Very Low-Complexity Hardware Interleaver for Turbo Decoding

Zhongfeng Wang, *Senior Member, IEEE*, and Qingwei Li

***Abstract*—This brief presents a very low complexity hardware interleaver implementation for turbo code in wideband CDMA (W-CDMA) systems. Algorithmic transformations are extensively exploited to reduce the computation complexity and latency. Novel VLSI architectures are developed. The hardware implementation results show that an entire turbo interleave pattern generation unit consumes only 4 k gates, which is an order of magnitude smaller than conventional designs.**

***Index Terms*—CDMA, interleaver, turbo codes, VLSI architecture.**

I. Introduction

TURBO code [1] invented in 1993 has been adopted in several industrial standards such as third generation CDMA systems [2], [3] due to its outstanding performance. Fig. 1 shows a turbo encoder structure and a serial turbo decoder structure, where $u[k], x_s, x_p^1$, and x_p^2 stand for source information bit, systematic bit, parity bit-1 and parity bit-2, respectively; y_s, y_p^1, and y_p^2 represent received soft symbols corresponding to x_s, x_p^1, and x_p^2, respectively; RSC stands for recursive systematic convolutional encoder; the soft-input soft-output (SISO) decoder outputs two soft messages at each time instance: the log likelihood ratio $L_R(k)$ and the extrinsic information $L_{\rm ex}(k)$. One of the key features of turbo code is the interleaver. At the encoder side, a block of information bits are interleaved and sent to RSC2 to generate parity bit-2. At the decoder side, the extrinsic information and y_s symbols are interleaved at the second decoding phase [4]. In practical implementation, the interleave process is performed by reading data in the interleaved order (note: the de-interleaving process is completed by writing data back to where they were loaded from. In this way, no de-interleave patterns are required). Therefore, an interleave pattern generation circuitry is needed, which serves as an address generator as shown in Fig. 1.

In WCDMA systems, turbo code block size varies from 40 to 5114 bits. Different block sizes require different interleave patterns. It can be derived that a ROM-based solution requires more than 100 M bits of storage for all the interleave patterns, which is unacceptable from the hardware cost point of view. In [5], a hardware interleaver solution was proposed by researchers in Cornell Broadband Communication Lab. The total hardware amounts to approximately 30 K gates. A processor-based solution presented in [6] used slightly more hardware while supporting turbo codes in CDMA2000 systems as well.

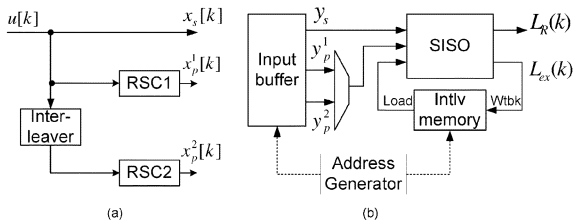

Fig. 1. (a) Turbo encoder structure. (b) Serial Turbo decoder structure.

Approaches for low power implementation of digital signal processing (DSP) systems have been addressed in many papers such as [7]. In this brief, we maximally exploit joint algorithm level, architecture level, and circuit level VLSI optimization approaches to eliminate costly multiplications, divisions, and modulo operations, in order to reduce the overall computation complexity, computing latency and power consumption of the target system. Our implementation results show that the proposed design has an order of magnitude lower hardware complexity than other published designs.

This brief is organized as follows. In Section II, the approach to compute the basic parameters is described. In Section III, we introduce two new methods to compute S and Q arrays, which are the two most complex parts in this interleaver. Then in Section IV, we briefly present some ways to save storage spaces. In Section V, we propose to change the permutation order, which can save some computation hardware and get rid of the delay of traditional permutation method. Section VI illustrates the VLSI design details and provides the implementation report. Finally, conclusions are drawn in Section VII. It should be mentioned that the idea of on-the-fly address generation by changing the permutation order discussed in Section V is similar to the approach proposed in [6], though the new work was independently developed.

Most of the variables used in the following discussion are matching the symbols used in standard [2]. (N matches for K in the standard, and, R for R, C for C, P for p, v for v, S for s, Q for q, U for U). For the detailed definition of each variable, please refer to [2].

II. Computation of Basic Parameters

A. Computation of R

$$R = \begin{cases} 5, & \text{if } (40 \le N \le 159) \\ 10, & \text{if } (160 \le N \le 200) \text{ or } (481 \le N \le 530) \\ 20, & \text{other } N. \end{cases} \quad (1)$$

Manuscript received November 29, 2006; revised January 30, 2007. Some information in this paper may be covered in a patent application. This paper was recommended by Associate Editor S. Tsukiyama.

Z. Wang was with Morphics Technology Inc, Campbell, CA 95008 USA. He is now with the School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331 USA (e-mail: zwang@eecs.orst.edu).

Q. Li is with the School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331 USA (e-mail: liqin@eecs.orst.edu).

Digital Object Identifier 10.1109/TCSII.2007.895313

The number of rows R is computed using (1), where N is the input parameter representing the block size. For the goal of low complexity, we use 3 cycles to make 3 comparisons: Cycle-1: check if $N > 200$, the decision bit is denoted as $D1$:$D1 = 1$ means $N > 200$; Cycle-2: check if $N > 480$, the decision bit is $D2$; Cycle-3: check $N > 530$ if the answer from Cycle-2 is "yes", otherwise check if $N \geq 160$, the decision bit is denoted as $D3$. The final value of R can be determined using a simple combinational logic based on the above three decision bits. To reduce the complexity of forthcoming computation, we only record the index of R:0 for $R = 5, 1$ for $R = 10$, and 2 for $R = 20$. Thus, we only need a 3-bit input and 2-bit output logic to determine R index.

B. Calculation of P, v, and C

The second step is to determine the prime number P and the number of columns C

$$P = \begin{cases} 53, C = P, & \text{if } (481 \leq N \leq 530) \\ \arg\min_P (P * R \geq N - R), & \text{other cases} \end{cases}$$

$$C = \begin{cases} P + 1, & \text{if } (P * R < N) \\ P - 1, & \text{if } (P * R \geq N + R) \\ P, & \text{otherwise.} \end{cases} \quad (2)$$

Based on the computation result from the first step, if $(D3 = 0)$ and $(D2 = 1)$, then $P = 53, C = P$. If this condition is not satisfied, we need to find a minimum prime number P such that $P * R \geq N - R$. A normal approach is to use binary search. As the total number of prime numbers to be considered is 52 (according to the WCDMA standard [2, Table II], P has 52 possible values), we need to perform 6 multiplication operation, 6 memory accesses and 12 addition/subtraction operations to determine the P value in general.

In this brief, we consider an indirect computation approach. Assume we store all P values $(7, 11, \ldots, 257)$ in a table (implemented with a ROM, starting with address "0"). To address the table for the target P value, we calculate an approximate index PI_2, by using some simple mapping function. Here, we construct such mapping function which guarantees the real P value to be stored in one of the four entries of the table indexed by $\mathrm{PI}_2 - 1, \mathrm{PI}_2, \mathrm{PI}_2 + 1$, and $\mathrm{PI}_2 + 2$ for any N and R. If $P[\mathrm{PI}_2] * R \geq N - R$, then check if $P[\mathrm{PI}_2 - 1] * R \geq N - R$. If $P[\mathrm{PI}_2] * R < N - R$, then see if $P[\mathrm{PI}_2 + 1] * R \geq N - R$. After 2 clock cycles, we will determine the index of target P. Thus, we can get P value and v value (the primitive root associated with prime number P, see standard [2, Table II]) if we store P and corresponding v in the same entry. The mapping function used in the design is a piecewise-linear function, which can be simply implemented with only add-and-shift operations.

III. Computation of S Array and Q Array

A. Computation of S Array

The S array is computed as follows with $S[0] = 1$:

$$S[k] = (v * S[k-1]) \bmod P, \qquad k = 1, 2, \ldots, P - 1. \quad (3)$$

Direct computation for S array will inevitably involve multiplications and modulo operations, which not only raises the hardware cost, but also increases the computing delay.

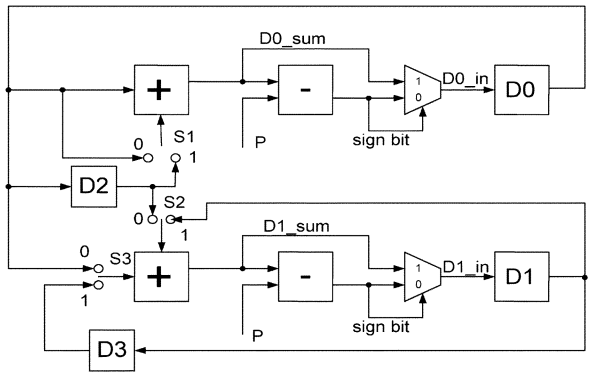

Fig. 2. Circuitry for computation of S array.

TABLE I
MUX Select Values at Different Clock Cycles

Clock Cycle	*S1*	*S2*	*S3*
0	0	1	0
1	0	0	0
2	1	1	1
3	0	0	1
4	0	1	0

Observe that v only has 6 values: 2, 3, 5, 6, 7, and 19. We propose to gradually compute $S[k]$ from $S[k-1]$. Assume $X < P$ and $Y < P$. We have the following:

$$(X + Y) \bmod P = \begin{cases} X + Y, & \text{if } (X + Y) < P \\ X + Y - P, & \text{otherwise.} \end{cases}$$

Since $S[k-1] < P, (S[k-1] * 2 \bmod P) = S[k-1] * 2$ or $S[k-1] * 2 - P$ depending on whether $S[k-1] * 2 < P$ or not. Let $X = S[k-1] * 2 \bmod P, Y = S[k-1]$, we can compute $S[k-1] * 3 \bmod P$ using (3). Fig. 2 shows the circuitry to compute $S[k]$ from $S[k-1]$ for any value of v.

The basic strategy of this design is to take different number of cycles to compute a new S value for different v. Specifically, we take one cycle to compute $(S[k-1] * 2) \bmod P$, two cycles to computer $(S[k-1] * 3) \bmod P$ and $(S[k-1] * 4) \bmod P$, three cycles for $v = 5$ and $v = 6$, four cycles for $v = 7$, and five cycles for $v = 19$. In case of $v = 19$, it takes five cycles per iteration to compute $S[k]$ from $S[k-1]$ (i.e., 5 cycles per S entry). During these five cycles, the register D0 sequentially outputs $2 * S[k-1] \bmod P, 4 * S[k-1] \bmod P, 6 * S[k-1] \bmod P, 12 * S[k-1] \bmod P$, and $24 * S[k-1] \bmod P$. The register D1 sequentially outputs $2 * S[k-1] \bmod P, 3 * S[k-1] \bmod P, 5 * S[k-1] \bmod P, 7 * S[k-1] \bmod P$, and $19 * S[k-1] \bmod P$. Note, this circuit only requires 4 adders and 4 registers and some simple switching/multiplexing elements. The selection signals of multiplexers $S1, S2$ and $S3$ can be generated from a small look-up table as indicated in Table I.

B. Computation of Q Array

The Q array is computed as follows according to the standard [2]: Compute $Q[j], j = 1, 2, \ldots, R - 1$, such that $\mathrm{GCD}(Q[j], P - 1) = 1$ and $Q[j]$ is a prime number, $Q[0] = 1, Q[j] > Q[j-1]$, and $Q[j] > 6$, where GCD stands for great common divisor function.

Directly computing GCD is a recursive process, and it is more complex than performing a few division operations. From simulation, we found out that the Q array is a subset of a group of sequential prime numbers W' (i.e., $1, 7, 11, 13, \ldots, 83, 89$). In particular, for any value of P, Q array is a subset of $R+2$ sequential prime numbers W (the first $R+2$ entries of W'), where R is the number of rows corresponding to the given N value. As Q array contains exactly R elements, we need to record at most two void indexes for each P. These void indices, which can be easily identified from simulations, refer to the indexes of the sequential prime number array W whose entries that do not belong to Q array. For instance, when $P = 53, R = 20, P-1 = 52 = 2*2*13, 13$ is the fourth entry of prime array W whereas not an element of Q array. So the void index is 3. From our simulation, we know the maximum void index is 20. Therefore, we need 5 bits to store one void index. If there is no void index, we store 0. There is only one case that Q has two void indices. When $P = 239, R = 20, P-1 = 238 = 2*7*17$; two prime numbers, 7 and 17, are not included in the Q array. Hence, two void indexes are 1 and 4. For this special case, we store $0 \times 01\_100$ (12 in decimal) in the table. Here since we know 12 is not a void index for any other value of P, we use it for this special case. Surely we could store another number that is not used for any cases. But this proposed setting will lead to minimal hardware cost since we can easily split $0 \times 01\_100$ into 0×01 and 0×100.

As will be shown in later discussions (Section V-B), what we really care is $Q[j] \bmod (P-1)$ instead of $Q[j]$ itself. We introduce a Q ROM that has 22 entries and stores $Q[j] - Q[j-1]$, i.e., the first entry stores 1, the second entry store $7-1=6$, the third entry stores $11-7=4$, *et al.* We will use the following circuitry to recursive compute $Q[j] \bmod (P-1)$ without introducing modulo operations.

It should be noted that the output of the circuit will be dropped when a void index matches the running index j.

IV. Storage of P, v, and Void Indices of Q

From the above discussions, it is clear that we need to store 52 sequential prime numbers P, their corresponding v values, as well as void index/indices for corresponding Q array. Since $P_{\max} = 257, v_{\max} = 19, Q_{\text{voidmax}} = 20$, A straightforward way needs 19 bits per entry. However, some approaches can be taken to save storage space.

A. Storage of P Values

The maximum prime number is 257, which requires 9 bits. If we do not store the least significant bit, we need 8 bits to store each P value. If we store $P' = (P-3)/2$ in the table, only 7 bits are needed per entry. In this case, we need one addition and one shift operations to recover P. A more aggressive approach is to store $P'' = (P-1)/2 - 3*i + 27$ in the table, where i denotes the index of the P value in the table. From simulation, we know P'' takes values from 0 to 30. Thus, we only need 5 bits to store for each P. We can recover P from P'' as follows:

$$P = (P'' + i + (i \ll 1) - 27) \ll 1 + 1. \tag{4}$$

The above computation involves three additional operations and one shift operation. In this design, we allocate 8 bits (by dropping the LSB) per entry to save further computation in recovering real value of P.

B. Storage of v Values

As v has only 6 values: 2, 3, 5, 6, 7, and 19. We use 3 bits to record the index of v, i.e., 0 for 2, 1 for 3, 3 for 5, 4 for 6, 6 for 7, and 7 for 19. Here we did not choose continuous indices, the reason is that our selection is optimized for computation of S array (refer to Section III-A). Specifically, the number of cycles per iteration in computing an S array entry is now directly related to the value of v index (denoted as v_idx) via a simple equation

$$\text{Clock\_cycle} = (v\_\text{idx} + 3) \gg 1 \tag{5}$$

where "$\gg$" denotes right shift operation. For instance, when $v = 3$, we need $(v\_\text{idx} + 3) \gg 1 = 2$ cycles to compute $S[k]$ from $S[k-1]$ while we need 5 cycles when $v = 19$.

As discussed in Section III, we need 5 bits to record void index for each P. In all, we need $8 + 3 + 5 = 16$ bits per entry (if we use 5 bits for each P, then 13 bit per entry) and we have 52 entries for the ROM.

V. On-Line Computation of Interleave Patterns

What we discussed before is the computation of the important parameters (R, P, v, C, S array, and Q array), which are used to compute the exact permutated address for each bit. Here we call the above process to calculate these parameters "Pre-computation." In this section, we discuss the method to compute valid interleave addresses one by one. We call this process "On-line computation." In practice, we output one valid interleave address almost every cycle.

A. Change of the Permutation Order

According to the 3GPP standard, the online operation order is: 1. intra-row permutation, 2.inter-row permutation, 3.read out by column, and 4.prune invalid bit. However, for practical implementations, this order is not the most efficient order and introduces unnecessary hardware and computational complexity. In later discussion, we proposed a method which is more efficient for smaller hardware area and higher speed.

Suppose the input bit stream is $A_0, A_1, \ldots, A_{N-1}$, after inserting the dummy bits and written by row, it becomes

$$\begin{bmatrix} X_{0,0} & X_{0,1} & \cdots & X_{0,C-1} \\ X_{1,0} & X_{1,1} & \cdots & X_{1,C-1} \\ \vdots & \vdots & & \vdots \\ X_{R-1,0} & X_{R-1,1} & \cdots & X_{R-1,C-1} \end{bmatrix}$$

and we have the relation $X_{i,j} = A_{i*C+j}$. If $i*C+j \geq N$, then $X_{i,j}$ is a dummy bit.

For the intra-row permutation, it calculates the parameter $U_{i,j}$, which is the original bit position of jth permuted bit of ith row, as

$$U_{ij} = S((j * r_i) \bmod (P-1)), \qquad j = 0, 1, \ldots, p-2 \tag{6}$$

where r array is defined as $r_{T(i)} = Q_i, i = 0, 1, \ldots, R-1$, and $T(i)_{i \in \{0,1,\ldots,R-1\}}$ is the inter-row permutation pattern pre-

defined according to different R values [2]. After the intra-row permutation, the pattern becomes

$$\begin{bmatrix} Y_{0,0} & Y_{0,1} & \cdots & Y_{0,C-1} \\ Y_{1,0} & Y_{1,1} & \cdots & Y_{1,C-1} \\ \vdots & \vdots & & \vdots \\ Y_{R-1,0} & Y_{R-1,1} & \cdots & Y_{R-1,C-1} \end{bmatrix}$$

where $Y_{i,j} = X_{i,Uij}$, such operation can be denoted as

$$\begin{aligned} &\text{loop } i : i \text{ from 0 to } R-1 \\ &\quad \text{loop } j : j \text{ from 0 to } C-1 \\ &\quad\quad \text{compute } U_{ij} \text{ based on (6)} \\ &\quad\quad Y_{i,j} = X_{i,Uij}. \end{aligned}$$

The inter-row permutation permutes the rows according to $T(i)$, where $T(i)$ is the original position of the jth permuted row. After inter-row permutation, the pattern becomes

$$\begin{bmatrix} Z_{0,0} & Z_{0,1} & \cdots & Z_{0,C-1} \\ Z_{1,0} & Z_{1,1} & \cdots & Z_{1,C-1} \\ \vdots & \vdots & & \vdots \\ Z_{R-1,0} & Z_{R-1,1} & \cdots & Z_{R-1,C-1} \end{bmatrix}$$

and $Z_{i,j} = Y_{T(i),j} = X_{T(i),U_{T(i)j}} = A_{T(i)*C+U_{T(i)j}}$, so the inter-row permutation can be expressed as

$$\begin{aligned} &\text{loop } i : i \text{ from 0 to } R-1 \\ &\quad \text{loop } j : j \text{ from 0 to } C-1 \\ &\quad\quad Z_{i,j} = Y_{T(i),j}. \end{aligned}$$

Therefore, the intra-row and inter-row permutation can be combined as

$$\begin{aligned} &\text{loop } i : i \text{ from 0to } R-1 \\ &\quad \text{loop } j : j \text{ from 0 to } C-1 \\ &\quad\quad \text{compute } U_{ij} \\ &\quad\quad Z_{i,j} = Y_{T(i),j} = X_{T(i),U_{T(i)\text{j}}}. \end{aligned}$$

After the permutations, the Z sequence is output by column, $Z_{0,0}, Z_{1,0}, \ldots, Z_{R-1,0}, Z_{0,1}, Z_{1,1}, \ldots, Z_{R-1,1}, \ldots, Z_{0,C-1}, \ldots, Z_{R-1,C-1}$. A straightforward method is to calculate Z matrix first, store the values, and then read it out by column. However, a wiser way is to do the permutation using j as outer loop, i as inner loop. Then the calculation sequence of Z is the same as output sequence. In this way, we can calculate an address, check if it is valid (not a dummy bit), and output it. Thereby, no extra storage space for Z and no latency introduced here.

Our method can be denoted as

$$\begin{aligned} &\text{loop } j : j \text{ from 0 to } C-1 \\ &\quad \text{loop } i : i \text{ from 0 to } R-1 \\ &\quad\quad \text{compute } U_{ij} \\ &\quad\quad \text{address} = T(i) * C + U_{T(i)j} \\ &\quad\quad \text{if address } \le N-1, \text{ output address.} \end{aligned}$$

We used this faster method in our implementations. Another thing worthy to be noticed is: after the change of permutation order, the calculation of U_{ij} becomes computing $U_{T(i)j}$. According to (6), we have

$$\begin{aligned} U_{T(i)j} &= S((j * r_{T(i)}) \bmod (P-1)) \\ &= S((j * Q_i) \bmod (P-1)). \end{aligned} \tag{7}$$

Therefore, another benefit of our transformation is: we avoid the step of computing the r array, which saves both the computation time and the RAM needed to store the r values. It should be mentioned that the processor-based turbo interleaver design in [6] changed the permutation order in a similar way to reduce complexity.

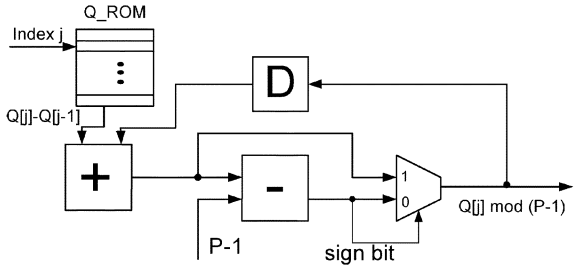

Fig. 3. Computation of $Q[j] \bmod (P-1)$.

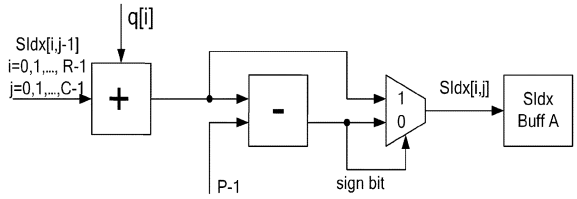

Fig. 4. Computation of the index of S array.

B. Computation of Index to S Array

Notice in (7), to compute U, $S\text{Idx} = (j * Q_i) \bmod (P-1)$ is needed, here $S\text{Idx}$ denotes S index. Fig. 4 demonstrates the circuitry we used to compute S index. To avoid the multiplication, we recursively calculate $S\text{Idx}(i,j)$ from $S\text{Idx}(i,j-1)$

$$\begin{aligned} S\text{Idx}_{i,j} &= (j * Q_i) \bmod (P-1) \\ &= (S\text{Idx}_{i,j-1} + Q_i) \bmod (P-1). \end{aligned} \tag{8}$$

Thus, from this equation, the values of Q array need to modulo $P-1$ before they are saved and used. This is the reason why we design the circuit in Fig. 3 to compute Q. It should be pointed out that similar incremental computation to (8) has been seen in [6]. However, this part of work was independently developed by the first author.

VI. VLSI Implementation

A. State Diagram

Please refer to Fig. 5.

B. Overall Block Diagram

The overall block diagram of the turbo interleaver address generator is shown in Fig. 6, where N is the turbo code block size. Task_begin signal indicates the start of computation, and task_kill signal forces the task to stop and return to the idle

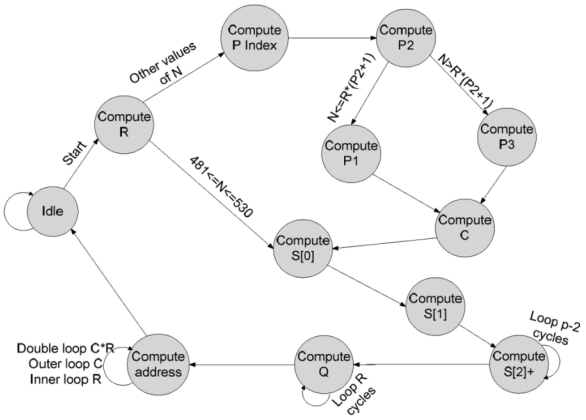

Fig. 5. State graph for the interleave address generator.

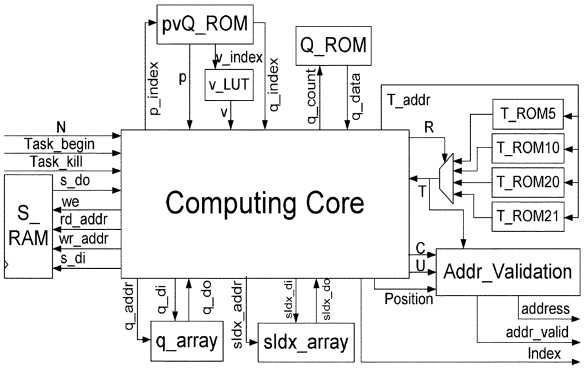

Fig. 6. Overall block diagram of the interleave address generator.

TABLE II
AREA COMPARISON OF INTERLEAVER IMPLEMENTATIONS

Implementation	*Size*	*Area(mm^2)*
ROM	>100Mb	>30.0
Design in [6] (0.25μm)	~32K gates	2.678
Design in [5] (0.25μm)	~30K gates	2.54
Our Design (0.18μm)	~4.0K gates	0.24

state. Signal "address" is the interleaved address output. "address_valid" indicates whether the output address comes from dummy bit or not. "Index" stands for the address of which input bit is calculated. For example, $\text{index} = 0, \text{address} = 33$ denotes that the first bit will be interleaved to the 34th position. S_RAM, Q_array, and SIdx_array are RAMs used to store S array, Q array, and $S\text{Index}$ array, respectively. T_ROM stores inter-row permutation pattern T. pvQ_ROM stores values of P, v, and void index of Q array. Q_ROM stores the 22 differential prime number sequences. The Computing Core contains the main finite state machine and other combinational and sequential logics to conduct control and computation.

C. Implementation Results

Please refer to Tables II and III. The architectures discussed above have been modeled using Verilog hardware description language. We have simulated and verified the design logic by comparing the output results to the C program. We have performed synthesis, optimization and place & route. The synthesis was targeted at SMIC 0.18-μm standard CMOS technology. The optimization goal was set as area. The total hardware cost is approximately 4.03 gates. The maximum clock frequency is 130 MHz. The required clock frequency is $2\ \text{M} * 6 * 2 = 24$ MHz, where we assume six iterations are performed for turbo decoding. This means that the real critical path in our design is significantly shorter than required. Thus, we can use significantly lower supply voltage to drive the circuit in order to quadratically reduce the power consumption. In brief, compared to the designs presented in [5] and [6], the proposed design is an order of magnitude more efficient in both area and power.

TABLE III
CLOCK CYCLES COUNTS FOR DIFFERENT BLOCK SIZES

Block Size	Pre-computation	On-line Computation	Bit/cycle
40	20	40	1
41	23	50	0.8
500	68	530	0.943
5,040	282	5,040	1
5,114	290	5,120	0.9988

VII. CONCLUSION

In this brief, we have presented a novel hardware interleaver architecture and implementation for 3 G WCDMA system. Various optimization techniques, specifically judicious algorithmic transformations and novel VLSI architectures, have been introduced and applied to this design. The implementation results demonstrate the benefits of these techniques, and show this design is an order of magnitude more efficient than the prior arts.

REFERENCES

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," in *Proc. IEEE ICC'93*, May 1993, vol. 2, pp. 1064–1070.
[2] *Technical Specification Group Radio Access Network; Multiplexing and Channel Coding (FDD)*, 3GPP TS25.212 v5.1.0, 3rd Generation Partnership Project, 2002.
[3] *Physical Layer Standard for CDMA2000 Spread Spectrum Systems*, 3GPP2 C.S0002-C, v1.0, 3rd Generation Partnership Project 2, 2002.
[4] Z. Wang, H. Suzuki, and K. Parhi, "VLSI implementation issues of turbo decoder design for wireless applications," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, 1999, pp. 503–512.
[5] P. Ampadu and K. Kornegay, "An efficient hardware interleaver for 3 G turbo decoding," *Proc. RAWCON'03*, pp. 199–201, Aug. 2003.
[6] M. Shin and I.-C. Park, "Processor-based turbo interleaver for multiple thrid-generation wireless standards," *IEEE Commun. Lett.*, vol. 7, no. 5, pp. 210–12, May 2003.
[7] K. Parhi, "Approaches to low-power implementations of DSP systems," *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, vol. 48, no. 10, pp. 1214–1224, Oct. 2001.