Implementation of Near Shannon Limit Error-Correcting Codes Using Reconfigurable Hardware

Benjamin Levine, R. Reed Taylor, Herman Schmit

***Abstract*— Error correcting codes (ECCs) are widely used in digital communications. Recently, new types of ECCs have been proposed which permit error-free data transmission over noisy channels at rates which approach the Shannon capacity. For wireless communication, these new codes allow more data to be carried in the same spectrum, lower transmission power, and higher data security and compression. One new type of ECC, referred to as "Turbo Codes," has received a lot of attention, but is computationally expensive to decode and difficult to realize in hardware. Low Density Parity Check Codes (LDPCs), another ECC, also provide near Shannon limit error correction ability. However, LDPCs use a decoding scheme which is much more amenable to hardware implementation. This paper will first present an overview of these coding schemes, then discuss the issues involved in building an LDPC decoder using reconfigurable hardware. We present a hypothetical LDPC implementation using a commercial FPGA, which will give an idea of future research issues and performance gains.**

***Keywords*— turbo codes, error correcting codes, low-density parity check codes**

I. Introduction

Error correcting codes (ECCs) allow effectively error-free communications to occur over noisy channels by encoding the data to be transmitted and using decoding algorithms to detect and correct errors in the received data. ECCs are already widely used in digital communication systems; however, the performance of most codes in use today falls well short of theoretical limits. Newer codes which approach these limits promise tremendous technical and economic advantages.

ECCs generally consist of two operations: encoding for transmission, and decoding to retrieve error-free data. The encoding process adds redundant information to the data being transmitted. It is this redundant information that allows for the detection and correction of errors. Some amount of computation is required to perform this error detection and correction. As the signal to noise ratio (SNR) of the transmission channel decreases, error correction becomes more difficult and may require the use of different types of ECCs.

The ability to transmit and receive data over channels with low SNRs carries many technical and economic advantages. Wireless devices employing error coding can reduce the strength of their signals. If signal strengths can be kept low, more devices can share the same frequency spectrum due to reduced inter-device interference. Lower signal strengths also decrease power requirements, an important consideration for battery-powered wireless devices. Being able to achieve low error rates over noisy channels facilitates the transmission of compressed and encrypted data, which have a low tolerance for errors. Better ECCs allow transmission of data at higher data rates as well as lower error rates. Higher data rates allow the delivery of multimedia content and other dense forms of data to portable devices. Coding schemes that are currently in wide use do not have as much error correction performance as would be desirable. The implementation of new ECCs with better properties will provide many benefits and will enable the development of new wireless devices and applications.

Shannon [1] showed that there is a limit to the rate at which data can be sent through a channel with a given SNR. This rate is called the *channel capacity*. By using a sufficiently sophisticated coding scheme, it is theoretically possible to transmit information at any rate less than or equal to the channel capacity with an arbitrarily small probability that the received data will be decoded incorrectly. Communication systems designers would like codes that allow data transmission at rates as close as possible to the channel capacity while keeping the error probability as low as necessary. ECCs that allow transmission of data at rates near the channel capacity with low probability of error are often referred to as "near Shannon limit codes." Until recently, there have been no known near Shannon limit ECCs with practical decoding algorithms. In the past several years, two general classes of practical near Shannon limit ECCs have been been developed. These are *turbo codes* and *low-density parity check codes.* These codes require computationally intensive decoding algorithms, but they allow transmission of data at low SNRs and high data rates, with error rates much lower than those afforded by other ECCs currently in use.

While the ECCs used in wireless devices are usually implemented in VLSI, and the efficient implementation of near Shannon limit codes in VLSI is an ongoing research topic (see [2], for example), there are several reasons to explore the implementation of these codes using reconfigurable hardware. These include algorithm design and testing, system prototyping and development, compatibility with diverse and evolving communications standards, and adaptive coding to match changing data and channel characteristics.

There is still substantial current research into determin-

The authors are with the Department of Electrical and Computer Engineering at Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213. E-mail addresses are {blevine, rt2i, herman}@ece.cmu.edu

0-7695-0871-5/00 $10.00 © 2000 IEEE

ing the most effective ECCs. Much of this research involves empirical testing of various algorithm parameters and must be done by simulation. Simulating ECCs can be very time-consuming in software, even with fast workstations. Reconfigurable hardware could provide a way to speed up these simulations and would also allow the testing of these algorithms with real data at system data rates. Once tested, the algorithms could be implemented using the same reconfigurable hardware for prototypes and low-volume systems.

Communications standards for wireless devices are still evolving, and near Shannon limit ECCs are not yet in wide use in these standards. By using reconfigurable hardware, wireless devices would have the flexibility to comply with new standards as they are developed and to comply with standards in use in different geographic locations.

Even when the use of these codes is well understood, and standards are well established, there will still be potential applications for reconfigurable hardware, such as adaptive error correction. A wireless device could be designed to adaptively switch between different ECCs depending on the signal environment, the data being transmitted, power requirements, and other changing parameters. For instance, when transmitting and receiving voice data to and from another portable device nearby, a simple code with low computation requirements would be sufficient, but if it was then necessary to transmit an encrypted file to a more distant device, a more complex code with much lower error rates could be used.

The purpose of this paper is threefold. First, we will introduce near Shannon limit ECCs in the context of their implementation in hardware. Turbo codes will be discussed in Section II and LDPCs will be discussed in Section III. In Section III-C, we will show reasons why LDPCs may be more amenable to hardware implementation than turbo codes. In Section IV we will introduce a possible implementation of an LDPC decoder as a starting place for further exploration. Section V presents conclusions and future work.

II. Turbo Codes

Turbo codes were introduced in 1993[3]. The term "turbo code" refers to ECCs that perform iterative, probabilistic decoding of data encoded using multiple, concatenated encoders.

A. Encoding

Turbo codes encode data by combining two or more encoders, either in parallel or serial, as well as a number of data interleavers. A typical turbo encoder is shown in Figure 2. This ***parallel concatenated turbo encoder*** uses two encoders, an interleaver, and a puncturer. The encoders are typically recursive systematic convolutional (RSC) encoders. A simple four state RSC encoder is shown in Figure 1. RSC encoders with more states are usually used in real systems, with 16 states being common. These encoders are simple to implement, as they consist of only a relatively small number of registers and modulo-2 adders. The two encoders used in a turbo encoder of this type are usually, but not necessarily, identical.

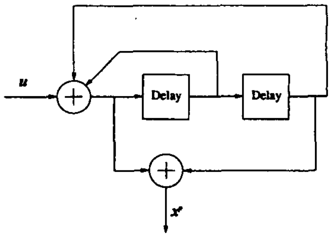

Fig. 1. Recursive systematic convolutional encoder.

The interleaver takes the input data and reorders it in a fixed, repeatable, pseudo-random manner, such that adjacent bit pairs in the original data are as distant as possible in the interleaved data. Assuming RSC encoders, each encoder produces one output bit for each input bit. This means that the turbo encoder shown would have a code rate $r = 1/3$, producing 3 bits of encoded data for every bit of input data. While codes with low rates are suitable for many applications, higher rate codes are desirable in others. When higher rate codes are needed, a puncturer can be used, as shown. The puncturer reduces the amount of data to be transmitted by throwing out some of the parity bits produced by the encoders. The decoder usually assumes that these discarded bits were all zeros (or all ones). Since less redundant information is transmitted, more computation is necessary to decode punctured turbo codes.

If our input data $\boldsymbol{u} = (u_1, \ldots, u_k)$ is a binary vector with length k, the interleaved data will be another binary vector $\hat{\boldsymbol{u}}$, also of length k, containing the same data, but in a different order. $\boldsymbol{u}$ is encoded by encoder 1, producing a binary vector of parity bits, $\boldsymbol{x}^{\boldsymbol{p1}}$, and $\hat{\boldsymbol{u}}$ is encoded by encoder 2, producing another binary vector of parity bits, $\boldsymbol{x}^{\boldsymbol{p2}}$. The subsets of each parity bit vector chosen by the puncturer will be designated $\boldsymbol{x}^{\boldsymbol{p1}'}$ and $\boldsymbol{x}^{\boldsymbol{p2}'}$, respectively.

One of the strengths of turbo encoding is that rather than using one encoder to generate one set of parity bits, multiple encoders are used to generate multiple sets of parity bits. The interleaver is designed to increase the distance between bits that are adjacent in the original data stream. Thus the data fed into the second encoder generates a set of parity bits very different from the parity bits produced by the first encoder. By using parity bits from both encoders, small amounts of redundancy provide a great deal of error correcting capability.

B. Decoding

A fairly complex decoder is needed to decode the turbo encoded transmitted data. The original data (also called the systematic data) and the parity bits from the encoder are sent over the communications channel and received corrupted by noise. The received version of the systematic data, $\boldsymbol{u}$, is designated $\boldsymbol{y}^{\boldsymbol{s}}$, and the received versions of the

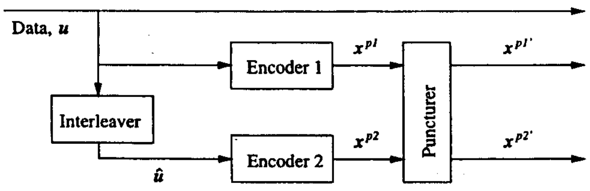

Fig. 2. Parallel concatenated turbo encoder.

punctured parity bits are designated $y^{p1'}$ and $y^{p2'}$. The received parity vectors are usually padded with zeros in those places where parity bits were thrown away by the puncturer, to produce vectors that are again k values long. The padded versions of the received parity data will be designated y^{p1} and y^{p2}, respectively.

A turbo decoder for the encoder in Figure 2 is shown in Figure 3. It consists of two decoders, one corresponding to each of the original encoders, two interleavers, and a de-interleaver. The de-interleaver simply rearranges the data in the inverse manner to the interleaver, such that data passed through the interleaver and then the de-interleaver (or vice-versa) would have its original order restored. Each decoder uses the received systematic data, parity data, and output of the other decoder to calculate probabilities about the contents of the original data. So initially decoder 1 uses y^{p1} and y^{s} to determine a set of probabilities for the values of the original data. Then decoder 2 uses y^{p2}, y^{s}, and the output of decoder 1 to refine these probabilities. These refined probabilities are passed back to decoder 1, which further refines them. The probabilities passed from one decoder to the next are referred to as extrinsic probabilities and are indicated as L^{e1} and L^{e2} in Figure 3. This iterative process repeats until either some stopping criteria are met, or more usually, after a fixed number of iterations.

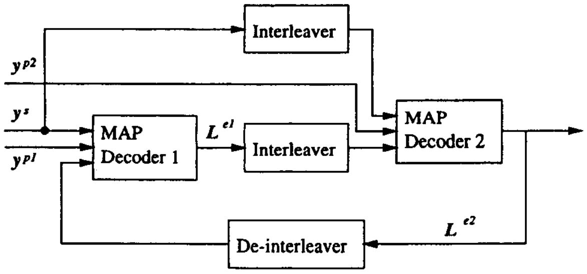

Fig. 3. Parallel concatenated turbo decoder.

The decoders used can be of any type that can decode the codes produced by the corresponding encoders. Usually a modified form of the BCJR algorithm [5] is used to decode the RSC encoders typically employed. This algorithm finds, the bit value giving the maximum *a posteriori* probability for each bit; that is, the probabilities given the observation of the received data. Decoders using this algorithm are often referred to as MAP decoders. Note that MAP decoders do not have to be used in turbo decoders; other types of decoding algorithms, such as the soft-output Viterbi algorithm (SOVA) [10] can also be used. However, MAP is a provably ideal decoder and thus gives the best performance. The basic idea of the MAP decoder is to determine the probabilities of values of bits in the original data, based on the observed values of the received data and to choose the value with the maximum probability. The probability that the ith bit in the original data, u_i was a zero, based on the observed data, is written as $P_i^0 = P(u_i = 0|y^s, y^{p1}, y^{p2})$ and the probability that this bit was a one is written as $P_i^1 = P(u_i = 1|y^s, y^{p1}, y^{p2})$. Note that the probabilities for each bit are calculated using all of the received data, not just the data corresponding to the particular bit. Determination of the decoded bit is straightforward; if $P_i^1 \geq P_i^0$ the ith decoded bit is set to be one, otherwise $P_i^1 < P_i^0$ and the ith decoded bit is set to be zero. Rather than keep track of the two probabilities, a log likelihood ratio of the probabilities, $L(u_i)$ is usually used, where $L(u_i)$ is defined as follows:

$$L(u_i) = \log\left(\frac{P_i^1}{P_i^0}\right) \tag{1}$$

The decision function is then to set the ith decoded bit to one if $L(u_i) \geq 0$ and to set the ith decoded bit to zero if $L(u_i) < 0$. The extrinsic values L^{e1} and L^{e2} are log likelihood ratios. The notation L_i^{e1} refers to the extrinsic value for $L^e(u_i)$ from decoder 1, and similarly for L_i^{e1}.

The MAP algorithm specifies how these posterior probabilities are to be determined, which is the core of the problem. A given RSC encoder has an associated trellis diagram, which details the different states that the encoder can be in, possible transitions between states, and the inputs and outputs associated with any given transition between states. The trellis diagram(s) for the encoders used must be known in order to determine these probabilities. Figure 4 shows a trellis diagram corresponding to the RSC encoder shown in Figure 1. The state that the encoder starts in is designated s_1, the next state is designated s_2, and so on through the last state, s_{k+1}. The encoder starts in State 0, so $s_1 = 0$. If the first encoder input, $u_1 = 0$, the encoder stays in State 0, meaning $s_2 = 0$, and outputs a parity bit $x_1^p = 0$, as indicated in the trellis diagram. If $u_1 = 1$, then encoder changes to State 1, so $s_2 = 1$, and $x_1^p = 1$. In this way the trellis diagram encapsulates the behavior of the RSC for all possible inputs.

It can be shown (see [5],[6]) that the maximum a posteriori probabilities for this decoding problem can be found from (2) and (3). The derivations of these equations can be found in the cited references and will not be discussed here.

$$P_i^0 = \sum_{S_0} \alpha(s_i)\gamma(s_i, s_{i+1})\beta(s_{i+1}) \tag{2}$$

$$P_i^1 = \sum_{S_1} \alpha(s_i)\gamma(s_i, s_{i+1})\beta(s_{i+1}) \tag{3}$$

The summation in (2) is over all encoder state transitions caused by an input data bit with value 0 and similarly

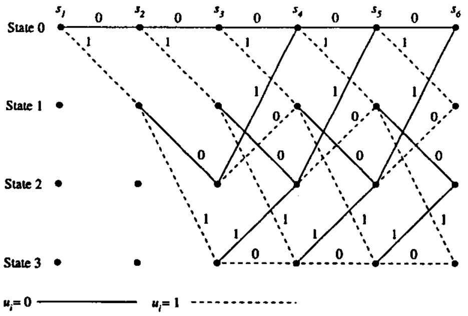

Fig. 4. Trellis diagram for four state RSC encoder. (After Fig. 6.7 in [4]).

the summation in (2) is over all encoder state transitions caused by an input data bit with value 1. Each transition consists of initial state s_i and a final state s_{i+1}. From (1), (2), and (3), we can compute the log likelihood ratio for the a posteriori data bit values as:

$$L(u_i) = \log\left(\frac{\sum_{S_1} \alpha(s_i)\gamma(s_i, s_{i+1})\beta(s_{i+1})}{\sum_{S_0} \alpha(s_i)\gamma(s_i, s_{i+1})\beta(s_{i+1})}\right) \tag{4}$$

The $\gamma(s_i, s_{i+1})$ term is the probability of a state transition from state s_i to state s_{i+1}, given that the encoder was in state s_i when bit u_i was encoded and given the received data bit y_i^s and the corresponding received parity bit y_i^p (This last is y_i^{p1} for decoder 1 and y_i^{p2} for decoder 2). $\gamma(s_i, s_{i+1})$ is referred to as the branch metric. The branch metric is used in (2) and (3), and is also needed for calculating the forward state metrics $\alpha(s_i)$ and the reverse state metrics $\beta(s_{i+1})$, as will be described below. The branch metric can be computed as:

$$\gamma(s_i, s_{i+1}) = P(s_{i+1}|s_i)P(y_i^s|s_i, s_{i+1}) \tag{5}$$

There are two components to this calculation, $P(s_{i+1}|s_i)$ and $P(y_i^s|s_i, s_{i+1})$. The first is the probability that we end up in state s_{i+1} given that we are in state s_i. Since we know the encoder trellis, this probability is either P_i^0 or P_i^1, depending on whether a data bit u_i with value zero or one produces the state transition in question. These are the values we are trying to determine, so for the purposes of the the branch metric calculation, we use the extrinsic probability, $L^e(u_i)$, from the other decoder. Since this is a log likelihood ratio, we must convert to the desired probability with (6). We know u_i in this case, since it is the data bit value necessary to create the state transition we are evaluating. A_i is a constant that will appear in the numerator and denominator of (4) and will therefore cancel, so we will ignore it.

$$P(s_{i+1}|s_i) = A_i \exp\left[\frac{(2u_i - 1)L^e(u_i)}{2}\right] \tag{6}$$

The second component to the calculation is the probability that we receive a bit y_i^s, given that the encoder made a transition from state s_i to state s_{i+1}. This depends on the encoder trellis and also on the channel properties and modulation. If we assume the channel has additive white Gaussian noise with variance σ^2, transmitted values of -1 for a 0 bit and $+1$ for a 1 bit, and code rate r, the probability can be calculated as shown in (7). Note that in addition to knowing the value u_i for the current transition, we also know what the corresponding parity bit x_i^p would be from our knowledge of the encoder trellis. The B_i term is another constant that, like A_i, will cancel in (4).

$$P(y_i^s|s_i, s_{i+1}) = B_i \exp\left[\frac{y_i^s(2u_i - 1) + y_i^p x_i^p}{\sigma^2}\right] \tag{7}$$

We can combine (5), (6), and (7) to give (8), where $C_i = A_i B_i$.

$$\begin{aligned}\gamma(s_i, s_{i+1}) = C_i &\exp\left[\frac{(2u_i - 1)L^e(u_i)}{2}\right] \\ &\cdot \exp\left[\frac{y_i^s(2u_i - 1) + y_i^p x_i^p}{\sigma^2}\right]\end{aligned} \tag{8}$$

The $\alpha(s_i)$ term in (4) represents the probability that at the time that data bit u_i was encoded, the encoder state was s_i, given the subset of received data values starting at y_1^s and ending at y_{i-1}^s. $\alpha(s_i)$ can be computed recursively from (9), where the summation is over $s_{i-1} \in A$, meaning all previous encoder states s_{i-1} that are connected in the trellis diagram to the current state, s_i.

$$\alpha(s_i) = \sum_{s_{i-1} \in A} \alpha(s_{i-1})\gamma(s_{i-1}, s_i) \tag{9}$$

The $\beta(s_{i+1})$ term in (4) represents the probability that at the time that data bit u_i was encoded, the encoder state was s_{i+1}, given the subset of received data values starting at y_k^s and ending at y_{i+1}^s. $\beta(s_i)$ can be computed recursively in a backwards fashion (starting from the end of the received data and progressing towards the beginning of the received data) from (10). The summation is over $s_{i+1} \in B$, meaning all future encoder states s_{i+1} that are connected in the trellis diagram to the current state, s_i.

$$\beta(s_i) = \sum_{s_{i+1} \in B} \beta(s_{i+1})\gamma(s_i, s_{i+1}) \tag{10}$$

In order to decode a block of data, k branch metrics must be computed from (8). Then the k forward state metrics must be found from (9), working iteratively through the data from beginning to end. The k reverse state metrics must also be calculated, using (10) and working iteratively through the data in the reverse direction. It is important to note that the state metrics for each i must be computed serially, as each is dependent on the previous value. This poses a problem for hardware implementation, as the amount of exploitable parallelism is limited by this aspect of the algorithm. The branch and state metric values can then be used to calculate the posterior probabilities using (4). These extrinsic probabilities are then passed to the other decoder, which must then do the same calculations. This constitutes one iteration of the algorithm, and there may be ten or twenty iterations before the decoding process is complete. It is clear from inspection of the relevant equations that this is a very computationally complex algorithm. Numerous multiplications and exponentials must be computed for each bit and the encoder trellis structure must be referenced to ensure that the correct terms are used in each calculation. There are some modifications that can be made to this algorithm that reduce the complexity somewhat. The so-called log-MAP variant takes the log of all the metrics so that the multiplications in (4) can be replaced by additions. The log values of the metrics can be determined directly with some manipulation of the relevant equations. This log-MAP algorithm is the usual choice for hardware implementations [2], [11], [12].

III. Low-Density Parity-Check Codes

Another, related type of ECC that can exhibit near Shannon limit performance is the low-density parity-check (LDPC) code [8][9]. Like turbo codes, they use an iterative decoding method which involves the calculation of probabilistic information that is passed from one iteration to another. The encoding schemes are quite different, however. Turbo codes use concatenated convolutional encoders and interleavers, whereas LDPCs use a parity check matrix for block encoding. As will be described, the encoding process for LDPCs is a simple multiplication of a sparse matrix and a vector. There are two main variations of LDPCs, Gallager codes and the more recent MN codes (which are a variation of Gallager codes, although they were developed independently). We will discuss Gallager codes exclusively in this paper. The differences are minor and are discussed in depth in [8].

A. Encoding

Given a binary data vector, $\boldsymbol{u}$, having length k, we can select a transmitted vector length n, giving a rate k/n code. This means we are introducing $m = n - k$ parity check bits. The transmitted vector, $\boldsymbol{t}$, is created by multiplying the source vector by a generator matrix $\boldsymbol{G}^{\boldsymbol{T}}$, such that $\boldsymbol{t} = \boldsymbol{G}^{\boldsymbol{T}}\boldsymbol{u} \bmod 2$. This generator matrix is derived from the parity check matrix, $\boldsymbol{A}$, which distinguishes one specific LDPC from another. The parity check matrix $\boldsymbol{A}$ can be created by randomly constructing an $m \times n$ matrix with exactly weight w per column (that is, there should be exactly w ones in each column) and weight $w(n/m)$ per row. This type of parity check matrix results in what is called a *regular* Gallager code; if the weight per row is not exactly $w(n/m)$, the resulting code is called an *irregular* Gallager code. Irregular Gallager codes have better error correcting properties [8] but it can be more difficult to implement decoders for irregular codes in hardware, as will be discussed below. There are other properties of parity check matrices that affect their performance, but they do not affect the decoding process and will not be addressed here.

We can use Gaussian elimination and reordering of the columns of $\boldsymbol{A}$ to produce an equivalent parity check matrix $\boldsymbol{H}$, of the form $\boldsymbol{H} = [\boldsymbol{P}|\boldsymbol{I_m}]$, where $\boldsymbol{P}$ is an $m \times k$ matrix containing the actual parity checks and $\boldsymbol{I_m}$ is the $m \times m$ identity matrix. From this form of the parity check matrix, we can create the generator matrix as:

$$\boldsymbol{G}^{\boldsymbol{T}} = \begin{bmatrix} \boldsymbol{I_k} \\ \boldsymbol{P} \end{bmatrix}$$

where $\boldsymbol{I_k}$ is the $k \times k$ identity matrix.

The complete LDPC coding scheme is shown in Figure 5. The source vector $\boldsymbol{u}$ is fed into the encoder to produce the encoded vector $\boldsymbol{t}$. The encoded vector is then sent through the channel and corrupted by noise. The corrupted version of this vector, as received by the decoder, is designated $\boldsymbol{r}$. The decoder uses $\boldsymbol{r}$ and knowledge of the encoder to decode the data. The operation of the decoder will be discussed next.

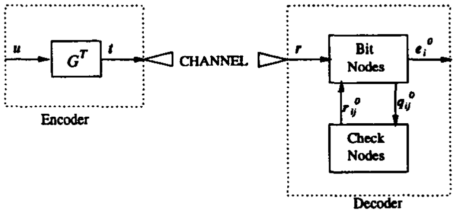

Fig. 5. LDPC encoder and decoder.

B. Decoding

The algorithm used to decode LDPCs is a message-passing algorithm known as the sum-product algorithm. It appears quite different from the MAP algorithm used for turbo coding, but it is theoretically related [7]. Similar to the algorithms used in the turbo decoder, the message-passing algorithm determines the a posteriori probabilities for bit values based on a priori information, improving the accuracy of these calculations with each iteration.

The decoding algorithm for LDPCs can be thought of as a bipartite graph, where two sets of nodes perform computations in parallel, then communicate with each other over connections described by the edges of the graph. The messages communicated between nodes consist of estimates of probabilities. The nature of the nodes in the graph and the structure of the graph's interconnections are completely described by the number and location of ones in the parity check matrix $\boldsymbol{A}$. There are two kinds of nodes, check nodes and bit nodes. The check nodes determine the probability that a parity check is satisfied if one particular data bit is set to be a one (or zero) and the other data bits have values with a probability distribution corresponding to the known a priori probabilities. The bit nodes determine the probability that a data bit has the value one (or zero), given the information from all of the other checks. Only bits and checks that are related by having a one at a specific corresponding location in the parity check matrix need to be considered in these calculations.

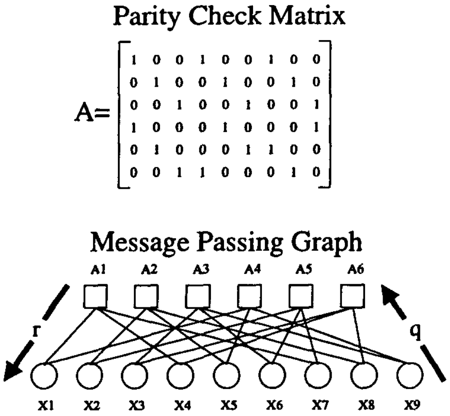

Fig. 6. The message-passing structure of a 6x9 LDPC.

As an example, take the matrix and graph shown in Figure 6. The square nodes along the top row of the graph represent the check nodes, of which there is one for each row of the matrix A. Each row represents a single parity check. Similarly, the round nodes at the bottom of the graph represent the bit nodes, of which there is one for each column in A, and thus one for each transmitted bit. The location of the ones and zeroes in A determine which nodes are connected in the message passing graph. Having a one at row j, column i simply indicates that check node j is connected to bit node i. Looking at the first row of A in Figure 6, one can see ones in the first, fourth, and seventh columns; this is reflected in the graph as connections between check node A1 (represented by the first row) and bit nodes X1, X4, and X7 (each represented by their respective columns). The number of ones in a row determines the number of ports that the corresponding check node will have, and the number of ones in a column will determine the number of ports that the corresponding bit node will have. In the case of regular parity check matrices, the total number of ones in each row will be equal to all other rows, and likewise for columns.

The check nodes generate r_{ij} values, where r^0_{ij} is the probability that check j is satisfied if it is assumed that data bit $t_i = 0$ and where r^1_{ij} is the probability that check j is satisfied if it is assumed that data bit $t_i = 1$. These probabilities are computed as shown in (11) and (12). The notation $i' \in row[j]\backslash\{i\}$ simply means the indices i' $(1 \leq i' \leq n)$ of all bits in row j $(1 \leq j \leq m)$ which have value 1, not including the current bit index, i.

$$r^0_{ij} = \frac{1}{2}[1 + \prod_{i' \in row[j]\backslash\{i\}} (q^0_{i'j} - q^1_{i'j})] \tag{11}$$

$$r^1_{ij} = \frac{1}{2}[1 - \prod_{i' \in row[j]\backslash\{i\}} (q^0_{i'j} - q^1_{i'j})] \tag{12}$$

The bit nodes generate the q_{ij} values, where q^0_{ij} is the probability that bit $t_i = 0$, given the values of all checks other than j and q^1_{ij} is the probability that bit $t_i = 1$, given the values of all checks other than j. These probabilities are computed as shown in (13) and (14). The notation $j' \in col[i]\backslash\{j\}$ means the indices j' $(1 \leq j' \leq m)$ of all checks in col i $(1 \leq i \leq n)$ which have value 1, not including the current check index, j. α_{ij} is a normalizing value chosen so that $q^0_{ij} + q^1_{ij} = 1$.

p^0_i and p^1_i represent the current estimate of the posterior probabilities for each bit. These are the extrinsic values, as discussed below, for all iterations after the first. For the first iteration, they are initialized to values determined by the data received from the channel. For instance, if the channel demodulator determines that the signal received is close to that expected for a one, it would assign a high p^1_i. It is acceptable if the channel supplies binary values for p^0_i and p^1_i.

$$q^0_{ij} = \alpha_{ij} p^0_i \prod_{j' \in col[i]\backslash\{j\}} r^0_{ij'} \tag{13}$$

$$q^1_{ij} = \alpha_{ij} p^1_i \prod_{j' \in col[i]\backslash\{j\}} r^1_{ij'} \tag{14}$$

In addition, the bit nodes also calculate the extrinsic probabilities, e_i, which are the computed posterior probabilities of bit t_i having a given value; i.e., e^0_i is the computed probability that bit $t_i = 0$. These extrinsic probabilities are used to determine what the decoded values are for each bit and are used in the bit nodes equations (13) and (14). The accuracy of these probabilities improves with each iteration of the algorithm. The extrinsic probability calculations are performed as shown in (15) and (16). The notation $j' \in col[i]$ means the indices j' $(1 \leq j' \leq m)$ of all checks in col i $(1 \leq i \leq n)$ which have value 1. α_{ij} is another normalizing value chosen so that $e^0_i + e^1_i = 1$.

$$e^0_i = \alpha_i \prod_{j' \in col[i]} r^0_{ij'} \tag{15}$$

$$e^1_i = \alpha_i \prod_{j' \in col[i]} r^1_{ij'} \tag{16}$$

C. Comparison to Turbo Codes

A comparison of the complexity of MAP decoding, log-MAP decoding, and LDPC decoding is shown in Table I. These numbers assume a rate $r = 1/3$ LDPC code and a turbo code with with 16 encoder states. The numbers shown are per iteration and per data bit; for instance, if there are 4000 data bits and 10 iterations are performed, then the total number of computations required is 40,000 times the numbers shown in the table. There are many different ways to implement these algorithms and differing complexity numbers will result from each, but these values are sufficient for general comparison of the type and number of operations required for each decoder. The message passing algorithm for decoding LDPCs is somewhat less complex in terms of operation count than the MAP

TABLE I
DECODER COMPLEXITY IN OPERATIONS PER BIT FOR EACH ITERATION.

Decoder	Add/Sub.	Mult.	Bit	Log	Exp	Div
MAP	17	97	1	1	1	2
log-MAP	192	2	64	32	32	0
LDPC	13	23	14	0	0	6

algorithm for decoding turbo codes, as shown in Table I. The check and bit node equations are relatively simple, and there are no logs or exponentials as there are for MAP decoding.

LDPCs have other properties that make them even more advantageous for implementation in hardware, however. The primary advantage is that there is much less data dependency in the LDPC algorithm. The calculations in each bit node are independent of all other bit nodes, and the calculations for each check node are independent of all other check nodes. Thus all of the check nodes could potentially be implemented so as to operate in parallel, as could all of the bit nodes. If one views the graph in Figure 6 as a data dependency graph for the LDPC decoder, it can be seen that it is quite wide and relatively shallow, meaning that there is a great deal of potentially exploitable parallelism.

The MAP algorithm requires the serial calculation of all of the state metrics. The trellis diagram in Figure 4 can be seen as an approximation of a portion of the data dependency graph for the MAP algorithm. If the trellis diagram is extended to length k and rotated 90 degrees, the resultant data dependency graph is narrow, and very deep, meaning that there is relatively little exploitable parallelism in the MAP algorithm. The only parallelism is across the encoder states, the number of which is relatively small (generally less than 64) compared to k, the number of serial calculations (typically in the hundreds or thousands).

In other words, in their most parallel possible implementation, LDPCs consist of tens of thousands of relatively simple computational elements all operating in parallel. Turbo codes, on the other hand, consist of a small number of parallel elements performing hundreds of thousands of sequential calculations. Clearly, LDPCs are preferable to turbo codes in terms of the exploitable parallelism they exhibit. The fully-parallelized LDPC algorithm has an additional advantage, in that the encoder properties are contained by the graph structure and thus no encoder information has to be stored explicitly, as the trellis information must be for MAP decoders.

D. Nodes

The message-passing algorithm is implemented by constructing a network of bit nodes and check nodes. The functionality of the nodes themselves depends only on the number of ports (or, the number of ones in the appropriate row or column) and not on the location of the nonzero bits. This means that a regular parity check matrix will define a system with only two unique kinds of nodes in it: bit nodes with some fixed number of ports, and check nodes with some other number of ports. This characteristic makes this algorithm attractive for implementation in hardware, since it can be implemented by a replicating a number of identical sub-units. Building a complete network of nodes is quite simple: merely replicate the appropriate nodes to match the number of rows and columns, then connect them according to the location of the ones in A. Irregular parity check matrices require sub-units with differing numbers of ports and thus it is somewhat more difficult to construct hardware decoders for irregular codes. Appropriate use of reconfigurable hardware can reduce the significance of this problem. Graphical representations of the internal structure of the bit and check nodes are shown in Figures 7 and 8.[1]

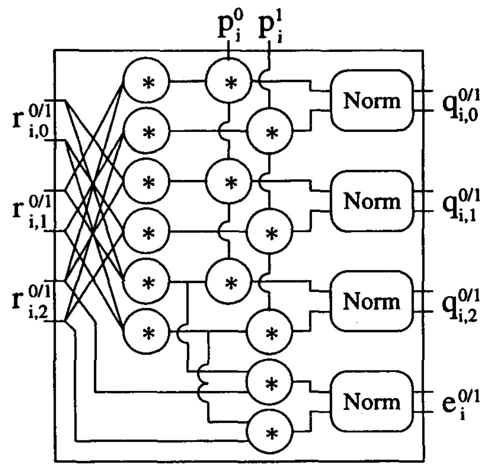

Fig. 7. Bit node i (with 3 ports)

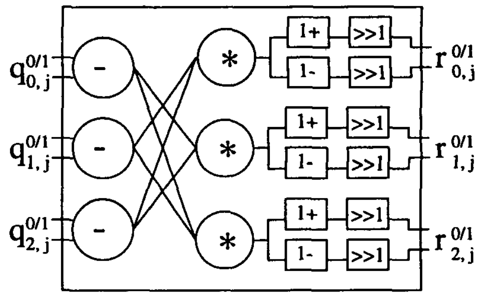

Fig. 8. Check node j (with 3 ports)

The bit nodes require normalizers for the values of q^0_{ij} and e^0_i to ensure that $q^0_{ij} + q^1_{ij} = 1$ and $e^0_i + e^1_i = 1$. Each of these pairs of values should always sum to 1 due to the fact

[1] Note that the bit nodes depicted have three ports rather than two, meaning that they are not identical to the bit nodes depicted in Figure 6. This was done for illustrative purposes, because bit nodes with two or fewer ports are deceptively simple in that they do not require two levels of multiplication.

that they represent a complete set of probabilities, but the calculated values will not usually have this property. The normalizer finds the scale factor α_{ij} for (13) and (14) and the scale factor α_i for (15) and (16). The scale factor is simply the reciprocal of the sum of the probabilities; for instance, $\alpha_{ij} = 1/(q_{ij}^{0'} + q_{ij}^{1'})$, where $q_{ij}^{0'}$ is the value calculated from (13) before normalization. The division required for normalization can be difficult to implement efficiently in hardware. One efficient implementation for the normalizer is shown in Figure 9. This circuit generates a correctly scaled dividend in the module labeled Divgen so that the quotient computed by the 8 Divcell modules has 8 bits of precision. The Varshift module rescales the result after the probability is multiplied by the scale factor. Only the normalized version of one probability needs to be produced, since the sum of both is known to be equal to one after normalization. The division is performed using a digit-recurrence algorithm [13] and the implementation of each Divcell is shown in Figure 10. Each Divcell is identical. The value for W_0 supplied to the first Divcell is the dividend, and the last Q value is the quotient. R is used internally only. This normalizer design can be easily pipelined for best performance.

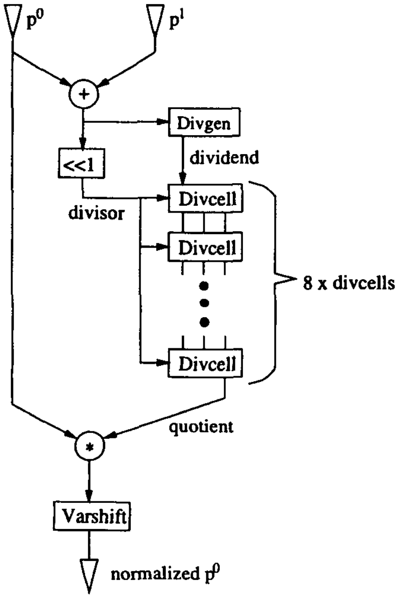

Fig. 9. Normalizer Circuit.

IV. Implementation of LDPCs

The bit and check nodes do not scale gracefully as their number of ports increases, primarily as a result of the multipliers and normalizers in the nodes. For instance, a 16-port bit node would require 32 15-operand multipliers, 32 2 operand multipliers, and 17 normalizers! Fortunately, even the very large parity check matrices currently used in the field are so sparse that they typically have 6 or fewer ones per column.

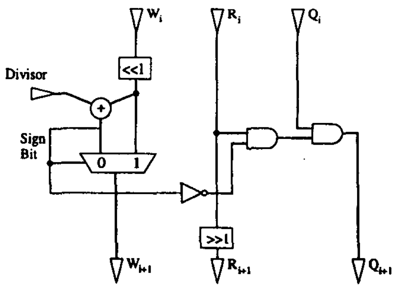

Fig. 10. Divider Cell.

On the other hand, the hardware needed to implement a parity check matrix A scales linearly with the size of A, given a fixed number of ones in every row and column. In other words, if the column weight is held constant, adding additional columns and rows to the matrix simply adds additional nodes to the graph; it does not increase the complexity of the nodes. This means that as long as an implementation of an LDPC decoding scheme fits in its entirety on a device, the design can be "grown" by simply adding more nodes and connecting them to their neighbors in the graph.

Bit nodes, and to a lesser extent, check nodes are quite large. On large reconfigurable devices, it should be easy to fit several of each; however, implementing very large numbers of nodes, or nodes with large numbers of ports, could be difficult. Figure 11 shows how many LUTs are consumed by check nodes (of varying bit-widths and numbers of ports) implemented on a Xilinx Virtex chip.

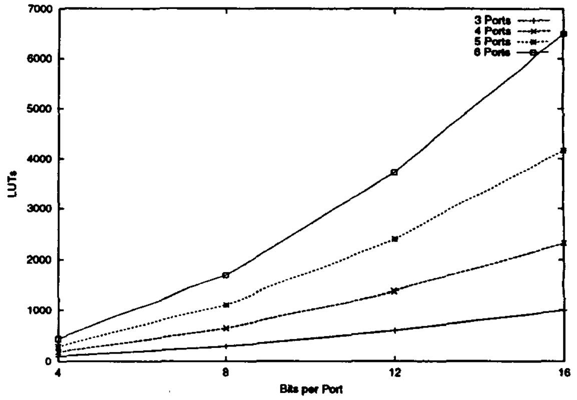

Fig. 11. LUTs consumed by a check node with various bit-widths and port counts

This means that the massively parallel implementation discussed in Section III-C will probably not fit on current reconfigurable devices. Fortunately, there are many possible approaches to building a complete LDPC decoder on a reconfigurable device which can only fit a few bit and check nodes. We will take a close look at one particular simple design, taking note of some of the issues involved in its implementation, and making some estimates as to its performance. We will also discuss a few other options for designing such a system.

A. A Two-Node System

In order to better understand the nature of the issues involved in implementing an LDPC decoder, consider a naïve two-node implementation. In this system, we will implement one bit node and one check node (like those shown in in Figures 7 and 8) on the reconfigurable device simultaneously, and then sequence the r and q values through them. The values will be stored in memory between iterations. A system of this kind is depicted in Figure 12.

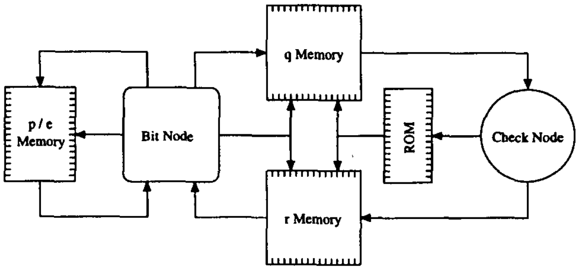

Fig. 12. Block diagram of a complete system for performing LDPC Decoding

This implementation functions as follows: The check node reads groups of values out of the q memory, processes them sequentially, and writes the resulting values into the r memory. Then, the bit node takes over, reading r and p values, and producing q and e values. Thus each location in memory corresponds to one connection in the bipartite graph, such as the one shown in Figure 6. The values themselves are represented in a fixed-point format with the radix positioned to represent probabilities ranging from 0 to 1. A savings in memory size can be achieved by only storing the r_0, p_0, q_0, and e_0 values; the r_1, p_1, q_1, and e_1 quantities can be recovered by subtracting from 1.

Accessing the values in memory is not complicated: if there are 4 ports into the check node, and 3 into the bit node, then each cycle the memory must be able to provide the appropriate number of values. For our simple implementation, a single-ported memory was used to limit the complexity of the memory. The port was clocked repeatedly to fetch and store the three or four values one at a time.

The bit node accesses the values in simple $(1 \dots N)$ order on its memory ports. The check node does the same over $(1 \dots M)$, but uses a ROM to reorder the sequence of values loaded to match the structure of the bipartite graph. Because the order and groupings of the memory accesses were known beforehand, the addresses in this ROM can easily be precomputed, and changing between one code and another (with the same M, N, and column weight) is just a matter of changing the values stored in the ROM.

TABLE II
Resources consumed by LDPC Decoder Components

Component	Resources	Percentage
Normalizers	1644 LUTs	54.1%
Bit Node	848 LUTs	27.9%
Check Node	512 LUTs	16.8%
Controller	35 LUTs	1.2%
Data Memory	9600 Bytes	64.0%
Address ROM	5400 Bytes	36.0%

An instance of the system shown in Figure 12, designed to run a rate 1/4 code with a block size of 1200 on a Xilinx Virtex chip took 3039 LUTs and a total of 9600 bytes of RAM, which is small enough to fit on many devices available today. This figure was achieved without the use of any special multipliers or memory structures. The multipliers were simply those generated by the synthesis tools, and the memory used was the on-chip RAM.

The synthesis tool and the chip specifications indicate that this implementation would be able to complete full bit and check node calculations at a frequency of 50 MHz. If the block size of the desired code is 1200 bits, and decoding is run for 10 full iterations before returning, this Virtex implementation will fully decode bits at a rate of 4 megabits per second. Since the code rate is 1/4, user data will be processed at 1 Mb/s.

One drawback to this implementation is that the bit and check nodes cannot operate simultaneously or they may overwrite data which has not yet been processed by the other node. This could be remedied by doubling the memory capacity, creating two banks each for the q, r, p, and e memories. Then, the nodes could operate on two datasets simultaneously, swapping between banks so as not to interfere with one another. This simple improvement would prevent half of the hardware from remaining idle during execution, and would double the performance of the algorithm.

This straightforward implementation shows that even if only two nodes of the bipartite graph described by the parity check matrix will fit on a device, the decoding algorithm can still be made to operate at a useful speed. Note that the implementation described here was designed for simplicity, and that optimizations to this implementation (e.g. using more efficient multipliers) or taking a more sophisticated approach to implementing the graph will bring considerable improvements in speed.

It is also worth noting that the ability to implement the entire graph at once on a device, exploiting all available parallelism, may not be as far off as it might seem. It will be several years before device capacity grows to the size necessary to implement LDPCs in their entirety, but modifications and simplifications may sufficiently reduce the hardware requirements of the decoder much sooner. For example, translating the entire algorithm into the log domain would turn the large multipliers into simple adders, bringing the hardware size down tremendously. Unfortunately, it is not obvious how to perform normalization in the log domain. Still, it is quite possible that commercial FPGAs available in the near future will be able to fit optimized LDPC decoders in their entirety.

B. Related Work

It is not easy to compare the performance figures given above to those reported in other research. LDPCs are new enough that there are no published results available for LDPC implementations on hardware at this time. Even the information on turbo code implementations is sparse; additionally, it is not generally clear how to fairly compare performance results for LDPCs with those for turbo codes. Reported performance figures for turbo decoder implementations, especially commercial products, often do not include the code characteristics, number of iterations, or actual supported data rates, making verification and comparison of performance even more difficult. In order to accurately compare ECC implementations, values for the actual error rate and user data rate, across identical channels and with identical channel modulation would need to be compared. Nonetheless, a survey of turbo decoder implementation performance results is included here, which generally fall into a range comparable with the results given above. Masera et al designed a CMOS implementation of a log-MAP decoder in 0.5 micron CMOS technology [2]. Their simulations indicate that this design should support a 2 Mbit/s data rate. The design is intended for deep space applications. Hong et al designed a comparable decoder in 0.6 micron CMOS [14]. Pietrobon developed a prototype multi-board system using FPGAs to implement the log-MAP algorithm with a 356 kbit/s data rate [12]. Several commercials turbo decoder implementations are now available. Advanced Hardware Architectures offers an ASIC that they claim can perform decoding using a proprietary algorithm at rates of 36 Mbit/s for 2 iterations only [15]. They are using a somewhat different type of coding than described herein, but claim similar error correction performance to turbo coding. Comatlas offers a turbo decoder ASIC with claimed 40 Mbit/s decoding rate [16]; no details as to the algorithm used or number of iterations were available. Small World Communications offers turbo decoder cores for Xilinx XC4000XV FPGAs and claim 31 Mbit/s decoder rates [17]. The supported data rates will be lower, proportional to the number of iterations.

V. Conclusions

Recently, several new families of error correcting codes have emerged which enable error-free communications at lower SNRs with higher bandwidth. In this paper, we examined two of these codes: turbo codes and low density parity check codes. They were evaluated in terms of their amenability to implementation in hardware. We concluded that LDPCs are more suitable because their decoders exhibit much more exploitable parallelism, the computations require operators that are more easily implementable in hardware, and they are more regular and tileable in nature than turbo decoders.

We presented an illustrative design which is implementable on present-day commercial FPGAs, and which should display performance comparable to current VLSI turbo code implementations. When devices grow or algorithms improve to allow the maximum amount of parallelism displayed by LDPCs to be exploited, LDPC decoding will be implementable as a tremendously high-speed procedure, enabling very profitable applications in wireless, digital storage, and other domains.

Acknowledgments

The authors would like to acknowledge the support of DARPA TTO under contract DABT 63-96-C-0083, and of Chameleon Systems.

References

[1] C.E. Shannon, "A mathematical theory of communications," *Bell Syst. Tech. J.*, vol.27, 1948, pp. 379-423, 623-657.

[2] G. Masera, G. Piccinini, M. Roch, and M. Zamboni, "VLSI architectures for turbo codes," *IEEE Tran. VLSI Sys.*, vol.7, no.3, Sep. 1999, p. 369-379.

[3] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and cecoding: Turbo codes," *Proc. 1993 Int. Conf. Comm.*, Geneva, Switzerland, May 1993, pp. 1064-1070.

[4] B. Sklar, *Digital Communications: Fundamentals and Applications*, Prentice Hall, 1988.

[5] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rates," *IEEE Trans. Information Theory*, Mar. 1974, p. 284-287.

[6] W. E. Ryan, "A turbo code tutorial," Unpublished, available at `http://www.ee.virginia.edu/research/CCSP/turbo_code/overview.html`

[7] R. McEliece, D. MacKay, and J. Cheng, "Turbo-decoding as an instance of Pearl's belief propagation algorithm," *IEEE J. Selected Areas in Comm.*, Vol.16, Feb. 1998, 140-152.

[8] D.J.C. MacKay, "Good error-correcting codes based on very sparse matrices,"

[9] R. G. Gallager "Low Density Parity Check Codes," *IRE Trans. Inform. Theory*, Jan. 1962, pp. 21-28.

[10] J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft-decision outputs and and its applications," *Proc. GlobeCom 1989*, pp. 1009-1013.

[11] S. Halter, M. Oberg, P.M. Chau, and P.H. Siegel, "Reconfigurable signal processor for channel coding and decoding in low SNR wireless communications," *1998 IEEE Workshop on Signal Processing Systems*, pp. 260-274.

[12] S.S. Pietrobon, "Implementation and performance of a turbo/MAP decoder," *International J. of Satellite Communications*, vol.16, no.1, Jan.-Feb. 1998, p. 23-46

[13] M.D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer, 1994.

[14] S. Hong, J. Yi, and W. Stark, "VLSI design and implementation of low-complexity adaptive turbo-code encoder and decoder for wireless mobile communications applications" *1998 IEEE Workshop on Signal Processing Systems*, p. 233-242.

[15] Advanced Hardware Architectures, Pullman, WA, USA, "AHA4501 Astro: 36 Mbit/sec turbo product code encoder/decoder," datasheet, available at http://www.aha.com

[16] Comatlas, Cesson-Sevigne, France, "CAS 5093: Turbo-code codec," datasheet, available at http://www.comatlas.fr/cas5093rev41may95p1.pdf.

[17] Small World Communications, Adelaide, Australia, "MAP04T Very High Speed MAP Decoder," datasheet, available at http://www.sworld.com.au/pub/map04t.pdf.