Design and FPGA Implementation of Min-Sum Algorithm and Its Variants

by

Sina Tolouei, M. Sc.

A thesis submitted to the

Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Applied Sciences

Ottawa –Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

Ottawa, Ontario

January, 2008

Your file Votre référence
ISBN: 978-0-494-36833-6
Our file Notre référence
ISBN: 978-0-494-36833-6

The undersigned hereby recommend to the faculty of Graduate Studies and Research acceptance of the thesis, "Design and FPGA Implementation of Min-Sum Algorithm and Its Variants", submitted by Sina Tolouei, in partial fulfillment of the requirements for the degree of Master of Applied Science.

Chair, Department of Systems and
Computer Engineering

Professor Amir H. Banihashemi,
Thesis Supervisor

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario
January, 2008

Abstract

In this thesis, the Min-Sum (MS) algorithm and its modified versions are studied. We study the effects of clipping and quantization on the performance of the successive relaxation min-sum (SR-MS) algorithm. We compare SR-MS with the modified MS algorithm with unconditional correction, known to be one of the best iterative algorithms in terms of performance/complexity tradeoff. We demonstrate that the optimal clipping threshold for SR-MS is a function of the code and in general increases with SNR and the number of quantization bits. For practical proposes however, the optimal clipping threshold can be assumed constant over a relatively small range of SNR values of interest. We also show that for the tested codes, SR-MS with 7-bit quantization outperforms floating-point belief propagation (BP) algorithm, particularly at high SNR values. This is while unlike BP, SR-MS does not require an estimate of SNR.

To verify the performance of the proposed modification in comparison with quantized MS and MS with unconditional correction, we implement the three algorithms on a Virtex-II Pro FPGA. The circuit utilization and throughput of the 4-bit quantized MS algorithm with its modified versions are compared. We show that for MS with unconditional correction, the circuit utilization increases by 2% and the throughput is the same as that of MS. Also, the results for SR-MS implementation show that the device utilization is increased by 22% and the throughput is decreased by 20%.

Acknowledgements

I would like to thank my supervisor Professor Amir H. Banihashemi for his support and encouragement. His guidance and insightful comments inspired me a lot during my research work.

I can not express my appreciation to my wonderful wife, Nona who has always been besides me with all her love and support. I also want to thank my parents, sister and brother, parents-in-law and brother-in-law who encouraged and supported me emotionally throughout my studies.

I would also like to thank Intel Corporation by supporting this project.

This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET).

Table of Contents

List of Tables

List of Figures

List of Appendices

List of Abbreviations and Symbols

Abbreviations

LDPC	Low-Density Parity-Check
BP	Belief Propagation
MS	Min-Sum
SR	Successive Relaxation
SNR	Signal-to-Noise Ratio
BER	Bit Error Rate
LLR	Log-Likelihood Ratio
AWGN	Additive White Gaussian Noise
BIAWGN	Binary Input Additive White Gaussian Noise
BPSK	Binary Phase Shift Keying
SP	Sum-Product
UC	Unconditional Correction
FPGA	Field Programmable Gate Array
MSB	Most Significant Bit
LUT	Look Up Table
FF	Flip Flop

Symbols

C	code
k	message length
n	codeword length
H	parity-check matrix
v	n-tuple
a	average energy per transmitted bit
$\frac{N_0}{2}$	power spectral density of AWGN
E_b	average energy per information bit
σ^2	variance of additive Gaussian random variable
c_i	bit to be decoded
P_i	probability of error
$L(q_{ij})$	message from variable node i to check node j
$L(r_{ji})$	message from check node j to variable node i
V_j	set of all the neighbors of check node *j*
$V_j \setminus i$	set of all neighbors of check node j excluding the one coming from variable node *i*
C_i	set of all the neighbors of variable node *i*
$C_i \setminus j$	set of all neighbors of variable node i excluding the one coming from check node j

$L()$	log function
$\phi()$	$\ln\frac{e^x+1}{e^x-1}$
c_{th}	clipping threshold
y_c	given non-negative correction value
X_t	messages at iteration t
X_{t+1}	messages at iteration t+1
β	relaxation factor

Chapter 1 Introduction and motivation

1.1 Introduction

In a digital communication system, the goal is to use the available resources like bandwidth and power efficiently to achieve a certain probability of error with less power. Error control coding consists of a channel encoder and a channel decoder and is one of the main parts of a digital communication system [1]. The channel encoder adds redundant bits at the transmitter to make data transmission robust to channel noise. The channel decoder uses the redundant bits to correct the errors and extracts the original sequence of information.

In 1993, Turbo codes were introduced [2] and they demonstrated the ability of iterative decoding to get near Shannon limit performance. Inspired by Turbo code results, low density parity check (LDPC) codes were rediscovered [3]. LDPC codes are linear block codes that use iterative decoding and were first designed by Gallager [4] in 1962, but because of limited computing and implementation power at that time, they were almost forgotten.

Gallager designed the Belief Propagation (BP) algorithm for decoding of LDPC codes [4]. Decoding complexity is the major issue associated with implementing the BP algorithm. The Min-Sum (MS) algorithm [5] was first introduced by Tanner [6]. In [6], MS was clearly discussed as the algorithm that converges to maximum likelihood codeword estimation on cycle-free graphs. Then it was used by Fossorier [7] as an approximation of BP in log-likelihood ratio (LLR) domain. The

MS algorithm is a less complex algorithm compared to BP. It also performs worse than the BP algorithm.

The MS algorithm is a suitable algorithm in terms of digital and analog implementations. In the last few years, many researchers have worked on modifying the digital MS algorithm to improve its performance [8-10].

In terms of implementation, after Turbo codes proved their significant performance, research has been performed to implement iterative decoders. Turbo codes were the first iterative decoders that were implemented using digital implementation [11] and then analog implementation [12]. After that, the first digital LDPC chip using BP was implemented in [13] and proved the ability of the LDPC codes to achieve high coding gain compared to Turbo codes. In implementing the BP algorithm, to reduce the hardware complexity, various partially parallel structures are designed and implemented on FPGA [14, 15].

On the other hand, since the MS algorithm has much lower complexity compared to BP, many researchers used MS and its modified versions in implementing the LDPC decoder. In [16, 17] a modified version of MS is designed using partially parallel architecture on a FPGA [16] and using fully parallel architecture on chip [17]. In [18], the modified MS proposed in [9-10] is implemented on chip using semi-parallel architecture. And in [19], a modified version of MS is implemented on a FPGA using a bit-serial serial architecture

An analog MS LDPC decoder is designed in [20]. Then in [21], the dynamics of analog decoding for LDPC codes is analyzed. It is shown in [21] that the dynamics of analog decoding can be approximated as the application of

successive relaxation (SR) method to iterative decoding. In this thesis, we apply (SR) to the quantized version of the MS algorithm. Our simulation results show that the new modified algorithm performs better than the previous modified MS algorithms, with no increase in the average number of iterations.

We also implement the MS algorithm and its modified versions on a Field programmable gate array (FPGA). We make changes to current designs to improve the performance of the implemented design; also we design a custom circuit for the SR-MS algorithm to reduce circuit complexity and utilization. Then we test the implemented design and verify the functionality of the design.

1.2 Thesis organization

Chapter 2 gives a background review on iterative decoding and BP and MS algorithms.

In Chapter 3, we analyze the MS algorithm and its modifications. First, the modifications proposed in [10] and [9] on the MS algorithm are presented. Then the successive relaxation (SR) is introduced. The SR is then applied to the MS algorithm and effects of clipping and quantization are studied. Finally, the convergence speed and the histogram for iteration numbers for MS and its modifications are discussed and presented.

In chapter 4, a design methodology for the MS algorithm and its modifications on a field programmable gate (FPGA) is presented. The design and test programs are done in Verilog hardware description language and in the Xilinx integrated software environment (ISE) based on a Virtex-II pro FPGA. The test circuits are designed and finally the test results are presented and discussed.

The results and the conclusions of this thesis are summarized in chapter 5 and ideas for future works are discussed.

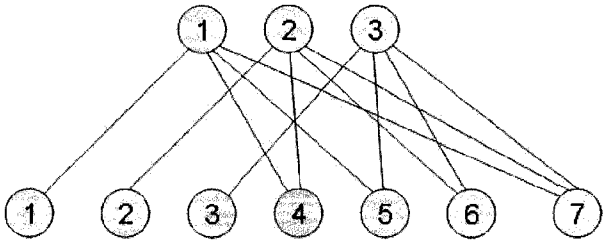
Chapter 2 LDPC codes and Decoding Algorithms

2.1 LDPC codes

A low-density-parity-check (LDPC) code is a linear block code defined by a sparse parity check matrix. Block codes are one of the major types of error correction codes [1]. In a block code C, the information sequence is segmented to message blocks of length k. These message blocks are encoded in blocks with block length n, and are called codewords. A linear block code C is defined by a parity check matrix H. The parity check matrix has n columns and m rows $(m = n-k)$. Any n-tuple v is a codeword in the code C if and only if $v.H^T = 0$. The code rate is defined as $R = k/n$. The LDPC code is called 'regular' if in its parity check matrix, the number of ones in each column is fixed, and the number of ones in each row is fixed. Otherwise, the LDPC code is called 'irregular'.

LDPC codes were first discovered by Gallager [4] in the early 1960s. His work was almost forgotten by coding researchers because of the complexity of his coding compared to other codes at that time. In 1981, Tanner [6] presented LDPC codes by a bipartite graph. This graph is now called the Tanner graph. There are two classes of nodes in a Tanner graph. The two classes are called variable nodes and check nodes. Nodes residing in different classes are connected through edges on the Tanner graph. Thus, in an (n,k) LDPC code, there are n variable nodes and $m = n-k$ check nodes. Check node j is connected with an edge to variable node i when h_{ji} in H is a '1'.

Figure 2-1 shows the Tanner graph and the H matrix of a (7, 4) code. In this code, we have 7 variable node and 3=7-4 check nodes. Variable nodes in the bottom are connected to check nodes in the top via edges. There are 12 edges in the Tanner graph of this code.



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

Figure 2-1 Tanner graph and H matrix of a linear block code (7, 4)

The sparse parity check matrix for LDPC codes is constructed using different algebraic and random methods. The encoding is done at the transmitter side using the generator matrix. The generator matrix is constructed from the parity check matrix. Decoding is done at the receiver side using iterative decoding based on the Tanner graph which is the major topic in this thesis. Decoding can be done using hard decision or soft decision decoding [1]. Soft decision decoding is more complex in the sense of calculations and gives better performance. Belief propagation (which is also known as sum-product) and min-sum are the two major types of soft decision decoding algorithms.

2.2 Channel Model

In this thesis, we consider binary-input additive white Gaussian noise (BIAWGN) channel. The input alphabet of the channel is $\{\pm 1\}$. The output of the channel is $y_i = x_i + n_i$, where x_i is the channel input at time *i* and $x_i \in \{\pm 1\}$ and n_i is the Gaussian random variable with zero-mean and variance σ^2.

The signal-to-noise ratio (SNR) is expressed using the term $\frac{E_b}{N_0}$, and is calculated using the following formula [22]:

$$\frac{E_b}{(\frac{N_0}{2})} = \frac{a}{2R\sigma^2}$$

In this formula, we assume the average energy per transmitted bit a is equal to 1. E_b is the average energy per information bit, $\frac{N_0}{2}$ is the power spectral density and σ^2 is the noise variance. Thus, for each SNR, we calculate the noise variance σ^2 using the following formula:

$$\sigma = \sqrt{\frac{1}{2 \times R \times (E_b / N_0)}}$$

$\frac{E_b}{N_0}$ is expressed in dB, and to convert it and use it in the above formula, we use the conversion $\frac{E_b}{N_0} = 10^{\frac{(\frac{E_b}{N_0})_{dB}}{10}}$.

2.3 Belief Propagation algorithm

The belief propagation algorithm works based on iterative decoding and message passing. The initial messages (vector y_i) come from the channel as real numbers instead of bits, and these messages are stored in variable nodes. Then, variable nodes send messages across the edges of the Tanner graph to the check nodes. The check nodes calculate the parity messages for each connected variable node based on the messages they received from other variable nodes connected to them. Check nodes send these parity messages back to the variable nodes along the edges of the Tanner graph. Next, variable nodes have to use the information they received from the check nodes to estimate the transmitted bit. Each variable node uses the messages received from all the connected check nodes to that variable node and estimates the transmitted bit. The sequence of bits in all variable nodes (vector $\hat{c}$) should form a codeword to be accepted as a corrected sequence. Hence, $\hat{c}.H^T = 0$ is checked. If this condition is satisfied, we have a codeword.

If the condition $\hat{c}.H^T = 0$ is not satisfied, variable nodes will create new messages for check nodes. Each variable node, for each connected check node, uses the information from the channel and messages from all other check nodes to create a new message for that check node. Then the variable nodes send the messages back to the check nodes. The iterative algorithm continues untill we reach the maximum number of iterations or $\hat{c}.H^T = 0$ is satisfied and we have a codeword.

The belief propagation algorithm which is also known as sum-product algorithm was originally introduced by [4]. The belief propagation algorithm is usually defined in probability and log-likelihood ratio domains.

In the probability domain, the majority of calculations are multiplications of probabilities. Probabilities are numbers between 0 and 1. Thus, these multiplications need a lot of resources in terms of implementations. Moreover, multiplying lots of probabilities can cause numerical stability issues. To overcome these problems, a general rule is to convert all calculations to the logarithm likelihood ratio (LLR) domain. All the multiplications are converted to additions in the logarithm domain. So in this domain, we have better numerical stability and low cost operation in variable nodes.

2.3.1 Belief Propagation algorithm in log-likelihood-ratio (LLR) domain

In the LLR domain [23], the logarithm of the ratio of the messages are calculated and passed along the Tanner graph. To decode bit c_i, we are interested in calculating the probability of error $P_i = P(c_i = 1 \mid y_i)$. This is the error probability assuming the all-zero vector is transmitted. To be able to make this assumption both the channel and the decoder have to be symmetric. The belief propagation algorithm in LLR domain has this property [24]. y_i denotes the message from channel to variable node i. We define notation $L(q_{ij})$ as the

message from the variable node i to check node j. $L(r_{ji})$ denotes the message from check node j to variable node i.

Based on the iterative algorithm process, the following steps are followed in the BP algorithm [23].

Step 0: Initializing variable nodes with $L(q_{ij})$ based on the information from the channel

$$L(q_{ij}) = L(c_i) = \log \frac{P(c_i = 0 \mid y_i)}{P(c_i = 1 \mid y_i)} = \log \frac{(1 + e^{-2y_i/\sigma^2})^{-1}}{(1 + e^{2y_i/\sigma^2})^{-1}} = \frac{2}{\sigma^2} y_i$$

Step 1: Check nodes calculate their messages for variable nodes.

$$L(r_{ji}) = (\prod_{i' \in V_j \setminus i} \alpha_{i'j}) \phi(\sum_{i' \in V_j \setminus i} \phi(\beta_{i'j}))$$

$$\alpha_{i'j} = sign(L(q_{i'j}))$$

$$\beta_{i'j} = |L(q_{i'j})|$$

$$\phi(x) = -\log(\tanh(\frac{x}{2})) = \log \frac{e^x + 1}{e^x - 1}$$

Step 2: Variable nodes calculate the digital estimate $\hat{c}_i$ of the current received sequence as follows:

$$L(Q_i) = L(c_i) + \sum_{j \in C_i} L(r_{ji})$$

$$\hat{c}_i = \begin{cases} 1 & if \quad L(Q_i) < 0 \\ 0 & otherwise \end{cases}$$

Step 3: In this step, exit conditions are checked. There are two exit conditions. First condition is if $\hat{c}.H^T = 0$, we have a codeword and the algorithm finishes

successfully. The second exit condition is if we have reached the maximum number of iterations. In this condition, the algorithm finishes but fails to correct the received sequence.

If neither of the two exit functions are satisfied, we continue the algorithm and we move to the next step.

Step 4: Variable nodes create new messages for check nodes as follows:

$$L(q_{ij}) = L(c_i) + \sum_{j' \in C_i \setminus j} L(r_{j'i})$$

Now these messages are sent to the check nodes and step 1 is repeated untill one of the exit conditions are met.

2.4 Min-Sum algorithm

In the Min-Sum algorithm [6, 7, 23], there is no change in variable node calculations and we use the message passing algorithm for decoding. The Min-Sum algorithm was first described by Tanner [6] and was used by Fossorier [7] as an approximation of BP in (LLR) domain.

We use the same notations used in BP-LLR algorithm in describing the steps in the Min-Sum algorithm. The Min-Sum algorithm has the following steps:

Step 0: The initial LLR values from the channel are normalized by $\frac{2}{\sigma^2}$:

$$L(q_{ij}) = L(c_i) = y_i$$

Step 1: Check nodes calculate their messages for variable nodes.

$$L(r_{ji}) = (\prod_{i' \in V_j \backslash i} \alpha_{i'j}).\min_{i' \in V_j \backslash i}(\beta_{i'j})$$

$$\alpha_{i'j} = sign(L(q_{i'j}))$$

$$\beta_{i'j} = \left|L(q_{i'j})\right|$$

Step 2: Variable nodes calculate the digital estimate $\hat{c}_i$ of the current received sequence as follows:

$$L(Q_i) = L(c_i) + \sum_{j \in C_i} L(r_{ji})$$

$$\hat{c}_i = \begin{cases} 1 & if \quad L(Q_i) < 0 \\ 0 & otherwise \end{cases}$$

Step 3: In this step, exit conditions are checked. There are two exit conditions. First condition is if $\hat{c}.H^T = 0$, we have a codeword and the algorithm finishes successfully. The second exit condition is if we have reached the maximum number of iterations. In this condition, the algorithm finishes but fails to correct the received sequence.

If neither of the two exit functions are satisfied, we continue the algorithm and we move to the next step.

Step 4: Variable nodes create new messages for check nodes as follows:

$$L(q_{ij}) = L(c_i) + \sum_{j' \in C_i \setminus j} L(r_{j'i})$$

Now these messages are sent to the check nodes and step 1 is repeated till one of the exit conditions are met.

As we can see, in the Min-Sum algorithm, in calculations in check nodes are simplified to finding the minimum values and they are now less complex compared to the BP algorithm.

Chapter 3 Modifications of Min-Sum and effects of quantization

3.1 Modifications of Min-Sum

In the MS algorithm, the hardware implementation of the algorithm is much simpler than the BP algorithm. However, the check nodes calculation approximation will cause performance degradation compared to the BP algorithm.

The BP algorithm is proven to have the best performance with infinite block length [5]. But in practical applications, having a code with infinite block length is impossible. In codes with finite block length, we have many short cycles. This will deteriorate the performance of the BP algorithm. So, in codes with finite block length, we can find an algorithm that performs even better than the BP. Hence, the goal is to modify the MS algorithm to improve its performance while keeping its simplicity. Different modifications are applied to the MS algorithm to improve its performance [8-10].

In [9] and [10], a correction method is introduced. In this method, a correction factor is applied to the magnitude of the output messages from the check nodes. In [9], this method is called offset BP-based algorithm. And in [10], it is called MS with unconditional correction (MS-UC).

As it is shown in section 2-4, the formula for calculating the variable node messages is:

$$L(r_{ji}) = (\prod_{i' \in V_j \backslash i} \alpha_{i'j}).\min_{i' \in V_j \backslash i}(\beta_{i'j})$$

After applying the correction, the formula will change to:

$$L(r_{ji}) = (\prod_{i' \in V_j \backslash i} \alpha_{i'j}).\max(\left|\min_{i' \in V_j \backslash i}(\beta_{i'j})\right| - y_c, 0)$$

Where y_c is called the correction factor.

In [10], the effect of clipping and quantization on the modified MS algorithm is also studied. To perform quantization, the incoming messages from the channel are first clipped symmetrically at a clipping threshold $[-c_{th}, c_{th}]$. The clipped values are then quantized uniformly in the clipping range $[-c_{th}, c_{th}]$. An optimum clipping threshold is found using extensive simulations for each code and it's shown that the clipping threshold is almost constant over a wide range of signal to noise ratios (SNR) and different quantization bits.

To study and compare the method used in this thesis with the result of the techniques used in [10], we perform simulation for an optimized irregular (1268, 456) code [25] and a regular (504,252) code [26]. For all the reported results in this thesis, the maximum number of iterations is chosen to be 200, and for each SNR value, we have 100 codeword errors.

For the (1268, 456) code and MS-UC algorithm, the optimal clipping threshold is found to be $c_{th} = 3$ [10]. Also, the optimal correction factor is found for different quantized versions of MS-UC. They are 1, 1, 2 and 3 for 4-bit, 5-bit, 6-bit and 7-

bit quantization, respectively. Figure 3-1 shows BER curves of (1268, 456) code for BP, MS, and quantized versions of MS-UC.

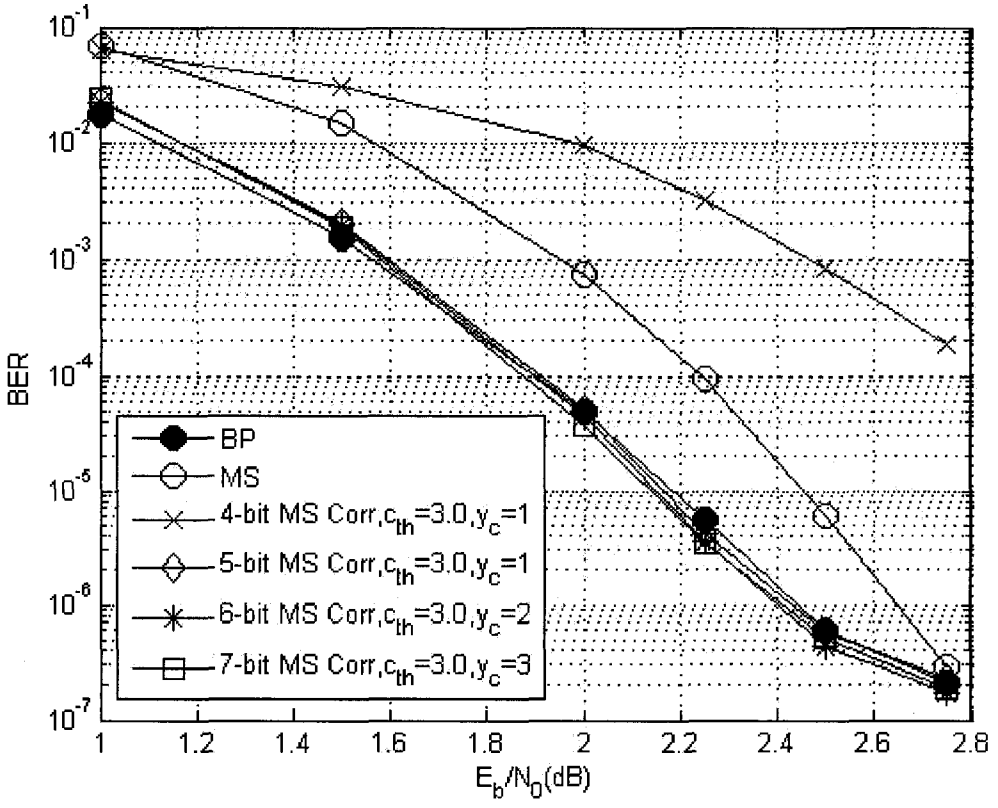


Figure 3-1: BER curves of BP, MS and quantized MS-UC for (1268, 456) code

The same simulations are performed for the (504, 252) code. For this code, the optimal clipping threshold is $c_{th} = 1.5$. And the optimal correction factors are 1, 1, 2 and 3 for 4-bit, 5-bit, 6-bit and 7-bit quantization. Figure 3-2 shows BER curves of (504, 252) code for BP, MS, and the quantized versions of MS-UC.

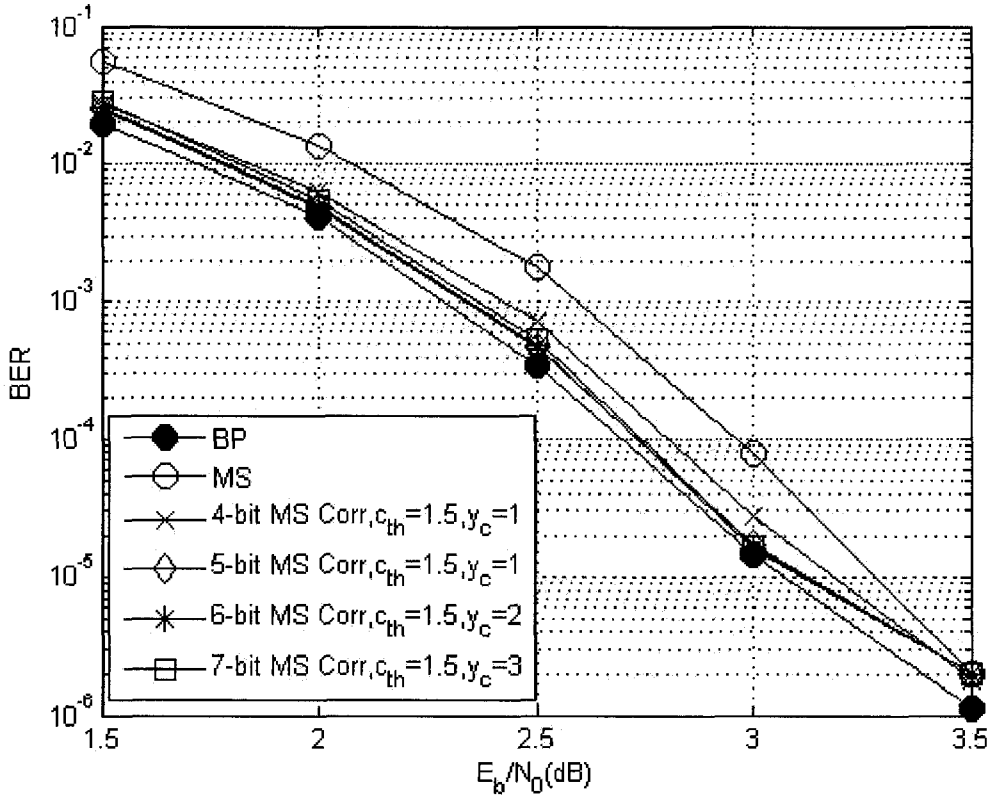


Figure 3-2: BER curves of BP, MS and quantized MS-UC for (504, 252) code

As we can see in Figures 3-1 and 3-2, with these modifications the performance of the MS algorithm gets much better even in the quantized version, and the gap between the performance of MS and BP algorithm is getting smaller. It is shown in [10] that the ideal MS-UC in some SNR slightly outperforms BP [10]. By term 'ideal' we mean floating point precision and 4-byte messages.

3.2 Successive Relaxation

Successive relaxation (SR) is a well-known iterative method used in solving systems of equations [27-29]. It is shown in [21] that the dynamics of analog

decoding can be approximated by the application of SR to the fixed point problem of iterative decoding.

In the conventional iterative decoding algorithm, in iteration $t+1$, the messages are created as a function of messages in iteration t. We can write the following equation:

$$X_{t+1} = h(X_t, X_0)$$

Where X_{t+1} is the messages created by the variable nodes at the iteration $t+1$, function $h(.)$ is a combination of operations performed in the variable nodes and the check nodes. X_0 is the received information from the channel, and X_t is the messages created by the variable nodes at iteration t.

In [21], the dynamics of an analog decoder is modeled by a system of first-order nonlinear differential equations. In an analog LDPC decoder, all of the messages are transferred between the variable nodes and the check nodes using analog continuous-time signals. Hence, in analog decoders there is no clock synchronization. So, the output of each module is propagated through the edges before reaching its steady state value. Also, different modules may have different delays. Using the Forward Euler method [30], these nonlinear equations and delays can be modeled as a simple difference equation as follows:

$$X_{t+1} = X_t + \beta(h(X_t, X_0) - X_t)$$

This equation is equivalent to the SR method to solve systems of equations [27-29]. The parameter β is called the "relaxation factor". if $\beta = 1$, SR will reduce to the conventional iterative algorithm. Based on [21], the optimal value of β that corresponds to minimum bit error rate (BER) is usually less than 1.

We can apply this method to the conventional iterative algorithm. We can see from the above equation that we should have memory in the variable nodes to keep the messages of the last iteration for calculating the messages for the current iteration. In the variable nodes, instead of sending the whole value of belief to check nodes, we move slightly along the line between the current value of messages and the newly created messages.

The SR method can be applied to both BP and MS algorithms in the LLR domain. In this thesis, we apply the SR method to the MS algorithm in the LLR domain (SR-MS-LLR). In (SR-MS-LLR), compared to (MS-LLR), there will be no changes of initial messages for variable nodes. Also, there will be no changes in the check node operations. The variable node messages are calculated by:

$$L(q_{ij}) := L(q_{ij}) + \beta\left(\left[L(c_i) + \sum_{j' \in C_i \setminus j} L(r_{j'i})\right] - L(q_{ij})\right)$$

The values of $L(q_{ij})$ on the right hand side of assignment are from the previous iteration.

To study the performance of (SR-MS-LLR), we perform simulations on (1268, 456) and (504, 252) codes. For each of these codes the optimum values of β is found using simulations. The optimal β is rather independent of SNR and is a function of the code and the maximum number of iterations. In SR, we are moving slightly along the line connecting the current value of messages and the newly created messages by a fraction of β. So we can also consider β as a measure of confidence about the created messages in the iterative process, This means that if for a code, we are more confident about the messages that are

being sent along the edges of the Tanner graph, we can take longer steps toward the next set of messages and the value of β is closer to 1. Also, regarding the maximum number of iterations, β will generally decrease with increasing the maximum number of iterations. Increasing the maximum number of iterations will give enough time to the algorithm to converge so the algorithm can now take smaller steps and β will decrease. For maximum number of iterations equal to 200, the optimum value of beta is found to be $\beta = 0.5$ for (504,252) code and $\beta = 0.7$ for (1268, 456) code. Figures 3-3 and 3-4 show the BER curves of (1268, 456) and (504, 252) codes, respectively. The BP and MS curves are also shown as reference.

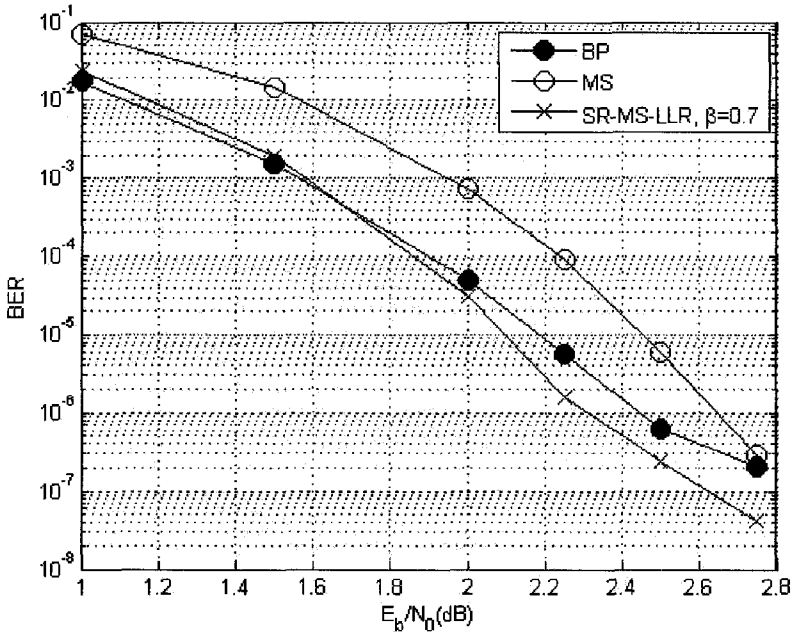


Figure 3-3 : BER curves of BP,MS, SR-MS-LLR for (1268, 456) code

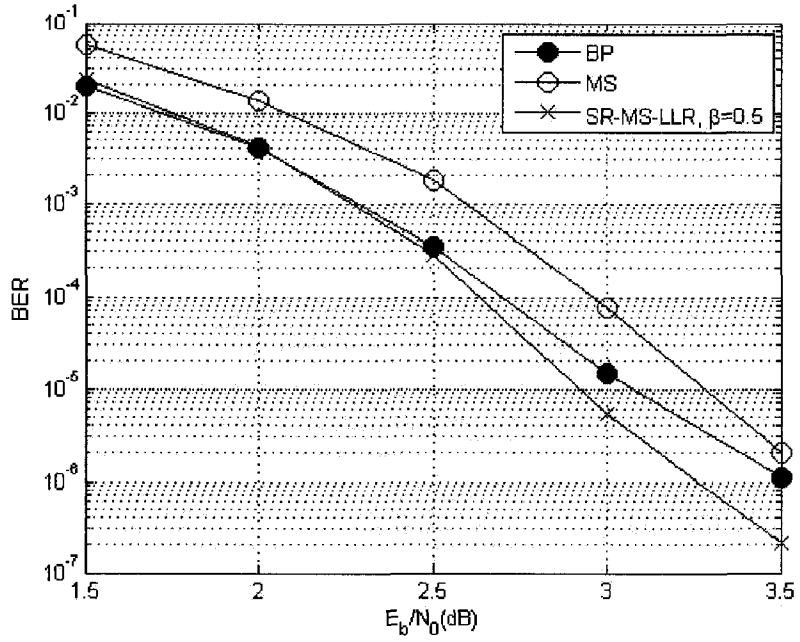


Figure 3-4 : BER curves of BP,MS, SR-MS-LLR for (504, 252) code

As can be seen, SR-MS is superior to MS and performs as well as BP in low SNR values and performs even better than BP in high SNR values.

3.3 Clipping and quantization on SR Min-Sum

The effects of clipping and quantization on the performance of MS-UC algorithm is studied in [10].

In this thesis, we study the effects of clipping and quantization on SR-MS-LLR algorithm.

Clipping is a mandatory step in any quantization scheme. The received real values from the channel are clipped symmetrically at a threshold $[-c_{th}, c_{th}]$. The clipped values are then quantized using uniform mid-tread quantization method

in the clipping range $[-c_{th}, c_{th}]$. For q quantization bits, we have $2^q - 1$ quantization intervals. Figure 3-5 shows an example of 3 bit uniform mid-tread quantization.

Clipping also is applied to the messages passed from variable nodes. If the output message of the variable node is bigger than $(2^{q-1} - 1)$ for positive numbers and smaller than $-(2^{q-1} - 1)$ for negative numbers, it is clipped to these limits and then sent to the check nodes.

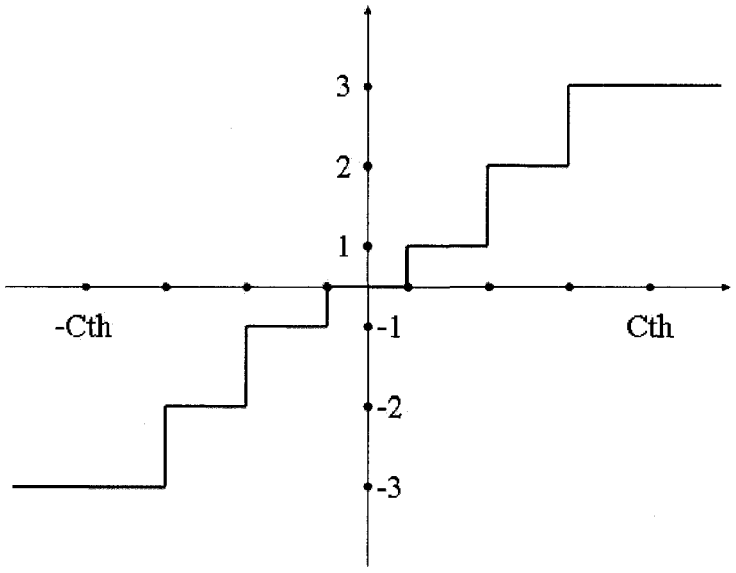


Figure 3-5: 3-bit uniform mid-tread quantization

The steps followed in SR-MS-LLR are as follows:

Step 0: The variable nodes are initialized with $L(q_{ij})$ based on the information received from the channel:

$$L(q_{ij}) = y_i$$

Then the values of $L(q_{ij})$ are clipped and quantized. Also, the values of $L(q_{ij})$ are stored in memory to be used on the next iteration. In other words, we need a memory element for each edge of the Tanner graph.

Step 1: The check nodes calculate the following messages (the same as the MS algorithm) for variable nodes:

$$L(r_{ji}) = (\prod_{i' \in V_j \setminus i} \alpha_{i'j}).\min_{i' \in V_j \setminus i}(\beta_{i'j})$$

$$\alpha_{i'j} = sign(L(q_{i'j}))$$

$$\beta_{i'j} = \left|L(q_{i'j})\right|$$

Step 2: The variable nodes calculate the digital estimate of current received sequence. (The same as BP and MS algorithms).

Step 3: The exit conditions are checked the same as BP and MS algorithms.

Step 4: The variable nodes create new messages for check nodes as follows:

$$L(q_{ij}) := L(q_{ij}) + \beta\left(\left[L(c_i) + \sum_{j' \in C_i \setminus j} L(r_{j'i})\right] - L(q_{ij})\right)$$

The values of $L(q_{ij})$ on the right hand side of the assignment come from the memory elements containing the values of $L(q_{ij})$ in the previous iteration.

Then the values of $L(q_{ij})$ are clipped and quantized. Also, the values of $L(q_{ij})$ are stored in memory to be used on the next iteration.

The iterative algorithm continues until the exit conditions are satisfied.

3.4 Simulation results and discussion

3.4.1 Performance simulations

The simulations are done on (1268, 456) and (504, 252) codes. First, for each number of quantization bits, it is investigated if the optimal c_{th} is fixed in different SNRs. Figure 3-6 (a)-(d) show our simulation results for each number of quantization bits on the (1268, 456) code in different SNR values.

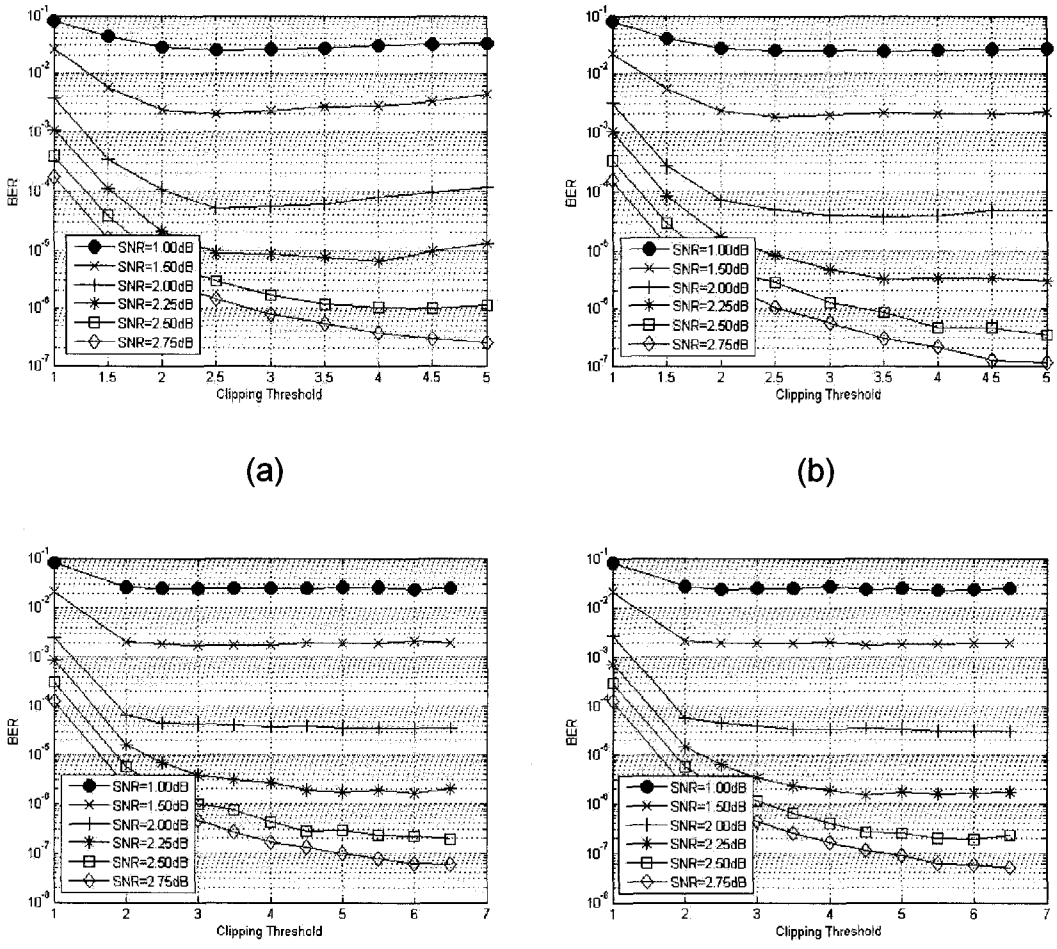
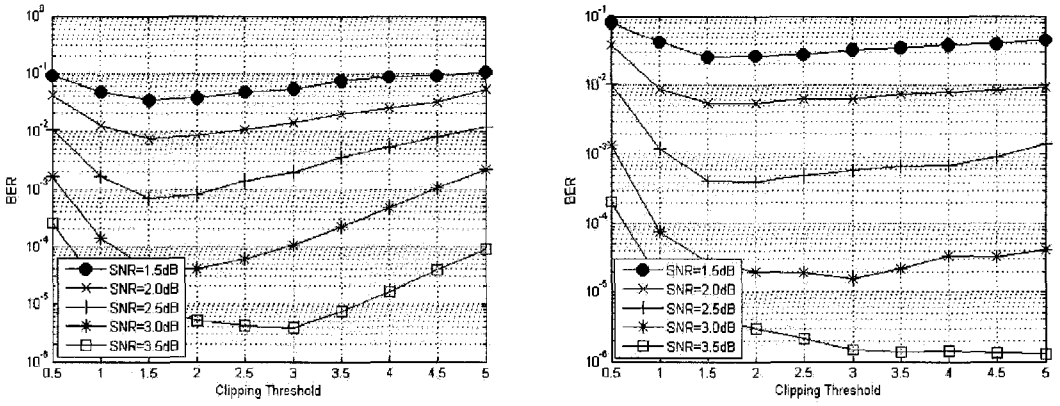
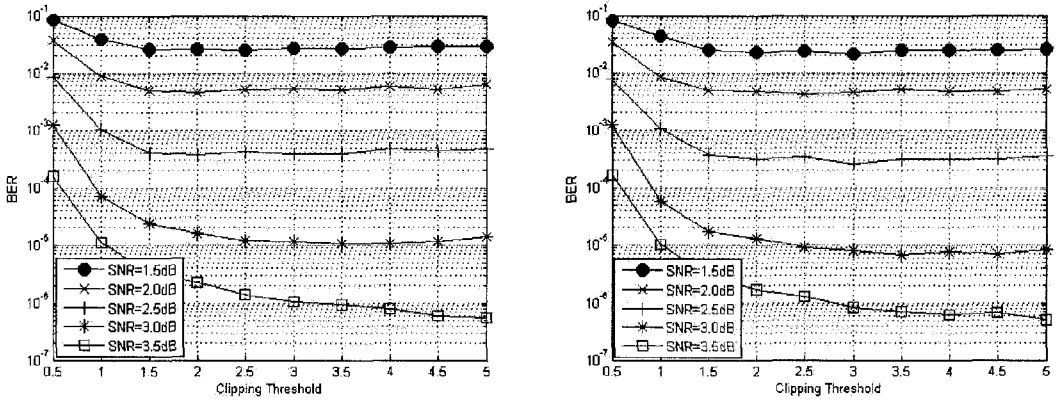


Figure 3-6: (a)-(d) Effect of c_{th} on BER performance of SR-MS for (1268, 456) code in different SNR values with 4-bit, 5-bit, 6-bit and 7-bit quantization schemes respectively

The same simulations are repeated for code (504, 252). The simulation results are shown in Figure 3-7 (a)-(d).



(a) (b)



(c) (d)

Figure 3-7: (a)-(d) Effect of c_{th} on BER performance of SR-MS for (504, 252) code in different SNR values with 4-bit, 5-bit, 6-bit and 7-bit quantization schemes respectively

In some curves in Figures 3-6 and 3-7, the curve is constant for a wide range of clipping thresholds after reaching the minimum. To have a visible Figure, the right hand side of those curves after the minimum is not shown. From the Figures above we will see that the optimal clipping threshold in general increases with

SNR. However, it can be assumed constant over a relatively small range of SNR values of interest. Also, it can be seen that optimal c_{th} for both codes increases with increasing the number of quantization bits. To show this effect more clearly, we represent the results in another format in Figures 3-8 and 3-9. Figure 3-8 (a)-(d), show our simulation results for different clipping thresholds on (1268, 456) code for each SNR with different number of quantization bits.

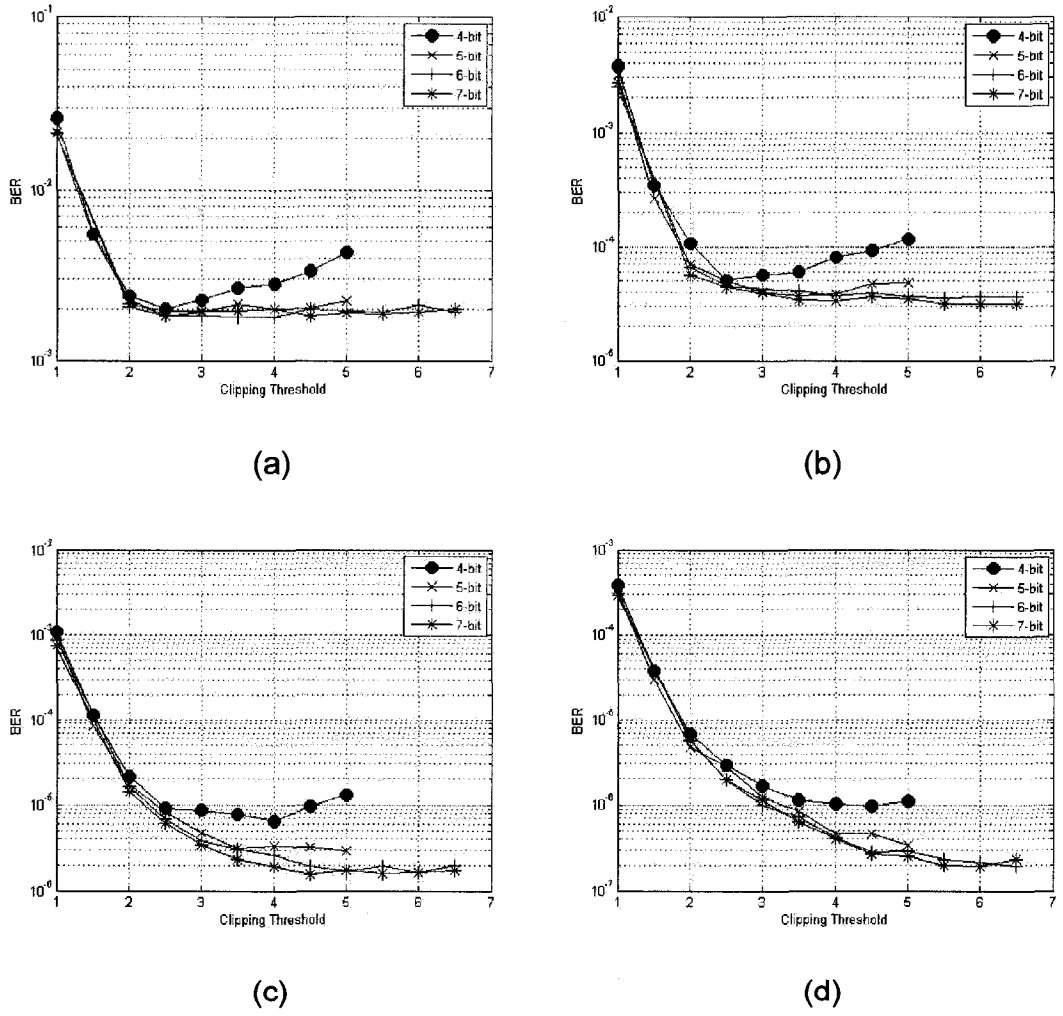


Figure 3-8: (a)-(d) Effect of c_{th} on BER performance of SR-MS for (1268, 456) code with different number of quantization bits at SNR=1.5dB, 2.0dB, 2.25dB, 2.5dB respectively

The same simulation results for code (504, 252) are shown in Figure 3-9 (a)-(d)

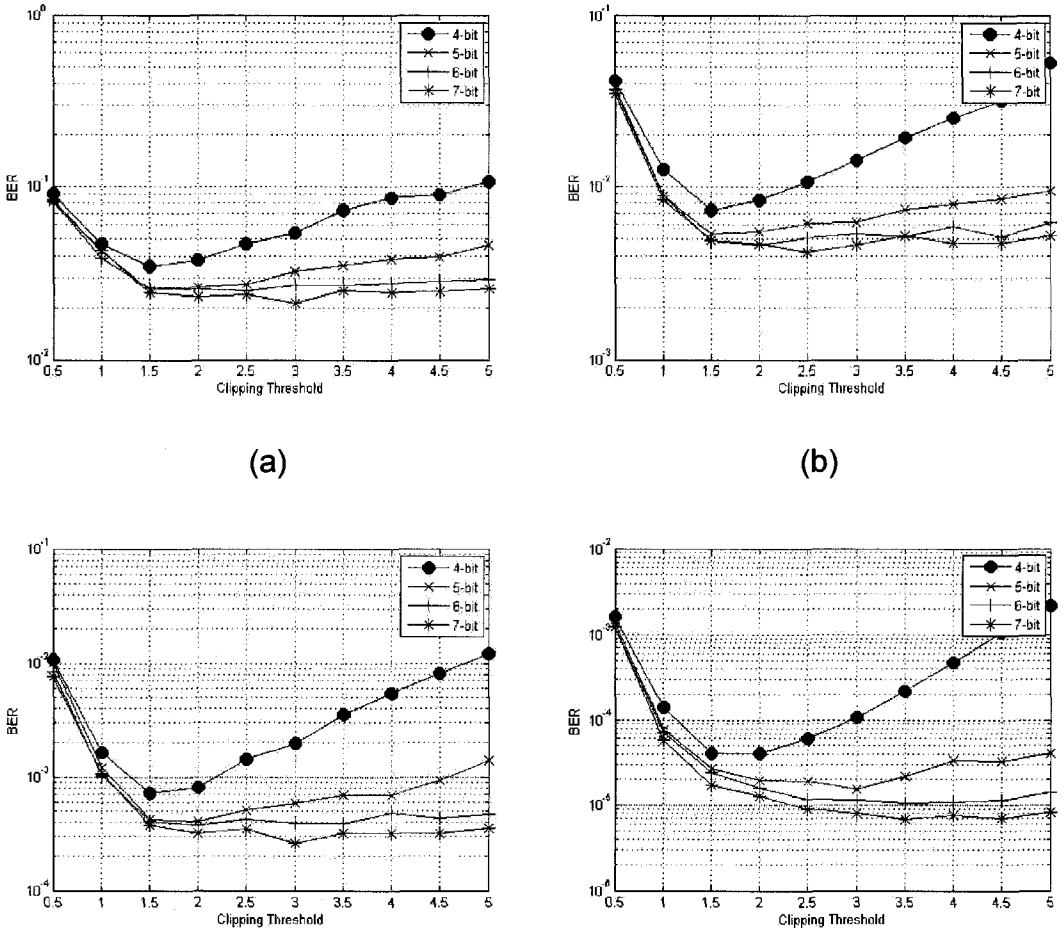


Figure 3-9: (a)-(d) Effect of c_{th} on BER performance of SR-MS for (504, 252) code with different number of quantization bits at SNR=1.5dB, 2.0dB, 2.5dB, 3.0dB respectively

The last two Figures clearly show that the optimum clipping threshold c_{th} is not fixed for different number of quantization bits.

Now we can choose the approximate optimum clipping threshold using Figures 3-6 to 3-9.

These values are listed in table 3-1.

Table 3-1: Approximate optimum c_{th} values for different number of quantization levels

Quantization bits	(504, 252) code	(1268, 456) code
4-bit	1.5	4
5-bit	3.0	5
6-bit	3.5	6.5
7-bit	4.0	6

Results from table 3-1 show that the optimal c_{th} is almost increasing with increasing the number of quantization bits. The only exception in this table is for (1268, 456) code, in which with 7- bit quantization, the optimal clipping threshold decreases by 0.5 from 6- bit.

If the clipping is done only on the initial values coming from the channel, the optimal clipping threshold should generally decrease with increasing SNR and number of quantization bits, but clipping is done on both the incoming messages from the channel and the messages sent from variable nodes. Thus, it increases with increasing the SNR and number of quantization bits. Clipping threshold is also a parameter of the code because different codes have different degrees for variable nodes. For variable nodes with higher degrees we would need a higher

clipping threshold as the messages generated at these variable nodes would be more reliable.

To compare the performance of the quantized version of SR-MS algorithm with other modified MS algorithms, the BER curve for two codes using different algorithms and different quantization levels are given in Figures 3-10 and 3-11. In these Figures, the BP and MS curves are also shown as reference.

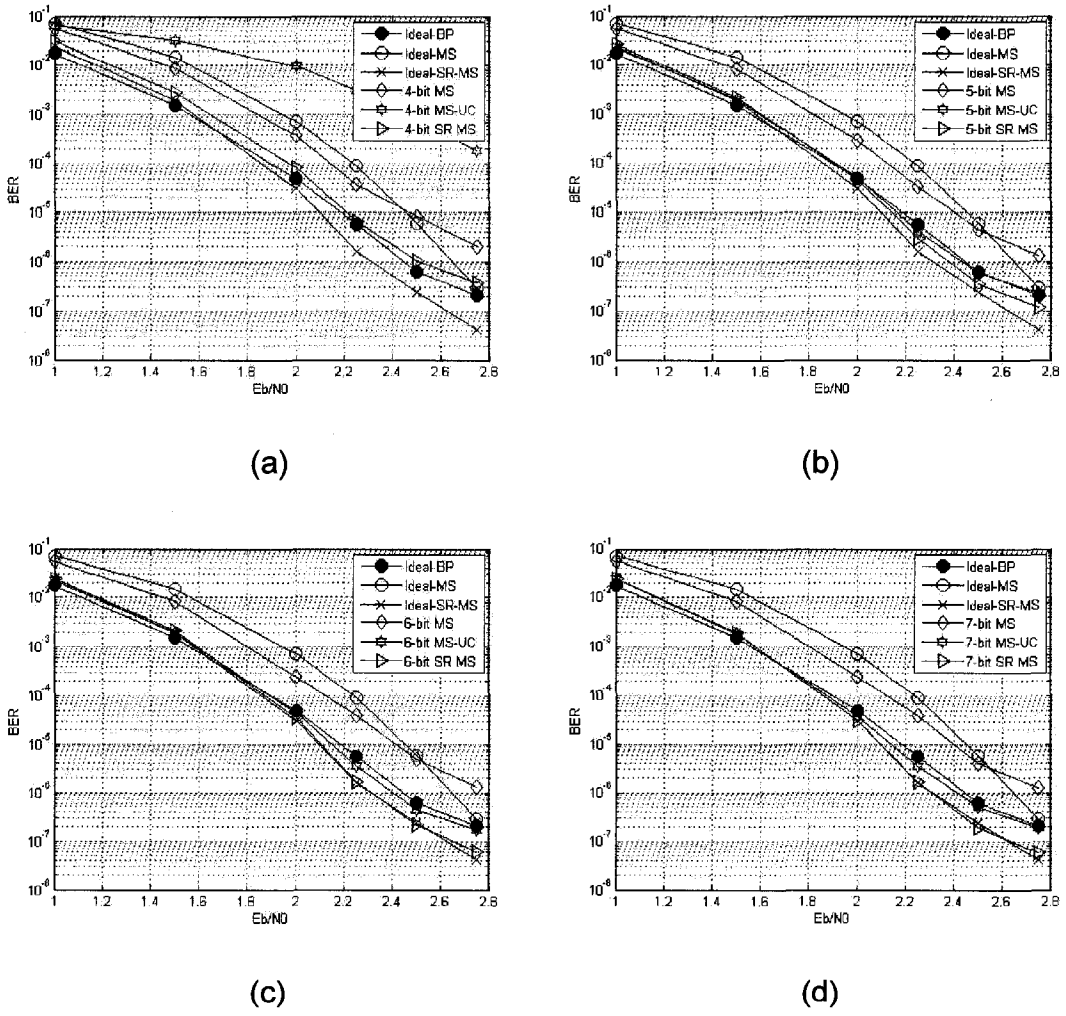


Figure 3-10: (a)-(d) BER curves of quantized SR-MS for (1268, 456) code with 4-bit, 5-bit, 6-bit and 7-bit quantization levels respectively

As can be seen in Figure 3-10 (a)-(d), the quantized SR-MS outperforms MS and MS-UC in all quantization levels. It even outperforms the ideal BP with 6-bit quantization at high SNRs. This can be seen in Figure 3-10 (c), where the SR-MS with 6-bit quantization outperforms the floating point BP by about 0.25 dB at $BER = 2 \times 10^{-7}$.

The same experiment is repeated for the (504,252) code and the results are shown in Figure 3-11 (a)-(d).

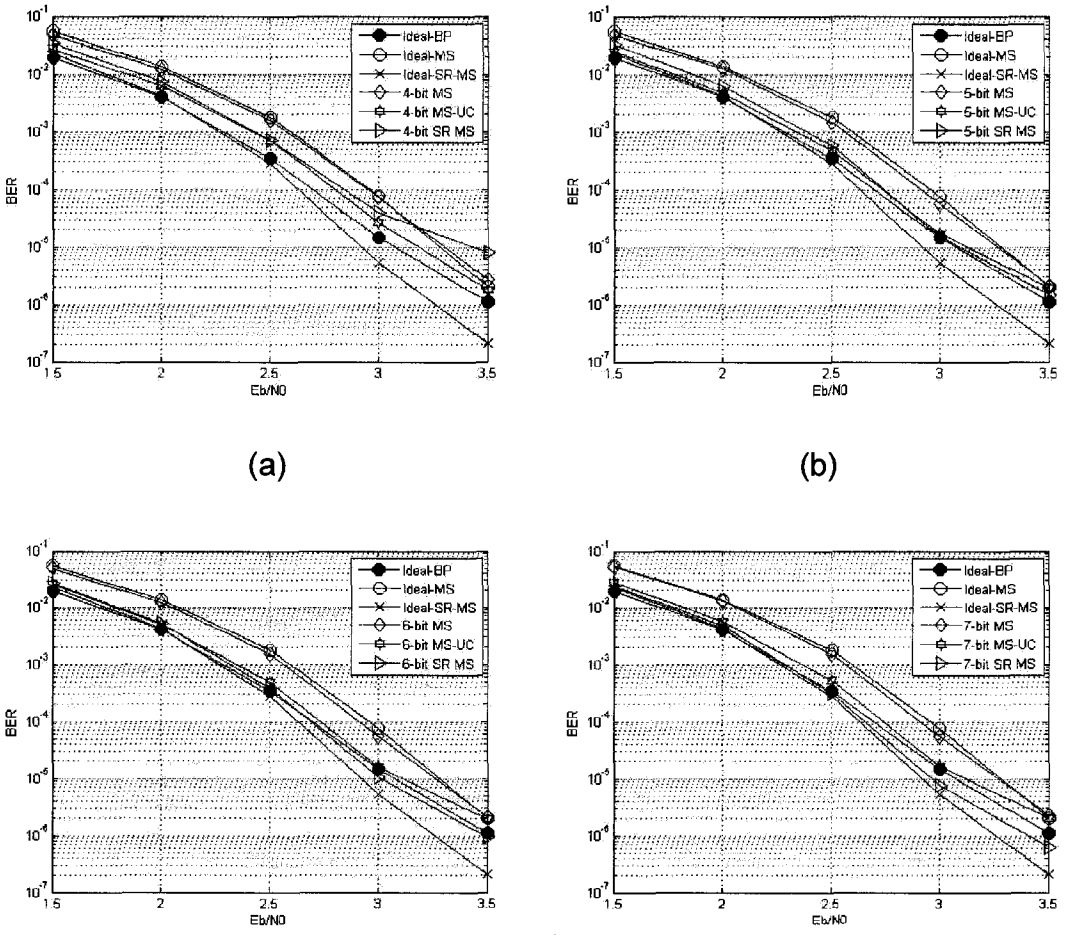


Figure 3-11: (a)-(d) BER curves of quantized SR-MS for (504, 252) code with 4-bit, 5-bit, 6-bit and 7-bit quantization levels respectively

The simulation results show that for the (504,252) code with 4-bit quantization, MS-UC outperforms the SR-MS. For 5-bit quantization, both algorithms perform almost the same. And for 6-bit and 7-bit quantization, SR-MS outperforms the MS-UC and the floating point BP. As an example, in Figure 3-11 (d), the SR-MS with 7-bit quantization outperforms the ideal BP by about 0.1 dB at $BER = 10^{-6}$.

3.4.2 Iteration distributions

SR-MS algorithm demonstrates superior performance compared to other modifications of the MS algorithm. It also outperforms the Ideal BP algorithm with a relatively small number of quantization bits. Here, we would like to investigate whether this increase in performance comes at any cost in terms of the number of iterations. We also calculate the average number of iterations for each decoding algorithm. The average number of iterations is an important factor in power consumption when we implement the algorithm. Figure 3-12 shows the histogram of the number of iterations for decoding the (1268, 456) code with different algorithms and 7-bit quantization in SNR=2.75dB.

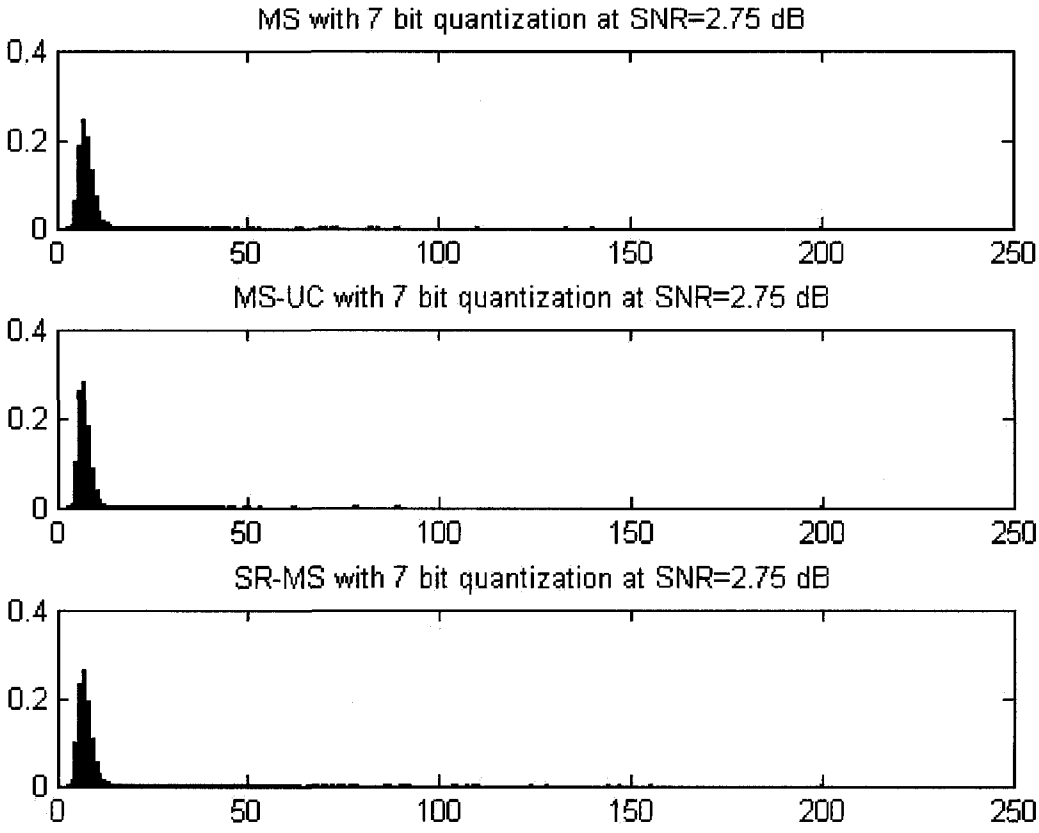


Figure 3-12: Distribution of iteration numbers for (1268, 456) code

The simulation results for (504, 252) code is performed at SNR=3.5dB and for 7-bit quantization. The simulation results are shown in Figure 3-13.

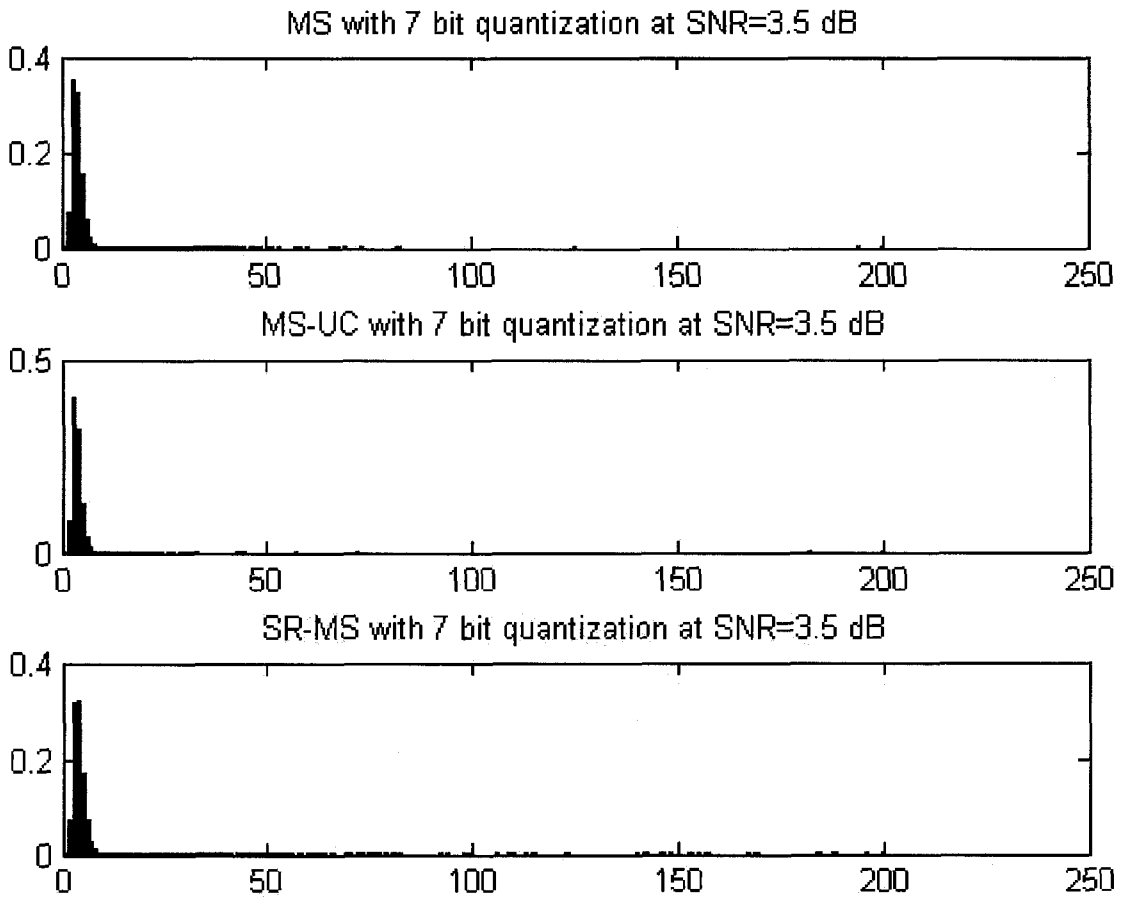


Figure 3-13: Distribution of iteration numbers for (504, 252) code

As we can see in Figures 3-12 and 3-13, there is no significant difference between the histograms of the three algorithms. In fact the average iteration number of the three algorithms for the (1268, 456) and (504, 252) codes are 7.8, 7.1, 7.3 and 3.9, 3.7, 4.0 respectively. We see the same trends in other SNR values. So the SR-MS has no significant effect on average decoding time.

Chapter 4 Digital implementation of Min-Sum Algorithm and its modifications

4.1 Overview

In this chapter, the design methodology for a parallel structured digital MS decoder and its modifications is presented. We use a field programmable gate array (FPGA) to implement the design. The language of programming is Verilog. Three decoding algorithms (MS, MS-UC and SR-MS) are designed and programmed into the FPGA. A control circuit is designed to control loading and unloading of data and signaling parameters inside the FPGA. A PC based test program is written to communicate with the FPGA, generate and send quantized input values perturbed by additive Gaussian noise into the FPGA and receive the decoded bit stream. The received bit stream is then compared to the randomly generated bit stream and the bit error rate (BER) is calculated.

The (504, 252) code is chosen for implementation. This code is a regular code with variable nodes of degree 3 and check nodes of degree 6. There are 1512 edges in the Tanner graph of this code.

4.2 Virtex-II Pro FPGA and Amirix Board

The Canadian Microelectronics Company (CMC) has provided system on chip research network (SOCRN) universities with a system prototyping environment: the Virtex-II Pro FPGA Prototyping Station. The Virtex-II Pro FPGA Prototyping station includes one AMIRIX AP1000 development board. This board has a XC2VP100 FPGA with large FPGA gate capacity (44000 logic slices) [31]. This board has two RS-232D serial ports to communicate with the FPGA.

4.3 Control system

The main idea is to design a control system to test the FPGA decoder. The control system consists of two components. The first component is in a PC equipped with MATLAB and a RS-232 port. The second component is inside the FPGA to provide RS-232 signaling, receive the incoming information and store them into memory, trigger decoder start and store the decoded bits, and finally transmit the decoded bits back to the PC. Based on the limitations on the FPGA, we choose 4-bit quantization for implementing the design. Expanding the design to more quantization bits would be straight forward if more space were available. Hence, 4-bit quantization is used for all the hardware designs in this thesis. As we will see, the proposed design methodology can be applied to other number of quantization bits.

4.3.1 FPGA control software

The top module of the FPGA is called "ms_504_252". The FPGA communicates with the PC through the serial port. Hence, the inputs to this module are clock (clk) and serial input (RxD). And the output is the serial output (TxD). For processing the input and output RS-232 signals, the signaling standards in [32] are used. Based on RS-232 standard, a control system is designed to get both input data and commands from the serial input data and send output messages and data via serial port. Table 4-1 shows the input commands and output algorithm that runs after processing each command. The source of this control program is listed in the appendix 1.

Table 4-1: FPGA control software

Input Command	Output Message	Description
X	RESET	- Reset all internal memory values to zero
L	LOAD	- Get all the input messages and store them into variable node registers. Input values are delimited by "." - Show "FULL" when all the input messages are received
R	READ	- Read the input messages from the variable node registers and send them back to the output delimited by "." - Show "DONE" when all the messages are sent to output
S	START	- Start decoding the input messages and store them into decoded memory - Show "END" when decoding is finished
O	OUTPUT	- Send decoded bits delimited by "." from variable nodes to the output - Show "DONE" when all the messages are sent to output

4.3.2 MATLAB control software

A MATLAB program generates a random sequence of bits with the length of 252 (this is the initial message block denoted with length k in section 2.1). This message block is encoded using the generator matrix constructed from the parity check matrix of the (504, 252) regular code. The result is the encoded codeword with block length n. The codeword is transmitted based on the channel model introduced in section 2.2. Then based on the considered SNR level, AWGN noise is added to the codeword. Then the sequence is clipped and quantized. A load command 'L' is sent to the FPGA to initiate the loading process. Then all the 504 input messages delimited with "." are sent to the serial port. Then an 'S' command is initiated and program waits to get 'END' back from the FPGA. An 'O' is sent to FPGA to get all the decoded bits. The decoded bits are compared to the original bits to see if we have any errors. This process is repeated until we get 100 codeword errors. Then the BER is calculated and reported. The source of this MATLAB program is listed in the appendix 2.

4.4 Implementation of Min-Sum

4.4.1 Datapath and module connections

As mentioned in section 4.1, the (504, 252) code is used in this design. The datapath of the decoder is the same datapath used in [13]. There are 2 separate datapaths for sending messages from variable nodes to check nodes and from

check nodes back to variable nodes. Using only one datapath for both messages will require more logic overhead, buffers and control signal distribution [13]. Therefore, a register is associated with each variable node to ensure correct synchronous execution of message passing algorithm [13].

The (504, 252) code has 1512 edges. So, considering 4-bit quantization, we will need 1512*2*4= 12096 wires to connect 504 variable nodes of degree 3 to 252 check nodes of degree 6. The module that connects all the nodes is called the "decoder" module. In this module, all the wires are defined and nodes are connected to each other. As an example of these connections, the Verilog definition of the first two variable nodes and the first two check nodes are listed below:

```
  check_node6 chknode1 (out1, in1, out2, in2, out3, in3, out4, in4, out5, in5, out6, in6);
  check_node6 chknode2 (out7, in7, out8, in8, out9, in9, out10, in10, out11, in11, out12,
in12);

  variable_node3 varnode1 (out181, in181, out301, in301, out1147, in1147, dec_scan_out0,
rec_scan_in0, pkt_start, clk, nrst, done);
  variable_node3 varnode2 (out811, in811, out1195, in1195, out1255, in1255, dec_scan_out1,
rec_scan_in1, pkt_start, clk, nrst, done);
```

The signals starting with 'in' denote the 4-bit input connections to the node and signals starting with 'out' denote the 4-bit output connection from the node. The 'rec_scan_in' signals denote the 4-bit information from the channel and the 'dec_scan_out' signals denote the 1-bit decoded output from the variable node. The input 'packet_start' signal denotes the control signal for the first iteration. The input clock and reset signals are denoted with 'clk' and 'nrst'.

A simple method to number the connections is proposed. The check nodes connections are simply numbered from 1 to the number of edges. To number the variable node connections, we can use the Alist format [26]. The Alist format lists the position of each '1' in the parity check matrix for all columns. Each column in the parity check matrix represents a variable node in the Tanner graph. We write a C program that reads the Alist format and generates the connections for each variable node.

4.4.2 Variable Node Architecture

The architecture used for the variable nodes with degree 3 is derived also from [13]. However, there are some essential changes applied to the architecture used in [13]. This architecture is shown in Figure 4-1

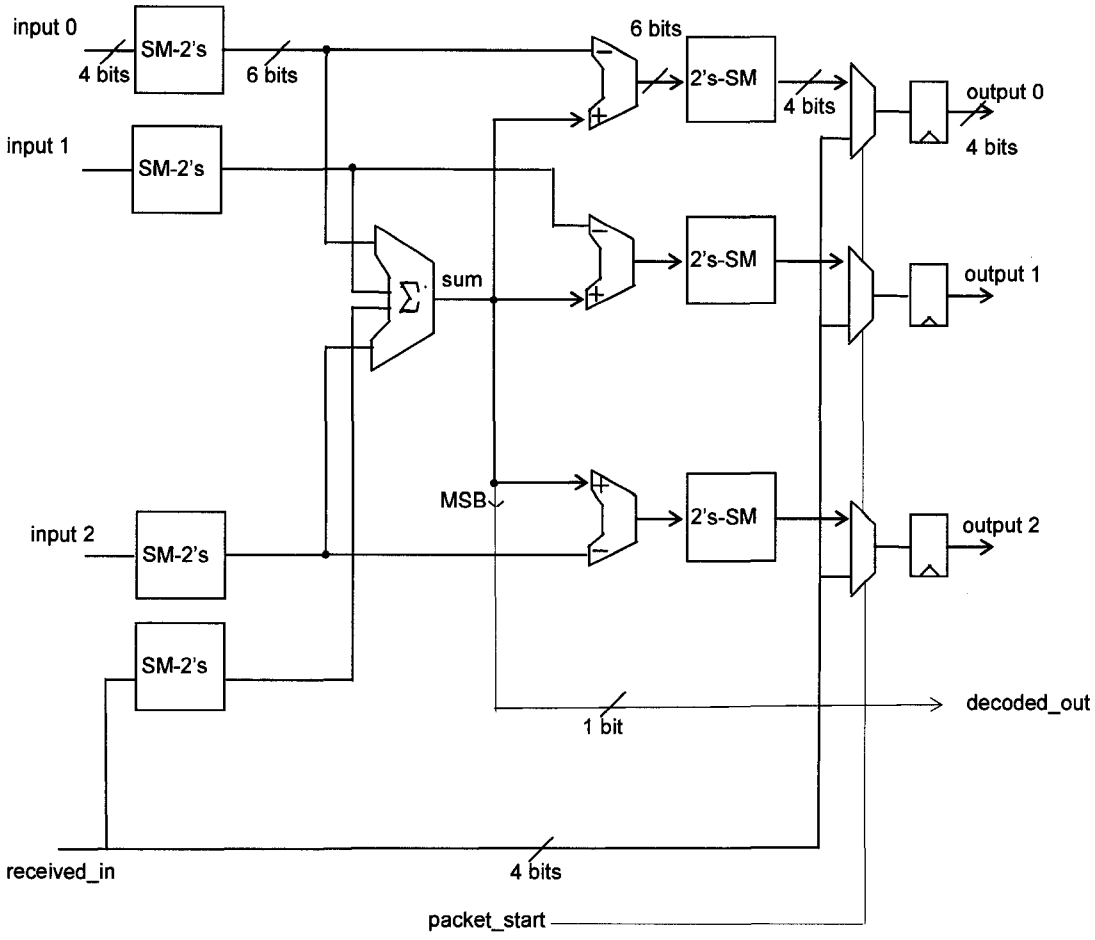


Figure 4-1: The architecture of variable nodes of degree 3 for MS

The inputs from the check nodes are called "input 0", "input 1" and "input 2". The input from the channel is called "received_in". The "packet_start" signal is set in the first iteration. So in the first iteration, the "received_in" value from the channel is copied to the output registers which are connected to check nodes. The "packet_start" signal is reset in the next iterations. After the first iteration, the inputs from the check nodes are first converted to 2's complement. The messages transferred between the variable nodes and the check nodes are in Sign-Magnitude (SM) domain. In a 4-bit quantized decoder, 1 bit is used for the

sign and 3 bits are used for the magnitude. To perform addition efficiently we have to change the representation domain to 2's complement. Then, all the 2's complement converted messages from the check nodes and the channel message are summed together. The sign (MSB) of this sum is the current estimate of the decoded bit at each variable node. Then the message for each check node is prepared by subtracting the input message of that check node from the total sum. These messages are converted from 2's complement to sign-magnitude domain and stored in the registers to be sent to check nodes.

There is a main difference between this design and the design presented in [13] that will improve the performance. The difference is that in [13], the 4-bit sign-magnitude messages are converted to 4-bit 2's complement messages. These messages are added together using a full adder. The variable nodes are of degree 3, so in the variable node we have 4, 4-bit inputs. We can easily have an overflow if we use only 4-bit for the messages inside the adder. And this overflow will change the output messages for the check nodes. For example, if we have '4' in input0, '3' in input1, '2' in input2 and '1' as the value from the channel, the total sum of these values will be '10'. With 4-bit representation, this will cause an overflow and the total value will be clipped to '7'. Then to create the messages for the check nodes, the input messages will be subtracted from the total sum. In this case, we will have 7-4=3 for 'output0', 7-3=4 for 'output1' and 7-2=5 for 'output2'. Now if we suppose we didn't have the overflow, using '10' as the total sum , we had 10-4=6 for 'output0', 10-3=7 for 'output1' and 10-2=8 and after clipping '7' for 'output2'. To prevent this, we calculate the maximum number

of bits needed inside the adder module by assuming the maximum input values for the inputs. In our design, we have 4 inputs with maximum value of '7'. So the maximum total sum would be '28' which is a 6-bit signed number. Messages are converted from 4-bit sign-magnitude to 6-bit 2's complement and passed to the full adder. Thus, the messages are clipped to $\pm(2^{q-1}-1)$ (q is the number of quantization bits) when converting them back from 6-bit 2's complement domain to 4-bit sign-magnitude domain. In this design, these messages are clipped to $\pm(2^{4-1}-1)=\pm 7$ and converted back to sign-magnitude presentation.

The variable node module is divided into the following 4 modules:

4.4.2.1: master_control:

The input to this module is the 'start' signal that is invoked when 'S' command is received. This module creates 'pkt_start' for the first iteration, and using a counter that counts down from the maximum number of iterations, generates the 'done' signal. The 'done' signal is used to stop decoding and storing the output bits into the output registers. In this design, we don't check the $v.H^T=0$ exit condition and we go to the maximum number of iterations on each decoding cycle. This will increase the power consumption and would affect the performance slightly. However, the circuit to check the $v.H^T=0$ condition is a complex circuit and will increase the size of the design in a way that we couldn't fit that in the 4-bit design.

4.4.2.2: sm2tc:

This module converts the 4-bit input sign magnitude message to a 6-bit 2's complement. The Verilog code for the conversion is given below:

```
assign out = in[3] ? {1'b1, 1'b1, in[3], (~in[2:0]+1)} : {2'b0, in};
```

4.4.2.3: variable_adder:

This module does all the additions and subtractions for creating the output bit and messages for check nodes. Figure 4-2 shows the architecture of the "variable_adder" module.

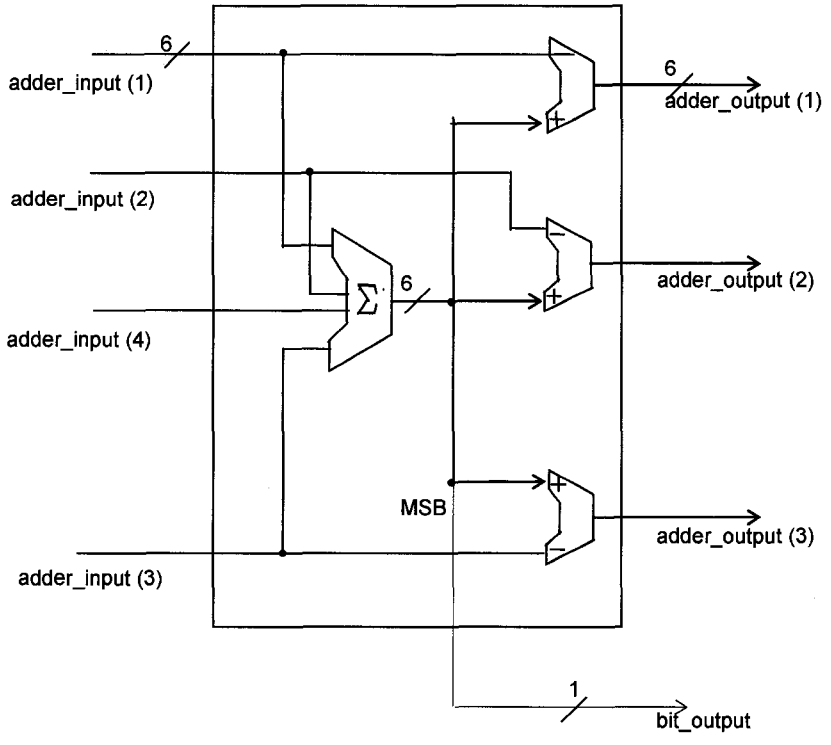


Figure 4-2: the architecture of the "variable_adder" module

4.4.2.4 tc2sm:

This module maps the 6-bit input 2's complement message to the output 4-bit sign-magnitude message. This is done through a lookup table. And this lookup table can be made for any levels of quantization. Table 4-2 shows the lookup table for 4-bit quantization.

Table 4-2 Lookup table mapping for 4-bit quantization

2's Complement	Sign-Magnitude
$(000000)_b$	$(0000)_b$= 0
$(000001)_b$	$(0001)_b$= 1
$(000010)_b$	$(0010)_b$= 2
$(000011)_b$	$(0011)_b$= 3
$(000100)_b$	$(0100)_b$= 4
$(000101)_b$	$(0101)_b$= 5
$(000110)_b$	$(0110)_b$= 6
$(000111)_b$	$(0111)_b$= 7
$(001xxx)_b$	$(0111)_b$= 7
$(01xxxx)_b$	$(0111)_b$= 7
$(111111)_b$	$(1001)_b$= -1
$(111110)_b$	$(1010)_b$= -2
$(111101)_b$	$(1011)_b$= -3
$(111100)_b$	$(1100)_b$= -4
$(111011)_b$	$(1101)_b$= -5
$(111010)_b$	$(1110)_b$= -6
$(11100x)_b$	$(1111)_b$= -7
$(110xxx)_b$	$(1111)_b$= -7
$(10xxxx)_b$	$(1111)_b$= -7

The sources of these modules are listed in the appendix 3.

4.4.3 Check Node Architecture

The check node architecture consists of two components. The messages from the variable nodes have 1-bit sign and 3-bit magnitude. The sign bits are XOR-ed together, and then the sign of the outgoing message on each edge of the Tanner graph is obtained as the XOR of the sign of the incoming variable message on that edge and the XOR of the signs of all the incoming messages. Figure 4-3 shows the architecture of the degree 6 sign update circuit.

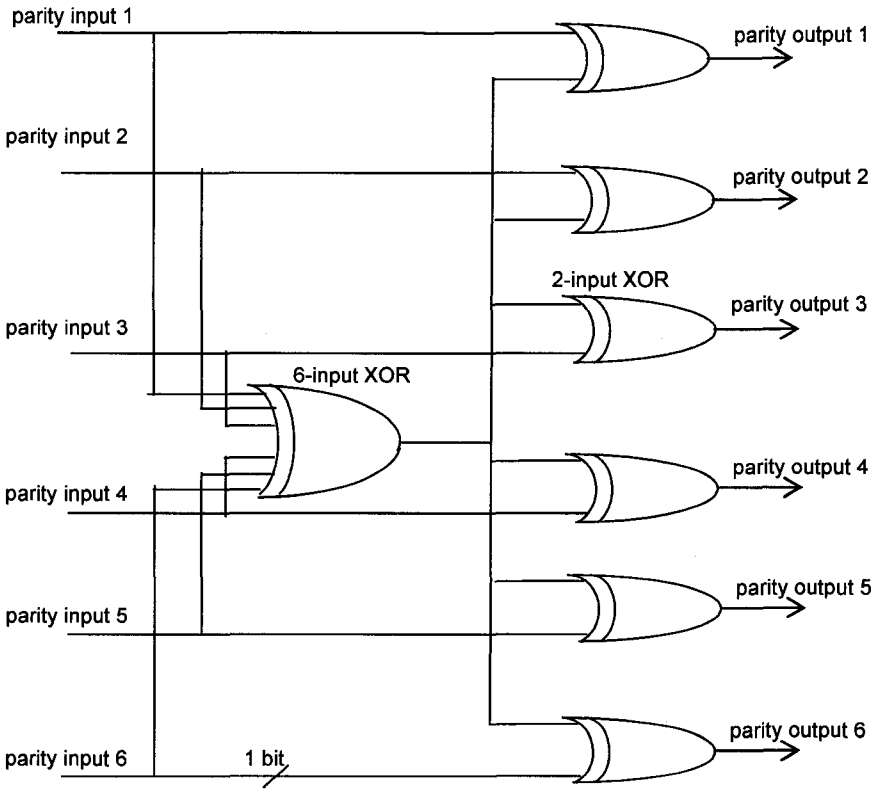


Figure 4-3- Architecture of sign update for a check node with degree 6

To calculate the magnitude of the messages in check nodes, 12 2-input minimum functions are used. The architecture of magnitude updates for check nodes with degree 6 is shown in Figure 4-4.

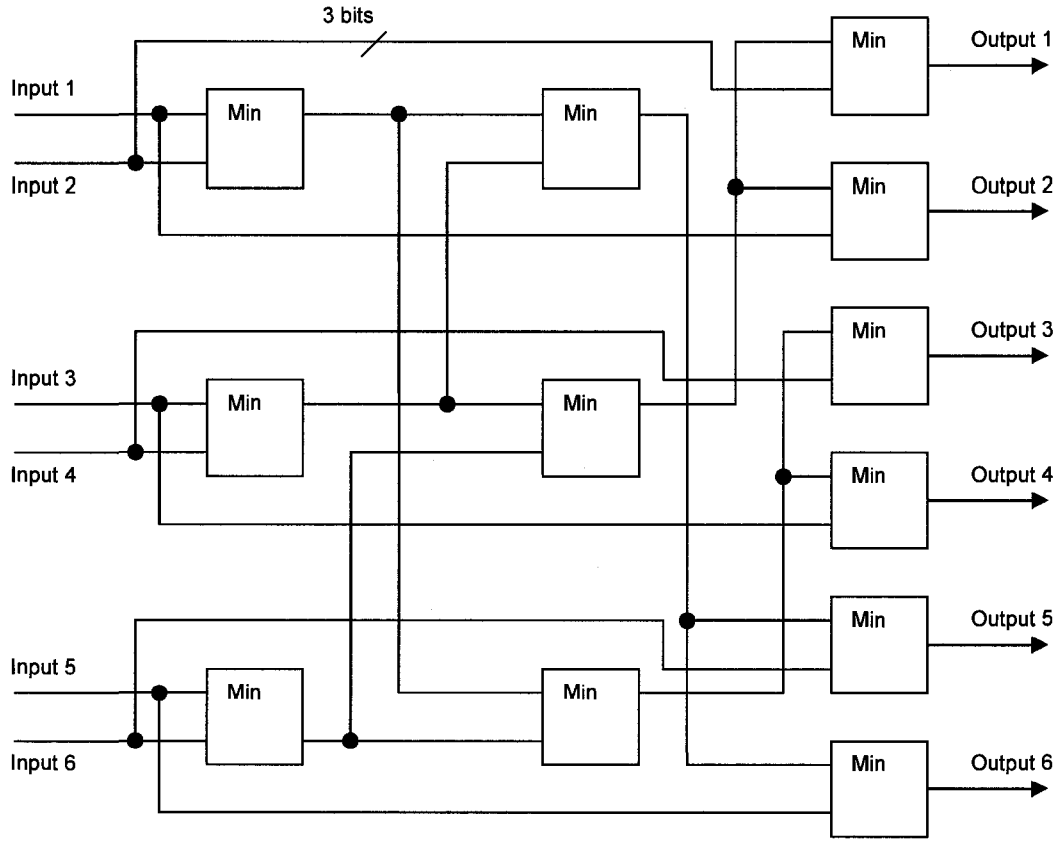


Figure 4-4- Architecture of magnitude update for check node with degree 6

As we can see in Figure 4-3 and 4-4, the check node architecture is all combinational logic. Check nodes update their value and send them back to variable nodes between clock cycles. So one whole iterations takes place in one clock cycle. The Verilog implementation of min function is presented below:

```
assign min12 = (min1 < min2) ? min1 : min2;
```

The source code of the check node with degree 6 used in this thesis is given in the appendix 4.

4.5 Implementation of Min-Sum with unconditional Correction

As mentioned in section 3-1, for the MS-UC algorithm, an optimum correction factor is subtracted from the magnitude of all the messages sent from the check nodes to variable nodes. So, there is no change in the variable node's structure. The only change would be in the check node's structure. The following Verilog code will be added to check node's code:

```
assign out1[2:0] = (min1 >= corr )? (min1-corr) : 0;
```

The parameter "corr" is the optimized correction factor. In terms of circuit design, the only complexity added for the MS-UC FPGA design is 252 (3-bit) unsigned adders.

4.6 Implementation of Min-Sum with Successive Relaxation

In the SR-MS algorithm, check nodes architectures are not changed. And the only change is in the variable nodes architecture. From section 3-2, the variable nodes update functions is as follows:

$$L(q_{ij}) := L(q_{ij}) + \beta\left(\left[L(c_i) + \sum_{j' \in C_i \setminus j} L(r_{j'i})\right] - L(q_{ij})\right)$$

Also, the optimum value of beta for this code is $\beta = 0.5$. The goal is to modify the variable nodes architecture to add minimum complexity to the circuit. Using multipliers and dividers will add very big modules to each variable node and will make the implementation of the code on the existing FPGA impossible. Since the constant β is a fixed number, instead of using complex multipliers and dividers we can use custom designed circuits with the help of shift registers or lookup tables to calculate the output messages.

The changes are made inside the "variable_adder" module. The registers we used to synchronize the message passing algorithm always have the values of messages of the variable nodes from the last iteration. So these registers are now used as extra inputs to the "variable_adder" module. Inside the "variable_adder" module, all the messages are in 2's complement format. This is considered when implementing the multiplier. The new "variable_adder" architecture is shown in Figure 4-5.

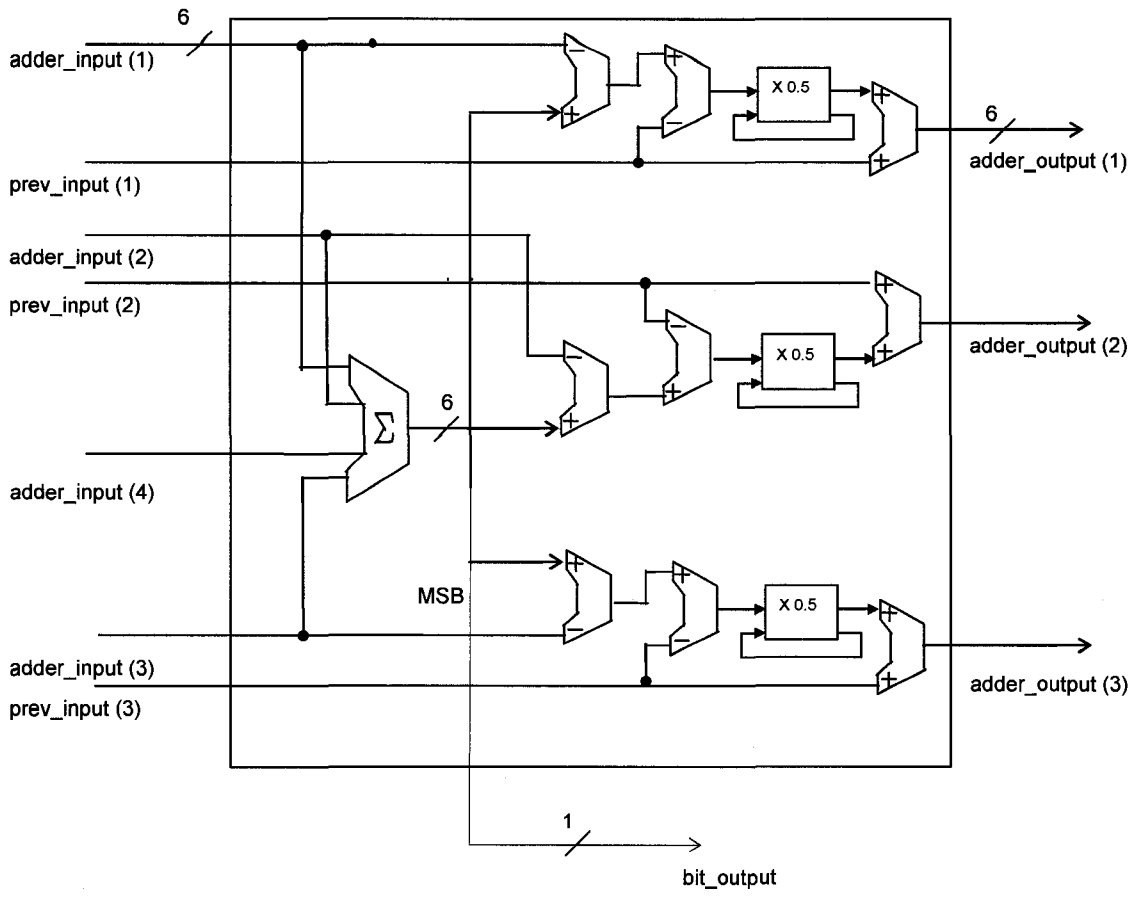


Figure 4-5: the architecture of the "variable_adder" module for SR-MS

For the (504, 252) code, $\beta = 0.5$. So the proposed architecture fits this code the best because we can divide any register by one half by doing a right shift. This shift register is shown in Figure 4-5. This module is designed with adding minimum complexity to the circuit which is adding 3 extra adders, 3 extra subtractors and 3 shift registers. All the messages are in 2's complement format, so special considerations should be made when performing the right shift operation to keep the sign unchanged. The Verilog code for this is listed below:

```
assign srr1 = sr1[5] ? {  (sr1[5:0] >> 1) + 6'b100000} : { sr1[5:0] >> 1 } ;
```

The above statement will ensure that when using the shift register for negative numbers, the sign is preserved. The Verilog source for the modified "variable_adder" module in SR-MS is listed in the appendix 5.

4.7 Verilog Test Benches Design

To test the functionality of the decoder modules, the Xilinx ISE behavioral simulation is used. In this methodology, to test the top modules, a (7, 4) code is designed and tested. The (7, 4) code has the Tanner graph shown in Figure 2-1. The clock frequency is set to 25MHz and maximum number of iterations is set to 200.

4.7.1 Top level verification

Figure 4-6 shows the test waveform for top level verification for the (7, 4) code. The Verilog source of this test program is presented in the appendix 6.

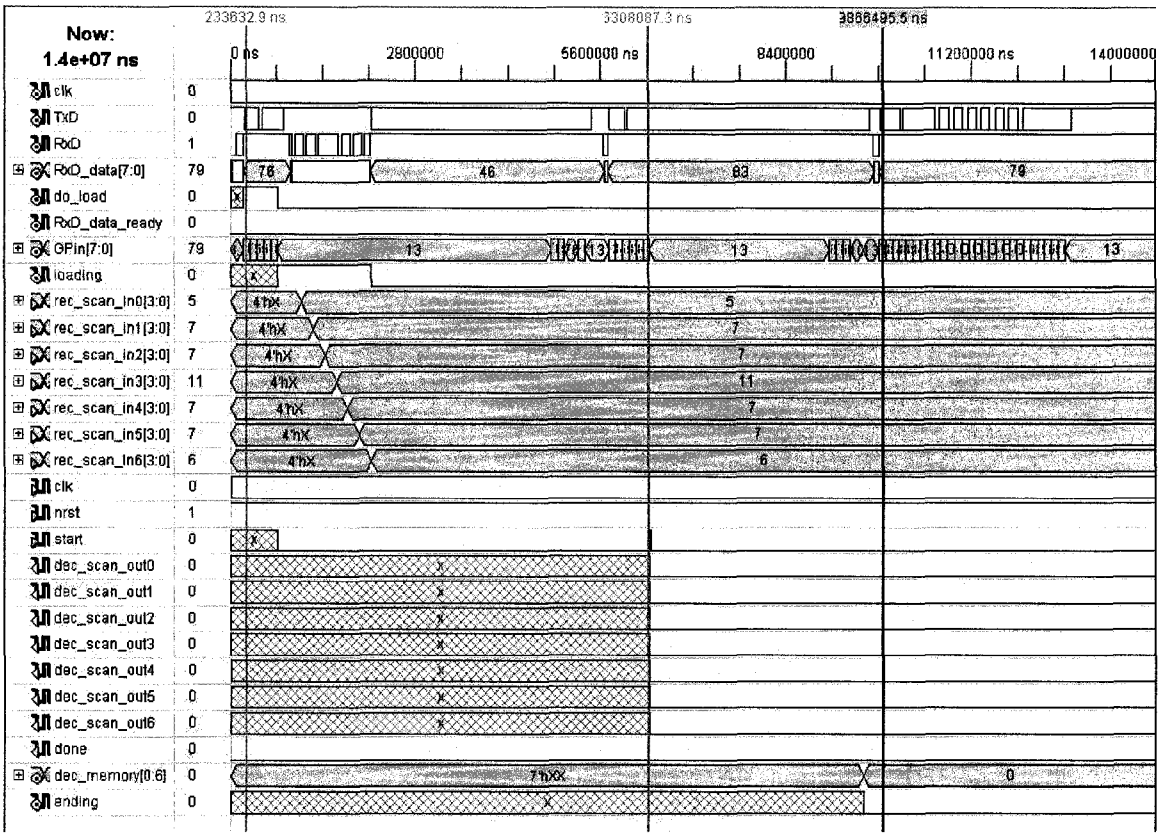


Figure 4-6 top level verification waveforms.

Figure 4-6 shows a complete decoding sequence that consists of loading data, starting the decoding and getting the output results. This is done using a serial port with the speed of 115200bps. The whole process for a (7,4) code will take about $14m\sec s$. The first marker at around $t \approx 200\mu s$ shows the 'L' command being sent. The second marker at $t \approx 5.3ms$ shows the 'S' command. The last marker at $t \approx 9.8ms$ shows the 'O' command. Figure 4-7 expands the time between $t \approx 6.34ms$ and $t \approx 6.35ms$. The actual decoding happens during a period of $200 * \frac{1}{25MHz} = 8\mu s$ which is shown in Figure 4-7. The 'start' and 'done' signals denote the request of decoding sent by the control circuit and the end of decoding message sent back by the decoder.

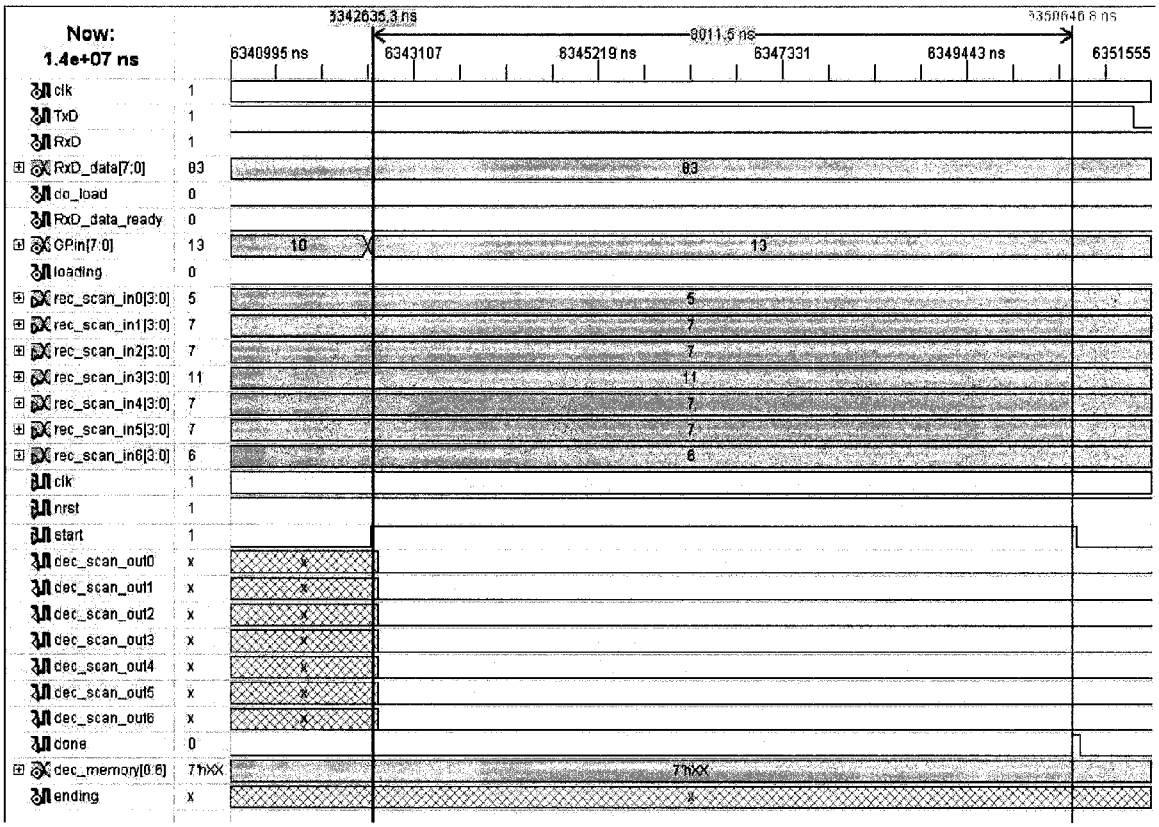


Figure 4-7 top level verification, the time between $t = 6.34ms$ and $t = 6.35ms$

4.7.2 Decoder module verification

To ensure the functionality of the main decoder module, a test bench is written to compare the values of edges at each iteration and compare it to the expected results. A sample of this test bench is shown in Figure 4-8. In this Figure, signals 'rec_scan_in0-6' denote the inputs to the decoder. Signals 'dec_scan_out0-6' denote the result of decoding. Signals starting with 'var_out' denote the edges connected to variable nodes and signals starting with 'out' denote the edges connected to check nodes. In the example shown in Figure 4-8, the values "3,5,2,1,0,5,9" (9 represents -1 in sign-magnitude representation) are sent to the

decoder. Each of the iterations is shown using a marker. After 4 iterations, the decoder converges to the values “0,0,0,0,0,0,0”.

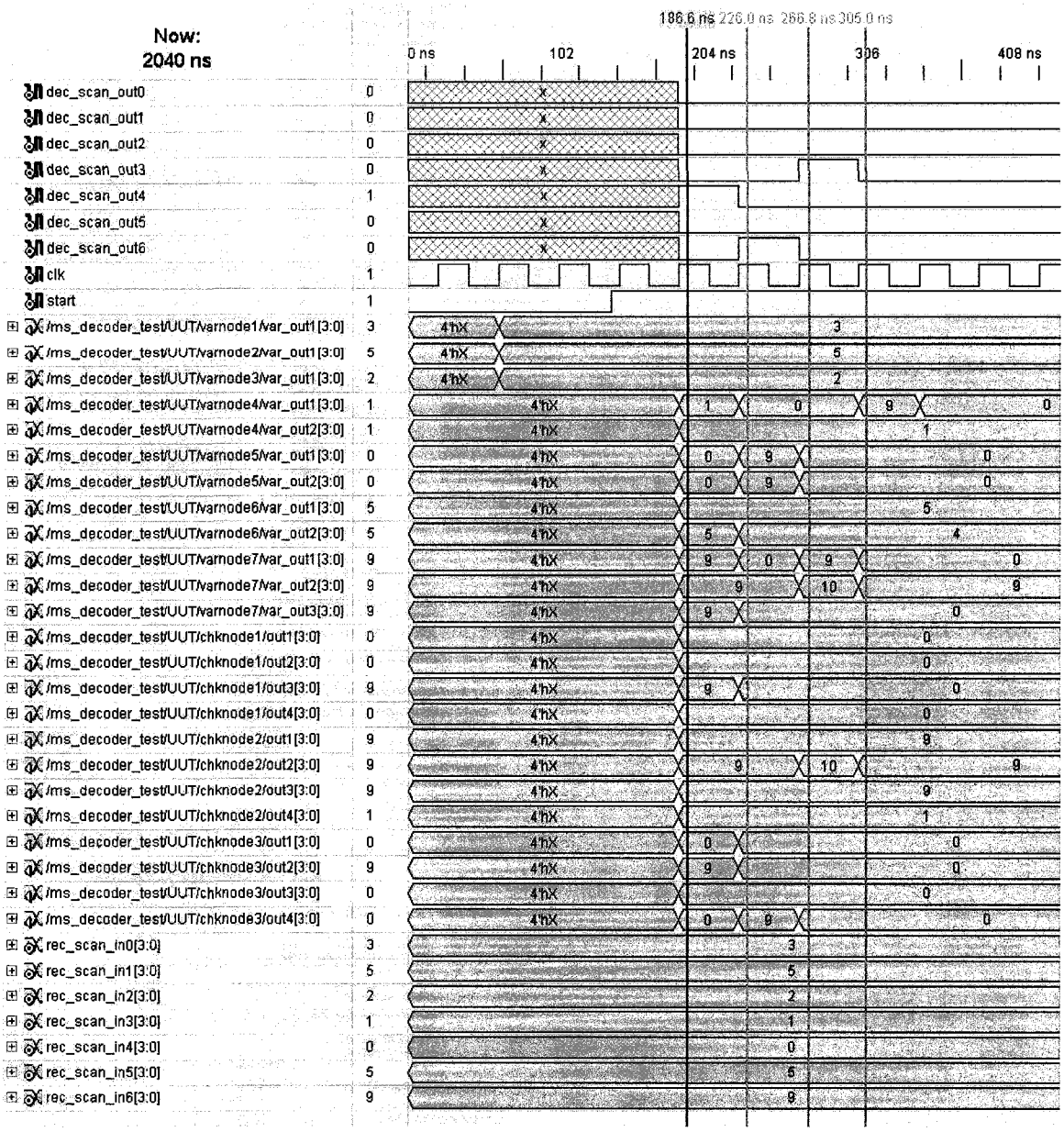


Figure 4-8 Decoder verification for (7, 4) code

4.8 Test Results

The performance of the (504, 252) code with 4-bit quantization is tested under different decoding algorithms. Figure 4-9 shows the BER curves obtained by simulation and FPGA.

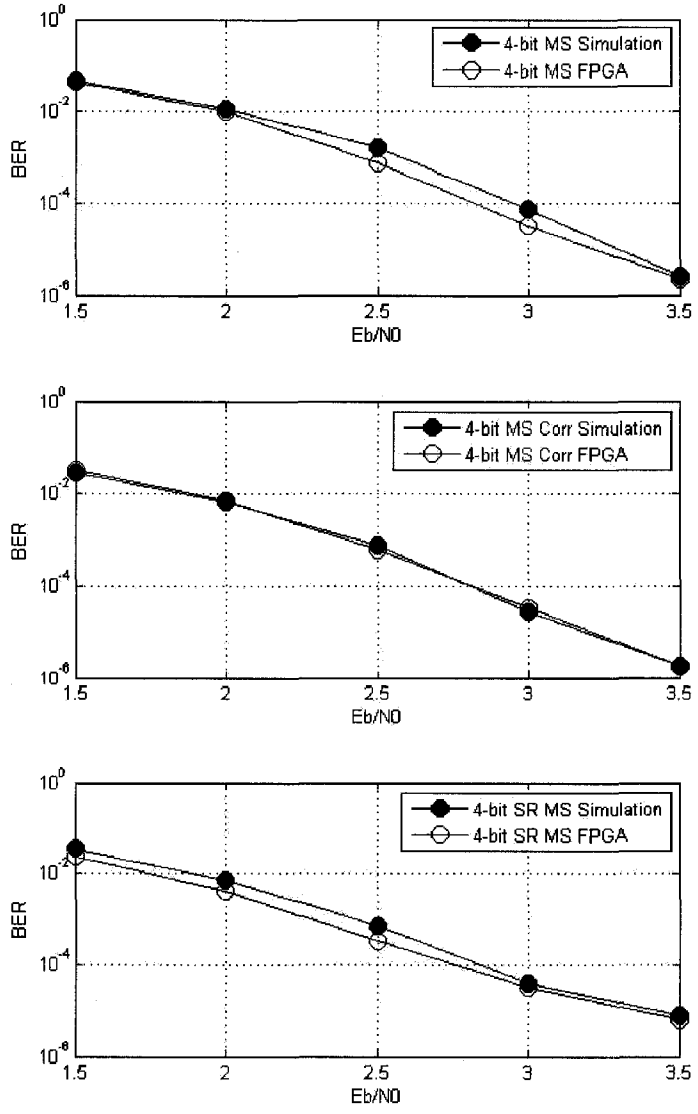


Figure 4-9 FPGA and simulation BER results for different decoding algorithms

Figure 4-9 verifies the functionality of the implemented algorithms. As we can see the BER performance of implemented algorithms are very close to the simulated decoders. The implemented algorithms perform the same as or better than the simulated algorithms for all SNR values. This source of this small variation is the difference between the noise values generators in the simulations using C functions and for testing the FPGA using Matlab functions.

4.9 Decoder throughput and utilization comparison

To compare the results from this work with recent implemented designs, the results of work in [19] is presented. In [19], a bit-serial architecture is used to connect the modules in a fully-parallel MS algorithm. The variable nodes and the check nodes communicate multi-bit messages over single wires. Thus, the architecture used in [19] is not a fully parallel architecture. Also, to reduce the check nodes complexity, a new approximation is introduced, in which in each check node, only one minimum magnitude is calculated and a corrective constant in then added to correct the check node outputs. Using this architecture the authors have implemented a (480, 355) code with 3-bit quantization and 15 iterations per frame which resulted in a 650 Mbps total throughput at clock frequency of 61 MHz. It is claimed that the implementation of [19] is the fastest FPGA LDPC decoder in the literature. The information throughput for the decoder of [19] which is the throughput based on input rate to the encoder is 480 Mbps.

Table 4-3 summarizes the FPGA implementation results for each implemented algorithm in this thesis. The information throughput is calculated based on 15 iterations for each block. Changing the maximum number of iterations to 15 will degrade the performance but we have chosen 15 iterations to have a fair comparison.

Table 4-3: Summary of FPGA implementation of (504, 252) code

Algorithm	Maximum Frequency	Information Throughput	Number of slice Flip Flops	Total Number of 4-input LUT	Total Equivalent gate count
MS	64 MHz	1.075 Gbps	8693 (9%)	66738 (75%)	560113
MS-UC	64 MHz	1.075 Gbps	8748 (9%)	68378 (77%)	570468
SR-MS	51 MHz	856 Mbps	8693 (9%)	85935 (97%)	762226
[19]	61 MHz	480 Mbps	Not reported	66588 (84%)	Not reported

As can be seen in table 4-3, the throughputs of all designed algorithms in this thesis are higher than the throughput in [19]. Compared to the MS algorithm, the circuit utilization increases about 2% with MS-UC and it will increase by 22% in SR-MS modification. When the circuit gets bigger, the critical path for minimum delay gets longer and so the maximum frequency and throughput will decrease. So there is a trade-off between higher performance and higher throughput in modifying the MS algorithm. The number of slice Flip Flops denotes the sequential circuits and number of registers used. In our design we have registers only in variable nodes and control circuits and there is no change in the sequential part when we modify the decoder in MS-US and SR-MS. So there is not much difference in the number of slice Flip Flops used. The total number of

4-input LUTs denotes the combinational circuits used and is changed in each modification. The logic utilization has also direct effect on power consumption. The results in table 4-3 show that although we are using a bigger code compared to the code in [19] and also we have 4-bit quantization instead of 3-bit quantization in [19], for MS and MS-UC in our design compared to [19], we have almost the same number of LUTs. So in terms of power consumption, we will have the same amount of power consumption. However, with 4-bit quantization we can get better performance while having higher throughput.

Chapter 5 Conclusion and Future work

5.1 Conclusion

In this thesis we have studied the MS algorithm and its modifications. We investigated the effects of clipping and quantization on application of successive relaxation to the MS algorithm. The optimum clipping threshold is shown to depend on the code and generally increase with SNR and the number of quantization bits. We demonstrated that for a given number of quantization bits, SR-MS is superior to other modifications of MS. We showed that for a regular (504, 252) code, with 7-bit quantization the BER performance of SR-MS outperforms the ideal BP algorithm and for an irregular (1268, 456) code, with 6-bit quantization the performance of SR-MS outperforms the ideal BP algorithm. Our results also indicate that the iterations distribution of MS-UC algorithm and SR-MS algorithm doesn't change significantly compared to the iterations distribution of the MS algorithm.

To investigate the performance of MS algorithm and its modified versions in a real implementation environment, all three algorithms are implemented in a Viretx-II pro FPGA. Specific modifications are made in the design of variable nodes to improve the performance, also check nodes are designed using combinational circuits to increase the speed. The SR-MS algorithm is also implemented using custom designed circuits to minimize the added complexity. A

4-bit quantized version of all three algorithms is implemented and tested. The test results verified the functionality of the design. The MS-UC algorithm adds 2% to the logic gates utilization and SR-MS algorithm adds 22% to the logic gates utilization. The logic utilization increases linearly with increasing the number of quantization bits, so this relative increase stays more or less constant. And in terms of performance, for example in (1268, 456) code, at BER=1E-6, with 7-bits quantization, MS-UC performs 0.3dB better than MS and SR-MS performs 0.6dB better than MS.

5.2 Future work

In this thesis we investigated the SR-MS and compared it with the best existing modifications of MS in terms of performance and complexity. And we implemented all the algorithms to investigate the added complexity for each implementation. SR-MS showed very good performance and increased complexity. Further research should be done on SR-MS algorithm to decrease its complexity. Relaxing the condition on passing only the extrinsic information in both check nodes and variable nodes can be one of the good ideas for research in this area. Another future area of research can be the asymptotic analysis of SR-MS using methods such as density evolution or EXIT charts.

In the implementation part, further modification should be done on the test bed to increase the test speed. The designed FPGA in this thesis is communicating with the PC using the serial port. This makes a bottleneck for the test system, because the speed of serial port is very low compared to the speed of decoding.

A system-on-chip can be designed to generate noise values on the chip to communicate with the FPGA. This will make the testing procedure much faster and simulating and testing easier for different codes and decoding algorithms. The next step will be investigating the trade-off between number of quantization bits, decoding algorithm and maximum clock frequency. Changing the number of quantization bits has a direct effect on the performance and clock frequency. Also, different decoding algorithms have different performances. So to have the optimum decoder, one should choose the optimum number of quantization bits for each decoding algorithm to get the maximum frequency.

Appendix 1, FPGA control modules for (7,4) code

```
`timescale 1ns / 1ps

  module sina_ms_7_4(clk, RxD, TxD);

  parameter vnodes=7; //i will be number from 0 to vnodes-1
  parameter bits=3;    //number of quantization bits (not including sign)

  input clk;
  input RxD;
  output TxD;

  reg nrst;
  reg [7:0] GPin;
  reg TxD_start;

  reg do_reset;
  reg do_load;
  reg do_read;
  reg do_start;
  reg do_output;
  reg do_full;
  reg do_end;
  reg do_done;

  reg [bits:0] temp_var;
  reg [16:0] vars;     //variable for going memory to store data for variable nodes

  reg loading;
  reg reading;
  reg outputing;
  reg starting;
  reg ending;

  reg [bits:0] rec_memory [0:vnodes-1];
  reg     dec_memory [0:vnodes-1];


  reg [5:0] tx_state;
  reg [5:0] rx_state;

  wire TxD_busy;
  wire RxD_data_ready;
  wire [7:0] RxD_data;


  wire dec_memory0;
  wire dec_memory1;
  wire dec_memory2;
  wire dec_memory3;
  wire dec_memory4;
  wire dec_memory5;
  wire dec_memory6;

  wire done;

  async_receiver  deserializer  (.clk(clk),  .RxD(RxD),  .RxD_data_ready(RxD_data_ready),
.RxD_data(RxD_data));
  // and here all data in GPin is transmitted to TX of serial port
  async_transmitter      serializer(.clk(clk),      .TxD(TxD),      .TxD_start(TxD_start),
.TxD_data(GPin), .TxD_busy(TxD_busy));

  always @(posedge clk)
  begin
       if(RxD_data_ready)
       begin
              case (RxD_data)
                     8'h58: //X
                     begin
                            do_reset <= 1'b1 ;     //when we press X, do reset
                            tx_state <= 1'b0 ;     //initializing output display counter
                     end
                     8'h4C: //L
                     begin
                            do_load  <= 1'b1 ;     //when we press L, do load
                            tx_state <= 1'b0 ;     //initializing output display counter
```

```
                end
                8'h52: //R
                begin
                        do_read  <= 1'b1 ;      //when we press R, do read
                        tx_state <= 1'b0 ;      //initializing output display counter
                end
                8'h53: //S
                begin
                        do_start <= 1'b1 ;      //when we press S, do start
                        tx_state <= 1'b0 ;      //initializing output display counter
                end
                8'h4F: //O
                begin
                        do_output <= 1'b1 ;     //when we press O, do output
                        tx_state  <= 1'b0 ;     //initializing output display counter
                end

                default:
                        TxD_start <= 1'b0 ;     //stop transmit
        endcase
end  // end if (RxD_data_ready)

else //~RxD_data_ready
begin
        TxD_start <= 1'b0;      //if there is not data, don't transit
end


//show "RESET" on output
if (do_reset & ~TxD_busy)
begin
        case (tx_state)
                0:  GPin  <= 8'h52;             //R
                2:  GPin  <= 8'h45;             //E
                4:  GPin  <= 8'h53;             //S
                6:  GPin  <= 8'h45;             //E
                8:  GPin  <= 8'h54;             //T
                10: GPin  <= 8'h0a;             //\n
                12: GPin  <= 8'h0d;             //\r
                13:
                begin
                        tx_state <= 1'b0;
                        do_reset <= 1'b0;               //finish reset
                        nrst            <= 1'b0;

                        rec_memory[0]<=0;
                        rec_memory[1]<=0;
                        rec_memory[2]<=0;
                        rec_memory[3]<=0;
                        rec_memory[4]<=0;
                        rec_memory[5]<=0;
                        rec_memory[6]<=0;

                        dec_memory[0]<= 1;
                        dec_memory[1]<= 1;
                        dec_memory[2]<= 1;
                        dec_memory[3]<= 1;
                        dec_memory[4]<= 1;
                        dec_memory[5]<= 1;
                        dec_memory[6]<= 1;

                        starting <=0;
                end
        endcase
        tx_state <= tx_state + 1 ;
        TxD_start <= 1'b1;                      //start transmit
end     //end (do_reset & ~TxD_busy)
//end show "RESET" on output

//show "LOAD" on output
if (do_load & ~TxD_busy)
begin
        case (tx_state)

                0:  GPin  <= 8'h4C;             //L
                2:  GPin  <= 8'h4F;             //O
                4:  GPin  <= 8'h41;             //A
                6:  GPin  <= 8'h44;             //D
                8:  GPin  <= 8'h0A;             //\n
                10: GPin  <= 8'h0D;             //\r
                11:
```

```
                begin
                        rx_state <= 0;
                        do_load  <= 0;          //finish load
                        vars            <= 0;           //initializing vars
                        loading  <= 1;          //start to load the data into memory
                        temp_var <= 0;     //initializing
                        starting <= 0; //stop current decoding
                end
        endcase
        tx_state <= tx_state + 1 ;
        TxD_start <= 1'b1;                      //start transmit
end     //end (do_load & ~TxD_busy)
//end show "LOAD" on output

//show "READ" on output
if (do_read & ~TxD_busy)
begin
        case (tx_state)
                0:  GPin  <= 8'h52;             //R
                2:  GPin  <= 8'h45;             //E
                4:  GPin  <= 8'h41;             //A
                6:  GPin  <= 8'h44;             //D
                8:  GPin  <= 8'h0A;             //\n
                10: GPin  <= 8'h0D;             //\r
                11:
                begin
                        tx_state <= 0;
                        do_read  <= 0;          //finish read
                        vars            <= 0;           //initializing vars
                        reading  <= 1;
temp_var <= rec_memory[0];
                end
        endcase
        tx_state <= tx_state + 1 ;
        TxD_start <= 1'b1;                      //start transmit
end     //end (do_read & ~TxD_busy)
//end show "READ" on output


//show "START" on output
if (do_start & ~TxD_busy)
begin
        case (tx_state)
                0:  GPin  <= 8'h53;             //S
                2:  GPin  <= 8'h54;             //T
                4:  GPin  <= 8'h41;             //A
                6:  GPin  <= 8'h52;             //R
                8:  GPin  <= 8'h54;             //T
                10: GPin  <= 8'h0A;             //\n
                12: GPin  <= 8'h0D;             //\r
                13:
                begin
                        tx_state <= 1'b0;
                        do_start <= 1'b0;               //finish start
                        starting <= 1'b1;               //run decoding
                end
        endcase
        tx_state <= tx_state + 1 ;
        TxD_start <= 1'b1;                      //start transmit
end     //end (do_start & ~TxD_busy)
//end show "START" on output

//show "OUTPUT" on output
if (do_output & ~TxD_busy)
begin
        case (tx_state)
                0:  GPin  <= 8'h4f;             //O
                2:  GPin  <= 8'h55;             //U
                4:  GPin  <= 8'h54;             //T
                6:  GPin  <= 8'h50;             //P
                8:  GPin  <= 8'h55;             //U
                10: GPin  <= 8'h54;             //T
                12: GPin  <= 8'h0A;             //\n
                14: GPin  <= 8'h0D;             //\r
                15:
                begin
                        tx_state  <= 0;
                        do_output <= 0;                 //finish read
                        vars              <= 0;         //initializing vars
                        outputing <= 1;
                        temp_var  <= dec_memory[0];
```

```
                end
        endcase
        tx_state <= tx_state + 1 ;
        TxD_start <= 1'b1;                      //start transmit
end     //end (do_output & ~TxD_busy)
//end show "OUTPUT" on output

//show "FULL" on output
if (do_full & ~TxD_busy)
begin
        case (tx_state)
                0:  GPin  <= 8'h46;             //F
                2:  GPin  <= 8'h55;             //U
                4:  GPin  <= 8'h4C;             //L
                6:  GPin  <= 8'h4C;             //L
                8:  GPin  <= 8'h0A;             //\n
                10: GPin  <= 8'h0D;             //\r
                11:
                begin
                        tx_state  <= 1'b0;
                        do_full   <= 1'b0;              //finish full
                end
        endcase
        tx_state <= tx_state + 1 ;
        TxD_start <= 1'b1;                      //start transmit
end     //end (do_full & ~TxD_busy)
//end show "FULL" on output

//show "DONE" on output
if (do_done & ~TxD_busy)
begin
        case (tx_state)
                0:  GPin  <= 8'h44;             //D
                2:  GPin  <= 8'h4f;             //O
                4:  GPin  <= 8'h4e;             //N
                6:  GPin  <= 8'h45;             //E
                8:  GPin  <= 8'h0A;             //\n
                10: GPin  <= 8'h0D;             //\r
                11:
                begin
                        tx_state  <= 1'b0;
                        do_done   <= 1'b0;              //finish done
                end
        endcase
        tx_state <= tx_state + 1 ;
        TxD_start <= 1'b1;                      //start transmit
end     //end (do_done & ~TxD_busy)
//end show "DONE" on output

//show "END" on output
if (do_end & ~TxD_busy)
begin
        case (tx_state)
                0:  GPin  <= 8'h45;             //E
                2:  GPin  <= 8'h4e;             //N
                4:  GPin  <= 8'h44;             //D
                6:  GPin  <= 8'h0A;             //\n
                10: GPin  <= 8'h0D;             //\r
                11:
                begin
                        tx_state  <= 1'b0;
                        do_end    <= 1'b0;              //finish end
                        ending   <= 1'b1;
                end
        endcase
        tx_state <= tx_state + 1 ;
        TxD_start <= 1'b1;                      //start transmit
end     //end (do_end & ~TxD_busy)
//end show "END" on output

//*****************************************************************
//                                              LOADING DATA TO MEMORY          //
//*****************************************************************
if(RxD_data_ready && loading)
begin
        if (RxD_data == 8'h2E) //.
                begin
                        rx_state <= 0 ;         //initializing output display counter
                        rec_memory[vars]<=temp_var;
                        temp_var <= 0;
```

```
                        vars <= vars + 1 ;
                        if ( vars == vnodes - 1)
                                begin
                                        vars            <= 0;
//initializing vars
                                        loading  <= 0;            //finish loading
                                        temp_var <= 0;        //initializing
                                        rx_state <= 0 ;//initializing output counter
                                        do_full  <= 1 ;   //show full on output
                                end

                        end
        else
                begin
                        if ( rx_state == 0 )
                                begin
                                        temp_var[bits:0]<= RxD_data;
                                        rx_state <= rx_state + 1;
                                end
                end


end  // end if (RxD_data_ready && loading)
//*****************************************************************
//                                        END OF LOADING DATA TO MEMORY //
//*****************************************************************
//*****************************************************************
//                                        READING CHANNEL DATA FROM MEMORY//
//*****************************************************************
if(~TxD_busy && reading)
begin
        case (tx_state)
                0:
                        begin
                                GPin  <= temp_var[bits:0];
                                tx_state <= tx_state + 1 ;
                                TxD_start <= 1'b1;              //start transmit
                        end
                2:
                        begin
                                GPin  <= 8'h2E;                                 //.
                                tx_state <= tx_state + 1 ;
                                TxD_start <= 1'b1;              //start transmit
                        end
                3:
                        begin
                                tx_state  <= 0;
                                vars <= vars + 1;
                                temp_var <= rec_memory[vars+1];
                        end
                default:
                        begin
                                TxD_start <= 1'b0;              //stop transmit
                                tx_state <= tx_state + 1 ;
                        end
        endcase

        if ( vars == vnodes )
                begin
                        vars            <= 0;                   //initializing vars
                        reading  <= 0;              //finish reading
                        temp_var <= 0;                      //initializing
                        tx_state <= 0 ;   //initializing output display counter
                        do_done  <= 1 ;           //show done on output
                        TxD_start <= 1'b0;                      //stop transmit
                end
end  // end if (~TxD_busy && reading)
//*****************************************************************
//              END OF READING CHANNEL DATA FROM MEMORY                         //
//*****************************************************************
//*****************************************************************
//              READING DECODED DATA FROM MEMORY                          //
//*****************************************************************
if(~TxD_busy && outputing)
begin
        case (tx_state)
                0:
                        begin
                                GPin  <= temp_var[0];
                                tx_state <= tx_state + 1 ;
                                TxD_start <= 1'b1;              //start transmit
```

```
                        end
                2:
                        begin
                                GPin  <= 8'h2E;                                  //.
                                tx_state <= tx_state + 1 ;
                                TxD_start <= 1'b1;              //start transmit
                        end
                3:
                        begin
                                tx_state  <= 0;
                                vars <= vars + 1;
                                temp_var <= dec_memory[vars+1];
                        end
                default:
                        begin
                                TxD_start <= 1'b0;              //stop transmit
                                tx_state <= tx_state + 1 ;
                        end
        endcase

        if ( vars == vnodes )
                begin
                        vars            <= 0;          //initializing vars
                        outputing <= 0;                   //finish outputing
                        temp_var  <= 0;                //initializing
                        tx_state  <= 0 ;  //initializing output display counter
                        do_done   <= 1 ;          //show done on output
                        TxD_start <= 1'b0;                    //stop transmit
                end
    end  // end if (~TxD_busy && reading)
    //*********************************************************************
    //              END OF READING DECODED DATA FROM MEMORY               //
    //*********************************************************************
    if (done)
        begin
        do_end <=1;                             //show end
        starting  <= 0;                //finish decoding
    end //end if done

    if (ending)
        begin
        dec_memory[0]<= dec_memory0;
        dec_memory[1]<= dec_memory1;
        dec_memory[2]<= dec_memory2;
        dec_memory[3]<= dec_memory3;
        dec_memory[4]<= dec_memory4;
        dec_memory[5]<= dec_memory5;
        dec_memory[6]<= dec_memory6;
        ending <=0;
        end

  end   //end always @(posedge clk)

  decoder ms_decoder
            (
                    rec_memory[0],    rec_memory[1],    rec_memory[2],    rec_memory[3],
rec_memory[4], rec_memory[5], rec_memory[6],
                    dec_memory0,      dec_memory1,      dec_memory2,      dec_memory3,
dec_memory4,   dec_memory5,   dec_memory6,

                    clk,
                    1'b1,          //nrst
                    starting,
                    done

            );


  endmodule
```

Appendix 2, Matlab control program

```
clc
clipping=1.5;
bit=4;
n=504;

snr_db=3.5;
net_file='504_ms_4_3.5_full.txt';

snr_dec=power(10,(snr_db/10));
rate=0.5;
sigma=sqrt(1/(2*rate*snr_dec));     % rate is 0.5

global use_last;
global y2;

use_last=0;
y2=0;

maxblockerror=100;
maxblocks=10e8;


rand('seed',457487);
randn('seed',457487);    %for awgn function

s1 = serial('COM1','BaudRate',115200);  %write

s1.Timeout=2;

s1.OutputBufferSize=2048;
s1.InputBufferSize=2048;

fopen(s1);

totalblockerror=0;
totalblocks=0;
totalbiterrors=0;
t = clock;
ber=0;
wer=0;

fid = fopen(net_file,'w+');
while (totalblockerror < maxblockerror &&  totalblocks < maxblocks)

    x1 = (sign(randn(1,size(G',1)))+1)/2; % random bits
    x2 = mod(x1*G',2); % encoding
    x = 1-2*x2; % BPSK modulation
    y=x + sigma*randn(1,length(y));

    for i=1:size(y,2)
```

```
    z(i)=quan(y(i),clipping,bit);

    if (z(i)<0)
        z(i)=abs(z(i))+8;
    end
end
z;

fwrite(s1,'L','int8','async');      %load

while (s1.bytesavailable<6)
end

out=char(fread(s1,6)'); %shows load

load=zeros(1,n*2);
for i=1:n
    load(i*2-1)=z(i);
    load(i*2)=46;
end

fwrite(s1,load,'int8','async'); %writing data

while (s1.bytesavailable<32)
end

out=char(fread(s1,32)');


fwrite(s1,'S','int8','async'); %start decoding


while (s1.bytesavailable<38)
end
out=char(fread(s1,38)');     %shows START and then END


fwrite(s1,'O','int8','async'); %get the decoded data

while (s1.bytesavailable<1022)
end
out=fread(s1,1022);

out=out(9:size(out,1)-6)';

for i=1:size(out,2)/2
    out1(i)=out(i*2-1);
end

errors=0;
for i=1:size(out1,2)
    if(out1(i) ~=x2(i) )
        errors=errors+1;
```

```
        end
    end

    totalblocks=totalblocks+1;
    if (errors~=0)
        totalblockerror=totalblockerror+1;
        totalbiterrors=totalbiterrors+errors;
        ber=totalbiterrors/(n*totalblocks);
        wer=totalblockerror/totalblocks;
        fprintf(fid,'%s :  Block Error = %d \t BER= %E \t WER= %E \t
TotalBlocks= %d\n',datestr(now),totalblockerror,ber,wer, totalblocks);
    else
        fprintf(fid,'%s :  %d\n',datestr(now), totalblocks);
    end

end


ber=totalbiterrors/(n*totalblocks);
wer=totalblockerror/totalblocks;
t=etime(clock, t)

fprintf(fid,'\n For SNR= %f  \tBER= %E \tWER= %E \tTotalBlocks=
%d\n',snr_db,ber,wer, totalblocks);
fprintf(fid,'\n Program took %6.0f seconds to run.\n', t);

serial_close;
fclose(fid);
```

Appendix 3, The tc2sm module

```
module ts2sm(out, in);

  input[5:0] in;
  output[3:0] out;
  reg[3:0] out;

  always @(in)
    begin
      casex(in)
                    6'b000000:      out = 4'b0000;//0
                    6'b000001:      out = 4'b0001;//1
                    6'b000010:      out = 4'b0010;//2
                    6'b000011:      out = 4'b0011;//3
                    6'b000100:      out = 4'b0100;//4
                    6'b000101:      out = 4'b0101;//5
                    6'b000110:      out = 4'b0110;//6
                    6'b000111:      out = 4'b0111;//7
                    6'b001xxx:      out = 4'b0111;//7
                    6'b01xxxx:      out = 4'b0111;//7

                    6'b10xxxx:      out = 4'b1111;//-7
                    6'b110xxx:      out = 4'b1111;//-7
                    6'b11100x:      out = 4'b1111;//-7
                    6'b111010:      out = 4'b1110;//-6
                    6'b111011:      out = 4'b1101;//-5
                    6'b111100:      out = 4'b1100;//-4
                    6'b111101:      out = 4'b1011;//-3
                    6'b111110:      out = 4'b1010;//-2
                    6'b111111:      out = 4'b1001;//-1

        default: out = 4'b0000;
      endcase
    end

endmodule
```

Appendix 4, The check_node module

```
`timescale 1ns/10ps

module check_node6
              (
               out1,                in1,
               out2,                in2,
               out3,                 in3,
               out4,                 in4,
               out5,                in5,
               out6,                 in6,

                                     clk
              );


     input clk;
   input[3:0] in1;
   input[3:0] in2;
   input[3:0] in3;
   input[3:0] in4;
   input[3:0] in5;
   input[3:0] in6;

   output[3:0] out1;
   output[3:0] out2;
   output[3:0] out3;
   output[3:0] out4;
   output[3:0] out5;
   output[3:0] out6;

   wire[2:0] rel_in1;
   wire[2:0] rel_in2;
   wire[2:0] rel_in3;
   wire[2:0] rel_in4;
   wire[2:0] rel_in5;
   wire[2:0] rel_in6;


      wire[3:0] out1;
      wire[3:0] out2;
      wire[3:0] out3;
      wire[3:0] out4;
      wire[3:0] out5;
      wire[3:0] out6;

   wire[2:0] min12;
   wire[2:0] min34;
     wire[2:0] min56;
      wire[2:0] min1234;
      wire[2:0] min3456;
      wire[2:0] min1256;

      assign rel_in1 = in1[2:0];
      assign rel_in2 = in2[2:0];
      assign rel_in3 = in3[2:0];
      assign rel_in4 = in4[2:0];
      assign rel_in5 = in5[2:0];
      assign rel_in6 = in6[2:0];


      assign min12 = (rel_in1 < rel_in2) ? rel_in1 : rel_in2;
      assign min34 = (rel_in3 < rel_in4) ? rel_in3 : rel_in4;
      assign min56 = (rel_in5 < rel_in6) ? rel_in5 : rel_in6;

      assign min1234 = (min12 < min34) ? min12 : min34;
      assign min3456 = (min34 < min56) ? min34 : min56;
      assign min1256 = (min12 < min56) ? min12 : min56;

      assign out1[2:0] = (rel_in2 < min3456 )? rel_in2 : min3456;
      assign out2[2:0] = (rel_in1 < min3456 )? rel_in1 : min3456;
      assign out3[2:0] = (rel_in4 < min1256 )? rel_in4 : min1256;
      assign out4[2:0] = (rel_in3 < min1256 )? rel_in3 : min1256;
      assign out5[2:0] = (rel_in6 < min1234 )? rel_in6 : min1234;
      assign out6[2:0] = (rel_in5 < min1234 )? rel_in5 : min1234;
```

```
        assign out1[3] = (in2==0 || in3==0 || in4==0 || in5==0 || in6==0 ) ? 0 : (in2[3] ^
in3[3] ^ in4[3] ^ in5[3] ^ in6[3]);
        assign out2[3] = (in1==0 || in3==0 || in4==0 || in5==0 || in6==0 ) ? 0 : (in1[3] ^
in3[3] ^ in4[3] ^ in5[3] ^ in6[3]);
        assign out3[3] = (in1==0 || in2==0 || in4==0 || in5==0 || in6==0 ) ? 0 : (in1[3] ^
in2[3] ^ in4[3] ^ in5[3] ^ in6[3]);
        assign out4[3] = (in1==0 || in2==0 || in3==0 || in5==0 || in6==0 ) ? 0 : (in1[3] ^
in2[3] ^ in3[3] ^ in5[3] ^ in6[3]);
        assign out5[3] = (in1==0 || in2==0 || in3==0 || in4==0 || in6==0 ) ? 0 : (in1[3] ^
in2[3] ^ in3[3] ^ in4[3] ^ in6[3]);
        assign out6[3] = (in1==0 || in2==0 || in3==0 || in4==0 || in5==0 ) ? 0 : (in1[3] ^
in2[3] ^ in3[3] ^ in4[3] ^ in5[3]);

  endmodule
```

Appendix 5, SR_variable_adder module

```
`timescale 1ns/10ps

module sr_variable_adder3(msb,eadder_input1, eadder_input2, eadder_input3, adder_input1,
adder_input2, adder_input3, adder_input4, var_outp1, var_outp2, var_outp3);

      input[5:0] adder_input1;
      input[5:0] adder_input2;
      input[5:0] adder_input3;
      input[5:0] adder_input4;
      input[5:0] var_outp1;
      input[5:0] var_outp2;
      input[5:0] var_outp3;

      output[5:0] eadder_input1;
      output[5:0] eadder_input2;
      output[5:0] eadder_input3;
      output     msb;

      wire[5:0] rowadder_input;

      wire[5:0] sr1;
      wire[5:0] sr2;
      wire[5:0] sr3;

      wire[5:0] srr1;
      wire[5:0] srr2;
      wire[5:0] srr3;


      assign rowadder_input = adder_input1 + adder_input2 + adder_input3 + adder_input4;

      assign sr1 = rowadder_input -adder_input1 -var_outp1;
      assign sr2 = rowadder_input -adder_input2 -var_outp2;
      assign sr3 = rowadder_input -adder_input3 -var_outp3;

      assign srr1 = sr1[5] ? { (sr1[5:0] >> 1) + 6'b100000} : { sr1[5:0] >> 1 } ;
      assign srr2 = sr2[5] ? { (sr2[5:0] >> 1) + 6'b100000} : { sr2[5:0] >> 1 } ;
      assign srr3 = sr3[5] ? { (sr3[5:0] >> 1) + 6'b100000} : { sr3[5:0] >> 1 } ;


      assign eadder_input1 = var_outp1+ srr1;
      assign eadder_input2 = var_outp2+ srr2;
      assign eadder_input3 = var_outp3+ srr3;

      assign msb = rowadder_input[5];


endmodule
```

Appendix 6, Top level verification module

```
`timescale 1ns / 1ps

module top_testing_v;

        parameter PERIOD = 40;

        // Inputs
        reg clk;
        reg RxD;

        // Outputs
        wire TxD;

        // Instantiate the Unit Under Test (UUT)
        sina_ms_7_4 uut (
                .clk(clk),
                .RxD(RxD),
                .TxD(TxD)
        );

        initial begin
                // Initialize Inputs
                clk = 1'b0;
                RxD = 1;

                // Wait 100 ns for global reset to finish
                #8700;
                RxD = 0;

                #78300; //this is delay, actual time is #8700 + #78300 =
                RxD = 1;

                #8700           //for stop bit

                #11000          //wait time for getting RxD_data_ready
                //now starting to do load

                //L=01001100
                //we have to send from last bit to first bit

                RxD=0;
                #26100;
                RxD=1;
                #17400;
```

```
RxD=0;
#17400;
RxD=1;
#8700;
RxD=0;
#8700;
RxD=1;
#9800;  //stop bit + wait to get RxD_data_ready (8700+1100)

#700000;        //wait for TXD to send "LOADING" to output

//start to input data

//5     00000101

#1                      RxD=0;  //start bit
#8700           RxD=1;  //bit 7
#8700           RxD=0;  //bit 6
#8700           RxD=1;  //bit 5
#8700           RxD=0;  //bit 4
#8700           RxD=0;  //bit 3
#8700           RxD=0;  //bit 2
#8700           RxD=0;  //bit 1
#8700           RxD=0;  //bit 0
#8700           RxD=1;  //stop bit
#9800;                  //wait to get RxD_data_ready (8700+1100=9800)

//.     00101110

#1                      RxD=0;  //start bit
#8700           RxD=0;  //bit 7
#8700           RxD=1;  //bit 6
#8700           RxD=1;  //bit 5
#8700           RxD=1;  //bit 4
#8700           RxD=0;  //bit 3
#8700           RxD=1;  //bit 2
#8700           RxD=0;  //bit 1
#8700           RxD=0;  //bit 0
#8700           RxD=1;  //stop bit
#9800;                  //wait to get RxD_data_ready (8700+1100=9800)

//7     00000111

#1                      RxD=0;  //start bit
#8700           RxD=1;  //bit 7
#8700           RxD=1;  //bit 6
```

```
#8700          RxD=1; //bit 5
#8700          RxD=0; //bit 4
#8700          RxD=0; //bit 3
#8700          RxD=0; //bit 2
#8700          RxD=0; //bit 1
#8700          RxD=0; //bit 0
#8700          RxD=1; //stop bit
#9800;                //wait to get RxD_data_ready (8700+1100=9800)

//.     00101110

#1                    RxD=0; //start bit
#8700          RxD=0; //bit 7
#8700          RxD=1; //bit 6
#8700          RxD=1; //bit 5
#8700          RxD=1; //bit 4
#8700          RxD=0; //bit 3
#8700          RxD=1; //bit 2
#8700          RxD=0; //bit 1
#8700          RxD=0; //bit 0
#8700          RxD=1; //stop bit
#9800;                //wait to get RxD_data_ready (8700+1100=9800)

//7     00000111

#1                    RxD=0; //start bit
#8700          RxD=1; //bit 7
#8700          RxD=1; //bit 6
#8700          RxD=1; //bit 5
#8700          RxD=0; //bit 4
#8700          RxD=0; //bit 3
#8700          RxD=0; //bit 2
#8700          RxD=0; //bit 1
#8700          RxD=0; //bit 0
#8700          RxD=1; //stop bit
#9800;                //wait to get RxD_data_ready (8700+1100=9800)

//.     00101110

#1                    RxD=0; //start bit
#8700          RxD=0; //bit 7
#8700          RxD=1; //bit 6
#8700          RxD=1; //bit 5
#8700          RxD=1; //bit 4
#8700          RxD=0; //bit 3
#8700          RxD=1; //bit 2
```

```
#8700          RxD=0; //bit 1
#8700          RxD=0; //bit 0
#8700          RxD=1; //stop bit
#9800;                //wait to get RxD_data_ready (8700+1100=9800)

//11    00001011

#1                    RxD=0; //start bit
#8700          RxD=1; //bit 7
#8700          RxD=1; //bit 6
#8700          RxD=0; //bit 5
#8700          RxD=1; //bit 4
#8700          RxD=0; //bit 3
#8700          RxD=0; //bit 2
#8700          RxD=0; //bit 1
#8700          RxD=0; //bit 0
#8700          RxD=1; //stop bit
#9800;                //wait to get RxD_data_ready (8700+1100=9800)

//.     00101110

#1                    RxD=0; //start bit
#8700          RxD=0; //bit 7
#8700          RxD=1; //bit 6
#8700          RxD=1; //bit 5
#8700          RxD=1; //bit 4
#8700          RxD=0; //bit 3
#8700          RxD=1; //bit 2
#8700          RxD=0; //bit 1
#8700          RxD=0; //bit 0
#8700          RxD=1; //stop bit
#9800;                //wait to get RxD_data_ready (8700+1100=9800)

//7     00000111

#1                    RxD=0; //start bit
#8700          RxD=1; //bit 7
#8700          RxD=1; //bit 6
#8700          RxD=1; //bit 5
#8700          RxD=0; //bit 4
#8700          RxD=0; //bit 3
#8700          RxD=0; //bit 2
#8700          RxD=0; //bit 1
#8700          RxD=0; //bit 0
#8700          RxD=1; //stop bit
#9800;                //wait to get RxD_data_ready (8700+1100=9800)
```

```
//.     00101110

#1                      RxD=0; //start bit
#8700           RxD=0; //bit 7
#8700           RxD=1; //bit 6
#8700           RxD=1; //bit 5
#8700           RxD=1; //bit 4
#8700           RxD=0; //bit 3
#8700           RxD=1; //bit 2
#8700           RxD=0; //bit 1
#8700           RxD=0; //bit 0
#8700           RxD=1; //stop bit
#9800;                  //wait to get RxD_data_ready (8700+1100=9800)

//7     00000111

#1                      RxD=0; //start bit
#8700           RxD=1; //bit 7
#8700           RxD=1; //bit 6
#8700           RxD=1; //bit 5
#8700           RxD=0; //bit 4
#8700           RxD=0; //bit 3
#8700           RxD=0; //bit 2
#8700           RxD=0; //bit 1
#8700           RxD=0; //bit 0
#8700           RxD=1; //stop bit
#9800;                  //wait to get RxD_data_ready (8700+1100=9800)

//.     00101110

#1                      RxD=0; //start bit
#8700           RxD=0; //bit 7
#8700           RxD=1; //bit 6
#8700           RxD=1; //bit 5
#8700           RxD=1; //bit 4
#8700           RxD=0; //bit 3
#8700           RxD=1; //bit 2
#8700           RxD=0; //bit 1
#8700           RxD=0; //bit 0
#8700           RxD=1; //stop bit
#9800;                  //wait to get RxD_data_ready (8700+1100=9800)


//6     00000110
```

```
#1                      RxD=0;  //start bit
#8700           RxD=0;  //bit 7
#8700           RxD=1;  //bit 6
#8700           RxD=1;  //bit 5
#8700           RxD=0;  //bit 4
#8700           RxD=0;  //bit 3
#8700           RxD=0;  //bit 2
#8700           RxD=0;  //bit 1
#8700           RxD=0;  //bit 0
#8700           RxD=1;  //stop bit
#9800;                  //wait to get RxD_data_ready (8700+1100=9800)

//.     00101110

#1                      RxD=0;  //start bit
#8700           RxD=0;  //bit 7
#8700           RxD=1;  //bit 6
#8700           RxD=1;  //bit 5
#8700           RxD=1;  //bit 4
#8700           RxD=0;  //bit 3
#8700           RxD=1;  //bit 2
#8700           RxD=0;  //bit 1
#8700           RxD=0;  //bit 0
#8700           RxD=1;  //stop bit
#9800;                  //wait to get RxD_data_ready (8700+1100=9800)

#3500000;               //wait for TXD to send "FULL"

//S     01010011

#1                      RxD=0;  //start bit
#8700           RxD=1;  //bit 7
#8700           RxD=1;  //bit 6
#8700           RxD=0;  //bit 5
#8700           RxD=0;  //bit 4
#8700           RxD=1;  //bit 3
#8700           RxD=0;  //bit 2
#8700           RxD=1;  //bit 1
#8700           RxD=0;  //bit 0
#8700           RxD=1;  //stop bit
#9800;                  //wait to get RxD_data_ready (8700+1100=9800)

#4000000;       //wait for TXD to end END

//O     01001111
```

```
                #1                      RxD=0; //start bit
                #8700           RxD=1; //bit 7
                #8700           RxD=1; //bit 6
                #8700           RxD=1; //bit 5
                #8700           RxD=1; //bit 4
                #8700           RxD=0; //bit 3
                #8700           RxD=0; //bit 2
                #8700           RxD=1; //bit 1
                #8700           RxD=0; //bit 0
                #8700           RxD=1; //stop bit
                #9800;                  //wait to get RxD_data_ready (8700+1100=9800)

        end

        always
                #(PERIOD/2) clk = ~clk;

endmodule
```

References

[1] S. Lin and D. J. Costello, *Error Control Coding : Fundamentals and Applications*, 2nd ed.Upper Saddle River, N.J.: Pearson-Prentice Hall, 2004.

[2] C. Berrou, A. Glavieux and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *Proc. Int. Conf. Comm,* pp. 1064-1070. 1993.

[3] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters,* vol. 33, pp. 457-458, March 1997.

[4] R. Gallager, "Low-density parity-check codes," *IEEE Trans. Inform. Theory,* vol. 8, pp. 21-28, Jan. 1962.

[5] N. Wiberg, "Codes and decoding on general graphs." Ph.D. dissertation, Linköping University, Sweden, 1996.

[6] R. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inform. Theory,* vol. 27, pp. 533-547, Sept 1981.

[7] M. P. C. Fossorier, M. Mihaljevic and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Trans. Comm.,* vol. 47, pp. 673-680, May 1999.

[8] J. Chen and M. P. C. Fossorier, "Density evolution for two improved BP-Based decoding algorithms of LDPC codes," *IEEE Comm. Letters,* vol. 6, pp. 208-210, May 2002.

[9] J. Chen and P. M. C. Fossorier, "Density evolution for BP-based decoding algorithms of LDPC codes and their quantized versions," in *Proc. IEEE Globecomm,* pp. 1378-1382 vol.2, 2002.

[10] 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, pp. 549-554, April 2005.

[11] S. Hong, J. Yi and W. E. Stark, "Design and implementation of a low complexity VLSI turbo-code decoder architecture for low energy mobile wireless communications," *J. VLSI Signal Processing,* vol. 24, pp. 43-57, 2000.

[12] V. C. Gaudet and P. G. Gulak, "A 13.3-Mb/s 0.35-/spl mu/m CMOS analog turbo decoder IC with a configurable interleaver," *IEEE J. Solid-State Circuits,* vol. 38, pp. 2010-2015, Nov 2003.

[13] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *IEEE J. Solid-State Circuits,* vol. 37, pp. 404-412, March 2002.

[14] T. Zhang and K. K. Parhi, "A 54 mbps (3,6)-regular FPGA LDPC decoder," in *IEEE Workshop on Signal Proc.* pp. 127-132. Oct 2002.

[15] F. Verdier and D. Declercq, "A low-cost parallel scalable FPGA architecture for regular and irregular LDPC decoding," *IEEE Trans. Comm.,* vol. 54, pp. 1215-1223, July 2006.

[16] M. Karkooti and J. R. Cavallaro, "Semi-parallel reconfigurable architectures for real-time LDPC decoding," in *Proc. Int. Conf. Inform. Tech.* pp. 579-585 vol.1. 2004.

[17] T. Ishikawa, K. Shimizu, T. Ikenaga and S. Goto, "High-throughput decoder for low-density parity-check code," in *Proc. Asia and South Pacific Conf. Design Auto.* pp. 2. 2006.

[18] K. K. Gunnam, G. S. Choi and M. B. Yeary, "A parallel VLSI architecture for layered decoding for array LDPC codes," in *Proc. Int. Conf. Embed. Syst. & VLSI Des.* pp. 738-743. 2007.

[19] A. Darabiha, A. C. Carusone and F. R. Kschischang, "A bit-serial approximate min-sum LDPC decoder and FPGA implementation," in *Proc. IEEE Circuits and Systems,* pp. 4. 2006.

[20] S. Hemati, A. H. Banihashemi and C. Plett, "A 0.18-um CMOS Analog Min-Sum Iterative Decoder for a (32,8) Low-Density Parity-Check (LDPC) Code," *IEEE J. Solid-State Circuits,* vol. 41, pp. 2531-2540, Nov 2006.

[21] S. Hemati and A. H. Banihashemi, "Dynamics and performance analysis of analog iterative decoding for low-density parity-check (LDPC) codes," *IEEE Trans. Comm.,* vol. 54, pp. 61-70, Jan 2006.

[22] T. M. Cover and J. A. Thomas, *Elements of Information Theory.* New York: Wiley, 1991.

[23] W. Ryan, "An introduction to low density parity check codes," University of Arizona, 2001. Unpublished.

[24] T. J. Richardson, M. A. Shokrollahi and R. L. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Inform. Theory,* vol. 47, pp. 619-637, Feb 2001.

[25] Y. Mao and A. H. Banihashemi, "A heuristic search for good low-density parity-check codes at short block lengths," in *Proc. IEEE ICC,* vol.1, pp. 41-44. 2001.

[26] D. J. Mackay. Encyclopedia of sparse graph codes. Available: http://www.inference.phy.cam.ac.uk/mackay/codes/data.html

[27] J. M. Ortega and W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in several Variables.* New York: Academic Press, 1970.

[28] D. M. Young and R. T. Gregory, *A Survey of Numerical Mathematics.* Addison-Wesley Pub. Co., 1972.

[29] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations.* , vol. 16, Philadelphia: Society for Industrial and Applied Mathematics, 1995.

[30] J. D. Lambert, *Numerical Methods for Ordinary Differential Systems : The Initial Value Problem.* New York: Wiley, 1973.

[31] Amirix Systems. AP1000 - virtex-II pro development board. Available: http://www.amirix.com/downloads/ap1000.pdf

[32] J. P. Nicolle. fpga4fun.com - serial interface (RS-232). Available: http://www.fpga4fun.com/SerialInterface.html