DEVELOPMENT OF A NON-BINARY ERROR CONTROL DECODER FOR SOLID STATE DRIVES

BY

OMOWUYI OMONIYI OLAJIDE

DEPARTMENT OF COMPUTER ENGINEERING

FACULTY OF ENGINEERING

AHMADU BELLO UNIVERSITY, ZARIA

NIGERIA.

MARCH, 2021

DEVELOPMENT OF A NON-BINARY ERROR CONTROL DECODER FOR SOLID STATE DRIVES

BY

Omowuyi Omoniyi OLAJIDE B.Eng. (ABU Zaria, 2014)
P17EGCP8074
omowuyi@gmail.com

A DISSERTATAION SUBMITTED TO THE SCHOOL OF POSTGRADUATE STUDIES, AHMADU BELLO UNIVERSITY ZARIA IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF MASTERS DEGREE IN COMPUTER ENGINEERING.

DEPARTMENT OF COMPUTER ENGINEERING
FACULTY OF ENGINEERING
AHMADU BELLO UNIVERSITY, ZARIA
NIGERIA

MARCH, 2021

DECLARATION

I declare that this dissertation entitled "**Development of A Non-Binary Error Control Decoder for Solid State Drives**" has been carried out by me in the Department of Computer Engineering, Ahmadu Bello University, Zaria as part of the requirements for the award of the degree of Master of Science in Computer Engineering. The information derived from the literature has been duly acknowledged in the text and a list of references provided. No part of this dissertation was previously presented for another degree or diploma at this or any other institution.

Omowuyi Olajide	____________________	____________________
(Student)	Signature	Date

CERTIFICATION

This dissertation entitled DEVELOPMENT OF A NON-BINARY ERROR CONTROL DECODER FOR SOLID STATE DRIVES by Omowuyi OLAJIDE meets the regulations governing the award of the degree of Master of Science (M.Sc.) in Computer Engineering of the Ahmadu Bello University and is approved for its contribution to knowledge and literary presentation.

Dr M. B. Abdulrazaq	___________________	___________________
(Chairman, Supervisory Committee)	Signature	Date
Dr E. A. Adedokun	___________________	___________________
(Member, Supervisory Committee)	Signature	Date
Prof. M. B. Mu’azu	___________________	___________________
(Head of Department)	Signature	Date
Prof. Sani. A. Abdullahi	___________________	___________________
(Dean, School of Postgraduate Studies)	Signature	Date

ACKNOWLEDGEMENT

My immediate heartfelt gratitude goes to the Almighty God for the successful completion of this work. I could only imagine what life would have been for me without Him. Lord, I am forever grateful.

Foremost, I would like to express my sincere gratitude to my supervisors, Dr. M. B. Abdulrazaq and Dr. E. A. Adedokun for their patience, motivation, enthusiasm, constant drive, and vast knowledge. My sincere appreciation also goes to Dr. I. J. Umoh for his excellent contribution and counselling. Their immeasurable guidance helped me shape the research problem and constantly provided me with the insight towards the research and writing the dissertation report. I feel so privileged and honoured to have such a wonderful combination of supervisory committee. God bless you Sirs.

My sincere appreciation goes to the Head of the Department, Prof. M. B. Mu'azu for his contributions. Thank you so much sir for the constant reminder of the need to complete this research work on time.

My appreciation also goes to Dr. Emmanuel Okafor, Dr. A. T. Salawudeen, Dr. B. O. Sadiq, Dr. Y. Basira, Dr Y. Ibrahim and Engr. S.Y. Muhammed for all their contributions, motivation and assistance.

I am thankful to all the lecturers of Computer Engineering Department, Ahmadu Bello University, namely; Dr. T. H. Sikiru, Dr. Y. A. Sha'aban, Dr. H. Bello Salau, Dr. I. A. Bello, Mrs Z. M Abubakar, Mr. H. Zaharadeen, Mr. A. Umar, Mr. O. Ajayi and Mr. S. Y. Ibrahim for their kind advice and motivation. Also to Prof. F.O Anafi for your wonderful contribution.

Last but not the least, special thanks and deepest appreciation go to my parent and siblings for their endless love, unconditional support, advice, understanding and prayers. I am really grateful to everyone, for the support and love, May the Lord God bless you all.

Omowuyi Olajide

March, 2021.

DEDICATION

This dissertation is dedicated to the Almighty God, the Holy Spirit my inspiration, and my beloved family.

TABLE OF CONTENTS

DECLARATION	ii
CERTIFICATION	iii
ACKNOWLEDGEMENT	iv
DEDICATION	vi
LIST OF TABLES	x
LIST OF FIGURES	xi
LIST OF APPENDICES	xii
LIST OF ABBREVIATIONS	xiii
ABSTRACT	1

CHAPTER ONE

INTRODUCTION

1.1	Background to the study	2
1.2	Significance of Research	4
1.3	Statement of The Problem	5
1.4	Aim and Objectives	5

CHAPTER TWO

LITERATURE REVIEW

2.1	Introduction	6
2.2	Review of Fundamental Concepts	6
2.2.1	Solid State Drive (SSD) Architecture	6
2.2.2	Flash Memory Organization	7
2.2.3	Galois Field	7
2.2.4	Error-Correcting Codes Used in SSDs	7
2.2.5	Throughput	11
2.2.6	Power Consumption	12
2.2.7	ZYNQ 7000 FPGA	12
2.2.7.1	Processing System	14
2.2.7.2	Logic Fabric	14
2.2.7.3	Slice	15
2.2.7.4	Configurable Logic Block (CLB)	15

2.2.7.5 Lookup Table (LUT) 15
2.2.7.6 Flip-flop (FF) 16
2.2.7.7 Switch Matrix 16
2.2.7.8 Carry Logic 16
2.2.7.9 Input / Output Blocks (IOBs) 17
2.2.8 Zynq and Altera SoC Architecture Comparison 17
2.2.9 Performance validation 19
2.3 Review of Similar Works 20

CHAPTER THREE

MATERIALS AND METHODS

3.1 Introduction 26
3.2 Materials 26
3.2.1 Computer System 26
3.2.2 Vivado 26
3.3 Methods 27
3.3.1 Design of the NB-LDPC Code 28
3.3.1.1 Generation of the Parity Check Matrix (H) 28
3.3.1.2 Generation of the NB-LDPC Code in Verilog 32
3.3.2 Emulation of the error control code decoder architecture 33
3.3.2.1 Module Library Declaration with Parameters 33
3.3.2.2 Insertion of Interconnect Network and Routing 34
3.3.2.3 Declaration of Check Equation & Iteration Limit 35
3.3.3 Synthesis of the Architecture on the Zynq FPGA 36

CHAPTER FOUR

RESULTS AND DISCUSSION

4.1 Introduction 38
4.2 Synthesis 38
4.2.1 Register Transfer Level Analysis 39
4.2.2 RTL Synthesized Design 40
4.3 Synthesis Utilization 42
4.3.1 Utilization Report 42

4.3.2 Power Usage 42
4.3.3 Performance Comparison and Analysis 45

CHAPTER FIVE

CONCLUSION AND RECOMMENDATION

5.1 Summary 47
5.2 Conclusion 47
5.3 Limitations 47
5.4 Significant Contributions 48
5.5 Recommendations for further work 48
REFERENCES 49
APPENDIX 52

LIST OF TABLES

Table 4.1: Summary of Utilization Report of Resource Usage

Table 4.2: Relationship Between Throughput and Power Consumption

Table 4.3: Summary of Power Consumption of Logic Resource

LIST OF FIGURES

Figure 2.1 Block Diagram of SSD (Eshghi & Micheloni, 2018)

Figure 2.2: Tanner Graph of H Matrix (Chang *et al.*, 2016a)

Figure 2.3: Block Diagram of Zynq 7000 Development Board (Xilinx & Inc, 2019)

Figure 2.4: Architectural overview of Zynq 7000 Development Board (Xilinx & Inc, 2019)

Figure 2.5: Logic Fabric and Its Constituent Elements (Crockett *et al.*, 2014)

Figure 2.6: Composition of a Configurable Logic Block (Crockett *et al.*, 2014)

Figure 2.7. High-Level Comparison of Zynq and Altera SoC Architectures (Koelling *et al.*, 2015)

Figure 3.1: Vivado Design Suite 2018.2 Start Page

Figure 3.2: The NB-LDPC Code in Verilog

Figure 3.3: Verilog Description of the LDPC Module

Figure 3.4: Routing and Interconnection in the LDPC Decoder

Figure 3.5: Check Equations and Iteration Limit Declaration

Figure 3.6: Architecture of the Non-binary LDPC Decoder

Figure 4.1: Project Manager Depicting Completed Synthesis of the Design

Figure 4.2: Elaborated View of the LDPC Logic

Figure 4.3: Synthesized Design Window of the LDPC Decoder

LIST OF APPENDICES

Appendix A Vivado Synthesized Designs

Appendix B Synthesis and Implementation Report

Appendix C Tile Properties

LIST OF ABBREVIATIONS

Acronym	Definition
ARM	Advanced Reduced Instruction Set Computing Machine
ATPG	Automated Test Pattern Generation
BCH	Bose Chaudhuri, Hocquenghem
BEL	Basic Element of Logic
BSR	Boundary Scan Register
BUF	General-Purpose Buffer
BUFGMUX	Global Clock MUX Buffer with Output State 0
CAD	Computer Aided Design
CN	Check Node
CPLD	Complex Programmable Logic Device
CSP	Chip Scale Package
CPM	Circular Permutation Matrix
CPU	Central Processing Unit
COB	Chip on Board
DDR-SDRAM	Double Data Rate Synchronous Dynamic Random Access Memory
DIL	Dual in-Line
DLL	Dynamic Link Library

DRAM	Dynamic Random Access Memory
DUT	Device Under Test
EDA	Electronic Design Automation
ECC	Error Correction Code
FDCE	D Flip-Flop with Clock Enable and Asynchronous Clear
FDPE	D Flip-Flop with Clock Enable and Asynchronous Preset
FDRE	D Flip-Flop with Clock Enable and Synchronous Reset
FF	Flip-flop
FIFO	First In First Out
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IBUFG	Dedicated Input Buffer with Selectable I/O Interface
IC	Integrated Circuit
ICT	In-Circuit Test
IDE	Integrated Development Environment
I/O	Input/Output
IP	Intellectual Property
ISP	In-System Programming

JTAG	Joint Test Action Group
LDPC	Low Density Parity Check
LLR	Log Likelihood Ratio
LUT	Look Up Table
MCM	Multi Chip Module
MCU	Microcontroller Unit
MUX	Multiplexer
MUXCY	Carry Logic Multiplexer
NB-LDPC	Non Binary Low Density Parity Check
NVM	Non-Volatile Memory
ODM	Original Design Manufacturer
OEM	Original Equipment Manufacturer
OE	Output Enable
PCB	Printed Circuit Board
PCI Express	Peripheral Connect Interface Express
PGA	Pin Grid Array
PIO	Programmable Input/Output
PL	Programmable Logic

PLD	Programmable Logic Device
PLL	Phase Locked Loop
PS	Processing System
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Level
SoC	System on Chip
SRAM	Static Random Access Memory
SRL	Serial Shift Register
SSD	Solid State Drive
SSRAM	Synchronous Static Random Access Memory
UART	Universal Asynchronous Receiver/Transceiver
USB	Universal Serial Bus
UUT	Unit Under Test
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
VN	Variable Node
XORCY	carry-XORs

XST	Xilinx Synthesis technology
ZM	Zero Matrix

ABSTRACT

This dissertation presents the development of a non-binary error control decoder for solid state drives that require high throughput when reading data for error correction. The miniaturization of chip fabrication has made flash memory cells of Solid State Drives (SSDs) susceptible to distortion and error. This is as a result of the continuous storage of bits unto a single cell, which eventually leads to an increase in the number of errors to be corrected by the decoder. Also, the representation of the messages passed between the variable node and the check node involve the use of large Galois fields, which eventually results in very high decoding complexity without leading to an increase in the decoding throughput. Bose-Chaudhuri-Hocquenghem (BCH) code previously utilized to correct multi-bit errors, causes the SSD controller to experience latency during decoding. In this work, a non-binary Low Density Parity Check (LDPC) code is used in conjunction with a small Galois Field (GF) of eight, a parallel architecture and a reduced iteration limit to develop an error control decoder for SSDs. The error control decoder was synthesized on a ZYNQ 7000 Series Field Programmable Gate Array (FPGA). The developed error control decoder achieved a throughput of 2.34Gbps at 125-MHz clock frequency and a maximum iteration limit of six (6). A total power of 0.223W was consumed by the decoder. The result shows an improvement in the throughput by 7.3%, and an increase in the power by 5.2% when compared with the decoder implemented by Toriyama *et al* 2018.

CHAPTER ONE

INTRODUCTION

1.1 Background to the study

Systems that perform on-line transaction processing such as cloud computing and virtualization, which implement Solid State Drives (SSDs) require very fast random access to data (Smith, 2020). The probability of the corruption of data occurring in the memory of semiconductor devices keeps increasing with the scaling of the technology unit (Zhao *et al.*, 2013). Storage semiconductor facilities being used include Random Access Memory (RAM), Read Only Memory (ROM) and flash memory (Eshghi & Micheloni, 2018). The disadvantages in error detection techniques make them unsuitable for use in semiconductor memories (Micheloni *et al.*, 2018). They include; cell deterioration, large area overhead, low throughput and very complex architectures (Zhao *et al.*, 2013). The drawbacks in the error detection techniques necessitate the development of a more reliable solution is sought that would provide better error resiliency. Such technique often involves the use of better correction codes that reduce memory cell deterioration and give high throughput. This technique is the Low Density Parity Check (LDPC) soft decoding scheme, which is also being used in communication systems. The errors that are prevalent in data memory can be single bit errors or multi-bit errors. Errors with single bit in data memory are rectified by using single bit error correction like Hamming code. Multi-bit errors are corrected using multi-bit error correction, like Bose-Chaudhuri-Hocquenghem (BCH) code (Reviriego *et al.*, 2012).

The advent of SSDs has revolutionized the memory industry. NAND Flash memories which make up SSDs have changed the way data memory is implemented. They are now used in many digital systems which include smartphones, tablets and cameras. SSDs have now become the

preferred application for Cloud computing, enterprise servers and ultra-modern laptops (Chang *et al.*, 2016a). Even in NAND memory, error correction is of utmost importance. This is because, the tendency for error to occur increases as more bits are stored in smaller cells. This necessitates the use of Error Correction Codes (ECCs). All ECCs, including LDPC codes, have a probability of failing at a given bit error rate. BCH codes can correct single bit errors effectively, but are not reliable when correcting multi-bit errors. As a result, BCH codes are still used when throughput and bandwidth are not critical requirements. Various literatures have not only established the excellent capabilities of LDPC codes, but also its drawbacks. (Nicola *et al.*, 2018). LDPC codes results in complex decoding algorithms and require large logic resource for implementation. Nonetheless, LDPC codes have become the preferred choice in the enterprise domain (Lee *et al.*, 2012). This is as a result of the excellent error correction capabilities and the ability of being able to handle both hard decoding (i.e. zero and one) and soft decoding (i.e. involving probability).

1.2 Significance of Research

NAND flash memory is a non-volatile, solid state storage medium that affords numerous powerful advantages over rotating magnetic storage such as hard disks: increased performance; higher density; higher reliability; and higher throughput. These advantages make flash memory ideal for use in portable devices, as well as in high-performance SSDs and server-side caching systems (Smith, 2020). NAND flash memory also have an undesirable feature-the memory cells deteriorate slightly with each program/erase (P/E) cycle. As each individual cell deteriorates, its ability to correctly hold a given bit state reduces, causing its read error rate to increase. At some point, the errors can no longer be corrected, damaging the cell (Smith, 2020). To date, error control codes like BCH code have worked well in solid state drives. But that is now changing as chip fabrication geometries shrink, and as densities increase from single- and multi- to three-level cells storing one, two or three bits, respectively (Lee *et al.*, 2012). Storing more bits in smaller cells makes it possible to fit more storage into smaller form factors, but the smaller/denser cells hold smaller charge and cause a spike in the raw bit error rate of data stored in the cells. NAND flash memory provides a fixed amount of storage for ECC. Given the fixed storage, BCH codes are only able to meet output bit error rate requirements with up to a certain value at considerable speed, and when the cells deteriorate beyond that point, they fail to work (Smith, 2020). LDPC error-correction technology, particularly through judicious use of soft-decision LDPC decoding, is able to meet output error rate requirements, and as a result can greatly extend the usable P/E cycles of NAND flash memory (Eshghi & Micheloni, 2018).

1.3 Statement of The Problem

Reduction in the shrinking of the chip technology unit makes cells of memory blocks subject to growing severe distortion, thereby causing bit errors that mainly weaken the storage performance and the reliability of the flash memory (Zhao *et al.*, 2013). Conventional BCH codes, which are currently implemented in all the commercial SSDs today, have become insufficient to correct these bit errors (Micheloni *et al.*, 2018). The speed of the SSD is limited by how fast the error correction code can decode the information and present it to the user (Smith, 2020). Therefore, there is need for an error control decoder that will improve the speed of the SSDs by increasing the write/read throughput through high parallelization capability and ease of decoding. LDPC codes employ full parallelization and Galois field representation over multiple read operations to determine the likelihood of each cell containing a bit value 1 or 0 at high speeds, thereby providing stronger protection, but at the cost of greater decoding latency and storage overhead (Micheloni *et al.*, 2018)

1.4 Aim and Objectives

The aim of this research work is to develop a non-binary error control decoder for solid state drives.

In order to achieve this aim, the following objectives are employed:

1. To develop a non-binary low density parity check code.
2. To emulate the NB-LDPC error control code decoder architecture.
3. To synthesize and compare the performance of the developed system with the work of (Toriyama & Markovic, 2018) using throughput and power consumption as performance metrics.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

The literature review consists of the review of fundamental concepts that are related to this work and the review of similar works.

2.2 Review of Fundamental Concepts

Fundamental concepts important to the research work are discussed as follows.

2.2.1 Solid State Drive (SSD) Architecture

Flash-memory-based SSDs are able to offer very much faster random access to data and high transfer speeds. SSDs are logic architectures where every part is soldered on a printed circuit board and is uniquely packaged. The SSD comprises of a bank of flash memories (or chips) and a controller, as illustrated in Figure. 2.1

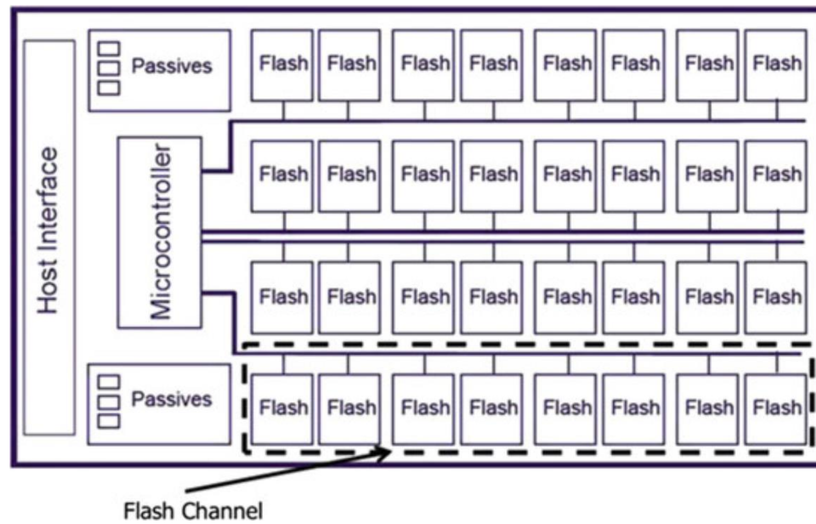


Figure 2.1: Block Diagram of SSD (Eshghi & Micheloni, 2018).

2.2.2 Flash Memory Organization

The flash memory is distributed across many flash chips, where they each possess from one to multiple dies. Dies are distinct sets of silicon wafer that are joined to the chip pins. It is illustrated in Figure 2.1. SSDs usually possess 5–18 chips per die, and can contain up to a maximum of 16 dies on each chip. Each one is joined to a single or multiple channels of the physical memory, and these channels are not duplicated within the chips (Chang *et al.,* 2016b).

2.2.3 Galois Field

A Galois field GF(q) is also called a finite field. This is a field that contains a finite number of elements. As with any field, a finite field is a set on which the operations of multiplication, addition, subtraction and division are defined and satisfy certain basic rules. The symbol q stands for q-ary and it represents the orders of the Galois field. In the construction of binary and non-binary QC LDPC codes, a nonzero element in GF(q) is represented by a non-binary Circular Permutation Matrix (CPM), while the zero element is represented by a zero matrix (ZM). CPM/ZM is a square matrix over GF(q)/zero where if every row of the matrix is the cyclic-shift of the row above it one place to the right, and the top row is the cyclic-shift of the last row one place to the right.

The number of edges that are incident with a variable node (or check node) in the Tanner graph of an LDPC code is called the variable node degree (or check node degree). The girth of an LDPC code is the length of the shortest cycle in its Tanner graph. Cycles, especially short cycles, leave a bad effect on the performance of LDPC decoders.

2.2.4 Error-Correcting Codes Used in SSDs

Conventional SSDs usually make use of one error correction code, which is the Bose–Chaudhuri–Hocquenghem (BCH) code. This is because, the BCH code permit multiple bit error

correction (Lee *et al.*, 2012), and single read of flash memory that generate error (Lee *et al.*, 2012). Low-density parity-check (LDPC) codes on the other hand use accumulated information over many read operations to ascertain the probability of each cell having a bit value one (1) or a zero (0) (Kumar, 2004), thus enabling a better and reliable protection, though at the expense of increased decoding latency and excessive area (Wang *et al.*, 2014).

2.2.4.1 Bose–Chaudhuri–Hocquenghem (BCH) Codes

Bose–Chaudhuri–Hocquenghem (BCH) codes (Lee *et al.*, 2012) have been adopted in SSDs during the past few years, as a result of their ability to detect and correct multi-bit errors, and at the same time ensuring that latency and hardware cost is reduced (Lee *et al.*, 2012). They are designed to ensure correction to a certain degree, of bit errors within each codeword. A stronger error correction strength demands more check bits or an increased codeword length.

2.2.4.2 Low-Density Parity-Check (LDPC) codes

Low-Density Parity-Check (LDPC) code (Kumar, 2004) is the correction code that is currently being used in state of the art SSDs. This is because they guarantee a better capability for error correction than BCH codes, but at an increased cost of storage (Wang *et al.*, 2014). An excellent LDPC code ensures that the rate of failure (that is, the percentage of reads where the code fail to correct the data) is lower compared to the expected rate for a given number of errors. Also, when SSDs are manufactured with error correction, the LDPC code used is made to be systematic, i.e., to contain the message (data) in correlation to the code-word.

An LDPC code is represented by the parity check matrix (H) using the tanner graph, where a part of the graph contains nodes that depict the bit in the code-word, and the other part of the graph has nodes that depict the equations of the parity check, that produce every parity bit (Kumar, 2004). When there are errors in a code-word gotten from memory, an LDPC decoder applies

belief propagation to decode iteratively the bits in the code-word with the highest probability to have a bit error (Cai *et al.*, 2017).

$$H = \begin{array}{c} \\ \left[\begin{array}{ccccccc} C_0 & \mathbf{C_1} & \mathbf{C_2} & C_3 & \mathbf{C_4} & \mathbf{C_5} & C_6 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] f_1 \end{array} \quad (2.1)$$

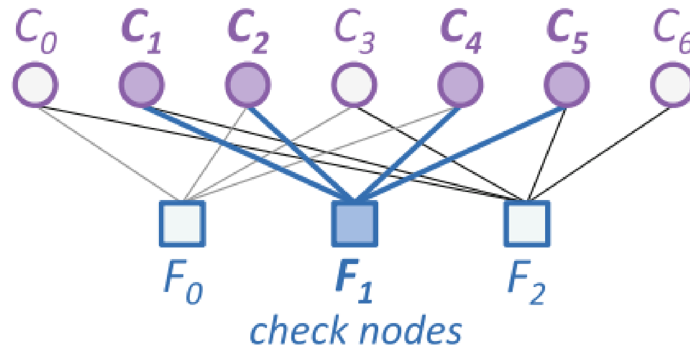


Figure 2.2: Tanner Graph of H Matrix (Chang *et al.*, 2016a)

An LDPC code is depicted using a matrix called parity check matrix (H) as shown in equation (2.1), where H is very sparse (i.e., the amount of (1) present in the matrix are few). Figure 2.2 illustrates such a matrix for a code-word c, with seven-bit. For an n-bit code-word that encodes a k-bit data message, H is sized to be an $(n - k) \times n$ matrix. In the matrix, every row depicts an equation of parity check, while every column depicts one of bits in the code-word. This matrix has three rows, as a result the correction of error implements three equations of parity check (represented as f). A non-zero value in H_{ij} shows that equation of parity check f_i has bit b_j. Every equation of parity check XORs all the code-word bits in the equation to determine if the output is zero. For illustration, equation of parity check f_1 from the matrix in Figure 2.2 is given in equation (2.2) as:

$$f_1 = c_1 \oplus c_2 \oplus c_3 \oplus c_4 = 0 \quad (2.2)$$

Therefore, c is a valid code-word only if

$$H.c^T = 0 \quad (2.3)$$

where c^T is the transpose of the code-word c.

For the purpose of belief propagation, H can be depicted using a tanner graph (Zhang *et al.*, 2011). A Tanner graph comprises check units, which depict the equations of parity check, and bit units, which depict the bits in the code-word. An edge joins a check unit Fi to a bit unit Cj only if equation of parity check fi has bit cj. Figure 2.2 depicts the graph that coincides with the H matrix in equation (2.1).

One of the function of the controller of the SSD is to read requests. As it does this, it extracts the *k*-bit message data from the code-word *r* that is kept in the flash memory. The LDPC decoder in an SSD has a primary function of decoding (Dolecek, 2014), by which it can correct the code-word *r* gotten, so as to get the original code-word *c* and extract the message data.

The decoding process involves five workflows. Two information set pieces are parameters to ascertain the likelihood of bit error: the likelihood that each bit in the code-word is a one (1) or zero (0), and (2) the equations of parity check. The decoder calculates an initial log likelihood ratio (LLR) for every bit of the kept code-word. A message LLR comprises of values of LLR for every bit, which are changed and transferred among the check units and bit units during each step. The iterative belief propagation updates the message LLR, to locate those bits that are most likely to have error.

A number of decoding algorithms exist that perform belief propagation (BP) for LDPC codes. The most prevalent is the min-sum (MS) algorithm (Chen & Wang, 2012), a simpler implementation of the original BP algorithm for low density parity check (Chang *et al.*, 2016a)

with high error correction capability. When the iterations of MS algorithm are performed, the decoder tracks a set of code-word bits with high likelihood of error and will then flip such bit.

The complexity of non-binary LDPC decoders is measured by the degree of multiplication of the Galois field used, the size of bits representing the messages passed between the variable and the check nodes, and the degree of parallelism (J. Lacruz *et al.*, 2016)

2.2.5 Throughput

Throughput is defined as the average number of input bits processed per second. The circuit usually processed is a sequential circuit. The fastest SSD today, (Samsung 860 EVO) has a sequential read of 550 (Mega Bytes per second) MBps (4.4Gbps) and a sequential write of 520MBps (4.16Gbps).

The throughput of a decoder is usually calculated using equation (2.4) (O. Lacruz *et al.*, 2015)

$$T = \frac{f_{clk} \times N \times p}{I_{max} \times (M + d_v \times D) + (q - 1)} Mbps \qquad (2.4)$$

where

$T = Throughput$

$f_{clk} = maximum\ clock\ frequency$

$I_{max} = maximum\ number\ of\ iterations$

$N = Total\ number\ of\ variable\ node\ units$

$M = Total\ number\ of\ check\ node\ units$

$p = parallelism\ factor$

$d_v = variable\ node\ degree$

$D = Pipeline\ stages\ used\ in\ the\ design$

$q = Galois\ field$

2.2.6 Power Consumption

The power consumed by the decoder architecture is divided into dynamic power and static power. Static power is power consumed when circuit activity is not taking place. For example, the power expended by a D flip-flop when neither the D input nor the clock have active inputs (i.e., all inputs are "static" because they are at fixed dc levels). Dynamic power is the power consumed when the inputs are active

Dynamic Power,

$$P_D = \beta * C * V_{DD}^2 * f_{clk} \tag{2.5}$$

Where, f_{clk} is the system frequency (125MHz), β is the activity factor (β = 1, because all nodes are switching at the same rate as the frequency), V_{DD} is the source voltage (1.2V) and C is the capacitance ($1.18 * 10^{-9}$F).

2.2.7 ZYNQ 7000 FPGA

The Zynq 7000 family is based on the Xilinx SoC architecture. These products incorporate a feature-rich or single-core or dual-core ARM Cortex based processing system and 28 nm Xilinx programmable logic (PL) in a single device. The ARM Cortex CPUs are the main building blocks of the processing system and also include on-chip memory, external memory interfaces, and peripheral connectivity interfaces.

The ZC702 evaluation board for the XC7Z020 SoC provides a hardware environment for developing and evaluating designs targeting the Zynq® XC7Z020-1CLG484C device. The ZC702 board provides features common to many embedded processing systems, including DDR3 component memory, a tri-mode Ethernet PHY, general purpose I/O, and two UART interfaces. Figure 2.3 shows the ZC702 board with all its labelled parts.

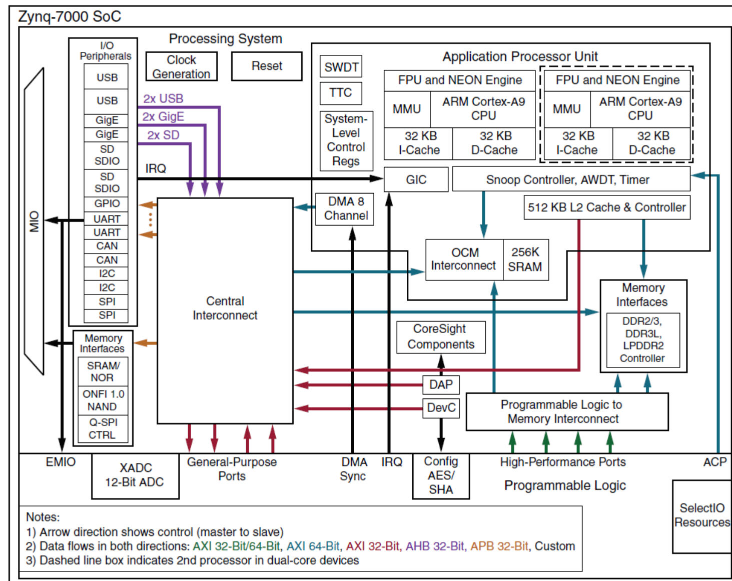


Figure 2.3: Block Diagram of Zynq 7000 Development Board (Xilinx & Inc, 2019)

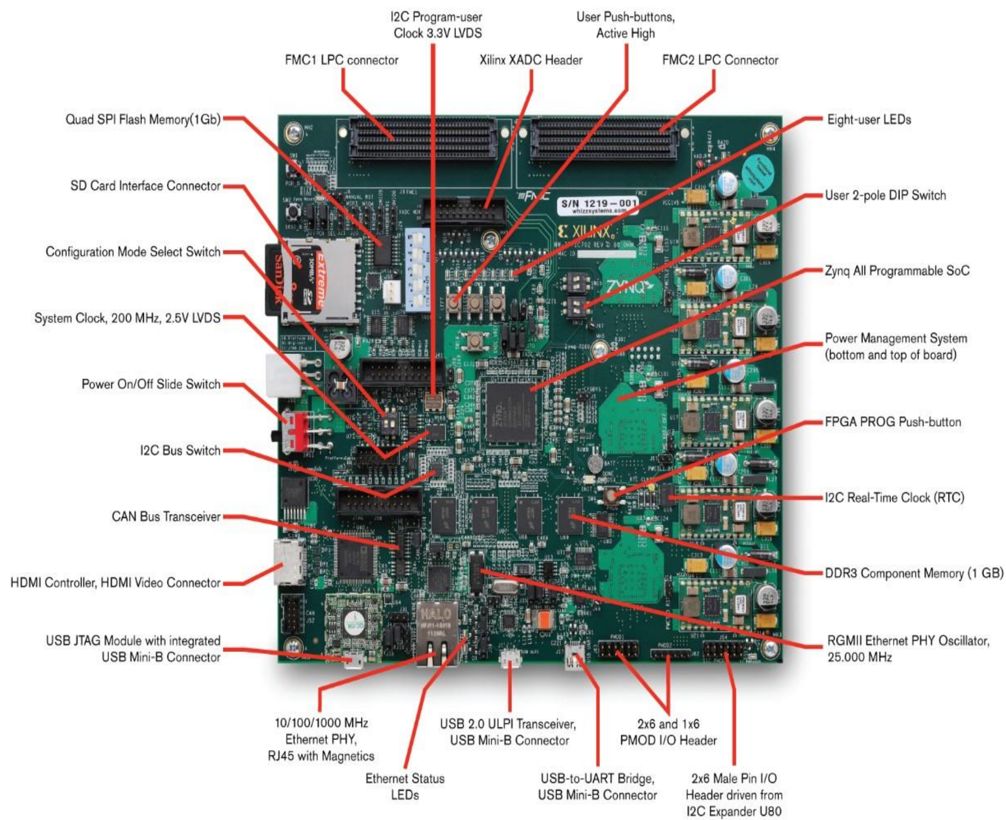


Figure 2.4: Architectural overview of Zynq 7000 Development Board (Xilinx & Inc, 2019)

SRAM-based FPGAs, Flash-based FPGAs, and Fuse and Anti-fuse-based FPGAs (one-time programmable) are examples of different types of FPGA. The complete decoder architecture is synthesized into the following unit blocks:

2.2.7.1 Processing System

All Zynq devices have the same basic architecture, and all of them contain, as the basis of the processing system, a dual-core ARM Cortex-A9 processor and a second principal part of the called the programmable logic (PL).

2.2.7.2 Logic Fabric

The PL part of the Zynq device is depicted in Figure 2.4, with various features highlighted. The PL is predominantly composed of general purpose FPGA logic fabric, which is composed of slices and Configurable Logic Blocks (CLBs), and there are also Input/ Output Blocks (IOBs) for interfacing.

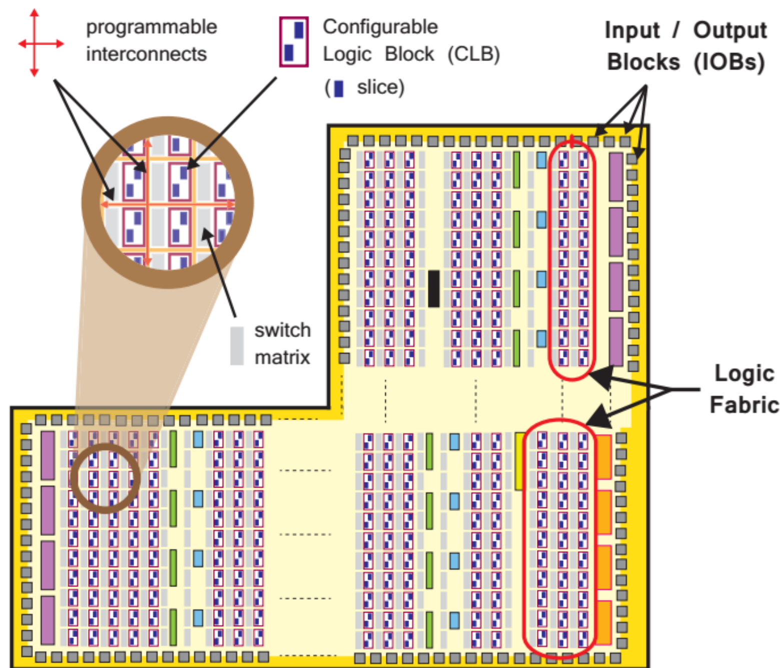


Figure 2.5: Logic Fabric and Its Constituent Elements (Crockett *et al.*, 2014)

2.2.7.3 Slice

A sub-unit within the CLB, which contains resources for implementing combinatorial and sequential logic circuits. As indicated in Figure 2.4, Zynq slices are used to compute 4 Lookup tables and 8 JK Flip-Flops.

2.2.7.4 Configurable Logic Block (CLB)

CLBs are small, regular blocks of logic elements that are laid out in a two-dimensional array on the PL, and connected to other similar resources via programmable interconnects. Each CLB programmed to be positioned next to a switch matrix and contains two logic slices, as shown in Figure 2.4.

2.2.7.5 Lookup Table (LUT)

A LUT, which stands for LookUp Table is basically a table that defines what the output is for any given input(s). In the context of combinational logic, it is the truth table. This truth table effectively defines how the combinatorial logic behaves. This flexible logic resource was used to implement

(i) A logic function of up to six inputs;

(ii) A small ROM;

(iii) A small RAM;

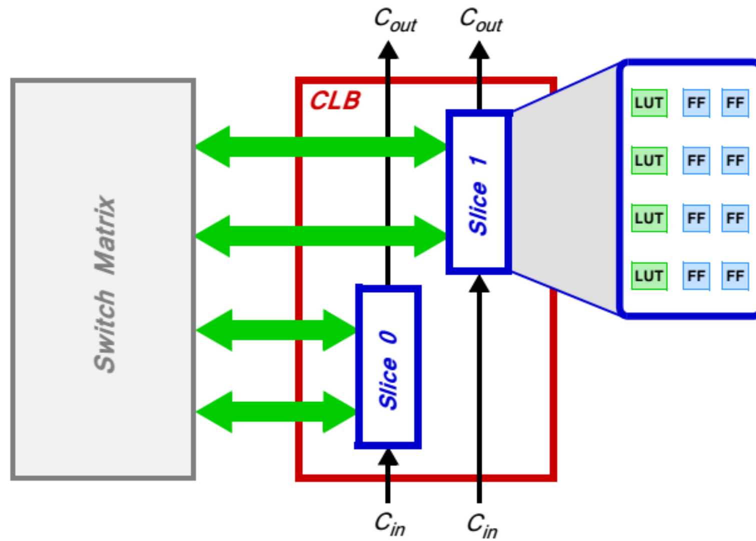


Figure 2.6: Composition of a Configurable Logic Block (Crockett *et al.*, 2014)

2.2.7.6 Flip-flop (FF)

This sequential circuit element implemented a 1-bit register, with reset functionality. One of the FFs is used to implement a latch.

2.2.7.7 Switch Matrix

The switch matrix sits beside each CLB. It is used to implement and provide a flexible routing facility for making connections between elements within a CLB; and from a CLB to other logic resources on the PL.

2.2.7.8 Carry Logic

Arithmetic circuits need intermediate signals to be transmitted between adjacent slices This was realized via the carry logic. The carry logic was used to constitute a chain of routes and multiplexers to connect slices in a vertical column.

2.2.7.9 Input / Output Blocks (IOBs)

IOBs are logic resources that provide interfacing between the PL logic blocks, and the physical device block used to connect to the external circuitry. Each IOB is used to handle a 1-bit input or output signal. The IOBs are situated around the perimeter of the device.

The Xilinx tools automatically inferred the required logic fabrics LUTs, FFs, IOBs, CLBs, switch matrix, RAM, etc., from the decoder design, and mapped them accordingly.

2.2.8 Zynq and Altera SoC Architecture Comparison

Altera® SoC FPGA and Xilinx Zynq All Programmable SoC integrate a dual-core ARM® Cortex®-A9 MP-Core™ processor and FPGA logic into a single programmable device. Since their introduction, both devices generated significant design activity within the embedded and FPGA development groups.

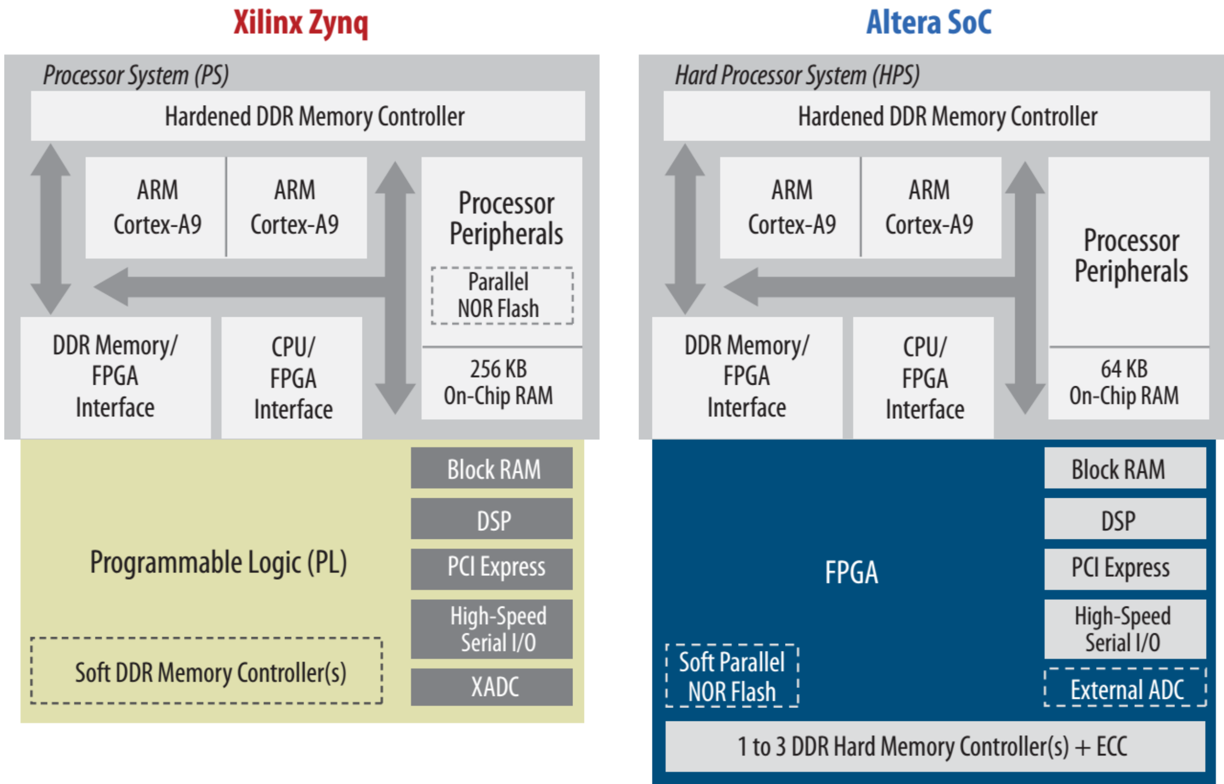


Figure 2.7. High-Level Comparison of Zynq and Altera SoC Architectures (Koelling *et al.*, 2015)

From a high level, the similarities between the Xilinx Zynq family and the Altera SoC family are readily apparent, as shown in Figure 2.6. Both device families integrate a high-performance, 32 bit dual-core ARM Cortex-A9 MP-Core processor along with their coupled peripherals, all linked to a modern FPGA architecture with integrated hard intellectual property (IP) blocks (digital signal processing (DSP) blocks, RAM blocks, PCI Express® (PCIe®) blocks, high-speed serial transceivers, and so on). In both device families, a hardened DDR memory controller primarily serves the ARM processors and optionally serves functions in the programmable logic (PL) or FPGA. Altera SoC also provide additional DDR hard memory controllers dedicated to FPGA-based functions. In Zynq, these DDR memory interfaces use PL resources.

The similarities between these two devices indicates that equivalent logic architectures can be implemented on both negligible disparities in results.

2.2.9 Performance validation

The main performance metric used for the validation of the error control decoder is the throughput power consumption. The throughput is calculated in bits per second (bps). The implementation of algorithms using HDL in synthesis tools, requires that the throughput is not obtained directly. It has to be found out manually. It is calculated using equation (2.4)

2.3 Review of Similar Works

The following literatures were reviewed and they give a proper perspective of the field domain.

Yang *et al.,* (2008) presented the implementation of the Min-Max (MM) algorithm, in the log likelihood ratio (LLR), as compared to the Min-Sum decoding algorithm. This was done through iterative decoding of non-binary LDPC codes over a Galois characteristic field of sixteen (16). All the symbols of the Galois field were used in the implementation. A Monte Carlo simulation was performed on both algorithms and the result revealed that the Min-Sum Algorithm required more number of iterations to converge as compared to the Min-Max Algorithm. This result showed that the complexity of the check unit architecture increased dramatically as the Galois field. This work experienced a very high decoding complexity in the non-binary decoder scheme which makes the Min-Max (MM) decoding scheme offer a large memory making it difficult for practical demonstration.

Lin *et al.,* (2010) proposed an efficient selective architecture of the Min–Max (MM) decoding algorithm. This algorithm had the property that it completely avoids the sorting process. Also, the paper presented an efficient very large scale integrated (VLSI) architecture for a non-binary Min–Max decoder. The non-binary decoder was synthesized and the results showed that the technique that was presented was efficient to a certain degree. In addition, the architecture of a check-node unit (CNU) implemented in the Min–Max decoder still remained the same without any modification. The check node unit of the Min–Max decoding algorithm had a very high decoding complexity and as a result could not be implemented practically with high parallelism.

Chen & Wang., (2012) proposed a simple implementation of the min-sum (MS) algorithm with minimal loss. This was done by introducing a set of hard messages exchanged between bit nodes and check nodes into the architecture scheme. These messages are different from the already

present variable and check node messages in the decoder architecture called soft messages. The hard messages introduced were used as the indication in selecting the most reliable messages. These messages were arranged such that their absolute values can be determined quickly without complex processing in the check unit. The architecture of the decoder proposed was implemented with a (620,310) non-binary code with a Galois field characteristic of 32, $GF\ (2^5)$ in a TSMC CMOS technology. The result of the implementation showed a throughput of 64 Megabits per second (Mbps) with fifteen (15) iterations. The proposed decoder achieved a very low throughput. The use of the Min-Sum Algorithm resulted in the increase in the already high decoding complexity.

Codes *et al.,* (2013) proposed a novel relaxed processing of the check node architecture for the min-max (MM) non-binary decoding algorithm. This was done by making each of the element of the Galois finite field of $GF(2^p)$ to uniquely represent a combination of p individual field elements. During implementation, the technique developed, first found a set of the p messages that are most reliable which are sent from the variable (bit) unit to the check node with individual elements, called the Minimum-Basis (MB). These messages are then derived from the Minimum-Basis and sent. This technique aimed at lowering the very high complexity of the check unit architecture. The proposed system experienced some loss in performance, but the complexity of the check unit architecture was lowered to an extent. In addition, efficient VLSI architectures were proposed for the check unit and the non-binary LDPC decoder. A non-binary LDPC code of (837,726) code over Galois field characteristic 32, $GF(2^5)$ was used. The proposed design achieved a throughput of $66Mbps$. But the achieved throughput was very low for SSD applications and there was a persistence of a very high decoding complexity.

Lacruz *et al.,* (2015) proposed a simplified implementation of the Min-Max algorithm. This algorithm was termed simplified Trellis Min-Max (TMM) algorithm. This was done by removing the subtraction block from the architecture and limiting the use of the minimum finder. This resulted in lowering the still very high complexity of the non-binary LDPC decoding architecture, by lowering the processing of the check node unit architecture. The messages of the check unit were calculated in a parallel way by making use of only the messages whose reliability is highest. The presented check unit architecture was achieved using a layered scheduling scheme that is horizontally computed. The decoder was implemented using a (837,726) non-binary LDPC code with a Galois field characteristic of 32, $GF(2^5)$. A throughput of 660 Mbps was achieved when the overall architecture of the decoder was implemented in a 90 nm CMOS technology at nine (9) iterations. But in this proposed decoder, a high throughput could not be attained. This made it unsuitable for applications that require high speed transfers. The hardware complexity of the decoder was still also very high, even with the modifications.

Lacruz *et al.,* (2016) proposed an innovative technique based on the trellis min–max algorithm for decoding NB-LDPC codes. This was done by lowering the amount of messages transferred between variable node units and check node units and also reducing the size of the memory used to store the intermediate messages. This resulted in a reduction in the decoding complexity and increased the throughput of the proposed decoder. The loss in performance of the proposed algorithm was small. In addition, the decoder was implemented using a layered decoding scheme and three non-binary LDPC codes, each having its own Galois field characteristic: (2304, 2048) over GF(2^4), (837, 726) in GF(2^5), and (1536, 1344) in GF(2^6). The decoder was implemented on a 90nm CMOS technology and attained a throughput of 1.08Gbps. Even with the reduction in the decoding complexity and the intermediate messages, the decoder still experienced high

hardware complexity as seen in the wiring congestion. The throughput was also below the desired speed for modern SSDs.

Thi & Lee., (2017a) proposed a basic-set trellis min–max algorithm. This was done by processing the check units in the decoder in a parallel manner. After this was done, the minimum of the messages sent from the check units to the variable units was computed using a minimum finder. This technique lowered the high complexity of the check unit. and also lowered the amount of variable unit messages and check unit messages that is to be stored in the memory. The decoder was implemented using a layered decoding scheme with two non-binary LDPC codes: (837, 726) NB-LDPC code and (1512, 1323) code, each having a Galois field characteristic of 32 $GF(2^5)$ and 64 $GF(2^6)$ respectively. The technology used is a 90nm CMOS. This decoder achieved a throughput of 1.67 Gbps and 1.4 Gbps respectively. The decoder still experienced high decoding complexity in the check node architecture with low throughput, which is not suitable for modern applications.

Thi & Lee, (2017b) proposed a new extra-two-column trellis min–max (MM), with forward-backward scheme and an architecture of a decoder based on only the computation of the first lowest values for non-binary LDPC codes. This was done by using an architecture called the one-minimum finder. This is a scheme that removed all the other values that are higher than the minimum value in the processing of the check units of the decoder architecture. This lowered the hardware complexity and the latency of the check unit scheme. The throughput of the decoder was also improved by the use of overlapping the unit architectures. The decoder was implemented using a layered decoding scheme and a (837, 726) non-binary LDPC code with a Galois characteristic field of 32, $GF(2^5)$. The technology used is a 90-nm CMOS and a throughput of 1.27 Gbps was achieved. As a result of the high Galois field used, the decoder still

has high complexity in the check node unit which resulted in complex interconnection network and large memory usage.

Choi *et al.*, (2017) proposed a non-binary LDPC decoder that is fully overlapped. This was implemented by using three different techniques. First, an early bubble check unit architecture was implemented by overlapping check node units and initiating quick parity checks. Second, the variable unit and the check unit were overlapped and thereafter stored in the same memory. Lastly, a redundant memory that can be used multiple times was used to store all the node unit architectures. The cumulative effect of all these techniques resulted in reduction of the initial latency of the check unit architecture and reduction in the latency of the decoder in hiding the latency of the check unit within the variable unit. This further decreased the latency and improved the throughput of the overall decoder. The whole decoder was implemented with a NB-LDPC code of (160,80) with 160 node units and 80 check units over Galois characteristic of sixty four (64), $GF(2^6)$. The achievable throughput was 2.22Gbps in a 65nm process technology. The whole decoder scheme suffered from high complexity in the interconnection network as a result of the high Galois field used.

Toriyama *et al.*, (2018) presented a NB-LDPC decoder that is suited to storage applications. with a throughput of 2.267Gbps. The non-binary code employed was a high rate code with a Galois field characteristic of eight (8). The work was achieved by the use of two decoding algorithms: the min-max decoding algorithm and the iterative hard decoding algorithm. The iterative hard decoding algorithm was implemented to reduce the complexity of the decoder that was incurred a result of the min-max algorithm. A logarithm quantization scheme was also used alongside the decoding algorithms to further reduce the complexity of the node unit of the decoder. A moderate throughput of 2.267Gbps was achieved. This was because of the high

number of the node units and the increase in the parallelism of the units. This is a moderately fast decoding throughput for a non-binary LDPC decoder. But this throughput was still will not meet the requirement of mission critical applications that require excellent error correction at very high throughput.

This review establishes the very high complexity of LDPC decoder schemes, and the need for a higher throughput with low power consumption during decoding. This laid the platform for the design of an error control decoder that decodes errors with high throughput.

CHAPTER THREE

MATERIALS AND METHODS

3.1 Introduction

In this chapter, the materials, the methods and reported procedure used for the implementation and synthesis of the non-binary error control decoder for solid state drives are described. The error control decoder architecture was written in Verilog with synthesis and implementation carried out in VIVADO Suite 2018.

3.2 Materials

The materials to be used for this research include the following:

1. A core i7 laptop with 2.6GHz speed and 16G RAM
2. Xilinx Field Programmable Gate Array (FPGA)

3.2.1 Computer System

Synthesis and implementation performed in this research work were carried out using the following materials:

i. Lenovo laptop computer with the following features
ii. Processor: Intel (R) Core (TM) i5-3470 CPU @ 2.60GHz
iii. Installed memory (RAM): 16.00GB (15.8 GB usable)
iv. System type: 64-bit Operating System, x64-based processor

3.2.2 Vivado

The Vivado design suite is the software package used to implement all the models developed in Verilog. For the purpose of this research, Vivado Suite 2018 version was used. Figure 3.1 illustrates the start page that comes up when the software is initialized. This page provides options to implement quick start, tasks and learning centre.

Figure 3.1: Vivado Design Suite 2018.2 Start Page

3.3 Methods

The methods followed to achieve the objectives are as follows:

1. Design of the NB-LDPC code:
    a. Generate the Parity-Check Matrix (H).
    b. Generate the NB-LDPC Code in Verilog.
2. Emulation of the error control code decoder architecture
    a. Build the module library based on the target LDPC code and declare the parameters and quantization bits
    b. Insertion of interconnect network and routing between the processing nodes
    c. Declaration of the decoding iteration limit
3. Synthesizing the architecture on the Zynq FPGA.
    a. The complete decoder is synthesized on the Zynq FPGA.

b. The performance of the developed system in terms of throughput and power consumption is compared with the work of (Toriyama & Markovic, 2018).

3.3.1 Design of the NB-LDPC Code

The design of the NB-LDPC code begins with the generation of the Parity-Check Matrix. The procedure is explained below:

3.3.1.1 Generation of the Parity Check Matrix (H)

The generation of the parity-check matrix begins with the creation of a non-binary base matrix (B_q). This base matrix is created from two arbitrary sets S_0 and $S_1$of a non-binary field. The process is as follows;

Let α to be a primitive element of Galois Field, GF(q) and q be the order of the field

where each non-zero element of GF(q) is written as α^i for integer i. Then,

Let $s = \{\alpha^0, \alpha^1, \alpha^2, \dots \dots \alpha^{q-2}\}$ be a set of nonzero elements of GF(q)

Let $B_q = [b_{i,j}]$, $0 \le i < m, 0 \le j < n$ be an m × n matrix over GF(q),

where $b_{i,j}$ is the non-zero elements of the base matrix and q denotes 'q-ary'

Let S_0 and S_1 be two additive subgroups of GF(q) with orders m and n, respectively, such that

$m + n \le q$ and $S_0 \cap S_1 = \{0\}$

Let η be any nonzero element in GF(q).

Then, the two sets S_0 and S_1 are represented as

$$S_0 = \{\alpha^{i_0}, \alpha^{i_1}, \dots., \alpha^{i_{m-1}}\} \tag{3.1}$$

$$S_1 = \{\alpha^{j_0}, \alpha^{j_1}, \dots., \alpha^{j_{n-1}}\} \tag{3.2}$$

Now, the value of m and n are selected so as to represent the minimum size of Galois field needed to create the base matrix.

Therefore, m = 29, and n = 9

$$m + n \leq q \quad (3.3)$$

$$29 + 9 \leq 64 \quad (3.4)$$

Therefore, Galois field $GF = 64 = 2^6$

Also a coefficient constant $\eta = 1$ is selected arbitrarily.

So, for field $GF(64), \eta = 1$

$$S_0 = \{\alpha^{i_0}, \alpha^{i_1}, \alpha^{i_2}, \alpha^{i_3}, \alpha^{i_4}, \alpha^{i_5}, \alpha^{i_6}, \alpha^{i_7}, \alpha^{i_8}\} \quad (3.5)$$

$$S_1 = \{\alpha^{j_0}, \alpha^{j_1}, \alpha^{j_2}, \alpha^{j_3}, \alpha^{j_4}, \alpha^{j_5}, \alpha^{j_6}, \ldots \ldots .., \ldots . \alpha^{j_n}\} \quad (3.6)$$

Therefore,

$$S_0 = \{\alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7, \alpha^8, \alpha^9\} \quad (3.7)$$

$$S_1 = \left\{ \begin{matrix} \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7, \alpha^8, \alpha^9, \alpha^{10}, \alpha^{11}, \alpha^{12}, \alpha^{13}, \alpha^{14}, \alpha^{15}, \alpha^{16}, \alpha^{17}, \alpha^{18}, \\ \alpha^{19}, \alpha^{20}, \alpha^{21}, \alpha^{22}, \alpha^{23}, \alpha^{24}, \alpha^{25}, \alpha^{26}, \alpha^{27}, \alpha^{28}, \alpha^{29}, \alpha^{30}, \alpha^{31} \end{matrix} \right\} \quad (3.8)$$

The base matrix is created using

$$\boldsymbol{B}_q = \begin{bmatrix} \eta\alpha^{i_0} + \alpha^{j_0} & \eta\alpha^{i_0} + \alpha^{j_1} & \eta\alpha^{i_0} + \alpha^{j_2} & \eta\alpha^{i_0} + \alpha^{j_3} & \eta\alpha^{i_0} + \alpha^{j_4} & \eta\alpha^{i_0} + \alpha^{j_5} & . & \eta\alpha^{i_0} + \alpha^{j_7} \\ & & & & & & . & \\ & & & & & & . & \\ & & & & & & . & \\ & & & & & & . & \\ & & & & & & . & \\ \eta\alpha^{i_7} + \alpha^{j_0} & \eta\alpha^{i_7} + \alpha^{j_1} & & & & & . & \eta\alpha^{i_7} + \alpha^{j_7} \end{bmatrix} \quad (3.9)$$

Therefore, substituting for the values into the base matrix gives

$$\boldsymbol{B}_q = \begin{bmatrix} \alpha^{4} & \alpha^{5} & \alpha^{6} & \alpha^{7} & \alpha^{8} & \alpha^{9} & \alpha^{10} & \alpha^{11} & \alpha^{12} & \alpha^{13} & \alpha^{14} & \alpha^{15} & \alpha^{16} & \alpha^{17} & \alpha^{18} & \alpha^{19} \\ \alpha^{5} & \alpha^{7} & \alpha^{10} & \alpha^{11} & \alpha^{12} & \alpha^{16} & \alpha^{17} & \alpha^{19} & \alpha^{22} & \alpha^{25} & \alpha^{31} & \alpha^{33} & \alpha^{35} & \alpha^{36} & \alpha^{39} & \alpha^{42} \\ \alpha^{6} & \alpha^{8} & \alpha^{11} & \alpha^{12} & \alpha^{13} & \alpha^{17} & \alpha^{18} & \alpha^{20} & \alpha^{23} & \alpha^{26} & \alpha^{32} & \alpha^{34} & \alpha^{36} & \alpha^{37} & \alpha^{40} & \alpha^{43} \\ \alpha^{7} & \alpha^{9} & \alpha^{12} & \alpha^{13} & \alpha^{14} & \alpha^{18} & \alpha^{19} & \alpha^{21} & \alpha^{24} & \alpha^{27} & \alpha^{33} & \alpha^{35} & \alpha^{37} & \alpha^{38} & \alpha^{41} & \alpha^{44} \\ \alpha^{8} & \alpha^{10} & \alpha^{13} & \alpha^{14} & \alpha^{15} & \alpha^{19} & \alpha^{20} & \alpha^{22} & \alpha^{25} & \alpha^{28} & \alpha^{34} & \alpha^{36} & \alpha^{38} & \alpha^{39} & \alpha^{42} & \alpha^{45} \\ \alpha^{9} & \alpha^{11} & \alpha^{14} & \alpha^{15} & \alpha^{16} & \alpha^{20} & \alpha^{21} & \alpha^{23} & \alpha^{26} & \alpha^{29} & \alpha^{35} & \alpha^{37} & \alpha^{39} & \alpha^{40} & \alpha^{43} & \alpha^{46} \\ \alpha^{10} & \alpha^{12} & \alpha^{15} & \alpha^{16} & \alpha^{17} & \alpha^{21} & \alpha^{22} & \alpha^{24} & \alpha^{27} & \alpha^{30} & \alpha^{36} & \alpha^{38} & \alpha^{40} & \alpha^{41} & \alpha^{44} & \alpha^{47} \\ \alpha^{11} & \alpha^{13} & \alpha^{16} & \alpha^{17} & \alpha^{18} & \alpha^{22} & \alpha^{23} & \alpha^{25} & \alpha^{28} & \alpha^{31} & \alpha^{37} & \alpha^{39} & \alpha^{41} & \alpha^{42} & \alpha^{45} & \alpha^{48} \\ \alpha^{12} & \alpha^{14} & \alpha^{17} & \alpha^{18} & \alpha^{19} & \alpha^{23} & \alpha^{24} & \alpha^{26} & \alpha^{29} & \alpha^{32} & \alpha^{38} & \alpha^{40} & \alpha^{42} & \alpha^{43} & \alpha^{46} & \alpha^{49} \end{matrix}$$

$$\begin{matrix} \alpha^{20} & \alpha^{21} & \alpha^{22} & \alpha^{23} & \alpha^{51} & \alpha^{53} & \alpha^{55} & \alpha^{56} & \alpha^{58} & \alpha^{59} & \alpha^{60} & \alpha^{61} & \alpha^{62} \\ \alpha^{43} & \alpha^{45} & \alpha^{47} & \alpha^{49} & \alpha^{52} & \alpha^{54} & \alpha^{56} & \alpha^{57} & \alpha^{59} & \alpha^{60} & \alpha^{61} & \alpha^{62} & \alpha^{63} \\ \alpha^{44} & \alpha^{46} & \alpha^{48} & \alpha^{50} & \alpha^{53} & \alpha^{55} & \alpha^{57} & \alpha^{58} & \alpha^{60} & \alpha^{61} & \alpha^{62} & \alpha^{63} & \alpha^{64} \\ \alpha^{45} & \alpha^{47} & \alpha^{49} & \alpha^{51} & \alpha^{54} & \alpha^{56} & \alpha^{58} & \alpha^{59} & \alpha^{61} & \alpha^{62} & \alpha^{63} & \alpha^{64} & \alpha^{65} \\ \cdots\cdots\alpha^{46} & \alpha^{48} & \alpha^{50} & \alpha^{52} & \alpha^{55} & \alpha^{57} & \alpha^{59} & \alpha^{60} & \alpha^{62} & \alpha^{63} & \alpha^{64} & \alpha^{65} & \alpha^{66} \\ \alpha^{47} & \alpha^{49} & \alpha^{51} & \alpha^{53} & \alpha^{56} & \alpha^{58} & \alpha^{60} & \alpha^{61} & \alpha^{63} & \alpha^{64} & \alpha^{65} & \alpha^{66} & \alpha^{67} \\ \alpha^{48} & \alpha^{50} & \alpha^{52} & \alpha^{54} & \alpha^{57} & \alpha^{59} & \alpha^{61} & \alpha^{62} & \alpha^{64} & \alpha^{65} & \alpha^{66} & \alpha^{67} & \alpha^{68} \\ \alpha^{49} & \alpha^{51} & \alpha^{53} & \alpha^{55} & \alpha^{58} & \alpha^{60} & \alpha^{62} & \alpha^{63} & \alpha^{65} & \alpha^{66} & \alpha^{67} & \alpha^{68} & \alpha^{69} \\ \alpha^{50} & \alpha^{52} & \alpha^{54} & \alpha^{56} & \alpha^{59} & \alpha^{61} & \alpha^{63} & \alpha^{64} & \alpha^{66} & \alpha^{67} & \alpha^{68} & \alpha^{69} & \alpha^{70} \end{bmatrix} \quad (3.10)$$

This results in a (9,29) Base matrix in primitive element form. Then a masking matrix (Z) is created for mapping unto the base matrix. This is to obtain a non-binary code with a minimum girth of eight (8). First a (9×29) matrix is created, with each element having weight one (1). Then a replacement of 1 to 0 is performed on the diagonal element starting from any position of the first row to the right at 45 degrees. After reaching the end of a column, the process starts again from the top of the next column, until the end of a row is reached. Then the process is repeated from the leftmost element of the next row. This continues until any 3×3 submatrix of masking matrix contains at least one 0-entry, and the masking matrix has the desired row and column weight factors.

Therefore,

$$\boldsymbol{Z} = \begin{bmatrix} 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&.&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&.&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&.&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&.&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&.&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&.&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&.&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&.&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&.&1&1&1&1 \end{bmatrix} \quad (3.11)$$

After permutations,

$$\widetilde{\boldsymbol{Z}} = \begin{bmatrix} 0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0 \\ 0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0 \\ 0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0 \\ 1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0 \\ 0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1 \\ 1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0 \\ 0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1 \\ 1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0 \\ 0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&0&1&0&0&0&0&1 \end{bmatrix} \quad (3.12)$$

Therefore, this results in a masking matrix with column weight of 3 ($d_v = 3$), and row weight of 9 ($d_c = 9$)

Now, mapping the masking matrix, Z unto the base matrix B_q gives

$$\boldsymbol{B_{q,mask}} = \begin{bmatrix} 0&0&6&0&8&0&10&0&0&0&0&15&0&17&0&19&0&0&0&0&51&0&55&0&58&0&0&0&0 \\ 0&0&0&11&0&16&0&19&0&0&0&0&35&0&39&0&43&0&0&0&0&54&0&57&0&60&0&0&0 \\ 0&0&0&0&13&0&18&0&23&0&0&0&0&37&0&43&0&46&0&0&0&0&57&0&60&0&62&0&0 \\ 7&0&0&0&0&18&0&21&0&27&0&0&0&0&41&0&45&0&49&0&0&0&0&59&0&62&0&64&0 \\ 0&10&0&0&0&0&20&0&25&0&34&0&0&0&0&45&0&48&0&52&0&0&0&0&62&0&64&0&66 \\ 9&0&14&0&0&0&0&23&0&29&0&37&0&0&0&0&47&0&51&0&56&0&0&0&0&64&0&66&0 \\ 0&12&0&16&0&0&0&0&27&0&36&0&40&0&0&0&0&50&0&54&0&59&0&0&0&0&66&0&68 \\ 11&0&16&0&18&0&0&0&0&31&0&39&0&42&0&0&0&0&53&0&58&0&62&0&0&0&0&68&0 \\ 0&14&0&18&0&30&0&0&0&0&38&0&42&0&46&0&0&0&0&56&0&61&0&64&0&0&0&0&70 \end{bmatrix} \quad (3.13)$$

This is the masked base matrix with column weight of 3 ($d_v = 3$), and row weight of 9 ($d_c = 9$)

Now, to construct the non-binary LDPC code, each non-zero element is replaced by a $(p \times p)$ CPM of $GF(8)$ and each zero element by a $(p \times p)$ ZM of zeros, where p represent the degree of parallelism. Toriyama *et al* (2018) made use of a 58 degree of parallelism. In this work, p is chosen to be 58. This is because, increasing the degree of parallelism, increases the throughput

of the error control decoder. Multiplying p by 10 gives 580. But when designing the decoder, the value 580 gives an error when synthesizing on the FPGA.

3.3.1.2 Generation of the NB-LDPC Code in Verilog

The dispersion of the CPM and the ZM into the matrix yields a sparse array of elements that result into the LPDC code. This matrix is then programmed in Verilog so as to be able to build the architecture of the error control decoder. Figure 3.2 illustrates the declaration of the nodes that make up the non-binary LDPC code. Each non-zero element in the matrix represent a connection between the variable node and the check node. Values within $GF(8)$ i.e. $\{0,1,2,3,4,5,6,7\}$ are used in the model, as seen in the initial value'3' used in Figure 3.2 below.

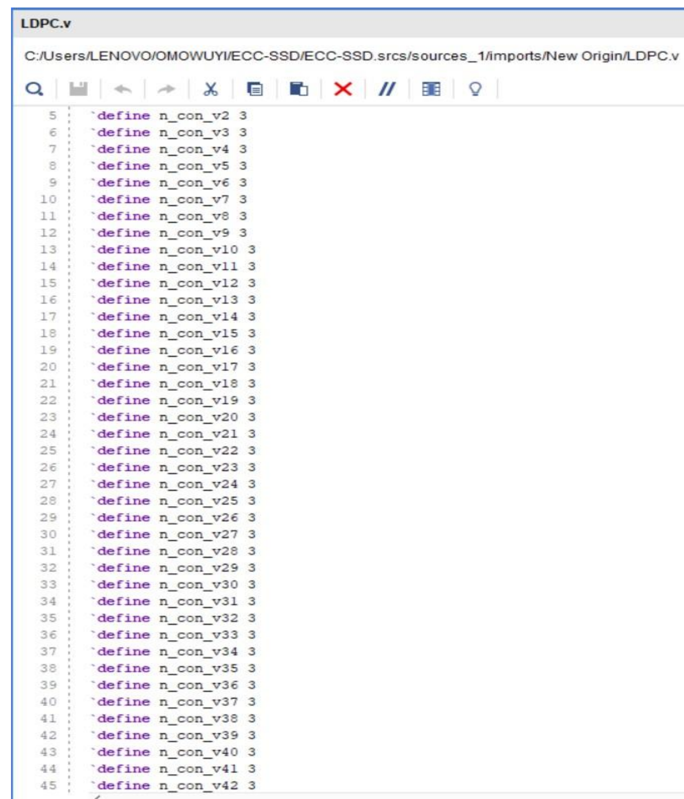


Figure 3.2: The NB-LDPC Code in Verilog

3.3.2 Emulation of the error control code decoder architecture

The emulation of the decoder begins with declaration of the module, ports and the signals. The parameters necessary for the full implementation of the LDPC module are declared.

3.3.2.1 Module Library Declaration with Parameters

Figure 3.3. Shows the LDPC module declaration. The LDPC module has its register, output and input ports declared, with their respective bit sizes. After the declaration, the wires that connect the nodes are labeled alongside their respective bits.

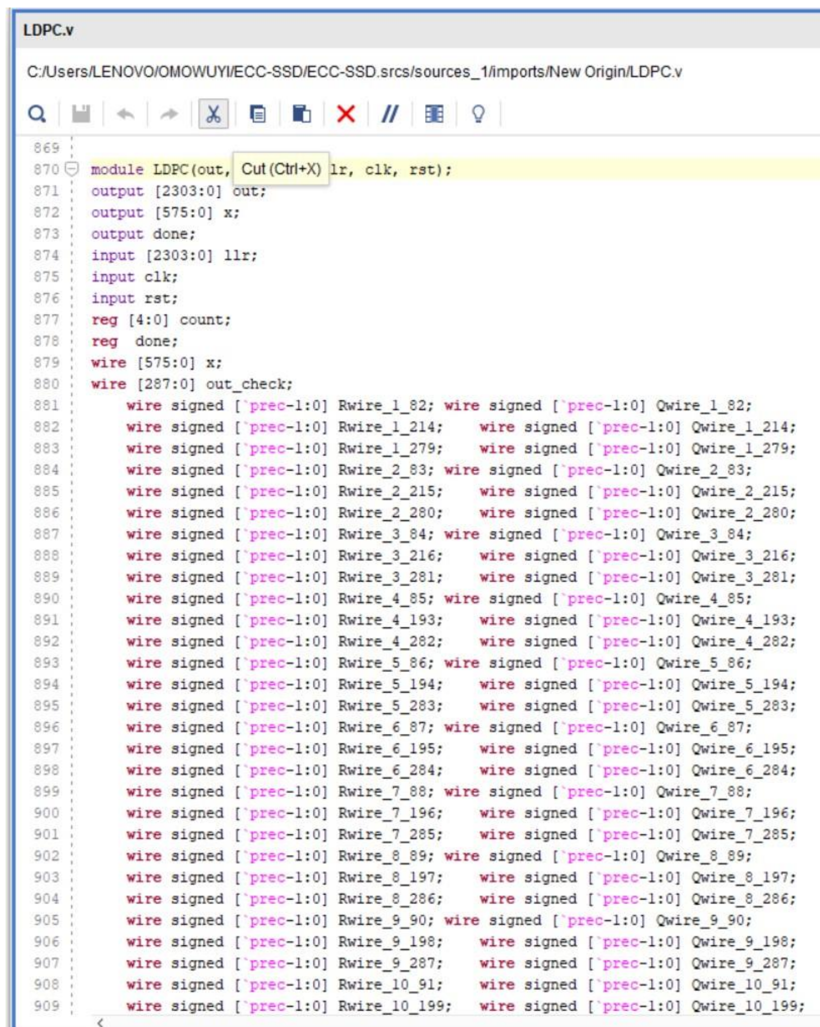


Figure 3.3: Verilog Description of the LDPC Module

The Q-wire signify the message transferred from the variable node to the check node, while the R-wire represent the message passed from the check node to the variable node.

3.3.2.2 Insertion of Interconnect Network and Routing

The messages passed between the nodes are done through routing networks. Bits and routing connections are assigned to every variable/check node connected to its respective check/variable node. Messages are passed only when there is a connection between nodes. Figure 3.4 shows the wires of the nodes and their respective bit assignments.

LDPC.v

C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/LDPC.v

```
2707     wire signed [`prec*`n_con_v1-1:0] Rwires_v1;    wire signed [`prec*`n_con_v1-1:0] Qwires_v1;
2708         assign Rwires_v1[3:0] = Rwire_1_82; assign Qwire_1_82 = Qwires_v1[3:0];
2709         assign Rwires_v1[7:4] = Rwire_1_214;    assign Qwire_1_214 = Qwires_v1[7:4];
2710         assign Rwires_v1[11:8] = Rwire_1_279;   assign Qwire_1_279 = Qwires_v1[11:8];
2711     wire signed [`prec*`n_con_v2-1:0] Rwires_v2;    wire signed [`prec*`n_con_v2-1:0] Qwires_v2;
2712         assign Rwires_v2[3:0] = Rwire_2_83; assign Qwire_2_83 = Qwires_v2[3:0];
2713         assign Rwires_v2[7:4] = Rwire_2_215;    assign Qwire_2_215 = Qwires_v2[7:4];
2714         assign Rwires_v2[11:8] = Rwire_2_280;   assign Qwire_2_280 = Qwires_v2[11:8];
2715     wire signed [`prec*`n_con_v3-1:0] Rwires_v3;    wire signed [`prec*`n_con_v3-1:0] Qwires_v3;
2716         assign Rwires_v3[3:0] = Rwire_3_84; assign Qwire_3_84 = Qwires_v3[3:0];
2717         assign Rwires_v3[7:4] = Rwire_3_216;    assign Qwire_3_216 = Qwires_v3[7:4];
2718         assign Rwires_v3[11:8] = Rwire_3_281;   assign Qwire_3_281 = Qwires_v3[11:8];
2719     wire signed [`prec*`n_con_v4-1:0] Rwires_v4;    wire signed [`prec*`n_con_v4-1:0] Qwires_v4;
2720         assign Rwires_v4[3:0] = Rwire_4_85; assign Qwire_4_85 = Qwires_v4[3:0];
2721         assign Rwires_v4[7:4] = Rwire_4_193;    assign Qwire_4_193 = Qwires_v4[7:4];
2722         assign Rwires_v4[11:8] = Rwire_4_282;   assign Qwire_4_282 = Qwires_v4[11:8];
2723     wire signed [`prec*`n_con_v5-1:0] Rwires_v5;    wire signed [`prec*`n_con_v5-1:0] Qwires_v5;
2724         assign Rwires_v5[3:0] = Rwire_5_86; assign Qwire_5_86 = Qwires_v5[3:0];
2725         assign Rwires_v5[7:4] = Rwire_5_194;    assign Qwire_5_194 = Qwires_v5[7:4];
2726         assign Rwires_v5[11:8] = Rwire_5_283;   assign Qwire_5_283 = Qwires_v5[11:8];
2727     wire signed [`prec*`n_con_v6-1:0] Rwires_v6;    wire signed [`prec*`n_con_v6-1:0] Qwires_v6;
2728         assign Rwires_v6[3:0] = Rwire_6_87; assign Qwire_6_87 = Qwires_v6[3:0];
2729         assign Rwires_v6[7:4] = Rwire_6_195;    assign Qwire_6_195 = Qwires_v6[7:4];
2730         assign Rwires_v6[11:8] = Rwire_6_284;   assign Qwire_6_284 = Qwires_v6[11:8];
2731     wire signed [`prec*`n_con_v7-1:0] Rwires_v7;    wire signed [`prec*`n_con_v7-1:0] Qwires_v7;
2732         assign Rwires_v7[3:0] = Rwire_7_88; assign Qwire_7_88 = Qwires_v7[3:0];
2733         assign Rwires_v7[7:4] = Rwire_7_196;    assign Qwire_7_196 = Qwires_v7[7:4];
2734         assign Rwires_v7[11:8] = Rwire_7_285;   assign Qwire_7_285 = Qwires_v7[11:8];
2735     wire signed [`prec*`n_con_v8-1:0] Rwires_v8;    wire signed [`prec*`n_con_v8-1:0] Qwires_v8;
2736         assign Rwires_v8[3:0] = Rwire_8_89; assign Qwire_8_89 = Qwires_v8[3:0];
2737         assign Rwires_v8[7:4] = Rwire_8_197;    assign Qwire_8_197 = Qwires_v8[7:4];
2738         assign Rwires_v8[11:8] = Rwire_8_286;   assign Qwire_8_286 = Qwires_v8[11:8];
2739     wire signed [`prec*`n_con_v9-1:0] Rwires_v9;    wire signed [`prec*`n_con_v9-1:0] Qwires_v9;
2740         assign Rwires_v9[3:0] = Rwire_9_90; assign Qwire_9_90 = Qwires_v9[3:0];
2741         assign Rwires_v9[7:4] = Rwire_9_198;    assign Qwire_9_198 = Qwires_v9[7:4];
2742         assign Rwires_v9[11:8] = Rwire_9_287;   assign Qwire_9_287 = Qwires_v9[11:8];
2743     wire signed [`prec*`n_con_v10-1:0] Rwires_v10;  wire signed [`prec*`n_con_v10-1:0] Qwires_v10;
2744         assign Rwires_v10[3:0] = Rwire_10_91;   assign Qwire_10_91 = Qwires_v10[3:0];
2745         assign Rwires_v10[7:4] = Rwire_10_199;  assign Qwire_10_199 = Qwires_v10[7:4];
2746         assign Rwires_v10[11:8] = Rwire_10_288; assign Qwire_10_288 = Qwires_v10[11:8];
2747     wire signed [`prec*`n_con_v11-1:0] Rwires_v11;  wire signed [`prec*`n_con_v11-1:0] Qwires_v11;
```

Figure 3.4: Routing and Interconnection in the LDPC Decoder

3.3.2.3 Declaration of Check Equation & Iteration Limit

The check equations that are used to determine the performance of the error control decoder are declared. The check nodes corresponding to the non-zero elements of the LDPC code form a series of equations that make up the combinatorial logic for the correction of errors in the system.

LDPC.v

C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/LDPC.v

```
8344  assign out_check[261] = x[70] + x[85] + x[198] + x[225] + x[549] + x[573];
8345  assign out_check[262] = x[71] + x[86] + x[199] + x[226] + x[550] + x[574];
8346  assign out_check[263] = x[48] + x[87] + x[200] + x[227] + x[551] + x[575];
8347  assign out_check[264] = x[10] + x[136] + x[178] + x[270] + x[289] + x[552];
8348  assign out_check[265] = x[11] + x[137] + x[179] + x[271] + x[290] + x[553];
8349  assign out_check[266] = x[12] + x[138] + x[180] + x[272] + x[291] + x[554];
8350  assign out_check[267] = x[13] + x[139] + x[181] + x[273] + x[292] + x[555];
8351  assign out_check[268] = x[14] + x[140] + x[182] + x[274] + x[293] + x[556];
8352  assign out_check[269] = x[15] + x[141] + x[183] + x[275] + x[294] + x[557];
8353  assign out_check[270] = x[16] + x[142] + x[184] + x[276] + x[295] + x[558];
8354  assign out_check[271] = x[17] + x[143] + x[185] + x[277] + x[296] + x[559];
8355  assign out_check[272] = x[18] + x[120] + x[186] + x[278] + x[297] + x[560];
8356  assign out_check[273] = x[19] + x[121] + x[187] + x[279] + x[298] + x[561];
8357  assign out_check[274] = x[20] + x[122] + x[188] + x[280] + x[299] + x[562];
8358  assign out_check[275] = x[21] + x[123] + x[189] + x[281] + x[300] + x[563];
8359  assign out_check[276] = x[22] + x[124] + x[190] + x[282] + x[301] + x[564];
8360  assign out_check[277] = x[23] + x[125] + x[191] + x[283] + x[302] + x[565];
8361  assign out_check[278] = x[0] + x[126] + x[168] + x[284] + x[303] + x[566];
8362  assign out_check[279] = x[1] + x[127] + x[169] + x[285] + x[304] + x[567];
8363  assign out_check[280] = x[2] + x[128] + x[170] + x[286] + x[305] + x[568];
8364  assign out_check[281] = x[3] + x[129] + x[171] + x[287] + x[306] + x[569];
8365  assign out_check[282] = x[4] + x[130] + x[172] + x[264] + x[307] + x[570];
8366  assign out_check[283] = x[5] + x[131] + x[173] + x[265] + x[308] + x[571];
8367  assign out_check[284] = x[6] + x[132] + x[174] + x[266] + x[309] + x[572];
8368  assign out_check[285] = x[7] + x[133] + x[175] + x[267] + x[310] + x[573];
8369  assign out_check[286] = x[8] + x[134] + x[176] + x[268] + x[311] + x[574];
8370  assign out_check[287] = x[9] + x[135] + x[177] + x[269] + x[288] + x[575];
8371  always@(posedge clk) begin
8372      if(rst) begin
8373          count <= 0;
8374          done <= 0;
8375      end
8376       else if((count == 6) || (out_check == 0)) begin
8377           done <= 1;
8378      end
8379      else if (count < 6) begin
8380          count <= count + 1;
8381      end
8382  end
8383  endmodule
8384
```

Figure 3.5: Check Equations and Iteration Limit Declaration

The procedural statement alongside the sensitivity list that begin the sequential circuit, determine the number of iterations that occurred for complete error correction as well as the iteration limit of the decoder. Figure 3.5 illustrates the output check equations and the iteration limit of the decoder circuit.

3.3.3 Synthesis of the Architecture on the Zynq FPGA

The decoder architecture is categorized by modules, and is briefly described by the logic resources used during synthesis and optimization. XST performs during FPGA synthesis, both mapping and optimization on the complete design. The complete decoder architecture as shown in Figure 3.6, is synthesized on the ZYNQ FPGA Board.

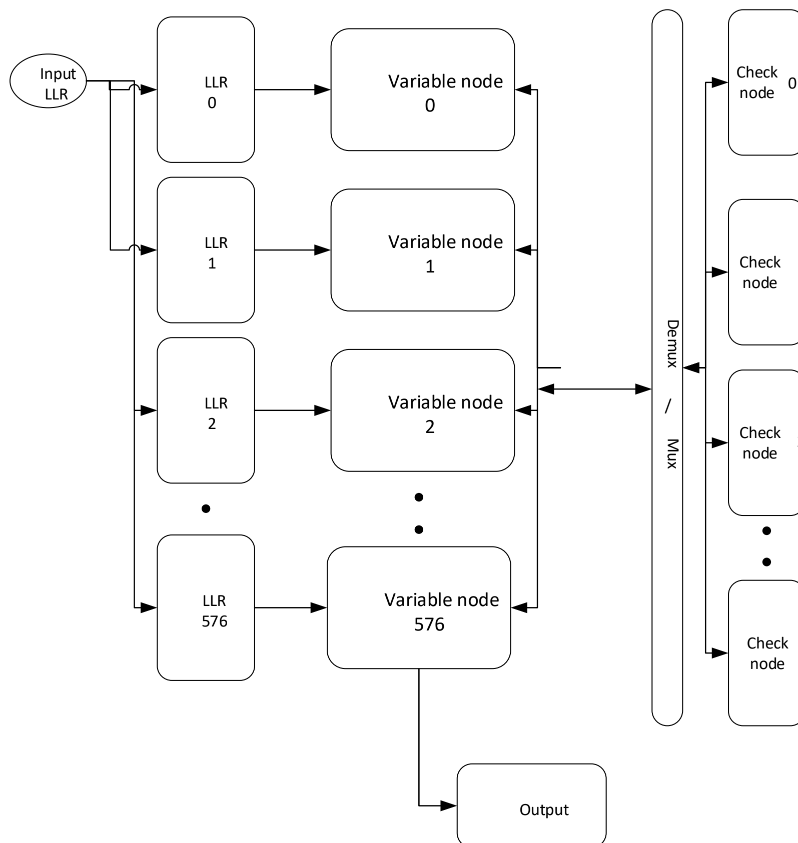


Figure 3.6: Architecture of the Non-binary LDPC Decoder

The variable node and check node units are arranged in parallel to achieve a very high throughput. This results in the utilization of more logic resources which will translate to a higher segmentation of the memory block. Data is passed between the variable and check node units through the multiplexer/demultiplexers.

The decoder calculates an initial log likelihood ratio (LLR) for every bit of the code. Message LLR comprising of values of LLR for every bit, are transferred among the check units and bit units during each iteration. The decoder allows each check unit use its information of parity check to know the degree that the value LLR of each bit is changed, using the current message LLRs from the variable units. Each variable unit collate the updated LLRs from each bit to produce a new LLR value, using the most current message LLRs from the check units. The equations of parity check are used to know if the values estimated by the new message LLRs for each unit have no error. The process stops when the estimated bit values have no error after the last iteration or when the maximum iteration limit is attained.

CHAPTER FOUR

RESULTS AND DISCUSSION

4.1 Introduction

In this chapter, the results for the research work are presented and discussed. The synthesis of the error control decoder is evaluated and analysed.

4.2 Synthesis

Vivado IDE 2018.2 performs the logic specification of the behaviour of the error control decoder at the RTL level. It is changed into an implementation of the architecture in logic gates. Figure 4.1 illustrates the window showing the complete synthesis of the error control decoder architecture

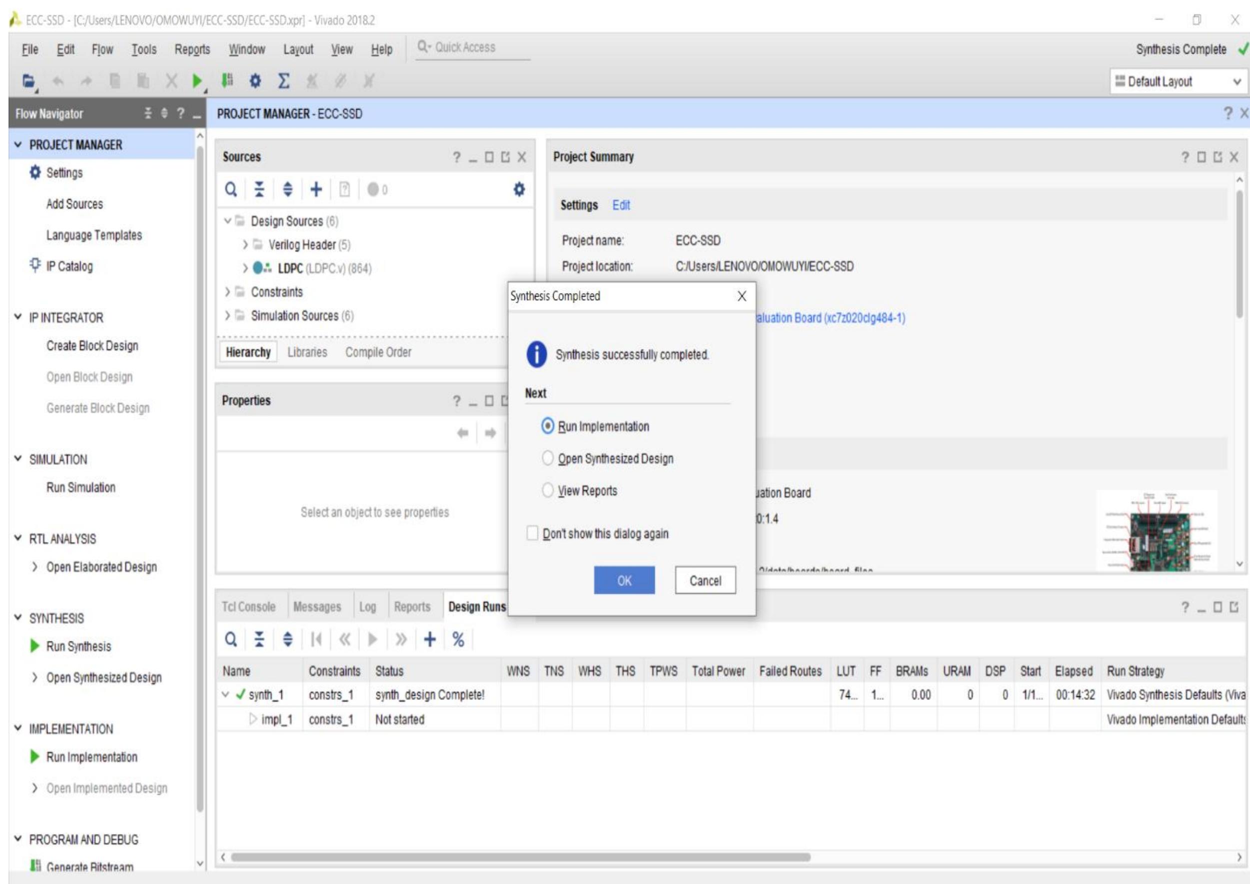


Figure 4.1: Project Manager Depicting Completed Synthesis of the Design

The Vivado project manager monitors the synthesis run from the Log window. The project manager gives a summarized detail of the synthesized design.

4.2.1 Register Transfer Level Analysis

Vivado 2018.2 transforms the RTL code into a gate-level description that goes through logic optimization and simplification. A gate-level netlist that is optimized is derived from the mapping of the logic gates. Reports that contain details about the usage of the cell and utilization is generated and documented in Appendix B. The output is a netlist of the synthesized modules and is illustrated in the top level schematic in Appendix A.

The Schematic window permits the display selective logic expansion. At the RTL level in Elaborated Design, the interpretation of the code is shown. The logic representation of the parity check matrix is built. The nodes in a layer are connected to the next layer where processing results are transferred. The Synthesis tool generates the gates. The gates, connectivity and hierarchy, are displayed at the upper level of the design, as shown in the schematic view of the LDPC logic in Appendix A

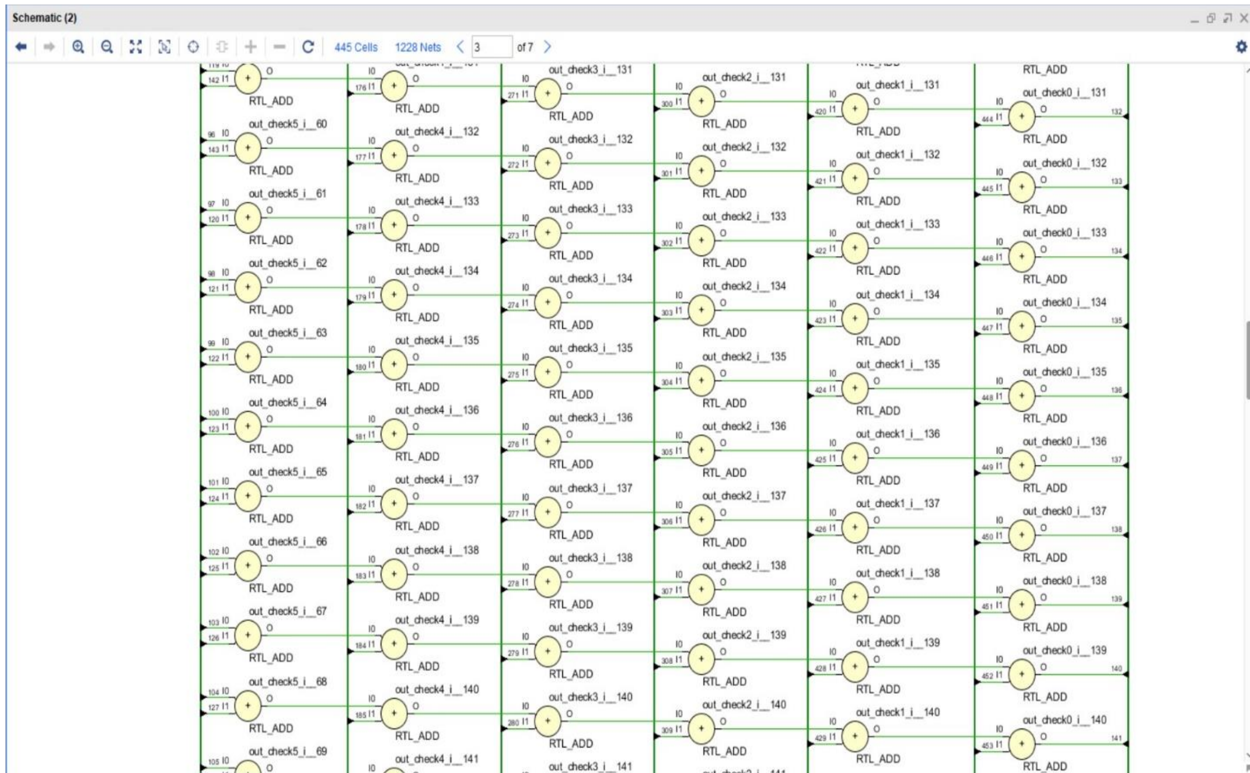


Figure 4.2: Elaborated View of the LDPC Logic

The elaborated view shown in Figure 4.2 depicts the adder-input and output circuitry of the nodes and the check equation logic. The elaborated window illustrates the RTL synthesized logic. For every iteration, the variable node is processed and the result is fed to the check node for further processing. Afterwards the check equation logic performs at each node the correction of errors. The final output result is gotten when the maximum iteration limit is reached or when the output value is zero.

4.2.2 RTL Synthesized Design

The Device window gives a graphical view of the synthesized design as well as the connections and, logic objects placement. The Synthesized Design window shown in Figure 4.3 illustrates a graphical implementation of the connection of the RTL logic of the error control decoder. Each tile is located sized in relative proportion to the others.

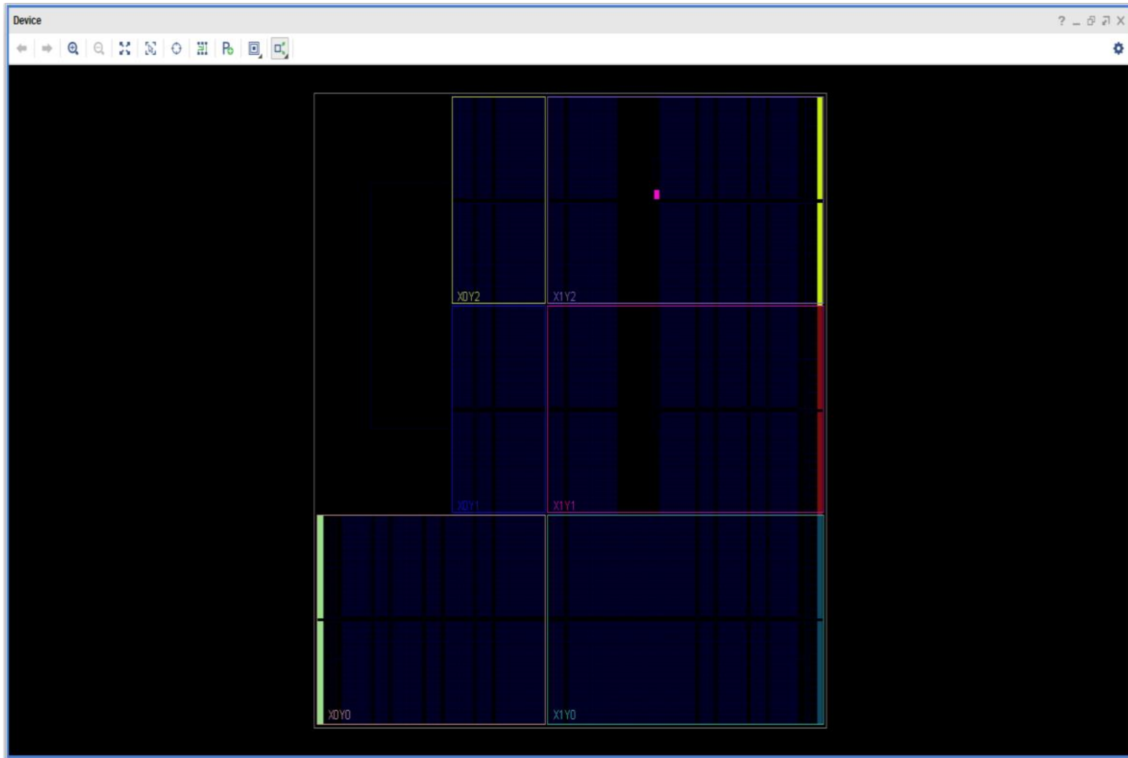


Figure 4.3: Synthesized Design Window of the LDPC Decoder

Figure 4.3 gives the synthesized design of the overall decoder. The complete rectangular block illustrates the ZYNQ architecture of the decoder. It is further divided into smaller rectangular regions that show the programmable system and the programmable logic. The top left section of the architecture is the programmable system which contains the ARM cortex processor. This processor is not utilized in this synthesized design. The remaining section of the architecture is the programmable logic, which contains the CLBs, IOBs, switch matrix, slices and RAMs. The small blue tiles represent the CLBs. The black rows and columns in-between the CLBs represent the switch matrix while the yellow, red, green and blue coloured tiles that are at the edges of the blocks represent the IOBs. The utilization and properties of each of the logic elements in the synthesized design in Figure 4.3 is provided in appendix B and C.

4.3 Synthesis Utilization

The Vivado IDE performs important functions of arranging, compiling and synthesizing all the Verilog source files and designs of the ECC decoder. This result is the synthesis utilization of every component of the ECC decoder

4.3.1 Utilization Report

After the completion of the synthesis, the design is analysed and report generated. The Utilization Report breaks down the design utilization with respect to resource type.

Table 4.1: Summary of Utilization Report of Resource Usage

Resource	Utilization	Available	Utilization (%)
LUT	49763	53200	96.00
FF	14598	106400	13.72

Table 4.1 gives the available resource, the number of logic resource utilized and the utilization ratio of the logic elements contained in the FPGA. The percentage usage of the LUT and FF are 96.00 and 13.72 percent respectively.

From both Table 4.1, it can be seen that more LUT than available were utilized by the decoder architecture. But the utilization of FF was within the resource provided by the FPGA. Appendix B and C give the detailed report of the utilization of the logic elements.

4.3.2 Power Usage

The power utilized by the decoder is calculated using equation (2.5) as follows:

$$P_D = (1 * 1.18 * 10^{-9} * \ 1.2^2 * \ 125 * 10^6) \quad (4.1)$$

$$P_D = 0.212W \quad (4.2)$$

Static power, $P_S = 0.011W$

Total power,

$$P = P_D + P_S = 0.212W + 0.011W \quad (4.3)$$

$$P = 0.223W$$

Where f_{clk} is the system frequency (125MHz), β is the activity factor (β = 1, because all nodes are switching at the same rate as the frequency), V_{DD} is the source voltage (1.2V) and C is the capacitance ($1.18 * 10^{-9}$F).

The degree of parallelism utilized is this work is 58. The dynamic power measured and the static power during implementation of the decoder at the chosen degree of parallelism are 0.02W/bit node and 0.083W/bit node respectively. Total measured power,

$$P_T = (0.02 + 0.083)W/bit\ node \quad (4.4)$$

$$P_T = 0.103W/bit\ node$$

Therefore, total power measured,

$$P_T = 0.103 * 58 = 5.97W$$

The power consumed by the decoder is summarized as shown in Table 4.2.

Table 4.2: Relationship Between Throughput and Power Consumption

Degree of parallelism	**Throughput (Gbps)**	**Calculated Power (W)**	**Measured Power (W)**
58	2.34	0.223	5.97
116	4.68	0.223	11.95
174	7.03	0.223	17.92
232	9.37	0.223	23.90
290	11.71	0.223	29.87
348	14.05	0.223	35.84
406	16.39	0.223	41.82
464	18.74	0.223	47.79
522	21.08	0.223	53.77
580	23.42	0.223	68.18

Table 4.2 shows the relationship between throughput and power consumption. As the degree of parallelism increases, so also does the throughput and the power measured. The calculated power (i.e dynamic power) remains constant. This is because the parameters that determine the calculated power do not change. This is seen in equation (2.5).

Table 4.3: Summary of Power Consumption of Logic Resource

Degree of parallelism	**Throughput (Gbps)**	**Calculated Power (W)**
0	0	0
25	0.47	0.053
50	0.94	0.096
75	1.41	0.138
100	1.87	0.181
125	2.34	0.223
150	2.81	0.266
175	3.28	0.308
200	3.75	0.351
580	23.42	0.223

Table 4.3, shows the relationship between the throughput and the calculated power (i.e dynamic power) as the frequency changes. From Table 4.3, it can be seen that as the frequency increases so also does the throughput and the power. This relationship is also captured in equation (2.5). The degree of parallelism is kept constant at 58.

4.3.3 Performance Comparison and Analysis

In order to evaluate the performance of the error control decoder, the throughput and power consumption are used as metric.

The throughput of the design is analysed as written in equation (2.4). From the synthesis of the decoder architecture, the following parameters were measured:

f_{clk} = 125MHz

$I_{max} = 6$

$N = 576$

M =288

$p = 58$

$d_v = 4$

$D = 2$

$q = 8$

Therefore, calculating the throughput:

$$T = \frac{125 \times 576 \times 58}{6 \times (288 + 4\ X\ 2) + (8 - 1)} Mbps \quad (4.5)$$

Therefore,

$$T = 2.34Gbps$$

This design outperforms that of Toriyama *et al* in terms of throughput. This is as result of certain factors, which are: i) increased parallel processing of the nodes, ii) improved frequency and iii) reduced number of iterations. In terms of amount of logic units, Toriyama *et al* (2018) used 142,822 LUTs as compared to 49763 LUTs and 14598 FFs used in this designed architecture.

In terms of power consumption, this decoder architecture consumed a total power of 0.223W, while that of Toriyama *et al* 2018 consumed 0.212W. This is as a result of the high utilization of logic elements by the decoder architecture on the FPGA.

CHAPTER FIVE

CONCLUSION AND RECOMMENDATION

5.1 Summary

In this dissertation, the development of an error control decoder for solid state drives has been presented. The synthesis of the hardware description language on an FPGA provides insight into the decoding capability of LDPC error control codes. The decoder architecture in terms of throughput and power consumption achieves 2.34Gbps at 0.223W as compared to 2.267Gbps at 0.212W of Toriyama *et al* 2018.

5.2 Conclusion

The research presents the development of an error control decoder for solid state drives. The non-binary error control decoder was developed in Verilog and synthesized on the ZYNQ 7000 FPGA. A parallel decoder architecture that utilizes all the nodes in the scheme utilizes a large amount of logic blocks of the ZYNQ development board. This decoder design synthesized on the FPGA achieves a throughput of 2.34Gbps, and consumed a total power of 0.223W. This throughput is critical for applications that need very fast storage drives. But the decoder consumed more power.

5.3 Limitations

In the process of implementation of the decoder architecture, it was discovered that, the number of input/output ports of the synthesized design exceeded that provided by the FPGA. A bank of FPGAs will be needed to accommodate the synthesized design. Due to the cost of purchasing the bank of FPGAs, a full implementation of the decoder architecture could not be accomplished.

5.4 Significant Contributions

The significant contributions by this research are as follows:

1. A non-binary LDPC error control decoder for SSDs was developed.
2. The developed algorithm exploited a parallel architecture that achieved a very high throughput. This is required in today's state of the art SSDs used in servers.
3. The developed decoder architecture achieved a 7.3% improvement in its throughput when compared with that of Toriyama *et al* 2018.

5.5 Recommendations for further work

The following considerations are recommended for further research:

1. Several fully parallel decoder cores can be instantiated for even higher throughput.
2. Banks of more powerful FPGAs can be utilized for the synthesis of the decoder and its architectural implementation.
3. Further improvements and optimization can be made by implementing the LDPC parallel decoder on an ASIC chip.
4. The complexity of the routing interconnect of the SSD decoder can be reduced by implementing lower bits for the check and bit nodes.

REFERENCES

Cai, Y., Ghose, S., Haratsch, E. F., Luo, Y., & Mutlu, O. (2017). Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives. *Proceedings of the IEEE*, *105*(9), 1666–1704. https://doi.org/10.1109/JPROC.2017.2713127

Chang, K. K., Nair, P. J., Lee, D., Ghose, S., Qureshi, M. K., & Mutlu, O. (2016a). Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. *Proceedings - International Symposium on High-Performance Computer Architecture*, *2016-April*, 568–580. https://doi.org/10.1109/HPCA.2016.7446095

Chang, K. K., Nair, P. J., Lee, D., Ghose, S., Qureshi, M. K., & Mutlu, O. (2016b). Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. *Proceedings - International Symposium on High-Performance Computer Architecture*, *2016-April*, 568–580. https://doi.org/10.1109/HPCA.2016.7446095

Chen, X., & Wang, C. L. (2012). High-throughput efficient non-binary LDPC decoder based on the simplified min-sum algorithm. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *59*(11), 2784–2794. https://doi.org/10.1109/TCSI.2012.2190668

Choi, I., Member, S., & Kim, J. (2017). *High-Throughput Non-Binary LDPC Decoder*. 1–12.

Codes, N. L. P., Cai, F., Member, S., Zhang, X., & Member, S. (2013). *Relaxed Min-Max Decoder Architectures for*. *21*(11), 2010–2023.

Crockett, L., Elliot, R., Enderwitz, M., & Stewart, R. (2014). *The Zynq Book Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. 484. Retrieved from All Papers/E/Elliot,Ross 2014 - The_Zynq_Book_ebook.pdf

Dolecek, L. (2014). Making Error Correcting Codes Work for Flash Memory Part I : Primer on ECC , basics of BCH and LDPC codes ECC is a must for Flash ! *Flash Memory Summit*.

Eshghi, K., & Micheloni, R. (2018). SSD architecture and PCI express interface. In *Springer Series in Advanced Microelectronics* (37),1–27. https://doi.org/10.1007/978-981-13-0599-3_1

Koelling, B., Manager, S., Product Marketing, S., & Bertholdt, J. (2015). *Altera Corporation Zynq to SoC FPGA Design Migration Tips and Techniques*. (April), 1–18. Retrieved from https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/tb-111-zynq-to-soc-fpga.pdf

Kumar, P. (2004). Book reviews - Error control coding; Fundamentals and applications. *IEEE Communications Magazine*, *21*(6), 48–49. https://doi.org/10.1109/mcom.1983.1091452

Lacruz, J., García-Herrero, F., Canet, M. J., & Valls, J. (2016). Reduced-Complexity Nonbinary LDPC Decoder for High-Order Galois Fields Based on Trellis Min-Max Algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *24*(8), 2643–2653. https://doi.org/10.1109/TVLSI.2016.2514484

Lacruz, J. O., García-Herrero, F., Canet, M. J., & Valls, J. (2016). High-Performance NB-LDPC

Decoder With Reduction of Message Exchange. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *24*(5), 1950–1961. https://doi.org/10.1109/TVLSI.2015.2493041

Lacruz, O., Garc, F., Declercq, D., Member, S., & Member, J. V. (2015). Simplified Trellis Min-Max Decoder Architecture. *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, (32), 1–10.

Lee, K., Lim, S., & Kim, J. (2012). *Decoder for NAND Flash Memory*. 413–415.

Lee, Y., Yoo, H., Yoo, I., & Park, I. C. (2012). 6.4Gb/s multi-threaded BCH encoder and decoder for multi-channel SSD controllers. *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, *55*(8), 426–427. https://doi.org/10.1109/ISSCC.2012.6177075

Lin, J., Sha, J., Wang, Z., & Li, L. (2010). An efficient VLSI architecture for nonbinary LDPC decoders. *IEEE Transactions on Circuits and Systems II: Express Briefs*, *57*(1), 51–55. https://doi.org/10.1109/TCSII.2009.2036542

Micheloni, R., Marelli, A., & Eshghi, K. (2018). *Inside solid state drives (SSDs)* (Vol. 37).

Nicola, B., Reis, R., Graziano, P., Masahiro, F., & Todd, A. (2018). *VLSI-SoC : Design and Engineering of Electronics Systems Based on New*. Springer.

Reviriego, P., Argyrides, C., & A. Maestro, J. (2012). Efficient error detection in Double Error Correction BCH codes for memory applications. *Microelectronics Reliability*, *52*(7), 1528–1530. https://doi.org/10.1016/j.microrel.2012.01.017

Smith, K. (2020). Low-Density Parity Check Error Correction for Solid State Storage. *Electronic Design*, (8) 1–13.

Thi, H. P., & Lee, H. (2017a). Basic-Set Trellis Min-Max Decoder Architecture for Nonbinary LDPC Codes with High-Order Galois Fields. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *26*(3), 496–507. https://doi.org/10.1109/TVLSI.2017.2775646

Thi, H. P., & Lee, H. (2017b). Two-Extra-Column Trellis Min-Max Decoder Architecture for Nonbinary LDPC Codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *25*(5), 1787–1791. https://doi.org/10.1109/TVLSI.2017.2647985

Toriyama, Y., & Markovic, D. (2018). A 2.267-Gb/s, 93.7-pJ/bit Non-Binary LDPC decoder with logarithmic quantization and dual-decoding algorithm scheme for storage applications. *IEEE Journal of Solid-State Circuits*, *53*(8), 2378–2388. https://doi.org/10.1109/JSSC.2018.2832851

Wang, J., Vakilinia, K., Chen, T. Y., Courtade, T., Dong, G., Zhang, T., … Wesel, R. (2014). Enhanced precision through multiple reads for LDPC decoding in flash memories. *IEEE Journal on Selected Areas in Communications*, *32*(5), 880–891. https://doi.org/10.1109/JSAC.2014.140508

Xilinx, & Inc. (2019). *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 SoC User Guide*.

850, 78. Retrieved from www.xilinx.com

Yang, L., Liu, F., & Li, H. (2008). Min-max decoding for non-binary LDPC codes. *Lecture Notes in Electrical Engineering*, *210 LNEE*, 125–134. https://doi.org/10.1007/978-3-642-34528-9_14

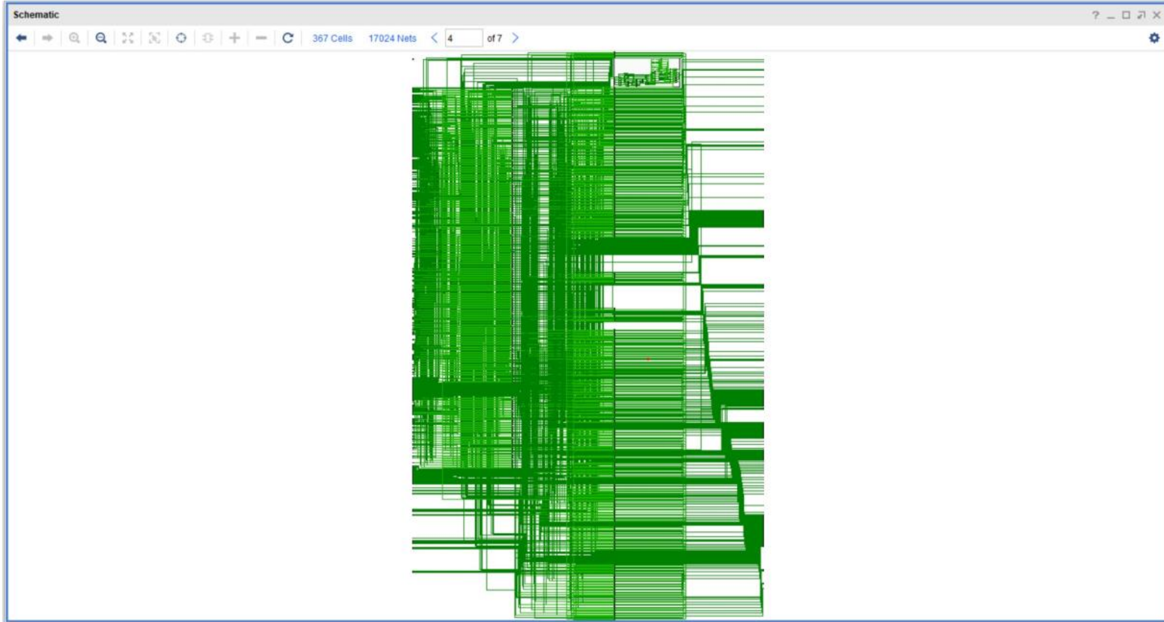
Zhang, X., Cai, F., Member, S., & Nb-ldpc, A. N. (2011). *Reduced-Complexity Decoder Architecture for*. *19*(7), 1229–1238.

Zhao, K., Wenzhe, Z., Hongbin, S., Zhang, T., Xiaodong, Z., & Zheng, N. (2013). LDPC-in-SSD: making advanced error correction codes work effectively in solid state drives. *Proceedings of 11th USENIX Conference on File and Storage Technologies*, 243–256.

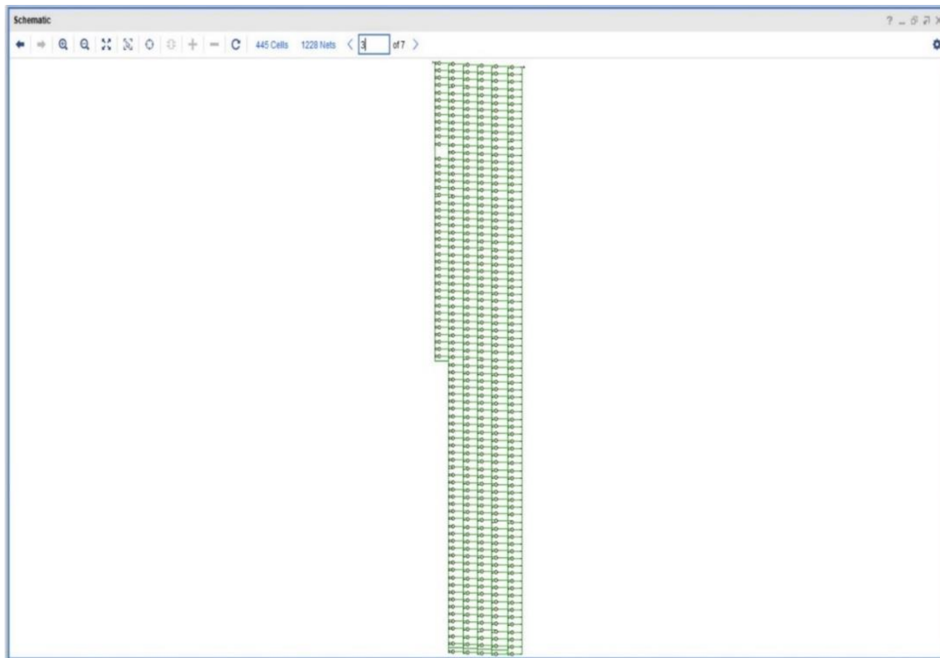
APPENDIX

APPENDIX A

Vivado Synthesized Designs



Top Level Schematic of the Synthesized Design.



Schematic View of the LDPC Logic

APPENDIX B

Synthesis and Implementation Report

*** Vivado Running

Vivado soft v2018.2 (64/bit)

Build 2258646 on Thu Jun 14 21:03:12 MDT 2018

**** IP Build 2256618 on Thu Jun 14 22:10:49 MDT 2018

** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

INFO: [Synth 8-2490] definition of module QAdd [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/QAdd.v:1]

INFO: [Synth 8-2490] definition of module QSub [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/QSub.v:1]

Commencing Synthesize: Time: Memory (MB): gain. = 109.793peak = 428.414 cpu = 00:00:04; elapse. = 00:00:02;

INFO: [Synth. 8-6157] synthesize module 'LDPC' [C:/Users./LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/LDPC.v:870]

INFO: [Synth 8-6157] synthesizing module 'CheckNode' [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/CheckNode.v:2]

Parameter num_connections bound. to: 6 – type.: integer

Parameter. prec bound. to: 4 - type: integer

INFO: [Synth 8-6157] synthesize module 'Comparator' [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs./sources.1/imports/New. Origin/Comparator.v:1]

Parameter prec bound. to: 4 – type. integer

INFO: [Synth. 8-6155] done synthesizing module 'Comparator' (1#1) [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/Comparator.v:1]

INFO: [Synth. 8-6155] done synthesize module 'CheckNode' (2#1) [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/CheckNode.v:2]

INFO: [Synth. 8-6157] synthesize module 'CheckNode__parameterized0' [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/CheckNode.v:2]

INFO: [Synth 8-6155] done synthesizing module 'VarNode__parameterized0' (5#1) [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/VarNode.v:4]

INFO: [Synth 8-6157] synthesizing module 'VarNode__parameterized1' [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/VarNode.v:4]

Parameter. num-connections bound-to: 2. – type-integer

Parameter. prec. bound. to: 4type: integer

INFO: [Synth. 8-6155] done synthesize module 'VarNode__parameterized1' (5#1) [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/VarNode.v:4]

INFO: [Synth. 8-6155] done synthesize module 'LDPC' (6#1) [C:/Users/LENOVO/OMOWUYI/ECC-SSD/ECC-SSD.srcs/sources_1/imports/New Origin/LDPC.v:870]

Complete Synthesize: Time: gain. = 198.594; elapsed. = 00:00:09. Memory. (MB): peak. = 517.215; cpu. = 00:00:09

Complete Constraint Validation: Time (s): gain. = 198.594; elapsed = 00:00:10. Memory. (MB): peak. = 517.215; cpu. = 00:00:09

Begin Timing Information & Loading Part

Loading. part: xc7z020clg484-1

Complete Timing Information & Loading Part: Time: elapsed. = 00:00:10. Memory. (MB): peak. = 517.215; gain. = 198.594; cpu. = 00:00:09;

INFO: [Device 21-403] Loading part xc7z020clg484-1

Complete RTL Optimization-Phase 2 : Time: cpu. = 00:00:24;.peak. = 626.969; gain. = 308.348; elapsed = 00:00:19; Memory (MB)

APPENDIX C

Tile Properties

Name:	CLBLM_L_X54Y24
Type:	CLBLM_L
Row:	131
Column:	133
Clock Region:	X1Y0
Number of Cell Pins:	0
No. of Cell:	0
No. of Ports:	0
No. of BELs:	120
No. of sites:	2
No. of nodes:	313
Number of Switchboxes:	0
Number of clock regions:	1

CLBLM_L_X54Y24

Site Type	Available	Required	% Util
PHY_CONTROL	1	0	0.00
PHASER_REF	1	0	0.00
IDELAYCTRL	1	0	0.00
MCME2_ADV	1	0	0.00
PLE2_ADV	1	0	0.00
BUFMCE	2	0	0.00

FIFO (OUT)	4	0	0.00
FIFO (IN)	4	0	0.00
PHASER_OUT/PHASER_OUT_PHY	4	0	0.00
PHASER_IN/PHASER_IN_PHY	4	0	0.00
BUFIO	4	0	0.00
BUFR	4	0	0.00
Block RAM Tile	30	0	0.00
RAMB36/FIFO	30	0	0.00
IBUFDS	48	0	0.00
Bonded IOB	50	0	0.00
IDELAYE2/IDELAYE2_FINEDELAY	50	0	0.00
ILOGIC	50	0	0.00
OLOGIC	50	0	0.00
RAMB18	60	0	0.00
DSPs	60	0	0.00
F8 Muxes	2500	0	0.00
LUT as Memory	4000	0	0.00
F7 Muxes	5000	0	0.00
Slice LUTs	10000	0	0.00
Logic LUT	10000	0	0.00
Slice	20000	0	0.00
Flip Flop (Register)	20000	0	0.00
Latch (Register)	20000	0	0.00