

# An FPGA Architecture for Low Density Parity Check Codes

Orlando J. Hernandez and Nathaniel F. Blythe  
Department of Electrical and Computer Engineering  
The College of New Jersey  
hernande@tcnj.edu

## Abstract

*Low density parity check (LDPC) codes are a family of linear block codes that can approach the Shannon limit to within less than a hundredth of a decibel, and along with Turbo codes are the codes of choice for all next-generation high-noise, high-rate communication systems. A generalized architecture is cost-prohibitive, and code-specific ASICs are not flexible enough for channels with dynamic noise parameters. In this paper we describe a field programmable gate array (FPGA) architecture for LDPC coding that allows for code-specific architectures while providing dynamic code selection. Gate and LUT counts in the encoder are examined for various codes, and size and timing results for different decoder parameters are compared.*

## 1. Introduction

Field programmable gate arrays have been useful tools for design and verification since their invention in 1984. Today, however, FPGAs are coming into their own as the target medium, rather than an intermediate step in chip design, as logic element counts reach into the hundreds of thousands and millions and clock speeds exceed those found in embedded systems. While FPGAs rapidly become more powerful in terms of size and speed, they still retain the main selling point – reconfigurability. The ability to dynamically reconfigure a system, either to make a small adjustment or even change its purpose entirely gives FPGA based solutions a large advantage over ASIC designs.

Low density parity check codes are a family of linear block codes constructed from sparse graphs, and exceed all other known codes in the race to the Shannon limit. LDPC codes can be found in many high-rate, high-noise next-generation technologies, notably deep-space probes, broadcast high definition television, and satellite-earth links.

Because of their near-capacity performance it is desirable to change code rates dynamically to match the channel characteristics. The complexity of encoding and decoding sparse graph codes makes generalized solutions

impractical, so ASIC designs are typically limited to a single code, or a particular code ensemble. This is inelegant; an FPGA solution can be reconfigured such that every code has an architecture tailored to its performance, any code (within the size and speed of the FPGA) can be used, and codes can be selected dynamically.

In this paper, we describe a reconfigurable architecture for encoding and decoding LDPC codes. The resulting intellectual property provides dynamic (mid-transmission) reconfigurability with code-specific architectures and no ensemble limitations.

## 2. Background

Low density parity check codes are the result of decades of work in information theory – there is a wide variety of background information that must be understood to properly use and implement LDPC codes. Most important are the fundamentals of information and communication theory – the noisy channel coding theorem, Shannon’s work on AWGN channels, convolution codes and error-correction, parity, and so on. We will cover the basic concepts that the reader must have to understand the content of this paper.

### 2.1. Information and Communication

Ultimately the goal of communication theory is the high-speed error-free transmission of data, a challenge that only exists due to noise and its relation to information. It is a basic tenet of information theory that efficiently transmitted information is indistinguishable from noise – a consequence of all possible transmitted symbols being equally likely from the perspective of the receiver (if the receiver had some advance knowledge of transmitted symbols, they need not be transmitted). Thus information to be transmitted must be modified in some way to distinguish it from noise; this is achieved by adding *redundancy*. Adding redundancy to the information increases the amount of data that must be transmitted, but allows the receiver to distinguish between noise and information, and correct errors.

The main goal of coding theory is to develop methods by which information can be transmitted with minimum redundancy and still allow for error detection and correction at the receiver. Because redundancy can take a variety of forms, codes are compared by *code rate*, the ratio of information to total transmitted data. Thus a coding method that adds as much redundancy as information would have a code rate of 0.5. A purely informational transmission would have a unity code rate. We wish to have as high a rate as possible so as to send as little additional data as necessary.

Claude Shannon's *noisy channel coding theorem* establishes a method to compute the upper bound on code rates for near lossless transmission as a function of channel characteristics. This means that we can determine the maximum code rate such that if we increase the amount of data encoded at one time (block size), the total bit-error rate (BER) of the system will decrease. The noisy channel coding theorem also states that there exists a coding method such that there is no lower bound on BER as block size increases; using the proper code we can increase the block size until the BER is as low as we desire.

Development of a code that fulfills the noisy channel coding theorem did not happen until 15 years after Shannon published "A Mathematical Theory of Communication", when Robert Gallager invented low density parity check codes. 1960's technology could not realize these codes, and it took another 33 years until David MacKay and Radford Neal rediscovered Gallager's work and the first implementations were developed [2]. Low density parity check codes are deceptively simple, but fully exploiting their error correcting capabilities is often held back by technological limitations.

## 2.2. Low Density Parity Check Codes

An LDPC code is a linear block code represented by a sparse Tanner graph. A Tanner graph is a bipartite graph with  $n$  *symbol* nodes and  $r$  *check* nodes, where  $n \geq r$ . Let  $S$  be the set of symbol nodes and  $C$  be the set of check nodes. Sparsity indicates that each symbol node has relatively few connections to check nodes; generally  $n$  and  $r$  are large to make this possible, meaning that LDPC codes operate with large block sizes. [1] A sample graph with  $n = 7$  and  $r = 3$  (which in reality is neither large enough nor sparse enough to be a good LDPC code) is shown in Figure 1.

The *code matrix*  $H \in GF(2)^{r,n}$  for this code is the adjacency matrix of the graph;  $H_{i,j} = 1$  if and only if  $C_i$  is connected to  $S_j$ . Let  $m = n - r$ . The *generator matrix*  $G \in GF(2)^{m,n}$  is constructed from the reduced row-echelon form of  $H$ ;  $H$  can be transformed to  $H = [P \in GF(2)^{r,m} \mid I \in GF(2)^{r,r}]$  via Gauss-Jordan

elimination or similar, at which point  $G = [I \in GF(2)^{m,m} \mid P^T \in GF(2)^{m,r}]$ . [5]

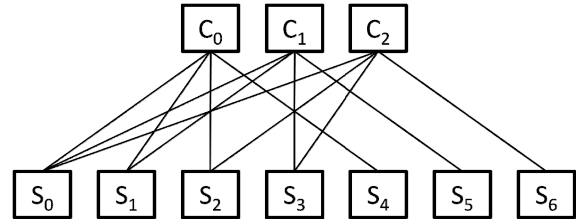


Figure 1. Tanner graph for a (7, 4) LDPC code [5]

Algebraically we can look at each check node as the sum modulo 2 of the connected symbol nodes. For the above code we arrive at the following parity check equations.

$$\begin{aligned} C_0 &= S_0 + S_1 + S_2 + S_4 \\ C_1 &= S_0 + S_1 + S_3 + S_5 \\ C_2 &= S_0 + S_2 + S_3 + S_6 \end{aligned}$$

These equations can be written in matrix form as below. The central matrix is the adjacency matrix for the graph – the code matrix  $H$ .

$$\begin{bmatrix} C_0 \\ C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \end{bmatrix}$$

$$C = H \times S$$

Looking at  $H$  we see that it is already in reduced row-echelon form, so the generator matrix can be easily computed.

$$H = \left[ \begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array} \right]$$

$$P = \left[ \begin{array}{cccc} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{array} \right]$$

$$\therefore G = \left[ \begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array} \right]$$

Both  $H$  and  $G$  completely describe the code; typically  $H$  is stored and  $G$  is generated as needed. The *code rate* of a code with code matrix  $H \in GF(2)^{r,n}$  is  $R = \frac{n-r}{n}$ , or equivalently for a code with generator matrix  $G \in GF(2)^{m,n}$ ,  $R = \frac{m}{n}$ . Equivalently, it could be said that such

a code provides  $r$  elements of redundancy. The valid codewords in  $H$  are the row vectors  $Y$  such that  $Y \times H^T = 0$ .

Encoding of a row vector  $X \in GF(2)^m$  to yield a valid codeword (row vector)  $Y \in GF(2)^n$  is performed by  $Y = X \times G$ . Decoding of  $Y$  to yield  $X$  is performed by  $X = [Y_0 \ Y_1 \ \dots \ Y_{m-1}]$ , ignoring elements  $Y_m$  through  $Y_{n-1}$ ; this is possible because the generator matrix contains an  $m \times m$  identity matrix at the front and so  $X$  is contained within  $Y$ .

In our previous example, a *source* word  $X = [1 \ 0 \ 1 \ 1]$  could be encoded as shown below.

$$Y = X \times G$$

$$Y = [1 \ 0 \ 1 \ 1] \times \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

$$= [1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1]$$

We can verify that  $Y$  is a valid codeword as follows.

$$Y \times H^T = 0$$

$$[1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1] \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [0 \ 0 \ 0]$$

Error correcting is performed by message-passing between check and symbol nodes. Typically the messages passed are log-likelihood (LLR) values, defined as below. Logarithms are natural. The probability function  $P$  is defined for the binary symmetric channel with probability of bit error  $p$ .

$$P: GF(2)^2, \mathbb{R} \rightarrow \mathbb{R} \equiv P(x = y, p) = \begin{cases} 1 - p & x = y \\ p & x \neq y \end{cases}$$

$$L: GF(2)^2 \rightarrow \mathbb{R} \equiv L(x, p) = \log \left( \frac{P(x = 0, p)}{P(x = 1, p)} \right)$$

One way to model the message passing algorithm is with a matrix  $M \in \langle \mathbb{R}, \mathbb{R} \rangle^{r \cdot n}$ . Each element in the matrix is a tuple, where  $M_{i,j}(0)$  is the message passed from check node  $C_i$  to symbol node  $S_j$ , and  $M_{i,j}(1)$  is the message passed from symbol node  $S_j$  to check node  $C_i$ . Note that because the code matrix is sparse this model is inefficient and unsuitable for implementation. When a possibly corrupted vector  $Y$  is received,  $M$  is initialized such that  $M_{i,j}(0) = L(Y_j, p) \forall i, j: H_{i,j} = 1$ . In words, elements in  $M$  that correspond to edges in the code graph are initialized with the LLR of the received symbol

corresponding to the symbol node that particular edge is attached to.

$M$  after initialization for our sample code, receiving  $Y$  as computed previously is shown below. Note that  $p$  is not indicated in the use of  $L$ ;  $p$  is constant during computation – generally a code is selected for a particular probability of bit error (the minimum redundancy necessary for a perfect code at a particular probability of bit error can be computed with the noisy channel coding theorem, and thus an LDPC code providing enough redundancy can be selected for a particular  $p$ ).

$$M = \begin{bmatrix} L(1),0 & L(0),0 & L(1),0 & - & L(0),0 & - & - \\ L(1),0 & L(0),0 & - & L(1),0 & - & L(0),0 & - \\ L(1),0 & - & L(1),0 & L(1),0 & - & - & L(1),0 \end{bmatrix}$$

Check nodes and symbol nodes then begin exchanging messages (in our model by filling in either the first or second element of each tuple for which  $H$  contains a 1), check nodes first. In other words check nodes fill in each  $M_{i,j}(1)$  for which  $H_{i,j} = 1$ , then symbol nodes fill in  $M_{i,j}(0)$  for which  $H_{i,j} = 1$ . This process is iterated, with quantization (to be discussed shortly) occurring after each iteration, until either a valid codeword is generated or a maximum number of iterations is reached. The check-to-symbol node message equation is shown below.

$$\forall i, j: H_{i,j} = 1,$$

$$M_{i,j}(1) = 2 \cdot \operatorname{atanh} \left[ \prod_{j': H_{i,j'} = 1, j' \neq j} \tanh \left( \frac{M_{i,j'}(0)}{2} \right) \right]$$

In words, each check node passes a message to each symbol node to which it is connected that consists of twice the hyperbolic arctangent of the product of the hyperbolic tangents of the messages received from all other symbol nodes to which it is connected, each halved.

The symbol-to-check node message equation is shown below [1], [4].

$$\forall i, j: H_{i,j} = 1, M_{i,j}(0) = L(Y_j) + \sum_{i': H_{i',j} = 1, i' \neq i} M_{i',j}(1)$$

Here each symbol node passes a message to each check node to which it is connected that consists of the LLR of the received bit associated with this symbol node, plus the sum of all received messages from all other check nodes to which it is connected.

After each iteration the error-corrected codeword  $\hat{Y}$  is quantized as follows [1], [4].

$$\forall j \in \{0, 1, \dots, n-1\} \hat{Y}_j = L(Y_j) + \sum_{i: H_{i,j} = 1} M_{i,j}(1)$$

$$\forall j \in \{0, 1, \dots, n-1\} \hat{Y}_j = \begin{cases} 0 & \bar{Y}_j \geq 0 \\ 1 & \bar{Y}_j < 0 \end{cases}$$

$\hat{Y}$  is then cross multiplied with the code matrix  $H$ ; if the result is 0 then the received vector has been corrected to a valid codeword, otherwise another iteration occurs. If the maximum number of iterations has occurred the received vector is released as is. Finally  $\hat{X}$  can be determined by extracting the first  $m$  elements of  $\hat{Y}$ , as discussed previously. If  $\hat{X}$  differs from  $X$  (which would not be known in a real system, of course) bit errors will occur.

### 3. Architecture

There are three fundamental data types involved in this architecture: encoded words, decoded words, and fixed point real numbers. Due to the large block sizes typically used, encoded and decoded words cannot be transferred on and off the device at once; data must be contained within shift registers so it can be exchanged with the outside world. For simplicity the same bus width is used for input and output to both the encoder and decoder. The architectures described are designed with a bus width of  $q$  bits. Decoded words have a width of  $m$  bits and encoded words have a width of  $n$  bits. Fixed point numbers are only of use to the decoder, and different widths will be examined.

Both the encoder and decoder consist of three main components, as in Figure 2.

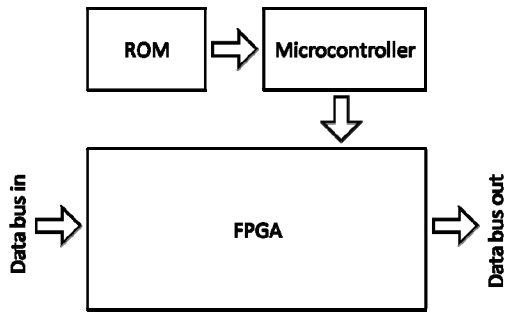


Figure 2. System architecture, top level

All code-specific architectures (bit files) are stored in the ROM (Flash ROM being most appropriate due to the size of bit files). The microcontroller can then load the FPGA with the selected code architecture, and reload during operation as desired. In this paper we will focus on the implementation of the architectures themselves, not the complete system.

### 3.1. Encoder Architecture

The encoder has the relatively simple job of multiplying an incoming vector by the generator matrix, which in reality is the computation of a series of independent sums modulo 2. The top level architecture can be seen in Figure 3.

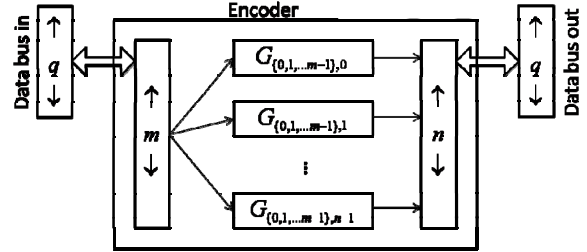


Figure 3. Encoder architecture

Each block labeled  $G_{\{0,1,\dots,m-1\},k}$  is a tree-structure of XOR operations on some subset of the incoming  $m$  bits to yield a single output bit. Note that the first  $m$  columns in  $G$  have magnitude 1 (from the identity sub-matrix), so in reality these trees would simply pass one particular bit of the input through to the output.

Total gate count is a function of the weight of the generator matrix and overlap between columns. While the code matrix  $H$  is very sparse with low overlap between rows,  $G$  can be significantly denser; as the block size increases, the ability of the synthesis tool to exploit redundancy in the XOR trees becomes important. The following table contains a comparison of total LUT count to total theoretical XOR operations for a variety of different sized codes, as synthesized by Xilinx ISE 9.1i for a Virtex 4 device with conservative optimization for area.

Table 1. Encoder synthesis

Code	Theor. XOR count	Inferred XOR count	LUT count	Max delay (ns)
96, 48	288	111	331	7.983
204, 102	612	360	1279	9.877
1008, 504	3024	4509	24643	11.62

### 3.2. Decoder Architecture

The decoder consists of a number of units: the check nodes, the variable nodes, and the quantizer. Because the code need not be regular (the number of variable nodes connected to each check node need not be constant) and the architecture should be tailored to each code, the collections of check nodes and variable nodes might not be homogenous.

Check nodes are examined first. A check node  $C_i$  has connections to  $n_i$  symbol nodes, across which messages are sent and received. To avoid implementing the hyperbolic tangent and arctangent functions in hardware we use the sign-min approximation.

$$\forall i, j: H_{i,j} = 1,$$

$$M_{i,j}(1) = \frac{\min_{j': H_{i,j'}=1, j' \neq j} |M_{i,j'}(0)|}{2 \prod_{j': H_{i,j'}=1, j' \neq j} \text{sgn}(M_{i,j'}(0))}$$

Computing the minimum of the incoming messages (excepting the message received from the symbol node to which the message we are computing is destined) and the sign of the product of the same is much easier than trigonometry. The check node architecture is shown in Figure 4. Here  $X_j$  is the message received from symbol node  $j$  and  $Y_j$  is the message to be transmitted to symbol node  $j$ .

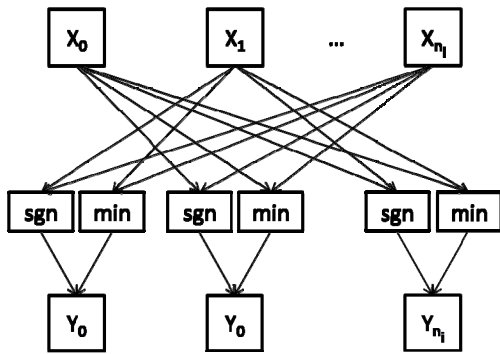


Figure 4. Check node architecture

Note that as per the algorithm, each incoming message is connected to all computation units *other* than those that generate the output for the corresponding outgoing message. A comparison of LUT counts for different message widths (bits of precision) and different weight check nodes is shown in Table 2. Synthesis was performed to optimize conservatively for speed.

Table 2. Check node synthesis

Connected nodes	Bits per message	LUT count	Max delay (ns)
4	4	108	4.987
4	8	232	4.987
4	16	456	4.987
8	4	521	5.717
8	8	1025	5.717

8	16	1953	5.717
16	4	2295	7.123
16	8	4507	7.123
16	16	8114	7.123

Next we turn our attention to the symbol nodes. A symbol node  $S_j$  has connections to  $r_j$  check nodes, as well as the message bit associated with this node. Typically the sum of all incoming messages and the LLR of the message bit is computed, and then each individual incoming message is subtracted from that sum to get each corresponding outgoing message. [2] This has the advantage of computing the LLRs for the quantizer as well (the complete sum prior to individual subtractions). The symbol node architecture is shown in Figure 5. Here  $X_i$  is the message received from check node  $i$  and  $Y_i$  is the message to be transmitted to symbol node  $i$ .

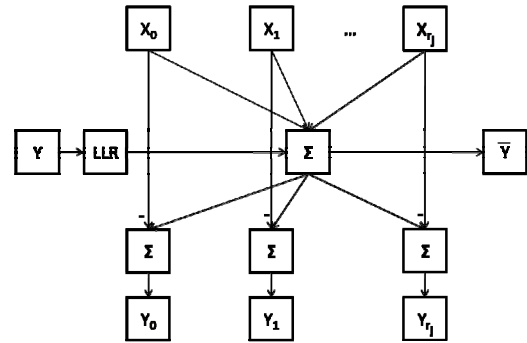


Figure 5. Symbol node architecture

The unit labeled “LLR” is simply a lookup table with two values, the LLR for a received 1, and the LLR for a received 0, both of which are constant and computed from the probability of bit error, as discussed earlier. A comparison of LUT counts for different message widths and different weight symbol nodes is shown in Table 3. As for the check nodes, synthesis was performed to optimize conservatively for speed.

Table 3. Symbol node synthesis

Connected nodes	Bits per message	LUT count	Max delay (ns)
4	4	36	8.786
4	8	68	9.502
4	16	132	9.774
8	4	72	11.228
8	8	136	12.702
8	16	264	13.544
16	4	144	15.022
16	8	272	17.717
16	16	528	20.487

Finally we implement the quantizer. Quantization has two steps: first, determining the discrete output of the error correcting process, and second, determining whether the result is an actual codeword. The discrete output can be computed directly from the sign of the LLRs calculated by the symbol nodes ( $\bar{Y}$ ), as shown in the original algorithm. The valid codeword check is performed by multiplying the candidate codeword by the transpose of the code matrix and ensuring that every element is 0. The quantizer architecture can be seen in Figure 6.

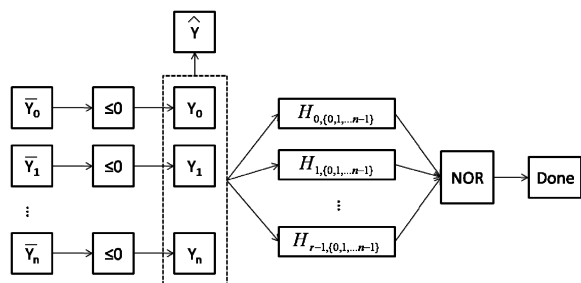


Figure 6. Quantizer architecture

Like the check and symbol nodes, the quantizer is synthesized with conservative optimization for speed. The results for various message widths and values of  $n$  are shown in Table 4.

Table 4. Quantizer synthesis

$n$	Bits per message	LUT count	Max delay (ns)
96	4	192	7.410
96	8	192	7.410
96	16	192	7.410
204	4	408	8.328
204	8	408	8.328
204	16	408	8.328
1008	4	2017	9.437
1008	8	2017	9.437
1008	16	2017	9.437

A complete decoder would consist of  $n$  symbol nodes,  $r$  check nodes, and a single quantizer, along with minimal control logic and input and output FIFOs (as in the encoder). The majority of the complexity in the design comes from the three modules presented here.

### 3.3. Available optimizations

Simplification and optimization of the encoder is a complex process. Each parity check equation (column of the generator matrix) can be rewritten in a large number of ways, due to the associativity of XOR operations. The

optimal organization for the generator matrix would be one that has the most overlap among columns, however there are too many possible combinations to be computed empirically. Synthesis tools will work to find an acceptable solution, but the processing time can be prohibitive. Thus it may be desirable to partially optimize the individual equation units in Figure 3 prior to synthesis.

The quantizer can generally not be simplified further due to the low overlap in the code matrix. The check nodes use the most area, but are faster than the symbol nodes. This is convenient because the low area of the symbol nodes means we can use additional area to speed up the computation: carry look-aheads can be implemented to reduce the worst-case propagation delay of the sum for a substantial increase in speed.

## 4. Conclusions

As the need for the capacity-approaching performance of low density parity check codes increases, so does the need for advanced architectures and mediums on which they can be implemented. FPGA solutions are needed to support code-specific architectures and remove artificial ensemble limitations imposed by ASIC designs that are not sufficient for channels with dynamic noise parameters.

The architecture presented in this paper supports any code rate and form, limited only by the available FPGA device. Decoding with the log-likelihood sum-product algorithm increases complexity, but still allows for optimization and yields acceptable areas and propagation delays. This results in a complete FPGA based low density parity check code system – a major step in the race to at-capacity communication and fundamental to supporting the information needs of next-generation technologies.

## 5. References

- [1] Hu, Xiao-Yu; Eleftheriou, Evangelos; Arnold, Dieter-Michael; Dholakia, Ajay. Efficient Implementations of the Sum-Product Algorithm for Decoding LDPC Codes. IBM Research, Zurich Research Laboratory, CH-8803 Ruschlikon, Switzerland, 2001.
- [2] Lechner, Gottfried; Bolzer, Andreas; Sayir, Jossy; Rupp, Markus. Implementation of an LDPC Decoder on a Vector Signal Processor. Proceedings of the 38th Asilomar Conference on Signals, Systems, and Computers, 2004.
- [3] MacKay, David; Neal, Radford. Near Shannon Limit Performance of Low Density Parity Check Codes. Electronics Letters, 1996.
- [4] Lee, Jason Kwok-San; Thorpe, Jeremy. Memory-Efficient Decoding of LDPC Codes. Jet Propulsion Laboratory, California Institute of Technology.
- [5] Sun, Jian. An Introduction to Low Density Parity Check Codes. Wireless Communication Research Laboratory, West Virginia University.