Multi-Threading and Hardware Acceleration with Low-Density Parity-Check Decoding

Samer Chomery,
Dept. of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada T6G 2V4
Email: schomery@ualberta.ca

***Abstract*—This project presents a computer architecture study on the effects of multithreading on low-density parity-check decoding. Hardware acceleration techniques are also considered to increase the performance of LDPC decoders. The multithreading simulation is aimed at the XInC microprocessor architecture that features a RISC architecture with 8 independent hardware threads. This paper presents background information, related work conducted in the LDPC decoding area that inspired this study, and a brief description of the architecture experiment and its procedures. The paper concludes with a summary of the results and discussion. The study showed that speedup gains of up to 9.563 times could potentially be achieved.**

***Index Terms*—Low-density parity-check codes, belief propagation algorithm, multithreading, hyper threading, hardware accelerator, LDPC decoder.**

I. Introduction

FORWARD error correction (FEC) codes are a technique that improves the reliability of information transmission over a noisy channel. FEC codes do not guarantee errorless transmission, however they attempt to reduce the probability of error to as small as desired. Low-density parity-check (LDPC) codes are a member of the FEC family, and they allow data transmission rates close to the theoretical maximum. Due to the impracticality of their implementations at the time they were introduced in the 60s, they were forgotten for years. The recent advancements achieved in the fields of information theory and VLSI design resulted in the resurgence of LDPC codes. Due to their superior performance, they are adopted for several communications standards and are being considered for next generation ones. Most of the current efforts in the LDPC field focus on low complexity implementations for their encoders and decoders.

Various LDPC decoder architectures have been proposed in literature, and some have been implemented and demonstrated in hardware. These architectures range from fully serial to fully parallel. It is viable to consider multithreading as an enhancement strategy for current LDPC decoder designs. Multithreading is not a new concept in computer architecture. It has evolved and changed from simple symmetric multiprocessing to hyper threading.

Motivated by the importance of LDPC decoders and the viability of multithreaded architectures, I introduce an architecture experiment to study the gains achieved from enhancing LDPC decoders with multithreading. I also propose a further enhancement step that features a hardware accelerator idea aiming at speeding internal computationally heavy loops.

I begin this paper by presenting background information of LDPC decoders and multithreading in Section II. The experiment's concepts are explained in Section III, and the procedures and challenges encountered are listed in Section IV. The experiment was conducted and the results were collected and presented with a brief discussion in Section V. I conclude with comments regarding the impact of multithreading on LDPC decoders and other similar architectures in Section VI

II. Background

To properly present the main architecture experiment in this paper, an introduction to the main two concepts involved is essential. These two concepts are LDPC decoding and processor multithreading.

A. LDPC Decoding

Low-Density Parity-Check Codes have become widely popular since their re-discovery in the 90s. Their current success and research focus is a result of their strong performance and practical implementation nature. LDPC codes can achieve near Shannon Limit performance. In fact it has been shown that they can get to within 0.0045 dB of this limit [1]. In addition to their strong performance capability, LDPC codes could be implemented in hardware using reduced complexity encoders and decoders. While the LDPC encoder is if $O(n^2)$ complexity, a great deal of research has been dedicated to reduce that. Some O(n) complexity encoder designs have been presented. The decoding algorithm used for LDPC codes is called the Belief Propagation (BP) algorithm and is of O(n) complexity.

Before I discuss the BP algorithm, I will present a few theoretical concepts that explain the idea of the LDPC codes.

LDPC codes are generated using a sparse parity-check matrix (*H*). The *H* matrix is largely populated by zeros, while the number of ones approaches zero. As the length of the matrix is increased the density of ones decrease, hence the low-density part of the name. It has been shown that decreasing the density of the ones leads to the increased performance of the codes.

The set of LDPC codes *C* is the set of vectors *x* in the right null-space of *H* (i.e. *H x = 0*). By manipulating the *H* matrix, a generating matrix *G* can be obtained and it satisfies (1)

$$GH^T = 0 \tag{1}$$

The *G* matrix is used to encode the user's data through matrix multiplication to produce the encoded stream. This is done by using (2)

$$x^T = u^T G \tag{2}$$

The matrix multiplication in (2) is the reason for the quadratic complexity of the encoding.

Another concept relevant and extremely instrumental to the LDPC codes is the bipartite or Tanner graph. This is a method used to visually represent the LDPC codes and their operation. In this graph, two subsets of nodes are present; *n* variable nodes corresponding to the length of the code, and *m* parity check nodes corresponding to the code bits, as shown in Fig 1. While there are no edges directly connecting elements of each subset, edges exist between the two subsets and they correspond to the ones in the parity check matrix *H*. From the graph, the concept of LDPC codes is visually realized. The variable nodes participating in the same parity-check operation are all connected to one check node, and each variable node is participating in several parity check operations, hence connected to several check nodes.

It is the parity-check redundancy that will allow the decoding of the original signal with a high level of error correction accuracy relative to conventional coding schemes.

Arriving at the decoder, the original encoded message is corrupted by channel noise. We define a parameter in (3) on which the decoding algorithm will heavily depend; the log-likelihood-ratio (LLR).

$$\mathrm{LLR}(x_r) = \log\left(\frac{P(x_r = 1 \mid (\text{received bit}))}{P(x_r = 0 \mid (\text{received bit}))}\right) \tag{3}$$

Where x_r represents a decoded bit in the received codeword. Based on the LLR parameter, the receiver decides on the value of the received bits with a certain degree of confidence.

If the algorithm stops here, there would be no gain in using the parity-check relations to improve the LLR values (level of confidence) of the received bits. Therefore an iterative mechanism must be introduced to help improve the level of belief on these LLR values. A special type of message-passing algorithm does just that through the propagation of the LLR (belief) values among the connected nodes of the Tanner graph discussed above. This explains the name Belief Propagation (BP) Algorithm. This decoding algorithm is an iterative algorithm that uses the following steps.

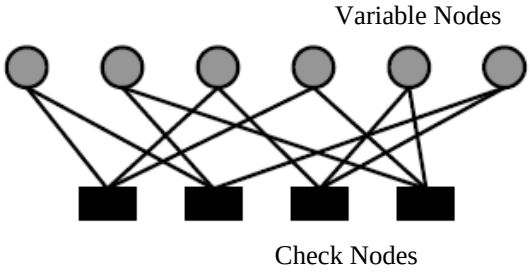

Fig. 1. The Tanner graph representing a parity-check matrix

- Initialization; the initial LLR values are read from the channel and sent to the corresponding check nodes.
- First half of iteration; the messages of the check nodes are computed using (4) and passed to the appropriate variable node.

$$CNode\_Message = 2\tanh^{-1}\left(\prod_i\left(\frac{LLR_i}{2}\right)\right) \tag{4}$$

  Where LLR_i represent the LLR values from the participating variable nodes.
- Second half of iteration; the messages of the variable nodes are computed using (5) and the arriving values from the check nodes. Then they are passed back to the appropriate check nodes.

$$VNode\_Message = \sum_j CNode\_Message_j \tag{5}$$

- Slice the LLR values of the bits using the zero threshold and compute a codeword.
- If the codeword satisfies the null space constraint, or if the algorithm reached the maximum number of iterations, then stop the algorithm.

After a few iterations, the LLR values will improve significantly and will eventually converge to their correct values.

B. Multithreading

Although faster clock speeds are an important method of delivering higher performance from microprocessors, clock frequency is not the only performance-improving factor. Another method is to attempt to perform more workload per clock cycle. Many techniques in computer architecture have tackled this idea and great progress has been made. Pipelining and instruction level parallelism (ILP) are such techniques, which in their essence mean using as much as possible of the processor's resources to perform more work during a fixed period of time, the clock period. Multithreading is another technique.

In Multithreading, multiple programs are loaded into memory and the processor, together with the operating system (OS), provide the illusion that all programs are running at the same time. While the processor can only execute one program at a time, the OS rapidly switches between running programs after a time interval called the time slice. Some programs might have longer time slices than others depending on their priority. In what is known as pre-emptive multi-tasking, the OS forces every program out of the processor's execution as soon as its time slice expires. This is done through utilizing the Interrupt Service Routine (ISR) so that when the program's

time slice expires an interrupt is generated and the interrupt handler will handle switching the processor between the tasks.

Before introducing the evolution of multi-threading, it is important to define several terms. A running program in the processor could also be termed a process. Each process has a context, which essentially refers to all the information about the state of that process in time. A process is comprised of threads. Each process has at least one main thread, and each thread has its own local context. This idea is the main reason why multi-threading works. Since each thread has its own context (local information), the processor can operate on it reasonably independently, thus creating the opportunity to maximize the processor's usage time. While one process is idle or waiting for a certain resource, the processor can go ahead and execute a non-related thread, hence efficiently increasing the performance.

Multithreading passed through several stages:

1) Symmetric multiprocessing (SMP): in conventional SMP, a second CPU is added, and this results in the ability to execute two processes at the same time, yielding less context switching and more execution time for each process.
2) Superthreading: Also known as time-slice multithreading. A superthreaded processor is capable of running multiple threads at the same time. In these processors, several instructions are issued at the same time, and they enter the several available functional units in the internal processor pipelines. All instructions must come from the same thread; therefore each thread is still bounded by a time slice. However, the ability to execute multiple instructions and performing less OS context switching (replaced by logic switching) increases the processors performance.
3) Hyper-Threading: Hyper-threading is similar to Superthreading, but instructions fetched can arrive from multiple threads. This allows the scheduling logic to flexibly use any available execution functional units and time slots. Therefore, it makes more efficient use of the processor's resources and further increases the performance.

II. Related work

The area of LDPC decoding hardware architectures is rich with contributions. Howland and Blanksby introduced their LDPC hardware decoder [2] that needed 690-mW to perform at 1-Gb/s with a rate 1/2 code. They proposed several architectures for the BP algorithm. The first featured the hardware-sharing concept. This architecture involved building small hardware functional units to handle the check-node and variable-node operations and a memory fabric to connect them and store the messages passed in between. However, the disadvantage of such approach is the fact that these functional units will be reused and the memory fabric will be accessed many times to perform the iterations of the algorithm. This reduces the potential throughput of the system, sine the iterations are performed serially. In their paper, they also proposed a parallel architecture for LDPC decoding. This architecture stems from the way the BP algorithm corresponds to the Tanner graph. The graph and its components (check and variable nodes) are instantiated in hardware, which means the check node and variable node functional units are implemented in hardware and wires route the connections between them. This design also means that, during each iteration, the functional units will be used once and the messages will propagate through the wires. This architecture allows all the nodes of one type to be updated in parallel. While having an extremely high throughput, this architecture consumes a large area and is hardware expensive.

Karkooti and Cvallaro [3], studied the differences between a fully serial LDPC decoder architecture and a fully parallel LDPC decoder architecture. The fully parallel LDPC architecture features speed and large area, while the fully serial one features small area and slower speed. In fact a fully serial architecture requires only one check-node functional unit and one variable-node functional unit, and they could be reused for all iterations. The fully parallel architecture, in comparison, requires no memory as all message values could be latched close to the functional units. In a tradeoff design, their paper introduced a semi-parallel architecture that balanced the gains between area and time.

The third related decoder architecture surveyed, was the one proposed by Swamy, Bates, and Brandon [4]. While their design is geared mainly towards ASIC implementations it provides a very fundamental idea to the experiment of my paper. They proposed a processor-based decoder architecture. To this end, a processor is a physical entity that contains delay elements and a check-node functional unit as well as a variable node functional unit. Each processor represents a signle iteration and could be cascaded with other processors in an ASIC implementation to construct the BP decoder.

III. The experiment

The architecture study performed in the paper is aimed at two architecture enhancement ideas, multithreading and hardware acceleration. This section presents the way I propose to study these ideas and their effect on LDPC decoding.

A. Multithreading and LDPC decoding

The idea of this architecture experiment was inspired from the processor-based LDPC decoder architecture described in the previous section. The processor components, as defined in [4], could be considered as processes following the definition we established earlier. Therefore it is possible to consider the multithreading approach and its effects on such processes.

A software version of the LDPC decoder provides a practical environment to experiment with multithreading. Attempting to speed up the decoder's program (process) through splitting the decoder's iterations into separate threads is the main goal for this architecture study. A comparison between a single-thread, serial architecture decoder and a

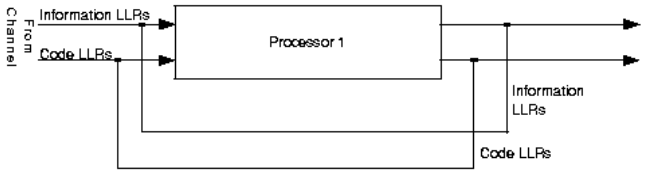

Fig. 2. The single-threaded LDPC decoder architecture. Figure modified from "Low-Density Parity-Check Convolutional Codes," http://www.ece.ualberta.ca/~sbates/LdpcWeb/circuits.html.

multithreaded serial but pipelined architecture decoder is conducted and the results are presented in Section V.

The first single-thread architecture, as shown in Fig. 2, employs a single process that receives the channel inputs and re-routes the results back into the input ports each iteration. In software, this architecture represents a for-loop structure that accesses memory inputs and stores the result back in memory during every iteration. The single-thread architecture will run the same number of instructions and will have the same number of memory accesses per iteration.

The multithreaded architecture, featured in Fig. 3, uses several threads to implement several iterations of the decoder process. Each processor (thread) takes the input of the previous processor or the channel in case of the first processor. The processors run the same instruction sequence and produce their outputs relatively at the same time. In software, this architecture corresponds to an unrolled loop with each iteration executing whenever the required functional units and memory inputs are available. This corresponds to an out-of-order execution therefore a handshaking mechanism needs to be implemented in each thread to ensure the sequential flow of data. Furthermore, this architecture is pipelined since after the last thread has received its valid inputs, a new set of outputs would be produced with each following iteration period.

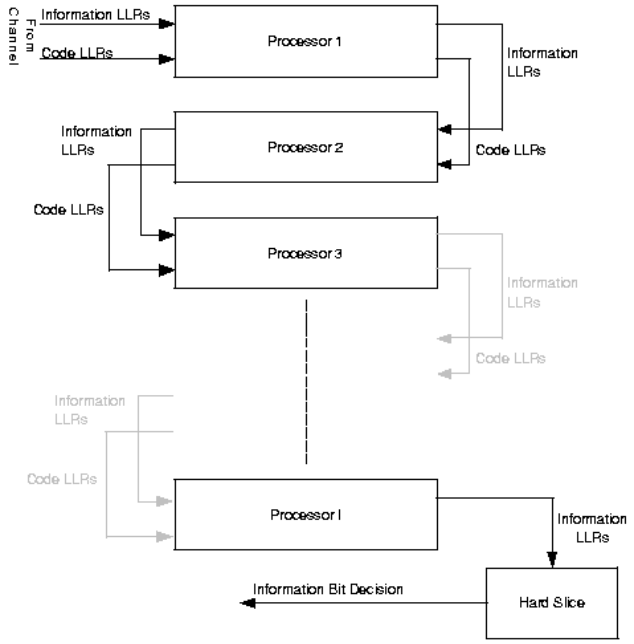

Fig. 3. The multithreaded LDPC decoder architecture. . Figure from "Low-Density Parity-Check Convolutional Codes," http://www.ece.ualberta.ca/~sbates/LdpcWeb/circuits.html.

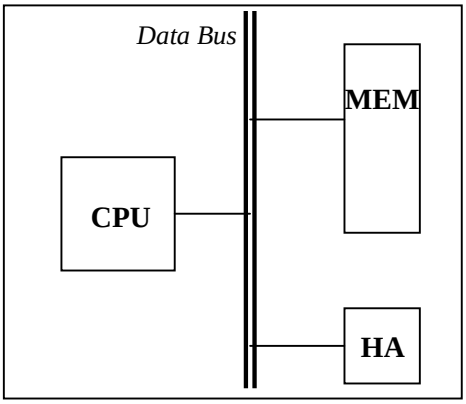

Fig. 4. Diagram of hardware accelerator and data bus organization

B. Hardware acceleration and the check nodes

Looking closely into the BP algorithm for decoding LDPC codes, it is apparent that the most computationally expensive operation happens in calculating the messages of the check nodes. However it is possible to approximate the $\tanh^{-1}$ equation at the check node [5] using the sign-min approach as shown in (6)

$$CNode\_Message = sign\{LLR_0 ....LLR_i\} \cdot \min\{LLR_0 ....LLR_i\} \quad (6)$$

Even after using the sign-min approximation, the check node operation remains to be the limiting stage of the BP algorithm. The main difference is that the sign-min representation is easier to implement in hardware.

In Software, the check node operations are performed through a subroutine that loads the required inputs from memory and stores the results back in memory after finishing its set of instructions. In an attempt to further speed up the decoder main loop, the check node operation is replaced with a Hardware Accelerator (HA) component that aims at reducing the check node's fraction of the workload. The dedicated HA is assumed to eliminate the check node's work cycles and complete the job in one clock cycle. A sketch of such organization is illustrated in Fig 4. The HA is attached to the main data bus that is used to access memory, and has the ability to read the data on the load lines.

IV. EXPIREMINT PROCEDURES AND TOOLS

The architecture experiment described in the previous section was conducted as follows:

1) A C-code implementation of the belief propagation algorithm discussed in [4] was used to simulate the behaviour of the decoder architecture. The code, attached in Appendix A, contains the decoder function and its check-node and variable-node subroutines.
2) The targeted multithreaded hardware device is the XInC micro processor [6] produced by Eleven

Engineering Inc. this processor is a 16-bit RISC processor with eight threads. The threads operate with independent execution of each other having access to the main memory and the peripheral bus. This is a main difference with typical interrupt-driven serial multithreaded processors, allowing for elimination of ISR handling, context switching, and real time operating system (RTOS) overhead. The 8 hardware threads are scheduled to run at 1/8 of the system clock.

3) The main decoder loop was identified within the provided C-code and was converted into the XInC assembly code using Eleven Engineering's C compiler. A few challenges were encountered during this process. The C compiler experienced difficulty interpreting C-code that used pointers and Array indexing memory accesses. Therefore the Decoder's C-Code was re-written to compensate for this incapability. Dummy variables were used to evaluate the indexing and pointer expressions and were accompanied with the memory access. For example:

   x = A[*exp*]; was changed to temp = *exp*; x = A;

   This transformation resulted in an equivalent code in terms of workload but incorrect operation. Furthermore, storing the result in temp, which is on the stack according to the XInC compiler, required subtracting a store instruction every time such transformation was done. Another issue was regarding global variables and constants; these needed re-declaration in every function resulting in incorrect extra instructions that were also subtracted from the total. The transformed C-code appears in Appendix B.
4) The C-code implementing the check node and variable node functions was transformed into the format suitable for the compiler. The code is attached in Appendix C.
5) The multithreading architecture required handshaking code on each thread to ensure the sequential flow of data. The C-code to perform these tasks was written and prepared for compilation, as shown in Appendix C.
6) Upon preparing the C-code, the compiler was utilized to generate the XInC assembly code corresponding to the decoder C-code.
7) The generated assembly code was profiled and the instructions were counted. In order to count the instructions, the generated code was moved into an Excel sheet where each line occupied a row and the number of empty lines was subtracted from the total number of lines to yield the number of instructions. A sample of generated XInC assembly code is attached in Appendix D.

The next section presents the results of the experiment and an analysis of the proposed architectures.

V. Results and Discussion

The real performance comparison metric in computer architecture experiments is the processor's execution time (CPU_{Time}). Which is defined by (7).

$$CPU_{Time} = CPI \times IC \times Cycle\_Time \quad (7)$$

Where *CPI* is Cycles per instruction, *IC* is Instruction count, and *Cycle_Time* is the system clock period [7].

For the purposes of this experiment, all instructions are assumed to have equal average *CPI,* Thus making *CPI* a constant in the equation. *Cycle_Time* does not vary within the code either. Therefore *IC* can loosely approximate CPU_{Time}. This is not the case in real applications where various instructions have various *CPI* and timing properties.

Based on the above, the various code segments were counted and the instruction counts are presented in Table I.

TABLE I
INSTRUCTION COUNTS FOR THE MAJOR CODE SEMENTS

Code Segment	Instruction Count
Full main loop	565
Main loop without for-loop instructions	551
Check node	379
Variable node	380
Sign2two function	27
Two2sign function	22
Full variable node	429
Inter-thread handshaking code	25

The values in Table I above were used in this experiment to conduct a comparison between four architectures.

- Single Thread; this architecture ran the main decoder for-loop and all its subroutines. This architecture is considered the base case.
- Single Thread with Hardware Accelerator; this architecture ran the main decoder for-loop without the check node code (eliminated by HA).
- Multithreaded; this architecture ran the unrolled main decoder code without the for-loop instructions. It ran all the subroutines, as well as the handshaking code.
- Multithreaded with Hardware Accelerator; this architecture ran the unrolled main decoder code without the for-loop instructions or the check node code. It also ran the handshaking code.

Table II provides a summary of instruction counts for the above-mentioned architectures and their speedup relative to the bases case.

TABLE II
ARCHITECTURE INSTRUCTION COUNTS AND SPEEDUP FACTORS

Architecture	Inst. Count	Speedup
Single Thread	9611	Base case
ST & HA	6958	1.381
Multi-Thread	1384	6.944
MT & HA	1005	9.563

The first two architectures iteration instruction counts were multiplied by a factor of 7 to simulate the loop iterating seven times on a single thread.

The two multithreaded architectures did not count the for-loop instructions. The inter-thread handshaking instructions were counted and represented 1.81% of the multithreaded architecture's instructions, and 2.49% of the multithreaded with HA one. This small overhead ration was the reason for not achieving a perfect 7x speedup for the multithreaded architecture. Each of these multithreaded architectures is assumed to spur threads running on seven of XInC's hardware threads. The eighth thread is assumed to be performing the master state machine's tasks controlling input/output functions and other data flow tasks.

From the results obtained above, it is apparent that having independent, yet communicating, hardware threads in a pipelined architecture resulted in a 6.944 times speedup over the single-thread sequential decoder. This speedup would be further decreased in real life operation due to various other overhead instructions and resource stalls.

The Hardware Accelerator component eliminated the slow running code of the check node to post a 1.381 times speedup over the base case. Combined gains from both enhancements resulted in a 9.563 times speedup.

Multithreading is a clever computer architecture technique that can provide valuable gains when implemented with real life applications, especially data processing applications such as the LDPC decoder design.

VI. Conclusion

In this paper I presented the background information on low-density parity-check decoders and their architectures, as well as a brief introduction to multithreading. The potential speedup gains that results from combining LDPC decoders with multithreading architectures were discussed. A processor-based LDPC decoder was integrated with multithreading and hardware acceleration to achieve up to 9.563 times speedup.

From the results and analysis, it was determined that the XInC microprocessor provides a commendable environment for LDPC decoder multithreading. Despite incurring some multithreading overhead, the speedup gains considerably out weighed the losses.

While most current LDPC decoding implementations are FPGA and ASIC designs, multithreading creates significant speedup opportunities, and is worth further exploring in real hardware implementations of LDPC decoders and other DSP applications.

References

[1] S.Y. Chung, G. D. Forney, T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Commun. Lett.*, vol. 5, pp. 58–60, Feb. 2001.

[2] C. J. Howland and A. J. Blanksby, "A 690-mW 1-Gb/s 1024-b, rate 1/2 low density parity check decoder," Solid-State Circuits, IEEE Transactions on, vol. 37, no. 3, pp. 404–412, March 2002.

[3] M. Karkooti, J. R. Cavallaro. "Semi-Parallel Reconfigurable Architectures for Real-Time LDPC Decoding," itcc, p. 579, International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 1, 2004.

[4] R. Swamy, S. Bates and T. L. Brandon, "Architectures for ASIC Implementations of Low-Density Parity-Check Convolutional Encoders and Decoders", Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Kobe, Japan, 2005.

[5] G. Richter, G. Schmidt, M. Bossert and E. Costa, "Optimization of a Reduced-Complexity Decoding Algorithm for LDPC Codes by Density Evolution", International Conference on Communication (ICC), Seoul, Korea, May 2005.

[6] Eleven Engineering Inc. XInC Datasheet, 2005

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

Acknowledgement

I would like to acknowledge Ramkrishna Swamy for providing the LDPC decoder C-code package for this project.

Appendix A

```
// decoder.h
//
// Author: Tyler Brandon, with help from Zhengeng Chen and Ramkrishna Swamy
//
// Created: Sept 21, 2005
//
// Desc: fast and simple
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "node.h"

// Global variables.
extern int numPhases;              // Number of phases (typically 129).
extern int numRows;                // Number of rows (typically 258).
extern int codeRate;       // Core rate (typically 2).
extern int maxDegree;

extern int magBits;        // Number of magnitude bits (typically 6).
extern int magBitMask;             // Magnitude bit mask (0011111).
extern int signBitMask;            // Sign bit mask      (1000000).
extern int satBitMask;     // Saturation bit mask(0100000).

int phaseDecoder;
int numProcessors;
int procDelta;
int procDeltaAddr;
int *decMem;
int **decMemPtr;


int decoder_init( int *pcmLLRs, int procs ){
   int i,j,p;

   // set the number of processors
   numProcessors = procs;
   procDelta = numRows * 4;
   procDeltaAddr = numPhases * maxDegree;

   // lets start the Decoder at phase 0
   phaseDecoder = 0;

   // allocate the memory for the decoder
   decMem = (int *)malloc( numProcessors * numRows * 4 * sizeof(int) );
   if( decMem == NULL )
         fprintf( stderr, "Failed to allocated LLR memory for %d processors\n", numProcessors );
   for( i=0; i<(numProcessors*numRows*4); i++ )
         decMem[i] = satBitMask;

   // allocate the memory for the decoder memory pointer (based on phase)
   decMemPtr = (int **)malloc( numProcessors * numPhases * maxDegree * sizeof(int *) );

   // for each phase determine the LLR accessed and point to the decoder memory position
   // that holds that LLR
   for( p=0; p<numProcessors; p++ ){
```

```
        for( i=0; i<numPhases; i++ ){
            for( j=0; j<maxDegree; j++ ){
                decMemPtr[p*procDeltaAddr + i*maxDegree+j] = &decMem[p*procDelta +
        pcmLLRs[i*maxDegree+j]];
            }
        }
    }
    return( 1 );
}

void decoder( int *info, int *code ){
    int i, p, row, rowv, sum;
    int i0, i1, i2, i3, c0, c1, c2, c3;

    // update the variable node rows with the new code and info values
    row = -codeRate*phaseDecoder;
    if( row < 0 )
        row += numRows;
    rowv = -codeRate - codeRate*phaseDecoder;
    while( rowv < 0 )
        rowv += numRows;

    for( p=(numProcessors-1); p>=0; p-- ){

        // Grap the inputs to this processor from the previous processor
        if( p != 0 ){
            decMem[p*procDelta + row] =                 decMem[(p-1)*procDelta + row];
            decMem[p*procDelta + row+1*numRows] =       decMem[(p-1)*procDelta +
        row+1*numRows];
            decMem[p*procDelta + row+2*numRows] =       decMem[(p-1)*procDelta +
        row+2*numRows];
            decMem[p*procDelta + row+3*numRows] =       decMem[(p-1)*procDelta +
        row+3*numRows];
            decMem[p*procDelta + (row+1)] =             decMem[(p-1)*procDelta + (row+1)];
            decMem[p*procDelta + (row+1)+1*numRows] =   decMem[(p-1)*procDelta +
        (row+1)+1*numRows];
            decMem[p*procDelta + (row+1)+2*numRows] =   decMem[(p-1)*procDelta +
        (row+1)+2*numRows];
            decMem[p*procDelta + (row+1)+3*numRows] =   decMem[(p-1)*procDelta +
        (row+1)+3*numRows];
        }else{ // the first processors, get the inputs from the channel
        decMem[row] = *code;
        decMem[row+1*numRows] = *code;
        decMem[row+2*numRows] = *code;
        decMem[row+3*numRows] = *code;
        decMem[(row+1)] = *info;
        decMem[(row+1)+1*numRows] = *info;
        decMem[(row+1)+2*numRows] = *info;
        decMem[(row+1)+3*numRows] = *info;
        }

      // check node
        cNodeOp( decMemPtr[p*procDeltaAddr + phaseDecoder*maxDegree],
            decMemPtr[p*procDeltaAddr + phaseDecoder*maxDegree+1],
            decMemPtr[p*procDeltaAddr + phaseDecoder*maxDegree+2],
            decMemPtr[p*procDeltaAddr + phaseDecoder*maxDegree+3],
            decMemPtr[p*procDeltaAddr + phaseDecoder*maxDegree+4],
            decMemPtr[p*procDeltaAddr + phaseDecoder*maxDegree+5] );

      // variable node
        vNodeOp( &decMem[p*procDelta + rowv],
            &decMem[p*procDelta + rowv+1*numRows],
            &decMem[p*procDelta + rowv+2*numRows],
            &decMem[p*procDelta + rowv+3*numRows],
            &decMem[p*procDelta + (rowv+1)],
            &decMem[p*procDelta + (rowv+1)+1*numRows],
            &decMem[p*procDelta + (rowv+1)+2*numRows],
            &decMem[p*procDelta + (rowv+1)+3*numRows] );

        // If we are at the last processors, capture the variable node outputs
      if( p == (numProcessors-1) ){
            c0 = decMem[p*procDelta + rowv];
            c1 = decMem[p*procDelta + rowv+1*numRows];
            c2 = decMem[p*procDelta + rowv+2*numRows];
            c3 = decMem[p*procDelta + rowv+3*numRows];
            i0 = decMem[p*procDelta + (rowv+1)];
            i1 = decMem[p*procDelta + (rowv+1)+1*numRows];
            i2 = decMem[p*procDelta + (rowv+1)+2*numRows];
            i3 = decMem[p*procDelta + (rowv+1)+3*numRows];
        }

    }

    // slice the last processor's output
    signMag2TwosComp( &i0 );
    signMag2TwosComp( &i1 );
    signMag2TwosComp( &i2 );
    signMag2TwosComp( &i3 );
    sum = i0 + i1 + i2 + i3;
    // less than zero its a 1, zero or more its a 0.
    if( sum < 0 )
        *info = 1;
    else
        *info = 0;

    // update the phase
    phaseDecoder++;
    if( phaseDecoder == numPhases )
        phaseDecoder = 0;
}
```

```
// node.h
//
// Author: Tyler Brandon
//
// Created: Aug 24, 2005
//
// Desc:  Fast and simple
//
#include <stdio.h>
#include <malloc.h>

extern int signBitMask;
extern int magBitMask;
extern int satBitMask;
extern int numRows;
extern int phase;

void signMag2TwosComp( int *val ){
    int sign;
    sign = *val & signBitMask;
    *val = *val & (magBitMask | satBitMask);
    if( sign )
        *val = -(*val);
}

void twosComp2SignMag( int *val ){
    if( *val < 0 )
        *val = (-(*val)) | signBitMask;
}

void cNodeOp( int *l0, int *l1, int *l2, int *l3, int *l4, int *l5 ){
    int min = 10000;
    int min2 = 10000;
    int mag0, mag1, mag2, mag3, mag4, mag5;
    int sign, sign0, sign1, sign2, sign3, sign4, sign5;
    int sat0, sat1, sat2, sat3, sat4, sat5;
    int *magPtr;

    sat0=sat1=sat2=sat3=sat4=sat5=0;

//    printf("phase %d, cnode before %x, %x, %x, %x, %x, %x\n", phase, *l0, *l1, *l2, *l3, *l4, *l5 );
    //printf("%02x %02x %02x %02x %02x %02x ", *l0, *l1, *l2, *l3, *l4, *l5 );

    // Obtain the sign bits
    sign0 = *l0 & signBitMask;
    sign1 = *l1 & signBitMask;
    sign2 = *l2 & signBitMask;
    sign3 = *l3 & signBitMask;
    sign4 = *l4 & signBitMask;
    sign5 = *l5 & signBitMask;

    // Obtain the magnitudes
    mag0 = *l0 & (magBitMask | satBitMask);
    mag1 = *l1 & (magBitMask | satBitMask);
    mag2 = *l2 & (magBitMask | satBitMask);
    mag3 = *l3 & (magBitMask | satBitMask);
    mag4 = *l4 & (magBitMask | satBitMask);
    mag5 = *l5 & (magBitMask | satBitMask);

    /*
    // check for saturation values
    if( *l0 == signBitMask ){
        mag0 = signBitMask;
        sign0 = 0;
    }
    if( *l1 == signBitMask ){
        mag1 = signBitMask;
        sign1 = 0;
    }
    if( *l2 == signBitMask ){
        mag2 = signBitMask;
        sign2 = 0;
    }
    if( *l3 == signBitMask ){
        mag3 = signBitMask;
        sign3 = 0;
    }
    if( *l4 == signBitMask ){
        mag4 = signBitMask;
        sign4 = 0;
    }
    if( *l5 == signBitMask ){
        mag5 = signBitMask;
        sign5 = 0;
    }
    */

    //printf("magBitMask %d\n", magBitMask );
    //printf("cnode mag %d, %d, %d, %d, %d, %d\n", mag0, mag1, mag2, mag3, mag4, mag5 );
```

```
//printf("signBitMask %d\n", signBitMask );
//printf("cnode sign %d, %d, %d, %d, %d, %d\n", sign0, sign1, sign2, sign3, sign4, sign5 );

// Calculate the sign.
sign = sign0 ^ sign1 ^ sign2 ^ sign3 ^ sign4 ^ sign5;

sign0 = sign0 ^ sign;
sign1 = sign1 ^ sign;
sign2 = sign2 ^ sign;
sign3 = sign3 ^ sign;
sign4 = sign4 ^ sign;
sign5 = sign5 ^ sign;

// assign the sign bits
*l0 = sign0;
*l1 = sign1;
*l2 = sign2;
*l3 = sign3;
*l4 = sign4;
*l5 = sign5;

// Find the minimum and 2nd minimum.
if( mag0 < min ){
        min2 = min;
        min = mag0;
        magPtr = &mag0;
}else if( mag0 < min2 ){
        min2 = mag0;
}
if( mag1 < min ){
        min2 = min;
        min = mag1;
        magPtr = &mag1;
}else if( mag1 < min2 ){
        min2 = mag1;
}
if( mag2 < min ){
        min2 = min;
        min = mag2;
        magPtr = &mag2;
}else if( mag2 < min2 ){
        min2 = mag2;
}
if( mag3 < min ){
        min2 = min;
        min = mag3;
        magPtr = &mag3;
}else if( mag3 < min2 ){
        min2 = mag3;
}
if( mag4 < min ){
        min2 = min;
        min = mag4;
        magPtr = &mag4;
}else if( mag4 < min2 ){
        min2 = mag4;
}
if( mag5 < min ){
        min2 = min;
        min = mag5;
        magPtr = &mag5;
}else if( mag5 < min2 ){
        min2 = mag5;
}

// Assign the minimum and 2nd minimum back to the magnitudes
mag0 = min;
mag1 = min;
mag2 = min;
mag3 = min;
mag4 = min;
mag5 = min;

if( magPtr == &mag0 )
        mag0 = min2;
if( magPtr == &mag1 )
        mag1 = min2;
if( magPtr == &mag2 )
        mag2 = min2;
if( magPtr == &mag3 )
        mag3 = min2;
if( magPtr == &mag4 )
        mag4 = min2;
if( magPtr == &mag5 )
        mag5 = min2;


// asign the magnitudes with the possibility of changing a sign bit to 1 (to represent saturation)
*l0 |= mag0;
*l1 |= mag1;
*l2 |= mag2;
*l3 |= mag3;
*l4 |= mag4;
*l5 |= mag5;

    // convert back to twos complement
//    signMag2TwosComp( l0 );
//    signMag2TwosComp( l1 );
//    signMag2TwosComp( l2 );
//    signMag2TwosComp( l3 );
//    signMag2TwosComp( l4 );
//    signMag2TwosComp( l5 );


//    printf("phase %d, cnode after %x, %x, %x, %x, %x, %x\n\n", phase, *l0, *l1, *l2, *l3, *l4, *l5 );
    //printf("%02x %02x %02x %02x %02x %02x\n", *l0, *l1, *l2, *l3, *l4, *l5 );
}

void vNodeOp( int *c0, int *c1, int *c2, int *c3, int *i0, int *i1, int *i2, int *i3 ){
    int a, b, c, d;
    int asat, bsat, csat, dsat;
    int x0, x1, x2, x3, y0, y1, y2, y3;
    int c0sat, c1sat, c2sat, c3sat, i0sat, i1sat, i2sat, i3sat;

    c0sat=c1sat=c2sat=c3sat=i0sat=i1sat=i2sat=i3sat=0;
    asat=bsat=csat=dsat=0;

    //printf( "%02x %02x %02x %02x %02x %02x %02x %02x ", *c0,*c1,*c2,*c3,*i0,*i1,*i2,*i3 );
    // check to see if we have a saturated value
    /*
    if( *c0 == signBitMask )
        c0sat = 1;
    if( *c1 == signBitMask )
        c1sat = 1;
    if( *c2 == signBitMask )
        c2sat = 1;
    if( *c3 == signBitMask )
        c3sat = 1;

    if( *i0 == signBitMask )
        i0sat = 1;
    if( *i1 == signBitMask )
        i1sat = 1;
    if( *i2 == signBitMask )
        i2sat = 1;
    if( *i3 == signBitMask )
        i3sat = 1;
    */
    c0sat = *c0 & satBitMask;
    c1sat = *c1 & satBitMask;
    c2sat = *c2 & satBitMask;
    c3sat = *c3 & satBitMask;

    i0sat = *i0 & satBitMask;
    i1sat = *i1 & satBitMask;
    i2sat = *i2 & satBitMask;
    i3sat = *i3 & satBitMask;

//    printf("phase %d, vnode before, code %x %x %x %x, info %x %x %x %x\n", phase, *c0, *c1, *c2,
        *c3, *i0, *i1, *i2, *i3 );

    // convert to twos-comp
    signMag2TwosComp( c0 );
    signMag2TwosComp( c1 );
    signMag2TwosComp( c2 );
    signMag2TwosComp( c3 );

    signMag2TwosComp( i0 );
    signMag2TwosComp( i1 );
    signMag2TwosComp( i2 );
    signMag2TwosComp( i3 );


    // partial summations
    a = *c0 + *c1;
    b = *c2 + *c3;

    c = *i0 + *i1;
    d = *i2 + *i3;

    // track saturation values
    asat = c0sat | c1sat;
    bsat = c2sat | c3sat;
    csat = i0sat | i1sat;
    dsat = i2sat | i3sat;

    // If one of the values being summed is a saturation value, just set the sum to saturation.
/*
    if( bsat | c1sat )
        x0 = satBitMask;
    else{
        // summation
        x0 = b + *c1;
        // put a cap if it overflowed
        if( x0 > magBitMask )
            x0 = magBitMask;
        else if( x0 < -magBitMask )
            x0 = -magBitMask;
```

```
        // convert the result back to sign-mag
        twosComp2SignMag( &x0 );
    }
*/
    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( bsat | c0sat )
        x1 = satBitMask;
    else{
        // summation
        x1 = b + *c0;
        // put a cap if it overflowed
        if( x1 > magBitMask )
            x1 = magBitMask;
        else if( x1 < -magBitMask )
            x1 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( &x1 );
    }

    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( asat | c3sat )
        x2 = satBitMask;
    else{
        // summation
        x2 = a + *c3;
        // put a cap if it overflowed
        if( x2 > magBitMask )
            x2 = magBitMask;
        else if( x2 < -magBitMask )
            x2 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( &x2 );
    }

    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( asat | c2sat )
        x3 = satBitMask;
    else{
        // summation
        x3 = a + *c2;
        // put a cap if it overflowed
        if( x3 > magBitMask )
            x3 = magBitMask;
        else if( x3 < -magBitMask )
            x3 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( &x3 );
    }


    // info summations
/*
    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( dsat | i1sat )
        y0 = satBitMask;
    else{
        // summation
        y0 = d + *i1;
        // put a cap if it overflowed
        if( y0 > magBitMask )
            y0 = magBitMask;
        else if( y0 < -magBitMask )
            y0 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( &y0 );
    }
*/
    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( dsat | i0sat )
        y1 = satBitMask;
    else{
        // summation
        y1 = d + *i0;
        // put a cap if it overflowed
        if( y1 > magBitMask )
            y1 = magBitMask;
        else if( y1 < -magBitMask )
            y1 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( &y1 );
    }

    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( csat | i3sat )
        y2 = satBitMask;
    else{
        // summation
        y2 = c + *i3;
        // put a cap if it overflowed
        if( y2 > magBitMask )
            y2 = magBitMask;
        else if( y2 < -magBitMask )
            y2 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( &y2 );
    }

    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( csat | i2sat )
        y3 = satBitMask;
    else{
        // summation
        y3 = c + *i2;
        // put a cap if it overflowed
        if( y3 > magBitMask )
            y3 = magBitMask;
        else if( y3 < -magBitMask )
            y3 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( &y3 );
    }

    // assign back sign-mag values
//    *c0 = x0;
    twosComp2SignMag( c0 );
    *c1 = x1;
    *c2 = x2;
    *c3 = x3;

//    *i0 = y0;
    twosComp2SignMag( i0 );
    *i1 = y1;
    *i2 = y2;
    *i3 = y3;

    //printf( "%02x %02x %02x %02x %02x %02x %02x %02x\n", *c0,*c1,*c2,*c3,*i0,*i1,*i2,*i3 );
//    printf("phase %d, vnode after, code %x %x %x %x, info %x %x %x %x\n\n", phase,
//            *c0, *c1, *c2, *c3, *i0, *i1, *i2, *i3 );

}
```

APPENDIX B

```
void main()
{

    // Decoder Parameters
    int numRows = 258;
    int numPhases = 129;
    int maxDegree = 6;
    int numProcessors = 8;
    int codeRate = 2;
    int phaseDecoder = 0;
    int procDelta = 1032;    // numRows * 4
    int procDeltaAddr = 774; // numPhases * maxDegree

    int decMem[8256];    // 8 * 258 * 4
    int decMemPtr[6192]; // 8 * 129 * 6

    // Assumed inputs in memory
    int info;
    int code;


    int i, p, row, rowv, sum;
    int i0, i1, i2, i3, c0, c1, c2, c3;


    int temp = 0; // for calculating memory access expressions

    // update the variable node rows with the new code and info values
    row = -codeRate*phaseDecoder;

    if( row < 0 )
        row += numRows;

    while( rowv < 0 )
    rowv += numRows;

    for( p=(numProcessors-1); p>=0; p-- ){
            if( p != 0 ){
                temp = p*procDelta + row;
            temp = (p-1)*procDelta + row;
                decMem = decMem;
                    temp = p*procDelta + row+1*numRows;
            temp = (p-1)*procDelta + row+1*numRows;
                decMem = decMem;
                    temp = p*procDelta + row+2*numRows;
            temp = (p-1)*procDelta + row+2*numRows;
                decMem = decMem;
                    temp = p*procDelta + row+3*numRows;
            temp = (p-1)*procDelta + row+3*numRows;
                decMem = decMem;
                    temp = p*procDelta + (row+1);
            temp = (p-1)*procDelta + (row+1);
                decMem = decMem;
                    temp = p*procDelta + (row+1)+1*numRows;
            temp = (p-1)*procDelta + (row+1)+1*numRows;
```

```
            decMem = decMem;
                temp = p*procDelta + (row+1)+2*numRows;
            temp = (p-1)*procDelta + (row+1)+2*numRows;
               decMem = decMem;
                temp = p*procDelta + (row+1)+3*numRows;
            temp = (p-1)*procDelta + (row+1)+3*numRows;
               decMem = decMem;


            }else{ // the first processors, get the inputs from the channel
                temp = row;
            decMem = code;
                temp = row+1*numRows;
            decMem = code;
                temp = row+2*numRows;
            decMem = code;
                temp = row+3*numRows;
            decMem = code;
                temp = (row+1);
            decMem = info;
                temp = (row+1)+1*numRows;
            decMem = info;
                temp = (row+1)+2*numRows;
            decMem = info;
                temp = (row+1)+3*numRows;
            decMem = info;
            }

         // Calling Check Node Operation - BottleNeck
         temp = p*procDeltaAddr + phaseDecoder*maxDegree ;
         temp = p*procDeltaAddr + phaseDecoder*maxDegree+1 ;
         temp = p*procDeltaAddr + phaseDecoder*maxDegree+2 ;
         temp = p*procDeltaAddr + phaseDecoder*maxDegree+3 ;
      temp = p*procDeltaAddr + phaseDecoder*maxDegree+4 ;
         temp = p*procDeltaAddr + phaseDecoder*maxDegree+5 ;

         cNodeOp( decMemPtr,decMemPtr,decMemPtr,decMemPtr,decMemPtr,decMemPtr );


         // Calling Variable Node Operation
         temp = p*procDelta + rowv ;
         temp = p*procDelta + rowv+1*numRows ;
         temp = p*procDelta + rowv+2*numRows ;
         temp = p*procDelta + rowv+3*numRows ;
         temp = p*procDelta + (rowv+1) ;
         temp = p*procDelta + (rowv+1)+1*numRows ;
         temp = p*procDelta + (rowv+1)+2*numRows ;
         temp = p*procDelta + (rowv+1)+3*numRows ;

      vNodeOp( decMem, decMem, decMem, decMem, decMem, decMem, decMem, decMem );

         // If we are at the last processors, capture the variable node outputs
         if( p == (numProcessors-1) ){
            temp = p*procDelta + rowv;
            c0 = decMem;
            temp = p*procDelta + rowv+1*numRows;
            c1 = decMem;
            temp = p*procDelta + rowv+2*numRows;
            c2 = decMem;
            temp = p*procDelta + rowv+3*numRows;
            c3 = decMem;
            temp = p*procDelta + (rowv+1);
            i0 = decMem;
            temp = p*procDelta + (rowv+1)+1*numRows;
            i1 = decMem;
            temp = p*procDelta + (rowv+1)+2*numRows;
            i2 = decMem;
            temp = p*procDelta + (rowv+1)+3*numRows;
            i3 = decMem;
         }


   }



}
```

APPENDIX C

```
void cNodeOp(int decMemPtr1, int decMemPtr2, int decMemPtr3, int decMemPtr4, int decMemPtr5, int
        decMemPtr6)
{


   // external parameters
   int signBitMask = 64; // Sign bit mask      (1000000).
   int magBitMask  = 31; // Magnitude bit mask (0011111).
   int satBitMask  = 32; // Saturation bit mask(0100000).
   int numRows = 258;
   int phase;


   int min = 10000;
   int min2 = 10000;
   int mag0, mag1, mag2, mag3, mag4, mag5;
   int sign, sign0, sign1, sign2, sign3, sign4, sign5;
   int sat0, sat1, sat2, sat3, sat4, sat5;
   int magPtr;

   sat0=sat1=sat2=sat3=sat4=sat5=0;

   // Obtain the sign bits
   sign0 = decMemPtr1 & signBitMask;
   sign1 = decMemPtr2 & signBitMask;
   sign2 = decMemPtr3 & signBitMask;
   sign3 = decMemPtr4 & signBitMask;
   sign4 = decMemPtr5 & signBitMask;
   sign5 = decMemPtr6 & signBitMask;

   // Obtain the magnitudes
   mag0 = decMemPtr1 & (magBitMask | satBitMask);
   mag1 = decMemPtr2 & (magBitMask | satBitMask);
   mag2 = decMemPtr3 & (magBitMask | satBitMask);
   mag3 = decMemPtr4 & (magBitMask | satBitMask);
   mag4 = decMemPtr5 & (magBitMask | satBitMask);
   mag5 = decMemPtr6 & (magBitMask | satBitMask);

   // Calculate the sign.
   sign = sign0 ^ sign1 ^ sign2 ^ sign3 ^ sign4 ^ sign5;

   sign0 = sign0 ^ sign;
   sign1 = sign1 ^ sign;
   sign2 = sign2 ^ sign;
   sign3 = sign3 ^ sign;
   sign4 = sign4 ^ sign;
   sign5 = sign5 ^ sign;

   // assign the sign bits
   decMemPtr1 = sign0;
   decMemPtr2 = sign1;
   decMemPtr3 = sign2;
   decMemPtr4 = sign3;
   decMemPtr5 = sign4;
   decMemPtr6 = sign5;

   // Find the minimum and 2nd minimum.
   if( mag0 < min ){
         min2 = min;
         min = mag0;
         //magPtr = mag0;
   }else if( mag0 < min2 ){
         min2 = mag0;
   }
   if( mag1 < min ){
         min2 = min;
         min = mag1;
         //magPtr = mag1;
   }else if( mag1 < min2 ){
         min2 = mag1;
   }
   if( mag2 < min ){
         min2 = min;
         min = mag2;
         //magPtr = mag2;
   }else if( mag2 < min2 ){
         min2 = mag2;
   }
   if( mag3 < min ){
         min2 = min;
         min = mag3;
         //magPtr = mag3;
   }else if( mag3 < min2 ){
         min2 = mag3;
   }
   if( mag4 < min ){
         min2 = min;
         min = mag4;
         //magPtr = mag4;
   }else if( mag4 < min2 ){
         min2 = mag4;
   }
   if( mag5 < min ){
         min2 = min;
         min = mag5;
         //magPtr = mag5;
   }else if( mag5 < min2 ){
         min2 = mag5;
   }

   // Assign the minimum and 2nd minimum back to the magnitudes
   mag0 = min;
   mag1 = min;
   mag2 = min;
   mag3 = min;
   mag4 = min;
   mag5 = min;

   if( magPtr == mag0 )
```

```
        mag0 = min2;
    if( magPtr == mag1 )
        mag1 = min2;
    if( magPtr == mag2 )
        mag2 = min2;
    if( magPtr == mag3 )
        mag3 = min2;
    if( magPtr == mag4 )
        mag4 = min2;
    if( magPtr == mag5 )
        mag5 = min2;


    // asign the magnitudes with the possibility of changing a sign bit to 1 (to represent saturation)
    decMemPtr1 |= mag0;
    decMemPtr2 |= mag1;
    decMemPtr3 |= mag2;
    decMemPtr4 |= mag3;
    decMemPtr5 |= mag4;
    decMemPtr6 |= mag5;

}


void signMag2TwosComp( int val ){

    // external parameters
    int signBitMask = 64; // Sign bit mask      (1000000).
    int magBitMask  = 31; // Magnitude bit mask (0011111).
    int satBitMask  = 32; // Saturation bit mask(0100000).
    int numRows = 258;
    int phase;

    int sign;
    sign = val & signBitMask;
    val = val & (magBitMask | satBitMask);
    if( sign )
        val = -(val);
}

void twosComp2SignMag( int val ){

    // external parameters
    int signBitMask = 64; // Sign bit mask      (1000000).
    int magBitMask  = 31; // Magnitude bit mask (0011111).
    int satBitMask  = 32; // Saturation bit mask(0100000).
    int numRows = 258;
    int phase;

    if( val < 0 )
        val = (-(val)) | signBitMask;
}

void vNodeOp(int decMem1, int decMem2, int decMem3, int decMem4, int decMem5, int decMem6, int
        decMem7, int decMem8 )
{

    // external parameters
    int signBitMask = 64; // Sign bit mask      (1000000).
    int magBitMask  = 31; // Magnitude bit mask (0011111).
    int satBitMask  = 32; // Saturation bit mask(0100000).
    int numRows = 258;
    int phase;

    int a, b, c, d;
    int asat, bsat, csat, dsat;
    int x0, x1, x2, x3, y0, y1, y2, y3;
    int c0sat, c1sat, c2sat, c3sat, i0sat, i1sat, i2sat, i3sat;

    c0sat=c1sat=c2sat=c3sat=i0sat=i1sat=i2sat=i3sat=0;
    asat=bsat=csat=dsat=0;

    c0sat = decMem1 & satBitMask;
    c1sat = decMem2 & satBitMask;
    c2sat = decMem3 & satBitMask;
    c3sat = decMem4 & satBitMask;

    i0sat = decMem5 & satBitMask;
    i1sat = decMem6 & satBitMask;
    i2sat = decMem7 & satBitMask;
    i3sat = decMem8 & satBitMask;


    // convert to twos-comp
    signMag2TwosComp( decMem1 );
    signMag2TwosComp( decMem2 );
    signMag2TwosComp( decMem3 );
    signMag2TwosComp( decMem4 );

    signMag2TwosComp( decMem5 );
    signMag2TwosComp( decMem6 );
    signMag2TwosComp( decMem7 );
    signMag2TwosComp( decMem8 );

    // partial summations
    a = decMem1 + decMem2;
    b = decMem3 + decMem4;

    c = decMem5 + decMem6;
    d = decMem7 + decMem8;

    // track saturation values
    asat = c0sat | c1sat;
    bsat = c2sat | c3sat;
    csat = i0sat | i1sat;
    dsat = i2sat | i3sat;


    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( bsat | c0sat )
        x1 = satBitMask;
    else{
        // summation
        x1 = b + decMem1;
        // put a cap if it overflowed
        if( x1 > magBitMask )
            x1 = magBitMask;
        else if( x1 < -magBitMask )
            x1 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( x1 );
    }

    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( asat | c3sat )
        x2 = satBitMask;
    else{
        // summation
        x2 = a + decMem4;
        // put a cap if it overflowed
        if( x2 > magBitMask )
            x2 = magBitMask;
        else if( x2 < -magBitMask )
            x2 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( x2 );
    }

    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( asat | c2sat )
        x3 = satBitMask;
    else{
        // summation
        x3 = a + decMem3;
        // put a cap if it overflowed
        if( x3 > magBitMask )
            x3 = magBitMask;
        else if( x3 < -magBitMask )
            x3 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( x3 );
    }

    /////////
    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( dsat | i0sat )
        y1 = satBitMask;
    else{
        // summation
        y1 = d + decMem5;
        // put a cap if it overflowed
        if( y1 > magBitMask )
            y1 = magBitMask;
        else if( y1 < -magBitMask )
            y1 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( y1 );
    }

    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( csat | i3sat )
        y2 = satBitMask;
    else{
        // summation
        y2 = c + decMem8;
        // put a cap if it overflowed
        if( y2 > magBitMask )
            y2 = magBitMask;
        else if( y2 < -magBitMask )
            y2 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( y2 );
    }

    // If one of the values being summed is a saturation value, just set the sum to saturation.
    if( csat | i2sat )
        y3 = satBitMask;
    else{
        // summation
        y3 = c + decMem7;
```

```
        // put a cap if it overflowed
        if( y3 > magBitMask )
            y3 = magBitMask;
        else if( y3 < -magBitMask )
            y3 = -magBitMask;
        // convert the result back to sign-mag
        twosComp2SignMag( y3 );
    }

    // assign back sign-mag values
//    *c0 = x0;
    twosComp2SignMag( decMem1 );
    decMem2 = x1;
    decMem3 = x2;
    decMem4 = x3;

//    *i0 = y0;
    twosComp2SignMag( decMem5 );
    decMem6 = y1;
    decMem7 = y2;
    decMem8 = y3;


}
```

```
void main()
{

    int out_full = 333;             // initial random value != 0 or 1

    // Start of Iteration
    while ((out_full-1) == 0);
      // processor input initialization goes here
    out_full = out_full - 1;   // extra load happens here


    // Processor loop code goes here


    // End of iteration
    while (out_full == 1);
      // processor output happens automatically
    out_full = 1;               // extra load happens here
}
```

APPENDIX D

```
XCC - XInC Cross Compiler version 1.36 (Beta) Feb 26 2003
Copyright (c) Eleven Engineering Incorporated 2002. All Rights Reserved.

Preprocessing...
---- 0.220 seconds
Cleaning up Preprocessor...
---- 0.010 seconds
Parsing...
---- 0.010 seconds
Checking Symbols...
---- 0.010 seconds
Generating Code...
////////////////////////////////////////////////////////////////
// C Runtime Environment Setup Code
////////////////////////////////////////////////////////////////

// Setup Program Base Address
        @ = 0xC000

#include <XInC_CRuntime.asm>


////////////////////////////////////////////////////////////////
// Compiled Code
////////////////////////////////////////////////////////////////

__F_cNodeOp:
                        st    fp, sp, 0
                        st    ra, sp, 1
                        add   fp, sp, 0
                        add   sp, sp, 35

                        mov   r0, 64
                        st    r0, signBitMask
                        mov   r0, 31
                        st    r0, magBitMask
                        mov   r0, 32
                        st    r0, satBitMask
                        mov   r0, 258
                        st    r0, numRows
                        mov   r0, 10000
                        st    r0, min
                        mov   r0, 10000
                        st    r0, min2
                        mov   r0, 0
                        st    r0, fp, 33

                        st    r0, fp, 32

                        st    r0, fp, 31

                        st    r0, fp, 30

                        st    r0, fp, 29

                        st    r0, fp, 28
                        ld    r0, fp, 2
                        ld    r1, fp, 8

                        and   r0, r0, r1

                        st    r0, fp, 22
                        ld    r0, fp, 3
                        ld    r1, fp, 8

                        and   r0, r0, r1

                        st    r0, fp, 23
                        ld    r0, fp, 4
                        ld    r1, fp, 8

                        and   r0, r0, r1

                        st    r0, fp, 24
                        ld    r0, fp, 5
                        ld    r1, fp, 8

                        and   r0, r0, r1

                        st    r0, fp, 25
                        ld    r0, fp, 6
                        ld    r1, fp, 8

                        and   r0, r0, r1

                        st    r0, fp, 26
                        ld    r0, fp, 7
                        ld    r1, fp, 8

                        and   r0, r0, r1

                        st    r0, fp, 27
                        ld    r0, fp, 9
                        ld    r1, fp, 10

                        ior   r1, r0, r1
                        ld    r0, fp, 2

                        and   r0, r0, r1

                        st    r0, fp, 15
                        ld    r0, fp, 9
                        ld    r1, fp, 10

                        ior   r1, r0, r1
                        ld    r0, fp, 3

                        and   r0, r0, r1

                        st    r0, fp, 16
                        ld    r0, fp, 9
                        ld    r1, fp, 10

                        ior   r1, r0, r1
                        ld    r0, fp, 4

                        and   r0, r0, r1

                        st    r0, fp, 17
                        ld    r0, fp, 9
                        ld    r1, fp, 10

                        ior   r1, r0, r1
                        ld    r0, fp, 5

                        and   r0, r0, r1

                        st    r0, fp, 18
                        ld    r0, fp, 9
                        ld    r1, fp, 10

                        ior   r1, r0, r1
                        ld    r0, fp, 6

                        and   r0, r0, r1

                        st    r0, fp, 19
                        ld    r0, fp, 9
                        ld    r1, fp, 10
```

```
                ior   r1, r0, r1
                ld    r0, fp, 7

                and   r0, r0, r1

                st    r0, fp, 20
                ld    r0, fp, 22
                ld    r1, fp, 23

                xor   r0, r0, r1
                ld    r1, fp, 24

                xor   r0, r0, r1
                ld    r1, fp, 25

                xor   r0, r0, r1
                ld    r1, fp, 26

                xor   r0, r0, r1
                ld    r1, fp, 27

                xor   r0, r0, r1

                st    r0, fp, 21
                ld    r0, fp, 22
                ld    r1, fp, 21

                xor   r0, r0, r1

                st    r0, fp, 22
                ld    r0, fp, 23
                ld    r1, fp, 21

                xor   r0, r0, r1

                st    r0, fp, 23
                ld    r0, fp, 24
                ld    r1, fp, 21

                xor   r0, r0, r1

                st    r0, fp, 24
                ld    r0, fp, 25
                ld    r1, fp, 21

                xor   r0, r0, r1

                st    r0, fp, 25
                ld    r0, fp, 26
                ld    r1, fp, 21

                xor   r0, r0, r1

                st    r0, fp, 26
                ld    r0, fp, 27
                ld    r1, fp, 21

                xor   r0, r0, r1

                st    r0, fp, 27
                ld    r0, fp, 22

                st    r0, fp, 2
                ld    r0, fp, 23

                st    r0, fp, 3
                ld    r0, fp, 24

                st    r0, fp, 4
                ld    r0, fp, 25

                st    r0, fp, 5
                ld    r0, fp, 26

                st    r0, fp, 6
                ld    r0, fp, 27

                st    r0, fp, 7
__IF_1_COND:
                ld    r0, fp, 15
                ld    r1, fp, 13

                sub   r0, r0, r1
                bc    LT, @+3
                mov   r0, false
                bra   @+2
                mov   r0, true
                add   r0, r0, 0
                bc    ZS, __IF_1_ELSE
__IF_1_THEN:

                ld    r0, fp, 13

                st    r0, fp, 14
                ld    r0, fp, 15

                st    r0, fp, 13

                bra   __IF_1_END
__IF_1_ELSE:
__IF_2_COND:
                ld    r0, fp, 15
                ld    r1, fp, 14

                sub   r0, r0, r1
                bc    LT, @+3
                mov   r0, false
                bra   @+2
                mov   r0, true
                add   r0, r0, 0
                bc    ZS, __IF_2_END
__IF_2_THEN:

                ld    r0, fp, 15

                st    r0, fp, 14
__IF_2_END:
__IF_1_END:
__IF_3_COND:
                ld    r0, fp, 16
                ld    r1, fp, 13

                sub   r0, r0, r1
                bc    LT, @+3
                mov   r0, false
                bra   @+2
                mov   r0, true
                add   r0, r0, 0
                bc    ZS, __IF_3_ELSE
__IF_3_THEN:

                ld    r0, fp, 13

                st    r0, fp, 14
                ld    r0, fp, 16

                st    r0, fp, 13

                bra   __IF_3_END
__IF_3_ELSE:
__IF_4_COND:
                ld    r0, fp, 16
                ld    r1, fp, 14

                sub   r0, r0, r1
                bc    LT, @+3
                mov   r0, false
                bra   @+2
                mov   r0, true
                add   r0, r0, 0
                bc    ZS, __IF_4_END
__IF_4_THEN:

                ld    r0, fp, 16

                st    r0, fp, 14
__IF_4_END:
__IF_3_END:
__IF_5_COND:
                ld    r0, fp, 17
                ld    r1, fp, 13

                sub   r0, r0, r1
                bc    LT, @+3
                mov   r0, false
                bra   @+2
                mov   r0, true
                add   r0, r0, 0
                bc    ZS, __IF_5_ELSE
__IF_5_THEN:

                ld    r0, fp, 13

                st    r0, fp, 14
                ld    r0, fp, 17

                st    r0, fp, 13

                bra   __IF_5_END
__IF_5_ELSE:
__IF_6_COND:
                ld    r0, fp, 17
                ld    r1, fp, 14

                sub   r0, r0, r1
                bc    LT, @+3
                mov   r0, false
```

```
                bra  @+2
                mov  r0, true
                add  r0, r0, 0
                bc   ZS, __IF_6_END
__IF_6_THEN:

                ld   r0, fp, 17

                st   r0, fp, 14

__IF_6_END:
__IF_5_END:
__IF_7_COND:
                ld   r0, fp, 18
                ld   r1, fp, 13

                sub  r0, r0, r1
                bc   LT, @+3
                mov  r0, false
                bra  @+2
                mov  r0, true
                add  r0, r0, 0
                bc   ZS, __IF_7_ELSE
__IF_7_THEN:

                ld   r0, fp, 13

                st   r0, fp, 14
                ld   r0, fp, 18

                st   r0, fp, 13

                bra  __IF_7_END
__IF_7_ELSE:
__IF_8_COND:
                ld   r0, fp, 18
                ld   r1, fp, 14

                sub  r0, r0, r1
                bc   LT, @+3
                mov  r0, false
                bra  @+2
                mov  r0, true
                add  r0, r0, 0
                bc   ZS, __IF_8_END
__IF_8_THEN:

                ld   r0, fp, 18

                st   r0, fp, 14

__IF_8_END:
__IF_7_END:
__IF_9_COND:
                ld   r0, fp, 19
                ld   r1, fp, 13

                sub  r0, r0, r1
                bc   LT, @+3
                mov  r0, false
                bra  @+2
                mov  r0, true
                add  r0, r0, 0
                bc   ZS, __IF_9_ELSE
__IF_9_THEN:

                ld   r0, fp, 13

                st   r0, fp, 14
                ld   r0, fp, 19

                st   r0, fp, 13

                bra  __IF_9_END
__IF_9_ELSE:
__IF_10_COND:
                ld   r0, fp, 19
                ld   r1, fp, 14

                sub  r0, r0, r1
                bc   LT, @+3
                mov  r0, false
                bra  @+2
                mov  r0, true
                add  r0, r0, 0
                bc   ZS, __IF_10_END
__IF_10_THEN:

                ld   r0, fp, 19

                st   r0, fp, 14

__IF_10_END:
__IF_9_END:
__IF_11_COND:
                ld   r0, fp, 20
                ld   r1, fp, 13

                sub  r0, r0, r1
                bc   LT, @+3
                mov  r0, false
                bra  @+2
                mov  r0, true
                add  r0, r0, 0
                bc   ZS, __IF_11_ELSE
__IF_11_THEN:

                ld   r0, fp, 13

                st   r0, fp, 14
                ld   r0, fp, 20

                st   r0, fp, 13

                bra  __IF_11_END
__IF_11_ELSE:
__IF_12_COND:
                ld   r0, fp, 20
                ld   r1, fp, 14

                sub  r0, r0, r1
                bc   LT, @+3
                mov  r0, false
                bra  @+2
                mov  r0, true
                add  r0, r0, 0
                bc   ZS, __IF_12_END
__IF_12_THEN:

                ld   r0, fp, 20

                st   r0, fp, 14

__IF_12_END:
__IF_11_END:
                ld   r0, fp, 13

                st   r0, fp, 15
                ld   r0, fp, 13

                st   r0, fp, 16
                ld   r0, fp, 13

                st   r0, fp, 17
                ld   r0, fp, 13

                st   r0, fp, 18
                ld   r0, fp, 13

                st   r0, fp, 19
                ld   r0, fp, 13

                st   r0, fp, 20
__IF_13_COND:
                ld   r0, fp, 34
                ld   r1, fp, 15

                sub  r0, r0, r1
                bc   EQ, @+3
                mov  r0, false
                bra  @+2
                mov  r0, true
                add  r0, r0, 0
                bc   ZS, __IF_13_END
__IF_13_THEN:
                ld   r0, fp, 14

                st   r0, fp, 15
__IF_13_END:
__IF_14_COND:
                ld   r0, fp, 34
                ld   r1, fp, 16

                sub  r0, r0, r1
                bc   EQ, @+3
                mov  r0, false
                bra  @+2
                mov  r0, true
                add  r0, r0, 0
                bc   ZS, __IF_14_END
__IF_14_THEN:
                ld   r0, fp, 14

                st   r0, fp, 16
__IF_14_END:
__IF_15_COND:
                ld   r0, fp, 34
                ld   r1, fp, 17

                sub  r0, r0, r1
```

```
                    bc    EQ, @+3
                    mov   r0, false
                    bra   @+2
                    mov   r0, true
                    add   r0, r0, 0
                    bc    ZS, __IF_15_END
__IF_15_THEN:
                    ld    r0, fp, 14

                    st    r0, fp, 17
__IF_15_END:
__IF_16_COND:
                    ld    r0, fp, 34
                    ld    r1, fp, 18

                    sub   r0, r0, r1
                    bc    EQ, @+3
                    mov   r0, false
                    bra   @+2
                    mov   r0, true
                    add   r0, r0, 0
                    bc    ZS, __IF_16_END
__IF_16_THEN:
                    ld    r0, fp, 14

                    st    r0, fp, 18
__IF_16_END:
__IF_17_COND:
                    ld    r0, fp, 34
                    ld    r1, fp, 19

                    sub   r0, r0, r1
                    bc    EQ, @+3
                    mov   r0, false
                    bra   @+2
                    mov   r0, true
                    add   r0, r0, 0
                    bc    ZS, __IF_17_END
__IF_17_THEN:
                    ld    r0, fp, 14

                    st    r0, fp, 19
__IF_17_END:
__IF_18_COND:
                    ld    r0, fp, 34
                    ld    r1, fp, 20

                    sub   r0, r0, r1
                    bc    EQ, @+3
                    mov   r0, false
                    bra   @+2
                    mov   r0, true
                    add   r0, r0, 0
                    bc    ZS, __IF_18_END
__IF_18_THEN:
                    ld    r0, fp, 14

                    st    r0, fp, 20
__IF_18_END:
                    ld    r0, fp, 15
                    ld    r0, fp, 2

                    ior   r1, r0, r1
                    st    r1, fp, 2
                    ld    r0, fp, 16
                    ld    r0, fp, 3

                    ior   r1, r0, r1
                    st    r1, fp, 3
                    ld    r0, fp, 17
                    ld    r0, fp, 4

                    ior   r1, r0, r1
                    st    r1, fp, 4
                    ld    r0, fp, 18
                    ld    r0, fp, 5

                    ior   r1, r0, r1
                    st    r1, fp, 5
                    ld    r0, fp, 19
                    ld    r0, fp, 6

                    ior   r1, r0, r1
                    st    r1, fp, 6
                    ld    r0, fp, 20
                    ld    r0, fp, 7

                    ior   r1, r0, r1
                    st    r1, fp, 7

__F_cNodeOp_END:
                    ld    ra, fp, 1
                    add   sp, fp, 0
                    ld    fp, fp, 0
                    jsr   ra, ra
```