

SANDIA REPORT

SAND2003-3962
Unlimited Release
Printed March 2004

Novel Methods for Ultra-compact Ultra-low-power Communications

Richard C. Ormesher
J. Jeff Mason
Vivian Guzman Kammler

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.doe.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



SAND2003-3962
Unlimited Release
Printed March 2004

Novel Methods for Ultra-compact Ultra-low-power Communications

LDRD Final Report

Richard C. Ormesher and Jeff Mason
Radar and Signal Analysis Department

Vivian Guzman Kammler
Digital Signal Processors Department

Abstract

This report describes a new algorithm for the joint estimation of carrier phase, symbol timing and data in a Turbo coded phase shift keyed (PSK) digital communications system. Jointly estimating phase, timing and data can give processing gains of several dB over conventional processing, which consists of joint estimation of carrier phase and symbol timing followed by estimation of the Turbo-coded data. The new joint estimator allows delay and phase locked loops (DLL/PLL) to work at lower bit energies where Turbo codes are most useful. Performance results of software simulations and of a field test are given, as are details of a field programmable gate array (FPGA) implementation that is currently in design.

This page intentionally left blank.

Contents

1	Introduction	7
2	Turbo decoding with the Integrated Viterbi Algorithm.....	8
3	Simulation Results	20
4	Field Test Results.....	23
5	FPGA Implementation.....	25
6	Software Implementation.....	62

This page intentionally left blank.

1 Introduction

This report describes a new algorithm for jointly tracking and decoding convolutionally encoded error control channel symbols in a phase-shift keyed (PSK) digital communications receiver. The new symbol tracking Viterbi decoder is put to use in a Turbo decoder, which has two component Viterbi decoders. We discuss our experiences in using this Turbo decoder in a direct-sequence spread-spectrum (DSSS) receiver, although the spreading is not essential to the new algorithms.

The new approach incorporates both symbol time and phase estimation within the Viterbi decoding process. We call this new approach the Integrated Viterbi Algorithm (IVA). The key idea is that making data decisions inside the Viterbi decoding process improves the performance of the time and phase tracking loops, compared to existing methods which makes data decisions prior to the Viterbi decoding process. Data decisions in this context are decisions regarding occurrences of step changes in phase due to the PSK modulation. The phase modulation steps must be recognized and removed from the phase measurements, which are then smoothed by the phase tracking, or phase locked loop (PLL), in order to produce estimates of underlying carrier phase. The mechanism of the improved decisions is through the correlation among adjacent convolutionally-encoded symbols. Within the Viterbi decoder a PSK decision can be based not solely on the measured phase of the symbol in question, but also on the phases and decisions regarding all other symbols on the path selected during the Viterbi decoding process.

Next, we show that the IVA can be incorporated into a Turbo decoder resulting in a tracking Turbo decoder that can operate at lower signal-to-noise ratio (SNR) than that of a delay and phase locked loop (DLL/PLL) using conventional data decisions. The proposed Integrated Turbo Algorithm (ITA) consists of the parallel concatenation of an IVA decoder and a standard soft-output Viterbi algorithm (SOVA) decoder. The IVA estimates timing and phase, performs the PSK symbol detection, including despreading in the DSSS system, and generates the soft PSK symbol values for input to the following standard SOVA decoder.

2 Turbo decoding with the Integrated Viterbi Algorithm

In this chapter we develop a Viterbi decoder with integrated (internal) delay and phase locked loops and then show how this Viterbi decoder can be used to make a Turbo decoder with integrated delay and phase tracking loops.

We assume that the incoming message consists of a preamble or acquisition sequence followed by a block of coded channel symbols that have been spread with a direct sequence spread spectrum (DSSS) code. The acquisition preamble's SNR is large enough to provide an estimate of the initial carrier frequency, phase, and symbol timing. We assume that the signal has both symbol timing and carrier frequency drift, due to clock errors and relative motion between the receiver and transmitter.

2.1 Received Signal Model

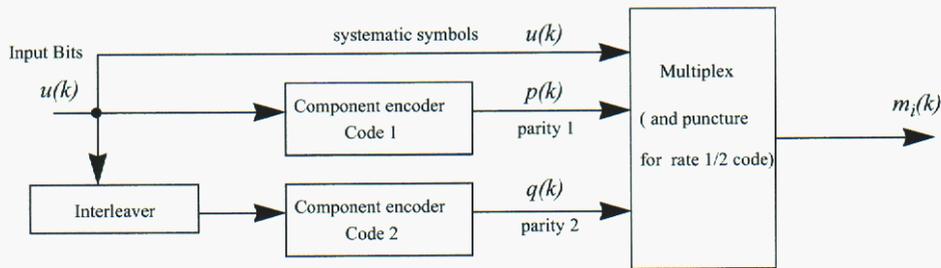
A sample at time t_n of a DSSS binary phase-shift keyed (BPSK) communications signal transmitted over an additive white Gaussian noise channel has the following complex form

$$y(t_n) = \sqrt{\frac{E_s}{T_s}} c_i(k, t_n - \tau(t_n)) m_i(k) e^{j\varphi(t_n)} + n(t_n) \quad (1)$$

where E_s denotes the constant symbol energy, $c_i(k, t)$ is the bipolar spreading function of time t for the k^{th} data bit, i is the symbol index for the k^{th} data bit and consist of the values $\{1,2\}$ for a 1/2 rate code, or $\{1,2,3\}$ for a rate 1/3 code, for example. T_s is the data symbol interval, $m_i(k)$ is the sequence of data symbols from the channel encoder output (see Figure 1) and for BPSK modulation takes on values of ± 1 , $\tau(t)$ is an unknown time-varying time delay, $\varphi(t)$ is an unknown time-varying carrier phase, $n(t)$ is zero mean complex Gaussian noise with variance $\sigma_n^2 = N_0/T_{ad}$ and n is the time sample index. The received signal is sampled such that

$$t_n - t_{n-1} = T_{ad} \quad (2)$$

where T_{ad} is the analog-to-digital converter sample interval.



Multiplexer output:

rate 1/2: $m_i(k) = \{m_1(k), m_2(k)\} = u(1), p(1), u(2), q(2), u(3), p(3), u(4), q(4) \dots$

rate 1/3: $m_i(k) = \{m_1(k), m_2(k), m_3(k)\} = u(1), p(1), q(1), u(2), p(2), q(2), u(3), p(3), q(3) \dots$

Figure 1. Turbo channel encoder

2.2 Conventional Time and Phase Synchronization

Coherent PSK communications require that the transmitter and receiver waveforms be synchronized. As mentioned above, the received signal (1) contains both an unknown timing term, $\tau(t)$, and phase term, $\varphi(t)$. These unknown terms are due to transmitter and receiver clock errors and RF channel dynamics. The receiver, therefore, needs to estimate and remove these unknown time and phase terms prior to despreading and detecting the channel symbols.

The conventional coherent demodulation approach, shown in Figure 2, is to employ phase and delay locked loops (PLL/DLL) to track and remove unknown phase and time terms prior to channel decoding. The weakness in this approach is that the data-aided loops are making hard channel symbol decisions without regarding the information imparted on the surrounding data symbols by the channel coding.

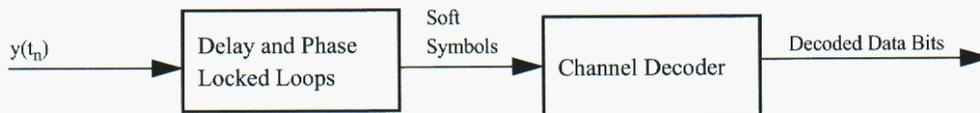


Figure 2. Conventional phase and delay locked loops precede the channel decoder.

Figure 3 is a block diagram for a conventional coherent data-aided Delay and Phase Locked Loop (DPLL). The input data stream, $y(t_n)$, is assumed to be a sampled complex signal defined by (1). The time tracking and phase tracking loops are implemented in parallel. The DLL consists of a pair of early and late correlators to track bit timing and despread the DSSS modulation. The data-aided PLL is implemented digitally with a Numerically Controlled Oscillator (NCO) and hard-symbol detector. The PLL performs the carrier phase tracking required to remove the unknown phase term,

$\varphi(t_n)$. The output of the middle correlator is the complex value of the despread symbol and the phase of this term is corrected by 0 or π radians according to the sign of the detected soft symbol (i.e., a data-aided loop). The phase corrected middle correlator output is fed into the loop filters. In Figure 3 we use the parameter l to indicate the index of a data symbol. Dropping the subscript on $m_i(k)$ indicates the alternate indexing scheme such that $m(l) = m(k + i/R)$ for a rate R code. We use this symbol index to illustrate how data symbols are processed in the conventional PDLL approach.

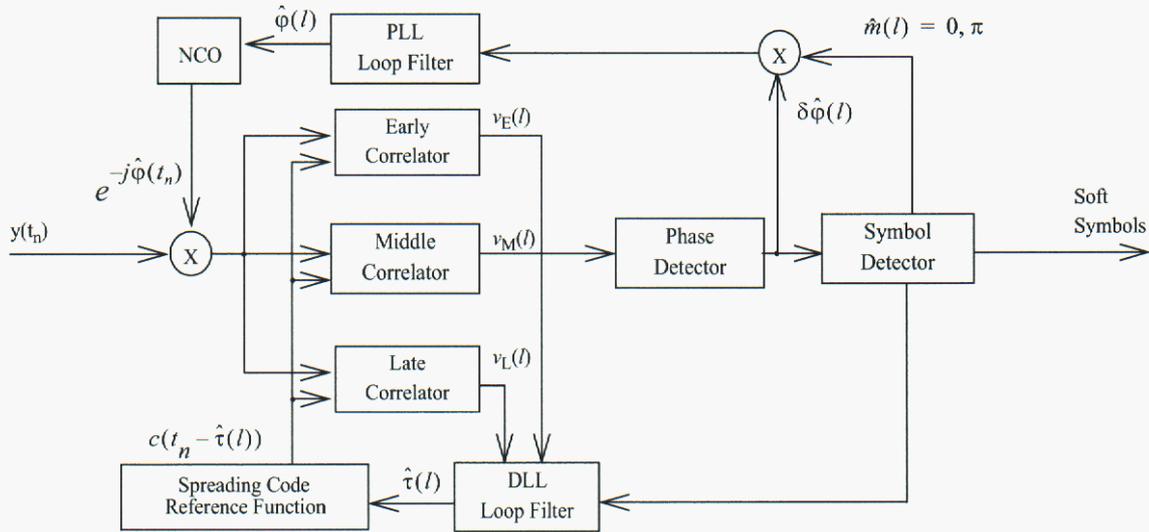


Figure 3. Conventional phase and delay locked loops

The equations for the correlators and phase detector are given as follows:

$$v_E(l) = \sum_{n=l \cdot N_c}^{n=l \cdot N_c + N_c - 1} y_{ref}(t_n + \hat{\tau}(t_n) + \delta T_c) y(t_n) e^{-j\hat{\phi}(t_n)} \quad (3)$$

$$v_M(l) = \sum_{n=l \cdot N_c}^{n=l \cdot N_c + N_c - 1} y_{ref}(t_n + \hat{\tau}(t_n)) y(t_n) e^{-j\hat{\phi}(t_n)} \quad (4)$$

$$v_L(l) = \sum_{n=l \cdot N_c}^{n=l \cdot N_c + N_c - 1} y_{ref}(t_n + \hat{\tau}(t_n) - \delta T_c) y(t_n) e^{-j\hat{\phi}(t_n)} \quad (5)$$

$$\delta \hat{\phi}(l) = \text{atan2} \left(\frac{\text{imag}(v_M(l))}{\text{real}(v_M(l))} \right) \quad (6)$$

$$\delta \hat{\tau}(l) = f(v_E(l) - v_L(l)) \quad (7)$$

where $y(n)$ is the sampled input signal; y_{ref} is the reference signal used to despread the input signal, v_E , v_M , and v_L are the integrate and dump outputs of the early, middle and late correlators, respectively; $\delta \hat{\phi}(l)$ is the instantaneous phase estimate of the input signal and is obtained from the output of the middle correlator, $\hat{\phi}(l)$ is the filtered phase estimate, $\delta \hat{\tau}(l)$ is the instantaneous time delay estimate of the input signal and is obtained from the output of the early and late correlators, v_E and v_L , $\hat{\tau}(l)$ is the filtered time delay estimate.

The receiver shown above is described in detail in the open literature [3]. Several observations, however, motivate us to consider a new approach for data-aided time and phase estimation:

- 1) Turbo codes can operate with very low symbol energy to noise power density ratios values ($E_s/N_0 < 0$ dB)
- 2) Conventional coherent data-aided DPLLs do not perform very well at these low values of E_s/N_0
- 3) The performance of the DPLL will improve if the SNR at the correlator output is increased from E_s/N_0 by removing the data modulation and coherently integrating over more than one symbol.

The first two observations result in conflicting requirements. To take advantage of turbo encoding we desire to operate at very low signal-to-noise ratios; however; the conventional DPLLs do not perform very well at these levels of signal-to-noise ratios. As suggested in observation three, this issue can be resolved if we can increase the coherent integration time of the early, middle, and late correlators. However, we cannot increase the coherent integration interval without first removing the unknown data symbol modulation.

In the following, we introduce a novel approach to solve this problem. Our approach integrates the DPLL into the Turbo decoder algorithm. The approach we present uses the estimated symbols from within the decoding process to remove a set of one or more unknown data symbols prior to the correlators (i.e., integrate and dump operation) used in the standard DPLL method.

This new approach results in two improvements relative to the standard DPLL approach. First, using the data symbol estimates from within the data decoder provides a better symbol estimator than the simple hard symbol detector used over a single data symbol as done in the standard DPLL. By using the data symbol estimates from within the decoder we are taking advantage of the information imparted on adjacent symbols by the channel encoder. This results in fewer phase modulation removal errors in the data-aided loop. Second, in the standard DPLL only a single symbol is detected and removed at a time and this limits the coherent integration time to a single symbol. With the new approach, we can estimate and integrate over several data symbols at a time as explained in section 2.4. This increases the coherent integration time and results in an increased signal to noise ratio at the output of the DPLL correlators. The end result is improved estimates of the instantaneous phase and time delay terms which allows the DPLL to work at lower E_s/N_0 .

2.3 Integrated Turbo Algorithm

A block diagram for a Turbo decoder that simultaneously despreads channel symbols, decodes channel symbols, and tracks timing and phase is shown in Figure 4. Similar to the standard Turbo decoder, two component decoders are linked together by an interleaver, a de-interleaver, and the *a priori* information that is passed between them. However, there are several differences between the new Turbo decoder, herein called the Integrated Turbo Algorithm (ITA), and the standard Turbo decoder.

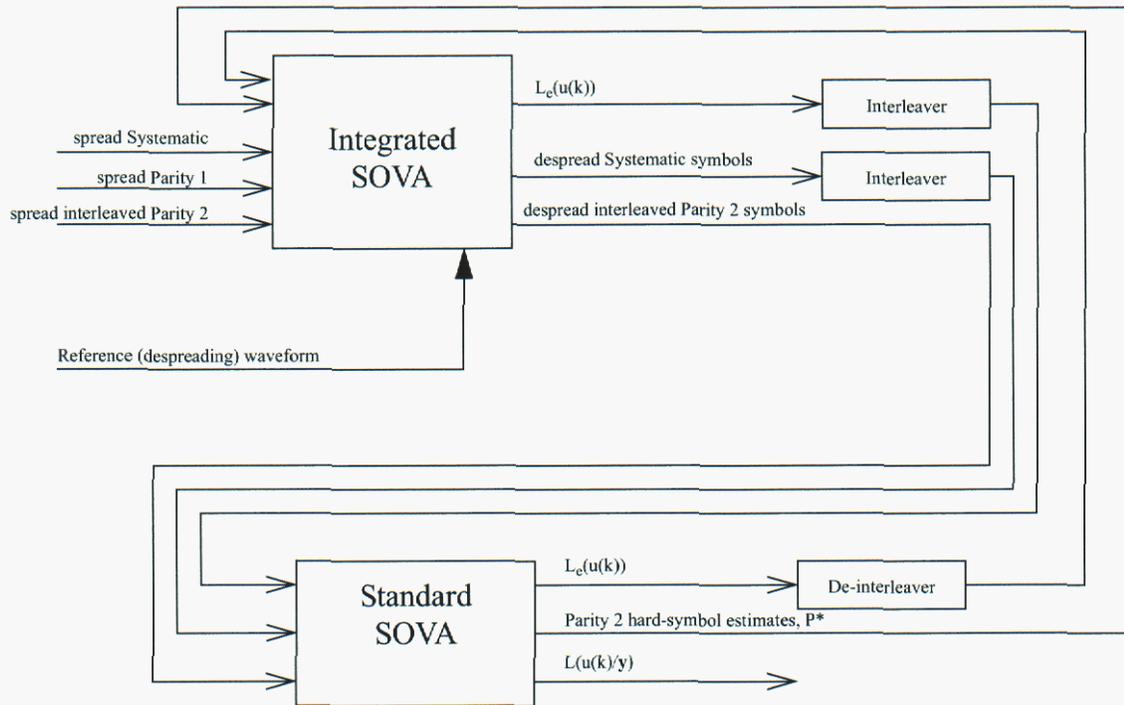


Figure 4. Integrated Turbo decoder block diagram

The first component decoder of the ITA is the Integrated Viterbi Algorithm (IVA). This is a Soft Output Viterbi Algorithm (SOVA) decoder that has been modified such that the time and phase tracking are implemented simultaneously within the decoding processes. The input into the IVA is the sampled complex received signal as defined in Eqn. (1). The input signal has a spreading sequence applied and has an unknown time-varying phase and time-delay. In the block diagram in Figure 4, we show the input signal partitioned into systematic and parity symbols. The operations required to despread the symbol and estimate and remove the unknown time and phase terms are implemented as an integral operation within the first component decoder.

The basic approach is to use a data-aided coherent Delay and Phase Lock Loop (DPLL) that is executed simultaneously during the Viterbi decoding process. We perform joint phase and time estimation on a spread spectrum BPSK signal. The output of the IVA is a set of despread soft symbols and the normal extrinsic information, $L_e(u_k)$, related to the information bit $u(k)$. The despread soft symbols are the set of soft symbols that would typically come from the output of a conventional DPLL preceding the channel decoder.

Because of interleaving, joint phase and time estimation, and data despreading can only be implemented within the first component decoder. This is because the input systematic symbols for the second component decoder are interleaved after transmission through the RF channel to match the order of parity symbols for the second component decoder which were interleaved before transmission as shown in Figure 1. This interleaving scrambles the phase on the systematic symbols making tracking impossible.

The output from the second component decoder consists of the extrinsic information, $L_e(u_k)$, related to the information bit, $u(k)$, the *a posteriori* Log-Likelihood Ratio (LLR) for the information bit $L(u(k)|y)$, and the hard-symbols estimates for the parity symbols from the second component decoder. The hard detected parity symbols are used in the DPLL of the first component decoder. In the next section, we describe the Integrated Viterbi Algorithm in more detail.

2.4 Integrated Viterbi Algorithm

In this section we describe how the DPLL is integrated into the Viterbi Algorithm. Recall that the Viterbi algorithm searches all possible paths in its associated trellis. The codeword (i.e. encoder output) is chosen with the smallest distance between the received symbol sequence and all possible codewords.

The decoding algorithm starts at the first received symbol and continues until the last received symbol is processed. At each bit interval, the path is reduced by deselecting codewords from all the possible remaining codewords. At each trellis two paths merge into a single state and the path with the smallest metric is eliminated as the optimal path leaving a single surviving path for each state. When the end of the trellis is reached the surviving path with the largest path metric is selected as the most likely path. At any bit interval within the decoding process there are 2^{K-1} possible surviving paths where K is the constraint length of the encoder and each of these paths has a unique associated codeword. Our goal is to use these codewords to implement the data-aided loop described above. Since there are 2^{K-1} possible surviving paths we shall implement a separate DPLL for each of these paths during the decoding process.

Consider the correlators defined by Eqn. (3) through Eqn. (5) above that are used to estimate instantaneous time delay, $\delta\hat{\tau}(l)$, and phase, $\delta\hat{\phi}(l)$, of the l^{th} symbol, $m(l)$ within the conventional data-aided DPLL. The signal outputs $v_E(l)$ and $v_L(l)$ are used in the delay computer to track and time align the input signal to a reference signal, while output $v_M(l)$ is used to estimate the phase of the current symbol. In each case, the correlators use the detected hard-symbol $\hat{m}(l)$ to remove the data modulation before the symbol phase and time estimate are made (i.e, a data-aided loop). In the loop shown in Figure 3 above, we see that the hard-symbol detection occurs after the basic integrate and dump operation from the middle correlator.

As previously mentioned, our goal is to improve the SNR at the output of the correlators, improve the instantaneous estimates of both symbol time and phase, and thereby, improve the overall performance of the DPLL. By examination of Eqn. (3) through Eqn. (5) we see that if the symbol values, $\hat{m}(l)$ are known *a priori* and time and phase are constant over several symbols then the correlators can integrate over several symbols before estimating time and phase. Define L as the number of bit intervals in which time and phase are considered constant enough to allow for coherent integration. Then, for a 1/2 rate component code, at any bit interval in the Viterbi

decoding process there are a set of $2*L$ symbol estimates for each surviving path. These symbols can be used to remove data symbol modulation and increase the correlator integration interval in a data-aided DPLL. One of the surviving paths will eventually be selected has the most likely path.

The following equations are Eqn. (3) through Eqn. (5) modified to implement coherent integration over a block of L Viterbi decisions for a 1/2 rate punctured Turbo code.

$$v_E(k, s_k) = \sum_{n=2kN_c}^{n=2kN_c+N_c-1} \hat{m}_1(k, s_k)y(t_n)e^{-j\hat{\phi}_{s_k}}y_{ref}(t_n + \hat{\tau}_{s_k} + \delta) + \sum_{n=2kN_c+2N_c-1}^{n=2kN_c+N_c} \hat{m}_2(k, s_k)y(t_n)e^{-j\hat{\phi}_{s_k}}y_{ref}(t_n + \hat{\tau}_{s_k} + \delta) + v_E(k, s_{k-1}) \quad (8)$$

$$v_M(k, s_k) = \sum_{n=2kN_c}^{n=2kN_c+N_c-1} \hat{m}_1(k, s_k)y(t_n)e^{-j\hat{\phi}_{s_k}}y_{ref}(t_n + \hat{\tau}_{s_k}) + \sum_{n=2kN_c+2N_c-1}^{n=2kN_c+N_c} \hat{m}_2(k, s_k)y(t_n)e^{-j\hat{\phi}_{s_k}}y_{ref}(t_n + \hat{\tau}_{s_k}) + v_M(k, s_{k-1}) \quad (9)$$

$$v_L(k, s_k) = \sum_{n=2kN_c}^{n=2kN_c+N_c-1} \hat{m}_1(k, s_k)y(t_n)e^{-j\hat{\phi}_{s_k}}y_{ref}(t_n + \hat{\tau}_{s_k} - \delta) + \sum_{n=2kN_c+2N_c-1}^{n=2kN_c+N_c} \hat{m}_2(k, s_k)y(t_n)e^{-j\hat{\phi}_{s_k}}y_{ref}(t_n + \hat{\tau}_{s_k} - \delta) + v_L(k, s_{k-1}) \quad (10)$$

$$\delta\hat{\phi}_{s_k} = \text{atan}\left(\frac{\text{imag}(v_M(k, s_k))}{\text{real}(v_M(k, s_k))}\right) \quad (11)$$

$$\delta\hat{\tau}_{s_k} = f(v_E(k, s_k) - v_L(k, s_k)) \quad (12)$$

In Eqn. (8) to Eqn. (10) s_k indicates the current state at the k^{th} bit interval in the Viterbi decoding process, s_{k-1} indicates the previous state for the selected path and N_c is the number of PN code chips per data symbol. In the 1/2 rate Turbo decoder $\hat{m}_i(k, s_k)$ is the symbol estimate at the k^{th} bit for state s_k while $i=1$ for the systematic symbol and $i=2$ for the parity symbol. Notice, that for a 1/2 rate encoder the correlators sum over two data symbols per decoding interval and will sum over a total of $2*L$ data symbols before they are dumped and phase and time estimates are updated. Also, notice that there is a unique DPLL implemented for each surviving path of the trellis. Eqn. (12) needs to use a two argument arctangent function whose range is $-\pi$ to π .

To illustrate the use of Eqn. (8) to Eqn. (12) assume that we are at the k^{th} bit interval in the Integrated Viterbi Algorithm. Then, at each state, s_k , there is a surviving path an associated Viterbi path metric (see Figure 5). In addition, each surviving path has a phase and time delay estimate, $\hat{\tau}_{s_k}$, and $\hat{\varphi}_{s_k}$, respectively. These are the time and phase terms that are estimated and tracked via the DPLL and are updated every L bit intervals.

For the correlators, we desire to sum the next $2*L$ data symbols using the decision sequence $\hat{m}_i(k, s_k), \dots, \hat{m}_i(k+L, s_{k+L})$ to remove the data modulation. Because we are in the middle of the Viterbi decoding process, there is a decision sequence from S_k to S_{k+L} for each survivor of state, S_{k+L} , at the $k+L$ bit.

To perform the summation we initialize the terms v_E, v_M and v_L to zero (i.e, at the start of a block of $2*L$ symbols) and continue with the Viterbi decoding process and sum the terms in Eqn. (8) to Eqn. (10), for each state, over the selected path.

For example, at the next bit interval, k , for each state, s_k , we perform the following steps:

- 1) Using the current time estimate, $\hat{\tau}_{s_{k-1}}$, and phase estimate, $\hat{\varphi}_{s_{k-1}}$, we despread the sampled data symbol sequence and calculate the transition metrics as follows

$$\begin{aligned} \gamma(s_{k-1}, s_k) = & \hat{m}_1 \sum_{n=2kN_c}^{n=2kN_c+N_c-1} y(t_n)y_{ref}(t_n + \hat{\tau}_{s_{k-1}})e^{-j\hat{\varphi}_{s_{k-1}}} + \\ & \hat{m}_2 \sum_{n=2kN_c+N_c}^{n=2kN_c+2N_c-1} y(t_n)y_{ref}(t_n + \hat{\tau}_{s_{k-1}})e^{-j\hat{\varphi}_{s_{k-1}}} \end{aligned} \quad (13)$$

where \hat{m}_1 and \hat{m}_2 are the codeword symbols associated with the branch transition from s_{k-1} to s_k . For the rate 1/2 (punctured) code the parity symbol is not sent on even numbered bits, as shown in Figure 1, so the second sum in Eqn. (13) must be set to zero every other bit period. For the rate 1/3 Turbo code the parity bits are always sent so both sums are computed.

- 2) Using the transition metrics the Viterbi path metric is updated and the codeword symbols $\hat{m}_1(k, s_k)$ and $\hat{m}_2(k, s_k)$ are selected according to the normal Viterbi Algorithm. In addition, the time, $\hat{\tau}_{s_{k-1}}$, and phase estimate, $\hat{\varphi}_{s_{k-1}}$, associated with the selected transition are propagated to the next bit iteration.
- 3) Using the selected codeword symbols and propagated phase and time estimates we perform the early, late, and middle correlation operations as defined in Eqn. (8) to Eqn. (10)

- 4) Steps one through three continue until the end of the block, $k+L$, occurs. At this point in time, the integrators in Eqn. (8) to Eqn. (10) are dumped and the results are fed to the delay computer and phase detector followed by the Loop filters to produce updated phase and time estimate which become available for the next block of $2*L$ symbols.

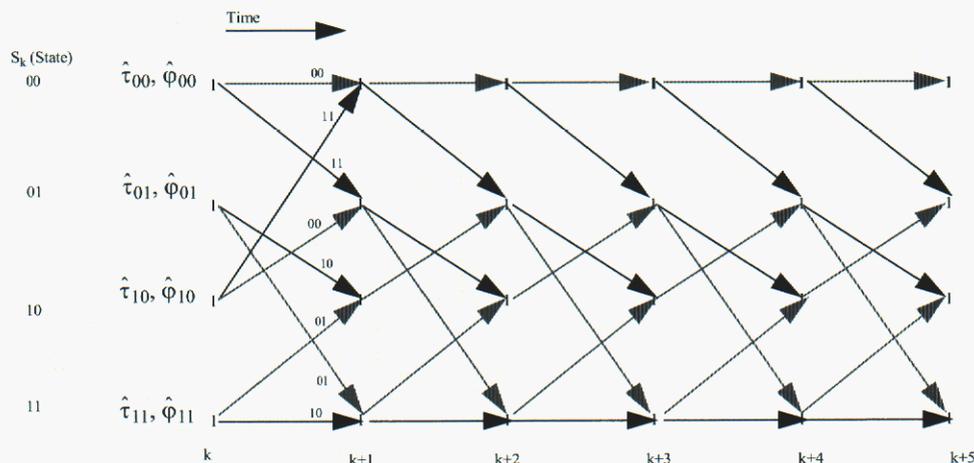


Figure 5. Example Trellis diagram for an 1/2 rate RSC code of constraint length 3

At the end of the trellis, as with the standard Viterbi Algorithm, we have a path metric for each surviving path and the ML path is the path with the largest metric. Also, since we are performing Turbo decoding, we need a soft output in the form of the *a posteriori* LLR $L(u(k)|y)$ for each decoded bit. To calculate $L(u(k)|y)$ we perform the normal trace-back operation for the Soft-Output Viterbi Algorithm (SOVA) [4].

Next, the output from the first component decoder, the IVA, is fed to the input of the second component decoder, a standard SOVA decoder. The input signal, therefore, needs to be despread prior to being used by the standard SOVA decoder. Hence, we use the IVA to provide the set of despread symbols that feed the second decoder. Note in Figure 4 that the systematic symbols are interleaved before presentation to the standard SOVA to agree with the bit order that was encoded in Figure 1. The output of the second decoder consist of the extrinsic information, $L_e(u(k))$, the *a posteriori* LLR $L(u(k)|y)$, and symbols estimates, P^* , for the second component code.

The parity symbols, P^* , shown in Figure 4, are created in the second SOVA and are used in the DPLL section of the first SOVA decoder. The use of these parity symbols are unique to the Integrated Turbo Algorithm and are used to increase to SNR at the DPLL correlators outputs. To illustrate their use, consider Eqn. (8), Eqn. (9), Eqn. (10) above for a rate 1/2 punctured Turbo code. The first term is a summation of systematic bits and the second term is a summation over the parity bits. For a rate 1/2 Turbo code the parity bits are alternately from encoder 1 and the encoder 2 as shown in Figure 1. The symbol estimates for the parity bits must be taken alternately from the two SOVA decoders. The second SOVA parity symbol estimates, P^* , are therefore used to remove the parity symbol modulation and allow for integration over all of $y(t_n)$ in Eqn. (8), Eqn. (9) and Eqn.

(10). This improves the SNR of the correlator outputs. In Viterbi metric computation Eqn. (13) for the rate 1/2 (punctured) code, the input signal $y(t_n)$ corresponding to the second component encoder parity symbols is still set to zero every other bit for.

The extrinsic information from the second SOVA decoder is deinterleaved and fed to the first component decoder. Only the second decoder works with interleaved data. As with the standard Turbo algorithm, the first component decoder now decodes the same input using the extrinsic information from the second decoder to improve the decoding process. The process is repeated until the Turbo decoder is terminated.

2.5 Chapter References

- [1] Pooi Y. Kam and Hsi C. Ho, "Viterbi Detection with Simultaneous Suboptimal Maximum Likelihood Carrier Phase Estimation," *IEEE Trans. Comm.*, Vol. 36, No. 12, pp. 1327-1330, Dec. 1988.
- [2] O. Macchi and L. Scharf, "A Dynamic Programming Algorithm for Phase Estimation and Data Decoding on Random Phase Channels," *IEEE Trans. Inform. Theory*, Vol. IT-27, No. 5, pp. 581-595, Sept. 1981
- [3] Proakis, John G., *Digital Communications*, 3rd Ed., 1995, McGraw Hill, Boston, Mass.
- [4] L. Hanzo, T. H. Liew, and B.L.Yeap, *Turbo Coding. Turbo Equalization and Space-Time Coding for Transmission over Fading Channels*, IEEE Press, 2002.

3 Simulation Results

The integrated DPLL/Turbo decoder, or ITA, has been implemented in C and exercised in a Monte Carlo simulation in order to evaluate performance in terms of bit-error rate (BER) at various levels of E_b/N_0 (bit energy to noise power density ratio).

Figure 6 below shows the observed performance of the new and prior approaches for the following set of parameters:

- $N_b = 1000$, data (message) block length in bits
- $K=5$, constraint length of the constituent RSC encoders
- $R = 1/3$, rate 1/3 (unpunctured) Turbo code

Many secondary parameters such as time and phase slew rates, loop bandwidths and spreading code and interleaver sequences are required to fully specify the simulation.

In these simulations a loop consisting of the following steps are repeated a large number of times at each E_b/N_0 .

- A random vector of 1000 data bits is generated.
- The data is Turbo encoded as shown in Figure 1 and converted to bipolar (BPSK) format.
- The Turbo channel symbols are spread using a 63 bit maximal-length PN sequence.
- The chipped data, which is over sampled, is now filtered and sampled asynchronously, approximately twice per chip, simulating the digital sampling done at a receiver having some clock rate error.
- The samples are put on a slewing RF center frequency, giving a complex signal, to which thermal noise is added to complete the simulation of the RF channel.
- The complex signal is then decoded/demodulated by the ITA and the number of bit errors is recorded.

Figure 6 shows that the proposed algorithm can provide significant performance gains over the prior approach. For example comparing the E_b/N_0 to achieve a BER of 1×10^{-5} we see that the ITA requires about 2 dB while the standard Turbo decoder following a DPLL requires a bit more than 6 dB, giving the new approach a 4 dB advantage at this operating point.

The curve for the baseband Turbo decoder is given in the plot as well. *Baseband* here means that there is no timing error and the signal phase angle remains exactly 0 so that tracking loops are unnecessary. The horizontal distance, at a given BER, between the baseband and either of the decoders with tracking loops shows the “implementation loss”, or loss of sensitivity due to imperfect tracking of the loops, for that tracking decoder. The large reduction in implementation loss of the integrated decoder under these conditions is readily apparent.

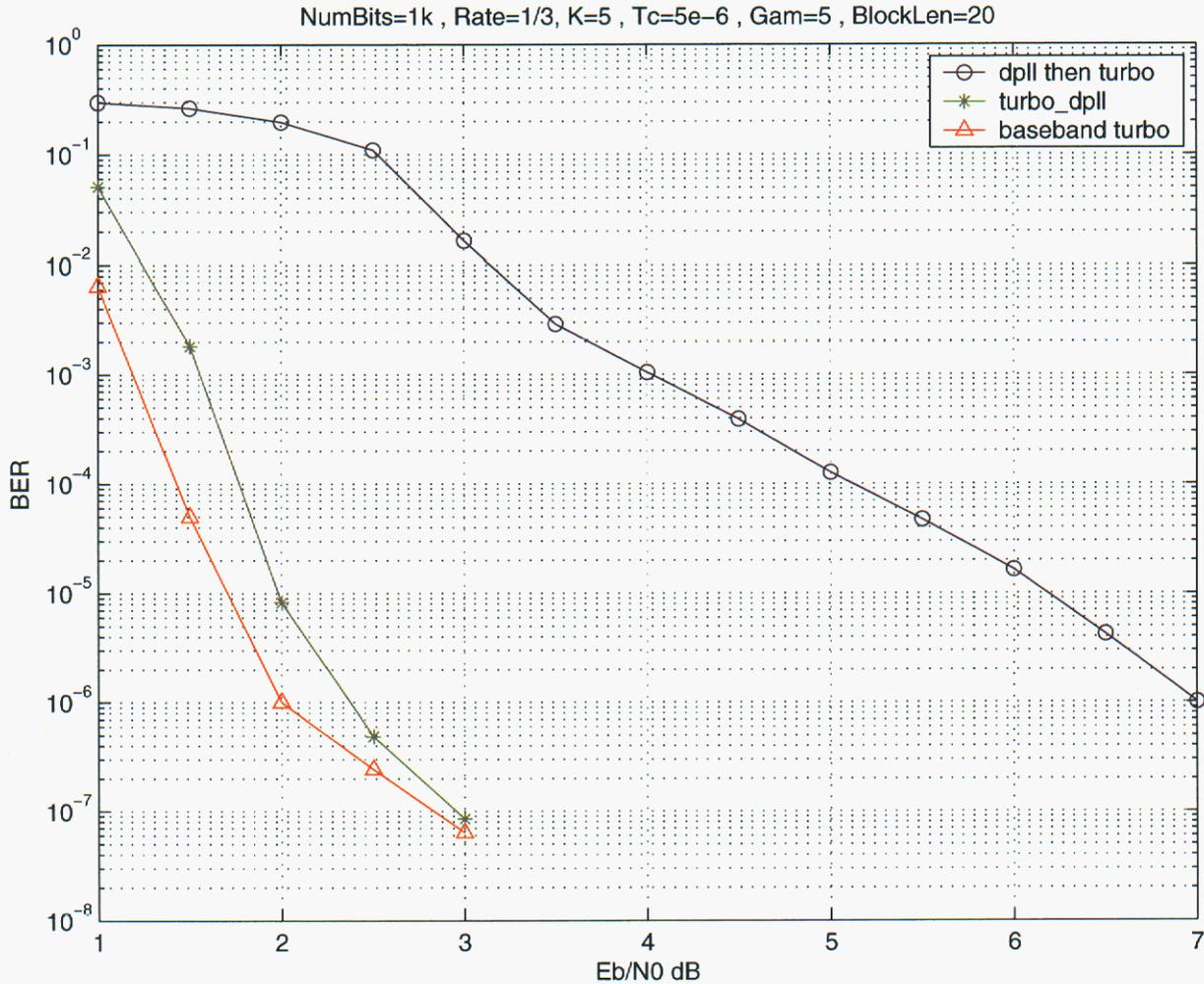


Figure 6. Comparison of performance of baseband Turbo, Integrated Turbo Algorithm, and Turbo following a DLL/PLL for a 1000-bit block, code rate 1/3, and constraint length 5.

It is instructive to see in Figure 7 and Figure 8 how baseband SOVA and Turbo decoder performance depends on two key parameters. These figures do not show performance for the Integrated Turbo Algorithm directly but recall that this would differ from the baseband Turbo performance only by the implementation loss. Comparing Figure 7 and Figure 8 you can see that while the SOVA decoder does not depend on the number of bits in the message, or interleaver block length, the Turbo performance is very strongly tied to this parameter. It is significant that performance improves strongly with block length, while processing burden is virtually unaffected. Decoding latency time increases with blocklength however, and this can become an issue in latency sensitive applications such as digital telephony.

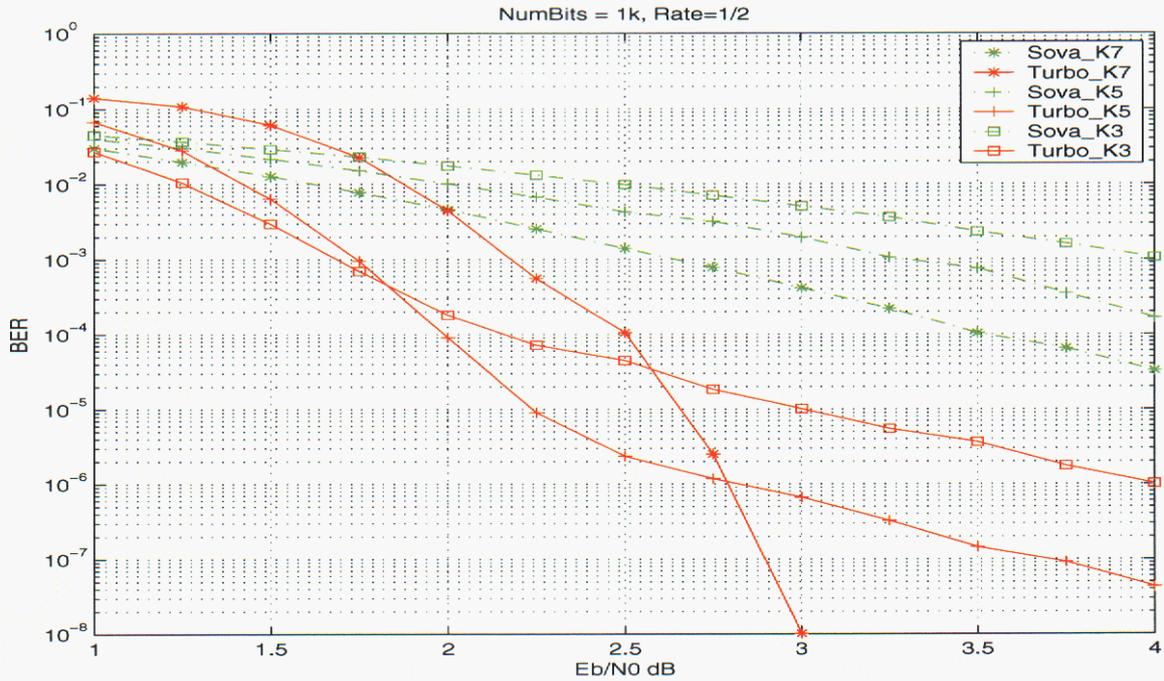


Figure 7. Comparison of performance of baseband 1,000 bit rate 1/2 SOVA and rate 1/2 (punctured) Turbo decoders for three different constraint lengths.

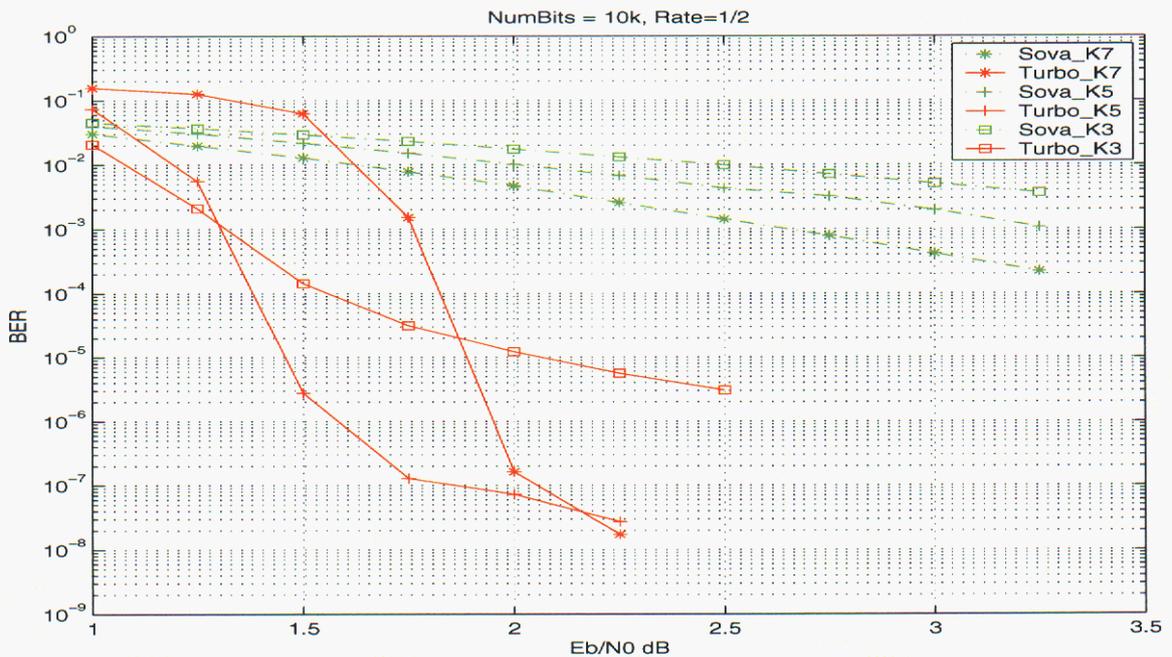


Figure 8. Comparison of performance of baseband 10,000 bit rate 1/2 SOVA and rate 1/2 (Punctured) Turbo decoders for three different constraint lengths.

4 Field Test Results

We were able to compare the performance of the Integrated Turbo Algorithm (ITA) to standard processing over an ISM band RF link using equipment assembled by the Goldmine LDRD, Project No. 26574. This link, shown in Figure 9 below, consists of an S-band transmitter, a C-band to S-band translator, or “bent-pipe”, and a C-band receiver. The transmitter and receiver were located on Sandia property on Kirtland Air Force Base while the bent pipe was carried on a gas balloon that was launched on September 29, 2003 from Ft. Summner, New Mexico and rose to an altitude of 80,000 feet.

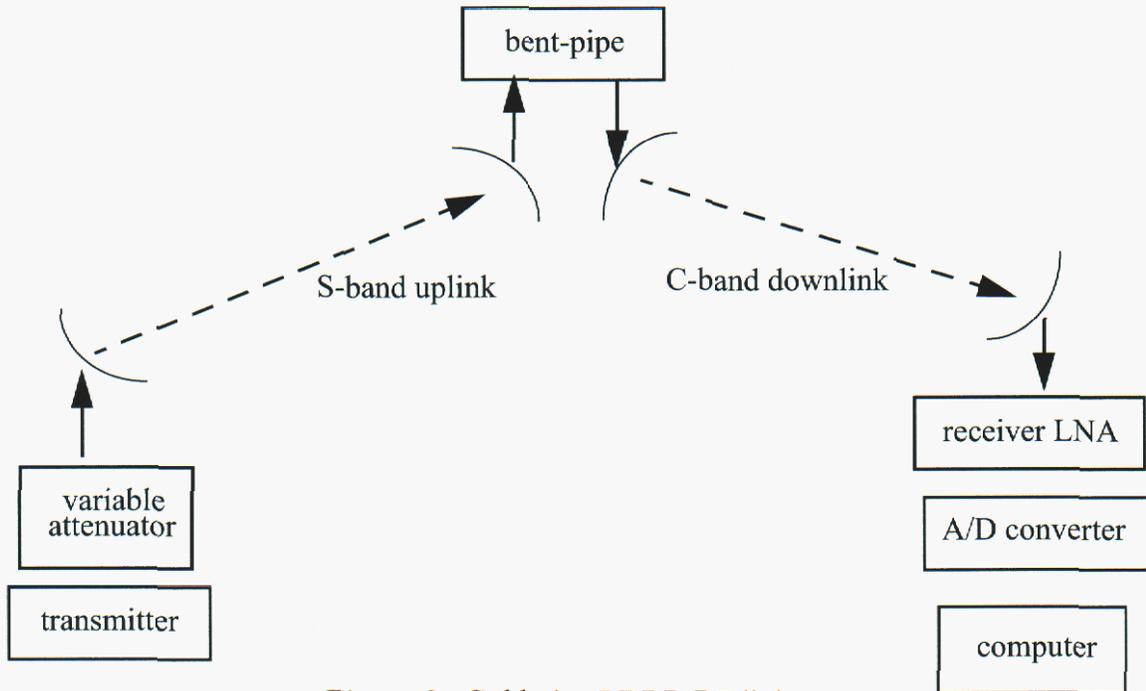


Figure 9. Goldmine LDRD RF link.

The transmitted signal was a 1000 bit message encoded by a rate 1/3 Turbo encoder, i.e. with 3 symbols per data bit, and then spread using 127 chips per symbol. This message was transmitted at 4 million chips per second (Mcps). Since there are 381 chips per bit (127 chips/symbols times 3 symbols/bit), the resulting bit rate is 4 million divided by 381 or about 10.5 kilobits per second. The duration of the 1000-bit messages is therefore just less than 0.1 second. The message was transmitted repetitively for 5 seconds at each of several transmitted power levels as controlled by the setting of a variable power attenuator inserted between transmitter and its antenna.

Columns 1 and 2 of Table 1 shows the Bit Error Rate (BER) both for the Integrated Turbo Algorithm and for conventional processing, a Turbo decoder following a DPLL, for different attenuator settings. Column 3 gives the average measured E_b/N_0 , which typically fluctuates a few

tenths of a dB from message to message. Each row of Table 1 gives the cumulative BER from 48 messages from the 5 second data collection at the indicated attenuator setting. As only 48,000 bits were sent at each attenuator setting the lower BER values are very rough.

Table 1: Bit Error Rates (BER) for old and new style processing of the Sept. 29 test data

Integrated Turbo BER	DPLL-then-Turbo BER	median E_b/N_0 (dB)	attenuation inserted (dB)
0	0	8.8	0
0	1.8×10^{-4}	4.0	1
0	2.9×10^{-3}	3.5	2
0	0.12	2.8	3
0.30	0.35	-0.5	4
0.37	0.43	-2.7	5
0.42	0.46	-4.8	6

The ITA out-performed conventional processing at all E_b/N_0 , and did not make any bit errors in the 4 runs with positive SNR. In the last 3 runs the SNR fell below the usable level for either processing scheme. The results for both new and old style processing agreed well with that observed in simulations as shown in Figure 6 for example. These results validate an expectation of a 4 dB gain at a BER of 10^{-5} for this ITA configuration (block length, constraint length, etc).

5 FPGA Implementation of Soft-Output Viterbi Decoder for Turbo Codes Based on C Algorithm

The Integrated Viterbi Algorithm (IVA) uses integrated delay and phase lock loops for decoding convolutionally encoded spread spectrum BPSK signals. The algorithm has been tested with C and Matlab and has demonstrated promising performance for bit error correction for use in low-power applications. The algorithm programmed in commercial digital signal processors, however, may not easily meet the speed required for desired bit rates. Here, an FPGA implementation is investigated.

The soft-output Viterbi algorithm (SOVA), although useful for simple convolutional codes, is also a main component in Turbo decoding. It is the most computationally complex part of the Turbo algorithm and requires the most hardware. This paper focuses on the implementation of this component, which we call SOVA1, in FPGA hardware. The output from this module is interleaved and iteratively refined through another Viterbi decoder that feeds back input into SOVA1. This second Viterbi decoder is a simplified version of the SOVA and is referred to as SOVA2. The interleavers, since inherently sequential, can be implemented in embedded software or in additional hardware.

The proposed hardware implementation is a “point” design with pre-defined parameters. These were chosen to test a useful design that may still fit on a single FPGA device. Although the design is synthesizable and provides size and speed estimates for a given set of design parameters, it has not yet been verified against software models and changes are still likely to be made.

The implementation described here uses a constraint length of 5, a code rate of 1/3, a message length of 1000 bits, and 512 chips per symbol. Our analysis concludes that this design could fit on a Xilinx Virtex-II XC2V8000, run at a 50 Mhz clock and operate at a maximum bit rate of 785 bps and chipping rate of 1.2 Mcps. These data rates are within the range where software models of the Turbo algorithm have demonstrated E_b/N_o savings of 3-5 dB over typical Viterbi decoder based systems.

5.1 Design Methodology

The hardware implementation is based primarily on C code describing SOVA behavior. Preliminary estimations on latency and size were based on line-by-line analysis of code.

To translate software into hardware, memory arrays are mapped to hardware memory elements and interfaces are converted to fixed widths. Iterations must be unrolled into multiple components to take advantage of parallelism in hardware.

Initial estimates were made using spreadsheets. The original C code was evaluated line by line to determine memory requirements and estimate clock cycles. Parameters affecting processing time and size include:

- Constraint length
- Number of chips per symbol

- Number of information bits
- Number samples per chip
- Number symbols per bit
- Oversampling rate
- Number of states computed in parallel
- Initial input width of signals
- Maximum bus width for signals
- Number of iterations through SOVA1/SOVA2 loop

We also must assume a maximum system bus-width and truncate data accordingly. If precision was maintained as values were added, multiplied, or otherwise operated on, system resources would be quickly exhausted. Since this has not yet been tested, however, any degradation in experimental results cannot be quantified.

5.2 Requirements

When implementing a software algorithm in hardware, several considerations are made to control size and speed:

- 1) All data becomes fixed-width in hardware. Input width and maximum bus width determines size of adders, multipliers, and memories. This also affects the ultimate precision of results. Data is treated as integers. If fractional bits are required, additional shifters may be needed to align data after arithmetic operations.
- 2) Memories are reorganized when advantageous to simplify addressing and speed up access time. Number of information bits, constraint length, and number of samples per chip determine amount of memory needed.
- 3) Iterations are unrolled and processed in parallel where possible. Dependencies between one iteration and the next will force the computation to be done recursively as with tracing forward or backward across message bits. In cases where computations may be done in parallel, as with states in the trellis, limited FPGA resources may also require computations to be done in segments.
- 4) Multiplies are best handled by embedded processors on the FPGA. The number of 18x18 multipliers available depends on the device chosen. Smaller multipliers can be handled in a lookup-table, and again, these resources are fixed. Limited multipliers may also force computations to be done sequentially.

For this design, we considered Xilinx's Virtex-II lineup of FPGAs. These are currently their highest density FPGAs, with system gate equivalents of up to 8 million.

Table 3. Xilinx Virtex-II FPGAs

Virtex II	XC 2V2000	XC 2V3000	XC 2V4000	XC 2V6000	XC 2V8000
System Gates	2M	3M	4M	6M	8M
Logic Cells	24,192	32,256	51,840	76,032	104,882
Slices	10,752	14,336	23,040	33,792	46,592
BRAM (Kbits)	1,008	1,728	2,160	2,592	3,024
18x18 Multipliers	56	96	120	144	168
Digital Clock Management Blocks	8	12	12	12	12
Max Dist. RAM Kb	336	448	720	1,056	1,456
Max Available User I/O	624	720	912	1,104	1,108

In order to estimate whether or not the Turbo design will fit on a single chip, we target the largest Virtex-II FPGA available. For the first prototype of this design, we assume certain parameters in hope of producing an implementation that can be tested on a single FPGA:

- Constraint length, $K=5$
- Number of chips per symbol = 512
- Number of information bits = 1000
- Number samples per chip = 2
- Number symbols per bit = 3
- Oversampling Rate = 8
- States computed in parallel = 8
- Input width = 8 bits
- Maximum data width = 32 bits
- Number of iterations through SOVA1/SOVA2 loop = 10

Using the above parameters, we were able to fit a design to the Virtex-II XC2V8000 part after synthesis as shown in Section 5.8, **SOVA1 and SOVA2 size and speed**. A comfortable utilization margin is desired for place and route, otherwise hand-optimizations may be necessary. Additional FPGA resources should also be available for the integration of the SOVA2 component and interleavers. A place and routed Turbo FPGA implementation has not yet been tried. Our analysis of latency and area is based on individual synthesis results of SOVA1 and SOVA2.

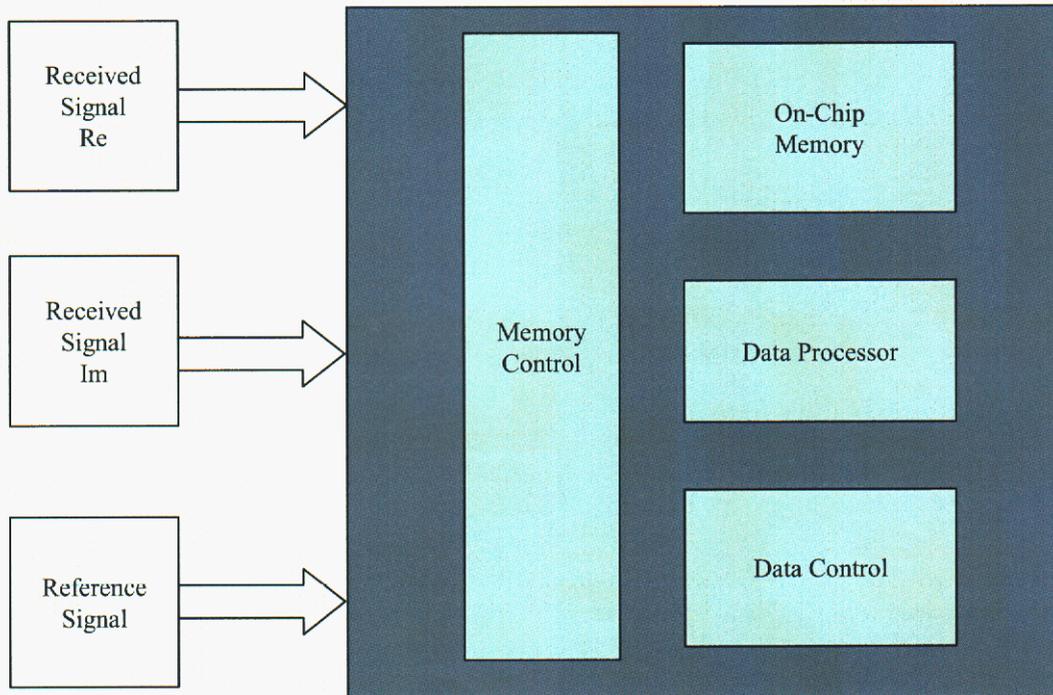
Methods for improving latency and tradeoffs for changing other parameters are also discussed in Section 5.8. Spreadsheets were refined after hardware implementation for better estimates of these tradeoffs.

5.3 FPGA Board

Since there is not enough memory available on an FPGA for all memory requirements, off-chip RAM needs to be available on the FPGA board. This is where received signals can be stored along with a pre-calculated reference signal. The board should have a standard JTAG interface and PROM for programming the Virtex-II. There should also be a way to initialize external RAM, preferably without using FPGA resources. The FPGA can treat the memory as read-only. An alternate FPGA for implementing second

Viterbi decoder and interleavers may also be desired to complete the Turbo decoder. Such a board may be commercially available for prototyping purposes.

The FPGA itself is organized in four main modules. The data processor provides all hardware for executing the algorithm. On-chip memory provides storage space for large arrays that are computed per iteration and needed for computations later in the algorithm. The data controller manages the data flow through the processor and to the other parts of the chip. The memory controller is enabled by the data controller, and handles transfers between FPGA and external RAM or local transfers that can be done without interrupting data flow. The memory controller updates local memories ahead of future iterations.



Off-Chip Memory

Xilinx Virtex II

Figure 10. Soval1 organization of FPGA and external memory

5.3.1 Data Processor

This module is what most closely correlates to the structure of the C algorithm. For a constraint length of 5, we must create the trellis for 16 states. Figure 11 represents the possible path from one state to the next for a given information bit. The “trace-forward” step is done for each information bit, as well as for the appended tail bits which guarantees that the trellis will end at a zero state. The total number of bits, *NUM_BITS*, including these tail bits is 1004. The real and imaginary parts of the received signal are input into this module, as well as its reference signal. For each state, these samples are despread into one systematic and two parity symbols. This is a multiply-accumulate operation, and given the number of multipliers available, all three symbols can be computed in parallel for 8 states at a time. The result is stored in on-chip memory.

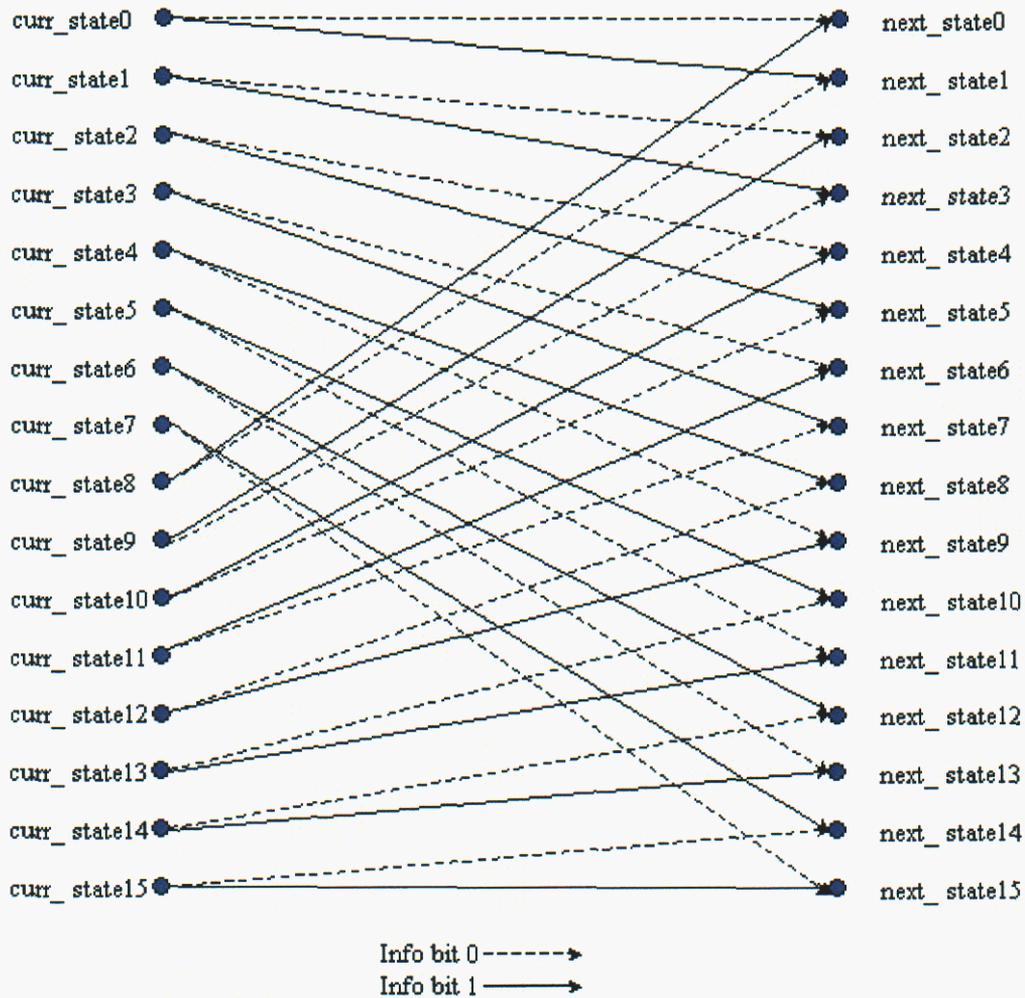
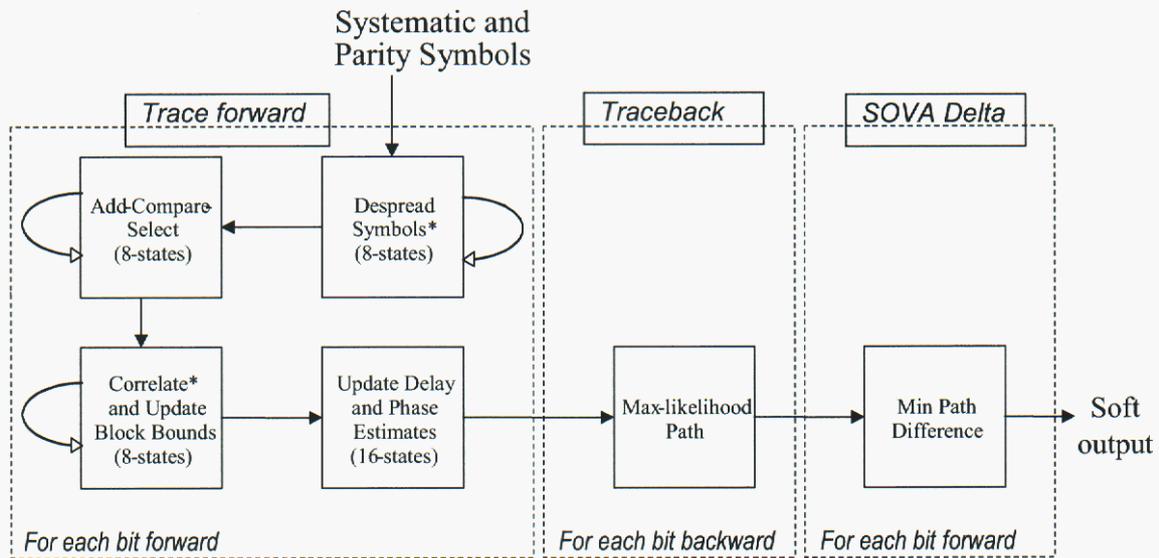


Figure 11. Trellis for K = 5. Solid line for bit = '1', dotted line for bit = '0'

Because the computation is done for 8 states, the **Despread Symbol** process must be executed twice before feeding input into the next stage as in Figure 12. All 16 symbols must be ready to compute path metrics in the **Add-Compare-Select** stage. Repeating this stage then gets delay estimates ready for the **Correlate and Update** stage, which also must be repeated. The **Update Delay and Phase** stage processes all states in parallel before the next bit of information while tracing forward.



*Complex Multiply and Accumulate hardware shared for despreading symbols and early and late correlators

Figure 12. Data Flow for computing soft outputs in data processor

After the trellis is created for the given message, the path with the best metric (max-likelihood) is selected by “tracing back” to the first bit of information in the **Traceback** stage. The final stage does one more pass through the information bits to determine the minimum path metric difference determined from any deviation from the max-likelihood path. This is referred to as the **SOVA Delta Loop**. The resulting soft output for each message is buffered locally or passed to the next device external to the FPGA.

The architectures for each of these sub-modules in the data flow are detailed in the following sections.

5.3.2 Despread Symbol

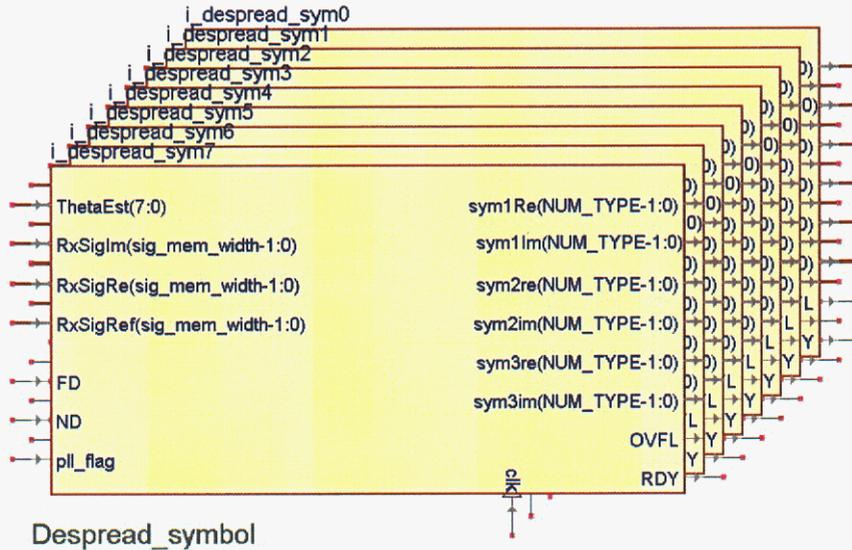


Figure 13. Despread_symbol component interface

Table 4. Despread_symbol component interface description

Despread Symbols		
Port	Direction	Description
ThetaEst[7:0]	in	Projected data delay estimate for this state
RxSigIm[7:0]	in	Imaginary component of received signal
RxSigRe[7:0]	in	Real component of received signal
FD	in	Indicates first RxSigRe, RxSigIm, and RxRef on bus for despreading signal. In the case of 128 chips per sample, this is the first of 256 iterations.
ND	in	Indicates new RxSigRe, RxSigIm, and RxRef remainder of despreading computation.
pll_flag	in	'1' if PLL is used. '0' if PLL off and assume signal is real.
sym1re[31:0]	out	Real component of first systematic symbol
sym1im[31:0]	out	Imaginary component of first systematic symbol
sym2re[31:0]	out	Real component of second systematic symbol
sym2im[31:0]	out	Imaginary component of second systematic symbol
sym3re[31:0]	out	Real component of parity symbol
sym3im[31:0]	out	Imaginary component of parity symbol
OVFL	out	Overflow flag for multiply and accumulates
RDY	out	Asserts when all iterations complete

This module performs the first 8-state parallel function in the bit-forward stage in Figure 12. It takes in real and imaginary components of received and reference signals, and performs a complex multiply. Code Excerpt 1 demonstrates the despreading of the first systematic symbol, *SysSymMat*. The variable *Nsamp* is the number of chips per symbol multiplied by the number of samples per chip. The *DelayEst* value is used in the addressing of the reference signal, *RxRef*, and is controlled externally (see Section 5.6:

Memory Control). The *SysSymMat* variable is stored in On-Chip Memory. Computations for parity symbols, *ParSym1Mat* and *ParSym2Mat*, are handled similarly.

Code Excerpt 1. Despreading symbols

```

1  for (state = 0; state < Nstates; state++) {
2
3      D = DelayEst[state];    // get projected delay estimate for this state
4
5      if (pll_flag)
6          ThetaEst = TwoPi*phi_nco[state];
7      else
8          ThetaEst = 0;      // turn PLL off, assume sig is real
9
10     // printf("ThetaEst=%f \n", ThetaEst );
11     re2 = cos( ThetaEst); im2 = -sin( ThetaEst); // pre-compute for loops
12
13     // despread first symbol
14     for ( i=0, symlre=symlim=0 ; i< Nsamp; i++) {
15         re1 = RxSigRe[index1+i] * RxRef[index1*Os+D+i*Os];
16         im1 = RxSigIm[index1+i] * RxRef[index1*Os+D+i*Os];
17         symlre += re1*re2-im1*im2;    // term1 = x1 * x2    (complex)
18         symlim += re1*im2+re2*im1;
19     }
20
21     SysSymMat[state][t]=symlre;
22     //...
23     // Despreading second and third symbols and assign
24     //...
25 }

```

Each state requires its own *DelayEst* and *phi_nco* memory arrays. *RxSigRe* and *RxSigIm* in lines 15 and 16 are also independent of state, so these can be read directly from external memory and fed to each of eight *despread_symbol* components. The *RxRef* signal, however, does depend on state and requires special handling to provide 8 unique words to the components. This is explained further in Section 5.6.3, **RxRef_ctrl**.

The complex multiply and accumulate hardware used in computing lines 14 to 19 is pictured below. This hardware is similar for despreading second and third symbols not described in code. The 8x8 multiply in lines 15 and 16 and the 16x16 multiply in lines 17 and 18 are done with dedicated multiplier resources on the Virtex-II part. For 1024 cycles (*Nsamp*), the mac1024 parts multiply and accumulate values. When *RDY* goes high, the symbol summations are valid.

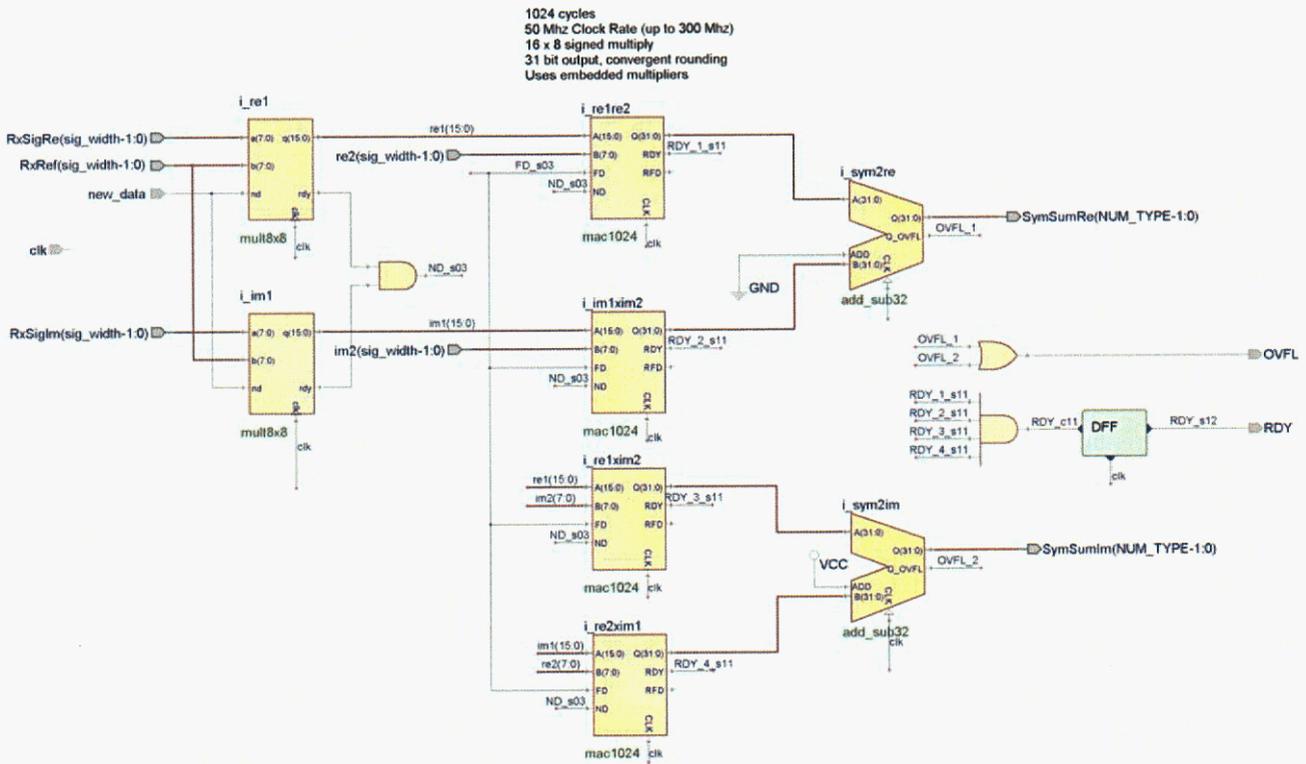


Figure 14. Complex Multiply and Add Hardware

The real and imaginary components of three output symbols are computed in parallel for 8 states. This utilizes 144 multipliers, which would be available on a Xilinx XC2V6000 part or larger. Results are written to On-Chip RAM and used in the **Add-Compare-Select** stage.

5.3.3 Add Compare Select

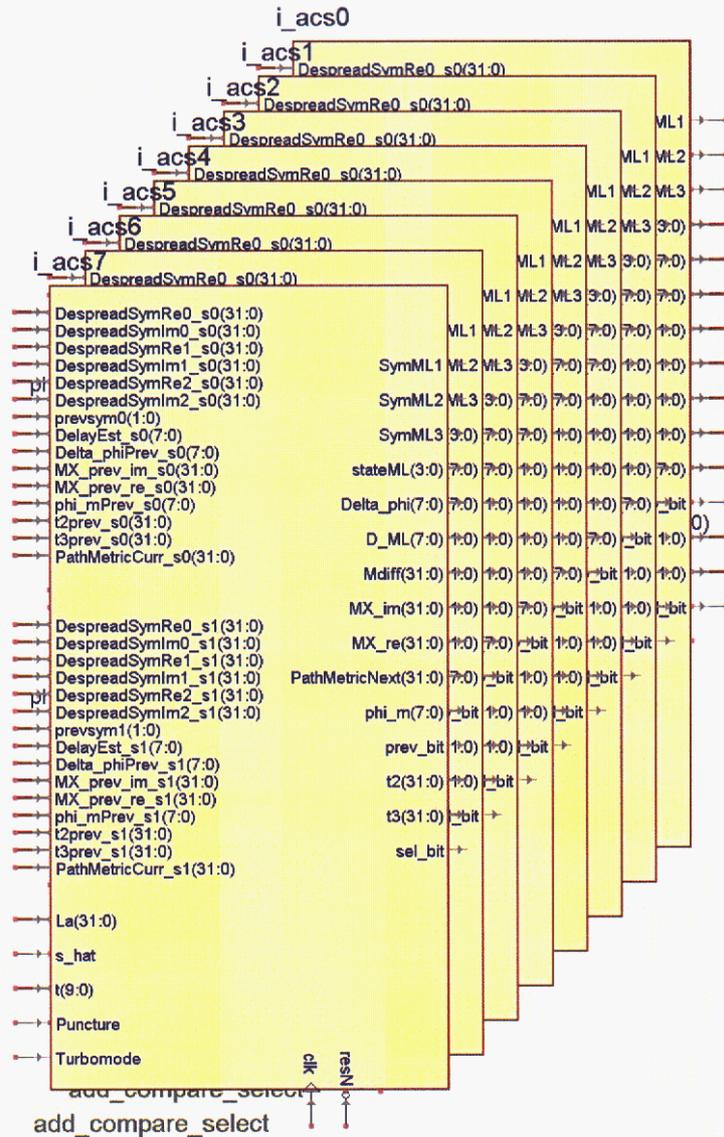


Figure 15. Add_Compare_Select component interface

Table 5. Add_Compare_Select component interface description

Add Compare Select		
Port	Direction	Description
DespreadSymRe0_s0[31:0]	in	Sym1re for case data bit = '0'
DespreadSymIm0_s0[31:0]	in	Sym1im for case data bit = '0'
DespreadSymRe1_s0[31:0]	in	Sym2re for case data bit = '0'
DespreadSymIm1_s0[31:0]	in	Sym2im for case data bit = '0'
DespreadSymRe2_s0[31:0]	in	Sym3re for case data bit = '0'
DespreadSymIm2_s0[31:0]	in	Sym3im for case data bit = '0'
prevsym0[1:0]	in	Symbol pair for data bit = '0'

DelayEst_s0[7:0]	in	Delay estimate for previous state for data bit = '0'
Delta_phiPrev_s0[31:0]	in	Phase rate per block for data bit = '0'
MX_prev_im_s0[31:0]	in	Real component of PLL accumulator for data bit = '0'
MX_prev_re_s0[31:0]	in	Imaginary component of pll accumulator for data bit = '0'
phi_mPrev_s0[31:0]	in	Model phas for data bit = '0'
t2prev_s0[31:0]	in	Intermediate summer for residual phase for data bit = '0'
t3prev_s0[31:0]	in	Intermediate summer, including t2, for residual phase for data bit = '0'
PathMetricCurr_s0[31:0]	in	Total metric for path with data bit = '0'
<i>/input ports repeated for data bit = '1'</i>	<i>in</i>	<i>...</i>
La[31:0]	in	A-priori info
s_hat	in	Hard parity symbol estimate
t[9:0]	in	Bit location in message
Puncture	in	If '1' punctured code and rate = 1/2, if '0' not punctured and rate = 1/3
Turbomode	in	1' if turbo cude, '0' if simple convolutional code
SymML1	out	Estimated systematic symbol
SymML2	out	Estimated parity symbol
SymML3	out	If punctured, not used. Else second parity symbol
stateML[3:0]	out	Estimated next state
Delta_phi[31:0]	out	Phase rate per block
D_ML[7:0]	out	DLL delay
Mdiff[31:0]	out	Difference between metrics
MX_im[31:0]	out	Imaginary component of pll integrator
MX_re[31:0]	out	Real component of pll integrator
PathMetricNext[31:0]	out	Path metric for next state in path
phi_m[31:0]	out	Model phase
t2[31:0]	out	Intermediate summer for residual phase
t3[31:0]	out	Intermediate summer, including t2, for residual phase
sel_bit	out	Selects estimated bit for updating delays in RxRef

In the **Add-Compare-Select** stage, metrics are compared to judge whether the most likely data bit in the current location of the message is a '0' or a '1'. Metrics for data bit '0' are computed as in Code Excerpt 2. Metrics for data bit '1' are computed similarly and in parallel.

This code is implemented using two's complement blocks and adders. The signal *sym1* in line 4 handled is used as a control to a two's complement block for evaluating *TranMet0Re* or *TranMet0Im* in lines 19 and 20. If the high bit of *Syms0* in line 4 is '1', the core function is bypassed. Otherwise, a two's complement of *sym1re* and *sym1im* in line 19 and 20 is output.

Code Excerpt 2. Computing metric for 0 data bit

```

1 state0 = prevstate0[state]; // get prev state associated with info bit=0
2 Syms0  = prevsym0[state];   // get sym pair associated with info bit=0
3
4 sym1 = (Syms0&2) ? 1 : -1; sym2 = (Syms0&1) ? 1 : -1; // split out syms
5
6 sym1re = DespreadSymRe[0][state0]; // get 1st despread sym from state0

```

```

7  sym1im = DespreadSymIm[0][state0];
8  sym2re = DespreadSymRe[1][state0]; // get 2nd despread sym from state0
9  sym2im = DespreadSymIm[1][state0];
10
11 if (TurboMode && !Puncture) {
12     sym3re = DespreadSymRe[2][state0];
13     sym3im = DespreadSymIm[2][state0];
14 }
15
16 if (TurboMode) {
17     if (Puncture) {
18         if (t%2) { // parity punctured
19             TranMet0Re = sym1 * sym1re;
20             TranMet0Im = sym1 * sym1im;
21             Sova2energy0Re = s_hat[2*t+1]*sym2re;
22             Sova2energy0Im = s_hat[2*t+1]*sym2im;
23         } else { // parity is available
24             TranMet0Re = sym1 * sym1re + sym2 * sym2re;
25             TranMet0Im = sym1 * sym1im + sym2 * sym2im;
26             Sova2energy0Re = 0;
27             Sova2energy0Im = 0;
28         }
29     } else { // unpunctured, third sym is parity for SOVA #2
30         TranMet0Re = sym1 * sym1re + sym2 * sym2re;
31         TranMet0Im = sym1 * sym1im + sym2 * sym2im;
32         Sova2energy0Re = s_hat[2*t+1] * sym3re;
33         Sova2energy0Im = s_hat[2*t+1] * sym3im;
34     }
35 } else { // not a Turbo code, simple conv code only
36     TranMet0Re = sym1 * sym1re + sym2 * sym2re;
37     TranMet0Im = sym1 * sym1im + sym2 * sym2im;
38     Sova2energy0Re = 0;
39     Sova2energy0Im = 0;
40 }

```

The **Add-Compare-Select** stage is completed in 5 clock cycles. In the first clock, path metrics for either case are computed in parallel. The next states in the trellis associated with bit ‘0’ or bit ‘1’ are called *state0* and *state1*, respectively. Using the symbol pair for *state0* and *state1* as bypass controls to two’s-complement blocks, BPSK modulation is removed from the despread symbols. In effect, the symbols are multiplied by 1 or -1. For every other iteration for a punctured code or every iteration for a non-punctured code, the transition metric is an addition of terms in the second clock. *Sova2energy* is computed at the output of another pair of two-complement blocks, controlled by hard *s_hat* symbols (meaning a value of ‘1’ or ‘-1’) produced by the Sova2 decoder. In the third and fourth clock cycles, path metrics are computed from both possible paths and are compared using a priori information, *La*. Finally, in the fifth clock, the best metric is selected and systematic and parity symbols are estimated. Accumulated delta and phase delays from *state0* or *state1* propagate to the current state.

5.3.4 Correlate and Update

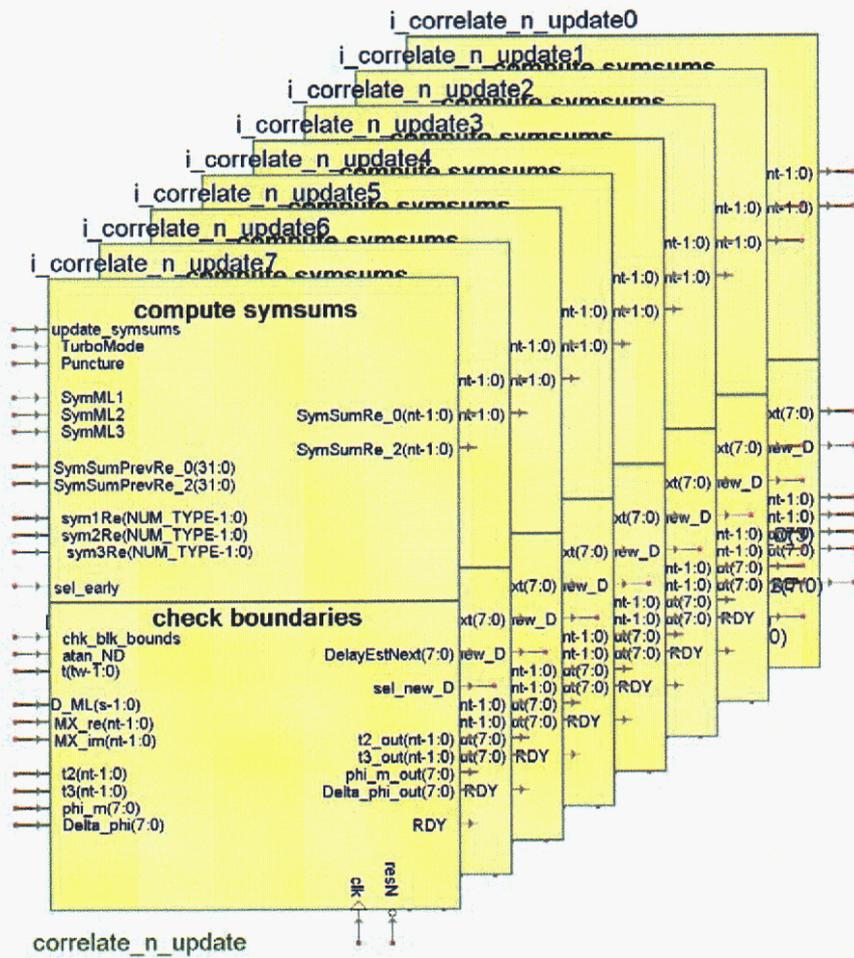


Figure 16. Correlate_n_update component interface

Table 6. Correlate_n_update component interface description

Correlate and Update		
Port	Direction	Description
update_symsums	in	Enables correlations to be compared
TurboMode	in	1' if turbo code, '0' if simple convolutional code
Puncture	in	If '1' punctured code and rate = 1/2, if '0' not punctured and rate = 1/3
SymML1	in	Estimated systematic symbol
SymML2	in	Estimated parity symbol
SymML3	in	If punctured, not used. Else second parity symbol
SymSumPrevRe_0[31:0]	in	Previous summation for early gate
SymSumPrevRe_2[31:0]	in	Previous summation for late gate
sym1Re[31:0]	in	Real component of first systematic symbol from <i>despread_symbol</i> component

sym2Re[31:0]	in	Real component of parity symbol from despread_symbol component
sym3Re[31:0]	in	Real component of second parity symbol from despread_symbol component
sel_early	in	If '1' calculate early correlation, else calculate late correlation
chk_blk_bounds	in	If '1', update DLL and PLL with new delays and phases
atan_ND	in	If '1', indicates new MX_{im} and MX_{re} ready for calculation of Δ_{phi}
t[9:0]	in	Bit location in message
D_ML[7:0]	in	DLL delay
MX_re[31:0]	in	Real component of pll integrator
MX_im[31:0]	in	Imaginary component of pll integrator
t2[31:0]	in	Intermediate summer for residual phase
t3[31:0]	in	Intermediate summer, including t2, for residual phase
phi_m[31:0]	in	Model phase
Delta_phi[31:0]	in	Phase rate per block
SymSumRe_0[31:0]	out	Summation for early gate
SymSumRe_2[31:0]	out	Summation for late gate
DelayEstNext[7:0]	out	Updated delay for next iteration
sel_new_D	out	If '0', delay is decremented, else delay is incremented
t2_out[31:0]	out	Updated intermediate summer for residual phase
t3_out[31:0]	out	Updated intermediate summer, including t2, for residual phase
phi_m_out[31:0]	out	Updated model phase
Delta_phi_out[31:0]	out	Updated phase rate per block
RDY	out	Ready asserted after updating DLL and PLL

The delay and phase lock loops (DLL and PLL) are maintained in the **Correlate and Update** stage. As in the *despread_symbol* block, the spread spectrum signals must be multiplied and accumulated over 1024 samples, only this time for both early and late delay estimates.

Code Excerpt 3. Early and late correlations

```

1 // Sum RxSig over block length L for use in DLL, first sum Early Gate
2 re2 = cos( ThetaEst); im2 = -sin( ThetaEst);
3
4 // Despread first symbol
5 for ( i=term1re=term1im=0 ; i< Nsamp; i++) {
6   re1 = RxSigRe[index1+i] * RxRef[index1*Os+D_ML+DeltaD11+i*Os];
7   im1 = RxSigIm[index1+i] * RxRef[index1*Os+D_ML-DeltaD11+i*Os];
8   term1re += re1*re2-im1*im2;
9   term1im += re1*im2+re2*im1;
10 }
11
12 // Despread second symbol
13 // ...
14
15 if (TurboMode && !Puncture) {
16   // Despread third symbol
17   // ...
18
19   SymSumRe[0][state] = SymML1*term1re + SymML2*term2re +

```

```

21  SymML3*term3re + SymSumPrevRe[0][stateML];
22 } else {
23  SymSumRe[0][state] = SymML1*term1re + SymML2*term2re +
24  SymSumPrevRe[0][stateML];
25
26 }
27
28 // Sum for Late Gate
29
30 // Despread first symbol
31 for ( i=term1re=term1im=0 ; i< Nsamp; i++) {
32  re1 = RxSigRe[index1+i] * RxRef[index1*Os+D_ML-DeltaD11+i*Os];
33  im1 = RxSigIm[index1+i] * RxRef[index1*Os+D_ML-DeltaD11+i*Os];
34  term1re += re1*re2-im1*im2;
35  term1im += re1*im2+re2*im1;
36 }
37
38 // Despread second symbol
39 // ...
40
41 if (TurboMode && !Puncture) {
42  // Despread third symbol
43  // ...
44
45  SymSumRe[2][state] = SymML1*term1re + SymML2*term2re +
46  SymML3*term3re + SymSumPrevRe[2][stateML];
47 } else {
48  SymSumRe[2][state] = SymML1*term1re + SymML2*term2re +
49  SymSumPrevRe[2][stateML];
50 }

```

Once again, *RxRef* values must be accessed externally from the FPGA. The first time, the *RxRef* address is shifted a location early for the multiply-accumulate. The second time, the *RxRef* address is shifted a location late. These summations are used as early and late correlators for maintaining delay and phase locked loops. See Section 5.6.3 for further details.

The *correlate_n_update* component monitors our location in the message, *t*, and determines when the iteration is at a block boundary:

Code Excerpt 4. Updating DLL and PLL at block boundary

```

1  // update DLL and PLL if at block boundary
2  if( (t+1) % BlockLen == 0)
3  {
4    if( SymSumRe[0][state] > SymSumRe[2][state] ) // assume PLL pulled in
5      DelayEstNext[state] = D_ML+d11_step; // retard delay
6    else
7      DelayEstNext[state] = D_ML-d11_step; // advance delay
8
9    // reset to 0 and integrate over next block
10   SymSumRe[0][state] = 0;
11   SymSumRe[2][state] = 0;
12
13   delta_phi[state] = atan2( MX_im[state], MX_re[state] )/TwoPi;
14   t2[state] = t2[state] + delta_phi[state];
15   t3[state] = t3[state]+t2[state];
16   Delta_phi[state] = K1*delta_phi[state] + K2*t2[state] + K3*t3[state];
17   phi_m[state] = phi_m[state] + Delta_phi[state];
18 } else {
19   // Propagate Delay Estimate for use in next iteration
20   DelayEstNext[state] = D_ML;
21 }
22 }

```

In this case, a *BlockLen* is defined to be 5 bits long. If the summation for the early correlation is greater than the summation for the late correlation, the delay estimates are incremented one step. Delay estimates are used primarily as address pointers to *RxRef*, *RxSigRe*, and *RxSigIm*, values stored in external memory. For an oversampling rate of 8, a *dll_step* is an interval of 8 address locations. Similarly, if the late estimate has better correlation, the address pointer is decremented 8 locations. This manipulation is handled in the *ram_ctrl* component, which transfers *RxRef* values to on-chip memory for parallel processing (Section 5.6.3).

To update phase rate in line 16 of Code Excerpt 4:

$$\Delta_{\phi} = K1 \cdot \tan^{-1} \left(\frac{MX_{im_state}}{MX_{re_state}} \right) + K2 \cdot t2 + K3 \cdot t3$$

where coefficients $K1 = 0.5463$, $K2 = 0.1768$, $K3 = 0.02470$ (in 32-bit fractional unsigned representation 0x8BDA5119, 0x2D42C3C9, 0x00652BD3C); MX_{im_state} and MX_{re_state} are PLL accumulators; and $t2$ and $t3$ are residual phase summers. In hardware, the arctan is computed using CORDIC algorithms. Although there is a 22 clock cycle latency in computation, the computation can be started after the MX_{im} and MX_{re} values are ready from the *add_compare_select* block. With two sets of signal despreading (1024 clock cycles each) in the **Correlate and Update** stage before the DLLs and PLLs need to be updated, there are more than enough clock cycles available before new outputs need to be ready. The multiplies are done in lookup tables (LUTs). Since $K1$, $K2$, and $K3$ are constant, 32-bit results for a 32x8 multiply can be accessed by an 8-bit address.

5.3.5 Update Delays and Sums

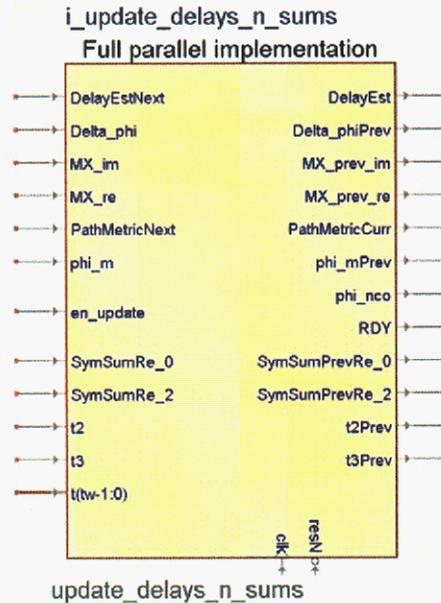


Figure 17. `update_delays_n_sums` component interface

Table 7. Update_delays_n_sums component interface description

Update Delay and Sums		
Port	Direction	Description
DelayEstNext[15:0][7:0]	in	Delay for next iteration for each state
Delta_phi[15:0][7:0]	in	Phase rate per block for each state
MX_im[15:0][31:0]	in	Imaginary component of pll integrator for each state
MX_re[15:0][31:0]	in	Real component of pll integrator for each state
PathMetricNext[15:0][31:0]	in	Path metrics for next state in path
phi_m[15:0][7:0]	in	Model phase for each state
en_update	in	Enables update for next iteration
SymSumRe_0[15:0][31:0]	in	Summation for early gate for each state
SymSumRe_2[15:0][31:0]	in	Summation for late gate for each state
t2[15:0][31:0]	in	Intermediate summer for residual phase for each state
t3[15:0][31:0]	in	Intermediate summer, including t2, for residual phase for each state
t[9:0]	in	Bit location in message
DelayEst[15:0][7:0]	out	Delay for each state
Delta_phiPrev[15:0][7:0]	out	Previous phase rate per block for each state
Mx_prev_im[15:0][31:0]	out	Previous imaginary component of pll integrator for each state
MX_prev_re[15:0][31:0]	out	Previous real component of pll integrator for each state
PathMetricCurr[15:0][31:0]	out	Current path metrics for each state in path
phi_mPrev[15:0][7:0]	out	Previous model phase for each state
phi_nco[15:0][7:0]	out	Previous phase correction for each state
RDY	out	Indicates update is done
SymSumPrevRe_0[15:0][31:0]	out	Previous summation for early gate for each state
SymSumPrevRe_2[15:0][31:0]	out	Previous summation for late gate for each state
t2Prev[15:0][31:0]	out	Previous intermediate summer for residual phase for each state
t3Prev[15:0][31:0]	out	Previous intermediate summer, including t2, for residual phase for each state

After the **Correlate and Update** stage has operated on all states, delays, metrics, and other accumulator sums can be updated for the next iteration through the received message. An *en_update* control from *data_ctrl* enables the *update_delays_n_sums* component (Figure 17) to update PLL and DLL quantities. This block propagates values from present state to next state to set up the next iteration of the trace-forward loop. It also computes phase correction, *phi_nco*, for each of the 16 states. The *comp_phi_nco* module implements:

$$phi_nco_{state} = phi_m_{state} - \frac{Delta_phi_{state}}{2} + \frac{rampDelta_phi_{state}}{BlockLen}$$

The block utilizes 2 32-bit adders and a right-shift for the middle term. Since *BlockLen* is constant, an 8-bit addressed LUT in conjunction with a 3x8 LUT computes the last term. The *ramp* signal is a 3-bit counter that resets at the end of a block, or 5 iterations through the trace-forward loop.

5.3.6 Traceback

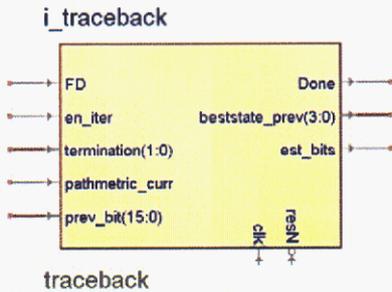


Figure 18. Traceback component interface

Table 8. Traceback component interface description

Traceback		
Port	Direction	Description
FD	in	Enables first iteration in traceback loop
en_iter	in	Enables next iteration
termination[1:0]	in	If '0', trace back from zero state, else trace back from state with highest metric
pathmetric_curr[15:0][31:0]	in	Current pathmetrics for each state
prev_bit[15:0]	in	Previous estimated bit
Done	out	Iteration done
beststate_prev[3:0]	out	Previous state for estimated bit
est_bits	out	Estimated bit

After the message has been traced forward across its length, the best path metric indicates the end of the max-likelihood path. By tracing back, the estimated bits and corresponding systematic and parity symbols along this max-likelihood path is determined.

Code Excerpt 5. Traceback for max-likelihood path

```

1 // if Term = 1 trace back from zero state
2 // if Term = 2 trace back from state with highest metric */
3 if (Term == 1)
4     beststate[NUM_BITS] = 0;
5 else
6     {
7         // find best metric
8         bestmetric = PathMetricCurr[0];
9         beststate[NUM_BITS] = 0;
10        for (i=0; i<NUM_STATES; i++)
11            {
12                if ( PathMetricCurr[i] > bestmetric)
13                    {
14                        bestmetric = PathMetricCurr[i];
15                        beststate[NUM_BITS] = i;
16                    }
17            }
18        }
19
20    for (t=NUM_BITS; t>0; t--)
21        {
22            est_bits[t] = ( prev_bit[t] & bitmap[ beststate[t] ] ) >> beststate[t];

```

```

23  if(est_bits[t])
24      beststate[t-1] = prevstate1[beststate[t]];           // est bit 1
25  else
26      beststate[t-1] = prevstate0[beststate[t]];           // est bit 0
27  if (TurboMode && !Puncture) {
28      DespreadSyms[3*t-3]=SysSymMat[beststate[t-1]][t-1];
29      DespreadSyms[3*t-2]=ParSym1Mat[beststate[t-1]][t-1];
30      DespreadSyms[3*t-1]=ParSym2Mat[beststate[t-1]][t-1];
31  } else {
32      DespreadSyms[2*t-2]=SysSymMat[beststate[t-1]][t-1];
33      DespreadSyms[2*t-1]=ParSym1Mat[beststate[t-1]][t-1];
34  }
35  }

```

At the first iteration, *en_iter* and *FD* input signals for the *traceback* block are asserted to initialize the last location in the message with *bestmetric* and *beststate*. For $t=1004$, a comparison tree finds the best metric for 16 states in 4 cycles. Tracing back, the estimated bit, *est_bits*, of '1' or '0' for the previous location determines the previous state. This value is used to address systematic and parity symbols for the bit location. Iterating to the 0th state takes another 1004 cycles. **Data Control** (Section 5.4) handles control for this recursive computation. **Memory Control** (Section 5.6) handles addressing for storing systematic and parity symbols for best states at each bit location.

5.3.7 Sova Delta

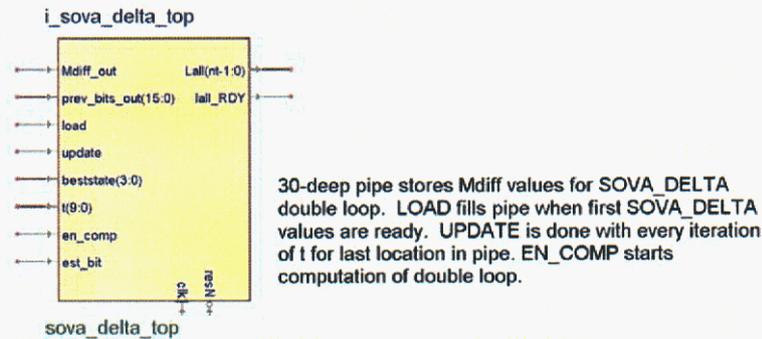


Figure 19. Sova_Delta_Top component interface

Table 9. Sova_Delta_Top component interface description

Sova Delta		
Port	Direction	Description
Mdiff_out[15:0][31:0]	in	Pathmetric differences for each state
prev_bits_out[15:0]	in	Previous bit estimates for each state
load	in	Loads Mdiff, prev_bit, beststate, and est_bit pipes for 30-deep sova delta calculation
update	in	Updates Mdiff, prev_bit, beststate, and est_bit pipes for 30-deep sova delta calculation after each iteration
beststate[3:0]	in	Best state for given bit location
t[9:0]	in	Bit location in message
en_comp	in	Enables nested sova delta loop computation
est_bit	in	Estimated bit
Lall[31:0]	out	Soft output for given bit location
lall_RDY	out	Indicates soft output ready for bit location

The *sova_delta_top* block finds the minimum path difference for an error path up to *SOVA_DELTA* bits from the current message location. It implements the following code:

Code Excerpt 6. Sova Delta loop

```

1  for (t=1; t<NUM_BITS+1; t++) // for each bit find Le
2  {
3      llr = BIG_POS; // set log-likelihood ratio to large number
4      for (i=0; i<=SOVA_DELTA; i++)
5      {
6          if( t+i < NUM_BITS+1) // do not go past end
7          {
8              error_bit = 1-est_bits[t+i]; //force an error at beginning of path
9              beststate_tb = beststate[t+i];
10             // trace back from bit error
11             for (j=i; j>0; j--)
12             {
13                 if(error_bit)
14                     beststate_tb = prevstate1[beststate_tb]; // est bit 1
15                 else
16                 {
17                     beststate_tb = prevstate0[beststate_tb]; // est bit 0
18                 }
19                 error_bit = (prev_bit[t+j-1]& bitmap[beststate_tb]) >> beststate_tb;
20             }
21             // after tracing back check if incorrect decision at stage t+i
22             // resulted an bit error at stage t.
23             if (error_bit != est_bits[t] )
24                 llr = min( llr, Mdiff[ beststate[t+i]][t+i]);
25         }
26     }
27     // calculate Lall for bit at stage t
28     // recall that llr is stored at t-1 while decoded bit is at t
29     Lall[t-1] = (2*(int)est_bits[t] -1)*llr;
30 }

```

In this implementation, *SOVA_DELTA* is 30 stages. If computed sequentially, as in software, this triple-nested loop would take

$$t_{SOVA_DELTA} = NUM_BITS * (SOVA_DELTA + 1) * \frac{SOVA_DELTA}{2}$$

iterations, or $1004 \times 30 \times 15 = 451800$ iterations. To optimize the computation, we structure the hardware as follows:

```

for all information bits {
    initialize log-likelihood ratio to be large integer
    for i={0,SOVA_DELTA},[1, SOVA_DELTA-1],...[15,SOVA_DELTA-15]{

        //sova_delta_i computation for i[0]
        force error at beginning of path for i[0]
        propagate error to location in path for i[0]
        if new path causes error, compare to log-likelihood ratio and keep minimum for
        i[0]

        //sova_delta_i computation for i[1]
        force error at beginning of path for i[1]
        propagate error to location in path for i[1]
        if new path causes error, compare to log-likelihood ratio and keep minimum for
        i[1]

    }
    compute soft output for bit location
}

```

Here, the j loop is absorbed into its parent i loop. The first *sova_delta_i* block takes 1 iteration to compute the loop for $i[0]=0$ and then *SOVA_DELTA* iterations to compute the loop for $i[1]=SOVA_DELTA$. The *sova_delta_i* block for the next i -pair, $[1, SOVA_DELTA-1]$, takes 2 iterations for the first loop and *SOVA_DELTA*-1 iterations for the next. The next i -pair takes 3 and *SOVA_DELTA*-2 iterations, and so on. Thus, all 15 pairs of *sova_delta_i* computations complete in *SOVA_DELTA*+1 iterations. The **Sova Delta Loop** for the entire message now takes

$$t_{SOVA_DELTA} = NUM_BITS * (SOVA_DELTA + 1)$$

iterations. Although the first loops through the *sova_delta_i* blocks may finish at different times, the blocks are in sync after the second loop. The control for *sova_delta_i* computations is shown below.

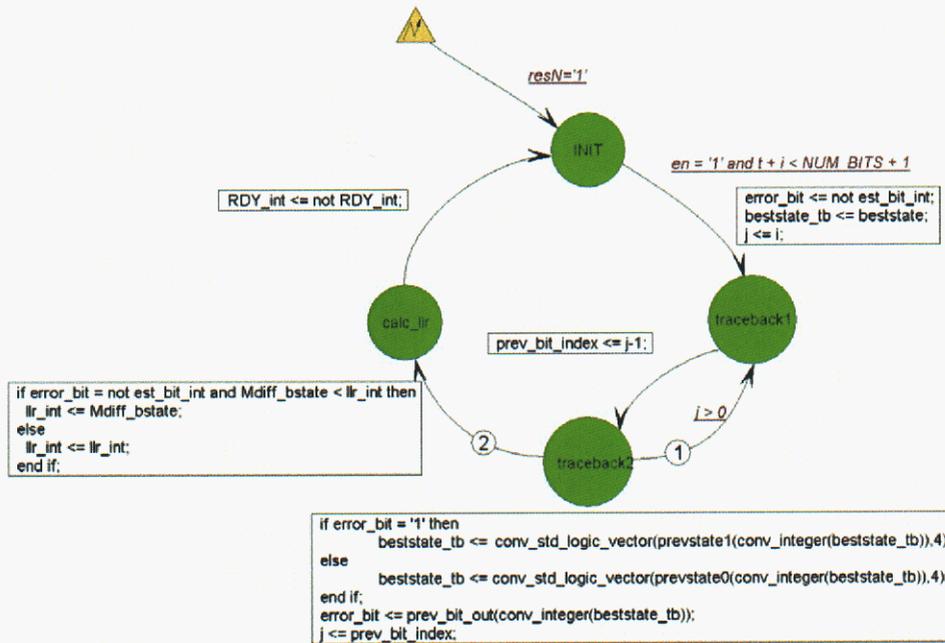


Figure 20. Sova_delta_i FSM

The finite state machine (FSM) computes *llr* (line 24) for $i[0]$ then $i[1]$. An error at location $t+i$ is assigned to *error_bit* (line 8), and the state corresponding to the max-likelihood path is assigned to *beststate_tb*. The j loop of lines 11-20 is executed between finite states *traceback1* and *traceback2*. If the forced error results in an alternate path through the trellis, *Mdiff_bstate* is the path metric difference between this alternate path and the estimated path.

Once again, memory access becomes an issue since all blocks need to access path metric differences, *Mdiff*, which are stored in RAM. For a given bit location, t , each i loop requires *Mdiff*[*beststate*[$t+i$]][$t+i$] in line 24. The *Mdiff* memory is organized as 1K x 512 bits. The wide output bus consists of 16 32-bit *Mdiff* values, each longword corresponding to a state in the trellis. The memory is addressed by the location in the message, or $t+i$ in this case. The path metric difference for an error along the path at this stage is selected by *beststate*[$t+i$].

For all i loops, $beststate[t]$ through $beststate[t+SOVA_DELTA]$ must be available. Since i loops are executed in parallel, a shift-register pipe 30 (or $SOVA_DELTA$) longwords deep is loaded with $Mdiffs[beststate[t]]$ to $Mdiffs[beststate[t+SOVA_DELTA]]$ during the last 30 iterations of the **Traceback** process. This is used in the first iteration of the **Sova Delta Loop**. For the next iteration, $Mdiff$ values are shifted such that $t_{new} \leq t+1$. The only $Mdiff$ value that needs updating for each t loop corresponds to $t_{new}+SOVA_DELTA$.

The $load_sd$ control signal is used when initializing the pipe with 30 $Mdiff$ values for $t=1$ to $SOVA_DELTA+1$. An $update_sd$ is used when updating for the next iteration, e.g. $t=2$ to $SOVA_DELTA+2$. Addressing is controlled by ram_ctrl while the $load_sd$ and $update_sd$ controls are asserted by $data_ctrl$.

Soft output, $Lall$, is generated in line 29 where the log-likelihood ratio, llr , is encoded as either a positive or negative fixed-point value. This is done through a twos-complement block with bypass capability.

5.4 Data Control

The $data_ctrl$ block is a finite state machine (FSM) which manages data flow through the processor. The top level FSM is shown in Figure 21.

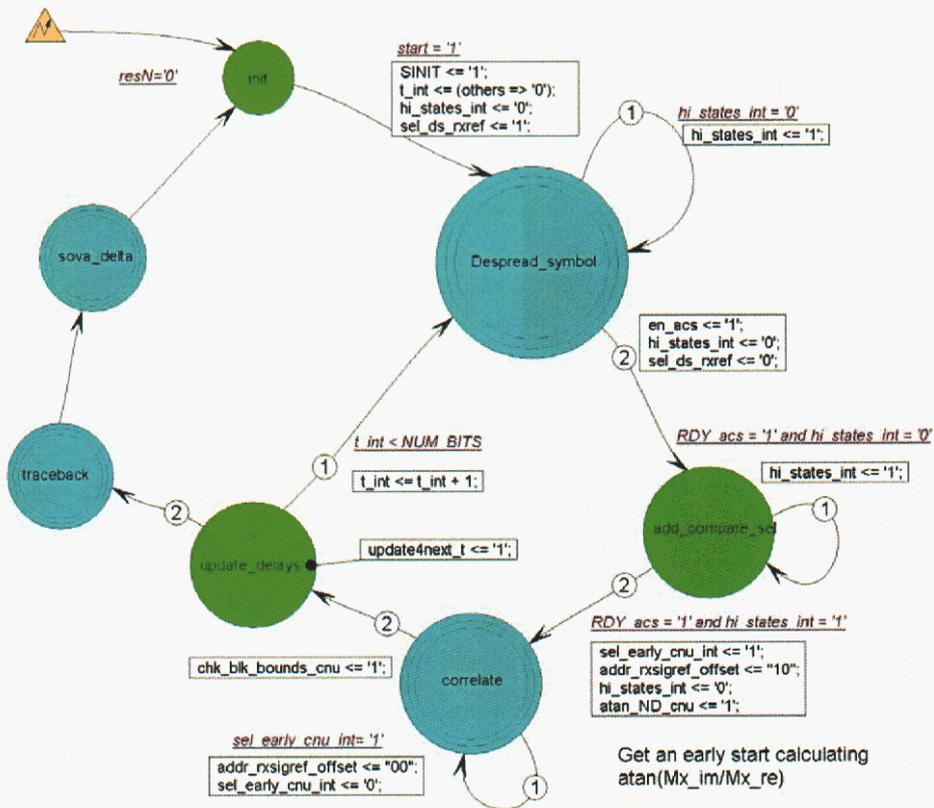
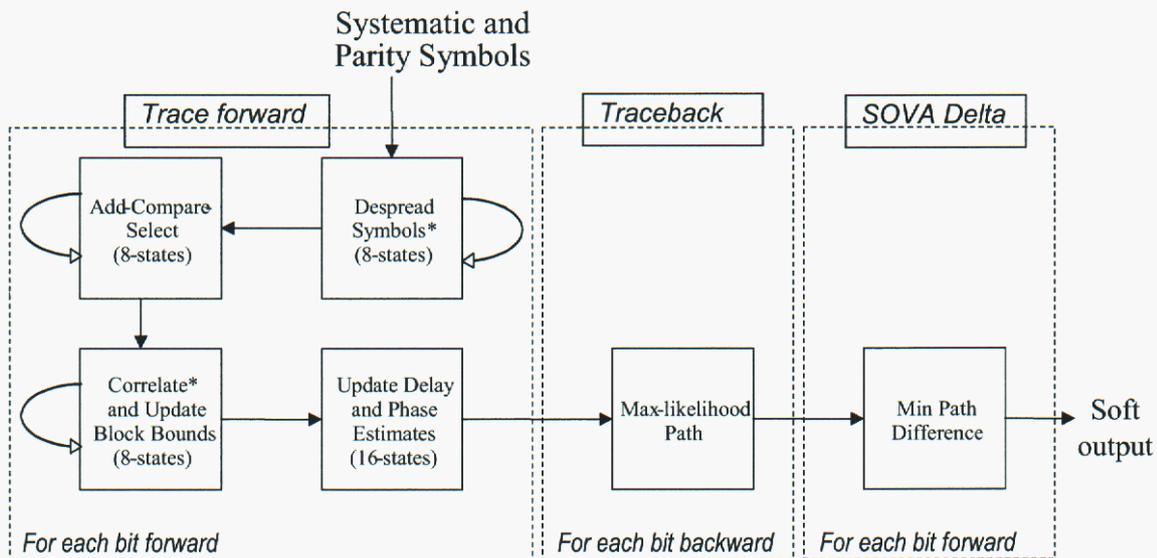


Figure 21. Finite state machine for data control block

Upon inspection, this resembles



*Complex Multiply and Accumulate hardware shared for despreading symbols and early and late correlators

Figure 12 in data flow. When the finite state machine is enabled, the **Despread Symbols** process is done for lower states, $hi_states_int = '0'$, and then for higher states, $hi_states_int = '1'$. The sel_ds_rxref signal is asserted and selects $RxRef$ outputs from on-chip memory for despreading symbols.

Despread symbols for all states in the trellis must be ready (RDY_acs) before the **add_compare_select** component can be enabled. Once enabled (en_acs), there is a 3 clock latency for selecting the metric for the best possible path to the next state in the trellis. The **Add-Compare-Select** process is done for lower states and then for higher states. The sel_ds_rxref signal at this point has already been de-asserted for $RxRef$ outputs used in the following **Correlate and Update** stage. The $atan_ND_cnu$ signal enables the arctan hardware in Section 5.3.4 to begin operating on new PLL accumulator values, MX_im and MX_re . The arc tangent is a CORDIC implementation using Xilinx's Core Generator. There is a 22 clock cycle latency for the computation, but this can be absorbed into the system's latency which is dominated by complex multiplies. The **Correlate and Update** stage itself completes 2 sets of complex multiplies, which require N_{samp} iterations each.

The $sel_early_cnu_int$ and $addr_rxsigref_offset$ assignments entering the **Correlate and Update** process set up memory writes for symbol summations ($SymSumRe$ in Code Excerpt 3) and memory reads for $RxRef$. Early correlations are done first, followed by late correlations.

After repeating the **Correlate and Update** process for lower and upper states, the DLL and PLL can be updated. The $chk_blk_bounds_cnu$ enables the **correlate_n_update** block to determine whether or not the next iteration falls on a block boundary, i.e. is a multiple of 5. If so, ram_ctrl is enabled to update $RxRef$ buffers.

The **Update Delays and Sums** process sets up values for the next iteration in the trace-forward loop. The *update4next_t* signal propagates symbol summations, path metrics, delays, and phase summations to the next state. *T_int* is incremented for *NUM_BITS* iterations through the trace-forward loop.

The **Traceback** loop control is implemented in the *traceback* state in the FSM. It is hierarchical and hides the loop in its child FSM. The *sova_delta* state, which implements the **Sova Delta Loop** in the FSM, is also hierarchical and hides a double-nested loop in its child FSM. These child FSMs, along with those for *Despread_symbol* and *correlate* states are described in the next sections.

5.4.1 Despread_symbol FSM

The *Despread_symbol* state machine is pictured in Figure 22. The *FD_despread_sym* and *ND_despread_sym* signals control the multiply and accumulate block in the *despread_symbol* hardware. The *rd_RxSigRef* signal enables another FSM in the *ram_ctrl* block for accessing the *RxRef* buffer in on-chip RAM. The complex multiply iterates for *Nsamp* cycles and has a 12 clock cycle latency. The *RDY_despread_sym* output of the *data_proc* block indicates that the computation is done. The *wr_despread_sym_res* signal is asserted for *ram_ctrl* to write systematic and parity results to on-chip RAM.

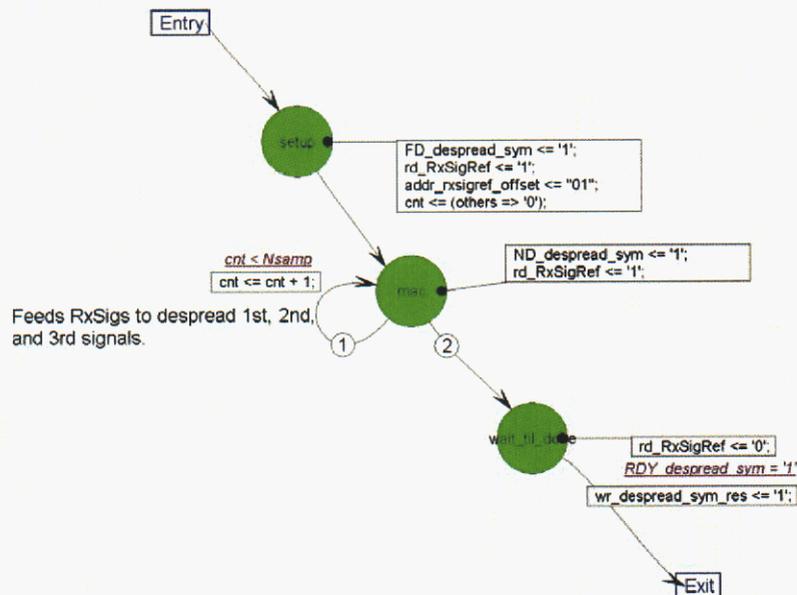


Figure 22. Finite state machine for despread_symbols

5.4.2 Correlate and Update FSM

The child FSM for the **Correlate and Update** process is pictured in Figure 23.

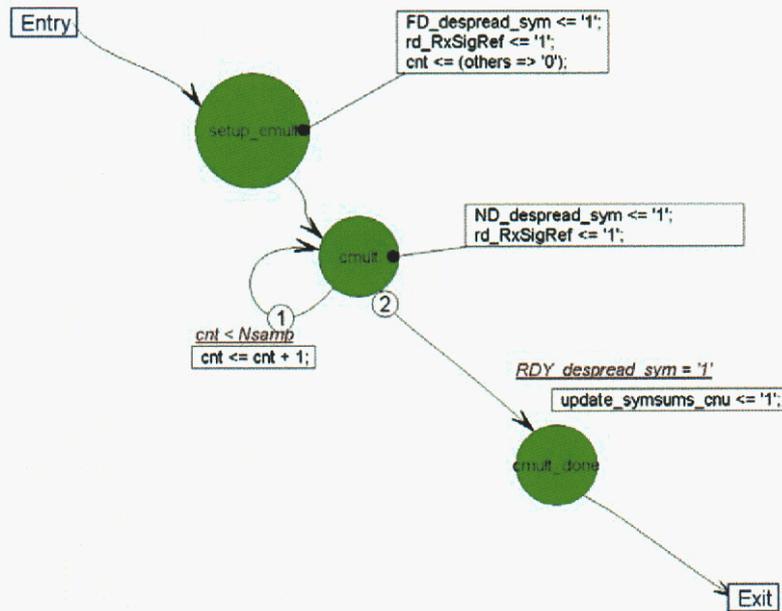


Figure 23. Finite state machine for correlation

The early and late correlators reuse the complex multiply hardware used in despreading symbols. Only the addressing to the *RxRef* buffers is different. The *addr_rxsigref_offset* of “10”, “01”, “00” addresses an early, middle, or late *RxSigRef* signals, respectively. More details are offered in Section 5.6.3. The *sel_early_cnu_int* signal (at the parent FSM level) toggles between designating an early estimate and late estimate for storing summations in the appropriate summation registers when *update_symsums_cnu* is asserted.

The comparison in lines 4-7 of Code Excerpt 4 determines how *RxSigRef* buffers will be updated for the next block of iterations (refer to Section 5.6.3: **RxRef_ctrl**). Correlation should take $2*(Nsamp+14)$ clock cycles.

5.4.3 Traceback FSM

Control for the **Traceback** loop is shown in Figure 24.

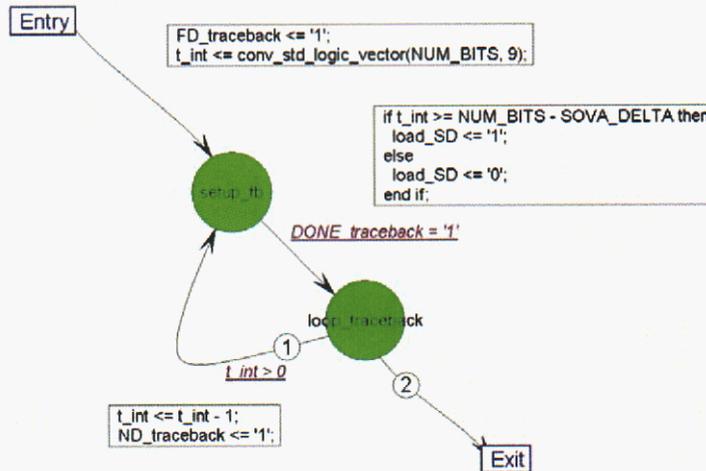


Figure 24. Finite state machine for traceback loop

This takes

$$t_{\text{traceback}} = \log_2(\text{NUM_STATES}) + \text{NUM_BITS} * t_{\text{loop}} + 1$$

cycles to complete. Time to select best metric takes $\log_2(\text{NUM_STATES})$ cycles, or 4 clock cycles. This selection is done before the first iteration of **Traceback** and is asserted by *FD_traceback*. Each iteration takes $t_{\text{traceback}}$ cycles, which is 3 clock cycles. At the end of every iteration, *ND_traceback*, indicates new systematic and parity symbols can be stored to memory. Another cycle is added for setup of the **Sova Delta Loop**. During this stage of data flow we also want to load the *Mdiff_pipe* that is needed in the **Sova Delta Loop**. This is loaded (*load_SD*) during the last *SOVA_DELTA* iterations of the **Traceback** process as the needed values become available.

5.4.4 Sova Delta Loop FSM

Data control for the **Sova Delta Loop** is pictured below.

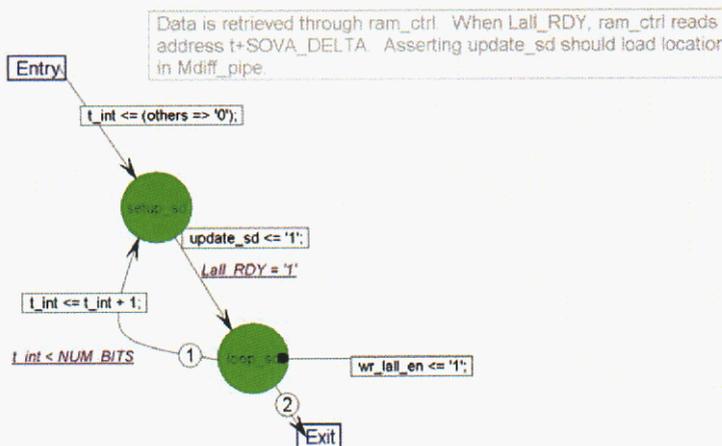


Figure 25. Finite state machine for Sova Delta loop

The **Sova Delta Loop**, as mentioned in Section 5.3.7, is a double-nested loop. *Data_ctrl*, however, only handles the outer-level iterations from 1 to *NUM_BITS* + 1. Control for the inner loop is summarized in Section 5.3.7. After each iteration, *lall* is written to external memory or another device for further processing.

5.5 On-Chip Memory

On-chip memory provides local memory access for data processor. Memory blocks may be implemented in the Virtex-II Block RAM or distributed RAM. Block RAM consists of dedicated memory resources that can be implemented as single or dual-port RAMs or ROMs. Alternatively, memory can use distributed resources utilizing LUTs.

In this implementation, block RAMs are used for each memory except for *rxref_buf3*, which are implemented using LUTs. Memories used on the chip are listed in Table 10.

	<i>size</i>	<i>Description</i>
SysSymMat	32 x 16Kbits	Systematic symbol generated in <i>despread_symbols</i>
ParSym1Mat	32 x 16Kbits	Parity symbol generated in <i>despread_symbols</i>
ParSym2Mat	32 x 16Kbits	For unpunctured Turbo-code, second parity symbol generated in <i>despread_symbols</i>
DespreadSyms	96 x 1Kbits	Final systematic and parity symbols (3 per address) generated in traceback loop
<i>est_bits</i>	1 x 1Kbits	Estimated bits from <i>add_compare_select</i>
<i>mdiff</i>	16 x 32 x 1Kbits	For each state, metric differences corresponding to each bit in message
<i>rxref_buf3</i>	16 x 3 x 24 x 1026 bits	For each state, 3 buffers corresponding to look-ahead, look-behind, and current correlations. Each buffer location contains 3 8-bit <i>RxSigRef</i> values from external memory for each systematic or parity symbol. The address depth of 1026 covers <i>Nsamp</i> iterations of a complex multiply plus extra locations at either end for first early-estimate values and final late-estimate values.
<i>beststate</i>	4 x 1Kbits	State corresponding to best path for every location in message

Table 10: On-Chip Memory Block

5.6 Memory Control

The purpose of this module is to handle data transfers to and from memory while minimizing clock cycles from data control. If a block of memory needs updating and data is available, then the process may be enabled with a single control signal from *data_ctrl*. Memory control, implemented in *ram_ctrl*, is needed for reading and writing despread symbols *SysSymMat*, *ParSym1Mat*, and *ParSym2Mat*; shift-registers *est_bits*, *beststate*, and *Mdiff* for **Sova Delta Loop** computations; and *RxRef* values for complex multiplies.

5.6.1 Despread Symbols Memory Control

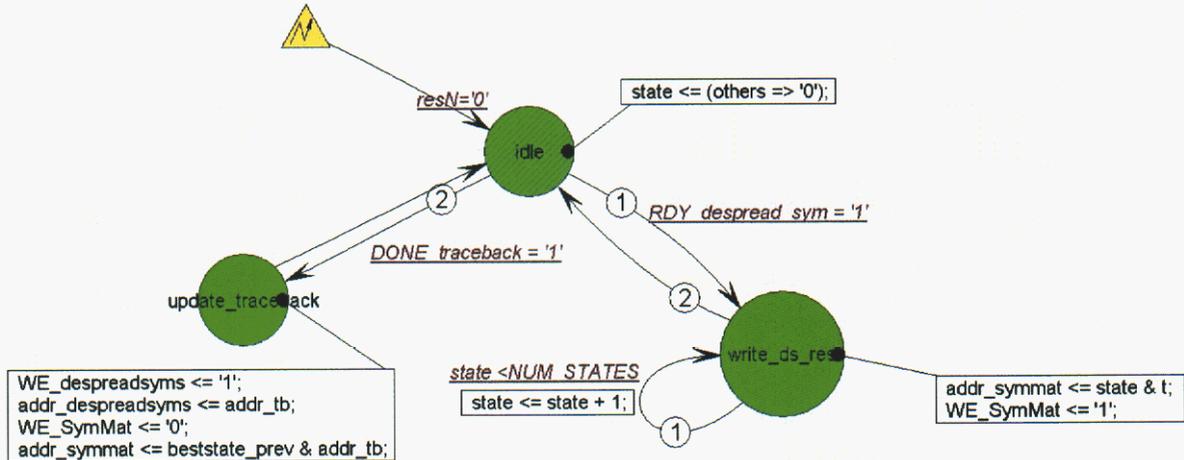


Figure 26. Finite State Machine for writing systematic and parity symbols and being despread

In the control pictured in Figure 26, the writes being handled in the *write_ds_res* state in the FSM are

```

SysSymMat[state][t]=sym1re;
ParSym1Mat[state][t]=sym2re;
ParSym2Mat[state][t]=sym3re;

```

This is from despadding symbols in line 21 of Code Excerpt 1, which is similar for first and second parity symbols. By default, addressing to these memories (*addr_symmat*) is set up for reads. The enable signal, *RDY_despread_sym*, from *data_ctrl* starts the write. The top 4 bits of *addr_symmat* is then set to *state=0* for the first state of the current iteration, *t*, in the trace-forward loop. The *SysSymMat*, *ParSym1Mat*, and *ParSym2Mat* memories, which are 32x16Kbit, are then sequentially written for the following states in the current iteration. This takes *NUM_STATES*3* clock cycles. The *state* signal is sent to a multiplexer to select the appropriate longword to be written to memory since *sym1re*, *sym2re*, and *sym3re* are produced for 8 states in parallel.

In the *update_traceback* state of the FSM, *SysSymMat*, *ParSym1Mat*, and *ParSym2Mat* (for non-punctured Turbo codes) are read in Code Excerpt 5 and written to *DespreadSyms*. The *DespreadSym* memory (Table 10) is 96 bits wide and all 3 32-bit symbols can be addressed by *addr_tb* (equivalent to *t-1*). *Beststate_prev* is the state for the estimated path at *t-1* and makes up the 4-MSBs of the *SymSymMat*, *ParSym1Mat*, and *ParSym2Mat* addresses. It determines the 3 symbols to be transferred to *DespreadSyms*.

5.6.2 Est_bits, beststate, and Mdiff memories

In Section 5.3.7, the *sova_delta_i* blocks required special handling for inputs for *SOVA_DELTA* computations. Shift registers are used to implement 30-deep pipes for *est_bits*, *beststate*, and *Mdiff* memories. The FSM for control is shown below.

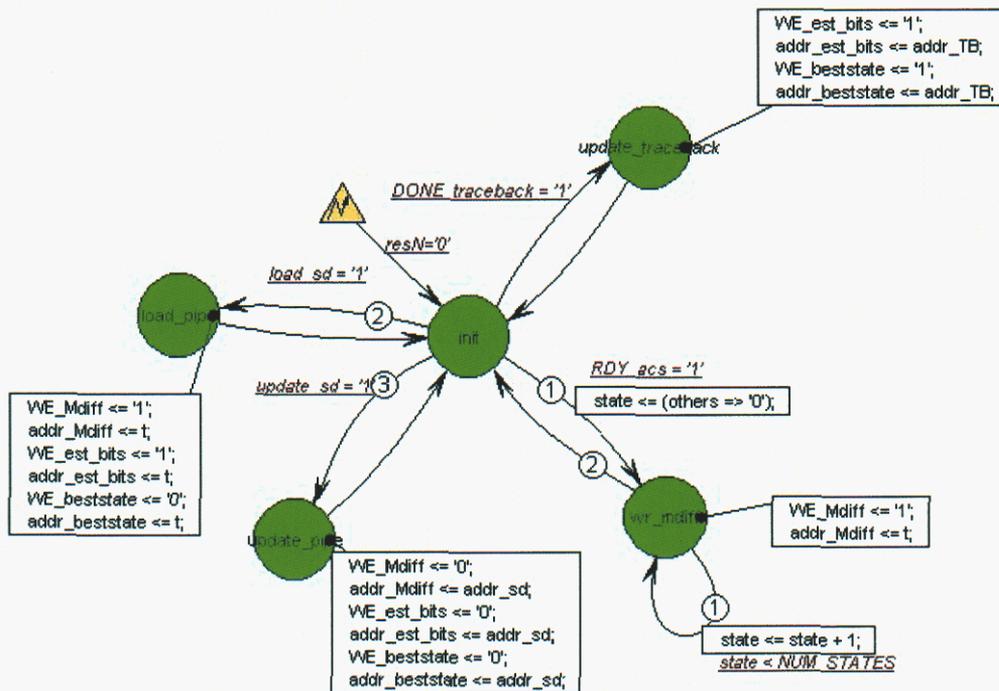


Figure 27. Reading and writing est_bits, beststate, and Mdiff memories

Est_bits is written in line 22 of the **Traceback** loop in Code Excerpt 5. *Beststate* is written in lines 15, 24, or 26. The enable signal *DONE_traceback* from *data_ctrl* puts the state machine in the *update_traceback* state where writes are enabled for the *est_bits* and *beststate* memories.

Mdiff, however, is written earlier, in the **Add-Compare-Select** stage. The *RDY_acs* signal from *data_ctrl* indicates when the FSM should be in the *wr_mdif* state to enable writes to *Mdiff* memory. Although *Mdiff* for all 16 states are ready, the values are written sequentially to memory.

Loading of 30-deep pipe for *est_bits*, *beststate*, and *Mdiff* is handled in the last iterations of the **Traceback** loop. This is done by the *load_pipe* state of the FSM when *data_ctrl* sends a *load_sd* signal to *ram_ctrl*. *Est_bits* and *beststate* memories must be of the Read-Before-Write type so that values can be read and shifted down the pipe in the same clock cycles as they are being written. *Mdiff* needs to be addressed and read along with *est_bits* and *beststate* in the last 30 iterations of the **Traceback** loop. The **Traceback** FSM in Figure 24 produces the *load_sd* signal for entering this *load_pipe* state where the shift-registers for *est_bits*, *beststate*, and *Mdiff* are initialized.

After each iteration for the **Sova Delta Loop**, *data_ctrl* sends an *update_sd* signal to *ram_ctrl* and puts the FSM in its *update_pipe* state. This sets up address and write signals to read the next *est_bits*, *beststate*, and *Mdiff* longwords to be updated in memory. The address, *addr_sd*, for this new longword corresponds to $t + SOVA_DELTA$ after t has

been incremented for the next iteration. *Data_ctrl* asserts *update_sd* after each iteration of the **Sova Delta Loop**.

5.6.3 RxRef_ctrl

In the Turbo algorithm, the RxRef buffer must be accessed for early, middle, and late correlations.

Code Excerpt 7. Despreading systematic symbol

```
for ( i=0, symlre=symlim=0 ; i< Nsamp; i++) {
    re1 = RxSigRe[index1+i] * RxRef[index1*Os+D+i*Os];
    im1 = RxSigIm[index1+i] * RxRef[index1*Os+D+i*Os];
    symlre += re1*re2-im1*im2;    // term1 = x1 * x2    (complex)
    symlim += re1*im2+re2*im1;
}
```

Index2 and *Index3* are used for parity symbols. For early and late estimations,

Code Excerpt 8. Early gate sum for systematic symbol

```
for ( i=term1re=term1im=0 ; i< Nsamp; i++) {
    re1 = RxSigRe[index1+i] * RxRef[index1*Os+D_ML+DeltaD11+i*Os];
    im1 = RxSigIm[index1+i] * RxRef[index1*Os+D_ML+DeltaD11+i*Os];
    term1re += re1*re2-im1*im2;
    term1im += re1*im2+re2*im1;
}
```

Code Excerpt 9. Late gate sum for systematic symbol

```
for ( i=term1re=term1im=0 ; i< Nsamp; i++) {
    re1 = RxSigRe[index1+i] * RxRef[index1*Os+D_ML-DeltaD11+i*Os];
    im1 = RxSigIm[index1+i] * RxRef[index1*Os+D_ML-DeltaD11+i*Os];
    term1re += re1*re2-im1*im2;
    term1im += re1*im2+re2*im1;
}
```

Note the different address schemes for *RxRef* in each example. *RxSigRe* and *RxSigIm* are not state dependant, so all 16 states can access the same memory locations. *RxRef*, however, has state dependency buried in *D* and *D_ML*.

For delay, *D*, in the first example in Code Excerpt 7, we have

```
D = DelayEst[state]
```

from line 3 in Code Excerpt 1. Since we want to compute 8 states in parallel, we need a buffer for each state that will cover *Nsamp* values for the complex multiply. To do this, we want to re-organize external memory onto the FPGA so that all 3 symbols can be accessed cycle by cycle.

```
RxRef[index1*Os+D+i*Os] = RxRef[Os*(index1+i)+D]
```

Since *Os* is constant (oversampling rate of 8 times) and *index1*, *index2*, and *index3* are constants, *RxRef* can then be sequentially accessed with *i* as with *RxSigRe* and *RxSigIm*. One such buffer is needed for each state with its individual offset, *D*.

In the second and third examples, which occur in the **Correlate_n_Update** stage, bit '0' or '1' has been estimated to determine whether

```
D_ML = DelayEst[prevstate0] OR D_ML=DelayEst[prevstate1].
```

This relates *D_ML* to *D* used for the first correlation, where *prevstate0* is equivalent to the previous state for an input of '0', and *prevstate1* is the previous state for an input of '1'. For these two possibilities, we need to map the delay, *D_ML*, to the appropriate

$RxRef$ buffer for bit ‘0’ or bit ‘1’. The trellis used for propagating from the current state to the next is shown in Figure 11. This corresponds to an array where:

$$\begin{aligned} prevstate0 &= \{0 \ 8 \ 1 \ 9 \ 2 \ 10 \ 3 \ 11 \ 12 \ 4 \ 13 \ 5 \ 14 \ 6 \ 15 \ 7\} \\ prevstate1 &= \{8 \ 0 \ 9 \ 1 \ 10 \ 2 \ 11 \ 3 \ 4 \ 12 \ 5 \ 13 \ 6 \ 14 \ 7 \ 15\} \end{aligned}$$

Thus, a ‘1’ input in $state15$ would stay in $state15$ for the next iteration since $prevstate1(15) = 15$. A ‘0’ input would propagate to $state14$ since $prevstate0(14) = 15$.

A multiplexer at the output of each $RxRef$ block can assign D_ML . The signal, sel_ds , selects the middle case for the **Despread Symbols** stage. For the **Correlate and Update** stage, the signal $sel_rxref(state)$ selects one of the two other cases for an information bit of ‘1’ or ‘0’.

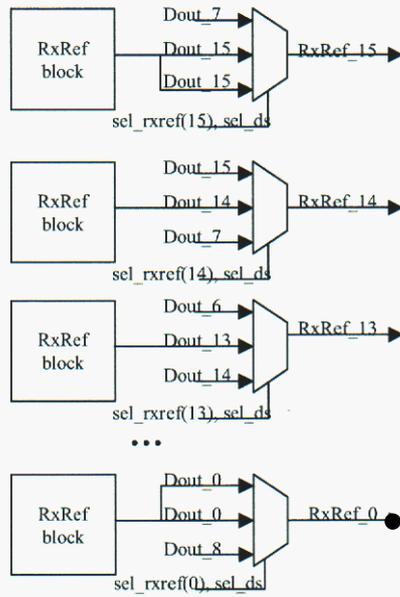


Figure 28. Assign $RxRef$ for despreading symbol and propagating input bits of ‘1’ or ‘0’

Now, extending into the case where early and late estimates are made, the $RxRef$ buffer should also allow for the $\pm DeltaDll$ values in Code Excerpt 8 and Code Excerpt 9. Since $DeltaDll = Os$,

$$RxRef[index1*Os + D_ML \pm DeltaDll + i*Os] = RxRef[Os*(index1+i\pm 1) + D_ML].$$

Again, $RxRef$ can be sequenced by the counter i . By loading in bytes offset by a constant $index1$, $index2$ and $index3$, our local $RxRef$ buffer can be addressed from $-Os + D$ to $Nsamp*Os + D$ as in Figure 29.

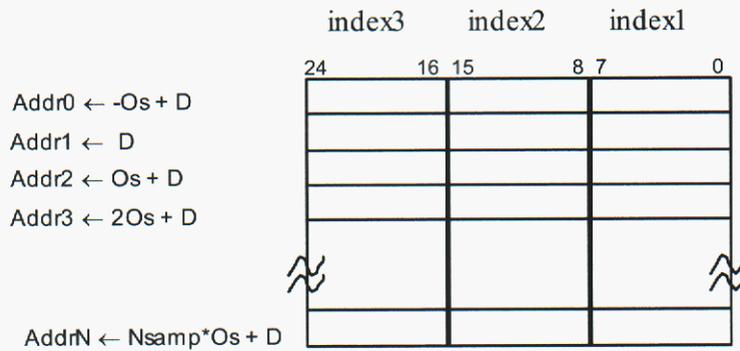


Figure 29. Mapping of RxRef in external memory to on-chip memory

Moreover, at each block boundary of length 5, the delays are incremented or decremented a step:

```

if(SymSumRe[0][state] > SymSumRe[2][state]) // assume PLL pulled in
    DelayEstNext[state] = D_ML+dll_step;      // retard delay
else
    DelayEstNext[state] = D_ML-dll_step;      // advance delay

```

The interval, *dll_step*, is a unit step, as opposed to *DeltaDLL*, which steps by 8 units. Timing becomes an issue since the *SymSumRe[0]* and *SymSumRe[2]* values are not ready until just before this update in the **Correlate_n_update** stage. The new *RxRef* values corresponding to the new *DelayEstNext*, however, are needed immediately for the next iteration, in the **Despread Symbols** stage. By triplicating the memories in Figure 29 for each state, we could assign a “look-behind” buffer for the case of *D_ML-dll_step* as well as “look-ahead” buffer for *D_ML+dll_step*. When the DLL is ready to be updated, the *RxRef* buffers for all states can be updated in parallel. Using Read-Before-Write memories keep data from being corrupted. Putting a second Read-Only port on the main (middle) buffer allows the **Despread Symbol** stage to process without added latency during the update. The structure of each *RxRef* block in Figure 28 is illustrated in Figure 29.

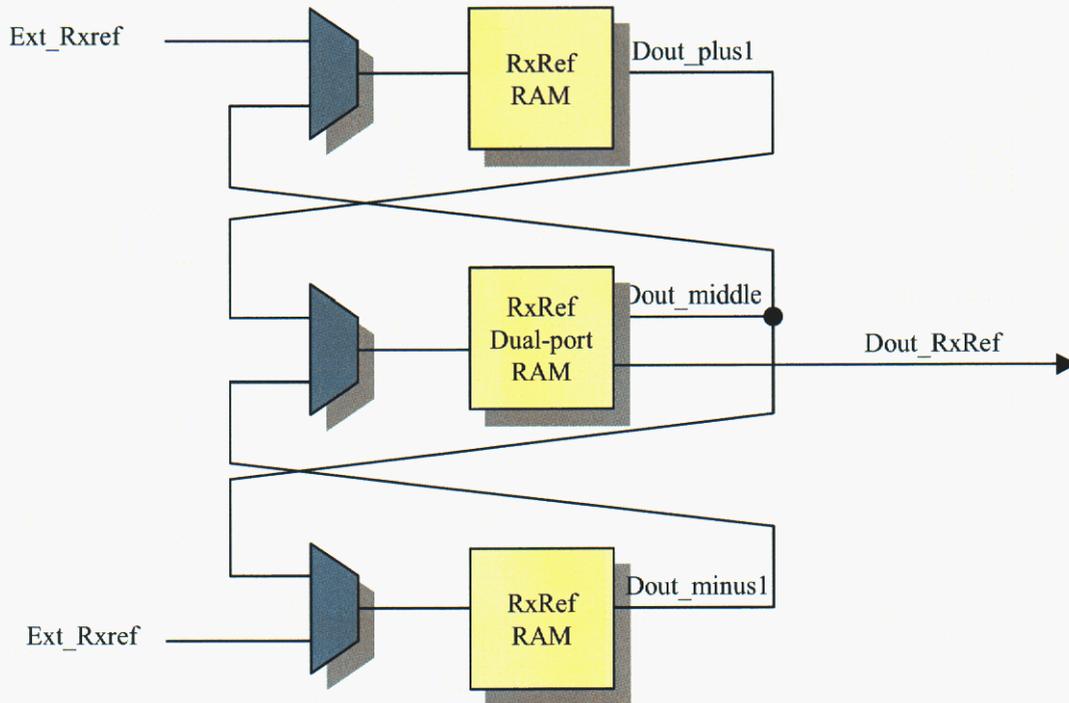


Figure 30. RxRef_blk implementation

A read-only port off the main buffer accesses *Dout_RxRef*. A read-write port on the same buffer is used to update the main buffer while transferring previous data to look-ahead or look-behind buffers.

In the case of advancing delay, the look-ahead buffer gets data from the middle buffer. The middle buffer simultaneously gets data from the look-behind buffer. The DLL update takes $N_{samp}+2$ clock cycles, roughly the length of the **Despread Symbol** stage where new data is immediately used. At this point, main buffers have been updated from local memory at the block boundary. But now either the look-ahead or look-behind buffer of each state needs to be updated from external memory. This update would also take $N_{samp}+2$ clock cycles. Since each iteration while tracing forward takes at least $6*N_{samp}$ cycles (8-states executed at a time for middle, early, and late correlations), there are enough clock cycles to update about 6 of these look-ahead or look-behind buffers in an iteration. With 5 iterations until buffers are needed for the next update, there are plenty of clock cycles to update 16 states can be sequentially updated with access to external memory.

5.7 SOVA2 implementation

The standard soft-output Viterbi decoder, as mentioned before, is not nearly as complex as the SOVA1 implementation. It shares the same *sova_delta_top* block and a similar recursive structure to the traceback function. Although it also has an iterative trace-forward computation, it does not require any complex multiplies for each information bit, as with SOVA1. Consequentially, the SOVA2 block is expected to contribute a small percentage to the Turbo decoders final latency and area. Synthesis results are summarized in the following section.

5.8 SOVA1 and SOVA2 size and speed results

Results from synthesis are as follows:

Table 11. Synthesis results for SOVA implementations

Module	#Flip Flops	#LUTs	#BRAMs	#MULT18x18	Max speed(MHz)
SOVA1 Top	46,183	60,637	158	144	63.1
Data Processor	47,152	55,377	0	144	73.0
Despread Symbols	11272	12232	0	144	126.6
Add Compare Select	7056	11448	0	0	85.1
Correlate and Update	21016	26200	0	0	73.0
Update Delay and Sums	5271	1184	0	0	241.5
Traceback	535	1032	0	0	150.6
Sova Delta Loop	2447	4729	0	0	98.6
Data Control	53	74	0	0	151.9
On-Chip Memory	27	6,990	158	0	174.9
Memory Control	49	176	0	0	180.8
SOVA2 Top	4,273	8,617	0	0	122.3
Sum SOVA1 and SOVA2	50,456	69,254	158	144	63.1
Xilinx VirtexII 2v8000	93,184	93,184	168	168	

The number of flip-flops and LUTs used in the top-level SOVA2 and SOVA1 modules and sub-modules are listed in Table 11. The sum total of these top-level component resources is then compared those available on the largest Xilinx Virtex-II part, the XC2V8000. According to these numbers, enough resources exist on this FPGA to implement both decoders on the same part. An interleaver and control unit for the SOVA1 and SOVA2 loop should fit on the remainder of the chip, although implementation of the entire Turbo decoder is not covered here.

The **Data Processor**, which makes up the bulk of the logic in SOVA1, is broken down further into sub-modules. The *despread_symbols*, *add_compare_select*, and *correlate_n_update* modules have been adjusted to reflect 8 instantiations of each. Note that the sum of the sub-module resources does not necessarily match those of the parent. Optimizations visible at higher levels in the design hierarchy allow the synthesis tool to minimize area. On the other hand, there may also be structures wrapped around sub-modules that add overhead resources to the parent.

Synthesis also estimates speed based on longest data paths within the design. Since resources have not been placed and routed, interconnect delay is not properly represented. The higher the utilization of the FPGA, the more significant this routing delay can become, thus the speed estimate is considered an upper bound. During verification, speed optimizations can be made along critical paths for improved timing performance. The maximum speeds for the SOVA1 and SOVA2 modules indicate that SOVA1 will be the bottleneck for the algorithm. With ideal routing, the part should run at 63 Mhz, but since this is hardly ever the case, we can expect the part to run slower. The design in its current form does not route using Xilinx's Project Navigator tools. Verification and

additional constraints will be necessary for computer-aided place and routing and final timing analyses. For our estimates, we use a clock speed of 50 Mhz.

Latencies for the SOVA1 and SOVA2 hardware implementations are summarized below.

Table 12. Clock cycle latencies for SOVA implementations

Data processor module	#clock cycles
SOVA1_top	6,327,213
Despread Symbols	2,080,288
Add Compare Select	10,040
Correlate and Update	4,170,617
setup complex mult	2,008
despread early estimate	2,080,288
update symbol sums	2,008
reset addresses	2,008
setup complex mult	2,008
despread late estimate	2,080,288
update symbol sums	2,008
chk block bounds	1
Update Delay and Sums	1,004
Traceback	2,008
Sova Delta Loop	63,256
SOVA2_top	70,284
Traceforward	5,020
Traceback	2,008
Sova Delta Loop	63,256
Interleavers (approx.)	1,004
Latency #cycles per 10 iterations	6.40E+07
Bit rate @ 50 Mhz(bps)	785
Chipping rate @ 50 Mhz (Mcps)	1.2

The overall latency of 64M clock cycles is the sum of latencies for SOVA1, SOVA2, and interleavers multiplied by 10, which is the number of times that the algorithm iteratively refines its output. Since the entire Turbo decoder and its interleavers have not been implemented, the interleaver and top-level latencies are approximate. Using a 50 Mhz clock a 1000 bit message can be calculated in 1.3 seconds. If 1000-bit messages were received back-to-back, 785 bps could be processed without overrunning the SOVA1 decoder. This would correspond to a 1.2 Mcps chipping rate.

The SOVA1 is shown to contribute to most (98.9%) of the overall latency for the Turbo decoder. The following trade-off analysis is done in terms of the *SOVA1_top* latency.

SOVA1_top latency is dominated by complex multiplies in the trace-forward loops of the data flow. Data and memory controls occur in parallel and do not contribute to overall latency. The latency of the complex multiplies grows with *N_{samp}*:

$$Latency_{comp_mult} = 6 * (Nsamp + Latency_{multiplier}) * NUM_BITS .$$

The latency in the multiplier, $Latency_{multiplier}$, in this case is 12 cycles. Complex multiplies are performed once during **Despread Symbols** and twice for early and late estimations during **Correlate and Update** stages. Each of these stages are repeated for each iteration.

The latency for hardware not dependant on $Latency_{comp_mult}$ is 86,349 clock cycles. The higher $Nsamp$ is, the closer it approximates the entire latency of the SOVA1 algorithm. Since $Nsamp$ is 1024 (512 chips per symbol and 2 samples per chip), the latency for complex multiplies in this implementation comprises 98.6% of **SOVA1_top** latency. Halving $Nsamp$ saves roughly half (48.7%) the overall **SOVA1_top** latency. Doubling $Nsamp$ would nearly double (197%) **SOVA1_top** latency. In this case, however, additional off-chip memory may be needed since the depth of $RxRef$ buffers also depend on $Nsamp$.

Increasing $Nsamp$ by increasing the number of chips per symbol may also require increasing maximum bus width allowed for signals. More multiply-accumulates means a greater chance of causing an overflow in the final result and a larger multiply-accumulator may be necessary. Other hardware and memory blocks would also need to accommodate the larger bus widths.

$Latency_{comp_mult}$, furthermore, depends on NUM_BITS . Raising the number of bits from 1000 to 10000 would increase latency another order of magnitude. Increasing number of bits per message would increase memory requirements in Table 10 and available BRAM on the Xilinx device may become a limiting factor. Thus, custom interfaces to off-chip RAM will likely be needed for messages longer than 1000 bits.

Changing constraint length would also affect the latency of the system. Constraint length, K , determines the number of states in a trellis. In this implementation, $K=5$ was chosen to target the largest Xilinx Virtex-II device. With this implementation, however, not all 16 states could be computed at once with resources on an FPGA. Lowering the constraint length to 3 would create a trellis of 4 states, and **despread_symbol**, **add_compare_select**, and **correlate_n_update** blocks could be instantiated 4 times instead of 8 for $K=5$. Not only would this save space on the FPGA, but all states could then be computed in parallel. This would lower overall **SOVA1_top** latency by 49.4%.

Another way to decrease the latency by 33% is to compute early and late summations for correlation in parallel. Even if reducing the number of dedicated 18x18 multipliers required for each complex multiply to 4 from 6, however, 192 multipliers would still be needed to compute 6 symbols in parallel for correlation. This number of multipliers is still not available on Xilinx parts, and again, there are not enough LUTs available on the chip to implement the extra multiplies. If we expect the Turbo decoder to operate on punctured code most of the time, however, we could achieve the same savings by implementing complex multipliers for just the first and second symbols. This would require only 128 dedicated 18x18 multipliers. But, if codes are not punctured, the savings are lost.

Presented here is essentially a “paper” design since no part of this design has yet been simulated and verified. The design, however, is written in synthesizable VHDL and therefore is ready to be tested against the algorithm written in C. Software should be modified to use fixed-point representations for verification with hardware. The software should also be modified to generate test vectors in intermediate points in the program for module-by-module verification. Having a bit-accurate software model would also simplify tradeoff analyses and allow design decisions to be made at a higher level before implementing a new hardware design. Using an environment such as SystemC would co-simulate hardware and software implementations. This would enable module by module testing of the hardware using a C-based testbench and speed up simulation and verification times.

This design targets the largest Xilinx Virtex-II FPGA available, the XC2V8000, which is equivalent to 8 million system gates. Its data rates target the region where the Turbo algorithm has demonstrated its strongest gains for 3 to 5 dB in E_b/N_o over typical systems used today. Using constraint length of 5, a code rate of 1/3, message length of 1000 bits, and 512 chips per symbol, we were able to synthesize a design on a single FPGA that would operate at a bit rate of 785 bps and a chipping rate of 1.2 Mcps.

6 Software Implementation

6.1 sova_dppll.c

The function **sova_dppll** is a soft output (and soft input) Viterbi algorithm (SOVA) decoder with integrated delay and phase lock loops for decoding convolutionally encoded (possibly as a constituent code in a Turbo code) spread spectrum BPSK signals. The encoder shift register connections are set in header file **scenario.h** as are other parameters such as constraint length, K , the number of data bits, the number of spread spectrum chips per symbol, the number of A/D samples per chip, etc. The decoder operation is effected by the boolean function parameter *TurboMode*, which indicates a turbo code or a simple convolutional code. **Sova_dppll** can be called by **main** in the file **run_sova_dll.c** with *TurboMode* set *False* or **main** in **run_turbo_dll.c**, via **turbo_dppll**, with the *TurboMode* flag set *True*. Note that the Turbo code can be punctured or not punctured (set in **scenario.h**) giving a rate 1/2 or a rate 1/3 Turbo code respectively, but the two constituent convolutional codes are fixed rate 1/2 codes. The convolutional encoders are recursive systematic convolutional (RSC) encoders meaning that the input data bit is one of the output channel symbols, called the “systematic” symbol, while the parity symbol is computed with the recursive structure shown in Figure 31, showing a binary shift register and mod-2 adders.

The RSC encoder block diagram is shown in Figure 31. In the file **encode_rsc.c** the initial state of the encoder is set to **0**, i.e. all registers are cleared. The feedback bit, F_b , is the mod-2 sum of the state of the encoder after shifting in the input bit, 0 or 1, *and*-ed with the register connection vector, g_j . This mod-2 sum is computed by using the result of the *and* as an index into *Partab*. This table performs the mod-2 sum by mapping all indices whose binary expansions have an odd number of 1’s to an output of 1 and all indices with an even binary representation to an output of 0. For example indexes of 1 and 2 are 01 and 10 in base 2 respectively and map to 1 while an index of 3, which is 11, maps to 0. The F_b bit replaces the input bit to create *state2* which is *and*-ed with g_2 to produce the parity bit output for this input bit. If the *Terminate* flag is *True* the last $K-1$ data bits are changed to values that force F_b to 0 thereby driving the final state to the **0** state. Since the tail bits can be overwritten no data is ever put there, the tail bits are initially all 0. The even outputs are simply the input bits, called the systematic symbols, and the odd outputs are the parity symbols.

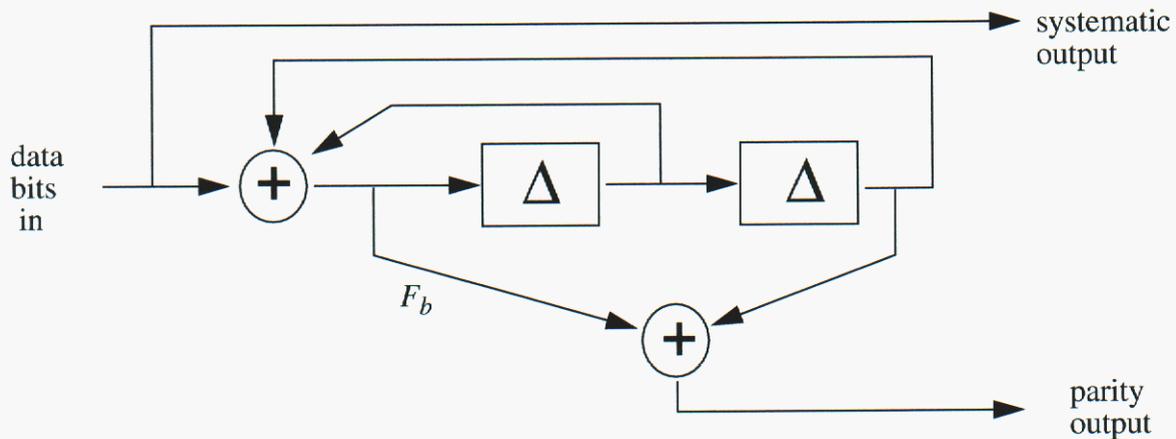


Figure 31. RSC encoder block diagram ($K=3$ example with $g_1 = 111$ and $g_2 = 101$)

The function `encode_turbo` calls `encode_rsc` twice as shown in Figure 32. The data is shown passing through the RSC1 block because a terminating tail is added to the data by RSC1 as described above. The data and tail are then interleaved by the block marked “ α ” and are encoded by RSC2. Note that the tail terminates RSC1 to the **0** state, but not RSC2, since interleaving scrambles the data and tail bits. The data bits (with tail), d_i , and the parity outputs from RSC1, p_i , and RSC2, q_i , are then multiplexed into a single stream. If the *Puncture* flag is *False* then the multiplexer output is $d_1, p_1, q_1, d_2, p_2, q_2 \dots$ or if the *Puncture* flag is *True* the multiplexer output is $d_1, p_1, d_2, q_2, d_3, p_3, d_4, q_4 \dots$, in other words all the data bits, or systematic symbols, are sent but only half of the parity symbols are sent. Figure 32 also shows the direct sequence spread spectrum (DSSS) chip sequence being generated and multiplied onto the encoder channel symbols.

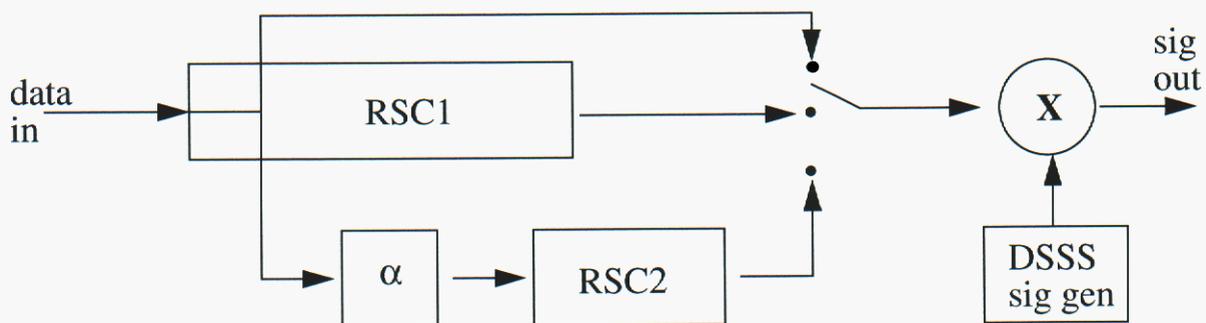


Figure 32. Turbo encoder

The binary signal leaving Figure 32 is impressed on an RF carrier (BPSK modulated) which is then received, converted to baseband and passed to the decoder shown in Figure 33. If `soval_dpll` is processing simple RSC encoded data, so that *TurboMode* is *False*, indicating that `encode_rsc`

was called directly by **main** rather than by **encode_turbo** the tail may be either terminating or nonterminating, and the appropriate decoder is just the Enhanced SOVA1 block.

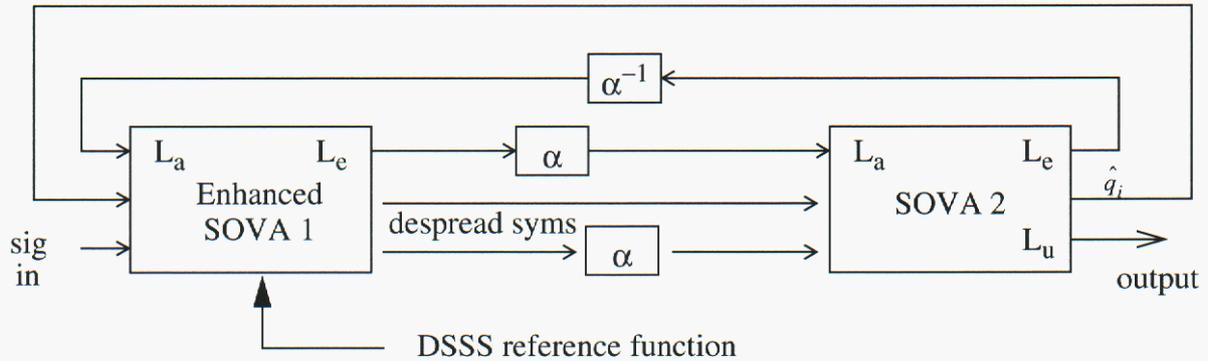


Figure 33. Turbo decoder with integrated phase and delay tracking loops in SOVA 1

For comparison, in Figure 34, we give the block diagram for a standard Turbo decoder that could be used following a standard “external” delay and phase locked loop (DPLL), as implemented in **run_dppll_then_turbo.c** and the functions that this main calls. In this decoder the phase-corrected, time-aligned, despread channel symbols are demultiplexed sending the systematic bits, d_i , and RSC1 parity bits, p_i , to decoder 1. The systematic bits are passed through an interleaver to put them in the order in which they were presented to RSC2 in Figure 32. The parity symbols are already in the interleaved order so they are passed to SOVA2 directly. The extrinsic information, L_e , generated by SOVA2 is deinterleaved to match the bit order for SOVA1 which is then used as a priori information, L_a . Extrinsic information computed by SOVA1 is interleaved for SOVA2 which works with interleaved data.

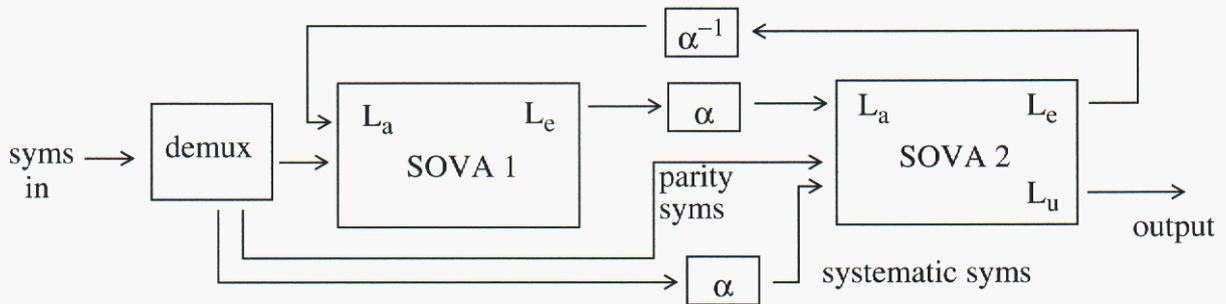


Figure 34. Standard Turbo decoder

Figure 33 has all of the elements in Figure 34 plus the additional DPLL functionality inside SOVA1. The symbol demultiplexing occurs after the spreading code removal which is part of the

DPLL function so the demultiplexing must occur within (can not precede) SOVA1 as well. The additional line running from SOVA2 to SOVA1, labeled \hat{q}_i , represents the hard parity symbol estimates that SOVA2 produces during the decoding process. These hard parity symbol estimates from SOVA2 are used by the “data-aided” tracking loops in SOVA1 along with hard parity and hard systematic symbol estimates from SOVA1 itself. (Hard symbols are +1 or -1, as opposed to soft symbols which are noisy.)

With all of the foregoing background material we are now ready to look at the coding of the enhanced SOVA decoder in the file `sova_dppll.c`. We assume a preamble detector has given us the initial DLL delay, *InitDelayEst*, and put the beginning of the signal squarely in the I (in-phase) channel. Keep in mind that in *TurboMode* there will be parity symbols from RSC2 that the Viterbi algorithm in SOVA1 will not use but the despreading and loop algorithms will use.

On the first call to `sova_dppll` the function `gen_tables` is called. This function generates the previous state and previous symbol tables, which for a given state and a given data bit, gives the previous state and the parity symbol associated with the transition between these two states. In `gen_tables` each state defines a bit pattern in the shift register to which both possible bit inputs, 0 and 1, are shifted. The feedback bit, F_b , is the mod-2 sum of the state of the encoder after shifting in the input bit, 0 or 1, *and*-ed with the tap code vector, g_1 . The mod-2 sum is computed by using the result of the *and* as an index into *Partab*. This table performs the mod-2 sum by mapping all indices whose binary expansion has an odd number of 1's to an output of 1 and all indices with an even binary representation to an output of 0. The F_b bit replaces the input bit to create *state2* which is *and*-ed with g_2 to produce the parity symbol output for this initial state and input bit. The *nextstate* for this initial state and input bit are stored too and used to generate the *prevstate* arrays. The *prevsym0* and *prevsym1* arrays give both the systematic symbol, which is the input data bit itself, and the parity symbols calculated above. Adding the systematic bit that precedes the parity bit is accomplished by adding 10 (binary) to *parity1* to give *prevsym1*.

The decoder begins by looping over the encoded bits, or equivalently the received symbol groups. There are 2 symbols per group for a punctured Turbo code, 1 systematic and 1 parity symbol. Recall the earlier description of the multiplexed output from `encode_turbo` where symbols 0,2,4... are systematic symbols, while symbol 1,3,5... are parity symbols, alternately from RSC1 and RSC2. The punctured parity symbols are simply not transmitted, the RF energy is put into the remaining symbols. (Punctured codes use less BW for a given data rate but provide less coding gain.) There are three symbols per group in an unpunctured Turbo code, the systematic bit and the parity from each of the two RSC encoders. If *TurboMode* is *False* the decoder expects a simple rate 1/2 RSC code with each symbol group having 2 symbols, the systematic symbol and the accompanying parity symbol.

For each bit (or symbol group) we calculate the index of the symbols, *index1*, *index2*, and *index3*, if applicable. These indexes are directly used with *RxSig* and are multiplied by the reference oversampling factor, *OS*, and offset by DLL delay, *D*, for the reference function. Over sampling of the reference allows the alignment of the signal and reference in the despreading correlators to be adjusted in small steps. Smaller steps gives better alignment and less scalloping loss.

We loop through each *state* despreading and phase unwrapping the signal. Note that the phase and delay estimates are state dependent, as calculated below, except for the first bit (first time step) where *D* is *InitDelayEst* for all states and *ThetaEst* is 0 for all states. The result is despread soft symbols that are saved in *SysSymMat[state][t]*, *ParSym1Mat[state][t]*, and *ParSym2Mat[state][t]*, from which the best soft symbols will be recovered during the traceback when we are finding the best path through the trellis. We also calculate and store the complex despread symbols for the current time step. These are stored in *DespreadSymRe[0-2][state]* and *DespreadSymIm[0-2][state]*. The first index gives the symbol number within the group, so *DespreadSymRe[0][state]* is the systematic symbol and *DespreadSymRe[1][state]* is a parity symbol (RSC1 parity for unpunctured or alternating RCS1/RSC2 for punctured). For an unpunctured Turbo code the RSC2 parity is stored in *DespreadSymRe[2][state]*. Think of *state*, the state at which we know *D* and *PhaseEst*, as being the node to the left of the transitions in the trellis. This concludes this loop over *state*.

Now in a second loop over state we do the “add-compare-select” operation, thinking of *state* now as the nodes on the right side of the transitions in the trellis, i.e. these are the states that we are trying to propagate parameters to. For each *state* we find which previous state, *state0*, we would have come from if the current bit were a 0, and which state, *state1*, if the bit were a 1. Likewise we get the hard symbol pairs, *Syms0* and *Syms1*, associated with the transitions between these particular states. Each of these holds exactly 2 symbols since the RSC encoders are rate 1/2 encoders. We split the two symbols out of *Syms0* and *Syms1* by masking with 2 (binary 10) and 1 (binary 01) and setting *sym1* and *sym2* to 1 or -1. These hard symbols are used to multiply the despread soft symbols to remove the BPSK modulation to give the transition metrics associated with the two possible data bit values, 0 or 1. The transition metrics are stored in variables with a base name of the form *TranMet*. We similarly remove the BPSK modulation from the RSC2 symbols using hard symbol estimates from SOVA2 which are stored in *s_hat[t]*, storing the results in variables with a base name of *Sova2energy*. The variable *s_hat* is produced by SOVA2 for the enhanced SOVA1 and consists of the interleaved systematic symbol estimates and the parity symbol estimates, \hat{q}_i , which is shown in Figure 33. Only the parity symbol estimates in *s_hat* are used at present. The *TranMet* variables are added to the previous state’s total path metric, *PathMetric[state0/state1][t]*, plus the a priori info, *La[t]*, to get the data-dependent total path metrics at the new state, *m0* and *m1*. These two quantities are then compared and the largest gives the assumed path into the new state. The *Sova2energy* variables are added to the transition metrics to form complex PLL integrators, *MX*, that span all of the symbols in the group.

If *m0* > *m1* then the decision is that the current bit is a 0 and the path metric calculated for the path associated with the 0 bit, *m0*, is stored in *PathMetric[state][t+1]*. The DLL delay, *D_ML*, is copied from *state0* as are the PLL variables. The PLL accumulators, *MX_re* and *MX_im*, are updated with the despread soft symbols after the BPSK modulation has been removed by multiplying them by the hard symbol estimates *sym1* and *sym2*. Else if *m1* > *m0* the current bit is decoded as a 1 and info from *state1* is propagated to the current state.

The next section of code are the early and late correlators. Here the spread spectrum signal and reference are again multiplied and summed as in the first state loop above except that now the reference is shifted early and late by *DeltaDll*. The all symbols in the Turbo code are included in the

integrators by using hard symbol estimates from both SOVA1 and SOVA2, s_{hat} . The soft symbol outputs from the correlators and multiplied by the hard symbol estimates to remove the BPSK modulation, before the products are added together.

The PLL and DLL quantities are propagated along these paths from previous state to current state at time step t so that the current states become the old states at $t+1$. The DLL and PLL are updated rather than just propagated when t is divisible by $BlockLen$. At a block boundary the DLL delay for this state is increased or diminished by 1 reference sample depending on whether the magnitude of the Early gate is larger or smaller than the magnitude of the Late gate. The PLL is also updated at block boundaries: residual phase is measured and added to the summer $t2$, which is added to the summer $t3$. The phase rate, $Delta_phi$, and model phase, phi_m , are then calculated.

The phase correction, phi_nco , is calculated as the model phase minus one half of the phase change across a block (i.e. model phase is referenced to center of block) plus the phase rate per sample times the sample number, $ramp$. $Delta_phi$ is the phase rate per block so $Delta_phi/BlockLen$ is the phase rate per sample.

This ends the second loop over $state$.

The variable $ramp$, is the sample number in the block so is reset when $t==BlockLen$ and incremented otherwise.

This ends the loop over t , the forward pass through the RSC code trellis.

We trace back through the trellis to find the max-likelihood path and transmitted data. The Enhanced SOVA decoder will have a terminating tail in a Turbo code scheme. If the boolean flag $Term$ is true we know that the transmitted symbols drove the encoder state to $\mathbf{0}$ at the end of the message. Hopefully the $\mathbf{0}$ state has the largest $PathMetric$ at the final time step but may not due to noise. If $Term$ is true we set $beststate[NUM_BITS] = 0$, otherwise we search through the final path metrics looking for the state with the largest path metric. From there we trace backwards using the $prev_bit$ array, and knowing this we can get the previous state. The previous bit for each state was stored in bit packed form in a 64-bit integer at each time step. (There are 64 states in a constraint length 7 code so this is the maximum constraint length for this code at this time.) When the data bit decision was for a 1, a 1 was *or*-ed into the integer at the correct bit position for the current state and this was repeated for each state. After packing an integer in this fashion at each time step we have $prev_bit[t]$ from which we can extract our bit decisions using the mask $bitmap$ which was used to *or* the bits into position. $bitmap[state]$ has a single bit set at the bit position $state$. To unpack a bit decision at the state $beststate$ we *and* $prev_bit$ with a mask which has a single bit in the $beststate$ position and then right-shift the result into the LSB. Given the current state and the previous bit decision we get the previous state and continue to trace back through the trellis making the array of bit decisions. We also collect the despread soft symbols that go with the bit decisions. Some of these will be processed by SOVA2 in a Turbo code.

The final section of code finds the minimum path metric difference between the ML path and any other path that produces an opposite bit decision. For each bit decision we check all possible paths back as far $SOVA_DELTA$, from **scenario.h**, time steps. At each stage the path deviation is found

by choosing the incorrect bit i.e., the opposite bit from the ML path from *est_bits* and *beststate*. Path metric differences were stored in *Mdiff* as they were computed in the forward trace. The final line of code in this module assigns a polarity to the minimum path metric that is based on the bit value, i.e. we create soft bipolar decisions from the hard bit decisions.

6.2 Code listing

Although many other files are required to build the executable images, listings of only the following files, specifically discussed in Chapters 5 or 6, are included in this section:

- scenario.h
- sova_dpll.c
- turbo_dpll.c
- encode_rsc.c
- gen_tables.c
- encode_turbo.c
- run_turbo_dll.c
- turbo.c
- sova.c
- dpll.c
- run_dpll_then_turbo.c

```

// scenario.h
//
// simulation parameters and constants that determine array sizes, etc that
// must be fixed at compile time
//
// Richard C. Ormesher, Jeff Mason, SNL dept. 2344, 5/1/02

// #include <limits.h>          // for INT_MAX = 2^31 - 1

#define min(a,b) ( ((a)<(b)) ? (a) : (b) )
#define round(a) ( floor(0.5+(a)) ) // like Matlab for a=0.5, not for a=-0.5

#if 1                          // choose fixed or floating point
    #define NUMTYPE double
#else
    #define NUMTYPE int
    #define FIXED
#endif

#define CONSTRAINT_LENGTH 5    // max of K=7 for long long prev_bit
#define PUNCTURE 0            // set to 0 for R=1/3, set to 1 for R=1/2
#define NUM_INFO 1000         // num of data bits before tail is added
#define NUM_CHIP_PER_SYM 127 // set chipping rate relative to sym rate
#define NUM_SAMP_PER_CHIP 3   // A/D sample rate
#define REF_OVER_SAMP 8       // reference function oversampling factor
#define TAIL_SYM 10           // allows for DLL movement & E-L gate overhang

// The following constants are used by the sova decoder and are not scenario
// dependent. They are currently set to match sova0.m for now.

#define SOVA_DELTA 30
#define BIG_NEG (-1000000)
// #define BIG_POS INT_MAX
#define BIG_POS 1000000 // give same results as Matlab for all SOVA_DELTA

// *** derived constants - no changes are necessary below this line ***

// RSC_G_1 is the tap code for the feedback bit in the Recursive Systematic
// Convolutional encoder and RSC_G_2 is the tap connection vector for the
// parity output. The LSB is the input so RSC_G_1 must be an odd number, ie
// g11=1 or the input data is not used immediately.

#if CONSTRAINT_LENGTH == 7
    #define NUM_STATES 64          // 2^(K-1)
    #define RSC_G_1 0x73          // code 1, used by decoder
    #define RSC_G_2 0x59          // code 2, used by decoder
#elif CONSTRAINT_LENGTH == 5
    #define NUM_STATES 16         // 2^(K-1)
    #define RSC_G_1 0x19          // CCSDS
    #define RSC_G_2 0x1b          // CCSDS
    // #define RSC_G_1 0x1f          // Sklar "Digital Communications"
    // #define RSC_G_2 0x11          // Sklar 2nd Ed. pg. 496
#else // CONSTRAINT_LENGTH == 3

```

```

#define NUM_STATES 4           // 2^(K-1)
#define RSC_G_1 0x7           // code 1, used by decoder
#define RSC_G_2 0x5           // code 2, used by decoder
#endif

#define NUM_BITS (NUM_INFO+CONSTRAINT_LENGTH-1) // includes tail bits

#if PUNCTURE
#define NUM_SYM_PER_BIT 2      // R=1/2
#else
#define NUM_SYM_PER_BIT 3      // R=1/3
#endif

#define CODE_RATE (1.0/(double)NUM_SYM_PER_BIT)
#define NUM_SYMS (NUM_BITS*NUM_SYM_PER_BIT) // num channel syms in msg
#define NUM_SAMP_PER_SYM (NUM_SAMP_PER_CHIP*NUM_CHIP_PER_SYM)
#define NUM_SAMP_IN_REF_TAIL (TAIL_SYM*NUM_SAMP_PER_SYM*REF_OVER_SAMP)
#define NUM_SAMP_IN_SIG (NUM_SYMS*NUM_SAMP_PER_SYM)
#define NUM_SAMP_IN_REF
(REF_OVER_SAMP*NUM_SAMP_IN_SIG+NUM_SAMP_IN_REF_TAIL)
#define NUM_STATES_2 (NUM_STATES/2) // half the states

```

```

// sova_dpll.c
//
// soft output Viterbi decoder with integrated DLL and PLL
//
// Richard C. Ormesher, Jeff Mason, SNL dept. 2344, 9/5/02

#include <stdio.h>
#include <stdlib.h>          // for exit()
#include <math.h>           // for M_PI, cos, sin, atan2
#include "scenario.h"       // simulation constants
#include "gen_tables.h"
#include "sova_dpll.h"     // make sure proto is up to date

#ifdef MATLAB_MEX_FILE
#include "mex.h"           // for mexPrintf()
#endif

static NUMTYPE MX_re[NUM_STATES];
static NUMTYPE MX_im[NUM_STATES];
static NUMTYPE MX_prev_re[NUM_STATES];
static NUMTYPE MX_prev_im[NUM_STATES];
static NUMTYPE t2[NUM_STATES], t2Prev[NUM_STATES];
static NUMTYPE t3[NUM_STATES], t3Prev[NUM_STATES];
static NUMTYPE phi_m[NUM_STATES], phi_mPrev[NUM_STATES];
static NUMTYPE Delta_phi[NUM_STATES], Delta_phiPrev[NUM_STATES];
static NUMTYPE DespreadSymRe[3][NUM_STATES];
static NUMTYPE DespreadSymIm[3][NUM_STATES];
static NUMTYPE SymSumRe[3][NUM_STATES];
static NUMTYPE SymSumPrevRe[3][NUM_STATES];
static NUMTYPE ParSym1Mat[NUM_STATES][NUM_BITS];
static NUMTYPE ParSym2Mat[NUM_STATES][NUM_BITS];
static NUMTYPE SysSymMat[NUM_STATES][NUM_BITS];
static NUMTYPE Mdiff[NUM_STATES][NUM_BITS+1];
static NUMTYPE PathMetricCurr[NUM_STATES];
static NUMTYPE PathMetricNext[NUM_STATES];

static int prevsym0[NUM_STATES];
static int prevsym1[NUM_STATES];
static int prevstate0[NUM_STATES];
static int prevstate1[NUM_STATES];
static int beststate[NUM_BITS+1], beststate_tb;
static int tables_ready = 0;
static int SymVal[4][2] = { {-1,-1}, {-1,1}, {1,-1}, {1,1} };

static unsigned char est_bits[NUM_BITS+1], error_bit;

static unsigned long long prev_bit[NUM_BITS+1]; // non-portable ?
static unsigned long long bitmap[NUM_STATES]; // non-portable ?

int sova_dpll( NUMTYPE *RxSigRe, NUMTYPE *RxSigIm, NUMTYPE *La,
              NUMTYPE *Lall, int Term, NUMTYPE *RxRef, int InitDelayEst,
              int BlockLen, NUMTYPE *DespreadSyms, int *s_hat,
              int TurboMode, int dll_step, int pll_flag, NUMTYPE *pll_params)

```

```

{

int i, j, t, state, state0, state1;
int Syms0, Syms1, SymsML, SymML1, SymML2, SymML3=0;
int Puncture = PUNCTURE;
int Nsamp = NUM_SAMP_PER_SYM;
int Nstates = NUM_STATES;
int Nbits = NUM_BITS;
int Os = REF_OVER_SAMP;
int DeltaDll = REF_OVER_SAMP;
int DelayEst[NUM_STATES];
int DelayEstNext[NUM_STATES];
int D, D_ML, stateML;
int index1, index2, index3=0, ramp=0;
int index3Max;
int sym1, sym2;

NUMTYPE m0, m1, llr, bestmetric;
NUMTYPE ThetaEst;
NUMTYPE TwoPi = 2.0 * M_PI;    // from math.h
NUMTYPE phi_nco[NUM_STATES];
NUMTYPE delta_phi[NUM_STATES];
NUMTYPE sym1re, sym1im, sym2re, sym2im, sym3re=0, sym3im=0;
NUMTYPE term1re, term1im, term2re, term2im, term3re, term3im;
NUMTYPE re1, im1, re2, im2;    // working variables for doing complex ops
NUMTYPE TranMet0Re, TranMet0Im, TranMet1Re, TranMet1Im;
NUMTYPE Sova2energy0Re=0, Sova2energy0Im=0;
NUMTYPE Sova2energy1Re=0, Sova2energy1Im=0;
NUMTYPE K1=pll_params[0];
NUMTYPE K2=pll_params[1];
NUMTYPE K3=pll_params[2];

// FILE *fpdebug;

// open debug file
// fpdebug=fopen("debugfile", "w");
// fprintf(fpdebug, "%d \n", RxSig[50] );

if (0) {
printf("** RxSigRe[0]=%f \n", RxSigRe[0] );
printf("** RxSigIm[0]=%f \n", RxSigIm[0] );
printf("** RxSigRe[End]=%f \n", RxSigRe[NUM_SAMP_IN_SIG-1] );
printf("** RxSigIm[End]=%f \n", RxSigIm[NUM_SAMP_IN_SIG-1] );
printf("** La[0]=%f \n", La[0] );
printf("** La[Nb]=%f \n", La[Nbits] );
printf("** Term=%d \n", Term);
printf("** RxRef[0]=%f \n", RxRef[0] );
printf("** RxRef[End]=%f \n", RxRef[NUM_SAMP_IN_REF-1] );
printf("** InitDelayEst=%d \n", InitDelayEst);
printf("** BlockLen=%d \n", BlockLen);
printf("** s_hat[0]=%d \n", s_hat[0] );
printf("** TurboMode=%d \n", TurboMode);
printf("** dll_step=%d \n", dll_step);
}

```

```

printf(" * pll_flag=%d \n", pll_flag);
printf(" * K1=%f, K2=%f, K3=%f \n", K1, K2, K3);
}

if (TurboMode && !Puncture)
    index3Max = 3*Nsamp*Nbits;
else
    index3Max = 2*Nsamp*Nbits;

if (index3Max > NUM_SAMP_IN_SIG) {
    printf("sova_dppll: index3 will over-run array RxSig -- aborting\n");
    exit(1);
}

if (index3Max*Os+InitDelayEst > NUM_SAMP_IN_REF) {
    printf("sova_dppll: index3 will over-run array RxRef -- aborting\n");
    exit(2);
}

if (!tables_ready) { // init bitmap on first call
    tables_ready=1;
    gen_tables( prevstate0, prevstate1, prevsym0, prevsym1);
    for (i=0, bitmap[0]=1; i<NUM_STATES-1; i++)
        bitmap[i+1] = bitmap[i] << 1;
}

// initialize vars
for(state = 0; state<NUM_STATES; state++)
{
    PathMetricCurr[state] = BIG_NEG; // set to large neg number
    DelayEst[state] = InitDelayEst; // init delay for all states
    phi_nco[state]=0;
    SymSumPrevRe[0][state] = 0;
    SymSumPrevRe[1][state] = 0;
    SymSumPrevRe[2][state] = 0;
    MX_prev_re[state]=0;
    MX_prev_im[state]=0;
    t2Prev[state]=0;
    t3Prev[state]=0;
    phi_mPrev[state]=0;
    Delta_phiPrev[state]=0;
}

PathMetricCurr[0] = 0; // start with state 0 */

/* Trace forward for t = 0 to NUM_BITS */
for (t=0; t<Nbits; t++)
{
    // printf(" %d \n",t+1);

    // calculate indexes into RxSig and RxRef for the t-th bit's symbols
    if (TurboMode && !Puncture) {
        index1 = 3*Nsamp*t;

```

```

index2 = index1+Nsamp;
index3 = index2+Nsamp;
} else {
index1 = 2*Nsamp*t;
index2 = index1+Nsamp;
}

for (state = 0; state < Nstates; state++) {

D = DelayEst[state]; // get projected delay estimate for this state

if (pll_flag)
ThetaEst = TwoPi*phi_nco[state];
else
ThetaEst = 0; // turn PLL off, assume sig is real

// printf("ThetaEst=%f \n", ThetaEst );
re2 = cos( ThetaEst); im2 = -sin( ThetaEst); // pre-compute for loops

// despread first symbol
for ( i=0, sym1re=sym1im=0 ; i< Nsamp; i++) {
re1 = RxSigRe[index1+i] * RxRef[index1*Os+D+i*Os];
im1 = RxSigIm[index1+i] * RxRef[index1*Os+D+i*Os];
sym1re += re1*re2-im1*im2; // term1 = x1 * x2 (complex)
sym1im += re1*im2+re2*im1;
}

SysSymMat[state][t]=sym1re;

// despread second symbol
for ( i=0, sym2re=sym2im=0 ; i< Nsamp; i++) {
re1 = RxSigRe[index2+i] * RxRef[index2*Os+D+i*Os];
im1 = RxSigIm[index2+i] * RxRef[index2*Os+D+i*Os];
sym2re += re1*re2-im1*im2;
sym2im += re1*im2+re2*im1;
}

ParSym1Mat[state][t]=sym2re;

if (TurboMode && !Puncture) { // despread third symbol
for ( i=0, sym3re=sym3im=0 ; i< Nsamp; i++) {
re1 = RxSigRe[index3+i] * RxRef[index3*Os+D+i*Os];
im1 = RxSigIm[index3+i] * RxRef[index3*Os+D+i*Os];
sym3re += re1*re2-im1*im2;
sym3im += re1*im2+re2*im1;
}
ParSym2Mat[state][t]=sym3re;
}

// save despread sym, each state has its own delay and phase estimates

DespreadSymRe[0][state] = sym1re;
DespreadSymIm[0][state] = sym1im;
DespreadSymRe[1][state] = sym2re;

```

```

DespreadSymIm[1][state] = sym2im;
if (TurboMode && !Puncture) {
    DespreadSymRe[2][state] = sym3re;
    DespreadSymIm[2][state] = sym3im;
}
}

/* for stage t+1 set all bits at each state to zero */
prev_bit[t+1] = 0;

for (state = 0; state < Nstates; state++) {

    /*** perform add-compare-select operation for SOVA # 1 ***
    // first compute metrics for a 0 data bit

    state0 = prevstate0[state]; // get prev state associated with info bit=0
    Syms0 = prevsym0[state]; // get sym pair associated with info bit=0

    sym1 = (Syms0&2) ? 1 : -1; sym2 = (Syms0&1) ? 1 : -1; // split out syms

    sym1re = DespreadSymRe[0][state0]; // get 1st despread sym from state0
    sym1im = DespreadSymIm[0][state0];
    sym2re = DespreadSymRe[1][state0]; // get 2nd despread sym from state0
    sym2im = DespreadSymIm[1][state0];
    if (TurboMode && !Puncture) {
        sym3re = DespreadSymRe[2][state0];
        sym3im = DespreadSymIm[2][state0];
    }

    if (TurboMode) {
        if (Puncture) {
            if (t%2) { // parity punctured
                TranMet0Re = sym1 * sym1re;
                TranMet0Im = sym1 * sym1im;
                Sova2energy0Re = s_hat[2*t+1]*sym2re;
                Sova2energy0Im = s_hat[2*t+1]*sym2im;
            } else { // parity is available
                TranMet0Re = sym1 * sym1re + sym2 * sym2re;
                TranMet0Im = sym1 * sym1im + sym2 * sym2im;
                Sova2energy0Re = 0;
                Sova2energy0Im = 0;
            }
        } else { // unpunctured, third sym is parity for SOVA #2
            TranMet0Re = sym1 * sym1re + sym2 * sym2re;
            TranMet0Im = sym1 * sym1im + sym2 * sym2im;
            Sova2energy0Re = s_hat[2*t+1] * sym3re;
            Sova2energy0Im = s_hat[2*t+1] * sym3im;
        }
    } else { // not a Turbo code, simple conv code only
        TranMet0Re = sym1 * sym1re + sym2 * sym2re;
        TranMet0Im = sym1 * sym1im + sym2 * sym2im;
        Sova2energy0Re = 0;
        Sova2energy0Im = 0;
    }
}

```

```

}

// next compute metrics for a 1 data bit

state1 = prevstate1[state]; // get prev state associated with info bit=1
Syms1 = prevsym1[state]; // get sym pair associated with info bit=1

sym1 = (Syms1&2) ? 1 : -1; sym2 = (Syms1&1) ? 1 : -1;

sym1re = DespreadSymRe[0][state1]; // get 1st despread sym from state1
sym1im = DespreadSymIm[0][state1];
sym2re = DespreadSymRe[1][state1]; // get 2nd despread sym from state1
sym2im = DespreadSymIm[1][state1];
if (TurboMode && !Puncture) {
    sym3re = DespreadSymRe[2][state1];
    sym3im = DespreadSymIm[2][state1];
}

if (TurboMode) {
    if (Puncture) {
        if (t%2) { // parity punctured
            TranMet1Re = sym1 * sym1re;
            TranMet1Im = sym1 * sym1im;
            Sova2energy1Re = s_hat[2*t+1]*sym2re;
            Sova2energy1Im = s_hat[2*t+1]*sym2im;
        } else { // parity is available
            TranMet1Re = sym1 * sym1re + sym2 * sym2re;
            TranMet1Im = sym1 * sym1im + sym2 * sym2im;
            Sova2energy1Re = 0;
            Sova2energy1Im = 0;
        }
    } else { // unpunctured, third sym is parity for SOVA #2
        TranMet1Re = sym1 * sym1re + sym2 * sym2re;
        TranMet1Im = sym1 * sym1im + sym2 * sym2im;
        Sova2energy1Re = s_hat[2*t+1] * sym3re;
        Sova2energy1Im = s_hat[2*t+1] * sym3im;
    }
} else { // not a Turbo code, simple conv code only
    TranMet1Re = sym1 * sym1re + sym2 * sym2re;
    TranMet1Im = sym1 * sym1im + sym2 * sym2im;
    Sova2energy1Re = 0;
    Sova2energy1Im = 0;
}

// update the path metric for the 2 possible paths into this state
m0 = TranMet0Re + PathMetricCurr[state0] - La[t]/2;
m1 = TranMet1Re + PathMetricCurr[state1] + La[t]/2;

// if(t==12)printf("TranMet0Re=%f,TranMet1Re=%f\n",TranMet0Re,TranMet1Re);
// if(t==12)printf("m0=%f, m1=%f\n",m0,m1);

// select best path based on current path metric m0 or m1
if( m0 > m1)
{
    // decide bit == 0
}

```

```

PathMetricNext[state] = m0;
Mdiff[state][t+1] = m0-m1;
SymsML = Syms0;           // symbol pair for ML path
D_ML = DelayEst[state0];  // propagate delay for ML path
if (t==1 )
    printf(" *** t=%d, D_ML+1=%d\n",t,D_ML+1);
stateML=state0;
MX_re[state] = MX_prev_re[state0] + TranMet0Re + Sova2energy0Re;
MX_im[state] = MX_prev_im[state0] + TranMet0Im + Sova2energy0Im;
t2[state] = t2Prev[state0];
t3[state] = t3Prev[state0];
phi_m[state] = phi_mPrev[state0];
Delta_phi[state] = Delta_phiPrev[state0];
} else {
PathMetricNext[state] = m1;
Mdiff[state][t+1] = m1-m0;
SymsML = Syms1;
prev_bit[t+1] |= bitmap[state]; // set bit at bit position "state" to 1
D_ML = DelayEst[state1];  // propagate delay for ML path
if (t==1 )
    printf(" *** t=%d, D_ML+1=%d\n",t,D_ML+1);
stateML=state1;
MX_re[state] = MX_prev_re[state1] + TranMet1Re + Sova2energy1Re;
MX_im[state] = MX_prev_im[state1] + TranMet1Im + Sova2energy1Im;
t2[state] = t2Prev[state1];
t3[state] = t3Prev[state1];
phi_m[state] = phi_mPrev[state1];
Delta_phi[state] = Delta_phiPrev[state1];
}

```

// If we are doing a punctured Turbo code we substitute the hard parity
// symbol estimate from SOVA #2 so E/L gate correlators can integrate
// all of RxSig (systematic bits plus parity bits for both decoders). If
// unpunctured Turbo then integrate across all three symbols. If SOVA
// only there are just the 2 symbols from this SOVA.

```
SymML1=SymVal[SymsML][0];    // systematic symbol
```

```

if (TurboMode)
if (Puncture) {
    if (t%2)           // odd time step
        SymML2=s_hat[2*t+1];    // parity symbol from SOVA #2
    else
        SymML2=SymVal[SymsML][1]; // SOVA 1 parity symbol
} else {
    SymML2=SymVal[SymsML][1];    // SOVA 1 parity symbol
    SymML3=s_hat[2*t+1];        // parity symbol from SOVA #2
}
else
    SymML2=SymVal[SymsML][1];    // SOVA 1 parity symbol

```

```
// printf("C: %d %d %d\n",SymML1,SymML2,SymML3);
```

// Sum RxSig over block length L for use in DLL, first sum Early Gate

```

re2 = cos( ThetaEst); im2 = -sin( ThetaEst);

for ( i=term1re=term1im=0 ; i< Nsamp; i++) {
  re1 = RxSigRe[index1+i] * RxRef[index1*Os+D_ML+DeltaDll+i*Os];
  im1 = RxSigIm[index1+i] * RxRef[index1*Os+D_ML+DeltaDll+i*Os];
  term1re += re1*re2-im1*im2;
  term1im += re1*im2+re2*im1;
}

for ( i=term2re=term2im=0 ; i< Nsamp; i++) {
  re1 = RxSigRe[index2+i] * RxRef[index2*Os+D_ML+DeltaDll+i*Os];
  im1 = RxSigIm[index2+i] * RxRef[index2*Os+D_ML+DeltaDll+i*Os];
  term2re += re1*re2-im1*im2;
  term2im += re1*im2+re2*im1;
}

if (TurboMode && !Puncture) {

  for ( i=term3re=term3im=0 ; i< Nsamp; i++) {
    re1 = RxSigRe[index3+i] * RxRef[index3*Os+D_ML+DeltaDll+i*Os];
    im1 = RxSigIm[index3+i] * RxRef[index3*Os+D_ML+DeltaDll+i*Os];
    term3re += re1*re2-im1*im2;
    term3im += re1*im2+re2*im1;
  }

  SymSumRe[0][state] = SymML1*term1re + SymML2*term2re +
    SymML3*term3re + SymSumPrevRe[0][stateML];

} else {

  SymSumRe[0][state] = SymML1*term1re + SymML2*term2re +
    SymSumPrevRe[0][stateML];

}

// Sum for Late Gate

for ( i=term1re=term1im=0 ; i< Nsamp; i++) {
  re1 = RxSigRe[index1+i] * RxRef[index1*Os+D_ML-DeltaDll+i*Os];
  im1 = RxSigIm[index1+i] * RxRef[index1*Os+D_ML-DeltaDll+i*Os];
  term1re += re1*re2-im1*im2;
  term1im += re1*im2+re2*im1;
}

for ( i=term2re=term2im=0 ; i< Nsamp; i++) {
  re1 = RxSigRe[index2+i] * RxRef[index2*Os+D_ML-DeltaDll+i*Os];
  im1 = RxSigIm[index2+i] * RxRef[index2*Os+D_ML-DeltaDll+i*Os];
  term2re += re1*re2-im1*im2;
  term2im += re1*im2+re2*im1;
}

if (TurboMode && !Puncture) {

```

```

for ( i=term3re=term3im=0 ; i< Nsamp; i++) {
    re1 = RxSigRe[index3+i] * RxRef[index3*Os+D_ML-DeltaDll+i*Os];
    im1 = RxSigIm[index3+i] * RxRef[index3*Os+D_ML-DeltaDll+i*Os];
    term3re += re1*re2-im1*im2;
    term3im += re1*im2+re2*im1;
}

SymSumRe[2][state] = SymML1*term1re + SymML2*term2re +
    SymML3*term3re + SymSumPrevRe[2][stateML];

} else {

SymSumRe[2][state] = SymML1*term1re + SymML2*term2re +
    SymSumPrevRe[2][stateML];

}

// update DLL and PLL if at block boundary
if( (t+1) % BlockLen == 0)
{

if( SymSumRe[0][state] > SymSumRe[2][state] ) // assume PLL pulled in
    DelayEstNext[state] = D_ML+dll_step;    // retard delay
else
    DelayEstNext[state] = D_ML-dll_step;    // advance delay

// reset to 0 and integrate over next block
SymSumRe[0][state] = 0;
// SymSumRe[1][state] = 0;
SymSumRe[2][state] = 0;

delta_phi[state] = atan2( MX_im[state], MX_re[state] )/TwoPi;
t2[state] = t2[state] + delta_phi[state];
t3[state] = t3[state]+t2[state];
Delta_phi[state] = K1*delta_phi[state] + K2*t2[state] + K3*t3[state];
phi_m[state] = phi_m[state] + Delta_phi[state];

} else {
// Propagate Delay Estimate for use in next iteration
DelayEstNext[state] = D_ML;
}

} /* end state loop */

// update PathMetric, Delay and symbol sum for next time iteration
for( state = 0; state< Nstates; state++)
{
    PathMetricCurr[state]=PathMetricNext[state];
    DelayEst[state] = DelayEstNext[state];
    SymSumPrevRe[0][state] = SymSumRe[0][state];    // early
    SymSumPrevRe[1][state] = SymSumRe[1][state];    // middle
    SymSumPrevRe[2][state] = SymSumRe[2][state];    // late
    MX_prev_re[state] = MX_re[state];
    MX_prev_im[state] = MX_im[state];
}

```

```

t2Prev[state] = t2[state];
t3Prev[state] = t3[state];
Delta_phiPrev[state] = Delta_phi[state];
phi_mPrev[state] = phi_m[state];
}

if( (t+1) % BlockLen == 0) {
for( state=0; state<Nstates; state++) {
    MX_prev_re[state] = 0; MX_prev_im[state] = 0;
}
ramp =0;
}

for( state=0; state<Nstates; state++) {
    phi_nco[state] = phi_m[state] - Delta_phi[state]/2 +
                    ramp*Delta_phi[state]/BlockLen;
}

ramp = ramp+1; // ramp linear phase correction across the block

} // end trace forward, t

// if Term = 1 trace back from zero state
// if Term = 2 trace back from state with highest metric */
if (Term == 1)
    beststate[NUM_BITS] = 0;
else
{
    // find best metric
    bestmetric = PathMetricCurr[0];
    beststate[NUM_BITS] = 0;
    for (i=0; i<NUM_STATES; i++)
    {
        if ( PathMetricCurr[i] > bestmetric)
        {
            bestmetric = PathMetricCurr[i];
            beststate[NUM_BITS] = i;
        }
    }
}

for (t=NUM_BITS; t>0; t--)
{
    est_bits[t] = ( prev_bit[t] & bitmap[ beststate[t] ] ) >> beststate[t];
    if(est_bits[t])
        beststate[t-1] = prevstate1[beststate[t]]; // est bit 1
    else
        beststate[t-1] = prevstate0[beststate[t]]; // est bit 0
    if (TurboMode && !Puncture) {
        DespreadSyms[3*t-3]=SysSymMat[beststate[t-1]][t-1];
        DespreadSyms[3*t-2]=ParSym1Mat[beststate[t-1]][t-1];
        DespreadSyms[3*t-1]=ParSym2Mat[beststate[t-1]][t-1];
    } else {

```

```

    DespreadSyms[2*t-2]=SysSymMat[beststate[t-1]][t-1];
    DespreadSyms[2*t-1]=ParSym1Mat[beststate[t-1]][t-1];
}
}

// Find the minimum path metric diff that corresponds to an error path with
// different information bit estimation. For each bit in est(t) check all
// possible paths up to Delta stages. At each stage the path deviation is
// found by choosing the incorrect bit i.e., the opposite bit from the
// chosen path. The path and bit is indicated in the arrays beststate[]
// and est_bits[].

for (t=1; t<NUM_BITS+1; t++) // for each bit find Le
{
    llr = BIG_POS; // set log-likelihood ratio to large number
    for (i =0; i<=SOVA_DELTA; i++)
    {
        if( t+i < NUM_BITS+1) // do not go past end
        {
            error_bit = 1-est_bits[t+i]; // force an error at beginning of path
            beststate_tb = beststate[t+i];
            // trace back from bit error
            for (j=i; j>0; j--)
            {
                if(error_bit)
                    beststate_tb = prevstate1[beststate_tb]; // est bit 1
                else
                {
                    beststate_tb = prevstate0[beststate_tb]; // est bit 0
                }
                error_bit = (prev_bit[t+j-1]& bitmap[beststate_tb]) >> beststate_tb;
            }
            // after tracing back check if incorrect decision at stage t+i
            // resulted an bit error at stage t.
            if (error_bit != est_bits[t] )
                llr = min( llr, Mdiff[ beststate[t+i]][t+i]);
        }
    }
    // calculate Lall for bit at stage t
    // recall that llr is stored at t-1 while decoded bit is at t
    Lall[t-1] = (2*(int)est_bits[t] -1)*llr;
}

return(0);
}

```

```

// turbo_dp11.c
//
// call sova_dp11 & sova to make turbo decoder with an integrated DLL/PLL
//
// Richard C. Ormesher, Jeff Mason, SNL dept. 2344, 9/5/02

#include <stdio.h>

#include "scenario.h" // simulation constants
#include "turbo_dp11.h" // make sure proto is up to date
#include "sova_dp11.h" // for sova_dp11_K3 proto
#include "sova.h" // for sova_K3 proto

#ifdef MATLAB_MEX_FILE
#include "mex.h" // for mexPrintf()
#endif

int turbo_dp11( NUMTYPE *Lall, NUMTYPE *RxSigRe, NUMTYPE *RxSigIm,
               NUMTYPE *RxRef, int *alpha, int niter, int BlockLen,
               int InitDelay, int *TxBits, int dll_step,
               int pll_flag, NUMTYPE *pll_params)

// Lall -- returned log-likelihood ratio for estimated bit, sign of Lall
// indicates bit value
// RxSig -- samples of spread spectrum signal
// alpha -- interleave pattern for decoder 2.
// niter -- number of iterations for sova decoder.
// RxRef -- reference function used to despread symbols

{

NUMTYPE DespreadSyms[NUM_SYMS];
NUMTYPE La[NUM_BITS];
NUMTYPE Le[NUM_BITS];
NUMTYPE SoftSyms1[NUM_BITS*2];
NUMTYPE SoftSyms2[NUM_BITS*2];
NUMTYPE TempV[NUM_BITS];

int s_hat[NUM_BITS*2];
int TurboMode; // tell sova_dp11 it is part of a turbo code
int i, j, k;
int status, Nerr1, Nerr2;
int Term; // flag indicating whether trellis is terminated or not
int Puncture=PUNCTURE;

// printf("InitDelay=%d \n", InitDelay);

// init extrinsic info and SOVA #2 hard sym estimates for 1st sova_dp11 call
for (i=0; i<NUM_BITS; i++)
Le[i]=0;

for (i=0; i<NUM_BITS*2; i++)
s_hat[i]=0;

```

```

// loop through niter iterations before decoding data
for (i=0; i<niter; i++)
{
// Decoder 1
// Deinterleave extrinsic info for decoder 1
for (j=0; j<NUM_BITS; j++)
    La[ alpha[j] ] = Le[j]/2;    // Le growth control

// if (i==1) for (j=0; j<2*NUM_BITS; j++) printf("%d\n",s_hat[j]);

// call sova_dpll
status = sova_dpll( RxSigRe, RxSigIm, La, Lall, Term=1, RxRef, InitDelay,
    BlockLen, DespreadSyms, s_hat, TurboMode=1, dll_step,
    pll_flag, pll_params);

// count the bit errors after sova_dpll
Nerr1=0;
for(j=0; j<NUM_BITS; j++) {
    // printf("%1.8f\n", Lall[j]);
    k = (0<Lall[j]) ? 1 : 0;
    if (k!=TxBits[j])
        Nerr1++;
}

// demultiplex the despread symbols that sova_dpll just produced
if (Puncture) {
    for (j=0; j<NUM_BITS; j++) {
        SoftSyms1[2*j]=DespreadSyms[2*j];    // systematic bits
        SoftSyms2[2*j]=DespreadSyms[2*alpha[j]];    // systematic bits
        if (j%2) {
            SoftSyms1[2*j+1]=0;    // punctured bits
            SoftSyms2[2*j+1]=DespreadSyms[2*j+1];    // parity bits
        } else {
            SoftSyms1[2*j+1]=DespreadSyms[2*j+1];    // parity bits
            SoftSyms2[2*j+1]=0;    // punctured bits
        }
    }
} else {
    for (j=0; j<NUM_BITS; j++) {
        SoftSyms1[2*j]=DespreadSyms[3*j];    // systematic bits
        SoftSyms2[2*j]=DespreadSyms[3*alpha[j]];    // systematic bits
        SoftSyms1[2*j+1]=DespreadSyms[3*j+1];    // parity bits
        SoftSyms2[2*j+1]=DespreadSyms[3*j+2];    // parity bits
    }
}

for(k=0; k< NUM_BITS; k++) {
    Le[k] = Lall[k] - 2*SoftSyms1[2*k] - La[k];
    // printf("%1.8f %1.8f %1.8f\n", Le[k], Lall[k], La[k]);
}

// Decoder 2
// interleave extrinsic info for decoder 2
for (j=0; j<NUM_BITS; j++) {

```

```

    La[j] = Le[alpha[j]];
}

// call sova
status = sova( SoftSyms2, La, Lall, s_hat, Term=2);

for(k=0; k< NUM_BITS; k++)
    Le[k] = Lall[k] - 2*SoftSyms2[2*k] - La[k];

// de-interleave the soft syms in order to count the bit errors
for (j=0; j<NUM_BITS; j++) {
    TempV[ alpha[j] ] = Lall[j];
    // printf("Lall[%d]=%15.5f, s_hat[%d]=%3d\n", j, Lall[j], s_hat[j]);
}

// count the bit errors this iter
Nerr2=0;
for(j=0; j<NUM_BITS; j++) {
    // printf(" %2d, TempV[j]=%fn", j, TempV[j]);
    k = (0<TempV[j]) ? 1 : 0;
    if (k!=TxBits[j])
        Nerr2++;
}

printf(" iter %2d, Lall[0]=%1.6f, Nerr1=%d, Nerr2=%d\n",
        i+1, TempV[0], Nerr1, Nerr2);

if (Nerr2==0) break; // cheat to save simulation time
}

// Estimate data bits from sign of Lall, use Le as temp array to deinterleave
for (j=0; j<NUM_BITS; j++)
    Le[ alpha[j] ] = Lall[j];

for (j=0; j<NUM_BITS; j++)
    Lall[ j ] = Le[j];

return(0);
}

```

```

// encode_rsc.c
//
// void encode_rsc( int *data,
//                 int *codeword,
//                 int Terminate)
//
// ABSRACT -- used to recursive systematic convolve (RSC) info bits
//           contained in data, up to a constraint length of 7
//
// INPUT
// data -- infomation bits, each int in the array contains a 1 or 0.
//       Length of data must include room for K-1 tail bits if
//       Terminate is True.
// Terminate -- boolean, if set indicates to Terminate encoder state to 0.
//           Tail bits are added to the data such that the encoder
//           ends with all zeros in shift register.
//
// RCO, JJM 10/16/02

#include "encode_rsc.h" // this guarantees that the proto is up to date
#include "scenario.h"   // for NUM_BITS and g1 & g2
#include <stdio.h>

// Partab is a LUT for the binary output of a 7 input mod-2 adder, the 7
// inputs are packed to form the index into table. Note max index is 2^7=128
// to accomodate K=7, K=3 only could be just the first row.

static unsigned char Partab[] = {
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1
};

void encode_rsc( int *data,
                int *codeword,
                int Terminate)

{
    int e,i;
    int Ntail=CONSTRAINT_LENGTH-1;
    int Nbits=NUM_BITS;

```

```

int g1=RSC_G_1, g2=RSC_G_2; // defined in scenario.h
int Fb, state2, state=0;

for(i=0;i<Nbits;i++)
{
    if (i > Nbits-Ntail-1 && Terminate)
    {
        // Terminate to zero state by changing tail bits so that Fb will be zero.
        data[i] = Partab[state & g1];
    }

    if( data[i] == 0)
    {
        // info bit = 0, bit 0 of state remains zero
        Fb = Partab[state & g1]; // feedback symbol
        codeword[2*i] = 0; // info bit
    }
    else
    {
        // info bit = 1, bit 0 of state is set to 1
        Fb = Partab[(state+1) & g1]; // Fb is the feedback symbol for g1 input
        codeword[2*i] = 1; // info bit
    }

    state2 = state | Fb; // input Fb into the shift register
    e = Partab[state2 & g2]; // get parity generated by g2
    codeword[2*i+1] = e; // parity bit
    state = 2*state2; // update state, g1 & g2 will mask out old bits
}
}

```

```

// gen_tables.c
//
// return prevstate0, prevstate1, prevsym0 & prevsym1
//
// use code from Sergio Benedetto, IEEE Tran on Comm, Vol 46 No 9, Sept 1998
//
// RCO, JJM

#include "scenario.h" // for CONSTRAINT_LENGTH, NUM_STATES, G1 and G2
#include "gen_tables.h" // make sure proto is up to date

// Partab[n] is the mod 2 sum of the binary digits of n

static unsigned char Partab[] = {
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0,
};

int gen_tables( int *prevstate0, int *prevstate1, int *prevsym0, int *prevsym1)
{
    int e, i, Mask=0;
    int g1=RSC_G_1, g2=RSC_G_2, Fb, state2;

```

```

int nextstate0[NUM_STATES], nextstate1[NUM_STATES];
int parity0[NUM_STATES], parity1[NUM_STATES];

for (i=0;i<CONSTRAINT_LENGTH-1;i++)
    Mask=(Mask<<1)|0x1;

for(i=0;i<NUM_STATES;i++)
{
    /* i is the state of the shift reg, 2*i shifts a zero in lsb */

    /* info bit = 0 */
    Fb = Partab[2*i & g1];    /* Fb is the feedback symbol for g2 input */
    state2 = 2*i | Fb;    /* input for second operation with g2 */
    e = Partab[state2 & g2];    /* get parity for feedback with g2 */
    parity0[i] = e;    /* parity bit */
    nextstate0[i] = ((i<<1)|Fb)&Mask;    /* shift Fb into state register*/

    /* info bit = 1 */
    Fb = Partab[(2*i+1) & g1];    /* Fb is the feedback symbol for g2 input */
    state2 = 2*i | Fb;    /* input for second operation with g2 */
    e = Partab[state2 & g2];    /* get parity for feedback with g2 */
    parity1[i] = e;    /* parity bit */
    nextstate1[i] = ((i<<1)|Fb)&Mask;    /* shift Fb into state register */

}

// get previous state given current state and sym
for (i=0; i<NUM_STATES; i++)
{
    prevstate0[ nextstate0[i] ] = i;
    prevstate1[ nextstate1[i] ] = i;
}

for (i=0; i<NUM_STATES; i++)
    prevsym0[i]=parity0[prevstate0[i]];

for (i=0; i<NUM_STATES; i++)
    prevsym1[i]=2+parity1[prevstate1[i]];

return(0);
}

```

```

// encode_turbo.c
//
// int encode_turbo(int *TxSym, int *Data, int *Alpha)
//
// ABSRACT
//
// Generate 1/2 or 1/3 Rate turbo code using Alpha as interleaver.
// Results are in TxSym. Turbo encoder uses two RSC encoders.
//
// INPUT
//
// Data -- data bits to be encoded - length must include room for tail,
//       i.e. length is NUM_BITS, which is defined in scenario.h
//
// Alpha -- interleave pattern, i.e. random indices, length NUM_BITS
//
// OUTPUT
//
// TxSym -- encoded symbols
//
// For Punctured code
//   TxSym = [d1, s11, d2 s22, d3, s13, d4 s24 ..
//
// For non-punctured code
//   TxSym = [d1, s11, s21, d2, s12, s22, d3, s13, s23, d4 ...
//
// where d1, d2, ... are info bits; s11, s12, .. are parity symbols
// from first encoder; and s21, s22 are parity from second encoder.
//
// RCO, JJM 4/30/02

#include <stdio.h>

#include "encode_turbo.h" // this guarantees that the protos are up to date
#include "encode_rsc.h"
#include "scenario.h" // for NUM_BITS and g1 & g2

// declare working arrays for encoding
static int codeword1[NUM_BITS*2];
static int codeword2[NUM_BITS*2];
static int data_int[NUM_BITS];

int encode_turbo(int *TxSym, int *Data, int *Alpha)
{
    int i, Nbits=NUM_BITS;
    int Puncture=PUNCTURE;
    int Terminate;

    // get codeword from first encoder and add tail to Data
    encode_rsc( Data, codeword1, Terminate=1);

    // interleave data bits
    for (i=0; i<Nbits; i++)

```

```

data_int[i] = Data[Alpha[i]];

// get codeword for second encoder
encode_rsc( data_int, codeword2, Terminate=0);

// create complete codeword
if( Puncture)
{
// Create punctured rate = 1/2 code
for (i = 0; i<Nbits; i=i+2)
{
TxSym[2*i] = Data[i];
TxSym[2*i +1] = codeword1[2*i+1];
}
for (i = 1; i<Nbits; i=i+2)
{
TxSym[2*i] = Data[i];
TxSym[2*i +1] = codeword2[2*i+1];
}
}
else
{
// create non-punctured rate 1/3 code
for (i=0; i<Nbits; i++)
{
TxSym[3*i] = Data[i];
TxSym[3*i +1] = codeword1[2*i+1];
TxSym[3*i +2] = codeword2[2*i+1];
}
}
return(0);
}

```

```

// run_turbo_dll.c
//
// call gen_turbo_sig() and turbo_dll()
//
// JJM, RCO 5/7/02

#include <stdio.h>
#include <stdlib.h>      // for exit
#include <math.h>       // for M_PI, sin, cos, sqrt

#include "scenario.h"
#include "gen_turbo_sig.h"
#include "turbo_dll.h"
#include "turbo_dppll.h"
#include "indexx.h"
#include "rng.h"

#include "turbo_dppll.h"
NUMTYPE RxSigRe[NUM_SAMP_IN_SIG];
NUMTYPE RxSigIm[NUM_SAMP_IN_SIG];

NUMTYPE RxChips[NUM_SAMP_IN_SIG];
NUMTYPE RxRef[NUM_SAMP_IN_REF];

int main()
{

double EbN0dB = 1.0;
double TxClockError = 5e-6;
double A=1;
double CodeRate = CODE_RATE;
double EbN0, EsN0, SNR, Sigma, Noise;

int BlockLen=20; // number of symbols integrated for each PLL and DLL step
int dll_step=1; // have dll step this many reference samples at a time
int Niter=10; // max number of turbo algorithm iterations

NUMTYPE La[NUM_BITS];
NUMTYPE Lall[NUM_BITS];

// NUMTYPE pll_params[3]={0.5463, 0.1768, 0.02470}; // under-damped 0.4 BLT
NUMTYPE pll_params[3]={0.3599, 0.06842, 0.005269}; // under-damped 0.2 BLT

int TxBits[NUM_BITS];
int RxBits[NUM_BITS];
int Alpha[NUM_BITS];
int TxSyms[NUM_SYMS];

int NumBits = NUM_BITS; // info plus tail bits
int NumSyms = NUM_SYMS;
int Nspc = NUM_SAMP_PER_CHIP;
int OS = REF_OVER_SAMP;
int K = CONSTRAINT_LENGTH;
int Lsig = NUM_SAMP_IN_SIG;

```

```

int Lref = NUM_SAMP_IN_REF;
int Ns=NUM_SAMP_PER_SYM;
int Nc = NUM_CHIP_PER_SYM;
int Puncture = PUNCTURE;

int i, n, Nerror;
int NumBlocks;
int delta_clock;
int Nf=OS*8;
int InitDelayEst=Nf+5-1; // hand tweak group delay

double Pi = M_PI;
double RxPhase;
double t=0, dt, delta_tot_phase;

int pll_flag=1; // 0 turns PLL off, Gam should be 0 for this case
double Gam=5; // linear phase rate

printf("size of NUMTYPE = %d bytes\n", sizeof(NUMTYPE));
printf("Ninfo = %d \n", NUM_INFO);
printf("K = %d\n", K);
printf("Puncture = %d, CodeRate = %g \n", Puncture, CodeRate);
printf("NumBits = %d \n", NumBits);
printf("NumSyms = %d \n", NumSyms);
printf("Nc = %d \n", Nc);
printf("Nspc = %d \n", Nspc);
printf("OS = %d \n", OS);
printf("g1 = %#0x, g2 = %#0x\n", RSC_G_1, RSC_G_2);
printf("dll_step = %d \n", dll_step);
printf("pll_flag = %d\n",pll_flag);
printf("InitDelayEst = %d \n", InitDelayEst);
printf("Niter=%d\n",Niter);
printf("K1=%f, K2=%f, K3=%f\n",pll_params[0],pll_params[1],pll_params[2]);

// Print the number of reference samples of clock drift over the signal, this
// is the no. of clicks the DLL must move, and must be less than NumBlocks.
NumBlocks=ceil((double)NumBits/BlockLen);
delta_clock=ceil(TxClockError*Nc*NumSyms*Nspc*OS);
printf("BlockLen = %d \n", BlockLen);
printf("delta_clock = %d\n",delta_clock); // units are reference samples
printf("NumBlocks=%d\n",NumBlocks);

// Print clock drift as the fractional rate, eg chips/chip, and in terms of
// the number of chips of adjustment that will be required of the DLL.
printf("TxClockError=%g\n",TxClockError);
printf("delta_clock = %1.1f chips\n",(double)delta_clock/(Nspc*OS));

if (delta_clock>NumBlocks*dll_step) {
    printf("decrease BlockLen, there are not enough blocks");
    return(2);
}

// RNGs are set both here for sig_gen and further below for some particular
// iteration from run_turbo_dll.m

```

```

set_useed(0); set_nseed(0);
printf("sig gen seeds: %u, %u\n", get_useed(), get_nseed());

gen_turbo_sig( TxClockError, Nf, Alpha, TxBits, TxSyms, RxChips, RxRef);

EbN0=pow(10,EbN0dB/10);
EsN0=CodeRate*EbN0;
SNR = EsN0/Ns;      // Input SNR at A/D bandwidth; Es/N0 - Gain
Sigma = A/sqrt(2*SNR);
printf("EbN0=%g dB\n",EbN0dB);
printf("Lsig = %d, Lref = %d\n", Lsig, Lref);

printf("Gam = %f\n",Gam);
delta_tot_phase=Gam/2; // total cycle of phase, assume sig is 1 sec long
dt=1.0/Lsig;          // time step, use this for now
printf( "delta_phase = %f cycles\n", delta_tot_phase);
printf( "delta_phase = %f cycles/block\n", delta_tot_phase/NumBlocks);

set_useed(0); set_nseed(0);
printf("iteration seeds: %u, %u\n", get_useed(), get_nseed());

for (i=0; i<NumBits; i++)
    La[i] = 0;

for (n=0; n<Lsig; n++) {
    RxPhase=Pi*Gam*t*t;
    t += dt;
    Noise=Sigma*gasdev();
    RxSigRe[n] = A*RxChips[n]*cos(RxPhase)+Noise;
    Noise=Sigma*gasdev();
    RxSigIm[n] = A*RxChips[n]*sin(RxPhase)+Noise;
}

turbo_dp11( Lall, RxSigRe, RxSigIm, RxRef, Alpha, Niter, BlockLen,
            InitDelayEst, TxBits, dll_step, pll_flag, pll_params);

// for(n=0; n<NumBits; n++) {
//   printf("%20.14e \n", Lall[n]);
// }

Nerror=0;
printf("Error Indices: ");
for(n=0; n<NUM_BITS; n++) {
    RxBits[n] = (0<Lall[n]) ? 1 : 0;
    if (RxBits[n]!=TxBits[n]) {
        Nerror++;          // which bits were in error ?
        printf("%d ",n+1); // number bits from 1 to compare to Matlab
    }
}
printf("\n");

printf("Nerror=%d\n",Nerror);

return(0);

```

}

```

// sova.c
//
// soft output viterbi algorithm decoder
//
// adapted from Yufei Wu's sova0.m and Phil Karn's C-code
//
// Richard C. Ormesher, Jeff Mason, SNL dept. 2344, 9/5/02

#include <stdio.h>
#include "scenario.h"
#include "gen_tables.h"
#include "sova.h" // include proto to guarantee its consistency

#ifdef MATLAB_MEX_FILE
#include "mex.h"
#endif

static int prevsym0[NUM_STATES];
static int prevsym1[NUM_STATES];
static int prevstate0[NUM_STATES];
static int prevstate1[NUM_STATES];

// i -- current state
// sym0 -- symbol weight for info bit = 0
// sym1 -- symbol ewight for info bit = 1
#define BUTTERFLY(i,sym0, sym1) { \
    s0 = prevstate0[i]; \
    s1 = prevstate1[i]; \
    m0 = pathmetric_curr[s0] + mets[sym0] - La[t]/2; \
    m1 = pathmetric_curr[s1] + mets[sym1] + La[t]/2; \
    if( m0 > m1) { \
        pathmetric_next[i] = m0; \
        Mdiff[i][t+1] = m0-m1; /* bit at state i defaults to 0 */ \
    } \
    else { \
        pathmetric_next[i] = m1; \
        Mdiff[i][t+1] = m1-m0; \
    } \
    if (i < NUM_STATES_2) \
        prev_bit_lo[t+1] |= bitmap[i]; /* set bit at state i to 1 */ \
    else \
        prev_bit_hi[t+1] |= bitmap[i-NUM_STATES_2]; \
    } \
}

static NUMTYPE mets[NUM_STATES], m0, m1, llr, bestmetric;
static NUMTYPE pathmetric_curr[NUM_STATES];
static NUMTYPE pathmetric_next[NUM_STATES];
static NUMTYPE Mdiff[NUM_STATES][NUM_BITS+1];
static unsigned prev_bit_lo[NUM_BITS+1];
static unsigned prev_bit_hi[NUM_BITS+1];
static unsigned char est_bits[NUM_BITS+1], error_bit;
static unsigned bitmap[NUM_STATES]; static int tables_ready = 0;
static int beststate[NUM_BITS+1], beststate_tb;
static int SYMVAL[4][2] = { {-1, -1}, {-1, 1}, {1, -1}, {1, 1}};

```

```

int sova( NUMTYPE *symbols, NUMTYPE *La, NUMTYPE *Lall,
          int *s_hat, int termination)

{

int i, j, t, state, s0, s1 ;

if (!tables_ready) { // init tables on first call
    tables_ready=1;
    gen_tables( prevstate0, prevstate1, prevsym0, prevsym1);
    for (i=0, bitmap[0]=1; i<NUM_STATES-1; i++)
        bitmap[i+1] = bitmap[i] * 2;
}

// Initialize path metric
for(state = 0; state<NUM_STATES; state++) {
    pathmetric_curr[state] = BIG_NEG; // set to large neg number
}

pathmetric_curr[0] = 0; // start with state 0

// init prev_bit
for(t=0; t<NUM_BITS; t++) {
    prev_bit_lo[t] = 0;
    prev_bit_hi[t] = 0;
}

// Trace forward for t = 0 to NUM_BITS
for (t=0; t<NUM_BITS; t++) {
    // calculate transition metric to t+1
    // this means that decoded bits start at t=1
    mets[0] = -symbols[2*t] - symbols[2*t+1];
    mets[1] = -symbols[2*t] + symbols[2*t+1];
    mets[2] = +symbols[2*t] - symbols[2*t+1];
    mets[3] = +symbols[2*t] + symbols[2*t+1];

    // for stage t+1 set all bits at each state to zero
    prev_bit_lo[t+1] = 0;
    prev_bit_hi[t+1] = 0;

    // calculate metric at next state
    for ( state=0; state<NUM_STATES; state++)
        BUTTERFLY( state, prevsym0[state], prevsym1[state]);

    for ( state=0; state<NUM_STATES; state++)
        pathmetric_curr[state]=pathmetric_next[state];
} // end trace forward

// if termination = 1 trace back from zero state
// if termination = 2 trace back from state with highest metric
if (termination == 1)
    beststate[NUM_BITS] = 0;

```

```

else {
// find best metric
bestmetric = pathmetric_curr[0];
beststate[NUM_BITS] = 0;
for (i=0; i<NUM_STATES; i++) {
  if ( pathmetric_curr[i] > bestmetric) {
    bestmetric = pathmetric_curr[i];
    beststate[NUM_BITS] = i;
  }
}
}

// trace back for end and get estimated bit value at each interval
for (t=NUM_BITS; t>0; t--) {

  if (beststate[t] < NUM_STATES_2)
    est_bits[t] = (unsigned char)
      ((prev_bit_lo[t] & bitmap[ beststate[t] ] ) >> beststate[t]);
  else
    est_bits[t] = (unsigned char)
      ((prev_bit_hi[t] & bitmap[beststate[t]- NUM_STATES_2]) >>
        (beststate[t]-NUM_STATES_2));

  if (est_bits[t]) {
    beststate[t-1] = prevstate1[beststate[t]]; // est bit 1
    s_hat[2*t-2] = 1; // systematic bit
    s_hat[2*t-1] = SYMVAL[prevsym1[beststate[t]][1]; // parity bit
  } else {
    beststate[t-1] = prevstate0[beststate[t]]; // est bit 0
    s_hat[2*t-2] = -1; // systematic bit
    s_hat[2*t-1] = SYMVAL[prevsym0[beststate[t]][1]; // parity bit
  }
}

// Find the minimum SOVA_DELTA that corresponds to an error path with
// different information bit estimation. For each bit in est(t) check all
// possible paths up to Delta stages At each stage the path deviation is
// found by chossing the incorrect bit i.e., the opposite bit from the
// chossen path. The path and bit is indicated in the arrays beststate[]
// and est_bits[].

for (t=1; t<NUM_BITS+1; t++) { // for each bit find Le

  llr = BIG_POS; // set log-likelihood ratio to large number
  for (i =0; i<=SOVA_DELTA; i++) {
    if( t+i < NUM_BITS+1) { // do not go past end

      error_bit = 1-est_bits[t+i]; // force an error at begining of path
      beststate_tb = beststate[t+i];

      // trace back from bit error
      for (j=i; j>0; j--) {
        if(error_bit)
          beststate_tb = prevstate1[beststate_tb]; // est bit 1
        else {

```

```

    beststate_tb = prevstate0[beststate_tb]; // est bit 0
}
if( beststate_tb < NUM_STATES_2)
    error_bit = (unsigned char)
        ((prev_bit_lo[t+j-1]& bitmap[beststate_tb]) >> beststate_tb);
else
    error_bit = (unsigned char)
        ((prev_bit_hi[t+j-1]& bitmap[beststate_tb-NUM_STATES_2]) >>
        (beststate_tb-NUM_STATES_2));
}

// after tracing back check if incorrect decion at stage t+i
// resulted an bit error at stage t.
if (error_bit != est_bits[t] )
    llr = min( llr, Mdiff[ beststate[t+i]][t+i]);

} // end if
} // end for i

// calculate Lall for bit at stage t
// recall that llr is stored at t-1 while decoded bit is at t
Lall[t-1] = (2*(int)est_bits[t] -1)*llr;

} // end for t

return(0);

}

```

```

// turbo.c
//
// RCO, JJM 4/30/02

#include <stdio.h>
#include "scenario.h"
#include "turbo.h" // make sure proto is up to date
#include "sova.h"

#ifdef MATLAB_MEX_FILE
#include "mex.h"
#endif

int turbo( NUMTYPE *Lall, NUMTYPE *RxSym, int *alpha, int niter, int * TxBits)
// Lall -- returned log-likelihood ratio for estimated bit, sign of Lall
// indicates bit value, length NUM_BITS
// RxSym -- soft symbols for decoders: ui, p1i, p2i, ui+1 ... length NUM_SYMS
// alpha -- interleave pattern for decoder 2, length NUM_BITS
// niter -- number of iterations for sova decoder.
// TxBits -- the binary data bits that were transmitted, used only to print
// the number of bit errors at each turbo iteration, length NUM_BITS

{

static NUMTYPE rec_s_1[NUM_BITS*2];
static NUMTYPE rec_s_2[NUM_BITS*2];
static NUMTYPE La[NUM_BITS];
static NUMTYPE Le[NUM_BITS];
static NUMTYPE TempV[NUM_BITS];

int puncture=PUNCTURE;
int i, j, k, status=0;
int Nerr1, Nerr2;
int s_hat[NUM_BITS*2];

// get symbols for decoder 1 and decoder 2
if (puncture == 1)
{
// if puncture == 1 then code is 1/2 rate
for (i=0; i<NUM_BITS; i++)
{
rec_s_1[2*i] = RxSym[2*i]; // info bit for encoder 1
rec_s_2[2*i] = RxSym[2*alpha[i]]; // interleave info bits for encoder 2

if (i%2) {
rec_s_1[2*i+1] = 0; // punctured sym gets a 0
rec_s_2[2*i+1] = RxSym[2*i+1]; // copy parity sym
} else {
rec_s_1[2*i+1] = RxSym[2*i+1]; // copy parity sym
rec_s_2[2*i+1] = 0; // punctured sym gets a 0
}
}
}
else

```

```

{
// if puncture ~ = 1 then code is 1/3 rate
for (i=0; i<NUM_BITS; i++)
{
rec_s_1[2*i] = RxSym[3*i]; // info bits
rec_s_2[2*i] = RxSym[3*alpha[i]]; // interleave info bits for encoder 2

rec_s_1[2*i+1] = RxSym[3*i+1]; // symbols from encoder 1
rec_s_2[2*i+1] = RxSym[3*i+2]; // symbols from encoder 2
}
}

// Initialise extrinsic information to zero
for (i = 0; i<NUM_BITS; i++)
Le[i] = 0;

// loop through niters iterations before decoding data
for (i=0; i<niter; i++)
{

// Deinterleave extrinsic info for decoder 1
for (j=0; j<NUM_BITS; j++)
La[ alpha[j] ] = Le[j]/2; // Le growth control

// call sova
status = sova(rec_s_1, La, Lall, s_hat, 1);

// count the bit errors after sova 1
Nerr1=0;
for(j=0; j<NUM_BITS; j++) {
k = (0<Lall[j]) ? 1 : 0;
if (k!=TxBits[j])
Nerr1++;
}

for(k=0; k< NUM_BITS; k++)
Le[k] = Lall[k] - 2*rec_s_1[2*k] - La[k];

// Interleave extrinsic info for decoder 2
for (j=0; j<NUM_BITS; j++)
La[j] = Le[alpha[j]];

// call sova
status = sova(rec_s_2, La, Lall, s_hat, 2);
for(k=0; k< NUM_BITS; k++)
Le[k] = Lall[k] - 2*rec_s_2[2*k] - La[k];

// de-interleave the soft syms in order to count the bit errors
for (j=0; j<NUM_BITS; j++)
TempV[ alpha[j] ] = Lall[j];

// count the bit errors this iter
Nerr2=0;
for(j=0; j<NUM_BITS; j++) {

```

```

k = (0<TempV[j]) ? 1 : 0;
if (k!=TxBits[j]) {
    // printf(" %2d, TempV[j]=%f\n", j, TempV[j]);
    Nerr2++;
}
}

#ifdef FIXED
    // printf(" iter %2d, Lall[0]=%d, Nerr2=%d\n", i+1, TempV[0], Nerr2);
#else
    // printf(" iter %2d, Lall[0]=%e, Nerr1=%d, Nerr2=%d\n",
    //          i+1, TempV[0], Nerr1, Nerr2);
#endif

if (Nerr2==0) break; // cheat to save simulation time

}

// Estimate data bits using sign of Lall
// return deinterleaved Lall, use Le as temp array

for (j=0; j<NUM_BITS; j++)
    Le[ alpha[j] ] = Lall[j];

for (j=0; j<NUM_BITS; j++)
    Lall[ j ] = Le[j];

return(status);

}

```

```

// dll.c
//
// C implementation of dll.m, a plain old DLL/PLL
//
// RCO, JJM 10/23/02

#include <stdio.h>
#include <math.h>          // M_PI, cos, sin, atan2
#include "scenario.h"      // simulation constants
#include "dll.h"          // make sure proto is up to date

void dll( NUMTYPE *SoftSyms, int pll_mode, int init_delay,
          NUMTYPE *RxSigRe, NUMTYPE *RxSigIm, int dll_step, NUMTYPE *RxRef,
          int Nacq, NUMTYPE K1, NUMTYPE K2, NUMTYPE K3, NUMTYPE K4, int OS)
{
    int m, n, k, D=init_delay;
    int NumSyms=NUM_SYMS;
    int Ns=NUM_SAMP_PER_SYM;
    int maxi, SigIndex, RefIndex;

    NUMTYPE TwoPi = 2.0 * M_PI;    // from math.h
    NUMTYPE ys_re, ys_im;
    NUMTYPE maxv, phi_nco;
    NUMTYPE phi_m[NumSyms];
    NUMTYPE phi_start[NumSyms];
    NUMTYPE DELTA_phi[NumSyms];
    NUMTYPE DTPP, delta_phi;
    NUMTYPE re1, im1, re2, im2;    // working variables for doing complex ops
    NUMTYPE EML[3][2];            // E,M,L correlators (summers)
    NUMTYPE v[3];
    NUMTYPE t2=0, t3=0, t4=0;

    phi_m[0]=0;
    phi_start[0]=0;
    DELTA_phi[0]=0;

    // init Early, Middle and Late gate summers
    for (k=0; k<NumSyms; k++) {

        DTPP = DELTA_phi[k]/Ns;    // calculate DELTA_phi per sample point

        for (m=0; m<3; m++)
            for (n=0; n<2; n++)
                EML[m][n]=0;      // init E,M,L correlators (summers)

        for (n=0; n<Ns; n++) {

            phi_nco = phi_start[k] + DTPP*n; // phase units of cycles

            SigIndex = k*Ns+n;      // calculate index into signal

            re1 = RxSigRe[SigIndex];
            im1 = RxSigIm[SigIndex];
            re2 = cos( TwoPi*phi_nco);

```

```

im2 = sin(-TwoPi*phi_nco);

ys_re = re1*re2 - im1*im2;
ys_im = re1*im2 + re2*im1;

RefIndex = SigIndex*OS+D; // calculate index into reference function

EML[0][0] += RxRef[RefIndex-OS]*ys_re; // early real
EML[0][1] += RxRef[RefIndex-OS]*ys_im; // early imag
EML[1][0] += RxRef[RefIndex]*ys_re; // middle real
EML[1][1] += RxRef[RefIndex]*ys_im; // middle imag
EML[2][0] += RxRef[RefIndex+OS]*ys_re; // late real
EML[2][1] += RxRef[RefIndex+OS]*ys_im; // late imag

}

v[0]=EML[0][0]*EML[0][0]+EML[0][1]*EML[0][1]; // early mag sqr
v[1]=EML[1][0]*EML[1][0]+EML[1][1]*EML[1][1]; // middle mag sqr
v[2]=EML[2][0]*EML[2][0]+EML[2][1]*EML[2][1]; // late mag sqr

for (maxv=maxi=n=0; n<3; n++) { // find max value and index
  if (v[n] > maxv) {
    maxv=v[n];
    maxi=n;
  }
}

// calc delta phi over last symbol, use the max val so that PLL pulls in
delta_phi = atan2( EML[maxi][1], EML[maxi][0] ) / TwoPi;

// Use the middle value because its the best guess at where the signal is.
// We assume that the DLL has pulled in by the time data modulation begins.
SoftSyms[k] = EML[1][0]/sqrt(v[1]);

// Compute the BPSK Data and flip phase if Data is a -1 (recall delta_phi is
// in cycles per sym). Perform 0/pi demod after the Nacq preamble symbols,
// this keeps delta_phi in quadrants 1 and 4, ie -0.25 <= delta_phi <= 0.25

if ( k+1 > Nacq ) {

  if ( delta_phi > 0.25 )
    delta_phi -= 0.5;

  if ( delta_phi < -0.25 )
    delta_phi += 0.5;

}

// compute next DLL step, delta_t

if (v[0] > v[2])
  D -= dll_step; // retard delay
else
  D += dll_step; // advance delay

```

```

// Compute terms for pll filter
t2 += delta_phi;
t3 += t2;
t4 += t3;

// estimate phase-rate in cyc/symbol
DELTA_phi[k+1] = K1*delta_phi + K2*t2 + K3*t3 + K4*t4;

// update next midbit phase estimate and calculate next starting NCO phase
if (pll_mode == 1) {
    // model phase is computed but not used in rate only feedback
    phi_m[k+1] = phi_m[k] + (DELTA_phi[k] + DELTA_phi[k+1])/2;
    phi_start[k+1] = phi_start[k] + DELTA_phi[k];
} else if (pll_mode == 2) {
    // update model phase and find start phase given rate and midbit value
    phi_m[k+1] = phi_m[k] + DELTA_phi[k+1];
    phi_start[k+1] = phi_m[k+1] - 0.5*DELTA_phi[k+1];
} else {
    printf("dpll.c: invalid pll_mode/n");
}
}
}

```

```

// run_dpll_then_turbo.c
//
// call gen_turbo_sig(), dpll() and turbo()
//
// JJM, RCO 5/7/02

#include <stdio.h>
#include <stdlib.h>      // for exit
#include <math.h>       // for M_PI, sin, cos, sqrt

#include "scenario.h"
#include "gen_turbo_sig.h"
#include "dpll.h"
#include "rng.h"

#include "turbo.h"

NUMTYPE RxSigRe[NUM_SAMP_IN_SIG];
NUMTYPE RxSigIm[NUM_SAMP_IN_SIG];
NUMTYPE RxChips[NUM_SAMP_IN_SIG];
NUMTYPE RxRef[NUM_SAMP_IN_REF];

int main()
{

double Gam = 10;
double EbN0dB = 4;
double TxClockError = 1e-5;
double CodeRate = CODE_RATE;
double EbN0, EsN0, SNR, Sigma;
double Pi = M_PI, RxPhase;
double A=1, t=0, dt, delta_tot_phase;
double K1=0.04996, K2=0.001133, K3=9.72e-6, K4=0;

int pll_mode=2; // mode 2 is phase and phase-rate
int dll_step=1; // have dll step this many reference samples at a time
int Niter=10; // max number of turbo algorithm iterations
int Nacq = 0; // no acq symbols to track

NUMTYPE La[NUM_BITS];
NUMTYPE Lall[NUM_BITS];
NUMTYPE SoftSyms[NUM_SYMS];

int TxBits[NUM_BITS];
int RxBits[NUM_BITS];
int Alpha[NUM_BITS];
int TxSyms[NUM_SYMS];
int RxSyms[NUM_SYMS];

int NumBits = NUM_BITS; // info plus tail bits
int NumSyms = NUM_SYMS;
int Nspc = NUM_SAMP_PER_CHIP;
int OS = REF_OVER_SAMP;
int K = CONSTRAINT_LENGTH;

```

```

int Lsig = NUM_SAMP_IN_SIG;
int Lref = NUM_SAMP_IN_REF;
int Ns=NUM_SAMP_PER_SYM;
int Nc = NUM_CHIP_PER_SYM;
int Puncture = PUNCTURE;

int i, n, Nerror;
int delta_clock;
int Nf=OS*8;
int InitDelayEst=Nf+7-1; // hand tweak group delay

float Noise;

// RNGs are set both here for sig_gen and further below for "iters",
// to give same results as run_dppll_then_turbo.m

set_useed( 0); set_nseed( 0);

printf("Ninfo = %d \n", NUM_INFO);
printf("K = %d \n", K);
printf("Puncture = %d, CodeRate = %g \n", Puncture, CodeRate);
printf("NumBits = %d \n", NumBits);
printf("NumSyms = %d \n", NumSyms);
printf("Nc = %d \n", Nc);
printf("Nspc = %d \n", Nspc);
printf("OS = %d \n", OS);
printf("dll_step = %d \n", dll_step);
printf("pll_mode = %d \n", pll_mode);
printf("InitDelayEst = %d \n", InitDelayEst);
printf("Niter=%d\n",Niter);

delta_clock=ceil(TxClockError*Nc*NumSyms*Nspc*OS);
printf("delta_clock = %d ref samps\n",delta_clock);

// Print clock drift as the fractional rate, eg chips/chip, and in terms of
// the number of chips of adjustment that will be required of the DLL.
printf("TxClockError=%g\n",TxClockError);
printf("delta_clock = %1.1f chips\n",(double)delta_clock/(Nspc*OS));

gen_turbo_sig( TxClockError, Nf, Alpha, TxBits, TxSyms, RxChips, RxRef);

EbN0=pow(10,EbN0dB/10);
EsN0=CodeRate*EbN0;
SNR = EsN0/Ns; // Input SNR at A/D bandwidth; Es/N0 - Gain
Sigma = A/sqrt(2*SNR);
printf("EbN0=%g dB\n",EbN0dB);
printf("Lsig = %d, Lref = %d\n", Lsig, Lref);

printf("Gam = %f\n",Gam);
delta_tot_phase=Gam/2; // total cycle of phase, assume sig is 1 sec long
dt=1.0/Lsig; // time step, use this for now
printf( "delta_phase = %f cycles\n", delta_tot_phase);
printf( "delta_phase = %f cycles/sym\n", delta_tot_phase/NumSyms);

```

```

for (i=0; i<NumBits; i++)
    La[i] = 0;

set_used( 0); set_nseed( 3614762644UL); // set RNGs for some particular iter

for (n=0; n<Lsig; n++) {
    RxPhase=Pi*Gam*t*t;
    t += dt;
    if (EbN0dB<100) {
        Noise = Sigma*gasdev();
        RxSigRe[n] = A*RxChips[n]*cos(RxPhase)+Noise;
        Noise = Sigma*gasdev();
        RxSigIm[n] = A*RxChips[n]*sin(RxPhase)+Noise;
    } else {
        RxSigRe[n] = A*RxChips[n]*cos(RxPhase);
        RxSigIm[n] = A*RxChips[n]*sin(RxPhase);
    }
}

dll( SoftSyms, pll_mode, InitDelayEst, RxSigRe, RxSigIm, dll_step,
    RxRef, Nacq, K1, K2, K3, K4, OS);

// for (i=0; i<NumSyms; i++)
// printf(“%f\n”,SoftSyms[i]);

Nerror=0;
printf(“symbol error indices: “);
for(n=0; n<NumSyms; n++) {
    RxSyms[n] = (0<SoftSyms[n]) ? 1 : -1;
    if (RxSyms[n]!=TxSyms[n]) {
        Nerror++; // which syms were in error ?
        printf(“%d “,n+1); // number syms from 1 to compare to Matlab
    }
}
printf(“\n”);
printf(“Nerr=%d, SER=%f\n”,Nerror, (double)Nerror/NumSyms);

turbo( Lall, SoftSyms, Alpha, Niter, TxBits);

// for(n=0; n<5; n++) {
// printf(“Lall[%d] = %20.14e \n”, n, Lall[n]);
// }

Nerror=0;
printf(“bit error indices: “);
for(n=0; n<NumBits; n++) {
    RxBits[n] = (0<Lall[n]) ? 1 : 0;
    if (RxBits[n]!=TxBits[n]) {
        Nerror++; // which bits were in error ?
        printf(“%d “,n+1); // number bits from 1 to compare to Matlab
    }
}
printf(“\n”);

```

```
printf("Nerr=%d, BER=%f\n",Nerror, (double)Nerror/NumBits);  
return(0);  
}
```

DISTRIBUTION

Unlimited Release

1	MS 0505	S. M. Becker	2341
5	MS 0505	V. Guzman Kammler	2341
1	MS 0509	M. W. Callahan	2300
1	MS 0519	L. M. Wells	2344
1	MS 0519	J. T. Cordaro	2344
1	MS 0519	A. W. Doerry	2344
1	MS 0519	B. D. Guess	2344
1	MS 0519	G. B. Haschke	2344
5	MS 0519	J. J. Mason	2344
5	MS 0519	R. C. Ormesher	2344
1	MS 0529	B. L. Remund	2340
1	MS 0529	B. L. Burns	2340
1	MS 0529	W. H. Hensley	2348
1	MS 0529	M. B. Murphy	2346
1	MS 0529	A. Martinez	2346
1	MS 0529	C. W. Ottesen	2346
1	MS 0529	K. W. Plummer	2346
1	MS 0529	D. A. Wiegandt	2346
1	MS 0529	K. W. Sorenson	2345
1	MS 0529	D. F. Dubbert	2345
1	MS 0965	J. A. Heise	5711
1	MS 0972	C. A. Boye	5710
1	MS 0980	R. Mata	5711
1	MS 0980	R. M. Axline	5711
1	MS 0980	T. D. Atwood	5711
1	MS 1155	J. A. Ramos	5532
1	MS 1155	D. D. Cox	5532
1	MS 9018	Central Technical Files	8945-1
2	MS 0899	Technical Library	9616
1	MS 0323	LDRD Office	1011
1	MS 0161	Patent and Licensing Office	11500