Contents

List of Figures

List of Tables

1. Implementation

1.1. 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 ?? 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.

1.1.1. 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 ????. 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.

1.1.2. 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

WTF is this ordering???

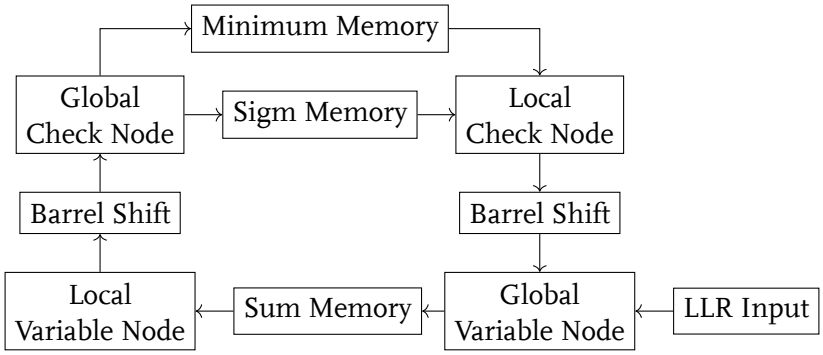


Figure 1.1.: Message and Memory architecture

node step each into a global calculation and a local one. The global check node calculates ???????? and the local computes ??. The same goes for the variable node where the sum over all inputs as in ?? and then for the local part the input is subtracted ??.

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

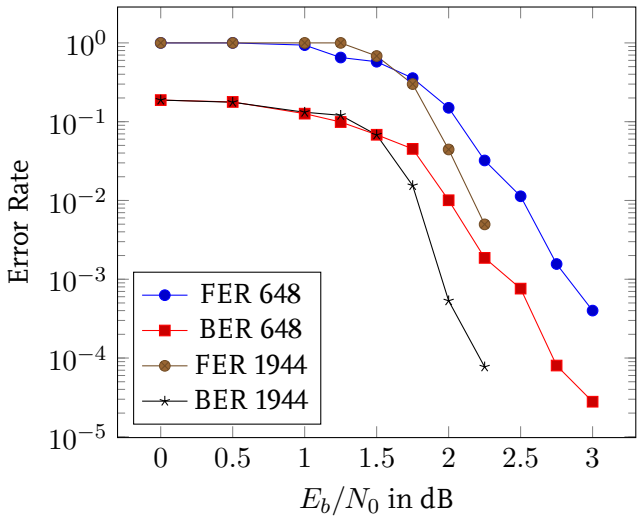
1.1.3. Results

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.

figure 1.2 shows performance of the basic min sum decoder. Here the waterfall region of the LDPC decoder is visible. I was not able to gather performance data for lower error rates because the software implementation is not optimized for speed.



LDPC codes from 802.11n all with rate 0.5.

Figure 1.2.: Frame and Bit Error Rates of the Basic Min Sum Algorithm

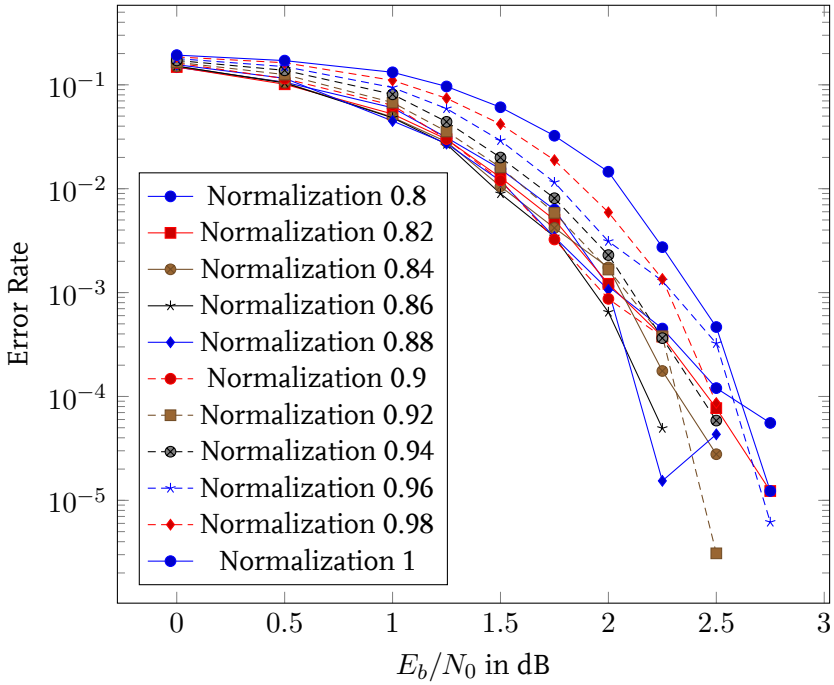


Figure 1.3.: Bit Error Rates of the Normalized Min Sum Algorithm for Different Normalization Factors

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?

1.2. 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.

1.2.1. 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.

1.2.2. Decoder

The decoder is implemented using the algorithm descibed in ??. This sections discusses how the algorithm is mapped to a hardware platform. As there are parameters that have to

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

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.

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 1.4 shows how the message LLR values are passed along the entities. Each iteration of the decoder is 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.

1.2.3. 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. In my script to generate the instructions I

continue with something about how great my fucking subroutine with a weee bit of string manipulation is

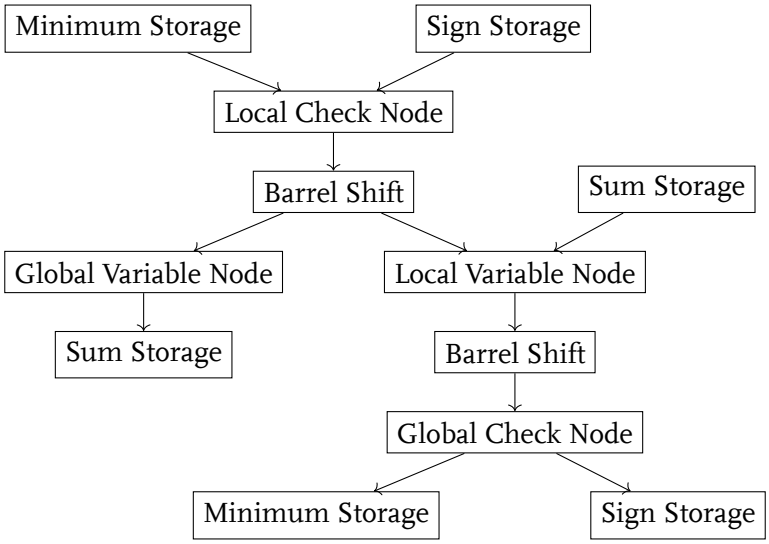


Figure 1.4.: Dataflow Diagram of the Decoder

LLR from min schematic maybe?

Figure 1.5.: Schematic Diagram of a Local Check Node

1.2.4. 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.

1.2.5. 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

$$q_{nm} = S(n) - r_{mn} \tag{1.1}$$

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

min entity schematic maybe?

Figure 1.6.: Schematic Diagram of a Global Check Node

1.2.6. Barrel Roll

I started first with a naive implementation for the barrel roll

```
1     entity dynamic_roll_sign is
2     generic (
3         DIRECTION : boolean --true means the same direction ...
              as fixed roll
4     );
5     port (
6         roll_count : in unsigned;
7         data_in : in min_signs_t;
8         data_out : out min_signs_t
9     );
10 end entity;
11
12 architecture base of dynamic_roll_sign is
13 begin
14     gen_i : for i in data_in'range generate
15     begin
16         data_out(i) <= data_in((i - to_integer(roll_count)) ...
               mod data_in'length) when not DIRECTION
17              else data_in((i + to_integer(roll_count)) mod ...
                   data_in'length));
18     end generate;
19 end architecture;
```

but this generates huge hardware. One of the primary problems 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:

```
1 data_out(i, j) <= data_in(add_mod(i - to_integer(roll_count), ...
      data_in'length), j) when not DIRECTION
2     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 1.7 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 in-

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

Table 1.1.: LUT Utilization of Different Barrel Roll Implementations

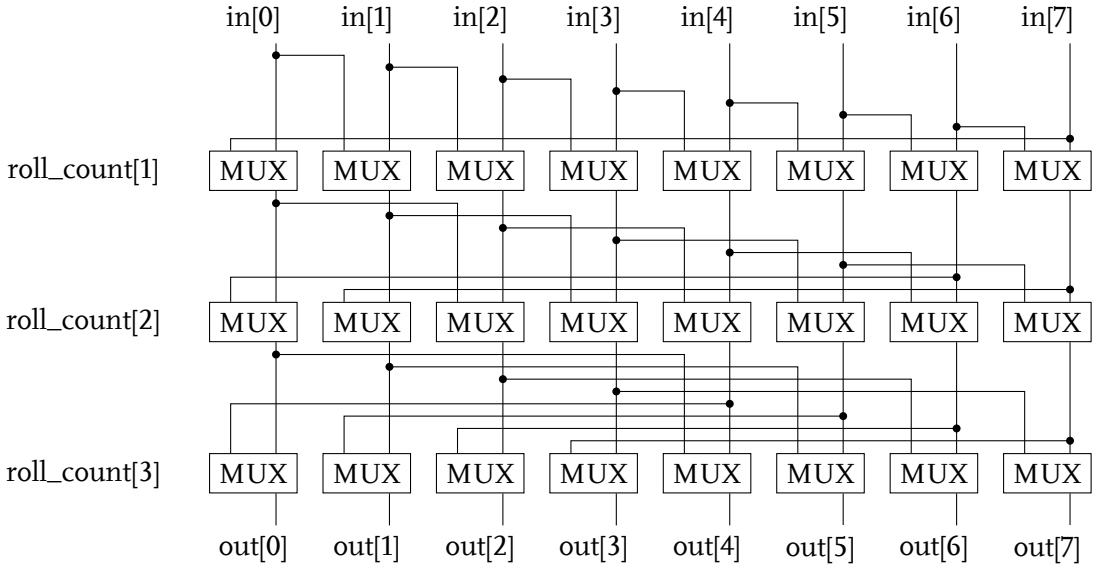


Figure 1.7.: Structure of a Logarithmic Barrel Shifter

put 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 1.1.

requrce usage!!

In table 1.1 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[3] and DVB-S2[1].

1.2.7. 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.

1.2.8. 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

1.2.9. 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[4]. I implemented it in a straightforward manner in VHDL. There were no difficulties as it consists of static shifts and bit boolean operators.

1.2.10. 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[2]. 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[5] 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 still 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. The output from the generator is a 26 bit fixed point number with 6 bit integer part.

1.2.11. LLR Conversion

The LLR calculation is done as descibed in **??**. The hardware block for the LLR takes y_i and $\frac{2}{\sigma^2}$ is precomputed on the ARM core and then sent as a 32 bit fixed point number with 5 bit integer part to the programmable system. The output of the channel is of bit width to that no rounding or overflow can occur. The result of this conversion is then rounded to the number of bits the encoder is using.

1.2.12. Result Accumulation

The results of each decoding are compared to the input to the channel. As the output of the decoder is a 32 bit axi stream multiple word have to be compared for each decoding. The Hamming distance between the expected and decoded codeword is summed over all the words within one frame and then sent to ARM core.

1.2.13. Control

The whole system is controlled by the ARM core of the SoC. The parameters of the test system are set first by the ARM and then the encoder decoder core is set to run. While it is running the bit error counts of each transmission are read out and summed. As each frame is handled separately by the result accumulator a frame error can be detected easily by checking if the bit error count for a frame differs from zero. This split in functionality simplifies testing for different parameters enormously because new software for the cpu can be written and compiled very quickly compared to synthesizing VHDL code for the FPGA part.

1.2.14. Results

The complete system was implemented on a ZedBoard with the ARM cpu controlling the test parameters for the encoder and decoder. table 1.2 shows the resource utilization of the project. In this case I used the decoder with 7 Bit LLR. Two decoders were implemented together on the FPGA to achieve higher throughput while testing. To achieve optimal performance of the decoder the input parameters have to be optimized. Especially the normalized min sum decoder is sensitive to the Normalization parameter. Therefore it is required to test the decoder with different parameters sets to find optimal decoding conditions. So I ran different tests first to find the best input scaling factor. For that I did a coarse sweep over the available number space. From figure 1.8 I deduce that the scaling factor should be small

	Overall	Codec Block	Decoder	AWGN Generators
Slice LUTs	34387 (64.64%)	16537 (31.08%)	8706 (16.36%)	1411 (2.65%)
Block RAM	32 (22.86%)	15.5 (11.07%)	13.5 (9.64%)	2 (1.43%)
DSP	32 (14.55%)	16 (7.27%)	0	8 (3.64%)

Table 1.2.: Overall Hardware Usage of the Test System Post Implementation

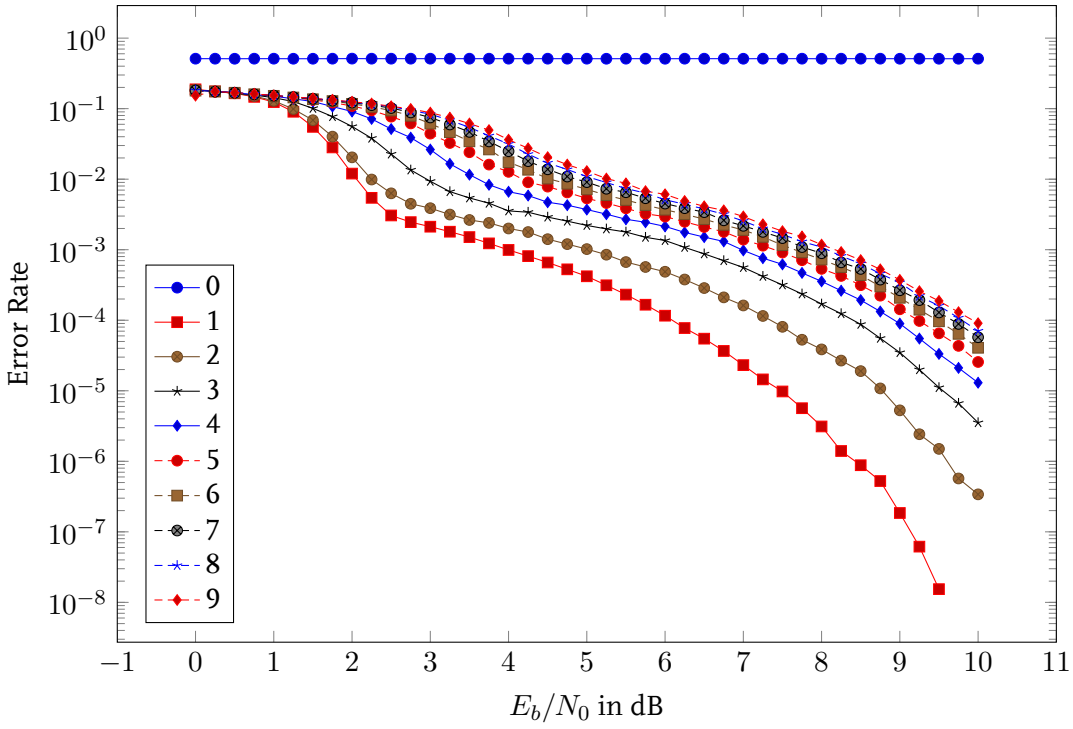


Figure 1.8.: Sweeping the Input Scaling Factor with 802.11 LDPC code 648 Bit block length and rate 0.5

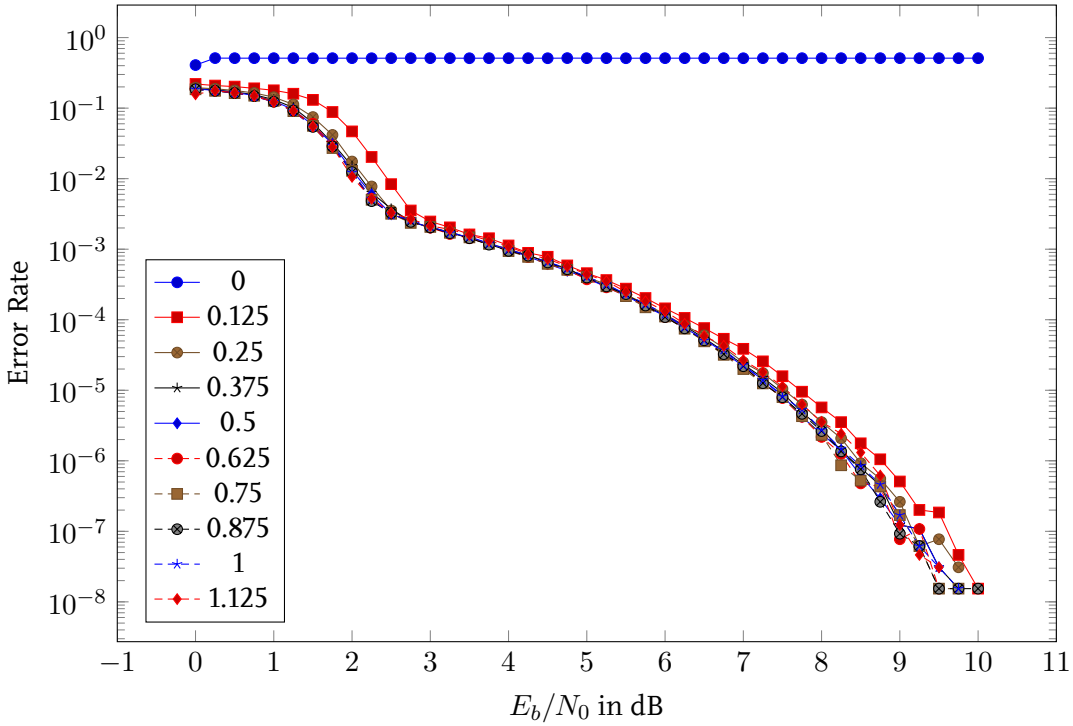


Figure 1.9.: Sweeping the Input Scaling Factor with 802.11 LDPC code 648 Bit block length and rate 0.5

so to optimize it further I swept a smaller range with more values. Even though figure 1.8 only shows values up to 9 the sweep continues up to the maximum input but there is no other optimum in the error performance. So the smaller range results in figure 1.9 show the optimal value being 0.75 for this system. So in the further analysis this is the value I used as the scaling factor. As

1.2.15. Possible Improvements

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

2. Results

A. Appendix

attach my files as a zip or something like that

Bibliography

[1] *Digital Video Broadcasting (DVB); Second generation framing structure, channelIn: (). M. Tanner. "A recursive approach to low complexity codes". In 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] 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.

[3] 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.

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

[5] 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.