Implementing the NASA Deep Space LDPC Codes for Defense Applications

Wiley H. Zhao, Jeffrey P. Long

***Abstract*—Selected codes from, and extended from, the NASA's deep space low-density parity-check (LDPC) codes are implemented for high speed defense applications. This is part of an effort to build Government reference waveform implementations to assist defense acquisition programs and to promote waveform re-use. Details of the decoder implementation, including memory layout, parallelization architecture, layered-decoding scheduling, and field programmable gate array (FPGA) resource utilization are presented.**

***Index Terms*—Forward Error Correction, Low Density Parity Check Codes, FPGA**

I. Introduction

THE low-density parity-check (LDPC) code is getting utilized for high speed high performance applications due to the ease of its parallel implementation in hardware and close to capacity performance. Waveforms employing LDPC codes include DVB-S2 [1] and IEEE 802.16 [2] and lately an increasing number of defense applications have begun to incorporate LDPC codes in their waveform designs, such as the Global Positioning Systems (GPS) [3] among others [4], [5]. As waveforms in defense applications are designed or upgraded, one can expect LDPC codes to find wider application in order to take the advantage of the high speed implementation and capacity approaching performance.

In this paper, we present the implementation of four LDPC codes based on the NASA deep space LDPC codes [6]. The work is a part of an effort to build Government reference waveform implementations to help reduce risk and promote waveform re-use across related defense acquisition programs. The codes are implemented in Very high speed integrated circuit Hardware Description Language (VHDL) with the intent of employing Field Programmable Gate Arrays (FPGA) to offer a re-programmable radio solution.

The paper begins with a brief description of the LDPC codes implemented along with the decoding algorithm used (Sec. II) and continues with a description of the architectural approach utilized (Sec III). This discussion will include details on layered processing and proper scheduling as a technique to increase throughput by pipe-lining multiple layers simultaneously in the decoding engine. In addition, the memory utilization and layout will be explained in the context of the layered processing approach. The paper will conclude with FPGA resource utilization for the implementation as well as the throughput achievable and performance curves (Sec. IV). Although not presented here it should be mentioned that a high speed parallel encoder was also developed as a complement to the decoder discussed in this work.

Wiley H. Zhao is with the MITRE Corporation, Bedford, MA, e-mail: wzhao@mitre.org.

Jeffrey P. Long is with the MITRE Corporation, Bedford, MA, e-mail: jplong@mitre.org.

II. The Implemented Codes and Decoding Algorithm Description

Four LDPC codes based on NASA's deep space LDPC codes are implemented in this effort. Two of them are rate 1/2 with the coded block size of 2048 and 16384 bits and the other two are rate 7/8 with the coded block size of 4096 and 16384 bits. These code rates and block sizes accommodate both power-limited, bandwidth-limited and short, long delay scenarios as the application requires. Among these codes, rate 1/2 2048-bit code is directly from [6]. The other three are extensions from [6].

NASA's deep space LDPC codes are Accumulate Repeat-4 Jagged-Accumulate (AR4JA) codes [7]. These are structured LDPC codes designed by lifting up a protograph parity check matrix into a larger parity check matrix consisting of circulants. Parity check matrices constructed with circulants are critical in implementing parallel decoding algorithms for high speed applications. The DVB-S2 codes and the IEEE 802.16 codes are of this type. However, unlike DVB-S2 codes, which consist of superimposed circulants, the NASA deep space LDPC codes consist of only simple circulants. This significantly simplifies the layered decoder implementation[8], [9]. The layered decoding algorithm has been the preferred decoding algorithm because of its faster convergence compared to the flooding algorithm [10].

For structured LDPC codes consisting of simple circulants, each row of circulants in the parity check matrix naturally becomes a "layer". The actual rows within a "layer" (i.e., check nodes, c-nodes) can be processed in parallel because they update different variable nodes (columns in the parity check matrix, v-nodes) without memory access conflicts. The decoding proceeds as follows:

- Step 0: initialize the v-node Log-likelihood-ratio (LLR) to the channel LLR and c-node to v-node messages to zero at the beginning of a code block

$$LQ_i = LLR(x_i) \quad (1)$$

$$Lr_{ji} = 0 \quad (2)$$

- Step 1, 2, and 3 are applied to each layer at first, then repeated for each decoding iteration
- Step 1: compute the v-node to c-node messages Lq_{ij}, for each c-node j in a layer

$$Lq_{ij} = LQ_i - Lr_{ji} \quad (3)$$

- Step 2: compute the c-node to v-node messages

$$Lr_{ji} = \prod_{i' \in V_j \backslash i} \text{sign}(Lq_{i'j}) \cdot f\left(\sum_{i' \in V_j \backslash i} f\left(|Lq_{i'j}|\right) \right) \quad (4)$$

- Step 3: update v-node LLRs

$$LQ_i = Lq_{ij} + Lr_{ji} \quad (5)$$

where LQ_i is the v-node LLR; $LLR(x_i)$ is the received channel LLR for bit i; Lr_{ji} is the message from c-node j to v-node i; $Lq_{ij}$is the message from v-node i to c-node j; V_j represents the set of v-nodes connected to c-node j; $V_j \backslash i$ represents V_j excluding v-node i. The function f is defined as

$$f(x) = \log \frac{e^x + 1}{e^x - 1} = -\log \tan\left(\frac{x}{2}\right) \quad (6)$$

and it satisfies a special "addition" rule

$$f\left(\sum_i f(\beta_i) \right) \equiv \bigoplus_i \beta_i \quad (7)$$

where the pairwise "addition" is

$$\begin{aligned} \beta_1 \bigoplus \beta_2 \quad &= \quad \min(\beta_1, \beta_2) + \log\left(1 + e^{-(\beta_1 + \beta_2)}\right) \\ &\quad - \log\left(1 + e^{-|\beta_1 - \beta_2|}\right). \end{aligned} \quad (8)$$

In this implementation, (4) is computed using the λ-min algorithm [11] with $\lambda = 3$, i.e., among the v-nodes in $V_j \backslash i$, the smallest three are used to compute the c-node to v-node messages using (7) and (8). Equation (8) is implemented by a small lookup table for the log function.

III. Hardware Architecture

Hardware implementation of a LDPC decoder can be very resource intensive. A well thought out architecture approach is the key to developing a decoder that is both high throughput and small in size. In addition extra care must be taken to tailor the implementation for an FPGA target. The hardware architecture approach taken here attempts to maximize throughput and minimize hardware complexity by utilizing a parallel layered processing scheme. This takes advantage of the structured nature of these NASA LDPC codes as well as the use of circulants in building the parity check matrix.

A. Decoder memory layout and parallelization scheme

For LDPC codes with parity check matrices made of circulants of size M (i.e., each circulant is a $M \times M$ rotated identity matrix), (3), (4), and (5) can be implemented in parallel by a factor up to M. For example, M LQ's and M Lr's are accessed in parallel to update M Lq's simultaneously. This is closely related to the memory layout of LQ and Lr. In the following, we follow a similar approach used for DVB-S2 codes [12]. Fig.1 illustrates how LQ and Lr is laid out in FPGA block RAM (BRAM). LQ's (initially populated with input LLRs) are partitioned into multiple rows, each with M values. Correspondingly, Lr's are partitioned into rows with M values each. Each memory access reads or writes one row of LQ or Lr with M values. The parallel decoding processing is equivalent to processing circulant by circulant. Each valid pair of LQ row and Lr row corresponds to a circulant in the parity check matrix. Since rotations in each circulant are different, it is necessary to align LQ and Lr to a common reference so that the 1 to M element of a Lr row corresponds to the 1 to M element of the corresponding LQ row. This is achieved by circularly left rotating a LQ row by the amount indicated by the corresponding circulant rotation. After the rotation, the first element in the rotated LQ corresponds to the first element of the corresponding Lr row as indicated in Fig.1.

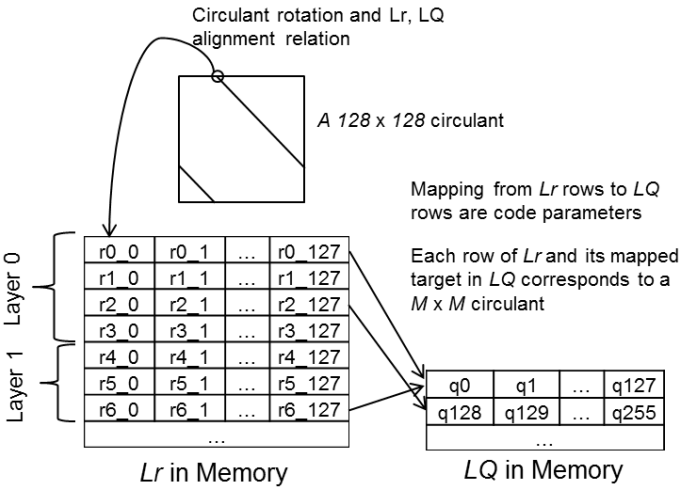

Figure 1. An example decoder memory layout of Lr and LQ for a hypothetical code with circulant size of $M = 128$. Each map from a row of Lr to a row of LQ corresponds to a circulant in the parity check matrix. The relation of the circulant rotation and the left rotation of a LQ row to align with the Lr row is indicated.

The mapping of Lr rows to LQ rows and the corresponding circulant rotations are the code parameters saved in the decoder.

For the four implemented LDPC codes, the circulant sizes are 128 (for 2048 rate 1/2), 1024 (for 16384 rate 1/2), 64 (for 4096 rate 7/8), and 256 (for 16384 rate 7/8). Full parallelization by M would require too much FPGA resources for large M (e.g., 1024). Further more, the four codes need to share the same decoder architecture so that they can be switch from one another without a complete FPGA reload. In the following, a scheme of parallelization by L, with L being a factor of M, is described following a similar approach in [13]. With this scheme, $L = 64$ is the maximum parallelization supported by all four codes implemented.

To create a parallel by L decoder, each row of LQ and Lr in Fig.1 is converted into $z = M/L$ rows by taking every other z'th element into a new row with width L. We call these smaller rows sub-rows. After the re-arrangement, the same memory access scheme now accesses sub-rows, L elements at a time. The layered decoding algorithm now works on sub-layers.

The circular left rotations that align sub-rows of LQ and Lr, and the sub-row processing order are related to the original

circulant rotation by the following

$$t_1 = \lfloor R_M / z \rfloor \quad (9)$$

$$t_2 = R_M \,\%\, z \quad (10)$$

$$A_s = (k_{sub} + t_2) \,\%\, z \quad (11)$$

$$R_s = \begin{cases} (t_1 + 1) \,\%\, L & A_s < t_2 \\ t_1 & A_s \geq t_2 \end{cases} \quad (12)$$

where $R_M \in [0,\, M-1]$ is the rotation of the circulant; $k_{sub} \in [0,\, z-1]$ is the sub-row processing order index; $A_s \in [0,\, z-1]$ is the index of the sub-row corresponding to the processing order index k_{sub}; $R_s \in [0,\, L-1]$ is the rotation of the LQ sub-row needed to align sub-rows of LQ and Lr. The symbol $\lfloor * \rfloor$ takes the integer portion of a division and $\%$ is the remainder operator. Fig.2 shows an example of this operation with a circulant of size $M = 9$ and a parallelization by $L = 3$.

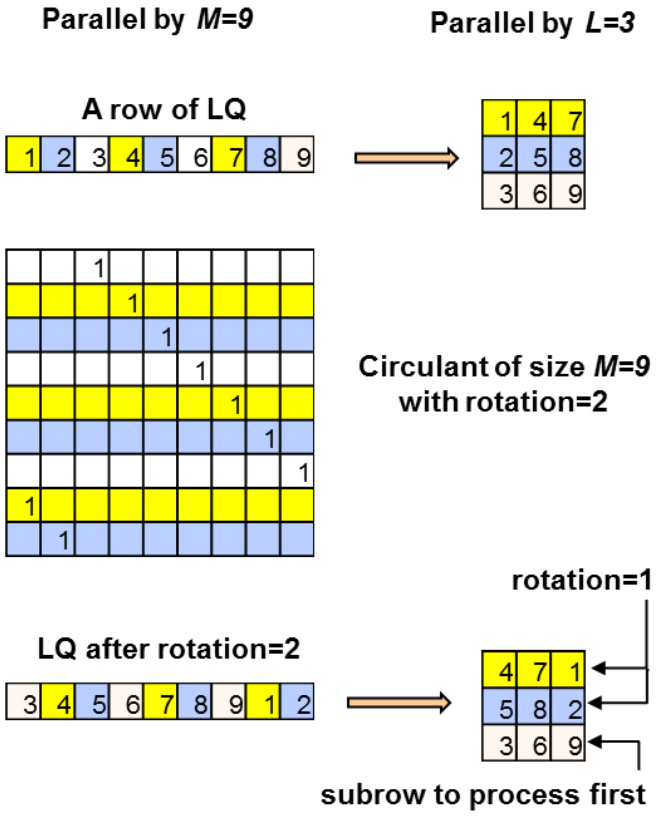

Figure 2. The example illustrates how to convert from the full parallelization by the circulant size $M = 9$ to a parallelization by $L = 3$. Different colored elements in a row of LQ are converted to $z = M/L = 3$ sub-rows. If the circulant associated with this row of LQ and a certain row of Lr (not shown) has a rotation value of 2, the left rotated LQ row for parallelization by M is shown at the bottom left. The rotation value of 2 is distributed among $z = 3$ sub-rows for parallelization by $L = 3$, the first two sub-rows are left rotated by 1 and the third sub-row is not rotated. The third sub-row is the first to be processed.

B. Decoding Core and Check Node Calculation

The decoder core shown in Fig. 3 is responsible for implementing the algorithm explained in Sec. II. Since the decoding is done in parallel, the diagram is color coded to show which elements are actually repeated in the hardware 64 times. The memory is constructed of FPGA block RAMs arranged in parallel to effectively build a single very wide word memory. The number of block RAM required depends on what the FPGA vendors offer in terms of the maximum word width per block RAM. Since the decoder is parallel by 64 each word in RAM is 64 times the width of each element. Details on the choice of metric width is given in Sec. IV. The majority of the RAM utilization is contained in the Lr and LQ RAM. The LQ RAM is initially loaded with the input LLRs as in step 0 and then continually updated each iteration. The multiplexer shown facilities that. The shifter is a simple 64 element wide barrel shifter pipe-lined to improve speed. Since the v-node to c-node calculation of step 1 is needed for the LQ updates calculated in step 3 a simple FIFO is utilized to temporarily store the step 1 results for step 3. A full memory is not usually needed here but the FIFO is still required to support the very wide word width.

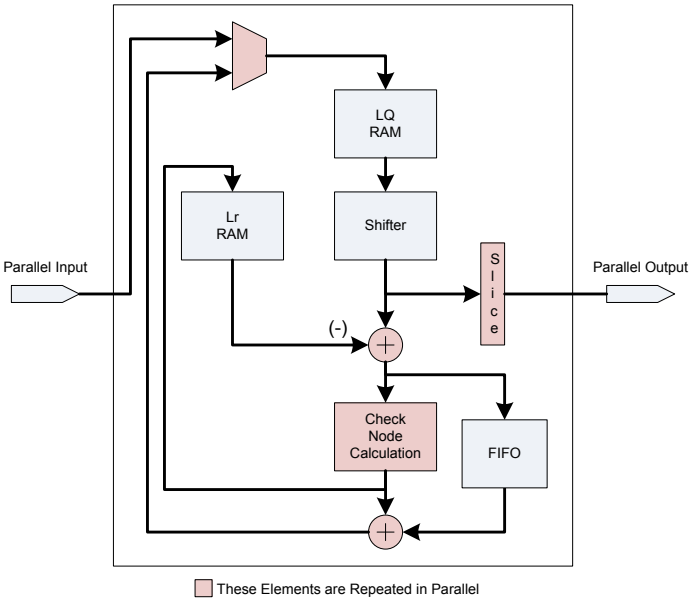

Figure 3. Decoding Core

The c-node to v-node calculation or check node calculation (Fig. 4) of step 2 utilizes a λ-min algorithm with $\lambda = 3$ as mentioned in Sec. II. This is implemented using a minimum sort routine followed by a calculation which performs the operation in (4).

The operation denoted as min* performs $\beta_1 \oplus \beta_2$ as in (8). The complete details of the min* operation are shown in Fig. 5 As mentioned the log function is implemented via a small look-up table. This table has only a few small values so it is hard-coded and synthesizes to combinatorial logic.

C. Layer Scheduling

The decoder processes multiple layers in each iteration. As a decoding iteration is essentially a process where you read memory, calculate some results and write back to memory, in theory it is possible to pipeline the operation to process

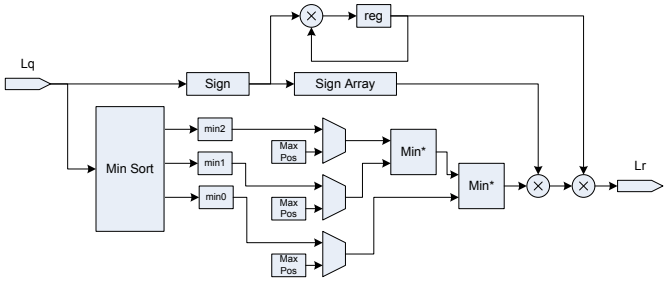

Figure 4. Check Node Calculation

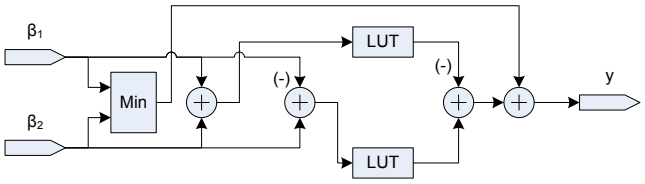

Figure 5. Min* Calculation

layers back to back and increase throughput. In practice there is latency in the calculation step and if a memory read requires data that is still processing through the pipeline then there will be an error. In our decoder implementation, layers are reordered to avoid such memory conflicts so that multiple layers can be pipe-lined to increase throughput. For codes that cannot avoid such memory conflicts, wait times are inserted in the pipeline flow so that the problem layer is completed and written into memory before the next layer is started.

D. Overall Architecture

As was shown in Fig. 3 the decoding core processes parallel sets of LQ and Lr in a single clock cycle. This creates a very efficient processing approach but does not lend itself to a very practical external interface. To facilitate data transfer in and out of the decoder core, two sets of memory are utilized. This memory serves several functions. It converts the serial stream of LLR data coming into the decoder into the wide parallel word and performs the reverse with the output data bits. It also performs the packing and unpacking of the incoming and outgoing data into the particular memory arrangement as mentioned in Sec. III-A. Lastly it acts similar to a double buffer arrangement to allow data to be continually transferred in and out while decoding. Fig. 6 shows how decoding and data transfer occurs. Since the decoding process requires the most amount of time, input data can be slowly read in and slowly read out while the decoding completes. We then take advantage of the high degree of parallelization and quickly transfer the contents in and out of the RAMs without significantly impacting throughput. A complete component (Fig. 7) includes this RAM at the input and output with additional control to synchronize the decoding and transfer functions.

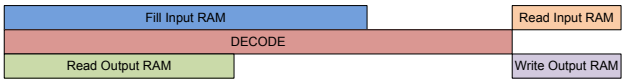

Figure 6. Decoder Processing Approach

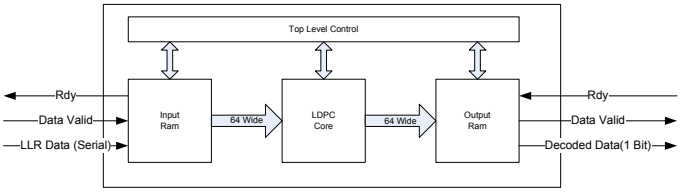

Figure 7. Decoder Top Level Diagram

IV. FPGA Resource Utilization, Throughput and Performance

This FPGA implementation was targeted at a Xilinx FPGA in the Virtex 6 family. While not the latest technology offered by Xilinx this FPGA is considered the stable choice for many of the FPGA processing cards currently offered on the market. As the results in Tab. I show the design does not exceed the size constraints of modern FPGAs and it would be possible to utilize multiple cores within a single FPGA in order to accommodate very high data rates.

Table I
FPGA Resource Utilization

Xilinx Virtex 6 LX 130	
Slice Registers	18627
Slice LUTS	25035
Block RAM (36Kb)	36

The design metric widths shown in Tab. II are perhaps the most critical in terms of determining overall size. They directly impact the number of memory elements required as well as the logic and register usage. Obviously the goal would be to make these metrics as small as possible but you can quickly encounter a loss in performance and may need to trade off size for performance.

Table II
Implementation Details

LLR Input Metric Width	5
Internal LQ Metric Width	8
Lr Metric Width	6

The decoder throughput is given in Tab. III. In the example given, the decoder is asked to do 30 iterations. The design as targeted for a Xilinx V6 is able to achieve a 175 Mhz clock rate. The throughput of each code is then calculated as shown. Obviously with fewer iterations the throughput could be increased so again a balance needs to be found trading throughput against performance. Of note is the row entitled "Extra Processing Delay". This is the extra wait time in clock

Table III
DECODER THROUGHPUT

	Rate 1/2 2K	Rate 1/2 16K	Rate 7/8 4K	Rate 7/8 16K
Clock Speed Achieved	175 MHz	175 MHz	175 MHz	175 MHz
M	128	1024	64	256
L	64	64	64	64
Rows/Iteration	120	960	252	1008
Iterations	30	30	30	30
Extra Processing Delay	75	3	149	104
Uncoded Throughput	31 Mbps	50 Mbps	52 Mbps	75 Mbps

cycles mentioned in Sec. III-C. This directly impacts the throughput and needs to be minimized as much as possible utilizing careful scheduling of the layers.

Fig.8 shows the simulated Bit Error Rate (BER) and the block, here referred as Frame, Error Rate (FER) for the four implemented codes using a fixed point decoder model running at 30 iterations.

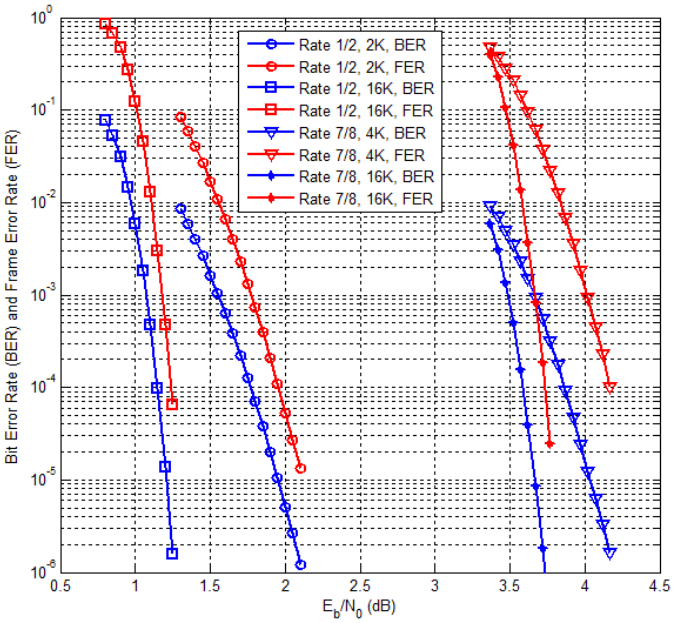

Figure 8. Simulated BER and FER curves for the four implemented codes using a fixed point decoder model running at 30 iterations.

V. CONCLUSIONS

In this work we have successfully implemented an extension to the NASA codes developed for Deep Space applications with potential use in various high-speed high-throughput defense applications. In addition we have shown how it is possible to achieve high throughput with few logic resources and still maintain decoding performance utilizing layered parallel processing. It is felt that the results presented here will be critical to helping LDPC codes find more application in Defense applications without the impact to Size Weight and Power (SWaP) as is traditionally thought. In addition the utilization of FPGAs ensures use in re-programmable radio applications. Future work will focus on further enhancements and optimizations to increase throughput and reduce size while maintaining the performance shown.

REFERENCES

[1] *Digital video broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broad-band satellite applications*, ETSI Std. EN 302 307 V1.2.1, 2009-08.

[2] *IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems*, IEEE Std. 802.16e, 2005.

[3] *Navstar GPS Space Segment/User Segment L1C Interfaces*, GLOBAL POSITIONING SYSTEMS WING (GPSW) SYSTEM ENGINEERING and INTEGRATION , Version 5.4, 15 October 2008.

[4] *High Data Rate - Radio Frequency Ground Modem, Phase 3, Modem Performance Specification*, The MITRE Corporation , Draft, 08 June 2012.

[5] *Joint Internet Protocol Modem Performance Specification*, Defense Information Systems Agency and Defense Communications and Army Transmission Systems , IS-GPS-800, 04 Sep 2008.

[6] *Low Density Parity Check Codes for Use in Near-Earth and Deep Space Applications*, Orange Book, Issue 2, Consulative Committee for Space Data Systems (CCSDS) Experimental Specification 131.1-O-2, 2007.

[7] D. Divsalar, C. Jones, S. Dolinar, and J. Thorpe, "Protograph based ldpc codes with minimum distance linearly growing with block size," in *Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE*, vol. 3, 2005, pp. 5 pp.–.

[8] M. Mansour and N. Shanbhag, "High-throughput ldpc decoders," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 11, no. 6, pp. 976–996, 2003.

[9] D. E. Hocevar, "A reduced complexity decoder architecture via layered decoding of ldpc codes," in *IEEE Workshop on Signal Processing Systems*, 2004, p. 107.

[10] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: M.I.T. Press, 1963.

[11] F. Guilloud, E. Boutillon, and J.-L. Danger, "Lambda-Min decoding algorithm of regular and irregular LDPC codes," in *3rd International Symposium on Turbo Codes and related topics, 1-5 september, Brest, France*, 2003.

[12] M. Eroz, F.-W. Sun, and L.-N. Lee, "An innovative low-density parity-check code design with near-shannon-limit performance and simple implementation," *Communications, IEEE Transactions on*, vol. 54, no. 1, pp. 13–17, 2006.

[13] M. Gomes, G. Falcao, V. Silva, V. Ferreira, A. Sengo, and M. Falcao, "Flexible parallel architecture for dvb-s2 ldpc decoders," in *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, 2007, pp. 3265–3269.