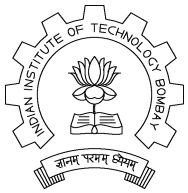

Department of Electrical Engineering, IIT Bombay

Internal Technical Report

FPGA Design for Decoder of Projective Geometry(PG)-based Low Density Parity Check(LDPC) Codes

Hrishikesh Sharma Sachin Patkar

July 29, 2010

Abstract

Explosive growth in Information Technology has produced the need of accurate ways of transmitting large amount of digital information over really long distances. Faster and efficient ways of encoding and decoding digital data to be transmitted across different kinds of channels have been developed over the years to maximize the utilization of the channel capacity and ensure error-free transmission. Low density parity check codes(LDPC), proposed by Robert Gallager in 1960, have been the subject of intense research and analysis over the past decade after their rediscovery in 1996. Since they possess the potential to perform very close to the Shannon limit, compared to any other code, the hindrances in the implementation of LDPC codes have been widely studied. High-throughput decoding of such codes has often been a challenge due to the large amount of resources required in fully parallel implementations and lower throughputs in serial implementations. Memory efficiency is a known obstacle in achieving high throughput. Many structures on their parity check matrices have been tried out to have efficient decoder designs.

In this report, we provide the detailed design of a fully-parallel LDPC decoder based on projective geometries, targetted for hardware emulation on Xilinx

Virtex-5 LX110T FPGA, and the corresponding platform XUP ML505. The design is based on memory-efficient communication primitives known as perfect access sequences. Simple switches are used to interface the nodes with a regular interconnect. Architectural concepts of speculative scheduling and microprogramming have also been used in the design.

1 Introduction

Rapid technological advances in the recent past have placed a recurrent demand on maximization of the speed and accuracy of modern communication systems. Error detection and correction in short as well as long range communication systems forms an integral part of the communication process. Efficient error correction schemes are, therefore, crucial to the functioning of the communication pipeline. Such schemes are especially important for maintaining data(information) integrity across noisy channels and less-than-reliable storage media. With the advent of wireless communication, where they channel noise is worse, importance of efficient error control has just increased. The simplest error correcting schemes involve the acknowledgement of error-free message reception, and retransmission of messages if required. However, *Forward Error Correction* Schemes, involving encoding the data with an error-correcting code (ECC) at the point of transmission and decoding the "most likely" data at the receiver end, are much more popular. Such schemes are used in used in computer data storage, for example CDs, DVDs and in dynamic RAM as well as in digital transmission, especially wireless communication.

LDPC codes are an emerging class of ECC codes which exhibit superior bit error rate (BER) performance, approaching the limit of Shannon capacity, over conventional channels. They require order of magnitude less arithmetic computations than equivalent Turbo decoders. Also, relative ease of decoder design coupled with better performance has made these codes candidates for usage in the recent generation of digital transmission and magnetic storage systems. These codes were proposed by Robert Gallager in his doctoral work at MIT, 1960 [6]. Due to the technological limitations in implementing them at that time, they were almost forgotten till they were rediscovered in 1995 by Mackay and Neal [14] and have been widely studied since then, especially due to their potential to perform close to the Shannon limit [27]. Their error correcting capability coupled with superior encoding techniques has brought about their acceptance as the new standard in a lot of digital transmission applications. As of now, LDPC are the standard of ECC in satellite transmission(DVB-S2) for digital television [4], 10GBase-T Ethernet (802.3an),

G.hn/G.9960(ITU-T Standard for networking over power lines, phone lines and coaxial cable) and WiMAX (IEEE 802.16e standard for microwave communications) [18].

LDPC codes are actually a class of linear block codes. Decoding of LDPC codes requires iterative methods, and is based on soft inputs from the channel. Many approximation algorithms have been designed for decoding LDPC codes[3]. All LDPC decoding algorithms require large number of parallel working memories, and hence designing for memory efficiency is one of the most challenging problems in its decoder design. In our work, we have focused on a hardware implementation of LDPC decoder, as part of a broader work of development of efficient methodologies for on-chip parallel computation. More specifically, in this work, we have designed for hardware scheduling of decoder that avoid memory bottlenecks arising out of conflicts, on-chip.

Unlike other codes, design of a LDPC decoder depends not only on code parameters(e.g., block length and code rate), but also on the type of the LDPC code. That is because the architecture of an LDPC decoder depends exclusively on the structure of the set of codes as represented by their parity check matrix $\mathbb{H}$. To achieve good throughput with reasonable decoder complexity, structured(in contrast to random) LDPC codes are used more often. Different code structures, in form of different H matrices, result in different architecture design, and hence different memory management schemes. Our choice of structures has been one whose parity-check matrix is derived out of geometry of projective spaces[4]. This choice of structure avoids memory conflicts, and also leads to lesser routing congestion.

Field Programmable Gate Arrays(FPGAs) are increasingly being seen as a promising avenue for High Performance Computing (HPC), especially with the introduction of high-end FPGAs such as Virtex-4/5 by Xilinx. These FPGAs are a very attractive choice for hardware prototyping due to their abundant local memory, embedded high-speed resources like DSP blocks, their reconfigurability etc. We selected FPGAs for prototyping our decoder architecture. The main challenge associated with FPGA-based design is to make the design as flexible in parameters as possible, small enough to meet the resource constraints with optimization in speed. A lot many researchers have reported prototype implementation results on FPGA for structured codes[6,7]. We present results for our first prototype based on fully parallel implementation of PG LDPC decoder. This implementation has helped as a stepping stone towards on-going implementation of semi-parallel design of the decoder. While a more complete picture of performance will emerge from the semi-parallel prototype, the objective of this implementation has been to demonstrate simplicity of design for PG-based LDPC codes, the high efficiency of the hardware due to the structure, apart from just BER performance. The implementation of our architecture has evolved by careful use of high-performing resources on modern

FPGAs. In this report, we present the design and implementation results for *fixed point logarithmic belief propagation(log-BP) decoding* for projective-geometry based codes, targeted at the *Virtex-5 LX330T* FPGA. We also present first time, the BER results for PG-based LDPC codes' decoding using log-BP algorithm.

The following sections are organized as follows. We first introduce LDPC Codes, and a complete derivation of a decoding algorithm for such codes. This is followed by explaining our choice of codes and their advantages. We also reproduce a parallel scheduling model to be used in decoder design [11]. With this background, we provide the design details. Especially, we give the details of the data path elements: processing units, memory blocks and interconnect, followed by the control path details in terms of microcode sequencing. We provide the overview of how we targetted such design for Xilinx FPGAs, followed by results based on synthesis and simulation. We conclude by noting down the future work possible.

2 LDPC Codes and Decoding

LDPC codes are capable of performing very close to the Shannon limit when decoded using probabilistic soft decision decoding process. In this decoding, knowledge of channel noise statistics is used to feed probabilistic information on received bits into the decoder. The reliability of bit information is successively improved over iterations, and the hard decision made post refinement. These iterative decoding schemes lead to *acceptable complexity* in decoder design.

2.1 Matrix Representation

LDPC codes are actually a class of **parity check codes**. Hence, they can be fully defined by a $\boldsymbol{m} \times \boldsymbol{n}$ parity check matrix $\mathbb{H}$, with an additional requirement that the matrix $\mathbb{H}$ be sparse. A valid codeword is said to be in the null space of the matrix $\mathbb{H}$, and hence satisfies the following.

$$\mathbf{Hc}^{\mathrm{T}} = 0 \tag{1}$$

All the vectors that satisfy the above equation are said to be valid codewords for the given code. The above product is equivalent to m *parity check constraints* involving any of the n bits of the vector. The rows of the $\mathbb{H}$ matrix thus represent the check constraints. Each of the columns of the matrix denotes a particular bit of the code and shows the

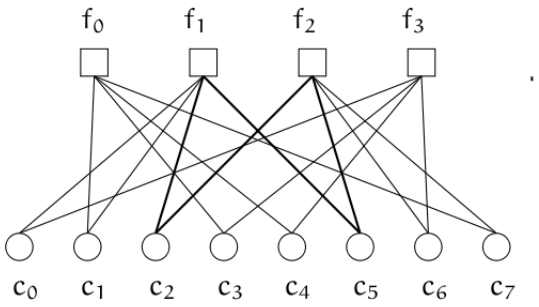

Figure 1: Tanner graph of a (2,4) regular code

constraints that it is involved in. If each of the code bits is present in the same number of check constraints, and each of the constraints involves the same number of bits, then the code is said to be a **regular** LDPC code(irregular otherwise). In general, the parity checks can be replaced by parity checks over *Galois fields* containing elements other than 0 and 1. Such class of codes are called polynomial codes.

2.2 Graphical Representation

The Tanner graph is a bipartite graphical representation of $\mathbb{H}$ having two sets of vertices: n bit nodes, and m parity-check nodes, for a $\boldsymbol{m} \times \boldsymbol{n}$ sized parity-check matrix. A bit node is connected to a check node if it is involved in the parity check denoted by the check node.So, an edge in the Tanner graph represents a non-zero entry in the matrix $\mathbb{H}$. In the Tanner graph of a regular code, all bit nodes have the same edge degree, and all check nodes have the same edge degree. A code is said to be (j,k) regular if each bit node is connected to j check nodes, and each check node is connected to k bit nodes. j and k are also called the column and the row degree of the matrix $\mathbb{H}$, respectively. An example Tanner graph of a (2,4) regular code is shown in Figure 1.

The Tanner graph representation is used to represent the exchange of information between the bit and the check nodes during the decoding algorithm. The exchange of data can be visualised to take place over the edges of the graph during each iteration of the decoder.

2.3 Derivation of Log Sum-Product Algorithm

The decoding process followed in this work is of *soft decoding* nature [9]. In the soft decoder, rather than flipping bits, probabilities are propagated through the Tanner graph, thereby accumulating evidence that the parity checks provide about the bits. The optimal(minimum probability of decoding error) decoder seeks a codeword $\hat{c}$ which maximizes $\mathrm{P}(\hat{c}|\mathbf{y},\mathrm{A}\hat{c} = 0)$, that is, the most probable vector which satisfies the parity checks, given set of received data $\mathbf{y} = [y_1, y_2, \cdots, y_N]$.

There exit optimal algorithms chasing this computation that provide the best *Frame Error Rate(FEC)* [7]. It is reached by the Viterbi Algorithm. Although block codes have a trellis representation, the complexity of their trellis increases exponentially with the size of the code. The decoding complexity for the true optimum decoding of an unstructured (i.e., random) code is *hence* exponential in K(for an (N,K,D) block code). Many suboptimal decoding algorithms(e.g., BCJR) have been proposed to work on reduced trellis size, but the complexity still remains high, especially when the codes have many parity checks involved. If one focusses on having optimal *Bit Error Rate*, then, using a simplification by MacKay [17], the decoder attempts to find a codeword having bits c_n which **maximize**

$$P(c_n|\mathbf{y}, \text{ all checks involving bit } c_n \text{ are satisfied})$$

That is, the *posterior probability* for a single bit given that only the checks on that bit are satisfied. As it turns out, even this easier, more computationally localized, task cannot be exactly accomplished due to approximations the practical algorithm must make. However, the decoding algorithm has excellent demonstrated performance and the complexity of the decoding is **linear** in the code length. Also, if the probability of symbol taking different values from its alphabet are all equal, then it can be shown in a straightforward way [7] that the abovementioned *Maximum a-posteriori(MAP)* decoding yields same results as *Maximum Likelihood(ML)* decoding. Hereafter, *for better readability and saving space*, we will use a **shortened notation**, $P(c_n|\mathbf{y})$, to imply $P(c_n|\mathbf{y}$, all checks involving bit c_n are satisfied).

A derivation of the decoding algorithm, using log-domain inputs, is distilled and captured here for understanding purposes, from [2], [7] and [17]. Computationally, this log-domain version avoids having to compute normalizations, which are present in the basic sum-product algorithm [17]. Also,

$$\hat{c}_n = 0 \text{ if } Pr(c_n = 0|\mathbf{y}) > Pr(c_n = 1|\mathbf{y}) \tag{2}$$

$$\hat{c}_n = 1 \text{ if } Pr(c_n = 0|\mathbf{y}) < Pr(c_n = 1|\mathbf{y}) \tag{3}$$

Hence if one considers log likelihood ratios as below, then using only the sign bit of LLR, one can estimate or decide the most probable value of c_n. The drawback of decoding based on this representation is that it is *only applicable to binary codes*. Luckily, the decoder we have designed deals with binary block codes only.

Let

$$\lambda\left(c_n|\mathbf{y}\right) \;=\; \log\,\frac{P(c_n=1|\mathbf{y})}{P(c_n=0|\mathbf{y})} \;=\; \log\,\frac{P(c_n=1|y_n,\{y_i:i\neq n\})}{P(c_n=0|y_n,\{y_i:i\neq n\})} \tag{4}$$

Using Baye's rule, the numerator can be re-written as

$$\begin{aligned} P(c_n=1|y_n,\{y_i:i\neq n\}) \;&=\; \frac{p(y_n,c_n=1,\{y_i:i\neq n\})}{p(y_n,\{y_i:i\neq n)\}} \\ &=\; \frac{p(y_n|c_n=1,\{y_i:i\neq n\})\cdot p(c_n=1,\{y_i:i\neq n\})}{p(y_n|\,\{y_i:i\neq n\})\cdot p(\{y_i:i\neq n\})} \end{aligned}$$

The noise that is *superimposed* on the transmitted codeword is in general assumed to be a **stochastic process**. If this sequence of (noise) random variables is also *independent, identically distributed(i.i.d.)*, then, given c_n, y_n is independent of $\{y_i : i \neq n\}$. Thence the expression above can be simplified as

$$\begin{aligned} &=\; \frac{p(y_n|c_n=1)\cdot p(c_n=1,\{y_i:i\neq n\})}{p(y_n|\,\{y_i:i\neq n\})\cdot p(\{y_i:i\neq n\})} \\ &=\; \frac{p(y_n|c_n=1)\cdot p(c_n=1|\,\{y_i:i\neq n\})}{p(y_n|\,\{y_i:i\neq n\})} \end{aligned}$$

The likelihood ratio in equation 4 can hence be written as

$$\underbrace{\lambda(c_n|\mathbf{y})}_{\text{Total Information}} \;=\; \underbrace{\log\,\frac{p(y_n|c_n=1)}{p(y_n|c_n=0)}}_{\text{Intrinsic Information}} \;+\underbrace{\log\,\frac{p(c_n=1|\,\{y_i:i\neq n\})}{p(c_n=0|\,\{y_i:i\neq n\})}}_{\text{Extrinsic Information}} \tag{5}$$

The total information represents the overall information of bit n. As pointed out earlier, the sign of total information leads to estimation of $\hat{c}_n$, while the magnitude of it represents the reliability of this estimate. Intrinsic information can be derived based on various channel models, and the formula are tabulated in [7]. A sample derivation of intrinsic information for AWGN channel model($= 4 \cdot y_n/\sigma^2$) can be found in [2]. It is obvious that the intrinsic term is determined by the explicit measurement y_n that affects the bit c_n. Similarly, it is obvious that the extrinsic term is determined by the information

provided by all the *other* observations, and the code structure. It essentially represents the **improvement** of information we *gain* by considering the fact that the coded symbols(bits) respect certain constraints. Hence one can try express the probabilities in the extrinsic term in terms of the parity checks, as in next section.

2.3.1 Belief Propagation on Cycle-free Tanner Graphs

Let $z_{m,n}$ denote the (partial) parity check computed using all the bits connected to the m^{th} (total) parity check associated with c_n, z_m, *except* for bit c_n. For setting up notation, let the set of bits that participate in check z_m be denoted as

$$\mathbb{N}_m \;=\; \{n:\; \mathbb{H}_{m,n} \;=\; 1\}$$

Also, let the set of bits that participate in check z_m *except* for bit n be denoted as

$$\mathbb{N}_{m,n} \;=\; \mathbb{N}_m \backslash n$$

Then,

$$z_{m,n} = \sum_{i \in \mathbb{N}_{m,n}} c_i \tag{6}$$

For further notation, let the set of checks in which bit c_n participates be denoted as

$$\mathbb{M}_n \;=\; \{m:\; \mathbb{H}_{m,n} \;=\; 1\}$$

Further, let

$$\mathbb{M}_{n,m} \;=\; \mathbb{M}_n \backslash m$$

be the set of checks in which bit c_n participates *except for* check m. The probability that $c_n = 1$ is the probability that the parity of **all the other bits** implied in **each** of the parity check in which c_n participates($\mathbb{M}_n$) is equal to 1 as well, so that the parity check equation could be satisfied(even parity). Hence if $c_n = 1$, then $z_{m,n} = 1$ for **all** the checks $m \in \mathbb{M}_n$. Similarly, if $c_n = 0$, then $z_{m,n} = 0$ for all $m \in \mathbb{M}_n$. Then, equation 5 can be rewritten as

$$\lambda(c_n|\mathbf{y}) \;=\; \text{Intrinsic Information} + \log \frac{P(z_{m,n} = 1 \text{ for all } m \in \mathbb{M}_n | \{y_i : i \neq n\})}{P(z_{m,n} = 0 \text{ for all } m \in \mathbb{M}_n | \{y_i : i \neq n\})}$$

On **Cycle-free Tanner graphs**, the events $z_{m,n} = 1$ for $m \;\in\; \{1, \cdots, |\mathbb{M}_n|\}$ are conditionally independent given $\{y_i :\; i \neq n\}$. This is because the set of bits associated

with $z_{m,n}$ will be independent of the bits associated with $z_{m',n}$, for $m' \neq m$, in cycle-free Tanner graphs, since the corresponding parity checks constraints m, n form **disjointed trees**. Hence the extrinsic information term of the above equation can again be rewritten as

$$\log \frac{\prod_{m\in\mathbb{M}_n} P(z_{m,n}=1|\{y_i : i\neq n)}{\prod_{m\in\mathbb{M}_n} P(z_{m,n}=0|\{y_i : i\neq n)} = \sum_{m\in\mathbb{M}_n} \log \frac{P(z_{m,n}=1|\{y_i : i\neq n)}{P(z_{m,n}=0|\{y_i : i\neq n)}$$

The individual probabilities in the numerator/denominator can be calculated using a theorem by Gallager [6], which also gives first hint towards possible involvement of recasting the above expression using hyperbolic functions.

- Consider a sequence of M independent binary digits a_i for which $\Pr(a_i = 1) = p_i$. Then the probability that $\{a_i\}_{i=1}^{M}$ contains an *even* number of 1's is $\frac{1}{2} + \frac{1}{2}\prod_{i=1}^{M}(1-2p_i)$.

Moving on, if we define the log likelihood ratio

$$\lambda(z_{m,n}|\{y_i : i\neq n\}) \;=\; \log \frac{P(z_{m,n}=1|\{y_i : i\neq n)}{P(z_{m,n}=0|\{y_i : i\neq n)}$$

Then, expanding using equation 6, where summation implies a binary XOR($\oplus$) operation, we have

$$\begin{aligned} \lambda(c_n|\mathbf{y}) &= \text{Intrinsic Information} + \textstyle\sum_{m\in\mathbf{M}_n} \lambda(z_{m,n}|\{y_i : i\neq n\}) \\ &= \text{Intrinsic Information} + \textstyle\sum_{m\in\mathbf{M}_n} \lambda\left(\sum_{j\in\mathbf{N}_{m,n}} c_j|\{y_i : i\neq n\}\right) \end{aligned} \tag{7}$$

Now, following the proof in [7], we have

$$\tanh\frac{1}{2}\log\frac{Pr(z_m=0|\bigcup D_j)}{Pr(z_m=1|\bigcup D_j)} \;=\; \prod_{j\in\mathbb{N}_m} \tanh\frac{1}{2}\log\frac{Pr(c_j=0|\bigcup D_j)}{Pr(c_j=1|\bigcup D_j)}$$

Where D_j represents the set of *other* bits on which calculation of c_j depends, i.e. $D_j =$ union of set of other bits used in parity check equations of set $\mathbb{M}_j$. These bits can be

seen as set of all the bit vertices *reachable* in the rooted tree depicted in figure 2, from the root c_j. By reversing the sign, we have,

$$-\tanh\frac{1}{2}\log\frac{Pr(z_m=1|\bigcup D_j)}{Pr(z_m=0|\bigcup D_j)} \;=\; \prod_{j\in\mathbb{N}_m}\tanh\left(-\frac{1}{2}\log\frac{Pr(c_j=1|\bigcup D_j)}{Pr(c_j=0|\bigcup D_j)}\right)$$

Due to **cycle freedom** of underlying graph, the dependent set of bits for each c_j are independent. Further, because LLR is a ratio, then by applying Baye's rule, D_j can be replaced by $\{y_i : i \in \mathbb{N}_{m,n}\}$(we are counting *other* bits) for ratio purposes. Hence

$$-\tanh\frac{1}{2}\log\frac{Pr(z_m=1|\{y_i:i\neq n\})}{Pr(z_m=0|\{y_i:i\neq n\})} \;=\; \prod_{j\in\mathbb{N}_m}\tanh\left(-\frac{1}{2}\log\frac{Pr(c_j=1|\{y_i:i\neq n\})}{Pr(c_j=0|\{y_i:i\neq n\})}\right)$$

Then, using the definition of λ above,

$$-\tanh\frac{1}{2}\left(\lambda(z_m)\right) \;=\; \prod_{j\in\mathbb{N}_m}\tanh\left(-\frac{1}{2}\lambda(c_j)\right) \tag{8}$$

Let S_j be the sign of $\lambda(c_j)$, which is a conditional LLR. Similarly, let M_j be the magnitude of $\lambda(c_j)$. Then,

$$\lambda(c_j) \;=\; S_j \times M_j$$

Using this in equation 8, and noting the fact that $\tanh(-x) \;=\; -\tanh(x)$, we have

$$\begin{aligned}
-\tanh\tfrac{1}{2}\left(\lambda(z_m)\right) &= \textstyle\prod_{j\in\mathbb{N}_m}\tanh\left(-\frac{1}{2}S_j\times M_j\right)\\
&= (-1)^{|\mathbb{N}_m|}\times\left(\textstyle\prod_{j\in\mathbb{N}_m}S_j\right)\times\prod_{j\in\mathbb{N}_m}\tanh\left(\tfrac{1}{2}M_j\right)\\
&= (-1)^{|\mathbb{N}_m|}\times\left(\textstyle\prod_{j\in\mathbb{N}_m}S_j\right)\times\log^{-1}\log\left(\prod_{j\in\mathbb{N}_m}\tanh\left(\tfrac{1}{2}M_j\right)\right)\\
&= (-1)^{|\mathbb{N}_m|}\times\left(\textstyle\prod_{j\in\mathbb{N}_m}S_j\right)\times\log^{-1}\sum_{j\in\mathbb{N}_m}\log\left(\tanh\left(\tfrac{1}{2}M_j\right)\right)
\end{aligned}$$

Or,

$$\lambda(z_m) \;=\; (-1)^{|\mathbb{N}_m|-1}\times\left(\prod_{j\in\mathbb{N}_m}S_j\right)\times 2\tanh^{-1}\log^{-1}\sum_{j\in\mathbb{N}_m}\log\left(\tanh\left(\frac{1}{2}M_j\right)\right)$$

Let us define a function $\phi(x) = -\log\tanh\frac{|x|}{2}$. It is easy to derive, using first principles, that $\phi^{-1}(x) = \phi(x)$. Then, the equation above can be relaid as

$$\begin{aligned}\lambda(z_m) &= (-1)^{|\mathbb{N}_m|-1} \times \left(\prod_{j\in\mathbb{N}_m} S_j\right) \times 2\tanh^{-1}\log^{-1}\left(-1 \times \sum_{j\in\mathbb{N}_m}\phi(M_j)\right) \\ &= (-1)^{|\mathbb{N}_m|-1} \times \left(\prod_{j\in\mathbb{N}_m} S_j\right) \times \phi^{-1}\left(\sum_{j\in\mathbb{N}_m}\phi(M_j)\right) \\ &= (-1)^{|\mathbb{N}_m|-1} \times \left(\prod_{j\in\mathbb{N}_m} S_j\right) \times \phi\left(\sum_{j\in\mathbb{N}_m}\phi(M_j)\right)\end{aligned} \tag{9}$$

Hence $\phi(x)$ represents a type of *input transformation* for the check nodes. Since originally in equation 7 we are interested in calculating $\lambda(z_{m,n})$, we have

$$\lambda(c_n|\mathbf{y}) = \text{Intrinsic Information} + \sum_{m\in\mathbb{M}_n}\left((-1)^{|\mathbb{N}_m|} \times \left(\prod_{j\in\mathbb{N}_{m,n}} S_j\right) \times \phi\left(\sum_{j\in\mathbb{N}_{m,n}}\phi(M_j)\right)\right) \tag{10}$$

There is a curious factor in the second additive term of above equation, $(-1)^{|\mathbb{N}_m|}$. If we were to define log-likelihood ratio as $\log\frac{Pr(c_i=0)}{Pr(c_i=1)}$ instead of $\log\frac{Pr(c_i=1)}{Pr(c_i=0)}$, then it is straightforward to see that this factor vanishes from second term. Whether one follows the former or the latter *convention* do design a decoder system is essentially still a system designer's choice.

2.3.2 Recursive Calculation

Each of the additive term within the second additive term in equation 10 can be thought of as the "message" which is passed from the **check** node m **to** the **bit** node n. However, it is also clear from the same equation that the computation of these check-to-bit messages requires the knowledge of $\lambda(c_j|\{y_i : i \neq n\})$, the conditional likelihoods of the bits which connect to the check equations of c_n. Calculation of these likelihoods is in fact the same problem as the computation of $\lambda(c_n|\mathbf{y})$, **but** by considering these bits c_j at the top of the **subtrees** resulting from the erasure of bit c_n. This **recursion** has to be processed until the *leaves* of the tree.

The terms $\lambda(c_j|\{y_i : i \neq n\})$ hence represent the "message" which is passed from the **bit** node j **to** the **check** node m. This "message" is the *collated* information that the bit node j gathers from various check nodes connected to it, *except* check node m, and passes it on to check node m. The term $\lambda(c_j|\{y_i : i \neq n\})$ **excludes** channel information that is related to bit n. This implies that information related to all the check equations

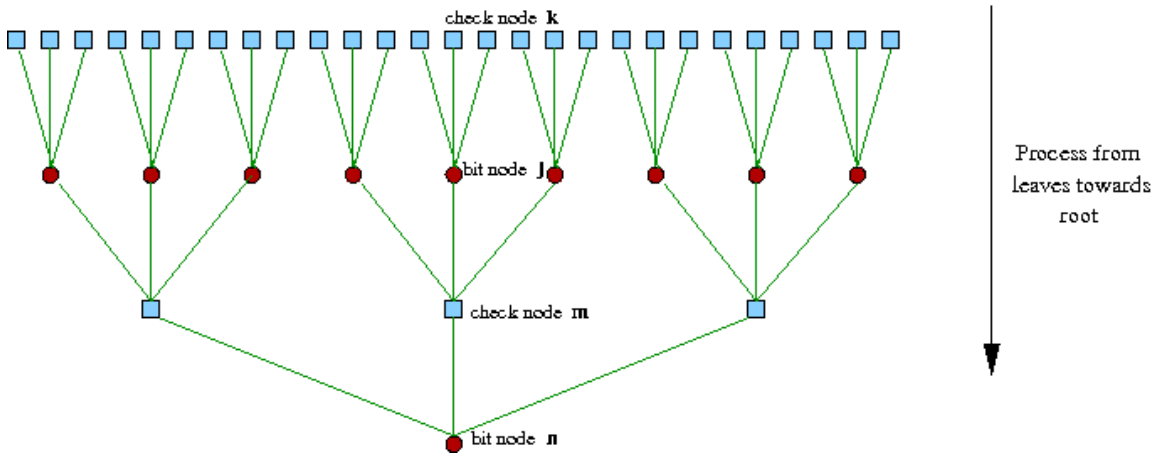

Figure 2: Recursive Calculation of Bit Likelihood Over a Tree

reachable by an edge in the tree rooted at bit node n(including check equation m) have to be excluded. In lack of cycles(especially 4-cycles), it is straightforward to see that between bit nodes j and n, no other check equation other than m is connected via an edge to both bit nodes in the tree diagram. Hence, correspondingly, information $\lambda(z_{m,n})$ from check node m is omitted from calculation of $\lambda(c_j|\{y_i:\ i \neq n\})$. Thus

$$\begin{aligned} \lambda(c_j|\{y_i:\ i \neq n\}) &= \lambda(c_j|\mathbf{y}) - \lambda(z_{m,n}) \\ &= (\text{Intrinsic Information})_j + \textstyle\sum_{k \in \mathbb{M}_{j,m}} \lambda(z_{k,j}) \end{aligned} \tag{11}$$

A figure depicting the recursive calculation of of $\lambda(c_n|\mathbf{y})$ is given in figure 2.

It is also imperative that the **leaf nodes** of the above tree has to be bit nodes. What is fundamentally available from channel observation is information about bits, and only that can drive the decoder subsystem. Further, at the leaf level, the extrinsic information accumulated from ficticious check nodes, that are actually absent, can be treated as 0 in equation 11.

The above recursive computation is also known as **Belief Propagation**. This is because each of the message, which denotes extrinsic information, represents a kind of *belief* that a node of some type can share about the possibility of value being taken by a node of other type(either bit value or the parity check value). Since such beliefs are made to propagate throughout the tree, the algorithm is indeed a belief-propagation algorithm(sometimes further called as *message-passing* algorithm).

2.3.3 Modification for Graphs with Cycles

For finite cycle-free graphs(trees), the log sum-product algorithm described so far is clearly **exact**. This algorithm is able to compute the maximum information for the root that can be obtained from all the observations. The algorithm is also **finite-time** on trees, given the depth of the tree.

In practice, we rarely get codes whose corresponding Tanner graphs can be **unrolled** into a tree alternating between row of bit nodes and row of check nodes. However, because all the operations of sum-product algorithm are local(as observed earlier), it may also be applied to graphs *with cycles*. The algorithm then becomes **iterative and approximate**.

Another issue with the above algorithm is that the node c_n, which was treated as the root of the tree, is not a distinguished root node, but is actually an *arbitrary node*. This issue can be dealt by considering each bit node c_n in turn as if it were the "root" of a tree. For each c_n, we consider each parity check z_m, $m \in \mathbb{N}_m$ associated with it, and compute check to bit messages $\lambda_{m,n}$ associated with it, and so on*ldots*. This variant of algorithm does not actually propagate information from leaf to root, but instead propagates information **throughout the graph** as if each node were the root. If there were no cycles in the tree, this algorithm would, in fact, *still* result in an exact computation at each node of the tree. But as bits connect to checks to other bits through the iterations, there must eventually be some cycles in the graph. These violate the independence assumptions and lead to only approximate results[2]. Simulations have shown that if length-4 cycles are avoided, then the results are still very impressive[20].

2.3.4 Inputs, Iterations and Output

The decoding algorithm derived above is clearly an *iterative process* of interchanging information between the two types of nodes on the bipartite Tanner graph. The pattern of interchange in an iteration is governed by the graph, which essentially depicts the parity check matrix $\mathbb{H}$. The fundamental input to the decoder system is the real-valued sequence of discrete samples at the reciever end of the channel. Using a particular noise model, these samples can be converted into intrinsic information samples, which are fed to the decoder. An example conversion has been covered in section 2.3.5. Every iteration, we try to estimate the value of bit, s_i, **at bit node**, from the total information available about the bit ($\lambda(c_n|\mathbf{y})$ using set of equations 2. These **decision equations**

can be expressed in terms of LLR as follows.

$$\hat{c}_n \;=\; 0 \Rightarrow \frac{Pr(c_n = 1|y)}{Pr(c_n = 0|y)} < 1 \Rightarrow \log \frac{Pr(c_n = 1|y)}{Pr(c_n = 0|y)} < 0 \tag{12}$$

$$\hat{c}_n \;=\; 1 \Rightarrow \frac{Pr(c_n = 1|y)}{Pr(c_n = 0|y)} > 1 \Rightarrow \log \frac{Pr(c_n = 1|y)}{Pr(c_n = 0|y)} > 0 \tag{13}$$

The iterative process is halted if after calculating the syndrome condition($\mathbb{H} \cdot \hat{\mathbf{s}}^T = 0$) over the estimated decoded vector d, at a given iteration, the resulting syndrome vector becomes the all-zero vector. If after several successive iterations the syndrome does not become the all-zero vector, the decoder is halted when it reaches a *given* predetermined number of iterations. In both cases, the decoder generates optimally decoded symbols or bits, in the a posteriori probability sense, but these will not form a code vector if the syndrome is not an all-zero vector. In this sense the sumproduct algorithm does not necessarily define the best estimate of the whole code vector that was initially transmitted through the channel.

The presence of cycles of relatively short lengths in the bipartite graph is virtually unavoidable when the corresponding LDPC code has good properties, but it is often possible to remove the shortest cycles(of length 4, 6, 8, etc.), or least reduce their number. The degrading effect of short-length cycles in the bipartite graph however diminishes as the code length increases and is strongly reduced if the code length is large (>1000 bits) [17].

2.3.5 Intrinsic Information for Gaussian Channels

In this section, we derive the intrinsic information value based on output of an Additive white Gaussian noise(AWGN) channel. AWGN is a channel model in which the *only* impairment to communication is a **linear** addition of wideband or white noise with a *constant spectral density* and a *Gaussian distribution* of amplitude. The model does not account for other channel issues such as fading, interference or dispersion etc. To understand AWGN model, we look at some definitions first.

Gaussian noise is statistical noise whose (amplitude) samples have a *probability density function(pdf)* of the *Gaussian distribution*. In other words, the values that the noise can take on are Gaussian-distributed.

The above definition says nothing of the *correlation* of the noise samples in time, or of the spectral density of the noise. Labeling Gaussian noise **additionally** as 'white' describes the correlation of the noise. In fact, white noise is another category of noise,

and hence white Gaussian noise is an *intersection category* of these noise models. **White noise** is essentially a statistical noise with a **flat** *power spectral density*. In other words, the signal contains equal power within a fixed bandwidth at any center frequency.

Gaussian noise is **not necessarily** always a white noise, since the definition of Gaussian noise does not put any constraint on correlation of various amplitude samples, which are Gaussian-distributed. Correlation between such samples in time or frequency domain leads to distribution of signal power spectrum.

Hence one can define the intersection category of **white Gaussian noise** as a random process N(t) which satisfies the following **additional constraints**.

1. $\mu_N = E[N(t)] = 0$ (mean, expected value)
2. For any time instants $t_1 < t_2 < \ldots < t_k$, $N(t_1),\ N(t_2),\ \ldots,\ N(t_k)$ are **independent** gaussian random variables.

If we take a sequence of i.i.d. Gaussian random variables with a probability density function N(0, σ^2), then also the above two constraints are met. Treating these Gaussian random variables as individual amplitude distribution of a white Gaussian noise process, it is easy to prove that the power spectral density N_0 is $\sigma^2/2$.

The **additive** white Gaussian noise channel can now be defined/represented by a series of outputs Y_i at discrete time event index i. Y_i is the sum of the input X_i and noise, N_i, where N_i is *independent and identically-distributed* and drawn from a *zero-mean* Gaussian distribution with variance σ^2. The N_i are further assumed to not be correlated with the X_i.

$$N_i \sim N(0, n)$$

$$Y_i = X_i + N_i$$

Let us assume that the modulation scheme been adopted post encoding of c_n on the transmitting end is Binary Phase Shift-keying(BPSK). In this scheme, each bit is mapped to amplitude level of either +A or -A. Since we use binary codes, each symbol of the codeword consists of only one bit, and hence simple schemes such as BPSK are applicable clearly. Let us further assume that the mapping of {0, 1} to {+A, -A} is **binary antipodal**, that is, {$0 \mapsto$ +A, $1 \mapsto$ -A}. Such a *choice* is clearly a *matter of convention*. Let the post-modulated analog signal be denoted by sequence t_n, where the index n essentially denotes a time period of the pulse on which the bit was modulated. Let us also assume that the communication channel has N($(\mu =)0$, σ^2) AWGN model, which

has a *probability density function* of

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{14}$$

Due to additive property of noise, if it is *superimposed* upon a signal having a constant amplitude of +A, then the new signal becomes yet another stochastic process with same variance, but with a mean shifted from 0 to +A. Also, when applied to a situation where $t_n = +\text{A}$, the probability density function of the this output signal clearly denotes the *posterior* probability, $p(y_n|t_n = +A)$. Thus

$$\begin{aligned} p(y_n|t_n = +A) &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_n - t_n)^2}{2\sigma^2}} \Big|_{t_n=+A} \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_n - A)^2}{2\sigma^2}} \end{aligned} \tag{15}$$

Using the convention of LLR, $\log \frac{p(...=1)}{p(...=0)}$, we have,

$$\begin{aligned} \text{Intrinsic Information} &= \frac{p(y_n|c_n=1)}{p(y_n|c_n=0)} \text{ , from equation 5} \\ &= \frac{p(y_n|t_n=-A)}{p(y_n|t_n=+A)} \\ &= \frac{e^{-\frac{(y_n+A)^2}{2\sigma^2}}}{e^{-\frac{(y_n-A)^2}{2\sigma^2}}} \\ &= -\frac{2 \cdot A \cdot y_n}{\sigma^2} \text{ , upon simplification} \end{aligned} \tag{16}$$

Generally A is *assumed* to be 1(normalized signal level) so that the encoded signal's power also turns out to be 1(normalized).

The above derivation makes use of two conventions: one for the definition of LLR, and one for the mapping of symbols to amplitude levels. Based on variations on these, different formulaes for intrinsic information can be derived using first principles, as in this section.

2.3.6 Impact of Conventions on Output Decision

So far we have seen two conventions in the algorithm. One convention pertains to whether LLR is represented as $\log \frac{(p=1\cdots)}{(p=0\cdots)}$, or whether as $\log \frac{(p=0\cdots)}{(p=1\cdots)}$. The other convention relates to whether the BPSK modulation is assumed to be binary antipodal($t_i = A \cdot (1 - 2 \cdot c_i)$) mapping, or binary podal($t_i = A \cdot (2 \cdot c_i - 1)$) mapping.

The impact of first convention is **change of sign of every data value, at every step** of decoding process, including the decision making process. The intrinsic information becomes $\frac{2 \cdot A \cdot y_n}{\sigma^2}$, hence opposite-signed data flows into the variable nodes, which in turn emit opposite-signed bit-to-check messages. The impact of LLR convention then also gets reflected in check node computation(refer equation 10) in terms of the factor $(-1)^{\mathbb{N}_m}$. Finally, at the time of decision making based on sign of **total LLR** in bit node, an opposite LLR condition, from equation 12, leads to making of right bit decision based on **opposite** comparison with 0.

The impact of following a particular convention with respect to BPSK modulation is slightly more subtle. If we change the BPSK mapping from e.g. binary antipodal to binary podal mapping, then the intrinsic information also changes sign(e.g. to $\frac{2 \cdot A \cdot y_n}{\sigma^2}$). However, given that for the same c_n, t_n now has an opposite sign, any *average case* noise superimposition will lead the samples y_n themselves to have opposite sign. Hence the effect of 2 multiplicative (-1) terms in the intrinsic information formula tends to cancel out even before it enters the iterative decoding process.

Hence, it does not matter what convention one follows, till one knows where all to make changes in computation, thus ensuring that the output will still be consistent.

2.3.7 Summary of Decoding Algorithm

Input (a) The Parity-check Matrix $\mathbb{H}$, (b) The maximum number of iterations(stopping criterion) **L**, and (c) the sequence of intrinsic inforamtions from channel samples, derived from equation 16.

Initialization (a) $\lambda^{[0]}(z_{m,n}|\{y_i : i \neq n\}) = 0$ for all (m, n) with $\mathbb{H}_{m,n} = 1$. (b) $\lambda_n^{[0]} =$ Intrinsic Information as per the channel model used. (c) Loop counter $l = 1$.

Check Node Update For each (m, n) with $\mathbb{H}_{m,n} = 1$, compute check(m)-to-bit(n) update message($\eta_{m,n}^{[l]}$) as

$$\left((-1)^{|\mathbb{N}_m|} \times \left(\prod_{j \in \mathbb{N}_{m,n}} S_j^{[l-1]} \right) \times \phi \left(\sum_{j \in \mathbb{N}_{m,n}} \phi(M_j^{[l-1]}) \right) \right)$$

Bit Node Update For each bit n $= 1, 2, \ldots, N$, compute using equation 10, via messages from check nodes m: $\mathbb{H}_{m,n} = 1$, as

$$\lambda^{[l]}(c_n|\mathbf{y}) \; = \; \text{Intrinsic Information} + \sum_{m \in \mathbb{M}_n} \eta_{m,n}^{[l]}$$

Decision Making on Syndrome Vector At each bit node, after computing total information $\lambda^{[l]}(c_n|\mathbf{y})$ every iteration, set s_n to 0 if $\lambda^{[l]}(c_n|\mathbf{y}) < 0$, or to 1 otherwise. If $H \cdot \mathbf{s}^T = 0$, then **Stop** decoding.
Otherwise, if l(number of iterations) is $< L$, the loop back to Check Node Update. Otherwise, declare a decoding failure and **Stop**.

2.3.8 Cycles, Convergence and Update Scheduling

The scheduling of the BP algorithm is the order in which the messages of the graph should be propagated. In the case when the Tanner graph has no cycles, this scheduling **does not** affect the convergence of the algorithm. However, presence of cycles does delay the convergence. Starting off from some node, extrinsic information can follow a cycle and **come back as redundant information** to the same node, being *part* of some message on the last step of the cycle. Back circulation of redundant information leads to delay in convergence. As an aside, presence of *short cycles* delays the convergence more, than presence of longer cycles.

The basic scheduling algorithm for sum-product, or belief propagation, is known as the *Flooding Schedule*. In this scheduling, nodes of one side all do their updates in parallel, after which nodes on the other side also do their updates in parallel. These two sets of updates put together form an iteration. This scheduling is what was hinted at in section 2.3.3.

It can be seen by unfolding a Tanner graph, that there **also** exist paths/cycles that are longer than the girth of the Tanner graph. Scheduling the message-based *alternating* bit and check node update along these paths leads to improvement in convergence performance. A class of decoding schedules, known as *Shuffle Schedules*, have been derived on this observation [7].

Banihashemi and others have studied various ways of deriving schedules based on graph concepts such as closed walks [22]. Another effort on deriving better schedule can be found in [19]. In general, most of the scheduling techniques, and a single, broad framework for describing these can be found in [7].

3 Projective Geometry and LDPC Codes

In this section, we present details of LDPC codes derived out of projective geometries, that we use for our decoder. In the next section, we try to justify why we choose

projective geometry based LDPC codes.

3.1 Code Construction from Projective Geometries

Two classes of LDPC codes (type-I and type-II) can be constructed based on the lines and planes of projective geometries over finite fields[12]. They have good minimum distance properties and they can be decoded in various ways. They can also be put in quasi-cyclic or cyclic form. This makes the hardware of the decoder easily practicable using shift registers as well as their encoding easier using the linear-feedback shift register (LFSR) technique.

Let $\mathbb{G}$ be a finite geometry with **n** points and **j** lines, and the properties of incidence on lines and points.

The $j \times n$ matrix $\mathbb{H} = [h_{i,j}]$ for such codes is constructed as follows.

- Rows and columns correspond to the lines and points of the geometry respectively
- $h_{i,j} = 1$ iff the ith line contains the jth point in $\mathbb{G}$ and vice versa.

This in turn implies that,

- Incidence properties of $\mathbb{G}$ ensure that the resulting code is *regular*, with weights as defined by the structure of the finite geometry.
- Any two columns have exactly one row where there is a '1' in both columns. This corresponds to the fact that any two points lie on exactly one line.
- Similarly, any two rows have **at most** one column where there is a '1' in both rows. This depends on whether the two corresponding lines are parallel, or they intersect.

A matrix $\mathbb{H}$ derived from such a $\mathbb{G}$, and its transpose, form two types of PG LDPC codes, called the *type I* and *type II* codes respectively. Both the codes hence have different codeword length(**n** and **j** respectively).

In decoding a linear block code such as LDPC with the *Sum Product Algorithm* and its variants, the performance very much **depends on cycles** of short lengths in its Tanner graph. A **cycle** in a (simple) graph is defined as a *sequence of connected edges*, which starts from a vertex and ends at the same vertex. The sequence further satisfies the

condition that no vertex, except the initial and the final vertex, appears in the sequence more than once. If there are short cycles, especially cycles of length 4, they make successive decoding iterations highly correlated[2]. Hence the decoding performance gets severely limited. It can be shown that the Tanner graph of type-I and type-II **PG codes don't contain cycles of length 4**. But they do contain cycles of length 6 [12]. Hence their decoding is expected to be better than that of codes having girth-4 cycles in their Tanner graphs.

Following [12], if the rows and columns of the matrix are derived from a general m-dimensional projective geometry, $\mathbf{PG}(\mathrm{m},2^s)$, over a $\mathrm{GF}(2^s)$, the rows and column weights will then be

$$w_r = 2^s + 1 \tag{17}$$

$$w_c = \left(\frac{2^{ms}-1}{2^s-1}\right) \tag{18}$$

The *null space* of the matrix $\mathbb{H}$ defines a codeword ensemble of length $\mathbf{j}$, given a $\mathbf{j}\times\mathbf{n}$ parity check matrix $\mathbf{H}$. The minimum distance properties of these codes provide them a superior coding gain. Techniques like column and row splitting can be applied to improve the code rate. It can be shown that one-step majority logic decoding can correct $\left[\frac{w_c}{2}\right]$ or fewer errors in a code vector of a type-I PG LDPC code. Hence the *minimum distance* of the code is **at least** $w_c + 1$ [12].

For our analysis, we shall only consider codes based on lines and points of a 2-d projective geometry, called the **projective plane**. Some of the characteristics of PG LDPC codes based on projective plane, that need to be considered for decoding are as follows.

- The Tanner graph of the code is symmetric with respect to the degree of nodes of any one kind, due to the duality of subspaces in the 2-d geometry.
- From the set of edges connected to one of the nodes in the graph, the set of edges connected to any other node on the same side of bipartite graph can be found out using *Shift Automorphisms*[11]. Hence the Tanner graph of the code is constructible from incidences of just one node on one side of partite graph.
- Since the geometry is derived from a Galois field on prime power p^s, the parameters of the code and hence the design constraints also depend on the prime power p^s.

For a LDPC code derived from $\mathrm{PG}(2,p^s)$, the important design parameters are the following.

- Code length = **n** = no.of points = p^{2s}+p^s+1
- Number of parity check equations = **j** = no.of lines = p^{2s}+p^s+1
- Row weight = Column weight = p^s+1

Then, for a **fully parallel** decoder for such a LDPC code, we have

- No. of bit processing nodes = p^{2s}+p^s+1
- Number of check processing nodes = **j** = no.of lines = p^{2s}+p^s+1
- Interconnect degree per node = p^s+1

3.2 LDPC codes based on PG(2,2^3)

The projective plane over **GF**(8) contains 73 points and 73 lines. The $\mathbb{H}$ matrix is formed by the incidence relationships between the lines and the points of the projective space, as usual.

Recall that codewords of LDPC codes over **GF**($\mathbf{2^s}$) contain symbols from the field **GF**(2): {0,1}. In general, a class of codes called *polynomial codes* over **GF**(**p**), where **p** is any prime number, can contain symbols from **GF**(**p**): {0,1,2,. . .,p-1}. The constraints for such codes are then defined over *modulo* ***p*** arithmetic [13]. For example, the value of a constraint may be given by ($\sum_i v_i y_i$) (mod **p**), where coefficients v_i's are the coefficients from the $\mathbb{H}$ matrix, and y_i's are the symbols of the input vector. For such generalized constraints, however, the $\mathbb{H}$ matrix must also contain symbols from the base field GF(p).

Some of the parameters of the code are enumerated below.

- No. of bits = $(2^3)^2$+ 2^3+ 1 = 73
- No of parity checks = $(2^3)^2$+ 2^3+ 1 = 73
- No. of bits per parity check = 2^3+ 1 = 9
- No. of parity check per bits = 2^3+ 1 = 9
- Rate of the code = $\frac{45}{73}$

The rate can be found by the formula given in [13]. For codes over PG(m+1,2^s), no. of parity check symbols in a codeword is

$$n - k \;=\; 1 + \binom{m+p-2}{m-1}^{s} \tag{19}$$

For the code under consideration, m=3, p=2 and s=3. Hence the number of parity symbols in a block of 73 symbols is 28. Hence the rate of the code = $\frac{73-28}{73} = \frac{45}{73} \approx 0.6$.

Codes of such length may be useful in magnetic recording channels where layered coding is used[8]. This involves using an outer code on a block which is already coded using an inner code. One or both of these codes may belong to the LDPC scheme. Trellis and RS codes are also popular in these applications.

In the remaining report, the design of the decoder for such codes is explained over. In the next subsection, we introduce Karmarkar's idea of *perfect access patterns*. We further see its adaptability to decoding of LDPC codes using the log sum-product algorithm. Then, we propose a design which makes use of the ideas entailed so far, to decode the PG ($2,2^3$) LDPC code.

4 Choice of Codes

LDPC decoder design has three main parts: *node unit design, interconnection network design and the memory management design*. Memory management, to some extent, is tied up with interconnect design, because the node units use the interconnect network only to access the message memory. It is the network that primarily determines system performance in hardware system design [28]. Updating of extrinsic information is where most of the time is spent by the decoder. Routing for direct wiring of the interconnect network based on a random general Tanner graph leads to routing congestion due to the disorganized nature of the defining graph. Such Tanner graphs also complicate the design of memory controllers due to unstructured addressing pattern. To reduce the complexity, some structure in the codes/graphs is needed. Our choice of Tanner graph hence has been a regular Tanner graph. Designing regular LDPC codes with structural properties such as girth is a combinatorial design problem. A nice work is by Shu Lin et al[12], who present a geometric approach to the design of LDPC codes based on the lines and points of various geometries over finite fields. They obtain four classes of cyclic LDPC codes with girth 6. For long codes, BER performance of these codes is very close to the Shannon limit. We have chosen type-I PG codes from their work. The required regularity is visible from making a Tanner graph by carving out two subspaces and their

interconnections from a projective geometry lattice. As far as we know, our work is its first extension towards efficient decoder design.

In general in the parity check matrix made out of subsumption relation between two projective subspaces, not all rows representing parity check equations are linearly independent. However, dropping such rows leads to slightly degraded performance[12], and hence there are generally few more check nodes in PG-based LDPC decoder than strictly required. Most of these codes have a rate of around 1/2, which can be boosted by a novel technique called column splitting. A high-level design for column-split decoder has also been worked out[21].

5 Parallel Scheduling Model

The scheduling model used in our design is based on Karmarkar's template [11]. Projective geometry lattices possess structural regularity, and this property has been exploited in scheduling of general parallel systems. Karmarkar was able to come up with a method to achieve various nice properties that one seeks during parallel system design. As an example, a 2-dimensional **PG** of order **s** contains $\mathbf{n = s^2 + s + 1}$ points and as many lines. Each line is connected to $\mathbf{s + 1}$ points and vice-versa. Given **n** processing units and a memory system partitioned into n memory blocks M_1, M_2, $\cdots$, M_n, Karmarkar's template can be applied by mapping memory blocks to points, and processing units to the lines. A memory block and a processing unit are connected if the corresponding point belongs to the corresponding line. Then any binary or multi-ary operation can be assigned to each processing unit such that fetch of operators is based on **perfect access patterns**(PAP) and **sequences**(PAS), as defined in [11]. A schedule for such a *collection of operations* leads to several important advantages, such as no memory conflict, utilization of all processing units, full utilization of memory bandwidth, load-balancing etc.

The perfect access patterns for a projective plane can be identified as follows. For projective plane, shift automorphism can be applied transitively on both lines and points. Hence one can use powers of it to "rotate among the points and lines such that each point/line along the "rotation path is visited exactly once. Let **l** denote a line generated by two points **a** and **b**. Then the perfect access *pattern* is defined by collection of lines made out of shift automorphism of these points. The perfect access *sequence* is a collection of such patterns, arising out of different pairs of points **a** and **b**, which are part of same line **l**. A perfect access pattern is depicted in figure 3, while a perfect access sequence is depicted in figure 4.

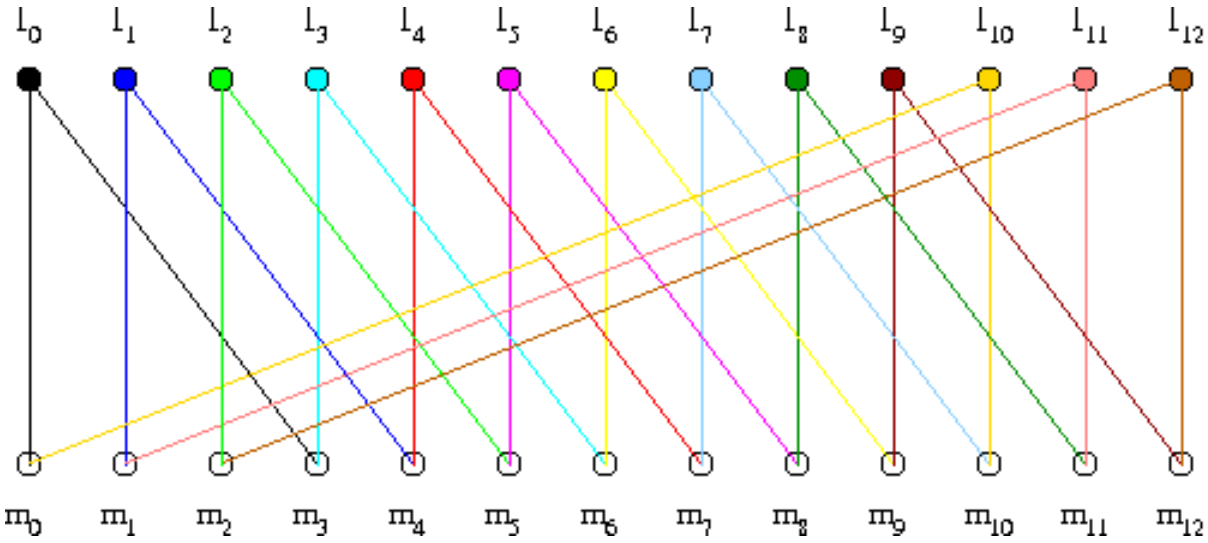

Figure 3: Example Perfect Access Pattern in Order-13 dimension-2 PG Graph

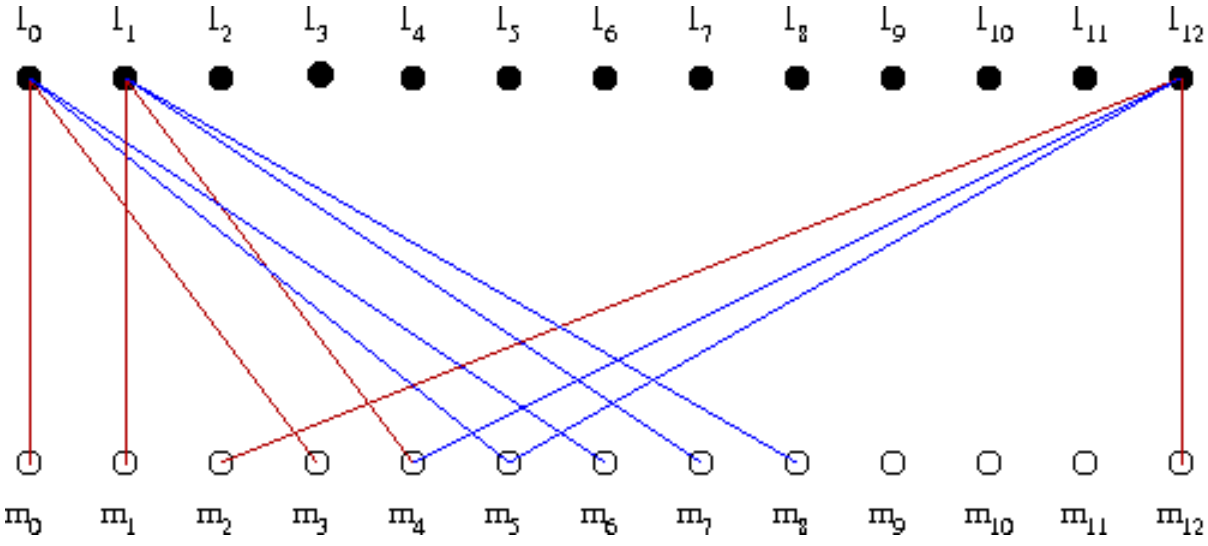

Figure 4: Example Perfect Access Sequence in Order-13 dimension-2 PG Graph

One can argue that the *behavior of regular PG-based LDPC decoder is made up of perfect access patterns and sequences* [21]. Once that is established, applicability and utility of Karmarkar's template is obvious. The two types of computation, done by bit nodes and check nodes, remain same over iterations. For each computation, the input messages(operands) have different values, and hence need to be stored in different memory blocks. Thus it is imperative that we first split the LDPC decoding into two computations having topology similar to that Karmarkar's template envisages. Fortunately, the bit nodes and check nodes work in *non-temporal-overlap* fashion. We do the splitting by designing in such a way that the memory block they access for operand input and output are non-common. Thus the problem of scheduling for PAP/PAS in LDPC decoding is decomposed into **two isomorphic scheduling problems**; see figure 5. Hence we map both the sub-computations to the same fixed projective plane. In the bit-node processing, the major processing involves adding the extrinsic information collected over all the connections of the particular bit-node to check nodes. This processing is same in nature on all bit-nodes. The number, and size of input coefficients that each bit-node deals with, is also fixed. By taking two inputs at a time for addition, we can schedule a binary operation on each bit-node processing unit, in every machine cycle. The set of concurrent operations in each cycle form a PAP, while the set of all operations within complete bit-node processing form the PAS. The processing is similar in check nodes, though in log(tanh()) domain, hence one can again find perfect access pattern in check-node processing as well. It can thus be observed that by the time a perfect sequence execution is over, (major) processing (part) required on a node is also over. Hence the PG-based LDPC code decoding algorithm does exhibit behavior which has a decomposition based on perfect access patterns and sequences. All that remains, then, is to deal with its refinement for detail design purposes.

For a more detailed picture of this mapping and applicability of Karmarkar's template, one can refer to [21].

6 Overall Hardware Model

A hardware implementation of the LDPC decoder has a number of components, each with its own set of design challenges and architecture options. An intuitive look at the algorithm suggests the following components.

- *Sequential logic design* for performing the bit node and the check node updates

- Interconnect design using direct wires and memory blocks, to realise the edges of the Tanner graph i.e. the connectivity between nodes
- Finite precision representation of the data elements, i.e. messages
- Memory access/storage for the data dependency between iterations
- Fast and efficient I/O for block decoding of the codes

Some of the salient features of the decoding algorithm, which have a direct consequence on the mapping of the algorithm to hardware, are as follows.

- In both the bit and check node updates, there is *no data dependency* between nodes of the same type. So, all bit node processing/check node processing can happen in parallel.
- The messages passed over the edges are carried forward from one iteration to the next one. Hence they have to be stored/accessed from some local/global memory.

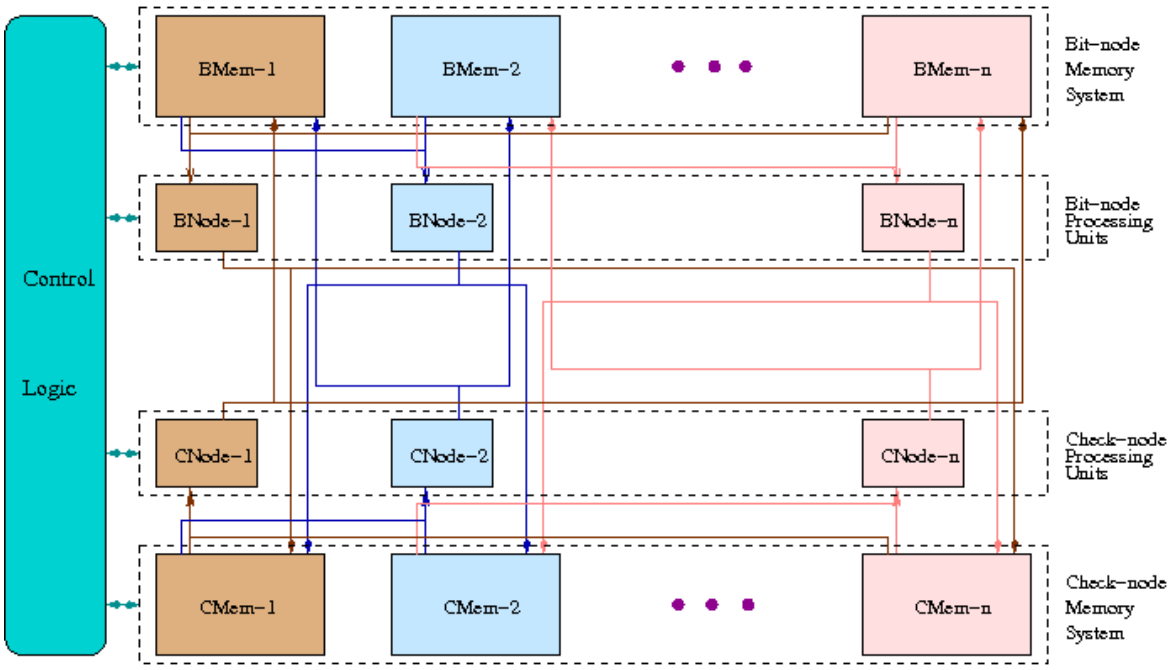

Figure 5: Symmetrical Decomposition of LDPC Decoding Computation

- The datapaths of the bit node and the check node processing units performing the bit node and the check node updates may contain complex functions that have to be quantized using *lookup tables* (LUT's).The precision involved in such quantization also affects the code performance of the entire system[10].

- The control path of the decoder does the scheduling of the node computations, and the address generation for data fetching/storing. Following section 2.3.8, *flooding schedules* have the highest degree of parallelism, where every node computation happens in parallel with nodes of the same type.

- Regular LDPC codes have the leverage offered by the **SIMD** architectural models, since data path of every check node is same, as well as the data path of bit nodes. Further, at a time, if flooding schedule is used, either bit nodes, or check nodes are doing the computation. Hence the difference remains only in different input data on which these nodes operate upon.

Interconnects have long proven the roadblock in VLSI implementations of LDPC decoders at large code lengths. In order that a VLSI LDPC decoder be effective, it should be able to meet following requirements to the maximal extent possible.

Conflict-free Memory Access LDPC decoding being communication-intensive(iterative message-passign algorithm)", memory accessing needs to be primarily optimized. Depending on the degree of the Tanner graph of the code, multiple processing elements will require data from a particular memory block during their execution. The memory sub-system hence should be free to maximum extent from various interconnect congestions, and in the best case, needs to be able to handle various concurrent memory access requests in a conflict free way.

Scalable There should be no major modifications are required for it to be able to decode codes of higher lengths.

Centralized Control logic Since the model of decoding computation is SIMD under flooding schedule, it is possible, within an iteration, to replicate/centralize the control path. This further reduces the logic density of the decoder circuit. This will however increase the length of wires to be routed across modules, and may require buffering along the path to offset the parasitics-driven wire delays.

Now we discuss the structure of a decoder design for PG LDPC codes, which tends to satisfy the above requirements.

6.1 Data Path Overview

Most of the computational logic of the design is contained in the node processing units. As discussed earlier, there are two types of nodes and memories in the decoder. The bit nodes read input messages from the check memories, write back to bit memories, and vice-versa. This completes an iteration of decoding, after which the complete datapath is traced back in next iteration, till the finish. The memory blocks and processing units are connected according to the geometry of the order-8, dimension-2 projective space. All components along the datapath are **synchronous, sequential logic elements**. The components are synchronous, since that eases the controlling based on microcode sequencing, detailed in next section. Each component is further controlled by its own enable signal, so that power wastage could be avoided. For speed and resource efficiency, we chose to implement the data path using **9-bit fixed-point arithmetic**, which has 1 sign, 3 integer and 5 fraction bits. It was found *sufficient* to represent LLRs for BER measurements that we planned. *Coincidentally*, it also helped in implementing memory blocks using block RAMs on Virtex 5 FPGA, which can be naturally configured having 9-bit ports. To avoid overflow during accumulations, the internal data path was made 13-bit wide in bit nodes, and 12-bits wide in check nodes. Since magnitude processing happens separately in check nodes than sign processing, we need a bit less in width of the magnitude data-subpath of check nodes. Bit nodes follow 2's complement arithmetic during their computation, while check nodes follow sign-magnitude arithmetic during their computation. For conflict-free memory accesses along the datapath, we choose to employ the perfect access sequences for governing the flow of messages. Memory blocks are designed using hard macros, such as BRAMs in FPGA implementation that we targetted, and are also assumed to be synchronous access storages. The time of flight of signal over the interconnect is assumed to be negligible enough to avoid wire pipelining in ASIC design.

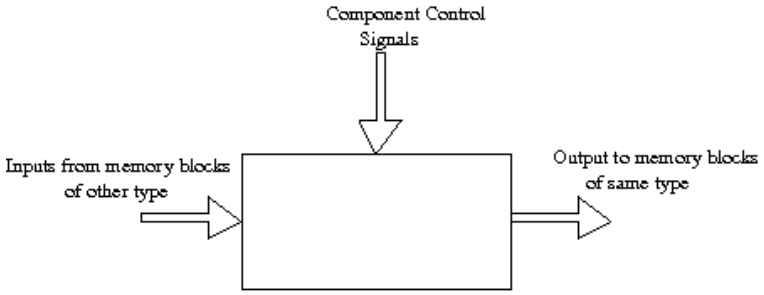

Figure 6: Node Processing Unit Interface

The bit (check) side units represent the points (lines) of the geometry, and correspondingly the columns (rows) of the parity check matrix $\mathbb{H}$. Every node processing unit has two inputs and two outputs, along with the control signals to enable various internal components. Binary perfect access sequences are obeyed during each read or write of data at the computation boundaries. The black box representation of each node processing unit is shown in figure 6.

6.2 Control Path Overview

To have a scalable control path, we chose to use the popular **microcode sequencing** template for control path design. Accordingly, all data path elements are synchronous sequential logic elements with their own enable signals. Such a template also helps from the requirement of being able to exploit the SIMD nature of computation by means of having a centralized control unit.

While implementing the control store, we do not need extra space in the table to handle branchings. In fact, we only have one branching at the end of the decoding, and that can be handled out of microcode-based sequencing as an exception case.

A disadvantage of using such a control path architecture is that the **implicit** pipeline stages are **unbalanced** in general. So the width of each clock pulse need not be **maximally utilized** to cover the propagation delay of various computing components. So while one component, when enabled, may take $T/2$ time to finish computing, then next component in datapath may take T time to finish its computation, starting from next clock edge. In this case, due to synchronous design, there is $T/2$ time, during which system is doing nothing, but which cannot be avoided. Unless one is careful during structural decomposition of the design to have similar-depth circuits(in terms of primitive gates), an asynchronous design is bound to give better performance over a microcode sequenced synchronous design.

7 Bit Node Architecture

Every bit processing unit has to perform the following update on its data(see equation 7).

$$\lambda(c_n|\mathbf{y}) \;=\; \text{Intrinsic Information} + \sum_{m \in \mathbf{M}_n} \lambda(z_{m,n}|\{y_i : i \neq n\}) \tag{20}$$

Hence each bit processing unit reads following two types of data from check memory blocks in appropriate cycles.

- Intrinsic information, which is best latched at the beginning of decoding iterations, and
- Extrinsic probabilities or check-to-bit messages $\lambda(z_{m,n}|\{y_i : i \neq n\})$ from check memory blocks

The value $\lambda(c_n|\mathbf{y})$ is essentially the *residue* for a check node m. It involves the sum of the intrinsic information, and all the incoming check-to-bit messages **except** the one from the check node m. There are *multiple options* for calculating the residue.

Direct For *every edge* incident on the node, the bit node updates can be computed *independently*. This would require independent datapaths for all the computations in a single bit node, thus requiring a huge amount of gates. Hence is suitable only for a small-sized Tanner graphs.

Trellis A forward-backward processing is performed to calculate the output messages, just like a systolic Trellis graph method. Such a method has been reported in [3], albeit for check nodes.

Total sum-first If the primary node operator(XOR in the case of check nodes, addition for bit nodes) is invertible(XOR and subtraction respectively), then the computation can be done as follows. All the messages from the incident edges can be combined first using the primary operator, and then individual, **inverted** messages can be combined to this *total sum* to get the value of updated messages for further propagation(see figure 7). This saves upon a number of gates, as the combining of **all** messages is a *shared computation*.

We use total-sum-first approach in our design, wherever possible. We will see instances of that in the next few sections. All the incoming messages are first summed up to calculate the **total sum**. The phase of time in which this computation is done is termed as the **accumulation scan**[21]. The second phase of computation, the calculation of residues, can be termed as the **output scan**.

In the next few sections, we will describe portions of the complete datapath of bit processing unit. The complete datapath is depicted in figure 8.

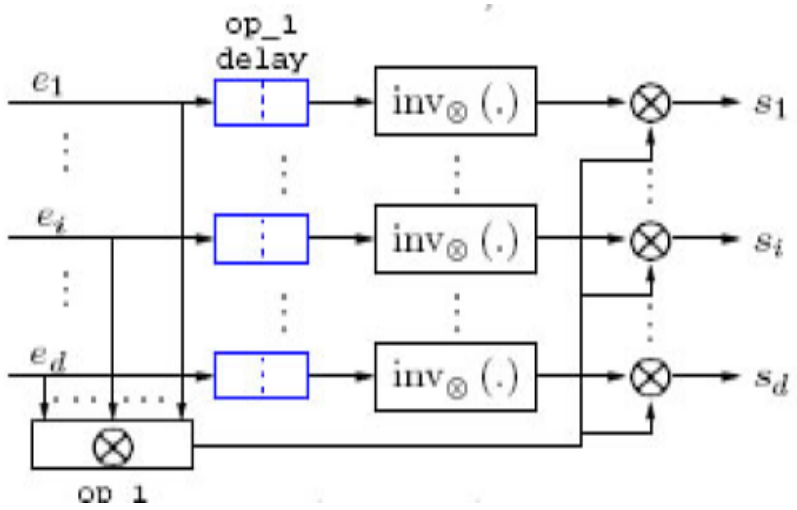

Figure 7: Total-sum-first calculation of Residues

7.1 Input Preprocessing

The accumulation scan is preceeded by preprocessing of bit node inputs, the check-to-bit messages. The data flowing through the decoder across various iterations is *signed data*, and hence *2's complement arithmetic* is preferred for implementing the computation, wherever possible(e.g. in bit processing units). However, as explained in next section, check processing units follow *sign-magnitude arithmetic*. Hence, bit processing units, as a *design option*, have their data inputs as well as outputs in sign-magnitude format. This in turn implies that the input data to bit processing units be first converted into 2's complement format, before being presented to various arithmetic components (such as signed adder) for computation.

A major part of computation in bit processing units is signed addition/accumulation of messages to form the *total sum*. **Repeated fixed-point addition** can potentially lead to overflow conditions. Hence we choose, in order to bring down loss of accuracy, a higher width for internal datapath so that there is **no truncation or rounding** involved during accumulation scan. At some point of time, we will need to bring down the datapath width back to the width of system's global datapath. By doing this shrinking at the end of output scan, the rounding errors that will happen on data after *subtraction*(in output scan), in general may be less than in the case when no higher internal bitwidth is employed.

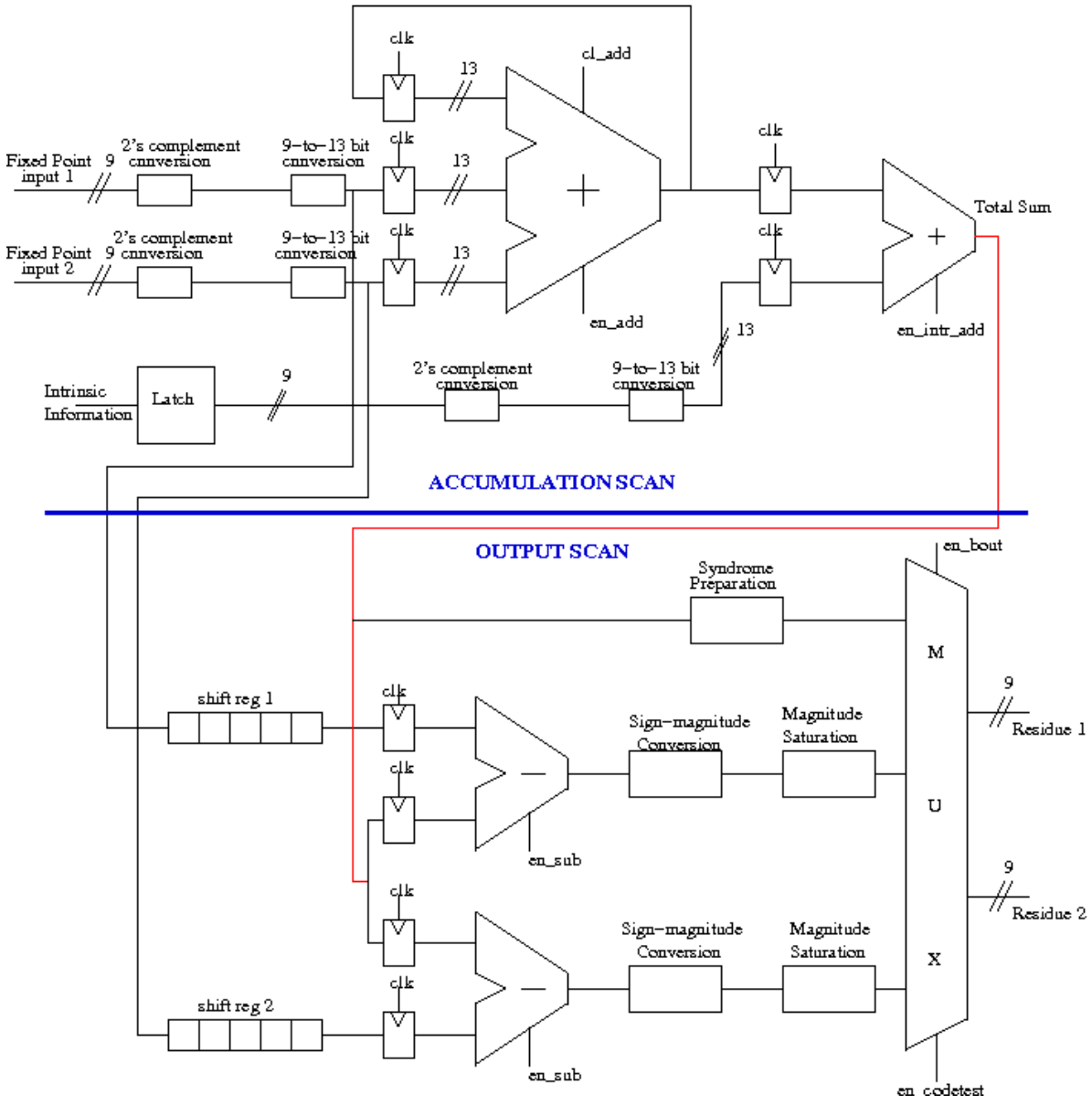

Figure 8: Complete Datapath of Bit Node

To expand a fixed point 2's complement datum to a higher bitwidth, we do an obvious *sign extension* of the datum. The number of additions that take place in accumulation scan are 9 right now, for the message inputs, and 1 more to add up the intrinsic infor-

Input (sign-magnitude)	2's complement
011010000	0000011010000
101000100	1111110111100

Table 1: 2's complement and sign extension

mation. Hence,the maximum possible number of bits in which we can represent a datum that is 10 times a 9-bit fixed point input, is $(9 + \lceil \log_2 10 \rceil)$, i.e. 13 bits. Hence we choose the internal datapath width of bit processing units to be 13 bits long, to accomodate worst-case accumulation overflows. This 13 bit representation is also a fixed point representation. Expecting the integer part to overflow, the representation consists of 1 bit sign + 7 bits integer + 5 bits of fraction.

Two examples of preprocessing of positive and negative incoming datum to the bit processing unit is shown in table 1.

7.2 Accumulation Scan

The first part of accumulation scan, in a *multi-cycle synchronous operation*, computes the total sum of all the preprocessed input messages, taking 2 inputs every clock cycle. This is done by implementing a 3-input, 13 bit clock-synchronous signed adder with a synchronous clear. As mentioned earlier, 2 inputs are processed at a time, because we are interested in leveraging the advantages of perfect access patterns during reading of check-to-bit messages from check memory blocks. Hence we take 5 cycles to add up 9 input messages. In general, for m inputs to the adder, the accumulation is complete in $\left\lceil \frac{m+1}{2} \right\rceil$ cycles. The 3^{rd} input of the adder generating the total sum is hence a feedback input from the output, which feeds back the partial sum of messages accumulated so far back into the addition process. To start with a partial sum of 0, a *synchronous clear* has been provided in the adder, which when enabled, sets the output to 13b'0. This particular instance of signed adder is shown in top right of the figure 8, where the accumulation scan itself is depicted as the upper half of the datapath (above the blue line which partitions the computation). We set the last (10^{th}) input in 5^{th} cycle to 13b'0 as a dummy input to the adder, since we have only 9 real inputs. This 10^{th} input is ensured as 13b'0 by the interconnection network between the memory blocks and the processing units. This will be explained in the interconnects section 10.

The second part of the scan comprises adding the intrinsic information, in form of a log-

likelihood ratio(LLR), to the total sum. This is accomplished by another signed adder, which adds the 2's complemented and sign-extended version of the intrinsic information value to the total sum. We use the same signed adder, but instead of implementing any feedback(which is not required for a single-cycle operation), we choose to provide the 3^{rd} input as 13b'0.

The synchronization of preprocessing of input values before they are added, and providing the enable signals to both the adders subsequent to input preprocessing, are taken care by a group of signals, which include enable of adders, clear of adders, sign-extension enable, etc.

As another design option, we latch the intrinsic information provided to the decoder system inside the bit processing unit. This is because in the worst case, we expect intrinsic information to be provided to the decoder system only at the time of beginning of decoding. Since in every iteration, bit processing units need this information for accumulation, we have to latch the data for these future usages. Intrinsic information is expected to be provided in sign-magnitude format itself, for sake of consistency of data inputs on the interface of the bit processing units. The preprocessing of intrinsic information is done once every iteration, whenever required.

Since the next, output scan is itself a multi-cycle computation, the output total sum is required to be kept constant for multiple cycles. This means we should be able to create a latch kind of effect at the end of second adder. We achieve so by making the each bit of the output of signed adder pass through a FDE macro(provided on Xilinx FPGAs), which preserves the bit value as signal on the output wire, even after its enable has been pulled down. For details, one may synthesize the signed adder, and see its technology view/map in various synthesis tools.

7.3 Output Scan

The residues are calculated for each bit processing unit, by subtracting the input messages from the total sum, in the same order that they were presented to adders during accumulation scan. Hence a shifted version of inputs is created as inputs to the output scan datapath. We need two fundamental components in the output scan.

- **Shift Registers**: Since the same inputs are used both in the accumulation scan as well as the output scan, a copy of them is tapped from the place where they are input to the first adder in the accumulation scan. Once an input during accumulation scan is converted into 2's complement form, and sign extended, it is passed

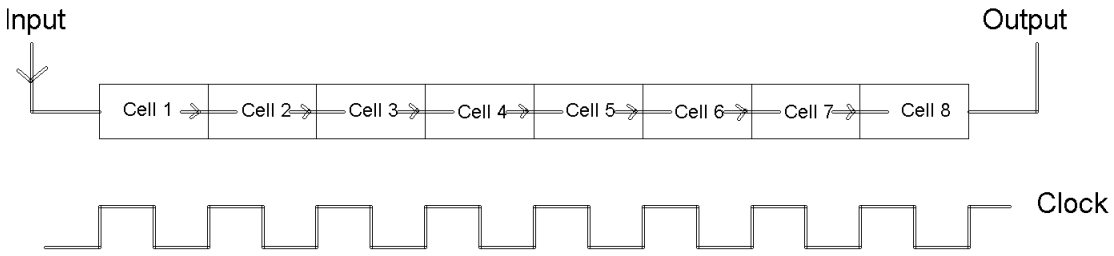

Figure 9: Shift Register block

through a synchronous shift register. Each cell of the shift register is hence 13-bits wide. Data shifts through the cells on the rising clock edge. The shift register block is shown in figure 9.

The number of cells in the shift register have been fixed to match the latency of the accumulation scan(6 cells). By the time when the accumulated total sum is stable on the output of the second signed adder, the first(second) shifted input is also stable on the first(second) shift register output. In the next 4 cycles, 3^{rd} and 4^{th}, 5^{th} and 6^{th} ... inputs appear at the output of the shift registers, as if the inputs were made to pass through some pipeline. There are two shift registers, one each to generate one of the two inputs for each of the two subtractors that calculate the residues. Since two inputs have been accessed per cycle during accumulation scan, only two residues are calculated for output per clock cycle. This reuse of the inputs alleviates the need to access the check memory blocks for a second time.

- **Subtractors:** The main computation in output scan is to calculate the residue by subtracting individual messages from the total sum, again in a *multi-cycle synchronous operation*. To again leverage perfect access patterns during output to bit memory blocks, 2 subtractions are performed every clock cycle. This is done by implementing a 2-input, 13 bit clock-synchronous signed subtractor. Each subtractor operates on 2's complement data, which are sampled on the rising clock edge. These subtractors are controlled by two enable signals. These enables are turned on only after the total sum has appeared, so as to save unnecessary computation and power wastage. The output of each subtractor is tristate only the enable is pulled down, i.e. the its inputs become invalid.

These particular instances of signed subtractors is shown in bottom middle of the figure 8, where the output scan itself is depicted as the lower half of the datapath (below the

blue line which partitions the computation).

7.4 Output Preprocessing

As mentioned before, the output of bit processing units is in sign-magnitude format(for ease of implementation of check processing units). The output of subtractors used in output scan produce data, which is 13 bit fixed point 2's complement data. Hence the data needs to be shrunk to 9 bits in sign-magnitude representation. Further, the magnitude part needs to be saturated from (7 bits integer + 5 bits fraction) to (3 bits integer + 5 bits fraction) to implement rounding off before the final write-back to bit memory blocks.

Each 13-bit output of (two) signed subtractors is first reconverted into its magnitude by calculating its 2's complement. The sign of to-be-output residue is assigned according to the MSB of subtractor's output. It is known that sign-extension preserves the sign of data across additions and subtractions. However, the magnitude of the subtractor output may be 12 bits long, and has to be desaturated to 8 bits. The maximum possible number that can be represented using 8 bits is $11111111_2(7.96875_{10})$. Also, if the integer part of the *absolute value* of subtractor's output is greater than 0000111_2,the (complete) 8-bit residue magnitude is pegged at 11111111_2.

Once both the sign and magnitude of a residue are generated in the proper format, it is written back to the bit memory block. The correct address to be written to is generated by the address generator to the memory, so the processing unit only performs a passive write by providing the data at the inputs at the memory blocks.

The output of the bit processing unit is further multiplexed with another signal other then the bit-to-check update messages. This signal represents the hard-estimate of the bit value, to be used for codeword testing. We explain this in the next section.

The control signals to the processing unit, and the write-enable for the ports of the memory blocks have been synchronized such that when the input to the memory block ports becomes valid and stable,the write-enable is asserted and data is written to the memory block. Since all stages in the processing datapath are synchronous,this serialization of control is straight forward. Every clock cycle after the first data on the memory block ports becomes valid, a new data is written into consecutive locations by sequential addressing. This will be discussed in detail in section 9.

7.5 Codeword Testing

During each iteration, to test whether an estimate for the codeword using the soft decoding algorithm is correct, a codeword test is performed by guessing a value for every bit, and then evaluating the constraints from the guessed values. Most of the decoding algorithms deploy this guessing technique, based on different criteria. For example, Majority Logic Decoding estimates the value of a bit based on the number of parity checks that the bit is involved in, and their success or failure. In a Sum-Product decoding algorithm, the guess value is based on the probability value of the bit being a 0 or 1 depending on the value of the constraints it is involved in.

Although the constraints are evaluated in the check nodes, the guess for the value of the bit is made in the bit node. From equation 7, we have,

$$\lambda(c_n|\mathbf{y}) \;=\; \text{Intrinsic Information} + \sum_{m\in\mathbf{M}_n} \lambda(z_{m,n}|\{y_i : i \neq n\})$$

This is the same as the accumulated sum(total sum) during the accumulation scan in the bit update. Hence,once the accumulation scan is complete, the bit processing unit can make an estimate about the value of the bit using the following condition(refer equation 2).

$$\begin{aligned} \hat{c}_n &= 0 \text{ if } Pr(c_n = 0|\mathbf{y}) > Pr(c_n = 1|\mathbf{y}) \\ \hat{c}_n &= 1 \text{ if } Pr(c_n = 0|\mathbf{y}) < Pr(c_n = 1|\mathbf{y}) \end{aligned}$$

The MSB of the total sum indicates whether it is positive or negative, because the data is in 2's complement form. This way, all guess values of bits can be allocated to either 0 or 1, at the end of the accumulation scan. Over the sequence of such guess values(the bit string/vector of guess values), $\mathbf{z}^T$, we need to perform $\mathbb{H} \cdot \mathbf{z}^T$, to check if channel information samples have now converged to a codeword guess. Given that rows of $\mathbb{H}$ are parity check constraints, this operation is essentially equalivalen to checking whether the guess value of bits satisfy the parity checks, using XOR operations. Since the hardware to do such operations is already present in check nodes, this computation of validating the guess value sequence is delegated to the check processing units. To pass the guess values(bit) as if they were bit-to-check messages to the check processing units, they have to pass through(be written to) bit memory blocks. Since the memory blocks are 9-bit wide, the guessed bit value is replicated for each of the 9 bits, to form a message called '**value vector**'. Hence, a bit with guess value of '0' will have a value vector of 000000000, and a bit with guess value of '1' will have a value vector of 111111111. This 'value vector'

is written to locations in the bit memories that are specifically reserved for passing the guess values to check processing units for constraint checking. In the geometry we have used, the correspoding locations are $\mathbf{11}_{10}$ **and** $\mathbf{12}_{10}$ of each memory block. The value is written into 2 addresses because the check nodes will again access them from the bit memories in a perfect access sequence (binary access). The generation of addresses by the address generators and the control signals for writing in the memory blocks are provided for by the centralized controller.

Thus the check processing units are enabled twice, once for check updates and once for codeword testing. In the first case, only the sign data-subpath is selectively enabled to do perform the computations. The execution of $\mathbb{H} \cdot \mathbf{z}^T$ boils down to stimulation of sign data subpath within the check processing units, where the sign bit during codeword test simply represents the guess value, not the sign of any input residue. The accumulation of all guess values in a total sum first implementation within the sign data subpath reveals whether the corresponding parity check constraint was satisfied, or not. Hence the outputs of 2-input XORs in figure 11 can be ignored during codeword testing.

It has to be noted that *when the 'value vector' is being written into the memory blocks, the bit residues continue being calculated alongside by rest of the datapath, irrespective of whether the estimate eventually turns out to be right or wrong*. This **forward speculative scheduling** is similar to the approach to conditional branching in the instruction pipeline in microprocessors, where filling of pipeline from next instruction continues irrespective of the satisfaction of the branching condition, which is only known after few cycles. Forward scheduling is justified by the following logic.

Suppose the codeword is correctly decoded in **n** iterations. In all but the *last iteration*, the estimate of the codeword is sure to fail. So, instead of *stalling* the decoding process for many cycles, to see if the estimate is right, in each of the *n-1* iterations, it is better to continue doing computation of bit updates, because these updates will only be required to be discarded in the last (and the only successful) iteration. This increases the average throughput of the decoder by decreasing the average latency of an iteration.

8 Check Node Architecture

Check-to-bit updates' calculation by check nodes is implemented in similar way as their bit counterparts, with a difference: the computations are done in the log(tanh()) domain. Hence the datapath of the check processing unit is more complex than the bit processor. Further, the sign and magnitude of inputs are are treated/used in computation

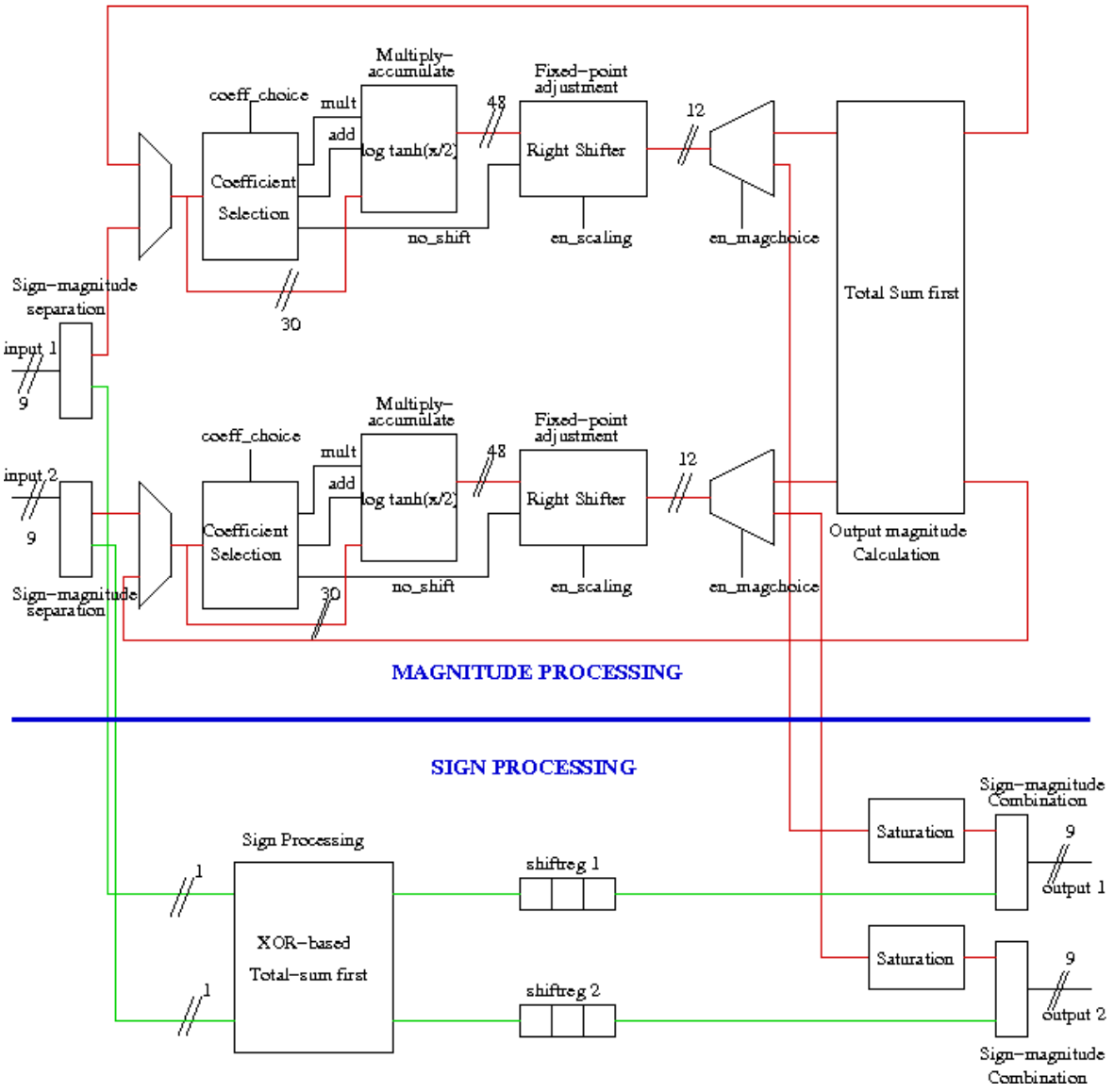

Figure 10: Complete Datapath of Check Node

along different data subpaths. Hence the design of these two subpaths can be treated separately.

8.1 Sign-Magnitude Separation

The input messages to check processing units is already in sign-magnitude form. Hence sign-magnitude separation just amounts taking the leading bit as sign, and the remaining bits as **fixed-point, positive** magnitude.

8.2 Sign Processing

The sign of the check-to-bit update message is generated in a manner similar to the magnitude(described later), albeit using a different operation. The sign of outgoing check-to-bit residue is calculated as follows(refer equation 10).

$$S_{m,n} = \prod_{j \in \mathbb{N}_{m,n}} S_j$$

where

$$S_j = \frac{|\lambda(z_{m,n})|}{\lambda(z_{m,n}}$$

If the data is represented in the LLR form, then the above description of computation of overall sign bit for a check-to-bit update for bit node **m** can be implemented using XOR of sign bits of all the bit-to-check updates received by that check node, except the node **m**. The XOR operation, from equation 7, can be modeled as 'sum' operation as well, as defined in *binary arithmetic*. So, similar to the total sum first approach, we can first find the total XOR of all the sign bits. In the place of subtraction for the magnitude of the residue, the inverse operation of XOR needs to be applied to remove individual signs from the accumulated XOR to get the residues. By noting that for a bit a, $a \oplus a = 0$, we have

$$(a \oplus b \oplus c) \oplus a = b \oplus c$$

$$(a \oplus b \oplus c) \oplus b = a \oplus c$$

$$(a \oplus b \oplus c) \oplus c = a \oplus b$$

We can see that XOR is self inverting. Hence the individual sign bits need to be XORed again with the total sign, to generate the individual signs of the outgoing residues. Thus,the sign processing is analogous to the magnitude processing, contains two scans. See figure 11 for complete depiction of sign processing.

- **Sign Accumulation scan**: The calculation of total sign is again a *multi-cycle synchronous operation*, taking 2 inputs every clock cycle. This is achieved using a 3-input, 1 bit clock-synchronous XOR with a synchronous clear. Again it takes 5 cycles to add up 9 input signs. The 3^{rd} input of the XOR generating the total sign is hence a feedback input from the output, which feeds back the partial sum of signs of messages accumulated so far, back into the addition process. To start with a partial sum of 0, a *synchronous clear* has been provided in the XOR, which when enabled, sets the output to '0'. It has to be noted that the 10^{th}input to the check processor is hardwired to 000000000. Hence the XOR of the 9 signs will not be impacted due to an additional XORing of a '0' in the 5^{th} cycle, which happens due to use of perfect access patterns. This hardwiring will be explained in the interconnects section 10.
- **Sign Output scan**: The sign part of outgoing residues are finally computed using two 2-input XOR gates, one for each sign input, by XORing the input signs with the total XOR(sign) from the accumulation scan. The reuse of the sign inputs is again achieved by implementing 2 shift registers of 1-bit wide cells to provide delayed inputs to the output scan's XOR gates, once the sign accumulation scan is complete. Path latencies are matched by the number of shift register cells, which is 5.

Control signals for the shift registers and *enable* for the 3-input and the 2-input XOR gates are provided at the time by lazy evaluation. That is, whenever a result of sub-computation is needed, and the inputs to the sub-computation are also valid, the control signals are correspondingly activated. of the valid inputs by the control logic. As pointed out later, the latency harmonization between the magnitude and the sign processing data sub-paths for check-to-bit updates is achieved by postponing the output scan to the last possible cycle(i.e. a cycle before sign-magnitude combination starts off).

8.3 Using Sign Datapath in Codeword Testing

In the description for bit processing units, it was stated that **just after** the accumulation scan in each bit processing unit, an estimate for the value of the bit is made, and is communicated to the check processing units via the bit memory blocks. This is because the sign data subpath of check processing units represent the constraints that need to be checked to see if the sequence of decoded bit values form a codeword($\mathbb{H} \cdot \mathbf{z}^T = 0$).

Hence, when enabled for codeword testing, the check processing units read the 'value vectors' in a perfect access sequence from the bit memory blocks that they are connected

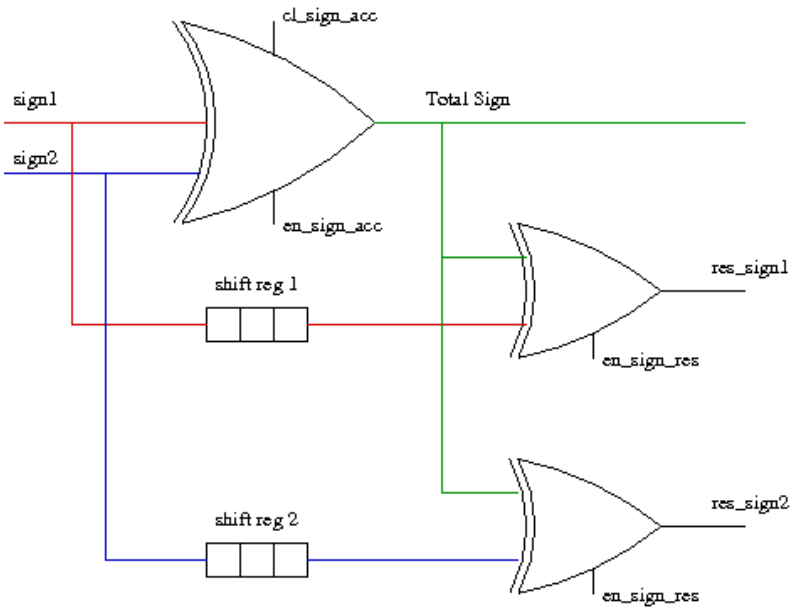

Figure 11: Sign Processing in Check Node

to. The sign bit of each datum read represents the value of the guess for that bit. Hence, the sign processing datapath is made to evaluate the total XOR of the signs of all the incoming input guess bits. This is equivalent to checking the parity constraint with the guesses for the bit. The magnitude datapath has no role to play during this phase, and hence all its enable signals are not pulled high in this time period.

After the sign accumulation scan, the value of each constraint is collected from the check nodes by the topmost component of hardware, and sequenced together. This sequence represents the **syndrome** of the guess vector. A *syndrome* is defined by the vector formed by the guess values for all bits [17]. If the syndrome of the guess vector is [0], the guess of all bits is said to form a valid codeword, thus leading to halting of the decoding process. A *valid_word* signal is asserted high and the guess vector is put out on the *output* ports. If the syndrome is not [0], the decoding continues and the *valid_word* signal is left asserted as low. This signal is therefore the indicator of the end of the block decoding.

It has to be noted that the codeword test process starts after the accumulation scan in the bit processing units, and continues in parallel to output scan of the same. So, *while the bit processing units are busy calculating the bit-to-check updates after the 'value vectors' have been written into bit memory blocks, the check processing units are evaluating the constraint values according to the guess values for each bit.* This *forward speculative*

scheduling has been argued and justified earlier, since only in the successful, which happens to be the last, iteration of the decoder, will the bit-to-check updates generated by the bit PE's during the codeword test need to be discarded. By generating it alongside always, we decrease the latency of each of the last but one iterations that precede it. It also keeps the processing units busy for a greater no. of cycles on an average, a feature desired by most parallel systems.

8.4 Magnitude Processing

As evident from 3^{rd} step of algorithm(refer section 2.3.7, in the magnitude data-subpath, inputs go through $\phi(x)$ transformation, before output's magnitude is formed in total sum first fashion. Further, the output' magnitude is then reconverted back using *inverse transformation* identical to $\phi(x)$. The traditional way to implement this function has been usage of lookup tables(LUT). However, each node will need its own copy of LUT to avoid memory access conflicts, and in a fully-parallel design, this cannot be afforded. Hence a **piece-wise linear approximation** using a *slightly modified* **Masera's model** [16] is used to implement this function. Using this model, computation of $\phi(x)$ boils down to a multiply-accumulate(MAC) operation for multiple input intervals. After such conversion, the two scans of accumulation and output take place in a way that is almost identical to that in the bit processing units.

The magnitude of the residue is, from equation 10,

$$\phi\left(\sum_{j\in\mathbb{N}_{m,n}}\phi(M_j)\right)$$

where

$$\phi(x) = -\log\tanh\left(\frac{|x|}{2}\right)$$

Incidentally, $\phi(x)$ is a self inverse function: $\phi(x) = \phi^{-1}(x)$. The magnitude part of residue calculation per check processing unit follows a data subpath that is shown in figure 12.

Traditionally, the convenient way to implement a real valued function in a *precision-constrained manner* has been a look-up table. In LUT implementation, the value of the function is looked up from a table by indexing into it using the input value, and then

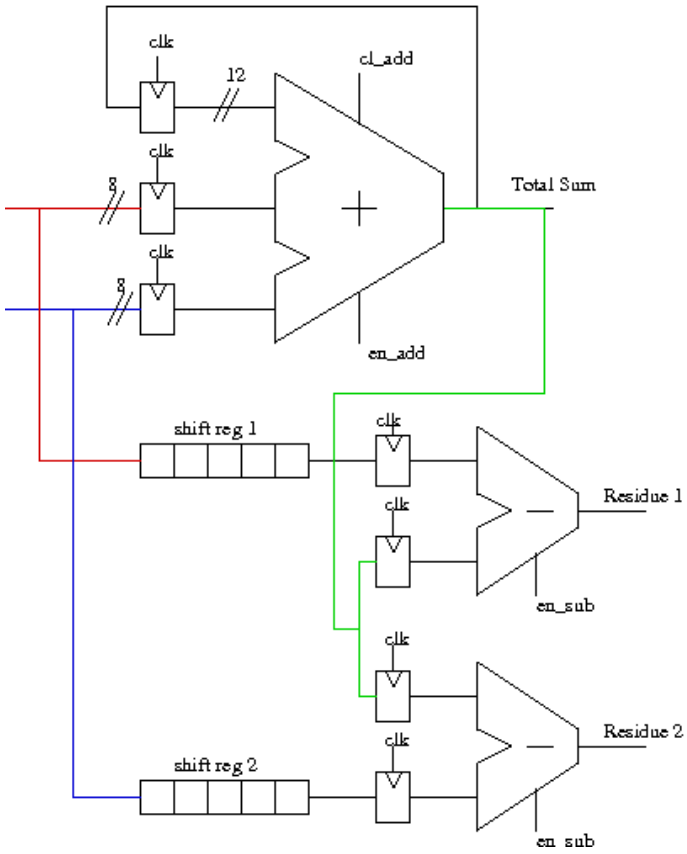

Figure 12: Magnitude Processing in Check Node

returned. Such tables are generally in the form of a structured hash table with input ranges providing a faster search. The simplest look-up can just be a ROM-based look up with a one-to-one correspondence between the address of the function value, and the range of the input. However, [15] shows that a large performance loss is induced by the quantization of $\phi(x)$. Also, its implementation typically requires very expensive look-up tables FPGA LUTs. The function is highly nonlinear and not limited/bounded, owing to the vertical asymptote at x = 0. Thus, a direct implementation by means of look-up tables (LUTs) would require large memories with lots of entries in order to achieve the proper decoding performance. Better results in terms of both performance and complexity are obtained by resorting to a modified decoding processing.

Masera et al have devised a piecewise linear approximation to evaluate the function [16], shown in Table 2. The approximation is shown to be a good solution when the

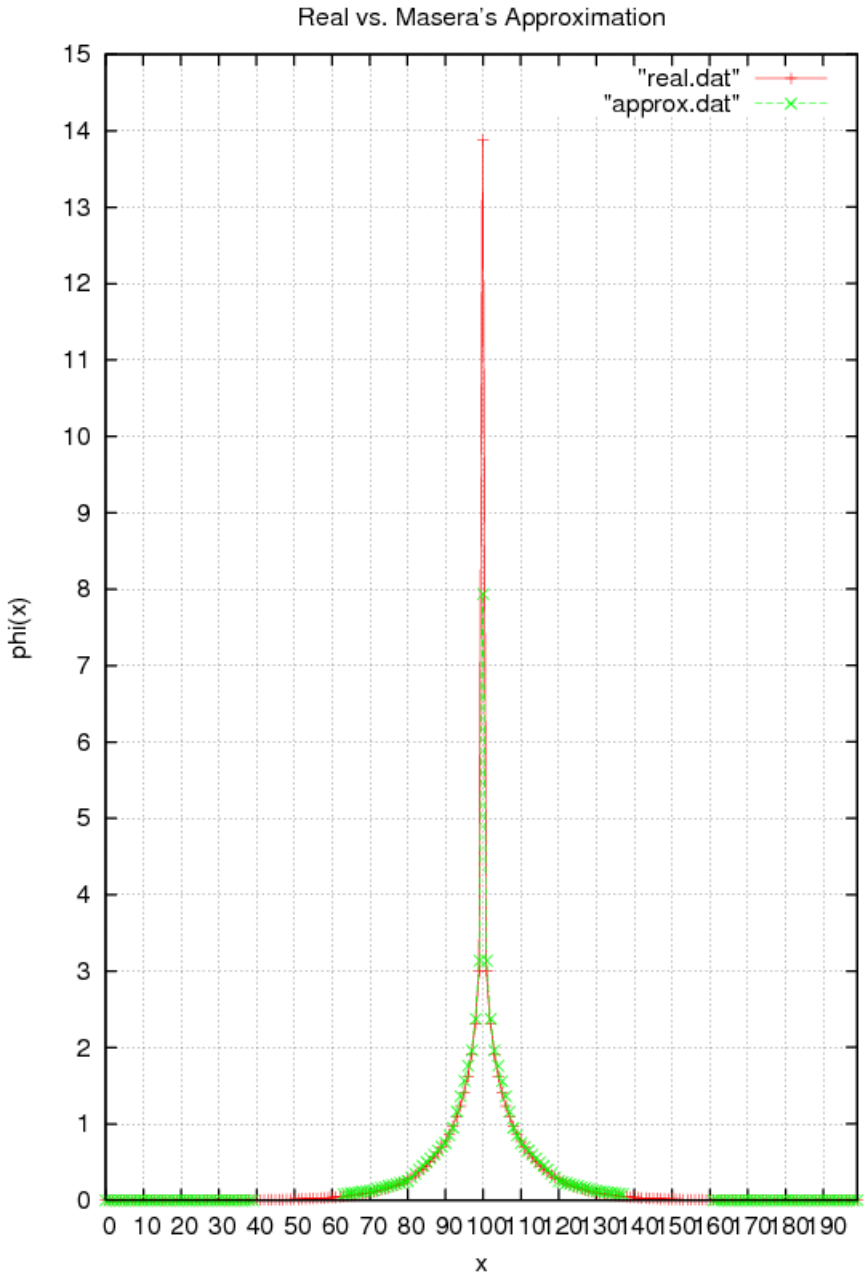

Figure 13: Actual vs approximation: ϕ(x)

$\phi(x)$	x
$-48x + 7.9375$	$\mid x \mid < 0.125$
$-7.5x + 3.875$	$\mid x \mid \leq 0.25$
$-2x + 2.5625$	$\mid x \mid \leq 0.75$
$-x + 1.75$	$\mid x \mid \leq 1$
$-0.5x + 1.25$	$\mid x \mid \leq 2$
$-0.125x + 0.5$	$\mid x \mid \leq 2.8125$
$-0.0625x + 0.3125$	$\mid x \mid \leq 3.75$
0.0625	$\mid x \mid \leq 6.0625$
0	otherwise

Table 2: Piecewise Linear Approximation for $\phi(x) = -\log\left(\tanh \mid \frac{x}{2} \mid\right)$

decoder is going to work near to the convergence and waterfall regions of the code (at low E_b/N_o values). It is expected to be fairly accurate since more linear pieces have been allocated in the range, where $\phi(x)$ is strongly nonlinear. Further, coefficients are chosen in order to be easily represented in fixed point notation, which avoids complex, potential, floating point multiplications. This approximation avoids numerical instability, because it is intrinsically limited when its argument is zero. Moreover, hardware cost has been reduced by selecting coefficients obtained through 'shift' operations, or sums of shifted data(e.g. $48x = 32x + 16x$), to reduce hardware complexity.

Figure 13 shows the comparison between the plots of the actual and the approximate functions.

The function is evaluated using the Digital Signal Processing DSP48E blocks available in the FPGA [23].The DSP48E slice supports many independent functions. These functions include multiply, multiply accumulate (MACC), multiply add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bit-wise logic functions, pattern detect, and wide counter etc. We configure the DSP48E slice to perform the multiply add function. Each DSP48E slice has a two-input *integer multiplier*, followed by multiplexers, and a three-input adder/subtractor/accumulator. We can use the multiplier to evaluate the multiplication of the input with the multiplicand, and then multiplex the intermediate result onto a input of the adder, which adds an offset constant to produce the final approximation to $\phi(x)$.

All the above linear functions are simultaneously evaluated for both the inputs, using one DSP block for each (input's) range. The multiplier within DSP48E block is a 25 $\times$

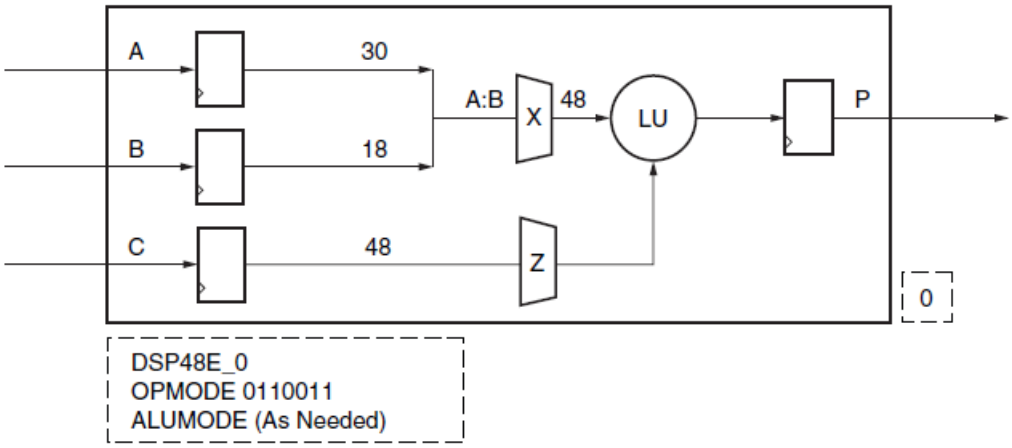

Figure 14: Cascading DSP multiply-add in a DSP48E slice

18 integer multiplier, and the *cascaded* adder/subtractor is a 48 bit adder/subtractor. The cascading of these two operators is set by the OPMODE and ALUMODE register values in the DSP48E slice. The block diagram of DSP48E slice is reproduced from [23] in figure 14.

To transform each of the check processing unit's inputs for DSP48E consumption, we zero-pad the magnitude part of input to make it 25-bit input. Further, based on the range of input, which is dyanmic in nature, we make a choice of the multiplicand and the addend in an earlier cycle. The multiplicand and the added are specific to the input value, and hence specific to the two inputs per iteration themselves. Since the same DSP48E slice has to operate on two different data, once for forward transformation $\phi(x)$ and once for reverse, **identical** transformation $\phi^{-1}(x)$, we need a multiplexer in from of DSP slice's multiplier input to multiplex 2 different value of the multipliers. In fact, the choice of multiplicand and the addend is done based on the output of this multiplexer.

A modification to Masera's model in [16] is that whenever $\mid x \mid \leq 6.0625$, $\phi(x) = 0.0625$ instead of -0.0625. This modification we had to undertake $\phi(x)$ is always a positive function, and experimentally we found this modification to bring down the difference between two curves(real vs. approximate) further down.

Another issue with employing the DSP48E slice is that the multiplier within this block is an integer multiplier. Hence the multiplier does not recognize any location of decimal

point. To overcome this issue, we made sure that every multiplicand to the multiplier is an integer. This is done by multiplying the multiplicand as per the modified Masera's model by 2 successively, *until* the multiplicand becomes an integer. Correspondingly, we multiply the addend also by the same factor, which is a power of 2. Thus, the approximation of $\phi(x)$: 0.5x + 1.25 when $|x| \leq 2$ becomes $\phi(x) : -x + 2.5$ when $|x| \leq 2$. Hence we need to correspondingly divide the output of DSP48E multiply-add operation by **same factor**(2 in this example) to get a proper fixed-point mutliply-add operation. The loss of precision due to this division is unavoidable, because our data representation is *fixed point*. The division is achieved by right-shifting the result by a copy of multiplying factor, which is stored at the time of scaling up the multiplicand and the addend. Since MAC operation takes 1 cycle, the scale factor is delayed by 2 cycles and then applied to the output of the DSP48E slice. Table 3 shows the actual versus evaluated functions in each DSP instance in a check processing unit.

Actual function	Evaluated function	Scaling factor
-48x+7.9375	-48x+7.9375	1
-7.5x+3.875	-15x+7.75	2
-2x+2.5625	-2x+2.5625	1
-x+1.75	-x+1.75	1
-0.5x+1.25	-x+2.5	2
-0.125x+0.5	-x+4	8
-0.0625x+0.3125	-x+5	16
0.0625	0x+0.0625	1

Table 3: Actual and evaluated functions in the DSP instances

It is the *beauty of Masera's model* that we are able to multiply the multiplicands by a factor that is a power of 2, and get an integer. In general multiplying a fraction by power of 2 not always leads to an integer. In fact, even while representing the addends which in general are deep fractions, we do not loose any precision.

The multiplier and the adder/subtractor within DSP48E block follow *2's complement arithmetic*. Hence, wherever needed, we do *sign-extension* of data(such as multiplicand) to make the input adhere to proper width.

The transformation of x to $\phi(x)$ takes one machine cycle. However, given that multiplier is involved, the propagation delay of MAC circuit is quite high. It was indeed found that MAC unit/DSP48E block falls in the critical path of the decoder system. Right now pipelining in the DSP block has been kept disabled, to decrease the overall latency of

the entire check node update. The options to change this is programmable pipelining of input operands, intermediate products, and accumulator outputs, which will enhance the throughput. The corresponding pipeline registers are introduced through configurable attributes of the DSP slice [23].

After the conversion, the two scans of *accumulation and output* take place almost identical to that in the bit processing units, with few exceptions as follows.

- **Adders**: The inputs to the adder are 8 bits(sans the sign bit). Also, these inputs are guaranteed to be *positive*, as the transformation $\phi(x)$ is *always* positive. Hence using 2's complement arithmetic during the scans is not required. The outputs of the adder are 12 bits long, to accomodate the maximum possible overflow of 4 bits for adding 9 inputs. Similar to the adders in the bit processing units, the enable control signal is provided to tri-state the output whenever required. Outputs are stable on the rising edge of the clock.

  The 10^{th} residue, fed in 5^{th} cycle to the adder, is $\phi(12b'0)$, since 12b'0 is input from the interconnect. Interestingly, this value is not 0. It is actually 11111100, and hence there is a **wrong** offset of 11111100 in the total sum calculated by this stage. This offset is taken care by the subtractors later.

- **Subtractors**: Two subtractors, each with a data width of 12 bits, calculate the residues in order of the arrival of inputs over 5 cycles. The output is stable on the rising clock edge.

  The first input of the subtractor, in all 5 cycles, is the *total sum* of residues' magnitudes, which is provided by the accumulation scan. To nullify the effect of wrong offset 11111100, the subtractors calculate in1-(in2+11111100), where in2 is the shifted version of each message input to the check processing units.

- **Shift registers**: To reuse the converted inputs while calculating the final check-to-bit messages, the bit-to-check messages are passed through a shift register. Each cell of the shift register is obviously 8 bits wide. Data shifts through cells synchronously with the positive clock edge. The latency of the shift register is matched to the latency of the accumulation scan, that is, 5 cycles.

As there are 9 inputs,the latencies of the accumulation and the output scans within magnitude processing are 5 clock cycles each. Similar to bit residues, these magnitudes undergo desaturation. After the magnitudes are pegged to 8 bits again,the residues have to be reconverted using the same function $\phi(x)$; the function is self-inverting. For this

purpose, we use the same DSP blocks again, whose inputs are multiplexed between the bit-to-check messages and the saturated residues, as described earlier. The *phase_choice* control signal drives this multiplexer's output. The choice of multiplicand and the addend is also made afresh, since these values depend on the value of x, the input to Masera's approximation model. This reuse of the DSP blocks is necessary to avoid duplication of resources.

8.5 Sign-Magnitude Recombination

After the inverse transformation of residue magnitudes, the outgoing sign is merged back into the outgoing message's magnitude. Hence there is a need for *latency harmonization* between sign and magnitude data subpaths. This is done by doing *lazy evaluation* of the final signs(or the output scan of the sign data subpath). The 2-input XORs are hence activated a cycle before sign-magnitude combination starts off. To be able to do so, the 3-input XOR gate, which is responsible for sign accumulation, is front-ended by a FDE macro, which acts as implicit latch to let the accumulated sign persist for multiple cycles(till another driver on the signal changes it). Further, the length of the shift registers in the sign processing datapath is also increased beyond 5 cycles need by sign accumulation scan, till the last possible cycle. With this lazy evaluation scheme, the sign and the corresponding 8-bit magnitude of the residue become stable in the same clock cycle, and hence can be combined(put in different bit intervals of output signal) right in the next cycle.

At the time of combining the sign and magnitude, care is required to be taken about the factor $(-1)^{|\mathbb{N}_m|}$ in check messages. One may refer to equation 10 to understand this factor.

The final residues are then written back into the check memory blocks, to be passed on to bit processing units for the next iteration.

9 Memory Architecture

Memory blocks are required for storing the inputs during their consumption over multiple cycles by a processing unit.As described earlier, there are two types of required memory blocks: bit memory blocks and check memory blocks. Each type of memory block is *written into* by processing units of the *same kind*, but are *read by* processing units of the *opposite kind*.

These memories need to be true dual-port, since each node consumes 2 inputs at a time. By the design, in any cycle, the messages being consumed in different nodes are never the same, and hence latency penalty arising out of memory conflicts do not arise. The memory architecture is distributed; each node unit owns a memory block of its own having a width of 9 bits and depth of 12 locations. The data corresponding to each node is stored in the same order as the line/point index in the corresponding geometry. Hence the address generation units of these memories are simple 4-bit counters. Since each block receives only two read requests per cycle and has dual-port capability, the entire access is collision-free, the greatest asset of our architecture.

9.1 Implementation of Memory Blocks

Xilinx FPGAs have embedded block RAM resources(hard macros), that are used to implement the memory blocks we desire. Since these macros are parameterized, we use block memory generator to instantiate memory blocks by providing certain values to the configurable parameters of block RAM macro. One can, in fact, use a single memory primitive to instantiate memory blocks of arbitrary widths and depths. The generator is available as a feature of the CORE Generator software [26]. We use the Core Generator to generate block RAMs configured to our design.

9.2 Configuring Block RAMs

Using Core Generator, a variety of memory block configurations can be generated using block RAMs, such as single/dual port memories, ROMs or RAMs etc. To implement a perfect access pattern, we need to fetch two residues per cycle, and similarly store back two residues per cycle later on. Hence we configure the memory blocks to be r/w memory of the form RAM, having 2 ports that can operate independently of each other(true dual port, see figure 15).

For dual-port read/write memories on Xilinx FPGAs, each port operates independently. Each port can be configured for clock, pipelining and operating mode features. The True Dual-port RAM provides two ports, **A** and **B**, through both of which read and write accesseses to various memory locations can be done.

The GUI of CORE Generator allows us to configure the RAMs. Since the block RAMs are of 36 kilobytes each, a natural bitwidth for each memory port is 9 bits, leading to 4096 physical locations within the memory block. *Coincidentally*, we have chosen the

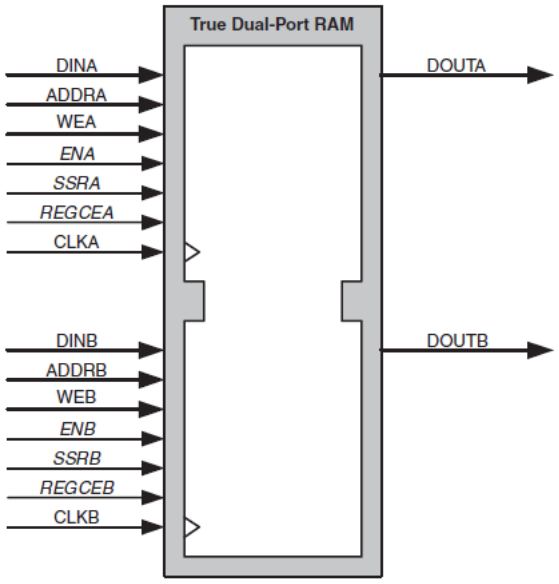

Figure 15: True Dual Port RAM

data/message bitwidth to be 9 bits as well. Hence configuring the parameter, (9-bit) byte write enable, to be true is of no particular advantage.

Further, each RAM is built up from the multiple fixed sized primitive RAM blocks within the FPGA. Since our data width is 9 bits, we can readily use the $2k \times 9$(or even $4k \times 9$) RAM primitives to build every bit and check RAM block. As such, the required user memory block size is achieved by concatenating this single physical memory primitive type in both width and depth. In our design, each user memory block has size requirement of 9 bits width and 12 locations depth, hence $2k \times 9$ memory primitive is best fit. Further, we use *fixed primitives* option of building user memory block, rather than *minimum area* option, since regularity is expected to help in layout of the design. The organization and the addressing of locations is explained in a later section.

There are three **operating modes** available for the RAM ports, which determine the relationship between the write and read interfaces for that port. Port **A** and port **B** can be configured independently with any one of three modes, because we are implementing a true dual port RAM. The modes differ in terms of the write cycles on the RAM ports.

All the 146 block RAMs in the decoder have been configured to operate in the **'NO CHANGE'** mode for both ports **A** and **B**. In *NO CHANGE* operating mode, the

driver/value on the output port remains unchanged, during a write operation at some location within the memory. As shown in figure 16, the data output is still the previous read data, and is unaffected by a write operation on the same port. This particular operating mode is needed to effect the simultaneous proceeding, without any stall, of 2 branches of computation in our decoder. So while check processing units need to read the *value vector* from bit memory block for codeword testing, bit processing units, at the same time, are ready with the bit-to-check update messages. To allow these messages to be written at some bit memory block locations, *different* from where value vector(s) have been stored, and at the same time allow the output of bit memory blocks to be latched to value vector for 5 cycles, the only option is to use *NO CHANGE* operating mode.

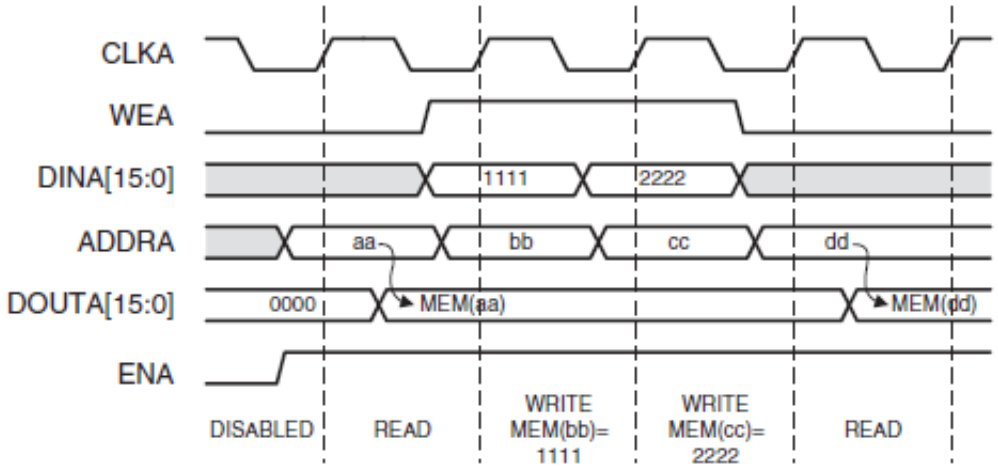

Figure 16: Example of NO CHANGE Memory Operating Mode

Thus, the reading and writing of data is performed by first enabling the entire memory block itself, for which a separate pin is configured. It is followed by providing the address on the address pins of a particular port, and then asserting/deasserting the *Write Enable* of the port. The *write enable* signal works dual-purpose: when 0, it implies a read operation in conjunction with block-level enable, and when 1, it implies a write operation. For the true dual port RAMs, both ports can be read/written to, independent of each other. The pipelining and the output register features have been right now *turned off*, to decrease the memory access latencies. In the current mode, both read and write access take one clock cycle. Also, we do not configure the RAM to have

a set/reset pin. Though the reset is not strictly required, we do an initialization of first 12 locations of the memory to 9b'0 just for precaution.

9.3 Assigning Messages to Memory Blocks

The memory blocks are connected to the processing units of the *opposite kind* as per the connections of the Tanner graph. The Tanner graph for our codes is itself the representation of interconnection between lines and points of PG(2,$\mathbb{GF}(2^3)$. We **fix** the convention such that the *bit* memory blocks and processing units correspond to the *points* of the geometry, while the *check* memory blocks and processing units correspond to the *lines*. By explicit construction, we find that point 0 of the geometry lies on the following set of lines in the geometry [21].

Point 0 : {*0,1,71,38,11,20,43,59,67*}

Similarly, other points in the geometry lie on the lines shown in the corresponding set, which can be obtained by using *shift automorphism* over the set representing lines incident on point 0.

Point 1: {*1,2,72,39,12,21,44,60,68*}

Point 2: {*2,3,0,40,13,22,45,61,69*}

...
...
...

Point 72: {*72,0,70,37,10,19,42,58,66*}

Thus, each bit memory block, corresponding to a point, is connected to the check processing units, representing the lines, that the point lies on. Analogous the bit memory blocks, check memory blocks are connected to bit processing units as per the incidence set of lines of the geometry:

Line 0: {*0,72,2,35,62,53,30,14,6*}

Line 1: {*1,0,3,36,63,54,31,15,7*}

...
...
...

Line 72: {*72,71,1,34,61,52,29,13,5*}

Each bit-to-check update message is written into the same indexed memory blocks, as

its generator bit processing unit has. However, this memory block, as a point, is read from check processing units, representing lines, to do their own computations. Same is the case for storage of check-to-bit update messages. Hence, as a byproduct, the ports of various memory blocks are connected as follows.

- The *input* data ports of each RAM/memory block are connected to the outputs of the processing units of the same kind.
- Clock input to all block RAMs is through the system clock
- The *output* data ports of each RAM/memory block are connected, via a *circular shift connector* at each port, to the processing unit of opposite kind. The connector loads the data available on RAM port on the appropriate wire in the interconnect. This part of design is explained in detail in the interconnect design, section 10.
- Address ports receive the addresses from respective address generators, which are common to all block RAMs of the same type.
- The block RAMs are selectively enabled based on the schedule of decoding. This enable signal is provided by the centralized control path.
- Similarly, the *write enable* signal is provided separately for each port by the centralized control path. However,the *write enable* for port **A** of all RAMs of the same type is common, and likewise for port **B**.

9.4 Internal Layout of Memory Blocks

The data corresponding to various check processing units is stored in a particular bit memory block in the same order, as the line index corresponding to its point index. For example,in bit memory block 0, data for check processing unit 0 is stored in location 0, in location 1 for processing uit 1, in location 2 for processing unit 71, and so on. The first 9 locations thus store the required message data. The order of internal storage is defined by the automorphisms that are used to generate the geometry as stated in [11]. Thus, even though the addresses are consecutive and the access sequential, the *organization of data is structured*. Hence the memory accesses are structured themselves.

The 10^{th} location in all the memories is set to 000000000 during the initialization sequence, and is never written to during any update. Data read from the 10^{th} location is dumped and not used. This design decision is explained in section 10.

Locations 11_{10} and 12_{10} in the bit memory blocks are reserved for storing the **'value vector'**. This storage happens after the guess for the bit value is made during the bit update phase of each iteration. These value vectors are then read by the check processing units to evaluate the constraint values from the guess value. A control signal to the address generator is used to load these special addresses on the memory block ports, when the value vector is to be written. Check memory blocks also have same number of reserved locations, since they are instantiated using the same component as the bit memory blocks. However, these locations(11_{10} and 12_{10}) of the check memory blocks are not used or accessed.

9.5 Address Generation

Address generation units provide the right addresses at the address ports, at the right time according to schedule(e.g. when data is stable and valid for a write operation). The address generation is different for bit and check memory blocks, but as such common/same for memory blocks of the same type. It generates addresses for both the ports of every memory block simultaneously. Since we use perfect access patterns to access memory blocks, every clock cycle, two addresses need to be generated. The access being sequential due to the sequence in which we internally store messages, memory address generation is also sequential, not random. Since the memory blocksare 12 locations deep, 4-bit addresses are required to access these memory locations.

During read/write access from both types of memories, data is accessed from sucessive locations in the RAM. Hence the address generation unit has two address output ports, which generate consecutive values of address. One of them is connected to the port **A** address bus of all memory blocks of the same type. The other is similarly connected to Port **B** address bus. Hence, port **A** addresses are always even, while port **B** addresses are always odd. Also, port **B** address is one location ahead of port **A**. I.e. when addrA is 0000_2, addrB is 0001_2 and so on. When read/write access is going on, then in every clock cycle, the address on each port is incremented by 1. Thus, addrA takes values of 0000_2, 0010_2, 0100_2, 0110_2 and 1000_2, while addrB takes values 0001_2, 0011_2, 0101_2, 0111_2 and 1001_2. These values cycle through in the next round of read/write access. In the reset state, controlled by an active high *start* signal to the address generation units, the port addresses are $0000_2$and 0001_2. It can be seen that there can never be any access collision between the ports of the memory blocks, due to this exclusive address generation.

During the codeword test phase, when the 'value vector' is being written into bit memory locations 1010_2 and 1011_2, the address generation units provide these special addresses

on the ports. An active high exclusive control signal is used to signal the start and the end of this write process. The check and bit address generation units are similar, except that this control signal *en_codetest* is always low for the check memory blocks' address generators.

The bit address unit is active during the bit update phase at three points. At one point, it aids the writing of the 'value vector', while at the other point, it aids the writing the bit-to-check residues calculated. Further, during check update calculations, bit memory blocks are being read by the check processing units, and hence bit addresses need to be continuously generated. Similarly, the check address unit is active when check memory blocks are being read by the bit processors for the bit update, and also during the check update for writing the check-to-bit residues into check memory blocks.

9.6 Intrinsic Latch

The only input to the decoder at the start is the soft estimate of the information received from the channel. This is in the form of channel, or *intrinsic information*. Since every iteration, intrinsic information needs to be added to form the total information within bit nodes, each bit processing unit needs to keep this piece of data latched within itself for future use across multiple iterations. Hence each bit processing unit reads these values from an input port, and uses a set of simple D-type latches to hold these values. There are therefore 73 instances of such 9-bit latches, having their own enable input signal.

10 Interconnect Architecture

Most of the VLSI implementations of LDPC decoding suffer from routing congestions due to the dense network of interconnects between the processors and memories. Interconnects end up consuming most of the floor space in the layout of the design [15]. Hence, efficient implementation of interconnects is critical to the efficiency of the decoder. To add to it, FPGAs are not optimized for global routing, but do provide good point-to-point connects in a localized area(neighboring CLBs).

The PG LDPC decoder for block size 73 has a not-so-dense network of wires between the processing units and memory blocks. There are two instances of global message carrying interconnects: those between the check memory blocks and the bit processing units, and the ones between bit memory blocks and the check processing units. Each memory block has 9 processing units connected to it, and vice versa. Thus,the total number of single

bit carrying wires is 146 *9*9 = 11826. Since the degree of each node is 9(can further grow as design scales), and given the classical flooding schedule that we use to compute the updates, bus architecture is not suitable for implementing the interconnect. This is because in a particular cycle, at least 9 units will try to simultaneously transmit on the bus, leading to *widespread congestion* in the interconnect. Hence, **Dedicated wiring** is used in to implement these connections within the *data path* in our architecture. The wires between the processing units and the memory blocks of the **same type** are expected to be *local*, due to the proximity in their placement. The wires required to interface control pins of multiple datapath elements, to centralized controller, are expected to be via global wiring.

Since 9 processing units access a memory block in total, over *multiple different cycles* using same two ports of the memory block, there is a requirement of diverting the data in each cycle, on the two ports, to the right processor in the right cycle. To achieve this, two connecting elements which act like circular shift switches are used in the datapath. We discuss these switches in section 10.2.

10.1 Tanner Graph Generation

The Tanner graph in for our purpose is same as the line-point incidence graph of a *projective plane* (dimension 2). In general, a projective space of dimension **n** over $\mathbb{GF}(2)$, $\mathbb{P}(n, \mathbb{GF}(q))$, has at least following two properties, arising out of inherent duality:

1. The number of subspaces of dimension m is equal to the number of subspaces of dimension $n - m - 1$.

2. The number of m-dimensional subspaces incident on each $n - m - 1$-dimensional subspace is equal to the number of $n - m - 1$-dimensional subspaces incident on each m-dimensional subspace.

Hence not only the number of bit nodes and number of check nodes is equal for our Tanner graph, but their incidence degrees to the nodes of other kind are also equal. As mentioned earlier, we associate each *point* of the projective plane to *bit nodes*, while we associate each *line* to *check nodes*. Two nodes are connected by an edge, if the corresponding line(point) contains(is contained in) the corresponding point. Hence the Tanner graph is essentially a *balanced, regular bipartite graph*.

The required projective space is generated from $\mathbb{GF}(2^3)$. In this projective space, the number of points (= number of hyperplanes) is 73. Each point is incident on 9 lines, and

vice-versa. The bipartite graph is constructed by using the point-line incidence relations of $\mathbb{P}(2, \mathbb{GF}(2^3))$. The points are generated using a primitive polynomial, which give the tapping points in a linear shift feedback register(LFSR). The primitive polynomial used to generate $\mathbb{GF}(2^3)$ is $x^3 + x^2 + 1$. To give each point an index, which is in form of a polynomial, we perform a *polynomial hashing* to generate a bitstring representation of each point. This hashing is essentially a bijection between polynomial coefficients and bit placeholders in the bitstring [21]. To identify the points lying on a line, we do a reverse construction. We take 2 points, and call it a particular line(any 2 points make up a line in projective plane). We identify the remaining 7 point by taking linear combination, modulo 2^3, of the polynomial reprsentation of these two points, with coefficients for linear combination varying from (1,1) to (7,7). (0,0) is not a valid combination of the two points. Any duplicate points(or polynomials) that are found are dropped out. The set of 9 points is then said to be lying on the particular line. All other lines are then obtained by applying *Shift automorphism* to the point set of this first line [11]. Table 4 gives the complete list of the lines and their adjacent points, a summary of which was presented earlier in section 9.3. To have a possible layout regularity, we introduce symmetry in the form that the edges incident on a vertex of side **A**(say, $V1$) are sorted with respect to increasing index numbers of the vertices reached in side **B**. This is the order in which the corresponding update messages are fed to the processing unit represented by $V1$. A similar strategy is used for ordering inputs for processing unit vertices on side **B**.

Table 4: Point-Line Adjacency List

Line	**Adjacent Points**
0	0, 1, 11, 20, 38, 43, 59, 67, 71
1	1, 2, 12, 21, 39, 44, 60, 68, 72
2	2, 3, 13, 22, 40, 45, 61, 69, 0
3	3, 4, 14, 23, 41, 46, 62, 70, 1
4	4, 5, 15, 24, 42, 47, 63, 71, 2
5	5, 6, 16, 25, 43, 48, 64, 72, 3
6	6, 7, 17, 26, 44, 49, 65, 0, 4
7	7, 8, 18, 27, 45, 50, 66, 1, 5
8	8, 9, 19, 28, 46, 51, 67, 2, 6
9	9, 10, 20, 29, 47, 52, 68, 3, 7
10	10, 11, 21, 30, 48, 53, 69, 4, 8
11	11, 12, 22, 31, 49, 54, 70, 5, 9
	Continued on next page

Table 4 – continued from previous page

Line	Adjacent Points
12	12, 13, 23, 32, 50, 55, 71, 6, 10
13	13, 14, 24, 33, 51, 56, 72, 7, 11
14	14, 15, 25, 34, 52, 57, 0, 8, 12
15	15, 16, 26, 35, 53, 58, 1, 9, 13
16	16, 17, 27, 36, 54, 59, 2, 10, 14
17	17, 18, 28, 37, 55, 60, 3, 11, 15
18	18, 19, 29, 38, 56, 61, 4, 12, 16
19	19, 20, 30, 39, 57, 62, 5, 13, 17
20	20, 21, 31, 40, 58, 63, 6, 14, 18
21	21, 22, 32, 41, 59, 64, 7, 15, 19
22	22, 23, 33, 42, 60, 65, 8, 16, 20
23	23, 24, 34, 43, 61, 66, 9, 17, 21
24	24, 25, 35, 44, 62, 67, 10, 18, 22
25	25, 26, 36, 45, 63, 68, 11, 19, 23
26	26, 27, 37, 46, 64, 69, 12, 20, 24
27	27, 28, 38, 47, 65, 70, 13, 21, 25
28	28, 29, 39, 48, 66, 71, 14, 22, 26
29	29, 30, 40, 49, 67, 72, 15, 23, 27
30	30, 31, 41, 50, 68, 0, 16, 24, 28
31	31, 32, 42, 51, 69, 1, 17, 25, 29
32	32, 33, 43, 52, 70, 2, 18, 26, 30
33	33, 34, 44, 53, 71, 3, 19, 27, 31
34	34, 35, 45, 54, 72, 4, 20, 28, 32
35	35, 36, 46, 55, 0, 5, 21, 29, 33
36	36, 37, 47, 56, 1, 6, 22, 30, 34
37	37, 38, 48, 57, 2, 7, 23, 31, 35
38	38, 39, 49, 58, 3, 8, 24, 32, 36
39	39, 40, 50, 59, 4, 9, 25, 33, 37
40	40, 41, 51, 60, 5, 10, 26, 34, 38
41	41, 42, 52, 61, 6, 11, 27, 35, 39
42	42, 43, 53, 62, 7, 12, 28, 36, 40
43	43, 44, 54, 63, 8, 13, 29, 37, 41
44	44, 45, 55, 64, 9, 14, 30, 38, 42

Continued on next page

Table 4 – continued from previous page

Line	Adjacent Points
45	45, 46, 56, 65, 10, 15, 31, 39, 43
46	46, 47, 57, 66, 11, 16, 32, 40, 44
47	47, 48, 58, 67, 12, 17, 33, 41, 45
48	48, 49, 59, 68, 13, 18, 34, 42, 46
49	49, 50, 60, 69, 14, 19, 35, 43, 47
50	50, 51, 61, 70, 15, 20, 36, 44, 48
51	51, 52, 62, 71, 16, 21, 37, 45, 49
52	52, 53, 63, 72, 17, 22, 38, 46, 50
53	53, 54, 64, 0, 18, 23, 39, 47, 51
54	54, 55, 65, 1, 19, 24, 40, 48, 52
55	55, 56, 66, 2, 20, 25, 41, 49, 53
56	56, 57, 67, 3, 21, 26, 42, 50, 54
57	57, 58, 68, 4, 22, 27, 43, 51, 55
58	58, 59, 69, 5, 23, 28, 44, 52, 56
59	59, 60, 70, 6, 24, 29, 45, 53, 57
60	60, 61, 71, 7, 25, 30, 46, 54, 58
61	61, 62, 72, 8, 26, 31, 47, 55, 59
62	62, 63, 0, 9, 27, 32, 48, 56, 60
63	63, 64, 1, 10, 28, 33, 49, 57, 61
64	64, 65, 2, 11, 29, 34, 50, 58, 62
65	65, 66, 3, 12, 30, 35, 51, 59, 63
66	66, 67, 4, 13, 31, 36, 52, 60, 64
67	67, 68, 5, 14, 32, 37, 53, 61, 65
68	68, 69, 6, 15, 33, 38, 54, 62, 66
69	69, 70, 7, 16, 34, 39, 55, 63, 67
70	70, 71, 8, 17, 35, 40, 56, 64, 68
71	71, 72, 9, 18, 36, 41, 57, 65, 69
72	72, 0, 10, 19, 37, 42, 58, 66, 70

10.2 Circular Shift switches

As pointed out earlier, two connecting elements which act like circular shift switches are used in the datapath to achieve 2-to-9, and 9-to-2 port multiplexing. The first one is called the memory switch, while the second one is called the processor switch.

10.2.1 Memory switch

Each of the 2 (data) output ports of each memory block is connected to a **1:5 circular shift** switch having 5 different outputs. These outputs are in the **same order**, as the incidence set of the point/line that the memory block represents. Hence there are 2 memory switches adjacent to each memory block, interfaced to 1 port each of the memory block. The role of each memory block is to place the data being read from the memory block, on certain wires that realize the Tanner graph interconnect, in a *circular manner*. Figure 17 shows the design of a memory switch interface to port A of the bit memory # 0, along with the connections.

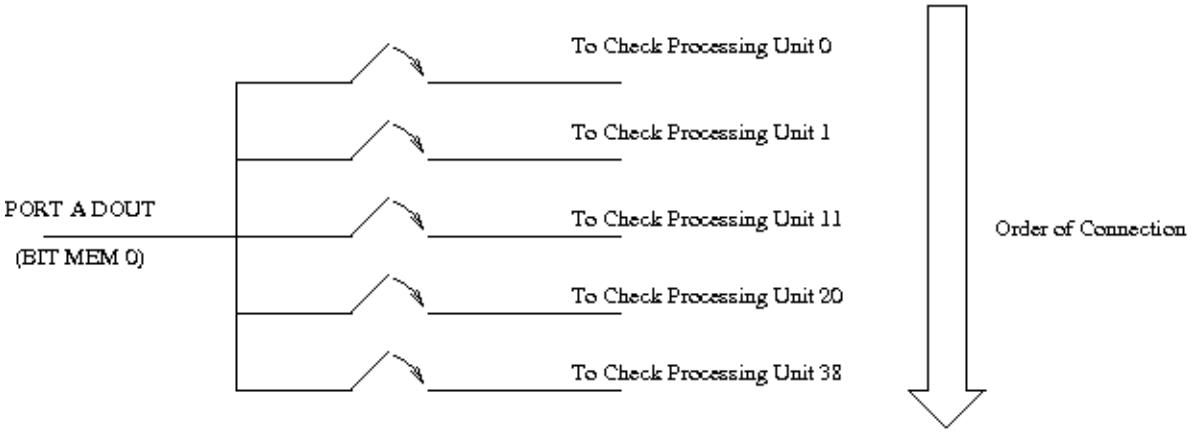

Figure 17: 2-to-9 Memory Switch

On every clock rising edge, the data on the input of switch is placed on to one of its outputs. The order of the outputs that it is placed on goes from output # 1 to output #5 and back to output # 1. Thus, in the first cycle, output # 1 gets the input data, in the next cycle output # 2 and so on. As the input data is continuously changing due to the read address of the block RAM port, different outputs(interconnect wires) get different data in the different cycles.

10.2.2 Processor switch

Since every processing unit gets data from 9 memory blocks in total, but can take in only 2 inputs per clock cycle, a 9-to-2 circular switch is required at the processing unit end too. The role of this circular shift switch to select the right data from all the incoming wires. Thus there is a 5:1 switch interface to each of the two input ports of each of the processing unit. This switch has 5 inputs and 1 output, which feeds one of the input ports of the processing unit. The switch circularly connects its inputs to the output on every clock edge, very much mirroring the function of a memory switch. Thus, the output would be connected to input # 1, input # 2..,input # 5 and back to input # 1 in that order over successive cycles. The processor switch for the second input of check processing unit # 0 is shown in figure 18.

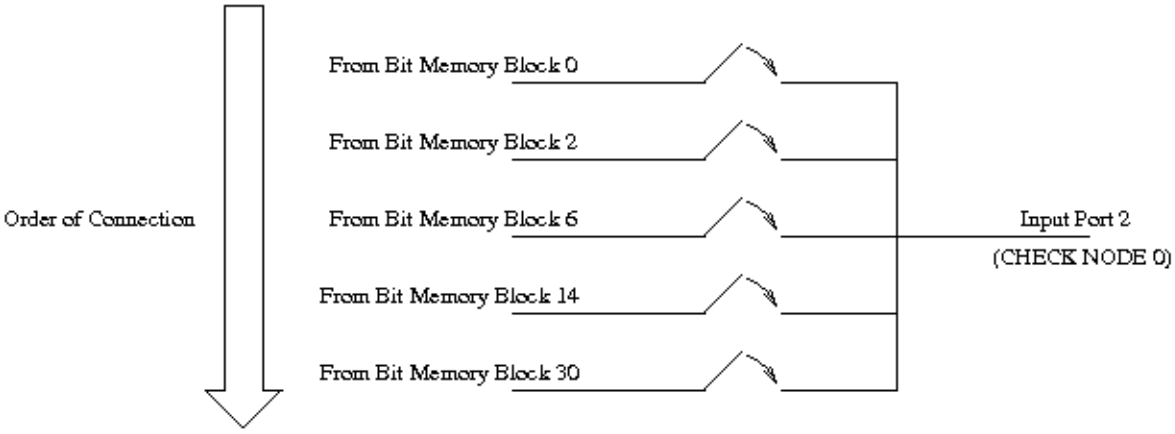

Figure 18: 9-to-2 Processor Switch

10.3 Detail Design of Interconnect between Switches

The connections between the two types of switches are made such that *perfect access patterns get exercised* during each cycle in which memory reads are being performed. For example, the (memory) switch of a bit memory block is connected to the processor switches of the check processing units that are going to read from that particular block, in the order of a perfect access pattern. As discussed earlier in section 9.3, bit memory block 0 will be accessed by check processing units corresponding to the lines in its incidence set.

Point 0 : {*0,1,11,20,38,43,59,67,71*}

Thus, the memory switch at port A is connected to the switches of processing units 0, 1, 11, 20 and 38, while that at port B to the switches of processing units 43, 59, 67 and 71. The last output of the port B switches is unconnected, or left *open*(there's no valid 10^{th} input data per processing unit). The order of the connections is maintained as per the *order from left to right* in the incidence set. Hence, the 1^{st} memory switch of bit memory block 0 passes on the data on 1^{st} output port of memory block, to lines {0,11,38,59,71} in 5 consecutive cycles, while the second switch passes on the data on 2^{nd} output port of memory block, to lines {1,20,43,67} in 4 consecutive cycles(fifth switching is ignored). Thus, the processor switch of check processing unit 0 is connected to output #1 of the memory switch, that of check processing unit 11 to output # 2 and so on. Data from the bit memory is placed on inputs of these memory switches in the same order.

Likewise, the inputs of the processor switches are connected in the order of the incidence set of the line that the processing unit represents.

Line 0: {0,2,6,14,30,35,53,62,72}

Hence connections from memory switches of the bit memory blocks 0, 6, 30, 53 and 72 are the inputs to switch **A** of the processing unit, while that from memory blocks 2,14,35 and 62 are inputs to the switch **B**, in that order. Data input to the processing unit is also selected in the same order. The last input to switch **B** is hardwired to 000000000. This is to insulate the value of the total sum in the accumulation scan from the invalid 10^{th} input, which may be randomly anything(an open connection).

Thus, in the first read access cycle, data from the first two locations (corresponding to the first two lines in the incidence set) of the bit memory is placed at its output ports. It is then transimitted to the first(out of 5) processor switches interfaced to each memory switch. The processor switches then switch over the data on their input ports, to the input ports of corresponding processing units. The order of the incidence set is followed strictly for every cycle, at both the switches. Hence there are no access collisions throughout the read cycle. It thus takes 2 clock cycles for the data, after it is stable on the memory block output, to reach the processing units. The address generation in the memory block is synchronized to ensure that the data from the right memory location is present on its port, before it is connected to a processor by enabling two shift switches.

The switches are provided with a control signal to tri-state the network when processors are not reading from memories.

11 Overall Datapath

The overall datapath starts at intrinsic latches, goes through an array of bit and check processing units, and stops after a check on the syndrome vector. While doing syndrome checking, we use speculative scheduling to boost the throughput. While the XOR data-subpath calculates the syndrome, we allow bit nodes to proceed with computation of message for next iteration, because they will be doing so in all but one(last) iteration.

LDPC decoders generally need clock and intrinsic data as inputs, and decoded codeword as their output. In our design, intrinsic data is loaded into latch by the decoder. The codeword is placed in a output register.

12 Control Path Architecture

For a parallel processing system like LDPC decoder, control signals need to be applied to multiple datapath elements simultaneously. Since we follow a flooding decoding schedule, multiple control signals required to drive multiple datapath elements in a particular cycle have the **same semantics** in general. Hence a compact, centralized controller is more hardware-efficient than a distributed controller, for generation of control signals, at the cost of global routing(higher wire delays), and high fan-outs. Co-ordination for the entire data flow along the datapath is done by the centralized controller by asserting the control signals in the right cycle, since the design is a synchronous design.

This centralized control unit is implemented using an architectural template called **microprogramming**, to achieve design flexibility and scalability. Microcodes are programmed into a ROM, called as **controller ROM**, acting as control store. The control signals are generated through low level instructions, analogous to microcodes. The microcodes are essentially *control bit-vectors*, each bit of which is mapped to a control signal used to control elements of the decoder datapath. Each control vector hence aids in execution of certain datapath elements in a particular cycle. A sequence of control vector thus sequences the execution of the complete datapath itself. Since in our design, the sequencing of datapath is identical across iterations, we limit the control path design to evolution of a sequence of microcodes signifying the executions in a sequence of machine cycles for one iteration only. When the current control vector is read from the ROM, the control signals in various parts of the design get asserted/deasserted depending on the bit pattern of the vector. Accordingly, the datapath executes some part of the current decoding step. To understand how the microcodes are evolved at design time, we have to take a look at the Moore machine of the decoder. Since the circuit is synchronous,

the state of the circuit is advanced by the clock input. The next state is determined by the current state, characteristic of a Moore machine.

12.1 Decoder State Machine

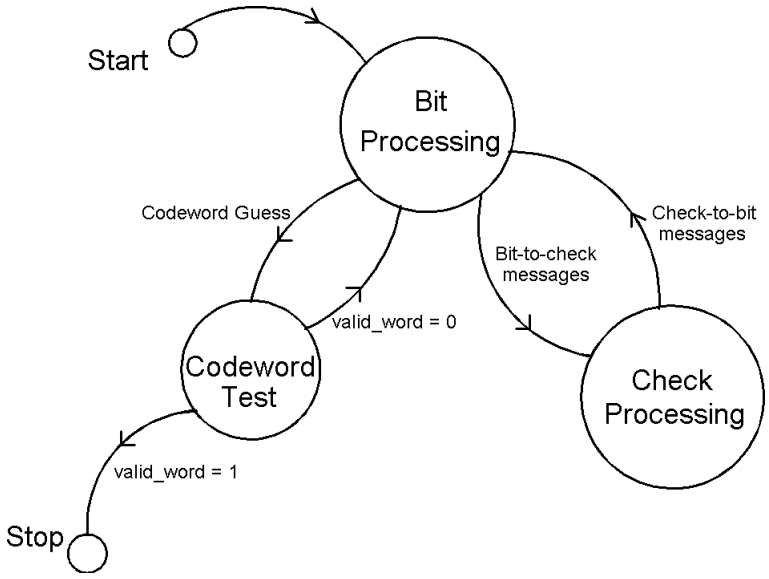

Figure 19: Finite State Machine of the decoder

There are three distinct states of computation: the bit processing,the check processing and the codeword test. Codeword test phase executes in parallel with the bit processing phase. The data exchanged between all the states is through the memories. The **START** state of the decoder is used to initialize the contents of all the memories in the design by reading in the input intrinsic values, and flushing the memory block contents. The process then enters the **bit processing** state. The **bit** and **check processing** states execute one after the other in an *iterative manner*, until the codeword test, which happens in parallel with the bit processing, returns a **true** value on the *valid_word* signal. Following this assertion, the decoding process transits into the **STOP**. This state signfies the finding of a valid codeword.

The computational steps are all synchronous on the rising clock edge. In few cases, the inputs are registered on the negative clock edge due to clock-independent latching/multiplexing of the inputs. But, since the setup and hold time constraints are fol-

lowed by the synthesis, inputs are never invalid before they are used for computations or for being written into memories. The enable signal of most computational blocks, when de-asserted, leds to tristating of their outputs. Address generation units also have control signals to start and stop the address generation.

12.2 Control Store

The RAM used to implement the control store is a single port RAM. It is 38 bits wide and 42 locations deep. The width 38 has been decided according to the total number of control signals in the decoder, plus a few additional bits to provide space for any future expansions. The depth of control store is equal to the number of clock cycles needed to complete one iteration, which includes execution of both bit nodes and check nodes. The first 19 bits of the control vector are dedicated to the check-side control signals, while the last 17 are dedicated to the bit side control. The complete sequence of microcodes for an iteration is loaded as an initialization file (*.coe file) for the control ROM.

12.3 Microcode Sequencer

The controller ROM needs an address generator to be able to write out control vector on its output port. The address generator is called a microcode sequencer. In our case, the addressing needed is sequential due to lack of branching. Hence the microsequencer is simply a counter acting on the positive clock edge. For the decoder, there is only one conditional execution: that of decoder's exit. This is handled as an exception outside the microsequencer. After each iteration, which is of 42 cycles, the microcode sequencer keeps coming back to 0^{th} microcode to help run the next iteration.

Whenever the seuqencer places an address on the address bus of controller ROM, a control vector from corresponding location is read out in the **next** cycle. The individual bits are then distributed as control signals throughout the design, which can then drive various datapath elements in their **next** cycle. The control vectors are thankfully predetermined due to the completely deterministic data flow.

The signal *valid_word* is the *reset* signal for the address generator of the control RAM. As soon as the signal goes high, indicating that a valid codeword has been found, the address generator stops generating the addresses, thus stopping the load and the decoding of microcodes.

In the simulation of the design, it was seen that there is a small but finite delay between the clock edge and the transition of the control signals in the datapath. This is there

due to the read access latency of the RAM. Hence the effecting of a control cycle takes place 1 cycles after the corresponding control vector has been read.

12.4 Design Schedule

The design schedule is the sequence of computations that is done within the **bit update** and **check update** states, iteratively, preceeded by any computations that are done in **START** state(such as latching intrinsic information), and succeeded by any computations that are done in **STOP** state. The **bit update** phase encapsulates a micro-level FSM of bit node computations, and similarly does **check update** state. These underlying micro-level FSM in both bit and check processing units capture the timing constraints for individual components of theirs. For example, in bit node, the write-enable for the ports of the bit memory blocks asserted only after(at next clock edge) the inputs at the ports become stable. Control signals are provided in parallel to all nodes of one kind for concurrent computation. Both the clock edges are used within bit nodes and check processing units, to expedite computation. The entire design is thus a synchronous design.

The schedule of an entire iteration is given in appendix A, tables 7, 8, 9, 10, 11 and 12. The details of this schedule are also explained in the same appendix.

13 Implementation Overview

The decoder was written in VHDL with a structural design hierarchy and the primitive units described behaviorally. It is targeted for the Xilinx Virtex 5 LX330T FPGA. As such, the computational resources even on the largest of FPGA are not sufficient to accommodate thousands of bit and check nodes required for parallel decoding of large practical LDPC codes. Hence the design that we could fit in has a size of PG(73), based on Galois field $\mathbb{GF}(2^3)$ [12]. The tools that were used were ISE 10.1 for IDE, Synplify Pro D-2009.12 for Synthesis and Modelsim 6.2 for simulation. Coregen 8.1 was used to design all the RAM/ROM cores.

14 Synthesis Results

Based on above implementation, we synthesized a decoder of length 73 and regularity 9 targeted to Xilinx Virtex 5 LX330T, having code rate of $\frac{45}{73}$. The resource utilization

statistics are shown in table 14. Since usage of DSP slice is replacable by custom logic, and pin count reducable by using a ROM to store intrinsics, it is evident that memory is the most critical resource in our implementation. The pre-routing maximum frequency was found to be around 130 MHz. The frequency did not change when we changed the length of the code to a lower value, 57, or the board within Virtex 5 family, such as SX240T. The critical path was located to run through the check node in this implementation. So the check nodes, specifically the DSP block within, was pipelined using one register. The maximum frequency increased to 155 MHz as a result of this change. The number of cycles per iterations are remained at 42 due to microprogramming-based control path design. The system throughput achieved at the non-pipelined frequency at practical SNRs(> 2) is ≥ 89 Mbps(see equation 21. This throughput will definitely get boosted up by further careful pipelining of overall design, and retiming the circuit. This throughput is in fact, sufficient for the requirements of WiMAX(75 Mbps) and DVB-S2(90 Mbps) [5]. Since the throughput increases at least linearly with code length, if we were to somehow fit a length-1057 PG LDPC decoder on some FPGA, then analysis shows that such a scaled design would have had a throughput of 1.06 Gbps, comparable to that achieved by well-known fully parallel ASIC design [1].

Table 5: Resource Utilization for Virtex 5 LX330T

Resource		Number	% Utilization
CLB Slices	6-input LUTs	35405	17%
	Flip-flops	25842	12.5%
DSP Slices(Multipliers)		146	76%
BRAMs		147	45.3%
Bonded IOBs		764	79.5%

14.1 Throughput Calculation

The throughput of the decoder system can be calculated as

$$T = \frac{73 \times f_{clk}}{42 \times n_{avg}} \text{mega symbols per second} \tag{21}$$

Here, f_{clk} is the maximum clock frequency, in MHz, obtained from synthesis of the circuit and n_{avg} is the average no. of iterations required to decode the code. Also recall that

42 is the no. of clock cycles taken for one iteration run and 73 is the length of the block decoded.

If a complete black box view of the system is taken, then another figure for the throughput, which has practically very less significance, can be calculated as

$$T = \frac{73 \times 9 \times f_{clk}}{42 \times n_{avg}} \text{mega symbols per second} \tag{22}$$

given that each input symbol is repesented as 9-bit fixed point data element at the input of the decoder system.

14.2 Experiments with Synthesis

During Synthesis with Synplify Pro, one option that was tried out was (automatic) LUT combining. During incremental synthesis of the system, it was observed that at times, even an inverter was mapped to a 6-input LUT on FPGA, which tantamounts to wastage of compuatational power of the basic cell of Virtex-5 FPGA, the 6-input LUT. A bigger sized combinational circuit can perhaps be packed into a 6-input LUT. However, after synthesis along with this option, the improvement in total number of LUTs/CLBs used came down by just 2-3%.

(Automatic) Retiming the design, so as to increase the maximum clock frequency achievable by the system, also did not show signficant improvement. As a tradeoff, due to buffer placements along the datapath, the number of LUTs used went up by a small factor.

(Automatic) Pipelining was another option, which is yet to be tried out.

To reduce the load of high-fanout components, a parameter called "fanout guide" was experiemented with. To reduce the maximum number of loads on the output of any cell, *timing-driven replication* is performed of the output driver. This leads to insertion of buffers, and hence to increase of number of LUTs used.

15 Simulation and Testing

Two kinds of off-board simulations were performed: a functional and a performance simulation. On-board performance simulation has been planned.

15.1 Functional Simulation

The idea behind functional simulation at behavioral level was to weed out any implementation errors that would have crept in, plus any design error which would have happened due to oversight. For this, a complete iteration of the decoder was simulated, cycle-by-cycle, and the value of various signals at intermediate points in datapath were tallied with hand calculations.

The starting point of such simulation was achieved by writing values 1_{10} through 10_{10} in first 10 location of each bit memory block at the initialization time(through an init file). This led to **non-trivial** execution of all datapath elements within the bit node and the check node. Further, to have non-trivial value vector as well, to check the codeword testing branch's proper functioning, value vectors were hard-coded to 9b'1(negative values) within the hardware model.

A lot many errors were thus located and fixed, which helped in saving time during further usage of the system. The entire simulation was carried out within the realm of Modelsim tool, without any further test setup being required.

15.2 Simulation for Performance Benchmarking

Modelsim simulations were done to detect the convergence and *Bit Error Rate* of decoding at various SNRs for measuring performance. The corresponding test strategy is described in the next section. Figures 20 and 21 show the decoding performance in terms of BER as well as average no. of iterations, assuming an AWGN channel and BPSK modulation scheme. These measurements were very close to the measurements done on the MATLAB model developed for the same decoder. Both the curves follow standard models. PG codes converge very fast under SPA decoding [12], and same seems to be true for log-SPA decoding as well. This is because given the medium code rates of PG codes, there are more parity checks involved per bit, and hence reliability updates converge fast. The BER curve shows all the three regions, though lack of precision due to shorter length of code makes it look more stretched out, and a higher error floor. The usefulness of these codes under various channel conditions is already established in [12].

15.2.1 Test Strategy

First, the generator matrix $\mathbb{G}$ of size 45×73 was generated from parity check matrix $\mathbb{H}$. For that, *Gauss-Jordan Elimination* was used, and any null rows found during the

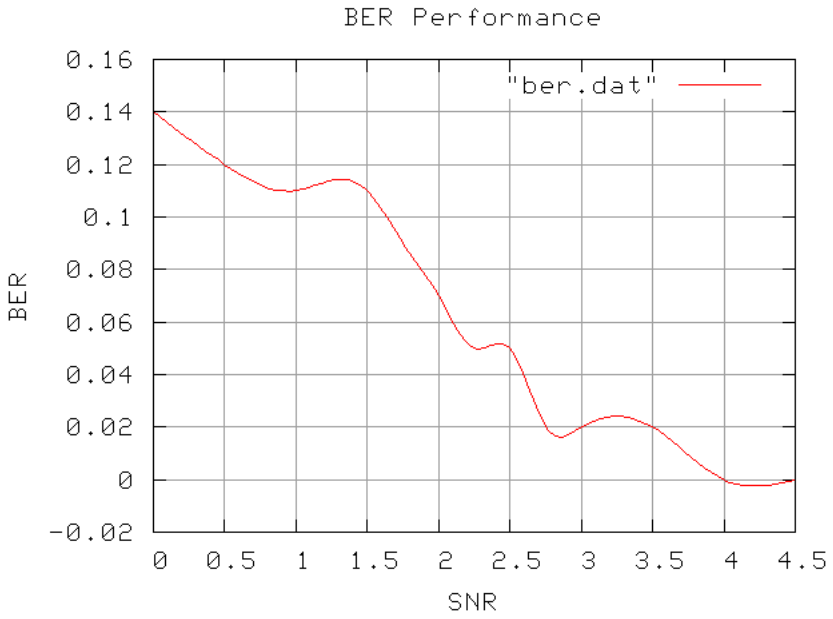

Figure 20: Transmission BER Performance

process were dropped. The generator matrix needs to be systematic(in $[\mathbf{I}\ \mathbf{K}^T]$ form), so that after decoding, the last 73-45=28 bits can be dropped. Else, the locations of parity bits will be smeared throughout, and the decoder will then need to use this location information additionally citekarkooti. Multiple(right now 10) 45-bit **pseudo-random binary inputs** were generated and encoded with $\mathbb{G}$, thus generating the 73×1 codeword vector. Binary antipodal modulation was performed on each bit of the codeword, using the mapping $\{'0' \rightarrow +1, '1' \rightarrow -1\}$. This leads to generation of the *transmitted bitstring*. With the modulation signal's amplitude normalized(± 1), the energy per bit of encoded, modulated signal is $E_s = 1(A^2$ in general).

The transmitted bitstring was then smeared with additive white gaussian noise(AWGN) to create the effect of passing through the channel. The *power spectral density* of AWGN was chosen so that a range of SNRs could be acheived for the received signal(a sequence of *real-valued* samples). Hence the AWGN process model has a mean of 0, and a variance of σ, where σ is to be varied downward in the set $\{0.84, 0.81, 0.78, 0.73, 0.68, 0.63, 0.57\}$. This corresponds to varying the SNR($\frac{E_b}{N_0}$) of received signal between 1.5 and 5, in dB. Now $E_b = E_s \times R$, where R is the *code rate*, and power spectral density $N_0 = \frac{\sigma^2}{2}$. Given

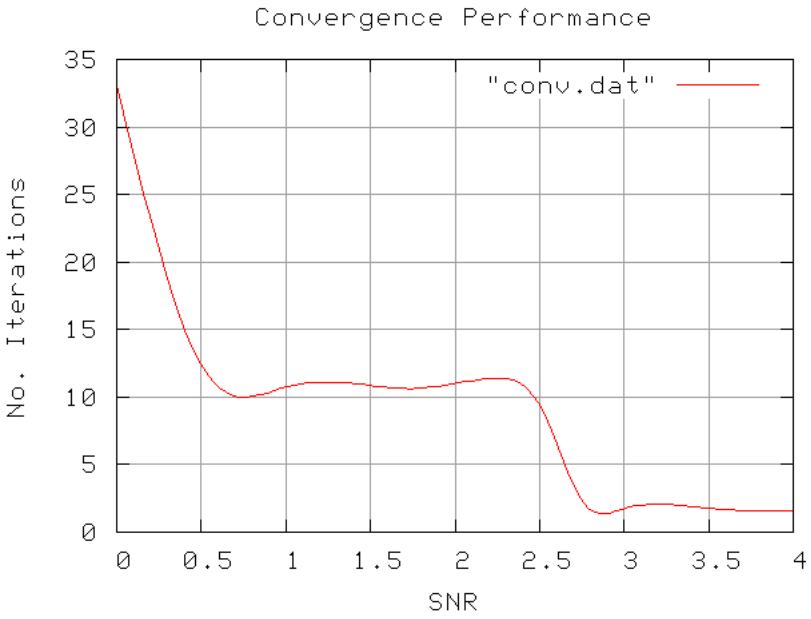

Figure 21: Convergence Performance

that encoded signal's power, E_s is 1, $\frac{E_b}{N_0} = \frac{2*R}{\sigma^2}$. Further, $10 \log_{10} \frac{1}{\sigma^2} = -20 \log_{10} \sigma$. By varying σ within the set $\{0.84, 0.81, 0.78, 0.73, 0.68, 0.63, 0.57\}$, we get the **SNR** set, or the *observation points* as $\{1.5, 1.83, 2.15, 2.73, 3.35, 4.01, 4.88\}$.

Each such sequence of 73 real samples, representing a channel measurement for a particular SNR, was first converted into sequence of intrinsic information, as per equation 16. This sequence of real-value intrinsic information was then digitized using *9-bit fixed point representation*. A collection of 73 such 9-bit inputs, acting as intrinsic information, was provided to the decoder system via the testbench. The testbench would then trigger off the decoding process. The decoder was made to stop at 50 iterations, and only at low SNRs, the decoder would sometimes hit this threshold. The *transmission* BER was measured by computing the distance between the 73 bit decoder's output(the final **value vector**), and the transmitted codeword.

At each SNR, 10 input sequences, each one a sequence of 73 9-bit data, were successively generated, and provided to decoder for decoding. The convergence and BER performance, *as measured at a particular SNR*, would then be the average of these 10 trials.

15.2.2 Test Setup

The entire process of 73 9-bit input(intrinsic information) generation is done by a matlab script. For a particular SNR, which is a constant provided to this script, it generates a set of 10 files, decoder_input$_i$, storing 10 different sequences of 73 9-bit binary inputs. These are the sequences of intrinsic information. The script also generates 10 files storing the trasmitted codeword(tx_codeword$_i$), which is to be used later to calculate the decoded codeword's distance from transmitted codeword.

The system test bench consists of a clock generator, a file i/o process to read in the intrinsic information from file, and another process to write the final value vector into a file called decoder_output. The testbench also writes into the same file, the number of iterations that were taken by the decoder for convergence. This testbench can not be run in a loop for multiple inputs.

Hence, a shell script, run.sh, has been created to run the decoder for simulation, without any interruption or manual intervention, 10 times in a stretch, using the decoder_input$_i$ series of files for intrinsic information. This script generates decoder_output$_i$ series of files. The idea here is to be able to do a single-run test for multiple inputs, per fixed SNR. This script needs to be run from a machine, on which **UNISIM** and **XilinxCoreLib** are installed as part of Modelsim [24].

Another shell script, analyse.sh, has also been provided to do post-processing, such as calculation of BER, given the set of decoder_output$_i$ files.

15.3 On-board Testing

Porting of the decoder implementation on Xilinx Virtex 5 LX110T is ongoing. Measurement of real performance such as maximum system frequency etc. will be done once this porting is over.

16 FPGA Implementation

16.1 Choice of design platform

Most of the applications that use LDPC codes have code lengths running into thousands of bits. The fastest decoding for LDPC codes is obtained through fully parallel decoding. Since the SP decoding algorithm is inherently parallel, the platform on which the design is

placed should be suited to accommodate and enhance parallel computing. Fully parallel LDPC decoders have benefit of higher throughput and power efficiency, but require the implementation of a large number(thousands) of concurrent processing elements, together with message passing within a congested routing network.

A practical implementation of the algorithm in either hardware or software is generally optimized for **silicon area, power, throughput, latency, flexibility of implementation, scalability etc**. While microprocessors and DSP blocks provide the most flexibility, complex computations can take multiple execution time units. Field programmable gate arrays(FPGAs) offer more opportunities for parallelism, but with reduced flexibility. FPGAs are intended for datapath-intensive designs, and thus have an interconnect grid optimized for local routing. Custom ASICs are well suited for direct mapped architectures, offering even higher performance with further reduction in flexibility. ASICs can, in fact, be able to meet the requirement of fitting thousands of processing units, but accompanying the VLSI implementation of a decoder are issues with interconnects like crosstalk, and long-distance routing problems. A first ASIC implementation of a fully parallel decoder shows that more than half the density on the ASIC is occupied by the interconnects [1].

Due to the structured nature of the node connections, FPGA fabric is well suited for the sparse interconnect networks for most PG LDPC codes. However, as pointed out earlier, the computational resources even on the largest of FPGA's are not sufficient to accomodate thousands of bit and check processing units required for parallel decoding of large practical LDPC codes. Even with this handicap, FPGAs still remain suitable for doing rapid prototyping, and iterative improvement of a system design. Hence we have chosen to implement a FPGA-based design to test and study the working of the concurrent flows of perfect access for LDPC decoding.

16.2 Components of the FPGA design

On Xilinx LX330T, we have been able to synthesize and implement a length-73 PG LDPC code's decoder. The degree of each node in its balanced, symmetric bipartite Tanner graph is 9. The decoding system contains of four main components.

1. **Processing Units**: They are the computational blocks that process the messages passed along the edges of the Tanner graph(bit and the check updates). In addition, the check processing units also perform the codeword test during each iteration of the algorithm.

2. **Memory Blocks**: Memory blocks are used to store data required between iterations. The size of the memory block required is dependent on the total number of edges in the Tanner graph of the code. Also, the precision of the data representing the messages directly affects the size of the memory blocks. The choice of the precision is discussed later. In the case of a structured graph like for the subspaces of a Projective Geometry, one can follow structured access patterns for memory access.

3. **Control Logic**: For parallel processing designs, the control logic is spread over a large no. of partitions. At times, there may be multiple instances of a control block, each for a localized area. Due to resource constraints for large designs on FPGA's, however, it is suitable to centralize the dispatch of control signals through wires to different parts of the datapath.

4. **Interconnects**: A medium dense network of interconnects results due to the incidence of the subspaces of the geometry. During placement and routing of the design,the interconnect length may not be optimized for all signals. So,the user may have to place and route under defined minimum local constraints. We shall take a look at this under the topic of synthesis.

16.3 Data representation

The choice of message precision is dictated by the **bit error rate** and the **resources of storage and computation available on the FPGA** [25]. The data used to represent the messages is in **9-bit fixed point** format, with the MSB representing the sign and the rest 8 bits for the magnitude. In the magnitude part, the most significant 3 bits are for the integer part, and the remaining 5 bits represent the fractional part of data. The table below shows few examples of numbers in decimal representation, and equivalent 9-bit fixed point representation.

Fixed point notation	Decimal equivalent
000011000	0.75
100001101	-0.40625
011000100	6.125
101111000	-3.75

Table 6: Fixed point notation of data

Hence the maximum magnitude that can be represented using this notation is 7.96875. However, the internal datapaths of the node processing units contain additional bits for allow *intermediate overflows* due to accumulations. At the communication frontiers, i.e. between processing units and memory blocks, only 9-bit representation is used.

16.4 Implementation Details

The functionality of each bit node is mapped to CLBs and BRAMs. BRAMs were also used to map the storage for intrinsic data. The entire datapath of bit nodes is otherwise mapped to CLBs, requiring 26 CLBs. Similarly, the functionality of each check node is mapped to CLBs and DSP slices. 2 DSP48E slices are used per check node to realize the multiply-add operations involved in $\phi(x)$ function. We set the alumode of the slice so that it performs a MAC operation, feeding from A, B and C inputs of the slice. The datapath involving DSP slices carries the magnitude part and has positive data, so the inputs are zero-padded before being fed in at A, B and C ports. The decimal point at output is adjusted dynamically to emulate fixed-point multiplication, by truncating few bits at the right of the output. The remaining datapath is otherwise mapped to CLBs, requiring 44 CLBs. The bit and check memory blocks are mapped onto BRAMs. They are configured to be true dual port memories. To avoid routing congestion on global routes, we have implemented a 50% reduction in global wires by time-multiplexing the two instances of PG interconnect: one between bit memory blocks and check nodes, and the other between check memory blocks and bit nodes. This mapping also made the synthesis process faster, discussed earlier.

17 Future Work

The design is far from complete. Some of the design changes that are required to be done are as follows.

1. Elimination of Coregen-generated BRAM cores. Across myriad of synthesis tools, Coregen-generated netlists are not portable. Also, since we are using 0.03% of BRAM storage space, its quite a wastage. Distributed memory implementation of memory blocks, using flip-flops that are only utilized upto 12.5%(refer table 14), looks clearly feasible on paper.
2. Memory pipelining, if necessary. BRAMs can be pipelined for better latency, in case they are retained.

3. Makind a *dual-data rate*(DDR) design. Even while retaining synchronous control architecture in form of microcode sequencing, one can try to utilize both the clock edges.

4. Right now, at beginning of every iteration, *same* intrinsic ifnormation is being re-latched. This has to be removed. Also, preprocessing of intrinsic information is happening every iteration, which can be avoided by latching data, after doing preprocessing.

5. Removal of bit and check memory initialization. It is via the .coe files. In fact, microcode sequencer needs to be modified so that it does not do the residue accumulation scan in *first iteration* at all.

6. To eliminate res_choice based process from check node. In such case, there can be junk values in most cycles on the check node outputs, res_out[1|2]. However, if we enable enable c_mem for writing in the check node outputs in right cycle, the outputs will contain non-junk data for that much time, which is an acceptable solution.

7. Since the computation within DSP slice is generally under critical path, and the multiplier within DSP slice is not pipelined, we can try implementing our own pipelined-multiplier plus accumulation(PMAC) unit, with a hope to get better overall system frequency.

8. Right now, saturation of magnitude in check processing units is happening just after accumulation scan. If saturation is done after output scan, post reverse phi transformation, where some subtractions take place, then this may cut down loss of precision at times(when the residue and total sum have the same sign).

9. In fact, ports of the adders used in check processing units can also be made of higher width, to further cut down on precision loss.

Some of the synthesis experiments that are remaining are as follows.

1. Synthesis with syn_dspstyle and syn_ramstyle directives

2. Synthesis with coregen netlists presence. So far coregen modules have been treated as black boxes.

3. Slack and Critical path analysis, and subsequent optimization

4. Manual retiming. Automatic retiming is not giving big-ticket advantages.

5. Introduction of asynchronosity in design. Since the pipeline stages *implicit* in synchronous microcode-based control architecture are not balanced, many components are wasting most part of clock period.

6. Doing constrained(UCF-based) synthesis.

At simulation level, following things can be tried out.

1. Check compatibility with algorithm

2. Sanitizing functional simulation against stable decoder model is some other higher-level language such as MATLAB.

To make the design fit and get ported on the locally available Xilinx Virtex-5 LX110T board, at least the following things need to be done.

1. Removing MAC/DSP slices and introducing BRAM LUTs. LX110T has lesser DSP slices than currently present in design.

2. PCI-E interfacing experiment. To provide intrinsic information real-time, this needs to be done.

3. Trying out using multiple clocks. Certain slow blocks such as DSP slices can perhaps be provided faster clocks. Synthesis using multiple clocks is supported by most synthesis tools.

Finally, to make design more scalable and maintenable, following things remain.

1. Making a single configuration file for hardware.

2. Introduction of fixed point configurability using generics. Right now 9-bit representation is being used, but for experiments on convergence, it should be changeable at one place.

3. Merging multiple shift registers into one using generics.

4. Use constants/generics to specify signal widths. Currently they are all hard codes.

The ultimate design change, of course, may be done by redesigning nodes to try out other different soft-decoding algorithms, such as min-sum family of algorithms!

References

[1] Andrew J. Blanksby and Chris J. Howland. Low density parity check code decoder. IEEE Journal of Solid-state Electronics, 37(3):404–412, March 2002.

[2] John Crockett. A hardware implementation of Low-density Parity-check coding for the Digital Video Broadcast satellite version 2 standard. Master's thesis, Utah State University, 2006.

[3] I. B. Djordjevic and B. Vasic. Projective geometry LDPC codes for ultralong-haul WDM high-speed transmission. Photonics Technology Letters, IEEE, 15(5):784–786, 2003.

[4] ETSI. EN 302 307: Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications, April 2005.

[5] Gabriel Falcao Paiva Fernandes, Vitor Manuel Mendes da Silva, Marco Alexandre Cravo Gomes, and Leonel Augusto Pires Seabra de Sousa. Edge stream oriented ldpc decoding. In Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pages 237–244. IEEE Computer Society, 2008.

[6] Robert Gallager. Low Density Parity Check Codes. M.I.T. Press, 1963.

[7] Frederic Guilloud. Generic Architecture for LDPC Codes Decoding. PhD thesis, ENST Paris, 2004.

[8] Yang Han and William E. Ryan. Ldpc coding for magnetic storage: low floor decoding algorithms, system design, and performance analysis. PhD thesis, University of Arizona, Tucson, AZ, USA, 2008.

[9] Sarah J. Johnson and Steven R. Weller. Low-density parity-check codes: Design and decoding, pages 1–18. Wiley Encyclopedia of Telecommunications, jan 2003.

[10] Marjan Karkooti. Semi-Parallel Architectures For Real-Time LDPC Coding. Master's thesis, Rice University, 2004.

[11] Narendra Karmarkar. A New parallel architecture for sparse matrix computation based on finite projective geometries. Proceedings of Supercomputing, 1991.

[12] Y. Kou, Shu Lin, and M. Fossorier. Low-density parity-check codes based on finite geometries: a rediscovery and new results. IEEE Transactions on Information Theory, 47(7):2711–2736, 2001.

[13] Shu Lin. On the Number of Information Symbols in Polynomial Codes. IEEE Transactions on Information Theory, pages 2711–2736, November 1972.

[14] David J.C. MacKay and Radford M. Neal. Near Shannon Limit Performance of Low Density Parity Check Codes. Electronics Letters, 32:1645–1646, 1996.

[15] Mohammad Mansour and Naresh Shanbhag. High throughput ldpc decoders. IEEE Transactions on VLSI Systems, 11(6):976–996, December 2003.

[16] G. Masera, F. Quaglio, and F. Vacca. Finite precision implementation of ldpc decoders. IEE Proceedings - Communications, 152(6):1098–1102, 2005.

[17] Jorge Castineira Moreira and Patrick Guy Farrell. Essentials Of Error-Control Coding. Wiley Interscience, second edition, October 2006.

[18] PACT XPP Technologies. White Paper: IEEE Std 802.16e^{TM} LDPC Decoder on XPP-III, August 2006.

[19] Predrag Radosavljevic, Alexandre de Baynast, and Joseph R. Cavallaro. Optimized Message Passing Schedules for LDPC Decoding. Asilomar Conference on Signals, Systems, and Computers, pages 591–595, November 2005.

[20] William E. Ryan. An Introduction to LDPC Codes, chapter 6.2. CRC Handbook for Coding and Signal Processing for Magnetic Recording Systems. CRC Press, August 2003.

[21] Hrishikesh Sharma. Optimal Projective Space Lattices based Architectures for LDPC Decoding. Technical report, Tata Consultancy Services, India, 2006.

[22] Hua Xiao and Amir H. Banihashemi. Graph-based message-passing schedules for decoding LDPC codes. IEEE Transactions on Communications, 52(12):2098–2105, 2004.

[23] Xilinx, Inc. Virtex-5 FPGA Xtreme DSP User Guide, version 3.3, January 2009.

[24] Xilinx, Inc. Xilinx Synthesis and Simulation Design Guide, version 10.1, April 2009.

[25] Xilinx, Inc. Xilinx Virtex-5 Family Overview, version 5.0, February 2009.

[26] Xilinx LogiCORE. Block Memory Generator Product Specification, version 2.8, September 2008.

[27] Sae young Chung, G. David Forney, Jr., Thomas J. Richardson, and Rdiger Urbanke. On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit. IEEE Communications Letters, 5:58–60, 2001.

[28] Tong Zhang and Keshab Parhi. A 54 Mbps (3,6)-regular FPGA LDPC decoder. IEEE Workshop on Signal Processing Systems, pages 127–132, October 2002.

A Complete Microcode Schedule

A.1 Schedule Details

Table 7: Complete Microcode Schedule

	en_intr_wr	**en_pmux_b**		**cl_add_b**	**en_add_b**	**en_shift_b**	**en_intr_add**
Cycle No.	cvec(39)	cvec(38)	cvec(37)	cvec(36)	cvec(35)	cvec(34)	cvec(33)
1	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0
5	0	1	0	1	0	0	0
6	0	1	0	0	1	1	0
7	0	1	0	0	1	1	0
8	0	1	0	0	1	1	0
9	0	0	0	0	1	1	0
10	0	0	0	0	1	1	0
11	0	0	0	0	0	1	1
12	0	0	0	0	0	1	0
13	0	0	0	0	0	1	0
14	0	0	0	0	0	1	0
15	0	0	0	0	0	1	0
16	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0
36	0	0	0	0	0	0	0

Continued on next page

Table 7 – continued from previous page

	en_intr_wr	en_pmux_b		cl_add_b	en_add_b	en_shift_b	en_intr_add
Cycle No.	cvec(39)	cvec(38)	cvec(37)	cvec(36)	cvec(35)	cvec(34)	cvec(33)
37	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0
39	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0
41	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0

Table 8: Complete Microcode Schedule

	en_codetest	codetest_wr	en_sub_b	en_res_conv	en_sat_b	en_out_b	badd_en
Cycle No.	cvec(32)	cvec(31)	cvec(30)	cvec(29)	cvec(28)	cvec(27)	cvec(26)
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0
12	1	1	1	0	0	1	1
13	0	1	1	1	0	0	1
14	0	0	1	1	1	0	0
15	0	0	1	1	1	1	1
16	0	0	1	1	1	1	1
17	0	0	0	1	1	1	1
18	0	0	0	0	1	1	1
19	0	0	0	0	0	1	1
20	0	0	0	0	0	0	1
21	0	0	0	0	0	0	1
22	0	0	0	0	0	0	1
23	0	0	0	0	0	0	1
24	0	0	0	0	0	0	1
25	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0
36	0	0	0	0	0	0	0
37	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0
39	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0
41	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0

Table 9: Complete Microcode Schedule

	en_wr_bmem_1	en_wr_bmem_2	en_mmux_b		netmux		decide_cword
Cycle No.	cvec(25)	cvec(24)	cvec(23)	cvec(22)	cvec(21)	cvec(20)	cvec(19)
1	0	0	0	0	0	0	0

Continued on next page

Table 9 – continued from previous page

	en_wr_bmem_1	**en_wr_bmem_2**	**en_mmux_b**		**netmux**		**decide_cword**
Cycle No.	cvec(25)	cvec(24)	cvec(23)	cvec(22)	cvec(21)	cvec(20)	cvec(19)
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0
13	1	1	0	0	0	0	0
14	0	0	0	0	0	0	0
15	0	0	1	0	0	0	0
16	1	1	1	0	0	0	0
17	1	1	1	0	0	0	0
18	1	1	1	0	0	0	0
19	1	1	1	0	0	0	0
20	1	1	0	0	0	0	0
21	0	0	0	0	0	0	0
22	0	0	1	0	0	0	1
23	0	0	1	0	0	0	0
24	0	0	1	0	0	0	0
25	0	0	1	0	0	0	0
26	0	0	1	0	0	0	0
27	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0
36	0	0	0	0	0	0	0
37	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0
39	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0
41	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0

Table 10: Complete Microcode Schedule

	cl_add_c	**cl_sign_acc**	**en_pmux_c**		**en_sign_shift**	**en_sign_acc**	**en_sign_res**
Cycle No.	cvec(18)	cvec(17)	cvec(16)	cvec(15)	cvec(14)	cvec(13)	cvec(12)
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0
16	0	1	1	0	0	0	0
17	0	0	1	0	0	1	0
18	0	0	1	0	0	1	0

Continued on next page

Table 10 – continued from previous page

	cl_add_c	cl_sign_acc	en_pmux_c		en_sign_shift	en_sign_acc	en_sign_res
Cycle No.	cvec(18)	cvec(17)	cvec(16)	cvec(15)	cvec(14)	cvec(13)	cvec(12)
19	0	0	1	0	0	1	0
20	0	0	1	0	0	1	0
21	0	0	0	0	0	1	0
22	0	0	0	0	0	0	0
23	0	1	1	0	0	0	0
24	0	0	1	0	1	1	0
25	0	0	1	0	1	1	0
26	1	0	1	0	1	1	0
27	0	0	1	0	1	1	0
28	0	0	0	0	1	1	0
29	0	0	0	0	1	0	0
30	0	0	0	0	1	0	0
31	0	0	0	0	1	0	0
32	0	0	0	0	1	0	0
33	0	0	0	0	1	0	0
34	0	0	0	0	1	0	0
35	0	0	0	0	1	0	0
36	0	0	0	0	1	0	0
37	0	0	0	0	1	0	1
38	0	0	0	0	1	0	1
39	0	0	0	0	1	0	1
40	0	0	0	0	1	0	1
41	0	0	0	0	0	0	1
42	0	0	0	0	0	0	0

Table 11: Complete Microcode Schedule

	phase_choice_c	coeff_choice_c	en_scaling_c	en_add_c	en_mag_shift_c	en_sub_c	en_sat_c
Cycle No.	cvec(11)	cvec(10)	cvec(9)	cvec(8)	cvec(7)	cvec(6)	cvec(5)
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0
24	0	1	0	0	0	0	0
25	0	1	0	0	0	0	0
26	0	1	1	0	0	0	0
27	0	1	1	1	1	0	0
28	0	1	1	1	1	0	0
29	0	0	1	1	1	0	0
30	0	0	1	1	1	0	0
31	0	0	0	1	1	0	0
32	1	0	0	0	1	1	0
33	1	0	0	0	1	1	1
34	1	1	0	0	1	1	1
35	1	1	0	0	1	1	1

Continued on next page

Table 11 – continued from previous page

	phase_choice_c	coeff_choice_c	en_scaling_c	en_add_c	en_mag_shift_c	en_sub_c	en_sat_c
Cycle No.	cvec(11)	cvec(10)	cvec(9)	cvec(8)	cvec(7)	cvec(6)	cvec(5)
36	1	1	1	0	0	1	1
37	1	1	1	0	0	0	1
38	1	1	1	0	0	0	0
39	1	0	1	0	0	0	0
40	1	0	1	0	0	0	0
41	1	0	0	0	0	0	0
42	1	0	0	0	0	0	0

Table 12: Complete Microcode Schedule

	en_reschoice_c	cadd_en	en_wr_cmem_1	en_wr_cmem_2	en_mmux_c
Cycle No.	cvec(4)	cvec(3)	cvec(2)	cvec(1)	cvec(0)
1	0	1	0	0	0
2	0	1	0	0	0
3	0	1	0	0	1
4	0	1	0	0	1
5	0	1	0	0	1
6	0	0	0	0	1
7	0	0	0	0	1
8	0	0	0	0	0
9	0	0	0	0	0
10	0	0	0	0	0
11	0	0	0	0	0
12	0	0	0	0	0
13	0	0	0	0	0
14	0	0	0	0	0
15	0	0	0	0	0
16	0	0	0	0	0
17	0	0	0	0	0
18	0	0	0	0	0
19	0	0	0	0	0
20	0	0	0	0	0
21	0	0	0	0	0
22	0	0	0	0	0
23	0	0	0	0	0
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	0
27	0	0	0	0	0
28	0	0	0	0	0
29	0	0	0	0	0
30	0	0	0	0	0
31	0	0	0	0	0
32	0	0	0	0	0
33	0	0	0	0	0
34	0	0	0	0	0
35	0	0	0	0	0
36	0	0	0	0	0
37	1	1	0	0	0
38	1	1	1	1	0
39	1	1	1	1	0
40	1	1	1	1	0
41	1	1	1	1	0
42	0	0	1	1	0

A.2 Schedule Description

In this section, we explain various *contiguous* portions of the schedule.

A.2.1 Cycle 1

In the first cycle, and only in first iteration, the intrinsic information which is expected to be ready before the decoder is enabled, needs to be latched within the decoder system immediately. The latching signal, en_intr_wr lasts exactly one cycle.

A.2.2 Cycles 1-5

Simultaneously, to start reading check-to-bit update messages from check memory blocks in every iteration **except** the first iteration, check address memory generation needs to be started, using cadd_en. Since it takes 5 cycles to read all messages, 2 at a time, the address generation lasts 5 cycles.

A.2.3 Cycles 2-6

Linked to cadd_en, check memory blocks get automatically enabled from next cycle. Since addresses are ready cycle 2 onwards, and memory blocks enabled for access as well, we need to keep en_wr_cmem[1|2] pulled low, so that read happens on the check memory blocks.

A.2.4 Cycles 3-7

From cycle 3 onwards, message data starts appearing on output ports of check memory blocks. It has to be appropriately distributed over different interconnect wires using memory muxes next. Hence memory muxes are now enabled for these 5 cycles. The corresponding signal is en_mmux_c.

A.2.5 Cycles 4-8

After memory muxes propagate this input message on the interconnect, processor muxes have to be enabled, so that they can provide the bit processing units the required inputs by sampling the right wires on the interconnect. The corresponding signal is en_pmux_b.

A.2.6 Cycles 5-9

The input signals are changed from 9-bit sign-magnitude form, to 13-bit 2's complement form suitable for internal datapath of the bit processing units. No enable signal is required for this processing.

A.2.7 Cycle 5

The output of 3-input adder being used in accumulation scan in bit processing units, which is also being used via feedback as 3^{rd} input, is cleared in this cycle before doing total sum of the input messages. The corresponding signal is cl_add_b.

A.2.8 Cycles 6-10

The total sum of input messages is performed via a 3-input adder, enabled using en_add_b, by taking 2 inputs at a time.

A.2.9 Cycles 6-14

Simultaneously, 2 shift registers, enabled using en_shift_b, take in the 2 inputs to the decoder, which they later present on their output during output scan. The last output of shift register should come out just before last round of subtraction happens in output scan. Hence the shift registers remain enabled till cycle 14.

A.2.10 Cycle 11

The total sum of messages needs to be further added with the intrinsic information at every bit processing unit, every iteration. This is done immediately in next cycle after accumulating the inputs, using en_intr_add signal.

A.2.11 Cycle 12

The hard decision on bit's estimated value, which needs to be made every iteration, is made in this cycle, using en_codetest signal. This is because in previous cycle, *total information* about the bit in the current iteration becomes available. This is one of those *exceptional* computations, that does not wait for clock edge to happen. The same

signal, at the next rising clock edge(clock number 13), leads to transfer of the value vector(computed asynchronously within previous cycle) onto outputs of bit processing units. Hence en_out_b is also simultaneously asserted in this cycle.

A.2.12 Cycles 12-15: Read after Write

The value vector sent out by each bit processing unit needs to be first written in special locations of bit memory blocks, 11 and 12. Subsequent to that, it needs to be read(by check processing units), and presented to check processing units for parity constraint checking. Hence in cycles 12 and 13, special addressses are generated by enabling signals testcode_wr and badd_en simultaneously for 2 cycles. At the rising edge of cycle 13, the value vector, enable signal for the bit memory blocks(en_bmem), and the addresses for the 2 locations where value vector is to be written, all get computed. Hence, by also asserting the memory write signals en_wr_bmem_[1|2] in this cycle, the write operation gets performed at the rising edge of cycle 14. By allowing the addresses to remain stable till next cycle, and also the bmem_en to remain 1 at the next clock edge, but by pulling down en_wr_bmem_[1|2] to 0(to signal a read), the value vector gets presented at the output ports of bit memory blocks during the cycle number 15.

A.2.13 Cycles 15-19

Since the bit memory outputs become available in cycle 15, bit memory muxes are enabled as well during this cycle, using en_mmux_b signal, to allow distribution of value vector in next 5 cycles. The same data is held on the output of bit memory blocks for 5 consecutive cycles due to design of the memory blocks("no change" configuration). Hence, the same value vector per bit gets distributed 9 times to 9 different check processing units in appropriate cycle.

A.2.14 Cycles 16-20

To propagate value vector further after it has appeared on interconnects, (check) processor muxes are enabled for 5 cycles using en_pmux_c signal, starting cycle 16.

A.2.15 Cycle 16

The parity from each bit estimate is accumulated using a 3-input XOR gate, one of whose inputs is the output itself, connected in a feedback mode. To clear any useless offsets,

the output of this sequential piece of logic is cleared, using cl_sign_acc signal, before the actual accumulation of parity starts.

A.2.16 Cycles 17-21

The accumulation of even parity is done using XOR gates over next five cycles, enabled via the en_sign_acc signal, using the bit estimate of various bits arriving at its inputs in these cycles.

A.2.17 Cycle 22

Since the final accumulated parity per check processing unit gets formed in cycle 22, checking the entire sequence of parities to be all-0-sequence is done in next cycle, by enabling the corresponding decide_cword signal in this(22nd) cycle.

A.2.18 Cycle 23

If the sequence of bit estimates was found to form a valid codeword, then the complete control path is disabled by disabling the microcode sequencing in this cycle, *asynchronously* via the valid_word signal. The decoder subsystem is then deemed to have converged upon a possible codeword that was transmitted to the reciever system.

A.2.19 Cycles 12-16: Winding back to Parallel Branching

While testing of estimated bit sequence as a codeword is being performed over cycles 12-23, simulataneously, bit nodes continue doing computation as if they were to prepare for another round of iteration. Hence, since the total information gets available in previous cycle (11th), the output scan starts from this cycle, via the signal en_sub_b. The calculation of residues is done by subtracting a delayed version of 2-inputs per cycle, from the total information, for next 5 cycles.

A.2.20 Cycles 13-17

Since the output of bit processing units is in sign-magnitude format, the sign and magnitude are extracted from the 2's complement residues prepared in last cycle, using en_res_conv signal, for 5 cycles.

A.2.21 Cycles 14-18

Since the output of bit processing units have 8 bits for magnitude representation, while the internal datpath of bit processing units have 13 bits, the magnitude is *saturated* to 8 bits in these 5 cycles, using en_sat_b signal.

A.2.22 Cycles 15-19

The saturated, 9-bit sign magnitude formatted outputs of bit processing units are presented at the output ports of the units in these 5 cycles, using the signal en_out_b. Simultaneously, to faciliate writing of these residues in appropriate locations of bit memory blocks, address generation for bit memory blocks, using perfect access patterns is started over, using assertion of badd_en signal.

A.2.23 Cycles 16-20

Starting cycle 16, the data at input ports of bit memory blocks is stable, as are the enable for the bit memory blocks(en_bmem) and the appropriate addresses. Hence, the write signals are asserted en_wr_bmem_[1|2] in this cycle, so that the write starts happening from next rising clock edge. Note that this write does not clash with the data being held stable at output ports(value vector), due to "no change" configuration.

A.2.24 Cycles 21-25

Since the value vector data has been distributed over the interconnect the requisite amount of cycles by now, one can immediately start preparing for check node updates, as soon as writing of bit-to-check update messages in bit memory blocks is over. For this, the badd_en is continued to be held high for 5 more cycles, between cycles 19 and 24. The address counter goes back to 0 at start of cycle 19, and re-counts from there. Pulling down signals en_wr_bmem_[1|2] to 0 leads to signalling of read operation to be performed. en_bmem remains at 1 during cycles 21-25, as it is derived from badd_en.

A.2.25 Cycles 22-26

Since the bit memory outputs again become available in cycle 22, bit memory muxes are enabled as well during this cycle, using en_mmux_b signal, to allow distribution of bit-to-check update messages in next 5 cycles.

A.2.26 Cycles 23-27

To propagate value vector further after it has appeared on interconnects, (check) processor muxes are enabled for 5 cycles using en_pmux_c signal, starting cycle 23.

A.2.27 Cycle 23

The sign from each check-to-bit message meant for next iteration is accumulated using a 3-input XOR gate, one of whose inputs is the output itself, connected in a feedback mode. To clear any useless offsets again, the output of is cleared, using cl_sign_acc signal, before the actual accumulation of sign starts.

A.2.28 Cycles 24-28

The total accumulation of sign of input messages is performed via a 3-input XOR gate, enabled using en_sign_acc, by taking 2 inputs at a time. The output of accumulation remains stable till at least cycle 40, when the signs of outgoing messages start getting computed using the accumulated sign.

A.2.29 Cycles 24-28

Simultaneously, the multiplicand and addend needed to transform the 2 inputs per cycle into log(tanh()) domain using DSP slices' MAC operation, are enabled to be chosen using coeff_choice_c signal in these 5 cycles. This choice is done on negative clock edges, starting the falling edge of 24th clock cycle.

A.2.30 Cycles 24-28

Since the same function is used to invert the transformation inputs, same DSP slices are used twice during the course of check node updates. Hence, at the first instance, check node inputs are multiplexed onto inputs of DSP slices during these cycles, using phase_choice_c signal. A '0' value of this signal signifies the accumulation phase, and hence the forward transformation of inputs. This multiplexing is done on negative clock edges, starting the falling edge of 24th clock cycle.

A.2.31 Cycles 24-32

Further simultaneously, 2 shift registers, enabled using en_sign_shift, take in the 2 input sign bits, which they later present on their output during output scan. The last output of shift register should come out just before last round of outgoing sign calculation happens in output scan. Hence the shift registers remain enabled till cycle 40.

A.2.32 Cycles 25-29

The magnitude part of input is transformed by approximating the transform to be a MAC function, to be performed by 2 DSP slices, starting cycle 25. No particular enable signal is required for this job.

A.2.33 Cycles 26-30

To account for fixed point during the *point-less* multiplication that happened within the DSP slice, shifting and scaling is performed post transformation. The shifting is enabled via en_scaling_c signal, and it starts off from cycle 26 onwards.

A.2.34 Cycles 27-31

The total sum of input messages is performed via a 3-input adder, unsigned this time, that is enabled using en_add_c, by taking 2 inputs at a time.

A.2.35 Cycles 27-43

Simultaneously, 2 shift registers, enabled using en_mag_shift_c, take in the 2 inputs to the decoder, which they later present on their output during output scan. The last output of shift register should come out just before last round of subtraction happens in output scan. Hence the shift registers remain enabled till cycle 43.

A.2.36 Cycles 32-36

Since the total information at the check node gets available in previous cycle (30th), the output scan starts from this cycle, via the signal en_sub_c. The calculation of residues is again done by subtracting a delayed version of 2-inputs per cycle, from the total

information, for next 5 cycles. Simultaneously, phase_choice_c signal is also asserted to '1' this cycle onwards, signifying the output scan.

A.2.37 Cycles 33-37

Since the output of check processing units have 8 bits for magnitude representation, while the internal datpath of check processing units have 12 bits, the magnitude is *saturated* to 8 bits in these 5 cycles, using en_sat_c signal.

A.2.38 Cycles 34-38

To invert the transformation before sending out the residue, same DSP slices are used again, as mentioned before. Hence, at this second instance, saturated magnitude outputs are multiplexed onto inputs of DSP slices during these cycles, using phase_choice_c signal, which has already been pulled up to a '1' value. This multiplexing is done again on negative clock edges, starting the falling edge of 34th clock cycle.

A.2.39 Cycles 34-38

Simultaneously, again, the multiplicand and addend needed to inverse-transform the outputs, 2 per cycle, using DSP slices' MAC operation, are enabled to be chosen using coeff_choice_c signal in these 5 cycles. This choice is again done on negative clock edges, starting the falling edge of 34th clock cycle.

A.2.40 Cycles 35-39

The magnitude part of would-be outputs are inverse-transformed by approximating using a MAC function, to be performed by 2 DSP slices, starting cycle 35. No particular enable signal is required for this job.

A.2.41 Cycles 36-40

To account for fixed point during the *point-less* multiplication that happened within the DSP slice, shifting and scaling is performed post transformation. The shifting is enabled via en_scaling_c signal, and it starts off from cycle 36 onwards.

A.2.42 Cycles 37-41

The scaled magnitudes are now multiplexed on appropriate wires of the 2 output ports of bit processing units, using en_reschoice signal, starting cycle 37 onwards. Simultaneously, calculation of final sign of each of these output messages is started off by pulling up the en_sign_res signal, which is appended to the magnitude as soon as it gets computed. Further simultaneously, to faciliate writing of these residues in appropriate locations of check memory blocks, address generation for check memory blocks, using perfect access patterns is started over, using assertion of cadd_en signal.

A.2.43 Cycles 38-42

Starting cycle 38, the data at input ports of check memory blocks is stable, as are the enable for the check memory blocks(en_cmem) and the appropriate addresses. Hence, the write signals are asserted en_wr_cmem_[1|2] in this cycle, so that the write starts happening from next rising clock edge.

A.2.44 Cycles 1-5: Over to Next Iteration

Since the bit-to-check update messages have been written into check memory blocks by end of cycle 42, one can immediately start preparing for bit node updates for next iteration. This is done by cycling around, and going back to processing as detailed in section A.2.2.