Implementation and Performance of a Turbo/MAP Decoder

Steven S. Pietrobon, *Senior Member, IEEE*

***Abstract* — The implementation and performance of a turbo/MAP decoder is described. A serial block MAP decoder operating in the logarithm domain is used to obtain a very high performance turbo decoder. Programmable gate arrays and EPROM's allow the decoder to be programmed for almost any code from 4 to 512 states, rate 1/3 to rate 1/7 (higher rates are achieved with puncturing), and interleaver block sizes to 65,536 bits. Seven decoding stages were implemented in parallel. For rate 1/3 and 1/7 16 state codes with an interleaver size of 65,536 bits and operating at up to 356 kbit/s the codec achieved an E_b/N_0 of 0.32 and –0.30 dB, respectively for a BER of 10^{-5}. BER's down to 10^{-7} were also achieved for a small increase in E_b/N_0. An efficient implementation of a continuous MAP decoder is also presented, along with a synchronisation technique for turbo decoders.**

***Index Terms* — turbo coding, MAP decoding, synchronisation.**

I. Introduction

ERROR control coding aims to correct errors caused by noise and interference in a digital communications scheme. For power limited schemes, the energy per bit to single sided noise density ratio (E_b/N_0) is desired to be as low as possible. Good examples of this are satellite and space communications where fairly low bandwidth efficiencies (K, in bits transmitted per signalling interval or bit/sym) of 0.1 to 2 bit/sym are used.

A paper submitted to the *International Journal of Satellite Communications* 21 February 1997. Revised 4 December 1997, 2 April 1998. This work was performed under the sponsorship of the Australian Research Council and University of South Australia. The material in this paper was partially presented at the International Symposium on Information Theory, Whistler, BC, Canada, Sep. 1995 and International Symposium on Information Theory and its Applications, Victoria, BC, Canada, Sep. 1996.

The author was with the Satellite Communications Research Centre, University of South Australia, The Levels SA 5095, Australia. He is now with Small World Communications, 6 First Avenue, Payneham South SA 5070, Australia.

Coding adds redundancy using special codes. A decoder will then use this redundant information to correct as many errors as possible. Shannon [1] showed that for an additive white Gaussian noise channel, the smallest E_b/N_0 that can be obtained for reliable transmission is

$$E_b/N_0 \geq \frac{2^K - 1}{K}. \quad (1)$$

As K approaches 0 or the required bandwidth approaches infinity, the smallest value of E_b/N_0 is ln 2 or approximately –1.59 dB. A typical scheme such as uncoded quadrature phase shift keying (QPSK) with K = 2 bit/sym requires an E_b/N_0 of 9.6 dB for a bit error ratio (BER) of 10^{-5}. Figure 1 plots K versus E_b/N_0 showing the Shannon capacity curve from (1) and the capacity curve when QPSK modulation is used [2]. Also plotted are the performances of uncoded QPSK and some other coding schemes at a BER less than or equal to 10^{-5}. Note that the Voyager and Galileo schemes use binary phase shift keying (BPSK) which would result in their bandwidth efficiencies being reduced by half. However, since Gray mapped QPSK can achieve the same performance as BPSK with twice the bandwidth efficiency, we plot the Voyager and Galileo schemes using QPSK.

The industry standard code for satellite communications is a rate 1/2, 64 state, non–systematic convolutional code [3] with a soft–decision Viterbi decoder. This code can achieve an E_b/N_0 of 4.2 dB at a BER of 10^{-5}, giving a 5.4 dB coding gain. In practise, the use of three bit soft–decisions and other quantisation effects in the decoder result in a 0.2 dB performance loss. Another 0.2 dB may be lost due to the use of differential encoding to resolve 180° phase ambiguities in the QPSK signal set.

To obtain better performance, the standard code has been concatenated with a Reed–Solomon (RS) outer code. The most famous example is the (255,223) GF(2^8) RS code with depth eight interleaving used on the Voyager space probes [3,4]. This scheme can achieve an E_b/N_0 of 2.53 dB at a BER of 10^{-6} and K = 0.875 bit/sym. A more advanced scheme uses a rate 1/4, 8192 state, non–systematic convolutional inner code and a time varying, depth eight GF(2^8) RS outer code with redundancy profile (94,10,30,10,60,10,30,10) giving a (255,223.25) code on average [5]. Four stages of iterative decoding is used to obtain an E_b/N_0 of 0.58 dB at a BER of 10^{-7} and K = 0.438 bit/sym. This is the most powerful code currently in use and is 1.5 dB from Shannon capacity at K = 0.438 bit/sym.

In 1993, Berrou et al. published a paper describing a new coding scheme called “turbo–codes” [6]. A rate 1/2 code is described that achieved the amazing performance of E_b/N_0 = 0.7 dB at a BER of 10^{-5}. This is only 0.7 dB and 0.5 dB from Shannon and QPSK capacity, respectively, at K = 1 bit/sym. The encoder consists of two parallel concatenated systematic convolutional encoders

separated by a random interleaver. The code in [6] used two punctured 16 state codes and a 65,536 bit interleaver. This code is shown as TC1 in Figure 1.

Decoding is performed iteratively. Each systematic code is decoded using a soft–in soft–out (SISO) decoder. The output of the first decoder feeds into the second decoder to form one turbo decoder iteration. Eighteen iterations were performed in [6]. The SISO decoder used in [6] is a modification of the maximum a posteriori (MAP) decoding algorithm [7]. A similar iterative decoding technique is described by Gallager in his 1962 paper on low density parity check codes [8]. Each parity bit and its associated checked information bits is treated as a single (k+1,k) block code. A simple soft–output MAP decoding algorithm is used for each of the parity bits. A second SISO MAP decoder is then used to decode the information bit associated with the j parity check bits. The process then repeats.

The MAP algorithm finds the most likely information bit to have been transmitted in a coded sequence. This is unlike the Viterbi algorithm [9] which finds the most likely sequence to have been transmitted. When the decoded BER is small, there is a negligible error performance difference between the MAP and Viterbi algorithms. Since the MAP algorithm is considerably more complex than the Viterbi algorithm it has thus been largely ignored. However, at low E_b/N_0 and high BER's, MAP can outperform soft–output Viterbi by 0.5 dB or more. For turbo codes this is very important since the output BER's from the first stages of iterative decoding can be very high. Thus any improvement that can be obtained at these high BER's will directly result in performance increases.

A practical application of turbo/MAP decoders for satellite communications is given in [10]. Various turbo coding schemes were investigated to achieve 2 bit/sym bandwidth efficiency over a high speed mobile satellite link. Using a rate 1/2 turbo code with a 4000 bit block size, 8 iterations, and 16QAM modulation, an E_b/N_0 of 3.25 dB could be achieved over an ideal AWGN channel for a BER of 10^{-5}. This is only 1.5 dB from Shannon capacity at 2 bit/sym. This code is shown as (2,1,4;12,8,M)16QAM in Figure 1. The notation (n,k,ν;m,I,j) is used to describe a turbo code with n = number of coded bits, k = number of information bits (the code rate $R = k/n$), ν = memory of individual encoders (the number of states is equal to 2^{ν}), $m = \log_2 N_i$ where N_i is the interleaver size, I = number of decoder iterations, and j = M or V indicates MAP or SOVA decoders, respectively.

The MAP algorithm does not have to be used in a turbo decoder. The simpler soft–output Viterbi algorithm (SOVA) [11–14] can be also used. However, since the output of SOVA does not provide as good a statistic as MAP and makes more errors, degradations of about 0.8 dB can be expected in overall performance [15]. SOVA has been used in [16] to implement a turbo decoder

on a chip. Two punctured eight state codes were used to obtain a rate 1/2 turbo code which achieves an E_b/N_0 of 2.6 dB at a BER of 10^{-5}. This is almost the same performance of the more complicated Voyager code! The interleaver size is 1024 bits and 2.5 iterations (five SOVA decoders) were used. The code is shown as TC3 in Figure 1.

A single iteration on a chip was implemented in [17]. This chip consists of two 16 state SOVA decoders and a 2048 bit interleaver/deinterleaver. A performance of $E_b/N_0 = 1.7$ dB at a BER = 10^{-5} is achieved after five iterations for a rate 1/2 turbo code. This is TC2 in Figure 1.

An obstacle to implementing the code in [6] is its sheer complexity. The MAP algorithm described in [6] is very complex and is not very amenable to hardware implementation. By operating the algorithm in the logarithm domain [18,19], the complexity can be greatly reduced. In this paper we describe the implementation and performance of a turbo/MAP decoder which can implement the code in [5]. The MAP decoder was designed to be very flexible and can be programmed to operate from 4 to 512 states and from rate 1/2 to rate 1/4. Punctured codes can also be implemented. The decoding speed ranges from 17.7 kbit/s for 512 states to 624.4 kbit/s for 4 states. A 16 state code operates at 356.8 kbit/s.

We first give the derivation of the MAP, log–MAP, and sub–MAP decoding algorithms. This is followed by a description of an implementation of the log–MAP algorithm. We then describe the iterative turbo decoding algorithm and give a description of its implementation. Actual performance curves of the decoder are presented followed by some comments on continuous decoding and synchronisation.

II. The MAP, Log−MAP and Sub−MAP Algorithms

We present here a full derivation of the MAP decoding algorithm for systematic convolutional codes on an additive white Gaussian noise (AWGN) channel. The derivation is similar to that in [18] with the final presentation of the algorithm being slightly simpler than the traditional presentations.

A. The MAP Decoding Algorithm

The origin of the MAP algorithm belongs to Chang and Hancock [20] who developed it to minimise the symbol (or bit) error probability for an inter–symbol interference (ISI) channel. Simultaneously, Bahl et al. [21] and McAdam et al. [22] developed the algorithm for use on coded channels. The MAP algorithm that was presented in [6] for systematic convolutional codes is very complicated. A simplified version of this algorithm is given in [18]. However, Berrou et al. changed over to the more traditional presentation [15,23] of the algorithm in [24].

For an encoder with ν memory cells, we define the encoder state at time k, S_k, as a ν–tuple, depending only on the output of each delay element. The information bit at time k, d_k, is associated

with the transition from time k to time k+1 and will change the encoder state from S_k to S_{k+1}. Also suppose that the information bit sequence $\{d_k\}$ is made up of $N-\nu$ independent bits d_k, taking values 0 and 1 with *a priori probability* (APrP) ξ_k^0 and ξ_k^1, respectively ($\xi_k^0 + \xi_k^1 = 1$). We let the encoder initial state S_1 be equal to zero. The last ν information bits ($d_{N-\nu+1}$ to d_N) are set to values that will force the state to 0 at time N+1 (i.e., $S_{N+1} = 0$). This will slightly reduce the rate of the encoder.

We consider a rate 1/2 systematic feedback encoder whose outputs at time k are the uncoded data bit, d_k, and the coded bit, c_k. These outputs are modulated with a BPSK or QPSK modulator and sent through an AWGN channel. At the receiver end, we define the received sequence

$$R_1^N = (R_1, \dots, R_k, \dots, R_N), \tag{2}$$

where $R_k = (x_k, y_k)$ is the received symbol at time k; x_k and y_k are defined as

$$x_k = (2d_k - 1) + p_k, \tag{3}$$

$$y_k = (2c_k - 1) + q_k, \tag{4}$$

with p_k and q_k being two independent normally distributed random variables with variance σ^2. We define the *likelihood ratio*, λ_k associated with each decoded bit d_k as

$$\lambda_k = \frac{P_r(d_k = 0|R_1^N)}{P_r(d_k = 1|R_1^N)}, \tag{5}$$

where $P_r(d_k = i|R_1^N)$, i = 0,1 is the *a posteriori probability* (APoP) of the data bit d_k. The APoP of a decoded data bit d_k can be derived from the joint probability defined by

$$\lambda_k^{i,m} = P_r(d_k = i, S_k = m|R_1^N), \tag{6}$$

and thus the APoP of a decoded data bit d_k is equal to

$$P_r(d_k = i|R_1^N) = \sum_m \lambda_k^{i,m}, \tag{7}$$

where i = 0,1 and the summation is over all 2^ν encoder states. From (5) and (7), the λ_k associated with a decoded bit d_k can be written as

$$\lambda_k = \frac{\sum_m \lambda_k^{0,m}}{\sum_m \lambda_k^{1,m}}. \tag{8}$$

The decoder can make a decision by comparing λ_k to a threshold equal to one

$$\hat{d}_k = \begin{cases} 0 \ ; \ \lambda_k \geq 1, \\ 1 \ ; \ \lambda_k < 1. \end{cases} \tag{9}$$

Using Bayes' rule, the joint probability from (6) can be rewritten as follows

$$\lambda_k^{i,m} = P_r(d_k = i, S_k = m, R_1^N)/P_r(R_1^N),$$

$$= P_r(R_1^{k-1}|d_k = i, S_k = m, R_k^N)P_r(R_{k+1}^N|d_k = i, S_k = m, R_k)$$
$$\times P_r(d_k = i, S_k = m, R_k)/P_r(R_1^N). \tag{10}$$

We have

$$P_r(R_1^{k-1}|d_k = i, S_k = m, R_k^N) = P_r(R_1^{k-1}|S_k = m) = \alpha_k^m, \tag{11}$$

since the assumption that $S_k = m$ implies that events before time k are not influenced by observations after time k. We define α_k^m as the forward state metric at time k and state m. Similarly, we have

$$P_r(R_{k+1}^N|d_k = i, S_k = m, R_k) = P_r(R_{k+1}^N|S_{k+1} = f(i,m)) = \beta_{k+1}^{f(i,m)}, \tag{12}$$

where $f(i,m)$ is the next state given an input i and state m. We define β_k^m as the reverse state metric at time k and state m. We define the branch metric as

$$\delta_k^{i,m} = P_r(d_k = i, S_k = m, R_k). \tag{13}$$

Substituting (11) to (13) in (10) we obtain

$$\lambda_k^{i,m} = \alpha_k^m \delta_k^{i,m} \beta_{k+1}^{f(i,m)} / P_r(R_1^N). \tag{14}$$

This result can be used to evaluate (8) as

$$\lambda_k = \frac{\sum_m \alpha_k^m \delta_k^{0,m} \beta_{k+1}^{f(0,m)}}{\sum_m \alpha_k^m \delta_k^{1,m} \beta_{k+1}^{f(1,m)}}, \tag{15}$$

where the summations are over all 2^ν states. The usual expression for (15) involves a double summation in both the numerator and denominator.

We want to show that (11) can be recursively calculated. We can express (11) as

$$\begin{aligned}
\alpha_k^m &= P_r(R_1^{k-1}|S_k = m) \\
&= \sum_{m'} \sum_{j=0}^{1} P_r(d_{k-1} = j, S_{k-1} = m', R_1^{k-1}|S_k = m) \\
&= \sum_{m'} \sum_{j=0}^{1} P_r(R_1^{k-2}|S_k = m, d_{k-1} = j, S_{k-1} = m', R_{k-1}) P_r(d_{k-1} = j, S_{k-1} = m', R_{k-1}|S_k = m) \\
&= \sum_{j=0}^{1} P_r(R_1^{k-2}|S_{k-1} = b(j,m)) P_r(d_{k-1} = j, S_{k-1} = b(j,m), R_{k-1}) \\
&= \sum_{j=0}^{1} \alpha_{k-1}^{b(j,m)} \delta_{k-1}^{j,b(j,m)},
\end{aligned} \tag{16}$$

where the first summation is from $m' = 0$ to $2^\nu - 1$ and $b(j,m)$ is the state going backwards in time from state m on the previous branch corresponding to input j. In a similar way we can recursively calculate the probability β_k^m from the probability β_{k+1}^m. Note that this is possible only after the whole block of data is received. Relation (12) becomes

$$\begin{aligned}\beta_k^m &= Pr(R_k^N|S_k = m)\\ &= \sum_{m'}\sum_{j=0}^{1} Pr(d_k = j, S_{k+1} = m', R_k^N|S_k = m)\\ &= \sum_{m'}\sum_{j=0}^{1} Pr(R_{k+1}^N|S_k = m, d_k = j, S_{k+1} = m', R_k) Pr(d_k = j, S_{k+1} = m', R_k|S_k = m)\\ &= \sum_{j=0}^{1} Pr(R_{k+1}^N|S_{k+1} = f(j,m)) Pr(d_k = j, S_k = m, R_k)\\ &= \sum_{j=0}^{1} \delta_k^{j,m}\beta_{k+1}^{f(j,m)}. \end{aligned} \tag{17}$$

Figure 2 gives a graphical illustration of the calculation of α_k^m and β_k^m. It is very similar to the architecture of the Viterbi algorithm. Where we add the branch metric to the state metric in the Viterbi algorithm, we multiply in the MAP algorithm. Where we find the minimum of the path metrics in the Viterbi algorithm, we add in the MAP algorithm. Thus, the add–compare–select (ACS) operation in the Viterbi algorithm becomes the multiply–add (MA) operation in the MAP algorithm.

The MAP algorithm works by first calculating the α_k^m's in the forward direction and storing the results. The β_k^m's are then calculated in the reverse direction. An important observation is that the $\delta_k^{j,m}\beta_{k+1}^{f(j,m)}$ term in (17) is also used in the calculation of λ_k in (15). Thus, while the β_k^m's are being calculated, λ_k should be calculated at the same time, reusing the $\delta_k^{j,m}\beta_{k+1}^{f(j,m)}$ terms to minimise the number of computations. If the block starts in state zero then we initialise $\alpha_1^0 = 1$ and $\alpha_1^m = 0$ for $m \neq 0$. A similar initialisation is performed for β_{N+1}^m if the block ends in state zero. If the block ends in an unknown state (which occurs when there is no termination) then $\beta_{N+1}^m = 1$ for all m.

The branch metric $\delta_k^{j,m}$ can be determined from the transition probability of the discrete memoryless channel and the APrP. From (13) and using Bayes' rule we have

$$\begin{aligned}\delta_k^{i,m} &= Pr(d_k = i, S_k = m, R_k)\\ &= Pr(R_k|d_k = i, S_k = m) Pr(S_k = m|d_k = i) Pr(d_k = i)\\ &= Pr(x_k|d_k = i, S_k = m) Pr(y_k|d_k = i, S_k = m)\xi_k^i/2^\nu, \end{aligned} \tag{18}$$

since p_k and q_k are independent, the current state is independent of the current input and can be in any of the 2^ν states, and $\xi_k^i = Pr(d_k = i)$ by definition. For an AWGN channel with zero mean and variance σ^2 (18) becomes

$$\delta_k^{i,m} = \frac{\xi_k^i}{2^\nu\sqrt{2\pi}\,\sigma}\exp\left(-\frac{1}{2\sigma^2}(x_k - (2i-1))^2\right)dx_k \frac{1}{\sqrt{2\pi}\,\sigma}\exp\left(-\frac{1}{2\sigma^2}(y_k - (2c^{i,m}-1))^2\right)dy_k$$

$$= \varkappa_k \zeta_k^i \exp\left(L_c(x_k i + y_k c^{i,m})\right), \tag{19}$$

where $\varkappa_k$ is a constant, dx_k and dy_k are the differentials of x_k and y_k, $L_c = 2/\sigma^2$, and $c^{i,m}$ is the coded bit given $d_k = i$ and $S_k = m$. Since the constant $\varkappa_k$ in (19) doesn't affect λ_k in (15) we can normally ignore $\varkappa_k$. In practise though, when calculating the forward and reverse state metrics, we let $\varkappa_k$ be equal to the inverse of the largest previous state metric. This normalises the new state metrics and ensures that the state metrics do not under or overflow.

If we substitute (19) into (15) we obtain

$$\begin{aligned}\lambda_k &= \frac{\zeta_k^0}{\zeta_k^1}\exp(-L_c x_k)\frac{\sum_m \alpha_k^m \exp(L_c y_k c^{0,m})\beta_{k+1}^{f(0,m)}}{\sum_m \alpha_k^m \exp(L_c y_k c^{1,m})\beta_{k+1}^{f(1,m)}}, \\ &= \zeta_k \exp(-L_c x_k)\zeta_k', \end{aligned} \tag{20}$$

where $\zeta_k = \zeta_k^0/\zeta_k^1$ is the input APrP ratio and ζ_k' is the output *extrinsic information*. One can think of ζ_k' as a correction term that changes the input information so as to minimise the probability of decoding error. This extrinsic information is very important in turbo decoding as it allows the corrections terms to be passed from one decoder to the next.

B. The log–MAP Decoding Algorithm

To minimise the decoding complexity, we would like to eliminate the multiply operations required by the MAP algorithm. This can be achieved by taking the logarithm (or negative logarithm) of the algorithm. Again, this technique was first used for the ISI channel [26,27]. It was later applied to the coding channel in [18,19,25,28,29].

Taking the negative logarithm, the multiplications in the algorithm are converted to additions. Adders are much easier to implement than multipliers. However, the additions are converted to the E operand defined below

$$\begin{aligned} a \ \mathrm{E} \ b &\equiv -\log_\varepsilon(\varepsilon^{-a} + \varepsilon^{-b}) \\ &= \min(a,b) - \log_\varepsilon\left(1 + \varepsilon^{-|a-b|}\right). \end{aligned} \tag{21}$$

The functions min*(a,b)* and $|a–b|$ can be easily determined using subtraction and multiplexer circuits. However, the function

$$\begin{aligned} f(z) &= \log_\varepsilon(1 + \varepsilon^{-z}) \\ &= c\ln(1 + e^{-z/c}) \end{aligned} \tag{22}$$

where $c = 1/\ln\varepsilon = \log_\varepsilon e$ would appear to be too complicated to be implemented. Figure 3 plots $f(z)$ against z for $c = 1$. We can see that $f(z)$ quickly decays to zero and has a maximum value of

$c \ln 2 \approx 0.693c$ (for $z \geq 0$). Thus, $f(z)$ can be easily implemented in a small lookup table. A range of lookup tables can be implemented to cover various values of c.

If we let

$$L_k = -\log_\varepsilon \lambda_k \tag{23a}$$

$$A_k^m = -\log_\varepsilon \alpha_k^m \tag{23b}$$

$$B_k^m = -\log_\varepsilon \beta_k^m \tag{23c}$$

$$D_k^{i,m} = -\log_\varepsilon \delta_k^{i,m} \tag{23d}$$

the MAP algorithm becomes

$$L_k = \mathop{\mathrm{E}}_{m=0}^{2^\nu - 1} A_k^m + D_k^{0,m} + B_{k+1}^{f(0,m)} - \mathop{\mathrm{E}}_{m=0}^{2^\nu - 1} A_k^m + D_k^{1,m} + B_{k+1}^{f(1,m)} \tag{24}$$

$$A_k^m = \mathop{\mathrm{E}}_{j=0}^{1} A_{k-1}^{b(j,m)} + D_{k-1}^{j,b(j,m)} \tag{25}$$

$$B_k^m = \mathop{\mathrm{E}}_{j=0}^{1} D_k^{j,m} + B_{k+1}^{f(j,m)} \tag{26}$$

where $\mathrm{E}_{j=0}^{l-1}\ a^j = a^0\ \mathrm{E}\ a^1\ \mathrm{E} \cdots \mathrm{E}\ a^{l-1}$. The branch metrics are

$$\begin{aligned} D_k^{i,m} &= -\log_\varepsilon \varkappa_k - \log_\varepsilon \zeta_k^i - A(x_k i + y_k c^{i,m}) \\ &= -\log_\varepsilon \varkappa_k - \log_\varepsilon \zeta_k^0 - i \log_\varepsilon(\zeta_k^1/\zeta_k^0) - A(x_k i + y_k c^{i,m}) \\ &= -K_k - (z_k + Ax_k)i - Ay_k c^{i,m} \end{aligned} \tag{27}$$

where K_k is a constant, $A = (2/\sigma^2)\log_\varepsilon e = L_c c$, and $z_k = -\log_\varepsilon \zeta_k$ is the log–APrP. To perform renormalisation we let K_k be equal to the smallest previous state metric. We can think of A as the no–noise amplitude of our demodulated and quantised signal, e.g., +1 could be equivalent to $A = 7 = 111_2$. Note that we can arbitrarily vary A and L_c to determine c and thus the values in a lookup table for (22). Alternatively, given L_c and a lookup table for a value of c, we can vary A. We can re–express (20) as

$$L_k = z_k + Ax_k + z_k' \tag{28}$$

where $z_k' = -\log_\varepsilon \zeta_k'$ is the extrinsic information from the log–MAP decoder.

C. The sub–MAP Decoding Algorithm

If we let $f(z) = 0$, then (24) to (26) becomes

$$L_k = \min_m \left(A_k^m + D_k^{0,m} + B_{k+1}^{f(0,m)}\right) - \min_m \left(A_k^m + D_k^{1,m} + B_{k+1}^{f(1,m)}\right) \tag{29}$$

$$A_k^m = \min\left(A_{k-1}^{b(0,m)} + D_{k-1}^{0,b(0,m)},\ A_{k-1}^{b(1,m)} + D_{k-1}^{1,b(1,m)}\right) \tag{30}$$

$$B_k^m = \min\left(D_k^{0,m} + B_{k+1}^{f(0,m)},\ D_k^{1,m} + B_{k+1}^{f(1,m)}\right). \tag{31}$$

This sub–optimal algorithm (which we shall call sub–MAP) has the advantage that it is independent of σ^2. Again, this algorithm was first derived for the ISI channel [26,27] and then later applied to the coding channel [19,30]. It has the same sub–optimal hard decision performance as the Viterbi algorithm [19]. Note that the calculation of the forward state metrics is exactly the same as the state metric (SM) calculation for the Viterbi algorithm. However, unlike the Viterbi algorithm, the SM's also need to be calculated in the reverse direction and the likelihood ratio determined.

III. Implementing the Log−Map Algorithm

As can be seen from (24) to (27), there are four major sections in a log–MAP decoder. These are the forward state metric calculator (FSMC), reverse state metric calculator (RSMC), log likelihood ratio calculator (LLRC), and the branch metric calculator (BMC). To minimise the decoder gate count, it was decided to have a serial implementation. That is, each of the SM's are computed one at a time. This is similar to previous serial Viterbi and trellis decoders constructed by the author [31,32]. Since there are 2^ν states, this implied that it would take at least 2^ν decoder clock (CLK) cycles to decode each bit. Thus the CLK frequency had to be at least 2^ν times greater than the data clock (DCLK) frequency.

To avoid problems with high speed clocks, it was decided to limit the CLK speed to 10 MHz. Also, the Xilinx XC3100A [33] series programmable logic was chosen due to its low cost and relatively high speed.

A. Branch Metric Calculator

From [34], the BM's are calculated as follows

$$D_k^{i,m} = |z_k + Ax_k|(i \oplus \mathrm{u}(z_k + Ax_k)) + |Ay_k|(c^{i,m} \oplus \mathrm{u}(Ay_k)) - K_k \tag{32}$$

where $i \oplus j$ is the modulo–2 sum of i and j and u(w) is the unit step function. As before, K_k is equal to the minimum previous state metric. In two's complement notation, u(w) corresponds to the logical inverse of the most significant or sign bit of w. It can be shown that

$$|w|(j \oplus \mathrm{u}(w)) = -wj + (w + |w|)/2. \tag{33}$$

The $-wj$ term in (33) directly corresponds to the terms in (27) with $w = z_k + Ax_k$ and $j = i$ or $w = Ay_k$ and $j = c^{i,m}$. The additional terms are constants dependent on k only and can be absorbed in K_k. We can see that when the BM in (32) is added to its state metric, the resulting value will always be greater than or equal to zero.

An important consideration is the value of the signal amplitude A. The optimum value of A can vary depending on the number of quantisation bits q and the noise variance σ^2 [36]. Figure 4

illustrates a model of the demodulator for the received signal x_k (as defined in (3)). A model of the branch metric calculator is also shown (we have assumed that $z_k = 0$ in this case). We assume that x_k is multiplied by some unknown fixed positive voltage V. The demodulator has an automatic gain control (AGC) circuit that effectively normalises the input signal to its mean absolute value, i.e.,

$$\begin{aligned} \mathrm{E}\Big[|Vx_k|\Big] &= V\mathrm{E}\Big[|2d_k - 1 + p_k|\Big] \\ &= V\left(\sigma\sqrt{\frac{2}{\pi}}\exp\left(-\frac{1}{2\sigma^2}\right) + 1 - 2Q\left(\frac{1}{\sigma}\right)\right) \\ &= V\mathrm{mag}(\sigma). \end{aligned} \tag{34}$$

Note that for high SNR (and low σ) that $\mathrm{mag}(\sigma) \approx 1$. However, for low SNR (and high σ) we have $\mathrm{mag}(\sigma) \approx \sigma\sqrt{2/\pi} \approx 0.798\sigma$. This is very important when trying to estimate the noise variance. A variance estimator that assumes $\mathrm{mag}(\sigma) = 1$ will work correctly only for high SNR.

For turbo codes where a low SNR is expected, a more complicated method is required to determine V and σ. We can see from (34) that we have two unknowns and one equation. To solve for V and σ we need another equation which can be obtained by estimating the square of the received signal,

$$\mathrm{E}\Big[(Vx_k)^2\Big] = V^2(1 + \sigma^2). \tag{35}$$

Figure 5 plots $\mathrm{mag}(\sigma)$ and $\mathrm{rms}(\sigma) = \sqrt{1 + \sigma^2}$. We can see that for low SNR, $\mathrm{rms}(\sigma) \approx \sigma$. Due to the complexity of (34), there does not seem to be a simple direct solution of the two equations. The estimation of (34) and (35) can performed digitally. By quantising (34) and (35) into say eight bits each, a $64\mathrm{K} \times 8$ lookup table can be used to output precomputed σ values quantised to eight bits each. Alternatively, the ratio of the square of (34) with (35) will give a single equation as a function of σ [35]. A smaller 256×8 lookup table can then be used to output σ.

The value of A compared to the dynamic range of an analog to digital (A/D) converter can greatly effect the decoded BER [36] (especially when the number of quantisation levels is small). We shall assume that there are $2^q - 1$ quantisation regions with a central "dead–zone". That is, a quantised "0" ranges from –0.5 to 0.5. The largest quantised value is $2^{q-1} - 1$ and ranges from $2^{q-1} - 1.5$ to infinity. As shown in our demodulator model in Figure 4, we shall assume that the demodulator scales the input by C before A/D conversion. In [36] computer simulations of the Viterbi algorithm showed that C should be less than one. Also, as the SNR is decreased, the value of A relative to the maximum quantised output should decrease as well. From Figure 4 we have that the "optimum" value of A is

$$A = \frac{C(2^{q-1} - 1)}{\mathrm{mag}(\sigma)}. \tag{36}$$

Analysing the simulations in [36], we found that $C = 0.65$ should give near optimum performance. However, for low σ it may be necessary to reduce C (and thus A) to keep the lookup tables for the E operand at a reasonable size.

For our BMC, it was decided to have $q = 6$ bit quantisation as a compromise between having good performance and decoder complexity. Each MAP decoder could be programmed for either rate 1/2, 1/3, or 1/4 operation. For rate 1/4 mode and $z_k = 0$, this implied that the maximum BM value is $n(2^{q-1} - 1) = 124$ where n is the number of coded bits. In turbo decoding mode, z_k can range from –128 to +127 which can greatly increase the maximum BM. Due to the limitation of the number of bits to represent each SM, the maximum BM was therefore limited to 127.

One BM is calculated each CLK cycle with a two CLK cycle pipeline delay (one cycle to determine the symbol and another cycle to perform the calculation). The BM is then passed onto the FSMC and an SRAM for storage. The BM's stored in the SRAM are then read out in reverse order for the RSMC. Since 2^{ν} BM's are calculated for N DCLK cycles, the total storage space required is $N2^{\nu}$. Although there are only 2^{n} BM's we chose to simply store and then later retrieve the previously calculated BM's for the RSMC.

Ideally, for the $N = 2^{16}$ turbo code in [6], the required memory storage is one megabyte (MB). This was too large and too expensive to implement, and so it was decided to limit the storage space to 64K. For $\nu = 4$ this implies that $N = 2^{12} = 4096$. A simplistic storage technique is to write the new data into one 64K RAM and read the old data from another 64K RAM. This requires a total of 128K of RAM. We can halve the amount of RAM by using the circuit shown in Figure 6.

When CLK is high, the previously stored BM is read out from the RAM and then latched on the falling edge of CLK. Simultaneously, a new BM is written into the RAM using the same address when CLK goes low. Thus, in one clock cycle we perform a read, followed by a write. Since the RAM address is inverted every N DCLK cycles, the BM's are read out in reverse order. A control signal to the $\overline{\text{CE}}$ input of the RAM is used to enable the storage of the new SM's at the correct time. In our design, two 35 ns 64K × 4 separate I/O SRAM's were used.

Our design could be programmed from 4 to 512 states, with a corresponding change in N from 16,384 to 128. However, to limit the decoding delay and storage requirements for the turbo decoder, the maximum block size for 4 and 8 states was reduced to 4096.

B. State Metric Calculators

The architecture of the FSMC and RSMC are very similar. A 35 ns 1K × 8 dual–port RAM is used to retrieve old SM's and store the new SM's. With eight bit precision, the SM's can range from

0 to 255. A further increase in precision would have greatly increased the complexity of the decoder. Thus, it was decided that the SM's would be represented by eight bits.

At the end of the previous block, the initial SM's are stored into one side of the RAM. For the FSM's the SM for state 0 is set to zero and the other states are set to 255 (the closest value to infinity). This corresponds to the sequence starting in state zero. For the RSM's if the final state is unknown all the initial SM's are set to 0, otherwise they are initialised in the same way as the FSM's.

To determine the read and write addresses for the SM's we need to examine the implementation of a rate 1/n systematic encoder. Berrou et al. [6] showed that a rate 1/2 systematic encoder can be implemented using a shift register as shown in Figure 7. We have that $S_k = \left(s_k^0, s_k^1, s_k^2, \dots, s_k^{\nu-1}\right)$ corresponds to the current encoder state, $G_i = \left(g_i^0, g_i^1, g_i^2, \dots, g_i^{\nu}\right)$ corresponds to the encoder code, and $g_i^j, s_k^j \in \{0, 1\}$, for $0 \le i \le n-1$. We assume that $g_i^0 = g_i^{\nu} = 1$ for $0 \le i \le n-1$ since this ensures that the free Hamming distance leaving and entering a state is at least $2n$.

For the forward SM's, we need to determine $b(d_{k-1}, S_k)$ from (25). If we let

$$b(d_{k-1}, S_k) = S_{k-1} = \left(s_k^1, s_k^2, \dots, s_k^{\nu-1}, s_k^0\right) \tag{37}$$

then

$$S_k = \left(d_{k-1} \oplus s_k^0 \oplus s_k^*, s_k^1, s_k^2, \dots, s_k^{\nu-1}\right) \tag{38}$$

where

$$s_k^* = \sum_{j=1}^{\nu-1} g_0^j s_k^j \bmod 2. \tag{39}$$

We can see that by reading two SM's we can generate two new SM's. Figure 8 gives a partial trellis for a $\nu = 4$ code. Thus, in the next data clock (DCLK) cycle the previously stored SM's are read out one at a time using the following forward read address (ignoring the subscript k)

$$R_f = \left(s^1, s^2, \dots, s^{\nu-1}, s^0\right). \tag{40}$$

A serial to parallel operation is performed and the two SM's are stored in a register for two CLK cycles. In the first CLK cycle, BM0 is added to the first SM and BM1 is added to the second SM to form the first new SM. In the next CLK cycle, the BM's are reversed to form the second new SM. After a delay from reading and calculating the new SM's (equal to five CLK cycles) the new SM's are written into the dual–port RAM with the following address (ignoring the subscript k)

$$W_f = \left(s^0 \oplus s^*, s^1, s^2, \dots, s^{\nu-1}\right). \tag{41}$$

From (26), we need to determine $f(d_k, S_k)$ for the reverse direction. If we let

$$f(d_k, S_k) = S_{k+1} = \left(d_k \oplus s_k^0 \oplus s_k^*, s_k^1, s_k^2, \dots, s_k^{\nu-1}\right) \tag{42}$$

then

$$S_k = \left(s_k^1, s_k^2, \ldots, s_k^{\nu-1}, s_k^0\right). \tag{43}$$

Thus, in a similar way to the FSMC the read and write addresses for the RSMC are

$$R_r = \left(s^0 \oplus s^*, s^1, s^2, \ldots, s^{\nu-1}\right). \tag{44}$$

$$W_r = \left(s^1, s^2, \ldots, s^{\nu-1}, s^0\right). \tag{45}$$

Since there is a delay in calculating the new SM's, we can't use the read/$\overline{\text{write}}$ technique as used for storing the BM's. With a $1\text{K} \times 8$ DP–RAM this implied the maximum number of states is 512 (half the memory size). The minimum number of states is 4 due to the restriction that $g_i^0 = g_i^\nu = 1$.

Since the forward SM's are used in the LLRC they also need to be stored and read out in reverse order. A circuit very similar to the BM storage circuit is used to perform this task. The old forward SM's that are read from the DP–RAM are the values that are stored in two $64\text{K} \times 4$ SRAM's.

An important part of the SM calculator is the implementation of the adders and the E operand. Figure 9 illustrates how the BM's are added to the SM's (similar to that in [37]). We see that the previous minimum SM is subtracted from the BM's before being added to the SM's. The output of the BM subtraction circuit has a two's complement output (we limit the output so that the range is from –128 to 127). The BM's are then added to the SM's (just as in the Viterbi algorithm). If the BM is positive, the SM limiting circuit is enabled. However, if the BM is negative, the limiting circuit is disabled to allow normal addition to occur. Since the BM can never be more negative than the smallest SM, the resulting SM will always be positive.

Figure 10 illustrates the E operand circuit. As shown in (21) we need to find the minimum of the two path metrics (PM) as well as the absolute difference. The carry–out of a subtraction circuit (with carry–in set to 0) is used to select the smallest path metric. This carry out is also used to invert the output of another subtraction circuit to give the absolute difference. When PM^0 is greater than PM^1 the output of the subtractor will give the correct positive output when the carry–in is equal to 1. However, if PM^0 is less than or equal to PM^1 then the output will be negative. Normally, we would have to invert the output and then add 1. However, by setting the carry–in to 0, we avoid the additional adder circuit. This is why the carry–out of the first subtractor goes into the carry–in of the second subtractor.

A circuit that uses a comparator, two multiplexers, and a subtractor can also be used to implement the minimum and absolute difference functions [29]. In XC3100A logic this would require at least 20 configurable logic blocks (CLB) to implement using eight bit arithmetic,

compared to 16 CLB's for our design (since the XNOR's can be absorbed into the second comparator).

In order to keep the SM's positive, we add $c\ln(2)$ to (21). Thus, if the absolute difference is equal to zero, we add zero to the minimum, otherwise we add some small value. As determined previously, the maximum value of $f(z)$ is $c\ln(2)$. For a rate 1/7 code, E_b/N_0 = –0.5 dB, and six bit quantisation we have $\sigma^2 = 3.93$ and using (36) A = 11.3 (which we round to 11). Thus $c = \sigma^2 A/2 = 21.6$ and the maximum value of $f(z)$ is 14.99 (15 after rounding). Thus only four bits are required to represent $f(z)$.

The smallest non–zero value of $f(z)$ that results in 0 after quantisation is 0.5. Solving for $f(z)$ = 0.5 we have

$$z = -c\ln(\exp(0.5/c) - 1). \tag{46}$$

For the above conditions we have z = 81.2. Since $f(z)$ monotonically decreases with z, all values of $f(z)$ for $z \geq 81.2$ will be less than 0.5 and so will be quantised to 0. Therefore, by limiting values above 81 (the quantised value of 81.2) from the absolute difference circuit to 81, we only require an 81×4 lookup table to implement $f(z)$. In our design, we limited the maximum address space to 63 in order to reduce the design complexity. This implied from (46) that the largest value of c is 17.835 (giving z = 63.5). Thus, if the optimum value of A causes c to be more than 17.835, we reduce A to give us the maximum c. For the above example this would be A = 9.08 (or 9 after rounding).

Figure 10 shows how the absolute difference output is limited to six bits and used to address a 64×4 lookup table to determine $f(z) + c\ln(2)$. The table look–up output is then added to the minimum circuit output. The adder output is limited to eight bits as shown. Just as in a Viterbi decoder, limiting the outputs of the adders will cause some degradation in performance. However, since it is the larger and thus less likely path metrics that are limited, this degradation will be very small.

C. Log–Likelihood Ratio Calculator

For the computation of λ_k^i (EM^i) we add the reverse path metric for bit i (RPM^i) to the corresponding forward SM that is read from the SRAM. This is shown in Figure 11. The summation gives a nine bit result which is not limited. A most significant sign bit is also added due to the effect of $f(z)$ (in this case we cannot add $c\ln(2)$ because we are E summing more than one term). The register is initialised to its maximum value (+511) to start the E summation (or "Eccumulation"). We could have used a multiplexer to initialise the EM^i register to the first summation. However, to reduce complexity a simple limiting circuit was used. Since on the first E summation +511 is

usually much greater the first metric, the EM^i register would be correctly initialised most of the time.

We then find the minimum and absolute difference between the current register output and the current metric (the summation of RPM^i and SM). We then perform the standard look–up operation, in this case subtracting the look–up result from the minimum. To decrease the delay between the register output to its input, we also register the outputs of the abs–min circuit. This forced us to insert a multiplexer after the first adder (the other input was the EM^i output) to allow the correct EM^i to be calculated due to pipeline delay.

After the EM^i's have been finally calculated (which are done in parallel), we subtract EM^1 from EM^0 and limit the output to +127 or –128 if necessary to give an eight bit result. An additional three clock cycles are required to perform the final calculation. Since the LLR are calculated with the reverse SM's, the LLR's are produced in reverse order. Thus, for a normal MAP decoder the output needs to be reversed in time in blocks of *N*.

D. Decoder Performance

Since up to two clock cycles are required for the decoder to start, two for the BMC, five for the SMC's, and three for the LLRC, a total of $2^\nu + 12$ clock cycles are required. All the logic was implemented in XC3100A–5 gate arrays which allowed a clock speed of f_c = 10 MHz. The SRAM's were 35 ns in speed. Also, ν bits were used to terminate the trellis which slightly reduces the speed of the decoder. The decoder speed is given by

$$f_d = \begin{cases} \dfrac{(2^{12} - \nu) f_c}{2^{12}(2^\nu + 12)} & : 2 \le \nu \le 4 \\ \dfrac{(2^{16-\nu} - \nu) f_c}{2^{16-\nu}(2^\nu + 12)} & : 5 \le \nu \le 9 \end{cases}. \tag{47}$$

The slowest decoder speed is for ν = 9 at 17.7 kbit/s and the fastest decoder speed is for ν = 2 at 624.7 kbit/s. For ν = 4 the speed is 356.8 kbit/s.

The SMC's and LLRC were implemented in one XC3190A–5 (7500 gate equivalent). The BMC was implemented in one XC3142A–5 (3700 gate equivalent) while two XC3130A–5's (2700 gate equivalent) were used to implement the control logic and address generation (as well as some additional functions for the turbo decoder). The encoder was implemented using an XC3142A–5. Both the encoder and decoder can be programmed with any code with $g_i^0 = g_i^\nu = 1$ through a series of DIP switches.

To test the decoder performance, AWGN noise was generated on a PC using a C++ program. The parallel port on the back of the PC was used to transmit eight bit quantised noise samples to

the decoder. An adder circuit in the encoder Xilinx chip adds the noise to the encoded signal. The adder circuit produces a 6 bit quantised result with a dead zone and 63 quantisation regions. Various noise generators were used, starting with one that had a period of 8.4×10^6 for the MAP decoder tests, 2.1×10^9 for the rate 1/3 turbo decoder tests, and 2.3×10^{18} for the rate 1/7 turbo decoder tests. This last decoder used the "standard" Lehmer uniform random number generator with multiplier 16807 and modulus 2^{31}–1 [38] together with the Box–Muller algorithm from [39, pp. 216–217] to generate 2^{23} eight bit quantised random numbers which are stored in RAM (after inputting A and σ). The dual 32 bit seed uniform random number generator from [40] was then used to randomly select the numbers from RAM and send them to the encoder.

Figure 12 gives the performance of a rate 1/4, 512 state systematic code with code polynomials $g_0 = 1753$, $g_1 = 1547$, $g_2 = 1345$, and $g_3 = 1151$ taken from [41]. The performance of the hardware log–MAP decoder is plotted, along with the performance of the hardware sub–MAP decoder and a software block Viterbi decoder. The signal amplitude A was fixed to 7 to avoid limiting the SM's too much. This is less than the "optimum" values for six bit quantisation, but with only eight bit state metrics, SM limiting becomes more of a factor. At low BER, we can see that the implementation loss is only 0.05 dB. Note that at low BER the ideal performance of the Viterbi and MAP algorithms are almost identical which allows us to make a comparison. At high BER, the MAP decoder gains about 0.5 dB over ideal Viterbi.

Also plotted on the graph is the sub–MAP decoder performance. This is where the E operand is simplified to the min function. This was done by setting the lookup tables in the MAP decoder to zero. The signal amplitude was also set to A = 7. As can be seen, at low BER the sub–MAP performance is almost identical to the MAP decoder. However, the sub–MAP decoder always performed worse than ideal Viterbi, losing about 0.3 dB at high BER.

Figure 13 shows the performance for a systematic rate 1/2, 16 state code with polynomials $g_0 = 37$ and $g_1 = 21$ from [6]. Since the rate loss from the tail bits is very small, we could compare our decoder results with a software MAP decoder from [42]. The hardware MAP decoder closely follows the computer simulation at high BER, losing 0.08 dB at low BER. The sub–MAP decoder loses about 0.5 dB at high BER and closely follows the MAP decoder at low BER.

IV. Turbo Coding and Decoding

The basic turbo encoder consists of two or more parallel concatenated systematic convolutional encoders separated by an interleaver of size N_i. Figure 14 shows the general implementation of a rate 1/3 turbo encoder from two constituent rate 1/2 encoders. The two coded outputs can be punctured in order to obtain higher rates. The INT block is the interleaver and the

DEL block is a delay circuit with a delay equal to the MAP decoder delay. The DEL circuit is required since the decoder for the interleaved data has to wait for the first decoder to output its data. Thus, the received data has to be appropriately delayed. Instead of the decoder delaying the received interleaved data, this delay is performed within the encoder, simplifying the decoder operation.

Figure 15 illustrates the basic decoding block in an iterative turbo decoder. In the first iteration we do not need to add the APrP for d_k to Ax_k. Since both $d_k = 0$ or 1 are equally likely, we have that $z_k^2 = 0$ (a superscript of 2 is used here as explained later) is added to Ax_k. We then decode the symbols from the first encoder. The output from the first MAP decoder is $\lambda_j^1 = Ax_j + z_j^1$ where the superscripts indicates which MAP decoder was used, and $j = k - D_d$ where D_d is the delay of the MAP decoder. This data is then interleaved to match the interleaved symbols from the second encoder.

The $Ax_j + z_j^1$ from the first MAP decoder is then fed into the second MAP decoder. In this case we have let the extrinsic information from the first MAP decoder become the APrP for the second MAP decoder. One can think of the first MAP decoder improving the SNR for Ax_j which effectively results in a lower BER for $\hat{d}_k$. The purpose of the interleaver is to randomise the "burst" errors that are characteristic of MAP and Viterbi decoders. The larger the interleaver the more randomised are the bursts of errors.

With an improved Ax_j, the second MAP decoder is able to correct even more errors. Its output is

$$\lambda_i^2 = z_i^1 + Ax_i + z_i^2 \tag{48}$$

where z_i^2 is the extrinsic information from the second MAP decoder and the subscript i is used to indicate the interleaved and delayed time index. We subtract $z_i^1 + Ax_i$ from λ_i^2 to obtain z_i^2 which is then deinterleaved and passed onto another iteration as $z_{k-\Delta}^2$ (where Δ is the total delay of the iteration). Just like in the second MAP decoder, this extrinsic information becomes the APrP for the first MAP decoder in the next iteration. The deinterleaver serves to randomise the burst errors from the second MAP decoder. Note that if we included z_i^1 with z_i^2, the deinterleaver would produce a delayed z_j^1 with "bursty" errors. Thus, feeding these bursty errors into the first MAP decoder will result in very poor performance from the decoder.

In the next iteration the first MAP decoder output is

$$\lambda_j^1 = z_j^2 + Ax_j + z_j^1. \tag{49}$$

In this case we subtract z_j^2 from λ_j^1 to give the desired $Ax_j + z_j^1$ which is then fed into the second MAP decoder as before. Again, we don't want to include z_j^2 since after interleaving it would be bursty again.

Obviously, in the first stage of decoding we want to obtain the lowest BER possible. However, at low SNR's a code's performance may perform opposite to what is expected. An important observation is that the more powerful a code is at low BER's, the *worse* it performs at low SNR and high BER's [42]. Thus, we don't want to choose too complex a code as iterative decoding will give poor performance. However, we also don't want to choose too weak a code (in terms of performance at low BER) as latter stages of decoding will not be powerful enough to correct any further errors. This was first noticed in [6] where a 16 state code was found to be optimal for a rate 1/2 turbo decoder.

A. Decoder Implementation

The additional circuits required for a turbo decoder are the delays for z_j^2 and $Ax_i + z_i^1$, the interleavers and deinterleavers, the delay for $Ax_k, Ay_k^1, \ldots, Ay_k^{n-1}$, an adder and two subtractors. For the last stage of decoding the deinterleaver can be used to deinterleave the second MAP decoder output with the addition of a multiplexer. With a rate 1/4 MAP decoder, a rate 1/7 turbo decoder could be constructed. To reduce the number of inputs, the data was received serially, one symbol at a time. Thus, BPSK modulation could be directly used, although with some modifications QPSK could also be used.

Each MAP decoder has a maximum delay of 4096 bits and as shown later, each interleaver has a maximum delay of 65,536 bits. Thus, the total delay for the seven symbols is $7 \times 2 \times$ (4096+65,536) = 974,848. Thus, six $1\text{M} \times 1$ SRAM's were used to delay the received data. The MAP decoder was initially designed to have a 16K block size, and thus a 16K delay (for $\nu = 2$). With this extra delay, six $256\text{K} \times 1$ SRAM's were included in the design, but are no longer necessary. Also, two $16\text{K} \times 4$ separate I/O SRAM's were used for each of the z_j^2 and $Ax_i + z_i^1$ delays, although two $4\text{K} \times 4$ SRAM's would be sufficient.

Since the interleaver and deinterleaver architecture are similar, we will only discuss the implementation of the interleaver. For the interleaver we have used the same read/$\overline{\text{write}}$ technique as used by the delay and reversing circuits elsewhere in the design (thus giving a delay equal to the interleaver block size). The maximum interleaver size is the same as in [6], i.e., N_i = 64K. Thus, only two $64\text{K} \times 4$ separate I/O SRAM's are required to implement the interleaver. However, the interleaver address generating circuit is a little more complex and is shown in Figure 16. At startup

the SEL line goes high and the counter output is stored into the SRAM. The SRAM output is then read out and latched by the D–FF's. Using this address the data is sequentially stored into the interleaver SRAM. At the same time SEL goes low and the interleaver address is read from the EPROM and stored into the SRAM, to be read out in the next interleaver block. The process repeats, interleaving the previous set of addresses.

Note that only one interleaver address generator (IAG) is implemented. Thus, only one set of EPROM's needs to be programmed with any interleaver that is desired. The time reversal of the MAP decoder output can also be incorporated into the interleaver EPROM reducing the delay and complexity of the decoder. Due to the two MAP decoder delays (equal to 8K), the interleaver addresses between each iteration also has to be delayed. We used two $8\text{K} \times 8$ SRAM's to perform this task. Since $8\text{K} \times 8$ SRAM's have common I/O, the circuit in Figure 17 was used to separate the I/O and allow the read/$\overline{\text{write}}$ technique to be used.

A disadvantage of the above scheme is that if an error occurs in the address generator, error propagation occurs and the decoder would have to be reset. However, in the many days of testing we performed on our codec, the decoder never had to be reset due to the interleaver failing or any other part of the decoder failing.

The first encoder sequence starts and ends in state zero through the use of a ν–bit tail. Interleaving is performed on all the information and tail bits. The second encoder sequence is also made to start in state zero. However, to keep the same decoder architecture for the second MAP decoder, the N bits are not forced to end in state zero. This implies that the final state is unknown. The second MAP decoder takes this into account by initialising all the reverse state metrics to zero.

Figure 18 shows a photograph of the encoder and IAG's. To the left hand side are seven DIP switches used to program the code polynomials for both the encoder and decoder. The bottom Xilinx chip performs the encoder function. The middle Xilinx chip is the address counter and multiplexer for the IAG.

Figure 19 shows a decoder iteration. The first decoder prototype was implemented using speedwire which allowed any design corrections to be easily made. This prototype was then implemented on a printed circuit board. From the top, the Xilinx chips are the control logic and address generator, MAP decoder 1, MAP decoder 2, data delay address generator and miscellaneous logic (left), and branch metric calculator (right). The large chips to the left and right of the Xilinx chips are the $1\text{K} \times 8$ dual–port SRAM's used to store the new and old state metrics (SM). The left chips store the reverse SM's and the right chips store the forward SM's. To the left of the control logic chip are the $64\text{K} \times 4$ SRAM's where we store the branch metrics to be used in calculating the

reverse SM's. The four 64K × 4 SRAM's to the right of the control logic chip are used to store the forward SM's to be used in calculating the log–likelihood ratio. Between the bottom two Xilinx chips are the 16K × 4 SRAM's for delaying the input and producing the extrinsic information. The 12 SRAM's at the bottom left of the board are for delaying the n 6–bit inputs (six 256K × 1 and six 1M × 1 SRAM's). The four 64K × 4 SRAM's at the bottom right hand corner perform the deinterleaving and interleaving of the data.

Figure 20 is a photograph of the completed codec. A 6U high 48 cm rack is used which contains from left to right, the encoder/interface card, turbo/MAP decoder card 1, interleaver address delay card 1, turbo/MAP decoder cards 2 to 5, interleaver address delay card 2, and turbo/MAP decoder cards 6 and 7. An additional 11 turbo/MAP decoder cards can fit within the rack to give a total of 18 iterations. Turbo/MAP decoder 7 has its switches in the up position, indicating the decoder is set up for seven iterations. Each iteration can also output the first decoder output, using a 16K × 4 SRAM from the second MAP decoder to reverse the data in time.

Note that our current decoder implementation is not able to automatically synchronise to a received signal. Instead, a signal from the encoder was used to synchronise the decoder. Future modifications may include a synchronisation word in the encoded block to allow synchronisation.

B. Turbo Decoder Performance

The first tests were made using a rate 1/3 turbo code using identical rate 1/2 16 state codes with polynomials $g_0 = 31$ and $g_1 = 33$ (in octal) [43]. These code polynomials were optimised for rate 1/3 turbo codes and were found to perform better than the codes from [6] (which were optimised for a rate 1/2 turbo code). The value of A was set to 15 for all E_b/N_0, which is close to the optimum values. An S = 31 [44], 65,536 bit interleaver was used (S = 31 implies that any two consecutive bits are separated by at least 31 other bits after interleaving). The actual rate is reduced by 4092/4096 = 1023/1024 due to the MAP block size being only 4096 bits, with 4 bits used as the tail. The "inner" code is terminated to state 0 while the "outer" code is not terminated. Shannon capacity at this rate is at an $E_b/N_0 = -0.55$ dB (the capacity at this rate with QPSK modulation is –0.49 dB).

In Figure 21 we plot BER versus E_b/N_0 for 6.5 and 7 iterations (a half iteration is the output of the first MAP decoder). We see that 10^{-5} and 10^{-6} is achieved at 0.32 dB and 0.38 dB, respectively. This is 0.87 dB and 0.93 dB away from rate 1/3 capacity for BER's of 10^{-5} and 10^{-6}, respectively. Unfortunately, 10^{-7} is close to an E_b/N_0 of 1.0 dB due to the BER flattening above 0.4 dB. We also see that above 0.4 dB, the BER from 6.5 iterations performs better than 7 iterations. This may be due to the second MAP decoder being unterminated or perhaps an effect of other non–linearities in the decoder.

Also note how the performance shallows between 0.35 and 0.4 dB. There appears to be a sudden change in slope. As can be seen, it is not an error floor since the BER does decrease with increasing E_b/N_0. The reason for this sudden change in slope is due to the small free distance of turbo codes [46]. This free distance has a spectral component much less than one which greatly reduces its error probability (in fact this reduction is inversely proportional to N_i [46]). However, since the free distance term has a shallow slope, this slope will appear at relatively high E_b/N_0.

Computer simulations for this scheme have not been previously performed. Thus, we make a comparison to the rate 1/3, 16 state, and N_i = 16,384 scheme in [45]. With 11 iterations they achieved an E_b/N_0 of 0.24 dB at a BER of 10^{-5}. This is 0.08 dB better than our scheme which has 7 iterations and an interleaver that is four times larger.

The first MAP decoder had mostly double bit error outputs at low BER. This is predicted in [46] where the use of systematic convolutional codes leads to error sequences of information weight two (corresponding to the error bit causing the sequence to leave the path and the error bit causing the sequence to return to the path). This is very important for turbo codes since it greatly increases its performance over the use of non–systematic encoders (which have single information bit error patterns) [46]. Also, systematic encoders perform slightly better over non–systematic encoder at low SNR [42] which is important in iterative decoding.

The output from the second MAP decoder at low BER's was mostly in single bit errors, indicating that the unterminated states could be causing a problem. Since there are N_i/N = 16 MAP blocks in each interleaver block, the effect of all these unterminated states could induce these single bit errors.

It was found that if the subtractor outputs in Figure 15 were limited to have a range from –128 to +127, the decoder would perform very poorly. When the extrinsic information is added to the received noisy sample (which ranges from –31 to +31) errors could be introduced due to non–linearities in the design. For example, say we receive +31 and add the extrinsic information +127 to it. The output of the first MAP decoder will be limited to +127. When we subtract the extrinsic information the information fed into the second MAP decoder will then be zero! The results shown in Figure 15 had the subtractor outputs limited from –64 to +63 to avoid this problem.

In Figure 22 we plot BER against the number of iterations for E_b/N_0 = 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, and 1.0 dB. Of interest is how the BER suddenly flattens after quickly decreasing for 0.4, 0.5, and 1.0 dB. Another point of interest is the potential of more iterations. The 0.2 dB curve indicates that further improvement is possible and the 0.1 dB curve might be able to reach low BER's as well.

Figure 23 shows the performance of our rate 1/7 16 state turbo decoder with code polynomials g_0 = 23, g_1 = 35, g_2 = 27, and g_3 = 37 from [45]. To obtain better performance, the extrinsic information subtractor outputs were limited from –96 to +95. The value of A was set to 7 to avoid limiting the state metrics too greatly. The 64K interleaver from [24] was tried, but was found to give inferior performance to our randomly generated interleaver (although it must be noted that this interleaver was designed for a rate 1/2 turbo code).

An E_b/N_0 of –0.30, –0.27, and –0.19 dB is achieved for BER's of 10^{-5}, 10^{-6}, and 10^{-7}, respectively. Note that the 10^{-7} BER is achieved with 6.5 iterations. Shannon capacity at this rate is at –1.12 dB, from which we are 0.82 dB away at a BER of 10^{-5}. With more iterations, we expect to reduce this gap by 0.1 to 0.2 dB.

V. Continuous Decoding and Synchronisation

A way of improving the decoding design is to use a continuous MAP decoder. A continuous decoder does not need a tail to be added to the sequence and only needs to synchronise to the n coded symbols. We will give a description of an efficient implementation of a continuous MAP decoder along with a synchronisation technique that can be used for a turbo decoder.

A. Continuous Decoding

A continuous decoding algorithm for the MAP algorithm was first presented for the ISI channel in [47] (with a complexity exponentially proportional to the decoder delay). A simpler algorithm for continuous decoding of convolutional codes was first described in [48] and later in [29,49,50]. A similar algorithm for the ISI channel is given in [51]. The n received data symbols (R_k) are stored from time k to $k + L - 1$. They are then read out in reverse order from time $k + L - 1$ to k and β_{k+1}^m determined recursively (starting with $\beta_{k+L}^m = 1$ for all m). The forward SM's α_k^m are also calculated as normal using the R_k that have been delayed by $2L$. Using α_k^m, $\delta_k^{i,m}$, and β_{k+1}^m, λ_k is determined as normal. We then increment k by one and repeat the whole algorithm. By making L sufficiently large, the β_{k+1}^m that is determined will be very close to the true β_{k+1}^m had β_{k+L}^m been known precisely. We can see that this algorithm is computationally intensive since each β_k^m is calculated L times instead of only once.

A computationally simpler algorithm was first mentioned in [18] and later in more detail in [52–54]. As for the previous algorithm we store R_k into a RAM of size nL. The reverse SM's are then calculated (starting with $\beta_{k+L}^m = 1$ for all m). However, the reverse SM's continue to be calculated from time k to $k - L + 1$ using the R_k from the reversing RAM. After the R_k have been delayed by $2L$ (using another RAM of size $2nL$) the forward SM's are calculated and stored in a

RAM of size $2^{\nu}L$. The time reversed forward SM's are combined with the last L reverse SM's to give a time reversed LLR output. We can see that only half the reverse SM's that are calculated are used. Thus, two reverse SM calculators are used in pipeline.

Figure 24 shows how R_k is stored and read for use by the reverse SMC's. In this case we have assumed that $L = 64$. The first RAM is used to time reverse R_k in blocks of L and the second RAM is used to delay the reversed R_k by $2L$. Two multiplexers are then used to alternatively select the RAM outputs for use by the two RSMC's.

Using the read/$\overline{\text{write}}$ technique, the total amount of RAM that is required is $5L \times nq + L2^{\nu} \times 8$ where we have assumed eight bit SM's and LLR. An additional $2^{\nu} \times 8$ bits would be required if the output had to be reversed in time. This is still considerably less complex than having L RSMC's as in the previous algorithm. If the decoder is to be used in a turbo decoder, this reversal can be performed within the interleaver. The total delay is $4L$ (with the LLR reverser) or $3L$ (without the LLR reverser).

B. A Synchronisation Technique for Turbo Decoders

In a turbo decoder an important consideration is being able to synchronise to the interleaved data of depth N_i. One could insert synchronisation words, but this increases complexity and decreases the bandwidth efficiency. The first MAP decoder needs to only synchronise to n possible states. This can be done by monitoring the average amplitude of L_k. When the decoder is out of synchronisation, the average value of $|L_k|$ will be lower than expected. This is due to the decoder not receiving a valid code sequence and producing an unreliable decoded sequence.

By accumulating $|L_k|$ for a certain length of time and comparing it to a threshold, the decoder can make a reliable decision as to whether it is synchronised. If the decoder is not synchronised, it tries the next state and then waits a decoder delay (in order for the decoder to produce stable data) before it starts monitoring $|L_k|$ again. This process continues until synchronisation is achieved.

The average synchronisation time (t_s) depends on the decoder delay (t_d), the time to average $|L_k|$ (t_a), and the number of synchronisation states (n_s). Since we can randomly start in any state (taking on average $(n_s - 1)/2$ attempts before synchronisation is achieved) we have that

$$t_s = (t_d + t_a)(n_s - 1)/2. \tag{50}$$

The worst case synchronisation time is only twice the average synchronisation time. For a turbo decoder we have

$$t_s = (t_d + t_a)(n - 1)/2 + (N_i + t_d + t_a)(N_i - 1)/2. \tag{51}$$

The first part of (51) corresponds to the first MAP decoder and the second part to the second MAP decoder. For large N_i, the synchronisation time will be dominated by the second MAP decoder.

For example, if $t_d = 3L = 192$, $t_a = 64$, $n = 2$, and $N_i = 65{,}536$ then $t_s = 2{,}155{,}839{,}488$ bits! At 2.048 Mbit/s is would thus take on average about 17.5 minutes to synchronise. Once synchronised, one should increase t_a so that the synchronisation circuit does not indicate a false alarm with very high probability.

A way of reducing the overall synchronisation time is to evenly distribute the synchronisation time between the two decoders. We could do this by adding modulo–2 a random binary sequence of length L_1 to the parity of the first individual code. This forces the inner decoder to synchronise to nL_1 states.

We also have that $L_1 L_2 = N_i$ (to keep the overall number of synchronisation states the same). The outer decoder now needs only to synchronise to L_2 states. The average synchronisation time is then

$$t_s = (t_d + t_a)(nL_1 - 1)/2 + (N_i + t_d + t_a)(L_2 - 1)/2. \tag{52}$$

One can then try various values of L_1 and L_2 to minimise (52). Assuming a real valued L_2, the optimum value of L_1 is (taking the differential of (52) and setting to zero)

$$L_1 = \sqrt{\frac{N_i(N_i + t_d + t_a)}{n(t_d + t_a)}}. \tag{53}$$

For our example we thus have $L_1 = 2901.96$ and $L_2 = 22.58$. Since L_1 and L_2 must be integers we let $L_1 = 2849$, $L_2 = 23$ and $N_i = 65{,}527 = 2^{16} - 9$. From (52) we have $t_s = 1{,}452{,}829$ bits, a nearly 1500 times reduction compared to the original scheme. At 2.048 Mbit/s, this corresponds to an average synchronisation time of 0.709 seconds.

Alternatively, we could let $L_1 = 4096$ and $L_2 = 16$ which only slightly increases t_s to 1,541,888 bits (a delay of 0.753 seconds at 2.048 Mbit/s). $L_1 = 2048$ and $L_2 = 32$ is only marginally slower at 1,543,936 bits or 0.754 seconds.

VI. Conclusions

The original turbo coding paper by Berrou et al. [6] presented a coding scheme that came within 0.7 dB of Shannon capacity. The MAP decoding algorithm that was presented was much too complicated to implement practically. We have re–derived the MAP algorithm to present it in as simple form as possible. By taking the logarithm of the MAP algorithm a realisable implementation could be achieved at high data rates.

The turbo decoder that we have constructed has indeed been able to verify the amazing performance presented in [6]. We were able to come within 0.8 dB of capacity with only seven iterations. With up to 18 iterations we can expect to reduce this amount by 0.1 to 0.2 dB. Thus, we have been able to demonstrate that near–Shannon performance can be achieved at high data rates.

We used a block type MAP algorithm which requires a lot of memory for its implementation. An efficient implementation of a continuous decoder has been presented. Continuous decoders have the advantage of being less complex, easier to synchronise, a much smaller delay, and have slightly better performance.

A synchronisation technique for turbo decoders which takes advantage of the low delay of continuous decoders has also been presented. It is shown that even for large interleaver sizes, automatic synchronisation can be achieved in a relatively small time interval.

ACKNOWLEDGMENT

The author would like to thank Riccardo Macri for providing the AWGN noise generating program.

REFERENCES

[1] C. E. Shannon, "A mathematical theory of communications," *Bell Sys. Tech. J.*, vol. 27, pp. 379–423, July 1948 and pp. 623–656, Oct. 1948.

[2] P. E. McIllree, "Channel capacity calculations for M–ary N–dimensional signal sets," M.Eng. Thesis, Uni. of South Australia, Feb. 1995.

[3] J. P. Odenwalder, "Optimal decoding of convolutional codes," Ph.D. dissertation, Uni. of California, Los Angeles, CA, 1970.

[4] E. C. Posner, L. L. Rauch, and B. D. Madsen, "Voyager mission telecommunication firsts," *IEEE Commun. Mag.*, vol. 28, pp. 22–27, Sep. 1990.

[5] S. Dolinar and M. Belongie, "Enhanced decoding for the Galileo low–gain antenna mission: Viterbi redecoding with four decoding stages," *JPL TDA Progress Report*, vol. 42–121, pp. 96–109, 15 May 1995.

[6] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error–correcting coding and decoding: Turbo–Codes," *IEEE Int. Conf. Commun.*, Geneva, Switzerland, pp. 1064–1070, May 1993.

[7] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv , "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. IT–20, pp. 284–287, Mar. 1974.

[8] R. G. Gallager, "Low–density parity check codes," *IRE Trans. Inform. Theory*, vol. IT–8, pp. 21–28, Jan. 1962.

[9] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inform. Theory*, vol. IT–13, pp. 260–269, Apr. 1967.

[10] S. A. Barbulescu, W. Farrell, P. Gray, and M. Rice, "Bandwidth efficient turbo coding for high speed mobile satellite communications," *Int. Symp. on Turbo Codes*, Brest, France, pp. 119–126, Sep. 1997.

[11] G. Battail, "Pondération des symbols décodés par l'algorithm de Viterbi," (in French), *Annales des Télécommun.*, vol. 42, pp. 1/1–1/8, Jan./Feb. 1987.

[12] J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft–decision outputs and its applications," *GLOBECOM'89*, Dallas, Texas, pp. 47.1.1–47.1.7, Nov. 1989.

[13] J. Huber and A. Rüppel, "Zuverlässigkeitsabschätzung für die ausgangssymbole von trellis–decodern," (in German), *AEÜ (Electronics and Communications)*, vol. 44, pp. 8–21, Jan./Feb. 1990.

[14] C. Berrou, P. Adde, E. Angui, and S. Faudeil, "A low complexity soft–output Viterbi decoder architecture," *IEEE Int. Conf. Commun.*, Geneva, Switzerland, pp. 737–740, May 1993.

[15] J. Hagenauer, P. Robertson, and L. Papke, "Iterative (Turbo) decoding of systematic convolutional codes with the MAP and SOVA algorithms," *ITG Tagung, Codierung für Quelle, Kanal und Übertragung*, Frankfurt, Germany, pp. 21–29, Oct. 1994.

[16] C. Berrou and G. Lochon, "CAS 5093 turbo–code codec," data sheet, COMATLAS, Chateaubourg, France, Nov. 1993.

[17] M. Jézéquel, C. Berrou, J. R. Inisan, and Y. Sichez, "Test of a turbo–encoder/decoder," *Turbo Coding Seminar*, Lund, Sweden, pp. 35–42, Aug. 1996.

[18] S. S. Pietrobon and S. A. Barbulescu, "A simplification of the modified Bahl decoding algorithm for systematic convolutional codes," *Int. Symp. Inform. Theory & its Applic.*, Sydney, Australia, pp. 1073–1077, Nov. 1994.

[19] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub–optimal MAP decoding algorithms operating in the log domain," *ICC'95*, Seattle, WA, USA, pp. 1009–1013, June 1995.

[20] R. W. Chang and J. C. Hancock, "On receiver structures for channels having memory," *IEEE Trans. Inform. Theory*, vol. IT–12, pp. 463–468, Oct. 1966.

[21] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Int. Symp. Inform. Theory*, Asilomar, CA, p. 90, May 1972.

[22] P. L. McAdam, L. R. Welch, and C. L. Weber, "M.A.P. bit decoding of convolutional codes," *IEEE Int. Symp. Inform. Theory*, Asilomar, CA, p. 91, May 1972.

[23] P. Robertson, “Improving decoder and code structure of parallel concatenated recursive systematic (turbo) codes,” *Int. Conf. Universal Personal Commun.*, San Diego, CA, pp. 183–187, Sep.–Oct. 1994.

[24] C. Berrou and A. Glavieux, “Near optimum error correcting coding and decoding: Turbo–codes,” *IEEE Trans. Commun.*, vol. 44, pp. 1261–1271, Oct. 1996.

[25] S. S. Pietrobon, “Implementation and performance of a serial MAP decoder for use in an iterative turbo decoder,” *IEEE Int. Symp. Inform. Theory*, Whistler, British Columbia, Canada, p. 471, Sep. 1995.

[26] J. Erfanian, S. Pasupathy, and G. Gulak, “Reduced complexity symbol detectors with parallel structures for ISI channels,” *IEEE Trans. Commun.*, vol. 42, pp. 1661–1671, Feb./Mar./Apr. 1994, Part III.

[27] W. Koch and A. Baier, “Optimum and sub–optimum detection of coded data disturbed by time varying intersymbol interference,” *GLOBECOM’90*, San Diego, CA, pp. 1679–1684, Dec. 1990.

[28] E. Villebrun, “Turbo–decoding with close–to–optimal MAP algorithms,” TU Munich, Diploma thesis, Sep. 1994.

[29] S. Benedetto, G. Montorsi, D. Divsalar, and F. Pollara, “Soft–output decoding algorithms in iterative decoding of turbo codes,” *JPL TDA Progress Report*, vol. 42–124, pp. 63–87, 15 Feb. 1996.

[30] J. Petersen, “Implementierungsaspekte zur symbol–by–symbol MAP decodieurung von faltungscodes,” *ITG Tagung, Codierung für Quelle, Kanal und Übertragung.*, Frankfurt, Germany, pp. 41–48, Oct. 1994.

[31] S. S. Pietrobon, “Discrete implementation of a NASA planetary standard Viterbi decoder,” *IREE Int. Electron. Conv. and Exhib.*, pp. 249–252, Sydney, Australia, Sep. 1987.

[32] S. S. Pietrobon and D. J. Costello, Jr., “A bandwidth efficient coding scheme for the Hubble Space Telescope,” *J. Electrical and Electron. Eng., Australia*, vol. 13, pp. 275–282, Dec. 1993.

[33] Xilinx, “The programmable logic data book,” San Jose, CA, 1996.

[34] O. M. Collins, “The subtleties and intricacies of building a constraint length 15 convolutional decoder,” *IEEE Trans. Commun.*, vol. 40, pp. 1810–1819, Dec. 1992.

[35] T. A. Summers and S. G. Wilson, “SNR mismatch and on–line estimation in turbo decoding,” *IEEE Trans. Commun.*, vol. 46, pp. 421–423, Apr. 1998.

[36] I. M. Onyszchuk, K.–M. Cheung, and O. Collins, "Quantization loss in convolutional decoding," *IEEE Trans. Commun.*, vol. 41, pp. 261–265, Feb. 1993.

[37] S. S. Pietrobon, J. J. Kasparian, and P. K. Gray, "A multi–D trellis decoder for a 155 Mbit/s concatenated codec," *Int. J. of Satellite Commun.*, vol. 12, pp. 539–553, Nov.–Dec. 1994.

[38] S. K. Park and K. W. Miller, "Random number generators: Good ones are hard to find," *Commun. of the ACM*, vol. 31, pp. 1192–1201, Oct. 1988.

[39] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, "Numerical recipes in C: The art of scientific computing," Cambridge University Press, Cambridge, 1988.

[40] P. L'Ecuyer, "Efficient and portable combined random number generators," *Commun. of the ACM*, vol. 31, pp. 742–749, 774, June 1988.

[41] P. J. Lee, "Further results on rate $1/N$ convolutional code constructions with minimum required SNR criterion," *IEEE Trans. Commun.*, vol. COM–34, pp. 395–399, Apr. 1986.

[42] J. Y. Couleaud, "High gain coding schemes for space communications," ENSICA Final Year Report, Uni. of South Australia, Sep. 1995.

[43] S. Benedetto and G. Montorsi, "Design of parallel concatenated convolutional codes," *IEEE Trans. Commun.*, vol. 44, pp. 591–600, May 1996.

[44] D. Divsalar and F. Pollara, "Multiple turbo codes for deep–space communications," *JPL TDA Progress Report*, vol. 42–121, pp. 66–77, 15 May 1995.

[45] D. Divsalar and F. Pollara, "On the design of turbo codes," *JPL TDA Progress Report*, vol. 42–123, pp. 99–121, 15 Nov. 1995.

[46] S. Benedetto and G. Montorsi, "Unveiling turbo codes: Some results on parallel concatenated coding schemes," *IEEE Trans. Inform. Theory*, vol. 42, pp. 409–428, Mar. 1996.

[47] K. Abend and B. D. Fritchman, "Statistical detection for communication channels with intersymbol interference," *Proc. IEEE*, vol. 58, pp. 779–785, May 1970.

[48] L.–N. Lee, "Real–time minimal–bit–error probability decoding of convolutional codes," *IEEE Trans. Commun.*, vol. COM–22, pp. 146–151, Feb. 1974.

[49] K.–H. Tzou and J. G. Dunham, "Sliding block decoding of convolutional codes," *IEEE Trans. Commun.*, vol. COM–29, pp. 1401–1403, Sep. 1981.

[50] X. Wang and S. B. Wicker, "A soft–output decoding algorithm for concatenated codes," *IEEE Trans. Inform. Theory*, vol. 42, pp. 543–553, Mar. 1996.

[51] Y. Li, B. Vucetic, and Y. Sato, "Optimum soft–output detection for channels with intersymbol interference," *IEEE Trans. Inform. Theory*, vol. 41, pp. 704–713, May 1995.

[52] S. A. Barbulescu, “Iterative decoding of turbo codes and other concatenated codes,” Ph.D. dissertation, *Uni. of South Australia*, pp. 23–24, Feb. 1996.

[53] S. S. Pietrobon, “Efficient implementation of continuous MAP decoders and a synchronisation technique for turbo decoders,” *Int. Symp. on Inform. Theory and its Applications*, pp. 586–589, Victoria, BC, Canada, Sep. 1996.

[54] A. J. Viterbi, “An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes,” submitted to *IEEE J. Sel. Areas Commun.*, Oct. 1996.

GLOSSARY OF ABBREVIATIONS

ACS	Add–Compare–Select
AGC	Automatic Gain Control
APoP	A Posteriori Probability
APrP	A Priori Probability
AWGN	Additive White Gaussian Noise
BER	Bit Error Ratio
BM	Branch Metric
BMC	Branch Metric Calculator
BPSK	Binary Phase Shift Keying
CE	Clock Enable
CLB	Configurable Logic Block
CLK	Clock
CP	Clock Input
CTL	Control
D–FF	Data Flip Flop
DCLK	Data Clock
DEC	Decoder
DEINT	Deinterleaver
DEL	Delay
DIP	Dual Inline Package
DP–RAM	Dual Port Random Access Memory
EM	Eccumalator Metric
ENC	Encoder
EPROM	Erasable Programmable Read Only Memory
FSM	Forward State Metric

FSMC	Forward State Metric Calculator
I/O	Input/Output
IAG	Interleaver Address Generator
INT	Interleaver
ISI	Intersymbol Interference
LLR	Log Likelihood Ratio
LLRC	Log Likelihood Ratio Calculator
MA	Multiply–Add
MAP	Maximum a Posteriori
MB	Megabyte
MUX	Multiplexer
OE	Output Enable
PC	Personal Computer
PM	Path Metric
QPSK	Quadrature Phase Shift Keying
RAM	Random Access Memory
RS	Reed Solomon
RSM	Reverse State Metric
RSMC	Reverse State Metric Calculator
SEL	Select
SISO	Soft–In Soft–Out
SM	State Metric
SMC	State Metric Calculator
SNR	Signal to Noise Ratio
SRAM	Static Random Access Memory
SOVA	Soft Output Viterbi Algorithm
WE	Write Enable
XNOR	Exclusive Negative OR

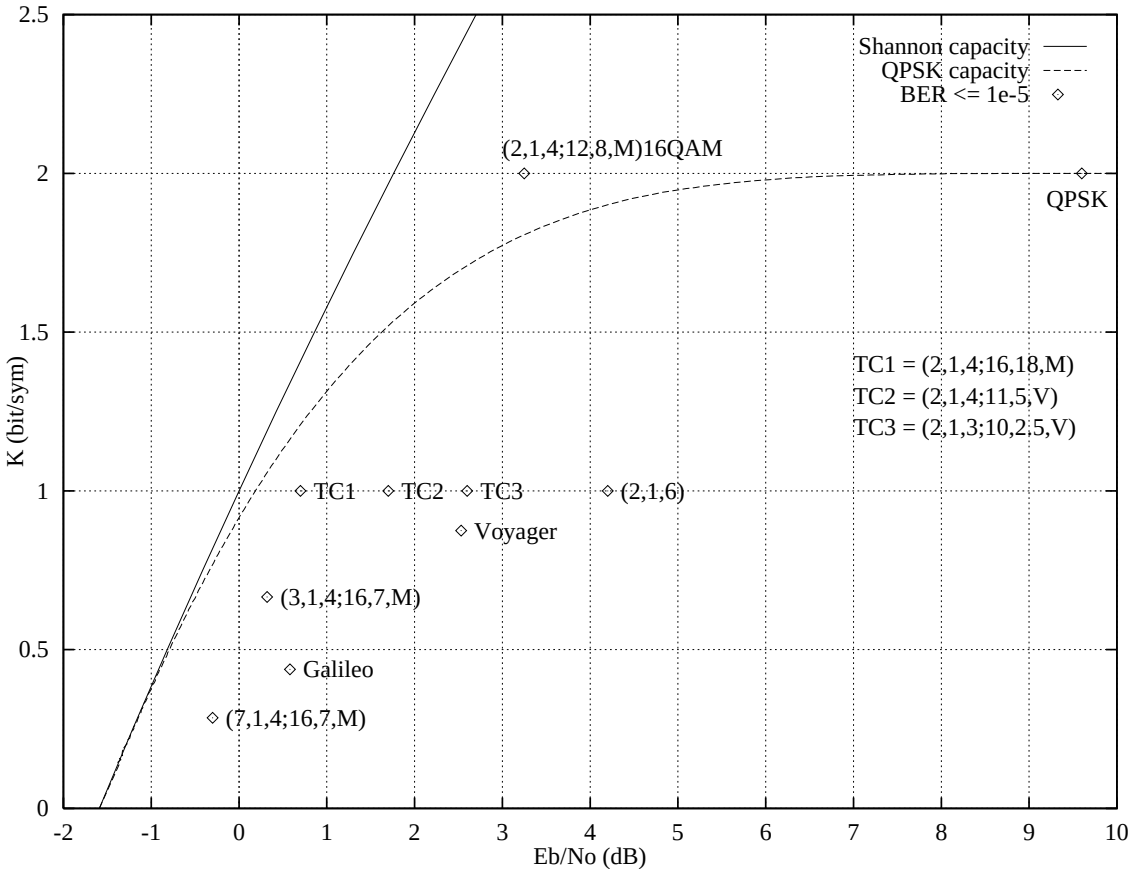

Figure 1: Shannon and QPSK capacity.

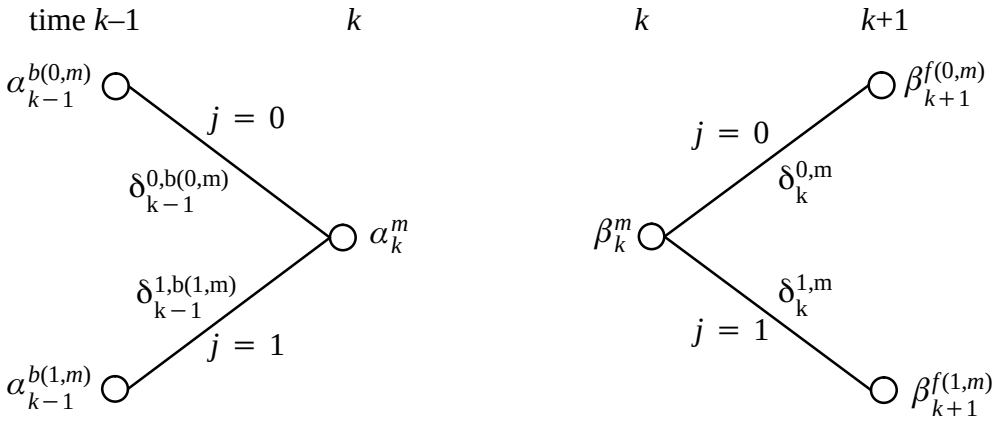

Figure 2: Graphical representation of calculation of α_k^k and β_k^m.

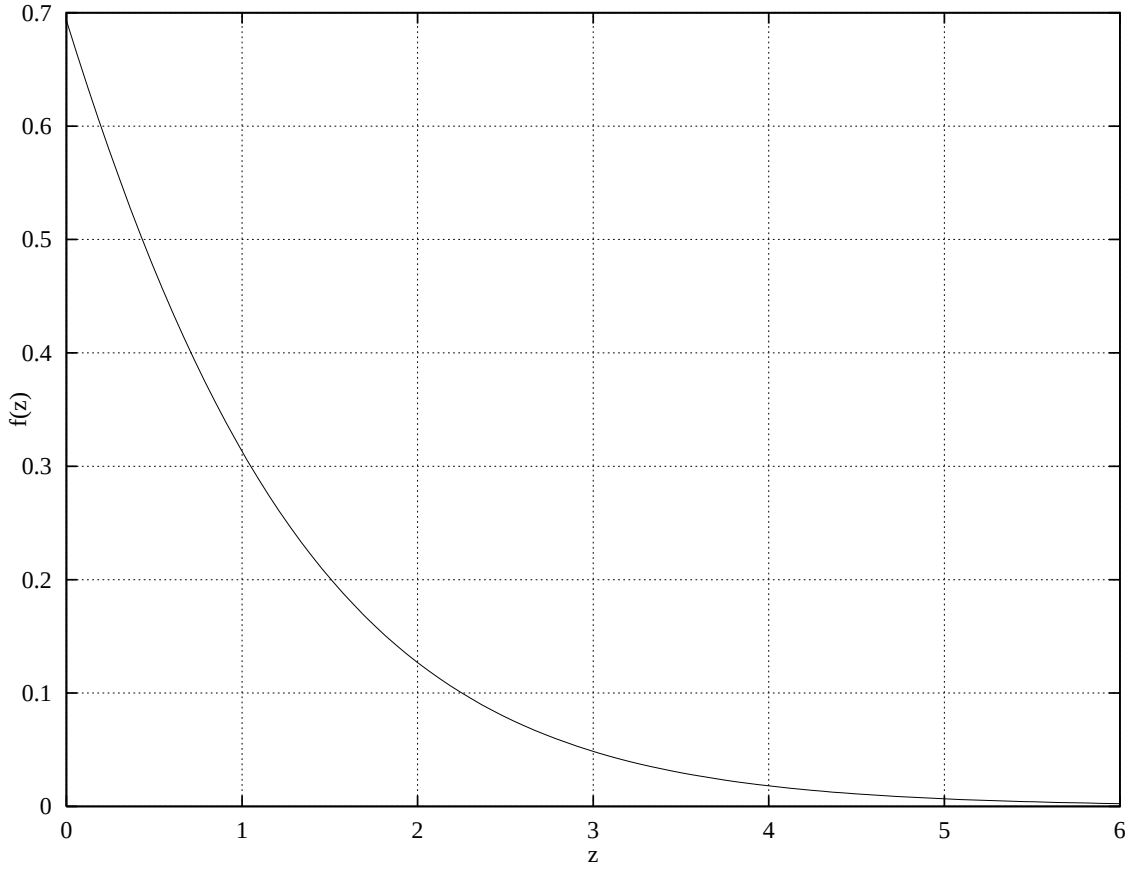

Figure 3: $f(z)$ versus z for $c = 1$.

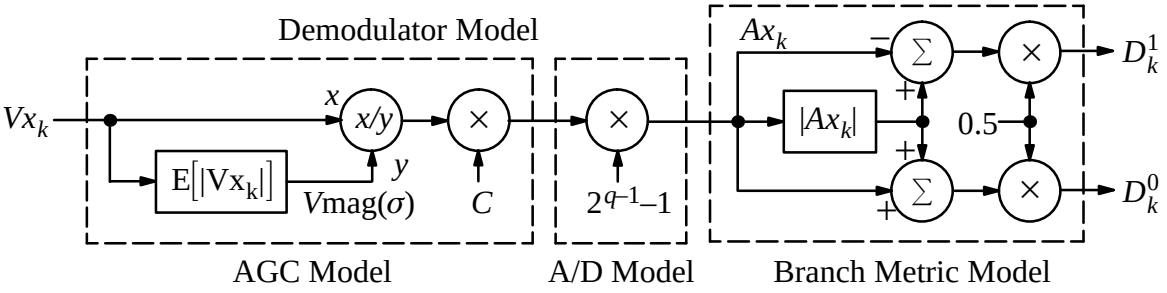

Figure 4: Demodulator and Branch Metric Model

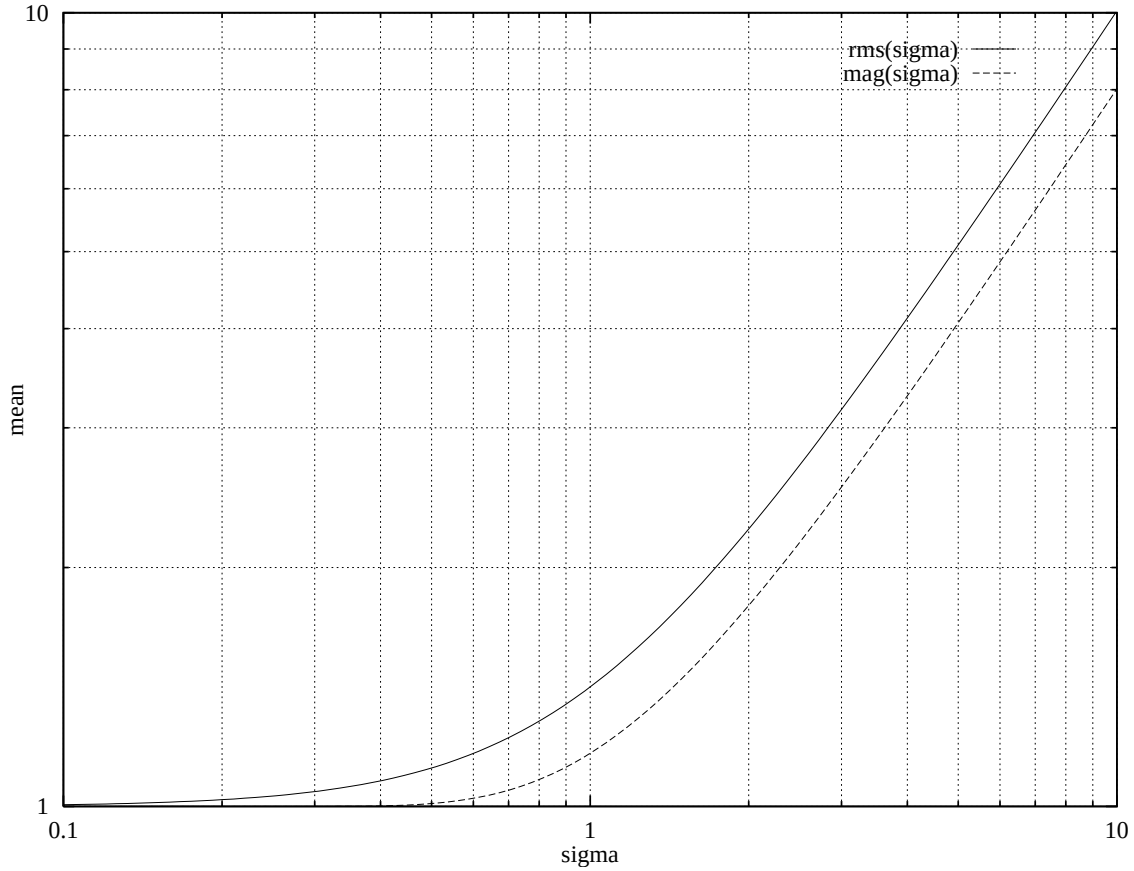

Figure 5: Absolute mean and root–mean–square of received signal.

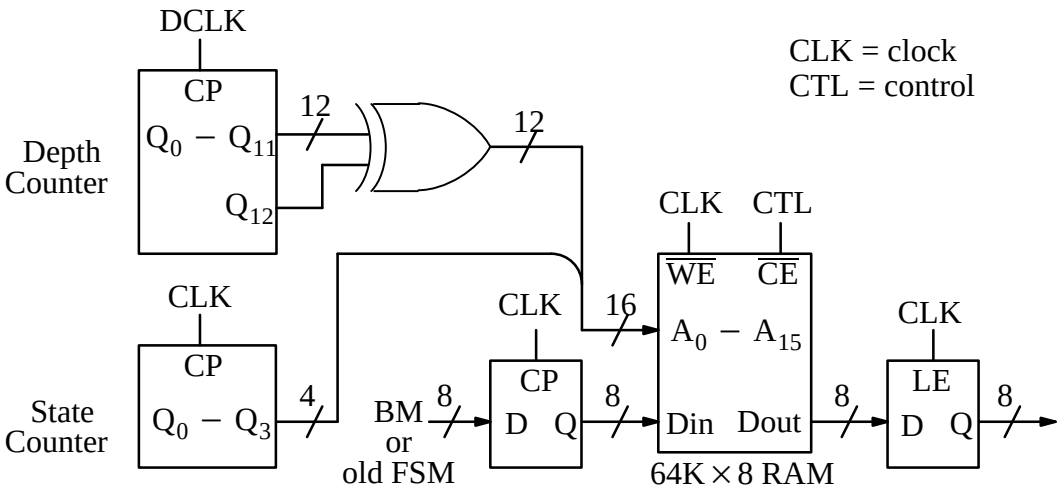

Figure 6: Branch and State Metric storage for block MAP decoder.

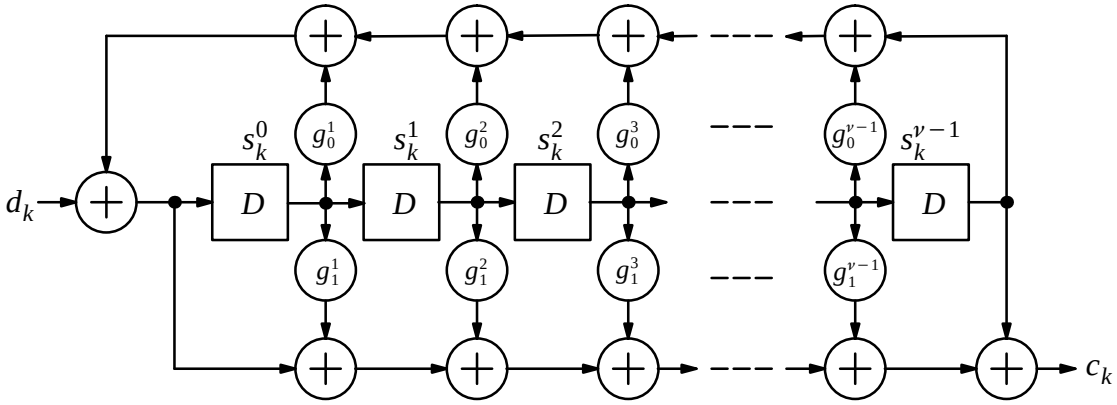

Figure 7: Rate 1/2 systematic convolutional encoder.

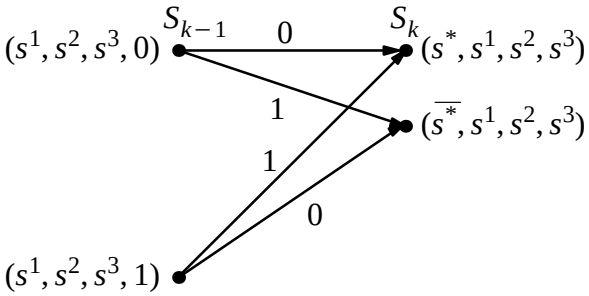

Figure 8: Sub–trellis for $\nu = 4$ rate 1/n systematic convolutional code.

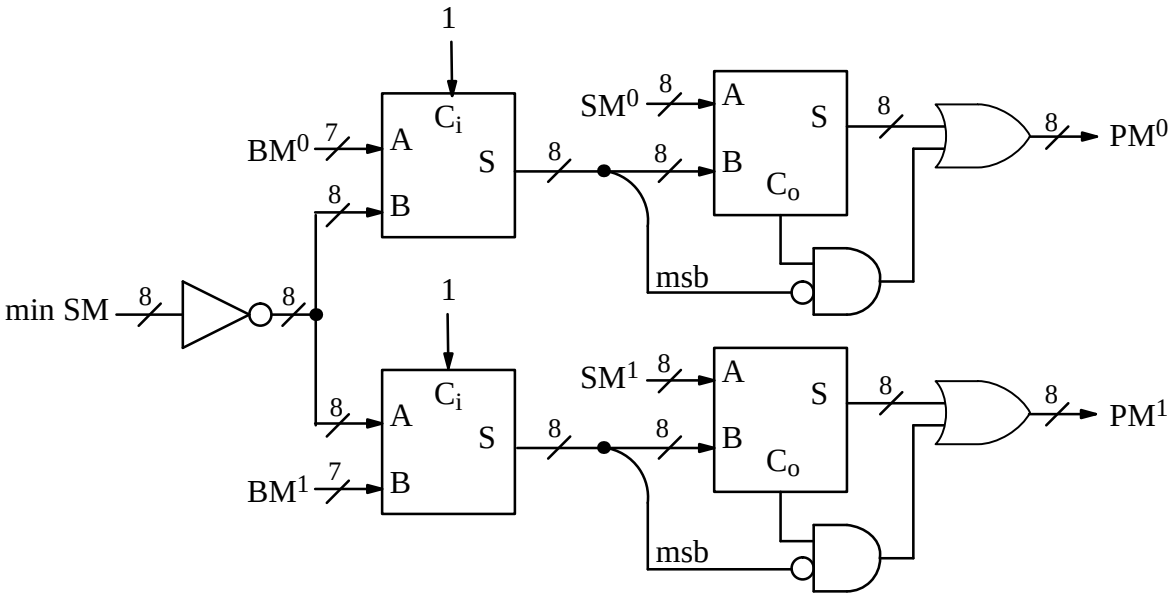

Figure 9: BM and SM adder circuit.

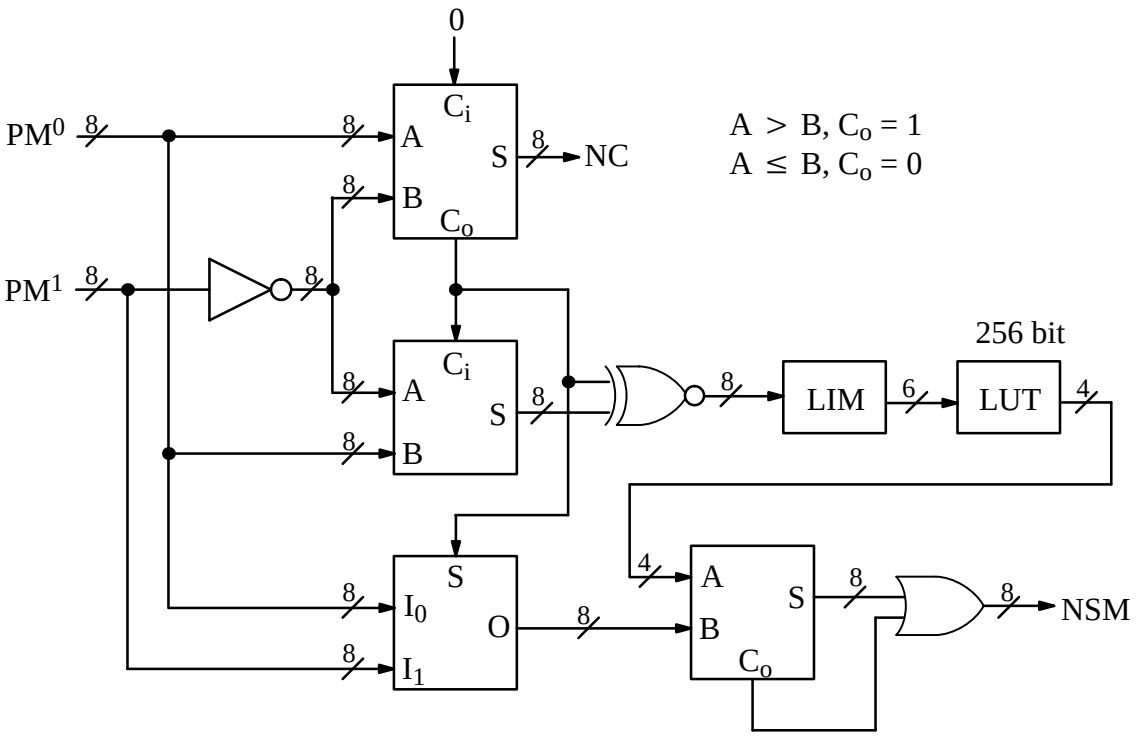

Figure 10: Add–Compare–Select–E circuit.

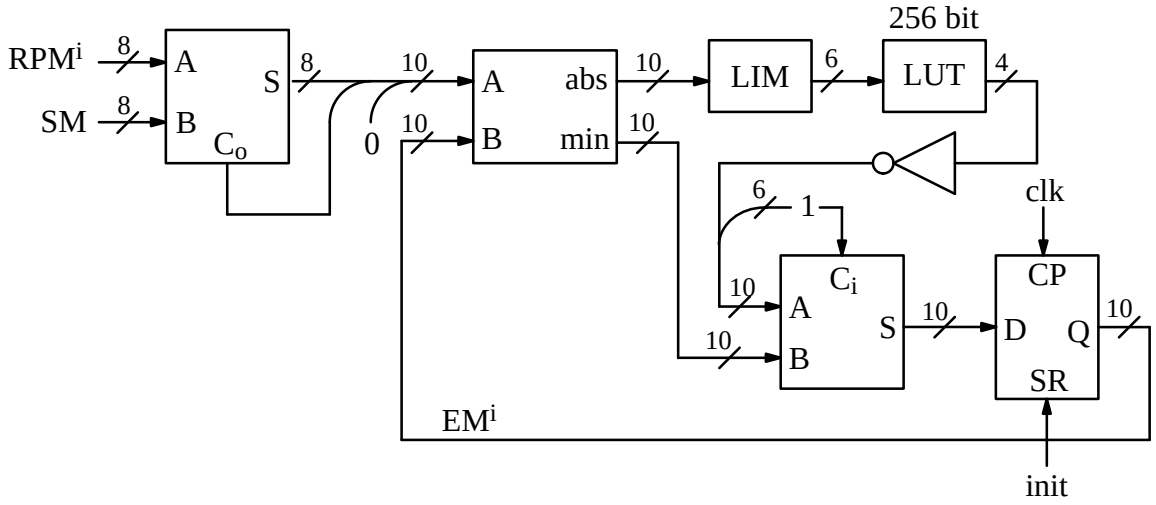

Figure 11: Eccumulator circuit.

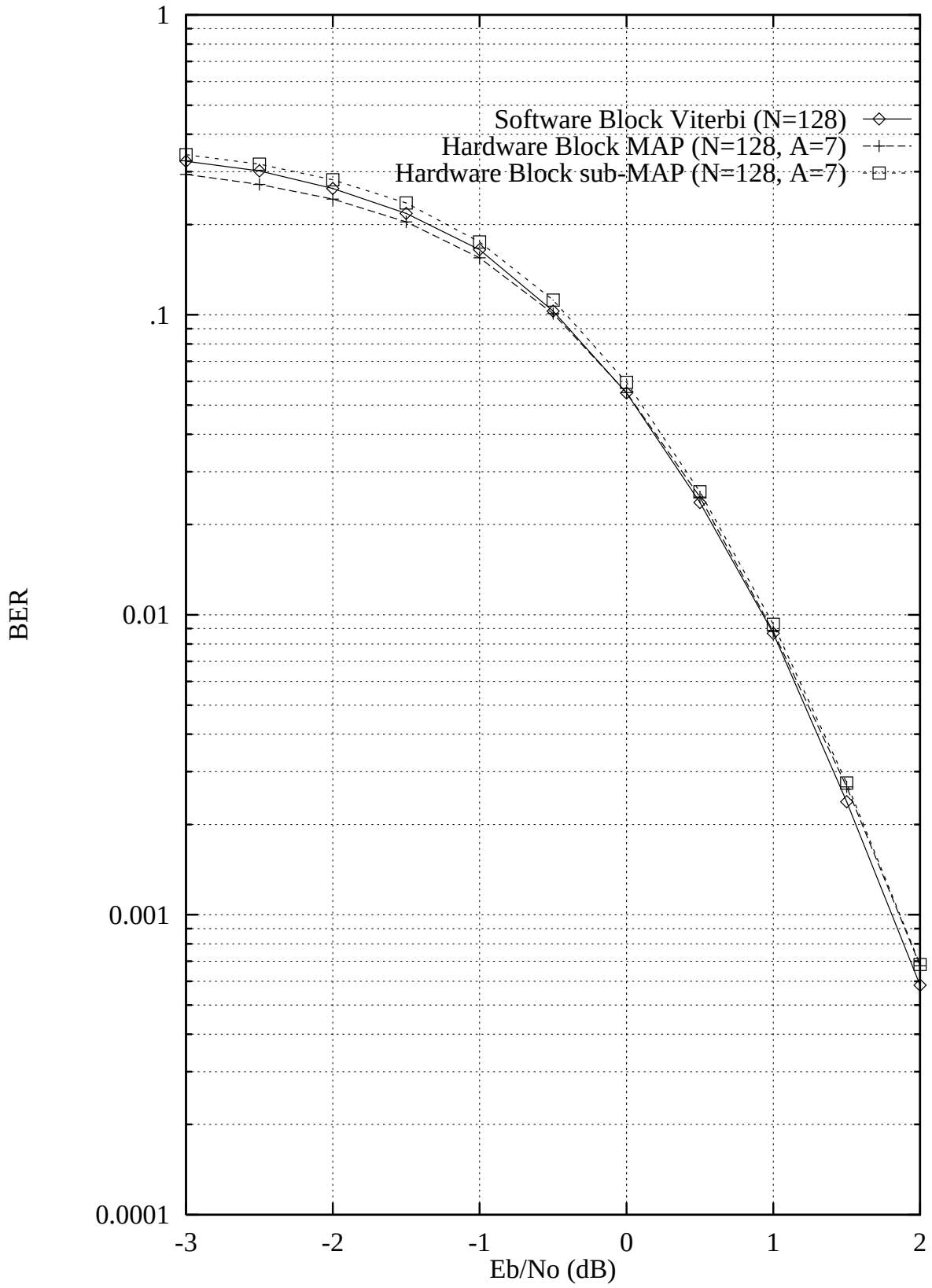

Figure 12: Systematic rate 1/4, 512 state BER performance.

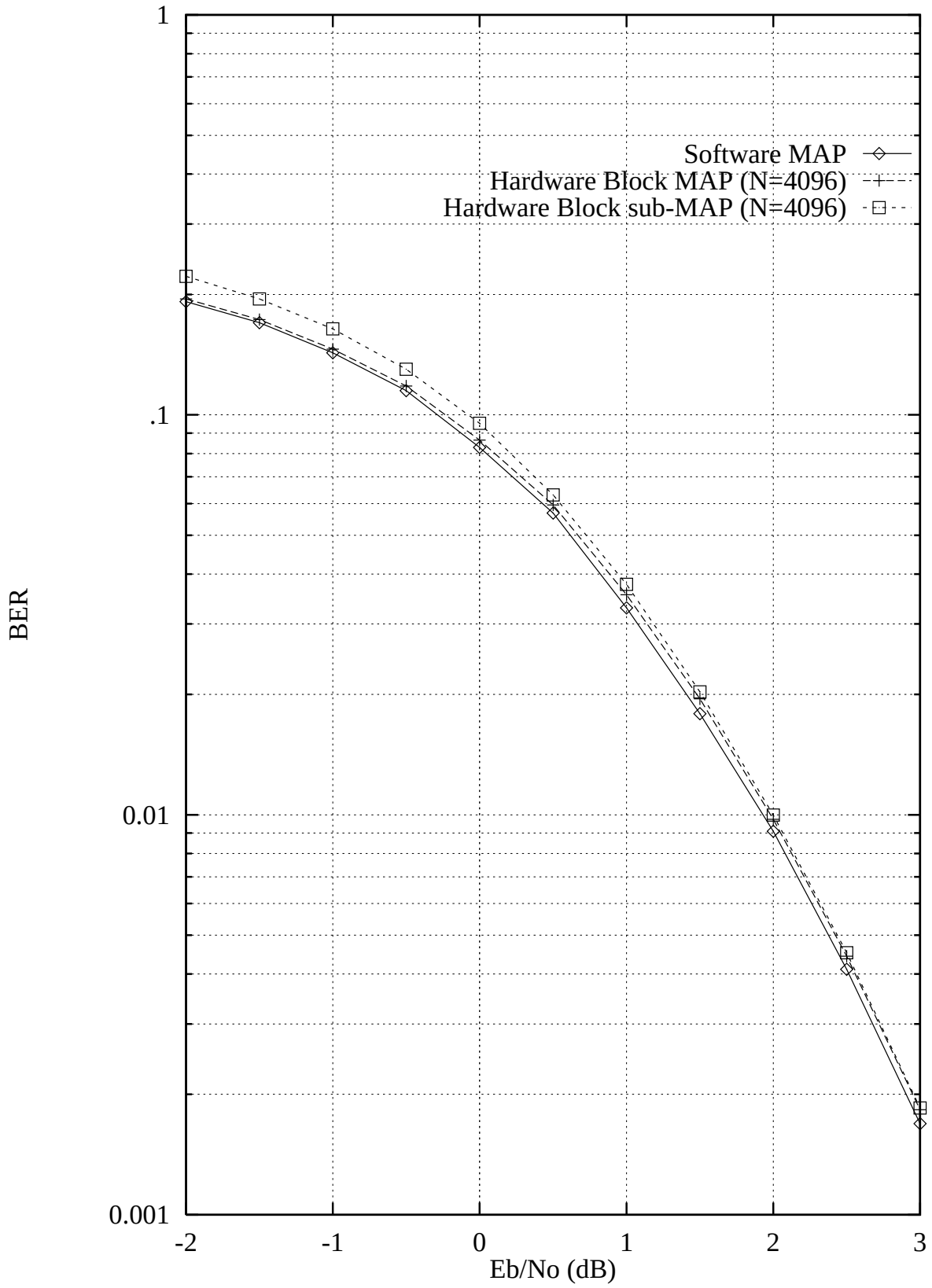

Figure 13: Systematic rate 1/2, 16 state (37,21) BER performance.

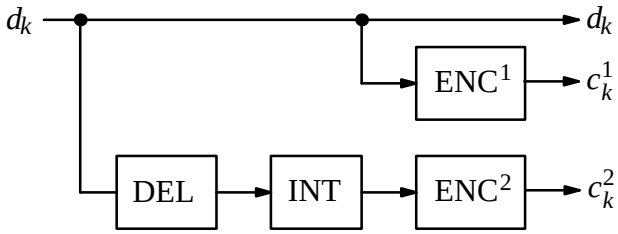

Figure 14: Rate 1/3 turbo encoder.

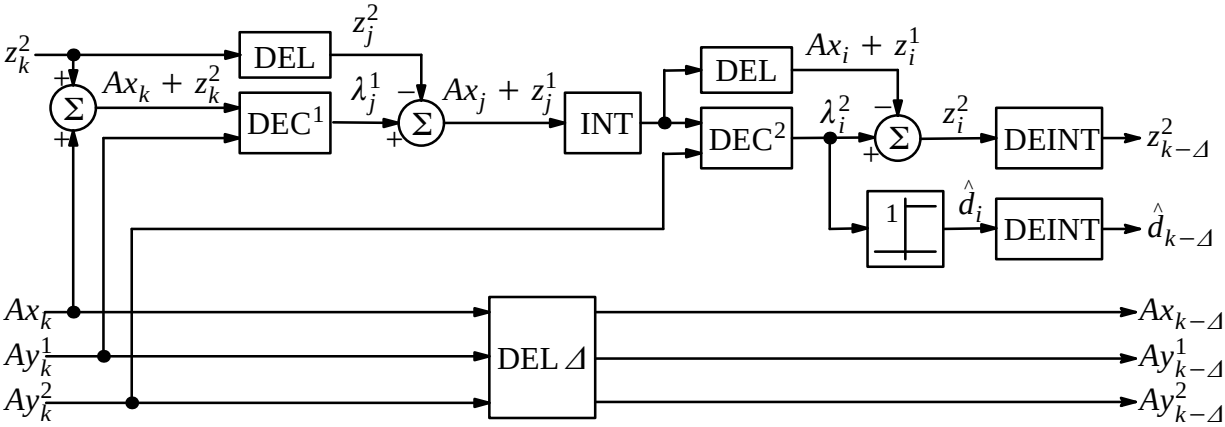

Figure 15: Iterative turbo decoder.

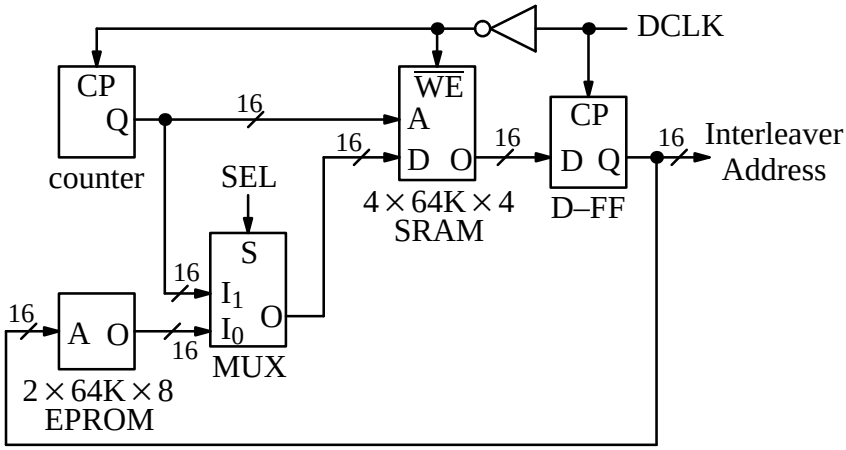

Figure 16: Interleaver address generator.

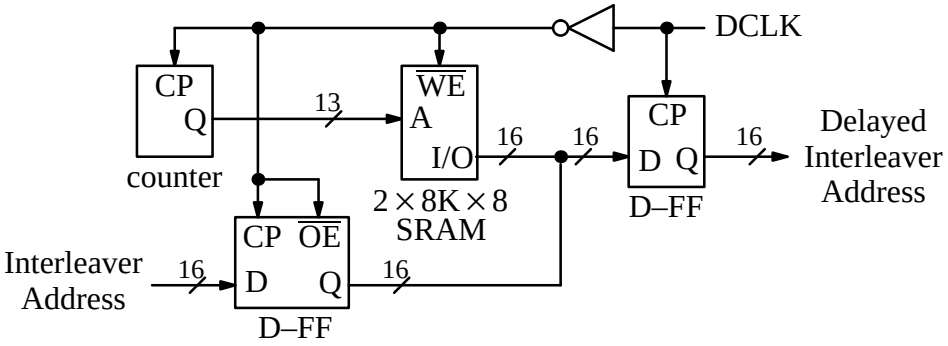

Figure 17: Interleaver address delay.

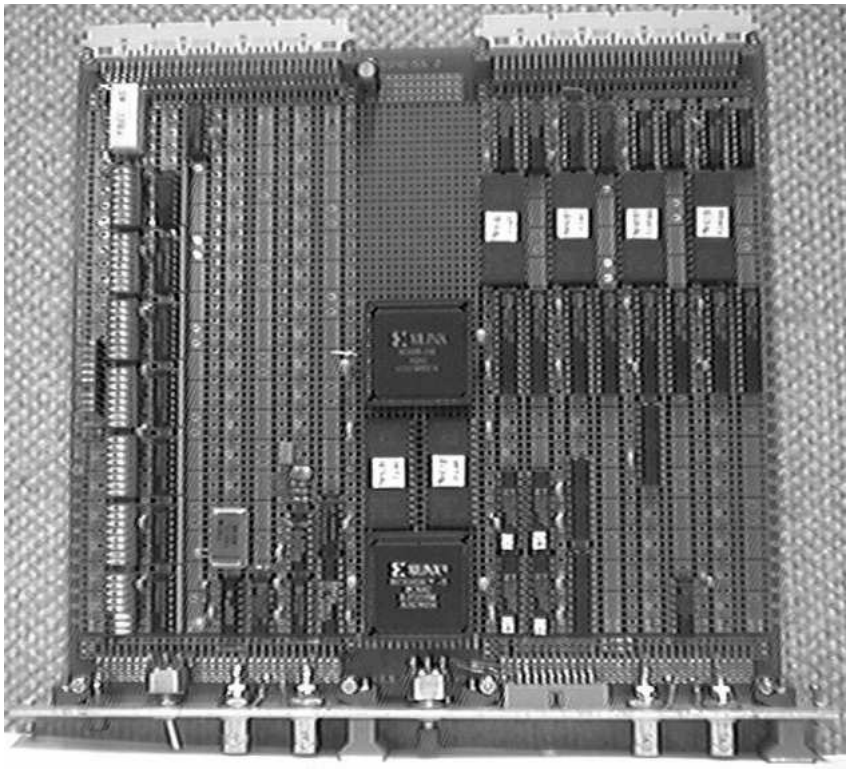

Figure 18: Turbo encoder and interleaver address generator.

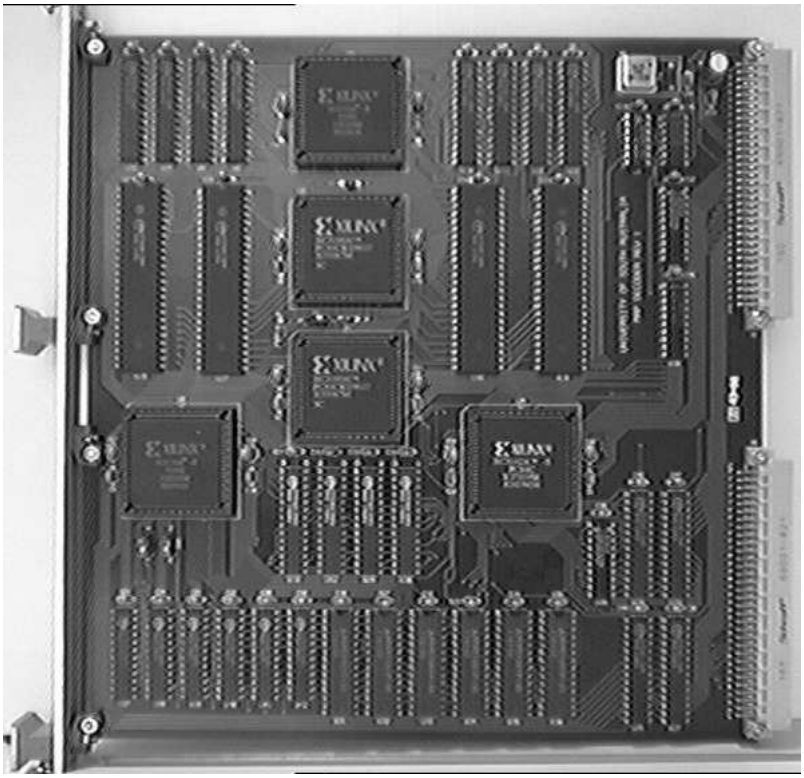

Figure 19: Turbo decoder iteration.

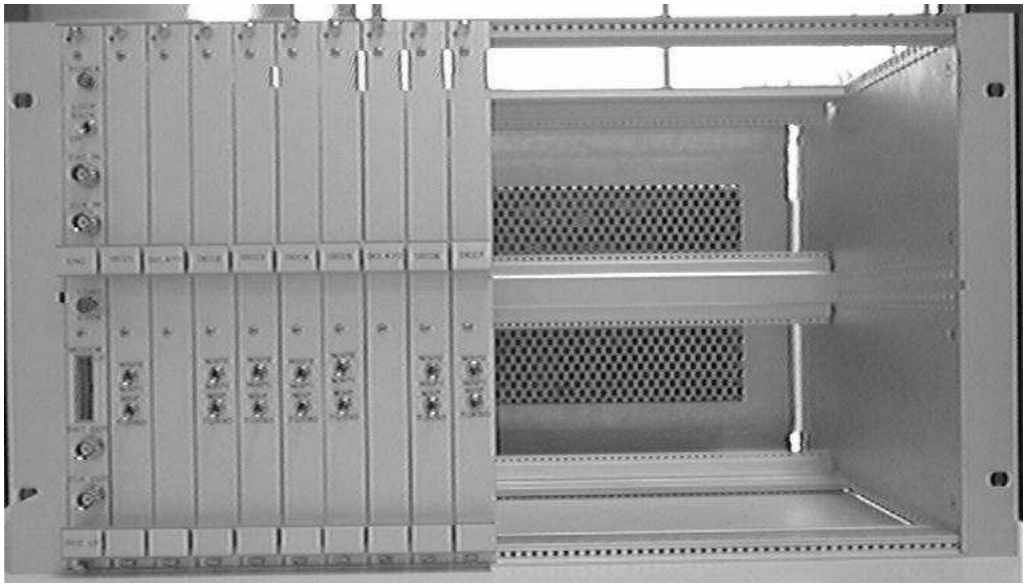

Figure 20: Turbo codec.

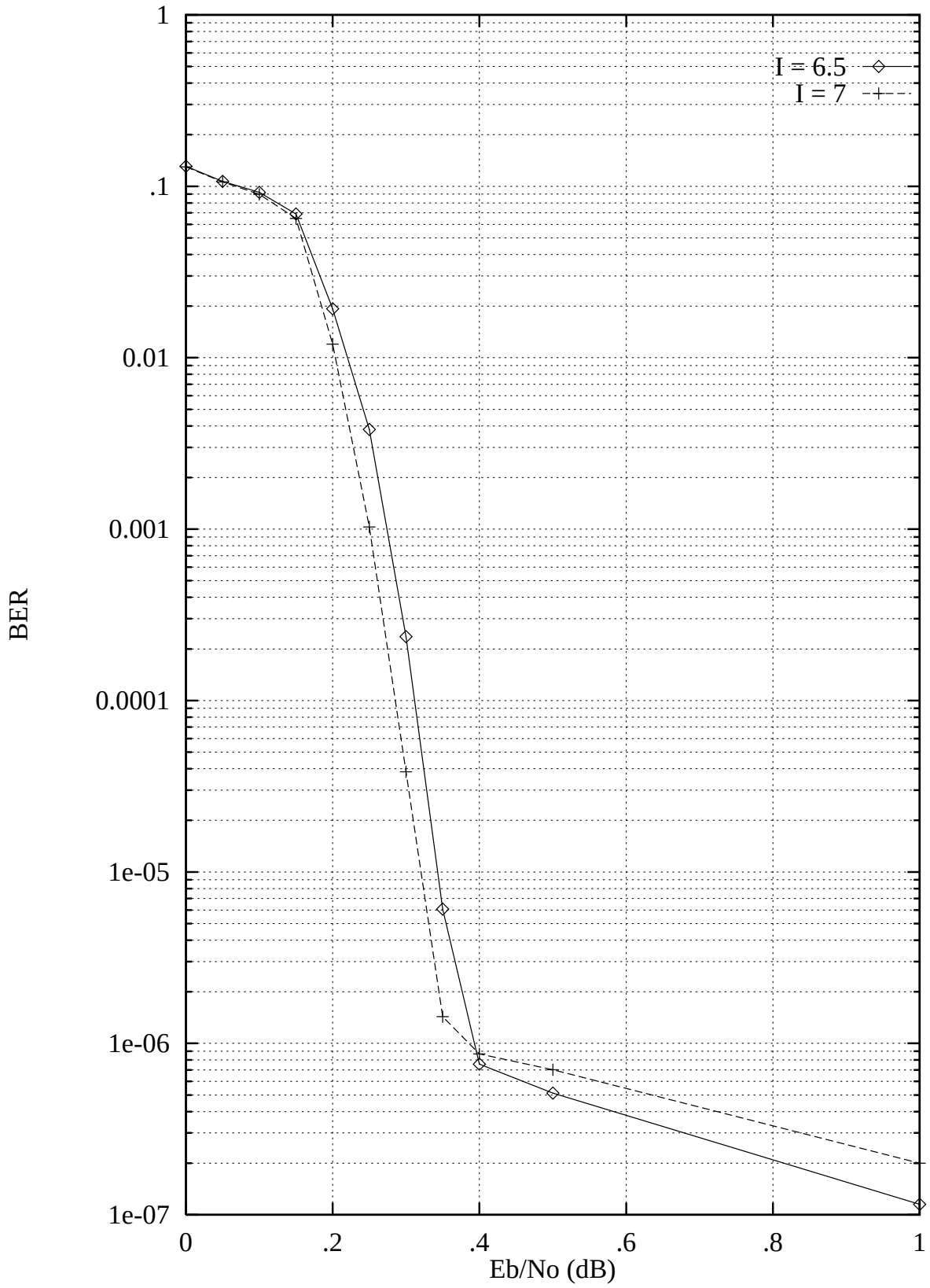

Figure 21: Rate 1/3, 16 state (31,33), N_i = 65,536 turbo decoder performance versus E_b/N_0.

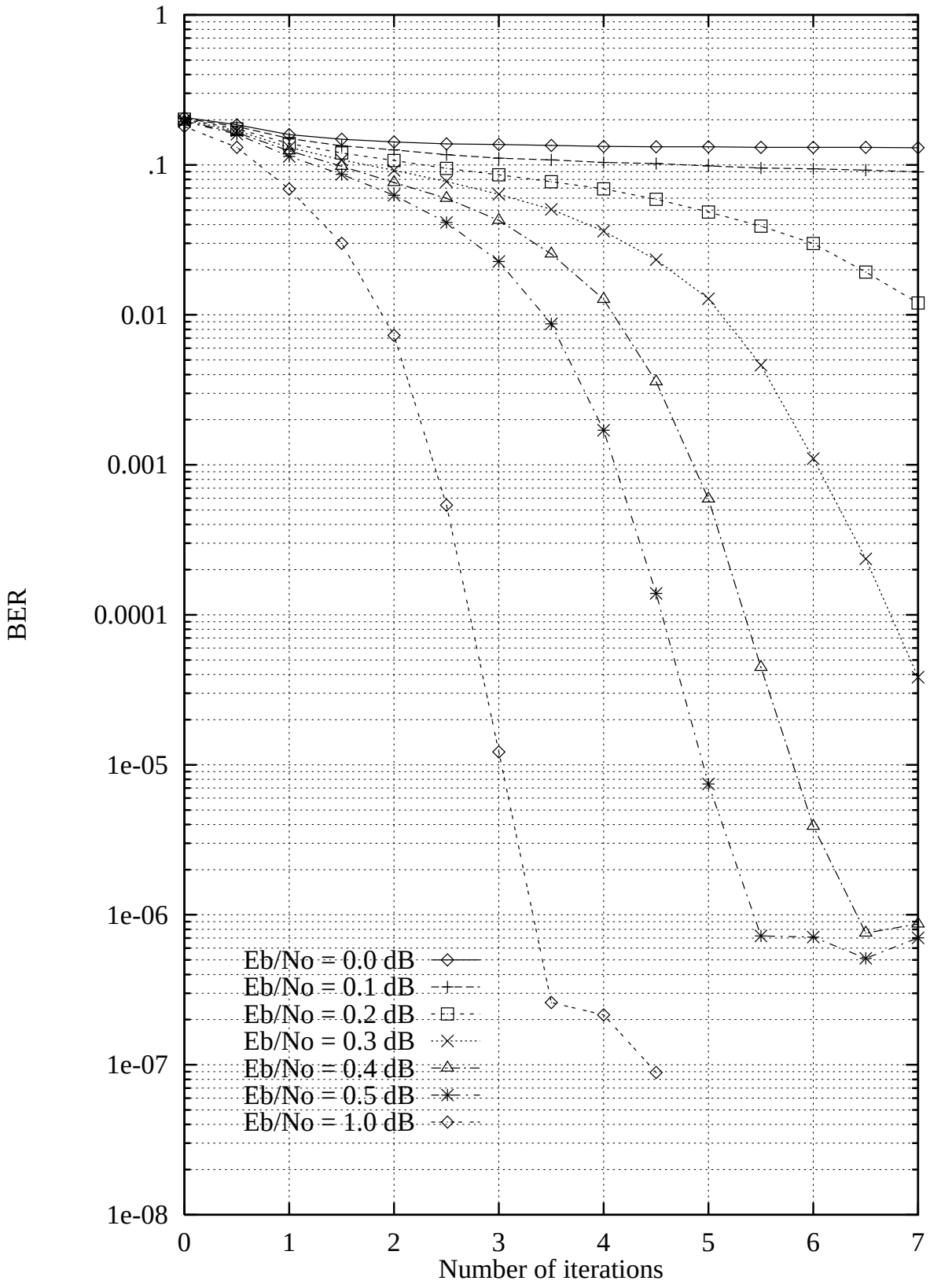

Figure 22: Rate 1/3, 16 state (31,33), N_i = 65,536 turbo decoder performance versus I.

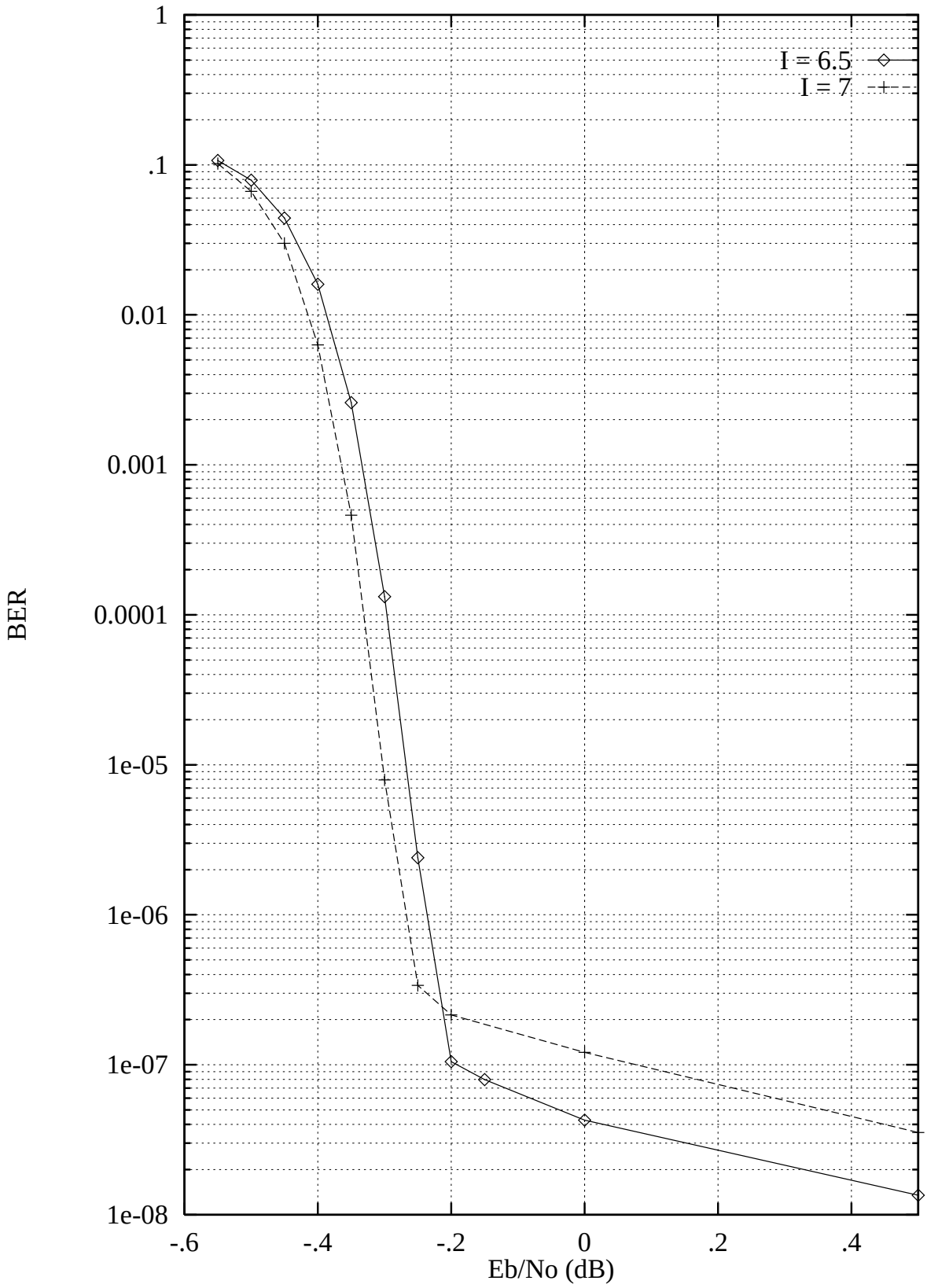

Figure 23: Rate 1/7, 16 state, N_i = 65,536 turbo decoder performance versus E_b/N_0.

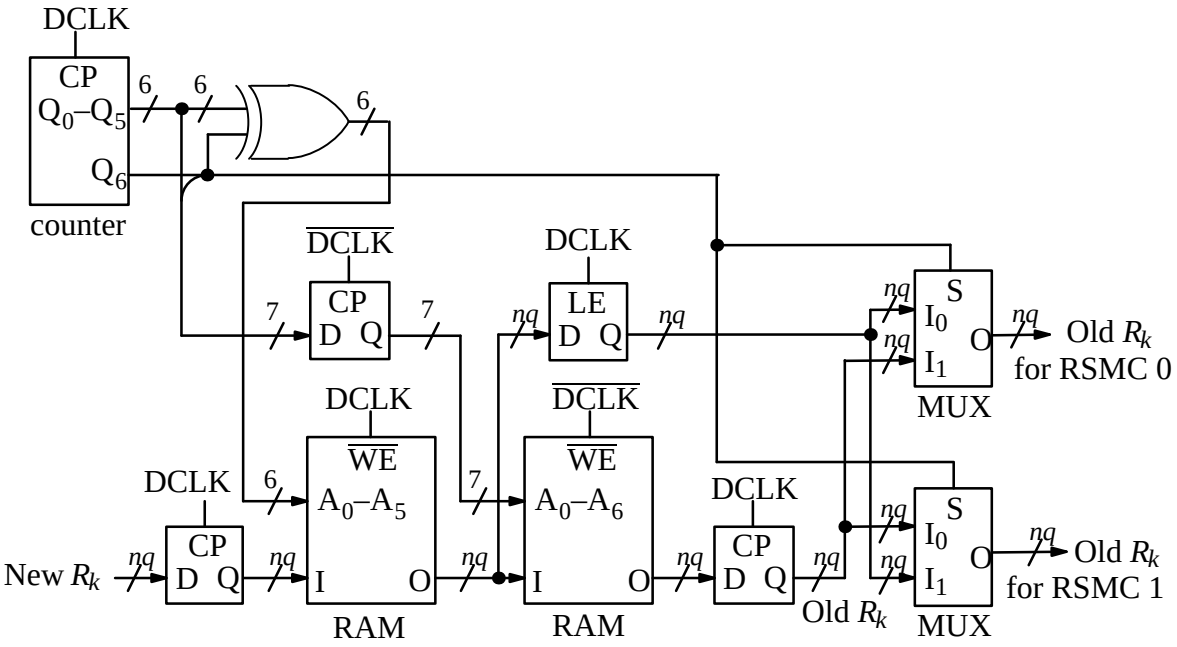

Figure 24: Storage of R_k for reverse SM calculators.