Configurable, High Throughput, Irregular LDPC Decoder Architecture: Tradeoff Analysis and Implementation

Marjan Karkooti, Predrag Radosavljevic and Joseph R. Cavallaro
Department of Electrical and Computer Engineering, Rice University, Houston, TX, 77005
{marjan, rpredrag, cavallar}@rice.edu

Abstract

Low Density Parity Check (LDPC) codes are one of the best error correcting codes that enable the future generations of wireless devices to achieve higher data rates. This paper presents a novel flexible decoder architecture for irregular LDPC codes that supports twelve combinations of code lengths -$648, 1296, 1944$ *bits- and code rates-* $1/2, 2/3, 3/4, 5/6$- *based on the IEEE* $802.11n$ *standard. All the codes correspond to a block-structured parity check matrix, in which the sub-blocks are either a shifted identity matrix or a zero matrix. A prototype of the LDPC decoder has been implemented and tested on a Xilinx FPGA and has been synthesized for ASIC.*

1. Introduction

Future generations of wireless devices have both high data-rate and high throughput requirements. Error correcting codes eliminate the need for re-transmission by correcting the errors that occur during transmission. Low Density Parity Check codes (LDPC) are one of the best known error correcting codes that have excellent decoding performance and high throughput.

Throughput and complexity of LDPC decoding depend on five major parameters: the number of bits in the codeword or 'block length', the ratio of the number of information bits to the codeword length or 'code rate', the complexity of computations at each processing node, the complexity of interconnection, and the number of times that local computations need to be repeated or the number of iterations. There is a trade-off between performance of the decoder and the complexity and speed of decoding. We will address these trade-offs throughout this paper in more detail.

One of the main advantages of LDPC codes is the inherent parallelism in the decoding process which can lead to a very high decoding throughput if used properly. The data dependence between neighboring nodes can be omitted by imposing a structure to the code matrix such as a block-structure [10]. This way, instead of one processing unit, several processing units - equal to the size of the sub-blocks S- can work simultaneously to reduce the decoding time, T, to T/S and hence increase the throughput by a factor of S. Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs) are very suitable for designing the encoder/decoders for LDPC codes because of the flexible number of processing units that can work in parallel rather than the limited number of processing units in Digital Signal Processors (DSPs) or general purpose processors (GPPs).

Most of the papers that discuss the architectures for LDPC decoders are based on a fixed code [5, 8] or scalable and support multiple rates but for a single code size. For example, a scalable structured LDPC decoder with relatively high data throughput is proposed in [9] for very long codeword lengths defined by the DVB-S2 standard. Also, authors in [6] and [12] offer multi-rate decoder designs based on structured parity check matrices (PCMs) for regular and irregular codes, using the standard belief propagation algorithm.

Every application has its own requirements and specific parameters. In order to decrease the design time and reuse the same hardware for many applications, a general, configurable, flexible architecture that supports several cases is of interest. This LDPC decoder should support a family of codes and should be able to switch between different code types seamlessly. For example, in wireless communication systems, when the transmission channel is good - meaning less errors during transmission- codes with fewer number of redundant bits or higher rates can be used. Or, shorter block lengths can be used when there is less data to be transferred. So, a multi-size, multi-rate LDPC decoder is a suitable choice for future generations of wireless devices.

In this paper, we present a novel high throughput LDPC decoder that supports multiple block lengths (648, 1296, 1944 bits) and multiple code rates ($1/2$, $2/3$, $3/4$, $5/6$) based on the IEEE $802.11n$ standard [1]. These codes are block-structured and their profile is near optimal leading to excellent performance of the decoder. The LDPC de-

coder has a semi-parallel design and supports twelve irregular codes with a small control and arithmetic logic overhead. It can also switch between different code sizes and code rates on the fly without any need for recompilation or re-synthesis. Furthermore, it uses layered belief propagation which converges twice as fast as the standard belief propagation algorithm, resulting in twice the throughput. This decoder architecture is easily expandable to support other code sizes/rates. A prototype of the decoder architecture is implemented on an FPGA and also synthesized for ASIC using Chartered Semiconductor $0.13\mu m$ CMOS technology. The details of the architectures are presented in sections 3 and 4.

2. Low Density Parity Check Codes

An LDPC code is a linear block code specified by a very sparse parity check matrix (PCM) $H_{(N-K)\times N}$, where nonzero entries are typically placed at random. For K information bits and a codeword length of N bits, the code rate is $R = K/N$. LDPC codes are usually represented by a bi-partite graph in which a variable node corresponds to a 'coded bit' or a PCM column, and a check node corresponds to a parity check equation or a PCM row. There is an edge between each pair of nodes if there is a 'one' in the corresponding PCM entry. During the decoding, messages are passed among the graph nodes. Log-likelihood ratios (LLRs) are used for representation of reliability messages.

The number of nonzero elements in each row or column of a PCM is called the 'degree' of that node. An LDPC code is regular or irregular based on the node degrees. If variable or check nodes have different degrees, then the LDPC code is irregular. Irregular codes usually have better performance than regular codes because the nodes with higher degrees converge faster and assist the nodes with lower degrees.

2.1. Layered belief propagation algorithm

The LDPC decoder architecture proposed in this paper utilizes the iterative layered belief propagation (LBP) algorithm as defined in [10]. This algorithm is a variation of the standard belief propagation [4], and achieves about two times faster decoding convergence due to the optimized scheduling of reliability messages [11]. The PCM can be viewed as a group of concatenated horizontal layers as shown in Fig. 1, where every layer represents the component code. The belief propagation algorithm is repeated for each horizontal layer and updated APP messages are passed between layers. Let R_{mj} denote the check node LLR message sent from the check node m to the variable node j. Let $L(q_{mj})$ denote the variable node LLR message sent from the variable node j to the check node m. The $L(q_j)$ $(j = 1, \ldots, N)$ represent the *a posteriori* probability ratio (APP messages) for all variable nodes (coded bits). The APP messages are initialized with the corresponding *a priori* (channel) reliability value of the coded bit j. For each variable node j inside the current horizontal layer, messages $L(q_{mj})$ that correspond to a particular check equation m are computed according to:

$$L(q_{mj}) = L(q_j) - R_{mj}. \quad (1)$$

For each check node m, messages R_{mj}, corresponding to all variable nodes j that participate in a particular parity-check equation, are computed according to:

$$R_{mj} = \prod_{j\prime \in N(m)\setminus\{j\}} \text{sign}\left(L(q_{mj\prime})\right) \Psi\left[\sum_{j\prime \in N(m)\setminus\{j\}} \Psi\left(L(q_{mj\prime})\right)\right], \quad (2)$$

where $N(m)$ is the set of all variable nodes from parity-check equation m, and $\Psi(x) = -\log\left[\tanh\left(\frac{|x|}{2}\right)\right]$. For the purpose of more efficient architecture implementation, updating of the check node messages in (2) is replaced with the modified min-sum approximation [3]. According to this solution, the updating of check node messages in the mth row of the kth decoding iteration is determined as:

$$R_{mj} \approx \prod_{j\prime \in N(m)\setminus\{j\}} \text{sign}\left(L(q_{mj\prime})\right) \times \max\left(\min_{j\prime \in N(m)\setminus\{j\}} |L(q_{mj\prime})| - \beta, 0\right), \quad (3)$$

where β is a correcting offset equal to a positive constant. The *a posteriori* reliability messages in the current horizontal layer are updated according to:

$$L(q_j) = L(q_{mj}) + R_{mj}. \quad (4)$$

Hard decisions can be made after every horizontal layer based on the sign of $L(q_j)$, $j = 1, \ldots, n$. If all parity-check equations are satisfied or the pre-determined maximum number of iterations is reached, then the decoding algorithm stops. Otherwise, the algorithm repeats from Eq. (1) for the next horizontal layer. Fig. 2 shows pseudo-code for decoding LDPC codes using the layered belief propagation algorithm.

2.2. Block-structured LDPC codes.

Recent hardware designs of semi-parallel LDPC decoders are mostly based on block-structured PCMs, for both regular [8] and irregular [7, 10] codes. The block structure has several advantages over randomly generated codes. First, it simplifies the interconnection network between the variable and check nodes and allows using permuters for address interleaving. Second, instead of having random addressing, addresses can be generated using linear equations. Third, it allows packed storage of several messages

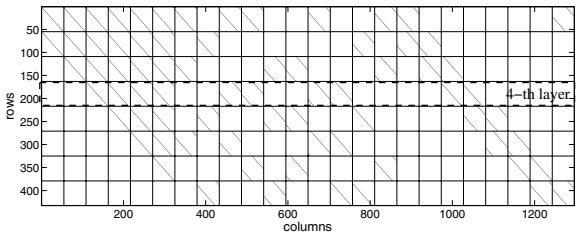

Figure 1. An overall view of a block-structured irregular parity-check matrix, $N = 1296$ bits and $R = 2/3$.

```
While iter<MaxIterations
  for layer=1:L
    for j=1:n, m=1:S
      - Calculate L(qmj), Rmj, L(qj).
    end.
    - Check if decoding conditions for the
      current layer are satisfied.
  end.
    If conditions for all the layers are
    satisfied then stop decoding otherwise
    continue to the next iteration.
end.
```

Figure 2. Pseudo-code for layered belief propagation algorithm.

in a single memory *word*. It should be noted that designing an architecture for the irregular code is more challenging because it has to be flexible to support different column degrees.

Our LDPC decoder is designed to support irregular block-structured PCMs proposed for the IEEE 802.11n wireless standard. Based on this standard, sub-block sizes are multiples of 27 and the code profiles are optimized to minimize the number of short cycles and therefore achieve excellent performance. As an example, a PCM of a code with a block length of 1296 bits and code rate of $2/3$ is shown in Fig. 1. In this figure the diagonal lines show the position of 'ones' in the matrix. The PCM is partitioned into square sub-matrices of size 54×54 with at most one nonzero entry per row/column. Each sub-matrix is either a shifted identity matrix or a zero matrix.

2.3. Three-stage pipelining principle

The decoding of each horizontal layer of the PCM is performed in three stages: memory reading stage, processing stage, and memory writing stage according to Eqs. (1), (2), and (4), respectively. In order to increase the decoding throughput, simultaneous execution of these three stages that belong to three consecutive horizontal layers is proposed. Three-stage pipelining is visualized in Fig. 3.

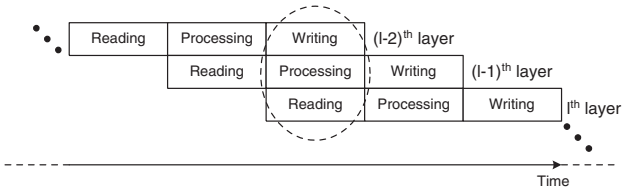

Figure 3. Three stage pipelining for three consecutive horizontal layers of PCM.

During the *reading* stage, APP messages that correspond to the PCM columns with non-zero entries at the lth horizontal layer are read from the appropriate memory, as well as the check node messages from the same layer updated in the previous decoding iteration. Variable node messages from the lth layer are updated according to (1). At the same time, the processing stage of the (l-1)th layer is executed: all check node messages that belong to this layer are computed simultaneously according to (2) based on updated variable node messages from the (l-1)th horizontal layer.

Pipelining of the writing stage (for the (l-2)th horizontal layer) requires some modifications. APP messages that correspond to all non-zero entries inside the (l-2)th horizontal layer are updated and written back in the appropriate memory. It is crucial to observe that Eq. (4) (for (l-2)th layer) cannot be directly applied since the most recently updated APP messages will not be utilized. After combining (1) and (4) for the (l-2)th layer, the APP reliability messages are updated according to:

$$L(q_j^{(k,l-2)}) = L(q_j^{(k,l-2-p)}) - R_{mj}^{(k-1,l-2)} + R_{mj}^{(k,l-2)}, \quad (5)$$

where $L(q_j^{(k,l-2-p)})$ is the latest updated value of the corresponding APP message passed to the (l-2)th layer at the kth decoding iteration. This value is updated p horizontal layers before the (l-2)th layer, not necessarily during the same kth decoding iteration. During the writing stage all newly updated check node messages from the $(l-2)$th horizontal layer are also written back in the appropriate check memory.

Three stage pipelining represents the maximum achievable level of pipelining since both read and write ports of the memory modules have 100% utilization during the decoding process.

3. Scalable decoder architecture

In this section, we present the description of our configurable, flexible LDPC decoder implementation. Based on the standard and the simulations to be presented in section 5, we have chosen these parameters for the decoder:

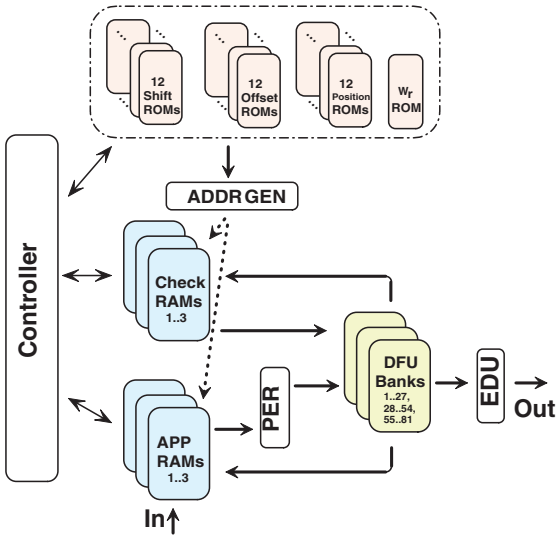

Figure 4. LDPC decoder block diagram.

- The decoding algorithm is layered belief propagation since it converges twice as fast as standard belief propagation which means that half the decoding iterations are needed to achieve the same performance.
- The PCM has block structure with blocks composed of a shifted identity matrix or a zero matrix.
- Supports irregular LDPC codes to preserve excellent performance of irregular random PCMs.
- Accepts block lengths of $N = 648, 1296, 1944$ bits as input, which corresponds to sub-block sizes of $S = 27, 54, 81$, respectively.
- Supports code rates: $R = 1/2, 2/3, 3/4, 5/6$.
- Inputs are the reliability of coded bits which are represented by $b = 7$ or 8-bit signed numbers.
- Flexible permuters are used for routing the messages to the proper nodes.

Fig. 4 shows a block diagram of the LDPC decoder. The decoding is based on Eqs. (1), (2), and (4) and the pseudo-code in Fig. 2. The design consists of memory blocks (RAMs and ROMs) for storing messages, processing blocks, permuters to route the data and a control unit. Decoding starts by reading a parameter that selects the code rate and code size of the input codeword block. According to this value, the appropriate values of the PCM are read from the Shift, Offset, Position and W_r [1] ROMs. These values are used to generate the addresses to read/write from/in the Check and APP memories. The decoder also inputs the reliability values and stores them in the APP memories. The APP values pass through the permuters to be routed to the correct decoding functional unit (DFU) for processing. After the processing stage, results enter the 'early detection unit' (EDU) to check if the valid codeword is found. Also, the results are written back to the APP and Check memories. The next layer/ iteration starts when the decoder reads a new set of values from the APP memories and starts processing them. The early detection unit also analyzes the decoded word in parallel with the main processing path. The decoder stops and outputs the resulting codeword when either of the following two conditions is satisfied. Either EDU detects a valid codeword or the maximum number of iterations is reached. The control unit controls the flow of data between all the units. We will discuss each block of the architecture in more detail in the following sections.

[1] w_r is number of nonzero values in each row of the parity check matrix.

3.1. Decoding functional unit banks

These blocks are the main processing part of the architecture since they update the Check and APP messages and calculate the new values. The number of parallel processing elements in these banks shows the parallelism factor of the design. For example, for the 1296 block code in Fig. 1, 54 processing nodes can work in parallel without any data dependency restrictions. To support different code lengths, the architecture should support a set of sub-block sizes of $S = 27, 54, 81$. Hence, we divide the decoding functional units into three banks, each of which contains 27 DFUs. For the size 1944 block, all three banks are used (81 DFUs), for the 1296 block, two of the banks are used (54 DFUs), and for the 648 block just one of the banks is in use. The architecture of the DFU is discussed in more detail in the next section.

3.1.1 Decoding functional unit

The decoding function unit (DFU) inputs w_r values from the check memories and w_r values from the APP memories, and updates new APP messages and check messages based on equations (1), (2) and (4).

Fig. 5 shows a block diagram of the DFU. The values of the APP and the check messages enter this unit and the difference is calculated based on Eq. (1). The sign and magnitude of inputs are separated and pass through separate computation paths based on Eq. (3). The serial Min-Sum unit inputs the absolute value of the APP messages and finds the two minimums of the w_r values and the relative index of these values compared to the first input. Then, the offset value is subtracted from the minimums. The intermediate output corresponding to each input is the minimum of all the other values. The sign of this output is the multiplication of the signs of all the inputs other than the input corresponding to this output. Then, the previous check node values are subtracted from these and the APP node values are added to the results to generate the final outputs (w_r of

them) based on Eq. (4). The outputs of all the S-DFUs are concatenated and stored in one address of the APP memories. It is important to note that we do not permute back these values; APPs are stored in the shifted order. The next layer uses values from the Offset ROM instead of the shift values in the permuter. This way we eliminate the need for another permuter which saves 8% of the total area.

3.1.2 Serial Min-Sum unit

This unit inputs w_r values and finds the two smallest values. Depending on the requirements on the architecture, this unit can be designed in two different ways: 'Parallel-Input' (PI) or 'Serial-Input' (SI). The PI generates results faster by using processing units in parallel. This approach is suitable for a decoder that supports regular LDPC codes [8] with a fixed number of inputs. Since we are dealing with irregular codes, the number of non-zero sub-matrices in each row of the PCM is different for each layer. This means that the number of inputs for the Min-Sum unit varies based on the w_r. In the irregular PCMs for our architecture, these values are between 7 and 21 for different cases. In order to support a variable number of inputs, we have designed a 'serial' Min-Sum unit. Although this unit runs more slowly than the parallel version, it has the flexibility of accepting any number of input values in serial using just one input port. In this way, the hardware can support irregular codes of different rates and sizes. The controller marks the beginning and end of the input sequence. The Min-Sum unit finds the two smallest numbers and their location in the sequence compared to the first input. Fig. 6 shows the block diagram of this unit.

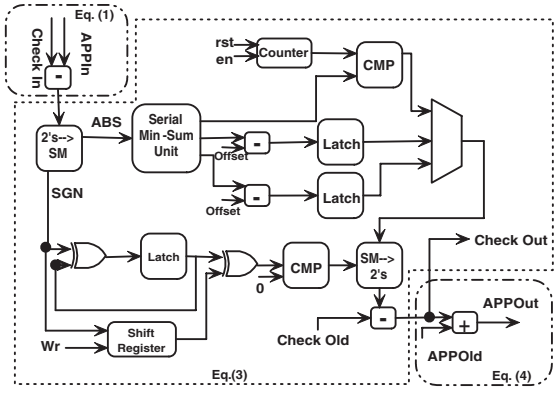

Figure 5. The decoding functional unit block diagram.

3.1.3 The flexible permuters

One of the main challenges of the LDPC decoder is 'routing' the messages from the memories to the correct processing units as quickly as possible. For a fixed decoder that supports a single block length/code rate, this can be done with a network of multiplexers to route the signals according to the shift values. We call these multiplexer networks a 'permuter'.

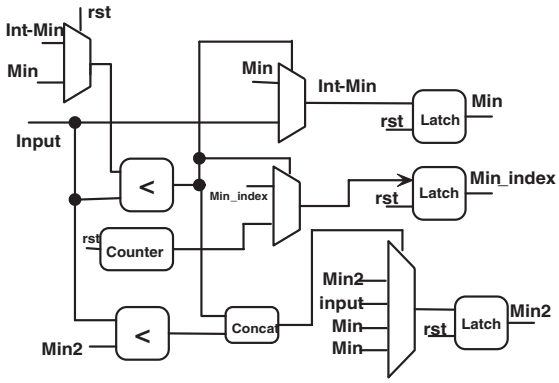

Figure 6. The serial min-sum unit block diagram.

Permuters are used to shift a block of S, b-bit numbers to generate the correct addressing based on the PCM. For example, a shift of s means that the order of the outputs should be $(s+1)\ldots S,1\ldots s$ instead of $1\ldots S$. Analysis was done on the structure of permuters to determine the best structure. We tested permuters of size 81 using different sizes/combinations of the multiplexers, and selected the one with the smallest area and highest speed, which is designed with four levels of $4:1, 4:1, 4:1, 2:1$, multiplexers with b-bit inputs/outputs.

Since permuters occupy a significant portion of the area, instead of having three permuters to support different values of S (corresponding to different code rate/size), we designed a flexible permuter of size 81, which can be used to permute any of the sub-block sizes. This flexible permuter is made by adding a layer of multiplexers to the original permuter of size 81 and selecting the proper signal for each case as shown in Fig. 7.

3.2. Memory organization

There are several ROM/ RAM blocks in this system. These blocks are divided into different banks for Check and APP messages and a few ROMs that store the parameters to regenerate the various parity check matrices.

3.2.1 Check memory blocks

Three memory blocks are used to store the check messages. In order to increase the throughput of the decoder and take advantage of the parallel processing, we use packed storage

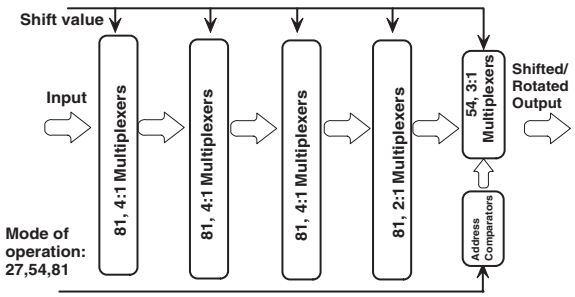

Figure 7. The flexible permuter block diagram.

of the check and APP values. Through the use of packing, S values are concatenated and stored in a single memory address that can be read in a single clock cycle. The permuters are used to split and route each single value to the corresponding DFU unit. Also, we have divided these memory blocks into three modules that accept the same address (to avoid extremely large memory word lengths). Each of these modules packs and stores 27, b-bit values per address. These values correspond to the check outputs for the DFUs numbered from 1 to 27, 28 to 54 and 55 to 81. In this way, by reading from a single address from the three memory blocks, all the 81 check values are ready for concurrent processing. These memories are dual-port to simultaneously read the next message and store the current message during pipelining.

3.2.2 APP memory blocks

Using the same packed storage concept, three dual port memories are used to store the APP messages. The APP values are also grouped and stored in a single memory address (groups of 27, b-bit numbers). During the initialization step, these messages are stored in the original order of the inputs. They are permuted based on the elements of the PCM to route to the correct DFU for processing. Depending on the block sizes, the APP memories can be divided into three or more modules.

3.2.3 Packed storage of multiple PCMs

Our flexible architecture supports $n = 3$ block lengths and $r = 4$ code rates which gives us $\beta = n \times r = 12$ combinations. These β parity check matrices that correspond to each code size/rate should be stored in ROM memories which can be very large if stored directly. The intuition behind reducing the memory required for storing the PCMs is to take advantage of the structure of the PCMs and store a few parameters and regenerate the matrix when needed. Considering the structure of the PCM, we can rebuild it by knowing a few parameters: block length, sub-block size, number of nonzero sub-matrices in each layer of the PCM, position of the nonzero blocks and the shift value to generate the shifted identity matrices. The PCM can be regenerated on the fly using counters and shifter/permuters.

We have added another value to these parameters namely 'offset' values. These values are the shift values with regard to the previous shift of the same row (see Fig. 1). For example, if the shift value for row k is s_k and the shift value for the next row is s_{k+1}, the offset o_{k+1} will be equal to: $o_{k+1} = s_{k+1} - s_k$. The decoder uses the value of o_{k+1} in the permuter to route the messages.

3.3. Early detection unit

In order to detect that the correct codeword is found, two sets of tests are performed after finishing the decoding of each layer. First, check if all the parity check equations are satisfied ($H_l \times c = 0$). Then, compare the signs of the updated APP messages with the previous values of these messages and check if the signs have changed: $(sign(Lq_j^{k,l}).sign(Lq_j^{(k,l-p)})) > 0$.

The outputs of a layer are valid if all the parity check equations are satisfied and there is no sign change in the results. It is required that L (total number of layers) consecutive layers satisfy these two constraints even if they belong to two consecutive iterations. Then, decoding stops and outputs the resulting codeword.

3.4. Multi-rate controller design

This block controls the flow of the messages into/out of different blocks during the decoding. The controller generates enable/reset and all the hand-shaking signals necessary for correct operation of the decoder based on Fig. 2. It also controls the counters that generate addresses for the ROM and RAM memories. The controller inputs the values of block size and code rate and also reads the number of nonzero blocks in each layer (w_r) from the w_r ROM. Based on these parameters, this unit controls the flow of data to the DFUs, Min-sum units, permuters and other blocks in the system. The main challenge of the controller is to keep track of the groups of w_r messages throughout the process and read/ process/ write them at the correct time.

3.5. Hardware overhead

In order to support the highest data rates for wireless networks, this decoder is designed to support the largest block length ($N = 1944$). For the smaller block lengths some of the DFU units and memories are unused and can be turned off/disabled using clock gating. Because of the data dependency, it is not possible to use a number of DFUs greater than the sub-block size. This is a limitation that is imposed by the PCM structure and the decoding algorithm. Another

Table 1. Gate count of the permuters.

Permuters	Gates ($b = 7$)	Gates ($b = 8$)	Normalized Overhead
$S = 27$	3,972	4,539	-
$S = 54$	10,209	11,667	-
$S = 81$	17,014	19,444	1
$3Per$	32,898	37,596	1.93
$FlexPer$	20,471	23,144	1.19

overhead in the design comes from storing the twelve PCMs in the memories corresponding to each combination of the code rate/size, which is almost 22 Kbits.

The flexible permuter also has some overhead. Table 1 shows the number of gates used for different permuter sizes. These numbers are calculated based on the number of gates that each mutiplexer/adder/subtractor requires. The flexible permuter ($FlexPer$) uses 39% less CMOS gates compared to the case of having three permuters of size 27, 54 and 81 ($3Per$). The $FlexPer$ has 19% overhead compared to the permuter with 81, b-bit inputs/outputs.

4. Hardware implementation of LDPC decoder

Four prototype architectures for the LDPC decoder have been implemented in Xilinx System Generator and targeted to a Xilinx Virtex4-xc4vfx60 FPGA. These cases include:

- LDPC decoder with pipeline, 7-bit messages (DecWP7)
- LDPC decoder with pipeline, 8-bit messages (DecWP8)
- LDPC decoder without pipeline, 7-bit messages (DecNP7)
- LDPC decoder without pipeline, 8-bit messages (DecNP8)

Table 2 shows the utilization statistics of all these architectures. A clock frequency of 160 MHz is achieved for all of these designs after place and route.

Table 2. Design statistics for the flexible LDPC decoder on Virtex4-xc4vfx60 FPGA.

Resource	*DecNP7*	*DecNP8*	*DecWP7*	*DecWP8*
Slices	11,328	12,633	13,665	15,235
FFs	12,368	13,823	13,617	15,239
LUTs	17,104	19,265	21,667	24,442
Block RAMs	87	87	123	123

The proposed decoder architecture is also synthesized for a Chartered Semiconductor 0.13μm, 1.2 V, CMOS technology using the BEE/Insecta design flow [2] and Synopsys tools. The Chartered memory compiler was used to generate efficient RAM and ROM blocks. Table 3 shows the area occupied by each part of the decoder with 8-bit messages and with pipelining (DecWP8) in square millimeters. This architecture runs at a clock speed of 412 MHz (worst case) and consumes 502 mW of dynamic power (estimated using Design Compiler). The total area for the decoder is $2.2036mm^2$ for the DecNP7, $3.3332mm^2$ for the DecWP7 and $2.4128mm^2$ for the DecNP8.

Table 3. ASIC design statistics for the DecWP8 flexible LDPC decoder.

Resource	*Area* mm^2	*Percentage*
Memory Banks	2.385	64.7%
DFU Banks	0.939	25.5%
Permuters	0.289	7.8%
Control	0.0029	0.1%
LDPC Decoder	3.6854	100%

5. Decoding throughput and performance

Fig. 8 shows frame error rate (FER) performance of the implemented decoder for a $2/3$-rate code with a code length of 1296 bits and layered belief propagation decoding algorithm for 6, 7 and 8-bit fixed point arithmetic precisions and floating point with and without pipelining. Both 7 and 8-bit fixed-point precisions have small loss compared to the floating point version. It should be noted that the pipelined version of LBP decoder achieves much higher throughput while introducing only a small loss compared to the decoder without pipelining.

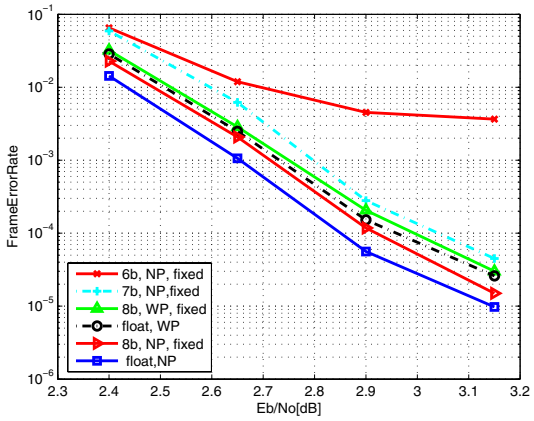

Figure 8. FER for irregular block-structured PCM (R=2/3, N=1296, MaxIter=15, Layered belief propagation).

We have estimated the decoding throughput (considering

the information bits) based on the average number of decoding iterations required to achieve a frame error rate of 10^{-4}, while the maximum number of iterations is set to 15. The clock frequency is 160 MHz for the decoder running on the FPGA and 412 MHz for the ASIC implementation. Fig. 9 shows the average throughput as a function of code rate and codeword size for the DecNP8 design. As an example, for a block length of 1944, rate $5/6$ code, the decoder implemented on the FPGA achieves an average throughput of 292 Mbits/second and the ASIC version achieves 736 Mbits/sec. Average FPGA latency of the decoder is between 5 and $11\mu sec$ for different block lengths/rates while the average ASIC latency is between 2.2 and $4.5\mu sec$. It is important to note that these numbers are for the architecture without pipelining. Pipelining can increase the average throughput approximately 3x. In that case the average throughput will be close to 900 Mbits/sec for FPGA and close to 2 Gbits/sec for ASIC. All the decoders achieve hundreds of Mbit/s throughput. The ASIC versions of the decoder with or without pipelining and the FPGA version with pipelining achieve the throughput required in the standard.

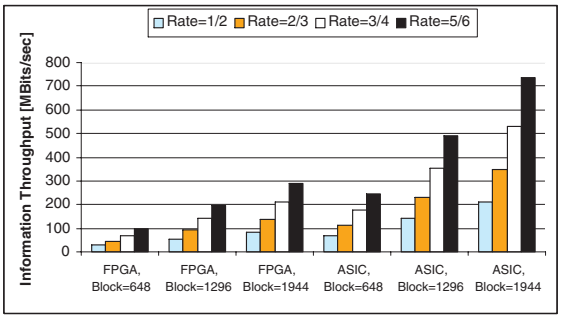

Figure 9. Average decoding throughput for different code rates and codeword lengths for the FPGA and ASIC architectures of DecNP8 based on average number of iterations.

6. Conclusions

We presented the design and implementation of a flexible, high throughput decoder architecture for structured irregular LDPC codes. The decoder supports a family of code sizes and rates and can switch between different cases on the fly. The decoder has a general structure and can be extended to support other code families. The flexible decoder is implemented on the FPGA and synthesized for ASIC. It achieves a throughput of several hundreds of Mbits/second.

7. Acknowledgements

This work was supported in part by Nokia Corporation and by NSF under grants EIA-0321266, CNS-0551692 and CNS-0619767. We would like to thank Yang Sun for his help in ASIC synthesis.

References

[1] *IEEE 802.11 Wireless LANsWWiSE Proposal: High Throughput extension to the 802.11 Standard*. IEEE 11-04-0886-00-000n.

[2] C. Chang, K. Kuusilinna, B. Richards, A. Chen, N. Chan, R. W. Brodersen, and B. Nikoliae. Rapid design and analysis of communication systems using the BEE hardware emulation environment. In *14th IEEE International Workshop on Rapid Systems Prototyping*, June 2003.

[3] J. Chen, A. Dholakai, E. Eleftheriou, M. Fossorier, and X. Hu. Reduced-complexity decoding of LDPC codes. *IEEE Transactions on Communications*, 53:1288 – 1299, Aug 2005.

[4] S. Y. Chung, T. Richardson, and R. Urbanke. Analysis of sum-product decoding of low-density parity-check codes using a Gaussian approximation. *IEEE Transactions on Information Theory*, 47:657–670, Feb. 2001.

[5] A. Darabiha, A.C. Carusone, and F.R. Kschischang. Multi-Gbit/sec low density parity check decoders with reduced interconnect complexity. In *IEEE International Symposium on Circuits and Systems,ISCAS 2005*, May 2005.

[6] L. Fanucci, M. Rovini, N.E. L'Insalata, and F. Rossi. High-throughput multi-rate decoding of structured low-density parity-check codes. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, pages 3539–3547, Dec. 2005.

[7] D. E. Hocevar. A reduced complexity decoder architecture via layered decoding of LDPC codes. In *IEEE Workshop on Signal Processing Systems,SIPS*, pages 107–112, 2004.

[8] M. Karkooti and J. R. Cavallaro. Semi-parallel reconfigurable architectures for real-time LDPC decoding. In *IEEE International Conference on Information Technology: Coding and Computing,ITCC 2004*, April 2004.

[9] F. Kienle, T. Brack, and N. Wehn. A synthesizable IP core for DVB-S2 LDPC code decoding. In *Proceedings of Design, Automation and Test in Europe*, 2005.

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

[11] P. Radosavljevic, A. de Baynast, and J.R. Cavallaro. Optimized Message Passing Schedules for LDPC Decoding. In *IEEE 39th Asilomar Conference on Signals, Systems and Computers*, pages 591–595, Nov. 2005.

[12] L. Yang, H. Lui, and C.-J.R. Shi. Code construction and FPGA implementation of a low-error-floor multi-rate low-density parity-check code decoder. *IEEE Transactions on Circuits and Systems I, Regular Papers, Accepted for future publication.*