RECONFIGURABLE SIGNAL PROCESSOR FOR CHANNEL CODING & DECODING IN LOW SNR WIRELESS COMMUNICATIONS

Steve Halter[†], Mats Öberg[*], Paul M. Chau[†], Paul H. Siegel[*]
[†]ICAS Center, University of California, San Diego
[*]Center for Wireless Communications, University of California, San Diego

Abstract - An area and computational-time efficient turbo decoder implementation on a reconfigurable processor is presented. The turbo decoder takes advantage of the latest sliding window algorithms to produce a design with minimal storage requirements as well as offering the ability to configure key system parameters via software. The parameter programmability allows the decoder to be used in a research environment to study less understood aspects of turbo codes.

1. INTRODUCTION

The flexibility and quick response of FPGA technology to the changing market scene has led it to be utilized to perform a variety of different tasks. In the process, a natural progression in terms of usage has quietly been transpiring. At first, systems were used primarily for their rapid prototyping capability in industry and system emulation in academia, and this remains largely the case today. However, as more people become aware of the applications of configware, this is beginning to change. Furthermore, as partial reconfigurable FPGAs with increasing circuit density become available, the technology will progress from dynamic to the more demanding on-the-fly reconfiguration necessary for the next generation of smart, adaptive hardware.

Configware represents a hybridization of the conventional usage of standard central processing units (CPUs) and full custom ASICs. Innovative hardware platforms, such as the UCSD-L3 ReConfigurable Processor Board (RCP) [1], blur the traditional boundaries that exist between hardware and software to bring out the best features of both worlds: a system that is blazingly fast but extremely adaptive to its operating environment. Configware systems like the RCP are fundamentally defined by their ability to adapt or completely reconfigure themselves, making them suitable for a whole host of traditional adaptive algorithms. Although current generation FPGAs do not have the circuit density nor low power advantages as do custom ASICs for small form factor wireless products, they can still significantly reduce the footprint in the other electronics in wireless and satellite communication (SATCOM) such as in the mobile base or ground stations.

The RCP has been designed to be a flexible processor and internal interconnect reconfigurable application-specific embedded computer. Current and forthcoming algorithms incorporated into the RCP include variable constraint length convolutional forward error correction, wavelet image compression, data encryption/decryption, and common signal processing algorithms such as filtering and fast Fourier transforms. A novel feature of the RCP is the capability to facilitate the optimization of the partitioning, mapping and scheduling of communications signal processing algorithms through Index Mapping [2], a processor-to/from-memory interconnection mapping and scheduling technique developed at UCSD. This method, applicable to both systolic and non-systolic parallel-pipelined datapath processing, allows the characterization of the intermediate algorithmic processing interconnection, memory and dataflow configuration, and scheduling. This characterization can then be transformed to optimize a desired figure of merit, such as area, speed, or power.

In this paper, we describe a flexible RCP-based architecture for an advanced forward-error-correction coding technique known as turbo coding. Turbo codes, introduced in 1993 [3], represent perhaps the most significant advance in coding for digital communications since the development of trellis-coded modulation in the early 1980's as they have been shown to achieve performance very close to the information-theoretic limits determined 50 years ago by Claude Shannon. Consequently, there has been an explosion of research on the theoretical and practical aspects of turbo coding, and it is expected that these techniques will find wide use in low signal-to-noise ratio applications, ranging from deep-space communications to cellular mobile radio systems.

A flexible, hardware embodiment of a turbo decoder, as described in this paper, may be used to investigate many of the design parameters that affect turbo code performance, including constituent convolutional encoder polynomials, interleaver mappings [4], puncturing rules [5], and information frame length. Issues pertaining to decoder convergence and stopping rules are also amenable to experimental study with this design. In addition, the hardware implementation provides a vehicle for evaluating the space-time-performance trade-offs of proposed architectures for various elements of a turbo coding system.

The outline of the paper is as follows. In Section 2, we review the important features of the RCP system. In Section 3, we describe the principles of turbo code design, as well as some of their planned and proposed applications in future digital communications systems. In Section 4, we present the details of the flexible turbo decoder implementation on the RCP. In Section 5, we summarize the paper and indicate directions for future research based upon our turbo decoder implementation .

2. RCP OVERVIEW

The ReConfigurable Processor Board (RCP) is designed as a flexible, high internal bandwidth application specific coprocessor that fits into a PC chassis utilizing a Peripheral Component Interconnect (PCI) interface. Its primary advantage over other systems is its ability to adapt to perform multiple functions efficiently. It is envisioned that the RCP can revolutionize satellite telemetry and data transfer tasks by providing extremely fast and flexible digital signal processing (DSP) functionality for forward error correction, compression/decompression, encryption/decryption and other necessary tasks required for communication over satellite links.

The RCP board layout consists of four Altera FLEX 10K70 FPGA processing elements that are connected to each other through direct local buses and through four field programmable interconnect ICube IQX160 PSID chips. These processing elements each have their own fast dual port 32k x 16 RAM that can be directly accessed by the PSIDs. Two more Altera 10K70 processing elements are utilized to take care of pre- and post-processing tasks such as data conversion, pruning and sorting. These processors are connected to the PCI bus through 4k x 18 FIFOs. The post-processor is further connected to a 1 M x 32 single port RAM for data storage of array calculation results and for storing additional data specifically for packet burst communications. Dedicated PCI control logic is used for programming, configuration and PCI interface tasks. A global bus connects all the processors together to a Altera PCI 10K20 device. Figure 1 is a picture of the RCP board.

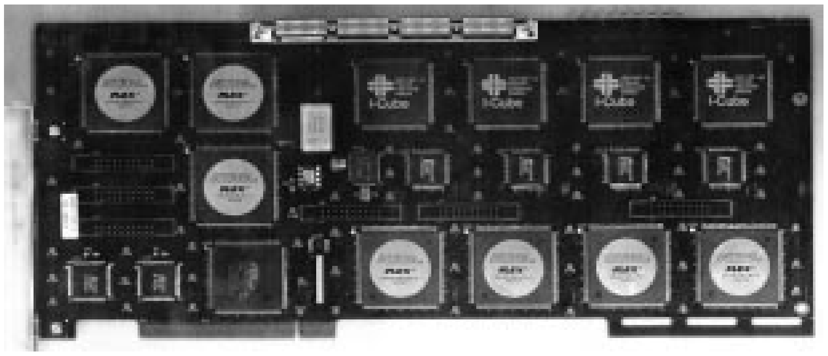


Figure 1: RCP Board

3. TURBO CODING FOR LOW SNR COMMUNICATIONS

Turbo codes were introduced by Berrou, Glavieux, and Thitimajshima [3] [6] in 1993 and are now widely recognized as a landmark development in error control coding. By combining a concatenation of convolutional codes, connected by an interleaver, with an iterative decoding algorithm, these codes achieve performance very close to information-theoretic limits. For example, in [3], a rate 1/2 turbo code was shown via simulation to achieve a bit error rate (BER) of 10^{-5} at a signal-to-noise ratio (SNR) of $E_b/N_0 = 0.7$ dB, only 0.5 dB from the theoretical limit for rate 1/2 codes on binary input Gaussian channels [7]. In comparison, the far more complex code designed for the Galileo spacecraft, consisting of a memory 14 convolutional inner code and a 16-byte Reed-Solomon outer code [8], requires SNR of about $E_b/N_0 = 1.0$ dB at rate R=0.22 for comparable error-rate performance, which corresponds to a gap of almost 2 dB from the Shannon-theoretic limit for that rate.

3.1 Background on Turbo Codes: Encoding and Decoding

A block diagram of a turbo coding system is shown in Figure 2. The turbo encoder is implemented with two recursive, systematic convolutional (RSC) encoders in parallel concatenation. A frame of *N* information bits is encoded by the first encoder, while the interleaver creates a prespecified, random-like permutation of the information, which is then encoded by the second encoder. The transmitted code sequence consists of the information bits along with the parity bits produced by the two encoders. Puncturing, or periodic deletion, of the parity bits is sometimes used to increase the overall code rate. The interleaver endows the turbo code with structural properties similar to those of a random block code, which Shannon proved could, on average, achieve performance close to the information-theoretic limit with maximum-likelihood decoding.

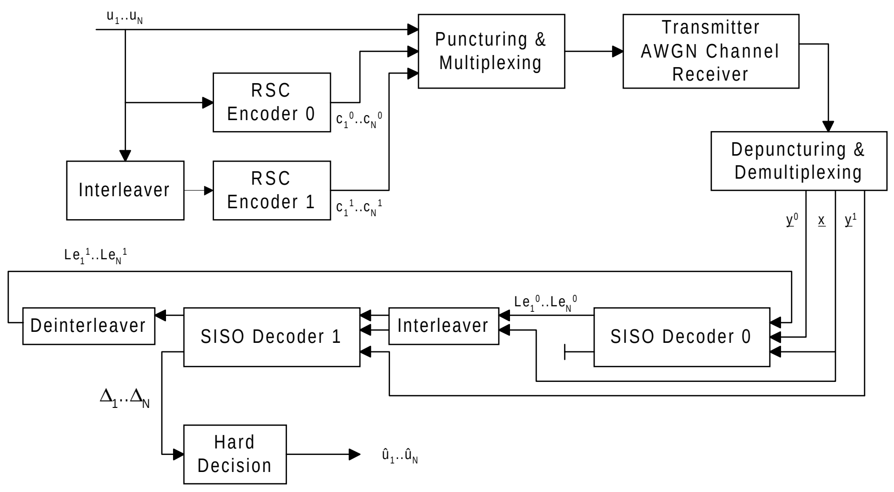


Figure 2: Turbo Code Encoder, Channel, and Turbo Code Decoder

However, the implementation of a maximum-likelihood (ML) sequence decoder for the turbo code would be prohibitively complex. Instead, a sub-optimal, but simple and effective, decoding architecture is used [3]. The decoder incorporates separate soft-input, soft-output decoders for each of the constituent convolutional codes, operating in an iterative and cooperative manner.

Each constituent decoder generates soft outputs in the form of *a posteriori* probabilities (APP) for the information bits. From these probabilities, the decoder extracts “extrinsic information” values that are provided to the other decoder as soft inputs that play the role of *a priori* probabilities for the information bits. At the start of the decoding, the noisy channel outputs corresponding to the information bits are used to initialize the prior probabilities. They are also available to each constituent decoder throughout the decoding procedure. The number of repetitions of this cycle of decoding and exchange of the extrinsic information is dictated by some stopping rule, often a prespecified limit on the number of iterations. The turbo decoder output is a hard-quantized log-APP ratio of an information bit u_n produced by the final decoding cycle.

More precisely, the *a posteriori* log-likelihood ratio (LLR) of an information bit u_n is expressed as

$$\Lambda_n = \log \frac{\Pr(u_n = 1 | observation)}{\Pr(u_n = 0 | observation)} \tag{1}$$

where we use *observation* to denote the noisy channel output sequence. The implementation of the APP calculation can use a modification of the BCJR algorithm, which was introduced by Bahl, Cocke, Jelinek, and Raviv [9] in the context of maximum-a-posteriori (MAP) bit decoding. The algorithm computes

$$\Lambda_n = \log \frac{\Pr(u_n = 1 | R_1^N)}{\Pr(u_n = 0 | R_1^N)} \tag{2}$$

where $R_1^N = (R_1, R_2, \ldots, R_N)$ denotes the received samples or observations, with $R_i = (x_i, y_i, L_i^{ext})$ consisting of the information samples x_i, the parity bit samples y_i, and the extrinsic information L_i^{ext}. The APP decoder computes the *a posteriori* probabilities

$$\Pr(u_n = i | R_1^N) = \frac{1}{\Pr(R_1^N)} \sum_{m,m':u_n=i} \Pr(u_n = i, S_n = m, S_{n-1} = m', R_1^N) \cdot \tag{3}$$

Here S_n refers to the state at time n in the trellis of the constituent convolutional code.

The terms in the summation can be expressed in the form

$$\Pr(u_n = i, S_n = m, S_{n-1} = m', R_1^N) = \alpha_{n-1}(m')\gamma_n^i(m', m)\beta_n(m), \tag{4}$$

where the quantity

$$\gamma_n^i(m', m) = \Pr(S_n = m, u_n = i, R_n | S_{n-1} = m') \tag{5}$$

is called a *branch metric*,

$$\alpha_n(m) = \Pr(S_n = m, R_1^n) \tag{6}$$

is called a *forward state metric*, and

$$\beta_n(m) = \Pr(R_{n+1}^N | S_n = m) \tag{7}$$

is called a *reverse* (or *backward*) *state metric*.

The branch metric depends upon the nominal information and parity bits labeling the trellis branch from state m' to state m, the noisy samples, and the extrinsic information provided by the other decoder. The forward and reverse state metrics are computed recursively by forward and backward recursions given by

$$\alpha_n(m) = \sum_{m',i} \alpha_{n-1}(m')\gamma_n^i(m',m)\,, \tag{8}$$

and

$$\beta_{n-1}(m') = \sum_{m,i} \beta_n(m)\gamma_n^i(m',m)\,. \tag{9}$$

In the implementation of the APP calculation, further simplifications are often used to reduce the computational complexity, as described in Section 4.1.

3.2 Turbo Code Performance and Applications

The simulated performance of a rate 1/2 turbo code, with constituent RSC encoder polynomials $(37,21)_{\text{octal}}$ and a pseudo-random interleaver of length N=10,000, is shown in Figure 3 below. A BER of 10^{-5} is achieved at a SNR of approximately 1 dB. For purposes of comparison, the figure also shows the performance of a rate 1/2, memory 14, maximum free-distance (2,1,14) convolutional code [10]. The modified MAP decoders for the turbo code require only 16 states each, while the ML decoder for the (2,1,14) convolutional code requires 2^{14} states. The performance improvement offered by the turbo code at low SNR is evident.

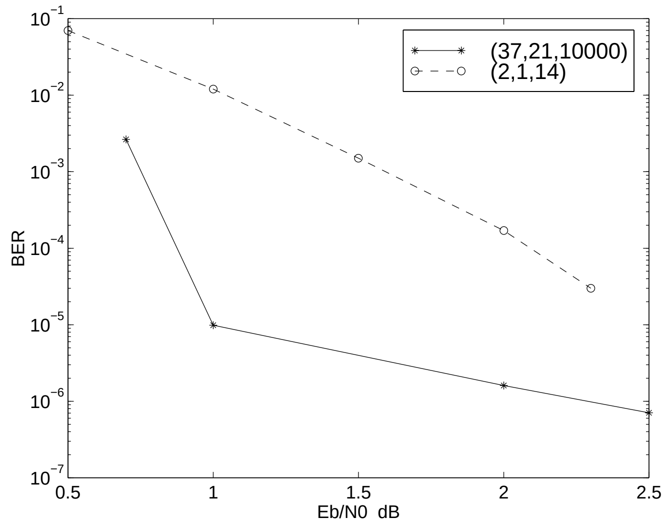


Figure 3: Performance of a rate ½ turbo code

The remarkable performance of turbo codes opens up a huge array of possible applications. Turbo codes have been shown to be viable for mobile radio communications in, for example, bearer services defined in the pan European FRAMES project [11]. Rowitch et al. [12] proposed a hybrid rate-compatible punctured turbo coding/ARQ scheme, which showed higher throughput than previous schemes based on conventional convolutional codes.

For latency critical applications such as two-way vocal communication in personal communication systems (PCS), the performance advantage provided by large interleaver lengths may be offset by unacceptable decoding delay. Divsalar and Pollara [13] have derived turbo codes with greatly reduced interleaver sizes that might be suitable for PCS applications. Turbo codes are currently being examined for use in the next generation of Code-Division Multiple-Access (CDMA) wireless systems.

4. TURBO DECODER IMPLEMENTATION ON THE RCP

The goal of the design is the implementation of a turbo decoder with a minimum rate of 1/3, which can be realized in the RCP FPGA array for prototyping and later be mapped to an ASIC with minimal storage requirements. Turbo decoders require the use of soft-input, soft-output (SISO) decoders in order to achieve good error-rate performance, and this motivates the use of MAP decoders, which were described in Section 3.1. Unfortunately, the MAP decoding algorithm is significantly more complex than the Viterbi algorithm because its implementation requires memory for metric storage that is proportional to the code block length. SISO Viterbi decoders, which have lower complexity than MAP decoders, have therefore been considered; however, it has been shown that their performance is inferior to that of MAP decoders [14].

Recently, Viterbi [15] proposed a MAP decoder architecture employing a “sliding window” technique which decouples the required memory size from the code block length. This permits the use of large interleavers, which give the best performance, without requiring prohibitively large memory resources. The sliding window architecture will be used for the MAP decoder implementation in this turbo decoder design.

Another issue arising in the implementation of the traditional MAP decoding algorithm is the extensive use of multiplication and exponentiation functions, which do not translate efficiently into hardware. Therefore, the turbo decoder design uses the log-MAP algorithm [16], obtained by transforming the MAP algorithm to the log-domain. This form of the algorithm requires only additions and eccumulation functions, which can be easily mapped to hardware.

4.1 Log-MAP Algorithm

In the interest of describing the algorithm being implemented, the mathematical formulas for the log-MAP algorithm are presented. For derivations of the following equations, refer to [3] and [16]. The log-likelihood ratio can be calculated based on an eccumulation of the forward and reverse state metrics and the branch metrics:

$$L_k = \mathop{\mathrm{E}}_{m=0}^{2^{\upsilon}-1}\left(A_k^m + D_k^{0,m} + B_{k+1}^{f(0,m)}\right) - \mathop{\mathrm{E}}_{m=0}^{2^{\upsilon}-1}\left(A_k^m + D_k^{1,m} + B_{k+1}^{f(1,m)}\right) \qquad (10)$$

where υ is the encoder memory, *m* is the trellis state, *f(j,m)* is the next state based on the input *j* going forward in time, A_k are the forward state metrics, B_k are the reverse state metrics, D_k are the branch metrics, and eccumulation is defined as:

$$a \,\mathrm{E}\, b = \min(a,b) - \log_{\varepsilon}\left(1+\varepsilon^{-|a-b|}\right). \qquad (11)$$

The state metrics are recursive calculations of branch metrics and the previous state metrics. They can be determined by forward and backward recursions through the trellis:

$$A_k^m = \mathop{\mathrm{E}}_{j=0}^{1}\left(A_{k-1}^{b(j,m)} + D_{k-1}^{j,b(j,m)}\right) \qquad (12)$$

$$B_k^m = \mathop{\mathrm{E}}_{j=0}^{1}\left(B_{k+1}^{f(j,m)} + D_k^{j,m}\right) \qquad (13)$$

$$D_k^{i,m} = -K_k - (z_k + x_k)i - y_k c^{i,m} \qquad (14)$$

where K_k is a constant, *b(j,m)* is the next state based on the input *j* going backwards in time, $c^{i,m}$ is the parity bit from state *m* given input bit *i*, and z_k is the input APP. Equation (15) represents another form of the log-likelihood ratio:

$$L_k = z_k + x_k + z_{k+1} \qquad (15)$$

Using the value calculated from (10), the output APP z_{k+1} can be determined from (15).

4.2 Viterbi Sliding Window Architecture for MAP Decoders

The central concept behind the Viterbi Sliding Window architecture (VSWA) is that the reverse state metric calculation, which is performed via a backward recursion through the trellis, does not have to start from the last time step of the trellis. Through the use of a sliding window of some length *L* and starting from some time point *k* in the trellis, reverse state metric calculations through *L* time steps will produce a good approximation of the reverse state metrics at time step *k-L*. The next *L* reverse state metrics can then be calculated starting from the approximation at time *k-L*. Hence, it is possible to “build-up” good approximations of reverse state metrics starting from any point in the trellis.

Without the use of this sliding window technique either all of the reverse or all of the forward state metrics would have to be stored while the other is being calculated. In the case of a code with a block length of 16,384 and an encoder of memory 7, the required state metric storage would be over 2M x *q*, where *q* is the quantization value of the state metrics. With the sliding window technique, the storage requirement for forward state metrics is now a function of the window length, *L*, equal to $L2^{v+1}$, where v is the memory length of the encoder. For comparison, using this sliding window technique, the state metric storage for a code of arbitrary block length, encoder memory 7 and window length *L*=64 would be no more than 16k x *q*. The window length *L* will be one of the programmable parameters in this design.

Viterbi [15] also suggests a method of implementing the sliding window technique that prevent pipeline delays through the use of dual reverse state metric calculators. Table 1 shows the timing information for the various metric calculators, where FSMC is the forward state metric calculator, RSMC0 and RSMC1 are the reverse state metric calculators, and LLC is the log-likelihood ratio (LLR) calculator. The shaded areas of the table indicate where the RSMCs are constructing approximate reverse state metrics; the non-shaded areas indicate where they are generating valid reverse state metrics. The LLR at time *k* is denoted λ_k, and the notation λ_x-λ_y in the table indicates that the LLRs for time steps *x* through *y* are being generated.

The table also shows the contents of the memory storing the forward state metrics. The notation $A_{x,y}$ indicates that the forward state metrics for time steps *x* through *y* are currently being stored in memory.

Time	**2L**	**3L**	**4L**	**5L**	**6L**	**7L**	**8L**
FSMC	0-L	L-2L	2L-3L	3L-4L	4L-5L	5L-6L	
RSMC0	2L-L	L-0	4L-3L	3L-2L	6L-5L	5L-4L	
RSMC1		3L-2L	2L-L	5L-4L	4L-3L		6L-5L
Output		λ_L-λ_0	λ_{2L}-λ_L	λ_{3L}-λ_{2L}	λ_{4L}-λ_{3L}	λ_{5L}-λ_{4L}	λ_{6L}-λ_{5L}
Memory	$A_{0,L}$	$A_{0,2L}$	$A_{L,3L}$	$A_{2L,4L}$	$A_{3L,5L}$	$A_{4L,6L}$	$A_{4L,6L}$

Table 1: Viterbi Sliding Window Architecture Pipeline Timing

4.3 Architecture

4.3.1 Overview Due to resource limitations on the RCP board, only one MAP decoder is instantiated in this design. However, this “restriction” in the architecture actually does not reduce the flexibility of the system. Lowering the non-punctured rate of a turbo code simply adds decoding stages to the iterative loop. Since all the MAP decoders are identical, this architecture can be easily expanded to support lower rate turbo codes with the only cost being additional control logic and computation time.

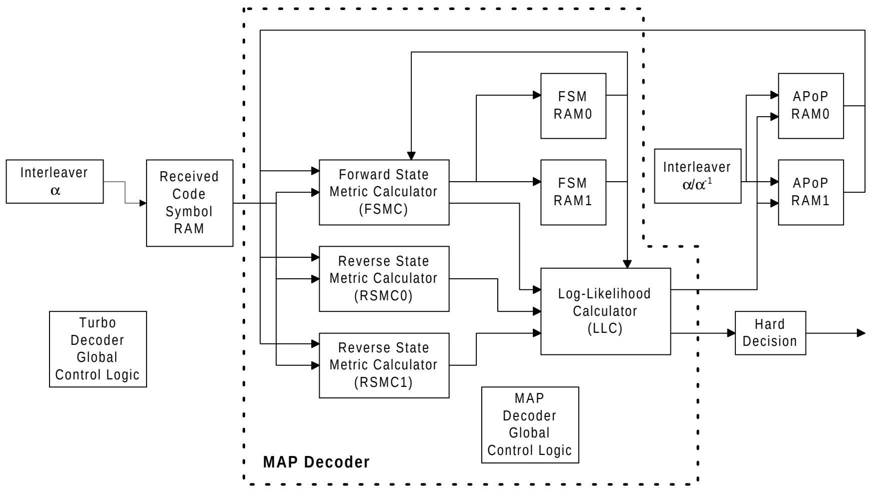

Interleaver
α
Received Code Symbol RAM
Forward State Metric Calculator (FSMC)
Reverse State Metric Calculator (RSMC0)
Reverse State Metric Calculator (RSMC1)
FSM RAM0
FSM RAM1
Log-Likelihood Calculator (LLC)
Interleaver
α/α⁻¹
APoP RAM0
APoP RAM1
Hard Decision
Turbo Decoder Global Control Logic
MAP Decoder Global Control Logic
MAP Decoder

Figure 4: Turbo Decoder Block Diagram

This implementation of a turbo decoder consists of the three state metric calculators (SMC) and the log-likelihood calculator (LLC) which constitute the MAP decoder, interleaver/deinterleaver memory, the temporary storage memory for the state metrics, and the hard decision block. In order to reduce computational complexity of the MAP decoder algorithm, the log-MAP algorithm is implemented. The architecture is designed to complete one MAP decoder stage at a time based on an inner-outer loop algorithm, where the outer loop steps through time, and the inner loop steps through the trellis states. The term “time instance” will be used frequently in describing the architecture. A “time instance” is one cycle of the outer loop or one set of trellis states. Figure 4 shows a high-level block diagram of the turbo decoder system.

4.3.2 State Metric Calculators (SMC) The state metric calculators are independent engines, running in parallel, with each metric calculator computing a single state metric every three clock cycles. The branch metrics are calculated on-the-fly with each SMC containing a pair of branch metric calculators. A branch metric calculator also yields a single branch metric every three clock cycles. The branch metrics could be computed once and stored for future use; however, a design trade-off was made in favor of a small amount of additional power consumption as opposed to significantly increased storage requirements. Due to the pipelined architecture, the additional computation time of computing the branch metrics on-the-fly is negligible except in cases of very small sliding window lengths with low encoder memory where the minimal overhead starts to become significant. The following diagram shows the datapath of a branch metric calculator.

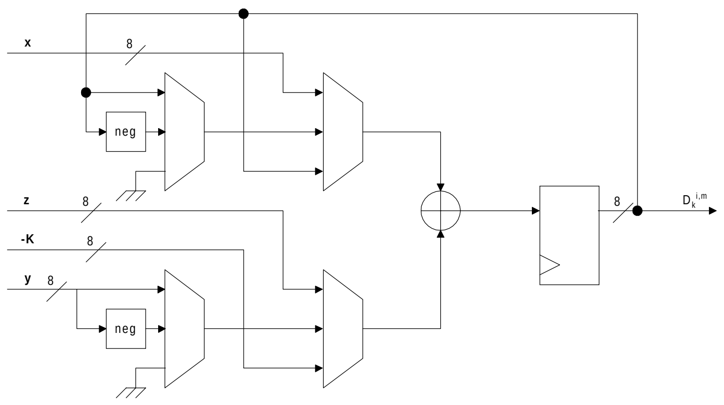


Figure 5: Branch Metric Calculator Datapath

4.3.2.1 Add-Eccumulate As revealed by examining the structure of the forward and reverse state metric equations of the log-MAP function, both metrics can be computed by add-eccumulation functions, where the add-eccumulation is analogous to the common multiply-accumulate function present in digital signal processors. The following diagrams show a high-level block diagram of the datapath of the 16-bit add-eccumulate block.

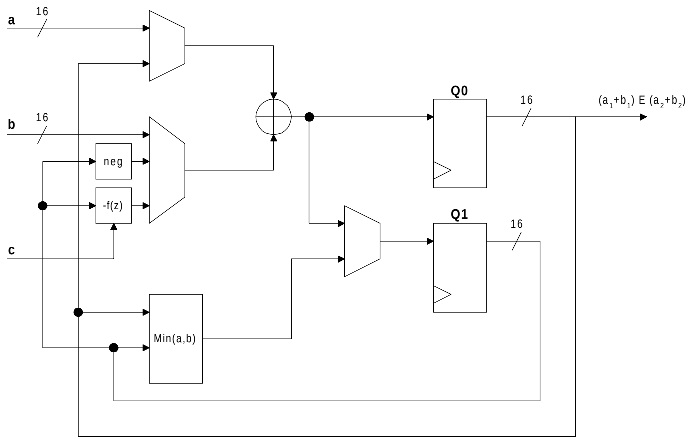


Figure 6: Add-Eccumulate Unit Datapath

The *log(·)* portion of the eccumulate function (11) is designated as *f(z)* in the block diagram above where $z = |a\text{-}b|$ and the input *c* is equal to *1 / ln(ε)*. The *f(z)* block is a series of lookup tables for various values of *c*.

The add-eccumulate block performs add-eccumulation in three clock cycles.

4.3.2.2 Forward State Metric Calculator (FSMC) The FSMC consists of a pair of branch metric calculators with a pair of add-eccumulate units and computes two state metrics every six clock cycles. The first three clock cycles are used to add the first pair of previous forward state metrics to their respective branch metrics, and the final three clock cycles are used to do the add-eccumulate with the next pair of previous forward state metrics and their respective branch metrics. The FSMC utilizes a pipelined architecture to enable the add-eccumulate units to calculate a pair of state metrics, while the branch metric calculators are determining the branch metrics from the next set of state metrics. This prevents the on-the-fly branch metric calculation from impacting the computation time.

In order to save power by reducing the amount of fetching of previous state metrics, the FSMC uses a butterfly calculation which is frequently used in Viterbi decoder implementations. The butterfly calculation exploits the fact that the states of the convolutional code trellis can be grouped into pairs of states with identical outgoing branch properties, except for their associated input bits. Figure 7 shows an example of a butterfly.

The forward state metrics of states S_1' and S_2' both require the forward state metrics of states S_1 and S_2. By using this butterfly technique, the forward state metrics of S_1' and S_2' are both calculated but the state metrics of S_1 and S_2 are only fetched once. This will yield a 2x reduction in the number of fetches from memory which can lead to a substantial savings in power consumption.

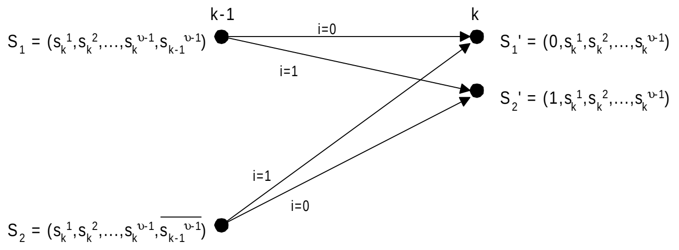

k-1
k
i=0
i=1
i=1
i=0
$S_1 = (s_k^1, s_k^2, \ldots, s_k^{\upsilon-1}, s_{k-1}^{\upsilon-1})$
$S_2 = (s_k^1, s_k^2, \ldots, s_k^{\upsilon-1}, \overline{s_{k-1}^{\upsilon-1}})$
$S_1' = (0, s_k^1, s_k^2, \ldots, s_k^{\upsilon-1})$
$S_2' = (1, s_k^1, s_k^2, \ldots, s_k^{\upsilon-1})$

Figure 7: Forward State Metric Butterfly

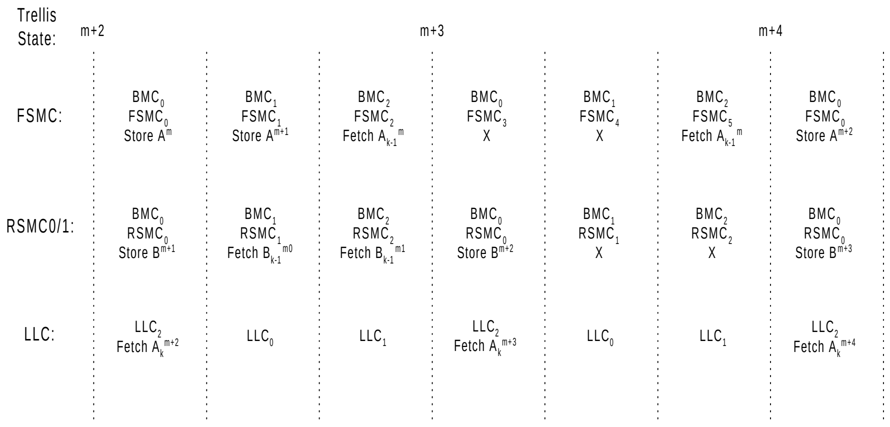
The FSMC uses a pair of storage RAMs in order to prevent RAM resource contention between the FSMC and log-likelihood calculator, which also needs to fetch forward state metrics. Each FSMC storage RAM stores the forward state metrics for a single sliding window.

4.3.2.3 Reverse State Metric Calculator (RSMC) The RSMC is very similar to the FSMC in the sense that it features a pair of branch metric calculators and produces a state metric every three clock cycles. However, unlike the FSMC, the RSMC only contains a single add-eccumulate unit in addition to a single adder. The second add-eccumulate unit can be simplified in the RSMC due to the fact the branch metrics are derived from the same trellis state whose reverse state metric is being calculated. The RSMC also uses a butterfly technique similar to the technique used in the FSMC to reduce the number of memory fetches.

The RSMC also contains a small block of internal memory for previous RSMC storage. The VSWA requires only the reverse state metrics from the previous time instance to be stored.

4.3.3 Log-Likelihood Calculator (LLC) The LLC is an independent engine which operates in parallel with the FSMC and RSMCs. Due to the fact that the log-likelihood function is also an add-eccumulate calculation, the LLC consists of a pair of add-eccumulate units which are used to eccumulate the two halves of the log-likelihood function. While the RSMCs and FSMCs are running, the LLC is adding and eccumulating the forward and reverse state metrics accordingly. When the FSMC and RSMCs have finished calculating state metrics for all trellis states for a given time instance, the LLC performs the subtraction to calculate the log-likelihood ratio and then performs two additional subtractions in order to calculate the APP value for the current time instance. The APoP RAMs in the block diagram are used to hold the APP values of the current MAP decoder as well as the values from the previous MAP decoder. Each APoP RAM has storage equal to the block length of each turbo decoder iteration. Upon completion of the LLC calculations, the log-MAP decoder moves on to the next time instance.

4.3.4 System Pipeline Figure 8 shows the steady-state pipeline of the log-MAP decoder. The subscripts to the various calculators show the current clock cycle. For example, $FSMC_1$ indicates the FSMC is in its second clock cycle. It is important to note that, as shown in the VSWA discussion, the FSMC and RSMCs are not processing the same trellis states at any given time.


Trellis State:
m+2
m+3
m+4
FSMC:
RSMC0/1:
LLC:
BMC_0 $FSMC_0$ Store A^m
BMC_1 $FSMC_1$ Store A^{m+1}
BMC_2 $FSMC_2$ Fetch $A_{k-1}{}^m$
BMC_0 $FSMC_3$ X
BMC_1 $FSMC_4$ X
BMC_2 $FSMC_5$ Fetch $A_{k-1}{}^m$
BMC_0 $FSMC_0$ Store A^{m+2}
BMC_0 $RSMC_0$ Store B^{m+1}
BMC_1 $RSMC_1$ Fetch $B_{k-1}{}^{m0}$
BMC_2 $RSMC_2$ Fetch $B_{k-1}{}^{m1}$
BMC_0 $RSMC_0$ Store B^{m+2}
BMC_1 $RSMC_1$ X
BMC_2 $RSMC_2$ X
BMC_0 $RSMC_0$ Store B^{m+3}
LLC_2 Fetch $A_k{}^{m+2}$
LLC_0
LLC_1
LLC_2 Fetch $A_k{}^{m+3}$
LLC_0
LLC_1
LLC_2 Fetch $A_k{}^{m+4}$
Figure 8: Steady-State Turbo Decoder Pipeline

4.4 System Parameters

As mentioned before, the architecture is designed to support programmable system parameters in order to allow their effects on performance to be measured in a research environment. Table 2 shows the range of the programmable system parameters.

Parameter	Range
Number of Encoder States ($m = 2^{\upsilon}$):	2 – 512
Sliding Window Length (L):	1 – 16384
Iteration Block Length (n):[1]	1 – 65536
Decoder Iterations (j):	1 – 512
Code Symbol Quantization (in bits):	1 – 8
Branch Metric Quantization (in bits):	1 – 8
State Metric Quantization (in bits):	1 – 16
Log-Likelihood Quantization (in bits):	1 – 16
APP Quantization (in bits):	1 – 8

[1]Note that the iteration block length is limited by the receive data storage on the RCP, not by the design.

Table 2: Programmable System Parameters

The only constraint on the system is $L2^{\upsilon+1} < 65536$, which is the maximum available forward state metric storage. The quantization factors were determined based on the predefined storage resources on the RCP board. The encoder generator matrix is also programmable via software, allowing the turbo decoder to support any convolutional encoder with a maximum of 512 states. The interleavers and deinterleavers, in order to support any possible type of interleaver, are implemented in RAM.

The system storage requirements are summarized in Table 3.

Data	Max. Storage Requirements
Forward State Metrics:	$L2^{\upsilon+1}$ x 16
Reverse State Metrics (per RSMC):	$2^{\upsilon+1}$ x 16
APP Storage:	$2^{\upsilon+1}$ x 8
Received Code Symbols:	N x 8

Table 3: Maximum Storage Requirements

4.5. Computation Performance and Area

The computation performance in terms of clock cycles / information bits can be determined by the following formula:

$$\frac{clock\ \ cycles}{info.bits} = (10 + (3 \times encoder\ \ states)) \times (\#of\ \ SISO\ \ decoders) \times (iterations)$$

The term 10 in the equation is the decoder overhead and the factor 3 is the number of clock cycles needed to calculate a single state metric.

The turbo decoder implementation uses approximately 70,000 gates and has an estimated critical path delay equivalent to the delay through two 3-input 16-bit multiplexers added to the delay of a 16-bit adder. The maximum frequency of the design can be determined when the critical path information is applied to the respective performance specifications of the fabrication process or programmable gate array desired. It is estimated that at least removing the programmable parameter features of the design could eliminate 20% of the gate count.

The decoder has been shown to be functionally correct via software simulations of the VHDL code. At the time of writing, the decoder is being integrated into the RCP board and performance simulations will soon be available.

5. CONCLUSIONS

In this paper, an RCP-based turbo decoder implementation has been presented which offers the unique feature of system-parameter programmability while still maintaining an area, power, and computation-time efficient design. This design can facilitate the study of viable turbo coding systems for commercial applications though the use of parameter programmability, in addition to offering the ability to process amounts of data which are simply not feasible with computer simulations.

The RCP turbo decoder architecture can also find use in other applications exploiting the turbo decoding principle, such as combined channel equalization and decoding for fading and intersymbol interference (ISI) channels [17], as well as serial and hybrid concatenation of convolutional and/or block error correcting codes [18][19].

ACKNOWLEDGMENTS

The authors are grateful for the assistance of Jeanette Arrigo, Dr. Kevin Page of Xenotran, Michael Crawford, Farhad Razavian and Farbod Samadian of L3 Communications and the California MICRO program in the development of the RCP. The authors also acknowledge the support provided by the NSF ICAS Center and the Center for Wireless Communications at UCSD. P. H. Siegel also acknowledges support by NSF grant NCR-9612802.

REFERENCES

[1] K.J. Page, J. Arrigo, and P. M. Chau, “ReConfigurable hardware based digital signal processing for wireless communications,” *Proc. of the SPIE: Adv. Signal Processing Algorithms, Architectures & Impl. VIII*, July 1997.

[2] K.J. Page, and P.M. Chau, “Index mapping for reconfigurable communications architectures,” 4th Reconfigurable Architectures Workshop (RAW-97) part of the 11th Int. Parallel Processing Symp. (IPPS-97), held April 1 - 5, 1997 at University of Geneva, Switzerland.

[3] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit error-correcting coding and decoding: turbo codes,” *Proc. 1993 IEEE Int. Conf. Commun. (ICC)*, Geneva, Switzerland, pp. 1064-1070, May 1993

[4] M. Öberg, and P.H. Siegel, “Application of distance spectrum analysis to turbo code performance improvement,” *Proc. 35th Allerton Conf. Commun. Contr. Comp.*, Monticello, IL, pp 701-710, Sept.-Oct. 1997.

[5] M. Öberg, A. Vityaev, and P.H. Siegel, “The effect of puncturing in turbo encoders,” *Proc. Int. Symp. on Turbo Codes & Related Topics*, Brest, France, pp. 184-187, Sept. 1997.

[6] C. Berrou, and A. Glavieux, “Near Shannon limit error-correcting coding and decoding: turbo codes,” *IEEE Trans. Commun.*, vol. 44, pp. 1261-1271, Oct. 1996.

[7] S. A. Butman, and R. J. McEliece, “The ultimate limits of binary coding for a wideband Gaussian channel,” DSN PR 42-22, May and June 1974, pp. 78-80, Aug. 15, 1974.

[8] D. J. Costello, Jr., and G. Cabiel, “The effect of turbo codes on Figure 1”, *Proc. Inform. Theory Workshop,* San Diego, CA, pp. 41-42, Feb. 1998.

[9] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, “Optimal decoding of linear codes for minimizing symbol error rate,” *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 248-287, Mar. 1974.

[10] L. Perez, J. Seghers, and D. J. Costello Jr., “A distance spectrum interpretation of turbo codes,” *IEEE Trans. Inform. Theory*, vol. 42, no. 6, pp. 1698-1709, Nov. 1996.

[11] P. Jung, J. Plechinger, M. Doetsch, and F. M. Berens, “Advances on the application of turbo-codes to data services in third generation mobile networks,” *Proc. Int. Symp. on Turbo Codes & Related Topics*, Brest, France, pp. 135-142, Sept. 1997.

[12] D. Rowitch, and L. B. Milstein, “Rate compatible punctured turbo (RCPT) codes in a hybrid FEC/ARQ system,” *Proc. Commun. Theory Mini-Conference of Globecom'97*, Phoenix, AZ, pp. 55-59, Nov. 1997.

[13] D. Divsalar, and F. Pollara, “Turbo codes for PCS applications,” *1995 Proc. IEEE Int. Conf. Commun*., Seattle WA, pp. 54-59, June 1995.

[14] S. Benedetto, D. Divsalar, G, Montorsi, and F. Pollara, “A soft-input soft-output maximum a posteriori (MAP) module to decode parallel and serial concatenated codes,” *TDA Progress Report 42-127*, Nov. 15, 1996

[15] A. Viterbi, “An intuitive justification and simplified implementation of the MAP decoder for convolutional codes,” *IEEE Journal on Select. Areas Commun*., vol. 16, no. 2, pp. 260-264, Feb. 1998.

[16] S. S. Pietrobon, "Implementation and performance of a turbo/MAP decoder," *Int. Journal of Sat. Commun.*, To appear.
Available at http://charli.Levels.UniSA.Edu.Au:80/~steven/turbo/turboMAP.ps.gz

[17] J.H. Lodge and M. Gertsman, "Joint detection and decoding by turbo processing for fading channel communications," *Proc. Int. Symp. on Turbo Codes & Related Topics*, Brest, France, pp. 88-95, Sept. 1997.

[18] D. Divsalar and F. Pollara, "Serial and hybrid concatenated codes with applications," *Proc. Int. Symp. on Turbo Codes & Related Topics* , Brest, France, Sept. 1997.

[19] F. Burkert and J. Hagenauer, "A serial concatenated coding scheme with iterative 'turbo' and feedback decoding," *Proc. Int. Symp. on Turbo Codes & Related Topics*, Brest, France, pp. 227-30, Sept. 1997.