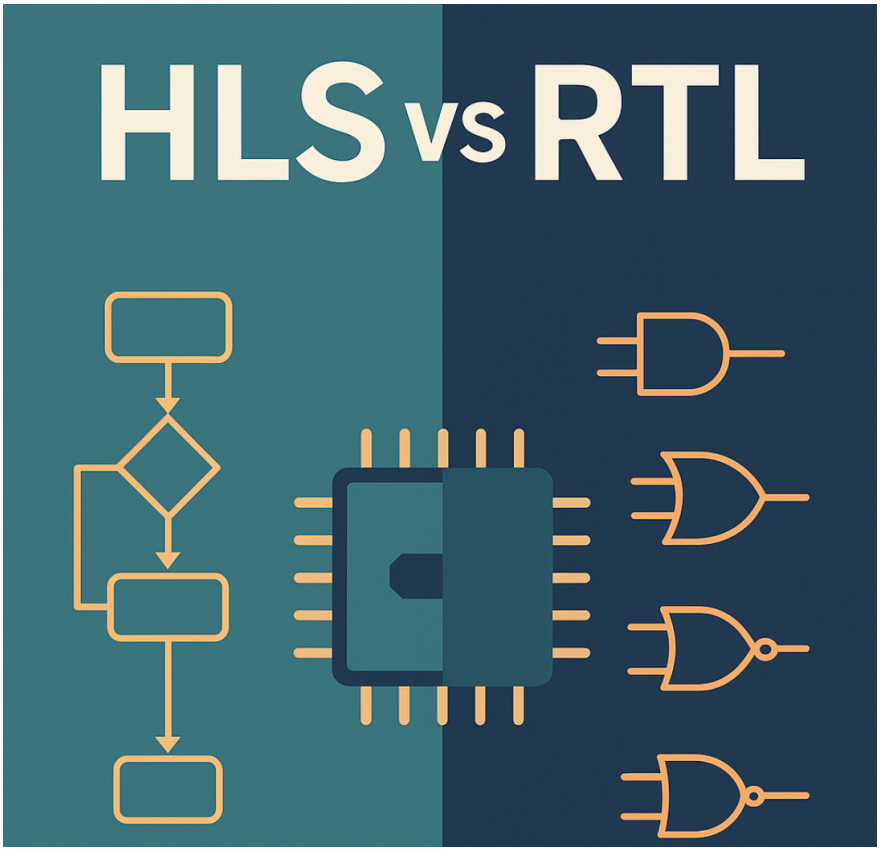
CHALMERS
UNIVERSITY OF TECHNOLOGY



Comparative Performance Analysis of High-Level Synthesis and RTL Implementations of LDPC Decoders

Master's thesis in Embedded Electronic System Design

Kavineshver Sivaraman Kathiresan

Prashanth Raj

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025

Master's thesis 2025

Comparative Performance Analysis of High-Level Synthesis and RTL Implementations of LDPC Decoders

Kavineshver Sivaraman Kathiresan
Prashanth Raj

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2025

Comparative Performance Analysis of High-Level Synthesis and RTL Implementations of LDPC Decoders

Kavineshver Sivaraman Kathiresan

Prashanth Raj

Supervisor: Lars Svensson, Department of Micro and Nanotechnology.
Examiner: Per Larsson-Edefors, Department of Micro and Nanotechnology.

Master's Thesis 2025
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Image showing HLS and RTL being compared

Typeset in LaTeX
Gothenburg, Sweden 2025

Abstract

Low density parity check codes are an important type of error-correcting codes used in many modern communications standards. Implementing LDPC decoders efficiently in hardware is a key requirement, as these systems demand both high throughput and low latency. This thesis compares two design approaches for LDPC decoder implementation: the conventional Register Transfer Level (RTL) method and High-Level Synthesis (HLS). The RTL design provides detailed control over timing and hardware resources but requires significant development effort. HLS, on the other hand, allows designers to describe the algorithm in C/C++ and semi-automatically generate RTL, reducing design time and offering faster exploration of architectural trade-offs. In this work, both RTL and HLS implementations of an LDPC decoder were developed and evaluated. Metrics such as latency, setup and hold times, maximum operating frequency, and resource utilization were compared.

Keywords: Low density parity check (LDPC) codes, Register Transfer Level (RTL), High Level Synthesis (HLS), HLS vs RTL, Comparative Analysis.

Acknowledgements

We would like to thank our supervisor Lars Svensson for his support throughout the project. During our weekly meetings, he offered valuable advice and guided us through practical implementation approaches. Furthermore, We would like to thank Xu Wang, Magnus Östgren, Erik Börjeson and Lucian Petrisor Ion for taking time to help us with theoretical discussions and practical implementation approaches.

Kavineshver Sivaraman Kathiresan, Prashanth Raj, Gothenburg, September 2025.

Contents

Glossary

ASIC	Application-Specific Integrated Circuit, a custom-designed chip for a specific application. 1
AWGN	Additive White Gaussian Noise . 15, 25
BER	Bit Error Rate. 15, 18, 25, 33, 39
BRAM	Block RAM, a type of memory block in FPGA devices used for high-speed memory storage and access, typically used for temporary data storage during computation. 12, 19, 20, 22–24, 27, 29, 31, 32, 37, 38
CCSDS	Consultative Committee for Space Data Systems. 2, 3, 7, 8, 15, 17, 23, 25
CN	Check nodes in the tanner graph for message passing. 9, 10, 23
DSP	Digital Signal Processing. 25, 29, 31, 32, 37
FEC	Forward Error Correction. 1, 6
FF	Flip Flop. 22, 32, 38
FPGA	A reconfigurable integrated circuit. 1, 12
FSM	Finite State Machine. 20, 39
HDL	Hardware Description Language. 10
HLS	High-Level Synthesis, a process that takes a high-level description of a design and converts it into hardware description language (HDL) code, typically RTL. ix, x, 1, 2, 5, 11, 12, 15–17, 22–25, 27, 31–34, 37, 38, 43
IP	Intellectual property. 17, 20, 24, 30
LDPC	Low Density Parity Check codes. 1–3, 5, 7, 8, 13, 15, 17, 25, 31, 43

LLR	Log Likelihood ratios for detemining the actual bits. 8–10, 15, 17, 18, 20–22, 24, 27, 29, 31–33
LUT	Look Up table. 17, 19–23, 27–30, 32, 38
RAM	Random Access Memory. 12
ROM	Read Only Memory. 20, 29
RTL	Register Transfer Level, a level of abstraction used to describe the operation of a digital circuit in terms of data transfers between registers. x, 1, 2, 5, 10–12, 15–17, 22–25, 27, 30, 33, 37, 38, 43
SNR	Signal to Noise Ratio. 15, 18, 25, 33, 35, 39
SPA	Sum Product Algorithm. 8, 15
VN	Variable nodes in the Tanner graph for message passing. 9, 10, 23

1

Introduction

Communication systems are fundamental to modern technological infrastructure enabling the exchange of information between individuals and devices. From simple user calls through mobile or complex data transfers through networks, communication is the backbone in various industries, including defense, transportation and scientific research. As technology evolves, communication systems are constantly being improved to handle high speed, reliability, secure transmission of data and error correction becomes an essential feature to ensure the data remain accurate, especially in challenging conditions [1].

In both space and telecommunication applications, long-distance communication channels are highly susceptible to noise, signal attenuation, and interference, making data transmission prone to errors. In space-based systems, such as satellites and interplanetary missions, the vast distances between transmitters and receivers further weaken signals as they travel through the atmosphere and space. Similarly, terrestrial long-distance telecommunication links face challenges due to channel impairments and limited bandwidth. The restricted power and bandwidth resources in both domains make it essential to optimize communication protocols for efficiency. To address these challenges, advanced error correction techniques are employed to enhance the robustness of communication systems over long distances and Forward Error Correction (FEC) is one of the effective methods. It plays a crucial role by adding redundancy into transmitted data, FEC enables receiver to detect and correct errors without the need for retransmission. This not only enhances the reliability of deep-space, satellite, and telecommunication links but also maximizes data throughput, which is critical for applications such as interplanetary missions, satellite telemetry, space-based imaging systems and long-haul telecommunication networks.

As communication systems evolve, so too must the methods used for error correction, particularly high speed applications like those in space. Low-Density Parity-Check (LDPC) codes have become increasingly popular in this applications in recent years, driven by advancements in hardware technologies. With the rise of complex FPGA and ASIC designs, LDPC codes leverage parallel computation for efficient error correction, making them a preferred choice in modern high-speed communication systems. Nowadays, HLS tools are used for the rapid prototyping of hardware design at the RTL. It is important to analyze how HLS designs differ in performance compared to traditional RTL designs as they often differ in performance from handcrafted RTL design. This thesis explores and analyzes the performance of HLS

versus RTL designs in the implementation of LDPC codes, focusing on potential improvements in HLS to enhance suitability of HLS for space and telecommunication applications.

1.1 Related Work

Research papers have demonstrated the implementation of LDPC codes using HLS. While the performance of HLS designs does not yet match that of RTL implementations, HLS remains a viable method for realizing functionality. The study in [2] evaluates the effectiveness of HLS for LDPC code implementation and compares its performance with RTL designs reported in other works. It concludes that HLS can achieve similar throughput to RTL designs but with increased resource utilization.

Additionally, the research in [3] investigates ways to enhance HLS implementation performance by modifying its high-level behavior. The requirements are derived from the Consultative Committee for Space Data Systems(CCSDS) documentation [4] [5], which serves as the foundation. Additionally, the LDPC decoder designed in RTL for space applications, as presented in [6] ,[7], and [8], along with the HLS-based implementations referenced in [9], and [10] serve as benchmarks for comparing performance across different configurations and algorithms. These works also demonstrate that it is feasible to implement LDPC decoding logic on a resource-constrained FPGA board, similar to the one used in this study.

We aim to design both RTL and HLS codes with same functionality for one to one comparison, and to identify where HLS loses performance in its RTL translation and explore how closely its performance can be brought to match RTL implementations.

1.2 Purpose and Goal

The primary objective of this thesis is to design an IP for LDPC codes with the specifications of space applications, implemented using both HLS and RTL. The performance of these implementations will be evaluated and compared against reference research works, as well as an analysis of HLS versus RTL performance. Specifically, a central goal is to explore the RTL implementation of HLS and analyze the factors contributing to performance differences, and try to bring the HLS performance as close to RTL performance by modifying HLS. The performance of our system is characterized by latency, bit error rate and resource utilization for a certain throughput.

The design is implemented on an Eclypse Z7 FPGA platform, which uses the ZYNQ XC7Z020 processor. The resources available on this may be insufficient for large-scale LDPC codes or high throughput applications as in [11]. Designing the implementation for the CCSDS specification on the Eclypse Z7 board is feasible with short block length (used for small data transmissions and emergency signaling in space missions). A similar implementation has also been demonstrated in [6] with code rate of 1/2 and short block length of 128 bits. One of the goals of this project is to explore the implementation of standardized block lengths, as used in deep space missions, on the Eclypse Z7 platform. Understanding the workflow of LDPC codes

and implementing them in accordance with CCSDS standards will be a challenging task. Additionally, optimizing the design to efficiently use memory and hardware resources while meeting stringent latency requirements presents a significant design hurdle.

1.3 Thesis Outline

This thesis is structured as follows: Chapter 1 presents the background and objectives of the work. Chapter 2 provides an overview of channel coding, with a focus on LDPC codes, including the algorithm description and the hardware platform employed. Chapter 3 outlines the implementation methodology and the software tools utilized. Chapter 4 details the design specifications of each implementation approach and the corresponding testing environment. Chapters 5 and 6 present the results obtained from each implementation, analyzing them in the context of the initial objectives. Finally, Chapter 7 concludes the thesis by summarizing the key findings.

2

Technical Background

In this chapter, we present the key concepts of digital communications and channel decoding, with focus on LDPC decoders. We then describe the process of generating the parity check matrix and the algorithm used to compute the final decoded bits. Finally, we outline the design flow in both RTL and HLS.

2.1 Digital Communication

Digital communication refers to the process of transmitting information using discrete signals of 0's and 1's over a physical medium such as cables, radio waves or optical fibers. This method is fundamental in modern communication systems and involves several key stages [12].

Source Encoding and Decoding : The source encoder compresses the information source by converting its bit sequence into a more efficient form with fewer bits. This can be lossless (e.g., computer files) or lossy (e.g., video, images, music). The source decoder reverses the encoding process, recovering the original sequence exactly for lossless compression or approximately for lossy compression.

Channel Encoder and Decoder: The channel encoder protects data from noise, distortion, and interference by introducing redundancy into the bitstream. It converts the input into a longer, structured sequence that enhances error resilience. The channel decoder aims to recover the original input by correcting errors introduced during transmission, using the redundancy added by the encoder.

Modulation and Demodulation: The modulator converts the encoded bitstream into a signal suitable for transmission over a specific channel. For wireless systems, this involves translating bits into high-frequency signals. The demodulator performs the inverse operation, recovering the original bitstream from the received signal.

Transmission Channel: The channel is the physical medium through which the modulated signal is transmitted or stored. It introduces noise, interference, and distortion, affecting signal integrity. Physically, the channel includes components like antennas, amplifiers, and filters in communication systems [13].

2.2 Channel Coding

Channel coding is a fundamental technique in digital communication used to improve the reliability of data transmission over noisy or unreliable channels. It introduces

structured redundancy into the transmitted message, allowing the receiver to detect and correct errors without needing retransmissions. The error control in digital communication can be done using FEC schemes. The error control in channel coding can be categorized into Automatic Repeat request (ARQ) and FEC. FEC codes are central to high-reliability systems like satellite communications and data storage. FEC codes can be linear or nonlinear, block codes or convolutional codes [14].

2.3 LDPC codes

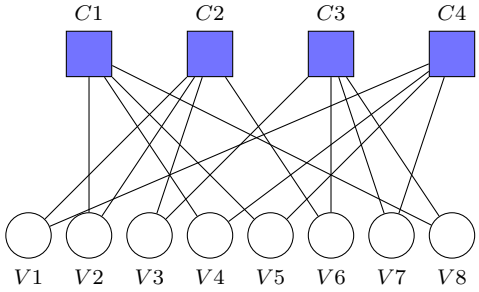
LDPC codes are a class of linear block codes known for their near-capacity performance across many data transmission and storage channels. Originally introduced by Gallager in his 1960 dissertation, LDPC codes were largely overlooked for decades until revived in the 1990s by researchers like MacKay and Luby.

A key contribution came from Tanner in 1981, who introduced Tanner graphs, a graphical way to represent LDPC codes via their sparse parity-check matrices. This sparsity allows efficient iterative decoding algorithms, which can achieve near-optimal error correction. This makes LDPC codes especially valuable in high-speed communication systems [15].

A binary (N,K) LDPC code is defined by its binary sparse parity-check matrix(H) of size (N-M x N), where N-K is the number of parity-check bits, N is the total codeword length and K is information bits and the code rate is defined as $R = K/N$, measures the efficiency of the encoding, as it represents the ratio of information bits to total number of bits in the codeword. The parity-check matrix H essentially defines the relationships between the information bits and parity-check bits, which are to form the complete codeword.

In the encoding process, the message is encoded using the parity-check matrix H, then the resulting encoded message is modulated and transmitted over the communication channel. The LDPC code is defined as the set of codewords that satisfy the equation $\mathbf{H}.\mathbf{c} = \mathbf{0}$, where '$c$' represents the encoded codeword. This equation ensures that the codeword satisfies the parity-check conditions defined by the matrix. The parity-check matrix has m rows and n columns, where each row and column contain only a small number of 1's. The sparsity of the matrix is crucial because it ensures that each parity-check equation involves only a small subset of the bits in the codeword, which is essential for efficient decoding. Moreover, the design of the matrix ensures that any two rows or columns share at most one 1 in the same position, which guarantees that the LDPC code has a well structured and robust framework for iterative decoding [12].

The above mentioned parity-check matrix is represented as Tanner graph as shown in figure 2.1. In Tanner graph N-K rows corresponds to check nodes represented as $C1$-$C4$ and N columns corresponds to the variable nodes represented from $V1$-$V8$. An edge is drawn between variable node i and check node j if and only if the corresponding element h_{ij}= 1 in the matrix H. The edges in the graph represent the relationships between the bits and their corresponding parity-check equations. This graph representation is particularly useful because it visualizes the structure of the code and aids in implementing efficient decoding algorithms like belief propagation.



C-nodes

$$C1 : 0 = V_2 + V_4 + V_5 + V_8$$
$$C2 : 0 = V_1 + V_2 + V_3 + V_6$$
$$C3 : 0 = V_3 + V_6 + V_7 + V_8$$
$$C4 : 0 = V_1 + V_4 + V_5 + V_7$$

V-nodes

Figure 2.1: Tanner graph and corresponding parity-check equations for an $(8, 4)$ code.

From the Tanner graph, we observe that each check nodes must satisfy its corresponding parity-check equation in order to ensure the correct decoding of the transmitted codeword. In the decoder, the received data is represented as a row vector $\hat{c}$ of length N. It is considered error-free only if it satisfies the parity-check equation $H\hat{c}^T = 0$. The parity check equation is obtained from parity check matrix, which is developed from a base sub matrix of size M specified in CCSDS.

2.4 CCSDS Standard

The CCSDS standard for LDPC codes refers to a specific set of error-correcting codes designed to improve the reliability of data transmission in space communications, where high bit error rates are common due to factors like noise and interference. It provides detailed information and structure for robust error correction methods to ensure reliable and efficient data transmission over long distances [4].

According to [5], the standard outlines the procedure for designing the parity check matrix H, as well as the method for constructing the permutation sub-matrices based on the block length and code rate for a given dimensional size.

2.4.1 Generation of the Parity-Check Matrix H

The matrix H is designed to be sparse, ensuring efficient iterative decoding. In equation 2.1, I_M represents $M\mathbf{x}M$ identity matrix and 0_M is the zero matrix. The permutation matrix Π_k is obtained from equation 2.2 [5].

$$H_{1/2} = \begin{bmatrix} 0_M & 0_M & I_M & 0_M & I_M \oplus \Pi_1 \\ I_M & I_M & 0_M & I_M & \Pi_2 \oplus \Pi_3 \oplus \Pi_4 \\ I_M & \Pi_5 \oplus \Pi_6 & O_M & \Pi_7 \oplus \Pi_8 & I_M \end{bmatrix} \tag{2.1}$$

Permutation matrix Π_k has non-zero entries in row i and column entries are defined by $\pi_k(i)$ for $i \in \{0, \ldots, M-1\}$:

$$\pi_k(i) = \frac{M}{4}\left[\left(\theta_k + \left\lfloor \frac{4i}{M} \right\rfloor\right) \bmod 4\right] + \left[\left(\phi_k\left(\left\lfloor \frac{4i}{M} \right\rfloor\right) + i\right) \bmod \frac{M}{4}\right] \tag{2.2}$$

where θ_k and $\phi_k(j)$ are defined as per the sub matrix table defined in CCSDS for the particular code rate and information block size.

Once the parity-check matrix is generated using the expression described in equation 2.1, it is used to verify the validity of the received codeword. If the received vector $\hat{c}$ does not satisfy the parity-check equation, it indicates that errors have occurred during transmission and thus error corrections is necessary. To correct these errors, the decoder applies an iterative belief propagation algorithm. The algorithm works by receiving log-likelihood values as input, which represent the likelihood of each bit being a 0 or 1. These values are then propagated as messages through the Tanner graph, passing from check nodes to variable nodes and vice versa. After a specified number of iterations, the decoder computes the final LLR values for each bit, which are then used to determine the most probable codeword. This process ensures that the transmitted message is reconstructed with high reliability, even in the presence of noise or interference.

2.5 LDPC Decoder

An LDPC decoder is the component responsible for recovering the original data from the encoded information. After encoding, redundancy is added to the transmitted bits so that errors introduced by noise in the communication channel can be corrected. After generating the parity-check matrix, the next step is to select an appropriate decoding algorithm.

2.5.1 Decoder Algorithm

The performance of the belief propagation algorithm can vary based on its implementation, careful consideration must be given to the algorithm's structure and computational requirements, especially for resource-constrained environments.

For this purpose, the sum-product algorithm(SPA) and the min-sum algorithm are two popular decoding methods for LDPC codes. These algorithms work by passing messages through the Tanner graph, iteratively updating the variable nodes based on the received log-likelihood values. While the sum-product algorithm offers the best error correction performance, it is more computationally expensive. On the other hand, the min-sum algorithm, a simplified version of SPA, provides a good balance between computational efficiency and decoding performance [16]. Here we have implemented the decoder with SPA.

2.5.2 Sum-product Algorithm

SPA offers the best performance in terms of error correction, but its computational complexity can be high, especially in resource-constrained environments such as space applications. However, SPA serves as an excellent starting point due to its ability to achieve near-optimal decoding. Given its performance, it was implemented in this work.

The SPA decoding process follows a series of iterative steps, where each iteration refines the estimation of the transmitted codeword, gradually towards the most likely transmitted message. The key steps of the algorithm are as follows [12]:

The initial message are calculate based on the received data(i.e., the received log-likelihood ratios L_j for each bit) and the parity-check matrix H.

1. Received LLRs:Each bit in the codeword is received with some noise, and the L_j is computed for each bit based on the received values. The LLR L_j is a measure of the reliability bit j, and it represents the logarithmic ratio of the probability that the bit is a 1 to the probability the bit is a 0.

$$L_j = \log\left(\frac{P(\hat{c}_j = 1)}{P(\hat{c}_j = 0)}\right) \tag{2.3}$$

2. **Initialization:** The next step is to send the initial messages from VNs to the CNs. Specifically, for each VN j connected to a CN i, the initial message L_{ji} is set to L_j, but as mentioned before only if the corresponding element in the parity-check matrix H is 1.

$$L_{ji} = L_j, \quad \text{if } h_{ij} = 1 \tag{2.4}$$

   here, h_{ij} is the element in the parity-check matrix H that indicates whether there is a connection between VN and CN. If h_{ij}, there is a connection, and the message L_{ij} is initialized to the received LLR value for that bit.

3. **Check Node Update:** After the initialization step, the decoder proceeds with the check node update phase. In this step, each CN updates its message to the connected VN based on the messages it has received in the previous iteration. The update equation used for this process is:

$$L_{ij} = 2 \cdot \tanh^{-1}\left(\prod_{j' \neq j} \tanh\left(\frac{1}{2} \cdot L_{j'i}\right)\right) \tag{2.5}$$

4. **Variable Node Update** : Once the check nodes have updated their messages, the next step is the Variable Node Update. In this step, the messages from the VN to the check nodes CN are updated based on the information received from the check nodes. This iterative process allows the variable nodes to refine their estimates of the transmitted bits based on both the received LLR and the messages from the check nodes.

$$L_{ji} = L_j + \sum_{i' \neq i} L_{i'j} \tag{2.6}$$

5. **Total LLR Computation:** After the VN update and CN update steps, the decoder computes the total LLR for each bit, which represents the final likelihood of the bit being 0 or 1. This final total LLR combines both the received information and the messages passed through the Tanner graph during the iterative decoding process.

$$L_j^{\text{total}} = L_j + \sum_i L_{ij} \tag{2.7}$$

here, L_{ij} is the message from CN to VN j, which represents the soft information passes during the decoding process from the CNs.

6. **Decision and Stopping Criterion:** The final step involves making a hard decision on each bit based on the total LLR and checking whether the decoded vector satisfies the parity-check condition.

$$\hat{v}_j = \begin{cases} 1, & \text{if } L_j^{\text{total}} < 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.8}$$

If the decoded vector $\hat{v}$ satisfied the parity-check condition:

$$\hat{v} \cdot H^T = 0$$

decoding is terminated. Otherwise, the algorithm repeats from Step 2 until the maximum number of iterations is reached.

2.6 RTL Design Concepts

RTL is a critical abstraction in digital hardware design, describing the flow of data between hardware registers and the logical operations performed on that data. RTL design serves as the foundation for implementing digital circuits on FPGA or ASIC. RTL models are typically written in Hardware Description Language (HDL) such as Verilog or VHDL. These languages allow precise control over timing, parallelism, and resource usage, making them ideal for performance-critical applications like LDPC decoder implementations in space and telecommunication systems.

2.6.1 RTL Design Flow

1. **Specification**: This is the first step in the RTL design flow. This step involves defining the exact algorithm or function that needs to be implemented in hardware. The goal is to establish what the system does at a functional level. In this case we define the specification of the decoding algorithm including how it operates on incoming data, what error-correction techniques are employed, and how the messages are passed between CN and VN.
2. **RTL Coding**: Once the specification is defined, the next step is to write the RTL code that captures the behavior of the system at the register level. In our case the code describes the flow of data between registers and the logical operations that are performed on that data.
3. **Simulation and Functional Verification**: After coding the RTL, the next step is simulation and functional verification to ensure that the design behaves correctly under various conditions.

4. **Synthesis**: Convert RTL code into gate-level netlists using tools like Xilinx Vivado, Intel Quartus, or Synopsys Design Compiler.
5. **Place-and-Route**: In this step, where the gate-level netlist is mapped onto physical layout of the chip. The goal is to place the logic gates and components onto fabric of an FPGA and to route the connections between them. Finally, they are physically mapped onto an FPGA or chip layout.
6. **Timing and Power Analysis**: The final step in the RTL design flow is to perform timing and power analysis. The goal is to evaluate the timing and power efficiency of the design to ensure that it meets the specified performance and energy consumption requirements [17] [18].

2.7 HLS Overview

HLS is an advanced design methodology that allows hardware engineers to describe digital systems using high-level programming languages such as C, C++, or SystemC. Instead of manually coding hardware at the RTL, designers write behavioral descriptions of the system, which HLS tools then translate into synthesizable RTL code.

HLS bridges the gap between software algorithm development and hardware design, making it particularly attractive for complex digital signal processing tasks such as LDPC encoding and decoding.

2.7.1 HLS Design Flow

1. **Algorithm Description**

   The goal is to describe the system's algorithm in a way that is independent of the hardware implementation. So in our case we write steps for the decoding using C.

2. **Directive Insertion**

   Once the high-level description is written, in the next step we tell the HLS tool on how to optimize the design for hardware synthesis like loop unrolling, pipelining, memory partitioning and latency control [2].

3. **Synthesis to RTL**

   The HLS compiler translates the high-level code into RTL (Verilog/VHDL), optimizing based on performance, resource, and area constraints.

4. **Verification and Co-simulation**

   The generated RTL will be co-simulated against the original C/C++ code to ensure functional equivalence.

5. **Implementation and Testing**

   The final RTL will be further synthesized and implemented on FPGA platforms using standard EDA tools [19].

2.8 FPGA Eclypse Z7

The Eclypse Z7 board is development board from Digilent, designed for high speed, real time embedded systems. The core of the board consists of Xilinx Zynq-7000 SoC, which has a dual-core ARM Cortex-A9 processor(PS) with an Artix-7 FPGA(PL) fabric on a single chip and they communicate primarily via the AXI(Advanced extensible Interface) interconnect. It has 1GB DDR3 RAM for the processor. Additionally, it supports a wide range of connectivity options, including but not limited to Ethernet, USB, and VGA capabilities, which expand the potential for various applications without additional hardware. It is fully compatible with the Xilinx Vivado Design Suite to facilitate development and maximize the board's capabilities [20].

2.8.1 Block Random Access Memory (BRAM)

RAM is a type of storage that allows both reading and writing of data. By providing an address and the data, you can write to a specified location in memory. Similarly, by providing just the address, you can read the stored data from that location.

BRAM, a type of RAM embedded within an FPGA, is an important storage resource in FPGA-based designs. As the name suggests, BRAM is a contiguous block of embedded memory resources that are fixed within the FPGA architecture. This dedicated storage unit is designed to offer fast data access by directly performing read and write operations without needing complex buses or external controllers [21].

Unlike DRAM (Dynamic RAM), which uses combinatorial logic for output, BRAM uses sequential logic for output. This means that when performing read operations, the address provided passes through internal registers, and as a result, BRAM requires a clock signal to retrieve the output data.

2.9 HLS Directives

HLS directives are compiler instructions that guide the HLS tool in how to translate the code into optimized RTL. These directives, applied as HLS pragmas used to optimize the design by reducing latency, improving performance and minimizing area and resource utilization. However, they must be applied carefully as some are interdependent and may affect each other's impact. While many pragmas are available in HLS, this design specially employs *PIPELINE*, *UNROLL*, *ARRAY_PARTITION*, *ARRAY_RESHAPE* to achieve performance improvements [22].

2.10 Performance Metrics

In communication systems, the effectiveness of data transmission and error correction techniques is typically assessed using several performance evaluation metrics.

The most important factors among these are BER vs SNR, the average number of iterations to converge, throughput, and latency [1].

- **SNR**: It measures the strength of the signal relative to background noise, expressed in decibels (dB). A higher SNR indicates a cleaner channel with fewer errors, while a lower SNR corresponds to noisier conditions.
- **BER**: It is defined as the ratio of the number of incorrectly decoded bits to the total number of transmitted bits. It is a key measure of the decoder's accuracy and reliability.
- **Iterations to Decode**: In iterative decoding algorithms such as the SPA, the number of iterations required for the decoder to converge (reach zero errors or a valid codeword) indicates the efficiency of the design. Fewer iterations reflect faster convergence and lower decoding latency.
- **Throughput**: It refers to the amount of data that the LDPC decoder can process per unit time, typically measured in bits per second (bps). Throughput depends on clock frequency, parallelism, and memory access efficiency.
- **Latency**: It is the total time required to decode a message from input to output, influenced by the number of iterations and processing speed.

By analyzing the BER versus SNR plots and tracking the average iterations needed to decode, the performance and robustness of different decoder implementations can be effectively compared across varying SNRs. Evaluating throughput and latency alongside BER and SNR provides a comprehensive understanding of the decoder's performance and its suitability for practical communication systems.

3

Methods

To evaluate the performance of LDPC decoder in RTL and HLS, the initial stage involved establishing a thorough understanding of channel coding principles and implementation based on CCSDS standards. This chapter explains the methodology, detailing the tools used for developing and testing the design for various scenarios.

3.1 MATLAB Implementation and Validation

The workflow begins with the implementation of LDPC decoder using the SPA algorithm in MATLAB. The encoder was implemented using the built-in function with 1024 random bits and text files. To simulate the effects of the transmission channel, the encoded bits were subjected to AWGN at varying SNR levels. For the decoder, LLR values were generated from the received signal and provided as inputs to the decoding design for processing. The decoded data was compared with the input random bits for different SNR and iterations. The BER across different SNR values and iterations were found and used for later comparison. This model provided a clear understanding of the decoding process and served as a reference for verifying correctness through simulation.

3.2 RTL and HLS Design

After validating the MATLAB model, the design was developed in both RTL and HLS implementations. Key components, including the sparse parity-check matrix generation, implementation of check node and variable node updates, total LLR calculation, decision making, and iterative decoding operations were developed as separate modules. This modular approach supports easier functional testing and integration.

3.3 Functional Verification

Test benches are created to verify the functionality of each RTL and HLS module. Outputs from these implementations are compared with results from the MATLAB model to ensure consistency. This step helps identify and resolve discrepancies early in the design process.

3.4 Synthesis and Optimization

Once verified, both RTL and HLS designs were checked for synthesizability using synthesis tools. Optimization was applied to improve design efficiency, reduce area, and enhance timing.

3.5 Comparison of RTL and HLS Designs

In the final stage, the RTL and HLS designs were compared in terms of performance, utilization, and throughput. The validated MATLAB model serves as the baseline for evaluating functional accuracy, while hardware metrics help assess efficiency and scalability.

3.6 Tools Used

To support the design and evaluation of the LDPC decoder, different tools were employed at various stages of development. Table 3.1 summarizes the primary tools, their purposes, and the languages used.

Table 3.1: Tools used

Tools	Purpose	Language
Matlab	Prototype	Matlab
Vitis	HLS development	C
Vivado	RTL Implementation	Verilog

4

Design

The design and implementation of both RTL and HLS are detailed in this chapter. Developing an LDPC design in RTL and HLS requires careful analysis of multiple factors in both the HLS tool and Vivado, including the trade-offs between fixed-point and floating-point representations, the use of inbuilt IPs versus LUT-based computations, the precision of LUTs, and overall resource utilization. These choices significantly impact the algorithm's behavior, memory usage, and timing constraints, ultimately influencing the overall performance of the design.

4.1 MATLAB Development

The MATLAB environment was chosen as the initial platform to prototype and validate the LDPC decoding algorithm due to its powerful matrix operations and ease of simulation. This phase served as a crucial baseline to ensure the functional corrtectness of the algorithm prior to hardware implementation.

4.1.1 Parity Check Matrix Generation

A parity-check matrix conforming to the CCSDS LDPC standard was first generated using MATLAB. The structure and sparsity of the matrix were carefully designed to reflect the CCSDS-defined quasi-cyclic LDPC format, which is optimized for high performance in space communications. This matrix defines the constraints that valid codewords must satisfy, forming the backbone of the decoding process.

4.1.2 Channel Simulation and Pre-processing

To simulate realistic communication conditions, binary input data was generated and modulated, followed by the addition of AWGN to emulate a noisy space channel. The modulated signal passed through the noisy channel and the LLR for each received bit was calculated using the noise variance. The LLRs, originally represented in floating-point, were initially rounded to fixed-point values to simplify hardware implementation. However, since this led to degraded performance, all operations were ultimately carried out in half-precision floating-point (16-bit) representation. A similar implementation approach was applied in both RTL and HLS.

4.1.3 Sum-Product Algorithm

The Sum-Product Algorithm was implemented with a maximum iteration limit of 20. The decoding process continues either until the parity-check condition is satisfied or the maximum number of iterations is reached.

4.1.4 Performance Evaluation

The MATLAB-based decoder was tested across a range of SNR values. The performance was evaluated in terms of BER, and the results were plotted to assess the robustness of the decoder under varying channel conditions.

This MATLAB implementation provided a vital reference model, enabling clear functional expectations before translating the design into RTL and HLS for hardware synthesis and performance comparison.

4.2 RTL Implementation

Following the functional validation of the LDPC decoder in MATLAB, the next phase involved translating the algorithm into a RTL design using Verilog HDL [23]. This step aimed to create a synthesizable and efficient hardware description of the decoder suitable for implementation on our FPGA, while preserving the algorithm's behavior and optimizing for speed, and resource utilization.

The RTL implementation began by breaking down the SPA into modular building blocks. Each block was mapped to a specific functional unit:

- **Sparse Parity Matrix Generation**: Logic to check 1's in the specific position in parity matrix.
- **LLR Initialization Module**: Responsible for storing and initializing the channel LLRs.
- **Check Node Unit (CNU)**: To implement tanh-based calculations, using a lookup table.
- **Variable Node Unit (VNU)**: To aggregate incoming messages and update bit estimates.
- **Decision Unit**: Compute the hard decision and verify if the decoded output satisfies the parity-check equation.
- **Iteration Control Unit**: Manage the scheduling and number of iterations for the decoding loop.

This modular approach enabled independent testing and verification of each component before integration into a full decoder architecture.

4.2.1 Overview

Figure 4.1 illustrates the block diagram of the SPA top module. The module receives LLR data as input and produces the decoded bit sequence as output. Upon receiving

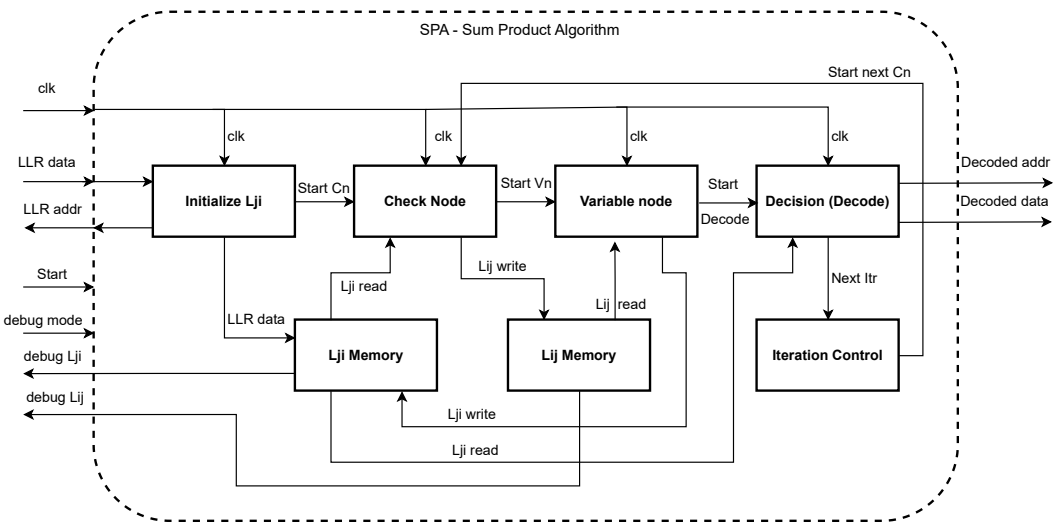


Figure 4.1: Block Diagram of SPA top module

the input data, the top module initializes the L_{ji} memory, which is implemented using BRAM. After initialization, it issues a start command to the check node sub module.

The top module also manages the iteration counter, repeatedly executing the decoding process until the maximum iteration limit is reached. Two key BRAM blocks, L_{ij} and L_{ji}, are continuously accessed and updated by the Check Node, Variable Node, and Decision units during each iteration. Additionally, a debug mode is implemented, when enabled (debug = 1), all processing operations are paused, and data retrieval from memory is allowed for inspection. The following subsections describe the functionality of these sub modules in detail.

4.2.2 Spare Parity Check Matrix

Storing the complete parity-check matrix H of size 1536×2560 directly would consume significant hardware resources. Initially, a combinational logic approach was considered to compute the sparse matrix on the fly. However, this method required extensive computations to identify the positions of 1's in each row and column, leading to high resource usage.

To optimize resource utilization, the sparse parity-check matrix was implemented using two separate arrays: one storing the column indices of the 1's for each row, and another storing the row indices of the 1's for each column. Since the number of 1's varies per row and per column, two additional arrays were used to store the count of 1's per row and per column, respectively.

This information allows efficient reconstruction of the sparse matrix during decoding without storing the full H matrix explicitly. The initial implementation used LUTs for storage, but to reduce resource consumption, these arrays were later migrated to BRAM's.

4.2.3 Check Node

The data format used in this design is half-precision floating-point (16-bit). Upon system initialization, the L_{ji} matrix is loaded with the external LLR data. Once this initialization is complete, the check node processing begins.

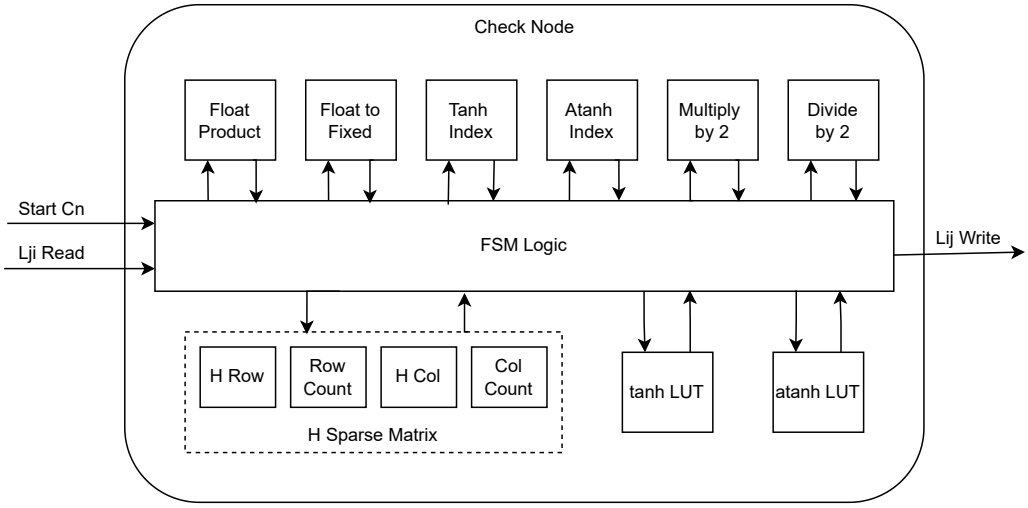


Figure 4.2: Block Diagram of check node

Figure 4.2 illustrates the block design of the check node. At the center lies the FSM logic, which controls the overall data flow. The modules at the bottom represent memory units, where the H matrix, along with the tanh and atanh lookup tables, are stored in BRAM's. The modules at the top perform the arithmetic operations directed by the FSM. The check node begins its operation upon receiving the *Start Cn* signal, processes the incoming L_{ji} data, and updates the corresponding L_{ij} values.

The check node unit computes the messages passed from check node to variable node. It uses the sparse parity-check matrix described above, accessing each row to find the indices of connected variable nodes. For each connected variable node j, the other connected variable nodes in the same row are identified. The corresponding L_{ji} values are fetched from BRAM, and the hyperbolic tangent function (tanh) is applied to half of each L_{ji} value.

The product of these tanh values (excluding the current node j) is computed, and its value is clamped within the range to avoid overflow issues in the subsequent atanh computation. The output of the atanh function, scaled by 2, updates the L_{ij} matrix for that node. Figure 4.3 shows the complete flow of data through the FSM logic with a state flow diagram.

Initially, the built-in IP core was used to compute the atanh function. However, this IP had a maximum output limit of 1.2, which was later found to critically affect decoder performance, the decoder failed to converge entirely. To address this, LUTs were implemented for the atanh function. Both the tanh and atanh functions were originally stored as LUTs in ROM, but these were later migrated to BRAMs to optimize resource usage.

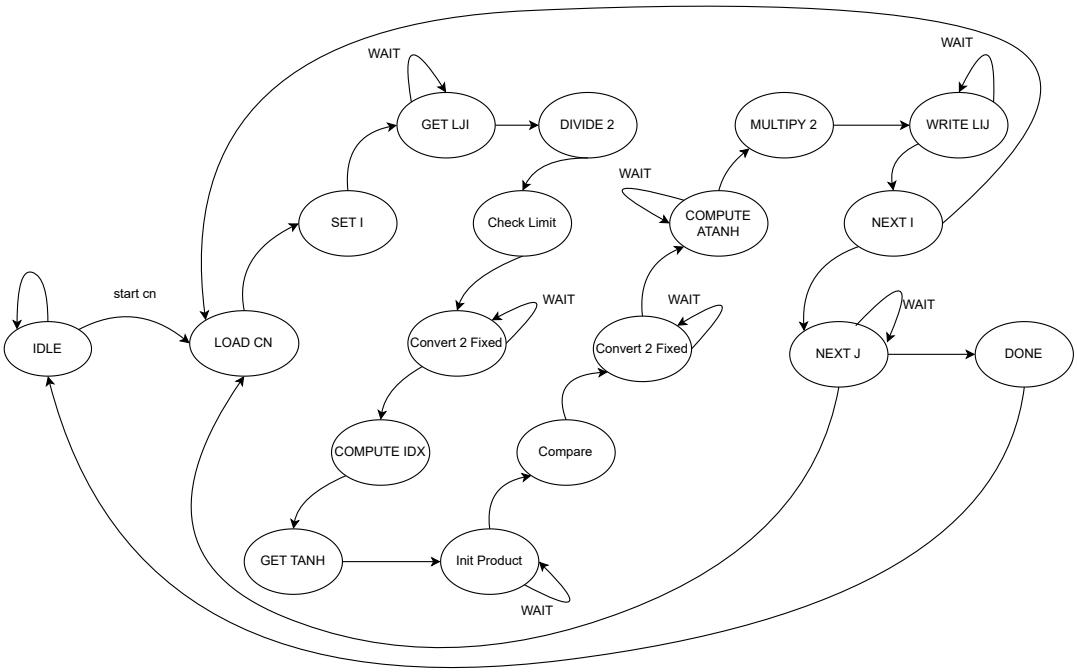


Figure 4.3: Simplified State Flow diagram of check node FSM

It was observed that the precision of the tanh and atanh functions significantly impacts decoder performance. The tanh LUT is implemented with a precision of 0.01, while the atanh LUT has a finer precision of 0.005. Insufficient precision in these functions leads to failure of decoder convergence.

4.2.4 Variable Node

The variable node unit updates messages passed from variable nodes to check nodes during each iteration of the decoding process. In this implementation, the variable node computation begins only after the check node has completed its updates. For each variable node j, all connected check nodes are identified using the sparse parity-check matrix, similar to the approach used in the check node unit.

For each connected check node i, the variable node aggregates the incoming messages L_{ij} from all other check nodes connected to variable node j, excluding the message from check node i itself. This aggregated sum is then added to the initial channel LLR value to compute the updated message L_{ji}. This updated value is stored back in memory and used in subsequent iterations.

4.2.5 Decision Unit

The total LLR for each variable node j is calculated by summing its initial LLR value with all incoming messages L_{ij} from connected check node. This total LLR is then used to make a hard decision on the decoded bit, if the total LLR is negative, the bit is decoded as 1, otherwise, it is decoded as 0.

4.3 Synthesis and Analysis

Post-verification, the RTL design was synthesized using Xilinx Vivado. Key performance metrics such as LUT and FF utilization, clock frequency, latency, power consumption were extracted and used as a basis for comparison with the HLS implementation.

4.4 HLS Implementation

After verifying the functionality in MATLAB, we designed the decoder logic using HLS. The implementation is performed using Xilinx Vivado HLS, a widely used tool for generating RTL from high-level C/C++ descriptions. The algorithm is designed using C functions [24], with a focus on preserving both the algorithmic fidelity and modularity. The top-level functions for synthesis will resemble the modules in RTL.

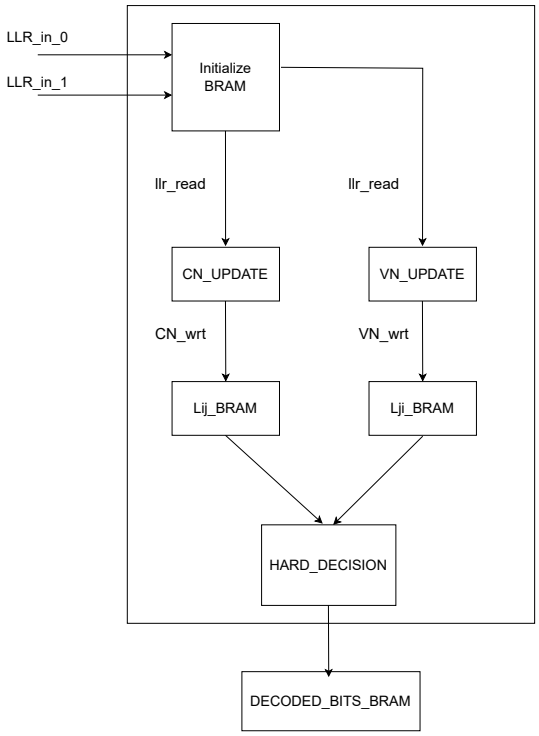


Figure 4.4: HLS Block Diagram

The above block diagram in figure 4.4 illustrates the data flow in HLS- based LDPC decoder. Initially, the LLR values are loaded into on-chip BRAM. The check-node and variable-node update blocks iteratively process these values, performing the required tanh and atanh to compute updated messages. After each computation, messages from check-node updates are stored in L_{ij}, while variable-node are stored in L_{ji}. These messages are passed between check-node and variable-node processing

stages for the configured number of iterations. Finally, *hard_decision* converts the values into decoded bits and are stored in *decoder_bram*.

In the Vitis environment, fixed-point data types such as *ap_fixed* and *ap_int* were used to balance numerical precision with hardware resource efficiency. The design was structured with clear modular functions, where each top-level function is synthesized into a corresponding RTL module. To improve performance, various HLS directives such as *HLS PIPELINE*, *HLS UNROLL*, and *HLS ARRAY_PARTITION* were applied to optimize loop execution, resource usage and memory access. Initially, only *HLS_PIPELINE* directive was applied to evaluate whether the design could successfully pipeline all functions without any stalling. Initially the design was synthesized without any *pragmas* and resulted in high slack and resource used. So, alternatively many *pragmas* were used in different combinations and with varying parameter setting, to explore their impact on the overall synthesis results and performance that are reflected in the Chapter 5.

As mentioned before, the parity-check matrix defined by the CCSDS standard forms the foundation of the entire algorithm and enables efficient iterative decoding. This matrix is generated in MATLAB and for decoding process, it is split into two different matrices *CN_node* and *VN_node* , which represent the sparse forms of the check nodes and variable nodes respectively. They both are stored separately in BRAM so that they can be accessed independently during the process.

Algorithm 1 LDPC Message Update Using HLS Math Functions

function UpdateMessage(iter, j2, col, i, k)
  $msg \leftarrow$ **if** $iter = 0$ **then** $llr\_in[j2]$ **else** $Lji[j2][col]$
  $half\_msg \leftarrow msg \times 0.5$
  $msg\_f \leftarrow$ `float`(msg)
  $tanh\_res \leftarrow$ `hls::tanh`$(0.5 \times msg\_f)$
  $prod \leftarrow prod \times$ `fixp_t`$(tanh\_res)$
  $prod\_f \leftarrow$ `float`$(prod)$
  if $prod\_f > 0.999$ **then**
    $prod\_f \leftarrow 0.999$
  end if
  if $prod\_f < -0.999$ **then**
    $prod\_f \leftarrow -0.999$
  end if
  $Lij[i][k] \leftarrow 2 \times$ `hls::atanh`$(prod\_f)$
end function

In the above Algorithm 1, the computation of messages exchanged between the CN and VN involves computationally intensive evaluations of the tanh and atanh functions. The design employs the *ap_fixed* data type, whereas the built-in tanh and atanh functions in HLS operate on *float* data, necessitating conversion from fixed-point to floating-point and subsequent reconversion to fixed-point. Alternative implementations based on tanh and atanh LUTs were designed to eliminate these conversions. These lookup tables approaches precomputed function values stored

in BRAM, which are accessed using input indices during every check node and variable node update. By replacing complex built-in functions with simple memory lookup, shortens the critical path and improves timing. As shown in Chapter 5, this approach achieves better timing and more efficient resource utilization compared to the built-in function approach.

The LLR values obtained from MATLAB and are stored in BRAM from which they are accessed in HLS for further processing.

In the next step, we design the check node and variable node updates which are interdependent. These updates are stored in local BRAM's and are not exposed externally as declaring them as ports would increase resource usage and complicate the design in Vivado.

4.5 Testing and Functional verification

After completing the design, a test bench was developed in HLS to verify the functionality of the decoder design. The test bench simulates the design using random input bit streams generated in MATLAB to ensure behavioral correctness. The test bench also takes in number of iterations to be performed and outputs the final decoded bits. Following functional verification, the design is synthesized, co-simulated, and exported as a synthesizable RTL IP core.

The exported IP is then integrated into Vivado block design. In this stage, the IP's exposed interfaces are connected to other system components to form a complete design. A Vivado test bench is also created, which drives signals such as address, enable and write controls to the IP core. These signals are used to read LLR data from a text file and write it into the appropriate addresses in BRAM and used in the decoder design.

5

Results

This chapter presents the results of the LDPC decoder design implementation across MATLAB, RTL and HLS. It includes comparison and analysis of the decoder's performance between design choices and hardware metrics at various stages of the design flow. The parity-check matrix, along with the check node and variable node updates, is generated in both MATLAB and RTL implementations, and the results are compared across different stages and iterations to ensure the functional correctness of the code. BER versus SNR plots are evaluated to measure the decoding accuracy and to validate the functional equivalence of the high-level MATLAB model and the synthesized hardware implementation. The chapter explores key performance metrics such as timing, power, throughput, latency and clock frequency, obtained from the HLS and RTL reports. Resource utilization details—including usage of LUTs, FFs, BRAMs and DSP blocks—are also analyzed to assess the hardware efficiency of the design.

5.1 Matlab

As part of the LDPC code implementation in MATLAB, the parity-check matrix H was generated based on the logic defined in the CCSDS standard, with an information bit length of k=1024 and a code rate of 1/2. Figure 5.1 below shows the graphical representation of the generated H matrix. This was visually compared with the expected matrix structure defined in the CCSDS standard, and the bit positions were cross-verified to ensure correctness.

The encoding process used this H matrix to encode the information bits, followed by Binary Phase Shift Keying (BPSK) modulation. The modulated signal was then modified, with AWGN noise added based on the specified SNR value. The resulting noisy signal was treated as the received input for the LDPC decoder. LLRs were computed from the received signal and fed into the SPA decoder. At each iteration, the number of bit errors was computed to evaluate decoder performance.

The BER vs. SNR graph in Figure 5.2 illustrates the decoding performance of the LDPC code using the SPA Algorithm. The evaluation was conducted over 100 frames, each containing 1024 bits, across multiple SNR values. The maximum number of decoding iterations was limited to 20, within which the decoder converged to zero errors for all 100 frames when SNR $\geq$ 3 dB and most of frames when SNR $\geq$ 2, for few iterations when SNR = 2, it converges before 25. The LDPC decoder converges to zero errors for all tested SNR values above 1 dB. To highlight

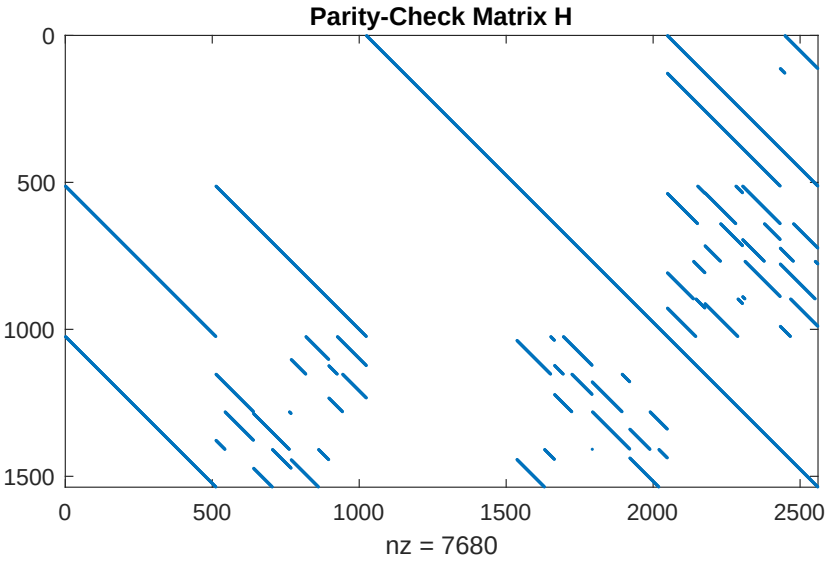


Figure 5.1: Parity Matrix

early convergence behavior and better observe the waterfall region, the BER after the 5th iteration was plotted for the different SNR points.

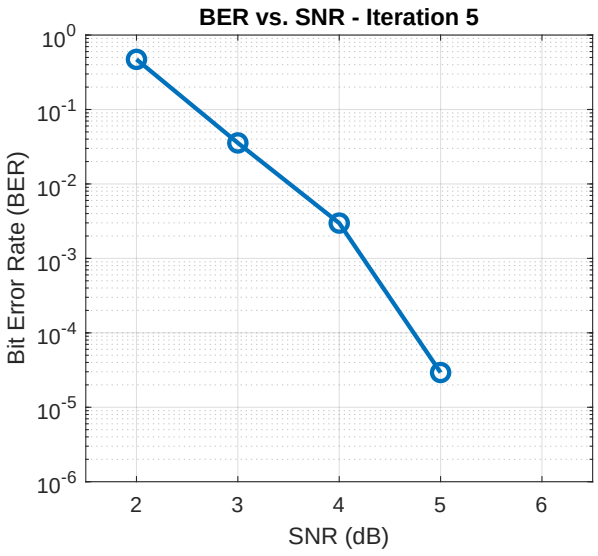


Figure 5.2: BER vs SNR at Iteration 5

In Figure 5.3, the number of iterations required for the decoder to converge to zero bit errors in average is shown for varying SNR values. The decoder fails to converge at SNR = 1 dB, indicating that the decoding threshold lies at or just above this value. For all SNR values greater than 1 dB, the decoder successfully converges

within a limited number of iterations.

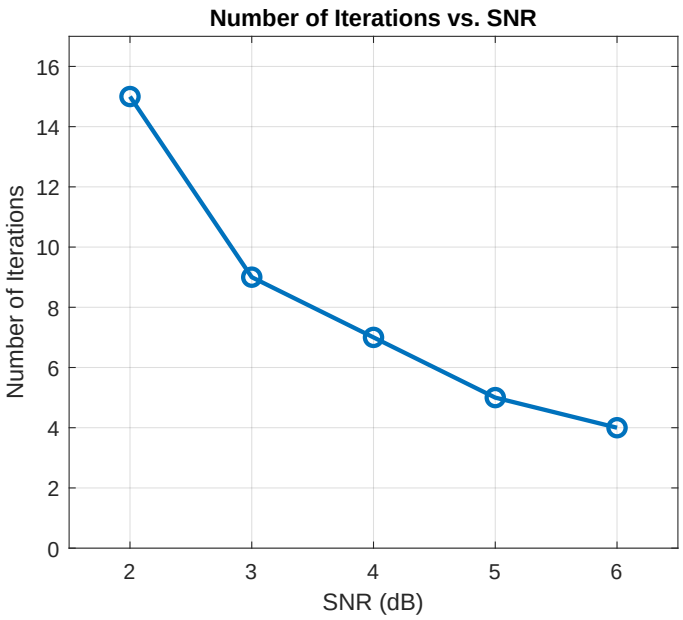


Figure 5.3: Number of Iterations to Converge Vs SNR

5.1.1 LLR calculation

The LLR values required for feeding into the decoder design for both RTL and HLS are generated in MATLAB and stored in text file. They are stored in as header files in HLS and during computation they are stored in 2 port BRAM for parallel access. In the RTL implementation, the LLR values are loaded into the design through the test bench and stored in LUTs, from which they are sequentially supplied to the SPA algorithm.

5.2 RTL

5.2.1 Sparse Parity Matrix Generation

The sparse parity matrix generation logic was implemented in RTL and verified using a test bench, which outputs the sparse matrix to a text file. Figure 5.4 matrix shown here is based on the sparse matrix output generated by the RTL implementation, verifying that the RTL logic produces the correct data format and structure.

5.2.2 SPA algorithm

In the SPA implementation, each submodule was first verified independently using a dedicated test bench, with waveform inspection in simulation to ensure that the expected values were produced at each processing stage.

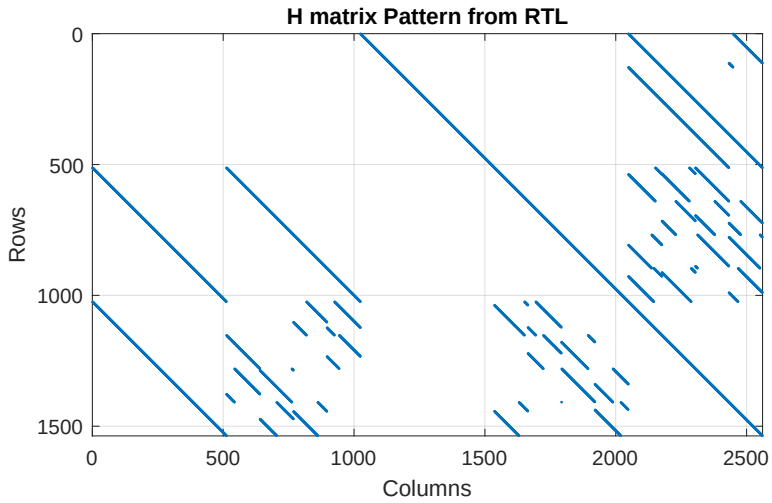


Figure 5.4: H matrix design in RTL verification

After integrating all submodules, the debug mode was employed to capture the contents of the L_{ji} and L_{ij} matrices at runtime, immediately after the check node and variable node updates in each iteration. These captured values were saved to a text file and compared against the corresponding L_{ji} and L_{ij} memories generated by the MATLAB reference model at each iteration.

The comparison revealed small numerical differences, primarily due to the approximations introduced by the LUT-based implementations of the tanh and atanh functions in the RTL design. Despite these discrepancies, the decoder output matched MATLAB in terms of overall decoding success, with both achieving correct results in the same number of iterations. In certain cases, the RTL version decoded some bits slower or faster than MATLAB, attributed to approximations in the LUTs.

To evaluate the numerical differences, the absolute error between the RTL and MATLAB values of L_{ji} and L_{ij} was calculated for all iterations. Table 5.1 presents the results for a representative decoding run at SNR = 3 dB during iteration 6, where the MATLAB model reduced the total number of bit errors from 74 to 3, while the RTL implementation reduced them from 74 to 4 for the same input data. In the following iteration, both implementations achieved zero bit errors, indicating successful decoding despite the minor numerical differences caused by LUT approximations.

Input (raw):

```
Lgtr fesiÇî LdPS"$åcmdeR¡èmbd!èope!éõ Qïrkc$%a @ ...
```

Decoded output:

```
Lets design LDPC decoder here Hope it works!!!
```

Decoded output at SNR = 3, Iteration 7 in both MATLAB and RTL

The tanh lookup table was approximated with a precision of 0.01, while the atanh lookup table was approximated with a finer precision of 0.005. Although the observed values differ slightly from the MATLAB reference, the overall data remains

Table 5.1: Numerical difference between RTL and MATLAB SPA values

L_{ji}			L_{ij}		
MATLAB	RTL	Difference	MATLAB	RTL	Difference
-22.04	-21.62	0.42	-6.50	-6.90	0.40
-27.91	-28.98	-1.07	-0.16	-0.24	0.25
26.86	27.71	-0.85	6.50	6.70	0.20
19.71	19.31	0.40	-7.62	-6.90	0.71
17.35	17.56	-0.21	4.80	4.67	0.14

sufficiently similar to be considered acceptable. With this precision range, the approximations have minimal impact on the functional correctness of the decoder.

However, it is crucial to maintain as fine a precision as possible. It was observed that increasing the precision beyond a certain point can cause the decoder to fail to converge altogether, while using a coarser precision results in slower convergence and longer decoding times. Therefore, a balance must be struck between precision and decoder performance.

5.2.3 Resource Utilization

Table 5.2 summarizes the overall FPGA resource utilization for the decoder design, targeted for a Xilinx Zynq-7000 device. The percentage values are calculated with respect to the available resources on the target FPGA.

During initial synthesis, implementing certain LUTs as ROMs resulted in LUT and flip-flop utilization exceeding 200% and 300%, respectively. To address this, storage structures such as L_{ji}, L_{ij}, and the H matrix were moved to BRAM. This optimization reduced logic utilization to acceptable levels, with BRAM usage at approximately 29%.

In the current design, the LLR computation step in section 2.5.2 still uses a LUT-based RAM. Replacing this with BRAM could further reduce LUT and FF utilization. DSP blocks are employed for half-precision floating-point multiplication and addition operations.

Table 5.2: RTL Resource Utilization Report

Resource	**Usage**	**Available**	**Utilization (%)**
LUT	18690	53200	35.13
LUTRAM	35	17400	0.20
FF	46738	106400	43.93
BRAM	40	140	28.57
DSP	11	220	5.00
IO	53	200	26.50
BUFG	2	32	6.25

5.2.4 Timing Report

Table 5.3 for a clock period of 10 ns (100 MHz), the design achieved a setup slack of 0.249 ns and a hold slack of 0.039 ns, indicating comfortable timing margins. The maximum achievable operating frequency was found to be approximately 121 MHz (period = 8.2 ns), at which the setup slack decreased to 0.015 ns. Reducing the clock period beyond this point resulted in negative setup slack, causing timing violations. For comparison with the HLS in-built function implementation, timing was also analyzed at 85 MHz (period = 11.75 ns), providing a more relaxed timing environment and larger margins.

Table 5.3: RTL timing Report

Category	**Worst Slack (ns)**		
Clk Frequency	**85 MHz**	**100 MHz**	**121 MHz**
Setup	0.251	0.249	0.015
Hold	0.052	0.039	0.034

5.2.5 Power Analysis

Table 5.4 summarizes the static and dynamic power consumption of the RTL decoder at different clock frequencies. As expected, dynamic power increases with frequency, since it is directly proportional to the number of switching events per second in the logic and routing resources.

$$P_{sw} = f{V_{DD}}^2 \sum_{i=1}^{N} (C_i \alpha_i) \tag{5.1}$$

In contrast, static power remains nearly constant, as it is primarily determined by leakage currents, which are independent of operating frequency.

Table 5.4: RTL Power Analysis

	Power (Watts)		
Clock Frequency	**85 MHz**	**100 MHz**	**121 MHz**
Dynamic Power	0.098	0.114	0.140
Static Power	0.111	0.112	0.112
Total On-chip power	0.209	0.226	0.252

5.3 HLS

5.3.1 Power Analysis

This section presents the power estimation for the exported IP for both built-in and LUT based implementation. We observe that the dynamic power accounting

for 54% and static device power accounting for the remaining 46%. Within the dynamic component, BRAM activity is the largest contributor at 37% followed by DSP usage at 20%, signal switching at 16% clock at 10% and I/O at 10%. The relatively high BRAM dynamic power arises from frequent memory access in the LDPC decoder's check-node and variable-node update stages, where multiple memory banks are active every clock cycle due to *ARRAY_PARTITION* and *ARRAY_RESHAPE* optimizations. DSP dynamic power primarily results from fixed-point arithmetic operations in the tanh and atanh computations, which execute in parallel across several processing elements created by *UNROLL* pragmas. Signal and clock power contributions reflect the switching activity in the interconnect fabric and distribution network required to drive these parallel data paths. I/O power stems from reading LLR inputs, writing decoded outputs, and handling control signals. Overall, the power profile reflects a design optimized for high throughput via parallelism, where the trade-off is increased dynamic power in memory, DSP, and interconnect resources.

For built-in functions we have clocks alone accounting for 31%. This elevated clock power is from the extensive use of loop pipelining and unrolling in the check-node and variable-node update stages, which significantly increase register usage and switching activity across the clock network. *HLS UNROLL* instantiates multiple parallel computation units, all toggling each clock cycle and *HLS ARRAY_PARTITION* breaks large memories like L_{ji} and L_{ij} into multiple smaller memories or registers, increasing the number of clocked elements. Full parallelism in message passing between check node and variable node creates wide buses with high switching activity and finally frequent BRAM and register access every clock cycle keeps both memory blocks and interconnect active. All these factors increase the switching activity of logic signals and especially the clock network, leading to the observed high clock power share and overall high dynamic power.

Table 5.5: HLS Power Analysis

	Built-in Function (W)	**LUT Function (W)**
Clock Frequency	**85 MHz**	**100 MHz**
Dynamic Power	0.227	0.129
Static Power	0.111	0.109
Total On-chip Power	0.338	0.238

5.3.2 Timing report

Initially, the HLS implementation employed the built-in tanh and atanh functions. This approach failed to meet a 10 ns timing constraint due to the overhead of fixed-point to floating-point and floating-point to fixed-point conversions inherent in these function calls. Furthermore, the use of *ARRAY_RESHAPE* pragmas in this version resulted in wide memory accesses, which increased routing complexity and further degraded timing performance. As a result, the design required a relaxed clock period of 11.80 ns to achieve timing closure, with the corresponding results summarized in table 5.6.

To address these issues, the built-in functions were replaced with LUT-based approximations, using precision steps of 0.01 for tanh and 0.005 for atanh. This eliminated the need for type conversions and reduced arithmetic complexity, thereby shortening the critical path. With this optimization, the design successfully met the 10 ns timing constraint, as reflected in the updated timing report in table 5.6. The timing report indicates that the design meets the required constraints, with no negative slack observed.

Table 5.6: Timing Analysis Summary

Category	Built-in Function Worst Neg Slack (ns)	LUT Function Worst Neg Slack (ns)
Setup	0.428	0.966
Hold	0.014	0.049

5.3.3 Resource Analysis

Table 5.7 presents the detailed resource utilization for the HLS-based implementation.

LUT usage stands at 15.37% of the available capacity, which, while not saturating the device, is still a notable share. This demand mainly comes from the arithmetic logic required for fixed-point tanh and atanh computations. Additionally, the use of *UNROLL* pragmas replicates computational units to execute multiple loop iterations in parallel, which further increases LUT consumption.

FFs consume only 3.99% of the total, but they play a critical role in holding sequential states, pipeline registers, and intermediate values. Loop pipelining increases FF usage proportionally to the pipeline depth and unroll factor, as more storage elements are required to sustain parallel execution.

DSP slice utilization is 24.55%, largely due to high-speed arithmetic operations such as multiply-accumulate, division, and transcendental approximations. In this design, tanh and atanh computations are the primary DSP consumers.

BRAM usage is at 31.43% because *Lij* and *Lji* and adjacency matrices are stored in BRAM, which are accessed frequently and often in parallel.

I/O resource utilization is relatively high at 70.50%, driven by the need to handle wide data buses for LLR inputs, decoded outputs, and control/status signals. The parallelized architecture requires multiple simultaneous data transfers, contributing to the heavy I/O demand.

Table 5.8 summarizes the complete resource utilization of the HLS design using built-in function. LUTs account for approximately 68% of the total logic usage, primarily due to the arithmetic operations involved in fixed-point tanh and atanh computations. The application of *UNROLL* pragmas further increases LUT demand by replicating computational hardware to enable parallel processing of loop iterations.

Table 5.7: Resource Utilization Report for LUT function

Resource	Usage	Available	Utilization (%)
LUT	8178	53200	15.37
FF	4245	106400	3.99
BRAM	44	140	31.43
DSP	54	220	24.55
IO	141	200	70.50

FFs and DSP slices together contribute to nearly full utilization in their respective categories. FFs are extensively used to store sequential states, pipeline registers, and intermediate computation results. While loop pipelining improves throughput, it proportionally increases FF requirements as more stages and parallel paths are instantiated. DSP slices are employed for high-speed arithmetic operations such as multiply-accumulate, division, and transcendental function approximations. In this design, tanh and atanh evaluations are the primary DSP consumers due to their iterative or polynomial approximation steps.

I/O resources are heavily utilized (over 70%) to interface with external data streams and control signals. This includes reading LLR input vectors, writing decoded bit sequences, and handling control/status signaling. The high I/O percentage is a result of wide data buses and simultaneous multi-port accesses required by the parallelized architecture.

Table 5.8: Resource Utilization Report for Built-in function

Resource	Usage	Available	Utilization (%)
LUT	37173	53200	69.87
LUTRAM	3065	17400	17.61
FF	40947	106400	38.48
BRAM	53	140	37.50
DSP	153	220	69.55
IO	141	200	70.50

5.4 Performance Analysis

In this subsection, we analyze the performance of the RTL and HLS implementations in comparison to the MATLAB reference. Key performance parameters evaluated include the average number of iterations required for convergence at different SNR values, BER, throughput, and overall latency of the design.

5.4.1 Iterations to Converge

During decoding, the algorithm repeatedly updates the messages between variable nodes and check nodes until the parity-check conditions are satisfied or a maximum

number of iterations is reached. This metric is important because it provides insight into the speed and efficiency of the decoder: fewer iterations indicate faster convergence and lower latency, while more iterations imply higher computational effort and increased decoding time.

RTL

Due to the long simulation time per iteration, BER performance was evaluated over 20 frames for SNR values ranging from 2 dB to 6 dB, with the maximum number of iterations limited to 15. For SNR values between 3 dB and 6 dB, the decoder consistently converged to zero errors well before reaching the iteration limit. At SNR = 2 dB, convergence occasionally occurred closer to 20 iterations.

In Figure 5.5, the average number of iterations required for convergence is shown for SNR values from 2 dB to 6 dB, comparing the RTL implementation (red line) with the MATLAB reference (blue line). Showing that the RTL implementation matches the MATLAB reference closely across all tested SNRs.

HLS

After completing the design, testing was performed by feeding randomly generated bits (for SNR values between 3 and 6) into both models. The results showed that, for SNR values of 5 and 6, the HLS implementation required the similar number of iterations to decode as the reference model, while for lower SNR values it required 2-3 additional iterations.

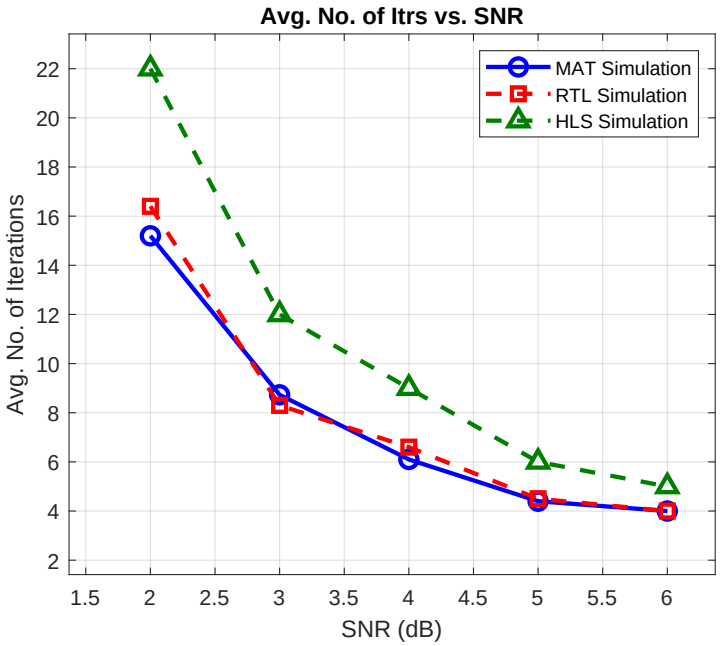


Figure 5.5: Average Number of Iteration to Converge Vs SNR for RTL, HLS and MATLAB Reference

5.4.2 Bit Error Rate

A lower BER indicates better error-correction performance, meaning the decoder can successfully recover the transmitted data even in the presence of noise. Evaluating BER across different SNR's allows for the assessment of the decoder's robustness under varying channel conditions, which is crucial for both space and telecommunication applications.

RTL

The BER vs. SNR plot in Figure 5.6, the decoder achieves zero errors for all SNR ≥ 3 dB within the 15-iteration limit. To highlight early convergence behavior and better visualize the waterfall region, the BER after the 5th iteration is also shown for each SNR value, alongside the HLS and MATLAB references for comparison.

HLS

For the BER vs. SNR plot in Figure 5.6, the decoder converges to zero errors for all SNR ≥ 3 dB within the 15-iteration limit. The characteristic waterfall behavior is also observed, although the performance is marginally worse than the reference implementation.

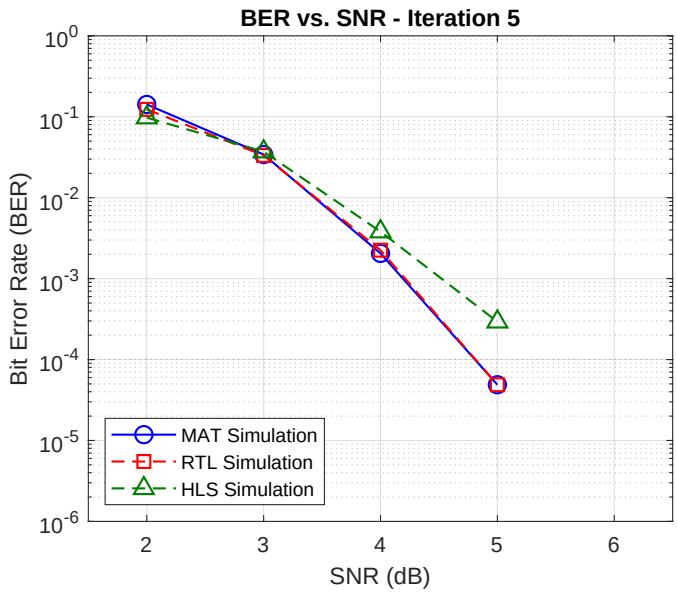


Figure 5.6: BER vs SNR for RTL, HLs and MATLAB Reference

5.4.3 Throughput and Latency

Throughput is the rate at which data is successfully decoded. It is a key performance metric because it indicates how fast the decoder can handle incoming data streams.

Latency is the total time required for the decoder to process a single message or codeword from input to output. It reflects the responsiveness of the decoder and is

influenced by factors such as the number of iterations, memory access times, and the degree of parallelism or pipelining in the design.

RTL

In simulation at 100 MHz, a single iteration takes approximately 20 ms to complete, at SNR values of 5 dB or 6 dB, the message typically converges by the 5th iteration, resulting in a throughput of only about 20.48 kbps for the codeword length 2048. This is significantly lower than industrial LDPC decoder implementations, which typically achieve throughput in the range of 100–800 gbps.

In this design, the overall latency to decode a message is approximately 100 ms, highlighting the drawback of a purely sequential architecture.

HLS

The HLS design achieves a throughput of 605 kbps with a latency of 3.74 ms, which, while improved over the RTL implementation, still falls significantly short of typical industry standards. This can be improved by ensuring parallelism at all critical stages, especially in the check node and variable node updates.

6

Discussion

In this chapter, we reflect on the results obtained and how they relate to the goals of the project.

6.1 Comparison of RTL and HLS

6.1.1 Timing Report

The initial HLS implementation using built-in functions failed timing at 100 MHz, exhibiting negative slack. In contrast, the LUT-based HLS implementation successfully met timing at the same frequency. Both the final HLS and RTL designs operate without negative slack at 100 MHz, with maximum achievable clock frequencies of 125 MHz for HLS and 121 MHz for RTL. The timing analysis at 100 MHz is summarized in Table 6.1.

Table 6.1: RTL vs HLS Timing Analysis

Category	**Worst Negative Slack (ns)**	
@100 MHz	**RTL Timing**	**HLS Timing**
Setup	0.428	0.966
Hold	0.014	0.049

6.1.2 Power

Comparing the power profiles of the RTL and HLS implementations. At 85 MHz, the initial HLS design exhibited higher dynamic power consumption than the RTL version. This increase is primarily driven by extensive loop unrolling and the use of multiple BRAM banks for parallel access. Additionally, the use of built-in tanh and atanh functions in the initial HLS version introduces fixed-point to floating-point conversions and complex arithmetic operations, which significantly increase DSP utilization and switching activity.

In contrast, the LUT-based HLS implementation shows a substantial reduction in both resource utilization, power, and timing delay. This improvement is mainly due to the combined effect of pipelining, loop unrolling, and replacing the built-in tanh/atanh calls with LUT-based approximations. These changes eliminate

Table 6.2: HLS Power Analysis

	RTL Power (W)	**HLS Power (W)**
Clock Frequency	**100 MHz**	**100 MHz**
Dynamic Power	0.114	0.129
Static Power	0.112	0.109
Total On-chip Power	0.226	0.238

costly floating-point conversions and reduce DSP demand, leading to lower switching power. From Table 6.2 although the HLS implementation exhibits slightly higher power consumption than the RTL design, the overall values are comparable.

6.1.3 Utilization

When comparing the utilization reports in Table 6.3, the HLS design uses far fewer FFs than the RTL counterpart because the HLS scheduler optimizes operation timing and avoids redundant registers and In the RTL design, the total L_j memory is implemented as LUT RAM instead of BRAM, using BRAM for L_j storage could reduce LUT and FF utilization.

Table 6.3: RTL vs HLS Utilization

Resource	**RTL Utilization (%)**	**HLS Utilization (%)**
LUT	35.13	15.37
FF	43.93	3.99
BRAM	28.57	31.43
DSP	5.00	24.55
IO	26.50	70.50

BRAM usage is slightly higher in HLS due to the *ARRAY_PARTITION* pragmas, which split large memories into multiple banks to enable concurrent access. DSP utilization is higher in HLS because it leverages dedicated DSP slices for high-speed arithmetic—particularly in fixed-point tanh/atanh approximations, multiplications, and divisions. Finally, the high I/O utilization in HLS stems from wide, parallel buses and simultaneous multi-port memory accesses generated by *UNROLL* and *ARRAY_PARTITION* directives, whereas the RTL design is sequential, resulting in lower I/O pin usage.

6.1.4 Performance

Iterations to Converge and Bit Error Rate

From Figures 5.5 and 5.6, we observe that the RTL simulation and the MATLAB reference model align very closely, indicating that the hardware implementation at the register-transfer level accurately reproduces the intended algorithm. On the other hand, the HLS simulation shows a noticeable deviation from the MATLAB/RTL

results, particularly at higher SNR values. This discrepancy arises mainly due to scaling differences, quantization effects, and synthesis overheads introduced by the HLS toolchain. Unlike RTL, which is bit-true to the algorithm, HLS often applies approximations, optimizations, or resource-sharing strategies that can slightly alter numerical precision, leading to the observed BER gap.

Throughput and Latency

For the RTL implementation, no pipelining was implemented, and all operations are executed sequentially under the control of a FSM. While the functionality is correct, the resulting throughput is extremely low, about 24.78 kbps with a latency of 100 ms. The decoder was initially developed with the assumption that pipelining could be added later once a functional RTL was achieved. However, it was later determined that implementing pipelining would require a complete redesign of the architecture. The restructuring strategies to overcome this performance limitation in the RTL design are discussed in section 6.2.

In the HLS implementation, pipelining directives were applied; however, pipelining was not achieved uniformly across all stages. Pipeline stalls were particularly evident during memory accesses (e.g., reading from the adjacency matrix) and within check node computations, preventing the pipeline from reaching optimal efficiency. Despite these bottlenecks, the HLS design still outperformed the RTL implementation in terms of throughput, owing to partial pipelining and a higher operating frequency. That said, there remains significant scope for improvement. Optimizing the scaling factors in the *ap_fixed* data type could yield a better balance between precision and resource utilization, thereby reducing unnecessary logic depth. Additionally, restructuring the code to minimize data dependencies and enhance memory access patterns would help alleviate pipeline stalls and further improve performance.

Table 6.4: Comparison of RTL and HLS Implementations with Reference works

	RTL	**HLS**	**Reference RTL** [8]	**Reference HLS** [9]
FPGA	Zynq 7020	Zynq 7020	Stratix 2	Virtex-7
Clock	121 MHz	100 MHz	128 MHz	100 MHz
H dimension	1536*2560	1536*2560	2304*1152	1000*1920
ELBs Used	46,738	16,356	69,100	16,019
Algorithm	Sum Product	Sum Product	Min-Sum	Min-sum
Throughput	24.78 Kbps	605 Kbps	768 Mbps	70 Mbps
Latency	100 ms	3.74 ms	–	–

6.1.5 Similar Implementation from references

The observed results and performance of our designs were compared with several reference papers. Table 6.4 summarizes similar LDPC code implementations and highlights the differences in hardware resource usage across various implementations. Direct comparison is challenging due to variations in FPGA architectures. To address this, [8] uses an approximate metric based on the fundamental FPGA building blocks 4-LUTs and FFs referred to as equivalent logic blocks (ELBs). ELBs

provide an estimate of the number of simple logic elements (one 4-LUT and one FF) required to implement each design. In our comparison, ELBs were calculated from the total LUT and FF counts that were used, providing a basis for fair comparison with our implementations. As discussed earlier, both the RTL and HLS designs demonstrate suboptimal performance, which could be significantly improved with architectural modifications and optimization techniques.

6.2 Development Effort

When comparing development effort and validation between RTL and HLS implementations, the differences are substantial. RTL design requires a detailed architectural plan before coding, with careful handling of control logic, memory management, and dataflow, making the development process significantly longer and more complex. Debugging and verification are also time-consuming, as errors are often tied to low-level signal interactions, requiring extensive simulation and waveform analysis. Since the decoder matrix size was large, the time required for debugging and validating the RTL design far exceeded the actual development effort. In fact, nearly twice as much time was spent on verification and fixing issues as compared to the initial implementation phase.

In contrast, HLS allows faster prototyping since designs are described in high-level C code, with tools automatically generating RTL. This not only reduces coding effort but also simplifies validation, as functional correctness can be tested at the C-level before synthesis. Moreover, verification in HLS is much faster, with easier integration of testbenches and faster simulation compared to RTL. For example, in HLS, running a few decoding iterations until reaching zero error could be completed within seconds, thanks to the high-level abstraction and faster C-based simulation. In contrast, the same test in RTL required more than an hour, since every clock cycle, signal transition, and memory access had to be evaluated at the gate level. This clearly illustrates how HLS accelerates the design–verification cycle, making it far more practical for rapid prototyping and iterative testing, whereas RTL simulations become a major bottleneck in terms of time and productivity.

6.3 Future Work

6.3.1 RTL Architecture Improvements

The current RTL implementation follows a purely sequential architecture controlled by an FSM. While functionally correct, this approach severely limits throughput. Future improvements should focus on two key strategies: deep pipelining and massive parallelism.

a) Introducing Pipelining

In the current design, memory access is serialized — an address is assigned, the FSM waits several cycles for data to be fetched, and only then is the next operation started. This prevents overlapping operations. With an efficient pipeline, each stage

(fetch, process, store) could operate concurrently, allowing new data to be processed every clock cycle.

b) Exploiting Parallelism

Currently, check node processing, variable node processing, and LLR updates are executed sequentially. By restructuring the architecture, these steps could be executed in parallel, significantly reducing overall latency.

With proper pipelining and parallelism, the decoder could move closer to the industrial-grade throughput range seen in optimized LDPC decoders.

6.3.2 HLS Improvement

The current HLS implementation has been optimized using directives for parallelism and pipelining, and while it is functionally correct, there is still room for improvement in both accuracy and performance. Key areas for future enhancement include:

- The design currently uses *ap_fixed<16,8>*. By carefully tuning the integer and fractional bit-widths, we can achieve a better trade-off between precision, resource usage, and timing performance. Dynamic range analysis can be performed to allocate more bits where necessary (e.g., LLR values, accumulation steps) and reduce unused precision in less critical paths.
- Optimize the stopping conditions for iterative algorithms to reduce unnecessary iterations, improving throughput and lowering power.

6.3.3 Data Streaming

All validations in this work were performed at the simulation level. For a more practical evaluation, it would be ideal to implement data streaming to the target FPGA board and assess the decoder's performance with real-time data. Simulation in Vivado is time-consuming, so real-time streaming and validation would provide a more efficient and effective means of testing, enabling performance measurements under realistic operating conditions.

7

Conclusion

Using LDPC decoders as the basis for comparing RTL and HLS implementations proved to be an effective choice. The large matrix size and high complexity of the decoder highlighted differences between the two approaches that might have been overlooked with simpler designs. Beyond performance metrics, factors such as development effort, testing and validation time, and overall ease of verification also show clear contrasts, giving a comprehensive view of the trade-offs between RTL and HLS.

Based on prior research [2], RTL implementations generally achieve superior performance compared to HLS, as they provide finer control over design aspects such as pipelining and parallelism. However, in this work, the RTL implementation delivered lower performance than HLS, primarily because the design was not fully optimized in terms of pipeline depth and parallel execution. In contrast, although our HLS implementation performed below the levels reported in related studies, it still outperformed our RTL design due to partial pipelining applied during development.

The RTL approach was not further optimized with advanced pipelining and parallelism strategies, as these require extensive planning, domain expertise, and significant development time, making the process considerably more complex and resource-intensive. Consequently, HLS proved more effective in this context, offering a faster path to functional results and making it well-suited for prototyping and early design exploration. By contrast, RTL remains the preferred option when maximum performance and hardware efficiency are required, provided sufficient time and expertise are available for thorough optimization.

Bibliography

[1] M. Rowshan, M. Qiu, Y. Xie, X. Gu, and J. Yuan, "Channel coding toward 6G: Technical overview and outlook," *IEEE Open Journal of the Communications Society*, vol. 5, pp. 2585–2685, 2024.

[2] J. Andrade, N. George, K. Karras, D. Novo, F. Pratas, L. Sousa, P. Ienne, G. Falcao, and V. Silva, "Design space exploration of LDPC decoders using high-level synthesis," *IEEE Access*, vol. 5, pp. 14 600–14 615, 2017.

[3] Y. Zhang, Q. Cao, J. Yao, and H. Jiang, "R-LDPC: Refining behavior descriptions in HLS to implement high-throughput LDPC decoder," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[4] *TC Synchronization and Channel Coding—Summary of Concept and Rationale*, Consultative Committee for Space Data Systems (CCSDS), October 2021, cCSDS 230.1-G-3. [Online]. Available: https://public.ccsds.org/Pubs/230x1g3e1.pdf

[5] *TM Synchronization and Channel Coding—Recommendation for Space Data System Standards*, Consultative Committee for Space Data Systems (CCSDS), 2022, CCSDS 131.0-B-5. [Online]. Available: https://ccsds.org/Pubs/131x0b5.pdf

[6] B. P. Swathi, K. Amulya, K. V. Divya Shree, M. N. Suma, and K. Keerthi, "Design and implementation of LDPC decoder using VHDL for space application," in *2024 2nd International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, 2024, pp. 1–8.

[7] C. Condo and G. Masera, "Unified turbo/LDPC code decoder architecture for deep-space communications," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 50, no. 4, pp. 3115–3125, 2014.

[8] P. Hailes, L. Xu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "A survey of FPGA-based LDPC decoders," *IEEE Communications Surveys and Tutorials*, vol. 18, no. 2, pp. 1098–1122, 2016.

[9] Y. Delomier, B. Le Gal, J. Crenne, and C. Jego, "Model-based design of efficient LDPC decoder architectures," in *2018 IEEE 10th International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, 2018, pp. 1–5.

[10] V. SILVA, "Design space exploration of LDPC decoders using high-level synthesis," *IEEE Access*, 2017.

[11] K. Cushon, P. Larsson-Edefors, and P. Andrekson, "Low-power 400-Gbps soft-decision LDPC FEC for optical transport networks," *IEEE Journal of Lightwave Technology*, vol. 34, no. 18, pp. 4304–4311, Sep. 2016.

[12] W. E. Ryan and S. Lin, "Coding and capacity," in *Channel Codes: Classical and Modern*, W. E. Ryan and S. Lin, Eds. Cambridge, UK: Cambridge University Press, 2009.

[13] E. A. Lee and D. G. Messerschmitt, *Digital communication.* Springer Science & Business Media, 2012.

[14] D. J. Costello and G. D. Forney, "Channel coding: The road to channel capacity," *Proceedings of the IEEE*, vol. 95, no. 6, pp. 1150–1177, 2007.

[15] W. E. Ryan *et al.*, "An introduction to LDPC codes," *CRC Handbook for Coding and Signal Processing for Recording Systems*, vol. 5, no. 2, pp. 1–23, 2004.

[16] Z. Tu and S. Zhang, "Overview of LDPC codes," in *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*, 2007, pp. 469–474.

[17] W. J. Dally, R. C. Harting, and T. M. Aamodt, *Digital Design Using VHDL: A Systems Approach.* Cambridge, UK: Cambridge University Press, 2016.

[18] F. Vahid, *Digital design with RTL design, VHDL, and Verilog.* John Wiley & Sons, 2010.

[19] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS today: successes, challenges, and opportunities," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–42, 2022.

[20] D. Inc., *Eclypse Z7 Hardware Reference Manual*, 2025, accessed: 2025-09-01. [Online]. Available: https://digilent.com/reference/programmable-logic/eclypse-z7/reference-manual

[21] AMD, *Block Memory Generator v8.4 Product Guide (PG058)*, 2021, accessed: 2025-09-01. [Online]. Available: https://docs.amd.com/v/u/en-US/pg058-blk-mem-gen

[22] AMD, *HLS Programmers Guide*, 2025, accessed: 2025-09-01. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/HLS-Programmers-Guide

[23] F. Pratas, J. Andrade, G. Falcao, V. Silva, and L. Sousa, "Open the gates: Using high-level synthesis towards programmable LDPC decoders on FPGAs," in *2013 IEEE Global Conference on Signal and Information Processing*, 2013, pp. 1274–1277.

[24] S.-D. Roh, K. Cho, and K.-S. Chung, "Implementation of an LDPC decoder on a heterogeneous FPGA-CPU platform using SDSoC," in *2016 IEEE Region 10 Conference (TENCON)*, 2016, pp. 2555–2558.

[25] K. S. Kathiresan, "LDPC decoder RTL implementation - sequential logic repository," https://github.com/BloodRainz/RTL_LDPC_SPA_Sequential, 2025.

A

Appendix 1

The complete source code for the RTL implementation of LDPC decoder design, including MATLAB simulations is available at the following GitHub repository:

`https://github.com/BloodRainz/RTL_LDPC_SPA_Sequential` [25]

Note: The repository is not yet fully organized, and a detailed user guide for setup and usage will be developed in the future.