Lehrstuhl für
Eingebettete Syteme
der Informationstechnik

Design and Implementation of an LDPC-based FEC encoder/decoder suitable for Storage devices

Schriftliche Prüfungsarbeit für die Masterprüfung der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum (Master-Prüfungsordnung für den Studiengang „Elektrotechnik und Informationstechnik" an der Ruhr-Universität Bochum vom 12. August 2013)

vorgelegt von:

Henry Bathke

Datum: November 6, 2018

erster Betreuer:	Prof. Dr.-Ing Michael Hübner
zweiter Betreuer:	M. Sc. Keyvan Shahin

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other Institution of University.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure, that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding.

Datum / Date	Unterschrift / Signature

1. Abstract

Contents

List of Figures

List of Tables

2. Motivation

3. Error Correcting Codes

For modern communications systems reliable data transmission and storage ist required. To achieve this goal usually error correcting codes are used. There are different possible codes available for error correction, but I will restrain myself to LDPC[5] codes in this thesis. As these codes can archive good performance and can be used at large block lengths[14]. This is especially useful for use with NAND based solid state drives.

When describing a block code there are important parameters as the message length k. The message is what is given into the encoder and the result from the decoder. The block length n, and the rate $R = k/n$.

In this case error correction code are used to add additional information to data to allow errors. The errors are then corrected with that information. The addition of additional information is also called redundancy. With this redundancy it is possible to lose information while data is transmitted over a channel and decode it after the channel. After decoding the original data is recovered by using the additional information. figure 3.1 shows such a system where information is transmitted, the channel can be of different type. It can for example be a wireless transmission or memory where information is first stored and later read back.

3.1. Low-Density Parity-Check (LDPC) Codes

The following section will describe LDPC codes invented by Robert Gallager[5]. Starting with a graph representation I will describe the LDPC code and then continue with a matrix representation. LDPC codes can be shown as a bipartite graph also called Tanner graph[15] based on their inventor. figure 3.2 shows an example of one, where the check and parity

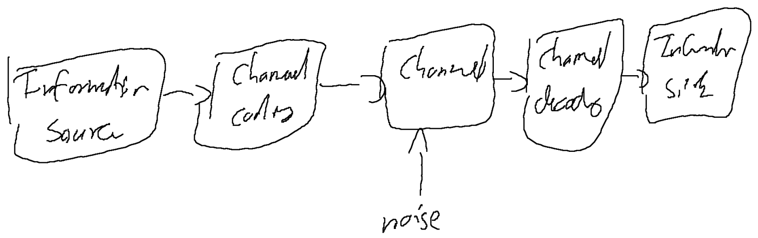


Figure 3.1.: A basic channel where information is transmitted

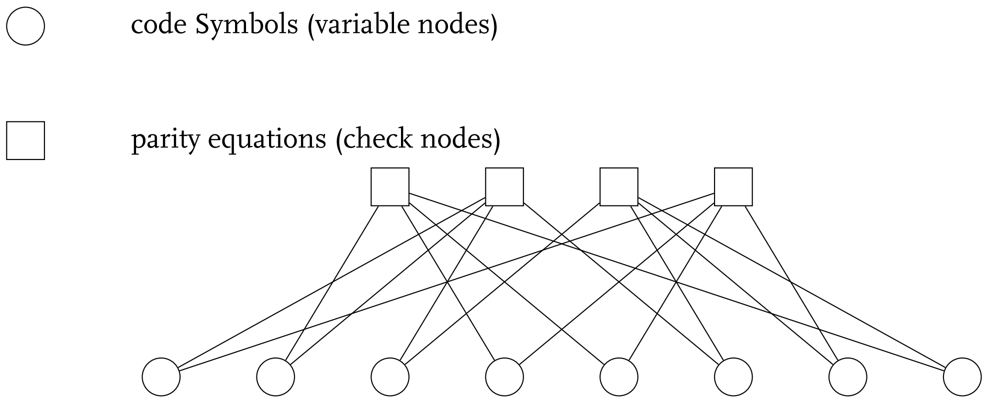


Figure 3.2.: An example Tanner graph.

nodes are connected by edges. This is an effective representation, moreover it will also help understanding the decoding algorithm later.

Instead of using the Tanner graph one can also use a matrix representation. In this matrix the ones represent the edges of the graph. Usually for a LDPC code the matrix is sparse or low density as the name implies. In equation (3.1) a matrix representing the same code as in the graph in figure 3.2 is shown. The $\boldsymbol{H}$ matrix is of size $(n-k)\times n$. And the possible code words are given by the null space of $\boldsymbol{H}$, so in other words c is a code word if and only if $c\boldsymbol{H}^T = \mathbf{0}$[13].

$$\boldsymbol{H} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{3.1}$$

3.1.1. Quasicyclic LDPC (QC-LDPC) codes

LDPC codes can be difficult to implement, especially randomly generated ones. On the other hand structured codes can be devised to be more easily implemented. One class of these structured codes are quasicyclic LDPC codes. In these codes the parity check matrix is only built from circulant matrices and zero matrices[4]. The circulant matrices are defined by their first row as the following rows are the first one shifted. The base matrix specifies where zero matrices and where rotated circulant matrices are placed. Each circulant matrix has size $p\times$. The parity check matrix has size $(n-k)\times n = p(N-K)\times pN$. Thus the base matrix $\boldsymbol{B}$ has $(N-K)\times N$ entries. Often times the circulant matrix is the identity matrix and is rotated by the corresponding amount in the base matrix. In this base matrix the elements can be either a shift factor smaller that the circulant size $0 \leq b_{ij} < p$ or -1 representing a zero matrix.

In the following example the circulant matrix is a 5×5 identity matrix.

$$\boldsymbol{B} = \begin{bmatrix} -1 & 0 & 2 & -1 \\ 0 & 1 & -1 & -1 \\ -1 & 1 & 0 & 2 \end{bmatrix} \tag{3.2}$$

After the expansion the matrix looks like this:

$$\boldsymbol{H} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{3.3}$$

more qc specifics?

3.1.2. Encoding

Genrator Matrix

For encoding the probably simplest algorithm is transforming the parity check matrix into systematic form $\boldsymbol{H} = \begin{bmatrix} -\boldsymbol{A}^T & \boldsymbol{I_{n-k}} \end{bmatrix}$. Where $\boldsymbol{I_{n-k}}$ is a $n-k \times n-k$ identity matrix and $\boldsymbol{A}$ has $k \times n-k$ elements. To archive this form one could for example use gaussian elimination. With $\boldsymbol{A}$ known we can construct the generator matrix $\boldsymbol{G} = \begin{bmatrix} \boldsymbol{I_k} & \boldsymbol{A} \end{bmatrix}$. Now encoding can be done with a simple matrix multiplication. With u the information word and v the code word is given by $v = \boldsymbol{G}u$.

Take for example the matrix from equation (3.1). If we use gaussian elimination to bring the right side to identity we are left with equation (3.4). Now we take the left part of the matrix and transpose it to get $\boldsymbol{A}$. With we build $\boldsymbol{G}$ in equation (3.5).

$$\boldsymbol{H} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

$$\boldsymbol{G} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

The main disadvantage of this strategy is the high computational complexity. When transforming the parity check matrix into systematic form we have a complexity of $\mathcal{O}(n^3)$. This is not to bad as it will mostly be done offline and only the $\boldsymbol{G}$ matrix stored in the encoder, but the bigger problem is that due to the gaussian elimination the matrix is no longer sparse. Thus the matrix multiplication will result in a complexity of $\mathcal{O}(n^2)$[11].

Approximate Lower Triangular Form

Richardson and Urbanke[13] describe a way to reorder the parity check matrix to reduce the encoding complexity. They bring the matrix into a so called approximate lower triangular form. This is done by only doing row and column permutation, so the low density of the matrix is kept. The resulting structure of the matrix is shown in figure 3.3. Especially advantageous is the reduced complexity for encoding, here the encoding complexity is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n + g^2)$, where g is the gap. This gap is the number of rows that cannot be brought into triangular form, as seen in figure 3.3. We can also write

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} & \boldsymbol{T} \\ \boldsymbol{C} & \boldsymbol{D} & \boldsymbol{E} \end{bmatrix} \tag{3.6}$$

the submatrices all have the dimensions given in figure 3.3. By multiplying equation (3.6) with

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} \\ -\boldsymbol{E}\boldsymbol{T}^{-1} & \boldsymbol{I} \end{bmatrix} \tag{3.7}$$

the resulting matrix is

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} & \boldsymbol{T} \\ -\boldsymbol{E}\boldsymbol{T}^{-1}\boldsymbol{A} + \boldsymbol{C} & -\boldsymbol{E}\boldsymbol{T}^{-1}\boldsymbol{B} + \boldsymbol{D} & \boldsymbol{0} \end{bmatrix} \tag{3.8}$$

. By splitting the codeword into three parts $c = \begin{bmatrix} s & p_1 & p_2 \end{bmatrix}$ and applying the definition for valid code words $\boldsymbol{H}^T = \boldsymbol{0}$. It splits into

$$\boldsymbol{A}s^T + \boldsymbol{B}p_1^T + \boldsymbol{T}p_2^T = 0 \tag{3.9}$$

$$\left(-\boldsymbol{E}\boldsymbol{T}^{-1}\boldsymbol{A} + \boldsymbol{C}\right)s^T + \left(-\boldsymbol{E}\boldsymbol{T}^{-1}\boldsymbol{B} + \boldsymbol{D}\right)p^T = 0 \tag{3.10}$$

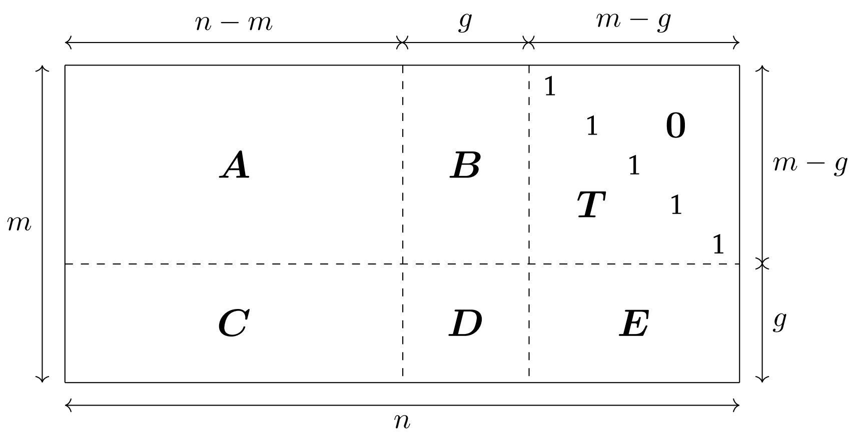


Figure 3.3.: Structure of a matrix in approximate lower triangular form.

Operation	Type
$\boldsymbol{A}s^T$	sparse multiplication
$\boldsymbol{T}^{-1}\boldsymbol{A}s^T$	sparse back substitution
$-\boldsymbol{E}\boldsymbol{T}^{-1}\boldsymbol{A}s^T$	sparse multiplication
$\boldsymbol{C}s^T$	sparse multiplication
$\left(-\boldsymbol{E}\boldsymbol{T}^{-1}\boldsymbol{A}s^T\right)+\left(\boldsymbol{C}s^T\right)$	vector addition
$\boldsymbol{\phi}^{-1}\left(-\boldsymbol{E}\boldsymbol{T}^{-1}\boldsymbol{A}s^T+\boldsymbol{C}s^T\right)$	dense $g\times g$ multiplication

Table 3.1.: Calculations for $p_1^T=\boldsymbol{\phi}^{-1}\left(-\boldsymbol{E}\boldsymbol{T}^{-1}\boldsymbol{A}+\boldsymbol{C}\right)s^T$

. The resulting equations for p_1 and p_2 are

$$p_1^T=-\boldsymbol{\phi}^{-1}\left(-\boldsymbol{E}\boldsymbol{T}^{-1}\boldsymbol{A}+\boldsymbol{C}\right)s^T \tag{3.11}$$

$$p_2^T=-\boldsymbol{T}^{-1}\left(\boldsymbol{A}s^T+\boldsymbol{B}p_1^T\right) \tag{3.12}$$

. The complexity of the computations can be reduced by computing $\boldsymbol{\phi}^{-1}$ offline. Offline meaning that it is precomputed and when encoding multiplying by the matrix. All the other matrix multiplications from equations (3.11) and (3.12) are done separately. Multiplications by $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{E}$ are sparse and the resulting complexity for these is $\mathcal{O}(n)$. The multiplication with $\boldsymbol{T}^{-1}$ is replaced by the system $x^T=\boldsymbol{T}y^T$. As $\boldsymbol{T}$ is a sparse lower triangular matrix the system can be solved by back substitution in $\mathcal{O}(n)$. The only part with higher complexity is the multiplication with the dense $g\times g$ matrix $\boldsymbol{\phi}$ where the complexity is $\mathcal{O}(g^2)$.

do i want do describe the algorithm to get into alt form? yes!

Operation	Type
$\boldsymbol{A}s^T$	sparse multiplication
$\boldsymbol{B}p_1^T$	sparse multiplication
$\left(\boldsymbol{A}s^T\right) + \left(\boldsymbol{B}p_1^T\right)$	vector addition
$-\boldsymbol{T}^{-1}\left(\boldsymbol{A}s^T + \boldsymbol{B}p_1^T\right)$	sparse back substitution

Table 3.2.: Calculations for $p_2^T = -\boldsymbol{T}^{-1}\left(\boldsymbol{A}s^T + \boldsymbol{B}p_1^T\right)$

3.1.3. Decoding

Decoding LDPC codes is a nontrivial problem, in fact maximum likelihood decoding is computationally infeasible. Therefore other decoding methods were developed. There are different decoding algorithms which differ in performance and complexity. I will start with a simple algorithm to introduce the required concepts and then go on to more performant ones.

Hard Descision Decoding

This algorithm works on binary input data and all the messages are also binary. The decoder basically follows the Tanner graph. The messages are passed along the edges and computations are made on the nodes.

1. To initialize all variable nodes have only the information of the received bit. So this is the information that is sent to the adjacent check nodes.

2. Each check node use the information sent to it to check if the bits sent to it are correct. If all check nodes are correct the algorithm terminates. Otherwise the check nodes calculate an answer to each variable node. This answer is computed by assuming that the information from all check nodes except the one the answer is directed to is correct.

3. All variable node use the incoming messages to determine their new value. The new value is for example determined by majority vote of the incoming messages and the current value of the value node.

4. Go to step 2.

Soft Descision Decoding

Instead of working with hard decisions soft decision algorithms work with probabilities. Herein lies the advantage that there is more information given about each bit. So if a probability is close to 0.5 the an error could be more likely. Algorithms like these are also known as belief propagation algorithms.

Sum Product

1. As information all the variable node send the only information they have to the check nodes. The nodes send the probability their bit is "1" to the check nodes. Thsese messages are q_{nm}. They are sent from the nth variable node to the mth check node.

2. The check nodes now calculate the message to each variable node using:

$$r_{mn}(0) = \frac{1}{2} + \frac{1}{2} \prod_{n' \in M(m) \text{without} n} (1 - 2q_{n'm}(1))$$
$$r_{nm}(1) = 1 - rnm(0)$$

   where r_{mn} denotes the message from the mth check node to the nth variable node.

3. Now the variable nodes have all required information to compute their new value.

$$q_{nm} = (1 - y_n)K_{nm} \prod_{m' \in N(n) \text{without} m} r_{m'n}$$
$$a = n$$

   where

decode dat

4. Channel Models

As the encoded message is passed through a channel I will introduce some basic channel models in this chapter. I will only work with binary codes throughout this thesis. For binary codes it is convenient to use $\{+1, -1\}$ as the alphabet. This simplifies the calculations of LLRs likewise it makes the bit energy E_b simple. Also the channel is memoryless, this may sound confusing at first when dealing with storage devices, but it says that each symbol is independently mangled by the channel. The input to the channel is in the input alphabet $\mathcal{X} = \{1, -1\}$. Whereas the the output alphabet $\mathcal{Y}$ will change depending on the channel. Now when transmitting a codeword $c \in \mathcal{X}^n$ the channel outputs $y \in \mathcal{Y}^n$ and it is the receivers task to compute the original codeword c from y. Ideally this is done with few errors and close to the channel capacity.

better word

4.1. Binary Erasure Channel (BEC)

The binary erasure channel is characterized with a single parameter $0 \leq \alpha < 1$ the erasure probability. The symbol for an erasure is $?$, therefore the output alphabet is $\mathcal{Y} = \{1, -1, ?\}$. The channel outputs x with probability $1 - \alpha$ and $?$ with probability α. figure 4.1 shows the transitions for a BEC. So for a codeword with large length n there will be $(1 - \alpha)n$ correct symbols, this suggests that the maximum rate is $1 - \alpha$. Elias[2] shows that this rate can be archived and also proves that this is the capacity.

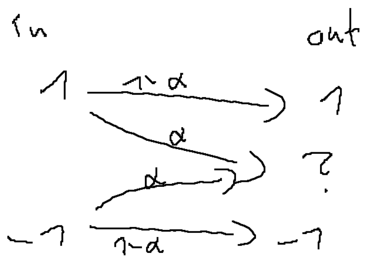


Figure 4.1.: Symmetric binary erasure channel

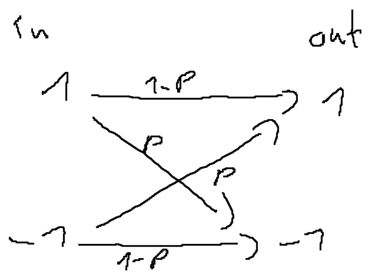


Figure 4.2.: Symmetric binary symmetric channel

4.2. BSC

A binary symmetric channel has the output alphabet $\mathcal{Y} = \{1, -1\}$. An incoming symbol has the probability p to create a crossover. With probability $1 - p$ the symbol is correctly transmitted and with probability p it is flipped. In figure 4.2 a graph shows these probabilities. A binary symmetric channel with crossover probability p has the capacity $1 - H(p)$ with $H(p) = -p \log_2 p - (1 - p) \log_2(1 - p)$ being the binary entropy function.

4.3. Additive White Gaussian Noise Channel

The additive white gaussian noise (AWGN) channel is the most important noise model for me as it characterizes flash memory well. It has a continuos output alphabet $\mathcal{Y}$. The model for the channel is $y = x+z$ where the input is $x \in \mathcal{X}$. z is a variable with normal distribution with 0 mean and variance σ^2. It has the distribution $f(z) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{z^2}{2\sigma^2})$. The capacity for this channel is $\frac{1}{2} \log_2(1 + \frac{1}{\sigma^2})$ as the Shannon limit shows. Often it is preferable to allow scaling of the input, so we use a signal to noise ratio E_b/σ^2. This allows the inputs to be arbitrary values then E_b is the energy of the transmission of a single bit. The σ^2 is frequently called N_0, the energy of the noise that is added in a single bit transmission. When using E_b/N_0 the capacity is $\frac{1}{2} \log_2(1 + \frac{E_b}{N_0})$. For example when having a rate of $\frac{1}{2}$ we get a minimum signal to noise ratio required of 1.

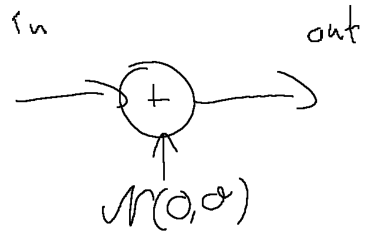


Figure 4.3.: Additive White Gaussian Noise Channel

5. Field Programmable Gate Array (FPGA)

In modern hardware design it is advantageous to have reprogrammable hardware elements. To create reconfigurable logic circuits basic elements consisting of lookup tables (LUT) and registers is built into an array and connected by programmable interconnects.

write some basics about FPGA

6. Flash Memory

Flash memory often in the form of solid state drives (SSD) are becoming more and more of primary storage for computers. The cost per storage has been decreasing steadily therefore driving adoption. Especially the low access latency and the high possible throughput compared so spinning disk hard drives are an advantage. In the application predominantly NAND type flash is used due to its higher density.

6.1. Flash Basics

NAND flash which will be discussed is of most interest for SSDs due to its high density and therefore lower cost per bit. figure 6.1 shows a floating gate transistor which is used to store data. By inserting charge into the floating gate it is possible to change the threshold voltage of the transistor. Multiple of these floating gate transistor are connected in series to for a so called NAND string. In this nand string the drain and source neighboring transistors are connected to form a continuos string as seen in figure 6.2. Also on the top is a normal bit line select transistor to connect the string to be read to a bit line. On the bottom is the ground select transistor to connect the string to ground[10, p. 22-24].

NAND Flash is arranged into blocks and pages. Due to its arrangement NAND memory can only be erased whole blocks at a time. Whereas each page can be programmed individually. When a cell is erased, in it considered "1". On the other hand when programmed it is "0". Programming is done by applying a high voltage to the control gate. This causes electrons from the channel to move through the lower oxide layer into the floating gate. The electrons are now trapped in the control gate under normal conditions. To erase a block the substrate for this block it raised to a high potential to pull the electrons back out of the floating gate[12].

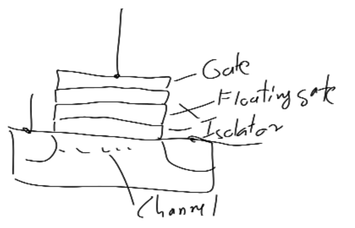


Figure 6.1.: A floating gate field effect transistor

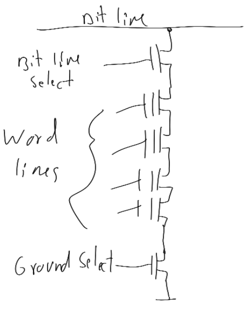


Figure 6.2.: Multiple transistors are arranged to make a NAND string

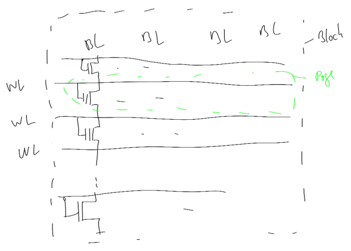


Figure 6.3.: Array Organization of NAND memory

6.1.1. Reading

To read a page the select word line is set at the read voltage. This read voltage is between the threshold voltage for a programmed and unprogrammed cell. Thus if the cell was programmed it will not conduct, on the other hand if it was unprogrammed the applied gate voltage is above the threshold voltage and the cell conducts. In figure 6.4 an example distribution for cell threshold voltages can be seen. The voltage at the bit line is measured using a read amplifier which then outputs the data.

6.1.2. Erasing

Erasing is done by lifting the substrate voltage to to high potential while keeping the gate voltage low. This forces the electrons trapped inside the floating gate through the insulator into the substrate. As the charge is now removed from the gate the threshold voltage is reduced to the erased state. The erase is done a whole block built like figure 6.3 at a time.

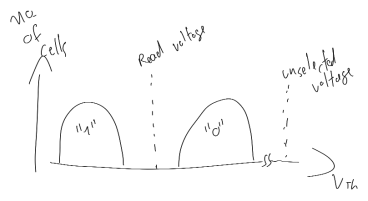


Figure 6.4.: Threshold Distribution of a Floating Gate Transistor[12]

6.1.3. Programming

When Programming the word line for the selected page is set to a high voltage. The bit lines where the cells should be programmed are kept at 0V. All other bit lines are set to high voltage to inhibit them. When the bit line is set to a high voltage the channel of the target cell is also at this voltage so the gate and channel are both at the hight potential.

6.2. Error Types

[18]

7. Approach

7.1. Algorithms

7.1.1. Min Sum Decoding

My architecture is somewhat based on Yanhuan Liu, Chun Zhang, Pengcheng Song, and Hanjun Jiang[7] paper. Although I change the stored values. For a message parsing LDPC decoder the straightforward implementation is to store the messages sent between the check and parity nodes. This results in needing to store multiple values per row of the parity check matrix. For example the LDPC code used for 802.11n with block length 1926 and rate 0.5 has row weights of 7 and 8. This requires to store 8 messages per row of the parity check matrix. The approach I chose is only suited to the min-sum algorithm and will result in a reduction of storage requirements. Instead of splitting the iteration at the message step it is in this case preferable to split at the minimum and the sum for the variable node calculation.

find nice source for 802.11n codes

For decoding the are two message types. The message from the variable nodes to the check nodes q_{nm} and the message going the other way r_{nm}. Decoding is done in different steps[3]:

1. Initialization
   The values from the channel are converted to LLRs y_n. These initial LLR values used as q_{nm} the input to the first check node.

2. Check Node Step
   Each check node m receives the messages from the variable nodes and calculates its response message.

$$r_{mn} = \left(\prod_{n' \in M(m)n} \operatorname{sign}(q_{n'm}) \right) \min_{n' \in vn\{m\}n} (|q_{n'm}|) \tag{7.1}$$

3. Variable node step
   Each variable node n receives the messages from the check nodes and calculates its response message.

$$q_{nm} = y_n + \sum_{m' \in N(n)m} r_{m'n} \tag{7.2}$$

4. Output Decision
   The result LLR values are updated.

$$L_n = y_n + \sum_{m \in N(n)} r_{m'n} \tag{7.3}$$

   And the result is calculated.

$$x_n = \begin{cases} 1, & \text{for } L_n < 0 \\ 0, & \text{else} \end{cases} \tag{7.4}$$

Instead of storing all the messages it is also possible to store the sign, the minimum, and the sum over all messages directed to a column of variable nodes. When storing the minimum it is not enough to just store the minimum. This arises due to the fact that each minimum calculation excludes the current check node. Therefore I store the smallest, the second smallest, and the position of the smallest number. The notation $\min^2$ is for the smallest argument but not including the element min returns. For example if I want to take the min^2 of $\{2, 5, 3, 2\}$ it would result in 2 as the first 2 is "used up" by the min, but there is another 2 available. If I call the minimum s, the second smallest element t, the product of all signs v, and the id k, I get the check node step split into two:

$$s(m) = \min_{n' \in M(m)}(|q_{n'm}|) \tag{7.5}$$

$$t(m) = \overset{2}{\min_{n' \in M(m)}}(|q_{n'm}|) \tag{7.6}$$

$$k(m) = \arg\min(|q_{n'm}|) \tag{7.7}$$

$$v(m) = \prod_{n' \in M(m)} \operatorname{sign}(q_{n'm}) \tag{7.8}$$

And for the check node calculation I can use the results from equations (7.5) to (7.8) to simplify the calculations for each check node.

$$r_{mn} = v(m) \operatorname{sign}(q_{nm}) c_{mn} \tag{7.9}$$

with

$$c_{mn} = \begin{cases} t(m), & \text{for } n = k(m) \\ s(m), & \text{else} \end{cases} \tag{7.10}$$

In the variable node step I instead calculate the sum over all messages:

$$S(n) = y_n + \sum_{m \in N(n)} r_{m'n} \tag{7.11}$$

With the help of this sum I can also calculate the messages from the variable nodes to the check nodes.

$$q_{nm} = S(n) - r_{mn} \tag{7.12}$$

The result step stays the same:

$$x_n = \begin{cases} 1, & \text{for } L_n < 0 \\ 0, & \text{else} \end{cases} \tag{7.13}$$

7.1.2. Normalized Min Sum

In the normalized min sum adaption proposed by Xiaofu Wu, Yue Song, Ming Jiang, and Chunming Zhao[17] the check node step is replaced by the following equation:

$$r_{mn} = \mu \left(\prod_{n' \in M(m)n} \operatorname{sign}(q_{n'm}) \right) \min_{n' \in vn\{m\}n} (|q_{n'm}|) \tag{7.14}$$

Where μ is called a normalization factor. For this factor the exist no analytical results so it has to be determined by simulations.

This changes the calculation for $s(m)$ and $t(m)$ of my version of the algorithm to:

$$s(m) = \mu \min_{n' \in M(m)} (|q_{n'm}|) \tag{7.15}$$

$$t(m) = \mu \overset{2}{\min_{n' \in M(m)}} (|q_{n'm}|) \tag{7.16}$$

7.1.3. Adaptive Normalized Min Sum

The adaptive algorithm changes the normalization factor so it is dependent on some value in the algorithm. That could for example be the iteration number or like in this case the correct check nodes. THe normalization factor μ is replaced by ν where ν is either μ or $\mu\eta$. h_m is the mth row of $\boldsymbol{H}$.

$$\nu = \begin{cases} \mu, & \text{for } h_m^T \cdot z = 1 \mod 2 \\ \mu\eta, & \text{else} \end{cases} \tag{7.17}$$

make image of the internal structure of a check node

Figure 7.1.: An Example Check Node

make image of the internal structure of a variable node

Figure 7.2.: An Example Variable Node

7.1.4. Offset Min Sum

7.1.5. Adaptive Offset Min Sum

7.2. Hardware Adaptation

To build hardware for an LDPC code first I have to decide which approach to take. Either a simpler approach that directly maps all check and variable nodes to hardware or only build part of the nodes in hardware and switch the data to these nodes. First I will show the simpler approach and explain its weakness.

7.2.1. Direct Mapping

When directly mapping the variable and check nodes to the hardware I instantiate all the variable and check nodes and then connect like the tanner graph. With this construction it is possible to generate fast decoders. The tradeoff is that it will produce a lot of hardware for all variable and check nodes. Also the interconnect between the nodes has a high complexity because of the unstructured parity check matrix.

Each check node requires as many inputs as there are ones in each row of the parity check matrix. The same is true for each column of the parity check matrix and the variable nodes. The total amount of check nodes equals the number of rows in the parity check matrix. Also the number of variable nodes is equal to the number of columns in the parity check matrix.

7.2.2. Quasi Cyclic Codes

When dealing with large codes it is preferable to have some structure that can be used to simplify the hardware implementation. Quasi cyclic codes have this property in that each submatrix is a rotated version of the identity matrix. To implement this I use one set of so this should be an image that has a rotated matrix to connect some row of values to the output column and then how rotating and connecting with identity gets the same result.

Figure 7.3.: Rotating and Matrix Connections

variable and check nodes sized to submatrix of the parity check matrix. I will take the matrix form section 3.1.1 and explain how I can use the cyclic structure of the matrix to make the calculations simpler.

To reduce the hardware requirements I use the submatrix structure of the overall parity check matrix. Instead of calculating all check nodes in parallel I compute only a group of the size of a submatrix. For example the first three rows of the matrix in equation (7.18) are calculated in parallel. For each nonzero submatrix of $\boldsymbol{H}$ I take the corresponding variable node values and add these into the state of the currently active check nodes. So in this case only the second and third submatrix are nonzero and only for these are the calculations done.

$$\boldsymbol{H} = \begin{bmatrix} 0&0&0&1&0&0&0&0&1&0&0&0 \\ 0&0&0&0&1&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&1&0&0&0&0&0&0 \\ 0&0&1&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&1&0&0&0&0&1 \\ 0&0&0&0&0&1&0&1&0&1&0&0 \\ 0&0&0&1&0&0&0&0&1&0&1&0 \end{bmatrix} \tag{7.18}$$

The variable node values have to be rearranged depending on the current submatrix. As all the submatrices are shifted identity matrices it is possible to use a barrel shifter to shift all variable node values so that only interconnects for a identity matrix have to be provided. figure 7.3 is an example how the data is first rotated and the passed through an identity matrix to get the same result als with a rotated matrix. I use this for the implementation to only have a single interconnect network representing an identity matrix.

explain roll mess and how to map QC to FPGA

8. Implementation

8.1. Simulation

For simulating the different algorithms I wrote Python scripts. I wrote an encoder using the ALT form. The encoder is a straightforward conversion of the algorithm in section 3.1.2 into python code. The code makes heavy use of numpy and scipy for their matrix manipulation abilities. The sparse matrices from the algorithm are also stored as scipy sparse matrices to reduce the memory usage and speed up the encoding process. The overall encoding is split into two parts the offline part done only once and the ending done for every codeword.

8.1.1. Encoder

The encoder is implemented in 2 different functions. One for the preprocessing where the parity check matrix already in ALT form is split into the required parts and Θ is inverted. The actual encoding function executes the steps from tables 3.1 and 3.2. For the sparse matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{E}$, and $\boldsymbol{T}$ the compressed sparse row matrix format from scipy is used. This format reduced the required memory significantly and accelerates the matrix vector multiplications slightly.

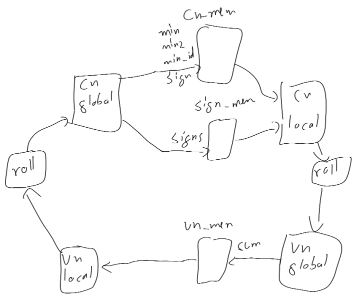
8.1.2. Decoder

I wrote the decoder to represent the way I planned the FPGA implementation. So the overall structure is doing the steps that the VHDL implementation will also do. I designed it in a way that the python functions that are the core of the decoder roughly map to the VHDL entities.

Hardware Overview

I chose to reduce the hardware complexity by splitting both the check node and the variable node step each into a global calculation and a local one. The global check node calculates equations (7.5) to (7.8) and the local computes equation (7.9). The same goes for the variable node where the sum over all inputs as in equation (7.11) and then for the local part the input is subtracted equation (7.12).

WTF is this ordering???



put the ugly paper drawing that i have all the time lying around maybe put the sizes of all dem signals here?

Figure 8.1.: Message and Memory architecture

write something how the thing is done in vivado and u no the block stuff and how the arm cores are used and so on

8.1.3. Results

The simulation implementation of the decoder is written as a Python script I run them and collect the results. To speed up simulation multiple instances of the script are run in parallel using GNU Parallel. This section lists the bit error rates of the different decoder optimizations.

Min-sum

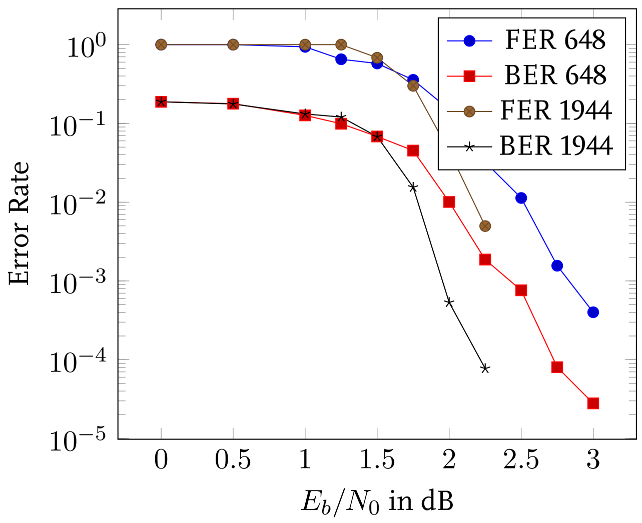
The basic min sum algorithm serves as a baseline for the upcoming improvements. I ran it for two different LDPC codes. Both of them are used by 802.11n for error correction. I chose to use the codes with rate 0.5 as there is a lot of information available.

Normalized Min-sum

Adaptive Normalized Min-sum

I was not able to improve the performance of the system by using the adaptive normalized min sum algorithm. The performance was on par with the normalized min sum and almost independent of the suppression parameter.

optimization with more dimensions?



LDPC codes from 802.11n all with rate 0.5.

Figure 8.2.: Frame and Bit Error Rates of the Basic Min Sum Algorithm

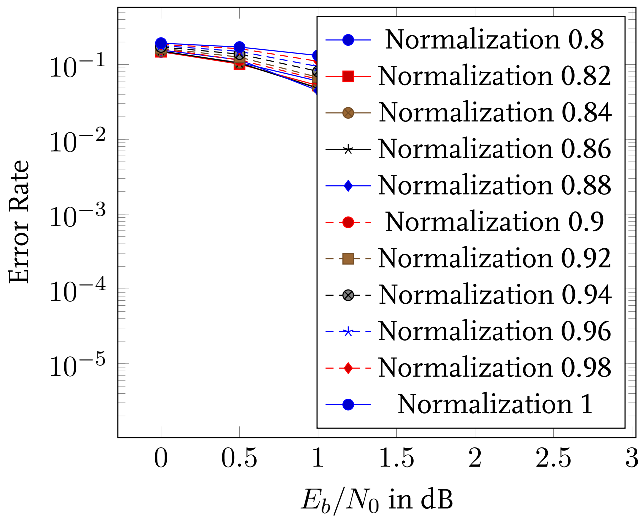


Figure 8.3.: Bit Error Rates of the Normalized Min Sum Algorithm for Different Normalization Factors

8.2. Hardware

For the hardware implementation an FPGA from Xilinx is used. The device is a "ZYNQ™-7000 SOC XC7Z020-CLG484-1". It in a system on a chip consisting of a dual core ARM processor and programmable logic. The decoder is completely implemented in the FPGA logic and the encoder is software running on the ARM cores. I wrote VHDL code for the encoder.

The parameters for the hardware code were generate using Python scripts.

8.2.1. Encoder

The VHDL implementation for the encoder does the sparse matrix multiplications as fully parallel hardware. All bits are computed in parallel and no intermediate register stages are used as the encoder implemented in this way is faster than required for the decoder anyways. The sparse back substitution of $x^T = \boldsymbol{T}y^T$ is done recursively exploiting the advantages of a lower triangular matrix. Any calculation for an output only depends on the inputs and the previously calculated bits. The VHDL code is generated using Python scripts where splitting of the input matrix and the other precomputations are done with the help of numpy. I wrote a simple framework to generate the VHDL code for the matrix multiplications and back substitutions from the precomputed numpy matrices.

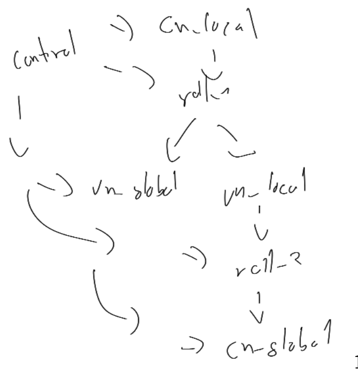
8.2.2. Decoder

The decoder is implemented using the algorithm descibed in chapter 7. This sections discusses how the algorithm is mapped to a hardware platform. As there are parameters that have to be optimized in oder to get decent decoding performance the decoder is designed in such a way that it is easy to change parameters as for example the bit width of the stored values or the used parity check matrix.

make some block diagramm of all my vhdl entities and their connections

Overall the decoder is written in VHDL but on file containing definitions for all signals and the decoder state machine is autogenerated with a python script. In this script it is possible to change the bitwidth of the LLRs and the parity check matrix.

The decoder is controlled by a state machine which reads the "instruction list" and outputs the control signal and the memory read and write addresses. This instruction list is also created by the decoder python script. There it is also possible to change the clock cycle in which each signal appears. This makes it easy to change the pipelining. figure 8.4 shows how the message LLR values are passed along the entities. Each iteration of the decoder is



maybe add the min mangle at the end

Figure 8.4.: Dataflow Diagram of the Decoder

split into two parts. The first part calculated the global check nodes results and the second part the global variable node results.

I will start with the global variable node pass as this has fewer steps and some of the steps are reused in the check node pass. First the local check node results are calculated. This is done from the stored minimum, second minimum, minimum id, and the signs. The output is either the minimum if the minimum id is not equal to the current position or otherwise the second minimum. The sign for this output is taken from the sign input, as all minimums are absolute values. Now the messages are passed through a barrel roll to shift the values to the appropriate positions. The rolled value is then accumulated for each column and at the end of the column stored.

The global check nodes pass starts the same as the global variable node one until the barrel roll. After rolling the LLRs are passed into the local variable node entity. The local variable node subtracts the incoming LLR from the sums the variable nodes calculated and outputs it. Now the LLRs are passed through a reverse roll to align them for the global check node. The global check node then does an accumulation. This consists of taking the minimum, second minimum, and the signs for the minimum. Also the signs of all incoming LLR values are output.

8.2.3. Control

The decoder is controlled by a state machine which reads the addresses for the memory and all control signals from a ROM. It als keeps track of the number of iterations already done and terminates if the iteration number reaches a specified maximum. Additionally the state machine stops the decoding process if the output vector is error free. The instructions consisting of addresses and control signal are generated by the Python script which also generates the other constants.

LLR from min schematic maybe?

Figure 8.5.: Schematic Diagram of a Local Check Node

min entity schematic maybe?

Figure 8.6.: Schematic Diagram of a Global Check Node

8.2.4. Check Nodes

As the check nodes are split into two steps I wrote two separate entities executing these operations. One is the local check node, computes the message LLR from the minimums, minimum sign, and LLR signs. From the controller it receives the current offset.

8.2.5. Variable Nodes

Also the variable nodes are split into two. The global pass sums all the incoming LLR values and at the end of each column it is stored into memory. The local check node retrieves these sums and calculates

$$q_{nm} = S(n) - r_{mn} \tag{8.1}$$

, where $S(n)$ is the stored column sum, r_{mn} and q_{nm} are the incoming and outgoing LLRs respectively.

8.2.6. Barrel Roll

I started first with a naive implementation for the barrel roll

```
    entity dynamic_roll_sign is
    generic (
        DIRECTION : boolean --true means the same direction ...
            as fixed roll
    );
    port (
        roll_count : in unsigned;
        data_in : in min_signs_t;
        data_out : out min_signs_t
    );
end entity;

architecture base of dynamic_roll_sign is
```

Algorithm	27 Block Length	200 Block Length
Modulo	5132	101533
Sub	540	28229
MUX2	473	5600

Table 8.1.: LUT Utilization of Different Barrel Roll Implementations

```
begin
    gen_i : for i in data_in'range generate
    begin
        data_out(i) <= data_in((i - to_integer(roll_count)) ...
            mod data_in'length) when not DIRECTION
             else data_in((i + to_integer(roll_count)) mod ...
                data_in'length));
    end generate;
end architecture;
```

but this generates huge hardware. The primary problem is the synthesis generates a division to implement the modulo operations. In the following I will show that the modulo can in this case be replaced by a conditional addition or subtraction. But first I have to set some limits for the inputs. I only allow roll values in the range $0 \leq$ `roll_count` $<$ `data_in'length`. From that and the possible values for i I get $0 \leq +$ `roll_count` < 2 `data_in'length`. Knowing this I can replace the modulo with a conditional addition or subtraction depending on the shift direction. So the lines writing to `data_out` are replaced by:

```
data_out(i, j) <= data_in(add_mod(i - to_integer(roll_count), ...
    data_in'length), j) when not DIRECTION
    else data_in(sub_mod(i + to_integer(roll_count), data_in'length), j);
```

Even more efficient in resource usage is a logarithmic barrel shifter. In this design I have shifts by powers of two and either use each shift or bypass it depending on the `roll_count` bits. In figure 8.7 you can see a block diagram of such an architecture. This design only requires wires and two input muxes. On the used FPGA each LUT can implement a 4 input LUT or multiple LUTs can work together to create muxes with more inputs. I tried the architecture with larger muxes but archived no smaller design. This optimization was probably already done by the synthesis tools and doing it manually did not change the resource usage. The differece between the barrel shifters is more pronounced at larger block sizes as is visible in table 8.1.

requrce usage!!

In table 8.1 you can see that for the block length of 200 it is not feasible to use the worse algorithm. And modern codes such as one proposed for NG-EPON[8] and DVB-S2[1].

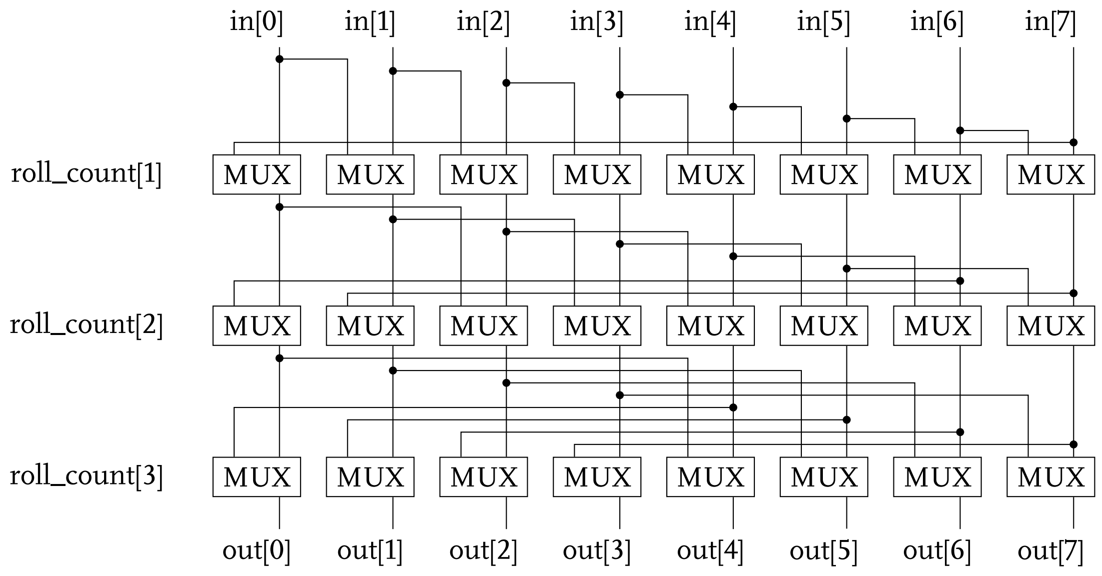


Figure 8.7.: Structure of a Logarithmic Barrel Shifter

8.2.7. Interface

I used AXI-Stream interfaces to input the LLR values into the decoder and to output the decoded bits. These interfaces transfer the bits tightly packed to maximize throughput. The stream interface is a good fit for this application as decoding is done on full frames and the is no need to change single words after a transfer. Also it is fast to implement.

8.2.8. Pipelining

If implemented without any pipeline steps the maximum clock for the 648 bits message length, rate 0.5, and 7 bit LLR is 27Mhz. By adding registers between some steps I reduce the critical path. By adding two stages I increase the maximum clock rate to 70Mhz. The pipeline stages are added between the local check node and the first barrel roll and after the second barrel roll. These position were determined by check the critical path timing in Vivado and adding registers to achieve a maximum critical path length of 10ns as to get a 100Mhz clock.

check real val

test

i prolly need one after the global check node to get the memory timing better

8.2.9. Possible Improvements

More pipelining In and output double buffering More pipeline stages u no dependency optimization

8.3. Data Source

For evaluating the whole pipeline a source of data to be encoded is required. I used a uniform random number generator. The numbers were generated using a Tausworthe generator[9]. I implemented it in a straightforward manner in VHDL. There were no difficulties as it consists of static shifts and bit boolean operators.

8.4. Channel Model

To test an error correction scheme a source of errors is required. This would usually be the flash memory in an solid state drive, but here I simulate it using gaussian noise. The gaussian noise is generated with the Box-Muller method using the hardware approach from Dong-U Lee, John D. Villasenor and Wayne Luk and Philip H. W. Leong[6]. Here I used Vivado HLS to create the hardware. The Box-Muller method requires fixed point numbers and with high level synthesis it is very fast to write the code for such a noise generator.

My tests concluded the Box Muller method is to hardware intensive when build with high level synthesis. I used a very efficient design by David B. Thomas[16] which uses less resources of the FPGA. The class of gaussian number generators described in Thomas paper have different qualities depending on the internal bitwidths. I chose the one with the highest quality as it is smaller than my Box-Muller implementation. To interface with my existing design I wrote an axi stream frontend to simply replace the other number generator. The Box-Muller method used for example 6845 LUTs compared to 683 LUTs for this method.

8.5. Result Accumulation

8.6. Control

The whole system is controlled by the ARM core of the SoC. All the inputs are circuitry

9. Results

A. Appendix

Bibliography

[1] *Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2)*. ETSI EN 302 307. V1.2.1. ETSI. 2009.

[2] P. Elias. In: (1955).

[3] A. A. Emran and M. Elsabrouty. "Simplified variable-scaled min sum LDPC decoder for irregular LDPC codes". In: *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*. 2014, pp. 518–523. DOI: `10.1109/CCNC.2014.6940497`.

[4] M. P. C. Fossorier. "Quasicyclic low-density parity-check codes from circulant permutation matrices". In: 50.8 (2004), pp. 1788–1793. DOI: `10.1109/TIT.2004.831841`.

[5] Robert R. Gallager. "Low-Density Parity-Check Codes". In: (1963).

[6] Dong-U Lee et al. *A hardware Gaussian noise generator using the Box-Muller method and its error analysis*. 2006. DOI: `10.1109/TC.2006.81`.

[7] Y. Liu et al. "A high-performance FPGA-based LDPC decoder for solid-state drives". In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2017, pp. 1232–1235. DOI: `10.1109/MWSCAS.2017.8053152`.

[8] Zheng Liu, Junwen Zhang, and Jun Shan Wey. *FEC for Upstream: 8K LDPC Code*. Tech. rep. 2018. URL: `http://www.ieee802.org/3/ca/public/meeting_archive/2018/01/wey_3ca_1a_0118.pdf`.

[9] PIERRE L'ECUYER. "MAXIMALLY EQUIDISTRIBUTED COMBINED TAUSWORTHE GENERATORS". In: *MATHEMATICS OF COMPUTATION* 65.213 (1996), pp. 203–213.

[10] Rino Micheloni, A. Marelli, and Kam Eshghi. *Inside Solid State Drives (SSDs)*. 2nd ed. Vol. 37. Springer, 2018. ISBN: 9789811305986, 9789811305993. DOI: `10.1007/978-981-13-0599-3`.

[11] Hanghang Qi and Norbert Goertz. "Low-Complexity Encoding of LDPC Codes: A New Algorithm and its Performance". In: ().

[12] B. Ricco et al. "Nonvolatile multilevel memories for digital applications". In: *Proceedings of the IEEE* 86.12 (1998), pp. 2399–2423. ISSN: 0018-9219. DOI: `10.1109/5.735448`.

[13] Thomas J. Richardson and Rüdiger L. Urbanke. *Efficient Encoding of Low-Density Parity-Check Codes*. 2001. DOI: `10.1109/18.910579`.

[14] Bashar Tahir, Stefan Schwarz, and Markus Rupp. "BER comparison between Convolutional, Turbo, LDPC, and Polar codes". In: (2017). DOI: `10.1109/ICT.2017.7998249`.

[15] M. Tanner. “A recursive approach to low complexity codes”. In: (1981). DOI: 10.1109/TIT.1981.1056404.

[16] D. B. Thomas. “FPGA Gaussian Random Number Generators with Guaranteed Statistical Accuracy”. In: (2014), pp. 149–156. DOI: 10.1109/FCCM.2014.47. URL: http://cas.ee.ic.ac.uk/people/dt10/research/rngs-fpga-normal.html.

[17] X. Wu et al. *Adaptive-Normalized/Offset Min-Sum Algorithm*. 2010. DOI: 10.1109/LCOMM.2010.07.100508.

[18] S. A. A. Zaidi et al. “FPGA Accelerator of Algebraic Quasi Cyclic LDPC Codes for lt;sc gt;nand lt;/sc gt; Flash Memories”. In: *IEEE Design Test* 33.6 (2016), pp. 77–84. DOI: 10.1109/MDAT.2015.2497322.