

# A Flexible Hardware Encoder for Low-Density Parity-Check Codes

Dong-U Lee and Wayne Luk  
Department of Computing  
Imperial College  
London  
United Kingdom  
{dong.lee, w.luk}@ic.ac.uk

Connie Wang, Christopher Jones,  
Michael Smith, and John Villasenor  
Electrical Engineering Department  
University of California  
Los Angeles  
USA  
{jad, christop, miksmith, villa}@icsl.ucla.edu

## Abstract

*We describe a flexible hardware encoder for regular and irregular low-density parity-check (LDPC) codes. Although LDPC codes achieve better performance and lower decoding complexity than Turbo codes, a major drawback of LDPC codes is their apparently high encoding complexity. Using an efficient encoding method proposed by Richardson and Urbanke, we present a hardware LDPC encoder with linear encoding complexity. The encoder is flexible, supporting arbitrary  $H$  matrices, rates and block lengths. An implementation for a rate 1/2 irregular length 2000 LDPC code encoder on a Xilinx Virtex-II XC2V4000-6 FPGA takes up 4% of the device. It runs at 143MHz and has a throughput of 45 million codeword bits per second (or 22 million information bits per second) with a latency of 0.18ms. The performance can be improved by exploiting parallelism: several instances of the encoder can be mapped onto the same chip to encode multiple message blocks concurrently. An implementation of 16 instances of the encoder on the same device at 82MHz is capable of 410 million codeword bits per second, 80 times faster than an Intel Pentium-IV 2.4GHz PC.*

## 1 Introduction

Forward error correction (FEC) [9] is a critical part of modern communications systems, where it is used to detect and correct errors introduced during a transmission over a channel. It relies on transmitting the data in an encoded form, such that the redundancy introduced by the coding allows a decoding device at the receiver to detect and correct errors. Using FEC, no request for retransmission is required, unlike systems which only detect errors usually by means of a checksum transmitted with the data. In many ap-

plications, a substantial portion of the baseband signal processing is dedicated to FEC.

In the past few years, LDPC codes [2, 3] have received much attention because of their excellent performance and the large degree of parallelism that can be exploited in the decoder. LDPC codes are being widely considered as the most promising candidate FEC scheme for many applications in telecommunications and storage devices. Recently, LDPC codes have been selected over Turbo codes [1] by Europe's DVB standards group for next-generation digital satellite broadcasting due to their superior performance. Provided that the information block lengths are long enough, performance close to the Shannon limit can be achieved with LDPC codes. Luby et al. [7] formally showed that properly constructed irregular codes can approach the channel capacity more closely than regular codes. LDPC codes exhibit an asymptotically better performance than Turbo codes and admit a wide range of tradeoffs between performance and complexity.

Although LDPC codes achieve better performance and have low decoding complexity compared to Turbo codes, one of the major drawbacks of LDPC codes lies in their apparently high encoding complexity. Whereas Turbo codes can be encoded in linear time, a straightforward implementation for an LDPC code has complexity quadratic in the block length. Note that the complexity referred to here is measured in the number of mathematical operations required per bit. In [10], Richardson and Urbanke (RU) show that linear time encoding is achievable through careful linear manipulation of 'good' LDPC codes. In their paper, they present methods to preprocess the parity check matrix  $H$  and a set of matrix operations to perform the actual encoding. We have implemented the preprocessing in software since it needs to be performed only once for a given  $H$  matrix. For the actual hardware encoder, we have identified the operations that can be run in parallel and scheduled the tasks to maximize throughput. In addition we have de-

signed an efficient memory architecture for storing sparse matrices.

The principal contribution of this paper is a fast and efficient hardware encoder for both irregular and regular LDPC codes based on the RU method. The novelties of our work include:

- software preprocessor to bring the parity check matrix  $H$  into an approximate lower triangular form;
- hardware architecture with an efficient memory organization for storing and performing computations on sparse matrices;
- implementation and evaluation of the encoder that is capable of 80 times speedup over a 2.4GHz PC; we also explore run-time reconfiguration opportunities.

The rest of this paper is organized as follows: Section 2 covers background material and previous work. Section 3 presents an overview of our approach. Section 4 describes how we preprocess the  $H$  matrix. Section 5 presents our hardware encoder architecture. Section 6 describes the main components used in our encoder. Section 7 discusses our implementation results. Section 8 offers conclusions and future work.

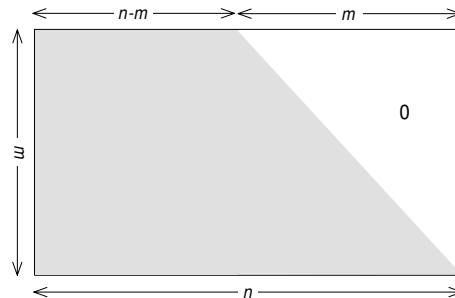
## 2 Background

A typical digital communication system is shown in Figure 1. The information bits are fed to the source encoder, which compresses the data. The channel encoder (LDPC) is where the work presented in this paper applies. It creates parity bits for a block of message generating codewords. These parity bits are used to detect and correct errors at the receiver. The codewords are sent through a digital channel, which is often modelled with AWGN. At the receiver end, soft decisions are fed to the channel decoder (LDPC), which iteratively decodes the received codeword and provides the decoded bits to the source decoder. The source decoder decompresses the decoded bits and outputs the recovered information bits. Details of our hardware LDPC decoder and Gaussian noise generator (which is needed to evaluate the performance of a particular code) can be found in [5] and [6].

LDPC codes are linear block codes. Encoding of such codes uses the following property:

$$Hx^T = 0. \quad (1)$$

where  $x$  represents the codeword and  $H$  represents the parity check matrix. A straightforward encoding scheme requires three steps: a) Gaussian elimination to transform the  $H$  matrix into a lower triangular form (Figure 2), b) split  $x$  into information bits and parity bits, i.e.,  $x = (s, p_1, p_2)$



**Figure 2. An equivalent parity-check matrix in lower triangular form. Note that  $n =$  block length and  $m =$  block length  $\times$  (1 - code rate).**

where  $s$  is vector of information bits,  $p_1, p_2$  are vectors of parity bits, c) solve the equation  $Hx^T = 0$  using forward-substitution. It takes about  $O(N^3)$  to perform Gaussian elimination. Since afterwards the  $H$  matrix will no longer be sparse, it takes  $O(N^2)$ , or more precisely,  $n^2 \left(\frac{r(1-r)}{2}\right)$  XOR operations for the actual encoding, where  $r$  is the code rate. The code rate is the ratio of information bits to codeword bits and has a value between 0 and 1. In order to reduce the quadratic complexity, Urbanke and Richardson took advantage of the sparsity of the  $H$  matrix. They found that in most cases, the encoding complexity is either linear or quadratic but quite manageable. For example, for a (3,6) regular code of length  $n$ , even though the complexity is still quadratic, the actual number of operations required is  $O(N)$  in addition to  $0.017^2 n^2$ . Since  $0.017^2$  is a very small number, the complexity of the encoder is still manageable for large  $n$ .

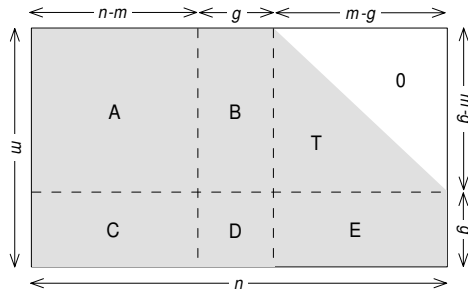
To the best of our knowledge, existing hardware LDPC encoders in the literature [4, 8, 13] employ the straightforward encoding method where a vector of information bits is multiplied by a dense generator matrix, which has complexity quadratic to the block length.

## 3 Overview

The RU algorithm consists of two steps: a preprocessing step and the actual encoding step. In the preprocessing step, row and column permutations are performed to bring the parity-check matrix  $H$  into an approximate lower triangular form (ALT) as shown in Figure 3. Since the transformation is accomplished by permutations only, the sparseness of the matrix is preserved. The actual encoding is carried out by matrix-multiplication, forward-substitution and vector addition operations. Since the preprocessing needs to be performed only once on a given  $H$  matrix, we exe-



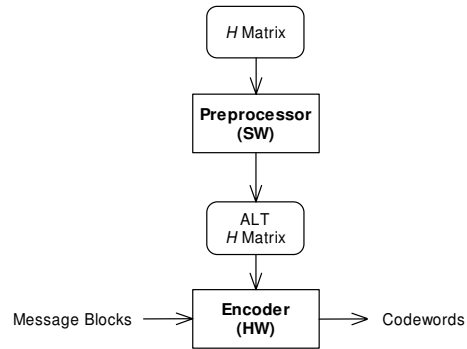
**Figure 1. A typical digital communication system. LDPC is considered for the channel encoder/decoder in this paper.**



**Figure 3. The parity-check matrix  $H$  in ALT form.  $A$ ,  $B$ ,  $C$ , and  $E$  are sparse matrices,  $D$  is a dense matrix, and  $T$  is a sparse lower triangular matrix.**

cutte this operation in software. The actual encoding step is done in hardware. The RU encoding algorithm is presented in [10] as a set of matrix operations. We have examined the algorithm and identified the operations that can be executed in parallel. The operations implemented in our hardware encoder are scheduled to maximize concurrency and throughput. Moreover, we are employing an efficient memory architecture for storing sparse matrices, which minimizes memory usage.

The basic framework of our encoder is shown in Figure 4. Our approach for LDPC encoding consists of two steps: preprocessing and hardware encoding. First, the original parity check matrix  $H$  is preprocessed with the RU algorithm to generate the appropriate look-up tables consisting of the six matrices needed by the hardware encoder. These matrices are generated from the RU algorithm and contain information on how the input message blocks are encoded to generate codewords. This preprocessing step is implemented in software and needs to be performed once for a given  $H$  matrix. The hardware encoder itself is implemented on an FPGA and uses the look-up tables (ALT  $H$  matrix) generated from the preprocessing step to encode the message blocks. Note that the preprocessing step does not involve any data. Hence, during a normal encoding operation only the hardware encoder is needed. Although our implementation is based on  $H$  matrices that are binary, GF(2),



**Figure 4. LDPC encoding framework.**

it can be extended to matrices that belong to higher order fields.

The RU algorithm and the hardware architecture proposed in this paper make no restrictions on the actual  $H$  matrix. This flexibility allows our hardware architecture to be used in any application involving LDPC codes. Different applications require different  $H$  matrices. Applications requiring low latency typically use shorter block lengths (less than 1000), while applications requiring operation very near the channel capacity require longer block lengths (more than 10000). Code rate  $r$  also influences the dimensions of the  $H$  matrix. Low code rates offer more error protection at the expense of information throughput and are often used when the signal-to-noise ratio (SNR) is very low (e.g. deep space communications). The dimensions of the  $H$  matrix are (block length  $\times$  (1 - code rate)) by (block length) as illustrated in Figure 2. Our hardware architecture is completely flexible in regards to block length and code rate.

Another issue related to encoder flexibility is the specific location of ones in the  $H$  matrix. Properly designed regular LDPC codes have performance that continues to improve as the SNR is increased. Irregular LDPC codes do not have this property, they have a so-called ‘error-floor’, meaning that after a certain level of performance is reached, the performance stops improving. For example, say a code operates at a bit-error rate (BER) of  $10^{-5}$  at 2dB SNR. If an error floor exists, the BER will be the same when the SNR is increased to 5dB. If an error floor was not present, then

the BER would improve to say  $10^{-6}$ . While regular LDPC codes have no error-floor, they do not perform as close capacity as irregular codes. This means that as the SNR is increased, the BER will decrease faster with irregular codes than with regular codes. An ideal code performs close to capacity and contains no error-floor. We have designed high-performance LDPC codes in [11] using special code construction techniques, which perform close to capacity and have reduced error-floors. Our hardware is completely flexible in regards to the location of ones in the  $H$  matrix, in other words, this hardware can encode any LDPC code that is created.

## 4 Preprocessing

In preprocessing, row and column permutations are performed to bring the  $H$  matrix into an ALT form. Richardson and Urbanke [10] introduced three algorithms to perform this task. There are three types of greedy algorithms:  $a$ ,  $b$  and  $c$ . We choose greedy algorithm  $a$  for our software pre-processor.

Preprocessing consists of two steps: triangulation and rank checking. Triangulation is the process of row and column permutations that produces an  $H$  matrix similar to the one shown in Figure 3, with the smallest gap  $g$  possible. Multiplying

$$\begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix} \quad (2)$$

from the left of  $Hx^T = 0$ , where  $H = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix}$  we get

$$\begin{bmatrix} A & B & T \\ -ET^{-1}A + C & -ET^{-1}B + D & 0 \end{bmatrix} \begin{bmatrix} s \\ p1 \\ p2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (3)$$

This gives two equations and two unknowns  $p_1, p_2$ . Define  $F = -ET^{-1}B + D$  and assume for the moment that  $F$  is nonsingular, solving for  $p_1, p_2$  yields

$$p_1^T = -F^{-1}(-ET^{-1}A + C)s^T. \quad (4)$$

and

$$p_2^T = -T^{-1}(As^T + Bp_1^T). \quad (5)$$

From Table 1 and Table 2 we can see that the complexity of the actual operations required to obtain  $p_1$  and  $p_2$  are mostly linear, except for the dense matrix multiplication  $-F^{-1}(-ET^{-1}A + C)s^T$ , which has complexity  $O(N^2)$ . Thus, we have achieved near linear encoding complexity.

Since the equation for  $p_1$  depends on the inverse of  $F$ , the method will only work when  $F$  is nonsingular (invertible). This requires additional rank checking before the actual encoding. To obtain  $F$ , we perform Gaussian elimination on the original  $H$  to bring it into the form

$$\begin{bmatrix} A & B & T \\ -ET^{-1}A + C & -ET^{-1}B + D & 0 \end{bmatrix} \quad (6)$$

If  $F$  is singular, we swap columns of  $F$  with columns to the left of  $F$  and keep doing this until  $F$  becomes nonsingular.

So far, we have shown the encoding complexity to be linear, except for the dense  $g \times g$  matrix multiplication, where  $g$  is the gap of the preprocessed  $H$  matrix (Figure 3). Thus for efficiency, we should make  $g$  as small as possible.

The greedy algorithm  $a$  is used to find the best possible lower triangularization of the parity matrix  $H$ . The algorithm begins by assigning  $Q = H^T$ . Then the following steps are applied:

1. Find a vector of indices to degree one rows in  $Q$  and call this vector  $\alpha$ . If  $\alpha$  is empty, remove the left most column of  $Q$  and repeat step 1 with the modified  $Q$  matrix. Let  $l$  equal to the length of the vector  $\alpha$ .
2. Modify  $Q$  so that the degree one rows indicated by the elements of  $\alpha$  are moved to the top of  $Q$  (row numbers 1 to  $l$ ).
3. Reorder the columns of the modified  $Q$  so that the rows that were moved to the top of the matrix form a diagonal. This step is known as diagonal extension.
4. Modify  $Q$  again by removing the first  $l$  rows and columns of modified  $Q$ .
5. Find a vector of indices to degree one rows in modified  $Q$  and call this vector  $\alpha$ . If  $\alpha$  is empty, the algorithm terminates. Otherwise, go to step 2.

To summarize the above algorithm, through row and column operations a small identity matrix is created at the top left corner of the main matrix, and then the rows and columns participating in the identity matrix are deleted. This is repeated until there are no more degree one rows in the remaining matrix. Thus, through row and column swapping, we have produced the  $H$  matrix in Figure 3.

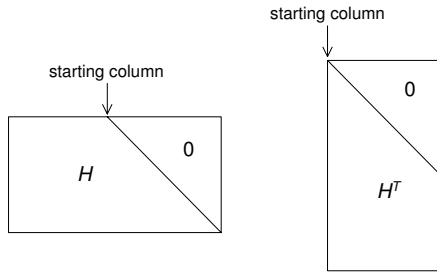
The gap size is the equal to the number of columns removed in step 1. Note that the gap size is further reduced by applying the greedy algorithm  $a$  to  $H^T$  rather than  $H$  itself. This is due to the different starting columns when applying step 1. Since  $H$  matrix is not a square matrix, we can only triangularize at most the number of columns as there are rows. As a result, if we were to try to carry out step 1 with a 'fat'  $H$  matrix, we can only look at part of the columns. For example, for a rate 1/2  $H$  matrix, in order to achieve

**Table 1. Computation of  $p_1^T = -F^{-1}(-ET^{-1}A + C)s^T$ . Note that  $T^{-1}[As^T] = y^T \Rightarrow Ty^T = [As^T]$ .**

index	operation	comment	complexity
1	$As^T$	multiplication by sparse matrix	$O(N)$
2	$T^{-1}[As^T]$	forward-substitution by sparse matrix	$O(N)$
3	$-E[T^{-1}As^T]$	multiplication by sparse matrix	$O(N)$
4	$Cs^T$	multiplication by sparse matrix	$O(N)$
5	$[-ET^{-1}As^T] + [Cs^T]$	vector addition	$O(N)$
6	$-F^{-1}[-ET^{-1}As^T + Cs^T]$	multiplication by dense $g \times g$ matrix	$O(N^2)$

**Table 2. Computation of  $p_2^T = -T^{-1}(As^T + Bp_1^T)$ .**

index	operation	comment	complexity
7	$As^T$	multiplication by sparse matrix	$O(N)$
8	$Bp_1^T$	multiplication by sparse matrix	$O(N)$
9	$[As^T] + [Bp_1^T]$	vector addition	$O(N)$
10	$-T^{-1}[As^T + Bp_1^T]$	forward-substitution by sparse matrix	$O(N)$



**Figure 5. Different starting columns for  $H$  and  $H^T$ .**

the result in Figure 3, we can only start with the middle column and work with the right half of the matrix to search for degree one rows. On the other hand, if we apply step 1 to a ‘skinny’  $H^T$  matrix, we can start with the very first column since the number of columns now is much less than the number of rows. This enables us to look at the entire matrix when searching for degree one rows. This extra degree of freedom results in better triangulation. The two different approaches are illustrated and compared in Figure 5.

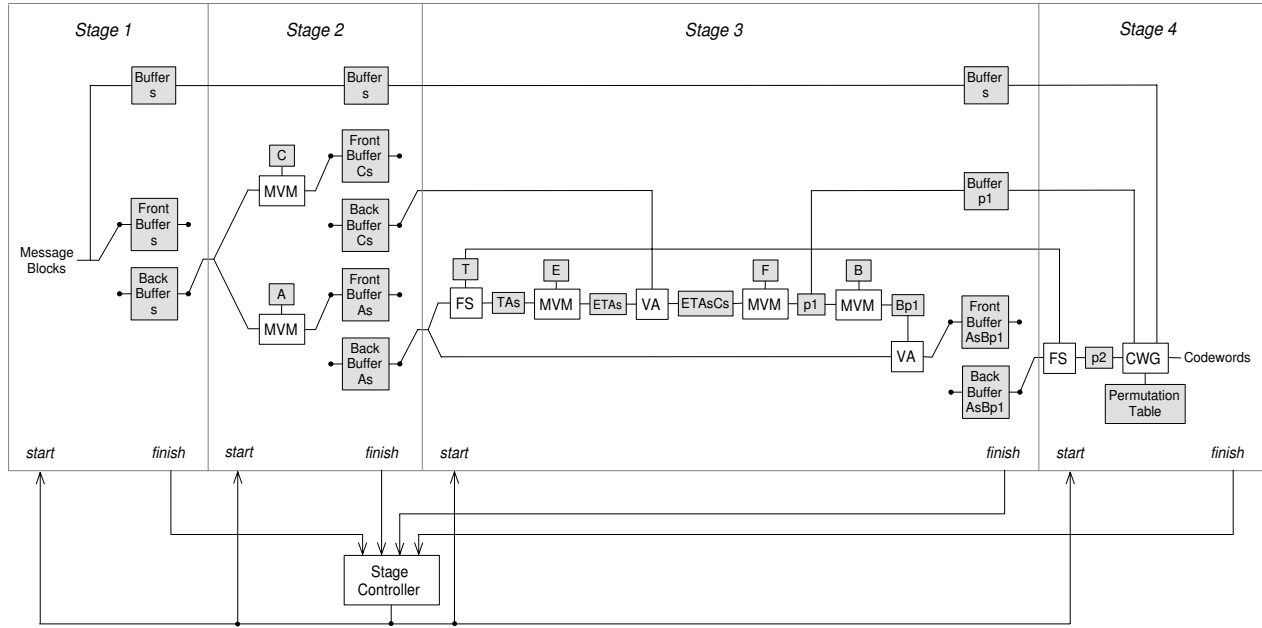
In addition to the greedy algorithm  $a$ , Urbanke and Richardson also introduced greedy algorithm  $b$  and greedy algorithm  $c$ . In algorithm  $b$ , rather than choosing the starting columns (starting point of triangulation) independent of one another, they are chosen based on the weights of the rows with which they are connected. This may reduce gaps but also requires more complicated processing. Greedy algorithm  $c$  is built upon algorithm  $b$  with a looser constraint on the weight distributions of rows and hence the definition of the starting columns with which they are connected.

Both  $b$  and  $c$  offer slightly smaller gaps in some cases; however, we have chosen  $a$  since it offers satisfactory results in all cases we have examined. Triangulation can be time-consuming with large block sizes. Since this step only needs to be performed once for a given  $H$  matrix, such overhead is tolerable.

## 5 Encoder Architecture

The hardware encoder computes the two parity parts  $p_1$  and  $p_2$  according to the operations described in Table 1 and Table 2. Operations that can be executed in parallel are identified and are scheduled to maximize parallelism. An overview of our hardware encoder architecture is shown in Figure 6. The operations are grouped into four stages that run in parallel and double buffering is used between the stages so that they can be executed in parallel. Each stage generates a ‘finish’ signal once its computation is completed. Once all stages are completed (which is when a codeword is generated), a ‘start’ signal is sent from the stage controller to each stage for the next execution. The stages have been carefully partitioned to balance the workloads between the stages, while minimizing the overall latency, idle times and buffering requirements. This flexible architecture supports any rate and block length, but has been specifically optimized for rate  $1/2$  codes.

The aim of dividing the encoding process into different stages is to balance the execution times among the stages, so that the idle time of any of the stages is minimized. Given that the rate is  $1/2$ , the gap is small and the edges (ones) of the  $H$  matrix are distributed in a random manner, the matrix  $A$  will contain nearly half of the edges of the entire preprocessed  $H$  matrix. Also, since the matrix  $T$  is lower-triangular, the number of its edges will be around half of



**Figure 6. Overview of our hardware encoder architecture. Double buffering is used between the stages for concurrent execution. Grey and white box indicate RAMs and operations respectively.**

A. Therefore, the computation  $As^T$  (operation 1) will take the longest. This is because the number of operations are proportional to the number of edges, which will be clarified later. Since the gap is small, operations involving  $B$ ,  $C$ ,  $E$  and  $F$  will be very fast.

In stage 1, we simply write the message block to buffers. Since the message block length is  $n - m$ , this stage will take  $n - m$  clock cycles. In stage 2, we perform operations 1 and 4 in parallel (the operations are listed in Table 1 and Table 2). We do not do any other operations in this stage, since subsequent operations are dependent on the result of operation 1 and operation 1 takes the most time. In stage 3, we perform all the remaining operations needed to compute  $p_1$  as well as operations 8 and 9. In stage 4, we perform operation 10 and codeword generation. This segmentation into four stages balances the workload across stages well for rate 1/2. In principle, we could parallelize some of the matrix-vector multiplications and forward-substitutions to get higher throughput. However, parallelizing those operations would involve duplicating the look-up tables (since dual-port RAM is the best we can get from current FPGAs), which would require significantly more area. Moreover, we can simply replicate many instances of the encoder on the same chip to process several message blocks in parallel without increasing RAM area needed for the look-up tables of the six matrices. These look-up tables can be shared among the encoder instances.

Depending on the channel conditions, codes with different rates perform better than others. For instance, when the SNR is low, higher rate codes are more appropriate. Therefore one could implement an adaptive LDPC encoder, which changes rate or block length depending on the channel conditions. Of course the LDPC encoder would have to be synchronized with an adaptive LDPC decoder. Although the architecture shown in Figure 6 could be used for different rates, it is optimized for rate 1/2 codes. Codes with different rates differ in the dimensions of the  $H$  matrix, leading to different edges ratios for the six matrices. Therefore, different scheduling of the operations are needed for different rates to maximize concurrency. Bit files of different designs optimized for different rates can be stored in memory, and run-time reconfiguration of FPGAs can be exploited to reconfigure the adaptive LDPC encoder at run-time for different channel conditions. Since reconfiguration can be performed in a matter of milliseconds on modern FPGAs, such adaptive LDPC encoders/decoders are viable options.

The main operations performed in the encoder are matrix-vector multiplication (MVM), forward-substitution (FS), vector addition (VA) and codeword generation (CWG). Codeword generation involves first constructing an intermediate codeword by writing  $(s, p_1, p_2)$  into a memory. Then according to the permutation table, which contains the information on the row permutations performed during

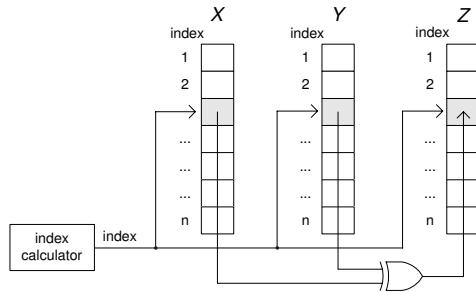


Figure 7. Circuit for vector addition.

the preprocessing step, the intermediate codeword is rearranged to generate the final codeword which is then valid with regards to the original  $H$  matrix. The hardware architectures for vector addition, matrix-vector multiplication and forward-substitution are described in the next section. Since we are dealing with a binary system, multiplications can be performed with an AND gate and additions with an XOR gate.

## 6 Components for the Encoder

### 6.1 Vector Addition

This involves the computation of  $X + Y = Z$ , where  $X$ ,  $Y$  and  $Z$  are vectors, and  $Z$  is what we are trying to compute. Since we are dealing with a binary system, vector addition can be simply achieved by performing XOR operations on the corresponding elements of the two vectors. The circuit for vector addition is shown in Figure 7. The index calculator increments the index every clock cycle.

### 6.2 Matrix-Vector Multiplication

This involves the computation of  $XY = Z$ , where  $X$  is a matrix,  $Y$  and  $Z$  are vectors, and  $Z$  is what we are trying to compute. We shall illustrate our approach with an example. Consider the multiplication of a  $5 \times 6$  matrix  $X$  by a vector  $Y$  to obtain a resulting vector  $Z$ . In this case,  $X$  is known from the preprocessing step and is sparse. It would be inefficient to store this matrix directly in a memory, since most of the locations will be zeroes. Instead, the location of the edges (ones) of each row is stored, with an extra bit indicating the end of a row. For example, if

$$X = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad (7)$$

Table 3. Matrix  $X$  stored in memory. The location of the edges of each row and an extra bit indicating the end of a row are stored.

address	0	1	2	3	4	5	6	7	8
data	3	5	1	2	4	6	0	3	4
end row	0	1	1	0	0	1	1	0	1

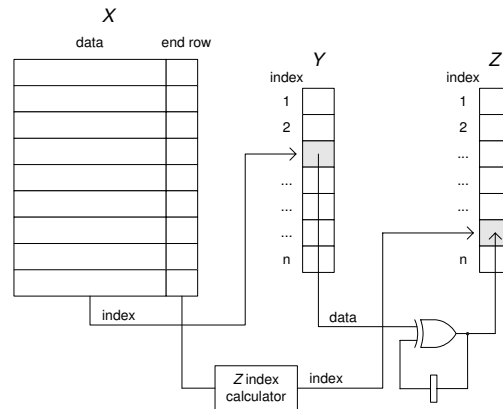


Figure 8. Circuit for matrix-vector multiplication.

it would be stored in memory as shown in Table 3. Memory #6 is a special case, indicating that the fourth row of matrix  $X$  has no edges.

The location of the edges of a row in  $X$  are used as bit selectors for the vector  $Y$ . This bit selecting process has the same effect as performing AND operations with the bits of a row in  $X$  and the bits in vector  $Y$ . XOR is performed on the selected bits to calculate the resulting bits for  $Z$ . This operation is performed for each row of  $X$  starting from the first one. Figure 8 shows our matrix-vector multiplication circuit. The  $Z$  index calculator calculates the location of the  $Z$  matrix to be written. The index is simply incremented every time there is an end of a row. It can be seen that the number of clock cycles required to compute  $Z$  is directly proportional to the number of edges in  $X$ .

### 6.3 Forward-Substitution

Consider the equation  $XZ = Y$ , where  $X$  is a lower-triangular matrix,  $Y$  and  $Z$  are vectors and  $Z$  is the vector

we want to compute.  $X$  is given by

$$\begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ x(2,1) & 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ x(3,1) & x(3,2) & 1 & 0 & \cdots & \cdots & \cdots & 0 \\ x(4,1) & x(4,2) & x(4,3) & 1 & 0 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x(n,1) & x(n,2) & \cdots & \cdots & \cdots & \cdots & x(n,n-1) & 1 \end{bmatrix}$$

One way to approach this problem is to take the inverse of  $X$  and compute  $Z = X^{-1}Y$ . However, matrix inversion is a complex procedure and requires a significant amount of processing time. Moreover, after inversion,  $X$  will be no longer sparse. A better way is to use forward-substitution exploiting the fact that  $X$  is lower triangular. The elements of the vector  $Z$  can be computed with the following set of equations.

$$\begin{aligned} z_1 &= y_1 \\ z_2 &= y_2 \oplus x(2,1)z_1 \\ z_3 &= y_3 \oplus x(3,1)z_1 \oplus x(3,2)z_2 \\ &\vdots \\ z_n &= y_n \oplus x(n,1)z_1 \oplus x(n,2)z_2 \oplus \cdots \oplus x(n,n-1)z_{n-1} \end{aligned}$$

This can be generalized as:

$$z_i = y_i \oplus \bigoplus_{j=1}^{i-1} x(i,j)z_j, \quad 1 \leq i \leq n \quad (8)$$

Just like the matrix-vector multiplication, to compute an element in  $Z$ , we need elements from  $X$  and  $Y$ . However, we also require the previous elements of  $Z$  that have been computed previously. Therefore, the circuit for forward-substitution is very similar to the one in Figure 8 with slight modifications as shown in Figure 9. The index calculator computes the memory location of  $Y$  to be read and  $Z$  to be written. From (8), these two addresses are identical. As in the matrix-vector multiplication case, the index calculator is incremented every time there is an end of a row and the clock cycles for the computation is proportional to the number of edges in  $X$ .

## 7 Implementation and Results

The preprocessor has been implemented using MATLAB. Preprocessing times for  $H$  matrices with rate 1/2 for various block lengths on Pentium-IV PC are shown in Table 4. A MATLAB tool we have developed, that constructs high-performance irregular LDPC codes with low error floors is used to generate the  $H$  matrices [11].

We can see that the preprocessing times for large block lengths can be very long. However, we are not too concerned about this since preprocessing needs to be performed only once for any given  $H$  matrix. We also observe that the

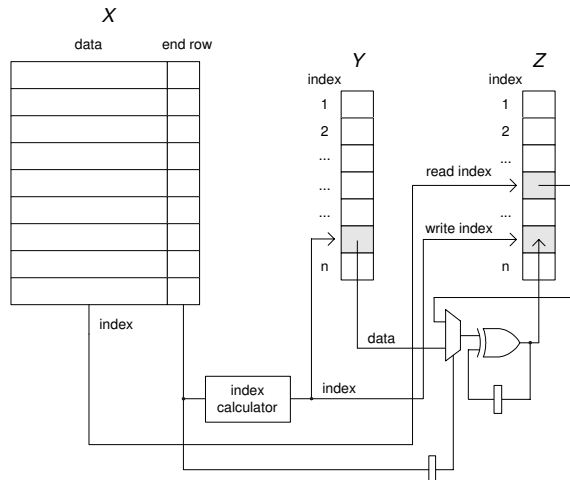


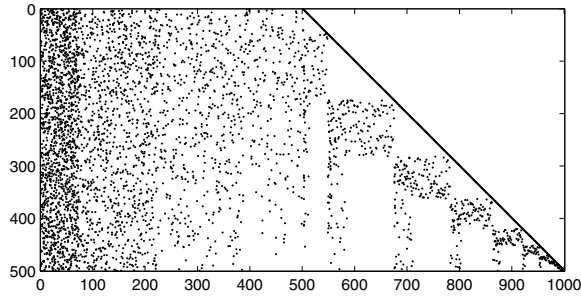
Figure 9. Circuit for forward-substitution.

Table 4. Preprocessing times and gaps for  $H$  matrices with rate 1/2 for various block lengths performed on a Pentium-IV 2.4GHz PC equipped with 512MB DDR RAM.

block length	preprocessing time (secs)	gap
500	3	2
1000	14	2
2000	83	2
4000	587	2
8000	3124	2

gap remains small even for large block lengths. The primary reason for these small gaps is the large number of rows in  $H^T$  whose degrees are less than three in their degree distributions. Low degree rows in  $H^T$  lead to high probabilities of finding degree one rows in the diagonal extension step of the greedy algorithm.

A scatter plot of a preprocessed irregular  $500 \times 1000$   $H$  matrix (i.e. block length of 1000 bits and rate 1/2) is shown in Figure 10. The diagonal ones of the matrix  $T$  can be clearly seen. Also, as expected since the gap is small ( $g=2$  in this case), the preprocessed  $H$  matrix consists of mainly  $A$  and  $T$ . The blocky artifacts next to the diagonal of  $T$  are created by the diagonal extension step of the greedy algorithm, during which an identity matrix is formed in every iteration. In Table 5, we show the number of edges for the



**Figure 10. Scatter plot of a preprocessed irregular  $500 \times 1000$   $H$  matrix in ALT form with a gap of two. Ones appear as dots.**

**Table 5. Dimensions and number of edges for the matrices  $A$ ,  $B$ ,  $T$ ,  $C$ ,  $F$  and  $E$  generated from a  $1000 \times 2000$  irregular  $H$  matrix.**

matrix	dimension	edges
$A$	$998 \times 1000$	6273
$B$	$998 \times 2$	998
$T$	$998 \times 998$	2398
$C$	$2 \times 1000$	10
$F$	$2 \times 2$	2
$E$	$2 \times 998$	6

six matrices for a preprocessed  $1000 \times 2000$  irregular  $H$  matrix. We observe that, the matrices  $A$ ,  $B$  and  $T$  contain most of the edges, indicating that operations involving them will dominate the encoding times.

The actual hardware encoder has been implemented using Xilinx System Generator. The codewords generated from our hardware encoder have been verified against our MATLAB model for correctness. Let  $e(A)$  denote the number of edges for the matrix  $A$ , and  $c(S1)$  denote the number of clock cycles taken by stage 1 (see Figure 6). The number of clock cycles taken by each stage is given by

$$\begin{aligned}
 c(S1) &= n - m \\
 c(S2) &= \max(e(A), e(C)) \\
 c(S3) &= e(T) + e(E) + (n - m) + e(P) + e(B) + (m - g) \\
 c(S4) &= e(T) + 2((n - m) + g + (m - g)).
 \end{aligned}$$

The number of clock cycles per codeword (CPC) is determined by the stage that takes the longest, i.e.

$$CPC = \max[c(S1), c(S2), c(S3), c(S4)].$$

For a given clock speed, the number of codewords per sec-

ond (CPS) is given by

$$CPS = \text{clock speed} / CPC.$$

Therefore the codeword throughput (bits per second) of the encoder is

$$\text{codeword bits throughput} = CPS \times \text{block size}$$

and the information throughput is given by

$$\text{information bits throughput} = \text{codeword throughput} \times \text{rate}.$$

The latency of the encoder is the time taken for the four stages to fill up. This is given by

$$\text{latency} = (4 \times CPC) / \text{clock speed}.$$

An encoder for block length of 2000 bits and rate 1/2 has been synthesized on a Xilinx Virtex-II XC2V4000-6 device. The design takes up 870 slices and 19 block RAMs, which uses approximately 4% of the device. The clock cycles taken by each of the four stages of this design are

$$\begin{aligned}
 c(S1) &= 1000 \\
 c(S2) &= 6273 \\
 c(S3) &= 5402 \\
 c(S4) &= 6398.
 \end{aligned}$$

We observe that the workloads across the stages are well balanced, which is the case with our architecture for all rate 1/2 codes. The design is capable of running at 143MHz and with a CPC of 6398 cycles, the resulting in a codeword throughput and latency is 45Mbps (million bits per second) and 0.179ms respectively. Implementation results for various encoders with block lengths ranging from 500 to 8000 bits for rate 1/2 codes are shown in Table 6. We see an increase in resources and latency with block length due to the increase in size of the  $H$  matrix. This increase in resources leads to reductions in clock speed and throughput due to routing delays. The usage of distributed RAMs and block RAMs have been carefully performed to minimize the waste of the 18Kb Virtex-II block RAMs. In Table 7, we show how the performance varies with different rates with a fixed block length of 2000 bits. The encoder is optimized for rate 1/2 codes (by dividing the operations shown in Figure 6), therefore we see the some performance loss for other rates.

Multiple instances of the encoder can be implemented on the same device to encode multiple message blocks in parallel. Note that RAMs for the six matrices describing the preprocessed  $H$  matrix can be shared among the encoders. This is because the six matrices are indexing the operands, and are used one by one in a linear manner. Synthesis results are shown in Table 8 for multiple instances of an encoder with block length of 2000 bits and rate 1/2. The design with 16 instances consumes 73% of the device and is capable of a codeword throughput of 410Mbps.

**Table 6. Hardware synthesis results on a Xilinx Virtex-II XC2V4000-6 FPGA for rate 1/2 for various block length.**

block length	edges	slices	block RAMs	speed (MHz)	throughput (Mbps)	latency (ms)
500	2418	562	12	161	50	0.040
1000	4859	682	13	152	48	0.084
2000	9687	870	19	143	45	0.179
4000	19452	1340	27	127	40	0.405
8000	38905	2148	49	110	34	0.937

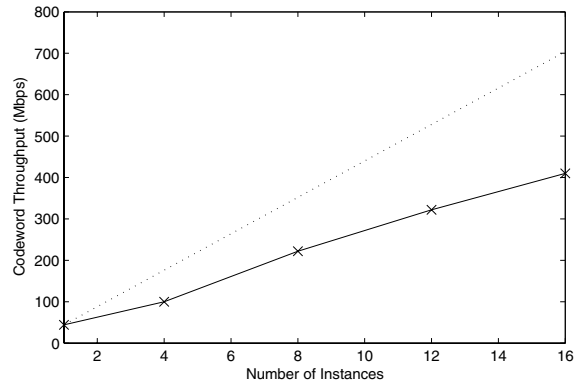
**Table 7. Hardware synthesis results on a Xilinx Virtex-II XC2V4000-6 FPGA for block length of 2000 bits for various rates.**

rate	edges	slices	block RAMs	speed (MHz)	throughput (Mbps)	latency (ms)
1/3	8896	1109	19	127	34	0.232
1/2	9687	870	19	143	44	0.179
2/3	9513	1065	18	125	33	0.235

Figure 11 shows how the number of encoder instances affects the codeword throughput. While ideally the throughput would scale linearly with the number of encoder instances, in practice the output rate grows slower than expected, because the clock speed of the design deteriorates as the number of noise generators increases. This deterioration is probably due to the increase in routing delays. Note that multiple FPGAs could be used to speed up the encoding even further. For instance, an implementation of three Xilinx Virtex-II XC2V4000-6 devices would be capable of a codeword throughput of 1.2Gbps for block length of 2000 bits and rate 1/2 codes.

**Table 8. Hardware synthesis results on a Xilinx Virtex-II XC2V4000-6 FPGA for block length of 2000 bits and rate 1/2 for different numbers of encoder instances.**

instances	slices	block RAMs	speed (MHz)	throughput (Mbps)	latency (ms)
1	870	19	143	44	0.179
4	3547	36	90	112	0.284
8	6978	60	89	222	0.288
12	12702	83	86	322	0.298
16	16906	107	82	410	0.312



**Figure 11. Variation of throughput the number of encoder instances. The dotted line shows the linear relationship between the output rate and the number of instances, if the clock speed does not deteriorate with the increasing number of instances.**

Our hardware implementations of the encoder for block length of 2000 bits and rate 1/2 has been compared to software implementations written in C. The results are shown in Table 9. It can be seen that our hardware designs are faster than software implementations by 10–300 times, depending on the device used and the resource utilization.

Regarding the feasibility of an adaptive LDPC encoder, the XC2V4000-6 FPGA has 15 million configuration bits [12]. The configuration bits can be fed to the device with eight bits in parallel at 50MHz, which is 400Mbps. So the entire device can be configured in around 35ms (smaller devices would take less time). If an adaptive LDPC encoder reconfigures itself every few seconds or tens of seconds, the overhead of the reconfiguration time would still be acceptable if the adapted encoder improves throughput and minimizes retransmission time.

## 8 Conclusion

We have described a hardware design of an efficient LDPC encoder based on the RU method. Whereas a straightforward implementation of an encoder has complexity quadratic in the block length, the RU method admits linear time encoding through careful linear manipulation of the parity matrix for both regular and irregular LDPC codes.

A preprocessor is written to optimize the parity check matrix through the row and column permutations, generating the look-up tables and parameters needed by the hardware encoder. An efficient architecture for storing and performing computations on sparse matrices has been discussed. The encoding steps have been scheduled into dif-

**Table 9. Performance comparison of block length of 2000 bits and rate 1/2 encoders: time for producing 410 million codeword bits. The software implementations are written in C and are compiled with Microsoft Visual C++ 6.0.**

platform	speed (MHz)	time (sec)
XC2V4000-6 FPGA, 16 encoder instances	82	1
XC2V4000-6 FPGA, 1 encoder instance	143	9
Intel Pentium-IV PC, 512MB DDR RAM	2400	80
Intel Pentium-III PC, 256MB SDR RAM	700	312

ferent stages optimizing concurrency while reducing idle times. Run-time reconfiguration of FPGAs can be used to load different designs optimized for various rates at run-time for an adaptive LDPC encoder.

Implementation results for encoders of various block lengths and rates have been presented. An encoder for block length of 2000 bits and rate 1/2 takes up 4% of resources on a Xilinx Virtex-II XC2V4000-6 device. It is capable of running at 143MHz resulting in a codeword throughput of 45Mbps and latency of 0.179ms. The performance can be improved by mapping several instances of the encoder onto the same chip to encode multiple message blocks concurrently. An implementation of 16 instances of the encoder on the same device at 82MHz is capable of 410 million codeword bits per second, 80 times faster than an Intel Pentium-IV 2.4GHz PC.

Ongoing and future work includes the implementation of an adaptive LDPC codec. This would involve supporting different  $H$  matrices at run-time and adaptively choosing the appropriate  $H$  matrix depending on the channel conditions, such as the signal-to-noise ratio.

## Acknowledgment

The authors thank George Constantinides and David Su-Ho Choi for their assistance. The support of Celoxica Limited, Xilinx Inc., the U.K. Engineering and Physical Sciences Research Council (Grant number GR/N 66599 and GR/R 31409), and the U.S. Office of Naval Research is gratefully acknowledged.

## References

- [1] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. In *Proc. IEEE Conf. on Comm.*, pages 1064–1070, 1993.
- [2] R. Gallager. Low-density parity-check codes. *IEEE Trans. on Inform. Theory*, 8:21–28, 1962.
- [3] R. Gallager. *Low-Density Parity-Check Codes*. MIT Press, Cambridge, MA, 1963.
- [4] C. Howland and A. Blanksby. A 220mW 1 Gb/s 1024-bit rate-1/2 low density parity check code decoder. In *Proc. IEEE Custom Integrated Circ. Conf.*, pages 293–296, 2001.
- [5] C. Jones, E. Vallés, M. Smith, and J. Villasenor. Approximate-min\* constraint node updating for LDPC code decoding. In *Proc. IEEE Military Comm. Conf. (MILCOM)*, 2003.
- [6] D. Lee, W. Luk, J. Villasenor, and P. Cheung. A hardware Gaussian noise generator for channel code evaluation. In *Proc. IEEE Symp. on Field-Prog. Cust. Comput. Mach. (FCCM)*, pages 69–78, 2003.
- [7] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman. Improved low-density parity-check codes using irregular graphs. *IEEE Trans. on Inform. Theory*, 47:585–598, 2001.
- [8] G. Mehta and H. Lee. An FPGA implementation of the graph encoder-decoder for regular ldpc codes. In *CRL Technical Report*. Communications Research Laboratory, University of Pittsburgh, 2002.
- [9] R. Morelos-Zaragoza. *The Art of Error Correcting Coding*. John Wiley & Sons, New York, NY, 2002.
- [10] T. Richardson and R. Urbanke. Efficient encoding of low-density parity-check codes. *IEEE Trans. on Inform. Theory*, 47:638–656, 2001.
- [11] T. Tian, C. Jones, J. Villasenor, and R. Wesel. Construction of irregular LDPC codes with low error floors. In *Proc. IEEE Int. Conf. on Comm. (ICC)*, volume 5, pages 3125–3129, 2003.
- [12] Xilinx Inc. Virtex-II platform FPGAs: detailed description. *Virtex-II Data Sheet*, 2003.
- [13] T. Zhang and K. Parhi. VLSI implementation-oriented (3,k)-regular low-density parity-check codes. In *Proc. IEEE Workshop on Signal Processing Systems (SiPS): Design and Implementation*, pages 25–36, 2001.