MAJOR PROJECT – EC498

IMPLEMENTATION OF LOW DENSITY PARITY CHECK CODES

Team Members

Akash A (191EC102)
Navratan (191EC133)

Mentor

Professor John D’Souza

National Institute of Technology, Karnataka Surathkal
NH66, Srinivasnagar Surathkal
Mangalore Karnataka – 575025

TABLE OF CONTENTS

1. PROJECT OBJECTIVES

The main objective of this project is to practically implement Low Density Parity Check (LDPC) codes at a hardware level and to verify its correct operation. A particular subclass of LDPC codes named Quasi Cyclic LDPC was adopted owing to ease of implementation. Methods to obtain the generator matrix in systematic circulant form followed by hardware encoding and decoding based on the Belief Propagation (BP) algorithm are implemented in C and Verilog.

2. INTRODUCTION

During the process of telecommunication, there are often errors introduced in the transmitted data no matter how good the channel may be. This occurs due to various sources of noise which may be either additive or multiplicative. At the receiver, our task is to inspect the received data and determine if any errors have occured and if yes, we further need to correct these errors so that we can obtain the transmitted data with as much integrity as possible. Errors may also occur with time when data is stored in memory. The process of error detection and correction is possible in the case of digital communication where we trasmit either 1's or 0's and hence the received data is also either a 0 or a 1. In analog communication schemes, error detection and correction is not possible since the transmitted data is in the form of continuous voltages or currents. This is one of the reasons why digital communication is widely used today. The process of error detection and correction is carried out by introducing extra bits known as parity bits into the channel along with the information bits. The parity bits add redundancy to the data in a structured way that facilitates reliable reconstruction of data at the receiver. The methodology of adding this redundancy is studied in the field of Error Control Codes and there are several classes of these codes available today which can achieve exceedingly low probabilities of error.

The question then arises as to what is the maximum capability can error correcting code can achieve. This answer was provided by Claude Shannon in his paper in 1948 where he showed that for a given channel with a particular bandwidth and signal to noise ratio (SNR), we could have reliable data transmission with arbitrarily small probabilities of error if the rate at which we are transmitted data is kept smaller than a maximum value. This maximum value is called the capacity of the channel and is related to the bandwidth and the SNR. For AWGN channels, the relation is given by:

$$C = B \log_2(1 + SNR)$$

Shannon showed that by using an arbitrarily long information block and by assigning an even longer code word to this block at random so that the final rate at which we are transmitting the information R is less than C, we can achieve almost zero probabilities of error. The limiting capacity value is called the Shannon limit and the process of forming blocks of information bits and then encoding them leads to a class of error control codes called block codes. Although this method helps us achieve our purpose, the complexity of encoding and decoding is huge for longer codes due to the large number of gates to be used. If the number of information bits is k, we would need to store a look up table with 2^k code words which would use too much storage for k values in the order of 1000. In order to reduce the complexity, we generally go for a subclass of block codes called linear block codes where we pick a set of *k* codewords that form a subspace and use the span of this subspace to generate all the 2^k code words. This is case for binary codes and the process can be easily extended to larger codes as well. The *k* code words are stored in memory in the form of a generator matrix **G** and logic is implemented at the encoder to generate linear combinations of the generator matrix to form the code words. In general, a code vector **v** can be obtained from the informatio vector **u** by premultiplying the generator matrix with the information vector.

$$v = u.G$$

Thus, the encoding complexity is simplified. The generator matrix is usually in systematic form where the information bits are grouped together to form a block followed by a block of parity bits. The systematic generator matrix is of the form:

$$\mathbf{G} = [\mathbf{I_k} \mid \mathbf{P}]$$

Here, $\mathbf{I_k}$ is a k x k identity matrix and **P** is a k x (n-k) parity check submatrix.

Similar to a generator matrix for encoding, every linear block code also has another matrix specified for it called the parity check matrix **H**. This matrix is formed by the null space of the code vectors in the generator matrix. Since the entire linear block code is formed by the row space of the generator matrix, any code vector in the linear block code must lie in the null space of the parity check matrix. Thus, at the decoder, to check if the code vector received is a valid code vector we can simply check if it lies in the null space of the parity check matrix. This comutation is called the syndrome **s** and is used for error detection. If there is an error, we will then need to identify the error and correct it. The parity check matrix in systematic form and the formula for syndrome computation is as shown below:

$$\mathbf{H} = [\mathbf{P^T} \mid \mathbf{I_{n-k}}]$$

$$\mathbf{s} = \mathbf{r}.\mathbf{H^T}$$

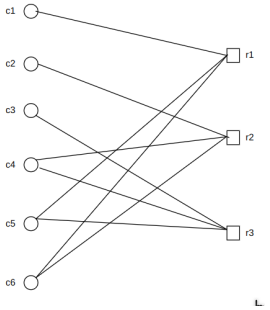
An optimal algorithm for decoding any linear block code is called maximum likelihood decoding where we compare the received code vector with each of the valid code vectors and select the valid code vector that has minimum distance from it. However, if we choose the information vector size to be large to get capacity achieving codes, there are a huge number of valid code vectors and hence the comparison becomes very difficult to perform. The solution to this issue was proposed by Gallager in the 1960s where he proposed the use of sparse parity check matrices and suggested an iterative decoding algortithm named Belief Propagation (BP) that was suboptimal, but could be implemented with low complexity. He showed that with a large number of iterations, the iterative algorithm could perform as well as the maximum likelihood decoding scheme. Such codes are called *Low Density Parity Check (LDPC)* codes and are the topic of discussion of this report.

2.1: Low Density Parity Check Codes

LDPC Codes are a subclass of error control codes for which the parity check matrix is a sparse binary matrix. The number of non-zero entries in the parity check matrix must be smaller than the number of rows in the matrix for the matrix to be considered sparse. Although the parity check matrix is sparse, it is not generally in systematic form. Thus, there is no guarantee that the parity check matrix in its systematic form or the systematic generator matrix is sparse. This means that a large number of code word bits are not zero after decoding and thus, the code will still have structured redundancy which helps to identify and correct errors.

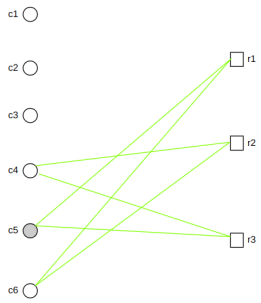
The parity check matrix in its sparse form is generally represented in the form of a bipartite graph called the Tanner graph which offers a lot of intuition while performing the encoding and decoding processes as well as the theoretical analysis of LDPC codes using techniques such as Density Evolution (DE). In the Tanner graph, the rows of the parity check matrix are represented on the right hand side in the form of check nodes. The columns of the parity check matrix are represented by convention on the left hand side in the form of bit nodes. For every non-zero entry in the parity check matrix, a connection is made between the bit node and check node corresponding to the row

and column at which the entry occurs. The Tanner Graph corresponding to a simple (6, 3) libear block code is as illustrated below along with its parity check matrix:



$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

An important observation that we can make from the Tanner graph is called the girth of the code. Girth is defined as the length of the shortest cycle in the Tanner graph that has the same start and end node and no other nodes or edges are repeated in the path. It is intuitive to see that due to the bipartite nature of the Tanner graph, the girth will always be an even number. Further, since there can only be a single direct edge between two nodes, girth of two is not possible. Thus, the Tanner graph has a girth of at least four. The girth of the parity check matrix is an important property which affects the performance of the code. The decoding of LDPC codes is generally based on belief propagation where probabilities are passed along the edges of the Tanner graph back and forth between the bit and check nodes. The prescence of cycles in the graph affects the independence of the passed probabilities and leads to poor performance. Thus, there is a need to realize parity check matrices with wide girth. In the parity check matrix we had described in the example above, we can observe that there is no cycle of length four. The girth of the Tanner graph is 6. One cycle of length 6 in the Tanner graph is illustrated below:



Example of Cycle in the Tanner Graph

It is common to define some structure in the parity check matrix of an LDPC code. One such way is where the number of ones in each row and column are equal. Such an LDPC code is called a regular LDPC code and is generally denoted as a (J, K) regular LDPC code where the parity check matrix contains J ones in each column and K ones in each row. If the parity check matrix does not contain a uniform number of ones in each row and column, the LDPC code realized is said to be an irregular LDPC code. It has been observed that regular LDPC codes are generally not capacity – achieving codes but give good performance in the threshold region of the BER – SNR curve while irregular LDPC codes are capacity–achieving and give good performance in the waterfall region but they tend to have large thresholds. Recent research attempts to exploit the advantages of both regular and irregular LDPC codes using Spatially Coupled LDPC (SCLDPC) codes where memory

is introduced into the encoder by introducing cross linking connections between indivual Tanner graphs of the linear block code. SCLDPC codes have been shown to outperform both regular and irregular LDPC codes in both the waterfall and threshold regions and are capacity – achieving. An example of a (2, 4) regular parity check matrix is as shown below:

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

3. CONSTRUCTION OF THE PARITY CHECK MATRIX:

In literature, there are two ways to generate parity check matrices: one is completely random computer – generated matrices with some specifications or constraints to ensure wide girth. Such parity check matrices are observed to have superior performance but the encoding and decoding complexities are larger due to the random nature of the matrix. A second method is to develop the matrix using some algebraic structure that simplifies the encoding and decoding procedure. Algebraically structured parity check matrixes have performance inferior to that of the randomly generated matrices but are shown to asymptotically approach the same performance for larger block lengths. We employ a specific subclass of structured matrices called quasi – cyclic parity check matrices (QCLDPC) which are large matrices composed of submatrices where the individual submatrices are circulants. A circulant is a matrix where every row is obtained by cyclically shifting the previous row by one position to the right and the first row is obtained by cyclically shifting the last row. An interesting property of any such circulants is that if ensure that the rows are obtained by cyclic shifts, the columns also become cyclic shifts of each other. The circulant nature of the submatrices greatly reduce the memory requirement since we only have to store the first row of each circulant and every other row is obtained by a cyclic shift of the first row.

We employ a particular subclass of QCLDPC codes where each of the circulants is a permutation matrix. A permutation matrix is an identity matrix whose every row is cyclically shifted by some integer. A permutation matrix is completely specified by specifying the shift of its first row from an identity matrix of the same size. Thus, the entire parity check matrix is completely specified by specifying a matrix of shifts from the identity matrix (the individual permutation matrices). Such a representation is called a protograph. Thus, in order to specify a QCLDPC parity check matrix based on permutation matrices, we simply specify the base matrix or protograph as shown below:

$$\boldsymbol{H}_{\boldsymbol{base}} = \begin{pmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,t} \\ h_{2,1} & h_{2,2} & \dots & h_{2,t} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ h_{c,1} & h_{c,2} & \dots & h_{c,t} \end{pmatrix}$$

Here each $h_{i,j}$ is an integer from 1 to b where b is the called the expansion factor. The base matrix described above forms a one-to-one correspondence with the larger parity check matrix where each entry in the base matrix is replaced with a permutation matrix of size b x b where the 1 in the first row appears at position $h_{i,j}$ in one-based indexing. The fnal parity check matrix is of the form as shown below:

$$H = \begin{pmatrix} H_{1,1} & H_{1,2} & \dots & H_{1,t} \\ H_{2,1} & H_{2,2} & \dots & H_{2,t} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ H_{c,1} & H_{c},2 & \dots & H_{c,t} \end{pmatrix}$$

It is intuitive that this procedure gives a (c x b, t x b) – regular LDPC parity check matrix. The resulting matrix also turns out to be sparse and thus, this gives a seneralized way of constructing a regular parity check matrix of any size. Thus, we only have to decide the entries in the base matrix. The important factor to take into consideration here is that we need a wide girth Tanner graph. It can be shown that if no element repeats within a row or column in the base matrix, the resulting parity check matrix will not have any cycles of length 4. Thus, it is possible to achieve a girth of atleast 6 by ensuring that there are no repetitions within a sinle row or column in the base matrix. This is known as the RC constraint. For small base matrices, we can make the RC constraint to be satisfied by randomly selecting values so that no values repeat within a row or column. For larger matrices, it becomes necessary to define a structured way to satisfy the RC constraint. One such method available in literature is to use the method of finite geometries to construct the code. Currently, we have implemented a QCLDPC code by arbitrarily selecting the base matrix to have a girth of atleast 6. Structural ways to realize larger sized base matrices remain to be studied.

4. ENCODING OF QCLDPC CODES:

In order to do the encoding of a linear block code, we need a systematic generator matrix. However, what we have constructed is a prity check matrix in circulant form which may not in the general case be systematic. Thus, we first need to realize, the generator matrix in systematic form. In order to have convenience while encoding, we tend to realize the generator matrix in the systematic circulant form where the generator matrix like the parity check matrix can be expressed as an array of circulants. We show in the succeeding section that the generator matrix when realized in systematic circulant form can be encoded using simple shift registers in linear time complexity thus minimizing both the area and delay of the hardware.

It must be noted that the circulant parity check matrix that we realized may or may not be full rank. The procedure to form the generator matrix is different in the two cases. We first present the case where the parity check matrix is full rank. The generator matrix in systematic circulant form can then be written in the form:

$$G = \begin{pmatrix} I & D \end{pmatrix} = \begin{pmatrix} I & O & \dots & O & G_{1,1} & G_{1,2} & \dots & G_{1,c} \\ O & I & \dots & O & G_{2,1} & G_{2,2} & \dots & G_{2,c} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ O & O & \dots & I & G_{c,1} & G_{c,2} & \dots & G_{c,c} \end{pmatrix}$$

where each $G_{i,j}$ is a b x b circulant, **I** is a b x b identity matrix and **O** is a b x b zero matrix. If the matrix **D** is full rank, we can show that the last c elements of the first row $\mathbf{z_i}$ of each block row of circulants is given by (since the product of each row in the generator matrix and the transpose of the parity check matrix must be a zero vector):

$$z_i = D^{-1} M u_i$$

where **M** is the submatrix formed by the first (t – c) block rows of **H** and $\mathbf{u_i}$ represents the first t-c elements of the ith block row of **G** in which only one element is 1 and the others are zero. Once we find $\mathbf{z_i}$, we can cyclically shift it to the right to find the elements of the other block rows and hence the entire generator matrix in circulant form is known.

The second case is where the parity check matrix **H** is not full rank or the matrix **H** is not invertible. In this case, we need to first form a matrix **D***by picking a set of *l* block columns from **H** such that c <= l <= t and the rank of **D*** is equal to the rank of the parity check matrix *r*. The original parity check matrix is then rearranged to form a new parity check matrix **H*** where the *l* block columns selected earlier are placed at the end of the new parity check matrix. The new parity check matrix after rearrangement can be written in the form:

$$\boldsymbol{H^*} = \left(\boldsymbol{M^*} \mid \boldsymbol{D^*}\right) \left(\begin{array}{cccc|ccc} \boldsymbol{H_{1,1}} & \boldsymbol{H_{1,2}} & \dots & \boldsymbol{H_{1,t-l}} & \boldsymbol{H_{1,t-l+1}} & \dots & \boldsymbol{H_{1,t}} \\ \boldsymbol{H_{2,1}} & \boldsymbol{H_{2,2}} & \dots & \boldsymbol{H_{2,t-l}} & \boldsymbol{H_{2,t-l+1}} & \dots & \boldsymbol{H_{2,t}} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \boldsymbol{H_{c,1}} & \boldsymbol{H_{c,2}} & \dots & \boldsymbol{H_{c,t-l}} & \boldsymbol{H_{c,t-l+1}} & \dots & \boldsymbol{H_{c,t}} \end{array}\right)$$

We can then find the generator matrix in systematic circulant form. The generator matrix that we need is an (t - r)b x tb matrix and the structure of the generator matrix is as follows:

$$\boldsymbol{G_{QC}} = \begin{pmatrix} \boldsymbol{G} \\ \dots \\ \boldsymbol{Q} \end{pmatrix}$$

The submatrices **G** and **Q** are both composed of arrays of circulants. The **G** submatrix is an (t - l)b x tb matrix of the form:

$$\boldsymbol{G} = \left(\begin{array}{cccc|ccc} \boldsymbol{I} & \boldsymbol{O} & \dots & \boldsymbol{O} & \boldsymbol{G_{1,1}} & \boldsymbol{G_{1,2}} & \dots & \boldsymbol{G_{1,l}} \\ \boldsymbol{O} & \boldsymbol{I} & \dots & \boldsymbol{O} & \boldsymbol{G_{2,1}} & \boldsymbol{G_{2,2}} & \dots & \boldsymbol{G_{2,l}} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \boldsymbol{O} & \boldsymbol{O} & \dots & \boldsymbol{I} & \boldsymbol{G_{t-l,1}} & \boldsymbol{G_{t-l,2}} & \dots & \boldsymbol{G_{t-l,l}} \end{array}\right)$$

The circulants $\mathbf{G_{i,j}}$ of **G** can be found in a manner similar to the first case by solving the matrix equation:

$$\boldsymbol{D^* z_i} + \boldsymbol{M^* u_i} = \mathbf{0}$$

The equation is guranateed to have a solution for $\mathbf{z_i}$ the rank of the matrix **D*** is r which is the same as the rank of the parity check matrix and the right hand side is a column from the column space of the parity check matrix which will also lie in the column space of **D***. The above equation is derived from the fact that the roduct of the generator matrix and the transpose of the parity check matrix must be a zero matrix.

The submatrix **Q** is an (l-r)b x tb matrix which is also composed of an array of circulants and can be written in the form:

$$Q = \left(\begin{array}{cccc|cccc} O & O & \dots & O & Q_{1,1} & Q_{1,2} & \dots & Q_{1,l} \\ O & O & \dots & O & Q_{2,1} & Q_{2,2} & \dots & Q_{2,l} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ O & O & \dots & O & Q_{l-r,1} & Q_{l-r,2} & \dots & Q_{l-r,l} \end{array}\right)$$

The entries $\mathbf{Q_{i,j}}$ can be found by solving the matrix equation:

$$\mathbf{D^* q_i = 0}$$

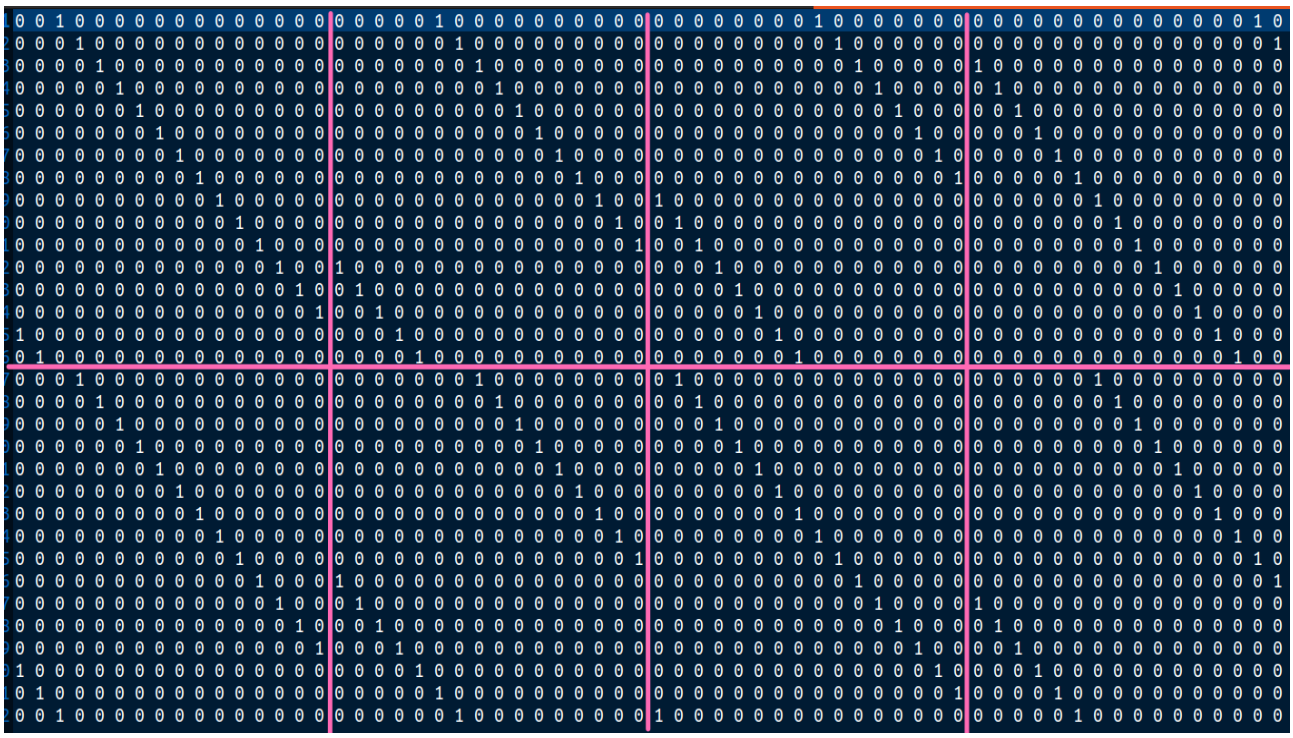
This is nothing but the null space of **D***. However, while solving this equation, it must be noted that the **Q** matrix must be found that its rows are all linearly ondependent and also linearly independent of the rows of **D**. This is done by finding the the number of linearly dependent colum ns in each of the block columns of **D***. Due to the cyclic structure of the block columns, we can regard the first set of columns to be linearly dependent and the rest to be linearly independent. When solving for $\mathbf{q_i}$, we set all the linearly dependent bits of $\mathbf{q_i}$ corresponding to the linearly dependent columns of **D***to zero except one bit which is the first bit in the ith block column. This gives the first row of the ith block row. The rest of the rows can be found by cyclically shifting the ith row. In this way the entire **Q** can be obtained. Once we get the two submatrices they can be stacked together vertically to get the entire generator matrix in systematic form.

4.1. Implementation:

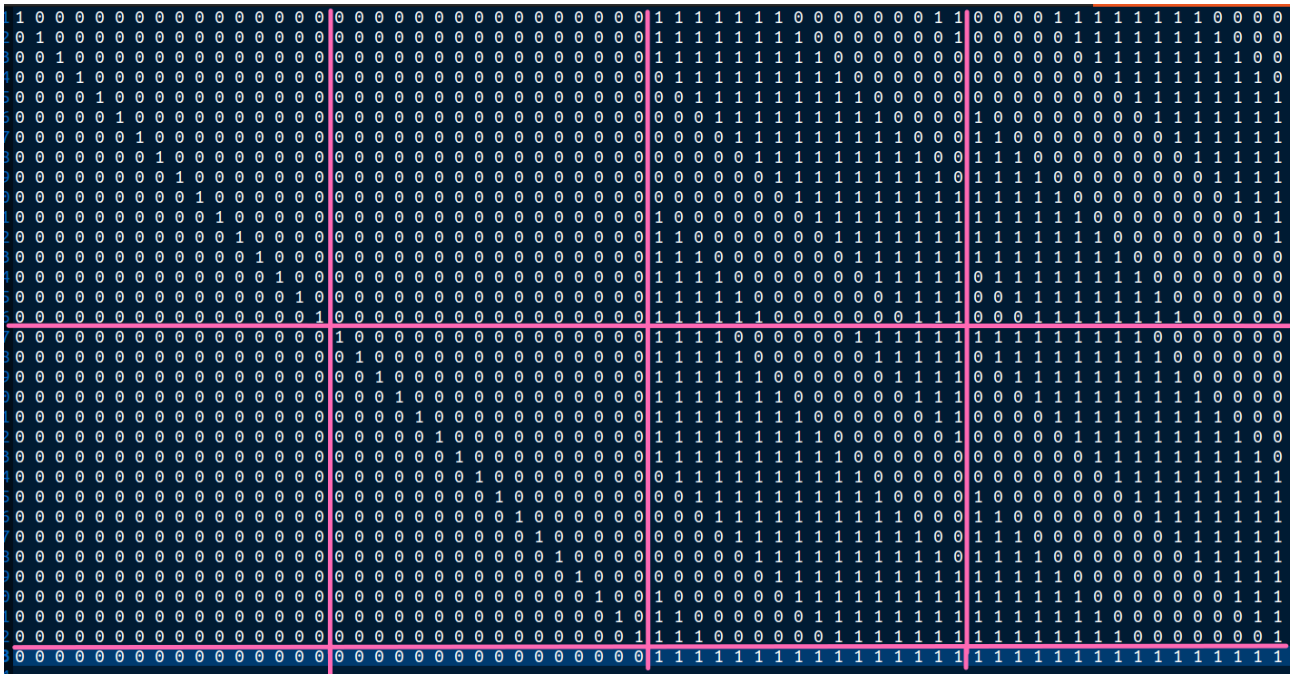
The base matrix was chosen to be a 2x4 matrix with an expansion factor b = 16. Thus, the entries in the matrix mustbe integers from 1 to 16. Since the base matrix was small, it could be easily chosen by trial and error to satisfy the RC constraint. The chosen base matrix is as shown below:

$$H = \begin{pmatrix} 3 & 6 & 9 & 15 \\ 4 & 8 & 2 & 7 \end{pmatrix}$$

This matrix was expanded by a factor 16 to get the final parity check matrix. The parity check matrix is as shown below:



This matrix was then row reduced by finding its row reduced echelon form. This was done by writing a C script. It was found that the parity check matrix had a rank of 31. Thus, it was not a full rank matrix and we had to adopt the second method of finding the generator matrix in systematic circulant form. We found by again by row reduction that the last 2 block columns of **H** together have a rank of 31 and hence could be considered to form the matrix **D***. The generator matrix required is a 33x64 matrix where the **G** submatrix is a 32x64 matrix and the **Q** submatrix is a 1x64 matrix. The generator matrix in systematic circulant form is as shown below:



It can be observed that each of the submatrices demarcated by the lines in the above figure is a circulant where succeeding rows are obtained by cyclically shifting the previous row. The **Q** submatrix is the given by the last row.

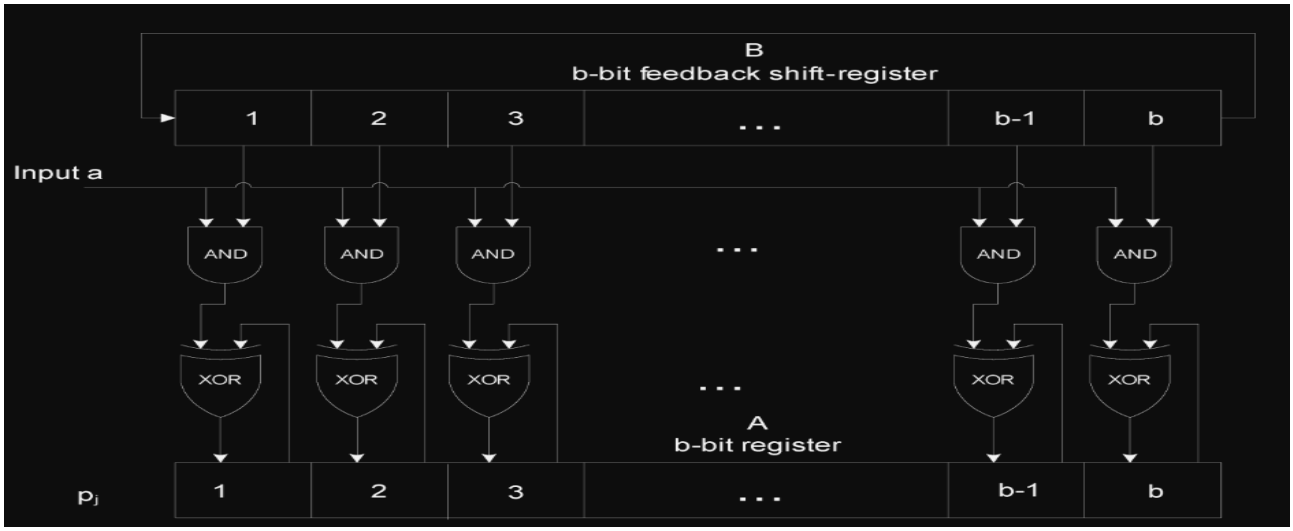
4.2. Hardware for Encoding:

An encoding circuit can be devised based on the circulant matrics of $\mathbf{G_{qc}}$ let $\mathbf{a} = (a_1, a_2, ..., a_{(tb-r)})$ be the information vector. **.** The encoded vector can be obtained by multiplying the information vector with the generator matrix. This is equivalent to a linear combination of the rows of **G** where the coefficients of the linear combination are the information bits. For a single circulant row of **G**, after the information bit is multiplied with the current row of the generator matrix, the next bit has to be multiplied with the cyclically shifted version of the previous row. This can be easily achieved by using a cyclic shift register to cyclically shift the row. Once a block row has been completed, the next block row is loaded into the cyclic shift register. Sums in each stage are accumulated in another register which stores the final result. The multiplication in a single step involves multiplication of a single information bit with an entire row and can be achieved using l*b AND gates. and can thus be realised using a single AND GATE. The hardware for encoding needs two registers and AND gates for each of the circulants of the generator matrix that is not the identity matrix or zero matrix. This circuit is called shift register add accumulator (SRAA) and is shown below:

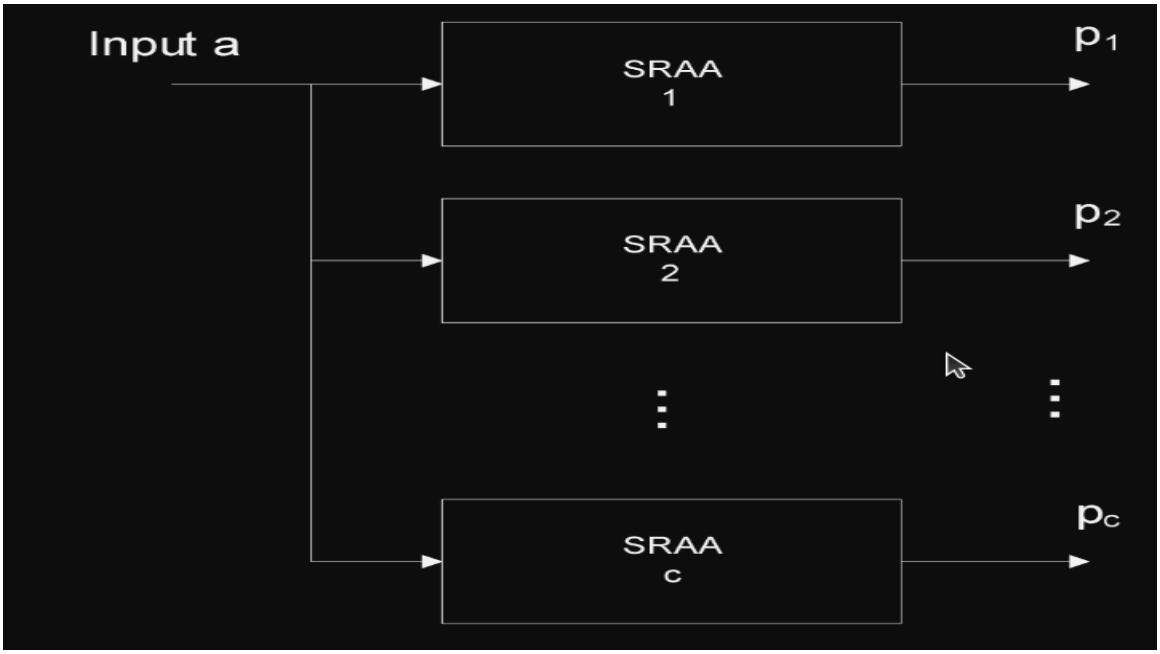
The information vector is divided into (l-r) sections where the first (t – r) sections contain b bits each and the last (l – r) sections contain number of information bits in each section equal to the number of linearly dependent columns in the column corresponding to that section which is also the number of rows in the ith block row $\mathbf{Q_i}$**.** The multiplication process is as shown below:

$$v = a_1 * G_1 + a_2 * G_2 + ... + a_{t-l} * G_{t-l} + a_{t-l+1} * Q_1 + ... a_{t-r} * Q_{l-r}$$

Here, each $\mathbf{a_i}$ represents a section of information bits as discribed above and $\mathbf{G_i}$ and $\mathbf{Q_i}$ represent the ith block rows of **G** and **Q** respectively. The evaluation of each term takes time equal to the number of information bits in that section and thus, the overall time taken is directly prpoportional to the number of information bits. The circuit thus operates with linear time complexity.



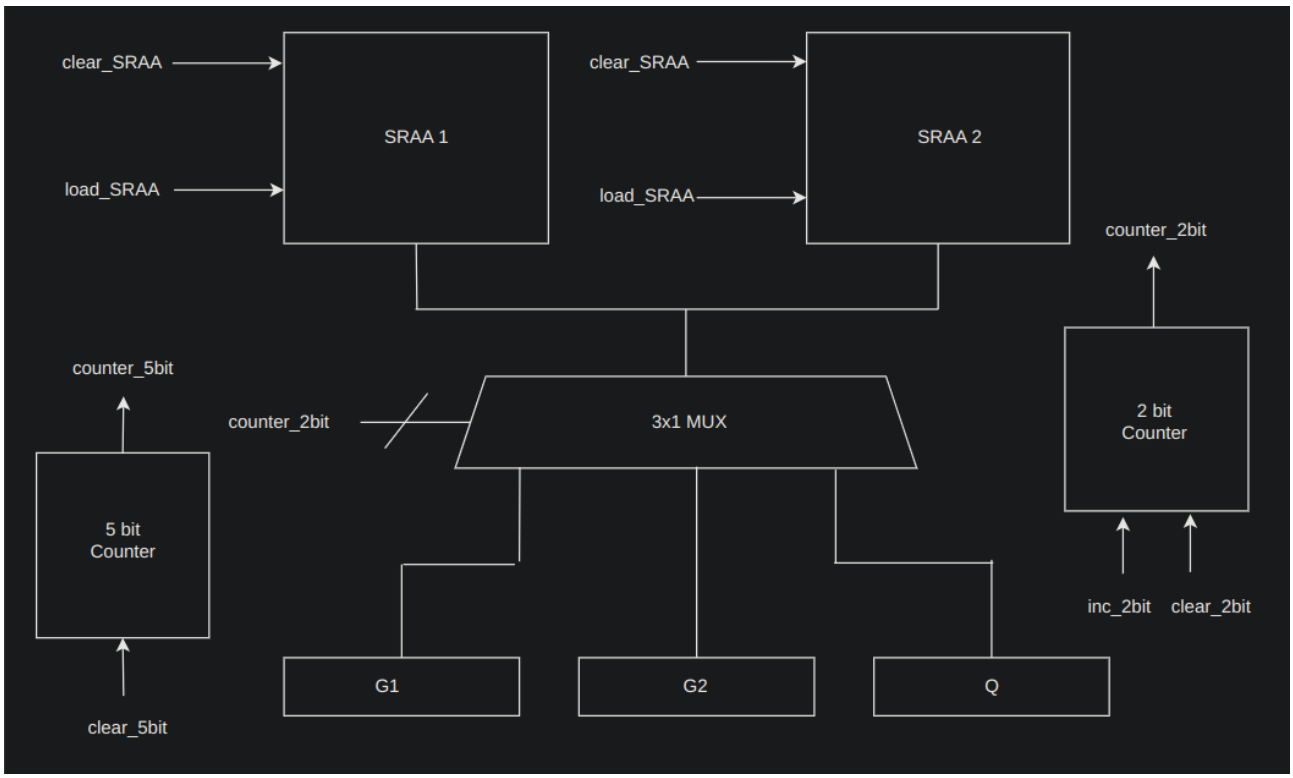
The circuit of the entire encoder needs l such SRAA circuits each for one of the circulants in the ith block row of **G** or **Q** and is as shown below:



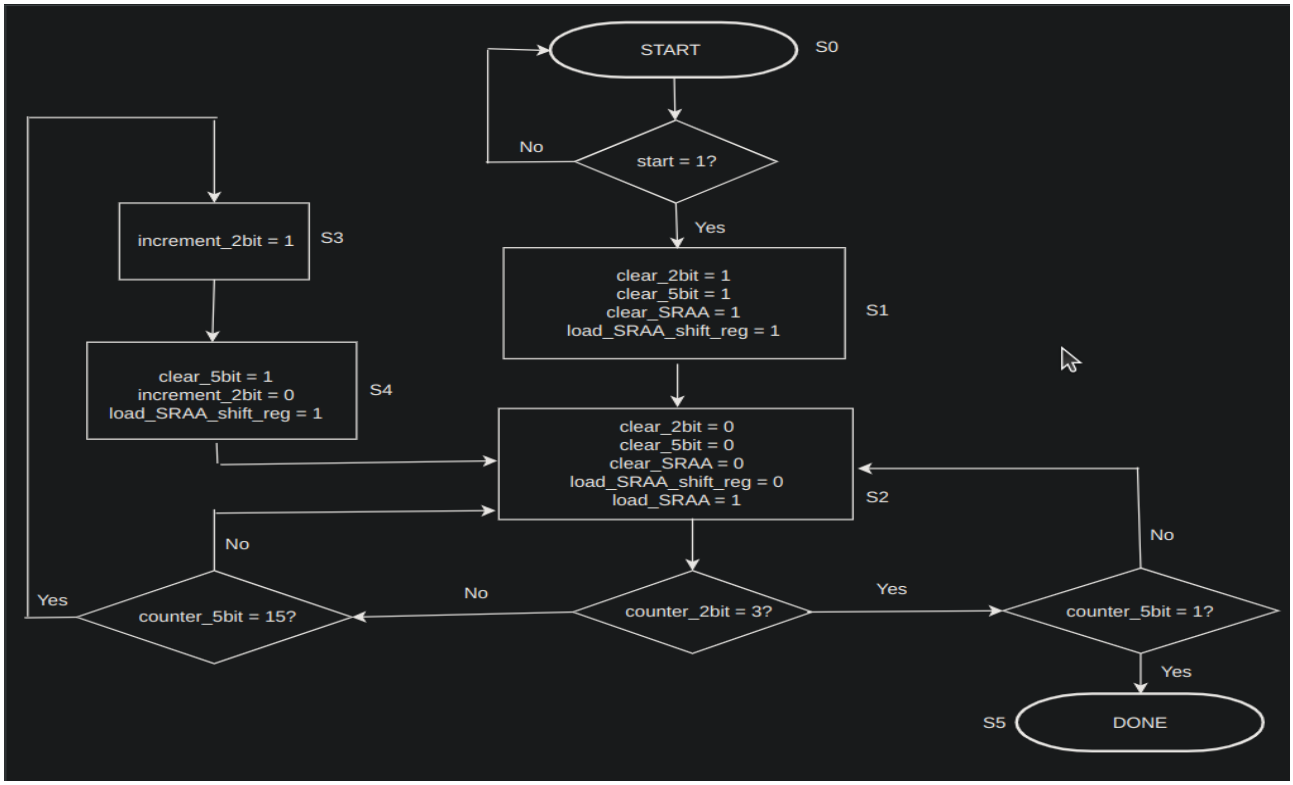
It is to be noted that this is the encoder hardware of only the first case where the parity check matrix is full rank and the matrix **D** is invertible. However, the similar hardware can be used for the second case as well. Further the hardware generates only the parity bits while the first (t – l) * b bits which are the unaltered information bits are also a part of the code word and need to be transmitted. The generator matrix is thus not fully in systematic form but only the first (t – l) * b bits are in systematic form and the encoded bits are same as the information bits.

4.3 Implementation of Encoding Hardware:

We used the circulant form of the generator matrix that was described earlier and modelled the SRAA hardware using Verilog to perform the encoding. Further, some additional hardware was required to ensure that the successive block rows are loaded into the shift registers at the right time. This was done using an RTL style approach where we modelled data paths and control paths and control signals were given to select the correct row from the generator matrix to be loaded into the shift registers. The data path and control path for the complete encoder are as shown below:



Data Path of the QCLDPC Encoder



Control Path of the QCLDPC Encoder

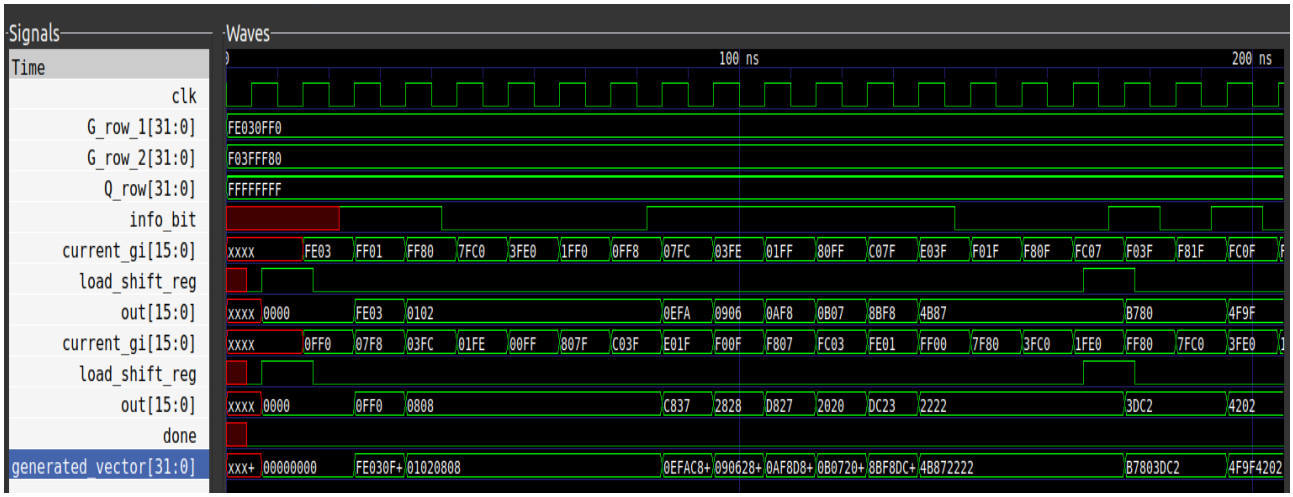
The encoder mainly works on the last two block columns of the circulant generator matrix which was shown earlier. Since there were two block columns, we required two SRAA units for each of the block columns. Further there are three block rows corresponding to $\mathbf{G_1}$, $\mathbf{G_2}$, and **Q** for which the

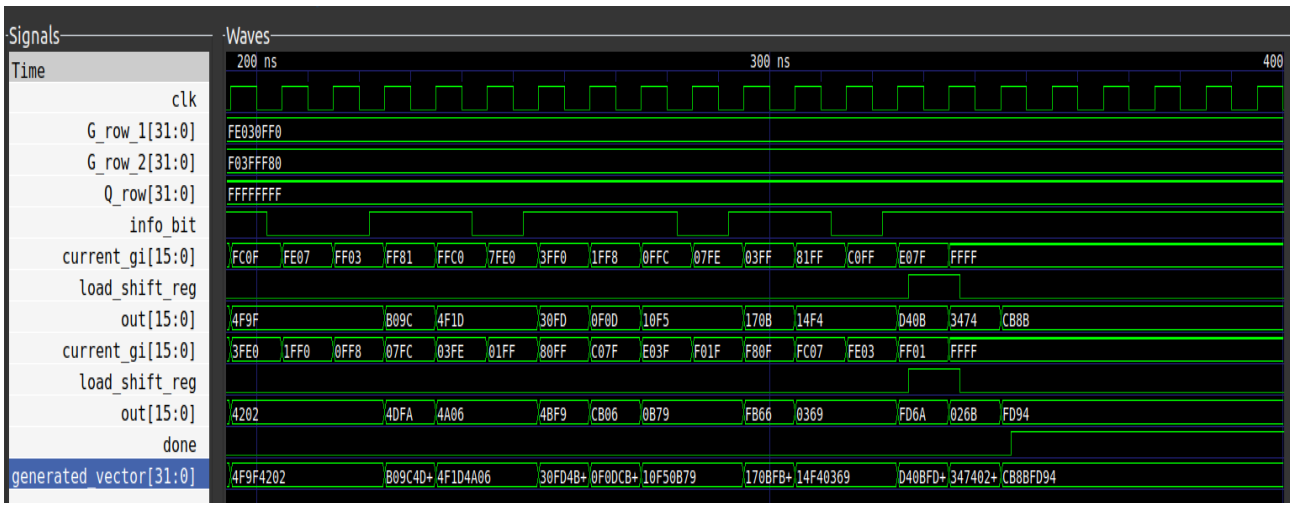
first row of these block rows are stored in three registers. A 3x1 multiplexer is used to select which block row is to be loaded into the shift register currently. The select lines for the multiplexer are the outputs of a 2-bit counter which is incremented at the right tim in the control path to select the correct block row. A 5-bit counter is used since each block row in the $\mathbf{G_i}$ submatrices consists of 16 rows that need to be processed. The output of the 5-bit counter is an input to the control path. In the control path, we first check the value of the two bit counter to see which block row is correctly being processed. If it is one of the $\mathbf{G_i}$, then the 5 bit counter is counted upto 16 times and if the value reaches 16, the counter is reset and the 2 bit counter is incremented. If the **Q** row is currently being processed, then we simple check for one iteration since the **Q** block row in this case has only one row. If one iteration is finished, then the encoder stops and we activate the “done” signal.

4.4 Results of the Encoder:

The data path and control path were modelled using Icarus Verilog and simulated using GTKWave waveform viewer. The encoder was tested for several case where 33 information bits in different combinations were provided and the parity bits were obtained from the simulation. Finally the first 32 message bits and the 32 parity bits were stacked together to form the final code word. In order to test if the final code word was correct, a C script was written to calculate the syndrome by multiplying the encoded vector with the transpose of the parity check matrix **H**. It was shown that for several randomly generated input cases tried, the syndrome always turned out to be the xero vector which verifies the correctness of the encoder. The waveform results and the generated parity vector for one particular case of the information vector is as shown below:

Information Vector Chosen **u** = (11000011111100010100110111011 0111)





Waveforms for the Encoder

```
VCD info: dumpfile QC_LDPC_ENCODER.vcd opened for output.
                  0: Last 32 Bits of Code Word = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                  7: Last 32 Bits of Code Word = 00000000000000000000000000000000
                 25: Last 32 Bits of Code Word = 11111110000000110000111111110000
                 35: Last 32 Bits of Code Word = 00000001000000100000100000001000
                 85: Last 32 Bits of Code Word = 00001110111110101100100000110111
                 95: Last 32 Bits of Code Word = 00001001000001100010100000101000
                105: Last 32 Bits of Code Word = 00001010111110001101100000100111
                115: Last 32 Bits of Code Word = 00001011000001110010000000100000
                125: Last 32 Bits of Code Word = 10001011111110001101110000100011
                135: Last 32 Bits of Code Word = 01001011100001110010001000100010
                175: Last 32 Bits of Code Word = 10110111100000000011110111000010
                195: Last 32 Bits of Code Word = 01001111100111110100001000000010
                225: Last 32 Bits of Code Word = 10110000100111000100110111111010
                235: Last 32 Bits of Code Word = 01001111000111010100101000000110
                255: Last 32 Bits of Code Word = 00110000111111010100101111111001
                265: Last 32 Bits of Code Word = 00001111000011011100101100000110
                275: Last 32 Bits of Code Word = 00010000111101010000101101111001
                295: Last 32 Bits of Code Word = 00010111000010111111101101100110
                305: Last 32 Bits of Code Word = 00010100111101000000001101101001
                325: Last 32 Bits of Code Word = 11010100000010111111110101101010
                335: Last 32 Bits of Code Word = 00110100011101000000001001101011
                345: Last 32 Bits of Code Word = 11001011100010111111110110010100
```

Verilog Results of the Final Codeword

The waveforms show how the individual contents of each of the SRAA units' shift registers and output registers change with time. The final parity bits are obtained by horizontally stacking the outputs of the two SRAA's. The final code word obtained by stacking both the information and parity bits is thus given by:

$$\mathbf{v} = (1100001111110001010011011101101111001011100010111111110110010100)$$

It can be verified either manually or by writing a C script that the syndrome of **v** is zero and it is thus a valid codeword.

5. DECODING OF LDPC CODES:

The decoding of LDPC codes as mentioned earlier is carried out using an iterative algorithm called ‘Belief Propagation’. We generally use what is called a soft decoder where instead of treating the received values as ‘0’ or ‘1’, we tend to look at them as like;lihoods. This is advantageous because it gives us more information about the received vector as compared to just two discrete values. For instance if we consider BPSK under AWGN, the transmitted voltage levels could be -1V for a zero and +1V for a 1. If the received value is -0.1V or -0.8V, it is treated the same in a hard decoder i.e. 0. However, these values become distinct in a soft decoder which provides us more advantage. Every row of the parity check matrix corresponds to a single parity check equation. The decoder that we build is a soft decoder where we pass log likelihood ratios along the edges of the ‘Tanner Graph’. Initially the received values are themselves treated as likelihoods and are passed from the bit nodes to check nodes. At each of the check nodes, the extrinsic likelihood for each of the bit nodes connected to it, is computed. The magnitude of the extrinsic likelihood is computed as the absolute minimum of all the bit nodes with exception to the bit node under consideration. The sign of the extrinsic likelihood is computes as the product of the signs of all the bit nodes excepting the one under consideration, multipied by -1. These likelihood values are sent back as messages from the check nodes to the bit nodes. At the bit node, the likelihood received from each of the check nodes tells us the probability about the same bit. Thus, this is similar to a repetition code and we can use the soft input , soft output(SISO) decoder for the repetition code. Thus, the entire belief propagation is a combinaton of two operations – a row operation where we find the minimum and second minimum and a column operation where we find the column sum of the matrix including the received value. After each row and column operation, the matrix is updated with new likelihood values and after a fix number of iterations the final column sum can be used to determine the receive vector after removing errors. Since we are using BPSK over an AWGN channel, any value greater than 0 represents logic 1 and any value less than 0 represents logic 0. This algorithm is known as the minsum algorithm or the message passing algorithm.

The steps in the decoding process can be summarized as follows:

Step 1: Initialize the parity check matrix. The number of received values is equal to the number of columns in the parity check matrix. Replce all the 1’s in the parity check matrix with the received value corresponding to that particular column.

Step 2: Perform row updation. For each row replace all non-zero elements with the absolute minimum of all the other non-zero elements for that particular row and calculate the sign accordingly. This can be done algorithmically by calculating the minimum of the row, its position and the second minimum of the row excluding the second minimum position. The minimum element is replaced by the second minimum and all other elements are replaced by the first minimum in that row.

Step 3: Calculate the sum of each column including the received value which gives the sum vector.

Step 4: Perform column updation. Each non-zero element in every column of the parity check matrix is replaced by the column sum of that column from the previous step after subtracting the value at that position in the parity check matrix. The subtraction is necessary to ensure that only extrinsic likelihoods are passed as passing intrinsic likelihoods could affect the performance of the algorithm.

Step 5: Repeat steps 2 to 4 for a fixed number of iterations.

Step 6: Once all the iterations are completed, the sum vector is used for decoding.If the values in the sum vector are larger than zero, it is interpreted as a '1' and if the values in the sum vector are smaller than '0', it is interpreted as a '0'.

5.1. C Implementation of the Belief Propagation Decoder:

We implemented the algorithm for SISO Belief propagation decoding for the simple (6, 3) parity check matrix in C. The number of iterations was taken to be 10. The parity check matrix used is as shown below:

$$\boldsymbol{H} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

The following were chosen for the encoded vector **v** and the received vector **r**.

$$\mathbf{v} = (1\ 0\ 1\ 1\ 0\ 1)$$

$$\mathbf{r} = (1.43, 0.1, 0.54, 0.23, 0.03, 0.85)$$

It can be observed that without decoding, the received vector would be decoded as (1 1 1 1 1 1) which would give rise to two errors. The first two and final iterations of belief propagation as realized from the C code is as shown below:

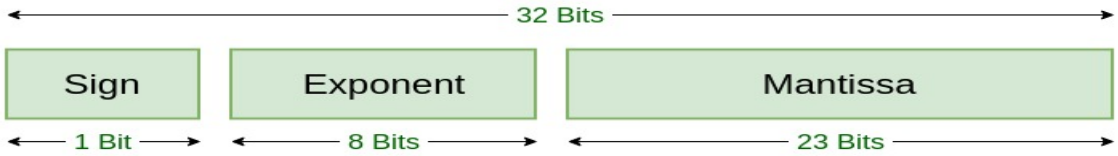
```
Initialized Storage Matrix:
1.430000 0.000000 0.000000 0.000000 0.030000 0.850000
0.000000 0.100000 0.000000 0.230000 0.000000 0.850000
0.000000 0.000000 0.540000 0.230000 0.030000 0.000000

Iteration 1
Check Node Updation:
-0.030000 0.000000 0.000000 0.000000 -0.850000 -0.030000
0.000000 -0.230000 0.000000 -0.100000 0.000000 -0.100000
0.000000 0.000000 -0.030000 -0.030000 -0.230000 0.000000

Sum Vector
1.400000 -0.130000 0.510000 0.100000 -1.050000 0.720000

Storage Matrix after Bit Node Update
1.430000 0.000000 0.000000 0.000000 -0.200000 0.750000
0.000000 0.100000 0.000000 0.200000 0.000000 0.820000
0.000000 0.000000 0.540000 0.130000 -0.820000 0.000000

Iteration 2
Check Node Updation:
0.200000 0.000000 0.000000 0.000000 -0.750000 0.200000
0.000000 -0.200000 0.000000 -0.100000 0.000000 -0.100000
0.000000 0.000000 0.130000 0.540000 -0.130000 0.000000
```

```
Sum Vector
1.630000 -0.100000 0.670000 0.670000 -0.850000 0.950000

Storage Matrix after Bit Node Update
1.430000 0.000000 0.000000 0.000000 -0.100000 0.750000
0.000000 0.100000 0.000000 0.770000 0.000000 1.050000
0.000000 0.000000 0.540000 0.130000 -0.720000 0.000000

...
...
...

Iteration 10
Check Node Updation:
0.100000 0.000000 0.000000 0.000000 -0.750000 0.100000
0.000000 -0.770000 0.000000 -0.100000 0.000000 -0.100000
0.000000 0.000000 0.130000 0.540000 -0.130000 0.000000

Sum Vector
1.530000 -0.670000 0.670000 0.670000 -0.850000 0.850000

Storage Matrix after Bit Node Update
1.430000 0.000000 0.000000 0.000000 -0.100000 0.750000
0.000000 0.100000 0.000000 0.770000 0.000000 0.950000
0.000000 0.000000 0.540000 0.130000 -0.720000 0.000000
```

We can observe that the final sum vector can be decoded as (1 0 1 1 0 1). The values are corrected in the first iteration itself but the likelihoods become stronger and more confident as we go towards higher number of iterations. Thus, we can observe that the belief propagation algorithm algorithm is working correctly. Further maximum likelihood decoding could correct only 1 error for this particular code but using this method, we are able to correct two errors which happened because of the soft nature of the decoder. This shows the advantages of a SISO decoder over a HIHO (Hard In Hard Out) decoder.

5.2 Verilog Implementation of the Belief Propagation Decoder:

Since Verilog is a Hardware Modelling Language for digital hardware, there is no separate provision for representing floating point numbers (The ral datatype that is available in Verilog is not synthesizable). Thus, different standards are available to represent floationg point numbers. We followed a popular standard kn own as the IEEE 32 bit floating point representation where each floating point number is represented by a set of 32 bits: the first bit is the sign bit which is ‘1’ for negative numbers and zero for positive numbers. The next eight bits correspond to the exponent and the least significant 23 bits correspond to the mantissa. While representing the mantissa, the leading ‘1’ to the left of the decimal point is ignored since it is obvious. The IEEE 32 bit floating point representation is illustrated in the figure below:



Single Precision
IEEE 754 Floating-Point Standard

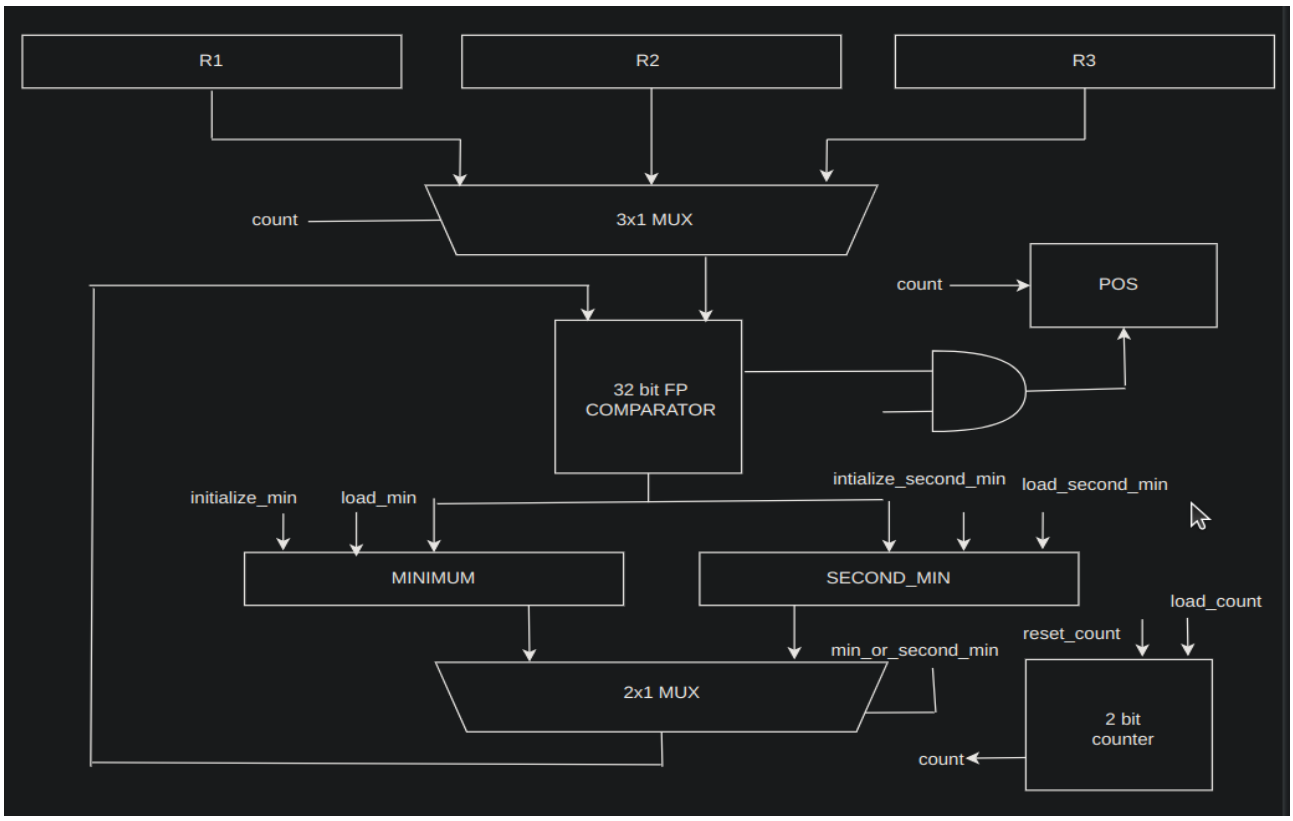
C scripts were written to convert the floating point numbers in the received vector to the IEEE standard representation and back. These were used to give input to the Verilog test bench and to test the final output for correctness.

Once the numbers were converted to their floating point representation, the next step was to implement the belief propagation algorithm. A bottom up approach was used where we first built row processors to process each of the rows individually followed by column processors to process columns individually. Each of the row and column processors was designed using an RTL based approach with individual state machines. The overall hardware was then built using a linked state machine where the control was given alternatively to the row and column processor and the iterations were carried out for a fixed number of times. After the iterations were completed, the the Verilog test bench was made to write all the reults into a text file similar to what we had done using the C code. The IEEE 754 standard numbers were then converted back to their actual floating point representation and the results were matched with the C scripts to check for correctness. A detailed description of the implementation of the row and column processors is as described below:

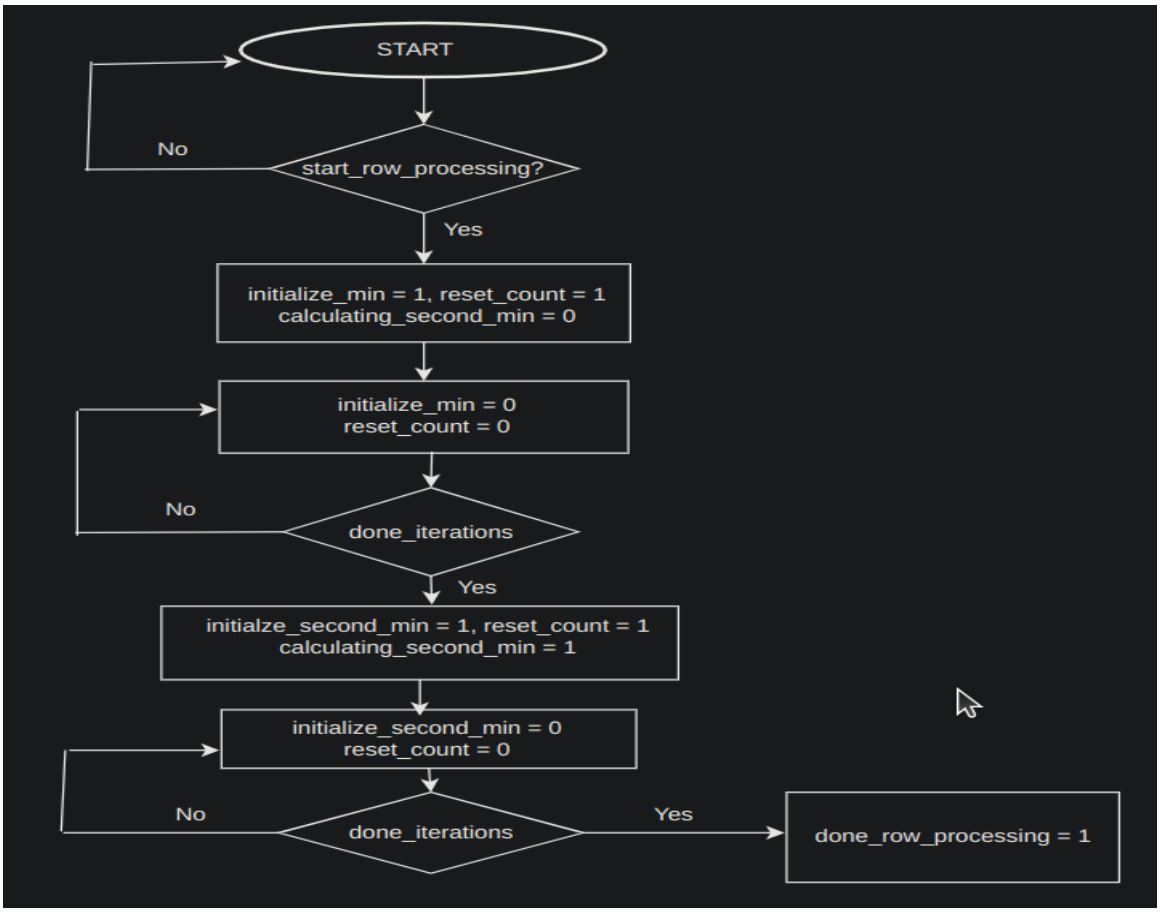
5.2.1 Row Processor: Calculation of Minimum and Second Minimum

The row processor had to calculate the minimum and second minimum of a set of three floating point numbers. Although this would be trivial to realize since there were only three numbers, in order to make the code extendible to larger block lengths, we used an iterative approach with a counter to compare each of the numbers. Each of the registers would be checked two times once to calculate the minimum and once to calculate the second minimum. The position of the minimum register was also stored in a separate register and is used when calculating the second minimum to skip that particular position. The data path and control path of the hardware to calculate the minimum and second minimum is as shown below.

The data path consists of three 16 bit input registers to store the three inputs. There are also two 16 bit output registers to store the minimum and the second minimum and a 2 bit register to store the position of the minimum. Initially the minimum and second minimum registers are set to all one’s. Then a 3 to 1 16 bit multiplexer is used to select each of the input registers and compare it with the minimum and second minimum registers. The select lines for the multiplexer comes from the output of a 2 bit counter which successively selects each of the registers. In the control path, first the minimum register is loaded and after the minimum has been computed, the second minimum is calculated. When calculating the second minimum, the position at which the first minimum occurs has to be skipped. This step is handeld in the counter by making the counter skip one of its states while incrementing.



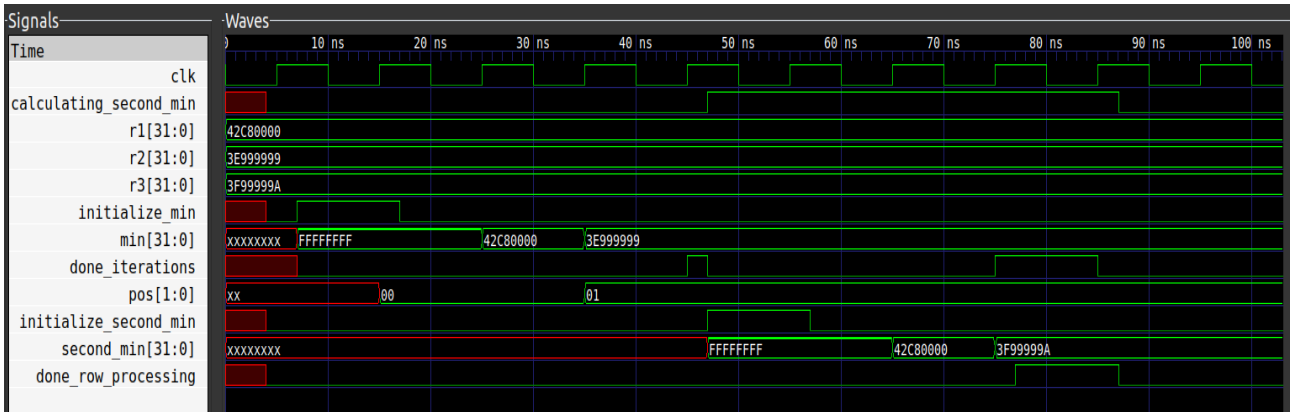
Data Path of Minimum and Second Minimum Calculator



Control Path of Minimum Second Minimum Calculator

Results of Calculation of Minimum and Second Minimum:

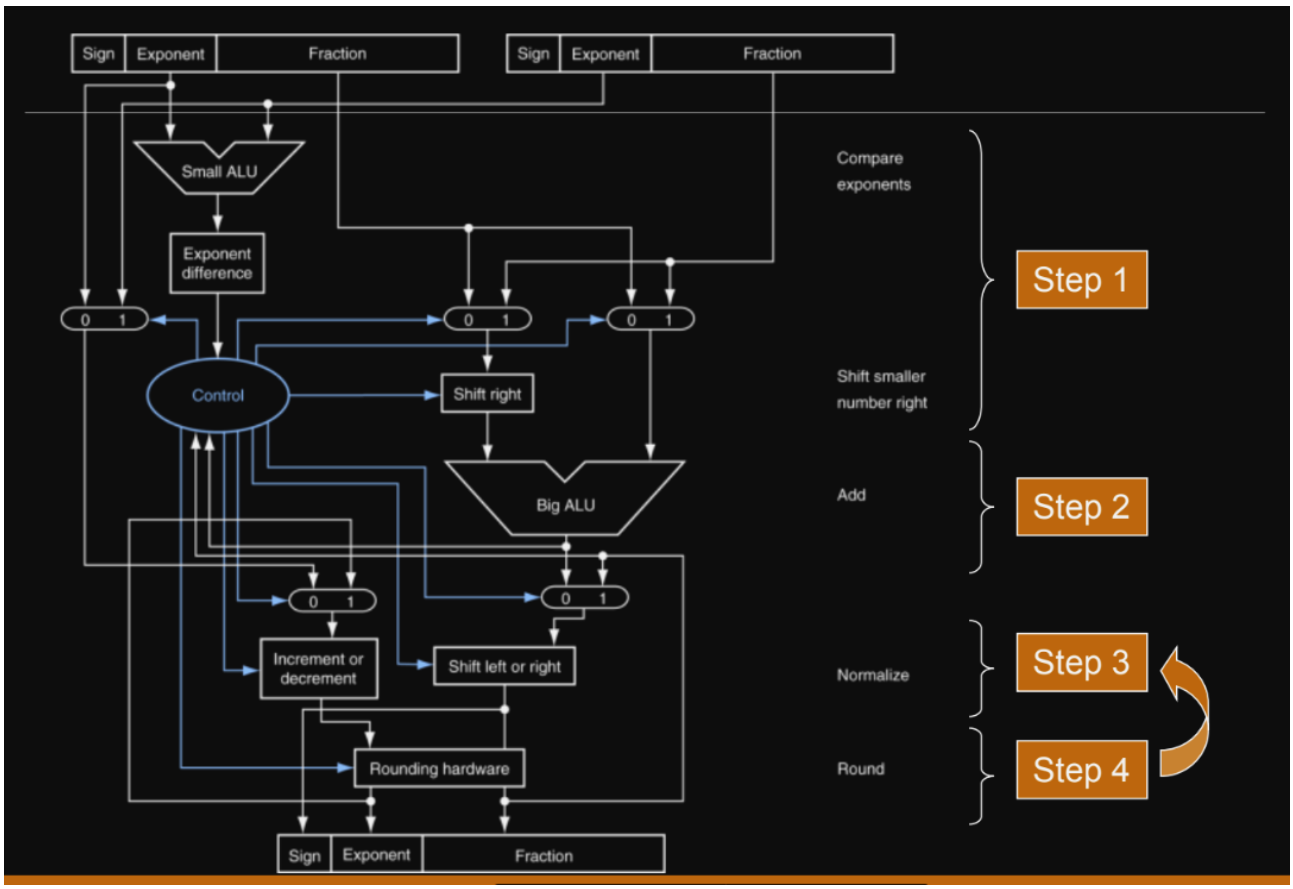
```
  0: Minimum = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, Position of Minimum = xx, Second Minimum = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  7: Minimum = 11111111111111111111111111111111, Position of Minimum = xx, Second Minimum = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 15: Minimum = 11111111111111111111111111111111, Position of Minimum = 00, Second Minimum = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 25: Minimum = 01000010110010000000000000000000, Position of Minimum = 00, Second Minimum = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 35: Minimum = 00111110100110011001100110011001, Position of Minimum = 01, Second Minimum = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 47: Minimum = 00111110100110011001100110011001, Position of Minimum = 01, Second Minimum = 11111111111111111111111111111111
 65: Minimum = 00111110100110011001100110011001, Position of Minimum = 01, Second Minimum = 01000010110010000000000000000000
 75: Minimum = 00111110100110011001100110011001, Position of Minimum = 01, Second Minimum = 00111111100110011001100110011010
```



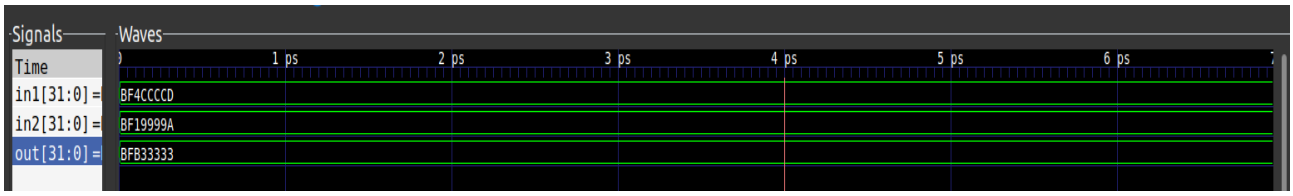
The above figures illustrate the calculation of minima and 2nd minima of 3 registers. The registers were intialised to values r1=100, r2=0.3, r3=1.2. We can observe that after the entire processing is done the value in the minimum register is 0.3 and the value of pos is 01 which represents r2. The value in the 2nd minimum register is 1.2. Thus, we verify that the row processor is operating correctly.

5.2.2 Column Processor: Calculation of Column Sums

In the column processor we need to evaluate the sum of individual columns along with the receive values. Since the (6,3) parity check matrix has atmost two ones in a single column, the maximum number of values we would need to add is 3 when we include the receive value. In order to implement the column sum calculator, we employ the design of a floating point adder to add two numbers. Addition of floating point numbers is much more difficult when compare to addition of normal binary numbers. In order to add floating point numbers in hardware, the exponents of the two numbers must first be matched. Normally, the number with smaller exponent has its mantissa shifted by an amount equal to the difference of the two exponents. The shifting is carried out using a barrel shifter. The mantissa are then added and the result is then again normalised to get it back to standard representation. In some cases, rounding hardware is also necessary but we didnot use this for simplicity. Normally, floating point addition has large delays and takes several clock cycles but we have implemented it in a single clock cycle for testing purposes. The data path of the floating point adder is as shown below:



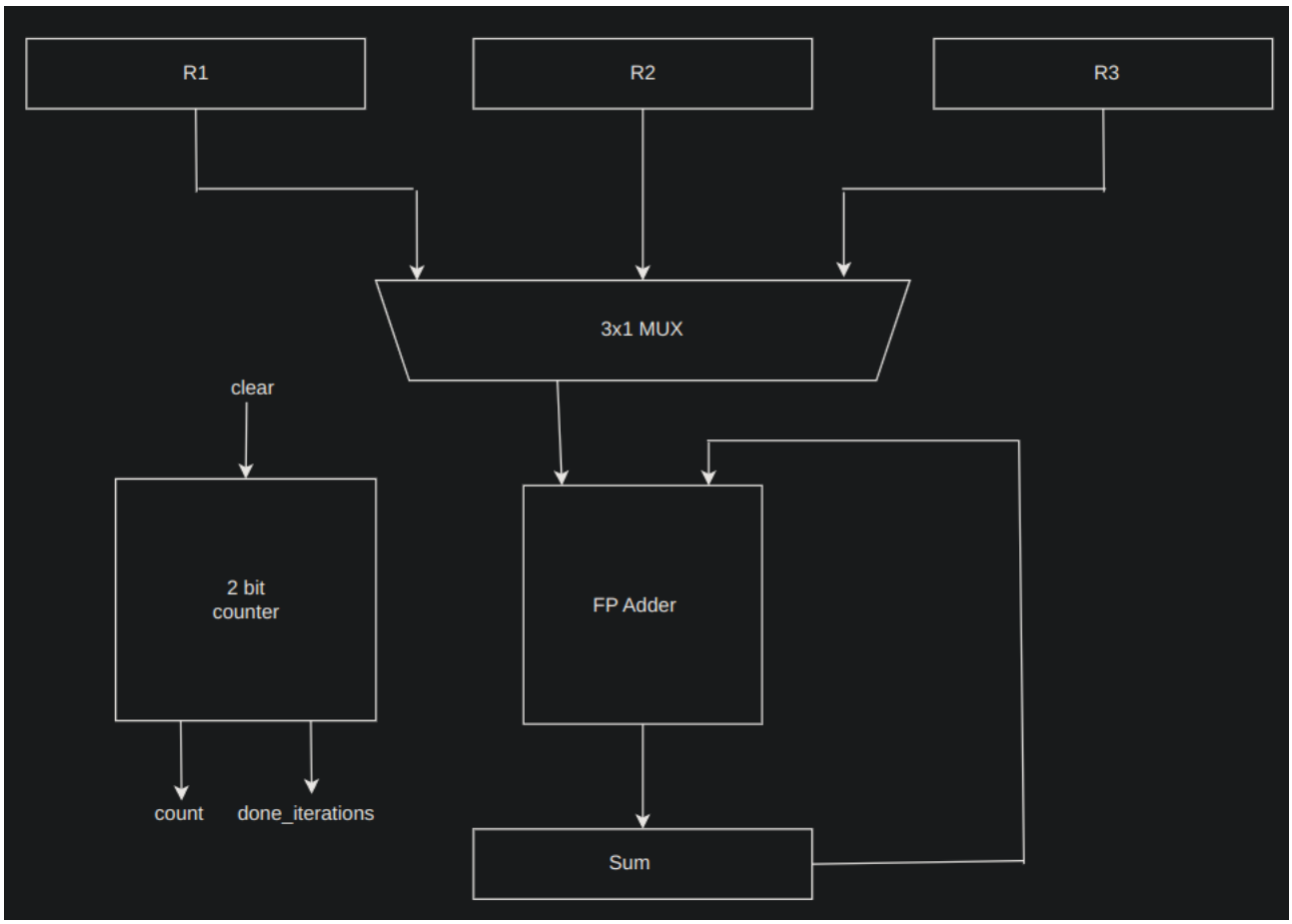
The results of the floating point adder implemented using Verilog are shown below:



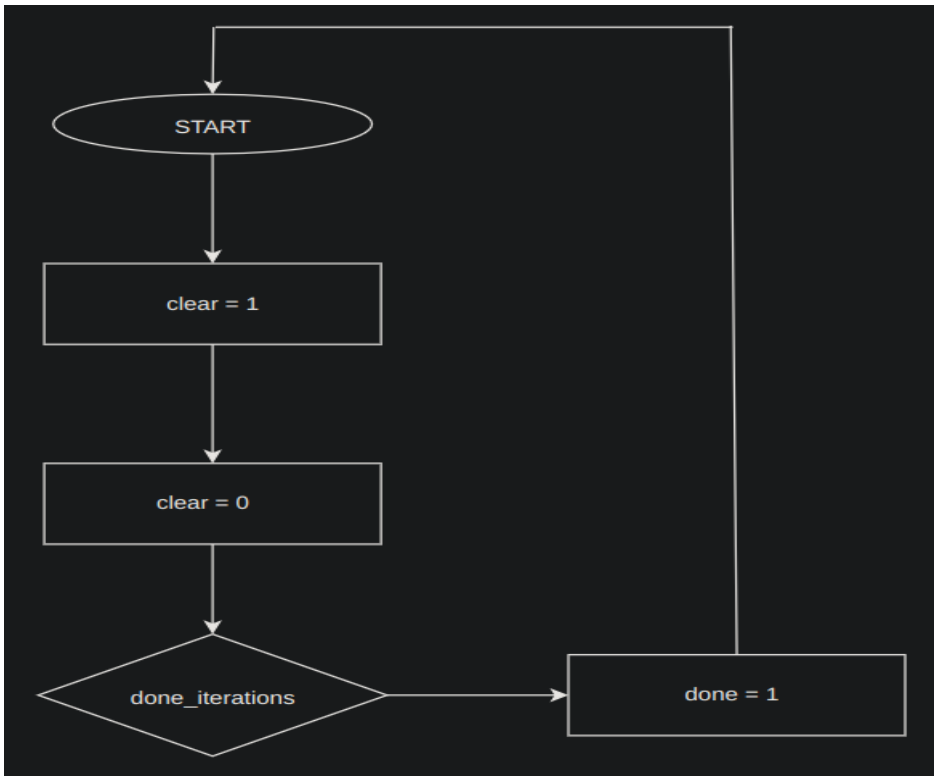
The floating point adder implemented is a signed adder. The two inputs are choosen to be -0.8 and 0.6 It can be observed that the final result is the floating point representation of -0.2 which is the expected sum. The adder was verified for several other test cases and was observed to give correct results. It can be observed that in the ideal case the delay of the adder is 0, but in the practical case the effect of delays on the timing of overall circuit needs to be taken into consideration.

This adder adds two numbers individually. We need to realise an accumulator which can add three numbers or in general any number of floating point numbers. In order to do this we used the floating point adder as a block and used RTL based design style to build the accumulator. The data path and the control path of the accumulator are as shown below.

In the data path, the floating point adder is used as a block unit and the sum register is always one of the inputs to the floating point adder. Initially, the sum register is cleared to zero and the registers in the column are then individually selected using a multiplexer. The select lines to the multiplexer are formed by a two bit counter whicb is incremented to 3. At this point all the register values have been added and the final result is available in the sum register.

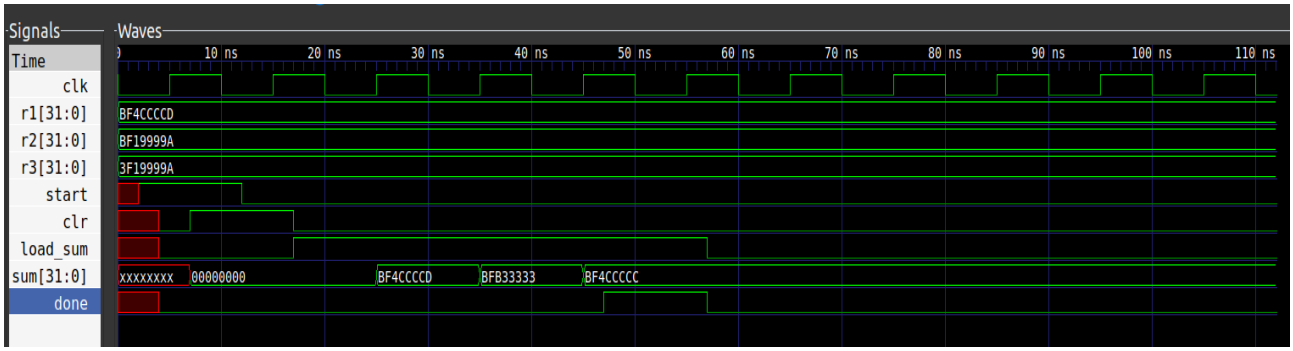


Data Path of the Accumulator



Control Path of the Accumulator

Results of the accumulator are as shown below:



Verilog Waveform of Accumulator

```
 0: Sum = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, Done = x
 4: Sum = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, Done = 0
 7: Sum = 00000000000000000000000000000000, Done = 0
25: Sum = 10111111010011001100110011001101, Done = 0
35: Sum = 10111111101100110011001100110011, Done = 0
45: Sum = 10111111010011001100110011001100, Done = 0
47: Sum = 10111111010011001100110011001100, Done = 1
57: Sum = 10111111010011001100110011001100, Done = 0
```

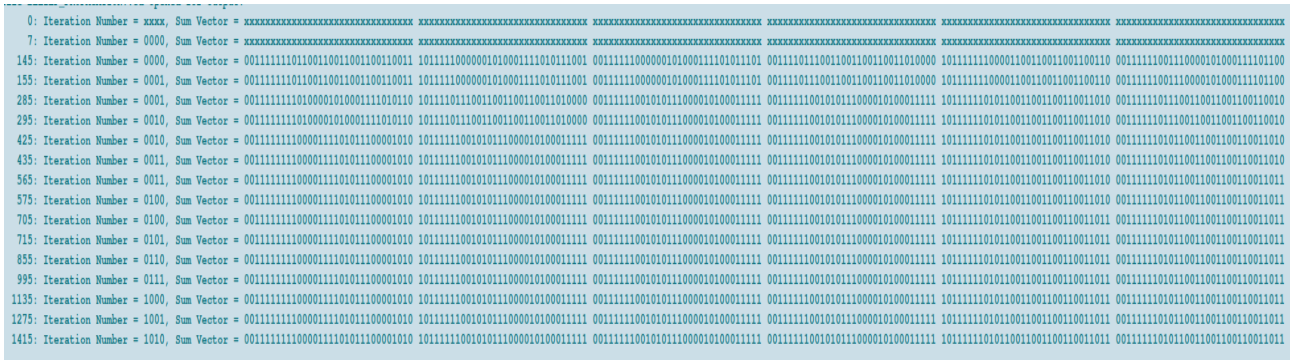
Final Output of Accumulator

The accumulator shown above was intialised with values -0.8, -0.6 and 0.6. The final value in the sum register was found to be -0.8 which is the correct value. Several other testcases were tried and the results was successfully varified. Thus, we have realised a column processor that is working correctly.

5.2.3 Complete hardware realisation of the belief propagation decoder:

The complete hardware for belief propagation was realised using the row and column processor created above. Since we have three rows and six columns, three row processors and six column processors were instantianted. The control path was designed using a linked state machine. In the first state, once the start signal was given the parity check matrix is first intialised with the received values. Control is then given to the row processor and row operation is done. Once the row processor gives the done signal, the parity check matrix is updated. Control is then given to the column processor, which calculated the column sum. Once this is done, the parity check matrix has to be again updated. But before doing this, we must first subtract the column sum value and the current value stored in the parity check matrix. This is done using the floating point adder and flipping the most significant bit of the values stored in the parity check matrix. Once the subtraction is done, the parity check matrix is again updated with the result value. Control is again given to the row processor and the process continues iteratively.

The final result of the belief propagation decoder are as shown below:



The screenshot above shows the results of the sum vector after each of the iterations. C scripts were written to convert the floating point representations back to their original form and the results matched exactly with the belief propagation results from the C implementation.

6. FUTURE WORK:

1. Development of testing strategies to verify the correctness of the algorithms exhaustively.
2. Extension of the current encodig and decoding algorithms to larger block lengths.
3. Constrution of parity check matrices using algebraic techniques.
4. Exploration of LDPC codes of different code rates.
5. Moduling an AWGN channel and plotting the BER VS SNR curve to demostrate performance.

7. REFERENCES:

1. **S. Lin and D.J. Costello** "*Error Control Coding; Fundamentals and Applications*", Pearson 2005

2. **Zongwang Li** et all "*Efficient Encoding of Quasi-Cyclic LDPC Codes*", IEEE TRANSACTIONS ON COMMUNICATIONS, VOL. 54, NO. 1, JANUARY 2006

3. **Professor Andrew Thangaraj,** "LDPC and Polar Codes used in the 5G Standard" NPTEL Lecture Series

4. **Helia Naeimi and Andre Dehon** "Fault Secure Encoder and Decoder for NanoMemory Applications", IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 17, NO. 4, APRIL 2009

8. GUIDE APPROVAL: