Design of a min-sum LDPC decoder for error correction

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Engineering in Microelectronic Systems

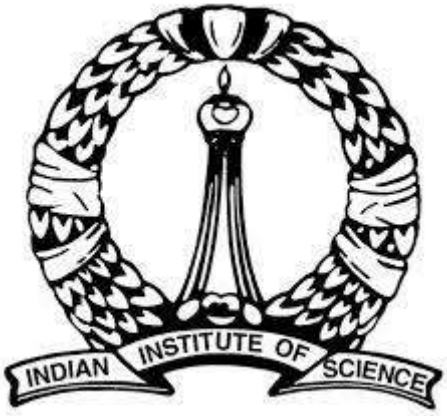
IN THE FACULTY OF ENGINEERING

by

Arijit Mondal

under the guidance of

Prof. Shayan G. Srinivasa



DEPARTMENT OF ELECTRONIC SYSTEMS ENGINEERING

INDIAN INSTITUTE OF SCIENCE

BANGALORE, INDIA

JUNE 2014

Abstract

Low-density parity-check (LDPC) codes are used in storage systems since they achieve error-correction performance very close to the Shannon limit for large block lengths. LDPC block codes admit parallel decoding with low complexity hardware implementation. In this thesis, we investigate into the design architecture of an array type LDPC code using non-layered min-sum algorithm. The performance of the decoding algorithm was first validated via simulations along with various fixed point quantization schemes. The detailed design of the decoding architecture was implemented on a field-programmable gate array (FPGA) kit with a short block length as an example. The schematics generated have been documented along with the synthesis results.

Acknowledgements

I would like express my gratitude towards my supervisor, Prof. Shayan G. Srinivasa, for his valuable advice, encouragement and guidance throughout the course of this project.

Contents

List of Figures

Chapter 1

1 Introduction

Error-correcting codes are used to automatically detect and correct errors in a received data signal. Codes with smaller error correction capabilities, such as the Bose-Chaudhuri-Hocquenghem (BCH) code [**1**], have been used previously for storage devices. However, as the device size scales down considerably, strong error correcting codes such as LDPC codes are needed. Soft decoding of LDPC codes yield better performance than hard decoding and is a state of the art technique in all commercial devices. Owing to the superior performance and suitability for hardware implementation, LDPC is a good choice for error correction in storage devices such as flash memories.

Low-density parity-check (LDPC) codes are a class of linear block codes. The name comes from the characteristic of their parity-check matrix which contains only a few 1's in comparison to the amount of 0's. They were first introduced by Gallager [**2**] in his PhD thesis in 1960, but were almost ignored until about 15 years ago due to computational and storage complexities for realizing the design architecture. With recent advances in Very Large Scale Integration (VLSI) technology, it has become more feasible to implement the LDPC decoder under practical constraints. LDPC codes have also been adopted by various standards such as DVB-S2, WiMAX (IEEE 802.16e) and 10G Ethernet (IEEE 802.3an).

Chapter 2

2 Theory of LDPC Codes

In this chapter, we will discuss the representation and the working of an LDPC code. We will highlight several variants of the decoding algorithm and present the min-sum algorithm with an example. This algorithm will be subsequently used for a design architecture towards this project.

2.1 Representation of LDPC Codes

There are two ways to represent LDPC codes. Like all linear block codes, they can be described via matrices. The second way is via a graphical description.

2.1.1 Matrix Representation

Let us consider the example of a low-density parity-check matrix,

$$H = \begin{bmatrix} 0\,1\,0\,1\,1\,0\,0\,1 \\ 1\,1\,1\,0\,0\,1\,0\,0 \\ 0\,0\,1\,0\,0\,1\,1\,1 \\ 1\,0\,0\,1\,1\,0\,1\,0 \end{bmatrix} \quad (1)$$

For regular LDPC codes (explained in section 3.2),

The matrix dimensions are n x m.

Let w_r be the number of ones in each row.

Let w_c be the number of ones in each column.

The matrix defined in equation (1) is a parity check matrix with dimension n x m for a *(8, 4)* code. For a matrix to be called low-density the two condition need to be satisfied, $w_c << n$ and $w_r << m$. In order to do this, the parity check matrix should usually be very large. So the example matrix in Eq. (1) can't really be called low-density.

2.1.2 Graphical Representation

Tanner introduced an effective graphical representation for LDPC codes. Not only do these graphs provide a complete representation of the code, but they also help to describe the decoding algorithm as explained later.

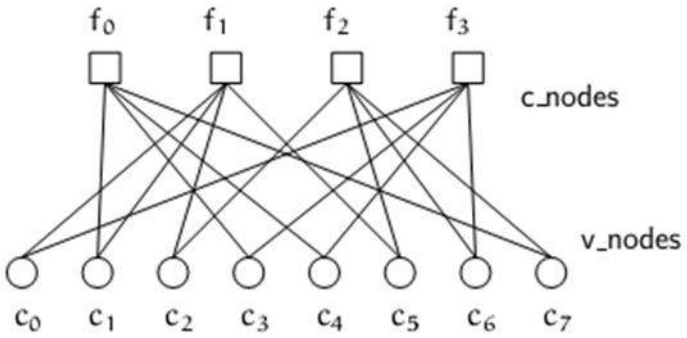


Figure 1 : Graphical representation of parity check matrix in Eq. 1.

Tanner graphs are bipartite graphs. That means that the nodes of the graph are separated into two distinctive sets and edges are only connecting nodes of two different types. The two types of nodes in a Tanner graph are called variable nodes (v-nodes) and check nodes (c-nodes). Fig. 1 is an example for such a Tanner graph and represents the same code as the matrix in H. The creation of such a graph is straight forward. It consists of m check nodes (the number of parity bits) and n variable nodes (the number of bits in a code word).Check node f_i is connected to variable node c_j if the element h_{ij} of H is a 1.

2.1.3 Regular and Irregular LDPC Codes

A LDPC code is called regular if w_c is constant for every column and $w_r = w_c \cdot (n/m)$ is also constant for every row. The example in Eq. 1 is regular with $w_c = 2$ and $w_r = 4$. We can also verify whether a code is regular or not from its graphical representation. If the number of incoming edges is same for all check nodes and also for all variable nodes, then it is a regular code. If H is low density but the number of *1's* in each row or column isn't constant, the code is called an irregular LDPC code. Degree constrained irregular LDPC codes give better decoding performance than regular ones, but the construction of the encoder as well as the decoder becomes much more complex.

2.1.4 Constructing LDPC Codes

There are several algorithms to construct suitable LDPC codes. Gallager [**2**] himself introduced one. Further, MacKay [**3**] proposed a way to semi-randomly generate sparse parity check matrices. Suitably chosen array codes also give good performance to the decoding algorithm. Constructing high performance LDPC codes is not a hard problem. In fact, completely randomly chosen codes are good with a high probability. The problem is that the encoding complexity of such codes is usually rather high.

2.2 Decoding Performance and Complexity

The feature of LDPC codes to perform near the Shannon limit [**1**] of a channel exists only for large block lengths. For example there have been simulations that perform within *0.04 dB* of the Shannon limit at a bit error rate of 10^{-6} with a block length of 10^7. An interesting fact is that those high performance codes are irregular.
The large block length results also in large parity-check and generator matrices. The complexity of multiplying a code word with a matrix depends on the amount of *1's* in the matrix. If we put the sparse matrix H in the form $[P^T\ I]$ via Gaussian elimination the generator matrix G can be calculated as $G = [I\ P]$. The sub-matrix P is generally not sparse so that the encoding complexity will be quite high.
Since the complexity grows in $O(n^2)$, even sparse matrices don't result in good performance if the block length gets very high. So iterative decoding (and encoding) algorithms are used. Those algorithms perform local calculations and pass those local results via messages. This step is typically repeated several times. The term "local calculations" indicates that a divide and conquer strategy, which separates a complex problem into manageable sub-problems. A sparse parity check matrix helps this algorithm in several ways. Firstly, it makes the local calculations simple and also reduces the complexity of combining the sub-problems by reducing the number of needed messages to exchange all the information. Secondly, it was observed that iterative decoding algorithms of sparse codes perform very close to the optimal maximum likelihood decoder.

2.3 Decoding LDPC Codes

The algorithm used to decode LDPC codes was discovered independently several times and comes under different names. The most common ones are the belief propagation algorithm, the message passing algorithm and the sum-product algorithm (SPA) [**4**].Only binary symmetric channels will be considered.

2.3.1 The Probability Domain Decoder

Soft-decision decoding of LDPC codes, which is based on the concept of belief propagation, yields a better decoding performance, and is therefore the preferred method. The underlying idea is exactly the same as in hard decision decoding. Before presenting the algorithm let us introduce some notations:

- $P_i = P_r(c_i = 1|y_i)$

- q_{ij} is a message sent by the variable node c_i to the check node f_j . Every message contains the pair $q_{ij}(0)$ and $q_{ij}(1)$ which stands for the amount of belief that y_i is a *'0'* or a *'1'*.

- r_{ji} is a message sent by the check node f_j to the variable node c_i . Again there is a $r_{ji}(0)$ and $r_{ji}(1)$ that indicates the (current) amount of belief that y_i is a *'0'* or a *'1'*.

The steps are given below:

1. All variable nodes send their q_{ij} messages. Since no other information is available at this step, $q_{ij}(1) = P_i$ and $q_{ij}(0) = 1 - P_i$.

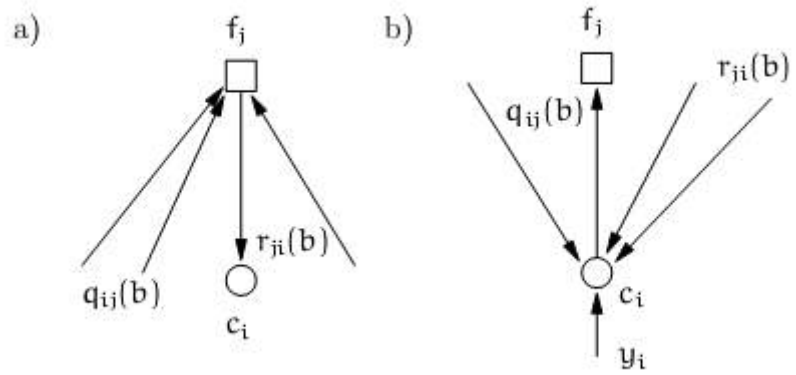


Figure 2: (a) Calculation of $r_{ji}(b)$ and (b) Calculation of $q_{ji}(b)$.

2. The check nodes calculate their response messages r_{ji} using Eq. (2) and (3):

$$r_{ji}(0) = \frac{1}{2} + \frac{1}{2}\prod_{i' \in V_j \backslash i}\left(1 - 2q_{i'j}(1)\right) . \quad (2)$$

$$r_{ji}(1) = 1 - r_{ji}(0) . \quad (3)$$

Eq. (2) indicates the probability that there is an even number of *1's* among the variable nodes except c_i, i.e., $r_{ji}(0)$ implies c_i is a *0*. This step and the information are used to calculate the messages are illustrated in Fig. 2.

3. The variable nodes update their response messages to the check nodes. This is done using Eq. (4) and (5).

$$q_{ij}(0) = K_{ij}(1 - P_i)\prod_{j' \in C_i \backslash j} r_{j'i}(0) \ . \quad (4)$$

$$q_{ij}(1) = K_{ij}P_i\prod_{j' \in C_i \backslash j} r_{j'i}(1) \ . \quad (5)$$

The constants K_{ij} are chosen in a way to ensure that $q_{ij}(0) + q_{ij}(1) = 1$. $C_i \backslash j$ means all check nodes except f_j are used in the computation. Fig. 2 illustrates the calculations in this step.

At this point, the v-nodes also update their current estimate $\hat{c}_i$ of the variable c_i. This is done by calculating the probabilities for *0* and *1* and voting for the most reliable quantity. This is done using Eq. (6) and (7).

$$Q_i(0) = K_i(1 - P_i)\prod_{j \in C_i} r_{ji}(0) . \quad (6)$$

$$Q_i(1) = K_i P_i \prod_{j' \in C_i} r_{ji}(1) . \quad (7)$$

Eq. (6) and (7) are quite similar to the ones to compute $q_{ij}(b)$, but the information from every c-node is used.

$$\hat{c}_i = \begin{cases} 1 & if\ Q_i(1) > Q_i(0) \\ 0 & else \end{cases} . \quad (8)$$

If the current estimated code word fulfils all the parity check equations, the algorithm terminates. Otherwise termination is ensured through a maximum number of iterations.

4. Go to step 2.

The soft decision decoding algorithm is a very simple variant and could be modified for performance improvements.

2.3.2 Problems in Probability Domain Decoding

- Multiplications are involved which are more hardware costly to implement than adders.
- Many multiplications of probabilities (sometimes *50-100*) are involved which could become numerically unstable. The results will come very close to zero for large block lengths.

To prevent this, it is possible to change into the log-domain and doing additions instead of multiplications. The result is a more numerically stable algorithm that has performance advantages since additions are less costly.

2.4 The Log Domain Decoder

2.4.1 Representation of Soft Information in Log Domain:

The information used in soft LDPC decoder represents bit reliability metric, log-likelihood-ratio (LLR) which is given by,

$$LLR(b_i) = \log\left(\frac{P(b_i=0)}{P(b_i=1)}\right), \quad (9)$$

where $P(b_i = 0)$ is the probability of the bit being *0* and $P(b_i = 1)$ is the probability of the bit being 1.

In the log domain, multiplications are converted to additions. For BPSK modulation scheme, i.e., where *0's* are transmitted as *+1's* and *1's* are transmitted as *-1's*,

The probability that the bit received is *1* is,

$$f_y(y|f = 1) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(y-1)^2}{2\sigma^2}} . \quad (10)$$

The probability that the bit received is *0* is,

$$f_y(y|f=-1) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(y+1)^2}{2\sigma^2}} . \quad (11)$$

Therefore, the log-likelihood ratio is

$$LLR = log \frac{f_y(y|f=1)}{f_y(y|f=-1)} = \frac{2y}{\sigma^2} . \quad (12)$$

2.4.2 Soft Information LLR Representation in a Graph

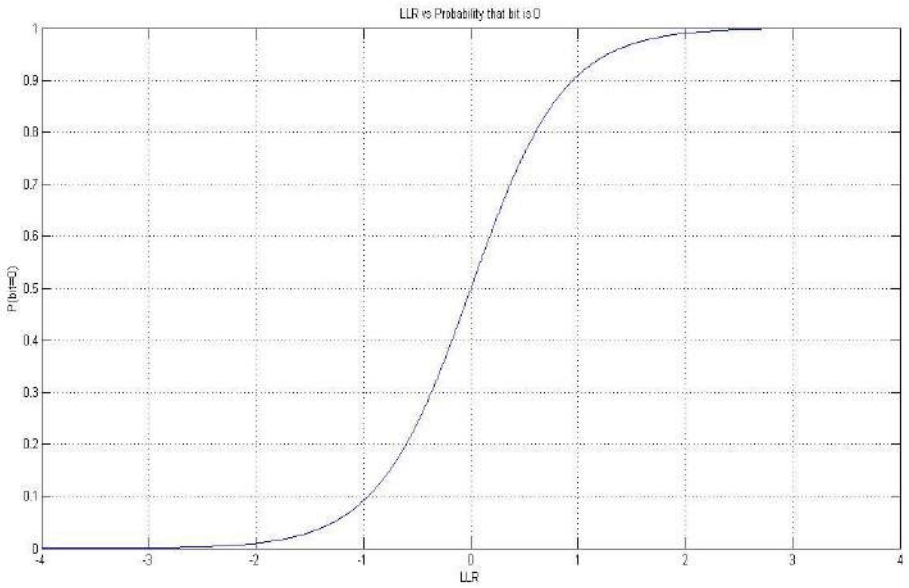


Figure 3 : Representation of LLR in a graph.

Message passing algorithms are iterative in nature. An LDPC decoding iteration consists of the following steps:

1) Upward pass - variable nodes pass their information to the check nodes.

2) Downward pass - check nodes send the updates back to variable nodes.

3) Syndrome check - checking whether $\boldsymbol{c}\, H^T = 0$ ($\boldsymbol{c}$ is the decoded codeword).

The process then repeats itself for several iterations until bits are corrected or maximum number of iterations is reached.

2.4.3 Sum Product Algorithm

The steps involved in the sum-product algorithm [**4**] are given below:

Step 1: Channel messages for each v-node via

$$m_0 = \frac{2y_i}{\sigma^2} \ . \tag{13}$$

Step 2: Update the variable node messages by

$$m^{(v)} = m_0 + \sum_{k=1}^{d_v-1} m_k^{(c)} \ . \tag{14}$$

Step 3: Update Check node messages by

$$m^{(c)} = \prod \alpha_k^{(v)} . \varphi\left(\sum_{k=1}^{d_c-1} \varphi\left(\beta_k^{(v)}\right)\right) \ . \tag{15}$$

Where $\alpha_k^{(v)}$ and $\beta_k^{(v)}$ are sign and magnitude of $m^{(v)}$

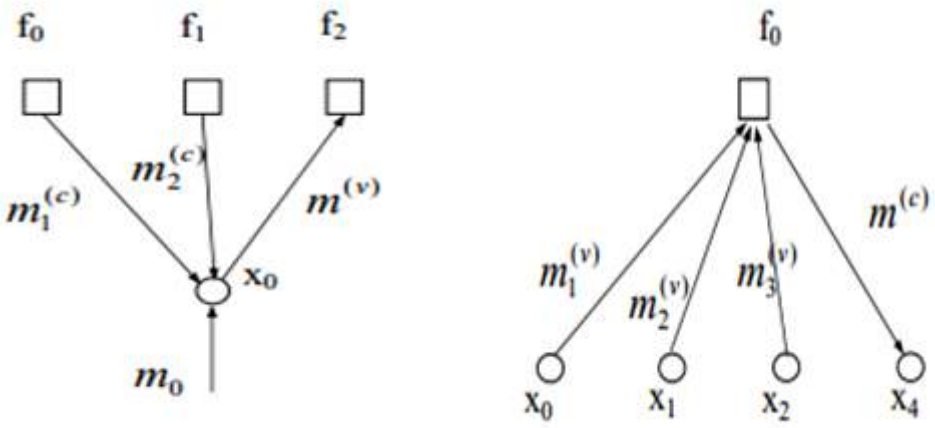


Figure 4: Check node update and variable node update.

Step 4: Compute $M^{(v)} = m_0 + \sum_{k=1}^{d_v-1} m_k^{(c)}$ and $\hat{x} = sign\left(M^{(v)}\right)$for each code bit to obtain $\hat{x}$ and hence $\hat{c}$.

Step 5: Stop if $\hat{c}H^T = 0$ or if maximum iterations is reached, otherwise go to step 2. The $\varphi(x)$ used in Eq. 15 is given by,

$$\varphi(x) = -\log\left(\tanh\left(\frac{x}{2}\right)\right) = log\frac{e^x+1}{e^x-1} \ . \tag{16}$$

2.4.4 Min-sum Algorithm as an Approximation of Sum Product Algorithm (SPA)

For understanding the min-sum algorithm [**5**], let us consider an update equation in the SPA decoder

$$L(r_{ji}) = (\prod_{i'} \alpha_{ij}).\varphi(\sum_{i'} \varphi(\beta_{i'j})) \,. \tag{17}$$

The shape of φ(x) is shown in Fig. 5.

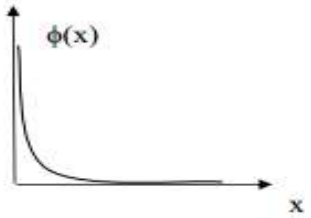


Figure 5 : Graphical Representation of φ(x).

Since the term corresponding to smallest β_{ij} dominates, we have

$$\varphi(\sum_{i'} \varphi(\beta_{ij})) = \varphi(\varphi(\min(\beta_{i'j}) = \min(\beta_{i'j}) \,. \tag{18}$$

The min-sum algorithm is thus simply the log-domain algorithm with step 3 replaced by

$$L(r_{ji}) = (\prod_{i'} \alpha_{ij}).\min(\beta_{i'j}) \,. \tag{19}$$

2.5 Min-sum Algorithm and its Variants for Hardware Friendly Design

2.5.1 Min-sum Algorithm

Steps:
Initial Variable to Check Node Pass: At beginning of the decoding process, variable nodes receive the LLR's. Variable nodes pass these values to check nodes.

1st Iteration:

i) **Check to Variable Pass** : For a certain n^{th} variable node it is connected to, a check node finds minimum of absolute values of other nodes except n^{th} node, and sends that value with a sign, so that modulo *2* sum is satisfied.

ii) **Variable to Check Pass**: A variable node sums up all the information it has received at the end of last iteration, except the message that came from m^{th} check node, and sends it to m^{th} check node.

Subsequent Iterations: The above process is repeated until the bits are corrected, or the stopping criterion is reached.

2.5.2 An Example

Let the H matrix used be $H = \begin{bmatrix} 1&1&1&0&0&0&0&0 \\ 0&0&0&1&1&1&0&0 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \end{bmatrix}$. (20)

Let us consider a frame [1 0 1 0 1 1 1 1] sent over the binary AWGN channel with bi-polar mapping. Let the frame sent be [-1 1 -1 1 -1 -1 -1 -1]. The channel adds AWGN noise with a variance of *0.5* so that the received frame is [-0.8 0.7 -0.9 0.7 0.5 -1.5 -2.4 -1.2]. The LLR's of the received frame are [-3.2 2.8 -3.6 2.8 2 -6 -9.6 -4.8]. The fifth value has an error. The message passing in the Tanner graph, during the decoding process is illustrated by Fig. 6.

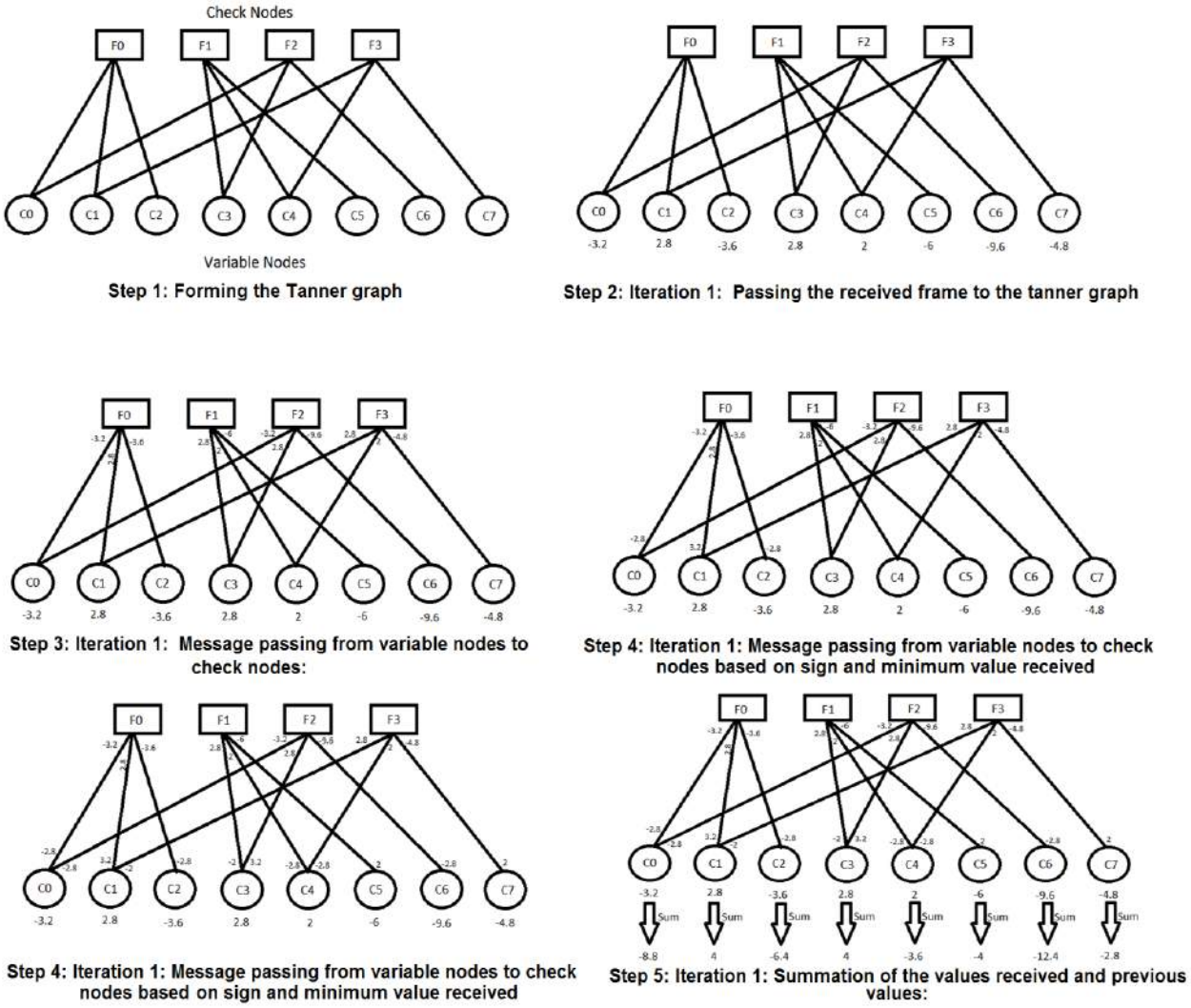


Figure 6: An example of min-sum algorithm explained with Tanner graph where the error correction occurs in the first iteration.

2.5.3 Another Example

Let the H-matrix be used is $$H = \begin{bmatrix} 1\,1\,0\,1\,0\,0\,0 \\ 0\,0\,1\,1\,1\,0\,0 \\ 0\,0\,0\,1\,0\,1\,1 \end{bmatrix}. \quad (21)$$

Let us consider a frame *[1 0 1 1 0 0 1]* being sent over the channel using bi-polar mapping. Let the frame sent is *[-1 1 -1 -1 1 1 -1]*. Suppose the channel adds AWGN noise with a variance of *0.5* so that LLR's of received frame is *[-8 -6 -11 -5 8 9 -12]*. The second received sample has an error. The decoding stages on the Tanner graph are illustrated by Fig. 7.

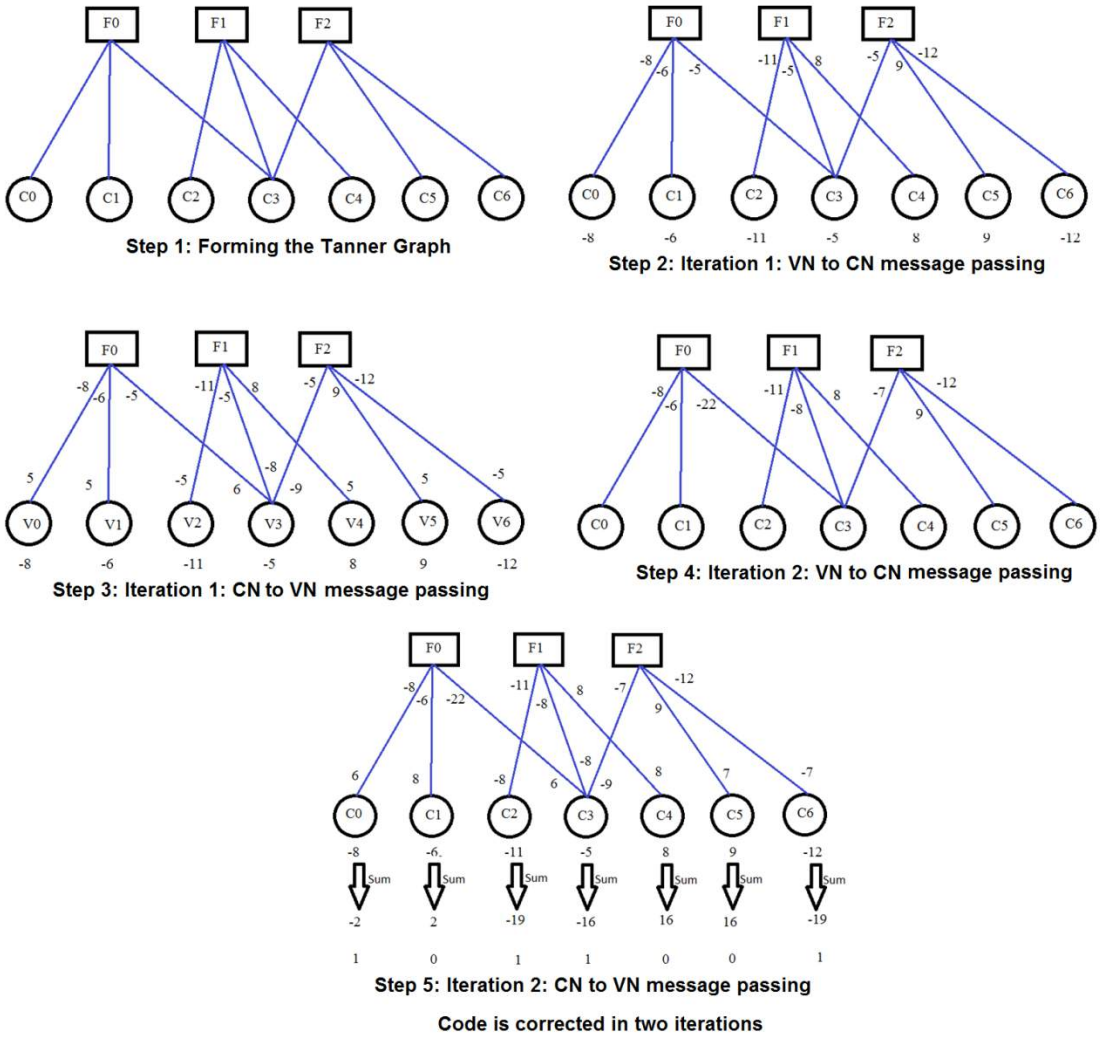


Figure 7: Another example of min-sum algorithm explained with Tanner graph where the error correction occurs in the second iteration.

2.5.4 Advantages of Min-sum Algorithm

1) The min-sum algorithm approximates the sum-product algorithm with simple check node update operations, that significantly reduce the VLSI implementation complexity .

2) Complex exponential or logarithmic calculations are avoided, leading to reduced hardware complexity.

2.5.5 Disadvantages of Min-sum Algorithm

The numbers of iterations required are often more than the SPA.

Chapter 3

3 Matlab Simulation Results

In this chapter we will show all the software simulation results for several variants of the algorithm for hardware friendly design. We will highlight various issues faced during experiments. Simulation results are performed to assess the SNR (Signal to Noise ratio) gain for floating point and with fixed point quantization. Using simulation results, we distill a fixed point design architecture.

3.1 Variants of the Min-sum Algorithm [6]

Normalized Min-sum Algorithm: In this variant of min-sum algorithm, a constant multiplicative factor is applied to the subsequent check node updates. For short block lengths, the normalized min-sum algorithm does not give a significant SNR improvement. However, when larger block lengths are used, significant SNR improvement is observed [**7**].

Layered Min-sum Algorithm: For hardware implementation requiring less area, layered min-sum LDPC might be a good choice. In this algorithm, the only difference is that the block rows are processed serially and after each block row operation, update takes place [**8**] [**9**].

3.2 Matlab Simulation and Verification of Min-sum Algorithm and its Variants

3.2.1 Theoretical Calculation of Raw Frame Error Rate (FER)

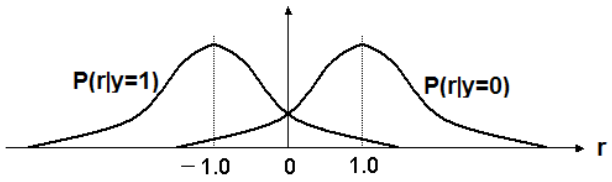


Figure 8: Graphical representation of BPSK data through AWGN channel.

Theoretical Results:

We know that the signal to noise ratio in *dB*, for a certain rate *R* and a variance σ^2, is given by,

$$SNR_{db} = 10log_{10}\frac{1}{R\sigma^2} \tag{22}$$

It can be shown that probability of error for such channel is

$$P_e = 2 * 0.5 * \int_0^{\propto} \frac{1}{2\pi\sigma^2} e^{\frac{-(x+1)^2}{2\sigma^2}} dx \tag{23}$$

Equation (23) can be written in form of complementary error function as

$$P_e = 0.5 * \text{erfc}\left(\frac{1}{\sqrt{2\sigma^2}}\right) \tag{24}$$

Frame error rate can be calculated from the BER using

$$FER = 1 - (1 - P_e)^{259} \tag{25}$$

Plotting the above FER will give results for *0* iterations and the plot is shown in Fig. 9.

Theoretical Plot with no Error Correction:

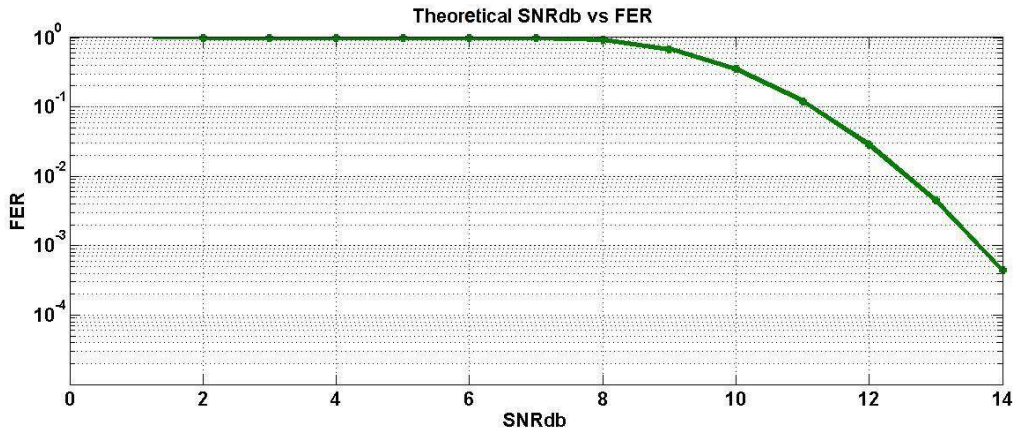


Figure 9: Theoretical SNR (dB) vs. FER without any error correction.

Choice of a Suitable H-matrix:

An array low-density parity-check [**10**] matrix for a regular quasi-cyclic LDPC code is specified by three parameters: a number *p* and two integers *k* (check-node degree) and *j* (variable-node degree) such that $j,k \leq p$. This is given by,

$$\begin{bmatrix} \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{I} & & \boldsymbol{I} \\ \boldsymbol{I} & \boldsymbol{\alpha} & \boldsymbol{\alpha^2} & \cdots & \boldsymbol{\alpha^{k-1}} \\ \boldsymbol{I} & \boldsymbol{\alpha^2} & \boldsymbol{\alpha^4} & & \boldsymbol{\alpha^{2(k-1)}} \\ & \vdots & & \ddots & \vdots \\ \boldsymbol{I} & \boldsymbol{\alpha^{j-1}} & \boldsymbol{\alpha^{2(j-1)}} & \cdots & \boldsymbol{\alpha^{(j-1)(k-1)}} \end{bmatrix} \tag{26}$$

where *I* is a *pxp* identity matrix, and *a* is a *pxp* permutation matrix representing a single right cyclic shift (or equivalently up cyclic shift) of *I*. The exponent of α in *H* is called the shift

coefficient and denotes multiple cyclic shifts, with the number of shifts given by the value of the exponent.

It has been seen that array codes give fairly reasonable performance to iterative decoding algorithms like LDPC algorithm. However, the design of good codes is a challenging task often handled via graph theoretic methods.

Simulation Statistics:

Array Code of size *259x148* was used as *H*-matrix

$$H = \begin{bmatrix} I & I & I & I & I & I & I \\ I & p & p^2 & p^3 & p^4 & p^5 & p^6 \\ I & p^2 & p^4 & p^6 & p^8 & p^{10} & p^{12} \\ I & p^3 & p^6 & p^9 & p^{12} & p^{15} & p^{18} \end{bmatrix} \quad (27)$$

I is an identity matrix of size *37x37*. *p* was constructed by shifting it left *1* column. So the rate of the code is *3/7*. We send all zeros in BPSK modulation, i.e., we send a frame containing *259 1's*. Additive white Gaussian noise (AWGN) is added to it. LLR's of received values is fed to the decoding algorithm, and it is checked whether it can decode it. Frame error rate is plotted against E_b/N_0 in dB for 0 iterations, 1 iteration, 5 iterations and *10 iterations* respectively.

3.2.2 Simulation with Floating Point Numbers

Frame error rate was plotted against E_b/N_0 in *db* for *0* iterations, *1* iteration, *5* iterations and *10* iterations respectively. *0* iterations implies that the code is not subjected to error correction and raw frame error rate is calculated. It is a relative measure for performance improvement. The curves so produced are called waterfall curves because of the gradual increase in steepness. Simulations were carried out in Matlab and the results are shown in Fig. 10.

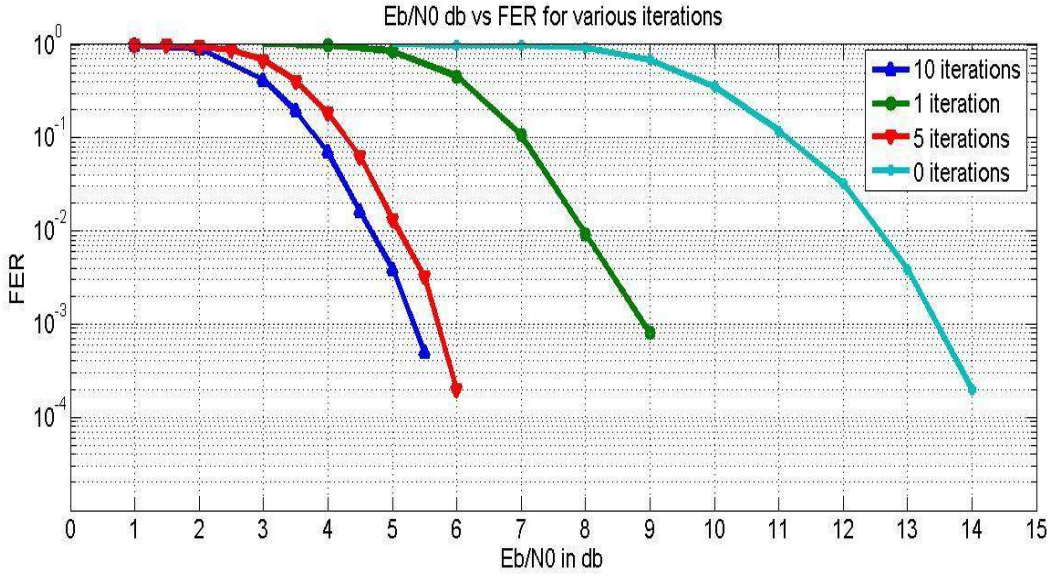


Figure 10: E_b/N_0 vs FER for floating point numbers for 0, 1, 5, 10 iterations respectively.

Inferences:

1) We see that with *1* iteration we get *4.5 dB* performance improvement.
2) With *5* iterations we get almost *3 dB* performance improvement over *1* iteration.
3) With *10* iterations there is only slight improvement of *0.3 dB* from *5* iterations.
4) A max iteration parameter must be optimally chosen to balance latency and SNR gain.

3.2.3 Simulation with Normalized Floating Point Numbers

In this variant of min-sum algorithm, after each iteration, a constant multiplicative factor is applied to the next check node updates. With short block length, this algorithm does not give significant improvement, but with large block lengths, improvements are observed. We have used multiplicative factors of *0.5, 0.6, 0.7 and 0.8* in our experiments. Simulations were carried out in Matlab and the results are shown in Fig. 11.

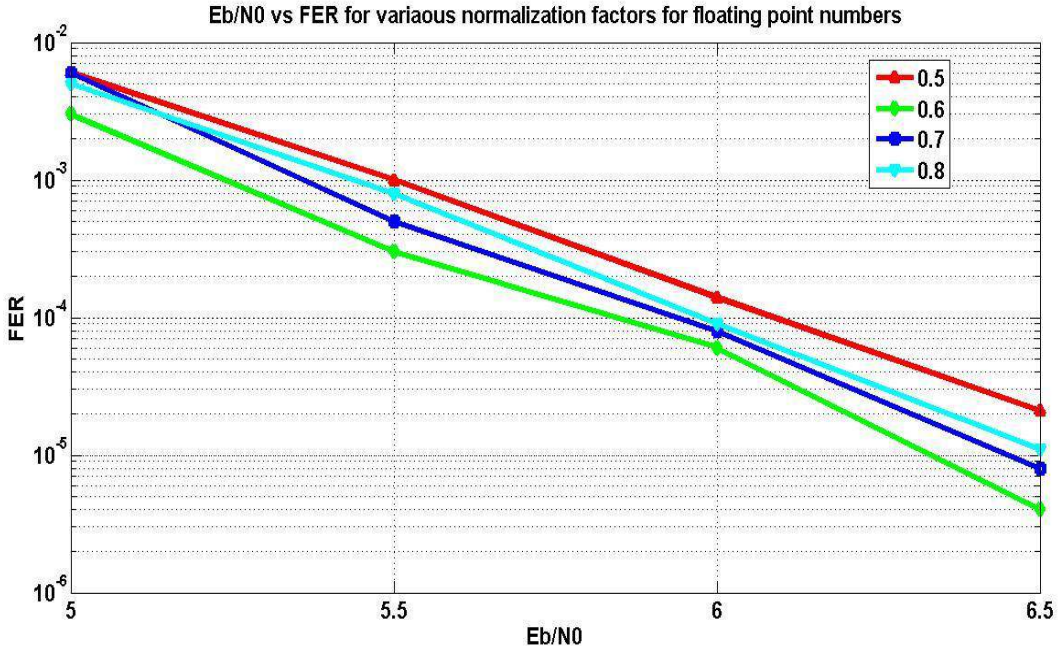


Figure 11: E_b/N_0 vs FER for normalized floating point numbers with normalizing factors 0.5, 0.6, 0.7, 0.8 for 5 iterations.

Inferences:

1) Performance improvements are seen as we decrease the multiplicative factor from *0.8* to *0.6*. But, with a multiplication factor of *0.5*, performance becomes worse than the other *3* cases. So, *0.6* seems to be the optimum value of the normalization factor for the present code.

3.2.4 Simulation with Quantized Floating Point Numbers

In this simulation, the floating point numbers were quantized to *5* bits, *6* bits and *7* bits respectively. Out of the total bit budget, *2* bits were allocated for the integer part, one for the sign and rest for the decimal part. Frame error rate was plotted against E_b/N_0 in *dB* for *5* iterations. Due to quantization error, the curves shift right and performance becomes poorer for a certain SNR. Also, probability of zero propagation increases. Simulations were carried out in Matlab and the results are shown in Fig. 12.

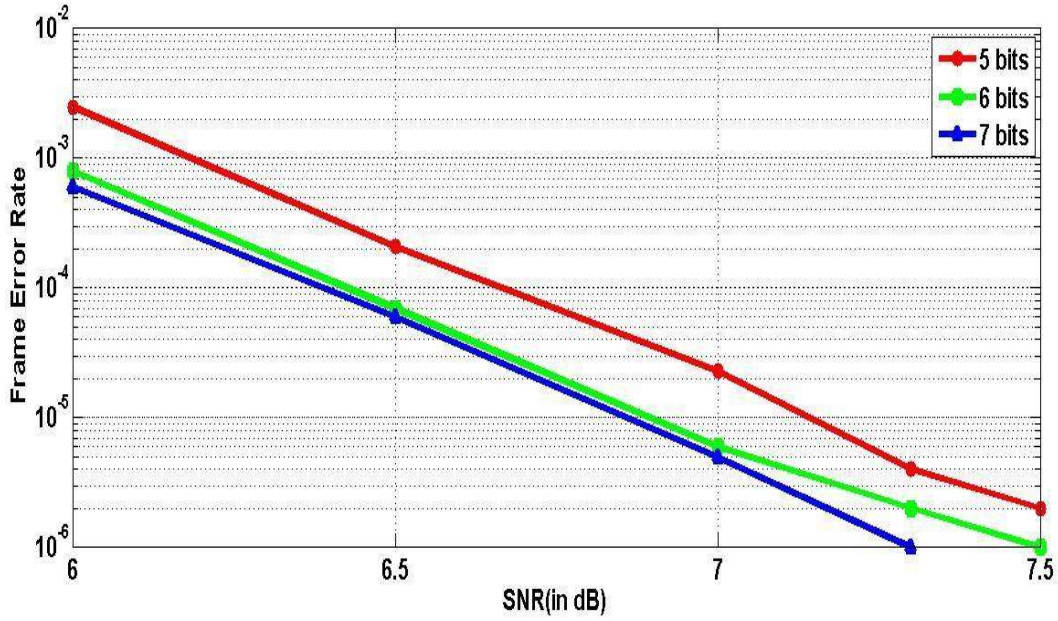


Figure 12: E_b/N_0 vs FER for quantized numbers, with 5, 6, 7 bits for 5 iterations.

Inferences:

1) *7* bit precision gives the highest performance.
2) There is very small observable difference between the *6* bit and *7* bit precision.
3) There is observable error floor at high SNR's due to quantization. This happens because of lower frame lengths and won't occur for large frame lengths.

3.2.5 Simulation with Quantized Values using Layered Decoding Algorithm.

In this simulation, the floating point numbers were quantized to *5* bits, *6* bits and *7* bits respectively. Layered decoding algorithm was applied. Out of the total *2* bits are for integer part, one for sign and rest for decimal part. Frame error rate is plotted against E_b/N_0 in *db* for *5* iterations. Simulations were carried out in Matlab and the results are shown in Fig. 13.

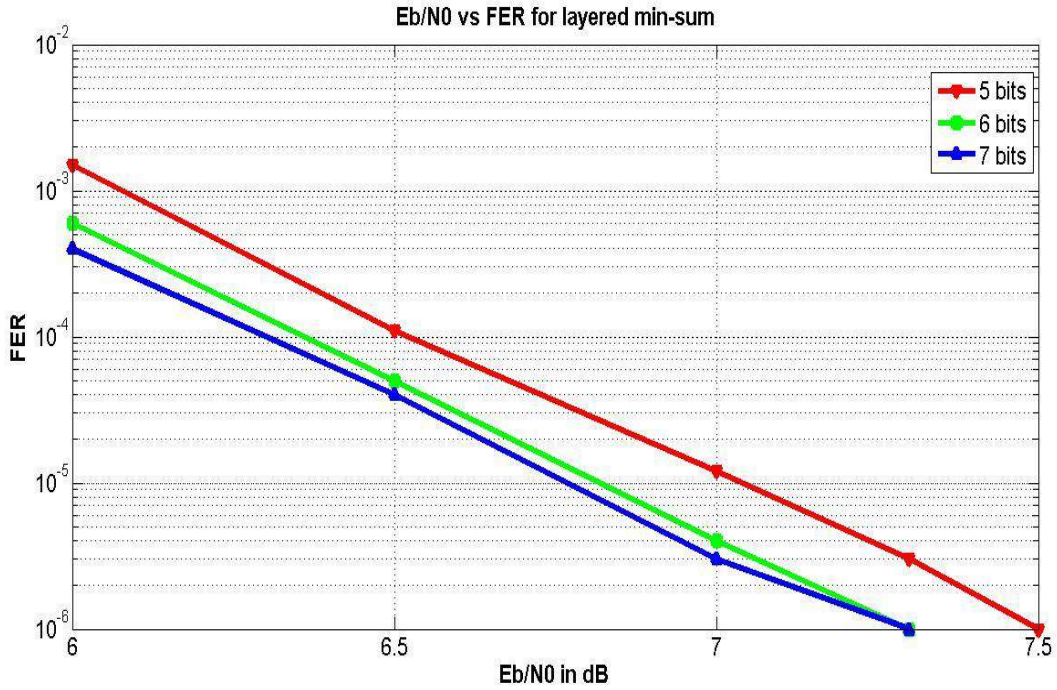


Figure 13: E_b/N0 vs FER for layered min-sum, with quantized numbers, with 5,6,7 bits for 5 iterations.

Inferences:

There is a little improvement from the quantized min-sum. This is because the block length is small. For larger block lengths, we will achieve better results. There is an observable error floor at high SNR's due to quantization.

3.3 Issues faced during Extreme Quantization

Propagation of 0: As shown in Fig. 14, message sent to the check node is *0* as sum of all other received messages is *0*. This *0* propagates in subsequent iterations leading to erroneous results.

Solution: A small number ε is sent instead of *0*, and its sign is determined by majority of sign of the other messages received. Here, we experiment with *-0.25*.

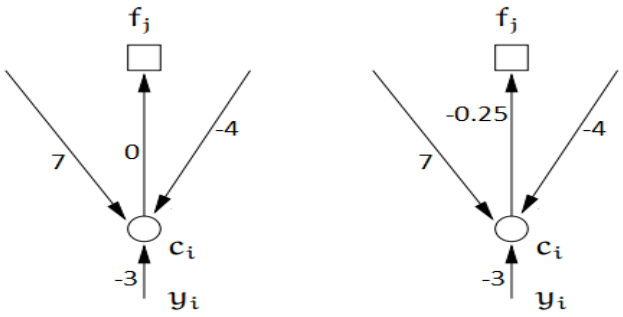


Figure 14: Propagation of 0 problem.

Mis-correction: For an error correcting code, when the received sequence falls into the decoding area of an erroneous code word, the decoder may deliver the wrong code word as the decoding result since the syndrome is satisfied. This may happen when a significant number of bits are in error.

Quantization Error: For hardware design, the LLRs need to be quantized, leading quantization errors. After doing summations of these values, they need to be quantized again. In this way, quantization error is a significant problem while doing hardware design.

Chapter 4

4 Design Architecture of the Decoder

In this chapter, we will discuss the design of the decoder on hardware. We will start from the top level view of the decoder and go one level deeper into the basic blocks. The decoder mainly consists of a check node unit, a variable node unit, and a barrel shifter to produce relative shifts in various block rows. There are two basic design strategies, the layered and the non-layered one. The non-layered one aims at higher throughput, with considerably higher area, while the layered one has lower throughput, with lower area requirement. *5* bit precision will be used throughout the architecture. But, owing to the additions taking place in VNU's and resulting increase in size of data bits, we will have to pad two *0's* to every input. For all further discussions, we will assume that two *0's* have already been padded to the input LLR's after the sign bit. Thus, the data length of *7* bits throughout the architecture.

In this project, we are building a non-layered architecture for higher speed. But, the layered architecture has also been investigated into. All the blocks have been explained in detailed along with necessary pipelining issues.

4.1 Non Layered Architecture

Basic Architecture of the Decoder: Fig. 15 shows a system comprising a low density parity check (LDPC) decoder. It generally includes a transmitter and a receiver. The receiver comprises of an I/O port, a processor, a memory and an LDPC decoder. The transmitter transmits signal encoded using LDPC to provide error correction. The transmitter may be wireless or wire-lined. I/O port detects the signal from the transmitter and can have necessary protocols for receiving the data. This signal is then provided to the LDPC decoder. The LDPC decoder detects and tries to correct the errors introduced in the signal. The processor controls the operations of the I/O port. Memory may be comprised of any suitable type of storage elements, like DRAM, SRAM or flash memory.

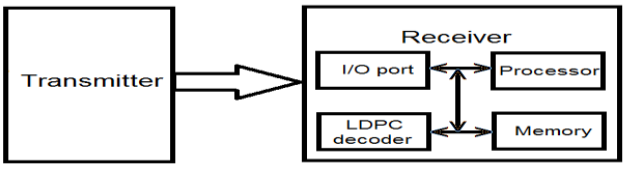


Figure 15: Basic Top Level Architecture.

Code Representation: An array low-density parity-check matrix for a regular quasi-cyclic LDPC code is specified by three parameters: an integer p and two integers k (check-node degree) and j (variable-node degree) such that $j,k \leq p$. This is given by eq. 28, where I is a p x p identity matrix, and a is a p x p permutation matrix representing a single right cyclic shift (or equivalently up cyclic shift) of I. The exponent of α in H is called the shift coefficient and

denotes multiple cyclic shifts with the number of shifts given by the value of the exponent. It has been seen that array codes give reasonable performance under the message passing algorithm.

$$\begin{bmatrix} I & I & I & & I \\ I & \alpha & \alpha^2 & \cdots & \alpha^{k-1} \\ I & \alpha^2 & \alpha^4 & & \alpha^{2(k-1)} \\ & \vdots & & \ddots & \vdots \\ I & \alpha^{j-1} & \alpha^{2(j-1)} & \cdots & \alpha^{(j-1)(k-1)} \end{bmatrix} \quad \textbf{(28)}$$

4.1.1 Check Node Unit

The check node unit in Fig. 16 emulates the computations that are done at a check node [**11**]. It sends the minimum of the values received from the variable discounting a certain variable node. The check node unit consists of four main parts, a minimum value N_1 and a second minimum value N_2 finder, a partial state which holds N_1 and N_2 temporarily and updates them on each clock cycle, a final state which contains the final N_1 and N_2 and a sign processing unit which finds out the sign of the value to be sent. Let us consider a *(3, 7)* code. In the first *7* clock cycles, incoming variable messages are compared to two up-to-date least minimum numbers, to generate new partial state. In this state, we have N_1 (first minimum value), N_2 (second minimum value) and the index of N_1. The final state is then calculated by normalizing the partial state by a normalizing factor. So, the final state consists of normalized values of $+N_1$, $-N_1$ and $+/-N_2$. *R* selector then assigns one of these *3* values based on the index of N_1 and sign of *R* messages generated by xor logic. We will do the CNU operations in signed magnitude arithmetic.

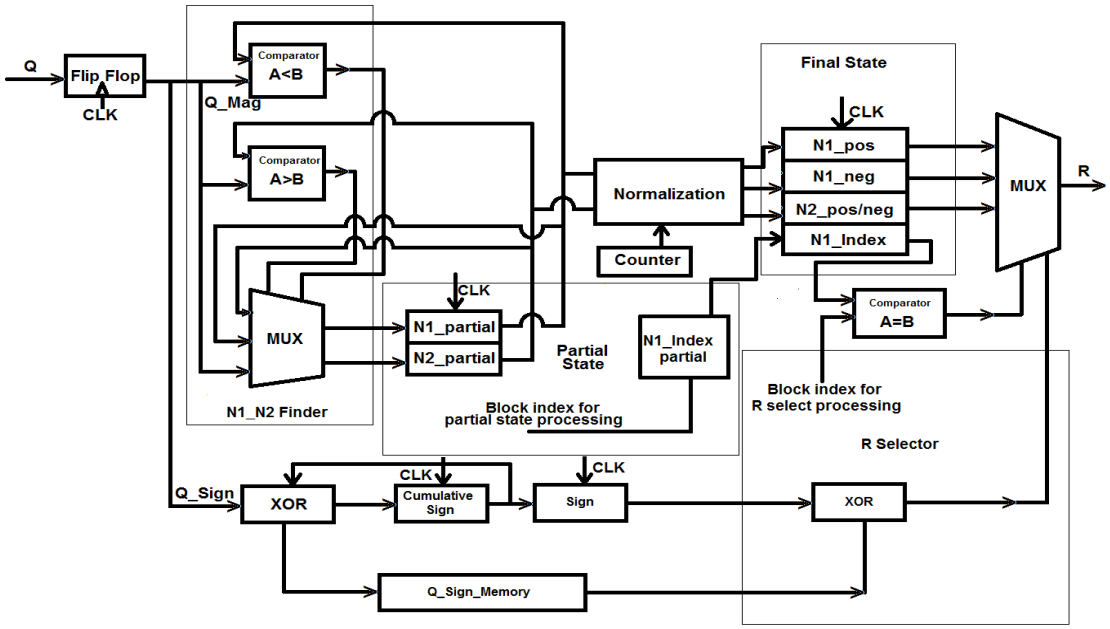


Figure 16: The check node unit consists of four main parts, a minimum value $N_1$and a second minimum value N_2 finder, a partial state which holds N_1 and N_2 temporarily and updates them on each clock cycle, and a final state which contains the final N_1 and N_2 and a sign processing unit which finds out the sign of the value to be sent. After a certain number of clock pulses, the final state will contain the actual values of N_1 and N_2.

4.1.2 Variable Node Unit

The variable node unit in Fig. 17 emulates the computations that are done at a variable node of a Tanner graph. Variable node unit has an adder block which adds all the log-likelihood ratio (LLR) values it receives and a subtractor which subtracts the value it receives from a check node from the partial sum, and sends it back to the check node. Hard decision is also computed from the sign of the sum.

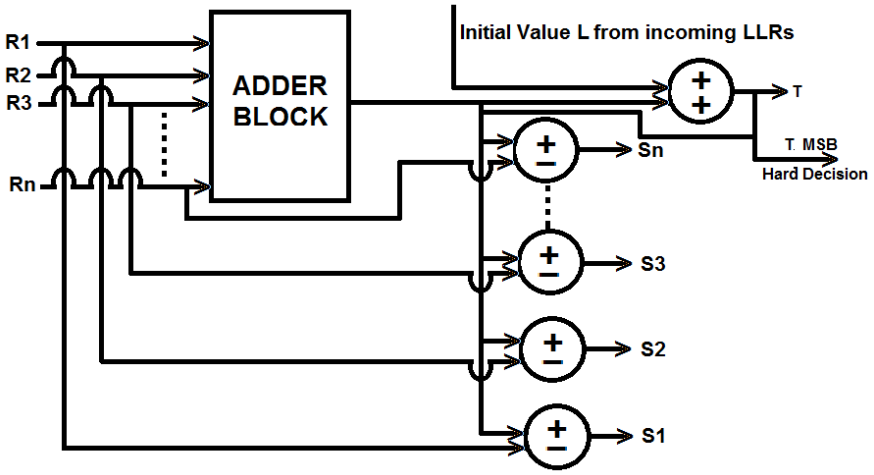


Figure 17: Variable node unit has an adder block which adds all the log-likelihood ratio (LLR) values it receives and a subtractor which subtracts the value it receives from a check node from the partial sum, and sends it back to the check node. Hard decision is also computed from the sign of the sum.

4.1.3 Architecture of the Barrel Shifter

Fig. 18 shows the architecture of barrel shifter as used in the design of the decoder. It contains *6* stages to produce shifts of *1, 2, 4, 8, 16, 32.* So, we can produce a maximum shift of 63. In this example we have a parallelism of *37* code. So barrel shifter should be able to produce at least a shift of 37. So we are using 6 stages. Each stage contains *37* numbers of *14:7* MUXes in parallel. The valued of shift to be produced is determined by a signal *shifter_b* consisting of 6 bits, because there are 6 stages. LSB of *shifter_b* is connected to the first stage. Next *LSB* is connected to 2^{nd} stage and in similar fashion, the MSB is connected to last stage. Each stage *n*(where *n=1,2,3,4,5,6*) produces a shift of 2^n or *0*, based on the value of the bit *shift_b(n-1)* received. Thus, it can produce any shift from *0* to *63*. But, we require a max shift of *37*. The propagation delay of the barrel shifter becomes a significant factor for large parallelism because, with the increase in the number of stages, the total delay of the barrel shifter increases and after some point it may not be possible to accommodate it within *1* clock cycle. Also, the area of the barrel shifter is a source of concern for large parallelism. One way to avoid large area may be to do things serially and produce *1* shift in *1* clock cycle. But, that will largely diminish the throughput of the decoder.

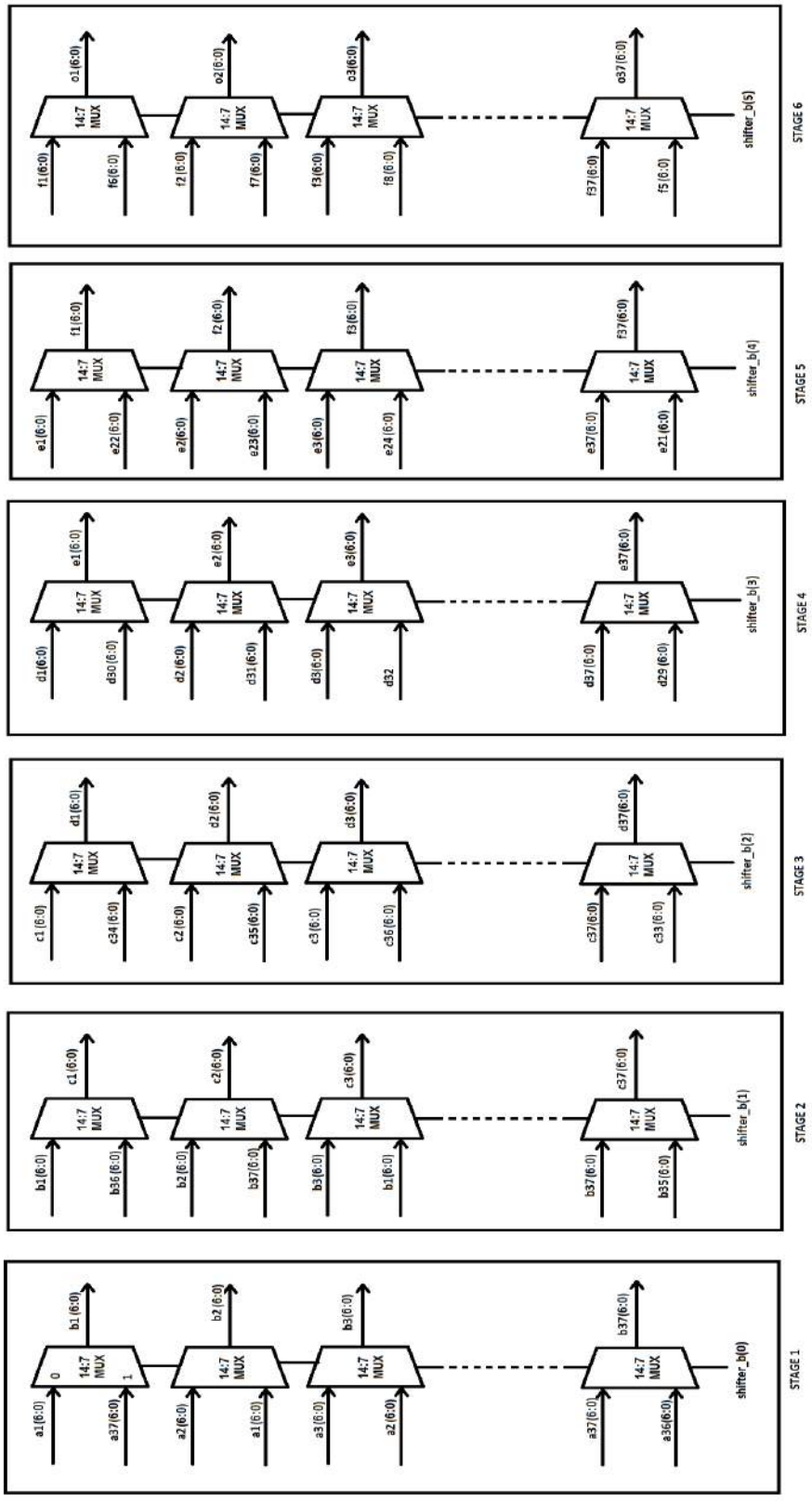


Figure 18 : Barrel shifter architecture: It has 6 stages which provide shifts of 1, 2, 4, 8, 16, 32 respectively. Shifter_b determines the amount of shift to be produced.

4.1.4 Non-layered Decoder Architecture at the Top Level

As said in the beginning of the chapter, non-layered decoding gives highest throughput. So, to build a high speed decoder, we are investigating into the non-layered architecture. Fig. 19 shows a block diagram of the non-layered decoder architecture, for a column weight 4 code, where messages are passed between CNUs and VNUs iteratively.

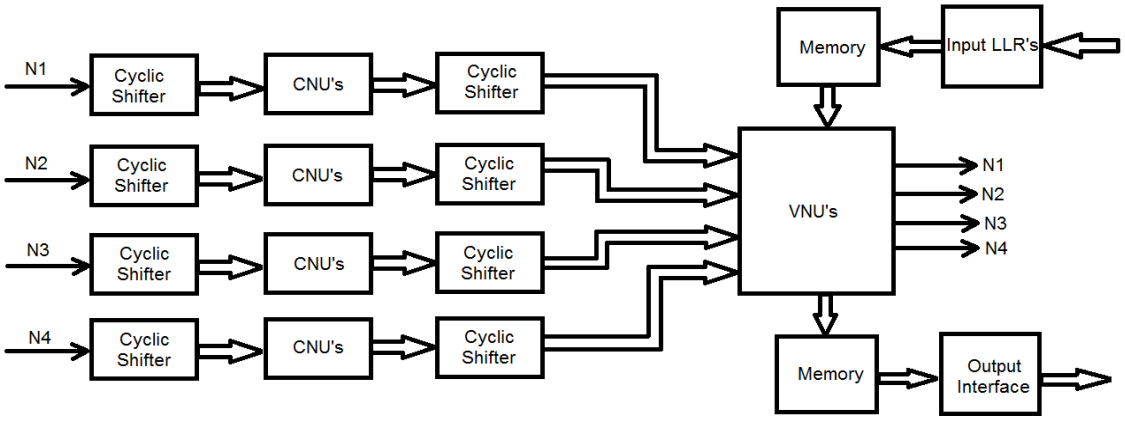


Figure 19: Non layered decoder architecture where the message is passed between CNUs and VNUs iteratively for a column weight 4 code.

Cyclic shifters generally consume approximately *10-20%* of the chip area based on the decoder's parallelization and constitute the critical path of the decoder. Using the properties of array codes, we can use constant wiring to achieve any cyclic shifts as each subsequent shift can be realized using feedback of a previous shifted value. By this, forward router between CNU and VNU and reverse router between VNU and CNU can be eliminated. This is because array codes have constant incremental shift in each block row. For the first block the shift and incremental shift is *0*. For the second block row, the shifts are [*0, 1, 2*] and incremental shift is *1*. For the third block row, the shifts are [*0, 2, 4*] and incremental shift is *2*, and it goes on in similar fashion, for other block rows. The technique of achieving various shifts using constant wiring, instead of cyclic shifters, was claimed in US Patent 8359522 [**12**]. The connection between CNU block arrays and VNUs is illustrated in Fig. 20.

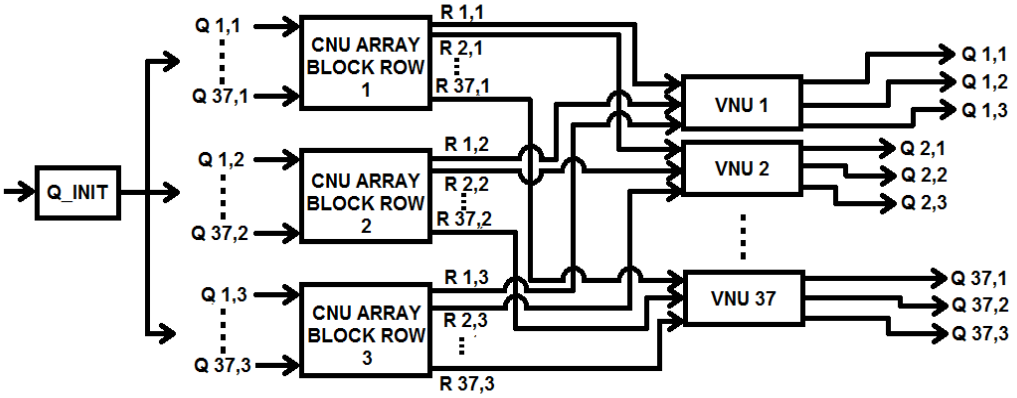


Figure 20: CNU block arrays and VNU connection for non-layered architecture without the use of cyclic-shifters.

The architecture for CNU block array 1 is shown in Fig. 21.

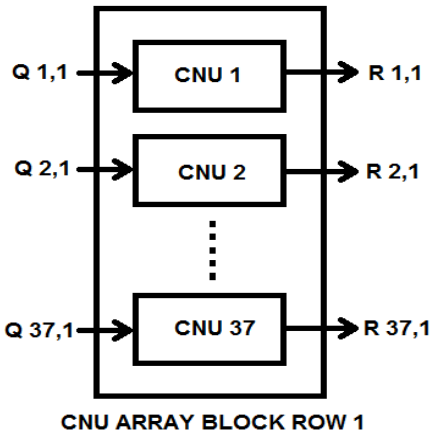


Figure 21: Block Diagram of the CNU block array 1.

CNU array block rows *2* and *3* are a little different from CNU array block row *1*, as it has to accommodate for the shifts that are not present in block row *1*. They are composed of dynamic CNUs [**12**] as shown in Fig. 23. In the dynamic CNUs, the minimum and second minimum finder of a CNU receives input from partial state of another CNU rather than its own partial state block. The final state block receives input from partial state and final state of another CNU.

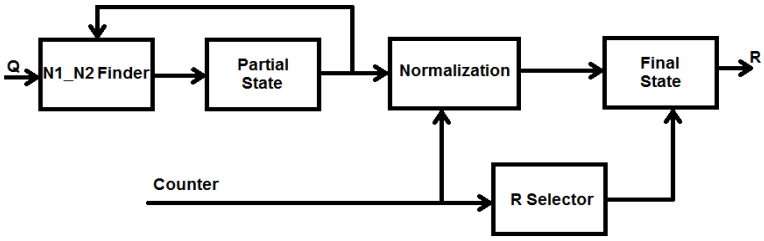


Figure 22: Block diagram of normal CNU for CNU block array 1.

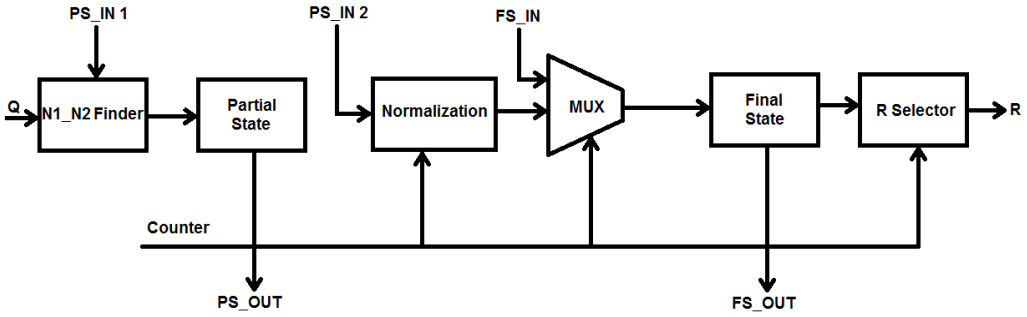


Figure 23: Block diagram of dynamic CNU for block row 2 and 3.

In the CNU array for the second block row of the *H*-matrix, *CNU74* gets its partial state from *CNU73*, *CNU73* gets its partial state from *CNU72* and so on. The schematic for the CNU array block row *2* is given in Fig. 24 [**12**].

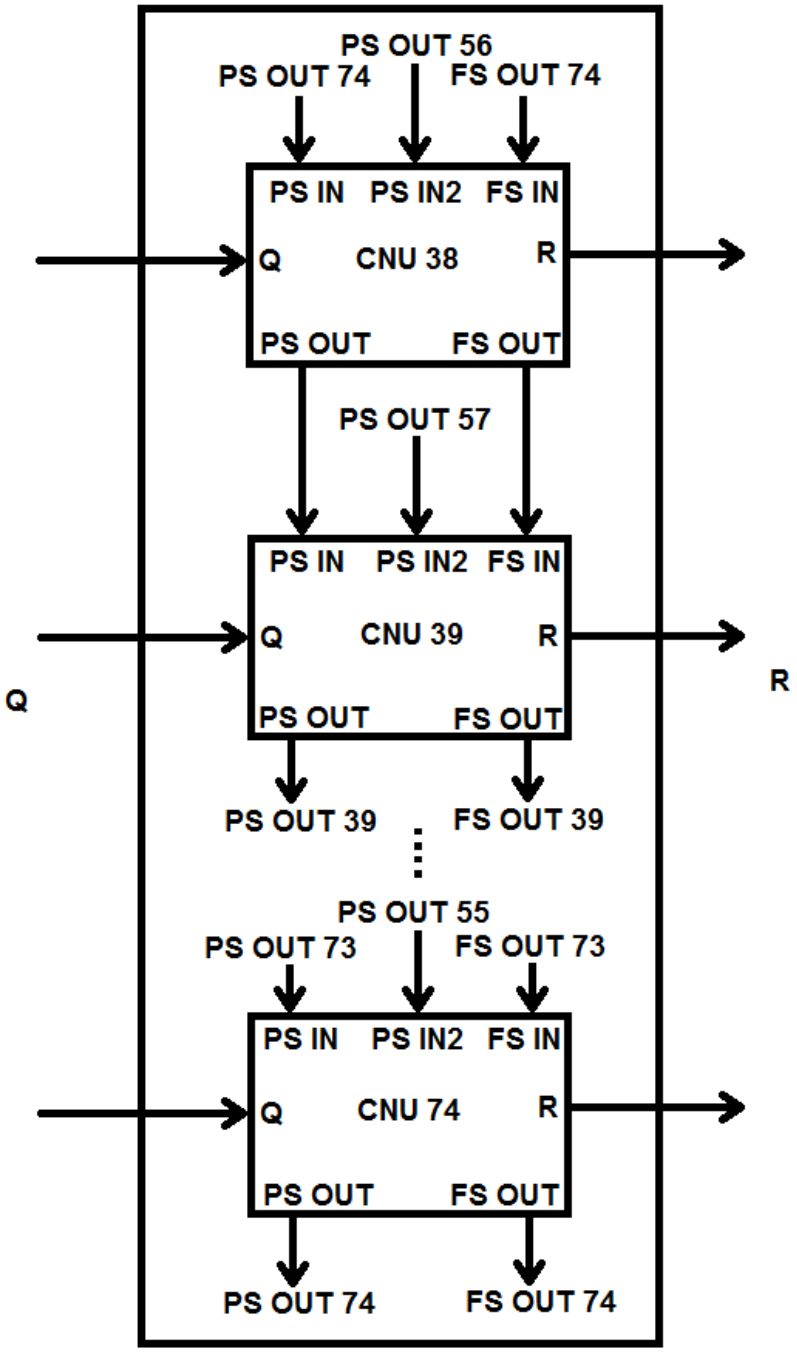


Figure 24: Block diagram of CNU block array 2 showing the connections made to achieve the required shifts [12**].**

In the CNU array for the third block row of the H-matrix, the connections are made in such a way that the connections between the partial state registers achieve cyclic shifts of [*0,2,4….*]. A similar principle is applied when making connections when making connections for the final state in the CNU array to achieve reverse routing. This is shown in Fig. 25 [**12**].

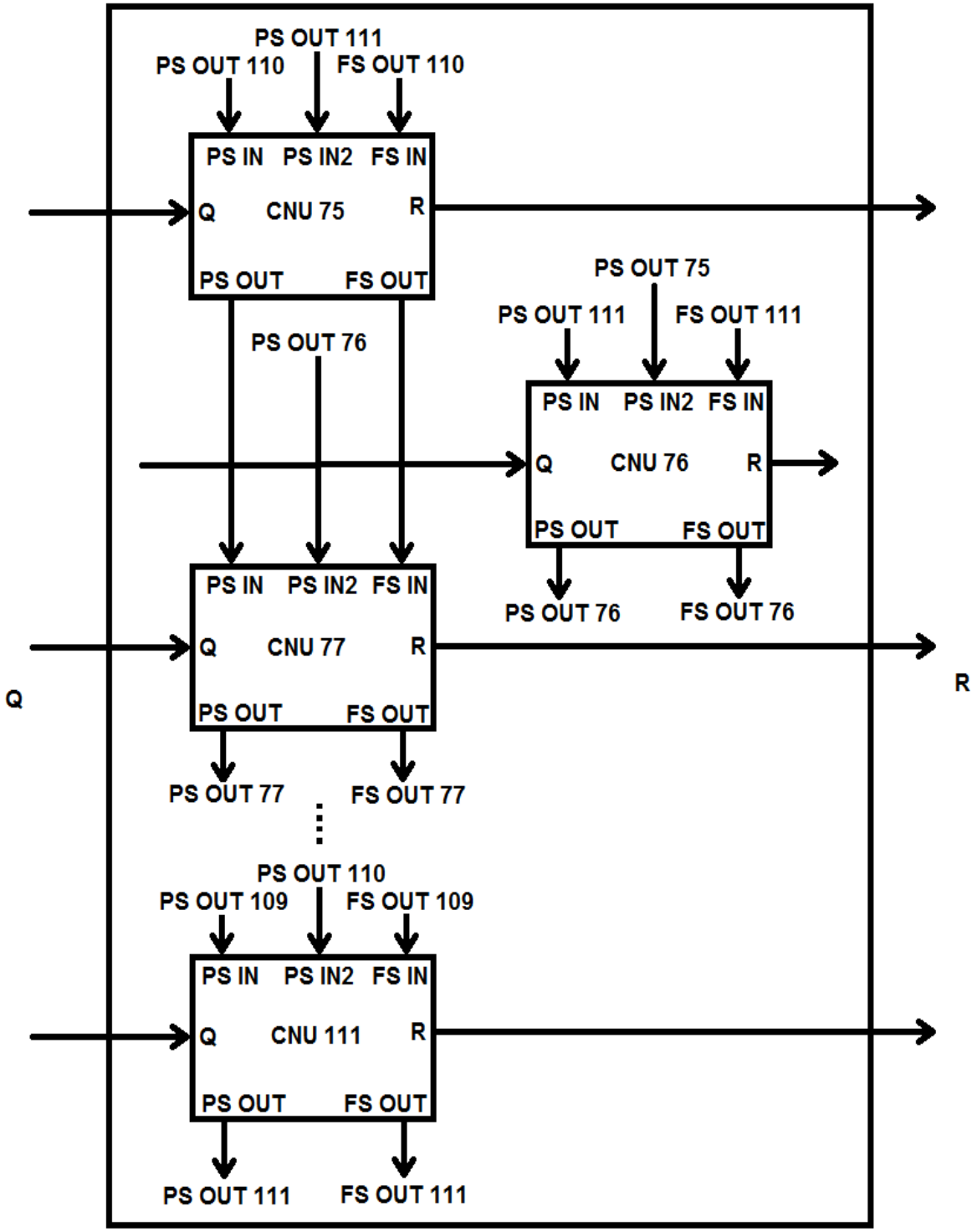


Figure 25: Block diagram of CNU block array 3 showing the connections between the CNUs to achieve the required shift [12**].**

4.1.5 Layered Architecture

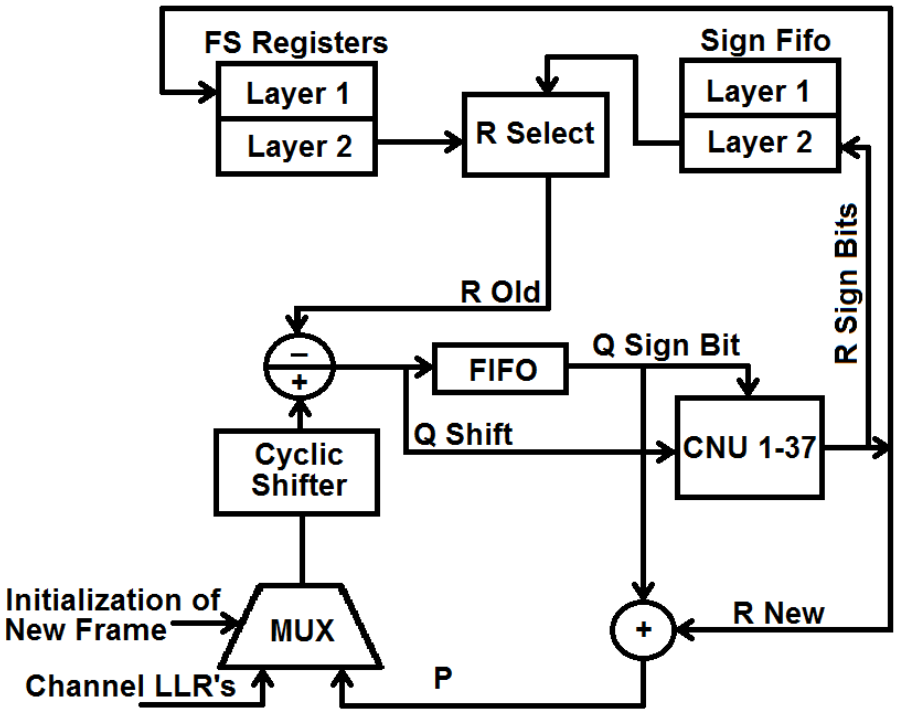


Figure 26: Layered architecture block diagram.

Fig. 26 illustrates the layered architecture for a regular array code of length *259* with check node degree *k=7*, variable node degree *j=3* and identity matrix size *p=37*. This example can be generalized into any other array code by taking the suitable parameters. Various *R* messages are distinguished in following fashion [**12**]:

R old: For certain iteration *i* and a layer *L* being processed, the messages that were computed for the layer *L+1* in the previous iteration are termed as *R old* messages:

R prev: The layer presently being processed in the present iteration has *R* messages from the previous iteration. These are termed as *R prev* messages.

R new: The messages being computed for the present layer for the present iteration are termed as *R new* messages.

The CNU array *1-37* is composed of *37* computation subunits that compute the partial state for each block row to produce the *R* messages in block serial fashion. The final state information of the previous block rows is required for the message passing algorithm. The final state information is stored in register banks. There is one register bank of depth *j-1* which is *2* in this example connected with each CNU. In addition to the shifted *Q* messages, the CNU array takes as input the sign information from previously computed *Q* messages in order to perform *R* selection operation. The sign bits are stored in sign FIFO. The total length of the sign FIFO is *k* and each block row has *p* one bit sign FIFOs. There are *j-1* such FIFO banks in total. *p R* select units are used for generation of *R* old. An *R* select unit generates the

R messages for *k* edges of a check node from three possible values stored in a final state register associated with that check node in serial fashion. The *R* select unit can be treated as a decompressor of the check node edge information which is stored in compact form in FS registers. This leads to substantial memory savings [**12**].

In the beginning of the decoding operation, the *P* vector is set to receive channel values in the first *k* clock cycles (first sub iteration) as the channel values arrive in chunks of *p*. The output of the *R* select unit is set to a zero vector in the beginning. The MUX at the input of the cyclic shifter takes care of this initialization.

The output of the cyclic shifter is fed to the CNU array serially, and is operated upon by the partial state stage. After *k* clock cycles the partial state processing is complete and the final state stage will produce final state for each check node in *1* clock cycle. After this, the *R* select unit within each CNU starts producing k values of check node messages in serial fashion. Thus, check node messages are produced in block serial fashion by the CNU array, as there are *p* CNUs in parallel. The *P* vector is then computed by adding the delayed version of the *Q* vector (which is stored in FIFO until serial CNU produces the output) to the output vector *R* of the CNU array. The *P* vector thus generated can be used immediately to generate the *Q* vector, which is input to the CNU array, as CNU array is ready to process the next block row. This is possible by splitting the CNU processing into *3* stages by pipelining so that partial stage and final stage can operate simultaneously on different block rows. The *P* message vector has to undergo a cyclic shift, based on the block row being currently processed, which is the amount of difference in shifts in the block row which is currently being processed and the block row which was last processed. Based on positive or negative value of the shift, up shift or down shift can be performed. The *R* message is subtracted from the shifted *P* message to produce the shifted version of the *Q* message.

Pipelining: Let us assume partial state of CNU is operating on 2^{nd} block row from clock cycles *0* to *6*. Final state stage cannot start until the end of the partial state processing that is clock cycle *7*. As soon as the final state is done in clock cycle *7*, *R* select is able to select the output *R* messages, and *P* and *Q* message processing starts. With the first block row of *Q* message ready, partial state for the next block row can be started immediately. In this way, pipelining can be done to increase the throughput. The control unit contains information about the array code parameters such as *j, k* and *q*. These parameters can be changed to support multi-rate coding.

Chapter 5

5 System Level Implementation on an FPGA Board

In this chapter, we will talk about how the design is implemented on a Kintex-7 FPGA kit. The whole architecture is explained in details. The blocks like CNU, VNU, barrel shifter, 14:7 MUX, signed to 2's complement number converter, 2's complement to signed number converter and the control unit have been implemented on Xilinx ISE platform using the hardware description language VHDL. After the synthesis of the codes, schematics were generated which have been shown. Also the blocks are verified on test benches, and results are shown. The synthesis reports have also been documented as generated in Xilinx ISE.

5.1 FPGA Kit Kintex-7 Details

We intend to implement the layered min-sum LDPC decoder on an FPGA kit. The non-layered architecture takes a larger amount of area than the layered one. We also intend to compare the area and throughput of the layered and non-layered architecture. All the implementations are targeted on a Kintex-7 FPGA board with model number Kintex-7 KC-705. Various features of the FPGA kit are given below:

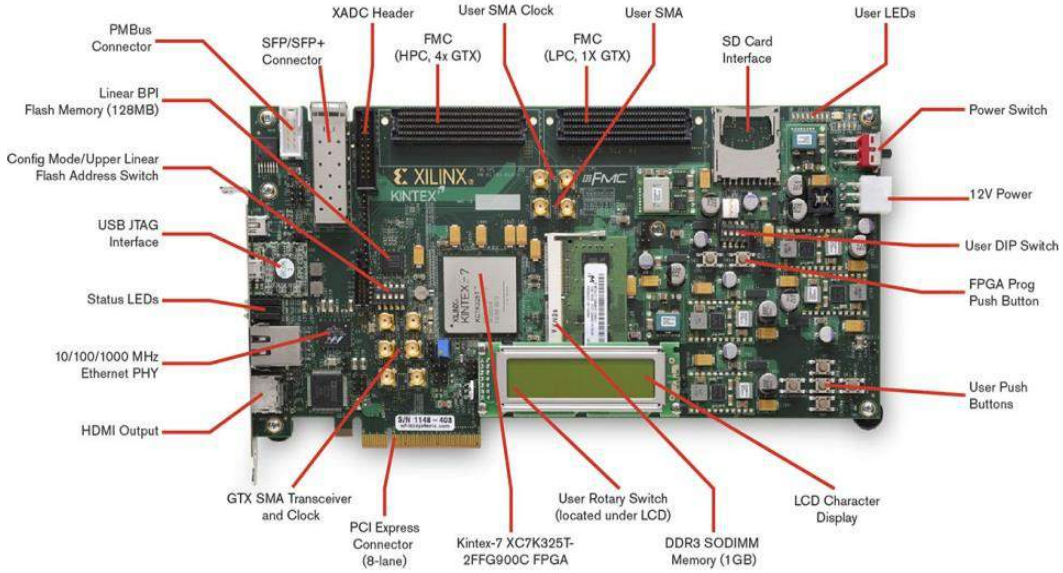


Figure 27: Picture of a Kintex 7 KC-705 FPGA kit.

The key features of the board are:

Configuration

- On-board JTAG configuration circuitry to enable configuration over USB
- 128MB (1024Mb) Linear BPI Flash for PCIe Configuration
- 16MB (128Mb) Quad SPI Flash

Memory

- 1GB DDR3 SODIMM 800MHz / 1600Mbps
- 128MB (1024Mb) Linear BPI Flash for PCIe Configuration
- 16MB (128Mb) Quad SPI Flash
- 8Kb IIC EEPROM

Communication & Networking
- Gigabit Ethernet GMII, RGMII and SGMII
- UART To USB Bridge

Display
- 2x16 LCD display
- 8x LEDs

Clocking
- Fixed Oscillator with differential 200MHz output
- Used as the "system" clock for the FPGA
- Programmable Oscillator with 156.250 MHz output

Control & I/O
- 5X Push Buttons
- 4X DIP Switches
- 7 I/O pins available through LCD header

Block diagram of the FPGA kit:

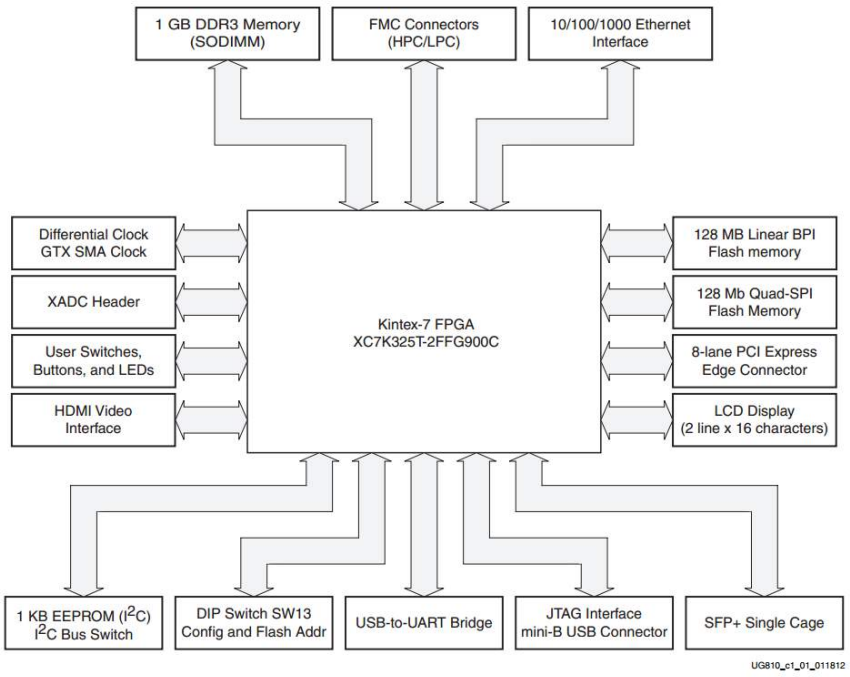


Figure 28: Kintex 7 KC-705 FPGA kit block diagram.

5.2 Architectural Details of the Decoder Designed on FPGA

The complete detailed block level diagram of the decoder is shown in Fig. 30. For verification of the concept, we used smaller block length *28*, with a smaller parity check matrix having size *21*x*28*. *5* bit precision was used for the design. But owing to the additions taking place in VNU's and resulting increase in size of data bits, we padded two *0's* to every input, so that input was available in 7 bit format. But, the architecture is perfectly scalable to relatively larger block lengths with appropriate sizing and changes in the *VHDL* code. We designed a non-layered decoder architecture to maximize the throughput. The architecture contains *7* basic blocks:

- Check Node Unit.
- Variable Node Unit.
- Barrel Shifter.
- Signed to 2's complement converter.
- 2's complement to signed converter.
- 14:7 Multiplexer (MUX).
- Control Unit.

The incoming values are stored in *7* parallel *FIFO*'s with a depth of at least *8* to accommodate for an extra incoming frame. As shown in the Fig. *FIFO 1* contains 1^{st}, 8^{th}, 15^{th} and 22^{nd} value. *FIFO 2* contains 2^{nd}, 9^{th}, 16^{th} and 23^{rd} value. It goes on in similar fashion and *FIFO 7* contains 7^{th}, 14^{th}, 21^{st} and 28^{th} value. They are stored in such way because in array type codes there is a relative shift of *1* between consecutive rows of the same block, and it would be easier to feed the values to *CNU*.

The barrel shifters are fed values through a *MUX* which determines whether the new frame has to be passed or to be used from the *VNU* outputs for the next iteration. We use a control signal "*newfrm*" for this determination. When the frame has been corrected or the maximum number of iterations has been reached, "*newfrm*" is forced a '*1*' and new frame is loaded. Otherwise, "*newfrm*" remains '*0*' and *VNU* messages are used.

The outputs of the *MUX*es are connected to 3 parallel barrel shifters. The barrel shifter contains *3* stages of *7* numbers of *14:7 MUX* each. The stages produce shifts of *1, 2,* and *4* respectively. A stage *n* can either produce a shift of 2^{n-1} or *0* based on the value of *shift_b(n-1)* received. So a barrel shifter can produce a maximum shift of *7*. The first barrel shifter is not required for array type codes, as for array codes the 1^{st} block row has *0* shifts in each block column as they contain identity matrices in the first block row. The next block rows have shifts of *[0, 1, 2, 3]* and *[0, 2, 4, 6]* respectively. These shifts are stored in registers and the control logic decides which shift is to be loaded at which clock cycle.

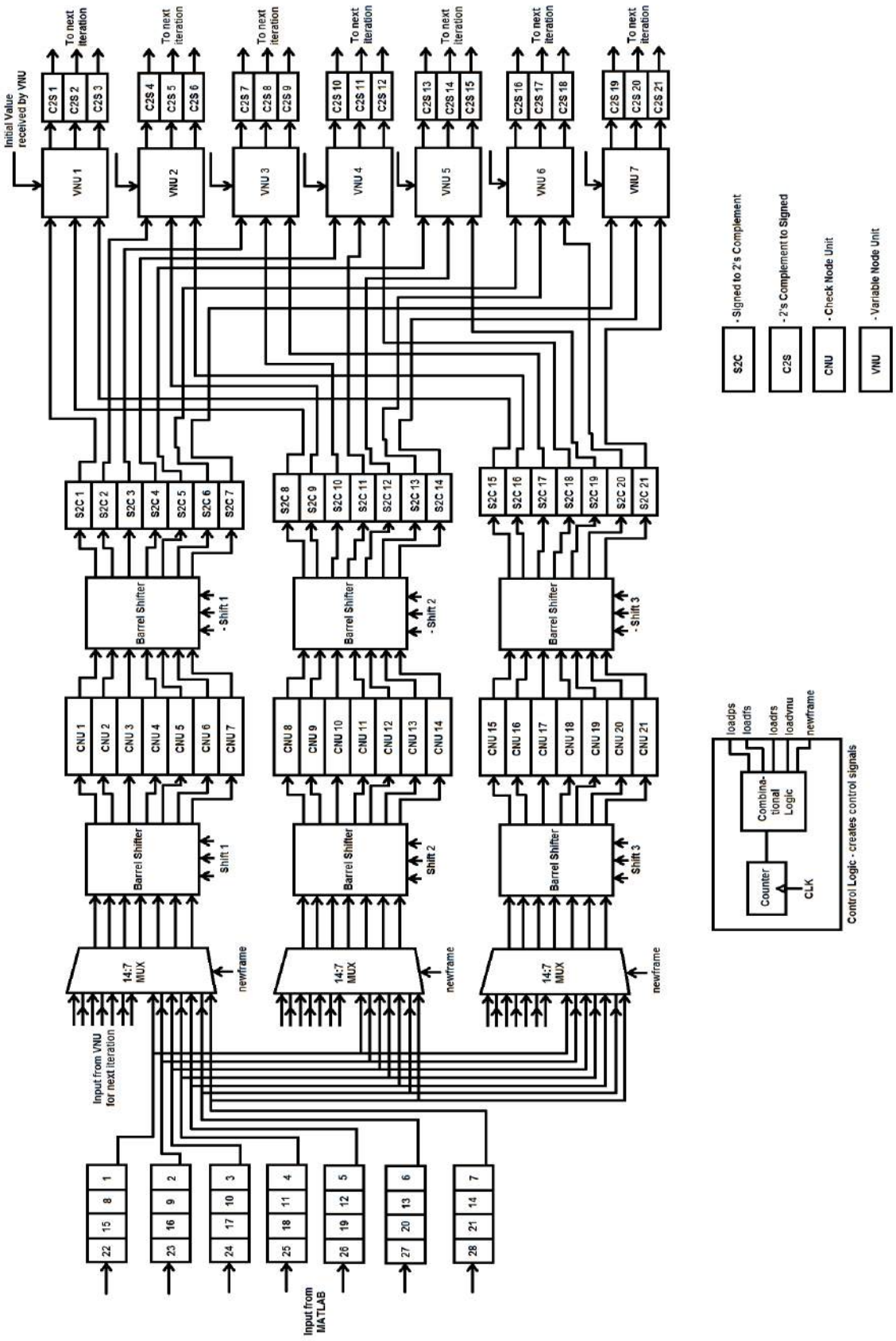


Figure 29 : Detailed block diagram of the system generated on FPGA.

The output of each barrel shifter is connected to *7 CNU*s. There is a total of *21 CNU*'s which operate in parallel. Each *CNU* receives a value per clock cycle for *4* clock cycles. The partial minimum and second minimum updates are done during this and after *4* clock cycles, the final state is updated after applying a normalization factor. This takes an extra clock cycle. So after *6* clock cycles, the final state is updated. After this the R select operation, i.e., the process of generation of the check node messages starts. Each clock cycle gives out *1 R* message. But, we do not need to wait for generation of all the *4* messages for starting the *VNU* operation. As soon as an *R* message is generated, it is sent to next stage containing 3 barrel shifters in parallel. These barrel shifters are needed to produce reverse routing, i.e., negating the effect of the shifts produced by the first set of barrel shifters. The shift entries to this set of barrel shifters are [*0, 0, 0, 0*], [*0, -1, -2, -3*] and [*0, -2, -4, -6*] respectively. The negative value indicates that it does opposite of the shift produced by the 1^{st} set of barrel shifters. In operation, it means that instead of doing down shifts, it produces up shifts.

For the previous part, we had been using signed number representation because, *CNU* works on the magnitude and signs separately, and there is no subtraction arithmetic operation involved. So, it is quite easier to use signed representation. But in the next part of the architecture, that is Variable Node Operation, additions and subtractions are involved. So, it is wise to convert the signed numbers into *2's* complementary form for the ease of these operations, as most of the VHDL libraries work on 2's complement arithmetic. So, the output of the 2^{nd} shift of barrel shifters is fed into *21* parallel *S2C*'s (Signed to *2's* complement converter) to achieve this operation. The *S2C* checks whether the number is positive or negative. For positive numbers it supplies the same output as input. But for negative numbers, it inverts all the bits in the number and adds *'1'* to it, to supply the output in *2's* complement form. The outputs of the *S2C's* are fed to *7* parallel VNU's which produce *21* messages per clock cycle. These messages are converted from *2's* complement to signed representation, are the *CNU* operations are to start next. This can be achieved by using *21* parallel *C2S's* (*2's* complement to signed converter). These blocks do the same operation as *S2C's*.

After *VNU* operation has been done, we need a syndrome check to ascertain whether the frame has been corrected or not. This is done by adding all the values a *VNU* receives from the *S2C's* in one clock cycle along with the intrinsic value and checking the sign bit of the result. This is done over *4* clock cycles (to incorporate for all the check node messages) for all the *7 VNU's*. We get *28* such sign bits. If all of them are *0*, the frame has been corrected. This is due to the fact that we have used an all zero codeword which was affected by AWGN noise, in a memory less channel, which means that all codewords will be affected in same way, and working on a certain codeword will give an estimate of the whole thing. Taking all zero codewords eases the process of syndrome check in hardware. If *1* or more of them are not *'0'* then syndrome check result is negative and the frame has not been corrected. Based on all these *28* results, a signal *"newfrm"* is created. *"newfrm"* is *'1'* only if all results are *'0'* or or maximum iteration limit has been reached. In this way the whole decoder works.

Th control logic block creates one bit signals like *"loadps", "loadfs", "loadrs"* and *"loadvnu". "loadps"* determines when to load new values into the *CNU*. It is *'1'* for *4* clock

cycles in each iteration. *"loadfs"* controls the loading of the final state registers from the partial state updates. It is *'1'* for only *1* clock cycle per iteration. *"loadrs"* determines when to load the *R* select unit. This also remains *'1'* for *4* clock cycle per iteration.

5.3 Detailed Clock Details and Throughput Calculation

Operations happening at various clock cycles are shown below:

1 – 14:7 Mux is loaded with first values.

2 – Barrel shifter is loaded with first values.

3 – CNU operation starts with partial state being loaded.

7 – Final State is loaded.

8 – R select opearation starts.

9 – Next set of barrel shifters are loaded.

10 – S2C's are loaded.

11 – VNU's are loaded.

12 – C2S's are loaded.

16 – Next iteration starts.

Throughput Calculation: Each iteration takes 15 clock cycles. With the worst case of 5 iterations, it will take 80 clock cycles. The throughput can be calculated theoretically as follows:

Let us assume that on an average it takes n number of iterations for each frame. Each frame has 28 x 7, i.e., 196 bits. The clock frequency is 100 MHz and clock period is this 10 ns. It takes 15 clock cycles per iteration. So for n iterations, it will take n x 15 number of clock cycles. So in 15n clock cycles we are getting 196 bits at output, or in other words, we are getting 196 bits every 150n nanoseconds. Therefore, in 1 second we are getting $196/(150n)*10^9$, i.e., (1306.667/n) Mbps. For example if the average number of iterations is 5, then throughput is 261.33 Mbps.

5.4 Basic Blocks as Generated in Xilinx ISE

5.4.1 CNU

Fig. 30 shows the block schematic of the CNU as generated in Xilinx ISE. It has one data input signal and three control input signals, “loadps”, “loadfs” and “loadrs”. It has a latency of 10 clock cycles. Fig. 32 shows the RTL schematic the CNU at one lower level.

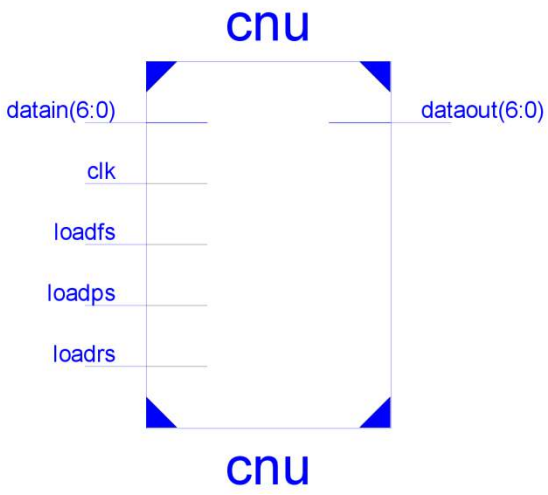


Figure 30: CNU block inputs and outputs. It has one data input signal and three control input signals, “loadps”, “loadfs” and “loadrs”.

5.4.2 VNU

Fig. 31 shows the block schematic of the VNU block as generated in Xilinx ISE. It has 4 data inputs and 3 data outputs. It has one control signal “loadvnu” as input. It has a latency of 1 clock cycle. Fig. 33 shows the RTL schematic of the VNU unit at one lower level.

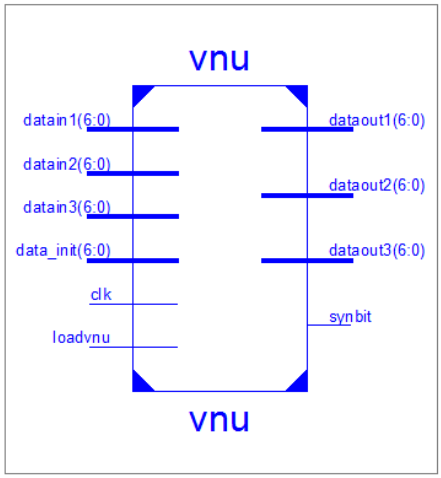


Figure 31 : VNU block inputs and outputs. It has4 data inputs and 3 data outputs. It has one control signal “loadvnu” as input.

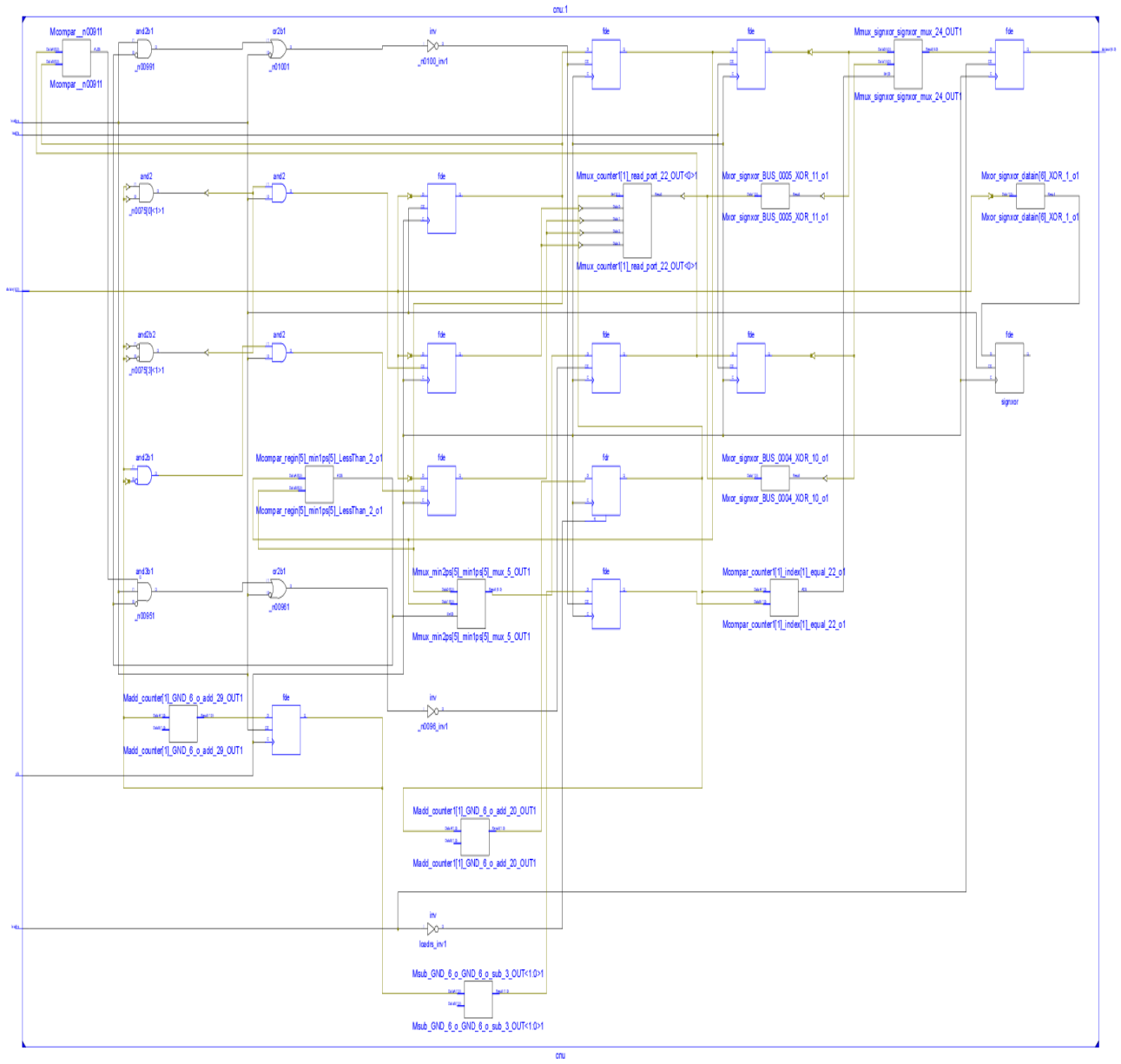


Figure 32 : CNU schematic as generated on Xilinx ISE.

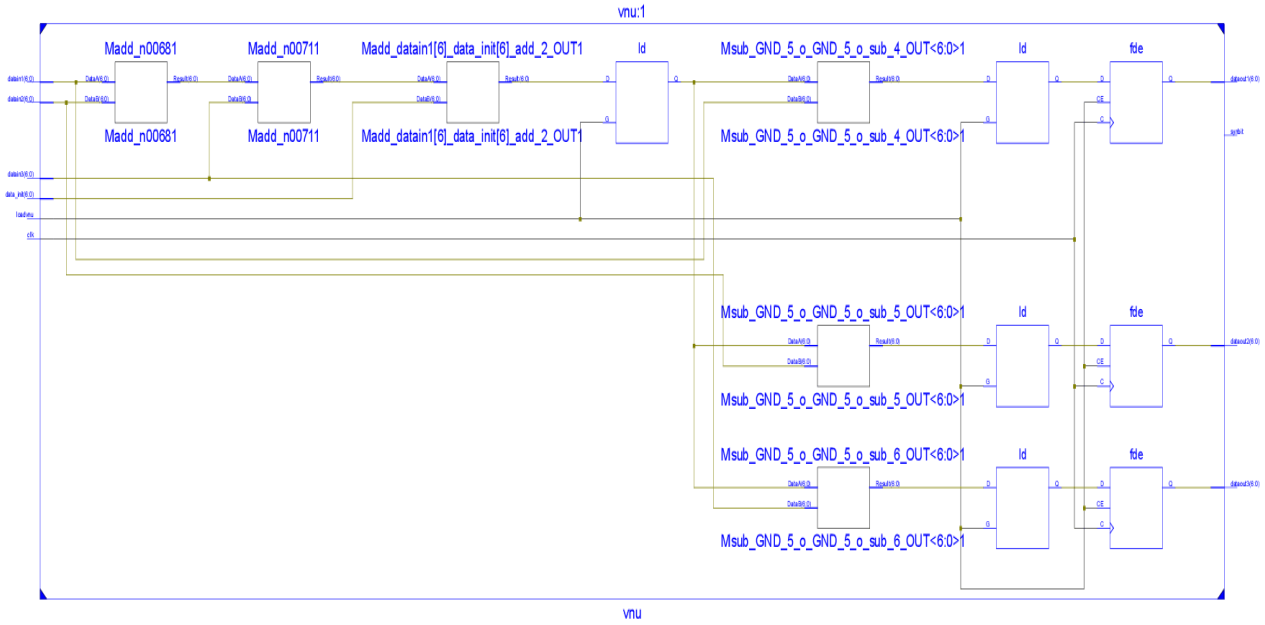


Figure 33 : VNU schematic as generated on Xilinx ISE.

5.4.3 Barrel Shifter

Fig. 34 shows the block schematic of the barrel shifter as generated in Xilinx ISE. It has 7 data inputs and an input "shift_b" which determines the amount of shift produced. It has 7 data outputs. Fig. 35 shows the RTL schematic of the barrel shifter at a lower level. It has a latency of one clock cycle.

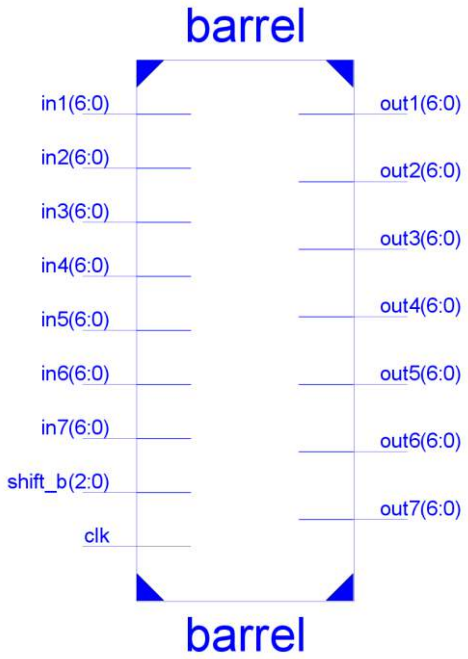


Figure 34 : Barrel shifter block inputs and outputs. It has 7 data inputs and an input "shift_b" which determines the amount of shift produced. It has 7 data outputs

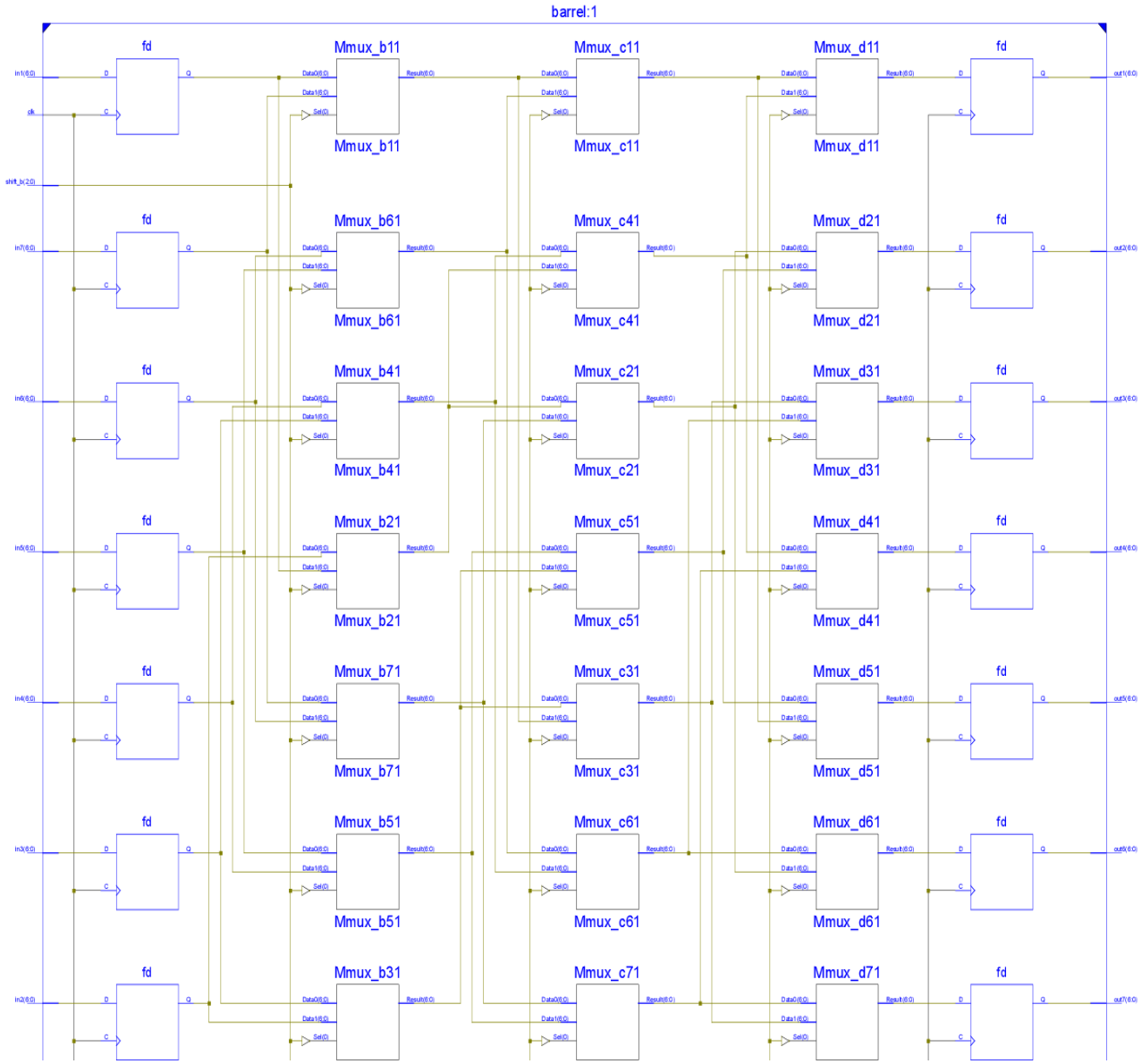


Figure 35 : Barrel shifter schematic as generated on Xilinx ISE. It has 3 stages which produce shifts of 1, 2, 4 respectively determined by value of shift_b.

5.4.4 Signed to 2's Complement Converter

Fig. 36 shows the block schematic of the S2C as generated in Xilinx ISE. It has 1 data input and 1 data output. Fig. 37 shows the RTL schematic of the S2C at a lower level. It has a latency of one clock cycle.

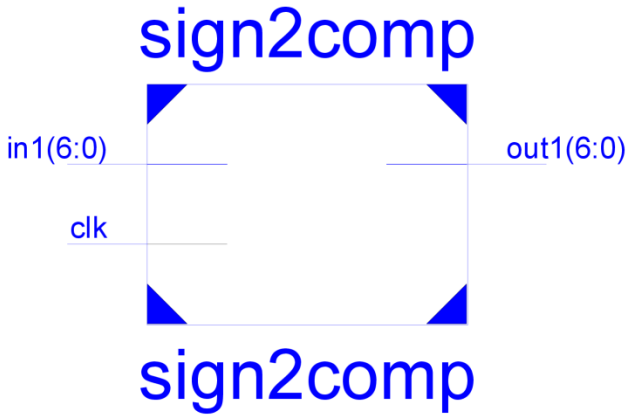


Figure 36 : Signed to 2's complement block inputs and outputs. It has 1 data input in signed form and 1 data output in 2's complement form.

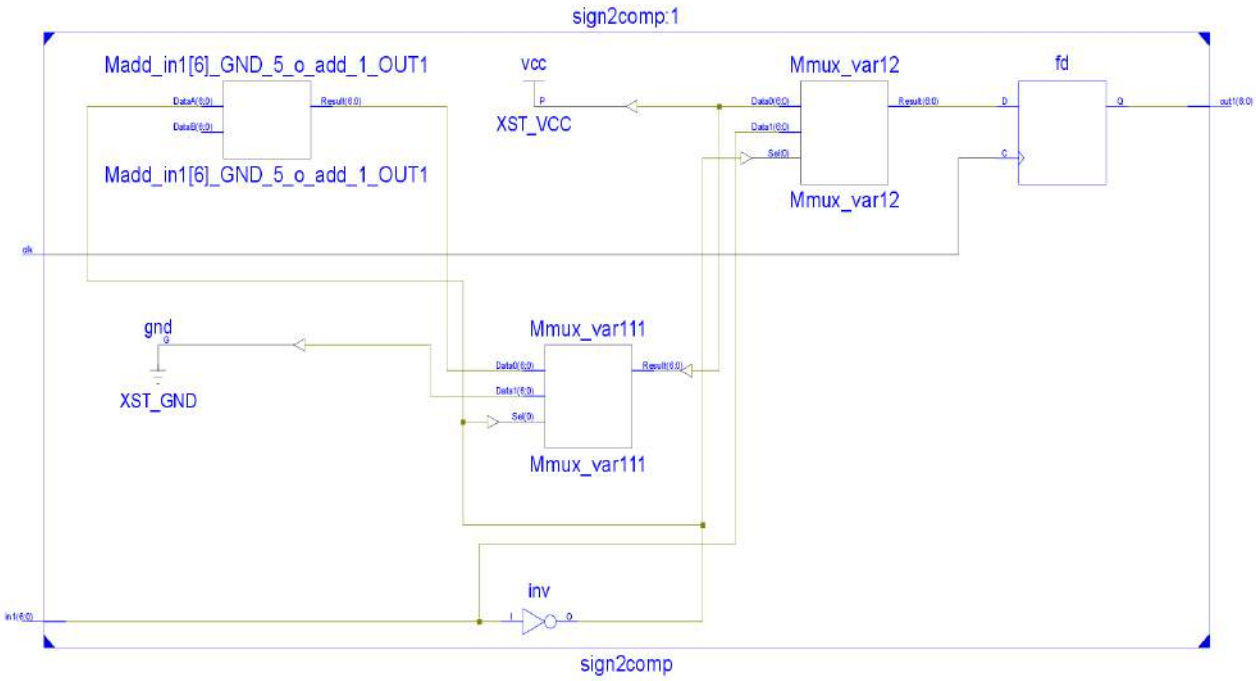


Figure 37 : Signed to 2's complement schematic as generated on Xilinx ISE.

5.4.5 14:7 Multiplexer

Fig. 38 shows the block schematic of the 14:7 MUX as generated in Xilinx ISE. It has 14 data inputs and a select input “newfrm”. It has 7 data outputs. Fig. 40 shows the RTL schematic of the barrel shifter at a lower level. It has a latency of 1 clock cycle.

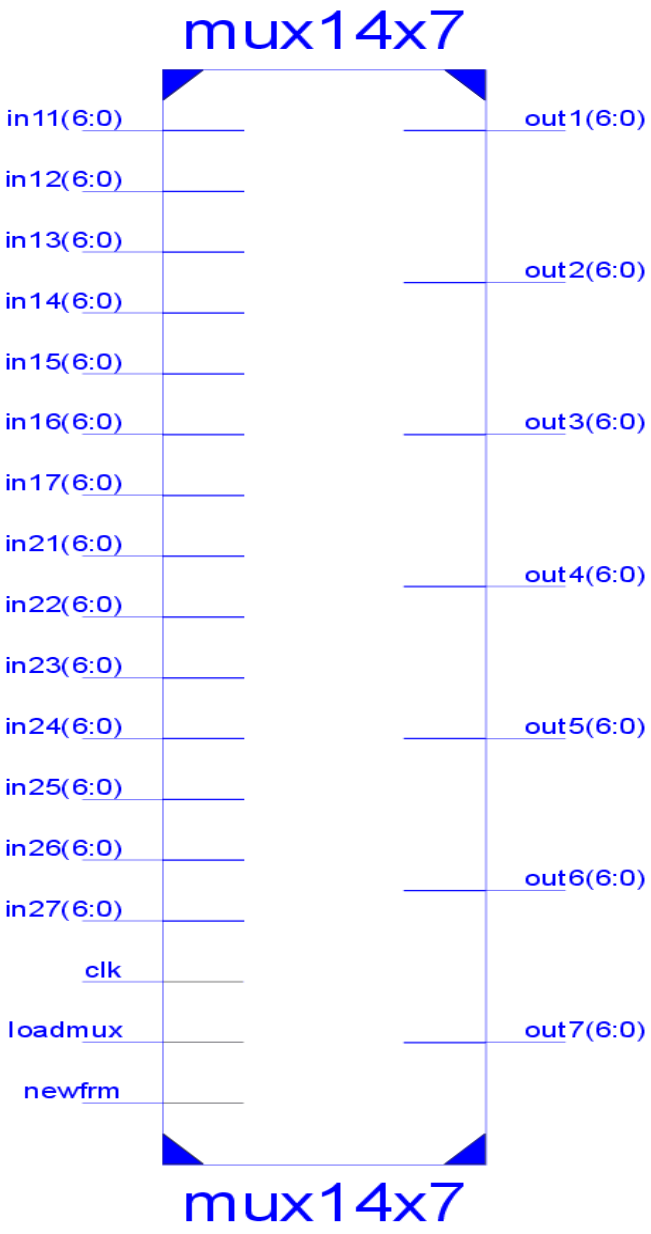


Figure 38 : 14:7 MUX inputs and outputs. It has 14 data inputs and a select input “newfrm”, based on which 7 data outputs are chosen.

5.4.6 Port Mapping of all the Blocks

Fig. 39 shows the block view of the whole architecture as generated in Xilinx ISE. Fig. 41 shows a lower level view of the whole architecture as generated in Xilinx ISE.

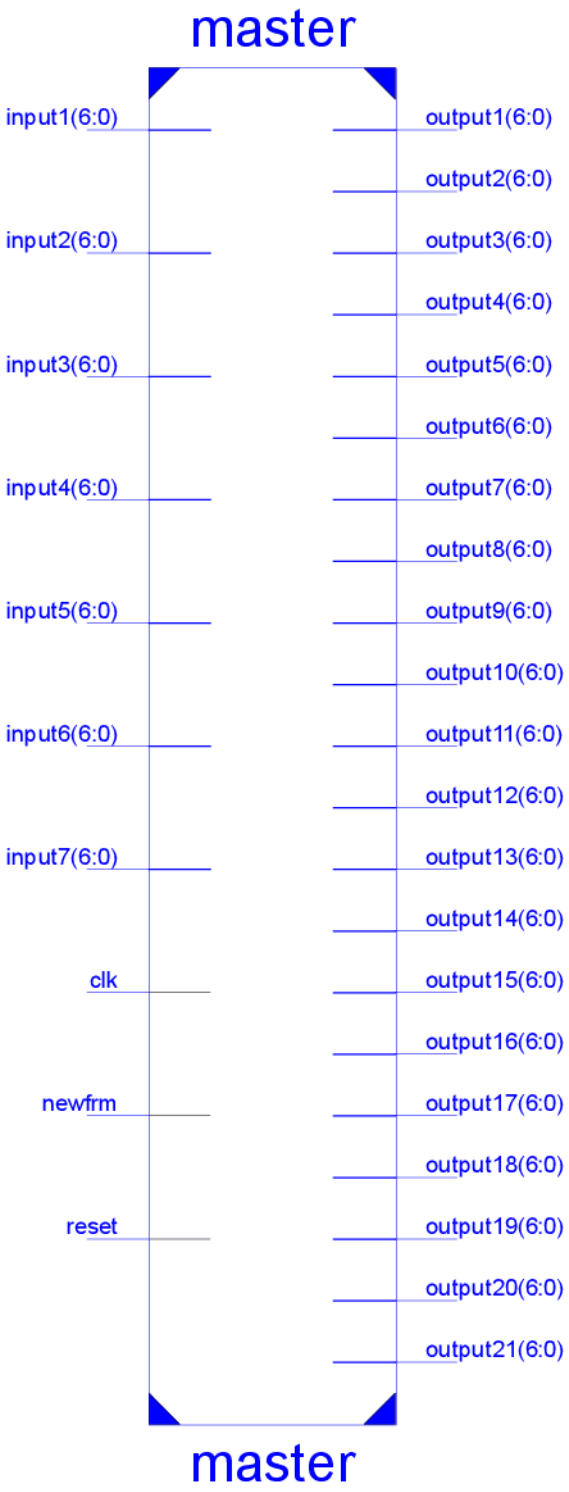


Figure 39 : Master block inputs and outputs. It contains all the blocks port mapped.

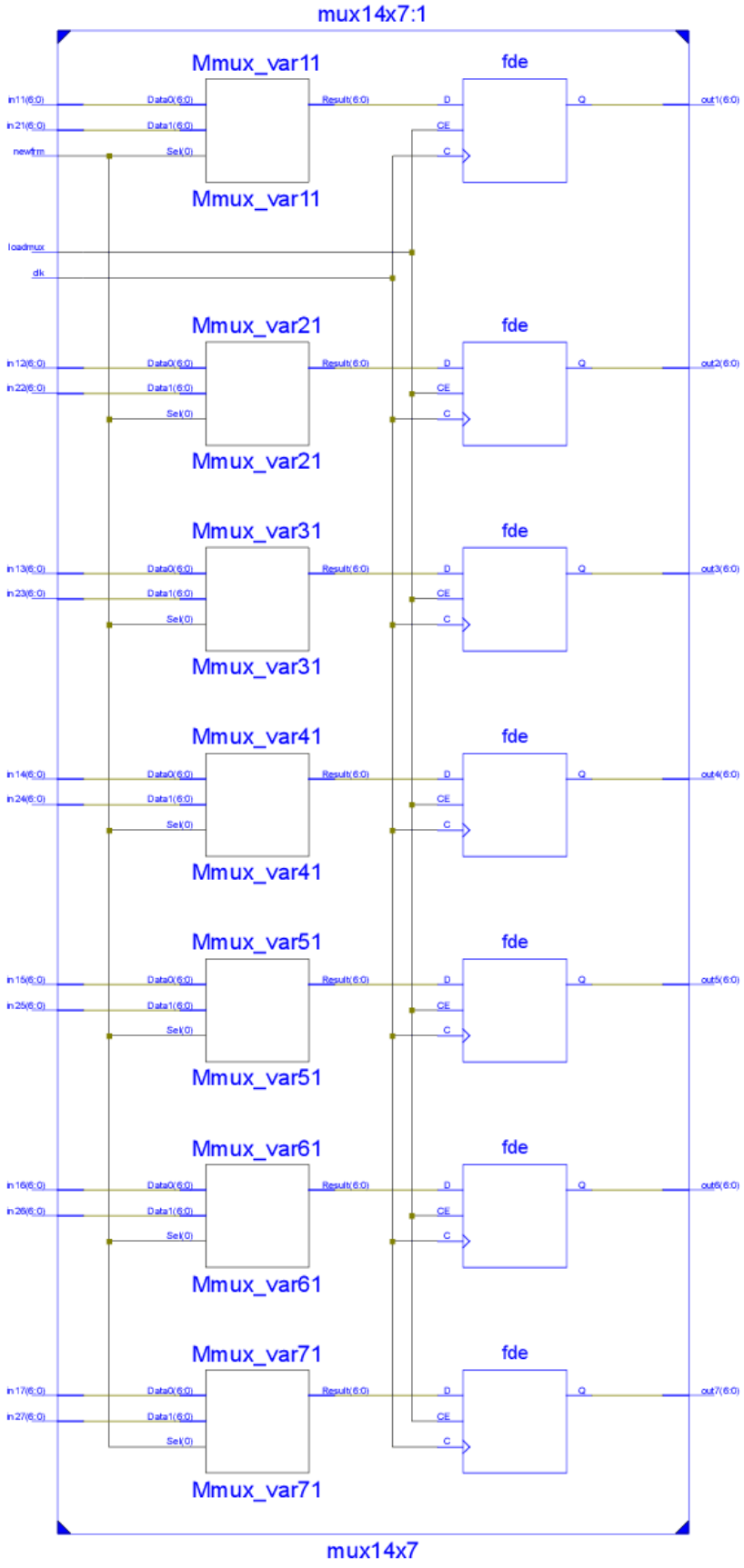


Figure 40 : 14:7 MUX schematic as generated on Xilinx ISE.

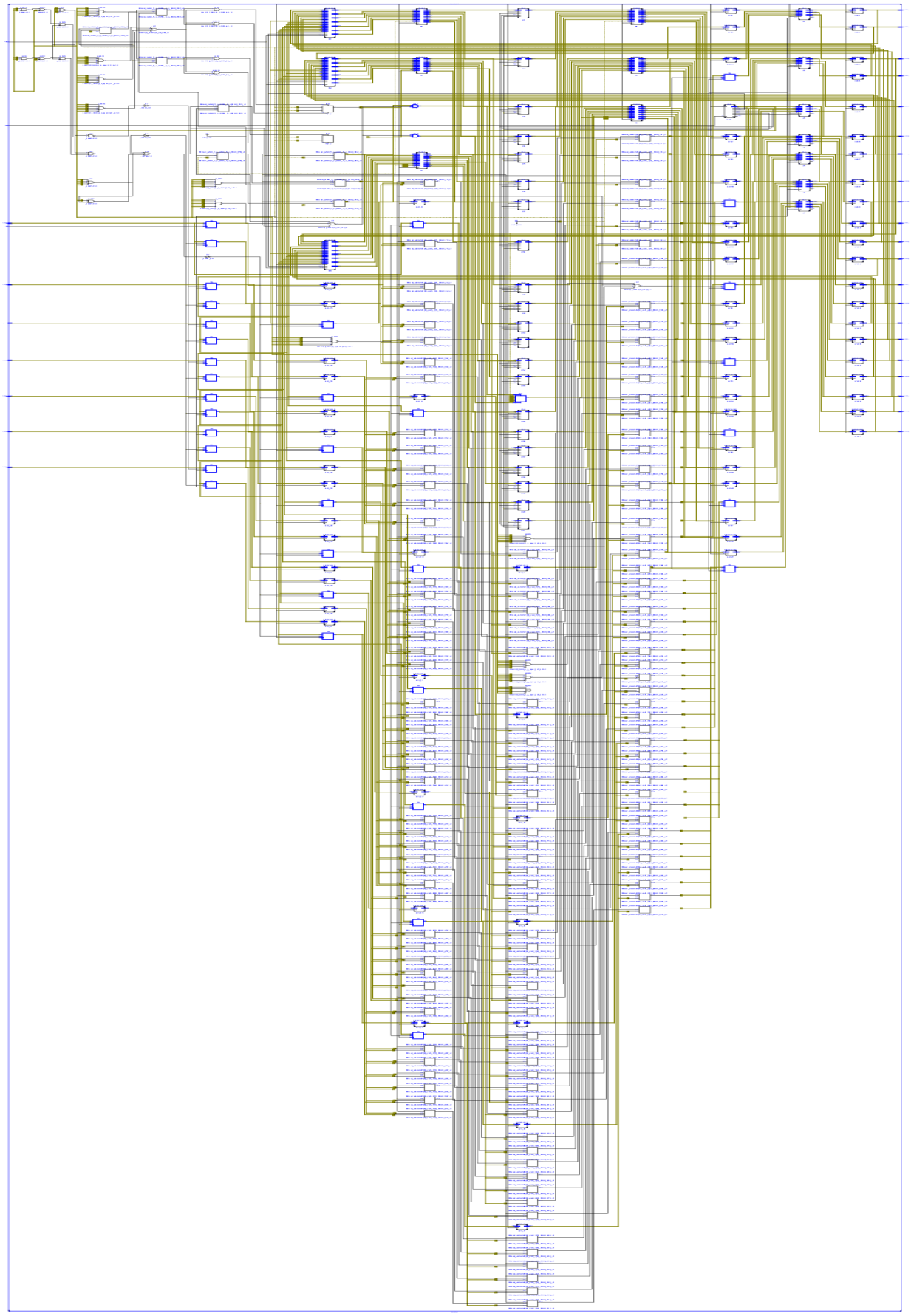


Figure 41 : Master block schematic as generated by Xilinx ISE.

5.4.7 Control Logic

The control logic block produces four control signals “loadps”, “loadfs”, “loadrs”, and “loadvnu”. It has a 5 bit counter, and the control signals are generated based on the state of the counter. The block schematic of the control logic block as generated by Xilinx ISE is shown in Fig. 42. The RTL schematic is shown in Fig. 43.

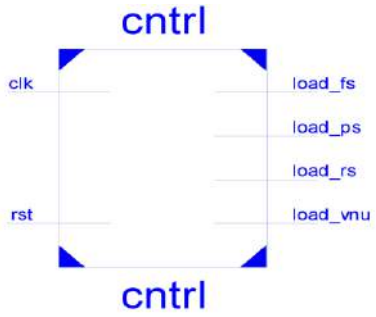


Figure 42 : Control unit block inputs and outputs. The control logic block produces four control signals “loadps”, “loadfs”, “loadrs”, and “loadvnu”.

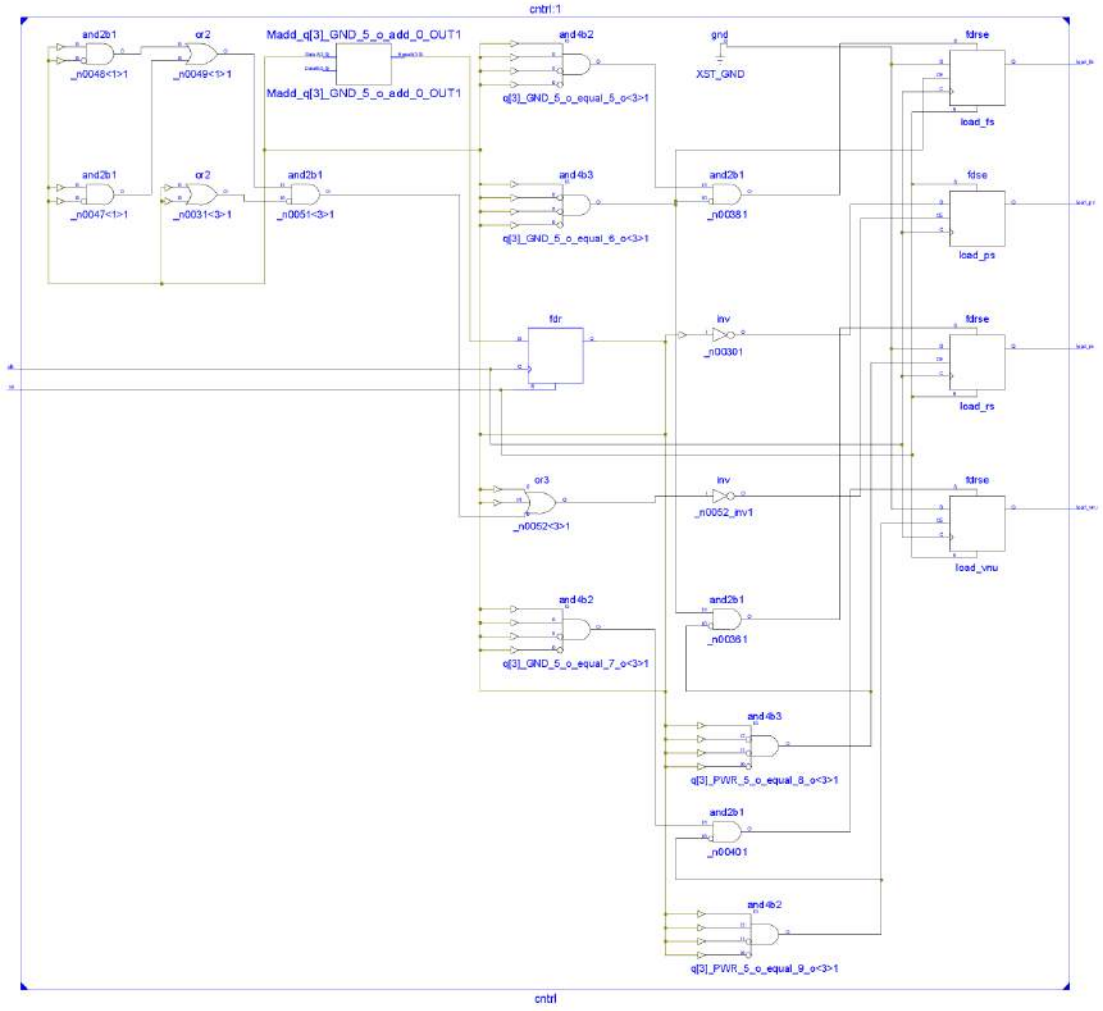


Figure 43 : Control Unit block schematic as generated on Xilinx ISE. It has a 5 bit counter, and the control signals are generated based on the state of the counter.

5.5 Behavioral Simulation Results

5.5.1 CNU Behavioral Simulation Test Bench Results and Synthesis Reports.

The behavioural simulation results on a test bench for the CNU unit are shown in Fig. 44.

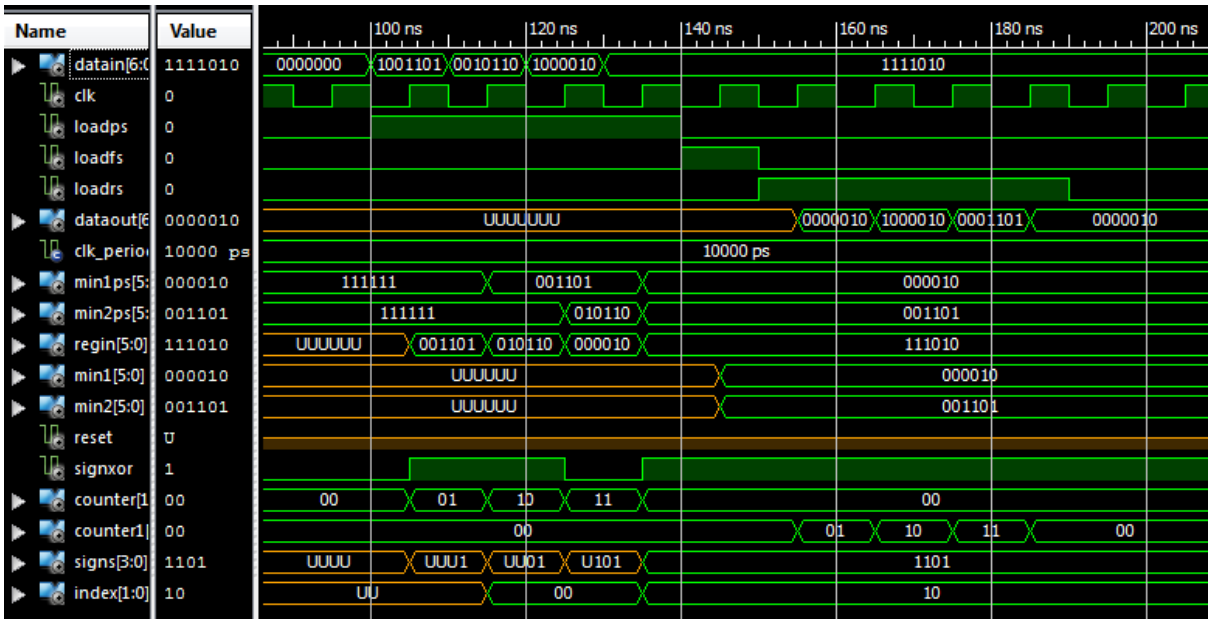


Figure 44 : CNU behavioral simulation on test bench results.

4 values enter the CNU in 4 clock cycles. When "loadps" is forced '1' partial state of minimum and second minimum finder starts. After 4 clock cycles, "loadfs" is forced '1' and final state update occurs at the 5th cycle. At the 6th clock cycle, the R select unit starts giving out R messages in serial fashion as seen in Fig. 44. datain(6:0) is the input vector and dataout(6:0) is the output vector. As, we can see, there is a latency of 5 clock cycles. The advanced HDL synthesis reports as generated in Xilinx ISE are as given below:

No. of Adders/Subtractors = 1
No. of 2-bit subtractor = 1
No. of Counters = 2
No. of 2-bit up counter = 2
No. of Registers = 44
No. of Flip-Flops = 44
No. of Comparators = 3
No. of 2-bit comparator equal = 1
No. of 6-bit comparator greater = 1
No. of 6-bit comparator lessequal = 1
No. of multiplexers = 3
No. of 1-bit 4-to-1 multiplexer = 1
No. of 6-bit 2-to-1 multiplexer = 1
No. of 7-bit 2-to-1 multiplexer = 1
No. of xors = 3

No. of 1-bit xor2 = 3
There was a total delay of 1.582ns (0.361ns logic, 1.221ns route)

5.5.2 VNU Behavioral Simulation Test Bench Results and Synthesis Reports.

The behavioural simulation results on a test bench for the barrel shifter unit are shown in Fig. 45.

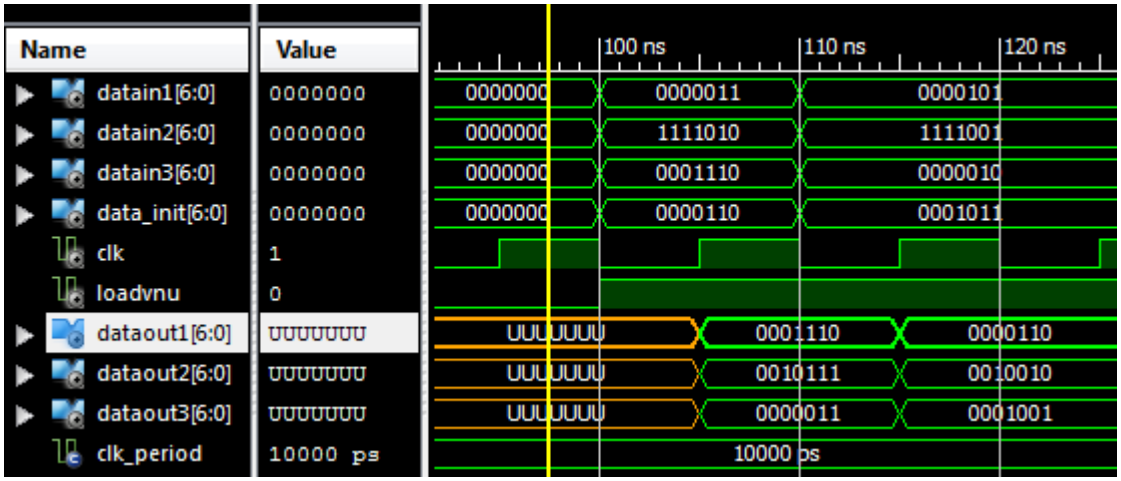


Figure 45 : VNU behavioral simulation on test bench results.

The advanced HDL synthesis reports are as given below:

No. of Adders/Subtractors = 5
No. of 7-bit adder = 5
No. of Registers = 21
No. of Flip-Flops = 21

There was a total delay of 1.799ns (1.098ns logic, 0.701ns route)

5.5.3 Barrel Shifter Behavioral Simulation Test Bench Results and Synthesis Reports.

The behavioural simulation results on a test bench for the barrel shifter unit are shown in Fig. 46.

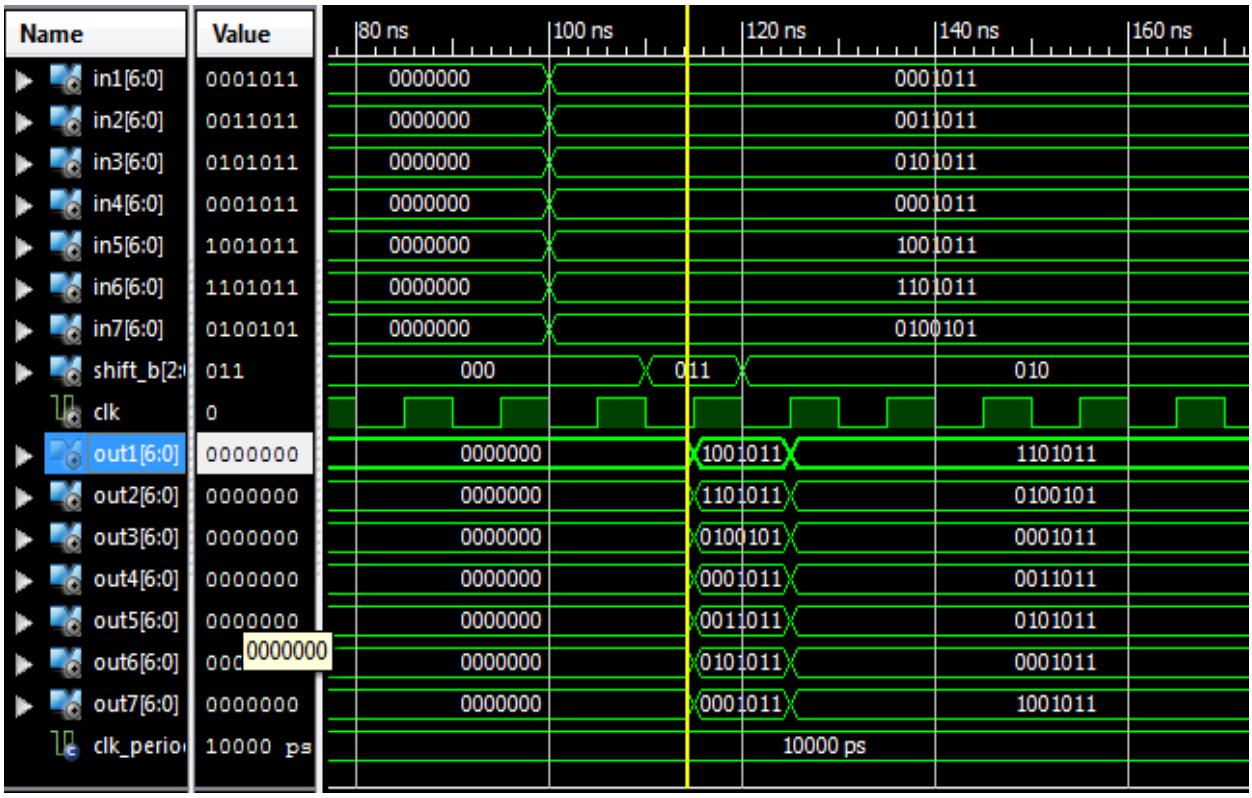


Figure 46 : Barrel shifter behavioral simulation on test bench results.

As shown in Fig. 46, the barrel shifter shifts the inputs by 3 and 2 respectively in 2 successive cycles. The results are found to be correctly tallying with simulation results.

The advanced HDL synthesis reports are as given below:

No. of Registers = 98
No. of Flip-Flops = 98
No. of total multiplexers = 63
No. of 1-bit 2-to-1 multiplexer = 49
No. of 7-bit 2-to-1 multiplexer = 14
Total Combinational Delay = 0.935ns

5.5.4 14:7 MUX Behavioral Simulation Test Bench Results and Synthesis Reports.

The behavioural simulation results on a test bench for the 14:7 MUX unit are shown in Fig. 47.

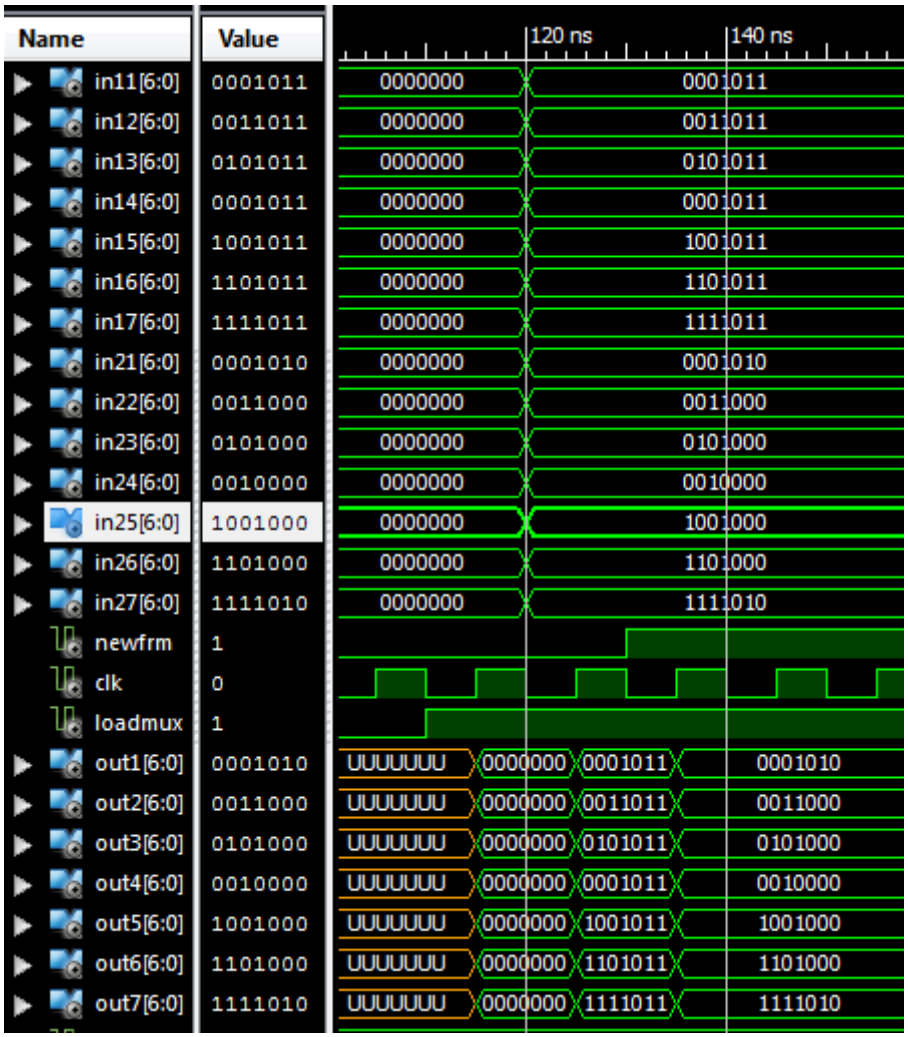


Figure 47 : 14:7 MUX behavioral simulation on test bench results.

As shown in Fig. 47, the MUX deliver the first set of 7 vectors when "newfrm" is '0' else it supplies the second set of 7 vectors and so it is verified that it works correctly.

The advanced HDL synthesis reports are as given below:

No. of Registers = 49
No. of Flip-Flops = 49
No. of multiplexers = 7
No. of 7-bit 2-to-1 multiplexer = 7
Total delay = 0.581ns (0.043ns logic, 0.538ns route)

5.5.5 Control Unit Behavioral Simulation Test Bench Results and Synthesis Reports.

The behavioural simulation results on a test bench for the control unit are shown in Fig. 48.

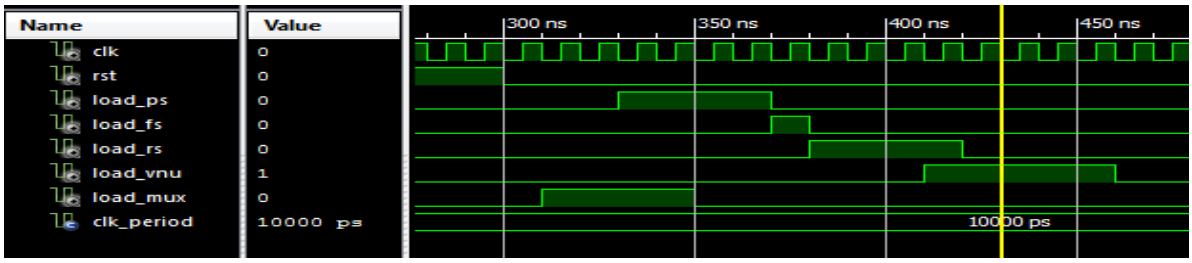


Figure 48 : Control unit behavioral simulation on test bench results.

Fig. 48 shows how the control signals are generated at various clock cycles. The advanced HDL synthesis reports are as given below:

No. of Counter = 1
No. of 5-bit up counter = 1
No. of Registers = 5
No. of Flip-Flops = 5
No. of Multiplexers = 3
No. of 1-bit 2-to-1 multiplexer = 3
Total delay = 0.849ns (0.275ns logic, 0.574ns route)

5.5.6 Signed to 2's Complement Block Behavioral Simulation Test Bench Results and Synthesis Reports.

The behavioural simulation results on a test bench for the signed to 2's complement unit is shown in Fig. 49.

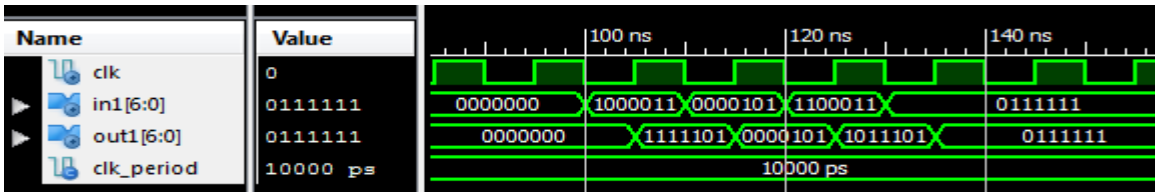


Figure 49 : Signed to 2's complement block behavioral simulation on test bench results.

The advanced HDL synthesis reports are as given below:

No. of Adders/Subtractors = 1
No. of 7-bit adder = 1
No. of Registers = 7
No. of Flip-Flops = 7
No. of Multiplexers = 2
No. of 7-bit 2-to-1 multiplexer = 2
Total delay = 0.841ns (0.086ns logic, 0.755ns route)

Chapter 6

6 Conclusions.

The primary purpose of the project is to design a generic LDPC decoder. The project started with an understanding of the concepts of LDPC codes and the reasons why it is so extensively used for error correction. Various variants of the error correcting decoding techniques were investigated with an eye towards performance and feasible hardware design,. We simulated variants of the min-sum algorithm along with quantization schemes keeping the bit budget, normalization factor in mind for feasible design architecture. Once the quantization parameters were fixed, the designs of the architecture with the various component blocks were constructed. The decoder components were implemented on a Xilinx ISE platform using VHDL language and tested using test benches. Though the architecture was designed for small code lengths, we believe it will be scalable for larger code lengths. All the results and reports have been documented. Various details like throughput and resource utilization have been documented.

From this project, we can conclude that LDPC gives reasonably good error correction for array type codes with smaller block lengths. Non-layered decoder gives high throughput but it takes large amount of area. So far the outputs at various stages of the decoder have been verified, but the whole architecture has to run on real time. Thus, a communication system between Matlab and the FPGA kit has to be established. Further, layered decoding algorithm can also be implemented for comparing it with the non-layered architecture. Non-uniform quantization needs to be investigated for lesser error floors [**13**]. Lower power designs must be investigated for the current needs of system-on-chip solutions (SoCs) [**14**].

.

References

[1] S. Lin and D. J. Costello, *Error Control Coding*, 2nd ed.: Pearson, 2004.

[2] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inform. Theory*, vol. 8, no. 1, pp. 21-28, Jan. 1962.

[3] D. J. C. MacKay, "Good Error-Correcting Codes Based on Very Sparse Matrices," *IEEE Trans. Inform. Theory*, vol. 45, no. 2, Mar. 1999.

[4] M. Daud, A. B. Suksmono, Hendrawan, and Sugihartono, "Comparison of decoding algorithms for LDPC codes of IEEE 802.16e standard," in *Intl. Conf. Tele. Sys.Serv. App.*, pp. 280-283, Oct. 2011.

[5] Y. Cao, "An improved LDPC decoding algorithm based on min-sum algorithm," in *Intl. Symp. Comm. Info. Tech.* , pp. 26-29, Oct. 2011.

[6] J. Zhao, F. Zarkeshvari, and A. H. Banihashemi, "On implementation of min-sum algorithm and its modifications for decoding low-density Parity-check (LDPC) codes," *IEEE Trans. Comm.*, vol. 53, no. 4, pp. 549 - 554, Apr. 2005.

[7] J. Chen, Y. Zhang, and R. Sun, "An improved normalized min-sum algorithm for LDPC codes," in *Intl. Conf. Comp. Info. Science*, pp. 509-512, Jun. 2013.

[8] S. Kim, G. E. Sobelman, and H. Lee, "A Reduced-Complexity Architecture for LDPC Layered Decoding Schemes," *IEEE Trans. VLSI*, vol. 19, no. 6, Jun. 2011.

[9] D. E. Hocevar, "A reduced complexity decoder architecture via layered decoding of LDPC codes," in *IEEE Workshop Sig. Proc. Syst.*, pp. 107-112, Oct. 2004.

[10] S. Ölçer, "Decoder Architecture for Array-Code-Based LDPC Codes," in *Globecom*, pp. 2046-2050, 2003.

[11] K. K. Gunnam, G. S. Choi, M. B. Yeary, and M. Atiquzzaman, "VLSI Architectures for Layered Decoding for Irregular LDPC Codes of WiMax," in *IEEE Intl. Conf. Comm.*, pp. 4542 - 4547, Jun. 2007.

[12] K. K. Gunnam and G. S. Choi, "Low Density Parity Check decoder for regular LDPC codes," US8359522, Jan. 2013.

[13] X. Zhang and P. H. Siegel, "Quantized min-sum decoders with low error floor for LDPC codes," in *IEEE Intl. Symp. Inform. theory*, pp. 2871-2875, Jul. 2012.

[14] A. Darabiha, A. C. Carusone, and F. R. Kschischang, "Power Reduction Techniques for LDPC Decoders," *IEEE Journ. Solid-state circuits*, vol. 43, no. 8, Aug. 2011.