University of Alberta

PARALLEL ENCODERS AND DECODERS FOR LOW-DENSITY PARITY-CHECK CONVOLUTIONAL CODES ON THE XINC™ MULTI-THREADED MICROPROCESSOR

by

Xin Sheng Zhou 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Spring 2008

Your file Votre référence
ISBN: 978-0-494-45917-1
Our file Notre référence
ISBN: 978-0-494-45917-1

Abstract

Low-density parity-check convolutional codes (LDPC-CCs) are a relatively new family of capacity-approaching codes. In this thesis, a novel hard-decoding Parallel Improved Bit Flipping (PIBF) algorithm is proposed. The LDPC-CC decoding processors are mapped to multiple threads on a multi-threaded microprocessor for parallel decoding. In addition, multiple bits can be decoded at another level of parallelism by using a microprocessor's built-in bit-wise parallelism within data words. A new bit flipping threshold pattern is proposed, that can achieve 2.5 dB coding gain compared to Gallager's original bit flipping algorithm at a Bit Error Rate (BER) of 10^{-4}. The decoding throughput is 24 times faster than the benchmark Min-Sum algorithm. Looping overhead was identified as a major bottleneck. Zero-overhead looping is therefore proposed as a desirable enhancement to the existing microprocessor. The emulation results show that the decoding throughput could thus be further increased by 16%.

To Hai Yan and Yi Ran

Preface

When I started my thesis project, the field of information theory had been explored for almost 60 years. The seminal work can be traced back to Claude Shannon's 1948 paper "A mathematical theory of communication" published in the Bell System Technical Journal. In that paper, Shannon showed that for any given channel bandwidth and signal power to noise power ratio (SNR), there exists a maximum bit rate at which information can be encoded and decoded without error at the receiver. Since then, information theorists have searched for code constructions whose performance could approach the Shannon Limit.

After 50 years of search, a near-optimal solution was described in the paper "Near Shannon limit error-correcting coding and decoding: Turbo codes" by C. Berrou, A. Glavieux and P. Thitimajshima on 1993's IEEE International Conference on Communications (ICC). Their result was not believed initially, with comments such as "They must have made a 3dB error" at the time, but their results were widely confirmed within the next year. Shortly after, Low Density Parity Check (LDPC) codes were re-discovered by other researchers who reported similar capacity-approaching performance.

Turbo codes and LDPC codes were researched extensively thereafter, mainly focussing on efficient and economic implementations. When I joined the VLSI lab at University of Alberta in 2005, a team led by Dr. Stephen Bates was designing the encoder and decoder for a variation of LDPC codes, called LDPC convolutional codes (LDPC-CCs). Their work on Field-Programmable Gate Array (FPGA) technology might be the first FPGA implementation of such codes in the world. The well-known Min-Sum (MS) algorithm was used and the decoder architecture was based on large amounts of memory. With the idea that the LDPC-CC decoding processors could be matched to a multi-threaded microprocessor architecture, my supervisor Dr. Bruce Cockburn proposed to implement LDPC-CCs on an 8-threaded microprocessor, called XInC, which was developed by an Edmonton-based company, Eleven Engineering Inc.

The purpose of the research is to investigate whether it is possible to efficiently implement the LDPC-CC encoding and decoding algorithms on microprocessors.

For one thing, the LDPC-CC algorithm could potentially share the host microprocessor with other algorithms and hence reduce the total product cost. In addition, the development cycle would be faster and development cost would be low compared to a high-performance Application-Specific Integrated Circuit (ASIC), with its expensive and slow design, fabrication and test development work.

However, implementations can be fully customized in hardware with ASIC technology while microprocessor architectures and instructions are standardized and slow to evolve. The algorithm core operations might be done within one clock cycle in an ASIC but require hundreds of clock cycles in a microprocessor. As a result, the algorithm could become inefficient to a degree that is totally unacceptable for important applications. In such a situation, we would have to find other solutions to increase the decoding efficiency.

As a result, both existing algorithms and microprocessors were subjected to further evaluation and optimization. Two approaches were considered in this thesis research project. First, the algorithm should fully utilize the advantages of existing multi-threaded microprocessor architectures and resources. Second, the underlying algorithm bottlenecks in microprocessor implementations should be identified so that new microprocessor hardware extensions could be added to eliminate or at least reduce such bottlenecks. To understand the algorithm and microprocessor better, an emulator for the XInC multi-threaded microprocessor was written. Through code profiling techniques, algorithm bottlenecks could be identified. Hardware extensions eliminating such bottlenecks could then be added to the emulator. The emulation results of those extensions could help to evaluate those add-on component's performance.

The research started with the implementation of the soft-decoding Min-Sum algorithm on a 12 Million Instructions per Second (MIPS) XInC-I multi-threaded microprocessor. The identical LDPC-CC decoding processors were distributed to multiple thread resources. Unfortunately, the decoding throughput was disappointingly slow at 2.2 Kbps. One reason for this low decoding throughput is that the XInC-I is aimed at simple, low-cost consumer applications and it has only 1% processing capability of today's general microprocessors. Another reason, as discussed above, is that ASICs and FPGAs can customize the algorithm in hardware while a microprocessor architecture is fixed and it takes more instruction cycles to complete decoding operations. In addition, the Min-Sum algorithm was emulated and the characteristics of the algorithm were determined by code profiling. The bottlenecks were found to be looping overhead and data movement.

The Min-Sum algorithm result showed that the XInC microprocessor might not be a good platform for LDPC-CCs. Luckily, following Dr. Stephen Bates' suggestion to consider the hard-decoding algorithm, the research continued and focused on Bit Flipping (BF) algorithms. Surprisingly, this algorithm appears to be almost

forgotten by other researchers. Only one related paper was found during the literature review. The reason might be its simple decoding method and relatively poor coding gain. Soon, bit flipping threshold patterns were found during the implementation of the original Gallager's Bit Flipping algorithm. This discovery improved coding gain by about 2.5 dB compared to Gallager's algorithm at a bit error rate of 10^{-4}. Later, the bit flipping algorithm was modified so that it could decode multiple bits at one time by using the microprocessor's built-in bit-wise parallelism without introducing much decoding complexity. For a 16-bit XInC-I multi-threaded microprocessor, the decoding throughput increases almost 12 times. The new algorithm is called the Parallel Improved Bit Flipping (PIBF) algorithm.

The PIBF algorithm was also evaluated using the XInC emulator. Almost 35% of the instructions are looping overhead and 30% of the instructions are data movement. For data movement, it seems hard to do anything since the changes would require modification of the whole microprocessor architecture. However, it should be easy to eliminate the looping overhead by adding a relatively small hardware extension: zero-overhead looping. Such a method is already widely used by many Digital Signal Processors (DSPs). This hardware extension was then added to the emulator. The emulation results showed that the decoding throughput could be further improved by 16%.

The current PIBF algorithm could decode LDPC-CCs at 56 Kbps on a 12 MIPS XInC-I multi-threaded microprocessor. Eleven Engineering has developed another 100 MIPS XInC-II microprocessor. If the PIBF algorithm is adapted to the XInC-II with 25% load, the decoding throughput could reach up to 116 Kbps. Furthermore, on a 1 GHz 64-bit multi-threaded microprocessor with 100% load dedicated to decoding, the throughput might reach up to 18.4 Mbps.

Although the current decoding throughput on XInC-I can only support relatively low-speed applications, the multi-threaded microprocessor architecture was shown to be a promising development platform for LDPC-CCs. With further research on this topic, it may be possible to further improve the decoding throughput and then possible to implement LDPC-CCs for high-speed applications on microprocessors.

Acknowledgements

I thank the Natural Sciences and Engineering Research Council of Canada (NSERC), Semiconductor Research Corporation (SRC) and the University of Alberta General Award which provided supporting funds. In addition, I am grateful to Eleven Engineering Inc. for providing two XInC demonstration boards.

I am especially grateful for the opportunity to work under the supervision of Dr. Bruce Cockburn and Dr. Stephen Bates. Thank you for having me as your student and for all the valuable advice and feedback which made the completion of this work possible.

My most heartfelt thanks go to my family, this work is as much your accomplishment as it is mine. All my love and appreciation to my wife Hai Yan and our child Yi Ran for enduring the sacrifice of my being away from home, and for their encouragement and patience throughout my degree.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

In a communication system or storage system, the physical signals that represent information or data can be contaminated by noise. Noise, as understood in electrical engineering, is viewed as unwanted deviations in a signal due to the net effect of uncertain or unknown underlying physical phenomena. Before Claude Shannon's work, it was generally believed that the fidelity of information represented using signals would be corrupted inevitably due to the presence of noise. Hence, it was believed to be impossible ever to be able to transmit or store data with 100% accuracy [1].

In 1948, Shannon published his paper "A mathematical theory of communication" [1]. In that paper, he showed that for any given channel bandwidth and signal power to noise power ratio (SNR), there exists a maximum bit rate at which information can be encoded and decoded without error at the receiver. This maximum possible error-free bit rate is also called Shannon's Channel Capacity or the Shannon Limit. However, he did not show how to reach that channel capacity in his paper. Since 1948, many researchers have tried to find suitable code constructions and their associated encoding and decoding methods that could approach the limit in an efficient way.

After 50 years of research, Turbo Codes [2] were reported to be the first capacity-approaching codes in 1993. Shortly after, Low Density Parity Check (LDPC) block-based codes [3] were found to be another class of capacity-approaching codes in 1996 [4]. A block-based code encodes the given information in fixed-sized blocks. Today, LDPC codes have been adopted in several communication standards, including the DVB-S2 standard [5] for the satellite transmission of digital television and the IEEE 802.16e standard [6] for wireless data networking services.

In 1999, A. Jiménez Felström and K. Zigangirov adapted the low-density parity-check concept from block-based codes to convolutional codes [7] and proposed Low-Density Parity-Check Convolutional Codes (LDPC-CCs) [8]. We will refer to the previous LDPC codes as Low-Density Parity-Check Block Codes (LDPC-BCs) since in those codes the information bits are encoded and decoded block-by-block. By contrast in LDPC-CCs, the entire data stream is encoded and decoded continuously without block boundaries. It was shown in [8] that LDPC-CCs have better coding gain than LDPC-BCs with the same memory capacity in the decoder circuit. The *coding gain* is defined as the reduction of the signal-to-noise ratio E_b/N_o in decibels when error control coding is used compared to uncoded data at some specified bit error rate. Furthermore, LDPC-CCs have a simpler encoder structure similar to other convolutional codes. Moreover, LDPC-CCs allow data sequences of arbitrary length to be encoded, making these codes especially attractive in situations where either (a) the data block does not fit into the fixed payload field of the available data frame, or (b) the data is produced continuously by a streaming application [9].

Initial decoder implementations for LDPC-CC have been reported on Field-Programmable Gate Arrays (FPGAs) [10] and Application-Specific Integrated Circuits (ASICs) [11]. However, implementing software-based decoders for LDPC-CCs on microprocessors could have several benefits:

1. The LDPC-CC algorithm could share the host microprocessor with other algorithms and hence reduce the total product cost.

2. The development cycle for a software decoder is faster, and the development costs and risks are lower compared to the expensive and slow fabrication process required by custom ASICs.

The project started with a conventional soft-decoding algorithm targeted for a microprocessor: the Min-Sum (MS) algorithm [12] [13]. This algorithm decodes the original encoded bits based on estimated signal reliability information derived from the received analog signal. However, when this algorithm is implemented on microprocessors, the decoding throughput is found to be rather slow. One reason for the low speed is that many operations, which could be customized in FPGAs or ASICs to be completed in one clock cycle, require several clock cycles to complete on a microprocessor. Another reason is that the reliability information associated with the input signal is represented as fixed-point numbers and the decoding algorithm is relatively complex. The third reason is that the existing algorithms do not fully exploit the advantages of existing microprocessor architectures.

As a result, another simpler decoding algorithm, the Bit Flipping (BF) algorithm [3], became a major focus of this research project. The sampled received analog signal is sent into a threshold device (comparator) that compares the signal with a reference voltage that lies between the expected '0' and '1' signals. The resulting binary '0' or '1' outputs from the comparator are used in the bit flipping decoding algorithm. In this research project, an Improved Bit Flipping (IBF) algorithm is proposed. The new algorithm uses a Bit Flipping Threshold Pattern (BFTP) to identify and then flip the suspect bits. The threshold pattern is the sequence of bit flipping thresholds used in the decoding processors when they make decisions about changing bit values to correct likely errors. At a Bit Error Rate (BER) of 10^{-4}, the IBF algorithm achieves 2.5 dB greater coding gain compared to Gallager's original

bit-flipping algorithm, which has a fixed conservative flipping threshold.

In addition, multi-threaded microprocessors are found to be potentially good candidates for LDPC-CCs. In the LDPC-CC decoder, the bit stream is processed by several identical decoding processors. A decoding processor performs a fundamental calculation in the process of detecting and correcting errors in the stream of incoming bits. These decoding processors could be mapped to multiple microprocessor threads and run in parallel. Moreover, bit flipping algorithms use simple bit manipulations, such as AND, OR, XOR. Microprocessors could exploit this fact by processing multiple bit operations in one instruction at the word level. For a 16-bit microprocessor, the decoding throughput could thus potentially improve 16 times. In practice, the speed-up will be less than 16 times since not all parts of the calculation can be made parallel (See Amdahl's Law [14]). This proposed bit-parallel bit flipping algorithm is called the Parallel Improved Bit Flipping (PIBF) algorithm.

The PIBF algorithm is analyzed using a microprocessor emulator. The emulator was written in the object-oriented programming language C++ at the beginning of this project. The multi-threaded microprocessor used in the research is called XInC [15]. This part was developed by Eleven Engineering Inc. in Edmonton, Canada for low-cost consumer products. Through code profiling, two main bottlenecks of the algorithm were found: looping overhead and data movement. A hardware extension, called zero-overhead looping, is proposed to eliminate the looping overhead. Using this extension, the decoding throughput could be increased further by about 16%.

The decoding throughput of the PIBF algorithm on a 12 Million Instructions Per Second (MIPS) XInC-I multi-threaded microprocessor was found to be 56 Kbps with an average of 27 instruction cycles required to decode each incoming information bit. If a 1-GHz 64-bit microprocessor with similar bit-parallel instruction is used, the decoding throughput might increase up to 18.4 Mbps, which could well

support some broadband access services such as Digital Subscribe Line (DSL) Internet access.

During this research project, Gallager's original bit flipping algorithm [3] was improved significantly. We believe that the multi-threaded microprocessor has now been shown to be a promising development platform for LDPC-CCs.

1.2 Thesis Organization

The remainder of this thesis is organized as follows.

In Chapter 2, background information on LDPC codes is presented. The concepts of block codes, convolutional codes, soft decoding and hard decoding, and iterative decoding are defined. Then several published encoding and decoding algorithms for LDPC-CCs are reviewed. At the end of this chapter, the architecture of the XInC-I multi-threaded microprocessor is presented.

In Chapter 3, the use of a Bit Flipping Threshold Pattern is proposed. The Improved Bit Flipping algorithm (IBF) is then described. Several threshold patterns are simulated and the pattern $(3-2)^+$ is found to be the best for a benchmark (128,3,6) LDPC-CC. In addition, the Parallel Improved Bit Flipping (PIBF) algorithm, which exploits microprocessors' built-in bit-wise parallelism to significantly increase the data throughput, is also described.

In Chapter 4, the architecture of the LDPC-CC encoding and decoding algorithm on the XInC multi-threaded microprocessor is presented. Implementations of the Min-Sum algorithm, the IBF algorithm and the PIBF algorithm on the XInC microprocessor are discussed. An evaluation of the trade-off between the coding gain and computing complexity is provided at the end of the chapter.

In Chapter 5, the LDPC-CC decoding algorithms are analyzed using a software-based XInC microprocessor emulator. The algorithm bottlenecks are identified as being looping overhead and data movement operations. A new hardware extension,

called zero-overhead looping, is proposed and emulated. The decoding throughput improvement of this proposed hardware enhancement is presented.

The thesis ends with discussions of possible future research directions and conclusions in Chapter 6.

Chapter 2

Background

2.1 A Brief History of Error Control Coding

In communication systems, information must be transmitted reliably from the transmitter end to the receiver end through a channel. However, due to signal distortions and inevitable noise sources present in the communications channel, the quality of the received signals can be degraded to the point where bit errors (e.g. bits that have been flipped from 0 to 1, or from 1 to 0) occur in the received bits. In order to detect and correct these errors, redundant information (i.e. check bits) can be added to the intended data (i.e. information bits) at the transmitter before the signal is sent into the channel [16]. The known relationship between the information and check bits is used at the receiver to detect and hopefully correct any errors in the information bits. This process is called channel coding in communication systems since the purpose of the coding is to combat the channel noise. The more general term "error control coding" is also used since the technique can not only correct the errors in a communication system, it also could be used to correct errors in other applications such as magnetic data storage systems.

A typical communication system is shown in Figure 2.1. The information bits first go through a source encoder. The purpose of the source encoder is to encode the information bits in some way that maximizes the information density contained

in the encoded bits by removing data redundancy. Typically, the information bits are segmented into small groups of bits and each group represented together as an information symbol from a non-binary alphabet. For example, two information bits could be grouped and represented as one of the four symbols: S_1='00', S_2='01', S_3='10', S_4='11'. The source encoder then encodes according to the probability that each information symbol appears in the symbol series. Symbols with large probability are represented with shorter codewords and symbols with small probability are represented with longer codewords. For example, the symbols S_1, S_2, S_3 and S_4 might get coded as '0', '10', '110' and '111', respectively. For a given information bit stream, the optimum encoded bit stream would have minimal length. From a statistical view, each bit in an efficiently encoded bit stream would have equiprobability of being '0' and '1'. The encoded information bits are then sent into the channel encoder. The function of the channel encoder is to add redundant bits in order to combat the effects of channel noise. After that, the output of channel encoder is imposed onto analog carrier signals by the modulator and sent through the channel. Within the channel, the signal is distorted and contaminated by noise and other environmental factors. When the signal is received, the receiver first samples the analog signal. The sampled signal could then be sent into a threshold device (comparator) to get binary digits '0' or '1' first, and then sent into a channel decoder. Alternatively, the sampled signals can be sent into the channel decoder directly as analog or sampled digital signals. If the channel decoder uses non-binary signals, it is called a soft-decoder. On the other hand, if the channel decoder processes binary digits, it is called a hard-decoder. The function of the channel decoder is to correct erroneous bits caused by channel noise. The decoded bits are finally forwarded to the source decoder for source decoding. After source decoding, we hope that the decoded bits are the same as the original information bits.

In the field of information theory, the information content of a symbol can be

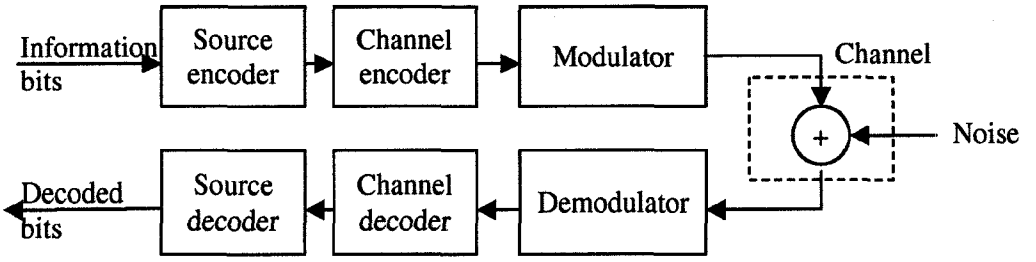

Figure 2.1: Communication Systems

defined numerically as the logarithm of the probability of the symbol as follows [1]:

$$I_j = \log_2(\frac{1}{P_j})\ bits$$

where P_j is the probability of the j-th symbol and I_j is the information carried by j-th symbol measured in bits. The average information content is defined to be

$$H = \sum_{j=1}^{m} P_j I_j = \sum_{j=1}^{m} P_j \log_2(\frac{1}{P_j})\ bits$$

where m is the total number of symbols. The average information content of a source of symbols is also called the entropy.

Before Claude Shannon, researchers thought that information could not be received with 100% accuracy due to inevitable signal distortion and contamination by noise. In his landmark 1948 paper [1], Shannon showed that for any given channel bandwidth and signal power to noise power ratio (SNR), there exists a maximum bit rate by which information can be encoded and decoded without error at the receiver.

The Shannon-Hartley Theorem [1] further states that for the special case of a signal plus Additive White Gaussian Noise (AWGN), the channel capacity C (in bits/second) is given by,

$$C = B \times \log_2(1 + \frac{S}{N})\ bits/second$$

where B is the finite channel bandwidth in Hertz (Hz) and S/N is the signal power to noise power ratio at the input to the receiver.

Additive White Gaussian Noise is widely used in communication systems as an ideal noise model. The noise component in physical systems is formed as the net effect of many different noise phenomena. According to the Central Limit Theorem, for statistically independent underlying noise sources, the probability distribution of the net total noise tends to become Gaussian as the number of statistically independent noise sources is increased without limit, regardless of the probability distribution of the noise sources being sampled, as long as the noise signals have a finite mean and a finite variance. Many real noise sources thus have the characteristics of a Gaussian distribution. For example, for thermal noise, the number of electrons in a resistor is very large and their random motions inside the resistor are statistically independent of each other, and hence the net produced thermal noise is Gaussian-distributed. The probability density function of Gaussian noise with zero mean can be written as,

$$\frac{1}{\sigma\sqrt{2\pi}}\exp^{-\frac{x^2}{2\sigma^2}}$$

where σ^2 is the variance or the average power of the noise signal. So-called white noise is a noise with a flat (i.e., frequency-independent) power spectral density spectrum. The power spectral density of white noise is defined as,

$$S_w(f) = \frac{N_o}{2}$$

where N_o denotes the average noise power per Hertz. The factor 1/2 has been included to indicate that half the power is associated with positive frequencies and half with negative frequencies [17]. In practice, white noise must be limited in bandwidth to avoid describing noise with infinite power.

The performance of an error control coding scheme is usually evaluated by expressing the probability that the decoded bits will be incorrect (the bit error rate) as a function of the signal-to-noise ratio E_b/N_o. Here E_b is the received signal energy over a one bit time interval. E_b/N_o is useful when comparing the bit error rate performance of different digital modulation schemes or channel coding schemes without needing to take the bandwidth into account.

According to Shannon's theorem, the code rate cannot exceed the channel capacity if we want to achieve error-free communication. The code rate is given by $R=K/N$ where it is assumed that for every K information bits, the encoder generates a total of N bits of data, of which N-K are redundant. For error-free communication over an AWGN channel,

$$\begin{aligned} R = \frac{1}{T_b} &\leq B\log_2(1+\frac{S}{N}) \\ &= B\log_2(1+\frac{E_b/T_b}{N_oB}) \end{aligned}$$

$$\frac{E_b}{N_o} \geq \frac{2^{\frac{1}{T_bB}}-1}{\frac{1}{T_bB}}$$

where R is the code rate and T_b is the time duration of one bit. In the ideal limiting case when unlimited channel bandwidth is available,

$$\frac{E_b}{N_o} \geq lim_{B->\infty}\frac{2^{\frac{1}{T_bB}}-1}{\frac{1}{T_bB}} = ln(2) = -1.59dB,$$

which is the absolute minimum E_b/N_o required for error-free communication. This limit is higher in real channels with finite bandwidth.

Although Shannon determined a precise upper bound on the channel capacity, he did not give constructions of codes that can actually achieve the channel capacity. Finding good code constructions that could approach the Shannon's channel

capacity has been an ongoing challenge in coding theory research field ever since [18].

One strategy for approaching the channel capacity for block codes is to use a large block length. Under that condition, a random coding scheme, which randomly picks codewords, is good enough to achieve the limit. It has been shown that such codes are likely to approach the capacity limit if the block length is indefinitely large [7]. However, this random coding scheme cannot be realized since it requires exponentially large memory space to store the mapping table that is required to map blocks of information bits to codewords. In addition, the time to search this table would also be exponentially large. For a moderate block length of 30, the table already has more than 1 billion entries. However, the block length normally requires more than 10^4 bits to approach the channel capacity within 0.6 dB [19].

As a result, practical codes have to use some pre-defined rules to construct the codewords in order to eliminate the mapping table. The encoder and decoder can then be operated based on these rules.

One simple method is to encode the information bits by simply repeating bits or blocks of bits two or more times using a repetition code. When a bit is flipped by an error, the error could be corrected according to the majority rule. For example, if a block of information bits is "101", we then may encode the block as "101 101 101". If one of the bits is in error and we receive "101 111 101", the decoder could deduce that "101" is the correct information bits since "101" is the result of a simple majority consensus.

Another widely used method is the parity check. A check bit is a bit that is added to a block of information bits to indicate whether either the 0's and 1's within that block is an even or odd number. If a single bit is flipped through the channel, the prearranged parity check constraint would be violated and the error would be detected by the fact that a parity check recomputed at the receiver would fail. In

[7], P. Elias showed that a typical parity check code with large block length used on a BSC channel could achieve a decoded bit error rate almost as small as the best possible code if the code rate is between what he called the critical rate and the channel capacity. As a result, if we could generate random-like long codewords with multiple parity checks, then the channel capacity can be approached.

Consequently, Elias sought random-like codes with special structure that permitted simple implementation without sacrificing the code's error correcting performance. Gallager, as Elias's student, proposed Low Density Parity Check (LDPC) codes [3] in 1962 motivated by the search for Elias's random-like codes. The resulting codes are now usually called Low Density Parity Check Block Codes (LDPC-BCs). The LDPC codes use multiple parity checks. In a LDPC-BC the density of 1's in the parity check matrix **H** (defined later) is low, that is, much less than 50%. In his paper, Gallager showed the distance property of LDPCs and gave a probabilistic decoding algorithm with promising empirical performance. However, LDPCs were then generally forgotten due to the assumption that the subsequently developed concatenated codes were probably superior for practical purposes [4]. The computational load of the decoding algorithm of LDPC-BCs was also considered to be impractically high.

After many years of search, the first capacity-approaching code, the so-called Turbo code [2], was described in a paper presented in 1993. Shortly after, in 1996, D. Mackay and R. Neal [4] found that LDPC-BCs with very sparse parity check matrix with an approximate probabilistic decoding algorithm could also approach Shannon's channel capacity. They later found that Gallager's work in the early 1960s on low-density parity-check codes should be credited as having proposed the first such codes.

In 1999, A. Jiménez Felström and K. Zigangirov applied the idea of the low-density parity-check matrix to convolutional codes, and then proposed Low Density

Parity Check Convolutional Codes (LDPC-CCs) [8]. In their paper, they gave a prototype hardware decoder which can be implemented conveniently by a cascade of physically identical decoding processors [8].

2.1.1 Block Codes

In a parity check operation, check bits can be generated by simple logical exclusive OR (XOR) operations. Table 2.1 shows the logical truth table of the XOR operation for two inputs.

Table 2.1: Exclusive OR (XOR) Truth Table

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

The XOR operation is often denoted using the symbol $\oplus$. As an example, the check bit for a block of four information bits "0110" is $0 \oplus 1 \oplus 1 \oplus 0 = 0$. Note that $\oplus$ is associative, so $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ for three binary values a, b and c. The information bits together with the one check bit "0110+0" are then transmitted together through the channel. At the receiver, the decoder performs XOR operations for both information bits and check bits. If the result is '0', the parity check constraint is obeyed. If the result is '1', the parity check constraint is failed. For example, if the first bit is in error, then the recomputed parity check result at the receiver would be $1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = 1$, and hence we know that the parity check constraint is failed and an error is detected. However, we still do not know which bit is in error now. Furthermore, if two bit errors occur, the errors cannot be detected since the parity check constraint would be obeyed. To unambiguously determine which single bit is in error, each bit could participate in multiple parity check operations. If a bit is in error, this fact would be indicated several times. For example,

in (7,4) Hamming codes, the codeword length is 7 bits. Four bits are information bits and remaining three bits are check bits. Three check bits are generated using the following parity checks: $P_1 = I_1 \oplus I_3 \oplus I_4$, $P_2 = I_1 \oplus I_2 \oplus I_3$, $P_3 = I_2 \oplus I_3 \oplus I_4$. When the coded bits are received, the decoder checks the following parity check constraints: $I_1 \oplus I_3 \oplus I_4 \oplus P_1$, $I_1 \oplus I_2 \oplus I_3 \oplus P_2$, $I_2 \oplus I_3 \oplus I_4 \oplus P_3$. These parity checks should all evaluate to the value 0. In the above (7,4) Hamming code example, if I_1 is received in error, then the second and third parity check constraints would fail and the first parity check constraint would be obeyed. Two failed constraints indicate that I_1 or I_3 has the most possibility of being in error. However, the obeyed constraint indicates that I_3 is probably correct. From all of the three parity check results, we could conclude that I_1 has the most probability in error. To correct this error, we could flip that bit from '1' to '0' or from '0' to '1' .

The set of all parity check constraints can be represented by a parity check matrix **H**. Each row of **H** corresponds to one parity check constraint. The 1's in the same row indicate those bits are involved in one parity check constraint. The (7,4) Hamming code parity check matrix is shown below.

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

There are 3 rows in this matrix, which means 3 parity check constraints should hold in each 7-bit block. Each row has four 1's, which identify which 4 of the 7 bits are involved in each parity check operation.

The encoded bits can be represented by a vector $\boldsymbol{x}$.

$$\mathbf{x} = (x_0, x_1, \ldots x_{N-1})$$

The parity check constraints can be represented completely with the following matrix equation.

$$\mathbf{x}\mathbf{H}^T = \mathbf{0}$$

where $\mathbf{H}^T$ denotes the transpose of matrix $\mathbf{H}$.

In addition, a generator matrix can be used for generating all possible codewords. If the generator matrix is $\mathbf{G}$ and the information bit vector is $\mathbf{c}$, then the generated codeword $\mathbf{w}$ is given by the following matrix equation.

$$\mathbf{w} = \mathbf{c}\,\mathbf{G}$$

The equivalent generator matrix $\mathbf{G}$ of the above (7,4) Hamming code is,

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

Block codes can be represented by a Tanner Graph [20]. The decoding operations associated with the received bits in a codeword are abstracted as so-called *variable nodes*. The decoding parity check operations are abstracted similarly as *check nodes*. The edges connecting variable nodes with check nodes indicate the parity check constraints between the bits. The Tanner Graph for the (7,4) Hamming code is shown in Figure 2.2. In the figure, variable nodes are represented as square nodes and check nodes are represented as circle nodes.

The *code distance* is defined as the minimum number of different bits in corresponding position between any two codewords. To maximize the coding gain, codes should be constructed to maximize the code distance. This means, in effect, that the number of bit errors that would be required to change one codeword into another valid codeword is maximized. Given a code with code distance N, $(N-1)/2$ errors could be corrected by seeking the closest correct codeword to any given error

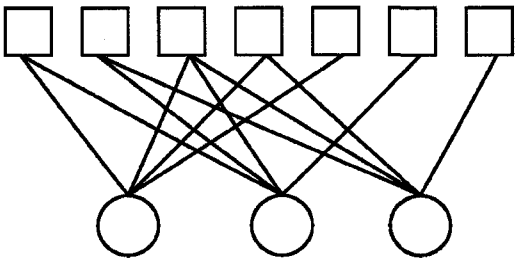

Figure 2.2: (7,4) Hamming Code Tanner Graph

word. For the (7,4) Hamming code, there are a total of $2^7 = 128$ words. From that space, only 8 words are chosen as valid codewords. The choice of these codewords uses the following approach: each valid codeword and 7 other words whose distance to that valid codeword is 1 are grouped. One example of (7,4) Hamming code codewords is shown in Table 2.2. The code distance between any two codewords can be verified to be equal to or greater than 3. If one bit is in error during the transmission, this error can be detected and corrected. For example, if the encoded codeword is "0000000" and one received bit is in error, the received bits would be any of "0000001", "0000010", "0000100", "0001000", "0010000", "0100000" or "1000000". The most possible codeword is "0000000" since the code distance between the erroneous received bits and "0000000" is one. However, we could not correct the received bits if two bit errors occur. For example, if "0000011" is received, then the received bits would be wrongly corrected as "0100011" rather than "0000000" since the code distance between "0100011" and "0000011" is one and the code distance between "0000000" and "0000011" is two.

In order to increase the code distance for the codes of the same code rate $R=K/N$, we generally need to increase the block length N. The number of valid codewords is 2^{NR}. When N increases to infinity, the ratio of the number of valid codewords to the total number of words $2^{-N(1-R)}$ goes to 0. As a result, the coding gain is increased as the average code distance between codewords is increased.

Like a Hamming code, low-density parity-check codes use multiple parity checks

Table 2.2: A (7,4) Hamming Code

Source	Codeword
0000	0000000
0001	0001101
0010	0010111
0011	0011010
0100	0100011
0101	0101110
0110	0110100
0111	0111001
1000	1000110
1001	1001011
1010	1010001
1011	1011100
1100	1100101
1101	1101000
1110	1110010
1111	1111111

and the parity check constraints are specified by a parity check matrix. The characteristic of a LDPC parity check matrix is that the 1's of the matrix have a relatively low-density compared to the 0's. For a (N,J,K) LDPC-BC, the matrix has N columns. Each column has J ones and each row has K ones. This means that each bit involves J parity check constraints and each parity check constraint involves K bits. Equivalently, in the Tanner graph, each variable node has J edges connected to check nodes and each check node has K edges connected to variable nodes. The value of J is called the variable node degree and K is called check node degree. The benefit of LDPC-BC's low-density parity check matrix is that the decoder structure is simpler. Each '1' in the parity check matrix implies the presence of an input to a network of XOR gates. A (20,3,4) LDPC parity check matrix is shown below.

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

Note that when we encode, the low-density parity-check matrix would need to be converted to its equivalent generator matrix, which normally is no longer low-density. As a result, the encoder computation complexity is typically $O(N^2)$, using big O complexity notation [21]. Finding more efficient encoding algorithms and code constructions to encode LDPC in linear time continues to be an active research area. Several methods are reported in [22].

2.1.2 Convolutional Codes

Convolutional codes first appeared in P. Elias's 1955 paper [7]. In general, convolutional code encoder structure is considered to be simpler than block code encoder structure since convolutional codes are encoded continuously while block codes are encoded block-by-block.

Figure 2.3 shows a traditional rate 1/2 convolutional code encoder. Each check bit is generated from the current information bit and two previous information bits stored in memory registers. The information bits and check bits are then interleaved by a multiplexer at the output.

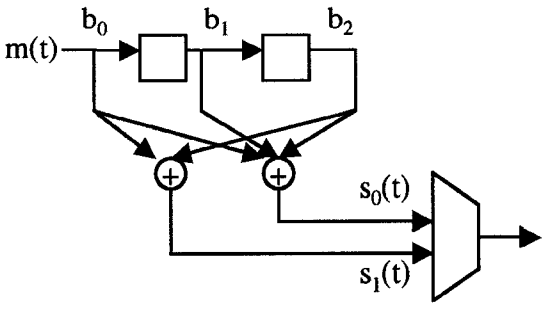

Figure 2.3: Rate 1/2 Convolutional Code

The transpose of the parity check matrix of a (M,J,K) LDPC-CC can be represented as follows:

$$H^T_{[0,n]} = \begin{bmatrix} h_1^{(0)}(0) & h_1^{(1)}(1) & \cdots & h_1^{(M)}(M) & 0 & \cdots & \\ h_2^{(0)}(0) & h_2^{(1)}(1) & \cdots & h_2^{(M)}(M) & 0 & \cdots & \\ 0 & h_1^{(0)}(1) & \cdots & \cdots & h_1^{(M)}(M+1) & & \\ \vdots & h_2^{(0)}(1) & \cdots & \cdots & h_2^{(M)}(M+1) & & \\ & 0 & \ddots & \cdots & \vdots & & \\ & & \ddots & & & 0 & \\ & & & & & h_1^{(M)}(n) & \\ & & & & & h_2^{(M)}(n) & \\ & & & & & \vdots & \\ & & & & & 0 \quad h_2^{(0)}(n) & \end{bmatrix}$$

The parity check constraint is based on a sliding memory window of size $2 \times M$. The value of a given parity check bit is a function of the $2M-1$ preceding bits in the coded bit stream. The number of 1's in each row is J, which means each bit is involved in J parity check constraints. The number of 1's in each column is K, which means each parity check constraint involves K bits. The parity check matrix is extended in a continuous way to accommodate the arriving stream of bits.

2.1.3 Survey of Decoding Algorithms

Many hard-decoding algorithms have been developed [18]. In a hard-decoding algorithm, the received analog signal is sent into a threshold device (comparator) and the binary output '0' or '1' is used for decoding in an entirely algebraic calculation.

In 1967, an efficient hard-decoding algorithm for convolutional codes, the Viterbi algorithm, was proposed [23]. One weakness of the Viterbi algorithm is that the decoding complexity increases exponentially when the total number of memory registers is increased. As a result, the total number of memory registers must in general be under 10.

Note that the received analog signal contains valuable probabilistic reliability information. For example, if '1' is transmitted as +1 V and '0' is transmitted as -1 V, then when the received analog signal is sampled as +0.8 V, we could say that the bit is more likely to be '1' than to be '0'. However, with very small probability the signal could in fact be a '0' that has been corrupted with a large amount of noise at the bit sampling instant. That reliability information is discarded when the sampled measurement goes through the threshold device. If the reliability information is exploited in the decoding algorithm, that algorithm is called a soft-decoding algorithm.

To distinguish the received bits, the binary bit from the threshold device is called a hard bit and a bit signal representation that includes reliability information is called a soft bit.

Another important concept in the LDPC decoding algorithm is iterative decoding. For each iteration, some of the hard-bits are corrected in hard-decoding algorithm, or the probabilities of the soft-bits as '0' or '1' are adjusted and eventually strengthened. The result of one iteration is used in the next iteration for further decoding. The received bits are hence decoded iteratively and the coding gain should be gradually increase until some iteration limit is reached.

2.2 Low Density Parity Check Codes

2.2.1 The LDPC-CC Encoder Structure

Figure 2.4 shows a (128,3,6) LDPC-CC encoder structure. The LDPC-CC encoder is similar to other convolutional code encoder structures. For a rate 1/2 (M,J,K) LDPC-CC encoder, a memory of length $2 \times M - 1$ is used. M previous information bits $V_1(t)$ to $V_1(t-M)$ and *M-1* previous check bits $V_2(t)$ to $V_2(t-M)$ are stored in this memory. The memory is organized as a first-in first-out (FIFO) queue. The newest information and check bits are pushed into the queue tail location and the oldest bits are removed from the queue head location. Five bits are chosen from the memory queue to generate the next check bit. The positions of the 5 chosen bits in the memory queue are determined by a position table, which is derived from the LDPC-CC parity check matrix. The position table is used in a round-robin fashion and current entry is indicated by the position table pointer. The check bits $V_2(t)$ and information bits $V_1(t)$ are interleaved in strictly alternating order as the encoder output.

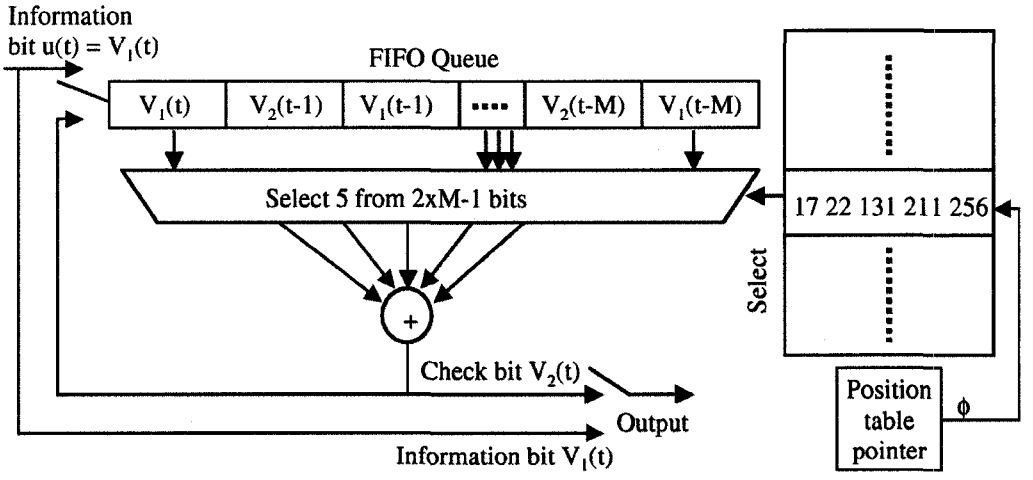

Figure 2.4: LDPC-CC Encoder Structure

2.2.2 The LDPC-CC Decoder Structure

2.2.2.1 Overview

The structure of the LDPC-CC decoder is shown in Figure 2.5. The LDPC-CC decoder is composed of several identical decoding processors for iterative decoding. Inside each decoding processor, multiple FIFO memory queues are used to store recently received bits. The received bit, either a soft bit or a hard bit, is sent into the first decoding processor's queue. Within each decoding processor, there is one check node calculation followed by one variable node calculation. The check node retrieves the bits for each parity check constraint from the memory queue according to the same position table entries used in the LDPC-CC encoder. The arrows in Figure 2.5 show in simplified form the movement of data read and then written by the check node CN. The six sources of bits (upper arrowheads) change from clock cycle to clock cycle, and can select elements from any of the three lower FIFO queues. The check node then checks this constraint. For hard bits, the result is simply whether the parity check constraint is obeyed or failed. For soft bits, the result is updated (and hopefully more accurate) reliability information for each bit. Then the results are stored back to the memory queue. As the new received bits are pushed into the tail location of the queue, the existing bits inside the queue are shifted one position towards to the queue head. By the time that the bit has reached the queue head, it has been checked by the check node J times. The variable node integrates the original received bit value and J check node results (i.e., sum them up). The new hard or soft bit is then forwarded to the next decoding processor for next iterative decoding. After the last decoding processor, the hard-decoding algorithm sends out the hard bits directly. For soft-decoding algorithm, the soft bits are forwarded by the last decoding processor to a hard decision device. The function of the hard decision device is to convert the soft bit into hard bit in binary format using a threshold comparator.

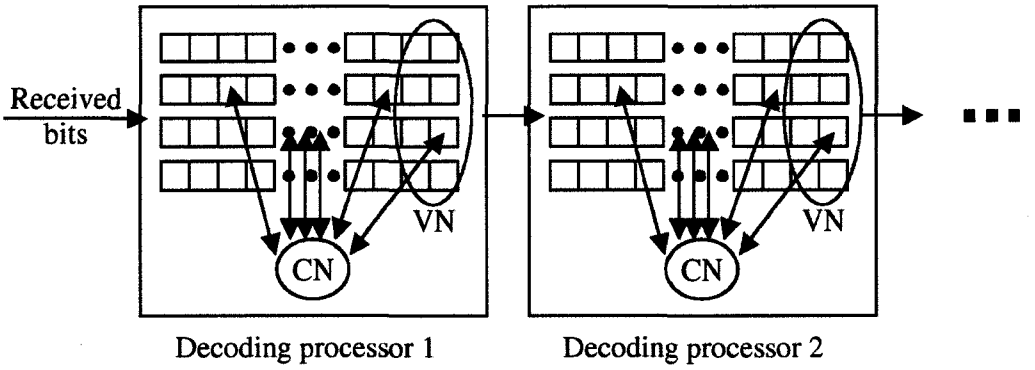

Figure 2.5: LDPC-CC Decoder Structure

There are several different decoding algorithms that vary in the operation of the check nodes and variable nodes. The Sum-Product (SP) algorithm [24] is a well-known soft-decoding algorithm. The soft bits are represented typically by fixed-point numbers. The operation in the check node requires both the hyperbolic tangent function and the inverse hyperbolic tangent function [24]. The computational complexity is large for the hyperbolic function and hence it is inconvenient in hardware implementations. But it could get better coding gain compared to other LDPC decoding algorithms. To avoid the hyperbolic function calculation, a lookup table might be used.

The Min-Sum (MS) algorithm [12] is another alternative simplified, soft-decoding algorithm that is weaker at detecting errors compared to the SP algorithm. In the MS algorithm, the hyperbolic function of SP algorithm is replaced by the minimum function and sign function. The simulation results show that the coding gain of the MS algorithm is only 0.2 dB less than that of the SP algorithm in Figure 3.1. However, the complex hyperbolic function is removed. Several improved MS algorithms that aim to recover part of the loss incurred by the Min-Sum algorithm are proposed in [25] [26] [27].

Gallager's Bit Flipping (BF) algorithm [3] is a hard-decoding algorithm that appeared in his 1962 thesis. This algorithm only uses hard bits. The reliability

information in the analog bit signals is not considered. The operation in its check node is a simple exclusive OR (XOR) operation. Since the BF algorithm omits the reliability information, it requires a relatively high signal-to-noise (SNR) ratio compared to the soft-decoding algorithm to achieve the same bit error rate as the SP or MS algorithms. However, the BF algorithm used with a LDPC-CC could still superior to other error control codes based on hard-decoding algorithms. In addition, the hard-decoding algorithm is generally simpler and faster than soft-decoding algorithms and the hardware implementation requires less circuitry and hence less power. For some high-speed communication systems, such as 100 Gbps or 1 Tbps optical system, the soft-decoding algorithms may not be fast enough and a hard-decoding algorithm might be the only choice.

In the following subsections, the Bit Flipping algorithm, the Sum-Product algorithm and the Min-Sum algorithm are briefly reviewed.

2.2.2.2 Bit Flipping Decoding Algorithm

Gallager's Bit Flipping algorithm is a hard-decoding algorithm. The check node operation is based on simple exclusive OR (XOR) operations. Each bit is associated with an error counter that records the total number of its failed parity check constraints. If the bit's error counter value is greater than the bit flipping threshold b, it would be flipped. The bit flipping procedure is shown in Figure 2.6. We assume in this example that the bit flipping threshold is 3. Most of the bits have no error. One of the bits has 3 failed parity check constraints. As a result, that bit would be flipped.

Gallager's BF algorithm was first applied to LDPC-BC codes. This algorithm can be described as follows:

Step 1: Compute all the parity-check constraints in the block. If all of the parity check constraints are obeyed, then stop decoding.

Step 2: Record the number f_i of failed parity-check constraints for each bit i

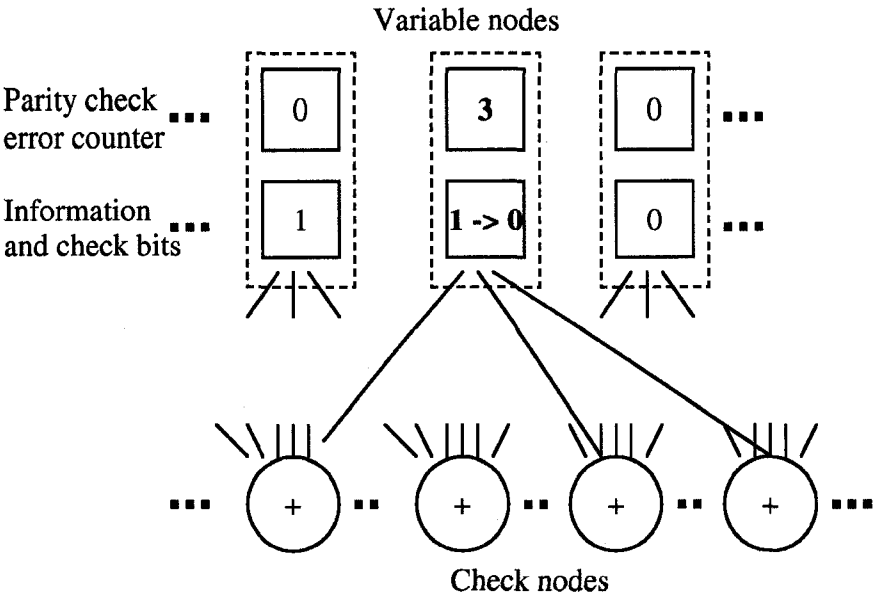

Figure 2.6: LDPC-CC Bit Flipping Procedure

during the parity check. When one parity check constraint has failed, all the error counters associated with the bits involved in this constraint are incremented by 1.

Step 3: Consider each bit in turn. If the error counter value of a particular bit exceeds the threshold, then flip that bit.

Step 4: Repeat steps 1 to 3 iteratively until all of the parity check constraints are satisfied in step 1 or until a predefined maximum iteration number is reached.

One improvement to the BF algorithm is proposed in [28]. The authors found that many correct bits are wrongly flipped during decoding and this problem degrades the coding gain. If we only flip the suspect bits with a pre-defined probability $p < 1$, then fewer correct bits would be erroneously flipped during one iteration and hence improve the coding gain.

2.2.2.3 Sum-Product Algorithm

The Sum-Product algorithm is an iterative soft-decoding algorithm based on belief message propagation [29]. The soft bits are represented using fixed-point numbers.

At the exit of the decoder, the soft-bits are converted to hard-bits by a hard decision device. In this research, the Sum-Product algorithm is used as a benchmark algorithm for performance comparison purposes since it has the best error-correcting performance.

Assume that we want to transmit a binary bit stream *b(k)* over a channel using Binary Phase Shift Keying (BPSK), where k is a discrete time index. Thus the amplitude of the modulated signal is +1 V when $b(k) = 0$ and is -1 V when $b(k) = 1$. The stream of transmitted bit voltages $t(k)$ can then be written as

$$t(k) = 1 - (2 \times b(k)).$$

For a sequence of noise samples $n(k)$, the sampled received bit stream *r(k)* can be expressed as

$$r(k) = t(k) + n(k).$$

To rewrite the above equation with *n(k)*, when a '0' is transmitted,

$$n(k) = r(k) - t(k) = r(k) - 1V.$$

When a '1' is transmitted,

$$n(k) = r(k) - t(k) = r(k) + 1V.$$

Several methods could be used to represent the reliability information of the received signal. One way is to use the amplitude of the received signal directly as the reliability measure. In this case, when the sampled signal amplitude is +1 V, the transmitted bit is most probably a '0'. When the sampled signal amplitude is -1 V, the transmitted bit is most probably a '1'. Another way is to use the Likelihood Ratio (LR) of the received signal as the reliability measure. When the received bit is measured as x V, the conditional probability of the transmitted bit as '0' is represented as,

$$P(t(k)=1|r(k)=x)$$

and the conditional probability of the transmitted bit as '1' is represented as,

$$P(t(k)=0|r(k)=x)$$

The LR is defined as the ratio of the above two conditional probability, represented as,

$$\begin{aligned} L(x) &= \frac{P(t(k)=1|r(k)=x)}{P(t(k)=-1|r(k)=x)} \\ &= \frac{P(n(k)=x-1)}{P(n(k)=x+1)} \end{aligned}$$

Assume that the channel is an Additive White Gaussian Noise (AWGN) channel. The noise has a zero mean with a power of σ^2. The probability density function of the AWGN channel is given by [1]:

$$p(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x)^2}{2\sigma^2}}.$$

Hence, the LR can be represented as

$$\begin{aligned} L(x) &= \frac{P(n(k)=x-1)}{P(n(k)=x+1)} \\ &= \frac{e^{-\frac{1}{2}(\frac{x-1}{\sigma})^2}}{e^{-\frac{1}{2}(\frac{x+1}{\sigma})^2}} \\ &= e^{\frac{4x}{2\sigma^2}} \end{aligned}$$

To remove the awkward exponential operation in the LR expression, the Log-Likelihood Ratio (LLR) is frequently used as the received signal reliability measure. The LLR over AWGN channels with BPSK signalling is defined as

$$l(x) = ln(L(x)) = \frac{4x}{2\sigma^2} \tag{2.1}$$

Assume that the LLR is used to represent soft bits in our SP algorithm. The SP algorithm for LPDC-BCs can then be briefly summarized as follows:

Step 1: Store the initial LLR value of each sampled bit in the variable nodes as lv_i^0, where the subscript i is the variable node label number and the superscript is the iteration number.

Step 2: The LLR values in the variable nodes are then sent to the check nodes. The following equations are evaluated in the check nodes,

$$lc_i^k = 2 \times tanh^{-1}(\prod_{j/i} tanh(\frac{lv_j^k}{2})),$$

where the notation $\prod_{j/i}$ means a repeated product where the product terms include all of the LLR values from variable nodes except the value of the i-th variable node. The check node results are then sent back to the associated variable nodes. Figure 2.7 shows the inputs and outputs to the SP algorithm check node operation.

Step 3: The variable node calculates new LLR values based on the check node results using the equation

$$lv_i^{k+1} = l_i^0 + (\sum_{j/i} lc_j^k),$$

The results are then sent back to check nodes for the next iteration. The calculation involves adding the variable node's initial LLR value with the results from all associated check nodes except the value of its own. For the last iteration, the results are sent into a hard decision device. Figure 2.8 shows the input and output parameters that are involved in the variable node operation.

Step 4: Repeat steps 2 and 3 for iterative decoding until the pre-determined maximum iteration number is reached.

Step 5: In the hard decision device, the initial LLR value for the bit is added to all the newest variable node results from step 3. Then a threshold device is used. If

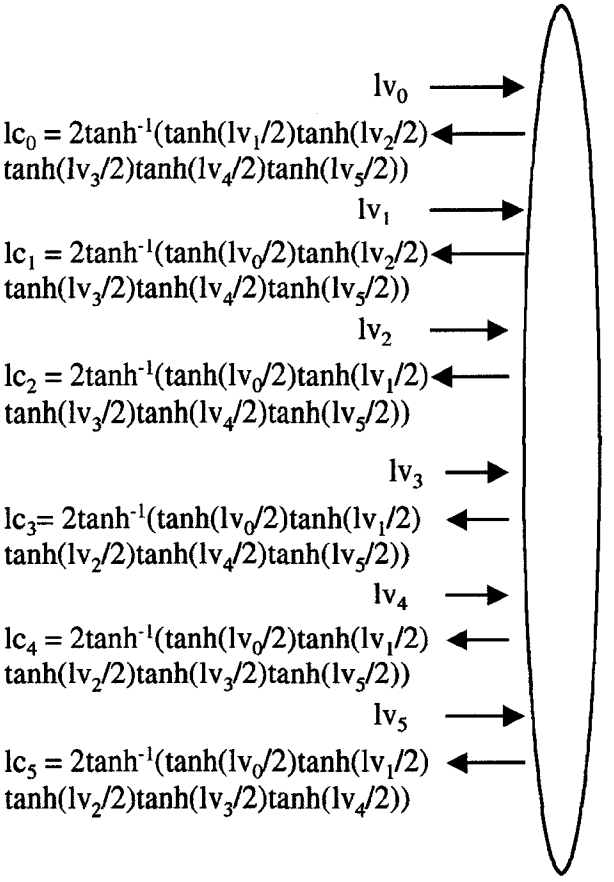

Figure 2.7: Sum-Product Algorithm Check Node Parameters for a Degree-6 Check Node

the sum result is greater or equal to 0, the soft-bit is decoded to '0'. Otherwise, it is decoded to '1'.

2.2.2.4 Min-Sum Algorithm

The Min-Sum (MS) algorithm [12] is another alternative soft-decoding algorithm. Compared to the Sum-Product algorithm, the inconvenient hyperbolic functions in the check nodes are replaced by the minimum and sign functions. The simplified operation in the Min-Sum check node is as follows:

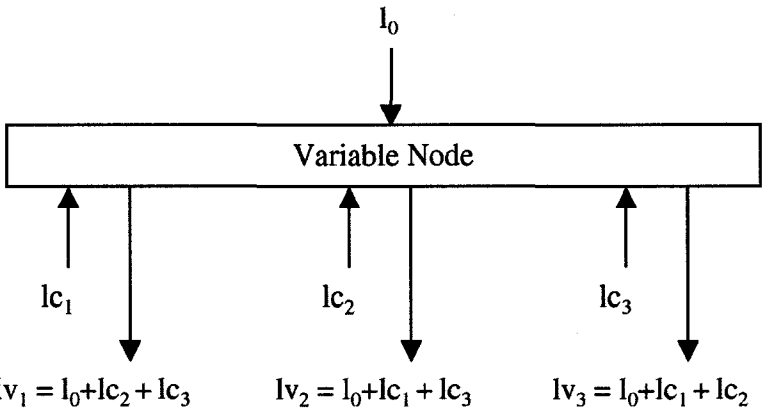

Figure 2.8: Sum-Product Algorithm and Min-Sum Algorithm Variable Node Operation

$$lc_i^k = \left[\prod_{j/i} sgn(lv_j^k)\right] (\min_{j/i} |lv_j^k|), \tag{2.2}$$

where sgn(x) denotes the sign function whose value is '1' when $x \geq 0$ and is '0' when $x < 0$. The product of sign functions can be implemented with simple exclusive OR (XOR) operations, and the minimum function can be implemented with one loop of arithmetic comparisons.

Figure 2.9 shows the input and output parameters for a MS check node with six inputs.

Figure 2.10 shows the input and output parameters for a MS check node with concrete numbers. The parity check constraint involves 6 bits. The soft bit inputs are lv_0=+0.3, lv_1=+0.7, lv_2=+1.0, lv_3=+0.6, lv_4=-0.1, lv_5=+1.5. Among them, the minimum magnitude +0.1 is from $lv_4 = -0.1$. As a result, the magnitude of the outputs $lc_0, lc_1, lc_2, lc_3, lc_5$ is 0.1. For output lc_4, the magnitude is the minimum magnitude of $lv_0, lv_1, lv_2, lv_3, lv_5$, which is +0.3 from $lv_0 = 0.3$. Sign of each output bit is found by multiplying the sign bit of the other five input soft bits. For example, the sign of lc_0 is from the multiplication of the sign bits of $lv_1, lv_2, lv_3, lv_4, lv_5$, which is $1 \times 1 \times 1 \times (-1) \times 1 = -1$. Finally, $lc_0 = -1 \times 0.1 = -0.1$.

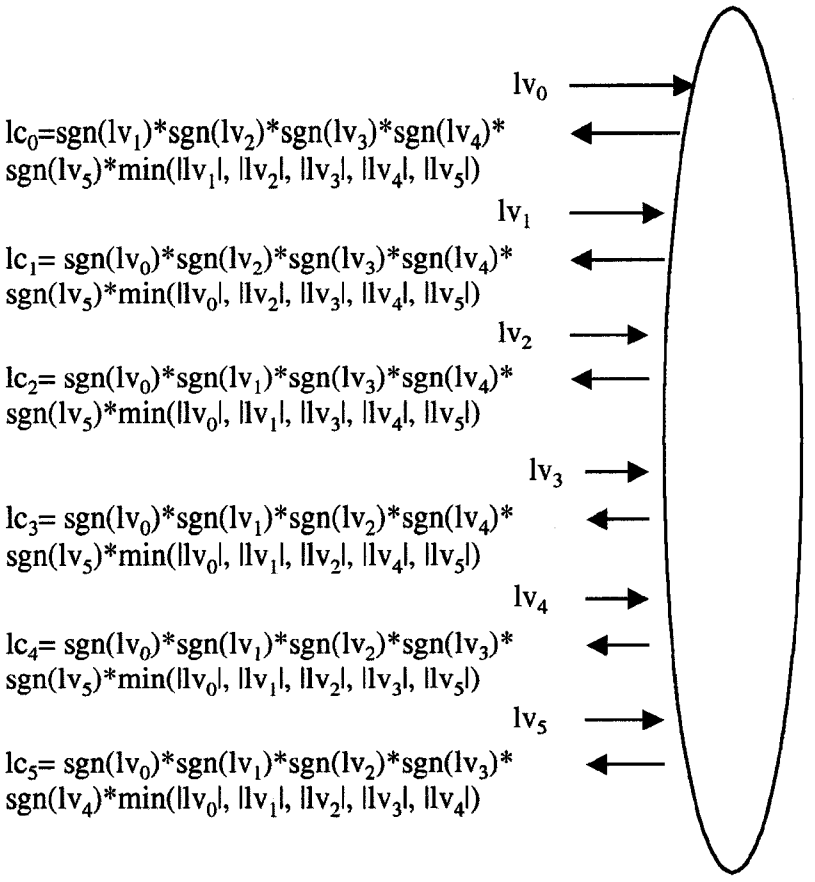

Figure 2.9: Min-Sum Algorithm Check Node Operation

2.3 Multi-threaded Microprocessors

In Flynn's taxonomy [30], computer architectures can be classified into four categories based on the number of concurrent instructions and data streams.

Single Instruction stream, Single Data stream (SISD) machines use a single processor to process a single data stream. Such a processor is also called a uniprocessor.

Single Instruction stream, Multiple Data stream (SIMD) machines use techniques to achieve data level parallelism. A SIMD processor consists of an array of

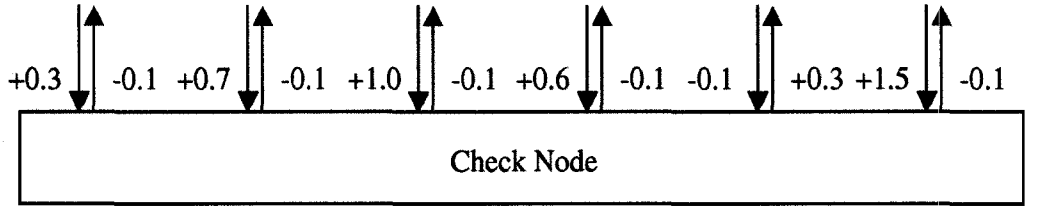

Figure 2.10: Min-Sum Algorithm Check Node Operation Example

processor elements (PEs). Multiple PEs execute the same instruction on their own data stream.

Multiple Instruction stream, Single Data stream (MISD) machines use many functional units performing on the same data. Not many machines of this type exist since MIMD and SIMD are often more appropriate for common data parallel techniques.

Multiple Instruction stream, Multiple Data stream (MIMD) machines use a number of processor elements. These PEs execute different instructions on different piece of data. Each PE has its own Arithmetic and Logic Unit (ALU) and control unit. PEs could be interconnected in some manner to allow for the exchange of data.

Figure 2.11 shows Flynn's computer architecture taxonomy.

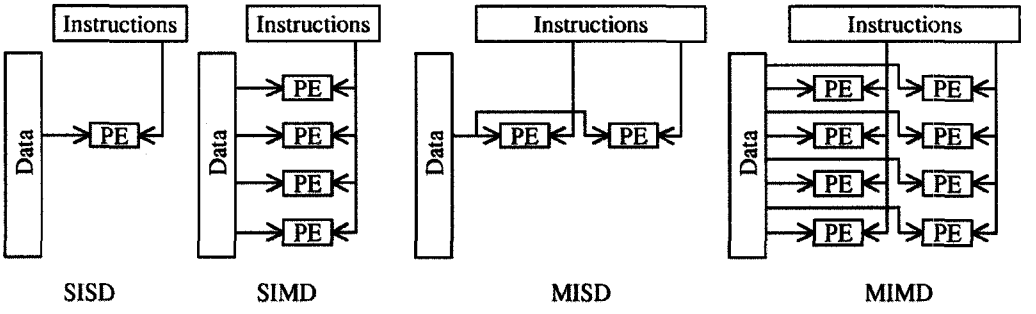

Figure 2.11: Flynn's Computer Architecture Taxonomy

In practice, many modern computers have hybrid architectures that combine, for example, aspects of SISD architecture with SIMD architecture, such as Intel's

Pentium microprocessor with MMX and Streaming SIMD Extensions (SSEs) [31].

One MIMD architecture that is intended to support flexible data exchange is called a *shared memory architecture*. Any processor can access memory modules through the interconnection network which connects microprocessors and memory modules. One problem with this architecture is the memory contention which occurs when two or more processors try to access the same memory block. Since one memory block usually has only one address bus and can only be accessed by one processor at a time, other processors have to wait until the first processor finishes accessing it.

Another architecture that facilitates data exchange is called a message passing architecture. Each processor has its own memory block attached to it. When data exchange is required, the requesting processor sends a message. In reply, the answering processor accesses its memory block and passes data on to the requesting processor through interconnection network. Memory contention problems are avoided since the memory block is only accessed by one processor.

In this research project, a XInC multi-threaded microprocessor is used. It was designed by Eleven Engineering Inc. in Edmonton, Canada. The XInC microprocessor was intended to be used in wireless or audio applications. It has its own unique structure which allows 8 instruction streams and 8 data streams. Inside a XInC there are 8 independent sets of program counters, general-purpose registers and conditional code registers. To simplify the microprocessor, only 18 instructions are implemented. Table 2.3 summarizes the XInC instruction set. A hardware semaphore mechanism can be used to manage access to the shared resources such as memory and I/O ports. Thus either Single Instruction Multiple Data (SIMD) mode or Multiple Instruction Multiple Data (MIMD) mode can be used.

In general, there are two main types of hardware multithreading implementation. A tutorial article about multithreading can be found in [32]. One type of

Table 2.3: XInC Instruction Set Summary

Mnemonic	Operands	Description
add	R1,R2,R3	2's complement add, $R1 = R2 + R3$
	R1,R2,K3	$R1 = R2 + K3$
and	R1,R2,R3	Bitwise and, $R1 = R2 \& R3$
	R1,R2,K3	$R1 = R2 \& K3$
bc	C1,K2	Conditional branch, if C1, PC=K2
bic	R1,R2,K3	Bit clear, $R1 = R2 \& (1 << K3)$
bis	R1,R2,K3	Bit set, $R1 = R2 \mid (1 << K3)$
bix	R1,R2,K3	Bit XOR, $R1 = R2 \wedge (1 << K3)$
bra	K1	Unconditional branch, $PC = PC + K1$
inp	R1,K2	Read input port, $R1 = input(K2)$
ior	R1,R2,R3	Bitwise inclusive or, $R1 = R2 \mid R3$
	R1,R2,K3	$R1 = R2 \mid K3$
jsr	R1,R2	Jump to/Return from subroutine, $R1 = PC; PC = R2$
	R1,K2	$R1 = PC; PC = K2$
ld	R1,R2,K3	Load from RAM, $R1 = *(R2 + K3)$
	R1,K2	$R1 = *K2$
mov	R1,K2	Move immediate, $R1 = K2$
outp	R1,K2	Write output port, $output(K2) = R1$
rol	R1,R2,R3	Bitwise rotate left, $R1 = R2 << R3$
	R1,R2,K3	$R1 = R2 << K3$
st	R1,R2,K3	Store to RAM, $*(R2 + K3) = R1$
	R1,K2	$*K2 = R1$
sub	R1,R2,R3	2's complement subtract, $R1 = R2 - R3$
thrd	R1	Get thread number, $R1 = thread\#$
xor	R1,R2,R3	Bitwise exclusive or, $R1 = R2 \wedge R3$
	R1,R2,K3	$R1 = R2 \wedge K3$

hardware multithreading is called *interleaved multithreading*, and XInC should be classified in this category. Other examples of interleaved multithreading processors include the Tera processor [33] and the HEP multiprocessor [34]. In interleaved multithreading, only one thread of instructions is executed in any given pipeline stage at a time. The purpose of this type of multithreading is to remove data dependency stalls due to one thread from the execution pipeline. Because one thread

is relatively independent from the other threads, there's less chance of one instruction in one pipeline stage needing an output from an older instruction of the other threads in the pipeline. An example of interleaved multithreading is shown below,

1. Cycle *i* : an instruction from thread A is issued
2. Cycle *i+1*: an instruction from thread B is issued
3. Cycle *i+2*: an instruction from thread C is issued
4. Cycle *i+3*: an instruction from thread A is issued

Another type of hardware multithreading is called *Simultaneous Multi-Threading* (SMT), which allows the instructions from more than one thread to be executed in any given pipeline stage at a time. It is a technique for improving the overall efficiency of superscalar CPUs. A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. Each functional unit is not a separate PE, but an execution resource within a single PE, such as an arithmetic logic unit, bit shifter or multiplier. A normal superscalar processor issues multiple instructions from a single thread in every clock cycle. In Simultaneous Multi-Threading, the superscalar processor can issue instructions from multiple threads in every clock cycle. An example of SMT is shown below.

1. Cycle *i*: instructions *j* and *j+1* from thread A; instruction *k* from thread B all simultaneously issued
2. Cycle *i+1*: instruction *j+2* from thread A; instruction *k+1* from thread B; instruction *m* from thread C all simultaneously issued
3. Cycle *i+2*: instruction *j+3* from thread A; instructions *m+1* and *m+2* from thread C all simultaneously issued

Unlike the general MIMD microprocessors which have multiple PEs, the XInC has only one time-shared PE. This processor element is divided into 8 pipeline stages. In a general-purpose microprocessor, the instruction pipeline breaks the in-

struction logic into several smaller blocks and inserts registers between the block. In this way the propagation delay between registers can be reduced and the system clock frequency can be increased. In addition, multiple instructions can share the pipeline and instruction throughput can be increased. For general-purpose microprocessors, successive instruction sequences from one thread are loaded into the pipeline, while in the XInC microprocessor, instructions are fetched from 8 instruction streams. That is, if for i_{th} clock cycle, the instruction in j_{th} instruction stream is fetched, then for the $(i+1)_{th}$ clock cycle, the instruction in the $(j+1)_{th}$ instruction stream would be fetched. For each system clock cycle, the instruction completes one of the eight pipeline stages. To complete the whole instruction, the XInC requires 8 system clock cycles. For each system clock cycle, one instruction is completed. The system clock frequency for XInC microprocessor is 12 MHz, and hence the maximum instruction throughput is 12 Million Instructions Per Second (MIPS).

Figure 2.12 shows the architecture of the XInC microprocessor and its memory model. The XInC microprocessor uses a shared memory architecture. There are a total of 16K words of RAM to store the program and data. The RAM is organized as 8 blocks of 2K words each. Different memory blocks can be accessed within one clock cycle at the same time, but the same memory block cannot be accessed more than once in the same cycle. The memory crossbar is responsible for routing the data to the addressed memory blocks. The processor has two ports that are connected to the memory crossbar. The instruction port is used to fetch instructions. The data port is used to access data. For the same system clock cycle, the instruction fetch component in the pipeline fetches an instruction through the instruction port for one thread while the memory access component in the pipeline fetches data through the data port for another thread. As a result, if the instruction and data stream are in the same memory block, it is possible to access the same memory

block in one clock cycle. If that happens, the instruction is still be fetched, but the data access is invalid. This memory contention could be eliminated easily by assigning the program space and data space into different memory blocks.

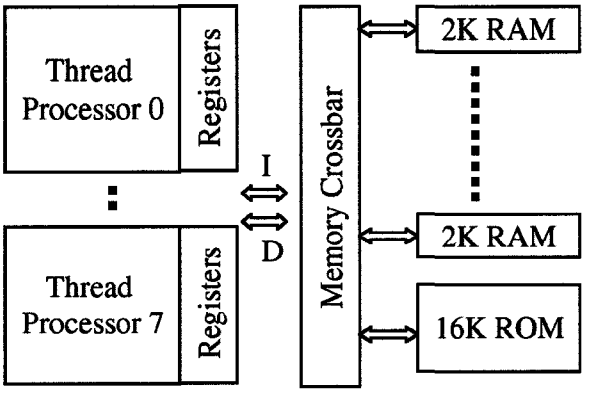

Figure 2.12: XInC Microprocessor Architecture

As mentioned earlier, for MIMD microprocessors, the shared memory architecture would cause memory contention if two or more PEs access the same memory block. The other processors have to wait until the first processor finishes its data access. However, in the XInC microprocessor, this problem does not exist since it has only one PE. The pipeline's memory access component processes one instruction in one clock cycle. Except for the above easily resolved instruction fetch and data access contention scenario, there is no memory contention for two data accesses.

In addition, the XInC microprocessor's pipeline architecture does not have the conditional branch problem that exists in a general-purpose microprocessor. Since in general-purpose microprocessors, the pipeline cannot know which branch should be taken for several clock cycles when it meets the conditional branch instruction, it cannot read in the following instructions in the same thread. This problem would degrade the pipeline's efficiency. To reduce this impact, branch prediction techniques could be used in a general-purpose microprocessor. However, in the XInC microprocessor, the pipeline gets instructions thread-by-thread. In the Tera proces-

sor [33], the processor logic selects a software-thread that is ready to execute and issues its next instruction. By contrast in the XInC, instruction streams are run in a fixed sequence from thread 0 to thread 7. The instruction in one thread requires 8 clock cycles to complete. The immediately following instruction in the same thread would not be fetched until the previous instruction in the same thread is complete. As a result, the conditional branch problem does not exist in the XInC microprocessor.

However, the XInC microprocessor still has potential data dependency problems between different threads due to pipelining. For example, when a memory write instruction is executed in one thread, the data would be written to memory after several clock cycles rather than written immediately. If a memory read instruction follows and it reads the same address in another thread, then the read instruction would read an old data since the new data has not been written yet. This hazard could be eliminated with additional hardware control, such as a hardware semaphore, but this approach requires overhead time to set and reset the hardware. Another way is to rely on software programmer or compiler to avoid this hazard. An example is Intel's Itanium, which relies on the compiler to decide which instructions can be executed in parallel and which must be executed serially [35][34].

Chapter 3

Improved Bit Flipping Algorithms

In this chapter, we begin by proposing the use of a Bit Flipping Threshold Pattern. The Improved Bit Flipping algorithm (IBF) is then described. Several alternative threshold patterns are simulated and the pattern $(3-2)^+ = (3-2-3-2...)$ is found experimentally to be the best pattern for the benchmark (128,3,6) LDPC-CC. In addition, the Parallel Improved Bit Flipping (PIBF) algorithm, which utilizes the microprocessors' built-in bit-wise parallelism, is also presented.

3.1 Bit Flipping Threshold Patterns

In this section, we present simulation evidence that demonstrates that for a well-studied (128,3,6) LDPC-CC, choosing a strictly alternating bit flipping threshold pattern $(3-2)^+$ can not only improve the coding gain, it can also speed up the decoding process. The regular expression operator $(.)^+$ denotes that the bit flipping threshold pattern within the bracket is repeated one or more times. We will be using regular expressions to describe bit flipping threshold patterns in the remainder of this thesis. Thus the pattern $(3-2)^+$ alternates the bit flipping threshold between a conservative '3' and an aggressive '2'. At a bit error rate of 10^{-4}, the $(3-2)^+$ pattern achieves a coding gain within 3.5 dB of that of the Min-Sum decoding algorithm, using only hard bits and six decoding processors. This decoding algorithm

requires much simpler hardware and lower power in a silicon implementation compared to decoding algorithms that process soft bits, such as the Min-Sum algorithm. The wiring and logic circuits required by hard bit processing is much simpler than that required for soft bits. What is more, the coding gain is 2.5 dB better than that of the Gallager's BF algorithm which uses a fixed pattern of $(3)^{+}$. The results will be detailed below.

3.1.1 LDPC-CC Improved Bit Flipping Decoding Algorithm

In a *(M,J,K)* LDPC-CC encoder (See Figure 2.4), both the previous M information bits and M check bits are stored in a First-In First-Out (FIFO) memory queue. The encoder selects *K-1* of the bits from the FIFO queue and generates one new check bit by simple exclusive OR (XOR) operation. The bit positions in the queue for the inputs of each parity check are determined by entries in a position table which is derived from the parity check matrix. Each encoded bit is involved in J parity check constraints, and each parity check constraints involves K earlier bits in the coded bit stream.

The encoded bits are modulated and transmitted as analog signals through a noisy channel. The shape of these signals is distorted by the addition of noise. There are typically many underlying components to this noise, and by the Central Limit Theorem, the distribution of the net noise amplitudes at the bit times will tend to be Gaussian. At the receiver end, a threshold device (comparator) is used to recover the binary digit '0' or '1'. Signal strength information, which may be related to the reliability of the recovered bit, is discarded. These binary bits are then sent into the LDPC-CC decoder for channel decoding.

LDPC-CC decoder is composed a unidirectional cascade of several identical decoding processors. In the bit flipping algorithm, each bit is checked by several parity check constraints in the check node. If a constraint fails, the error counter

of all of the associated input bits is incremented by 1. When the error counter is greater than the bit flipping threshold, the bit would be flipped in the belief that the bit's old value was probably be incorrect.

The intuition behind the Bit Flipping algorithms is that the greater the number of failed parity check constraints involving a particular bit, the higher should be the probability that the bit is in error. The best choice of bit flipping threshold b might be based on the parity check matrix characteristics and the estimated signal-to-noise ratio. If b is chosen to be too small (i.e., the threshold is too aggressive), then too many correct bits would be wrongly flipped and the algorithm might not converge on the correct data. On the other hand, the bit flipping threshold b should be set to a sufficiently small value so that suspect error bits that are indicated only by a fewer number of failed parity check constraints can get flipped. If b is chosen to be too high (i.e, the algorithm is too conservative), then correction of suspect bits is too difficult to trigger and convergence on the correct data might be too slow or may get stuck in local minima with some errors left uncorrected.

With respect to the bit flipping decoding algorithm, we define the *bit flipping threshold pattern* $(b_1 - b_2 - b_3 - ...)^+$ to be the sequence of bit flipping thresholds which are used in the 1st, 2nd, 3rd, etc. decoding processors. If the pattern is shorter than the desired number of decoding processors, then the sequence is repeated as often as necessary, starting again each time at b_1. The original Gallager bit flipping algorithm can be considered as having the bit flipping threshold pattern $(3)^+$.

Gallager's BF algorithm was developed to decode LDPC-BCs, but here we need to decode LDPC-CCs. Essentially LDPC-BC decoding algorithms perform iterative decoding in time whereas LDPC-CC decoding algorithms perform iterative decoding in space over a cascade of pipelined decoder processors [8]. Our improved bit flipping algorithms, modified for LDPC-CCs and based on bit flipping threshold patterns, share the following structure:

Step 1: Within the FIFO memory queue of information and check bits inside each decoder processor, find the number f_i of failed parity-check constraints for each bit i for the entire time that the bit spends shifting through the decoding processor.

Step 2: Once a bit reaches the head of a decoding processor, if the number of failed parity check constraints for a bit exceeds the present bit flipping threshold b, as specified by the bit flipping threshold pattern, then that bit is flipped before it enters into the next decoding processor.

Step 3: After a bit exits the last decoding processor, the decoding process for that bit is finished and the decoded bit is sent out.

For our benchmark (128,3,6) LDPC-CC, each information and check bit is involved in 3 parity check constraints. The bit flipping threshold b could be chosen to be '3' or '2'. In this situation, $b = 3$ indicates that all the parity check constraints have failed and $b = 2$ indicates that a majority of the parity check constraints have failed. Threshold value '1' is not used since when a single parity check constraint has failed, all the error counters of the bits that are associated with that constraint would be incremented by one. The threshold '1' would be undesirable since an incorrect decision to flip one bit would propagate incorrect decisions to flip all of the bits involved in parity check constraints.

For the first several decoding processors, b could be set to the conservative threshold value '3', that is, to flip the bit only when all three of its parity check constraints fail. When most of the bits indicated by three failed parity check constraints are corrected, b could be set to an aggressive threshold value '2' to flip the bits indicated by two or more failed parity check constraints. The overall bit error rate might rise at this time since many correct bits might be wrongly flipped due to the aggressive '2'. But this temporary bit error rate increment is necessary since error bits only indicated by two failed parity check constraints could be corrected in

this step. Otherwise, an overly conservative strategy would prevent some incorrect bits from being flipped and thus the bit error rate would be stuck at a higher level. After *b=2*, *b* could be set to conservative '3' again for the following decoding processors to drop the bit error rate down to a lower level. The threshold *b* could in this way be alternated between '3' and '2' for the decoding processors. In our research, we assumed that the bit flipping threshold pattern is set for a fixed number of decoding processors to a threshold value of '3' followed by a threshold value of '2'. Thus we focussed our attention on the threshold patterns as $(3-2)^+$, $(3-3-2)^+$, $(3-3-3-2)+$. This restriction was supported by experimental evidence from many simulation trials. If the threshold pattern starts with threshold value '2', such as $(2-3)^+$, simulation results show that the coding gain is worse since too many corrected bits are wrongly flipped in the first decoding processor.

3.1.2 Simulation Results

3.1.2.1 Coding Gain

Figure 3.1 shows simulation results of bit error rate versus signal-to-noise ratio E_b/N_o for an uncoded BPSK signal, the BF algorithm with bit flipping threshold pattern $(3)^+$, the BF algorithm with the bit flipping threshold pattern $(3-2)^+$, the Min-Sum algorithm and the Sum-Product algorithm for the benchmark (128,3,6) LDPC-CC. To determine the maximum coding gain of each algorithm, sixty decoding processors were used. Each plotted point corresponds to at least 100 error events to ensure statistically reliable values. At a bit error rate of 10^{-4}, the $(3-2)^+$ pattern achieves a coding gain that is only 3.5 dB less than that of the Min-Sum soft-decoding algorithm. What is more, the coding gain is 2.5 dB better than that of Gallager's BF algorithm with a fixed pattern of $(3)^+$. Other bit flipping threshold patterns (e.g. $(3-3-2)^+$, $(3-3-3-2)^+$, $(3-3-3-3-2)^+$ and $(3-3-3-3-3-2)^+$ were also simulated using 60 decoding processors. Their

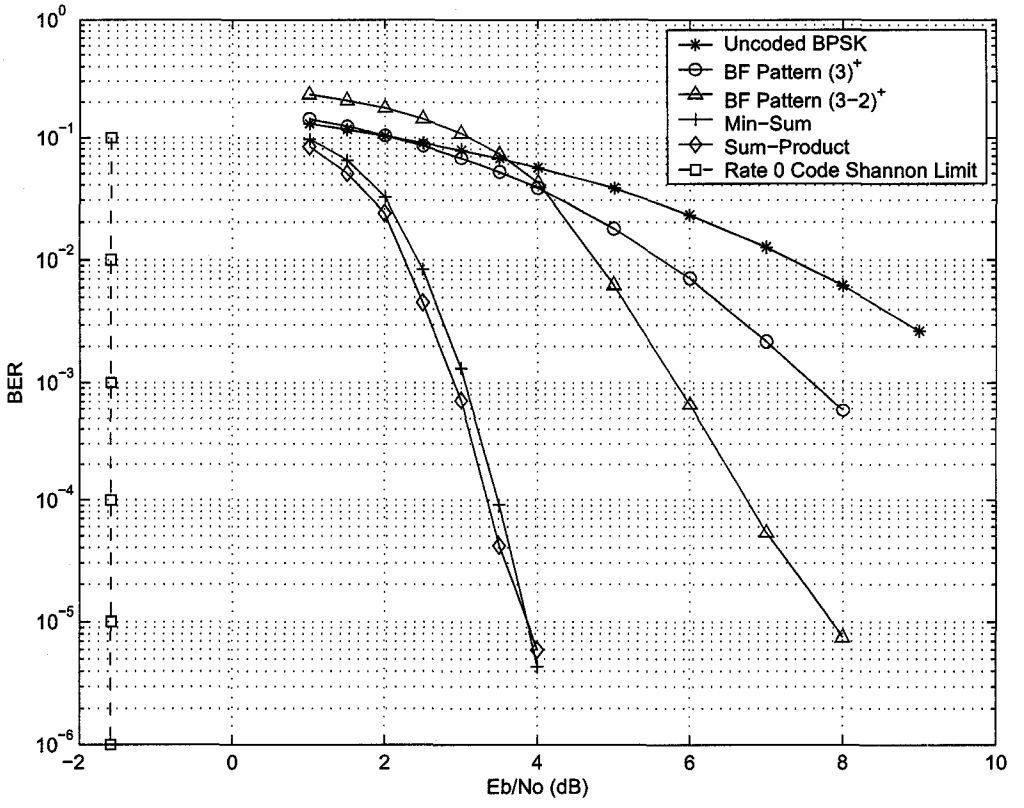

Figure 3.1: BER of Uncoded BPSK, BF Pattern $(3)^+$ and $(3-2)^+$, Min-Sum and Sum-Product algorithm for a (128,3,6) LDPC-CC

curves show essentially the same coding gain of the pattern $(3-2)^+$, so it appears that the coding gain is almost the same if the threshold pattern is alternated strictly between '3' and '2' with an adequate number of decoding processors. Observe in Figure 3.1 that the BF decoding algorithm has some coding gain only for signal-to-noise ratios E_b/N_o of above 4 dB. Below 4 dB, the bit error rate of the bit flipping algorithm is higher than the uncoded BPSK signal. The reason might be that when the signal-to-noise ratio is low, too many errored bits are present in the bit stream. In this situation, the parity check results and the associated bit flipping actions become unreliable and hence more error bits get be generated than get corrected.

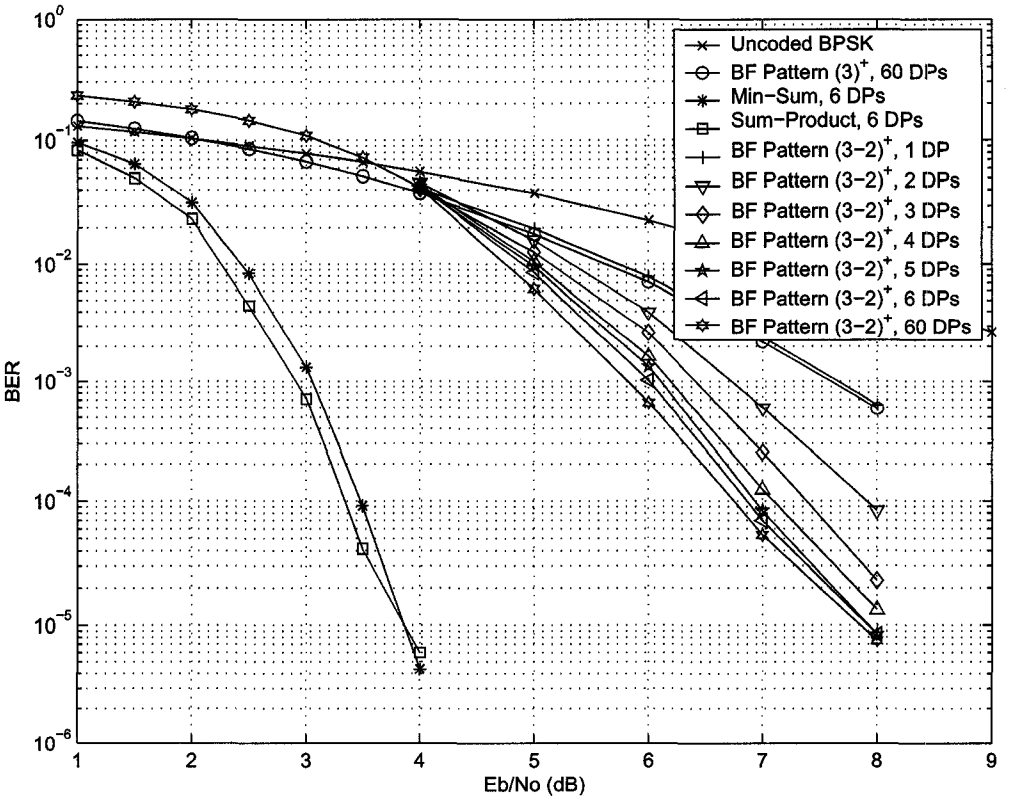

Figure 3.2: BER of BF Pattern $(3-2)^+$ after 1-6 and 60 Decoding Processors for a (128,3,6) LDPC-CC

3.1.2.2 Decoding Processor Number

Figure 3.2 shows the simulation results of bit error rate versus signal-to-noise ratio E_b/N_o for different numbers of decoding processors, given the same $(3-2)^+$ pattern. In the figure, the coding gain is increased notably for the first six decoding processors. But beyond six decoding processors, there are only small further improvements in the coding gain. For the (128,3,6) LDPC-CC, the curve for 6 decoding processors approaches within 0.2 dB of the curve for 60 decoding processors. It would thus appear that 6 decoding processors might be a good trade-off point between coding gain and required decoding processors for the (128,3,6) LDPC-CC.

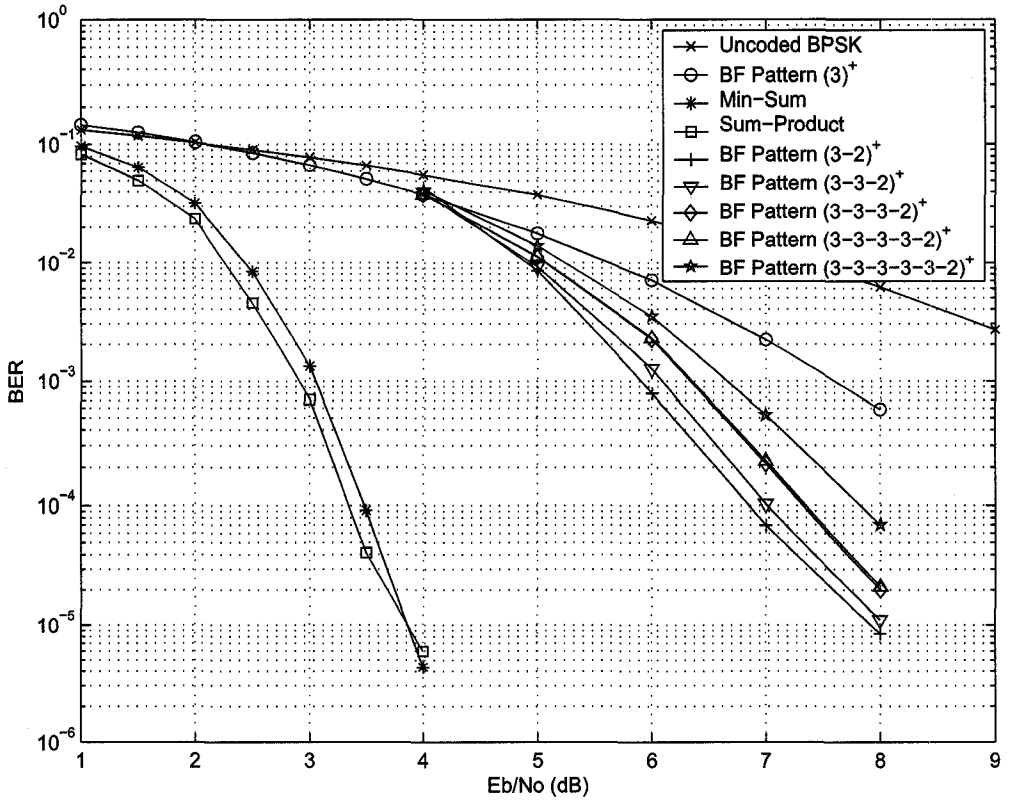

Figure 3.3: BER of BF Pattern $(3-2)^+$,$(3-3-2)^+$,$(3-3-3-2)^+$,$(3-3-3-3-2)^+$ and $(3-3-3-3-3-2)^+$ for (128,3,6) LDPC-CC with 6 Decoding Processors

3.1.2.3 Error Correction Convergence Speed

Figure 3.3 shows the simulation results for different bit flipping threshold patterns using six decoding processors. In the figure, it is evident that for all of the considered signal-to-noise ratios E_b/N_o, the $(3-2)^+$ bit flipping threshold pattern has the lowest BER and the $(3-3-3-3-3-2)^+$ pattern has the highest BER.

In Figure 3.4, the BER for different bit flipping threshold patterns using different numbers of decoding processors was simulated at the same signal-to-noise ratio $E_b/N_o = 6dB$. It appears that the BF threshold pattern $(3-2)^+$ has the fastest error correction convergence speed compared to its peers. In other words, the bit flipping

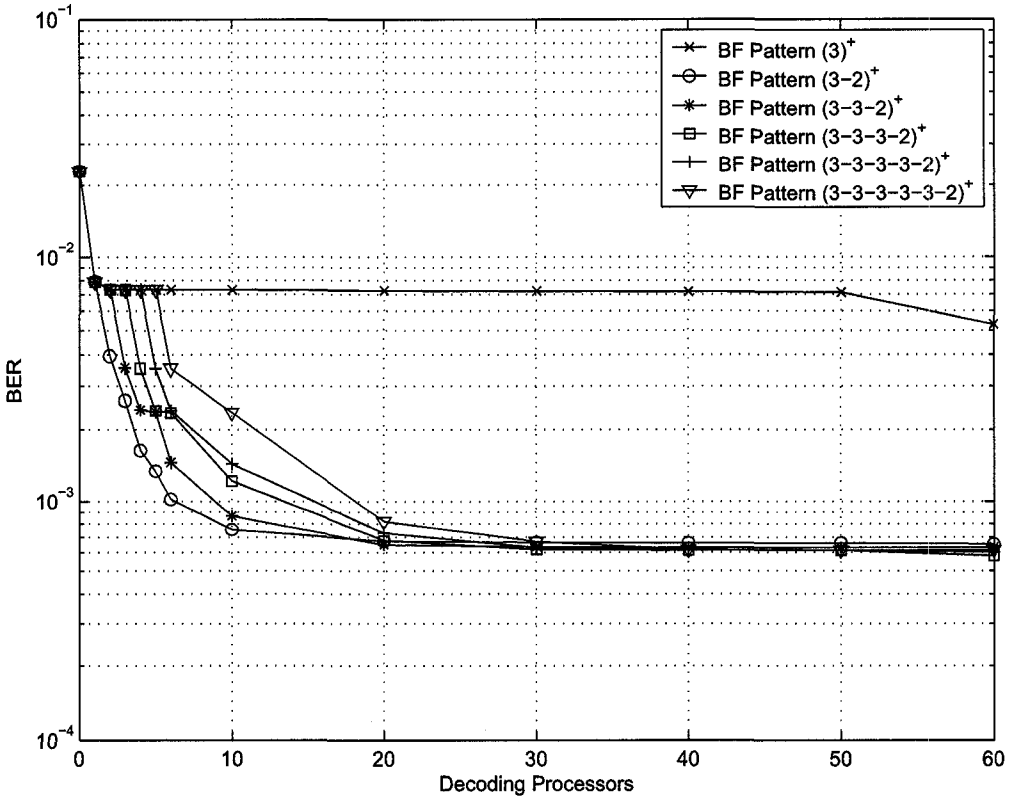

Figure 3.4: BER of BF Pattern $(3-2)^+$,$(3-2)^+$,$(3-3-2)^+$,$(3-3-3-2)^+$,$(3-3-3-3-2)^+$ and $(3-3-3-3-3-2)^+$ after Different Number of Decoding Processors, $E_b/N_o = 6dB$

threshold pattern $(3-2)^+$ requires fewer decoding processors to achieve the same bit error rate compared with the other patterns. Note that Gallager's BF algorithm, with threshold pattern $(3)^+$, gets stuck at a relatively high BER. It appears that by alternating the threshold between '3' and '2', the decoding algorithm is able to escape from local minima that prevent convergence in the presence of some errors.

From the above simulation results, it can be concluded empirically that for the benchmark (128,3,6) LDPC-CC, the bit flipping decoding algorithm using the bit flipping threshold pattern $(3-2)^+$ with 6 decoding processors achieves the best compromise between coding gain and required decoding processors.

3.2 The Parallel Improved Bit Flipping Algorithm

The Bit Flipping algorithm is based on simple bit manipulations such as the logical AND, OR and XOR operations in check nodes and variable nodes. In microprocessors, bit manipulation operations are typically designed to be 8-, 16-, 32- or 64-bits wide. As a result, we developed the Parallel Improved Bit Flipping (PIBF) algorithm to exploit this built-in parallelism in microprocessors.

The encoder structure of our proposed Parallel Improved Bit Flipping algorithm (PIBF) is almost the same as the non-parallel encoder. The only difference is that the bit stream is processed as 16-bit words since the XInC microprocessor is a 16-bit microprocessor, and XOR instructions are executed at the word level. We will call the 16-bit word a hard word. In PIBF, the bit stream could be viewed as 16 interleaved streams. Sixteen bits from each of these streams are packed into a single 16-bit word.

In Figure 3.5, the IBF and PIBF algorithm examples are shown. For the IBF algorithm, the parity check constraints are based on bits. For example, the constraint illustrated in the figure is

$$bit_1 \oplus bit_{18} \oplus bit_{23} \oplus bit_{132} \oplus bit_{212} \oplus bit_{257}$$

In PIBF, the same position table in IBF is used. But the parity check constraints are based on 16-bit hard words. As a result, the constraint is

$$word_1 \oplus word_{18} \oplus word_{23} \oplus word_{132} \oplus word_{212} \oplus word_{257}$$

If we consider the PIBF algorithm in terms of bit position offsets, the above word-based constraints could be written as 16 bit-based constraints as follows:

$$bit_1 \oplus bit_{289} \oplus bit_{369} \oplus bit_{2113} \oplus bit_{3393} \oplus bit_{4113}$$

$$bit_2 \oplus bit_{290} \oplus bit_{370} \oplus bit_{2114} \oplus bit_{3394} \oplus bit_{4114}$$

...

$$bit_{16} \oplus bit_{304} \oplus bit_{384} \oplus bit_{2128} \oplus bit_{3408} \oplus bit_{4128}$$

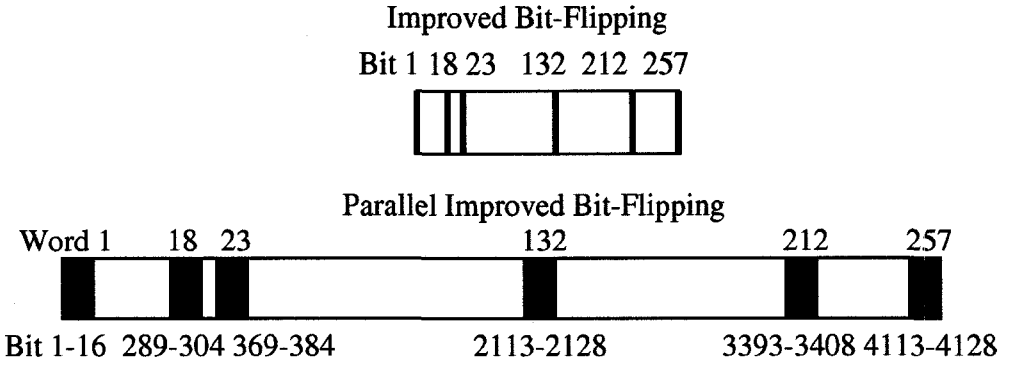

Figure 3.5: IBF and PIBF Algorithm Demonstration

In Figure 3.6, five words are chosen from the FIFO queues based on the position table in the encoder. We see that sixteen equivalent groups of bit parity check operations are performed at the same time. After four XOR instructions, sixteen check bits are generated. Here we call the block of sixteen check bits a check word.

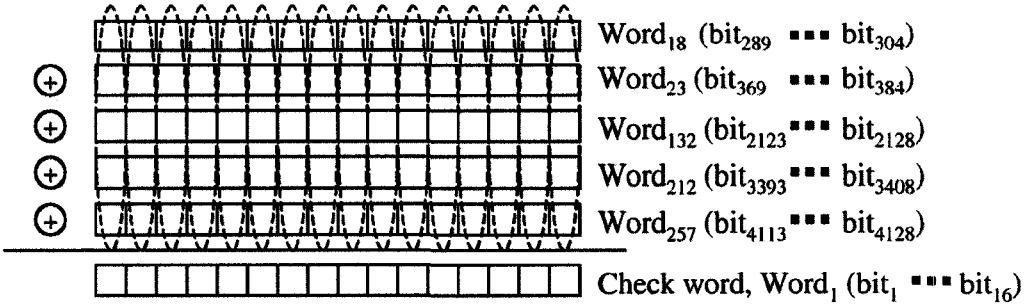

Figure 3.6: PIBF Encoding

In the PIBF decoder check node, parity checks are based on six words as shown in Figure 3.7. Five XOR instructions are executed. The check node checks whether

the parity check constraints on sixteen groups of bits are obeyed or failed. If one group of the parity checks fails, the corresponding bit of the result word would be '1'. Here we call the result word the check error pattern.

The PIBF decoder for the (128,3,6) LDPC-CC is now designed as follows. Four memory FIFO queues are used. The first queue stores the received hard words. The other 3 queues store the check error patterns from check node results since each word involves 3 parity check constraints. When a new word is shifted into the queue head, the variable node retrieves three check error patterns and calculates how many errors are associated with each bit. Assuming the bit flipping threshold pattern $(3-2)^+$ is used, we should determine whether the failed parity check constraint number for each bit is equal to '3', or greater or equal to'2'.

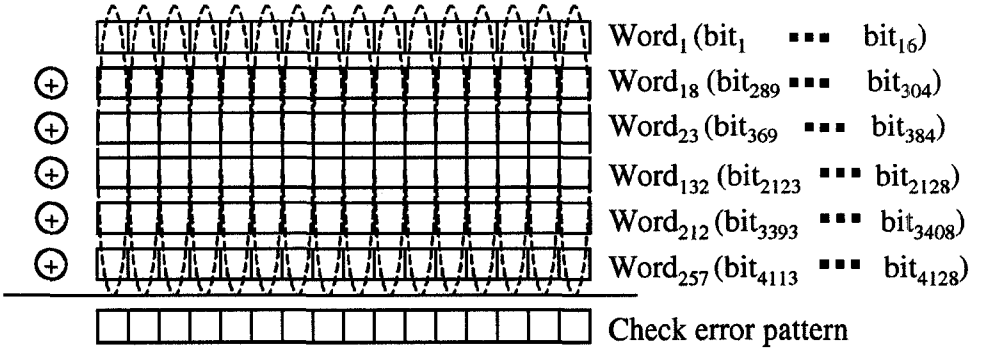

Figure 3.7: PIBF Decoding

One method for calculating the number of failed parity check constraints of each bit is to shift and add. Each check error pattern is shifted right by 1 bit and then the shifted-out bits are added. However, for a 16-bit word, this method would require 16 shifts and additions and this would make the variable node slow.

In our PIBF algorithm, another method is used. Two new patterns are generated by word-level instructions on the three check error patterns. One pattern determines whether all three parity check constraints have failed. We call this word result Flipping Pattern 1 (FP1). It could be calculated by ANDing three check error patterns

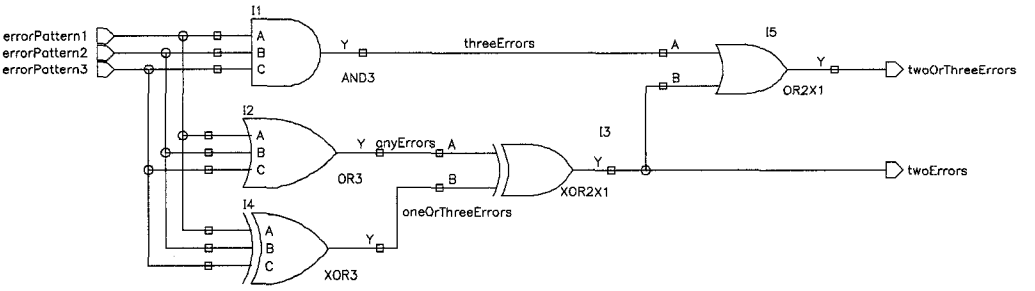

Figure 3.8: PIBF Algorithm Error Pattern Processing

as follows:

$Y_1 = E_1\ AND\ E_2\ AND\ E_3$ (3 errors)

The second word result indicates whether the number of failed parity check constraint is greater or equal to 2 which we call here Flipping Pattern 2 (FP2). The following expressions are evaluated:

$Y_1 = E_1\ AND\ E_2\ AND\ E_3$ (3 errors)

$Y_2 = E_1\ OR\ E_2\ OR\ E_3$ (at least 1 error)

$Y_3 = E_1\ XOR\ E_2\ XOR\ E_3$ (1 error or 3 errors)

$Y_4 = Y_2\ XOR\ Y_3$ (2 errors)

$Y_5 = Y_1\ OR\ Y_4$ (2 errors or 3 errors)

If the decoding processor's bit flipping threshold is '3', XORing the received hard word with the FP1 will flip all the bits in the word whose failed parity check constraint number is '3'. If the decoding processor's bit flipping threshold is '2', XORing the received hard word with the FP2 will flip all the bits whose failed parity check constraint number is '2' or '3'.

The above logical operations can also be implemented equally well in FPGAs or ASICs. The corresponding logic gate schematic is shown in Figure 3.8. The inputs are three check error patterns. The outputs are FP1 and FP2. Note that all of the inputs and outputs are 16-bit words, and all the logical gates are actually stacks of gates that are 16 levels high.

3.3 Conclusion

In this chapter, the Improved Bit Flipping algorithm and Parallel Improved Bit Flipping algorithm were proposed based on the use of the Bit Flipping Threshold Pattern. At a BER at 10^{-4}, the Bit Flipping Threshold Pattern $(3-2)^{+}$ achieves 2 dB better coding gain than the original Bit Flipping algorithm with the fixed Bit Flipping Threshold Pattern $(3)^{+}$, which was used in Gallager's bit-flipping algorithm. In addition, the Parallel Improved Bit Flipping algorithm exploits a microprocessor's built-in bit-wise logical instructions which could execute the decoding algorithm 16, 32, or 64 bits a time on a 16-bit, 32-bit or 64-bit microprocessor, respectively.

Chapter 4

LDPC-CCs on Multi-threaded Microprocessors

In this chapter, we describe how the Min-Sum algorithm, Improved Bit Flipping algorithm and Parallel Improved Bit Flipping algorithm were implemented on the XInC multi-threaded microprocessor. The structure and trade-offs between computational complexity and coding gain of these decoding algorithms are discussed in detail.

4.1 Memory Organization and Flow Chart

In LDPC-CC decoding algorithms, memories are used to store the received soft bits and hard bits, check error patterns, and intermediate soft bit results. The XInC multi-threaded microprocessor has a single global memory shared by all threads. LDPC-CC decoding processors read data from the memory, process them, and store the results back. These decoding processors are distributed onto several XInC threads to achieve parallel decoding. As a parallel algorithm on a multi-threaded microprocessor, it is important to prevent memory access hazards among the threads. In other words, when a memory location is being accessed by one thread, this location should not be accessed by other threads. Otherwise, when the

first thread writes to the memory and the second thread reads from the same memory location, old data might be read by the second thread. Also when two different data word are written to the same memory location by two different threads, the first written data would be overlapped.

A hardware semaphore mechanism might be used to manage shared hardware resources, such as memory. A semaphore is a binary variable which has two states: locked and unlocked. Such a variable can be assigned to a shared resource to regulate access to that resource. When the resource is in use, the semaphore is in the "locked" state. Otherwise, it is in the "unlocked" state. Before the shared resource is accessed, the semaphore state should be queried. If the semaphore is in the "locked" state, the thread has to wait until the semaphore goes back to the "unlocked" state. If the semaphore is in the "unlocked" state, the shared resource could be accessed. When the shared resource is accessed, the thread which uses the resource must write the semaphore to the locked state to indicate to the other threads that the resource is in use. When the operation is finished, the semaphore must be unlocked to release the resource.

However, for intensive memory access algorithms, such as the LDPC-CC decoder, the semaphore mechanism would likely be too inefficient. For example, in the parallel improved bit flipping algorithm, 30% of the instructions are used for memory accesses. If each memory access instruction requires two additional instructions to lock and unlock an associated semaphore, then the algorithm would require an additional 60% instructions.

In our LDPC-CC decoding algorithms, the memory access hazard is resolved by organizing the memory into different memory spaces. Each thread has exclusive access to its own memory space. In this, different threads are not allowed to access the same memory location at the same time. Hence, the semaphore mechanism is avoided.

In Figure 4.1, the memory organization for the LDPC-CC decoder is shown. As mentioned in Chapter 2.2.2.1, FIFO queues are used in the LDPC-CC decoding processors. Those queues are organized as circular buffers in the algorithm. Instead of moving the data in the FIFO queue, a circular pointer is moved to indicate the memory address of the queue tail. For each decoding phase, the circular pointer is moved counter-clockwise one step (that is, one information bit and one check bit memory space). Each decoding phase for each decoding processor includes one check node operation followed by one variable node operation. As shown in the figure, the decoding processors on threads 2 to 7 access their own memory spaces. Additional memory is allocated at the queue tail for data input operations and at the queue head for data output and hard decision operations. As a result, these operations on thread 1 could be run without memory access hazards with respect to the decoding processors on threads 2 to 7.

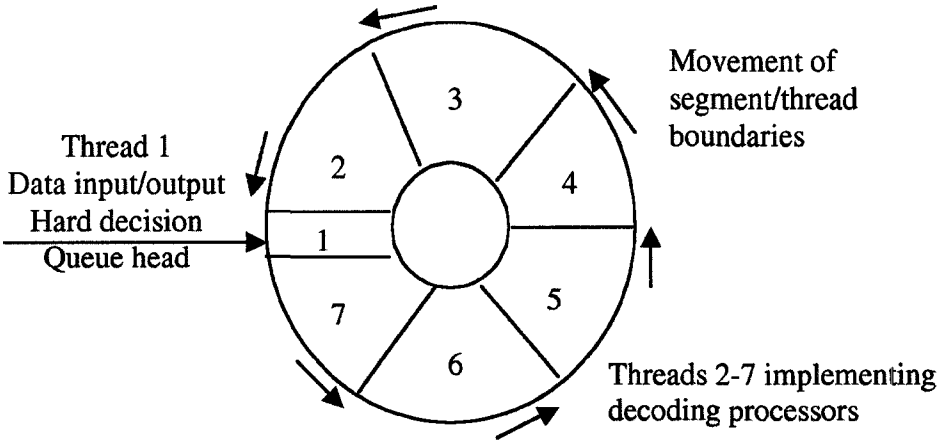

Figure 4.1: LDPC-CC Parallel Algorithm Memory Organization

Figure 4.2 shows the LDPC-CC encoder and decoder algorithm flow chart for the XInC microprocessor. The LDPC-CC encoding algorithm is implemented on thread 0. LDPC-CC decoding processors are implemented on thread 1 to thread 7. The LDPC-CC decoding algorithm can be divided into several operations: the data

input and output operations, the hard decision operation and the decoding processor (including a variable node and a check node). In thread 1, a decoder controller is implemented to synchronize the decoding phases on six decoding processors and to control data input and output. At the beginning of the decoding phase, the decoding processors are started by the decoding controller at the same time. At the end of the decoding phase, the decoder controller queries the status of the decoding processors (When the decoding processor is completed, it is left in a endless loop state). Only after all of the decoding processors have completed does the decoder controller start the next decoding phase. In addition to the decoder controller, hard decision operations on the output produced by thread 7 and data input and output operations are also implemented on thread 2. These activities execute in parallel with the decoding processors on threads 2 to 7.

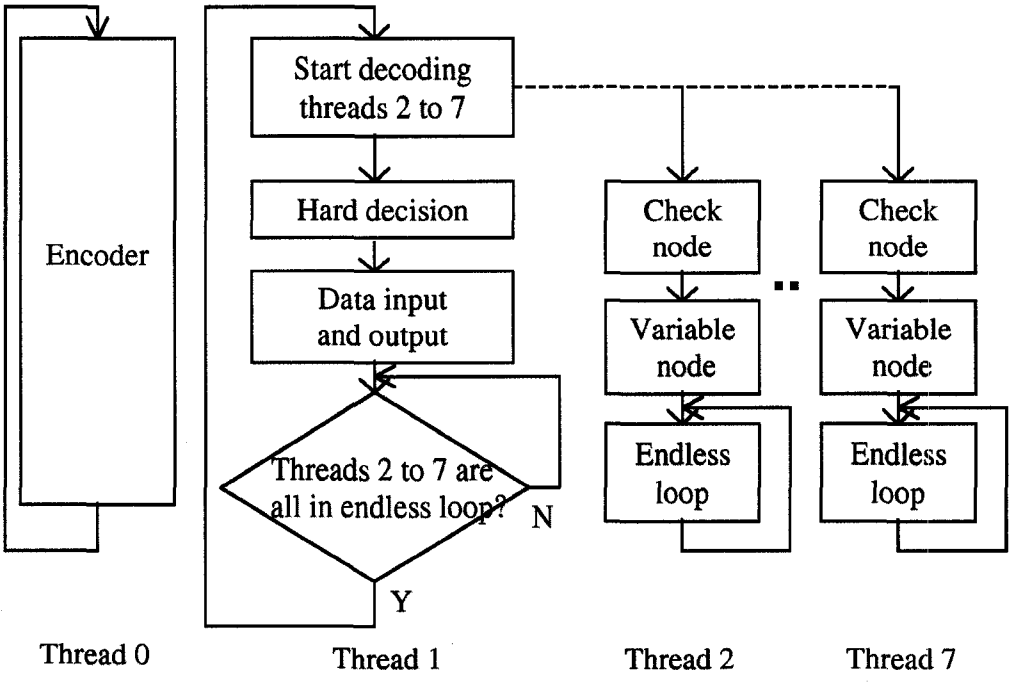

Figure 4.2: LDPC-CC Parallel Algorithm Flow Chart

4.2 LDPC-CC Decoder Implementation

4.2.1 The Min-Sum Algorithm

The Min-Sum algorithm is a soft-decoding algorithm. In our implementation, a soft bit is represented by an integer. Ideally, a Log-Likelihood Ratio (LLR) in Equation 2.1 is used to represent the reliability information in a sample of the received signal. For a (128,3,6) LDPC-CC, four FIFO queues are used. When the received soft bit comes in, it is written to the queue tail of all four queues of the first decoding processor.

Figure 4.3 shows the Min-Sum algorithm check node operation. The check node reads soft bits from memory queues 1 to 3. Each coded bit is involved in three parity check constraints. If the check node is verifying the i-th parity check for an input bit k, the check node reads the soft bit from queue i under the location for bit k, where $1 \leq i \leq 3$. When the check node finishes its operation, the result is written back to the same queue location where the soft bit was read from.

The check node operation was specified mathematically in Equation 2.2. For the sign function *sgn(.)* in the equation, the operand's most significant bit is retrieved. The exclusive OR(XOR) operation among those bits gives the sign function result. The absolute value function $|.|$ requires a comparison with the value zero to determine whether the operand is positive or negative. If it is negative, the operand is subtracted from 0 to get its absolute value. In the check node, for each input number, the output magnitude should be the minimum value of all other input numbers. In our minimum function *min(.)* implementation, a single loop is used to find two minimum numbers for a set of soft bit inputs. Soft bit outputs are produced from the soft bit inputs. For the minimum input, the corresponding output is the value of the second minimum (i.e., the minimum of the other inputs). For the remaining inputs, the corresponding outputs are assigned the value of the minimum input.

Figure 4.4 shows the variable node operation. When the soft bit is at the queue

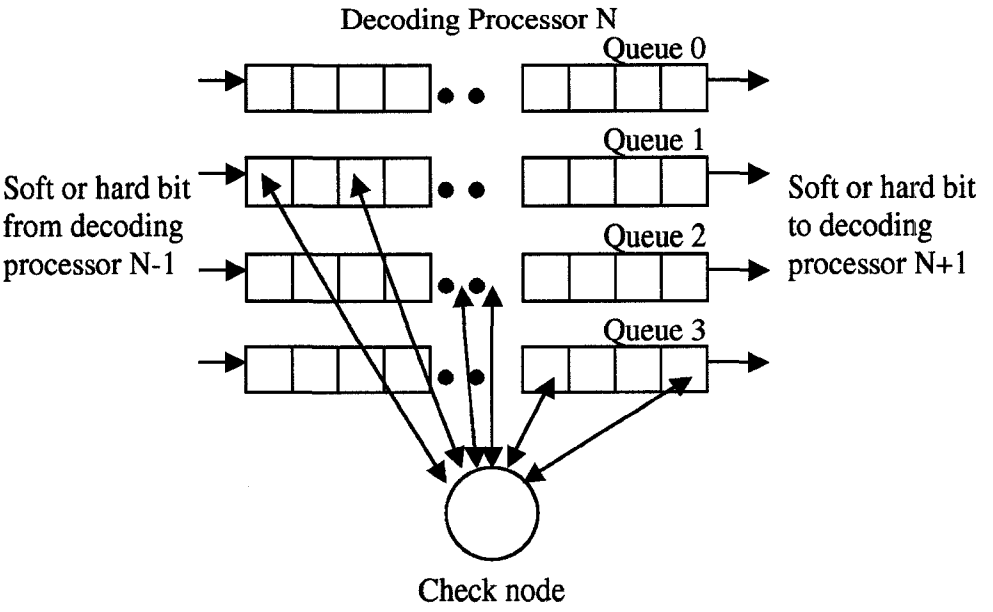

Figure 4.3: Min-Sum Algorithm and Parallel Bit Flipping Algorithm Check Node Operation

head of a decoding processor, all its associated parity check operations involving that bit have been finished. The corresponding parity check results have been stored in queues 1 to 3. Then the variable node operation is performed. For each exit value in queues 1 to 3, the exit value from queues 0 to 4, except the value in that queue itself, would be added and stored back. The received soft bit in queue 0 is not changed. The results at the queue heads are forwarded to the next decoding processor, or to the hard decision operation if the present decoding processor is the last one in the decoder cascade.

During the whole decoding process, one copy of the original received soft bits is kept in queue 0. This soft bit information is also pushed in queues 1 to 3 at the first decoding processor in the decoder cascade. However, in the following decoding processors, the input data that is written to the tails of queues 1 to 3 is shifted in from the heads of the corresponding queues of the previous decoding processor.

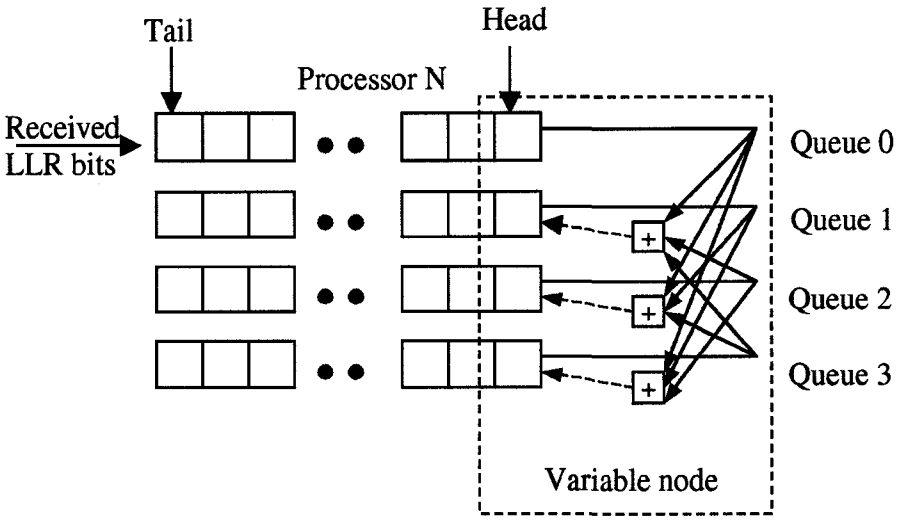

Figure 4.4: Min-Sum Algorithm Variable Node Operation

The probability of bit errors should gradually be reduced as soft bits shift in the rightward direction through the cascade of decoding processors.

After the data exits the rightmost decoder processor, a hard decision operation determines the binary output. In Figure 4.5, the hard decision operation is shown. First, all the values from queue 0 to 3 are added together. If the sum is greater or equal to 0, the hard output bit is decoded as '0'. Otherwise, the bit is decoded as '1'.

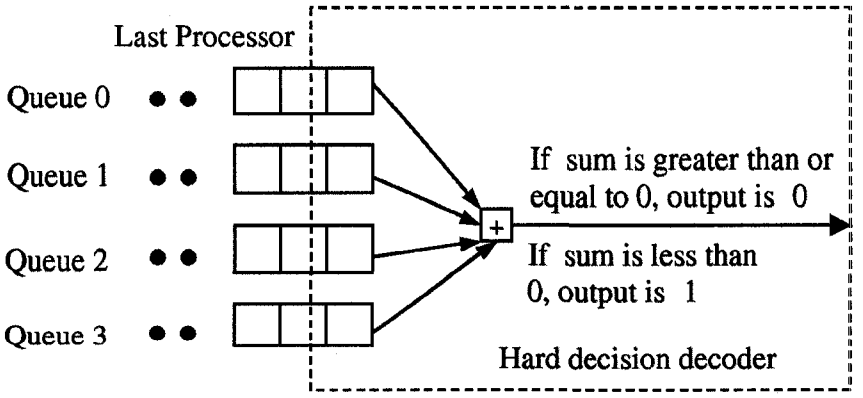

Figure 4.5: Min-Sum Algorithm Hard Decision Operation

4.2.2 The Improved Bit Flipping Algorithm

The Improved Bit Flipping algorithm is a hard-decoding algorithm. The received signal is sampled and the threshold device is used to generate the hard bit '0' or '1'. The algorithm then processes this hard bit data. In our IBF algorithm, for implementation simplicity, each hard bit is actually represented by a multi-bit integer, but only the least significant bit is used.

Figure 4.6 shows the IBF algorithm check node operation. Two FIFO queues are required instead of the four queues in the Min-Sum algorithm. Queue 0 is used to store the received hard bits or flipped bits. Queue 1 is used to store the check error counter values. The check node reads hard bits from queue 0. The exclusive XOR operation among those bits gives the parity check result. If the parity check fails, all of the check error counters of the associated bits are incremented by 1.

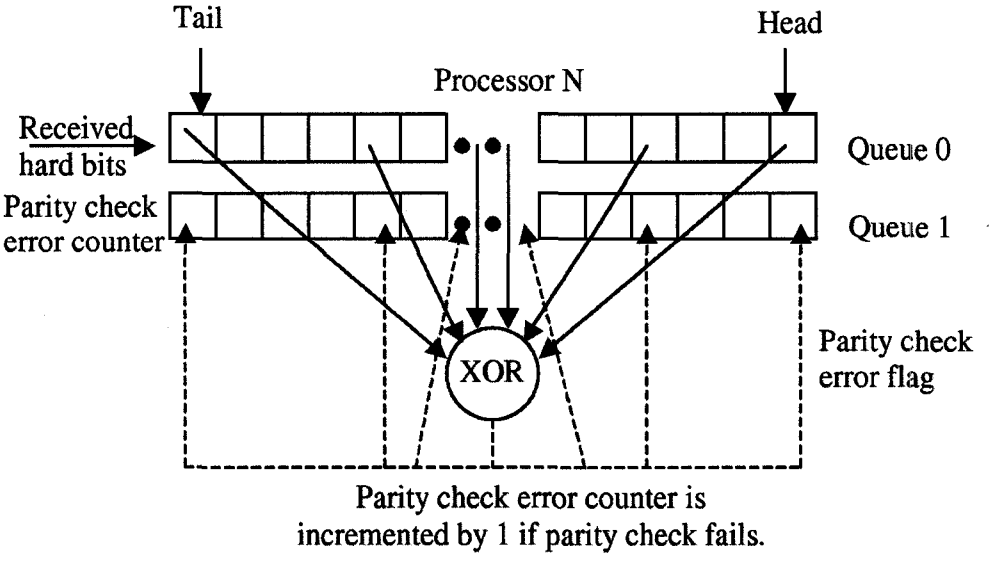

Figure 4.6: Improved Bit Flipping Algorithm Check Node Operation

Figure 4.7 shows the IBF algorithm variable node operation. At the exit of each decoding processor, the variable node checks whether the check error counter exceeds the bit flipping threshold of that decoding processor. The bit in queue 0

is flipped if the check error counter equals to or exceeds the threshold. The check error counter is then reset to 0 just before it is shifted over to the next decoding processor.

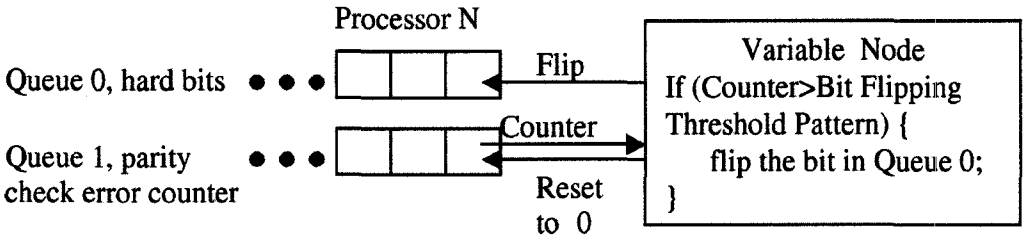

Figure 4.7: Improved Bit Flipping Algorithm Variable Node Operation

A hard decision operation is not required in the IBF algorithm since the bits are already represented in the hard bit format. The decoded bit at the head of queue 0 would be output directly from the last processor in the cascade without any further processing.

4.2.3 The Parallel Improved Bit Flipping Algorithm

In the Parallel Improved Bit Flipping algorithm, hard bits are used. For the XInC multi-threaded microprocessor, the datapath width is 16. Hence 16 hard bits can be packed into one 16-bit word for parallel decoding. Bit-wise logical instructions can then be used to implement the decoding operation.

Figure 4.3 shows the PIBF algorithm check node operation. Four FIFO queues are used. Queue 0 is used to store received hard words and intermediate flipped hard word results. The received hard words are stored to the tails of queues 0 to 3 in the first decoding processor. The input of the following decoding processor is shifted out from the head of the previous decoding processor's result. Check nodes read hard words from queues 1 to 3. The bit-wise exclusive OR instruction is used to calculate the parity check constraints among those hard words. The results are check error patterns and these are stored back to the queue. As a result, before the

check node operation, queues 1 to 3 store hard bits. After the check node operation, queues 1 to 3 store check error patterns. Thus as data shifts rightwards along queues 1 to 3, the hard bits are eventually all overwritten with check error patterns.

Figure 4.8 shows the PIBF algorithm variable node operation. At the exit of each decoding processor, the variable node gets the check error patterns from the head of queues 1 to 3 and calculates the Flipping Pattern FP1 and FP2. The hard words in queue 0 are then flipped by XORing the flipping pattern computed by the variable node. At last, the flipped word in queue 0 is copied to queues 1 to 3 for the next decoding processor.

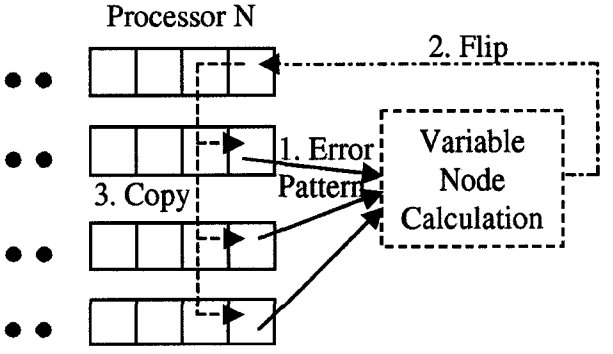

Figure 4.8: Parallel Bit Flipping Algorithm Variable Node Operation

At the end of the decoder, the word at the head of queue 0 is output directly. No hard decision is necessary since the packed bit values are binary already.

4.3 Computational Complexity and Coding Gain

In this section, we evaluated the trade-offs between the computational complexity and the coding gain of three LDPC-CC decoding algorithms.

In Table 4.1, the computational complexity and coding gain of the three alternative algorithms are shown. MS denotes the Min-Sum algorithm (Chapter 4.2.1), IBF denotes the Improved Bit Flipping algorithm (Chapter 4.2.2), and PIBF denotes

the Parallel Improved Bit Flipping algorithm (Chapter 4.2.3). The decoding instruction cycles/thread is the required number of the instruction cycles for the LDPC-CC decoding processor running on XInC thread 2 to thread 7. The decoded bits/second is calculated by

$$\frac{System\ Clock\ Frequency}{Decoding\ Instruction\ Cycles/Thread \times 8(threads)}.$$

We also included the decoded bits/second measure to help evaluate the scenario of using the new 100 MIPS XInC-II when 25% of the load can be used for LDPC-CC decoding. The result shows that the PIBF algorithm could produce 116 Kbps of decoded bit throughput, which could support CD quality audio applications. Note that IBF and PIBF requires a higher signal to noise ratio Eb/No for the same bit error rate than that of the Min-Sum algorithm. The required Eb/No for BER at 10^{-5} is also listed in the table.

All of the algorithms were run on the XInC-I multi-threaded microprocessor at 12 MHz. Six decoding processors are used on thread 2 to thread 7.

Table 4.1: LDPC-CC Decoding Algorithm Computational Complexity

	MS	IBF	PIBF
Decoding Instruction Cycles/Thread	679	315	432
Decoded Bits/Second	2.2 Kbps	4.8 Kbps	55.6 Kbps
Decoded Bits/Second assuming 100 MIPS XInC-II with 25% load	4.6 Kbps	9.6 Kbps	116 Kbps
Possible Application		Compressed voice	CD quality audio
Eb/No at BER=10^{-5}	3.8 dB	7.8 dB	7.8 dB

The Min-Sum (MS) algorithm has the most computational complexity among the three implemented algorithms. The decoded bits/second is only 2.3 Kbps. This bit rate would be too low for even compressed voice data. To achieve a bit error rate of 10^{-5}, the signal to noise ratio Eb/No requires at least 3.8 dB.

In the IBF algorithm, the time consuming minimum function in the Min-Sum algorithm is replaced by the exclusive OR instruction for the parity check and the addition for the check error counter. As a result, the decoding throughput is increased by 113% to 4.8 Kbps. Since only hard bits are used for decoding, to achieve a bit error rate of 10^{-5}, the minimum required signal-to-noise ratio Eb/No is 7.8 dB.

The PIBF algorithm is more complex than the IBF algorithm. Essentially more operations are required in the variable node operation to calculate the Flipping Pattern. With respect to the number of decoding clock cycles per thread, PIBF requires 40% additional instruction cycles compared with the IBF algorithm. However, due to the bit-wise parallelism, the decoding throughput could increase 10.3 times. In addition, the coding gain is at the same level of the IBF algorithm according to the simulation results.

Chapter 5

Hardware Optimization and Extensions

5.1 The XInC Emulator

A XInC emulator was designed at the beginning of this research project to allow the outputs and performance metrics to be predicted for alternative algorithms running with any given input. Specifically, the XInC emulator could be used for the following purposes.

First, the required instruction cycles per decoded bit of alternative LDPC-CC decoding algorithms can be evaluated. In the XInC, some instructions are 16-bit instructions while other instructions are 32-bit instructions (or 2-word instructions). To execute a 2-word instruction, two passes are required through the data processing pipeline. The number of instruction cycles per decoded bit is defined as the total number of instruction words required per decoded information bit. For a (128,3,6) LDPC-CC, each decoding phase decodes 1 information bit and 1 check bit. As a result, the instruction cycle per decoded bit is equal to the total number of instruction words that are required to implement one LDPC-CC decoding processor. The instruction cycles per decoded bit is the primary cost measurement for decoding algorithms since it is not related to hardware-specific parameters such as clock

frequency. It is a natural measurement of software algorithm complexity, which gives us how many instruction words are required to decode one information bit. Inside the XInC emulator, code profiling is used to determine exact cycle counts for program modules as well as instruction frequencies. This technique is widely used by software engineers to investigate the behavior of programs. For software engineers, code profiling is generally used to measure the frequency and duration of each function call. It can thus be used to identify the main bottlenecks in a program at the function call level. Software engineers can then focus their efforts on the most critical bottleneck functions. Instead of measuring the duration of function calls, the XInC emulator measures the frequency of instructions and the frequency of instruction formats. The frequency of an instruction is the total number of times that one instruction at a specific memory address is executed. There are a total of 18 instructions and 31 instruction formats in the XInC (Note: some instructions have 2 instruction formats). The frequency of an instruction format is defined to be the total number of times that one instruction of the given format are executed. This lower-level code profiling method can help to identify the algorithm bottlenecks at the instruction level.

Second, the emulator can be used to evaluate different hardware designs without having to build costly physical hardware system prototypes. It also allows us to simulate the effects of possible modifications to the hardware components. For example, some algorithm bottlenecks might be reduced or eliminated if certain additional hardware components were to be added. Before adding them to the real hardware, they could be added to the emulator model first. The performance of these new components could then be evaluated and confirmed using the emulator.

Third, using an emulator helps when debugging algorithms. More detailed debugging information, such as register values, can be collected and then displayed later for any clock cycle.

In the XInC emulator, the multi-threaded microprocessor is simulated on a clock cycle basis. Machine code is the input to the emulator. For each simulated clock cycle, one instruction word is read and its behaviour on the registers is simulated. Inside the emulator, memories, general-purpose registers, program counter registers and condition code registers are encapsulated by C++ objects. The emulator can be configured to execute a predefined number of clock cycles, or it can be allowed to execute instructions until the next breakpoint is encountered in the program. All the values of registers, memories and I/O ports can be read out. The frequency of each instruction and frequency of each instruction format are recorded as the emulator is running.

5.2 Hardware Optimization

The LDPC-CC decoding algorithm was analyzed on the XInC emulator. By code profiling, algorithm bottlenecks could be identified. In addition, various ways of extending the XInC microprocessor were studied to eliminate bottlenecks in the LDPC-CC application.

Table 5.1 shows the code profiling results for the check node calculation of the Bit Flipping algorithm for the same benchmark (128,3,6) LDPC-CC. The code segment reads hard bit inputs from the memory and calculates the check result. Through code profiling, we determined that the Bit Flipping algorithm check node operation needs 744 instructions. In addition, we could see that the actual check operation only takes one instruction at address 0xC3E1. The remaining instructions are used for reading the data from the memory. Those instructions include looping overhead, memory movement and pointer calculations.

Table 5.2 shows the frequency of the various instruction formats for a Min-Sum decoding processor which includes a check node and a variable node. There are a total of 31 instruction formats in the XInC. This profiling result helps us to

Table 5.1: Instruction Frequencies for the Bit Flipping Algorithm Check Node Calculation

Address	Frequency	Machine Code	Assembly Code
			// calculate check result
0xC3C6	6	0x1B80	mov r3,0
0xC3C7	6	0x2B80	mov r5,0
0xC3C8			loadPeTheseLLRsCond:
0xC3C8	42	0x4DFA	sub r1,r5,nCheckDegMax
0xC3C9	42	0x081A	bc ZS,loadPeTheseLLRsEnd
0xC3CA	36		loadPeTheseLLRsBody:
0xC3CA	36	0xB2A7	ld r6,r2,pnCheckDegRowPosition
0xC3CB	36	0x332E	add r6,r6,r5
0xC3CC	36	0x23F2 0xF0DC	ld r4,r2,pnPosition
0xC3CE	36	0x232C	add r4,r4,r5
0xC3CF	36	0x03F6 0xD290	ld r0,r6,matOnesInPcmYRowPosition
0xC3D1	36	0x8AAD	ld r1,r2,pnSymbolMatLLRPosition
0xC3D2	36	0x0BC1 0x0408	add r1,r1,nCodeC*nBufWidth* nProcSize
0xC3D4	36	0x0B41	sub r1,r1,r0
0xC3D5	36	0x03C1 0xE7C0	sub r0,r1,nBufLength
0xC3D7	36	0x2C01	bc NC,replacePnTheseRows
0xC3D8	35	0x0101	bra storePnTheseRows
0xC3D9			replacePnTheseRows:
0xC3D9	1	0x4800	add r1,r0,0
0xC3DA			storePnTheseRows:
0xC3DA	36	0x03F6 0xD596	ld r0,r6,matOnesInPcmX
0xC3DC	36	0x0308	add r0,r0,r1
0xC3DD	36	0x03FC 0xF154	st r0,r4,pnMatLLRPosition
0xC3DF	36	0x03F0 0xD89C	ld r0,r0,matLLRBuffer
0xC3E1	36	0x1DC3	xor r3,r3,r0
0xC3E2	36	0x6D01	add r5,r5,1
0xC3E3	36	0x01E4	bra loadPeTheseLLRsCond
0xC3E4	36		loadPeTheseLLRsEnd:

identify which instruction formats are the most frequently executed. As shown in the table, the three most frequent instructions are *bra k1 1* (unconditional branch), *add r1,r2,k3* (2's complement add) and *bc c1,k2* (conditional branch). Most of these instructions are associated with the looping overhead. The fifth to seventh most frequent instructions are *ld* (load from RAM) and *st* (store to RAM). They are associated with data movement.

After reviewing these profiling results, we grouped all the instructions into three groups: looping overhead, data movement and other operations. The looping overhead group includes *bra*, *add* and *bc*. Data movement group includes *ld* and *st*. The last operation group includes all the other instructions.

The instruction frequency for each group is shown in Table 5.3. In the table, MS stands for Min-Sum; IBF stands for Improved Bit Flipping algorithm; and PIBF stands for Parallel Improved Bit Flipping algorithm.

From Table 5.3, the looping overhead group accounts for 38%, 26% and 35% of the total instructions in MS, IBF and PIBF algorithm. The reason is that many parts of the algorithms are repeated, such as: loading and storing soft bits from memory, and performing check and variable node operations among these bits. In addition, most of the instruction segments inside the loop are short, making the looping overhead a relatively large proportion of the whole algorithm.

In traditional looping, the programmer generally requires 3 instructions to control the loop: 1) testing whether the loop ends , 2) jumping to the beginning of the loop, and 3) incrementing or decrementing the loop counter.

To eliminate the looping overhead, zero-overhead looping could be used. This hardware technique is already used in many Digital Signal Processors (DSPs) to improve the efficiency of general-purpose microprocessors when executing the loops that commonly appear in signal processing applications. By using specialized hardware, looping is controlled without cycles.

Table 5.2: Instruction Format Frequency of a Min-Sum Decoding Processor including a Check Node and a Variable Node

Instruction Format	Frequency
bra k1 1	672
add r1,r2,k3 1	414
bc c1,k2 1	329
add r1,r2,r3	285
ld r1,r2,k3 2	237
ld r1,r2,k3 1	174
st r1,r2,k3 2	128
mov r1,k2	117
sub r1,r2,r3	100
xor r1,r2,r3	96
st r1,r2,k3 1	94
add r1,r2,k3 2	87
ior r1,r2,r3	54
and r1,r2,r3	36
outp r1,k2	25
and r1,r2,k3	20
inp r1,k2	16
bis r1,r2,k3	12
jsr r1,k2	10
rol r1,r2,k3	10
bic r1,r2,k3	8
jsr r1,r2	8
thrd r1	6
ld r1,k2	4
st r1,k2	2
mov r1,k2 2	1
bc c1,k2 2	0
ior r1,r2,k3	0
rol r1,r2,r3	0
bra k1 2	0
bix r1,r2,k3	0

Table 5.3: LDPC-CC Decoding Algorithm Operation Frequency

Instruction Group	**MS**	**IBF**	**PIBF**
Looping Overhead	38%	26%	35%
Data Movement	19%	38%	30%
Other Operations	43%	36%	35%

Data movement is another significant activity of the LDPC-CC decoding algorithm. A large amount of memory storage is used to store the soft bit or hard bit information and intermediate check node and variable node results, and many instructions are required to read and manipulate the data. 19%, 38% and 30% of the total instructions are recorded for this purpose in MS, IBF and PIBF algorithms, respectively. In ASICs or FPGAs, this bottleneck may be minimized by customized memory design. For example, multiple data words could be accessed in parallel in one clock cycle by exploiting a multi-port memory. However, this is not available if we are limited to the original XInC architecture. Permitting the use of multi-port memory would require major changes to the given architecture.

5.2.1 Zero Overhead Looping

As shown in Table 5.3, more than 26% of the instructions in the decoding processor are associated with looping overhead. In the XInC, a typical loop might require 3 instructions (sub, bc, bra) for looping control, as shown below:

```
    mov r0,#loopCounter ; initialize loop counter
loopCont:
    sub r0,r0,1 ; decrement the loop counter, this is overhead
    bc ZS, loopEnd ; determine if looping ends, this is overhead
    . . .
    bra loopCont ; branch to the loop beginning, this is overhead
loopEnd:
```

In the XInC emulator, the zero-overhead looping mechanism was added and emulated in behavioral way. The looping time could be reduced by up to 75% if three instructions were used to control the loop and only one instruction was inside the loop. For each XInC thread, several new register sets are required to provide zero-overhead looping. Each register set includes a zero-overhead loop counter, a looping start register and a looping end register. The loop counter indicates how many loop iterations remains to be executed. The loop counter is decremented by 1 when the end of the loop is reached. The looping start register and looping end register store the looping program start address and looping program end address. In addition, two new instructions are created. One instruction will be denoted by movZOLR Rx, #counter. It initializes the looping counter *Rx* as #counter. The second new instruction will be denoted by setZOLA Rx,#endAddress. It copies the current Program Counter (PC) value to the looping start register. In addition, #endAddress is stored in the looping end register.

The new assembly language code of the zero-overhead looping could be,

```
movZOLR R0, #counter ; set loop counter
setZOLA R0, loopend ; set looping start address and end address
. .
loopend:
```

In Figure 5.1, the zero-overhead looping mechanism is represented in a flow chart. This algorithm would be enabled to run once for every instruction as long as the corresponding assembly language loop is enabled. It runs in parallel with instruction execution. The zero-overhead looping hardware monitors the Program Counter (PC) and looping end register. When the program counter value equals the looping end address, the PC is reloaded with the looping start address and the loop counter is decremented by 1. When the loop counter is decremented to 0, the looping is finished.

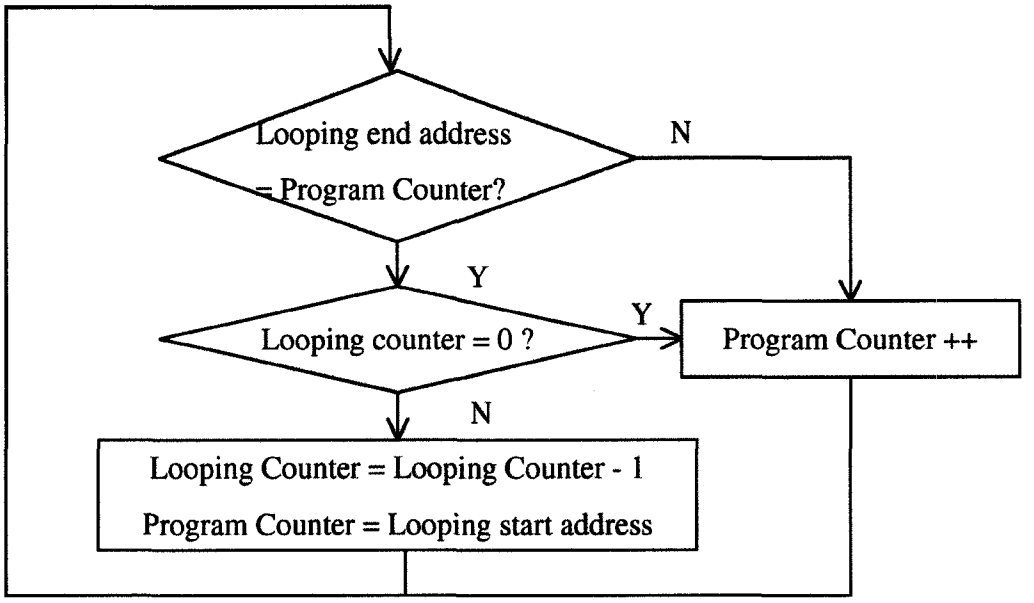

Figure 5.1: Zero-overhead Looping Flow Chart

In Figure 5.2, the example zero-overhead looping circuitry is given. The looping start register stores the looping program start address. The looping end register stores the looping program end address. The loop counter register stores the loop counter. The upper XOR gate is used to compare whether the looping program has reached the end of the program loop. If the program counter equals to the value in the looping end register, the XOR gate result is '0'. The bottom XOR gate is used to compare the loop counter with '0' to determine if the loop is finished. When the loop counter is decremented to '0', the XOR gate result is '0'. Two NOR gates and one AND gate are used to check whether both two XOR gate results are '0'. If this condition is true, the loopControl signal is '1'. The loop counter is added by "-1" and the program counter is set as the looping start register value.

It is also possible to use more complicated zero-overhead looping mechanisms. For example, the loop counter could be replaced by a loop counter start register, a loop counter end register, a loop counter step register and a loop counter direction

flag register. In this way, the loop counter could have more flexible loop start, loop end, loop step and loop direction.

Note that the maximum number of allowed nested loops in a program would be limited by the number of zero overhead loop circuits. This is a limitation, but a minor one for many signal processing algorithms.

5.2.2 Performance Evaluation of the Zero-overhead Looping

Zero-overhead looping was emulated behaviorally on the XInC emulator. After the LDPC-CC PIBF decoding algorithm was re-implemented and simulated, its decoding throughput was found to be further increased by 16%. Without zero-overhead looping, the total number of instructions per decoding processor is 356 and the total number of instruction cycles per decoding processor is 432. With zero-overhead looping, the total number of instructions per decoding processor is reduced by 22% to 278 and the total instruction cycles per decoding processor is reduced by 16% to 364.

Figure 5.3 shows the number of instruction cycles per decoding processor for the Min-Sum algorithm, Improved Bit Flipping algorithm, Parallel Improved Bit Flipping algorithm and Parallel Improved Bit Flipping algorithm with zero-overhead looping. The check node operation and variable node operation instruction cycles in one decoding processor are shown. The total number of instruction cycles is the sum of those two instruction cycles.

Min-Sum algorithm has complex check node and variable node operations such as the minimum function. Its check node requires 422 instruction cycles and its variable node requires 257 instruction cycles. The Improved Bit Flipping algorithm uses much simpler logical instructions, such as XOR, AND and OR, and its instruction cycles are consequently fewer than that of the Min-Sum algorithm. Its check node requires 262 instruction cycles and its variable node requires 53 instruction

cycles. The Parallel Improved Bit Flipping algorithm has a more complex variable node than the Improved Bit Flipping algorithm, which calculates the flipping patterns, FP1 and FP2. Its check node requires 252 instruction cycles and its variable node requires 180 instruction cycles. The Parallel Improved Bit Flipping algorithm with zero-overhead looping removes the looping overhead, so it requires less instruction cycles than the Parallel Improved Bit Flipping algorithm. Its check node instruction cycle count is 205 and its variable node cycle count is 159.

Figure 5.4 shows the decoding throughput of the 12 MIPS XInC-I microprocessor for the Min-Sum algorithm, the Improved Bit Flipping algorithm, the Parallel Improved Bit Flipping algorithm and the Parallel Improved Bit Flipping algorithm with zero-overhead looping. The decoding throughput is measured as the total decoded information bit per second. The Min-Sum algorithm's decoding throughput is 2.2 Kbps. The Improved Bit Flipping algorithm uses simple logical operations in the decoding process and the decoding throughput is 4.8 Kbps. The Parallel Improved Bit Flipping algorithm uses bit-wise parallelism to decode 16 bits at a time. Its decoding throughput is 55.6 Kbps. When the zero-overhead looping feature is used in Parallel Improved Bit Flipping algorithm, the decoding throughput could further be increased to 65.9 Kbps.

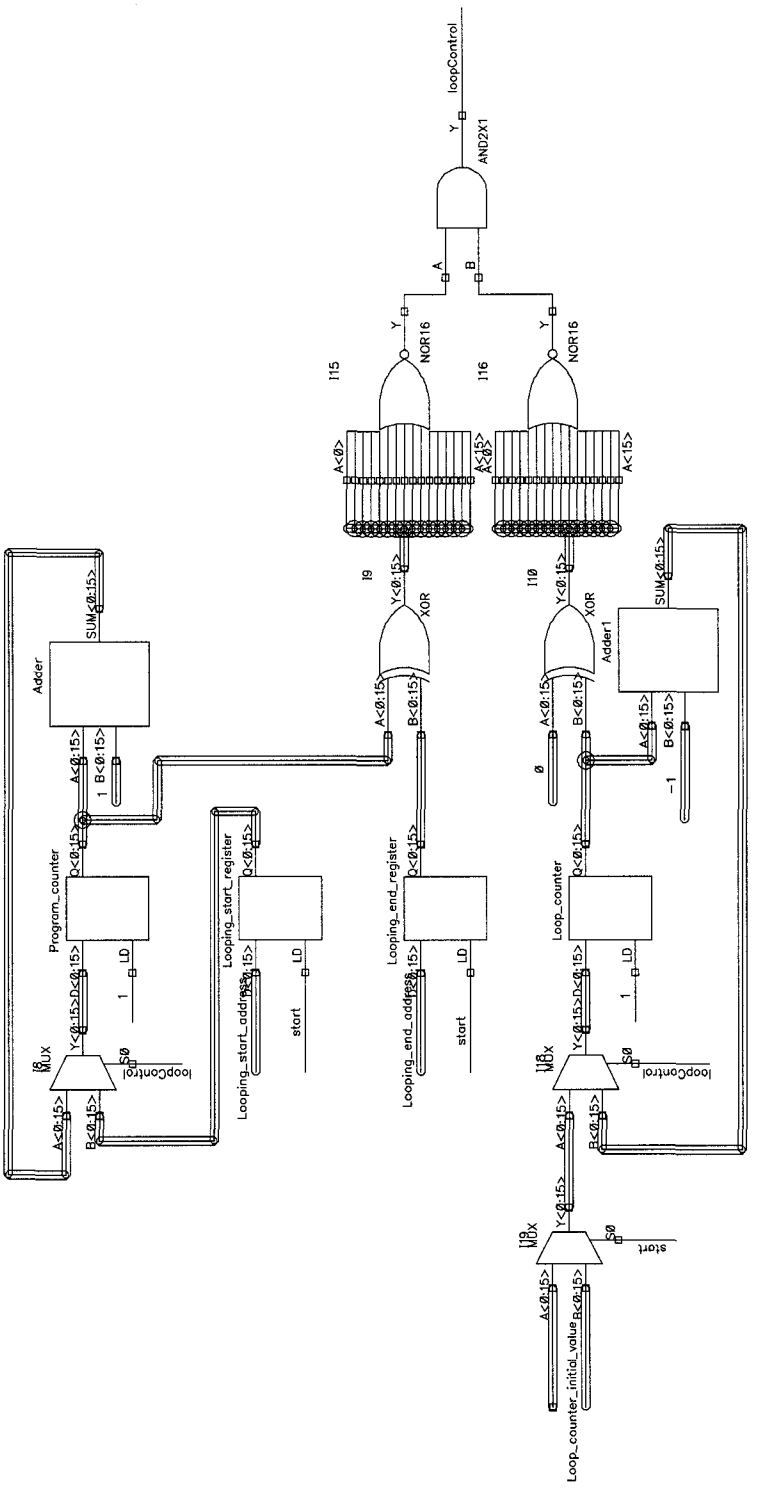

Figure 5.2: Example Zero-overhead Looping Circuitry

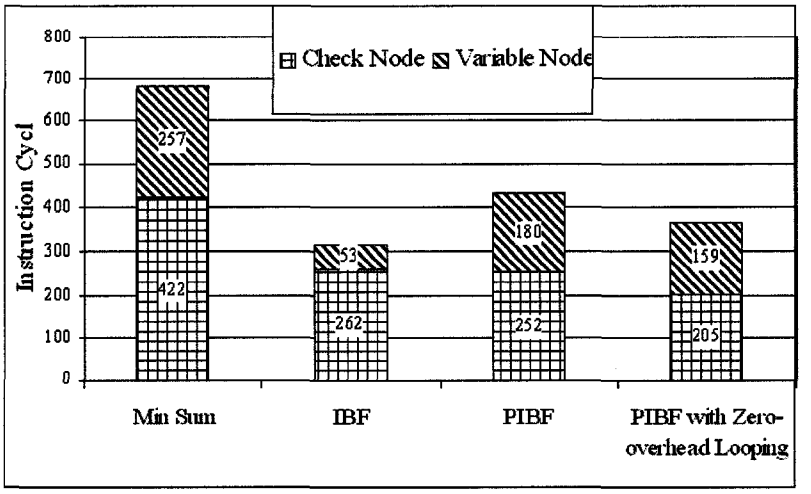

Figure 5.3: Comparison of Decoding Processor Instruction Cycle Counts

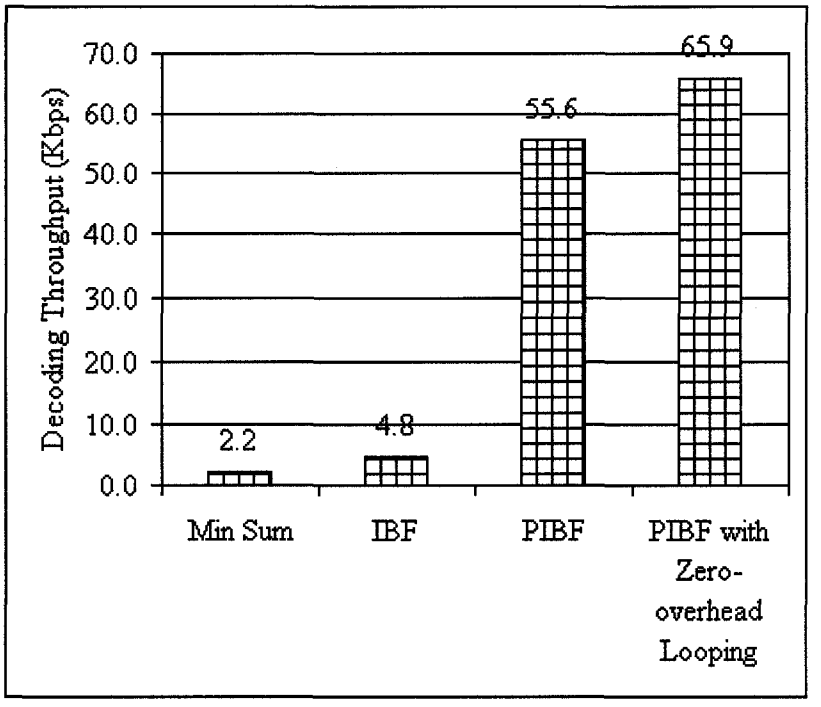

Figure 5.4: Comparison of Algorithm Throughput

Chapter 6

Future Research Directions and Conclusions

6.1 Future Research Directions

In this section, some preliminary ideas that might further improve the current algorithms and coding gain are discussed.

6.1.1 Longer LDPC-CCs

(M,J,K) LDPC-CCs with a larger memory window M might be implemented on multi-threaded microprocessors to increase the coding gain. The code implementations investigated in this thesis are readily extendible to handle such a change, provided there is sufficient available memory capacity in hardware.

For LDPC-BCs, the bit stream is decoded block-by-block. Check nodes and variable nodes in one iteration could be placed on silicon chip in parallel and they could be run at the same time. The same check nodes and variable nodes are shared by the following decoding iterations for the same block. For a LDPC-BC with a block length more than a thousand bits, thousands of variable nodes and hundreds of check nodes would be required. As a result, wire routing among these nodes becomes a critical challenge to implement long LDPC-BCs.

For LDPC-CCs, the bit streams is decoded continuously. Multiple decoding processors, which are each analogous to one decoding iteration for LDPC-BCs, are realized in parallel. Only one check node and one variable node are required in each decoding processor. Hence, the wire routing problem is greatly reduced for LDPC-CCs compared to LDPC-BCs.

Since the wire routing problem for LDPC-CCs is greatly reduced, longer LDPC-CCs could be implemented to get better coding gain. The computational complexity of LDPC-CC decoder would stay the same, except that longer LDPC-CCs would require more memories. In our current implementation, only the (128,3,6) LDPC-CC was used. If the LDPC-CC length could be increased to more than 1000, the coding gain might improve more than 2 dB according to simulation results reported in [8].

6.1.2 Precision

In the Bit Flipping algorithm, hard bits are used for decoding. During the decoding process, these bits are still kept as hard bits. Only one bit is used to represent the received signal.

In contrast, in the Min-Sum algorithm, soft bits are used. Received signals are represented as fixed-point numbers or integers. However, parity check operations on soft bits require complex computing methods and relatively large memory capacity. Received signals are normally sampled, quantized and then sent to the decoder directly.

In Figure 6.1, the Min-Sum algorithm was simulated with soft bit precisions of 2-, 3-, 4- and 8- bits. The bit error rate versus the signal-to-noise ratio E_b/N_o result is shown. The results confirm that coding gain is improved if more bits are used to represent the signal. When a soft bit is represented by a 4-, 3- and 2-bit number, the coding gain is close to that of the real number within 0.2, 0.5 and 2 dB, respectively.

When a soft bit is represented by a 8-bit number, the coding gain is almost the same as that of the real number.

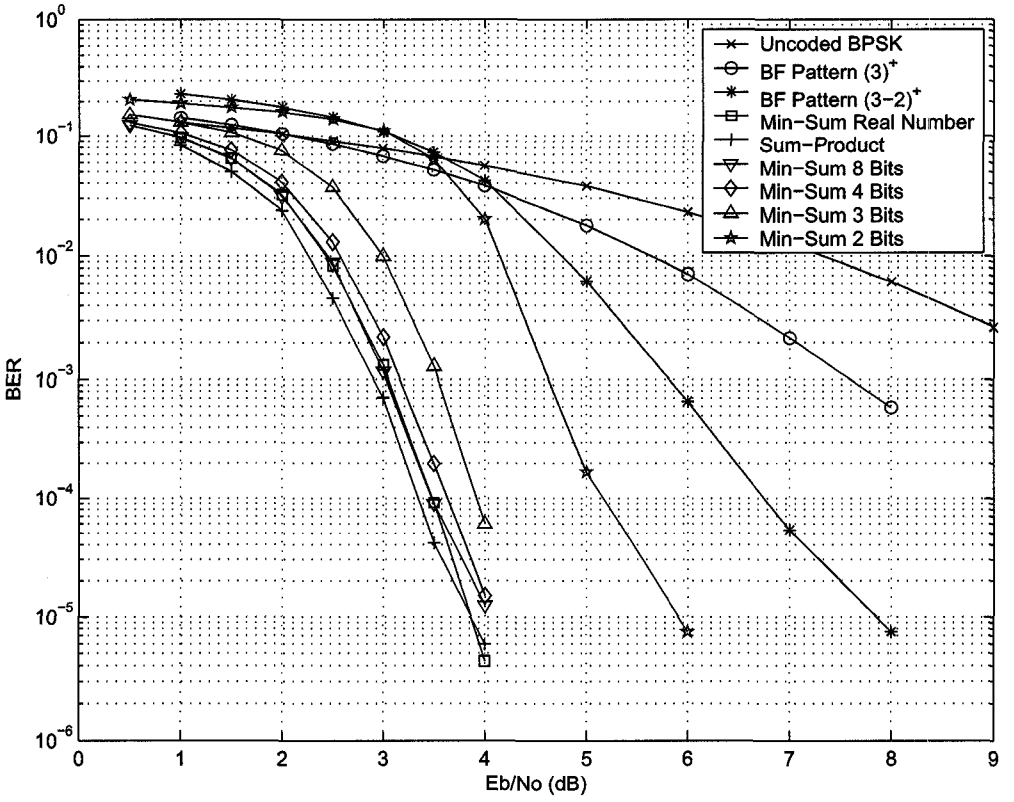

Figure 6.1: Min-Sum Algorithm Performance with Precision 8- ,4- ,3- and 2-Bits

According to this simulation results, decoding algorithm complexity could be further analyzed for different precisions and to determine their corresponding coding gains.

The coding gain of the Bit Flipping algorithm could be increased by increasing the soft bit precision from one bit to more bits. This would allow the algorithm to exploit some of the reliability information. For example, we could use "strong 1", "weak 1", "strong 0", "weak 0"soft bit values in a 2-bit representation. Note that the parity check operations could still be relatively simple logical operations. These

operations might be only a little more complex than that of the 1-bit algorithm. Bit-wise parallelism might be used. Hence, the coding gain could be increased by introducing more than a single bit of precision.

On the other hand, the precision of the Min-Sum algorithm could be limited to 4 bits with only 0.2 dB of coding gain loss compared to the real number representation. Since microprocessors could compute 16-, 32- or 64-bits one at a time, a modified Min-Sum algorithm might be able to decode 2, 4 or 8 bits concurrently. Such parallel algorithms could also be further investigated.

When only a few bits (for example, less than 3 bits) are used to represent the received signal samples, the check node and variable node operation could use a small lookup table (or a small network of logic gates) to replace the parity check operation. For example, for a 3-bit Min-Sum algorithm, 1 bit is used for the sign and 2 bits are used for the magnitude. The look-up table length for check nodes is only $2^{12} = 4096$ if the logic operation is based on the 2-bit magnitude for a (N,3,6) LDPC-CC. With look-up tables, the check node calculation for 2-bit soft bits might be as fast as for the bit-flipping algorithm.

6.1.3 Hybrid Decoder Design

Hybrid decoder designs might be worth investigating when the Signal-to-Noise Ratio cannot be pre-determined. In such a situation, the decoders could switch between the Min-Sum algorithm and the Bit Flipping algorithm according to the channel conditions (e.g. Signal-to-Noise Ratio). When the SNR is high, the Bit Flipping algorithm could be used to reduce the decoding time and save power consumption. When the SNR is low, Min-Sum algorithm could be used to achieve better coding gain in exchange of more computing complexity.

6.1.4 Adaptive Bit Flipping Algorithm

As we discussed bit flipping in Chapter 3.1, some error bits might never get flipped (i.e. corrected) if the threshold is too conservative. On the other hand, an overly aggressive threshold could cause some of the correct bits to be flipped erroneously. According to the simulation result, the best threshold value might be related to the bit error rate. As a result, the bit flipping algorithm might automatically choose the proper threshold value according to the estimated bit error rate. If the estimated bit error rate is increasing, the algorithm could decide that the current threshold value is too aggressive and is thus causing too many errors, and hence a more conservative threshold should be chosen. On the other hand, if the bit error rate could not be reduced further using the current threshold value, the algorithm could decide that the current threshold value might be too conservative. Then a more aggressive threshold value could be used for further decoding. Such adaptive algorithms might help to increase the coding gain and speed up the algorithm further.

6.2 Main Contributions and Conclusions

Error control coding is widely used in the communication field to combat transmission errors caused by noise disturbances. By adding appropriate redundant information to the transmitted information, the contaminated signal can be recovered at the receiver without error. In his landmark 1948 paper [1], Claude Shannon showed that for any given channel bandwidth and signal power to noise power ratio (SNR), there exists a maximum bit rate at which information can be encoded and decoded without error at the receiver. Since then, information theorists have searched for coding methods whose performance could approach the Shannon Limit. After 50 years of research, low-density parity-check block codes (LDPC-BCs) were found to be a class of capacity-approaching codes within 0.0034 dB of the Shannon Limit [4]. However, these codes require complex encoding and decoding algorithms to

implement. Current research on LDPC codes is focused mainly on code constructions with implementable encoding and decoding algorithms.

One of the difficulties when implementing LDPC block codes is their complex encoder since the generator matrix is no longer low density. Moreover, Low Density Parity Check Convolutional Codes (LDPC-CCs) have a simpler encoder structure inherited from traditional convolutional codes.

Current popular decoding algorithms for LDPC codes process bit reliability information from the received signal. These algorithms involve relatively complex fixed-point calculations. In order to realize the desired decoding throughput, these decoders have to be implemented in ASIC or FPGA technology. However, there are many advantages to implementing LDPC codes in microprocessors. First, the encoder and decoder algorithms could share the same microprocessor resources with other algorithms and this would reduce product costs. As an example of a wireless audio application, audio compression and decompression algorithms could be implemented with LDPC-CC encoding and decoding algorithms in the same microprocessor. Second, the development cycle when using a microprocessor is fast and the development cost is low while the ASIC fabrication process is expensive and costs several months of engineering time.

In this thesis, several decoding algorithms were investigated. The main contributions of this research project are as follows.

1. The multi-threaded microprocessor was found to be a suitable architecture for implementing LDPC-CC decoding algorithms. The iterative decoding process was realized by several identical decoding processors which could be mapped to multiple threads on a multi-threaded microprocessor. In addition, the built-in bit-wise parallelism in microprocessor could decode 16, 32, or 64 bits in one time when the bit flipping algorithm is used.

2. The Bit Flipping algorithm was found to be suitable for high Signal-to-Noise

Ratio applications. Popular decoding algorithms, such as the Sum-Product algorithm or the Min-Sum algorithm, might not be suitable for microprocessors since they require a relatively large amount of calculation on fixed-point numbers. On the contrary, the BF algorithm only uses hard bit and logical operations and can be implemented by exploiting the bit-wise parallelism that is already present in the instruction sets of most microprocessors.

3. The coding gain of the original Bit Flipping algorithm was improved by the discovery of the bit flipping threshold pattern. Through simulation, the best pattern was determined for a (128,3,6) LDPC-CC. Alternating between a conservative threshold (where a bit is flipped only when all the parity check constraints fail) and an aggressive threshold (where a bit is flipped when the majority of the parity check constraints fail) not only improves the coding gain, it also speeds up the error correction convergence. The Improved Bit Flipping algorithm with bit flipping threshold pattern $(3-2)^+$ achieves 2.5 dB better coding gain compared to Gallager's original algorithm with a fixed bit flipping threshold pattern $(3)^+$ at a Bit Error Rate of 10^{-4}. It also has the coding gain within 3.5 dB of the Min-Sum algorithm. In addition, the decoding throughput is 24 times faster than the benchmark Min-Sum algorithm.

4. A XInC emulator was built to quantitatively analyze the performance of the algorithms. Looping overhead and data movement were identified as being the main bottlenecks. Zero-overhead looping was added to the emulator to permit experiments that could measure the benefits of the new feature on performance. The emulation results show that the decoding throughput could be further increased by 16% using this hardware improvement.

Bibliography

[1] C. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, 1948.

[2] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," in *Proc. 1993 IEEE international conference on communications*, 1993, pp. 1064–1070.

[3] R. G. Gallager, "Low density parity check codes," *IRE Trans. Inform. Theory*, vol. 8, pp. 21–28, Jan. 1962.

[4] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *IEE Electron. Lett.*, vol. 32, pp. 1645–1655, Aug. 1996.

[5] "Digital video broadcasting (DVB) user guidelines for the second generation system for broadcasting, interactive services, news gathering and other broadband satellite applications (DVB-S2)," Tech. Rep. TR 102 376 V1.1.1, European Telecommunications Standards Institute, Feb 2005.

[6] IEEE Computer Society, the IEEE Microwave Theory, and Techniques Society, "IEEE standard for local and metropolitan area networks part 16: Air interface for fixed and mobile broadband wireless access systems," Tech. Rep. IEEE Std 802.16e-2005 and IEEE Std 802.16-2004/Cor1-2005, Institute of Electrical and Electronics Engineers, Feb 2006.

[7] P. Elias, "Coding for noisy channels," in *IRE Conv. Rec., pt.4*, Mar 1955, pp. 37–46.

[8] A. Jiménez Felström and K. Zigangirov, "Time-varying periodical convolutional codes with low-density parity-check matrix," *IEEE Trans. Inf. Theory*, vol. 45, pp. 2181–2191, Sept. 1999.

[9] Z. Chen, S. Bates, and X. Dong, "Low-density parity-check convolutional codes applied to packet based communication systems," in *Proc. IEEE GLOBECOM 2005*, Nov. 2005, vol. 3, pp. 1250–1254.

[10] S. Bates and G. Block, "A memory-based architecture for FPGA implementations of low-density parity-check convolutional decoders," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2005, pp. 336–339.

[11] R. Swamy, S. Bates, and T. L. Brandon, "Architectures for ASIC implementations of low-density parity-check convolutional encoders and decoders," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2005, pp. 4513–4516.

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

[13] N. Wiberg, *Codes and decoding on general graphs*, Ph.D. thesis, Linkoping Univ., 1996.

[14] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach, Third Edition*, Morgan Kaufmann, 2002.

[15] Eleven Engineering Incorporated, *XInC User Guide*, 2002.

[16] Shu Lin and Daniel J. Costello, *Error Control Coding, Second Edition*, Prentice Hall, 2004.

[17] Leon W. Couch, *Digital and Analog Communication Systems, Seventh Edition*, Prentice Hall, 2006.

[18] D. J. Costello and G. D. Forney, "Channel coding: The road to channel ca-

pacity," *Proceedings of the IEEE*, vol. 95, pp. 1150–1177, June 2007.

[19] C. Schlegel and L. C. Perez, "On error bounds and turbo codes," *IEEE Commun. Lett.*, vol. 7, pp. 205–207, July 1999.

[20] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inf. Theory*, vol. IT-27, pp. 533–547, Sept. 1981.

[21] T. Richardson and R. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, pp. 638–656, Feb. 2001.

[22] T. J. Richardson and R. L. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, pp. 638–656, 2001.

[23] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inf. Theory*, vol. 13, pp. 260–269, 1967.

[24] F. R. Kschischang, B. J. Frey, and H. A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inf. Theory*, vol. 47, pp. 498–519, Feb. 2001.

[25] S. L. Howard, C. Schlegel, and V. C. Gaudet, "Degree-matched check mode decoding for regular and irregular LDPCs," vol. 53, pp. 1054–1058, Oct. 2006.

[26] J. Chen and M. P. C. Fossorier, "Density evolution for two improved BP-based decoding algorithms of LDPC codes," *IEEE Commun. Lett.*, vol. 6, no. 5, pp. 208–210, May 2002.

[27] J. Zhang, M. Fossorier, D. Gu, and J. Zhang, "Two-dimensional correction for min-sum decoding of irregular LDPC codes," *IEEE Commun. Lett.*, vol. 10, no. 3, pp. 180–182, Mar. 2006.

[28] N. Miladinovic and M. Fossorier, "Improved bit-flipping decoding of low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 51, pp. 1594–1606, Apr. 2005.

[29] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, 1988.

[30] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, no. 9, pp. 948–960, Sept. 1972.

[31] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manuals*, 2002.

[32] G. Byrd and M. Holliday, "Multithreaded processor architectures," *IEEE Spectrum*, vol. 32, no. 8, pp. 38–46, Aug. 1995.

[33] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith, "The Tera computer system," in *Proceedings of the 1990 International Conference on Supercomputing*, 1990, pp. 1–6.

[34] R. Krishnaiyer, D. Kulkami, D. Laven, L. Wei, C.-C. Lim, J. Ng, and D. Sehr, "An advanced optimizer for the IA-64 architecture," *IEEE Micro*, vol. 20, no. 6, pp. 60–68, 2000.

[35] Jay Bharadwaj, William Y. Chen, Weihaw Chuang, Gerolf Hoflehner, Kishore Menezes, Kalyan Muthukumar, and Jim Pierce, "The Intel IA-64 compiler code generator," *IEEE Micro*, vol. 20, no. 5, pp. 44–53, 2000.

Appendix A

XInC Emulator C++ Source Code

```
// Filename: xinc.cpp
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Class xinc implementation
//
// Date: Jan 24, 2008

#include "xinc.h"
#include "xinclib.h"
#include <cstring>
#include <string>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <conio.h>
#include <cstdio>

using namespace std;

xinc::xinc ()
{
    systemClockCycle=0;

    for (int i=0;i<8;i++) {
        isTwoWordsInstruction[i]=false;
        firstWord[i]=0;
        braStatus[i]=0;

    }

    newestLog="";

    for (i=0;i<65536;i++) {
        addressStat[i]=0;
    }
```

```
    fileLog=false;
    screenLog=false;
    serialInOpen=false;
    serialOutOpen=false;
    instructionLog2=false;

    instructionName[0]="add r1,r2,r3";
    instructionName[1]="add r1,r2,k3 1";
    instructionName[2]="add r1,r2,k3 2";
    instructionName[3]="and r1,r2,r3";
    instructionName[4]="and r1,r2,k3";
    instructionName[5]="bc c1,k2 1";
    instructionName[6]="bc c1,k2 2";
    instructionName[7]="bic r1,r2,k3";
    instructionName[8]="bis r1,r2,k3";
    instructionName[9]="bix r1,r2,k3";
    instructionName[10]="bra k1 1";
    instructionName[11]="bra k1 2";
    instructionName[12]="inp r1,k2";
    instructionName[13]="ior r1,r2,r3";
    instructionName[14]="ior r1,r2,k3";
    instructionName[15]="jsr r1,r2";
    instructionName[16]="jsr r1,k2";
    instructionName[17]="ld r1,r2,k3 1";
    instructionName[18]="ld r1,r2,k3 2";
    instructionName[19]="ld r1,k2";
    instructionName[20]="mov r1,k2";
    instructionName[21]="mov r1,k2 2";
    instructionName[22]="outp r1,k2";
    instructionName[23]="rol r1,r2,r3";
    instructionName[24]="rol r1,r2,k3";
    instructionName[25]="st r1,r2,k3 1";
    instructionName[26]="st r1,r2,k3 2";
    instructionName[27]="st r1,k2";
    instructionName[28]="sub r1,r2,r3";
    instructionName[29]="thrd r1";
    instructionName[30]="xor r1,r2,r3";
    instructionName[31]="movZOLR r1,k3";
    instructionName[32]="setZOLA r1,k3";

    for (i=0;i<33;i++) instructionStat[i]=0;

};


bool xinc::load(char* filename)
{
    ifstream fp_in;

    if (strcmp(&filename[strlen(filename)-3],"hex")!=0) {
        cout << "Error: File should be ended with \"hex\"" << endl;
        return false;
    }
    fp_in.open(filename,ios::in | ios::binary );

    if (fp_in.fail())
    {
        cout << "Error: Input file open fail!" << endl;
        return false;
    }
```

```
    int i=0;
    unsigned char firstByte,secondByte;
    short twoBytes;
    firstByte=fp_in.get();
    secondByte=fp_in.get();
    while (!fp_in.eof()) {

        twoBytes=(secondByte << 8) | firstByte;

        if (i>=49152) {
            ram[i-49152].setValue(twoBytes);
        }
        i++;
        firstByte=fp_in.get();
        secondByte=fp_in.get();
    }
    fp_in.close();
    cout << "File loaded successfully!" << endl;
    strncpy (filenameRoot,filename,strlen(filename)-4);
    filenameRoot[strlen(filename)-4]=0;
    return true;

};


void xinc::setTwoWordsInstruction(bool isTwoWords,short threadNum)
{
    isTwoWordsInstruction[threadNum]=isTwoWords;
}

bool xinc::getTwoWordsInstruction(short threadNum)
{
    return isTwoWordsInstruction[threadNum];
}

void xinc::setFirstWord(short firstWordValue,short threadNum)
{
    firstWord[threadNum]=firstWordValue;
}

short xinc::getFirstWord(short threadNum)
{
    return firstWord[threadNum];
}


bool xinc::isThreadRun(int threadNum)
{
    if ((peripheralRegisterWrite[4].getValue() & (1 << threadNum)) ==0) {
        return true;
    }
    else {
        return false;
    }
};

void xinc::iMov(int threadNum,short instruction)
{
    if (getTwoWordsInstruction(threadNum)) {
```

```
            short r= (getFirstWord(threadNum) & 0x3800) >> 11;
            short k2=instruction;
            thread[threadNum].setR(k2,r);
    //      cout << "mov R1,K2 W2, second word" ;

            addLog ("mov R"+unsignedLong2String(r)+","+signedShort2String(k2)
                +"(0x"+short2HexString(k2)+")",threadNum);
    }
    else if ((instruction & 0xc7c0) == 0x0380) { // mov R1,K2 W1
        short r=(instruction & 0x3800) >> 11;
        short k2=signedExtension(instruction & 0x3f,6);
        thread[threadNum].setR(k2,r);
    //  cout << "mov R1,K2 W1" ;
        addLog ("mov R"+unsignedLong2String(r)+","+signedShort2String(k2)+"(0x"+
            short2HexString(k2)+")",threadNum);

    }
    else if ((instruction & 0xc7ff) == 0x03e8) { // mov R1,K2 W2
        setTwoWordsInstruction(true,threadNum);
        setFirstWord(instruction,threadNum);
    //  cout << "mov R1,K2 W2" ;


    }

}

void xinc::iOutp(int threadNum,short instruction)
{
    string log;

    if (getTwoWordsInstruction(threadNum)) {

    }
    else {
        unsigned short k2=instruction & 0x7f;
        short r=(instruction & 0x3800) >> 11;
        peripheralRegisterWrite[k2].setValue(thread[threadNum].getR(r));


        if (!peripheralRegisterWrite[k2].getIsOutputSet()) {
            peripheralRegisterWrite[k2].setIsOutputSet(true);
        }
        else {

            // cout << "Warning: Output" << k2 << "already set!" << endl;
        }


        if (k2==32) {
            if (SPI0tx==1) {
                SPI0tx=0;
                short controlByte;
                controlByte=thread[threadNum].getR(r);
                if (controlByte==0x4000) {    // readConfiguration
                    peripheralRegisterRead[k2].setValue(0x00);
                }
                else if (controlByte==0x00) {     // readByte
                    if (serialInOpen) {
                        if (!fp_serialIn.eof()) {
```

```
                        peripheralRegisterRead[k2].setValue(fp_serialIn.get());
                    }

                }
            }
            else if ((controlByte <0) && ((unsigned short)controlByte != 0xc00f)) {
                //writeByte  // 0xc00f is writeConfigure
                if (serialOutOpen) {
                    fp_serialOut.put((unsigned char) (controlByte & 0xff));
                    fp_serialOut.flush();
                    peripheralRegisterRead[k2].setValue(0x00);

                }

            }
            else {
                peripheralRegisterRead[k2].setValue(0x00);
                cout << "Error: I/O control word error - " <<
                     thread[threadNum].getR(r) << endl;
            }

        }
        else {
            SPI0tx=1;
            short controlByte;
            controlByte=thread[threadNum].getR(r) & 0xff;
            if (controlByte==0x40) {    // readConfiguration
                peripheralRegisterRead[k2].setValue(0x40);
            }
            else if (controlByte==0x00) {     // readByte
                peripheralRegisterRead[k2].setValue(0xc2);
            }
            else if ((short)(controlByte << 8)<0) {     //writeByte
                peripheralRegisterRead[k2].setValue(0x00);
            }
            else {
                peripheralRegisterRead[k2].setValue(0x00);
                cout << "Error: I/O control word error - " <<
                    thread[threadNum].getR(r) << endl;
            }

        }

    } // SPI0tx

//      cout << "outp r1,k2" ;
    log="outp R"+unsignedLong2String(r)+","+signedShort2String(k2)+
        "(0x"+short2HexString(k2)+")";
    log= log+ " IO[0x"+short2HexString(k2)+"]="+
        signedShort2String(thread[threadNum].getR(r))+"(0x"+
        short2HexString(thread[threadNum].getR(r))+")  ";
    addLog (log,threadNum);
  }
}

void xinc::iBra(int threadNum,short instruction)
{
  string log;
```

```
    if (getTwoWordsInstruction(threadNum)) {
            short k1= instruction ;
            log="bra "+signedShort2String(k1)+"(0x"+short2HexString(k1)+")";
            log= log+ " PC=0x"+short2HexString((unsigned)
                (thread[threadNum].getPc()+k1+1))+" ";
            addLog (log,threadNum);

            thread[threadNum].setPc((unsigned)(thread[threadNum].getPc()+k1));
    //      cout << "bra k1 W2, second word" ;


    }
    else {

        if ((instruction & 0xff) != 0x0) { // bra K1 W1
             short k1= signedExtension(instruction & 0xff,8);
             if (k1 != -1) {
                 braStatus[threadNum]=0;
             }
             else {
                 if (braStatus[threadNum]>=1) {
                    braStatus[threadNum]=2;
                     // bra @ many times, no log for clean
                 }
                 else {
                     braStatus[threadNum]=1;     // bra @ first times
                     peripheralRegisterRead[4].setValue(peripheralRegisterRead[4].
                        getValue() | (1<< threadNum));
                 }
             }
             log="bra "+signedShort2String(k1)+"(0x"+short2HexString(k1)+")";
             log= log+ " PC=0x"+short2HexString((unsigned)
                (thread[threadNum].getPc()+k1+1))+" ";
             if (braStatus[threadNum]!=2) {
                addLog (log,threadNum);
             }
             thread[threadNum].setPc((unsigned)(thread[threadNum].getPc()+k1));
             instructionStat[10]++;
             newestILog2=newestILog2+"|10";
        //   cout << "bra K1 W1" ;

        }
        else { // bra K1 W2
            setTwoWordsInstruction(true,threadNum);
            setFirstWord(instruction,threadNum);
            instructionStat[11]++;
            newestILog2=newestILog2+"|11";
    //      cout << "bra K1 W2" ;
        }
    }
}

void xinc::iAdd(int threadNum,short instruction,short method)
{
    short r1,r2,r3;
    short add1,add2,addResult;
    int addResult1;
    string log="";
    switch (method)
    {
```

```
    case 0:

        r1=(instruction & 0x3800) >> 11;
        r2=(instruction & 0x7) ;
        r3=(instruction & 0x38) >> 3;

        add1=thread[threadNum].getR(r2);
        add2=thread[threadNum].getR(r3);


//      cout << "add r1,r2,r3" ;
        log="add R"+unsignedLong2String(r1)+",R"+
            unsignedLong2String(r2)+",R"+unsignedLong2String(r3)+" ";


        break;
    case 1:
        r1=(instruction & 0x3800) >> 11;
        r2=(instruction & 0x700) >> 8;
        add1=thread[threadNum].getR(r2);
        add2= signedExtension(instruction & 0xff,8);
//        cout << "add r1,r2,k3" ;
        log="add R"+unsignedLong2String(r1)+",R"+
            unsignedLong2String(r2)+","+signedShort2String(add2)
            +"(0x"+short2HexString(add2)+")";
        break;
    case 2:
        if (getTwoWordsInstruction(threadNum)) {
            r1=(getFirstWord(threadNum) & 0x3800) >> 11;
            r2=(getFirstWord(threadNum) & 0x7) ;
            add1=thread[threadNum].getR(r2);
            add2=instruction;
//          cout << "add r1,r2,k3 W2" ;
            log="add R"+unsignedLong2String(r1)+",R"+
                unsignedLong2String(r2)+","+
                signedShort2String(add2)+"(0x"+short2HexString(add2)+")";

        }
        else {
            setTwoWordsInstruction(true,threadNum);
            setFirstWord(instruction,threadNum);
//          cout << "add r1,r2,k3 w1" ;
            return;
        }
        break;
    default:

}

addResult=add1+add2;
addResult1=(unsigned short)add1+(unsigned short)add2;
log= log+ " R"+unsignedLong2String(r1)+"="+
    signedShort2String(addResult)+"(0x"+short2HexString(addResult)+") ";
if (addResult==0) {
    thread[threadNum].setStatusZ();
    log=log+"Z=1,";
}
else {
    thread[threadNum].clrStatusZ();
    log=log+"Z=0,";
```

```
    }
    if (addResult <0) {
        thread[threadNum].setStatusN();
        log=log+"N=1,";

    }
    else {
        thread[threadNum].clrStatusN();
        log=log+"N=0,";
    }
    if ((add1>0 && add2>0 && addResult <0) || (add1<0 && add2<0 && addResult >0)) {
        thread[threadNum].setStatusV();
        log=log+"V=1,";
    }
    else {
        thread[threadNum].clrStatusV();
        log=log+"V=0,";
    }
    if (addResult1 >65535) {
        thread[threadNum].setStatusC();
        log=log+"C=1";
    }
    else {
        thread[threadNum].clrStatusC();
        log=log+"C=0";
    }
    thread[threadNum].setR( addResult,r1);
    addLog (log,threadNum);


}

void xinc::iAnd(int threadNum,short instruction,short method)
{
    short r1,r2,r3;
    short and1,and2,andResult;
    string log;

    switch (method)
    {
        case 0:

            r1=(instruction & 0x3800) >> 11;
            r2=(instruction & 0x7) ;
            r3=(instruction & 0x38) >> 3;

            and1=thread[threadNum].getR(r2);
            and2=thread[threadNum].getR(r3);
    //        cout << "and r1,r2,r3" ;
            log="and R"+unsignedLong2String(r1)+",R"+
                unsignedLong2String(r2)+",R"+unsignedLong2String(r3)+" ";

            break;

        case 1:
            if (getTwoWordsInstruction(threadNum)) {
                r1=(getFirstWord(threadNum) & 0x3800) >> 11;
                r2=(getFirstWord(threadNum) & 0x7) ;
                and1=thread[threadNum].getR(r2);
```

```
                and2=instruction;
//              cout << "and r1,r2,k3 w2" ;
                log="and R"+unsignedLong2String(r1)+",R"+
                    unsignedLong2String(r2)+","+signedShort2String(and2)
                    +"(0x"+short2HexString(and2)+")";

            }
            else {
                setTwoWordsInstruction(true,threadNum);
                setFirstWord(instruction,threadNum);
//              cout << "and r1,r2,k3 w1" ;
                return;
            }
            break;
        default:
            ;
    }

    andResult=and1 & and2;
    log= log+ " R"+unsignedLong2String(r1)+"="+
        signedShort2String(andResult)+
        "(0x"+short2HexString(andResult)+") ";


    if (andResult==0) {
        thread[threadNum].setStatusZ();
        log=log+"Z=1,";
    }
    else {
        thread[threadNum].clrStatusZ();
        log=log+"Z=0,";
    }
    if (andResult<0) {
        thread[threadNum].setStatusN();
        log=log+"N=1,";
    }
    else {
        thread[threadNum].clrStatusN();
        log=log+"N=0,";
    }
    if ((and1|and2) != and2 ) {
        thread[threadNum].setStatusV();
        log=log+"V=1,";
    }
    else {
        thread[threadNum].clrStatusV();
        log=log+"V=0,";
    }
    if ((andResult>0) && (andResult < 256)) {
        thread[threadNum].setStatusC();
        log=log+"C=1";
    }
    else {
        thread[threadNum].clrStatusC();
        log=log+"C=0";
    }
    thread[threadNum].setR( andResult,r1);
    addLog (log,threadNum);
}
```

```
void xinc::iBc(int threadNum,short instruction,short method)
{
    short displacement;
    short condition;
    string log;

    switch (method)
    {
        case 0:

            displacement=signedExtension(instruction & 0xff,8);
            condition=(instruction & 0x3c00) >> 10;
    //      cout << "bc c1,k2" ;

            log="bc C"+unsignedLong2String(condition)+","+
                signedShort2String(displacement)+"(0x"+
                short2HexString(displacement)+")";
            break;

        case 1:
            if (getTwoWordsInstruction(threadNum)) {
                displacement=instruction;
                condition=(getFirstWord(threadNum) & 0x3c00) >> 10;
    //          cout << "bc c1,k2 w2" ;

                log="bc C"+unsignedLong2String(condition)+","+
                    signedShort2String(displacement)+"(0x"+
                    short2HexString(displacement)+")";
            }
            else {
                setTwoWordsInstruction(true,threadNum);
                setFirstWord(instruction,threadNum);
//              cout << "bc c1,k2 w1";
                return;
            }
            break;
        default:
            ;
    }
    bool test=false;
    switch (condition) {
        case 0:
            test=thread[threadNum].isC1();   // c
            break;
        case 1:
            test=thread[threadNum].isV1();   // v
            break;
        case 2:
            test=thread[threadNum].isZ1();   // z
            break;
        case 3:
            test=thread[threadNum].isN1();   // n
            break;
        case 4:
            test=thread[threadNum].isC1() ||
                thread[threadNum].isZ1();    // c|z
            break;
        case 5:
            test=xor2(thread[threadNum].isN1(),
                thread[threadNum].isV1());   // n ^ v
```

```
            break;
        case 6:
            test=xor2(thread[threadNum].isN1(),
                thread[threadNum].isV1()) || thread[threadNum].isZ1();
            // (n^v)|z
            break;
        case 7:
            test=thread[threadNum].isN1() || thread[threadNum].isZ1();
            // n|z
            break;
        case 8:
            test=!thread[threadNum].isC1(); // !c
            break;
        case 9:
            test=!thread[threadNum].isV1();  // !v
            break;
        case 10:
            test=!thread[threadNum].isZ1();  // !z
            break;
        case 11:
            test=!thread[threadNum].isN1();  // !n
            break;
        case 12:
            test=!(thread[threadNum].isC1() || thread[threadNum].isZ1());
            // !(c|z)
            break;
        case 13:
            test=!xor2(thread[threadNum].isN1(),thread[threadNum].isV1());
            // !(n^v)
            break;
        case 14:
            test=!(xor2(thread[threadNum].isN1(),thread[threadNum].isV1())
             || thread[threadNum].isZ1());  // ((n^v)|z)
            break;
        case 15:
            test=!(thread[threadNum].isN1() || thread[threadNum].isZ1());
            // !(n|z)
            break;
        default:
            cout << "bc condition error: condition:" << condition << endl;

    }
    if (test) {
        log= log+ " PC=0x"+short2HexString(thread[threadNum].getPc()+
            displacement+1)+" ";
        addLog (log,threadNum);
        thread[threadNum].setPc(thread[threadNum].getPc()+displacement);

    }
    else {
        addLog (log,threadNum);
    }

}

void xinc::iBic(int threadNum,short instruction)
{
    short r1,r2;
    short op1,op2,opResult;
    string log;
```

```
    r1=(instruction & 0x3800) >> 11;
    r2=(instruction & 0x7) ;

    op1=thread[threadNum].getR(r2);
    op2=(instruction & 0x78) >> 3;
    opResult=op1 & (~(1<<op2));

    log="bic R"+unsignedLong2String(r1)+",R"+unsignedLong2String(r2)+","+
        signedShort2String(op2)+"(0x"+short2HexString(op2)+")";
    log= log+ " R"+unsignedLong2String(r1)+"="+signedShort2String(opResult)+"(0x"+
        short2HexString(opResult)+") ";

    if (opResult==0) {
        thread[threadNum].setStatusZ();
        log=log+"Z=1,";
    }
    else {
        thread[threadNum].clrStatusZ();
        log=log+"Z=0,";
    }
    if (opResult<0) {
        thread[threadNum].setStatusN();
        log=log+"N=1,";
    }
    else {
        thread[threadNum].clrStatusN();
        log=log+"N=0,";
    }
    if (((op1&(1<<op2)) >> op2)==1 ) {
        thread[threadNum].setStatusV();
        log=log+"V=1,";
    }
    else {
        thread[threadNum].clrStatusV();
        log=log+"V=0,";
    }
    if ((opResult>0) && (opResult < 256)) {
        thread[threadNum].setStatusC();
        log=log+"C=1";
    }
    else {
        thread[threadNum].clrStatusC();
        log=log+"C=0";
    }

    thread[threadNum].setR( opResult,r1);
//  cout << "bic r1,r2,k3" ;
    addLog (log,threadNum);

}

void xinc::iBis(int threadNum,short instruction)
{
    short r1,r2;
    short op1,op2,opResult;
    string log;
```

```
    r1=(instruction & 0x3800) >> 11;
    r2=(instruction & 0x7) ;

    op1=thread[threadNum].getR(r2);
    op2=(instruction & 0x78) >> 3;
    opResult=op1 | (1<<op2);

    log="bis R"+unsignedLong2String(r1)+",R"+unsignedLong2String(r2)+","+
        signedShort2String(op2)+"(0x"+short2HexString(op2)+")";

    log= log+ " R"+unsignedLong2String(r1)+"="+signedShort2String(opResult)
        +"(0x"+short2HexString(opResult)+") ";

    if (opResult==0) {
        thread[threadNum].setStatusZ();
        log=log+"Z=1,";
    }
    else {
        thread[threadNum].clrStatusZ();
        log=log+"Z=0,";
    }
    if (opResult<0) {
        thread[threadNum].setStatusN();
        log=log+"N=1,";
    }
    else {
        thread[threadNum].clrStatusN();
        log=log+"N=0,";
    }
    if (((op1&(1<<op2)) >> op2)==1 ) {
        thread[threadNum].setStatusV();
        log=log+"V=1,";
    }
    else {
        thread[threadNum].clrStatusV();
        log=log+"V=0,";
    }
    if ((opResult>0) && (opResult < 256)) {
        thread[threadNum].setStatusC();
        log=log+"C=1";
    }
    else {
        thread[threadNum].clrStatusC();
        log=log+"C=0";
    }

    thread[threadNum].setR( opResult,r1);
//  cout << "bis r1,r2,k3" ;
    addLog (log,threadNum);
}

void xinc::iBix(int threadNum,short instruction)
{
    short r1,r2;
    short op1,op2,opResult;
    string log;


    r1=(instruction & 0x3800) >> 11;
```

```
    r2=(instruction & 0x7) ;

    op1=thread[threadNum].getR(r2);
    op2=(instruction & 0x78) >> 3;
    opResult=op1 ^ (1<<op2);
    log="bix R"+unsignedLong2String(r1)+",R"+unsignedLong2String(r2)+
        ","+signedShort2String(op2)+"(0x"+short2HexString(op2)+")";

    log= log+ " R"+unsignedLong2String(r1)+"="+signedShort2String(opResult)
        +"(0x"+short2HexString(opResult)+") ";

    if (opResult==0) {
        thread[threadNum].setStatusZ();
        log=log+"Z=1,";
    }
    else {
        thread[threadNum].clrStatusZ();
        log=log+"Z=0,";
    }
    if (opResult<0) {
        thread[threadNum].setStatusN();
        log=log+"N=1,";
    }
    else {
        thread[threadNum].clrStatusN();
        log=log+"N=0,";
    }
    if (((op1&(1<<op2)) >> op2)==1 ) {
        thread[threadNum].setStatusV();
        log=log+"V=1,";
    }
    else {
        thread[threadNum].clrStatusV();
        log=log+"V=0,";
    }
    if ((opResult>0) && (opResult < 256)) {
        thread[threadNum].setStatusC();
        log=log+"C=1";
    }
    else {
        thread[threadNum].clrStatusC();
        log=log+"C=0";
    }

    thread[threadNum].setR( opResult,r1);
//  cout << "bix r1,r2,k3" ;
    addLog (log,threadNum);

}

void xinc::iInp(int threadNum,short instruction)
{

    unsigned short k2=instruction & 0x7f;
    short r=(instruction & 0x3800) >> 11;
    short opResult=peripheralRegisterRead[k2].getValue();
    string log;


    thread[threadNum].setR(opResult,r);
```

```
    log="inp R"+unsignedLong2String(r)+","+signedShort2String(k2)+
        "(0x"+short2HexString(k2)+")";

    log= log+ " R"+unsignedLong2String(r)+"="+signedShort2String(opResult)+
        "(0x"+short2HexString(opResult)+") ";

    if (opResult==0) {
        thread[threadNum].setStatusZ();
        log=log+"Z=1,";

    }
    else {
        thread[threadNum].clrStatusZ();
        log=log+"Z=0,";

    }
    if (opResult<0) {
        thread[threadNum].setStatusN();
        log=log+"N=1,";
    }
    else {
        thread[threadNum].clrStatusN();
        log=log+"N=0,";
    }
    if ((opResult&1)==1 ) {
        thread[threadNum].setStatusV();
        log=log+"V=1,";
    }
    else {
        thread[threadNum].clrStatusV();
        log=log+"V=0,";
    }
    if ((opResult>0) && (opResult < 256)) {
        thread[threadNum].setStatusC();
        log=log+"C=1";
    }
    else {
        thread[threadNum].clrStatusC();
        log=log+"C=0";
    }


//      cout << "inp r1,k2" ;
        addLog (log,threadNum);


}

void xinc::iIor(int threadNum,short instruction,short method)
{
    short r1,r2,r3;
    short op1,op2,opResult;
    string log;

    switch (method)
    {
        case 0:
```

```
        r1=(instruction & 0x3800) >> 11;
        r2=(instruction & 0x7) ;
        r3=(instruction & 0x38) >> 3;

        op1=thread[threadNum].getR(r2);
        op2=thread[threadNum].getR(r3);
    //    cout << "ior r1,r2,r3" ;
        log="ior R"+unsignedLong2String(r1)+",R"+unsignedLong2String(r2)+
            ",R"+unsignedLong2String(r3)+" ";
        break;

    case 1:
        if (getTwoWordsInstruction(threadNum)) {
            r1=(getFirstWord(threadNum) & 0x3800) >> 11;
            r2=(getFirstWord(threadNum) & 0x7) ;
            op1=thread[threadNum].getR(r2);
            op2=instruction;
//          cout << "ior r1,r2,k3 w2" ;
            log="ior R"+unsignedLong2String(r1)+",R"+unsignedLong2String(r2)+
                ","+signedShort2String(op2)+"(0x"+short2HexString(op2)+")";
        }
        else {
            setTwoWordsInstruction(true,threadNum);
            setFirstWord(instruction,threadNum);
//          cout << "ior r1,r2,k3 w1" ;
            return;
        }
        break;
    default:
        ;
}

opResult=op1 | op2;
log= log+ " R"+unsignedLong2String(r1)+"="+
    signedShort2String(opResult)+"(0x"+short2HexString(opResult)+") ";

if (opResult==0) {
    thread[threadNum].setStatusZ();
    log=log+"Z=1,";
}
else {
    thread[threadNum].clrStatusZ();
    log=log+"Z=0,";
}
if (opResult<0) {
    thread[threadNum].setStatusN();
    log=log+"N=1,";
}
else {
    thread[threadNum].clrStatusN();
    log=log+"N=0,";
}
if ((op1 & op2) == op2 ) {
    thread[threadNum].setStatusV();
    log=log+"V=1,";
}
else {
    thread[threadNum].clrStatusV();
    log=log+"V=0,";
}
```

```
    if ((opResult >0) && (opResult < 256)) {
        thread[threadNum].setStatusC();
        log=log+"C=1";
    }
    else {
        thread[threadNum].clrStatusC();
        log=log+"C=0";
    }
    thread[threadNum].setR( opResult,r1);
    addLog (log,threadNum);
}

void xinc::iJsr(int threadNum,short instruction,short method)
{
    short r1,r2;
    short op,op2;
    string log;

    switch (method)
    {
        case 0:

            r1=(instruction & 0x3800) >> 11;
            r2=(instruction & 0x7) ;
            op2=thread[threadNum].getR(r2);
    //      cout << "jsr r1,r2" ;
            log="jsr R"+unsignedLong2String(r1)+",R"+
                unsignedLong2String(r2)+" ";

            break;

        case 1:
            if (getTwoWordsInstruction(threadNum)) {
                r1=(getFirstWord(threadNum) & 0x3800) >> 11;
                op2=instruction;
    //          cout << "jsr r1,k2 w2" ;
                log="jsr R"+unsignedLong2String(r1)+",
                    signedShort2String(op2)+"(0x"+short2HexString(op2)+")";

            }
            else {
                setTwoWordsInstruction(true,threadNum);
                setFirstWord(instruction,threadNum);
    //          cout << "jsr r1,k2 w1" ;
                return;
            }
            break;
        default:
            ;
    }

    op=thread[threadNum].getPc();

    log= log+ " R"+unsignedLong2String(r1)+"=0x"+short2HexString((short)(op+1))+" ";
    log= log+ " PC=0x"+short2HexString((short)(op2))+" ";
    addLog (log,threadNum);

    thread[threadNum].setPc((short)(op2-1));
    thread[threadNum].setR((short)(op+1),r1);
```

```
}

void xinc::iLd(int threadNum,short instruction,short method)
{
    short r1,r2;
    short op1,op2,opResult;
    string log;

    switch (method)
    {
        case 0:

            r1=(instruction & 0x3800) >> 11;
            r2=(instruction & 0x700) >> 8;

            short displacement;
            displacement=signedExtension(instruction & 0xff,8);

            op1=thread[threadNum].getR(r2);
            op2=op1+displacement;

    //      cout << "ld r1,r2,k3" ;

            log="ld R"+unsignedLong2String(r1)+",R"+unsignedLong2String(r2)
                +","+signedShort2String(displacement)+"(0x"+
                short2HexString(displacement)+")";
            break;
        case 1:
            if (getTwoWordsInstruction(threadNum)) {
                r1=(getFirstWord(threadNum) & 0x3800) >> 11;
                r2=(getFirstWord(threadNum) & 0x7) ;
                op1=thread[threadNum].getR(r2);
                op2=op1+instruction;
    //          cout << "ld r1,r2,k3 w2" ;

                log="ld R"+unsignedLong2String(r1)+",R"+unsignedLong2String(r2)+
                    ","+signedShort2String(instruction)+
                    "(0x"+short2HexString(instruction)+")";
            }
            else {
                setTwoWordsInstruction(true,threadNum);
                setFirstWord(instruction,threadNum);
    //          cout << "ld r1,r2,k3 w1" ;
                return;
            }


            break;
        case 2:
            if (getTwoWordsInstruction(threadNum)) {
                r1=(getFirstWord(threadNum) & 0x3800) >> 11;

                op2=instruction;
    //          cout << "ld r1,k2 w2" ;

                log="ld R"+unsignedLong2String(r1)+","+signedShort2String(op2)
                    +"(0x"+short2HexString(op2)+")";
            }
```

```
            else {
                setTwoWordsInstruction(true,threadNum);
                setFirstWord(instruction,threadNum);
        //      cout << "ld r1,k2 w1" ;
                return;
            }
            break;
        default:
            ;
    }

    opResult=ram[(short)(op2-49152)].getValue();

    log= log+ " R"+unsignedLong2String(r1)+"="+signedShort2String(opResult)
        +"(0x"+short2HexString(opResult)+") ";

    if (opResult==0) {
        thread[threadNum].setStatusZ();
        log=log+"Z=1,";
    }
    else {
        thread[threadNum].clrStatusZ();
        log=log+"Z=0,";
    }
    if (opResult<0) {
        thread[threadNum].setStatusN();
        log=log+"N=1,";
    }
    else {
        thread[threadNum].clrStatusN();
        log=log+"N=0,";
    }
    if ((opResult&1)==1 ) {
        thread[threadNum].setStatusV();
        log=log+"V=1,";
    }
    else {
        thread[threadNum].clrStatusV();
        log=log+"V=0,";
    }
    if ((opResult>0) && (opResult < 256)) {
        thread[threadNum].setStatusC();
        log=log+"C=1";
    }
    else {
        thread[threadNum].clrStatusC();
        log=log+"C=0";
    }

    thread[threadNum].setR(opResult,r1);
    addLog (log,threadNum);


}

void xinc::iRol(int threadNum,short instruction,short method)
{
    short r1,r2,r3;
    short op1,op2,opResult;
    string log;
```

```
switch (method)
{
    case 0:

        r1=(instruction & 0x3800) >> 11;
        r2=(instruction & 0x7) ;
        r3=(instruction & 0x38) >> 3;

        op1=thread[threadNum].getR(r2);
        op2=thread[threadNum].getR(r3) % 16;

//      cout << "rol r1,r2,r3 " ;
        log="rol R"+unsignedLong2String(r1)+",R"+
            unsignedLong2String(r2)+",R"+unsignedLong2String(r3)+" ";

        break;
    case 1:
        r1=(instruction & 0x3800) >> 11;
        r2=(instruction & 0x7);
        op1=thread[threadNum].getR(r2);
        op2= (instruction & 0x78) >> 3;
//        cout << "rol r1,r2,k3 " ;
        log="rol R"+unsignedLong2String(r1)+",R"+
            unsignedLong2String(r2)+","+signedShort2String(op2)
            +"(0x"+short2HexString(op2)+")";



        break;

    default:
        ;
}

unsigned short mask1;
if (op2>0) {
    mask1=(1<<op2)-1;
    opResult=((mask1<<(16-op2) & op1)>> (16-op2)) |
        ((((mask1<<(16-op2)) ^ 0xffff)&op1) <<op2);


}
else if (op2<0) {
    mask1=(1<<abs(op2))-1;
    opResult=(((mask1^0xffff)&op1)>>(abs(op2))) |
        ((mask1 & op1) << (16-abs(op2)));
}
else {
    opResult=op1;
}

log= log+ " R"+unsignedLong2String(r1)+"="+signedShort2String(opResult)+
    "(0x"+short2HexString(opResult)+") ";

if (opResult==0) {
    thread[threadNum].setStatusZ();
    log=log+"Z=1,";
}
else {
```

```
            thread[threadNum].clrStatusZ();
            log=log+"Z=0,";
        }
        if (opResult<0) {
            thread[threadNum].setStatusN();
            log=log+"N=1,";
        }
        else {
            thread[threadNum].clrStatusN();
            log=log+"N=0,";
        }
        if ((opResult&1)==1 ) {
            thread[threadNum].setStatusV();
            log=log+"V=1,";
        }
        else {
            thread[threadNum].clrStatusV();
            log=log+"V=0,";
        }
        if ((opResult>0) && (opResult < 256)) {
            thread[threadNum].setStatusC();
            log=log+"C=1";
        }
        else {
            thread[threadNum].clrStatusC();
            log=log+"C=0";
        }
        thread[threadNum].setR( opResult,r1);
        addLog (log,threadNum);

}

void xinc::iSt(int threadNum,short instruction,short method)
{
    short r1,r2;
    short op1,op2,opResult;
    string log;

    switch (method)
    {
        case 0:

            r1=(instruction & 0x3800) >> 11;
            r2=(instruction & 0x700) >> 8;

            short displacement;
            displacement=signedExtension(instruction & 0xff,8);

            op1=thread[threadNum].getR(r2);
            op2=op1+displacement;

    //      cout << "st r1,r2,k3" ;

            log="st R"+unsignedLong2String(r1)+",R"+unsignedLong2String(r2)+
                ","+signedShort2String(displacement)+
                "(0x"+short2HexString(displacement)+")";

            break;
        case 1:
            if (getTwoWordsInstruction(threadNum)) {
```

```
                r1=(getFirstWord(threadNum) & 0x3800) >> 11;
                r2=(getFirstWord(threadNum) & 0x7) ;
                op1=thread[threadNum].getR(r2);
                op2=op1+instruction;
//              cout << "st r1,r2,k3 w2" ;
                log="st R"+unsignedLong2String(r1)+",R"+unsignedLong2String(r2)+
                    ","+signedShort2String(instruction)+"(0x"+
                    short2HexString(instruction)+")";

            }
            else {
                setTwoWordsInstruction(true,threadNum);
                setFirstWord(instruction,threadNum);
//              cout << "st r1,r2,k3 w1" ;
                return;
            }


            break;
        case 2:
            if (getTwoWordsInstruction(threadNum)) {
                r1=(getFirstWord(threadNum) & 0x3800) >> 11;

                op2=instruction;
//              cout << "st r1,k2 w2" ;
                log="st R"+unsignedLong2String(r1)+","+
                    signedShort2String(op2)+"(0x"+short2HexString(op2)+")";

            }
            else {
                setTwoWordsInstruction(true,threadNum);
                setFirstWord(instruction,threadNum);
//              cout << "st r1,k2 w1" ;
                return;
            }
            break;
        default:
            ;
    }

    opResult=thread[threadNum].getR(r1);
    ram[(short)(op2-49152)].setValue(opResult);
    log=log+"RAM[0x"+short2HexString(op2)+"]="+
        signedShort2String(opResult)+
        "(0x"+short2HexString(opResult)+")";
    addLog (log,threadNum);

}

void xinc::iSub(int threadNum,short instruction)
{
    short r1,r2,r3;
    short sub1,sub2,subResult;
    int subResult1;
    string log;

    r1=(instruction & 0x3800) >> 11;
    r2=(instruction & 0x7) ;
    r3=(instruction & 0x38) >> 3;
```

```
    sub1=thread[threadNum].getR(r2);
    sub2=thread[threadNum].getR(r3);
    subResult=sub1-sub2;
    subResult1=sub1-sub2;

    log="sub R"+unsignedLong2String(r1)+",R"+unsignedLong2String(r2)
        +",R"+unsignedLong2String(r3)+" ";
    log= log+ " R"+unsignedLong2String(r1)+"="+signedShort2String(subResult)
        +"(0x"+short2HexString(subResult)+") ";

    if (subResult==0) {
        thread[threadNum].setStatusZ();
        log=log+"Z=1,";
    }
    else {
        thread[threadNum].clrStatusZ();
        log=log+"Z=0,";
    }
    if (subResult<0) {
        thread[threadNum].setStatusN();
        log=log+"N=1,";
    }
    else {
        thread[threadNum].clrStatusN();
        log=log+"N=0,";
    }
    if ((sub1>0 && sub2<0 && subResult<0) ||
        (sub1<0 && sub2>0 && subResult>0)) {
        thread[threadNum].setStatusV();
        log=log+"V=1,";
    }
    else {
        thread[threadNum].clrStatusV();
        log=log+"V=0,";
    }

//  if ((subResult1>32767) || (subResult1<-32768)) {
    if (sub1<sub2) {
        thread[threadNum].setStatusC();
        log=log+"C=1";
    }
    else {
        thread[threadNum].clrStatusC();
        log=log+"C=0";
    }

    thread[threadNum].setR( subResult,r1);

//    cout << "sub r1,r2,r3" ;
    addLog (log,threadNum);


}

void xinc::iThrd(int threadNum,short instruction)
{
    string log;
    short r1=(instruction & 0x3800) >> 11;
    thread[threadNum].setR(threadNum,r1);
    log="thrd R"+unsignedLong2String(r1);
```

```
    log= log+ " R"+unsignedLong2String(r1)+"="+
        signedShort2String(threadNum)+
        "(0x"+short2HexString(threadNum)+") ";
    addLog (log,threadNum);
//  cout << "thrd r1" ;
}

void xinc::iXor(int threadNum,short instruction,short method)
{
    short r1,r2,r3;
    short op1,op2,opResult;
    string log;

    switch (method)
    {
        case 0:

            r1=(instruction & 0x3800) >> 11;
            r2=(instruction & 0x7) ;
            r3=(instruction & 0x38) >> 3;

            op1=thread[threadNum].getR(r2);
            op2=thread[threadNum].getR(r3);
    //        cout << "xor r1,r2,r3" ;
            log="xor R"+unsignedLong2String(r1)+",R"+
                unsignedLong2String(r2)+",R"+unsignedLong2String(r3)+" ";

            break;

        case 1:
            if (getTwoWordsInstruction(threadNum)) {
                r1=(getFirstWord(threadNum) & 0x3800) >> 11;
                r2=(getFirstWord(threadNum) & 0x7) ;
                op1=thread[threadNum].getR(r2);
                op2=instruction;
//              cout << "xor r1,r2,k3 w2" ;
                log="xor R"+unsignedLong2String(r1)+",R"+
                    unsignedLong2String(r2)+","+signedShort2String(op2)
                    +"(0x"+short2HexString(op2)+")";

            }
            else {
                setTwoWordsInstruction(true,threadNum);
                setFirstWord(instruction,threadNum);
    //          cout << "xor r1,r2,k3 w1" ;
                return;
            }
            break;
        default:
            ;
    }

    opResult=op1 ^ op2;
    log= log+ " R"+unsignedLong2String(r1)+"="+
        signedShort2String(opResult)+"(0x"+short2HexString(opResult)+") ";

    if (opResult==0) {
        thread[threadNum].setStatusZ();
        log=log+"Z=1,";
```

```
    }
    else {
        thread[threadNum].clrStatusZ();
        log=log+"Z=0,";
    }
    if (opResult<0) {
        thread[threadNum].setStatusN();
        log=log+"N=1,";
    }
    else {
        thread[threadNum].clrStatusN();
        log=log+"N=0,";
    }
    if ((op1 & op2) == op2 ) {
        thread[threadNum].setStatusV();
        log=log+"V=1,";
    }
    else {
        thread[threadNum].clrStatusV();
        log=log+"V=0,";
    }
    if ((opResult>0) && (opResult < 256)) {
        thread[threadNum].setStatusC();
        log=log+"C=1";
    }
    else {
        thread[threadNum].clrStatusC();
        log=log+"C=0";
    }
    thread[threadNum].setR( opResult,r1);
    addLog (log,threadNum);

}

void xinc::runThread()
{
    unsigned long cycle=getSystemClockCycle();

    int threadNum=cycle%8;

    if (threadNum == 0) {
        newestILog2=newestILog2+unsignedLong2String(cycle);
    }

    if (isThreadRun(cycle%8)) {
//  cout << "System Clock Cycle " << cycle << ":" ;
//  cout << "Thread " << threadNum << " is running." ;

    // if two word instruction, execute at second run
    short instruction;

    if (thread[threadNum].getPc()<16384) {
        instruction=rom[thread[threadNum].getPc()].getValue();
    }
    else if (thread[threadNum].getPc()>=0xc000)  {
        instruction=ram[thread[threadNum].getPc()-0xc000].getValue();
    }
    else {
        cout << "Error: PC out of RAM and ROM range. PC="
            << thread[threadNum].getPc()-0xc000 << endl;
```

```
}


if (getTwoWordsInstruction(threadNum)) {
    if ((getFirstWord(threadNum) & 0xc7ff) == 0x03e8) { // mov R1,K2 W2
        iMov(threadNum, instruction);
    }
    else if ((getFirstWord(threadNum) & 0xff00) == 0x0100) { // bra K1 W2
        iBra(threadNum, instruction);
    }
    else if ((getFirstWord(threadNum) & 0xc7f8) == 0x3c0) { // add
        iAdd(threadNum, instruction, 2);
    }
    else if ((getFirstWord(threadNum) & 0xc7f8) == 0x3c8) { // and r1,r2,k3
        iAnd(threadNum, instruction, 1);
    }
    else if ((getFirstWord(threadNum) & 0xc7f8) == 0x3d0) { // ior r1,r2,k3
        iIor(threadNum, instruction, 1);
    }
    else if ((getFirstWord(threadNum) & 0xc7f8) == 0x3d8) { // xor r1,r2,k3
        iXor(threadNum, instruction, 1);
    }
    else if ((getFirstWord(threadNum) & 0xc300) == 0x0) { // bc c1,k2
        iBc(threadNum, instruction, 1);
    }
    else if ((getFirstWord(threadNum) & 0xc3fc) == 0x3ec) { // jsr r1,k2
        iJsr(threadNum, instruction, 1);
    }
    else if ((getFirstWord(threadNum) & 0xc7f8) == 0x3f0) { // ld r1,r2,k3
        iLd(threadNum, instruction, 1);
    }
    else if ((getFirstWord(threadNum) & 0xc7ff) == 0x3ea) { // ld r1,k2
        iLd(threadNum, instruction, 2);
    }
    else if ((getFirstWord(threadNum) & 0xc7f8) == 0x3f8) { // st r1,r2,k3
        iSt(threadNum, instruction, 1);
    }
    else if ((getFirstWord(threadNum) & 0xc7ff) == 0x3eb) { // st r1,k2
        iSt(threadNum, instruction, 2);
    }
    else if ((getFirstWord(threadNum) & 0xff1f) == 0x3901) { // movZOLR r1,k3
        iMovZOLR(threadNum, instruction);
    }
    else if ((getFirstWord(threadNum) & 0xff1f) == 0x3902) { // setZOLR r1,k3
        iSetZOLA(threadNum, instruction);
    }
    else {
    }
    setTwoWordsInstruction(false, threadNum);
}
else if ((instruction & 0xc7c0) == 0x0380) { // mov R1,K2 W1

    iMov(threadNum, instruction);
    instructionStat[20]++;
    newestILog2=newestILog2+"|20";

}
else if ((instruction & 0xc7ff) == 0x03e8) { // mov R1,K2 W2
    iMov(threadNum, instruction);
    instructionStat[21]++;
```

```
    newestILog2=newestILog2+"|21";

}
else if ((instruction & 0xc780) == 0x0280) { // outp r1,k2
    iOutp(threadNum,instruction);
    instructionStat[22]++;
    newestILog2=newestILog2+"|22";
}
else if ((instruction & 0xff00) == 0x0100) { // bra K1 & bra K1 W2
    iBra(threadNum,instruction);


}
else if ((instruction & 0xc7c0) == 0x0300) { // add R1,R2,R3
    iAdd(threadNum,instruction,0);
    instructionStat[0]++;
    newestILog2=newestILog2+"|0";

}
else if ((instruction & 0xc000) == 0x4000) { // add R1,R2,K3 W1
    iAdd(threadNum,instruction,1);
    instructionStat[1]++;
    newestILog2=newestILog2+"|1";
}
else if ((instruction & 0xc7f8) == 0x3c0) { // add R1,R2,K3 W2
    iAdd(threadNum,instruction,2);
    instructionStat[2]++;
    newestILog2=newestILog2+"|2";
}
else if ((instruction & 0xc7c0) == 0x340) { // sub r1,r2,r3
    iSub(threadNum,instruction);
    instructionStat[28]++;
    newestILog2=newestILog2+"|28";
}
else if ((instruction & 0xc7c0) == 0x540) { // and r1,r2,r3
    iAnd(threadNum,instruction,0);
    instructionStat[3]++;
    newestILog2=newestILog2+"|3";
}
else if ((instruction & 0xc7f8) == 0x3c8) { // and r1,r2,K3
    iAnd(threadNum,instruction,1);
    instructionStat[4]++;
    newestILog2=newestILog2+"|4";
}
else if ((instruction & 0xc7c0) == 0x580) { // ior r1,r2,r3
    iIor(threadNum,instruction,0);
    instructionStat[13]++;
    newestILog2=newestILog2+"|13";
}
else if ((instruction & 0xc7f8) == 0x3d0) { // ior r1,r2,K3
    iIor(threadNum,instruction,1);
    instructionStat[14]++;
    newestILog2=newestILog2+"|14";
}
else if ((instruction & 0xc7c0) == 0x5c0) { // xor r1,r2,r3
    iXor(threadNum,instruction,0);
    instructionStat[30]++;
    newestILog2=newestILog2+"|30";
}
else if ((instruction & 0xc7f8) == 0x3d8) { // xor r1,r2,K3
```

```
    iXor(threadNum,instruction,1);
    instructionStat[31]++;
    newestILog2=newestILog2+"|31";
}
else if ((instruction & 0xc7c0) == 0x500) { // rol r1,r2,r3
    iRol(threadNum,instruction,0);
    instructionStat[23]++;
    newestILog2=newestILog2+"|23";
}
else if ((instruction & 0xc780) == 0x600) { // rol r1,r2,k3
    iRol(threadNum,instruction,1);
    instructionStat[24]++;
    newestILog2=newestILog2+"|24";
}
else if ((instruction & 0xc780) == 0x200) { // inp r1,k2
    iInp(threadNum,instruction);
    instructionStat[12]++;
    newestILog2=newestILog2+"|12";
}
else if ((instruction & 0xc300) == 0x0) { // bc c1,k2
    if ((instruction & 0xff)==0x0) {
        iBc(threadNum,instruction,1); //bc c1,k2 w2
        instructionStat[6]++;
        newestILog2=newestILog2+"|6";
    }
    else {
        iBc(threadNum,instruction,0); //bc c1,k2 w1
        instructionStat[5]++;
        newestILog2=newestILog2+"|5";
    }
}
else if ((instruction & 0xc780) == 0x680) { // bic r1,r2,k3
    iBic(threadNum,instruction);
    instructionStat[7]++;
    newestILog2=newestILog2+"|7";
}
else if ((instruction & 0xc780) == 0x700) { // bis r1,r2,k3
    iBis(threadNum,instruction);
    instructionStat[8]++;
    newestILog2=newestILog2+"|8";
}
else if ((instruction & 0xc780) == 0x780) { // bix r1,r2,k3
    iBix(threadNum,instruction);
    instructionStat[9]++;
    newestILog2=newestILog2+"|9";
}
else if ((instruction & 0xc7f8) == 0x3e0) { // jsr r1,r2
    iJsr(threadNum,instruction,0);
    instructionStat[15]++;
    newestILog2=newestILog2+"|15";
}
else if ((instruction & 0xc7fc) == 0x3ec) { // jsr r1,k2
    iJsr(threadNum,instruction,1);
    instructionStat[16]++;
    newestILog2=newestILog2+"|16";
}
else if ((instruction & 0xc000) == 0x8000) { // ld r1,r2,k3 w1
    iLd(threadNum,instruction,0);
    instructionStat[17]++;
    newestILog2=newestILog2+"|17";
```

```
    }
    else if ((instruction & 0xc7f8) == 0x3f0) { // ld r1,r2,k3 w2
        iLd(threadNum,instruction,1);
        instructionStat[18]++;
        newestILog2=newestILog2+"|18";
    }
    else if ((instruction & 0xc7ff) == 0x3ea) { // ld r1,k2
        iLd(threadNum,instruction,2);
        instructionStat[19]++;
        newestILog2=newestILog2+"|19";
    }
    else if ((instruction & 0xc000) == 0xc000) { // st r1,r2,k3 w1
        iSt(threadNum,instruction,0);
        instructionStat[25]++;
        newestILog2=newestILog2+"|25";
    }
    else if ((instruction & 0xc7f8) == 0x3f8) { // st r1,r2,k3 w2
        iSt(threadNum,instruction,1);
        instructionStat[26]++;
        newestILog2=newestILog2+"|26";
    }
    else if ((instruction & 0xc7ff) == 0x3eb) { // st r1,k2
        iSt(threadNum,instruction,2);
        instructionStat[27]++;
        newestILog2=newestILog2+"|27";
    }
    else if ((instruction & 0xc7ff) == 0x3e9) { // thrd r1
        iThrd(threadNum,instruction);
        instructionStat[29]++;
        newestILog2=newestILog2+"|29";
    }
    else if ((instruction & 0xff1f) == 0x3901) { // movZOLR r1,k3
        iMovZOLR(threadNum,instruction);
        instructionStat[32]++;
        newestILog2=newestILog2+"|32";
    }
    else if ((instruction & 0xff1f) == 0x3902) { // setZOLA r1,k3
        iSetZOLA(threadNum,instruction);
        instructionStat[33]++;
        newestILog2=newestILog2+"|33";
    }
    else {
        cout.setf(ios::hex);
        cout << "The instruction cannot be decoded, address: 0x"
            << short2HexString(thread[threadNum].getPc()) <<
            " instruction: 0x" << short2HexString(instruction)
            << endl; // decode error
    }


    thread[threadNum].setPc(thread[threadNum].getPc()+1);
//  cout << endl;
    }
    else {
    //  cout << "Thread " << threadNum << " is not running.";
    //  cout << endl;
        newestILog2=newestILog2+"|-";
    }

    if (threadNum == 7) {
```

```
        newestILog2=newestILog2+"\n";
    }
    zolProcess(threadNum);
    setSystemClockCycle(getSystemClockCycle()+1);
    if (cycle%8 ==0)  {
        ioProcess();

    }

};


void xinc::ioProcess()
{
    if (peripheralRegisterWrite[1].getIsOutputSet()) {
        unsigned short pntr=peripheralRegisterWrite[3].getValue();
        unsigned short threadNum=(pntr & 0x38) >> 3;
        thread[threadNum].setPc(peripheralRegisterWrite[1].getValue());
        peripheralRegisterWrite[1].setValue(0);
        peripheralRegisterWrite[1].setIsOutputSet(false);
        peripheralRegisterWrite[3].setIsOutputSet(false);

    }

    if (peripheralRegisterWrite[0].getIsOutputSet()) {
        unsigned short pntr=peripheralRegisterWrite[3].getValue();
        unsigned short threadNum=(pntr & 0x38) >> 3;
        unsigned short registerNo=(pntr & 0x7) ;
        thread[threadNum].setR(peripheralRegisterWrite[0].getValue(),registerNo);
        peripheralRegisterWrite[0].setValue(0);
        peripheralRegisterWrite[0].setIsOutputSet(false);
        peripheralRegisterWrite[3].setIsOutputSet(false);

    }

    peripheralRegisterRead[4].setValue(peripheralRegisterRead[4].getValue()
        & ~peripheralRegisterWrite[4].getValue());
        // if thread is stop , clear SCUBkpt


}

void xinc::runSystemClockCycles(unsigned long length)
{


    unsigned long j=0;
    while ((!_kbhit() && (length==0)) || ((length!=0) && (j<length))) {
        cout.unsetf(ios::hex);
     // cout << "System Clock Cycle " << j << endl;

    // if thread 0 run, run thread 0
        runThread();
        j++;

    // if thread 7 run, run thread 7


    // process i/o
    // process scupc write, scucc write, scu register read, scu register write
```

```
    // second run to process scupc read, scucc read


//      printRegister();
//      printIo(0,4);

    }

}

void xinc::printRegister()
{
    cout.setf(ios::hex|ios::right);
    cout << "  PC      R0      R1      R2      R3      R4      R5
        R6      R7      N Z V C" << endl;
    for (int i=0;i<8;i++) {

        cout << i << " ";

        cout <<"0x" ;
        cout.width(4);
        cout.fill('0');
        cout << short2HexString(thread[i].getPc()) << " ";
        for (int j=0;j <8; j++) {

            cout << "0x" ;
            cout.width(4);
            cout.fill('0');
            cout <<  short2HexString(thread[i].getR(j)) << " ";
        }
        cout.width(1);
        cout <<  ( (thread[i].getStatus() & 8) >> 3 ) << " ";
        cout.width(1);
        cout <<  ( (thread[i].getStatus() & 4) >> 2 ) << " ";
        cout.width(1);
        cout <<  ( (thread[i].getStatus() & 2) >> 1 ) << " ";
        cout.width(1);
        cout << ( (thread[i].getStatus() & 1)  ) << " ";

        cout << endl ;
    }
   cout << endl;
   cout << "  ZOLR0  ZOLR1  ZOLR2  ZOLR3  ZOLR4  ZOLR5  ZOLR6  ZOLR7" << endl;
   for (i=0;i<8;i++) {
        cout << i << " ";
        for (int j=0;j <8; j++) {

            cout << "0x" ;
            cout.width(4);
            cout.fill('0');
            cout <<  short2HexString(thread[i].getZOLR(j)) << " ";
        }
        cout << endl ;
   }
   cout << endl;
   cout << "  ZOLAS0 ZOLAS1 ZOLAS2 ZOLAS3 ZOLAS4 ZOLAS5 ZOLAS6 ZOLAS7" << endl;
   for (i=0;i<8;i++) {
        cout << i << " ";
        for (int j=0;j <8; j++) {
```

```
            cout << "0x" ;
            cout.width(4);
            cout.fill('0');
            cout <<  short2HexString(thread[i].getZOLAS(j)) << " ";
        }
        cout << endl ;
    }
    cout << endl;
    cout << "  ZOLAE0 ZOLAE1 ZOLAE2 ZOLAE3 ZOLAE4 ZOLAE5 ZOLAE6 ZOLAE7" << endl;
    for (i=0;i<8;i++) {
        cout << i << " ";
        for (int j=0;j <8; j++) {

            cout << "0x" ;
            cout.width(4);
            cout.fill('0');
            cout <<  short2HexString(thread[i].getZOLAE(j)) << " ";
        }
        cout << endl ;
    }
}

void xinc::runTo(short address,char* logfilename)
{
    int i=0;
    while (!_kbhit()) {

        runSystemClockCycles(1);

        if ((short)thread[systemClockCycle%8].getPc()==address) return;

        // debug purpose begin
        //check nodes
//      if ((unsigned short)thread[systemClockCycle%8].getPc()==0xc39a) {
//          for (int j=0;j<6;j++) {
//              for (int k=0;k<6;k++) {
//                  cout << ram[0xf18c+j*6+k-0xc000].getValue() << " ";
//              }
//              cout << endl;
//              for ( k=0;k<6;k++) {
//                  cout << ram[0xf168+j*6+k-0xc000].getValue() << " ";
//
//              }
//              cout << endl;
//              for ( k=0;k<6;k++) {
//                  cout << ram[0xf144+j*6+k-0xc000].getValue() << " ";
//
//              }
//              cout << endl;
//          }
//      }

    //  if ((unsigned short)thread[systemClockCycle%8].getPc()==0xc382) {
    //      cout<<  short2HexString(thread[1].getR(6)) << " ";
    //  }
        // debug purpose end

        if (fp_serialIn.is_open()) {
            if (fp_serialIn.eof()) {
                cout << "Read file end!" << endl;
```

```
                return;
            }
        }
        if (i==1000) {
            if (!saveNewestLog(logfilename)) {
                cout << "Log file wrong!";
            }
            i=0;
        };
        i++;

    }
    if (!saveNewestLog(logfilename)) {
        cout << "Log file wrong!";
    };
}

void xinc::reset()
{
    // Initialization
    int i;
    for (i=0;i<8;i++) {
        thread[i].setPc(0);
        for (int j=0;j<8;j++) thread[i].setR(0,j);
        thread[i].setStatus(0);
    }

    for (i=0;i<16384;i++) {
        ram[i].setValue(0);
        rom[i].setValue(0);
    }

    for (i=0;i<128;i++) {
        peripheralRegisterRead[i].setValue(0);
        peripheralRegisterRead[i].setIsOutputSet(false);
        peripheralRegisterWrite[i].setValue(0);
        peripheralRegisterWrite[i].setIsOutputSet(false);


    }

    systemClockCycle=0;

    // set eeprom subprogram
    rom[10].setValue((short)0x13e8);
    rom[11].setValue((short)0xc003);
    rom[12].setValue((short)0x0be2);

    // set thread 0 pc

    //  setPc(0xc000,0);
    thread[0].setPc(0xc000);
    // set thread 0 run
    peripheralRegisterWrite[4].setValue(254); // 11111110

    for (i=0;i<65536;i++) {
        addressStat[i]=0;
    }

    for (i=0;i<32;i++) instructionStat[i]=0;
```

```
}

void xinc::setSystemClockCycle(unsigned long cycleNumber)
{
    systemClockCycle=cycleNumber;
}

unsigned long xinc::getSystemClockCycle()
{
    return systemClockCycle;
}

void xinc::printRam(unsigned short start,unsigned short end)
{
    cout.setf(ios::hex|ios::right);
    unsigned short address;
    address = start - start % 8;
    while (address <= end) {
        cout << "0x" ;
        cout.width(4);
        cout << short2HexString(address) << " - " ;
        for (int i=0;i<8;i++) {
            if ((address+i)>=start && (address+i)<=end) {
                cout << "0x" ;
                cout.width(4);
                cout << short2HexString(ram[address+i-0xc000].getValue()) << " " ;
            }
            else {
                cout << "        ";
            }

        }
        cout << endl;

        address=address+8;
    }



}

void xinc::printIO(unsigned short start,unsigned short end)
{
    int i;
    cout.setf(ios::hex|ios::right);
    unsigned short address;
    address = start - start % 8;
    while (address <= end) {
        cout << "W 0x" ;
        cout.width(4);
        cout.fill('0');
        cout << short2HexString(address) << " - " ;
        for (i=0;i<8;i++) {
            if ((address+i)>=start && (address+i)<=end) {
                cout << "0x" ;
                cout.width(4);
                cout << short2HexString(peripheralRegisterWrite[address+i].
                    getValue()) << " ";
```

```
            }
            else {
                cout << "        ";
            }

        }
        cout << endl;
        cout << "R 0x" ;
        cout.width(4);
        cout.fill('0');
        cout << short2HexString(address) << " - " ;
        for (i=0;i<8;i++) {
            if ((address+i)>=start && (address+i)<=end) {
                cout << "0x" ;
                cout.width(4);
                cout << short2HexString(peripheralRegisterRead[address+i].
                    getValue()) << " ";
            }
            else {
                cout << "        ";
            }

        }
        cout << endl;
        address=address+8;
    }


}

bool xinc::saveNewestLog(char* logfilename)
{
    ofstream fp_out;
    bool wrong=true;
    if (logfilename[0]==0) {
        newestLog="";

    }
    else {

        fp_out.open(logfilename,ios::out | ios::app  );

        if (fp_out.fail())
        {
            wrong=false;
        }

        fp_out << newestLog;
        fp_out.close();
        newestLog="";
    }
    if (instructionLog2) {
        char logFilename2[100];
        logFilename2[0]=0;
        strcat(logFilename2,filenameRoot);
        strcat(logFilename2,".il2");
        fp_out.open(logFilename2,ios::out | ios::app  );
        if (fp_out.fail())
        {
            cout << "Instruction Log 2 file open fail!";
```

```
        }
        else {
            fp_out << newestILog2;
            fp_out.close();
        }

    }
    newestILog2="";



    return wrong;
}

void xinc::addLog(string log,unsigned short threadNum)
{
    string currentLog;
    unsigned short currentAddress;

    if (getTwoWordsInstruction(threadNum)) {
        currentAddress=thread[threadNum].getPc()-1;
    }
    else {
        currentAddress=thread[threadNum].getPc();
    }

    addressStat[currentAddress]++;


    if (screenLog || fileLog) {
        currentLog=unsignedLong2String(getSystemClockCycle())+".T"+
            unsignedLong2String(threadNum)+".A0x";

        currentLog=currentLog+short2HexString(currentAddress);

        currentLog=currentLog+"."+log+"\n";
        if (screenLog) {
            cout << currentLog;
        }
        if (fileLog) {
            newestLog=newestLog+currentLog;
        }
    }
}

bool xinc::setSerialInFile(char* filename)
{


    if (filename[0]==0) {
        serialInOpen=false;
        return false;
    }
    if (fp_serialIn.is_open()) {
        fp_serialIn.close();
        serialInOpen=false;
    }
    fp_serialIn.open(filename,ios::in | ios::binary );
```

```
    if (fp_serialIn.fail())
    {
        serialInOpen=false;
        return false;
    }
    serialInOpen=true;
    return true;
}

bool xinc::setSerialOutFile(char* filename)
{


    if (filename[0]==0) {
        serialOutOpen=false;
        return false;
    }
    if (fp_serialOut.is_open()) {
        fp_serialOut.close();
        serialOutOpen=false;
    }
    fp_serialOut.open(filename,ios::out | ios::binary );

    if (fp_serialOut.fail())
    {
        serialOutOpen=false;
        return false;
    }
    serialOutOpen=true;
    return true;
}

void xinc::closeSerialFile()
{

    if (fp_serialIn.is_open()) {
        fp_serialIn.close();
    }
    if (fp_serialOut.is_open()) {
        fp_serialOut.close();
    }



}

void xinc::setScreenLog(bool val)
{
    screenLog=val;
}

void xinc::setFileLog(bool val)
{
    fileLog=val;
}

void xinc::setInstructionLog2(bool val)
{
    instructionLog2=val;
}
```

```
void xinc::printStat(unsigned short start, unsigned short end, short sortMethod)
{
    unsigned long a[65536][2];
    for (unsigned int i=0;i<65536; i++) {
        a[i][0]=i;
        a[i][1]=addressStat[i];
    }
/*  for (i=start;i<=end-1;i++) {
        int max=i;
        long maxvalue=a[i][1];
        for (unsigned int j=i+1;j<=end;j++) {
            if (maxvalue < a[j][1]) {
                max=j;
                maxvalue=a[j][1];
            }
            else if ((maxvalue== a[j][1]) && (a[i][0]> a[j][0])) {
                max=j;
            }
        }
        long temp=a[max][1];
        a[max][1]=a[i][1];
        a[i][1]=temp;
        temp=a[max][0];
        a[max][0]=a[i][0];
        a[i][0]=temp;
    }
*/
    for (i=start; i<=end; i++) {
        if (a[i][1] != 0) {
            cout << "0x" << short2HexString(a[i][0]) << ":" << a[i][1] << endl;
        }
    }
}

void xinc::resetStat()
{
    for (unsigned  int i=0; i<=65535; i++) {
        addressStat[i]=0;
    }
}

void xinc::printInstructionStat()
{
    unsigned long a[INSTRUCTION_NUMBER][2];
    for (short i=0;i<INSTRUCTION_NUMBER; i++) {
        a[i][0]=i;
        a[i][1]=instructionStat[i];
    }
    for (i=0;i<INSTRUCTION_NUMBER-1;i++)
        for (short j=i+1;j<INSTRUCTION_NUMBER;j++) {
            if (a[i][1]< a[j][1]) {
                long temp=a[j][1];
                a[j][1]=a[i][1];
                a[i][1]=temp;
                temp=a[j][0];
                a[j][0]=a[i][0];
                a[i][0]=temp;
            }
```

```
        }


    for (i=0; i<INSTRUCTION_NUMBER; i++) {
        cout << instructionName[a[i][0]] << ":" << a[i][1] << endl;
    }
}
void xinc::resetInstructionStat()
{

    for (unsigned  short i=0; i<INSTRUCTION_NUMBER; i++) {
        instructionStat[i]=0;
    }
}

void xinc::iMovZOLR(int threadNum,short instruction)
{
    short r1;
    string log;

    if (getTwoWordsInstruction(threadNum)) {
        r1=(getFirstWord(threadNum) & 0xe0) >> 5;



        thread[threadNum].setZOLR(instruction,r1);

        log="iMovZOLR R"+unsignedLong2String(r1)+","+
            signedShort2String(instruction)+"(0x"+
            short2HexString(instruction)+")";
        addLog (log,threadNum);
    }
    else {
        setTwoWordsInstruction(true,threadNum);
        setFirstWord(instruction,threadNum);
        return;
    }


}

void xinc::iSetZOLA(int threadNum,short instruction)
{
    short r1;
    unsigned short pc;
    string log;

    if (getTwoWordsInstruction(threadNum)) {
        r1=(getFirstWord(threadNum) & 0xe0) >> 5;



        thread[threadNum].setZOLAE(instruction,r1);
        pc=thread[threadNum].getPc()+1;
        thread[threadNum].setZOLAS(pc,r1);

        log="isetZOLA R"+unsignedLong2String(r1)+" end:"+
            signedShort2String(instruction)+"(0x"+
            short2HexString(instruction)+
            ")" + "start: " + signedShort2String(pc) +
```

```
            " (0x" + short2HexString(pc) + ")";
        addLog (log,threadNum);
    }
    else {
        setTwoWordsInstruction(true,threadNum);
        setFirstWord(instruction,threadNum);
        return;
    }


}

void xinc::zolProcess(unsigned short threadNum) {

    for (int i=0;i<7;i++) {
        if ((thread[threadNum].getZOLAE(i)==thread[threadNum].getPc())
             && (thread[threadNum].getZOLR(i)>0)) {
            thread[threadNum].setPc(thread[threadNum].getZOLAS(i));
            thread[threadNum].decreaseZOLR(i);
            if (thread[threadNum].getZOLR(i)==0) {
                thread[threadNum].setZOLAE(0,i);
                thread[threadNum].setZOLAS(0,i);
            };

        }
    }
}

// Filename: xincapp.h
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Main XInC
//
// Date: Jan 24, 2008

#include <iostream>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>

using namespace std;

#define LOGLEVEL 1

#include "xinc.h"

int main ()
{
    xinc xinc1;
    char filename[255],logfilename[255],serialInFile[255],serialOutFile[255];
    char memoryAddress1[10],memoryAddress2[10];
    char statAddress1[10],statAddress2[10];
    unsigned long clockCycle;
    filename[0]=0;
    logfilename[0]=0;
    serialInFile[0]=0;
```

```
serialOutFile[0]=0;
bool screenLog=false;
bool fileLog=false;
bool statistics=false;
bool instructionLog2=false;

while (true) {
    cout << endl;
    cout << "1. run - 1 system clock cycle" << endl;
    cout << "2. run - 8 system clock cycles" << endl;
    cout << "3. run - system clock cycles" << endl;
    cout << "4. run - breakpoint" << endl;
    cout << "5. print - registers" << endl;
    cout << "6. print - memory" << endl;
    cout << "7. print - I/O" << endl;
    cout << "8. load program " << filename << endl;
    cout << "9. reset" << endl;
    cout << "a. log file on " << logfilename << endl;
    cout << "b. log file off" << endl;
    cout << "c. Serial in file: " << serialInFile << endl;
    cout << "d. Serial out file: " << serialOutFile << endl;
    cout << "e. Screen log is " ;
    if (screenLog) {
        cout << "on" << endl;
    }
    else {
        cout << "off" << endl;
    }
    cout << "f. print program address hit statistics" << endl ;
    cout << "g. reset program address hit statistics" << endl ;
    cout << "h. print instruction type statistics" << endl ;
    cout << "i. reset instruction type statistics" << endl ;
    cout << "j. instruction type 2 statistics log is "  ;
    if (instructionLog2) {
        cout << "on" << endl;
    }
    else {
        cout << "off" << endl;
    }
    cout << "z. exit emulation" << endl ;

    cout << ">";

    char ch;
    cin >> ch;
    switch (ch) {
        case '1':
            if (filename[0]==0) {
                cout << "Hex file has not loaded!";
            }
            else {
                xinc1.runSystemClockCycles(1);

                if (!xinc1.saveNewestLog(logfilename)) {
                    cout << "Log file wrong!";
                };

            }
            break;
        case '2':
```

```
    if (filename[0]==0) {
        cout << "Hex file has not loaded!";
    }
    else {
        xinc1.runSystemClockCycles(8);

        if (!xinc1.saveNewestLog(logfilename)) {
            cout << "Log file wrong!";
        };

    }
    break;
case '3':
    if (filename[0]==0) {
        cout << "Hex file has not loaded!";
    }
    else {
    cout << "Please input the clock cycles:" ;
    cin >> clockCycle ;
    xinc1.runSystemClockCycles(clockCycle);

    if (!xinc1.saveNewestLog(logfilename)) {
        cout << "Log file wrong!";
    };

    }
    break;
case '4':
    if (filename[0]==0) {
        cout << "Hex file has not loaded!";
    }
    else {
    cout << "Please input breakpoint address (eg. 0xc000):" ;
    cin >> memoryAddress1 ;
    xinc1.runTo(hexString2Short(memoryAddress1),logfilename);


    }
    break;
case '5':
    xinc1.printRegister();
    break;
case '6':
    cout << "Please input memory address range(eg 0xc000 0xc001):" ;
    cin >> memoryAddress1 >> memoryAddress2;

    xinc1.printRam(hexString2Short(memoryAddress1),
        hexString2Short(memoryAddress2));
    break;
case '7':
    cout << "Please input I/O address range (eg 0x0 0x3):" ;
    cin >> memoryAddress1 >> memoryAddress2;

    xinc1.printIO(hexString2Short(memoryAddress1),
        hexString2Short(memoryAddress2));
    break;
case '8':
    cout << "Please input the filename:" ;
    cin >> filename;
    xinc1.reset();
```

```
    if (!xinc1.load(filename)) {
        filename[0]=0;
    };
    if (xinc1.setSerialInFile(serialInFile)) {
        cout << "Serial in file is set." << endl;
    }
    else {
        cout << "Error: Serial in file is not set." << endl;
        serialInFile[0]=0;
    }
    if (xinc1.setSerialOutFile(serialOutFile)) {
        cout << "Serial Out file is set." << endl;
    }
    else {
        cout << "Error: Serial Out file is not set." << endl;
        serialOutFile[0]=0;
    }
    break;
case '9':
    cout << "Reset XinC chip" << endl;
    xinc1.reset();
    xinc1.load(filename);
    if (xinc1.setSerialInFile(serialInFile)) {
        cout << "Serial in file is set." << endl;
    }
    else {
        cout << "Error: Serial in file is not set." << endl;
        serialInFile[0]=0;
    }
    if (xinc1.setSerialOutFile(serialOutFile)) {
        cout << "Serial Out file is set." << endl;
    }
    else {
        cout << "Error: Serial Out file is not set." << endl;
        serialOutFile[0]=0;
    }
    break;
case 'a':

    cout << "Please input the filename:" ;
    cin >> logfilename;
    cout << "Log is on." << endl;
    xinc1.setScreenLog(true);
    fileLog=true;
    break;
case 'b':
    logfilename[0]=0;
    cout << "Log is off." << endl;
    xinc1.setScreenLog(false);
    fileLog=false;
    break;
case 'c':

    cout << "Please input serial in file: " ;
    cin >> serialInFile;

    if (xinc1.setSerialInFile(serialInFile)) {
        cout << "Serial in file is set." << endl;

    }
```

```
    else {
        cout << "Error: Serial in file is not set." << endl;
        serialInFile[0]=0;

    }
    break;
case 'd':

    cout << "Please input serial out file: " ;
    cin >> serialOutFile;
    if (xinc1.setSerialOutFile(serialOutFile)) {
        cout << "Serial Out file is set." << endl;

    }
    else {
        cout << "Error: Serial Out file is not set." << endl;
        serialOutFile[0]=0;

    }

    break;
case 'e':

    screenLog=!screenLog;
    xinc1.setScreenLog(screenLog);
    break;
case 'f':
    cout << "Please input address range(eg 0xc000 0xc001):" ;
    cin >> statAddress1 >> statAddress2;

    xinc1.printStat(hexString2Short(statAddress1),
        hexString2Short(statAddress2),0);

    break;
case 'g':
    xinc1.resetStat();
    cout << "Program address hit is resetted" << endl ;

    break;
case 'h':

    xinc1.printInstructionStat();

    break;
case 'i':
    xinc1.resetInstructionStat();
    cout << "Instruction statistics is resetted" << endl ;

    break;
case 'j':

    instructionLog2=!instructionLog2;
    xinc1.setInstructionLog2(instructionLog2);
    break;
case 'z':
    xinc1.closeSerialFile();
    return 0;
default:
    ;
```

```
        }

    }


    return 0;
}


// Filename: xinclib.cpp
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Function Library Implementation
//
// Date: Jan 24, 2008

#include "xinclib.h"
#include <cstring>
#include <string>
#include <iostream>

using namespace std;

short signedExtension (short original, int bit)
{
    if ((original & (1<<(bit-1))) == 0) {

        return original;
    }
    else {
        short mask=0;
        for (int i=0;i<16;i++) {
            mask=(mask << 1);
            if (i< 16-bit) {
                mask=mask | 1;

            }
        }
        return (original | mask) ;
    }
}

bool xor2(bool op1,bool op2)
{
    if (op1==op2) {
        return false;
    }
    else {
        return true;
    }
};

short hexString2Short(char* in)
{
    short result;
    result=0;
```

```
for (unsigned int i=0;i<strlen(in);i++) {
    switch (in[i]) {
    case '0':
        result=result*16;
        break;
    case '1':
        result=result*16+1;
        break;
    case '2':
        result=result*16+2;
        break;
    case '3':
        result=result*16+3;
        break;
    case '4':
        result=result*16+4;
        break;
    case '5':
        result=result*16+5;
        break;
    case '6':
        result=result*16+6;
        break;
    case '7':
        result=result*16+7;
        break;
    case '8':
        result=result*16+8;
        break;
    case '9':
        result=result*16+9;
        break;
    case 'A':
    case 'a':
        result=result*16+10;
        break;
    case 'B':
    case 'b':
        result=result*16+11;
        break;
    case 'C':
    case 'c':
        result=result*16+12;
        break;
    case 'D':
    case 'd':
        result=result*16+13;
        break;
    case 'E':
    case 'e':
        result=result*16+14;
        break;
    case 'F':
    case 'f':
        result=result*16+15;
        break;
    default:
        result=0;
        break;
```

```
        }

    }
    return result;
}

string short2HexString(unsigned short number)
{
    int a[4];
    string hexString;
    a[0]= (number & 0xf000) >> 12;
    a[1]= (number & 0x0f00) >> 8;
    a[2]= (number & 0x00f0) >> 4;
    a[3]= (number & 0x000f) ;
    hexString="";
    for (int i=0;i<4;i++) {
        switch (a[i]) {
        case 0:
            hexString=hexString+"0";
            break;
        case 1:
            hexString=hexString+"1";
            break;
        case 2:
            hexString=hexString+"2";
            break;
        case 3:
            hexString=hexString+"3";
            break;
        case 4:
            hexString=hexString+"4";
            break;
        case 5:
            hexString=hexString+"5";
            break;
        case 6:
            hexString=hexString+"6";
            break;
        case 7:
            hexString=hexString+"7";
            break;
        case 8:
            hexString=hexString+"8";
            break;
        case 9:
            hexString=hexString+"9";
            break;
        case 10:
            hexString=hexString+"a";
            break;
        case 11:
            hexString=hexString+"b";
            break;
        case 12:
            hexString=hexString+"c";
            break;
        case 13:
            hexString=hexString+"d";
            break;
        case 14:
```

```
            hexString=hexString+"e";
            break;
        case 15:
            hexString=hexString+"f";
            break;

        }

    }

    return hexString;

}

string signedShort2String (signed short number)
{
    string result="";
    signed short current=number;
    if (number < 0) {

        current=(number^0xffff) +1;


    }

    if (current==0) return "0";
    if (current==-32768) return "-32768";

    while (current>0) {
    unsigned short digit=current % 10;
    switch (digit) {
    case 0:
        result="0"+result;
        break;
    case 1:
        result="1"+result;
        break;
    case 2:
        result="2"+result;
        break;
    case 3:
        result="3"+result;
        break;
    case 4:
        result="4"+result;
        break;
    case 5:
        result="5"+result;
        break;
    case 6:
        result="6"+result;
        break;
    case 7:
        result="7"+result;
        break;
    case 8:
        result="8"+result;
        break;
    case 9:
        result="9"+result;
```

```
        break;


    }
    current = (current - digit) / 10 ;
    }
    if (number<0) result="-"+result;
    return result;

}

string unsignedLong2String(unsigned long number)
{
    unsigned short digit;
    unsigned long current=number;
    string result="";
    if (current==0) return "0";
    while (current>0) {
    digit=current % 10;
    switch (digit) {
    case 0:
        result="0"+result;
        break;
    case 1:
        result="1"+result;
        break;
    case 2:
        result="2"+result;
        break;
    case 3:
        result="3"+result;
        break;
    case 4:
        result="4"+result;
        break;
    case 5:
        result="5"+result;
        break;
    case 6:
        result="6"+result;
        break;
    case 7:
        result="7"+result;
        break;
    case 8:
        result="8"+result;
        break;
    case 9:
        result="9"+result;
        break;


    }
    current = (current - digit) / 10 ;
    }
    return result;
}

// Filename: xincMemory16.cpp
// Author: Xin Sheng Zhou
```

```
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Class xincMemory16 Implementation
//
// Date: Jan 24, 2008

#include "xincMemory16.h"


xincMemory16::xincMemory16 ()
{
    memory=0;
};


void xincMemory16::setValue(short mem) {
    memory=mem;

};

short xincMemory16::getValue() {
    return memory;
}

// Filename: xincPeripheralRegister.cpp
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Class xincPeripheralRegister Implementation
//
// Date: Jan 24, 2008

#include "xincPeripheralRegister.h"

xincPeripheralRegister::xincPeripheralRegister () : xincMemory16()
{

        isInputSet=false;
        isOutputSet=false;


};

void xincPeripheralRegister::setIsOutputSet ( bool outputSetValue)
{
    isOutputSet=outputSetValue;
}

bool xincPeripheralRegister::getIsOutputSet ()
{
    return isOutputSet;
};

// Filename: xincThreadRegister.cpp
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
```

```
// University of Alberta
//
// Description:
// Class xincThreadRegister Implementation
//
// Date: Jan 24, 2008

#include "xincThreadRegister.h"


xincThreadRegister::xincThreadRegister ()
{
    status=0;
};


void xincThreadRegister::setPc(unsigned short newPC)
{
    pc.setValue(newPC);
};

unsigned short xincThreadRegister::getPc()
{
    return pc.getValue();
};

void xincThreadRegister::setStatus(unsigned char newStatus)
{
    status=newStatus;
};

unsigned char xincThreadRegister::getStatus()
{
    return status;
};

void xincThreadRegister::setR(short newR,short rNum)
{
    r[rNum].setValue(newR);
};

short xincThreadRegister::getR(short rNum)
{
    return r[rNum].getValue();
};

void xincThreadRegister::setStatusN ()
{
    status=status | 0x8;
};

void xincThreadRegister::clrStatusN ()
{
    status=status & 0xf7;
};

void xincThreadRegister::setStatusZ ()
{
    status=status | 0x4;
};
```

```
void xincThreadRegister::clrStatusZ ()
{
    status=status & 0xfb;
};

void xincThreadRegister::setStatusV ()
{
    status=status | 0x2;
};

void xincThreadRegister::clrStatusV ()
{
    status=status & 0xfd;
};

void xincThreadRegister::setStatusC ()
{
    status=status | 0x1;
};

void xincThreadRegister::clrStatusC ()
{
    status=status & 0xfe;
};

bool xincThreadRegister::isN1()
{
    if ((status & 8) ==0) {
        return false;
    }
    else {
        return true;
    };
};

bool xincThreadRegister::isZ1()
{
    if ((status & 4) ==0) {
        return false;
    }
    else {
        return true;
    };
};

bool xincThreadRegister::isV1()
{
    if ((status & 2) ==0) {
        return false;
    }
    else {
        return true;
    };
};

bool xincThreadRegister::isC1()
{
    if ((status & 1) ==0) {
        return false;
```

```
    }
    else {
        return true;
    };
};

void xincThreadRegister::setZOLR(unsigned short newZOLR, short rNum)
{
    zolr[rNum].setValue(newZOLR);
};

unsigned short xincThreadRegister::getZOLR(short rNum)
{
    return zolr[rNum].getValue();
};

void xincThreadRegister::increaseZOLR(short rNum)
{
    zolr[rNum].setValue(zolr[rNum].getValue()+1);
};

void xincThreadRegister::decreaseZOLR(short rNum)
{
    zolr[rNum].setValue(zolr[rNum].getValue()-1);
};

void xincThreadRegister::setZOLAS(unsigned short newZOLAS, short rNum)
{
    zolr[rNum].setAddressStart(newZOLAS);
};

unsigned short xincThreadRegister::getZOLAS(short rNum)
{
    return zolr[rNum].getAddressStart();
};
void xincThreadRegister::setZOLAE(unsigned short newZOLAE, short rNum)
{
    zolr[rNum].setAddressEnd(newZOLAE);
};

unsigned short xincThreadRegister::getZOLAE(short rNum)
{
    return zolr[rNum].getAddressEnd();
};

// Filename: xincZOLR.cpp
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Zero overhead looping implementation
//
// Date: Jan 24, 2008

#include "xincZOLR.h"

xincZOLR::xincZOLR()
{
    memory=0;
```

```
    addressStart=0;
    addressEnd=0;
};


void xincZOLR::setValue(unsigned short mem) {
    memory=mem;

};

unsigned short xincZOLR::getValue() {
    return memory;
}

void xincZOLR::decreaseValue() {
    memory--;

};

void xincZOLR::increaseValue() {
    memory++;

};

void xincZOLR::setAddressStart(unsigned short addr) {
    addressStart=addr;

};

unsigned short xincZOLR::getAddressStart() {
    return addressStart;
}

void xincZOLR::setAddressEnd(unsigned short addr) {
    addressEnd=addr;

};

unsigned short xincZOLR::getAddressEnd() {
    return addressEnd;
}

// Filename: xinc.h
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Class xinc
//
// Date: Jan 24, 2008

#include <string.h>
#include <string>
#include <iostream>
#include <fstream>
#include "xinclib.h"
#include "xincThreadRegister.h"
#include "xincPeripheralRegister.h"
```

```
#define INSTRUCTION_NUMBER 32

using namespace std;

class xinc
{
private:

    // hardware memories, registers, io buffers
    xincMemory16 ram[16384],rom[16384];
    xincPeripheralRegister peripheralRegisterRead[128],peripheralRegisterWrite[128];
    xincThreadRegister thread[8];

    // instuctions
    void iMov(int threadNum,short instruction);
    void iOutp(int threadNum,short instruction);
    void iBra(int threadNum,short instruction);
    void iAdd(int threadNum,short instruction,short method);
    void iAnd(int threadNum,short instruction,short method);
    void iBc(int threadNum,short instruction,short method);
    void iBic(int threadNum,short instruction);
    void iBis(int threadNum,short instruction);
    void iBix(int threadNum,short instruction);
    void iInp(int threadNum,short instruction);
    void iIor(int threadNum,short instruction,short method);
    void iJsr(int threadNum,short instruction,short method);
    void iLd(int threadNum,short instruction,short method);
    void iRol(int threadNum,short instruction,short method);
    void iSt(int threadNum,short instruction,short method);
    void iSub(int threadNum,short instruction);
    void iThrd(int threadNum,short instruction);
    void iXor(int threadNum,short instruction,short method);
    void iMovZOLR(int threadNum,short instruction);
    void iSetZOLA(int threadNum,short instruction);

    // emulator internal use
    unsigned long systemClockCycle;

    bool isTwoWordsInstruction[8];
    unsigned short firstWord[8];

    string newestLog;

    short braStatus[8];
    short SPI0rx,SPI0tx;
    bool screenLog;
    bool fileLog;
    bool instructionLog2;
    bool serialInOpen;
    bool serialOutOpen;

    char serialInFile[255],serialOutFile[255];
    ifstream fp_serialIn;
    ofstream fp_serialOut;

    long addressStat[65536];
    long instructionStat[32];
    string instructionName[32];
    char filenameRoot[100];
    string newestILog2;
```

```
    unsigned long getSystemClockCycle ();
    void setSystemClockCycle(unsigned long cycleNumber);


    void setTwoWordsInstruction (bool isTwoWords, short threadNum);
    bool getTwoWordsInstruction (short threadNum);
    void setFirstWord (short firstWordValue, short threadNum);
    short getFirstWord (short threadNum);
    bool isThreadRun(int threadNum);
    void runThread ();
    void ioProcess ();
    void addLog(string log,unsigned short threadNum);
    void zolProcess(unsigned short threadNum);


public:

    xinc ();
    void reset ();
    bool load (char* filename);
    void runSystemClockCycles (unsigned long length);
    void runTo (short address,char* logfilename);
    void printRegister();
    void printRam(unsigned short start,unsigned short end);
    void printIO(unsigned short start,unsigned short end);
    bool saveNewestLog(char* logfilename);
    bool setSerialInFile(char* filename);
    bool setSerialOutFile(char* filename);
    void closeSerialFile();
    void setScreenLog(bool val);
    void setFileLog(bool val);
    void setInstructionLog2(bool val);
    void printStat(unsigned short start, unsigned short end, short sortMethod);
    void resetStat();
    void printInstructionStat();
    void resetInstructionStat();
};

// Filename: xinclib.h
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Function Library
//
// Date: Jan 24, 2008

#ifndef xinclib
#define xinclib

#include <string>

using namespace std;

short signedExtension(short original, int bit);
bool xor2(bool op1,bool op2);
short hexString2Short(char* in);
string short2HexString(unsigned short number);
```

```
string unsignedLong2String(unsigned long number);
string signedShort2String(signed short number);

#endif

// Filename: xincMemory16.h
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Class xincMemory16
//
// Date: Jan 24, 2008


#ifndef xincMemory

#define xincMemory

class xincMemory16 {
private:
    short memory;  // Memory

public:
    xincMemory16 ();

    void setValue (short mem);
    short getValue ();

};

#endif

// Filename: xincPeripheralRegister.h
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Class xincPeripheralRegister
//
// Date: Jan 24, 2008

#include "xincMemory16.h"

class xincPeripheralRegister: public xincMemory16 {
private:
    bool isOutputSet;
    bool isInputSet;

public:
    xincPeripheralRegister ();

    bool getIsOutputSet ();
    void setIsOutputSet (bool outputSetValue);

};

// Filename: xincThreadRegister.h
```

```
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Class xincThreadRegister
// Including PC, Register, Zero overhead looping register set
//
// Date: Jan 24, 2008

#include "xincMemory16.h"
#include "xincZOLR.h"

#ifndef threadRegister

#define threadRegister

class xincThreadRegister {
protected:
    xincMemory16 pc;      // Program Counter
    xincMemory16 r[8];    // Registers
    xincZOLR zolr[8];     // Zero overhead looping register sets
    unsigned char status;  // Condition code

public:
    xincThreadRegister ();
    void setPc(unsigned short newPC);
    unsigned short getPc();
    void setStatus(unsigned char newStatus);
    unsigned char getStatus();
    void setR(short newR,short rNum);
    short getR(short rNum);

    void setZOLR(unsigned short newZOLR,short rNum);
    unsigned short getZOLR(short rNum);
    void increaseZOLR(short rNum);
    void decreaseZOLR(short rNum);
    void setZOLAS(unsigned short newZOLAS,short rNum);
    unsigned short getZOLAS(short rNum);
    void setZOLAE(unsigned short newZOLAE,short rNum);
    unsigned short getZOLAE(short rNum);

    void setStatusN ();
    void clrStatusN ();
    void setStatusZ ();
    void clrStatusZ ();
    void setStatusV ();
    void clrStatusV ();
    void setStatusC ();
    void clrStatusC ();
    bool isN1();
    bool isZ1();
    bool isV1();
    bool isC1();
};

#endif

// Filename: xincZOLR.h
// Author: Xin Sheng Zhou
```

```
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Class xincZOLR
// Zero overhead looping registers
// Date: Jan 24, 2008

class xincZOLR {
private:
    unsigned short memory; // Program Counter
    unsigned short addressStart;  // Looping start register
    unsigned short addressEnd;    // Looping end register

public:
    xincZOLR();

    void setValue (unsigned short mem);
    unsigned short getValue ();
    void decreaseValue ();
    void increaseValue ();
    void setAddressStart (unsigned short addr);
    unsigned short getAddressStart ();
    void setAddressEnd (unsigned short addr);
    unsigned short getAddressEnd ();

};
```

Appendix B

LDPC-CC Encoding and Decoding Algorithm on XInC Microprocessor Assembly Language

B.1 LDPC-CC Encoding Algorithm Assembly Language on XInC

```
// Filename: Main.asm
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// LDPC-CC Encoder
//
// Date: Jan 24, 2008

#include "..\..\XInC Library\XInC.h"
#include "Constants.h"

#define __T0__
#define __T1__

//=============================================================================
// Code and Data Size:
//  After assembly, check the values assigned to these constants in the list file.
//=============================================================================

SizeOfAppCode = (__AppCode_End__ - __AppCode_Start__)
SizeOfAppData = (__AppData_End__ - __AppData_Start__)
SizeOfShortData = (__ShortData_End__ - __ShortData_Start__)

FreeAppCodeSpace = (__AppData_Start__ - __AppCode_End__) // If any of these three
FreeAppDataSpace = (kRAM_End - 127 - __AppData_End__)    // constants are negative,
```

```
FreeShortDataSpace  = (kRAM_End - __ShortData_End__)     // there is an overflow.

//=============================================================================
// Header Files:
//   This section includes files defining constants.
//=============================================================================


//=============================================================================
// Code Space:
//   Only Code should be included in this segment.
//=============================================================================

@ = kRAM_Block0_Start
__AppCode_Start__:

//-----------------------------------------
// Initialization Code

#include "Init.asm"

    bra Thread0

#include "..\..\XInC Library\XPD_Echo.asm"
#include "..\..\XInC Library\XPD_Echo_Data.asm"
#include "..\..\XInC Library\LEDs.asm"

//-----------------------------------------
// Thread Code

#ifdef __T0__
Thread0:                            // Thread 0 Code
    bra Thread0
#endif

#ifdef __T1__
Thread1:                            // Thread 1 Code

    mov r1, kXPD_BaudRate_115200 + kXPD_ClockLE_12MHz
    jsr r6, XPD_Configure

// Initialize the LEDs
    jsr r6, InitializeLEDs
    mov r1, 0xFFFF
    jsr r6, TurnOnLEDs // Turn on all LEDs to indicate the program has loaded

// Initialize FIFO queue

__FOR_1_INIT:
    mov r0, 0
    st  r0, i
__FOR_1_COND:
    ld  r0, i
    mov r1,encoderQueueSize
    sub r0,r0,r1
    bc  ZS, __FOR_1_END
__FOR_1_BODY:
    ld  r0,i
    mov r1,0
    st  r1,r0,queue
```

```
__FOR_1_INCR:
    ld   r0, i
    add  r2, r0, 1
    st   r2, i
    bra  __FOR_1_COND
__FOR_1_END:

_ENCODING_START:
    jsr  r6, XPD_ReadByte     // Read data
    jsr  r6, XPD_WriteByte    // Echo data
    ld   r0,head  // queue[head]=current
    st   r1,r0,queue
    add  r0,r0,1  // head++
    st   r0,head

//  if (head==encoderQueueSize) head==0;
    mov  r2,encoderQueueSize
    sub  r0,r0,r2
    bc   ZS, _IF_1_RUN_1
    bra  IF_1_CONT_1
_IF_1_RUN_1:
    mov  r0,0
    st   r0,head
_IF_1_CONT_1:
// queue[head]=0
    ld   r0,head
    mov  r2,0
    st   r2,r0,queue

//Get first data
    ld   r0,head
    add  r0,r0,encoderQueueSize+1
    ld   r2,positionTablePointer
    ld   r2,r2,matrix
    sub  r0,r0,r2
    sub  r2,r0,encoderQueueSize
    bc   NS,_SAVE_N
    ld   r1,r2,queue
    bra  _SAVE_CONT
_SAVE_N:
    ld   r1,r0,queue
_SAVE_CONT:

// positionTablePointer++
    ld   r0,positionTablePointer
    add  r0,r0,1
    st   r0,positionTablePointer

// checkNum=0
    mov  r0,0
    st   r0,checkNum

_CHECK_COND:
    ld   r0,checkNum   // checkNum > checkDegree?
    add  r0,r0,1  // checkNum++
    st   r0,checkNum
    sub  r0,r0,checkDegree
    bc   ZS,_CHECK_END
_CHECK_PROCESS:
```

```
// Get following data
    ld   r0,head
    add  r0,r0,encoderQueueSize+1
    ld   r2,positionTablePointer
    ld   r2,r2,matrix
    sub  r0,r0,r2
    sub  r2,r0,encoderQueueSize
    bc   NS,_SAVE_N_1
    ld   r4,r2,queue
    bra  _SAVE_CONT_1
_SAVE_N_1:
    ld   r4,r0,queue
_SAVE_CONT_1:

// positionTablePointer++
    ld   r0,positionTablePointer
    add  r0,r0,1
    st   r0,positionTablePointer

// Xor two numbers
    xor  r1,r1,r4

// Branch to parity check start
    bra  _CHECK_COND
_CHECK_END:


    ld   r0,positionTablePointer      // positionTablePointer > positionTableSize?
    sub  r0,r0,positionTableSize
    bc   ZS,_PHASE_CONT
    bra  _PHASE_END
_PHASE_CONT:
    mov  r0,0
    st   r0,positionTablePointer
_PHASE_END:


    ld   r0,head    //queue[head]=current
    st   r1,r0,queue
    add  r0,r0,1
    st   r0,head

// if (head==encoderQueueSize) head==0;
    mov  r2,encoderQueueSize
    sub  r0,r0,r2
    bc   ZS, _IF_1_RUN
    bra  _IF_1_CONT
_IF_1_RUN:
    mov  r0,0
    st   r0,head
_IF_1_CONT:

// Echo the parity check result
    ior  r1,r1,560
    jsr  r6, XPD_WriteByte

_ENCODING_END:
    bra  _ENCODING_START

stop:
```

```
    bra stop
#endif


//-----------------------------------------
// Other Source Files


__AppCode_End__:

//=============================================================================
// Data Space:
//  All Data must be in a separate 2kWord Memory Block from any Code.
//=============================================================================

@ = (@ + 0x800 - 1) & -0x800   // Round up to the next 2kWord Memory Block
__AppData_Start__:

#include "Long_Data.asm"

__AppData_End__:

//=============================================================================
// Short Address Space:
// Any Data placed in this space may be accessed with a single word
// instruction.
//=============================================================================

@ = kRAM_End - 127              // Start of the short address space
__ShortData_Start__:

#include "Short_Data.asm"

__ShortData_End__:
```

B.2 LDPC-CC PIBF Algorithm with Zero-overhead Looping on XInC

```
// Filename: Main.asm
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// LDPC-CC Decoder
// Parallel Improved Bit Flipping Algorithm
// With Zero-overhead Looping
//
// Date: Jan 24, 2008

#include "..\..\XInC Library\XInC.h"
#include "Constants.h"

// Start thread 1 at the beginning
#define __T1__
```

```
//=============================================================================
// Code and Data Size:
//  After assembly, check the values assigned to these constants in the
//  list file.
//=============================================================================

SizeOfAppCode   = (__AppCode_End__ - __AppCode_Start__)
SizeOfAppData   = (__AppData_End__ - __AppData_Start__)
SizeOfShortData = (__ShortData_End__ - __ShortData_Start__)

FreeAppCodeSpace = (__AppData_Start__ - __AppCode_End__) // If any of these three
FreeAppDataSpace  = (kRAM_End - 127 - __AppData_End__) // constants are negative,
FreeShortDataSpace  = (kRAM_End - __ShortData_End__) // there is an overflow.

//=============================================================================
// Code Space:
//  Only Code should be included in this segment.
//=============================================================================

@ = kRAM_Block0_Start
__AppCode_Start__:

//-----------------------------------------
// Initialization Code

#include "Init.asm"

    bra @

#include "..\..\XInC Library\XPD_Echo.asm"
#include "..\..\XInC Library\XPD_Echo_Data.asm"
#include "..\..\XInC Library\LEDs.asm"

//-----------------------------------------
// Thread Code

Thread1:  // Thread 1 Code

    mov r1, kXPD_BaudRate_115200 + kXPD_ClockLE_12MHz
    jsr r6, XPD_Configure

// Initialize the LEDs
    jsr r6, InitializeLEDs
    mov r1, 0xFFFF
    jsr r6, TurnOnLEDs  // Turn on all LEDs to indicate the program has loaded

// Distribute Decoding Proceesor to threads

    mov r0,0
distributeProcessorID:
    add r5,r0,2
    st r0,r5,processorID
    add r0,r0,1
    sub r5,r0,nProcNum
    bc ZS, distributeProcessIDEnd
    bra distributeProcessorID
distributeProcessIDEnd:

// Start decoding thread 2-7
```

```
    mov r3,0xfffc  // Give r3 initial value, only thread 0 and 1 is running
    mov r4,2   // Starting from thread 2
setThread:
    rol r2,r4,3
    ior r2,r2,7
    outp r2, SCUpntr
    ld r2,r4, SP_ADDRESS
    outp r2, SCUreg
    mov r2,Thread2
    outp r2, SCUpc
    mov r1,1
    rol r1,r1,r4
    xor r1,r1,0xffff
    and r3,r3,r1 // r3 controls which thread will run.
    add r4,r4,1
    mov r2,2
    sub r2,r4,r2
    sub r2,r2,nProcNum
    bc ZS, setThreadEnd
    bra setThread
setThreadEnd:
    st r3,threadPattern

// initial matLLRBuffer with einitLLR
    mov r0, 0
    mov r1,nBufLength
    mov r2,0
initialMatLLRBuffer:
    sub r3,r0,r1
    bc ZS, initialMatLLRBufferEnd
    st r2,r0,matLLRBuffer
    add r0, r0, 1
    bra initialMatLLRBuffer
initialMatLLRBufferEnd:

    mov r1,0
    st r1,pnSymbolDegPointer // pnSymbolDegPointer=0
    st r1,blockRowPosition

// begin initial pnPosition
// This block can be removed if the check degree is a constant number.
    mov r0,0
    mov r2,0
initialPnPositionCond:
    sub r1,r0,nProcNum
    bc ZS,initialPnPositionEnd
initialPnPositionBody:
    st r2,r0,pnPosition
    add r2,r2,nCheckDegMax
    add r0,r0,1
    bra initialPnPositionCond
initialPnPositionEnd:
// end initial pnPosition

// Begin initialize processorPhase, pnCheckDegRowPosition
    mov r0,0
    mov r2,nProcSize // r2: current phase
    mov r3,nProcSize*nCheckDegMax // r3: current phase row position
    mov r4,nProcNum
initializeProcesorPhaseCond:
```

```
    sub r1,r0,r4
    bc ZS,initializeProcesorPhaseEnd
initializeProcesorPhaseBody:
    sub r3,r3,nProcSize*nCheckDegMax
    sub r2,r2,nProcSize
    bc NS,initializeProcesorPhaseMod
    bra initializeProcesorPhaseStore
initializeProcesorPhaseMod:
    add r3,r3,nCodeT*nCheckDegMax
    add r2,r2,nCodeT
    bc NS,initializeProcesorPhaseMod
initializeProcesorPhaseStore:
    st r3,r0,pnCheckDegRowPosition
    add r0,r0,1
    bra initializeProcesorPhaseCond
initializeProcesorPhaseEnd:
// End initialize processorPhase

// Begin initialize pnSymbolDegRowPosition
    mov r3,0
    mov r0,-nCodeM*nCodeC+nProcSize*nCodeC
initializepnSymbolDegRowPositionCond:
    sub r1,r3,nProcNum
    bc ZS,initializepnSymbolDegRowPositionEnd
    sub r0,r0,nProcSize*nCodeC
addPnSymbolDegRowPosition:
    add r0,r0,nCodeT*nCodeC
    bc NS, addPnSymbolDegRowPosition
subPnSymbolDegRowPosition:
    sub r1,r0,nCodeT*nCodeC
    bc NS,subPnSymbolDegRowPositionEnd
    mov r0,r1
    bra subPnSymbolDegRowPosition
subPnSymbolDegRowPositionEnd:
    add r3,r3,1
    bra initializepnSymbolDegRowPositionCond
initializepnSymbolDegRowPositionEnd:
// End initialize pnSymbolDegRowPosition

//Begin initialize pnSymbolMatLLRPosition
    mov r3,nProcNum-1
    mov r1,2*nBlockLength
initializepnSymbolMatLLRPositionCond:
    sub r0,r3,0
    bc NS, initializepnSymbolMatLLRPositionEnd
initializepnSymbolMatLLRPositionBody:
    st r1,r3,pnSymbolMatLLRPosition
    add r1,r1,nEachProcBufLength
    sub r3,r3,1
    bra initializepnSymbolMatLLRPositionCond
initializepnSymbolMatLLRPositionEnd:
//End initialize pnSymbolMatLLRPosition

//Begin initialize procMemoryStart
    mov r3,0
    mov r1,0
initializeProcMemoryStartCond:
    sub r0,r3,nProcNum
    bc ZS, initializeProcMemoryStartEnd
initializeProcMemoryStartBody:
```

```
    st r1,r3,procMemoryStart
    add r1,r1,nEachProcBufLength
    add r3,r3,1
    bra initializeProcMemoryStartCond
initializeProcMemoryStartEnd:
//End initialize ProcMemoryStart

// input first data
    ld r4,blockRowPosition
    mov r5,0
inputDataCond1:
    sub r6,r5,nCodeC
    bc ZS,inputDataEnd1
    jsr r6, XPD_ReadByte
    and r1,r1,0xff
    rol r0,r1,8
    jsr r6, XPD_ReadByte
    and r1,r1,0xff
    ior r1,r1,r0     // r1:input data
    mov r2,nSymbolDegMax+1

// store input data to matLLRBuffer
    mov   r3, 0
storeDataCont1:
    sub r6,r3,r2
    bc  ZS, storeDataEnd1
    add r6,r4,r3
    st r1,r6, matLLRBuffer
    add   r3, r3, 1
    bra storeDataCont1
storeDataEnd1:

    add r4,r4,nBufWidth
    add r5,r5,1
    bra inputDataCond1
inputDataEnd1:

// Set start of the blockRowPosition as nBlockLength
    mov r1,nBlockLength
    st r1,blockRowPosition

// Decoding start
decodingStart:
startParallel:
// start parallel decoding
    mov r4,2   // Decoding processor thread starting from 2
setThread1:
    rol r2,r4,3
    ior r2,r2,7
    outp r2, SCUpntr
    ld  r2,r4, SP_ADDRESS
    outp r2, SCUreg
    mov r2,Thread2
    outp r2, SCUpc
    add r4,r4,1
    sub r2,r4,nProcNum+2
    bc ZS, startThread1
    bra setThread1
startThread1:
    ld r3,threadPattern
```

```
    outp r3, SCUstop
endParallel:

// input data
    ld r4,blockRowPosition
    mov r5,0
inputDataCond:
    sub r6,r5,nCodeC
    bc ZS,inputDataEnd
    jsr r6, XPD_ReadByte
    and r1,r1,0xff
    rol r0,r1,8
    jsr r6, XPD_ReadByte
    and r1,r1,0xff
    ior r1,r1,r0     // r1:input data

    ld r0, pnSymbolDegPointer
    ld r2, r0, pnSymbolDeg
    add r2,r2,1      // r2 <- pnSymbolDeg[pnSymbolDegPointer] +1
    add r0,r0,1
    st r0, pnSymbolDegPointer  // pnSymbolDegPointer++
    sub r6,r0, nCodeC*nCodeT   // pnSymbolDegPointer mod nCodeC*nCodeT
    bc ZS, mod2
    bra mod2Cont
mod2:
    st r6, pnSymbolDegPointer
mod2Cont:

    mov   r3, 0
storeDataCont:
    sub r6,r3,r2
    bc  ZS, storeDataEnd
    add r6,r4,r3
    st r1,r6, matLLRBuffer
    add   r3, r3, 1
    bra storeDataCont
storeDataEnd:
    add r4,r4,nBufWidth
    add r5,r5,1
    bra inputDataCond
inputDataEnd:
// Data input end

// Hard Decision begin
    ld r2,blockRowPosition
    add r2,r2,nBlockLength
    sub r1,r2,nBufLength
    bc NC,modDone
    mov r1,r2
modDone:
    mov r3,0
hardDecisionCont:
    sub r0,r3,nCodeC
    bc ZS,hardDecisionEnd

// Echo result
    st r1,r7,10
    mov r1,r0
//  jsr r6,XPD_EchoHex
    jsr r6, XPD_EchoUnsignedDec
```

```
    mov r1,space
    jsr r6,XPD_EchoString
printEnd:

    ld r1,r7,10
    add r3,r3,1
    add r1,r1,nBufWidth
    bra hardDecisionCont
hardDecisionEnd:
// Hard Decision end

// Pointer Increment
// Begin blockRowPosition=(blockRowPosition+blockLength) mod nBufLength
// This is the first item of pnTeseRows[nThisNode]
    ld r1,blockRowPosition
    add r1,r1,nBlockLength
    sub r0,r1,nBufLength
    bc NS,storeBlockRowPositionFromR1
    st r0,blockRowPosition
    bra storeBlockRowPositionEnd
storeBlockRowPositionFromR1:
    st r1,blockRowPosition
storeBlockRowPositionEnd:
// End blockRowPosition=(blockRowPosition+blockLength) mod nBufLength

// Query if decoding processor is finish?
    ld r3,threadPattern
    bc ZS, queryEnd
    ior r3,r3,0xff03
    xor r3,r3,0xffff
queryThreadStatus:
    inp r1, SCUbkpt
    and r1,r1,0xfc
    sub r1,r1,r3
    bc ZS,threadDone
    bra queryThreadStatus
threadDone:
// Stop Thread 2 - Thread 6
    mov r3,0xfc
    outp r3, SCUstop
queryEnd:

decodingEnd:
    bra decodingStart

stop:
    bra stop

Thread2:

    thrd r1
    ld r2,r1,processorID // r2: processorID

//Begin calculate pnSymbolMatLLRPosition
    ld r0,r2,pnSymbolMatLLRPosition
    add r0,r0,nCodeC*nBufWidth
    sub r1,r0,nBufLength
    bc NC, replacePnSymbolMatLLRPosition
    bra storePnSymbolMatLLRPosition
replacePnSymbolMatLLRPosition:
```

```
    mov r0,r1
storePnSymbolMatLLRPosition:
    st r0,r2,pnSymbolMatLLRPosition
// End calculate pnSymbolMatLLRPosition

// Begin load peTheseLLRs
    mov r3,0
    mov r5,0   // r5 : nThisNode
    0x3921,0x5 // movZOLR r1,5
loadPeTheseLLRsCond:
    0x3922,loadPeTheseLLRsEnd  // setZOLA r1,
loadPeTheseLLRsBody:
// r6: position in matOnesInPcmYRowPosition and matOnesInPcmX
    ld r6,r2,pnCheckDegRowPosition
    add r6,r6,r5
// r4: position in peTheseLLRs and pnMatLLRPosition
    ld r4,r2,pnPosition
    add r4,r4,r5
    ld r0,r6,matOnesInPcmYRowPosition
    ld r1,r2,pnSymbolMatLLRPosition
    add r1,r1,nCodeC*nBufWidth*nProcSize
    sub r1,r1,r0
    sub r0,r1,nBufLength
    bc NC,replacePnTheseRows
    bra storePnTheseRows
replacePnTheseRows:
    add r1,r0,0
storePnTheseRows:
    ld r0,r6,matOnesInPcmX
    add r0,r0,r1
    st r0,r4,pnMatLLRPosition
    ld r0,r0,matLLRBuffer
    xor r3,r3,r0
    add r5,r5,1
loadPeTheseLLRsEnd:
// End load peTheseLLRs

// Begin Check Node Operation
    ld r5,r2,pnPosition   // load position first
    st r2,r7,0    // push r2 to the buffer

    mov r2,r3

// Store back to matLLRBuffer
    mov r4,0
checkNodeCond3:
    0x3921,nCheckDegMax-1 // movZOLR r1, nCheckDegMax-1
    0x3922,checkNodeEnd3   // setZOLR r1, checkNodeEnd3
checkNodeBody3:
    add r3,r5,r4
    ld r3,r3,pnMatLLRPosition
    st r2,r3,matLLRBuffer
    add r4,r4,1
checkNodeEnd3:

    ld r2,r7,0    // pop up r2
// End check node operation, r5 is released

// Begin variable node , only r2 is used at this stage
    ld r1,r2,pnSymbolMatLLRPosition
```

```
    0x3901,nCodeC-1 // movZOLR r0, nCodeC-1
    0x3902,variableNodeEnd   // setZOLR r0, variableNodeEnd
variableNodeCond:

variableNodeBody:
    st r2,r7,0
    mov r6,65535  // r6: all 3 errors
    mov r4,0 // r4: at least 1 error
    mov r2,0 // r2: 1 error or 3 errors
    mov r5,1 // r5, loop variable
    0x3921,nSymbolDegMax-1 // movZOLR r1, nSymbolDegMax-1
    0x3922,eTotalSumAddEnd // setZOLR r1, eTotalSumAddEnd
eTotalSumAddCond:
    add r0,r1,r5
    ld r3,r0,matLLRBuffer
    and r6,r6,r3
    ior r4,r4,r3
    xor r2,r2,r3
    add r5,r5,1
eTotalSumAddEnd:
    xor r4,r4,r2 // 2 errors
    ior r4,r4,r6 // 2 errors or 3 errors
    ld r2,r7,0
// r6(3 errors), r4 (3 or 2 errors)
    and r3,r2,1 // Which processor?
    bc ZC,twoMoreErrors
    ld r3,r1,matLLRBuffer
    xor r3,r3,r6
    bra storeFlippingResult
twoMoreErrors:
    ld r3,r1,matLLRBuffer
    xor r3,r3,r4

// Store flipping result
storeFlippingResult:
    mov r5,0
storeCont:
    sub r4,r5,nSymbolDegMax+1
    bc ZS,endStore
    add r4,r1,r5
    st  r3,r4,matLLRBuffer
    add r5,r5,1
    bra storeCont
endStore:

eTmpEnd:
    add r1,r1,nBufWidth
variableNodeEnd:
// End variable node

// pnCheckDegRowPosition=(pnCheckDegRowPosition+nCheckDegMax)
// mod nCodeT*nCheckDegMax
    ld r0,r2,pnCheckDegRowPosition
    add r0,r0,nCheckDegMax
    sub r1,r0,nCodeT*nCheckDegMax
    bc NC,replaceCheckDegRowPosition
    bra storeCheckDegRowPosition
replaceCheckDegRowPosition:
    add r0,r1,0
storeCheckDegRowPosition:
```

```
    st r0,r2,pnCheckDegRowPosition

bra @


//-----------------------------------------
// Other Source Files

_AppCode_End__:

//=========================================================================
// Data Space:
//  All Data must be in a separate 2kWord Memory Block from any Code.
//=========================================================================

@ = (@ + 0x800-1) & -0x800 // Round up to the next 2kWord Memory Block
__AppData_Start__:

#include "Long_Data.asm"

__AppData_End__:

//=========================================================================
// Short Address Space:
//  Any Data placed in this space may be accessed with a single word
// instruction.
//=========================================================================

@ = kRAM_End - 127  // Start of the short address space
__ShortData_Start__:

#include "Short_Data.asm"

__ShortData_End__:
```

B.3 LDPC-CC Min-Sum Algorithm Assembly Language on XInC

```
// Filename: Main.asm
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// LDPC-CC Decoder
// Min-Sum Algorithm
//
// Date: Jan 24, 2008

#include "..\..\XInC Library\XInC.h"
#include "Constants.h"

// Define the initial running threads
#define __T0__
#define __T1__
```

```
//=============================================================================
// Code and Data Size:
//  After assembly, check the values assigned to these constants in the
//  list file.
//=============================================================================

SizeOfAppCode       = (__AppCode_End__ - __AppCode_Start__)
SizeOfAppData       = (__AppData_End__ - __AppData_Start__)
SizeOfShortData     = (__ShortData_End__ - __ShortData_Start__)

FreeAppCodeSpace    = (__AppData_Start__ - __AppCode_End__)
FreeAppDataSpace    = (kRAM_End - 127 - __AppData_End__)
FreeShortDataSpace  = (kRAM_End - __ShortData_End__)

//=============================================================================
// Code Space:
//  Only Code should be included in this segment.
//=============================================================================

@ = kRAM_Block0_Start
__AppCode_Start__:

//-----------------------------------------
// Initialization Code

#include "Init.asm"

bra @

#include "..\..\XInC Library\XPD_Echo.asm"
#include "..\..\XInC Library\XPD_Echo_Data.asm"
#include "..\..\XInC Library\LEDs.asm"

//-----------------------------------------
// Thread Code

#ifdef __T0__
Thread0:                        // Thread 0 Code
    bra Thread0
#endif

Thread1:                        // Thread 1 Code

    #include "Thread1.asm"

    mov r1, kXPD_BaudRate_115200 + kXPD_ClockLE_12MHz
    jsr r6, XPD_Configure

// Initialize the LEDs
    jsr r6, InitializeLEDs

    mov r1, 0xFFFF
    jsr r6, TurnOnLEDsm // Turn on all LEDs to indicate the program has loaded

// Distribute proceesorID
    mov r0,0
_DISTRIBUTE_PROCESSOR_ID:
    add r5,r0,2
    st r0,r5,processorID
    add r0,r0,1
```

```
    sub r5,r0,nProcNum
    bc ZS, _END_DISTRIBUTE_PROCESSOR_ID
    bra _DISTRIBUTE_PROCESSOR_ID
_END_DISTRIBUTE_PROCESSOR_ID:

// Start parallel decoding
    mov r3,0xfffc  // Give r3 initial value, only thread 0 and 1 is running
    mov r4,2    // Decoding processor from thread 2
_SET_THREAD:
    rol r2,r4,3
    ior r2,r2,7
    outp r2, SCUpntr
    ld  r2,r4, SP_ADDRESS
    outp r2, SCUreg
    mov r2,Thread2
    outp    r2, SCUpc

// r3 controls which thread will run.
    mov r1,1
    rol r1,r1,r4
    xor r1,r1,0xffff
    and r3,r3,r1
    add r4,r4,1
    mov r2,2
    sub r2,r4,r2
    sub r2,r2,nProcNum
    bc ZS, _START_THREAD
    bra _SET_THREAD
_START_THREAD:
    st r3,threadPattern

// Initialize matLLRBuffer with eInitLLR
__FOR_2_INIT:
    mov r0, 0
    mov r1,nBufLength
    mov r2,eInitLLR
__FOR_2_COND:
    sub r3,r0,r1
    bc  ZS, __FOR_2_END
__FOR_2_BODY:
    st  r2,r0,matLLRBuffer
__FOR_2_INCR:
    add   r0, r0, 1
    bra __FOR_2_COND
__FOR_2_END:

    mov r1,0
    st r1,pnSymbolDegPointer
    st r1,blockRowPosition

// Initialize pnPosition
    mov r0,0
    mov r2,0
initialPnPosition_Cond:
    sub r1,r0,nProcNum
    bc ZS,initialPnPositionEnd
initialPnPositionBody:
    st r2,r0,pnPosition
    add r2,r2,nCheckDegMax
    add r0,r0,1
```

```
    bra initialPnPosition_Cond
initialPnPositionEnd:

// Initialize processorPhase, pnCheckDegRowPosition
    mov r0,0
    mov r2,nProcSize // r2: current phase
    mov r3,nProcSize*nCheckDegMax // r3: current phase row position
    mov r4,nProcNum
initializeProcesorPhase_Cond:
    sub r1,r0,r4
    bc ZS,initializeProcesorPhase_End
initializeProcesorPhase_Body:
    sub r3,r3,nProcSize*nCheckDegMax
    sub r2,r2,nProcSize
    bc NS,initializeProcesorPhase_Mod
    bra initializeProcesorPhase_Store
initializeProcesorPhase_Mod:
    add r3,r3,nCodeT*nCheckDegMax
    add r2,r2,nCodeT
    bc NS,initializeProcesorPhase_Mod
initializeProcesorPhase_Store:
    st r2,r0,processorPhase
    st r3,r0,pnCheckDegRowPosition
    add r0,r0,1
    bra initializeProcesorPhase_Cond
initializeProcesorPhase_End:

// Initialize pnSymbolDegRowPosition
    mov r3,0
    mov r0,-nCodeM*nCodeC+nProcSize*nCodeC
initializepnSymbolDegRowPosition_Cond:
    sub r1,r3,nProcNum
    bc ZS,initializepnSymbolDegRowPosition_End
    sub r0,r0,nProcSize*nCodeC
add_pnSymbolDegRowPosition:
    add r0,r0,nCodeT*nCodeC
    bc NS, add_pnSymbolDegRowPosition
sub_pnSymbolDegRowPosition:
    sub r1,r0,nCodeT*nCodeC
    bc NS,sub_pnSymbolDegRowPosition_End
    mov r0,r1
    bra sub_pnSymbolDegRowPosition
sub_pnSymbolDegRowPosition_End:
    st r0,r3,pnSymbolDegRowPosition
    add r3,r3,1
    bra initializepnSymbolDegRowPosition_Cond
initializepnSymbolDegRowPosition_End:

// Initialize pnSymbolMatLLRPosition
    mov r3,nProcNum-1
    mov r1,2*nBlockLength
initializepnSymbolMatLLRPosition_Cond:
    sub r0,r3,0
    bc NS, initializepnSymbolMatLLRPosition_End
initializepnSymbolMatLLRPosition_Body:
    st r1,r3,pnSymbolMatLLRPosition
    add r1,r1,nEachProcBufLength
    sub r3,r3,1
    bra initializepnSymbolMatLLRPosition_Cond
initializepnSymbolMatLLRPosition_End:
```

```
// Initialize procMemoryStart
    mov r3,0
    mov r1,0
initializeProcMemoryStart_Cond:
    sub r0,r3,nProcNum
    bc ZS, initializeProcMemoryStart_End
initializeProcMemoryStart_Body:
    st r1,r3,procMemoryStart
    add r1,r1,nEachProcBufLength
    add r3,r3,1
    bra initializeProcMemoryStart_Cond
initializeProcMemoryStart_End:

// input data
    ld r4,blockRowPosition
    mov r5,0
    inputDataCond1:
    sub r6,r5,nCodeC
    bc ZS,inputDataEnd1
    jsr r6, XPD_ReadByte
    and r1,r1,0xff
    rol r0,r1,8
    jsr r6, XPD_ReadByte
    and r1,r1,0xff
    ior r1,r1,r0     // r1: input data
    ld r0, pnSymbolDegPointer
    ld r2, r0, pnSymbolDeg
    add r2,r2,1 // r2: pnSymbolDeg[pnSymbolDegPointer] +1
    add r0,r0,1
    st r0, pnSymbolDegPointer // pnSymbolDegPointer++
    sub r6,r0, nCodeC*nCodeT  // pnSymbolDegPointer mod nCodeC*nCodeT
    bc ZS, _MOD21
    bra _MOD2_CONT1
_MOD21:
    st r6, pnSymbolDegPointer
_MOD2_CONT1:

__FOR_3_INIT1:
    mov   r3, 0
__FOR_3_COND1:
    sub r6,r3,r2
    bc  ZS, __FOR_3_END1
__FOR_3_BODY1:
    add r6,r4,r3
    st r1,r6, matLLRBuffer
__FOR_3_INCR1:
    add   r3, r3, 1
    bra __FOR_3_COND1
__FOR_3_END1:

    add r4,r4,nBufWidth
    add r5,r5,1
    bra inputDataCond1
inputDataEnd1:
    mov r1,nBlockLength
    st r1,blockRowPosition

_DECODING_START:
_START_PARALLEL:
```

```
// start decoding processors
    mov r4,2    // Decoding processor starting from thread 2
_SET_THREAD1:
    rol r2,r4,3
    ior r2,r2,7
    outp r2, SCUpntr
    ld  r2,r4, SP_ADDRESS
    outp r2, SCUreg
    mov r2,Thread2
    outp r2, SCUpc
    add r4,r4,1
    sub r2,r4,nProcNum+2
    bc ZS, _START_THREAD1
    bra _SET_THREAD1
_START_THREAD1:
    ld r3,threadPattern
    outp    r3, SCUstop
_END_PARALLEL:

// input data
    ld r4,blockRowPosition
    mov r5,0
    inputDataCond:
    sub r6,r5,nCodeC
    bc ZS,inputDataEnd
    jsr r6, XPD_ReadByte
    and r1,r1,0xff
    rol r0,r1,8
    jsr r6, XPD_ReadByte
    and r1,r1,0xff
    ior r1,r1,r0    // r1 : input data
    ld r0, pnSymbolDegPointer
    ld r2, r0, pnSymbolDeg
    add r2,r2,1 // r2 : pnSymbolDeg[pnSymbolDegPointer] +1
    add r0,r0,1
    st r0, pnSymbolDegPointer // pnSymbolDegPointer++
    sub r6,r0, nCodeC*nCodeT  // pnSymbolDegPointer mod nCodeC*nCodeT
    bc ZS, _MOD2
    bra _MOD2_CONT
_MOD2:
    st r6, pnSymbolDegPointer
_MOD2_CONT:

__FOR_3_INIT:
    mov   r3, 0
__FOR_3_COND:
    sub r6,r3,r2
    bc  ZS, __FOR_3_END
__FOR_3_BODY:
    add r6,r4,r3
    st r1,r6, matLLRBuffer
__FOR_3_INCR:
    add   r3, r3, 1
    bra __FOR_3_COND
__FOR_3_END:

    add r4,r4,nBufWidth
    add r5,r5,1
    bra inputDataCond
inputDataEnd:
```

```
// Begin Hard Decision
    ld r2,blockRowPosition
    add r2,r2,nBlockLength
    sub r1,r2,nBufLength
    bc NC,Mod_Done
    mov r1,r2
Mod_Done:

    mov r3,0
    hardDecision_Cond:
    sub r0,r3,nCodeC
    bc ZS,hardDecision_End
hardDecision_Body:
    ld r4,r2,pnSymbolDegRowPosition
    add r4,r4,r3
    ld r4,r4,pnSymbolDeg
    add r4,r4,1 // pnSymbolDeg[nThisPhase*nCodeC+i]+1
    mov r6,0  // r6, eTotalSum
    mov r5,0 // r5, loop variable
    eTotalSum_Add_Cond1:
    sub r0,r5,r4
    bc ZS,eTotalSum_Add_End1
    add r0,r1,r5

    ld r0,r0,matLLRBuffer

    add r6,r6,r0  // eTotalSum=eTotalSum+ ...
    add r5,r5,1
    bra eTotalSum_Add_Cond1
eTotalSum_Add_End1:

    st r1,r7,10
    xor r1,r6,0
    bc NS,printOne
    mov r1,zero
    jsr r6,XPD_EchoString
    mov r1,space
    jsr r6,XPD_EchoString
    bra printEnd
printOne:
    mov r1,one
    jsr r6,XPD_EchoString
    mov r1,space
    jsr r6,XPD_EchoString
printEnd:
    ld r1,r7,10

    add r3,r3,1
    add r1,r1,nBufWidth
    bra hardDecision_Cond
hardDecision_End:

// Pointer Increment

// Begin blockRowPosition=(blockRowPosition+blockLength) mod nBufLength
// This is the first item of pnTeseRows[nThisNode]
    ld r1,blockRowPosition
    add r1,r1,nBlockLength
    sub r0,r1,nBufLength
```

```
    bc NS,storeBlockRowPositionFromR1
    st r0,blockRowPosition
    bra storeBlockRowPositionEnd
    storeBlockRowPositionFromR1:
    st r1,blockRowPosition
    storeBlockRowPositionEnd:

// Query if decoding processor is finished
    ld r3,threadPattern
    bc ZS,_QUERY_END
    ior r3,r3,0xff03
    xor r3,r3,0xffff
_QUERY_THREAD_STATUS:
    inp r1, SCUbkpt
    and r1,r1,0xfc
    sub r1,r1,r3
    bc ZS,_THREAD_DONE
    bra _QUERY_THREAD_STATUS
_THREAD_DONE:
// Stop Thread 2 - Thread 7
    mov r3,0xfc
    outp r3, SCUstop
_QUERY_END:

_DECODING_END:
    bra _DECODING_START

stop:
    bra stop

Thread2:  // Decoding processors
    thrd r1
    ld r2,r1,processorID  // r2: processorID

//Calculate pnSymbolMatLLRPosition
    ld r0,r2,pnSymbolMatLLRPosition
    add r0,r0,nCodeC*nBufWidth
    sub r1,r0,nBufLength
    bc NC, replacePnSymbolMatLLRPosition
    bra storePnSymbolMatLLRPosition
replacePnSymbolMatLLRPosition:
    add r0,r1,0
storePnSymbolMatLLRPosition:
    st r0,r2,pnSymbolMatLLRPosition

//Load peTheseLLRs
    ld r1,r2,processorPhase
    ld r3,r1,pnCheckDeg
    mov r5,0
load_peTheseLLRs_Cond:
    sub r1,r5,r3
    bc ZS,load_peTheseLLRs_End
load_peTheseLLRs_Body:
    ld r6,r2,pnCheckDegRowPosition
    add r6,r6,r5
// current LDPC-CC decoder processor's position in pnTheseCols
    ld r4,r2,pnPosition and pnTheseRowsTemp
    add r4,r4,r5
    ld r0,r6,matOnesInPcmYRowPosition
// pnTheseRows
```

```
    ld r1,r2,pnSymbolMatLLRPosition
    add r1,r1,nCodeC*nBufWidth*nProcSize
    sub r1,r1,r0
    sub r0,r1,nBufLength
    bc NC,replacePnTheseRows
    bra storePnTheseRows
replacePnTheseRows:
    add r1,r0,0
storePnTheseRows:
    ld r0,r6,matOnesInPcmX
    add r0,r0,r1
    st r0,r4,pnMatLLRPosition
    ld r0,r0,matLLRBuffer
    st r0,r4,peTheseLLRs
    add r5,r5,1
    bra load_peTheseLLRs_Cond
    load_peTheseLLRs_End:
// End load peTheseLLRs

// Check Node Operation
// calculate two minimum absolute number and the sign
    ld r5,r2,pnPosition
    st r2,r7,0    // r2 is not used in check node
    mov r0,eInitLLR   // r0:the minimum number value
    mov r1,eInitLLR   // r1:the second minimum number value
    mov r4,r3
    st r3,r7,1
    mov r3,0   // r3: sign
    sub r4,r4,1
checkNode_Cond1:
    sub r2,r4,0
    bc NS,checkNode_End1
checkNode_Body1:
    add r2,r5,r4
    ld r2,r2,peTheseLLRs
    bc NS,negative
    bra absolute_end
negative:
    bix r3,r3,15
negative_end:
absolute:
    xor r2,r2,0xffff
    add r2,r2,1
absolute_end:
    sub r6,r1,r2
    bc NS,compare_end
    sub r6,r0,r2
    bc NS,second_min
    add r1,r0,0x0
    add r0,r2,0x0
    bra compare_end
second_min:
    add r1,r2,0x0
compare_end:
    sub r4,r4,1
    bra checkNode_Cond1
checkNode_End1:
    mov r2,r3
    ld r3,r7,1
```

```
// calculate the data based on sign and
// two minimum absolute number
    mov r4,0
checkNode_Cond3:
    sub r6,r4,r3
    bc ZS,checkNode_End3
checkNode_Body3:
    st r3,r7,1
    add r3,r5,r4
    ld r6,r3,peTheseLLRs
    bc NS,absolute3
    sub r6,r6,r0
    bc ZS,minimum_number1
    add r6,r0,0
    bra minimum_number_end1
minimum_number1:
    add r6,r1,0
minimum_number_end1:
    add r2,r2,0 // r2:sign
    bc NS, opposite_1
    bra opposite_1_end
opposite_1:
    xor r6,r6,0xffff
    add r6,r6,1
opposite_1_end:
    bra absolute3_end
absolute3:
    xor r6,r6,0xffff
    add r6,r6,1
    sub r6,r6,r0
    bc ZS,minimum_number
    add r6,r0,0
    bra minimum_number_end
minimum_number:
    add r6,r1,0
minimum_number_end:
    add r2,r2,0
    bc NS,absolute2_end
    xor r6,r6,0xffff
    add r6,r6,1
absolute2_end:
absolute3_end:

    ld r3,r3,pnMatLLRPosition
    st r6,r3,matLLRBuffer  // store back to matLLRBuffer
    ld r3,r7,1 // pop r3
    add r4,r4,1
    bra checkNode_Cond3
checkNode_End3:
    ld r2,r7,0
// Check node operation end

// Variable node
    ld r1,r2,pnSymbolMatLLRPosition
    mov r3,0
variableNode_Cond:
    sub r0,r3,nCodeC
    bc ZS,variableNode_End
    variableNode_Body:
    ld r4,r2,pnSymbolDegRowPosition
```

```
    add r4,r4,r3
    ld r4,r4,pnSymbolDeg
    add r4,r4,1 // pnSymbolDeg[nThisPhase*nCodeC+i]+1
    mov r6,0  // r6, eTotalSum
    mov r5,0 // r5, loop variable
eTotalSum_Add_Cond:
    sub r0,r5,r4
    bc ZS,eTotalSum_Add_End
    add r0,r1,r5
    ld r0,r0,matLLRBuffer
    add r6,r6,r0  // eTotalSum=eTotalSum + ...
    add r5,r5,1
    bra eTotalSum_Add_Cond
eTotalSum_Add_End:

// r6(eTotalSum) is used from previous
    mov r5,1
eTmp_Cond:
    sub r0,r5,r4
    bc ZS,eTmp_End
    add r0,r1,r5
    ld r0,r0,matLLRBuffer
    sub r0,r6,r0
    st r3,r7,2
    sub r3,r0,eInitLLR
    bc NC,greater_than_eInitLLR
    sub r3,r0,-eInitLLR
    bc NS,less_than_minus_eInitLLR
    bra eTmp_compare_end
greater_than_eInitLLR:
    mov r0,eInitLLR
    bra eTmp_compare_end
less_than_minus_eInitLLR:
    mov r0,-eInitLLR
eTmp_compare_end:
    add r3,r1,r5  // store back to matLLRBuffer
    st r0,r3,matLLRBuffer
    ld r3,r7,2
    add r5,r5,1
    bra eTmp_Cond
eTmp_End:
    add r3,r3,1
    add r1,r1,nBufWidth
    bra variableNode_Cond
    variableNode_End:
// End variable node

// (processorPhase++) mod nCodeT
    ld r0,r2,processorPhase
    add r0,r0,1
    sub r1,r0,nCodeT
    bc NC,replaceProcessorPhase
    bra storeProcessorPhase
replaceProcessorPhase:
    add r0,r1,0
storeProcessorPhase:
    st r0,r2,processorPhase

// pnCheckDegRowPosition=(pnCheckDegRowPosition+nCheckDegMax)
// and mod nCodeT*nCheckDegMax
```

```
    ld r0,r2,pnCheckDegRowPosition
    add r0,r0,nCheckDegMax
    sub r1,r0,nCodeT*nCheckDegMax
    bc NC,replaceCheckDegRowPosition
    bra storeCheckDegRowPosition
replaceCheckDegRowPosition:
    add r0,r1,0
storeCheckDegRowPosition:
    st r0,r2,pnCheckDegRowPosition

// pnSymbolDegRowPosition=(pnSymbolDegRowPosition+nCodeC)
// mod nCodeT*nCodeC
    ld r0,r2,pnSymbolDegRowPosition
    add r0,r0,nCodeC
    sub r1,r0,nCodeT*nCodeC
    bc NC,replacePnSymbolDegRowPosition
    bra storePnSymbolDegRowPosition
replacePnSymbolDegRowPosition:
    add r0,r1,0
storePnSymbolDegRowPosition:
    st r0,r2,pnSymbolDegRowPosition

bra @

//-----------------------------------------
// Other Source Files

__AppCode_End__:

//=============================================================================
// Data Space:
//  All Data must be in a separate 2kWord Memory Block from any Code.
//=============================================================================

@ = (@ + 0x800-1) & -0x800   // Round up to the next 2kWord Memory Block
__AppData_Start__:


#include "Long_Data.asm"

__AppData_End__:

//=============================================================================
// Short Address Space:
//  Any Data placed in this space may be accessed with a single word
// instruction.
//=============================================================================

@ = kRAM_End - 127              // Start of the short address space
__ShortData_Start__:

#include "Short_Data.asm"

__ShortData_End__:
```

B.4 LDPC-CC Encoding and Decoding Data Definition

```
// Filename: Short_Data.asm
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Short Data Definition
//
// Date: Jan 24, 2008

i: @=@+1
i1: @=@+1

head: @=@+1
positionTablePointer: @=@+1
checkNum: @=@+1
nThisBlock: @=@+1
pnSymbolDegPointer: @=@+1
threadPattern : @=@+1
nThisBlock_nCheckDegMax: @=@+1
nThisProc: @=@+1
blockRowPosition: @=@+1

procDisp:
0*nProcSize*nCheckDegMax % (nCodeT*nCheckDegMax)
1*nProcSize*nCheckDegMax % (nCodeT*nCheckDegMax)
2*nProcSize*nCheckDegMax % (nCodeT*nCheckDegMax)
3*nProcSize*nCheckDegMax % (nCodeT*nCheckDegMax)
4*nProcSize*nCheckDegMax % (nCodeT*nCheckDegMax)
5*nProcSize*nCheckDegMax % (nCodeT*nCheckDegMax)

matOnesCol: @=@+6
processorID: @=@+8
threadTest: @=@+8
pnCheckDegRowPosition: @=@+nProcNum
pnSymbolMatLLRPosition: @=@+nProcNum

// Filename: Long_Data.asm
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
// Long Data Definition
//
// Date: Jan 24, 2008

T0_SP:  @ = @ + kStackSize
T1_SP:  @ = @ + kStackSize
T2_SP:  @ = @ + kStackSize
T3_SP:  @ = @ + kStackSize
T4_SP:  @ = @ + kStackSize
T5_SP:  @ = @ + kStackSize
T6_SP:  @ = @ + kStackSize
T7_SP:  @ = @ + kStackSize

queue: @=@+257

matrix:
```

```
// The parity check matrix of the benchmark (128,3,6)
// LDPC-CC was provided to us courtesy of Dr. Kamil
// Zigangirov, Department of Electrical Engineering,
// University of Notre Dame, IN 46556, U.S.A

pnCheckDeg:
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6,6
6,6,6,6,6,6,6,6,6

pnSymbolDeg:
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3,3,3
3,3,3,3,3,3,3,3

matOnesInPcmY:
// The parity check matrix of the benchmark (128,3,6)
// LDPC-CC was provided to us courtesy of Dr. Kamil
// Zigangirov, Department of Electrical Engineering,
// University of Notre Dame, IN 46556, U.S.A

matOnesInPcmYRowPosition:
// The parity check matrix of the benchmark (128,3,6)
// LDPC-CC was provided to us courtesy of Dr. Kamil
// Zigangirov, Department of Electrical Engineering,
// University of Notre Dame, IN 46556, U.S.A
```

```
matOnesInPcmX :
// The parity check matrix of the benchmark (128,3,6)
// LDPC-CC was provided to us courtesy of Dr. Kamil
// Zigangirov, Department of Electrical Engineering,
// University of Notre Dame, IN 46556, U.S.A

matLLRBuffer: @=@+nBufLength
pnPosition : @=@+nProcNum
procMemoryStart: @=@+nProcNum
pnTheseRows : @=@+nCheckDegMax*nProcNum
pnTheseRowsTemp : @=@+nCheckDegMax*nProcNum
pnTheseCols: @=@+nCheckDegMax*nProcNum
pnMatLLRPosition : @=@+nCheckDegMax*nProcNum
peTmpLLRs: @=@+nCheckDegMax*nProcNum
peTheseLLRs : @=@+nCheckDegMax*nProcNum
variableNodeTmp :@=@+nCodeC*nBufWidth*nProcNum

variableNodeTmpPosition1 :
0,2,4,6,8,10
variableNodeTmpPosition :
0,4,8,12,16,20,24,28,32,36,40,44

SP_ADDRESS:
T0_SP
T1_SP
T2_SP
T3_SP
T4_SP
T5_SP
T6_SP
T7_SP

THREAD_ADDRESS:

Thread1
Thread2

Start_String: "Start Encoding",0

Start_String_Decoding: "Start Decoding",0

endOfLine: 0x0d,0x0a,0

space: " ",0
zero: "0",0
one: "1",0
```

B.5 LDPC-CC Encoding and Decoding Constant Definition

```
// Filename: Constants.h
// Author: Xin Sheng Zhou
// Department of Electrical and Computer Engineering
// University of Alberta
//
// Description:
```

B.5: LDPC-CC Encoding and Decoding Constant Definition

```
// Constant Definition
//
// Date: Jan 24, 2008

#define kStackSize  64
#define kSPI0CS_Semaphore    kHardwareSemaphore0
#define kDevLEDs_Semaphore   kHardwareSemaphore2

#define encoderQueueSize 257
#define checkDegree 6
#define phaseNum 129
#define positionTableSize phaseNum*checkDegree

#define nCodeM 128
#define nInfo 1
#define nCodeC 2
#define nSymbolDegMax 3
#define nCodeT 129
#define nCheckDegMax 6
#define nProcSize nCodeT
#define nBufWidth nSymbolDegMax+1
#define nProcNum 6
#define decoderThreadNum 6 // This is the total threads used for decoder
#define nBufHeight nCodeC*(nProcNum*nProcSize+2)
#define nBufLength nBufWidth*nBufHeight
#define nBlockLength nBufWidth*nCodeC
#define nEachProcBufLength nCodeC*nBufWidth*nProcSize
#define fraction 3
#define eInitLLR 1000 << fraction
```