A High-Throughput, Flexible LDPC Decoder for Multi-Gb/s Wireless Personal Area Networks

Matthew Weiner
Borivoje Nikolic

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-177
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-177.html

December 22, 2010

A High-Throughput, Flexible LDPC Decoder for Multi-Gb/s Wireless Personal Area Networks

by Matthew Weiner

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report:

Committee:

Professor Borivoje Nikolić

Research Advisor

Date

* * * * * *

Professor Venkat Anantharam

Second Reader

Date

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Low density parity check (LDPC) codes are a class of powerful error correcting codes that can perform close to the Shannon limit with sub-optimal decoding schemes. The decoding algorithm, called the sum-product or belief propagation algorithm, lends itself to a parallelized implementation, which allows higher throughputs than competing codes, such as turbo codes. Virtually all modern high performance wired and wireless communication standards include an option for using LDPC codes, and emerging standards use it as the main choice for error correction coding. Examples of current standards include 10GBASE-T for ethernet, DVB-S2 and -T2 for satellite and terrestrial digital television, and WiMAX for high speed wireless internet, while emerging standards are the 802.11ad, WiGig, and 802.15.3c for 60GHz wireless LAN and PAN. Wireline links have moderately varying operating conditions, so standards generally require only one LDPC code of a particular rate to achieve the desired error performance. Decoders that operate on only one code are called fixed decoders. In contrast, wireless links have widely varying operating conditions because users move around, objects block or reflect the signal, and other users generate interference. By providing several LDPC code options, the system can measure the channel conditions and choose the code rate that will give the desired error performance while maximizing throughput. Using multiple LDPC codes may benefit the system's bit error rate and throughput, but it complicates the decoder implementation, resulting in larger and higher power designs. Flexible decoders work with multiple LDPC codes.

1.1 Problem Statement

Flexible LDPC decoders dissipate more power and occupy more area than their fixed counterparts because of the overhead required to reconfigure the structure of the decoder. The hardware overhead comes in different forms depending on the code's underlying structure. For example, in [23] the decoder makes all processing units larger in order to accommodate the worst-case code, and in [14], the decoder time multiplexes large routers. Unfortunately, emerging wireless PANs demand flexible decoders, high-throughput ($>$ 1Gb/s), and low-power ($<$ 50mW), but no current designs have been able to deliver this due to the excessive overhead of flexibility. This work develops a highly parallelized, fully pipelined decoder architecture that meets all three constraints. It features granular check nodes, the option for a variable depth pipeline, no explicit memories, and an architecture that accommodates aggressive voltage and frequency scaling.

1.2 Scope of Work

This work designs a low-power, high-throughput architecture compatible with any LDPC matrices containing groups of non-overlapping layers. It gives a heuristic method for finding a near-optimal architecture and details the structure of each of the decoder's building blocks. To demonstrate the capabilities of proposed design, an entire decoder compatible with the 802.11ad wireless PAN/LAN standard was created in Simulink using Xilinx System Generator blocks and synthesized using Synopsys' Design Compiler. Timing was verified and power consumption was found for the worst-case matrix at the two throughput levels used for low-power operation.

1.3 Related Work

Although no significant work has been published on decoders for multi-Gb/s wireless PANs, much work has been done on reconfigurable decoders for other application spaces. In particular, Karkooti et. al. developed flexible designs based on the 802.11n standard in [13]. They processed one sub-matrix at a time according to the layered decoding schedule, which gives automatic flexibility since supporting different matrices is just a matter of running more sub-iterations within one iteration. To support the different blocks lengths in the code, they designed the memory for the largest code and shut off portions of it for the smaller codes. They also included an early detection unit that terminated decoding if a full syndrome check passed. In [23] Shih et. al. created an excellent design for the WiMAX standard that achieved a power dissipation of 52mW for the worst-case matrix in

the standard at the maximum required throughput of 222Mb/s. They also design for the worst-case and shut off unused portions of the hardware for smaller codes, but they used an approximate and faster early termination method to increase throughput at the cost of performance. Additionally, they reordered the matrices to make them more friendly for implementation. Many other designs have been developed for the WiMax standard, such as [17] and [10]. Another popular standard is the DVB-S2 and -T2 standards for digital video broadcasting. These codes are extremely long and the submatrices are large, which makes their design challenging. Generally, the decoder processes one submatrix at a time using layered belief propagation and has large memories. Only one permuter is put on the chip because of its physical size. Published decoders include [20] and [5]. The throughput of all the above decoders suffers because of the lack of memory bandwidth and the degree of serialization used. Even if the clock frequency could be increased to increase throughput, the power would increase dramatically.

1.4 Research Contributions

This work extends the fixed architecture in [26] to flexible applications without significant overhead. It accomplishes this primarily by developing two ways of exploiting the characteristics of a group of LDPC codes where the codes have multiple layers with non-overlapping columns. The first method makes the check nodes granular, where they can either be coarse or fine. For example, a single coarse check node that handles, say, 16 inputs can break into two finer check nodes that handle 8 inputs each. Second, the pipeline depth decreases for lower rate codes with large submatrices because less stages of the check nodes' compare-select tree need to be traversed because the number of inputs are less for the finer check nodes. This is only used with large submatrices because the depth of the check nodes is large enough to justify pipelining it.

In addition, this design removes the bubbles in the previous work's pipeline by changing the pipeline depth and modifying the variable nodes to process two independent frames at once. This more than doubles the possible throughput because each iteration takes less cycles and two frames are processed simultaneously instead of one.

1.5 Organization

The organization of the rest of the report is as follows. Chapter 2 begins by introducing general and structured LDPC codes as well as the decoding algorithms. Chapter 3 discusses the high-level

hardware architectures of LDPC decoders. Chapter 4 discusses an architecture developed for high-throughput wireless PAN applications, and Chapter 5 describes the implementation of a decoder with the proposed architecture for the 802.11ad standard. Preliminary results from synthesis are also presented. Finally, Chapter 6 concludes the thesis with a summary.

Chapter 2

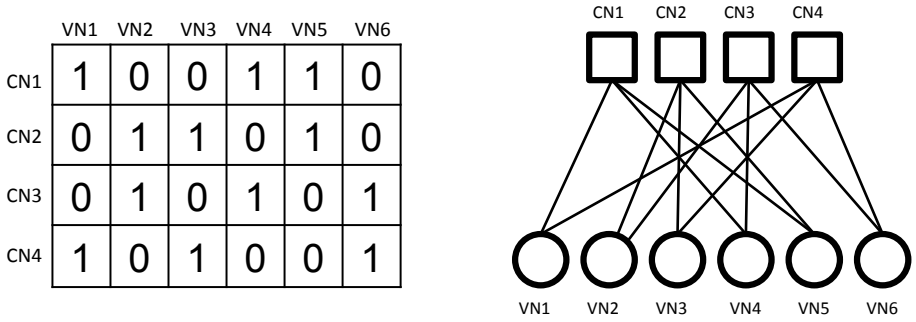
LDPC Codes and Decoding

Gallager invented low density parity check (LDPC) codes along with an iterative decoder in 1963 [9], but it was largely ignored because the current technology could not implement the complex hardware architectures required by the codes. After the invention and success of turbo codes, attention returned to iterative decoding, and MacKay rediscovered LDPC codes in the 1990s [18]. This time the community took interest, and the advancements of many researchers crowned LDPC codes as the new king of error correcting codes. First and foremost, Richardson et. al. showed irregular LDPC codes decoded with a sub-optimal algorithm can be designed to perform extremely close to the Shannon limit [21], with one code performing within 0.0045dB of the Shannon limit [3]. With this great potential, many people wanted to test the practical performance of the codes. The first fabricated decoder implemented a (1024, 512) LDPC code with a fully parallel architecture and a 1Gb/s throughput [1]. The design uncovered a design issue; the routing congestion, not the gate count, limited the size and construction of the realizable LDPC codes. Later designs almost exclusively backed off from the fully parallel architecture, opting instead for various types of serialization, and implemented regular or structured codes to reduce the wiring problem [19]. Almost all standards today use regular or irregular structured codes, such as WiMAX and 802.11ad.

2.1 LDPC Codes

Low density parity check codes are a class of linear block codes defined by a sparse $M \times N$ parity check matrix $\mathbf{H}$. N represents the number of bits in the code, called the block length, and M represents the number of parity checks. The rate of the code R is defined as $R = (N - M)/N$ and gives the fraction of information bits in each codeword. In the simple example in Figure 2.1(a),

the first row of the matrix specifies that bits 1, 4, and 5 should satisfy an even parity constraint, meaning their modulo-2 sum should equal zero, the second row specifies that bits 2, 3, and 5 should satisfy even parity, and likewise for the rest of the rows. A regular code has the property that all rows have the same number of ones and all the columns have the same number of ones, called the row degree d_c and column degree d_v, respectively. An irregular code is any code that is not regular. The parity check matrix of an LDPC code can be illustrated graphically using a factor graph (also called a Tanner graph) as shown in Figure 2.1(b). Each bit is represented by a variable node (a circle in the figure and referred to as a VN), and each parity constraint is represented by factor or check node (a square in the figure and referred to as a CN). An edge exists between a variable node i and check node j if and only if $\mathbf{H}(j,i) = 1$, and no nodes of the same type can directly connect to each other. Variable and check nodes that share an edge are called neighbors and are in each other's neighborhoods Col[i] and Row[j], respectively. A path that can be traced from one node in the graph back to the same node in the graph while not passing through any other node more than once is called a cycle. The performance of a LDPC code is determined by the girth of its factor graph, where girth is the size of the smallest cycle in the graph.



(a) (b)

Figure 2.1: LDPC code example: (a) parity check matrix representation, (b) Tanner graph representation.

2.2 Decoding Algorithm

A generic digital communications system has a transmitter, channel, and receiver, and the goal is to communicate information across the channel while keeping the number of bits received in error as small as possible. The transmitter contains a source that generates information, and encoder that adds some redundancy into the information, and a modulator that maps the information onto

symbols that can be sent across the channel. The channel models how the symbols are transformed as it travels through a medium, and it usually has a probabilistic description. The receiver has a detector, which decides what bit(s) the received symbol represents and a decoder that tries to use the redundant information added by the encoder to retrieve the correct information that the source generated.

Figures 2.2 and 2.3 show two similar communication systems. Both have a binary source, the same encoder, a BPSK modulator that maps $0, 1$ to $1, -1$, and an AWGN channel. The two schemes vary in the types of detectors and decoders used on the receiver side. The scheme in Figure 2.2 uses a hard detector and hard, non-iterative decoder, and the scheme in Figure 2.3 has a soft detector and soft, iterative decoder followed by a one bit quantizer.

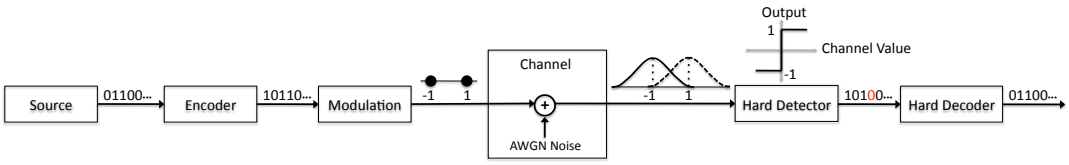


Figure 2.2: Model of a digital communications system with a hard detector and hard, one-shot decoder.

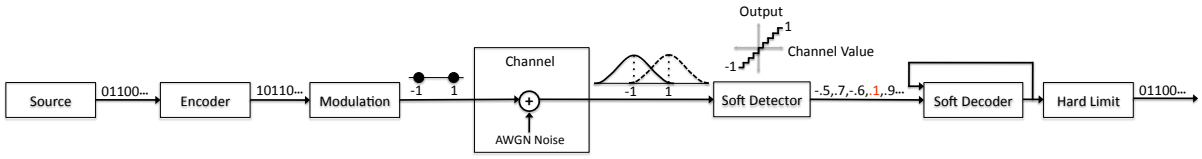


Figure 2.3: Model of a digital communications system with a soft detector and soft, iterative decoder.

Hard information has a one bit resolution, so it can only take on a zero or one value. To generate hard inputs, the detector uses a thresholding rule which says if the received value is within some range, the received bit is a one, otherwise it is a zero. The thresholds are determined by the channel statistics and the probability of the source generating a one or a zero, and the general formulation is given by the maximum a posteriori (MAP) or the maximum likelihood (ML) decision rule. Hard decoders process hard inputs and produce hard outputs. Soft information has multi-bit resolution, so it can represent not only whether the received bit is a one or a zero (determined by the sign), but also the reliability of the decision (determined by the magnitude). The reliability can be interpreted as a probability that the sign has the correct value. To generate soft inputs, the detector looks at the probability that the received value is a one or a zero and either gives one of the probabilities to

the decoder or some function of the ratio of the probabilities, usually the natural logarithm of the ratio. Soft decoders process the soft inputs, never making a hard decision until an output is required. They exploit the reliability information to make more informed decisions than hard decoders and consequently outperform hard decoders, but their complexity scales at least linearly with the number of reliability bits.

One-shot decoders take their inputs, process them once, and produce outputs. Alternatively, iterative decoders take their inputs, process them, produce outputs, and use the outputs as new inputs to the decoder. They repeat this cycle until some stopping criterion is reached or it reaches a maximum number of iterations. This approach is particularly useful when the inputs and outputs are soft. One-shot decoders have the advantage of being faster and less complex, but iterative decoders have superior performance because they can refine their estimates many times so that even if they make a wrong decision at first, it can be corrected in later iterations.

LDPC decoders use soft information and a sub-optimal iterative decoding algorithm called belief propagation, a form of the message passing algorithm (MPA). For a joint probability function that can be represented with a factor graph, the MPA marginalizes the joint density for all of the variable nodes simultaneously by passing soft information messages back and forth between variable and check nodes. The variable to check messages are based on the channel information and some of the messages from neighboring check nodes, and the check to variable messages are based on some of the messages from neighboring variable nodes. Only some of the messages are used because the MPA passes extrinsic, or new, information so that nothing is double counted. In other words, a variable to check message from variable node v_i to check node c_j must take all messages from neighboring check nodes into account *except* the message sent in the previous round from c_j to v_i. The same reasoning applies for check to variable messages.

The message passing algorithm is optimal for tree-like factor graphs because the messages that get passed back and forth between the nodes eventually reach the end of the tree and cannot be propagated any further. When the graph has cycles, as all LDPC codes have, the messages end up circulating around the graph and the values on the nodes may never converge. For well designed LDPC codes this is not an important issue because the girth of the graph is large enough to act tree-like over the number of iterations performed, so LDPC codes decoded with MPA can have near optimal performance. Another important note is that the MPA is fundamentally parallel, which makes it ideal for hardware implementation. The specific form of the MPA used in LDPC decoding will be discussed in Chapter 2.2.1.

2.2.1 Sum-Product Algorithm

The sum-product algorithm (SPA) is the traditional formulation of the message passing algorithm. In this application, the messages being passed by all of the nodes represent probabilities, also called beliefs, leading to its other name, belief propagation. Specifically, the message passed from variable node v_i to check node c_j measures the probability that v_i is a certain value given the channel realization and all the check to variable messages passed to v_i except from c_j in the previous iteration. Likewise, the message passed from c_j to variable node v_i measures the probability that v_i has a certain value given all the messages passed to c_j in the previous iteration except from v_i. Note that the SPA assumes all symbols in the received vector $\mathbf{y}$ are only correlated to the corresponding symbol in the transmit vector $\mathbf{x}$ and independent from all other symbols in the received vector ($y_i \perp\!\!\!\perp y_j \quad \forall i \neq j$), meaning that the channel is memoryless. Only the equations for the messages will be given below; for a general derivation of the SPA, see [16] and, for a derivation specific to LDPC, see [7].

In the probability domain, SPA runs as follows:

- **Step 0: Initialize.** All variable nodes load in their prior values based on the channel realization $\mathbf{y}$, which are the probabilities that the corresponding bit equals zero and one, and set these as their messages to the check nodes:

$$q_{ij}(0) = p_i^{pr}(0)$$
$$q_{ij}(1) = p_i^{pr}(1)$$

  where $q_{ij}(n)$ is the variable to check message from variable node i to check node j, and $p_i^{pr}(0)$ and $p_i^{pr}(1)$ are the probabilities from the channel that bit i equals 0 and 1, respectively. Note that two messages are sent from each node ($n = 0, 1$), one representing the bit equaling one and one for the bit equaling zero.

- **Step 1: Update check nodes.** Check nodes update their messages to each variable node based on the incoming variable node messages. The messages from check node j to neighboring variable node i is

$$r_{ij}(0) = \frac{1}{2}\left(1 + \prod_{i' \in \text{Row}[j] \setminus \{i\}} (q_{i'j}(0) - q_{i'j}(1))\right)$$
$$r_{ij}(1) = \frac{1}{2}\left(1 + \prod_{i' \in \text{Row}[j] \setminus \{i\}} (q_{i'j}(0) - q_{i'j}(1))\right)$$

where $r_{ij}(n)$ are the check to variable messages from check node j to variable node i, and Row$[j]$ is the set of variable nodes neighboring check node j (so Row$[j] \setminus \{i\}$ is the set of neighboring variable nodes except node i). Notice that the information that variable node i previously sent to check node j is removed, or marginalized, so that only extrinsic information is passed.

- **Step 2: Update variable nodes.** Variable nodes update their messages based on the newly computed check messages and their prior value. The messages from variable node i to neighboring check node j is

$$q_{ij}(0) = c_{ij} p_i^{pr}(0) \prod_{j' \in \text{Col}[i] \setminus \{j\}} r_{ij'}(0)$$
$$q_{ij}(1) = c_{ij} p_i^{pr}(1) \prod_{j' \in \text{Col}[i] \setminus \{j\}} r_{ij'}(1)$$

where c_{ij} is a normalizing factor to ensure $q_{ij}(0) + q_{ij}(1) = 1$, and Col$[i]$ is the set of check nodes neighboring variable node i (so Col$[i] \setminus \{j\}$ is the set of neighboring check nodes except node j). Again, the information check node j sent to variable node i is removed so that only extrinsic information is passed.

- **Step 3: Compute variable node output.** Step 2 computes the new messages that the variable nodes will send to the check nodes in the next iteration, which only contains extrinsic information. To obtain the posterior probabilities for each bit ($p_i^{post}(n) = \Pr(x_i = n|\mathbf{y})$), check node j's message is not ignored and a new normalization constant c_i is used in place c_{ij}:

$$p_i^{post}(0) = c_i p_i^{pr}(0) \prod_{j' \in \text{Col}[i]} r_{ij'}(0)$$
$$p_i^{post}(1) = c_i p_i^{pr}(1) \prod_{j' \in \text{Col}[i]} r_{ij'}(1)$$

$p_i^{post}(n)$ is the soft output value of variable node i, and to obtain the hard output b_i, use the decision rule

$$b_i = \begin{cases} 0, & \text{if } p_i^{post}(0) \geq p_i^{post}(1) \\ 1, & \text{otherwise} \end{cases}$$

The hard decisions can be used to check if all the parity checks are satisfied (or appoximately satisfied) and end the algorithm early, which is called early termination.

- **Step 4: Iterate.** Repeat Steps 1 through 3 until the algorithm converges by some metric

such as all parity checks are satisfied (early termination) or until a preset maximum number of iterations is reached.

For digital circuits, the multiplications required by the probability domain SPA use large amounts of hardware and power. Instead of using probabilities, the same information can be conveyed with log-likelihood ratios (LLR), which changes the multiplications to additions, which are more implementation friendly. LLRs for the priors, check to variable messages, variable to check messages, and posteriors are defined in Equation 2.1, respectively.

$$L^{pr}(x_i) = \log\frac{\Pr(x_i = 0|y_i)}{\Pr(x_i = 1|y_i)} = \frac{2}{\sigma^2}y_i \tag{2.1a}$$

$$L(r_{ij}) = \log\frac{r_{ij}(0)}{r_{ij}(1)} \tag{2.1b}$$

$$L(q_{ij}) = \log\frac{q_{ij}(0)}{q_{ij}(1)} \tag{2.1c}$$

$$L^{post}(x_i) = \log\frac{p^{post}(0)}{p^{post}(1)} \tag{2.1d}$$

where σ^2 is the noise variance of the channel, and the second equality of Equation 2.1a is only true for an AWGN channel.

In the LLR domain, the SPA now proceeds as follows:

- **Step 0: Initialize.** Load all variable nodes with their corresponding $L^{pr}(x_i)$ and send these to all neighboring check nodes.

- **Step 1: Update check nodes.** Check nodes update their messages to each variable node based on the incoming variable node messages. Equations 2.2 and 2.3 give the message from check node j to neighboring variable node i.

$$L(r_{ij}) = \Phi^{-1}\Bigg(\sum_{i' \in \text{Row}[j]\setminus\{i\}} \Phi(|L(q_{i'j})|)\Bigg)\Bigg(\prod_{i' \in \text{Row}[j]\setminus\{i\}} \text{sgn}(L(q_{i'j}))\Bigg) \tag{2.2}$$

  where

$$\Phi(x) = -\log\left(\tanh\left(\frac{1}{2}x\right)\right), \quad x \geq 0 \tag{2.3}$$

  Due to the properties of the Φ function, the sign and magnitude can be kept track of separately.

- **Step 2: Update variable nodes.** Variable nodes update their messages based on the newly computed check messages and their prior value. Equation 2.4 gives the message from variable

node i to neighboring check node j.

$$L(q_{ij}) = \sum_{j' \in \text{Col}[i] \setminus \{j\}} L(r_{ij'}) + L^{pr}(x_i) \tag{2.4}$$

- **Step 3: Compute variable node output.** As with the probability-domain SPA, Step 2 computes the extrinsic information to pass on to the check nodes in the next iteration, so to capture all the information and obtain the posterior LLRs, the message from check node j is added to Equation 2.4, as shown in Equation 2.5.

$$L^{post}(x_i) = \sum_{j' \in \text{Col}[i]} L(r_{ij'}) + L^{pr}(x_i) \tag{2.5}$$

  $L^{post}(x_i)$ is the soft output of variable node i, but for early termination or at the end of decoding hard decisions are needed. To make a hard decision on $L^{post}(x_i)$, choose whether a zero or one is most likely, which is determined by the sign of the LLR. Equation 2.6 formalizes this rule.

$$b_i = \begin{cases} 0, & \text{if } L^{post}(x_i) \geq 0 \\ 1, & \text{if } L^{post}(x_i) < 0 \end{cases} \tag{2.6}$$

- **Step 4: Iterate.** Repeat Steps 1 through 3 until the algorithm converges by some metric such as all parity checks are satisfied or until a preset maximum number of iterations is reached.

On a tree-like graph, the order in which messages are passed, called the schedule, does not have any effect since eventually all of the information is propagated to all nodes. For graphs with cycles, the schedule can have a large impact on the convergence rate and performance of the algorithm. The schedule described above where first all check nodes are updated and then all variable nodes are updated is called the flooding schedule. An alternative schedule called the layered decoding schedule, which works well for the structured codes described in Chapter 3.2, updates a subset of the check nodes in a "layer" of the matrix, updates all of the variable nodes, moves on to the next layer of check nodes, and again updates all of the variable nodes. This continues until all of the check nodes have been updated, and then continues again from the first layer in the next iteration. This method has become quite popular because the decoder converges two times faster than with the flooding schedule [12], but it has the drawback that pipelining the design becomes difficult and requires high degrees of serialization. Finally, dynamic schedules have been proposed that selectively update the nodes farthest from convergence. The nodes are selected by a metric such as the difference in the

magnitudes of the current and previous messages passed to the node [25]. To date dynamic schedules have offered only a minor improvement in performance for a large increase in complexity.

2.2.2 Modified Min-Sum Algorithm

Although using the SPA in the LLR domain offers significant simplifications for implementation over the probability domain SPA, it still is not well-suited for digital circuits. The Φ function in Equation 2.2 involves the standard and inverse hyperbolic tangent function, which must be implemented with a look-up table (LUT). This introduces additional quantization effects that reduce the performance of the decoder. In place of sampling the Φ function, Equation 2.2 can be approximated by noting that the magnitude of $L(r_{ij})$ is usually dominated by the minimum $|L(q_{i'j})|$ term. This is true because Φ is monotonically decreasing and $\Phi = \Phi^{-1}$. The sum in Equation 2.2 is then dominated by $\Phi(L(q_{i'j})_{min})$, which determines the value of the Φ^{-1} of the sum. Therefore, as shown in [11] and [8], the check to variable message equation of Equation 2.2 becomes

$$L(r_{ij}) = \min_{i' \in \text{Row}[j] \setminus \{i\}} |L(q_{i'j})| \prod_{i' \in \text{Row}[j] \setminus \{i\}} \text{sgn}(L(q_{i'j})) \tag{2.7}$$

With the use of Equation 2.7 in place of Equation 2.2 and all other equations of the SPA kept the same, the algorithm is called the min-sum algorithm. Notice that min-sum replaces the LUTs needed for the Φ function with implementation friendly comparisons used for the minimum operation.

Using the smallest magnitude in the min-sum equation above results in an overestimation of the check to variable message because, if all the terms were used, the sum would be larger and the result of the Φ^{-1} would be smaller. There are two common options to correct this: normalize the magnitudes by a constant α that is greater than one or subtract a fixed offset β, making sure the magnitude does not fall below zero [2]. These correction methods, collectively known as the modified min-sum algorithm, are given in Equations 2.8 and 2.9, respectively.

$$L(r_{ij}) = \frac{\min_{i' \in \text{Row}[j] \setminus \{i\}} |L(q_{i'j})|}{\alpha} \prod_{i' \in \text{Row}[j] \setminus \{i\}} \text{sgn}(L(q_{i'j})) \tag{2.8}$$

$$L(r_{ij}) = \max\{\min_{i' \in \text{Row}[j] \setminus \{i\}} |L(q_{i'j})| - \beta, 0\} \prod_{i' \in \text{Row}[j] \setminus \{i\}} \text{sgn}(L(q_{i'j})) \tag{2.9}$$

The α or β parameters must be tuned for each code and each quantization in order to optimize performance. Also, the correction parameters can be designed to have different values at different times, such as on different iterations. Unless the α correction parameter is a factor of two, the

β correction has a better implementation since it uses a subtractor instead of a divider. The β correction also has a finer tuning range than the α correction. If dividing by a factor of two fixes the error from using the min-sum approximation, use α correction; otherwise, use β correction.

Chapter 3

LDPC Decoder Architectures

Any implementation of an algorithm has a large design space that must be explored to arrive at the optimal solution. The first task a designer faces involves the choice of the basic architecture. As noted in Chapter 2.2, the message passing algorithm is inherently parallel, but that does not mean that a designer should parallelize as much as possible. No one architecture is best for all parity check matrices because the matrix size, matrix structure, and other factors play into the decoder's overall power and performance. In order to understand how to choose a decoder architecture, this chapter describes the tradeoffs between different architectural choices.

3.1 Basic Architectures

LDPC decoders have three basic architecture options: fully parallel, serial, and parallel-serial. Fully parallel architectures have one hardware instance for every variable node (VN) and every check node (CN), and the edges in the graph directly map to wires between the VNs and CNs, as demonstrated in [1]. The top of Figure 3.1 depicts a parallel implementation of the LDPC example from Chapter 2.1. All messages are stored within the nodes, and no memory is needed for the matrix because it is encoded in the wiring. Fully parallel decoders have the potential for extremely large throughputs since a full iteration takes the minimal number of clock cycles. However, the amount of routing increases drastically with increasing block length and decreasing code rate, so routing congestion becomes a significant problem. Routing tools need more area to route the wires, logic density drops, and silicon area goes to waste. Additionally, the long wires determine the critical path, throughput ends up suffering accordingly, and power increases because of switching activity of the long wires. Since all interconnections are preset, the decoder cannot implement more than one matrix, ruling

out fully parallel decoders in flexible applications, and it can only implement the flooding schedule.

In contrast to a fully parallel decoder, serial decoders use a single or a small number of hardware VNs and CNs along with large memory banks. Taking the case of the fully serial design, a single VN and a single CN both connect to a memory that stores the variable to check (V2C) and check to variable (C2V) messages. The bottom of Figure 3.1 shows a fully serial implementation of the running LDPC matrix example. If using the flooding algorithm, an iteration starts with the CN loading the variable to check messages from memory for all the VNs in the CN's neighborhood. It processes the messages according to the min-sum update equation, and write back the result to memory. The CN does this for every check node in the factor graph. Next, the VN loads in all check to variable messages from the CNs in its neighborhood and processes them according to the standard SPA equation. The VN writes the result back to memory, and repeats the process for every variable node in the factor graph. The entire process repeats for the next iteration. Instead of using the flooding schedule, the fully serial decoder can also implement the layered decoding schedule where a layer is a single row in the parity check matrix. The throughput of the fully serial decoder pales in comparison to that of the fully parallel decoder because it is limited by memory bandwidth, but it has no routing congestion problem because the only wires go from the single VN and CN nodes to the memory. Additionally, it occupies far less area than fully parallel decoders, despite the large memories needed. The design can be partially parallelized by adding more hardware nodes, but the memory bandwidth limits the amount of parallellization. Serial decoders offer the most flexible solution because the routing effectively becomes generating addresses in memory, which can be loaded in from off-chip, so it can implement arbitrary matrices. The address generation, or scheduling, can be difficult to design for irregular codes.

Parallel-serial architectures lie in between the fully parallel and serial options. The idea is to parallelize enough such that the wiring overhead does not become a problem and the throughput comes as close as possible to that of an ideal parallel decoder. Consider a decoder with as many VNs as columns and half as many CNs as rows in the parity check matrix. The decoder can complete a full iteration in two sub-iterations by reusing the same physical CNs (PCN) twice as different effective CNs (ECN), and it uses less wiring than the fully parallel design because there are fewer PCNs. The caveat is that the routing between one sub-iteration and the next needs to change because the same VN may connect to the first use of the PCN but not the second use. Some form of a router takes care of this varying routing, which regularizes the wiring since the router takes the place of complicated wires, so the place and route tool uses less long wires and increases logic density. In addition, this architecture has potential for flexible decoders since it already has some elements of

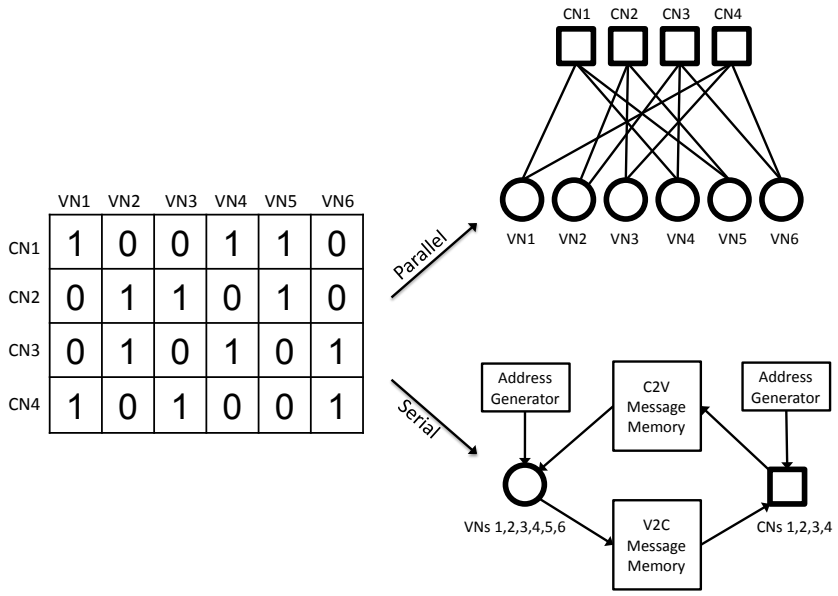


Figure 3.1: Fully parallel and fully serial architectures for the same LDPC matrix.

reconfigurability in the routing.

Despite the promise of parallel-serial designs, they become extremely complicated for general matrices. Consider an architecture that has three variable nodes and three check nodes; Figure 3.2 shows how the decoder would have to process a subsection of an arbitrary LDPC matrix. In the first sub-iteration, VN3 connects to PCN2 and PCN3 and VN2 connects to only PCN3. In the second sub-iteration, VN2 connects to PCN2 and PCN3, while VN3 and VN1 connect to PCN1. Comparing the two sub-iterations, PCNs 1 and 3 switch between having one or two connections, and VNs 2 and 3 switch between from having one and two connections. Imagine if the matrix had a row of all ones; it would have to accept three inputs in that sub-iteration! The router would have to deliver any subset of inputs to all outputs, and the nodes would have to accept as many inputs as nodes connected to the other side of the router. These constraints complicate the design of both to the point where it is not practical to implement this architecture for arbitrary matrices.

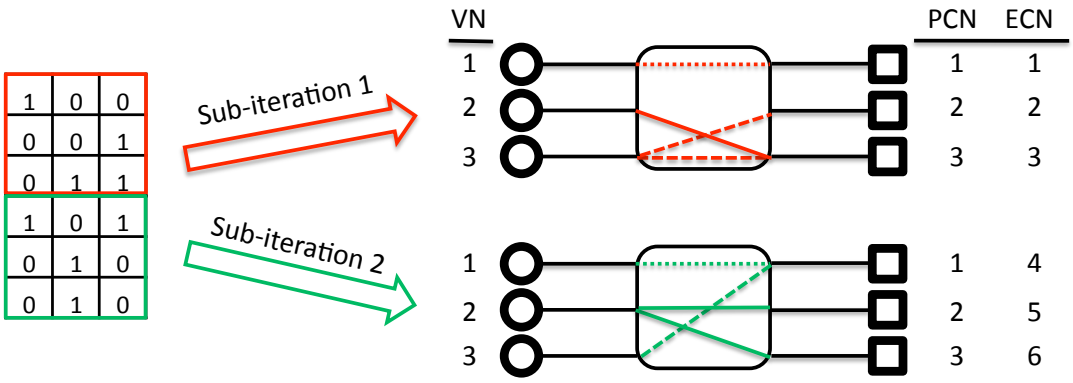


Figure 3.2: Routing complications for general parallel-serial architectures.

3.2 Structured LDPC Codes and Decoders

To aid in the design of fixed and flexible decoders, classes of structured LDPC codes referred to broadly as architecture-aware (AA) codes were developed including quasi-cyclic [24, 19], finite geometry [15], Reed-Solomon based codes[6], and many others. An AA matrix is itself composed of smaller $L \times L$ matrices, called submatrices. Generally, the submatrices are permutations or cyclic shifts of the $L \times L$ identity matrix or are $L \times L$ all-zero matrices. Each row of submatrices is called a layer. Figure 3.3 shows a simple example of an (16, 8) AA code with 4×4 submatrices. The base matrix contains only the shift amounts, and the full matrix has the shifted identity matrices substituted for each base matrix entry. Despite having structure, the codes can perform within a few fractions of a dB of random, irregular codes. Many standards use AA codes, such as 10GBASE-T, WiMAX, DVB-T2, DVB-S2, and 802.11ad.

0	3	1	
2	1	0	3

(a)

1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0
0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	1	0	1	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0
0	1	0	0	1	0	0	0	0	0	0	1	0	0	1	0

(b)

Figure 3.3: Example of an AA matrix with 4×4 submatrices; (a) base matrix, (b) full matrix.

This code construction gives a natural grouping of nodes and eases the parallel-serial decoder's routing problems with arbitrary matrices. Let V be the set of the L variable nodes that share a

column with a non-zero submatrix i, and let C be the set of the L check nodes that share a row with submatrix i. Because the submatrices are permutations of the identity matrix, each node in V has exactly one connection to a node in C, and vice versa. The L VNs can perform their calculations in parallel since they do not affect any of the same check nodes, and they are combined into a group called a variable node group (VNG). Similarly, the L CNs can be formed into a check node group (CNG). The two groups have fixed connections to a barrel shifter, which efficiently performs cyclic shifting in hardware. Figure 3.4 shows how the routing changes when processing submatrices with different shift amounts; the router is greatly simplified from the example in Figure 3.2.

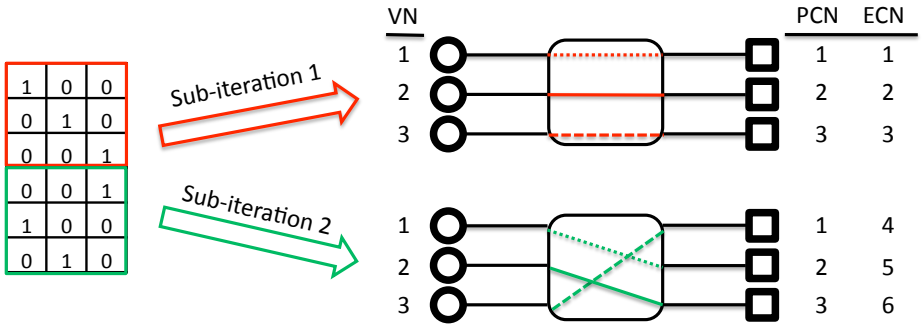


Figure 3.4: Simplified routing with AA matrices.

The basic unit to parallelize becomes a VNG and CNG instead of a VN and a CN. The most flexible and one of the more popular designs is the block-serial approach where only one VNG and CNG are implemented. This is the AA code version of the serial decoder, but the AA structure allows a higher degree of parallelization with no significant increase in hardware complexity beyond adding extra copies of the nodes. Fully block parallel designs lie on the other end of the design space. These are easier to implement than general fully parallel decoders because the routing congestion is reduced thanks to the shifters handling most of the complicated routing. Fully block parallel decoders have a small degree of flexibility due to the reconfigurable shifters. In general, the flexibility of a block-parallel architecture increases with decreasing parallelization, but throughput increases with increasing parallelization. In order to achieve a balance of flexibility and throughput, a designer can choose an intermediate degree of parallelization, which comes naturally from the AA structure.

Chapter 4

Proposed Architecture

To address the needs of the multi-Gb/s wireless application space, an architecture based on the fully pipelined, fixed decoder in [26] is developed. This chapter discusses the reasons for modifying that architecture, the features added, the general architecture, and a heuristic method for optimizing the parallelization and other decoder parameters.

4.1 Architecture Description

Decoders for wireless PANs must have flexibility, high-throughput, and low-power, which constrain the choice of basic architectures described in Chapter 3. A fully parallel design does not have the flexibility required, and a more block-serial design would not meet even the minimum throughput requirements without consuming several hundred milliwatts of power due to a high operating frequency. More likely, a parallel-serial design with a high degree of parallelism would meet the requirements. However, the more parallel the decoder, the more inefficient memory based architectures become because the message memories have to be split up into smaller and smaller banks. The overhead of the memories becomes comparable to the actual memory cells, and power dissipation increases dramatically. Fully pipelined architectures avoid this problem because they have no overhead associated with them. Although the pipeline registers use most of the decoder's power, the overall decoder uses less power than the memory-based architecture. For these reasons the proposed architecture builds off of the highly parallel, fully pipelined decoder in [26]. The decoder must use the flooding schedule because the layered schedule requires too many stalls in highly parallel, pipelined designs due to data dependencies.

Since the base architecture was not designed for flexibility or low rate codes, it does not perform

efficiently in wireless PAN applications. This work modifies the architecture by exploiting the non-overlapping layer structure of a common class of LDPC matrices used in wireless standards. It also adds the capability to process multiple frames simultaneously without significant overhead and improves the building blocks to aid in switching between different matrices. The final architecture has decreased power, increased efficiency, and flexibility within the class of codes considered. Although this architecture relies on a particular matrix structure, it does not tie itself to any one standard; the techniques developed apply to any decoder that uses a matrix with non-overlapping groups of layers.

4.2 Matrix Structure Exploitation

4.2.1 Matrices with Non-Overlapping Layers

Many emerging high-throughput wireless standards, such as 802.11ad and 802.15.3c, use both high and low rate LDPC matrices. The high rate codes have a small number of layers with few all-zero submatrices, but the low rate codes have more, sparser layers. If the decoder processes a matrix layer by layer, the high rate codes can be processed in fewer sub-iterations with less hardware wasted on all-zero submatrices compared to the low rate codes. However, the low rate codes have a special structural feature; they have groups of non-overlapping layers. Specifically, those matrices have groups of l layers with at most $N/(L \cdot l)$ non-zero submatrices that do not overlap, where N is the block length of the code and L is the dimension of the submatrix. Figure 4.1 shows two cases with $l = 2$ and $l = 4$. The non-overlappling layers can effectively be combined together to form a compressed matrix, and each compressed layer can be processed as one. To accomplish this, the check node architecture can be modified.

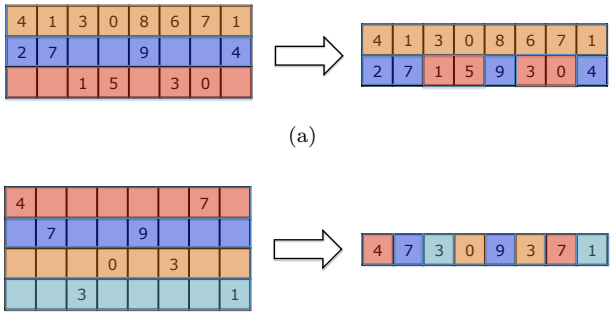


Figure 4.1: Examples of compressing matrices with : (a) two non-overlapping layers ; (b) four non-overlapping layers.

4.2.2 Check Node Granularity

Check nodes for the min-sum algorithm are almost universally implemented as a tree of compare-select (CS) blocks, shown in Figure 4.2, which search for the first and second minima of its inputs. In general, tree structures compute a local result for subsets of the input data and combine the local

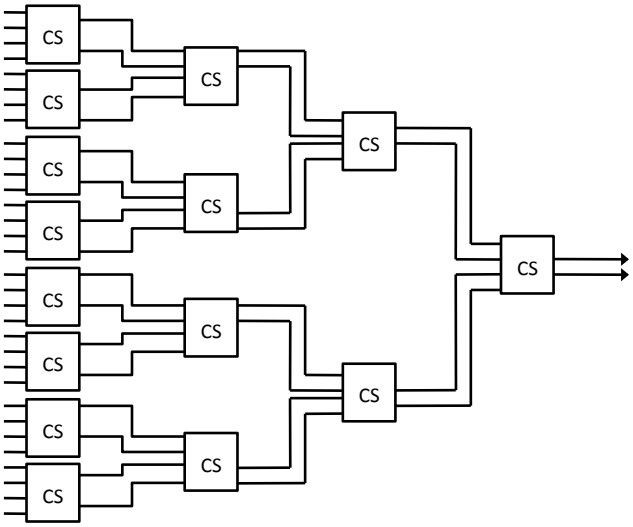


Figure 4.2: Standard check node architecture consisting of a tree of compare-select (CS) blocks.

results in successive stages to reach the overall result. In the case of the CS tree, the minimum two values of the four inputs are selected and passed on to the next stage to be compared with an identical block's minimum two outputs. Specifically, a CS block in the first level of the tree computes the minima for a subset of four inputs, a CS block in the second level computes the minima for a subset of eight inputs, all the way to the final CS block computing the first and second minima for all inputs. If the CS tree has as many inputs as the maximum row degree and the current layer has all non-zero submatrices (meaning there are as many incoming messages as tree inputs), the decoder needs to use the entire tree to compute the two minima. However, for the same tree and a layer that has half as many non-zero submatrices, the tree finishes the result one stage earlier than the previous case assuming all the inputs are routed to the same half of the tree. Now, if two layers do not overlap and have a degree of at most half of the maximum possible, their messages can be processed separately in the top half and bottom half of the tree, and their outputs must be taken at the second to last stage of the CS tree. Depending on how the non-zero submatrices are interleaved between the layers, this method requires extra routing before and after the check node to ensure the messages from the first layer's corresponding variable nodes connect to the top of the tree and

the second layer's corresponding variable nodes connect to the bottom of the tree. This method of processing multiple layers with with different parts of the check node is called check node granularity because a coarse check node can be used as two finer check nodes. This description equally applies to the case with l non-overlapping layers, where each layer gets routed to the same $1/l^{\text{th}}$ portion of the tree, and the check node acts as l smaller check nodes.

To illustrate, consider a decoder that has the variable nodes fully parallelized and the check nodes fully layer-serialized, meaning the number of check nodes equals the number of rows in a submatrix, and that the parity check matrix has three rows and eight columns of submatrices. The first layer contains no all-zero submatrices, and the second and third layers each have half of their submatrices equal to all-zero submatrices. Also, none of the non-zero submatrices in the second and third layers are located in the same column, so together the second and third layers have the same degree as the first layer. An example matrix is shown on the left of Figure 4.3. To process the first layer, the variable nodes send their messages to the corresponding input on the check node. To process the second and third layers simultaneously, the variable nodes connected to the second layer route their inputs to the top of the CS tree, and the variable nodes connected to the third layer route their inputs to the bottom of the tree. Figure 4.3 shows this process for one of the check nodes with the first layer being processed at the top-right of the figure and the second and third layers being processed at the bottom-right. The variable nodes in the figure represent the one variable node out of the variable node group that connects to the check node.

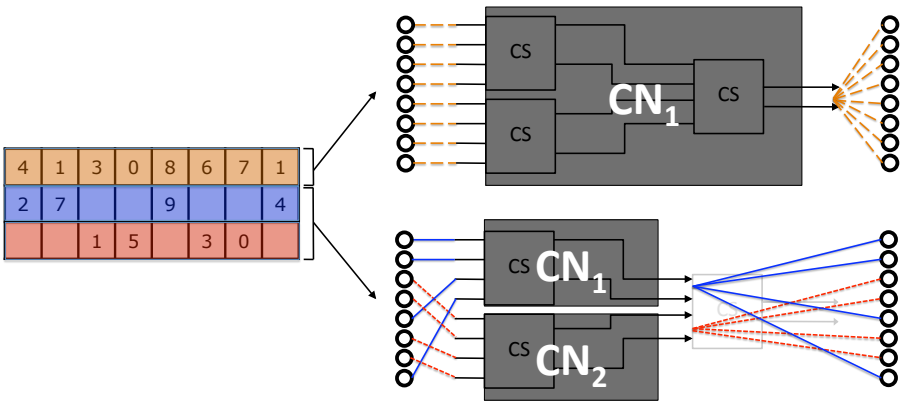


Figure 4.3: Example implementation of check node granularity.

This method is not restricted to this architecture because this easily could have represented an architecture with serialized variable nodes and more parallelized check nodes.

4.2.3 Variable Depth Pipeline

If the row degree of any of the layers in at least one of the matrices is large, it may be necessary to have pipeline stages within the check node since it needs several CS stages to compute its result. Additionally, if any of the matrices have a large l, the output must be taken early in the tree. By putting the pipeline stage at the same point where an output must be taken, a pipeline stage can be removed for the large l matrices. The pipeline depth varies based on the current code, which reduces the worst-case number of cycles to decode a frame. This lowers the decoder's frequency and saves power. Figure 4.4 shows the case with and without the variable depth pipeline.

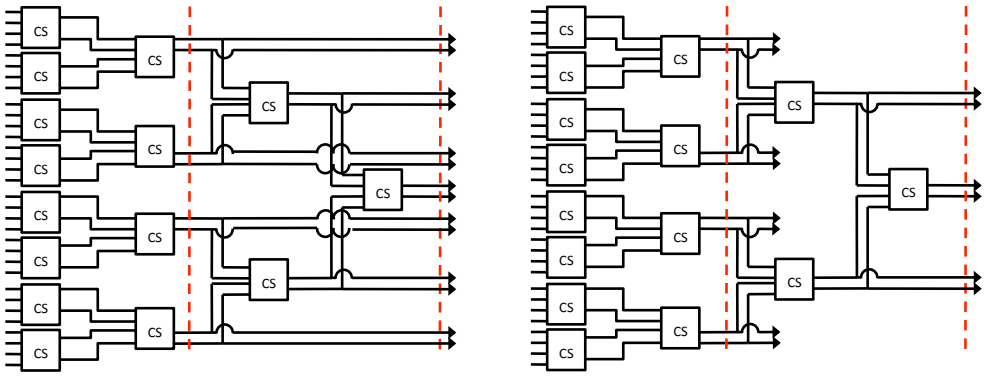


Figure 4.4: Check nodes that : (a) do not have and (b) have the variable depth pipeline; the dashed line indicates a register and the arrow indicates an output.

4.3 Architecture Overview

The decoder has five basic building blocks: variable nodes, check nodes, barrel shifters, pre-routers, and post-routers. Figure 4.5 shows a generic block diagram of the overall decoder if the check nodes are fully layer serialized, and Figure 4.6 shows the if decoder the variable nodes are fully parallelized and the check nodes are not fully layer serialized. Chapter 5 explains the need to differentiate between these two cases. The number and construction of each block depends on the parallelization, number of frames processed simultaneously, and pipeline depth.

4.3.1 Variable Node

The VN implements Equation 2.4 and performs both the CN and VN marginalizations, which keeps all memory within the VN. It accepts C unmarginalized C2Vs, where C depends on the CN

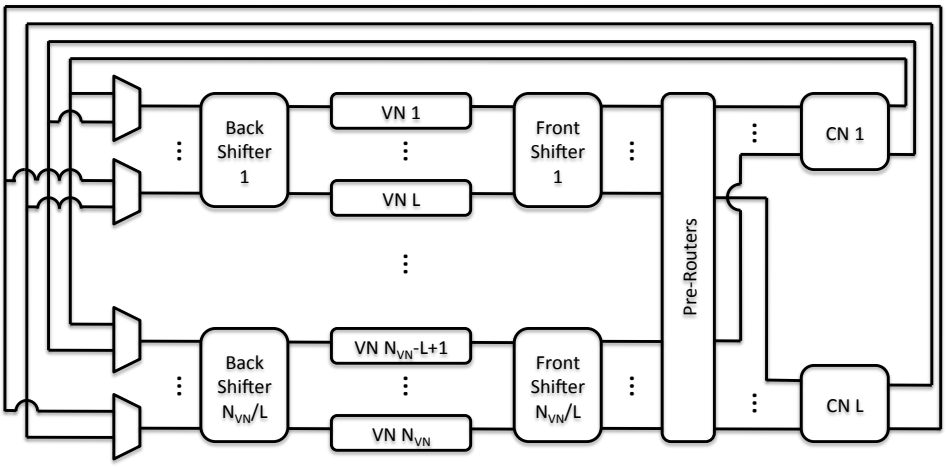


Figure 4.5: Block diagram of general overall decoder with CNs layer serialized and VNs possibly serialized; N_{VN} is the total number of hardware VNs.

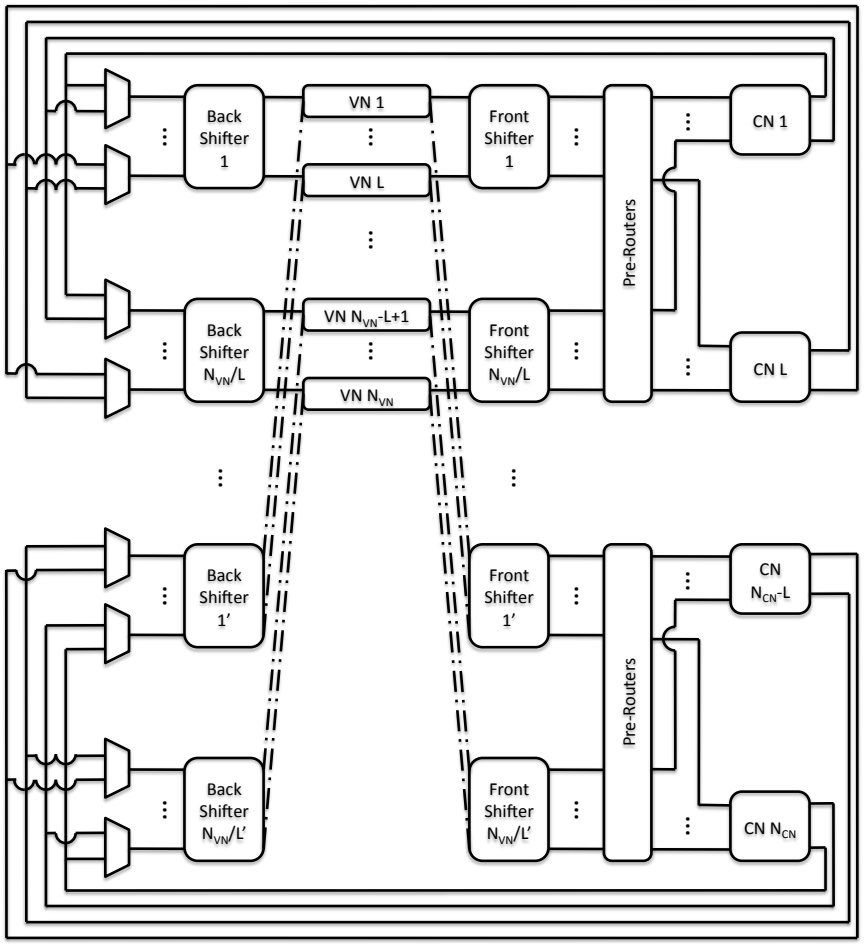


Figure 4.6: Block diagram of general overall decoder with fully parallel VNs and CNs possibly parallelized; N_{VN} and N_{CN} are the total number of hardware VNs and CNs, respectively.

parallelization level, marginalizes them, and performs the β min-sum correction from Equation 2.9. It accumulates the marginalized, corrected messages as well as storing them in a shift register. After all C2Vs have arrived and accumulation finishes, including adding in the prior value, the stored C2Vs are used to marginalize the V2C messages. These are sent out to the appropriate CNs and also stored in another shift register for marginalizing the next C2Vs. Based on the accumulated value, the VN outputs a hard decision according to Equation 2.6. Figure 4.7 shows a block diagram of the VN.

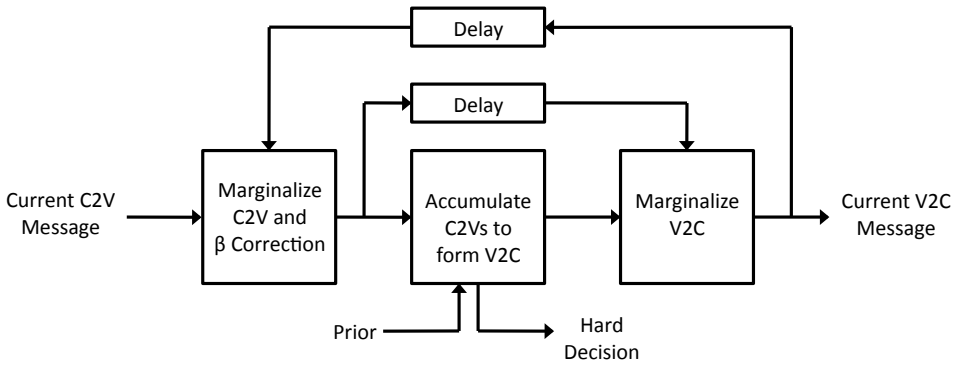


Figure 4.7: Variable node block diagram.

Although the architecture in [26] is fully pipelined, it does not use its pipeline efficiently due a data dependency caused by waiting for all C2Vs to arrive before sending out the next V2Cs. In fact, if the serialization is increased, the pipeline needs more stalls to avoid sending out incorrect messages. To fix this, the variable nodes are modified so that they can process more than one frame at a time. The changes include adding extra registers to hold the extra frame's prior and C2V accumulation results and adding or changing the size of muxes inside the accumulation block. This increases the throughput by a factor equal to the number of additional frames processed, while increasing power by a fraction of the increased throughput. The increased throughput can be traded for linear power savings by frequency scaling or super-quadratic savings by voltage-frequency scaling if possible. Chapter 5.3.1 shows how to modify the variable nodes for processing two frames simultaneously.

4.3.2 Check Node

Since the VN performs the marginalization in Equation 2.9, the CN only needs to find the first and second minima of all the incoming V2Cs' magnitudes, which it does using a compare-select tree. A sorting block at the beginning arranges pairs of inputs in ascending order, and subsequent stages of

the tree select the minimum two out of four inputs. The CN also needs to find the product of the incoming V2Cs' signs, which it does with an XOR tree. The number of inputs and stages for both trees depend on the size of the submatrices and the level of VN parallelization. The CN can take its outputs at different stages in the tree as described in Chapter 4.2, for which it will need muxes to select the correct output. Depending on the pipeline register placement, it can have a variable depth.

4.3.3 Barrel Shifters

In order to simplify and reduce the wiring, VNs and CNs connect to barrel shifters to implement the circular shifts required by each submatrix (see Chapter 3.2). Barrel shifters implement cyclic shifts using $\lceil \log_b(n) \rceil$ stages of n b:1 muxes each, where n is the number of inputs to the shifter. The wiring between each stage implements exponentially increasing shifts, which allows any shift between 0 and $n-1$. Depending on the direction of the wiring, the same general design can implement either left or right shifts. A control signal determines the shift amount at any given time.

The decoder requires two sets of shifters, one for the V2Cs, called the front shifters, and one for the C2Vs, called the back shifters. Both sets are needed because the messages must be routed to the correct CNs from the VNs, and then the messages from the CNs must go back to the same VNs. The messages cannot be stored in a permuted order in the VNs since the decoder does not use a layered schedule and all of the memory resides within the VNs.

4.3.4 Pre- and Post-Routers

When processing non-overlapping layers, barrel shifters alone cannot guarantee that each layer's inputs will go to the same section of each CN. As Chapter 4.2 described, granular CNs need two extra sets of routing to allow this. Pre-routing comes before the CN and selects which VNGs connect to a particular section of the CN. Post-routing comes after the CN and, for each VNG, selects which section's output to send back to the VNGs. The routing between the VNs and CNs in Figure 4.3 illustrates the routers' function. Both types of routers are made from a small number of muxes, but the exact implementation depends on matrix properties. The routing between the front shifters and the pre-routers is complex, but the routing between the post-routers and back shifters is simple. In the case of processing one layer, the routers do not shift the messages.

4.4 Heuristic Architecture Optimization

The proposed architecture has a large design space, so just knowing that the design should be highly parallelized does not give enough information to design an optimal decoder. Architecture explorations perform a sensitivity-based analysis by varying the basic parameters of the architecture in order to find the optimal combination [22]. The parallelization of the variable nodes and check nodes, the pipeline depth, and the number of independent data frames processed simultaneously make up the set of basic parameters for this LDPC decoder. A full exploration would take too long to finish due to generating many designs and performing synthesis runs for each design, so instead a heuristic based method is developed to find a near-optimal architecture. The idea is to find the worst-case operating frequency the decoder must run at to meet throughput requirements for the target operating class and then find how much power and area the basic blocks use. Since the analysis is somewhat rough, some heuristic rules must be applied to find a good architecture. The steps for the search are detailed below:

- **Step 1: List out all feasible combinations of basic parameters.** The pipeline depth and VN and CN parallelizations are orthogonal design parameters, so any combination can be chosen. However, too large a pipeline depth will have significant power and timing overhead for the registers, and too small a parallelization will require the operating frequency to be too high. The designer should put upper limits on the pipeline depth and the VN and CN serialization to avoid excessive design time. Serialization factors (SF) indicate the level of parallelization. A $\mathrm{SF}_{\mathrm{CN}}$ of 2 means that the design has half the number of check nodes as the fully parallel design.

  The parallel-serial stream architecture does not allow the VNs to be serialized unless the CNs are fully layer serialized. To see this, consider a decoder that has $\mathrm{SF}_{\mathrm{VN}} = 3$ and the $\mathrm{SF}_{\mathrm{CN}} = 2$ with submatrices of size nine. In this design, a single VN acts as three different VNs in the same submatrix in three consecutive cycles, which implies that the submatrix size must be divisible by the serialization factor. Figure 4.8 illustrates how the decoder would process a sample matrix, where each color corresponds to the columns processed in the same cycle. Looking across the rows of the first layer of submatrices, such as the one boxed, all the VNs with a one in the same row are processed in the same cycle. However, looking across the rows of the second layer, the VNs with ones in the same row are not processed in the same cycle, which means the inputs to the second layer's check nodes are not correct.

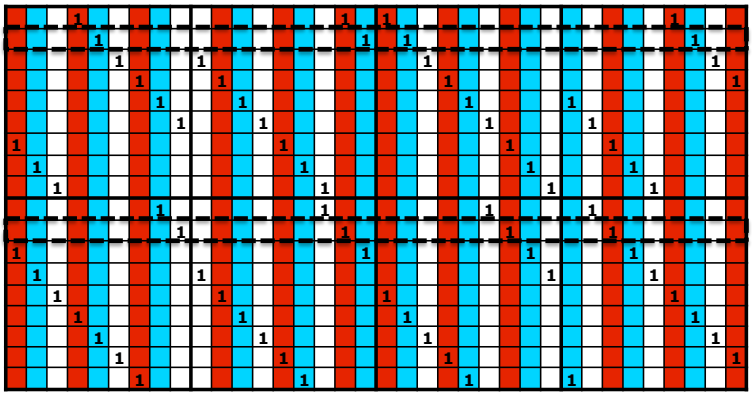
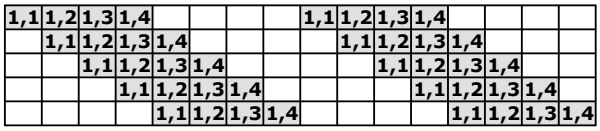


Figure 4.8: Example of why decoder cannot have serialized VNs and non-fully layer serialized CNs.

The maximum number of frames processed simultaneously depends directly on the pipeline depth and the parallelizations because these parameters determine the size of the bubble in the pipeline (see Step 2), but any number up to the maximum can be chosen for designs with fully parallel VNs. If the VNs are serialized, a new frame cannot be loaded into the pipeline if one finishes decoding before the rest; the decoder must wait until all frames finish decoding before loading in a set of new frames. This arises from timing issues with the pipeline (again, see Step 2), and to fix it would have a large hardware cost making it not attractive. Since processing multiple frames adds large amounts of hardware to the VNs increasing the area and possibly the power greatly, it may not be advantageous to process the maximum number of frames possible.

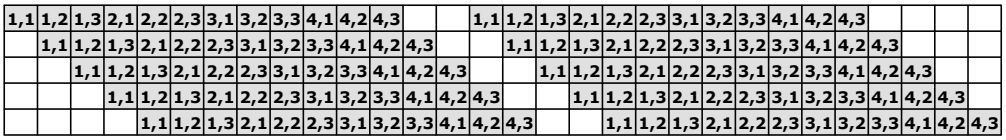
- **Step 2: Estimate the pipeline for each combination of parameters.** Each combination of parameters from Step 1 has one unique worst-case pipeline associated with it, which can be used to estimate the number of cycles per iteration. For example, some codes may require more sub-iterations than others, so their pipeline will be longer. The actual logic in each stage does not matter since only the number of cycles is of interest. When designing the pipeline, the dependencies between the same frame in different iterations must be considered. For instance, the next iteration on a frame, which starts with the VN sending out new V2Cs, cannot begin until the variable node has finished accumulation of all the C2Vs, but an independent data frame can begin processing as long as it does not interfere with any of the hardware the first frame is using in the same cycle. Also, for designs with serialized VNs, the next iteration can start for one set of effective VNs that a single physical VN acts as even if the other effective VNs are still accumulating. To illustrate, for $\mathrm{SF}_{\mathrm{VN}} = 2$, the next iteration can begin with

the first set of effective VNs while the second set are accumulating their last C2V. Figure 4.9 shows examples of the pipelines that can be constructed for a matrix having four layers. The numbers in each block indicate the VN and CN sub-iteration number, respectively, and the color indicates the data frame number. The first and second pipelines require eight cycles per iteration, and the third pipeline uses 14 cycles per iteration. Notice that the bubble in the first pipeline is large with respect to the occupied time, so it is reasonable to consider processing an extra frame during the idle time. Also, note that the serialized VN pipeline has extra cycles at the end of each iteration that must be accounted for in the total number of cycles per frame. These extra cycles cause the complication with processing multiple frames for decoders with serialized VNs, but it can be fixed by adding extra stalls into the pipeline such that the next iteration begins after all effective VNs finish decoding. However, the third pipeline's bubble is much smaller relative to the occupied time when compared to the first's, so it does not make as much sense to use extra hardware to try to fill up the bubble.



(a)

(b)



(c)

Figure 4.9: Examples of pipeline constructions used for frequency estimation : (a) $SF_{VN} = 1$, $SF_{CN} = 4$, Frames = 1; (b) $SF_{VN} = 1$, $SF_{CN} = 4$, Frames = 2; (c) $SF_{VN} = 3$, $SF_{CN} = 4$, Frames = 1.

- **Step 3: Estimate the worst-case minimum frequency for each combination.** Find the maximum average number of iterations among code rate at the required BER/FER. This information, along with the number of cycles per iteration and the standard specifications, are

used to estimate the worst-case operating frequency of the architecture by calculating

$$f_{min} = \frac{TI_{avg}C}{NF} \tag{4.1}$$

where f_{min} is the minimum operating frequency, T is the required throughput, I_{avg} is the average number of iterations for the decoder to converge for the given code, C is the number of cycles per iteration for the given pipeline and VN and CN serializations, N is the block length of the code, and F is the number of frames processed simultaneously.

- **Step 4: Eliminate architectures above minimum operating frequency limit.** To have less options to consider in the final step, architectures with f_{min} above an informed choice of a threshold are eliminated. The rationale behind this step is that voltage-frequency scaling saves dramatic amounts of power, and architectures that require a large f_{min} cannot be voltage-frequency scaled without extremely deep pipelining, which would offset the savings from scaling due to increased register power. Although some designs may have an f_{min} below this amount for its smaller pipeline depths, they cannot actually run at the required frequency at that pipeline depth.

- **Step 5: Find the power and area for the basic blocks.** The final evaluation consists of building and performing an initial synthesis on the basic blocks of each architecture, whose construction for one of the designs is detailed in Chapter 5.3, and estimating the power and area of the overall decoder by scaling the numbers of the blocks appropriately. For example, if the design has 2000 VNs, the power and area of the VN block is scaled by a factor of 2000. Only a first synthesis pass is performed in order to reduce the time required for this step. Both power and area are necessary for determining the (near) optimal design because the power that the first pass synthesis run reports does not take into account the wiring overhead (wire delay, power, and buffering) that arises from having a large number of VNs and CNs. Total logic area gives an indication of the wiring overhead, so it acts as another metric to judge the optimality of the design. Intuitively, the optimal architecture will balance the overall power and area, not have only the smallest area or only the smallest power numbers.

Chapter 5

Decoder Implementation

To show the advantages of the proposed architecture, a decoder has been implemented that complies with the 802.11ad standard for wireless LAN, which uses several LDPC matrices with groups of non-overlapping layers, requires multi-Gb/s throughputs, and targets low-power mobile applications. This chapter describes the standard, the heuristic architecture exploration, the construction of the decoder's building blocks, and the results from synthesis in a 65nm low-power CMOS technology.

5.1 The 802.11ad Standard

The 802.11ad single carrier standard, which has the same PHY layer as the WiGig standard, defines four architecture-aware LDPC matrices of rates 1/2, 5/8, 3/4, and 13/16 with a common block length of 672 bits. Figure 5.1 defines the base matrices for each of the codes. A numerical entry in the matrix represents a cyclicly shifted 42×42 identity matrix with the shift amount given by the number, and the entries with no number represent all-zero matrices. Figure 5.2 gives several examples of the shifted identity matrices that are substituted for non-blank entries in the base matrix.

All of the matrices are irregular, having a constant column degree d_v and varying row degree d_c. The maximum d_c decreases as the code rate decreases. Table 5.1 lists the relevant degree information for the matrices.

There are three classes of operation for LDPC codes: low-power operation (class 1), medium throughput operation (class 2), and high-throughput operation (class 3), although only the first two modes are expected to be used in applications due to power constraints of the target systems. Table 5.2 gives details on the specifications for each of the operating classes. The maximum FER and

Rate 13/16

29	30	0	8	33	22	17	4	27	28	20	27	24	23		
37	31	18	23	11	21	6	20	32	9	12	29	10	0	13	
25	22	4	34	31	3	14	15	4	2	14	18	13	13	22	24

Rate 3/4

35	19	41	22	40	41	39	6	28	18	17	3	28			
29	30	0	8	33	22	17	4	27	28	20	27	24	23		
37	31	18	23	11	21	6	20	32	9	12	29		0	13	
25	22	4	34	31	3	14	15	4		14	18	13	13	22	24

Rate 5/8

20	36	34	31	20	7	41	34		10	41					
30	27		18		12	20	14	2	25	15	6				
35		41		40		39		28			3	28			
29		0			22		4		28		27	24	23		
	31		23		21		20		9	12			0	13	
	22		34	31		14		4						22	24

Rate 1/2

40		38		13		5		18							
34		35		27			30	2	1						
	36		31		7		34		10	41					
	27		18		12	20				15	6				
35		41		40		39		28			3	28			
29		0			22		4		28		27		23		
	31		23		21		20			12			0	13	
	22		34	31		14		4				13		22	24

Figure 5.1: LDPC matrices for the 802.11ad standard.

$$p_0 = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ & & \ddots & & \\ 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}, \quad p_1 = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ \cdots & 0 & 1 & 0 & \cdots \\ & & & \ddots & \\ 0 & \cdots & 0 & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \end{bmatrix}, \quad p_{41} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 \\ & \ddots & & & \\ \cdots & 0 & 1 & 0 & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

Figure 5.2: Examples of cyclicly shifted identity matrices (subscript gives shift amount).

Table 5.1: Degree information for matrices.

Matrix	d_c	$d_{v,min}$	$d_{v,max}$
13/16	3	14	16
3/4	4	13	15
5/8	4	7	10
1/2	4	5	8

BER tolerable by any of the classes are 10^{-4} and approximately 10^{-5}, respectively.

Table 5.2: 802.11ad standard specifications.

Operating Class	Code Rates	Modulation	Coded Throughput (Mb/s)
1	1/2, 5/8, 3/4, 13/16	BPSK	1540
2	1/2, 5/8, 3/4, 13/16	QPSK	3080
3	1/2, 5/8, 3/4	16-QAM	6160

As noted in Chapter 3.2, LDPC matrices are built with structure in order to simplify decoder design, but they can have structure beyond that of the basic architecture aware matrix. If exploited, it can allow further optimizations that increase throughput and lower power. For the matrices of the 802.11ad standard, each lower rate matrix is created by adding layers of submatrices onto the higher rate code and eliminating some of the entries in the layers that came from the higher rate code. Figure 5.3 shows how the rate 3/4 matrix is constructed from the rate 13/16 matrix and how the rate 5/8 matrix is constructed from the rate 3/4 matrix. This structure can be used to compactly store the matrix in memory for all code rates and to switch between decoding each matrix seamlessly.

More importantly, all of the matrices have the non-overlapping layer property introduced in Chapter 4.2. Trivially, the rate 13/16 matrix is already three layers, and the rate 3/4 matrix is four layers. For the rate 5/8 matrix, the first two rows are part of their own sets, the third and fifth rows form another set, and the fourth and sixth rows form the fourth set. For the rate 1/2 matrix, the first and third layers form one set, the second and fourth layers form another set, the fifth and seventh layers form a third set, and the sixth and eighth layers form the final set. Figure 5.4 illustrates this property for the rate 5/8 and 1/2 matrices.

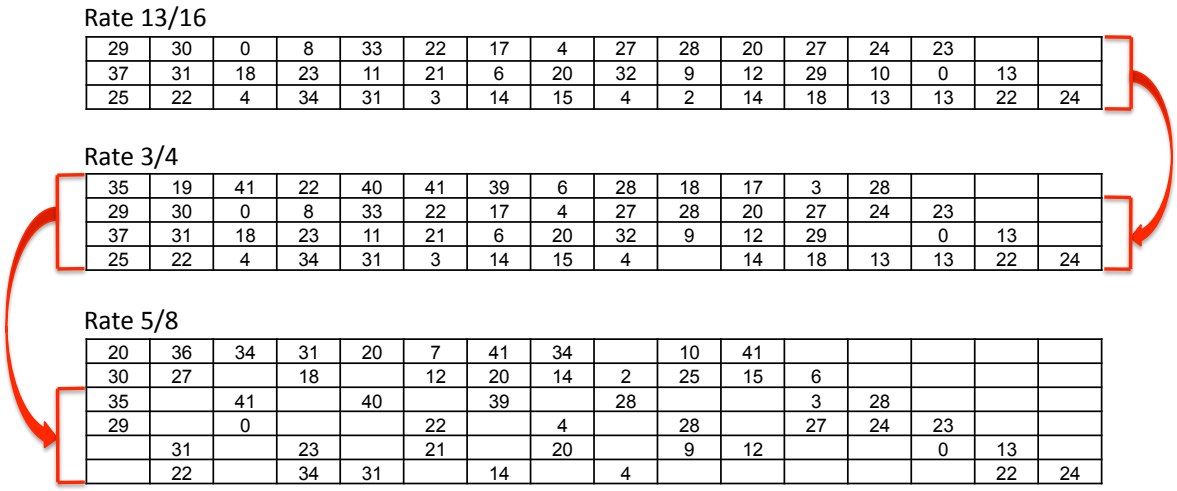
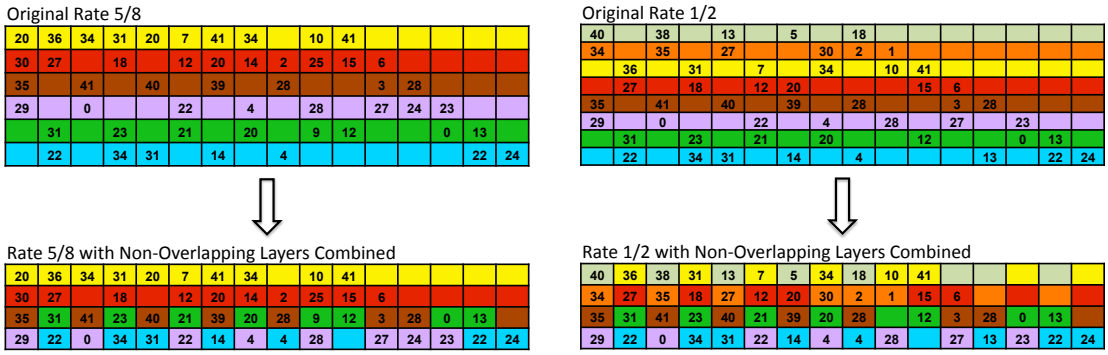


Figure 5.3: Construction of lower rate matrices from the higher rate matrices.



(a) (b)

Figure 5.4: The four disjoint sets of non-overlapping layers for the (a) rate 5/8 and (b) rate 1/2 matrices.

5.2 Architectural Design

Before performing the architectural analysis, the quantization, min-sum correction method, correction parameter, and the maximum number of iterations per frame must be determined because they affect the average number of iterations that the decoder requires to converge and thus the minimum operating frequency. A higher required operating frequency may require a highly parallelized architecture, while a lower required operating frequency will allow more serialized architectures that potentially can consume less power. Afterwards, the decoder's parameters are chosen using a heuristic architecture search.

5.2.1 Quantization, Min-Sum Correction, and Maximum Iterations

When implementing any system in hardware, it cannot have infinite precision because a finite number of bits are used to represent values. Floating point representation offers a large range and fine resolution, but it is complicated to implement. Also, LDPC decoders do not require a large range of numbers because the LLRs tend to take on values in a small range around zero. Instead, a fixed point representation is used because its arithmetic operations are easier to implement and can represent the LLRs with sufficient accuracy. Fixed point numbers have X bits before and Y bits after an implied binary decimal point, denoted as QX.Y. Wordlength refers to the number of bits used to represent a number and equals $X + Y$. Note that there are multiple quantizations, combinations of X and Y, for a given wordlength.

Decreasing the wordlength decreases the size of the decoder's datapath, which reduces wiring complexity and power, and the benefit is more pronounced with higher degrees of parallelization. However, doing this too aggressively causes the error correcting performance of the decoder to deviate from the ideal infinite precision implementation because fewer bits either have less resolution, less range, or both. Simulation provides a good way of determining the minimum wordlength that gives acceptable performance loss, called implementation loss (L_{im}). For LDPC decoders, wordlengths under four bits cause unacceptable implementation loss, and wordlengths over six bits have only a small performance benefit. The system's performance is measured by the bit error rate (BER), which is the fraction of incorrectly decoded bits over the total number of decoded bits, and the frame error rate (FER), which is the fraction of decoded words with at least one incorrectly decoded bit over the total number of words decoded.

In addition to the quantization, the min-sum correction method and correction parameter value affect the overall performance of the system. For this design the β offset method was used because it

has a finer tuning range for coarse quantizations. For example, with Q5.0 quantization a β correction scheme can have $\beta = 1$, but the smallest increment the α correction scheme can implement is $1/\alpha = 0.5$ if the division is implemented with a shifter. The choice of β must be included as a free parameter in the simulation, and typical values range from zero to 1.5.

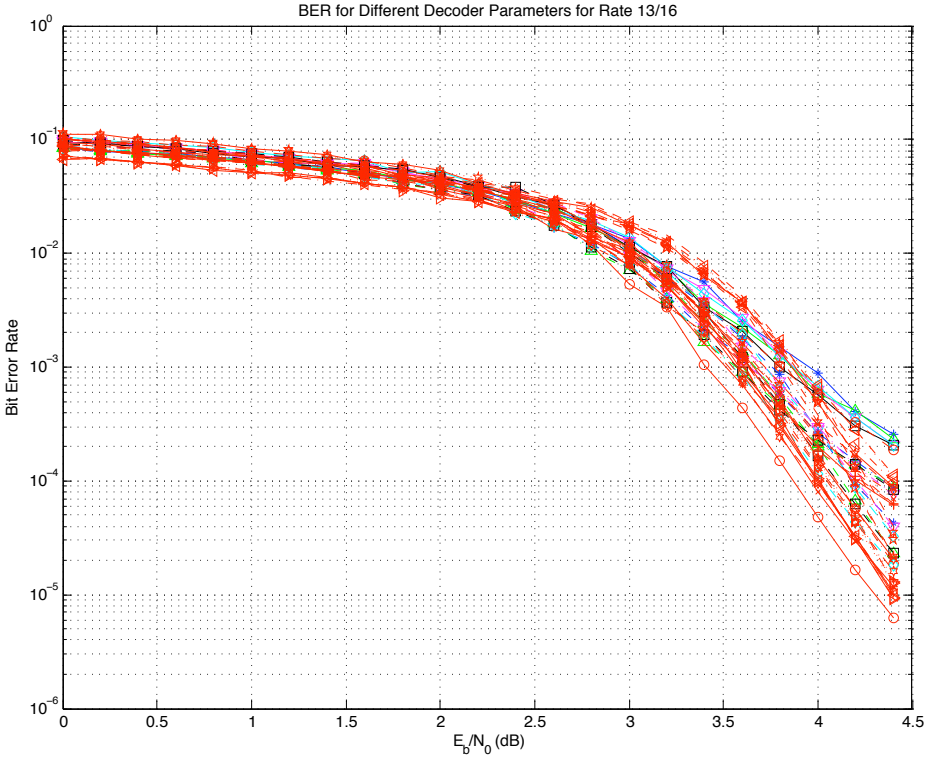
Finally, the maximum number of iterations per frame (I_{max}) can have a large effect on the performance of the decoder. A small value does not allow the frames that have a moderate number of errors to be decoded correctly, reducing performance. A large value allows frames that have too many errors to iterate for many cycles, which increases the average number of iterations of the decoder and benefits performance marginally. Simulation provides a way to find a reasonable value of I_{max} that does not compromise the decoder's performance or increase the average iterations, so I_{max} is included as another free parameter in the simulation. Values of I_{max} around 15 give a good balance between performance and average iterations.

A C++ simulation of the LDPC decoder was created that takes combinations of the above three variables as input arguments and outputs BER, FER, and average iteration plots for each combination over a specified E_b/N_o range for each code rate in the 802.11ad standard. Also included on the plots are the results of the exact sum-product algorithm and a floating point implementation of the min-sum algorithm with its optimal β parameter so that the implementation loss and deviation from the ideal deocding algorithm can be determined. Figure 5.5 shows the results for all combinations of variables for the rate 13/16 code. Displaying the curves for all combinations of parameters clutters the results and makes them difficult to interpret, so the rest of the result plots show only the curves for combinations of β and I_{max} that give the best performance for a particular quantization. Note that performance strictly increases for increasing I_{max}, so the curves shown will be for the I_{max} after which performance increases negligibly. These results are shown in Figures 5.6-5.9. Table 5.3 summarizes the implementation losses for each combination of parameters shown in the plots.

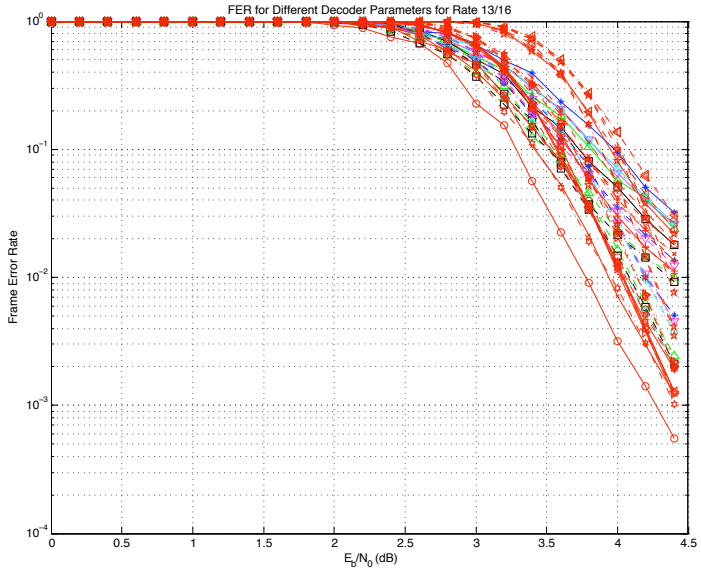
Table 5.3: Implementation loss of the optimal combination of parameters for each quantization and code rate.

Code Rate	Quantization	β	I_{max}	L_{im} (dB)
13/16	Q4.0	1.0	15	0.31
	Q4.1	0.5	15	0.35
	Q4.2	0.5	15	0.36
	Q5.0	1.0	15	0.01
	Q5.1	0.5	15	< 0.01
	Q6.0	1.0	15	0.01
3/4	Q4.0	1.0	15	> 0.25
	Q4.1	1.0	15	> 0.25
	Q4.2	1.0	15	> 0.25
	Q5.0	1.0	15	0.01
	Q5.1	1.0	15	< 0.01
	Q6.0	1.0	15	0.01
5/8	Q4.0	1.0	15	0.35
	Q4.1	0.5	15	0.28
	Q4.2	0.5	15	0.31
	Q5.0	1.0	15	0.11
	Q5.1	0.5	15	< 0.01
	Q6.0	1.0	15	0.09
1/2	Q4.0	1.0	15	> 0.50
	Q4.1	0.5	15	> 0.50
	Q4.2	0.5	15	> 0.50
	Q5.0	1.0	15	0.23
	Q5.1	0.5	15	< 0.01
	Q6.0	1.0	15	0.22

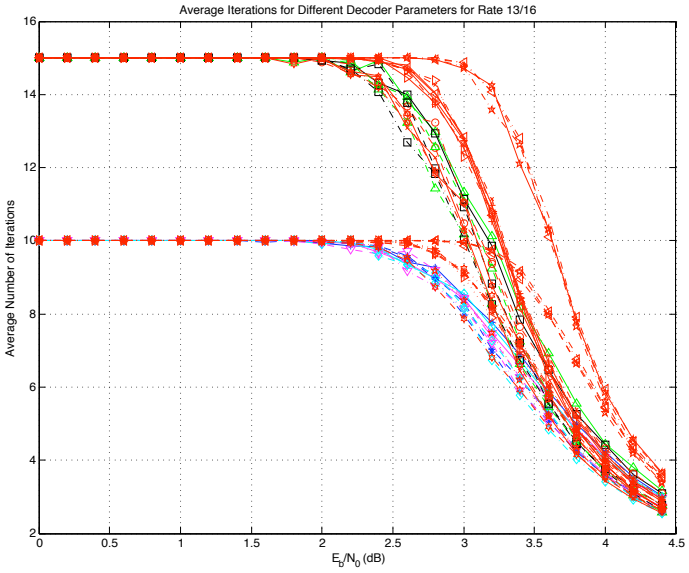
Allowing a maximum implementation loss of 0.25dB for any of the code rates, the best combination of parameters that gives the minimum wordlength is Q5.0 quantization with $\beta = 1$ and $I_{max} = 15$.



(a)

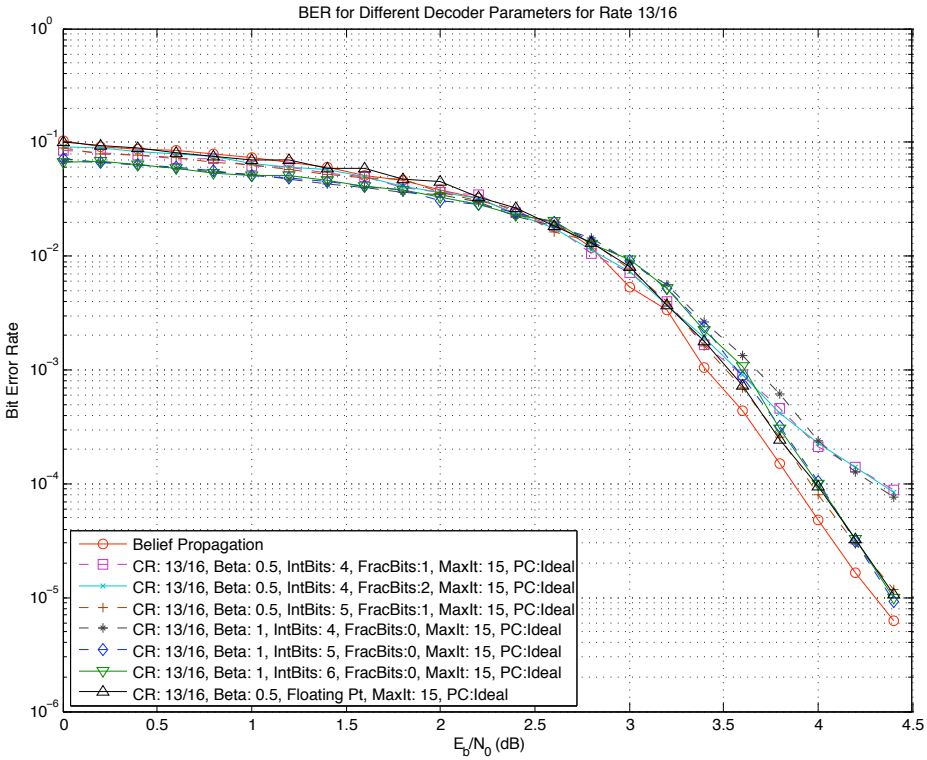


(b)

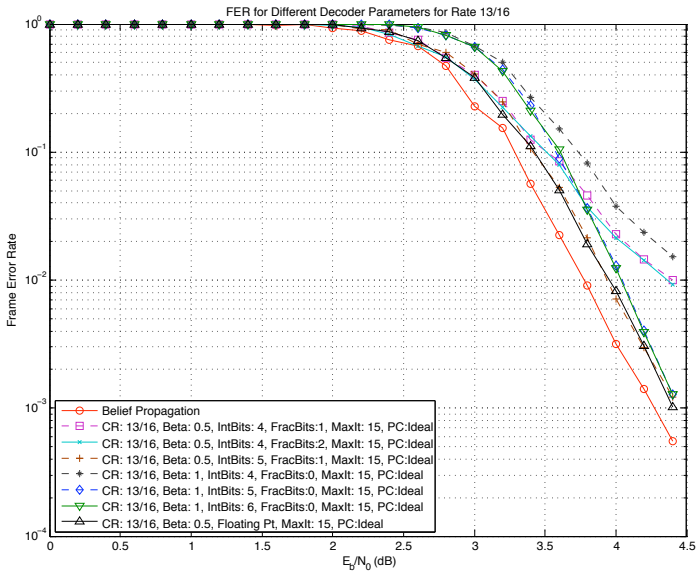


(c)

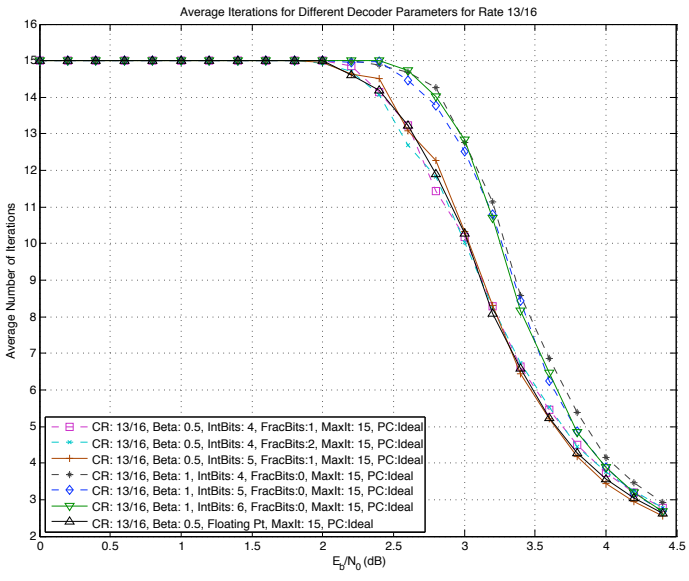
Figure 5.5: Simulated performance for all parameter combinations for the 802.11ad rate 13/16 code : (a) BER; (b) FER; (c) average iterations.



(a)

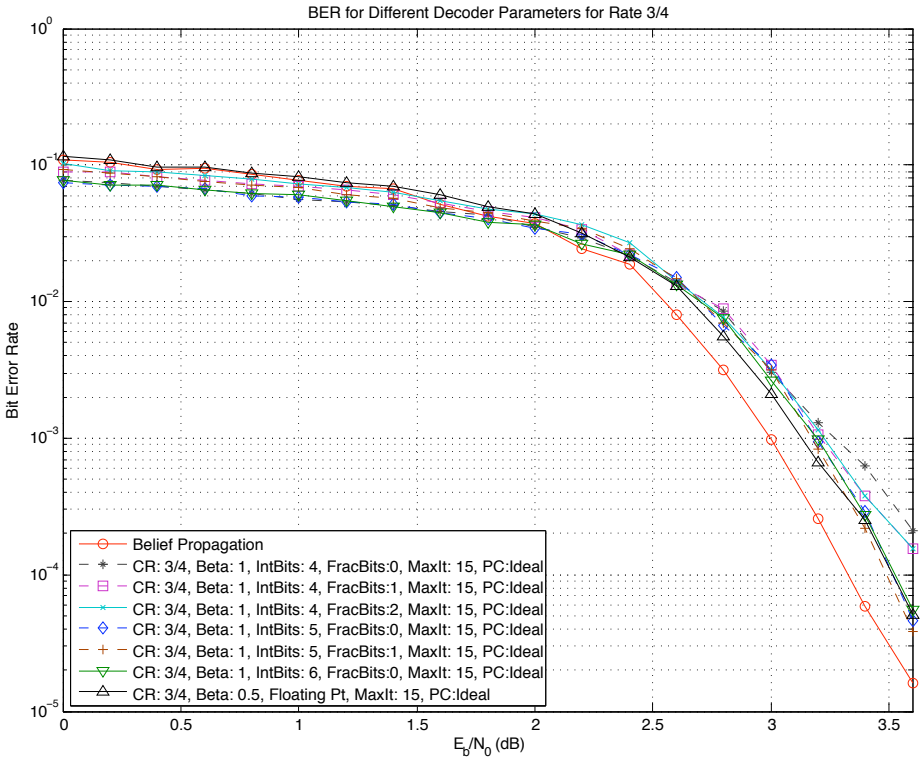


(b)

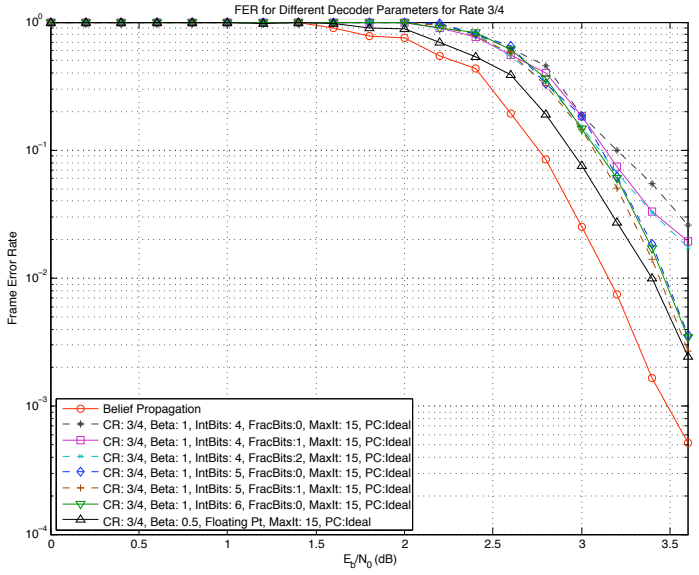


(c)

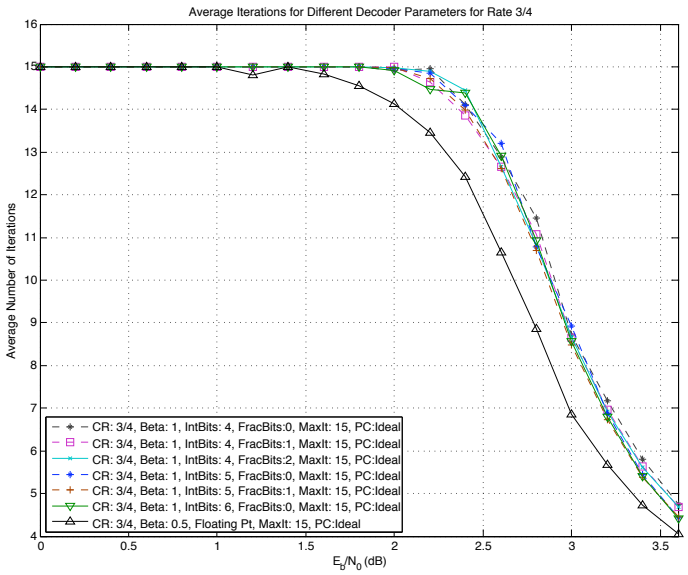
Figure 5.6: Simulated performance for the best parameter combinations for the 802.11ad rate 13/16 code : (a) BER; (b) FER; (c) average iterations.



(a)

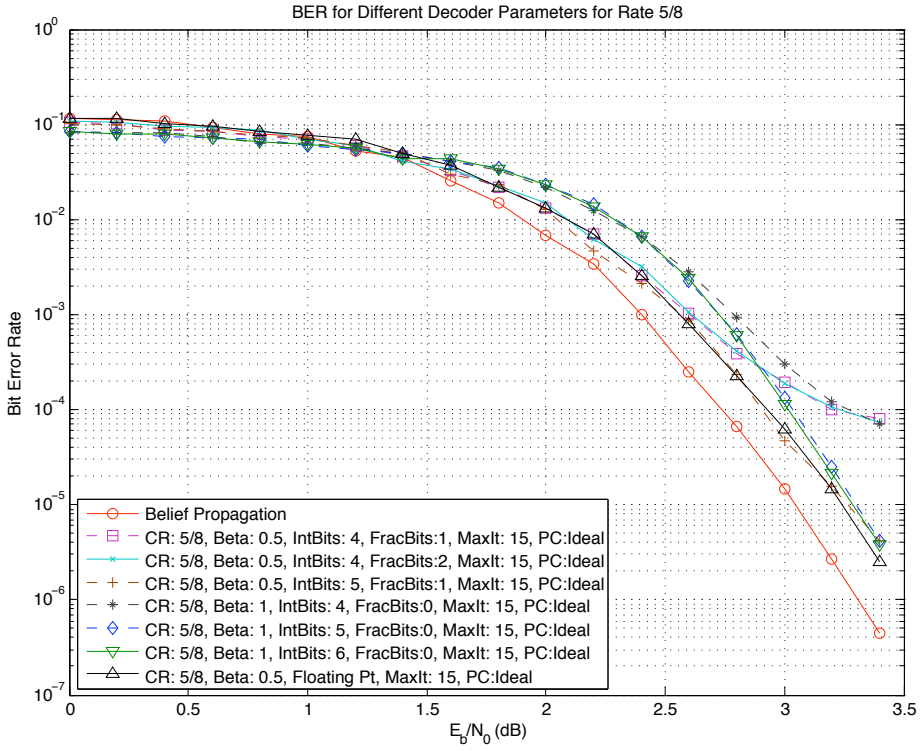


(b)

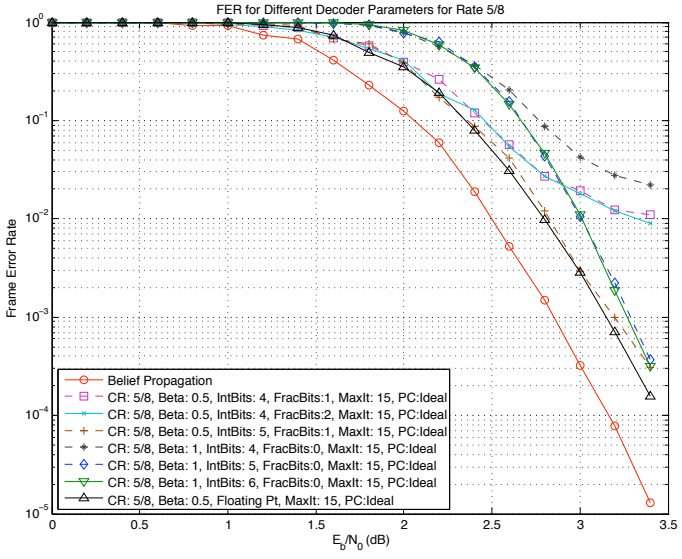


(c)

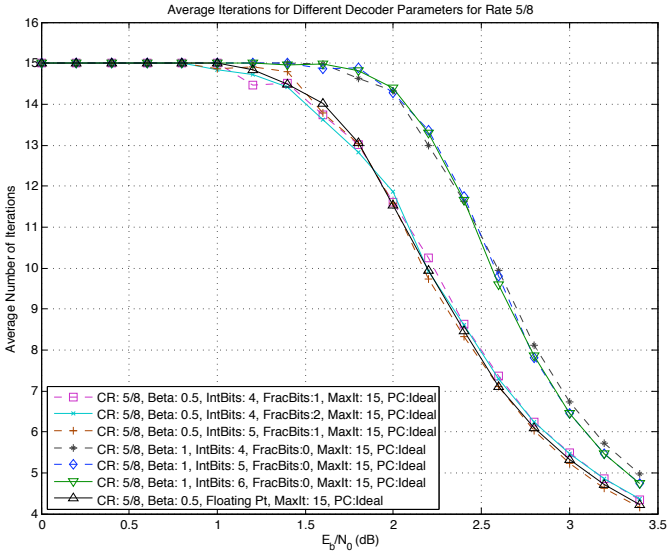
Figure 5.7: Simulated performance for the best parameter combinations for the 802.11ad rate 3/4 code : (a) BER; (b) FER; (c) average iterations.



(a)

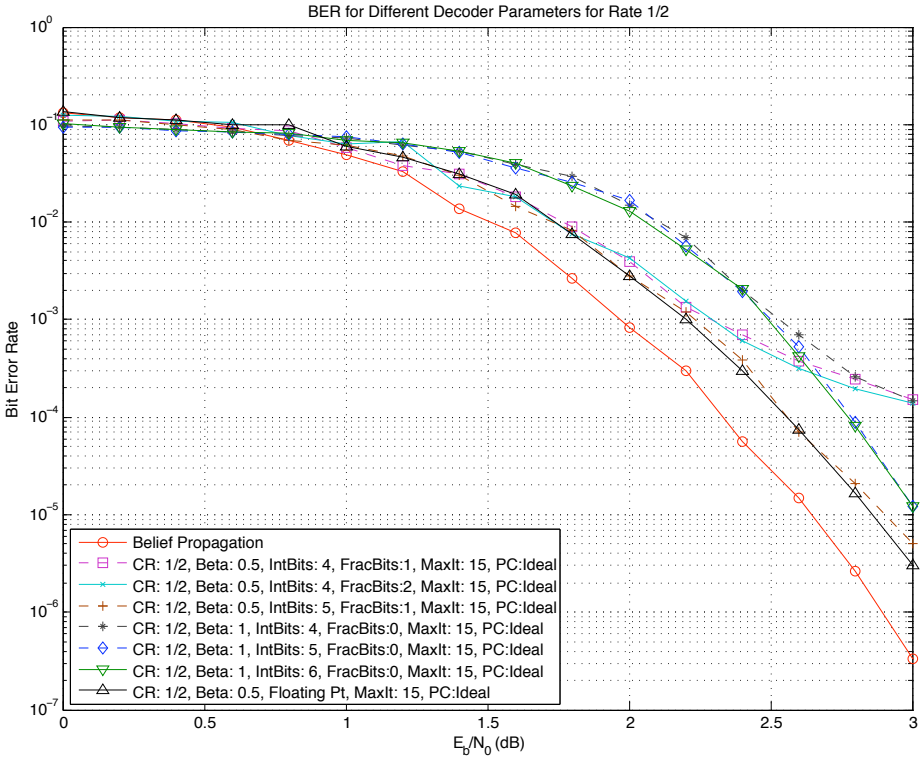


(b)

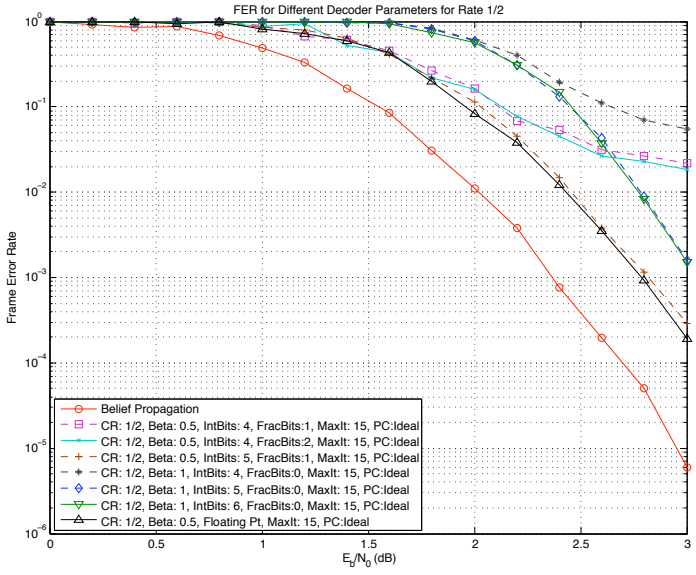


(c)

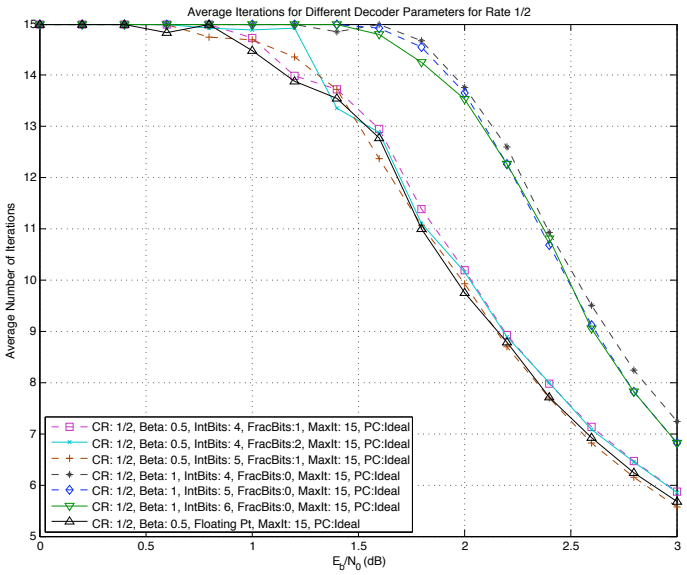
Figure 5.8: Simulated performance for the best parameters combinations for the 802.11ad rate 5/8 code : (a) BER; (b) FER; (c) average iterations.



(a)



(b)



(c)

Figure 5.9: Simulated performance for the best parameter combinations for the 802.11ad rate 1/2 code : (a) BER; (b) FER; (c) average iterations.

5.2.2 Architecture Selection

The architecture selection proceeds according to the method outlined in Chapter 4.4:

- **Step 1: List out all feasible combinations of basic parameters.** Table 5.4 lists the combinations of parameters that were considered for the 802.11ad decoder design. Serialization factors (SF) indicate the level of parallelization.

Table 5.4: Combinations of design parameters to consider in the architecture exploration.

SF_{VN}	SF_{CN}	Pipeline Depths	Frames
1	1	1-15	1
		2-15	2
		3-15	3
		4-15	4
		5-15	5
1	2	1-15	1
		3-15	2
		5-15	3
1	4	1-15	1
		5-15	2
		9-15	3
2	4	1-15	1
3	4	1-15	1
6	4	1-15	1
7	4	1-15	1

- **Step 2: Estimate the pipeline for each combination of parameters.** All pipelines with a depth from 1 to 15 were created for the combinations of parameters in Table 5.4.

- **Step 3: Estimate the worst-case minimum frequency for each combination.** Table 5.5 summarizes the results of applying Equation 4.1 to all the combinations of parameters in Table 5.4.

- **Step 4: Eliminate architectures above minimum operating frequency limit.** The upper limit on the minimum operating frequency was set to 500MHz based on initial estimates of power obtained through synthesis of the basic building blocks. Also, those pipelines that could not run at the required frequency for their pipeline depth were eliminated. For example, the $SF_{VN} = 3$, $SF_{CN} = 4$ design cannot possibly run at 440MHz with only one pipeline stage because the logic delay through the entire decoder is too long, and any of the reasonable length pipelines have an $f_{min} > 500$MHz. After elimination, only the designs with $SF_{VN} = 1$ and $SF_{VN} = 2$ remain.

Table 5.5: Minimum frequency (MHz) for the combinations of VN and CN serializations, pipeline depths, and number of frames processed for the medium throughput operating class.

SF_{VN}	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**2**	**3**	**6**	**7**
SF_{CN}	**1**	**1**	**1**	**1**	**1**	**2**	**2**	**2**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
Frames	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**1**	**1**	**1**
1 Stage	37					73			147			293	440	880	1027
2 Stage	73	37				110			183			298	445	885	1031
3 Stage	110	55	37			147	73		220			335	449	889	1036
4 Stage	147	73	49	37		183	92		257			371	486	894	1040
5 Stage	183	92	61	43	37	220	110	73	293	147		408	523	898	1045
6 Stage	220	110	73	55	44	257	128	86	330	165		445	559	903	1050
7 Stage	257	128	86	64	51	293	147	98	367	183		481	596	940	1054
8 Stage	293	147	98	73	59	330	165	110	403	202		518	633	976	1091
9 Stage	330	165	110	83	66	367	183	122	440	220	147	555	669	1013	1128
10 Stage	367	183	122	92	73	403	202	134	477	238	159	591	706	1050	1164
11 Stage	403	202	134	101	80	440	220	147	513	257	171	628	743	1086	1201
12 Stage	440	220	147	110	88	477	238	159	550	275	183	665	779	1123	1238
13 Stage	477	238	159	119	95	513	257	171	587	293	196	701	816	1160	1274
14 Stage	513	257	171	128	103	550	275	183	623	312	208	738	853	1196	1311
15 Stage	550	275	183	138	110	587	293	196	660	330	220	775	889	1233	1348

Table 5.6: Area estimates for decoder options remaining after f_{min} pruning.

SF_{VN}	1				2
SF_{CN}	1	2	4		
Frames	1	1	1	2	1
Area (mm^2)	3	2	1	1.3	0.4

- **Step 5: Find the power and area for the basic blocks.** Table 5.6 summarizes the approximate area for the overall decoders with their minimum pipeline depth that had an area less than or equal to $3mm^2$, which is a fairly large area for only the logic gates. Cutting all designs with a logic area greater than or equal to $2mm^2$, chosen based on experience to be too large, leaves only three combinations of SF_{VN}, SF_{CN}, and F. All allowable pipeline depths for these combinations must still be explored. The $SF_{VN} = 1$, $SF_{CN} = 4$ decoder that processes two frames simultaneously with five pipeline stages has the lowest power of the remaining decoder designs and becomes the chosen design. Note that this uses the minimum number of pipeline stages for this level of VN and CN parallelization because it gives enough timing slack to allow full voltage-frequency scaling. Any designs that give more slack than necessary for voltage-frequency scaling cannot be optimal since it just adds extra register power.

5.3 Functional Design

The decoder has 672 VNs that are combined into 16 groups of 42VNs. Each VN within this group connects to one port of a 42-input barrel shifter to implement the submatrix shifts. The outputs of the barrel shifter are further routed by the pre-routers, which connect to one of the 16 inputs of the 42 CNs. The outputs of each CN go through inverse shifting using post-routers and another set of barrel shifters. The design has several levels of hierarchy in order to keep irregular wiring local and make the global wires as regular as possible. Figure 5.10 shows the high-level connection of all the blocks.

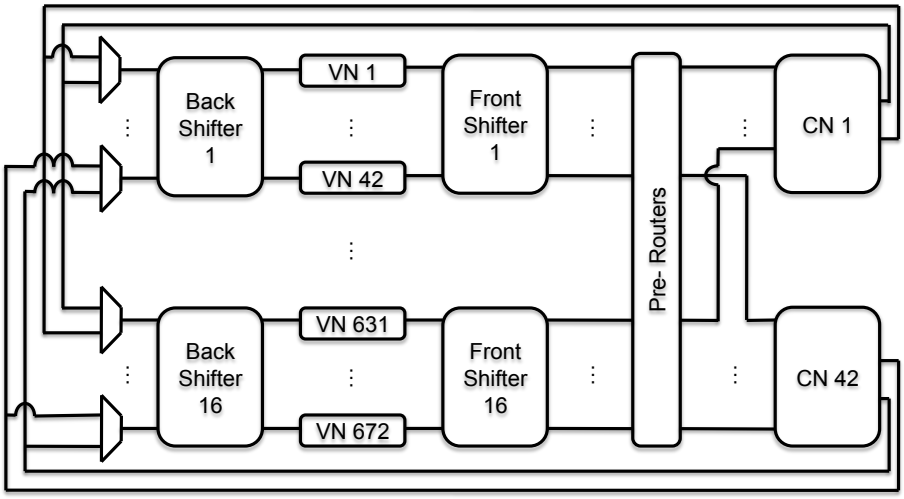


Figure 5.10: Block diagram of decoder.

Since the design serializes the CNs, they are time-multiplexed to act as different CNs in each cycle. For example, in one cycle, each VN sends a single message that has been marginalized for the CNs in the first layer. The CN has all of the information it needs to compute the new check to variable (C2V) message from all CNs in the first layer, and, after it has finished processing the inputs, it sends back a single message to all neighboring VNs. In the next cycle, the VNs send another single message that has been marginalized for the CNs in the second layer, so the same CNs can compute the C2V message from the CNs of the second layer. This continues for all the layers in the matrix, with serial messages being passed back and forth from VNs to CNs.

The decoder uses the flooding schedule because layered decoding has too many dependencies to be used effectively in a highly parallel, fully pipelined design. It processes one layer per pipeline stage, and the VNs accumulate one frame's messages while sending out the other's.

5.3.1 Variable Nodes

When starting to decode a new data frame, the VN loads in a prior value to either the first or second frame's prior and accumulation registers. Over the next three to four cycles (depending on the code rate), it sends variable to check (V2C) messages to its neighboring CNs in sign/magnitude format. Since the VN performs the C2V marginalization, it locally stores the marginalized V2C messages it just sent out in a shift register. Once the unmarginalized C2V message returns in sign/magnitude format, the VN marginalizes the magnitude by comparing the first minimum magnitude to the V2C magnitude from the shift register. If they match, the VN uses the second minimum magnitude; otherwise, it uses the first minimum magnitude. Then, it performs the β min-sum correction on the marginalized C2V magnitude with the optimal value $\beta = 1$ using an unsigned subtractor that saturates at zero. At the same time, the VN marginalizes the C2V sign by adding modulo two the V2C sign from the shift register and the incoming C2V sign. The sign and magnitude of the marginalized C2V are concatenated and converted to two's complement.

The VN accumulates the marginalized, corrected C2Vs, with the prior value added in when the first C2V arrives, and it also stores them individually in another shift register. After all marginalized C2V messages have been accumulated, the VN calculates the new V2C messages by subtracting out the saved C2V values from the accumulated value, which is the V2C marginalization. The VN converts the marginalized V2C to sign/magnitude format and sends it out to the VN's neighboring CN in the current layer. Also, the sign bit of the accumulated value is used as the hard decision and for early termination. Figure 5.11 shows a detailed schematic of the VN.

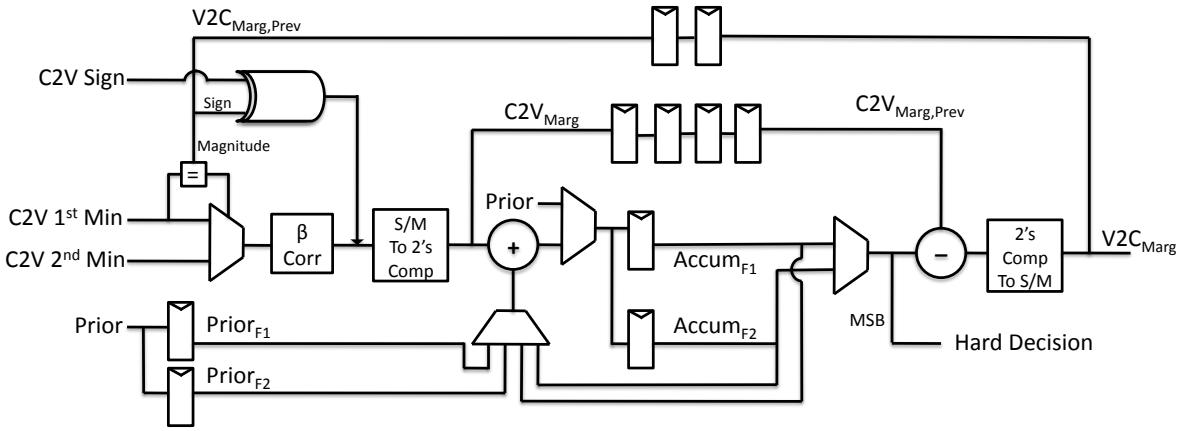


Figure 5.11: Variable node architecture.

5.3.2 Barrel Shifters

The front shifters implement right shifts, and the back shifters implement left shifts so that both can use delayed versions of the same shift amount control signals. Both have 42 inputs, which translates to six stages of 2:1 muxes. Since some matrices are all-zeros, VNs must pass dummy messages to the CNs and vice versa so that the decoder's performance does not degrade. However, neither the VNs nor CNs know when they process an all-zeros matrix since only the barrel shifters use the shift amounts. By encoding the all-zeros matrix as an unused shift value, the barrel shifters output a dummy message whenever they see the special shift value. This requires an extra stage of 42 of muxes at the end of each barrel shifter that selects between the shifted output or the dummy output. For the front barrel shifters, the dummy output equals all maximum values, which equals 15 for 5-bit sign-magnitude, because this will not affect finding the first and second minima in the check node. For the back barrel shifters, the dummy output equals all zeros because it does not affect the final sign or the accumulation result.

For the fully parallelized VN and layer serialized CN architecture, the decoder needs 16 front and back barrel shifters, one for each submatrix in a layer of the parity check matrix **H**.

5.3.3 Variable Node Group

When synthesizing a design, keeping closely related modules within a single level of hierarchy helps reduce routing overhead and increases the place and route (PAR) tool's efficiency. Groups of 42 VNs that belong to the same submatrix column in the **H** matrix only connect to the front and back barrel shifters, so they are placed into a larger group with both barrel shifters, called a variable node group (VNG). The PAR tool places the VNs and shifters next to each other and uses only short wires. If they were not in the same level of hierarchy, there is no guarantee that the tool would choose such a favorable placement. VNs 1-42, back shifter 1, and front shifter 1 in Figure 5.10 make up one of the 16 VNGs in the decoder.

5.3.4 Check Nodes and Check Node Group

To accommodate the higher rate codes, the CN has 16 inputs, and, as explained in Chapter 4.2, the CN must output the results from the second-to-last and last stages of the magnitude and sign computation trees in order to allow processing of two non-overlapping layers. To be compatible with the post-routing, which accepts two pairs of first and second minima, the CN unconditionally outputs the results from the bottom half of the trees and selects between the final stage's and

top-half's results for the other output. Figure 5.12 shows the block diagram of the check node's magnitude computation tree, and Figure 5.13 shows the implementation of the sort and compare-select (CS) blocks. Since the chain of comparators form the critical path in the CN, each comparator is implemented as a tree comparator. Although this increases power slightly (the tree structure has more logic, but there are few CNs compared to VNs and only two stages in the tree), it decreases the critical path significantly.

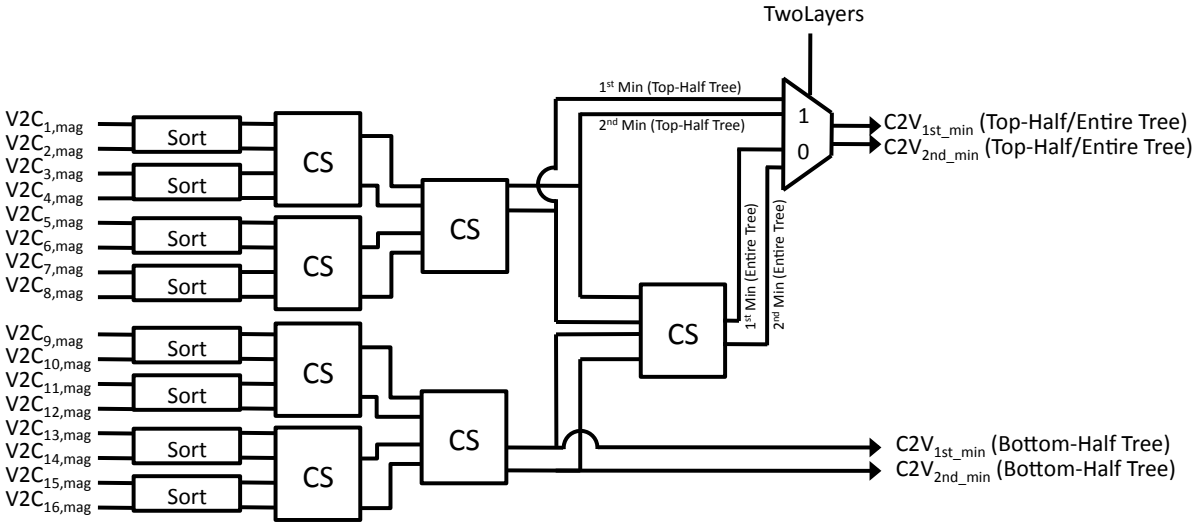


Figure 5.12: Check to variable message magnitude computation.

A separate tree of XOR gates computes the product of the inputs' sign as required by the last term in Equation 2.9. This tree does not affect the critical path and has a straightforward design as shown in Figure 5.14.

Since the design is layer serialized, the decoder has as many CNs as rows in a submatrix, which equals 42. The 42 CNs are grouped together in a single layer of hierarchy called the check node group (CNG) in order to simplify routing.

5.3.5 Pre- and Post-Routing

Since all of the codes have at most two layers in the groups of non-overlapping layers, the pre- and post-routers must only ensure the first layer's inputs go to the top half of each CN and the second layer's inputs go to the bottom half. Figure 5.15 demonstrates the function of the pre- and post-routers for a portion of the rate 1/2 matrix.

Pre-routers have a simple structure, but the routing from the barrel shifters to the pre-routers is

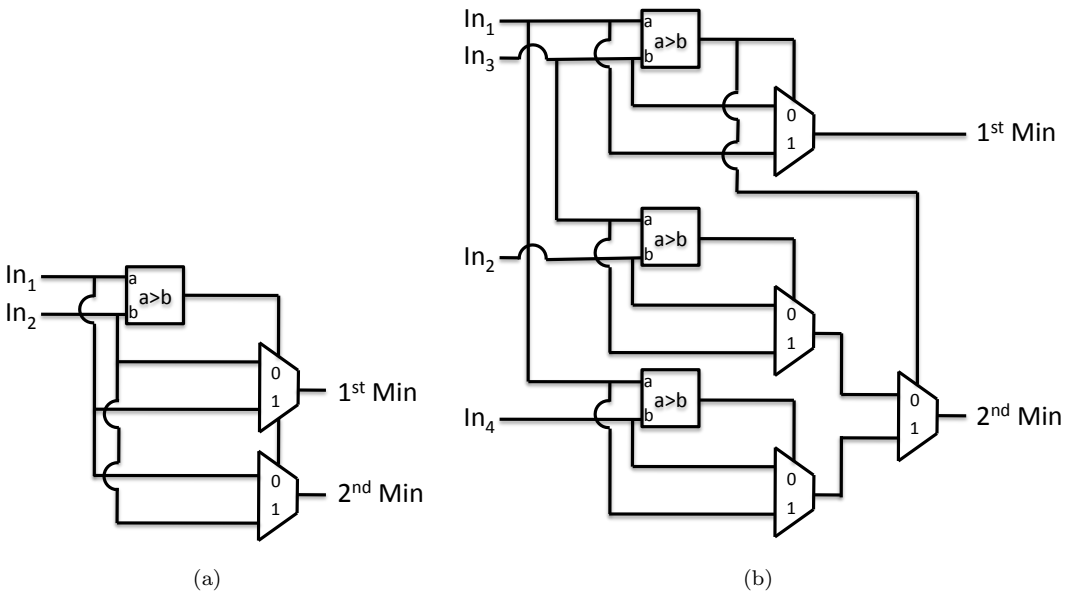


Figure 5.13: Structure of check node building blocks : (a) sort; (b) compare-select.

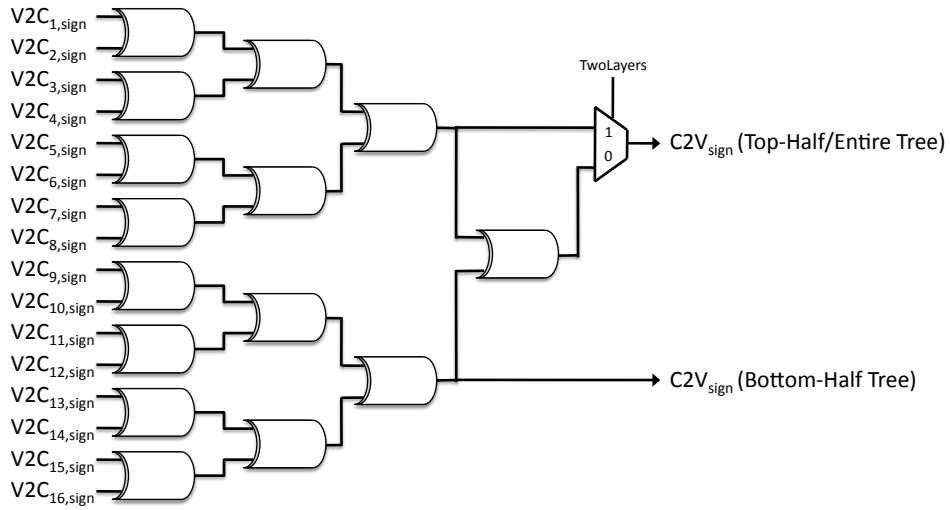


Figure 5.14: Check to variable message sign computation.

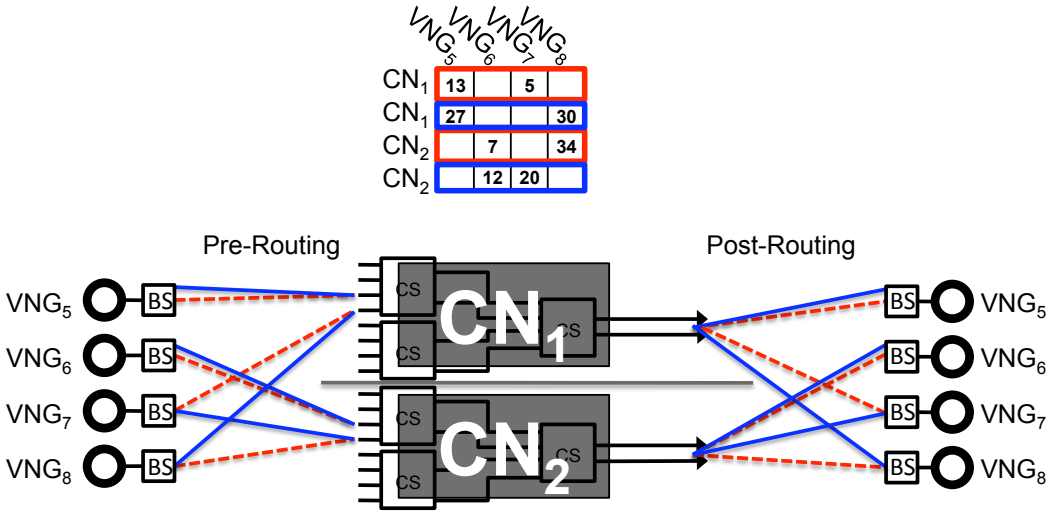


Figure 5.15: Illustration of pre- and post-routing for a section of the rate 1/2 matrix.

complex. Looking at the pairs of non-overlapping layers in the lower rate 802.11ad matrices, groups of two consecutive submatrices come from different layers, except for a small number of special cases. Figure 5.16 illustrates this with the rate 1/2 matrix, where the pairs of non-overlapping layers have been moved next to each other. Each group of 2x2 submatrices has degree one, except for the two highlighted cases where the matrix has degree two. Focusing on the first case, the pre-routers for

40		38		13		5		18							
	36		31		7		34		10	41					
34		35		27			30	2	1						
	27		18		12	20				15	6				
35		41		40		39		28			3	28			
	31		23		21		20			12			0	13	
29		0			22		4		28		27		23		
	22		34	31		14		4				13		22	24

Figure 5.16: 802.11ad rate 1/2 matrix with pairs of non-overlapping layers moved next to each other, 2x2 groups of submatrices boxed, and the non-degree one groups highlighted.

each CN input on the top half of the tree must choose between one of the two submatrices in its corresponding 2×2 group (the one that comes from the top layer), which requires a 2:1 mux. The pre-routers for each CN input on the bottom half of the tree have the same inputs, but make the opposite choice. Now, taking into account the special cases, a 4:1 mux replaces the 2:1 mux so that each group can select one of the current 2x2 group's columns, the previous group's second column, or the next group's first column. To be concrete, consider the highlighted submatrices in Figure

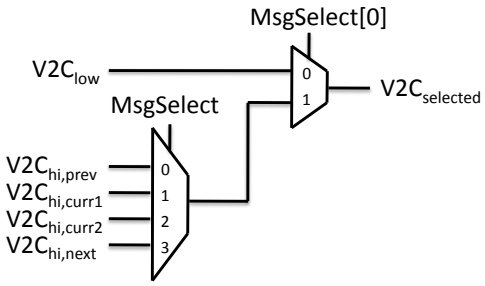
5.16. All 2x2 groups up to the special cases have degree one, so they can use 2:1 muxes (or 4:1 muxes so that the decoder can extend to other matrices that have more of these special cases). The pre-routers for the first highlighted group choose the first column in the current group and the next group's first column so that it can route the first selected column to the top of the CN and the other selected column to the bottom of the CN. Likewise, the pre-routers for the next group select the second column of the previous group and the second column of the current group so that it can send the previous group's column to the top half of the CN and the current group's column to the bottom half of the CN. To handle the matrices without non-overlapping layers, a 2:1 mux chooses between the 4:1 mux's output and the current column's output. Figure 5.17 shows both the pre-router's construction and how wires are routed to and from it for a single CN. All other CNs have essentially duplicates of the hardware and routing, except they use different outputs of the front barrel shifters (CN i uses only the i^{th} output of each front barrel shifter). A controller computes the control signals based on the shift amount and row position of the submatrix, which are stored in the matrix memory, and the decoder has a total of 672 pre-routers, one for each individual CN input.

Both the structure and routing for the post-routers, shown in Figure 5.18 is fairly straightforward compared to that of the pre-router. Each CN has its own post-router, which has 16 2:1 muxes that choose the correct C2V to send to each neighboring VN. The outputs of CN1's post-routers connect to the first input of all back barrel shifters, since CN1 takes the first input from all front barrel shifters. The rest of the CNs have corresponding routing. Like the pre-router, a controller computes the control signals based on the shift amount and row position of the submatrix. The decoder has a total 42 post-routers, one for each CN.

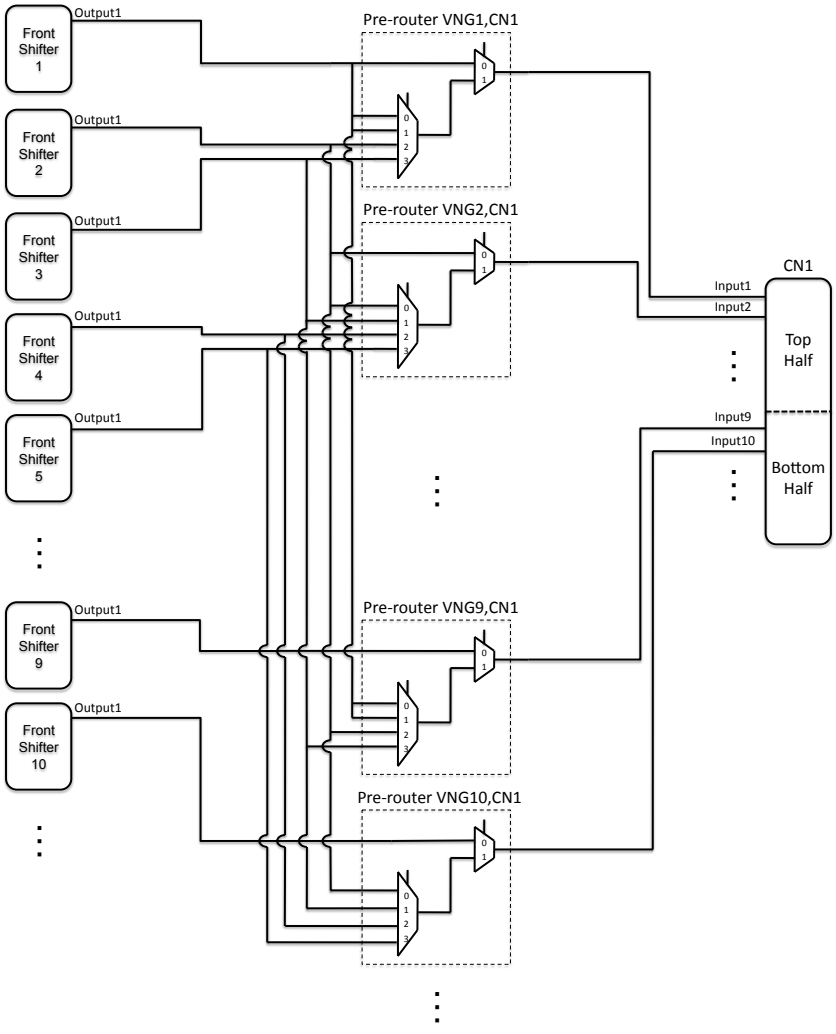
As seen in Figures 5.17(b) and 5.18(b), the routing between the pre-routers and CNs and between the CNs and post-routers is regular, but the routing between the VNs and the routers is irregular. In order to keep the irregular routing local, the pre- and post-routers are included in the same level of hierarchy as all of the variable node groups. This makes the global wiring between the routers and CNs regular, increasing logic density and decreasing power and timing overhead.

5.3.6 Pipeline

The decoder has five pipeline stages and processes two independent data frames simultaneously. Each full iteration takes three sub-iterations for the rate 13/16 code and four sub-iterations for the rest, one for each time-multiplexed use of the check nodes. In the first stage, the VN outputs the

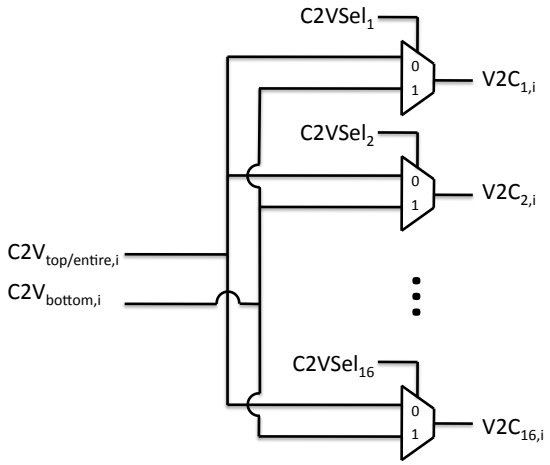


(a)

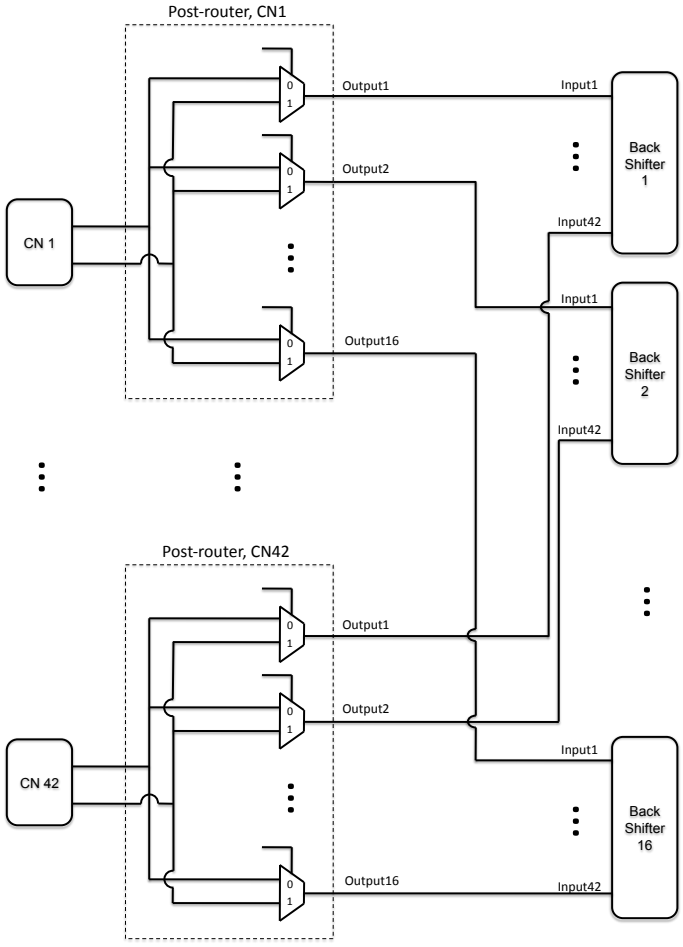


(b)

Figure 5.17: Implementation of the pre-routers required by granular check nodes : (a) structure of individual pre-routers; (b) routing between front barrel shifters, pre-routers, and CNs.



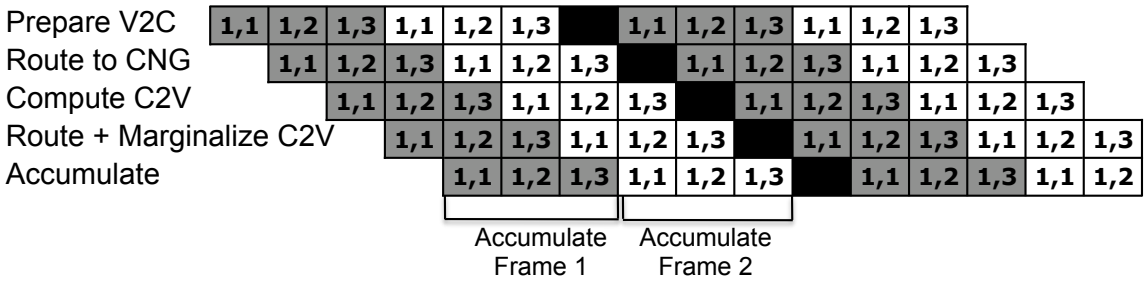
(a)



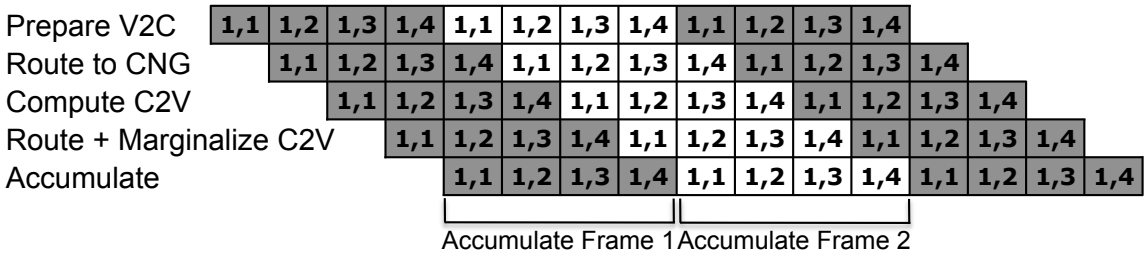
(b)

Figure 5.18: Implementation of the post-routers required by granular check nodes : (a) post-router structure for CN i; (b) routing between CNs, post-routers, and back barrel shifters.

marginalized V2C and the barrel shifter reorders the messages. The second stage consists of the pre-routing and global wiring to the check node. The CNs process their inputs in the third stage and route the messages back to the VNs across the global wires in the fourth stage. In the fifth stage, the VN accumulates the serial messages over three or four cycles. The accumulation would normally cause a four cycle bubble in the pipeline due to the dependency between accumulating all the C2Vs and sending out the next V2Cs. With the chosen decoder parameters, the bubble is just large enough to accommodate processing a second frame. This reduces the bubble to one cycle for the rate 13/16 code and removes it completely for the other rates because no dependencies exist between the two frames. Figure 5.19 gives the pipeline diagram for the decoder with three sub-iterations and four sub-iterations. The values inside each element before the comma indicate the VN sub-iteration number, the values after the comma indicate the CN sub-iteration number, and the shading indicates the frame number. Blacked out elements indicate unused cycles.



(a)



(b)

Figure 5.19: Pipeline diagrams for the (a) rate 13/16 code and (b) rate 1/2, 5/8, and 3/4 codes.

5.3.7 Other Blocks

Matrix Memory

To allow the CN granularity in this architecture, the matrix memory stores the shift amounts of the compressed matrix, meaning shift amounts of the non-overlapping rows are compressed into the same row in memory. For the rate 1/2 and 5/8 codes, the memory stores the matrices on the bottom of Figure 5.4, and for the rate 3/4 and 13/16 codes, the memory stores the standard matrices in Figure 5.1. This has the benefit of eliminating some of the wasted memory on the all-zeros matrices and reducing the overall amount of memory. The barrel shifters use the submatrix shift amounts, with the right-shifting front barrel shifters using the value immediately and the left-shifting back barrel shifters using a delayed version (matched to the pipeline delay) so that the back shifter performs the inverse shift of the front shifter. To control the pre- and post-routers, the architecture requires another small memory to store whether the submatrix comes from the top non-overlapping layer or the bottom non-overlapping layer, where top is stored as a 0 and bottom as a 1. If a layer does not have a corresponding non-overlapping layer, the memory stores all zeros for that layer's row, which is arbitrary because the controller does not use the row position memory for these layers. This costs only one extra bit of memory for every submatrix, so the total memory is still smaller than storing all submatrix shift amounts for the largest matrix. A controller uses these values to perform the appropriate selection in the pre- and post-routers. For the rate 1/2 code, the controller always uses the row position memory, and, for the rate 3/4 and 13/16 codes, it never uses this memory. For the rate 5/8 code, the controller uses a counter that keeps track of the compressed layer being processed to decide whether to use the row position memory (first two layers) or not (last two layers). Based on this discussion, the decoder stores a 4×16 matrix of 6-bit shift amounts and another 4×16 binary matrix of row positions. Due to the small storage requirement, the memory is implemented with flip-flops. A user can stream bits into a dedicated scan chain to program both the shift amount and row position memories.

Early Termination

LDPC decoders can terminate decoding before the maximum number of iterations has been reached by detecting early convergence. To do this exactly, calculate

$$\mathbf{s} = \mathbf{d}\mathbf{H}^T \tag{5.1}$$

where $\mathbf{s}$ is the syndrome, $\mathbf{d}$ is the decoded vector after some number of full iterations, and $\mathbf{H}$ is the parity check matrix. $\mathbf{s}$ will contain all zeros if $\mathbf{d} \in C$, where C is the codeword space; otherwise, $\mathbf{s}$ will have non-zero elements. To test if $\mathbf{d}$ satisfies all parity checks, meaning $\mathbf{d} \in C$, a condition denoted by p, use the decision rule

$$p = \begin{cases} \text{true}, & \text{if } \sum_i d_i = 0 \\ \text{false}, & \text{if } \sum_i d_i > 0 \end{cases} \tag{5.2}$$

If an LDPC chip has both an encoder and decoder, which occurs often, the decoder can feed the information portion of $\mathbf{d}$ to the encoder and compare the output parity bits to those of $\mathbf{d}$. The two sets of parity bits will only match if $\mathbf{d} \in C$. Note that the encoder usually does not run while the decoder runs, so the encoder would be free during decoding.

For some LDPC chips, it may not be realistic to have an encoder as well as a decoder on the same chip, such as a decoder test chip or because of area and power constraints. In the first case, it is possible to emulate an encoder because the user has control over the input sequences to the chip. By selecting a single codeword to test, which is valid because all codewords are symmetric in the AWGN channel with BPSK or QPSK modulation, the decoder looks for an exact match to the selected test codeword. For example, selecting the all-zeros codeword as the test codeword, the decoder checks to see if the sum of the elements of the hard decision vector is zero, which implements Equation 5.2 for a specific codeword. This is a bit more idealistic than the ideal parity check because it only converges if the exact codeword is reached instead of any codeword within C, but the probability of decoding to an incorrect codeword is negligible at realistic E_b/N_0.

For non-test chips that do not have an encoder, the designer can still implement an exact parity check, although it requires a reasonable amount of hardware. Alternatively, methods exist that approximate parity checks or use heuristic rules to detect possible convergence. First, instead of checking the unmarginalized V2C signs (equivalent to the hard decisions) for parity, check the sign of the marginalized V2Cs. The CNs already check this parity, so this approximate check requires no extra hardware [4]. However, there is no guarantee that it correctly detects convergence, and, through simulation, it has been found to only work well for high rate codes. Second, the decoder can check if the current iteration's hard decision is the same as that from the previous iteration. If no bits flip, it is possible that the decoder has converged [23]. Through simulation, this has been found to only work for very low E_b/N_0; it cannot be used at the error rates required by many standards.

This chip has options to either use the test-chip method with the all-zeros codeword or approxi-

mate parity check method. See Chapter 5.3.8 for more on how the chip uses these early termination methods.

5.3.8 Test Structures

The decoder has both manual and automated test functionality. The manual version tests specific patterns to isolate errors, and it can model any channel statistics desired, from AWGN to partial response. On the other hand, the automated version is used to quickly obtain performance results and realistic power estimates, but only tests the all-zeros codeword under AWGN channel assumptions.

Manual Testing

The manual version begins by loading a shift register with a sequence of noisy priors generated off-chip by a script. Since this is done in software, the input priors can be an arbitrary codeword distorted by any channel model desired. The decoder then processes the priors and outputs the decoded bits using a different shift register. Concurrently, the script runs the same codeword through a C++ model of the decoder with the same quantization and maximum number of iterations. When both results are available, the script compares the output sequence from the decoder to the output from the model and records any sequences that do not match. The decoder cannot use the all-zeros codeword early termination method since arbitrary codewords are inputted, but it can use the approximate method or no early termination. Due to pad constraints, a single shift register loads all the priors serially, and a second shift register outputs all the hard decisions serially. This limits the speed of the manual tests, making it less useful for obtaining performance results, especially for lower BER levels.

Automated Testing

In contrast to the manual testing procedure, the automated version performs tests rapidly but only gives overall performance in terms of BER, FER, and the average number of iterations per frame, not the individual sequences that have errors. The only codeword used for automated testing is the all-zeros codeword because an encoder would be needed on chip to encode a string of i.i.d. bits and get a valid codeword. Since all codewords are symmetric under the assumption of i.i.d. input bits, using the all-zeros codeword gives valid performance results.

To run the tests, a script serially streams bits into a scan chain to set the noise variance σ^2, the number of frame errors to record before ending the test, the maximum number of frames to decode,

the maximum number of iterations per frame, and the code rate. A separate scan chain is used to load the matrix memories so that whenever the parameters of the test change, the shift amounts and row positions do not need to be reloaded. Next, testing begins by generating the priors using AWGN noise generators based on the Box-Muller algorithm and arithmetic circuits to adjust the E_b/N_0 and calculate the log-likelihood ratios according to Equation 2.1a. The designs for the noise generators, E_b/N_0 adjustment circuits, and log-likelihood calculation circuits are the same as that used in [26]. Note that the design does not match the Gaussian distribution accurately past 4.8σ, but that is not required for accurate simulation because, at high E_b/N_0, the errors are dominated by absorbing sets, not outliers generated by the Gaussian distribution [26].

Four prior generation circuits connect to the prior scan chain (also used in manual testing) at equidistant locations, which were the maximum number that could fit on the chip in a reasonable area. Since the priors used for one codeword will not be flushed out of the shift register before the next codeword is loaded, the noise between consecutive codewords will be correlated. In other words, some of the priors used in one codeword will be used for different bits in the next codeword. However, this does not affect the accuracy of the tests as verified through simulation. To detect early convergence, the decoder can use either early termination mechanism explained in Chapter 5.3.7. Using the all-zeros method emulates having an encoder on-chip that would check exact parity, whereas the approximate method would go on a chip that cannot afford an encoder. If no early termination is wanted, both blocks can be disabled so that the decoder only terminates after a fixed number of iterations. After decoding a frame finishes, the decoder calculates the number of bit errors by summing the number of ones in the hard decision (because the input codeword is always all-zeros) and records whether or not there was a frame error. These values are added to running sums of the bit errors and frame errors, respectively, and the number of iterations is added to its own running sum. When the decoder either records the specified number of frame errors or reaches the maximum number of frames to decode, it stops and outputs the total number of bit errors, frame errors, iterations, and frames tested using another shift register. The script interfacing with the chip records these values, loads new parameters into the scan chain, and continues with the next test.

5.4 Results

To achieve a 3Gb/s throughput, the decoder must operate at 150MHz for the rate 1/2 code, which requires the highest frequency out of all the codes because it takes the most iterations to converge. In order to accommodate voltage/frequency scaling, the decoder was synthesized at 200MHz to allow

for at least a 25% reduction in supply voltage. Any extra slack allows for extra voltage/frequency scaling.

When synthesized in a low-power 65nm CMOS technology at 1.2V and 200MHz using Synopsys' Design Compiler, the reported power consumption for the decoder is 213mW, the area is 1.3mm^2, and the design has a slack of 20% of the period. After voltage and frequency scaling down to 0.8V, the power for the 3Gb/s mode is 84mW. With pure frequency scaling, the power for the 1.5Gb/s mode is 42mW, although there might be extra voltage scaling possible. Table 5.7 gives the details of the synthesis results.

Table 5.7: Results of the 802.11ad LDPC decoder synthesis.

Technology	ST 65nm low-power CMOS	
Core Area	1.3mm^2	
Operating Class	1	2
Decoding Throughput	1.54Gb/s	3.08Gb/s
Clock Frequency	75MHz	150MHz
Supply Voltage	0.8V	0.8V
Power Consumption	42mW	84mW

Almost 60% of the power goes into the variable nodes because they are the most complicated and numerous objects in the design. Explicit pipeline registers registers use half (20%) of the remaining power, and the front and back shifters and check nodes consume around 4-6% of the overall power each. The pre- and post-routers use negigible portion of the decoder's power, less than 4% together, which shows the extra overhead added to allow granular check nodes costs little in the final design (since timing was not affected significantly either). The relative power consumed by pipeline registers is misleading because the variable node contains many implicit pipeline registers. When they are taken into account, pipeline registers consume 67.5% of the total power, showing the cost of using a fully pipelined decoder. Table 5.8 summarizes the breakdown of the power consumption.

Table 5.8: Breakdown of power consumption in 802.11ad LDPC decoder.

Block Name	Number	Total Power (%)
Variable Node	672	59.1
Pipeline Register	4	19.5
Back Shifter	16	6.2
Front Shifter	16	4.4
Check Node	42	4.4
Pre-router	672	2.5
Post-router	42	1.4
Other Logic	-	2.5
Total Register Power	-	67.5

Chapter 6

Conclusion

6.1 Advances

This work introduced a highly-parallelized, fully-pipelined LDPC decoder architecture suitable for multi-Gb/s wireless PANs and developed a heuristic method to intelligently choose a specific architecture out of the large design space. As a proof of concept, a decoder was implemented that fully complies with the specifications of the 802.11ad standard, including supporting all four matrices for two out of three throughput levels. Since low-power applications will use only the two lowest throughput modes, the decoder was designed for these modes, although it could be modified to work for the third mode by adding an extra pipeline stage (which would increase all modes' power). Using the heuristic search, the chosen decoder has fully parallelized VNs, fully layer serialized CNs, and five pipeline stages and processes two frames simultaneously. This design was synthesized in a 65nm low-power CMOS technology and consumes 42mW at 1.54Gb/s and 84mW at 3.08Gb/s, which is the lowest power flexible LDPC decoder that has a throughput on the order of what emerging wireless PANs require.

6.2 Future Work

Although this work chooses an architecture based on a heuristic search that should yield a near-optimal design, the decoder has no guarantee of optimality. To find the optimal design, a more thorough sensitivity-based methodology needs to be developed that accounts for the following tradeoffs: (1) hardware savings from serialization versus the power increase from higher operating frequencies, (2) increased voltage-frequency scaling allowed by increased pipeline depth versus extra power from

additional pipeline registers, and (3) extra hardware and higher activity factor versus increased throughput (or lower operating frequency for fixed throughput) for processing extra frames simultaneously. One way to implement the methodology would be to build up a library characterizing the power and area of the basic building blocks as a function of frequency, supply voltage, and other important parameters, and then use the library to find the optimal architecture given the standard requirements. For instance, given the block length, submatrix size, and required throughput, the methodology outputs the optimal VN and CN serialization, pipeline depth, and number of independent frames processed. In addition, the decoder designed in this thesis will be run through a place and route tool and taped out to verify the results.

References

[1] A. J. Blanksby and C. J. Howland. A 690mW 1Gb/s 1024-b, Rate-1/2 Low-Density Parity-Check Code Decoder. *IEEE Journal of Solid State Circuits*, 37(3):404412, March 2002. 5, 15

[2] Chen, J. and Dholakia, A. and Eleftheriou, E. and Fossorier, M.P.C. and Hu, X.-Y. Reduced-Complexity Decoding of LDPC Codes. *IEEE Transactions on Communications*, 53(8):1288–1299, August 2005. 13

[3] S. Chung, G. D. Forney Jr., T. J. Richardson, and R. L. Urbanke. On the Design of Low-Density Parity-Check Codes Within 0.0045 dB of the Shannon Limit. *IEEE Communications Letters*, 5(2):5860, February 2001. 5

[4] Darabiha, A. and Chan Carusone, A. and Kschischang, F.R. Power Reduction Techniques for LDPC Decoders. *IEEE Journal of Solid State Circuits*, 43(8):1835 –1845, August 2008. 57

[5] J. Dielissen, A. Hekstra, and V. Berg. Low Cost LDPC Decoder for DVB-S2. In *Proc. Design, Automation and Test in Europe, 2006*, pages 130–135, 2006. 3

[6] Djurdjevic, I. and Jun Xu and Abdel-Ghaffar, K. and Shu Lin. A Class Of Low-Density Parity-Check Codes Constructed Based on Reed-Solomon Codes with Two Information Symbols. *IEEE Communications Letters*, 7(7):317 – 319, July 2003. 18

[7] J. L. Fan. *Constrained Coding and Soft Iterative Decoding for Storage*. PhD thesis, Stanford University, 1999. 9

[8] Fossorier, M.P.C. and Mihaljevic, M. and Imai, H. Reduced Complexity Iterative Decoding of Low-Density Parity Check Codes Based on Belief Propagation. *IEEE Transactions on Communications*, 47(5):673–680, May 1999. 13

[9] R. G. Gallager. *Low Density Parity Check Codes*. PhD thesis, M.I.T., 1963. 5

[10] K. K. Gunnam, S. S. Choi, M. B. Yeary, and M. Atiquzzaman. VLSI architectures for Layered Decoding for Irregular Codes of WiMax. In *IEEE International Conference on Communications*, pages 4542–4547, June 2007. 3

[11] Hagenauer, J. and Offer, E. and Papke, L. Iterative Decoding of Binary Block and Convolutional Codes. *IEEE Transactions on Information Theory*, 42(2):429–445, March 1996. 13

[12] Hocevar, D.E. A Reduced Complexity Decoder Architecture Via Layered Decoding of LDPC Codes. In *IEEE Workshop on Signal Processing Systems*, pages 107–112, October 2004. 12

[13] M. Karkooti, P. Radosavljevic, and J. Cavallaro. Configurable LDPC Decoder Architectures for Regular and Irregular Codes. *Journal of Signal Processing Systems*, 53:73–88, May 2008. 2

[14] F. Kienle, T. Brack, and N. When. A Synthesizable IP Core for DVB-S2 LDPC Decoding. In *Proc. Design, Automation and Test in Europe, 2005*, volume 3, pages 100–105, March 2005. 2

[15] Kou, Y. and Lin, S. and Fossorier, M.P.C. Low-Density Parity-Check Codes Based on Finite Geometries: A Rediscovery and New Results. *IEEE Transactions on Information Theory*, 47(7):2711 –2736, November 2001. 18

[16] Kschischang, F.R. and Frey, B.J. and Loeliger, H.-A. Factor Graphs and the Sum-Product Algorithm. *IEEE Transactions on Information Theory*, 47(2):498 –519, February 2001. 9

[17] C. Liu, S. Yen, C. Chen, H. Chang, C. Lee, Y. Hsu, and S. Jou. An LDPC Decoder Based on Self-Routing Network for IEEE 802.16e Applications. *IEEE Journal of Solid State Circuits*, 43(3):684–694, March 2008. 3

[18] David J. C. MacKay and Radford M. Neal. Near Shannon Limit Performance of Low Density Parity Check Codes. *Electronics Letters*, July 1996. 5

[19] M. M. Mansour and N. R. Shanbhag. Low-Power VLSI Decoder Architectures for LDPC Codes. In *Proc. International Symposium on Low Power Electronics and Design*, pages 284–289, August 2002. 5, 18

[20] G. Masera, F. Quaglio, and F. Vacca. Implementation of a Flexible LDPC Decoder. *IEEE Transactions on Circuits and Systems*, 54(6):542–546, June 2007. 3

[21] T. J. Richardson, Shokrollahi M. A., and R. L. Urbanke. Design of Capacity-Approaching Irregular Low-Density Parity-Check Codes. *IEEE Transactions on Information Theory*, 47(2):619–637, February 2001. 5

[22] Sheikh, F. and Ler, M. and Zlatanovici, R. and Markovic, D., and Nikolic, B. Power-Performance Optimal DSP Architectures and ASIC Implementation. In *Fortieth Asilomar cCnference on Signals, Systems and Computers, 2006.*, pages 1480 – 1485, November 2006. 28

[23] X. Shih, C. Zhan, C. Lin, and A. Wu. An 8.29mm^2 52mW Multi-Mode LDPC Decoder Design for Mobile WiMAX System in 0.13µm CMOS Process. *IEEE Journal of Solid State Circuits*, 43(3):672–683, March 2008. 2, 57

[24] Shu Lin and Lei Chen and Jun Xu and Djurdjevic, I. Near Shannon Limit Quasi-Cyclic Low-Density Parity-Check Codes. In *IEEE GLOBECOM '03*, volume 4, pages 2030 – 2035, December 2003. 18

[25] Vila Casado, A.I. and Griot, M. and Wesel, R.D. Informed Dynamic Scheduling for Belief-Propagation Decoding of LDPC Codes. In *IEEE International Conference on Communications*, pages 932–937, June 2007. 13

[26] Z. Zhang and Anantharam, V. and Wainwright, M.J. and Nikolic, B. An Efficient 10GBASE-T Ethernet LDPC Decoder Design With Low Error Floors. *IEEE Journal of Solid State Circuits*, 45(4):843 –855, April 2010. 3, 20, 26, 59