
Figure 2.5: Subgraph of Tanner graph showing message passing from variable node to check node.	14
Figure 2.6: Subgraph of Tanner graph showing message passing from check node to variable node.	14
Figure 3.1: Logic element architecture on the Stratix FPGA.	19
Figure 3.2: Design flow in Quartus.	20
Figure 3.3: Compilation report.	21
Figure 3.4: Timing simulations.	22
Figure 3.5: PowerPlay power analyzer design flow.	23
Figure 3.6: PowerPlay power analyzer summary.	24
Figure 3.7: ASIC design flow.	24
Figure 3.8: Design flow in RTL Compiler.	25
Figure 3.9: Design flow in Encounter.	26
Figure 4.1: Overview of the LDPC encoder.	31
Figure 4.2: Application of greedy algorithm A on H	32
Figure 4.3: Distribution of number of one's in each row of P_2 matrix for an irregular H of size 504×1008	34
Figure 4.4: Complete system of the generic encoder.	35
Figure 4.5: Circuit for multi clocked inner product (MCIP).	35
Figure 4.6: Circuit for single clocked inner product (SCIP).	36
Figure 5.1: Encoding process.	44
Figure 5.2: Overview of encoding process.	44

Figure 5.3: Logic elements vs. code lengths for different code rates.	48
Figure 5.4: Complete structured encoder system.....	48
Figure 6.1: Overview of the encoding process.	52
Figure 6.2: Computation of V using (a) row parallelization method and (b) column parallelization method.....	53
Figure 6.3: Storing base parity-check matrices, H_b , for different code rates.....	55
Figure 6.4: Finite state machine for the multiplication module.....	57
Figure 6.5: Hardware block diagram for the multiplication module.	58
Figure 6.6: Finite state machine for the division module.	59
Figure 6.7: Hardware block diagram for the division module.....	60
Figure 6.8: Computation of an element of H_1 , $h_{1(i,j)}$, from an element of H_{b1} , $h_{b1(i,j)}$	62
Figure 6.9: Computation of a column of H_1 from a column of H_{b1}	63
Figure 6.10: Computation of e_p	65
Figure 6.11: Computation of V	66
Figure 6.12: Complete system of the flexible multi-code rate and multi-code length structured LDPC encoder.....	69
Figure 6.13: Synthesized flexible multi-code rate and multi-code length LDPC structured encoder in Cadence RTL Compiler.	70
Figure 6.14: Layout view of the flexible multi-code rate and multi-code length LDPC structured encoder in Cadence Encounter.....	71
Figure 6.15: Layout view of the flexible multi-code rate and multi-code length LDPC structured encoder in Cadence ICFB.	72
Figure 7.1: BER vs. SNR performance using different decoding algorithms.	74
Figure 7.2: Quantization of φ	76
Figure 7.3: BER vs. SNR performance for different quantization levels of log-likelihood ratios.	77
Figure 7.4: BER vs. SNR for different values of maximum number of decoding iterations.....	78
Figure 7.5: Conversion of 2's complement to sign magnitude representation.....	81
Figure 7.6: Conversion of sign magnitude to 2's complement representation.....	81
Figure 7.7: Computation of magnitude of check node values.	83
Figure 7.8: Computation of sign of check node values.	83

Figure 7.9: Computation of check node values.	84
Figure 7.10: Computation of variable node values.	85
Figure 7.11: Design of LDPC decoder.	87
Figure 7.12: Complete decoder system.	89
Figure 7.13: Histogram of decoding iterations required by codewords for varying SNR.	91
Figure 7.14: Surface plot of f for varying SNR, $Iter_{Max}$ and weights.	98
Figure A.1: A 1/2 rate convolutional encoder with constraint length 7.	110
Figure B.1: The main Quartus II display.	112
Figure B.2: Creation of new project.	113
Figure B.3: Adding design files.	113
Figure B.4: Choose the device family and a specific device.	114
Figure B.5: Other EDA tools can be specified.	114
Figure B.6: Summary of the project settings.	115
Figure B.7: Flow summary of the compilation report.	115
Figure B.8: Creating vector waveform file.	116
Figure B.9: Waveform editor window.	116
Figure B.10: Insert node or bus dialog box.	117
Figure B.11: Selecting nodes to insert into the waveform editor.	117
Figure B.12: Simulator settings.	118
Figure B.13: Timing simulation report.	118
Figure B.14: Creating .saf file.	119
Figure B.15: PowerPlay power analyzer tool.	119
Figure B.16: Power settings.	120
Figure B.17: Add power input file.	120
Figure B.18: PowerPlay power analyzer summary.	121
Figure C.1: Synthesized convolutional encoder in RTL Cadence.	123
Figure C.2: Basic design import.	126
Figure C.3: Advanced design import.	126
Figure C.4: After importing the design.	127
Figure C.5: Specify floorplan.	127
Figure C.6: After floorPlan.	128

Figure C.7: Add rings.	128
Figure C.8: After adding rings.	129
Figure C.9: Add stripes.	129
Figure C.10: After adding stripes.	130
Figure C.11: Special route.	130
Figure C.12: After special route.	131
Figure C.13: Place.	131
Figure C.14: After placing cells.	132
Figure C.15: NanoRoute.	132
Figure C.16: After nanoRoute.	133
Figure C.17: Add filler.	133
Figure C.18: After adding fillers.	134
Figure C.19: Verify connectivity.	134
Figure C.20: Verify geometry.	135
Figure C.21: GDS export form.	135
Figure C.22: Stream in form.	136
Figure C.23: User-defined data form.	136
Figure C.24: Options form.	137
Figure C.25: DRC form.	137
Figure C.26: Import Verilog in.	138
Figure C.27: Schematic view.	139
Figure C.28: Extracted view.	139
Figure C.29: LVS.	140

List of Tables

Table 1.1: Coding schemes for different standards.	3
Table 2.1: Steps for computation of parity bits p_1 and p_2	13
Table 4.1: Density (number of one's) of H , P_1 and P_2 matrices.....	33
Table 4.2: Synthesis results of encoder implementation using MCIP on Stratix EP1S80F1508C5.	37
Table 4.3: Synthesis results of encoder implementation using SCIP on Stratix EP1S80F1508C5.	37
Table 4.4: Synthesis results of complete encoder system using MCIP and SCIP implemented on Stratix EP1S80F1508C5.	39
Table 4.5: Synthesis results of LDPC encoder designed by Lee.	39
Table 5.1: Synthesis results of structured encoder using LDPC codes defined in 802.16e.....	47
Table 5.2: Comparison of information data rate without I/O serialization of our proposed structured encoder with the encoder presented by Kim.....	49
Table 5.3: Comparison of coded data rate with I/O serialization of our proposed structured encoder with the encoder presented by Kim.	50
Table 6.1: Rate select values for different code rates.	56
Table 6.2: Length select values for different code lengths.	56
Table 6.3: Number of clock cycles required for computation of H_1 from H_{b1} for different code rates.	63
Table 6.4: Synthesis results of the flexible encoder for structured LDPC codes.	68
Table 6.5: Latency involved in computation of H for different code rates.....	68
Table 6.6: Synthesis results of flexible multi-code rate and multi-code length LDPC encoder in Cadence RTL Compiler.	71
Table 7.1: Quantization of φ	76
Table 7.2: Quantization of log-likelihood ratios.....	77
Table 7.3: Look up table for φ	79
Table 7.4: Quantization of log-likelihood ratios.....	80
Table 7.5: Synthesis results of the LDPC decoder.	88

Table 7.6: Synthesis results of LDPC decoder of code length 1024 and code rate 1/2 in Cadence RTL Compiler.	90
Table 7.7: Erroneous codewords, <i>errors</i> , for varying SNR and <i>Iter_{Max}</i>	92
Table 7.8: Decoder latency, <i>delay</i> , for varying <i>Iter_{Max}</i>	92
Table 7.9: PowerPlay power analysis report of the decoder of size 64 × 128.	93
Table 7.10: <i>Energy</i> (pJ) required for varying SNR and <i>Iter_{Max}</i>	94
Table 7.11: Weighing coefficients of <i>error</i> (α), <i>energy</i> , (β), and <i>delay</i> (γ).	96
Table 7.12: Minimum <i>f</i> corresponding to <i>Iter_{Max}</i> , SNR and weights.	97
Table D.1: Quantization of φ	141
Table D.2: Quantization of log-likelihood ratios.	142

Acknowledgements

I would like to first express my deepest appreciation for the technical guidance, invaluable suggestions and support given by my advisor, Dr. Don M. Gruenbacher. I would like to thank Dr. William B. Kuhn and Dr. Balasubramaniam Natarajan for serving on my committee and also for their guidance and suggestions during my doctoral program. I am grateful to Dr. Daniel A Andresen for serving as an outside committee member. My special thanks to Dr. Itzhak Ben-Itzhak for serving as an outside chairperson.

I would like to thank my parents and sister for their unconditional love, encouragement and support throughout my life. I am grateful to my husband, Satish, for his support, cooperation and being there for me whenever needed. I thank my in-laws for their support.

I thank all my friends Ali, Samer, Shilpa, Lutfu, Shekhar, Shama, Mandar, Gayathri, Murali, Seemanti and Jeet for their support and companionship during my stay at KSU. I owe my special thanks to Lakshmi, Prasad and their kids (Abhinav and Ananya) for welcoming me into their home. I would also like to thank Anupama, Praveen, Samatha and Harish for making my stay at KSU a pleasant one. I appreciate the faculty families Muthukrishnan's, Krishnaswamy's and Pahwa's for welcoming us (me and Satish) into their homes, helping and advising us whenever needed.

I would like to thank Sharon and Sam for helping me whenever needed. I thank Dr. Tom Pearson for giving me an opportunity to work at USDA as a research assistant. I am very grateful to Dr. Kameran Azadet for giving me an opportunity to work at LSI Corporation as an intern. Finally, I would like to thank Electrical and Computer Engineering department for providing partial support through teaching assistantship.

Dedication

To
Mom, Dad, Sister
and
Husband

CHAPTER 1 - Introduction

Channel coding theory began when Shannon applied probability theory to study the communication system. Shannon showed that for a given transmission rate less than or equal to channel capacity, the errors induced by the noisy channel can be reduced to a desired level by using a proper coding scheme [1]. Channel codes that can detect and correct the errors occurred during the transmission through a communication channel are called error correcting codes. Channel coding minimizes the effect of channel noise by using a channel encoder and decoder at the transmitter and receiver respectively. The channel encoder encodes the message bits by adding redundant bits to generate each codeword. The channel decoder in the receiver exploits the redundant bits in the received codeword and retrieves the actual message bits. Forward error correction (FEC) is a system of error control for data transmission. FEC codes detect and correct errors without requiring retransmission. Low-density parity-check codes (LDPC) are a type of FEC codes used for error detection and correction.

Low-density parity-check codes were invented by Gallager [2], [3]. LDPC codes have recently received much attention because of their efficient decoding algorithm, excellent error correcting capability and their performance close to the Shannon limit for large code lengths [4]. LDPC codes are proposed as an optional code in many IEEE standards. In [5], Europe's DVB standards group has selected LDPC codes due to their superior performance over Turbo codes for next generation digital satellite broadcasting. LDPC codes have already been verified and adopted by digital video broadcasting (DVB-S2) satellite broadcast and 10-Gbit Ethernet-over-copper system specifications [6]. LDPC codes are widely being considered as next-generation error correcting codes for many real applications such as telecommunications and magnetic storage.

The objective of this work is to develop a flexible hardware encoder and decoder for LDPC codes. The design methodologies used for the implementation of a LDPC encoder and decoder are flexible in terms parity-check matrix, code rate and code length. The following section presents the motivation behind this work. The current state of LDPC encoder and decoder implementations in hardware is presented in the next subsection. Finally the accomplishments of this work and organization of the thesis are illustrated.

1.1 Motivation

Future wireless systems need extremely fast and flexible architectures to support varying standards, algorithms and protocols with high data rates. Software radio is a widely proposed solution for these systems [7]. A software radio is a wireless communication device in which all of the signal processing is implemented in software. By simply downloading a new program, a software radio is able to interoperate with different wireless protocols, incorporate new services, and upgrade to new standards [8]. Cognitive radio (CR) is the next step in the evolution of software-defined radio (SDR). The cognitive radio concept was invented and presented by J. Mitola [9, 10]. It takes SDR's ability to adapt to changing communication protocols and frequency bands and adds a new dimension which is the ability to perceive the world around it and learn from experience [11, 12] and adapt to optimize the use of available resources.

The two primary objectives of the cognitive radio are to provide highly reliable communication whenever and wherever needed and to utilize the radio spectrum efficiently [13]. Cognitive radio is able to work in different frequency bands and various wireless channels and supports multimedia services such as voice, data and video [14]. Cognitive radio is a new paradigm in wireless communication that holds promise for new and better services to many markets, including public safety, military etc. Based on both current and previous channel characteristics, the radio would know what to do, where to go and how to make the operating changes without the user's intervention and without interfering with other communication equipment. Some of the radio's other cognitive abilities include determining its location, sensing spectrum use by neighboring devices, changing frequency, adjusting output power or even altering transmission parameters and characteristics [15-17].

Hence, there is a need to develop appropriate hardware which can be easily configured and adapted to varying coding parameters. The future of cognitive radio primarily depends on the availability of flexible and reliable hardware architectures. In this thesis, an attempt is made to develop a flexible and reliable architecture which would aid the future development of cognitive radio.

Any radio with the capacity to jump around the spectrum optimizing for power, range and required data rates, will, at the very least, require an extremely flexible RF front end [18]. The technical means to dynamically assign or utilize spectrum involves: (1) highly adaptive modulation and coding techniques (2) multidimensional/hybrid multiple access techniques (3) a

spectrum and resource aware MAC/link layer (4) flexible networking and (5) spectrum awareness and multilayer resource management [19]. Coding techniques used in different wireless standards are shown in Table 1.1. Convolutional codes, Reed-Solomon codes, Turbo codes, Low-density parity-check (LDPC) codes are some of the common error correcting codes currently being used in different standards and their bit error probability is shown in Figure 1.1 [20].

Table 1.1: Coding schemes for different standards.

Parameter	IEEE 802.11a	IEEE 802.11n	DVB-T	IEEE 802.16
Error correcting codes	Convolutional codes	Convolutional /LDPC codes	Reed-solomon codes, Convolutional codes	Reed-solomon-convolutional codes, LDPC codes (optional)
Net data rate (Mbps)	Up to 54	200	49.8-131.67	Excess of 120
Code rate	1/2, 2/3, 3/4	1/2, 2/3, 3/4, 5/6	1/2, 2/3, 3/4, 5/6, 7/8	1/2, 2/3, 3/4

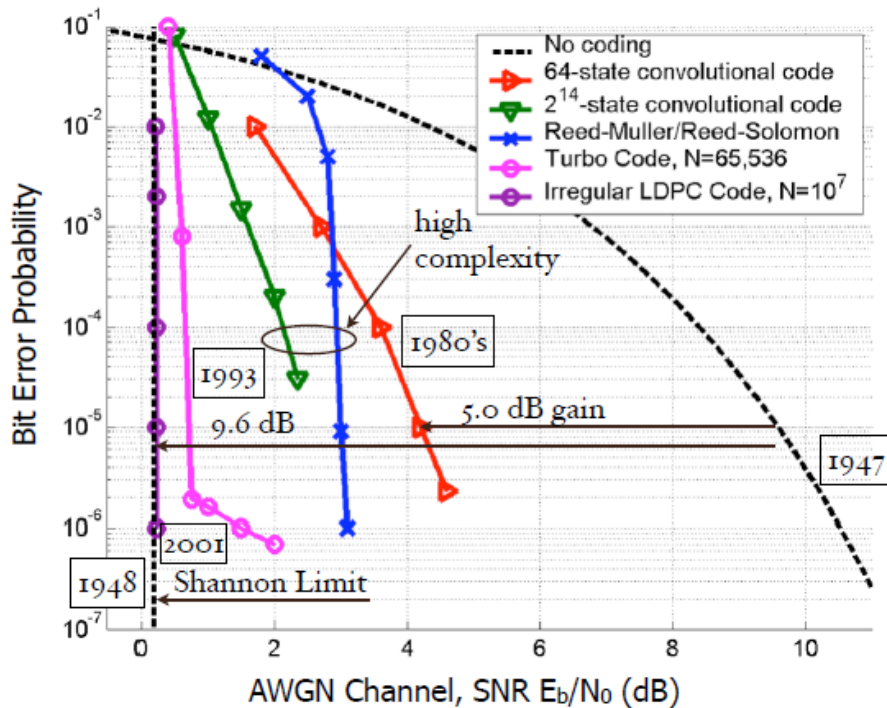


Figure 1.1: Comparison of bit error probability of error correcting codes [20].

Reviewing the work reported in this research area and industry, LDPC codes are found to be the leading error correcting codes. Most of the architectures for encoding and decoding of

LDPC codes are based on regular or structured LDPC codes. However in [21], it was shown that properly constructed irregular LDPC codes can approach the channel capacity more closely than regular LDPC codes. And also, most of the work in the literature shows that the LDPC encoder and decoder are implemented with fixed parameters such as fixed code rate and code length. But cognitive radio as explained earlier requires flexibility in both code rate and code length. Hence, there is a need for designing a flexible LDPC encoder/decoder. Also, high speed encoder and decoder are necessary as applications require more bandwidth. The focus of this work is to develop hardware for LDPC encoder and decoder that are flexible in terms of code rate and code length for a reconfigurable radio. The designs are applicable to both structured and any randomly generated regular and irregular LDPC codes.

1.2 Literature Review

In the following subsections, the current state of work published in the hardware implementations of both the encoder and decoder for LDPC codes are presented.

1.2.1 Encoder Implementation

The major drawback of LDPC codes is its high encoding complexity, in spite of the better performance and lower decoding complexity. The complexity is referred to the number of operations required per bit. A straightforward implementation of an LDPC encoder has complexity quadratic in the code length whereas turbo codes can be encoded in linear time. Even though LDPC codes are difficult to implement due to high encoder complexity, recent developments have led to more efficient encoder structures which are typically limited in their encoding rates.

A variety of encoder architectures have been presented in the past. Richardson showed that the encoding complexity can be reduced from $O(n^2)$ to either linear or quadratic. For example, a $(3, k)$ -regular code of length n requires about $0.017^2 n^2 + O(n)$ operations [22]. The parity-check matrix is initially transformed into an approximate lower triangular form. Then encoding is performed on the approximate lower triangular form of the parity-check matrix using the greedy algorithm.

Zhang et al. [23, 24] proposed a systematic efficient encoding scheme by effectively exploiting the sparseness of its $(3, k)$ -regular LDPC codes. A design approach is presented by Zhong in [25] for a LDPC system hardware implementation by jointly conceiving irregular code

construction and VLSI implementations. The encoding algorithms in [23-25] are similar to that of Richardson's greedy algorithm [22], except that these algorithms do not contain any back-substitution operations because of the structural property of their parity-check matrices. To take the advantage of the parity-check matrix structure in [23-24], the parity-check matrix is transformed into an approximate upper triangular matrix rather than lower triangular form.

In [26], Echard introduced an ensemble of quasi-regular low-density parity-check codes called as π rotation LDPC codes. In [27], Kim presented high-performance parallel implementations of an encoder and decoder for a parallel concatenated parity-check class of LDPC codes. In [28], Miles implemented a radiation tolerant encoder in 0.25μ CMOS based on a novel method for deriving regular quasi-cyclic LDPC codes. These encoding methodologies assume a particular structure for the parity-check matrix such as regular, quasi-regular and parallel concatenated LDPC codes. Hence the encoding methodologies developed are applicable to those particular LDPC codes and cannot be used for other structured or non-structured LDPC codes.

In [29-33], a hardware design of an efficient LDPC encoder was described based on the method proposed by Richardson and Urbanke in [22]. In [29], the encoder for code length of 2000 bits and rate 1/2 has a coded data rate of 45 Mbps. The coded data rate can be increased to 410 Mbps by implementing 16 instances of the encoder on the same device. In [30-32] an implementation of a real-time programmable irregular LDPC encoder as specified in the IEEE P802.16E/D7 standard was presented. The encoder is implemented on a reconfigurable instruction cell architecture and has a data rate from 10-19 Mbps. The design presented has a maximum data rate of 78 Mbps with the use of pipelining and multiple cores. In [33], a LDPC encoder is implemented for structured LDPC codes as defined in both IEEE 802.16e and IEEE 802.11n. An architecture for a structured LDPC encoder has been presented that supports IEEE 802.11n [34].

An encoder and decoder for LDPC codes defined in IEEE 802.16e are developed and their coded data rate is dependent on the clock frequency at which they run [35]. The basic encoder and decoder area is 20 K and 125 K gates respectively. The LDPC encoder core in [36] provides a complete encoding solution for the codes defined in IEEE 802.16e. A major feature of the core is that it has an extremely low latency and the encoded packet is available at the output in seven clock cycles. The coded data rate is equal to the clock rate of the encoder.

Most of the encoder methodologies presented above assume some kind of structure for the parity-check matrix. Also the encoder design parameters, the code rate and the code length, are fixed. Hence there is a need to design a LDPC encoder that is flexible in terms of parity-check matrix, code rate and code length for cognitive radio.

1.2.2 Decoder Implementation

Several algorithms are proposed for LDPC decoding. LDPC codes which can approach Shannon's limit by using an iterative decoding algorithm called belief propagation. This algorithm is also called as sum-product algorithm or message passing algorithm [37]. By using log-likelihood ratios (LLR) as messages (logarithmic message passing algorithm) the hardware implementation has become much easier when compared to the message passing algorithm. The implementation complexity is further reduced by simplifying the process for updating check nodes, which is the most extensive part of the message passing algorithm. This algorithm is called the min-sum algorithm [38]. Later on, several algorithms were introduced by modifying the min-sum algorithm [39-41] to bridge the gap in the performance between the min-sum and message passing algorithms.

A LDPC decoder can be implemented using serial, parallel or partially parallel architectures. In [42], a fully-parallel irregular LDPC decoder is synthesized using 0.18 μm CMOS technology and achieves a data rate of 1 Gbps for code length of 648 and rate 5/6. In [43], a 1024-b, rate-1/2 LDPC decoder is implemented using a parallel architecture. The design achieves a coded data rate of 1 Gbps. This performance is achieved by exploiting the inherent parallelism and rapid convergence of the message passing decoding algorithm. In [44], a 1/2 rate, 2048 codeword, (3, 6) regular LDPC code has been analyzed. The data rate and complexity analysis is performed for the VLSI implementation of an LDPC decoder using both fully and partially parallel architectures. In [45], the decoder is designed using a serial architecture and has a moderate data rate. The decoding algorithm proposed in their paper belongs to the class of min-sum with a correction factor. The correction factor is updated from the log-likelihood ratio values. The decoder is peripherally connected to the embedded PowerPC processor of a Xilinx Virtex-II Pro FPGA and is managed by the processor. This method of hardware/software implementation provides the maximum flexibility for the development and rapid prototyping of the hardware-based simulator system.

In [46], a parameterized decoder that supports the LDPC code in the IEEE 802.16e standard, is presented which requires code rates of $1/2$, $2/3$ and $3/4$, with block sizes varying from 576 to 2304. The decoder is synthesized with Texas Instruments' 90 nm ASIC process technology, with a target operation frequency of 100 MHz, 15 decoding iterations, and the maximum data rate is up to 256 Mbps. Similar flexible multi-rate multi-code length structured LDPC decoder is designed in [47-49]. The IP core in [50] provides a hardware efficient implementation of an LDPC decoder for IEEE 802.16e. The design covers the entire IEEE802.16e LDPC specification and implements an early stop detection function. The data rate of the decoder is dependent on the clock frequency. The decoded throughput is up to 168 Mbps for a Virtex-4 with a -12 speed grade.

In [51] the energy consumption of a quantized LDPC decoder is computed. It is shown that the energy consumption of the decoder increases exponentially with the number of quantization bits. A new architecture is proposed in [52] which reduces memory access, hence power consumption, without sacrificing performance. It is shown that through an interconnect-driven code design approach, coupled with a dynamic addressing scheme and an optimized version of the BCJR algorithm for computing reliabilities, power savings of up to 85.64% can be achieved [53]. A low-power real-time decoder that provides constant-time processing of each frame using dynamic voltage and frequency scaling is presented in [54]. VLSI architectures for low-density parity-check decoders amenable to low-voltage and low-power operation are investigated in [55]. In this paper, highly-parallel decoder architecture with low routing overhead is described. Dynamic power is reduced by using an efficient method to detect early convergence of the iterative decoder and terminate the computations.

The performance of the LDPC decoder depends on various factors such as the decoding algorithm, the architecture, the quantization of log-likelihood ratios and the maximum number of decoding iterations. The maximum number of decoding iterations used for the decoding process determines the data rate and latency of the LDPC decoder. After performing maximum number of decoding iterations, the codeword is then estimated. Most of the decoders presented above do not estimate the codeword and check its validity after each iteration. In order to save decoder power consumption and to decrease the latency, a decoder design that verifies the codeword after each iteration and stops the decoding process when the estimated codeword is correct is needed. In [55], the parity of the normal variable-to-check messages is checked after each iteration. If the

parity check is satisfied then the codeword is estimated at the beginning of the next iteration and the decoding process is stopped. In [42], the codeword is estimated after every iteration but it is validated in the next iteration. These two decoder designs in [42] and [55], take an additional iteration to stop the decoding process after the decoder decoded the correct codeword. An attempt is made to find an optimum value for the maximum number of decoding iterations for a given parity-check matrix and SNR by minimizing the error, delay and energy.

1.3 Accomplishments

In this section the important contributions of this thesis are presented. As shown in the previous sections, current hardware implementations of LDPC encoders and decoders use either cyclic, quasi-cyclic or some regular pattern in the parity-check matrix, H . In this work, both an LDPC encoder and decoder are developed that are flexible in terms of parity-check matrix, code rate and code length. Here is a list of the significant contributions of this work.

1. A generic encoder is designed and tested for any randomly generated LDPC codes.

Two new encoder designs were developed that achieve much higher data rates while requiring more area for implementation. The designs developed can be used for both structured and any randomly generated regular and irregular parity-check matrices as they are independent of the structure of the LDPC codes.

2. An encoder for structured LDPC codes is designed and tested.

An encoder architecture that adheres to the structured LDPC codes defined in the IEEE 802.16e standard was developed. The design methodology with minor modifications can be used for other similar structured LDPC codes defined in different standards.

3. A flexible multi-code rate and multi-code length LDPC Encoder is designed and tested

A flexible encoder design that accommodates different code lengths and code rates has been developed. This design methodology does not require re-synthesis of the Verilog code to change the encoder parameters (code length and code rate). This design methodology developed with minor modifications can be used for other similar structured LDPC codes.

4. A LDPC decoder for randomly generated LDPC codes is designed and tested

A decoder design methodology that is independent of the structure in the LDPC codes has been developed and implemented. Hence it is applicable to both structured and any randomly generated regular and irregular LDPC codes. This design validates the estimated codeword after every iteration and stops the decoding process when the correct codeword is estimated which would save the power consumption.

5. Optimum value of the maximum number of decoding iterations

The maximum number of decoding iterations plays a major role in determining the decoder performance and latency. A procedure/method is presented to find an optimum maximum number of decoding iterations for a given parity-check matrix and SNR.

1.4 Organization of Dissertation

A brief introduction on LDPC codes is presented in chapter 2. In this chapter, various encoding and decoding algorithms are presented. In chapter 3, the design tools Altera Quartus, Cadence and Matlab, used in the implementation of LDPC encoder and decoder are presented. The two design methodologies used for the encoder design implementation for randomly generated LDPC codes along with the results are presented in chapter 4. The encoder design methodology, the implementation and the results for a structured LDPC codes are discussed in chapter 5. In chapter 6, the design methodology, hardware implementation and the results for a flexible multi-code rate and multi-code length encoder for structured LDPC codes are presented. In chapter 7, the decoder design methodology, hardware implementation and results are presented. For a given parity-check matrix and SNR, an optimum maximum number of decoding iterations are evaluated.

CHAPTER 2 - Low-Density Parity-Check Codes

In this chapter, low-density parity-check codes are introduced and the details of encoding and decoding algorithms are presented. LDPC codes were invented by R. G. Gallager [2][3]. LDPC codes are linear block codes specified by a parity-check matrix, H , which is sparse. There are two types of LDPC codes, regular and irregular LDPC codes. Regular LDPC codes are defined by parity-check matrices with a fixed number of non-zero entries (usually 1's) in each row and column known as row weight, w_r , and column weight, w_c , respectively. Irregular LDPC codes are defined by parity-check matrices having a variable number of non-zero entries in each row and column. In general, irregular LDPC codes have better error-correcting performance than that of regular LDPC codes. In this work, Mackay's parity-check matrices [56] for both regular and irregular LDPC codes are used. LDPC codes can be represented using a bipartite graph, also called as Tanner graph, where one set of nodes represents the codeword, also known as variable nodes, and the other set of nodes, called check nodes, represents the parity check constraints. Messages are passed between check and variable nodes along the edges, $L(r_{ji})$ and $L(q_{ij})$. Each edge in the Tanner graph corresponds to a '1' in H . An example of a 4×8 rate 1/2 parity-check matrix is shown in Figure 2.1. Each row of the parity-check matrix represents a check node and each column represents a variable node. Check node ' j ' is connected to variable node ' i ' if the corresponding element h_{ji} of H is 1. The Tanner graph representation of the parity-check matrix in Figure 2.1 is shown in Figure 2.2. In this work, parity-check matrices that are binary is considered.

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

Figure 2.1: Parity-check matrix.

2.1 Encoding

Encoding of LDPC codes uses the following property

$$Hx^T = 0^T, \tag{2.1}$$

where vector x represents the codeword, H is the parity-check matrix, and 0 is a zero vector. The codeword x consists of information bits, s , and parity bits, p . Parity bits are computed from the

information bits. Two of the common encoding methods are presented in the following subsections.

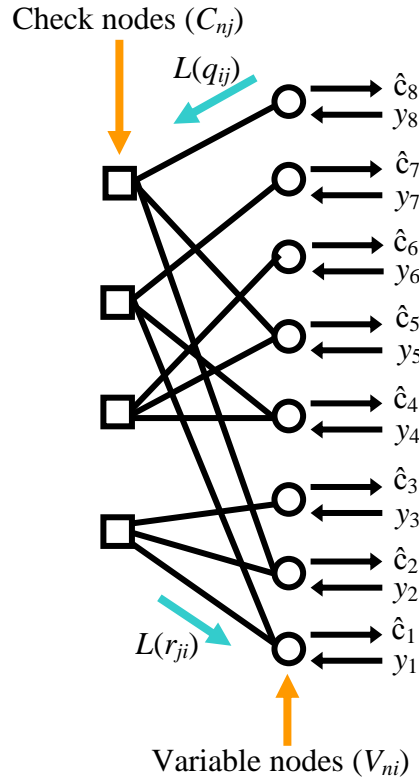


Figure 2.2: Tanner graph representation of parity-check matrix.

2.1.1 Generic Encoding

Let $H = [H_1 \ H_2]$ be the $m \times n$ parity-check matrix with sub-matrices H_1 and H_2 having the dimensions $m \times k$ and $m \times m$, respectively. For the remainder of this thesis, these dimensions are not explicitly designated. The most straight forward encoder implementation requires three steps. In the first step, the parity-check matrix, H , is transformed to an equivalent lower triangular form as shown in Figure 2.3. The second step is to take the codeword, x , and split it into its k information bits, s , and its m parity bits, p . *i.e.*, $x = [s \ p]$. In the third step, the parity bits p are obtained by solving Equation 2.1:

$$Hx^T = 0^T,$$

$$[H_{1_{m \times k}} \quad H_{2_{m \times m}}] \begin{bmatrix} s_{k \times 1} \\ p_{m \times 1} \end{bmatrix} = 0_{m \times 1}^T,$$

$$H_1 s + H_2 p = 0 \text{ or } p = H_2^{-1} H_1 s. \tag{2.2}$$

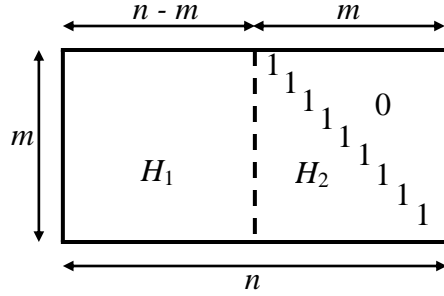


Figure 2.3: A parity-check matrix in equivalent lower triangular form.

In [29], it is stated that transforming the parity-check matrix into lower triangular form using Gaussian elimination requires about $O(n^3)$ operations. Since the transformed parity-check matrix is no longer sparse, the actual encoding requires $O(n^2)$ operations. More precisely the actual encoding requires $n^2 \left(\frac{r(1-r)}{2} \right)$ XOR operations where r is the code rate.

2.1.2 Efficient Encoding

Richardson and Urbanke [22] showed that linear time encoding is achievable through careful linear manipulation of LDPC codes. Using row and column permutations only, the parity-check matrix is transformed into an approximate lower triangular form, H_{pre} , as shown in Figure 2.4. The parity-check matrix is still sparse and T is in lower triangular form with ones along the diagonal. The gap, g , is made as small as possible because the encoding complexity is upper-bounded by $n + g^2$ [22].

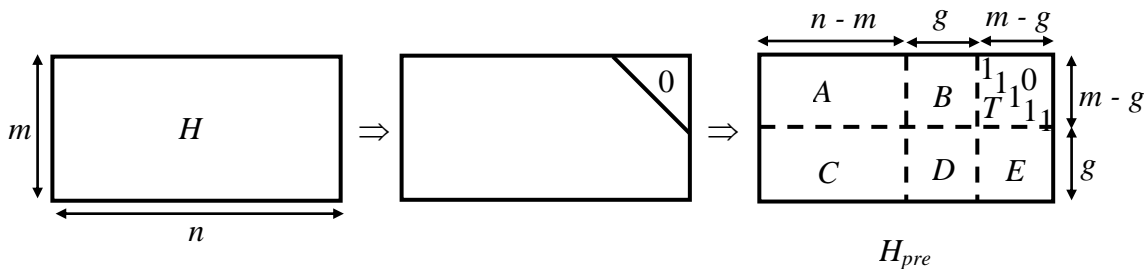


Figure 2.4: Parity-check matrix in approximate lower triangular form, H_{pre} , and its division of sub-matrices.

The encoding procedure is as follows: The codeword x is given by $x = [s \ p_1 \ p_2]$, where s are information bits and p_1 and p_2 are parity bits of length $n-m$, g and $m-g$, respectively. Equation

2.1 can also be represented as $H_{pre}x^T = 0^T$ and is solved to compute the parity bits. This expression is pre-multiplied by $\begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix}$ to obtain

$$\begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix} \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix} \begin{bmatrix} s \\ p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \quad (2.3)$$

For binary parity-check matrices, Equation 2.3 can then be separated into two equations as shown below

$$As + Bp_1 + Tp_2 = 0, \text{ and} \quad (2.4)$$

$$(ET^{-1}A + C)s + (ET^{-1}B + D)p_1 = 0. \quad (2.5)$$

Let $\phi = ET^{-1}B + D$, and assume that ϕ is nonsingular, the parity bits are given by

$$p_1 = \phi^{-1}(ET^{-1}A + C)s, \text{ and} \quad (2.6)$$

$$p_2 = T^{-1}(As + Bp_1). \quad (2.7)$$

The steps used to compute the parity bits are summarized [22] in Table 2.1.

Table 2.1: Steps for computation of parity bits p_1 and p_2 .

Step No.	Computation of p_1	Computation of p_2
1	As	As
2	$T^{-1}As$	Bp_1
3	$ET^{-1}As$	$As + Bp_1$
4	$ET^{-1}As + Cs$	$T^{-1}(As + Bp_1)$
5	$\phi^{-1}(ET^{-1}As + Cs)$	

In [22], it is found that by using this method, the encoding complexity is either linear or quadratic but quite manageable. For example, a (3, k)-regular code of length n requires about $0.017^2n^2 + O(n)$ operations. The complexity of the encoder is still manageable for large n since the 0.017^2n^2 is a very small number. The encoding complexity for all optimized irregular LDPC codes is linear because the expected gap, g , is actually of the order less than \sqrt{n} , and the required amount of preprocessing is of order at most $n^{3/2}$.

2.2 Decoding

Message passing [37] is an iterative algorithm commonly used in decoding LDPC codes. Each iteration of message passing consists of updating outgoing messages from both variable and check nodes. In one half of the iteration, each variable node, V_{ni} , passes all its information to each of the connected check nodes, C_{nj} , excluding the information the receiving check node already possesses. Consider the sub-graph of the Tanner graph shown in Figure 2.2 corresponding to the first column of H and is shown in Figure 2.5. An example of message passing between V_{n1} to C_{n3} is shown in Figure 2.5. The check node C_{n3} receives information from the channel, y_1 , and the extrinsic information node V_{n1} received from check nodes C_{n1} in the previous half iteration. In the other half iteration, the information is passed from check node to variable nodes excluding the information the receiving variable node already possesses. Figure 2.6 shows the sub-graph of the Tanner graph in Figure 2.2 corresponding to the first row of H .

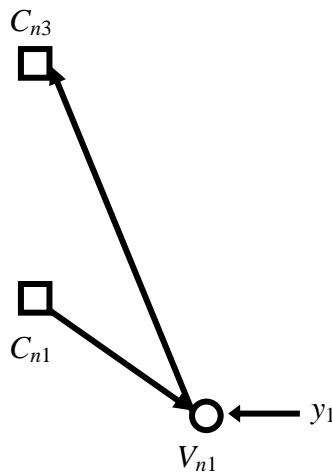


Figure 2.5: Subgraph of Tanner graph showing message passing from variable node to check node.

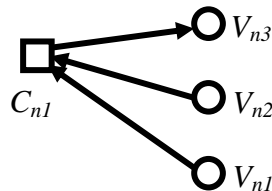


Figure 2.6: Subgraph of Tanner graph showing message passing from check node to variable node.

An example for passing the information from C_{n1} to V_{n3} is shown in Figure 2.6. The variable node V_{n3} receives the information from C_{n1} which it received from variable nodes V_{n1} and V_{n2} in the previous half iteration. After each iteration, the decoder checks if the estimated codeword satisfies Equation 2.1. If the decoder finds the correct codeword then the process is stopped. If not the process of decoding continues until the estimated codeword satisfies Equation 2.1 or reaches the maximum number of decoding iterations.

Using this message passing algorithm, LDPC codes can be efficiently decoded. This message passing algorithm is also known as sum-product algorithm. Since the direct implementation of the message passing algorithm will result in high hardware complexity due to a large number of multiplications, a logarithmic message passing algorithm is used to reduce complexity. The logarithmic message passing algorithm allows all of the multiplications to be converted into additions, making it more easily implemented in hardware. In fact, both message passing and logarithmic message passing decoding algorithms realize the same decoding rule. The summary of the logarithmic message passing algorithm is presented in the following subsection:

2.2.1 Logarithmic Message Passing Algorithm

Before presenting the summary of the logarithmic message passing algorithm, an overview of the notation used is presented below:

- R_j : The set of column locations of the 1's in the j^{th} row of H .
- $R_{j|i}$: The set of column locations of the 1's in the j^{th} row of H , excluding location i .
- C_i : The set of row locations of the 1's in the i^{th} column of H .
- $C_{i|j}$: The set of row locations of the 1's in the i^{th} column of H , excluding location j .
- y : Received codeword corresponding to the transmitted codeword x .
- \hat{c} : Estimated codeword.
- P_i : $\Pr(c_i = 1|y_i)$.
- $b \in \{0,1\}$.
- $q_{ij}(b)$: probability that $c_i = b$ given the information from all neighboring check nodes, except check node at position j .
- $r_{ji}(b)$: probability that j^{th} check Equation being satisfied given $c_i = b$ and information from all variable nodes except from the variable node at location i .

- $Q_{ij}(b)$: probability that $c_i = b$ given the information from all the check nodes.
- $L(c_i) = \log \left(\frac{\text{pr}(c_i = 0 | y_i)}{\text{pr}(c_i = 1 | y_i)} \right)$.
- $L(r_{ji}) = \log \left(\frac{r_{ji}(0)}{r_{ji}(1)} \right)$.
- $L(q_{ij}) = \log \left(\frac{q_{ij}(0)}{q_{ij}(1)} \right)$.
- $L(Q_i) = \log \left(\frac{Q_i(0)}{Q_i(1)} \right)$.

The steps for the logarithmic message passing decoding algorithm are as follows:

Step 1: The messages originating from variable nodes, $L(q_{ij})$, as shown in Figure 2.2 are initialized using

$$L(q_{ij}) = L(c_i) = \frac{2y_i}{\sigma^2}, \quad (2.8)$$

where y_i is the received code word and σ^2 is variance of the channel noise.

Step 2: The messages originating from check nodes, $L(r_{ji})$, as shown in Figure 2.2 are computed from $L(q_{ij})$

$$L(r_{ji}) = \prod_{i \in R_{j \setminus i}} \alpha_{ij} \cdot \varphi \left(\sum_{i \in R_{j \setminus i}} \varphi(\beta_{ij}) \right) \quad (2.9)$$

where $\alpha_{ij} = \text{sign}(L(q_{ij}))$, $\beta_{ij} = |L(q_{ij})|$ and $\varphi(z) = -\log(\tanh(z/2)) = \log \frac{e^z + 1}{e^z - 1}$. (2.10)

Step 3: $L(q_{ij}) = L(c_i) + \sum_{j \in C_{i \setminus j}} L(r_{ji})$. (2.11)

Step 4: $L(Q_i) = L(c_i) + \sum_{j \in C_i} L(r_{ji})$. (2.12)

Step 5: for $\forall i$, the codeword is estimated from $L(Q)$

$$\hat{c}_i = \begin{cases} 1 & \text{if } L(Q_i) < 0 \\ 0 & \text{if } L(Q_i) \geq 0 \end{cases}. \quad (2.13)$$

Step 6: The decoding process is stopped

$$\text{if } \hat{c}H^T = 0 \quad (2.14)$$

or the number of decoding iterations = maximum number of decoding iterations

else

repeat starting from step 2.

2.2.2 Minimum Sum Algorithm

The minimum sum algorithm is also called as Min-Sum algorithm and is essentially the same as the logarithmic message passing algorithm. The Min-Sum algorithm follows the same exact steps as that of logarithmic message passing algorithm except for step 2 which is modified. The following function

$$\varphi\left(\sum_{i \in R_{j|i}} \varphi(\beta_{ij})\right) \quad (2.15)$$

is approximated by the simple expression given below

$$\left(\min_{i \in R_{j|i}} \beta_{ij}\right) \quad (2.16)$$

i.e., the minimum value of β_{ij} . This substitution is due to the fact that $\varphi(z)$ is maximum when z is minimum and also $\varphi(\varphi(z)) = z$. Therefore,

$$\varphi\left(\sum_{i \in R_{j|i}} \varphi(\beta_{ij})\right) \cong \varphi\left(\varphi\left(\min_{i \in R_{j|i}} \beta_{ij}\right)\right) = \min_{i \in R_{j|i}} \beta_{ij}, \quad (2.17)$$

and the resulting step 2 becomes

$$\text{Step 2: } L(r_{ji}) = \left(\prod_{i \in R_{j|i}} \alpha_{ij}\right) \cdot \left(\min_{i \in R_{j|i}} \beta_{ij}\right) \quad (2.18)$$

where $\alpha_{ij} = \text{sign}(L(q_{ij}))$ and $\beta_{ij} = |L(q_{ij})|$.

Min-Sum algorithm simplifies the check node computation because there is no need to compute φ of the variable node values. Using Min-Sum algorithm may reduce the chip area for the implementation when compared to logarithmic message passing algorithm because φ which is typically implemented using look up table (LUT) in hardware is no longer required.

2.2.3 Modified Minimum Sum Algorithm

The modified minimum sum algorithm is similar to the minimum sum algorithm except for a small modification in step 2 of the Min-Sum algorithm procedure. The bit error rate performance of the decoder is degraded due to the approximation shown in Equation 2.17. To improve the decoding performance, the step 2 of the Min-Sum algorithm is again modified as shown below and is called as modified minimum sum algorithm.

$$\text{Step 2: } L(r_{ji}) = \left(\prod_{i \in R_{ji}} \alpha_{ij} \right) \cdot \left(\min_{i \in R_{ji}} \beta_{ij} - k \right) \quad (2.19)$$

where $\alpha_{ij} = \text{sign}(L(q_{ij}))$, $\beta_{ij} = |L(q_{ij})|$ and k is a constant value.

2.2.4 Other Decoding Algorithms

Bit flipping and layered decoding algorithms are some of the other LDPC decoding algorithms. The BER performance of these algorithms is inferior to the performance of the message passing algorithm. A brief summary of these algorithms are presented below.

Gallager's bit flipping algorithm is used for decoding binary symmetric channel [20]. As shown in Figure 2.2, there are two sets of nodes: check and variable nodes. For a received codeword, parity check is performed on each check node. For each variable node, the check nodes that are connected to this variable node and failed the parity check constraints are counted. The codeword bit associated with the variable node that has the largest number of failed parity checks is flipped. This process is repeated until all the parity checks are satisfied or a stopping condition is reached.

The layered decoding algorithm is a variation of the standard message passing algorithm [57]. The parity-check matrix consists of shifted identity sub matrices that are concatenated in horizontal layers. The message passing algorithm is performed on each horizontal layer and the updated a posteriori probability messages are passed between the horizontal layers [46]. Because of this optimized message scheduling the algorithm convergence rate is doubled [58].

CHAPTER 3 - Design Tools for FPGA and ASIC Implementation

In this chapter, the design tools used to accomplish this work are introduced. Quartus, Cadence and Matlab are used for the implementation of both the encoder and decoder of LDPC codes. The encoder and decoder of LDPC codes are implemented on field-programmable gate arrays (FPGA) using Altera Quartus. A flexible multi-code rate and multi-code length encoder for structured LDPC codes is also implemented on an application specific integrated chip (ASIC) using Cadence. Matlab is used to analyze, simulate, preprocess and generate Verilog hardware description language (HDL) modules for all of the encoder and decoder designs.

3.1 Altera Quartus

A FPGA is a reprogrammable integrated circuit which is usually designed using a HDL or schematic entry. For larger designs, using a HDL is easy and more appropriate. A FPGA can be typically used to implement any logical function that an ASIC could perform. The ability to update/modify the functionality of the design is a huge advantage in a FPGA when compared to an ASIC. FPGAs contain programmable logic components called logic array blocks (LABs). In the Altera Stratix FPGA device the logic array consists of LABs, with 10 logic elements (*LEs*) in each LAB. An *LE* is the smallest unit of logic providing efficient implementation of user logic functions. The *LE* provides advanced features with efficient logic utilization. Each *LE* contains a four-input look up table (LUT), which is the function generator that can implement any function of four variables as shown in Figure 3.1 [59].

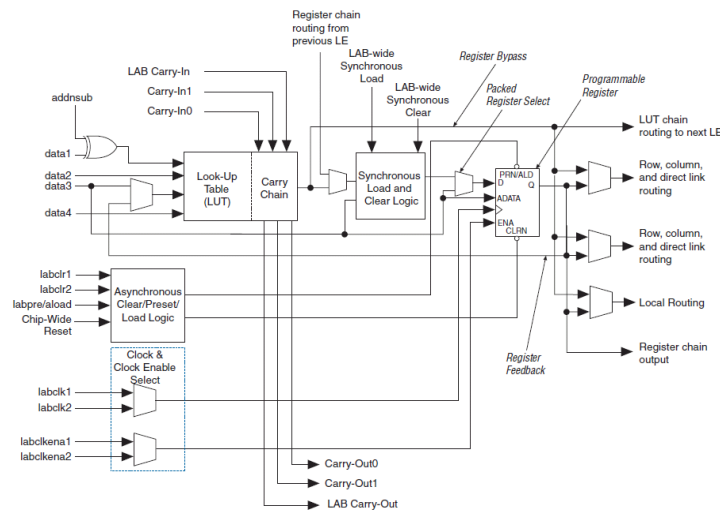


Figure 3.1: Logic element architecture on the Stratix FPGA [59].

The flow for the implementation of a design in Quartus II is shown in Figure 3.2 [60]. The desired circuit is specified by using a HDL such as Verilog HDL or VHDL. In this work, Verilog HDL was used exclusively. This design is synthesized into a circuit that consists of logic elements and memory blocks provided in the FPGA. The synthesized circuit is tested to verify its functional correctness. When checking functional correctness, simulation timing issues are not considered. The fitter tool determines the best placement of the synthesized *LEs* into the targeted FPGA device. It also chooses routing wires in the chip to make the required connections between the specific *LEs* being utilized.

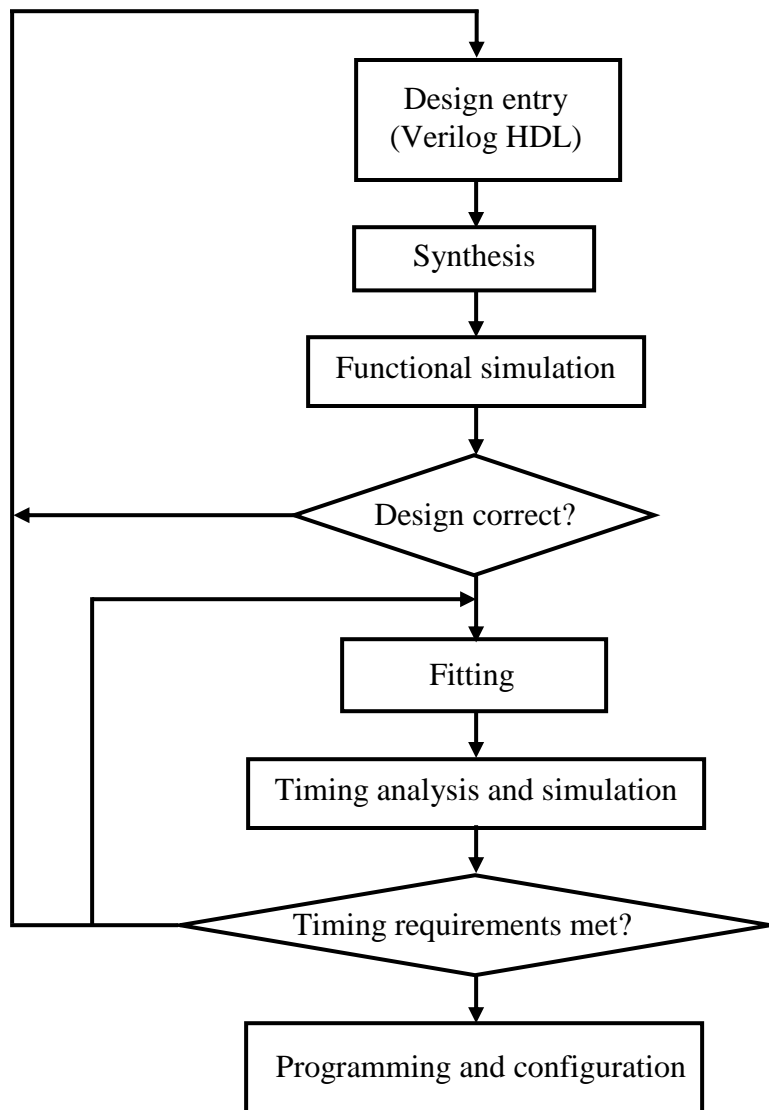


Figure 3.2: Design flow in Quartus.

The next step is timing analysis, during which propagation delays along the various paths in the fitted circuit are analyzed to provide an indication of the expected performance of the

circuit. The fitted circuit is tested to verify both its functional correctness and timing. The designed circuit is implemented in a physical FPGA chip by programming the configuration switches that configure the *LEs* and establish the required wiring connections. The compilation, simulation and power analysis on a design in Quartus are presented in the following sections.

3.1.1 Compilation

The Verilog HDL code is processed by several Quartus II tools that analyze the code, synthesize the circuit, and generate an implementation for the target FPGA chip. These tools are controlled by the application program called the compiler. Once the design is created in Verilog HDL, it needs to be compiled in Quartus. Compilation converts the design into a bitstream that can be downloaded into FPGA. The most important output of compilation is a SRAM Object File (.sof), which is used to program the device. The compilation also generates other report files such as timing, area, etc., that provide information about the code as it compiles. Figure 3.3 is an example of the compilation report.

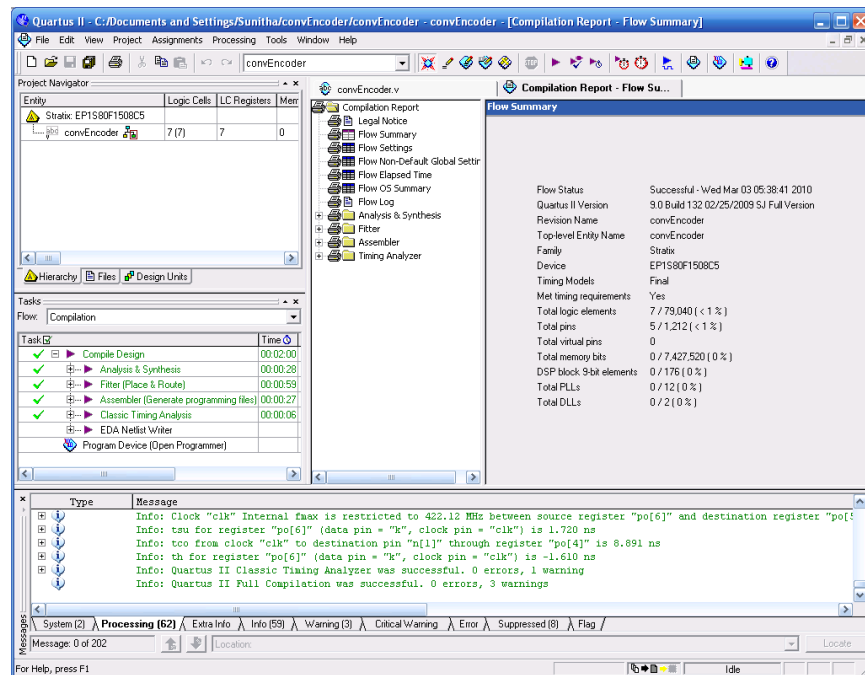


Figure 3.3: Compilation report.

3.1.2 Simulations

A designed circuit can be simulated in two ways: functional and timing simulations [61]. Functional simulations are used to verify the functional correctness of the designed circuit and it

is assumed that the logic elements and interconnection wires have zero propagation delays of the signals. This takes much less time, because the simulation can be performed simply by using the logic expressions that define the circuit. In timing simulations, all propagation delays are taken into account and thus exhibiting the actual behavior of the design when implemented on the FPGA device. In this work timing simulations are performed on the encoder/decoder designs compiled in Quartus. The encoded and decoded codeword obtained from timing simulations are compared with the codeword obtained using Matlab for verification. Figure 3.4 shows an example of timing simulations.

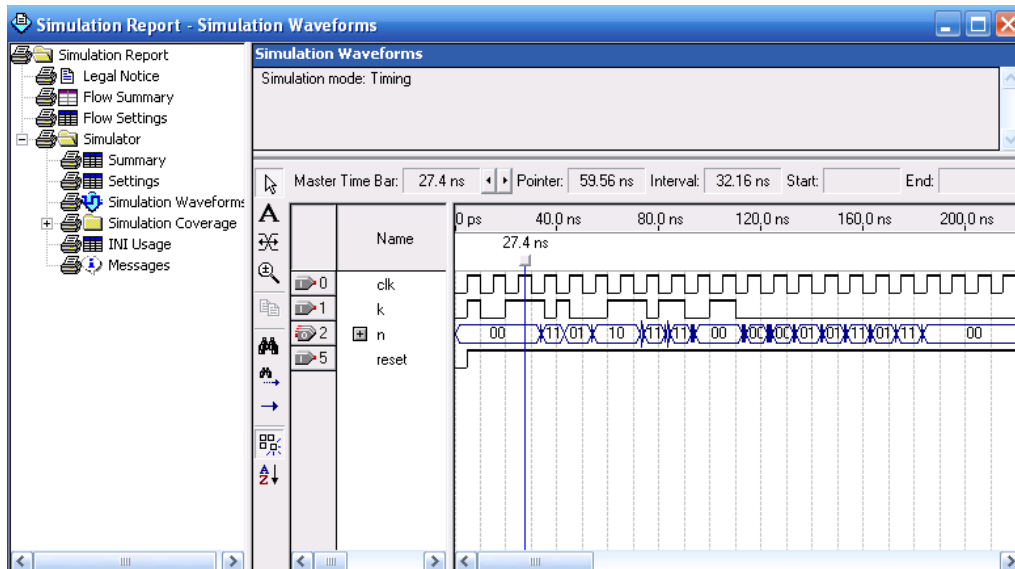


Figure 3.4: Timing simulations.

3.1.3 Power Analysis

Power plays an important design consideration as the designs grow larger and process technology continues to shrink [62]. Power consumed by the design compiled in Quartus can be analyzed using the PowerPlay power analysis tool. There are two PowerPlay power analysis tools: PowerPlay early estimator spreadsheet and PowerPlay power analyzer. PowerPlay early estimator spreadsheet can be used during early design stages and gives a rough estimate of the power consumption. The PowerPlay power analyzer tool offers improved accuracy over the PowerPlay estimator spreadsheet since it examines actual device resource usage, place and route information and information on activity rates of all signals in response to a specific input stimulus. Its accuracy is further improved by adding realistic timing simulation vectors.

PowerPlay power analyzer tool provides both static and dynamic power consumption estimates. Static power is defined as the power consumed regardless of signal/data activity. Dynamic power is the additional power consumed due to data switching activity or toggling. The design flow of PowerPlay power analyzer is shown in Figure 3.5. The PowerPlay power analyzer requires the design to be synthesized and fitted to the target device. The PowerPlay power analyzer directly reads the waveforms generated by a design simulation. The static probability and toggle rate for each signal is calculated from the simulation waveform and is stored in a signal activity file (.saf). The summary of the PowerPlay power analyzer compilation report is as shown in Figure 3.6, which consists of the estimated total thermal power dissipation of the design. The total thermal power includes dynamic, static and I/O thermal power dissipation. The compilation report also includes a confidence metric that reflects the overall quality of the data sources for the signal activities.

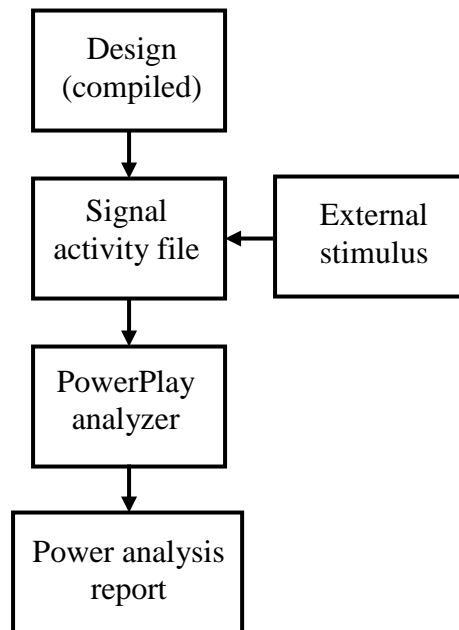


Figure 3.5: PowerPlay power analyzer design flow.

3.2 Cadence

An application specific integrated circuit is an integrated circuit customized for a particular use. In this work, Cadence is used for an ASIC design because it is widely used in the industry. Using Cadence, an ASIC can be designed from textual description Verilog HDL to layout without using any additional softwares. An ASIC design is performed using the standard cell library provided by Virginia Polytechnic Institute and State University [63]. The advantage

of using a standard cell library is to save time. Using a predesigned and pretested standard cell library also reduces the design implementation risk. In this work, an ASIC implementation of the structured encoder is performed using Cadence. The design flow of an ASIC implementation in Cadence is as shown in Figure 3.7. The Verilog HDL design is first synthesized in Cadence RTL Compiler. The synthesized design then goes through place and route using Cadence Encounter. The final layout of the design from Cadence Encounter can be imported into Cadence ICFB and where design rule checks are performed.

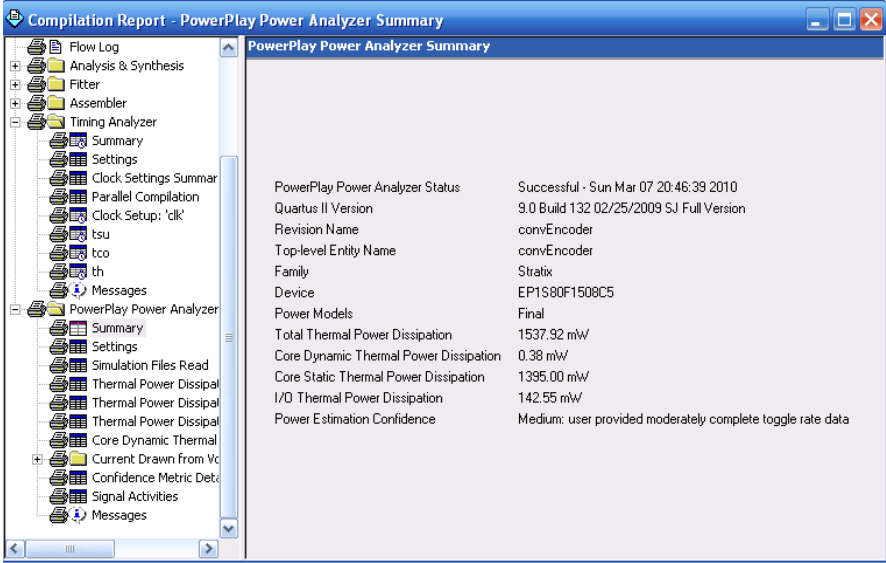


Figure 3.6: PowerPlay power analyzer summary.

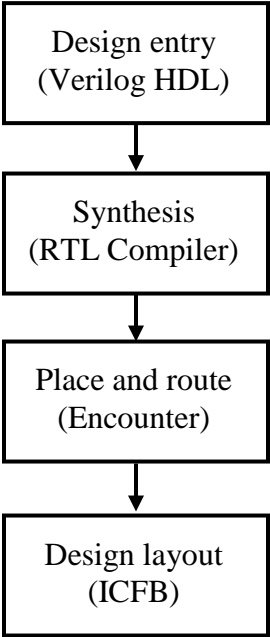


Figure 3.7: ASIC design flow.

3.2.1 RTL Compiler

RTL Compiler is used to synthesize design in Verilog HDL. The RTL Compiler design flow is shown in Figure 3.8 [64, 65]. After invoking RTL Compiler, the Verilog HDL files are first read and checked for syntax and synthesis policy checks. Then the design is built using generic components. The library search path and library that will be used for the design synthesis needs to be specified. The design is read and creates HDL independent objects in HDL-intermediate format and stores it in a design library. During elaboration the top-level design is bound with all the designs and packages.

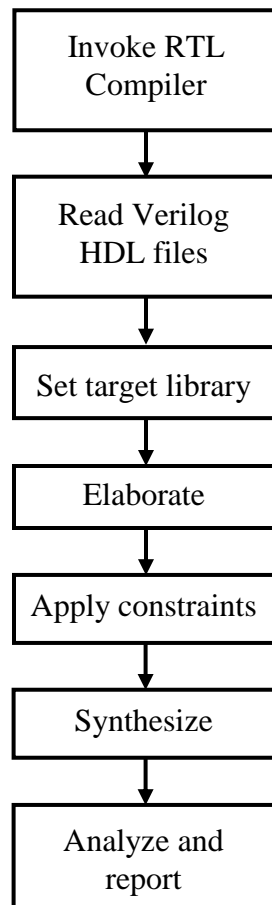


Figure 3.8: Design flow in RTL Compiler.

At this stage of the design, additional constraints are applied. The constraints typically include defining any clocks. Other constraints like operating requirements, setup/hold times, I/O delays, etc., are specified. The design is synthesized during which logic optimization is performed. The design is mapped to actual gates from the target technology library, producing a circuit that hopefully meets the requirements. If the design is successfully synthesized, then it has

been fully mapped to gate-level. The generated mapped design can be used by Cadence Encounter to place and route the design. Finally the timing, power and area of the design can be analyzed using the tools in RTL Compiler.

3.2.2 Encounter

After the design is synthesized in RTL Compiler, Encounter is used to perform automatic placement and routing of the synthesized design. A place and route (PNR) tool takes a gate-level netlist as the input and determines how each gate should be placed on the chip. The design flow in Encounter is as shown in Figure 3.9 [66]. Encounter is invoked and the design synthesized in RTL Compiler is imported. A Floorplan is performed on the imported design. Die size and core margins of the chip are specified. The die size is chosen such that the router would have enough space to be able to place all the metal interconnects and any buffers needed during optimization.

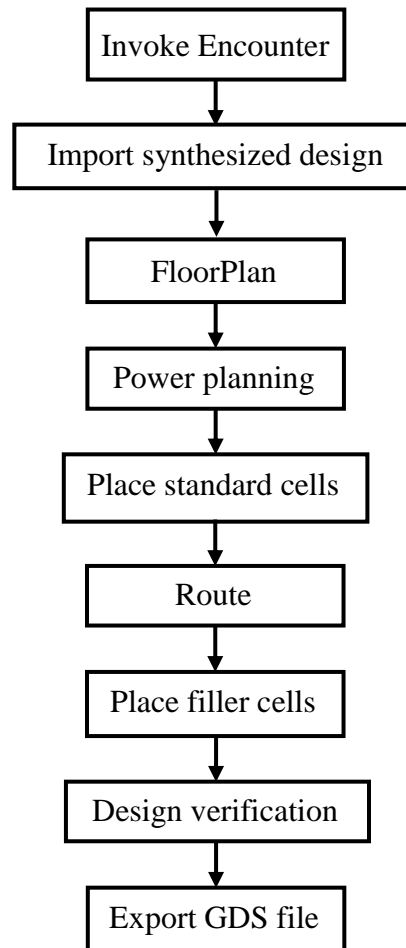


Figure 3.9: Design flow in Encounter.

The next step is power planning. Power rings and stripes are configured. The width of the power lines are determined by the size of the chip. Wider lines are used for bigger designs. Sroute is performed to do the final power routing. Standard cells are placed in the design. The design is routed using NanoRoute. Filler cells are added to allow the wells to be at the same potential. Connectivity and geometry of the design layout are verified. The design should pass these tests. The foundry needs the design in GDS format. Next, the design is exported to a GDS file during which a new Verilog netlist based on placement and routed design is generated.

3.3 Matlab

Various design and performance evaluation aspects of LDPC encoding and decoding algorithms were performed using Matlab. In the encoding algorithm, the codeword generated from Matlab is used to validate the codeword obtained from the encoder implemented on FPGA. Encoder for randomly generated LDPC codes is developed using Richardson's greedy algorithm [22]. According to greedy algorithm, the parity-check matrix should be transformed into lower triangular form for the encoding process. This step is called as preprocessing of parity-check matrix. The encoding process is further simplified and the details are presented in chapter 4. Two matrices P_1 and P_2 are computed for the encoding process. The preprocessing of parity-check matrix and the computation of matrices P_1 and P_2 are performed in Matlab.

The decoder performance is based on several decoder parameters such as decoding algorithm, quantization of the log-likelihood ratios, and maximum number of decoding iterations. The decoder simulations are performed in Matlab using different decoding algorithms. Based on the simulation results, the decoding algorithm that gives the best BER performance is chosen for decoder hardware implementation. Different quantizations of log-likelihood ratios and non-linear function φ used in the decoding process are evaluated using Matlab. Simulations are also performed in Matlab by varying the maximum number of decoding iterations for the decoding process. Based on these simulation results the quantization of the log-likelihood ratios and maximum number of decoding iterations for the decoding process are chosen. The details and the results of the simulations are presented in chapter 7.

Matlab is also used to generate Verilog modules required for the implementation of encoder and decoder of the LDPC codes in hardware. The encoder and decoder of LDPC codes are designed to have flexibility in code length and code rate. In order to incorporate this

flexibility, the Verilog HDL module parameters have to accommodate the updates and/or changes. Therefore, Matlab script to generate a generic Verilog module was written. Based on the desired LDPC codes, the code length and code rate are selected and the corresponding Verilog HDL modules can then be generated by running the Matlab script. The Verilog HDL files are written using the fprintf (write data to text file) command in Matlab. Following is an example script to generate a Verilog HDL module which implements a 1/2 rate convolutional encoder with variable constraint length.

Example: Matlab file to generate Verilog HDL module to design a 1/2 rate convolutional encoder with variable constraint length.

```
clear all; clc;
% Example: 1/2 rate convolutional encoder with constraint length 7
% Parameters needs to be changed based on desired convolutional encoder
N = 7;
g0 = [6 4 3 1 0];
g1 = [6 5 4 3 0];
% open the file with write permission
fid1 = fopen('convEncoder.v','w');
% write the required data to the file
fprintf(fid1,'module convEncoder(n, k, clk, reset);\n');
fprintf(fid1,'parameter N = %d;\n',N);
fprintf(fid1,'input k, clk, reset;\n');
fprintf(fid1,'output wire [1:0] n;\n');
fprintf(fid1,'reg [N-1:0] p0;\n');
fprintf(fid1,'always@(negedge reset or posedge clk)\n');
fprintf(fid1,'if (~reset)\n');
fprintf(fid1,'\tp0 = {(N){1'b0}};\n');
fprintf(fid1,'else\n');
fprintf(fid1,'\tp0 = {k, p0[%d-1:1]};\n\n',N);
fprintf(fid1,'assign n[0] = ');
for i = 1:length(g0)-1
```

```
fprintf(fid1,'p0[%d]^',g0(i));
end
fprintf(fid1,'p0[%d];\n',g0(length(g0)));
fprintf(fid1,'assign n[1] = ');
for i = 1:length(g1)-1
fprintf(fid1,'p0[%d]^',g1(i));
end
fprintf(fid1,'p0[%d];\n\n',g1(length(g1)));
fprintf(fid1,'endmodule\n');
% close the file when finished
fclose(fid1);
```

CHAPTER 4 - Encoder Design for Randomly Generated Low-Density Parity-Check Codes

In general, encoder implementations of LDPC codes are optimized for area due to their high complexity. Such designs usually have relatively low data rate. In this chapter, two new encoder designs are presented that achieve much higher data rates while requiring more area for implementation.

In this chapter, two encoder design methodologies and their implementation results are presented. The key aspects of the design are summarized as follows:

- The efficient algorithm presented in 2.1.2 is used to develop a hardware implementation of faster encoders for LDPC codes. The specific efficient algorithm is the greedy algorithm A presented by Richardson and Urbanke in [22].
- The encoder designs are independent of code length, code rate and structure of the parity-check matrix. Hence these designs can be used for both structured and any randomly generated regular and irregular parity-check matrices.
- The encoder uses a direct implementation which sacrifices area for increased speed, but this is necessary as applications require more bandwidth.
- The design is implemented using Mackay's regular and irregular LDPC codes [56]. For this purpose, 1/2 rate regular LDPC codes with code lengths of 256, 512 and 1024 and 1/2 rate irregular LDPC codes with code lengths of 504 and 1008 are considered.
- One of the designs achieves encoding rates of up to 844 Mbps. Both of the designs presented can fit on FPGAs currently available.

4.1 Encoder Design

An overview of the LDPC encoding process is shown in Figure 4.1. The encoding process consists of two steps. In the first step, the parity-check matrix is transformed to approximate lower triangular form, H_{pre} . For any given parity-check matrix, this step needs to be performed only once and hence this step can be performed offline in software such as Matlab. In the second step, hardware encoding is performed. Since the codeword is obtained using the modified parity-check matrix, it needs to be rearranged to obtain the final codeword with regard to the original parity-check matrix.

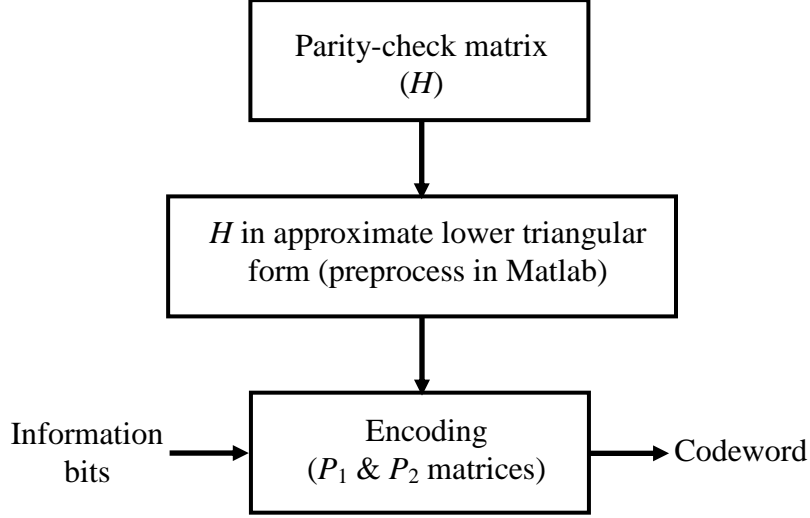


Figure 4.1: Overview of the LDPC encoder.

4.1.1 Preprocessing

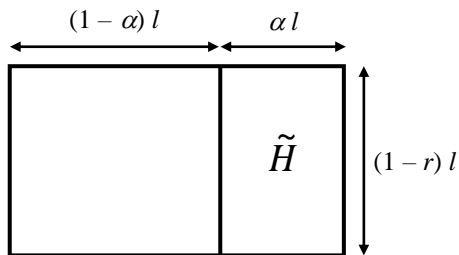
In preprocessing, the parity-check matrix is first transformed to approximate lower triangular form, H_{pre} . This processing requires the following three steps:

1. Any variable node (i.e., any column in H) which is connected to a degree-one (i.e., rows of H having one non-zero element) check node (i.e., row in H) is declared to be known. For any given H , each column in H is declared independently to be known with probability $(1-\alpha)$ or, otherwise, to be an unknown (erasure). The $(1-\alpha)l$ known columns are then reordered to form the leading columns of the matrix H as shown in Figure 4.2 (a) where l is the number of columns.

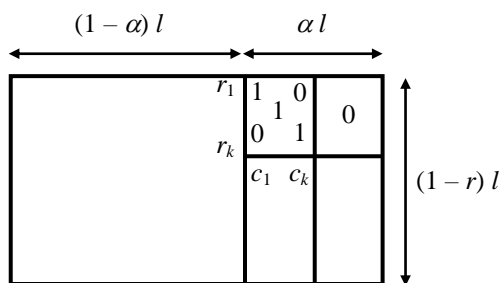
2. Assuming that the residual matrix, \tilde{H} , has rows of degree-one, the columns connected to degree-one rows are then identified. Let these columns be c_1, \dots, c_k and let r_1, \dots, r_k be the degree-one rows such that c_i is connected to r_i . These new known columns and their associated rows are ordered along a diagonal as shown in Figure 4.2 (b).

3. Furthermore, step 2 is repeated until all the degree-one rows are exhausted. If this procedure does not stop prematurely then the row gap is $(1-\alpha)l$ and the column gap is $(1-r-\alpha)l$ as shown in Figure 4.2 (c). If the procedure terminates before all columns are exhausted then the remaining columns are reordered to the left. Let the remaining columns be δl then the column gap is $(1-\alpha+\delta)l$ and the row gap is $(1-r-\alpha+\delta)l$. For a given parity-check matrix, H , this preprocessing needs to be performed only once. Hence this step is performed in Matlab. The

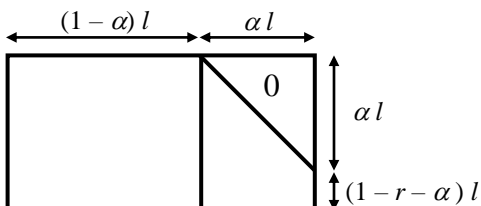
obtained H_{pre} is divided into sub-matrices as shown in Figure 4.2 (d). All these matrices are sparse and T is lower triangular with ones along the diagonal.



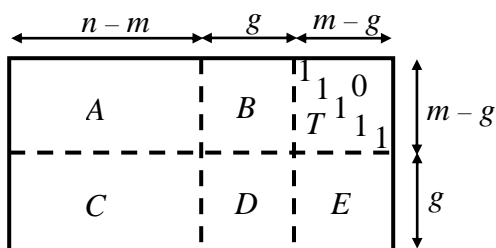
(a) Parity-check matrix after the application of first step in greedy algorithm A.



(b) Parity-check matrix after the application of second step in greedy algorithm A.



(c) Parity-check matrix after the application of third step in greedy algorithm A.



(d) The parity-check matrix in approximate lower triangular form, H_{pre} , and its division of submatrices.

Figure 4.2: Application of greedy algorithm A on H .

The parity bits are computed using Equations 2.6 and 2.7, which can also be written as shown below in Equations 4.1 and 4.2. The parity bits p_1 and p_2 , are obtained by multiplying information bits, s , with matrices P_1 and P_2 .

$$\begin{aligned}
p_1 &= \phi^{-1}(ET^{-1}A + C)s \\
&= P_1s, \text{ where } P_1 = \phi^{-1}(ET^{-1}A + C). \tag{4.1}
\end{aligned}$$

$$\begin{aligned}
p_2 &= T^{-1}(As + Bp_1) \\
&= T^{-1}(As + B\phi^{-1}(ET^{-1}A + C)s) \\
&= T^{-1}(A + B\phi^{-1}(ET^{-1}A + C))s \\
&= P_2s, \text{ where } P_2 = T^{-1}(A + B\phi^{-1}(ET^{-1}A + C)). \tag{4.2}
\end{aligned}$$

Only the P_1 and P_2 matrices are required for encoding LDPC codes. For the computation of the P_1 and P_2 matrices, the inverse of ϕ matrix, i.e., $(-ET^{-1}B + D)^{-1}$ is used. Therefore the ϕ matrix has to be non-singular. If ϕ is singular, then the columns of B are swapped with the columns in A until ϕ matrix is non-singular. This complete process, transforming H to approximate lower triangular form, H_{pre} , and obtaining matrices P_1 and P_2 , is performed in Matlab. While a smaller gap, g , is suggested as outlined in [22], here it is only necessary that $g > m/2$. This is because both P_1 and P_2 work independently and concurrently, so making one considerably more compact than the other does not lead to an encoder which is faster.

Once the P_1 and P_2 matrices are defined in Matlab, the next step is to find the best way to store these matrices on the chip. For doing this, matrices P_1 and P_2 are computed using Matlab for both regular and irregular parity-check matrices of different sizes. The number of one's in each matrix is shown in Table 4.1.

Table 4.1: Density (number of one's) of H , P_1 and P_2 matrices.

H	No. of one's in H	No. of one's in P_1	No. of one's in P_2
Irregular parity-check matrix			
252×504	2014	15878	15839
504×1008	4033	63359	63493
Regular parity-check matrix			
256×512	1536	14638	14456
512×1024	3072	52279	56871

From Table 4.1, it can be observed that the matrices P_1 and P_2 are not sparse. For example, the distribution of number of one's in the rows of a P_2 matrix for an irregular parity-check matrix of size 504×1008 is shown in Figure 4.3. This indicates that approximately 220-290 one's are in

every row vector of P_2 . While a sparse representation for the P_1 and P_2 matrices can be utilized, it is more efficient to use a dense representation of the matrices due to the dense properties of the matrices themselves.

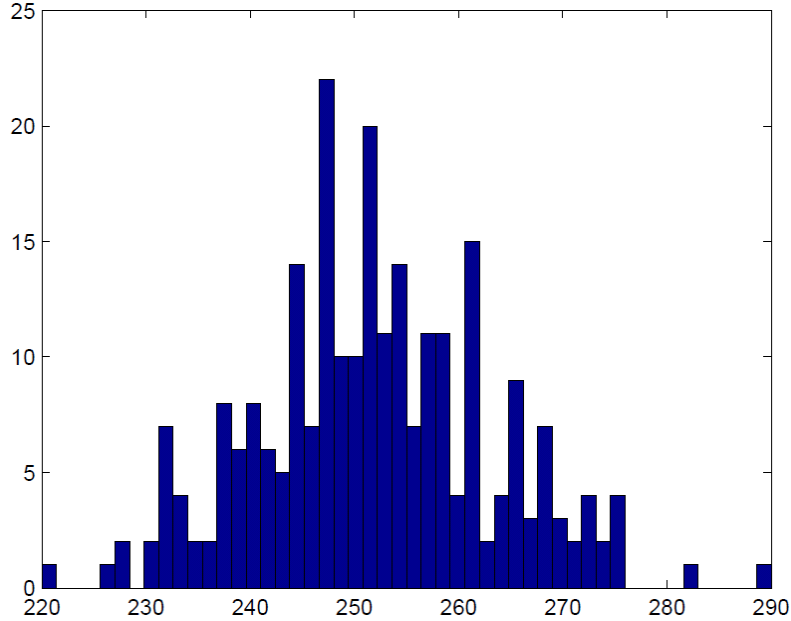


Figure 4.3: Distribution of number of one's in each row of P_2 matrix for an irregular H of size 504×1008 .

4.1.2 Hardware Implementation

The Hardware implementation of the encoder, as shown in Figure 4.4 and Equations 4.1 and 4.2, is to multiply the information bits, s , with matrices P_1 and P_2 to obtain parity bits p_1 and p_2 respectively. The multiplication and addition in binary system can be performed with an AND gate and an XOR gate respectively. The length of information bits is $n-m$. The encoder assumes that the information bits are available and the latency involved in reading the information bits is not considered. Therefore, a serial input interface is used to read the information bits and it requires $n-m$ clock cycles. The information bits are read using a faster clock Clk_s and the encoding is done using a slower clock, Clk_e .

Matrices P_1 and P_2 are stored on logic elements in arrays so that the data can be retrieved simultaneously for all rows. This will help to reduce the latency involved if the matrices are stored in the onboard RAM. To maximize the parallelism, matrix-vector multiplication is performed by a multiple vector-vector multiplications (inner product) in parallel. Each vector-

vector multiplication can be performed in two ways. The first method, multi clocked inner product (MCIP), requires m clocks to compute one inner product while the second method, single clocked inner product (SCIP), computes the inner product in a single clock.

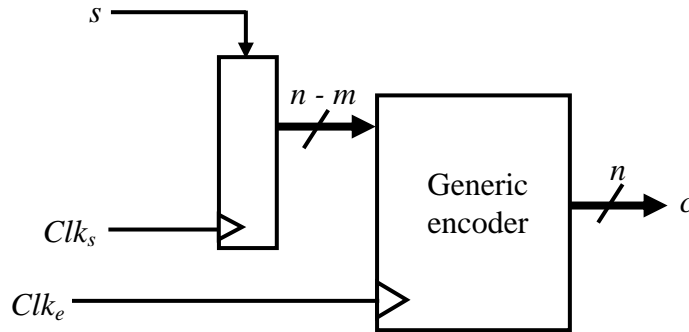


Figure 4.4: Complete system of the generic encoder.

4.1.2.1 Multi Clocked Inner Product

In this method the inner product of vectors In_1 and In_2 , each of length m , is performed one bit at a time as illustrated in Figure 4.5. Bit-wise multiplication is performed on each clock cycle with a single AND gate, and a running single-bit sum, Out , of the products is achieved using a single XOR gate. If vectors In_1 and In_2 are stored in m -bit shift registers, then this method requires m clock periods to calculate their inner product. Multiple instantiations of this inner product module may be implemented in parallel on all the arrays of matrices P_1 and P_2 to obtain the parity bits p_1 and p_2 . Therefore, the coded data rate is determined by the minimum period of the shift register clock, Clk_e .

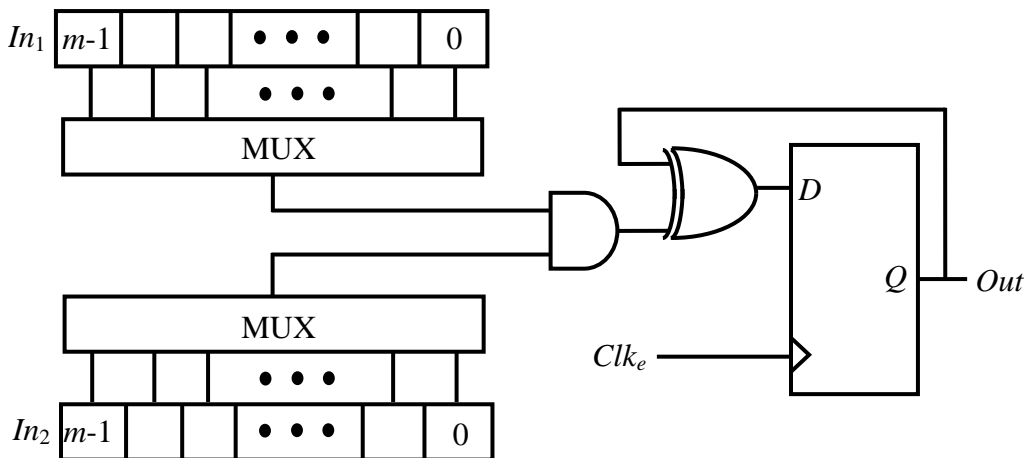


Figure 4.5: Circuit for multi clocked inner product (MCIP).

4.1.2.2 Single Clock Inner Product

This method is similar to the multi clocked inner product except that the inner product is done in a single clock cycle. The illustration of this method is shown in Figure 4.6. In a single clock, as shown in Figure 4.6, all pairs of bits from vectors In_1 and In_2 are ANDed and the output of each AND gate is XORed to obtain the inner product, Out . This procedure may also be performed in parallel on all arrays of matrices P_1 and P_2 . The coded data rate of this method is determined by the maximum propagation delay from any bit in In_1 and In_2 to the output, Out . This delay will set the minimum period of the encoder clock, Clk_e .

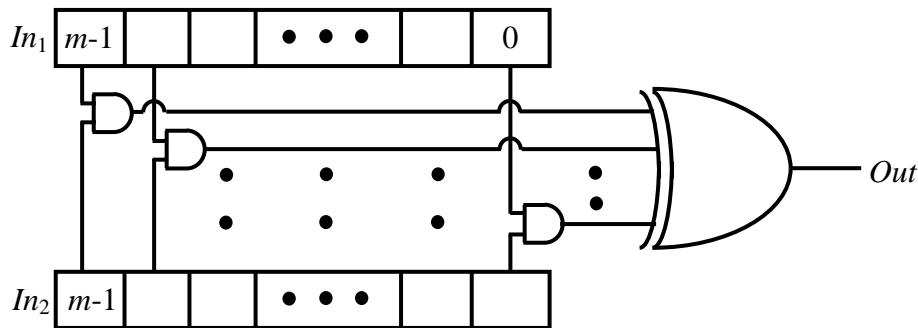


Figure 4.6: Circuit for single clocked inner product (SCIP).

After obtaining the parity bits p_1 and p_2 using either of the above methods, the information and parity bits are rearranged to obtain the final codeword with respect to the actual parity-check matrix, H . The number of clock cycles required for a codeword using the multi clocked inner product (MCIP) method is equal to $m + 1$ where the m clock cycles are required to compute the inner product and the additional clock is required to rearrange the codeword bits. When single clocked inner product (SCIP) is used the number of clock cycles required for codeword is two where the inner product is obtained in one clock cycle and the codeword bits are rearranged in the second clock cycle.

4.2 Results

As described in section 3.3, Verilog modules for these encoder designs are generated using a Matlab script. The Verilog modules are then synthesized using Quartus and implemented on a Stratix EP1S80F1508C5 FPGA. A computer is used to send the information bits to the FPGA and to read each resulting codeword from the encoder. For verification, the hardware

encoder output is then compared to the corresponding codeword generated by Matlab. A comparison of the performance of both design strategies was achieved by implementing each version for different types and sizes of rate 1/2 parity-check matrices. As discussed previously, preprocessing was performed on each H using Matlab. This is not a major concern because this step is performed only once for a given parity-check matrix. The actual hardware encoder results are shown in Tables 4.2 and 4.3. The maximum clock speed shown is that of the encoder clock, Clk_e . The coded data rate and latency calculations are based on the internal encoder design and not on any special I/O limitations.

Table 4.2: Synthesis results of encoder implementation using MCIP on Stratix EP1S80F1508C5.

H	Encoder implementation using MCIP				
	LE 's	CPC	Clk_e (MHz)	Coded data rate (Mbps)	Latency (μ s)
Reg 128×256	2014	129	76.65	152.12	1.683
Reg 256×512	6580	257	60.73	120.98	4.232
Reg 512×1024	22978	513	46.02	91.86	11.147
Irreg 252×504	7485	253	69.73	138.9	3.628
Irreg 504×1008	28459	505	51.57	102.94	9.793

Table 4.3: Synthesis results of encoder implementation using SCIP on Startix EP1S80F1508C5.

H	Encoder implementation using SCIP				
	LE 's	CPC	Clk_e (MHz)	Coded data rate (Gbps)	Latency (ns)
Reg 128×256	1143	2	319.49	40.9	6.26
Reg 256×512	3508	2	262.4	67.18	7.62
Reg 512×1024	12664	2	318.47	163.06	6.28
Irreg 252×504	5450	2	238.55	60.12	8.38
Irreg 504×1008	22249	2	316.56	159.54	6.32

The encoder implementation using both methods MCIP and SCIP assumes that all input data bits are available for encoding, so any serialization delay factor is not included in the results shown in Tables 4.2 and 4.3. The important observation here is that using a SCIP encoder has a huge advantage in terms of data rate and latency over the MCIP encoder. This is somewhat expected for the defined architecture of each system. Another interesting observation is the difference in area for each design. One would normally expect the MCIP to require fewer *LEs*, but the converse is actually true. This is due to the implementation of huge multiplexer's required by MCIP in the FPGA. May be SCIP encoder required fewer *LEs* than MCIP encoder due to the statically defined P_1 and P_2 matrices. It is observed that increasing the code length decreases the clock speed. Also an irregular LDPC code takes considerably more area than a regular LDPC code. Again this can be attributed to a difference in potential optimization for the two different codes as they are defined in P_1 and P_2 .

If one wants to consider both encoders under a serial input stream then an input shift-register needs to be added to both the MCIP and SCIP encoders. The latency in reading the information bits is m/Clk_s . Let the encoding clock frequency be Clk_e , which is equal to the maximum clock frequency of the synthesized designs shown in Tables 4.2 and 4.3. The latency of the complete system, shown in Figure 4.4, is the maximum value of $[m/Clk_s, CPC/Clk_e]$, which becomes CPC/Clk_e for MCIP encoder and m/Clk_s for SCIP encoder. Therefore the coded data rate is equal to $m \times Clk_e / (CPC \times code\ rate)$ for MCIP encoder and $Clk_s / (code\ rate)$ for SCIP encoder. The synthesis results of the complete generic encoder system using MCIP and SCIP are shown in Table 4.4. It can be observed from Tables 4.2 and 4.4 that the MCIP encoder coded data rate is not affected by I/O serialization. However, coded data rate of the complete encoder system using SCIP becomes limited by the speed of the shift register, which in this case is 422.12 MHz.

The coded data rate decreases with the increase in the size of the parity-check matrix for the MCIP encoder whereas it is independent of the size of the parity-check matrix for the SCIP encoder. This encoding process is not restricted by the properties of the original H matrix, and it is also completely flexible with respect to code length and code rate. Hence it can encode any LDPC codes. Although the implementation is based on H matrices that are binary, it can be extended to matrices that belong to higher order fields. All of the designs presented can fit on FPGAs currently available.

Table 4.4: Synthesis results of complete encoder system using MCIP and SCIP implemented on Stratix EP1S80F1508C5.

H	Complete system of MCIP encoder		Complete system of SCIP encoder	
	Latency (μ s)	Coded data rate (Mbps)	Latency (μ s)	Coded data rate (Mbps)
Regular 128×256	1.683	152.12	0.303	844.24
Regular 256×512	4.232	120.98	0.606	844.24
Regular 512×1024	11.147	91.86	1.213	844.24
Irregular 252×504	3.628	138.9	0.597	844.24
Irregular 504×1008	9.793	102.94	1.194	844.24

Both of the implementations presented here provide a significant increase in coded data rate compared to the design presented in [29]. Lee in [29] implemented an encoder on a Xilinx Virtex-II XC2V4000-6 using Richardson and Urbanke's encoding algorithm. Table 4.5 shows the implementation results for various encoders presented in [29] with code lengths ranging from 500 to 8000 bits for rate 1/2. In order to compare the results of the encoder presented in [29] to our design [67], the slices (smallest unit of logic in Xilinx) required by the design in [29] need to be converted to logic elements. Xilinx Virtex-II XC2V4000 FPGA has 23040 slices [68] and its equivalent Stratix logic elements are 57600 [69]. Therefore, 1 slice of Xilinx is approximately equal to 2.5 logic elements. The number of approximate logic elements required by the encoder presented in [29] is also included in Table 4.5.

Table 4.5: Synthesis results of LDPC encoder designed by Lee [29].

H	Coded data rate (Mbps)	Slices	Equivalent LEs
250×500	50	562	1405
500×1000	48	682	1705
1000×2000	45	870	2175
2000×4000	40	1340	3350
4000×8000	34	2148	5370

In order to maximize the coded data rate, the design presented in [29] uses multiple instances of the encoder, which is not required in our design [67]. In [29], for code length of 2000 and code rate 1/2, it is shown that by using 16 encoder instances instead of one encoder instance the coded data rate is increased from 45 Mbps to 410 Mbps which requires 16906 slices. In this case, the equivalent logic elements are 42265. To get a higher data rate, the design presented in [29] also requires large area for its implementation.

From Tables 4.4 and 4.5, it can be observed that for code lengths of 500 and 1000, the coded data rate of our designs is greater than the design presented in [29]. The coded data rate of the encoder implemented using MCIP method is greater than or equal to twice the coded data rate of the design in [29]. The encoder implemented using the SCIP method has a very high coded data rate approximately 17 times the coded data rate of the design presented in [29]. Although the required area for our design is significantly larger, its use in high-speed applications would not require the parallelization that other designs propose.

CHAPTER 5 - Encoder Design for Structured Low-Density Parity-Check Codes

In the previous chapter, a LDPC encoder for randomly generated LDPC codes was presented. Due to the randomness in the LDPC codes, the encoder implementation requires large area. The use of structured LDPC codes decreases the encoding complexity and also provides design flexibility.

In this chapter, the encoder design and its hardware implementation for structured LDPC codes are described.

- An encoder architecture is presented that adheres to the structured LDPC codes defined in the IEEE 802.16e standard. The encoder supports codes with rates 1/2, 2/3, 3/4 and 5/6 and code lengths ranging from 576-2304.
- The coded data rate is equal to 844, 633, 562 and 506 Mbps for code rates 1/2, 2/3, 3/4 and 5/6 respectively. For a given code rate, the coded data rate is constant for varying code lengths.
- The design methodology is flexible in terms of both the code rate and code length. Hence the design can also be used for similar structured LDPC codes defined in other standards.

5.1 Structured LDPC Codes

The parity-check matrices defined in the IEEE 802.16e standard are used for the encoder implementation. Standard IEEE 802.16e defines LDPC codes as a set of one or more fundamental LDPC codes. Each of the fundamental codes support code lengths from 576 to 2304 with code rates of 1/2, 2/3 A, 2/3 B, 3/4 A, 3/4 B and 5/6. The parity-check matrix, H , is of size $m \times n$, where m is the number of parity-check bits in the code and n is the length of the code. The parity-check matrix H is expanded from a base parity-check matrix, H_b . The size of H_b is $m_b \times n_b$ where $m_b = m/z$, $n_b = n/z = 24$ and z is an integer greater than zero. The value of m_b varies with code rate of the LDPC codes. Its value is equal to 12, 8, 6 and 4 for code rates of 1/2, 2/3, 3/4 and 5/6. A parity-check matrix is obtained by replacing each -1 in H_b with a $z \times z$ zero matrix, each 0 with a $z \times z$ identity matrix, and any element greater than zero with a $z \times z$ permutation matrix. The permutation matrix is an identity matrix that has been circularly right shifted by the associated value specified in H_b .

$$H_{b_2} = \begin{bmatrix} 95 & 0 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & 0 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 0 & 0 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & 0 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & 0 & 0 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & 0 & 0 & -1 \\ 0 & -1 & -1 & -1 & -1 & -1 & 0 & 0 \\ 95 & -1 & -1 & -1 & -1 & -1 & -1 & 0 \end{bmatrix} \quad (5.4)$$

5.2 Design Methodology

Using structured LDPC codes considerably simplifies the encoder and makes the design straightforward compared to other encoders. A modified version of the generic encoding method described in 2.1.1 is used for hardware implementation of structured LDPC codes. Encoding of LDPC codes also uses the property $H_b x^T = 0^T$, where x and H_b is the base parity-check matrix.

Codeword x may be split into the information bits, S , and parity bits, p , i.e., $x^T = \begin{bmatrix} S \\ p \end{bmatrix}$. The size of S is $k_b \times 1$ and the size of p is $m_b \times 1$, so

$$H_b x^T = 0^T$$

becomes

$$\begin{bmatrix} H_{b_1} & H_{b_2} \end{bmatrix} \begin{bmatrix} S \\ p \end{bmatrix} = 0^T. \quad (5.5)$$

Expanding and solving for p one finds

$$H_{b_1} S + H_{b_2} p = 0 \quad (5.6)$$

$$p = H_{b_2}^{-1} H_{b_1} S. \quad (5.7)$$

Matrix $H_{b_2}^{-1}$ is no longer sparse when compared to H_{b_2} . Therefore, direct implementation of Equation 5.7 has high encoding complexity. However, the parity bits are easily solved by exploiting the dual diagonal structure of the H_{b_2} matrix, which is explained in subsection 5.3.2. Let the product of matrices H_{b_1} and S be denoted by V . Therefore, for modulo 2 operations, Equation 5.6 can also be written as

$$H_{b_2} p = H_{b_1} S = V. \quad (5.8)$$

The parity bits, p are obtained by solving

$$H_{b_2} p = V. \quad (5.9)$$

5.3 Hardware Implementation

The encoder implementation is performed in two steps. First step, the product of matrices H_{b1} and information bits, S , is computed and is denoted by V . In the second step, the parity bits, p , are computed by solving Equation 5.9. As will be explained in section 5.3, computation of V and p each require a time period of t . Therefore the encoding process is done in a period of $2t$ as shown in Figure 5.1. To reduce the time period required for the encoding process to half of its time period, the two steps required for the computation of V and p can be performed in a pipeline fashion as shown in Figure 5.2. First, V is computed for the first set of information bits. Then the parity bits, p , are computed. During the computation of p for the first set of information bits, V is computed for the second set of information bits simultaneously. This pipeline implementation of the encoder increases the encoding data rate by decreasing the time required for the encoding process.

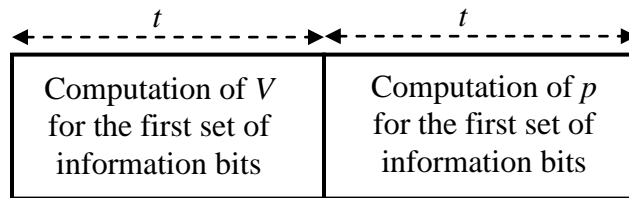


Figure 5.1: Encoding process.

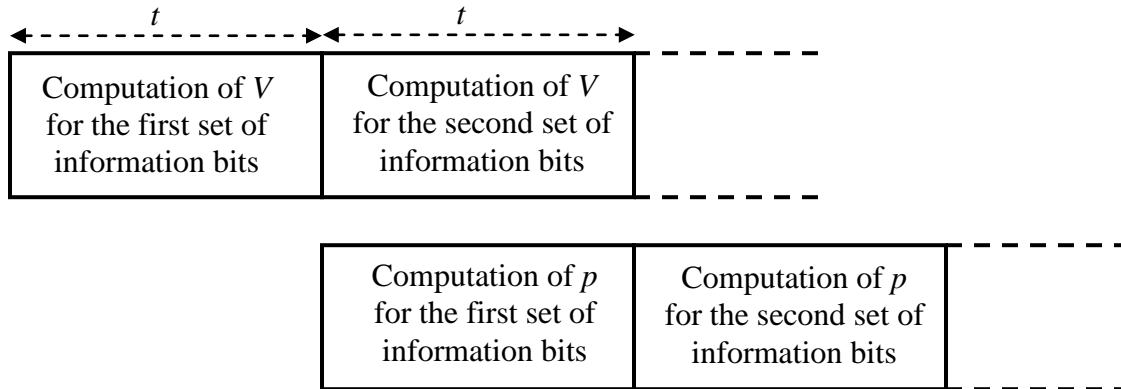


Figure 5.2: Overview of encoding process.

5.3.1 Computation of V

The first step in the encoder implementation is the computation of V , the product of matrix H_{b1} and vector S . This can be obtained by vector-vector multiplication of each row in H_{b1} with the column vector S . To maximize the efficiency, vector-vector multiplication is performed in parallel.

5.3.1.1 Vector-Vector Multiplication

The matrix H_{b1} is a sparse matrix with each element representing a zero matrix, identity matrix, or permutation matrix of size $z \times z$. The inner product, v_i , an element of the vector V , is obtained by multiplying the i th row in H_{b1} with S as shown

$$v_i = [h_{b1(i,0)} \quad h_{b1(i,1)} \quad \dots \quad h_{b1(i,k_b-1)}] \begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_{k_b-1} \end{bmatrix} \quad (5.10)$$

$$\begin{aligned} &= h_{b1(i,0)}s_0 + h_{b1(i,1)}s_1 + \dots + h_{b1(i,k_b-1)}s_{k_b-1} \\ &= \sum_{j=0}^{k_b-1} e_{p(i,j)} \end{aligned} \quad (5.11)$$

$$\text{where } e_{p(i,j)} = h_{b1(i,j)}s_j = \begin{cases} 0 & \text{if } h_{b1(i,j)} = -1, \\ s_j & \text{if } h_{b1(i,j)} = 0, \\ s_j(h_b) & \text{if } h_{b1(i,j)} = h_b. \end{cases} \quad (5.12)$$

The product of $h_{b1(i,j)}$, an element in H_{b1} , with s_j , a vector in S , is denoted by $e_{p(i,j)}$. It is defined as shown in Equation 5.12, where $s_j(h_b)$ is the circular right shifted version of the vector s_j and the circular right shift value is defined by $h_{b1(i,j)}$. An additional clock cycle is required to add all the elements of e_p to obtain v . In modulo 2, v is obtained by performing an XOR operation on all the elements of e_p .

5.3.1.2 Computation of e_p

Vector e_p is defined as the product of the $z \times z$ matrix h_{b1} and a $z \times 1$ vector s as shown in Equation 5.11. Each $s_j(h_b)$ is obtained by circular right shifting the vector s_j by a particular shift value, h_b , defined by $h_{b1(i,j)}$. If the value of h_b is greater than $z/2$, then a circular right shift is performed on s_j and the number of shifts required to obtain the corresponding $e_{p(i,j)}$ is equal to $h_b - z/2$. If the value of h_b is less than $z/2$, then a circular left shift is performed on s_j and the number of shifts required to obtain corresponding $e_{p(i,j)}$ is equal to h_b . If a shift is performed on each clock cycle, then the computation of e_p requires $z/2$ clock cycles.

5.3.2 Computation of Parity Bits

The second step in the encoding process includes the computation of parity bits. Equation 5.9 can be rewritten as shown in Equation 5.13. Solving, we get $h_b(0)p_0 + p_1 = v_0$, $p_1 + p_2 = v_1$, \dots , $h_b(j)p_0 + p_j + p_{j+1} = v_j$, \dots , $h_b(m_b - 1)p_0 + p_{m_b-1} = v_{m_b-1}$. Adding all of these equations results in $p_0 = v_0 + v_1 + \dots + v_{m_b-1}$. p_0 can be computed in a single clock cycle by XORing all the elements of v . Once p_0 is obtained, the remaining parity bits can be computed by using the following expressions: $p_1 = v_0 + h_b(0)p_0$, $p_2 = v_1 + p_1$ etc., where $h_b(0)p_0$ is now the shifted version of p_0 whose shift value is defined by $h_b(0)$. The $h_b(0)p_0$ is computed using the method described in above subsection 5.3.1.2. This procedure would require $z/2$ clock cycles. Once p_0 and $c_{(0,0)}p_0$ are obtained, they can be used to compute the values of the remaining parity bits. This step can be performed in a single clock period.

$$\begin{bmatrix} h_b(0) & 0 & & & & & \\ \cdot & 0 & 0 & & -1 & & \\ \cdot & & 0 & 0 & & & \\ h_b(j) & & & \cdot & \cdot & & \\ \cdot & & & & 0 & 0 & \\ \cdot & & & & -1 & 0 & 0 \\ h_b(m_b - 1) & & & & & 0 & 0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ p_{m_b-1} \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ v_{m_b-1} \end{bmatrix}. \quad (5.13)$$

5.4 Results

Verilog modules are again generated using a Matlab script as explained in section 3.3. A hardware implementation was performed on an Altera Stratix EP1S80F1508C5 FPGA using Quartus II for synthesis. The synthesis results for different code lengths and code rates are shown in Table 5.1. In Table 5.1, variable z , represents the size of the sub-matrix in the base matrix, H_b , and is equal to $n/24$. Column LE denotes the number of logic elements required for the implementation of the encoder on the FPGA, while CPC represents the number of clock cycles required per codeword for encoding. CPC is equal to the maximum number of clock cycles required for computation of V and p . The computation of V requires $z/2 + 3$ clock cycles in which $z/2$ clock cycles are required to compute e_p , and three clock cycles are required for loading and processing the data and computing V . Computation of p requires $z/2 + 3$ clock cycles, in which

one clock cycle is used for computing p_0 , $z/2$ clock cycles are required to compute $h_b(0)p_0$, and the remaining two clock cycles are required for loading the data and computing the remaining parity bits. Hence this method requires $z/2 + 3$ clock cycles. Clk_e in Table 5.1 represents the encoder clock. From Table 5.1, it is observed that increasing the code length increases LEs and CPC . Synthesis results of LEs required for different code lengths and code rates is shown in Figure 5.3. The coded data rate is equal to $m \times Clk_e / (CPC \times code\ rate)$.

Table 5.1: Synthesis results of structured encoder using LDPC codes defined in 802.16e.

n	z	Code rate 1/2			Code rate 2/3		
		LE	Clk_e (MHz)	Coded data rate (Gbps)	LE	Clk_e (MHz)	Coded data rate (Gbps)
576	24	3391	192.23	7.38	4039	176.71	6.78
960	40	5100	159.57	6.66	6056	169.2	7.07
1440	60	7012	164.83	7.2	8080	158.25	6.9
1920	80	8924	148.72	6.64	10408	153.73	6.87
2304	96	10339	148.41	6.70	12008	141.02	6.38

n	CPC	Code rate 3/4			Code rate 5/6		
		LE	Clk_e (MHz)	Coded data rate (Gbps)	LE	Clk_e (MHz)	Coded data rate (Gbps)
576	15	4421	189.07	7.27	4295	193.23	7.54
960	23	6593	170.88	7.13	6400	174.52	7.28
1440	33	8749	165.73	7.23	8472	161.47	7.04
1920	43	11063	152.18	6.8	10704	147.43	6.59
2304	51	12727	152.65	6.89	12306	150.69	6.8

For any code rate and code length the coded data rate varies from 6.3 to 7.5 Gbps. These calculations are based on the internal encoder design and not on any special I/O limitations. The encoder implementation assumes that all input data bits are available for encoding, so I/O serialization factors are not included in the results. In order to consider the encoder implementation under I/O serialization, a shift register needs to be added as shown in Figure 5.4.

The data rate thus becomes limited by the speed at which the shift register can run, Clk_s , which is 422.12 MHz.

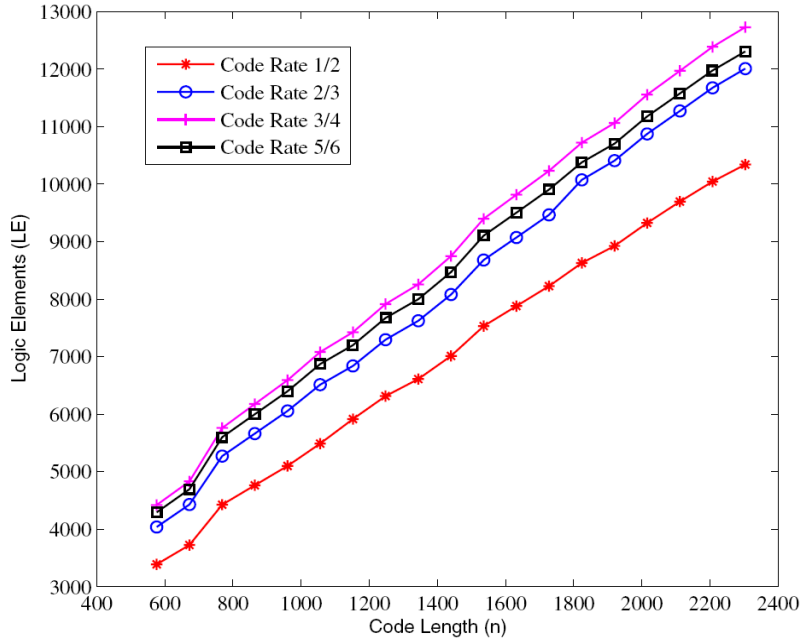


Figure 5.3: Logic elements vs. code lengths for different code rates.

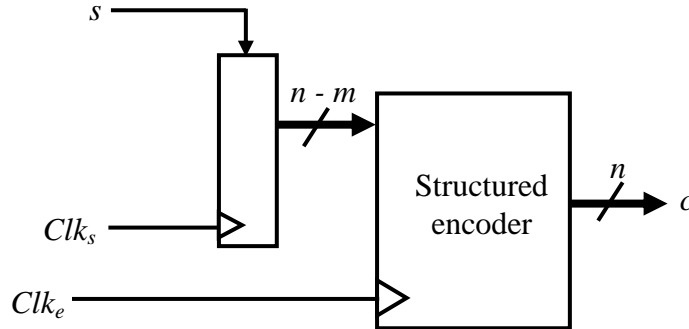


Figure 5.4: Complete structured encoder system.

The latency of the encoder considering I/O serialization is the maximum of $[m/Clk_s, CPC/Clk_e]$ which is m/Clk_s . The coded data rate of the encoder considering I/O serialization is equal to $Clk_s/(code\ rate)$. Thus, the coded data rate value is constant for different code lengths. The coded data rate is equal to 844, 633, 562 and 506 Mbps for code rates 1/2, 2/3, 3/4 and 5/6 respectively. The design methodology of our proposed encoder accommodates different code lengths and code rates. The encoder design presented can easily fit on FPGA's and has a significant high information data rates. This value is significantly high when compared to the coded data rate of the encoders presented in [32] and [33].

In [32], an encoder is implemented on a reconfigurable instruction cell architecture which is an ultra low power, high performance, ANSI-C programmable embedded core. The encoder is implemented using Richardson and Urbanke’s algorithm and LDPC codes defined in IEEE 802.16e. The encoder data rate achieved without pipelining is in the range from 10 to 19 Mbps while with pipelining it is in the range from 26 to 47 Mbps. The encoder data rate can be increased to 78 Mbps by using multiple cores.

In [33], an LDPC encoder is implemented based on Richardson and Urbanke’s method using LDPC codes defined in IEEE 802.16e and IEEE 802.11n. Their method is based on a semi-parallel architecture using cyclic right shift registers and XORs. The information data rate, which is equal to the product of coded data rate and code rate, is computed for different code lengths and is shown in Table 5.2. The *LEs* required for the implementation of our proposed structured encoder [67] and the encoder in [33] is also shown in Table 5.2. The coded data rate of 1/2 rate LDPC codes defined in IEEE 802.16e with I/O serialization of our proposed structured encoder compared with the encoder in [33] is shown in Table 5.3.

Table 5.2: Comparison of information data rate without I/O serialization of our proposed structured encoder with the encoder presented by Kim [33].

Code length	Encoder design	Information data rate (Gbps)	<i>LE</i>
576	[33]	1.55	1265
	Our proposed structured encoder [67]	3.69	3391
960	[33]	1.41	2078
	Our proposed structured encoder [67]	3.33	5100
1440	[33]	1.41	2835
	Our proposed structured encoder [67]	3.6	7012
1920	[33]	1.26	3657
	Our proposed structured encoder [67]	3.32	8924
2304	[33]	1.25	4305
	Our proposed structured encoder [67]	3.35	10339

From Tables 4.5, 5.2 and 5.3, our proposed structured encoder has highest data rate when compared with the encoders presented in [29], [32] and [33] but requires more area when compared to encoders presented in [29] and [33]. With the increase in the length of the codeword, the coded data rate of our proposed design is constant and is equal to 844 Mbps for code rate 1/2 whereas the codeword data rate of encoder presented in [33] decreases as shown in Table 5.3.

Table 5.3: Comparison of coded data rate with I/O serialization of our proposed structured encoder with the encoder presented by Kim [33].

Code length	Our proposed structured encoder coded data rate (Mbps)	Reference [33] coded data rate (Mbps)
768	844	462
1536	844	416

CHAPTER 6 - Flexible Multi-Code Rate and Multi-Code Length Encoder for Structured Low-Density Parity-Check Codes

Design methodologies presented in chapters 4 and 5 can be used for different code rates and code lengths. However, the design has to be re-synthesized in order to change the code rate or code length of the LDPC codes. In this chapter, the design of a flexible encoder for structured LDPC codes is presented. The design methodology and the implementation results are provided. The key contributions of the flexible multi-code rate and multi-code length encoder for structured LDPC codes are presented below:

- A single flexible encoder that accommodates multiple code lengths and code rates of structured LDPC codes defined in IEEE 802.16e standard is designed which does not require re-synthesis of the Verilog code in order to change the encoder parameters (code length and code rate).
- The flexible encoder for structured LDPC codes is implemented on both an FPGA and ASIC.
- The coded data rate of the synthesized encoder is 844, 633, 562 and 506 Mbps for code rates $1/2$, $2/3$, $3/4$ and $5/6$ respectively. For a given code rate, the coded data rate is constant for varying code lengths.
- The same design methodology with minor modifications can be used for other LDPC codes with structure similar to those specified in IEEE 802.16.

6.1 Design Methodology

The encoder implementation is similar to that explained in chapter 5 except that the parity-check matrices for all different code rates have to be stored on chip in order to design a flexible encoder. In this method, a flexible encoder is developed using structured LDPC codes defined in the IEEE 802.16e standard. Depending on the desired code rate and code length the corresponding parity-check matrix is computed on chip from its base parity-check matrix and is stored on chip which is used for the encoding process. This design methodology accommodates the code rates $1/2$, $2/3$, $3/4$ and $5/6$ and code lengths ranging from 576-2304 bits.

The encoder implementation is performed in four steps as shown in Figure 6.1. It is assumed that the user specifies the desired code rate and code length. In the first step, H is

computed from its corresponding H_b , and is stored on the chip temporarily for the encoding process until code length or code rate of the LDPC codes is changed. In the second step, the inner product, e_p , of the elements of H_{b1} and S are computed. In the third step, V is calculated which is the product of matrices H_{b1} and S . Parity bits are computed from V by solving Equation 5.13 in the final step.

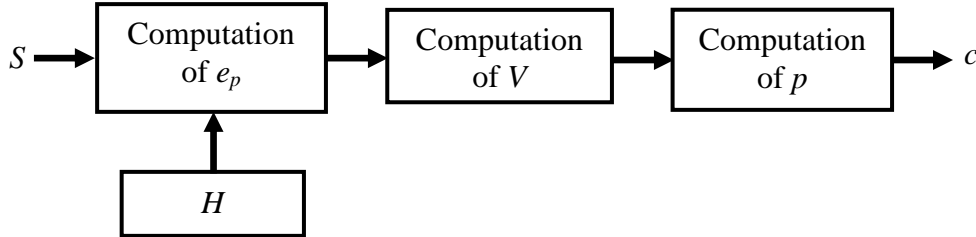


Figure 6.1: Overview of the encoding process.

6.2 Hardware Implementation

Hardware implementation of each of the blocks shown in Figure 6.1 is presented in this section.

6.2.1 Storing Base Parity-Check Matrices for Different Rates of LDPC Codes

In the IEEE 802.16e standard, there are a total of six different base parity-check matrices corresponding to the six different code rates: 1/2, 2/3 A, 2/3 B, 3/4 A, 3/4 B and 5/6. All six H_b 's are stored on chip to design a flexible encoder accommodating all different code rates. In general, the base parity-check matrices are sparse in nature. As described in section 5.1, H_b can be split into two matrices H_{b1} and H_{b2} . Because of the sparse nature of H_{b1} , only the non-negative elements are stored on the chip instead of all the elements in the matrix. In the encoding process, the V matrix is computed to obtain the parity bits. V is obtained by vector-vector multiplication of each row or column in H_{b1} with the column vector S . The inner product, e_p , of the elements of H_{b1} , and S is obtained by circularly right shifting a block of S , vector s , by a particular shift value determined by h_{b1} , so a shift register is needed to compute e_p . More details of the encoding process are presented in the next subsection. First, the best method to store H_b for efficient encoding is explored.

To maximize the efficiency, vector-vector multiplication is performed in parallel. V can be obtained using two methods as shown in Figure 6.2. In Method I, row parallelization, as shown in Figure 6.2 (a), vector-vector multiplication can be performed on each row, R , of H_{b1}

and vector S to obtain an element in V . This process can be performed in parallel on all rows of H_{b1} simultaneously to obtain all the elements of V . In Method II, column parallelization, as shown in Figure 6.2 (b), bitwise multiplication is performed on each column, C , of H_{b1} with a block of S , vector s of size $z \times 1$, in parallel and then all the product vectors are added to obtain V . All the base parity-check matrices for different code rates are evaluated to determine the best method for the implementation of vector-vector multiplication.

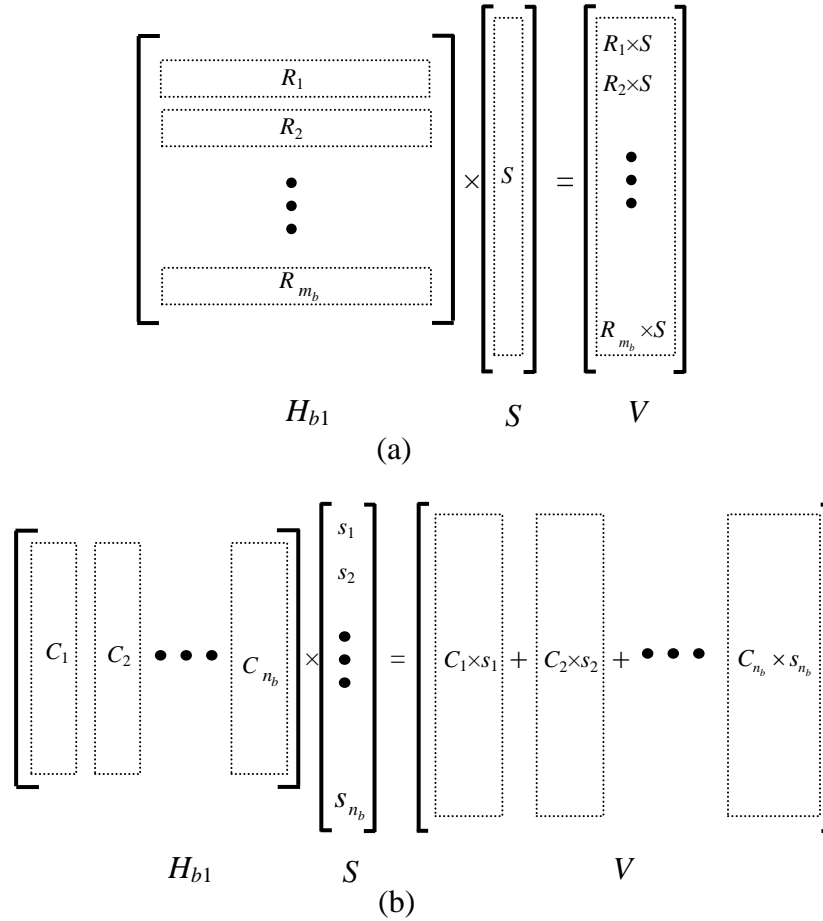


Figure 6.2: Computation of V using (a) row parallelization method and (b) column parallelization method.

In the base parity-check matrices for all different code rates there are a maximum of 6 and 18 non-negative elements in each column and row respectively. Computation of V using row parallelization method would require 18 instantiations of a shift register whereas using column parallelization method would require only 6 instantiations of a shift register. Computation of V by using the row parallelization method would require the entire vector S , while only a block of S of size $z \times 1$ is required when column parallelization method is used. Implementation of the

column parallelization method would require less area than the row parallelization method while the latency involved in reading the information bits is also decreased. Hence for computation of V , the column parallelization method is chosen.

To design an LDPC encoder that is flexible with code rate, six base matrices of H_{b1} corresponding to code rates 1/2, 2/3 A, 2/3 B, 3/4 A, 3/4 B and 5/6 need to be stored on the chip. To store each H_{b1} using sparse representation, six non-negative elements per column of H_{b1} are required. The information needed to store an element is its location (i.e. row number) and its value. The maximum value of an element in H_{b1} is 95 which require 7 bits for representation. Matrix H_{b1} for code rates 1/2, 2/3, 3/4 and 5/6 has 12, 8, 6 and 4 rows respectively, which require 4 bits for its representation of a maximum value 12. Therefore, a total of 11 bits are used to represent an element in H_{b1} .

Each column of H_{b1} is stored in an array. The values are stored in registers instead of RAM modules available on an FPGA so that the same design can be implemented on an ASIC without any modification. Each element in this array represents the concatenation of all the non-negative elements in each column of H_{b1} . As mentioned above, there are a maximum of 6 non-negative elements in each column of H and each element in H requires 11 bits for its representation. Therefore the size of an element in the H_{b1} array is 66 bits. As an example, to store H_{b1} of code rate 2/3 B requires: non-negative elements in the first column of H_{b1} located at row locations 1, 3, 5 and 7, and their corresponding values are 2, 10, 23 and 32 respectively. This can be denoted as (1, 2), (3, 10), (5, 23) and (7, 32). Six elements are stored per column. If any column has non-negative elements less than 6 then the remaining elements are denoted as (0, 127). The size of the H_{b1} array is equal to the number of columns in H_{b1} . For code rates 1/2, 2/3, 3/4 and 5/6 the size of the H_{b1} array is 12, 16, 18 and 20 respectively. For example, the size of H_{b1} with code rate 2/3 is 16×1 where an element in H_{b1} is 66-bits long.

The H_{b2} matrix for all different code rates has the same pattern except for the location of non-negative elements in its first column. The first column in H_{b2} has 3 non-negative elements. Two of these non-negative elements are located on the top and bottom of the first column and are assigned equal shift values. The third non-negative element is located anywhere in the middle of the column. Also one of the non-negative element's shift value is equal to zero. For the encoding process, a non-negative element with shift value greater than zero is only needed from the first column of H_{b2} . Therefore, one non-negative element's shift value and its location are stored

instead of the entire first column. For all possible code rates and code lengths its corresponding scaling value is computed and is stored in a look up table.

6.2.2. Parity-Check Matrix

The base parity-check matrix is defined for the largest code length ($n = 2304$) for each code rate. The set of shift values, $h_{b(i,j)}$, in the H_b are used to determine the shift sizes, $h_{b(i,j)}$, of H for varying code lengths of the same code rate. Each H_b has n_b columns equal to 24, and the expansion factor z_f is equal to $n/24$ for code length n . For example, code length n equal to 2304 has the expansion factor z_f of 96. For code rates 1/2, 2/3 B, 3/4 A, 3/4 B, and 5/6, the shift sizes, $h_{b(i,j)}$, of H for a code length corresponding to expansion factor z_f are derived from $h_{b(i,j)}$ by scaling $h_{b(i,j)}$ proportionally as

$$h_{(i,j)} = \begin{cases} h_{b(i,j)}, & h_{b(i,j)} \leq 0 \\ \left\lfloor \frac{h_{b(i,j)} z_f}{z_0} \right\rfloor, & h_{b(i,j)} > 0 \end{cases} \quad (6.1)$$

where $\lfloor w \rfloor$ denotes the floor of w and $z_0 = 96$. For the code rate 2/3 A, the shift sizes, $h_{b(i,j)}$, of H for a code length corresponding to expansion factor z_f is defined using the modulo function

$$h_{(i,j)} = \begin{cases} h_{b(i,j)}, & h_{b(i,j)} \leq 0 \\ \text{mod}(h_{b(i,j)}, z_f), & h_{b(i,j)} > 0 \end{cases} \quad (6.2)$$

For any given code rate and code length, the parity-check matrix needs to be computed only once. All six H_{b1} matrices are stored on the chip as shown in Figure 6.3.

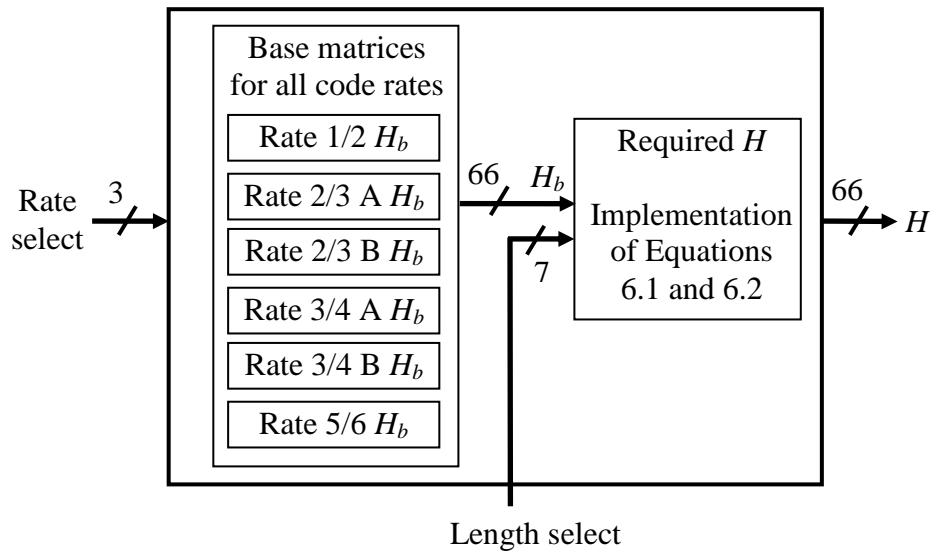


Figure 6.3: Storing base parity-check matrices, H_b , for different code rates.

Based on the desired code rate and code length, the rate select and length select inputs are chosen. Rate select values for different code rates are shown in Table 6.1. The length select input is equal to $n/24$ where n is the code length and length select values for varying code lengths are shown in Table 6.2.

Table 6.1: Rate select values for different code rates.

Code rate	Rate select
1/2	001
2/3 A	010
2/3 B	011
3/4 A	100
3/4 B	101
5/6	110

Table 6.2: Length select values for different code lengths.

Code length	Length select
576	0011000
672	0011100
768	0100000
864	0100100
960	0101000
1056	0101100
1152	0110000
1248	0110100
1344	0111000
1440	0111100
1536	1000000
1632	1000100
1728	1001000
1824	1001100
1920	1010000
2016	1010100
2112	1011000
2208	1011100
2304	1100000

The required H is computed from its corresponding H_b using Equations 6.1 and 6.2 as shown in Figure 6.3. Equations 6.1 and 6.2 are implemented using simple multiplication and division modules.

6.2.2.1 Multiplication

The multiplication of two unsigned binary integers, In_1 and In_2 , each of length 7 bits creates a product, Out , of length 14 bits. The finite state machine of the multiplication module is shown in Figure 6.4.

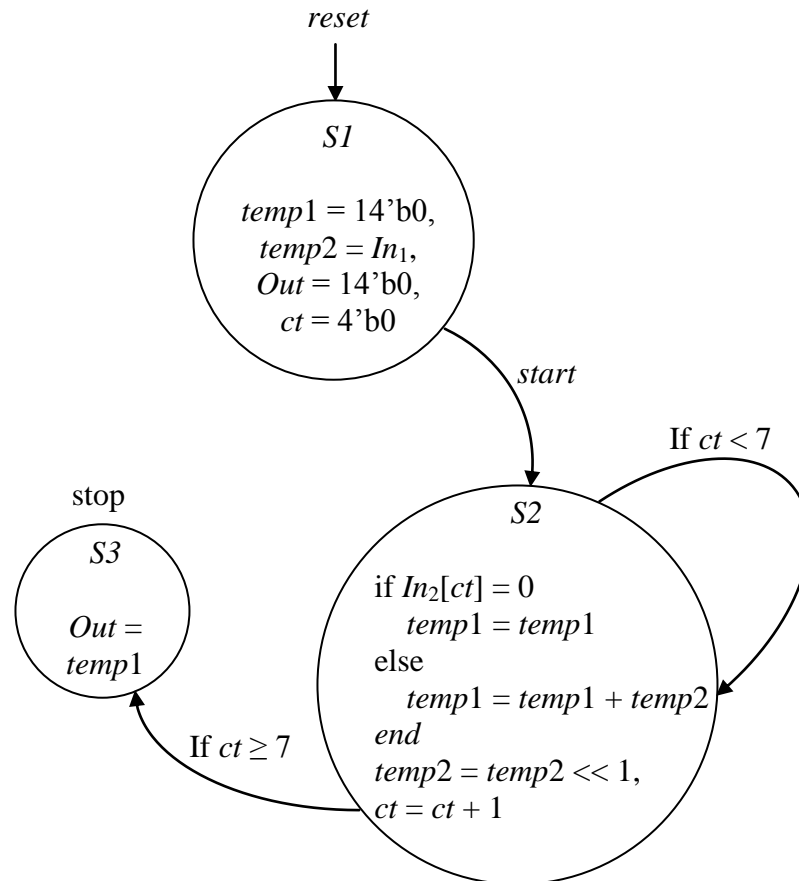


Figure 6.4: Finite state machine for the multiplication module.

The multiplication process is controlled by the input, $start$, as shown in Figure 6.4. If the input, $start$ is 0 then the machine stays in state $S1$ where all the values used in the multiplication process are initialized. When $start$ is equal to 1, the multiplication process begins by loading the inputs In_1 , multiplicand, and In_2 , multiplier. The variable $temp2$ is assigned the value of the multiplicand, In_1 , and counter ct is initialized to zero and then the state machine is moved to state $S2$. In state $S2$, the multiplier bit located at ct is obtained. If $In_2[ct]$ value is 1 then the variable

$temp1$ is added to $temp2$ and the sum is assigned to $temp1$ otherwise $temp1$ remains the same. The ct is incremented by 1 and $temp2$ is shifted left by one bit. If the value of ct is less than 7 then the state machine remains in the same state $S2$. Otherwise, it is moved to state $S3$. In state $S3$, the product, Out , is assigned the value of $temp1$ and the multiplication process is stopped.

The hardware block diagram of the multiplication module is shown below in Figure 6.5.

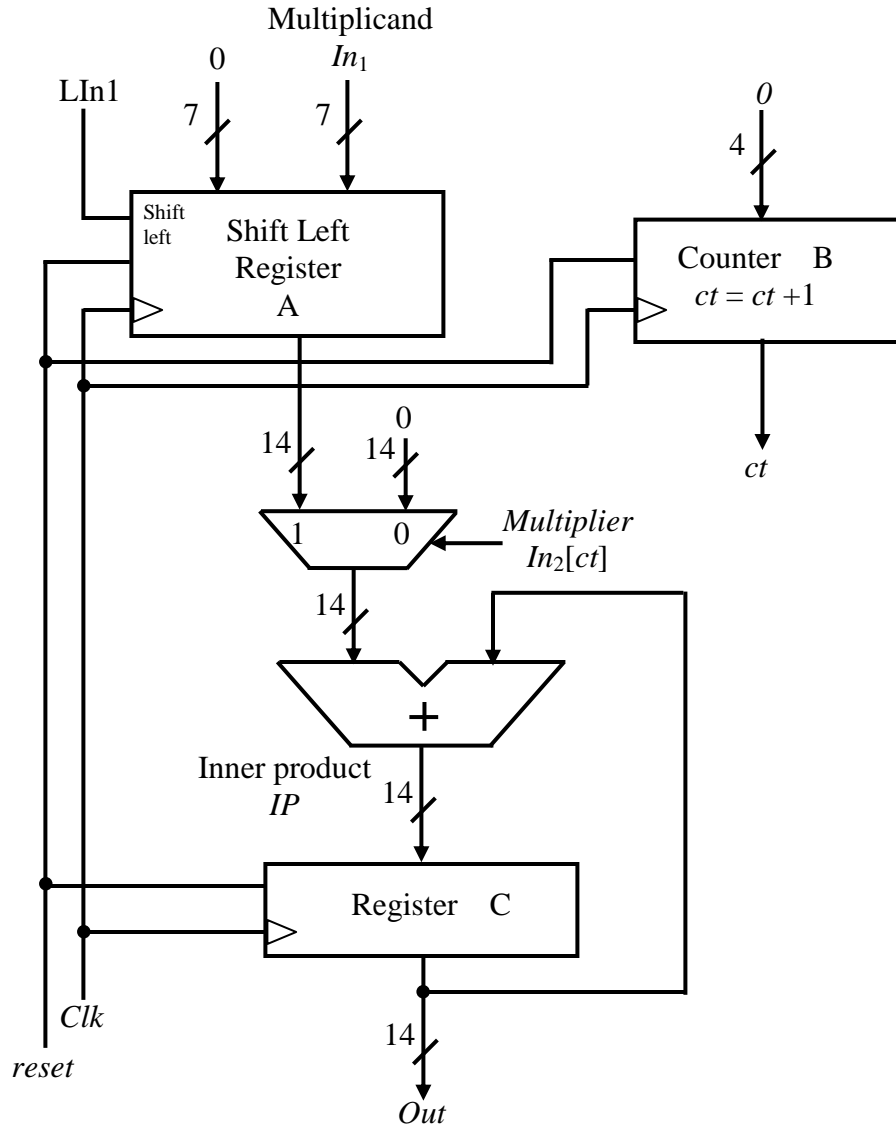


Figure 6.5: Hardware block diagram for the multiplication module.

First step is to initialize all the values. Since the product is 14-bit long, the most significant bits of the multiplicand, In_1 , are assigned 7 zeros. For every clock cycle, the multiplicand, In_1 , is shifted left by one bit and the counter ct is incremented by 1. The multiplier, In_2 , bit located at ct controls the multiplexer output. The multiplexer output is assigned the output of shift register A

or 14-bit zeros when the $In_2[ct]$ is equal to 1 or 0 respectively. The inner product, IP , is obtained by adding the multiplexer output and the product, Out , of the register C. This multiplication module requires 7 clock cycles to obtain the final product, Out .

6.2.2.2 Division

The division of two unsigned binary integers, In_1 and In_2 , each of length 14 bits creates a quotient, Q , and remainder, R , each of length 14 bits. The finite state machine of the division module is shown in Figure 6.6 and its hardware block diagram is shown in Figure 6.7.

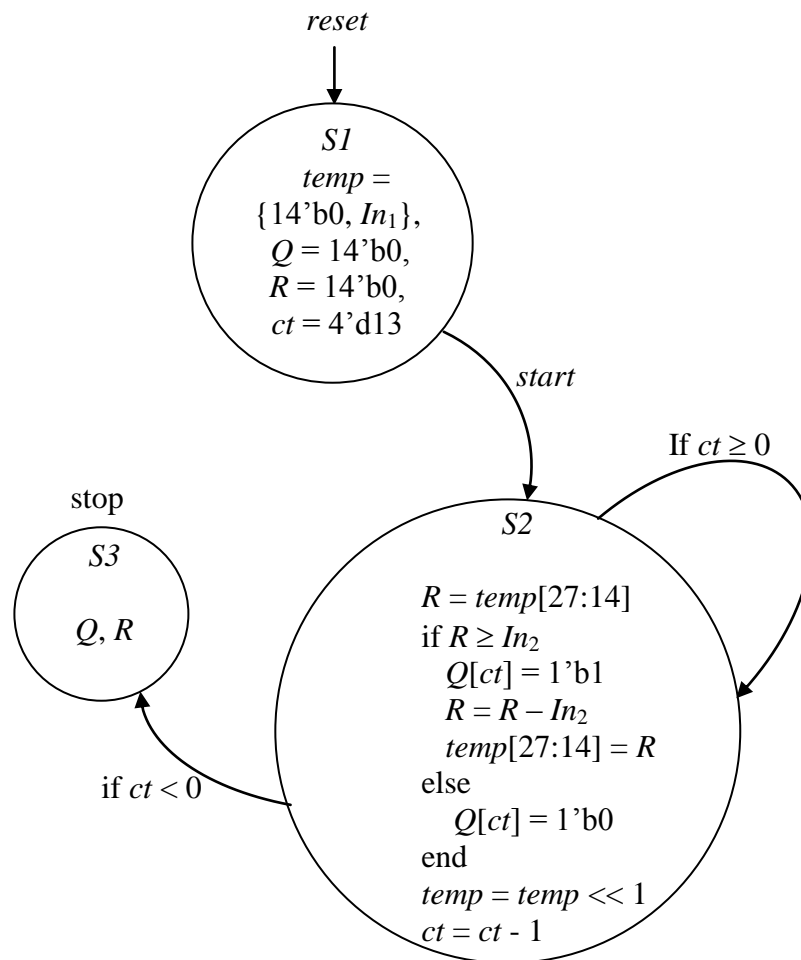


Figure 6.6: Finite state machine for the division module.

The division process is controlled by the input, $start$, as shown in Figure 6.6. If the $start$ input is 0 then the machine stays in state $S1$ where all the values used in the division process are initialized. When $start$ is equal to 1, the division process begins by loading the inputs In_1 , dividend, and In_2 , divisor. The variable $temp$ is assigned a value equal to the concatenation of

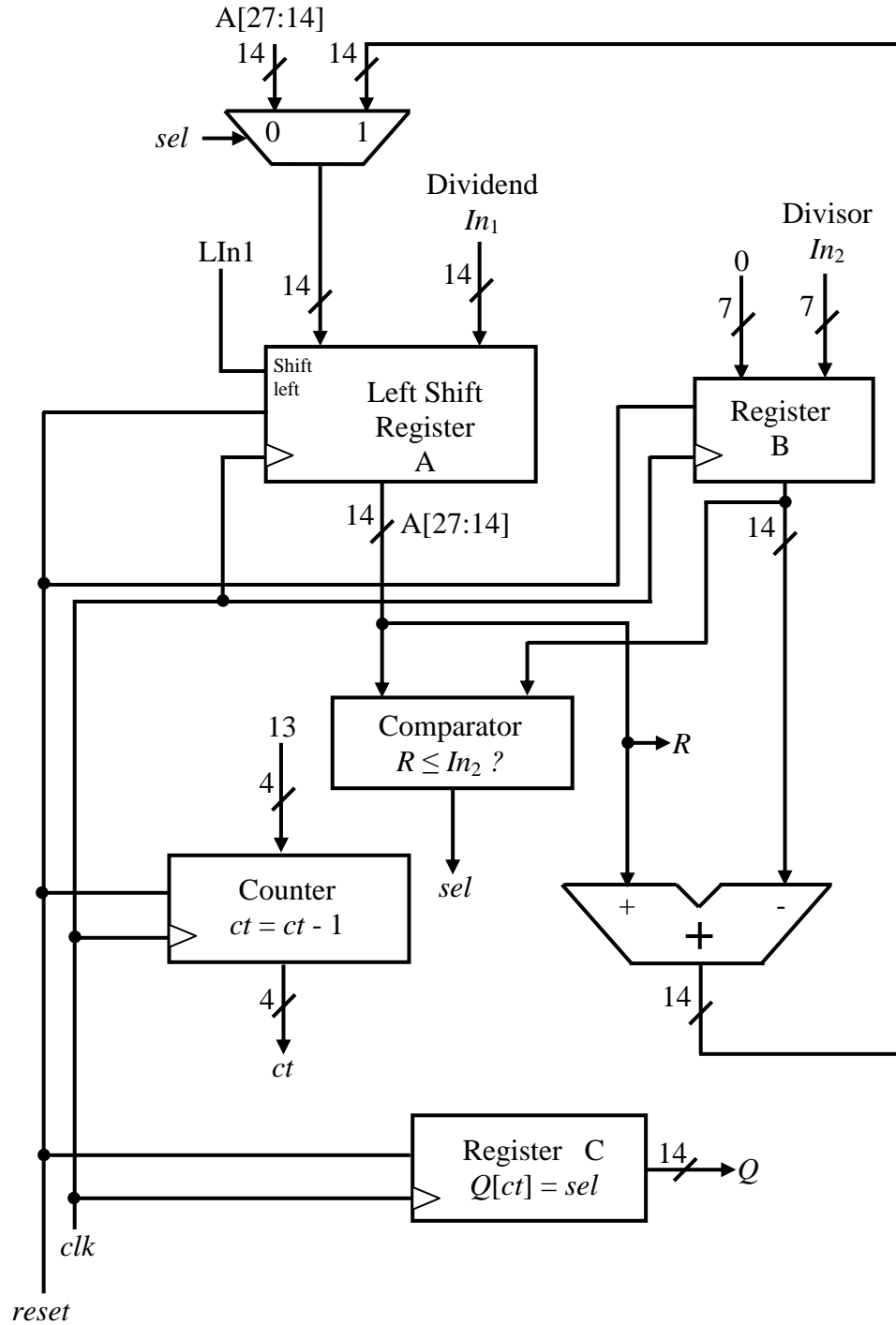


Figure 6.7: Hardware block diagram for the division module.

14-bit zeros and In_1 . The counter ct is assigned a value equal to 13 and the state machine is moved to the state $S2$. In state $S2$, R is assigned a value equal to $temp[27:14]$. If R is greater than or equal to In_2 then the Q bit located at ct is assigned a value of 1 and R is assigned a new value equal to $R - In_2$. The $temp[27:14]$ bits are reset with the updated value of R . If R is less than In_2 ,

then the Q bit located at ct is assigned a value of 0. The variable $temp$ is shifted left by one bit and ct is decremented by 1. If the value of ct is greater than or equal to zero then the state machine remains in the same state $S2$ otherwise it is moved to state $S3$. In state $S3$, the division process is stopped and the quotient and remainder are obtained.

First step in the hardware block diagram of the division module as shown in Figure 6.7 is to load inputs dividend, In_1 , and divisor, In_2 in left shift register A and register B respectively. Variable ct is assigned a value of 13. For each clock cycle, In_1 , is shifted to the left by one bit. The 14 most significant bits of the left shift register A is equal to the remainder, R , of the division module. For every clock cycle, R is compared with In_2 . If $R \geq In_2$ then output of the comparator, sel , is assigned a value equal to 1. Otherwise, sel is assigned a value of 0. Register C stores the quotient value where $Q[ct]$ is equal to sel . The output of the multiplexer is assigned to the 14 most significant input bits of the left shift register A. The output of the multiplexer is equal to $R - In_2$ or $A[27:14]$ when the value of sel is 1 or 0 respectively. For every clock, counter, ct , is decremented by one. When ct reaches a value of 0 then the division process is completed. This module requires 14 clock cycles to produce the quotient, Q , and the remainder, R .

6.2.2.3 Computation of H_1

As explained in section 6.2.1, only the elements of H_{b1} whose values are greater than or equal to zero are stored on the chip. From Equations 6.1 and 6.2, each element of parity-check matrix is computed from the base parity-check matrix and is implemented as shown in Figure 6.8. For code rates 1/2, 2/3 B, 3/4 A, 3/4 B and 5/6 the H_1 is computed using multiplication and division modules using Equation 6.1 as shown in Figure 6.8. For a desired code rate the corresponding base parity-check matrix element, $h_{b1(i,j)}$, is multiplied with the corresponding expansion factor, z_f , and this product is then divided by $z_0 = 96$ to obtain the value of the element in H_1 , $h_{1(i,j)}$. The value of the element $h_{1(i,j)}$ is equal to the quotient of a division module. For code rate 2/3 A, the modulus function in Equation 6.2 is implemented using a division module as shown in Figure 6.8. For a desired code rate the corresponding base parity-check matrix element, $h_{b1(i,j)}$, is divided by the corresponding expansion factor, z_f , to obtain the value of the element in H_1 , $h_{1(i,j)}$. The value of the element $h_{1(i,j)}$ is equal to the remainder of the division module. Based on the desired code rate an appropriate *select* value is chosen as shown in Figure 6.8.

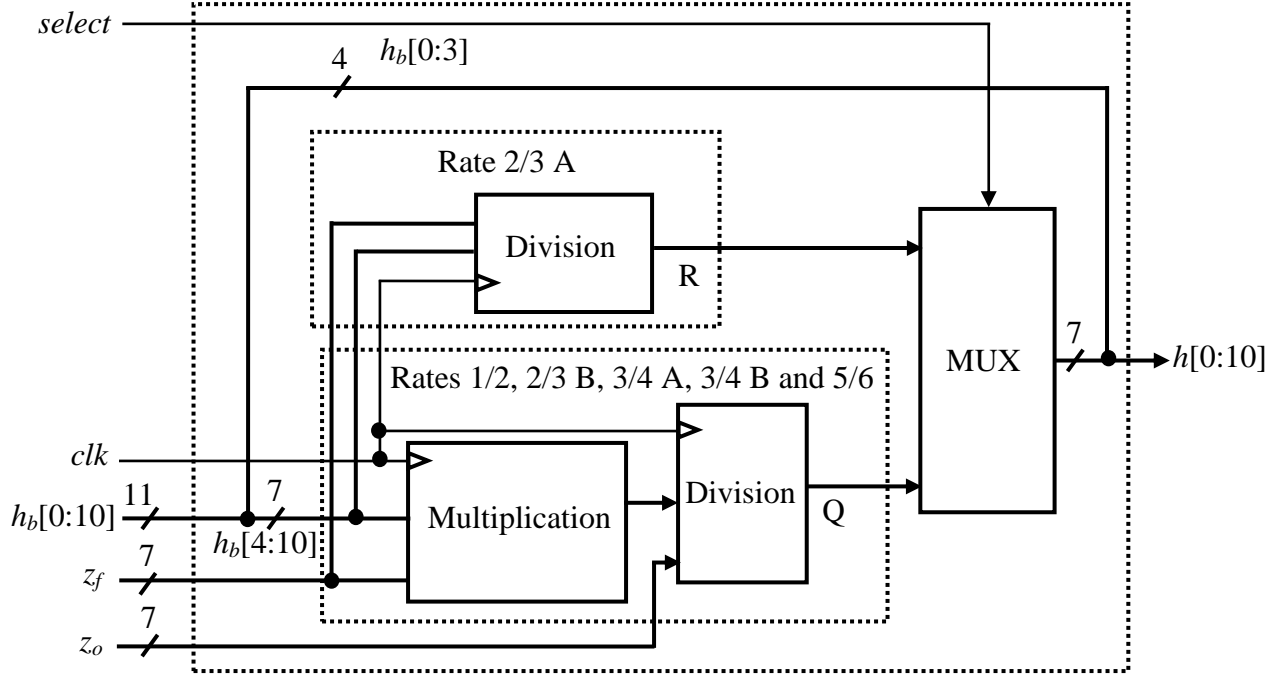


Figure 6.8: Computation of an element of H_1 , $h_{1(i,j)}$, from an element of H_{b1} , $h_{b1(i,j)}$.

The element of H_1 obtained using Equations 6.1 and 6.2 is either equal to the quotient, Q , or the remainder, R , of the division module as shown in Figure 6.8. The quotient and remainder obtained using the division module presented in subsection 6.2.2.2 are each of length 14 bits. The element value in any H_1 does not exceed 96 and therefore 7 bits are sufficient to represent its value. Therefore, 7 least significant bits of the quotient and remainder are sufficient and are only used for the computation of the elements of H_1 .

There are 6 non-negative elements in each column of H_{b1} as explained in subsection 6.2.1. Therefore 6 instantiations of the design that computes an element of H_1 as shown in Figure 6.8 are required to compute all the six elements that are located in each column of H_1 . In one clock cycle, elements located in a column of H_1 , h_1 , is computed from corresponding elements in each column of H_{b1} , h_{b1} , as shown in Figure 6.9.

6.2.2.4 Latency for the Computation of H

For the computation of a single element in H_1 from its corresponding H_{b1} requires 23 clock cycles i.e., 8 clock cycles to perform multiplication and 15 clock cycles to perform division. This is the case for code rates 1/2, 2/3 B, 3/4 A, 3/4 B and 5/6 whereas for code rate 2/3 A computation of an element of H_1 requires 15 clock cycles because it only requires the division operation. The number of clock cycles required for the computation of all the elements in H_1

from H_{b1} is shown in Table 6.3. The number of elements in H_{b1} array is equal to the number of columns of H_{b1} . For code rates 1/2, 2/3, 3/4 and 5/6 the number of elements in H_{b1} array are 12, 16, 18 and 20 respectively.

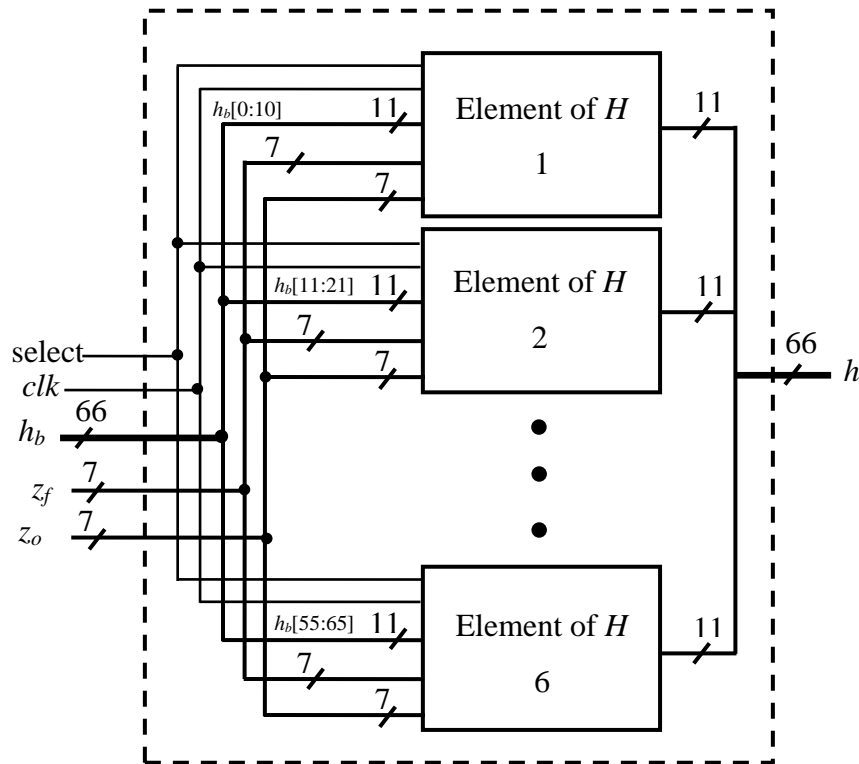


Figure 6.9: Computation of a column of H_1 from a column of H_{b1} .

Table 6.3: Number of clock cycles required for computation of H_1 from H_{b1} for different code rates.

Code rate	No. of elements in H_{b1} array	No. of clock cycles required per element in H_{b1} array	Total No. of clock cycles required for computation of H_1
1/2	12	23	276
2/3 A	16	15	240
2/3 B	16	23	368
3/4 A	18	23	414
3/4 B	18	23	414
5/6	20	23	460

The total number of clock cycles required for the computation of all the elements of H_1 from H_{b1} is equal to the number of clock cycles required per element in H_{b1} array times the number of elements in H_{b1} array.

Only one element value of H_2 is required for encoding process. Its value for different code rates and code lengths is computed and stored in a look up table which can be obtained in one clock cycle. Therefore the total number of clock cycles required for obtaining H is equal to the total number of clock cycles required for computation of H_1 . The latency involved in the computation of H is equal to product of the total number of clock cycles required for computation of H and time period of the clock.

6.2.3 Computation of e_p

The first step in the encoder implementation is the computation of V , the product of matrices H_{b1} and S . The matrix H_1 is a sparse matrix with each element representing either a zero matrix, identity matrix or permutation matrix of size $z \times z$. Vector e_p is defined as the product of an element of H_1 , h_1 , which is a matrix of size $z \times z$, and an element is S , s , which is a vector of size $z \times 1$. The product vector, e_p , is obtained by circularly right shifting the vector s by a particular shift value determined by h_1 . As shown in Equation 6.3, e_p is equal to 0 or s (itself) or s_f if h_1 is equal to -1 or 0 or f respectively.

$$e_p = h_1 s = \begin{cases} 0, & \text{if } h_1 = -1, \\ s, & \text{if } h_1 = 0, \\ s_f, & \text{if } h_1 = f. \end{cases} \quad (6.3)$$

Vector s_f is the circular right shifted version of the vector s and the circular right shifted value f is defined by h_1 .

Figure 6.10 shows the computation of e_p . The code length of the LDPC codes varies from 576-2403. The base matrix has 24 columns. Therefore the size of each element in the base matrix, z , vary from 24 – 96 (i.e., 576/24 - 2304/24). In order to accommodate different code lengths of LDPC codes, the size of the shift register is chosen to be 96. The shift register is hardcoded for all possible shift values (0-95), so that the circular right shifted version of s is obtained in one clock cycle. This kind of implementation will occupy more area than a single shift register, but will ultimately achieve high encoding data rates. In one clock cycle, a 66-bit element from H_1 and a 96-bit block of s are read. Depending on code length, the size of s may vary from 24-96. If the size of s is less than 96, then the remaining bits are assigned zeros. As

shown in Figure 6.10, six circular right shift operations are performed by six shift registers in parallel whose shift values and locations are obtained from an element of H_1 . Each of the e_p obtained from the circular right shift register is assigned to one of 12 outputs of the demultiplexer based on its location (row number). For example, if the row number is equal to 3 then e_p is assigned to the third output of the demultiplexer. All of the demultiplexer outputs are added to obtain the inner products $e_{p1}, e_{p2}, \dots, e_{p12}$.

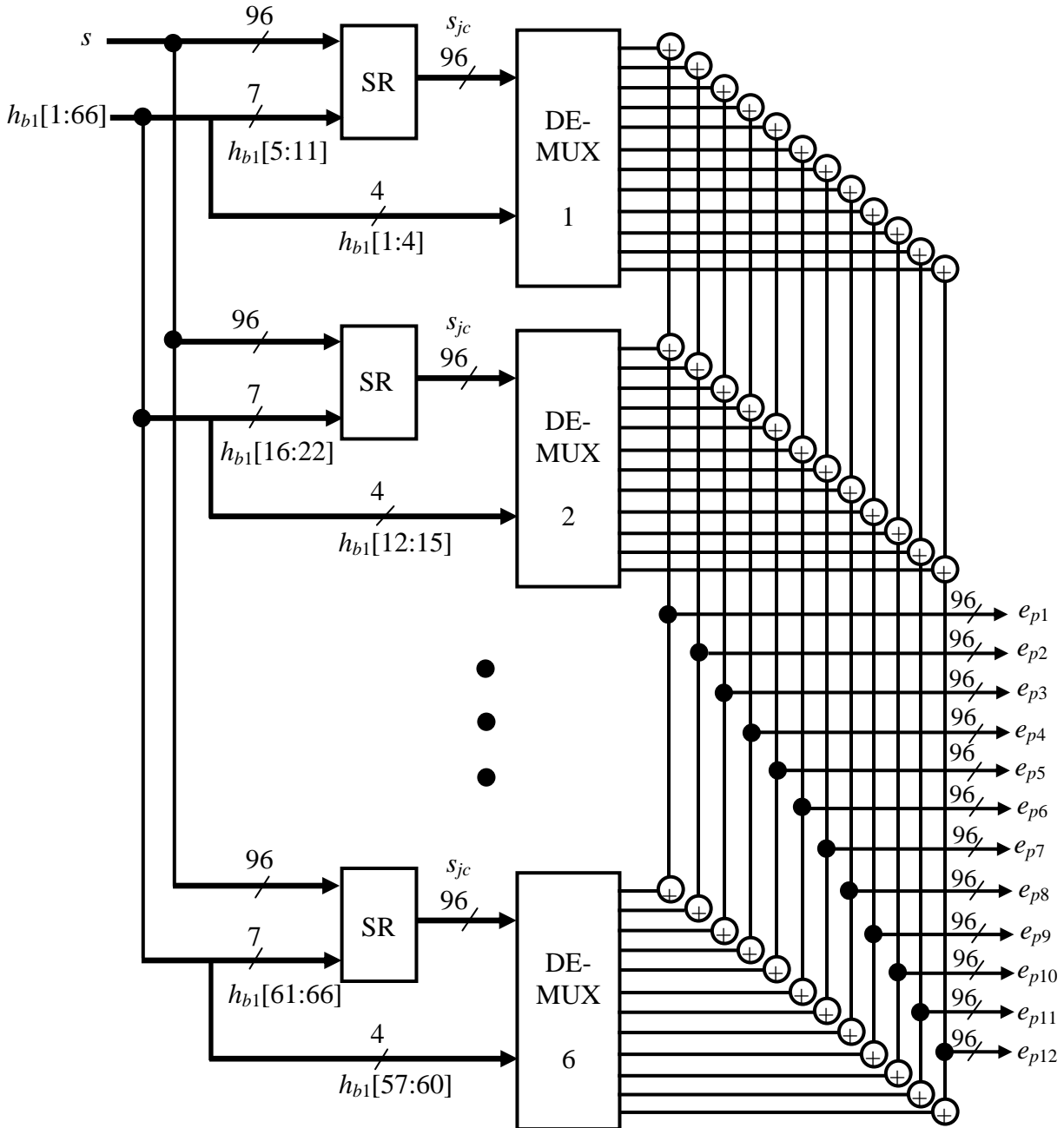


Figure 6.10: Computation of e_p .

6.2.4 Computation of V

The third step in the encoding process is the computation of V , the product of matrices H_{b1} and S . An element of the vector $V (v_i)$ is obtained, from

$$v_i = \sum_{j=1}^k e_{pi} \quad \text{where } k = \begin{cases} 12 & \text{if code rate} = 1/2 \\ 16 & \text{if code rate} = 2/3 \\ 18 & \text{if code rate} = 3/4 \\ 20 & \text{if code rate} = 5/6 \end{cases} \quad (6.4)$$

and is shown in Figure 6.11.

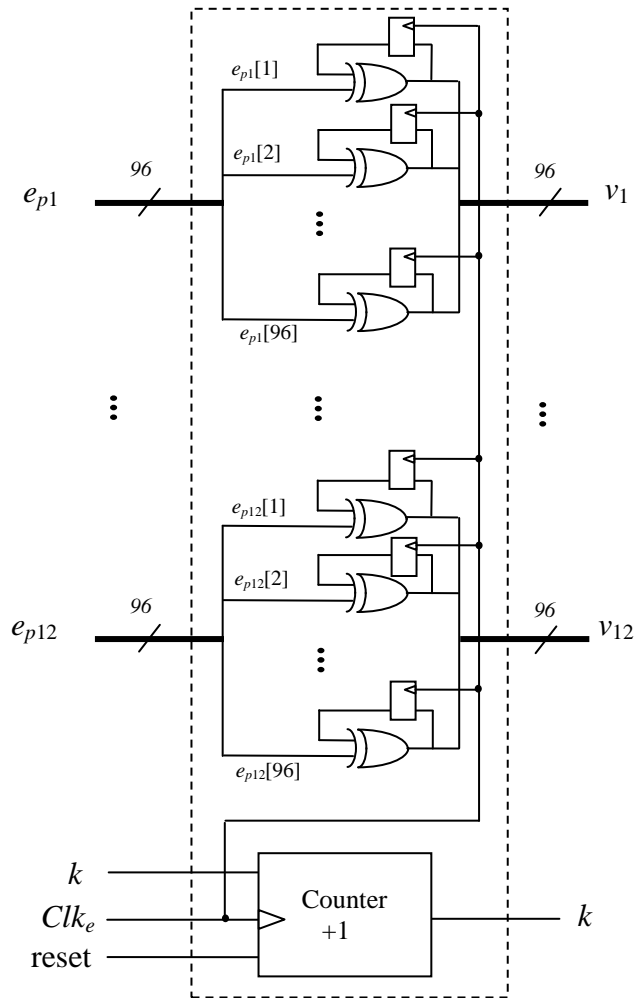


Figure 6.11: Computation of V .

v_i is obtained by adding the inner product (e_p) 12, 16, 18 and 20 times for code rates 1/2, 2/3, 3/4 and 5/6, respectively. In modulo 2, addition is performed using the XOR gate.

6.2.5 Computation of Parity Bits

The final step in the encoding process includes the computation of parity bits. Once V is computed, parity bits are obtained by solving Equation 5.13. Solving Equation 5.13, we get $h_b(0)p_0 + p_1 = v_0$, $p_1 + p_2 = v_1$, ... , $h_b(j)p_0 + p_j + p_{j+1} = v_j$, ..., $h_b(m_b - 1)p_0 + p_{m_b - 1} = v_{m_b - 1}$. Adding all the Equations, one obtains $p_0 = v_0 + v_1 + \dots + v_{m_b - 1}$. The addition is performed by XORing all the elements of v . This is the case for all the code rates except for code rate 3/4 B. For code rate 3/4 B, $h_b(0)p_0 = v_0 + v_1 + \dots + v_{m_b - 1}$. p_0 is obtained by circularly right shifting the sum of $v_0 + v_1 + \dots + v_{m_b - 1}$ by a value equal to $z_f - h_b(0)$. This step can be performed in a single clock cycle by using an additional shift register. Once p_0 is obtained, the remaining parity bits can be computed from solving the above expressions i.e., $p_1 = v_0 + h_b(0)p_0$, $p_2 = v_1 + p_1$ etc.,. To obtain p , all the parity expressions shown above are hard coded in the design for all different code rates. Hence parity bits are computed in one clock cycle.

6.3 Results of the Flexible Structured Encoder Implemented on an FPGA

A hardware implementation was performed on an Altera Stratix EP1S80F1508C5 FPGA using Quartus II for synthesis. Verilog modules generated from Matlab scripts were used for the implementation. The results are shown in Table 6.4. Due to the restriction on the number of input/output pins on the FPGA the code length is restricted to the range of 576-2016. The number of logic elements required for the implementation of the encoder on the FPGA are 34,100 (43%). Of the two clock signals being used, Clk , is a faster clock used to compute the required H_1 and is equal to 69.76 MHz. The other clock, Clk_e , is a slower clock used for the computation of the parity bits and is equal to 27.23 MHz.

In order to accommodate all the code lengths (576-2304) on the chip the number of output pins is reduced and the design is re-synthesized. To reduce the number of output pins, sum of the parity bits is read instead of individual parity bits. It is observed that this design occupies 40936 (52%) LEs which is more than that of the earlier design implementing only code length from 576-2016. The increase in the LEs is due to the addition operation performed on the parity bits. The clock frequencies Clk and Clk_e are 77.10 MHz and 26.65 MHz respectively. It can be concluded that if the design is synthesized on a larger chip with a large number of input and output pins, then the encoder design with more code lengths can be accommodated. Also, the

design would require less than 40936 *LEs* and operate with the same clock frequencies. The lowest values of Clk and Clk_e are considered for the coded data rate and latency computations.

In Table 6.4, *CPC* represents the number of clock cycles required per codeword for encoding. *CPC* is equal to number of clock cycles required for computation of V and p . The number of clock cycles required to compute V is equal to the number of columns in H_{b1} for a given code rate. Computation of p requires one clock cycle. Hence this method requires 13, 17, 19 and 21 clock cycles for code rates 1/2, 2/3, 3/4 and 5/6 respectively. Column Clk_e in Table 6.4 represents the encoder clock which can run at 26.65 MHz for any code rate and code length.

Table 6.4: Synthesis results of the flexible encoder for structured LDPC codes.

Code rate	Clk_e (MHz)	<i>CPC</i>	m	Coded data rate (Mbps)
1/2	26.65	13	288 – 1152	1180 – 4724
2/3	26.65	17	384 – 1536	903 – 3612
3/4	26.65	19	432 – 1728	808 – 3232
5/6	26.65	21	480 – 1920	730 – 2924

The latency involved in the computation of H from H_b is equal to product of the total number of clock cycles required for computation of H and the clock time period. The latency involved in computing H for different code rates is shown in Table 6.5. The synthesized Clk frequency for computation of H is 69.76 MHz. So the time period of the clock is 14.34 ns. For any given code rate and code length, computation of the required H is done only once. Therefore the latency involved in computation of the parity-check matrix is not included in the coded data rate calculations.

Table 6.5: Latency involved in computation of H for different code rates.

Code rate	Latency involved in computation of H (μ s)
1/2	3.958
2/3 A	3.442
2/3 B	5.277
3/4 A	5.937
3/4 B	5.937
5/6	6.596

The coded data rate is equal to $m \times Clk_e / (CPC \times code\ rate)$. The coded data rate decreases with an increase in the code rate and increases with increase in the code length as shown in Table 6.4. For a code length of 576, the coded data rate ranges from 1180-730 Mbps for different code rates. When the code length is increased to 2304, the coded data rate increases and is in the range of 4724-2924 Mbps. These calculations are based on the internal encoder design and not on any special I/O limitations. The encoder implementation assumes that all input data bits are available for encoding, so serialization factors are not included in the results. In order to consider the encoder implementation under serialization, a shift register needs to be added as shown in Figure 6.12. The coded data rate thus becomes limited by the speed at which the shift register can run, Clk_s , which is 422.12 MHz.

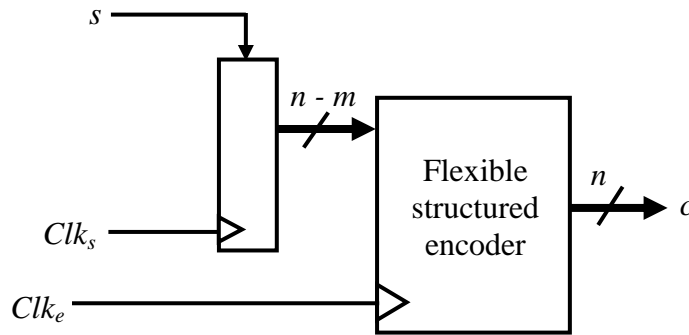


Figure 6.12: Complete system of the flexible multi-code rate and multi-code length structured LDPC encoder.

The latency of the encoder considering I/O serialization is the maximum of $[m/Clk_s, CPC/Clk_e]$ which is m/Clk_s . The coded data rate of the encoder considering I/O serialization is equal to $Clk_s / (code\ rate)$. Thus, the coded data rate value is constant for different code lengths. The coded data rate is 844, 633, 562 and 506 Mbps for code rates 1/2, 2/3, 3/4 and 5/6 respectively. This value is significantly high when compared to a coded data rate of range 10-19 Mbps obtained for same LDPC codes [32]. From Tables 4.5, 5.3 and 6.4 it can be observed that the proposed encoder has very high coded data rate when compared to the encoders in [29] and [33]. In [29], for code length of 2000 and code rate 1/2, it is shown that by using 16 encoder instances instead of one encoder instance, the coded data rate is increased from 45 Mbps to 410 Mbps which requires 16906 slices. In this case, the equivalent logic elements are 42265. The coded data rate of our flexible structured encoder is equal to 844 Mbps which is more than twice the coded data rate of the encoder presented in [29] while requiring less area than the encoder in [29]. A single design accommodates different code lengths and code rates. Re-synthesis of the

code is not required in order to change code rate or code length. The encoder design presented can easily fit on FPGAs and has a high coded data rate. The flexible structured encoder is also implemented on ASIC. The details are presented in the next section.

6.4 Implementation of a Flexible Multi-Code Rate and Multi-Code Length Structured Encoder on an ASIC

Implementation of the flexible multi-code rate and multi-code length structured encoder on an ASIC is performed using Cadence. An ASIC is designed using the standard cell library provided by Virginia Polytechnic Institute and State University [63]. The same Verilog design that is used for the FPGA implementation is also used for the implementation of the ASIC. The Verilog design is synthesized in Cadence RTL by following the procedure presented in section C.2 of appendix C. The synthesized design in RTL Compiler is shown in Figure 6.13. The synthesized results of the flexible multi-code rate and multi-code length structured encoder are shown in Table 6.6.

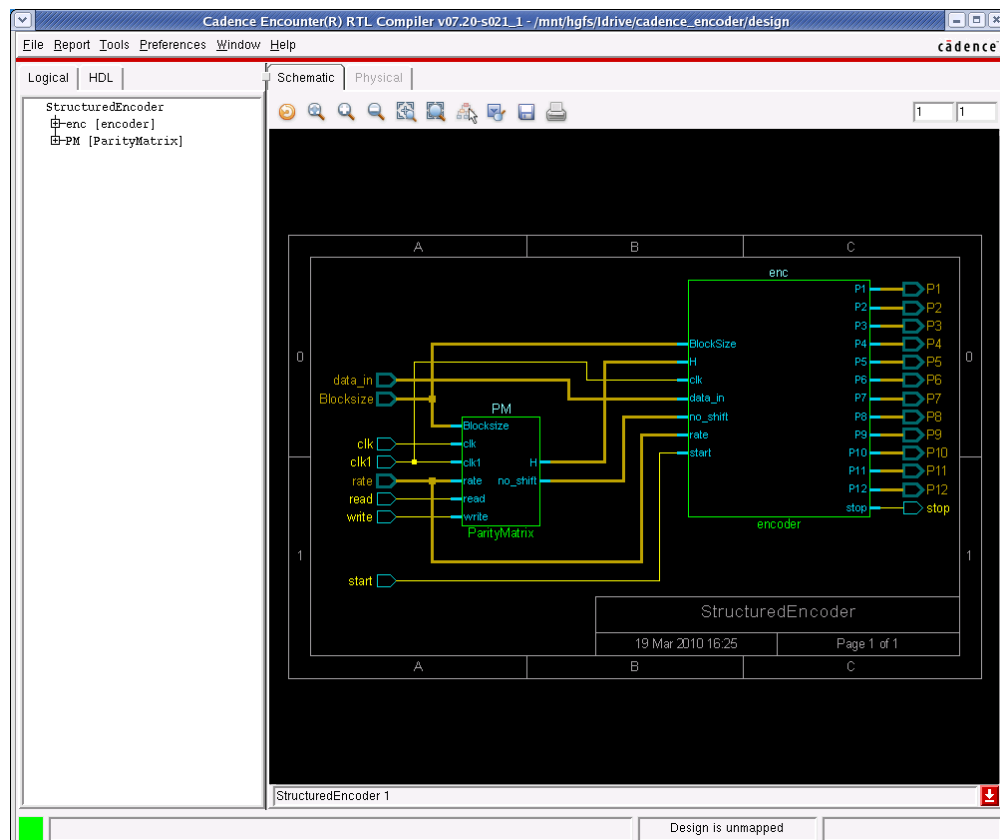


Figure 6.13: Synthesized flexible multi-code rate and multi-code length LDPC structured encoder in Cadence RTL Compiler.

Once the design is synthesized it is then placed and routed in Cadence Encounter following the procedure presented in section C.3 of appendix C. The final layout of the design in Encounter is shown in Figure 6.14. The design is saved in GDS format. Figure 6.15 shows the imported layout of the encoder in Cadence ICFB.

Table 6.6: Synthesis results of flexible multi-code rate and multi-code length LDPC encoder in Cadence RTL Compiler.

Parameter	Value
Code length	576 – 2304 bits
Code rate	1/2, 2/3 A, 2/3B, 3/4 A, 3/4 B, 5/6
Technology	0.25 μm
Gate count	116.5 K
Clock frequency	215.66 MHz

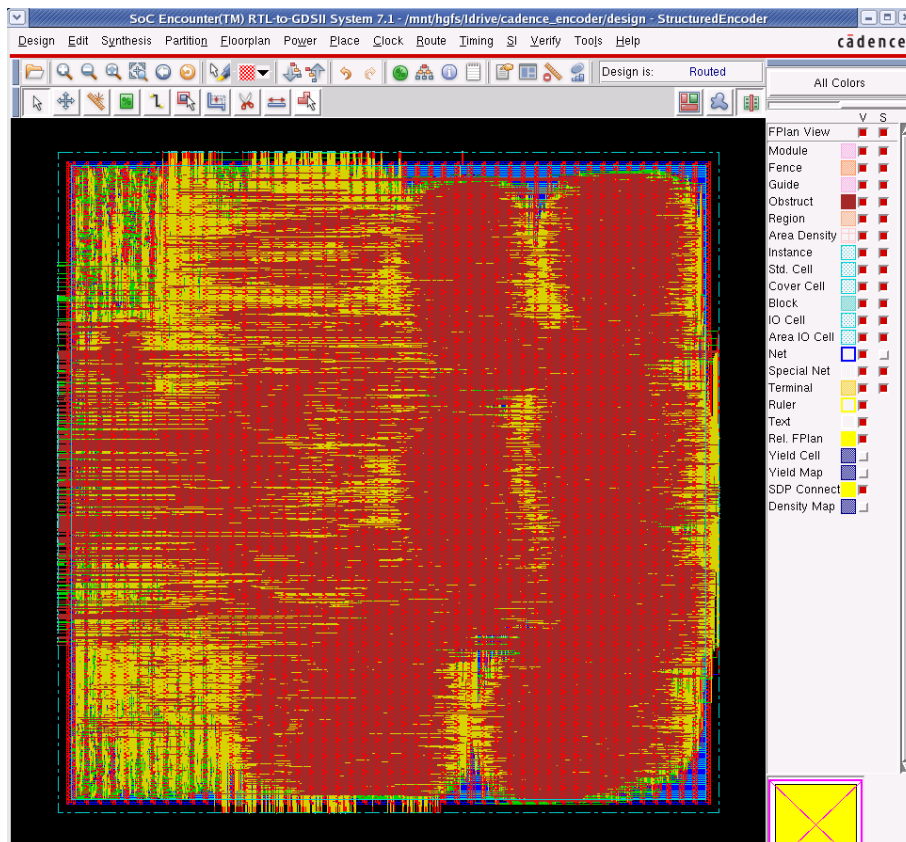


Figure 6.14: Layout view of the flexible multi-code rate and multi-code length LDPC structured encoder in Cadence Encounter.

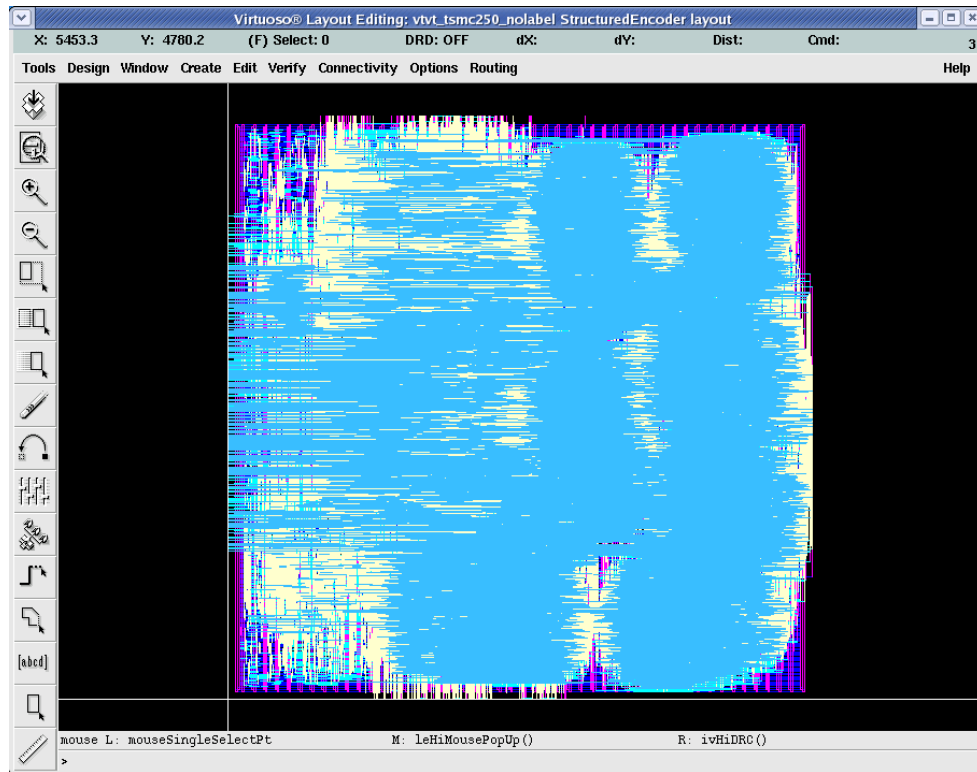


Figure 6.15: Layout view of the flexible multi-code rate and multi-code length LDPC structured encoder in Cadence ICFB.

CHAPTER 7 - Decoder for Low-Density Parity-Check Codes

LDPC encoder designs are presented in the earlier chapters. In this chapter a LDPC decoder is designed in order to encompass the complete LDPC codec. The factors affecting the LDPC decoder bit error rate (BER) performance are studied. Decoder design and its hardware implementation are presented.

The key aspects of the LDPC decoder presented in this chapter are summarized as follows:

- Decoder design methodology does not consider any structure in the LDPC codes. Hence it is applicable to both structured and any randomly generated LDPC codes.
- The decoder performance is affected by various design parameters such as the decoding algorithm, the design architecture, the quantization of log-likelihood ratios and the number of decoding iterations. All of these parameters are analyzed, and the best design parameters are chosen based on BER performance.
- Several decoding algorithms are proposed for the implementation of a LDPC decoder. From Matlab simulations, it is observed that logarithmic message passing algorithm gives the best BER performance.
- A parallel architecture yields high data rate while a serial architecture yields low data rate. In this work, the parallel architecture is chosen because of the desired high data rate.
- Different quantization of log-likelihood ratios is analyzed. It is observed that 6-bit quantization yields an acceptable BER performance reducing the implementation complexity of the design.
- The maximum number of decoding iterations affects the decoder BER performance and the decoding latency. The optimum maximum number of decoding iterations is chosen from BER simulations.
- For different SNR the number of decoding iterations required for the decoding process varies. Also for a given SNR, different codewords require different number of decoding iterations. Unlike other designs that perform fixed number of decoding iterations, the estimated codeword is verified after every iteration in this design. The decoding process is stopped when the correct codeword is estimated.

- For a given SNR and parity-check matrix, the procedure to find an optimum value of the maximum number of decoding iterations by minimizing the error, delay and energy is presented.
- The coded data rate of the decoder is dependent on the length of the codeword. Its value increases with increase in the code length. The design is applicable to both structured and any randomly generated regular and irregular LDPC codes.

7.1 Study of LDPC Decoder Parameters

The performance of the LDPC decoder depends on various factors such as decoding algorithm, architecture, quantization of log-likelihood ratios and maximum number of decoding iterations. Mackay’s parity-check matrices [56] are used to evaluate all these decoder parameters.

7.1.1 Decoding Algorithm

Sum product algorithm, minimum sum algorithm and modified minimum sum algorithm are some of the primary algorithms used for decoding LDPC codes, and they were explained in section 2.2. In this subsection, a decoding algorithm that gives better BER performance is explored. All the decoding algorithms are implemented in Matlab. The simulation results of BER performance for varying SNR are shown in Figure 7.1. It can be observed from Figure 7.1 that

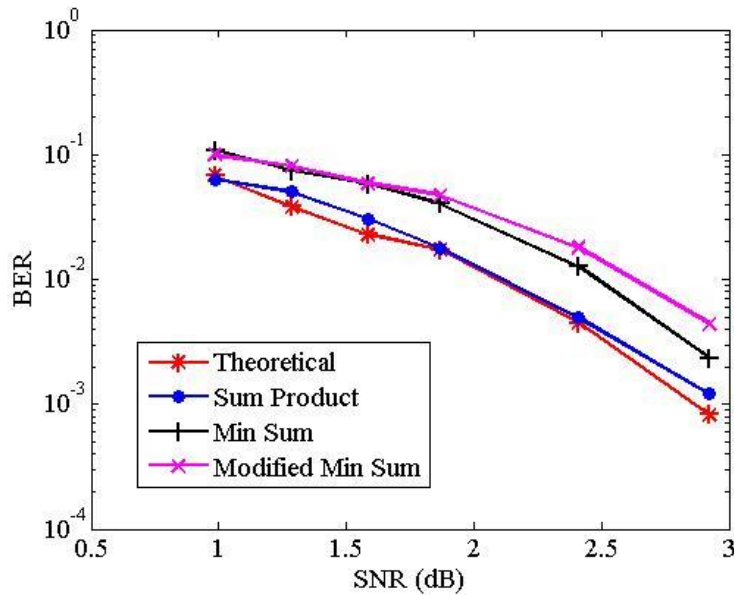


Figure 7.1: BER vs. SNR performance using different decoding algorithms.

the performance of the modified minimum sum algorithm is comparable to that of the minimum sum algorithm, and that the sum product algorithm gives the best BER performance. Hence the sum product algorithm is used for decoder implementation. However, the implementation complexity of the sum product algorithm is high when compared to min-sum algorithm as discussed on section 2.2.2.

7.1.2 Decoder Architecture

A serial architecture for the decoder implementation is efficient in terms of hardware resources but yields low data rate. Using a parallel architecture yields high data rate at the expense of large hardware resources. In this work, a parallel architecture is chosen for the decoder implementation because of the desired high data rate.

7.1.3 Quantization

Because decoders are implemented using digital logic, quantization is present on the log-likelihood ratios as they are passed between the check and variable nodes of the Tanner graph. This will also influence the BER performance of the LDPC codes, and is one of the most important factors that influences the hardware implementation of the decoder. If more bits are used to represent the log-likelihood ratios, then the performance of the decoder is increased because of the improved accuracy. However, this will also increase the number of logic elements required for the implementation of the decoder. It also slows down the decoder process and increases latency. In this subsection, the number of bits required without compromising performance and latency is evaluated.

7.1.3.1 Quantization of φ

The quantization of φ is important in determining the corresponding quantization of log-likelihood ratios. φ is a non-linear function and is defined below, but a linear approximation with a sufficient number of levels can still provide a performance close to that of the double precision case.

$$\varphi(z) = \log \frac{e^z + 1}{e^z - 1} \quad (7.1)$$

Figure 7.2 shows the quantization effect on $\varphi(z)$. From Figure 7.2, it can be observed that double precision $\varphi(z)$ is approximately zero for z equal to 3.5. Therefore, 2 bits are chosen to represent

the integer part of z . $\varphi(z)$ is computed by varying the number of bits needed to represent the fraction part of z from 1 to 4 and is shown in Table 7.1. From Figure 7.2, 5-bit quantization of $\varphi(z)$ provides performance close to double precision $\varphi(z)$. Therefore, 5-bit quantization is chosen to represent $\varphi(z)$.

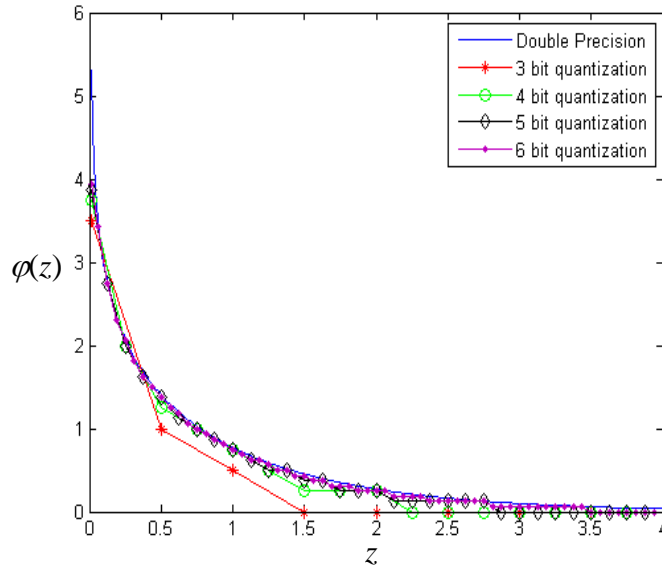


Figure 7.2: Quantization of φ .

Table 7.1: Quantization of φ .

Quantization	Integer	Fraction
3-bit	2	1
4-bit	2	2
5-bit	2	3
6-bit	2	4

7.1.3.2 Quantization of Log-Likelihood Ratios

Figure 7.3 shows the effect of quantization on BER performance for varying SNR. Simulations are performed by using 1 bit for sign, 2 bits for integer part and varying number of bits to represent the fractional part as shown in Table 7.2. The number of bits to represent the fractional part is varied from 1 to 4 in increments of 1 and the simulation results are shown in Figure 7.3. It can be observed from Figure 7.3, that the 6-bit quantization (1 bit for sign, 2 bits

for integer and 3 bits for fraction) gives a comparable performance to that of double precision. Hence 6-bit quantization is chosen to represent log-likelihood ratios.

Table 7.2: Quantization of log-likelihood ratios.

Quantization	Sign	Integer	Fraction
4-bit	1	2	1
5-bit	1	2	2
6-bit	1	2	3
7-bit	1	2	4
8-bit	1	3	4

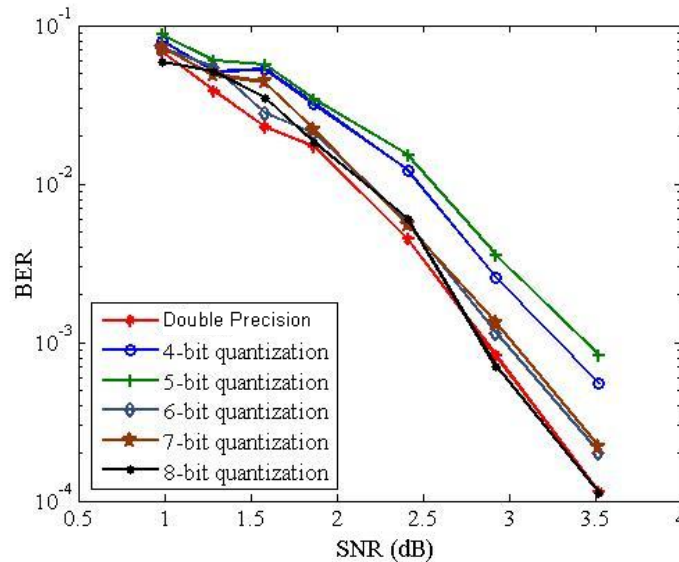


Figure 7.3: BER vs. SNR performance for different quantization levels of log-likelihood ratios.

7.1.4 Maximum Number of Decoding Iterations

The maximum number of decoding iterations determines the maximum latency of the decoder. With an increase in the number of decoding iterations the performance improves at the cost of increased latency. In this subsection the maximum number of decoding iterations required is chosen based on the trade-off between performance and latency. Figure 7.4 shows the BER performance when varying the SNR for different values of maximum number of decoding

iterations. It can be observed from Figure 7.4, that a maximum of 20 decoding iterations can be chosen for better BER performance without significant impact on the decoder performance.

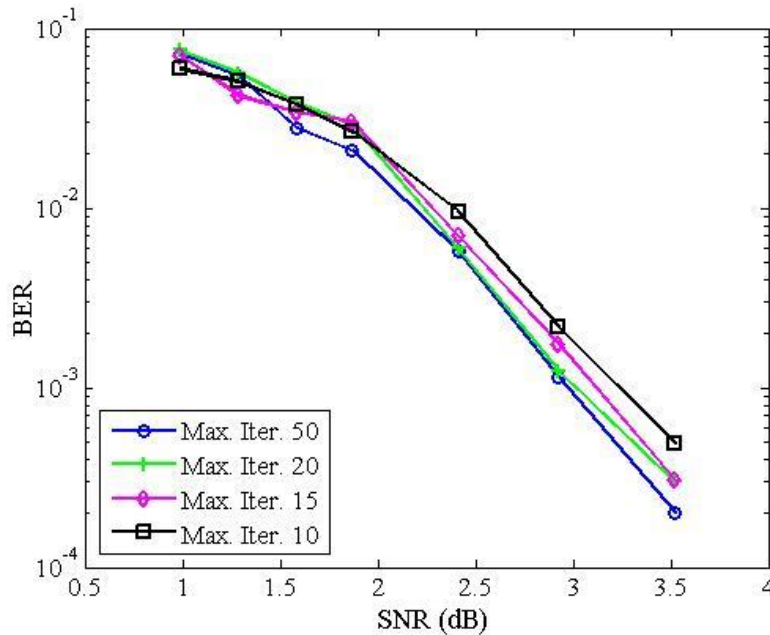


Figure 7.4: BER vs. SNR for different values of maximum number of decoding iterations.

7.2 Design and Implementation of LDPC Decoder on FPGA

Once the required parity-check matrix is chosen all the parameters discussed above can be obtained from simulations to help determine the best hardware implementation of the LDPC decoder. The logarithm message passing algorithm presented in 2.2.1 is used for the decoder implementation. In this section the details of the design and the hardware implementation of LDPC decoder are described.

7.2.1 Quantization

The number of bits required to represent the log-likelihood ratios used in the decoding process is presented in this subsection. This is the most important issue in hardware implementation of the decoder because decoding performance and complexity are dependent on the number of bits used to represent the log-likelihood ratios. The quantization of function $\varphi(z)$ and the log-likelihood ratios passed between the check and variable nodes are presented.

7.2.1.1 Quantization of φ

In subsection 7.1.3.1, it is shown that 5-bit quantization is required to represent $\varphi(z)$ without compromising much on performance and latency. From step 2 of the logarithmic message passing algorithm it can be observed that $\varphi(z)$ is computed for positive values of z . Therefore, it is sufficient to store the values of $\varphi(z)$ for only positive values of z . Implementation of the log and tanh functions in $\varphi(z)$ requires a lot of hardware and has high complexity. As an alternative $\varphi(z)$ is computed for different values of z , which are then stored in a look up table (LUT). By using 5-bit quantization, the minimum and maximum positive values that can be represented are 0 and 3.875 respectively, and z is varied from 0 to 3.875 in increments of 0.125 ($=1/2^3$). Its binary equivalent representation ranges from 0 to 31 in increments of 1. Since $\varphi(z)$ theoretically obtains its maximum value of infinity when z is equal to 0, the 5-bit quantized version of $\varphi(z)$ is limited to 3.875. Therefore $\varphi(z)$ also varies from 0 to 3.875. The actual value, binary equivalent and binary representation of z and $\varphi(z)$ when z is equal to 0, 1, 2 and 3 are shown in Table 7.3. For all the values of z ranging from 0 to 3.875, the actual value, binary equivalent and binary representation of z and $\varphi(z)$ are given in Table D.1 in Appendix D.

Table 7.3: Look up table for φ .

Actual value		Binary equivalent		Binary representation	
z	$\varphi(z)$	z	$\varphi(z)$	z	$\varphi(z)$
0	3.875	0	31	00000	11111
1.000	0.750	8	6	01000	00110
2.000	0.250	16	2	10000	00010
3.000	0.000	24	0	11000	00000

7.2.1.2 Quantization of Log-Likelihood Ratios

In subsection 7.1.3.2, it is shown that 6-bit quantization is used to represent the log-likelihood ratios without compromising much on performance and latency. The representation of log-likelihood ratios is similar to that of $\varphi(z)$ but the extra 6th bit is used to represent the sign of the message. 2's complement notation is used to represent the log-likelihood ratios. The range of the log-likelihood ratios using 6-bit quantization varies from -4 to +3.875 in increments of 0.125. Its binary equivalent is -32 to +31. The actual value, binary equivalent and 2's complement

representation of certain log-likelihood ratios is shown in Table 7.4. For all the values of log-likelihood ratios ranging from -4 to +3.875 its actual value, binary equivalent and 2's complement representation is given in Table D.2 in Appendix D.

Table 7.4: Quantization of log-likelihood ratios.

Actual value	Binary equivalent	2's Complement representation
0	0	000000
1.125	9	001001
2.375	19	010011
3.625	29	011101
-3.125	-25	100111
-1.875	-15	110001
-0.625	-5	111011

7.2.1.3 Conversion of Log-Likelihood Ratios from One Form to Another Form of Representation

During the decoding process, log-likelihood ratios passed between variable and check nodes are in 2's complement representation, while $\varphi(z)$ is in sign magnitude representation. Hence there is a need to convert from one form of representation to another form. In this subsection, the conversion of 2's complement to sign magnitude representation and vice-versa are presented.

During check node processing, the received variable node values in 2's complement representation need to be converted to sign magnitude representation. This conversion is performed as shown in Figure 7.5. If the MSB of the input is equal to 0 then the output, *Out*, is equal to the input, *In*. Otherwise, the input bits, *In*, are inverted and 1 is added to convert the input, *In*, into sign magnitude representation. The sign bit, s_g , is the MSB of the input, *In*.

The check node values are in sign magnitude representation. These values need to be converted back to 2's complement representation for the computation of variable node values. This conversion is performed as shown in Figure 7.6. If the input s_g is equal to 0 then the output, *Out*, is equal to the concatenation of bit s_g and the 5 input bits, *In*. Otherwise, the output, *Out*, is equal to the concatenation of s_g bit and the 2's complement of the input bits, *In*.

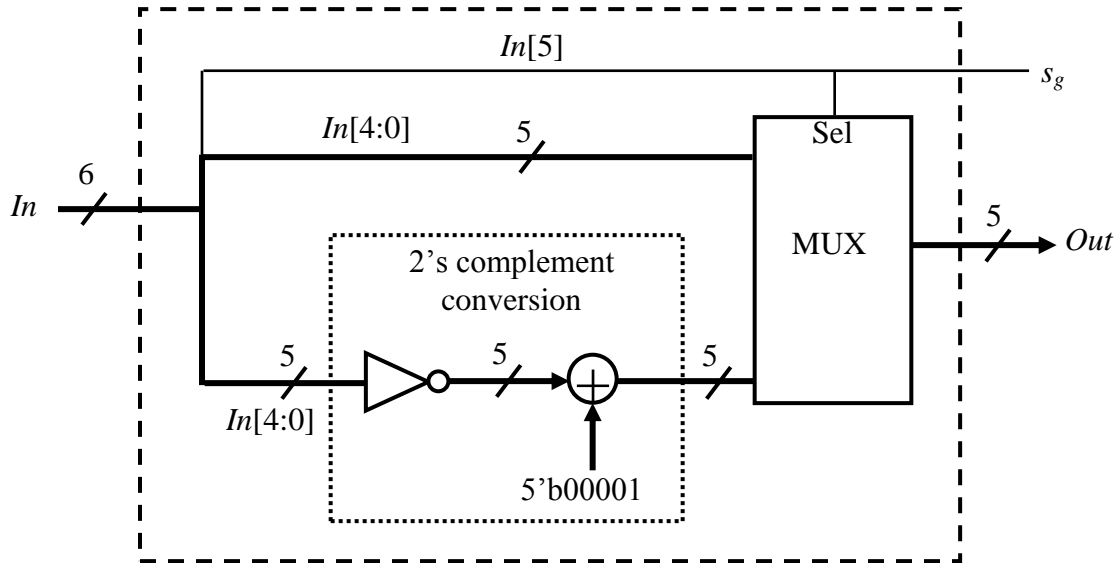


Figure 7.5: Conversion of 2's complement to sign magnitude representation.

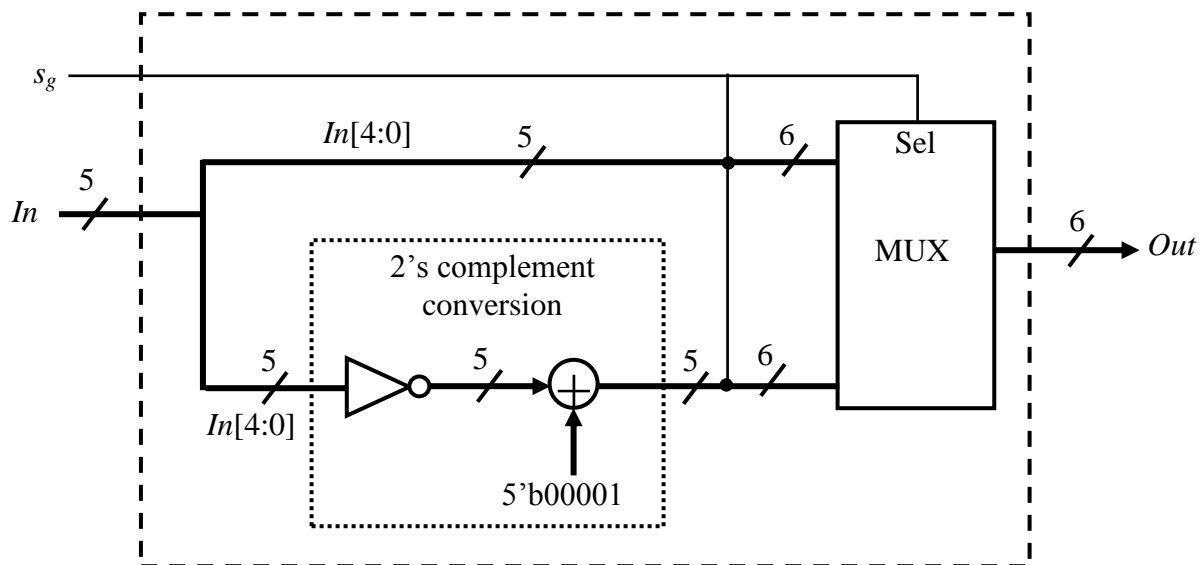


Figure 7.6: Conversion of sign magnitude to 2's complement representation.

7.2.2 Initialization of Decoder Process

In the decoder implementation it is assumed that the initial log-likelihood values, $L(c_i)$, are available to the decoder which is equal to $\frac{2y_i}{\sigma^2}$, where y is the received code word and σ^2 is the channel noise variance. These values are computed and quantized as shown in Table 7.4. This step needs to be performed only once for a given codeword and is performed off-chip. If the value of $L(c_i)$ is greater than +3.875 (its binary equivalent is 31) then its value is assigned to

+3.875. Similarly, if the value of $L(c_i)$ is less than -4 (its binary equivalent is -32) then its value is assigned to -4. As shown in step 1 of the logarithmic message passing algorithm presented in 2.2.1, the variable nodes $L(q_{ij})$ are initialized and is equal to $L(c_i)$.

7.2.3 Check Node Processing

Step 2 of the logarithmic message passing algorithm presented in 2.2.1 is to compute the check nodes values. The check node values, $L(r_{ji})$, are computed from the variable node values, $L(q_{ij})$ as shown below

$$L(r_{ji}) = \prod_{i \in R_{j \setminus i}} \alpha_{ij} \cdot \varphi \left(\sum_{i \in R_{j \setminus i}} \varphi(\beta_{ij}) \right) \quad (7.2)$$

where $\alpha_{ij} = \text{sign}(L(q_{ij}))$, $\beta_{ij} = |L(q_{ij})|$ and $\varphi(z) = \log \frac{e^z + 1}{e^z - 1}$.

The magnitude of the check nodes is obtained by computing

$$|L(r_{ji})| = \varphi \left(\sum_{i \in R_{j \setminus i}} \varphi(\beta_{ij}) \right). \quad (7.3)$$

The implementation of Equation 7.3 is shown in Figure 7.7. In this implementation φ is obtained from the look up table in Table 7.3. The sign of the check node values can be found from as shown below, and it is implemented in hardware as shown in Figure 7.8.

$$\text{sign}\{L(r_{ji})\} = \prod_{i \in R_{j \setminus i}} \alpha_{ij} \quad (7.4)$$

Figure 7.7 shows the computation of the magnitude of check node values associated with parity-check matrix of row weight w_r . For regular LDPC codes the row weight w_r is equal to a constant value. Therefore, one implementation for a given w_r would be sufficient for the entire decoder. For irregular LDPC codes, w_r can be different for each row and therefore several different implementations for each w_r , are needed for its decoder. Based on the type of LDPC codes, one or more of these check node processing designs are implemented.

The magnitude and sign of a check node is obtained using the magnitudes and sign bits of the other $w_r - 1$ check nodes, respectively, as shown in Figures 7.7 and 7.8. For a check node

shown in Figure 7.7, the φ outputs of all the w_r inputs are added excluding the φ output of its input.

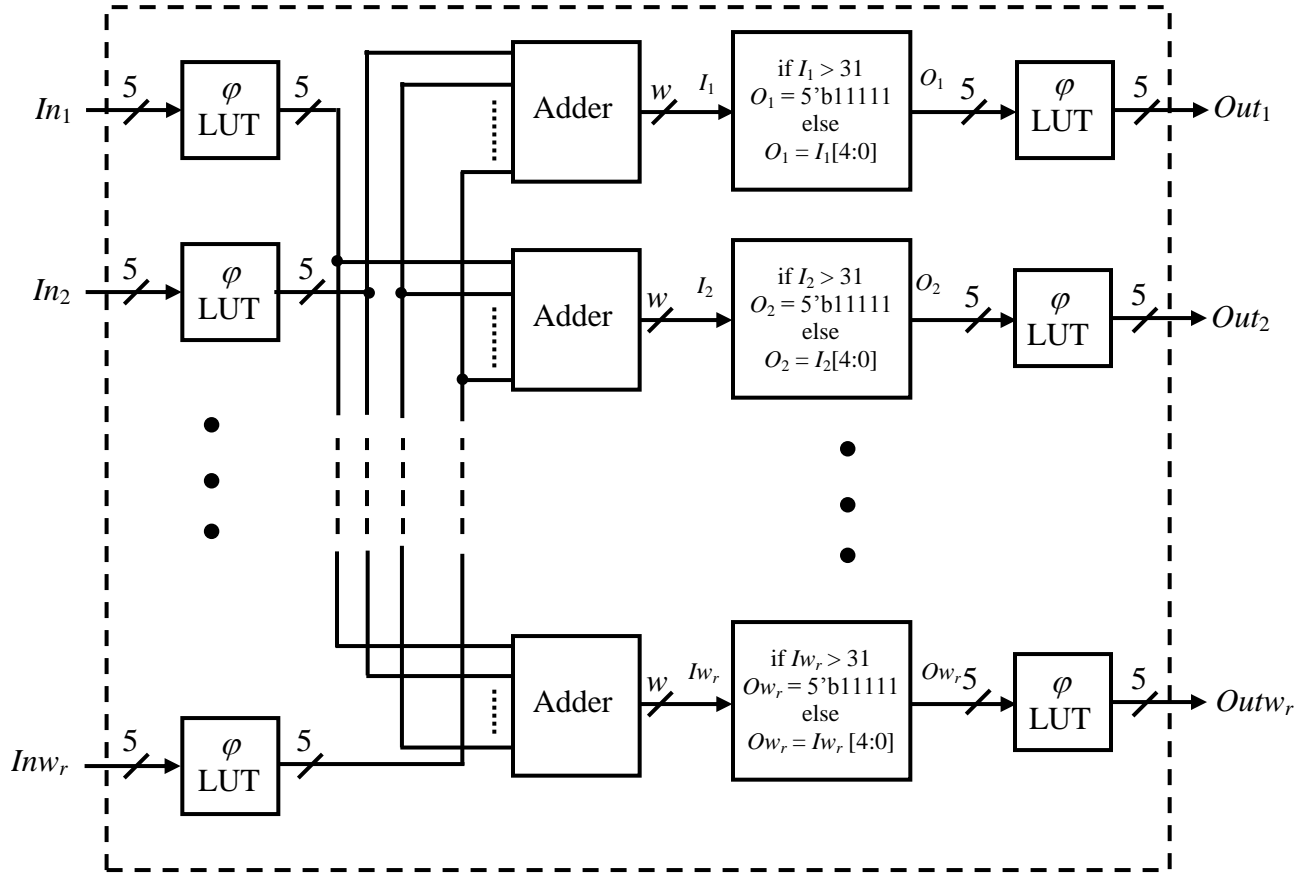


Figure 7.7: Computation of magnitude of check node values.

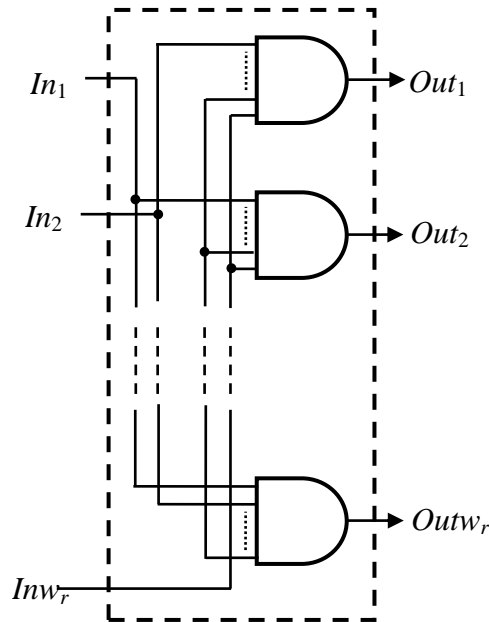


Figure 7.8: Computation of sign of check node values.

For example, the check node, In_1 , φ of $In_2, In_3, \dots In_{w_r}$ are added to obtain I_1 . This resulting sum is equal to the w -bit long, when w_r-1 outputs of φ , each of length 5-bit, are added. w is equal to $\lceil \log_2(w_r-1) \times 31 \rceil$ where $\lceil \rceil$ denotes the ceiling function. This w -bit I_1 is truncated to 5 bits by assigning a value of 31 when its value is greater than 31. The magnitude of the check node, In_1 , is updated by obtaining φ of I_1 and is assigned to output, Out_1 . The same procedure is followed to update the magnitudes of the other check nodes as shown in Figure 7.7. From Figure 7.8, the sign of the check node is obtained by ANDing all the sign bits of the w_r check nodes excluding its sign bit. For example, the sign of In_1 is obtained by ANDing $In_2, In_3, \dots In_{w_r}$ and is assigned to Out_1 . Similarly other check nodes sign bits are updated as shown in Figure 7.8. The check node values are computed as shown in Figure 7.9 by combining the magnitude and sign bits computed using Equations 7.3 and 7.4.

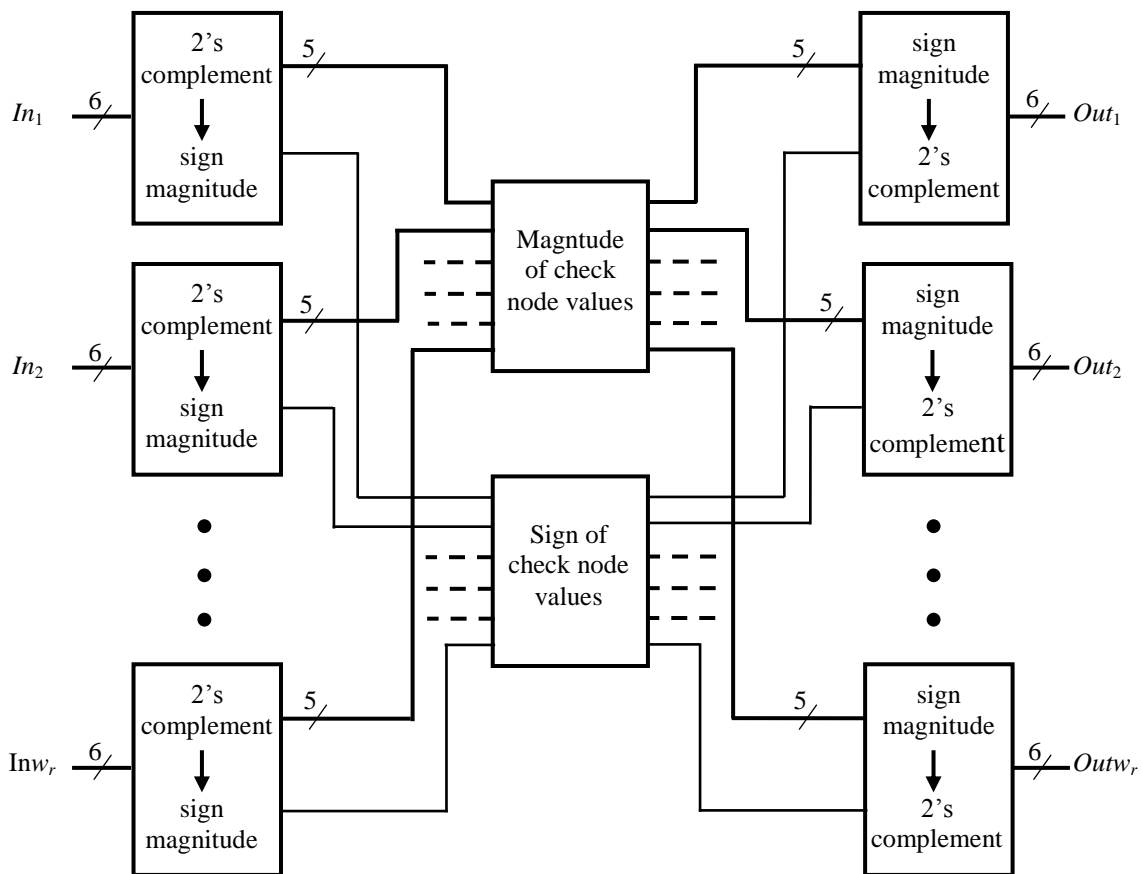


Figure 7.9: Computation of check node values.

The received variable node values are first converted into sign magnitude representation from 2's complement representation. The magnitude and sign of the check node values are

computed from Figures 7.7 and 7.8 and is converted back to 2's complement representation. These values are used for the computation of variable nodes.

7.2.4 Variable Node Processing

The variable node values are obtained from the check node values using step 3 of the decoding algorithm in 2.2.1:

$$L(q_{ij}) = L(c_i) + \sum_{j \in c_i \setminus j} L(r_{ji}) \quad (7.5)$$

This is also shown in Figure 7.10 for LDPC codes with column weight w_c .

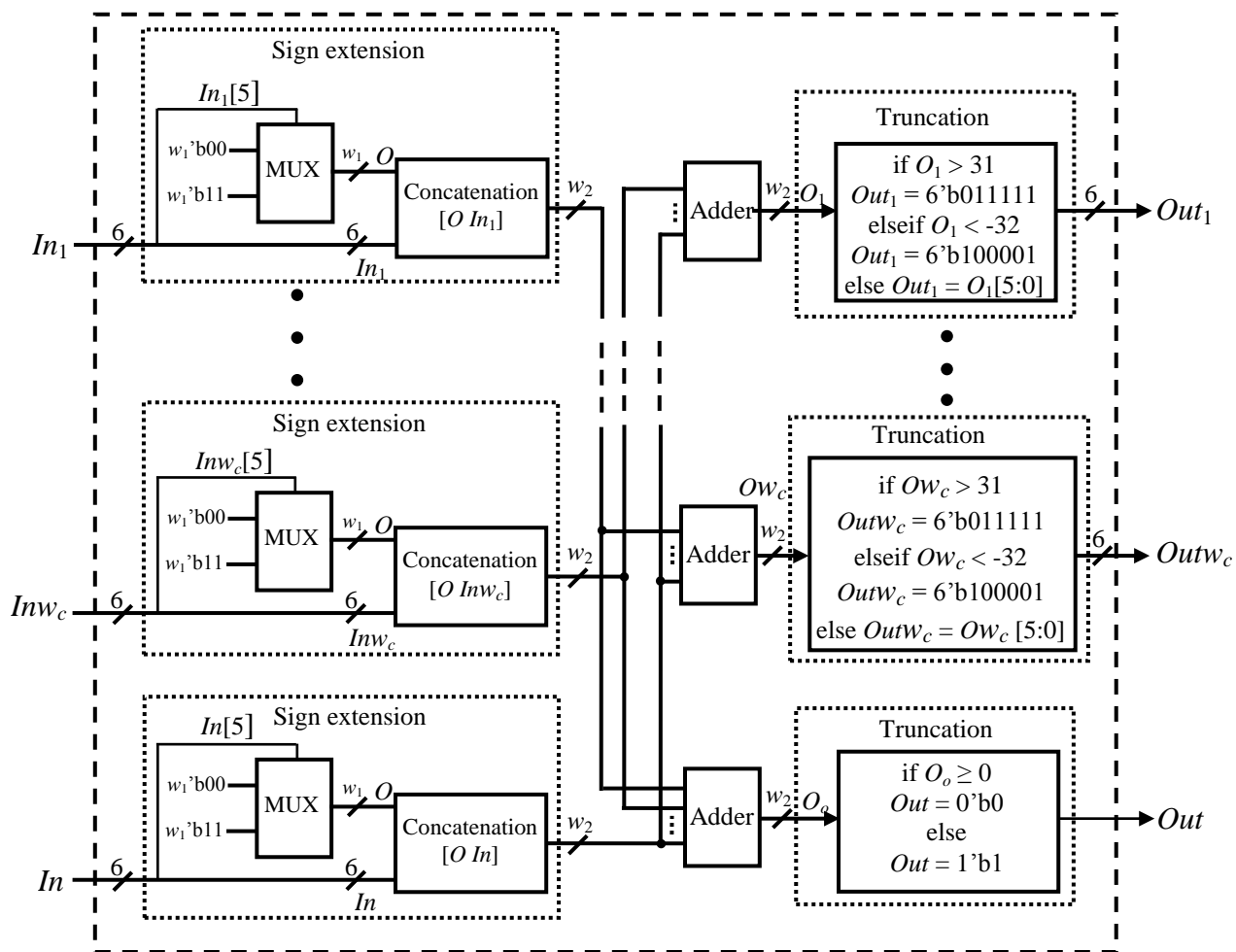


Figure 7.10: Computation of variable node values.

For regular LDPC codes the column weight w_c is equal to a constant value whereas for an irregular LDPC code column weights can be different for different columns. Depending on w_c ,

the design in Figure 7.10 is modified accordingly. Based on the type of LDPC codes, one or more of these variable node processing designs are implemented.

Each variable node is updated by adding all the $w_c + 1$ inputs excluding the input of the node itself. When w_c inputs, each of length 6-bits, are added this results in a sum equal to w_2 -bits long. The value w_2 is equal to $\lceil \log_2(w_c-1) \times 63 \rceil$ where $\lceil \cdot \rceil$ denotes the ceiling function. As shown in Figure 7.10, a sign extension is performed on all the $w_c + 1$ inputs. The number of bits appended to the variable nodes is equal to w_1 where $w_1 = w_2 - 6$. The updated variable node is now of length w_2 -bits which also needs to be truncated to 6 bits. Truncation is performed by assigning the variable node a value of 31 and -32 when its value is greater than 31 or less than -32, respectively.

Step 4 of the decoding algorithm is computation of $L(Q_i)$ defined as

$$L(Q_i) = L(c_i) + \sum_{j \in c_i} L(r_{ji}). \quad (7.6)$$

Step 5 is making the decision on the received codeword based on the value of $L(Q_i)$ and is defined as

$$\hat{c}_i = \begin{cases} 1 & \text{if } L(Q_i) < 0 \\ 0 & \text{if } L(Q_i) \geq 0 \end{cases}. \quad (7.7)$$

For all the values of i if the value of $L(Q_i)$ is greater than or equal to zero then the received bit of the codeword is declared to be 0 or else 1.

Steps 4 and 5 of the decoding algorithm are also included in the computation of the variable node values as shown in Figure 7.10. $L(Q)$ is obtained using Equation 7.6 and its implementation is similar to that of the variable node computation except $L(Q)$ is obtained by adding all the $w_c + 1$ inputs. The codeword is estimated using Equation 7.7 and O_o and Out in Figure 7.10 represents the $L(Q)$ and \hat{c}_i respectively. From Equation 7.7, the estimated codeword bit, Out , is assigned a value of 0 if O_o is greater than or equal to zero otherwise Out is assigned a value of 1.

7.2.5 End of Decoding Process

After each decoding iteration a decision is made on the codeword. The estimated codeword, \hat{c} , is then verified by multiplying it with the parity-check matrix. If the resultant vector is zero, i.e., $\hat{c}H^T = 0$, then the received codeword is decoded correctly or else the

decoding process is continued. This process is continued until the received codeword is decoded correctly or it has reached the fixed maximum number of decoding iterations.

The product of $\hat{c}H^T$ is of size $1 \times n$ and is hard coded in the design. An element of the product of $\hat{c}H^T$ is obtained by multiplying \hat{c} with a column of H^T . This can be implemented by XORing the bits of \hat{c} positioned at the corresponding locations of 1's in each column of H .

7.2.6 Decoder

Figure 7.11 shows the decoder implementation. The decoder implementation consists of four blocks which are initialization, computation of check and variable nodes and validation of the estimated codeword.

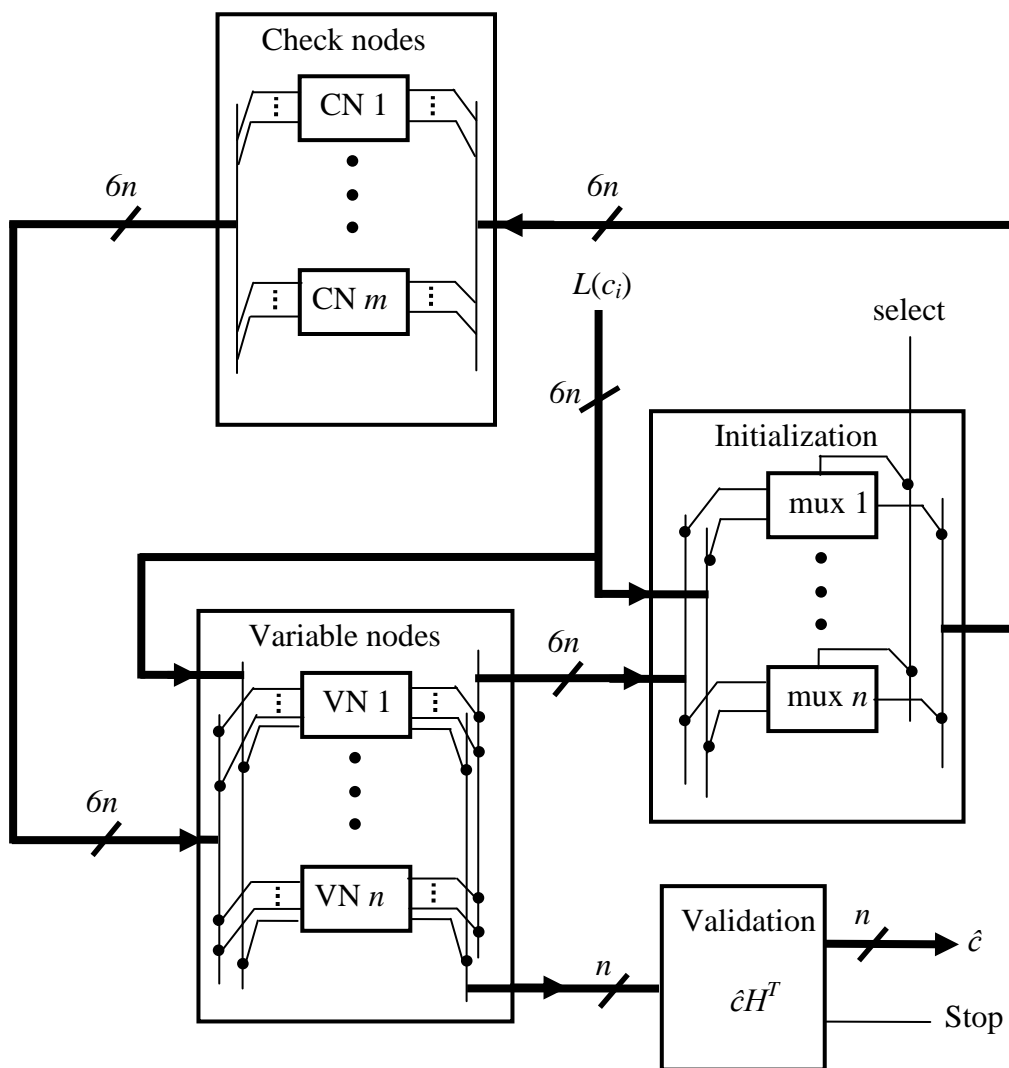


Figure 7.11: Design of LDPC decoder.

Variable nodes are initialized from the received codeword. Using these variable node values, check node values are computed in the first half of the decoding iteration. The design for the computation of check nodes shown in Figure 7.9 is replicated m times to update the entire check node values of the decoder.

The variable node values are updated in the other half of the decoding iteration from the check node values. The design for the computation of variable nodes shown in Figure 7.10 is replicated n times to update the entire variable node values of the decoder. After completion of a decoding iteration, an estimate is made on the received codeword and is checked for validity. If the codeword is decoded correctly then the decoding process is stopped. Otherwise, it is continued till it reaches the maximum number of decoding iterations.

7.3 Results

A hardware implementation was performed on an Altera Stratix EP1S80F1508C5 FPGA using Quartus II. Verilog modules generated again from Matlab script are used for the implementation. The design requires a large number of LEs because of the use of parallel decoder architecture. Because of the restrictions on LEs a decoder with a small code length can only be implemented on the available FPGAs. Decoders for code lengths 64 and 128 are implemented on FPGA for a regular and irregular parity-check matrix of sizes 64×128 and 32×64 respectively. The results are shown in Table 7.5. With an increase in the code length the number of logic elements required by the decoder also increases. Let the decoding clock frequency be Clk_d , which is equal to the maximum clock frequency of the synthesized designs shown in Table 7.5. From section 7.1.4, the maximum number of decoding iterations ($Iter_{Max}$) is chosen to be 20. The latency of the decoder can be computed by $Iter_{Max}/Clk_d$ and is shown in Table 7.5. The coded data rate of the decoder can be computed from Equation 7.8 and is shown in Table 7.5. The coded data rate increases with increase in the code length.

$$coded\ data\ rate = n \times \frac{Clk_d}{Iter_{Max}} \quad (7.8)$$

Table 7.5: Synthesis results of the LDPC decoder.

H	LE	Clk_d (MHz)	Coded data rate (Mbps)	Latency (μs)
Irreg 32×64	24384	36.48	116.74	0.55
Reg 64×128	49985	34.47	220.61	0.58

These calculations are based on the internal decoder design and not on any special I/O limitations. The decoder implementation assumes that all input data bits are available for decoding, so serialization factors are not included in the results. In order to consider the decoder implementation under serialization, a shift register needs to be added. The complete decoder with I/O serialization is shown in Figure 7.12. The coded data rate thus becomes limited by the speed at which the shift register can run, Clk_s , which is 422.12 MHz. The latency in reading codeword is n/Clk_s . The latency of the complete decoder system is equal to the maximum value of $[n/Clk_s, Iter_{Max}/Clk_d]$. For small code lengths the latency is equal to $Iter_{Max}/Clk_d$. Therefore the coded data rate of the complete decoder is same as that of the decoder without I/O serialization and is equal to $n \times Iter_{Max}/Clk_d$.

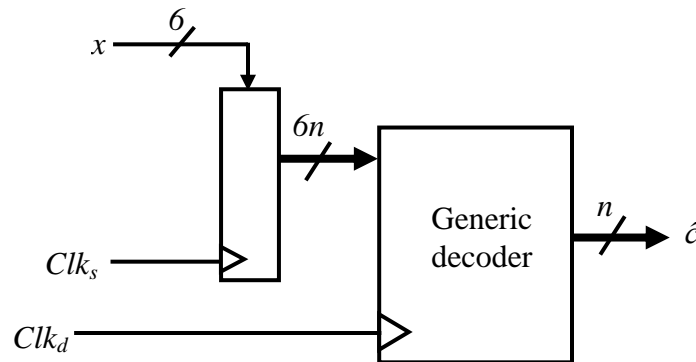


Figure 7.12: Complete decoder system.

The LDPC decoder implementation on FPGA is restricted to small code lengths because of the huge hardware requirement. The decoder coded data rate is directly proportional to the code length. Therefore, implementing a decoder in an ASIC would accommodate decoders with large code lengths and hence increases the coded data rate. An LDPC decoder with code length of 1024 and code rate 1/2 is synthesized in Cadence RTL Compiler and the synthesis results are presented in Table 7.6. The coded data rate of the decoder without I/O serialization is 3.17 Gbps which is much higher than the decoders presented in [42] and [43].

Table 7.6: Synthesis results of LDPC decoder of code length 1024 and code rate 1/2 in Cadence RTL Compiler.

Parameter	Our proposed decoder	Decoder in [42]	Decoder in [43]
Code length	1024	648	1024
Code rate	1/2 Irregular	5/6 Irregular	1/2 Regular
Technology	0.25 μm	0.18 μm	0.16 μm
Gate count	820.3 K	842 K	1750 K
Clock frequency	61.89 MHz	111 MHz	64 MHz
Data rate	3.17 Gbps	1 Gbps	1 Gbps
Maximum decoding iterations	20	10	64

7.4 Optimization of Decoder Parameters

In this section, an attempt is made to find an optimum number of maximum decoding iterations for a given SNR based on erroneous codewords (*error*), energy required (*energy*) and latency of the decoding process (*delay*).

7.4.1 Erroneous Codewords

In the earlier section 7.1.4, it was discussed that the maximum number of decoding iterations ($Iter_{Max}$) plays an important role in the error performance of the decoder. The maximum number of decoding iterations varies with the parity-check matrix and SNR. An example parity-check matrix of size 64×128 is considered to show the affect of SNR on $Iter_{Max}$. Decoder simulations are performed in Matlab, and 1000 codewords are decoded for a given SNR. Figure 7.13 shows the histograms of the decoding iterations required by the codewords for varying SNR using a maximum of 50 decoding iterations. From Figure 7.13, it can be observed that for low SNR (0.9844 dB), a large number of codewords (64 %) require 50 iterations whereas for high SNR (2.9226 dB) only a few codewords (4 %) require 50 iterations. For all values of SNR, the codewords that require 50 decoding iterations may still not be corrected. Number of

erroneous codewords, *errors*, for varying $Iter_{Max}$ from 5 - 50 in increments of 5 are shown in Table 7.7.

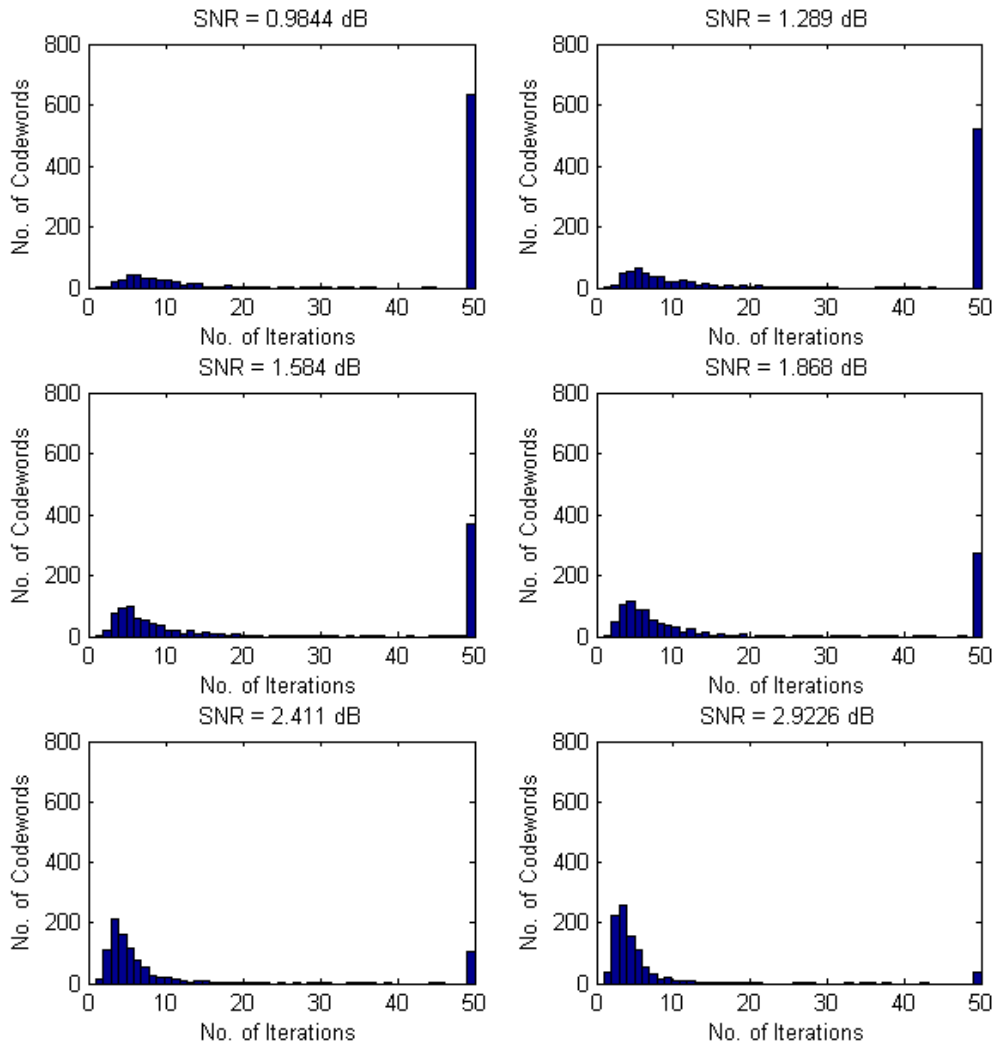


Figure 7.13: Histogram of decoding iterations required by codewords for varying SNR.

From Table 7.7, it can be observed that the number of erroneous codewords decreases with the increase in the maximum number of decoding iterations. This decrease in the erroneous codewords is initially large and then flattens out with an increase in the maximum number of decoding iterations. For example, at SNR equal to 0.9844 dB, when the maximum number of decoding iterations is increased from 5 to about 20, the decrease in the erroneous codewords is large. However, the decrease in the erroneous codewords is small when the number of decoding iterations is increased from 20 to 50.

Table 7.7: Erroneous codewords, errors, for varying SNR and $Iter_{Max}$.

$Iter_{Max}$ \ SNR (dB)	5	10	15	20	25	30	35	40	45	50
0.9844	946	780	703	678	664	654	647	645	644	643
1.289	890	684	604	574	554	541	538	533	529	529
1.584	806	513	438	403	396	388	381	376	375	369
1.868	732	425	337	309	300	293	289	284	280	278
2.411	505	204	156	133	126	119	116	112	112	109
2.9226	325	89	62	51	48	45	44	40	40	40

7.4.2 Decoder Delay

The latency involved in the decoding process for I number of iterations is equal to $I \times t$, where t is the time required for one iteration. From the decoder design, the time required for one decoding iteration is equal to one clock period and from Quartus compilation report, t is equal to 29 ns. Latency is independent of SNR. The latency involved in the decoding process for varying $Iter_{Max}$ is shown in Table 7.8.

Table 7.8: Decoder latency, delay, for varying $Iter_{Max}$.

$Iter_{Max}$	Latency (ns)
5	145
10	290
15	435
20	580
25	725
30	870
35	1015
40	1160
45	1305
50	1450

7.4.3 Decoder Energy

Power analysis is performed using the PowerPlay power analyzer described in section 3.1.3 on the decoder implemented. For analysis, two codewords are considered at SNR of 0.9844 dB, where one codeword is not decoded correctly even after 50 iterations and the other codeword is decoded correctly in 7 iterations. Power analysis is performed for a maximum of 20 iterations

and the total thermal power dissipation, TTPD, core dynamic thermal power dissipation, CDTPD, core static thermal power dissipation, CSTPD, and I/O thermal power dissipation, IOTPD, are obtained from power analysis compilation report and is shown in the Table 7.9. CSTPD is constant and is equal to 1395 mW.

Table 7.9: PowerPlay power analysis report of the decoder of size 64 × 128.

Iteration No.	Time (ns)	Codeword corrected in 7 Iterations (SNR 0.9844 dB)			Codeword not corrected in 51 Iterations (SNR 0.9844 dB)		
		TTPD (mW)	CDTPD (mW)	IOTPD (mW)	TTPD (mW)	CDTPD (mW)	IOTPD (mW)
1	0-30	3699	2134	170	3459	1168	896
2	30-60	7717	5870	451	7149	5323	432
3	60-90	5690	4140	155	5226	3643	188
4	90-120	5214	3651	168	5142	3546	200
5	120-150	5212	3694	123	4802	3284	123
6	150-180	5256	3693	168	4649	3131	123
7	180-210	5445	3927	123	4441	2929	117
8	210-240	4966	3442	130	4356	2787	175
9	240-270	4517	3031	91	4328	2803	130
10	270-300	3221	1728	97	4435	2891	149
11	300-330	1662	146	91	4479	2922	162
12	330-360	1499	0.6	104	4598	3060	142
13	360-390	1486	0.1	91	4445	2901	149
14	390-420	1493	0.2	97	6984	5466	123
15	420-450	1486	0.3	91	4772	3165	213
16	450-480	1512	0.7	117	4772	3165	213
17	480-510	1486	0.1	91	4772	3165	213
18	510-540	1493	0.2	97	4772	3165	213
19	540-570	1486	0.18	91	4772	3165	213
20	570-600	1499	0.45	104	4772	3165	213

From Table 7.9, it can be observed that TTPD is maximum after initialization because of the high signal activity. The TTPD decreases with increase in the number of decoding iterations. In the case when the codeword is corrected in 7 iterations, the CDTPD is negligibly small after 11 decoding iterations and the TTPD is almost equal to or little higher than CSTPD. The TTPD decreases with increase in the number of decoding iterations for the codeword that did not decode correctly after 51 iterations. The TTPD value reaches a constant value equal to 4772 mW after 14 iterations. This is because the variable and check node message values become stagnant after 14 iterations and the signals stop toggling.

From Table 7.9, the decoder average TTPD per iteration while decoding ($P_{OW_{Run}}$) and idle ($P_{OW_{Idle}}$) are computed and are equal to 5462 mW and 1510 mW respectively. The $P_{OW_{Run}}$ and $P_{OW_{Idle}}$ are obtained from Table 7.9 using TTPD of the codeword that decodes correctly in 7 iterations. The $P_{OW_{Run}}$ is obtained by averaging the TTPD during 7 decoding iterations (time period of 0-210 ns). The $P_{OW_{Idle}}$ is obtained by averaging the TTPD when the decoder is idle, i.e., after the decoder estimated the correct codeword. From Table 7.9, $P_{OW_{Idle}}$ is equal to average of the average of TTPD from 300-600 ns. The power dissipated during the time interval of 210-300 ns is not considered in the calculations of $P_{OW_{Run}}$ and $P_{OW_{Idle}}$ because the decoder has estimated correct codeword and has not reached an idle state yet.

The energy required by the decoder for a given SNR can be computed from $Iter_{Max}$, the average number of decoding iterations, the time per iteration and the power dissipation rate during the decoding process. The energy required is computed from Eq. 7.9 and is shown in Table 7.10.

Table 7.10: Energy (pJ) required for varying SNR and $Iter_{Max}$.

$Iter_{Max}$ \ SNR (dB)	5	10	15	20	25	30	35	40	45	50
0.9844	795	1591	2386	3181	3977	4772	5567	6020	6244	6468
1.289	795	1591	2386	3181	3977	4772	5224	5448	5672	5896
1.584	795	1591	2386	3181	3977	4201	4424	4648	4872	5096
1.868	795	1591	2386	3181	3405	3630	3853	4077	4301	4525
2.411	795	1591	2043	2267	2491	2715	2939	3163	3387	3611
2.9226	795	1248	1472	1696	1920	2144	2368	2592	2815	3040

The average number of decoding iterations, $Iter_{Avg}$, required for SNR values of 0.9844, 1.289, 1.584, 1.3868, 2.411 and 2.9226 dB obtained from Matlab simulations are 37, 32, 25, 20, 12 and 7 respectively.

$$energy = \begin{cases} Iter_{Max} \times Pow_{run} \times t, & \text{if } Iter_{Max} < Iter_{Avg} \\ ((Iter_{Avg} \times Pow_{run}) + ((Iter_{Max} - Iter_{Avg}) \times Pow_{Idle})) \times t, & \text{if } Iter_{Max} > Iter_{Avg} \end{cases} \quad (7.9)$$

7.4.4 Optimization

The following two cases are considered to find an optimum value of $Iter_{Max}$ for a given SNR by attempting to minimize *error*, *delay* and *energy*.

Case I:

The optimum value of $Iter_{Max}$ for a given SNR is obtained by minimizing the *error*, *energy* and *delay*. A function for a given SNR and $Iter_{Max}$ can be expressed in terms of *error*, *energy* and *delay* and is shown as

$$f(SNR, Iter_{Max}) = \alpha \times error + \beta \times energy + \gamma \times delay \quad (7.10)$$

where α , β and γ are weighing coefficients of *error*, *energy* and *delay* respectively. For a given SNR, all the values of *error*, *delay* and *energy* shown in Tables 7.7, 7.8 and 7.10 are normalized by their respective maximum values. Numerically $f(SNR, Iter_{Max})$ is evaluated by varying the values of α , β and γ from 0 to 1 in increments of 0.1 such that $\alpha + \beta + \gamma = 1$. For example, $\alpha = 0.1$, $\beta = 0.2$ and $\gamma = 0.7$. The values of the weights α , β and γ are shown in Table 7.11. α is incremented from 0 to 1 in steps of 0.1. For each value of α , γ is decremented from $1 - \alpha$ to 0 in steps of 0.1. β is chosen such that the value is equal to $1 - \alpha - \gamma$.

The plots of f for varying SNR, $Iter_{Max}$ and weights are shown in Figure 7.14. For each SNR and $Iter_{Max}$, the corresponding weights of the minimum value of f are shown in Table 7.12. From Figure 7.14 and Table 7.12, it can be observed that when *error* is not considered i.e., $\alpha = 0$ then f is minimum and its value increases with increase in the $Iter_{Max}$. Fewer decoding iterations would be optimum when latency is given priority. It can also be observed from Figure 7.14 and Table 7.12, that when $\alpha = 1$ then f decreases with increase in the $Iter_{Max}$ and it obtains minimum value for largest value of $Iter_{Max}$. This means that when error is minimized the decoder requires larger value of $Iter_{Max}$.

Table 7.11: Weighing coefficients of error (α), energy, (β), and delay (γ).

Weights	α	β	γ
1	0	0	1
2	0	0.1	0.9
3	0	0.2	0.8
4	0	0.3	0.7
5	0	0.4	0.6
6	0	0.5	0.5
7	0	0.6	0.4
8	0	0.7	0.3
9	0	0.8	0.2
10	0	0.9	0.1
11	0	1	0
12	0.1	0	0.9
13	0.1	0.1	0.8
14	0.1	0.2	0.7
15	0.1	0.3	0.6
16	0.1	0.4	0.5
17	0.1	0.5	0.4
18	0.1	0.6	0.3
19	0.1	0.7	0.2
20	0.1	0.8	0.1
21	0.1	0.9	0
22	0.2	0	0.8
:	:	:	:
:	:	:	:
54	0.8	0.2	0
55	0.9	0	0.1
56	0.9	0.1	0
57	1	0	0

Table 7.12: Minimum f corresponding to $Iter_{Max}$, SNR and weights.

SNR (dB)		$Iter_{Max}$									
		5	10	15	20	25	30	35	40	45	50
0.9844	f_{Min}	0.1	0.2	0.3	0.4	0.5	0.6	0.684	0.682	0.681	0.68
	α	0	0	0	0	0	0	1	1	1	1
	B	0	0	0	0	0	0	0	0	0	0
	γ	1	1	1	1	1	1	0	0	0	0
1.289	f_{Min}	0.1	0.2	0.3	0.4	0.5	0.6	0.605	0.599	0.594	0.594
	α	0	0	0	0	0	0	1	1	1	1
	β	0	0	0	0	0	0	0	0	0	0
	γ	1	1	1	1	1	1	0	0	0	0
1.584	f_{Min}	0.1	0.2	0.3	0.4	0.491	0.481	0.473	0.467	0.465	0.458
	α	0	0	0	0	1	1	1	1	1	1
	β	0	0	0	0	0	0	0	0	0	0
	γ	1	1	1	1	0	0	0	0	0	0
1.868	f_{Min}	0.1	0.2	0.3	0.4	0.41	0.4	0.395	0.388	0.383	0.38
	α	0	0	0	0	1	1	1	1	1	1
	β	0	0	0	0	0	0	0	0	0	0
	γ	1	1	1	1	0	0	0	0	0	0
2.411	f_{Min}	0.1	0.2	0.3	0.263	0.25	0.237	0.23	0.222	0.222	0.216
	α	0	0	0	1	1	1	1	1	1	1
	β	0	0	0	0	0	0	0	0	0	0
	γ	1	1	1	0	0	0	0	0	0	0
2.9226	f_{Min}	0.1	0.2	0.191	0.157	0.148	0.139	0.135	0.123	0.123	0.123
	α	0	0	1	1	1	1	1	1	1	1
	β	0	0	0	0	0	0	0	0	0	0
	γ	1	1	0	0	0	0	0	0	0	0

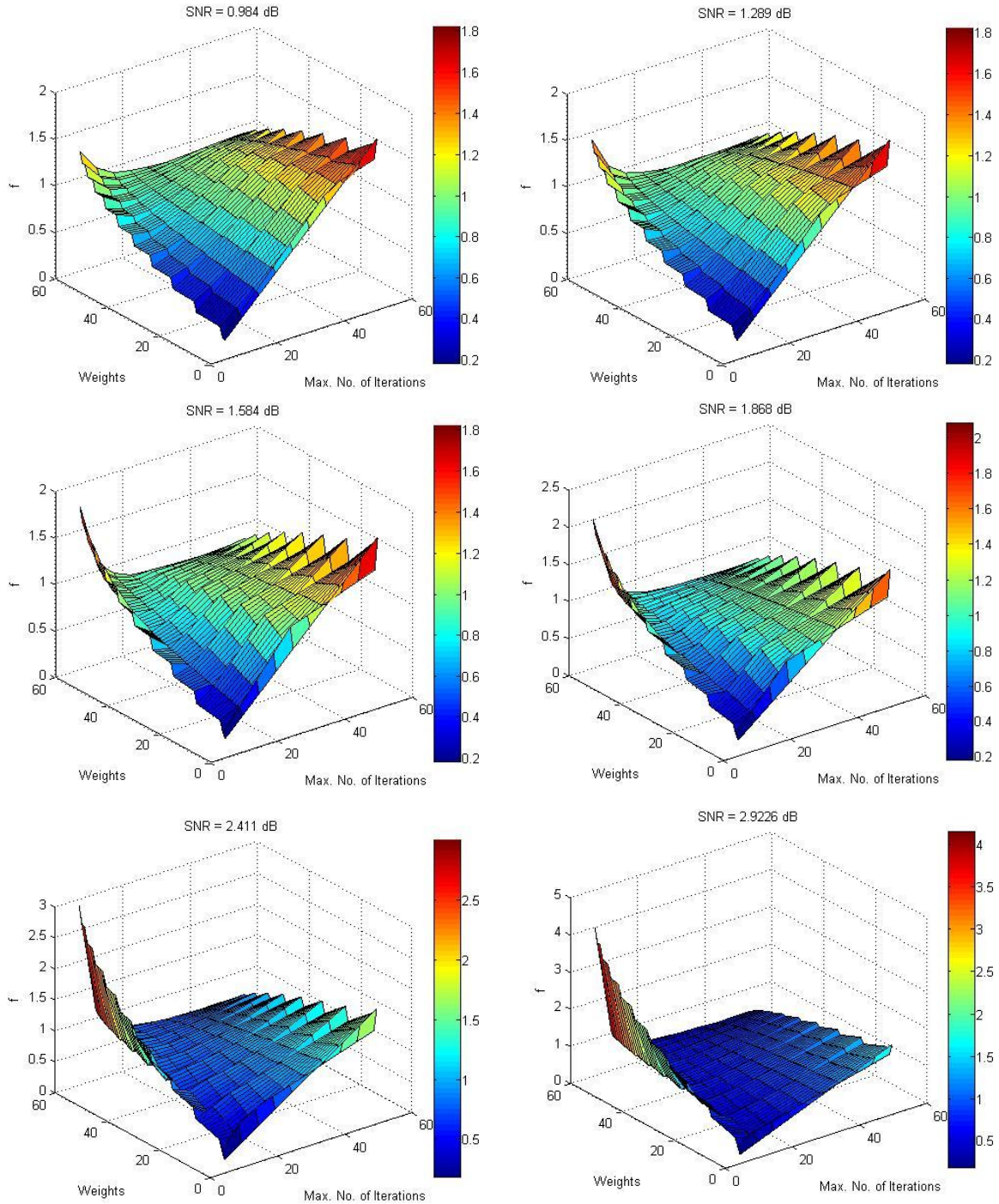


Figure 7.14: Surface plot of f for varying SNR, $Iter_{Max}$ and weights.

Case II:

In this case the optimum value of $Iter_{Max}$ for a given SNR can be found by minimizing one of the parameters of *error*, *delay* and *energy* while constraining the other two parameters.

1. For a given SNR, the optimum value of $Iter_{Max}$ can be obtained by minimizing *error* and constraining the *delay* and *energy* as shown below

$$f(SNR, Iter_{Max}) = \min \text{error} \quad (7.11)$$

such that $delay < T_{min}$ and
 $energy < E_{min}$.

For a given set of values of T_{min} and E_{min} , $Iter_{Max}$ can be obtained from Tables 7.8 and 7.10 respectively.

2. For a given SNR, the optimum value of $Iter_{Max}$ can be obtained by minimizing *delay* and constraining the *error* and *energy* as shown below

$$f(SNR, Iter_{Max}) = \min \text{delay} \quad (7.12)$$

such that $error < P_e$ and
 $energy < E_{min}$.

For a given set of values of P_e and E_{min} , $Iter_{Max}$ can be obtained from Tables 7.7 and 7.10 respectively.

3. For a given SNR, the optimum value of $Iter_{Max}$ can be obtained by minimizing *energy* and constraining the *delay* and *error* as shown below

$$f(SNR, Iter_{Max}) = \min \text{energy} \quad (7.13)$$

such that $delay < T_{min}$ and
 $error < P_e$.

For a given set of values of T_{min} and P_e , $Iter_{Max}$ can be obtained from Tables 7.8 and 7.7 respectively.

There are always constraints on *error* performance, *energy/power* and *delay* to develop designs for real time applications. *Error* and *delay* determine the quality of the performance and the speed. *Energy/power* influence the battery power required. For example, to find an optimum maximum number of decoding iterations for a given SNR of 1.868 dB, minimizing the *error* when *delay* and *energy* are constrained to less than 600 ns and 4000 pJ respectively can be obtained as follows.

For *delay* to be less than 600 ns, the corresponding $Iter_{Max}$ can be obtained from Table 7.8 and is equal to 20. The $Iter_{Max}$ when *energy* is less than 4000 pJ can be obtained from Table 7.10 and is equal to 40. In order to satisfy both *delay* and *energy* constraints $Iter_{Max}$ cannot exceed 20. From Table 7.7, *error* is minimum for a given SNR and $Iter_{Max}$ if it has less number of erroneous

codewords. For SNR of 1.868 dB, *error* is minimum when $Iter_{Max}$ is 50. But for given constraints on *delay* and *energy*, the minimum *error* occurs for $Iter_{Max}$ of 20. Therefore, $Iter_{Max}$ in this case is 20. Similarly $Iter_{Max}$ can be obtained for other constraints on *error*, *delay* and *energy* as explained for cases I and II. By repeating this procedure on other parity-check matrices, the optimum value of $Iter_{Max}$ for that particular parity-check matrix at a given SNR can be obtained.

CHAPTER 8 - Conclusion

Low-density parity-check codes are being used in many applications because of their excellent coding performance. A flexible hardware encoder and decoder for LDPC codes which would aid in the future development of cognitive radio are developed. The design methodologies used for the implementation of both a LDPC encoder and decoder are flexible in terms of parity-check matrix, code rate and code length.

In this work, four encoder designs are proposed yielding very high data rates. The encoder designs presented can fit on currently available FPGAs. As the density and size of FPGAs continue to increase and the demand from high-speed applications also increase, encoders similar to this will become more commonplace. The data rate of these encoders is restricted by the I/O serialization required to convert between the serial data stream(s) and the corresponding block processing.

Two of these encoder designs can be used for both structured and non-structured LDPC codes. These designs are more efficient for small code lengths while requiring large FPGAs for longer code lengths. The two other encoder designs are proposed for structured LDPC codes because of their use in IEEE communication standards. Using structured LDPC codes decreases the encoding complexity and also provides design flexibility. The same design methodology with minor modifications can also be used for similar structured LDPC codes defined in other standards. One of the structured encoder designs has flexibility in terms of both the code rate and code length. This design methodology does not require re-synthesis of the Verilog code to change the code rate and code length of the LDPC encoder. The design flexibility in both code rate and code lengths can be utilized in a real time implementation of LDPC codecs for new technologies such as cognitive radio which needs physical reconfigurability. A flexible encoder design for structured LDPC codes is also implemented on both an FPGA and an ASIC.

In this work, a decoder is also designed for LDPC codes. The design methodology does not consider any structure in the LDPC codes. Hence it is applicable to both structured and non-structured LDPC codes. The decoder has to be optimized for BER performance, hardware complexity, and power consumption. The maximum number of decoding iterations used for the decoding process plays an important role in determining the decoder BER performance, latency and power consumption. Most of the earlier decoder designs found to be available prior to this

work, always decode for a fixed number of iterations after which an estimate of the codeword is calculated. This leads to unnecessary delay and power consumption, especially in higher SNRs where the correct codeword is available within a few iterations. In [55], the parity of the normal variable-to-check messages is checked after each iteration. If the parity check is satisfied then the codeword is estimated at the beginning of the next iteration and the decoding process is stopped. In [42], the codeword is estimated after every iteration but it is validated in the next iteration. So these two methods would take an extra iteration to stop the process after the decoder decoded the correct codeword. In our design, the codeword is estimated and checked for validity after every iteration. In a clock cycle, a complete decoding iteration is performed; codeword is estimated and is validated. The area required to implement this logic is very small when compared to the rest of the design. The decoding process is stopped if the estimated codeword is correct; otherwise it is continued until it reaches the maximum number of decoding iterations. This logic will decrease the decoding latency which in turn saves the power consumed by the chip and increases the data rate. The proposed decoder can be implemented on FPGAs for only small code lengths. However, for large code lengths it is shown that the design can be implemented on an ASIC.

The major contributions of this work can be summarized as follows:

- A generic encoder is designed that achieves high data rates. This design methodology can be used for both structured and any randomly generated regular and irregular LDPC codes.
- An encoder is designed for structured LDPC codes defined in the IEEE 802.16e standard. This design methodology can be used for other similar structured LDPC codes such as IEEE 802.11n.
- A flexible multi-code rate and multi-code length LDPC encoder is designed for structured LDPC codes defined in IEEE 802.16e standard accommodating code lengths ranging from 576-2304 with code rates of $1/2$, $2/3$, $3/4$ and $5/6$.
- A LDPC decoder is designed that can be used for both structured and any randomly generated regular and irregular parity-check matrices.
- Procedure to determine the optimum maximum number of decoding iterations for a given parity-check matrix and SNR is presented.

8.1 Future Work

Although significant advances have been made during this work, there are several areas in which further investigation would be useful.

- Decrease the latency involved in the computation of parity-check matrix from its corresponding base parity-check matrix.

Base parity-check matrices, H_{b1} , of the structured LDPC codes are stored on the chip to design a flexible encoder accommodating different code lengths and code rates. Based on the desired code length and code rate, the parity-check matrix is computed from its corresponding base parity-check matrix and is stored temporarily until the code rate or the code length is changed. This step needs to be performed only once for a desired code length and code rate. The latency involved in the computation of parity-check matrix may affect the overall latency of the encoder when the code rate and code lengths are changed frequently. The computation latency of H_1 can be reduced by using efficient multiplication and division modules. Latency can be further reduced by computing all the columns of H_1 in parallel.

- Stopping the decoding process

During simulations, it was observed that some codewords are not corrected even after performing the maximum number of decoding iterations. Identifying such codewords and stopping the decoding process would decrease the power consumption of the chip. Also this logic would decrease the decoding latency and increase the decoder data rate.

- A flexible LDPC codec system can be designed that could accommodate parity-check matrices of different standards.

CHAPTER 9 - References

- [1] C. E. Shannon, "A mathematical theory of communications," *Bell System Technical Journal*, 1948.
- [2] R. G. Gallager, "Low-density parity-check codes," M.I.T. Press, Cambridge, MA, 1963.
- [3] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inform.Theory*, vol. IT-8, pp. 21-28, Jan. 1962.
- [4] D. J. C. Mackay and R. Neal, "Near Shannon limit performance of low density parity check codes," *Electron Letter*, vol. 33, pp. 457-458, Mar. 1997.
- [5] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," *IEEE Conference on Communications*, pp. 1064-1070, 1993.
- [6] N. Weste and D. J. Akellern, "VLSI for OFDM," *IEEE Communication Magazine*, vol. 36, Issue 10, pp. 127-131, Oct. 1998.
- [7] S. Rajagopal, S. Rixner and J. R. Cavallaro, "A programmable baseband processor design for software defined radios," *45th Midwest Symposium on Circuits and Systems*, vol. 3, pp. 413 – 416, Aug. 2002.
- [8] A. Shah, "An introduction to software radio," Vanu Inc.
<http://vanu.com/resources/intro/SWRprimer.pdf>, Nov. 2006.
- [9] J. Mitola, "Cognitive radio," Ph.D. thesis, KTH, Stockholm, 2000.
- [10] J. Mitola III and G. Q. Maguire Jr., "Cognitive radio: making software radios more personal," *IEEE Personal Communications*, vol. 6, Issue 4, pp. 13-18, Aug. 1999.
- [11] P. Mohanem, "Cognitive trends in making: future of networks," *15th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, vol. 2, pp. 1449-1454, Sept. 2004.
- [12] B. Fette, "Three obstacles to cognitive radio,"
<http://www.eetimes.com/showArticle.jhtml?articleID=29100657&printable=true>, Nov. 2006.
- [13] S. Haykin, "Cognitive radio: brain-empowered wireless communications," *IEEE Journal on Selected Areas in Communications*, vol. 23, Issue 2, pp. 201-220, Feb. 2002.

- [14] Q. Zhang, F. W. Hoeksema, A. B. J. Kokkeler and G. J. M. Smit, "Towards cognitive radio for emergency networks," <http://eprints.eemcs.utwente.nl/2758/01/BookChapter.pdf>, Nov. 2006.
- [15] B. Fette, "Cognitive radio shows great promise," <http://www.cotsjournalonline.com/magazine/articles/view/100206>, Nov. 2006.
- [16] W. Krenik and A. Batra, "Cognitive radio techniques for wide area networks," *42nd Proceedings of Design Automation Conference*, pp. 409-412, June 2005.
- [17] T. A. Weiss and F. K. Jondral. "Spectrum pooling: an innovative strategy for the enhancement of spectrum efficiency". *IEEE Communication Magazine*, vol. 24, no. 3, pp. S8-S14, Mar. 2004.
- [18] J. Walko, "Cognitive radio," *IEE Review*, vol. 51, Issue 5, pp. 34-37, May 2005.
- [19] R. Berezdivin, R. Breinig and R. Topp, "Next-generation wireless communications concepts and technologies," *IEEE Communications Magazine*, vol. 40, Issue 3, pp. 108-116, Mar. 2002.
- [20] B. Kurkoski, "Introduction to low-density parity check codes," http://www.lit.ice.uec.ac.jp/kurkoski/teaching/portfolio/uec_s05/S05-LDPC%20Lecture%201.pdf, April 2010.
- [21] M. Luby, M. Mitzenmacher, A. Shokrollahi and D. Spielman, "Improved low density parity check codes using irregular graphs," *IEEE Trans. Inform. Theory*, vol. 47, pp. 585-598, 2001.
- [22] T. J. Richardson and R. L. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 638-656, Feb. 2001.
- [23] T. Zhang and K. K. Parhi, "VLSI implementation-oriented (3, k)-regular low-density parity-check codes," *Proc. IEEE Workshop on Signal Processing Systems (SiPS): Design and Implementation*, pp. 25-36, 2001.
- [24] T. Zhang and K. K. Parhi, "A class of efficient-encoding generalized low-density parity-check codes," *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 4, pp. 2477 -2480, May 2001.
- [25] H. Zhong and T. Zhang, "Joint code-encoder-decoder design for LDPC coding system VLSI implementation," *proceedings of the 2004 International Symposium on Circuits and Systems*, vol. 2, pp. 389-392, 2004.

- [26] R. Echard and S. Chang, "The π -rotation low-density parity check codes," *IEEE Global Telecommunications Conference*, vol. 2, pp. 980-984, Nov. 2001
- [27] S. Kim, G. E. Sobelman and J. Moon, "Parallel VLSI architectures for a class of LDPC codes," *IEEE International Symposium on Circuits and Systems*, vol. 2, pp 93-96, May 2002.
- [28] L. Miles, J. Gambles, G. Maki, W. Ryan and S. Whitaker, "An (8158, 7136) low-density parity-check encoder," *IEEE Proceedings of the Custom Integrated Circuits Conference*, pp. 699-702, Sept. 2005.
- [29] D. U. Lee, W. Luk, C. Wang, C. Jones, M. Smith and J. Villasenor, "A flexible hardware encoder for low-density parity-check codes," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 101 - 111, April 2004.
- [30] Z. Khan, T. Arslan and S. Macdougall, "A real time programmable encoder for low density parity check code as specified in the IEEE P802.16E/D7 standard and its efficient implementation on a DSP processor," *IEEE International SOC Conference*, pp. 17-20, Sept. 2006.
- [31] Z. Khan and T. Arslan, "A real time programmable encoder for low density parity check code targeting a reconfigurable instruction cell architecture," *IEEE International Conference on Field Programmable Technology*, pp. 245-248, Dec 2006.
- [32] Z. Khan and T. Arslan, "Pipelined implementation of a real time programmable encoder for low density parity check code on a reconfigurable instruction cell architecture," *Proceedings of the conference on Design, automation and test in Europe*, pp. 349-354, 2007.
- [33] J. K. Kim, H. Yoo, "Efficient encoding architecture for IEEE 802.16e LDPC codes," *IEICE Trans. Fundamentals*, vol. E91-A, No.12, Dec. 2008.
- [34] Y. Sun, M. Karkooti and J. R. Cavallaro, "High throughput, parallel, scalable LDPC encoder/decoder architecture for OFDM systems," *IEEE Dallas/CAS Workshop on Design, Applications, Integration and Software*, pp. 39-42, Oct. 2006.
- [35] "IEEE 802.16 LDPC encoder/decoder core,"
<http://www.turbobest.com/WhitePaper80216eLDPC.pdf>, May 2010.
- [36] "802.16 LDPC encoder v1.0,"

http://www.xilinx.com/support/documentation/ip_documentation/ldpc_802_16_enc_v1_0.pdf, May 2010.

- [37] D. J. C. Mackay, "Good error-correcting codes based on very sparse matrices," *IEEE International Symposium on Information Theory*, vol. 45, Issue 2, pp. 399-431, 1999.
- [38] M. P. C. Fossorier, M. Mihaljevic and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Transactions on Communications*, vol. 47, Issue 5, pp. 673-680, 1999.
- [39] E. Eleftheriou, T. Mittelholzer and A. Dholakia, "Reduced-complexity decoding algorithm for low-density parity-check codes," *Electronics letters*, vol. 37, Issue 2, pp. 102-104, 2001.
- [40] J. Zhang, M. Fossorier, D. Gu and J. Zhang, "Two-dimensional correction for min-sum decoding of irregular LDPC codes," *IEEE Communications Letters*, vol. 10, Issue 3, pp. 180-182, 2006.
- [41] M. Jiang, C. Zhao, L. Zhang and E. Xu, "Adaptive offset min-sum algorithm for low-density parity check codes," *IEEE Communications Letters*, vol. 10, Issue 6, pp. 483-485, 2006.
- [42] Q. Wang, K. Shimizu, T. Ikenaga and S. Goto, "A power-saved 1Gbps irregular LDPC decoder based on simplified Min-Sum algorithm," *International symposium on VLSI Design Automation and test*, pp. 1-4, 2007.
- [43] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *IEEE Journal of Solid-State circuits*, vol. 37, pp. 404-412, 2002.
- [44] L. Fanucci and F. Rossi, "A throughput / complexity analysis for the VLSI implementation of LDPC decoder," *Proceedings of the fourth IEEE International Symposium on Signal Processing and Information Technology*, pp. 409-412, 2004.
- [45] S. M. E. Hosseini, K. S. Chan and W. L. Goh, "A reconfigurable FPGA implementation of an LDPC decoder for unstructured codes," *2nd International Conference on Signals, circuits and Systems*, pp. 1-6, 2008.
- [46] Y. Zhu, Y. Chen, D. Hocevar and M. Goel, "A reduced-complexity, scalable implementation of low density parity check decoder," *IEEE Workshop on Signal Processing Systems Design and Implementation*, pp. 83-88, 2006.

- [47] P. Radosavljevic, A. D. Baynast, M. Karkooti, and J. R. Cavallaro, "Multi-rate high-throughput LDPC decoder: tradeoff analysis between decoding throughput and area," *IEEE 17th International Symposium on Personal, Indoor and Mobile radio Communications*, pp. 1-5, 2006.
- [48] M. Karkooti, P. Radosavljevic and J. R. Cavallaro, "Configurable, high throughput, irregular LDPC decoder architecture: tradeoff analysis and implementation," *Proceedings of the IEEE 17th International conference on Application-Specific Systems, Architectures and Processors*, pp. 360-367, 2006.
- [49] P. Radosavljevic, A. D. Baynast, M. Karkooti and J. R. Cavallaro, "High-throughput multi-rate LDPC decoder based on architecture-oriented parity check matrices," *European Signal Processing Conference*, Sept. 2006.
- [50] "IEEE 802.16e (WiMAX) LDPC decoder IP core,"
http://www.hitechglobal.com/ipcores/WiMAX_LDPC_Decoder.htm, May 2010.
- [51] Z. H. Kashani and M. Shiva, "Power optimised channel coding in wireless sensor networks using low-density parity-check codes," *Institution of Engineering and Technology Communications*, vol. 1, pp. 1256-1262, 2007.
- [52] L. Yijun, M. Ellassal and M. Bayoumi, "Power efficient architecture for (3,6)-regular low-density parity-check code decoder," *Proceedings of the 2004 International Symposium on Circuits and Systems*, vol. 4, pp. 81-84, 2004.
- [53] M. M. Mansour and N. R. Shanbhag, "Low-power VLSI decoder architectures for LDPC code," *International Symposium on Low Power Electronics and Design*, pp. 284-289, 2002.
- [54] W. Wang, G. Choi, "Minimum-energy LDPC decoder for real-time mobile application," *Proceedings of the conference on Design, automation and test in Europe*, pp. 343-348, 2007.
- [55] A. Darabiha, A. C. Carusone and F. R. Kschischang, "Power reduction techniques for LDPC decoders," *IEEE Journal of Solid-State Circuits*, vol. 43, pp. 1835-1845, 2008.
- [56] D. J. C. Mackay, "Encyclopedia of sparse graph codes,"
<http://www.inference.phy.cam.ac.uk/mackay/codes/data.html>, April 2010.
- [57] M. M. Mansour and N. R. Shanbhag, "High-throughput LDPC decoders," *IEEE Transactions on Very Large Scale Integration Systems*, pp. 976-996, Dec. 2003.

- [58] P. Radosavljevic, A. D. Baynast and J. R. Cavallaro, "Optimized message passing schedules for LDPC decoding," *39th Asilomar Conference on Signals, Systems and Computers*, pp. 591-595, 2005.
- [59] Stratix II Architecture," http://www.altera.com/literature/hb/stx2/stx2_sii51002.pdf, March 2010.
- [60] "Quartus II introduction using verilog design,"
ftp://ftp.altera.com/up/pub/Tutorials/DE2/Digital_Logic/tut_quartus_intro_verilog.pdf,
March 2010.
- [61] Z. G. Vranesic, "Tutorial 1 – Using quartus II CAD software,"
http://www.eecg.toronto.edu/~zvonko/AppendixB_quartus.pdf, March 2010.
- [62] "PowerPlay power analysis," http://www.altera.com/literature/hb/qts/qts_qii53013.pdf,
March 2010.
- [63] Standard cell library download, <http://www.vtvt.ece.vt.edu/vlsidesign/download.php>,
March 2004.
- [64] "FreePDK 45nm: A variation-aware design kit for 45nm,"
<http://vcag.ecen.okstate.edu/projects/scells/OSUFreePDK.php>, March 2010.
- [65] Y. Pu, J. P. D. Gyvez, "Cadence SoC encounter RTL-to-GDSII system,"
http://www.es.ele.tue.nl/~gyvez/5kk60/Lab3_manual.pdf, March 2010.
- [66] "Place and route using cadence SOC Encounter,"
http://www.vtvt.ece.vt.edu/vlsidesign/tutorialCadence_socEncounter.php, March 2004.
- [67] S. Kopparthi and D. M. Gruenbacher, "A high speed flexible encoder for low density parity check codes," *49th IEEE International Midwest Symposium on Circuits and Systems*, Aug. 2006.
- [68] "Xilinx Virtex-II series FPGAs,"
<http://www.xilinx.com/publications/matrix/virtexmatrix.pdf>, April 2010.
- [69] F. Rivoallon, "Comparing Virtex-II and Stratix logic utilization,"
http://www.xilinx.com/support/documentation/white_papers/wp161.pdf, April 2010.
- [70] S. Kopparthi and D. M. Gruenbacher, "Implementation of a flexible encoder for structured low-density parity-check codes," *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing 2007*, pp. 438-441, Aug. 2007.

Appendix A - Design of a Convolutional Encoder in Verilog HDL

A design created in Verilog HDL is used to illustrate the procedures to synthesize and place and route a design in FPGA and ASIC using Quartus and Cadence respectively. In this appendix, Verilog HDL design of a 1/2 rate convolutional encoder with constraint length 7 is used to demonstrate these procedures.

A.1 Convolutional Encoder

Convolutional encoding is used in forward error correcting codes. Convolutional encoding is a bit-level encoding technique where it calculates and adds the redundant bits for every input data bit, based on the polynomials. A 1/2 rate convolutional encoder with constraint length 7 with polynomials defined as $g^{(0)} = 1111001 = (171)_8$ and $g^{(1)} = 1011011 = (133)_8$ is shown in Figure A.1.

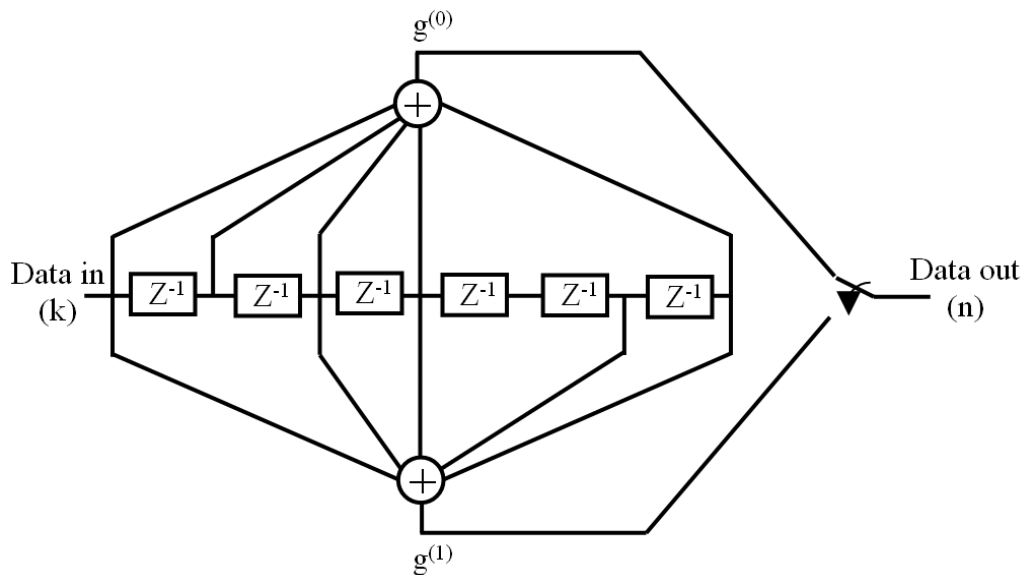


Figure A.1: A 1/2 rate convolutional encoder with constraint length 7.

Design using Verilog HDL for convolutional encoder shown in Figure A.1 is as follows:

convEncoder.v

```
module convEncoder (n, k, clk, reset);  
input k; // input to encoder  
input clk, reset; // clock and reset for the encoder
```

```
output [1:0] n; // outputs of the encoder
wire [1:0] n;
reg [6:0] po;

always @(negedge reset or posedge clk)
    if (~reset)
        po = 7'b0;
    else
        po = {k, po[6:1]};

assign n[0] = po[6]^po[4]^(po[3]^po[1]^po[0]);
assign n[1] = po[6]^po[5]^po[4]^po[3]^po[0];
endmodule
```


Appendix B - FPGA Implementation using Quartus

Quartus software makes it easy to implement a desired logic circuit by using a programmable logic device such as FPGA. In this appendix, the implementation of a design specified by Verilog HDL in Quartus II is presented as discussed in section 3.1. Graphical user interface is used to invoke Quartus II commands.

B.1 Creating a Project

Each logic circuit, or sub-circuit, being designed in Quartus II is called a project. The software works on one project at a time and keeps all the information for that project in a single directory. Start the Quartus II software and the main Quartus II display is as shown in Figure B.1. Procedure to implement the design in Quartus II using Verilog HDL is illustrated by using an example of the convolutional encoder presented in appendix A. New project needs to be created to start working on a new design.

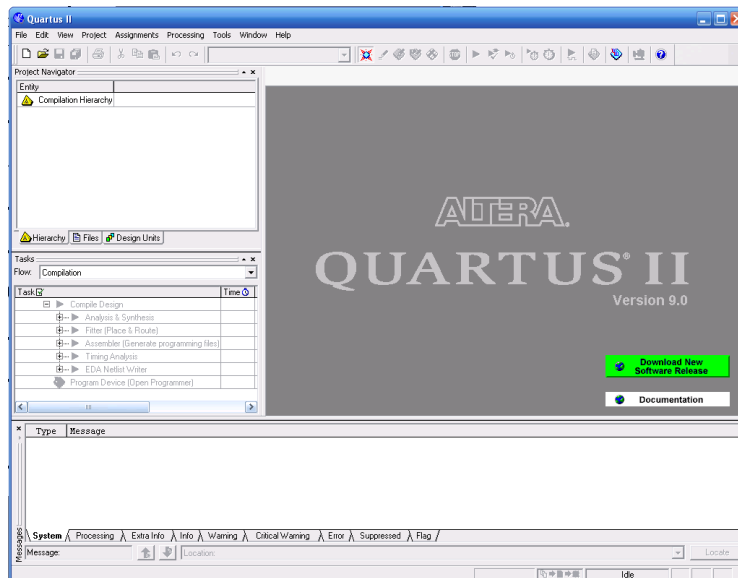


Figure B.1: The main Quartus II display.

New project is created by selecting File → New project wizard. A window pops up requesting name and directory of the project as shown in Figure B.2. Choose the working directory and the Verilog HDL file. The directory and the project name are assigned. The name of the project and top-level design entity of the project are same. Click next and another window pops up requesting the file name. Add all the files required for the project as shown in Figure B.3. Choose next. A window pops up asking for device type in which the designed circuit is

implemented as shown in Figure B.4. Choose Stratix as the target device family. From the list of available devices, choose the device called EP1S80F1508C5 which is the FPGA used on Altera's Stratix board. Press next, which opens the window in Figure B.5. The user can specify any third-party tools that should be used. A commonly used term for CAD software for electronic circuits is EDA (Electronic Design Automation) tools. Since third-party tools are not being used nothing is chosen in this window. Click next. A summary of the chosen setting appears in the screen shown in Figure B.6.

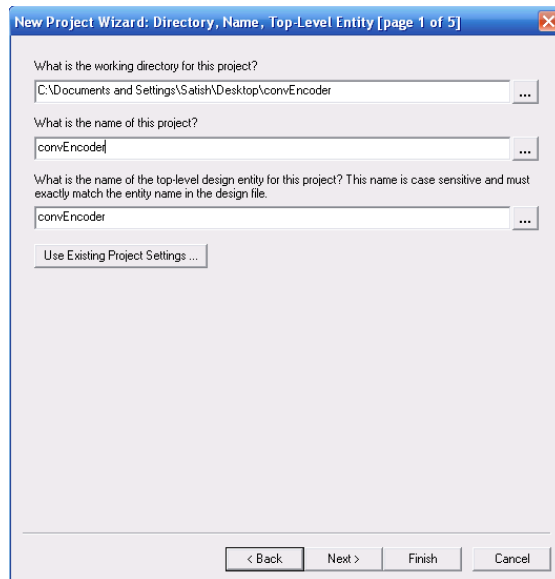


Figure B.2: Creation of new project.

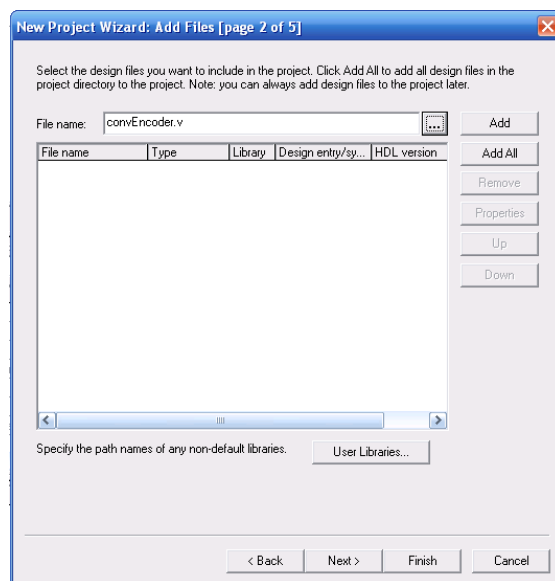


Figure B.3: Adding design files.

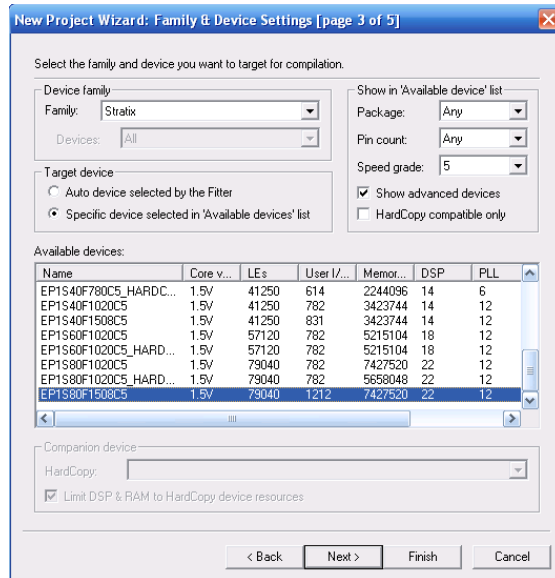


Figure B.4: Choose the device family and a specific device.

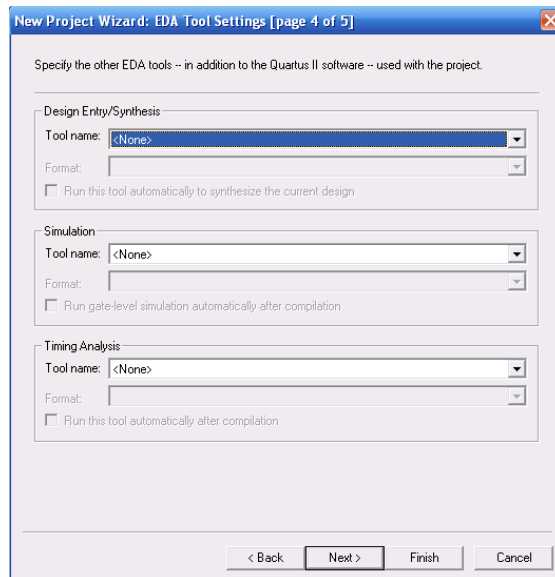


Figure B.5: Other EDA tools can be specified.

B.2 Compilation of the Project

Run the compiler by selecting Processing → Start compilation. As the compilation progresses through various stages, its progress is reported in a window on the left side of the Quartus II display. Successful or unsuccessful compilation is indicated in a pop-up box at the end of the run. Clicking ok leads to the Quartus II display in Figure B.7. In the message window, at the bottom of the Figure, various messages are displayed. In case of errors, the relevant messages

are shown. When the compilation is finished, a compilation report is generated. The flow summary of the compilation report is shown in Figure B.7. For the implementation of convolutional encoder on a Stratix FPGA chip requires 7 logic elements and 5 pins

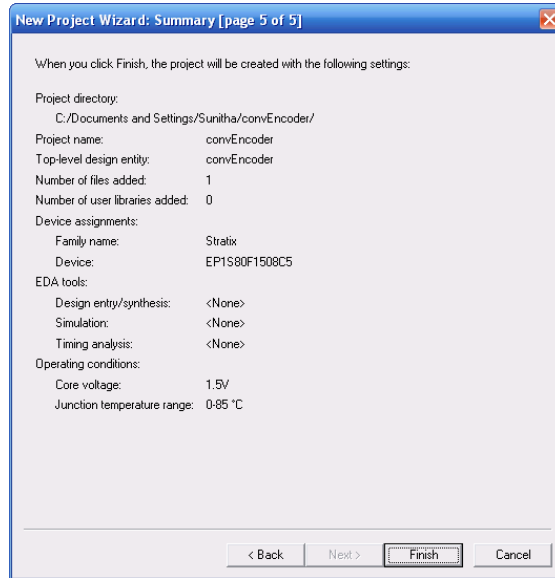


Figure B.6: Summary of the project settings.

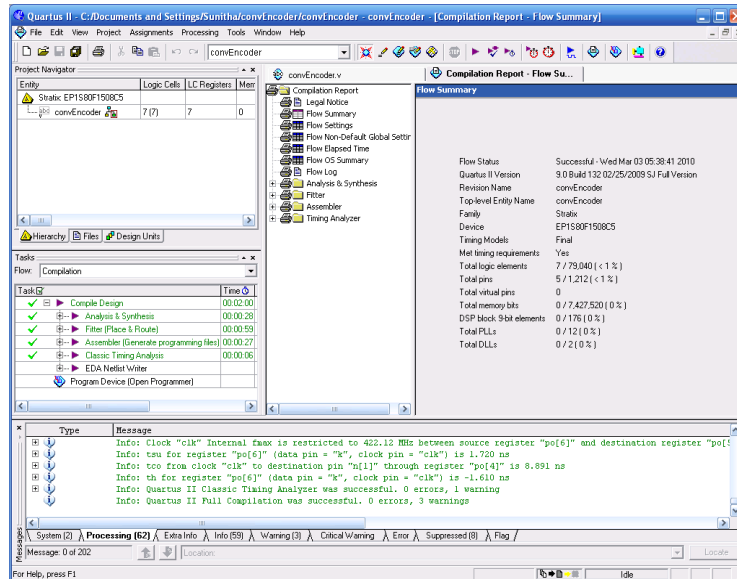


Figure B.7: Flow summary of the compilation report.

B.3 Timing Simulation

Timing simulations are performed on the design to check its behavior before implementing the design on the FPGA device. Before the design can be simulated, it is necessary

to create the desired waveforms to represent the input signals. All the inputs and outputs are specified. Open the waveform editor window by selecting File → New. A window pops up as shown in Figure B.8, choose vector waveform file and click ok. New waveform editor window opens as shown in Figure B.9.

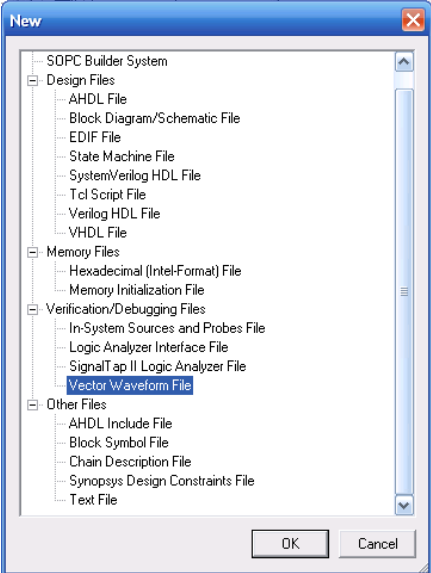


Figure B.8: Creating vector waveform file.

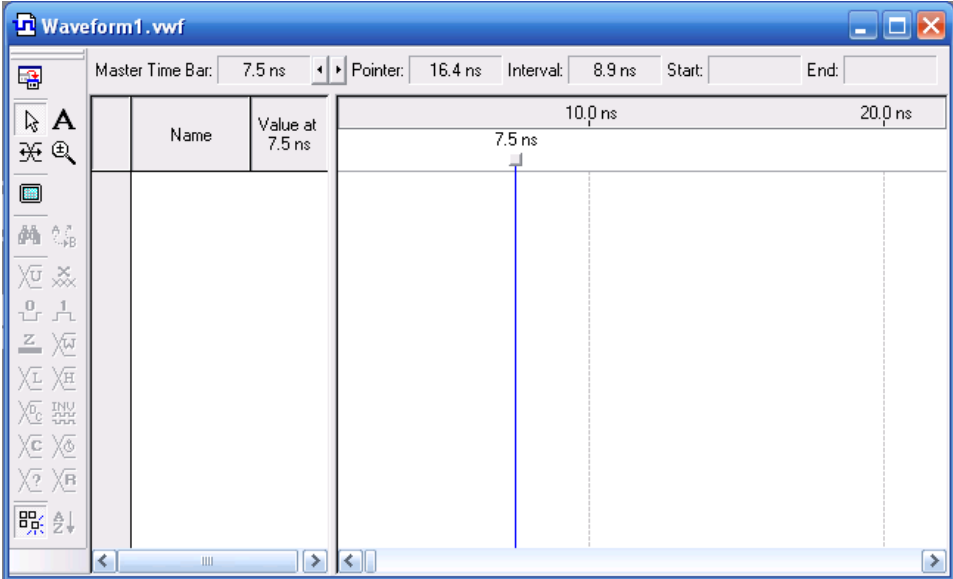


Figure B.9: Waveform editor window.

Set the desired simulation to run by selecting Edit → End Time and enter 200 ns in the dialog box. To include the input and output nodes of the design click Edit → Insert → Insert Node or Bus to open the window shown in Figure B.10. Click on node finder to open the window shown in Figure B.11 or type the name of the signal in Name part of the Figure B.10.

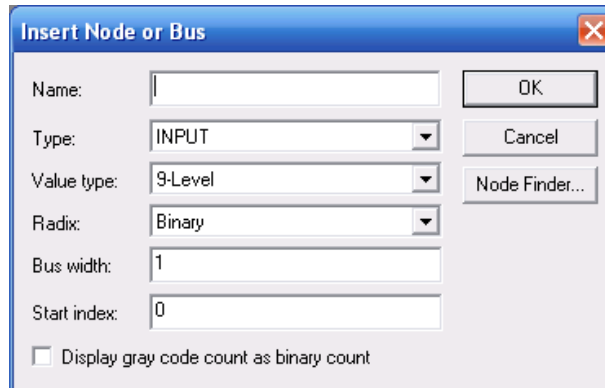


Figure B.10: Insert node or bus dialog box.

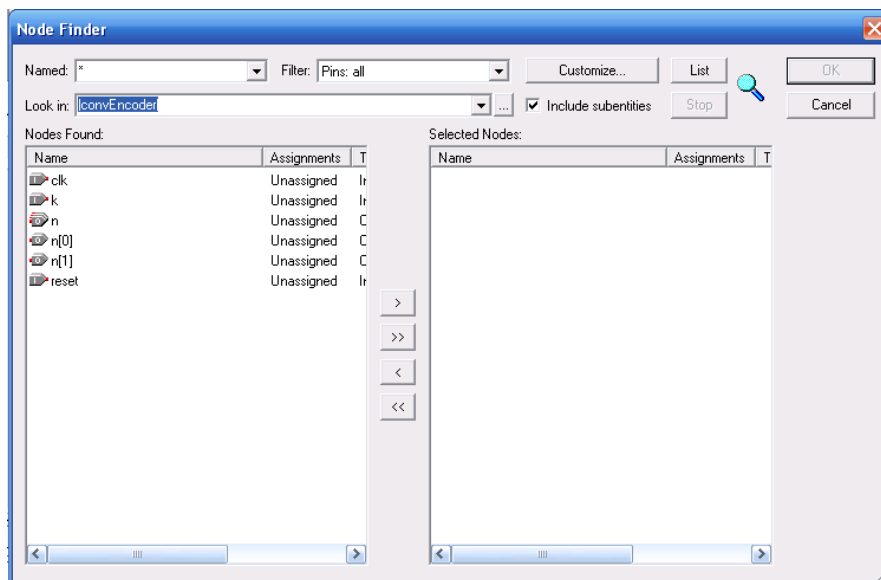


Figure B.11: Selecting nodes to insert into the waveform editor.

In Figure B.11 select pins, all in filter and click List. Select the required pin under nodes found on the left side of the window shown in Figure B.11 and click > sign to add the node to selected nodes on the right side of the window shown in Figure B.11. Add each pin or make multiple pin selections by simultaneously pressing shift button on the keyboard. Input nodes are assigned a desired waveform by selecting the waveform name and right click → Value to assign

desired value. Save the waveform file. Timing simulations can be performed by selecting Assignments → Settings → Simulator settings as shown in Figure B.12. Choose timing as the simulation mode and the waveform as the simulation input and click ok. Start simulation by selecting Processing → Start simulation. The obtained simulated waveform is as shown in Figure B.13.

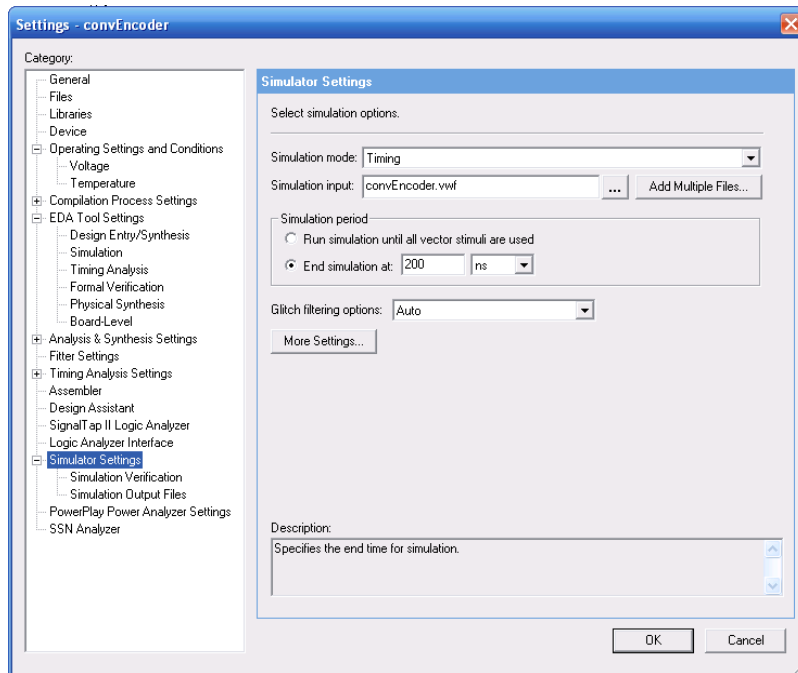


Figure B.12: Simulator settings.

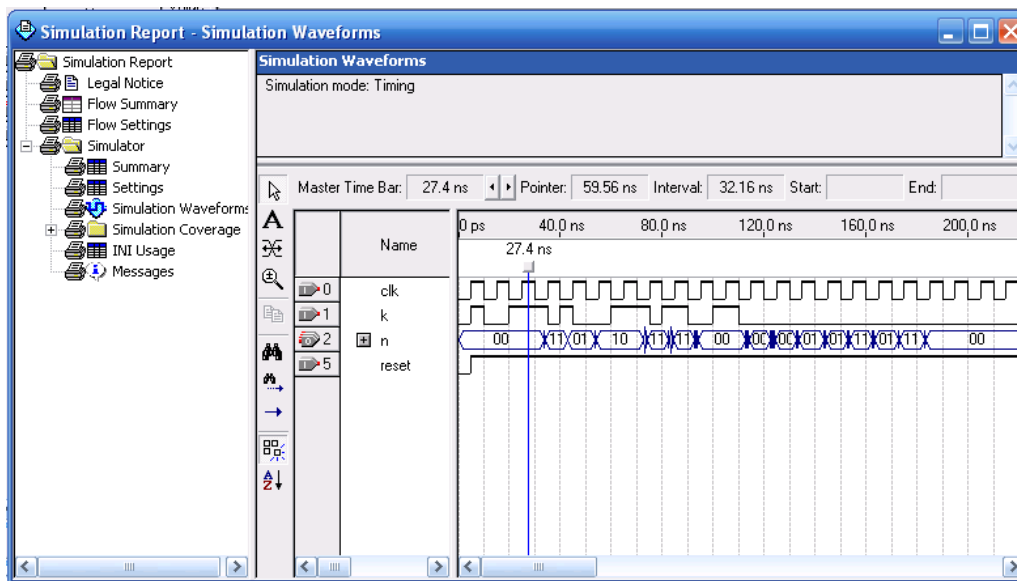


Figure B.13: Timing simulation report.

B.4 Power Analysis

PowerPlay power analyzer tool of Quartus II is used to perform power analysis. During simulator settings the simulation output files are created as shown in Fig B.14. Check the generate signal activity file under signal activity output for power analysis and specify the name of the .saf file. Signal activity file is generated when timing simulation is performed. PowerPlay power analyzer tool is started by selecting Processing → PowerPlay power analyzer tool and is shown in Figure B.15.

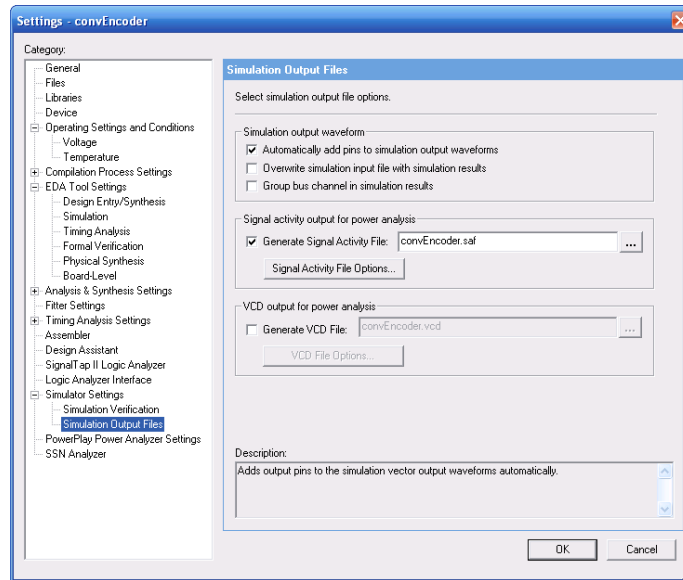


Figure B.14: Creating .saf file.

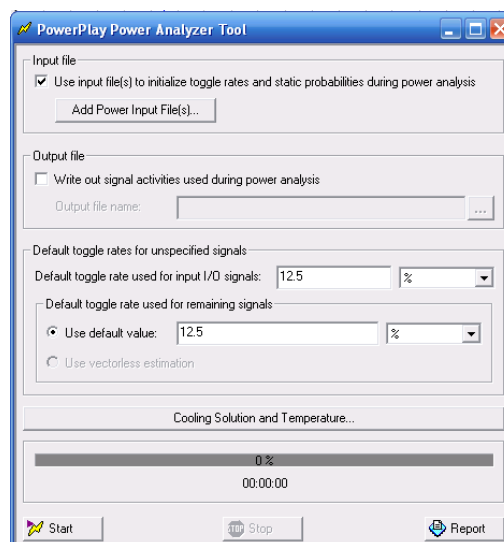


Figure B.15: PowerPlay power analyzer tool.

Check the option use input file(s) to initialize toggle rates and static probabilities during power analysis under Input file. Click add power input file(s) and a window pops up as shown in Figure B.16. In this window, check the option use input file(s) to initialize toggle rates and static probabilities during power analysis under select the power analyzer options. Click add and a window pops up as shown in Figure B.17. Choose the file under file name and select signal activity file under input file type and click ok. Click ok on power setting window. Power analysis is performed by clicking start button on PowerPlay power analyzer tool. When power analysis is finished a window pops up stating PowerPlay power analysis successful.

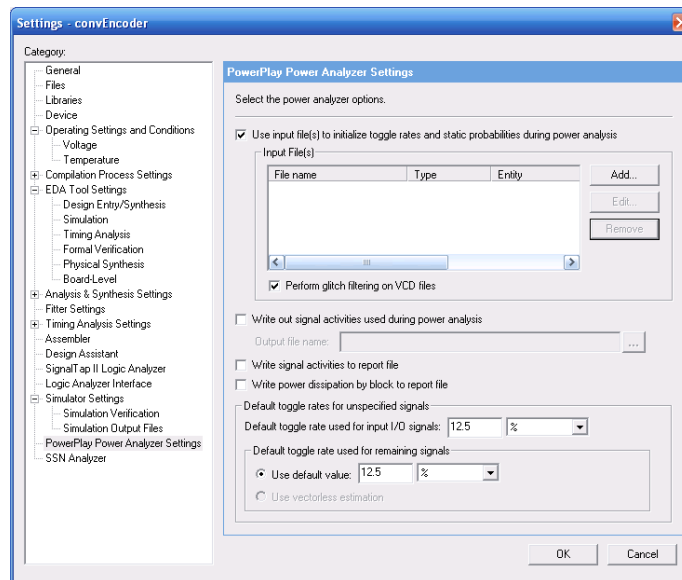


Figure B.16: Power settings.

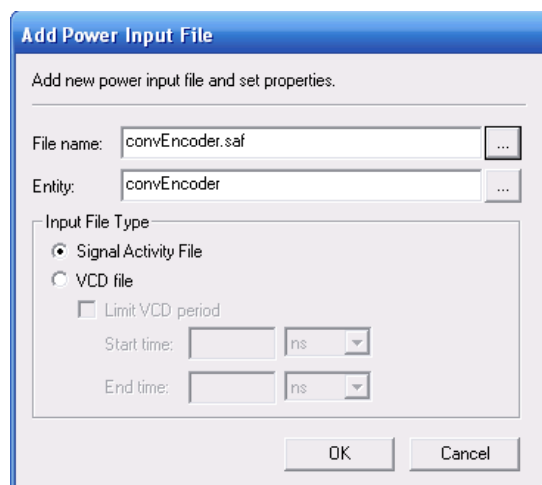


Figure B.17: Add power input file.

Click on the report button on PowerPlay power analyzer tool to view the PowerPlay power analyzer summary as shown in Figure B.18. Summary report consists of estimated total thermal, dynamic, static and I/O thermal power consumption of the design.

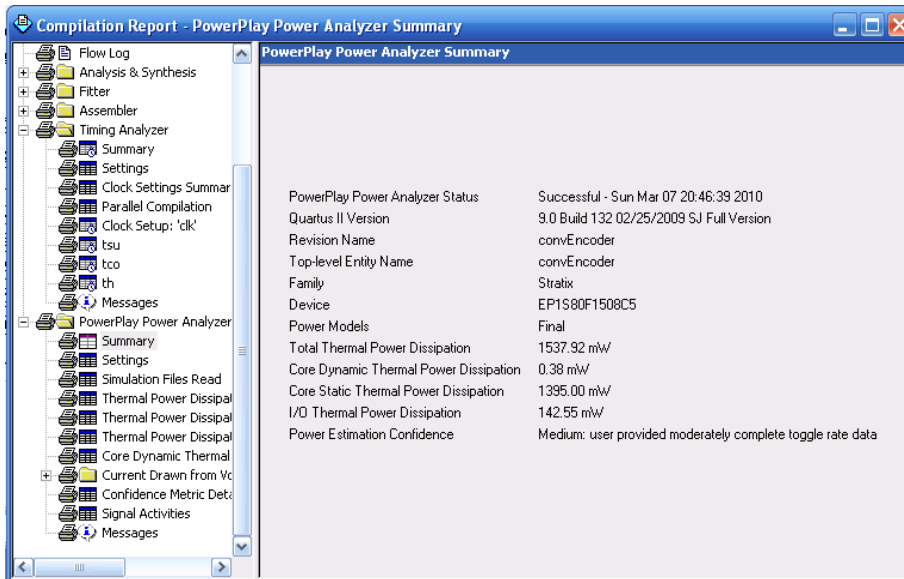


Figure B.18: PowerPlay power analyzer summary.

Appendix C - ASIC Implementation using Cadence

Procedure to synthesize and place and route a design in Cadence using an example is presented. The design in Verilog HDL is synthesized using RTL Compiler. The synthesized design is then place and route in Encounter.

C.1 Initial Setup

Standard cell library developed at Virginia Polytechnic Institute and State University is used to place and route the design. The standard cell library VTVT_TSMC250 design kit is downloaded from the following link <http://www.vtvt.ece.vt.edu/vlsidesign/download.php>. Unzip the files and copy the directory named vtv_tsmc250_release under your UNIX directory. The cadence files are available under directory vtv_tsmc250_release/Cadence_Libraries for the actual physical layout of the standard cells.

Cadence environmental variables need to be set up for verifying the design in Cadence Virtuoso which is explained in the later section.

- Create a script file called cadence-script and include the text below in the file
setenv USE_NCSU_CDK
setenv CDK_DIR /cadence/tools/dfII/local/ncsu_rel_1.5.1
- Source the cadence_script by using following commands
/bin/csh
source /idrive/cadence_script

Following steps are performed before synthesizing the design:

- Move the standard cell library layout directory vtv_tsmc250 into the cadence directory (directory created to run the project in this example).
- Add the library to cadence library manager by adding the line below in the cds.lib file
INCLUDE /cadence/tools/dfII/local/ncsu_rel_1.5.1/cdssetup/cds.lib
DEFINE vtv_tsmc250_nolabel ./vtv_tsmc250_nolabel
- Copy vtv_tsmc250.lib, vtv_tsmc250.lef, vtv_tsmc250.tf, vtv_SocE2df2.map and vtv_tsmc250_StreamIn.map into libs directory
- Copy display.drf from /cadence/tools/dfII/local/ncsu_rel_1.5.1/cdssetup/display.drf to the current directory

- Attach library vtv_tsmc250 to TSMC_CMOS025_DEEP techfile by doing the following steps:
 - Invoke cadence
icfb
 - In CIW → Tools → Technology File Manager → Attach
 - Design Library: vtv_tsmc250_nolabel
 - Technology Library: NCSU_TechLib_tsmc03d
 - The standard cell views are now available in the Library Manager

C.2 Synthesis of Verilog HDL Modules in RTL Compiler

In this section the steps followed to synthesize the design using RTL Compiler are presented.

- Invoke RTL Compiler
rc -gui
- Run the script
File → source script → rc.tcl

The synthesized convolutional encoder design is shown in Figure C.1.

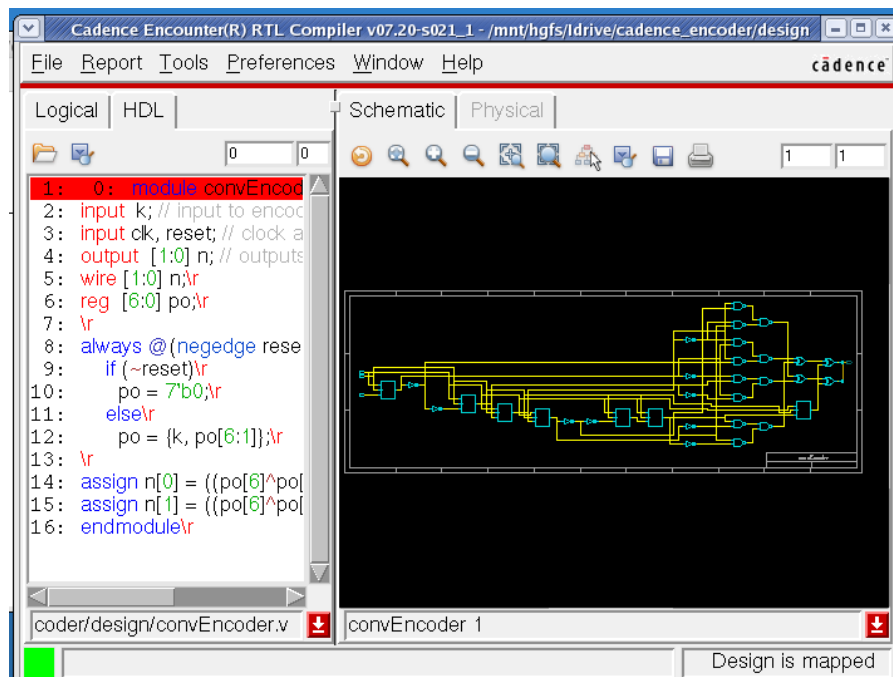


Figure C.1: Synthesized convolutional encoder in RTL Cadence.

The script file used to generate the synthesized design is given below:

RTL script file rc.tcl

```
# Step1: Specify Verilog HDL design files
# All HDL files, separated by spaces
set hdl_files {/mnt/hgfs/Idrive/cadence_encoder/design/convEncoder.v}
# The Top-level Module
set DESIGN convEncoder
# Set clock pin name in design.
Set clkpin clk
# Target frequency in MHz for optimization
set delay 100
#*****/
# Target Library path is set
# NO further changes past this point
set_attribute lib_search_path
{/cadence/tools/dfII/local/ncsu_rel_1.5.1/lib/NCSU_TechLib_tsmc03d}
set_attribute library {/mnt/hgfs/Idrive/cadence/libs/vtvt_tsmc250.lib}
# Verilog HDL files are read
read_hdl ${hdl_files}
# Design is elaborated
elaborate $DESIGN
# Apply Constraints
set clock [define_clock -period ${delay} -name ${clkpin} [clock_ports]]
external_delay -input 0 -clock clk [find / -port ports_in/*]
external_delay -output 0 -clock clk [find / -port ports_out/*]
# Sets transition to default values for Synopsys SDC format, fall/rise
# 400ps
dc::set_clock_transition .4 clk
# Design is checked
check_design -unresolved
```

```

report timing -lint
# Synthesis of the design
synthesize -to_mapped
# Analyzing and reporting
report timing > timing.rep
report gates > cell.rep
report power > power.rep
# Generating synthesized design
write_hdl -mapped > ${DESIGN}.vh
write_sdc > ${DESIGN}.sdc
puts "Synthesis Finished!      "
puts "Check timing.rep, area.rep, gate.rep and power.rep for synthesis results"

```

C.3 Place and Route using Cadence Encounter

Once the design is synthesized in Cadence RTL it is then place and route in Cadence Encounter. The following steps are performed to place and route the design [67].

Step 1: Invoke Encounter: Invoke Encounter from the design directory by using the following command

- encounter

Step 2: Import Design: Import the synthesized design by selecting the following options under basic and advanced tab as shown in Figures. C.2 and C.3.

- Design → Import design

Basic tab → Verilog Netlist:	Files: convEncoder.vh
	By User: convEncoder
Timing Libraries:	Common Timing Libraries: vtv_tsmc250.lib
	LEF Files: vtv_tsmc250.lef
	Timing Constraint File convEncoder.sdc (optional)

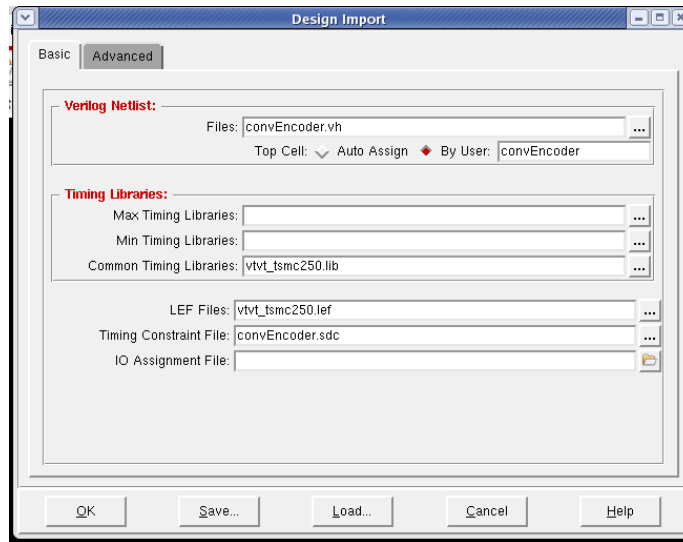


Figure C.2: Basic design import.

- Advanced Tab → Power Power Nets: vdd
Ground Nets: gnd

Leave all the other fields as default. Click OK.

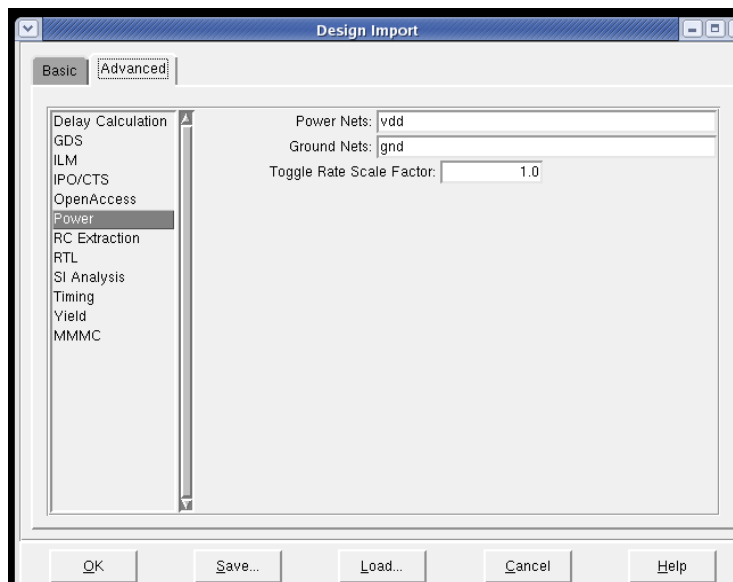


Figure C.3: Advanced design import.

- After importing the design a window appears as shown in Figure C.4 showing the initial floorplan.

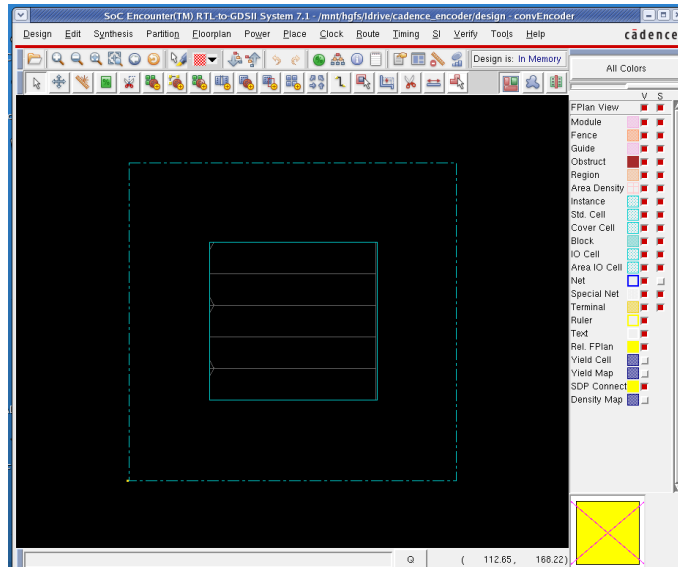


Figure C.4: After importing the design.

Step 3: Floor Planning: Depending on the size of the design the floorplan is specified. Figures C.5 and C.6 show the specify floorplan and after floorplan windows respectively.

- Floorplan → Specified Floorplan

Basic → Die Size by: Width: 500, Height: 500
 Core Margins by: Core to IO Boundary
 Core to Left: 38 Core to Top: 38
 Core to Right: 38 Core to Bottom: 38

Click OK.

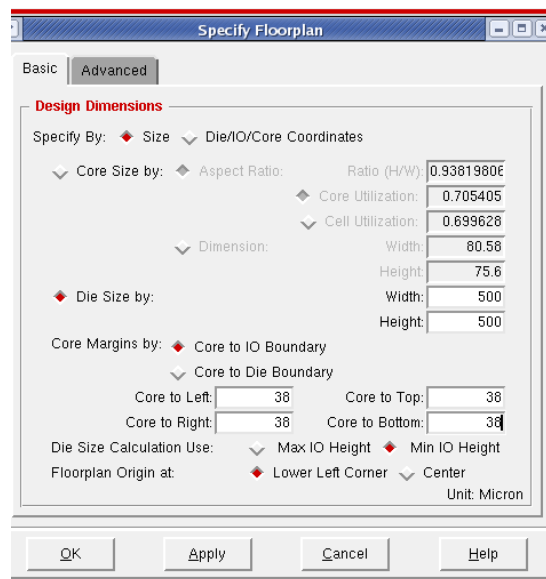


Figure C.5: Specify floorplan.

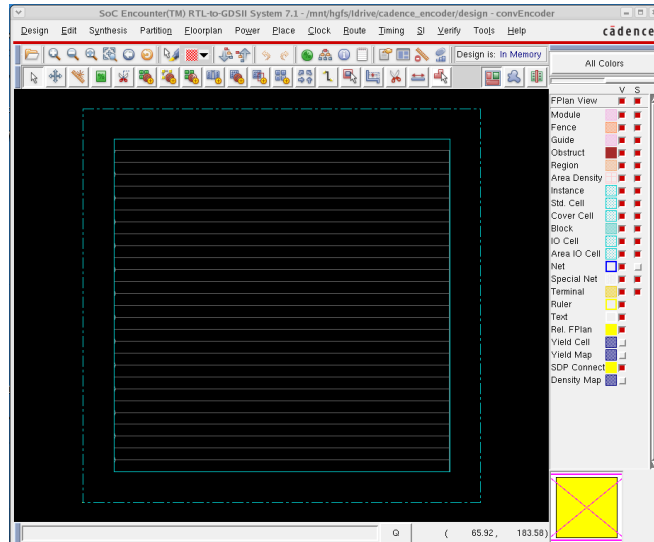


Figure C.6: After floorPlan.

Step 4: Power Planning: Rings and stripes are added. Windows for add rings and after adding rings are shown in Figures. C.7 and C.8 respectively. Similarly windows for add stripes and after adding stripes are shown in Figures. C.9 and C.10 respectively.

- Power → Power Planning → Add Rings

Basic → Ring Configuration: Layer: Width: 10.8, Spacing: 2.16

Click OK.

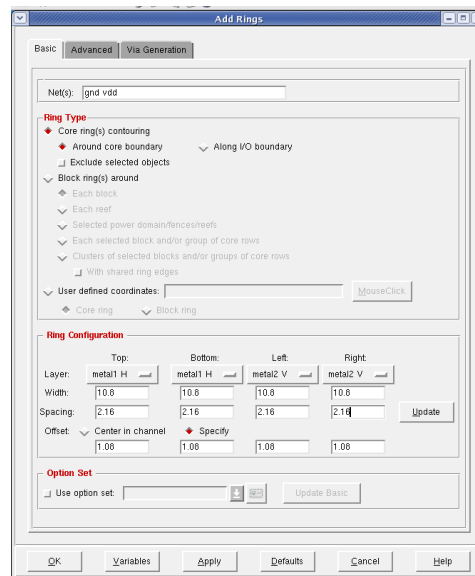


Figure C.7: Add rings.

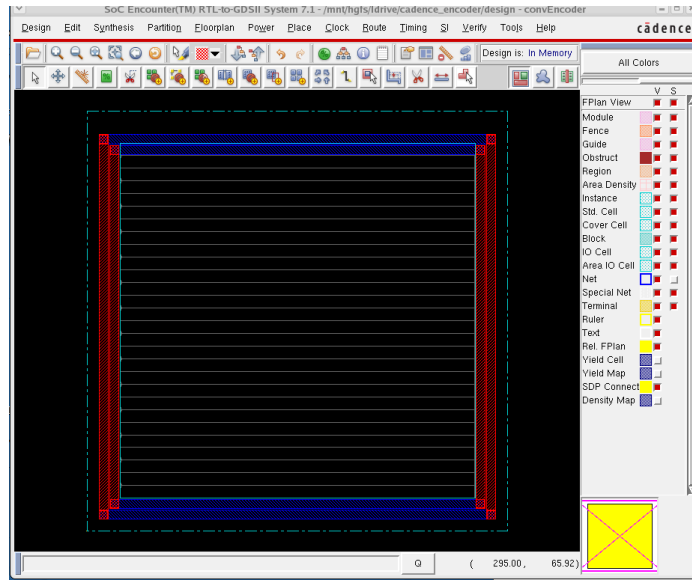


Figure C.8: After adding rings.

- Power → Power Planning → Add Stripes

Set Configuration: Layer: Metal2

Direction: Vertical

Width: 10.8

Spacing: 2.16

Click OK.



Figure C.9: Add stripes.

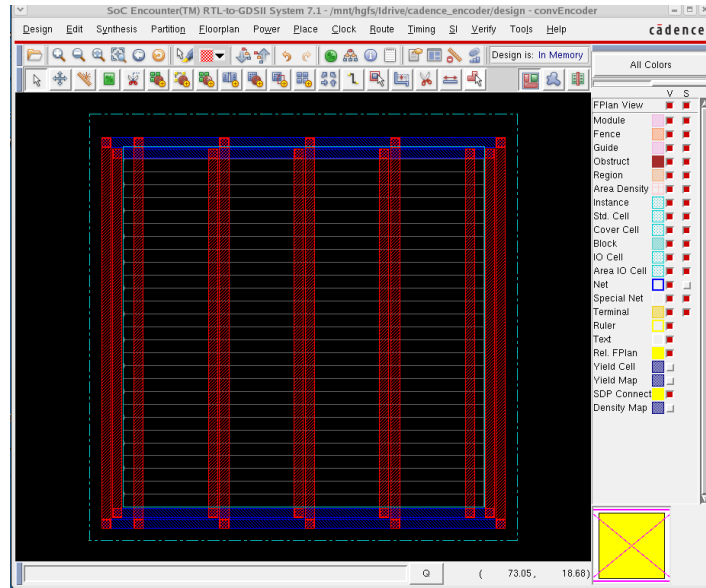


Figure C.10: After adding stripes.

Step 5: Special Route: SRoute is performed to do the final power routing and is shown in Figure C.11. Figure C.12 shows the routed design.

- Route
- Route → Special Route
- Click OK

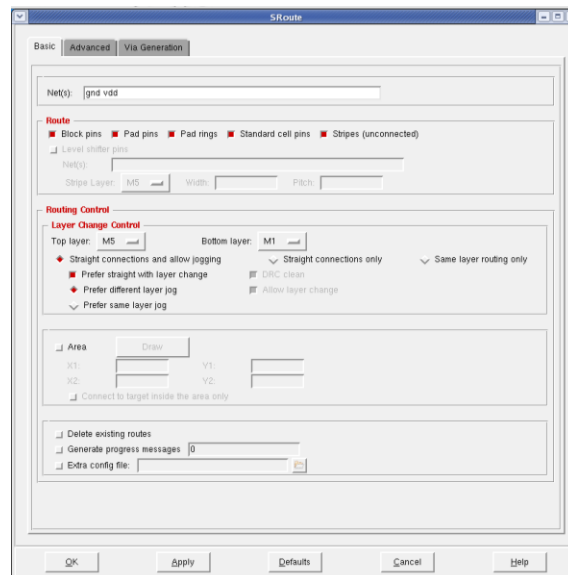


Figure C.11: Special route.

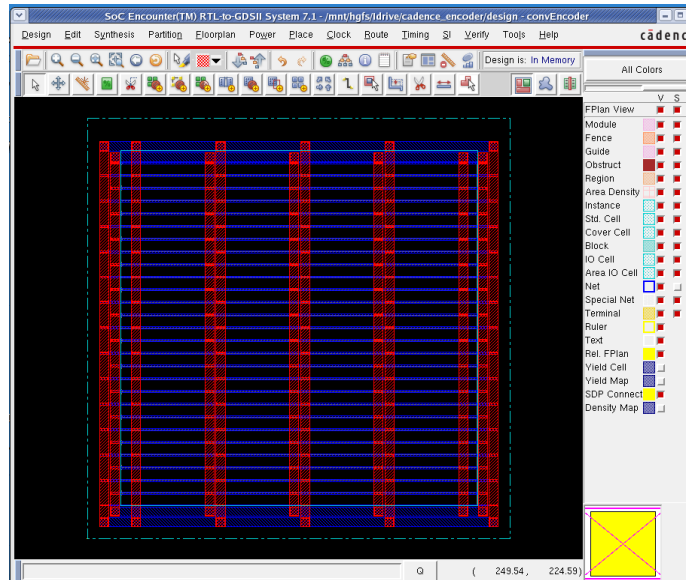


Figure C.12: After special route.

Step 6: Place: Design is Placed by filling the form as shown in Figure C.13. Change the view from floorplan to physical view by selecting the appropriate view as shown in Figure C.14.

- Place

Place → Standard Cells

Chose Run Full Placement

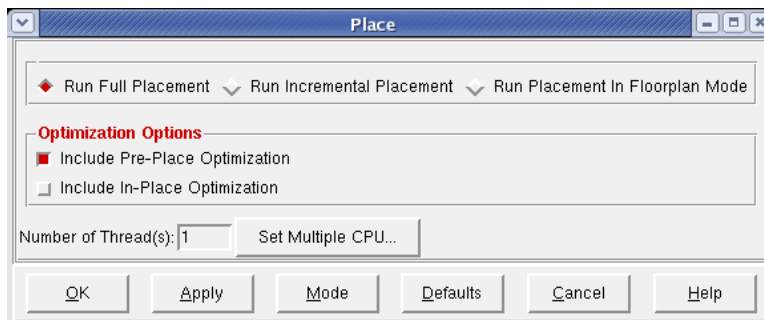


Figure C.13: Place.

- Set View option to Physical View

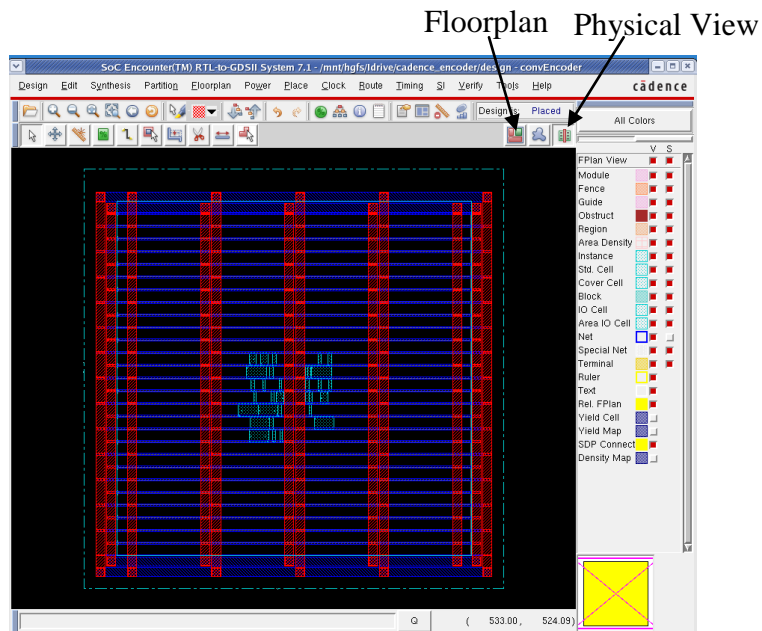


Figure C.14: After placing cells.

Step 7: NanoRoute: For global routing nanoRoute is used. Figures C.15 and C.16 show the options chosen for NanoRoute and window after NanoRoute is performed respectively.

- Route
Route → Nanoroute → Route
Click OK

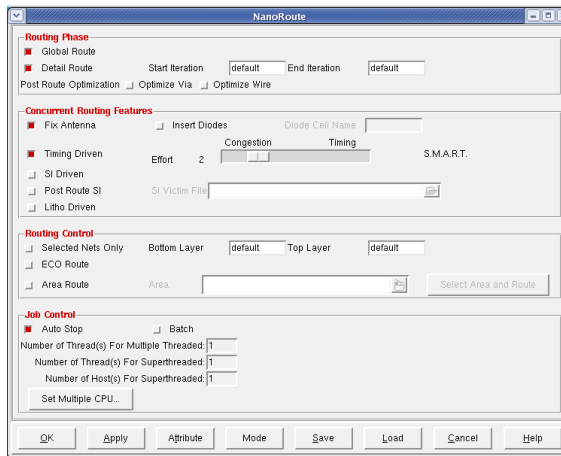


Figure C.15: NanoRoute.

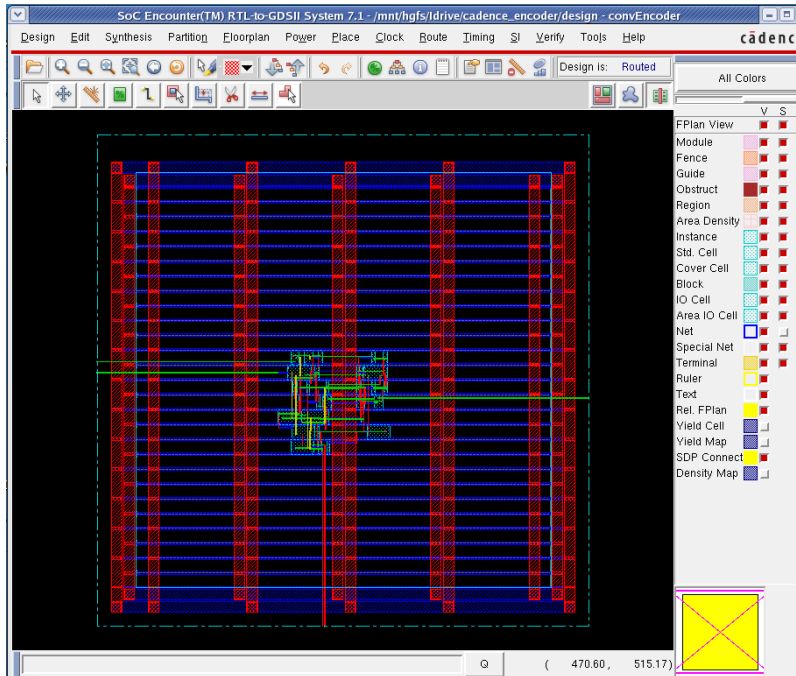


Figure C.16: After nanoRoute.

Step 8: Place: Filler cells are added to allow all the wells to be at the same potential. Place options window is as shown in Figure C.17. Figure C.18 shows design after placing the filler cells.

- Place
Place → Filler → Add filler
Click OK

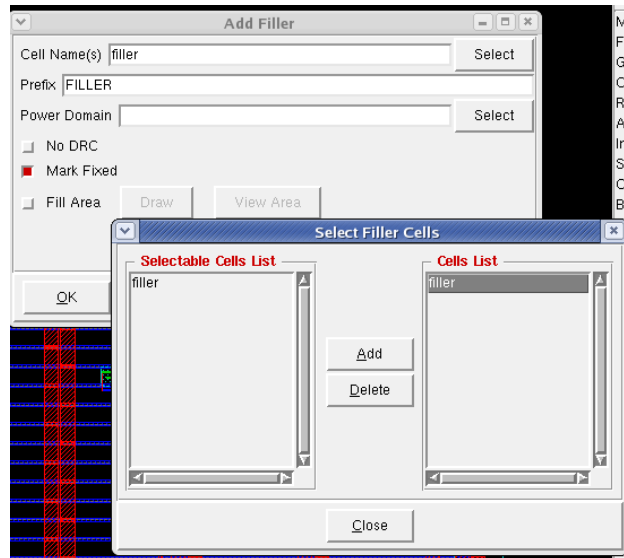


Figure C.17: Add filler.

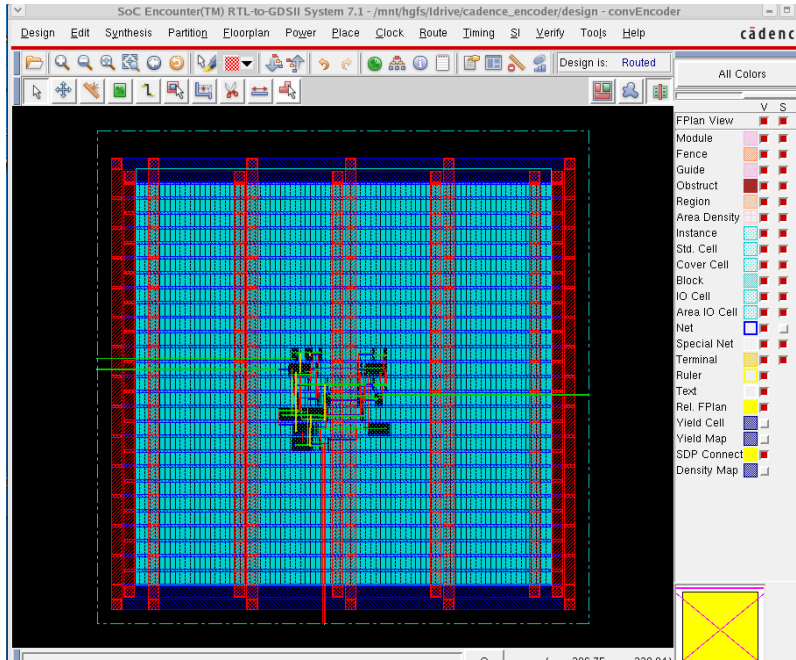


Figure C.18: After adding fillers.

Step 9: Verify: Final layout of the design is verified. Design connectivity and the geometry are verified by following commands. Design should pass selected tests. Connectivity and geometry options are as shown in Figure C.19 and C.20 respectively.

- Verify
 - Verify → Verify Connectivity
 - Click OK.

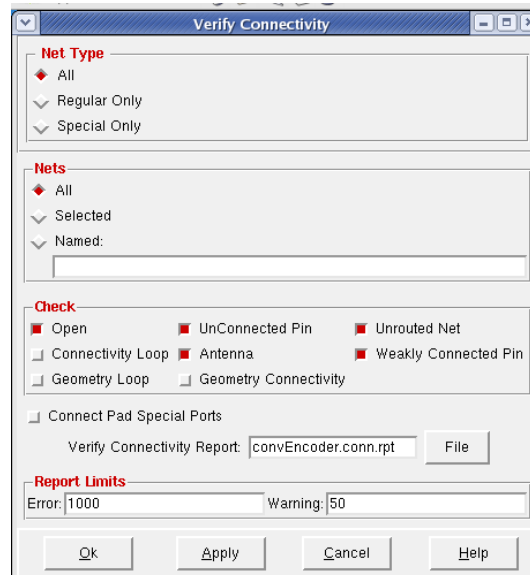


Figure C.19: Verify connectivity.

- Verify → Verify Geometry
Click OK.

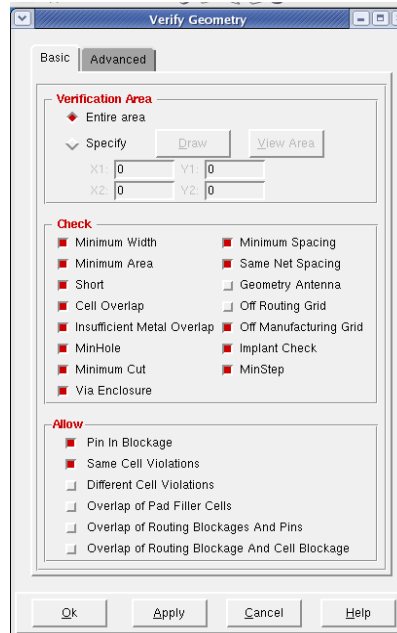


Figure C.20: Verify geometry.

Step 10: Export: The design is saved and its GDS file is exported. Figure C.21 shows the GDS export form.

- Export GDS

Design → Save → GDS

Output Stream File: convEncoder.gds

Map File: vtv_SocE2df2.map

Click OK.

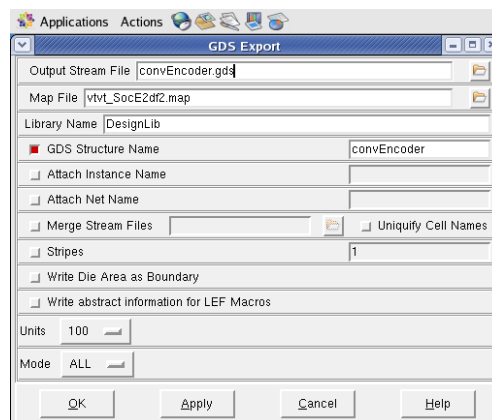


Figure C.21: GDS export form.

C.4 Verification of the Design

The layout of the design generated in Cadence Encounter is imported into Cadence icfb to verify if the Encounter has properly generated the design. Also to check if the generated design is DRC clean. Verification of the design is performed as follows:

The layout generated in the Encounter is imported into Cadence Virtuoso.

Step 1: Start Cadence icfb

Step 2: In the CIW → File → Import → Stream..

In the Stream In form fill the following as shown in Figure C.22.

- Run directory: .
- Input file: convEncoder.gds
- Library name: vtv_tsmc250

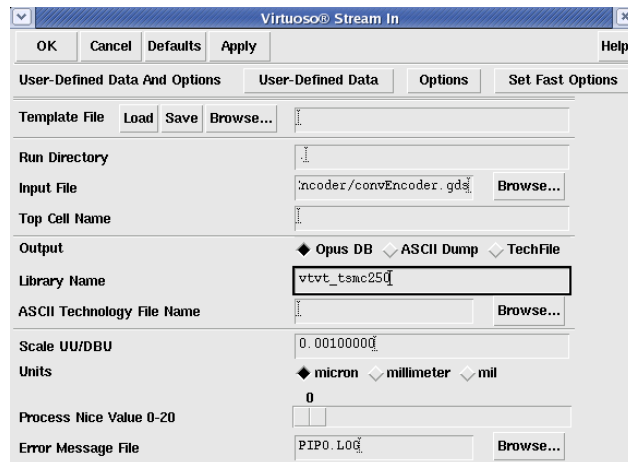


Figure C.22: Stream in form.

Step 3: Select User-Defined data:

Fill the details as shown in Figure C.23.

- Layer map table: vtv_tsmc250_StreamIn.map

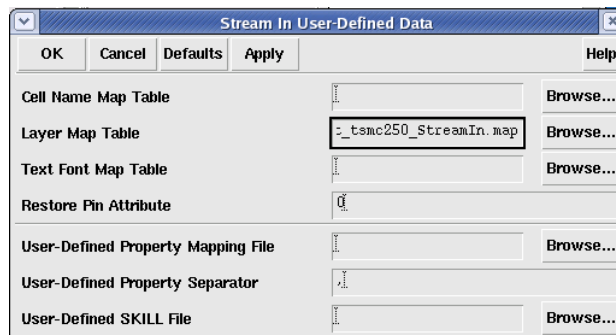


Figure C.23: User-defined data form.

Step 4: Select options:

- Fill as shown in Figure C.24. Click OK.

A pop-up message appears indicating that PIPO STRMIN completed successfully.

The 'Stream In Options' dialog box contains the following settings:

- Report Bad Polygons:
- Snap XY to Grid Resolution:
- Convert Array to Simple Mosaic:
- Skip Undefined Layer-Purpose Pair:
- Convert Zero Width Paths to: lines ignore
- Case Sensitivity: preserve upper lower
- Text Case Sensitivity: preserve upper lower
- Convert Nodes to: dots ignore
- Keep PCells:
- Replace [] with <>:
- Merge Undefined Purpose to drawing:
- Precision Report:
- Ignore BOX Record:
- Retain Reference Library (No Merge):
- Do Not Overwrite Existing Cell:
- Filter Out Warning/Information Messages:
- Filter Out Unmapping Warning:
- Hierarchy Depth Limit: 20
- Maximum Vertices in Path/Polygon: 1024
- Rod Directory: [empty]
- Reference Library Order: [empty]
- Keep Stream Cells:

Figure C.24: Options form.

Step 5: In the layout view, as shown in Figure C.25.

- Verify → DRC... and select OK. Design must be DRC clean.

The 'DRC' dialog box contains the following settings:

- Checking Method: flat hierarchical hier w/o optimization
- Checking Limit: full incremental by area
- Coordinate: [empty]
- Switch Names: [empty]
- Run-Specific Command File: [empty]
- Inclusion Limit: 1000 Limit Rule Errors: 0
- Join Nets With Same Name: Limit Run Errors: 0
- Echo Commands:
- Rules File: divaDRC.rul
- Rules Library: TechLib_tsmc03d
- Machine: local remote [empty]
- Use Error Database: [empty]

Figure C.25: DRC form.

The synthesized design in Cadence RTL is imported into a schematic in Cadence icfb.

Step 6: Start Cadence icfb

Step 7: In the CIW → File → Import → Verilog...

Fill in the form as shown in Figure C.26.

- Target library name: convEncoder_design
- Reference library: vtv_tsmc250_basic
- Verilog files to import: convEncoder.vh
- Import structural modules as: Schematic
- Power net name: VDD
- Ground net name: GND

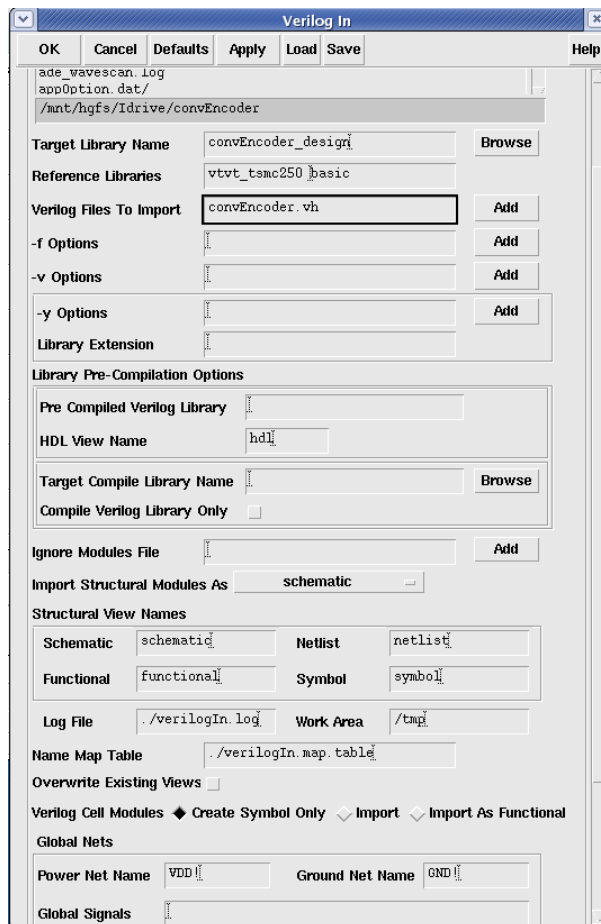


Figure C.26: Import Verilog in.

The synthesized design is shown in Figure C.27.

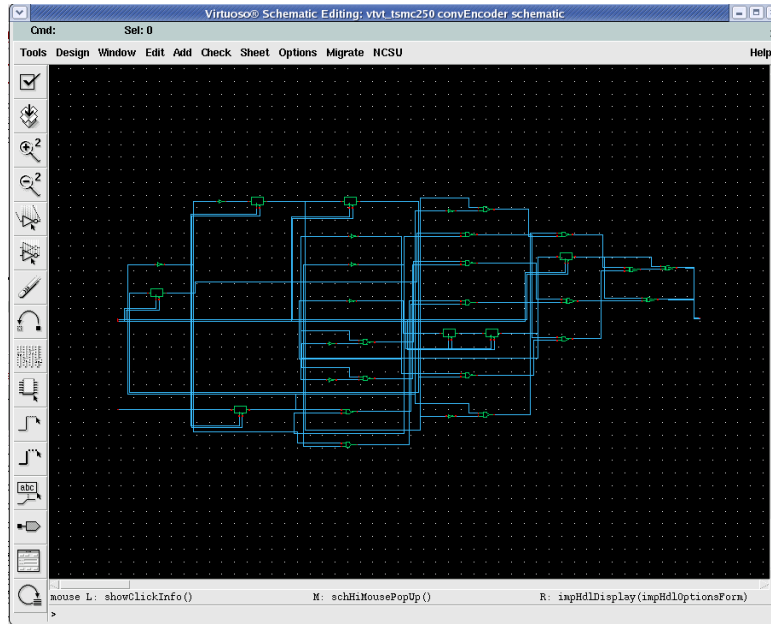


Figure C.27: Schematic view.

To check if schematic and layout have the same netlist, LVS is run on both schematic and layout.

Step 8: Open both the schematic and layout views

Step 9: Extract the layout using Verify → Extract → OK. The extracted view of the convolutional encoder is shown Figure C. 28.

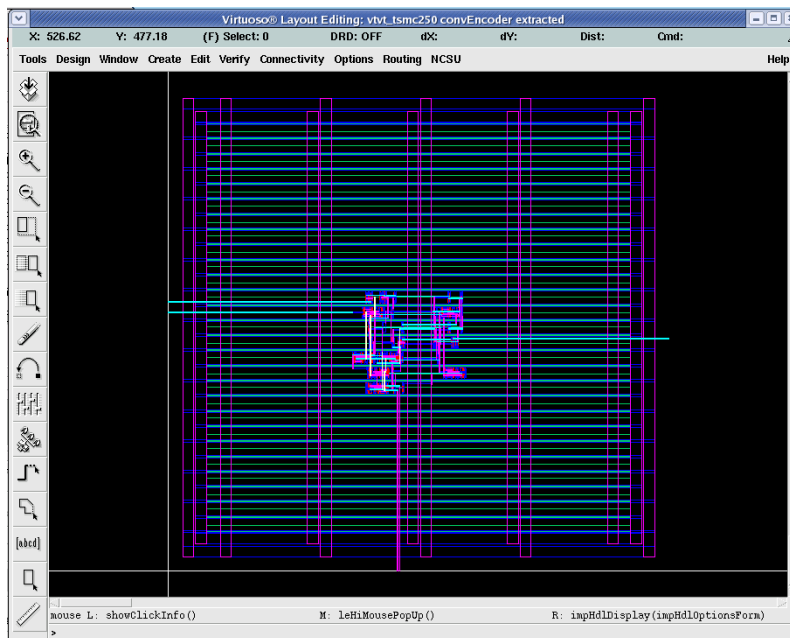


Figure C.28: Extracted view.

Step 10: Open the extracted view and perform LVS by choosing Verify → LVS as shown in Figure C.29.

A pop-up window appears notifying successful completion or failure of the LVS. In the LVS window click output to get the information regarding the LVS run.

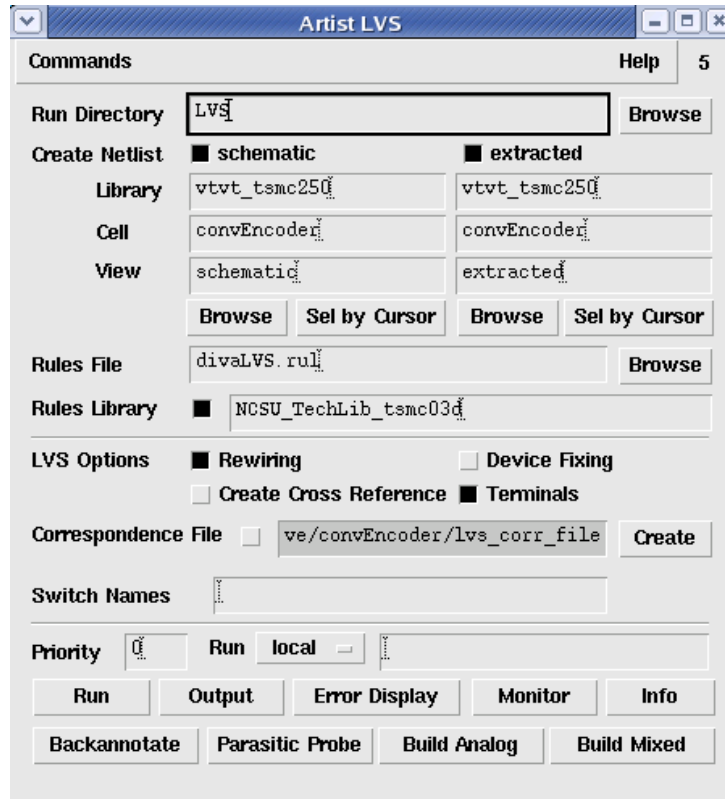


Figure C.29: LVS.

Appendix D - Quantization of Log-Likelihood Ratios in Decoder Implementation

In subsection 7.2.1.1 and 7.2.1.2, it is shown that 5-bit and 6-bit quantizations are required to represent $\varphi(z)$ and log-likelihood ratios, respectively, without compromising much on performance and latency. The actual value, binary equivalent and binary representation of z , $\varphi(z)$ and log-likelihood ratios are shown in Table D.1 and D.2 respectively.

Table D.1: Quantization of φ .

Actual Value		Binary Equivalent		Binary Representation	
z	$\varphi(z)$	z	$\varphi(z)$	z	$\varphi(z)$
0	3.875	0	31	00000	11111
0.125	2.750	1	22	00001	10110
0.250	2.000	2	16	00010	10000
0.375	1.625	3	13	00011	01101
0.500	1.375	4	11	00100	01011
0.625	1.125	5	10	00101	01010
0.750	1.000	6	8	00110	01000
0.875	0.875	7	7	00111	00111
1.000	0.750	8	6	01000	00110
1.125	0.625	9	5	01001	00101
1.250	0.500	10	4	01010	00100
1.375	0.500	11	4	01011	00100
1.500	0.375	12	3	01100	00011
1.625	0.375	13	3	01101	00011
1.750	0.250	14	2	01110	00010
1.875	0.250	15	2	01111	00010
2.000	0.250	16	2	10000	00010
2.125	0.125	17	1	10001	00001
2.250	0.125	18	1	10010	00001

2.375	0.125	19	1	10011	00001
2.500	0.125	20	1	10100	00001
2.625	0.125	21	1	10101	00001
2.750	0.125	22	1	10110	00001
2.875	0.000	23	0	10111	00000
3.000	0.000	24	0	11000	00000
3.125	0.000	25	0	11001	00000
3.250	0.000	26	0	11010	00000
3.375	0.000	27	0	11011	00000
3.500	0.000	28	0	11100	00000
3.625	0.000	29	0	11101	00000
3.750	0.000	30	0	11110	00000
3.875	0.000	31	0	11111	00000

Table D.2: Quantization of log-likelihood ratios.

Actual Value	Binary Equivalent	2's Complement Representation
0	0	000000
0.125	1	000001
0.250	2	000010
0.375	3	000011
0.500	4	000100
0.625	5	000101
0.750	6	000110
0.875	7	000111
1.000	8	001000
1.125	9	001001
1.250	10	001010
1.375	11	001011
1.500	12	001100

1.625	13	001101
1.750	14	001110
1.875	15	001111
2.000	16	010000
2.125	17	010001
2.250	18	010010
2.375	19	010011
2.500	20	010100
2.625	21	010101
2.750	22	010110
2.875	23	010111
3.000	24	011000
3.125	25	011001
3.250	26	011010
3.375	27	011011
3.500	28	011100
3.625	29	011101
3.750	30	011110
3.875	31	011111
-4.000	-32	100000
-3.875	-31	100001
-3.750	-30	100010
-3.625	-29	100011
-3.500	-28	100100
-3.375	-27	100101
-3.250	-26	100110
-3.125	-25	100111
-3.000	-24	101000
-2.875	-23	101001
-2.750	-22	101010

-2.625	-21	101011
-2.500	-20	101100
-2.375	-19	101101
-2.250	-18	101110
-2.125	-17	101111
-2.000	-16	110000
-1.875	-15	110001
-1.750	-14	110010
-1.625	-13	110011
-1.500	-12	110100
-1.375	-11	110101
-1.250	-10	110110
-1.125	-9	110111
-1.000	-8	111000
-0.875	-7	111001
-0.750	-6	111010
-0.625	-5	111011
-0.500	-4	111100
-0.375	-3	111101
-0.250	-2	111110
-0.125	-1	111111