Improve the Usability of Polar Codes: Code Construction, Performance Enhancement and Configurable Hardware

by

Shuanghong Sun

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in the University of Michigan
2017

Doctoral Committee:

Associate Professor Zhengya Zhang, Chair
Professor Michael P. Flynn
Associate Research Scientist Darren S. McKague
Professor S. Sandeep Pradhan
Assistant Professor Thomas A. Schwarz

Shuanghong Sun

shuangsh@umich.edu

ORCID iD: 0000-0003-1158-4331

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Professor Zhengya Zhang for his guidance and support throughout the years. All his students appreciate his inspiration on research ideas, his dedication to the group and his consideration to the students. His advices and encouragements made it possible for me to grow in the professional aspects, and to have a broader sight and a maturer mind. I would like to thank Professor Michael Flynn, Professor Sandeep Pradhan and Doctor Darren McKague for the contributive discussions in my research projects and Professor Thomas Schwarz for participating in my dissertation committee and evaluating my dissertation.

My research was sponsored by Intel, NASA, NSF (CCF-1054270) and the University of Michigan. I would like to thank my sponsors for the support.

To receive education in the University of Michigan and to work with many talented individuals are precious fortune to me. Collaboration with colleagues gives me valuable experience, which is provided by Dr. Youn Sung Park, Dr. Phil Knag, Sung-Gun Cho, Thomas Chen, Yaoyu Tao, Dr. Chunyang Zhai and Dr. Yong Lim. I would like to thank Dr. Chia-Hsiang Chen and Dr. Jung Kuk Kim for the mentorship and patience. I would like to thank my group mates Shiming Song, Wei Tang, Chester Liu, Ching-En Lee and Teyuh Chou, who supported me by not only technical suggestions but also enjoyable friendship.

Finally, I present my deepest appreciation to my parents Ming Sun and Yongfen Sun, who contribute everything possible to their only child, especially on her educa-

tion. I would like to thank my husband Yefei Wang for his never-ending support and encouragement over 3-hour jet lag, and I appreciate the struggle and smoothness we went through.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Error-correcting codes (ECC) have been widely used for forward error correction (FEC) in modern communication systems to dramatically reduce the signal-to-noise ratio (SNR) needed to achieve a given bit error rate (BER). Newly invented polar codes have attracted much interest because of their capacity-achieving potential, efficient encoder and decoder implementation, and flexible architecture design space. This dissertation is aimed at improving the usability of polar codes by providing a practical code design method, new approaches to improve the performance of polar code, and a configurable hardware design that adapts to various specifications.

State-of-the-art polar codes are used to achieve extremely low error rates. In this work, high-performance FPGA is used in prototyping polar decoders to catch rare-case errors for error-correcting performance verification and error analysis. To discover the polarization characteristics and error patterns of polar codes, an FPGA emulation platform for belief-propagation (BP) decoding is built by a semi-automated construction flow. The FPGA-based emulation achieves significant speedup in large-scale experiments involving trillions of data frames. The platform is a key enabler of this work.

The frozen set selection of polar codes, known as bit selection, is critical to the error-correcting performance of polar codes. A simulation-based in-order bit selection method is developed to evaluate the error rate of each bit using Monte Carlo simulations. The frozen set is selected based on the bit reliability ranking. The resulting code construction exhibits up to 1 dB coding gain with respect to the conventional bit selection.

To further improve the coding gain of BP decoder for low-error-rate applications, the decoding error mechanisms are studied and analyzed, and the errors are classified based on their distinct signatures. Error detection is enabled by low-cost CRC concatenation, and post-processing algorithms targeting at each type of the error is designed to mitigate the vast majority of the decoding errors. The post-processor incurs only a small implementation overhead, but it provides more than an order of magnitude improvement of the error-correcting performance.

The regularity of the BP decoder structure offers many hardware architecture choices. Silicon area, power consumption, throughput and latency can be traded to reach the optimal design points for practical use cases. A comprehensive design space exploration reveals several practical architectures at different design points. The scalability of each architecture is also evaluated based on the implementation candidates.

For dynamic communication channels, such as wireless channels in the upcoming 5G applications, multiple codes of different lengths and code rates are needed to fit varying channel conditions. To minimize implementation cost, a universal decoder architecture is proposed to support multiple codes through hardware reuse. A 40nm length- and rate-configurable polar decoder ASIC is demonstrated to fit various communication environments and service requirements.

CHAPTER I

Introduction to Polar Code

1.1 Background and Motivation

Massive amounts of data are being transmitted electronically over communication channels in our daily lives. In order to control errors induced by noise, or equivalently to reduce the energy of transmission at an acceptable error rate, channel coding has been widely deployed in modern digital communications. The simplified work flow of channel coding is shown in Figure 1.1. The data sender encodes messages into codewords, where the number of bits in the codeword is more than that in the message, but the correspondence between the message and the codeword is one-to-one. The codeword is then transmitted over a communication channel, where the channel introduces noise to the transmitted codeword, possibly causing bit flips in the received word. The receiver uses a decoder to recover the message from the received word.

There are a few aspects to evaluate the quality of a channel code. Let us first assume a perfect decoder that can always find the closest codeword from the received word, so that decoding error can be corrected if the codeword and the received word are not too far away from each other. However a perfect decoder can be too costly to be practical, so an ideal code is one that not only offers a good error-correcting capability, but also lends itself well to an efficient decoder design. In the most demanding

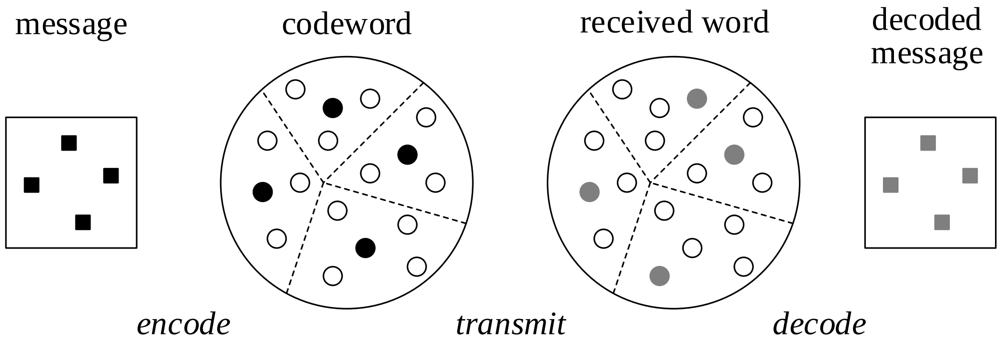


Figure 1.1: Channel coding work flow.

applications, the decoders need to provide a short decoding latency and a high data rate.

In the recent decades, low-density parity-check (LDPC) codes and turbo codes have been widely used in digital communication, because of their good error-correcting performance and low complexity. They are called capacity-approaching codes because the gap between the code performance and the Shannon limit is very small. Recently, polar codes have been discovered to have even lower complexity and capacity-achieving performance. Much research effort has been put into moving polar codes towards practice. In this dissertation, algorithm, architecture and implementation co-design of polar decoders will be presented to demonstrate the feasibility of polar codes in next-generation digital communication.

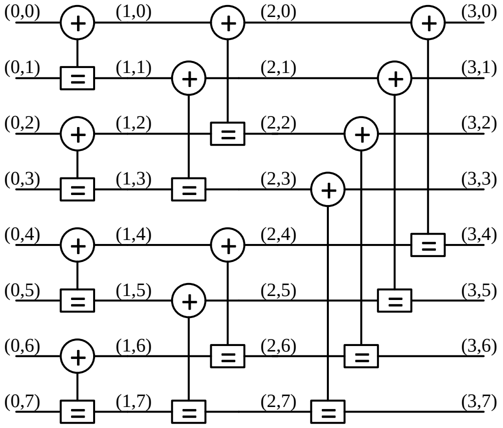
1.2 Code Structure

Newly invented polar codes [1] have attracted much interest because of their promising capacity-achieving potential and efficient encoder and decoder implementation. Compared to the state-of-the-art turbo codes and LDPC codes, the factor graph of any length $N = 2^n$ polar code is predefined, there is no polarity check bits.

A polar code is described by an $N \times N$ generator matrix G, where $N = 2^n$, and G is the n-th Kronecker power of matrix $\left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right]$ [1]. The G matrix of an 8-bit polar code

$$\begin{array}{c|cccccccc|}
 & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\
u_0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
u_3 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
u_4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
u_5 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
u_6 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
u_7 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}$$

(a)



(b)

Figure 1.2: (a) H matrix of an 8-bit polar code, and (b) an 8-bit polar encoder.

is shown in Figure 1.2(a), along with the factor graph that describes the encoder in Figure 1.2(b). In the factor graph, an encoding PE includes a "+" and a "=": the "+" symbol represents a modulo-2 addition, and the "=" symbol represents a pass through.

To encode, an N-bit message $\mathbf{u}$ is inputed from the left hand side of the factor graph, and an N-bit codeword $\mathbf{x}$ is obtained from the right hand side, i.e., $\mathbf{x} = \mathbf{u}G$. The codeword $\mathbf{x}$ is modulated and sent over a communication channel and the channel injects noise to the codeword and produces noisy codeword $\mathbf{y}$. A polar decoder will attempt to recover $\mathbf{u}$ from $\mathbf{y}$. The decoding can be visualized using the same factor graph shown in Figure 1.2(b), except that the input $\mathbf{y}$ is received from the right hand side, and the decoded message $\hat{\mathbf{u}}$ is obtained from the left hand side.

1.2.1 Channel Polarization

In a polar code of a sufficiently long block length, the bit channel that an individual message bit passes through (the channel includes the encoder) polarizes to be either highly reliable or highly unreliable, an effect known as channel polarization [1]. Figure 1.3 illustrates the polarization effect of a $N = 64$ code. To make use of channel

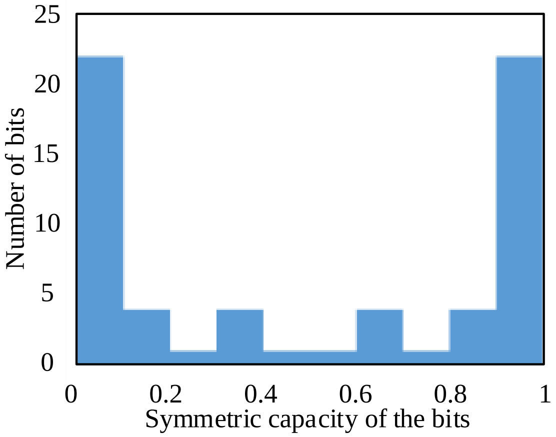


Figure 1.3: Bit channel capacity of a $N = 64$ polar code.

polarization, the bits that correspond to the unreliable channels will be frozen, i.e., send a known value such as 0, and the bits that correspond to the reliable channels will be used to send information.

The code rate of a polar code depends on the number of bits in $\mathbf{u}$ that are frozen. The set of indices of the information bits is called the information set, marked as $\mathcal{A}$, and its complement $\mathcal{A}^c$ is the frozen set. $\mathcal{A}$ is a subset of $\{0, 1, ..., N-1\}$, such that u_i is an information bit if $i \in \mathcal{A}$. Deciding the information set $\mathcal{A}$ is called bit selection. Bit selection has a strong influence on the error-correcting performance of polar codes.

1.2.2 Recursive Code Construction

The generator matrix G can be formulated recursively, i.e., $G_N = F^{\otimes n} = F^{\otimes n-1} \otimes F = G_{N/2} \otimes F$. More generally, G_M is a sub-matrix of G_N if $M < N$. The factor graphs of codes with different lengths are shown in Figure 1.4. The factor graph of $N = 2$ is also the basic computing node shown in Figure 1.4(a). The stage indices and bit indices of each node are marked on Figure 1.4(a), where $j \in [0, n-1]$ and $i = p \cdot 2^{j+1} + q$, where $p \in [0, 2^{n-j-1} - 1]$ and $q \in [0, 2^j - 1]$. An N-bit factor graph

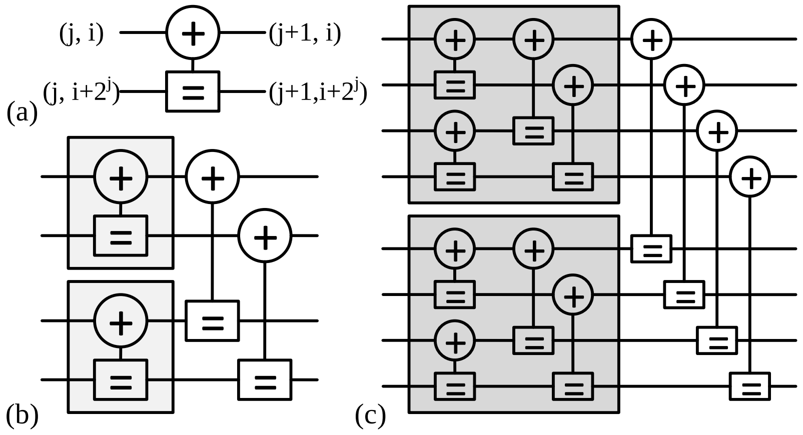


Figure 1.4: Factor graphs of (a) N=2, (b) N=4 and (c) N=8.

consists of $\log N$ stages of nodes and each stage contains $N/2$ nodes. A factor graph of length M is a subgraph of that of length N if $M < N$, as shown in Figure 1.4.

1.3 Decoding Schemes

Successive cancellation (SC) and belief propagation (BP) are the two main decoding algorithms of polar codes. SC has better error-correcting performance over BP [2], however the SC algorithm decodes bit by bit in a serial manner, so the latency of SC decoding is $\mathcal{O}(N)$ [3, 4]. BP on the other hand, uses a flooding schedule to allow N messages to be passed in parallel, thereby reducing the decoding latency to $\mathcal{O}(\log N)$. Other decoding methods are mostly derived based on these two. Much effort has been made to improve the error-correcting performance of polar codes, such as list SC decoding [5] that preserves a list of candidate decoding decisions, applying BP calculation in SC scheduling [6], and concatenation with outer codes [7–9]. Implementations based on the above algorithms [10, 11] have shown performance improvement of polar codes, at the cost of hardware requirements and/or design complexity.

1.3.1 Successive Cancellation

In SC decoding, the message bits u_0 to u_{N-1} are decoded successively. If $i \in \mathcal{A}^c$, $\hat{u}_i = 0$; otherwise $\hat{u}_i$ is decoded by the maximum likelihood decision rule:

$$\hat{u}_i = \begin{cases} 0 & \text{if } \frac{P(y,\hat{u}_0^{i-1}|u_i=0)}{P(y,\hat{u}_0^{i-1}|u_i=1)} > 1 \\ 1 & \text{otherwise} \end{cases}$$

where $P(y, \hat{u}_0^{i-1}|u_i = b)$ refers to the probability that the received vector is y and the previously decoded bits being $\hat{u}_0$ through $\hat{u}_{i-1}$, given the current bit being b, $b \in \{0, 1\}$. The ratio of the probability given $u_i = 0$ over the probability given $u_i = 1$ is the likelihood ratio (LR) of bit u_i [12]. In particular, the data processing diagram of an $N = 8$ SC decoding is shown in Figure 1.5, and the LR is calculated below.

$$LR_{j,i} = \begin{cases} f(LR_{j+1,i}, LR_{j+1,i+2^j}) & \text{if } B(j,i) = 0 \\ g(\hat{s}_{j,i-2^j}, LR_{j+1,i-2^j}, LR_{j+1,i}) & \text{if } B(j,i) = 1 \end{cases}$$

where j is the stage index and i is the bit index, and $\hat{s}_{j,i} \in \{0, 1\}$ is the encoded node based on $\hat{u}_i$'s, i.e., $\hat{s}_{1,0} = \hat{u}_0 \oplus \hat{u}_1$; $B(j, i) \equiv \lfloor i/2^j \rfloor \mod 2$; and

$$f(a, b) = \frac{1 + ab}{a + b}$$
$$g(\hat{s}, a, b) = a^{1-2\hat{s}} b$$

To simplify calculation carried out by hardware, log likelihood ratio (LLR) are introduced by taking logarithm of LR, so that f and g functions can be simplified to sum-product functions [12]:

$$f(a, b) = 2 \tanh^{-1}(\tanh(a/2) \tanh(b/2)) \tag{1.1}$$
$$g(\hat{s}, a, b) = a(-1)^{\hat{s}} + b$$

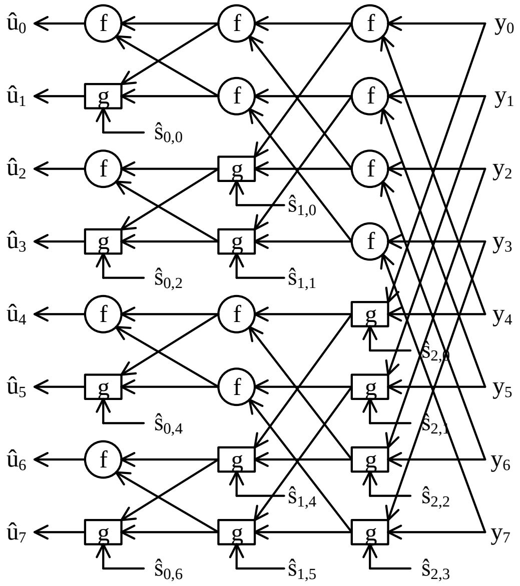


Figure 1.5: SC decoding process of the 8-bit polar code.

f is further simplified by min-sum approximation, and f and g functions can be reduced to:

$$f(a,b) \approx \text{sign}(a)\text{sign}(b)\min(|a|,|b|) \tag{1.2}$$

$$g(\hat{s},a,b) = \begin{cases} b+a & \text{if } \hat{s}=0 \\ b-a & \text{if } \hat{s}=1 \end{cases}$$

Therefore, only adders are needed to implement a decoder in hardware.

The data dependency between bit i and all the previous bits 0 through $i-1$ dictates the order of the decoding to be serial, which limits the latency to be proportional to N. In particular, the scheduling of a $N=8$ SC decoding is shown in Table 1.1 and the latency of decoding an N bit code is $2N-2$.

Table 1.1: SC Decoding Schedule of the 8-bit Polar Code.

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14
stage 3	$f_{3,0}$ $f_{3,1}$ $f_{3,2}$ $f_{3,3}$							$g_{3,4}$ $g_{3,5}$ $g_{3,6}$ $g_{3,7}$						
stage 2		$f_{2,0}$ $f_{2,1}$			$g_{2,2}$ $g_{2,3}$				$f_{2,4}$ $f_{2,5}$			$g_{2,6}$ $g_{2,7}$		
stage 1			$f_{1,0}$	$g_{1,1}$		$f_{1,2}$	$g_{1,3}$			$f_{1,4}$	$g_{1,5}$		$f_{1,6}$	$g_{1,7}$

1.3.2 Belief Propagation

The BP algorithm decodes bits u_0 to u_{N-1} in parallel. BP decoding works by passing the frozen set information from left to right (in R propagation or simply R-prop) and passing the channel output $\mathbf{y}$ from right to left (in L propagation or simply L-prop) following the factor graph. One R-prop and one L-prop constitute a decoding iteration. Convergence can usually be reached after a few iterations. It is customary to permute the original factor graph in Figure 1.2(b) to the form shown in Figure 1.6 in a bit-reversal manner [13], so that the wiring between stages are kept the same to simplify a time-multiplexed implementation.

Note that although SC and BP decoding work on the same factor graph, the major difference between the two is that BP does not impose a sequential order of decoding, and the messages are “flooded” across the factor graph.

Unlike SC where there is only one LLR for each node on the factor graph, there are two LLRs representing each node in BP, namely the left-bound messages (L messages) and right-bound messages (R messages). The basic computational node used in BP decoding is shown in Figure 1.7 [13], where j is the stage index and i is the bit index.

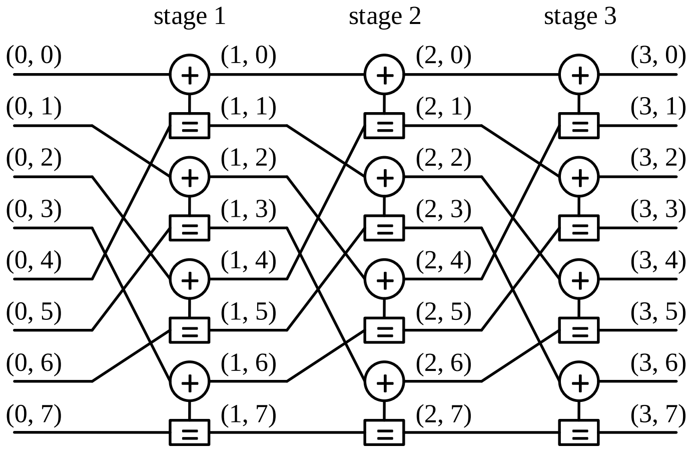


Figure 1.6: Permuted factor graph of the 8-bit polar code.

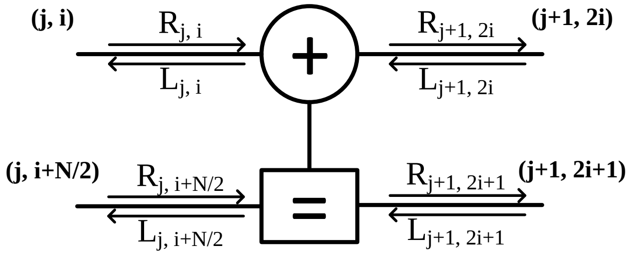


Figure 1.7: Basic computation node of a BP decoder.

The L messages and the R messages are calculated by

$$\begin{aligned}
L_{j,i} &= f(L_{j+1,2i+1} + R_{j,i+N/2},\ L_{j+1,2i}) \\
L_{j,i+N/2} &= f(L_{j+1,2i},\quad R_{j,i}) \ + \ L_{j+1,2i+1} \\
R_{j+1,2i} &= f(L_{j+1,2i+1} + R_{j,i+N/2},\ R_{j,i}) \\
R_{j+1,2i+1} &= f(L_{j+1,2i},\quad R_{j,i}) \ + \ R_{j,i+N/2}
\end{aligned} \tag{1.3}$$

where the f function is identical to the f function used in SC decoding, i.e., $f(a, b) \approx \text{sign}(a)\text{sign}(b)\min(|a|, |b|)$.

Although Figure 1.7 indicates that a node computes four output messages at a time – two L messages and two R messages, the calculation can be made uni-directional at any given time. That is, in an R-prop, a node computes only two R messages; and

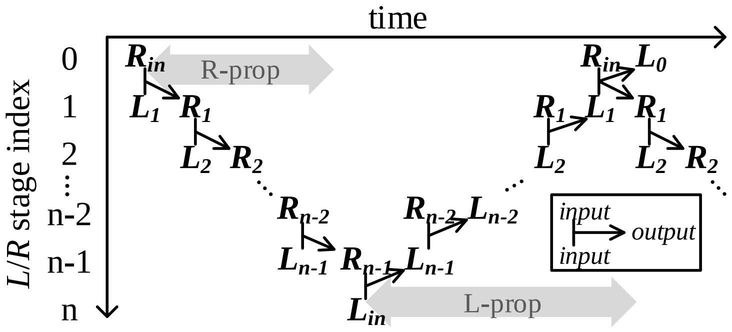


Figure 1.8: Processing flow of iterative BP decoding.

Table 1.2: BP Decoding Schedule of the 8-bit Polar Code.

Iteration	1					...	n_{it}				
CC	1	2	3	4	5	...	1	2	3	4	5
R-prop stage	1	2				...	1	2			
L-prop stage			3	2	1	...			3	2	1

in an L-prop, a node computes only two L messages. In addition, the calculations in R-prop and L-prop are identical, enabling the same hardware to be used in both R-prop and L-prop.

A decoding iteration starts with R-prop from stage 1 to stage $\log N - 1$ to propagate frozen set information, followed by L-prop from stage $\log N$ to stage 1 to propagate channel outputs, as shown in Figure 1.8. The loop of R-prop followed L-prop is iterated n_{it} times. The BP scheduling of $N = 8$ is shown in Table 1.2. At the end of each iteration, the $L_{0,i}$ messages, or simply $\boldsymbol{L_0}$, produced by stage 1 in the L-prop are taken as the soft decisions. The signs of the soft decisions are the hard decisions. The FER and BER improve with more iterations. The decoding latency is $n_{it}(2 \log N - 1)$, assuming each stage being processed in parallel and n_{it} iterations are performed.

1.3.3 Comparison of SC and BP

Apart from aforementioned differences of the two decoding schemes in throughput and latency, the implementation cost and error-correcting performance are the other two aspects of interest. To maximize SC throughput, a total number of $N/2$ hardware computational nodes are required, with each node able to calculate the f and g function. Similarly for a stage-parallel BP decoder, the same amount of arithmetic logic is needed, except that the computational logic is fully used in BP, but partially used in SC in most cases. Memory needs to hold the intermediate results for future use. The maximum amount of data need to be stored by SC is N LLR, however the requirement grows to $N \log N$ for BP. Due to the complicated scheduling of SC, the control logic of a SC decoder is more complex than that of BP.

Recall that the polarization effect is a result of SC decoding is performed, and SC exhibits better error-correcting performance than BP [2]. The conventional bit selection is done based on a BEC channel for a SC decoder. Not surprisingly, the same bit selection yields worse error-correcting performance for BP decoding in a practical additive white Gaussian noise (AWGN) channel. In Figure 1.9, the frame error rate (FER) and bit error rate (BER) of the $(256, 128)$ polar code using SC and BP decoding in an AWGN channel are compared [14]. BP decoding requires an approximately 0.3 dB higher signal-to-noise ratio (SNR) to achieve the same FER. A better bit selection is needed to improve the error-correcting performance of BP decoding to match SC decoding.

It is difficult to analytically derive the optimal bit selection for BP decoding, because the flooding of messages becomes intractable after a few steps without any approximation.

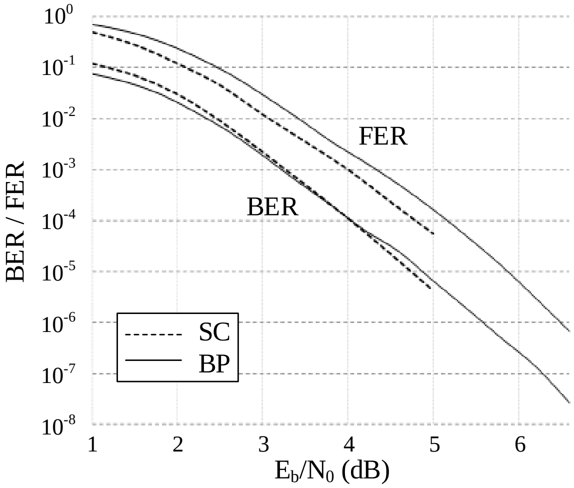


Figure 1.9: FER and BER of a $(256, 128)$ polar code decoded by SC and BP in an AWGN channel.

CHAPTER II

Design of Error-Correcting Codes using FPGA

2.1 FPGA Emulation for High-Performance ECC

FPGAs have been widely used in practice to prototype ECC decoders [13,15–23]. An FPGA can be easily reconfigured to evaluate different architecture and quantization choices. As a hardware platform, FPGA implementation truthfully captures the hardware resource utilization, placement and routing complexity, timing, and critical paths. After the register-transfer level (RTL) is functionally proven on FPGA, it can be easily translated to an application-specific integrated circuit (ASIC) implementation. For these reasons, FPGA prototyping has become a necessary step before an ASIC chip implementation.

Another important use of FPGAs is to emulate the ECC decoder and evaluate its BER performance. Software simulations of random test vectors together with corner cases are often sufficient for functional verification, but a high-performance ECC decoder needs to be verified down to very low BER levels, e.g., 10^{-12} or even 10^{-15}. It is the very rare events that dictate the performance of the ECC decoders, and the nature of the rare events may not be known beforehand. Simulating ECC decoders with as many random vectors as possible is the only way to even find out these rare events. Software simulation of a high-performance ECC decoder can reach a BER of 10^{-6} to 10^{-8} within a reasonable amount of time, which is not close to what is needed by many practical applications. On an FPGA platform, an ECC decoder can be parallelized to achieve near real-time operations, allowing very low BERs to be measured.

2.1.1 Construction of the decoder emulation platform

A hardware emulation platform on FPGA is made up of four blocks, encoder, channel model, decoder, and error collector, as shown in Figure 2.1. The emulation flow consists of the following steps: 1) an encoder takes a block of binary inputs and produces a codeword (also known as a frame), and the codeword bits are translated to real values in time domain using a modulator, e.g., binary phase-shift keying (BPSK) maps 0 to $s(t)$ and 1 to $-s(t)$ for transmission; 2) the channel model generates noise to corrupt the transmitted values, and the results are the channel output; 3) a decoder recovers the input message from the channel output; 4) the decoded message is compared with the input message to check whether the channel output is decoded correctly. If the channel output is decoded incorrectly, the number of wrong bits in the frame is added to the bit error count, and the frame error count is increased by one. The bit error count and frame error count are used to calculate the BER and frame error rate. The encoder can also be replaced by a memory that stores codewords, but the number of codewords that can be stored will be limited by the memory size.

At low SNR, i.e., when the noise level is high, decoding errors will occur frequently, and it will not take many input frames to get a sufficient number of errors for an accurate estimate of BER and FER. However, at high SNR, the noise level is low, and decoding errors will happen rarely. An accurate estimate of BER and FER will take a large number of input frames and it will dictate the length of the emulation. Low BER emulation is a bottleneck in decoder emulation.

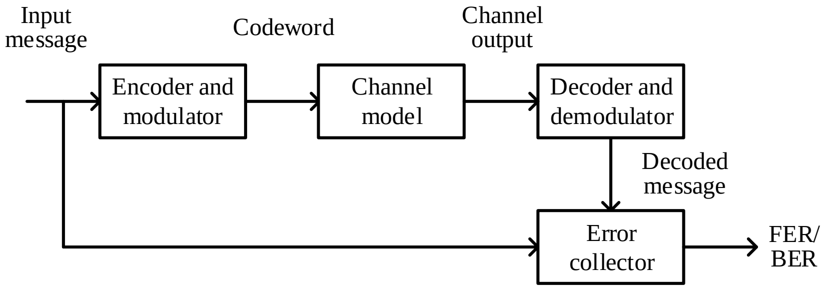


Figure 2.1: An ECC emulation platform.

2.1.2 Challenges of FPGA design and emulation

Despite the many advantages of FPGA prototyping of ECC decoders, it is however not as easy as software simulation. FPGA design requires much effort into creating the hardware architecture, RTL coding and timing verification, simulation, and iterations of synthesis, place and route to ensure timing closure, and creating the interface for input and output. To make it worse, the FPGA design may need to be changed many times to evaluate architecture and quantization choices, and every redesign involves more effort than a code change for software simulation.

In the early stages of an application design cycle, it is often not known what ECC and decoding algorithm to use, how to quantize the decoding algorithm, and not to mention hardware architecture and RTL design. Therefore, the FPGA design presents a barrier to the application community who would otherwise benefit the most from doing FPGA prototyping.

ECC decoder emulation requires test vectors to be generated on FPGA as the inputs to the ECC decoder. The test vectors need to reflect the use case of each application. Implementing realistic channel models on FPGA to generate test vectors to support real-time decoding is another challenge.

With test vector generation on FPGA, BER and FER can be measured by decoder emulation on FPGA. BER and FER serve as statistical indicators of how well an ECC works and how well the decoder is implemented, but they do not yield any insight into why decoding fail in some cases, and how to fundamentally improve code and decoder design. Trying to improve code and decoder design based on BER alone is inefficient. It is necessary to collect more information in addition to BER and FER to make FPGA emulation truly useful.

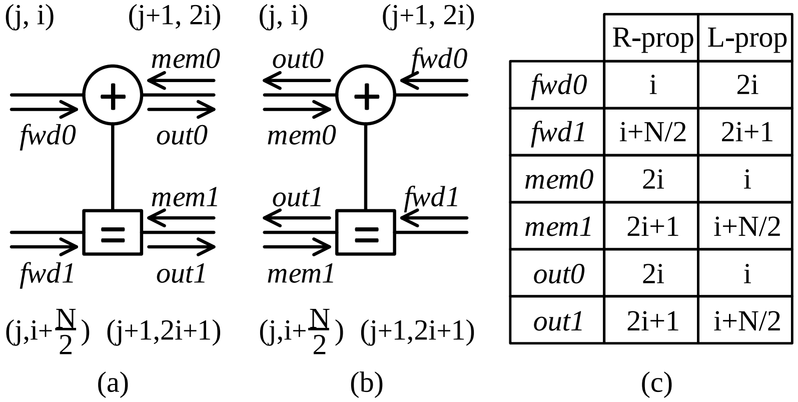
2.1.3 FPGA-aided ECC design

In this chapter, we demonstrate a semi-automated design flow based on parameterized modules to simplify the design of ECC decoder emulation on FPGA [21]. FPGA can be used to capture additional information in decoding to help uncover rare events that determine the performance of the ECC. The additional information enables new code and decoder designs for a significant improvement in BER.

We will use FPGA to study the construction of polar codes [1]. FPGA emulation reveals large variations of bit channel reliabilities. Improved code construction can be found by bit-by-bit BER measurement using FPGA emulation. Another case is to use FPGA to study the error floor problem in LDPC codes [24]. FPGA emulation uncovers rare events that underpin the error floor [18, 24]. Improved decoding algorithm can be designed to avoid the fundamental cause of the error floors for an improved BER [25]. Detailed discussions on LDPC decoder emulation are included in Appendix A.

2.2 Fast Emulation Platform for Polar Code

Polar code is provably capacity-achieving. However, the recent practical implementations of polar codes do not have better error-correcting performance than LDPC codes of similar code length and rate. In order to investigate the cause of errors of polar codes, a fast emulation on FPGA is necessary to catch rare-case events for further analysis. In this section, mapping of a column-parallel BP decoder on FPGA is



	R-prop	L-prop
fwd0	i	2i
fwd1	i+N/2	2i+1
mem0	2i	i
mem1	2i+1	i+N/2
out0	2i	i
out1	2i+1	i+N/2

Figure 2.2: Data flow of a PE in (a) R-prop and (b) L-prop, and (c) table of input and output indexing.

introduced, and the performance enhancement methods based on the FPGA platform will be presented in Chapter III.

2.2.1 Architecture Building Blocks

2.2.1.1 Processing Element

A processing element (PE) implements the functionality of equation set (1.3). Taking a close look at the equations, the calculation of R messages in R-prop and the calculation of L messages in L-prop are identical, and R-prop and L-prop take place alternately. Therefore, a PE only requires one set of hardware that is made up of two f operators to compute compare-select and two adders. Indicated by Figure 1.8, when calculating the R/L messages, the two R/L inputs are the results of the previous stage, which we name fwd as they are forwarded from the previous stage, and the other two L/R inputs are the historical results from the previous iteration, which we name mem as they are read from the memory. The data flow of a PE is shown in Figure 2.2, and the hardware mapping of a PE shown in Figure 2.3.

Although the math is expressed in a sign-magnitude form, a two's complement representation is more friendly for hardware implementation where well-design adders can be directly employed, and therefore the datapath is in two's complement. The

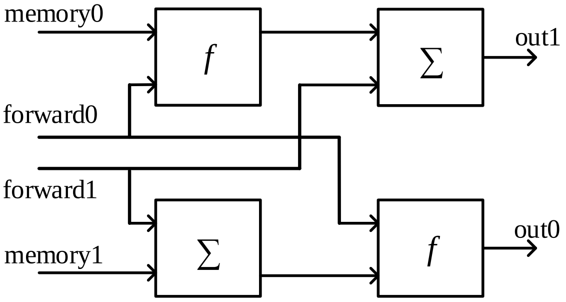


Figure 2.3: PE block diagram.

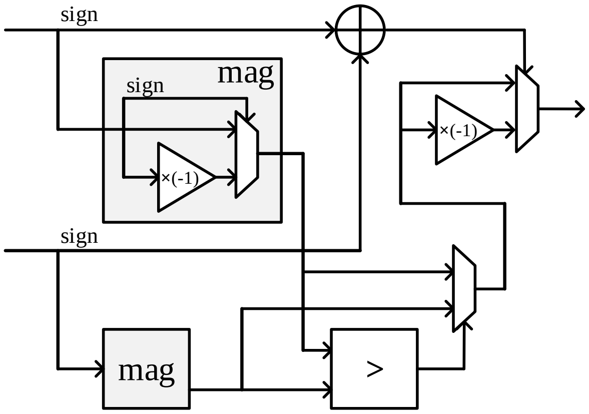


Figure 2.4: f function block diagram.

f function computes the XOR of the signs of the two inputs, and it compares and selects the smaller of the two input magnitudes as shown in Figure 2.4. The input and output of the PE are in two's complement form, so the magnitude and sign are extracted before the compare-select operation, and the sign-magnitude form is converted back to two's complement form for output.

2.2.1.2 Memory

One column of the nodes in the factor graph produce N Q-bit new messages, where Q is the message word length, which normally ranges from 4 to 6 bits, and the messages are saved in memory. Accounting for both R and L propagation, the total memory requirement is $2NQ\log_2 N$ to store all the intermediate R and L messages,

Table 2.1: BP Memory Schedule of the 8-bit Polar Code.

Cycle		1	2	3	4	5	6	7	8	9	10	11	12
Rmem	clm 1		W	—	R				W	—	R		
	clm 2			WR						WR			
Lmem	clm 0*						SD						SD
	clm 1	R				W	—	R				W	—
	clm 2	—	R		W	—	—	—	R		W	—	—
Share	clm 0*						SD						SD
	clm 1	R$_L$	W$_R$	—	R$_R$	W$_L$	—	R$_L$	W$_R$	—	R$_R$	W$_L$	—
	clm 2	—	R$_L$	WR$_R$	W$_L$	—	—	—	R$_L$	WR$_R$	W$_L$	—	—

*: Soft decision output, no storage needed —: Memory word occupied

which can dominate the area and power consumption in BP decoders.

The memory access follows a well-defined pattern, and Table 2.1 shows the memory schedule of two BP iterations, where W and R abbreviate read and write respectively, and WR stands for read-after-write in the same access. Note that the illustrated schedule assumes PEs of zero latency. In a practical implementation, the scheduling needs to be relaxed. In R-prop, L messages are consumed, and their places in memory can be replaced by newly produced R messages, and vice versa. As a result, the memory size can be reduced by half to $NQ \log_2 N$. Note that the output messages of the final column of L-prop are the soft decisions therefore do not need to be saved, so the memory size is further reduced to $NQ(\log_2 N - 1)$. Dual-port memories are required to share the L and R storage to support simultaneous read and write access.

The memory can be implemented in SRAM arrays or register files. In a highly parallel architecture, the memory is implemented in register files to support wide parallel access by many PEs; in a less parallel architecture, the memory can be implemented in SRAM to save area.

2.2.2 Decoder Architecture

The factor graph of a polar code Figure 1.6 can be mapped to a fully parallel architecture with each node mapped to a processing element and edges mapped to wires. For an N-bit polar code, a fully parallel architecture requires $\frac{N}{2}\log N$ PEs and $QN\log N$ wires for connecting the PEs. The fully parallel architecture offers the highest throughput, but the large number of PEs coupled with numerous wires complicate the design, making it less scalable. The natural way to partition the fully parallel architecture is along the stage boundaries. Such a partition results in a stage-parallel architecture that utilizes only one column of $\frac{N}{2}$ PEs for a N-bit polar code. The column of PEs will be time-multiplexed between $\log N$ stages, sacrificing throughput by a factor of $\log N$ but reducing the implementation complexity by approximately the same factor. The reduction in complexity is an important consideration as it ensures that a decoder for a sufficiently long code can be mapped to widely available FPGA platforms, and a lower complexity translates to faster hardware synthesis, placement and routing. The reduction in throughput can be recouped by using parallel hardware modules.

In the stage-parallel architecture, the wiring pattern between stages is the same, but switches are still needed to enable the use of the same PEs for both R-prop and L-prop in message-passing decoding. Specifically, two sets of PE input switches and one set of PE output switches are required implemented in N 2-to-1 MUXs to choose between R-prop and L-prop, as shown in Figure 2.5. Appropriate PE inputs also need to be chosen for different stages within one iteration. The inputs from forwarding are selected by N 3-to-1 MUXs to choose among forwarding, loading test vector $\boldsymbol{L_{in}}$ at the start of L-prop, and loading frozen set information $\boldsymbol{R_{in}}$ at the start of decoding. The inputs from memory are selected by N 2-to-1 MUXs to choose between memory read and loading frozen set information at the start of decoding, as shown in Figure 2.6.

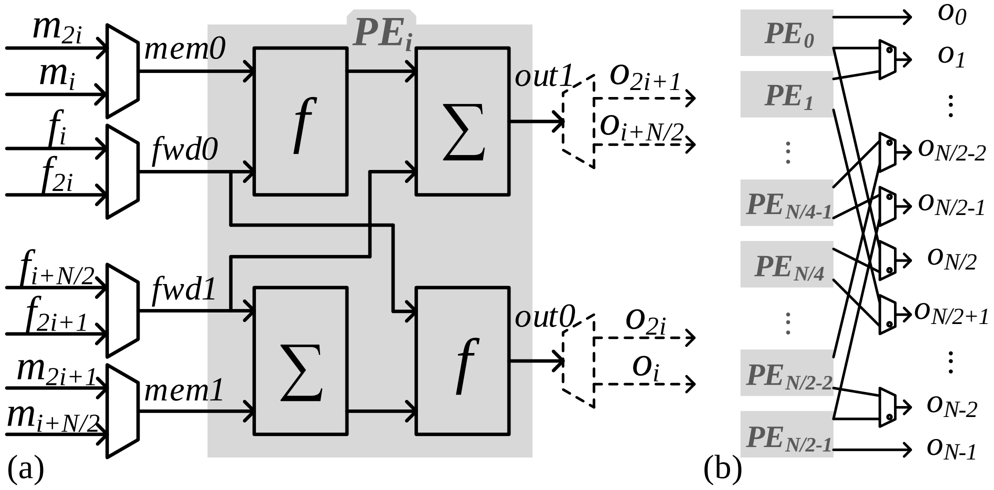


Figure 2.5: (a) PE with input and output routing, and (b) output routing of a stage of $N/2$ PEs.

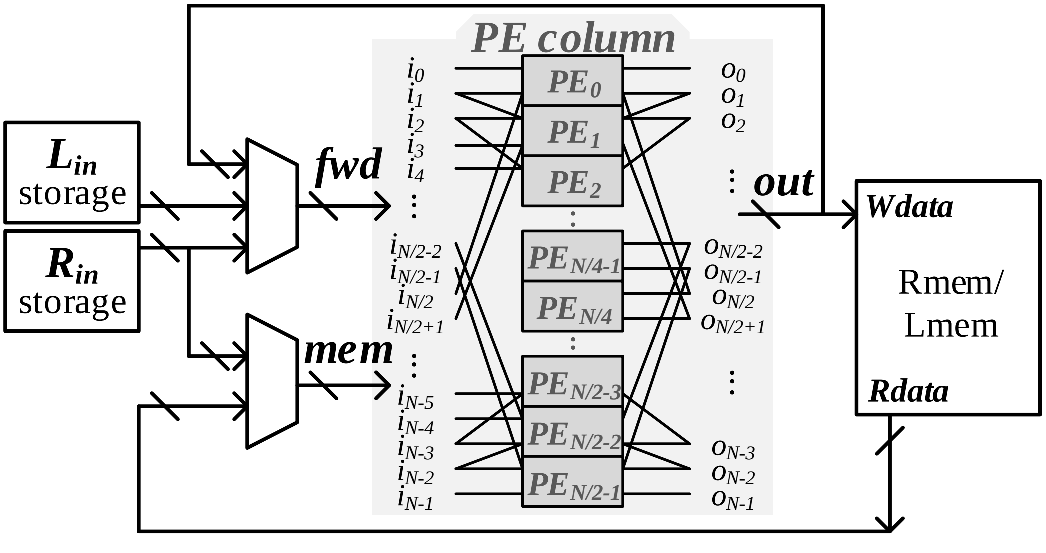


Figure 2.6: Connections between PEs and storages.

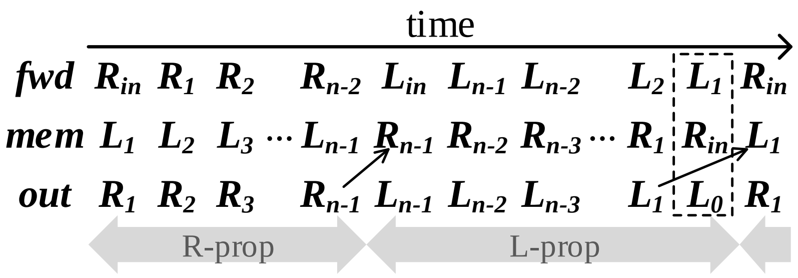


Figure 2.7: Input and output time table of the PEs in stage-parallel BP decoding.

Combining the processing flow in Figure 1.8 with the PE mapping in Figure 2.5, we draw the time table of the PEs in Figure 2.7. The last timestamp of L-prop, highlighted in dash, is unnecessary except for the last iteration, because the soft decision $\boldsymbol{L_0}$ is not useful in the subsequent calculation. Note that although $\boldsymbol{R_{n-1}/L_1}$ is forwarded from the previous stage, it goes to the $\boldsymbol{mem}$ port of the PEs, indicated by the arrows.

We use two single-port memories to implement Lmem and Rmem, and the switching between the two memories is implemented by a pair of NQ-bit-long MUXs and DEMUXs. To perform real-time emulation, an AWGN channel emulator and a built-in tester are integrated with the decoder to provide test vectors and to collect decoding errors. Our AWGN channel emulator was based on AWGN noise generators implemented using Box-Muller Transform. These AWGN noise generators can be conveniently instantiated through Xilinx LogiCORE. The AWGN noise is scaled according to the given channel SNR and added to BPSK-modulated bits in forming the input vectors for the decoder.

The decoder takes the frozen set information as R inputs ($\boldsymbol{R_{in}}$) and the channel outputs as L inputs ($\boldsymbol{L_{in}}$). R_{in} of a bit indicates whether the bit is free or frozen. R_{in} is set to 0 if the bit is free, and R_{in} is set to the maximum allowed value if the bit is frozen to 0. Lmem and Rmem entries are all initialized to 0. The datapath is implemented in a 3-stage pipeline shown in Figure 2.8. In stage one, two L (R) messages are read from Lmem (Rmem); in stage two, two R (L) messages are forwarded from pipeline

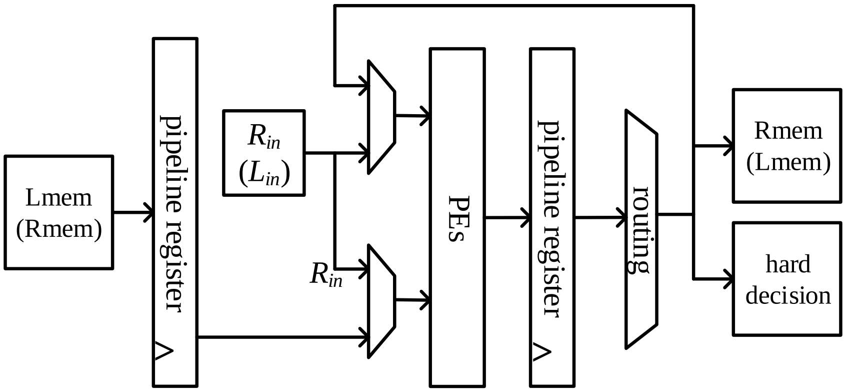


Figure 2.8: PE design in a 3-stage pipeline.

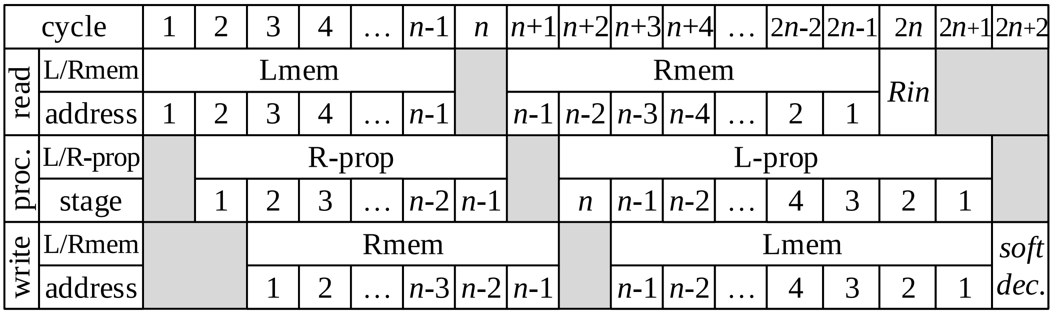
registers, and the PE calculates two R (L) messages; in stage three, the two R (L) messages are routed to the PEs for the next stage and a copy of the messages is written to Rmem (Lmem).

In a pipelined implementation, pipeline registers are inserted after the PE as output registers. The output messages of one column of nodes can be directly forwarded to the next column of nodes in high-throughput designs. Using the stage-parallel architecture, each decoding iteration takes $2\log N - 1$ clock cycles with $\log N - 1$ stages of R-prop followed by $\log N$ stages of L-prop.

2.2.3 Design Methodology

The decoder is built using a semi-automated method using a module library and an assembly script to facilitate its reuse. The module library is made up of the common blocks required for a decoder, including PE, multiplexer, and memory blocks. The blocks are parameterized to make them as general as possible to reduce the redesign effort. If parameterization cannot be easily done for some block, different versions of the block will be included in the library.

We use a script to automate the assembly of the blocks and set the module design parameters. The script takes the code block length and word length as design parameter, along with other implementation details such as desired fan-outs as inputs,



	cycle	1	2	3	4	…	*n*-1	*n*	*n*+1	*n*+2	*n*+3	*n*+4	…	2*n*-2	2*n*-1	2*n*	2*n*+1	2*n*+2
read	L/Rmem	Lmem							Rmem							*Rin*		
	address	1	2	3	4	…	*n*-1		*n*-1	*n*-2	*n*-3	*n*-4	…	2	1			
proc.	L/R-prop		R-prop							L-prop								
	stage		1	2	3	…	*n*-2	*n*-1		*n*	*n*-1	*n*-2	…	4	3	2	1	
write	L/Rmem			Rmem							Lmem							*soft dec.*
	address			1	2	…	*n*-3	*n*-2	*n*-1		*n*-1	*n*-2	…	4	3	2	1	

Figure 2.9: Scheduling of the BP polar decoder on FPGA.

and it instantiates PEs and memory blocks, and connects them. The script also sets values of the parameters. The library and script method allows one to easily construct different decoders in minutes, significantly reducing the hardware design effort. Note that synthesis and place-and-route are still needed, but they can be done efficiently using commercial FPGA design automation tools.

In order to support different experiments without having to redesign the hardware, the decoder also incorporates run-time tunable parameters, including decoding iterations and algorithmic knobs such as offset correction and iteration limits, and the channel emulator can be tuned to provide different SNRs and interpretations. These parameters are inputs of the FPGA model at run time.

The latency of the stage-parallel decoder design on FPGA is $2\log N + 2$ cycles per iteration, and Figure 2.9 shows the cycle-by-cycle schedule of one iteration. Two cycles are needed to fill the 3-stage pipeline and an extra cycle stall at cycle n+1 between R-prop and L-prop removes a read-after-write hazard. Given n_{itr} iterations and f_{clk} clock frequency, the throughput of the FPGA decoder design is $f_{clk}N/(n_{itr}(2\log N + 2))$. For example, with $f_{clk} = 100$ MHz and $n_{itr} = 15$, a 1024-bit stage-parallel decoder achieves a decoding throughput of 310 Mb/s.

The FPGA resource utilization is listed in Table 2.2 for N-bit stage-parallel decoders ($N = 256, 512, 1024$). Even the largest design listed in the table consumes only a small fraction of the available resources on a Xilinx Virtex-6 SX475T FPGA. There

Table 2.2: Hardware Utilization of Stage-Parallel BP Decoders on a Xilinx Virtex-6 SX475T FPGA.

N	256	512	1024	total available resources
Reg	13k	20k	36k	595k
LUT	22k	44k	86k	298k
Slice	6k	15k	35k	74k
RAMB36E1	78	122	216	2128
DSP48E1	12	18	33	2016

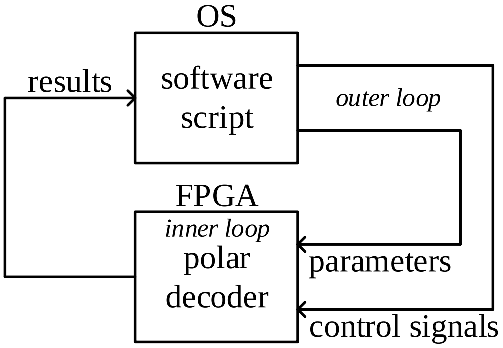


Figure 2.10: Test setup block diagram.

is ample room to support decoders for even longer codes for practical applications, where the block length is usually limited to a few Kb.

The frozen set is fed to FPGA as an N-bit input pattern, indicating whether each bit is free or frozen, and the decoder circuit design is independent of the frozen set that is used. In this work, we are interested in how each frozen set affects the error-correcting performance of the code, the results of which guides the fine-tuning of the emulation. Figure 2.10 shows the test setup for bit selection: the inner loop of the BP decoding is done on FPGA, and the outer loop is done in software which samples results from FPGA and sets run-time parameters and control signals to the decoder on FPGA.

2.3 Summary

In this chapter, we show how an FPGA platform can be used to design high-performance error-correcting codes and decoding algorithms. By emulating the decoders on FPGA, we can achieve significant speedups to measure the BER and FER down to very low levels, which is a necessity for high-performance error-correcting codes. We show how the decoders can be designed using a library and script approach to reduce the design effort and facilitate reuse. The FPGA platform will incorporate run-time configurability to support many emulation experiments using only one piece of hardware. The emulation hardware can be used to study rare events through a capture-and-replay approach, and an iterative emulation loop can be used for code and decoder design and verification.

The emulation platform is set up for BP decoding of polar codes. The application of the platform will be discussed in Chapter III. The use cases can be expanded to many other problems that require real-time emulation and study of rare events that are beyond the reach of software simulations.

CHAPTER III

Frozen Bit Selection and Tuning Knobs

3.1 Introduction to Bit Selection

When the block length of a polar code is sufficiently long, the capacity of the effective channel that each bit passes through polarizes to either close to 1 or almost 0 [1]. High-capacity and highly reliable bits are used to carry information, and low-capacity unreliable bits are frozen to 0 to guarantee a good error rate. The bits that carry information are called information bits. The selection of the set of frozen (free) bits is crucial to the error-correcting performance of polar codes. The code rate is adjusted by the size of the frozen set, without changing the factor graph of polar codes.

The selection of the frozen set, known as bit selection, is determined by the error probability or erasure probability of each bit. In SC decoding, the erasure probability of each bit can be derived for a binary erasure channel (BEC) [1]. Figure 3.1 shows the capacities of the channel each bit passes through for a $N = 1024$ polar code, with erasure rate $\epsilon = 0.5$ in Figure 3.1(a) and $\epsilon = 0.3$ in Figure 3.1(b). Note that the channel contains an encoder, and the derivation assumes SC decoding [1]. The frozen set is chosen to be the set of bits with low-capacity channels. The frozen set is dependent on the communication channel's statistical characteristic. For a fixed code rate R_c, the frozen sets are different for channels of different statistics, as evidenced

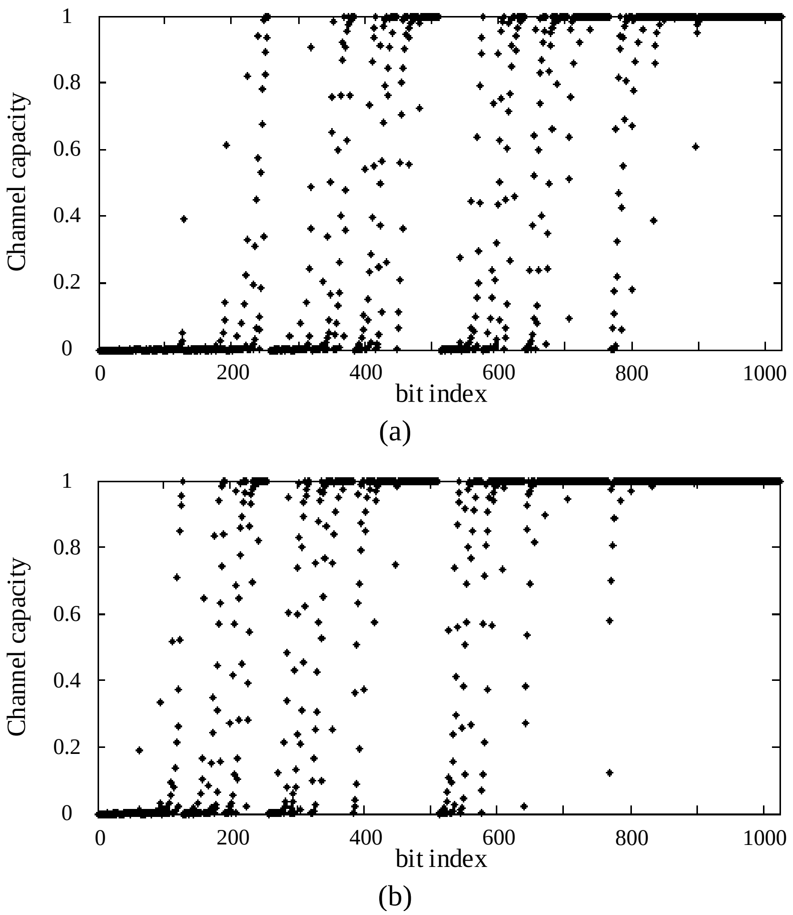


Figure 3.1: Channel capacity of the 1024-bit polar code in BEC channel with erasure rate of (a) 0.5 and (b) 0.3.

by the difference between Figure 3.1(a) and Figure 3.1(b). Moreover, the frozen set also depends on the decoding algorithm.

Arikan used the Bhattacharyya parameter as an upper bound of the erasure probability of each bit in SC decoding [1]. In a binary erasure channel (BEC), the Bhattacharyya parameter equals the erasure probability, and it can be efficiently evaluated with linear complexity. However, for an arbitrary binary memoryless channel, the complexity of the method becomes exponential in code length. Mori and Tanaka proposed density evolution to evaluate the bit error probability [26, 27], as SC decoding of each bit can be modeled as BP decoding in a tree structure. However, Mori and Tanaka's method is also bottlenecked by high computational complexity due to high memory usage that grows exponentially with code length.

Tal and Vardy extended this method by quantization to reduce the memory requirement [28]. The method obtains a lower bound and an upper bound on the bit error probability given a specified maximum number of quantization levels. The number of quantization levels needs to be high to achieve a good accuracy. Alternatively, Trifonov used Gaussian approximation in density evolution to reduce its computational complexity [29]. These simplified density evolution methods offer substantial speedup and simplification of the bit error evaluation, but they also require approximations. Another drawback of these methods is that they require the channel model to be known in advance.

In section 3.3, a simulation approach to evaluate the error probability of each bit will be introduced as an alternative method to density evolution. Inspired by Mori and Tanaka's formulation that views every step of SC decoding as BP decoding in a tree structure, we use Monte Carlo simulations of BP decoding to evaluate the error probability of each bit. Each simulation is exact and does not rely on any approximation. The simulation accounts for the finite code length, loops, numerical quantization, etc. The simulation method will be particularly useful in handling practical channels that sometimes have no closed-form mathematical representation. The bit error probabilities obtained from simulations account for practical non-idealities, including decoder implementation and its numerical precision. The simulation-based bit selection algorithm uses N sets of Monte Carlo simulations, one for each bit of an N-bit polar code.

For an N-bit polar code, the simulation-based bit selection method is done in N steps, where each step involves Monte Carlo simulations to measure the BER of one bit. At the end of N steps, a ranked BER list is produced. The simulation-based bit selection method can be easily extended to other rates. In particular, our method produces a bit error probability ranking, and designing a different rate code is simply picking number of bits to freeze following the bit error probability ranking. Our method can also be extended to other code lengths. To facilitate it, we have created a library and script approach, which takes minimal effort to construct a decoder for a different code length. We use the fast FPGA described in 2.2 to achieve significant accelerations. The bit selection process requires almost no supervision, and it can be done entirely autonomously. Compared to other published FPGA-based polar decoder emulators [30, 31], our platform is used specifically for bit selection. In addition to accelerating polar decoding, a software loop around the FPGA accelerator was added to set up the frozen patterns and collect the appropriate BERs. We have designed new approaches to speed up Monte Carlo simulations to make practical bit selections feasible. Although we use FPGA in this work, the simulation-based bit selection method can be programmed on a GPU or CPU cluster to achieve acceleration.

As a proof-of-concept, we demonstrate the simulation-based bit selection for three polar codes with block lengths of 256 bits, 512 bits, and 1024 bits. The results show up to 0.9 dB improvement in SNR (E_b/N_0) in BP decoding over the conventional bit selection.

3.2 Simple Bit Selection Methods

We first develop two simple bit selection methods and discuss their weaknesses. The simple methods are developed for BP decoding using 256-bit polar codes. The results are applicable to codes of longer block lengths.

3.2.1 One-Time Rank-and-Freeze

The bit selection should be based on the error probability of each bit – the most reliable bits are used as information bits, and the least reliable ones are frozen. To measure the error probability of each bit of an N-bit polar code, we start with the rate-1 code using the method below.

1. Set all bits as information bits.

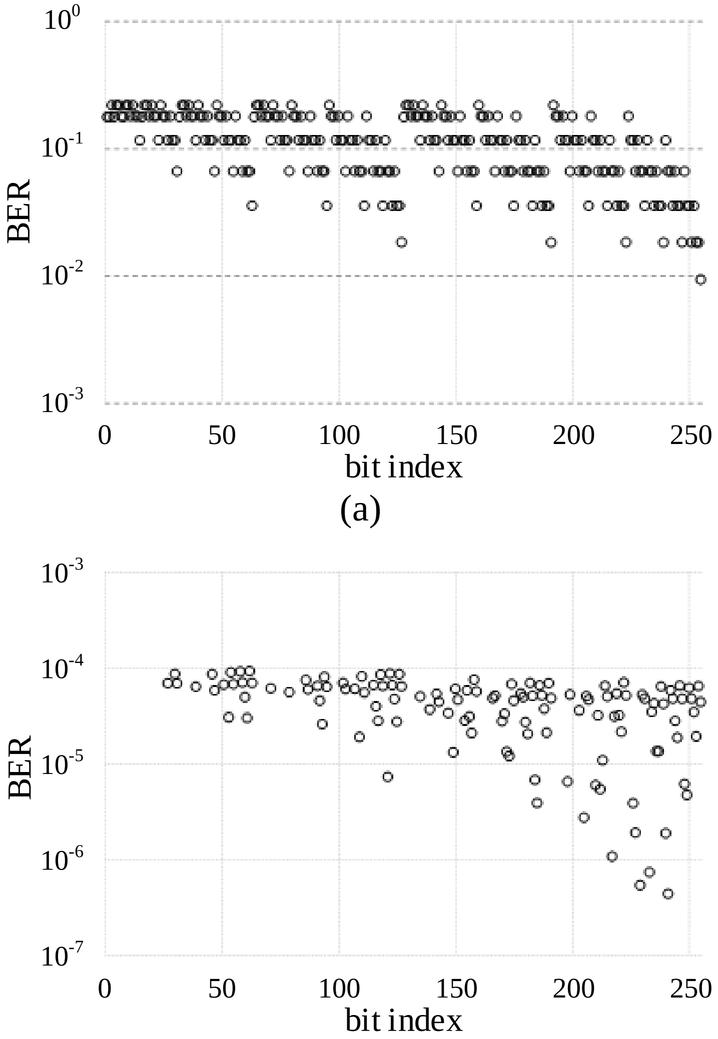
2. Run Monte Carlo BP decoding simulations and measure the error rate of each bit.

3. Rank the bits based on error rate and freeze the $N - K$ least reliable bits to obtain the bit selection for an (N, K) polar code.

We call this bit selection method one-time rank-and-freeze. Step 2 of this algorithm dominates the overall compute time, and we use FPGAs to accelerate the Monte Carlo decoding simulations. The complexity of this bit selection method is $\mathcal{O}(N_{MC})$, where N_{MC} is the number of Monte Carlo decoding simulations.

Using a 6-bit fixed-point BP decoding in an AWGN channel (at an SNR of 7 dB), the measured BER of each bit of the (256, 256) polar code is shown in Figure 3.2(a). The BER spreads over one order of magnitude. Based on the BER ranking, we select a rate-0.5 (256, 128) code with 128 bits of the worst BER frozen. The BER of each information bit of the rate-0.5 (256, 128) code spreads over two orders of magnitude, and improves by more than three orders of magnitude over the rate-1 code, as shown in Figure 3.2(b). The difference between Figure 3.2(a) and Figure 3.2(b) shows the effect of freezing low-capacity bits: by freezing high-error bits, the performance of the remaining bits can be significantly improved.

However, at a relatively high SNR of 7 dB, a BER of nearly 10^{-4} is far from satisfactory for a rate-0.5 code. Experiments at higher or lower SNR show no obvious improvement. The simple one-time rank-and-freeze method does not work well because it violates the precondition of channel polarization. The derivation of channel polarization is based on the precondition that when decoding bit i, all the former bits from 0 to $i - 1$ are already known [1]. The one-time rank-and-freeze method evaluates the BER of bit i, while allowing all the remaining bits to be free. The flooding of messages back-and-forth over the factor graph allows low-capacity bits to



(a)

(b)

Figure 3.2: BER of each bit of (a) a (256,256) polar code and (b) a (256,128) polar code.

affect the decoding of high-capacity bits. As a result, a high BER measured using this method does not necessarily indicate a low-capacity bit; and similarly, a low BER does not necessarily indicate a high-capacity bit either. Due to the unreliable BER measurement, this simple bit selection method is unsatisfactory.

3.2.2 Iterative Rank-and-freeze

To account for the inter-bit dependence, the one-time rank-and-freeze method is refined to an iterative rank-and-freeze method. The idea is that instead of ranking all bits and freezing $N - K$ bits at one time, a part of $N - K$ bits are frozen at a time. After a part is frozen, the remaining information bits are evaluated and ranked again, based on which the next part of the frozen bits are chosen, until the desired code rate is obtained. The method is described below, where N_{it} is the number of iterations to be used.

1. Set all bits as information bits.
2. For $i = 1$ to N_{it}
    (a) Run Monte Carlo BP decoding simulations and measure the error rate of each information bit.
    (b) Rank the information bits based on error rate and freeze the M_i least reliable bits.

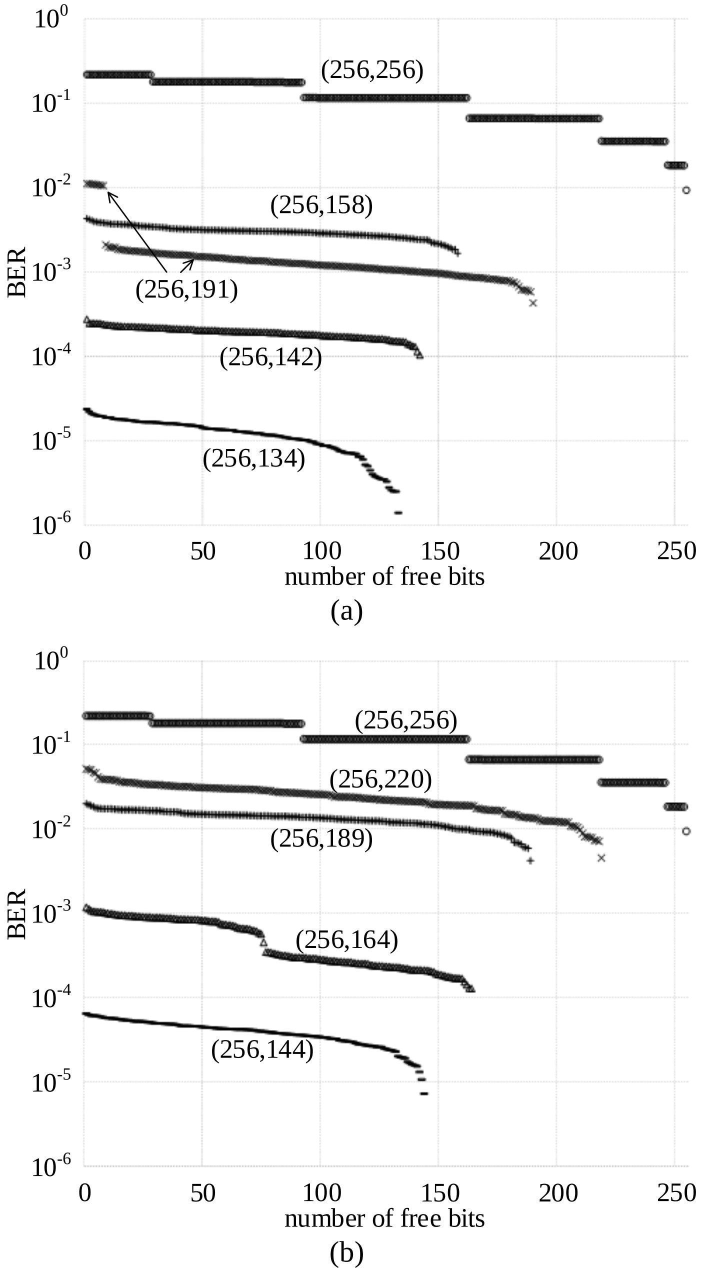
Note that the number of bits to freeze in iteration i, namely M_i, is chosen such that $\sum_{i=1}^{N_{it}} M_i = N - K$. If smaller M_i values are chosen, more iterations are needed. The complexity of the iterative rank-and-freeze algorithm is $\mathcal{O}(N_{it}N_{MC})$. We use FPGA to accelerate the inner loop (Monte Carlo simulation), and the outer loop (iteration) is done using a script that interacts with the FPGA accelerator.

For ease of illustration, the BER of each information bit of a 256-bit polar code is sorted and displayed in a distribution shown in Figure 3.3. Figure 3.3(a) and

Figure 3.3(b) are two examples of the outcomes of running four iterations of iterative rank-and-freeze algorithm following slightly different procedures. In the first example shown in Figure 3.3(a), the numbers of bits frozen in each iteration are {65, 98, 114, 122}, resulting in a (256, 191) code, a (256, 158) code, a (256, 142) code, and a (256, 134) code after the first, second, third and final iteration. In the second example shown in Figure 3.3(b), the numbers of bits frozen in each iteration are {36, 67, 92, 112}, resulting in a (256, 220) code, a (256, 189) code, a (256, 164) code, and a (256, 144) code. It is evident from both examples that the impact of frozen set on BER is significant: after a few unreliable bits are frozen, the BER of the remaining bits are enhanced. As expected, the refined method produces better bit selections. However, the choices of the number of iterations and the number of bits to freeze in each iteration play important roles.

A closer look at the results unveils more insights. First, the bit selection is different depending on how the iterative procedure is carried out and how many bits are frozen in every iteration. For example, the BER of the (256, 142) code in the first example is higher than the BER of the (256, 144) code in the second example, although the former is a lower rate code. Second, the BER of the bit selection does not improve in a monotonic fashion with more iterations. For example, in Figure 3.3(a), the BER of the (256, 158) code produced in the second iteration is not uniformly better than the BER of the (256, 191) code produced in the first iteration, although the former is derived from the latter by freezing some of the latter's information bits.

The iterative rank-and-freeze method improves upon the one-time rank-and-freeze method by freezing a portion of the bits at a time when evaluating BERs. In each iteration, the bits of the worst BER are frozen. In the next iteration, these bits will no longer affect decoding. Although the bits of the worst BER are not necessarily the bits of the lowest capacity, the bits of the worst BER contain at least a portion of the bits of the lowest capacity. This is why the iterative rank-and-freeze method



(a)

(b)

Figure 3.3: Distribution of BER of each bit of two 256-bit polar codes using iterative rank-and-freeze algorithm.

can perform better than the one-time rank-and-freeze method. However, how well the iterative rank-and-freeze method works depends on how many of the worst BER bits are frozen in each iteration. Figure 3.3(a) and Figure 3.3(b) illustrate two different outcomes depending on the number of bits frozen in each iteration. We note that the iterative rank-and-freeze method still violates the precondition of channel polarization. Therefore, the iterative rank-and-freeze method is still unsatisfactory.

3.3 In-Order Bit Selection Algorithm

In deriving channel polarization, SC decoding was used to decode polar codes in order, i.e., from u_0 to u_{N-1} [1]. u_0 is decoded first given channel outputs; next, given u_0 and channel outputs, u_1 is decoded; next, given u_0^1 (represents bits u_0 to u_1) and channel outputs, u_2 is decoded, and so on. The SC decoding of u_i depends only on the previously decoded bits u_0^{i-1} and channel outputs. If u_0^{i-1} is frozen, decoding of $\hat{u}_0^{i-1}$ is guaranteed to be correct and therefore $\hat{u}_i$ depends effectively only on channel outputs, and the capacity of u_i can be accurately measured by the error probability.

Similarly in BP decoding, as an approximation of SC, if bits u_0^{i-1} are frozen and bits u_{i+1}^{N-1} are free, the error probability of u_i can be accurately measured. The data dependency under such condition in BP is the same as SC. The error probability measurement allows us to properly rank the bits and perform bit selection. This in-order bit selection method is elaborated below.

1. For $i = 0$ to $N - 1$

   (a) If $i = 0$, then Set all bits as information bits.

   If $i \geq 1$, then freeze bits u_0^{i-1} and set bits u_i^{N-1} as information bits.

   (b) Run Monte Carlo BP decoding simulations and measure the error rate of u_i.

2. Rank the bits based on error rate and freeze the $N - K$ least reliable bits to obtain the bit selection for an (N, K) polar code.

Unlike the previous two methods, the in-order bit selection method follows the derivation of channel polarization. When evaluating the error probability of bit i, all the former bits from 0 to $i - 1$ are already frozen. In this way, the measured error probability of each bit will be reliable, as they cannot be affected by the former frozen low-capacity bits. Therefore, the bit selection using the in-order method is also reliable. The complexity of the in-order bit selection algorithm is $\mathcal{O}(NN_{MC})$.

The in-order bit selection method requires reliable measurement of error probability using Monte Carlo simulations. The number of Monte Carlo simulations, N_{MC}, depends on error rate. The lower the error rate, the more the number of Monte Carlo simulations is required to collect enough errors. In short, the complexity of our method scales inversely with BER. As the code length increases, N increases and BER decreases, which in turn increases N_{MC}. The exact complexity scaling factor depends on how BER decreases with increasing code length. If we hold code length constant and decrease code rate, BER decreases, which in turn increases N_{MC}. The exact complexity scaling factor depends on how BER decreases with decreasing code rate. To speed up in-order bit selection, we use FPGA to accelerate the inner loop (Monte Carlo simulation), and the outer loop (For $i = 0$ to $N - 1$) is done using a script that interacts with the FPGA accelerator.

The number of Monte Carlo runs N_{MC} can be adjusted for each bit. For a reliable bit, more Monte Carlo runs are required to collect enough errors to obtain a statistically significant error probability measurement, and to differentiate the reliabilities of different bits for ranking and bit selection. On the other hand, for an unreliable bit, the number of Monte Carlo runs can be reduced to save time. Based on this idea, we designed a three-pass scheme. In each pass, we run the in-order bit selection algorithm with a higher N_{MC}. The least reliable bits are identified and excluded

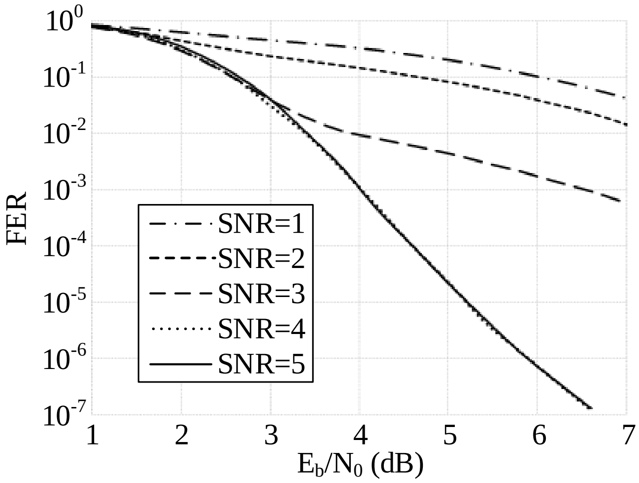


Figure 3.4: FER performance of five (256, 128) polar codes designed at SNR from 1 dB to 5 dB.

in the first pass, followed by the medium reliable bits in the second pass, and the most reliable and hard-to-distinguish bits in the third pass. Designing low rate codes requires more passes as only a small set of the best bits are chosen.

3.3.1 Optimal SNR for Bit Selection

We use an AWGN channel in the Monte Carlo simulations. The best bit selection is expected to vary across SNR. To confirm, we first obtain the bit selections using the in-order method at different SNRs, and then test the performance of the bit selections. The FER of the five (256, 128) codes with bit selections done at SNR from 1 dB to 5 dB are shown in Figure 3.4. The performance of the five codes vary widely. The codes designed at low SNRs performs worse, especially at moderate to high SNR. The codes designed at high SNRs exhibit much better performance throughout the SNR range.

To understand the implication of SNR on the bit selection, we use Table 3.1 to show the percentage of errors that are due to minimum-distance errors for six (256, 128) polar codes with different frozen sets. Each code's performance is displayed in one column: the bit selections of the first five codes are done using the in-order bit

Table 3.1: Percentage of Minimum-Distance Errors of Six $(256, 128)$ Polar Codes Decoded by BP.

		bit selection SNR					conventional* bit selection
		1 dB	2 dB	3 dB	4 dB	5 dB	
decoding SNR	1 dB	30%	20%	0	0	0	2%
	2 dB	75%	50%	5%	0	0	5%
	3 dB	95%	95%	20%	0	0	7%
	4 dB	1	95%	85%	0	0	30%
	5 dB	1	1	1	0	0	30%
	6 dB	1	1	1	0	0	30%

*: bit selection for BEC channel at $\epsilon = 0.5$

selection method at SNR from 1 dB to 5 dB, and the last column shows the results for the bit selection for the BEC channel at $\epsilon = 0.5$. Not surprisingly, at a high (decoding) SNR, the majority of the errors are due to minimum-distance errors. Therefore if the bit selection algorithm is done at a high SNR, the resulting bit selection will reduce the minimum-distance errors and increase the minimum distance of the code.

Comparing the results presented in Table 3.1 across the columns in one row, the frozen set selected at a high SNR yield fewer minimum-distance errors when simulated in the same channel condition, indicating a larger minimum distance for these codes. Therefore it is confirmed that a bit selection done at a high SNR increases the minimum distance of the code.

To run bit selection using the simulation-based in-order selection method, we need the SNR to be sufficiently high, such that the minimum-distance errors dominate the error profile. However, our method is not sensitive to SNR. The optimal design SNR is found to be around 4 dB for an AWGN channel, and it is independent of rate or code length of practical interest, i.e., from 256 to 4K.

3.3.2 Speed Improvement Using Early Termination

BP decoding is iterative and more iterations tend to improve the error-correcting performance. For the results presented so far, we have used a maximum iteration l = 15 to limit the simulation time. To further speed up the bit selection, one effective approach is early termination [32], since the decoding for the majority of the inputs converges after a small number of iterations (much less than 15). However, unlike an LDPC code, there is not a clear convergence indicator for polar codes.

We propose an approximate convergence detection by monitoring the hard decisions for consecutive iterations. If each bit obtains an identical hard decision for t consecutive iterations, the decoder is allowed to terminate. The cost of implementing consecutive decision matching is relatively low, requiring only $(t-1)$ 2-input XNOR gates per bit, and one $N(t-1)$-input AND gate at the top level. To prevent misdetection, a second criterion is added to ensure that a minimum number of iterations m is met. As Figure 3.5 shows, the code designed with early termination ($m = 3$, $t = 3$ and $l = 15$) has similar error-correcting performance as the code designed without early termination ($l = 15$), but the bit selection with early termination can be done up to 5 times faster.

3.3.3 Footprint Reduction

Part of the bit selection method is implemented on FPGA and part is implemented in software script that interacts with the FPGA. To support the in-order bit selection, we need to measure a bit's error probability. So a N-to-1 MUX, done in a tree structure, is added to the decoder on FPGA to select one of N bit decisions. To run the bit selection, the software script sets up the the frozen set for the decoder, and selects the appropriate bit to monitor its BER. Once the BER is properly measured, the software script moves to the next bit.

The decoder used for bit selection can adopt a short word length to reduce its foot-

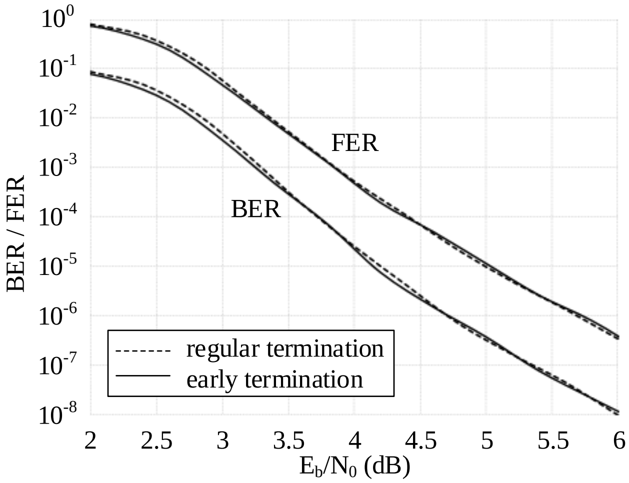


Figure 3.5: FER performance of two (1024, 512) polar codes, one designed without early termination and one designed using early termination.

print on an FPGA and to reduce the minimum clock period. For example, comparing a 6-bit decoder with an 8-bit decoder implemented in the identical stage-parallel architecture on an FPGA, the 8-bit decoder costs 10% more registers, 20% more LUTs, 30% more slices and 25% more RAM, and the minimum clock period has to be relaxed by 40%. Figure 3.6 shows similar error-correcting performance of several bit selections done using different decoders. In particular, bit selection A is done using a Q6.0 (6-bit integer including a sign bit, and 0-bit fraction) decoder; bit selection B is done using a Q6.2 decoder; and bit selection C is done using a Q8.0 decoder. The performance of the three bit selections is simulated using a Q6.0 decoder. The similar error-correcting performance justifies a smaller decoder design to be used to permit a higher degree of parallelism to speed up bit selection.

3.3.4 Performance Results

3.3.4.1 Codes of Different Block Lengths

We used the in-order bit selection method to design three polar codes, a (256, 128) code, a (512, 256) code, and a (1024, 512) code. The FER and BER of the three codes using BP decoding are compared with the bit selection for BEC channel

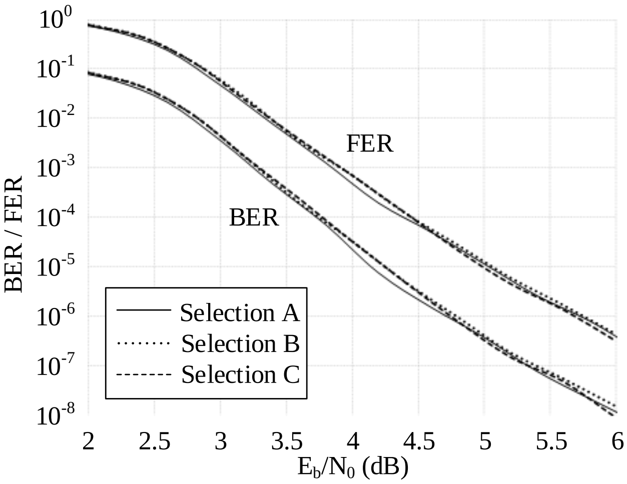
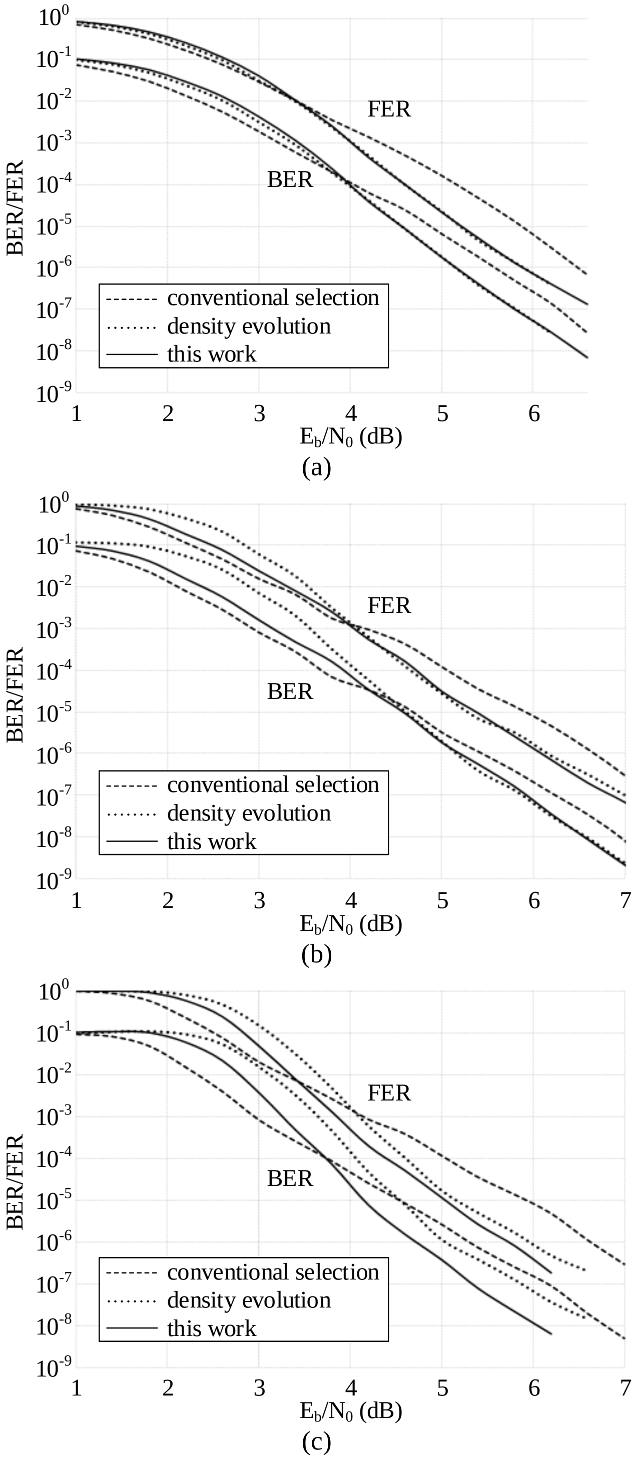


Figure 3.6: FER performance of three (1024,512) polar codes that are designed using different fixed-point quantization schemes.

at $\epsilon = 0.5$ and the bit selection by density evolution for AWGN channel at 4 dB [28] in Figure 3.7. Compared to the BEC and density evolution code designs, the codes designed by the in-order bit selection method demonstrate better coding gains at an FER below 10^{-3}. At an FER of 10^{-6}, the (256, 128) code, the (512, 256) code, and the (1024, 512) code designed by the in-order bit selection method achieve 0.6 dB, 0.6 dB, and 0.9 dB coding gain over the BEC codes. Compared to the codes designed by density evolution, our (256, 128) code has similar performance; our (512, 256) code performs slightly better at low SNR, and our (1024, 512) code shows improvement by a fraction of a dB.

3.3.4.2 Iteration limit

With early termination, the decoder has to run for at least t iterations in order to check whether the decoding has converged, so t sets the lower bound on the number of iterations (therefore $m >= t$). An upper bound, l, is set to prevent the decoder from being trapped in a case that never converges. Figure 3.8 shows FER and BER of a Q6.0 (1024, 512) polar code, with early termination ($t = 3$). The FER and BER improve significantly when l increases from 10 to 20, but the improvement



(a)

(b)

(c)

Figure 3.7: Performance of (a) (256,128) (b) (512,256) (c) (1024,512) polar codes designed with in-order bit selection.

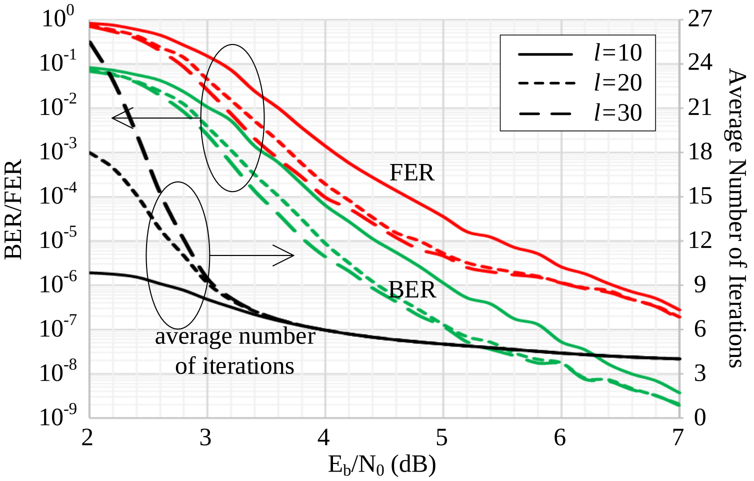


Figure 3.8: Effect of maximum number of iteration on performance and latency.

is marginal when l increases to 30. For different l values, the average number of iterations falls below 6 for SNR (E_b/N_0) higher than 4 dB, therefore the throughput becomes independent from l when $l \geq 10$.

3.3.4.3 Effectiveness on SC Decoder

The bit selection designed by our in-order method not only exhibits better error-correcting performance in BP decoding, it also demonstrates better performance in SC decoding as shown in Figure 3.9. The same $(1024, 512)$ codes were used in both a BP decoder and a SC decoder, where the iteration limit of BP decoding is set to 30 with early termination ($m = 3$, $t = 3$ and $l = 30$). For a SC decoder, our bit selection provides 0.7 dB coding gain compared to the conventional bit selection at FER of 10^{-4}. SC decoding provides a slightly better error-correcting performance than BP decoding.

3.3.4.4 FPGA Speedup

The in-order bit selection for a 256-bit polar code requires on the order of 50k Monte Carlo simulations per bit at 1 dB SNR, or 2M simulations per bit at 4 dB SNR, as more simulations are necessary at a high SNR due to the lower BER. As

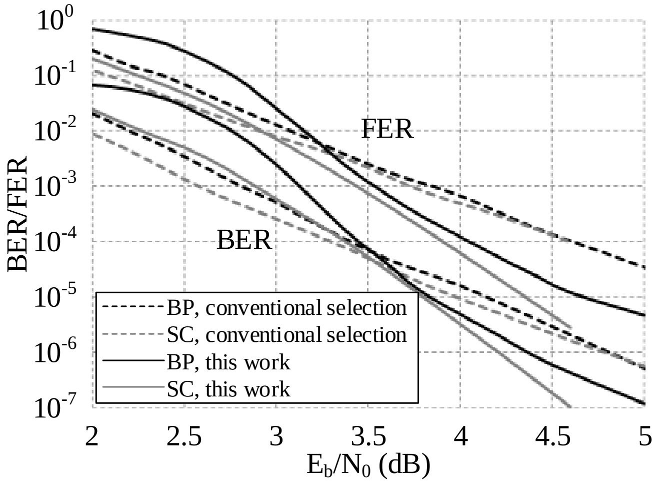


Figure 3.9: Performance of (1024,512) polar codes on BP and SC decoders.

discussed previously, to obtain a good bit selection, the bit selection needs to be done at a relatively high SNR instead of a low SNR. For a longer code, the simulation time increases further due to the lower BER and more bits in a longer code. The number of simulations per bit for a 1024-bit code is one order of magnitude higher than a 256-bit code.

In Table 3.2, we compare the time required for the bit selections using a compiled C code running on an Intel Core i7-4790K processor (quad core, 8M cache, 4.40 GHz, 32GB memory) without multi-threading or SIMD extension and using a Xilinx Virtex-6 SX475T FPGA (100MHz) to provide acceleration. The decoder is implemented in a stage-parallel. The speedup by FPGA is significant: at 4 dB SNR, the 256-bit code bit selection requires 1.8 days on a microprocessor but only 22 minutes with the FPGA – a 120 times speed up. The estimated C code simulation time for the 1024-bit code is about 1 year, making it impractical. However, with the FPGA, the bit selection can be done in 1 day – a 360 times speedup. The comparison demonstrates the need for acceleration in simulation-based bit selections. One can also use GPUs with massive parallel threads to speed up simulation.

Table 3.2: Time Required for Bit Selection in Software (C Simulation on a Microprocessor) and FPGA.

	256-bit code			1024-bit code		
SNR	N_{MC}*	μP*	FPGA	N_{MC}	μP	FPGA
1	50k	65 min	33 sec	500k	8.8 day	38 min
2	200k	4.3 hr	2.2 min	2M	35 day	2.5 hr
3	500k	10.8 hr	5.5 min	5M	3 mon	6.3 hr
4	2M	1.8 day	22 min	20M	1 yr	1 day
5	5M	4.5 day	55 min	50M	2.5 yr	2.6 day

*: number of Monte Carlo simulations per bit *: Microprocessor

3.4 Summary

In this chapter, we present a simulation-based bit selection method and acceleration to design polar codes for BP decoding. Starting with two hypothetical bit selection methods and an analysis of their weaknesses, we present an in-order bit selection method that bases the frozen set selection on an accurate evaluation of the bit error probability. The simulation-based algorithm accounts for practical nonidealities, including finite block length, fixed-point quantization, and practical channel models. The method is applicable for different code rates and code lengths. The results are demonstrated in three code designs, a $(256, 128)$ code, a $(512, 256)$ code, and a $(1024, 512)$ code, that outperform conventional code designs by 0.6 dB, 0.6 dB, and 0.9 dB, respectively, in BP decoding at an FER of 10^{-6}. The code construction also shows significant performance improvement on an SC decoder.

To improve the throughput of the decoding, early termination is used to reduce the average latency, which also lowers error rate at moderate SNR region. At low SNR where the magnitude of the messages is small, offset correction minimizes the min-sum approximation error and lowers the error rate by an order of magnitude. At high SNR, adaptive quantization provides more dynamic range with a fixed short word length in implementation, which lowers the error rate by orders of magnitude.

To speed up the simulation-based bit selection, we make use of a stage-parallel BP decoder design on FPGA. The inner loop of the bit selection algorithm is done on FPGA to cut the simulation time by orders of magnitude. To further speed up the bit selection, we implement an early termination scheme to shorten the decoding latency by up to 5 times. As a result, the bit selection for a 256-bit polar code takes only 22 minutes and a 1024-bit polar code takes 1 day, making it feasible for designing codes of practical block lengths.

CHAPTER IV

Error Patterns and Their Mitigation Methods

Polar codes are the first provably capacity-achieving error-correcting codes for any binary-input discrete memoryless channels (B-DMC) [1], and the error-correcting capability of polar code holds high promise. The two main decoding algorithms are SC [1] and BP [13]. SC exhibits a better error-correcting performance than BP [2], and list decoding [33], viewed as an enhanced SC, further improves the performance but with an increased complexity. BP on the other hand, provides a higher throughput and a reduced latency, but it sacrifices error-correcting performance.

BP is an iterative message passing algorithm operating on a factor graph. The same BP decoding algorithm has been widely used in decoding low-density parity-check (LDPC) codes. Despite the impressive performance seen in decoding LDPC codes, BP has shown a weakness known as the error floor phenomenon [24]. Error floors occur at moderate to high SNR levels, preventing the waterfall-like improvement in error rate with increasing SNR. In the error floor region, decoding errors are dominated by a small number of fixed patterns known as trapping sets [24].

In this work, we analyze the error patterns in the BP decoding of polar codes. The decoding errors are classified and the factors affecting decoding, including channel SNR, code design, decoding algorithm, and implementation, are analyzed. Based on the insights, we provide preliminary ideas of how the decoding errors can be mitigated.

4.1 Error Classification

To understand how decoding fails, we obtain hard decisions $\hat{u}$ at the end of each BP decoding iteration. Due to the lack of a definitive convergence check in polar decoding, we use hard decisions from consecutive iterations to decide whether decoding has converged. If hard decisions over consecutive iterations agree, we consider it has converged.

4.1.1 Unconverged Error

If hard decisions fail to agree within a maximum allowed iteration limit, and there is no defined pattern of error, we call it an unconverged error. Unconverged errors are most common at a low SNR level where the channel is noisy and the decoder is unable to resolve the errors.

Assume an all-zero codeword is transmitted using binary phase-shift keying (BPSK) modulation; and a bit u_i is decoded correctly if the soft decision of $\hat{u}_i > 0$ and incorrectly otherwise. A plot of the soft decisions of $\hat{u}$ is shown in Figure 4.1, illustrating an unconverged error for a (256, 128) code. The decision threshold is 0. The soft decisions keep flipping across the decision threshold, and incorrect soft decisions hover around the decision threshold. There is no obvious pattern in the decisions, and more iterations do not help to find correct convergence.

4.1.2 Converged Error

As SNR increases, unconverged errors start to disappear, and errors of systematic patterns start to emerge. The majority of systematic error patterns we found are attributed to converging to wrong codewords, or falling to local minima of BP decoding operating on loopy factor graphs. The factor graphs of polar codes contain loops, so a flooding BP decoder is not immune to local minima problems.

If hard decisions are stable and agree over consecutive iterations, but the decoded

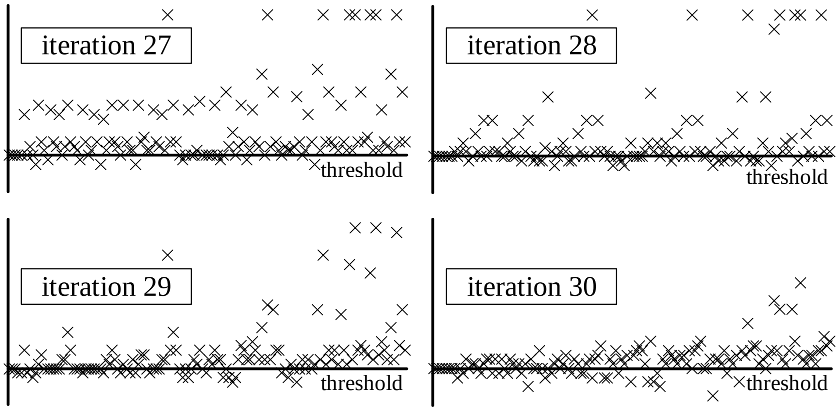


Figure 4.1: Soft decisions of an unconverged error in BP decoding of a (256, 128) code.

message is incorrect, i.e., $\hat{u} \neq u$, we call it a converged error. Converged errors are most common at moderate to high SNR, and it usually takes only a small number of iterations to reach a steady state, as illustrated in Figure 4.2 for a converged error in the decoding of a (256, 128) code. In this example, within two or three iterations, the soft decisions of a small number of bits are found to be trapped in wrong decisions, and they cannot be recovered using more iterations.

The particular case illustrated in Figure 4.2 represents a local minimum state in BP decoding. The few incorrect bits reinforce the wrong decisions among themselves through loops in the factor graph, making it impossible to make any progress towards convergence, which is similar to a trapping set found in the BP decoding of LDPC codes.

4.1.3 Oscillation Error

Loops in the factor graph allow the propagation of incorrect messages through BP decoding, causing oscillations. As incorrect messages travel around a loop, the decisions also go through a round of changes.

If hard decisions are unstable and change periodically over iterations, we call it an oscillation error. Although an oscillation error is also an unconverged error, an

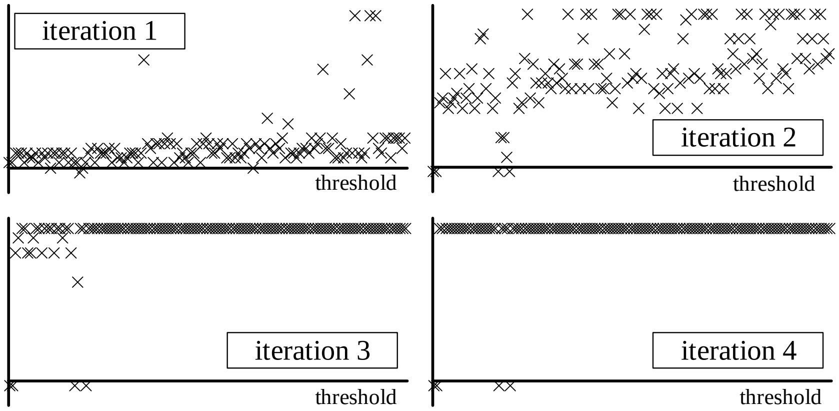


Figure 4.2: Soft decisions of a converged error in BP decoding of a (256, 128) code.

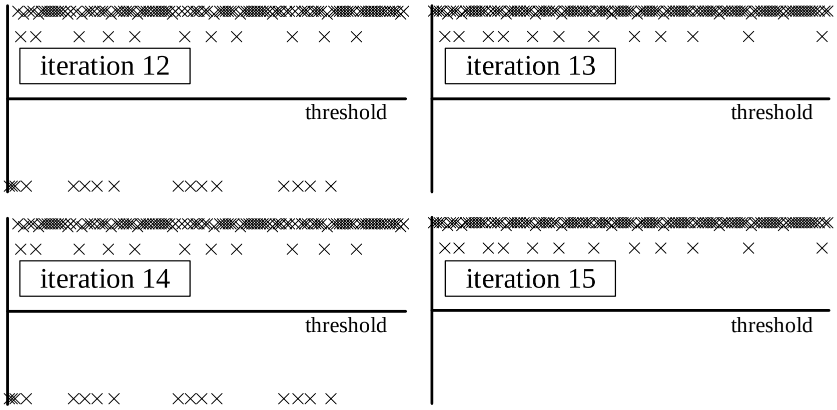


Figure 4.3: Soft decisions of an oscillation error in BP decoding of a (256, 128) code.

oscillation error features a pronounced pattern of periodic changes. An example of the oscillation error is shown in Figure 4.3. The illustrated error has an oscillation period of 2 iterations. A group of bits are incorrect in iteration 12; the incorrect bits all turn correct in iteration 13, but they turn incorrect again in iteration 14. If decoding is terminated in iteration 13, decoding would be done correctly. However, there is no way for the decoder to decide when to terminate in the absence of a definitive convergence detector in polar codes. Relying on checking hard decisions over consecutive iterations does not help terminate an oscillation error in the right iteration.

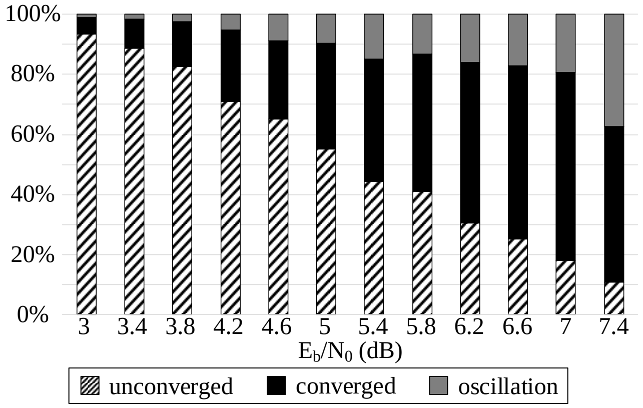


Figure 4.4: Error distribution for BP decoding of a (256, 128) code.

4.1.4 Error Distribution

In Figure 4.4, we show the statistical breakdown of errors at each SNR point for the BP decoding of a (256, 128) code. At a low SNR level, unconverged errors dominate; as SNR increases, converged errors and oscillation errors become dominant. The error breakdown demonstrates the importance of fixing the loopy behavior of BP decoding and of designing polar codes with a large minimum distance to improve the error-correcting performance.

4.2 Factors Affecting Decoding

To gain an insight into decoding errors, we adjust code and decoder design parameters and analyze the corresponding changes in error rate and error breakdown.

4.2.1 Code Design

Take the rate-0.5 (256, 128) code in Figure 4.4 as reference. We increase the rate of the 256-bit code from 0.5 to 0.53 and plot the error distribution and error rate of the rate-0.53 code in Figure 4.5. The axes and markers of the distribution in Figure 4.5 are identical to those in Figure 4.4, so they are omitted in Figure 4.5 for simplicity. All the later plots follow the same convention, unless they are explicitly

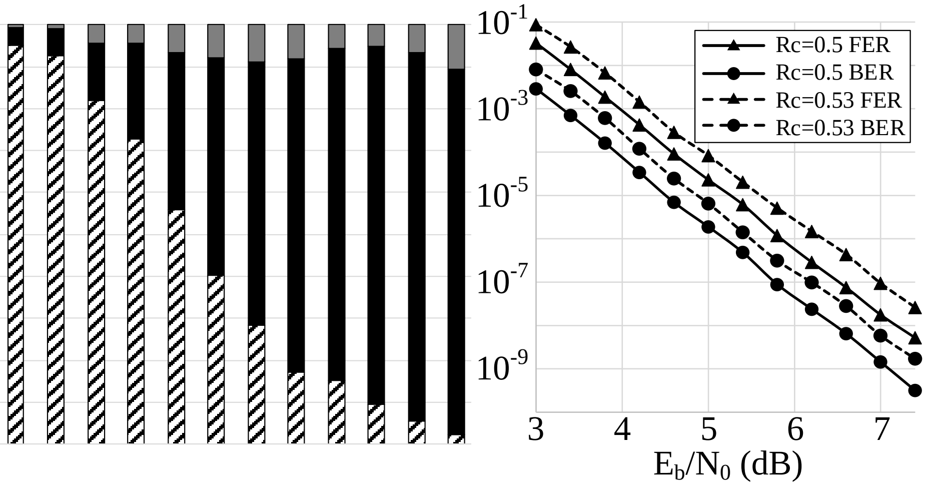


Figure 4.5: Error distribution and error rate for BP decoding of a (256, 136) code.

marked.

The error rate of a higher rate code is worse as expected. The number of converged errors at a high SNR level is noticeably higher. More converged errors can be explained by more information bits in a higher rate code resulting in more codewords, or a more crowded codeword space, making it more likely to converge to a wrong codeword.

As we increase the block length from 256 to 1024 while keeping the code rate of 0.5, the error distribution and error rate of the (1024, 512) code are shown in Figure 4.6. The error rate of the (1024, 512) code improves over the (256, 128) code, but it suffers from an error floor at FER below 10^{-7}. Compared to the (256, 128) code, the (1024, 512) code has fewer converged errors but more oscillation errors. The (1024, 512) code has a stronger polarization effect, so it is expected to outperform the (256,128) code. However, with two more stages in the factor graph, the factor graph of the (1024, 512) code contains more loop configurations than the (256, 128) code, resulting in more oscillation errors.

We also note that with more processing stages in a larger factor graph, numerical saturation occurs more easily. When reliable bits are saturated, they are less effective

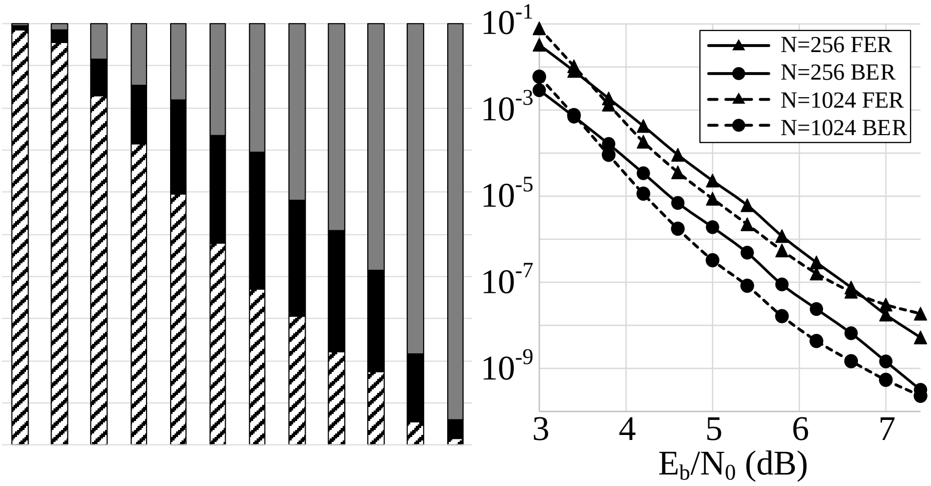


Figure 4.6: Error distribution and error rate for BP decoding of a (1024, 512) code.

in preventing incorrect bits from propagating. Allocating more bits to cover a larger numerical range is expected to alleviate the problem.

4.2.2 Decoder Implementation

The choice of fixed-point quantization affects the decoding performance [34]. In the above simulations, the Q7.-1 (6 bits covering the range of -64 to 62 with a resolution of 2) fixed-point quantization was used. Keeping the same 6-bit word length, the Q6.0 quantization covers the range of -32 to 31 with a resolution of 1, and the Q5.1 quantization covers the range of -16 to 15.5 with a resolution of 0.5. The quantizations used in Figure 4.7, Q5.1, Q6.0, Q7.-1 and Q8.-2, all share the same 6-bit wordlength, but they result in different error-correcting performance.

At a low SNR level, a quantization with a finer resolution improves numerical accuracy; and at a high SNR level, a quantization with a larger range prevents clipping and yields better performance. The comparison between the two codes in Figure 4.7 shows that a longer code requires a higher range to obtain the expected performance, and a larger range also alleviates the error floor problem.

Min-sum approximation is often applied to simplifying the log-likelihood ratio cal-

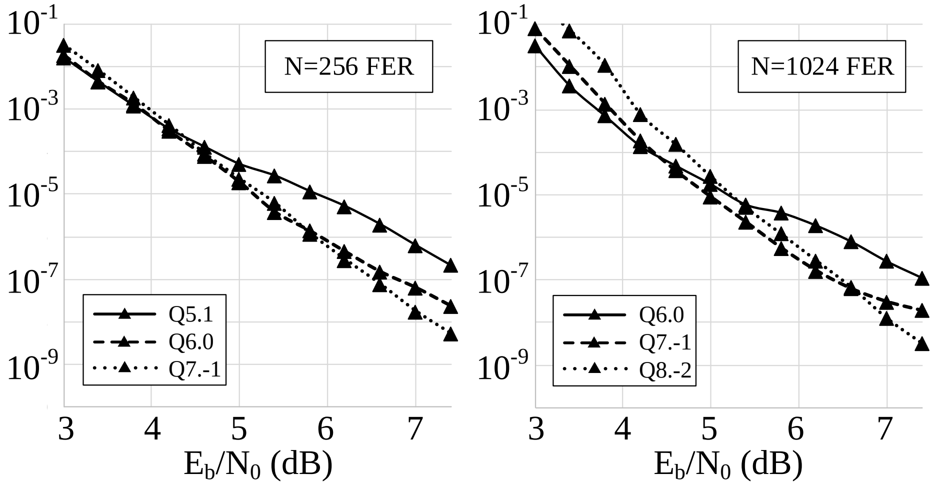


Figure 4.7: Error rate for BP decoding of a (256, 128) code and a (1024, 512) code that are implemented in different quantization schemes.

culations. The associated min-sum approximation error can be compensated by offset correction. Offset correction is especially effective at a low SNR level, as illustrated in Figure 4.8. The number of unconverged errors is reduced, as offset correction reduces approximation errors and improves the decoding performance.

4.3 Error Detection

As discussed above, a BP polar decoder is unaware of whether decoding has converged. An iteration-by-iteration hard decision check detects unconverged errors, but it fails to detect converged errors, which account for 20% to 90% of the errors. An iteration-by-iteration hard decision check detects oscillation errors, but it fails to find the right iteration to terminate decoding and stop oscillations. Therefore, we add a low-cost error detection scheme to catch the majority of the undetected errors.

4.3.1 CRC Concatenation

The error detection scheme is based on concatenating polar code with cyclic redundancy check (CRC) that consumes only a small number of parity bits but provides

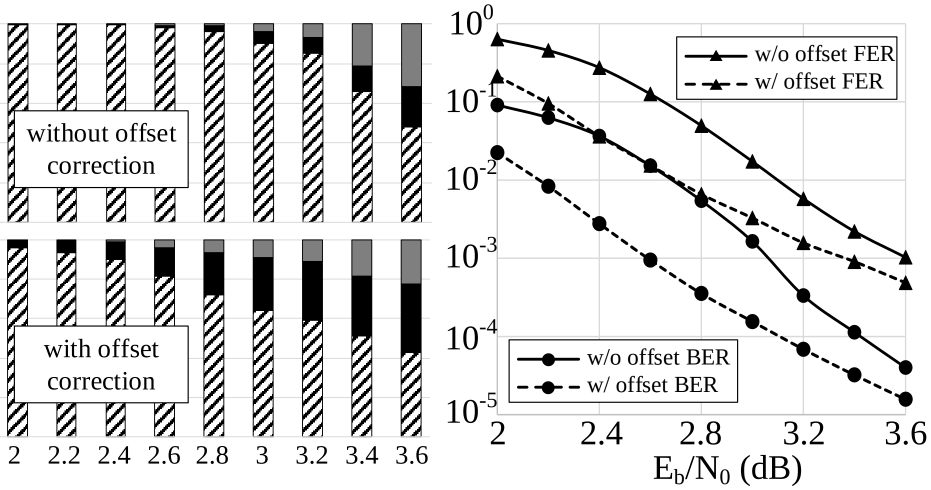


Figure 4.8: Error distribution and error rate for BP decoding of a (1024, 512) code at low SNR with and without offset correction.

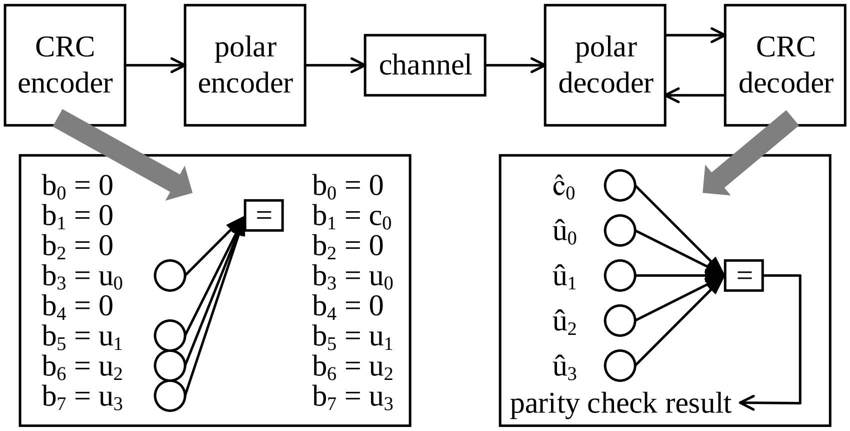


Figure 4.9: Illustration of concatenation of polar code with CRC.

a good detection capability. A CRC codec is added outside the polar codec, illustrated in Figure 4.9. The CRC encoder generates parity bits for an input message. The message bits along with the parity bits are remapped to the information bits of the polar code. On the decoder side, the CRC decoder checks if the hard decisions obtained by the polar decoder is a valid CRC codeword at the end of each decoding iteration.

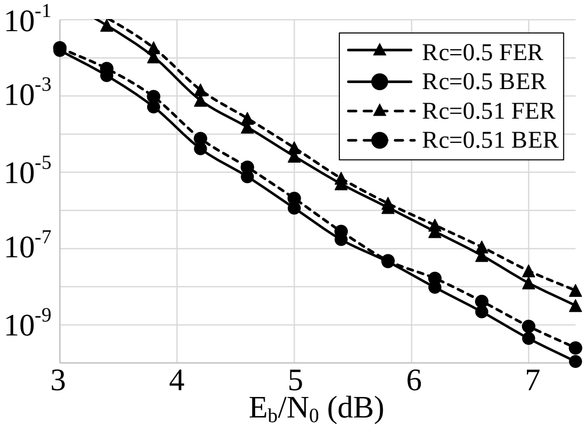


Figure 4.10: Error rate for BP decoding of a (1024, 512) code before and after CRC concatenation.

4.3.2 Performance Loss

A CRC-n code generates n parity bits, which covers up to $2^n - 1 - n$ message bits. We employed CRC-8 in the rate-0.5 256-bit polar code and CRC-10 in the rate-0.5 1024-bit polar code. Our simulation shows that more than 99% of the previously undetected errors are detected by CRC. Note that CRC concatenation increases the code rate. If the overall code rate is kept the same, CRC concatenation results in a slight performance loss.

In the 256-bit code and the 1024-bit code, CRC concatenation increases the code rates from 0.5 to 0.53 and 0.5 to 0.51, respectively. The performances are compared in Figure 4.5 and Figure 4.10. The coding gain of the 256-bit code is reduced by approximately 0.3 to 0.5 dB, but the loss is much smaller in the 1024-bit code due to the negligible number of parity bits relative to the block length.

4.3.3 CRC-Based Termination

We make use of CRC to reliably determine when to terminate decoding, i.e., if CRC passes, the iterative decoding is terminated. The effect of CRC-based termination in the BP decoding of two polar codes is shown in Figure 4.11, where the white space above the bars represents the percentage of errors being resolved with

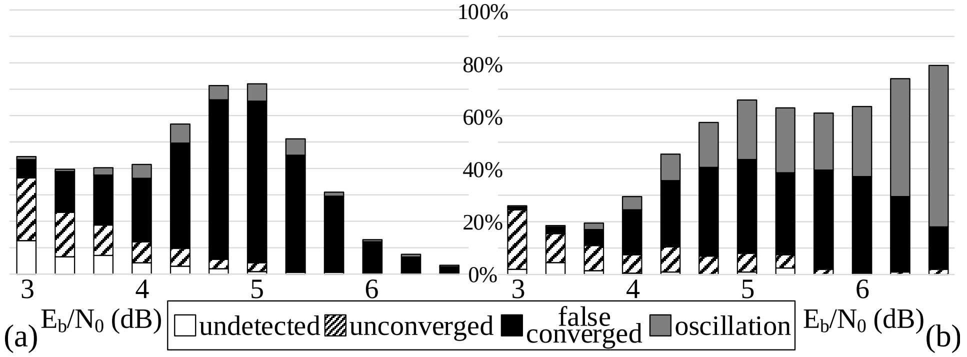


Figure 4.11: Error distribution for BP decoding of (a) a (256, 128) and (b) a (1024, 512) polar code with CRC-based termination.

the proper termination based on CRC. We observe that BP decoding is able to produce error-free messages in some iterations even if the decoding itself is not stable. The CRC-based termination helps to lock in the correct codeword before decoding diverges.

CRC-based termination helps resolve most of the unconverged errors and a portion of oscillation errors. The remaining errors are dominated by converged errors and oscillation errors. One approach to fix the remaining errors is via post-processing by perturbation. Prior work in LDPC post-processing points out that perturbation is especially beneficial when a BP decoder is trapped in a local minimum [25]. From a cost standpoint, post-processing can be implemented as part of BP decoding with biased messages.

4.4 Error Mitigation

False converged errors and oscillation errors determine the error-correcting performance of BP decoding at moderate to high SNR level. A false converged error represents a steady state that "traps" the decoder. To escape, the steady state needs to be perturbed. Compared to a false converged error, an oscillation error provides direct clues as to which bits are stable and unstable. The insight can be exploited to stop oscillations by enhancing the stable bits and perturbing the unstable bits. Regular BP decoding can be used to clean up the errors caused by perturbation. Excessive perturbation needs to be unrolled in solving unconverged errors. These intuitive ideas form the basis of our post-processing methods.

In BP decoding, the frozen set information is propagated from left to right of the factor graph. The frozen set information is carried by the so-called R messages. If a bit u_i is frozen, we set the R message of u_i to the maximum positive value, which effectively biases $\hat{u}_i$ to 0 to overtake the effect of the extrinsic messages. If u_i is free, the R message of u_i is set to zero, so that $\hat{u}_i$ is unbiased and entirely determined by the extrinsic messages. In summary, if $i \in \mathcal{A}^c, R_i = M_{max}$; if $i \in \mathcal{A}, R_i = 0$.

In post-processing, we tune the R messages of the information bits to introduce perturbation. This could also be understood as biasing, or partially freezing the information bits towards one direction or another: a positive R message biases an information bit towards 0; and vice versa.

Since perturbation injects noise to the system, to quantify the effect of perturbation in post-processing, we define a cost function based on the decoded soft decisions.

$$C(\mathbf{x}) \triangleq \sqrt{\frac{\sum_{i \in \mathcal{A}} (M_{max} - \mathbf{x}_i)^2}{|\mathcal{A}|}},$$

where $\mathbf{x}$ is the vector of soft decisions of the decoded bits and $\mathbf{x}_i \in [-M_{max}, M_{max}]$, and $\mathcal{A}$ is the set of information bits. Assume an all-zero codeword and a quantized BP decoder, and M_{max} is the maximum magnitude of a soft decision. The cost function is essentially a measure of the normalized average distance between the decoded soft decisions $\mathbf{x}$ and the transmitted codeword.

4.4.1 Post-Process Converged Error

To fix false converged errors, we will apply a small perturbation to destabilize the converged state. A balancing act is needed as perturbation increases the noise, and will likely cause errors to be made. Therefore the perturbation needs to be kept low and below a threshold level, and perturbation should be applied discriminatingly. We use the soft decision of a bit as an indication of the reliability of the bit's hard decision: if the magnitude of the soft decision is high, the hard decision is most likely correct, and vice versa. Therefore, we enhance the reliable bits and perturb the unreliable bits at the same time. We use regular BP decoding to clean up the errors introduced by perturbation. The post-processing method is described in Algorithm 1.

Algorithm 1: Post-processing false converged errors

```
for $iter\_count = 1$ to $iter\_limit$ do
    if CRC fails && hard decisions are consistent then
        for $i \in \mathcal{A}$ do
            if $|\mathbf{x}_i| > M_{threshold}$ then
                if $R_i == 0$ then
                    $R_i = sign(\mathbf{x}_i) \times M_0$
                else if $|R_i| < M_{limit}$ then
                    $R_i = sign(\mathbf{x}_i) \times |R_i| \times c$
                else
                    $R_i = -sign(R_i) \times M_0$
            else
                if $R_i == 0$ then
                    $R_i = rand(\pm 1) \times M_0$
                else if $|R_i| < M_{limit}$ then
                    $R_i = -sign(\mathbf{x}_i) \times |R_i| \times c$
                else
                    $R_i = -sign(R_i) \times M_0$
```

The post-processing method starts by recognizing whether a bit is reliable. If the soft decision $\mathbf{x}_i$ reaches a set threshold $M_{threshold}$, indicating the decision being reliable, the bit is enhanced by setting R_i to M_0 in the same direction as the hard decision, i.e., the sign of $\mathbf{x}_i$. Setting R_i in the same direction as $\mathbf{x}_i$ will amplify its influence on neighboring bits in subsequent iterations. If a bit is unreliable because $\mathbf{x}_i$ is below $M_{threshold}$, the bit is perturbed by randomly setting R_i to either M_0 or $-M_0$ to bias the bit towards 0 or 1, respectively. (The notation $rand(\pm 1)$ in the algorithm

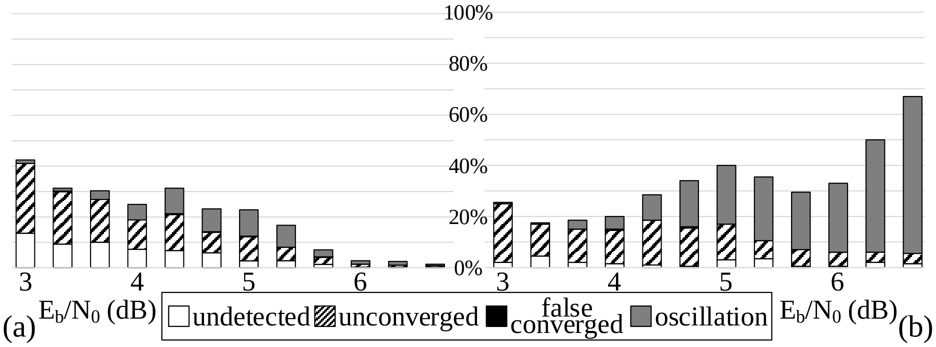


Figure 4.12: Error distribution after post-processing false converged errors in BP decoding of (a) a (256, 128) CRC-concatenated polar code and (b) a (1024, 512) CRC-concatenated polar code.

refers to randomly picking 1 or -1.) The goal of the post-processing is to push the state out of false convergence. After the perturbation, regular BP decoding is applied to clean up the errors introduced by the perturbation in an attempt to move towards convergence.

The post-processing is considered successful if either CRC is satisfied, indicating correct convergence (except for a very few number of undetected errors), or the hard decisions no longer remain consistent from one iteration to the next, indicating the decoding has escaped the false convergence. If one attempt of post-processing is not successful, a second attempt using stronger enhancement and perturbation is applied ($c > 1$ in the algorithm). The post-processing attempt continues until a bias threshold M_{limit} is reached to prevent excessive noise injection from perturbation. In case of M_{limit} being reached, the post-processing can be restarted with a small bias M_0 in the opposite direction.

The error breakdown after applying post-processing to false converged errors is shown in Figure 4.12. The post-processing resolves the majority of the false converged errors by BP decoding following perturbation, and part of them are turned to either unconverged errors or oscillation errors. False converged errors become negligible following the post-processing.

The iteration-by-iteration plots of the cost function of BP decoding resulting in

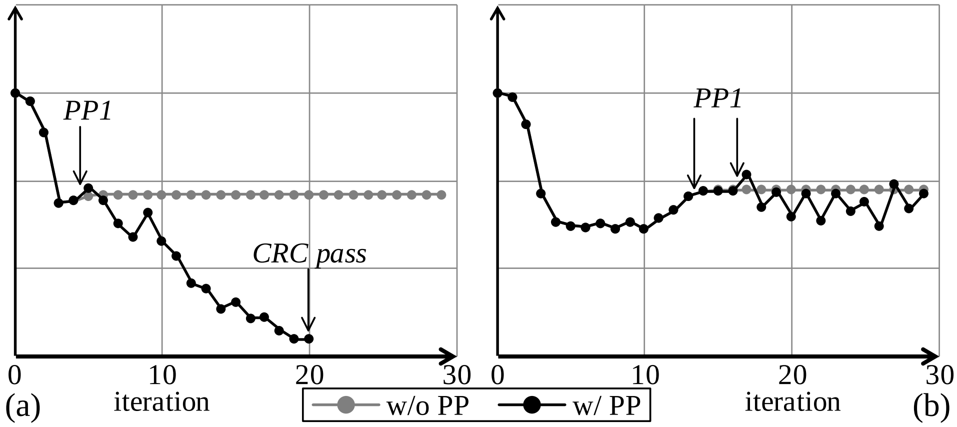


Figure 4.13: Cost functions of BP decoding with and without applying post-processing Algorithm 1 (PP1): (a) an example of a false converged error resolved by Algorithm 1, and (b) an example of a false converged error that is not resolved by Algorithm 1.

false converged errors are shown in Figure 4.13. In the example illustrated in Figure 4.13(a), the cost initially descends in BP decoding, and then false convergence is detected and post-processed at iteration 5. The cost first rises due to perturbation, allowing the decoder to escape false convergence. Regular BP decoding follows post-processing. It cleans up the errors due to perturbation and converges to the correct codeword. In the example illustrated in Figure 4.13(b), false convergence is detected and post-processed at iteration 14; and again detected and post-processed at iteration 17. Regular BP decoding follows each post-processing attempt, but the error turns into an unconverged error that cannot be solved by Algorithm 1 alone.

4.4.2 Post-Process Oscillation Error

In post-processing false converged errors, we used the magnitude of soft decisions to guide whether to apply enhancement or perturbation. Oscillation errors, on the other hand, provide direct clues of which bits are reliable and which ones are not. Unstable bits change their hard decisions periodically and are considered unreliable, and stable ones are consistent and considered reliable. To stop oscillations, stable bits are enhanced and unstable bits are perturbed. The post-processing method is described in Algorithm 2.

Algorithm 2: Post-processing oscillation errors

```
for $iter\_count$ = $1$ to $iter\_limit$ do
    if $oscillation\ is\ detected\ (of\ period\ T)$ then
        for $i \in \mathcal{A}$ do
            if $sign(\mathbf{x}_i)\ is\ consistent\ in\ T\ iterations$ then
                if $R_i == 0$ then
                    $R_i = sign(\mathbf{x}_i) \times M_0$
                else if $|R_i| < M_{limit}$ then
                    $R_i = sign(\mathbf{x}_i) \times |R_i| \times c$
            else
                if $R_i == 0$ then
                    $R_i = rand(\pm 1) \times M_0$
                else if $|R_i| < M_{limit}$ then
                    $R_i = -R_i \times c$
                else
                    $R_i = -sign(R_i) \times M_0$
```

An oscillation error is detected by checking the consistency of hard decisions over consecutive iterations. Enhancement and perturbation are applied to the stable and unstable bits respectively using the similar approaches in Algorithm 1, starting by biasing the R messages by a small amount M_0 and letting regular BP decoding iterations clean up the errors. If post-processing is unsuccessful after one attempt, another attempt is used with stronger enhancement and perturbation until a biasing threshold of M_{limit} is reached.

The error breakdown after applying post-processing to both false converged errors and oscillation errors is shown in Figure 4.14. The vast majority of false converged errors and oscillation errors are resolved, and the remaining errors are almost all unconverged errors. Note that an error can evolve from one type to another in the post-processing procedure, so Algorithm 1 and 2 need to be employed jointly.

The plots of the cost function of BP decoding resulting in oscillation errors are shown in Figure 4.15. Figure 4.15(a) illustrates an oscillation error. The error is

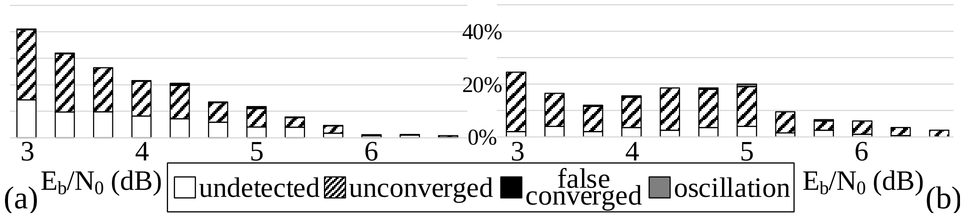


Figure 4.14: Error distribution after post-processing oscillation errors and false converged errors in BP decoding of (a) a (256, 128) CRC-concatenated polar code and (b) a (1024, 512) CRC-concatenated polar code.

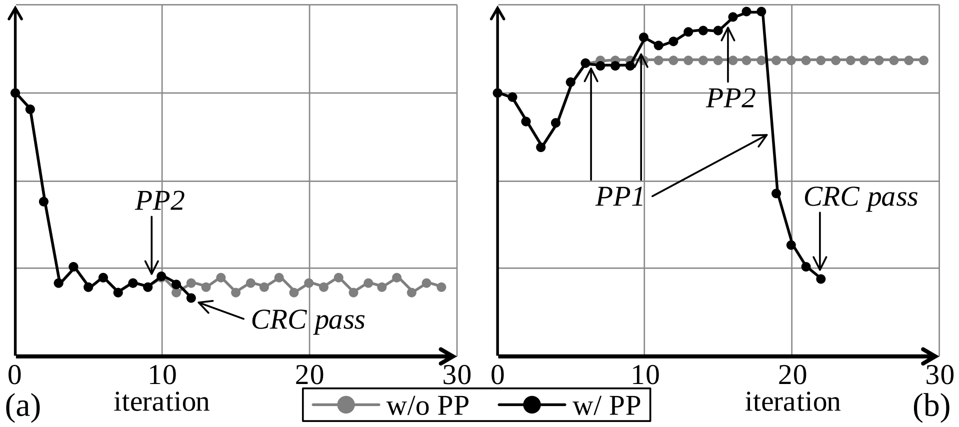


Figure 4.15: Cost functions of BP decoding with and without applying post-processing Algorithm 1 and 2 (PP1 and PP2): (a) an example of an oscillation error resolved by Algorithm 2, and (b) an example of a false converged error evolving to an oscillation error that is resolved by Algorithm 1 and 2.

detected and post-processed at iteration 9 using Algorithm 2. After a few iterations of regular BP decoding, the error is resolved. Figure 4.15(b) illustrates a false converged error. False convergence is detected at iteration 7 and post-processed by Algorithm 1, and again at iteration 10. Following the second post-processing attempt, the error evolves to an oscillation error at iteration 16, and it is post-processed by Algorithm 2. Then it turns to another false converged error at iteration 19. Finally, the error is successfully resolved following a post-processing by Algorithm 1.

4.4.3 Post-Process Unconverged Error

With the vast majority of the false converged errors and oscillation errors removed, the only dominant errors left are unconverged errors. Many of the unconverged errors are in fact due to the perturbation applied to unreliable bits during post-processing. To resolve unconverged errors, the perturbation needs to be “unrolled” to enable convergence.

We can select stable bits using some criteria, e.g., bits that remain consistent with soft decisions of high magnitude, and unstable bits using the opposite characteristic. To resolve unconverged errors, we enhance the stable bits, but weaken the perturbation on the unstable bits to facilitate convergence. The post-processing method is formulated in Algorithm 3.

Algorithm 3: Post-processing unconverged errors

```
for $iter\_count = 1$ to $iter\_limit$ do
    if $iter\_count > iter\_threshold$ && no oscillation is detected then
        for $i \in \mathcal{A}$ do
            if $sign(\mathbf{x}_i)$ is consistent over iterations && $|\mathbf{x}_i| > M_{threshold}$ then
                if $R_i == 0$ then
                    $R_i = sign(\mathbf{x}_i) \times M_0$
                else if $|R_i| < M_{limit}$ then
                    $R_i = sign(\mathbf{x}_i) \times |R_i| \times c$
            else if $sign(\mathbf{x}_i)$ is inconsistent over iterations then
                $R_i = R_i/c$
                if $|R_i| < M_{limit}$ then
                    $R_i = -R_i$
```

An error is marked as an unconverged error if BP decoding fails to converge after a sufficient number of BP iterations, and no oscillation is detected. In formulating the post-processing method, we combine the strategies in Algorithm 1 and 2 to determine stable and unstable bits. The stable bits are enhanced using the same approach as in Algorithm 2. For the unstable bits, the perturbation is weakened by gradually

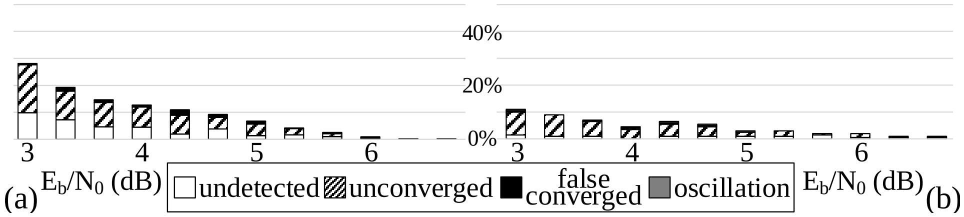


Figure 4.16: Error distribution after post-processing unconverged errors, oscillation errors and false converged errors in BP decoding of (a) a (256, 128) CRC-concatenated polar code and (b) a (1024, 512) CRC-concatenated polar code.

reducing the bias in R messages, essentially undoing the perturbation to push towards convergence. If a bit meets neither stable nor unstable conditions, its R message will remain untouched. Most of the unconverged errors are resolved after applying Algorithm 3 as shown in Figure 4.16.

The plots of the cost function of BP decoding resulting in unconverged errors are shown in Figure 4.17. Figure 4.17(a) illustrates an unconverged error, which is post-processed by Algorithm 3 at iteration 33. The error evolves to an oscillation error at iteration 35, and is post-processed by Algorithm 2. Finally the error turns back to an unconverged error. After two post-processing attempts at iteration 37 and 39 by Algorithm 3, decoding converges. Figure 4.17(b) illustrates an oscillation error, which is post-processed by Algorithm 2 at iteration 31. The error evolves to an unconverged error and resolved after post-processing using Algorithm 2 at iteration 33.

4.5 Results

The post-processing methods presented above can be efficiently implemented in BP decoding. Error detection is done by CRC and monitoring iteration-by-iteration hard decisions. If the decoding of an input frame does not converge within an iteration limit, or if the decoding converges but fails CRC, an error is detected. Post-processing will only be applied to the detected errors.

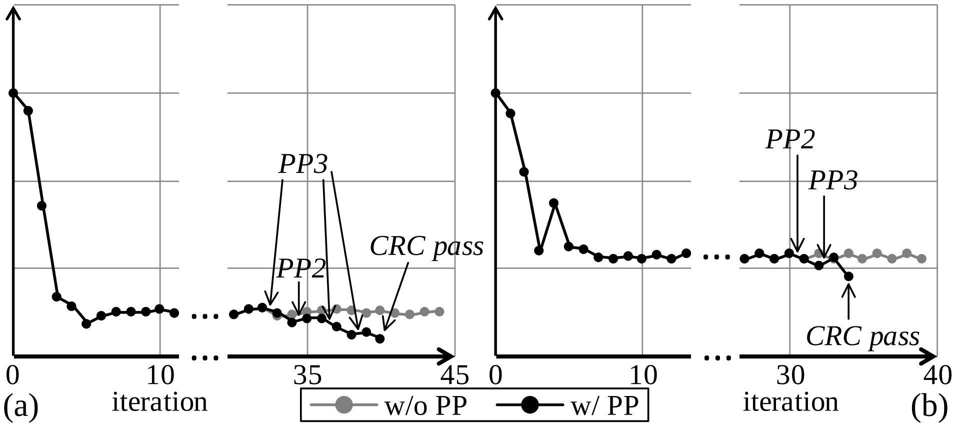


Figure 4.17: Cost functions of BP decoding with and without applying post-processing Algorithm 2 and 3 (PP2 and PP3): (a) an example of an unconverged error resolved by Algorithm 2 and 3, and (b) an example of an oscillation error resolved by Algorithm 2 and 3.

The error-correcting results of post-processing are demonstrated in two code examples, a (256, 128) code and a (1024, 512) code, as shown in Figure 4.18. Despite the decrease in coding gain due to CRC concatenation, post-processing easily recoup the loss, and improve the coding gain of the (256, 128) code by 0.8 dB at FER of 10^{-8} and the coding gain of the (1024, 512) code by more than 1 dB at FER of 10^{-8}.

With post-processing, BP decoding overtakes SC decoding in performance as shown in Figure 4.19. The decoders are implemented in the same fixed-point quantization. The BP performance was obtained by FPGA emulation, while the SC performance was obtained by simulation. Due to the long latency in SC decoding and the slower software simulation, the SC error rate curves only extend to 10^{-6} in Figure 4.19. In decoding the (256, 128) code and the (1024, 512) code, BP decoding with post-processing outperforms SC decoding by 0.4 dB and 0.5 dB, respectively, at moderate SNR.

The error-correcting results shown above are based on a low-overhead implementation, where the numerical value of the same parameter in different types of the errors are set to be the same, i.e., M_0 in the three algorithms have the same value, etc. The values can be finer tuned for each error type for a better correction result

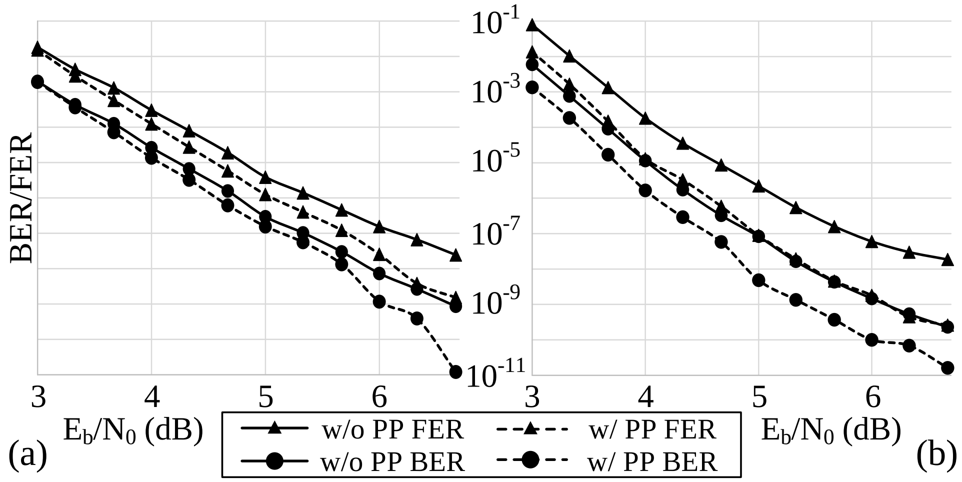


Figure 4.18: Error rates of BP decoding with and without post-processing for (a) a (256, 128) polar code, and (b) a (1024, 512) polar code using 6-bit quantized decoders.

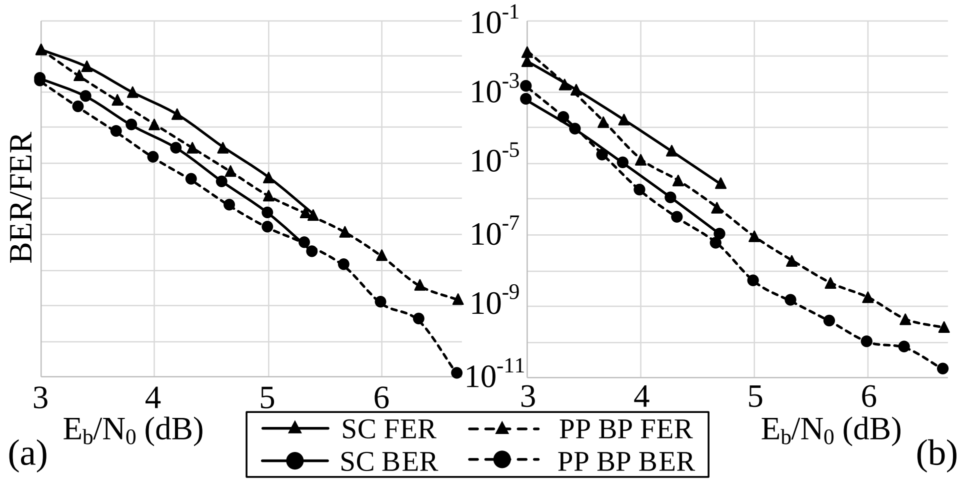


Figure 4.19: Error rates of SC decoding and BP decoding with post-processing for (a) a (256, 128) polar code, and (b) a (1024, 512) polar code using 6-bit quantized decoders.

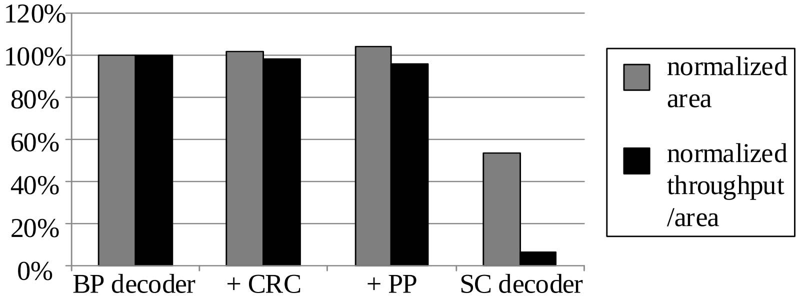


Figure 4.20: Normalized silicon area and throughput/area ratio of different designs based on chip synthesis in a 45nm CMOS technology.

with more hardware overhead. Although the choice of the parameter values depends on quantization, the same choice of the values are valid for different codes as long as their fix-point representation yields a similar relative dynamic range of the LLRs, e.g., we use the same set of relative values for a Q5.1 (256, 128) code and a Q6.0 (1024, 512) code.

To evaluate the cost of implementing post-processing in hardware, we performed chip synthesis in a 45nm CMOS technology, the results of which are presented in Figure 4.20. A stage-parallel BP decoder [34] is used as the baseline. Adding a CRC encoder and decoder adds a 1.8% area overhead, and adding post-processing costs an additional 2.5% area overhead.

Since post-processing is conditionally invoked, it does not affect the average throughput and latency of BP decoding. A classic SC decoder uses half of the area as a stage-parallel BP decoder, but its throughput and latency are significantly worse than the BP decoder. The figure of merit, in terms of throughput over area, of the BP decoder with post-processing is an order of magnitude better than an SC decoder as shown in Figure 4.20. The better error-correcting performance, the higher throughput and lower latency are the key advantages of BP decoding with post-processing that make it more competitive than SC decoding.

4.6 Summary

In this chapter, we classify BP decoding errors into three categories: unconverged errors, converged errors, and oscillation errors. We analyze the important factors affecting decoding, including code design and decoder implementation. While unconverged errors and oscillation errors are detectable by checking the hard decisions in each iteration for consistency, converged errors are undetected. Concatenation of polar codes with CRC enables the detection of undetected errors and the proper termination of BP decoding. CRC-based termination helps to remove a large portion of unconverged errors and oscillation errors.

We present post-processing methods targeting converged errors, oscillation errors, and unconverged errors that dominate the error-correcting performance of BP decoding of polar codes. Post-processing is designed based on BP using modified frozen set information. The modification is implemented by biasing the R messages based on BP message passing results. For false converged errors, enhancement is applied to the information bits of high reliability, and perturbation is applied to those of low reliability by randomizing biasing of R messages to escape false convergence. For oscillation errors, enhancement is applied to the stable bits to further strengthen these bits, and perturbation is applied to the unstable bits to stop oscillation. For unconverged errors, enhancement is applied to the stable bits, and perturbation on the unstable bits is unrolled to encourage convergence. In all three cases, BP decoding is used to clean up the errors introduced by perturbation.

Results show that post-processing of BP decoding improves the error rates by an order of magnitude or more at moderate to high SNR level, demonstrating better error-correcting performance than SC decoding. Post-processing can be efficiently implemented on a BP decoder with negligible hardware overhead, and it does not affect the average throughput and latency of BP decoding, thereby making BP decoding more competitive than SC decoding for practical high-performance applications.

CHAPTER V

Flexible Architecture and Configurable Hardware

5.1 Family of Architectures

The architecture of a single-column stage-parallel BP polar decoder has been discussed in section 2.2, and in this section, a wider design space of BP architecture will be explored. Due to the regularity of polar codes, the computation logic, i.e., the columns of PEs, can be folded or unfolded to create variations of the architecture. Within the design space, one can trade among area, power, latency and throughput to find a better fit of the application specifications. We will assume the same three-stage pipelined datapath as done in the FPGA model for all architectures for comparison.

5.1.1 Fully Parallel

Direct mapping the complete factor graph of a $N = 2^n$-bit polar decoder into hardware results in a fully parallel design shown in Figure 5.1(a). The design is composed of n PE columns each consisting of $N/2$ PEs, performing R-prop in one clock cycle by reading all the L messages and buffering all the R results in the intermediate storage, followed by performing L-prop in a similar manner. The mux inputs are marked based on their selection condition. The intermediate storage has to be implemented in registers so that extremely high bandwidth can be supported.

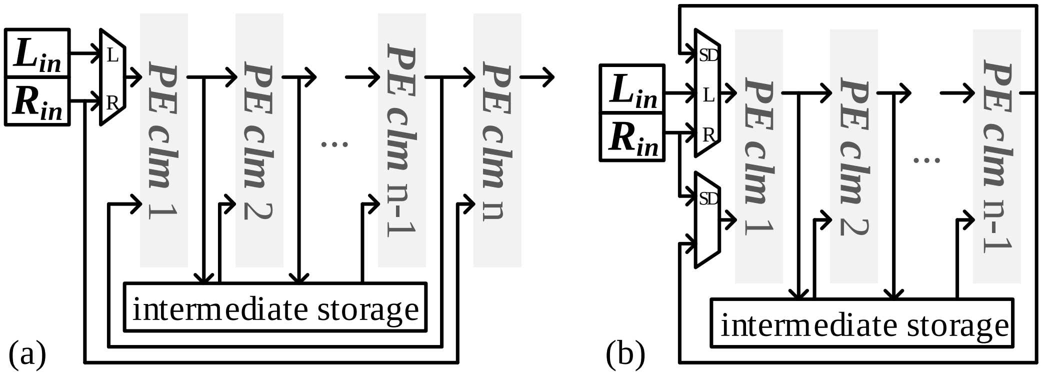


Figure 5.1: (a) Fully parallel design (b) with the last column saved.

We notice that column n is redundant in R-prop and is optional in L-prop unless in the last iteration. Alternatively, the last column can be merged with the first column, as shown in Figure 5.1(b), saving some area with the cost of 1 clock cycle to calculate the soft decisions, when the muxes select the inputs marked as SD.

The fully parallel design provides the fastest data rate and shortest latency, with a high area and power consumption. Each BP iteration takes 2 clock cycles, but the clock period is proportional to $\log N$.

5.1.2 Stage Parallel

Given that all the stages in R-prop and L-prop execute identical operations, respectively, it is possible to use only a small number of PE columns to serialize the stage operations, saving the majority of combinational logic at the cost of throughput. We show the single-column design in Figure 5.2(a) and double-column designs in (b) and (c). The mux inputs marked as L/R are selected by the first stage of either L-prop or R-prop.

The single-column design shown in Figure 5.2(a) follows the same connection as the FPGA model shown in Figure 2.6, where the design contains only one PE column, consisting of $N/2$ PEs, and processes one stage of BP decoding at a time. Each iteration takes $2 \log N$ clock cycles. Ideally the critical path lies in the PE logic,

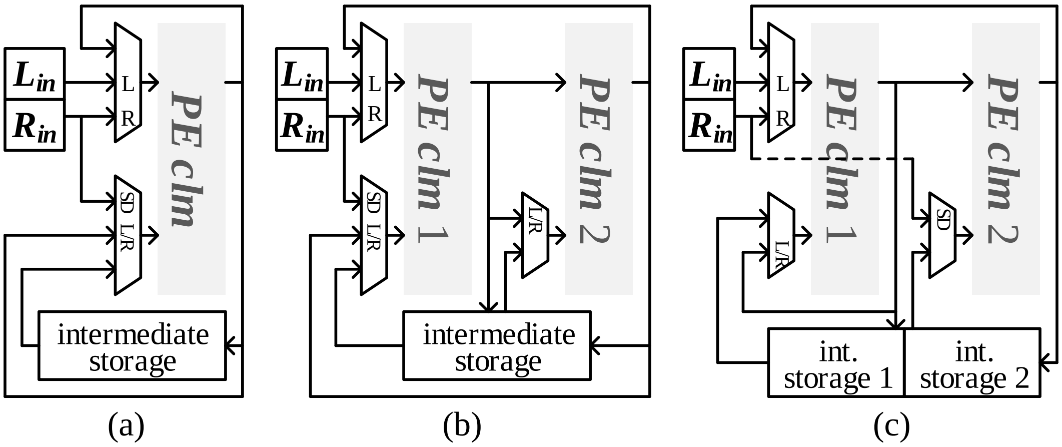


Figure 5.2: (a) Single-column design and double-column design with (b) n being odd and (c) n being even.

but the memory access latency of intermediate storage could surpass the PE logic latency, resulting in a longer clock period.

The double-column design contains two PE columns and processes two stages at a time. Since both PE columns access the intermediate storage, the memory bandwidth has to be doubled. The cycle count per iteration is reduced by half to $\log N$, but the clock period is expected to be extended due to the more complex two-stage PE logic.

Different code block lengths lead to slightly different designs. The design shown in Figure 5.2(b) corresponds to n being an odd integer, and (c) for n being even. We draw the time table of these two cases in Figure 5.3, and highlight the SD stage in dash and the L/R mux selections in arrows. The memory implementation in these two cases also varies, and we list the memory access patterns in Table 5.1. When n is odd, the two columns access the same pairs of entries in R-prop and L-prop, except that the order of each pair is flipped. The memory can be implemented with a word length of $2NQ$, with the ability of swapping half words, and with halved depth of $(n-1)/2$. When n is even, column 1 only accesses odd indexed entries and column 2 only accesses even indexed entries. The memory can be constructed by two halves, each with a word length of NQ, and with a depth of $n/2$ in the odd half and of $(n-2)/2$ in the even half.

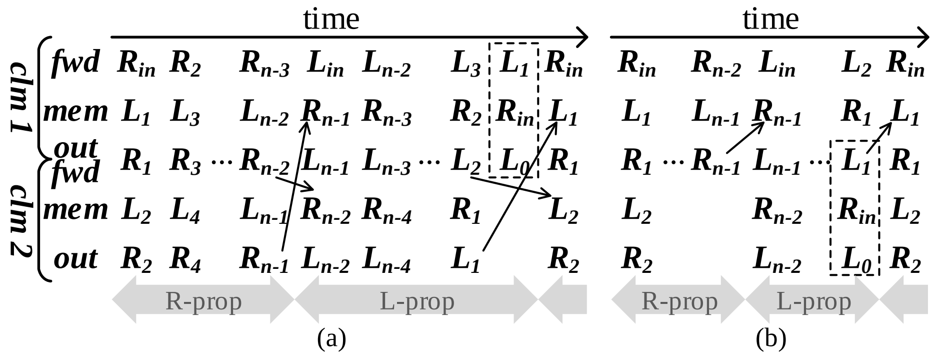

time
time
clm 1
clm 2
fwd R_{in} R_2 R_{n-3} L_{in} L_{n-2} L_3 L_1 R_{in}
mem L_1 L_3 L_{n-2} R_{n-1} R_{n-3} R_2 R_{in} L_1
out R_1 R_3 ··· R_{n-2} L_{n-1} L_{n-3} ··· L_2 L_0 R_1
fwd
mem L_2 L_4 L_{n-1} R_{n-2} R_{n-4} R_1 L_2
out R_2 R_4 R_{n-1} L_{n-2} L_{n-4} L_1 R_2
R_{in} R_{n-2} L_{in} L_2 R_{in}
L_1 L_{n-1} R_{n-1} R_1 L_1
R_1 ··· R_{n-1} L_{n-1} ··· L_1 R_1
L_2 R_{n-2} R_{in} L_2
R_2 L_{n-2} L_0 R_2
R-prop
L-prop
R-prop
L-prop
(a)
(b)

Figure 5.3: Time table of double-column design with (a) n being odd and (b) n being even.

Table 5.1: Memory Access Patterns of Double-Column Designs

		R-prop					L-prop				
odd n	columm 1	1	3	...	n-4	n-2	n-1	n-3	...	4	2
	columm 2	2	4	...	n-3	n-1	n-2	n-4	...	3	1
even n	columm 1	1	3	...	n-3	n-1	n-1	n-3	...	3	1
	columm 2	2	4	...	n-2		n-2	n-4	...	2	

5.1.3 Sub-stage Parallel

The structure of polar codes tells us that a short code is a subsection of longer codes. The stage-wise identical structure of BP decoding in addition to the regular structure of polar codes allows a BP decoder to process longer codes through sub-stage parallel architectures.

In the architectures introduced above, a stage of N-bit code is mapped to a column of $N/2$ PEs, and in Figure 5.4 we show a PE column being repeated twice to construct a $2N$-bit stage or four times for a $4N$-bit stage, with the routing of the PE column preserved. In the $2N$-bit case (shown in (b) and (d)), the connections in second and third quarter on the left hand side are swapped, with each quarter containing $N/2$ connections. We show the corresponding half-column architecture in Figure 5.5(a). In the $4N$-bit case (shown in (e)), the swapping is still based on groups of $N/2$ connections, or 1/8 with respect to the block length $4N$, in the same manner as a

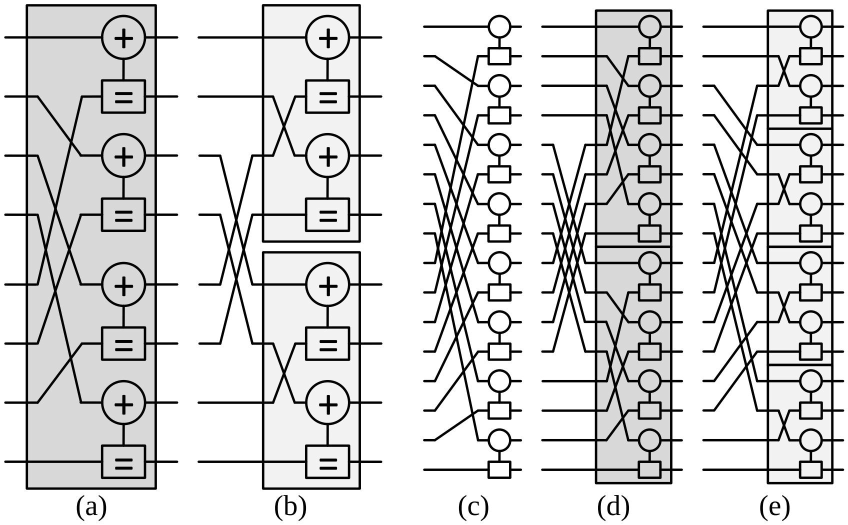


Figure 5.4: (a) An 8-bit column (b) composed of two copies of a 4-bit column. (c) A 16-bit column (d) composed of two copies of an 8-bit column or (e) four copies of a 4-bit column.

8-bit column (shown in (a)). The corresponding quarter-column architecture is shown in Figure 5.5(b).

The sub-stage parallel designs turn out identical in Figure 5.5, and appear similar to the single-column one in Figure 5.2(a), except that the data buses are split into halves. The architectures are designed based on the scheduling shown in Figure 5.6. In the notations $\boldsymbol{R_j^{p,q}}$, j is the stage index, and p and q are the part indices, where $\{p, q\} \in [0, 3]$ in half-column and $\{p, q\} \in [0, 7]$ in quarter-column. The scheduling

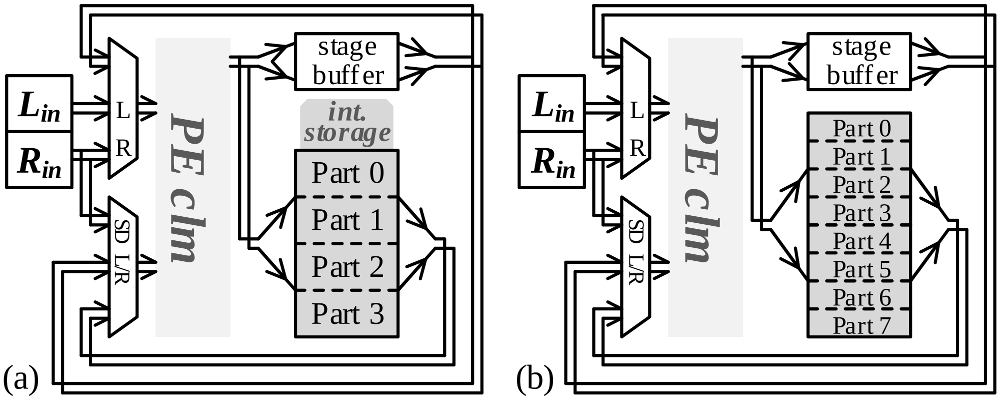
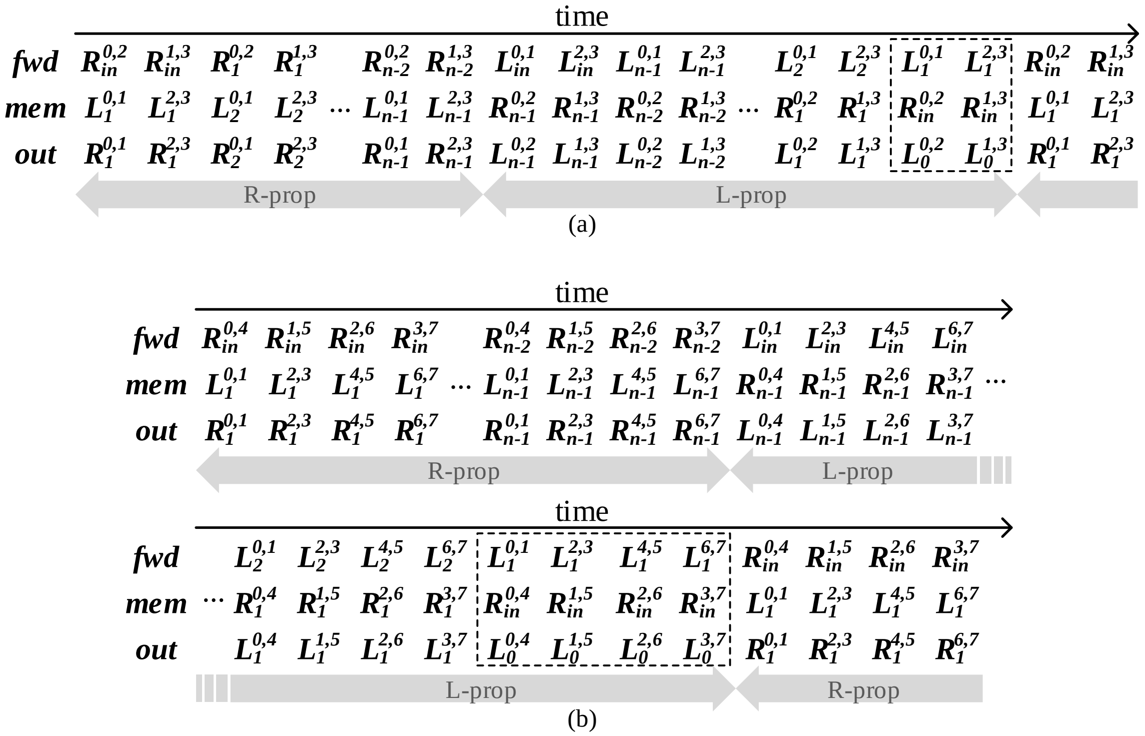


Figure 5.5: (a) Half-column design and (b) quarter-column design.


time
fwd $R_{in}^{0,2}$ $R_{in}^{1,3}$ $R_{1}^{0,2}$ $R_{1}^{1,3}$ $R_{n-2}^{0,2}$ $R_{n-2}^{1,3}$ $L_{in}^{0,1}$ $L_{in}^{2,3}$ $L_{n-1}^{0,1}$ $L_{n-1}^{2,3}$ $L_{2}^{0,1}$ $L_{2}^{2,3}$ $L_{1}^{0,1}$ $L_{1}^{2,3}$ $R_{in}^{0,2}$ $R_{in}^{1,3}$
mem $L_{1}^{0,1}$ $L_{1}^{2,3}$ $L_{2}^{0,1}$ $L_{2}^{2,3}$ ··· $L_{n-1}^{0,1}$ $L_{n-1}^{2,3}$ $R_{n-1}^{0,2}$ $R_{n-1}^{1,3}$ $R_{n-2}^{0,2}$ $R_{n-2}^{1,3}$ ··· $R_{1}^{0,2}$ $R_{1}^{1,3}$ $R_{in}^{0,2}$ $R_{in}^{1,3}$ $L_{1}^{0,1}$ $L_{1}^{2,3}$
out $R_{1}^{0,1}$ $R_{1}^{2,3}$ $R_{2}^{0,1}$ $R_{2}^{2,3}$ $R_{n-1}^{0,1}$ $R_{n-1}^{2,3}$ $L_{n-1}^{0,2}$ $L_{n-1}^{1,3}$ $L_{n-2}^{0,2}$ $L_{n-2}^{1,3}$ $L_{1}^{0,2}$ $L_{1}^{1,3}$ $L_{0}^{0,2}$ $L_{0}^{1,3}$ $R_{1}^{0,1}$ $R_{1}^{2,3}$
R-prop
L-prop
time
fwd $R_{in}^{0,4}$ $R_{in}^{1,5}$ $R_{in}^{2,6}$ $R_{in}^{3,7}$ $R_{n-2}^{0,4}$ $R_{n-2}^{1,5}$ $R_{n-2}^{2,6}$ $R_{n-2}^{3,7}$ $L_{in}^{0,1}$ $L_{in}^{2,3}$ $L_{in}^{4,5}$ $L_{in}^{6,7}$
mem $L_{1}^{0,1}$ $L_{1}^{2,3}$ $L_{1}^{4,5}$ $L_{1}^{6,7}$ ··· $L_{n-1}^{0,1}$ $L_{n-1}^{2,3}$ $L_{n-1}^{4,5}$ $L_{n-1}^{6,7}$ $R_{n-1}^{0,4}$ $R_{n-1}^{1,5}$ $R_{n-1}^{2,6}$ $R_{n-1}^{3,7}$ ···
out $R_{1}^{0,1}$ $R_{1}^{2,3}$ $R_{1}^{4,5}$ $R_{1}^{6,7}$ $R_{n-1}^{0,1}$ $R_{n-1}^{2,3}$ $R_{n-1}^{4,5}$ $R_{n-1}^{6,7}$ $L_{n-1}^{0,4}$ $L_{n-1}^{1,5}$ $L_{n-1}^{2,6}$ $L_{n-1}^{3,7}$
R-prop
L-prop
time
fwd $L_{2}^{0,1}$ $L_{2}^{2,3}$ $L_{2}^{4,5}$ $L_{2}^{6,7}$ $L_{1}^{0,1}$ $L_{1}^{2,3}$ $L_{1}^{4,5}$ $L_{1}^{6,7}$ $R_{in}^{0,4}$ $R_{in}^{1,5}$ $R_{in}^{2,6}$ $R_{in}^{3,7}$
mem ··· $R_{1}^{0,4}$ $R_{1}^{1,5}$ $R_{1}^{2,6}$ $R_{1}^{3,7}$ $R_{in}^{0,4}$ $R_{in}^{1,5}$ $R_{in}^{2,6}$ $R_{in}^{3,7}$ $L_{1}^{0,1}$ $L_{1}^{2,3}$ $L_{1}^{4,5}$ $L_{1}^{6,7}$
out $L_{1}^{0,4}$ $L_{1}^{1,5}$ $L_{1}^{2,6}$ $L_{1}^{3,7}$ $L_{0}^{0,4}$ $L_{0}^{1,5}$ $L_{0}^{2,6}$ $L_{0}^{3,7}$ $R_{1}^{0,1}$ $R_{1}^{2,3}$ $R_{1}^{4,5}$ $R_{1}^{6,7}$
L-prop
R-prop

(a)

(b)

Figure 5.6: Time table of (a) half-column design and (b) quarter-column design.

indicates that the storages require two read ports and two write ports, each with a halved word length of $NQ/2$ bits. The memory depth (of each part) is $4n$ in half-column and $8(n+1)$ in quarter-column. An extra stage buffer is required for sub-stage parallel designs due to the PE results not being immediately used in the next clock cycle, as one stage of long codes takes several clock cycles to complete.

The sub-stage parallel designs save area by reducing number or PEs, while the latency is $2^{k+1}(\log N + k)$ cycles per iteration for a $2^k N$-bit code, and the clock period is expected to be slightly extended compared to the single-column design due to the addressing logic in the stage buffer.

5.1.4 Comparison

We summarize the section by plotting the normalized area and throughput of different designs of different code length and memory implementations in Figure 5.7.

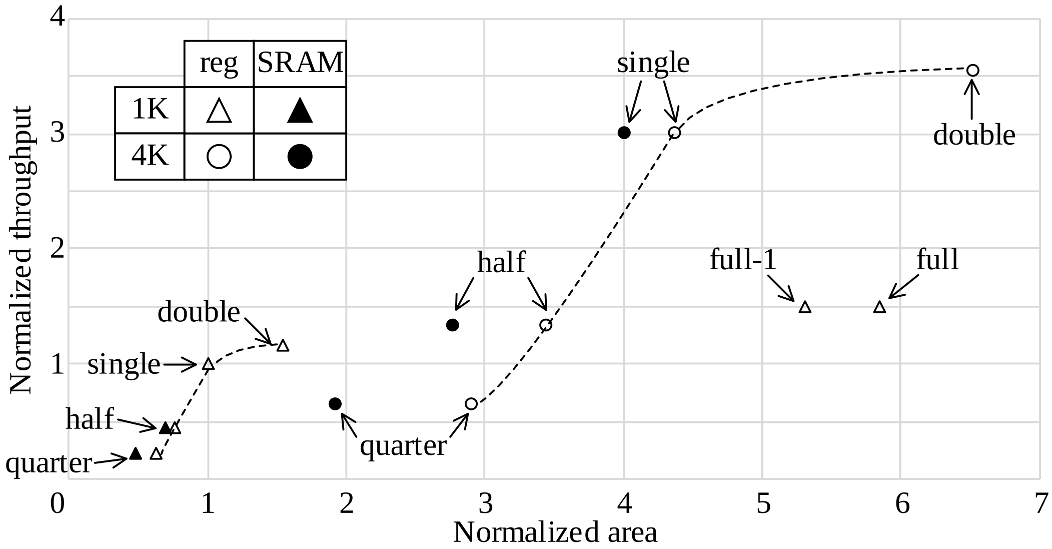


Figure 5.7: Normalized area and throughput of different designs of 1K-bit or 4K-bit BP decoders with register-based or SRAM-based memory.

We show the data points of all the six architectures introduced above for a 1K-bit code, and SRAM-based memory is only applicable to the sub-stage parallel designs where the memory depths are less shallow. For the 4K-bit code, we omit the fully parallel designs due to their large area.

Looking at the register-based 1K-bit decoders (triangle marks, no fill), we find the stage-parallel design the most efficient in terms of throughput per area. The fully parallel designs do not provide much improvement on the throughput, while demanding significant area increment. The sub-stage parallel designs sacrifice much throughput, and they do not save any area due to the constant memory size and additional stage buffer. SRAM-based memory provides area savings, and the savings can be more significant for a more serial design.

The area of 4K-bit decoders scales approximately linearly with respect to the 1K-bit decoder, but the clock period is much longer due to more congested routing. In addition, the cycle count also grows with $\log N$, therefore the 4K-bit designs are lagging in throughput. Although the hardware efficiency of a long code is generally worse than a short code, the error-correcting performance of a long code is better.

Table 5.2: Synthesized Decoder Designs

	area (mm^2)	frequency (MHz)	power (mW)	throughput (Gbit/s) *itr*=5	energy (pJ/bit)
Single	1.325	625	339	6.40	53.0
column	1.404	769	442	7.87	56.2
Double	2.033	400	359	6.83	52.6
column	2.136	476	447	8.12	55.0
Half	1.089	556	236	2.92	80.8
column	1.152	714	318	3.75	84.8

We show the synthesis results of a single-column, a double-column and a half-column design of a 1K-bit BP decoder in 65nm CMOS technology, which are listed in Table 5.2 for comparison. The architectures are designed for a $Q = 6$ 1024-bit polar code, each of which is implemented in two designs that are presented in two rows: a low-power (LP) design and a high-performance (HP) design. The LP design is more compact and consumes less power, but it sacrifices performance. The HP design uses upsizing and buffering, i.e., utilizes more area, to improve performance. In the following, we discuss the trade-off between the three architectures based on the LP designs. The HP designs follow the same trade-off.

The single-column LP design achieves a throughput of 6.4 Gb/s in decoding the 1024-bit polar code, consuming 53 pJ/b. The double-column LP design uses 53% more silicon area to increase the throughput to 6.83 Gb/s, consuming a lower energy of 52.6 pJ/b. Considering the large increase in area, the gain in throughput is relatively low, making the single-column architecture more advantageous, especially in terms of area efficiency.

The half-column LP architecture saves only 18% of the area, but the throughput drops to 2.92 Gb/s. The result is largely due to the constant memory size and the extra register file and muxing. The energy increases to 80.8 pJ/b, making it less attractive than the single-column and double-column architectures. Folding the architecture further is unlikely to yield any improvement due to the constant memory size and the increased control overhead.

5.2 Multi-mode Hardware

For dynamic communication channels where the channel condition fluctuates with time and location, such as wireless channels, multiple codes of different lengths and code rates are needed to support variable channel quality. To keep the implementation cost low, it is desirable to use a configurable decoder compatible with multiple codes. However, supporting configuration in a decoder design can cost throughput, power and significant silicon area. Peak throughputs of reported configurable LDPC and turbo decoder chips have not reached Gb/s and the area efficiency is still below Gb/s/mm^2 [35–37].

The upcoming 5G wireless systems are expected to accommodate even more heterogeneity in quality of channels and quality of service. Instead of handcrafting LDPC or turbo codes and building less-efficient configurable decoders, we opt for polar code [1] as a versatile FEC and take advantage of polar code's regular structure in designing a compact, multi-Gb/s, length- and rate-configurable belief propagation (BP) decoder. The decoder learns the optimal code by selecting the most reliable bits to transmit information through channel-adaptive training, thereby supporting dynamic channels and heterogeneous applications. The demonstrated chip prototype features compact area, high performance, configurability and channel-adaption that are well suited for the upcoming 5G wireless communications.

We take advantage of the highly structured factor graphs of polar codes and the intrinsic tunability of the frozen set to design a configurable belief propagation polar decoder that supports different code lengths and rates. A memory remapping technique is employed to improve the hardware efficiency and design scalability. A low-cost channel-adaptive trainer is added on chip to select the frozen set based on the operating channel. Channel-adaptive codes enhance the error-correcting performance compared to classic polar code designs.

5.2.1 Datapath

The architecture of the configurable design can be illustrated by the figure for the quarter-column design shown in Figure 5.5(b), as the hardware structure looks similar for single-column, half-column and quarter-column designs. The design contains a column of 256 PEs and operates on an $N = 512$ code in the single-column mode. We time-multiplex the PE column for a $2N = 1024$ code in the half-column mode and a $4N = 2048$ code in the quarter-column mode, using the structures shown in Figure 5.4 (b) and (e), respectively. The cycle count per iteration is $2^{k+1}(\log N + k)$ for a $2^k N$-bit code, as discussed in 5.1.3. In this ASIC, $N = 512$ and $k \in \{0, 1, 2\}$, so the throughput degradation from a 512-bit code to a 2K-bit code is 19%.

Switching between the modes is accomplished by different addressing schedule from/to the storages and the stage buffer. The addressing implements the dynamic connections between the groups of $N/2$ PEs and the storages or the stage buffer, where each mode requires a specific pattern of the dynamic connection as shown in Figure 5.4 (b) and (e). The datapath of this ASIC is in a 3-stage pipeline, following a memory read, PE process, and memory write order. The detailed scheduling in different modes will be discussed later.

5.2.2 Memory Implementation

The memory size is determined by the longest supported code length, which is $2^k NQ(\log N + k - 1) = 120\text{K}$ (message length $Q = 6$ in this ASIC). We consider implementing the memory using SRAM due to the large size. However, as is mentioned in Section 5.1.3 that the memory requires two read ports and two write ports in sub-stage parallel modes, which is not available in the commercial SRAMs, so we look into

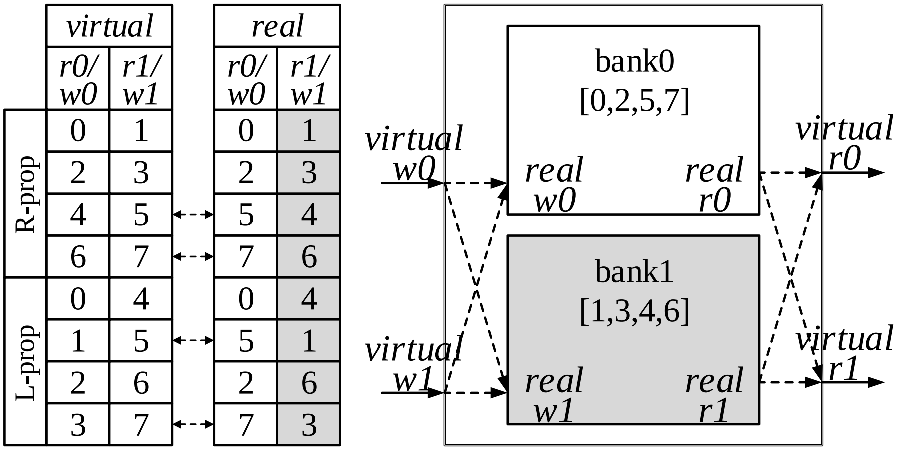


Figure 5.8: Memory access patterns in quarter-column mode with port remap.

memory access patterns and seek more possibilities of 4-port SRAM implementation.

We summarize the memory access patterns of the quarter-column mode in Figure 5.8 based on the schedule shown in Figure 5.6(b). The virtual ports are seen by the PEs, where read/write port0 accessing parts $v_0 \in \{0, 1, 2, 3, 4, 6\}$ are seen by the first half of the PEs, and read/write port1 accessing parts $v_1 \in \{1, 3, 4, 5, 6, 7\}$ are seen by the second half of the PEs. We note that v_0 and v_1 are not exclusive, and that is the reason why a 4-port memory is needed. Then we add selected swapping of port0 and port1 shown in Figure 5.8, where the real ports are seen by the memory. With real port0 accessing parts $r_0 \in \{0, 2, 5, 7\}$ and port1 accessing the rest parts $r_1 \in \{1, 3, 4, 6\}$, the memory can be split into two banks, and each can be implemented by a dual-port SRAM with a width of $NQ/2 = 1.5\text{K}$ and a depth of $2^k(\log N + k - 1) = 40$.

With the virtual 4-port memory being implemented by two SRAMs for the most complex quarter-column mode, we check the compatibility of the half-column mode and we show the memory access patterns in Figure 5.9, based on the schedule shown in Figure 5.6(a) . We note that PEs need to see parts $\{0, 2\}$ or $\{1, 3\}$ at the same time (in L-prop), but the two ports cannot access the same SRAM bank. We resolve this conflict by remapping entry 2 to 4 and 3 to 5, so that the same port remapping used

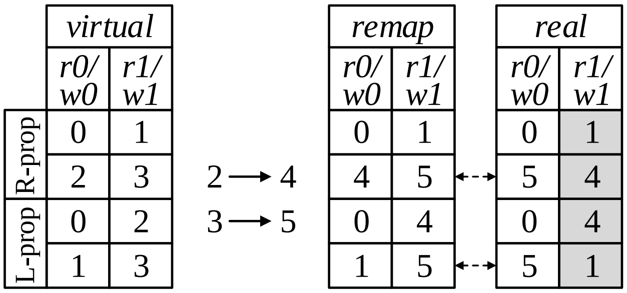


Figure 5.9: Memory access patterns in half-column mode with entry remap and port remap.

in the quarter-column mode is valid for half-column mode too. The single-column mode only accesses part $\{0, 1\}$, which is also valid. Such entry remapping and port remapping techniques are applicable to configurable designs that support an arbitrary level of sub-stage parallelism.

The pipelined datapath allows one cycle of worst-case read access, which is demanding for the SRAM. We add 3K bypass registers to mimic memory read-after-write in this ASIC to match the pipeline. Another option is to stall the pipeline 5.9% of the time in single-column mode, 5.3% of the time in half-column mode, and 2.4% of the time in quarter-column mode without area overhead.

Compared with a register-based design, the SRAM-based design saves 20% area and 31% power. The SRAM-based configurable design can be efficiently scaled up, e.g., support even longer 4K-bit or 8K-bit codes, which only grows the area by 15% or 45%, respectively, while the register-based design is expected to grow by 80% or 250%, respectively.

5.2.3 Stage Buffer

As is discussed in Section 5.1.3, a stage buffer is needed to temporarily hold sub-stage results. The stage buffer is implemented in register files, because it is in the critical path. The straightforward implementation is a 4-port register file that stores 8 words, corresponding to 8 parts of data in one stage. Writing to the register file is through eight 2-to-1 input muxes and reading is through two 8-to-1 output muxes. Each input or output port of the mux has a width of 1.5Kb. The input 2-to-1 muxes need to be expanded to 3-to-1 muxes if the register file does not have an enable input. In the single-column decoding mode, data are passed from the inputs to the stage buffer to the outputs of the stage buffer, requiring an extra port to be added to the input muxes to support bypass. This straightforward implementation costs much area and long reading latency, so we customize the design to reduce the size of the register file and the cost of the muxes.

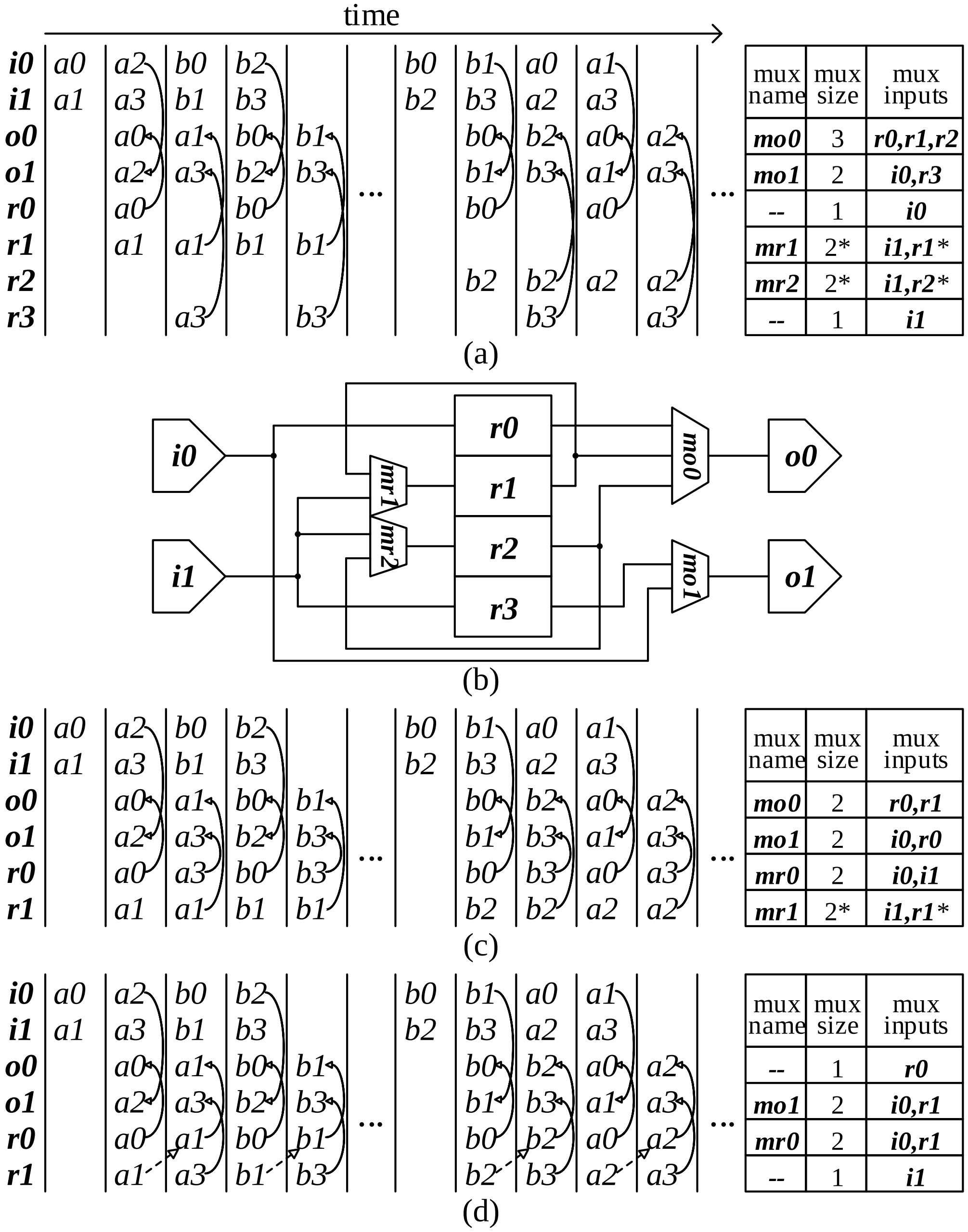
We begin by illustrating the data flow along with the mux configuration in the half-column decoding mode that is explained in Figure 5.6(a). We show the cycle-by-cycle inputs to the stage buffer ($\boldsymbol{i0}$ and $\boldsymbol{i1}$), outputs from the stage buffer ($\boldsymbol{o0}$ and $\boldsymbol{o1}$), contents of the stage buffer, and mux configurations in Figure 5.10. a and b represent the data from two consecutive decoding stages, and the number 0 to 3 following a or b is the part index. For example, part 0 of stage a is $a0$. Recall that the results of each decoding stage are divided into 4 parts in the half-column decoding mode or 8 parts in the quarter-column decoding mode. In the first cycle illustrated in Figure 5.6(a), the results from the PEs, $a0$ and $a1$, are sent to the stage buffer, followed by $a2$ and $a3$ in the second cycle. While in the second cycle, $a0$ and $a2$ are forwarded to the PEs for the second stage processing, which produces $b0$ and $b1$ in the third cycle. In R-prop, stage a is processed prior to b, while in L-prop, stage a is processed after b. The arrows indicate bypass in the same cycle. An "*" means that one mux input is removed by making use of the register file's enable input. A 1-to-1 mux is replaced by a wire.

The simple scheduling shown in Figure 5.10(a) corresponds to the block diagram shown in (b), where there are four parts of the register file, $\boldsymbol{r0}$, $\boldsymbol{r1}$, $\boldsymbol{r2}$ and $\boldsymbol{r3}$, and each part stores a part of a stage. The scheduling requires one 3-to-1 mux $\boldsymbol{mo0}$ and three 2-to-1 muxes $\boldsymbol{mo1}$, $\boldsymbol{mr0}$ and $\boldsymbol{mr1}$, each with a port width of 1.5Kb. The schedule indicates that only half of the register file is occupied at any time, so we merge the parts $\boldsymbol{r1}$ with $\boldsymbol{r2}$, and $\boldsymbol{r0}$ with $\boldsymbol{r3}$, as shown in (b). Merging reduces the register file size by 3Kb, and allows the 3-to-1 $\boldsymbol{mo0}$ to be reduced to a 2-to-1 mux. To further reduce the cost of muxing, we allow forwarding between parts, i.e., data in $\boldsymbol{r1}$ is forwarded to $\boldsymbol{r0}$ directly by connecting them together, as indicated by the dashed arrows in Figure 5.10(c). Forwarding reduces the number of muxes by half. The control of muxing, enable and forwarding is orchestrated by a 2-state FSM, as illustrated in Figure 5.11, where the select signal of mux $\boldsymbol{mr0}$ is $sr0$, and the select signal of mux $\boldsymbol{mo1}$ is $so1$, etc.

The same merging and forwarding techniques can be applied to the slightly more complex quarter-column decoding mode. Since reading from the stage buffer is in the critical path, we keep the output mux size by only connecting $\boldsymbol{o0}$ and $\boldsymbol{o1}$ to $\boldsymbol{r0}$ and $\boldsymbol{r1}$, respectively, as these paths are already established in the half-column decoding mode as shown in Figure 5.10(c). The control of writing to the stage buffer is further simplified by connecting $\boldsymbol{i0}$ and $\boldsymbol{i1}$ to $\boldsymbol{r2}$ and $\boldsymbol{r3}$, respectively, if $\boldsymbol{i0}$ and $\boldsymbol{i1}$ are not directly forwarded to the outputs.

Using the schedule shown in Figure 5.6(b), we show the register arrangement in the R-prop in Figure 5.12(a). The writing and reading paths are highlighted in dashed ovals and rounded rectangles, respectively. Note that $a4$ is the input in the third cycle and the output in the fourth cycle, so it is written directly to $\boldsymbol{r1}$, instead of $\boldsymbol{r2}$, and the storage of $\boldsymbol{r2}$ remains unchanged in the fourth cycle. The arrangement that uses the minimum muxing is summarized in the table in Figure 5.12(a).

We show the L-prop schedule in Figure 5.12(b). The dashed arrows indicate the additional muxing paths not covered by the setup for R-prop, and the muxes need to be expanded as shown in the table of Figure 5.12(b). The muxing expansion can be avoided if we swap the processing order of the second quarter and the third quarter of each stage, i.e., processing $b2$ and $b6$ prior to $b1$ and $b5$, as shown in Figure 5.12(c).



i0	a0	a2	b0	b2			b0	b1	a0	a1		
i1	a1	a3	b1	b3			b2	b3	a2	a3		
o0		a0	a1	b0	b1			b0	b2	a0	a2	
o1		a2	a3	b2	b3	...		b1	b3	a1	a3	...
r0		a0		b0				b0		a0		
r1		a1	a1	b1	b1							
r2								b2	b2	a2	a2	
r3			a3		b3				b3		a3	

mux name	mux size	mux inputs
mo0	3	***r0,r1,r2***
mo1	2	***i0,r3***
--	1	***i0***
mr1	2*	***i1,r1****
mr2	2*	***i1,r2****
--	1	***i1***

(a)

(b)

i0	a0	a2	b0	b2			b0	b1	a0	a1		
i1	a1	a3	b1	b3			b2	b3	a2	a3		
o0		a0	a1	b0	b1			b0	b2	a0	a2	
o1		a2	a3	b2	b3	...		b1	b3	a1	a3	...
r0		a0	a3	b0	b3			b0	b3	a0	a3	
r1		a1	a1	b1	b1			b2	b2	a2	a2	

mux name	mux size	mux inputs
mo0	2	***r0,r1***
mo1	2	***i0,r0***
mr0	2	***i0,i1***
mr1	2*	***i1,r1****

(c)

i0	a0	a2	b0	b2			b0	b1	a0	a1		
i1	a1	a3	b1	b3			b2	b3	a2	a3		
o0		a0	a1	b0	b1			b0	b2	a0	a2	
o1		a2	a3	b2	b3	...		b1	b3	a1	a3	...
r0		a0	a1	b0	b1			b0	b2	a0	a2	
r1		a1	a3	b1	b3			b2	b3	a2	a3	

mux name	mux size	mux inputs
--	1	***r0***
mo1	2	***i0,r1***
mr0	2	***i0,r1***
--	1	***i1***

(d)

Figure 5.10: Stage buffer scheduling of a half-column design implemented by (a) regular register file with its block diagram in (b), (c) with $\boldsymbol{r1}$ merged with $\boldsymbol{r2}$ and $\boldsymbol{r1}$ merged with $\boldsymbol{r3}$, and (d) with $\boldsymbol{r1}$ to $\boldsymbol{r0}$ propagation.

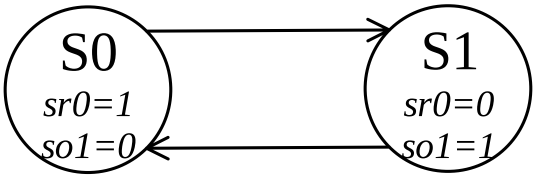


Figure 5.11: Stage buffer controller of a half-column design implemented by a 2-state FSM.

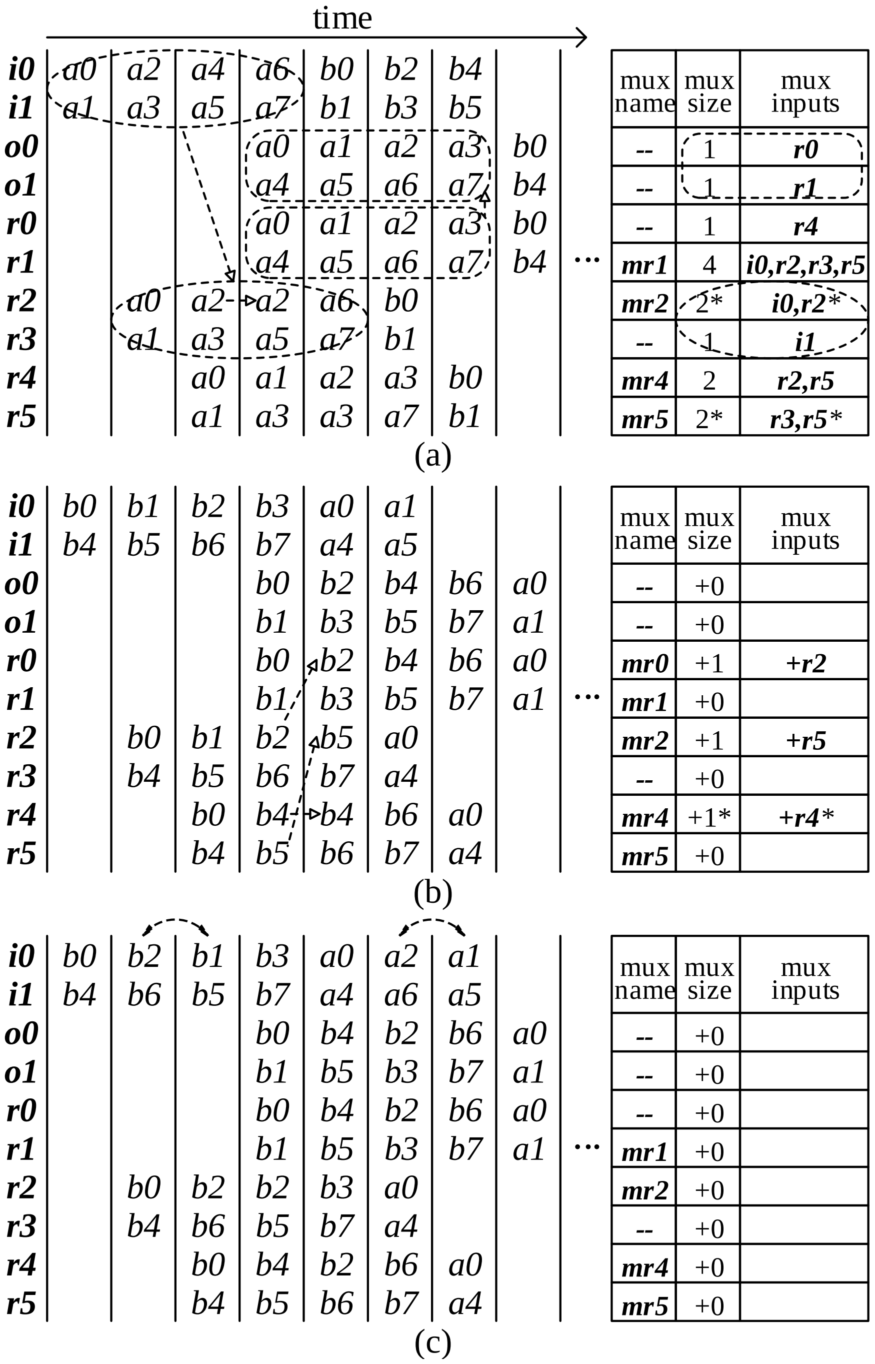
The control FSM is shown in Figure 5.13.

In conclusion, by merging, forwarding and selective swapping, we cut 3Kb from the stage buffer design, and reduce the size of input and output muxes. As a result, the clock period is shortened by 8.3%.

5.2.4 Floorplan

As introduced in 5.2.2 that the memory consists of two banks and each bank stores 40 1.5Kb-wide words. The memory is wide and shallow. The 1.5Kb-long word is formed by concatenating 12 128-bit segments, so one memory bank is composed of 12 SRAMs and the entire memory consists of 24 SRAMs. The placement of the SRAMs is shown in Figure 5.14. We put pairs of SRAMs back-to-back and let the ports face outwards. Bank0 and bank1 of each segment is placed closely, because they are the two inputs of the port switching mux described in 5.2.2. To balance the length and width of the chip, we place 12 SRAMs (or two banks of 6 segments) on the left and another 12 SRAMs on the right. The word segment indices are marked in bold italic font.

Having the freedom of arranging the location of the 12 segments, we put the segments close to each other if they are logically related, to ease the routing and to reduce chip area and clock period. The connectivity of the segments can be derived based on the input/output connectivity of a PE column. Ideally we need to find the connectivity among every 1/12 of a column, but 12 is not a power of 2, so instead we sketch the connectivity among every 1/8 of a column for an approximate reference



time

										mux name	mux size	mux inputs
i0	a0	a2	a4	a6	b0	b2	b4					
i1	a1	a3	a5	a7	b1	b3	b5					
o0				a0	a1	a2	a3	b0		--	1	r0
o1				a4	a5	a6	a7	b4		--	1	r1
r0				a0	a1	a2	a3	b0		--	1	r4
r1				a4	a5	a6	a7	b4	...	mr1	4	i0,r2,r3,r5
r2		a0	a2	a2	a6	b0				mr2	2*	i0,r2*
r3		a1	a3	a5	a7	b1				--	1	i1
r4			a0	a1	a2	a3	b0			mr4	2	r2,r5
r5			a1	a3	a3	a7	b1			mr5	2*	r3,r5*

(a)

										mux name	mux size	mux inputs
i0	b0	b1	b2	b3	a0	a1						
i1	b4	b5	b6	b7	a4	a5						
o0				b0	b2	b4	b6	a0		--	+0	
o1				b1	b3	b5	b7	a1		--	+0	
r0				b0	b2	b4	b6	a0		mr0	+1	+r2
r1				b1	b3	b5	b7	a1	...	mr1	+0	
r2		b0	b1	b2	b5	a0				mr2	+1	+r5
r3		b4	b5	b6	b7	a4				--	+0	
r4			b0	b4	b4	b6	a0			mr4	+1*	+r4*
r5			b4	b5	b6	b7	a4			mr5	+0	

(b)

										mux name	mux size	mux inputs
i0	b0	b2	b1	b3	a0	a2	a1					
i1	b4	b6	b5	b7	a4	a6	a5					
o0				b0	b4	b2	b6	a0		--	+0	
o1				b1	b5	b3	b7	a1		--	+0	
r0				b0	b4	b2	b6	a0		--	+0	
r1				b1	b5	b3	b7	a1	...	mr1	+0	
r2		b0	b2	b2	b3	a0				mr2	+0	
r3		b4	b6	b5	b7	a4				--	+0	
r4			b0	b4	b2	b6	a0			mr4	+0	
r5			b4	b5	b6	b7	a4			mr5	+0	

(c)

Figure 5.12: Stage buffer scheduling of a quarter-column design (a) R-prop in regular order (b) L-prop in regular order and (c) L-prop in a permuted order.

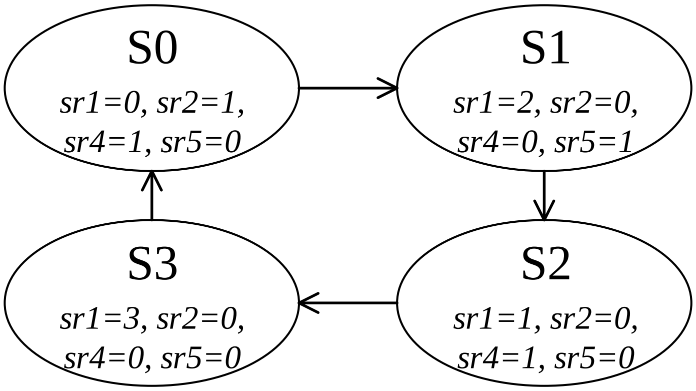


Figure 5.13: Stage buffer controller of a quarter-column design implemented by 4-state FSM.

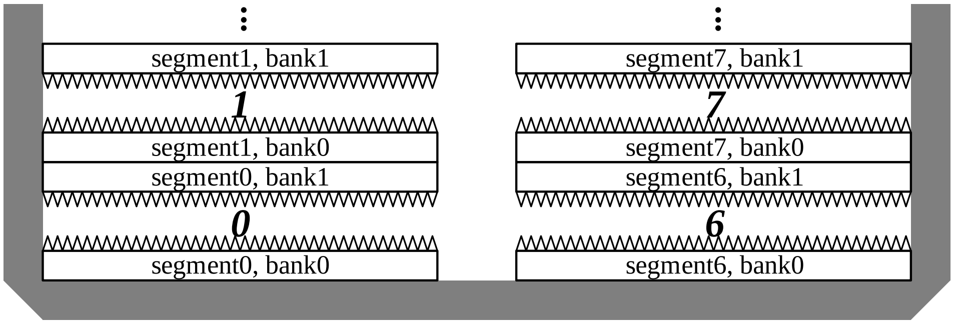


Figure 5.14: SRAM placement.

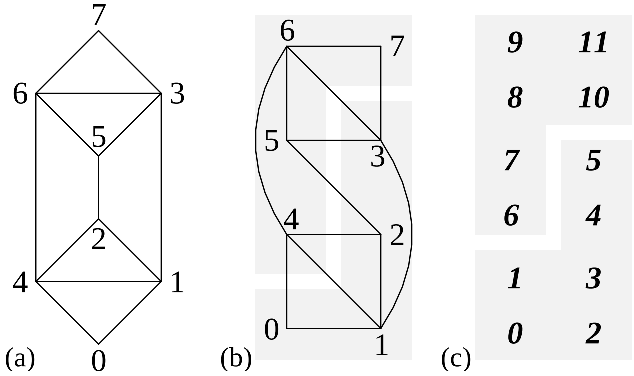


Figure 5.15: (a) Sub-word connections, (b) rotate to a floorplan and (c) relate to the 12 segments.

in Figure 5.15(a). The connectivity graph is rotated to arrive at (b), and the 12 segments are placed accordingly, as shown in (c). Using such an arrangement of the segments, the standard cell density is increased by 3.6% and the clock frequency is improved by 4.2% compared to the straightforward placement shown in Figure 5.14.

We separate the power supplies for the SRAM and the standard cells on this chip to measure the power of each part. Because the separated power routing takes more routing traces, the chip area grown by 8.5% to provide more available traces for signal routing.

5.2.5 Chip Implementation

The test chip is manufactured in 40nm CMOS technology. The decoder core consists of 256 PEs, a 120Kb virtual four-port memory that is connected to the PEs via routers, and a 9Kb stage forward buffer to forward messages between consecutive stages, as shown in Figure 5.16. Apart from the main datapath, a 12Kb frame storage keeps one frame of channel inputs accessible. The hard decision is an on-chip checker that collects error rates and helps to perform early termination introduced in section 3.3.2. The channel adaptive trainer applies the in-order bit selection algorithm introduced in section 3.3, as shown in Figure 5.17.

During the training phase, with cooperation of the transmitter that sends all-0

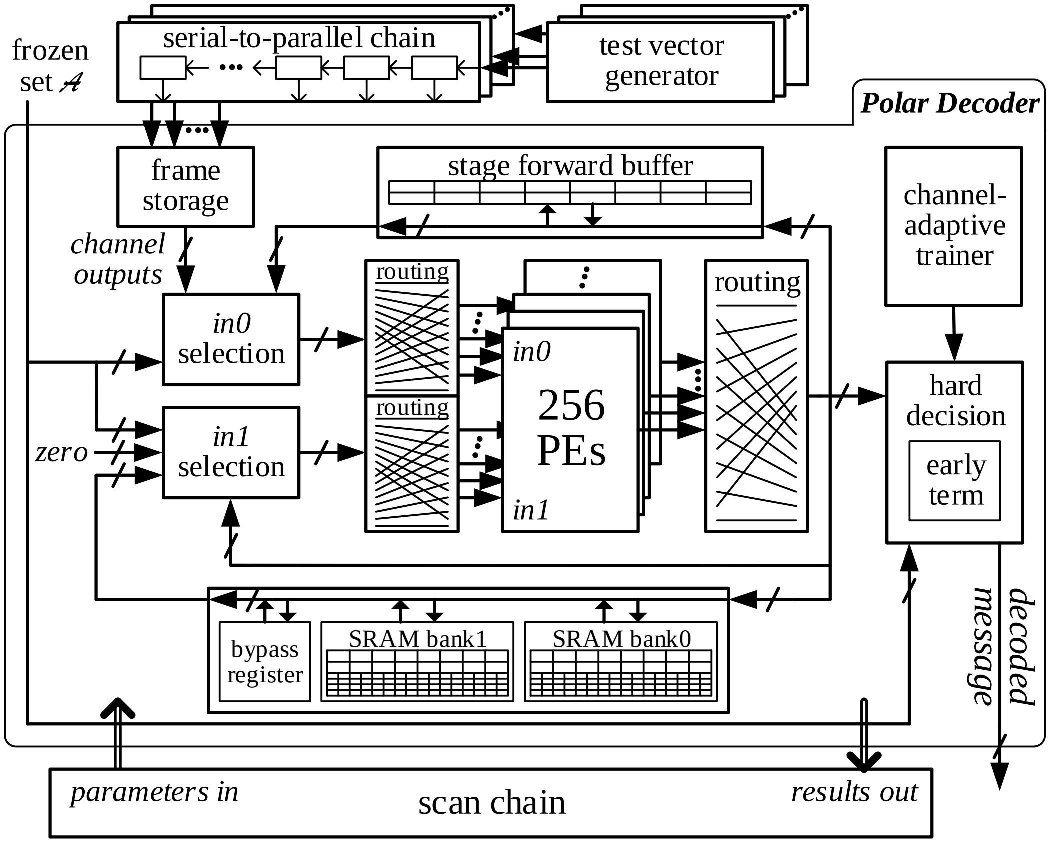


Figure 5.16: Block diagram of the configurable BP polar decoder chip.

codewords, the decoder in the receiver evaluates BER of each bit using an empirical method motivated by density evolution [28]. The trainer ranks the bits based on their BER, and the optimal free (frozen) bits are learned and fed back to the transmitter. Following the training phase, both the transmitter and receiver can agree to use the optimal bit selection for transmission to take advantage of channel-adaptive code. The results of channel-adaptive code is shown in Figure 5.18(a).

When implementing the f function, the original hyperbolic expression in equation 1.1 is simplified by the min-sum approximation, which leads to arithmetic inaccuracy and degradation of error-correcting degradation. Such approximation error can be overcame by scaling, however a multiplier is expensive in hardware, and therefore the correction is usually implemented by offset correction.

Since the distribution of f function result can spread a wide range, a desired offset

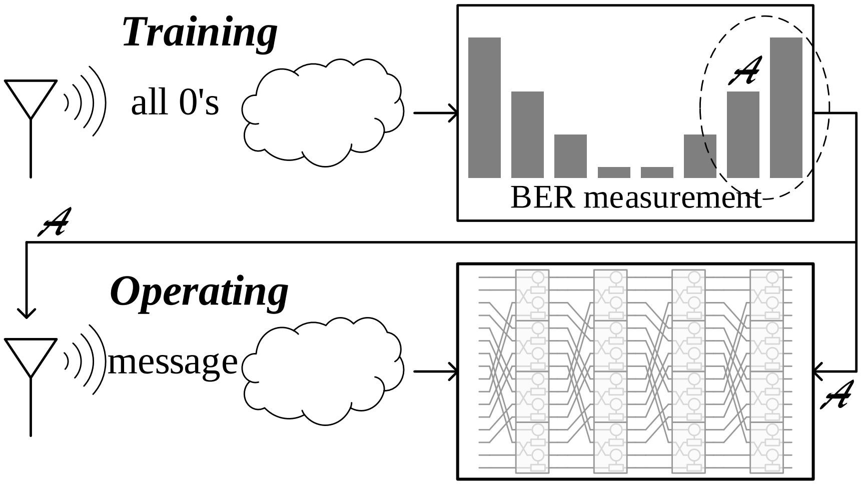


Figure 5.17: Channel-adaptive training flow.

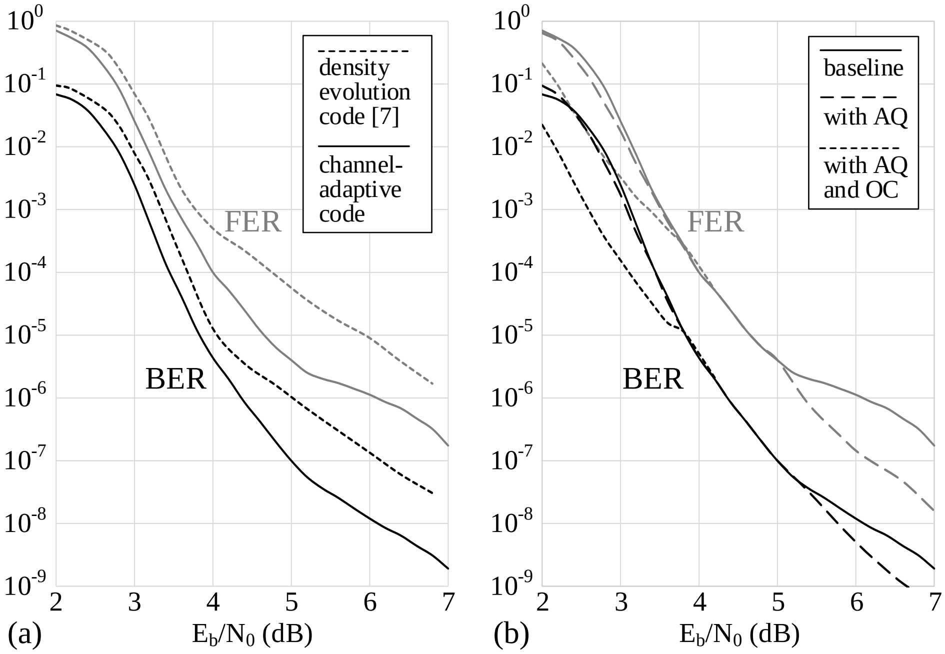


Figure 5.18: Improved coding gain using (a) channel-adaptive code, and (b) SNR-adaptive tuning including adaptive quantization (AQ) and offset correction (OC).

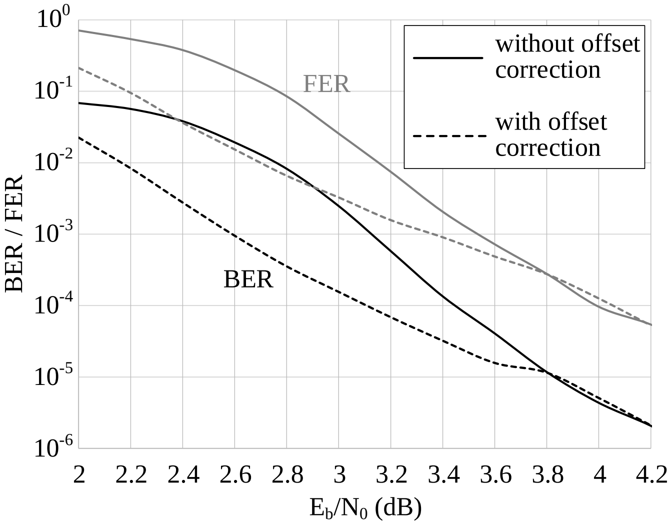


Figure 5.19: Offset correction effect on a (1024, 512) code.

should also scale with the magnitude of the result, which requires a few comparators and a subtracter before the output of each f function. Simulation has shown that the effect of offset correction is significant in low SNR but negligible in high SNR. Therefore to minimize the hardware overhead, the offset correction is designed only for small magnitude, using only one comparator and subtracter. Figure 5.19 shows that the error rates at low SNR can be lowered by more than an order of magnitude with the help of offset correction.

In almost all decoder designs, the datapaths use a fixed number of bits to represent the numbers, and the word length is desired to be as short as a few number of bits. How to allocate the finite number of bits involves a trade-off between precision and range. For example, a word with a fixed length of $Q = 6$ can represent an integer in the range of $[-32, 31]$ with a precision of 2^0; or it can represent a smaller range of $[-8, 7.75]$ with a fine precision of 2^{-2}; or it can also represent a larger range of $[-128, 124]$ with a coarse precision of 2^2, i.e., the lower 2 bits are rounded off.

Table 5.3 shows the statistics of the priors, and Figure 5.20 shows that at a low SNR, a fine precision leads to a better performance despite the lower range; while at a high SNR, when the mean and variance of the messages are larger, a higher range

Table 5.3: Prior Statistics

SNR	0	1	2	3	4	5	6	7
μ	2.00	2.52	3.17	3.99	5.02	6.32	7.96	10.02
σ	2.02	2.26	2.53	2.84	3.18	3.56	4.00	4.49
min	-7	-8	-8	-9	-9	-10	-10	-10
max	11	13	14	17	19	22	26	30

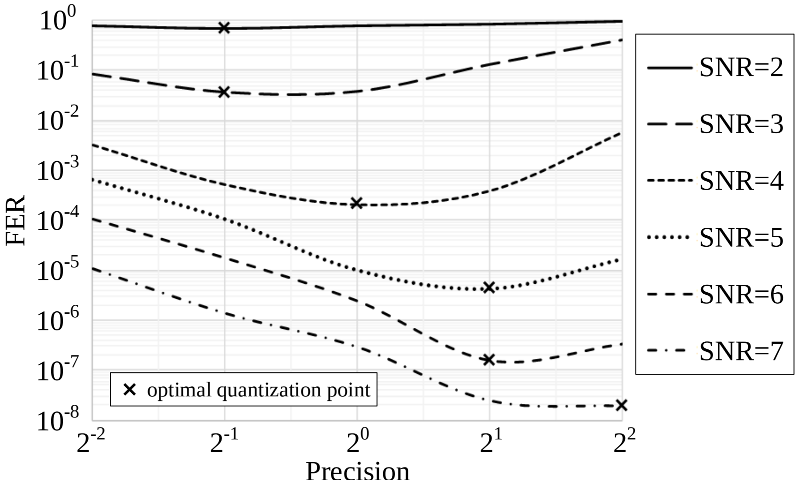


Figure 5.20: Quantization effect on FER of a $Q = 6$ (1024, 512) code.

improves performance. With a fixed word length, it is desirable to use an adaptive quantization depending on channel SNR.

Conditional subtraction at the f function inside PE is added to implement offset correction, and several MUXes at the test vector generator implements the adaptive quantization. The results of SNR-adaptive performance tuning are shown in Figure 5.18(b).

In addition to the polar decoder core, a few peripherals are included to test the chip. A set of test vector generators and serial-to-parallel chains feed the core with random streams as inputs received from the channel. A two-phase scan chain programs parameters into the core and extracts the results out from the core.

The microphotograph of the 40nm CMOS test chip is shown in Figure 5.21. The polar decoder core occupies 1.07mm^2 with a 76.2% standard cell utilization. The

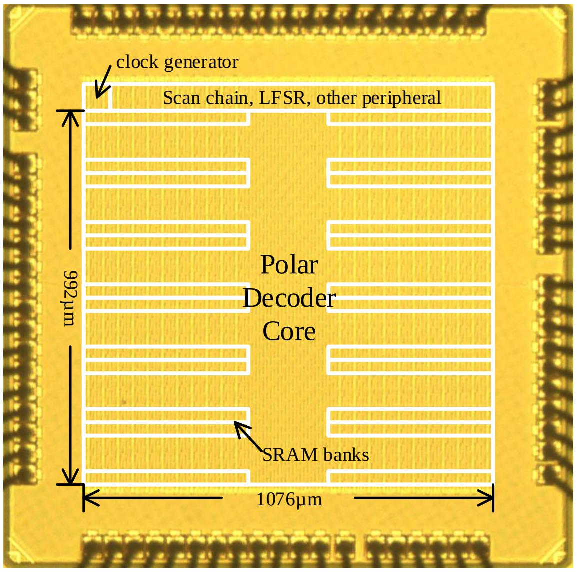


Figure 5.21: chip microphotograph.

memory consists of 24 dual-port physical SRAMs with 12 SRAMs per bank. The 12 SRAMs together provide a sufficiently long word needed by 256 PEs working in parallel. The power supply of the SRAMs is isolated from the rest of the core so that the supply voltages can be tuned separately.

5.2.6 Chip Measurement

Figure 5.22 is the chip testing setup photo. The test chip is measured under room temperature. The voltage and frequency measurements are shown in Figure 5.23(a), and the throughputs are reported based on 5 decoding iterations. Peak throughputs of 3.78Gb/s and 3.06Gb/s are achieved in decoding 512-bit polar codes and 2048-bit polar codes, respectively, at 900mV nominal supply voltage. The power and energy measurements in decoding 512-bit polar codes are shown in Figure 5.23(b). At 627MHz, the decoder achieves a peak throughput of 3.78Gb/s, consuming 170pJ/bit. Voltage scaling to 530mV for the core and 480mV for the memory lowers the energy to 39.3pJ/bit at 91.4MHz, providing a throughput of 551Mb/s.

The chip demonstrates high versatility compared to the state-of-the-art single-

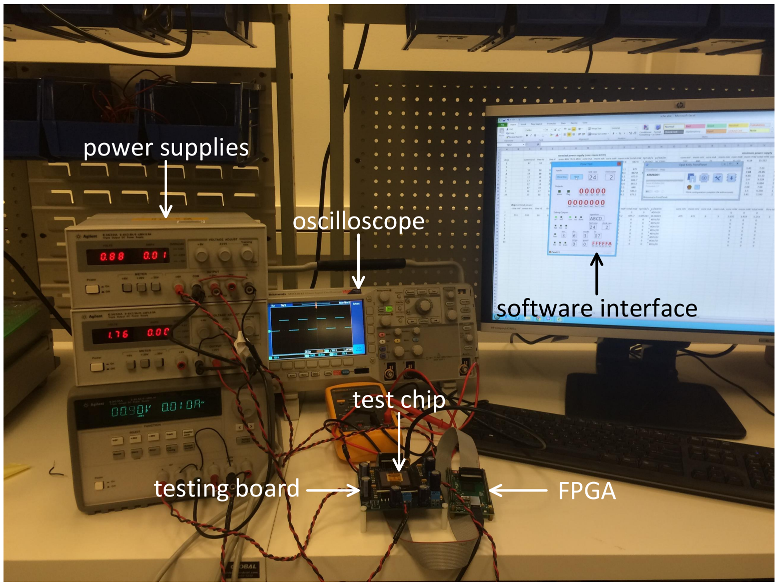
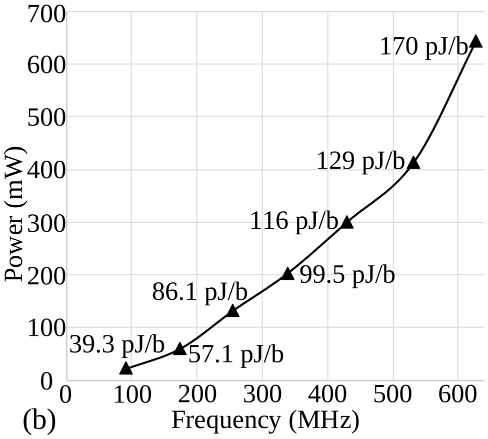


Figure 5.22: Test setup photo.

V_{CORE} (mV)	V_{MEM} (mV)	Freq. (MHz)	Throughput (Mb/s)	
			N=512	***N***=2048
900	900	627	3779	3059
735	850	532	3202	2592
690	800	430	2591	2097
665	680	339	2040	1651
645	570	255	1539	1246
555	520	174	1051	851
530	480	91.4	551	446

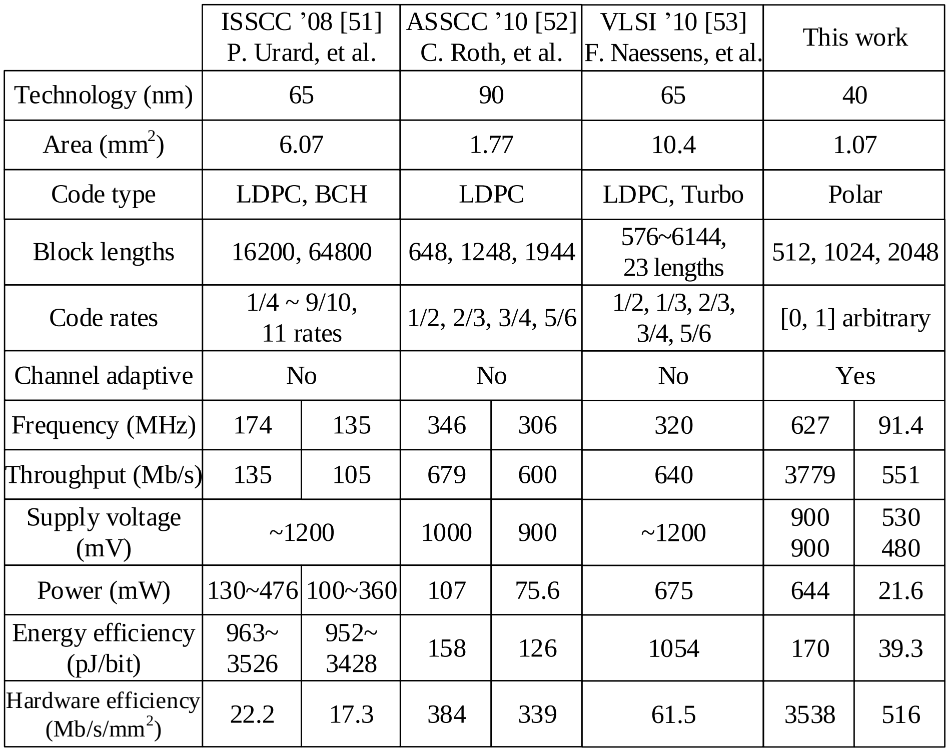
(a)



(b)

Figure 5.23: chip measurement.

Table 5.4: Comparison to recently published configurable FEC decoders.



	ISSCC ’08 [51] P. Urard, et al.		ASSCC ’10 [52] C. Roth, et al.		VLSI ’10 [53] F. Naessens, et al.	This work	
Technology (nm)	65		90		65	40	
Area (mm^2)	6.07		1.77		10.4	1.07	
Code type	LDPC, BCH		LDPC		LDPC, Turbo	Polar	
Block lengths	16200, 64800		648, 1248, 1944		576~6144, 23 lengths	512, 1024, 2048	
Code rates	1/4 ~ 9/10, 11 rates		1/2, 2/3, 3/4, 5/6		1/2, 1/3, 2/3, 3/4, 5/6	[0, 1] arbitrary	
Channel adaptive	No		No		No	Yes	
Frequency (MHz)	174	135	346	306	320	627	91.4
Throughput (Mb/s)	135	105	679	600	640	3779	551
Supply voltage (mV)	~1200		1000	900	~1200	900 900	530 480
Power (mW)	130~476	100~360	107	75.6	675	644	21.6
Energy efficiency (pJ/bit)	963~ 3526	952~ 3428	158	126	1054	170	39.3
Hardware efficiency (Mb/s/mm^2)	22.2	17.3	384	339	61.5	3538	516

mode polar decoder [38]. With a maximum iteration of 30 and early termination, a rate-0.5 512-bit channel-adaptive polar code learned by the trainer for the AWGN channel outperforms the theoretical code design [28] by 0.3dB at FER of 10^{-4}. Increasing the code length from 512 bits to 2048 bits improves the coding gain by 0.5dB at the cost of 19% throughput. The code rate can be adjusted lower to further improve the SNR, or higher to increase the information throughput.

Comparison to the recently published configurable FEC decoders [35–37] is shown in Table 5.4. This chip is the first configurable FEC decoder that exceeds Gb/s in throughput. The energy efficiency of the chip is comparable to the state-of-the-art configurable LDPC decoder chip [36], and the area efficiency of the chip is almost an order of magnitude higher. The chip is the first channel-adaptive configurable FEC decoder suitable for 5G wireless communications.

5.3 Summary

In this chapter, a family of BP polar decoder architectures for polar codes and their gate-level implementations to demonstrate the trade-offs between area, power, and throughput. The single-column architecture stands out as the most efficient when considering both area and energy consumption.

A length- and rate-configurable BP polar decoder chip is presented to demonstrate the flexibility and scalability of the architecture, so that polar code has the prospect to serve as a universal ECC that can adapt to different communication specifications. The virtual four-port memory implementation minimized the dominant portion of the power and area consumption, which makes larger designs more feasible.

CHAPTER VI

Conclusion

In this dissertation, the algorithm, architecture and implementation of polar codes are studied and analyzed from the system level to circuit level; software simulation, FPGA emulation and circuit integration are used to produce concrete prototypes from concepts; and approaches of improving the usability of polar codes are discovered.

We developed an FPGA emulation platform for studying rare-case error events of polar codes. To reduce design effort, we constructed the emulation platform by the parameterized blocks and scripts. We propose a simulation-based bit selection that is applicable to any physical channel. The simulation-based bit selection provides a channel-adaptive bit selection for the optimal error-correcting performance. The error-correcting performance is further improved by inspecting the error patterns and fine tuning the decoding algorithm. In particular, the errors are categorized based on their unique signatures, and targeted post-processing is applied to each type of errors. These algorithm-level improvements boost the error-correcting performance of polar codes, without adding any implementation complexity.

The regularity of the polar code structure enables many design choices. For a specific use case, the optimal architecture can be chosen from the design space to best meet the specifications. The single-column architecture is found to naturally utilize the intrinsic stage-parallelism of BP decoding, and it is shown to reach a good balance between performance and cost. We further investigate alternative architectural choices and the tradeoffs. A more parallel architecture offers a higher throughput if more area can be afforded. The high memory bandwidth of a highly parallel architecture requires using registers, which could introduce high cost in area and power. A less parallel architecture saves area by less computation logic and condensed SRAM-based memory. The architecture investigation leads to a configurable architecture, which supports different codes using different degrees of parallelism.

With this study, we improved the error-correcting performance of polar codes and make them a competitor to the high-performance ECCs such as LDPC and turbo codes, and we implemented an efficient configurable polar decoder that can adapt to the changing wireless channels for 5G communications.

APPENDICES

APPENDIX A

Case Study – LDPC Emulation for Error Floor Exploration

LDPC codes are widely used for communication and storage systems, including Ethernet [39], Wi-Fi [40], WiMAX [41], satellite [42], as well as magnetic [43] and solid-state storage [44]. Properly designed LDPC code is capacity-approaching [45], referring to approaching the ultimate lowest SNR needed for a reliable transmission. Much research has been done on LDPC codes and codec designs in the past ten years, leading to well-known code constructions as well as efficient and high-performance encoder and decoder designs.

The error-correcting performance of an LDPC code is often shown in a plot of BER or FER against SNR. As SNR increases, BER and FER improve dramatically, producing a curve that looks like a waterfall, as illustrated in Figure A.1. However, a key problem with many LDPC codes is that they often cease to work well at low BER, causing the waterfall curve to bend in a phenomenon called error floor [24]. The error floor presents a big problem to applications that require low error rates, as an excessive SNR is needed to reach low error rates, costing extra power.

In this section, we show how to use FPGA to investigate the error-correcting performance of LDPC codes, and find the root cause of error floors. The root cause motivates an improved decoding algorithm that overcomes the error floor problem.

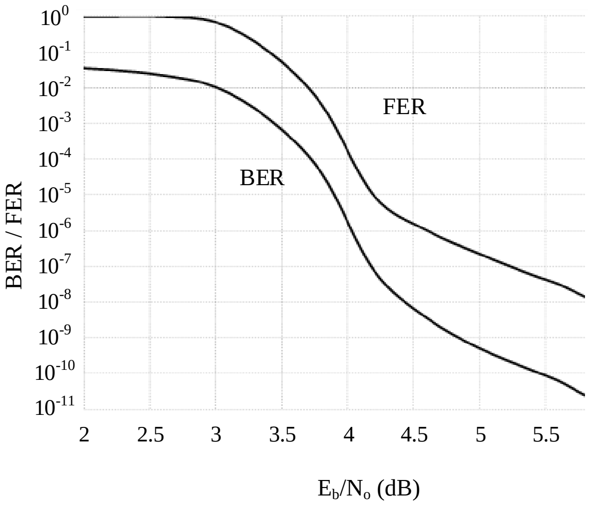


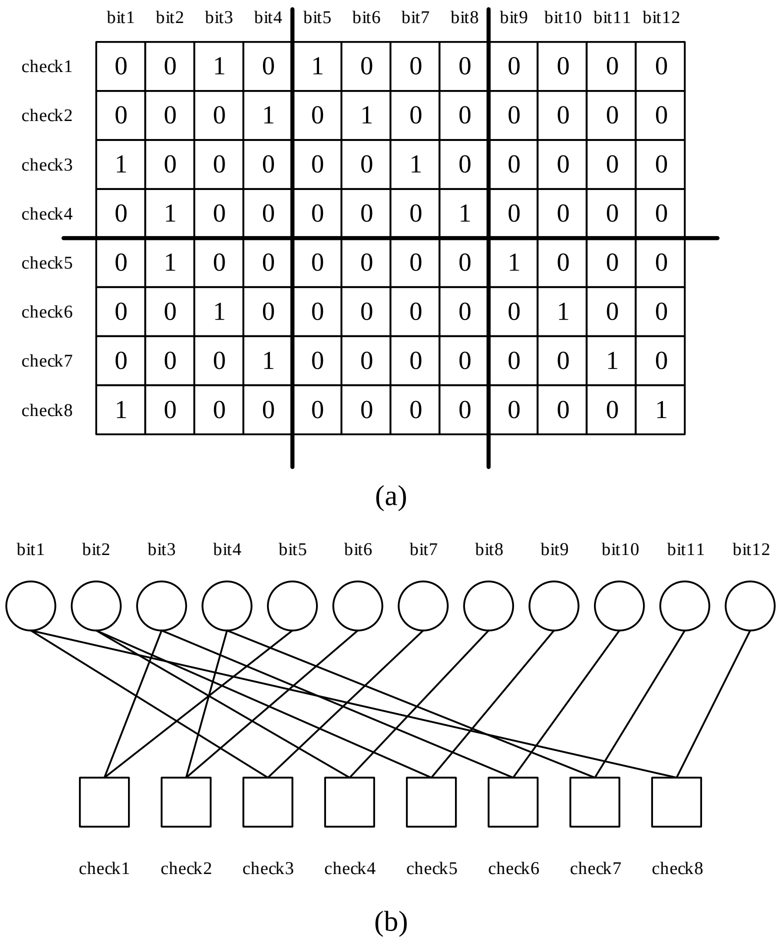
Figure A.1: Waterfall curve of an LDPC code.

An FPGA is an integral part of the investigation and is heavily utilized for decoder emulation and rare event discovery.

A.1 Introduction to LDPC code and decoder design

An LDPC code is defined by a $m \times n$ parity-check matrix H, where n is block length of the code, or number of bits in the codeword, and m is the number of parity checks [46,47]. An 8×12 H matrix example is shown in Figure A.2(a) that represent a 12-bit codeword and 8 parity checks. For example, the first row of the H matrix specifies that the XOR of bit 3 and bit 5 need to satisfy even parity.

The H matrix can be graphically represented using a factor graph, shown in Figure A.2(b), where each bit in the codeword is drawn as a circle, known as variable node (VN) and each check is drawn as a square, known as check node (CN), and an edge connects VN j and CN i if the corresponding entry in the H matrix, $H(i, j) = 1$.



	bit1	bit2	bit3	bit4	bit5	bit6	bit7	bit8	bit9	bit10	bit11	bit12
check1	0	0	1	0	1	0	0	0	0	0	0	0
check2	0	0	0	1	0	1	0	0	0	0	0	0
check3	1	0	0	0	0	0	1	0	0	0	0	0
check4	0	1	0	0	0	0	0	1	0	0	0	0
check5	0	1	0	0	0	0	0	0	1	0	0	0
check6	0	0	1	0	0	0	0	0	0	1	0	0
check7	0	0	0	1	0	0	0	0	0	0	1	0
check8	1	0	0	0	0	0	0	0	0	0	0	1

(a)

(b)

Figure A.2: (a) H matrix of an LDPC code, and (b) factor graph representation of an LDPC code.

Table A.1: Structure of LDPC Codes used in Communication Standards

Standard	z	n_b	m_b	n	m
IEEE 802.11ad [48]	42	16	3, 4, 6, 8	672	126 to 336
IEEE 802.15.3c [49]	21	32	4, 8, 16	672	84 to 336
IEEE 802.11n [40]	27, 54, 81	24	4, 6, 8, 12	648 to 1944	108 to 972
IEEE 802.16e [41]	24, 28, ..., 96	24	4, 6, 8, 12	576 to 2304	96 to 1152
IEEE 802.3an [39]	64	32	6	2048	384

A.1.1 Structured LDPC codes

Almost all the latest LDPC codes used in practical applications have adopted a structured design, where the H matrix is constructed using permutation matrices. Permutation matrix is a square binary matrix that has exactly one 1 in each row and one 1 in each column and 0 in all the remaining entries. The structured H matrix greatly simplifies the decoder design. A further simplification of the code construction is to use identity matrix, its cyclic shifts and zero matrix to construct the H matrix.

Figure A.2(a) is a structured H matrix with m_b rows ($m_b = 2$) and n_b columns ($n_b = 3$) of $z \times z$ submatrices ($z = 4$). Each submatrix is an identity matrix, a cyclic shifted identity matrix, or a zero matrix. Table A.1 lists a number of the latest applications that have adopted LDPC codes whose H matrices are defined this way. Note that in most of the cases, a set of LDPC codes are defined that follow a similar structure with a fixed n_b, and a variable m_b to adjust the code rate, i.e., number of parity checks, and a variable z to adjust the code block length.

A.1.2 LDPC decoding

An LDPC code is commonly decoded by the belief propagation (BP) algorithm [50] following the factor graph [51] that defines the code. BP is an intrinsically parallel algorithm. It relies on passing messages between VNs and CNs following the edges of the factor graph. The decoding usually converges after a few iterations. A high throughput can be achieved by a parallel decoder implementation, and FPGA is an ideal platform for accelerating decoding.

To start BP decoding, the noise-corrupted channel output values are used to calculate the prior log-likelihood ratio (LLR), i.e., the probability of the bit being 1 over being 0. The VNs in the factor graph are initialized with the prior LLRs. The VNs will then send prior LLRs to their adjacent CNs. The message passing from VNs to CNs are usually ordered based on the H matrix using row processing followed by column processing.

Row processing is done row by row over the H matrix. In processing row i, the prior LLRs of all VN j's, where $H(i, j) = 1$, are passed to CN i. Each CN receives the LLR messages from the adjacent VNs and computes an extrinsic LLR for each VN. The extrinsic LLR represents the likelihood of a bit given the LLRs from all other bits that are part of the parity check. Row processing is done after m steps.

Column processing is done column by column over the H matrix. In processing column j, VN j sums up the extrinsic messages from all CN i's, where $H(i, j) = 1$, to compute the posterior LLR of VN j. A hard decision is made based on the posterior LLR. Column processing is done after n steps. Column processing can be interleaved with row processing by saving the extrinsic messages in VNs, so that VNs can accumulate the extrinsic LLRs on the fly without requiring a separate column processing step. Similarly, row processing can also be interleaved with column processing.

One decoding iteration consists of one row processing followed by one column processing, or one row processing with interleaved column processing. If the hard decisions after a decoding iteration satisfy all the parity checks, decoding is complete. A decoding iteration limit is often imposed to terminate decoding if it fails to converge.

A.2 Decoder emulation

To investigate the error-correcting performance of LDPC codes, it is necessary for the decoder emulation to achieve a high throughput. For example, a 100 Mb/s FPGA decoder allows us to measure the BER down to 10^{-10} in one day (assume collecting at least 1000 bit errors to estimate the BER reliably). To achieve a high throughput, the decoder needs to be parallelized on FPGA.

A.2.1 Decoder architectures

For structured LDPC codes, there are a number of options to parallelize the decoder design using row-parallel and column-parallel architectures. A row-parallel architecture implements row processing with interleaved column processing using n_b VN processing elements (PE) and 1 CN PE, so that n_b VN PEs can pass messages to 1 CN PE at the same time. The latency of one iteration in the row-parallel architecture is proportional to m. In a pipelined implementation, one iteration takes m clock cycles, so the throughput of a row-parallel decoder is approximately $\frac{nf}{mn_{it}}$, where f is the clock frequency and n_{it} is the number of decoding iterations. As an example, a row-parallel LDPC decoder for the rate-1/2 IEEE 802.11ad LDPC code [48] has a throughput of 20 Mb/s using a 100 MHz clock frequency and 10 decoding iterations.

A column-parallel architecture implements column processing with interleaved row processing using m_b CN PEs and 1 VN PE. Its latency is n clock cycles and the throughput is approximately $\frac{nf}{nn_{it}}$. The row- and column-parallel architectures can be further parallelized by processing multiple rows or multiple columns concurrently. For example, a z-row-parallel architecture processes z rows concurrently using $zn_b = n$ VN PEs and z CN PEs for a latency of m_b cycles per iteration. The throughput of this architecture is approximately $\frac{nf}{m_b n_{it}}$, which is z times faster than the row-parallel architecture.

VN and CN PEs are implemented using 4 to 8-bit fixed-point add, subtract, and compare, and the implementation cost is relatively low. However, for decoder emulation, the channel model is also needed to generate the inputs to the VN PEs, and it may become a bottleneck in highly parallel architectures.

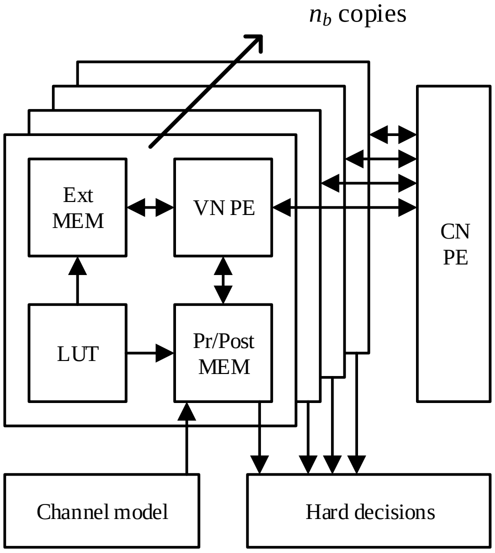


Figure A.3: A row-parallel LDPC decoder architecture.

A.2.2 FPGA implementation

The row-parallel architecture is shown in Figure A.3. In this design, n_b VN PEs pass messages to 1 CN PE for one row processing step. The CN computes extrinsic messages that are passed back to each VN PE for column processing. To complete one iteration of decoding, the m row processing steps are carried out serially. Each VN PE acts as the one of z VNs in each step. The VN PE carries a z-entry prior LLR memory with one entry per VN, a z-entry posterior LLR memory also with one entry per VN, and an m-entry extrinsic LLR memory to store the LLRs from m CNs. Each VN PE also uses an m-entry lookup table (LUT) to store the address of the extrinsic memory to read from and write to in each row processing step. The decoder designs are quantized to 4 to 8 bits, and both the arithmetic and the memory are quantized to 4 to 8 bits.

The decoder emulation is controlled based on the state machine shown in Figure A.4(a). The decoder starts in the idle state. After receiving a start signal, the state transitions to the load state, and each VN PE loads z prior LLRs over z clock

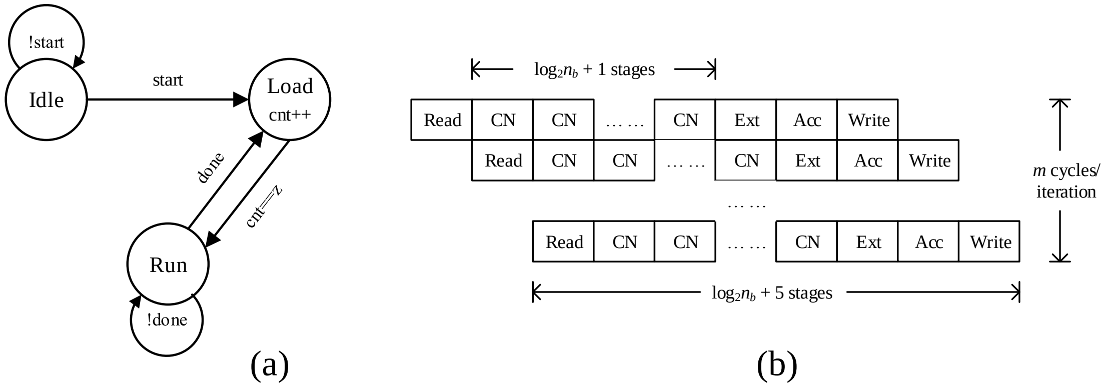


Figure A.4: (a) Emulation control, and (b) pipeline timing of a row-parallel decoder.

cycles. After loading is complete, the decoder transitions to the run state, an iteration counter counts from 1 to n_{it}. In each iteration, an address counter counts from 1 to m that tracks the m row processing steps. Hard decisions of the VNs are made at the end of each iteration. If all the parity checks are satisfied or if the iteration limit is reached, a done signal is generated to terminate the run state and transition to the load state to process the next input frame. The decoder will run continuously between the load and the run states.

A sample pipeline chart of a row-parallel decoder is shown in Figure A.4(b). In this design, each VN takes one cycle to read from memory and prepare LLR message to CN. CN uses a compare-select tree of $\log_2 n_b + 1$ stages to compute LLR messages to the VNs. The returned LLR messages are post-processed by the VNs in one cycle to calculate the extrinsic LLR, and then accumulated in one cycle to calculate the posterior LLR. In the final cycle, the posterior LLR is written back to memory. The pipeline latency is $\log_2 n_b + 5$ cycles. A complete decoding iteration in a row-parallel architecture takes $m + \log_2 n_b + 4$ cycles.

An additive white Gaussian noise (AWGN) channel model runs continuously to provide inputs to the decoder. The SNR is adjusted by scaling the standard deviation of the AWGN noise. To measure the BER and FER, a frame counter is used to track the number of input frames, a frame error counter and a bit error counter to record the number frame errors and bit errors, respectively. A frame count limit can be set, so that when the frame count reaches the limit, both the frame error counter and bit error counter will freeze to get a snapshot of the error counts.

A typical decoder emulation is done in the following sequence: 1) set the SNR and the frame count limit; 2) reset all counters and start emulation; and 3) after reaching the frame count limit, read the recorded bit error count and frame error count to calculate BER and FER.

A.2.3 FPGA design flow

To lower the design barrier and facilitate reuse, the decoder emulation platform can be parameterized to support all structured LDPC codes, such as the ones shown in Table A.1. Using a row-parallel architecture, the parameter n_b determines the number of VN PEs and the fan-in and fan-out of the CN PE. The parameters z and m_b determine the control schedule, and the memory size (including prior LLR memory, posterior LLR memory, and extrinsic LLR memory). The H matrix of the code determines the address LUT. These parameters are design-time configurable.

There are additional parameters that are important for code and decoder designs, including word length and quantization of the LLR messages, decoding iteration limit, channel SNR, and algorithmic knobs. These parameters can be divided into two categories: design-time configurable or run-time configurable. Word length and quantization are design-time configurable, and decoding iteration limit and channel SNR are run-time configurable.

In designing the emulation platform, it is important to build in the run-time configurability so that one design can be used for many different experiments. The design-time configurability can be facilitated by a systematic design flow.

The design flow consists of two steps: building a module library, and assembling the modules using a script. For example, an LDPC decoder library includes VN PE, CN PE, channel model (AWGN noise generator), controller, and error checker. Parameters such as z, m_b, address LUT entries, word length and quantization are coded as parameters in the modules. These modules are made as generic as possible to make it reusable.

The next step is to use a script to assemble the modules based on the target LDPC code. The script will perform the following tasks: 1) read the H matrix, build the address LUT, and initialize design-time configurable parameters, including z, m_b, address LUT, word length, and quantization; 2) instantiate n_b VN PEs, 1 CN PE, channel model, controller, and error checker, and connect them in a complete decoder; 3) insert interface registers for the run-time configurable parameters, including decoding iteration limit, channel SNR, and frame count limit, and for controlling the emulation platform, including start of emulation and reset of frame counter and error counter. The complete decoder is synthesized, and the bit file mapped on FPGA for emulation.

The emulation can be further automated by a script that polls the emulation status, frame count, error count, and sweeps the SNR and frame count limit to obtain the BER and FER points without manual intervention.

The library and script design flow simplifies the decoder design. Run-time configurability is built in the modules and design-time configurability is done by scripting. For example, to investigate a different word length and quantization, we can rerun the script with different parameter settings without the need of changing the underlying modules. The modules library can be used to build different LDPC decoders, e.g., those listed in Table A.1.

A.3 Study of LDPC error floors using emulation

Figure A.1 is the waterfall plot of an LDPC code captured by FPGA emulation. The code exhibits an error floor below BER of 10^{-8}. The goal of an error floor investigation is to answer three questions: 1) what is the cause of the error floor? 2) how do quantization and word length affect the error-correcting performance (as practical decoders are always quantized)? 3) how to lower or remove the error floor? The three questions can be answered through experiments using FPGA decoder emulation.

A.3.1 Capture and replay of rare events

The study of the LDPC error floors requires capturing of rare events that cause the decoder to fail. That is, if the error floor occurs at BER of 10^{-8}, we need to study the rare events that occur with a probability of 10^{-8} and below, which is within reach of a fast FPGA emulator. However, one difficulty with an FPGA emulator is that only low bandwidth outputs, e.g., bit error count and frame error count, are directly observable. These statistical measures are meaningful but they do not yield an in-depth understanding of the rare events that cause error floors.

A better approach is to capture the inputs that induce rare events through FPGA emulation, and reproduce the rare events by simulating the captured events in software. Recording the inputs on FPGA requires a large memory. For example, for a 2 Kb LDPC code, storing 6-bit soft inputs for a 2 Kb frame requires 12 Kb storage. A modern FPGA device contains more than 1 Mb on-chip memory, enough to store more than 100 frames of soft inputs. To capture these rare events, additional logic is added to the emulation platform to enable storing the input frame when decoding fails to converge after a sufficient number of iterations.

To reproduce the rare events in software, the captured inputs from FPGA emulation are replayed in a bit- and cycle-accurate simulation. The simulation of error events indicates that when an LDPC decoder fails to converge in the error floor region,

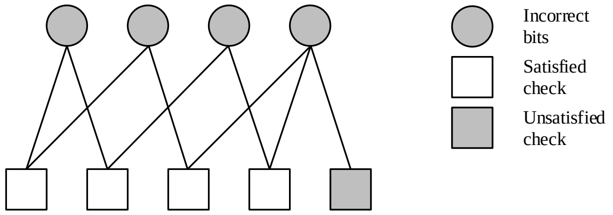


Figure A.5: Illustration of a (4,1) trapping set.

not all its parity checks are satisfied [24]. From the unsatisfied check nodes, a local minimum state called trapping set [24] can be traced. A trapping set is fundamentally caused by the cycles in the code structure. A (4, 1) trapping set in the rate-5/6 IEEE 802.11n code is shown in Figure A.5. Note that it is a subset of the factor graph of the LDPC code. Suppose all the bits in the trapping sets are initialized to the incorrect value, the incorrect bits will reinforce themselves in BP decoding. Once the decoder is trapped in this state, it will be unable to make any progress towards convergence, because the reinforcement from the incorrect bits is strong enough to overcome any attempt to correct the bits.

The weight of a trapping set is often lower than the minimum distance of the code, thus trapping set errors happen with a much higher probability than the minimum-distance errors. As a result, the waterfall curve bends, causing error floor. The weight, structure, and count of the trapping sets determine the location and slope of the error floor.

Using the design flow outlined previously, we can build different versions of the LDPC decoder for emulation using different word length and quantization. By comparing the error and trapping set profiles, it was discovered that a short word length makes trapping sets happen more frequently [52] due to numerical saturation. When messages saturate in BP decoding, the soft decoding degenerates to a decoding based on majority counting, which is more easily trapped.

The capture-and-replay approach makes it possible to narrow down on the rare error-floor events to find out the cause of the error floors and the word length and quantization effects on the error floors.

A.3.2 Emulation-based iterative investigation loop

New decoding algorithms can be designed to lower or remove the error floor. One such approach is called post-processing [25, 53]. The idea is to first decode using BP. If decoding fails to converge, it is most likely that the decoder has entered a trapping set. The trapping set can be perturbed, either by erasure [53] or noise injection [25] to make it unstable, and then the decoder will have a chance to reconverge to the global minimum.

The capture-and-replay method is essential in the investigation of the post-processing algorithm. FPGA emulation can be used to capture the inputs that cause error floors, and a software simulation is built to mimic the hardware operations and then modified to incorporate post-processing. Post-processing is then applied to the captured inputs in software simulation. The post-processing algorithm can be iteratively improved by adjusting the location and the amount of perturbation to achieve good performance.

After the improved algorithm is proven in software simulation using captured inputs, the improved algorithm needs to be put back in FPGA emulation for a vigorous test. Any residual errors will motivate another iteration of algorithm improvement and verification by FPGA emulation.

The iterative investigation loop allows the post-processing algorithm to be incrementally refined. An example of the BER and FER plot of the LDPC decoder for the rate-5/6 IEEE 802.11n LDPC code before and after post-processing is shown in Figure A.6 for comparison.

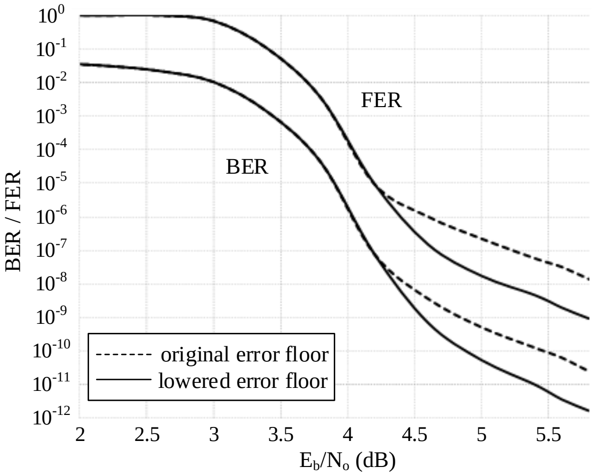


Figure A.6: Improved error-correcting performance by post-processing.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] E. Arikan, “Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels,” *IEEE Trans. Inf. Theory*, vol. 55, no. 7, pp. 3051–3073, Jul. 2009.

[2] B. Yuan and K. K. Parhi, “Architecture optimizations for BP polar decoders,” in *IEEE Int. Conf. Acoustics, Speech and Signal Process.*, May 2013, pp. 2654–2658.

[3] C. Leroux, I. Tal, A. Vardy, and W. J. Gross, “Hardware architectures for successive cancellation decoding of polar codes,” in *IEEE Int. Conf. Acoustics, Speech and Signal Process.*, May 2011, pp. 1665–1668.

[4] C. Zhang and K. Parhi, “Low-latency sequential and overlapped architectures for successive cancellation polar decoder,” *IEEE Trans. Signal Process.*, vol. 61, no. 10, pp. 2429–2441, May 2013.

[5] I. Tal and A. Vardy, “List decoding of polar codes,” *IEEE Trans. Inf. Theory*, vol. 61, no. 5, pp. 2213–2226, May 2015.

[6] U. U. Fayyaz and J. R. Barry, “Low-complexity soft-output decoding of polar codes,” *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 958–966, May 2014.

[7] K. Niu and K. Chen, “CRC-aided decoding of polar codes,” *IEEE Commun. Lett.*, vol. 16, no. 10, pp. 1668–1671, October 2012.

[8] Y. Wang and K. R. Narayanan, “Concatenations of polar codes with outer BCH codes and convolutional codes,” in *52nd Ann. Allerton Conf. Commun., Control, and Computing*, Sept 2014, pp. 813–819.

[9] J. Guo, M. Qin, A. Guillen i Fabregas, and P. H. Siegel, “Enhanced belief propagation decoding of polar codes through concatenation,” in *IEEE Int. Symp. Inform. Theory*, June 2014, pp. 2987–2991.

[10] C. Zhang, X. You, and J. Sha, “Hardware architecture for list successive cancellation polar decoder,” in *IEEE Int. Symp. Circuits and Syst.*, June 2014, pp. 209–212.

[11] J. Lin, C. Xiong, and Z. Yan, “Reduced complexity belief propagation decoders for polar codes,” in *IEEE Workshop Signal Process. Syst.*, Oct 2015, pp. 1–6.

[12] C. Leroux, A. J. Raymond, G. Sarkis, and W. J. Gross, "A semi-parallel successive-cancellation decoder for polar codes," *IEEE Trans. Signal Process.*, vol. 61, no. 2, pp. 289–299, Jan. 2013.

[13] A. Pamuk, "An FPGA implementation architecture for decoding of polar codes," in *Int. Symp. Wireless Commun. Syst.*, Nov. 2011, pp. 437–441.

[14] E. Arikan, "Systematic polar coding," *IEEE Commun. Lett.*, vol. 15, no. 8, pp. 860–862, Aug. 2011.

[15] T. Zhang and K. Parhi, "A 54 Mbps (3,6)-regular FPGA LDPC decoder," in *IEEE Workshop Signal Process. Syst.*, Oct 2002, pp. 127–132.

[16] Y. Chen and D. Hocevar, "A FPGA and ASIC implementation of rate 1/2, 8088-b irregular low density parity check decoder," in *IEEE Global Telecommun. Conf.*, vol. 1, Dec 2003, pp. 113–117.

[17] L. Sun, H. Song, Z. Keirn, and B. Kumar, "Field programmable gate array (FPGA) for iterative code evaluation," *IEEE Trans. Magn.*, vol. 42, no. 2, pp. 226–231, Feb 2006.

[18] Z. Zhang, L. Dolecek, B. Nikolic, V. Anantharam, and M. Wainwright, "Investigation of error floors of structured low-density parity-check codes by hardware emulation," in *IEEE Global Telecommun. Conf.*, Nov 2006.

[19] X. Chen, J. Kang, S. Lin, and V. Akella, "Accelerating FPGA-based emulation of quasi-cyclic LDPC codes with vector processing," in *Conf. Design, Autom. Test in Europe*, 2009, pp. 1530–1535.

[20] D. Chang, F. Yu, Z. Xiao, Y. Li, N. Stojanovic, C. Xie, X. Shi, X. Xu, and Q. Xiong, "FPGA verification of a single QC-LDPC code for 100 Gb/s optical systems without error floor down to BER of 10^{-15}," in *Optical Fiber Commun. Conf.*, 2011.

[21] H. Li, Y. S. Park, and Z. Zhang, "Reconfigurable architecture and automated design flow for rapid FPGA-based LDPC code emulation," in *ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2012, pp. 167–170.

[22] C. Chang, K. Kuusilinna, B. Richards, A. Chen, N. Chan, R. Brodersen, and B. Nikolic, "Rapid design and analysis of communication systems using the BEE hardware emulation environment," in *IEEE Int. Workshop Rapid Syst. Prototyping*, June 2003, pp. 148–154.

[23] J. Liang, R. Tessier, and D. Goeckel, "A dynamically-reconfigurable, power-efficient turbo decoder," in *IEEE Symp. Field-Programmable Custom Comput. Mach.*, April 2004, pp. 91–100.

[24] T. Richardson, "Error floors of LDPC codes," in *Allerton Conf. Commun. Control Comput.*, vol. 41, no. 3, 2003, pp. 1426–1435.

[25] Z. Zhang, L. Dolecek, B. Nikolic, V. Anantharam, and M. Wainwright, "Lowering LDPC error floors by postprocessing," in *IEEE Global Telecommun. Conf.*, Nov 2008.

[26] R. Mori and T. Tanaka, "Performance and construction of polar codes on symmetric binary-input memoryless channels," in *2009 IEEE Int. Symp. Inf. Theory*, June 2009, pp. 1496–1500.

[27] ——, "Performance of polar codes with the construction using density evolution," *IEEE Commun. Lett.*, vol. 13, no. 7, pp. 519–521, July 2009.

[28] I. Tal and A. Vardy, "How to construct polar codes," *IEEE Trans. Inf. Theory*, vol. 59, no. 10, pp. 6562–6582, Oct 2013.

[29] P. Trifonov, "Efficient design and decoding of polar codes," *IEEE Trans. Commun.*, vol. 60, no. 11, pp. 3221–3227, November 2012.

[30] C. Xiong, Y. Zhong, C. Zhang, and Z. Yan, "An fpga emulation platform for polar codes," in *2016 IEEE Int. Workshop Signal Process. Syst. (SiPS)*, Oct 2016, pp. 148–153.

[31] J. Wuthrich, A. Balatsoukas-Stimming, and A. Burg, "An fpga-based accelerator for rapid simulation of SC decoding of polar codes," in *2015 IEEE Int. Conf. Electron., Circuits, and Syst. (ICECS)*, Dec 2015, pp. 633–636.

[32] B. Yuan and K. K. Parhi, "Early stopping criteria for energy-efficient low-latency belief-propagation polar code decoders," *IEEE Trans. Signal Process.*, vol. 62, no. 24, pp. 6496–6506, Dec 2014.

[33] I. Tal and A. Vardy, "List decoding of polar codes," in *IEEE Int. Symp. Inf. Theory*, July 2011, pp. 1–5.

[34] S. Sun and Z. Zhang, "Architecture and optimization of high-throughput belief propagation decoding of polar codes," in *2016 IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2016, pp. 165–168.

[35] P. Urard, L. Paumier, V. Heinrich, N. Raina, and N. Chawla, "A 360mW 105Mb/s DVB-S2 compliant codec based on 64800b LDPC and BCH codes enabling satellite-transmission portable devices," in *2008 IEEE Int. Solid-State Circuits Conf. - Dig. Tech. Papers*, Feb 2008, pp. 310–311.

[36] C. Roth, P. Meinerzhagen, C. Studer, and A. Burg, "A 15.8 pJ/bit/iter quasi-cyclic LDPC decoder for IEEE 802.11n in 90 nm CMOS," in *2010 IEEE Asian Solid-State Circuits Conf.*, Nov 2010, pp. 1–4.

[37] F. Naessens, V. Derudder, H. Cappelle, L. Hollevoet, P. Raghavan, M. Desmet, A. M. AbdelHamid, I. Vos, L. Folens, S. O'Loughlin, S. Singirikonda, S. Dupont, J. W. Weijers, A. Dejonghe, and L. V. der Perre, "A 10.37 mm2 675 mW reconfigurable LDPC and Turbo encoder and decoder for 802.11n, 802.16e and 3GPP-LTE," in *Symp. VLSI Circuits*, June 2010, pp. 213–214.

[38] Y. S. Park, Y. Tao, S. Sun, and Z. Zhang, "A 4.68Gb/s belief propagation polar decoder with bit-splitting register file," in *Symp. VLSI Circuits*, Jun. 2014, pp. 1–2.

[39] "IEEE standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications," *IEEE Std 802.3an-2006*, 2006.

[40] "IEEE standard for information technology– local and metropolitan area networks– specific requirements– part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment 5: Enhancements for higher throughput," *IEEE Std 802.11n-2009*, Oct 2009.

[41] "IEEE standard for local and metropolitan area networks part 16: Air interface for fixed and mobile broadband wireless access systems amendment 2: Physical and medium access control layers for combined fixed and mobile operation in licensed bands and corrigendum 1," *IEEE Std 802.16e-2005 and IEEE Std 802.16-2004/Cor 1-2005*, 2006.

[42] A. Morello and V. Mignone, "DVB-S2: The second generation standard for satellite broad-band services," *Proc. IEEE*, vol. 94, no. 1, pp. 210–227, Jan 2006.

[43] H. Song and J. Cruz, "Reduced-complexity decoding of Q-ary LDPC codes for magnetic recording," *IEEE Trans. Magn.*, vol. 39, no. 2, pp. 1081–1087, Mar 2003.

[44] S. Tanakamaru, Y. Yanagihara, and K. Takeuchi, "Over-10×-extended-lifetime 76%-reduced-error solid-state drives (SSDs) with error-prediction LDPC architecture and error-recovery scheme," in *IEEE Int. Solid-State Circuits Conf.*, Feb 2012, pp. 424–426.

[45] T. Richardson, M. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 619–637, Feb 2001.

[46] R. Gallager, "Low-density parity-check codes," *IRE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, January 1962.

[47] D. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inf. Theory*, vol. 45, no. 2, pp. 399–431, Mar 1999.

[48] "IEEE standard for information technology–telecommunications and information exchange between systems–local and metropolitan area networks–specific requirements-part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment 3: Enhancements for very high throughput in the 60 GHz band," *IEEE Std 802.11ad-2012*, Dec 2012.

[49] "IEEE standard for information technology - telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements. part 15.3: Wireless medium access control (MAC) and physical layer (PHY) specifications for high rate wireless personal area networks (WPANs) amendment 2: Millimeter-wave-based alternative physical layer extension," *IEEE Std 802.15.3c-2009*, Oct 2009.

[50] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan Kaufmann, 1988.

[51] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 498–519, Feb 2001.

[52] Z. Zhang, L. Dolecek, B. Nikolic, V. Anantharam, and M. Wainwright, "Design of LDPC decoders for improved low error rate performance: quantization and algorithm choices," *IEEE Trans. Commun.*, vol. 57, no. 11, pp. 3258–3268, Nov 2009.

[53] Y. Han and W. Ryan, "Low-floor decoders for LDPC codes," *IEEE Trans. Commun.*, vol. 57, no. 6, pp. 1663–1673, June 2009.