12-2020

# Hardware Implementations of CCSDS Deep Space LDPC Codes for a Satellite Transponder

Dana R. Sorensen
*Utah State University*

# HARDWARE IMPLEMENTATIONS OF CCSDS DEEP SPACE LDPC CODES FOR A SATELLITE TRANSPONDER

by

Dana R. Sorensen

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

| | |
|---|---|
| Jacob H. Gunther, Ph.D.<br>Major Professor | Todd K. Moon, Ph.D.<br>Committee Member |

Charles M. Swenson, Ph.D.
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2020

# ABSTRACT

Hardware Implementations of CCSDS Deep Space LDPC Codes for a Satellite Transponder

by

Dana R. Sorensen, Master of Science

Utah State University, 2020

Major Professor: Jacob H. Gunther, Ph.D.
Department: Electrical and Computer Engineering

LDPC (Low Density Parity Check) codes offer error-correcting capability that approaches the Shannon Limit. A set of short block-length LDPC codes has recently been standardized by the CCSDS (Consultative Committee for Space Data Systems) for use with satellite telecommands. This report describes and implements state-of-the art decoding algorithms and architectures with varying degrees of serialization. These implementations are compared by implementation loss, speed, and complexity. A trade study is performed to determine the best decoder architecture in the context of the CCSDS code on the Iris small satellite radio platform. A simplified SNR (Signal-to-Noise Ratio) estimation algorithm is developed and implemented for use with the decoders, and an LDPC encoder with a novel flexible-width design is implemented to support the ten CCSDS telemetry codes.

(64 pages)

# PUBLIC ABSTRACT

Hardware Implementations of CCSDS Deep Space LDPC Codes for a Satellite Transponder

Dana R. Sorensen

Error-correction coding is a technique that adds mathematical structure to a message, allowing corruptions to be detected and corrected when the message is received. This is especially important for deep space satellite communications, since the long distances and low signal power levels often cause message corruption. A very strong type of error-correction coding known as LDPC codes was recently standardized for use with space communications. This project implements the encoding and decoding algorithms required for a small satellite radio to be able to use these LDPC codes. Several decoder architectures are implemented and compared by their performance, speed, and complexity. Using these LDPC decoders requires knowledge of the received signal and noise levels, so an appropriate algorithm for estimating these parameters is developed and implemented. The LDPC encoder is implemented using a flexible architecture that allows the entire standardized family of ten LDPC codes to be encoded using the same hardware.

CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# NOTATION

**Symbols**

| | |
|---|---|
| $n$ | length of the codeword for a given code |
| $k$ | dimension of a given code (or length of message) |
| $R$ | code rate |
| $G$ | generator matrix |
| $H$ | parity check matrix |
| $\mathbf{m}$ | input message vector |
| $\mathbf{c}$ | codeword vector |
| $M$ | PCM submatrix size |
| $m$ | PCM circulant size |
| $v$ | index of variable nodes |
| $c$ | index of check nodes |
| $\mathcal{V}(c)$ | set of variable nodes connected to check node $c$ |
| $\mathcal{C}(v)$ | set of check nodes connected to variable node $v$ |
| $\mathcal{V}(c) - \{v\}$ | variable nodes connected to check node $c$, excluding $v$ |
| $L_{v \to c}$ | log-likelihood message from variable node $v$ to check node $c$ |
| $L_{c \to v}$ | log-likelihood message from variable node $c$ to check node $v$ |
| $i$ | received symbol index |
| $L_i$ | log-likelihood channel input |
| $y_i$ | received symbol |
| $b_i$ | transmitted bit value |
| $z_i$ | sampled AWGN noise |
| $E_c$ | energy per coded bit |
| $\alpha$ | signal amplitude |
| $\sigma^2$ | noise variance in the AWGN channel |

# ACRONYMS

| | |
|---|---|
| AR4JA | Accumulate, Repeat-by-4, Jagged Accumulate |
| AWGN | Additive White Gaussian Noise |
| BCH | Bose-Chaudhuri-Hocquenghem |
| BP | Belief Propagation algorithm |
| BRAM | Block RAM |
| CCSDS | Consultative Committee for Space Data Systems |
| CMEM | Check Node Memory |
| CNU | Check Node Unit |
| dB | Decibel |
| DSN | Deep Space Network |
| FF | Flip-flop |
| FPGA | Field Programmable Gate Array |
| IDMEM | Identity Memory |
| JPL | Jet Propulsion Laboratory |
| LDPC | Low Density Parity Check |
| LLR | Log-Likelihood Ratio |
| LUT | Lookup Table |
| MS | Min-Sum algorithm |
| PCM | Parity Check Matrix |
| RCE | Recursive Convolutional Encoder |
| SDL | Space Dynamics Laboratory |
| SNR | Signal-to-Noise Ratio |
| SP | Sum-Product algorithm |
| TC | Telecommand |
| VLSI | Very Large-Scale Integration |
| VMEM | Variable Node Memory |
| VNU | Variable Node Unit |

# CHAPTER 1

# INTRODUCTION

This project proposes a review of existing state-of-the-art LDPC decoding algorithms and implementation architectures, specifically targeting CCSDS deep space telecommand and telemetry LDPC codes on the Iris Radio platform. Algorithms and their variants are compared by implementation loss, speed, and complexity. Several architectures are implemented, and the best decoder architecture in the context of the CCSDS codes is selected for use on the Iris Radio.

Thousands of papers about LDPC codes have been published in the last few decades, exploring code design, analysis, and decoding algorithms. As a result, many codes and variations of decoding algorithms exist, but it is not always clear what the various trade-offs are for each variation, especially for a given code.

In 2017, the Consultative Committee for Space Data Systems (CCSDS) released a revision to the TC Synchronization and Channel Coding standard [1], adding the specifications for a new short block-length Low Density Parity Check (LDPC) code. This code outperforms the legacy BCH code in its error-correct capabilities, with the potential to function at several dB below the previous BCH telecommand threshold.

The CCSDS has also defined several LDPC codes for use on satellite telemetry (space-to-earth) links. These codes have performance comparable to turbo codes, but at higher rates. Encoding algorithms are compared and the best candidate is implemented for the Iris Radio.

The Iris Radio is a DSN-compatible small satellite transceiver designed by JPL and now produced by SDL. LDPC encoding and decoding is a desired feature that is not currently supported by the radio. The best algorithms resulting from this study will be selected for use on the Iris Radio.

The contributions of this project include a novel flexible-width design for recursive

convolutional encoders, a simplified SNR estimation algorithm for the decoder, and a serial decoder design particularly well suited to the CCSDS codes.

## 1.1 Project Objectives and Criteria

This project started with the following objectives:

- Characterize the performance and complexity of existing LDPC decoding algorithms and architectures.
- Optimize existing algorithms for use with the CCSDS Telecommand LDPC code.
- Produce a functional LDPC decoder hardware implementation for use on the Iris Radio.

Due to sufficient funds and time, the following objectives were added:

- Explore the design space for LDPC encoders, specifically relating to the CCSDS telemetry codes.
- Implement an LDPC encoder for use on the Iris Radio.

Each algorithm and its associated architectures are evaluated based on the these three main criteria:

- Implementation loss. Many algorithms sacrifice some error-correcting performance for reduced complexity. This is measured by generating bit error rate plots at various signal-to-noise ratios.
- Speed. In general, higher speeds require higher FPGA utilization and may also sacrifice error-correcting performance. The speed constraint is also driven by the highest uplink rate the Iris Radio supports, and is measured in clock cycles per bit per iteration.
- FPGA utilization. An efficient algorithm and architecture uses fewer FPGA resources to implement. These resources include LUTs, FFs, and BRAMs.

These criteria are interdependent. In general, one is improved at the expense of the other two. Appropriate bounds are set on each criteria based on what the Iris Radio supports, and then a trade study is performed to balance the trade-offs and select the best implementation for the target code and platform.

## 1.2 Project Overview

Figure 1.1 diagrams the methodology that was used to meet the project objectives. The project started with an in-depth review of prior work and current state-of-the art LDPC encoding and decoding algorithms and architectures. Several architectures were selected for prototyping in MATLAB, and then implemented on a Xilinx FPGA platform. These implementations were then characterized based on error-correcting performance, speed, and FPGA utilization. Based on these metrics, one encoding implementation and one decoding implementation were selected for use on the Iris Radio.

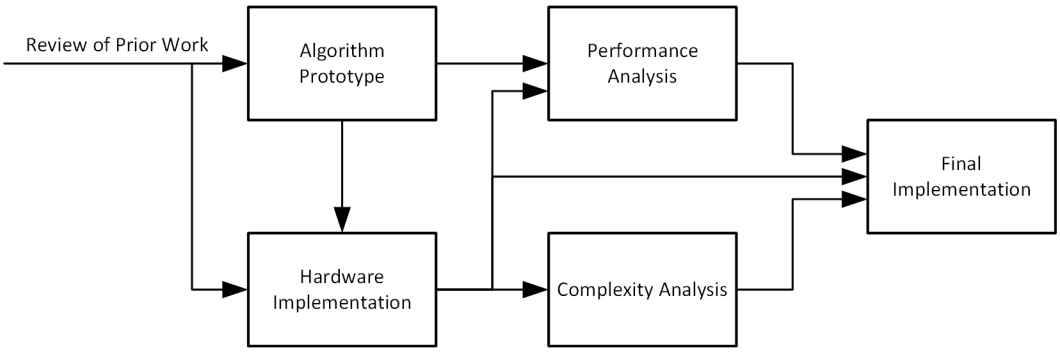



Fig. 1.1: Functional Diagram of Research Method

## 1.3 History of LDPC Codes

Low Density Parity Check codes were originally proposed by Gallager in 1962 [8]. They showed promise, but were too computationally complex for that time to be of much use. As a result, they sat largely unnoticed until the mid 1990s with the work of MacKay, who showed that LDPC codes can approach the Shannon limit of performance [9, 10]. The formulation of the iterative Belief Propagation (BP) decoding algorithm and the advent of the modern

FPGA and VLSI technology has enabled practical LDPC encoding and decoding.

Improvements to the structure of the codes have also been made. The original codes proposed by Gallager had regular weights (the number of 1s of each row and column were constant) and were limited by poor error floor performance. New irregular LDPC codes offer much better performance in the error floor region.

## 1.4 Overview of LDPC Codes

An LDPC code is any code having a sparse (low density) parity check matrix. The sparseness of the matrix reduces the complexity of encoding and decoding, and has been shown to not reduce the performance of the code [10].

### 1.4.1 Structure of LDPC Codes

A Low Density Parity Check (LDPC) code is any error-correcting code with a sparse parity check matrix, hence the name. The sparsity of the parity check matrix facilitates the decoding process, without compromising the error-correcting performance. LDPC codes have been proven to approach the channel capacity [8] and are relatively simple to encode and decode. These traits make the LDPC family of codes some of the best known codes. In order to reach capacity, however, the block sizes must be sufficiently large.

An LDPC code has no required structure, and may be completely randomly generated. In fact, such a randomly generated code in general has good minimum distance properties, but can be computationally burdensome to encode and decode due to the lack of structure. Subclasses of LDPC codes have been proposed, such as accumulate-repeat codes, quasi-cyclic protograph-based codes, and more. The additional structure imposed allows for an efficient decoder design.

There are two classes of LDPC codes: regular and irregular. Regular codes are described by a parity check matrix which contains a constant number of ones (a constant weight) in each column, and a constant weight in each row. Irregular codes do not have that constraint, and so may have any number of ones in each row or column (while still

being "sparse"). Research has shown that irregular codes generally perform better than regular codes.

Since LDPC codes are quite unconstrained in structure, being any code with a sparse parity check matrix, encoding is generally achieved by multiplication with the generator matrix.

Decoding is usually performed using a belief propagation (BP) algorithm. This is an iterative decoding algorithm which uses posterior probabilities to inform estimates about received bit values, which are then used to produce updated posterior probabilities, and so on. These algorithms are nearly optimal and generally converge within a few iterations, unless the SNR is very low.

Because LDPC codes are so loosely defined, a wide range of trade-offs exists for LDPC codes. Other codes may only perform well at high rates, for example, whereas good LDPC codes exist for a wide variety of block sizes and rates.

### 1.4.2 Performance of LDPC codes

LDPC codes have been shown to approach the channel capacity for large block sizes [10]. They have performance comparable to that of turbo codes, and with the advantage of generally fewer computations per bit to decode [11]. They are especially good at higher rates [3], and thus complement the good performance of Turbo codes at low rates.

One of the limitations of LDPC code performance is the presence of an error floor region [12]. As the SNR increases, the performance of LDPC codes increases exponentially until it hits this error floor. The location and size of the error floor is a major factor in LDPC code design. Modern irregular LDPC codes perform much better in this regard than the original regular ones.

A good LDPC code can have a large minimum distance, allowing a decoder to use the codeword criterion as a stopping condition and to detect errors with an undetected error rate of less than $10^{-10}$ at any Eb/N0 [11].

### 1.4.3 Challenges

One challenge of LDPC codes is the encoding complexity. Matrix-vector multiplication generally has a complexity of $O(n^2)$. Sparseness of the parity check matrix is not a sufficient condition for sparseness of the generator matrix, so the full complexity is usually required. Since their discovery, some efficient algorithms have been proposed to achieve nearly linear encoding complexity for most codes [13]. This difficulty has also been overcome by designing codes with structured generator matrices. For example, some codes have a block-circulant generator matrix, which is easily implemented using shift registers.

Decoding is an iterative process, so research is being done to find the most efficient ways to converge as quickly as possible.

Because LDPC codes perform better with larger block sizes, many codes have a large parity check matrix (PCM). This can make it difficult to characterize some aspects of the codes, since a brute force computer search (e.g. for minimum distance or codeword weights) would take far too long.

LDPC codes are an active research topic, and hundreds of papers are published each year addressing code design, efficient decoding algorithms, error floor mitigation, and more.

## 1.5 Overview of CCSDS LDPC Codes

There are several codes defined by the CCSDS for both telecommand (earth-to-space) and telemetry (space-to-earth). Since the focus of this project is to implement a telecommand decoder and telemetry encoder on a satellite, the parity check matrices (for decoding) are the focus of the discussion on telecommand codes, and the generator matrices (for encoding) are the focus of the discussion on telemetry codes.

### 1.5.1 Telecommand Codes

In 2017, the Consultative Committee for Space Data Systems (CCSDS) published an update to the Recommended Standard for TC Synchronization and Channel Coding [1]. This update added the option to use short block-length LDPC codes to encode uplink telecommands sent to spacecraft. The specification defines two systematic rate 1/2 codes,

with codeword lengths $(n = 128, k = 64)$ and $(n = 512, k = 256)$. These new LDPC codes are intended to replace the legacy BCH codes which, while easy to encode and decode, have much worse performance than modern codes. There are a handful of papers describing the approach taken for designing these codes [11, 14, 15].

The telecommand codes have short block lengths for LDPC codes, but they have a relatively large minimum distance. They were chosen because with satellite communications, the commands are generally short and known in advance. In contract, telemetry payloads generally require a higher-throughput channel for downlink. Nevertheless, they achieve significantly better performance than the legacy BCH codes used. Figure 1.2 compares the word error rates of the CCSDS-defined telecommand coding schemes.

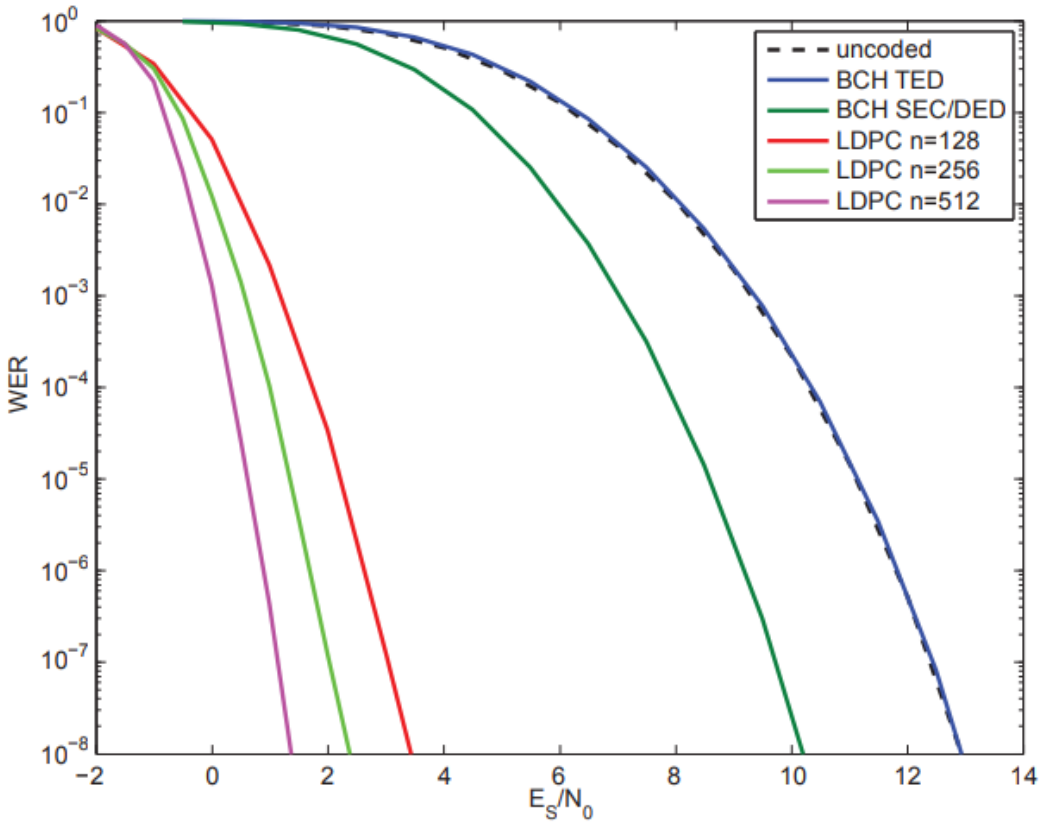



Fig. 1.2: Word Error Rates for Several CCSDS Uplink Coding Schemes (from [1]).

Both codes are constructed from a protograph [14]. A small Tanner graph called a protograph is constructed as shown in Figure 1.3, where circles represent variable nodes, squares represent check nodes, and edges represent the bi-directional message passing paths.

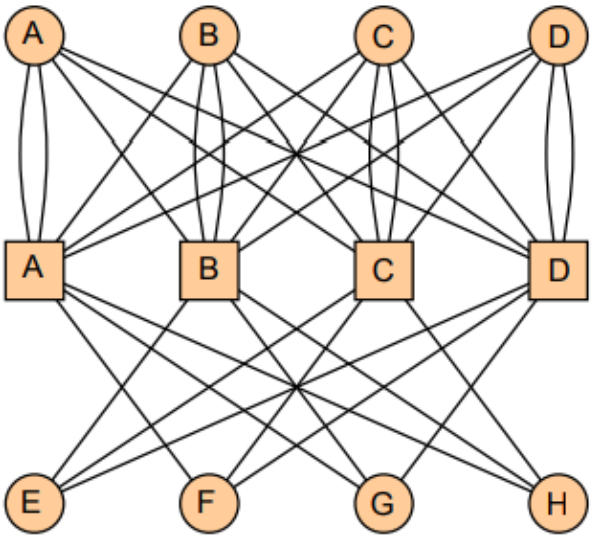



Fig. 1.3: Protograph for Short Block Length LDPC Codes (from [1]).

The protograph is duplicated $M$ times ($M = 16$ for the shorter code and $M = 64$ for the larger) to produce larger codes. Each bundle of $M$ edges is cut, cyclically permuted, then reattached. The result is a code with a block-circulant parity check matrix with row weight 8 and column weights of 3 and 5. The full PCM for the $n = 128$ code is shown in Figure 1.4, where a dot represents a non-zero entry at that location. The cyclic permutations were selected so that both codes have 1s on the main diagonal of the left half of the PCM. This is used to simplify the memory addressing in the serial decoder.

The underlying protograph structure is taken advantage of in the serial decoder (Section 3.4.3), and the structure of the cyclic permutations is taken advantage of in the partially-parallel decoder (Section 3.4.2).

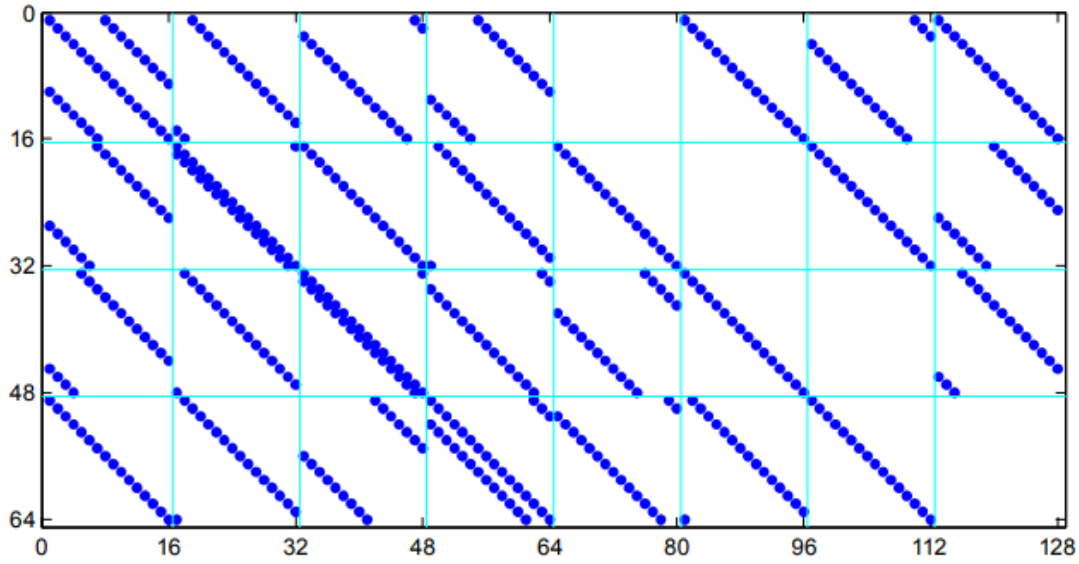



Fig. 1.4: Parity Check Matrix for the (128,64) LDPC Code (from [1]).

### 1.5.2 Telemetry Codes

In August 2011, the CCSDS standardized 10 systematic LDPC codes for satellite telemetry [4, 11]. One is a high-rate $R = 223/255$ (approximately 7/8) code with a block size of $k = 7136$. The others are a family of 9 codes with rates 1/2, 2/3, and 4/5, and block sizes k=1024, 4096, and 16384. The various options of block size and code rate allow engineers to manage trade-offs and select a code best matched to a particular mission.

A bit error rate comparison of CCSDS-defined telemetry codes is shown in Figure 1.5. Note that Turbo and LDPC codes both perform close to capacity.

The advantage of the LDPC codes over Turbo codes is the higher code rate. This means that more data can be transmitted with less overhead, suitable for higher data rates. This of course comes at the penalty of coding gain, but this is a managed trade-off. Figure 1.6 shows the bit error rate in the symbol domain to accentuate the effect of the various rates. Notice that the LDPC codes form a natural extension of the existing turbo codes, and their bit error rate curves are spaced approximately 2 dB apart in the symbol domain.

The performance of these codes complements the existing Turbo codes standardized by the CCSDS. The Turbo codes perform very well at low rates, whereas the LDPC codes are

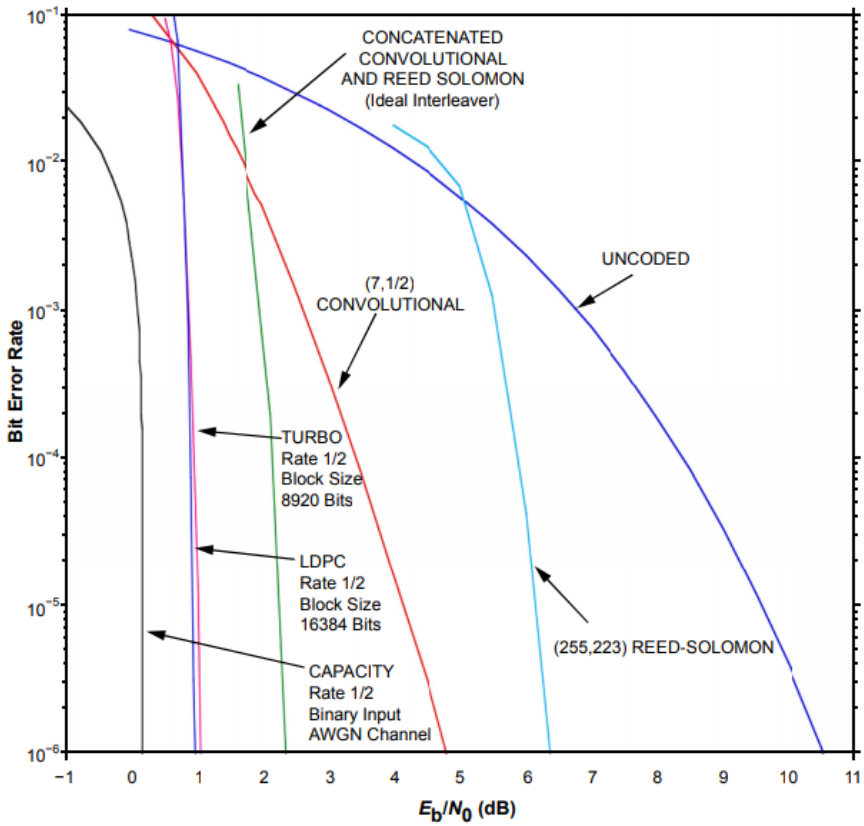



Fig. 1.5: Performance Comparison of CCSDS Convolutional, Reed-Solomon, Concatenated, LDPC, and Turbo Codes (from [2]).

defined at higher rates. This gives designers a larger solution space, depending on whether the communications are bandwidth-limited or power-limited.

Figure 1.7 illustrates this idea. If a system is power-limited, the codes on the far left will perform well. If the system is bandwidth-limited, the codes at the bottom left will perform well.

The construction of these codes is not covered here [11], since it was not taken into account in the design of the encoder. However, these codes do have systematic block-circulant generator matrices, the structure of which was taken advantage of in order to construct a low-complexity encoder.

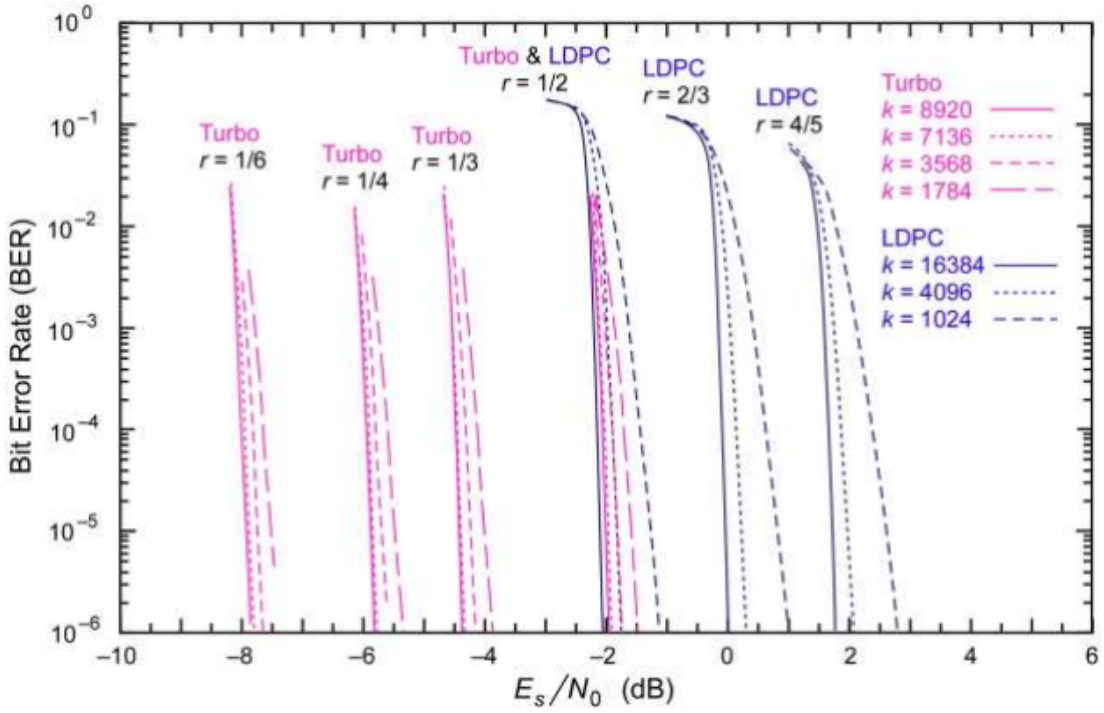



Fig. 1.6: LDPC and Turbo Code Comparative Performance (from [3]).

**Rate 7/8 Code**

The so-called "rate 7/8" code is actually a rate $R = 223/255$ code formed from a basic $(8176, 7156)$ code, which is then expurgated, shortened, and extended to form the recommended $(8160, 7136)$ code (see Figure 1.8). These sizes were chosen because the rate and dimensions match those of the CCSDS-defined Reed-Solomon Interleave-4 code. The code dimensions are also divisible by 32 for convenient processing.

The basic code's PCM is a $2 \times 16$ block array of $511 \times 511$ circulants. Each circulant has a row and column weight of 2, resulting in the code having row weights of 32 and column weights of 4.

A systematic block-circulant generator matrix is provided along with the specification. This generator matrix is what is used for the encoder implementation for this project.

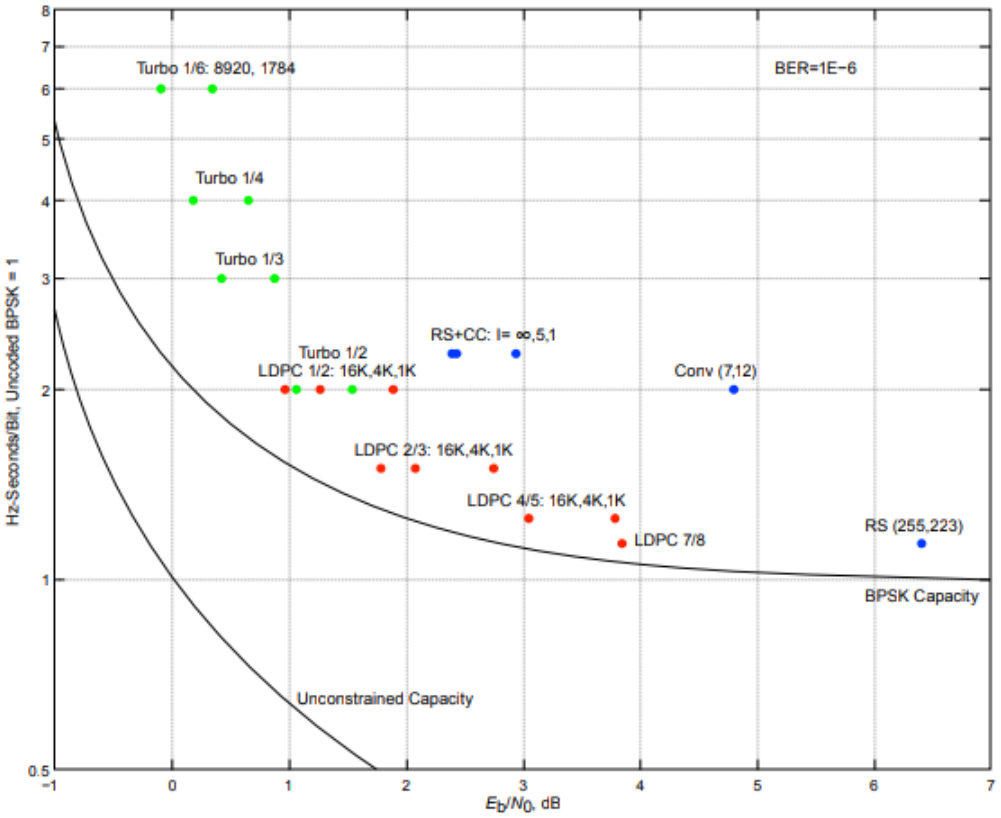



Fig. 1.7: Power Efficiency versus Spectral Efficiency for Several CCSDS Codes (from [2]).

**Family of 9 AR4JA Codes**

The remaining CCSDS telemetry codes are a family of 9 AR4JA (Accumulate, Repeat-by-4, Jagged Accumulate) codes [16, 17]. This family consists of every combination of the block-lengths $k = 1024, 4096, 16384$ and code rates $R = 1/2, 2/3, 4/5$. The codes are designed so that a constant block-length $k$ can be used across several rates, allowing the spacecraft's command and data handling system to maintain consistent data frame sizes without requiring modification to change rates.

The specification provides the PCMs for each code, from which the generator matrices can be derived. The parity check matrices and generator matrices for these codes are block-circulant, and the last block-column of symbols is punctured. The power-of-two block lengths and small-integer rate ratios simplify the implementation.

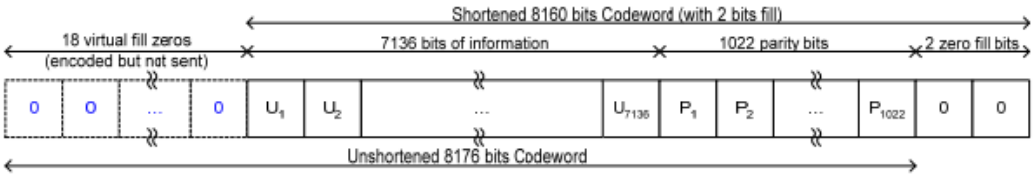



Fig. 1.8: CCSDS Rate 7/8 Codeword (from [4]).

## 1.6 Iris Radio

The Iris radio is a small-satellite radio (about 0.5 U) that is compatible with NASA's DSN. Historically, only large satellites have been able to push beyond low-earth orbit. However, with the rising popularity of small satellites about the size of a shoe box, more and more missions are pushing the boundaries of what can be achieved using such a small form factor. The Iris Radio supports these types of missions by allowing the satellites to communicate with the earth at large distances and relatively low signal-to-noise ratios.

Iris is a software-defined radio, meaning that most of the signal processing is performed in the digital domain on an FPGA. This allows new features, such as new coding standards, to be implemented without any hardware changes.

### 1.6.1 History

Iris was originally designed by the Jet Propulsion Laboratory. It has been licensed to the Space Dynamics Laboratory in Logan, Utah for fabrication, testing, and improvements.

Iris has flight heritage on the MarCO-A and MarCO-B satellites, which have the fame of being the first interplanetary CubeSats. This pair of communication relays used Iris radios to relay information about the entry, descent, and landing of NASA's InSight Mars lander in real-time back to earth.

Over time, Iris has evolved as new features and capabilities have been added. Each new addition allows the radio to function in new situations and work for new missions and new scenarios.

### 1.6.2 LDPC Addition

In an effort to improve the functionality of the Iris radio, a trade study was performed which enumerated the capabilities of NASA's Deep Space Network, then compared those capabilities with the functions of the Iris Radio.

It was determined that the recently-standardized LDPC codes were a great candidate for improvement, since they provide significant coding gains for telecommands, and higher coding rates for telemetry. This improvement is also an example of the advantages of a software-defined radio, since the additional features require no hardware changes.

In order to meet the requirements of a wide range of customers, the goal is to support all CCSDS-defined LDPC codes at all rates and block sizes.

# CHAPTER 2

# ENCODER DESIGN

The encoder targets the family of 9 AR4JA telemetry codes, along with the rate 7/8 telemetry code. Since the implementation is for a satellite radio, encoding of the telecommand codes is not considered.

## 2.1 Overview of Encoding Algorithms and Architectures

The most direct approach to LDPC encoding is matrix-vector multiplication, $\mathbf{c} = \mathbf{m}G$. Although an LDPC parity check matrix is sparse, the generator matrix is generally dense, so the encoding complexity is quadratic in the message length $k$. The memory requirements to store a dense generator matrix can also be very expensive. In the case of the $r = 1/2$, $k = 16384$ telemetry code, storing the non-systematic part of the generator matrix would require nearly 270 million bits! Until 2001, this was the accepted way of encoding an LDPC code, and was one of the primary challenges for LDPC codes.

In 2001, Richardson and Urbanke showed that by manipulating the parity check matrix to an approximate lower-triangular form and decomposing into component submatrices, the encoding complexity can be nearly linear, with the quadratic part dependent only on a small gap $g$ [13]. The larger the gap, the more difficult it is to encode. Many families of codes have a gap of 0, meaning that encoding is truly linear in complexity. The CCSDS codes unfortunately have a gap of $(n - k)/2$ [18], meaning that the RU decoding method has no advantage over a direct encoder for the CCSDS codes. Additionally, the RU decoder introduces latency and is not as flexible, so supporting all ten telemetry codes would be very difficult.

Fortunately, the CCSDS telemetry codes have an important property that aids encoding: the generator matrices are block-circulant. Shift registers can be used to significantly reduce the complexity of the operation and the size of the generator matrix to be stored.

The natural implementation shown in Figure 2.1 can encode messages in real time with serial inputs. It requires storage of the first row of each circulant block of the generator matrix, $(n-k)/m$ shift registers of length $m$, $2(n-k)$ memory cells, and $(n-k)$ binary multiply-accumulate operations per cycle. A message is encoded in $k$ cycles.

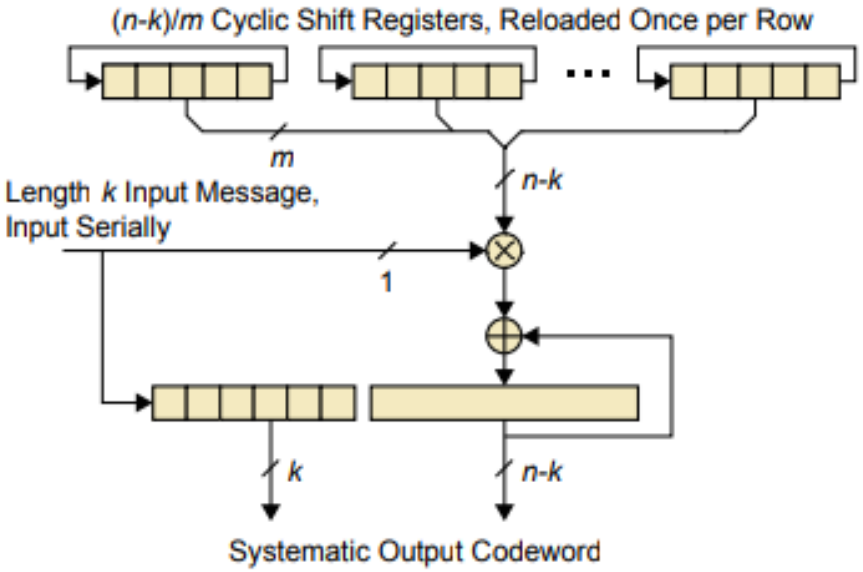



Fig. 2.1: A Simple Block-Circulant LDPC Encoder (from [2]).

A slightly better approach than the conceptually-simple shift register is to use a recursive convolutional encoder (RCE) [19], as shown in Figure 2.2. Instead of shifting the generator matrix bits, the shifts are built into the accumulator. This way, the generator matrix bits are stationary and can be replaced by combinatorial function generators or block memory. This architecture requires only $(n-k)$ bits of memory and $n-k$ binary multiply-accumulate operations per cycle. Messages are again encoded in $k$ cycles.

The implementation for this project uses a modified RCE approach, in which the RCE instances are designed to have a flexible recursion length (shown in Figure 2.2 as $m$), allowing the entire family of CCSDS codes to use the same RCEs on the FPGA. This length is configurable at run-time. Due to the number of supported codes, block RAM is

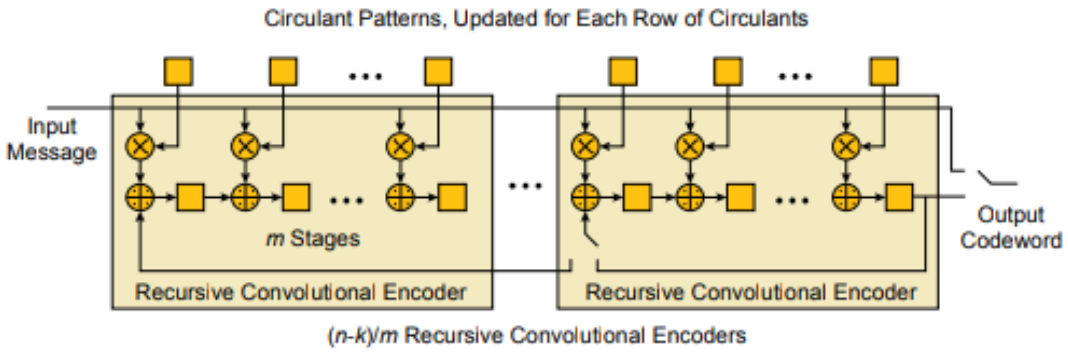



Fig. 2.2: An RCE Architecture for LDPC Encoding (from [2]).

used to store the first row of each circulant block of the generator matrices. The RCEs are additionally modified to use block RAM to store intermediate results, partially serializing the process and reducing resource utilization.

## 2.2 Modifications to Traditional RCE

The RCE architecture described in [18, 19] was modified to increase flexibility and decrease resource utilization. These changes allow the same physical encoder to support all 10 CCSDS telemetry codes, and allow the designer to control the trade-off between speed and resource utilization.

### 2.2.1 Serialization

In order to decrease resource utilization, the RCE encoding process is partially serialized. Intermediate results are stored in block RAM and recalled for later use. If this intermediate result memory were not used, the largest code (k=16384, r=1/2) would require 16384 flip flops for the shift registers. With this modification, the number of flip flops for that code can be reduced to 2048.

In a regular RCE implementation, $b = (n - k)/m$ RCEs of length $m$ would be instantiated, where $b$ is also the number of parity bit block-columns in the generator matrix. For example, the AR4JA codes have $b = 8$. In a regular RCE implementation, all 8 cyclic block columns would be encoded simultaneously by 8 distinct RCEs, each of length $m$.

With this modification, a smaller number of RCEs $1 \leq f \leq b$ can be instantiated. For each incoming message bit, the instantiated RCEs operate on the first $f$ circulant block-column(s) of the generator matrix. Then the $f \times m$ accumulated bits are written to working memory, and the next $f$ block-columns' saved bits are read from the same memory. These bits are used with the same input message bit to perform computations for the next $f * m$ parity bits, then are written back to working memory. This procedure is repeated until all of the parity bit computations have been performed for the input message bit.

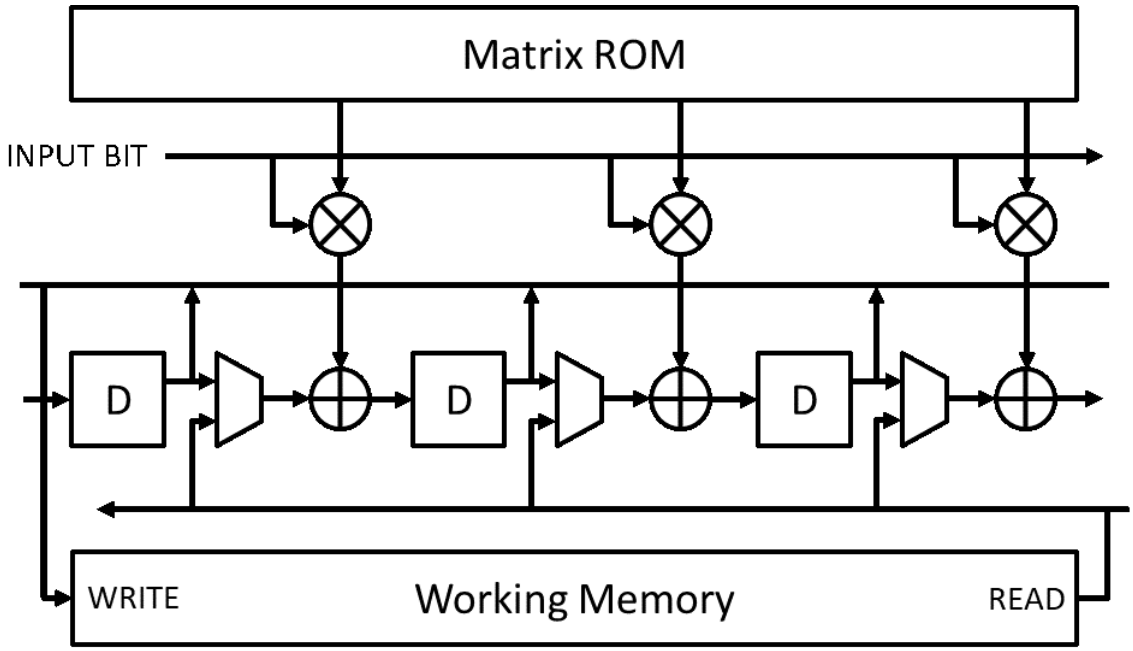



Fig. 2.3: Serialized RCE Architecture

This working memory architecture is shown in Figure 2.3. Once a set of binary multiply-and-accumulate operations have finished, the results are written to the working memory and the next set is loaded into the shift registers. This architecture allows the working memory to be bypassed when encoding is finished in order to read out coded parity bits.

The obvious drawback of serialization is a reduction of speed. A traditional RCE implementation can perform the $(n-k)$ computations for each input bit in a single cycle. The serialized implementation takes $\lceil b/f \rceil$ cycles to perform the same work. This shows

that a linear decrease in utilization corresponds to a linear decrease in speed.

### 2.2.2 Flexible Recursion Length

The second modification allows all 10 CCSDS telemetry codes to use the same encoder hardware. The traditional LDPC RCE encoder implementation uses $b = (n-k)/m$ distinct RCEs, each of which has a recursion length of $m$.

Instead, this implementation uses a single shift register with the length $(n-k)$ of the largest supported code, then inserts optional recursion points at specific lengths. The result is a runtime-configurable recursion length that can be modified to support the any of the ten codes.

An example flexible shift register is shown in Figure 2.4, where each square is a D flip-flop, with multiplexers interspersed along the chain. Depending on which multiplexer inputs are selected, this circuit can be four shift registers of length 4, two shift registers of length 8, or one shift register of length 16. Therefore this shift register could be used in a flexible RCE of length 16, and would create run-time configurable RCEs with lengths of $m$=4, 8, or 16.

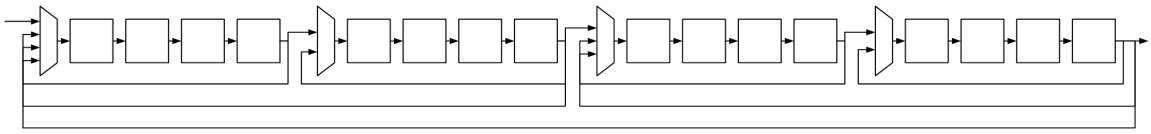


Fig. 2.4: A Flexible-Length Shift Register for use in a Flexible RCE

Since all cycle lengths with the exception of the rate 7/8 code are powers of two, the implementation is straightforward and can be created using a VHDL generate-for loop. In order to support the rate 7/8 code, additional multiplexers are inserted to enable a recursion length of 511.

## 2.3 Memory Organization

This section describes the memory organization used to create a low-resource encoder.

### 2.3.1 Generator Matrix ROM

BRAM memory was chosen for the matrix ROM for several reasons:

- Even just storing the first row of each block-circulant, the memory requirements are about 540 kb. This is too much for distributed memory, and function generators would also become too complex.
- BRAM can have very wide data buses. We use memory with data widths of 512 to 2048 bits.
- The Iris radio has a lot of unused BRAM.

The ROM has a read width equal to the size of the flexible RCE, and as many addresses as necessary to store each code.

To store a code, the generator matrix is reduced to the first rows of each circulant block. Columns corresponding to systematic and punctured bits are discarded. The remaining columns are bit-reversed so that the column of the first parity bit is on the right-hand side, allowing the parity bits to be read out of the RCE shift register when computations are finished. These rows are then broken into chunks equal to the maximum size of the flexible RCE.

For some of the shorter block-length codes, the number of parity bits $(n - k)$ may be less than the maximum size of the flexible RCE. In this case, the row is right-aligned in memory, with padded zeroes on the left.

### 2.3.2 Working Memory

The working memory must be capable of reading and writing large data widths each cycle, so dual-port BRAM was chosen. This could have been implemented using distributed RAM, but Iris has a shortage of free LUTs and plenty of unused BRAM.

The AR4JA codes have a maximum serialization factor of 8, so the working memory must hold at least 8 addresses.

## 2.4 Code Support

Table 2.1 breaks down the number of parity bits ($n-k$), cyclic length ($m$), and number of RCEs ($b$) for each of the CCSDS telemetry codes.

Table 2.1: Cyclic Lengths for CCSDS Telemetry Generator Matrices

| **Code** | **n − k** | **m** | **b** |
|---|---|---|---|
| R=1/2, k=1024 | 1024 | 128 | 8 |
| R=2/3, k=1024 | 512 | 64 | 8 |
| R=4/5, k=1024 | 256 | 32 | 8 |
| R=1/2, k=4096 | 4096 | 512 | 8 |
| R=2/3, k=4096 | 2048 | 256 | 8 |
| R=4/5, k=4096 | 1024 | 128 | 8 |
| R=1/2, k=16384 | 16834 | 2048 | 8 |
| R=2/3, k=16384 | 8192 | 1024 | 8 |
| R=4/5, k=16384 | 4096 | 512 | 8 |
| R=7/8, k=7136 | 1022 | 511 | 2 |

The original goal was to support all ten CCSDS telemetry codes. In order to keep resource utilization low, for this implementation a flexible RCE of length 512 was created, which enables support for the k=1024, 4096 codes, and the rate 7/8 code. In order to support the larger block length codes, an RCE of at least length 2048 would have to be used, which uses at least four times the resources. The generator matrices for the k=16384 codes are also rather large, and require many more BRAMs to store. Storing only the k=16384 codes uses 28 BRAMs, whereas all the other codes together, including the rate 7/8 code, use just 8 BRAMS. Since the performance between the $k = 4096$ and $k = 16384$ codes is very similar, the decision was made to drop support for the $k = 16384$ codes for the present.

The AR4JA codes have power-of-two cyclic block lengths $m$, and so are easy to implement in the flexible RCE. They also have $4m$ punctured bits. These punctured bits are simply not calculated, and the associated columns of the generator matrix are not stored.

The rate 7/8 codeword starts with 18 virtual zeros that are encoded, but not transmitted. These virtual zeros are handled by simply starting the cycle counter at 18 instead of

0 for the rate 7/8 case. After encoding is finished, the encoder state machine outputs two extra zero-fill bits, as per the code specification.

# CHAPTER 3

# DECODER DESIGN

Many LDPC decoding algorithms exist, and each algorithm has variations and parameters that can be used to decrease complexity or increase performance. When implemented in hardware, decoding algorithms may have several feasible high-level architectures, each with its own trade-offs [20]. Within each architecture, design choices such as the number of bits used for soft decoding or the implementation of a particular function create an even larger solution space.

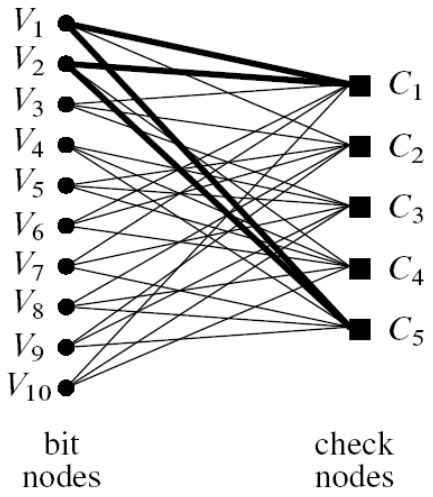
Some studies have been performed surveying the various LDPC decoding methods [20], others describe methods of reducing complexity [21,22], and many resources exist describing existing decoding algorithms [23, 24]. However, these resources do not optimize for specific codes, but rather describe generalized algorithms. This chapter applies the results of this prior work to the CCSDS telecommand LDPC codes.

## 3.1 Overview of Decoding Algorithms

Modern LDPC decoding algorithms are generally based on the belief propagation (BP) message-passing algorithm. Belief propagation is an iterative algorithm that uses received soft symbols to calculate probabilities or log-likelihood ratios for each bit, then uses those probabilities to update parity check information, then uses that parity check information to update the bit probabilities, and so on. Eventually the algorithm converges on the closest codeword.

This decoding process can be visualized at a high level using a Tanner graph, as shown in Figure 3.1 with an example PCM. Each column of the PCM corresponds to a variable (bit) node, and each row corresponds to a check node. The 1s in the PCM represent edges of the graph connecting the nodes.

The variable nodes are initialized using the channel inputs. Each check node calculates




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

Fig. 3.1: Tanner Graph Representation of an Example Parity Check Matrix (from [5])

the bit probabilities for each of the variable nodes connected to it. The variable nodes use this updated information to produce new soft bit estimates. This process continues, with information "sloshing" back and forth between variable and check nodes until a stopping criteria is met. For LDPC codes, the stopping criteria is usually the codeword criteria (i.e. once a correct codeword is produced, decoding stops), but other stopping criteria have been proposed which can detect convergence and cycles [25].

The details and derivation of the decoding process have been covered extensively [5], and will not be related in their entirety here. One part of the decoding process that stands out is the Check Node Update (CNU) rule used. The CNU governs how input messages from the variable nodes are translated into output messages of the check nodes.

The optimal CNU in a cycle-free graph uses an operation known as the "tanh rule", since it involves computations of the hyperbolic tangent and its inverse. This expensive operation is computed between each of the check node inputs minus one, for each of the inputs. When the tanh rule is used, the decoder is referred to as a *sum product* (SP) decoder.

A common, simplified approximation to the sum-product decoder is the *min-sum* (MS) decoder. This decoder uses a CNU algorithm in which the smallest input (disregarding the

input to which the output belongs) is used as the output message, with its sign modified based on the product of all the inputs. This decoder is much easier to implement, but comes at the cost of a fraction of a dB loss in coding gain. The CNU implementation is discussed in Section 3.2.

The Variable Node Update (VNU) is generally the same for each variation of the belief propagation algorithm. The output messages each consist of a simple leave-one-out sum of each of the inputs. Its implementation is discussed in Section 3.3.

A further simplified decoding algorithm is the bit-flipping algorithm [26]. This decoder uses hard symbols to achieve very high throughputs at the expense of performance. Because performance is important and high throughputs are not required for this application, a bit-flipping decoder was not implemented.

For this project, the min-sum algorithm was select for implementation due to its speed a relatively good performance.

### 3.2 Check Node Update

Each of this project's decoder implementations uses the min-sum (MS) algorithm. The CNU for the MS algorithm is given by

$$L_{c\to v} = \left(\prod_{v'} \operatorname{sign}(L_{v'\to c})\right) \min_{v'} |L_{v'\to c}| \tag{3.1}$$

where $v' \in \mathcal{V}(c) - \{v\}$. This has two separable parts: the sign and the magnitude.

The output $L_{c\to v}$ message sign is the product of all the input messages' signs, excluding one. Since each message is represented as a two's complement binary word, the sign bits of the corresponding messages can be simply XORed together to produce the output sign.

The target code has row weights of eight. Performing an XOR on seven bits for each of the eight messages would require 48 operations. Instead, an XOR operation is performed between each of the eight inputs to find the total sign, then at the output this product is XORed once more by the corresponding message input, resulting in just 15 operations (see Figure 3.2).

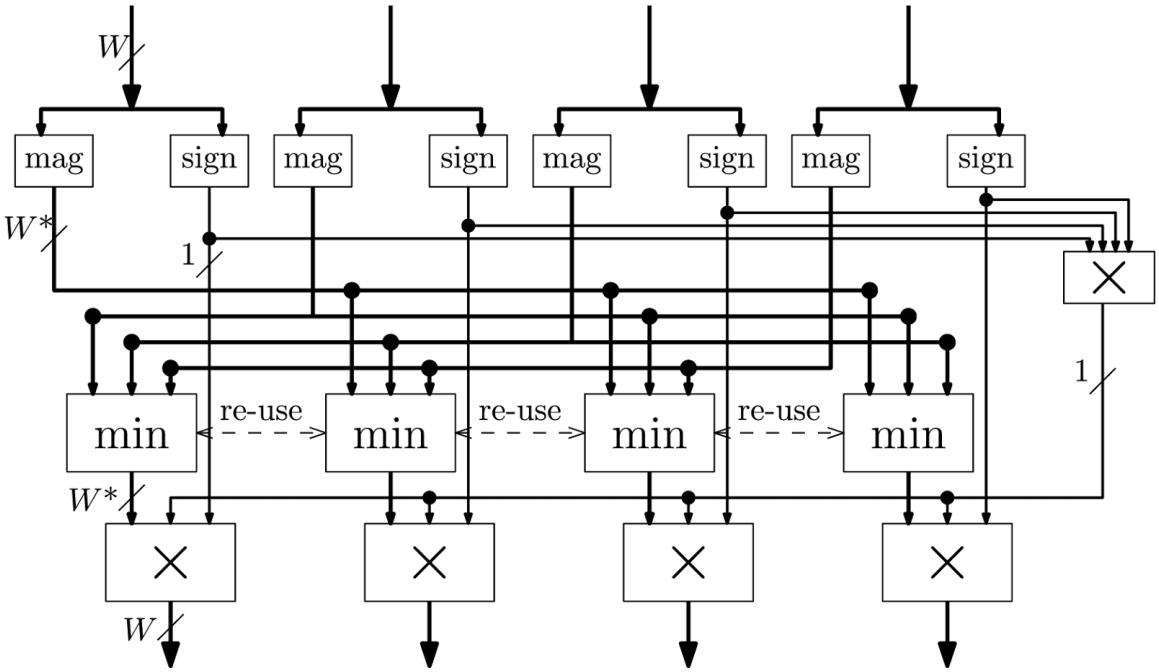



Fig. 3.2: Check Node Update Architecture (from [6]).

Careful analysis of equation (3.1) shows that for a set of given input messages, the output magnitude is either the minimum or sub-minimum of the inputs. Rather than performing a minimum search eight times, a single minimum, sub-minimum search can be used [7, 27–29].

With only eight inputs, this search can be performed in a tree-like fashion with three layers as in Figure 3.3. This allows the entire CNU to operate in a single clock cycle.

### 3.3 Variable Node Update

The variable node update (VNU) is the same for both the MS and SP algorithm. The update is described by

$$L_{v\to c} = L_v + \sum_{c'\in\mathcal{C}(v)-\{c\}} L_{c'\to v} \tag{3.2}$$

Each output message is simply the addition of all the input messages (excluding the one from the destination check node) and the received channel LLR. Because addition is a reversible operation (an added number can later be subtracted without loss of information)

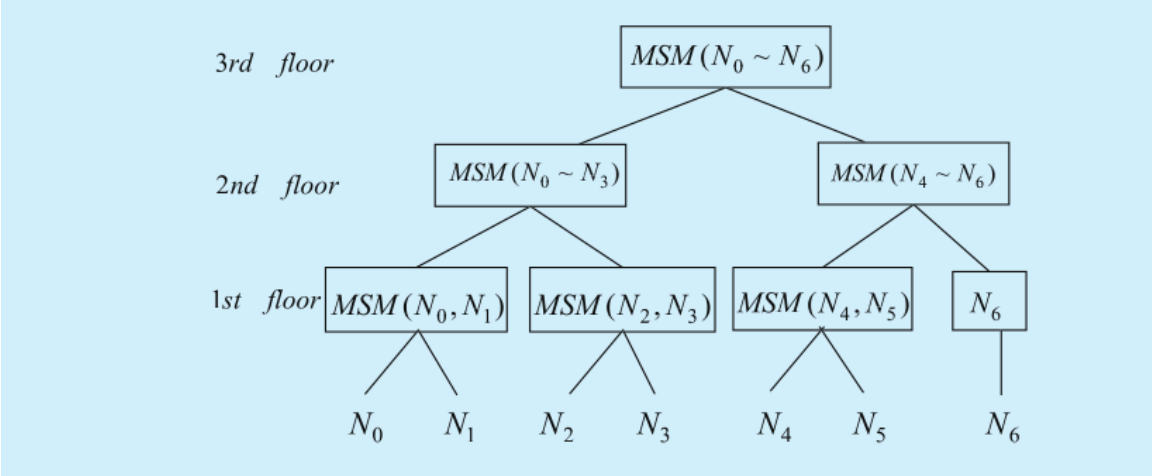



Fig. 3.3: Tree Structure of Min, Sub-Min Calculation (from [7]).

this can be efficiently computed using an adder tree as in Figure 3.4. The target code has row weights of three and five, so the tree needs at most three levels.

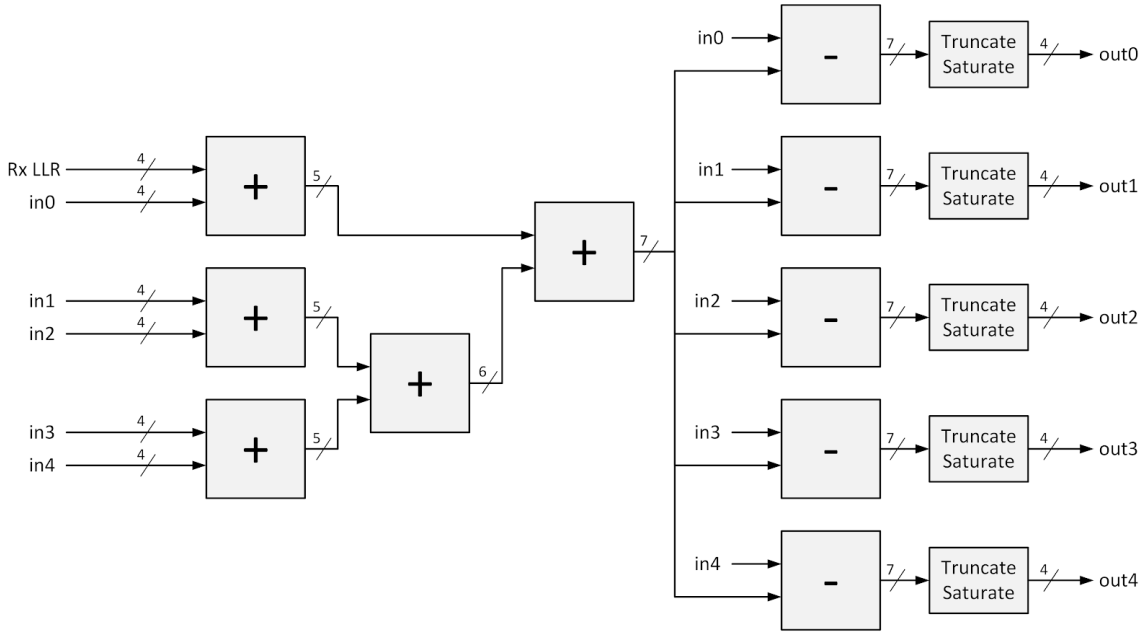



Fig. 3.4: Tree Structure of Variable Node Operations

First, all the inputs, including the channel input, are summed using the adder tree. Then, at each of the outputs the corresponding input message is subtracted from the total.

In order to avoid overflow, each level of the tree adds an extra bit of bit growth. The final results are truncated with saturation back to 4 bits. The effect of truncation and saturation on the decoder performance is explored in [30] and Section 4.2.2.

## 3.4 Architectures

Several implementation architectures exist for each decoding algorithm. These architectures generally have similar performance, but trade between speed and resource usage.

For this project, three architectures were chosen to implement the min-sum decoding algorithm. First is a fully parallel decoder (Section 3.4.1) which is extremely fast, but also extremely resource intensive. Next is a partially parallel architecture (Section 3.4.2) which performs smaller groups of operations simultaneously. Last is a fully serial architecture (Section 3.4.3) which performs all computations serially, but uses the fewest resources.

### 3.4.1 Fully Parallel Decoder

The fully parallel decoder is the simplest conceptually. This decoder instantiates all check nodes and all variables nodes in hardware, along with all the interconnections between them. Figure 3.5 shows how each variable node and check node are arranged. The interconnections between them follow the PCM for the code, and the channel inputs are used to initialize the variable nodes. Every VNU is computed simultaneously, followed by every CNU. This architecture is the most expensive resource-wise, but is also the fastest.

The fully parallel decoder follows a flooding schedule. First, the variable nodes are initialized with the received LLR values. Then, each parity check node runs simultaneously, sending updated messages back to the variable nodes. Next, each variable node runs simultaneously, sending updated messages back to the check nodes. Both check node and variable node updates take a single cycle each, so a full decoding iteration can be accomplished in just two clock cycles. Compare this to a general CPU software implementation, where a single iteration can easily take tens of thousands of clock cycles.

This speed comes at the expense of a high resource usage. Each check and variable node takes up space in the FPGA fabric. Additionally, the routing between then is nontrivial.

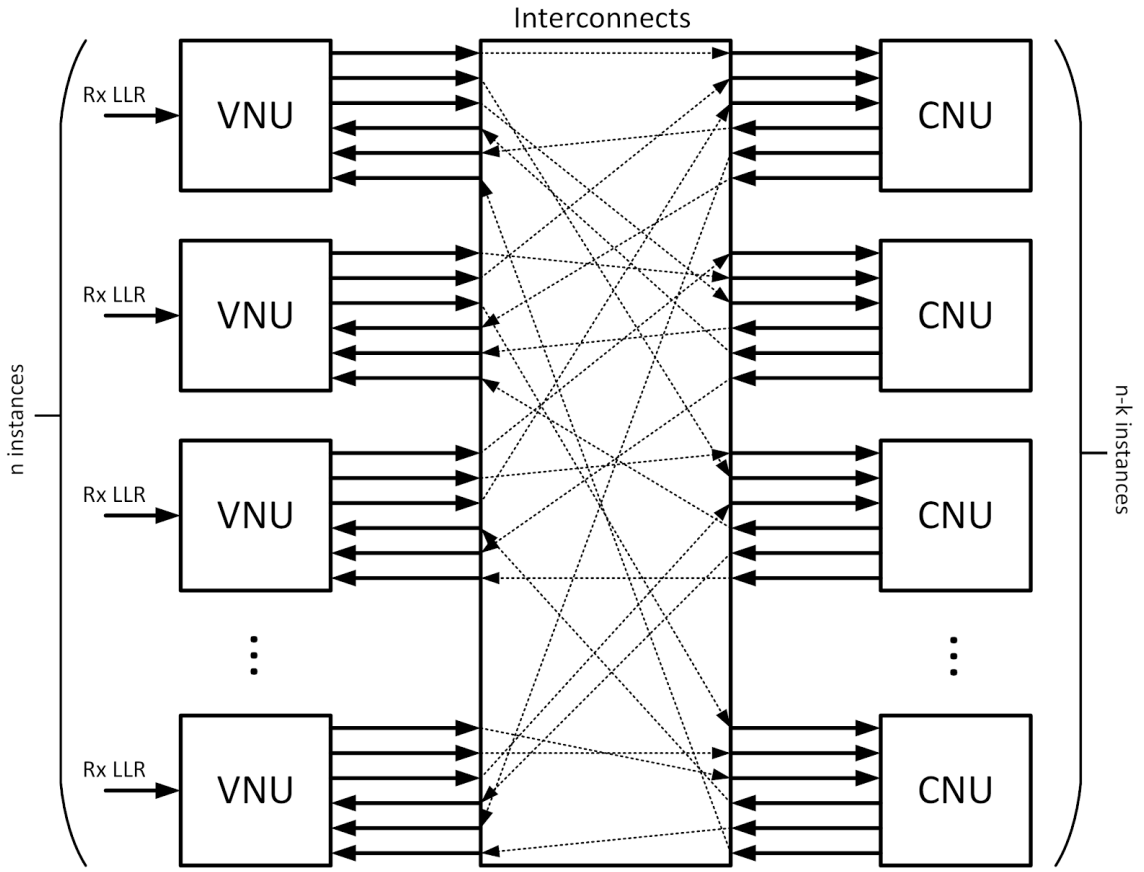



Fig. 3.5: Fully Parallel LDPC Decoder

As the size of the parity check matrix grows, the interconnect routing quickly increases to become a congested mass of wires with large critical paths and large power dissipation. Because of this, a fully parallel decoder is not suitable for large codes. Fortunately, for the CCSDS short block length code ($n = 128, k = 64$), a fully parallel implementation fits in the FPGA. The larger ($n = 512, k = 256$) code was not implemented.

One big implementation advantage of the fully parallel decoder is the simplicity of memory organization. For the other two architectures implemented, memory management takes up a significant fraction of the resource utilization. In the fully-parallel case, no extra hardware is required to manage the memory, since each node contains just a single register for its outgoing message.

The fully parallel implementation used the min-sum CNU algorithm. One big downside

of having so many check nodes instantiated, is that a single change to the check node update computation has an enormous impact on resource utilization. For example, switching from the MS algorithm to the SP algorithm would incur the difference in resource utilization per CNU by $n - k = 64$ or $n - k = 256$ times!

Another disadvantage of the fully-parallel decoder is that it is fixed to just a single parity check matrix. The node interconnects can be re-routed by reprogramming the FPGA, but at run-time they are fixed, with no flexibility. If a family of codes shares parts of the the same PCM this can be exploited, but the CCSDS telecommand codes do not share any part of the same PCM.

### 3.4.2 Partially Parallel Decoder

The partially-parallel decoder takes the middle ground of the three architectures. Like the fully-parallel architecture, several check nodes and variable nodes are instantiated in parallel. However, only a chosen subset of the total number is instantiated. This gives the designer more control over the speed/resource usage trade-off, since the number of nodes instantiated can be chosen.

This implementation closely followed the one described in [6]. Figures 3.6 and 3.7 show the variable node and check node update paths, respectively. Messages are stored in block RAM. In this implementation, 16 messages can be read and written simultaneously.

This architecture follows a different schedule than the other two. Rather than updating the check nodes, then updating the variable nodes separately afterwards, check nodes and variable nodes are updated simultaneously, block by block. This is possible because the of the intermediate message storage between updates. The result is that some of the check node updates' messages are newly updated, while others are not. While not ideal, this allows check node and variable node updates to happen together, resulting in a 2x speedup. A small performance penalty comes from using this schedule.

The main challenge of the partially-parallel decoder is the memory access architecture. In the case of the CCSDS codes, the parity check matrices are block-circulant, and this can be taken advantage of in the memory architecture. For this reason, this type of architecture

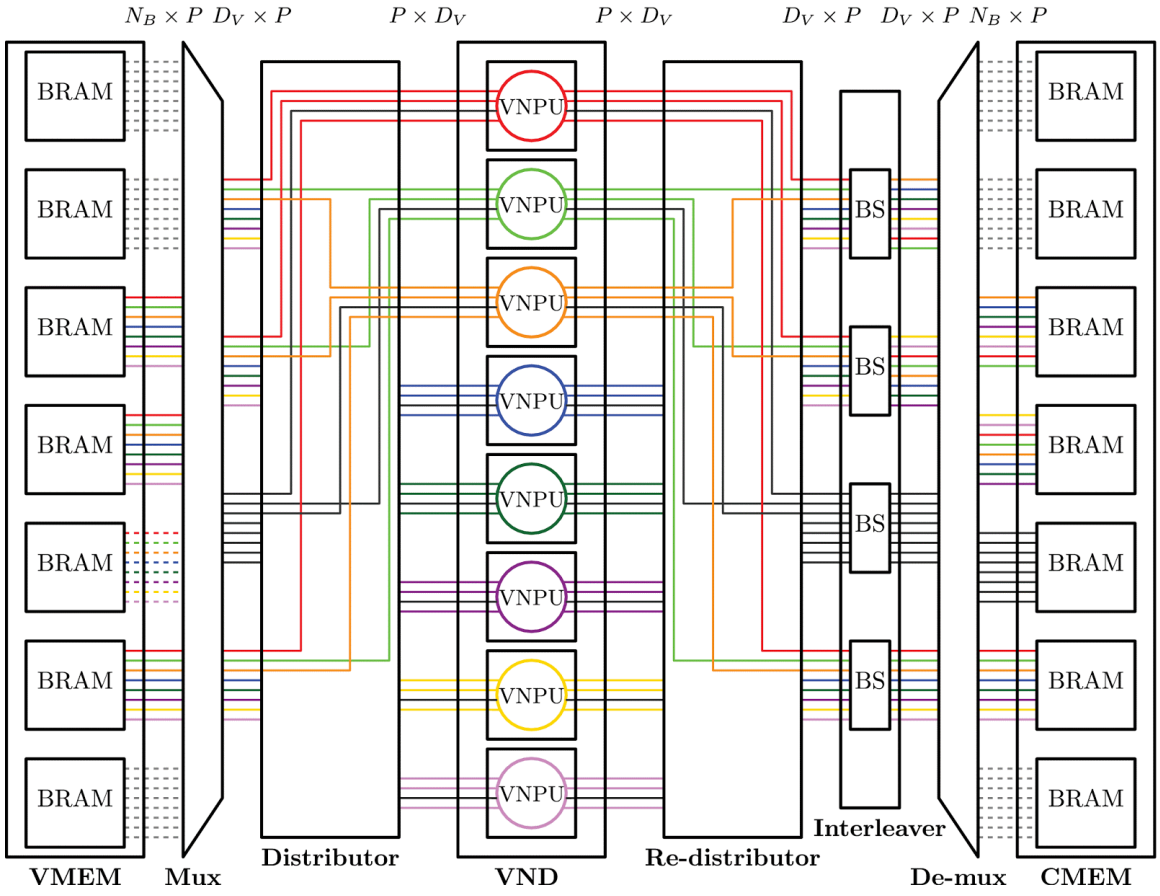



Fig. 3.6: Partially Parallel Variable Node Update (from [6]).

is generally only used for codes with block-circulant parity check matrices, otherwise the complexity of the memory accesses mitigates any benefit of the parallelism. For example, if we let the number of instantiated nodes equal the size of a circulant sub-matrix of the PCM, all the associated messages can be read in blocks of 16 (or whatever the circulant size is). Block RAMs are used to efficiently store message results and provide wide data buses.

The partially-parallel architecture can be flexible, supporting several different quasi-cyclic codes at runtime. Only the read and write addresses and the interleaver barrel shifter settings change between codes. In this project, however, only the $(n = 128, k = 64)$ code was implemented, with 16 VNUs and 8 CNUs (since there are half as many CNU computations to perform). The result can perform a full iteration in just $n/16 = 8$ cycles.

The architecture described in [6] was designed for block-circulant PCMs where each

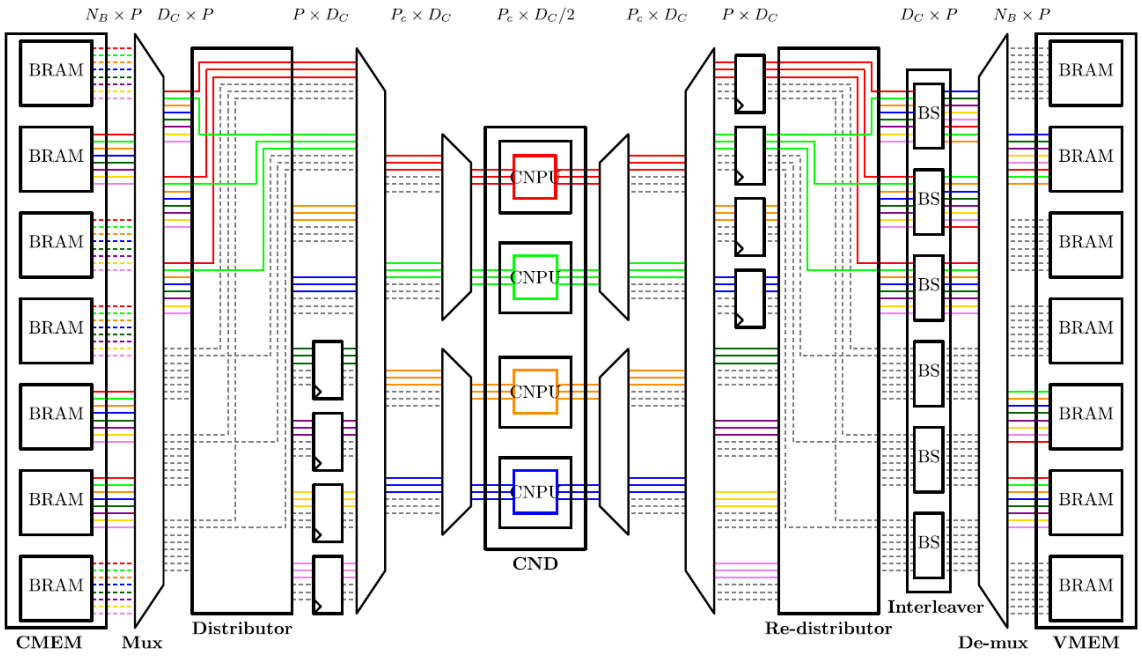



Fig. 3.7: Partially Parallel Check Node Update (from [6]).

circulant submatrix is a permutation matrix (has a column and row weight of one). The CCSDS telecommand code circulant submatrices have weights of 0, 1, or 2. The memory access architecture had to be modified slightly to support this different code.

### 3.4.3 Fully Serial Decoder

The fully serial decoder is on the opposite end of the spectrum from the fully parallel decoder. It instantiates just a single check node and a single variable node. This decoder also follows a flooding schedule, but in this case each update is computed sequentially in time. First, each variable node is initialized with the received LLRs, one at a time. Next, each parity check node runs sequentially, reading the required variable node messages from memory, then writing its outputs back into memory. Then, each variable node runs sequentially, reading check node messages from memory and writing output messages back into memory. Two memory banks are needed: one for check node messages and one for variable node messages.

This specific implementation was created specifically for the CCSDS code. The lessons

learned from implementing and modifying the partially parallel decoder were used to design the memory architecture. The first implementation used a serial memory architecture which was slow and had a high overhead. The second implementation documented here is much improved and runs several times faster, while requiring significantly fewer resources.

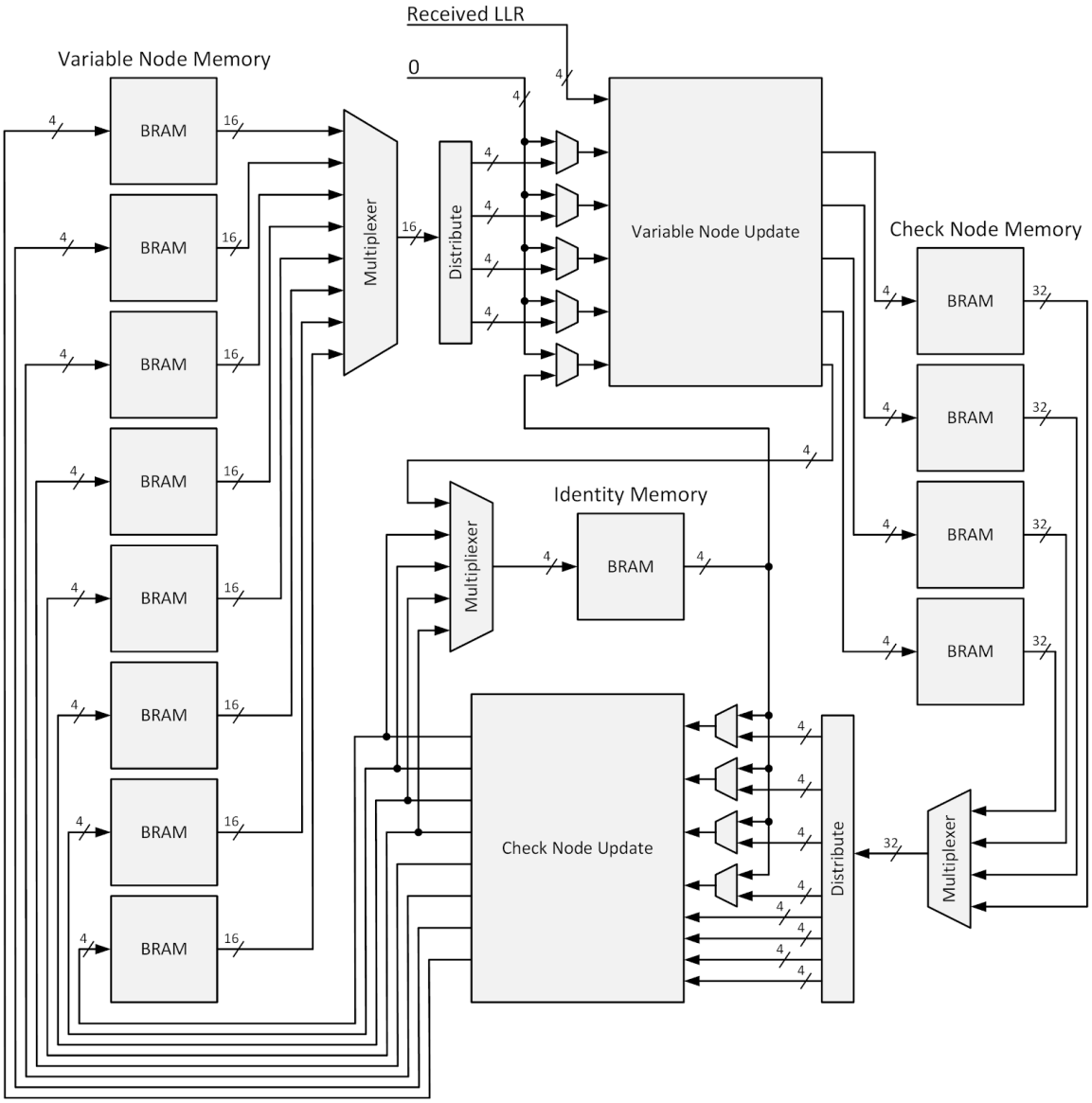



Fig. 3.8: Full Serial Decoder Architecture

Figure 3.8 shows an overview of the architecture. The major design challenge of the

fully serial decoder, like the partially parallel one, is in the memory architecture. How does a check node know which variable node messages to use as input, and how does it know where to store them?

The implemented decoder takes advantage of the protograph structure of the parity check matrix (see Figure 1.3). The variable node memory bank has 8 BRAM memories for the 8 variable nodes in the protograph, and the check node memory bank has 4 BRAM memories for the 4 check nodes in the protograph. These BRAMs have read port widths sufficient to read all the messages required for a single node update simultaneously, and write port widths to write a single message at a time. For the CCSDS code and 4-bit messages, the VMEM (variable node memory) BRAMs have read ports of 16 bits (four messages) and write ports of 4 bits. The CMEM (check node memory) BRAMs have read ports of 32 bits (eight messages) and write ports of 4 bits.

Variable and check node updates are pipelined. Each message update requires three cycles, but the pipelined throughput is one update per cycle. For example, in order to perform the variable node update for variable node $v$:

1. In the first cycle, all the required input messages $L_{c' \to v}$, where $c' \in \mathcal{C}(v)$ are read simultaneously from a single VMEM BRAM.

2. In the second cycle, the chosen BRAM output is selected using a multiplexer. In this implementation, the 16-bit data contains four 4-bit messages, which are input to the VNU, along with the channel LLR input. The VNU performs its update as described in Section 3.3.

3. In the third cycle, the VNU outputs $L_{v \to c'}$. Each of the messages is written to a separate CMEM BRAM in appropriate locations.

The pipelining means that while messages from variable node $v$ are being written into CMEM, the VNU is operating on messages for variable node $v+1$, and messages for $v+2$ are being read from VMEM. The operations for a check node update follow those of a variable node update.

An example quasi-cyclic parity check matrix with $8 \times 8$ submatrices is shown in Figure 3.9. Filled squares represent non-zero entries of the PCM. This matrix is smaller than the CCSDS-defined PCMs, but will be used to demonstrate the memory organization. Each check-node memory block (labeled CMEM $X$) holds one block-row of the PCM, and each variable-node memory block (labeled VMEM $X$) holds one block-column of the PCM. Each read address of a VMEM block contains a single column's worth of messages: all the messages required to compute a variable-node update.

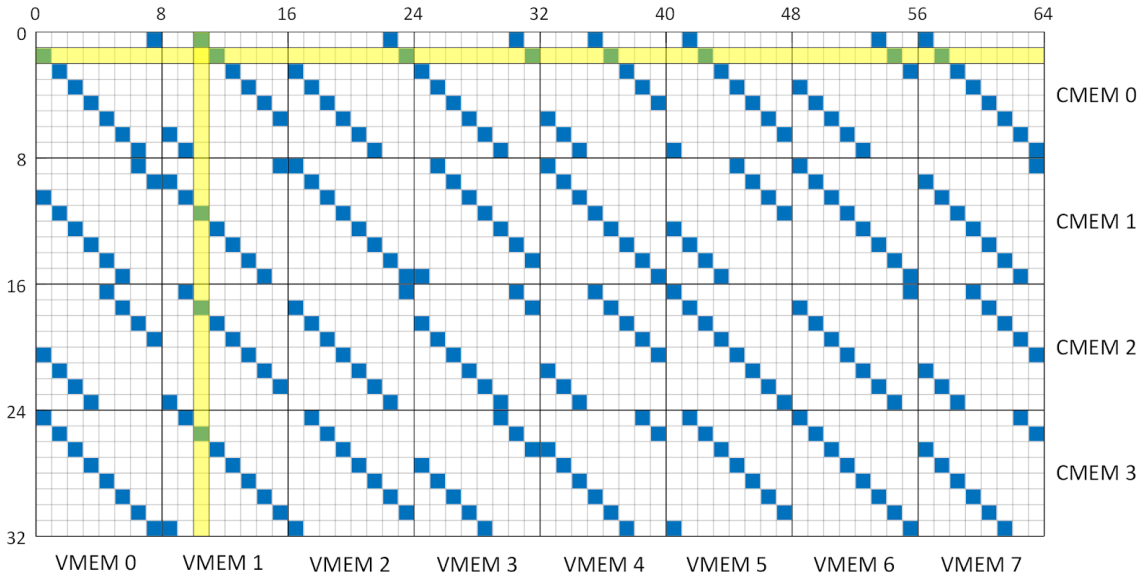



Fig. 3.9: Example PCM During Decoding

For example, while updating variable node $v = 10$, the highlighted column in Figure 3.9 is read from address $v \bmod 8 = 2$ of VMEM $\lfloor v/8 \rfloor = 1$. The four messages contained therein are input to the VNU. The four VNU output messages are then written to address $v = 10$ of each of the CMEMs. Each CMEM is aware of the cyclic shifts of the submatrices it contains, and internally adjusts its write address using simple modulo-8 addition prior to writing.

While updating check node $c = 1$, address $c \bmod 8 = 1$ of CMEM $\lfloor c/8 \rfloor = 0$ is read. The eight messages contained therein are input to the CNU. The eight CNU output messages

are then written to address $c = 1$ of each of the VMEMs. Each VMEM is aware of the cyclic shifts of the submatrices it contains, and internally adjusts its write address using simple modulo-8 addition prior to writing.

The actual telecommand PCMs defined by the CCSDS are larger, and have extra properties. Some of the cyclic blocks are all zeros, and there is an extra diagonal of ones along the identity of the PCM, as shown in Figure 1.4.

To accommodate the occasional zero sub-matrix, each input to the variable node update module can be zeroed out using a multiplexer. To accommodate the extra ones on the identity, an extra BRAM is used. Since the ones are on the identity, their read and write addresses are the same for both the variable node and the check node updates. And since each update is performed sequentially, the same BRAM can be used for the check node and variable node messages (each is overwritten by the other). The output messages of this identity memory are multiplexed into the check node update inputs when appropriate.

The variable node hardware accommodates five input messages. When an update requiring only three messages is computed, the extra two inputs are zeroed out by the multiplexers in order to not affect the computation.

The fully-serial decoder has many advantages. First, it requires very few resources. There is the small overhead of the memory access architecture, but there is only a single instance of both the check and parity nodes.

The decoder supports both CCSDS telecommand codes with very little modification. Both the $(n = 128, k = 64)$ and $(n = 512, k = 256)$ codes have the same structure. The only difference is the size of the submatrices ($m = 16$ for the shorter code, and $m = 64$ for the longer) and the cyclic permutations of each of the submatrices. Since the BRAMs hold only the messages (and not the blank spaces in the sparce PCM), the port widths are the same. Both sets of permutations are stored in the CMEM and VMEM.

Another advantage of the fully-serial architecture is that the variable node or check node update designs have much less effect on the overall resource utilization. In the fully-

parallel case, any increased cost for the check node update was multiplied $n-k$ times. For the fully-serial decoder, the increase is only incurred once. This allows for a better check node design and better performance without greatly increasing resource usage.

The obvious drawback to a serial decoder is the slower speed. Even operating at a single cycle per update, it still requires at least $2n-k$ cycles to complete a single iteration. Because the CCSDS codes have short block lengths, the serial decoder is still feasible.

### 3.5 SNR Estimate

The belief-propagation algorithm requires LLRs to be computed from the received channel inputs. The received signal level for bit $i$ is $y_i = \alpha b_i + z_i$, where $\alpha$ is the signal amplitude ($|\alpha| = \sqrt{E_c}$), $b_i$ is either $+1$ or $-1$ based on the transmitted bit value, and $z_i$ is AWGN noise with power $\sigma^2 = N_0/2$. The associated LLR [5] is calculated by

$$L_i = \frac{2\alpha}{\sigma^2} y_i \tag{3.3}$$

This essentially requires knowledge of the signal-to-noise ratio (SNR), $\alpha^2/\sigma^2$. This information can be provided by the user, or it can be estimated.

One simple and effective way of estimating the SNR is by comparing received symbols to an expected pilot sequence. The CCSDS Communications Link Transmission Unit (CLTU) has a 64-bit start sequence when used with LDPC coding [1]. If we assume we have frame synchronization, then we know the $b_i$ for $i = 1 \ldots 64$ and we can obtain an estimate of $\alpha$ by

$$\hat{\alpha} = \frac{1}{64}\sum_{i=1}^{64} b_i y_i = \frac{1}{64}\sum_{i=1}^{64} \alpha b_i^2 + b_i z_i = \alpha + \frac{1}{64}\sum_{i=1}^{64} b_i z_i \approx \alpha \tag{3.4}$$

because $b_i^2 = 1$ and the last term is approximately zero due to the effect of averaging the noise.

The noise variance can also by estimated by observing that $z_i \approx y_i - \hat{\alpha} b_i$, so that

$$\hat{\sigma}^2 \approx \frac{1}{64}\sum_{i=1}^{64} z_i^2 \approx \frac{1}{64}\sum_{i=1}^{64} (y_i - \hat{\alpha} b_i)^2 \tag{3.5}$$

One of the difficulties with using this approach in a streaming architecture is the frame synchronization requirement. By the time the pilot symbols are known to have been received, the codeword is being received. This means there must be a delay while SNR estimation is performed, and memory to store the pilot symbols for later use.

The calculations of equations (3.4) and (3.5) can be simplified by assuming that no received symbols have enough noise to cross the zero decision threshold. This assumption is valid only at high SNRs, but analysis will show that it is good enough for this situation. With this assumption we see that

$$b_i y_i \approx |y_i| \tag{3.6}$$

Using equation (3.6), equation (3.4) becomes

$$\widetilde{\alpha} = \frac{1}{64}\sum_{i=1}^{64}|y_i| \tag{3.7}$$

which can be estimated using a simple absolute value circuit and an accumulate-and-dump filter. Equation (3.5) becomes

$$\begin{aligned}\widetilde{\sigma}^2 &= \frac{1}{64}\sum_{i=1}^{64}(y_i - \widetilde{\alpha}b_i)^2 = \frac{1}{64}\sum_{i=1}^{64}\left(y_i^2 - 2\widetilde{\alpha}y_i b_i + \widetilde{\alpha}^2 b_i^2\right) \\ &= \frac{1}{64}\sum_{i=1}^{64}y_i^2 - 2\widetilde{\alpha}\frac{1}{64}\sum_{i=1}^{64}|y_i| + \widetilde{\alpha}^2 = \frac{1}{64}\sum_{i=1}^{64}\left(y_i^2\right) - \widetilde{\alpha}^2\end{aligned} \tag{3.8}$$

which can be obtained from a multiplier, an accumulate-and-dump filter, and a sum. These combined improvements can be used to create a very efficient SNR estimator that works for higher SNRs, shown in Figure 3.10. In addition, the estimator no longer requires frame synchronization or pilot symbols! The estimator can be used in a streaming fashion on an arbitrary number of samples.

But what about the low SNR case? The CCSDS telecommand codes have a threshold of approximately 3.5 dB. Under simulation, the average estimate error and standard deviation of the estimate error are plotted for various values of Eb/N0 in 3.11.

We can see that the estimator begins to break down at about 5 dB. Lower than 0 dB

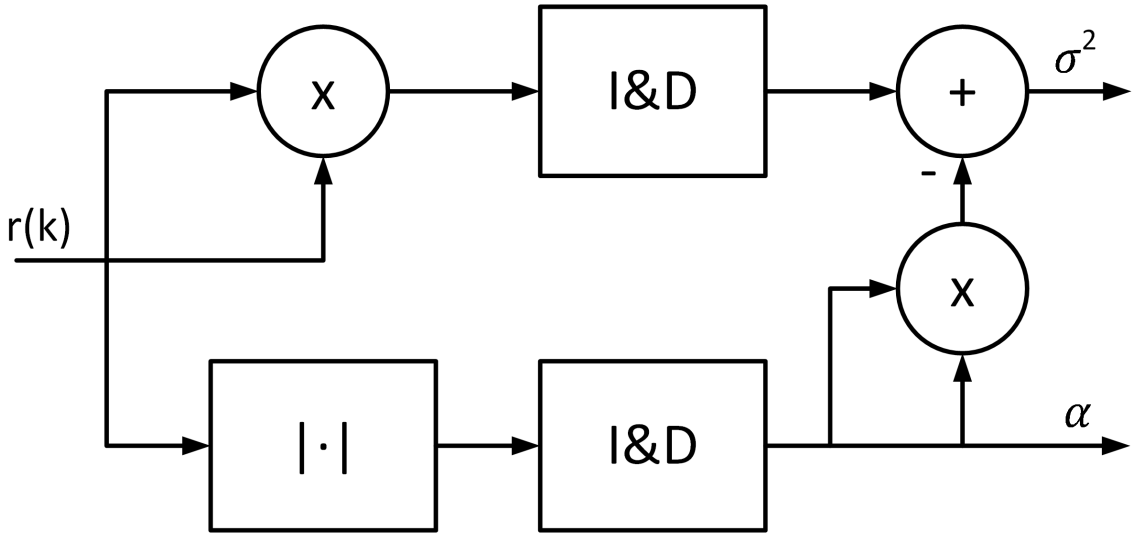



Fig. 3.10: Simple SNR Estimator for Input LLRs

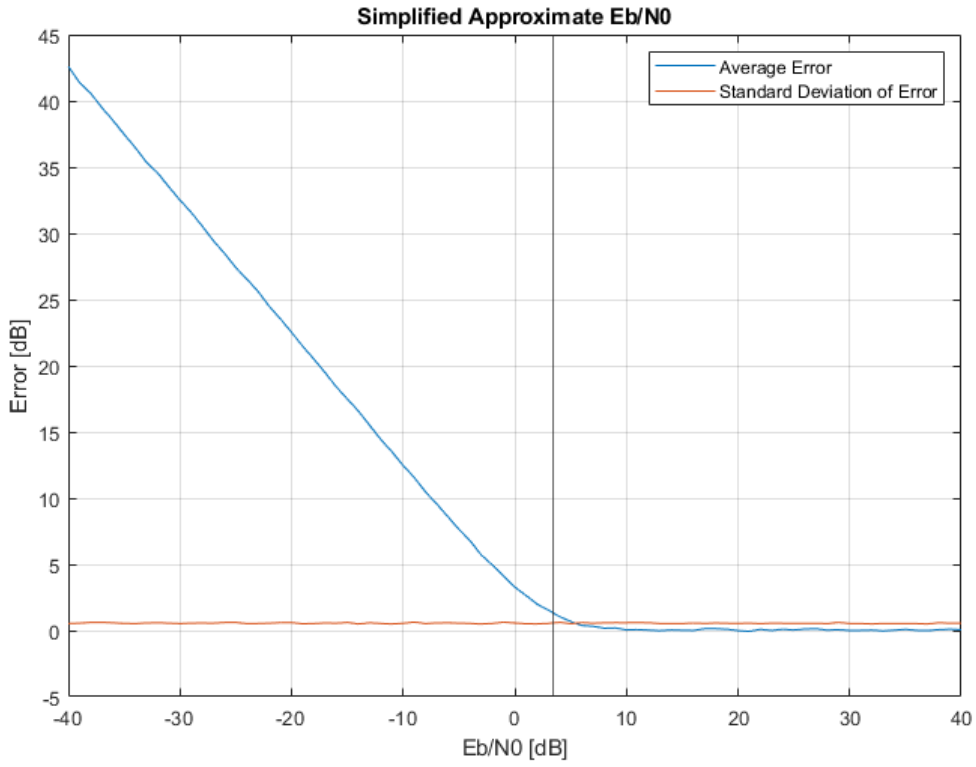



Fig. 3.11: SNR Estimator Average and Standard Deviation of Error

it has a significant bias in the error. How much does the bias at 3.5 dB and above affect the decoder? Bit error rate curves were generated by simulation in MATLAB and shown

in Figure 3.12 for the following cases:

- Accurate knowledge of the SNR (blue)
- Assume the SNR is always 20 dB, regardless of true SNR (red)
- A worst-case estimate using the simplified algorithm described above. The SNR estimate is (True SNR) + (average estimator error at that Eb/N0) + 2*(stddev of estimator error at that Eb/N0) (yellow).

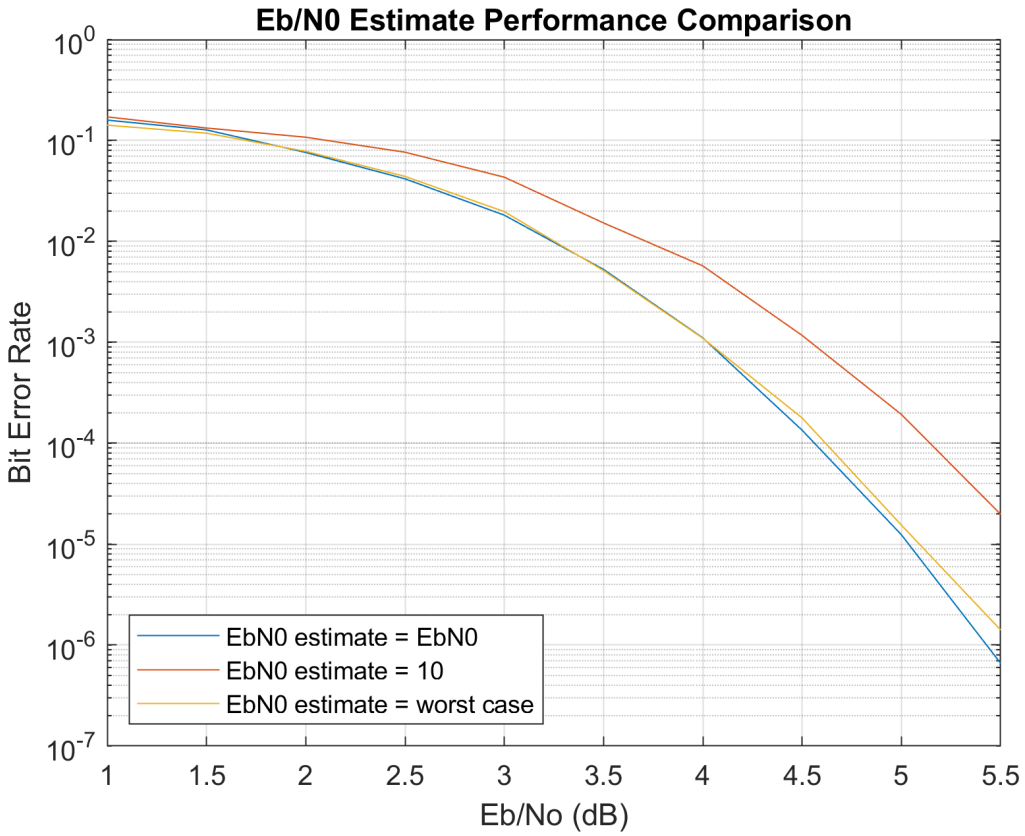



Fig. 3.12: BER Comparison Using Different SNR Estimators

There is a small fraction of a dB performance loss using this estimator. However, this is the worst case estimate, meaning that the estimator will always perform *at least* as good as indicated in the figure. This shows that the simplified estimator is sufficient for this code.

# CHAPTER 4

# RESULTS AND ANALYSIS

This chapter discusses the testing methods and results for both the telemetry encoder and telecommand decoders discussed in previous chapters.

## 4.1 Encoder Results

The encoder performance is measured using the metrics of correctness, speed, and size. These attributes are discussed in the following sections.

### 4.1.1 Testing Methods

In order to test correctness, the encoder was first tested with HDL simulation using Modelsim. After basic functionality was demonstrated with a sample message block of alternative ones and zeros, a full suite of tests was designed to demonstrate full functionality and code support for the seven supported codes.

The simulation testing suite consisted of ten tests for each supported code. Each test consisted of a single codeword, designed for easy debugging and thorough testing of edge cases. The ten test messages were:

- Iris blank frames (repeated 32-bit 0xdeadc0de words)
- All ones
- All zeros
- Alternating 0s and 1s
- Alternating 1s and 0s
- Five random messages

Once the correct encoding of messages was shown, the encoder was integrated into the Iris Transponder. A full-scale downlink test was then performed for each of the supported codes at various data rates. Each test consisted of encoding tens of thousands of random

codewords encoded, modulated, and transmitted, then received, demodulated, and decoded. The tests operated at a high SNR and the decoder was monitored for the number of bit errors in the messages. The pass criteria was for no bit errors to be present before decoding, indicating that each message block was properly encoded. These tests were repeated extensively at several data rates for each of the seven supported codes.

### 4.1.2 Utilization

The encoder utilization depends on the supported codes and the degree of serialization as described in Section 2.2.1. The final configuration used for Iris supported all the 1024 and 4096 block-length codes, as well as the 7136-bit rate 7/8 code. The flexible RCE had a length of 512. The final utilization count is listed in Table 4.1.

Table 4.1: Encoder Resource Utilization

| **Virtex6 Resource** | **Number used** | **Percent of FPGA** |
|---|---|---|
| Logic LUT | 645 | 0.81% |
| Memory LUT | 0 | 0.00% |
| FF | 586 | 0.37% |
| Block RAM 18K | 2 | 0.38% |
| Block RAM 36K | 14 | 5.30% |
| DSP48 | 0 | 0.00% |

In order to support all the length 16384 codes, the RCE length would have to be at least 2048, approximately quadrupling the utilization.

### 4.1.3 Speed

The final implementation used a flexible RCE of length 512. Because each code has a different block-cyclic length, the degree of serialization required is different for each code. The degree of serialization controls the maximum speed for a given clock rate.

The codes with fewer than 512 parity bits fit inside the 512 RCE bits, and so require a single clock cycle per input bit. The longest code has eight blocks of 512 parity bits, and so requires eight clock cycles per input bit. Given the Iris system clock frequency of 50

Table 4.2: Encoder Speed

| Code | Cycles per Input Bit | Max Speed (Coded Mbps) |
|---|---|---|
| R=1/2, k=1024 | 2 | 10 |
| R=2/3, k=1024 | 1 | 10 |
| R=4/5, k=1024 | 1 | 10 |
| R=1/2, k=4096 | 8 | 4.55 |
| R=2/3, k=4096 | 4 | 7.14 |
| R=4/5, k=4096 | 2 | 10 |
| R=7/8, k=7136 | 2 | 10 |

MHz, and the required number of clock cycles for the state machine to operate correctly, the maximum operating speed in coded Mbps is also listed in Table 4.2. At the time of this writing, the Iris transponder has an output symbol rate limit of 6.25 Msps, so each of the codes perform at the highest Iris speeds, except for the R=1/2, k=4096 code.

Due to the implementation of the controlling state machine, the smallest number of cycles between coded bits is 5, resulting in the maximum 10 Mbps speeds listed.

### 4.1.4 Trade-offs

The only variable parameters for the encoder are the speed and utilization. The encoding speed could be increased linearly by increasing the size of the RCE used (doubling the utilization would double the max speed).

### 4.1.5 Discussion

The final result is an encoder supporting seven of the ten LDPC telemetry codes at the maximum Iris rate, while using less than one percent of the FPGA resources.

The flexibility of the design means that future improvements, such as supporting the k=16384 codes, or increasing the maximum speed, are nearly trivial to implement and come at the expected linear cost of utilization.

Improvements to the controlling state machine could also be made to add some additional speed without much resource increase, if needed.

## 4.2 Decoder Results

The metrics of the decoder implementations are correctness, performance, speed, and size. These attributes are compared across the three implemented architectures in the following sections.

### 4.2.1 Testing Methods

The decoder was first tested in simulation for correctness. Python was used to generate 100 random codewords at a configurable SNR. These codewords were then decoded in the simulation. At high SNRs, the decoders were verified to detect correct codewords immediately. At lower SNRs, the expected distribution of iteration counts was observed.

Next, a hardware testbed was created. The ML605 Virtex6 development board was used, since it has an FPGA of the same family as the Iris radio. An interface was created to allow a host computer to transfer generated LLR inputs to the decoder over ethernet. The decoded results are transferred back to the host computer using the same ethernet interface.

This hardware testbed was used to generate bit error rate curves and utilization measurements for the three architectures and various configurations.

### 4.2.2 LLR Bit Width Comparison

The decoder implementation must truncate LLR messages to a finite bit width. The choice of bit width directly affects the decoder's performance [30]. In addition to the bit width of the messages, the number of fractional bits included in that width also affect the performance.

In order to characterize the effects of finite word size, the fully-parallel decoder was tested at various bit widths and fractional lengths and BER curves were generated for each version, shown in Figure 4.1. The notation used is X.Y, where X refers to the total number of bits in the two's complement representation, and Y refers to the number of fractional bits included. For example, an LLR using a 5.1 representation can represent values from -16 to 15.5, with an LSB resolution of 0.5. Increasing the fractional size to 5.2 represents values from -8 to 7.75 with an LSB resolution of 0.25.

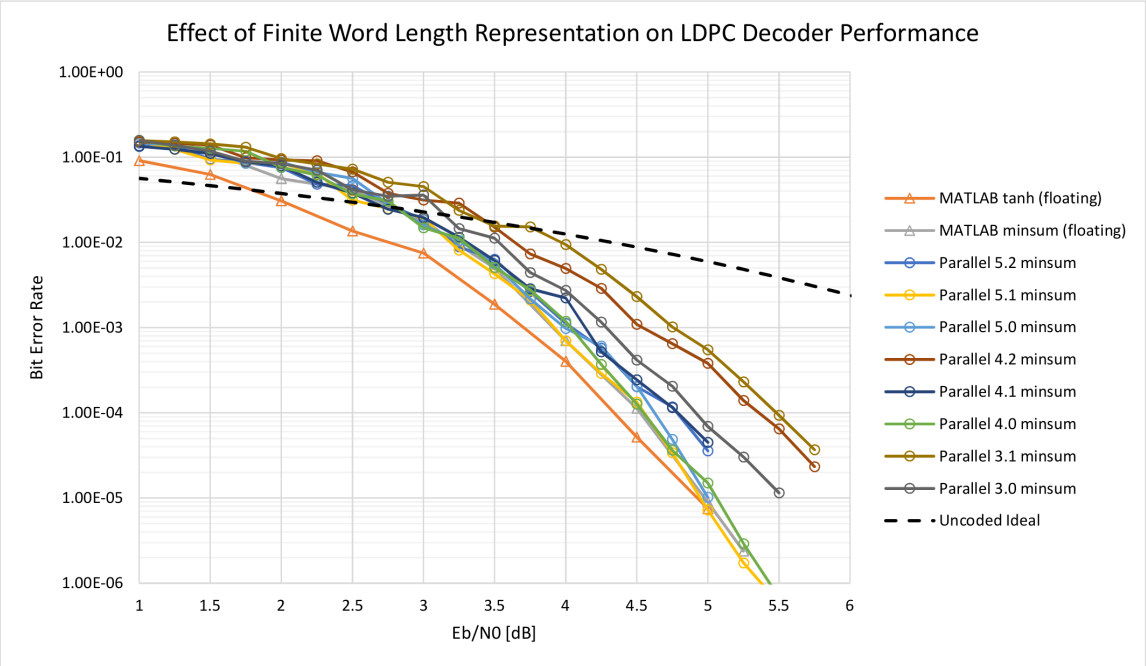



Fig. 4.1: Comparison of BERs for Various Word Configurations

The two MATLAB reference series in Figure 4.1 used floating point arithmetic. The best performing finite representation is 5.1, which performs nearly as well as the floating point representation. The next best is 4.0, which suffers an approximately 0.1 dB loss compared to the floating point MS reference. As a result of this study, 4.0 was chosen as the best representation for implementation, demonstrating excellent performance with a manageable word size.

### 4.2.3 Comparison of Architectures

Each of the architectures was implemented using an LLR word size of 4.0. The performance was characterized by generating BER curves, shown in Figure 4.2

The performance reference is a MATLAB simulation using the SP algorithm and floating point LLRs. The fully parallel decoder operates with an approximately 0.4 dB loss compared to the reference, and the partially parallel decoder operates with approximately 0.7 dB loss. The extra loss from the partially parallel decoder is likely due to its non-optimal schedule.

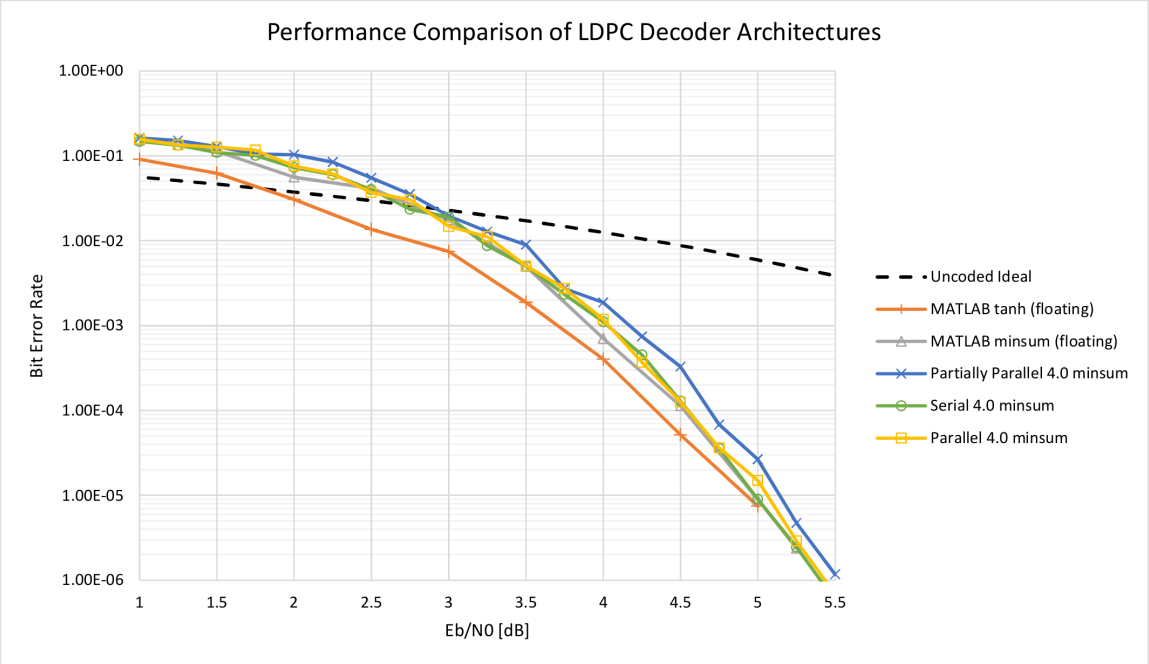



Fig. 4.2: Comparison of BERs for LDPC Decoder Architectures (TODO: 4.0 wordsize)

The resource utilization and maximum throughput for the three implementations are shown in Table 4.3. The throughput is measured as the maximum number of coded bits that can be decoded per second assuming an average of 25 iterations per codeword. The performance is measured in dB loss from the MATLAB reference (lower is better).

Table 4.3: Trade-off Comparison for Three Implementation Architectures

| | | **Fully Serial** | **Partially Parallel** | **Fully Parallel** |
|---|---|---|---|---|
| Utilization | Logic LUT | 1092 | 8910 | 17770 |
| | Memory LUT | 0 | 2048 | 0 |
| | FF | 754 | 3040 | 5332 |
| | Block RAMs | 9 | 0 | 0 |
| Throughput (Mbps) | | 1.333 | 32 | 128 |
| Performance (dB loss) | | 0.4 | 0.7 | 0.4 |

### 4.2.4 Discussion

All three architectures performed well, and each has its own strengths and weaknesses. As far as throughput per resource used is concerned, the fully parallel is the clear winner.

The fully parallel decoder requires only 2 clock cycles per iteration. The partially parallel decoder needs 8, and the fully serial needs $2n - k$ cycles per iteration.

The fully serial architecture was chosen for use on Iris for several reasons. First, it uses very few resources compared to the other architectures. In total it uses approximately 1% of the FPGA resources. Second, although it has much lower throughput than the others, Iris is currently limited to a 1 Mbps uplink symbol rate, so it performs faster than is required for Iris. It also performs better than the partially parallel implementation.

# CHAPTER 5
# CONCLUSION

## 5.1 Discussion

All of the objectives of this project were met. The design spaces of LDPC encoders and decoders were explored, especially relating to the CCSDS-defined codes. An LDPC encoder and several LDPC decoders were implemented with various trade-offs. The trade-offs were considered and specific implementations were chosen for inclusion on the Iris radio.

The original goal of supporting all 10 LDPC telemetry codes was met with an encoder supporting 7 of the 10 codes. The $k = 16384$ codes were deemed too costly to implement, given that they only perform a small fraction of a dB better than the $k = 4096$ codes. Adding support for the $k = 16384$ codes would require more than 4x the resources than the final chosen encoder.

## 5.2 Future Work

The encoder and decoder both meet the requirements for integration into the Iris radio platform. However, further work could be done to improve performance, increase speed, and decrease resource utilization.

First, the decoder remains to be integrated into Iris. The encoder has been fully integrated and extensively tested with promising results. The decoder will be integrated at a future date.

The encoder state machine is not fully optimized and consequently the maximum throughput is limited. Decreasing the number of cycles required between input bits would have a fairly significant increase in maximum speed, especially for the codes already requiring few clock cycles per bit.

Work could also be done to pipeline the encoder so that new messages can be computed

while the parity bits of the previous message are being transmitted. The speed under this schedule would increase by $1/R$.

The AR4JA structure of the telemetry codes was not taken advantage of for the encoder. Further work may find a way to exploit this structure to create a very fast, low-resource encoder. However, such an encoder would not be compatible with the rate 7/8 C2 code, and so supporting all codes would require an additional encoder.

The implemented decoders use a flooding or modified flooding schedule. Other decoding schedules exist, such as the serial schedule [31] and layered decoding [32–34], which can purportedly converge up to 50% more quickly, meaning that less logic is required to achieve the same throughput and error performance. Alternatively, the CNU and VNU operations of the flooding schedule could be made to overlap in the regions without memory access conflicts, increasing throughput [35].

The decoder message memory could be optimized. There are several schemes presented in the literature that take advantage of the fact that the min-sum check node output messages have one of two magnitudes [36,37]. Only those two magnitudes can be stored, along with the signs, then retrieved later. This scheme could reduce memory usage significantly. Other techniques in the literature could be explored and implemented [38].

The only CNU algorithm implemented was the min-sum algorithm. Many alternative CNU algorithms exist, such as min-sum offset and scale corrections [39,40], the product-sum algorithm [21], and more. Increases of the CNU utilization have a significant negative effect on utilization for the parallelized architectures, but for the serial architecture, the increase is only counted once. Performance could improve by at most 0.4 dB with a different CNU algorithm.

# REFERENCES


[1] *TC Synchronization and Channel Coding*, ser. Blue Book, No. 3, Consulative Committee for Space Data Systems (CCSDS) Recommendation for Space Data System Standard 231.0-B-3, Sep. 2017. [Online]. Available: https://public.ccsds.org/Pubs/231x0b3.pdf

[2] *TM Synchronization and Channel Coding—Summary of Concept and Rationale*, ser. Green Book, No. 3, Consulative Committee for Space Data Systems (CCSDS) Informational Report 130.1-G-3, Jun. 2020. [Online]. Available: https://public.ccsds.org/Pubs/130x1g3.pdf

[3] JPL, "Telemetry Data Decoding," in *Telecommunications Link Design Handbook*, ser. 810-005, Jan. 2013, no. 208. [Online]. Available: https://deepspace.jpl.nasa.gov/dsndocs/810-005/208/208B.pdf

[4] *TM Synchronization and Channel Coding*, ser. Blue Book, No. 3, Consulative Committee for Space Data Systems (CCSDS) Recommendation for Space Data System Standard 131.0-B-3, Sep. 2017. [Online]. Available: https://public.ccsds.org/Pubs/131x0b3e1.pdf

[5] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. New York, NY, USA: Wiley-Interscience, 2005.

[6] P. Hailes, L. Xu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "A Flexible FPGA-Based Quasi-Cyclic LDPC Decoder," *IEEE Access*, vol. 5, pp. 20 965–20 984, 2017.

[7] Y. Wang, D. Liu, L. Sun, and S. Wu, "Real-time implementation for reduced-complexity LDPC decoder in satellite communication," *China Communications*, vol. 11, no. 12, pp. 94–104, Dec. 2014.

[8] R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962.

[9] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, Jul. 1948.

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

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

[12] Y. He, J. Yang, and J. Song, "A survey of error floor of LDPC codes," in *2011 6th International ICST Conference on Communications and Networking in China (CHINACOM)*, Aug. 2011, pp. 61–64.

[13] T. Richardson and R. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 638–656, Feb. 2001.

[14] J. Thorpe, "Low-Density Parity-Check (LDPC) Codes Constructed from Protographs," *Interplanetary Network Progress Report*, vol. 154, pp. 1–7, Aug. 2003. [Online]. Available: http://adsabs.harvard.edu/abs/2003IPNPR.154C...1T

[15] *Next Generation Upink*, ser. Green Book, No. 1, Consulative Committee for Space Data Systems (CCSDS) Informational Report 230.2-G-1, Jul. 2014. [Online]. Available: https://public.ccsds.org/Pubs/230x2g1.pdf

[16] A. Abbasfar, D. Divsalar, and K. Yao, "Accumulate repeat accumulate codes," in *International Symposium onInformation Theory, 2004. ISIT 2004. Proceedings.*, Jun. 2004, pp. 505–.

[17] D. Divsalar, S. Dolinar, and C. Jones, "Low-rate LDPC codes with simple protograph structure," in *Proceedings. International Symposium on Information Theory, 2005. ISIT 2005.*, Sep. 2005, pp. 1622–1626, iSSN: 2157-8117.

[18] D. Theodoropoulos, N. Kranitis, and A. Paschalis, "An efficient LDPC encoder architecture for space applications," in *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Jul. 2016, pp. 149–154, iSSN: 1942-9401.

[19] K. Andrews, S. Dolinar, and J. Thorpe, "Encoders for block-circulant LDPC codes," in *Proceedings. International Symposium on Information Theory, 2005. ISIT 2005.*, Sep. 2005, pp. 2300–2304, iSSN: 2157-8117.

[20] P. Hailes, L. Xu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "A Survey of FPGA-Based LDPC Decoders," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 2, pp. 1098–1122, 2016.

[21] X.-Y. Hu, E. Eleftheriou, D.-M. Arnold, and A. Dholakia, "Efficient implementations of the sum-product algorithm for decoding LDPC codes," in *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No.01CH37270)*, vol. 2, Nov. 2001, pp. 1036–1036E vol.2, iSSN: null.

[22] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X.-Y. Hu, "Reduced-complexity decoding of LDPC codes," *IEEE Trans. Commun.*, vol. 53, no. 8, pp. 1288–1299, Aug. 2005.

[23] V. Savin, "Chapter 4 - LDPC Decoders," in *Academic Press Library in Mobile and Wireless Communications*, D. Declerq, M. Fossorier, and E. Biglieri, Eds. Oxford: Academic Press, Jan. 2014, pp. 211–259. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123964991000042

[24] V. A. Chandrasetty and S. M. Aziz, "Chapter 5 - LDPC decoder architectures," in *Resource Efficient LDPC Decoders*, V. A. Chandrasetty and S. M. Aziz, Eds. Academic Press, Jan. 2018, pp. 55–67. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128112557000058

[25] A. Darabiha, A. Chan Carusone, and F. R. Kschischang, "Power Reduction Techniques for LDPC Decoders," *IEEE J. Solid-State Circuits*, vol. 43, no. 8, pp. 1835–1845, Aug. 2008.

[26] K. Le, F. Ghaffari, D. Declercq, B. Vasic, and C. Winstead, "A novel high-throughput, low-complexity bit-flipping decoder for LDPC codes," in *2017 International Conference on Advanced Technologies for Communications (ATC)*, Oct. 2017, pp. 126–131, iSSN: 2162-1039.

[27] C.-L. Wey, M.-D. Shieh, and S.-Y. Lin, "Algorithms of Finding the First Two Minimum Values and Their Hardware Implementation," *IEEE Trans. Circuits Syst. I*, vol. 55, no. 11, pp. 3430–3437, Dec. 2008.

[28] P. Sreemohan and N. Sebastian, "FPGA implementation of min-sum algorithm for LDPC decoder," in *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, May 2017, pp. 821–826, iSSN: null.

[29] F. Gutierrez, G. Corral-Briones, D. Morero, T. Goette, and F. Ramos, "FPGA implementation of the parity check node for min-sum LDPC decoders," in *2012 VIII Southern Conference on Programmable Logic*, Mar. 2012, pp. 1–6, iSSN: null.

[30] N. Kanistras, I. Tsatsaragkos, I. Paraskevakos, A. Mahdi, and V. Paliouras, "Impact of LLR saturation and quantization on LDPC min-sum decoders," in *2010 IEEE Workshop On Signal Processing Systems*, Oct. 2010, pp. 410–415, iSSN: 1520-6130.

[31] E. Sharon, S. Litsyn, and J. Goldberger, "Efficient Serial Message-Passing Schedules for LDPC Decoding," *IEEE Trans. Inf. Theory*, vol. 53, no. 11, pp. 4076–4091, Nov. 2007.

[32] D. Hocevar, "A reduced complexity decoder architecture via layered decoding of LDPC codes," in *IEEE Workshop onSignal Processing Systems, 2004. SIPS 2004.*, Oct. 2004, pp. 107–112, iSSN: null.

[33] K. Zhang, X. Huang, and Z. Wang, "High-throughput layered decoder implementation for quasi-cyclic LDPC codes," *IEEE J. Sel. Areas Commun.*, vol. 27, no. 6, pp. 985–994, Aug. 2009.

[34] Y.-M. Chang, A. I. V. Casado, M.-C. F. Chang, and R. D. Wesel, "Lower-Complexity Layered Belief-Propagation Decoding of LDPC Codes," in *2008 IEEE International Conference on Communications*. Beijing, China: IEEE, 2008, pp. 1155–1160. [Online]. Available: http://ieeexplore.ieee.org/document/4533261/

[35] J.-H. Yoon and J. Park, "An Efficient Memory-Address Remapping Technique for High-Throughput QC-LDPC Decoder," *Circuits, Systems, and Signal Processing*, vol. 33, no. 11, pp. 3457–3473, Nov. 2014. [Online]. Available: https://doi.org/10.1007/s00034-014-9808-3

[36] C. Jones, E. Valles, M. Smith, and J. Villasenor, "Approximate-MIN constraint node updating for LDPC code decoding," in *IEEE Military Communications Conference, 2003. MILCOM 2003.*, vol. 1, Oct. 2003, pp. 157–162 Vol.1.

[37] A. Prabhakar and K. Narayanan, "A memory efficient serial LDPC decoder architecture," in *Proceedings. (ICASSP '05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, vol. 5, Mar. 2005, pp. v/41–v/44 Vol. 5, iSSN: 2379-190X.

[38] X. Chen, J. Kang, S. Lin, and V. Akella, "Memory System Optimization for FPGA-Based Implementation of Quasi-Cyclic LDPC Codes Decoders," *IEEE Trans. Circuits Syst. I*, vol. 58, no. 1, pp. 98–111, Jan. 2011, conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers.

[39] M. Xu, J. Wu, and M. Zhang, "A modified Offset Min-Sum decoding algorithm for LDPC codes," in *2010 3rd International Conference on Computer Science and Information Technology*, vol. 3, Jul. 2010, pp. 19–22.

[40] A. A. Emran and M. Elsabrouty, "Generalized simplified variable-scaled min sum LDPC decoder for irregular LDPC codes," in *2014 IEEE 25th Annual International Symposium on Personal, Indoor, and Mobile Radio Communication (PIMRC)*, Sep. 2014, pp. 892–896, iSSN: 2166-9589.