Evaluation of flexible SPA based LDPC decoder using hardware friendly approximation methods

DEEPAK YADAV
AFSHIN SERAJ
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Evaluation of flexible SPA based LDPC decoder using hardware friendly approximation methods

Deepak Yadav
deepak.yadav.943@student.lu.se
Afshin Seraj
afshin.seraj.877@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Erik Ledfelt (Ericsson)
Alberth Arvidsson (Ericsson)
Magnus Malmberg (Ericsson)
Liang Liu (LTH)

Examiner: Erik Larsson

September 22, 2017

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Low Density Parity Check (LDPC) coding has recently become a hot topic for its near-shannon performance, and will be used in the next generation of telecommunication systems (5G). the original algorithm for decoding LDPC codes is Sum-Product Algorithm (SPA). As the frequently-used Min-Sum Algorithm (MSA) is an overestimation of SPA, more accurate approximations of SPA are demanded in high-accuracy applications. This thesis studies the different ways of approximating the SPA and evaluates the cost and accuracy of an SPA decoder based on an optimum approximation. In a general comparison between main approximation methods, Simple Piecewise Linear (SPWL) approximation showed the most area-efficiency. After studying the different mathematical formats of SPA, the Soft-XOR based format with forward-backward scheme was found to be the most hardware-efficient one. Its core function (Soft-XOR) was implemented with Centered Recursive Interpolation (CRI) approximation, which achieved the highest efficiency, compare to other approximations. A Check Node Unit (CNU), which is the main computational part of LDPC decoders, was implemented based on the CRI-based Soft-XOR. The CNU uses a pipe-lined forward-backward architecture, and its speed and area are flexible and can be changed in instantiation. A SPA decoder with flooding schedule based on the developed CNU is estimated to have an area of 1.6M as equivalent gate count, with a clocking frequency of 1.25GHz. The accuracy loss, compare to floating point SPA, is 0.3dB for 10 iterations and a throughput of 10Gb/s. The accuracy loss becomes less than 0.1dB for 20 iterations with a throughput of 5Gb/s and the same area. Comparing MSA and SPA, the developed SPA CNU is 2.1 times larger than the developed MSA CNU, but gains 0.3dB more accuracy for 10 iterations. The accuracy gain increases with higher number of iterations. A comparison with other works is also provided. The IEEE 802.11n Wi-Fi standard is used for the decoder and the 18nm CMOS technology is used for synthesis.

Acknowledgment

We would like to express our gratitude towards our thesis supervisors Erik Ledfelt, Albert Arvidsson and Magnus Malmberg at Ericsson for their continuous guidance and support. We are also thankful to Charlotte Sköld for providing us with this opportunity to learn. Special thanks to all the Ericsson ASIC IP1 team members with whom we have had great time.
We would also like to thank our academic supervisior Liang Liu at LTH.

Popular Science Summary

You have to get lost before you can be found, a quote by Jeff Rasley goes very well for Low Density Parity Check (LDPC) codes. First invented by Gallager in 1962 but kind of lost during the journey of evolution of telecommunication networks because of their high complexity and demanding computations, which technology was not so advanced to handle, at that time. However, during late 1990s, success of turbo codes invoked the re-discovery of codes. Recently it has attracted tremendous research interest among the scientific community, as today's technology is advanced enough and to make LDPC decoders completely commercial. In a wireless network, the information is not just simply sent, but first encoded. In a sense, all the transmitted bits are tied together, according to some mathematical rules. Therefore, if noise destructs parts of the information while traveling, the LDPC decoder at the receiver side, can automatically detect and retrieve those parts, based on the other parts of the code. Here, our main focus is on the decoder. For actual hardware implementation of the decoder, some level of approximation of the ideal algorithm is always necessary, which reduces the accuracy depending on the approximation.
Ericsson is developing the next-generation wireless network for 5G, and already possesses a form of "Min-Sum" approximation of the LDPC decoder. As the current requirements demand more accurate decoders, the goal of this thesis is to find and evaluate a more accurate but more costly version of LDPC decoder, with more flexibility. Thus, several promising approximation methods were selected and evaluated based on their complexity, cost, and their accuracy towards error correction. After performing several trade-offs, an approximation method is chosen and the cost of a LDPC decoder using that approximation, is derived. With this acquired data, a trade-off between accuracy and cost can be made, depending on the application.

Table of Contents

List of Figures

List of Tables

Acronyms

BER Bit Error Rate.

BP Belief Propagation.

BPSK Binary Phase-Shift Keying.

CN Check Node.

CNU Check Node Unit.

CRI Centered Recursive Interpolation.

LDPC Low Density Parity Check.

LLR Log-likelihood Ratio.

MSA Min-Sum Algorithm.

NOF_BITS Number of bits.

RAM Random Access Memory.

ROM Read Only Memory.

SNR Signal-to-Noise Ratio.

SPA Sum-Product Algorithm.

SPWL Simple Piecewise Linear.

VN Variable Node.

Chapter 1
Introduction

1.1 Background

Low Density Parity Check (LDPC) coding technique has regained a lot of attention, due to its capability to reach Shannon's limits, i.e. transmitting maximum possible amount of data with minimum power, and therefore, it will be used for the next generation of telecommunication systems (5G). The theoretical algorithm that runs very successfully on LDPC decoders, is Sum-Product Algorithm (SPA). However, the hardware implementation of a pure-SPA algorithm requires implementing trigonometrical functions, such as *tanh*, which make the hardware unfeasibly expensive. Therefore, lots of research has been done to estimate the behavior of pure-SPA, in both algorithmic level, and hardware level. On algorithmic level, the Min-Sum Algorithm (MSA) approximation, has reduced the hardware size considerably, by introducing a SNR degradation of 0.8dB, compare to SPA [1],[2]. To compensate for the over-estimation of MSA, two successful branches of MSA, known as "Offset Min-Sum(OMS)" and "Normalized Min-Sum (NMS)", have reduced the SNR degradation to the range of 0.2dB for NMS and 0.5dB for OMS, by introducing slightly more cost and complexity [3],[4]. The complexity comes from the fact that OMS/NMS require a channel estimation to tune their scaling/offset factors, based on the channel's noise. Also, the channel mismatch effect induces a higher error floor for NMS [1]. However, these two algorithms have found extensive commercial use. On the hardware level, also, there exists a variety of hardware-friendly approximations of SPA, and the mathematical functions involved in its CNU (Check-Node Unit), that is the main computational part [5],[6],[7]. These approximations put LDPC decoders on a spectrum, ranging from the most accurate one, to the least costly one, that is "Min-Sum". This thesis investigates high-precision LDPC decoders from a hardware perspective, and its goals are as follows:

1.2 Goals

1. Exploring different hardware-based approximation methods.
2. Finding the best approximation of SPA, from a hardware perspective.
3. Developing a flexible CNU in RTL, based on the chosen approximation.

4. Evaluating the cost and precision of a total high-precision LDPC decoder, based on the developed CNU, and assesses if the increase in precision justifies the cost.

1.3 Structure

Chapter 2 is dedicated to the investigation of different approximation methods and their comparison in terms of precision and hardware-cost. In chapter 3, we will cover the theoretical part of LDPC decoder, compare its different mathematical formulations, and select a hardware-efficient one for implementation. chapter 4 is specified to the core function of the LDPC decoder. In this chapter, a number of its different implementations with different approximations will be described and assessed in RTL level, and the best one will be selected. In chapter 5, there is an investigation of the BER/SNR behavior of the simulated SPA with different quantizations (number of bits) and number of iterations. The BER/SNR performance of MSA is also provided to compare. The chapter concludes with a brief explanation for choosing the fixed-point number representation for the system. Chapter 6 describes the RTL (system-verilog) implementation of a flexible pipelined forward-backward Check Node Unit (CNU) with the selected core function. The number of inputs that the developed CNU accepts each cycle, can be adjusted by changing a variable in the code, which can provide the opportunity to easily tune the speed, cost and latency (critical path), before fabrication. Also, in this chapter a flooding LDPC decoder architecture, based on the developed CNU is suggested and its cost is estimated. Synthesis results and conclusion are provided in chapters 7 and 8, respectively.

Chapter 2

Approximation Methods

In this chapter, several popular approximation methods will be focused on, and will be compared to each other in terms of precision and estimated hardware cost. As case study, the Hyperbolic tangent (tanh) function has been chosen. Hyperbolic tangent function is among the most frequently used functions, and has applications in LDPC decoders, as will be discussed in section 3.4. Moreover, its common, yet challenging, shape provides a suitable framework to investigate and fairly compare the approximation methods. In this chapter, after an introduction to the function, 7 candidate approximation methods are briefly introduced, among which 5 are simulated in C++ to assess their precision versus area consumption. Methods used for rough estimation of area at algorithm level (C++), will be explained later.

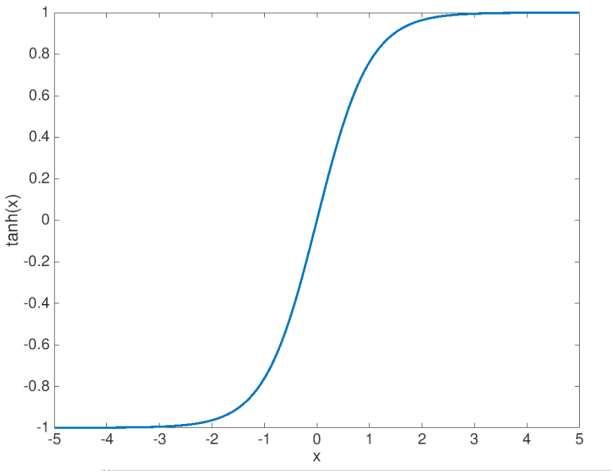


Figure 2.1: Hyperbolic Tangent Function

2.1 Candidate Approximation Methods

The Tanh() function is shown in Figure 2.1. Two properties of this function can be exploited to make the approximating hardware more efficient. One is that

tanh(x) is almost constant for $x < -4$ and $x > 4$, and another one is that tanh() is an odd function $tanh(-x) = -tanh(x)$. Thus, the range to be considered for approximation can be limited to $0 < x < 4$.

2.1.1 Isosceles Triangular Approximation

The derivative of the tanh() function resembles an isosceles Triangle that can be estimated as Equation 2.1 . Integrating this equation, an estimation of tanh() function is achieved in Equation 2.2 [8]. Computation of this equation needs one multiplier, an adder and a shifter. This method is proven to have worse area-precision characteristic than LUT in [9] and is therefore not implemented in this work.

$$tanh = \begin{cases} 1 - \frac{|x|}{2} & 0 \leq x \leq 2 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

$$tanh^{'} = \begin{cases} x - 0.25 * sign(x) * x^2 & 0 \leq |x| \leq 2 \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

2.1.2 Look-up Table (LUT)

One of the most used method of approximation is selecting a few points in the curve and storing them in a LUT. A LUT is simply a mapping of each subrange of inputs to a certain output. Therefore, the maximum error occurs in the middle of each subrange, and, of course, more points result in better accuracy. A typical improvement is storing the mean amount of output in each subrange, and reducing the error by half [10] (which is also used in this work). LUTs are constructed in two different ways. The straight forward one is storing the selected outputs in a ROM. 2^i outputs must be stored, where "i" is the number of bits in the input. A decoder maps the inputs to their corresponding outputs in ROM. The other method is called "bit-level mapping" and is simply mapping the inputs to outputs with purely combinational logic. This enables the synthesizer to optimize away some logic, after the input/output pattern is known [9].
Since in this comparison C++ is used to estimate area, ROM-based method is considered for comparison, because its area can easily be estimated with mathematical expressions. In both methods, LUT area can be much smaller, if selected inputs are equally distributed over the total range, which makes the decoder much smaller. Such equally spaced inputs also provide excellent framework for LUT to be combined with other methods. Here, two combinational methods, named SPWL and LUT+CRI, enormously benefit from such a setting, as will be shown later. However, if inputs are not equally spaced, we can concentrate more inputs where more change in the function occurs, and reduce the number of required stored points. Such method is called RALUT (Range Addressable Look-Up Table) and is discussed in [9]. Other methods in between these two extremes of LUT and RALUT also can be used to reduce the area of decoder and yet benefit from higher accuracy per number of stored points. For instance, In [11], the curve is

divided into 3 subregions in a RALUT fashion and then, each of this subregions is divided into 8 equal subregions in a LUT fashion.

2.1.3 Piece-wise Non-linear Approximation (PWNL)

The total range $0 < x < 4$ can be divided into N sub-regions, and each sub-region is approximated with a second order polynomial $(ax^2 + bx + c)$ and the coefficients (a,b and c) are stored in LUTs (Look-Up Table). This method is accurate but needs three multiplication, and consequently, a relatively higher area/delay compare to other methods.

2.1.4 Coefficient-based Piece-wise Linear Approximation (CPWL)

As shown in Figure 2.2, PWL is Similar to the previous method, but each sub-region is approximated with a line (first-order equation).

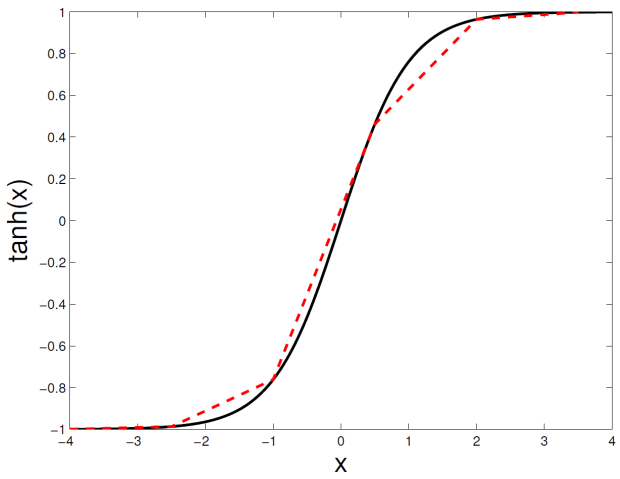


Figure 2.2: PWL concept [9]

There are two methods for PWL hardware implementation. The first is CPWL which includes storing coefficients "a" and "b" in equation: $(ax + b)$ in LUT, and the computation will be based on these two values. The disadvantage is having to use a multiplier, which can be expensive, and is therefore not implemented here. The second method [12] is here called SPWL (Simple Piece-wise Linear Approximation) and can be considerably more area efficient. It is described in 2.1.5.

2.1.5 Simple Piecewise Linear (SPWL)

Instead of storing the coefficients, a number of points in the curve can be chosen to be stored in a LUT, and computation of Equation 2.3 can be based on them.

$$y = \frac{y_h - y_l}{x_h - x_l}(x - x_l) + y_l \tag{2.3}$$

To improve precision, instead of approximating the function with a straight line between the two end points (y_h, x_h) and (y_l, x_l), one can use another straight line that minimizes the error, possibly by half, without loosing any area [12][13]. In other words, instead of saving y_h and y_l as $tanh(x_h)$ and $tanh(x_l)$ respectively, slightly modified values can be saved in ROM. This approach is not taken in our C++ implementation.

At first sight, a divider and a multiplier are needed to compute Equation 2.3. However, if the points are equally distributed on X axis, as discussed in 2.1.2, $x_h - x_l$ is a known number which makes the devision trivial. Also, if $x_h - x_l$ is a power of 2, the devision is a simple shifter and the multiplication can be simplified. In this case, $x - x_l$ can actually be a few LSB bits of the input. If the number of the input bits is I and the number of subregions is N, then the last M bits (LSB bits) of the input are chosen as the multiplicand $(x - x_l)$, where M is derived from Equation 2.4.

$$M = I - log_2(N) \tag{2.4}$$

Following this pattern in our C++ implementation, we sweep I, N and O(number of output bits) and by obtaining their corresponding precision and area consumption, we are able to choose an optimum value for these three variables.

2.1.6 Centered Recursive Interpolation (CRI)

CRI (Centered Recursive Interpolation) is a recursive algorithm that estimates the function after a known number of clock cycles. The theory is specified in [14] and used to approximate the sigmoid function in [15]. First the curve under interest is initially estimated with a few lines, all tangent to the curve (here, 2-line case is studied, g=x and g=1, which do not need any computation to be calculated). An optimum choosing of *delta* gives best precision. The code and the figures showing the approximation in each step, are shown in Figure 2.3. The primary advantage of CRI is absent of any multiplier and memory. It is verified that accuracy of CRI does not improve much for "q" more than 4. In the example of Figure 2.3, q=2 is chosen. As shown, the precision improves with clock cycle.

From the hardware perspective, the iterations could be performed all in one clock cycle, or each in one cycle. The disadvantage of the latter is more latency and complexity due to added registers to the design, while the former suffers from higher area consumption. Here, we consider the former alternative for our later analysis.

2.1.7 SPWL+CRI

To get better precision than SPWL, we presented a combined method of SPWL and CRI. In this method, other than fetching $y(x_1) = y_l$ and $y(x_2) = y_h$ from ROM, as in SPWL, $y(x_0)$ and $y(x_3)$ are also fetched. Therefore, similar to SPWL, enough data is available to derive the two initial lines for CRI, according to Equation 2.5.

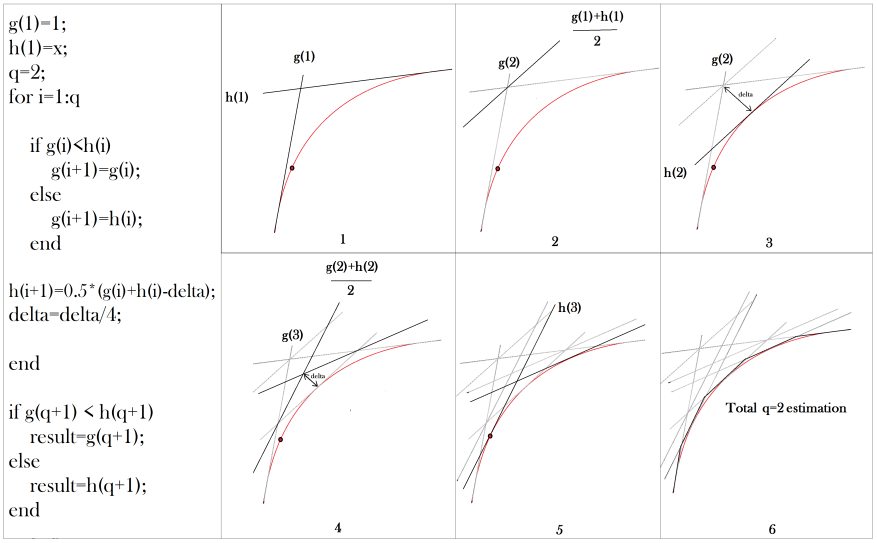


Figure 2.3: Algorithm description of CRI

$$y(1) = \frac{y(x_3) - y(x_2)}{x_2 - x_1}(x - x_2) + y_2 \quad y(2) = \frac{y(x_1) - y(x_0)}{x_2 - x_1}(x - x_1) + y_1 \qquad (2.5)$$

Therefore, two simple multipliers and a CRI computation is added to gain more precision.

2.2 Behavioral Model

A behavioral model is realized in C++ in order to implement the above algorithms in fixed point number representation. Figure 2.4 describes how the model works. A parametrized fixed point class is written in C++ in order to convert input data into fixed point. In Figure 2.4, *Fixedpoint* block takes input data and it requires number of integer bits and fraction bits as an input from user. The output from this block will be fixed point data based on given integer and fraction bits. In *Algorithm* block, a particular algorithm is selected to produce an approximation of *tanh* function.

A reference model of *tanh* function is generated inside reference block by using same input data. Output from reference and algorithm block are compared to get precision. Area calculation are done based on area equation described in below section. Area versus precision curves was plotted in order to compare accuracy of different method.

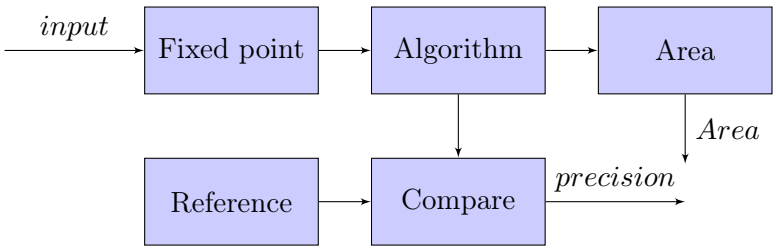


Figure 2.4: Block diagram for behavioral model

2.3 Result

The results of the 5 implemented algorithms are presented below. The unit for area is the number of full adders(FA), which when multiplied by 28 (the number of transistors in a FA), gives the number of transistors used in the design. The Area estimation is based on the Table 2.1.

Table 2.1: Estimation of area based on simple logic blocks (the unit is a FA or 28 transistors)

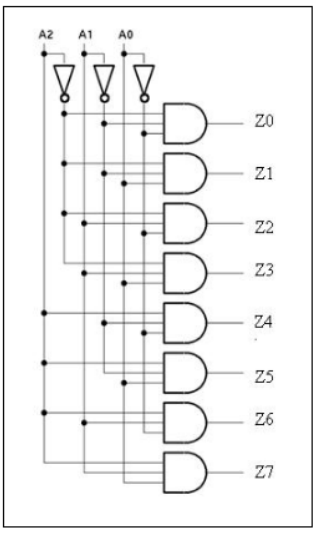
Logic	Area	description
Adder	N	N= no. of bits for inputs
Multiplier	$N * M$	N= no. of bits for input 1;M=no. of bits for input 2
Comparator	$0.64N$	N= no. of bits for inputs
And/OR gate	$(N-1) * 0.21$	N= no. of inputs
Decoder	$2^N * (N-1) * 0.21$	N= no. of inputs
ROM	$(M * N)/(28 * 2)$	M= Length; N=Width

The area of the comparator(Figure 2.5) and AND gate in Table 2.1 are calculated in Equation 2.7 and Equation 2.6, respectively. The area of the decoder(Figure 2.5), which will be used in 2.3.2, is calculated in Equation 2.8. In th table, Inverters in *decoder* are omitted and ROM is without *decoder*.

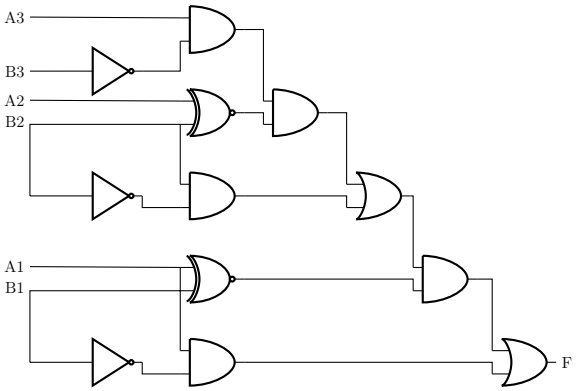
$$\begin{aligned} area(AND) &= \frac{6 * N}{28} \\ &= 0.21N \end{aligned} \tag{2.6}$$

$$\begin{aligned} area(Comparator) &= \frac{NOT(2 * N) + AND(6 * (2N-1)) + XOR(4N) + or(6 * (N-1))}{28} \\ &= 0.64N \end{aligned} \tag{2.7}$$

$$area(Decoder) = \frac{6 * (2^i * (i-1))}{28} \tag{2.8}$$



(a) Comparator



(b) Decoder

Figure 2.5: Comparator and Decoder circuits

2.3.1 Piecewise non-linear approximation

The approximated area for 3 multiplications and 2 additions is calculated by Equation 2.9. The area for the LUT for storing the coefficients is omitted, as it is much smaller compare to the rest of the design.

$$area = i^2 + 3ij + 4i + 2j \tag{2.9}$$

where, i= number of input bits and j=number of coefficient bits.

As mentioned in the above section, we will get different precision and area with different number of sub regions. So in order to find out good number of sub regions, the whole range is divided into different number of sub regions and for each case, area and precision was calculated. Figure 2.6 shows the area versus precision plot for different number of sub regions or curves. Area was calculated by sweeping number of fraction bits for input as well as coefficients while keeping output bits the same. It can be seen from the plot that precision is improved with increase in number of subregions. After analyzing the result, six number of curves was selected for further optimization, i.e. when $(i \neq j)$.

In order to find out best combination of input, coefficient and output bits, a framework was made that will select different combination of these three and produce error and area for each case. Figure 2.7 shows results from above framework, for 6 curves. In the plot each data point represents different combination of bits for each of three. From Figure 2.7 minimum precision and minimum area combination needs to be selected as final result. Figure 2.8 shows best data points

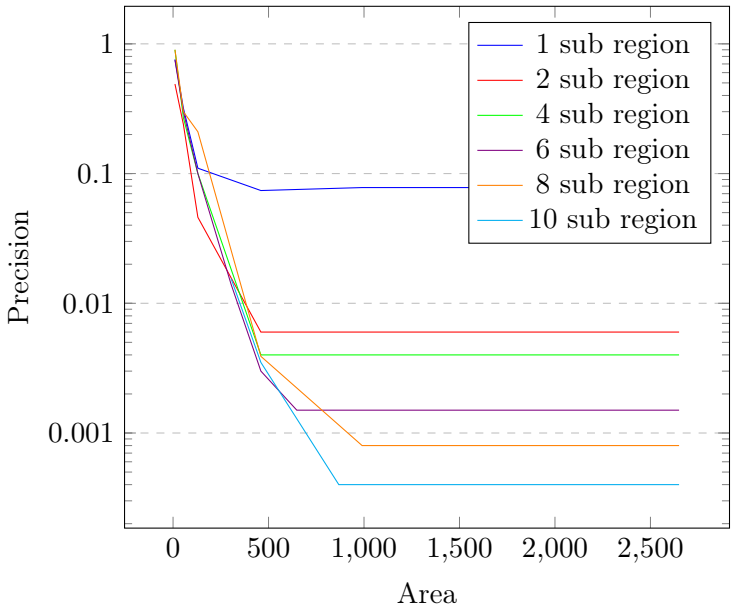


Figure 2.6: Area vs Precision for different sub regions

selected from Figure 2.7.

2.3.2 LUT

Equation 2.10 is used to calculate the area for this method. The first term computes the area of decoder, and the second term is the area of the ROM (see Table 2.1).

$$area = \frac{6 * r * (\log_2 r - 1) + (r - 1) * o}{28} \tag{2.10}$$

where o=number of output bits and r=number of regions
Figure 2.9 shows a curve for area and precision. The figure was plotted by sweeping output bits and number of regions, and selecting the best combinations. From figure, it can be seen that as we increase number of regions which means more number of LUTs, we get better precision but also bigger area.

2.3.3 Simple piecewise linear approximation

Implementation of SPWL approximation requires a multiplier and a look-up table. According to theory (2.1.5), a part of the input bits goes to LUT and the rest (a few LSB bits) is the multiplicand, according to Equation 2.4. Area of SPWL is calculated as Equation 2.11, that is the area of the LUT added to the area of the multiplier (the last term).

$$area = \frac{6 * r * (\log_2 r - 1) + (r - 1) * o}{28} + o * (i - \log_2 r) \tag{2.11}$$

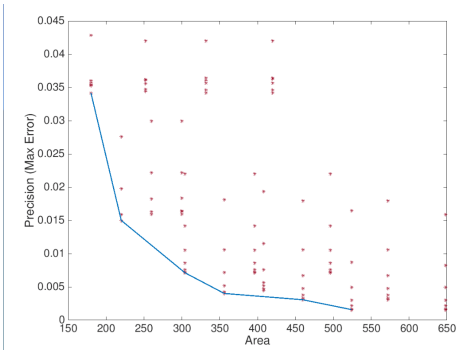


Figure 2.7: Area vs precision for different combinations in 6-curve PWNL

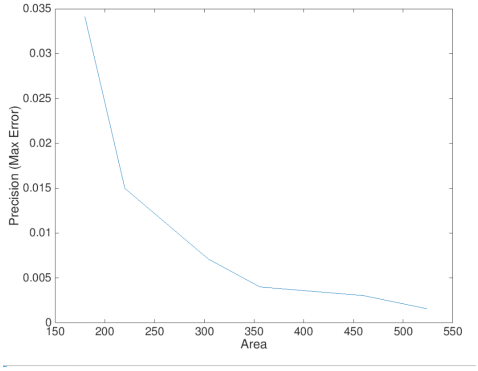


Figure 2.8: Area vs Precision for best combinations (6-curve PWNL)

where i=number of input bits, o=number of output bits and r=number of regions By sweeping the above three variables in the model and choosing the optimum combinations, we calculated the corresponding area. Figure 2.10 shows the variation in precision with respect to area. The minimum of all these curves will be derived and considered for our total comparison.

2.3.4 CRI

CRI algorithm requires only adders, comparators and shifters. To calculate the approximation area, Table 2.1 is used.

$$area = n*(2*adders+1*comparator)+1*comparator = n*(2.7*i)+0.6 \quad (2.12)$$

where i=number of input bits, n=number of interpolation
By sweeping input bits as well as number of interpolation, Figure 2.11 was plotted.

2.3.5 SPWL+CRI

From section 2.3.3, we observed that SPWL is giving best precision. To improve it even further, some computation was added to SPWL. Equation 2.13 was used to calculate its area. Compare to Equation 2.11, the area of ROM is twice (two values are stored for each entry, which are tanh() and delta), as well as that of multiplier (two multipliers are used). The last term denotes the CRI area. We swept the same variables as in the SPWL case, and by selecting the optimum combinations, plotted the area-precision curve in Figure 2.12 and Figure 2.13.

$$area = \frac{6*r*(\log_2 r-1)+r*o}{28} + 2*o*(i-\log_2 r)+9*o \quad (2.13)$$

where i=number of input bits, o=number of output bits and r=number of regions

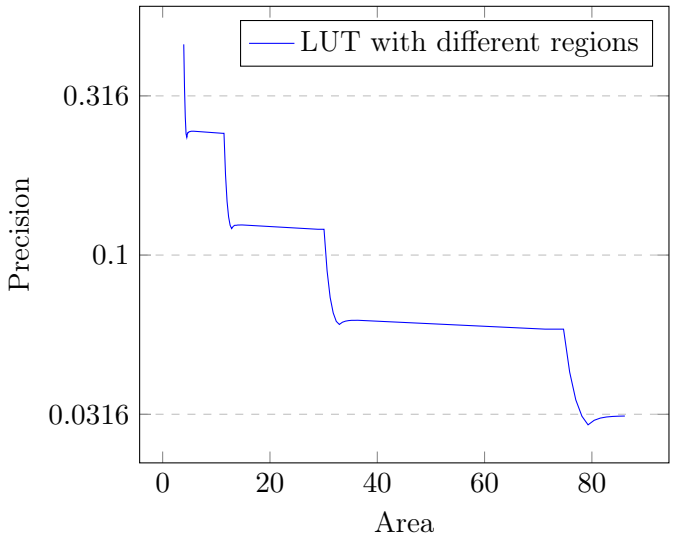


Figure 2.9: Area vs Precision

2.4 Conclusion

Figure 2.12 shows the behavior of the five implemented methods. It is apparent that, for less number of regions (and consequently less area and error) SPWL is outperforming others but as we go for more regions for instance 256 or 512, SPWL+CRI showed better performance. Figure 2.13 shows this trend. CRI saturates very soon and LUT's size increases dramatically as better precisions are required. For PWNL, the area consumption is the highest and it shows the worst performance, when small area is concerned.

One conclusion that can be taken from this figure, is that LUT alone is not the best solution, especially if precisions better than 0.01 are required. This is because the area of the decoder increases exponentially (Equation 2.10) . Therefore, LUT can work well as a primary course approximator, and adding a computational method to its results is highly beneficial. Heavier computations pay off for better precisions. This pattern is seen in Figure 2.13, noting that SPWL+CRI has more computational power compare to SPWL. With this pattern in mind, it is possible that for even better precisions than 0.000001, the PWNL becomes the best alternative among these 5 methods, as it has the most computational power.

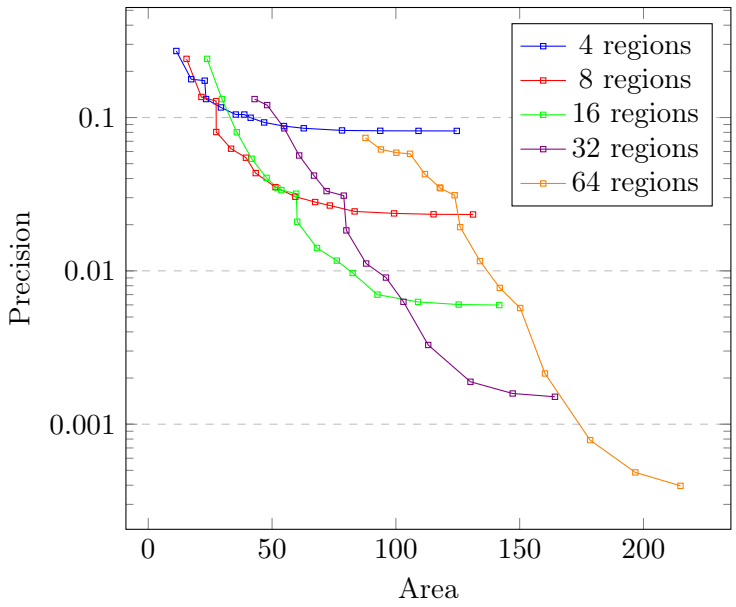


Figure 2.10: Comparison of area vs precision with different regions for SPWL

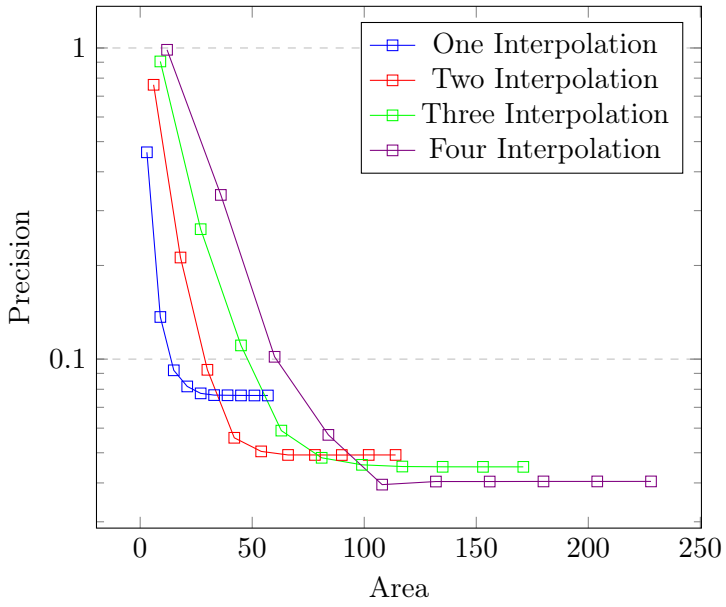


Figure 2.11: Area vs Precision for CRI

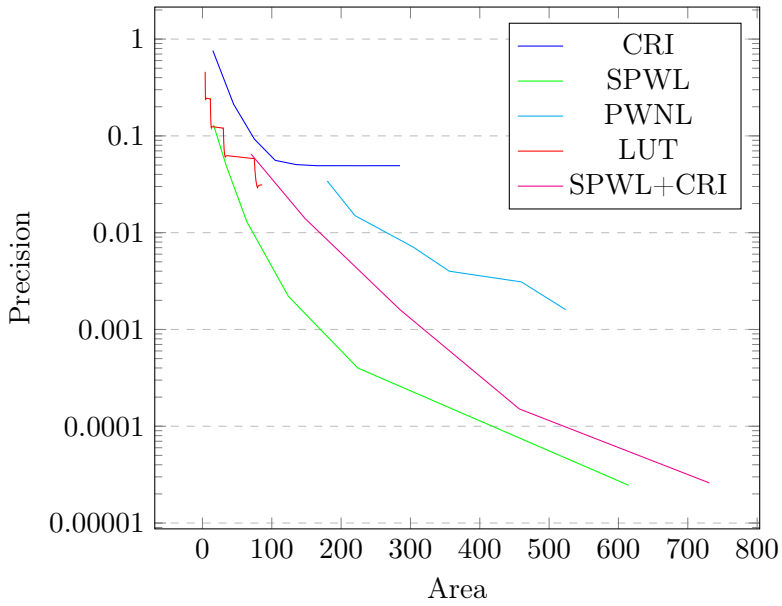


Figure 2.12: Comparison of different approximation method

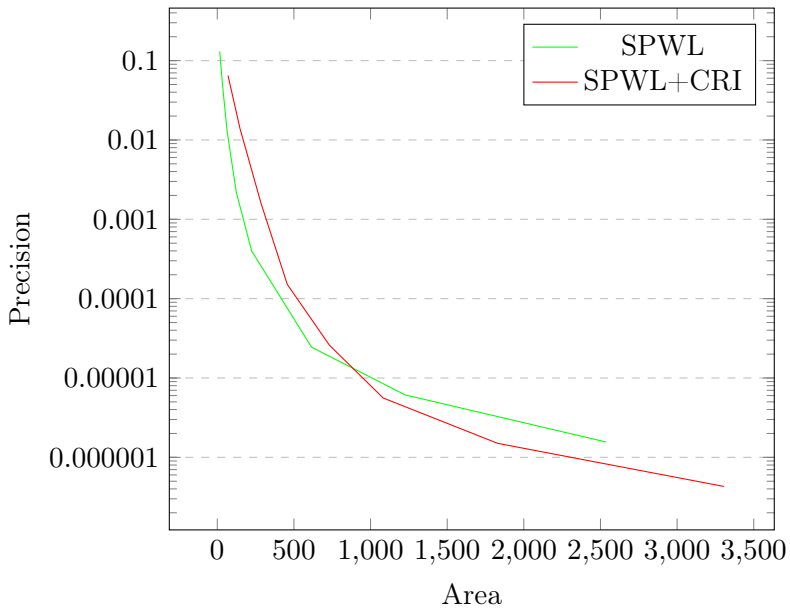


Figure 2.13: SPWL vs SPWL+CRI

Chapter 3
Low Density Parity Check (LDPC)

3.1 Introduction

As the signal is transmitted from the transmitter to the receiver, a lot of noise is added to it. A very basic approach for the receiver to convert the transmitted analog information is to its corresponding bital value is equalizing the signal with the demodulator (rounding the analog signal to the closest defined value, that is mapped to a defined set of bits). Such an approach is called "hard decision". A better approach that is more immune to noise, is adding some extra bits to the signal before transmitting it. Therefore the bital message is coded to a larger message (called codeword) and then, will be transmitted through the channel. As later will be shown, coding highly increases the system's efficiency, meaning that we can achieve a certain Bit Error Rate (BER) with consuming lower power or Signal-to-Noise Ratio (SNR).

Low Density Parity Check (LDPC) is one of the most capable coding schemes that was first introduced by Gallager[16] and has widely been used in telecommunication systems. Among the algorithms to decode LDPC codes, Sum-Product Algorithm (SPA) and Min-Sum Algorithm (MSA) are the most common ones. SPA is focused in this work, where all the bits in the received codeword (which is 5-bit long in Figure 3.1) communicate with each other and detect the mistakes caused by noise and recover the original message (2-bit long in Figure 3.1). This is why SPA is also called Belief Propagation (BP) algorithm. MSA is an efficient approximation of the SPA, normally resulting in inferior accuracy (BER) but more chip area efficiency.

Figure 3.1 shows the overview of a basic wireless system, that is considered in this thesis. Since the focus of the thesis is on the decoder, Binary Phase-Shift Keying (BPSK) modulation/demodulation has been used, which is relatively simple.

3.2 Modulation and LLRs

In Figure 3.1, BPSK modulation is used, which, for example, performs the mapping: $[0, 1] \rightarrow [+1, -1]$. This modulation is called "Living-Zero" modulation, and will be assumed as default in the remaining. The output of the demodulator could be either bits or Log-likelihood Ratio (LLR) values of bits. In the first

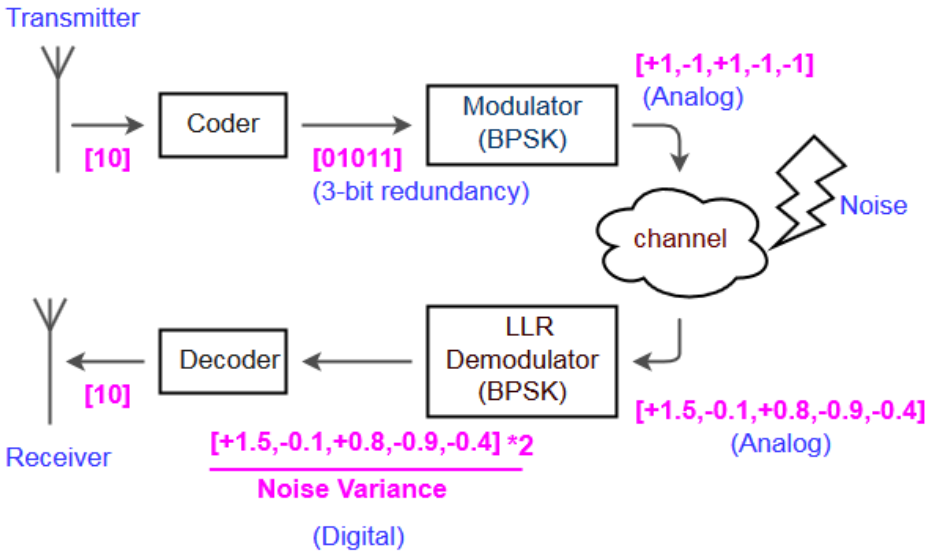


Figure 3.1: Overall view of a simple communication system

case, the demodulator equalizes its inputs, by finding the closest bital codeword that the analog signal is most likely representing. This is called "hard decision". The decoder is very easy in this case, as it only maps the found codeword to its corresponding message.

In the second case, which is the case for LDPC codes, the demodulator outputs a LLR value (also called soft-value), corresponding to each bit. The LDPC decoder will use these LLRs to make "soft decisions" on its inputs. In LLR case, the demodulator (Figure 3.1), first detects the phase of the analog input, digitalizes it, and finally converts it into an LLR value. We here show the digitalized input, as variable "u". Equation 3.1 shows how the LLR value of each input (corresponding to one bit) is calculated [17].

$$LLR(u) = log\frac{P(\ bit\ 0\ was\ sent)}{P(\ bit\ 1\ was\ sent)} \tag{3.1}$$

In Equation 3.1, the numerator is the probability that bit '0' was sent, and the denominator is the probability that bit '1' is sent. if the numerator is bigger than the denominator, the LLR becomes positive. therefore, a positive LLR is more likely to represent a '0' rather than '1'. Obviously, unsure transmitted bits have their LLR values closer to 0, as both numerator and denominator are close to each other, in these cases.

The LLR value of a bit is very descriptive, because its sign determines whether the bit is probably '1' or '0', and its magnitude shows how much this probability can be counted on, or how sure that prediction is. For example, an LLR value of "+0.1" means that the actual bit is more likely to be '0', but we are not very sure about that.

The probabilities in Equation 3.1 can easily be calculated, in case of BPSK. If

the channel has white-Gaussian Noise, which is a usual case, Equation 3.2 holds:

$$P(\,modulated\,bit = x) = \frac{1}{2\pi\sigma^2} * exp[-\frac{(u-x)^2}{2\sigma^2}] \tag{3.2}$$

Where the modulated bit (x) can be '+1' (in case of bit '0') or '-1' (in case of bit '1'). Equation 3.1 can then be rewritten as Equation 3.3. Therefore, in BPSK case, LLR value production is reduced to a mere multiplication. This is also shown in Figure 3.1.

$$LLR(u) = log\frac{\frac{1}{2\pi\sigma^2} * exp[-\frac{(u-1)^2}{2\sigma^2}]}{\frac{1}{2\pi\sigma^2} * exp[-\frac{(u+1)^2}{2\sigma^2}]} = \frac{2}{\sigma^2} * u \tag{3.3}$$

3.3 Sum-Product Algorithm (SPA)

LDPC decoder receives a codeword, except that the elements of the codeword are not bits, but Log-likelihood Ratio (LLR) values (please refer to section 3.2). How SPA maps this codeword of LLRs to the correct bital codeword, and later the original message, is discussed in this section.

At the transmitter side, the LDPC coder converts the message to the bital codeword by the binary matrix G, as shown in Equation 3.4, where "c" is the codeword.

$$message * G = c \tag{3.4}$$

The binary parity-check matrix (H) is used at the decoder, and is a matrix of size $M * N$, which is related to G by equation $G * H^T = 0$. The number of columns (N) in H equals the size of the codeword, that should be decoded and the number of rows (M) equals the size of the uncoded message. H is a low density matrix in which most elements are '0'. In SPA, each row of H represents a "Check Node (CN)" and each column represents a "Variable Node (VN)", and a '1' in H dictates that the corresponding VN (representing that column) should be connected to the corresponding CN (representing that row). An example of a H matrix with (N=8) VNs and (M=4) CNs, and its corresponding graph is shown in Figure 3.2.

0	1	0	1	1	0	0	1
1	1	1	0	0	1	0	0
0	0	1	0	0	1	1	1
1	0	0	1	1	0	1	0

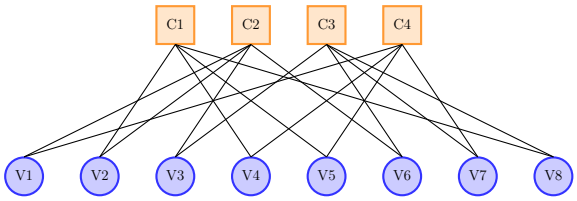


Figure 3.2: H matrix(4,8) and its corresponding graph

SPA works as connected CNs and VNs constantly communicate with each other, by sending messages through the connections. Each VN represents a bit in the codeword. First, each VN gets its corresponding LLR in the received codeword, that here we call the VN's *intrinsic LLR*. In the beginning, VNs send their LLRs to their connected CNs. Each CN processes all the messages sent from its

connected VNs, and finally predicts a value for each VN. Each VN then gets all these suggested values from its connected CNs, adds them all together and also to its *intrinsic LLR*, and obtains a more precise LLR. This concludes one iteration. If the stop criteria is not met, each VN computes and sends its so-far computed LLR (with little change) to each CN and iterations continue, until the stop criteria is met. As the algorithm proceeds, the computed LLR for each VN gets closer to either '+1' or '-1'.
For notations, if the m^{th} CN is connected to the n^{th} VN, the message from that CN to that VN is shown as $\Lambda_{m\to n}$, and the message in the opposite direction is $\lambda_{n\to m}$. At the beginning of the algorithm, the messages are initialized as follows:

$$\Lambda_{m\to n} = 0,\ \ \lambda_{n\to m} = u_i \tag{3.5}$$

where U is the codeword, and u_n is the n^{th} variable of the codeword (in LLR). Each iteration of the algorithm executes the following three steps [18]:

step (i) (CN update)

All the CNs produce their messages to all their connected VNs. If the number of VNs is N, the message from m^{th} CN to the n^{th} VN can be computed with different formulas, among which, Equation 3.6 is a good example. Section 3.4 is devoted to elaboration of these different formulas, and covers the general mathematics of Check-Node computation.

$$\Lambda_{m\to n} = 2tanh^{-1} \prod_{n=1,n\neq n'}^{N} tanh[\frac{\lambda_{n'\to m}}{2}] \tag{3.6}$$

The notation $(n' \neq n)$ in Equation 3.6 denotes that, to compute the message to a certain VN, the message from all the other connected VNs is taken into account, unless the message that has come from that VN.

step (ii) (VN update)

All the VNs produce their messages to all their connected CNs. If the number of CNs is M, The message from n^{th} VN to the m^{th} CN is computed according to Equation 3.7.

$$\lambda_{n\to m} = u_n + \sum_{m=1,m\neq m'}^{M} \Lambda_{m\to n} \tag{3.7}$$

step (iii) (stop criterion)

At each iteration, a more precise codeword is expected to be acheived. The obtained codeword(c) at the end of each iteration is determined as:

$$\forall n,\ c_n = u_n + \sum \Lambda_{m\to n} \tag{3.8}$$

There are many different stop criteria for SPA. One is based on the fact that for each valid codeword 'c', $H * (\hat{c})^T = 0$, where $\hat{c}$ is the transformation of 'c' to a digital codeword, using hard-decision. This condition can be checked at the end of each iteration. Another way is performing parity check at CNs, depending on whether "Even" or "Odd" parity check is used. In case of Even parity check, there should be an "Even" number of '1's (negative LLRs) in the set of variables (VNs) connected to each CN. In this case, each CN can check this condition by first mapping positive entries to '0' and negative entries to '1' and digitally XOR them. If the result is '0', it means there has been an "Even" number of positive entries, and the condition is satisfied. When all the CNs are satisfied, the algorithm stops. Based on the same rule, each CN can multiply all of its incoming LLR messages together. A positive result implies that there has been an even number of negative LLRs among them, and the CN is satisfied. A simpler way of stoping the algorithm is when the algorithm reaches a certain number of iterations, but this could lead to inefficiency in time.

Once the set criteria is met, the already-obtained codeword ('c' which is in LLR) is a defined coodeword and corresponds to a defined message. The codeword 'c' is then transformed to digital codeword ($\hat{c}$) using hard-decision. Then, according to Equation 3.4, the corresponding actual message can be calculated as:

$$\hat{c} * G^{-1} = message \tag{3.9}$$

3.4 CN update Methods

3.4.1 ⊞-based

In this section, the operation of a CN will be covered (please refer to [17] for more elaborate explanations). CN operation is based on the parity-check rule, that is, in case of "Even" parity check, the number of 1's (negative LLRs) in the messages from VNs, must be even. To calculate the message to a given VN, the CN processes the LLRs that it receives from other VNs. Let's assume there is an even number of 1's from other VNs. Thus, it is likely that this VN represents a '0' bit (or a negative LLR). Therefore, CN predicts and sends a negative LLR to this VN. This message will have its share in reducing the so-far predicted LLR of that VN (Equation 3.8), that will be used in next iteration (Equation 3.7). In this way, as algorithm proceeds, wrong LLRs gradually change their signs. If there are N number of VNs connected to a CN, in order for the CN to compute a proper LLR for the N^{th} variable (V_n), Equation 3.10 must be calculated (refer to Equation 3.1 for definition of LLR).

$$\Lambda_{The\,CN\rightarrow n} = log\frac{P(\,correct\,bit\,for\,V_n = 0)}{P(\,correct\,bit\,for\,V_n = 1)} = log\frac{P(\text{"}Even\text{"}\ no.\,of\ 1's\,in\,other\,VNs)}{P(\text{"}Odd\text{"}\ no.\,of\ 1's\,in\,other\,VNs)} \tag{3.10}$$

As briefly discussed in 3.3, if a set of digital bits (0,1) get XORed with each other, and the result is '0', it means there has been an "Even" number of '1's in that set. As a result,

$$\Lambda_{m\to N} = ln\frac{P(\hat{u}_1 \oplus \hat{u}_2 \oplus \hat{u}_3 \ldots \oplus \hat{u}_{N-1} = 0)}{P(\hat{u}_1 \oplus \hat{u}_2 \oplus \hat{u}_3 \ldots \oplus \hat{u}_{N-1} = 1)} \tag{3.11}$$

For notations, $\hat{u}_n$ is the correct n^{th} digital bit of the codeword, and $\oplus$ is the digital XOR. u_n is the message received from V_n, which is $\lambda_{n\to m}$. $LLR(u_n)$ is the LLR form of that message. To calculate the probabilities in Equation 3.11, we need the functionality of digital XOR:

$$P(\hat{u}_1 \oplus \hat{u}_2 = 0) = P(\hat{u}_1 = 1).P(\hat{u}_2 = 1) + P(\hat{u}_1 = 0).P(\hat{u}_2 = 0) \tag{3.12}$$

Also, from Equation 3.2:

$$LLR(u) = \frac{P(\hat{u} = 1)}{P(\hat{u} = 0)} = \frac{P(\hat{u} = 1)}{1 - P(\hat{u} = 1)} \to P(\hat{u} = 1) = \frac{e^{LLR(u)}}{1 + e^{LLR(u)}} \tag{3.13}$$

and,

$$P(\hat{u} = 0) = 1 - P(\hat{u} = 1) = \frac{1}{1 + e^{LLR(u)}} \tag{3.14}$$

Replacing Equation 3.13 and Equation 3.14 in Equation 3.12 results in:

$$P(\hat{u}_1 \oplus \hat{u}_2 = 0) = \frac{e^{LLR(u_1)}}{1 + e^{LLR(u_1)}}.\frac{e^{LLR(u_2)}}{1 + e^{LLR(u_2)}} + \frac{1}{1 + e^{LLR(u_1)}}.\frac{1}{1 + e^{LLR(u_2)}} \tag{3.15}$$

With the help of the above equation, and computing $P(\hat{u}_1 \oplus \hat{u}_2 = 1)$ with the same procedure, we can define and calculate an important double-input function, called Soft-XOR and denoted as $\boxplus$:

$$LLR(u_1) \boxplus LLR(u_2) = ln\frac{P(\hat{u}_1 \oplus \hat{u}_2 = 0)}{P(\hat{u}_1 \oplus \hat{u}_2 = 1)} = ln\frac{1 + e^{LLR(u_1)}e^{LLR(u_2)}}{e^{LLR(u_1)}e^{LLR(u_2)}} \tag{3.16}$$

Soft-XOR ($\boxplus$) is associative and commutative, and it can be proved that Equation 3.11 can be computed as:

$$\Lambda_{m\to N} = LLR(u_1) \boxplus LLR(u_2) \boxplus \ldots LLR(u_{N-1}) = \sum_{j=1}^{j=N-1} \boxplus u_j \tag{3.17}$$

Thus, the operation of CN can be simply summarized; i.e. to generate the massage to any VN, all the messages coming from "other" VNs have to be Soft-XORed, and the result will be the desired message.

3.4.2 $tanh - based$

By using advanced mathematics, it can be proven that Equation 3.17 can be rewritten as Equation 3.18 [17].

$$\sum_{j=1}^{j=N} \boxplus u_j = 2tanh^{-1} \prod_{n'=1}^{N} tanh[\frac{\lambda_{n' \to m}}{2}] \tag{3.18}$$

$\Lambda_{m \to n}$ is then calculated by eliminating $\lambda_{n \to m}$ from the above equation, which leads to Equation 3.6, which is mentioned in 3.4.

3.4.3 $\Phi - based$

CN-update step is the hardware-consuming part of the algorithm. Equation 3.6 in its current form requires some multiplications and also some approximators for the $tanh$ and one for $tanh^{-1}$ function. [7] suggests an efficient way to implement this. Since in most fixed-point hardware implementations, the use of summation is preferred over multiplication [7], and Equation 3.6 can be rewritten to replace the multiplication with summation (by taking advantage of the fact that multiplication is converted to summation in Log-domain). As elaborated in [19], If we define variable λ_i as Equation 3.19 :

$$\lambda_i = \prod_{n' \neq n} tanh[\frac{\lambda_{n' \to m}}{2}] \tag{3.19}$$

Thus:

$$ln(\lambda_i) = \sum_{n' \neq n} ln(tanh[\frac{\lambda_{n' \to m}}{2}]) \tag{3.20}$$

Therefore, the CN-update equation (Equation 3.6) can be rewritten as:

$$\Lambda_{m \to n} = 2tanh^{-1}(exp(\sum_{n' \neq n} ln(tanh[\frac{\lambda_{n' \to m}}{2}]))) \tag{3.21}$$

If function Φ is defined as:

$$\Phi = -tanh^{-1}(exp(x)) = -ln(tanh(x)) \tag{3.22}$$

The ultimate $\Phi - based$ CN-update equation becomes:

$$\Lambda_{m \to n} = \Phi(\sum_{n' \neq n} \Phi[\frac{\lambda_{n' \to m}}{2}]) \tag{3.23}$$

As seen, this equation benefits from multiple summations, instead of multiplications. However, the disadvantage is that Φ function is highly non-linear; As it has one infinity on X-axis and another on Y-axis, and the latter one must be properly taken care of. [6] has approximated this function using PWL approximation with 9 lines, and explained the Φ-based schematic of the CNU.

3.5 IEEE 802.11n H Matrix

Section 3.3 has described the parity check matrix (H). IEEE 802.11n standard was chosen in order to implement a LDPC decoder, in this work. This section will discuss the H matrix for this standard. There are 12 different H matrixes depending on 3 different codeword block lengths (1944, 1296,648 bits). And each codeword block length can be implemented using 4 different code rates i.e. $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$, $\frac{5}{6}$, each is suitable for a specific noise level in the communication channel. Figure 3.3 shows a parity check matrix for block length of 1944 bits and code rate of $\frac{5}{6}$. As can be seen, the elements of this matrix are numbers, instead of bits (0 or 1). Each number, represents a square matrix of bits (0's or 1's) with size $(sub\ block\ size) * (sub\ block\ size)$. For each block length, there is a specific sub-block size. For example, for the block length of 1944, the sub-block size is 81. An element with number "n" specifies that a unity matrix of size 81*81 (in this case) must be rotated "n" times, and then be put at that location. Right-Rotation means all the elements of each column will be tranferred to their right column, and the right-most column comes to the location of the first column (Figure 3.4). Hence, there are 324 check nodes and 1944 variable nodes for Figure 3.3.

$$H = \begin{array}{cccccccccccccccccccccccc}
13 & 48 & 80 & 66 & 4 & 74 & 7 & 30 & 76 & 52 & 37 & 60 & - & 49 & 73 & 31 & 74 & 73 & 23 & - & 1 & 0 & - & - \\
69 & 63 & 74 & 56 & 64 & 77 & 57 & 65 & 6 & 16 & 51 & - & 64 & - & 68 & 9 & 48 & 62 & 54 & 27 & - & 0 & 0 & - \\
51 & 15 & 0 & 80 & 24 & 25 & 42 & 54 & 44 & 71 & 71 & 9 & 67 & 35 & - & 58 & - & 29 & - & 53 & 0 & - & 0 & 0 \\
16 & 29 & 36 & 41 & 44 & 56 & 59 & 37 & 50 & 24 & - & 65 & 4 & 65 & 52 & - & 4 & - & 73 & 52 & 1 & - & - & 0
\end{array}$$

Figure 3.3: H matrix $(block\ length = 1944bits, code\ rate = \frac{5}{6})$

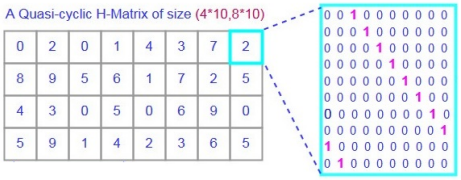


Figure 3.4: Right Rotation

Let's assume H is cyclic entry in a H matrix and sub block size is 81 bits. So, in case of 0 for the first element, we have path from variable node to check node like $[VN_1 \rightarrow CN_1, VN_2 \rightarrow CN_2, \ldots, VN_{81} \rightarrow CN_{81}]$. But in case of a rotation according to values in H matrix, from Figure 3.3, first element is 13. After cyclic shifts we get new path from variable node to check node like $[VN_1 \rightarrow CN_{14}, VN_2 \rightarrow CN_{15}, \ldots, VN_{81} \rightarrow CN_{13}]$. An entry of "-" in H matrix means those variable nodes are not connected to any check nodes. For instance, in Figure 3.3 first row and 24th column is "-", means last 81 variables are not connected anywhere. To handle this routing, we need a permutation network that will route incoming information to correct variable nodes and check nodes.

3.6 Conclusion

Evaluating these three popular CN computation methods from hardware perspective, one can argue that ϕ function has two infinities, which make the PWL approximation quite costly. Also, the the $\tanh -based$ method requires a lot of multipliers, that are expensive in fixed-point hardwares. Therefore, the $\boxplus$-based method was selected as the most hardware-friendly one among these three methods, as it requires the approximation of the $\boxplus$ function, which does not have any infinities, and its approximation can be simplified, as will be illustrated in the next chapter.

Chapter 4

Approximation of ⊞ function

As discussed in section 3.4.1, ⊞ can be a core function for CN calculation, and for its hardware simplicity, it was chosen over the other computation schemes, in this work. The equation of this double-input function is described in Equation 4.1, and is plotted in Figure 4.1.

$$a \boxplus b = ln\frac{1 + e^a e^b}{e^a e^b} \tag{4.1}$$

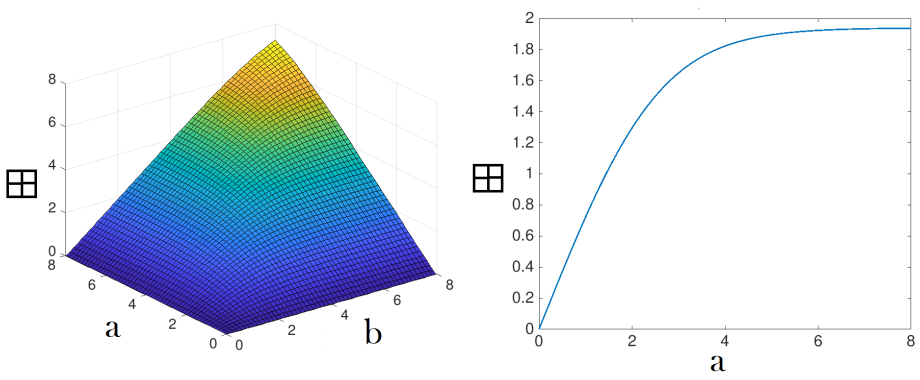


Figure 4.1: soft XOR function

In Figure 4.1, the figure on the left shows the ⊞ when the inputs are swept from 0 to +8. The figure on the right is basically an intersection of the left figure, where "b" is set at the constant value of "2".

Equation 4.1 can be made simpler to approximate by some mathematical manipulations, and can be rewritten as Equation 4.2 [20]

$$a \boxplus b = sign(a)sign(b)Min^*(|a|, |b|) \tag{4.2}$$

where,

$$Min^*(a, b) = min(a, b) - ln(1 + e^{-|a-b|}) + ln(1 + e^{-|a+b|}) \tag{4.3}$$

In this way, the sign calculation (which requires a simple digital XOR) can be departed from the absolute (magnitude) calculation, which is done by Min^* function. Min^* takes the absolutes of both inputs and generates the absolute of the result. It is worth noting that for positive inputs: $\boxplus = Min^*$. For simplicity, from now on we only consider positive inputs, where this condition holds. The remaining of this section is dedicated to different approximations of Equation 4.3.

4.1 Min-Sum Approximation

Equation 4.3 has three terms. The first term ($min(a, b)$) contains the largest portion of the total function of Min^*. In Figure 4.2, the right figure shows this term and its closeness to the complete function. Therefore, this term can be an approximation of the Min^* function, and the LDPC decoding algorithm that is based on such approximation is called Min-Sum Algorithm (MSA). This algorithm is very popular, especially for its small cost.

4.2 Double-PWL Approximation

In applications where more precision is required, the approximation of the second and third term of Equation 4.3 is also considered. For both these terms, the approximation of the function $ln(1 + e^{-|x|})$ is required. The left figure in Figure 4.2 is approximating this function, using PWL with one single line (Equation 4.4) [5].

$$ln(1 + e^{-|x|}) \sim max(0.625 - \frac{|x|}{4}, 0) \tag{4.4}$$

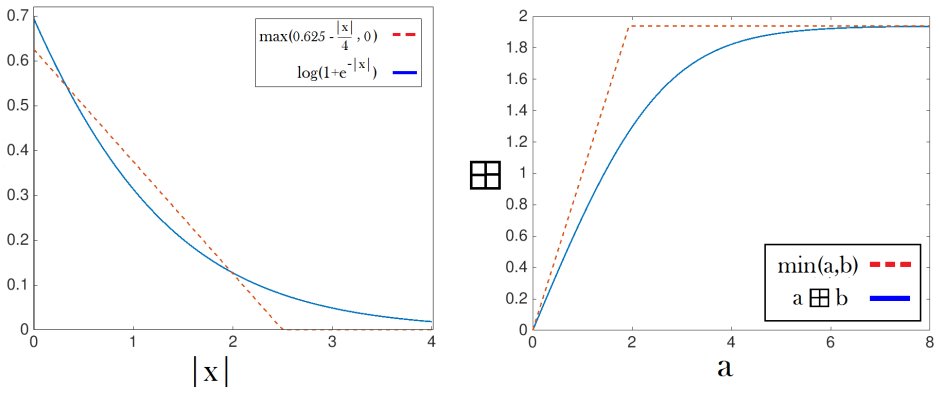


Figure 4.2: Soft XOR Approximation

4.3 Single-PWL Approximation

In order to reduce hardware cost but maintain precision, this approximation is based on considering only the first and second term of Equation 4.3 and the last term can be ignored, and the error will not be considerable [22][20]. The reason for ignoring the third term is that for $(a+b) > 2.5$, $log(1+e^{-(a+b)}) \sim 0$, and therefore this equation only has value for small values of "a" and "b". Also, in our C++ simulations that deals a normal range of $-8 < a, b < 8$, it was observed that small values barely have an effect on the overall result. However, in some applications that deal with small LLRs (probably where Noise variance (σ^2) is considerably high as shown in Equation 3.3), ignoring this equation might deteriorate the performance. Therefore, Equation 4.3 can be approximated as:

$$Min^*(a,b) = min(a,b) - max(0.625 - \frac{|a-b|}{4}, 0) \tag{4.5}$$

where $|a-b|$ is in fact $max(a,b) - min(a,b)$ in hardware implementation.

4.4 CRI-based Approximation

In this method, we utilized the CRI technique (discussed in 2.3.4) to estimate the Min^* function. Here, we have used a single-step CRI ($q = 1$). For this, two lines that are tangent to the original curve are needed as initial approximation, plus a suitable $delta(\Delta)$. As the initial tangent lines, we can use $y = a$ and $y = b$. Figure 4.6 shows the original curve, and these two lines. The approximated curve is shown in Equation 4.6.

$$Min^*(a,b) = min(a,b,\frac{a+b}{2} - \Delta) \tag{4.6}$$

The calculation of Δ is explained as follows. As seen in Figure 4.6, Δ is the difference between the intersection of the two lines and the original curve, where $a = b$. According to Equation 4.3, this difference is:

$$\Delta = ln(1+e^0) - ln(1+e^{-(a+b)}) \tag{4.7}$$

for $(a+b) > 2.5$, $log(1+e^{-(a+b)}) \sim 0$. As discussed in the above section, this term can be ignored, without considerable loss of accuracy. Therefore, Δ is estimated as:

$$\Delta \sim 0.625 \tag{4.8}$$

As Figure 4.6 shows, $\Delta = 0.625$ makes the $\frac{a+b}{2} - \Delta$ line to be tangent to the original curve, at $a = b$ point. As illustrated in section 2.3.4, Δ can be quite flexible, and can be lowered by some small amount, so that the $\frac{a+b}{2} - \Delta$ line passes through the original curve, instead of being tangent to it. Through MATLAB simulations, it was observed that instead of $\Delta = 0.625$, using 0.8 or 0.9 results in better accuracy. Therefore, the Min^* is estimated as Equation 4.9. It is worth to note that in this equation, the absolute function will not be implemented

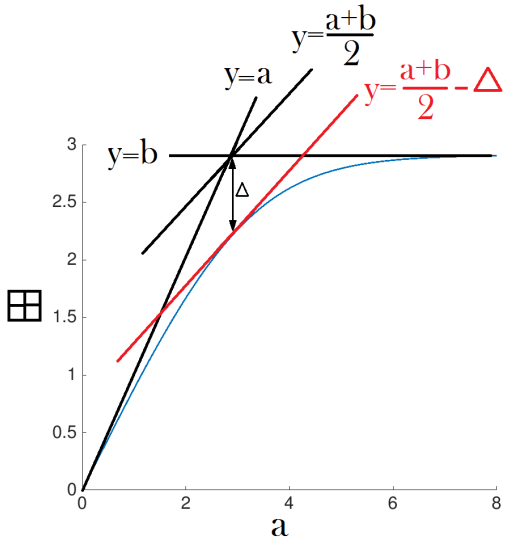


Figure 4.3: CRI-based approximation of Soft-XOR (b=3)

in hardware, as negative values become naturally large, as the MSB bit becomes '1'.

$$Min^*(a,b) = min(a,b,|\frac{a+b}{2} - 0.8|) \tag{4.9}$$

In [23], the Min^* equation is estimated as Equation 4.10 which, if rewritten in a more hardware-friendly form, resembles Equation 4.9, with some modifications.

$$Min^*(a,b) = max(min(a,b) - max(0.9 - \frac{|a-b|}{2}, 0), 0) \tag{4.10}$$

Finally, For those applications in which LLR values are considerably small (probably because of high Noise Variance), ignoring the $log(1+e^{-(a+b)}) \sim 0$ might deteriorate the performance. In those applications, this term can be approximated with a single line (Equation 4.4) and $\Delta \sim min(0.8, \frac{a+b}{4}) \sim min(0.8, \frac{max(a,b)}{2})$ can be achieved (denoted as Low-LLR CRI). Its hardware implementation consumes slightly more area than Equation 4.9, but approximates the small LLRs more accurately. In this work, we will regard Equation 4.9 as the CRI approximation.

4.5 Result and Conclusion

Five common methods to approximate the ⊞ function have been explored. Figure 4.5 and Figure 4.6 depict these approximations for large LLRs and small LLRs, respectively. The simulation result of BER/SNR of the LDPC decoder is shown in Figure 4.4, for each of these approximations. In this picture, SPA denotes the exact SPA algorithm. Figure 4.4 shows the approximations for different ranges. in

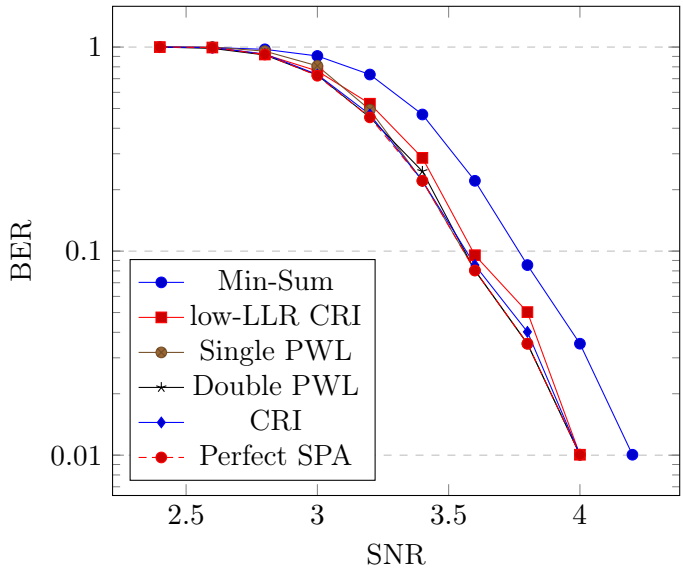


Figure 4.4: SNR vs BER for different soft-xors

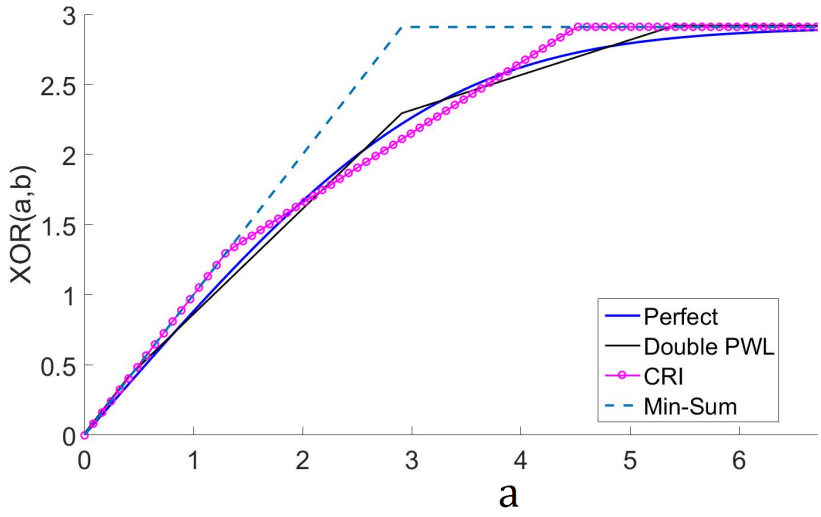


Figure 4.5: Soft-xor approximations for large LLR ranges (b=3). Single-PWL, Double-PWL and low-LLR CRI all work identically.

low LLRs, CRI version performs only slightly better than "Min-Sum", but in large LLRs, behaves almost similar to Double-PWL. The RTL results of the approximations are also provided in chapter 7. "min-sum" has the worst performance but smallest area, as it uses only one comparator. However, simulations show that as soon as some computations are added to Min-Sum, its performance starts to

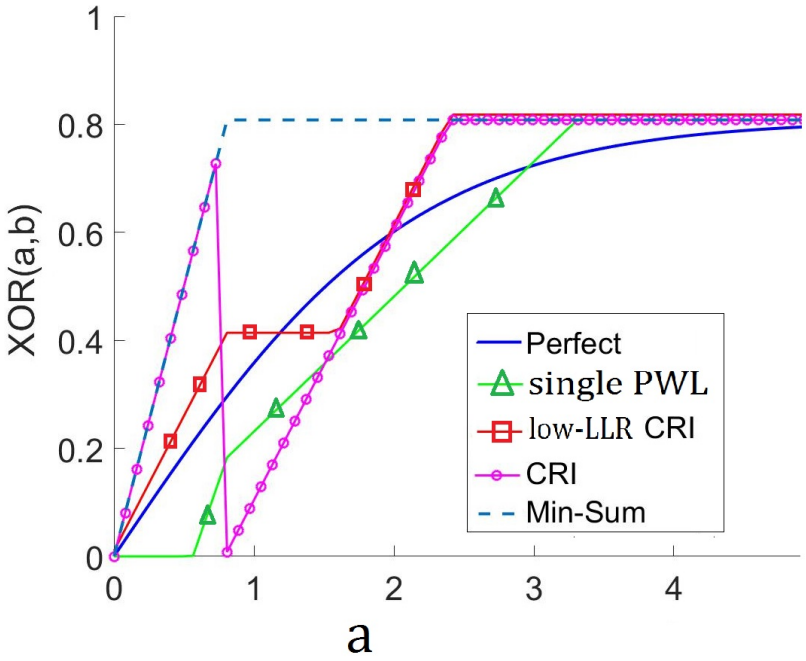


Figure 4.6: Soft-xor approximations for small LLR ranges (b=0.8). Double PWL works almost identical to low-LLR CRI.

rapidly improve and become very close to SPA, which suggest that SPA algorithm is highly robust to inaccuracies, and rough estimations perform very well. It can be seen that all the mentioned approximation methods demonstrate an excellent performance in regard of accuracy, except for "Min-Sum". Thus, when it comes to adding computations to ⊞, the cost must be the main concern. CRI-based method manages to add the smallest computation, and there is a decrease of 62% in BER of the "CRI-based" with offset 0.8, compare to "Min-Sum". We have tried two different version of CRI-based approximation with different offsets, and offset of "0.8" is more optimal. Overall, it can be seen that the CRI-based ⊞ approximation, with relatively small area consumption, is a suitable choice for applications in which a near-SPA BER/SNR is required. "Min-Sum" based ⊞ is the best candidate where hardware cost is the primary concern. As the final note, it was shown that although PWL was determined as the most efficient approximation method in chapter 2, but in soft-xor case, CRI outperformed PWL, which illustrates that the efficient approximation depends on the type of the function, as well. Soft-xor is a double-input function and does not require high accuracies. Also, its special mathematics made it possible to be approximated in more area-efficient ways than using LUT or PWL.

Chapter 5
Simulation Results

In this chapter we will discuss and analyze the SPA behavior (BER/SNR) and its dependency on the number of iterations and number of bits (quantization). The analysis is based on a C++ simulator that is formerly developed to simulate a complete SPA decoder with flooding schedule. It takes its inputs from a BPSK demodulator, and computes the decoded bital message and then calculates its corresponding BER versus SNR (BER/SNR).

5.1 Effect of iterations on BER

This section is about how number of iterations are effecting decoding performance. Figure 5.1 shows that we can get same BER performance by increasing iterations at low SNR and there is reduction in BER as iterations increases. This was expected because with every iteration received input route between variable node and check node, which decrease the error. However, using high number of iterations for simulation might not very efficient because simulation time would increase very much. But in hardware this would done very fast. Therefore, to reduce simulation time, number of iterations are kept to 10 for other simulations. In RTL design number of iterations is used as stopping criteria.

5.2 Effect of total bits on BER for SPA

This section describes the effect of total bits of variables on bit error rate. Figure 5.2 shows our simulated results. It is obvious that as we decrease number of bits error would increase because much information can lost in those bits. From simulation we found that only two number of integer bits are enough to represent LLR integer parts. Therefore, in all curves integer bits were kept same while fractional bits were varying. As SPA is memory hungry, so even an increase of single bit is very expensive. Finding a good number of bits that can give reasonable decoding performance is required. In Figure 5.2 first curve is giving worst performance because of very less number of information bits. But as we increase information bits, the BER would improve. However, after 4 information bits there is not much improvement. And "*Full*" curve is without any fixed point limitation. Through this figure it can be seen that BER for 8 information bits are almost equivalent to

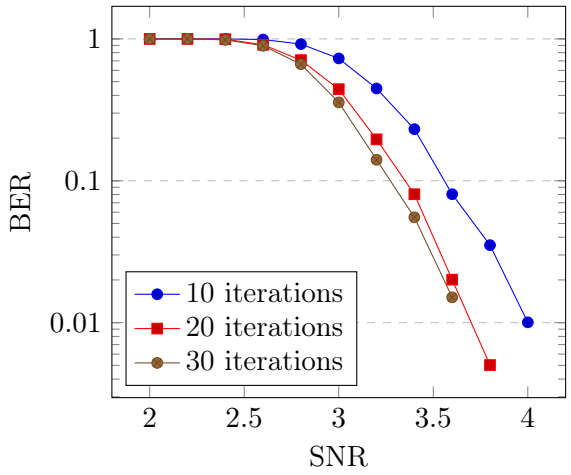


Figure 5.1: SNR vs BER for different iterations

full precision.

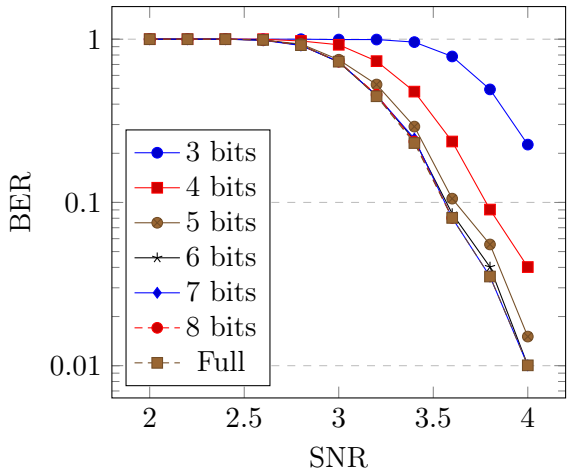


Figure 5.2: SNR vs BER, for different number of bits

In Section 5.1, we discussed about impact of iterations on BER. As we know by increasing iterations, BER will decrease but on the other hand the total time of decoding would also increase. Hence there is a trade-off between iterations and decoding time. Figure 5.3 compares different number of iterations with different number of bits. In this figure, for an instance at 3.4 SNR first case which is 7 bits and 10 iterations has a BER of 0.246. But an increase of iterations by 2 and decrease of bits by 2 gave better result. By doing so, BER was decreased from 0.246 to 0.196, which means a decrement of 0.05. However, decoding time is increased but we save 2 bits and this has major effect on memory requirement.

Total memory required is $(7776 * NOF_BITS)$bits. If we would have chosen 7 bits and 10 iterations, then our memory size would be of 54432 bits. But for 5 bits and 12 iterations, memory size required is 38880 bits. Our net memory saving is around 15.5K bits. As this save is only for one RAM and we have 4 RAMs, so net save would be 68K bits which is quite high. If even low BER is a requirement then 6bits with 12 iterations would also be a reasonable selection. This has a quite close performance to the ideal case. Therefore, from this we can conclude that an increase of 20% in iteration and decrease of 28% in bits would reduce the memory size by 28% for one RAM. Now if we compare 6 bits, 12 iterations and 5 bits, 12 iterations, memory size is decrease by 20% while BER is increase by 25%.

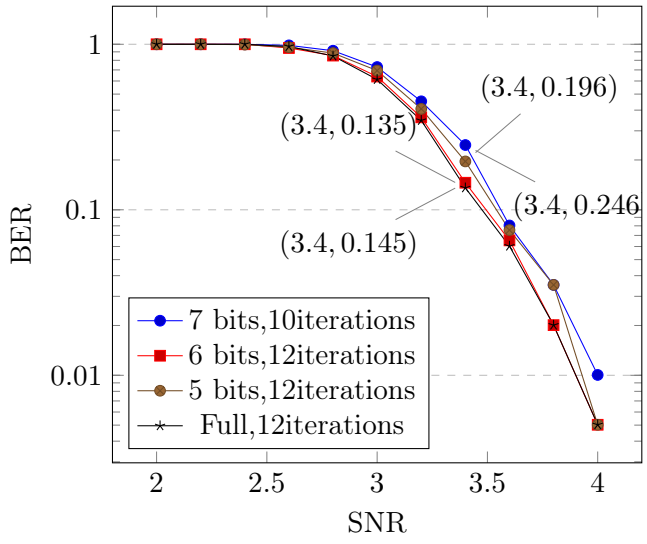


Figure 5.3: Comparison of SNR vs BER for different bits and iterations

5.3 Number Representation

Among the three popular numeric systems, that are fixed-point, floating point (FP) and logarithmic (LNS), the fixed-point has been selected in this work, and all the simulations, approximations and hardware implementations are based on that. There are two reasons that FP and LNS were not selected for SPA application:

5.3.1 SPA's Incompatibility with FP and LNS

LNS and FP provide higher accuracy for representing low values, but lower accuracy for larger values. However, as shown in Figure 4.4 and Figure 4.6 in chapter 4, the Soft-XOR approximations that exhibit low accuracy for estimating small values perform as adequately as others, but those that estimate large values with low accuracy, perform poorly in BER/SNR performance. This illustrates that

large values are more important for CN computation of SPA (at least for our case study with normal noise level). Therefore, LNS and also FP behave in the opposite direction of the accuracy of SPA. However, a number representation that is inverse of LNS could work in favor of the mentioned property of SPA. For example, in such a numeric system, 3 is represented by 8 if the base is 2. Exploration of this system remains for future work.

5.3.2 Difficult Addition in FP and LNS

Another limitation on FP, LNS and it's inverse is the VNU. VNU's main task is addition, which is relatively difficult for any numeric system other than fixed-point. For LNS and its inverse, one way is that some hardware is added to the I/Os of all CNUs to act as converters to/from fixed-point. Therefore, VNU handles fixed-point additions and will be small. For LNS, the added hardware has to perform logarithm to all inputs and inverse-logarithm to all outputs of CNUs. Another way is performing addition without such converters, which is much larger than fixed-point addition.

Chapter 6
Hardware Architecture

This chapter describes the SPA ASIC architecture of flooding LDPC decoder. The overall architectures were designed to be compatible with any IEEE 802.11n code rates. IEEE 802.11n has three different codeword block length of 1944, 1296, 648 bits and each block length has four different code rate i.e $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$, $\frac{5}{6}$. The difference between these block lengths is number of variables and number of entries in their respectively H matrix. In case of large codeword block length, the memory bandwidth increases and vice-versa. For 648 bits block length, sub block size is 27 bits, for 1296 bits sub block size is 54 bits and for 1944 bits sub block size is 81 bits. However, the increase in sub block size is a multiple of 3, this symmetry can be utilized in design of a configurable hardware for all block lengths. However, the memory overhead leads to a trade-off situitation, while designing a configurable LDPC decoder, because of large sub-block sizes. For instance in case of 1944 bits block length, if we process one entry from H matrix at a time, then the incoming data will be $81 * NOF\_BITS$ bits, where $NOF\_BITS = number\ of\ bits\ of\ a\ variable$. So, for $NOF\_BITS = 7$ (which is a usual case), it requires a memory with a width of of 567-bits, which is quite large and inefficient. Also, the speed of decoding is dependent on how many entries in H matrix we process at a time. If we process more than one entry at a time, then memory bandwidth would increase accordingly. These problems needs to be tackled properly, in order to design an efficient LDPC decoder.
Therefore, we decided to divide a RAM into three parts, each one with a width of $27 * NOF\_BITS$ bits. More detail explanation can be found in memory section below. Figure 6.1 shows the top level architecture of a LDPC decoder based on SPA. RAMs and ROM are used to store all the necessary information. Permutation network will perform cyclic shift of incoming variables based on entries in a particular H matrix. CNU is the Check Node Unit, which updates all the incoming information coming from variable nodes. Similarly VNU is Variable Node Unit which update all the incoming variables. Detailed explanation of every block in Figure 6.1 is described below.

6.1 CNU Implementation

Check Node Update unit is the major processing unit as described in section 3.4. To implement the SPA CNU in hardware, a forward-backward [18] ap-

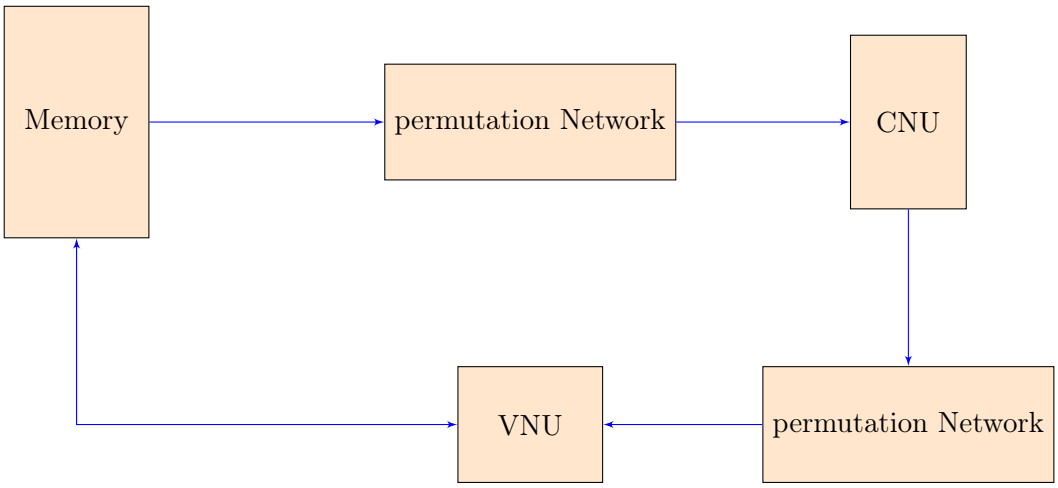


Figure 6.1: Top level architecture

proach was chosen. There is also the parallel architecture, but it needs the inverse function of soft-xor which is much more cumbersome to approximate [18]. In the forward-backward architecture, consider a check node m that has t_x connections to variable nodes, N(m) = n_1,n_2,n_3,....,n_{t_x}. The incoming variables are $\lambda_{n_1 \to m}(u_{n_1})$, $\lambda_{n_2 \to m}(u_{n_2})$, $\lambda_{n_3 \to m}(u_{n_3})$, $\lambda_{n_4 \to m}(u_{n_4})$,$\lambda_{n_{t_x} \to m}(u_{n_{t_x}})$. CNU unit will calculate new messages for variable nodes as follows $\Lambda_{m_1 \to n}(u_{n_1})$, $\Lambda_{m_2 \to n}(u_{n_2})$, $\Lambda_{m_3 \to n}(u_{n_3})$, $\Lambda_{m_4 \to n}(u_{n_4})$,....., $\Lambda_{m_{t_x} \to n}(u_{n_{t_x}})$.
From [18] CNU is divided into three parts forward update, backward update and merge. Forward update is defined as: $f_1 = u_{n_1}$, $f_2 = f_1 \boxplus u_{n_2}$, $f_3 = f_2 \boxplus u_{n_3}$,.....,$f_{t_x} = f_{t_{x-1}} \boxplus u_{n_{t_x}}$. Similarly backward update is defines as $b_{t_x} = u_{n_{t_x}}$, $b_{t_{x-1}} = b_{t_x} \boxplus u_{n_{t_x}}$, $b_{t_{x-2}} = b_{t_{x-1}} \boxplus u_{n_{t_{x-2}}}$,....., $b_1 = b_2 \boxplus u_{n_1}$. The result of forward and backward can be expressed as follows, to generate the outgoing messages:

$$\begin{aligned} \Lambda_{m \to n_1}(u_{n_1}) &= \mathcal{L}(b_2), \\ \Lambda_{m \to n_i}(u_{n_i}) &= \mathcal{L}(f_i - 1 \boxplus b_i + 1), i = 2, 3, 4, ..., t_x - 1, \\ \Lambda_{m \to n_{t_x}}(u_{n_{t_x}}) &= \mathcal{L}(f_{t_{x-1}}). \end{aligned} \tag{6.1}$$

Where $\boxplus$ is a soft-XOR. The architecture of CNU was designed so that it can process more than one entry in the H matrix. Therefore, the more entries CNU can process at a time, the faster the decoder is. The number of processed entries at a cycle is a parameter in the code (cnu rate) and can be set before fabrication. Figure 6.2 shows the hardware representation of Equation 6.1. In this example, CNU can process four entries in the H matrix. But this leads to a higher memory overhead, which is a bottleneck in this algorithm, as discussed in section 6.3. As described in section 3.5, one entry in the corresponding H matrix encompasses a number of variables that is equal to its sub block size, which is 81 in this case. So, it requires 81 CNU instances in order to process one element of H matrix and the number of instances increases as we increase the number of sub-blocks processed at a cycle.

Figure 6.4 and Figure 6.3 show the data flow and timing of CNU respectively. CNU proceeds in a row-wise fashion in H matrix. An enable signal is used to trigger CNU. The main challenge in the forward-backward architecture is that the merge&backward stage can not be started before the forward stage is completed for the complete line of H matrix. Therefore, to pipe-line the architecture, a number of registers are needed to store the results of forward stage and also the coming entries, so that after finishing the forward stage, the stored tokens in the registers can be consumed by the second stage. As values in registers are being consumed by the second stage, their locations in the pipe-lining registers are given to the new values coming from forward stage and CNU inputs. Therefore, the data flow can be continuous. *first* and *last* are the signals that are used in order to locate first and last elements of the row. These signals are needed by backward and merge stage. The CNU requires 24 clock cycles to process one row, if CNU is processing one entry at a time. The output of CNU will then pass through a permutation network for cyclic shift and will be stored in corresponding RAMs.

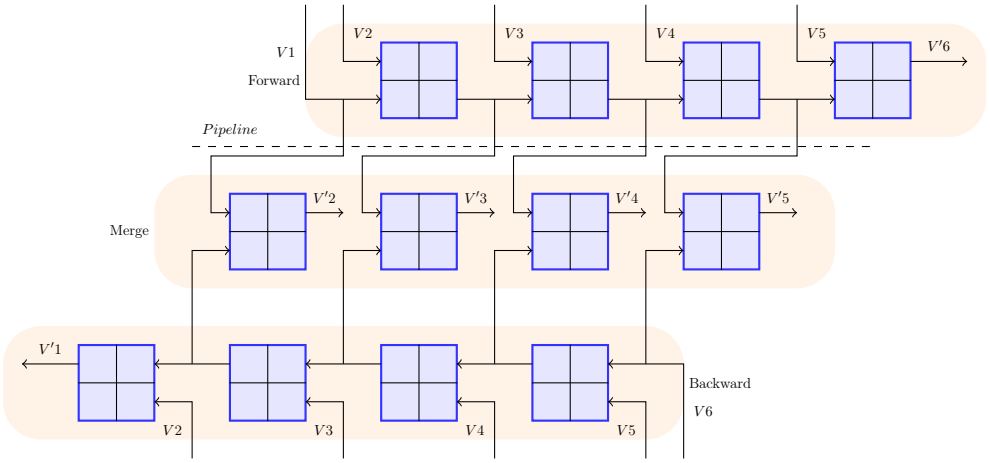


Figure 6.2: Forward-Backward Architecture

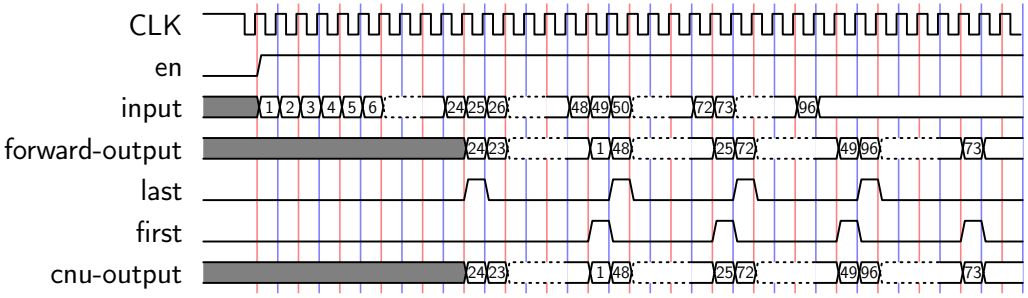


Figure 6.3: CNU timing diagram

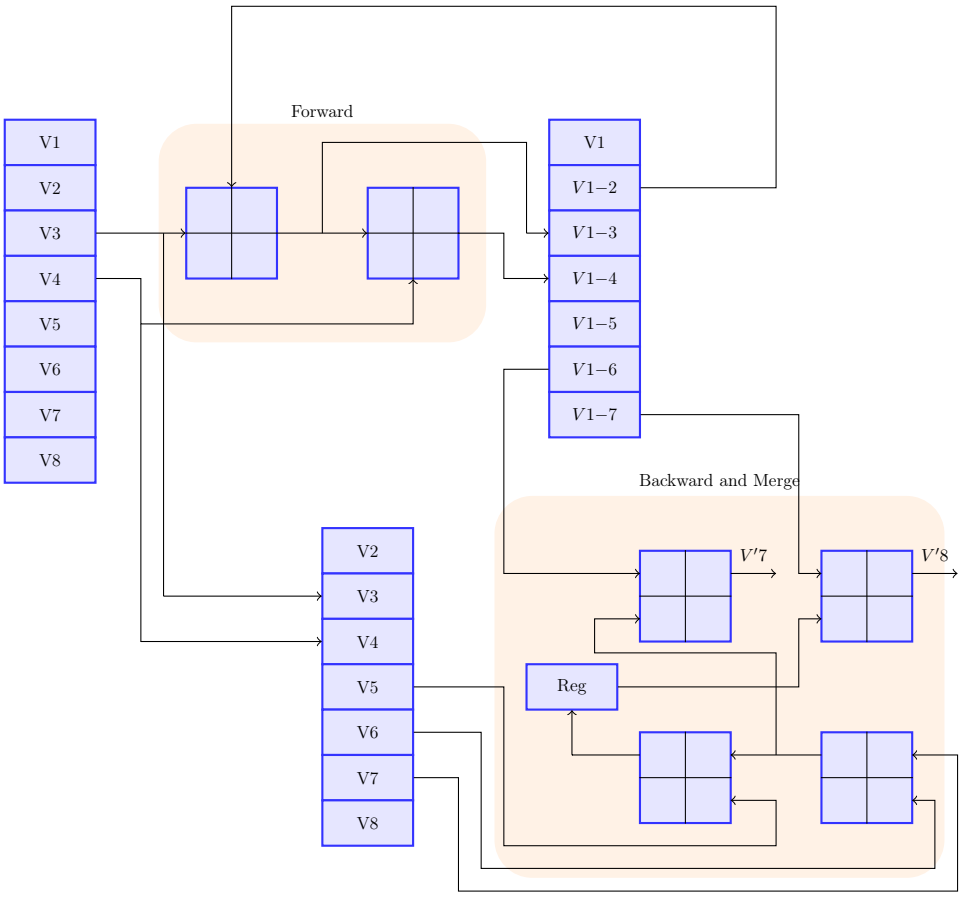


Figure 6.4: CNU hardware architecture for SPA

6.1.1 Min-Sum CNU Implementation

In a similar fashion to SPA CNU, a simple flexible Min-Sum based CNU was implemented with the same I/O pattern. The below equation describes the behavior of this CNU.

$$\Lambda_{m\to n} = \prod_{n\neq n'}^{N} sign(\lambda_{n'\to m}) * \min_{n\neq n'} |\lambda_{n'\to m}| \tag{6.2}$$

According to the equation, the signs can be easily computed by digital XORs. To compute the absolute values of the messages, two minimums are needed to be derived. The first minimum is sent to the all the variable nodes, except the one which has sent this minimum to the CNU. To this variable node, the second minimum will be sent. Figure 6.5 shows the hardware, for absolute value computation. Each CF (Core Function) takes an absolute input, compares it with the two minimums achieved so far, and obtains the first and second minimums

out of these three numbers. Then, it sends the two obtained minimums to the next CF. Therefore, each CF does two comparisons. "Index" stores the number of the message that has given the first minimum, and is therefore updated by CFs whenever the first minimum is updated. The minimums and the index are saved at the end of each cycle, in the registers. When inputting one line of the H matrix is complete, the final minimums and index are handed to the output stage, and are stored there. The output stage starts giving output, based on the stored minimums. For Sign computation, the signs of all the coming inputs are stored in a 24-bit register, in order. Also, a single bit register stores the XOR of all the signs stored in this register. At the end of one line, this single bit register contains the XOR of the complete line, and will be handed to the output stage, along with the minimums and index. As the inverse of XOR is XOR, the signs of the n^{th} output is this single bit XORed with the n^{th} bit of the 24-bit register.

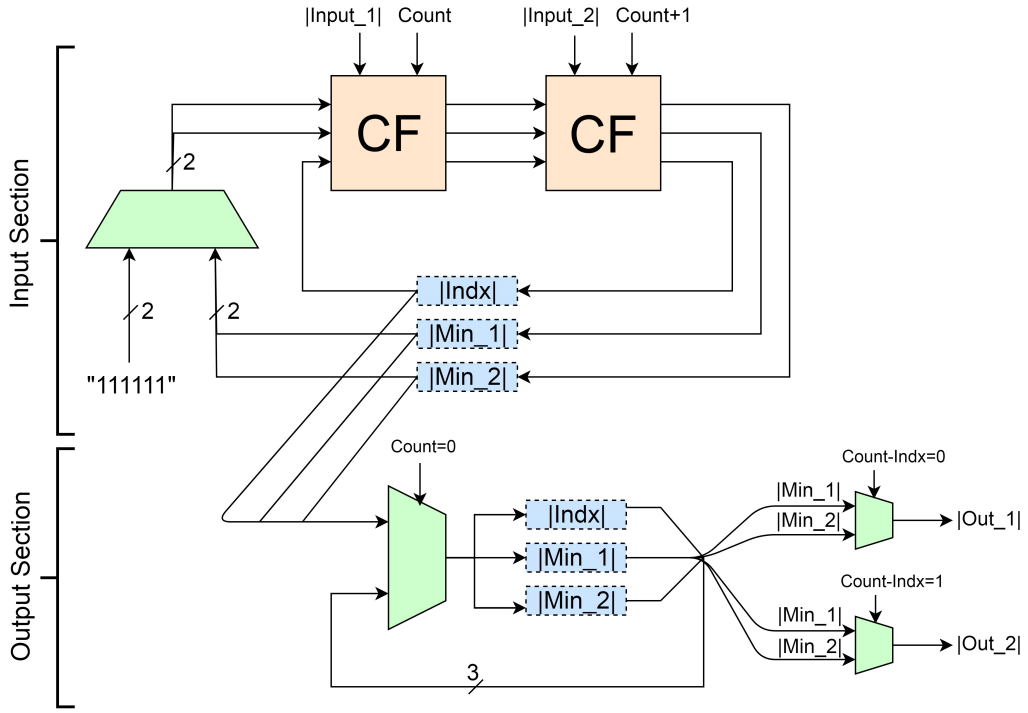


Figure 6.5: Implemented Min-Sum CNU architecture

6.1.2 CNU Trade-offs

When it comes to CNU hardware design, there are some trade-offs present. As mentioned earlier, CNU could process many number of elements from H matrix at the same time, which determines the speed of the design. This parameter is denoted as "cnu rate", which is the number of elements from H matrix that enter the CNU in each cycle. Therefore, an evaluation needs to be done in order to find

an efficient "cnu rate". After performing some simulation based on decoding speed as well as area consumption, we could choose a cost-efficient "cnu rate". However, by increasing the speed, memory bandwidth exponentially increases. For instance, memory bandwidth required for cnu rate = 1 is $81 * NOF\_BITS$ but for CNU rate = 2 this would increase to $2 * 81 * NOF\_BITS$, which is quite high and probably impractical. One solution could be to use temporary registers but using registers are very costly. Another one is to use more number of RAMs as RAM memories, while the total RAM area remains the same. However, dividing the memory to small portions results in area increase. Therefore, an efficient selection of cnu rate is very effective.

6.2 Permutation Network

As described in section 3.5, to perform cyclic shifts, a permutation network is required. A permutation network behaves like a switch, whose job is to route data to a specific output node. For example, before doing cyclic shift , input node 1 was connected to output node 1 but after doing a certain number of cyclic shifts input node 1 is now connected to output node 4. Here we need to route data from a variable node to a check node and vice-versa. Therefore, Barrel shifter is used to perform this routing. A Barrel shifter performs shifting in stages. For instance, first stage of it will do 8 bits shifting, second will do 4 bits shifting and so on. Number of stages can be calculated as $log_2 N$, where N is the number of bits in a data. As it requires data to be in a form of 2^x, so the number of stages will be calculated based on 128 bits instead of 81 bits.
Therefore, it requires 7 stages to perform 81 bit cyclic shifts. As mentioned above, it has 7 different stages doing 2^N shifts. As it is just shifting, multiplexers can be used to design the whole network. Being completely combinational in design is also an advantage because the area cost will be less. Figure 6.6 shows an example of how a 6 bit barrel shifter shifts data by using three shifting stages 4 bit, 2 bit and 1 bit. Each multiplexer is 2:1 and in each stage, multiplexers has same control signal. So, for 81 bits shift, it requires seven stages, every stage will perform a number of shifts that are a power of 2. In total, 567 number of 2:1 multiplexers are required and the whole rotation is controlled via a 7 bit control signal which is obtained through stored coefficients in a ROM. As mentioned in section 3.5, some entries in H matrix are " − ". For those, there is no need to do any shifts. Therefore, in order to detect " − ", 2 extra bits in concatenation with the 7 bits are stored in the ROM. Therefore, the controller can detect how many entries it has to skip in the next cycle.

6.2.1 VNU

Variable Node Unit (VNU) is simply an adder and a subtracter. It fetches data from RAMs and updates the variables and send it back to check node for next iteration.

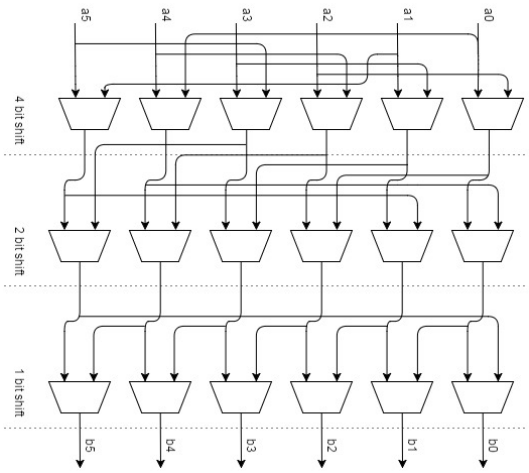


Figure 6.6: Barrel shifter

6.3 Memory

6.3.1 RAM

One of the main challenges in SPA is handling of data. This is because after expansion of H matrix to bits, we will have 1944 variable nodes and 324 check nodes for the highest block length and code rate (more detailed can be found in section 3.5). Each variable node will route a certain amount of bits, which is denoted as NOF_BITS. As we are using flooding schedule, all the messages of the previous iteration are needed in order to generate a new message. In other words, to update messages of first row in H matrix, we requires previous iteration's messages of all rows of H matrix except the first one. Therefore, storing all the results of each iteration in some form is necessary, which necessitates the use of multiple RAMs. Another bottleneck is the bandwidth of a RAM. For instance, a sub block size of 81 bits, requires a bandwidth of $81 * NOF\_BITS$, where NOF_BITS is the number variable bits. So, if NOF_BITS are 7 then bandwidth of an individual would be 576 bits which is somehow large. One way to overcome this problem is to reduce data bandwidth but this will reduce decoding speed. Hence, we partitioned a RAM into three sub-RAMs with $\frac{1}{3}$ of previous bandwidth but same depth. This was decided because of the fact that IEEE802.11n has three different sub block sizes which are multiples of 3 e.g. 27,54 and 81, so for lower sub block sizes, rest of the sub-rams can be easily switched off, which would contribute to the flexibility of the architecture.
Therefore, one RAM is sub-divided into three parts in order to design a flexible architecture for all sub block sizes. Therefore, each of them has a bandwidth of "$27 * NOF\_BITS$". In total, there are four RAMs, each sub-divided into three. One RAM is used to store received data. Other three RAMs are used to decode received data. Total memory requirement is "$96 * 81 * NOF\_BITS = 7776 * NOF\_BITS$" bits, which in case of 7, becomes around 50K-bit. In Figure

6.8, R0, R2, R3 are of size "$24 * 81 * NOF\_BITS = 1944 * NOF\_BITS$", where 24 is the depth and "$81 * NOF\_BITS$" is the width, while R1 has the size of "$6 * 81 * NOF\_BITS = 7776 * NOF\_BITS$", with the same width.

6.3.2 ROM

As elements of H matrix for different block length and code rate don't change, a ROM is used to store H matrix elements. The width of ROM is decided by the maximum sub block size which is 81, so 7 bits are enough to store all elements. The depth is dependent on the number of entries. For 1944 block length and $\frac{5}{6}$ code rate, number of elements are 79. Therefore, in case of configurable architecture, all elements of H matrix for every code rate and block length are stored in a big ROM. As mentioned in section 3.5, there is no need to store "-" elements. Instead, some extra bits are concatenated with 7 bits to "leap" over the "-" elements, which will considerably increase the speed in low code rates.

6.4 Data flow

This section describes the data flow of the complete LDPC decoder architecture. Figure 6.8 and Figure 6.7 show data flow architecture and memory accesses respectively. In Figure 6.8, M0, M1, M2, M3, M4 are data routing multiplexers and R0,R1,R2,R3 are RAMs, among them R0 is single port while R1,R2,R3 are dual ports. The received input that needs to be decoded is in R0, as initial variables, and remains unchanged throughout the decoding. According to the decoding algorithm, there is no need to perform variable node update in the beginning of the first iteration. Thus, in the very beginning of computation, data from R0 will be sent to CNU for check node update via permutation network by selecting respective signal for M3, as shown in Figure 6.7. Also each element of H matrix has a location in R1 matrix, which the respective variable will be written on, or read from, when they are processed. In the first iteration, the results get accumulated and written on R2. Thus, at the end of this iteration, the addition of all the results of each column are in one location of R2. In second iteration, these values are only read, then the previous value of the variable under process (which is in R1) is subtracted from them and the result will go to CNU. This emulates the function of VN update rule. The results are accumulated in R3, in the second iteration. Therefore, R2 and R3 change position in every iteration, alternatively. M5, M2 and M1 are responsible for this alternation. M0 excludes the first iteration from receiving any data from RAMS, as there is no previous data at that time.
The whole CNU is divided into three sub-parts in a bunch of 27 each, to make it work with any sub block size for IEEE802.11n standard (so that some parts could be switched off for flexibility).

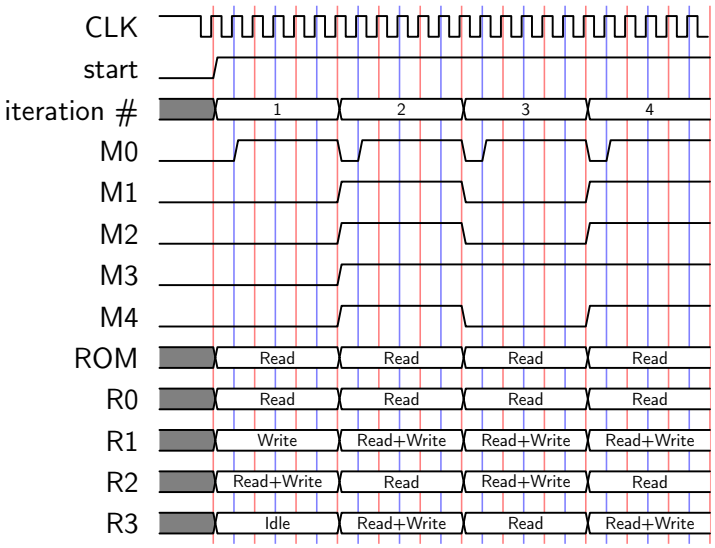


Figure 6.7: Memory access timing diagram

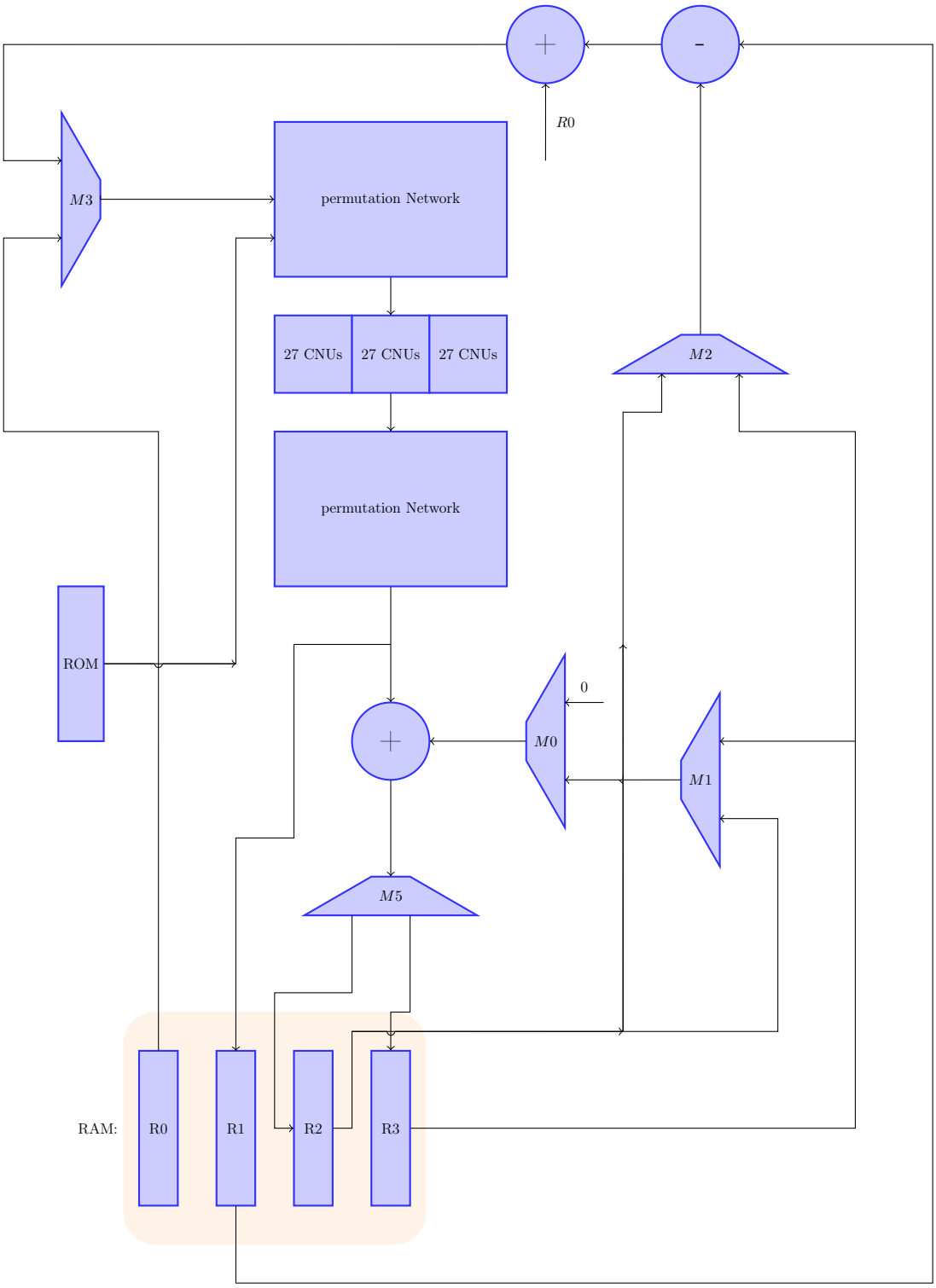


Figure 6.8: Data Flow

Chapter 7
Synthesis

In this chapter, we will present our synthesis results. An ASIC library was used to synthesize RTL code. Due to confidentiality issues, we will not present actual synthesis numbers; Instead, their normalized values or percentage values will be presented. Here, the synthesis result of the developed CNU and that of the estimated SPA LDPC decoder will be discussed. Also a area/delay comparison of different approximations of soft-xor according to section 3.16 is given.

For the Soft-XOR($\boxplus$) function, we have synthesized its 5 different hardware implementations, described in chapter 4. Table 7.1 shows the synthesis area results after normalization, as well as number of basic components used in each. As discussed in section 3.16, they only require adders, comparators and shifters. Among different implementations, the "Min-Sum" soft-xor is taking the least area, followed by the simple "CRI-based" implementation. However, the CRI version demonstrates a much superior BER/SNR performance, as illustrated. The Maximum area is consumed by "double PWL", which uses the complete Equation 4.3 to approximate. On the other hand, "single PWL" version consumes almost half the area of "double PWL", as in this case only two terms of Equation 4.3 were approximated, resulting in Equation 4.5. As a result of this evaluation, we have decided to choose the "CRI" Soft-XOR for CNU implementation. It is also important to note that the "min-sum (1-min)" core function that is reported in this table, is the core function that, if used in a forward-backward architecture, performs the "min-sum" algorithm. However, typical architectures that perform this algorithm, use a core function similar to "Min-Sum (2-mins)" as the core function, because it has to perform two comparisons [4]. It takes 3 inputs and must determine the two minimums. It is also used in the implemented Min-Sum CNU discussed in section 6.1.1.

Table 7.2 shows the synthesis area of the total decoder, with different parts of the implemented CNU. For this table, CN.rate=1, i.e. each CNU has one soft-xor in forward stage and two soft-xors in backward & merge stage; And 81 CNUs are present for this decoder, which consume 88% of the total area, among which most of the area is consumed by Registers and forward stage. It should be noted that the multiplexers and wires necessary for routings of the registers are counted in the forward stage. The implemented permutation network is consuming 7 percent of area, but there are much more advanced methods to design this combinational component. VNU which only consists of two adders and five multiplexers con-

Table 7.1: Area comparison for different implementations of Soft-Xor

Soft-Xor	Comparator	Adder	Area (unit)	Delay (unit)
Min-Sum (1-min)	1	0	13	0.5
Min-Sum (2-min)	2	1	70	0.8
Double PWL[5]	1	6	100	1.15
Single PWL [19]	1	3	72	1.3
low-LLR CRI	2	2	72	1.5
CRI	2	2	55	1

sumes the least hardware area.
As mentioned before, area of every unit is dependent on NOF_BITS. Two synthesis were performed with two different NOF_BITS, which are 7 bits and 5 bits. Obviously this will also have an effect on the decoding performance, which is discussed in section 5. However, by changing NOF_BITS to 5 bits, our hardware cost would reduce considerably. Column 3 in Table 7.2 shows the percentage of decrease in area for 5 bits as compared to 7 bits. The CNU has maximum reduction in area (32%) because of the area of the registers. While for permutation network and VNU, the decrease is 22% and 28% respectively, as none of them have registers, but only adders and multiplexers. The memory area will be saved as well. Although reducing bits increases the error, but 5-bit demonstrates an acceptable error rate, as discuss in section 5.

Table 7.2: Synthesis Area results of implemented flooding SPA-based decoder architecture

Block	Total Area, 7bits (%)	Decrement in Area, 5bits (%)
CNU	88	32
Forward	47	25
Backward & Merge	18	52
Registers	35	30
Permutation Network	4.8	22
VNU	0.2	28
Memory	7	32

In Table 7.3 we present the area distribution in a SPA-based CNU. The complete area is divided into combinational and sequential part. Sequential area is occupied by registers while combinational area consists of soft-xors and remaining logic (Multiplexers and wires). We have swept the CNU.rate, which is a factor of speed of decoder, for values of 1,4 and 8. In case of 1, number of soft-xors used are only 3 i.e. 1 in forward stage and 2 in backward & merge stage. In this case, most of area is consumed by registers and remaining logic, which route the data to the right registers. However, as we increase CNU rate, the soft-xors start to dominate the area. Therefore, for rate 8, soft-xor area overshadows the sequential

area as well as remaining logic area. This implies that at high speeds, the soft-xor approximations in chapter 4 become important. For all the synthesis results, the CRI-based Soft-XOR is used.

Table 7.3: Area distribution for SPA CNU

CNU Rate	Combinational (%)		Sequential (%)
	Soft-Xor(%)	Remaining Logic(%)	
1	16	44	40
4	40	30	30
8	58	19	23

Apart from the synthesis of SPA-based CNU, we have also synthesized a simple but flexible Min-Sum based CNU in 6.1.1. Table 7.4 shows the comparison between these two, while sweeping the CNU.rate (with 7 number of bits). In the Min-Sum CNU, the core function includes 2 comparisons, because two smallest values of the incoming LLRs have to be determined. This CNU is noticeably smaller than SPA CNU, in which a high percentage of area is consumed by a large number of registers (around 49 in this case) and their corresponding wires and multiplexers.

Table 7.4: Area comparison of implemented SPA and estimated Min-Sum CNU

CNU	Combinational (unit)	Sequential (unit)	Total
SPA			
CNU Rate			
1	342	235	577
4	505	235	740
8	765	235	1000
Min-Sum			
CNU Rate			
1	84	41	125
4	234	46	280
8	424	46	470

Finally, the decoder based on the implemented SPA CNU is compared to some other works in Table 7.5. Area is represented as Gate Count and throughput is computed as [24]:

$$Throughput = \frac{codeword}{decodingtime} = \frac{codeword * frequency}{Iteration.no * cycles\,per\,iteration} \tag{7.1}$$

The parameter throughput/area is defined as area efficiency and can be used for comparison. In this table, the SPA decoder is measured for cnu rate of 4. The

properties of SPA decoder for 3 different cnu rates are listed in Table 7.6. As observed, low cnu rates are showing superior performance due to smaller critical path and consequently higher frequency, as cnu rate is the number of Soft-XORs tied in serial. However, this is on the condition that such high frequencies are available. Therefore, for efficient design, cnu rate should be chosen based on the available frequencies.

Table 7.5: LDPC Decoder Comparison

	SPA	[25],2014	[26],2015	[3],2015
Standard	802.11n	802.11ad	802.11ad	802.11n/802.16e
Code Length	648 - 1944	672	672	576-2304
Code-Rate	All	All	1/2	All but 5/6
Sub-block size	27,54,81	42	42	27,54,81
Algorithm	SPA	Min-Sum	Min-Sum	NMS
Schedule	Flooding	Flooding	Layered	Layered
Quantization bits	7	-	-	7
Memory bits	55k	-	-	68k
process(nm)	18	28	28	40
Iteration number	10	∼3.75	3	10
cycles/iter.no	24	-	15-25	30-60
Frequency (MHz)	1250	32-260	400	290
Throughput (Gbps)	10	1.5-12	7	0.47-1.88
Area (Gate count)	1.6M	1.5M	0.28M	2.56M
Throughput/Area	6.3k	1-8k	25k	0.18-0.72k

* For the SPA decoder, the values are estimated based on the implemented CNU with cnu.rate=4 (Memories are estimated as register banks).
** Area is shown as Gate Count, i.e. the synthesis area divided by the area of a NAND2.
*** In [3], both 802.11n and 802.16e are supported, but only the data for 802.11n is mentioned in this table.

7.1 Conclusion

Overall, CNU is taking most of the area of SPA decoder, as it is obvious because of the existence of 49 registers in each CNU. The used memory capacity is very high in comparison to that of registers, but due to the fact that the area cost for RAM is almost $\frac{1}{10}$ of register, area consumption of the memory is only 7% in this design, for cnu rate=1. In CNU, a large amount of area was paid as remaining logic (wires and multiplexers), which are necessary for routing correct data to registers, and also to achieve flexibility for different CNU.rates. Also, area cost for Min-Sum CNU is smaller than SPA CNU for mentioned reasons. When it comes to Soft-Xor implementation, CRI based approximation method is considered as the best

Table 7.6: Decoder with proposed CNU and different speeds

cnu.rate	1	4	8
cycles/iter	96	24	16
Clock Frequency (GHz)	5	1.25	0.625
Throughput (Gbps)	10	10	7.5
Area (Gate count)	1M	1.6M	1.9M
Throughput/Area	10k	6.3k	4k

* Due to difficult RAM handling for cnu rate higher than 1, the memories for cnu rates larger than 1, are measured as register banks.
** Due to clocking constraints, some of the above frequencies may not be allowed.

choice from synthesis point of view, as well as error correction simulations. It is also worth to mention that cnu rate directly affects the critical path and therefore the frequency. Hence, the right choice of cnu rate based on the working frequency leads to optimum design.

Chapter 8
Conclusion

A SPA-based LDPC decoder was evaluated from a hardware perspective. To be able to approximate the algorithm better, an initial study was performed on comparison of different approximation methods (PWL,LUT,CRI), in which PWL proved to be the most hardware-efficient. Also, different mathematical schemes for CNU computation were assessed and $\boxplus$-based forward-backward architecture was selected. For the $\boxplus$ (the core function of CNU), an approximation based on CRI was made that demonstrated the highest efficiency, comparing to other approximations. A flexible and pipe-lined version of forward-backward CNU, based on $\boxplus$ was implemented in 18nm CMOS technology. The number of input tokens per cycle (cnu rate) for the CNU can be easily changed before fabrication. The implemented SPA CNU is 2.1 times larger than an implemented Min-Sum-based CNU, for cnu rate of 8. The discrepancy decreases for higher cnu rates (cnu rate is the number of input tokens in each cycle). Based on the developed CNU, the properties of a total SPA decoder was measured. The SPA decoder demonstrates an area of 1.6M equivalent gate count for a throughput of 10Gbps, with 7-bit quantization, 10 iterations and flooding schedule. A comparison with state-of-the-art shows that SPA decoder is generally larger than its Min-Sum counterparts. In return, the SPA gain in accuracy (BER/SNR) compare to Min-Sum, is $\sim$0.3dB for 10 iterations. The gain can reach $\sim$0.6dB (which is merely 0.1dB less accurate than floating-point SPA) with 20 iterations and a throughput of 5Gbps and the same area. For this study, the IEEE 802.11n WiFi standard was used with a code-length of 1944. This results form a trade-off situation between Min-Sum and SPA decoder and mark the SPA decoder as a legitimate option for applications in which very high accuracies, low BER floor and low dependence on channel properties are required.

8.1 Future Work

The future work suggestions can be listed as the following:

1. Future work could include the RTL implementation of a total SPA-based decoder.
2. In this study, the algorithm stops after a constant number of iterations. However, more advanced stop criteria that can detect when the algorithm

has converged, can increase the throughput by 60% without imposing any increase on cost or delay [28].

3. As the numeric system for SPA, the inverse-LNS, discussed in section 5.3.1, can be explored to see if it can be an efficient alternative to fixed-point system.
4. For high CNU rates, RAM handling becomes difficult, which might justify using Registers instead. One solution could be dividing each element of H matrix into several horizontal sub-elements and dealing with them separately.
5. Most of the SPA CNU area is consumed by pipe-lining registers, to avoid long critical path. incorporating the "$\lambda - min$" algorithm could be beneficial as SPA is performed on only λ number of minimums in a set of data. Therefore SPA can be executed in 1 clock cycle which eliminates the need for pipe-lining registers. Even in case of $\lambda = 2$, the $\lambda - min$ hardware is almost identical to Min-Sum hardware, but $\lambda - min$ is more accurate [29].

References

[1] S. Myung, S. I. Park, K. Kim, J. Li, S. Kwon, J. Kim, "Offset and Normalized Min-Sum Algorithms for ATSC 3.0 LDPC Decoder", 2017, IEEE transactions on Broadcasting.

[2] Y. Xu, L. Szczecinski, B. Rong, F. Labeau, D. He, Y, Wu, W. Zhang, "Variable LLR scaling in Min-Sum Decoding for Irregular LDPC codes", 2014, vol. 60, No. 4, IEEE transactions on Broadcasting.

[3] W. Zhang, S. Chen, X. Bai and D. Zhou, "A full layer parallel QC-LDPC decoder for WiMAX and Wi-Fi," 2015, IEEE 11th International Conference on ASIC (ASICON) in Chengdu.

[4] W. Zhang, S. Chen, X. Bai and D. Zhou, "Approximate Algorithms for Identifying Minima on Min-Sum LDPC Decoders and Their Hardware Implementation," 2015, vol. 62, no. 8, pp. 766-770, IEEE transactions on circuits and systems.

[5] D. Bao, B. Xiang, R. Shen, A. Pan, Y. Chen, X. Y. Zeng, “Programmable Architecture for Flexi-Mode QC-LDPC Decoder Supporting Wireless LAN/MAN Applications and Beyond”, vol. 57, No. 1, 2010, IEEE Transactions on Circuits and Systems.

[6] G. Masera, F. Quaglio, F. Vacca, “Finite Precision Implementation of LDPC Decoders”, 2005, Vol. 152, Issue. 6, IEEE proceedings - communications.

[7] S. Papaharalabos, P. Sweeney, B.G. Evans, P.T. Mathiopoulos, G. Albertazzi, A. Vanelli-Coralli, G. Corazza, “Modified Sum-product Algorithm for Decoding Low-density Parity Check Codes”, 2007, Vol. 1, Issue. 3, IET Communications.

[8] C. Lin, J. Wang, “A bital Circuit Design of Hyperbolic Tangent Sigmoid Function for Neural Networks”, 2008, IEEE International Symposium on Circuits and Systems (ISCAS).

[9] K. Leboeuf, A. Hosseinzadeh Namin, R. Muscedere, H. Wu, M. Ahmadi, “High Speed VLSI Implementation of the Hyperbolic Tangent Sigmoid Function”, 2009, Third International Conference on Convergence and Hybrid Information Technology (ICCIT’08).

[10] P. Meher, “An Optimized Lookup-Table for the Evaluation of Sigmoid Function for Artificial Neural Networks”, 2010, 18th IEEE conference on VLSI System on Chip.

[11] B. Zamanlooy, M. Mirhassani, “Efficient VLSI Implementation of Neural Networks with Hyperbolic Tangent Activation Function”, 2014, IEEE Transaction on Very Large Scale Integration (VLSI) Systems.

[12] B. Parhami, "Computer Arithmetic Algorithms and Hardware Designs", 2000, Oxford University press.

[13] A. Armato, L. Fanucci, G. Pioggia, D. Rossi, “Low-error Approximation of Artificial Sigmoid Function and Its Derivative”, 2009, Electronics Letters.

[14] J. M. Tarela, K. Basterretxea, I. Campo, M. V. Martinez, E. Alonso, “Optimised PWL Recursive Approximation and its Application to Neuro-Fuzzy Systems”, 2002, Elsevier.

[15] K. Basterretxea, J. M. Tarela, I. Campo, “bital Design of Sigmoid Approximator for Artificial Neural Networks”, 2002, Electronics Letters.

[16] R. G. Gallager, “Low density Parity Check Codes”, Cambridge, MIT press, 1963.

[17] J. Hagenauer, E. Offer, L. Papke, “Iterative Decoding of Binary Block and Convolutional Codes”, 1996, Vol. 2, No. 2, IEEE Transactions on Information Theory.

[18] J. Hagenauer, E. Offer, L. Papke, “Efficient Implementation of the Sum Product Algorithm for Decoding LDPC codes”, 2001, Vol. 2, IEEE Global Telecommunication Conference (GLOCOM).

[19] A. Blanksby, C. Howland, “A 690-mW 1-Gb/s 1024-b, Rate-1/2 Low-Density Parity-Check Code Decoder”, 2003, Vol. 37, No. 3, IEEE Journal of Solid-State Circuits.

[20] C. Jones, S. Dolinar, K. Andrews, D. Divsalar, Y. Zhang, W. Ryan, “Functions and Architectures of LDPC Decoding”, 2007, IEEE Information Theory Workshop.

[21] Mohammad. M. Mansour, “A Turbo-decoding message Passing Algorithm for Sparse Parity Check ”, 2006, vol. 54, No. 11, IEEE Transactions on Signal Processing.

[22] F. Zarkeshvari and A. H. Banihashemi, "On implementation of min-sum algorithm for decoding low-density parity-check (LDPC) codes", 2002, vol. 2, No. 1, Global Telecommunications Conference.

[23] L. Sakai, W. Matsumoto, and H. Yoshida, “Reduced complex- ity decoding based on approximation of update function for low-density parity-check codes”, 2007, Vol. J90-A, no. 2, IEICE Transaction.

[24] B. Xiang, D. Bao, S. Huang and X. Zeng, "An 847-955 Mb/s 342-397 mW Dual-path Fully-Overlapped QC-LDPC Decoder for WiMAX system in 0.13 um CMOS", 2011, vol. 46, no. 6, IEEE Journal of Solid-State Circuits.

[25] M. Weiner, M. Blagojevic, S. Skotnikov, A. Burg, P. Flatresse, B. Nikolic, "A Scalable 1.5-to-6Gb/s 62-to-38.1 mW LDPC Decoder for 60GHz Wireless Networks in 28nm UTBB FDSOI", 2014, ISSCC, Session 27, Energy Efficient Digital Circuits.

[26] M. Li, Y. Lee, Y. Huang and L. Perre, "Area and Energy Efficient 802.11ad LDPC Decoding processor", 2015, Vol. 51, No. 4, pp. 339-341, Electronics Letters.

[27] M. Li, J. Weijers, V. Derudder, I. Vos, M. Rykunov, S. Dupont, P. Debacker, A. Dewilde, Y. Huang, L. Perre, W. Thillo, "An Energy Efficient 18Gbps LDPC Decoding Processor for 802.11ad in 28nm CMOS", 2015, IEEE Asian Solid-state Circuits Conference.

[28] T. Chen, "An Adaptive Algorithm and Stopping Criterion for LDPC decoding", Journal of China University of Science, 2011.

[29] E. Boutillon, F. Guillou, J. Danger, "lambda-Min Decoding Algorithm of Regular and Irregular LDPC Codes", 3nd International Symposium on Turbo Codes and Related Topics, 2003, Brest, France.

Printed by Tryckeriet i E-huset, Lund 2017

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2017-600

http://www.eit.lth.se