A Table-Based Algorithm for Pipelined CRC Calculation

Yan Sun and Min Sik Kim
School of Electrical Engineering and Computer Science
Washington State University
Pullman, Washington 99164-2752, U.S.A.
Email: {ysun,msk}@eecs.wsu.edu

***Abstract*—In this paper, we present a fast cyclic redundancy check (CRC) algorithm that performs CRC computation for any length of message in parallel. For a given message with any length, the algorithm first chunks the message into blocks, each of which has a fixed size equal to the degree of the generator polynomial. Then it performs CRC computation using only lookup tables among the chunked blocks in parallel and the results are combined together by XOR operations. It was feedback in the traditional implementation that makes pipelining problematic. In the proposed algorithm, we solve this problem and implement a pipelined calculation of 32-bit CRC in SMIC 0.13 μm CMOS technology. Our algorithm allows calculation over data that is not the full width of the input. Furthermore, the pipeline latency is very short in our algorithm, and this method allows easy scaling of the parallelism while only slightly affecting timing. The simulation results show that our proposed pipelined CRC is more efficient than the current CRC implementations.**

I. Introduction

A CRC (Cyclic Redundancy Check) [1] is a popular error-detecting code computed through binary polynomial division. To generate a CRC, the sender treats binary data as a binary polynomial and performs the modulo-2 division of the polynomial by a standard generator (e.g., CRC-32 [2]). The remainder of this division becomes the CRC of the data, and it is attached to the original data and transmitted to the receiver. Receiving the data and CRC, the receiver also performs the modulo-2 division with the received data and the same generator polynomial. Errors are detected by comparing the computed CRC with the received one. The CRC algorithm only adds a small number of bits (32 bits in the case of CRC-32) to the message regardless of the length of the original data, and shows good performance in detecting a single error as well as an error burst. Because the CRC algorithm is good at detecting errors and is simple to implement in hardware, CRCs are widely used today for detecting corruption in digital data which may have occurred during production, transmission, or storage. And CRCs have recently found a new application in universal mobile telecommunications system standard for message length detection of variable-length message communications [3].

Communication networks use protocols with ever increasing demands on speed. The increasing performance needs can be fulfilled by using ASICs (Application Specific Integrated Circuits) and this will probably also be the case in the future. Meeting the speed requirement is crucial because packets will be dropped if processing is not completed at wire speed. Recently, as the high throughput required protocols emerged, such as IEEE 802.11n WLAN and UWB (Ultra Wide Band), new protocols with much higher throughput requirement are on the way. For example, 10 Gbps IEEE 802.3ak was standardized in 2003, and now work has begun on defining 100 Gbps IEEE 802.3. So how to meet these requirement becomes a more important issue. CRC is used in most communication protocols as an efficient way to detect transmission errors, and thus high-speed CRC calculation is also demanded. In order to support these high throughput CRC at a reasonable frequency, processing multiple bits in parallel and pipelining the processing path are desirable.

Traditionally, the LFSR (Linear Feedback Shift Register) circuit is implemented in VLSI (Very-Large-Scale Integration) to perform CRC calculation which can only process one bit per cycle. Recently, parallelism in the CRC calculation becomes popular, and typically one byte or multiple bytes can be processed in parallel. A common method used to achieve parallelism is to unroll the serial implementation. Unfortunately, the algorithms used for parallelism increase the length of the worst case timing path, which falls short of ideal speedups in practice. Furthermore, the required area and power consumption increases with the higher degree of parallelism. Therefore, we seek an alternative way to implement CRC hardware to speed up the CRC calculation while maintaining reasonable area and power consumption.

In summary, this paper proposes a hardware architecture for calculating CRC that offers a number of benefits. First of all, it calculates the CRC of a message in parallel to achieve better throughput. It does not use LFSRs and does not need to know the total length of the message before beginning the CRC calculation. While the algorithm is based on lookup tables, it adopts multiple small tables instead of a single large table so that the overall required area remains small.

The remainder of this paper is structured as follows. Section II surveys related work on CRC algorithms. Section III details the design of proposed CRC algorithm. Section IV describes the implementation and evaluates the performance of our algorithm. Section V concludes the paper.

II. Related Work

In traditional hardware implementations, a simple circuit based on shift registers performs the CRC calculation by handling the message one bit at a time [4]. A typical serial CRC using LFSRs is shown in Fig. 1.

Fig. 1 illustrates one possible structure for CRC32. There are a total of 32 registers; the middle ones are left out for brevity. The combinational logic operation in the figure is the XOR operation. One bit is shifted in at each clock pulse. This circuit operates in a fashion similar to manual long division. The XOR gates in Fig. 1 hold the coefficients of the divisor corresponding to the indicated powers of x. Although the shift register approach to computing CRCs is usually implemented in hardware, this algorithm can also be used in software when bit-by-bit processing is adequate.

Today's applications need faster processing speed and there has been much work on parallelizing CRC calculation. Cheng and Parhi discussed unfolding the serial implementation and combined it with pipelining and retiming algorithms to increase the speed [5]. The parallel long Bose-Chaudhuri-Hocquenghen (BCH) encoders are based on the multiplication and division operations on the generator polynomial, and they are efficient to speed up the parallel linear feedback shift register (LFSR) calculation [6], [7]. Unfortunately, the implementation cost is rather high. Another approach to unroll the serial implementation was proposed by Campobello et al. [8] using linear systems theory. This algorithm is, however, based on the assumption that the packet size is a multiple of the CRC input size. Satran and Sheinwald proposed an incremental algorithm in which CRC is computed on out-of-order fragments of a message [9]. A CRC is computed incrementally and each arriving segment contributes its share to the message's CRC upon arrival, independently of other segment arrivals, and can thus proceed immediately to the upper layer protocol. A number of software-based algorithms have also been proposed [10], [11], [12], [13]. [12] proposed a scheme to calculate CRC in parallel and the idea has been widely used today. Walma designed a hardware-based approach in [14], which focus on the pipelining of the CRC calculation to increase the throughput.

In this paper, we investigate the implementation of the pipelined CRC generation algorithm using lookup tables. Our intent is not to invent new error detection codes but to come up with new techniques for accelerating codes that are well-known and in use today.

III. Proposed CRC Algorithm

A. Properties of CRC

When data are stored on or communicated through media that may introduce errors, some form of error detection or error detection and correction coding is usually employed. Mathematically, a CRC is computed for a fixed-length message by treating the message as a string of binary coefficients of a polynomial, which is divided by a generator polynomial, with the remainder of this division used as the CRC. For an l-bit message, $a_{l-1}a_{l-2}\dots a_0$, we can express it as a polynomial as follows:

$$A(x) = a_{l-1}x^{l-1} + a_{l-2}x^{l-2} + a_{l-3}x^{l-3} + ... + a_0 \quad (1)$$

where a_{l-1} is the MSB (Most Significant Bit) and a_0 is the LSB (Least Significant Bit) of the message. Given the degree-m generator polynomial,

$$G(x) = g_m x^m + g_{m-1}x^{m-1} + g_{m-2}x^{m-2} + \ldots + g_0 \quad (2)$$

where $g_m = 1$ and $g_i \in \{0, 1\}$ for all i, $A(x)$ is multiplied by x^m and divided by $G(x)$ to find the remainder. The CRC of the message is defined as the coefficients of that remainder polynomial. Namely, the polynomial representation of the CRC is

$$CRC[A(x)] = A(x)x^m \bmod G(x) \quad (3)$$

using polynomial arithmetic in the Galois field of two elements, or GF(2). After CRC processing is completed, the CRC is affixed to the original message and sent through a channel (e.g., stored on a disk and subsequently retrieved, or received from a communication channel). The presence of errors may be detected by recomputing the CRC using the received message including CRC and verifying the newly computed CRC by the same generator $G(x)$. If the computed CRC does not match (the remainder is not zero), one or more errors have been introduced by the channel. If the computed CRC matches (the remainder is zero), the received message is assumed to be error-free, although there is a small probability that undetected errors have occurred. For a suitably chosen $G(x)$, the probability of undetected error is approximately 2^{-m}.

For our parallel CRC design, the serial computation demonstrated above should be rearranged into a parallel configuration. We use the following two theorems to achieve parallelism in CRC computation.

Theorem 1: Let $A(x) = A_1(x) + A_2(x) + \ldots + A_N(x)$ over GF(2). Given a generator polynomial $G(x)$, $CRC[A(x)] = \sum_{i=1}^{N} CRC[A_i(x)]$.

Theorem 2: Given $B(x)$, a polynomial over GF(2), $CRC[x^k B(x)] = CRC[x^k CRC[B(x)]]$ for any k.

Both theorems can be easily proved using the properties of GF(2).

Theorem 1 implies that we can split a message $A(x)$ into multiple blocks, $A_1(x)$, $A_2(x)$, ..., $A_N(x)$, and compute each block's CRC independently. For example, suppose that $A(x) = a_{l-1}x^{l-1} + a_{l-2}x^{l-2} + a_{l-3}x^{l-3} + ... + a_0$ represents a l-bit message, and that it is split into b-bit blocks. For simplicity, let's assume that l is a multiple of b, namely $l = Nb$. Then the ith block of the message is a b-bit binary string from the $(i-1)b + 1$st bit to the ibth bit in the original message, followed by $l - ib$ zeros, and thus we get $A_i(x) = \sum_{k=l-ib}^{l-(i-1)b-1} a_k x^k$. Fig. 2 shows this case when each block is n byte long, or $b = 8n$.

Theorem 2 is critical in the pipelined calculation of $CRC[A_i(x)]$. As shown in Fig. 2, $A_i(x)$ has many trailing zeros, and its number determines the order of $A_i(x)$. It means

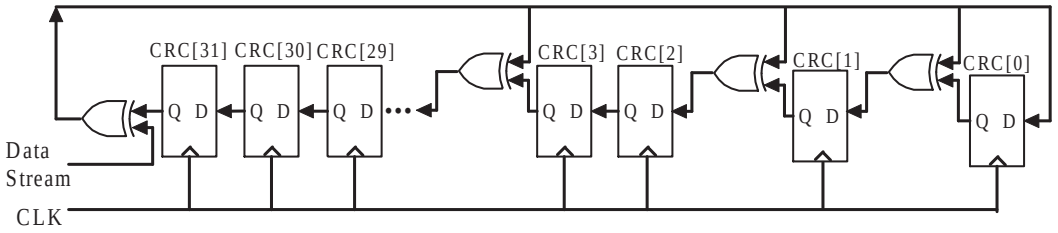

Fig. 1. Serial CRC circuit using LFSRs

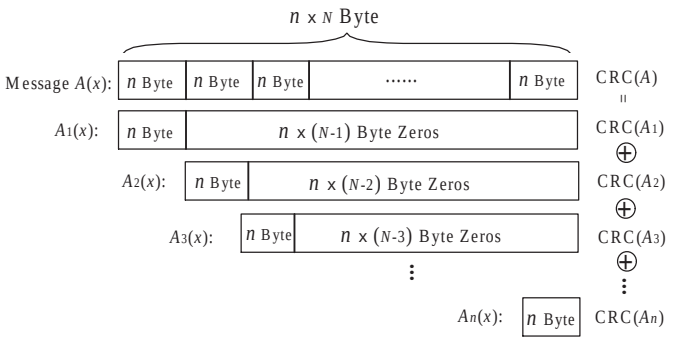

Fig. 2. Alternative CRC calculation

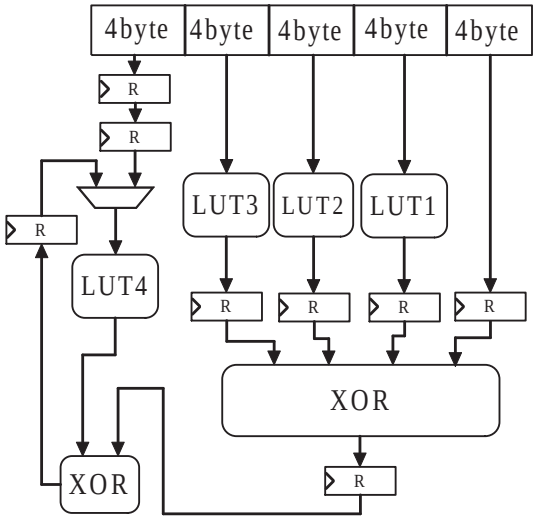

Fig. 3. Proposed pipelined CRC architecture

that the length of the message should be known beforehand to have correct $A_i(x)$. Thanks to Theorem 2, however, we can compute the CRC of the prefix of $A_i(x)$ including the b-bit substring of the original message, and then update the CRC later when the number of following zeros is known.

B. Pipelined Architecture

Based on the discussion in Section III-A, we design our CRC algorithm as follows.

The first step is to divide a message into a series of n-byte blocks. A single block becomes a basic unit in parallel processing. To make description simple, we assume that the message length is the multiple of n so that every block has exactly n bytes. We will later address the case where the last block is shorter than n bytes.

Ideally, all blocks could be processed in parallel and the results can be combined to get the CRC of the whole message, as shown in Fig. 2. However, this is impractical because the number of blocks may be unknown and the number of parallel CRC calculations is limited by available hardware. Suppose that we can perform four CRC calculations simultaneously. Then we can use the method in Fig. 2 to compute the CRC of the first four blocks; $A_1(x)$ becomes the first block followed by $3n$ bytes of zeros, $A_2(x)$ the second block followed by $2n$ bytes of zeros, A_3 the third block followed by n bytes of zeros, and A_4 the fourth block itself. Combining every CRC using XOR results in the CRC for the first n bytes of the message. This step of processing the first four blocks and getting the CRC is the first iteration. Let's call this CRC CRC_1. In the second iteration, the next four blocks are processed to produce their combined CRC. Note that this result combined with $CRC[x^{4\cdot8\cdot n}\,CRC_1]$ is the CRC for the first eight blocks because of Theorem 2. We can continue until we reach the end of the message. It is implemented in Fig. 3.

It has five blocks as input; four of them are used to read four new blocks from the message in each iteration. They are converted into CRC using lookup tables: LUT3, LUT2, and LUT1. LUT3 contains CRC values for the input followed by 12 bytes of zeros, LUT2 8 bytes, and LUT4 4 bytes. Note that the rightmost block doesn't have any lookup table. It is because this architecture assumes CRC-32, the most popular CRC, and 4-byte blocks. If the length of a binary string is smaller than the order of the CRC generator, its CRC value is the string itself. Since the rightmost block corresponds to A_4, it doesn't need any following zero and thus its CRC is the block itself. The results are combined using XOR, and then it is combined with the output of LUT4, the CRC of the value from the previous iteration with 16 bytes of zeros appended. In order to reduce the critical path, we introduce another stage called the pre-XOR stage right before the four-input XOR gate. This makes the algorithm more scalable because more blocks can be added without increasing the critical path of the pipeline. With the pre-XOR stage, the critical path is the delay of LUT4 and a two-input XOR gate, and the throughput is 16 bytes per cycle.

The architecture in Fig. 3 shows further optimization: the

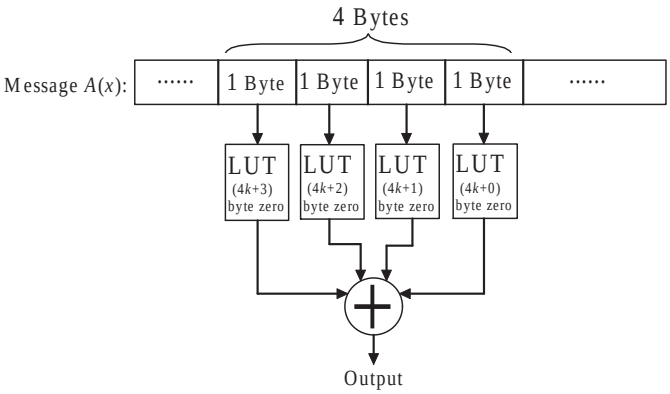

Fig. 4. Zero block lookup tables

TABLE I
COMPARISON BETWEEN OUR ALGORITHM AND EXISTING ALGORITHMS

Algorithms	Maximum Frequency (MHz)	Area	Throughput (Gbps)
Ordinary CRC algorithm in [12]	347.8	3576	20.68
Pipelined CRC algorithm in [14]	561.1	16494	35.28
Proposed algorithm	878.3	14587	56.21

leftmost 4-byte block. Since the CRC of the first block is the first block itself, it can be easily combined with the following four blocks by appending 16 bytes of zeros using LUT4. To exploit this, the first iteration loads the first five blocks from the message. The multiplexer is set to choose the leftmost block. In this way, the CRC for five blocks is calculated in the first iteration. Two registers below the leftmost block are needed to delay for two clock cycles while other blocks' CRCs are combined. From the second iteration, four blocks from the message are loaded, and the multiplexer chooses the result from the previous iteration.

C. Lookup Table

We need to calculate the CRC of every block followed by zeros, whose length varies from 4-byte to 16-byte in our example. For faster calculation, our algorithm employs lookup tables which contain pre-calculated CRCs. Note that we need to calculate CRCs for a 4-byte block followed by 16 bytes, 12 bytes, 8 bytes, and 4 bytes of zeros. Thus, we need 4 lookup tables. The key point is that although the input length may be as long as 20 bytes, only the first 4 bytes need actual calculation because of Theorem 1. When implemented using a lookup table, however, the CRC for 4 bytes still requires as many as 2^{32} entries in the table. Instead, we maintain four small lookup tables as shown in Fig. 4, where each table handles one byte. The table size is 1K bytes, containing 2^8 entries with 32 bits each.

Each lookup table should pre-compute the CRC of each byte followed by a different number of zeros. For the kth block in one iteration, there should be 4k byte zeros should be added in addition as shown in Fig. 4, where $k = 0, 1, 2, 3$. The outputs of lookups should also be combined together using XOR.

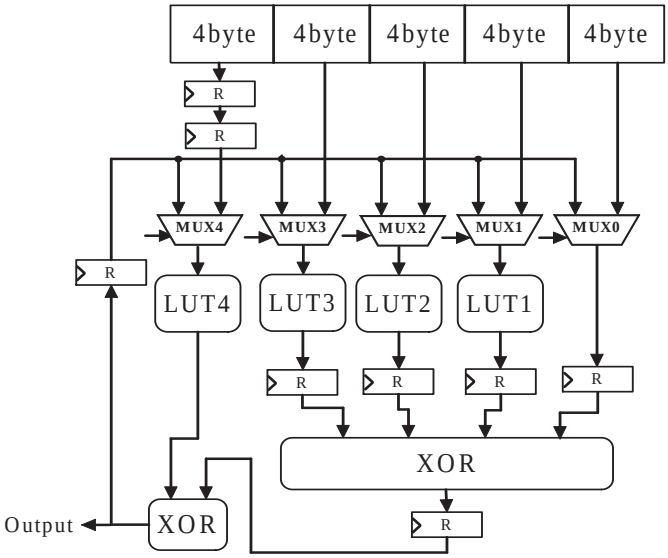

Fig. 5. Pipelined CRC architecture for unknown packet size in advance

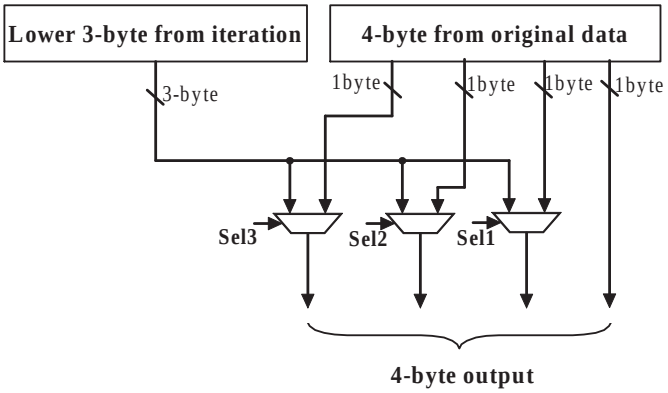

Fig. 6. Structure of MUX0

D. Last Block Size

When the message size is not a multiple of $4n$, the number of bytes processed in each iteration, the architecture in Fig. 3 fails to calculate the correct CRC. If the last block has less than $4n$ bytes, only those bytes should be loaded instead of $4n$ bytes. They occupy the lower bottom of the input in the last iteration. The remaining input up to 4 bytes are filled with the CRC from the previous iteration, and the rest with zero. To implement this, we introduce four multiplexers in Fig. 5, in addition to the one for LUT4. Their structures are shown in Fig. 6 and Fig. 7.

Since the last block size is between 1 and 16, it can be encoded with four bits, 0000 being size 1, 0001 being 2, and so on. Let this 4-bit encoding be $w = w_3w_2w_1w_0$. In the last iteration, 16 bytes consisting of the last block preceded by zeros are loaded. Then the multiplexers select the value from the previous iteration depending on w. For example, the rightmost multiplexer in Fig. 6 must select a byte from the previous iteration (the last byte) when the last block size of 1, namely $w = 0000$. Otherwise it must select the last block.

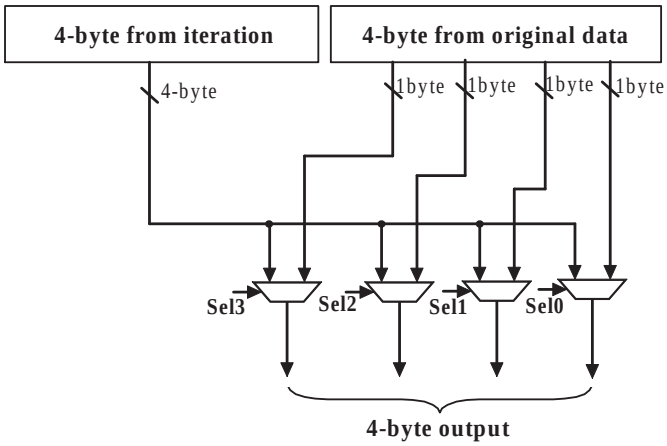

Fig. 7. Structure of MUX1 $\sim$ MUX4

Similarly, the middle multiplexer in Fig. 6 must select the previous iteration when the last block size is 1 or 2. In general, the kth rightmost multiplexer must select the previous iteration if the last block size is between $k-3$ and k.

Before the last iteration, all the multiplexers in MUX0 $\sim$ MUX3 select original data, and all the multiplexers in MUX4 select the last iteration.

IV. Evaluation

In this section, we need to evaluate the performance of our proposed algorithm, including the throughput, memory consumption and other logic usage. Furthermore, we need to test the correctness of the proposed algorithm.

A Verilog implementation of the proposed algorithm has been created for CRC32. We compared our algorithm with a typical parallel CRC algorithm in [12] and a pipelined CRC algorithm in [14]. All of the algorithms were implemented with 1.2 V power supply using the SMIC 0.13 μm CMOS technology.

In Table II, we present the amount of cache memory, critical path, and throughput as we vary the degree of parallelism, the number of blocks that can be processed in parallel. As expected, the cache memory size and throughput are proportional to the parallelism, but the critical path remains same.

TABLE II
Proposed pipelined CRC

Parallelism	Lookup table (KB)	Critical path	Throughput (bytes/cycle)
2	8	# + 5 XOR	8
4	16	# + 5 XOR	16
8	32	# + 5 XOR	32
16	64	# + 5 XOR	64

The synthesis results of the three algorithms are shown in Table I when the size of input is 64-bit. Obviously, pipelining or parallel approaches increase throughput at the cost of space. Still, our algorithm achieves 60% more throughput than the previous pipelined algorithm while occupying less area. Another advantage compared with [14] is that our algorithm does not require LFSR logic or inversion operations.

We compared the outputs of three different algorithms with the same input cases to test the correctness of our proposed algorithm, and we focused on the special cases which usually contain errors. The simulation results showed that our algorithm can work correctly.

V. Conclusion

We presented a fast cyclic redundancy check (CRC) algorithm that using lookup tables instead of linear feedback shift registers. The algorithm implements a pipeline architecture and performs CRC computation for any length of message in parallel. Given more space, it achieves considerably higher throughput than existing serial or byte-wise lookup CRC algorithms. Its throughput is also higher than the previous pipelined approach while consuming less space. With little delay increase in the critical path, the throughput can be improved further by increasing parallelism.

References

[1] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, Jan. 1961.
[2] K. Brayer and J. J. L. Hammond, "Evaluation of error detection polynomial performance on the AUTOVON channel," in *Conference Record of National Telecommunications Conference*, vol. 1, 1975, pp. 8–21 to 8–25.
[3] S. L. Shieh, P. N. Chen, and Y. S. Han, "Flip CRC modification for message length detection," *IEEE Transactions on Communications*, vol. 55, no. 9, pp. 1747–1756, 2007.
[4] T. V. Ramabadran and S. S. Gaitonde, "A tutorial on CRC computations," *IEEE Micro*, vol. 8, no. 4, pp. 62–75, 1988.
[5] C. Cheng and K. K. Parhi, "High-speed parallel CRC implementation based on unfolding, pipelining, and retiming," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 53, no. 10, pp. 1017–1021, 2006.
[6] X. Zhang and K. K. Parhi, "High-speed architectures for parallel long bch encoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 7, pp. 872–877, 2005.
[7] K. K. Parhi, "Eliminating the fanout bottleneck in parallel long bch encoders," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 3, pp. 512–516, 2004.
[8] G. Campobello, G. Patane, and M. Russo, "Parallel CRC realization," *IEEE Transactions on Computers*, vol. 52, no. 10, pp. 1312–1319, 2003.
[9] J. Satran, D. Sheinwald, and I. Shimony, "Out of order incremental CRC computation," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1178–1181, 2005.
[10] D. Feldmeier, "Fast software implementation of error detection codes," *IEEE/ACM Transactions on Networking (TON)*, vol. 3, no. 6, pp. 640–651, 1995.
[11] S. Joshi, P. Dubey, and M. Kaplan, "A new parallel algorithm for CRC generation," in *ICC2000: Proceedings of IEEE International Conference on Communications*, 2000, pp. 1764–1768.
[12] A. Simionescu, "CRC tool computing CRC in parallel for ethernet," in *http://www.nobugconsulting.ro/crcldetails.htm.*, 2001, pp. 1–5.
[13] M.E.Kounavis and F.L.Berry, "Novel table lookup-based algorithms for high-performance CRC generation," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1550–1560, 2008.
[14] M. Walma, "Pipelined cyclic redundancy check (CRC) calculation," in *ICCCN'07: Proceedings of 16th International Conference on Computer Communications and Networks*, 2007, pp. 365–370.