FPGA implementation of a Flexible LDPC decoder

David Hayes (3018978)

October 28, 2008

Academic Supervisor: Steven Weller

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Engineering in Telecommunication Engineering at The University of Newcastle, Australia.

October 28, 2008

Contents

Abstract

This project has implemented a Low Density Parity Check (LDPC) decoder in field programmable gate array (FPGA) hardware. The key design feature of the decoder is flexibility, being able to support different codes and rates. The advantages of being able to test a variety of codes such as those found in communications standards including Wireless LAN (IEEE 802.lln), WIMAX (IEEE 801.16e) and DVB-S2, on a single hardware platform will make the transition from theory to practice a much smoother one.

The LDPC decoder designed is flexible, it can accept a new code without any redesign, the only thing that changes is the ROM configuration. It has been tested via its serial Matlab interface and has been functionally verified, performing to within 1dB of a Matlab floating point testbench. The decoder has also been shown to operate correctly on large codes and as performance demands increase, it has been shown to scale up to larger designs well.

Contributions

This project implements in hardware, a flexible iterative decoder based on a recently published paper [1]. The author has made the following contributions towards the project.

1. Understand, develop, implement and verify in MATLAB, a scheduling algorithm that allows the flexibility in the LDPC decoder
2. Understand, develop, implement in MATLAB, an algorithm that configures the Benes crossbar switches
3. Develop and implement and verify in VHDL the correct operation of the Benes crossbar switches
4. Design a completely flexible LDPC architecture
5. Develop and simulate a correctly functioning LDPC decoder in VHDL
6. Create a MATLAB testbed for verification of the LDPC decoder
7. Implement the LDPC decoder on FPGA hardware
8. Successfully verify the LDPC decoders operation via communication between FPGA hardware and MATLAB testbench

David Hayes Steven Weller

Chapter 1

Introduction

1.1 Project Overview

This project details the design, implementation and testing of a flexible Low Density Parity Check (LDPC) decoder built on Field Programmable Gate Array (FPGA) hardware. The design goal is to implement a *flexible* LDPC decoder, that can handle different types of codes, enabling a variety of codes to be tested without redesigning the decoder. One of the project's long term goals is to integrate the decoder onto Multiple Input Multiple Output (MIMO) communications testbed that the Signal Processing and Microelectronics (SPM) group at the University of Newcastle, are developing.

1.1.1 Error Correction Coding

Error correction coding is a means whereby errors introduced into a communications system can be both detected and corrected by the receiver. It is a key part of the communications system, essential in our modern information based society. Compact Disks, CD-ROM's DVD's mobile phones and hard disk drives all employ error correction coding to enable reliable information transfer.

In 1948 Shannon published a paper that revolutionised communications. He found that there is a fundamental limit to how much information can be reliably sent over a channel (its *capacity*), related to the noise present [2]. Although he defined a limit, the proof is non-constructive, it only shows how good the best possible code will perform. In proving

the channel capacity, Shannon used random codes of infinite length; while these are not practical it suggests that to find an encoding scheme that approaches this fundamental limit we should use codes that are both random and have long length.

Low Density Parity-Check (LDPC) Codes are a class of block code that satisfy both having a long length and randomness. These codes have the best performing to date, with their capacity within 0.0045 dB of the Shannon limit [3]. It is their excellent performance coupled with their low decoding complexity that has seen LDPC codes included in standards for data transmission such as WIMAX (IEEE 802.16e), Digital Video Broadcasting (DVB-S2) and as candidate for the new Wireless LAN standard (IEEE 802.n) and in emerging mobile phone standards (3G-Long Term Evolution). While LDPC codes are finding their way into standards for data transmission (mandated and optionally), currently there is no off the self hardware that a manufacturer can purchase to develop such devices that may utilise LDPC codes.

One of the primary goals of realising an LDPC decoder in hardware is flexibility (the ability to decode different codes without needing to redesign the hardware), enabling the above codes (and others) to be tested on a hardware platform, putting theory into practice.

1.1.2 Achievements

A flexible LDPC decoder has been designed in VHDL using Altera's Quartus II 7.0 software. The scheduling and ROM configuration has been implemented in Matlab. The decoder has been placed on an Altera Cyclone II (EP2C35F672C6N) DE2 board. The design has been verified through a serial Matlab interface to the Testbench, confirming the correct operation of the decoder

1.2 Block Codes

A (n, k) block code takes k bits (message bits) at a time and produces n bits (code bits). Block codes introduce redundancy so that errors may be corrected. Let u be the collection of k message bits, denoted a *message word*, and c be the collection of n encoded nits, denoted a *codeword*.

$$u = \begin{bmatrix} u_0 & u_1 & \dots & u_{k-1} \end{bmatrix}$$

$$c = \begin{bmatrix} c_0 & c_1 & ... & c_{n-1} \end{bmatrix}$$

An important parameter of a block code is its *code rate (r)*. It is a measure of the amount of information (message) bits sent per codeword. The more information, the less redundancy and fewer errors the code can correct, conversely with a high amount of redundancy the transmitter is spending less time sending information.

The codewords are encoded through a *generator matrix* G by the following identity:

$$c = u * G$$

Where G is a (n,k) binary matrix.
For example a (7,4) code:

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.1}$$

For every generator matrix G, there exists many *parity check matrices (H)* that satisfy :

$$G * H^T = 0 \tag{1.2}$$

Continuing the example shown in Equation 1.1, a parity check matrix is given by:

$$H = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix} \tag{1.3}$$

The parity check matrix has the following important property, c is a codeword if and only if it satisfies:

$$c * H^T = 0. \tag{1.4}$$

That is all the codewords lie in the nullspace of H. The condition in Equation 1.4 imposes linear constraints among the bits of c called parity check equations. The parity check matrix of 1.3 gives rise to the following parity check equations:

$$c_0 \oplus c_3 \oplus c_5 \oplus c_6 = 0$$

$$c_1 \oplus c_3 \oplus c_4 = 0$$

$$c_2 \oplus c_4 \oplus c_5 = 0$$

where the $\oplus$ operation is an addition in modulo-2 or simply an XOR operation.

For short block codes, a decoding scheme known as syndrome decoding, which utilises H, can be used. This form of decoding is not practical for larger block lengths and other decoding algorithms are used.

1.3 Low Density Parity Check Codes

Low-Density-Parity-Check (LDPC) Codes are a form of forward error correction codes, first discovered by Gallager in 1962 [4]. They are a block code whose parity-check matrix (H) contains very few non zero entities. It is the sparseness of H that guarantees decoding complexity and minimum distance both grow linearly with block length.

The largest difference between LDPC codes and other block codes is how they are decoded. Traditional block codes are usually decoded using some form of Maximum-Likelihood (ML) algorithms. LDPC codes are decoded iteratively using the graphical properties of the parity check matrix H.

1.3.1 Code construction

In contrast to other block codes, LDPC codes are constructed using the parity-check matrix not generator matrix. Codes can be constructed using either the matrix itself or its graphical representation, the Tanner graph.

An LDPC parity-check matrix is considered (w_c, w_r)-*regular* if each code bit is contained in a fixed number of, w_c parity checks and each parity-check equation contains a fixed number,w_r, of code bits [5].

If the constraints for (w_c, w_r)-*regular* are relaxed, then an irregular code can be constructed. For an irregular parity-check matrix, the fraction of columns with weight i is denoted v_i and the fractions of rows with weight i by h_i. Collectively v and h is called the degree distribution of the code. Generally irregular codes perform better then regular codes, at the cost of hardware decoding complexity [6].

1.3.2 Graphical representation of LDPC codes

The parity check matrix can also be represented in a graphical form known as a Tanner graph. A Tanner graph consists of two sets of vertices: n vertices for the codeword bits (bit nodes), and m vertices for the parity-check equations (check nodes). An edge joins a bit node to a check node, if that bit is included in the corresponding parity-check equation. The number of edges in the tanner graph is equal to the number of ones in the parity check matrix. Figure 1.1, shows a Tanner graph representation of the following parity check matrix.

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{1.5}$$

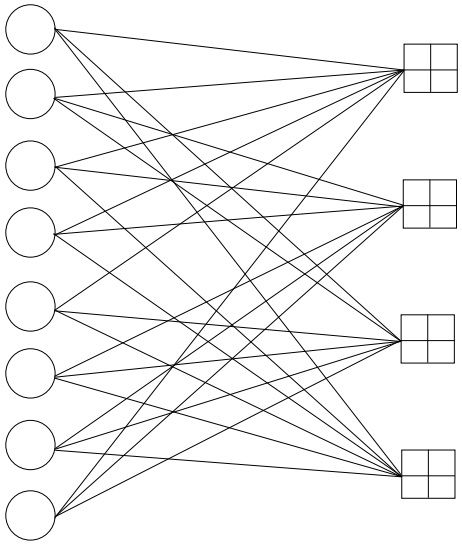


Figure 1.1: A Tanner graph representation of a parity-check matrix.

1.3.3 Decoding - Message Passing

As discussed earlier, generally, block codes are decoded using some form of *Maximum Likelihood (ML)* decoding. This problem is classed as *NP-Complete* which means that there

is no known solution which runs in polynomial time. For the large block lengths used by LDPC codes, ML decoding is not feasible, instead, the *Message Passing* (also known as belief propagation) is used.

It is called message passing as messages are passed back and forwards along the edges of the tanner graph. The messages are probabilistic reliability information about the transmitted bit expressed as *Log Likelihood Ratios (LLRs)*. Expressing them in this manner allows the product of probabilities to be reduced to the addition of LLR's reducing implementation complexity.

The aim of sum-product decoding is to compute the *maximum a posteriori probability MAP* for each codeword bit $P_i = P\{c_i = 1|N\}$, which is the probability that the ith codeword bit is a 1 conditional on the event N that all the parity check equations are satisfied. The extra information about bit i received from the parity checks is called the *extrinsic* information about bit i.

The sum-product algorithm iteratively computes an approximation of the MAP value for each code bit. The algorithm converges to the MAP probabilities only if the Tanner graph is cycle free. If the graph contains cycles then the extrinsic information becomes correlated with the a priori bit probability, preventing the a posteriori probabilities being exact. The extrinsic message from check node j to bit node i, $E_{j,i}$ is the LLR probability that bit i causes the parity check equation j to be satisfied. The probability that the parity check equation is satisfied if bit i is a 1 is

$$E_{ij} = 2\tanh^{-1}\left(\prod_{\alpha\in V[j]\ \alpha\neq i}\tanh\left(\frac{M_{\alpha j}}{2}\right)\right) \quad [5] \tag{1.6}$$

where $M_{\alpha j}$ is the LLR the current estimate availible to check equation j, of the probability that the bit α is a one. V_j is the set of bit nodes connected to check node j. Note that the messages sent from the check nodes to the bit nodes does not include the information that the bit node already has.

The message that is sent from the bit nodes to the check nodes is given by

$$Q_{ij} = \lambda_j + \sum_{\alpha\in C[j]\ \alpha\neq i} R_{\alpha j} \quad [5] \tag{1.7}$$

where C_j is the set of check nodes connected to bit node j and λ_i is the LLR of the a priori

message probabilities. Again the message sent from the bit nodes to the check nodes does not include the information that the check node already has.

The sum product algorithm consists of the following steps. With the the input the LLRs for the a priori message probabilities, the parity check matrix H and the maximum number of allowed iterations I_{max}.

- Initialise

  Set $Q_{ij} = \lambda_j$, this initialises the check nodes with the a priori message probabilities.

- Update Check Messages

  For each check node j, and for every bit node associated with it j compute:

$$R_{ij} = 2 \tanh^{-1} \prod_{\alpha \in V[j] \alpha \neq i} \tanh \left(\frac{Q_{\alpha j}}{2} \right) \tag{1.8}$$

- Test for Valid Codeword

  Make a tentative decision on codeword

$$L_i = \lambda_j + \sum_{j \in C[j]} Q_{ij} \tag{1.9}$$

$$z_i = \begin{cases} 1, & L_i \leq 0, \\ 0, & L_i > 0 \end{cases}$$

  If number of iterations is I_{max} or a valid codeword has been found ($Hz^T = 0$) then finish

- Update Bit Messages For each bit node j, and for every check node associated with it j compute:

$$Q_{ij} = \lambda_j + \sum_{\alpha \in C[j] \ \alpha \neq i} R_{\alpha j}[k-1] \tag{1.10}$$

  Proceed back to Update Check Messages.

Chapter 2

Literature Review

The first goal of this project was to research literature to find how LDPC decoders are implemented in hardware. After a thorough search, many papers were found detailing hardware implementations. They were categorised into which architecture they used, and then ranked in importance based on the design goal of a flexible decoder that be able to be implemented onto a FPGA platform. A brief discussion of the main architectures found, including their top diagrams, speed, flexibility and suitability for a FPGA follows.

2.1 Serial Decoder

A serial decoder is the simplest decoder in terms of hardware cost. It consists of a single check node, a single variable node and memory. The variable nodes are updated, one at a time, then the check nodes are updated in the same serial manner. This is the type of implementation that would arise from writing code that simply implements the sum product algorithm either on a computer or as for a soft core on an FPGA. The main advantage of a serial implementation is that it is very flexible, supporting different block sizes and code rates, with only a new parity matrix to be loaded into memory. Unfortunately this approach is too slow for anything except simulations. [7]. The number of clock cycles required for each iteration of the serial decoder is approximately the twice number of edges. Figure 2.1 shows the architecture for a serial decoder.

Example

The proposed code for wireless LAN, IEEE 802.11n is a $(1944, 792)$ code with 6803 edges.

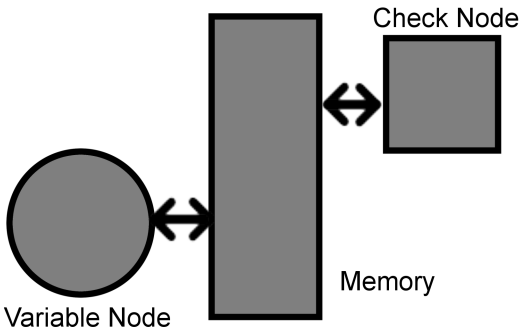


Figure 2.1: Serial Decoder Architecture.

It would take approximately 2×6803 or 13606 clock cycles to complete one iteration of the decoder.

The fact that nothing could be found in the literature implementing a serial LDPC decoder shows that it is not feasible for any application where you might want to use LDPC codes.

2.2 Analog Implementation

While the majority of the papers looked used digital signals to implement the decoder architecture, a number of papers use analog methods. While it is infeasible to implement an analog decoder on an FPGA (FPGA devices can only process digital signals), it is nonetheless interesting to see analog methods applied to solve a purely digital problem, especially when they perform comparably to their digital counterparts.

There were two approaches to implementing the sum product algorithm found, one was to use the non-linearities of charging a Field Effect Transistor (FET) [8, 9], while the other was to use a combinations of resistors and capacitors, creating RC differential equations [10], shown in Figure 2.2

2.3 Parallel

From Figure 1.1 and the sum product algorithm shown in Section 1.3.3 it can be seen that there is a lot of parallelism to exploit in an LDPC decoder. A fully-parallel decoder, in

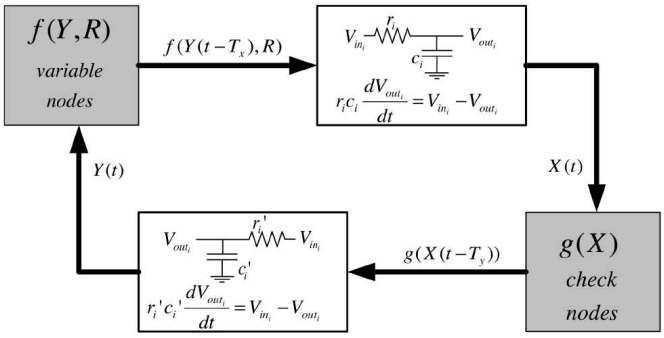


Figure 2.2: Analog implementation [10]

one clock cycle completes the update check message step, and then in the next completes the update bit message step. This allows a marked increase in decoding speed compared to the serial decoder. Papers that were found to be looking at parallel architectures are [11, 12, 13, 14]

Example

Continuing the example from the serial case, it would take 2 clock cycles to complete one iteration of the decoder.

Figure 2.3 shows the architecture of the parallel decoder proposed by Blanksby and Howland.

However, this speed increase comes at the cost of both implementation complexity and flexibility. A parallel decoder implements every check and variable node once in hardware and are routed together as described by the Tanner graph of the code [11]. While the check and variable nodes themselves are simple the interconnection framework is not. The number of wires is given by:

$$\text{message wires} = \text{number of graph edges} \times \text{number of bits/message} \times 2$$

The factor of 2 comes from Blanksby and Howland deciding to use 2 sets unidirectional message wires as opposed to 1 set of bidirectional wires to reduce logic requirements and eliminate the need for bi-directional buffers. The parallel implementations that have been investigated [11, 12, 13], all use 4 bits per message; one for the sign and 3 bits for the magnitude, while Nagarajan's design uses both 4 and 5 bits per message [14].

In contrast it has been found that between 4 and 8 bits per message are needed to give good performance of the decoder [15, 7, 16], which suggests that in the design of parallel decoders, error performance has been sacrificed for throughput. Although the fully parallel designers may be suitable for high bitrate decoders operating at a comparatively high SNR.

The number of message wires needed for a parallel decoder excludes *Field-Programmable-Gate-Array* (FPGA) as the target device, it instead requires custom silicon.
To alleviate the difficulty in routing the signal wires, a three dimensional design has been proposed with 7 levels [11], a three dimensional 3-tiered approach was also proposed [13].

None of the studied parallel decoder implementations [11, 12, 13, 14] have any flexibility in regards to code design. The routing between check and bit nodes needs to be redesigned for another code realisation, while a different code rate or degree distribution may not be possible without a complete redesign. This is in addition to the fact they the above parallel decoders were all designed on custom silicon, which precludes any prospect of reprogramming.

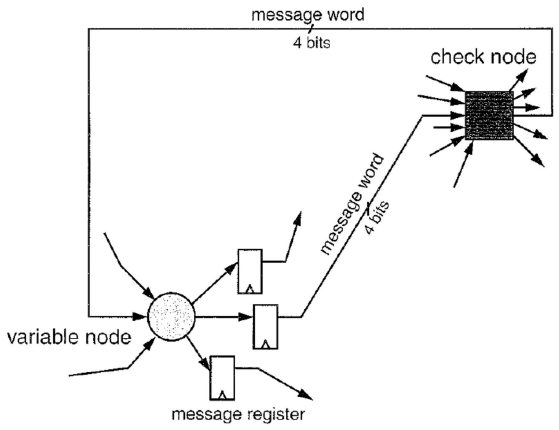


Figure 2.3: Parallel Decoder Architecture [11]

2.4 Partially Parallel

In-between the extremes of a serial and a parallel implementation lies partially parallel implementations. In a partially parallel design, some check and bit nodes are realised in hardware, this involves sharing as in the serial case. It also necessitates the use of memory

to store messages between between clock cycles. It also offers the benefit of parallelism found in a parallel decoder because multiple bit and check nodes can run in parallel.

There is a trade off between speed and complexity, increasing the number of check and bit nodes realised in hardware increases the speed of each decoder iteration but it also increases the complexity. Serial and parallel architectures could be considered subclasses of the partially parallel decoder (serial would have only *one* bit and check node, while parallel would have *all* bit and check nodes realised in hardware).

The major issue with this design is memory collisions. If two nodes try to access the same memory during the same clock cycle a collision occurs and one of the nodes has to stall for a cycle, which decreases the throughput of the decoder. The frequency of these collisions can be reduced by two methods utilised by Masera et al's implementation, a top-level diagram can be seen in figure 2.4. The interconnection has been implemented using a form of crossbar switch known as a Benes network, it allows complete flexibility connecting nodes together. For each code a routing configuration is generated offline using an algorithm described in [1] and in this report Chapter 4. This algorithm configures the Benes network for each clock cycle of the decoder, implementing one iteration.

The other key to reducing memory collisions is modifying the belief propagation algorithm. Check to bit and bit to check messages from an individual node are very similar, the only difference being the message they propagate along an edge ignores the effect of the message that came from that edge. These messages can be changed from unicast to broadcast if the node the message is sent to keeps a copy of the message it sent in the previous half-iteration. This greatly reduces the number of messages that have to be sent between nodes, with the only cost being nodes must have memory to store the previously sent message.

Lee and Ryu have also used the Benes network in their partially parallel LDPC decoder [16]. Their design being highly complex and deeply pipelined, is suitible only for Application Specific Integrated Circuit (ASIC) implementation. My design is loosely based on that of Masera et al's, using the scheduling described in [1]. The key difference being mine is based on unicast messages, while Masera et al's is based on multicast. For my design this makes the interlever more complex while the bit and check node processors as well as the scheduling is simpler.

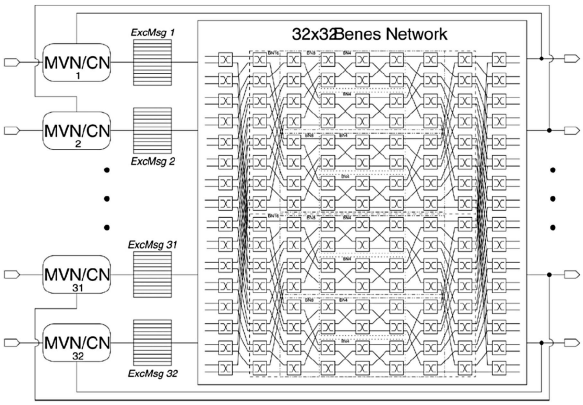


Figure 2.4: Partially Parallel Decoder Architecture [7]

2.5 Specific Code Design

A number of papers detailed methods to simplify the decoding process of LDPC codes by a specific design of the parity check matrix. By enforcing restrictions on the types of codes supported the decoder can simplified by exploiting the properties of the code such as regularity of the parity check matrix reducing its storage requirements on the decoder [17].

Hardware aware, (collision free codes) can also be designed to eliminate collisions found in partially parallel architectures and achieve high throughputs [18].

The key goal of the project is to implement a *flexible* LDPC decoder; designing a decoder based on specific code design techniques and restricting the types of supported codes to a greatly reduced subset would not meet the specifications of the project.

Chapter 3

Flexible Decoder Implementation

Having investigated the various types of LDPC decoder implementations in the literature, a decision was made on the architecture of the design. A partially parallel design was chosen, as it has the advantage of a serial design, being flexible, while having some of the speed advantages offered by a parallel design. A compromise between decoding speed and implementation complexity can be made by varying the number of parallel processing nodes in the design.

The main design goal is flexibility, the decoder should be able to implement any given code (up to maximum design constraints), without changing the design of the decoder.

3.1 Overview

The architecture consists of a number (P) of processors, a message permutation block and a control logic block, as seen in figure 3.1.

There is a smaller number of bit (check) processors then bit (check) nodes in the Tanner Graph meaning that each bit (check) processor is assigned a subset of these nodes. The processors themselves are responsible for storing the incoming messages, performing the node operations and forwarding the outgoing messages, while the assignment of the nodes to processors is handled by the control unit.

The decoding process follows four distinct parts as shown in figure 3.2. The bit to check and check to bit half iterations are repeated a predetermined number of times before outputting the decoded codeword. The number of iterations is likely to be small, around

ten to keep the decoding time small. No checks for a valid codeword are performed; the time to perform a check is as long as a bit (or check) half iteration and so performing one each iteration would increase the iteration time by one third. With a small number of iterations, the benefit of early termination is likely to be outweighed by the increase in cycle time.

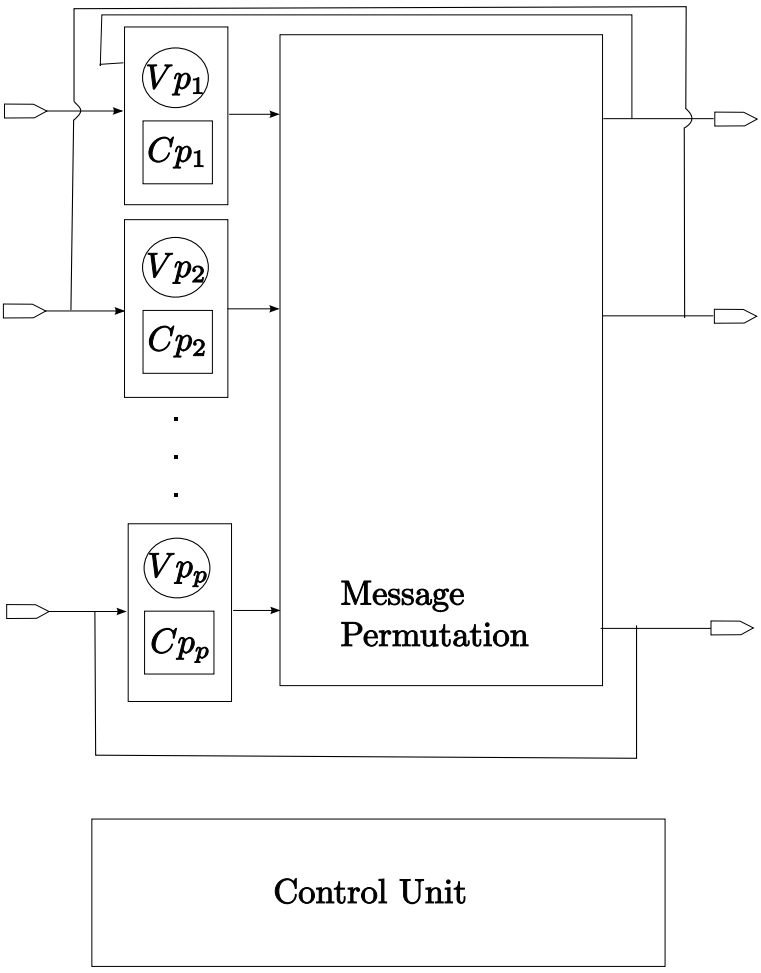


Figure 3.1: Top block diagram for decoder.

- Initialisation

  During initialisation, channel measurements are loaded into the bit processor blocks.

- Bit to Check

  During the bit to check half iteration, the bit node processor performs the bit node function as described in Equation 1.10. In the first iteration there are no incoming

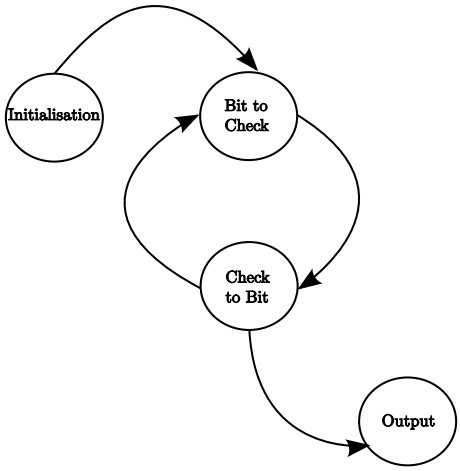


Figure 3.2: State machine for decoder.

messages and the processor outputs the channel measurement. The messages flow from the bit node processors, through the message permuting block and to the correct check node processor in order.

- Check to Bit
  During the bit to check half iteration, the check node processor performs the check node function as described in Equation 1.8. The messages flow from the check node processors, through the message permuting block and to the correct bit node processor in order.

- Output
  After a number of iterations of bit to check and check to bit cycles, the decoder moves to the output stage.

3.2 Message Permutation Block

The message permutation block lies at the heard of the flexibility of the decoder. Its purpose is to connect the bit node processors to the check node processors (and vice versa), so that the check node processor receives the incoming messages in the order of the Tanner graph. Chapter **??** examines in detail how the permutation block achieves this and how its configured.

The message permutation block consists of two Benes crossbar switches with a bank of interleavers in between as shown in figure 3.3. The incoming messages are first permuted in space by the left most Benes network and then stored in the interleaver banks. The interleaver banks then permute the outgoing messages in time before they are again permuted in space by the right most Benes network. These time and space permutations give complete flexibility, allowing incoming messages to arrive at the correct output in the required order. The configuration for the Benes networks and the interleaver bank is varied every clock cycle according to the scheduling configuration stored in the control unit. The algorithm to determine the scheduling is described in detail in chapter 4.

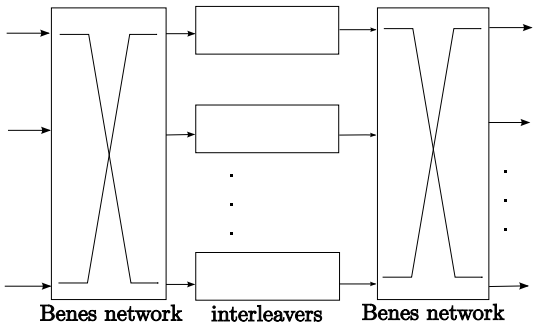


Figure 3.3: Message permutation block.

3.3 Configuring the Benes Network Switch Settings

A Benes network is a multistage switching network formalised by Benes [19], for telephone switching circuits. It eliminates the need for a large capacity crossbar switch by replacing it with a number of simple 2 input, 2 output, switching elements as shown in figure 3.4. A N (where $N = 2^r$) input network can be built recursively using elementary cells and 2 $\frac{N}{2}$ input networks, as seen in figure 3.5. I_i are elementary switches connected to the inputs, each switch connects to the upper (P_A) and lower (P_B) $\frac{N}{2}$ Benes networks. O_i are elementry switches connected to the outputs of the Benes network, each switch is connected to the upper (P_A) and lower (P_B) permutation sub-blocks. This process of decomposition is repeated until only elementary cells remain.

For an N input Benes Network there will be $N \log_2 N - 1$ stages, and a $\frac{N}{2}$ switches per stage. Waksman has proved that one of the switches in each level is redundant and can be

set arbitrarily [20]. The top right switch will be used and it will (arbitrarily) be in the reset state, as shown in figure 3.5. The number of switches required will be $N \log_2 N - N + 1$.

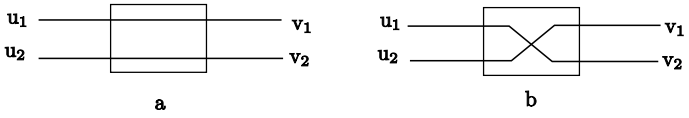


Figure 3.4: Elementary Switching elements (a) Reset. (b) Set.

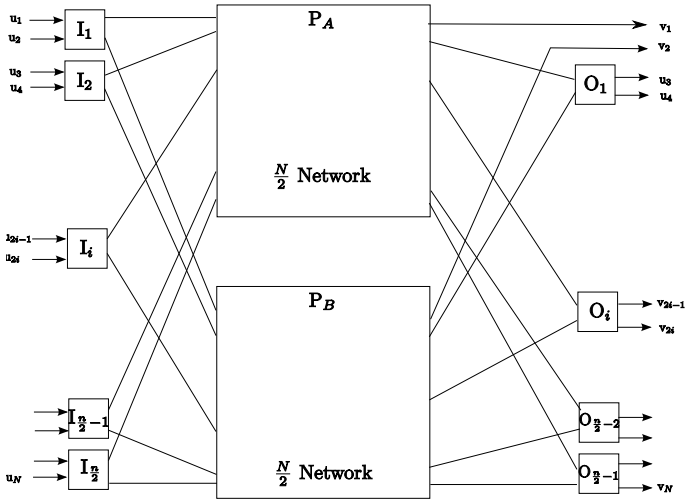


Figure 3.5: N Input Benes Network

3.3.1 Overview

The process of finding the switch settings for given input output settings is broken up into 3 tasks and is based on the work of Waksman [20].

- Obtain Permutation Matrix

  A permutation matrix is obtained, starting with setting the top right switch, that details which inputs and outputs are connected to the top $\frac{N}{2}$ permutation sub-block (P_A) and which are connected to the bottom $\frac{N}{2}$ permutation sub-block (P_B).

- Find Switch Settings

  The permutation matrix is used to find the settings of the elementary switches.

- Find Permutation Mapping for Upper and Lower Sub-blocks
  Knowing which inputs and outputs are connected to the upper and lower permutation sub-blocks, the process can be decomposed recursively. We start the process again on the upper block and then the lower block. This process repeats until we reach the elementary switch sub-block.

3.3.2 An Example

An explanation of the algorithm for setting the switches in a Benes netowrk is best served by an example:
Consider an eight input network with the required permutation $\pi = \{5, 4, 7, 1, 2, 8, 6, 3\}$. Referring to Figure 3.5, this means that input one, u_1 is connected to output 5 v_5, i_2 is connected to v_4 and so on.

To find the permutation matrix, we note that v_1 is connected to P_A, and from the given permutation, v_1 is connected to u_4. So we fill an 'A' in column 1, row 4. We also know that v_2 is connected to u_5 are and they are both connected to P_B (see figure 3.5) so we place a 'B' in column 2, row 5. On the input side, u_4 is connected to P_A and so u_3 must be connected to P_B so we place a 'B' in row 3 column 7 (as u_3 is v_7). On the output side, we know that v_7 is connected to P_B and so v_8 must be connected to P_A so we place an 'A' in column 8, row 6. Knowing u_6 is connected to P_A, u_5 must be connected to P_B.

Now we try to place an 'B' in row 5 column 2, but it is already there. There are 4 inputs i_1,i_2, i_7 and i_8, as well as 4 outputs v_3, v_4, v_5 and v_6 that are unassigned to P_A and P_B. This means that we can arbitrarily choose where to send one of the inputs/outputs, lets say i_1 to P_A. So we set row 1, column 5 to 'A' and repeat the process until we have the permutation matrix as shown in Table 3.1

We can now proceed to the second step, finding the switch settings for the elementary switches. To find the switches on the input side we look at the columns of the permutation matrix. I_1 is given by rows 1 and 2, I_2 is given by rows 3 and 4 and so on. To find I_1, we note that 'A' occurs in an odd row (1), so it is in the reset position. For I_2, 'A' occurs in an even row so it is in the set position. Similarly, I_3 and I_4 are also in the set position.

For the switches on the output side, we look at the rows of the permutation matrix. O_1 is given by columns 3 and 4, O_2 is given by columns 4 and 5 and O_3 is given by columns 6

u\v	1	2	3	4	5	6	7	8
1					A			
2				B				
3							B	
4	A							
5		B						
6								A
7						B		
8			A					

Table 3.1: Permutation Matrix

and 7 (columns 1 and 2 are not needed as there is no switch). 'A' appears in columns 3,5 and 8 which in the same manner as the inputs gives O_1 and O_2 being reset and O_3 being set. The configuration of the network is shown in Figure 3.6.

The third and final task is to find the permutation mapping for the upper and lower blocks. Start by grouping the permutation matrix into 2x2 squares. The permutation mapping for P_A is given by the location of 'A's in these squares, while P_B is given by the location of 'B's, as shown in Table 3.2. This gives $\pi_{P_A} = \{3, 1, 4, 2\}$ and $\pi_{P_B} = \{2, 4, 1, 3\}$. We now repeat steps to find the switch settings for the upper and lower permutation sub-blocks.

u\v	1	2	3	4
1			1	
2				
3	1			1
4		1		

u\v	1	2	3	4
1		1		
2				1
3	1			
4			1	

Table 3.2: Permutation Mappings for P_A and P_B

The recursive decomposition process ends when we reach the a sub-block containing only an elementary cell (2x2 block). If the permutation is $\pi = \{1, 2\}$, then the switch is in its reset position, otherwise its set. Figure 3.6 gives the final switch settings. Note that if the decision to set i_1 was made differently we would have a different set of switch settings, but they would still give the valid configuration.

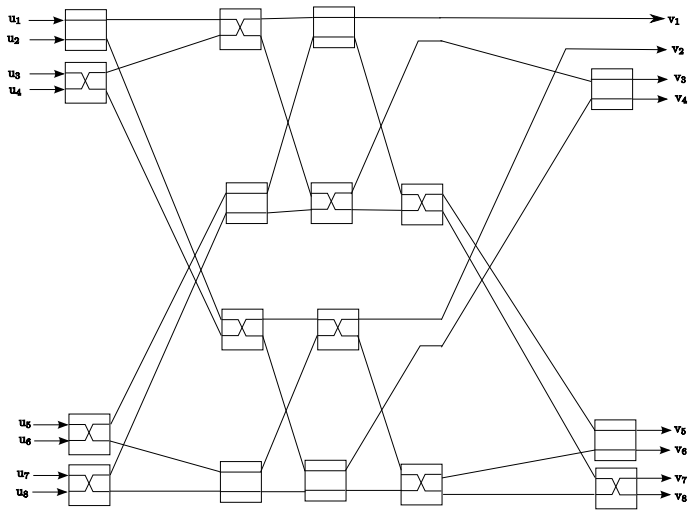


Figure 3.6: N Input Benes Network

3.4 Control Unit

The control unit is responsible for implementing the state machine as shown in figure 3.2, which involves setting the relevant control lines for the bit and check processors as well as the message permutation block. The configuration for the code is stored in a ROM block within the control unit, which contains the time varying settings for the Benes networks and the interleaver banks in the message permutation unit as well as the assignment of bit and check nodes to the respective processors. Although this implementation uses ROM for the configuration file, another non-volatile storage system such as flash memory could be used.

3.5 Flexibility

This decoder architecture, through the use of the message permutation block is flexible in that it allows any particular code to be used. To utilise a new code, the configuration is generated offline then downloaded to the ROM block within the control block with no change to the decoder design. The ROM block contains the time varying configuration of the Benes networks, the interleaver banks and the assignments of bit and check nodes to processors. The architecture also supports irregular codes with some constraints.

Chapter 4

Scheduling

LDPC codes have a high degree of parallelism in the decoder, each node can work in parallel with others of the same kind. With a higher degree of parallelism, there is faster decoding, at the expense of of higher complexity.

In a partially parallel design, there are P check and P bit node processors, each working in parallel. The main issue with this design is handling how memory is accessed. Consider the half iteration when the bit nodes are active. To ensure that each check node processor can receive the correct message every cycle, the outgoing messages from the bit nodes must be permuted in both time (interleavers) and space (crossbar switches).

If there are P processors then there will need to be P simultaneous accesses to memory to read the input and write the output. There must therefore be at least P memory banks so that memory can be accessed without collisions. A pictorial view of this is shown in figure 4.1. A correct scheduling (generally there is more then one), ensures that there are no collisions in the memory accesses for reading and writing. If there are L edges in total, the set of P processors will need to be accessed $\lceil L/P \rceil$ times to complete a half-iteration. Each access requires a schedule defining which edge variables are needed by each processor and in which order those edge variables are read in.

The difficulty with LDPC codes is that the scheduling for reading and writing to the memory at the variable node half iteration is different to the scheduling for the check node half iteration. This means that a different set of scheduling constraints arise for the two half iterations. The scheduling defines which variables are read simultaneously, and also

dictates which variables must be placed into the different memory banks.

The scheduling gives the time varying configurations for the crossbars and the interleaver bank as seen in figure 4.3.

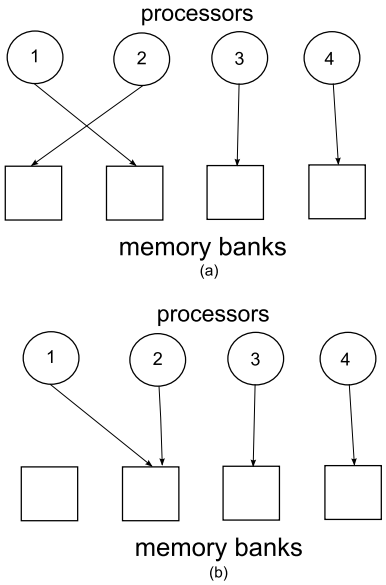


Figure 4.1: (a) No collisions. (b) A collision occurs.

4.1 Algorithm

4.1.1 Overview

The first task is to decide which bit (check) nodes will operate in parallel. The parity check matrix H, is represented as a Tanner graph, see figure 4.2. This allows the edges to be numbered. The Tanner graph is partitioned, placing the bit vertices into P sets, with an equal number of edges in each set, and the check vertices into P sets, again with an equal number of edges in each set. The sets represent the group of bit (check) nodes which are assigned to each bit (check) processor. One node from each set will run in parallel. Figure 4.5 shows a partitioned Tanner graph.

The choice of partitioning will affect the performance of the decoder, so a graph partitioning package, METIS [21], is used which minimises the number of intersections between partitions. Experimental results have shown that graph partitioning can decrease the half

cycle iteration time, on average by 20% [7] by reducing the number of potential memory collisions.

The second task is to graph edge variables to the memory banks. A mapping function is then generated, mapping the edges of the Tanner Graph to the memory banks. The aim of the mapping function is to avoid the memory collisions when each of the bit (check) nodes acting in parallel all access memory at the same time.

The third and final task is to use the mapping function to determine the configuration required for the crossbar switches and the interleaver banks. These tasks are outlined in more detail in the following.

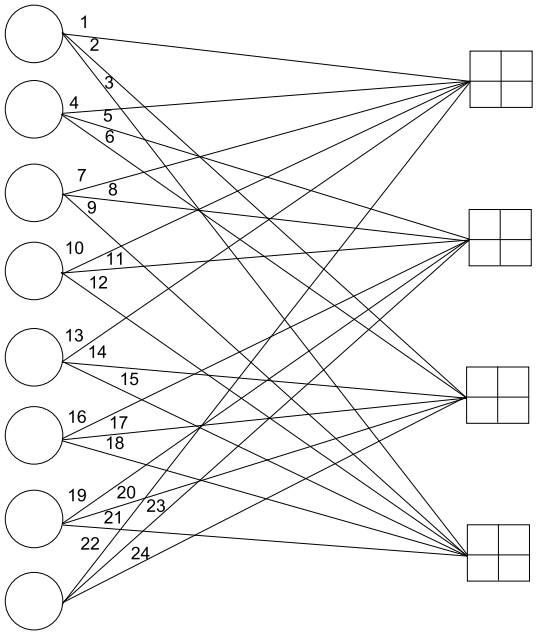


Figure 4.2: Numbered Tanner graph

4.1.2 Notation

Throughout the remainder of the chapter the following symbols are used.

H, denotes the parity check matrix of the code.

L, denotes the numbers of edges in the code.

P, denotes the number of processors operating in parallel.

$M \triangleq \lceil \frac{L}{P} \rceil$, represents the number of parallel operations to complete a half cycle.

$\mathcal{P}_1$, the set mapping bit nodes to processors.

$\mathcal{P}_2$, the set mapping check nodes to processors.

$\mathcal{V}_i$, a subset of $\mathcal{P}_1$ containing the edges that are being processed in parallel.

$\mathcal{V}_i^{'}$, a subset of $\mathcal{P}_2$ containing the edges that are being processed in parallel.

$\mathcal{M}$, the function mapping edges to memory banks.

Vp_j, the jth bit node processor.

Cp_j, the jth check node processor.

$\beta_j(i)$, the time varying configuration of the left most crossbar switch.

$\beta_j^{'}(i)$, the time varying configuration of the right most crossbar switch.

$\delta_i(j)$, the time varying configuration of the ith interleaver bank.

4.1.3 Mapping Function Formal Procedure

The following procedure is based the work of Tarable et al [1]. Consider a set of L elements (edges) $\mathcal{V} = \{v_1, ...v_L\}$. We then have two partitions (not to be confused with the graph partitioning)

$$\mathcal{P}_1 = \{\mathcal{V}_1, ..., \mathcal{V}_M\}$$

and

$$\mathcal{P}_2 = \{\mathcal{V}_1^{'}, ..., \mathcal{V}_M^{'}\}$$

with $\mathcal{P}_1$ containing the assignments of the edges to the bit node processors and $\mathcal{P}_2$ containing the assignments of the edges to the check node processors. Where $\mathcal{V}_i(j)$ contains the edge that is being processed by the jth bit node processor at time i. $\pi(k), k \in \{1, ..., L\}$, is a permutation matrix from $\mathcal{P}_1$ to $\mathcal{P}_2$. All the subsets $\mathcal{V}_i, \mathcal{V}_i^{'}, i = 1, ..., M$, have the same number of elements $P \triangleq L/M$, where P is the number of parallel bit and node processors and L is the number of edges in the code. The subsets contain the edges that are being accessed at the same time.

In the case that M is not a divisor of L, extra, dummy edges are introduced into $\mathcal{V}, \mathcal{P}_1$ and $\mathcal{P}_2$. Although, ultimately, these dummy edges will be instantiated on the decoder, they do not effect its operation as they are simply ignored. Given $\mathcal{P}_1$ and $\mathcal{P}_2$, a mapping function is defined to map the L edge variables to the P memory banks so that for all i in M, each of the edges in $\mathcal{V}_i$ is in a separate memory bank.

A mapping function $\mathcal{M} : \{1, ..., L\} \rightarrow \{1, ..., P\}$ is defined as a mapping function for $(\mathcal{P}_1, \mathcal{P}_2)$ if it satisfies the following conditions for every $j, j^{'} = 1, ..., L, j \neq j^{'}$ [1].

$$v_j, v_{j'} \in \mathcal{V}_i \text{ for some } i \Rightarrow \mathcal{M}(j) \neq \mathcal{M}(j^{'}) \tag{4.1}$$

$$v_j, v_{j'} \in \mathcal{V}_i^{'} \text{ for some } i \Rightarrow \mathcal{M}(j) \neq \mathcal{M}(j^{'}) \tag{4.2}$$

or equivalently, elements belonging to the same subset in either partition are mapped to different memory banks.

The procedure to find a valid mapping function M, is made up of several cycles, each one starting with a blank in the mapping function and ending when no collisions are produced (collisions are defined as a mapping that does not satisify conditions 4.1 and 4.2). At the conclusion of a cycle, if there are no more blanks the procedure is over, otherwise another blank is chosen.

Let $L(\mathcal{V}_i)$ (resp. $L(\mathcal{V}_i^{'})$) be the set of values $\{1, ..., P\}$ that are not yet assigned to $\mathcal{V}_i$ (resp., $\mathcal{V}_i^{'}$).

At the beginning of a cycle, a preliminary mapping function $\mathcal{M}^{(0)}$ maps $\{1, ...L\}$ into the set $\{1, ...P\} \cup \{\emptyset\}$. $(\mathcal{M}^{(0)})^{-1}(\emptyset)$ is the set of blanks. At the ith step of the cycle, an updated preliminary mapping function $\mathcal{M}^{(i)}$ is produced. Where the apices (0),(1)... refer to the variables in the corresponding step of the cycle [1].

At the beginning of the cycle, a blank is identified in position $j^{(0)} \in \mathcal{V}_{i^{(0)}} \cap \mathcal{V}_{i^{(1)}}^{'}$. If the intersection between $L(\mathcal{V}_{i^{(0)}})$ and $L(\mathcal{V}_{i^{(1)}}^{'})$ is non empty, then there is a value that fills the blank without creating a collision. $\mathcal{M}^{(1)}(j^{(0)})$ will be found in the intersection, while for $j \neq j^{(0)}$, $\mathcal{M}^{(1)}(j) = \mathcal{M}^{(0)}(j)$

If, however, the intersection between $L(\mathcal{V}_{i^{(0)}})$ and $L(\mathcal{V}_{i^{(1)}}^{'})$ is empty, then pick $m \in L(\mathcal{V}_{i^{(0)}})$ and $n \in L(\mathcal{V}_{i^{(1)}}^{'})$. Assign $\mathcal{M}^{(1)}(j^{(0)}) = m$, while for $j \neq j^{(0)}$, $\mathcal{M}^{(1)}(j) = \mathcal{M}^{(0)}(j)$. Then repeat the following steps for $k = 1, 2, ...$:

1. Search for

   $$j^{(2k-1)} \in \mathcal{V}_{i^{(2k-1)}}^{'} \cap \mathcal{V}_{i^{(2k)}}$$

   such that $\mathcal{M}^{(2k-2)}(j^{(2k-1)}) = m$ and $j^{(2k-1)} \neq j^{(2k-2)}$.
   If such a $j^{(2k-1)}$ is found,
   $\mathcal{M}^{(2k-1)}(j^{(2k-1)}) = n$ and $\mathcal{M}^{(2k-1)}(j) = \mathcal{M}^{(2k-2)}(j)$, for $j \neq j^{(2k-1)}$, and proceed to step 2. Otherwise stop the cycle. [1]

2. Search for

$$j^{(2k)} \in \mathcal{V}_{i^{(2k)}} \cap \mathcal{V}'_{i^{(2k+1)}}$$

such that $\mathcal{M}^{(2k-1)}(j^{(2k)}) = n$ and $j^{(2k)} \neq j^{(2k-1)}$.
If such a $j^{(2k)}$ is found,
$\mathcal{M}^{(2k)}(j^{(2k)}) = m$ and $\mathcal{M}^{(2k)}(j) = \mathcal{M}^{(2k-1)}(j)$, for $j \neq j^{(2k)}$, and proceed to step 1.
Otherwise stop the cycle. [1]

At the conclusion of a cycle, the sets $L(\mathcal{V}_i)$ and $L(\mathcal{V}'_i)$ $\forall i = 1, ..., M$ must be updated.

4.1.4 Circuit Configuration

The mapping function is then used to determine the configuration of the three blocks used for the variable node half-iteration, as seen in figure 4.3.

The left-most block is a crossbar (Benes Network), with P input and P output pins used to implement the mapping function found earlier. The ith input pin at time j is connected to the $\beta_j(i)$ output pin. Where $\beta_j(i) = \mathcal{M}((i-1)M + j)$ [1].

The middle blocks are interleaver banks which arrange the information in the order needed by the processors on the right hand side based on the permutation π. δ_i is defined by considering the set of M indices k in $\{1, ...L\}$, such that $\mathcal{M}(k) = i$, called $\mathcal{M}^{-1}(i)$. Given a $k \in \mathcal{M}^{-1}(i)$, let j be the integer in $\{1, ..M\}$ for which $j =_M k$ and j' be the integer in $\{1, ..M\}$, for which $j' =_M \pi^{-1}(k)$. Where $_M$ is equal modulo M. Then, $\delta_i(j') = j$. [1]

The right-most block is another crossbar switch, identical to to the left-most one. The ith output pin at at time j is connected to the $\beta'_{j(i)}$ input pin. Where $\beta'_j(i) = \mathcal{M}(\pi((i-1)M+j))$.

In the other half-iteration, where information travels from the check node to the variable node, the order of the blocks is inverted. The blocks' functions are also inversed, as can be seen in figure 4.4. For the left-most crossbar, the ith output pin at time j is connected to the $\beta'_{j(i)}$ input pin. The interlever is given by δ_i^{-1}, and the for the right-most crossbar, the ith input pin at time j is connected to the $\beta_j(i)$ output pin.

The above algorithm is only valid when $\mathcal{P}_1$ is in numerical order. When METIS is used, it is highly unlikely that this will be the case, so the edges are renumbered to ensure that $\mathcal{P}_1$ is in numerical order.

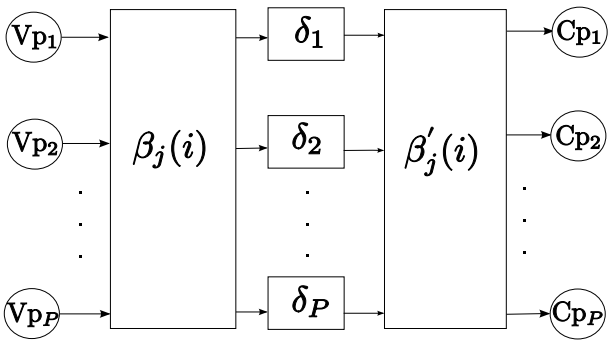


Figure 4.3: Configuration for variable node half-iteration

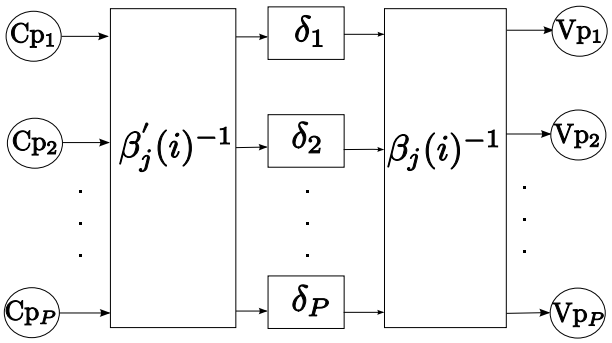


Figure 4.4: Configuration for check node half-iteration

4.2 An Example

Consider the (3,6)-regular (8,4) rate 0.5 code, given by the parity check matrix 4.3. In this example we will use 4 processors and there are 24 edges, so that gives:

$L = 24$

$P = 4$

$M = \frac{L}{P} = 6$

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{4.3}$$

There are $L = 24$ edges and $P = 4$ processors, giving $M = 6$ edges per processor. Figure 4.5 shows the Tanner graph representation of 4.3, partitioned by hand.

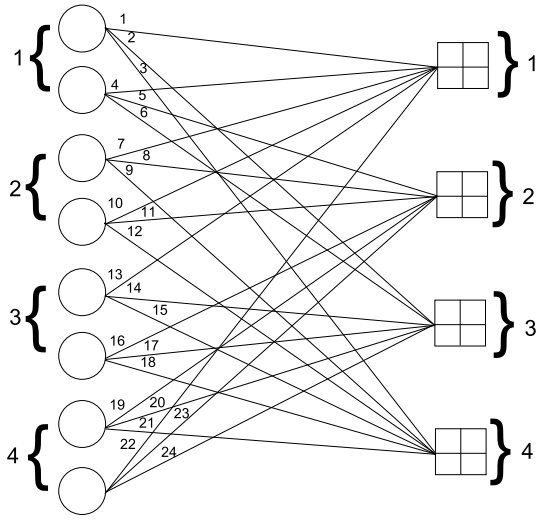


Figure 4.5: Partitioned Tanner Graph

To find $\mathcal{P}_1 = \{\mathcal{V}_1, ..., \mathcal{V}_M\}$, we note that $\mathcal{V}_i(j)$ contains the edge that is being processed by the jth bit node processors at time i. i.e. at time i=1, processor 1 is using edge 1, processor 2 is using edge 7, processor 3 is using edge 13 and processor 4 is using edge 19. So $\mathcal{V}_1 = (1, 7, 13, 19)$, continuing for all i gives,

$$\begin{aligned} \mathcal{P}_1 = \{&(1,7,13,19)\,,(2,8,14,20)\,,(3,9,15,21)\,, \\ &(4,10,16,22)\,,(5,11,17,23)\,,(6,12,18,24)\}. \end{aligned} \tag{4.4}$$

To find $\mathcal{P}_2 = \{\mathcal{V}'_1, ...\mathcal{V}'_M\}$ we note that $\mathcal{V}'_i(j)$ contains the edges that are being processed by the jth check node processors at time i. i.e. at time i=1, processor 1 is using edge 1, processor 2 is using edge 5, processor 3 is using edge 2 and processor 4 is using edge 3. This gives

$$\begin{aligned} \mathcal{P}_2 = \{&(1,5,2,3)\,,(4,8,6,9)\,,(7,11,14,12)\,, \\ &(10,16,17,15)\,,(13,19,20,18)\,,(22,23,24,21)\}. \end{aligned} \tag{4.5}$$

Now it can be seen that the permutation (π) mapping from $\mathcal{P}_1$ to $\mathcal{P}_2$ is:

$$\pi = (1, 4, 7, 10, 13, 22, 5, 8, 11, 16, 19, 23, 2, 6, 14, 17, 20, 24, 3, 9, 12, 15, 18, 21), \tag{4.6}$$

and its inverse π^{-1} is

$$\pi^{-1} = (1, 13, 19, 2, 7, 14, 3, 8, 20, 4, 9, 21, 5, 15, 22, 10, 16, 23, 11, 17, 24, 6, 12, 18). \quad (4.7)$$

It is useful to represent the mapping function $\mathcal{M}$ as a $P \times M$ matrix, whose (i, j)th element is:

$$\mathcal{M} = \begin{pmatrix} \mathcal{M}(1) & \mathcal{M}(2) & ... & \mathcal{M}(M) \\ \mathcal{M}(M+1) & \mathcal{M}(M+2) & ... & \mathcal{M}(2M) \\ ... & ... & ... & ... \\ \mathcal{M}(3M+1) & \mathcal{M}(3M+2) & ... & \mathcal{M}(4M) \end{pmatrix}. \quad (4.8)$$

We start with $\mathcal{M}$ consisting of blanks, $L(\mathcal{V})$ and $L(\mathcal{V}')$ containing $\{1, ..., 4\}$ for all i. Starting in the first position with $j^{(0)} = 1$, we can see that edge 1 occurs in $\mathcal{V}_1$ and $\mathcal{V}_1'$. $L(\mathcal{V}_1) \cap L(\mathcal{V}_1') = \{1, 2, 3, 4\}$, picking the first element we have $\mathcal{M}(1) = 1$ and

$$\text{Ł}(\mathcal{V}) = \{(2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)\}.$$

$$\text{Ł}(\mathcal{V}') = \{(2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)\}.$$

Continuing with the next blank, $j^0 = 2 \in \mathcal{V}_2 \cap \mathcal{V}_1'$. We have
$L(\mathcal{V}_1) \cap L(\mathcal{V}_1') = \{2, 3, 4\}$, and again taking the first element we have $\mathcal{M}(2) = 2$ and

$$\text{Ł}(\mathcal{V}) = \{(2, 3, 4), (1, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)\}.$$

$$\text{Ł}(\mathcal{V}') = \{(3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)\}.$$

This process continues until we have the following:

$$\mathcal{M} = \begin{pmatrix} 1 & 2 & 3 & 1 & 4 & 2 \\ 2 & 3 & 4 & 2 & 1 & 3 \\ 3 & 4 & 1 & 3 & - & - \\ - & - & - & - & - & - \end{pmatrix}$$

with,

$$Ł(\mathcal{V}) = \{(4), (1), (2), (4), (2,3), (1,4)\}.$$

$$Ł(\mathcal{V}') = \{\emptyset, \emptyset, \emptyset, (4), (1,2,4), (1,2,3,4)\}.$$

With $j^{(0)} = 17 \in \mathcal{V}_5 \cap \mathcal{V}_4'$,we have $L(\mathcal{V}_5) \cap L(\mathcal{V}_4') = \emptyset$. So taking the first element of $L(\mathcal{V}_5)$, we have $m = 2$, and similarly $n = 4$. We set $\mathcal{M}(17) = 2$ and proceed to the first step in the iterative part of the algorithm described in section 4.1.3.

We search for $j^{(1)} \in \mathcal{V}_4' \cap \mathcal{V}_{i^{(2)}}$ such that $\mathcal{M}(j^{(1)}) = m$ and $j^{(1)} \neq j^{(0)}$. The possible candidates are $(10, 16, 15)$, and we find that $j^{(1)} = 10$ satisfies the conditions with $i^{(2)} = 4$. We then set $\mathcal{M}(10) = n$. Now we proceed to the second step in the algorithm.

We search for $j^{(2)} \in \mathcal{V}_4 \cap \mathcal{V}_{i^{(3)}}'$ such that $\mathcal{M}(j^{(2)}) = n$ and $j^{(2)} \neq j^{(1)}$. The possible candidates are $(4, 16, 22)$, and we find that none of them satisfy the conditions. So the cycle ends and we procede to the next blank in the mapping function after updating $\mathcal{M}, L(\mathcal{V})$ and $L(\mathcal{V}')$. If we had found a candidate $(j^{(2)})$, we would set $\mathcal{M}(j^{(2)}) = m$ and proceed to back to step 1 of the algorithm.

We now have:

$$\mathcal{M} = \begin{pmatrix} 1 & 2 & 3 & 1 & 4 & 2 \\ 2 & 3 & 4 & 4 & 1 & 3 \\ 3 & 4 & 1 & 3 & 2 & - \\ - & - & - & - & - & - \end{pmatrix}$$

and,

$$Ł(\mathcal{V}) = \{(4), (1), (2), (2), (3), (1,4)\}.$$

$$Ł(\mathcal{V}') = \{\emptyset, \emptyset, \emptyset, \emptyset, (1,2,4), (1,2,3,4)\}.$$

Following this algorithm to its completion yields:

$$\mathcal{M} = \begin{pmatrix} 1 & 2 & 3 & 2 & 4 & 1 \\ 2 & 3 & 4 & 4 & 1 & 3 \\ 3 & 4 & 1 & 3 & 2 & 2 \\ 4 & 1 & 2 & 1 & 3 & 4 \end{pmatrix}$$

$\beta_j(i)$ is given by the (i,j)th element of $\mathcal{M}$. To find $\beta_j^{'}(i)$, we construct the matrix whose columns correspond to the subsets $\mathcal{V}^{'}$ of $\mathcal{P}_2$(equation 4.5):

$$\beta_j^{'}(i) = \begin{pmatrix} \mathcal{M}(1) & \mathcal{M}(4) & \mathcal{M}(7) & \mathcal{M}(10) & \mathcal{M}(13) & \mathcal{M}(22) \\ \mathcal{M}(5) & \mathcal{M}(8) & \mathcal{M}(11) & \mathcal{M}(16) & \mathcal{M}(19) & \mathcal{M}(23) \\ \mathcal{M}(2) & \mathcal{M}(6) & \mathcal{M}(14) & \mathcal{M}(17) & \mathcal{M}(20) & \mathcal{M}(24) \\ \mathcal{M}(3) & \mathcal{M}(9) & \mathcal{M}(12) & \mathcal{M}(15) & \mathcal{M}(18) & \mathcal{M}(21) \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 2 & 2 & 4 & 3 & 1 \\ 4 & 3 & 1 & 3 & 4 & 3 \\ 2 & 1 & 4 & 2 & 1 & 4 \\ 3 & 4 & 3 & 1 & 2 & 2 \end{pmatrix}$$

To determine δ_1 we find $k \in \mathcal{M}^{-1}(1)$, giving $k = \{1, 6, 11, 15, 20, 22\}$. Remembering equation 4.7 and starting with $k = 1$ we have

$j =_M 1 = 1$

$j^{'} =_M \pi^{-1}(1) =_M 1 = 1,$

so we have $\delta_1(j^{'}) = j$. Continuing with $k = 6$ we have

$j =_M 6 = 6$

$j^{'} =_M \pi^{-1}(6) =_M 14 = 2,$

so we have $\delta_1(2) = 6$. Following this process to completion yields:

$$\delta_1 = (1, 6, 5, 3, 2, 4)$$

$$\delta_2 = (2, 4, 1, 5, 6, 3)$$

$$\delta_3 = (3, 2, 6, 4, 1, 5)$$

$$\delta_4 = (5, 3, 2, 4, 1, 6)$$

So now we have the necessary parameters for the blocks when the variable nodes are active as shown in figure 4.3. When the check nodes (see figure 4.4) are active δ_i is replaced with δ_i^{-1}. The $\beta_j(i)$ and $\beta_j^{'}(i)$ are swapped with this important difference. When the check nodes are active, $\beta_j(i)$ specifies that the ith *output port* is connected to the $\beta_j(i)$th *input port* at time j. This is the opposite of what occurs when the variable nodes are active, so to keep the configuration of the crossbars consistent (ie the *input* is i and the *output* is $\beta(i)$, we

say that the parameter for the left-most crossbar is $\beta_j^{'}(i)^{-1}$ and the parameter for the right most crossbar is $\beta_j(i)^{-1}$, as shown below for $\beta_j(i)^{-1}$.

$$\beta_j(i)^{-1} = \begin{pmatrix} 1 & 4 & 3 & 4 & 2 & 1 \\ 2 & 1 & 4 & 1 & 3 & 3 \\ 3 & 2 & 1 & 3 & 4 & 2 \\ 4 & 3 & 2 & 2 & 1 & 4 \end{pmatrix}$$

Chapter 5

MATLAB® Software

MATLAB® has been used to develop software which accepts a code defined by its parity check matrix (H), and produces ROM files defining the scheduling for the decoder. The ROM files can then be downloaded onto the decoder, allowing it to implement a new LDPC code.

MATLAB® was also used to develop a test bench for software verification of the decoder's operation.

5.1 Matrix to Tanner Graph

The code shown in Appendix A.1, matrix2graph.m converts a parity check matrix to a Tanner graph and saves the result to disk in preparation for graph partitioning.

5.2 Graph Partitioning

The graph file resulting from the previous section is passed to METIS [21], a graph partitioning package. The graph is partitioned into P (the number of processors running in parallel on the decoder) partitions, minimising the number of intersections between partitions. This can reduce the potential number of collisions in the scheduling and speed the decoder performance by 20% on average [7]. The output file from METIS must be converted into two sets ($\mathcal{P}_1$ and $\mathcal{P}_2$) containing the assignment of nodes to processors during the bit and check half-iterations respectively. Appendix A.2 shows plist2part.m which is

responsible for this process.

METIS is not optimised for bipartite graphs, and will produce partitions that do not contain an equal number of edges in both the bit and check node half iterations. As a consequence, plist2part.m attempts to even the partitions up at the expense of a small increase in intersections between partitions.

If the number of edges in the graph is not an integer divisor of of the number of parallel processors, then dummy edges must be introduced to satisfy this condition. These edges will be instantiated on the decoder (they will be passed between the bit and check nodes through the message permutation block), but they won't affect its operation as the decoder knows the how many edges are connected to every node through the ROM configuration blocks. It is in this step that the dummy edges (if needed) are added.

5.3 Scheduling

The next task is to find a mapping function as defined in Chapter 4. Appendix A.3 shows the code responsible for this. The mapping function maps each edge of the code's Tanner graph to an interleaver bank, ensuring that there they can be accessed in parallel without collisions. Once a valid mapping function is found (satisfying equations 4.1 and 4.2), it is used to determine the configuration for the crossbar switches and and interleaver banks.

The Scheduling that was found in this section needs further processing. It is implementation neutral, in that it defines the crossbar's input and output without considering how the crossbar might implement it. It also only considers the bit node half-iteration.

There is a simple relationship between the bit node half-iteration and the check node half-iteration scheduling which is explained in Section 4.1.4. It is implemented as the switches are set in the following section. To generate the output of the decoder as per Equation 1.9, the message permutation block is used again to obtain a hard decision codeword. This requires a new, trivial scheduling explained in the following.

The output of the decoder will be sent from the bit node processors, through the 1st Benes network and into the interlever banks. During this process the left most Benes network is set so to the identity permutation (the nth input is connected to the nth output). This means that the nth interleaver bank contains the codebits associated with the nth bit node processor. Now all that remains is to set the interleaver banks on the right most switch

so that the codebits appear on the output of the 1st output of the message permutation block, in the order to produce the the decoded codeword. The other outputs of the message permutation block are unused. These switch settings are appended to the ones already found.

5.4 Benes Network Configuration

The files benes-main.m, benes.m, forward.m, backward.m and inverse.m in Appendix A.4 implement the Benes Network switch setting as described in Section 3.3. The switch settings output is converted to hexadecimal to eliminate any potential precision inaccuracies when converting from binary to decimal.

There are two recursive components in setting the switches, forward.m sets a matrix entry on the input side and calls backward.m which sets the corresponding matrix entry on the output side. If there is another switch to set backwards.m calls forward.m and the process repeats. The second recursive part is benes.m, which is decomposes the switch into sub-blocks and repeatedly calls itself until an elementary switch is reached, while setting the outside switches.

5.5 ROM Definition Files

After the scheduling is defined and the switches are set, the last remaining task in obtaining configuration files for a particular code is to save them to ROM definition files to be used by the hardware programmer. Appendix A.5 shows how print-files.m achieves this. The code saves configuration in the format used by Altera's QUARTUS II software.

5.6 Test bench

In order to verify the output of the decoder, a Matlab test bench was created which implements the sum-product algorithm, see Appendix A.6. It uses the same lookup table as the decoder, the only difference between the decoder and the Matlab version is the ordering of the addition operations. This may produce a difference between the hardware and matlab results but it is expected to be small.

Matlab's serial interface is used to communicate with the decoder over a serial link compare the simulation results with hardware.

Chapter 6

Flexible LDPC Decoder Hardware

The flexible LDPC decoder has been designed in a mixture of VHDL and schematic capture in Altera's Quartus II 7.0. A proof of concept (POC) was first designed and following its success a prototype to gather results. Both the proof of concept and the prototype have been targeted at Altera's DE2 Cyclone II development board. The POC has only 4 parallel processing units and only supports a code size of 8x4. While the prototype can implement any (3,6)-regular LDPC code up to size 480x240 and supports 8 parallel processing units. The prototype differs from the full design by having less parallel bit and check node processing units (8 vs 32), as well as supporting a smaller maximum code size, 480x240 (3,6)-regular. The prototype has serial RS-232 input/output (IO) communications while it is planned that the full design will utilise ethernet IO communications.

The full design will be targeted at the Stratix II development board, and is planned to support codes of size up to 1920x960 and irregular codes. Having four times the number of parallel bit and check node processors, it will be four times as fast per clock cycle as the prototype.

The flexibility of the design, for both the prototype and the full design, allows a new code to be decoded by only changing the ROM configuration files.

Figure 6.1 shows the block diagram of the prototype decoder.

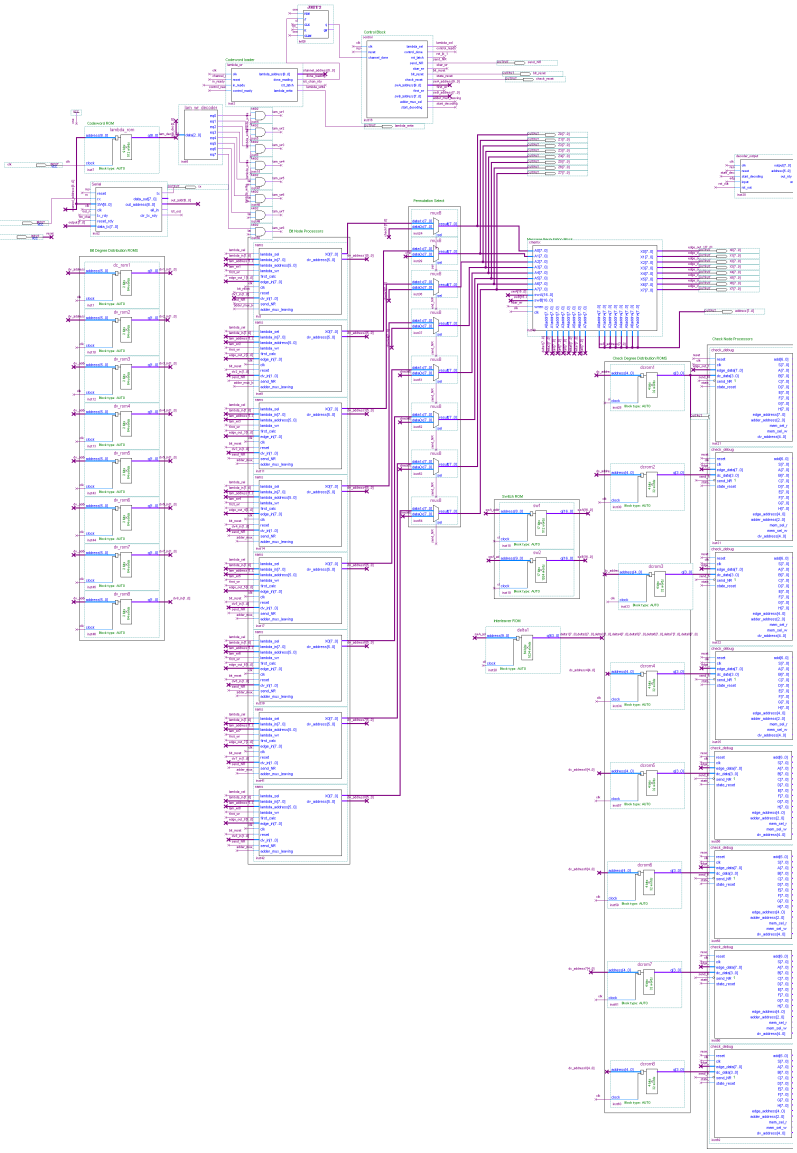


Figure 6.1: Block diagram of the decoder

6.1 Potential Implementation Issues

The first step in designing the hardware implementation of the LDPC decoder was to investigate the potential implementation issues. The first task was to investigate and to decide on the representation of numbers within the decoder, as their representation will directly influence how every component of the decoder design.

6.1.1 Number representation

The numbers that are represented on the hardware are the messages moving through the sum product algorithm. In the algorithm the numbers are real, but instantiating multiple floating point units to deal with addition is not practical, so a compromise between numerical accuracy and design complexity was made and fixed point decided on. The range of numbers is both positive and negative, so a two's compliment number system is used. This means that only one type of adder is needed as adding a mix of positive and negative numbers in two's compliment is not a special case.

Numerical accuracy affects the operation of the decoder, especially since it is iterative, although errors may start small, over a number of iterations they have the potential to accumulate. It has been found that between 5 and 8 bits of precision are needed for for an LDPC decoder to meet target BER standards [15, 16]. An eight bit two's compliment representation with a 4 bit fractional part was decided on. This gives a range of -8.000 to +7.9375 with increments of 0.0625. The range will be restricted on the negative side to -7.9375, to eliminate a possible bias in the decoder for negative numbers.

6.1.2 Multiplication and Trigonometric Functions

The two operations of the sum product algorithm seen in Section 1.3.3, are reproduced here for clarity.

$$Q_{ij} = \lambda_j + \sum_{\alpha \in C[j]\ \alpha \neq i} R_{\alpha j}[k-1] \tag{6.1}$$

$$R_{ij} = 2\tanh^{-1} \prod_{\alpha \in V[j] \alpha \neq i} \tanh\left(\frac{Q_{\alpha j}}{2}\right) \tag{6.2}$$

It can be seen that Equation 6.1 contains only additions which are easy to implement in hardware, while Equation 6.2 contains both multiplication and hyperbolic trigonometric

functions. While hyperbolic trigonometric functions can be implemented on hardware [22], the number that would be required for this design is prohibitive. The number of multipliers needed would also be prohibitive. Fortunately there is a solution to both the hyperbolic tangent and the multipliers problem. Introduce a function $\psi(x)$ such that:

$$\psi(x) = -\ln\left(\tanh\left|\frac{x}{2}\right|\right) = \ln\frac{\left(1+e^{-|x|}\right)}{\left(1-e^{-|x|}\right)}. \tag{6.3}$$

Now as $\tanh(x)$ is an even function,

$$\tanh\frac{x}{2} = \tanh\left|\frac{x}{2}\right|.\delta_{ij} \tag{6.4}$$

where δ_{ij} is the product of sign of the messages.
Substituting Equation 6.3 into 6.2 gives:

$$R_{ij}[k] = 2\tanh^{-1}\left(\exp\left(-\sum_{\alpha\in V[j]\alpha\neq i}\psi(Q_{\alpha j}[k-1])\right).\delta_{ij}\right) \tag{6.5}$$

Noting that

$$\tanh^{-1}(x) = \frac{1}{2}\ln\left(\frac{1+x}{1-x}\right)$$

and $\tanh^{-1}(x)$ is even, we can pull δ_{ij} outside and simplify.

$$R_{ij}[k] = \ln\frac{1+\exp\left(-\sum_{\alpha\in V[j]\alpha\neq i}\psi(Q_{\alpha j}[k-1])\right)}{1-\exp\left(-\sum_{\alpha\in V[j]\alpha\neq i}\psi(Q_{\alpha j}[k-1])\right)}.\delta_{ij}$$

$$R_{ij}[k] = \psi\left(\sum_{\alpha\in V[j]\alpha\neq i}\psi(Q_{\alpha j}[k-1])\right) \tag{6.6}$$

So now we have eliminated the multiplication within the decoder, replacing it with addition and introducing a new function ψ. The details of the implementation will be discussed in the following section.

6.1.3 Non Linear Function ψ

The function ψ is highly non-linear and not limited due to the vertical asymptote at $x = 0$. Three types of implementations that have been discussed in literature were found [7, 23],

a Piece-wise Linear (PWL) implementation, an approximation using base 2 arithmetic and a direct Lookup Table (LUT). It was found that the PWL implementation was able to perform better then the base 2 approximation with 3 mantissa bits and 4 fractional bits [7].

We compared two options, a PWL implementation and a LUT implementation. The PWL implementation based on [7]. The coefficients were designed to be easily represented in fixed point notation in order avoid multiplications, instead a series of shifts and additions were used.

The LUT was based on [23] with 256 table entries. Upon building and compareing the PWL and the LUT implementations it was found that the PWL was very expensive in terms of resources almost prohibitively so compared to a direct LUT implementation, the PWL occupied 76% more space and was 8% slower [23].

It was decided to implement ψ with a LUT as shown in B.1.

6.2 Adders

For the LDPC decoder, there is a need to add multiple input messages in parallel. The number of inputs to the adder dictates the maximum supported degree weight of the bit and check nodes. The maximum degree weight for the check nodes is the number of inputs into the adder, while for the the bit nodes it is one less, due to one of the inputs being used for the incoming channel measurement λ as per Equation 6.1.

For the prototype it was decided to to support a maximum degree weight of 3 for the bit nodes and 6 for the check nodes. It was decided to handle the parallel inputs with tree adders, as opposed to carry save adders, due to the complexity of the latter and the small number of operands. Ripple, carry look ahead and carry select adders were investigated and ranked on their performance and complexity. It was found that the carry select adder was the fastest and most complex (and consumed the most power), while ripple adder was the slowest but least complex [24]. The carry look ahead adder was the best compromise between speed and complexity and so it was chosen for the decoder.

The carry lookahead adder calculates the carry signals in advance, based on the input signals. The implementation of the carry lookahead adder is based on the VHDL code provided in [24] as seen in Appendix B.2.

In this design there is no traditional processor, so there is no checking for error conditions

such as overflow. This means that errors in addition will go unchecked, for example $4.5 + 4.5 = -7$, is a serious error. To eliminate this error the output of the adder is hard-limited if it goes beyond its range in either the positive or negative direction. The output of each adder is checked for overflow and if it has occurred, the output is set to the positive or negative limit 7.9375 as appropriate. Figure 6.2 shows an eight input tree adder using carry look ahead adders.

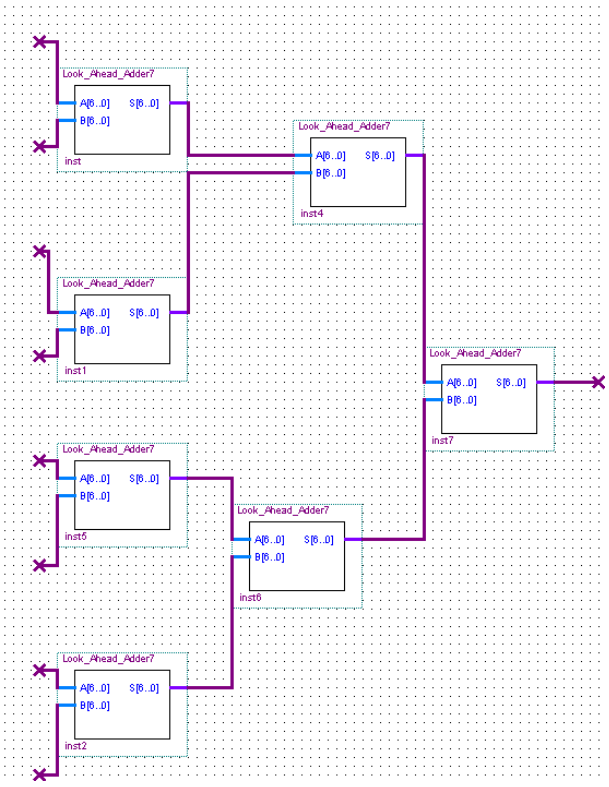


Figure 6.2: 8 input tree adder

6.3 Bit Node Processor

The bit node processor is responsible for receiving messages from the check node through the message permutation block, processing the messages and outputting them to the message permutation block. It is important to note that the bit node processor is responsible for keeping track of the individual bit nodes in the code. From the perspective of the LDPC

control unit, the bit nodes send a stream of messages with no distinction as to which belong to what bit node.

The bit node processor has a control signal that controls its function, reading messages into the message RAM or sending messages. The signal only has effect when the processor is coming out of reset. Figure 6.3 shows the block diagram of the bit node processor.

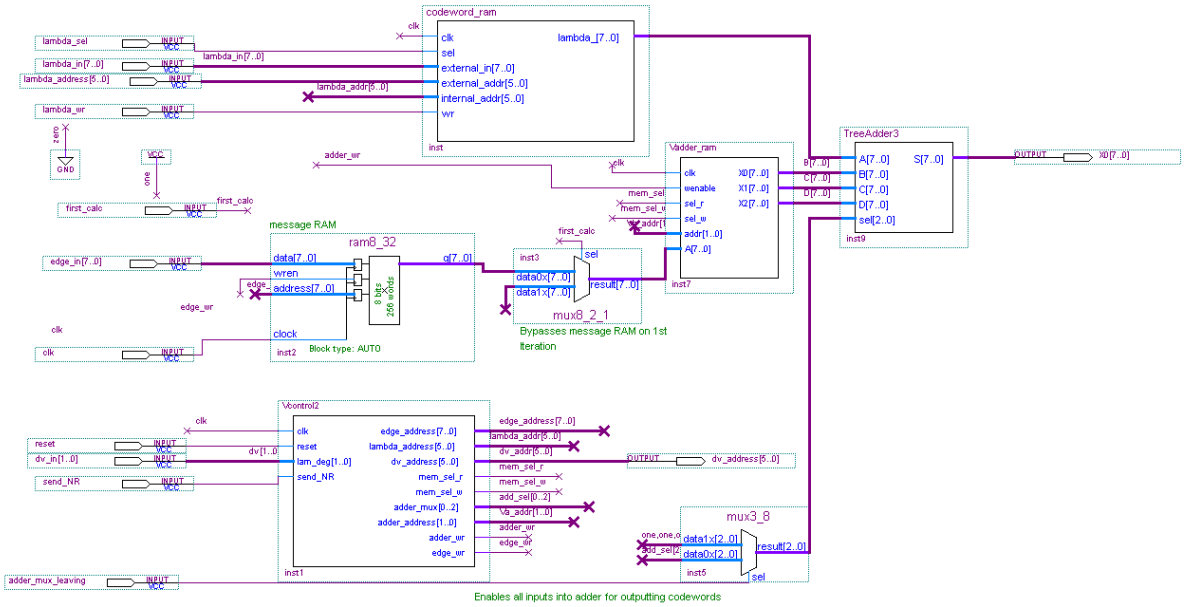


Figure 6.3: Block diagram of bit node processor

6.3.1 Adder RAM unit

In order to process one message per clock cycle, the adder must have all messages associated with a bit node available. In order to achieve this the Adder RAM unit implements a serial to parallel converter, making the messages associated with the current node being processed available. When it is time to process a new bit node the processor would ordinarily have to wait for the serial to parallel converter to load all the necessary messages. To avoid stalling the processor for every bit node, the converter has two memories. It serially loads the messages for the *next* bit node into one memory while the other memory with the messages for the *current* bit node is available in parallel for the adder. The control unit is responsible for switching the memories via the control line. The code implementing this block can be

seen in Appendix B.3.

6.3.2 Codeword RAM Unit

The codeword loading process (see Section 6.5) takes as many clock cycles as the length of the codeword (in the prototype 480), which is a long time to stall the decoder. The decoding process takes more clock cycles then the number of code-bits and so there is time while the decoder is processing messages to load the next codeword (if it is available). It works in much the same way as the adder RAM unit except that it is controlled by the main control unit.

There are two codeword RAM's, with the control line selecting which one is available to the bit node adder and which is available to load code-bits. With this configuration the decoder is able to load the next codeword while its still processing the current one. At the conclusion of decoding a codeword, the control unit signals to the codeword loader that a new codeword can be loaded. Figure 6.4 shows the block diagram of the unit.

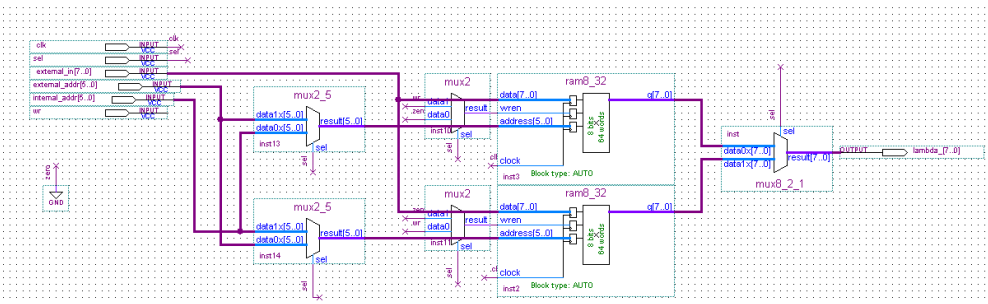


Figure 6.4: Block diagram of codeword RAM unit

6.3.3 Control Unit

When the bit node processor is receiving messages, the control unit sets the write enable for the message RAM and increments the address, ensuring the the incoming messages are stored in the correct location.

When the processor is processing and sending messages, the control unit loads the address's for the *next* bit node into the adder RAM unit. This introduces a one bit node latency into the decoder before the first bit node's messages can be sent. The adders

for the processor have input enables so that the message being sent along an edge of the Tanner graph doesn't contain the message that came from it in the previous check node half iteration. This is achieved with multiplexes that send zero to the adder instead of the message.

The control unit increments the address of the channel measurements RAM block so that the channel measurement associated with the currently processing bit node is available to the adder. The control unit also increments the address for the degree weight RAM block, which is used by the control unit to determine how many edges to load into the Adder RAM unit for each bit node. Although the prototype design utilises a fixed degree weight (regular codes), it is hoped the final design will have variable degree weight (irregular codes), so the control unit for the bit (and check) node processors has been designed with this in mind.

The bit node processor loads the correct number of messages for the next bit node to be processed by reading its degree weight from the degree weight RAM. At the same time its processing the current bit node, setting the address of the codeword RAM so that the adder has the current code-bit. When the messages of the bit node are being calculated, the control unit disables the corresponding adder input, eliminating the effect of incoming message from the outgoing message.

On the first iteration the message ram is uninitialised, so the main control unit asserts a control signal which bypasses the message RAM via a multiplexer. The code that implements the bit node processor control unit can be found in Appendix B.3

6.4 Check Node Processor

The check node processor is identical to the bit node processor except for the adder and there being no codeword RAM or its associated control unit. For the check node processor the adder accepts 8 inputs (only 6 are used in the prototype). At each adder input has a $\psi(x)$ lookup table, the output of the adder is passed through another $\psi(x)$ lookup table which also performs the sign correction as described in Section 6.1.2. The sign correction is performed by XOR'ing the sign bit (MSB) of the inputs together and if the result is a '1', then the result of the LUT is made negative. Figure 6.5 shows the block diagram of the check node processor, while Appendix B.4 shows the code for the control unit. When

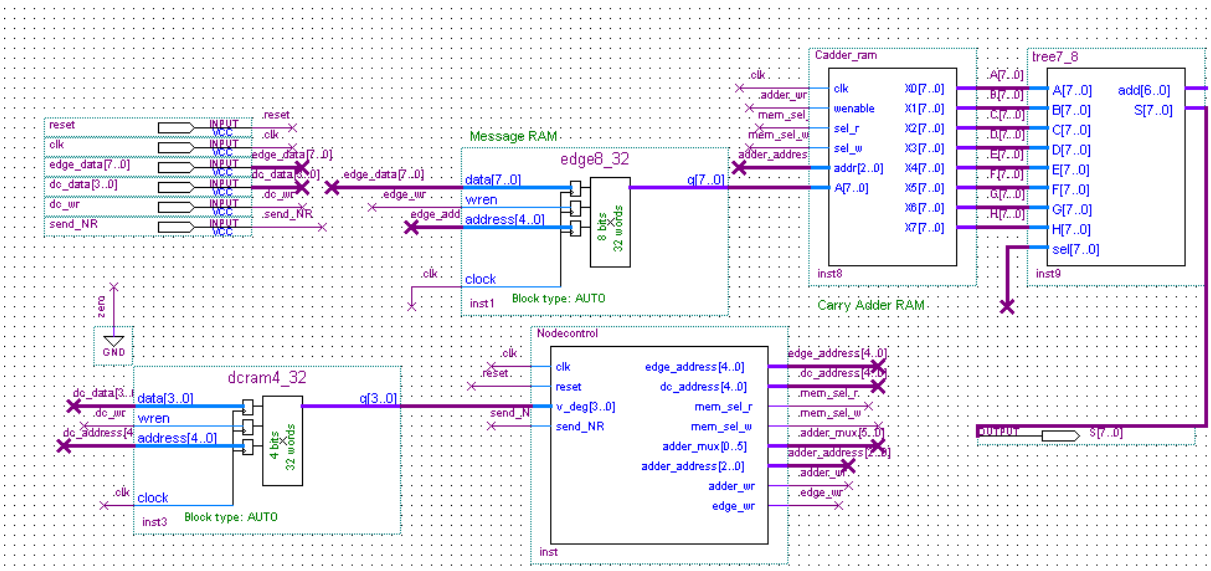


Figure 6.5: Block diagram of check node processor

an input is not needed on the check node processor adder, its input is replaced with the binary representation of 7.9375 (01111111), this is because $\psi(7.9375) = 0$ and in this way the unused input will not affect the sum. Figure 6.6 shows the check node processor adder.

6.5 Codeword Loader

For the flexible LDPC decoder to be able to decode a codeword, it must be able to load a codeword, this task is assigned to the codeword loader. The codeword loader waits for a signal from the IO unit signifying that a new codeword is available. The codeword ROM contains which bit node processor each code-bit is assigned as well as the order. The codeword loader uses the ROM to select the assigned bit node processor and the address of each code-bit. Figure 6.7 shows the block diagram of the codeword loader while Appendix B.5 shows the code implementing the control block.

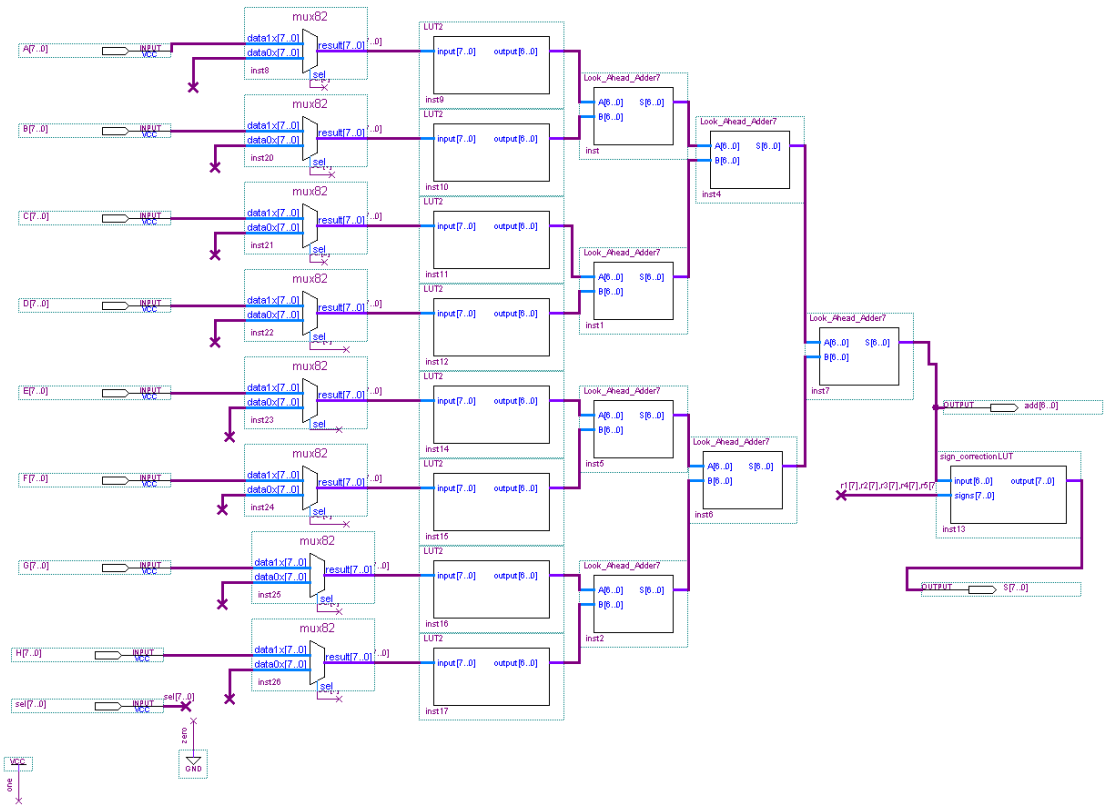


Figure 6.6: Block diagram check node processor adder

6.6 Message Permutation Network

The permutation network lies at the heart of the LDPC decoders flexibility. The incoming messages are first permuted in space via the first Benes network, and then in time by the interleaver banks and finally in space again by the second Benes network. Given the correct scheduling defined in Chapter 4, any code can be supported by the permutation network. Figure 6.8 shows the block diagram of the message permutation network.

6.6.1 Benes Network

Benes networks were used as the crossbar switches due to having the lowest implementation complexity and control logic requirements of any of the alternatives [16, 7]. The Benes network is constructed recursively starting with the 2x2 elementary switch and using it to

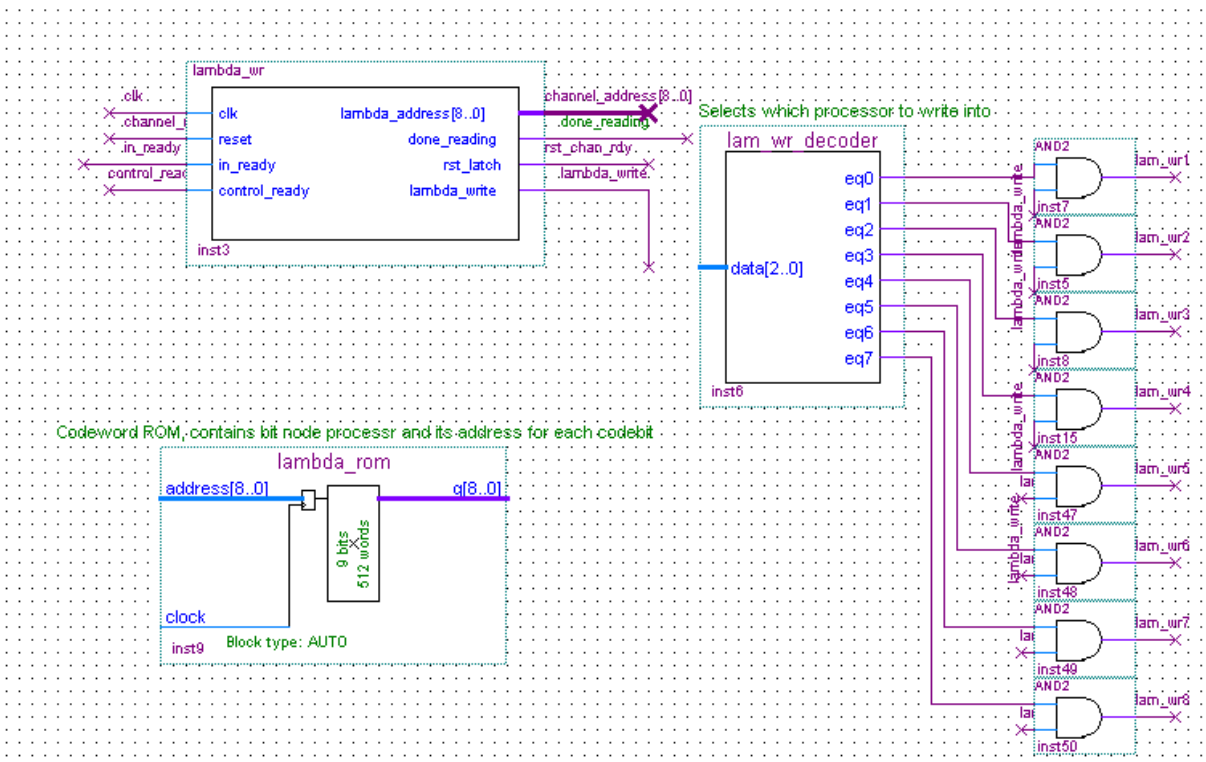


Figure 6.7: Codeword loader block diagram

create a 4x4 switch, then using that to create an 8x8 switch and so on. The network is controlled by setting or resetting each of the 2x2 elementary switch cells using the methodology shown in Section 3.3. The control inputs are read from the input and output switch ROM files. The address for each configuration is controlled by the control unit. There are more configurations for the output switch rom file due to it being used to output the decoded codeword. Figure 6.9 shows the block diagram of the Benes network.

6.6.2 Interlever Banks

The interleaver banks are implemented in hardware as dual port RAM blocks. The write address is incremented, storing the incoming messages in the order they are received, while the read address is the permutated order they are read out. The *ith* interlever bank's read order is set by δ_i as defined in Section 4.1.4. The interlever block diagram is shown in figure 6.10

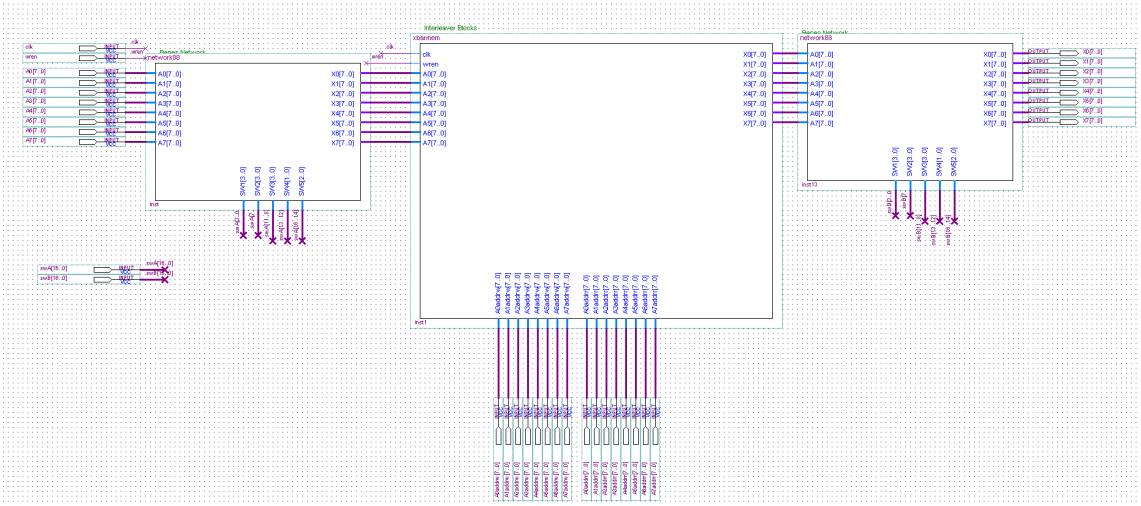


Figure 6.8: Block diagram of message permutation network

6.7 Decoder Output

The decoder output block reads the decoded codeword from the 1st output of the message permutation block. It performs a hard decision on each code-bit by storing the sign. If the code-bit is negative it stores a '1' while it stores a '0' if its positive. It performs a serial to parallel conversion 8 bits at a time, storing the result in decoded RAM. Each word of decoded RAM thus has 8 hard decision decoded code-bits.

6.8 Input and Output

Input and output from the board for the proof of concept and prototype is by serial RS-232 communications. A UART has been created based on the code of Juin [25]. The communications parameters are 28800 bits per second, 8 data bits, 1 stop bit with even parity. Input is received from the Matlab testbench and when a complete codeword is received a signal is sent to the codeword loader so that it can be loaded into the correct bit node processors.

When the codeword is decoded, at the conclusion of the control unit's final_out stage, the IO block sends the contents of decoded RAM over the UART back to the Matlab testbench.

It is hoped that for the full design ethernet communications will be used, possibly

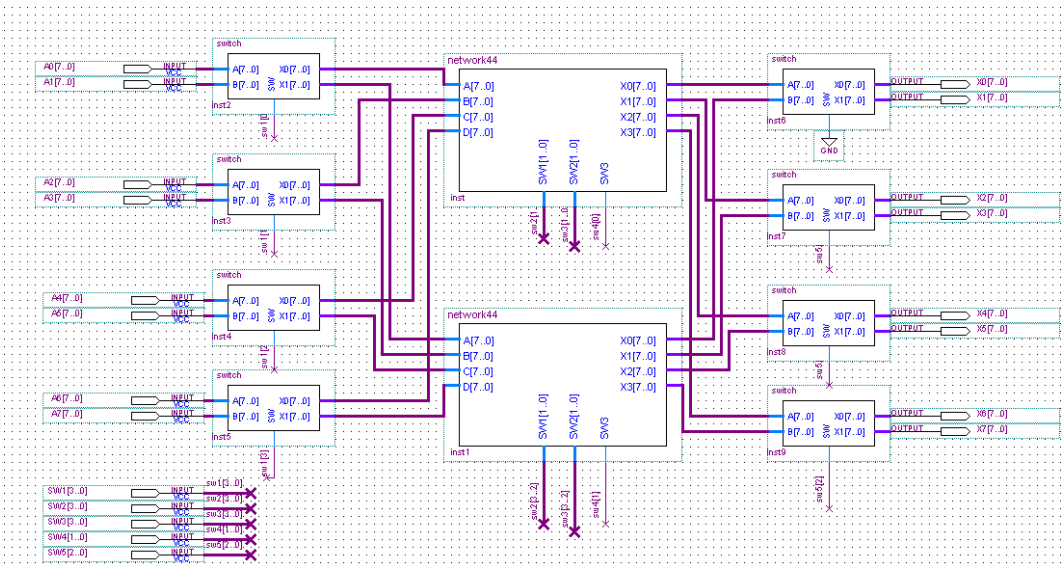


Figure 6.9: Benes network block diagram

utilising a web server on the Stratix II board and a web client on a PC.

6.9 Control

The main control unit implements a 12 stage state machine, controlling the all of the parts of the decoder. The control unit reads the switch and interleaver ROM's to control the permutation network, which is at the heart of the decoders flexibility. The state transition diagram is shown in figure 6.11

State idle

The decoder starts in this state, and remains here until a new codeword is available to decode. In this state writing to the permutation network is disabled. When a codeword is available the decoder proceeds to the new_message state, flipping the codeword RAM select signal so that the new message is available to the bit node processor. The bit and check node processors are held in the reset state.

State new_message

In this state the control unit sets a signal to bypass the bit node message RAM blocks as they are uninitialised. The decoder proceeds straight into the bit wait state and drops the reset on the bit node processors.

State bit_wait

With the reset dropped on the bit node processors they start sending messages, but there is a latency introduced of one bit node (in the prototype this is 3 clock cycles) by the bit node adders so the decoder must wait until the bit node processors output messages. As the decoder proceeds to the bit_in state it increments the iterations counter.

State bit_in

In this state write is enabled into the permutation block and the messages from the bit node processor are written in. The control block increments the address of the switch ROM so that the the bit node messages are stored in the correct interleaver blocks. When all the messages from the bit node processors have been written into the message permutation block the decoder proceeds to the bit_out state.

State bit_out

The reset on the check node processors is dropped while the writing to the message permutation block is disabled. The decoder increments the address of the switch and interleaver ROMs so that the check node can receive the correct message. When all of the messages have been loaded into the check node processors' message RAM the decoder proceeds to the bit2check state.

State bit2check

In this state the functions of the bit node and check node processors are reversed. The check node will be sending messages and the bit node receiving. The input into the message permutation block is set to the check node processors. The check node processors are reset

for a clock cycle to switch them from receiving to sending messages. The decoder proceeds to the check_wait state.

State check_wait

As in the bit_wait state, the decoder has to wait one check node (in the prototype this is 6 clock cycles) for the check node processors to start sending messages. The decoder then proceeds to the check_in state.

State check_in

In this state write is enabled for the permutation message block and the messages from the check node processors are stored in the interleaver blocks. The control block increments the address of the switch ROM to ensure that the check node messages are stored in the correct interleaver blocks. When all the messages from the check node processors have been written into the message permutation block the decoder proceeds to the check_out state.

State check_out

The bit node procssors' reset is dropped while writing to the message permutation block is disabled. The decoder increments the address of the switch and interleaver ROMs so that the bit node processors can receive the correct messages. When all of the messages have been loaded into the bit node processors' message RAM the decoder proceeds to the check2bit state.

State check2bit

In this state the functions of the bit node and check node processors are reversed. The bit node processors will be sending messages and the check node processors receiving. The input into the message permutation block is set to the bit node processors. The bit node processors are reset for a clock cycle to switch them from receiving to sending messages. If the iteration count is less then the predetermined number (10 in the prototype) the decoder moves to the bit_wait state, otherwise it moves to the final_wait state.

State final_wait

This state is similar to the bit_wait state, the reset being dropped on the bit node processors, the decoder is waiting until the bit node processors start sending messages. The control unit sets a signal that causes *all* of the inputs into the adder of the bit node processor to be used. In doing this the bit node processors calculate the following for every bit node, implementing equation 1.10, thus calculating the messages ready for hard decision decoding.

$$L_j = \lambda_j + \sum_{\alpha \in C[j]} R_{\alpha j}[k] \tag{6.7}$$

The decoder proceeds to the final_in state.

State final_in

In this state the messages from the bit check nodes are stored in the interleaver banks. The input switch ROM is unused however, with the input from each processor being stored in its respective interleaver bank. When all the messages have been sent to the interlever banks the decoder proceeds to the final_out state.

State final_out

The decoder increments the address of the output switch and interleaver ROMs. This produces the decoded codeword on the 1st output of the permutation network. The decoder output block performs a hard decision on the codeword and stores the result in the decoded RAM. The decoder has now decoded a codeword and proceeds back to the beginning, the idle state, to process another.

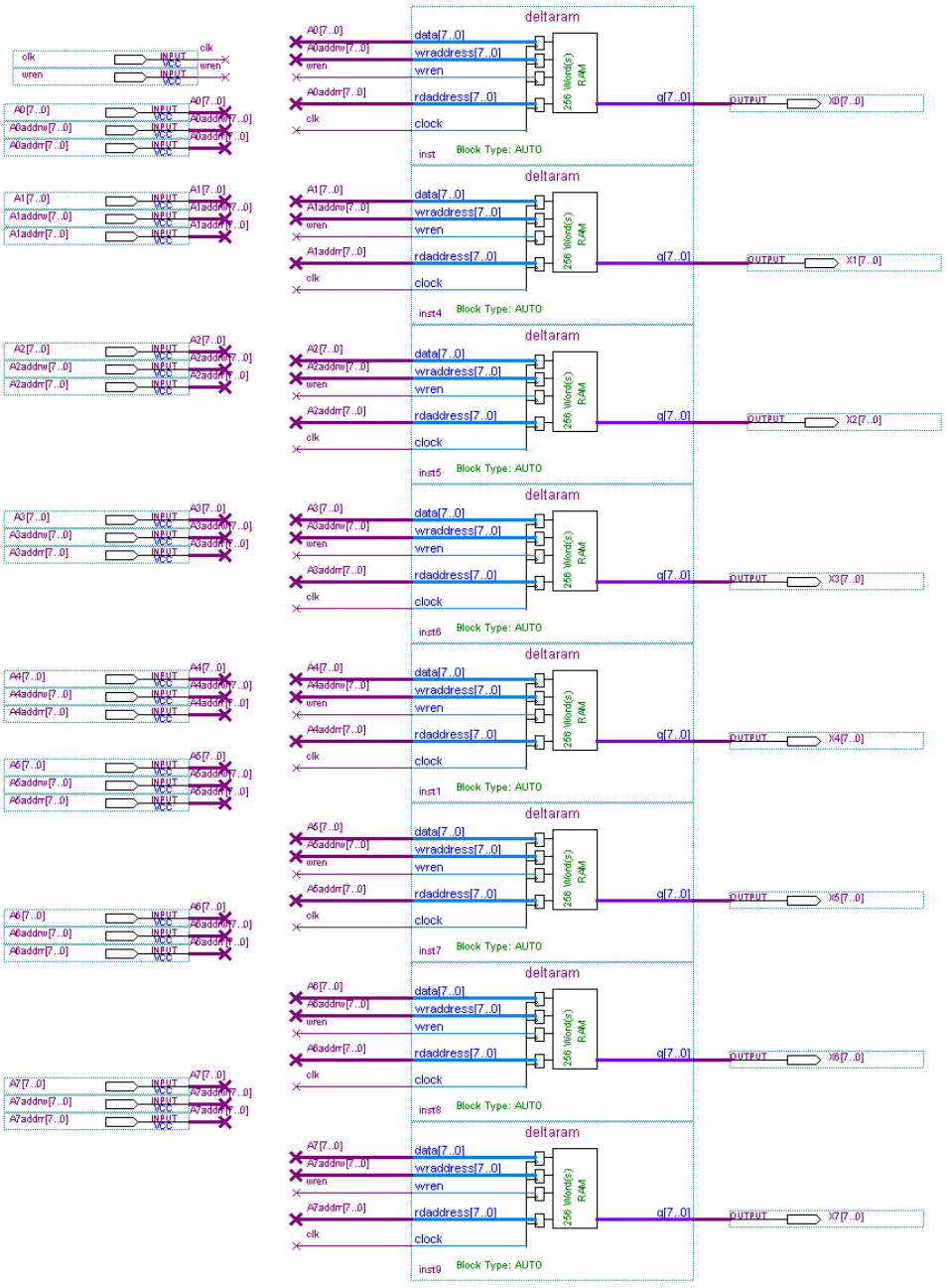


Figure 6.10: Interleaver banks block diagram

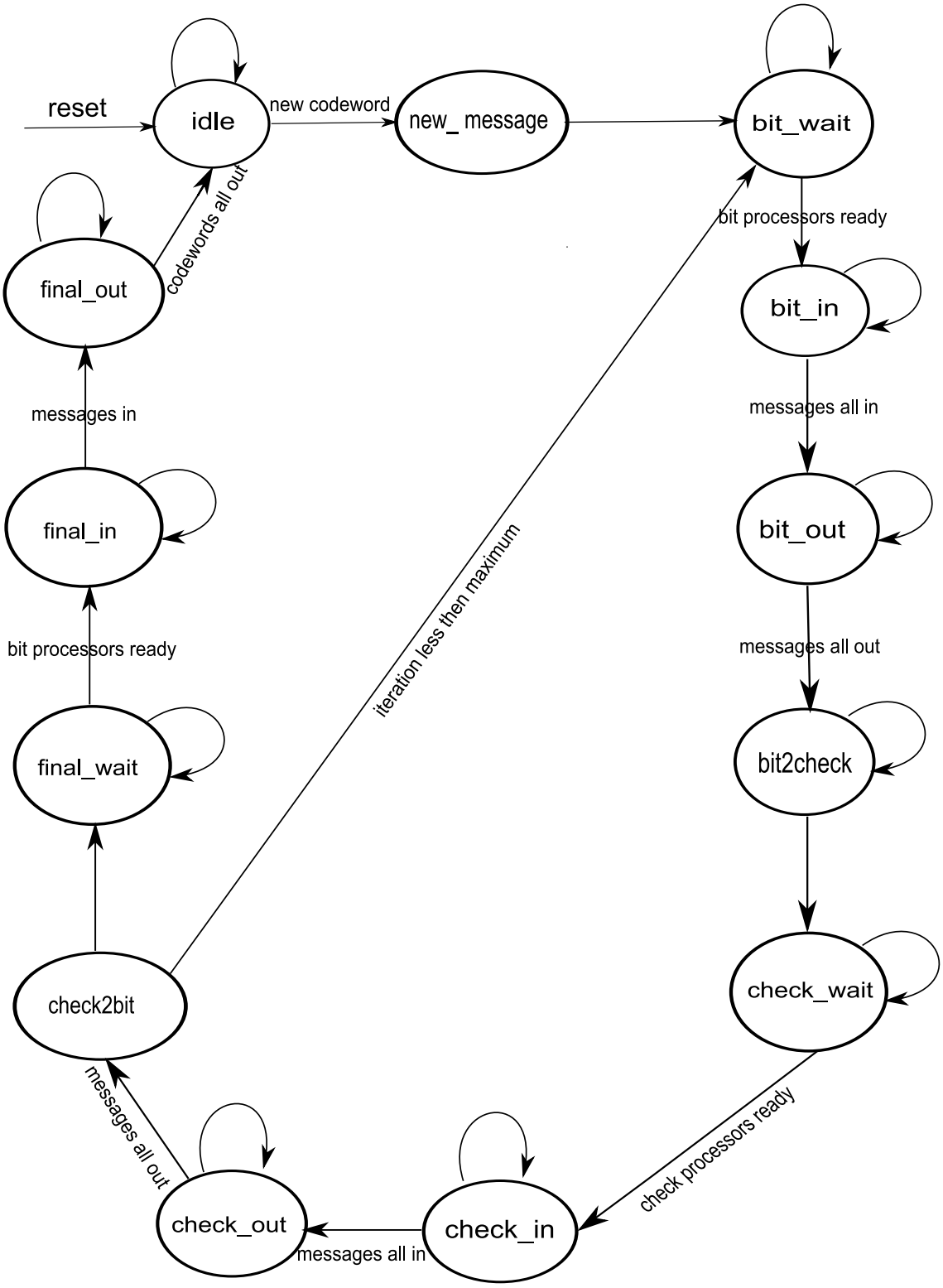


Figure 6.11: Control unit state machine

Chapter 7

Results

A number of tests have been run to determine the correct operation of the proof of concept hardware decoder. The decoder Hardware has been compared with Matlab simulations in both fixed and floating point. A decoded codeword is compared between the prototype decoder and the matlab testbench. Finally the resource usage of the proof of concept and prototype decoders is measured.

7.1 Decoder Hardware verification via Matlab Simulation

The proof of concept (POC) decoder was placed on a Altera Cyclone II (EP2C35F672C6N) DE2 board. The decoder utilises four parallel processors as described in Chapter 6. The following parity check matrix was used to test the decoder:

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{7.1}$$

The Matlab testbench was used to simulate transmission of Binary Phase Shift Keying (BPSK) over an Additive White Gaussian Noise (AWGN) channel using a serial communications RS-232 link.

The signal to noise ratio (SNR) was varied from 0 to 10 dB in 1dB increments, with 200,000 codewords transmitted per SNR point. This gives 1,600,000 code-bits per SNR

point and 17,600,000 codebits for the entire simulation. Each decoder ran for 10 iterations.

The same codewords were decoded with the Matlab testbench using the same fixed point representation as the decoder, as well as a floating point representation and are shown in Figure 7.1.

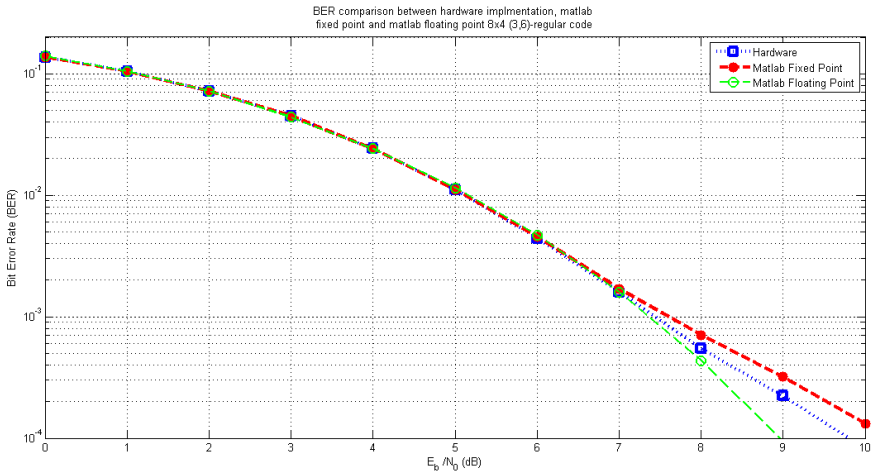


Figure 7.1: Results of Hardware and Matlab comparison

At SNRs 6 dB below, the hardware decoder, Matlab fixed point and Matlab floating point are in complete agreement. As expected, at higher SNR's the floating point outperforms, the Matlab fixed point and the hardware decoder due to superior precision in a floating point representation. At best the Matlab floating point is only performing 0.5dB better then the fixed point representations, which is surprising considering that only 8 bits are used to store and calculate messages throughout the decoder.

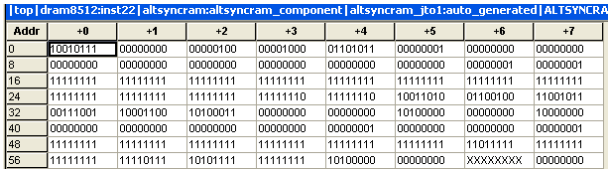
At SNRs 7dB and above, the hardware decoder and fixed point Matlab simulation begin to diverge, although at worst case its less then 0.5dB. The reason for the discrepancy is due the order in which the Matlab testbench performs addition operations. Both the hardware decoder and Matlab testbench perform limiting on the sum if it would overflow, but the hardware decoder uses a tree adder; operands are added together and the results added with other operand addition results. The testbench keeps a running sum and adds the operands one at a time, performing an overflow check at the conclusion of each addition.

The result is that due to the hardware decoder and testbench performing overflow limiting in different ways, the testbench will not be bit accurate with the hardware decoder

and such a discrepancy was expected. Note that the poor BER performance in Figure 7.1 (compared to capacity approaching LDPC codes) is due to the short code length and too many 4-cycles in the parity check matrix H.

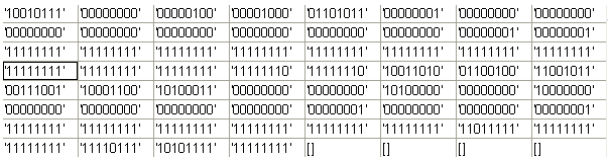
7.2 Decoded Codeword Comparison of Prototype and Testbench

The 480x240 $(3,6)-regular$ parity check matrix can be found online [26]. It was loaded onto the decoder and was simulated for 10 iterations. Appendix C.1 also shows the codeword that was decoded. Figure 7.2 shows the decoded codeword in the RAM block of the prototype decoder, while Figure 7.3 shows the decoded codeword in the Matlab testbench.



Addr	+0	+1	+2	+3	+4	+5	+6	+7
0	10010111	00000000	00000100	00001000	01101011	00000001	00000000	00000000
8	00000000	00000000	00000000	00000000	00000000	00000000	00000001	00000001
16	11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
24	11111111	11111111	11111111	11111110	11111110	10011010	01100100	11001011
32	00111001	10001100	10100011	00000000	00000000	10100000	00000000	10000000
40	00000000	00000000	00000000	00000000	00000001	00000000	00000000	00000001
48	11111111	11111111	11111111	11111111	11111111	11111111	11011111	11111111
56	11111111	11110111	10101111	11111111	10100000	00000000	XXXXXXXX	00000000

Figure 7.2: Results of Hardware and Matlab comparison



'10010111'	'00000000'	'00000100'	'00001000'	'01101011'	'00000001'	'00000000'	'00000000'
'00000000'	'00000000'	'00000000'	'00000000'	'00000000'	'00000000'	'00000001'	'00000001'
'11111111'	'11111111'	'11111111'	'11111111'	'11111111'	'11111111'	'11111111'	'11111111'
'11111111'	'11111111'	'11111111'	'11111110'	'11111110'	'10011010'	'01100100'	'11001011'
'00111001'	'10001100'	'10100011'	'00000000'	'00000000'	'10100000'	'00000000'	'10000000'
'00000000'	'00000000'	'00000000'	'00000000'	'00000001'	'00000000'	'00000000'	'00000001'
'11111111'	'11111111'	'11111111'	'11111111'	'11111111'	'11111111'	'11011111'	'11111111'
'11111111'	'11110111'	'10101111'	'11111111'	[]	[]	[]	[]

Figure 7.3: Results of Hardware and Matlab comparison

7.3 Resource usage by device entity

The proof of concept (POC) and prototype decoders were placed onto a Altera Cyclone II (EP2C35F672C6N) DE2 board. Their resources usage on the FPGA in terms of lookup tables is shown below in Tables 7.1 and 7.2

It can be seen that in both the proof of concept and prototype designs the check node processors dominate the resource usage. The $\psi(x)$ lookup tables are responsible for this, they are 100 logic elements each and there are 9 instantiated for each check node processor.

	Logic Cells (%)	Total
Message Permutation Block	3.1	164
Bit Node Processors	19	1000
Check Node Processors	76.1	4000
Control Unit	1.7	90

Table 7.1: Proof of concept resource usage

	Logic Cells (%)	Total
Message Permutation Block	4.7	556
Bit Node Processors	20.6	2240
Check Node Processors	73.4	8648
Control Unit	1.3	136

Table 7.2: Prototype resource usage

While the check node processors account for the largest part of the design, they are not prohibitively complex, and should not stop a 32 parallel processor design being implemented on Altera's Stratix II EP2S60F672C5E2 FPGA. The proof of concept and prototype decoders occupy 17% and 31% of the Cyclone II FPGA respectively.

Tables also show that when the number of parallel processing elements is doubled and the maximum supported code is 60 times larger, the complexity of the design increases by slightly less then two fold. The shows that the complexity of the decoder is increasing linearly with the number of parallel processing nodes and thus the scalability of the decoder.

Chapter 8

Conclusion and Further Work

From the results presented in Chapter 7, it can be seen that the LDPC decoder is a success. Given a (3,6)-regular LDPC code, the Matlab software created, defines a scheduling, configures Benes network switches and generates ROM definition files for Altera's Quartus to use when downloading onto the Cyclone DE2 Development board. All thats needed to utilise the new code on the LDPC decoder is to inform Quartus that the ROM definition files are changed and to download them onto the board - no reconfiguration is necessary.

Using the Matlab serial interface between the LDPC decoder and the Matlab testbench, the LDPC decoder has been tested and it functionality verified. Indeed it has been shown in Figure 7.1 that not only is the performance of the decoder within 0.5 dB of the finite precision testbench, it is also within the BER performance of the floating point implementation.

The operation of the LDPC decoder has been tested and verified with larger codes and it has been shown to scale well with increasing parallel processing units.

The next step in the design of a flexible LDPC decoder is to increase the number of parallel processing units to 32. This will give it a speed boost of 400% compared to the prototype. This design will not fit on the Cyclone DE2 FPGA, and so the next step is to transition the chip to the Stratix II EP2S60F672C5E2 FPGA. Both the transition to a new platform as well as increasing the decoder throughput will necessitate a faster input output interface. The final goal is to implement an ethernet interface between the FPGA and a PC.

It was planned to integrate the LDPC decoder within the SPM group's MIMO wireless testbed project, but some critical components of the MIMO testbed are not yet ready. However the completed LDPC decoder will be able to be uploaded to the testbed FPGA as soon as the test bed is ready.

Appendix A

MATLAB® Code Listing

partNmap.m

```
% This file accepts a parity check matrix, the number of partitions and a
% filename.  Its ultimate purpose is to generate ROM configuration blocks
% for a given code for use by Altera's Quartus II in device programming.
%
% It converts the parity check matrix into a Tanner graph, passes it
% through matrix partitioning software, checks the partitions are equal on
% both the bit and check half iterations, followed by finding the
% scheduling function.
% The next step is to generate the Benes network switch settings, followed
% by generating the rom files and writing them to disk.
%
%
% Author: David F Hayes
%
%

function partNmap(H,parts,fn)
```

```
matrix2Graph(H,fn);          %Convert H to Tanner graph, saving to disk

%Call METIS to parition the graph, again saving results to disk
str = ['metis\pmetis ',fn, ' ',num2str(parts)];
system(str);

%Load partitioning, check its even on both bit and check sides and put in
%a sutible format for the scheduling
[p1,p2,Vn,Cn] = plist2part(H,[fn,'.part.',num2str(parts)],parts);

%Crete a scheduling function for the code, giving crossbar switch and
%interleaver settings
[b1,b2,delta] = mapping(H,p1,p2,parts);

%Generate Benes network switch settings including output
[sw1a,sw2a] =  generateSwitches(parts,b1,b2,Vn);

%Generate interleaver settings including output
delta_f = generateDelta(H,delta);

print_files(H,Vn,Cn,delta_f,sw1a,sw2a);
```

A.1 Matrix to Tanner Graph

matrix2Graph.m

```
% This file accepts a parity check matrix and outputs its corresponding
% Graph representation, with the first line containing the number of
% vertices and edges and the following containing the edges connected to
% to the vertex n (line n-1).  The bit nodes appear before the check nodes
%
% Author: David F Hayes
```

```
%

%
function H = matrix2Graph(H,fn)

[M,N] = size(H);
[I,J]=ind2sub([M,N],find(H)); %Find position of 1s
edges=size(I,1);
F = fopen(fn,'w');
%Print number of vertices and edges
fprintf(F,'%d %d\n',(M+N),edges);
%Print bit nodes
for i=1:N
    fprintf(F,'%d ',N+I(find(J==i))');
    fprintf(F,'\n');
end
%Print check nodes
for i=1:M
    fprintf(F,'%d ',J(find(I==i))');
    fprintf(F,'\n');
end
fclose(F);
```

A.2 Graph Partitioning

plist2part.m

```
% This file converts the output from METIS graph partitioning and produces
% 2 sets P1 & P2 containing the assignment of bit and check nodes to
% processors during the bit and check half-iterations.
% It also attempts to even the partitions up between bit and check
```

```
% half-iterations, giving an error when it cannot.
% Author: David F Hayes
%
%

function [P1,P2,Vn,Cn] = plist2part(H,fn,partitions)
rand('twister',5489) %Repeatable random number generation
edges=sum(sum(H));
M=ceil(edges/partitions);
%load partition
partition = load(fn);
%partition = partition+1;
%partition = ([1;1;2;2;3;3;4;4;1;2;3;4]);
partition = [ones(1,60),2*ones(1,60),3*ones(1,60),4*ones(1,60),5*ones(1,60),6*ones(1,
[Mh,Nh] = size(H);
[I,J] = find(H);
%create P1 (bit half-iteration)
for z=1:1000
    plist = zeros(partitions,2*M);
    for i=1:Nh
        ind = find(plist(partition(i),:)==0,1);
        edge = find(J==i)';
        plist(partition(i),ind:ind+size(edge,2)-1)=edge;
    end
    %Check which partitions are currently under or over full
    overfull = find(plist(:,M+1)~=0);
    underfull = find(plist(:,M)==0);
    %If none are overfull then we are done
    if(size(overfull,1)==0)
        break;
    end
```

```
    %Randomly move a node from a overfull partition to an underfull one
    over = overfull(ceil(rand()*size(overfull,1)));
    under = underfull(ceil(rand()*size(underfull,1)));
    lst = find(partition(1:Nh)==over);
    partition(lst(ceil(rand()*size(lst,1))))=under;

end
%Done now we move onto the check half-iteration
Vn = partition(1:Nh);
P1=plist(:,1:M);
if size(overfull,1)~=0
    error('unbalanced','unbalanced');
end

partition = partition(Nh+1:Nh+Mh);
for z=1:1000
    plist = zeros(partitions,2*M);
    for i=1:Mh
        ind = find(plist(partition(i),:)==0,1);
        edge = find(I==i)';
        plist(partition(i),ind:ind+size(edge,2)-1)=edge;
    end
    %Check which partitions are currently under or over full
    overfull = find(plist(:,M+1)~=0);
    underfull = find(plist(:,M)==0);
    %If none are overfull then we are done
    if(size(overfull,1)==0)
        break;
    end
    %Randomly move a node from a overfull partition to an underfull one
    over = overfull(ceil(rand()*size(overfull,1)));
```

```
    under = underfull(ceil(rand()*size(underfull,1)));
    lst = find(partition(1:Mh)==over);
    partition(lst(ceil(rand()*size(lst,1))))=under;
end
%Done with check half-iteration
Cn = partition(1:Mh);
if size(overfull,1)~=0
    error('unbalanced','unbalanced');
end
P2=plist(:,1:M);
%Insert dummy edges if there M/L isn't an integer
if size(underfull)~=0
    under1 = find(P1==0);
    under2 = find(P2==0);
    for i = 1:size(under2)
        P1(under1(i)) = i+edges;
        P2(under2(i)) = i+edges;
    end
end
```

A.3 Scheduling

mapping.m

```
% This file produces a mapping function defining the scheduling of the
% decoder.  The mapping function is then used to determine the
% configuration of the crossbar switches and the interleaver banks.
%
% Author: David F Hayes
% P - the number of parallel processors
% p1 - set contaning the nodes associated with the P processors in the bit
```

```
% node half iteration
% p2 - set contaning the nodes associated with the P processors in the check
% node half iteration
%
% b1,b2 - the configuration of the left and right crossbar switches
% delta - the configuration of the interleaver banks
%
function [b1,b2,delta] = mapping(p1,p2,P)

%Define some constants
L=size(p1,1)*size(p1,2);
M=L/P;

%Algorithm only works if p1 is in numerical order, so we reorder the edge
%numbering to ensure that it is.

p11=zeros(P,M);
p21=zeros(P,M);
p22=p2';
for i=1:L
    p11(ceil(i/M),myMod(i,M)) = i;
    p21(ceil(i/M),myMod(i,M)) = find(p1'==p22(i));
end
p1 = p11;
p2 = p21;

%create pi permutation
pi = zeros(1,size(p2,2)*size(p2,1));
for i = 1:P
    pi((i-1)*M+1:i*M)=p2(i,:);
end
```

```
%create empty mapping function and L variables
Map = zeros(1,L);
Lv = zeros(P,M);
Lv1= Lv;
for i = 1:P
    Lv1(i,:)=i;
end
Lv2 = Lv1;
Lv = Lv1;
%for every edge
for j=1:L
    J=j;
    jv1=myMod(find(p1'==J),M);
    jv2=myMod(find(p2'==J),M);

    a = nonzeros(intersect(Lv1(:,jv1),Lv2(:,jv2)));

    if size(a,1)>0
        %We have an assignment without collision
        Map(j) = a(1);
        Lv1(a(1),jv1)=0;
        Lv2(a(1),jv2)=0;
    else
        %No assignment without collision possible, create collision
        m = nonzeros(Lv1(:,jv1));
        n = nonzeros(Lv2(:,jv2));
        m = m(1);
        n = n(1);
        Map(J) = m;
        %now try to fix it
        while 1
```

```
    a = p2(find(p2(:,jv2)~=J),jv2); %list of elements in V' that may need upd
    found = false;
    for i=1:length(a)
        if (Map(a(i))==m)
            found=true;
            J=a(i);
            Map(J) = n;
            jv1=myMod(find(p1'==J),M);
        end
    end
    %fixed it but may have caused another one
    if found==true
        a = p1(find(p1(:,jv1)~=J),jv1); %list of elements in V that may need
        found = false;
        for i=1:length(a)
            if (Map(a(i))==n)
                found = true;
                J=a(i);
                Map(J) = m;
                jv2=myMod(find(p2'==J),M);
            end
        end
        if found==false
            %no more collisions
            break
        end
    else break %no more collisions
    end
end
%update variables
Lv1 = Lv;
```

```
        Lv2 = Lv;
        for i=1:j
           Lv1(Map(i),myMod(find(p1'==i),M)) = 0;
           Lv2(Map(i),myMod(find(p2'==i),M)) = 0;
        end
    end
end
%were done with mapping now create b1,b2 and delta
b1 = zeros(P,M);
b2 = zeros(P,M);
delta = zeros(P,M);
piinv = zeros(1,L);
%create permutation inverse piinv
for i=1:L
piinv(pi(i))=i;
end
for i = 1:P
    a = find(Map==i);
    for j = 1:M
        delta(i,myMod(piinv(a(j)),M))= myMod(a(j),M);
        b1(i,j) = Map((i-1)*M + j);
        b2(i,j) = Map(pi((i-1)*M + j));
    end
end
```

A.4 Benes Network

benes-main.m

```
% This function configures the switches for a Benes network given a
% permutation piT.
```

```
% The switches are converted to hexadecimal format for printing to file
%
% Author: David F Hayes
%
function sw = benes_main(piT)
global s;
global N;
clear sw;
N = 4;
%piT = [16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1];
%piT = [32,31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,
%piT = [17,16,18,15,19,14,20,13,21,12,22,11,23,10,24,9,25,8,26,7,27,6,28,5,29,4,30,3,
%piT = [1,5,9,12,15,18,21,24,27,30,2,6,10,13,16,19,22,25,28,31,3,7,11,14,17,20,23,26,
s = zeros(1,N*log2(N)-N+1);
benes(piT,N,1);
for i = 1:(N*log2(N)-N)/4
sw(i) = dec2hex(bi2de(s((i-1)*4+1:i*4)));
end
sw((N*log2(N)-N)/4+1) = dec2hex(bi2de(s(N*log2(N)-N+1)));
sw = fliplr(sw)
```

benes.m

```
% This recursively sets the switches associated with the benes network, it
% accepts a permutation, a size and a parameter d, which refers to the
% level of the subblock from the top ie a top block is one while a bottom
% block is 2.
% The level is used to determine which switches to set within the switch
% vector.
% Depth Example for a 8x8 network left is the 8x8 block, middle (1,2) are the
% upper and lower 4x4 sub-blocks and the right most (1,2,3,4) are the 2x2
% blocks
%
```

```
%     1
%   1<
%  /  2
%1<
%  \  3
%   2<
%     4
%
%
% Author: David F Hayes
%
function b = benes(newPi,n,d)
global P;
global pi;
global piinv;
global s
global N
% Formulas for calculating switch positions for input and output side
% Obtained through Voodoo sacrifices
switch_start = (N/2)*log2(N/n)+ (d-1)*n/2;
switch_end = (N/2)*log2(N)+(N/2)*(log2(n)-3+4/n)+(d-1)*(n/2-1)-1;

pi = newPi;
piinv = inverse(pi);

%This is the termination case we have reached an elementry position
if (n==2)
    s(switch_start+1) = pi(1)-1;

else
    P = zeros(n,n);
```

```
    %1 - Partition A, -1 Partition B
    %Fill the matrix
    while (sum(sum(abs(P)))<n)
        a = find(sum(P,1)==0); %find all entries that have not been filled
        a = a(1); %take the first
        P(piinv(a),a) = 1;
        forward(piinv(a),-1);
    end

    %Set switches
    [a,b] = ind2sub([n,n],find(P'==1));
    for i=1:n/2
        s(switch_start+i) = abs(rem(b(i),2)-1);
    end
    [a,b] = ind2sub([n,n],find(P==1));
    for i=2:n/2
        s(switch_end+i) = abs(rem(b(i),2)-1);
    end
    %Find permutation for subblocks and recursively find
    Pa = inverse(ceil(a/2)');
    [a,b] = ind2sub([n,n],find(P==-1));
    Pb = inverse(ceil(a/2)');
    benes(Pa,n/2,d*2-1);
    benes(Pb,n/2,d*2);
end;
b = 0;
```

forward.m

```
% Sets switch position on the input side and if empty calls backward.m to
% set the corresponding switch on the output side
% Author: David F Hayes
%
```

```
function A = forward(indx,entry)
global P;
global pi;

if (rem(indx,2))
    x = indx+1;
else
    x = indx-1;
end
if (P(x,pi(x)) ==0)
    P(x,pi(x)) = entry;
    backward(pi(x),entry*-1);
end
```

backward.m

```
% Sets switch position on the output side and if empty calls forward.m to
% set the corresponding switch on the input side
% Author: David F Hayes
%
function A = backward(indx,entry)
global P;
global piinv;
if (rem(indx,2))
    x = indx+1;
else
    x = indx-1;
end
if (P(piinv(x),x) ==0)
    P(piinv(x),x) = entry;
    forward(piinv(x),entry*-1);
end
```

inverse.m

```
% Finds the inverse of a function pi(x)  assumes that pi(x) is single
% valued.
% Author: David F Hayes
%
function inv = inverse(pi)

inv = zeros(1,size(pi,2));
for i = 1:size(pi,2)
    inv(pi(i))=i;
end
```

A.5 ROM Definition Files

print-files.m

```
% Prints variables to ROM files
% Author: David F Hayes
%
%
%
function print_files(H,Vn,Cn,delta_f,sw1a,sw2a)
a = sum(H,1);
% Prints the degree of each variable and check node associated with each
% processor
for i=1:N
    print_ram_dec(['variable_deg',num2str(i),'.mif'],a(find(Vn==i)),2,64);
end
a = sum(H,2);
for i=1:N
    print_ram_dec(['check_deg',num2str(i),'.mif'],a(find(Cn==i)),8,32);
end
```

```
%print interleaver and switchs
print_ram_hex('delta1.mif',delta_f,64,1024)
print_ram_hex('sw1.mif',sw1a,17,512);
print_ram_hex('sw2.mif',sw2a,17,1024);

%Find and print which processor and at what time the incoming channel is
%associated with.
lambda_rom = zeros(1,size(H,2));
for i=1:size(H,2)
    lambda_rom(i) = (Vn(i)-1)*2^6+find((find(Vn==(Vn(i))))'==i)-1;
end
print_ram_dec('lambda_rom.mif',lambda_rom',9,512)
```

print-ram-dec.m

```
% Writes variable to ROM file, the values are in decimal.
% Author: David F Hayes
%
%
%
function print_ram_dec(filename,values,width,depth)
F = fopen(filename,'w');
fprintf(F,'WIDTH = %d;\n',width);
fprintf(F,'DEPTH = %d;\n',depth);
fprintf(F,'ADDRESS_RADIX = UNS;\n');
fprintf(F,'DATA_RADIX = UNS;\n');
fprintf(F,'CONTENT BEGIN\n');
for i = 1 : size(values)
    fprintf(F,'\t%d  :  %d;\n',i-1,values(i));
end
for i=size(values,1):depth-1
    fprintf(F,'\t%d  : 0;\n',i);
```

```
end
fprintf(F,'END;');
```

print-ram-hex.m

```
% Writes variable to ROM file, the values are in hexadecimal.
% Author: David F Hayes
%
%
%
function print_ram_hex(filename,values,width,depth)
F = fopen(filename,'w');
fprintf(F,'WIDTH = %d;\n',width);
fprintf(F,'DEPTH = %d;\n',depth);
fprintf(F,'ADDRESS_RADIX = UNS;\n');
fprintf(F,'DATA_RADIX = HEX;\n');
fprintf(F,'CONTENT BEGIN\n');
for i = 1 : size(values,1)
    fprintf(F,'\t%d  :  %s;\n',i-1,values(i,:));
end
for i=size(values,1):depth-1
    fprintf(F,'\t%d  : 0;\n',i);
end
fprintf(F,'END;');
```

A.6 Testbench

LDPCSimF.m

```
% Calculates the number of errors for LDPC sum prod decoding using phi LUT
% H matrix
% EbNo SNR
% niter, number decoder iterations
```

```
% Y channel measurement
% Author: David F Hayes
%
%
%
function [numerr] = LDPCSimF(H,EbNo,niter,Y)

%Sends Y codeword with AWGN
%and calculates number of bit errors per iteration
%%Initialisation here
%
[I,J]=ind2sub(size(H),find(H ~=-0));
[M,N]=size(H);
L=zeros(1,N);
%PHI LUT
phi = [127,79,47,40,32,30,28,26,24,22,20,18,16,15,14,13,12,12,11,11,10,10,9,9,8,8,7,7
E=zeros(M,N);
Col=sortrows([I,J]);
Row=[I,J];
%Y=[-0.1,0.5,-0.8,1,-0.7,0.5];
%Y=[-0.63,-0.83,-0.73,-0.04,0.1,0.95,-0.76,0.66,-0.55,0.58];
%niter=3;

%Y = Y + sig*randn(1,N); %%Send all 0's with awgn

%%Calculations Here
r=EbNo*4*Y/2;  %rate = 1/2 ebno is really ecno
r = round(r*16)/16;
r(r>7.9375) = 7.9375;
r(r<-7.9375) = -7.9375;
%r= [2,1.875,1.75,2.1875,2.375,-.3125,2.1875,1.875];
```

```
%load r;
Ma=zeros(M,N);
for i = 1:length(J)
    Ma(I(i),J(i))=r(J(i));
end


numerr = zeros(1,niter);
incorrect = 0;
dv=sum(H,1);
dc=sum(H,2);
for i=1:niter
    index=0;
    %%Check messages
    for j=1:M %%across rows
        for k = 1:dc(j) %%for each parity check equn
            E(j,Col(index+k,2)) = 0;
            sg = 1;

            for l = 1:dc(j) %%product rule on other bits
                if (k ~= l)
                    E(j,Col(index+k,2)) = E(j,Col(index+k,2)) + phi(abs(16*Ma(j,Col(i
                    a = sign(Ma(j,Col(index+l,2)));
                    if (a == 0)
                        a=1;
                    end
                    sg = sg*a;
                    if (E(j,Col(index+k,2)) > 127)
                        E(j,Col(index+k,2)) = 127;
                    end
                end
```

```
        end

        E(j,Col(index+k,2)) = phi(E(j,Col(index+k,2))+1)*sg/16;

    end
    index = index + dc(j);
end
L=r+sum(E,1);
L(L==0) = 1;
z = -sign(L)/2 +0.5;
numerr(1,i) = sum(z);
%syndrome=mod(H*z',2)';
if (sum(z)==0)
    %break
end
index = 0;

for j=1:N %down columns
    for k = 1:dv(j)  %%for each message
        Ma(Row(index+k,1),j) = r(j);
        for l = 1:dv(j) %%sum on other messages
            if (k ~= l)
                Ma(Row(index+k,1),j) = Ma(Row(index+k,1),j) +E(Row(index+l,1),j);
                if (Ma(Row(index+k,1),j) > 7.9375)
                    Ma(Row(index+k,1),j) = 7.9375;
                end
                if (Ma(Row(index+k,1),j) < -7.9375)
                    Ma(Row(index+k,1),j) = -7.9375;
                end
            end
        end
```

```
        end
        index = index + dv(j);
    end

end
```

LDPCTest.m

```
% LDPC test
% Generates a parity check matrix and sends all 0's in the presence of
%AWGN noise
%clear all;close all;
global E Ma
%%Initialisation
randn('state',0);
niter_decoder = 10; %Number of decoder iterations
n = 1e6;             %Number of message iterations

dv=3;
dc=6;
N=480;
M=240;

%Serial comms settings
s = serial('com1');
s.BaudRate = 28800;
s.Parity = 'even';
s.FlowControl = 'none';
fopen(s);
%H = genH(N,M,dv,dc);
%load H;
```

```
EbNodB=linspace(2,10,9)
EbNo=10.^(EbNodB/10);
ber=zeros(length(EbNo),niter_decoder);
ber_mine = ber;
incorrect = 0;
NotInf=zeros(length(EbNo));
rate=1/2;
Eb=1;
a = zeros(1,8);
for i = 1:length(EbNo)

    N0=Eb/(EbNo(i)*rate);
    sig = sqrt(N0/2);
    for j=1:n

        Y = ones(1,length(N)) + sig*randn(1,N); %%Send all 0's with awgn
        r=EbNo(i)*4*Y/2;  %rate = 1/2 ebno is really ecno
        %make sure the range of the channel fits in the decoders range
        r = round(r*16)/16;
        r(r>7.9375) = 7.9375;
        r(r<-7.9375) = -7.9375;
        r = r*16;

        %convert to signed decimal IE (+-)xxx.xxxx
        for ij = 1:8
            a(ij) = r(ij);
            if (r(ij) <0)
                a(ij) = 256 + a(ij);
            end
```

```
        end
        fwrite(s,a);
        %wait until decoder finishes
        while(s.BytesAvailable == 0)
        end

        result = fread(s,s.BytesAvailable);
        t = 0;
        %convert answer to binary
        b = dec2bin(result(1),8);
        for ij = 1:8
            if b(ij) == '1'
                t = t+1;
            end
        end
        %compare with sumulation
          tmp=LDPCSimF(H,EbNo(i),niter_decoder,Y);
          if (sum(isnan(tmp)) ==0)
              ber(i,:)= ber(i,:) +tmp;
              NotInf(i) = NotInf(i) +1;
          end
        ber_mine(i,:)= ber_mine(i,:) +t;

    end
%       ber(i,:)=ber(i,:)/(j*N);
    ber_mine(i,10)=ber_mine(i,10)/(j*N)
    i

end
```

Appendix B

VHDL Code

B.1 ψLookup Table

LUT.vhdl

```
--LUT for psi
--David F Hayes
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity LUT2 is
port(
input : IN std_logic_vector(7 downto 0);
output : OUT std_logic_vector(6 downto 0));
end LUT2;

architecture behavior of LUT2 is
begin

process(input)
VARIABLE inp : INTEGER range -128 to 127;
```

```
begin

inp := TO_INTEGER(SIGNED(input));
case inp is
when -128 =>
output <= "0000000";
when -127 =>
output <= "0000000";
when -126 =>
output <= "0000000";
when -125=>
output <= "0000000";
when -124=>
output <= "0000000";
when -123=>
output <= "0000000";
when -122=>
output <= "0000000";
when -121=>
output <= "0000000";
when -120=>
output <= "0000000";
when -119 =>
output <= "0000000";
when -118 =>
output <= "0000000";
when -117 =>
output <= "0000000";
when -116 =>
output <= "0000000";
when -115 =>
```

```
output <= "0000000";
when -114 =>
output <= "0000000";
when -113 =>
output <= "0000000";
when -112 =>
output <= "0000000";
when -111 =>
output <= "0000000";
when -110 =>
output <= "0000000";
when -109 =>
output <= "0000000";
when -108 =>
output <= "0000000";
when -107 =>
output <= "0000000";
when -106 =>
output <= "0000000";
when -105 =>
output <= "0000000";
when -104 =>
output <= "0000000";
when -103 =>
output <= "0000000";
when -102 =>
output <= "0000000";
when -101 =>
output <= "0000000";
when -100 =>
output <= "0000000";
```

```
when -99 =>
output <= "0000000";
when -98 =>
output <= "0000000";
when -97 =>
output <= "0000001";
when -96=>
output <= "0000001";
when -95=>
output <= "0000001";
when -94=>
output <= "0000001";
when -93=>
output <= "0000001";
when -92=>
output <= "0000001";
when -91=>
output <= "0000001";
when -90=>
output <= "0000001";
when -89=>
output <= "0000001";
when -88=>
output <= "0000001";
when -87=>
output <= "0000001";
when -86=>
output <= "0000001";
when -85=>
output <= "0000001";
when -84 =>
```

```
output <= "0000001";
when -83 =>
output <= "0000001";
when -82 =>
output <= "0000001";
when -81 =>
output <= "0000001";
when -80 =>
output <= "0000001";
when -79 =>
output <= "0000001";
when -78 =>
output <= "0000001";
when -77 =>
output <= "0000001";
when -76 =>
output <= "0000001";
when -75 =>
output <= "0000001";
when -74 =>
output <= "0000001";
when -73 =>
output <= "0000001";
when -72 =>
output <= "0000001";
when -71 =>
output <= "0000001";
when -70 =>
output <= "0000001";
when -69 =>
output <= "0000001";
```

```
when -68 =>
output <= "0000001";
when -67 =>
output <= "0000001";
when -66 =>
output <= "0000001";
when -65 =>
output <= "0000001";
when -64 =>
output <= "0000001";
when -63 =>
output <= "0000001";
when -62 =>
output <= "0000001";
when -61 =>
output <= "0000001";
when -60 =>
output <= "0000001";
when -59 =>
output <= "0000001";
when -58 =>
output <= "0000001";
when -57 =>
output <= "0000001";
when -56 =>
output <= "0000010";
when -55 =>
output <= "0000010";
when -54 =>
output <= "0000010";
when -53 =>
```

```
output <= "0000010";
when -52 =>
output <= "0000010";
when -51 =>
output <= "0000010";
when -50 =>
output <= "0000010";
when -49 =>
output <= "0000010";
when -48 =>
output <= "0000010";
when -47 =>
output <= "0000010";
when -46 =>
output <= "0000010";
when -45 =>
output <= "0000010";
when -44 =>
output <= "0000011";
when -43 =>
output <= "0000011";
when -42 =>
output <= "0000011";
when -41 =>
output <= "0000011";
when -40 =>
output <= "0000011";
when -39 =>
output <= "0000011";
when -38 =>
output <= "0000011";
```

```
when -37 =>
output <= "0000011";
when -36 =>
output <= "0000100";
when -35 =>
output <= "0000100";
when -34 =>
output <= "0000100";
when -33 =>
output <= "0000100";
when -32 =>
output <= "0000100";
when -31 =>
output <= "0000101";
when -30 =>
output <= "0000101";
when -29 =>
output <= "0000110";
when -28 =>
output <= "0000110";
when -27 =>
output <= "0000111";
when -26 =>
output <= "0000111";
when -25 =>
output <= "0001000";
when -24 =>
output <= "0001000";
when -23 =>
output <= "0001001";
when -22 =>
```

```
output <= "0001001";
when -21 =>
output <= "0001010";
when -20 =>
output <= "0001010";
when -19 =>
output <= "0001011";
when -18 =>
output <= "0001011";
when -17 =>
output <= "0001100";
when -16 =>
output <= "0001100";
when -15  =>
output <= "0001101";
when -14  =>
output <= "0001110";
when -13 =>
output <= "0001111";
when -12  =>
output <= "0010000";
when -11  =>
output <= "0010010";
when -10  =>
output <= "0010100";
when -9  =>
output <= "0010110";
when -8  =>
output <= "0011000";
when -7  =>
output <= "0011010";
```

```
when -6  =>
output <= "0011100";
when -5  =>
output <= "0011110";
when -4  =>
output <= "0100000";
when -3  =>
output <= "0101000";
when -2  =>
output <= "0101111";
when -1  =>
output <= "1001111";
when 0  =>
output <= "1111111";
when 127 =>
output <= "0000000";
when 126 =>
output <= "0000000";
when 125=>
output <= "0000000";
when 124=>
output <= "0000000";
when 123=>
output <= "0000000";
when 122=>
output <= "0000000";
when 121=>
output <= "0000000";
when 120=>
output <= "0000000";
when 119 =>
```

```
output <= "0000000";
when 118 =>
output <= "0000000";
when 117 =>
output <= "0000000";
when 116 =>
output <= "0000000";
when 115 =>
output <= "0000000";
when 114 =>
output <= "0000000";
when 113 =>
output <= "0000000";
when 112 =>
output <= "0000000";
when 111 =>
output <= "0000000";
when 110 =>
output <= "0000000";
when 109 =>
output <= "0000000";
when 108 =>
output <= "0000000";
when 107 =>
output <= "0000000";
when 106 =>
output <= "0000000";
when 105 =>
output <= "0000000";
when 104 =>
output <= "0000000";
```

```
when 103 =>
output <= "0000000";
when 102 =>
output <= "0000000";
when 101 =>
output <= "0000000";
when 100 =>
output <= "0000000";
when 99 =>
output <= "0000000";
when 98 =>
output <= "0000000";
when 97 =>
output <= "0000001";
when 96=>
output <= "0000001";
when 95=>
output <= "0000001";
when 94=>
output <= "0000001";
when 93=>
output <= "0000001";
when 92=>
output <= "0000001";
when 91=>
output <= "0000001";
when 90=>
output <= "0000001";
when 89=>
output <= "0000001";
when 88=>
```

```
output <= "0000001";
when 87=>
output <= "0000001";
when 86=>
output <= "0000001";
when 85=>
output <= "0000001";
when 84 =>
output <= "0000001";
when 83 =>
output <= "0000001";
when 82 =>
output <= "0000001";
when 81 =>
output <= "0000001";
when 80 =>
output <= "0000001";
when 79 =>
output <= "0000001";
when 78 =>
output <= "0000001";
when 77 =>
output <= "0000001";
when 76 =>
output <= "0000001";
when 75 =>
output <= "0000001";
when 74 =>
output <= "0000001";
when 73 =>
output <= "0000001";
```

```
when 72 =>
output <= "0000001";
when 71 =>
output <= "0000001";
when 70 =>
output <= "0000001";
when 69 =>
output <= "0000001";
when 68 =>
output <= "0000001";
when 67 =>
output <= "0000001";
when 66 =>
output <= "0000001";
when 65 =>
output <= "0000001";
when 64 =>
output <= "0000001";
when 63 =>
output <= "0000001";
when 62 =>
output <= "0000001";
when 61 =>
output <= "0000001";
when 60 =>
output <= "0000001";
when 59 =>
output <= "0000001";
when 58 =>
output <= "0000001";
when 57 =>
```

```
output <= "0000001";
when 56 =>
output <= "0000010";
when 55 =>
output <= "0000010";
when 54 =>
output <= "0000010";
when 53 =>
output <= "0000010";
when 52 =>
output <= "0000010";
when 51 =>
output <= "0000010";
when 50 =>
output <= "0000010";
when 49 =>
output <= "0000010";
when 48 =>
output <= "0000010";
when 47 =>
output <= "0000010";
when 46 =>
output <= "0000010";
when 45 =>
output <= "0000010";
when 44 =>
output <= "0000011";
when 43 =>
output <= "0000011";
when 42 =>
output <= "0000011";
```

```
when 41 =>
output <= "0000011";
when 40 =>
output <= "0000011";
when 39 =>
output <= "0000011";
when 38 =>
output <= "0000011";
when 37 =>
output <= "0000011";
when 36 =>
output <= "0000100";
when 35 =>
output <= "0000100";
when 34 =>
output <= "0000100";
when 33 =>
output <= "0000100";
when 32 =>
output <= "0000100";
when 31 =>
output <= "0000101";
when 30 =>
output <= "0000101";
when 29 =>
output <= "0000110";
when 28 =>
output <= "0000110";
when 27 =>
output <= "0000111";
when 26 =>
```

```
output <= "0000111";
when 25 =>
output <= "0001000";
when 24 =>
output <= "0001000";
when 23 =>
output <= "0001001";
when 22 =>
output <= "0001001";
when 21 =>
output <= "0001010";
when 20 =>
output <= "0001010";
when 19 =>
output <= "0001011";
when 18 =>
output <= "0001011";
when 17 =>
output <= "0001100";
when 16 =>
output <= "0001100";
when 15  =>
output <= "0001101";
when 14  =>
output <= "0001110";
when 13 =>
output <= "0001111";
when 12  =>
output <= "0010000";
when 11  =>
output <= "0010010";
```

```
when 10  =>
output <= "0010100";
when 9  =>
output <= "0010110";
when 8  =>
output <= "0011000";
when 7  =>
output <= "0011010";
when 6  =>
output <= "0011100";
when 5  =>
output <= "0011110";
when 4  =>
output <= "0100000";
when 3  =>
output <= "0101000";
when 2  =>
output <= "0101111";
when 1  =>
output <= "1001111";

end case;
end process;
end behavior;
```

B.2 Carry Lookahead Adder

look_ahead_adder.vhd

```
library IEEE;
use ieee.std_logic_1164.all;
```

```
entity Look_Ahead_Adder is
port( A, B :  in std_logic_vector( 7 downto 0 );
--carry_in : in std_logic;
--carry_out :  out std_logic;
S  : out std_logic_vector( 7 downto 0 ) );
end Look_Ahead_Adder;


architecture RTL of Look_Ahead_Adder is
component carry_generator
port( P, G  : in std_logic_vector(7 downto 0);
--C1 : in std_logic;
C : out std_logic_vector(8 downto 0));
end component;

component half_adder
port( A, B : in std_logic_vector( 7 downto 0 );
P, G : out std_logic_vector( 7 downto 0) );
end component;
For CG : carry_generator Use entity work.carry_generator(RTL);
For HA : half_adder Use entity work.half_adder(RTL);
signal tempG, tempP :  std_logic_vector( 7 downto 0 );
signal tempC  : std_logic_vector( 8 downto 0 );
begin
HA: half_adder port map( A=>A, B=>B, P =>tempP, G=>tempG );
CG: carry_generator port map( P=>tempP, G=>tempG, C=>tempC );
with tempC(8 downto 7) select S <=
"01111111" when "01",
"10000001" when "10",
(tempC(7 downto 0 ) xor tempP) when others;
--S <= tempC( 7 downto 0 ) xor tempP;
```

```
--carry_out <= tempC(8);
end;
```

carry_generator.vhd

```
library IEEE;
use ieee.std_logic_1164.all;
entity carry_generator is
port( P , G : in std_logic_vector(7 downto 0);
--C1 : in std_logic;
C : out std_logic_vector(8 downto 0));
end carry_generator;
architecture RTL of carry_generator is
begin
process(P, G)
variable tempC : std_logic_vector(8 downto 0);
begin
tempC(0) := '0';
for i in 0 to 7 loop
tempC(i+1) := G(i) or (P(i) and tempC(i));
end loop;
C <= tempC;
end process;
end;
```

half_adder.vhd

```
%Half adder
library IEEE;
use ieee.std_logic_1164.all;
entity half_adder is
port( A, B : in std_logic_vector( 7 downto 0 );
P, G : out std_logic_vector( 7 downto 0 ) );
end half_adder;
```

```
architecture RTL of half_adder is
begin
P <= A xor B;
G <= A and B;
end;
```

B.3 Bit Node Processor

Vadder_ram.vhd

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity Vadder_ram is
port(
clk   : IN std_logic;
wenable : IN std_logic;
sel_r : IN std_logic;
sel_w : IN std_logic;
addr : IN integer range 0 to 2;
A : IN std_logic_vector(7 downto 0);


X0 : OUT std_logic_vector(7 downto 0);
X1 : OUT std_logic_vector(7 downto 0);
X2 : OUT std_logic_vector(7 downto 0));
end Vadder_ram;

architecture behavior of Vadder_ram is
TYPE vector_array IS ARRAY (0 TO 2) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL memory0 : vector_array;
```

```
SIGNAL memory1 : vector_array;
begin
process(clk,sel_r,sel_w,addr,A,wenable,memory0,memory1)
begin
if (clk'event AND clk='1') then
if (wenable='1') then
if (sel_w='0') then
memory0(addr) <= A;
else
memory1(addr) <= A;
end if;
end if;
end if;
if (sel_r='1') then
X0 <= memory1(0);
X1 <= memory1(1);
X2 <= memory1(2);
else
X0 <= memory0(0);
X1 <= memory0(1);
X2 <= memory0(2);
end if;
end process;
end behavior;
```

Vcontrol.vhd

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity Vcontrol2 is
port(
```

```
clk : IN std_logic;
reset : IN std_logic;
lam_deg : IN integer range 0 to 3;
send_NR : IN std_logic; --1   SEND EDGES, 0 RECIEVE EDGES

edge_address : OUT std_logic_vector(7 downto 0);
lambda_address : OUT std_logic_vector(5 downto 0);
dv_address : OUT std_logic_vector(5 downto 0);
mem_sel_r : BUFFER std_logic;
mem_sel_w : BUFFER std_logic;
adder_mux : OUT std_logic_vector(0 to 2);
adder_address : OUT std_logic_vector(1 downto 0);
adder_wr : OUT std_logic;
edge_wr : OUT std_logic);
end Vcontrol2;

architecture behavior of Vcontrol2 is
type state_type is (rst,zero,one,two,recv);
signal state_w : state_type;
signal state_r : state_type;
begin
process(clk,reset)
VARIABLE edg_address_base : integer range 0 to 255;
VARIABLE  lam_address  : integer range 0 to 63;
VARIABLE  dv_addres   : integer range 0 to 63;
VARIABLE dv_previous : integer range 0 to 3;
VARIABLE dv_current : integer range 0 to 3;
VARIABLE dv_current_r : integer range 0 to 3;
begin
if (reset='1') then
state_r <= rst;
```

```
state_w <= rst;

elsif (clk'event AND clk='1') then
case state_w is
when rst =>
lambda_address <= "111111";
lam_address := 63;
dv_address <= "000000";
dv_addres := 0;
mem_sel_r <= '1';
mem_sel_w <= '0';
adder_mux <= "000";
adder_address <= "00";
dv_current := 3;
dv_previous :=3;
dv_current_r :=3;
if (send_NR = '1') then
state_w <= zero;
edg_address_base := lam_deg -1;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
adder_wr <= '1';
edge_wr <= '0';
else
state_w <= recv;
edg_address_base :=0;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
adder_wr <= '0';
edge_wr <= '1';
end if;
when zero =>
dv_previous := dv_current;
```

```
dv_current := lam_deg;
state_w <= one;
mem_sel_w <= NOT mem_sel_w;

if (edg_address_base = 2) then
dv_addres := dv_addres+1;
end if;


edge_address <= std_logic_vector(to_unsigned(edg_address_base-1,8));
adder_address <="00";
dv_address <=std_logic_vector(to_unsigned(dv_addres,6));
adder_wr <= '1';
edge_wr <= '0';

when one =>
adder_address <="01";


if (dv_current = 2) then
adder_mux(2) <='0';
state_w <= zero;
edg_address_base := edg_address_base+lam_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
dv_addres := dv_addres+1;
else
state_w <= two;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-2,8));
end if;
dv_address <=std_logic_vector(to_unsigned(dv_addres,6));
adder_wr <= '1';
```

```
edge_wr <= '0';

when two =>
adder_address <="10";
edg_address_base := edg_address_base+lam_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
state_w <= zero;
dv_addres := dv_addres+1;
dv_address <=std_logic_vector(to_unsigned(dv_addres,6));
adder_wr <= '1';
edge_wr <= '0';

when recv =>
adder_wr <= '0';
edge_wr <= '1';
edg_address_base := edg_address_base +1;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
lambda_address <= std_logic_vector(to_unsigned(edg_address_base,6));
dv_address <= "000000";
mem_sel_r <= '1';
mem_sel_w <= '0';
adder_mux <= "000";
adder_address <= "00";
state_w <=recv;
end case;
case state_r is
when rst =>
if (send_NR = '1') then
state_r <= zero;
else
state_r <= recv;
```

```
end if;
when zero =>
mem_sel_r <= NOT mem_sel_r;
adder_mux <= "110";
if (dv_current_r = 2) then
adder_mux(0) <= '0';
end if;
lambda_address <= std_logic_vector(to_unsigned(lam_address,6));
state_r <= one;


when one =>
adder_mux <= "101";
if (dv_current_r = 2) then
adder_mux(0) <= '0';
lam_address := lam_address +1;

dv_current_r := dv_previous;
state_r <= zero;
else
state_r <= two;
end if;
lambda_address <= std_logic_vector(to_unsigned(lam_address,6));

when two =>
adder_mux <="011";
lam_address := lam_address +1;
dv_current_r := dv_previous;
state_r <= zero;
lambda_address <= std_logic_vector(to_unsigned(lam_address,6));
when recv =>
```

```
state_r <=recv;
end case;

end if;
end process;
end behavior;
```

B.4 Check Node Processor

nodecontrol.vhd

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity Nodecontrol is
port(
clk : IN std_logic;
reset : IN std_logic;
state_reset : IN std_logic;
v_deg : IN integer range 0 to 8;
send_NR : IN std_logic; --1   SEND EDGES, 0 RECIEVE EDGES

edge_address : OUT std_logic_vector(7 downto 0);
dc_address : OUT std_logic_vector(4 downto 0);
mem_sel_r : BUFFER std_logic;
mem_sel_w : BUFFER std_logic;
adder_mux : OUT std_logic_vector(0 to 5);
adder_address : OUT std_logic_vector(2 downto 0);
adder_wr : OUT std_logic;
edge_wr : OUT std_logic);
end Nodecontrol;
```

```
architecture behavior of Nodecontrol is
type state_type1 is (rst,zero,one,two,three,four,five,recv);
type state_type2 is (rst,zero,one,two,three,four,five,recv);
signal state_w : state_type1;
signal state_r : state_type2;
begin
process(clk,reset,send_NR,v_deg)
VARIABLE edg_address_base : integer range 0 to 255;
VARIABLE  dc_addres   : integer range 0 to 31;
VARIABLE dc_previous : integer range 0 to 8;
VARIABLE dc_current : integer range 0 to 8;
VARIABLE dc_current_r : integer range 0 to 8;
begin
if (reset='1') then
dc_address <= "00000";
dc_addres := 0;
mem_sel_r <= '1';
mem_sel_w <= '0';
adder_mux <= "000000";
adder_address <= "000";
dc_current := 6;
dc_previous :=6;
dc_current_r :=6;
state_w <= rst;
state_r <= rst;
edg_address_base := v_deg -1;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
adder_wr <= '1';
edge_wr <= '0';
elsif (clk'event AND clk='1') then
```

```
case state_w is
when rst =>
edg_address_base := v_deg -1;
adder_wr <= '1';
edge_wr <= '0';
dc_current := 6;
dc_previous :=6;
dc_current_r :=6;
if (send_NR = '1') then
state_w <= zero;
else
state_w <= recv;
edg_address_base := 255;

end if;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
if (state_reset = '1') then
state_w <= rst;
end if;
when zero =>
dc_previous := dc_current;
dc_current := v_deg;
state_w <= one;
mem_sel_w <= NOT mem_sel_w;

if (edg_address_base = 7) then
dc_addres := dc_addres+1;
end if;


edge_address <= std_logic_vector(to_unsigned(edg_address_base-1,8));
```

```
adder_address <="000";
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;
when one =>
adder_address <="001";


if (dc_current = 2) then
state_w <= zero;
edg_address_base := edg_address_base+v_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
dc_addres := dc_addres+1;
else
state_w <= two;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-2,8));
end if;
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;
when two =>
adder_address <="010";


if (dc_current = 3) then
```

```
state_w <= zero;
edg_address_base := edg_address_base+v_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
dc_addres := dc_addres+1;
else
state_w <= three;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-3,8));
end if;
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;
when three =>
adder_address <="011";


if (dc_current = 4) then

state_w <= zero;
edg_address_base := edg_address_base+v_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
dc_addres := dc_addres+1;
else
state_w <= four;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-4,8));
end if;
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
```

```
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;
when four =>
adder_address <="100";


if (dc_current = 5) then

state_w <= zero;
edg_address_base := edg_address_base+v_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
dc_addres := dc_addres+1;
else
state_w <= five;
edge_address <= std_logic_vector(to_unsigned(edg_address_base-5,8));
end if;
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;
when five =>
adder_address <="101";


state_w <= zero;
edg_address_base := edg_address_base+v_deg;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
```

```
dc_addres := dc_addres+1;
dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
adder_wr <= '1';
edge_wr <= '0';
if (state_reset = '1') then
state_w <= rst;
end if;

--when six =>
--adder_address <="110";
--if (dc_current = 7) then
--
--state_w <= zero;
--edg_address_base := edg_address_base+v_deg;
--edge_address <= std_logic_vector(to_unsigned(edg_address_base,5));
--dc_addres := dc_addres+1;
--else
--state_w <= seven;
--edge_address <= std_logic_vector(to_unsigned(edg_address_base-7,5));
--end if;
--dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
--adder_wr <= '1';
--edge_wr <= '0';
--when seven =>
--
--adder_address <="111";
--edg_address_base := edg_address_base+v_deg;
--edge_address <= std_logic_vector(to_unsigned(edg_address_base,5));
--state_w <= zero;
--dc_addres := dc_addres+1;
--dc_address <=std_logic_vector(to_unsigned(dc_addres,5));
```

```
--adder_wr <= '1';
--edge_wr <= '0';
when recv =>
adder_wr <= '0';
edge_wr <= '1';
edg_address_base := edg_address_base +1;
edge_address <= std_logic_vector(to_unsigned(edg_address_base,8));
dc_address <= std_logic_vector(to_unsigned(edg_address_base,5));
mem_sel_r <= '1';
mem_sel_w <= '0';
adder_mux <= "000000";
adder_address <= "000";
state_w <=recv;
if (state_reset = '1') then
state_w <= rst;
end if;

end case;
case state_r is
when rst =>
if (send_NR = '1') then
state_r <= zero;
else
state_r <= recv;
end if;
if (state_reset = '1') then
state_r <= rst;
end if;
when zero =>
adder_mux <= "111110";
state_r <= one;
```

```
mem_sel_r <= NOT mem_sel_r;
if (state_reset = '1') then
state_r <= rst;
end if;

when one =>
adder_mux <= "111101";
if (dc_current_r = 2) then
dc_current_r := dc_previous;
state_r <= zero;
else
state_r <= two;
end if;
if (state_reset = '1') then
state_r <= rst;
end if;

when two =>
adder_mux <= "111011";
if (dc_current_r = 3) then

dc_current_r := dc_previous;
state_r <= zero;
else
state_r <= three;
end if;
if (state_reset = '1') then
state_r <= rst;
end if;

when three =>
```

```
adder_mux <= "110111";
if (dc_current_r = 4) then

dc_current_r := dc_previous;
state_r <= zero;
else
state_r <= four;
end if;
if (state_reset = '1') then
state_r <= rst;
end if;

when four =>
adder_mux <= "101111";
if (dc_current_r = 5) then

dc_current_r := dc_previous;
state_r <= zero;
else
state_r <= five;
end if;
if (state_reset = '1') then
state_r <= rst;
end if;

when five =>
adder_mux <= "011111";
dc_current_r := dc_previous;
state_r <= zero;
if (state_reset = '1') then
state_r <= rst;
```

```
end if;

--when six =>
--adder_mux <= "10111111";
--if (dc_current_r = 7) then
--dc_current_r := dc_previous;
--state_r <= zero;
--else
--state_r <= seven;
--end if;

--when seven =>
--adder_mux <="01111111";


--dc_current_r := dc_previous;
--state_r <= zero;
when recv =>
state_r <=recv;
if (state_reset = '1') then
state_r <= rst;
end if;
end case;
if (dc_current_r = 2) then
adder_mux(0 to 5) <="000000";
elsif (dc_current_r = 3) then
adder_mux(0 to 4) <="00000";
elsif (dc_current_r = 4) then
adder_mux(0 to 3) <="0000";
elsif (dc_current_r = 5) then
adder_mux(0 to 2) <="000";
```

```
--elsif (dc_current_r = 6) then
--adder_mux(0 to 1) <="00";
--elsif (dc_current_r = 7) then
--adder_mux(0) <='0';
end if;
end if;
end process;
end behavior;
```

B.5 Codeword Loader

lambda_write.vhd

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity lambda_wr is
port(
clk : IN std_logic;
reset : IN std_logic;
in_ready : IN std_logic; --latched high when data is ready from outside world, reset
control_ready : IN std_logic; --the control logic is ready to accept some more inputs


lambda_address : OUT std_logic_vector(8 downto 0);
done_reading : OUT std_logic; --we're done reading and the control logic can take it
rst_latch : OUT std_logic; --goes high for one cycle after we're finished reading to
lambda_write : OUT std_logic);

end lambda_wr;
```

```
architecture behavior of lambda_wr is
type state_type is (idle,processing,finish);
signal state : state_type;
begin
process(clk,reset)
VARIABLE  lam_address  : integer range 0 to 511;
begin
if (reset = '1') then
lambda_write <='0';
lambda_address <= "000000000";
lam_address := 0;
done_reading <= '0';
rst_latch <= '0';
state <= idle;
elsif (clk'event AND clk='1') then
case state is
when idle =>
lambda_write <='0';
rst_latch <= '0';
lambda_address <= "000000000";
lam_address :=0;
if (in_ready = '1' AND control_ready = '1') then
state <= processing;
else
state <=idle;
end if;
done_reading <= '0';
when processing =>
lambda_write <='1';
done_reading <= '0';
rst_latch <= '0';
```

```
if (lam_address = 480) then
state <= finish;
else
state <=processing;
end if;
lambda_address <= std_logic_vector(to_unsigned(lam_address,9));
lam_address := lam_address + 1;
when finish =>
done_reading <= '1';
rst_latch <= '1';
lambda_write <='0';
lambda_address <="000000000";
state <= idle;
end case;
end if;
end process;
end behavior;
```

Appendix C

Prototype Decoder Test Matrix and Vector

C.1 Transmitted Codeword

0.0625000000000000 0.125000000000000 0.187500000000000
0.250000000000000 0.312500000000000 0.375000000000000
0.437500000000000 0.500000000000000 0.562500000000000
0.625000000000000 0.687500000000000 0.750000000000000
0.812500000000000 0.875000000000000 0.937500000000000
1 1.06250000000000 1.12500000000000 1.18750000000000
1.25000000000000 1.31250000000000 1.37500000000000
1.43750000000000 1.50000000000000 1.56250000000000
1.62500000000000 1.68750000000000 1.75000000000000
1.81250000000000 1.87500000000000 1.93750000000000
2 2.06250000000000 2.12500000000000
2.18750000000000 2.25000000000000 2.31250000000000
2.37500000000000 2.43750000000000 2.50000000000000
2.56250000000000 2.62500000000000 2.68750000000000
2.75000000000000 2.81250000000000 2.87500000000000

```
2.93750000000000 3 3.06250000000000
3.12500000000000 3.18750000000000 3.25000000000000
3.31250000000000 3.37500000000000 3.43750000000000
3.50000000000000 3.56250000000000 3.62500000000000
3.68750000000000 3.75000000000000 3.81250000000000
3.87500000000000 3.93750000000000 4
4.06250000000000 4.12500000000000 4.18750000000000
4.25000000000000 4.31250000000000 4.37500000000000
4.43750000000000 4.50000000000000 4.56250000000000
4.62500000000000 4.68750000000000 4.75000000000000
4.81250000000000 4.87500000000000 4.93750000000000
5 5.06250000000000 5.12500000000000
5.18750000000000 5.25000000000000 5.31250000000000
5.37500000000000 5.43750000000000 5.50000000000000
5.56250000000000 5.62500000000000 5.68750000000000
5.75000000000000 5.81250000000000 5.87500000000000
5.93750000000000 6 6.06250000000000
6.12500000000000 6.18750000000000 6.25000000000000
6.31250000000000 6.37500000000000 6.43750000000000
6.50000000000000 6.56250000000000 6.62500000000000
6.68750000000000 6.75000000000000 6.81250000000000
6.87500000000000 6.93750000000000 7
7.06250000000000 7.12500000000000 7.18750000000000
7.25000000000000 7.31250000000000 7.37500000000000
7.43750000000000 7.50000000000000 7.56250000000000
7.62500000000000 7.68750000000000 7.75000000000000
7.81250000000000 7.87500000000000 7.93750000000000
-8 -7.93750000000000 -7.87500000000000
-7.81250000000000 -7.75000000000000 -7.68750000000000
-7.62500000000000 -7.56250000000000 -7.50000000000000
-7.43750000000000 -7.37500000000000 -7.31250000000000
```

-7.25000000000000 -7.18750000000000 -7.12500000000000
-7.06250000000000 -7 -6.93750000000000
-6.87500000000000 -6.81250000000000 -6.75000000000000
-6.68750000000000 -6.62500000000000 -6.56250000000000
-6.50000000000000 -6.43750000000000 -6.37500000000000
-6.31250000000000 -6.25000000000000 -6.18750000000000
-6.12500000000000 -6.06250000000000 -6
-5.93750000000000 -5.87500000000000 -5.81250000000000
-5.75000000000000 -5.68750000000000 -5.62500000000000
-5.56250000000000 -5.50000000000000 -5.43750000000000
-5.37500000000000 -5.31250000000000 -5.25000000000000
-5.18750000000000 -5.12500000000000 -5.06250000000000
-5 -4.93750000000000 -4.87500000000000
-4.81250000000000 -4.75000000000000 -4.68750000000000
-4.62500000000000 -4.56250000000000 -4.50000000000000
-4.43750000000000 -4.37500000000000 -4.31250000000000
-4.25000000000000 -4.18750000000000 -4.12500000000000
-4.06250000000000 -4 -3.93750000000000
-3.87500000000000 -3.81250000000000 -3.75000000000000
-3.68750000000000 -3.62500000000000 -3.56250000000000
-3.50000000000000 -3.43750000000000 -3.37500000000000
-3.31250000000000 -3.25000000000000 -3.18750000000000
-3.12500000000000 -3.06250000000000 -3 -2.93750000000000
-2.87500000000000 -2.81250000000000 -2.75000000000000
-2.68750000000000 -2.62500000000000 -2.56250000000000
-2.50000000000000 -2.43750000000000 -2.37500000000000
-2.31250000000000 -2.25000000000000 -2.18750000000000
-2.12500000000000 -2.06250000000000 -2
-1.93750000000000 -1.87500000000000 -1.81250000000000
-1.75000000000000 -1.68750000000000 -1.62500000000000
-1.56250000000000 -1.50000000000000 -1.43750000000000

```
-1.37500000000000 -1.31250000000000 -1.25000000000000
-1.18750000000000 -1.12500000000000 -1.06250000000000
-1 -0.937500000000000 -0.875000000000000
-0.812500000000000 -0.750000000000000
-0.687500000000000 -0.625000000000000 -0.562500000000000
-0.500000000000000 -0.437500000000000 -0.375000000000000
-0.312500000000000 -0.250000000000000 -0.187500000000000
-0.125000000000000 -0.0625000000000000 0
0.0625000000000000 0.125000000000000 0.187500000000000
0.250000000000000 0.312500000000000 0.375000000000000
0.437500000000000 0.500000000000000 0.562500000000000
0.625000000000000 0.687500000000000 0.750000000000000
0.812500000000000 0.875000000000000 0.937500000000000
1 1.06250000000000 1.12500000000000 1.18750000000000
1.25000000000000 1.31250000000000 1.37500000000000
1.43750000000000 1.50000000000000 1.56250000000000
1.62500000000000 1.68750000000000 1.75000000000000
1.81250000000000 1.87500000000000 1.93750000000000
2 2.06250000000000 2.12500000000000
2.18750000000000 2.25000000000000 2.31250000000000
2.37500000000000 2.43750000000000 2.50000000000000
2.56250000000000 2.62500000000000 2.68750000000000
2.75000000000000 2.81250000000000 2.87500000000000
2.93750000000000 3 3.06250000000000
3.12500000000000 3.18750000000000 3.25000000000000
3.31250000000000 3.37500000000000 3.43750000000000
3.50000000000000 3.56250000000000 3.62500000000000
3.68750000000000 3.75000000000000 3.81250000000000
3.87500000000000 3.93750000000000 4 4.06250000000000
4.12500000000000 4.18750000000000 4.25000000000000
4.31250000000000 4.37500000000000 4.43750000000000
```

4.50000000000000 4.56250000000000 4.62500000000000
4.68750000000000 4.75000000000000 4.81250000000000
4.87500000000000 4.93750000000000 5 5.06250000000000
5.12500000000000 5.18750000000000 5.25000000000000
5.31250000000000 5.37500000000000 5.43750000000000
5.50000000000000 5.56250000000000 5.62500000000000
5.68750000000000 5.75000000000000 5.81250000000000
5.87500000000000 5.93750000000000 6
6.06250000000000 6.12500000000000 6.18750000000000
6.25000000000000 6.31250000000000 6.37500000000000
6.43750000000000 6.50000000000000 6.56250000000000
6.62500000000000 6.68750000000000 6.75000000000000
6.81250000000000 6.87500000000000 6.93750000000000
7 7.06250000000000 7.12500000000000 7.18750000000000
7.25000000000000 7.31250000000000 7.37500000000000
7.43750000000000 7.50000000000000 7.56250000000000
7.62500000000000 7.68750000000000 7.75000000000000
7.81250000000000 7.87500000000000 7.93750000000000
-8 -7.93750000000000 -7.87500000000000
-7.81250000000000 -7.75000000000000 -7.68750000000000
-7.62500000000000 -7.56250000000000 -7.50000000000000
-7.43750000000000 -7.37500000000000 -7.31250000000000
-7.25000000000000 -7.18750000000000 -7.12500000000000
-7.06250000000000 -7 -6.93750000000000
-6.87500000000000 -6.81250000000000 -6.75000000000000
-6.68750000000000 -6.62500000000000 -6.56250000000000
-6.50000000000000 -6.43750000000000 -6.37500000000000
-6.31250000000000 -6.25000000000000 -6.18750000000000
-6.12500000000000 -6.06250000000000 -6
-5.93750000000000 -5.87500000000000 -5.81250000000000
-5.75000000000000 -5.68750000000000 -5.62500000000000

```
-5.56250000000000 -5.50000000000000 -5.43750000000000
-5.37500000000000 -5.31250000000000 -5.25000000000000
-5.18750000000000 -5.12500000000000 -5.06250000000000
-5 -4.93750000000000 -4.87500000000000
-4.81250000000000 -4.75000000000000 -4.68750000000000
-4.62500000000000 -4.56250000000000 -4.50000000000000
-4.43750000000000 -4.37500000000000 -4.31250000000000
-4.25000000000000 -4.18750000000000 -4.12500000000000
-4.06250000000000 -4 -3.93750000000000
-3.87500000000000 -3.81250000000000 -3.75000000000000
-3.68750000000000 -3.62500000000000 -3.56250000000000
-3.50000000000000 -3.43750000000000 -3.37500000000000
-3.31250000000000 -3.25000000000000 -3.18750000000000
-3.12500000000000 -3.06250000000000 -3
-2.93750000000000-2.87500000000000 -2.81250000000000
-2.75000000000000 -2.68750000000000 -2.62500000000000
-2.56250000000000 -2.50000000000000 -2.43750000000000
-2.37500000000000 -2.31250000000000 -2.25000000000000
-2.18750000000000 -2.12500000000000 -2.06250000000000 -2
```

Bibliography

[1] A. Tarable, S. Benedetto, and G. Montorsi, "Mapping interleaving laws to parallel turbo and LDPC decoder architectures," *Information Theory, IEEE Transactions on*, vol. 50, no. 9, pp. 2002–2009, Sept. 2004.

[2] C. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 1, pp. 379–423, 1948.

[3] S.-Y. Chung, J. Forney, G.D., T. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 db of the shannon limit," *Communications Letters, IEEE*, vol. 5, no. 2, pp. 58–60, Feb 2001.

[4] R. Gallager, "Low-density parity-check codes," *Information Theory, IEEE Transactions on*, vol. 8, no. 1, pp. 21–28, Jan 1962.

[5] S. Johnson, "Introduction to ldpc codes," in *ACoRN Spring School on Coding, Multiple User Communications and Random Matrix Theory*, 2006.

[6] T. Richardson and R. Urbanke, "The capacity of low-density parity check codes under message-passing decoding," *IEEE Trans. Inform. Theory*, vol. 47, pp. 599–618, 2001.

[7] G. Masera, F. Quaglio, and F. Vacca, "Implementation of a flexible LDPC decoder," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 54, no. 6, pp. 542–546, June 2007.

[8] V. G. Chris Winstead, Nhan Nguyen and C. Schlegel, "Low-voltage cmos circuits for analog iterative decoders," *Circuits and Systems, IEEE Transactions on*, vol. 53, Apr. 2006.

[9] C. Kong and S. Chakrabartty, "Analog iterative ldpc decoder based on margin propagation," *Circuits and Systems, IEEE Transactions on*, vol. 54, pp. 1140 – 1114, Dec. 2007.

[10] S. Hemati and A. Banihashemi, "Dynamics and performance analysis of analog iterative decoding for low-density parity-check (ldpc) codes," *Communications, IEEE Transactions on*, vol. 54, pp. 61–70, Jan. 2006.

[11] A. Blanksby and C. Howland, "A 690-mw 1-gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *Solid-State Circuits, IEEE Journal of*, vol. 37, no. 3, pp. 404–412, Mar 2002.

[12] G. C. Luca Fanucci, Pasquale Ciao, "Design of a fully-parallel high-throughput decoder for turbo gallager codes," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, no. 7, pp. 1976–1986.

[13] L. Zhou, C. Wakayama, and C.-J. Shi, "Cascade: A standard supercell design methodology with congestion-driven placement for three-dimensional interconnect-heavy very large-scale integrated circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 7, pp. 1270–1282, July 2007.

[14] V. Nagarajan, S. Laendner, N. Jayakumar, O. Milenkovic, and S. P. Khatri, "High-throughput vlsi implementations of iterative decoders and related code construction problems," *J. VLSI Signal Process. Syst.*, vol. 49, no. 1, pp. 185–206, 2007.

[15] G. Masera, F. Quaglio, and F. Vacca, "Finite precision implementation of LDPC decoders," *Communications, IEE Proceedings-*, vol. 152, no. 6, pp. 1098–1102, 9 Dec. 2005.

[16] Jong-Yeol and H.-J. Ryu, "A 1-gb/s flexible ldpc decoder supporting multiple code rates and block lengths," *Consumer Electronics, IEEE Transactions on*, vol. 54, pp. 417–424, May 2008.

[17] S. Johnson and S. Weller, "Resolvable 2-designs for regular low-density parity-check codes," *Communications, IEEE Transactions on*, vol. 51, pp. 1413–1419, Sept. 2003.

[18] M. Mansour and N. Shanbhag, "High-throughput LDPC decoders," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 11, pp. 976–996, Dec. 2003.

[19] V. E. Benes, "Permutation groups, complexes and rearrangeable connecting network," *Bell System Technical Journal*, vol. 43, no. 4, pp. 1619–1640, 1964.

[20] A. Waksman, "A permutation network," *J. ACM*, vol. 15, no. 1, pp. 159–163, 1968.

[21] "METIS serial graph partitioning and fill reducing matrix ordering." ⟨http://glaros.dtc.umn.edu/gkhome/metis/metis/overview⟩.

[22] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, (New York, NY, USA), pp. 191–200, ACM, 1998.

[23] F. Quaglio, F. Vacca, and G. Masera, "Low complexity, flexible ldpc decoders," in *14th IST Mobile & Wireless Communications Summit*, 2005.

[24] "Digital Design and Synthesis Lecture 2 Slides, Concordia University Canada." ⟨http://users.encs.concordia.ca/∼asim/COEN_6501/Lecture_Notes/Lecture_Notes.htm⟩.

[25] "Implementation of a digital UART by VHDL." ⟨http://www.ee.ualberta.ca/∼elliott/ee552/studentAppNotes/1999f/UART/uart.html⟩.

[26] "Test Matrix." ⟨http://sigpromu.org/dhayes/480_240.txt⟩.