RC22353 (W0202-073) February 28, 2002
Electrical Engineering

IBM Research Report

Towards an Energy-Efficient 3GPP Turbo Decoder Implementation --- Part I: Interleaver Architecture

Paul K. Ampadu, Stephen Kosonocky, Kevin Kornegay
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Towards an Energy-Efficient 3GPP Turbo Decoder Implementation — Part I: Interleaver Architecture

Paul Ampadu[1], Stephen Kosonocky, and Kevin Kornegay[1]

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

We present an energy-efficient approach for VLSI implementation of the Third Generation Partnership Project (3GPP) Turbo Coding interleaver algorithm. Unlike previous implementations, this interleaver does not employ a host processor, such as microprocessor, DSP, or microcontroller to program an address generator memory, but instead uses a dedicated hardware datapath to compute addresses on the fly, eliminating the overhead associated with instruction fetching and decoding. This further obviates the need for large instruction memory and ROM. Our architecture employs a distributed storage approach, leading to less power consumption per operation, since only relevant memory strips are activated as needed. Moreover, by separating the interleaver function itself from the data memory, our method makes it possible to reuse the same hardware interleaver in both encoder and decoder.

Index Terms — turbo codes, error control codes, interleavers, VLSI signal processing, 3G.

Introduction

It is well known that maximum hardware parallelism can reduce power consumption in VLSI systems substantially. This is because the system can be operated at the lowest frequency possible, allowing a reduction in supply voltage, leading subsequently to a reduction in overall power dissipation. Thus the subject of directly mapping algorithms to hardware has received considerable attention, particularly in the realm of wireless signal processing, where the need for portability places increased constraints on the power requirements of the system. Moreover, if properly done, direct mapping to hardware can reduce computation times, through parallelism, so that in the end, the overall energy per operation is improved.

Research in coding focuses on the need to transport information reliably and efficiently from one point to another, through a non-ideal (noisy) physical channel. Evolution towards hand-held multimedia and portable systems, and the need for these systems to communicate over the extremely hostile wireless channel pose important practical coding design and implementation challenges. Clearly, the performance of such systems is a result of complex design choices made at the algorithm, architecture, logic, circuit, and technology phases. Two major goals drive our direct-mapped approach to implementing turbo coding and decoding for third generation (3G) systems. First, these functions are both computationally intensive and power hungry, making the task of achieving an optimal balance among power, speed, and area difficult. Second, we seek to develop an efficient design methodology for related systems that minimizes the time-to-market from initial algorithm description phase to testable hardware.

[1] Cornell University, School of Electrical and Computer Engineering, Ithaca, NY 14850

3G systems based on the 3GPP air interface standard must support turbo coding at data rates of around 2Mbps, under efficient use of the available spectrum, while maintaining low mobile station power. An important design issue for such systems is the appropriate hardware/software partitioning. In most cases, a prudent separation is to use custom hardware accelerators for computation-intensive portions and a programmable host processor (such as, DSP or microprocessor) for low processing portions, as well as, control and sequencing operations. This approach also allows programming flexibility and relieves the host to perform its activities more efficiently. A hardware accelerator allows computationally intensive functionality, such as channel processing (e.g. Viterbi or Turbo), to be performed efficiently using dedicated hardware that may run in parallel with a host processor (requiring subroutine calls to hardware and proper interrupt handling), or run instead of the processor (with ISA or other enhancements to the host).

Turbo Codes

Turbo codes [1], also known as Parallel Concatenated Convolutional Codes, promise near Shannon-limit error-correction at reduced complexity, and as a result, have been touted as one of the greatest breakthroughs in communications since Shannon's seminal paper [2]. Since their inception in 1993, turbo codes have sparked great interest amongst coding theorists and researchers, as well as, the engineering design community; and this appeal continues to grow. Superior error-correction in the presence of excessive noise and interference will be a fundamental requirement in future generation communication systems, such as has already been seen in the 3G wireless standards supporting WCDMA (3GPP) [3] and CDMA2000 (3GPP2) [4]. Fig. A-1 in Appendix A depicts the 3GPP rate-1/3 turbo encoder structure. The encoder consists of a parallel concatenation of two identical rate-1/2 recursive systematic convolutional (RSC) encoders, separated by a pseudo-random interleaver. The outputs are multiplexed to form a single serial output for modulation and transmission. Fig. A-2 (Appendix A) is a generalized architecture for turbo decoding. The turbo decoder consists of soft-input soft-output (SISO) *maximum a posteriori* (MAP) component decoders that share information cooperatively to produce better estimates of the received symbols in the overall received sequence. The estimates, in the form of probabilities, are interleaved and deinterleaved between SISO computations.

Naturally, the significant error-rate improvements ascribed to turbo codes come at some computational cost. Whereas it is desired to reduce power consumption in mobile terminals through parallel VLSI implementation, the inclusion of channel encoders and decoders only worsens the problem, since these components typically constitute some of the most power-hungry blocks in a digital communication system. The presence of large interleavers impacts the delay performance and memory requirements of such a system tremendously. The computational intensity and the strict requirement of soft-input soft-output parameter passing between component decoders limit overall performance. Moreover, traditional parallel techniques, such as pipelining, cannot be efficiently exploited because of the iterative nature of the component decoders. To achieve an optimum balance among the usually opposing constraints of very low symbol error rate (SER), high throughput, low power consumption, small form factor, low cost and ease of adaptation requires painstaking effort at all levels of the optimization hierarchy — from efficient simplification of algorithms to proper exploitation of the targeted technology.

3GPP Turbo Code Interleaver

The error performance of turbo codes is determined by the distance spectrum of the codes, a parameter that is affected both by the encoder structure and the interleaver type and depth. In general, the deeper and more random an interleaver, the better the codebook minimum free distance, and hence, the better the error performance of the codes. “External” interleavers are customarily used to spread out burst errors to a fading channel (e.g. wireless channel). A non-uniform internal interleaver is employed in the 3GPP turbo coding specification, in order to randomize the data sequence from the encoder and achieve maximum entropy. This also ensures that the parity sequences generated by the two RSC encoders are as uncorrelated as possible, and that the minimum free distance of the code is as large as possible. Moreover, the presence of a pseudo-random internal interleaver allows one to create long block codes from small memory convolutional codes, as well as, permit the use of iterative suboptimal decoding algorithms in the turbo decoder.

The 3GPP turbo code internal interleaver allows any input bit sequence of length K, satisfying $40 = K = 5114$. If K falls outside this range, appropriate zero padding or frame truncation is employed. The algorithm proceeds as follows:

1. Row-wise bits input to a rectangular matrix with padding. Zero padding is necessary whenever the matrix cannot be filled with the data sequence completely, that is, $K < R*C$, where K is the length of the input sequence, R is the number of rows, and C is the number of columns of the rectangular matrix.
2. Intra-row permutation of the rectangular matrix, based on a base sequence constructed using the sequence length K as parameter.
3. Inter-row permutation of the rectangular matrix, based on a well-defined inter-row permutation pattern, using the sequence length K as parameter.
4. Column-wise bits output from rectangular matrix with pruning. Pruning is necessary whenever $K < R*C$, so that only valid data is read from the rectangular matrix.

The innovation of our method is the ability to combine steps 2 through 4 in a one-step equation that allows real-time address computation. Before describing the details of our architecture, we provide below a simple example to demonstrate the essence of the complex 3GPP interleaver algorithm. Appendix B contains the relevant 3GPP turbo coding formal specification, and Appendix C is a C-language program that implements the algorithm directly.

ILLUSTRATIVE IMPLEMENTATION OF 3GPP INTERLEAVER ALGORITHM:
The following example is based on an input sequence length K = 55. Please refer to Appendix B for a side-by-side interpretation of the following steps:
(1) Determine number of rows R, from K: Since $40 = K = 159$, $R = 5$.

(2) Find p from Prime_ROM (52-element, 9-bit integer lookup table) using R and K: $p = 11$. That is, choose first p satisfying $K \leq R*(p+1)$.
Prime_ROM

7	11	13	17	19	23	29	31	37	41	43	47	53
59	61	67	71	73	79	83	89	97	101	103	107	109
113	127	131	137	139	149	151	157	163	167	173	179	181
191	193	197	199	211	223	227	229	233	239	241	251	257

(3) Determine number of columns C, from p, R and K:
Since K = R*p, C = p = 11.

(4) Now, create R x C (5 x 11) sub-matrix from allocated Rect_Matrix (20x256), and order the resulting matrix appropriately with row-wise input addressing. Convention used throughout this paper (and in the 3GPP specification) is to start addressing from address zero (0).

Rect_Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…	…	…
0	0	1	2	3	4	5	6	7	8	9	10							
1	11	12	13	14	15	16	17	18	19	20	21							
2	22	23	24	25	26	27	28	29	30	31	32							
3	33	34	35	36	37	38	39	40	41	42	43							
4	44	45	46	47	48	49	50	51	52	53	54							
5																		
6																		
7																		
8																		
9																		
10																		
11																		
…																		
…																		
…																		

(5) Select the primitive root v, from Root_ROM, corresponding to the same address location as p chosen above, that is, v = 2. This step can be done in parallel with step (2) above.

Root_ROM

3	2	2	3	2	5	2	3	…		…		

(6) Construct the *base sequence (or s-memory)*, of size p-1 (size 10 in this case), for intra-row permutation. The default allocated memory for the s-memory is 256 8-bit words. Since this step uses only v and p, it can be generated as soon as they are known. The s-memory contents are computed as follows:
s [0] = 1
s [j] = (v*s [j-1])%p

s-memory (size 256, 8-bit words)

1	2	4	8	5	10	9	7	3	6	…			…

(7) Create the *least primes permuter sequence* q of size R (size 5 in this case). The default allocated memory is max R (20, 9-bit words). The q-memory contents are computed as follows:
q [0] = 1
q [j] = prime, such that gcd (prime, p-1) = 1

q memory

1	7	11	13	17	…								…

(8) Select the *inter-row permutation pattern T* from ROM, corresponding to the input K.

T = T_Vector_A (Stored in ROM)

4	3	2	1	0

(9) Obtain the *permuted least primes permuter sequence (or r-memory)*, given by
r [T [i]] = q [i]

r-memory (size 20, 9-bit words)

17	13	11	7	1	…					…

(10) Generate the intra-row permutation matrix U_ij as follows:

```
For each row 0 to (R-1)
        For columns 0 to (p-2)
                U_ij (column j) = s [(j*r [row no.]) % (p-1)]
End//
```

Since C = = p, fill column 10 (last column) with 0 for all rows. The contents of U_ij will be used to index into Rect_Matrix to create Temp_Matrix (our intra-row permuted matrix before inter-row permutations).

U_ij

	0	1	2	3	4	5	6	7	8	9	10	11	…	…	…
0	1	7	5	2	3	10	4	6	9	8	0				
1	1	8	9	6	4	10	3	2	5	7	0				
2	1	2	4	8	5	10	9	7	3	6	0				
3	1	7	5	2	3	10	4	6	9	8	0				
4	1	2	4	8	5	10	9	7	3	6	0				
5															
6															
7															
8															
9															
10															
11															
…															
…															
…															

(11) Perform the intra-row permutations as follows:

```
For Each Row 0 to (R-1)
    For columns 0 to (p-1)
        Set Temp_Matrix content = Rect_Matrix [U_ij content]
```

Temp_Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	…	…	…
0	1	7	5	2	3	10	4	6	9	8	0					
1	12	19	20	17	15	21	14	13	16	18	11					
2	23	24	26	30	27	32	31	29	25	28	22					
3	34	40	38	35	36	43	37	39	42	41	33					
4	45	46	48	52	49	54	53	51	47	50	44					
5																
6																
…																
…																
…																

(12) Perform the inter-row permutations, based on T, as follows:

```
For j=0 to (R-1)
    For i=0 to (C-1)
        Do Rect_Matrix [i] = Temp_Matrix [T [j]*C + i]
```

Rect_Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	…	…	…
0	45	46	48	52	49	54	53	51	47	50	44					
1	34	40	38	35	36	43	37	39	42	41	33					
2	23	24	26	30	27	32	31	29	25	28	22					
3	12	19	20	17	15	21	14	13	16	18	11					
4	1	7	5	2	3	10	4	6	9	8	0					
5																
…																
…																
…																

(13) Read (with pruning) column by column: The interleaved address sequence is then as obtained as follows:

45	34	23	12	1	46	40	24	19	7	48	38	26	20	5	52	35	30	17	2

49	36	27	15	3	54	43	32	21	10	53	37	31	14	4	51	39	29	13	6

47	42	25	16	9	50	41	28	18	8	44	33	22	11	0

//End Interleaving Procedure

While the mathematical aspects of non-uniform interleaving are non-trivial and remain a research challenge [5], our primary focus is to reinterpret and simplify the 3GPP-defined interleaver algorithm and map it efficiently to VLSI. Our choice of architecture is driven by a power-aware system design, minimal area, and the need to hide the long interleaver latency. Thus, we seek a method that allows computation of the interleaved addresses on the fly, with minimal complexity, and eliminates as much storage as possible. One of the first simplification mechanisms is a conceptual separation of the data sequence (bits or symbols) from the address mapping function. This simple separation of address from data allows us to find efficient ways to describe the interleaving function without regard to the data type, storage medium, or modulation/demodulation scheme (3GPP supports single-bit modulation, e.g. BPSK, as well as multi-bit modulation, e.g. QPSK).

Generally characterized by its storage and delay, an interleaver is a single-input single-output finite-state machine which receives symbols from a fixed alphabet $\mathbf{c} = (c_1, c_2, c_3 \ldots c_K)$ and outputs identical symbols in a different temporal order $\mathbf{c} = (c_1, c_2, c_3 \ldots c_K)$. Formally then, it is a bijection p: K $\mapsto$ K, such that $c_{p(i)} = c_i$. To compute the address mapping directly, we partition the algorithm into a preparatory phase (with minimal delay) and a real-time address generation phase [6] whose total delay is a function of the input sequence length K.

The preparatory phase consists of the following steps:

(1) Determine R, p, v, and C from the value of K.
(2) Construct the base sequence or s-memory sequence of size (p-1) recursively, using the primitive root v and prime p obtained above. Simultaneously, construct the least primes permuter sequence q, and use it to compute the r-memory (permuted least primes sequence).

Since the maximum value of p allowed is 257, the worst-case number of cycles required for step (2) above is 256 (p-1) cycles, for generating the s-memory. Aside from the significant reduction in memory (and corresponding power), this approach allows us to hide a portion of the interleaver delay, since this preparatory phase is easily performed while reading the bit (or symbol) sequence into the data memory. Moreover, multiple multiplexed frame lengths can be pipelined with this approach, so that the overall throughput is improved.

The real-time address generation phase comprises intra-row and inter-row permutations performed directly on each address location, as well as, column-wise reading with pruning, all computed in a single equation. As a result, the latency and computational requirement of this phase are a function of the particular input sequence length. The generalized real-time jth interleaved address is computed from the preparatory stage s-memory, r-memory, and T ROM using the relationship:

Interleaved_Address [j] = C * T [j%R] + s [((j/R) * r [T [j%R]])%(p-1)]... (1)

The "%" and "/" represent the modulo (remainder after division) and integer division operators respectively, and [] is an indexing operator.

To see why this is so, consider the R x C, 2-dimensional array M represented below. Without loss of generality, M has been drawn as a 5 x 11 array, following our previous illustrative example for K = 55. The contents of M represent the order in which the array is read, that is, beginning with address 0 and reading column-wise from top to bottom.

M	0	1	2	3	4	5	6	7	8	9	10
0	0	5	10	15	…				…	45	50
1	1	6	11	16						46	51
2	2	7	12	17						47	52
3	3	8	13	18						48	53
4	4	9	14	19	…				…	49	54

The column-wise reading of M (without pruning) of the j^{th} output address (beginning with $j = 0$) is computed as

$$\text{Output}\ [j] = M\ [(j\%R), (j/R)] \ldots (2)$$

where M [i, j] represents the address corresponding to row i, column j of the 2D array M. For example, the $j = 0^{th}$ address (corresponding to M [0,2]) is computed as follows:

$$\text{Output}\ [0] = M\ [(0\%5), (0/5)] = M\ [0, 0]$$

Similarly, the $j = 10^{th}$ address (corresponding to M [0,2]) is computed as follows:

$$\text{Output}\ [10] = M\ [(10\%5), (10/5)] = M\ [0, 2]$$

Working backwards,

$$M\ [i, j] = L\ [T\ [i], j] \ldots (3)$$

where L represents the 2D matrix after intra-row permutation, but before inter-row permutation based on the T [j] sequence stored in ROM.

Repeating this procedure,

$$L\ [i, j] = K\ [i, U\ [j, i]] \qquad \ldots (4)$$

where K is the array before intra-row permutations, given by

$$K\ [i, j] = \text{Input}\ [i*C + j] \qquad \ldots (5)$$

and U [j, i] is the intra-row permutation matrix (Recall that intra-row permutations are performed on each row separately), given by

$$U\ [j, i] = s\ [(j * r\ [i])\ \%\ (p-1)], \text{ for the general case}^{2}.$$

[2] This formulation represents the common overall equation. U_ij is defined differently for each separate case of C=p, C=p-1, and C=p+1. Refer to Appendix C for the detailed implementation.

Substituting equations (5), (4), and (3) in equation (2), we obtain,

Output [i] = K [T [i%R], U [(i/R), T [i%R]]]
= K [T [i%R], s [((i/R) * r [T [i%R]]) % (p-1)]], for the general case.
= Input[C * T [i%R] + s [((i/R) * r [T [i%R]]) % (p-1)]]

Appendix D contains the C-language code that implements our architecture using equation (1) to compute the interleaved addresses directly. The output of this program (Appendix D) has been compared with the output from the direct implementation (Appendix C) of the algorithm, for all valid input sequence lengths K. We have verified that both implementations yield the same address sequences for all valid K. Fig. 1 is a simplified block diagram of our real-time address interleaver architecture. Fig. A-4 shows the encoder structure based on this approach. Fig. A-3 shows our architecture for computing R, C, p, and v in the preparatory phase.

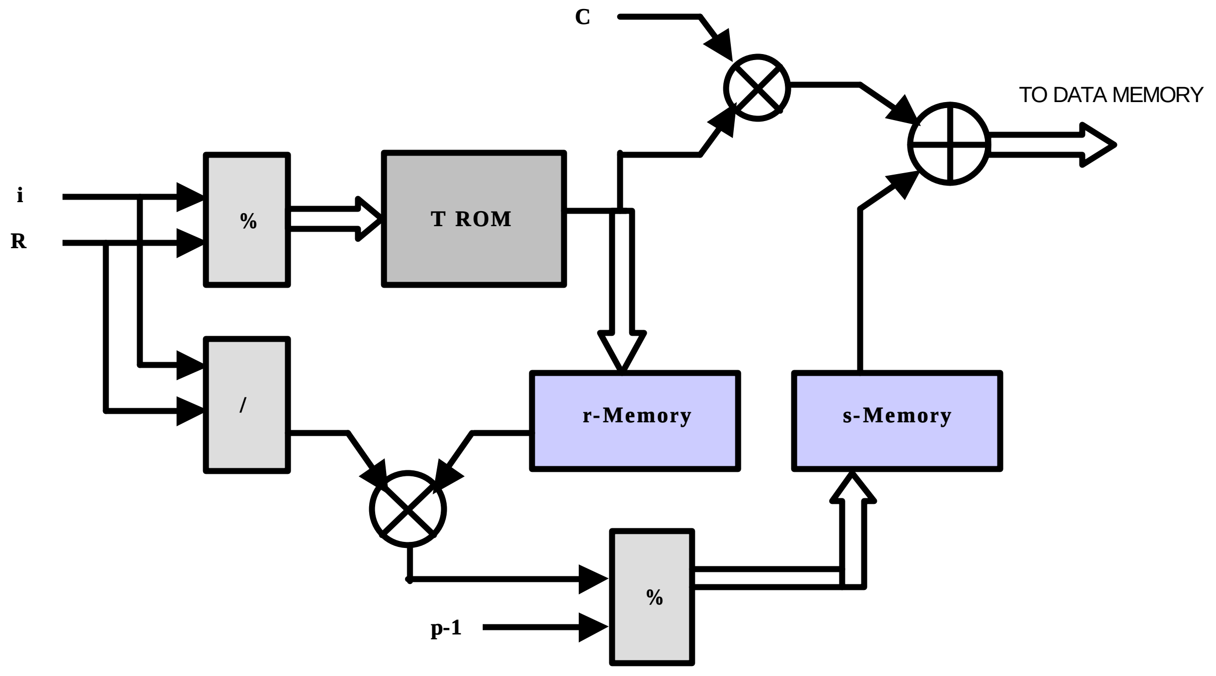

Fig. 1. Our 3GPP real-time address generator architecture

Results and Discussions

In certain coding experiments (where only a few frame lengths are allowed), it is possible to store the entire interleaved sequence for each of the possible frame lengths in a small lookup table. For the 3GPP interleaver, the range of allowable frame lengths is very large ($40 \leq K \leq 5114$), and the permutation patterns are different for dissimilar sequence lengths. In this case, such a brute force approach that stores the interleaved addresses for each of 5K possible input sequence lengths would require roughly 25MB (5K x 5K) ROM storage. Since it is inconceivable that one would build an interleaver using such an approach, we'll compare our estimated results with a more reasonable possible implementation. Consider the option of implementing the interleaver using a software programmable processor that programs an interleaver address memory, given a particular value of the input sequence length. This approach would require at least a 5K memory, aside from the data memory itself, ignoring the memory required to hold the programming instructions and the processor instruction set. It is thus fair to conclude that our approach eliminates at least a 5K-word memory, each of width 13-bits.

We will estimate the total energy expended per operation, area, and delay, using a recent low-power CMOS ASIC library 1-write, 1-read SRAM (5K x 13-bits) with the following specifications:

Dimensions	350μ x 460μ
Average Read/Write Capacitance, C	60pF
Cycle Time, T	2ns
Nominal Supply Voltage, V	1.2V

It is easy to see that the removal of this memory translates to a 0.16-mm^2 (350μ x 460μ) reduction in interleaver area. The total energy expended per complete interleaving operation for this SRAM can be estimated as follows:

$$E_{Total} = \Sigma \left(\tfrac{1}{2}CV^2\right)$$

where the sum is over the input frame length. Using the parameters in the table above, we see that the energy per write operation is 50pJ. In the worst case where 5K addresses are interleaved, the total energy consumed is 0.25μJ. We can assume that the total energy consumed in the host for computing the interleaved addresses is similar to that used in our on the fly computation approach. Thus, the overall energy savings per complete interleaving operation of the longest input sequence (5114) is about 0.5μJ (since an equivalent 0.25μJ of energy is consumed in reading all 5K addresses for indexing into a data memory).

In computing the total energy per operation, we have neglected the DC and leakage components usually associated with random access memories. We have also ignored the intermediate memory required by the software programmable approach to temporarily hold addresses between intra-row and inter-row permutations. Our approach, which accepts a slight penalty in computational complexity, allows us to use a minimal 256, 8-bit register array for the base sequence (s-memory), further reducing area and power. Furthermore, portions of the datapath not in use can be completely shut off, resulting in further reduction in power.

In terms of delay, a software processor based approach would require about 5000 cycles to program and load the interleaver addresses of a maximum length frame (5K symbols) into a single-write single-read port low-power memory. Aside from the load on the host bus, the delay overhead for interleaving would be approximately 10μs. Much of this latency cannot be hidden over the shared bus. In comparison, the total number of cycles to configure the single port 256-entry s-memory with similar access time is only about 0.5μs, a 20x delay improvement.

We have made our estimates based on worst-case assumptions. While the actual performance gains of our approach may be close to the estimated, it is possible that when the distribution of frame lengths is uniform, the actual performance penalty of the software programmable approach may not be that significant, compared to our approach. Furthermore, a software programmable approach allows one the flexibility of algorithm improvement and executable updates, a property that is lacking in our approach.

Conclusions

Turbo codes will continue not only to pervade future communication systems, but will also find immediate practical use in related systems. In addition to the value added in developing a turbo decoder in hardware, the lessons we learn from a thorough understanding of how turbo codes achieve their striking coding gains will help us improve other traditional error-control systems. Implementing the complex interleaver in hardware further allows us to address and overcome some of the difficulties associated with VLSI implementation of other related hardware accelerators. We acknowledge that the improved gain attributed to turbo codes comes at the expense of increased computation and slight complexity. Nevertheless, no coding scheme in existence today achieves the high coding gain of turbo codes with such relatively minimal complexity. This paper has outlined some of the associated challenges in implementing the 3GPP interleaver and described in detail some of the techniques used to overcome them. In particular, we have described architectural innovations that have allowed us to hide the long interleaver latency and reduce power consumption and area.

Future Work

The next phase of research will focus on the SISO component decoders. The Viterbi algorithm is a well-known optimal decoding algorithm that minimizes the probability of sequence error but fails to yield the maximum a posteriori probability (MAP) for each bit (or symbol). The optimal decoding algorithm for minimizing bit error probability is the BCJR algorithm [7]. The BCJR algorithm must be modified for recursive codes, however, and in addition, simplified to reduce its computational complexity for silicon implementation. Several modifications based on this notion have been proposed [8, 9, 10] for different implementation platforms. The proposed algorithms have been scattered in the literature and, in general, have not considered the impact of the complex 3GPP interleaver function on their performance. The next phase of research will examine, through both detailed study of the theory and extensive simulations in Matlab and C, the effect of the various decoding algorithms on speed, power, and SER performance, in the context of the 3GPP turbo code specification. A judicious choice of decoding algorithm that provides the best compromise among power, delay, and area, while maintaining the ultra-low error-rate performance will then be made. The selected algorithm will subsequently be mapped to VLSI hardware, following a procedure similar to the one outlined in this paper. We also plan to design and build the custom address generation datapath in VLSI.

Acknowledgments

The primary author wishes to thank the *Team Turbo* members (Dan Knebel, Kevin Warren, Stephen Kosonocky, and Uzi Shvadron[3]) and his friend and mentor, Jose Tierno, for their technical help. He is also indebted to his managers, Stephen Kosonocky and Kevin Warren for providing this excellent opportunity. He'd like to thank IBM for supporting his graduate study through a doctoral fellowship and multiple internships.

[3] IBM Research, Haifa, Israel.

References

1. C. Berrou, A. Glavieux, and P. Thitimajshima, “Near-Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes,” *Proc. 1993 IEEE Int. Comm. Conf.* (Geneva, Switzerland, May 1993), pp. 1064-1070.
2. C. E. Shannon, “A Mathematical Theory of Communication,” *Bell System Tech. Journal*, vol. 27, pp. 379-423, 1948.
3. 3rd Generation Partnership Project (3GPP) TSG-RAN: “Multiplexing and Channel Coding (FDD),” Release 4, Version 4.2.0, Sept. 2001. Document available at www.3gpp.org.
4. 3rd Generation Partnership Project 2 (3GPP2), “Physical Layer Standard for CDMA2000: Spread Spectrum Systems,” Release A, Version 5.0, July 2001. Available at www.3gpp2.org.
5. C. Berrou and A. Glavieux, “Near Optimum Error Correcting Coding and Decoding: Turbo Codes,” IEEE Transactions on Communications, vol. 44, no. 10, pp 1261-1271, Oct. 1996.
6. P. Ampadu, “3GPP Turbo Code Internal Interleaver,” Presentation at IBM T. J. Watson Summer Student Series, Aug. 2001.
7. L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, “Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate,” IEEE Transactions on Information Theory, pp 284-287, March 1974.
8. S. Benedetto et al., “A Soft-Input Soft-Output APP Module for Iterative Decoding of Concatenated Codes,” IEEE Communication Letters, vol. 1, no. 1, pp. 22-24, Jan. 1997.
9. S. Pietrobon, “Implementation and Performance of a Turbo/MAP Decoder,” International Journal of Satellite Communications, vol. 16, pp 23-46, 1998.
10. P. Robertson, E. Villebrun, and P. Hoeher, “A Comparison of Optimal and Suboptimal MAP Decoding Algorithms Operating in the Log Domain,” Proc. 1995 Int. Conf. On Comm., pp 1009-1013.

Appendix A: Simplified block diagrams

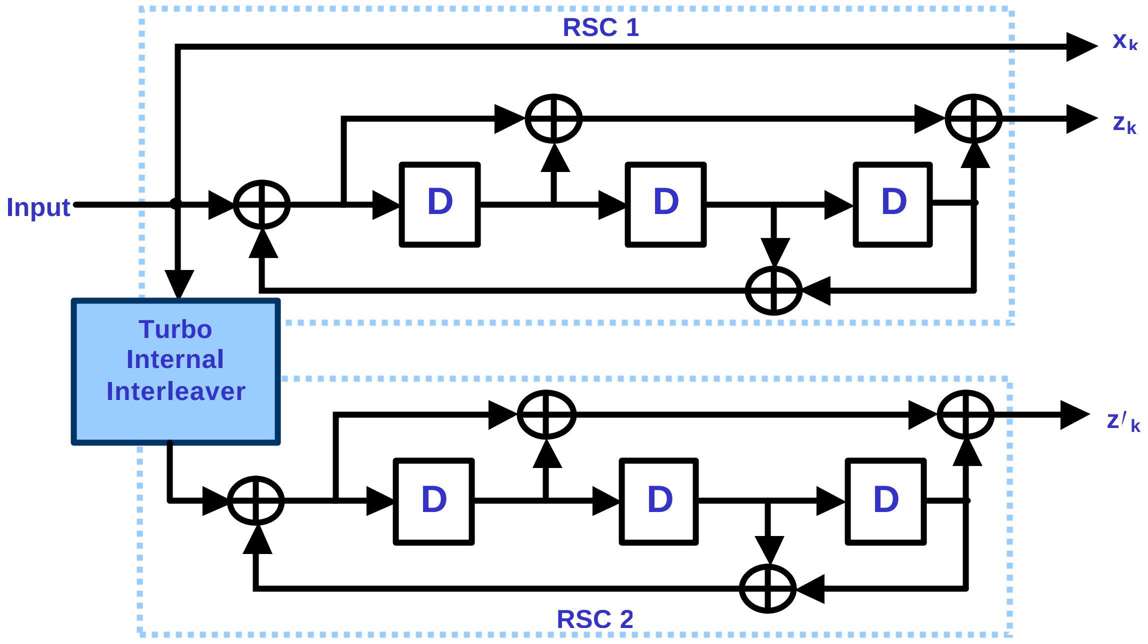

Fig. A-1. 3GPP specified turbo encoder structure

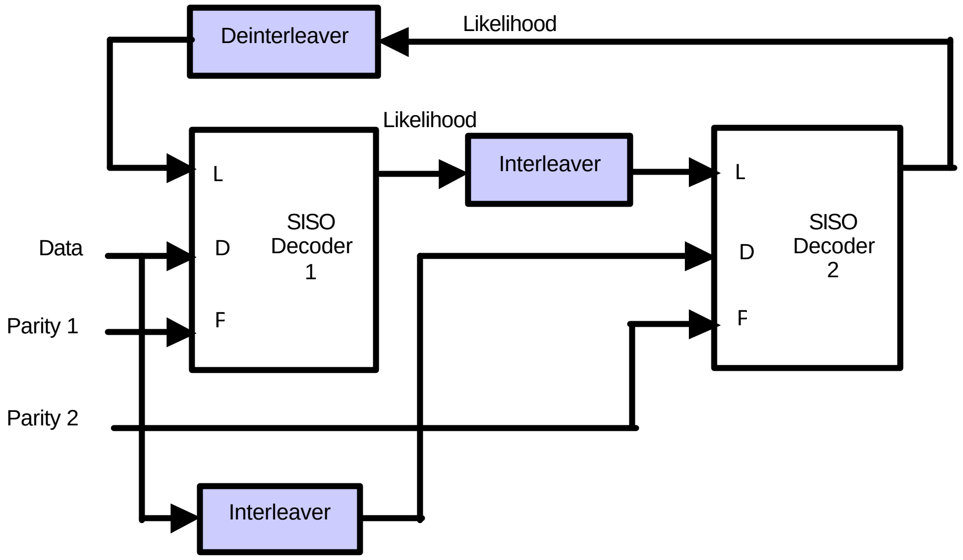

Fig. A-2. Generalized architecture for turbo decoding

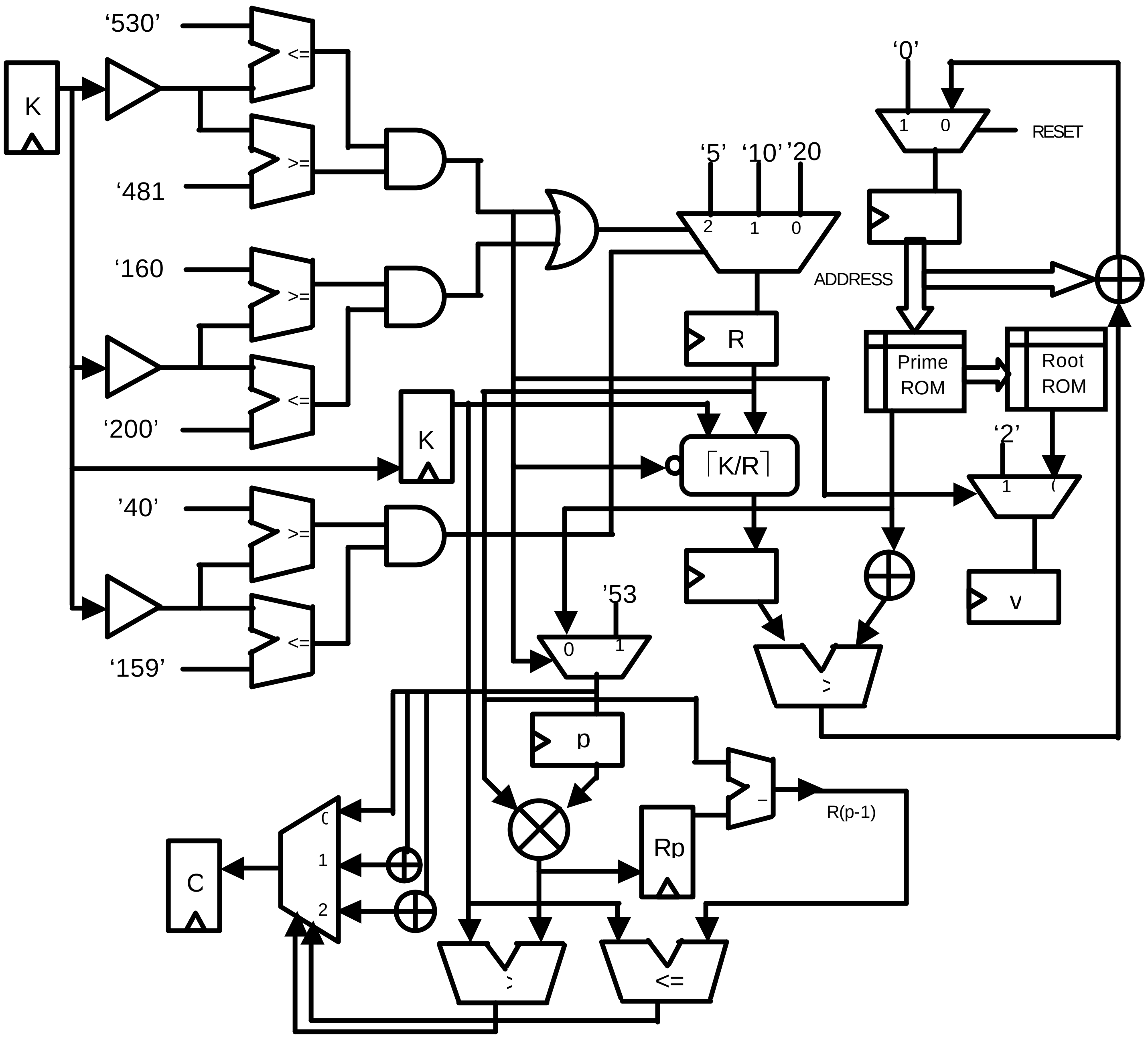

Fig. A-3. Preparatory phase architecture for R, C, p, v computation.

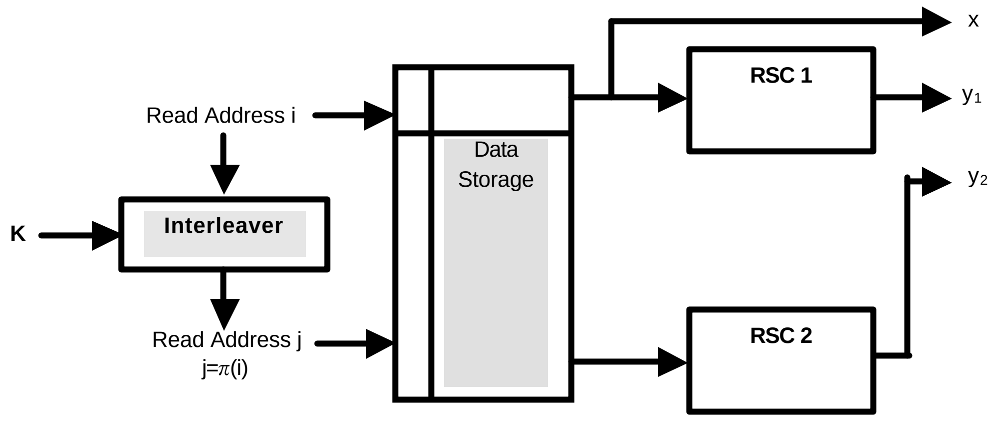

Fig. A-4. Modified 3GPP encoder structure using our proposed architecture

Appendix B: 3GPP Turbo Code Specification

3GPP TS 25.212 V4.2.0 (2001-09)

Technical Specification

3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Multiplexing and channel coding (FDD) (Release 4)

4.2.3.2 Turbo coding

4.2.3.2.1 Turbo coder

The scheme of Turbo coder is a Parallel Concatenated Convolutional Code (PCCC) with two 8-state constituent encoders and one Turbo code internal interleaver. The coding rate of Turbo coder is 1/3. The structure of Turbo coder is illustrated in figure 4.

The transfer function of the 8-state constituent code for PCCC is:

$$G(D) = \left[1, \frac{g_1(D)}{g_0(D)} \right],$$

where

$$g_0(D) = 1 + D^2 + D^3,$$

$$g_1(D) = 1 + D + D^3.$$

The initial value of the shift registers of the 8-state constituent encoders shall be all zeros when starting to encode the input bits.

Output from the Turbo coder is

$$x_1, z_1, z'_1, x_2, z_2, z'_2, \ldots, x_K, z_K, z'_K,$$

where $x_1, x_2, \ldots, x_K$ are the bits input to the Turbo coder i.e. both first 8-state constituent encoder and Turbo code internal interleaver, and K is the number of bits, and $z_1, z_2, \ldots, z_K$ and $z'_1, z'_2, \ldots, z'_K$ are the bits output from first and second 8-state constituent encoders, respectively.

The bits output from Turbo code internal interleaver are denoted by $x'_1, x'_2, \ldots, x'_K$, and these bits are to be input to the second 8-state constituent encoder.

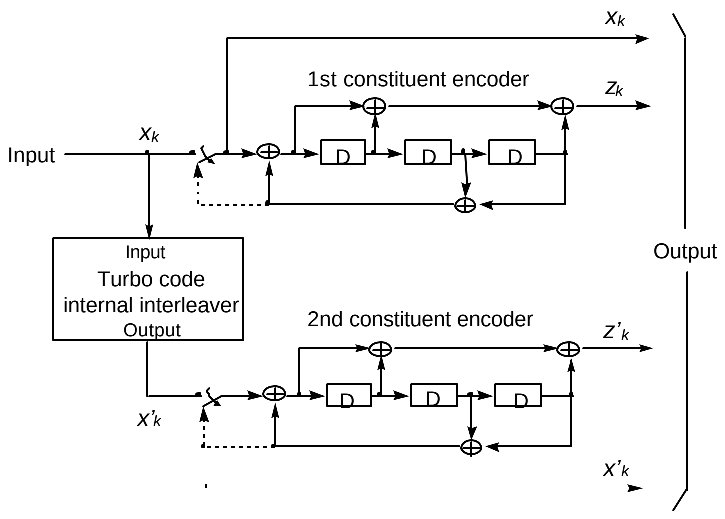

Figure 4: Structure of rate 1/3 Turbo coder (dotted lines apply for trellis termination only)

4.2.3.2.2 Trellis termination for Turbo coder

Trellis termination is performed by taking the tail bits from the shift register feedback after all information bits are encoded. Tail bits are padded after the encoding of information bits. The first three tail bits shall be used to terminate the first constituent encoder (upper switch of figure 4 in lower position) while the second constituent encoder is disabled. The last three tail bits shall be used to terminate the second constituent encoder (lower switch of figure 4 in lower position) while the first constituent encoder is disabled.
The transmitted bits for trellis termination shall then be:

$$x_{K+1}, z_{K+1}, x_{K+2}, z_{K+2}, x_{K+3}, z_{K+3}, x'_{K+1}, z'_{K+1}, x'_{K+2}, z'_{K+2}, x'_{K+3}, z'_{K+3}.$$

4.2.3.2.3 Turbo code internal interleaver

The Turbo code internal interleaver consists of bits-input to a rectangular matrix with padding, intra-row and inter-row permutations of the rectangular matrix, and bits-output from the rectangular matrix with pruning. The bits input to the Turbo code internal interleaver are denoted by $x_1, x_2, x_3, \Lambda, x_K$, where K is the integer number of the bits and takes one value of $40 \le K \le 5114$. The relation between the bits input to the Turbo code internal interleaver and the bits input to the channel coding is defined by $x_k = o_{irk}$ and $K = K_i$.

The following subclause specific symbols are used in subclauses 4.2.3.2.3.1 to 4.2.3.2.3.3:

K	Number of bits input to Turbo code internal interleaver
R	Number of rows of rectangular matrix
C	Number of columns of rectangular matrix
p	Prime number
v	Primitive root
$\langle s(j) \rangle_{j \in \{0,1,\Lambda,p-2\}}$	Base sequence for intra-row permutation
q_i	Minimum prime integers

r_i	Permuted prime integers
$\langle T(i) \rangle_{i \in \{0,1,\Lambda,R-1\}}$	Inter-row permutation pattern
$\langle U_i(j) \rangle_{j \in \{0,1,\Lambda,C-1\}}$	Intra-row permutation pattern of *i*-th row
i	Index of row number of rectangular matrix
j	Index of column number of rectangularmatrix
k	Index of bit sequence

4.2.3.2.3.1 Bits-input to rectangular matrix with padding

The bit sequence $x_1, x_2, x_3, \mathrm{K}, x_K$ input to the Turbo code internal interleaver is written into the rectangular matrix as follows.

(1) Determine the number of rows of the rectangular matrix, *R*, such that:

$$R = \begin{cases} 5, \text{ if } (40 \le K \le 159) \\ 10, \text{ if } ((160 \le K \le 200) \text{ or } (481 \le K \le 530)) \\ 20, \text{ if } (K = \text{any other value}) \end{cases} .$$

The rows of rectangular matrix are numbered 0, 1, …, R - 1 from top to bottom.

(2) Determine the prime number to be used in the intra-permutation, *p*, and the number of columns of rectangular matrix, *C*, such that:

if (481 $\le K \le$ 530) then

$p = 53$ and $C = p$.

else

Find minimum prime number p from table 2 such that

$$K \le R \times (p+1),$$

and determine C such that

$$C = \begin{cases} p-1 & if \quad K \le R \times (p-1) \\ p & if \quad R \times (p-1) < K \le R \times p \\ p+1 & if \quad R \times p < K \end{cases} .$$

end if

The columns of rectangular matrix are numbered 0, 1, …, C - 1 from left to right.

Table 2: List of prime number *p* and associated primitive root *v*

p	*v*	*p*	*v*	*p*	*v*	*p*	*v*	*p*	*v*
7	3	47	5	101	2	157	5	223	3
11	2	53	2	103	5	163	2	227	2
13	2	59	2	107	2	167	5	229	6
17	3	61	2	109	6	173	2	233	3
19	2	67	2	113	3	179	2	239	7
23	5	71	7	127	3	181	2	241	7
29	2	73	5	131	2	191	19	251	6
31	3	79	3	137	3	193	5	257	3
37	2	83	2	139	2	197	2		
41	6	89	3	149	2	199	3		
43	3	97	5	151	6	211	2		

(3) Write the input bit sequence $x_1, x_2, x_3, \ldots, x_K$ into the $R \times C$ rectangular matrix row by row starting with bit y_1 in column 0 of row 0:

$$\begin{bmatrix} y_1 & y_2 & y_3 & \ldots & y_C \\ y_{(C+1)} & y_{(C+2)} & y_{(C+3)} & \ldots & y_{2C} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ y_{((R-1)C+1)} & y_{((R-1)C+2)} & y_{((R-1)C+3)} & \ldots & y_{R\times C} \end{bmatrix}$$

where $y_k = x_k$ for $k = 1, 2, \ldots, K$ and if $R\times C > K$, the dummy bits are padded such that $y_k = 0 or 1$ for $k = K + 1, K + 2, \ldots, R\times C$. These dummy bits are pruned away from the output of the rectangular matrix after intra-row and inter-row permutations.

4.2.3.2.3.2 Intra-row and inter-row permutations

After the bits-input to the $R\times C$ rectangular matrix, the intra-row and inter-row permutations for the $R\times C$ rectangular matrix are performed stepwise by using the following algorithm with steps (1) – (6):

(1) Select a primitive root *v* from table 2 in section 4.2.3.2.3.1, which is indicated on the right side of the prime number *p*.

(2) Construct the base sequence $\langle s(j)\rangle_{j\in\{0,1,\ldots,p-2\}}$ for intra-row permutation as:

$s(j) = (n \times s(j-1)) \bmod p$, $j = 1, 2,\ldots, (p - 2)$, and $s(0) = 1$.

(3) Assign $q_0 = 1$ to be the first prime integer in the sequence $\langle q_i\rangle_{i\in\{0,1,\ldots,R-1\}}$, and determine the prime integer q_i in the sequence $\langle q_i\rangle_{i\in\{0,1,\ldots,R-1\}}$ to be a least prime integer such that g.c.d(q_i, p - 1) = 1, $q_i > 6$, and $q_i > q_{(i-1)}$ for each $i = 1, 2, \ldots, R - 1$. Here g.c.d. is greatest common divisor.

(4) Permute the sequence $\langle q_i\rangle_{i\in\{0,1,\ldots,R-1\}}$ to make the sequence $\langle r_i\rangle_{i\in\{0,1,\ldots,R-1\}}$ such that

$r_{T(i)} = q_i$, $i = 0, 1, \ldots, R - 1$,

where $\langle T(i)\rangle_{i\in\{0,1,\ldots,R-1\}}$ is the inter-row permutation pattern defined as the one of the four kind of patterns, which are shown in table 3, depending on the number of input bits K.

Table 3: Inter-row permutation patterns for Turbo code internal interleaver

Number of input bits	Number	Inter-row permutation patterns

K	of rows R	$<T(0), T(1), \ldots, T(R-1)>$
$(40 \le K \le 159)$	5	<4, 3, 2, 1, 0>
$(160 \le K \le 200)$ or $(481 \le K \le 530)$	10	<9, 8, 7, 6, 5, 4, 3, 2, 1, 0>
$(2281 \le K \le 2480)$ or $(3161 \le K \le 3210)$	20	<19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 16, 13, 17, 15, 3, 1, 6, 11, 8, 10>
K = any other value	20	<19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 10, 8, 13, 17, 3, 1, 16, 6, 15, 11>

(5) Perform the i-th ($i = 0, 1, \ldots, R - 1$) intra-row permutation as:

```
if ($C = p$) then
    $U_i(j) = s((j \times r_i) \mathrm{mod} (p-1))$,  $j = 0, 1, \ldots, (p - 2)$, and $U_i(p - 1) = 0$,
    where $U_i(j)$ is the original bit position of $j$-th permuted bit of $i$-th row.
end if
if ($C = p + 1$) then
    $U_i(j) = s((j \times r_i) \mathrm{mod} (p-1))$,  $j = 0, 1, \ldots, (p - 2)$.  $U_i(p - 1) = 0$, and $U_i(p) = p$,
    where $U_i(j)$ is the original bit position of $j$-th permuted bit of $i$-th row, and
    if ($K = R \times C$) then
        Exchange $U_{R-1}(p)$ with $U_{R-1}(0)$.
    end if
end if
if ($C = p - 1$) then
    $U_i(j) = s((j \times r_i) \mathrm{mod} (p-1)) - 1$,   $j = 0, 1, \ldots, (p - 2)$,
    where $U_i(j)$ is the original bit position of $j$-th permuted bit of $i$-th row.
end if
```

(6) Perform the inter-row permutation for the rectangular matrix based on the pattern $\langle T(i) \rangle_{i \in \{0,1,\Lambda,R-1\}}$,

where $T(i)$ is the original row position of the i-th permuted row.

4.2.3.2.3.3 Bits-output from rectangular matrix with pruning

After intra-row and inter-row permutations, the bits of the permuted rectangular matrix are denoted by y'_k:

$$\begin{bmatrix} y'_1 & y'_{(R+1)} & y'_{(2R+1)} & \mathrm{K} & y'_{((C-1)R+1)} \\ y'_2 & y'_{(R+2)} & y'_{(2R+2)} & \mathrm{K} & y'_{((C-1)R+2)} \\ \mathrm{M} & \mathrm{M} & \mathrm{M} & \mathrm{K} & \mathrm{M} \\ y'_R & y'_{2R} & y'_{3R} & \mathrm{K} & y'_{C \times R} \end{bmatrix}$$

The output of the Turbo code internal interleaver is the bit sequence read out column by column from the intra-row and inter-row permuted $R \times C$ rectangular matrix starting with bit y'_1 in row 0 of column 0 and ending with bit y'_{CR} in row R - 1 of column C - 1. The output is pruned by deleting dummy bits that were padded to the input of the rectangular matrix before intra-row and inter row permutations, i.e. bits y'_k that corresponds to bits y_k with $k > K$ are removed from the output. The bits output from Turbo code internal interleaver are denoted by $x'_1, x'_2, \ldots, x'_K$, where x'_1 corresponds to the bit y'_k with smallest index k after pruning, x'_2 to the bit y'_k with second smallest index k after pruning, and so on. The number of bits output from Turbo code internal interleaver is K and the total number of pruned bits is:

$$R \times C - K.$$

Appendix C: Direct C implementation of interleaver algorithm

```
/***************************************************************************************/
/***************************************************************************************/
/* Fuction Name:        fn_Interleaver                                               */
/* File Name:           INTERLEAVER.C (Interleaver.c)                                */
/* Author:              Paul Ampadu                                                  */
/* Company:             IBM T.J. Watson Research Center & Cornell University         */
/* Date:                June 25, 2001                                                */
/* Last Modified:       July 5, 2001                                                 */
/*                                                                                   */
/* Comments:            Based on V3.5.0 (2000-12) of 3GPP TS 25.212 Rel 1999         */
/*                                              */
/* FUNCTIONALITY: Implements 3GPP Turbo Code Internal Interleaver.                   */
/* User chooses Interleave (Deint=0) or Deinterleave (Deint=1)                       */
/* All notation follows 3GPP TS 25.212 v3.5.0 standard                               */
/*                                                                                   */
/* FUNCTIONS USED
            int         fn_gcd(int x,int y)     Returns greatest common divisor of x and y      */
/*                                                                                  */
/***************************************************************************************/
/***************************************************************************************/

#include "interleaver.h"

int fn_Interleaver(double Input_Sequence[MAX_LENGTH_K],int K,double *Output,int Deint)
{//Begin Function fn_Interleaver()

// INTER-ROW PERMUTATION PATTERNS

/* Store Vectors of Inter-Row Permutation Patterns */
int T_Vector_A[5]       = {4, 3, 2, 1, 0};
int T_Vector_B[10]      = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
int T_Vector_C[20]      = {19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 16, 13, 17, 15, 3, 1, 6, 11, 8, 10};
int T_Vector_D[20]      = {19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 10, 8, 13, 17, 3, 1, 16, 6, 15, 11};

int *T = NULL;                                  /* Will point to selected permutation pattern vector */

/* Declaration of Local and Temporary Variables */

int Rect_Matrix[MAX_ROWS*MAX_COLS];                     /* Rectangular Array Matrix */
int Temp_Matrix[MAX_ROWS*MAX_COLS];                     /* Temporary Matrix */
int Matrix_Index;                                       /* Computed Matrix Index */

int U_ij[MAX_ROWS*MAX_COLS];                            /* Intra-Row Permutation Matrix */

int Sequence[MAX_LENGTH_K];                             /* Interleaved Sequence */
int Sequence2[MAX_LENGTH_K];                            /* Temporary Sequence */

int R;                                                  /* Number of Rows of Rectangular Matrix */
int C;                                                  /*   Number   of   Columns   of   Rectangular
Matrix */
int p,v;                                                /* p Prime Number, v Primitive Root */
int p_Index;                                            /* Prime Number Index in Prime_Table */

int s[256];                                     /* Base Sequence for Intra-Row Permutation */
int q[20];                                      /* Least Primes Permuter Sequence */
int r[20];                                      /* Permuted Least Primes Sequence */
int i,j,temp;                                   /* Temporary Variables and Counters */
```

```
/* Table of Ordered Prime Numbers p from 7 to 257 */
int Prime_Table[52]      =
{7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,163
,167,173,179,181,191,193,197,199,211,223,227,229,233,239,241,251,257};

/* Table of Associated Primitive Roots for Ordered Prime_Table */
int Root_Table[52]       =
{3,2,2,3,2,5,2,3,2,6,3,5,2,2,2,2,7,5,3,2,3,5,2,5,2,6,3,3,2,3,2,2,6,5,2,5,2,2,2,19,5,2,3,2,3,2,6,3,7,7,6,3};

/*********************************************************************************************/

/********** STEP 1: BITS-INPUT TO RECTANGULAR MATRIX WITH PADDING   **********/

/*********************************************************************************************/
// DETERMINE NUMBER OF ROWS (R), PRIME NUMBER (p), AND NUMBER OF COLUMNS (C) OF
RECTANGULAR MATRIX
int Var;                                                    /* Dummy Case Variable */
if((481<=K) && (K<=530))
{
        Var = 0;
        R = 10;
        p = 53;
        C = p;
        v = 2;                                              /* Set v Corresponding to p = 53 */
}
else
{
        if ((40<=K) && (K<=159))
        {
                Var = 1;
                R = 5;
        }
        else if ((160<=K) && (K<=200))
        {
                Var = 2;
                R = 10;
        }
        else
        {
                Var = 3;
                R = 20;
        }
        //FIND MINIMUM PRIME p FROM PRIME_TABLE SUCH THAT K <= R(p+1)
        /* This implementation obviates need for division */

        i = 0;
        while (K > (R*Prime_Table[i] + 1))
                i++;

        p = Prime_Table[i];
        p_Index = i;                                        /* Store Index in p_Index, for Later Use */

//DETERMINE NUMBER OF COLUMNS C
        if (K <= R*(p-1))
                C = p - 1;
        else if (K > R*p)
                C = p +1;
        else
                C = p;
} // End Else
```

```
// CREATE (RXC) MATRIX FOR K-INPUT WITH PADDING

/* Compute Index of Rect_Matrix Corresponding to 2D Logical Rectangular Array */
i = 0;
while (i < K)
{
        Matrix_Index = (i/C)*MAX_COLS + (i%C);
        Rect_Matrix[Matrix_Index] = i;                          /* Build K Input SubMatrix */
        i++;
}       // End While


// PAD REMAINING (RxC)-K INDICES WITH DUMMY BITS (-1) FOR LATER PRUNING
/* Padding Necessary Only If (R*C)>K. Don't Care About Remaining (MAX_ROW*MAX_COL)-(R*C) Indices */

if (R*C > K)
{
        for (i=K; i<(R*C); i++)
        {
                Matrix_Index = (i/C)*MAX_COLS + (i%C);
                Rect_Matrix[Matrix_Index] = -1;
        }
} // End If

/*********************************************************************************************/

/********** STEP 2: INTRA-ROW AND INTER-ROW PERMUTATIONS  **********/

/*********************************************************************************************/
// Part 1: SELECT PRIMITIVE ROOT v FROM ROOT_TABLE CORRESPONDING TO PRIME p

v = Root_Table[p_Index];


// Part 2: CONSTRUCT BASE SEQUENCE s[j] (of Size p-1) FOR INTRA-ROW PERMUTATIONS

s[0] = 1;               /* Initialize First Element of Base Sequence to 1 */

for (j=1; j<=(p-2); j++)
        s[j] = (v*s[j-1]) % p;

// Part 3: CREATE PERMUTER SEQUENCE q[i] (of Size R)

q[0] = 1;                       /* Initialize First Prime in Least Prime Sequence to 1 */
j = 1;
i = 0;

/* The sequence satisfies q[i]>6, and q[i]>q[i-1] */

while (j < R)
{
        if (fn_gcd(Prime_Table[i], (p-1)) == 1)
        {
                q[j] = Prime_Table[i];
                j++;
        }

        i++;
} //End While
```

```
switch (Var)
{
case 1:         T = T_Vector_A;                                 // 40<=K<=159
                        break;
case 0:
case 2:
                        T = T_Vector_B;                         // 160<=K<=200 OR 481<=K<=530
                        break;
default:
                if ((2281<=K && K<=2480) || (3161<=K && K<=3210))
                        T = T_Vector_C;                         // 2281<=K<=2480 OR 3161<=K<=3210
                else
                        T = T_Vector_D;                         // K is any other value
} //End Switch



// CREATE PERMUTED SEQUENCE r[i] DERIVED FROM q[i]

for (i=0; i<R; i++)
                r[T[i]] = q[i];

// Part 5: PERFORM INTRA-ROW PERMUTATIONS

for (i=0; i<R; i++)
                for (j=0; j<(p-1); j++)
                        U_ij[i*MAX_COLS + j] = s[(j*r[i])%(p-1)];



if (C == p)
                for (j=0; j<R; j++)                             // Set End Column of Each Row to 0
                        U_ij[(j*MAX_COLS)+(p-1)] = 0;



else if (C == (p+1))
{
                for (j=0; j<R; j++)
                {
                        U_ij[(j*MAX_COLS)+(p-1)] = 0;           // Set Last-but-One Column to 0 and
                        U_ij[(j*MAX_COLS)+p] = p;               // Set End Column to p
                }

                if (K = R*C)
                {
                        temp = U_ij[(R-1)*MAX_COLS];
                        U_ij[(R-1)*MAX_COLS] = U_ij[(R-1)*MAX_COLS + p];
                        U_ij[(R-1)*MAX_COLS+p] = temp;
                }
}



else                                                            //(C == (p-1))
{
                for (i=0; i<R; i++)
                        for (j=0; j<p-1; j++)
                                U_ij[i*MAX_COLS + j] = U_ij[i*MAX_COLS + j] - 1;
}
```

```
/* Perform Intra-Row Permutations */
for (j=0; j<R; j++)
{
        for(i=0; i<C; i++)
        Temp_Matrix[j*MAX_COLS + i] = Rect_Matrix[(j*MAX_COLS) + U_ij[j*MAX_COLS + i]];
}



// Part 6: INTER-ROW PERMUTATION (BASED ON {T[i]} PATTERN) FOLLOWING INTRA-ROW MATRIX
PERMUTATIONS

for (j=0; j<R; j++)
{
        for (i=0; i<C; i++)
                Rect_Matrix[j*MAX_COLS + i] = Temp_Matrix[T[j]*MAX_COLS + i];
}




/****************************************************************************************************
/

/**********                STEP 3: BITS-OUTPUT FROM RECTANGULAR MATRIX WITH PRUNING
**********/

/****************************************************************************************************
/

// READ AND PRUNE BITS-OUTPUT FROM RECTANGULAR MATRIX

j=0;

/* Bit Output Sequence Read Out Column by Column */

for (i=0; i<R*C; i++)
{
        Matrix_Index = (i%R)*MAX_COLS + (i/R);
        if (Rect_Matrix[Matrix_Index] != -1)
        {
                Sequence[j] = Rect_Matrix[Matrix_Index];
                j++;
        }
}
```

```
/*************************************************************************************************/

/********** MISCELLANEOUS OPERATIONS **********/

/*************************************************************************************************/

// CHOOSE BETWEEN INTERLEAVER AND DEINTERLEAVER
// INTERLEAVER (DEINT = = 0)
if (Deint= =0)
{
        for (i=0; i<K; i+ +)
        {
                *Output = Input_Sequence[Sequence[i]];
                Output+ +;
        }
}
// DEINTERLEAVER (DEINT = = 1)
else
{

        for (i=0; i<K; i+ +)
                Sequence2[Sequence[i]] = i;

        for (i=0; i<K; i+ +)
        {
                *Output = Input_Sequence[Sequence2[i]];
                Output+ +;
        }
}

return 0;

} // End Function fn_Interleaver()
```

Appendix D: C implementation of our architecture

```
/*******************************************************************************************/
/*******************************************************************************************/
/* Fuction Name:        fn_Interleaver                                                   */
/* File Name:           INTERLEAVER.C (Interleaver.c)                                    */
/* Author:              Paul Ampadu                                                      */
/* Company:             IBM T.J. Watson Research Center & Cornell University             */
/* Date:                June 25, 2001                                                    */
/* Last Modified:       July 5, 2001                                                     */
/*                                                                                       */
/* Comments:            Based on V3.5.0 (2000-12) of 3GPP TS 25.212 Rel 1999             */
/*                                                                                       */
/* FUNCTIONALITY: Implements 3GPP Turbo Code Internal Interleaver.                       */
/*          User chooses Interleave (Deint=0) or Deinterleave (Deint=1)                  */
/*          All notation follows 3GPP TS 25.212 v3.5.0 standard                          */
/*                                                                                       */
/* FUNCTIONS USED
            int         fn_gcd(int x,int y)     Returns greatest common divisor of x and y    */
/*                                                                                       */
/*******************************************************************************************/
/*******************************************************************************************/
#include "interleaver.h"

int fn_Interleaver(double Input_Sequence[MAX_LENGTH_K],int K,double *Output,int Deint)
{//Begin Function fn_Interleaver()

// ROM STORAGE
/* Store Vectors of Inter-Row Permutation Patterns */
int T_Vector_A[5]       = {4, 3, 2, 1, 0};
int T_Vector_B[10]      = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
int T_Vector_C[20]      = {19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 16, 13, 17, 15, 3, 1, 6, 11, 8, 10};
int T_Vector_D[20]      = {19, 9, 14, 4, 0, 2, 5, 7, 12, 18, 10, 8, 13, 17, 3, 1, 16, 6, 15, 11};

/* Table of Ordered Prime Numbers p from 7 to 257 */
int Prime_Table[52]     =
{7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,163
,167,173,179,181,191,193,197,199,211,223,227,229,233,239,241,251,257};

/* Table of Associated Primitive Roots for Ordered Prime_Table */
int Root_Table[52]= {3,2,2,3,2,5,2,3,2,6,3,5,2,2,2,2,7,5,3,2,3,5,2,5,2,6,3,3,2,3,2,2,6,5,2,5,2,2,2,19,5,2,3,2,3,2,6,3,7,7,6,3};

int *T = NULL;                                  /* Will point to selected permutation pattern vector */
int Data_Memory[MAX_LENGTH_K];
int Sequence[MAX_LENGTH_K];
int Sequence2[MAX_LENGTH_K];

/* Declaration of Local and Temporary Variables */
int R;                                          /* Number of Rows of Rectangular Matrix */
int C;                                          /* Number of Columns of Rectangular Matrix */
int p,v;                                        /* p Prime Number, v Corresponding Primitive Root */
int p_Index;                                    /* Prime Number Index in Prime_Table */
int s[256];                                     /* Base Sequence for Intra-Row Permutation */
int q[20];                                      /* Least Primes Permuter Sequence */
int r[20];                                      /* Permuted Least Primes Sequence */
int i,j;                                        /* Temporary Variables and Counters */

int Interleaved_Address;
```

```
/*****************************************************************************************************/

/**********  STEP 1: BITS-INPUT TO RECTANGULAR MATRIX WITH PADDING            **********/

/*****************************************************************************************************/


// DETERMINE NUMBER OF ROWS (R), PRIME NUMBER (p), AND NUMBER OF COLUMNS (C) OF
RECTANGULAR MATRIX

int Var;                                          /* Dummy Case Variable */

if((481<=K) && (K<=530))
{
        Var = 0;
        R = 10;
        p = 53;
        C = p;
        v = 2;                                    /* Set v Corresponding to p = 53 */
}

else
{
        if ((40<=K) && (K<=159))
        {
                Var = 1;
                R = 5;
        }
        else if ((160<=K) && (K<=200))
        {
                Var = 2;
                R = 10;
        }
        else
        {
                Var = 3;
                R = 20;
        }

        //FIND MINIMUM PRIME p FROM PRIME_TABLE SUCH THAT K <= R(p+1)
        /* This implementation obviates need for division */
        i = 0;

        while (K > (R*Prime_Table[i] + 1))
                i++;

        p = Prime_Table[i];
        p_Index = i;                              /* Store Index in p_Index, for Later Use */

        //DETERMINE NUMBER OF COLUMNS C
        if (K <= R*(p-1))
                C = p - 1;
        else if (K > R*p)
                C = p +1;
        else
                C = p;
} // End Else
```

```
i = 0;
while (i < K)
{
        Data_Memory[i] = i;                         /* Build K Input Sub matrix */
        i++;
}       // End While
/*******************************/

// PAD REMAINING (RxC)-K INDICES WITH DUMMY BITS (-1) FOR LATER PRUNING
/* Padding Necessary Only If (R*C)>K */
if (R*C > K)
{
        for (i=K; i<(R*C); i++)
        {
                Data_Memory[i] = -1;
        }
} // End If

// Part 1: SELECT PRIMITIVE ROOT v FROM ROOT_TABLE CORRESPONDING TO PRIME p
v = Root_Table[p_Index];

// Part 2: CONSTRUCT BASE SEQUENCE s[j] (of Size p-1) FOR INTRA-ROW PERMUTATIONS

s[0] = 1;           /* Initialize First Element of Base Sequence to 1 */

for (j=1; j<=(p-2); j++)
        s[j] = (v*s[j-1]) % p;

// Part 3: CREATE PERMUTER SEQUENCE q[i] (of Size R)

q[0] = 1;                                   /* Initialize First Prime in Least Prime Sequence to 1 */
j = 1;
i = 0;

/* The sequence satisfies q[i]>6, and q[i]>q[i-1] */
while (j < R)
{
        if (fn_gcd(Prime_Table[i], (p-1)) == 1)
        {
                q[j] = Prime_Table[i];
                j++;
        }

        i++;
} //End While
switch (Var)
{
case 1:  T = T_Vector_A;                            // 40<=K<=159
                break;
case 0:
case 2:
                T = T_Vector_B;                     // 160<=K<=200 OR 481<=K<=530
                break;
default:
        if ((2281<=K && K<=2480) || (3161<=K && K<=3210))
                T = T_Vector_C;                     // 2281<=K<=2480 OR 3161<=K<=3210
        else
                T = T_Vector_D;                     // K is any other value
} //End Switch
```

```
// CREATE PERMUTED SEQUENCE r[i] DERIVED FROM q[i]

for (i=0; i<R; i++)
        r[T[i]] = q[i];

//Permuted Sequence Array with pruning
if (C == p)
{
        i=0;

        for (j=0; j<R*(p-1); j++)
        {
                Interleaved_Address = C*T[j%R] + s[((j/R) * r[T[j%R]]) % (p-1)];
                if ((0<=Interleaved_Address)&&(Interleaved_Address<K))
                {
                        Sequence[i] = Data_Memory[Interleaved_Address];
                        i++;
                }
        }
        for (j=R*(p-1); j<(R*C); j++)
        {
                Interleaved_Address = C*T[j%R];
                if ((0<=Interleaved_Address)&&(Interleaved_Address<K))
                {
                        Sequence[i] = Data_Memory[Interleaved_Address];
                        i++;
                }
        }

}
else if (C == (p-1))
{
        i=0;
        for (j=0; j<R*(p-1); j++)
        {
                Interleaved_Address = C*T[j%R] + (s[((j/R) * r[T[j%R]]) % (p-1)]) - 1;
                if ((0<=Interleaved_Address)&&(Interleaved_Address<K))
                {
                        Sequence[i] = Data_Memory[Interleaved_Address];
                        i++;
                }
        }
}

else    //C == (p+1)
{
        i=0;

        for (j=0; j<R*(p-1); j++)
        {
                Interleaved_Address = C*T[j%R] + s[((j/R) * r[T[j%R]]) % (p-1)];
                if ((0<=Interleaved_Address)&&(Interleaved_Address<K))
                {
                        Sequence[i] = Data_Memory[Interleaved_Address];
                        i++;
                }
        }
        for (j=R*(p-1); j<(R*p); j++)
        {
                Interleaved_Address = C*T[j%R];
```

```
                    if ((0<=Interleaved_Address)&&(Interleaved_Address<K))
                    {
                              Sequence[i] = Data_Memory[Interleaved_Address];
                              i++;
                    }
          }

          for (j=(R*p); j<(R*(p+1)); j++)
          {
                    Interleaved_Address = C*T[j%R] + p;
                    if ((0<=Interleaved_Address)&&(Interleaved_Address<K))
                    {
                              Sequence[i] = Data_Memory[Interleaved_Address];
                              i++;
                    }
          }

}



/*************************************************************************************************/

/********** MISCELLANEOUS OPERATIONS   **********/

/*************************************************************************************************/

// CHOOSE BETWEEN INTERLEAVER AND DEINTERLEAVER

// INTERLEAVER (DEINT == 0)
if (Deint==0)
{
          for (i=0; i<K; i++)
          {
                    *Output = Input_Sequence[Sequence[i]];
                    Output++;
          }
}


// DEINTERLEAVER (DEINT == 1)
else
{

          for (i=0; i<K; i++)
                    Sequence2[Sequence[i]] = i;

          for (i=0; i<K; i++)
          {
                    *Output = Input_Sequence[Sequence2[i]];
                    Output++;
          }
}

return 0;

} // End Function fn_Interleaver()
```