Low Density Parity Check Code Implementation

Senior Project Final Report

Students:
Matthew Pregara
Zachary Saigh

Advisors:
Dr. In Soo Ahn
Dr. Yufeng Lu

Department of Electrical and Computer Engineering
Bradley University
05/2013

Table of Contents

Abstract 5

1. Introduction 6

1.1 Objective 7

1.2 Project Description 7

1.3 Linear Block Coding 8

1.4 LDPC Codes 10

1.5 Log-Likelihood-Ratio Algorithm 11

1.6 Functional Requirements 14

2. Project Approach 15

2.1 Simulink Design 15

2.2 Pre-Implementation Challenges 15

2.3 Simulink Encoder Implementation 17

2.4 Xilinx Encoder Implementation 19

2.5 Simulink Decoder Implementation 20

2.6 Constructing a Lookup Table 22

2.7 Simulation Results 23

3. Conclusion 38

3.1 Future Work 38

Appendix A 41

List of Figures and Equations

Figures

Figure 1: FEC Code Performance Comparison 6
Figure 2: High level LDPC System Block Diagram 7
Figure 3: Hamming Code Block diagram 8
Figure 4a: Encoding for Linear Block Codes 9
Figure 4b: Modulo-2 addition used for parity bit calculations 9
Figure 5: Tanner Graph with corresponding H matrix 11
Figure 6: Tanner graph for a (10,5) code with received values 12
Figure 7: Calculation for Edge between V_1 and C_1 13
Figure 8: BER Comparison between different Iteration Numbers 14
Figure 9: Simulink LDPC System Model 15
Figure 10: Low level Simulink Encoder 17
Figure 11: Low Level Encoder Verifcation 18
Figure 12: Xilinx Encoder 19
Figure 13: Xilinx Encoder Verifcation 20
Figure 14: Low Level Simulink Decoder 21
Figure 15: Phi(x) Lookup Table Comparison 22
Figure 16: Lookup Table Performance Comparison 23
Figure 17: Complete Lookup Table Function Block 24
Figure 18: Lookup Table Output 25
Figure 19: Lookup Table Memory 25
Figure 20: Index Cap Block 26
Figure 21: Lookup Index Number 26
Figure 22: Absolute Value Function 27
Figure 23: Edge Calculation Block 27
Figure 24: Edge Cycle Through System 28
Figure 25: Max Count Block 29
Figure 26:Product/Sign Block 30
Figure 27: Summation Block 30
Figure 28: Edge Calculation Blocks that make up a Node 31

Figure 29: Total Check Node View 32
Figure 30: Zoomed-In Check Node View 32
Figure 31: Full Decoder in Xilinx........ 33
Figure 32: Variable Node Calculation Blocks 34
Figure 33: Variable Node Calculation Block Internal View 35
Figure 34: Complete Decoder System 36
Figure 35: Received Codeword Block Internals 37
Figure 36: Control/Timing Block 38

Equations

Equation 1: Phi Function........ 13
Equation 2: Square Add Function 16
Equation 3: Max* Function........ 16

Abstract

In communication systems, forward error correction (FEC) codes have been widely used to battle data errors caused by transmission through a noisy channel. By adding extra bits to the end of message bits, a certain number of bit errors can be detected and corrected without frequent retransmission. Low density parity check (LDPC) is a powerful FEC coding scheme which can achieve good error performance under very low signal-to-noise ratios [1]. A communications system utilizing LDPC code is able to get very close to the channel capacity limit established by Claude Shannon in the 1940's. In addition, LDPC codes have lower complexity in the decoding process compared to other FEC codes. With advances in computing power, LDPC codes have been adopted in many high speed communication standards such as digital video broadcasting, WiMAX, 4G wireless systems, among others.

This project explores the process as well as the feasibility of implementing the design of a LDPC system on a field programmable gate array (FPGA). The Xilinx System Generator block set is used in conjunction with MATLAB and Simulink for FPGA co-simulation.

1. Introduction

In digital communication systems, there are two different ways commonly used to reduce errors caused by a noisy channel. One is called Automatic Repeat request (ARQ). With ARQ, a receiver is able to detect if a message was received in error. If an error occurred, the receiver sends a request back to the transmitter to repeat the message that was in error [1]. This method of error correction requires a two-way channel and is not a feasible method for one-way systems such as digital video broadcasting.

Another way to reduce errors in a communication system is to use a Forward Error Correction (FEC) scheme. The FEC scheme adds redundant information to a message in the form of extra bits called parity bits. Using this redundant information, the receiver is able to detect and correct a message without requesting a retransmission [1].

Low density parity check (LDPC) codes are a powerful FEC coding scheme that can achieve good error performance under very low signal-to-noise ratios. A communication system utilizing an LDPC code is able to operate very close to the channel capacity limit established by Claude Shannon in the 1940's [2]. Figure 1 compares the performance of various coding schemes used in the communication industry. It shows that LDPC codes achieve the same code rate of 0.5 as many other codes. It can operate close to the Shannon Capacity Bound in a lower signal power environment.

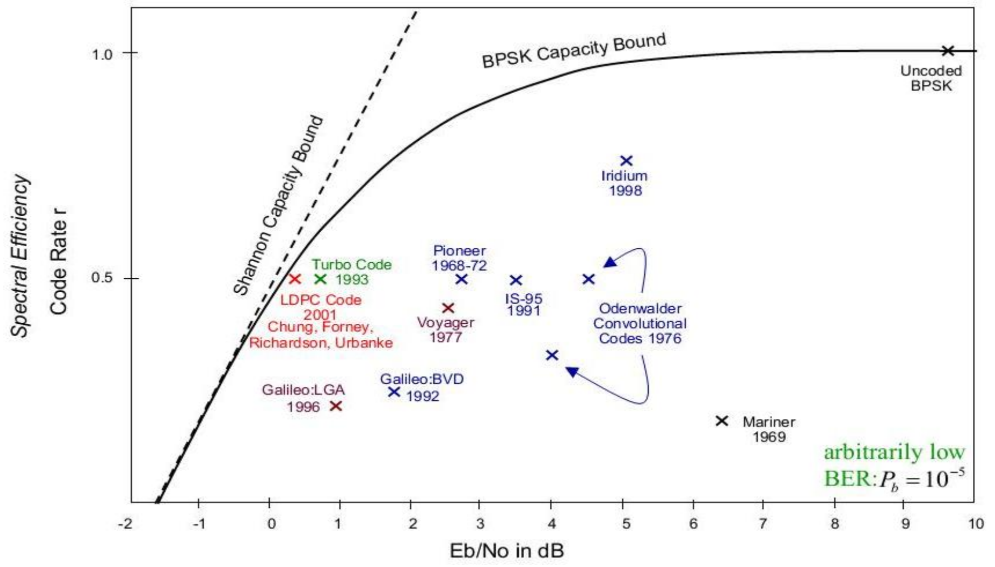


Figure 1: FEC Code Performance Comparison [3]

In addition to their good performance, LDPC codes have lower complexity in the decoding process compared to other FEC codes such as Turbo codes [3]. LDPC codes were developed in 1960 by Robert G. Gallager at MIT. With recent advances in parallel computing power, LDPC codes have been re-discovered and studied. They are used in many high speed communication standards such as digital video broadcasting, WiMAX, 4G wireless systems, to name a few [2].

1.1 Objective

The project aims to explore the use of Field Programmable Gate Arrays (FPGAs) in the encoding and decoding of messages using an LDPC FEC scheme. Decoding algorithms used for LDPC codes require large amounts of data to be processed in a short time. For this reason, the ability of FPGAs to process large amounts of data in parallel is highly desirable in real time communication systems.

Using the Xilinx System Generator tools along with Simulink, an LDPC system could be quickly prototyped on FPGAs. Xilinx System Generator tools allow developers to construct systems using specialized blocksets within the Simulink environment.

1.2 Project Description

This project was initially broken down into three phases. In phase 1, the goal was to gain a good understanding of LDPC codes and decoding techniques by implementing a system simulation using MATLAB code as well as a Simulink model. Figure 2 depicts a high level block diagram of a coding system.

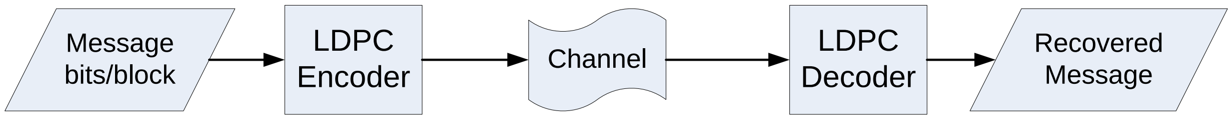


Figure 2: High level LDPC System Block Diagram

In phase 2, the goal was to break down the Simulink model into the most fundamental blocks available. It would allow for an easier transition, that is, by using Xilinx System Generator blocks, to construct an encoder and decoder for FPGA implementation. The Simulink model created in phase 1 is used to verify the design in Xilinx System Generator. Finally, in phase 3 of this project, the Xilinx encoder and decoder developed in phase 2 would be ported to an FPGA. The performance of the FPGA LDPC code system will be compared with simulation results from phase 1. Specifications such as bit rate, channel bandwidth requirements, and error detection capability will then be analyzed.

LDPC codes with a larger size have a better error correction performance with much higher cost of hardware implementation. In this project, LDPC codes with a relatively small block size are chosen for System Generator design on FPGA.

1.3 Linear Block Coding

LDPC codes stem from another type of FEC scheme called linear block codes [4]. To describe how LDPC codes operate, consider how typical linear block codes operate. Hamming code is given as an example. Figure 2 illustrates how a Hamming Code system operates.

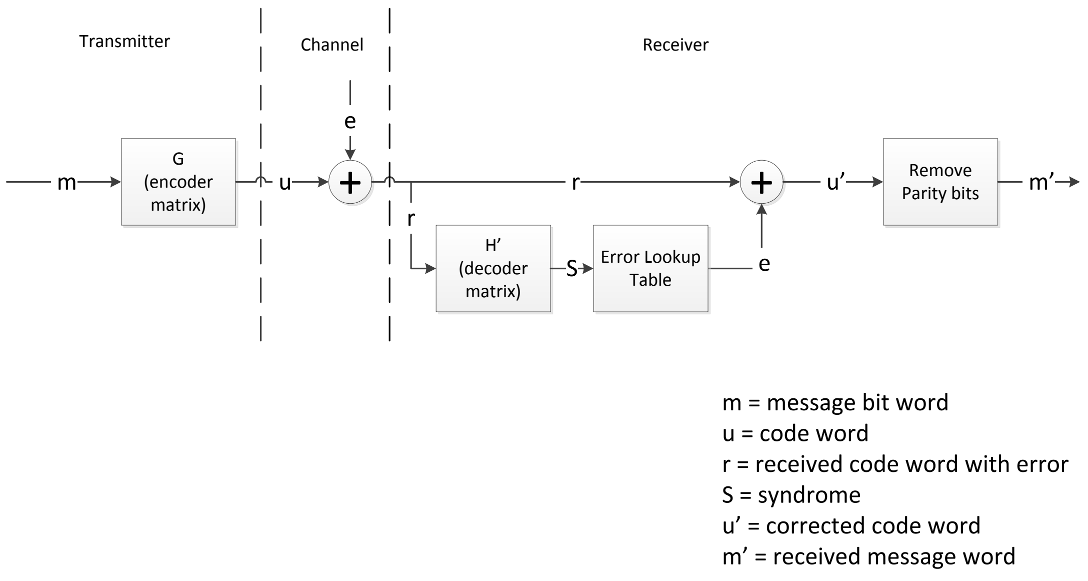


Figure 3: Hamming Code Block diagram

To describe linear block codes, (n,k) notation is commonly used. The k value denotes the number of bits in a message and the n value denotes the number of bits in the transmitted codeword. All linear block codes, including LDPC, utilize an encoder matrix, **G**, that adds redundant information to a message to be transmitted. The redundant information is used to detect and correct errors at the receiver. This redundant information is embedded in the extra bits called parity bits that are appended to a message.

In order for the system to work properly, the encoder and decoder matrices, **G** and **H** respectively, share a special relationship. The **G** and **H** matrices are orthogonal to each other, meaning that $\mathbf{G}*\mathbf{H}^{\mathbf{T}} = 0$. Multiplying a message by **G** maps it to a larger binary field which increases the Hamming distance between messages [1]. In Figure 3, the encoding process for a (7,4) code is illustrated.

$$\begin{bmatrix}1 & 0 & 1 & 1\end{bmatrix}\left[\begin{array}{cccc|ccc}1 & 0 & 0 & 0 & 1 & 1 & 0\\ 0 & 1 & 0 & 0 & 0 & 1 & 1\\ 0 & 0 & 1 & 0 & 1 & 1 & 1\\ 0 & 0 & 0 & 1 & 1 & 0 & 1\end{array}\right] = \begin{bmatrix}1 & 0 & 1 & 1 & p_1 & p_2 & p_3\end{bmatrix}$$

Figure 4a: Encoding for Linear Block Codes

$$\begin{aligned}p_1 &= m_1 + m_3 + m_4 = 1+1+1 = 1\\ p_2 &= m_1 + m_2 + m_3 = 1+0+1 = 0\\ p_3 &= m_2 + m_3 + m_4 = 0+1+1 = 0\\ &= \begin{bmatrix}1 & 0 & 1 & 1 & 1 & 0 & 0\end{bmatrix}\end{aligned}$$

Figure 4b: Modulo-2 addition used for parity bit calculation where the message [1 0 1 1] and the generator matrix is a (7,4) matrix as shown above.

After the message is encoded, it is transmitted through a channel where it might be corrupted by noise. For this project a binary symmetric channel is assumed with additive white Gaussian noise (AWGN).

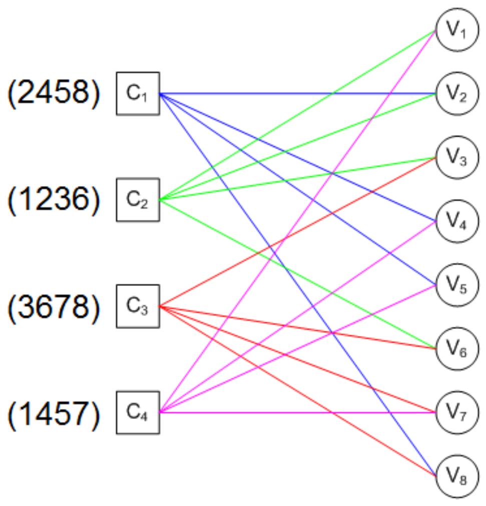
At the decoder, a corrupted codeword (**r**) is received and decoded using the $\mathbf{H}^{\mathbf{T}}$ matrix. This operation results in a value called the syndrome (**S**). If the syndrome is equal to zero, there are no errors detected. If the syndrome is not zero, the value is used to refer to a look-up table listing error patterns that result in the given syndrome. When the bit error pattern is identified, it is used to correct the corrupted codeword by flipping the incorrect bits. Finally, the parity bits generated by the encoder matrix **G** are removed and the result is the corrected message [1].

1.4 LDPC Codes

LDPC codes are different than other linear block codes in a few key ways. First of all, the **G** and **H** matrices of LDPC codes are constructed differently. For example, in Hamming code, the **G** matrix is constructed using a primitive polynomial of x^n+1 over Galois Field of GF(2). The **H** matrix is then constructed by performing certain matrix manipulations on **G** [1]. The resulting **H** matrix is orthogonal to **G**.

In contrast, for LDPC codes, the **H** matrix is constructed first as a sparse matrix. In other words, **H** has very few 1's to 0's. Hence the name "Low Density." The bit error rate (BER) performance of LDPC codes are heavily dependent upon on the makeup of **H**. For this reason, multiple algorithms used to populate **H** have been developed to attempt to maximize BER performance as well as to decrease complexity [3]. It will be shown later how the sparser the matrix the lower the complexity of the decoder. It has also been shown that randomly populating **H** can result in a relatively good BER performance [4]. After **H** is obtained, matrix manipulations are used to obtain an orthogonal **G** [3]. This project is not concerned with methods to generate parity check matrices, therefore a single matrix is used throughout.

Another difference between block codes and LDPC codes is the decoding process. In the Hamming code example described previously, the decoding process that uses hard clipped received values, or hard decision decoding. With LDPC codes, a soft decision decoding algorithm that retains the received magnitude of each bit is used [5]. This algorithm is more easily described using a graphical representation of the **H** matrix called a tanner graph [4], illustrated in Figure 5.



(8,4) Example

$$H = \begin{array}{cccccccc|c} V_1 & V_2 & V_3 & V_4 & V_5 & V_6 & V_7 & V_8 & \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & C_1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & C_2 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & C_3 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & C_4 \end{array}$$

*Figure 5: Tanner Graph with corresponding **H** matrix* [4]

To construct a tanner graph, first, create a check node (squares denoted by C's) for every row in **H**. Next, create a variable node (circles denoted by V's) for every column in **H**. Where there is a 1 in **H**, draw a line between a check node and variable node that corresponds to its location [4]. For example, there is a 1 in the first row and second column of **H**, so a line is drawn connecting C_1 to V_2.

1.5 Log-Likelihood-Ratio Algorithm

Multiple decoding algorithms exist that can be used with LDPC codes. The decoding algorithm used in this project is called the belief propagation with Log-Likelihood-Ratio (LLR) algorithm [4]. From here on it will be referred to as simply the LLR algorithm. To explain the algorithm, a tanner graph for a (10,5) code shown in Figure 6.

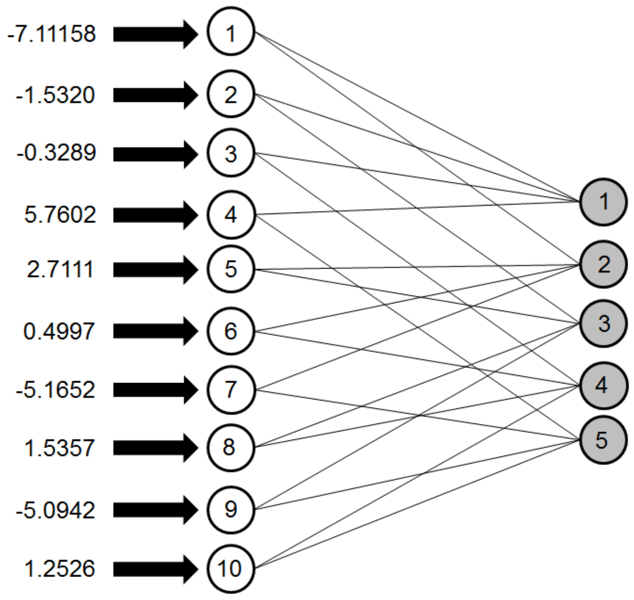


Figure 6: Tanner graph for a (10,5) code with received values

The algorithm operates as follows.

1. The received codeword value for each bit is fed into a V node.
2. Each V node sends its value to the connected C nodes.
3. C nodes calculate an edge value to send back to each V node that it believes is the correct value for that V node.
4. Each V node sums the values received from C nodes with the original received value and updates its current value.
5. Repeat steps 2 - 4 for a predefined number of iterations.
6. V nodes calculate their final values and are hard clipped, the values are the corrected results.

The calculation that each check node performs to determine the new values to send back is illustrated in Figure 7. In this example the edge value that C_1 sends back to V_1 is calculated. In this process, C_1 ignores the value from V_1, and it takes the values of the other 3 connected V nodes. Each of those values is passed through the algorithm in Figure 7 one at a time. The algorithm strips the sign bit off the inputs and updates the product of them. Simultaneously, it takes the absolute value of the inputs and performs the Phi function (see

Equation 1), and then takes the sum. Finally, the product and the sum are multiplied and the output is fed back to V_1. Check nodes repeat this process for each connected V node.

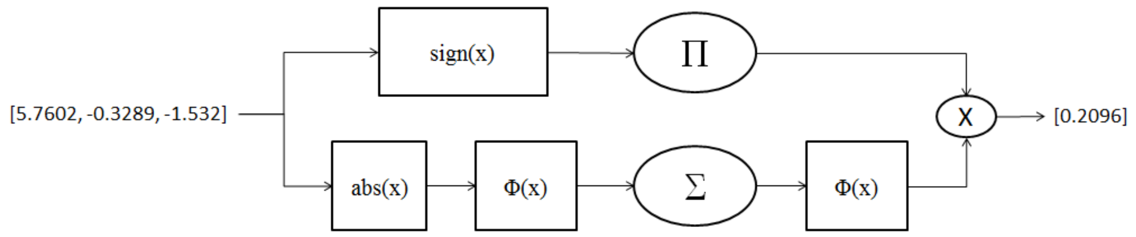


Figure 7: Calculation for Edge between V_1 and C_1

$$\Phi(x) = \ln\left(\frac{e^x + 1}{e^x - 1}\right) \tag{1}$$

Each time the process is repeated, the probability that the received data is corrected increases. By increasing the number of repetitions a lower BER can be achieved. However, increasing the number of iterations also increases the amount of time it takes to decode [5]. In practical systems, the number of iterations must be limited in order to read the message in the required time. Figure 8 shows the effect of increasing the iteration number on the BER rate.

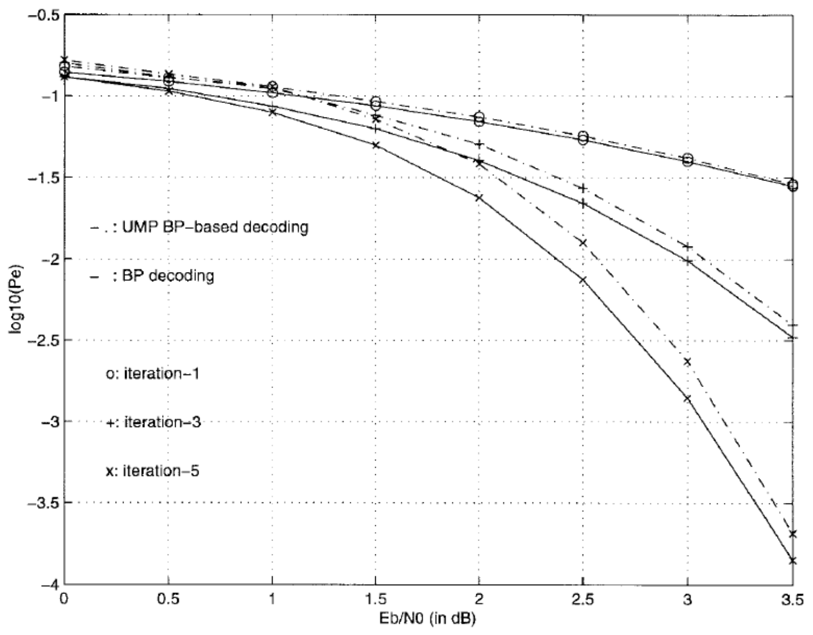


Figure 8: BER Comparison Between different Iteration Numbers [5]

1.6 Functional Requirements

At the beginning of the project, functional requirements are set based on what could realistically achieved and what is required for a practical LDPC system. They are listed as follows.

- Xillinx Spartan 3E board shall be used for implementation with the on board 50MHz clock
- The decoder shall perform at least 5 decoding iterations, as seen in Figure 8
- The system shall use the Log-Likelihood Belief Propagation algorithm
- The system shall perform at a code rate of ½
- The system codeword size for FPGA implementation shall be 1008 bits long
- The system message word size for FPGA implementation shall be 504 bits long
- The system shall use less than 500,000 logic gates on the FPGA

2. Project Approach

After gaining a fundamental understanding of LDPC codes, the next step taken was to perform and analyze a MATLAB implementation of an LDPC system. It included the simulation of an encoder, AWGN channel effects, and two different decoding algorithms.

2.1 Simulink Design

After becoming familiar with the MATLAB implementation, Simulink is used to design a LDPC system. This model was to be the basis of the LDPC project. In order to get a system working quickly, LDPC encoder and decoder blocks from Simulink Communications Toolbox were used. These blocks allow for setting the **G** and **H** matrices as well as the number of decoding iterations. Figure 9 shows the results from the design using Communication Toolbox blocks.

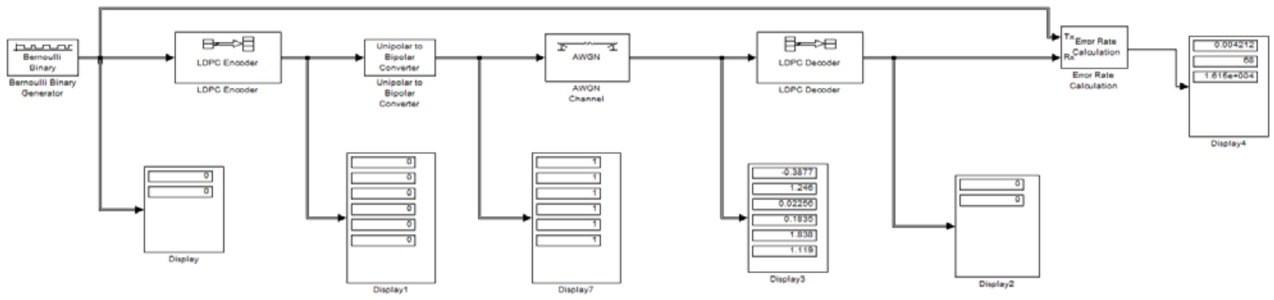


Figure 9: Simulink LDPC System Model

After completion and simulation of this system, the next objective was to break down the encoder and decoder into more fundamental blocks.

2.2 Pre-Implementation Challenges

One of the first challenges encountered during this project was the Simulink decoder design. Because of the iterative process of LDPC decoding, developing a decoder in Simulink proved to be extremely difficult. The value that this step of the project would provide to the end result was not worth the time it would require to complete it. Therefore, the focus was shifted to directly constructing a decoder using Xilinx System Generator blocks.

The decoding algorithm used in MATLAB implementation codes was chosen to be implemented in hardware using Xilinx System Generator blocks. However, because decoding algorithms use a non-linear logarithmic function, which is difficult to implement in hardware.

The MATLAB algorithm using Square-Add function was initially used, the other choice being the LLR algorithm previously discussed. The Square-Add function (see Equation 2) utilized a non-linear function called the MAX* function (see Equation 3), which an approximation was able to be determined for. The approximation would allow the function to be straightforwardly implemented on hardware.

$$\text{Square - Add}(L_1, L_2) = MAX * (L_1, L_2) - MAX * (0, L_1 + L_2) \tag{2}$$

$$MAX * (L_1, L_2) = MAX(L_1, L_2) + \ln\left(1 + e^{|L_1 - L_2|}\right) \tag{3}$$

Multiple simulations were performed using the Square-Add decoder and generated a BER plot over a range of E_b/N_0 from 0 to 10 dB. The system appeared to be working as it achieved a better BER than an un-coded system, however it still performed poorly. At this stage the poor performance was attributed to the relatively small (10,5) code used in the performance evaluation.

In a later stage of the project, during the construction of the decoder in Xilinx, it was discovered why this algorithm performed so poorly. The reason for the poor performance was due to using the Square-Add function in place of Phi(x) function; that was in the Log-Likelihood algorithm. A realization was made that the algorithms associated with each function do not operate in the same way.

Since a large amount of time was spent on the LLR algorithm, the continuation of the same algorithm was used, but this time with the correct function needed, Phi(x) (eq. 1). Thus the next challenge that was presented, was to develop an approximation or a lookup table for the non-linear phi function in order to implement it on hardware.

2.3 Simulink Encoder Implementation

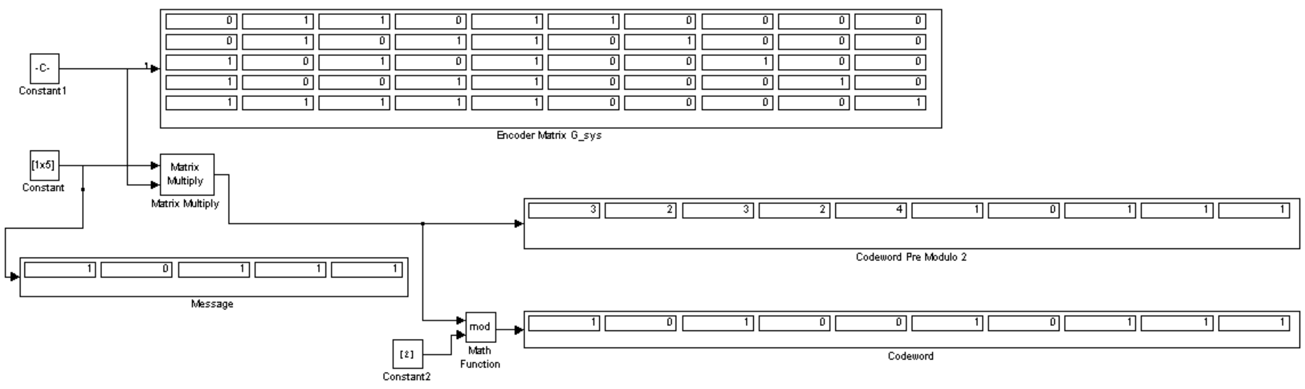


Fig. 10: Low Level Simulink Encoder

After the construction of the high level block diagram of the LDPC system [Fig. 9] using the built in Simulink blocks for the encoder and the decoder, low level encoder construction began. As can be seen in Figure 10, blocks that were lower level, thus closer to what is needed to do implementation in hardware, were used in the next stage of the implementation process. There is a hard-coded constant that holds the **G** matrix, and it is displayed by the 5x10 matrix seen in the display. The message used is hard coded as well, being multiplied by the **G** matrix. The result is the codeword before any modulo-2 math is performed. The pre-modulo-2 codeword has values such as 0, 1, 2, 3, etc. These values need to be changed by using the modulo-2 math operator, and in this case, a final codeword of [1 0 1 0 0 1 0 1 1 1] that is to be transmitted is acquired.

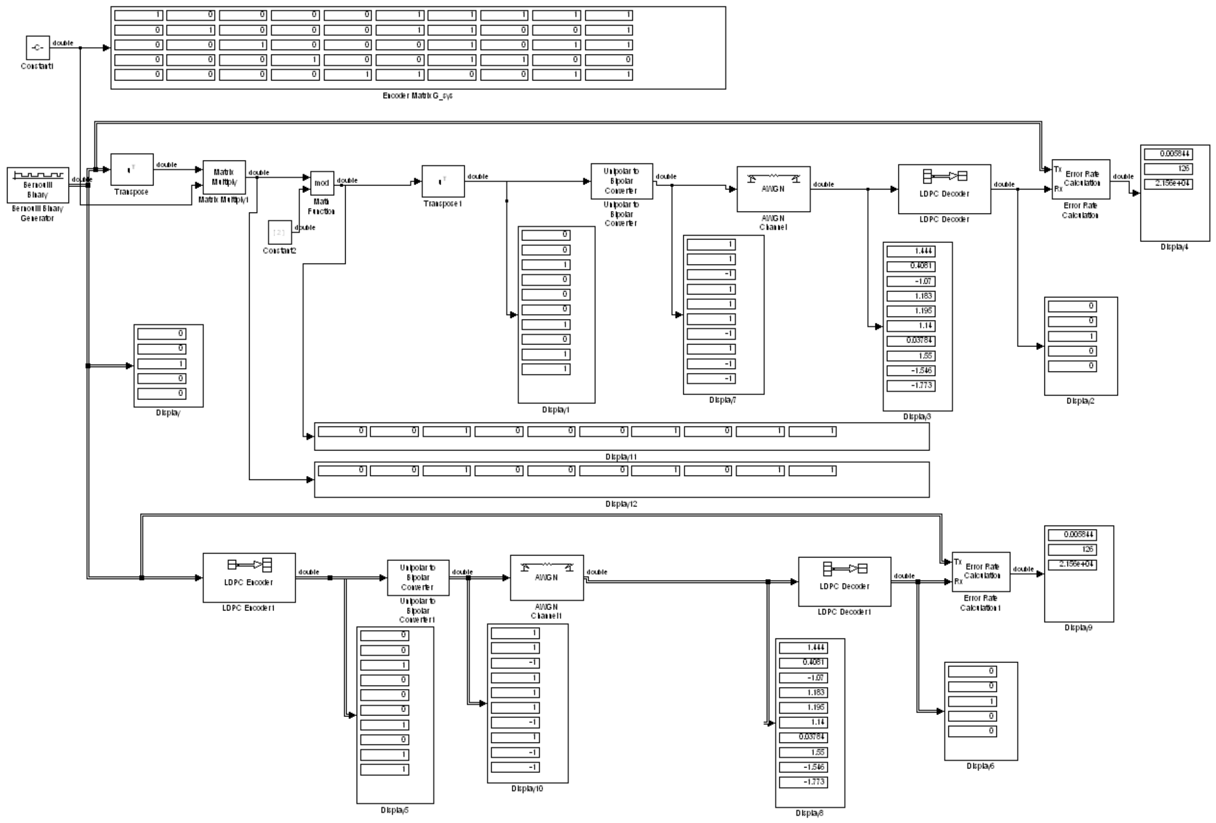


Fig. 11: Low Level Encoder Verification

Figure 11 includes both the high level LDPC system and the low level encoder. They run side by side for the easy verification of LDPC system. It is verified that the probability of bit error and the number of errors coming out of both systems are the same. It concludes that the low level encoder is working as expected, and the hardware implementation stage using the Xilinx blockset would be the next.

2.4 Xilinx Encoder Implementation

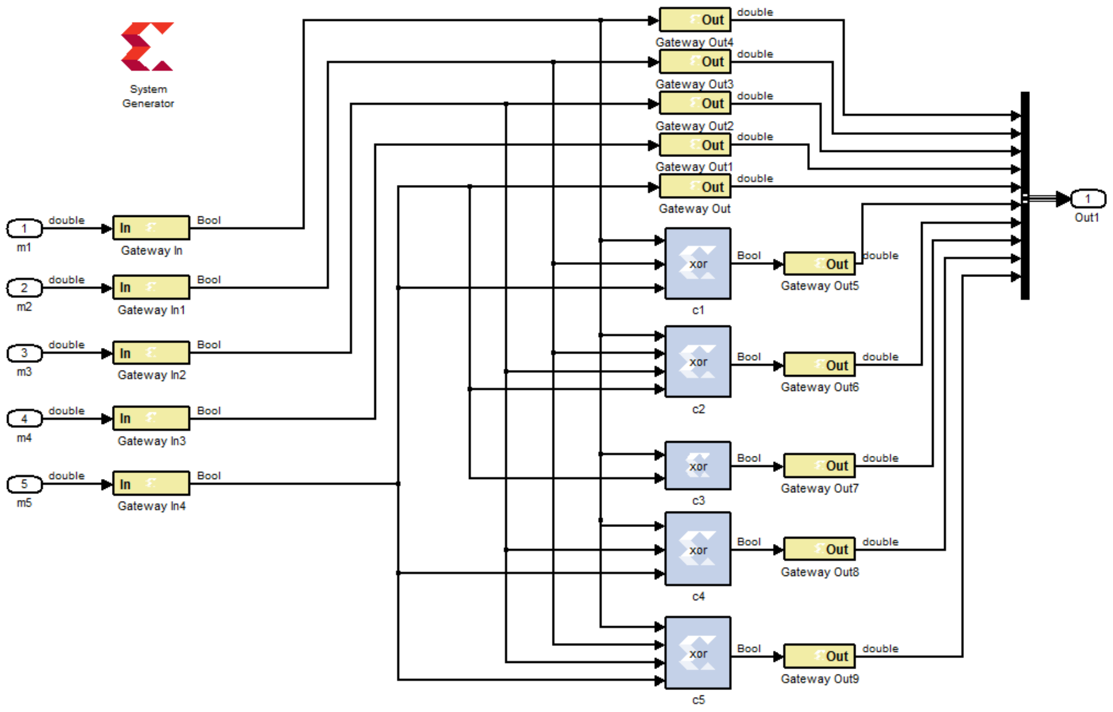


Fig. 12: Xilinx Encoder

Figure 12 shows the encoder design using Xilinx System Generator, where 5 bits sent into the encoder are left unchanged. They are passed directly to the output, this is due to the systematic **G** matrix, and the five message bits are retained in the codeword. The parity bits are constructed as the last 5 bits in the codeword. Exclusive OR blocks (XOR) are used for parity bit generation. The first parity bit is determined from 1st, 2nd, and 5th message bit, thus by XOR these signals, the corresponding parity bit value is obtained. This same process is done for each parity bit.

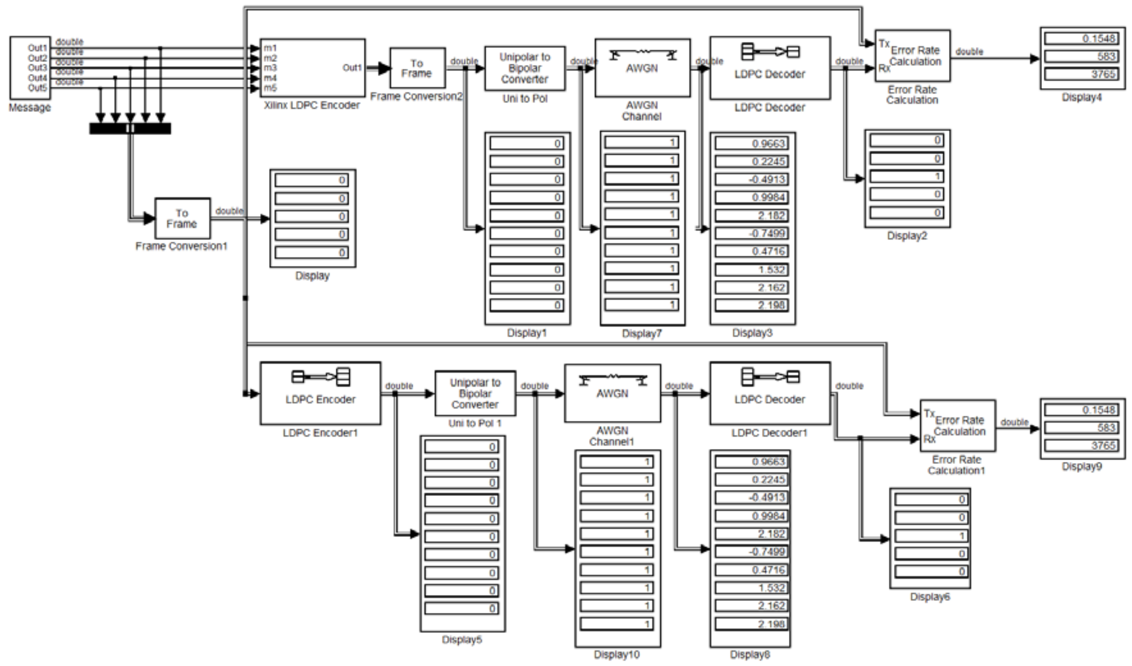


Fig. 13: Xilinx Encoder Verification

Verification on the encoder made with Xilinx blocks is performed against the LDPC system provided in the communication toolbox. This is the same system implemented in Figure 9. It can be noted, that the Xilinx encoder that was built performs exactly identical to the LDPC encoder from the communication toolbox.

2.5 Simulink Decoder Implementation

Moving on from the encoder's evolution through each of its stages, the decoder side of the project will be broken down and explained. The decision was made to move directly to using the Xilinx blockset rather than spending a great deal of time developing a purely low level Simulink model. It was thought that a very low level Simulink implementation would be just as difficult if not more, than a Xilinx block set model.

The MATLAB code seen in the Appendix was used extensively in building and verifying the decoder. A low level approach was taken; then working up to a higher level, throughout the stages of this implementation. Getting the correct MATLAB code to use was key in having a working LDPC system and being able to verify its operation. There were problems that arose with the MATLAB code that used the Square-Add function (equation 2) as stated previously. Edge values would not be updated and decoding wasn't being performed properly. By using a MATLAB code that used the Phi(x) function (equation 1), but with the same algorithm, the system worked as expected. The alternative algorithm that utilizes the Square-Add function could have been studied in more detail, but as time was an issue, the selection of the algorithm had already been studied was used and the Phi(x) function was to be implemented rather than Square-Add.

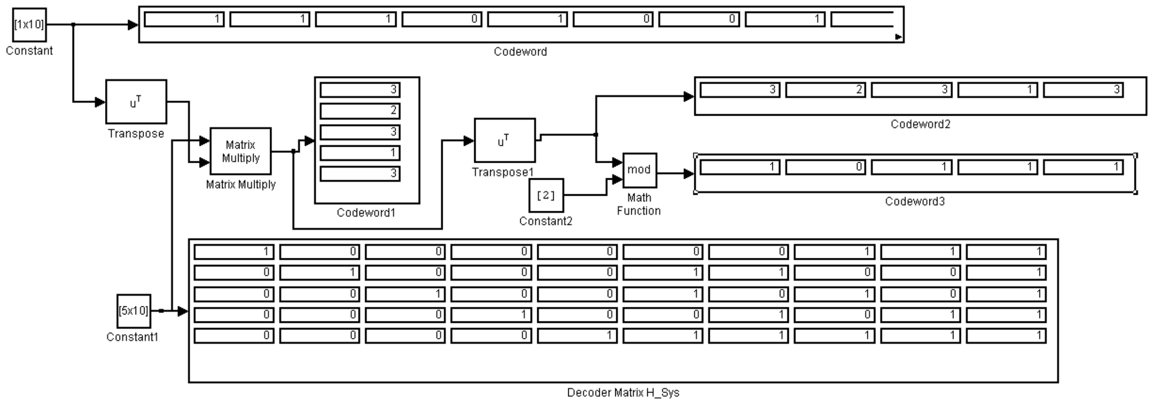


Fig. 14: Low Level Simulink Decoder

After explaining the encoder in low level Simulink blocks, a general decoder system in Simulink was implemented as well. This system takes in the received codeword, then multiplies it by the H, decoder matrix. Before this is done, a transpose block is needed, so the inner dimensions of both the matrices agree in order for matrix multiplication to be performed since Simulink uses column vectors. After this is completed, the pre-modulo 2 message will be acquired. Then a modulo-2 operation is performed, resulting in the original message being generated. This is however, a basic decoder; with Hamming code, $\mathbf{H}^{\mathbf{T}}$ would be used to find the syndrome, rather than m, the original message (see Figure 3). This Figure also varies from the LDPC decoder in another way as well. It differs from a true LDPC decoder since it doesn't perform any iterations, or mathematical calculations at the "nodes". After this Simulink model was completed, the rest of the decoder implementation was done with the Xilinx blockset seen in the following sections.

The "Phi" algorithm was one of the first subsystems to be implemented when starting the Xilinx implementation. The Phi function is a non-linear function, which cannot be easily performed in hardware. The decision on using a lookup table to overcome this challenge was made. Simulations were run with the MATLAB code to verify if a lookup table was a viable option in this system. As can be seen in the next section, the lookup tables performed very close to the original Phi function.

2.6 Constructing a Lookup Table

To construct a lookup table for Phi(x), plotting the function was performed over a range of positive inputs. Positive inputs of Phi(x) are the only values to be taken into account because the Log-Likelihood algorithm strips the sign bit before passing a value to Phi(x). Next a determination was made for the cut-off points for the lookup table. These were the maximum and minimum points the lookup table would cover. Chosen values of x = 0 for the minimum and x = 8 for the maximum. A number of evenly spaced points between 0 and 8 which would be the values included in the lookup table were generated. Next, values of Phi(x) were calculated for each input. Because Phi(x) is asymptotic towards 0, a value to clip Phi(0) to was made to be 3. Phi(x) look-up table comparisons are shown in Figure 15.

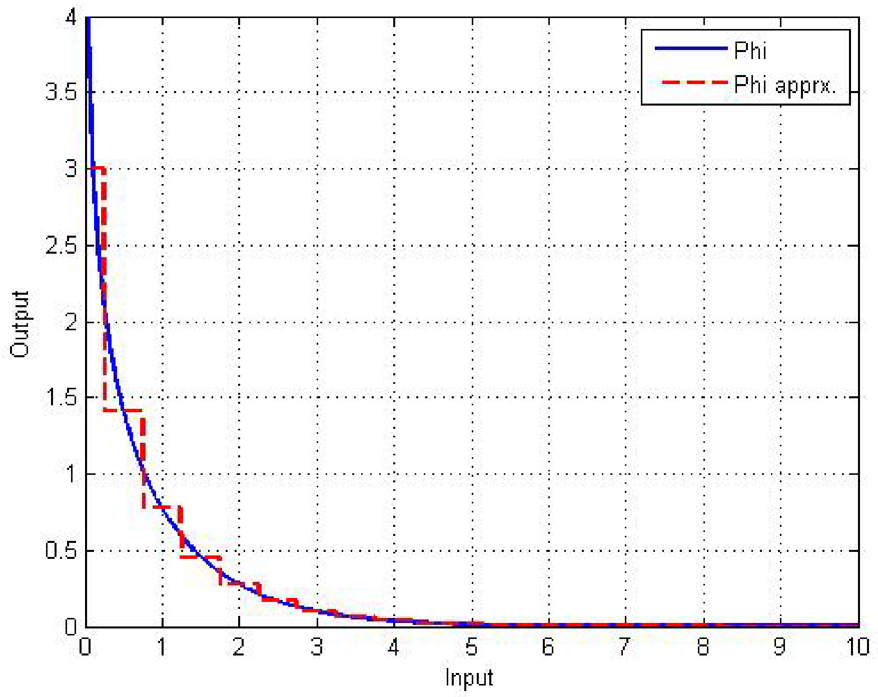


Figure 15: Phi(x) Lookup Table Comparison

2.7 Simulation Results

The LDPC system was simulated next using different sized lookup tables in order to see if there was a large difference in performance between the different tables.

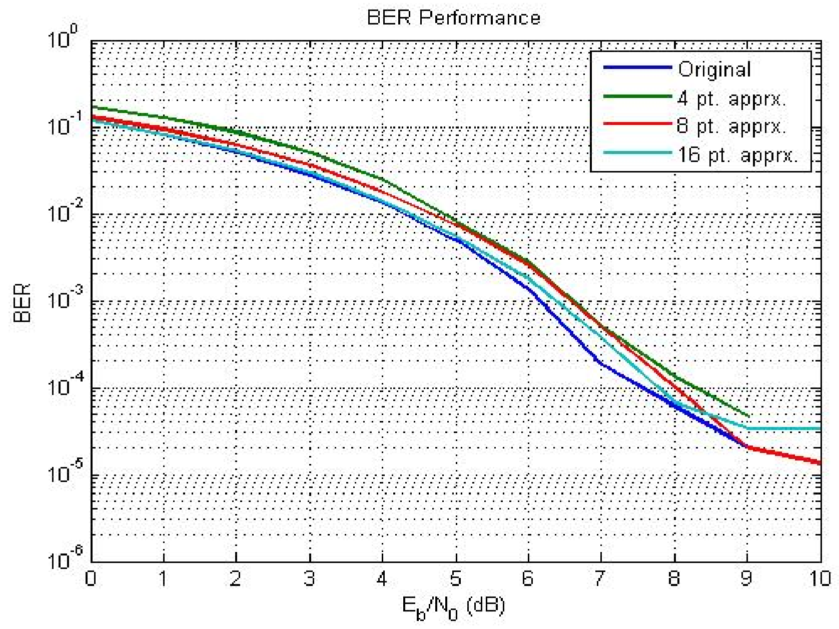


Figure 16: Lookup Table Performance Comparison

Figure 16 shows the results of the simulations using different sized lookup tables. Note that at E_b/N_0 values above 7dB result in an unreliable BER calculation. The simulations show that at some high E_b/N_0 values a smaller lookup table performs better than a larger one. This is likely because we did not simulate enough messages to obtain statistically significant results. Each data set shown in Figure 16 can take up to 20 minutes to simulate. Because the code's performance at lower E_b/N_0 values is more important, re-simulation was not necessary.

As a result of the simulations, a 16 point lookup table was chosen for FPGA implementation. The 16-point table provided performance similarly to the original phi function at lower values of E_b/N_0, with less hardware cost.

Below is the Phi function system used in the Xilinx implementation (see Figure 17). The edge value is passed into the first block; the output is the absolute value of the input. The absolute value of the edge is fed into the second block, the "Lookup_Index_Number" block. This will calculate the index value of the Phi that needs to be selected for the given input. The "Phi_Selector" block is where the previously computed index value selects the memory location to be output, the corresponding memory location will be the corresponding Phi value based on the original input.

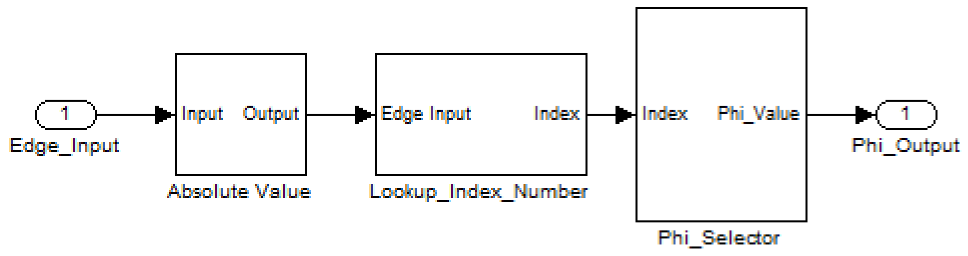
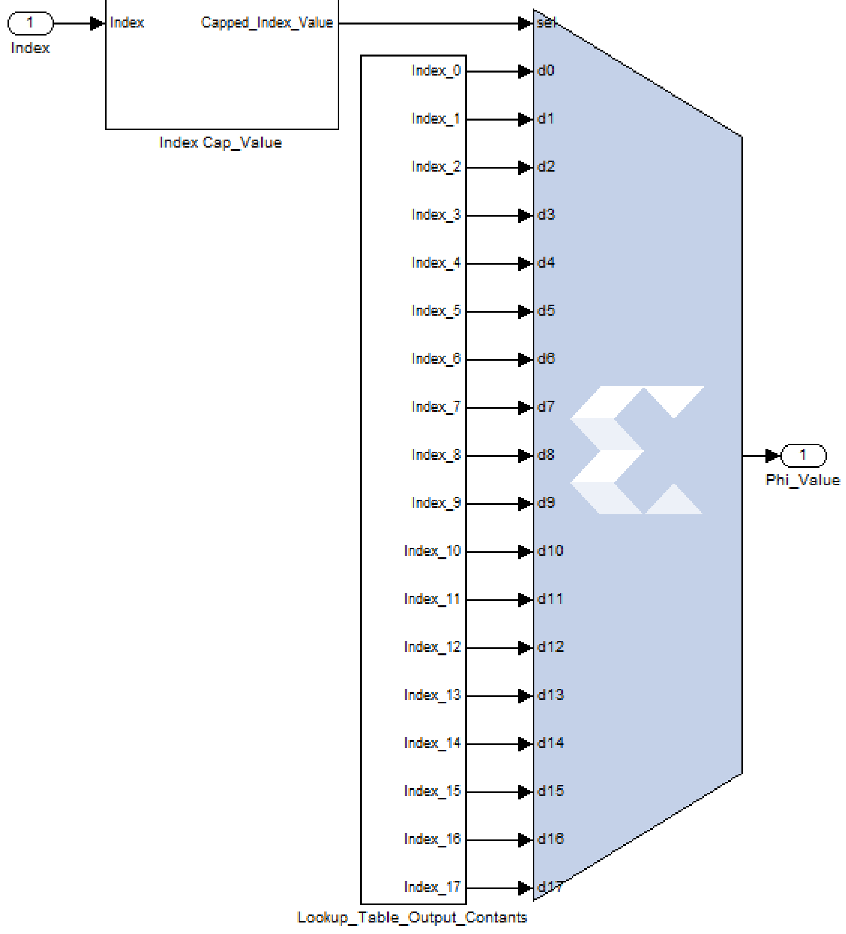


Figure 17: Complete Lookup Table Function Block

Taking a look at the “Phi_Selector” block first from Figure 17, it is seen in Figure 18 that a lookup table is housed inside that block. A selection of a lookup table size was made and then allocated memory space to hold the lookup table values. A MUX was used that is controlled by the index control line; that determines what output value for Phi to use. This value is fed into the MUX, which selects the corresponding Phi value to be output. The MUX and the lookup table values can be seen in Figures 18 and 19, respectively. The block called “Index_Cap_Value” can be seen in Figure 18 as well. This block makes sure that an index value computed before is a value that can be fed into the MUX without causing errors with the operation. The internals of the “Index_Cap_Value” block can be seen in Figure 20.



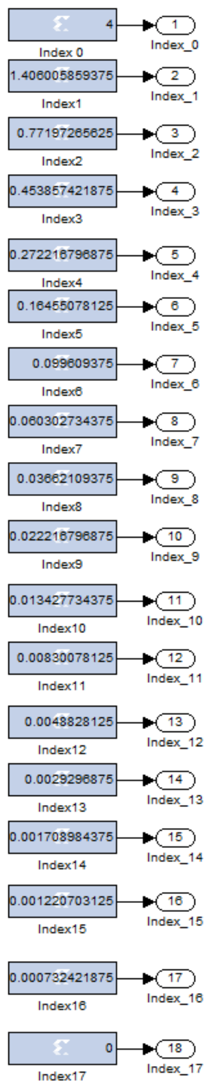


Figure 18: Lookup Table Output

Figure 19: Lookup Table Memory

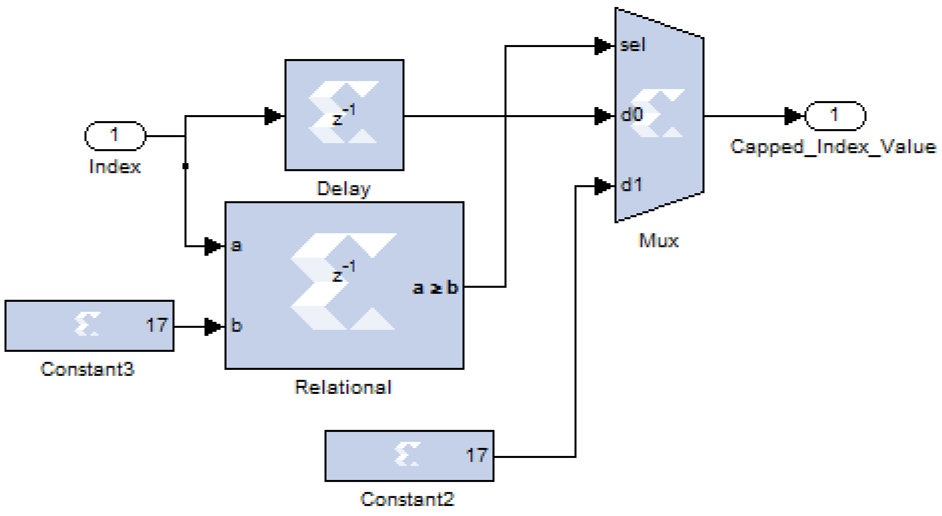


Figure 20: Index Cap Block

The middle block seen in Figure 17, is called" Lookup_Index_Number", and that is exactly what it does. Below, Figure 21 displays the internals of that block.

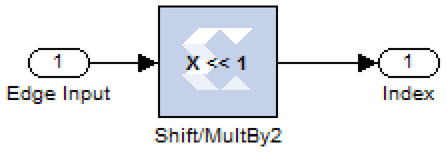


Figure 21: Lookup_Index_Number

The setup of the values we will except into the lookup table and the size of the lookup table made the calculation of the index value fairly straight forward. The only operation done is a shift left. This shift left is equivalent to multiplication by the value of 2. For example, let us say the input is a value of a 2, or 0010. If a shift left operation is done, we get, 0100, which is a binary 4, this value is sent to the multiplexer seen in Figure 18. The value selected from the multiplexer is the 4th memory spot, and the value saved there corresponds to an Edge input of 2 going into the Phi function.

The 3rd value, in Figure 17 that will be explained is actually the first to be done. This block is the absolute value block and it can be seen in Figure 22 below.

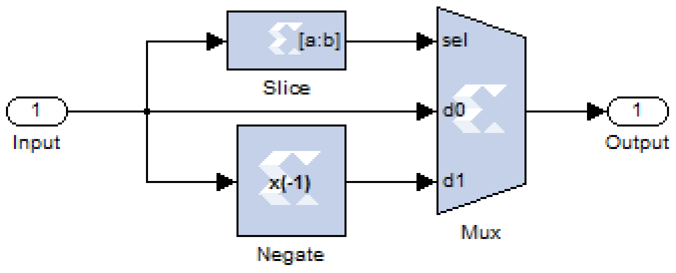


Figure 22: Absolute Value Function

The absolute value function brings in an input, and slices off the 1st bit, which will be the sign bit of the input value. This sign bit is sent to the selector input of the MUX. If the bit is a 1, then the edge was a negative value, and the negative of the input value is taken by using the "Negate" block. This will result in the positive value. If the sign bit is a 0 however, stating the original input is already positive, then the output is the original input by selecting the d0 on the MUX.

Figure 23 shows the overall block set up of the unit that does the "Node" calculations to update the edge value to be sent back to the variable nodes.

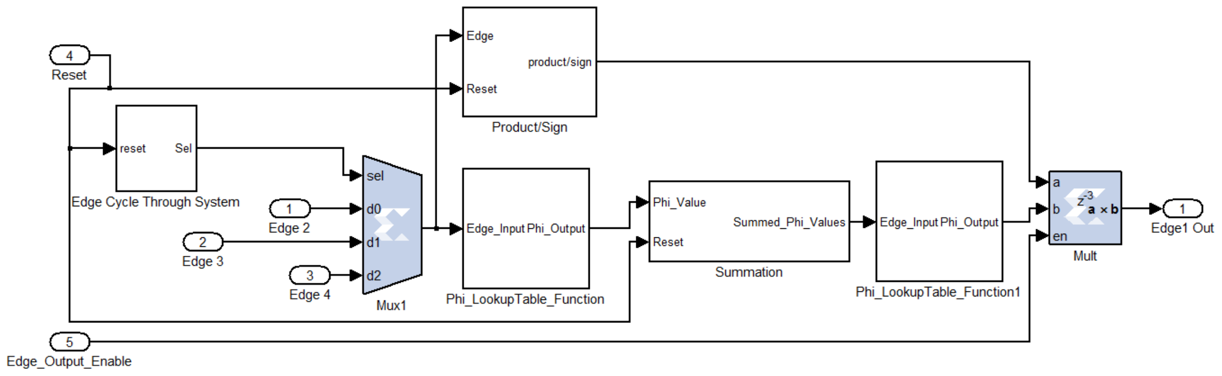


Figure 23: Edge Calculation Block

As can be seen in Figure 23 above, the previously explained "Phi_LookupTable_Function" is used twice in this block. This set up is based off the attached MATLAB code in the Appendix. Three edge values in this case are used to update "Edge1 Out". Edge 2, Edge 3 and Edge 4 are inputs into the multiplexer in the system. The "Edge_Cycle_Through_System" block starts with first edge, waits for the first edge to finish computing before sending the next edge into the system to be calculated. The edge values from the multiplexer are separated into two paths, as seen in Figure 7. One path, the upper path, finds the sign of the edge value that is input, updates the product value and saves it into a memory location. The second path, the lower path, does the Phi function block as already explained. From there, it is passed into the "Summation" block. This block saves the edge values together, then once all edges are summed, passed this value through the Phi function once again, and performs a multiplication on the sign bit. These other blocks will be explained in detail in the next coming sections.

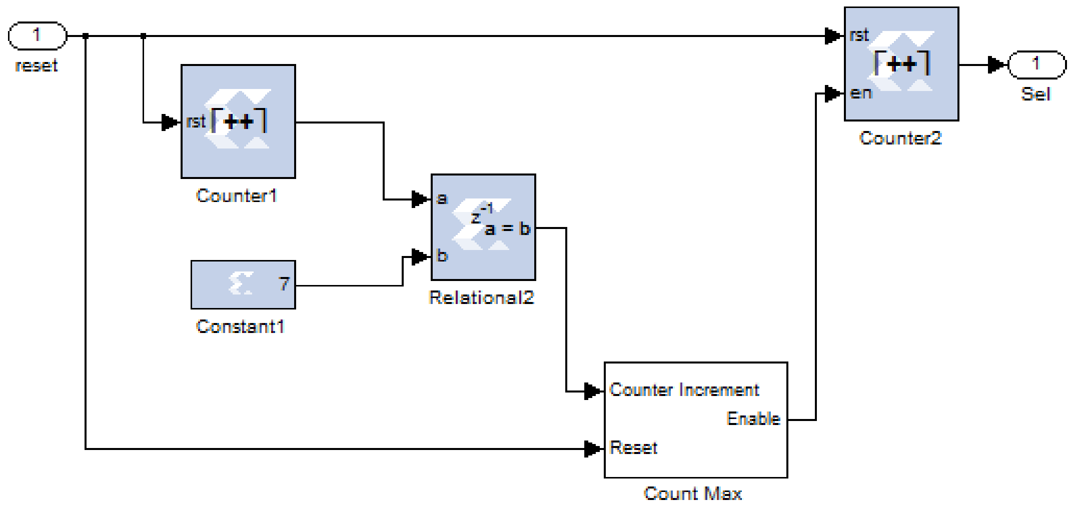


Figure 24: Edge_Cycle_Through_System

Figure 24, as seen above, is responsible for cycling through each edge value to then have it sent to the rest of the edge calculation section. A dilemma that needs to be taking into account would have to be how fast to cycle through the input edge values. It is possible to go into blocks to verify what the latency of each unit is. This was also taken into account when making a counter that starts with the first edge, counts to a given value, allowing the edge value to get through all the calculation blocks, and the block sends the next edge value into the system. A global reset is used to reset the system.

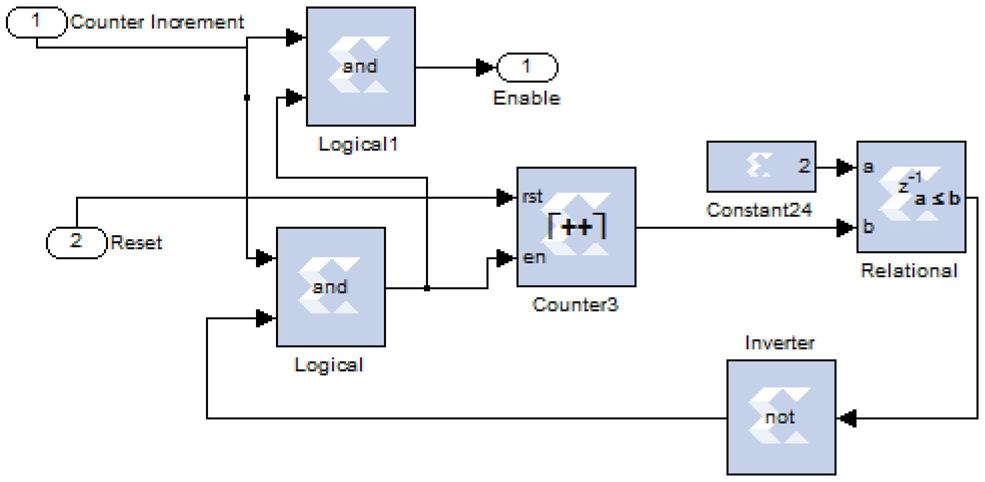


Figure 25: Max_Count Block

The above Figure, Figure 25, is the "Max_Count" block. This is used in the system seen in Figure 24. Max_Count controls how the "Enable" signal acts, that is fed to the counter that controls the edge values to be input into the system. This system takes the "counter increment" signal and passes it through some logical blocks which will determine if the count value is greater than the value of edges, if this is the case, the system stops counting and disable the counter, also by doing this it sets the enable back to 0.

The next block to examine from Figure 23, is the Product/Sign block. The Product/Sign Block can be seen below in Figure 26. The first operations is to slice off the first bit, which is the sign bit. This value is either a 1 or 0, and controls a multiplexer, that feeds in a constant of a positive 1 or a negative 1. From there, the value is multiplied with the previously saved value. If the value is different than the previously saved value, it is saved into the register, in this block called Register1. Otherwise, if the value is the same, no update is needed.

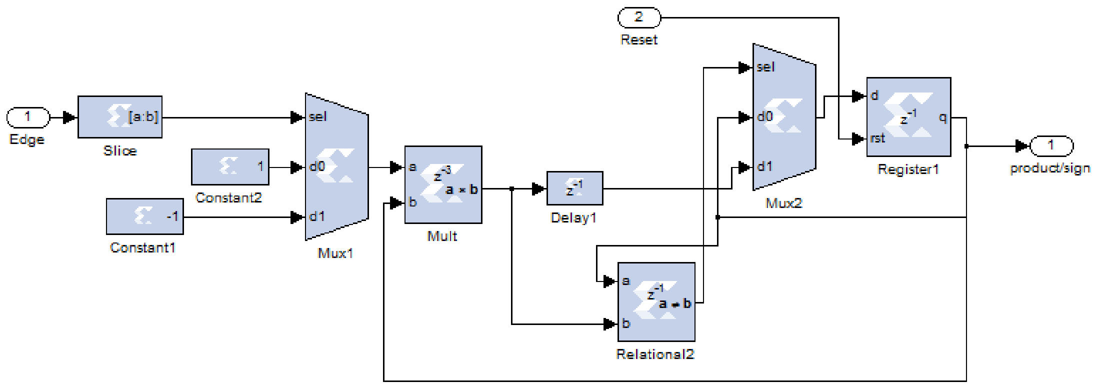


Figure 26: Product/Sign Block

The final block to converse about from Figure 23 would be the “Summation” block. This can be seen below in Figure 27.

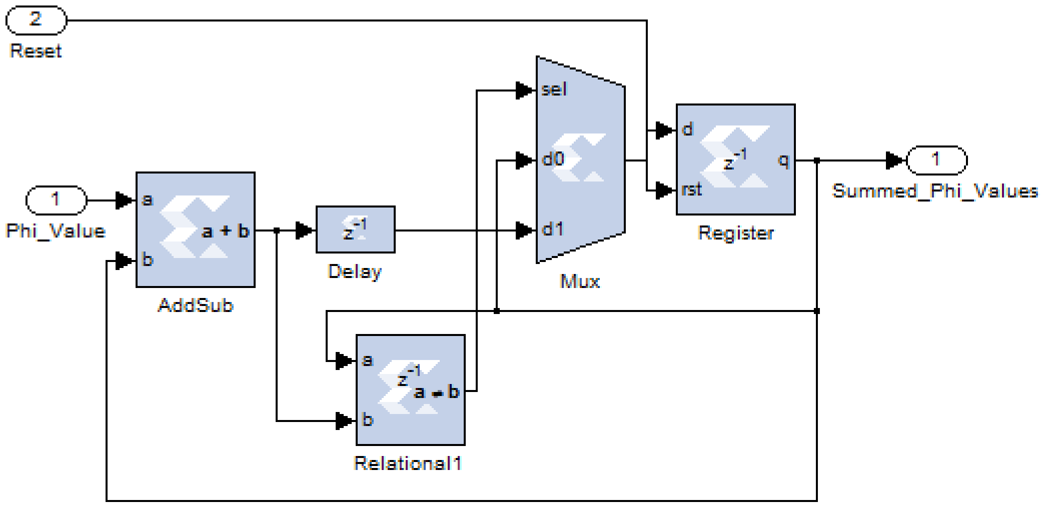


Figure 27: Summation Block

The “Summation” block takes in the most recently computed Phi value, and adds it to the value of the already summed Phi values. After the two values are added together the summed value is sent into both a delay block and a relational block. The relational block checks to see if the new value is different than the previous value. If the values differ, then the new value needs to be store, this is controlled by the multiplexer, which is sent a 1 from the relational block, if a new value is sensed. If no new value is sensed, then nothing changes, and the original value saved in the register is maintained.

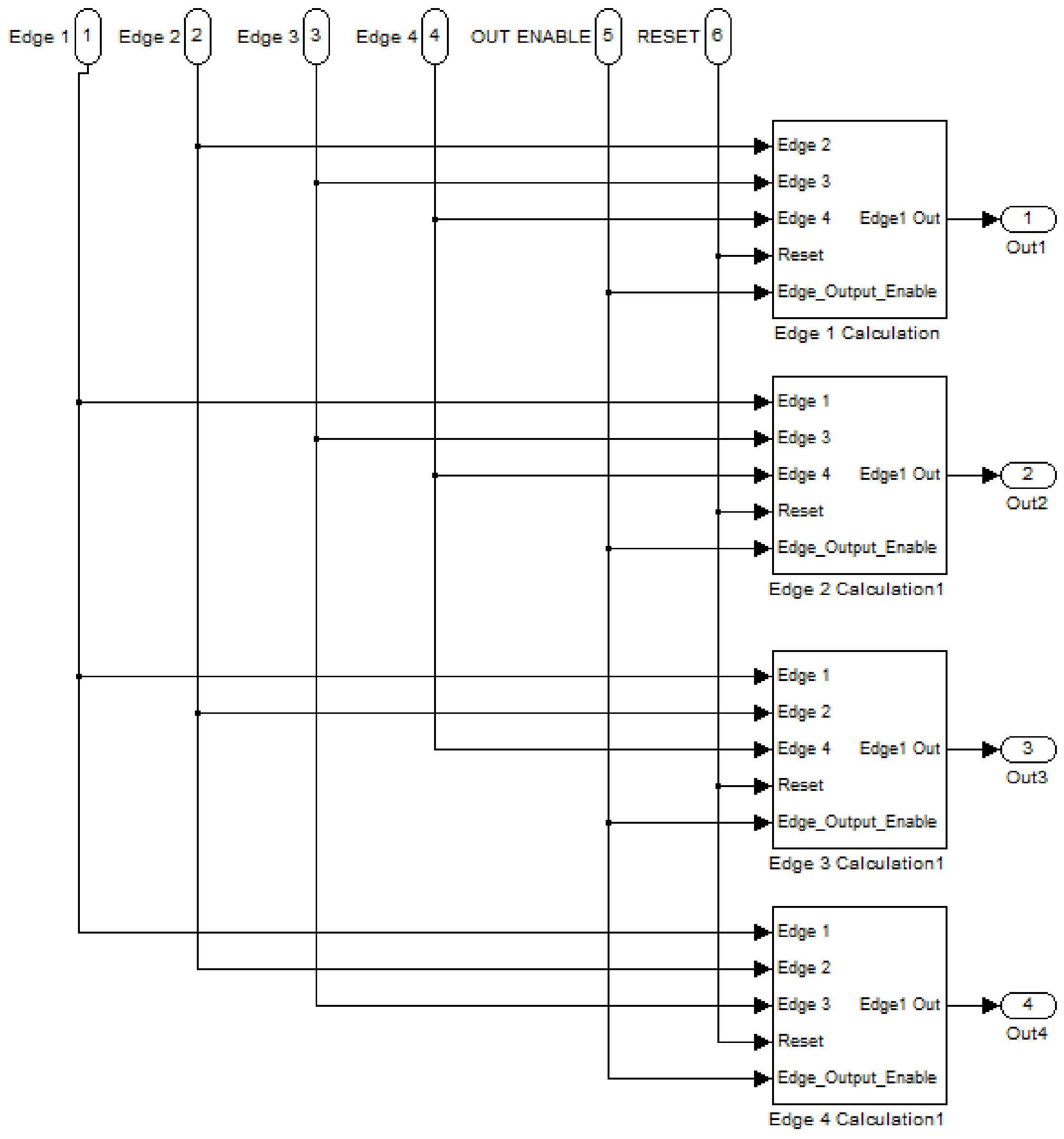


Figure 28: Edge Calculation Blocks that make up a Node

Figure 28 shows four edge calculation blocks. These edge calculation blocks are the same as seen in Figure 23. These four edge calculation blocks represent the first check node as seen from the Tanner graph in Figure 6. As Figure 6 shows, there are 4 edges connected to the check node, thus 4 edge values need to be calculated.

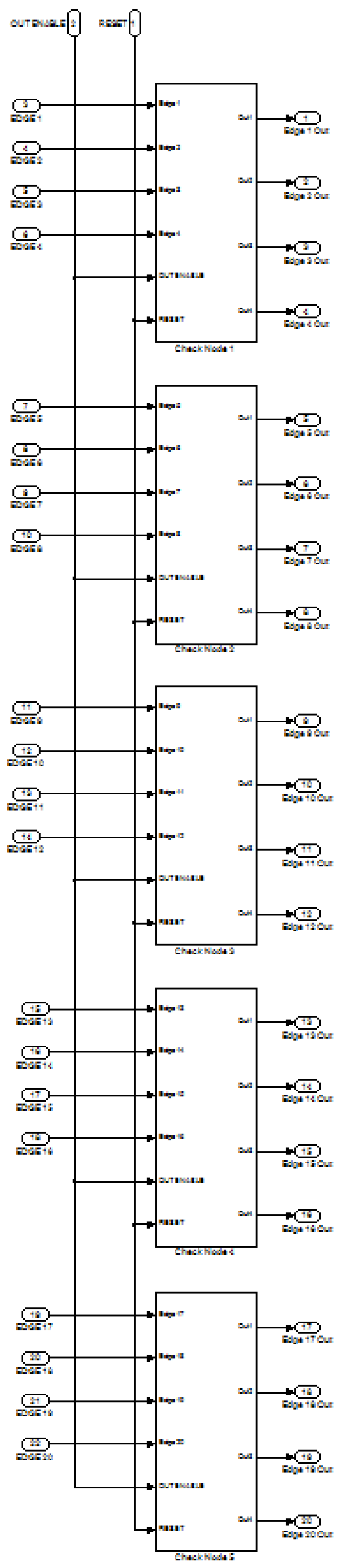


Figure 29: Total Check Node View

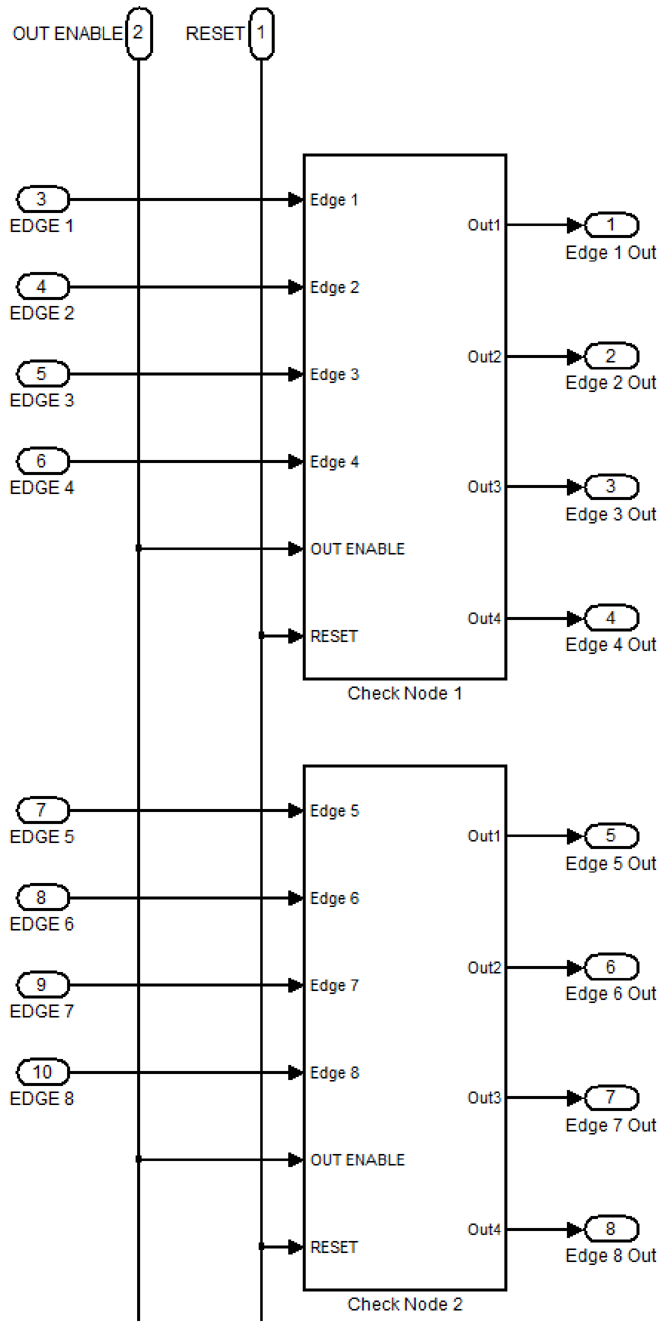


Figure 30: Zoomed-In Check Node View

Figures 29 and 30 show the check nodes that correspond to the Tanner graph in Figure 6. Figure 29 shows all the check nodes in this system. There can clearly be seen that there are 5 check nodes, the same number as seen on the Tanner graph in Figure 6.

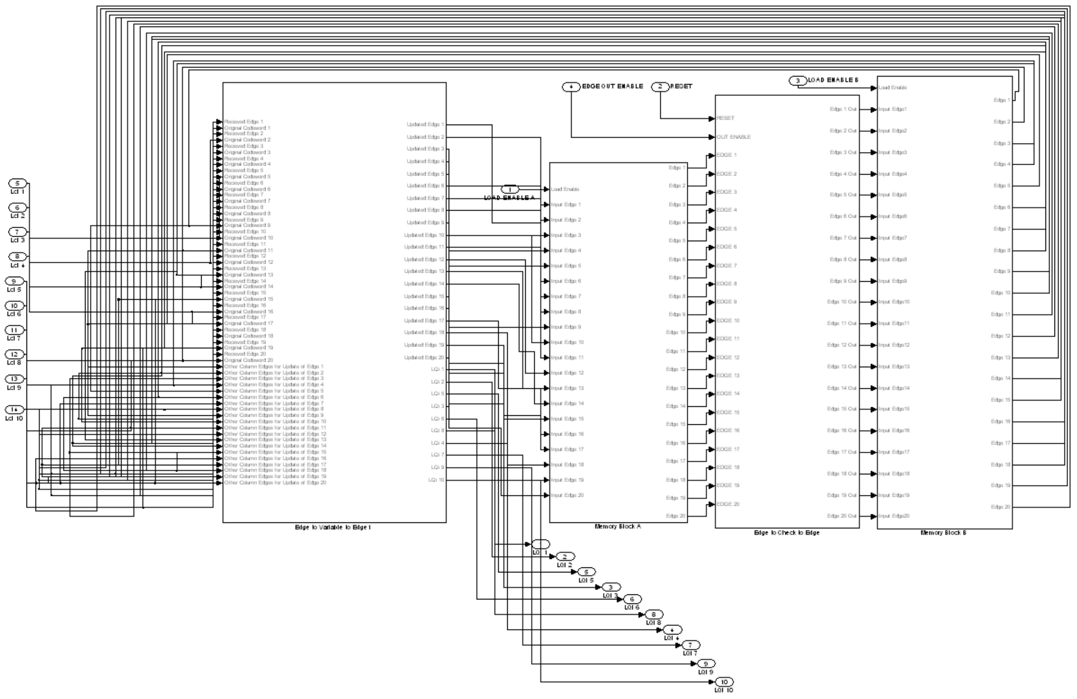


Figure 31: Full Decoder in Xilinx

Figure 31 shows the overall block diagram of the decoder. There are 4 main blocks that make up the decoder system. The far left block, which will be looked into detail next, is the block that performs the calculations that happen at the variable nodes. The next block, the middle left block is a memory block to stored edge values sent from the variable nodes to the check nodes. The middle right block is the check node block, which we looked into great detail in the previous sections. The final far right block is another memory block. This is used to store the edge values being sent from the check nodes to the variable nodes.

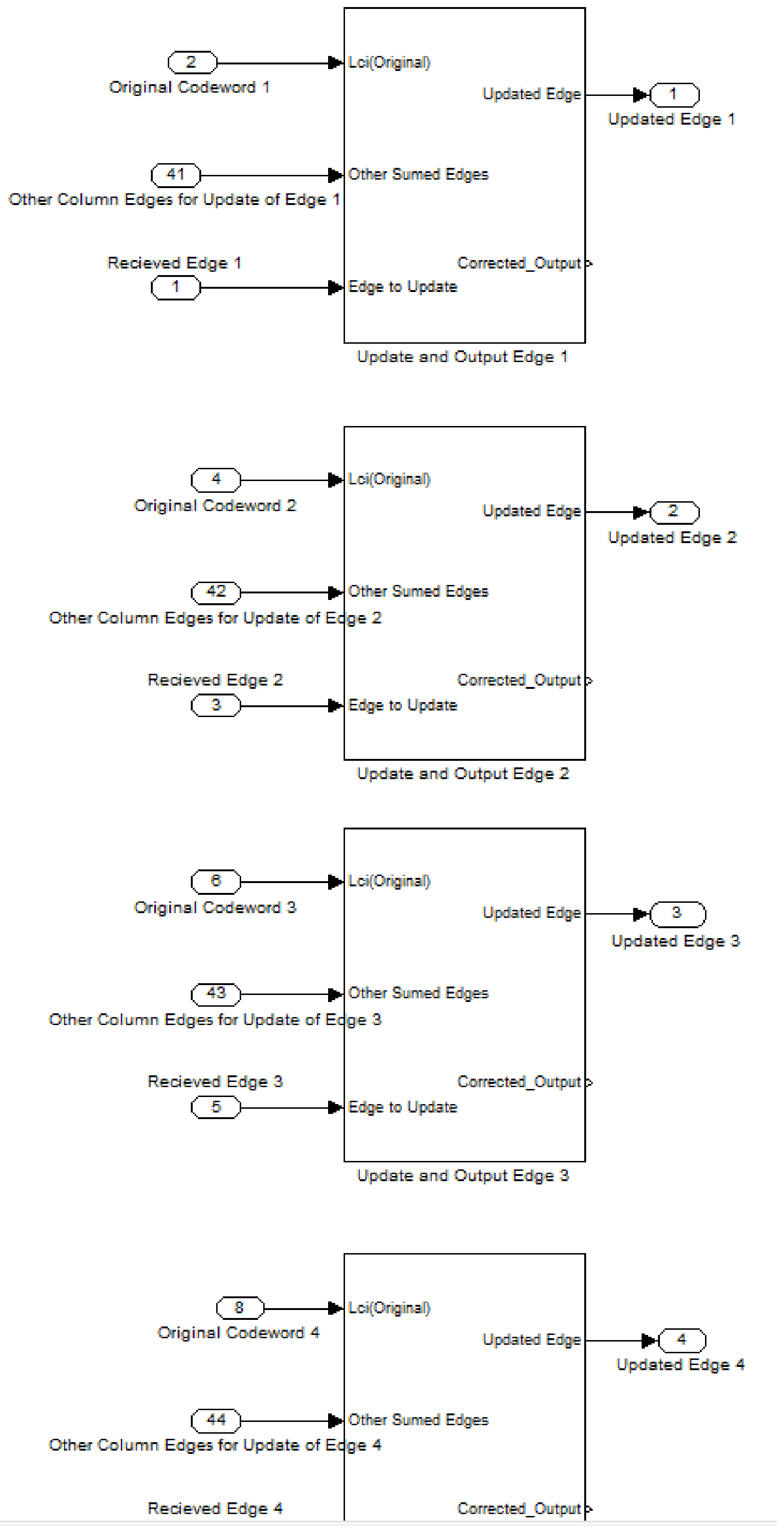


Figure 32: Variable Node Calculation Blocks

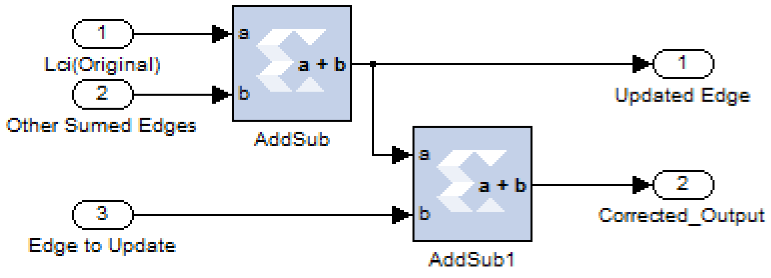


Figure 33: Variable Node Calculation Block Internal View

Figure 32 and 33 show how the calculations at the variable nodes are performed. Figure 31 shows a section of the 20 edge calculation blocks for each edge node. There are two outputs from each one of these edge calculation blocks. There is the output called “Updated Edge” which is the value to be used if another iteration is to be performed. The second output, called “Corrected_Output” is the final value the variable node calculated the corrected codeword to be, this output is used if no more iterations are to be performed. The two adder blocks original on how the MATLAB code in the Appendix specifies. The original Lci value is added to the other summed edges. For example, take a look at the Tanner graph in Figure 6, variable node 1 has two edges connected to it. Let us say that an update to edge 5 it to be performed, that is to be send back to check node 2. All edge values are taken, except for edge 5 and summed up, in this case there is only one, so no summation is needed. Thus, edge 1 is fed directly into the system, and labeled “Other Summed Edges”. This value is added to the original received value Lci and is the newly updated value for edge 5 and is then sent to check node 2. Another addition is performed inside the block and that is for the final corrected output as stated below. This is adding back the edge that was omitted in the previous calculation to the total sum, this value is then sent as the corrected output.

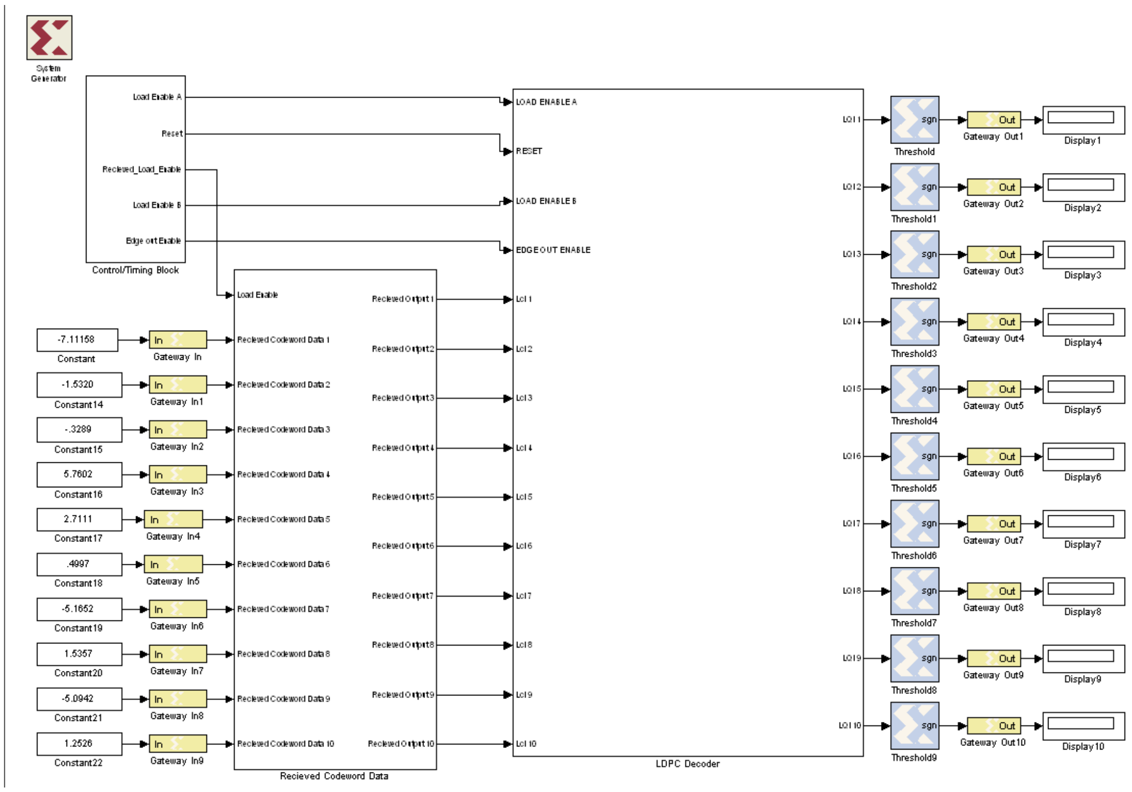


Figure 34: Complete Decoder System

Figure 34 shows the complete decoder system. The details of the “LDPC Decoder” block can be found in Figure 31. Two more blocks are needed to make the decoder in Figure 34 complete. The first is the “Control/Timing Block” which is in charge of controlling when to enable or disable certain blocks and when to reset values to zero. The other block is the one seen in the lower left. This is where the received values are stored, loaded into the “LDPC Decoder” block and also used for variable node calculations.

In Figure 35 below, the “Received Codeword Block” from Figure 34 is shown in detail. The value from the outside is loaded when the “Load Enable” signal goes high, otherwise when low, the value is saved in the register.

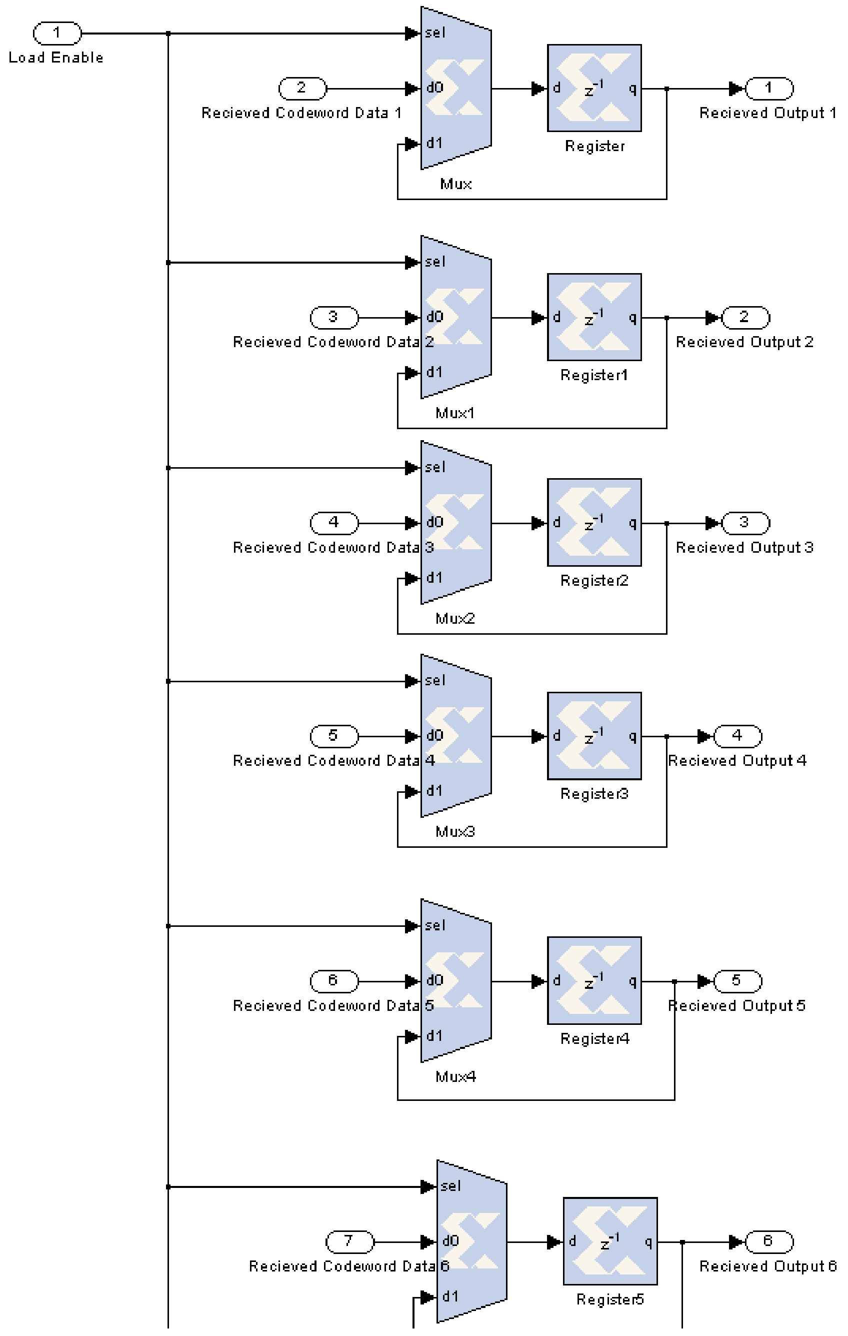


Figure 35: Received Codeword Block Internals

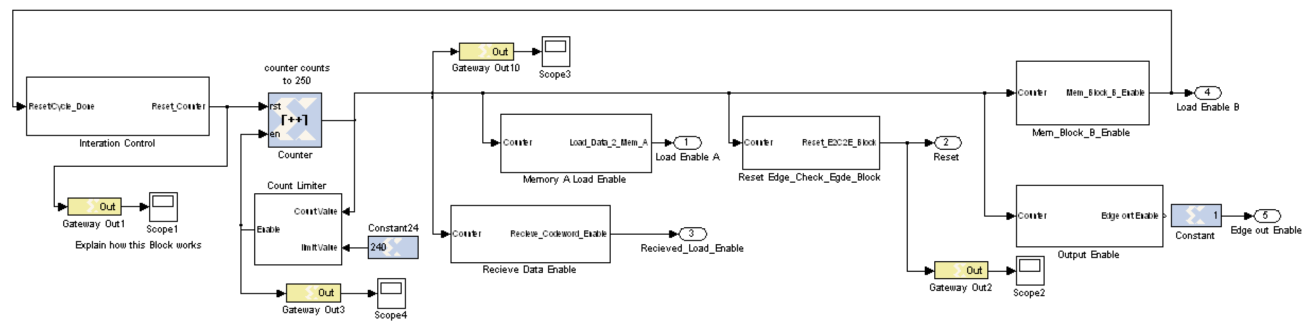


Figure 36: Control/Timing Block

This completes the main section of the decoder, but a way is needed to control the flow of data from each node block. This is done by the iteration control block that is seen above. The iteration control has the max number of iterations hard coded into it. This then enables the main count that counts to a value that allows the whole system to complete with enough time for each block to perform its operation. After this main count is done, it decrements the iteration count until the max number of iterations is completed. There are extra blocks that set either an enable or a reset high when counter gets to the corresponding count value. For example, when the counter is between counts X and count Y, the system knows to perform the check node calculation block. Otherwise we do not perform this calculation and hold this block at a reset value of 1, so nothing is performed here.

3. Conclusion

The objective of this project was to explore the use of FPGAs in the encoding and decoding process of LDPC codes. The system was developed and simulated using Xilinx system generator. Even though it was not downloaded and tested on an FPGA, this project stands to show the difficulties one may encounter when designing an LDPC system. The project also illustrates performance trade-offs that need to be considered in the design process.

Whichever decoding algorithm is chosen for an LDPC system, an approximated function or lookup table is necessary to remove non-linear functions for FPGA implementation. The BER performance of the system depends on the accuracy of the approximation or the lookup table.

3.1 Future Work

Some future work for a follow up to this project would include, but not limited to expanding the size of the G and H matrix dimensions. One of the memory blocks for the edge nodes could be eliminated, freeing up space in the system. Another change would be to optimize the system in terms of speed, size or power consumption.

References

[1] B. Sklar, Digital Communications: Fundamentals and Applications, Prentice Hall PTR, 2001.

[2] M. Karkooti, *Semi-Parallel Architectures For Real-Time LDPC Coding,* Houston, TX: Rice University, 2004.

[3] M. Valenti, "Iterative Solutions Coded Modulation Library," 3 October 2005. [Online]. Available: http://www.cs.wvu.edu/~mvalenti/documents/CmlTheory.pdf. [Accessed May 2013].

[4] T. Ta, "A Tutorial on Low Density Parity-Check Codes," [Online]. Available: http://www.ece.umd.edu/~tta/resources/LDPC.pdf. [Accessed 2013].

[5] M. P. C. Fossorier, M. Mihaljevic and H. Imai, "Reduced Complexity Iterative Decoding of Low-Density Parity Check Codes Based on Belief Propagation," *IEEE Transactions on Communications,* vol. 47, no. 5, pp. 673-680, 1999.

Appendix A

LDPC Simulation

LDPC using Gallager's method in log probability domain rewritten by In Soo Ahn on Feb. 23, 2007 Modified May 20, 2008 Modified April 8, 2013 by Matthew Pregara

```
clear all
clc
```

1) Simulation Parameters

```
Niter = 2;          % number of iterations
EsN0_dB = -3:1:7;   % SNR range to plot over
msg_num = 2000;     % number of messages per SNR value
```

2) Parity Check Matrix initialization

```
H =[1 1 1 1 0 0 0 0 0 0; ...
    1 0 0 0 1 1 1 0 0 0; ...
    0 1 0 0 1 0 0 1 1 0; ...
    0 0 1 0 0 1 0 1 0 1; ...
    0 0 0 1 0 0 1 0 1 1];

[rows, cols] = size(H);
[G_sys, H_sys] = gen_Gsys_from_H(H);
%check = mod(G_sys*H_sys',2);  % to see if orthogonal
```

3) Initialize the matrices

```
snr_range = length(EsN0_dB);
Lqij = zeros(rows, cols);
Lcix = zeros(rows, cols);
BER = zeros(1,snr_range);
BERuncoded = zeros(1,snr_range);
```

4) Channel Properties/Conversions

```
EbN0_dB = EsN0_dB+10*log10(2);  % Bit to Noise energy
EsN0=10.^(0.1.*EsN0_dB);        % Symbol to Noise energy
sigma2=1./(2*EsN0);             % Equivalent noise variance
amp = 1;                        % Transmitter amplitude
```

5) Generate messages

Generates all messages ahead of loop iterations. Thus, each iteration of the SNR loop uses the exact same messages.

```
m = randsrc(msg_num,rows,[0 1]);
```

6) SNR Loop

```
parfor p = 1:snr_range
```

```
% Space Allocation/Declaration is placed here so that MATLAB can safely
% or correctly parallelize the loop.
Lrji = zeros(rows, cols);
LQi = zeros(1, cols);
c_recovered = zeros(1,cols);
tot_errors = 0;
c_rx = zeros(1,cols);
rx_errors = 0;
```

7) Message Repition Loop

```
for t = 1:msg_num
```

7.1) Random Channel, and Receiver

```
    c = mod(m(t,:)*G_sys,2);%codeword
    c_bipolar = (2.*c-1);    %Baseband BPSK codeword
    r = amp.*c_bipolar + sqrt(sigma2(p)).*randn(1,cols); % received
```

7.2) Set channel posterior probabilities

```
    Lci=2.*r./sigma2(p);    % channel posterior prob of the RX codeword
    Lcix = repmat(Lci, [rows 1]);
    Lqij = H.*Lcix;
```

8) Decoding algorithm

Bulk of the program. Phi function is expensive.

```
    for n = 1:Niter       % number of iterations
        % update Lrji
        for j = 1:rows,   % update extrinsic msg from check node to variable node
            for i = 1:cols,
                if (H(j,i) == 1)
                    prod = 1;
                    sum_tmp = 0;
                    for k = 1:cols,
                        if k ~=i  && (H(j,k) == 1)
                            prod = prod * sign(Lqij(j,k));
                            sum_tmp = sum_tmp + phi(abs(Lqij(j,k)));
                        end  % of if
                    end % of k-loop
                    Lrji(j,i) = prod*phi(sum_tmp);
                end % of outer if
            end % i-loop
        end% % of j-loop

        % update Lqij
        for i = 1:cols, % update extrinsic msg from variable node to check node
            sum_tmp = 0;
            for j = 1:rows,
                if (H(j,i) == 1),
                    for k = 1:rows,
                        if k ~= j &&(H(k,i) == 1),
                            sum_tmp  = sum_tmp + Lrji(k,i);
                        end  % of if
                    end % of k-loop
                    Lqij(j,i) = Lci(i) + sum_tmp;
                end % of outer if
            end % of j
        end % of i

     % Update LQi
     LQi = Lci + sum(Lrji);
     end % of decoder iterations
```

8.1) Make Final Decision

```
        c_recovered(LQi>0) = 1;
        c_recovered(LQi<=0) = -1;
        c_rx(Lci>0) =  1; % hard clipped received
        c_rx(Lci<=0) = -1;
```

9) Find Number of errors

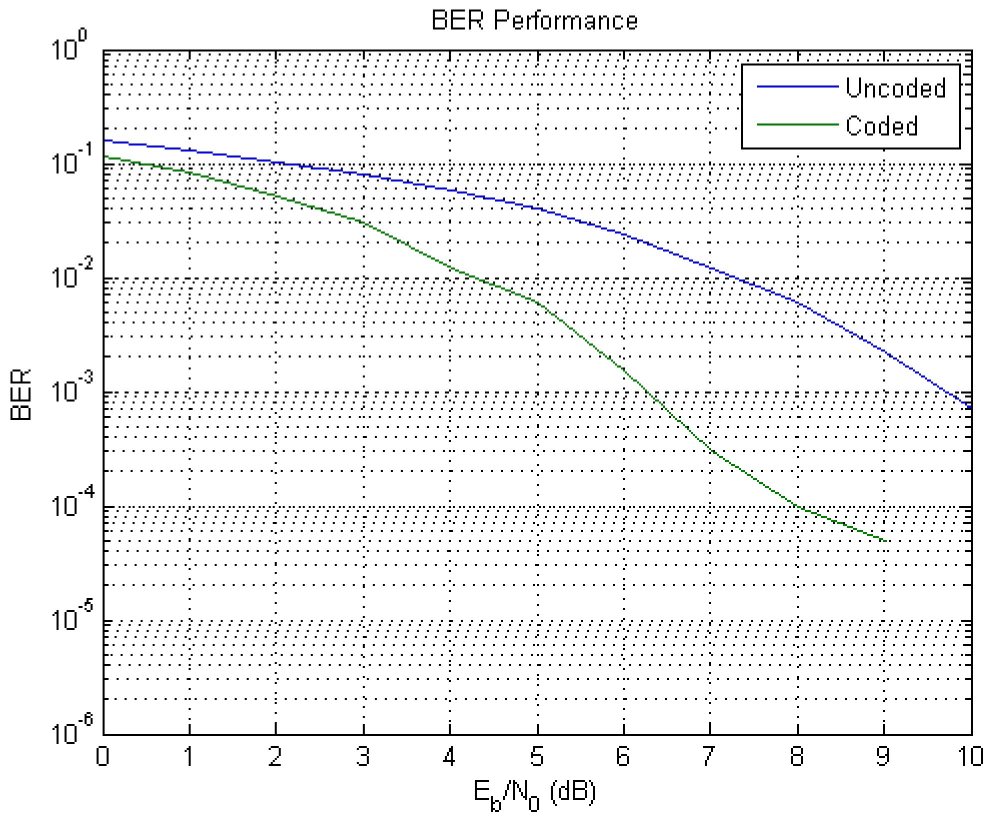
```
        diff = c_bipolar - c_recovered;
        tot_errors = tot_errors + length(diff(diff~=0));
        diffrx = c_bipolar - c_rx;
        rx_errors = rx_errors + length(diffrx(diffrx~=0));
    end % of t (message loop)

    BER(p) = tot_errors/(msg_num*cols);
    BERuncoded(p) = rx_errors/(msg_num*cols);
    tot_errors = 0;
end % of p (SNR loop)
```

10) Plot the results

```
semilogy(EbN0_dB, BERuncoded, EbN0_dB, BER)
grid on;
legend('Uncoded', 'Coded');
xlim([0 10]);
ylim([10e-7 1]);
title('BER Performance')
xlabel('E_b/N_0 (dB)')
ylabel('BER')

save phi_ln_msg20k_iter1.mat;
```



Published with MATLAB® R2012b