International Journal of the Physical Sciences Vol. 6(10), pp. 2338-2347, 18 May, 2011
Available online at http://www.academicjournals.org/IJPS
DOI: 10.5897/IJPS11.424
ISSN 1992 - 1950 ©2011 Academic Journals

Full Length Research Paper

An FPGA realization of simplified turbo decoder architecture

Shivani Verma[1]* and Kumar S.[2]

[1]Department of Electronics and Communication Engineering, ASET, Bijwasan, New Delhi, India.
[2]Department of Electronics and Communication Engineering, Thapar University, Patiala, Punjab, India.

Accepted 07 April, 2011

The key issue of applying Turbo codes is to find an efficient implementation of turbo decoder. This paper addresses the implementation of a simplified and efficient turbo decoder in field programmable gate array (FPGA) technology. A simplified and efficient implementation of a Turbo decoder with minor performance loss has been proposed. An integer Turbo decoder based on the standard 2's complement number system after considering the issues of dynamic range, truncation effect and other algorithm related subjects has been introduced. The efficient implementation comes from algorithm modification, integer arithmetic and compact hardware management. Based on the Max-Log-MAP decoding algorithm, the branch metric is modified by weighting a priori value, resulting in a significant BER improvement. The Turbo decoder takes in 8-level integer inputs generates 7-bit soft-decisions and calculates all metrics on integers, avoiding complex floating point or fixed-point arithmetic. By manipulating memory address, delay associated with interleaving and de-interleaving is eliminated, resulting in much higher throughput. Also, by taking advantage of identical decoder function, Turbo decoder is implemented in a single-decoder structure, making efficient use of memory and logic cells.

Key words: Turbo, Max-Log-MAP, field programmable gate array, bit error ratio.

INTRODUCTION

Turbo codes are error-correcting codes based on parallel concatenation of convolutional codes and can achieve near channel capacity performance with a long block size and many decoding iterations. Due to its outstanding performance, Turbo coding has extensive applications in error prone environments such as wireless personal communications and deep-space communications. The key issue of applying Turbo codes is to find an efficient implementation of Turbo decoder. This requires simplification of complex Turbo decoding algorithms and compact hardware management. This paper addresses the implementation of a simplified and efficient Turbo decoder in field programmable gate array (FPGA) technology. FPGAs offer attractive advantages over other implementations of application specific integrated circuits (ASICs). Specifically, they are easily configurable, reducing significantly development time and non-recurring engineering costs. A design can be easily conceptualized, tailored to a specific application, then implemented and tested at low cost and minimal risk. That is, implementing a specified functionality using an FPGA, while simultaneously trying to optimize the architecture across the dimensions of silicon area, system throughput and power consumption. The low unit cost for the smaller and slower devices is also attractive. Consequently, for high performance applications, special attention must be devoted to architectural and algorithmic issues, if designs well matched to the capabilities of FPGAs are to be realized. For estimating the states or outputs of a Markov process observed in the white noise, symbol-by-symbol MAP algorithm is optimal. However, this algorithm, even in its recursive form, poses technical difficulties because numerical representation of probabilities, non-linear functions and because of mixed multiplications and additions of these values.

The Log-MAP algorithm, which is the MAP decoding algorithm in logarithmic domain, is optimum for estimating the outputs of a Markov process. However, the optimal Turbo decoding algorithm (Choi et al., 2006) is

*Corresponding author. E-mail: shivani.pasricha@gmail.com. Tel: +91-9990940202.

computationally intensive due to the function max*(x, y) = ln ($e^x + e^y$). The max*(x, y) function can be expressed as:

$$\max^*(x, y) = \max(x, y) + \ln(1+ e^{-|x-y|})$$

that is, $\max^*(x, y) = \max(x, y) + f_{MAP}(|x-y|)$

The max* operation is equivalent to finding the maximum of the two inputs and then adding a "correction term $f_{MAP}(|x-y|)$. A straightforward implementation of the Log-MAP is to store the 8 to 16 values of the correction term in a look-up table. Another approach is to set correction term to zero to obtain an extremely simple implementation at the expense of around 0.5 dB loss in coding gain. However, both the approaches are not well suited for the emerging global third generation Wideband CDMA wireless systems. First, with the intrinsic interference-limited nature of CDMA, the system capacity is directly linked to the operating SNR for a particular quality of service requirement. For certain setups, the loss of 0.5 dB coding gain can amount to 10% loss in capacity. Hence, it is preferable to use more accurate implementations for performance improvement. Secondly, to support enough decoding iterations for the highest data rate (1 Mbps), the look-up table approach could turn out to be cumbersome as multiple look-up tables are required for a wide range of signal-to-noise (SNRs), and in turn will also increase the hardware cost. The correction term is approximated (Yuan and Ye, 2008) by linear function and the performance of the linear-MAP was shown to be close to the optimal solution. In our implementation, a modified Max-Log-MAP decoding algorithm is proposed to effectively close the performance gap between Max-Log-MAP. This approach is simpler than the linear-MAP algorithm. Another set of sub-optimal decoding algorithms is based on the soft-output viterbi algorithm (SOVA). It was shown (Boutillon et al., 2007) that the best performance that the SOVA-based algorithm can achieve is the same as that of Max-Log-MAP (Lingyan et al., 2006).

Computational complexity is estimated for the most popular Turbo decoding algorithms: Log-MAP, Max-Log-MAP and SOVA (Fagoonee and Honary, 2004). Complexity studies showed that Max-Log-MAP is the best compromise between performance and complexity. Therefore, our implementation will be based on the Max-Log-MAP algorithm. Regarding Turbo decoder implementations, several interesting implementations were recently proposed, most of which are based on fixed-point arithmetic. Since fixed-point operations require multiplications and divisions for normalization, computational complexity is still high. In this paper, we propose an integer Turbo decoder based on the standard 2's complement number system after considering the issues of dynamic range, truncation effect and other algorithm related subjects. The turbo decoder considered in this paper has the following specifications:

i) Code rate R = 1/3.
ii) Generator polynomial: $g = (13, 15)_{oct}$.
iii) Puncturing pattern: even/odd parity.
iv) Block size: N = 40 to 5114 bits.
v) Interleaver: S-random interleaver with S = 18.
vi) Trellis termination: none.
vii) Considered modulation: BPSK'.

TURBO CODES STRUCTURE

Turbo encoders consist of two recursive systematic convolutional (RSC) encoders and a random interleaver between them as shown in Figure 1a. The conventional Turbo decoders contain two SISO (soft input soft output) decoders, which are associated with the two RSC encoders, and an interleaver and a de-interleaver between those two decoders as depicted in Figure 1b. The SISO decoders generate soft outputs, which represent how reliable the outputs are. In the MAP-based algorithm, the decoding process consists of two steps forward recursion and backward recursion. During a forward recursion, for each trellis transition of the branch metrics are calculated and stored and the forward node metrics are updated. After receiving the whole block of noisy codewords, the decoder starts off backward recursion to generate soft-decisions. In the following, we summarize the MAP, Max-Log-MAP decoding algorithms and then present our modification.

MAP algorithm

Let $u = (u_1, u_2, \ldots, u_N)$ be the binary random variables representing information bits. In the systematic encoders, one of the outputs $x_s = (x^s_1, x^s_2, \ldots, x^s_N)$ is identical to the information sequence u. The other is the parity information sequence output $x_p = (x^p_1, x^p_2, \ldots, x^p_N)$. We assume BPSK modulation and an AWGN channel with noise spectrum density N_o. The noisy versions of the outputs is $y_s = (y^s_1, y^s_2, \ldots, y^s_N)$ and $y_p = (y^p_1, y^p_2, \ldots, y^p_N)$, and $y = (y_s, y_p)$ is used for simplicity. In the MAP decoder, the decoder decides whether $u_k = +1$ or $u_k = -1$ depending on the sign of the following log-likelihood ratio (LLR):

$$L_R(u_k) = \log \frac{P(u_k = +1 \mid y)}{P(u_k = -1 \mid y)} \quad (1)$$

Let S_k denote the state of the encoder at time k. It can take values from 0 to 2M-1 where M is the number of memory elements in the encoder. LLR can be rewritten as:

$$L_R(u_k) = \log \frac{\sum_{s_k} \sum_{s_{k-1}} \gamma_1(y_k, S_{k-1}, S_k) \alpha_{k-1}(S_{k-1}) \beta_k(S_k)}{\sum_{s_k} \sum_{s_{k-1}} \gamma_0(y_k, S_{k-1}, S_k) \alpha_{k-1}(S_{k-1}) \beta_k(S_k)} \quad (2)$$

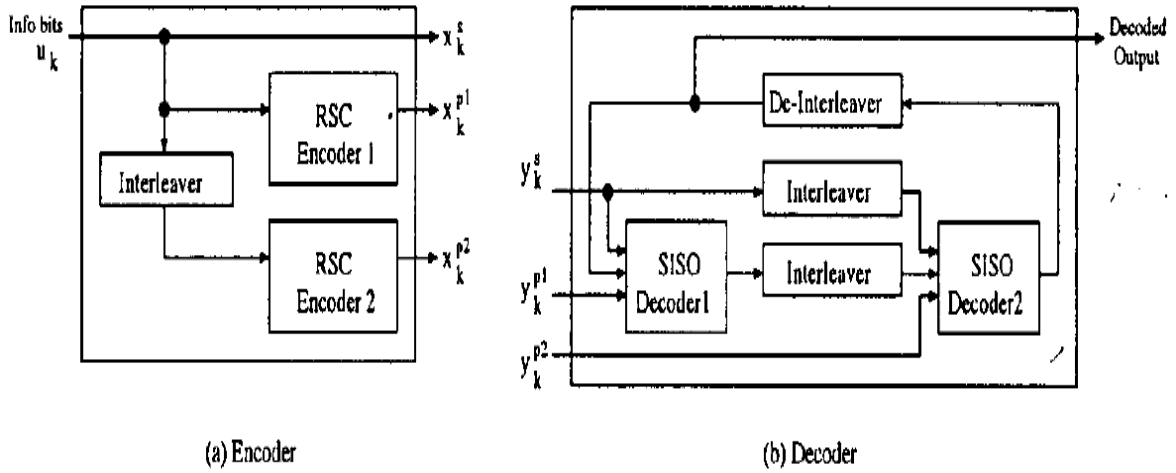

Figure 1. Turbo code structure.

Where α is the forward recursion metric, β is the backward recursion metric and γ_i is the branch metric. They are defined as:

$$\alpha_k(S_k) = \sum_{S_{k-1}} \sum_{i=0} \gamma_i(y_k, S_{k-1}, S_k).\alpha_{k-1}(S_{k-1}) \quad (3)$$

$$\beta_k(S_k) = \sum_{S_{k+1}} \sum_{i=0} \gamma_i(y_{k+1}, S_k, S_{k+1}).\beta_{k+1}(S_{k+1}) \quad (4)$$

$$\gamma_i(y^s_k, y^p_k), S_{k-1}, S_k) = q(u_k = i | S_k, S_{k-1}) p(y^s_k | u_k = i) p(y^p_k | u_k = i, S_k, S_{k-1}) P_r(S_k | S_{k-1}) \quad (5)$$

The parameter q (u_k = i/S_k, S_{k-1}) is either one or zero depending on whether u_k = i is possible for the transition from state S_{k-1} to S_k or not. Calculating p (y^s_k | u_k = i) and p (y^p_k| u_k = i, S_k, S_{k-1}) is trivial if the channel is AWGN. The last component P_r (S_k|S_{k-1}) usually has a fixed value for all k. However, this is not the case in the iterative decoding.

The 'a priori' probability of information bits generated by the other MAP decoder must be considered in turbo decoders.

Max-Log-MAP algorithm

In order to avoid the complexity of multiplications (Fowdur and Soyjaudah, 2009; Yuanfei et al., 2009) these equations can be converted to the additive form using the following logarithmic quantities:

$$\bar{\alpha}_k(S_k) = \log(\alpha_k(S_k)), \bar{\beta}_k(S_k) = \log(\beta_k(S_k)), \bar{\gamma}_k(S_k) = \log(\gamma_k(S_k)) \quad (6)$$

The forward recursion and the backward recursion are now represented in the additive form:

$$\bar{\alpha}_k(S_k) = \max^*_{(S_{k-1}, i)} (\bar{\gamma}_i(y_k, S_{k-1}, S_k) + \bar{\alpha}_{k-1}(S_{k-1})) \quad (7)$$

$$\bar{\beta}_k(S_k) = \max^*_{(S_{k+1}, i)} (\bar{\gamma}_i(y_{k+1}, S_k, S_{k+1}) + \bar{\beta}_{k+1}(Sk+1)) \quad (8)$$

Where max* is a maximization function with a correction term:

$$\max^*_i A_i = A_M + \log(1 + \sum_{i \neq M} \exp(A_i - A_M)) \quad (9)$$

$$A_M = \max_i A_i \quad (10)$$

The corrective term can be implemented using a small look-up table. The branch metrics are calculated as:

$$\bar{\gamma}_k(y_k, S_{k-1}, S_k) = \frac{1}{2}[L^e_{in}(u_k) u_k + L_c y^s_k u_k + L_c y^p_k x^p_k] \quad (11)$$

Where $L^e_{in}(u_k)$ is the a-priori information calculated by the other decoder and $L_c = \frac{4E_c}{N_o}$.

As mentioned earlier, an AWGN channel is assumed and N_o is noise spectral density and E_c is the energy per coded bit. The signal noise ratio $\frac{E_c}{N_o}$ has to be estimated to calculate the branch metrics. Using Equation 9, the LLR is represented as:

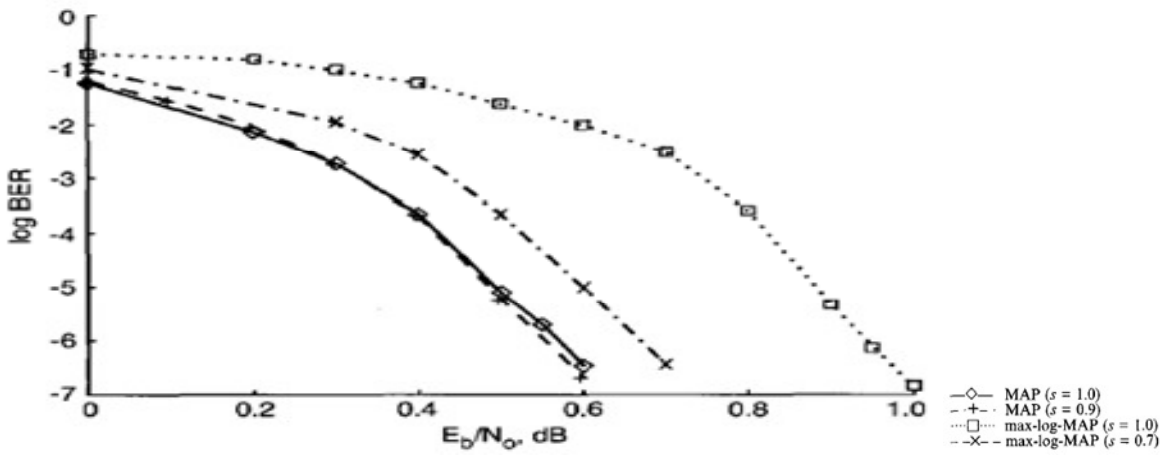

Figure 2. Performance of Turbo code with different scaling factors and block length 5114 bits.

$$L_R(u_k) = \max^*_{(S_k,S_{k-1})} (\bar{\gamma}_1(y_k, S_{k-1}, S_k) + \bar{\alpha}_{k-1}(S_{k-1}) + \beta_k(S_k)) - \max^*_{(S_k,S_{k-1})} (\bar{\gamma}_0(y_k, S_{k-1}, S_k) + \bar{\alpha}_{k-1}(S_{k-1}) + \bar{\beta}_k(S_k)) \quad (12)$$

In the iterative decoding, LLR is divided into three terms:

$$L_R(u_k) = L_c y_k^s + L_{in}^e(u_k) + L_{out}^e(u_k) \quad (13)$$

The last term is called “extrinsic information” and only this term should be fed back to the input of the other decoder as a-priori information. Therefore, $L_c y_k^s + L_{in}^e(u_k)$ must be subtracted from $L_R(u_k)$ before it is fed back to the other decoder. $L_R(u_k)$ and $L_{out}^e(u_k)$ are used to terminate the iteration of the Turbo decoding. Therefore, $L_{out}^e(u_k)$ can be expressed as:

$$L_{out}^e(u_k) = \frac{L(u_k)}{2} - (y_k^s + L_{in}^e(u_k)) \quad (14)$$

As earlier said, the term $L_{out}^e(u_k)$ is the information exchanged between the constituent decoders. In our implementation, we apply the Max-Log-MAP algorithm with a modification of the branch metrics. We weigh a-priori values with a scaling factor s, resulting in the following:

$$L_{out}^e(u_k) = \left[\frac{L(u_k)}{2} - (y_k^s + L_{in}^e(u_k)) \right] s \quad (15)$$

The effect of scaling factor on the BER performance was studied and at least 1000 errors were collected. Figure 2 shows the BER performance of the best evaluated scaling factor compared to the standard Max-Log-MAP decoding algorithm for block length 5114 with s = 1.0 and AWGN and s = 0.7 gives the best BER performance for our Turbo Code. We find that a properly selected scaling factor can improve the performance of Max-Log-MAP by 0.3 dB, giving a near optimal (Log-MAP) BER performance. The BER improvement is due to the fact that the scaling factor s can effectively mitigate error propagation through iterations. The scaling factor s allows us the variation in the information exchanged between the decoders. In qualitative terms, a large value of s makes the previous decoder outcome dominate the current decoding results, whereas a small value of s makes one decoder less dependent on the other decoder’s result.

TURBO DECODER ARCHITECTURE

The implementation architecture for a Turbo decoder can take a serial or a parallel approach. In the serial approach, a pair of decoders is used repetitively and the data input is processed at the higher speed (denoted as L bps in Figure 3) than the speed of incoming received bits (denoted as K bps). On the other hand, the parallel form of Turbo decoder would require multiple pair of MAP decoders and huge amount of memory for interleavers, de-interleavers and received data buffers. Unless the very high-speed decoder is needed, the serial approach would be practical.

MAP decoder architecture

Now the implementation has been reduced to a series of decoder operations, the obvious remaining drawback of the Map algorithm is the excessive memory required. The entire history of the state metrics must be stored out to the end of the trellis, at which point the backward

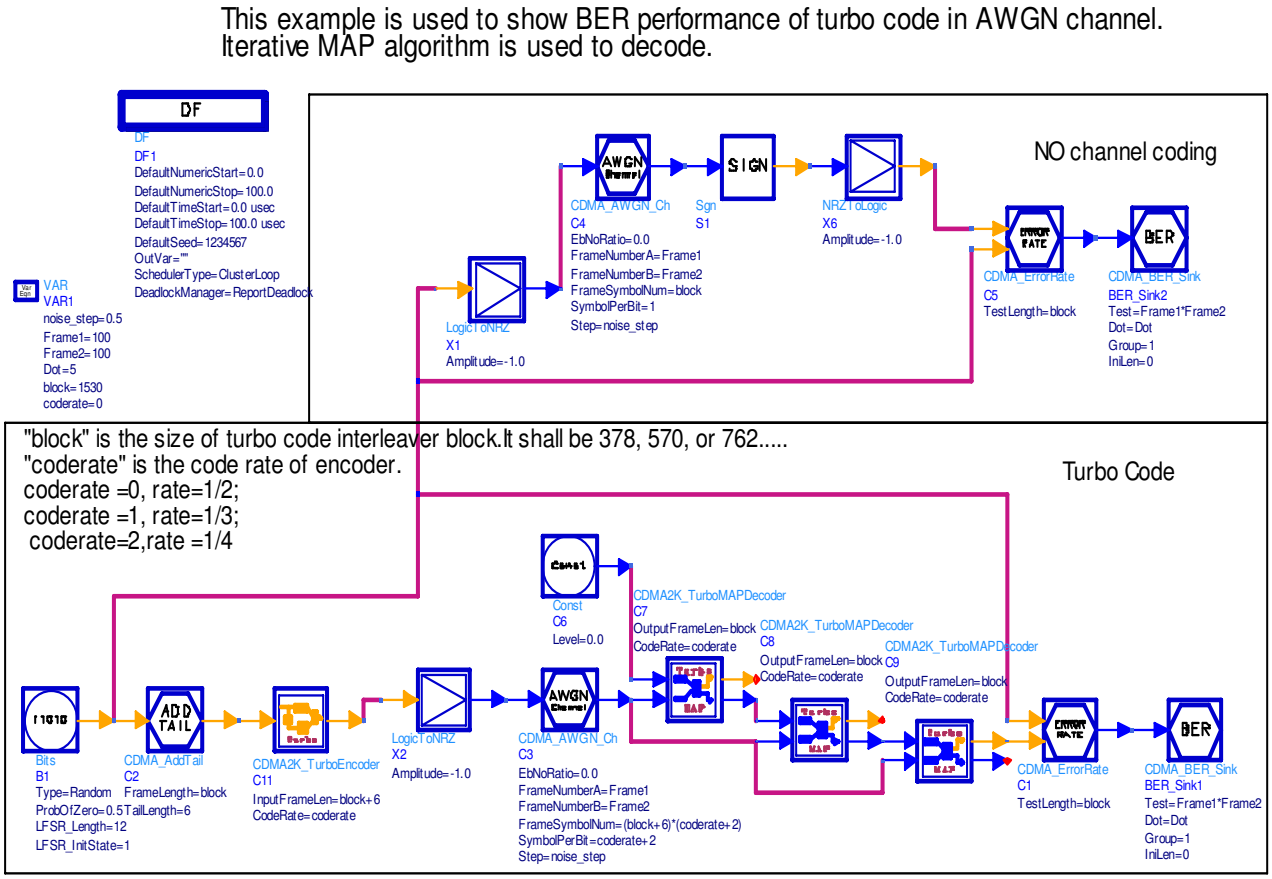

Figure 3. Simulation set-up for turbo codes.

algorithm begins and decisions can be output starting with the last branch, without the need to store any but the last set of state metrics computed backward. This storage requirement is obviously very large; for an 8-state code, assuming 7-bit state metrics, it would require 56 bits of storage per branch, for a total of 56,000 bits for a 1000-bit block, which in any case is the minimal for Turbo code performance. Our implementation of MAP decoder is based on a technique proposed in (Benedetto and Montorsi, 1996) which reduces the memory requirement for a 8-state code to just a few thousand bits, independent of the block length(Kim et al., 2000). The technique can best be described by referring to the timing diagram of Figure 4, which indicates the bit processing times for one forward processor and two backward processors operating in synchronism with the received branch symbols that is, computing one set of state metrics during each received branch time (bit time for a binary trellis). The basic idea behind this approach is that Viterbi Algorithm can start cold in any state at any time. After a few constraint lengths, the set of state metrics are as reliable as if the process had been started at the initial (or at the final) node. Let's say that this learning period for the trellis is L branches, which are 16 in case of 8-state code. This is equally true for forward as well as backward algorithm, and assumes that subtracting at every node an equal amount from each, normalizes all state metrics. Let the received branch symbols be delayed by 2 L branch times. Then the forward algorithm will start at branch time 2 L. And also, it will compute all the state metrics for each node every branch time and storing these in memory. The first backward processor starts at the same time, but processes backward from the 2 Lth node, setting every initial state metric to the same value, not storing anything until branch time 3 L, at which point it has built up reliable state metrics and it encounters the last of the first set of L forward computed metrics. (In Figure 4, the top line indicates the time index; the remaining lines are labeled according to the times at which the branches are processed. Also, unreliable metric branch computations are shown as dashed lines).

At this point the Lth branch soft decisions are output by performing the generalized dual-maxima process, and the backward processor proceeds until it reaches the initial node at time 4 L. Meanwhile, starting at time 3 L, the second backward processor begins processing with equal metrics at node 3 L, discarding all metrics until time 4 L, when it encounters the forward algorithm having

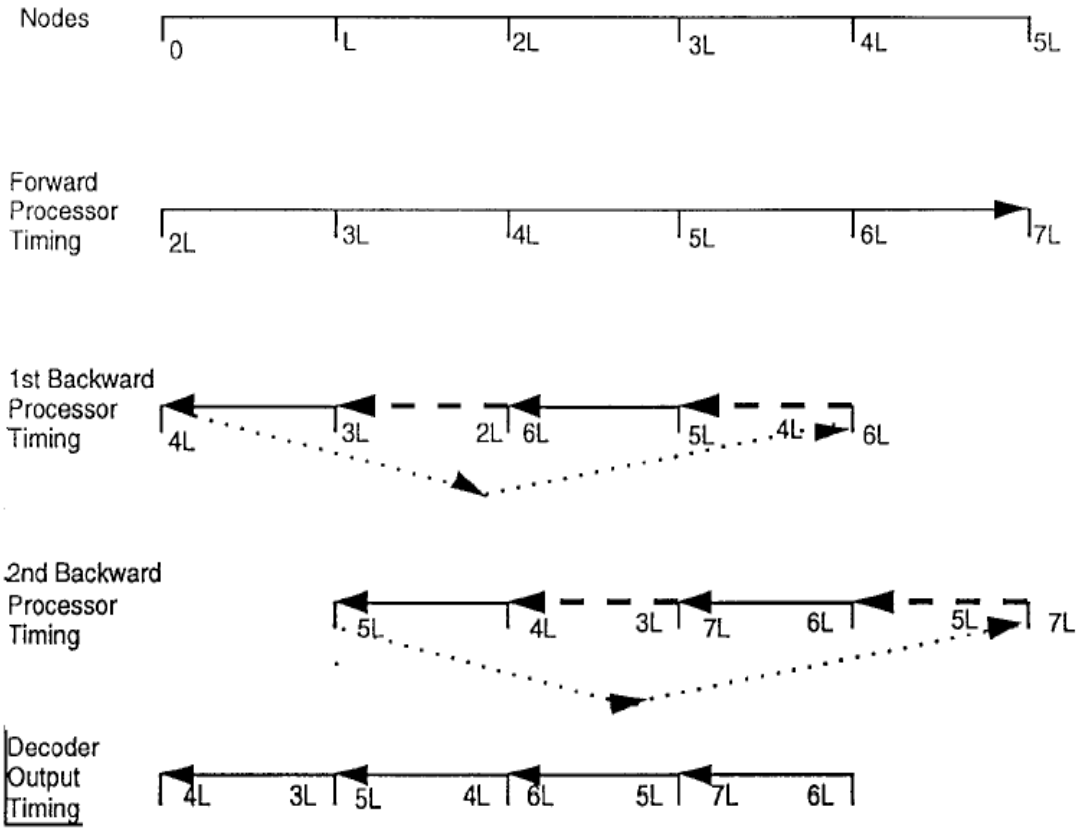

Figure 4. Timing diagram of forward and backward processors.

computed the state metrics for the 2 Lth node. The generalized dual-maxima process is then turned on until time 5 L, at which point all soft decision outputs from the 2 Lth to the Lth node will have been output. The two backward processors hop forward 4 L branches every time they have generated backward 2 L sets of state metrics, and they time-share the output processor since one generates the useful metrics, which are combined with those of the forward algorithm. For the backward algorithms, nothing needs to be stored except the metric set of the last node. The forward algorithm needs to store only 2 L sets of state metrics, since after its first 2 L computations (performed by time 4 L), its first set of metrics will be discarded, and the blank storage space can then be filled starting with the forward-computed metrics for the (2 L +1) th node (at branch time 4 L +1). Thus, the storage requirements for a 8-state code using 7-bit state metrics 112 L bits in all, which for L = 16 amounts to approximately 1792 bits. The forward recursion (α-unit) and the backward recursion unit (β-unit) are identical except for the direction of recursion.

The α-unit is shown in Figure 5. It should be noted that the state metrics keep on increasing as the recursion goes on. Therefore, we need to adopt some normalization scheme to avoid the explosion of the state metrics.

Numerical range

Most Turbo decoder implementations are based on fixed-point arithmetic (Fowdur and Soyjaudah, 2009). Therefore, in the implementation of Turbo decoder, a significant effort must be focused on dynamic range, number density and normalization before choosing a number system. Since, our design is a simplified structure of Turbo decoder (of course, without significant loss in BER performance), we chose standard 2's complement integer representation. For efficient implementation, it is required to estimate the numerical range of various metrics such that only a necessary number of bits would be used for each metric. In our implementation, assuming that the demodulator output produces 8-level (3-bit) output, the 3-bit value is converted into a 4-bit integer value ranging from –4 to +4 (without 0). With our modified Max-Log-MAP (s = 0.7), eight iterations and 3 dB SNR (E_b/N_o), simulations were performed to deduce the range of soft-decisions and the results indicate that they lie in the range of –48 to +48. In the same way, the extrinsic values range from –30 to +30 and branch metrics from –13 to +13. As a result, 6 bits are assigned for extrinsic values and soft-decisions, 5 bits for branch metrics and 8 bits for internal metrics.

Compared to fixed-point implementations, which

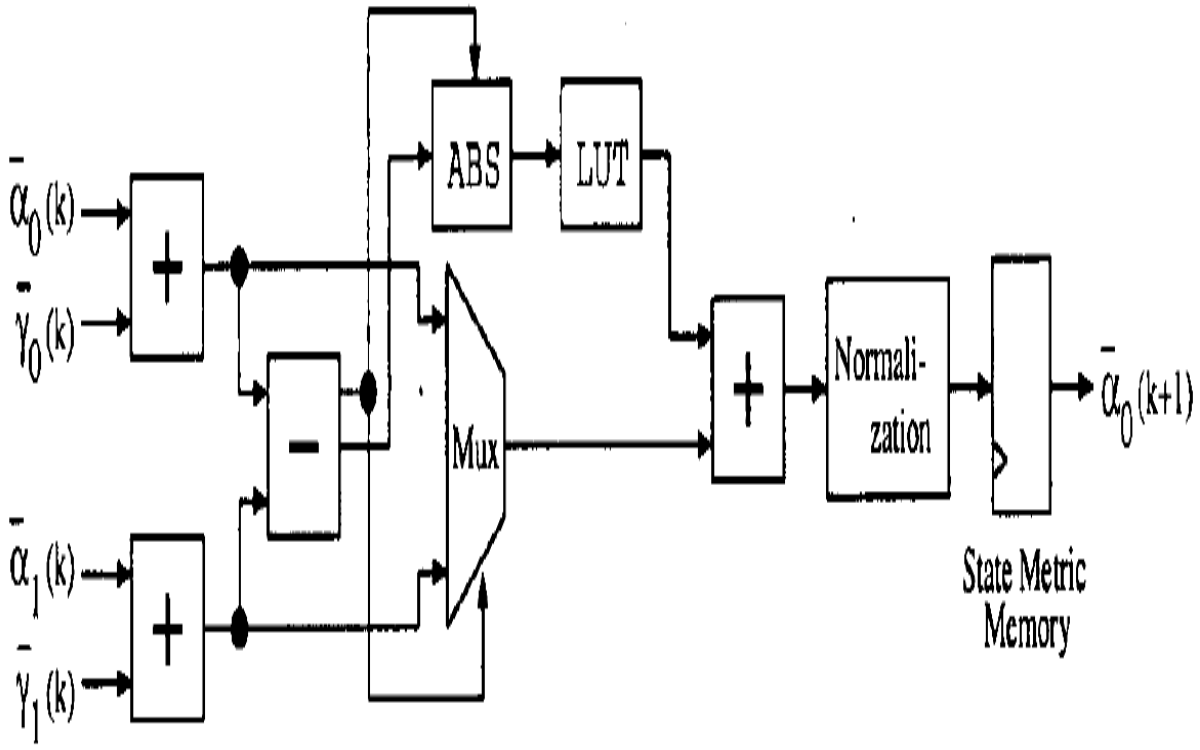

Figure 5. α-unit (forward recursion unit).

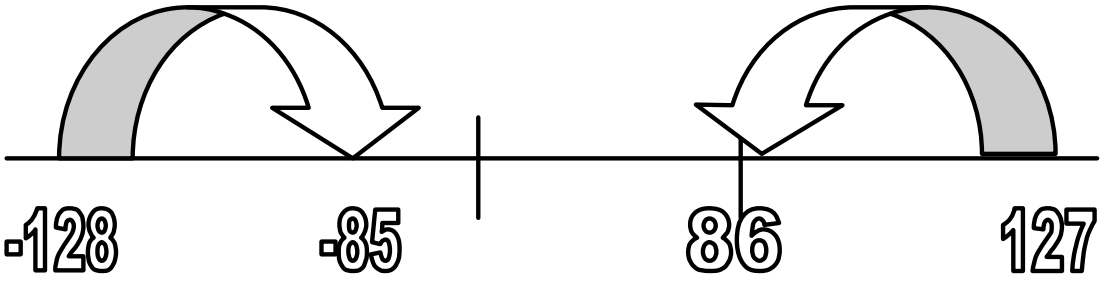

Figure 6. Forward/Backward node metric normalization.

require 8 bits for extrinsic values, 7 bits for demodulator outputs and at least 10 bits for internal metrics, it is clear to see the benefit of the integer based implementation.

Node metric normalization

As the recursion process progresses, forward/backward node metrics accumulate, this can easily overflow and may underflow during the first few updates. The overflow/underflow problem can be solved by metric normalization. A threshold value P is used, which is compared with all the node metrics for each decoded bit in both forward and backward recursion. If the absolute value of any node metric is greater than P, then all the node metrics are shifted towards the center as shown in Figure 6.

Note that the shifting is done for all node metrics so that the soft-decision values are not affected. The threshold value is chosen such that the update (the node metric plus a branch metric) does not cause overflow and the logic required for comparison is minimized. In our implementation, the threshold values are set to be –85 and 86. With normalization, node metrics can be represented using 8 bits.

Truncation effect

Truncation occurs in branch metric calculation (Benedetto

Table 1. Example of truncated results by different truncation method.

Z	-4	-3	-2	-1	0	1	2	3	4
Z/2	-2.0	-1.5	-1.0	-0.5	0	0.5	1.0	1.5	2.0
Round(Z/2)	-2	-2	-1	-1	0	1	1	2	2
Fix(Z/2)	-2	-1	-1	0	0	0	1	2	2
Floor(Z/2)	-2	-2	-1	-1	0	0	1	1	2
Ceil(Z/2)	-2	-1	-1	0	0	1	1	2	2

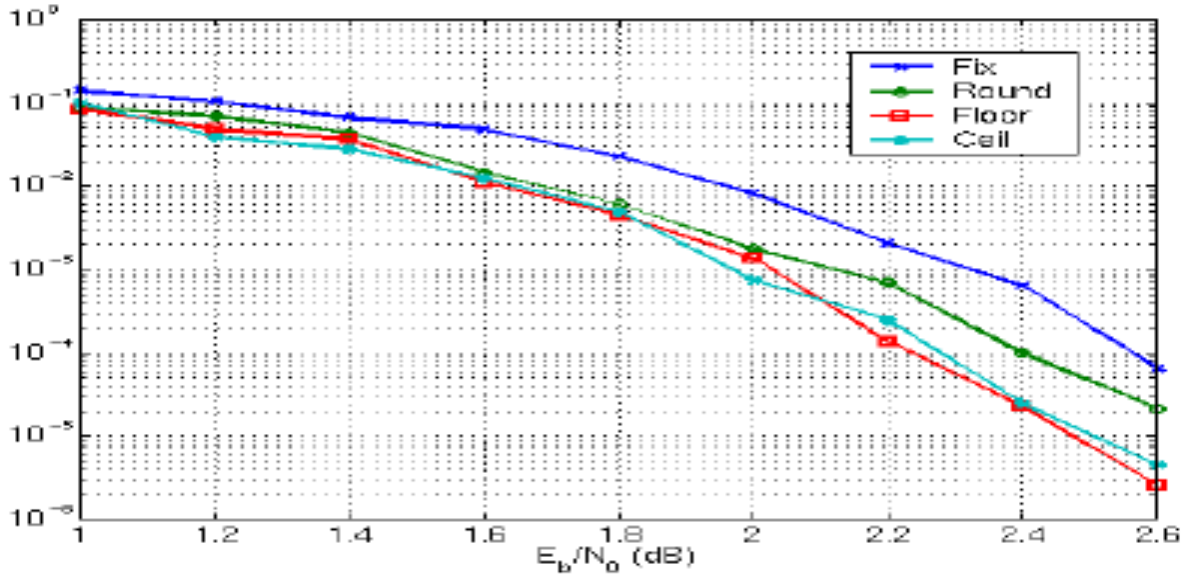

Figure 7. Truncation effect on BER.

and Montorsi, 1996) due to the division by 2 and the a-priori scaling factor s. Four truncation methods are considered round, fix, ceil and floor; each generate different truncation results as shown in Table 1. Although the round function is more systematic than other truncation functions, simulations in Figures 7 and 8 shows that the floor and ceil functions perform better than the other two methods. In our implementation, the floor function for truncation is used.

FPGA IMPLEMENTATION

Interleaver/De-interleaver

Turbo decoder requires an interleaver and a de-interleaver to perform proper permutation of systematic bits and extrinsic values. In a conventional turbo decoder design, there will be a separate functional block to carry out the interleaving/de-interleaving function, which will require a fixed amount of time to permute/un-permute its contents (Yuanfei et al., 2009). However, in our implementation, all the permutations are performed by address manipulation that requires no addition delay in the processing. Figure 9 shows the address generator that generates addresses for a decoder to read/write required information from a memory. Because the first decoder processes un-permuted information, systematic bits and extrinsic value are read sequentially from 0 to N-1. They are then processed and stored in the same address. For the second decoder, information is processed based on the permutation order; therefore, the sequential index is now used to retrieve a permutation index from ROM. This ROM based address that serves as the index is then used to retrieve the required information for processing. Once the information is processed, the result is again stored in the same index completing the interleaving/de-interleaving process.

All the information is updated and stored properly by manipulating the memory address for decoders; therefore each decoder is ready to decode immediately after the other decoder is finished, resulting no delay to process interleaving/de-interleaving.

The overall decoding process takes far less clock cycles to finish up compared to that with traditional interleaving/de-interleaving process.

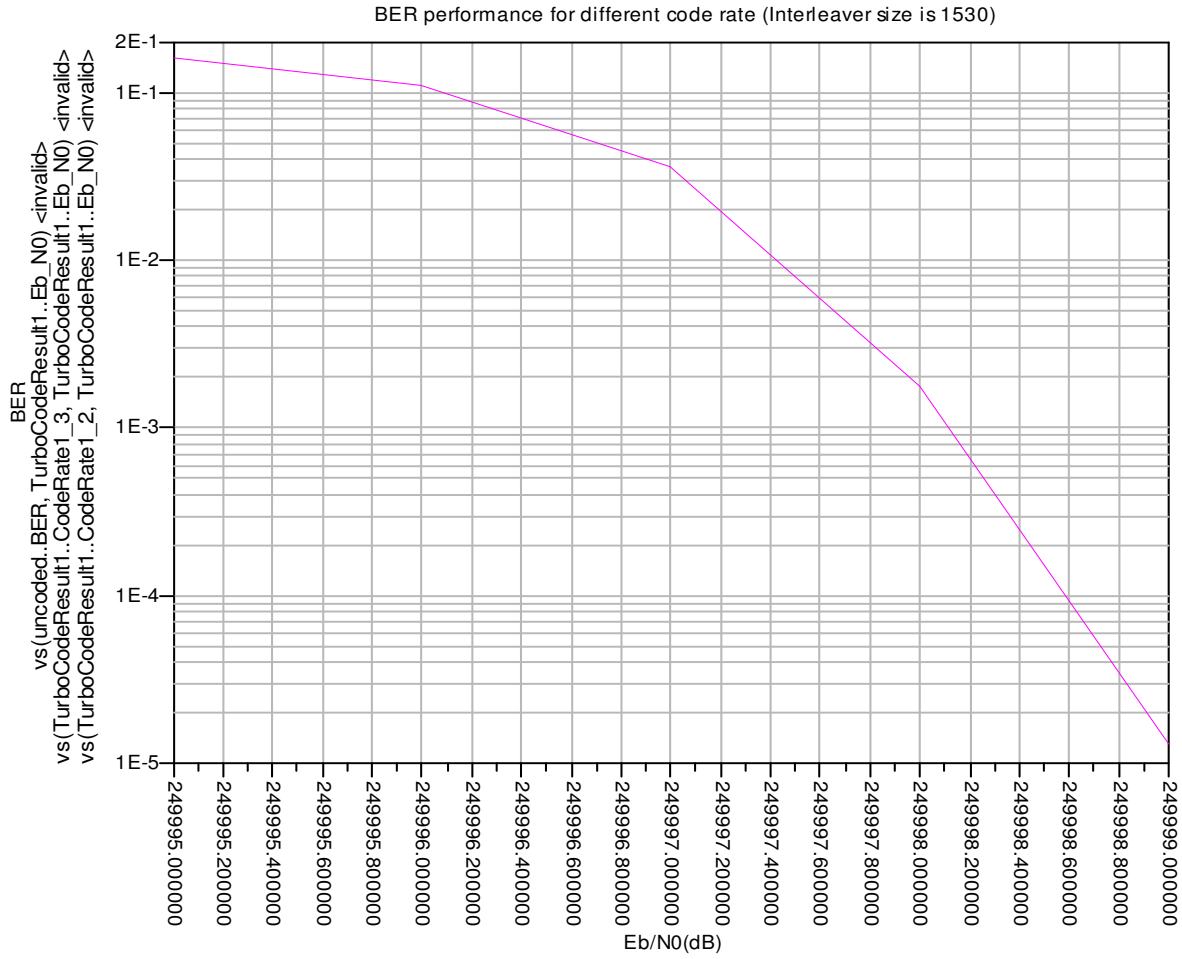

Figure 8. BER versus E_B/N_O.

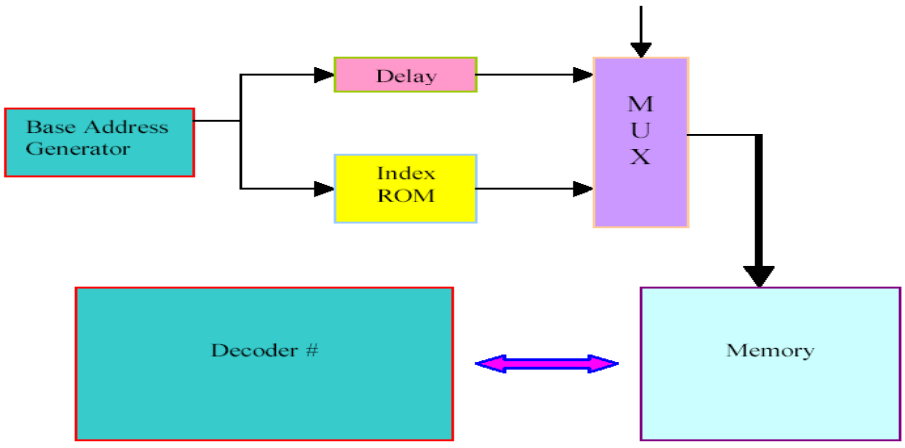

Figure 9. Address generator for interleaving and de-interleaving function.

Single decoder design

In our implementation, it is assumed that the same constituent code is used in the turbo encoder, and then the constituent decoders are identical. Therefore, the turbo decoder can be simplified by using a single decoder

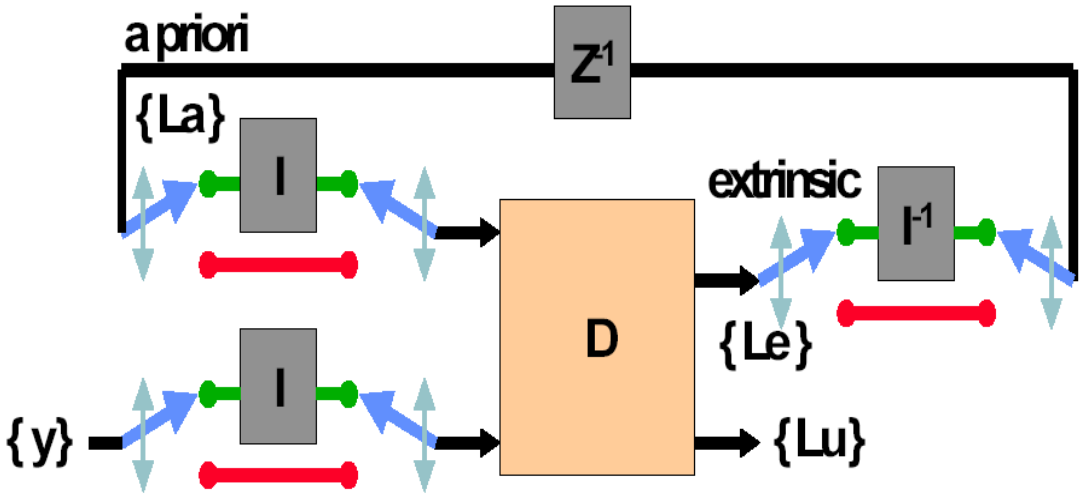

Figure 10. Single decoder design for Turbo decoder.

as shown in Figure 10. The switch is toggled according to the decoder number being operational while processing the data bits. When the first decoder is to be used, the switches are set to their lower position, which bypasses all interleaving and de-interleaving functions. When the first decoder has finished, the switches are then moved to the upper position to reconfigure the design as the second decoder that functions on permuted information. This process is repeated until the required number of iterations is completed.

RESULTS AND CONCLUSION

Turbo decoder was described in Verilog hardware description language and implemented on Xilinx Virtex series XCV300E FPGA chip. The design utilized almost 3447 out of 6912 logic cells and approximately 28 RAM blocks out of 48 total RAM blocks in the device. In addition, the design required no external components and consumes approximately 695 mW during normal operation. The FPGA-based turbo decoder with 8 iterations can operate more than 1 Mbps throughput at a clock rate of 25 MHz. In this paper, we have demonstrated a simplified and efficient implementation of a Turbo decoder with minor performance loss. The efficient implementation comes from algorithm modification, integer arithmetic and compact hardware management. Based on the Max-Log-MAP decoding algorithm, we modify the branch metric by weighting a priori value, resulting in a significant BER improvement. The Turbo decoder takes in 8-level integer inputs generates 7-bit soft-decisions and calculates all metrics on integers, avoiding complex floating point or fixed-point arithmetic. By manipulating memory address, delay associated with interleaving and de-interleaving is eliminated, resulting in much higher throughput. Also, by taking advantage of identical decoder function, we implemented our Turbo decoder in a single-decoder structure, making efficient use of memory and logic cells.

REFERENCES

Benedetto S, Montorsi G (1996). "Unveiling Turbo Codes: Some Results on Parallel Concatenated Coding Schemes." IEEE Trans. Inf.Theory, 42(2): 409-428.

Boutillon E, Douillard C, Montorsi G (2007). "Iterative Decoding of Concatenated Convolutional Codes: Implementation Issues", Proc. IEEE., 95(6): 1201-1227.

Choi DG, Jeong JH, Kim HM, Jung JW (2006). "High-Speed Adaptive Turbo Decoding Algorithm and Its Implementation." APCC'06. Asia-Pacific Conf. Commun., pp. 1–5.

Fagoonee L, Honary B (2004). "Application of turbo codes\l to tactical communications." "Application of turbo codes\l to tactical communications." Comput. Netw., 46(5): 741-749.

Fowdur P, Soyjaudah KMS (2009). "Joint source channel decoding and iterative symbol combining with turbo trellis-coded modulation." Sig. Process., 89(4): 570-582.

Kim DW, Taek WK, Jun RC, Jun JK (2000). "A modified two-step SOVA-based turbo decoder with a fixed scaling factor." Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. 2000 IEEE Int. Symp., 4: 37-40.

Lingyan S, Hongwei S, Keirn Z, Kumar BVKV (2006). "Field programmable gate array (FPGA) for iterative code evaluation." IEEE Trans. Magnet., 42(2): Part 1.

Yuan J, Ye W (2008). "A novel block turbo code\l for high-speed long-haul DWDM optical communication systems",Optik – Int. J. Light Electron Opt., In Press.

Yuanfei N, Jianhua G, Yong W (2009). "Iterative\l SNR estimation using a priori information", Digital Sig. Process., 19(2): 278-286.