

2020

# A universal maximum likelihood decoder using noise guessing

---

<https://hdl.handle.net/2144/40724>

*"Downloaded from OpenBU. Boston University's institutional repository."*

BOSTON UNIVERSITY  
COLLEGE OF ENGINEERING

Thesis

**A UNIVERSAL MAXIMUM LIKELIHOOD DECODER  
USING NOISE GUESSING**

by

**VAIBHAV BANSAL**

B.S., Thapar University, 2018

Submitted in partial fulfillment of the  
requirements for the degree of  
Master of Science

2020

© 2020 by  
VAIBHAV BANSAL  
All rights reserved

Approved by

First Reader

---

Rabia Yazicigil Kirby, Ph.D.  
Assistant Professor of Electrical and Computer Engineering

Second Reader

---

Muriel Médard, Sc.D.  
Cecil H. Green Professor of Electrical Engineering  
Massachusetts Institute of Technology

Third Reader

---

Ajay Joshi, Ph.D.  
Associate Professor of Electrical and Computer Engineering

*Never stop fighting until you arrive  
at your destined place - that is, the unique you.  
Have an aim in life, continuously acquire knowledge, work hard,  
and have perseverance to realise the great life.* Dr. A.P.J Abdul Kalam

## Acknowledgments

I would like to thank Professor Rabia Yazicigil, for being a great advisor, a source of constant motivation and being supportive throughout the duration of the research project. Despite the numerous responsibilities, she always found the time to talk, discuss and helped me grow as a researcher. I am forever grateful to her for inspiring me to take up research and giving me the opportunity to be a part of this project. I want to thank her for creating a collaborative and conducive environment in the WISE circuits group, where my colleagues became some of my best friends.

I would like to thank my thesis committee members Professor Muriel Medard and Professor Ajay Joshi, for giving me feedback on my research and for supporting me in my endeavors. I would also like to thank them for facilitating me by providing the necessary resources to make this thesis successful.

I would like to thank Professor Ken Duffy for collaborating on the project and providing constant guidance and feedback over the course of the research project. I would like to thank Professor Anantha Chandrakasan for his guidance, advice and encouragement during the course of the project. I would like to thank Dr. Chiraag Juvekar and Professor Priyanka Raina for providing feedback and advice on the design process.

I would like to thank Amit Solomon, Qijun "Mandy" Liu and Wei An, for collaborating and helping me on the project. I would like to thank Utsav Banerjee and Alex ji from *ananthagroup* to help with the design process and guide me in the various steps of the tapeout.

I would like to thank all the members of WISE circuits group, for making the lab a fun and exciting place to work at. In particular, I would like to thank Timur Zirtiloglu and Arslan Riaz for the help and feedback they gave me on my research.

I would like to thank Tushar Goel and Sohail Budhwani for being wonderful

roommates and all the joyous times we spent together. Finally, I would like to thank my father, Naresh Bansal, my mother Seema Bansal, my sister, Kritika Bansal for their love, encouragement and support.

# A UNIVERSAL MAXIMUM LIKELIHOOD DECODER USING NOISE GUESSING

VAIBHAV BANSAL

## ABSTRACT

Wireless communication technologies lie at the forefront of cutting edge and form the backbone of the Internet and Data first era that we live in. The need for High-speed data communication also exacerbates the need of data reliability. Data is encoded before transmission to ensure that it is faithfully reproduced at the receiver. Decoding an arbitrary code has been described as a NP-complete problem. As a result of this, previous works have developed decoders that are specific to certain codes, as an approximation of Maximum Likelihood Decoding. This co-development of codes and decoding schemes, however, limits the functionality of the decoders, which can only work with a finite number of encoding schemes that were designed for it. It has also been seen that the performance of these decoders degrade as we increase the code-rate.

In our proposed approach we leverage a new algorithm, Guessing Random Additive Noise Decoding (GRAND) algorithm, for realizing Maximum Likelihood (ML) decoding based on noise, contrasting traditional algorithms which decode the information directly. Since GRAND decodes the noise rather than the information, it reduces computational complexity and storage. In contrast to traditional architectures, GRAND decoder can be designed independently of the encoder due to its dependency only on the noise making it a universal maximum-likelihood decoder. Hence this architecture is agnostic to any coding scheme. GRAND algorithm is also proven to be capacity achieving when using random code-books. The decoder works

for high-rate, small block-size code-words, at low latency and low complexity, making it ideal for implementing in the control channel.

Our approach holistically develops and integrates GRAND and embedded security to demonstrate a secure hardware solution that has high-energy efficiency with low latency and low complexity performance metrics addressing next-generation communication system requirements. We present preliminary estimates of throughput around 250 Mbps, at a Bit Error Rate of  $10^{-3}$ , with an energy per bit value of 16.5 pJ/b at a clock frequency of 50 MHz for a supply voltage of 0.9 V.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Thesis Contributions . . . . .	5
1.3	Thesis scope and Organisation . . . . .	6
<b>2</b>	<b>Channel Decoding</b>	<b>8</b>
2.1	Background: Coding Theory . . . . .	8
2.2	GRAND algorithm . . . . .	10
<b>3</b>	<b>Universal Noise-Centric Decoder</b>	<b>13</b>
3.1	Background . . . . .	13
3.2	Universal Decoder Hardware Architecture . . . . .	15
3.3	Error Generator . . . . .	16
3.3.1	Distance Logic . . . . .	17
3.3.2	Pattern Generator . . . . .	18
3.3.3	Error Shifter . . . . .	20
3.4	Matrix-Vector Multiplier . . . . .	21
3.4.1	Conventional hardware implementation . . . . .	22
3.4.2	In-Memory Computation . . . . .	25
3.4.3	Architecture . . . . .	26
3.5	GRAND Chip Implementation . . . . .	30
3.6	Power and Energy Consumption . . . . .	33
3.7	Results . . . . .	34

<b>4 Conclusions</b>	<b>36</b>
<b>References</b>	<b>38</b>
<b>Curriculum Vitae</b>	<b>40</b>

# List of Tables

2.1	Probability distribution of errors of different hamming weights for different BER values . . . . .	12
3.1	Probability distribution of errors of different hamming weights and their cycle requirements . . . . .	31
3.2	Energy per decoded bit throughput for different operating frequencies of the chip . . . . .	34
3.3	Performance Comparison with the State-of-the-Art Decoders for 5G Communications, Polarbear (Giard et al., 2017) and Huawei (Liu et al., 2018) . . . . .	35

# List of Figures

1·1	Data flow in a real communication channel . . . . .	3
2·1	Maximum Likelihood Decoding (Duffy et al., 2018) . . . . .	9
2·2	Maximum Likelihood Decoding through Noise Guessing (Duffy et al., 2019) . . . . .	11
2·3	Complexity of ML decoding with code-rate and code-length for a bit flip probability of $10^{-2}$ (Duffy et al., 2019) . . . . .	12
3·1	Top level block diagram of the chip . . . . .	13
3·2	Decoding algorithm flowchart . . . . .	15
3·3	Code-word is found in the Syndrome calculator for Error Hamming Weight 0 . . . . .	16
3·4	Code-word is found in the Primary block for Error Hamming Weight 1 and 2 . . . . .	17
3·5	Code-word is found in the Secondary block for Error Hamming Weight 3	17
3·6	Error Vectors mapping to distance pairs $(D1, D2)$ . . . . .	19
3·7	The Distance Logic . . . . .	20
3·8	The Error Shifter . . . . .	21
3·9	An example of matrix multiplication in decimal system . . . . .	22
3·10	An example of matrix multiplication in binary system . . . . .	23
3·11	A conventional method to perform matrix multiplication using memory units . . . . .	24

3.12	The active high bits of the second vector acts as the selection switch for the parity matrix . . . . .	26
3.13	Multiplication module in Primary block using Single-Port SRAM. . .	27
3.14	Multiplication module in Secondary block using Single Port SRAM. Since the error vector $E$ contains only up to 3 active high bits at sparse locations within the vector, they are used to select the columns from the parity matrix which are XORed together to generate the final product. . . . .	28
3.15	Multiplication module in Primary block using Dual-Port SRAM . . .	28
3.16	Multiplication module in Secondary block using Dual-Port SRAM . .	29
3.17	Time interleaved architecture for re-randomizing the code-books . . .	32

## List of Abbreviations

ADC	.....	Analog-to-Digital Converter
BER	.....	Bit Error Rate
BSC	.....	Binary Symmetric Channel
CA	.....	CRC Assisted
CRC	.....	Cyclic Redundancy Check
DAC	.....	Digital-to-Analog Converter
DP_SRAM	.....	Dual Port SRAM
GRAND	.....	Guessing Random Additive Noise Decoding
IID	.....	Independent and Identically Distributed
LDPC	.....	Low Density Parity Check
ML	.....	Maximum Likelihood
NA	.....	Not Applicable
SMS	.....	Short Message Service
RM	.....	Reed Muller
SCL	.....	Successive Cancellation List
SP_SRAM	.....	Single Port SRAM
SRAM	.....	Static Random Access Memory

# Chapter 1

## Introduction

### 1.1 Motivation

Communication is one of the most essential tools that facilitate connections and interactions. It continues to form the backbone of all human activity, be it social or economical. The communication industry was revolutionized when Alexander Graham Bell invented the telephone 1876. Communication which used to take days or weeks could now be done in a matter of minutes. We saw the advent of the first mobile telephony systems in the 1980s which revolved around phone calls using bulky handheld devices. When the next generation of mobile communication technology landed in the 1990s, we saw widespread adoption of text messages through SMS. Another decade later, the 3G technology enabled us to access emails on our phones, and brought many businesses in grasp of their hands. It wasn't until the fourth generation of wireless technology when video streaming became mainstream. Each generation brought with it additional bandwidth to play with, higher throughput, and lower latency operations, which made it possible to see such unprecedented advancements in media consumption, and gave rise to new businesses and business models. These advancements amalgamated into the electronic telecommunication technologies and transformed the mobile phones from a niche luxury to a basic necessity.

Walking into the next decade, we can see the advent of the fifth generation of wireless communication technology, more commonly referred to as 5G, which is a suite of telecom technologies that are standardized by industry-led bodies. Following

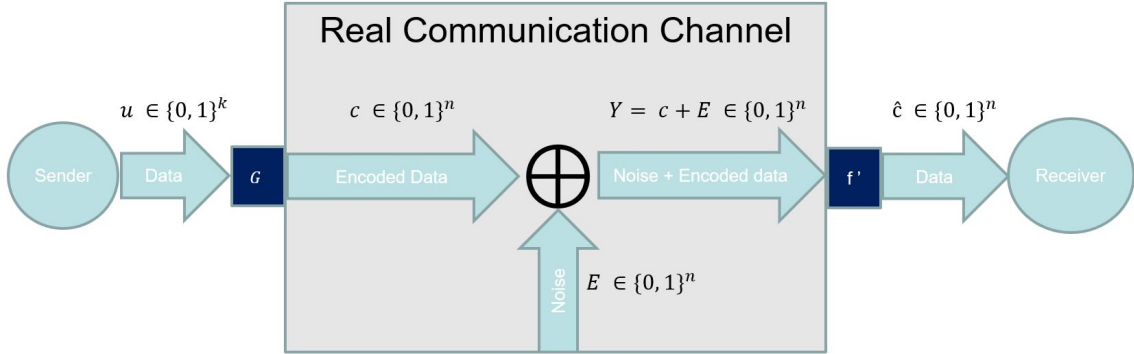
similar pattern of the past, 5G brings higher data rates, more reliable and ubiquitous communications, ultra-low latency services, which find themselves being utilized for operations such as haptic communications, remote surgery, centralized gaming services, and 5G-enabled services for specific industries such as automotive networks and satellite services. An important thing to note is that the integration of 5G with other technologies does not follow the traditional assignment of roles between mobile telephone service providers and equipment manufacturers, and brings new entrants into the 5G market. (Médard, 2020)

Millions of mobile and wireless communication devices are being used everyday, transmitting and receiving unprecedented amounts of data every second. These communications are susceptible to various different types of disturbances, be it noise in transmission channels, poor signal strength, interference from other devices communicating simultaneously, or jamming etc. Owing to these phenomena, the bits of data received could be different from the ones that were transmitted. To ensure that the data is faithfully received at the other end, we use different techniques such as channel and source coding which preserve the reproducibility of the data to some finite extent.

Channel coding is defined as adding redundancy in the data that is to be transmitted, to ensure that the receiver identifies the originally sent bits. Reed-Solomon codes (Reed and Solomon, 1960), Hadamard codes (Bell, 1966), and Hamming codes (Hamming, 1950) are some of the most commonly used examples of channel coding methodologies.

Linear codes are one of the most commonly used type of codes. Let  $u$  be any string of information comprising of symbols from alphabet  $\mathcal{X}$  that has to be transmitted. The linear code is defined as a vector  $c$ , such that

$$c = uG$$



**Figure 1.1:** Data flow in a real communication channel

where  $G$  is known as the generator matrix. The set of all  $n$  bits wide vectors  $c$  is known as the code-book. Linear codes can be of two types: systematic codes and non-systematic codes. All codes that can be represented in the form

$$c = [u, r]$$

are called systematic codes, where  $u$  is a  $k$  bits wide string of information that is to be transmitted, and  $r$  is a string  $n - k$  bits wide, which is the redundancy added during the encoding process. All other codes which cannot be defined using the expression above are called non-systematic codes.

Figure 1.1 describes the the system in a real communication channel. In a typical communication system, the data to be transmitted is compressed into smaller packets through source coding. Let  $u$  be the output of source coding. Channel coding is performed on these data packets to add redundancy, which allows for error detection and correction at the receiver. The code-word  $c$  is now modulated over a carrier signal and transmitted through a wireless channel. Since the transmission channel is not perfect, some random channel noise  $E$  gets added to the code-word.  $E$  can be envisioned as another  $n$  bits wide vector that gets added to the code-word, representing the corruption observed in data during transmission. When the carrier signal is demodulated at

the receiver, we get the channel output as  $Y = c + E$ , another  $n$  bit wide vector. To obtain the data that was transmitted, a channel decoding algorithm, say  $f$ , tries to decode the data received at the channel output and produce the expected code-word  $\hat{c}$ , another  $n$  bit wide vector. The various redundancies that were added to the data during transmission make it possible to retrieve transmitted data from the corrupted channel output.

Most codes are co-designed along with a number of decoding schemes, which makes it necessary to change the decoder if the codes are to be changed. Reed-Mueller (RM) (Reed, 1953; Muller, 1954) and Majority Logic, Cyclic-Redundancy-Check Assisted (CA) Successive Cancellation List (SCL) decoding for CA-Polar codes (Arikan, 2008) and Low Density Parity Check (LDPCs) (Gallager, 1962) and Belief Propagation (BP). Since the codes have to be amenable to the decoding algorithms, they are limited in their construction.

Only a limited code rates are possible for different code-lengths. The code rate  $R$  is defined as the ratio of the number of information bits  $k$  to the code-length  $n$ .

$$R = \frac{k}{n}$$

Although work on short length codes (Spectre, 2020) indicates support for all code-lengths, the bounds of feasibility do not produce codes and associated decoders. At short code-lengths, the choice for code-rates goes below capacity. To achieve higher rates, conventional codes require longer code-lengths. To put things into perspective, lets take an example of the 3GPP 5G NR data channel (ETSI, 2018). In typical cellular communications, the Bit Error Rate (BER), which is defined as the number of bit errors per unit time, generally ranges from  $10^{-2}$  to  $10^{-4}$ . Assuming a Binary Symmetric Channel model, which is defined as a channel capable of sending and receiving only one of the two symbols (0 or 1) with a crossover probability  $p$ , the

channel capacity is of the order of 0.92 to 0.999. The capacity of the channel is given by

$$C = 1 - H_b(p)$$

where  $H_b$  is the binary entropy function. The 3GPP 5G NR data channel uses 3840-bit LDPCs with rates  $R$  as low as 0.2, and 8448-bit LDPCs with  $R$  between 0.33 to 0.91. In addition to that, to make the channels appear IID, interleaving takes place over thousands of bits. (ETSI, 2018) These schemes introduce latency and not even come close to achieving channel capacity. In Maximum Likelihood (ML) decoding, the received code-word  $x$  is compared to all the code-words in the code-book. The code-word that is the closest to the input code-word is chosen as the output code-word. A usual implementation of the Channel decoder usually focuses on finding a match for the code-word received within the code-book. This inherently necessitates the knowledge of the encoding scheme for the decoder, hence their inter-dependency. On the contrary, our decoder focuses on finding the error that the code-word might have incurred during the transmission. This makes the decoder agnostic to the encoding scheme since we can decode any type of linear codes using this algorithm. We also demonstrate that the system is capacity achieving with Random Linear codes for small code-lengths.

## 1.2 Thesis Contributions

This thesis presents a universal noise-centric channel decoder hardware using Guessing Random Additive Noise Decoding algorithm (Duffy et al., 2019). The universal decoder architecture achieves  $5\times$  reduction in energy consumption per decoded bit for a bit error rate (BER) of  $10^{-3}$ ,  $1.33\times$  increase in throughput, while using  $8\times$  smaller code-length at a high code rate ( $R$ ) of 0.8 compared to the state-of-the-art decoder implementations for polar codes used for 5G and next-generation wireless

communications (Giard et al., 2017), (Liu et al., 2018). Further, we propose new avenues for security via hardware-architected solutions.

Thesis contributions include:

1. Developing a novel hardware architecture to generate possible noise sequences in decreasing order of their likelihood to reduce energy consumption.
2. Achieving low latency in matrix-vector multiplication operations by exploiting sparsity of error vectors.
3. Designing a time-interleaved decoder with built-in security which switches between multiple parity check matrices ( $H$ ) on the fly enabling re-randomization of the code-book for every code-word with zero dead zone in decoding.

### 1.3 Thesis scope and Organisation

This thesis presents a hardware architecture that is realizing a Universal Maximum Likelihood Decoder using the Guessing Random Additive Noise Decoding (GRAND) algorithm, which is capacity achieving when used with random codes (Duffy et al., 2019). We demonstrate that the decoder achieves low latency, with an order of magnitude lower energy consumption than the state-of-art decoders, while ensuring security of the operation, making it ideal for use in the next-generation of wireless technologies. For our purposes, we are going to assume a BSC model with the crossover probability of  $p = 10^{-3}$ .

We discuss channel decoding in Chapter 2 and study its impact on the field of wireless communication. We then discuss the details of the GRAND algorithm and how it performs Maximum Likelihood (ML) decoding through noise-guessing.

In Chapter 3 of the thesis, we discuss the system architecture and hardware implementation of the GRAND algorithm. We go over the design methodology behind the

various blocks in the chip and evaluate the various decisions taken throughout the implementation process. We discuss a novel method to generate possible error sequences in an error generator, which is essential for the operation of the algorithm. We also utilize the channel noise statistics to optimize the architecture for energy consumption and power by splitting the error generator hardware into separate units called Primary and Secondary blocks to generate error vectors of specific hamming weights. The chapter also contributes by outlining a method to implement low-latency binary matrix-vector multiplication leveraging the sparsity of the noise sequences.

Chapter 4 summarizes the thesis contributions, and compares the performance against the other state-of-the-art decoders published in literature. It also discusses some of the open research questions and opportunities to build on the current architecture to improve latency, throughput, and energy consumption, while maintaining the universality of the decoding approach.

## Chapter 2

# Channel Decoding

### 2.1 Background: Coding Theory

In an ideal communication channel, data transmitted across a channel is received by the user without any interference or noise. In a real communication channel, however, errors such as data corruption in the form of bit flips and data erasures are commonplace. This raises the need to have sophisticated mechanisms to encode the data during transmission and to decode the data received from the channel output, which might have errors included in it. One of the ways to do this is channel coding, where data is padded with extra bits of information which help in the detection and correction of the received data at the receiver's end. Some of the commonly used codes are Cyclic Redundancy Check (CRC) codes (Peterson and Brown, 1961), Reed-Muller codes (Reed, 1953; Muller, 1954), Reed-Solomon codes (Reed and Solomon, 1960), Hadamard codes (Bell, 1966), Hamming codes (Hamming, 1950), etc.

The idea of channel coding was perceived by Claude Shannon, in his landmark paper (Shannon, 1948) in 1948. His idea of a code-book was as follows. Consider a block of  $n$  symbols from an alphabet  $\mathcal{X}$  of size  $|\mathcal{X}|$ , such that

$$\mathcal{X}^n = \{x^{n,0}, x^{n,1}, \dots, x^{n,|\mathcal{X}|^n-1}\}$$

Out of the  $|\mathcal{X}|^n$  possible binary strings of length  $n$ , he suggested to choose  $|\mathcal{X}|^{nR}$

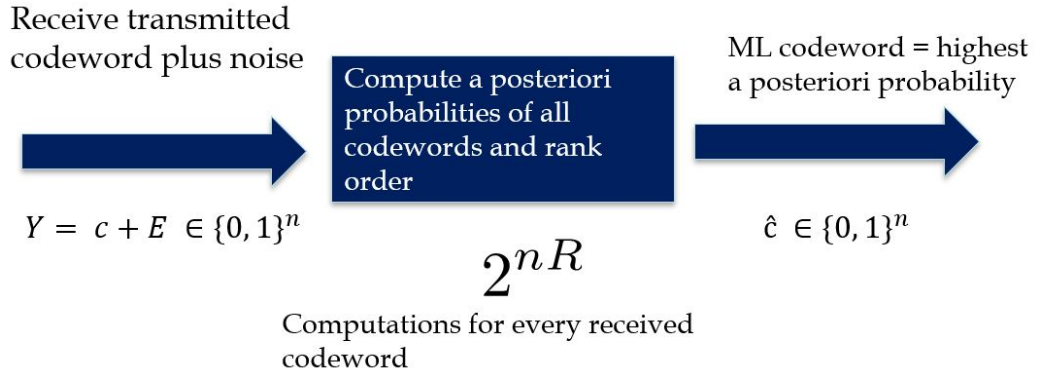
strings, drawn uniformly at random, where  $R$  is the code rate, defined as

$$R = \frac{k}{n}$$

The chosen  $|\mathcal{X}|^{nR}$  strings constitute the code-book  $\mathcal{C}$ , such that

$$\mathcal{C} = \{c^{n,0}, c^{n,1}, \dots, c^{n,|\mathcal{X}|^{nR}-1}\} \subset \mathcal{X}^n$$

In the example of a Binary Symmetric Channel,  $\mathcal{X} = \{0, 1\}$  and  $|\mathcal{X}| = 2$ .



**Figure 2.1:** Maximum Likelihood Decoding (Duffy et al., 2018)

One of the many decoding methodologies is Maximum Likelihood (ML) decoding where we compute a posteriori probabilities of all the code-words and rank order them. Consider a discrete channel with channel inputs  $\mathcal{C}$  and outputs  $Y^n$ , which take value in  $\mathcal{X}^n$ . The channel inputs are corrupted with a random noise  $E^n$ , that is independent of the inputs and also present in  $\mathcal{X}$ . The function of the channel is represented by  $\oplus$ , and can be described as

$$Y^n = c^n \oplus E^n$$

The function is assumed to be invertible, such that given  $\mathcal{C}$  and  $Y$ , we can determine the noise as

$$c^n = Y^n \ominus E^n$$

For the implementation of ML decoding, the  $\mathcal{C}$  is first shared between the sender and the receiver. For each channel output  $y^n$  that we receive, we compute the probability of it being in the code-book by

$$p(y^n|c^{n,i}) = P(y^n = c^{n,i} \oplus E^n) \quad \forall i = \{0, 1, \dots, 2^{nR} - 1\} \quad (2.1)$$

This algorithm, however, has high complexity and is extremely computationally intensive. Let  $n = 1024$  and  $R = 0.9$ . Therefore we now have

$$2^{1024 \times 0.9} \approx 10^{277}$$

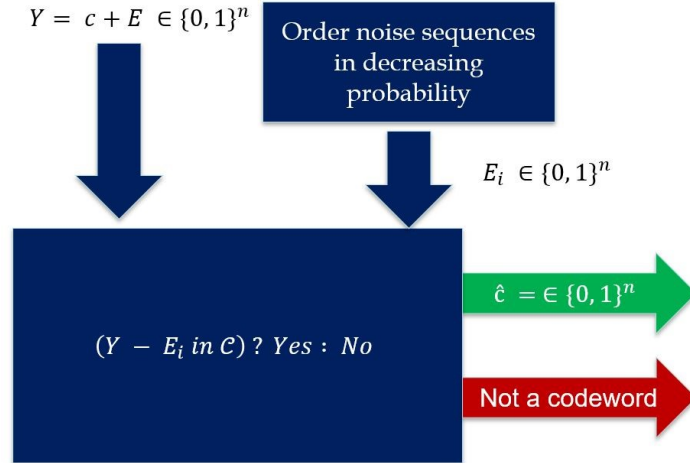
code-words in the code-book  $\mathcal{C}$ . In order to do ML decoding, one has to perform  $10^{277}$  computations for each received code-word, and the hardware has to store or needs low-latency access to as many code-words. This limitation makes ML decoding impractical to achieve in real life.

## 2.2 GRAND algorithm

The decoding through the Guessing Random Additive Noise Decoding algorithm (Duffy et al., 2019) focuses on the errors that might get incurred instead of finding the code-words themselves and can be given by

$$c^{n,*} \in \operatorname{argmax}(p(y^n|c^{n,i}) : c^{n,i} \in \mathcal{C}) = \operatorname{argmax}\{P(E^n = y^n \ominus c^{n,i}) : c^{n,i} \in \mathcal{C}\} \quad (2.2)$$

The working of the algorithm can be understood from Figure 2.2. Instead of the most likely code-word, we generate a series of possible noise sequences, also referred to as error vectors, in the decreasing order of the probability of their occurrence. These error vectors are representations of the noise that might have gotten added into the code-word during transmission. We subtract these error vectors from the received code-word and check against the code-book for membership. The first error



**Figure 2.2:** Maximum Likelihood Decoding through Noise Guessing (Duffy et al., 2019)

vector that produces a member of the code-book is the resultant code-word, and the process is called ML decoding through noise guessing. The process requires us to know the characteristics of the channel of communication, such as the expected uncoded Bit Error Rate (BER), and a model of the noise, such as the Binary Symmetric Channel (BSC), that we might observe. This is particularly important to determine the error vector sequences in their decreasing order of likelihood. Since the errors in the channel are not dependent on the codes being used, any kind of linear codes can be used with the algorithm, making the algorithm code-book agnostic and compatible with different schemes. Hence making it one of the most flexible algorithms to decode.

We can find out the probability of having an error of hamming weight  $i$  using the following expression

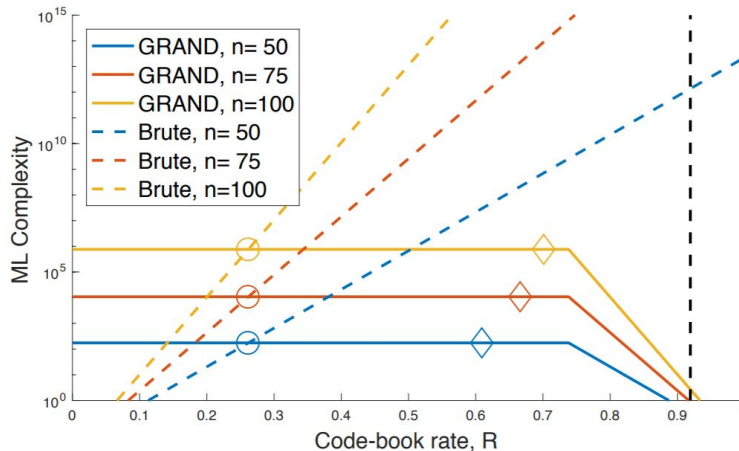
$$P(i) = \binom{128}{i} (BER)^i (1 - BER)^{128-i}$$

where  $BER$  is the uncoded Bit Error Rate of the channel. Table 3.1 presents the

different hamming weights of the errors along with the probability of occurrence for different uncoded  $BER$  values.

Hamming weight	$BER = 10^{-2}$	$BER = 10^{-3}$	$BER = 10^{-4}$
0	0.2762	0.8798	0.9872
1	0.3572	0.1127	0.0126
2	0.2291	0.0072	$8.02 \times 10^{-5}$
3	0.0972	0.0003	$3.37 \times 10^{-7}$
4	0.0306	$9.42 \times 10^{-6}$	$1.11 \times 10^{-9}$

**Table 2.1:** Probability distribution of errors of different hamming weights for different BER values



**Figure 2.3:** Complexity of ML decoding with code-rate and code-length for a bit flip probability of  $10^{-2}$  (Duffy et al., 2019)

Figure 2.3 shows the complexity (average number of guesses made per received bit) for GRAND algorithm and brute-force over a range of block lengths  $n$ . The vertical dashed line shows the channel capacity for BSC. The computational complexity for brute-force, which computes the conditional probability of each code-word in the code-book, increases rapidly with the rate. For GRAND however, the complexity of guessing the noise decreases as the rate is increased. The circles in the plot indicate the code-rate at which the complexity of guessing the noise is lesser than the brute-force approach.

## Chapter 3

# Universal Noise-Centric Decoder

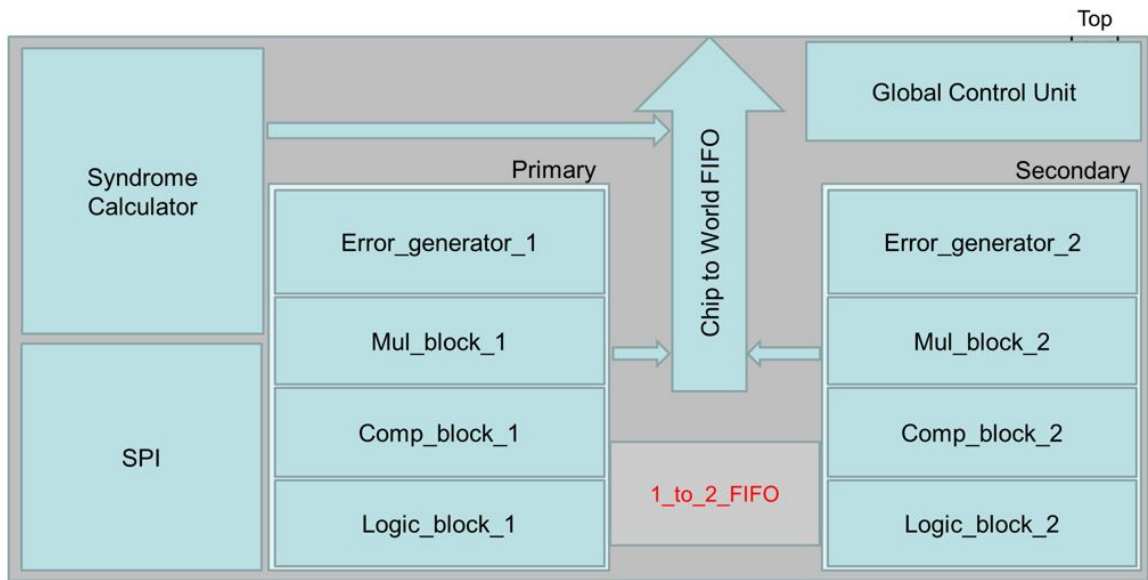


Figure 3.1: Top level block diagram of the chip

### 3.1 Background

Most decoding schemes revolve around finding out the code-words from a list of known code-words, which require the decoder to be co-developed with the encoding scheme and a decoding algorithm. This helps them achieve higher throughput, lower energy consumption, and lower latency operations. In doing so, the hardware is naturally locked onto a particular code-book and therefore rendered useless if the code-book gets changed. Instead of finding out the code-word itself, we focus on finding the error that might have been introduced during the transmission of the signal. The

error can be guessed with high probability given the statistical characteristics of the channel. The GRAND algorithm generates error vectors in the decreasing order of their probability. We perform parity checks to match against the value received. This gives us the error that was introduced during transmission, and allows us to generate the expected code-word from the channel output. This process is better known as Syndrome decoding (Masnick and Wolf, 1967).

Let  $c$  be the code-word that was sent over the channel and  $E$  be the noise that got added into it during transmission giving us

$$Y = c \oplus E$$

as the channel output. Let  $G$  be the generator matrix used to generate code-word  $c$  and  $H$  be its corresponding parity matrix. Multiplying both sides with parity matrix  $H$  gives us

$$HY = Hc \oplus HE$$

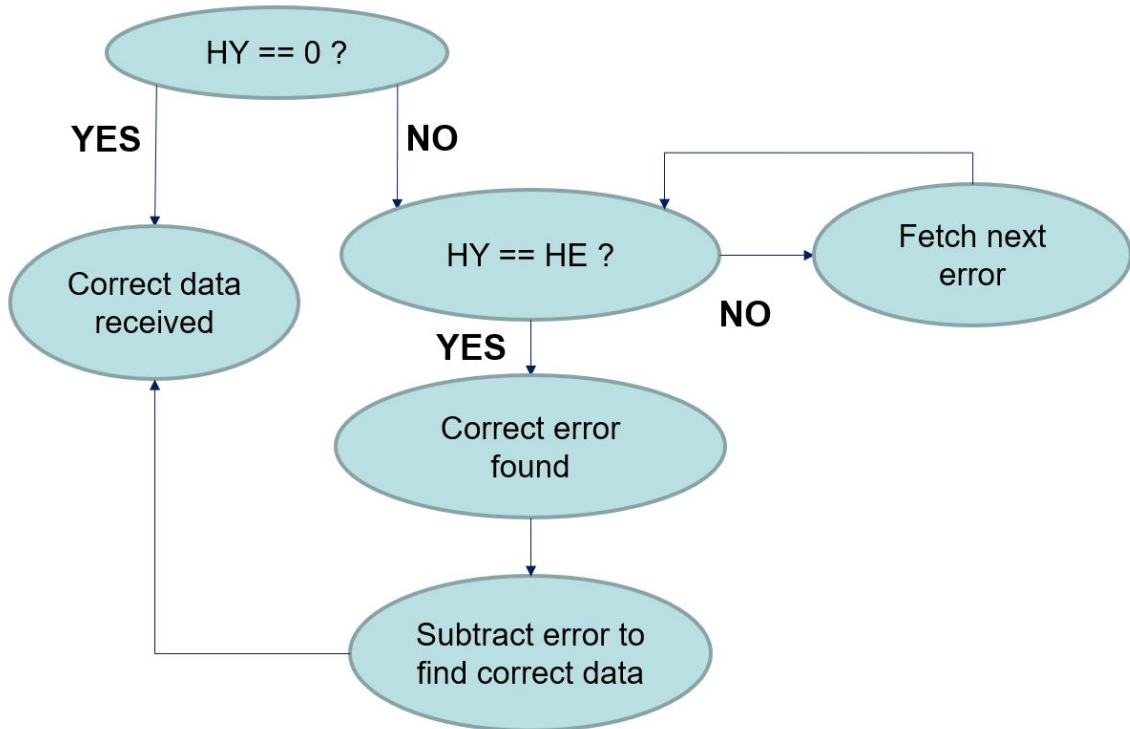
Syndrome decoding takes advantage of the property of  $H$  matrix such that

$$Hc = 0$$

which gives us

$$HY = HE$$

To perform ML decoding, we look at a pre-computed table of  $HE$  mapping to  $E$ . GRAND however, does not depend on a look up table. Instead, we use the channel noise characteristics to generate the noise sequence in the decreasing order of the probability and use the code-book membership as a hash to verify the error.

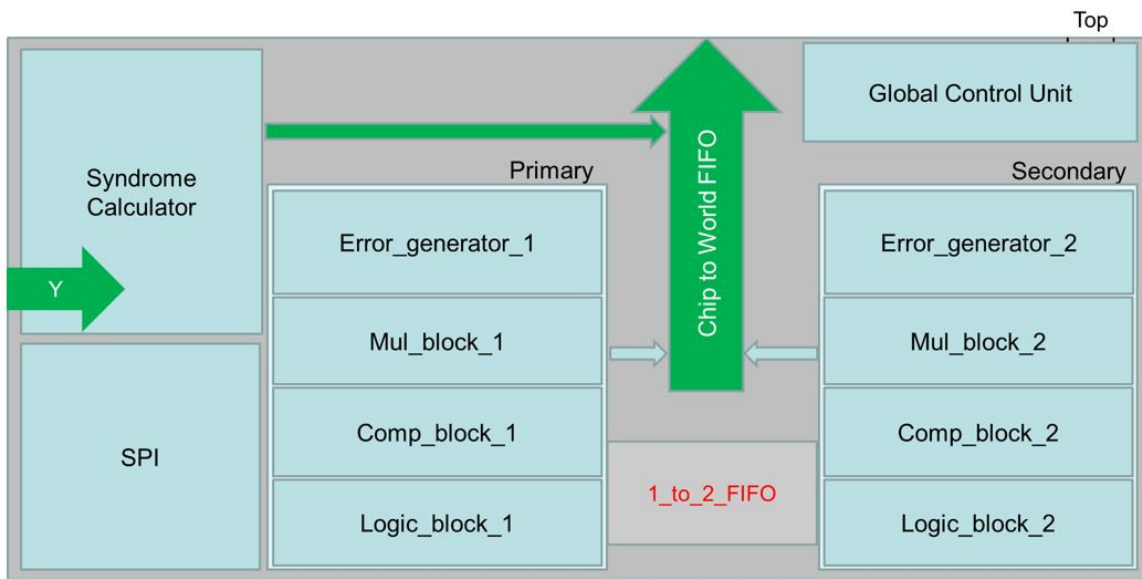


**Figure 3-2:** Decoding algorithm flowchart

### 3.2 Universal Decoder Hardware Architecture

Figure 3-1 shows the hardware architecture of the universal decoding approach. Figure 3-2 depicts the algorithm flowchart. The channel output  $Y$  is a 128-bit wide vector ( $n = 128$ ), which is passed to the Syndrome calculator block, where  $Y$  is multiplied with the parity-check matrix  $H$  to calculate the syndrome of the channel output. The dimensions of the parity matrix  $H$  is governed by the code-length  $n$  and the minimum supported code-rate  $R$ , and the dimensions can be given as  $n - k \times n$ . For a minimum supported rate of 0.656, we get  $k = 84$ , therefore the dimensions of the parity matrix  $H$  becomes  $44 \times 128$ . The syndrome is a zero-vector *iff* the channel output is a member of the code-book. This implies that no error that was introduced in the transmission and we got the correct code-word. If the syndrome is not a zero-vector, then the product is passed on to the Primary Block, where we

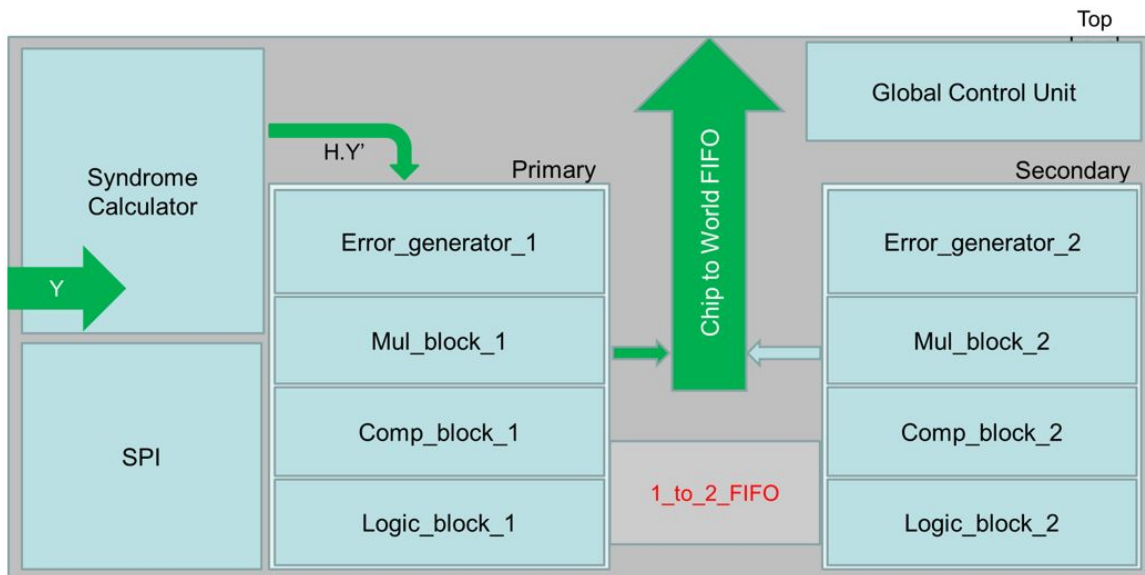
generate the error vectors with hamming weight of one and two. It is highly likely that the error incurred is found in this block. If we find the error, we generate the code-word and pass it on to the output. If the error is not found, then we pass on the product to the Secondary block, which generates the error vectors with hamming weight of three. The same process is repeated again. If error is still not found, we abandon the search for the same and expect the system to re-transmit.



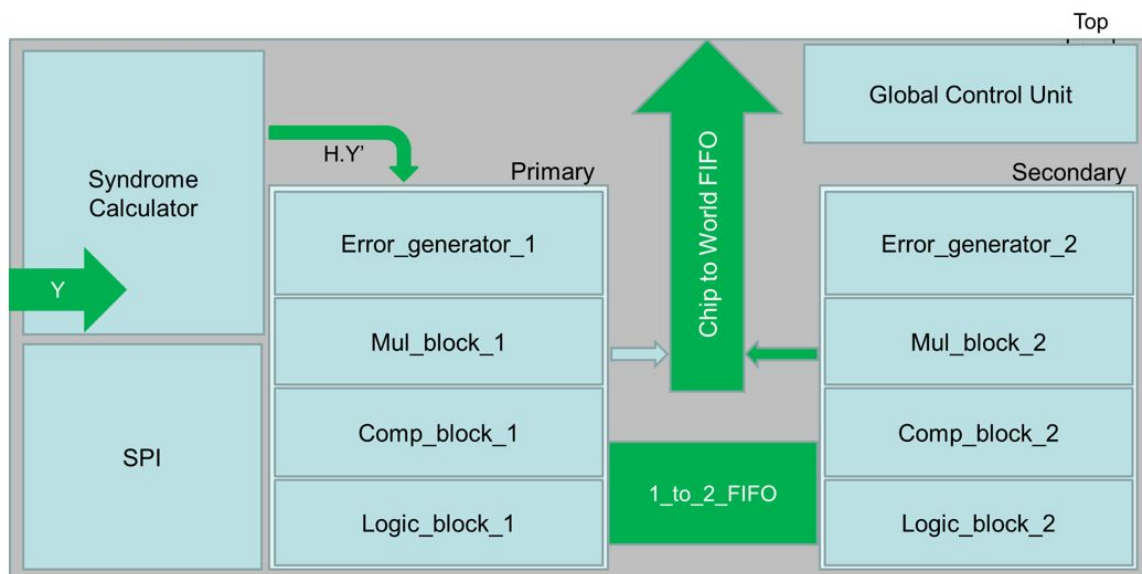
**Figure 3-3:** Code-word is found in the Syndrome calculator for Error Hamming Weight 0

### 3.3 Error Generator

The error generator creates ordered error sequences in parallel based on the decreasing order of their likelihood of occurrence. The error generator hardware consists of three modules; Distance Logic, Pattern Generator, and Error Shifter. Each of the blocks will be briefly discussed in the following subsections.



**Figure 3-4:** Code-word is found in the Primary block for Error Hamming Weight 1 and 2



**Figure 3-5:** Code-word is found in the Secondary block for Error Hamming Weight 3

### 3.3.1 Distance Logic

We consider an example case study of universal decoding in a BSC with a maximum of three error bits in the received code-word. We can expect the bit flips to emerge

anywhere throughout the vector. When we arrange the expected error vectors in the decreasing order of their probability, we see a pattern emerging. These patterns are visualized in Figure 3-6. We use **D1** to define a pattern with two bit flips, where D1 represents the distance between the two active high bits (higher significant bit included). With each subsequent high bit, we would need an extra variable to define the pattern. Therefore, to define a pattern with three bit flips, we add a second variable **D2**, which represents the distance between the most significant high bit and the next significant high bit (higher significant bit included). As we move down in the probability order, we see the value of D1 and D2 changing in a particular order. The first pattern contains one active bit, and is treated as a special case, using  $(D1, D2) = (0, 0)$ . The values of the  $(D1, D2)$  pair can be given by

$$\{(D1, D2) \mid D1 \in [0, 127] \wedge D2 \in [0, 126] \wedge D1 = 127 - D2\}$$

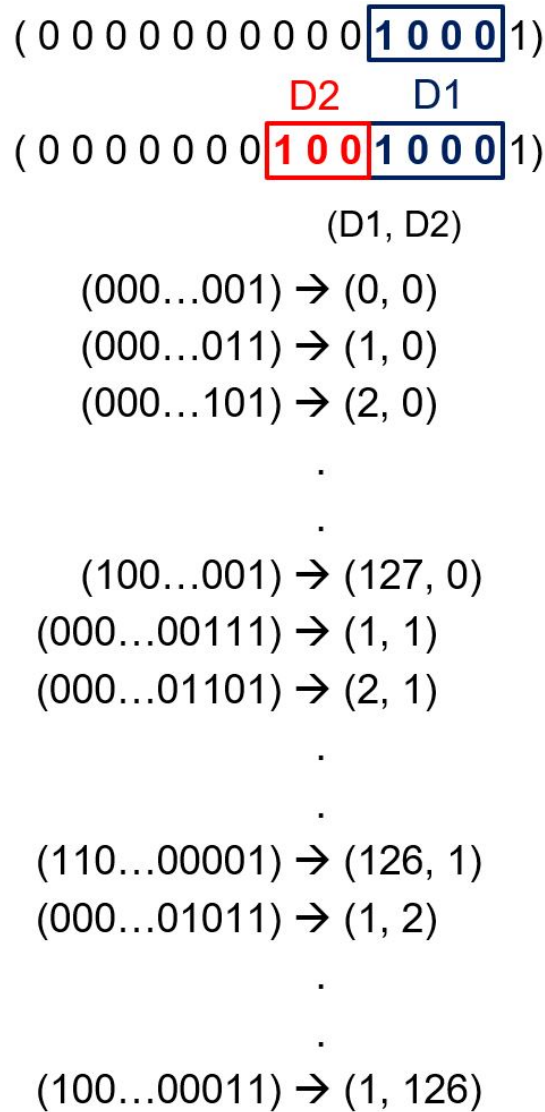
The Distance Logic takes into consideration this pattern of  $(D1, D2)$  pairs and sequentially generates these values upon getting the appropriate overflow signal.

The Distance logic is realized as counters. On the basis of the various conditional statements becoming true, the values of the D1 and D2 are incremented. These values are used as input by the next module called the Pattern Generator.

### 3.3.2 Pattern Generator

The pattern generator takes the  $(D1, D2)$  pair from the Distance logic and constructs the input seed pattern for the Error shifter module. It uses the following expression to generate the pattern

$$X = 1 + 2^{D1} - (D1 == 0) + 2^{D1+D2} - (D2 == 0)2^{D1}$$



**Figure 3-6:** Error Vectors mapping to distance pairs  $(D1, D2)$

The output of the pattern generator is the indices of the active high bits of the error vector. For example, for the error vector  $\vec{e} = (0000\dots010001)$ , the pattern generator gives 0 and 4 as its output. Similarly, for an error vector  $\vec{e} = (0000\dots10101)$ , it gives 0, 2 and 4 as the output.

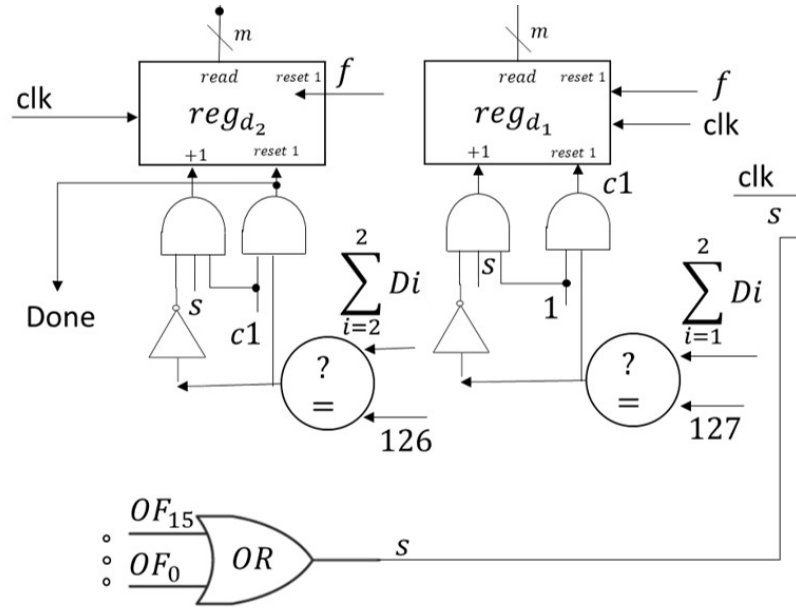
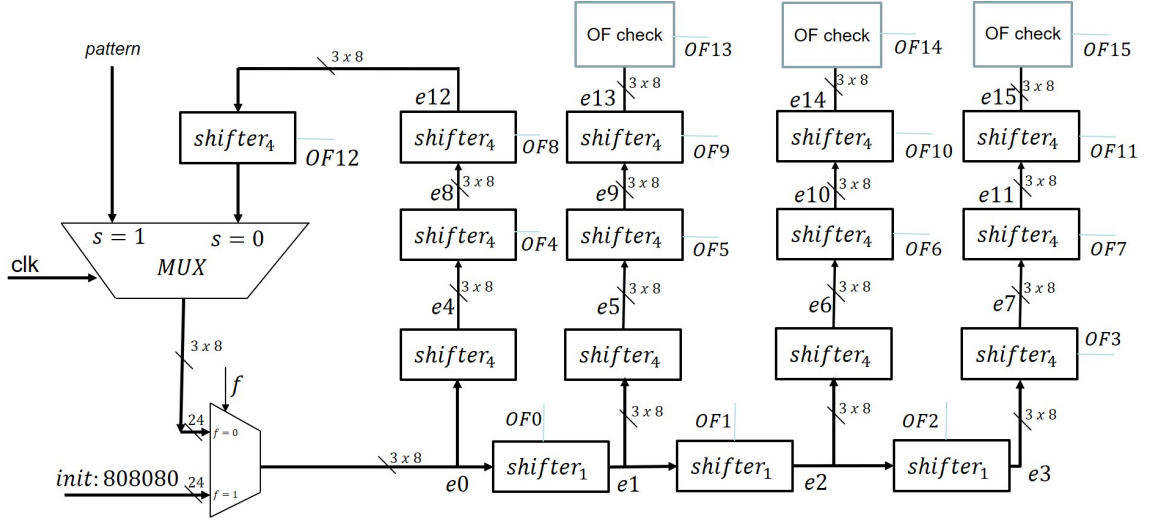


Figure 3.7: The Distance Logic

### 3.3.3 Error Shifter

The output from the Pattern generator always starts from the Least significant side of the error vector. In previous discussion, we established that the errors could materialize anywhere within the code-length. Therefore we need a mechanism that can use the seed patterns generated by the pattern generator and generate the shifted patterns to give the various possible error vectors.

The error shifter module takes the seed pattern from the pattern generator and cyclically shifts the pattern towards the Most significant bit, essentially performing the Logical Shift Left (LSL) operation on the vector. In practice, it receives the indices of the active high bits, so to perform a LSL operation, it adds 1 to the existing values of the indices. The module is capable of generating as many as 16 error vectors at the same time, in each clock cycle, which helps to greatly parallelize the operation. The error vectors are generated in 4 branches in parallel as shown in Figure 3.8. This helps in reducing the critical path significantly, thereby helping with the timing of



**Figure 3-8:** The Error Shifter

the module.

In addition to generating the error vectors, the error shifter is also responsible for generating the overflow signals. An overflow is defined as the condition where the MSB of the error vector goes high. This condition is chosen because a further LSL operation on the vector will reduce the number of high bits in the vector, thereby changing its hamming weight. The overflow signals are sent collectively to the distance logic, where a bitwise OR operation is performed on all the overflow values. The result is used to increment the counters for  $(D1, D2)$  pair, to generate the next seed pattern. This process is repeated till it reaches  $(1, 126)$ . The total number of cycles required for different hamming weight of the error vectors is shown in Table 3.1.

### 3.4 Matrix-Vector Multiplier

Since the architecture has heavy matrix-vector multiplication dependencies for the several noise sequences (error vectors), you explored architecture and circuit-level design techniques to achieve low latency and complexity. We have two matrices

$[A]_{m \times n}$  and  $[B]_{n \times o}$ , then their product can be defined as

$$[C]_{m \times o} = [A]_{m \times n} [B]_{n \times o}$$

We need to note here that the number of columns in the first matrix should match with the number of rows in the second matrix. For a matrix-vector multiplication, either  $o = 1$ , or  $m = 1$ , to represent a vector with one dimension. This defines the size of the  $[C]$ . The individual elements of  $[C]$  can be given as

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj} \quad (3.1)$$

$$H_{2 \times 4} \cdot E_{4 \times 1} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$H \cdot E = Z_{2 \times 1} = \begin{bmatrix} 10 \\ 20 \end{bmatrix}$$

**Figure 3-9:** An example of matrix multiplication in decimal system

This same can be extended to the binary radix. Here, a row of the first matrix is multiplied by the column of the second matrix (mod 2) and then their products are added together (mod 2). We can also say that the multiplication of each element is analogous to bit-wise AND operation in the binary space, and the addition is analogous to bitwise XOR operation. (Direct, 2020) (DeTurck, 2019)

### 3.4.1 Conventional hardware implementation

Equation 3.1 is used to calculate the individual elements of the product matrix  $[C]$ . The matrix multiplication operation can be implemented in hardware by incorpo-

$$H_{2 \times 4} \cdot E_{4 \times 1} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

$$H \cdot E = Z_{2 \times 1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

**Figure 3.10:** An example of matrix multiplication in binary system

rating memory units such as Read Only Memory (ROM) or Static Random Access Memory (SRAM) for storing matrix elements. The basic operation of the multiplication in hardware can be described as follows:

**Step 1** Read and store a column of the second matrix in registers.

**Step 2** Read one row of the first matrix and store in registers.

**Step 3** Multiply individual elements and store the respective products in registers.

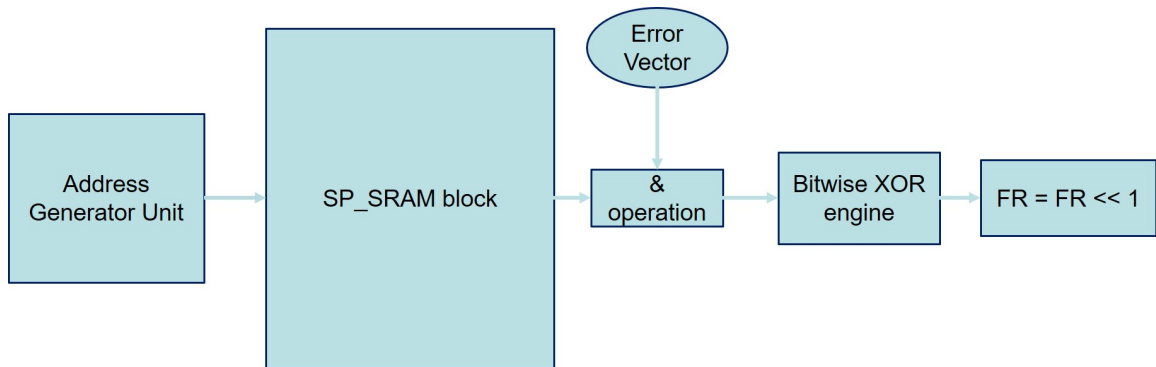
**Step 4** Add the results to get the final value of the element.

**Step 5** Repeat the same till all the rows are exhausted and one column of the resultant matrix is obtained.

**Step 6** Repeat the above steps for each column of the second matrix.

The successful completion of the steps above give us the final product of the matrices. Since the hardware can exploit parallelism, its possible to make this seemingly laborious task faster. For example, we can perform simultaneous multiplication of the elements and store the results in parallel. We can also perform multi-input additions to perform it faster. Do note, that we need to consider the area and energy consumption aspects of the implementation. As it is evident, it is a trade-off between performance metrics such as latency, energy consumption, and area. Since we are

dealing with binary operands, we can further optimize this process by using simpler units such as bitwise AND and XOR logic gates, instead of typical MAC (Multiply and accumulate) units, to perform the same functionality.



**Figure 3-11:** A conventional method to perform matrix multiplication using memory units

Since the design relies heavily on the matrix-vector multiplication aspect for the functionality, it is imperative that the implementation is low latency. Figure 3-11 shows the block diagram of a conventional Von-Neumann architecture based implementation.

Since the channel output  $Y$  is a  $(128 \times 1)$  vector, it can be directly stored in a register. We read the individual rows of the SRAM which contains the first matrix (called  $H$  from here on). This data is then bitwise-ANDed with  $Y$ . The result is then fed into an XOR engine, which takes the 128 bits of the product and provides a single bit of output. This is one of the elements of the final result. This value is then fed into a Shift register to generate the final resultant vector. Since there are 44 rows in our  $H$  matrix, this operation will take at least as many cycles to generate the final product. Due to the low latency requirement of the operation, the cost of having 44-45 cycles for the multiplication itself doesn't make a good design choice.

### 3.4.2 In-Memory Computation

One of the possible alternatives can be In-memory computation. The implementation described above is based off of the Von-Neumann architecture (von Neumann, 1993). According to the Von-Neumann architecture, compute and storage units are separated. The data that is supposed to be processed is first fetched from the storage location, brought to the computation unit through various busses, then the computation is performed, and the result is stored back at the memory location. This approach is useful when the type of calculation is not repetitive and diverse. For our case, the multiplication was always of the same type and only one of the operand was constantly changing. Instead of fetching the operands from the memory, in-memory computation emphasizes on integrating the memory and the compute unit. This saves the latency introduced by repetitively accessing the same data and transporting it to and from the compute unit. Two of the suitable architectures were X-SRAM (Agrawal et al., 2018) and Conv-SRAM (Biswas and Chandrakasan, 2018).

X-SRAM uses the inherent structures of the SRAM modules to generate computation, by the addition of two transistors to form a read path and 4 skewed inverters to produce NAND, NOR and XOR outputs. Although it is not required to always fetch the data from the SRAM, and bringing it to the compute unit, it does have additional challenges associated with it. One of them being the high area cost for implementation. For each computed bit, it requires two read transistors and 4 skewed inverters, which quickly adds to the overall cost as the size of the data is increased. Another problem with this type of implementation would be that the data that has to be multiplied will have to be stored at multiple locations, hence adding on to the area cost significantly. This also presents problems in writing to the memory, since writing the vector at multiple locations will require additional cycles.

Conv-SRAM uses the translation of the data into an analog voltage equivalent and

then 'multiplying and averaging' the same with the weights stored in the specifically designed Conv-SRAM modules. This implementation was also not suitable for our use case because of the requirements of additional Digital-to-Analog Converter (DAC) and Analog-to-Digital Converter (ADC) units required to make this work adds area overhead. In addition to that, the advantage of the approach is realized with larger decimal based data rather than single bit binary data.

### 3.4.3 Architecture

$$\begin{aligned}
 H_{44 \times 128} \cdot E_{128 \times 1} &= \begin{bmatrix} 1 & 0 & 0 & \dots & \dots & 0 \\ 0 & 1 & 0 & \dots & \dots & 1 \\ 0 & 1 & 1 & \dots & \dots & 0 \\ 1 & 1 & 0 & \dots & \dots & 0 \\ 0 & 1 & 1 & \dots & \dots & 1 \\ \vdots & \vdots & \vdots & \dots & \dots & \vdots \\ \vdots & \vdots & \vdots & \dots & \dots & \vdots \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \\
 H \cdot E = Z_{44 \times 1} &= \begin{bmatrix} 1 \oplus 0 \oplus 0 \oplus \dots \oplus 0 \\ 0 \oplus 1 \oplus 0 \oplus \dots \oplus 0 \\ 0 \oplus 1 \oplus 0 \oplus \dots \oplus 0 \\ 1 \oplus 1 \oplus 0 \oplus \dots \oplus 0 \\ 0 \oplus 1 \oplus 0 \oplus \dots \oplus 0 \\ \vdots \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}
 \end{aligned}$$

Addr of Column 127

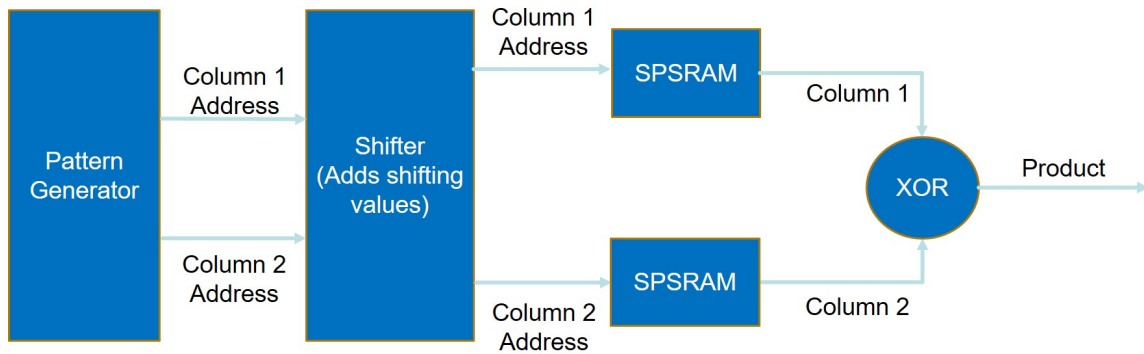
Addr of Column 126

**Figure 3-12:** The active high bits of the second vector acts as the selection switch for the parity matrix

We analyze the data structure of the the error vector  $E$  and the parity matrix  $H$ . An uncoded  $BER$  of  $10^{-3}$  is a reasonable measure of the noise rate that we can expect in a real communication channel. We check only up to hamming weight of three because the probability of getting an error of hamming weight 4 or greater is negligible at 0.00094% (Table 2.1). Therefore, the  $E$  vector will at-most contain 3 high bits out of 128. The probability of any bit to flip during transmission is equally likely in a Binary Symmetric Channel, therefore the bit flip can occur anywhere in

the code-length. This makes the error vector sparse in nature as the active high bits are not concentrated at any location. Figure 3-12 shows how the multiplication in the binary regime could be seen as a selection procedure, where the active high bit of vector  $E$  'selects' the particular column of the  $H$  matrix. These selected columns are then XORed together to get the final product. This operation can be performed by using various hardware architectures. For instance, ROM can be used to store the  $H$  matrix and then read the columns of the parity check matrix. However, ROM is hard-coded. Once fabricated, the data inside them cannot be altered, because of which, we cannot change the  $H$  matrix when we might want to decode a different type of code. An alternative approach is to use SRAMs. SRAMs are re-writable, therefore giving us the freedom of altering the  $H$  matrix at will.

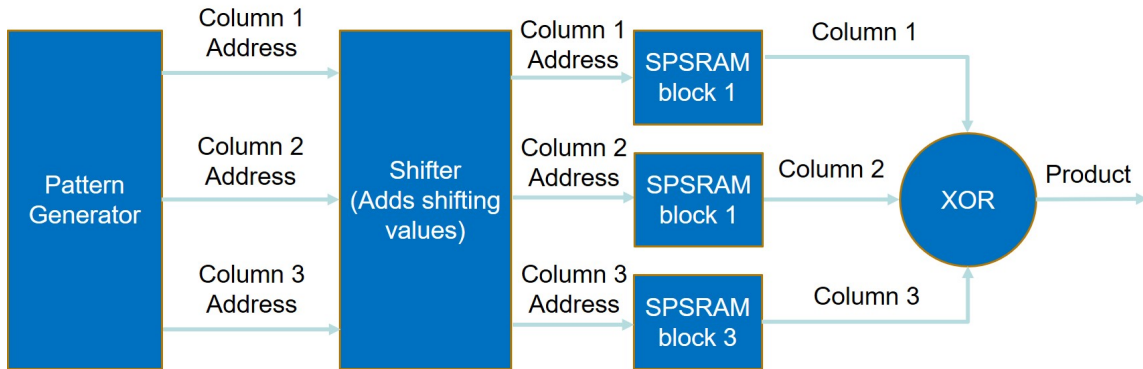
### Single-Port SRAM Implementation



**Figure 3-13:** Multiplication module in Primary block using Single-Port SRAM.

As the name suggests, a Single Port SRAM (SP\_SRAM) allows us to read or write only on one address at once. Therefore, each cycle we get one value from the output port of the SRAM.

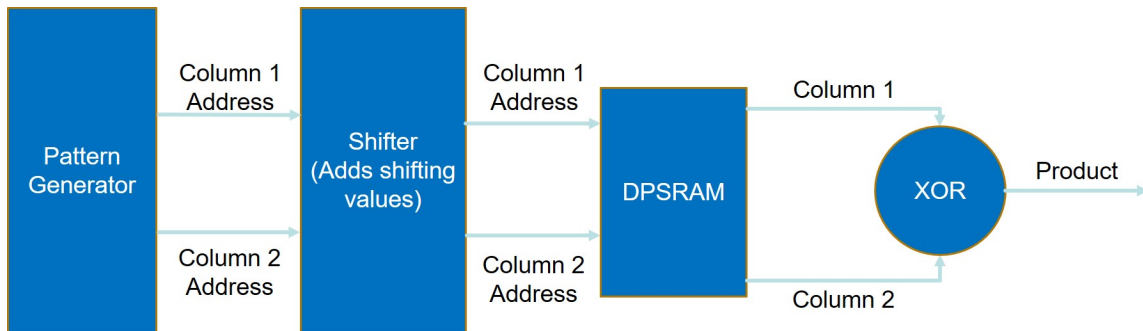
Since the Primary block only generates errors up to the weight of 2, we need at most 2 SRAM blocks to generate the final product. The secondary block, however



**Figure 3-14:** Multiplication module in Secondary block using Single Port SRAM. Since the error vector  $E$  contains only up to 3 active high bits at sparse locations within the vector, they are used to select the columns from the parity matrix which are XORed together to generate the final product.

generates errors up to the weight of 3, so to accommodate that, we need 3 SRAM blocks to have 3 simultaneous values.

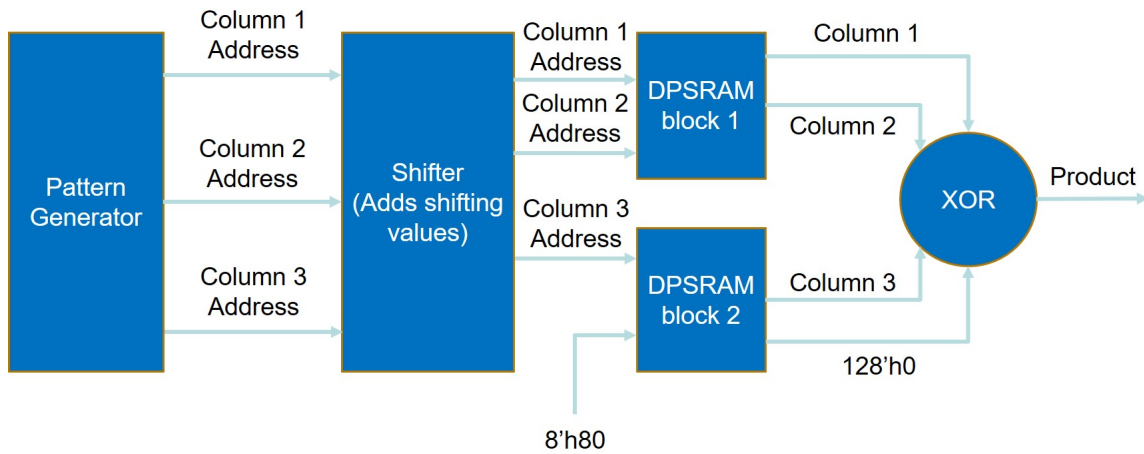
### Dual-Port SRAM Implementation



**Figure 3-15:** Multiplication module in Primary block using Dual-Port SRAM

Dual Port SRAMs (DP\_SRAM) allows us to access two memory locations at once through its two access ports, enabling us to get 2 columns of the  $H$  matrix each cycle, hence reducing the SRAM block requirement essentially by half.

In the primary block, we need at most 2 columns at once, therefore we can now just have one DP\_SRAM block instead of 2 SP\_SRAM blocks. For the Secondary



**Figure 3-16:** Multiplication module in Secondary block using Dual-Port SRAM

block, we need 3 access lines, therefore we need to have 2 DP\_SRAMs, which renders one of the ports of the second SRAM not being used. During multiplication, the second port of the second DP\_SRAM is turned off through the means of the port enable signal. This helps us save switching power, while still giving us the ability to use the two ports to write into the SRAM.

### Comparison

The DP\_SRAM provides more flexibility in the design, compared to its SP\_SRAM counterpart, as it allows for two concurrent reads as well as two concurrent writes. This helps in reducing the latency for writing into the SRAM, as well as helps in saving the area since the required number of SRAMs are halved. As shown in Figure 3-16, one of the ports of the DP\_SRAM is not given a valid column address to generate a value. One can make an argument to use one SP\_SRAM and one DP\_SRAM, to reduce redundancy in the design. However, it increases the complexity in terms of writing the data into the SRAM as it requires additional logic.

As it has been demonstrated, this implementation can be used to perform complex matrix-vector multiplication with at a very low latency of 1 cycle. This however,

comes with its own trade-offs. This method is very much dependent on the structure of the data on which the computations are performed. The sparsity in the error vector was leveraged for implementing this design of multiplication. If the sparsity of the active high bits reduce, or the number of active high bits increase, the implementation does not scale well in terms of area and power consumption. To cater for the increased number of active high bits in the error vector, more sets of SRAMs will have to be added to generate as many columns in one cycle. In addition to that, if the code-length is changed from 128-bit to 256-bit or 1024-bit, it will require a larger SRAM, which is not suitable from a size perspective of the chip, as larger SRAM sizes will require larger die area. These trade-offs leave scope for further research from an implementation standpoint.

### 3.5 GRAND Chip Implementation

The earliest iterations of the design consisted of two computation blocks, the syndrome calculator and the error generator. The syndrome calculator is responsible for generating the syndrome value by multiplying the channel output  $Y$  with the parity matrix  $H$ . The product is a zero-vector if and only if the channel output  $Y$  is a valid code-word. If the value is non-zero, the channel output  $Y$  and the syndrome value  $H.Y$  is passed onto the error generator. The error generator starts generating the errors in sequence starting from errors with Hamming weight 1 going up to the Hamming weight of 3. The channel output is abandoned if the Hamming weight of the error is greater than 3.

Since the algorithm takes the probability distribution of the error vectors into consideration, it presents an opportunity to optimize for the different hamming weights of the errors. Table 2.1 in Section 2.2 shows the probabilities of having an error of different hamming weights for multiple uncoded  $BER$  values. The error generator

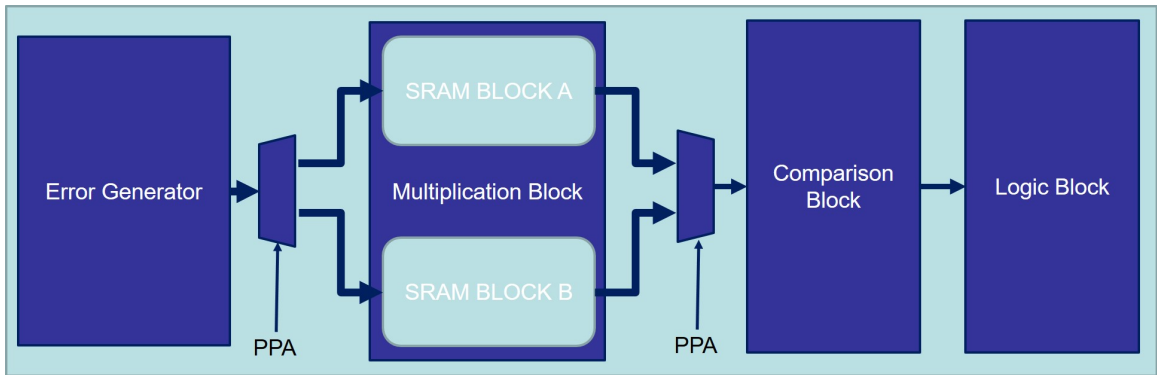
is capable of generating as many as 16 error vectors in parallel. We can figure out the number of unique error vectors that we need to check for each channel output with hamming weight  $i$  using  $\binom{n}{i}$ , where  $n$  is the code-length. Table 3.1 presents the different hamming weights of the errors along with the required number of cycles to generate them.

Hamming weight	Probability of bit-flip	No. of error vectors	No. of cycles
0	0.8798	NA	NA
1	0.1127	128	8
2	0.0072	8128	584
3	0.0003	341376	25840
4	$9.6 \times 10^{-6}$	10668000	NA

**Table 3.1:** Probability distribution of errors of different hamming weights and their cycle requirements

It is evident from Table 3.1 that 87.98% of the received code-words will not have any error. These code-words are directly sent to output from the Syndrome calculator. Out of the remaining 12.02% of the channel outputs, 11.27% of the outputs have an error with 1 bit flip, 0.72% outputs have errors with 2 bit flips, and only 0.03% outputs have an error with 3 bit flips. Another takeaway from Table 3.1 is that it takes almost  $50\times$  more cycles to generate all the error vectors with Hamming weight of three compared to error vectors one and two combined. While the error generator is processing the channel output which has an error of hamming weight 3, other channel outputs keep piling up for processing. The latency of generating error vectors of hamming weight three becomes a bottleneck. The probability distribution of the various error vectors however, allowed us to decouple the single error generator into two separate units called the Primary block and the Secondary block. The Primary block is responsible for generating errors of hamming weight one and two, whereas the Secondary block generates all the errors of hamming weight three. This approach allows us to process the channel outputs in parallel. While the channel output with 3 bit flips is being processed in the secondary block, the primary block could process

the incoming channel outputs for errors with up to 2 bit flips. Do note, that the syndrome calculator is also actively able to process the incoming channel outputs. This allows for increased parallelism in the design through pipelining, which allows multiple channel outputs to be processed at the same time. This approach also allows us to reduce our operating frequency further as the secondary block, which happens to be the bottleneck in terms of operation frequency does not need to be active for as many cycles and can be run at a lower frequency to meet the throughput target.



**Figure 3-17:** Time interleaved architecture for re-randomizing the code-books

Another goal with the design of the chip was to introduce a physical layer of security in the communication and decoding methodology. This work achieves that by re-randomizing the code-book in a time-interleaved architecture, by storing two sets of the parity matrix in two different sets of SRAMs. Consider for example two generator matrices  $G_0$  and  $G_1$  with the corresponding parity matrices being  $H_0$  and  $H_1$ . The code-words that were encoded using the generator matrix  $G_0$  can only be decoded using  $H_0$ . Similarly, the code-words encoded with  $G_1$  can be decoded with  $H_1$ . This is used as the working principle behind the re-randomizing of the code-books. Figure 3-17 shows the high level diagram of the implementation of the interleaving architecture. The two different parity matrices are stored in two different sets of SRAMs on the chip. On the basis of a tag value assigned to each channel output at

the input, the system automatically determines the appropriate H matrix and utilizes it for the processing. Additional control also allows to change the parity matrices in one of the sets while the other one is being used for decoding the incoming channel output, thereby eliminating downtime for changing the parity matrix. A random number generator is usually used to generate a string of values to determine the sequence of the parity check matrices to be used. We generate the sequence at the start of the chip operation and store it off chip. The values are then used during the operation thereby amortizing the energy costs of the operation throughout the functioning of the decoder.

### 3.6 Power and Energy Consumption

The goal of the thesis was to design and implement an energy-efficient channel decoder capable of decoding both systematic and non-systematic codes while maintaining the throughput levels conforming to 3GPP 5G standards (ETSI, 2018). We evaluate the average energy consumption per code-word by taking into consideration the probability distribution of the error vectors introduced during transmission and using the peak power of the modules that will be processing any given code-word. We know that

$$energy = power \times time$$

To calculate the energy per decoded bit, we check the average number of cycles it takes for each code-word to be processed in the respective blocks, i.e. Syndrome calculator, Primary or Secondary module. For each hamming weight  $i$ , we check the number of cycles taken by each code-word and multiply it with its probability of occurrence and the time period, to get the operation time.

$$time_i = \frac{cycles \times P(E(i))}{freq}$$

The time is then multiplied with the respective block’s power to get the energy consumed

$$energy_i = time_i \times power_b$$

where  $power_b$  is the power of the block being used to process. This is repeated for all the blocks the code-word will be processed in throughout the operation. For example, a code-word with an error of hamming weight 0 is processed in syndrome calculator only, whereas a code-word with error of hamming weight 3 is processed in syndrome calculator, primary block as well as secondary block. The total energy is then divided by the payload of the code-word  $k$  to get the energy per decoded bit. Table 3.2 shows the change of energy per bit values with changing the operating frequency of the chip. The energy per decoded bit does not change with the frequency due to the relationship between the power consumption of a block and the time taken for the computation.

$$power_b \propto freq$$

$$time_i \propto \frac{1}{freq}$$

$$power_b \times time_i \propto 1$$

Frequency (MHz)	Energy per decoded bit (pJ/bit)	Throughput (Mbps)
10	16.5	50
50	16.5	250
100	16.6	500

**Table 3.2:** Energy per decoded bit throughput for different operating frequencies of the chip

### 3.7 Results

Table 3.3 presents a comparison with the state-of-the-art implementations in published literature. We work with a code-length  $n = 128$ , whereas PolarBear (Giard

et al., 2017) uses  $n = 1024$  and Huawei (Liu et al., 2018) uses  $n = 32768$ . This work achieves a  $5\times$  lower energy per decoded bit value of 16.6 pJ/bit compared to Polarbear’s 95pJ/bit, in a noisier channel with an uncoded  $BER$  of  $10^{-3}$ , compared to PolarBear’s target uncoded  $BER$  of  $10^{-5}$ . Energy per decoded bit value was not reported for Huawei. It also achieves a  $1.33\times$  higher peak throughput of 250 Mbps compared to PolarBear’s 167.8 Mbps, although Huawei’s chip achieves a higher throughput of 2599 Mbps. We are also able to achieve a higher code-rate of 0.8, compared to code-rate of 0.5 for both PolarBear and Huawei.

Performance Metrics	This work	PolarBear	Huawei
BER	$10^{-3}$	$10^{-5}$	NA
Rate	0.8	0.5	0.5
Code-length (n)	128	1024	32768
Frequency (MHz)	50	336	1000
Throughput (Mbps)	250	167.8	2599
Energy per bit (pJ)	16.6	95	NA
Process	40 nm	28 nm	16 nm
Supported code-book	Code agnostic	CA-Polar	CA-Polar

**Table 3.3:** Performance Comparison with the State-of-the-Art Decoders for 5G Communications, Polarbear (Giard et al., 2017) and Huawei (Liu et al., 2018)

## Chapter 4

# Conclusions

To summarize, this thesis presents a hardware implementation of a Universal Noise-centric Maximum Likelihood decoder using the Guessing Random Additive Noise Decoding (GRAND) algorithm (Duffy et al., 2019). We demonstrated that our implementation of the GRAND algorithm is able to provide a  $5\times$  improvement in the energy consumption per decoded bit for an uncoded  $BER$  of  $10^{-3}$  and a  $1.33\times$  improvement in throughput while using an  $8\times$  smaller code-length and a high code-rate  $R$  of 0.8 compared to existing state-of-the-art implementations (Giard et al., 2017) (Liu et al., 2018). In addition, we presented a time-interleaved architecture to re-randomize the code-books which provides a physical layer of security. We achieved these improvements through various optimization techniques, such as (1) implementing a novel architecture to generate possible noise sequences in the decreasing order of their likelihood to conserve energy, (2) reducing latency in the computation heavy operation of matrix-vector multiplication by exploiting the sparsity of the error vectors, and (3) designing a time-interleaved architecture with built-in security which switches between multiple parity check matrices on the fly enabling re-randomization of the code-book for every code-word with zero dead zone in decoding.

This project can be extended in several possible directions, for example, the current implementation relies heavily on the SRAMs for its multiplication operations. In-memory computation techniques as presented in (Biswas and Chandrakasan, 2018) and (Agrawal et al., 2018) are interesting approaches to reduce the dependence on

SRAMs and move away from the Von-Neumann approach (von Neumann, 1993) to perform matrix-vector multiplication. We could also explore optimal syndrome calculation hardware to reduce complexity, latency, and maximize energy efficiency. It would also be interesting to utilize characteristics of modulation schemes to implement modulation-aware decoders, which shall further optimize the energy per decoded bit by efficient noise guessing. Another possible improvement could be made by adding flexibility to the error generator to support multiple channel noise models, for example, bursty errors in the channel.

## References

- Agrawal, A., Jaiswal, A., Lee, C., and Roy, K. (2018). X-sram: Enabling in-memory boolean computations in cmos static random access memories. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(12):4219–4232.
- Arikan, E. (2008). Channel polarization: A method for constructing capacity-achieving codes. In *IEEE International Symposium of Information Theory*, pages 1173–1177.
- Bell, D. A. (1966). Walsh functions and hadamard matrixes. *Electronics Letters*, 2(9):340–341.
- Biswas, A. and Chandrakasan, A. P. (2018). Conv-ram: An energy-efficient sram with embedded convolution computation for low-power cnn-based machine learning applications. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 488–490.
- DeTurck, D. (accessed: 8 August 2019). Binary matrices. <https://www.math.upenn.edu/~deturck/m170/wk8/lecture/matrix.html>.
- Direct, S. (accessed: 27 February 2020). Matrix multiplication. <https://www.sciencedirect.com/topics/computer-science/matrix-multiplication>.
- Duffy, K. R., Li, J., and Médard, M. (2019). Capacity-achieving guessing random additive noise decoding. *IEEE Transactions on Information Theory*, 65(7):4023–4040.
- Duffy, K. R., Medard, M., and Li, J. (2018). Maximum likelihood decoding by guessing noise. University Lecture. [http://ita.ucsd.edu/workshop/18/files/abstract/abstract\\_163.txt](http://ita.ucsd.edu/workshop/18/files/abstract/abstract_163.txt).
- ETSI (2018). 3GPP TS 38.212, 5G-NR-Multiplexing and channel coding.
- Gallager, R. (1962). Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28.
- Giard, P., Balatsoukas-Stimming, A., Müller, T. C., Bonetti, A., Thibeault, C., Gross, W. J., Flatresse, P., and Burg, A. (2017). Polarbear: A 28-nm fd-soi asic for decoding of polar codes. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 7(4):616–629.

- Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160.
- Liu, X., Zhang, Q., Qiu, P., Tong, J., Zhang, H., Zhao, C., and Wang, J. (2018). A 5.16gbps decoder asic for polar code in 16nm finfet. In *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*, pages 1–5.
- Masnick, B. and Wolf, J. (1967). On linear unequal error protection codes. *IEEE Transactions on Information Theory*, 13(4):600–607.
- Muller, D. E. (1954). Application of boolean algebra to switching circuit design and to error detection. *Transactions of the IRE professional group on electronic computers*, (3):6–12.
- Médard, M. (2020). Is 5 just what comes after 4? *Nature Electronics*, 3(1):2–4.
- Peterson, W. W. and Brown, D. T. (1961). Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235.
- Reed, I. S. (1953). A class of multiple-error-correcting codes and the decoding scheme. Technical report, Massachusetts Institute of Technology Lexington Lincoln lab. <https://apps.dtic.mil/dtic/tr/fulltext/u2/025814.pdf>.
- Reed, I. S. and Solomon, G. (1960). Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304.
- Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423.
- Spectre (accessed: April 22, 2020). <https://github.com/yp-mit/spectre>.
- von Neumann, J. (1993). First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75.

## CURRICULUM VITAE

