High Performance 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 an arbitrary 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 computes CRC for the chunked blocks in parallel using lookup tables, and the results are combined together with XOR operations. In the traditional implementation, it is the feedback that makes pipelining problematic. In the proposed algorithm, we solve this problem by taking advantage of CRC's properties and pipelining the feedback loop. The short pipeline latency of our algorithm enables a faster clock frequency than previous approaches, and it also allows easy scaling of the parallelism while only slightly affecting timing. We build a Verilog implementation our pipelined CRC. The simulation results show that our approach is faster and more space-efficient than previous CRC implementations; with 256 bits input each clock cycle, our approach achieves about 36% throughput improvement with about 12% area reduction.**

***Index Terms*—Keywords: CRC, lookup table, pipelining**

I. Introduction

A CRC (Cyclic Redundancy Check) [2] 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 [3]). 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. They are also used in the universal mobile telecommunications system standard for message length detection of variable-length message communications [4], [5].

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, protocols for high throughput have emerged, such as IEEE 802.11n WLAN and UWB (Ultra Wide Band), and new protocols with even 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. Thus, how to meet these requirements becomes more and more crucial. 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 the area and power consumption requirements at a reasonable level.

In summary, this paper proposes a table-based 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

A preliminary version of this paper was presented at IEEE International Conference on Communications, May 2010 [1].

©2013 Engineering and Technology Publishing
doi: 10.12720/jcm.8.2.128-135

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 implementations. Section III describes the properties of CRC that are useful in parallelizing its operation. Section IV details the design of proposed CRC algorithm. Section V describes the implementation and evaluates the performance of our algorithm. Section VI 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 [6]. A typical serial CRC circuit 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 data 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 [7]. 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 [8], [9]. Unfortunately, the implementation cost is rather high because of the complexity of multiplication and division operations. Another approach to unroll the serial implementation was proposed by Campobello et al. [10] 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 [11]. In their algorithm, a CRC is computed incrementally and each arriving segment contributes its share to the message's CRC upon arrival, independently of other segments' arrivals, and can thus proceed immediately to the upper layer protocol. A number of software-based algorithms have also been proposed [12]–[15], as well as FPGA-based approaches [16]–[18]. Simionescu proposed a scheme to calculate CRC in parallel [14] and the idea has been widely used. Walma designed a hardware-based approach [19], where he compared the area and throughput of pipelined and non-pipelined CRC designs and proposed a pipelined CRC calculation to increase the throughput. Our approach is also based on a pipelined parallel CRC calculation, and makes further improvement by utilizing lookup tables efficiently in the pipelined parallel CRC design.

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. 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,

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

with $g_m = 1$ 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), $CRC[x^k B(x)] = x^k\, CRC[B(x)] \bmod G(X)$ for any k.

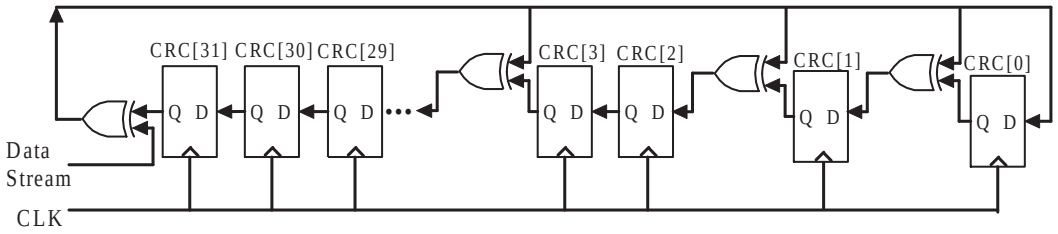


Fig. 1. Serial CRC circuit using LFSRs

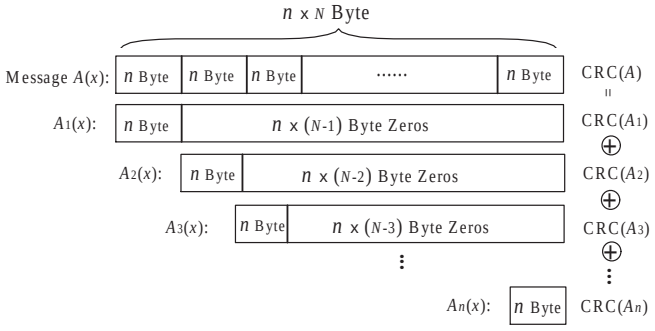


Fig. 2. Alternative CRC calculation

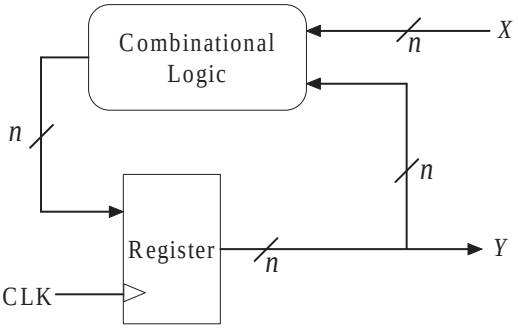


Fig. 3. Parallel CRC structure

Both theorems can be easily proved using the properties of GF(2). See Appendixes for the proofs.

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 bytes 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 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 $A_i(x)$ containing the b-bit substring of the original message, and then update the CRC later when the number of following zeros is known.

IV. Proposed Architecture

To solve the problem of the parallel calculation of CRC, we first try to simulate the behavior of a serial feedback shift register in Fig. 1 with a parallel finite state machine shown in Fig. 3.

At every clock pulse, the parallel machine receives n input bits at a time and must return the updated CRC value. The length of the data may not be known in advance. In Fig. 3, X represents the next n input bits of the message, and Y is the current CRC value, which is calculated with the data bits preceding X. The combinational logic must produce the CRC value for the all data bits up to X, inclusively, from the new input bits X and the current CRC value Y. In other words, the output of the combinational logic is a function of X and Y only. If Y is the current content of the serial feedback shift register in Fig. 1 and X is the next n input bits, the output of the combinational logic must be the same as the content of the shift register after n clocks.

Below we describe our pipelined CRC algorithm based on this parallel structure and the properties of CRC discussed in Section III.

A. Pipelining

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. We will address the case with a message of an arbitrary length in Section IV-C.

Ideally, all blocks could be processed in parallel and the results could 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 m CRC calculations simultaneously, with $m < N$. Then we can use the method in Fig. 2 to compute the CRC of the first m blocks; $A_1(x)$ becomes the first block followed by $n(m-1)$ bytes of zeros, $A_2(x)$ the second block followed by $n(m-2)$ bytes of zeros, and so on, and A_m is the mth block itself. Combining every CRC using XOR results in the CRC for the first nm bytes of the message. This step of processing the first m blocks and getting the CRC is

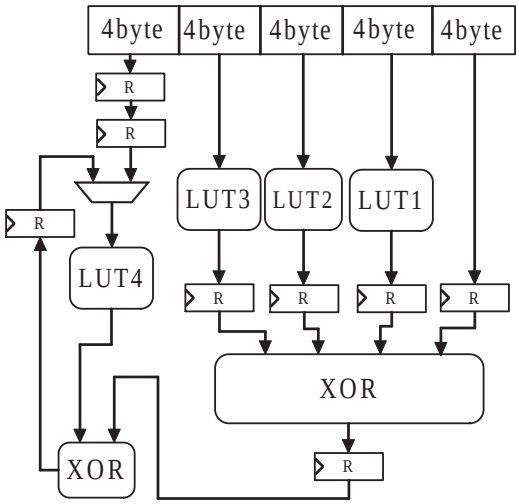


Fig. 4. Proposed pipelined CRC architecture

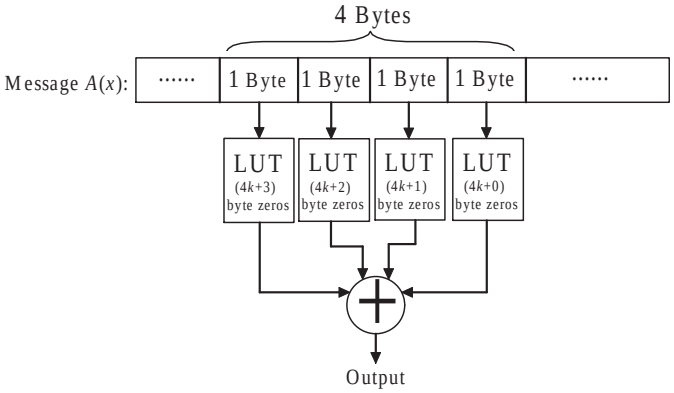


Fig. 5. Small lookup tables to construct LUTk in Fig. 4

the first iteration. Let's call this CRC value CRC_1. In the second iteration, the next m blocks are processed to yield CRC_2, the CRC value of those blocks. Then, by Theorem 2, $CRC[x^{8nm}\,CRC_1]$ is the CRC value of the first m bytes followed by nm bytes of zeros, and thus $CRC[x^{8nm}\,CRC_1] \oplus CRC_2$ is the CRC value of the first $2m$ blocks. We can continue iterations until we reach the end of the message. This implementation with $m = 4$ is shown in Fig. 4.

The pipelined architecture in Fig. 4 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 LUT1 4 bytes. Note that the rightmost block does not need 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 degree of the CRC generator, its CRC value is the string itself. Since the rightmost block corresponds to A_4, it does not have 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 concatenated. In order to shorten 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. 4 also shows further optimization: the 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 property, 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. This additional logic is especially useful when there are a large number of short messages and each of them needs its own CRC.

B. Lookup Tables

The architecture in Fig. 4 calculates the CRC of every block followed by zeros, whose length varies from 4 bytes to 16 bytes in our example. For faster calculation, our algorithm employs lookup tables which contain pre-calculated CRCs. Note that we need four different lookup tables so as to calculate CRCs for 4-byte blocks followed by 16 bytes, 12 bytes, 8 bytes, and 4 bytes of zeros. Note that, although the input length may be as long as 20 bytes in the case of the first block, only the first four 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. 5, where each table handles a single byte. Each small LUT in Fig. 5 contains 1 KB, or 2^8 entries with 32 bits each.

A small LUT should store the precomputed CRC of each byte followed by a different number of zeros: 0, 1, 2, and 3. For the kth lookup table LUTk in Fig. 4, where $k = 1, 2, 3, 4$, there must be additional $4k$ bytes of zeros as shown in Fig. 5. The outputs of LUTs are combined together using XOR.

Lookup tables can be replaced by XOR trees to achieve smaller area and to reduce pipeline latency. However, we do not perform this optimization in our approach based on two considerations: first, we want to make our approach more general, and our approach can be easily modified and used in CPU-based calculations, and lookup table is more suitable in this situation. Second, it is not flexible to convert lookup tables into XOR trees because we need to update the lookup tables in some applications.

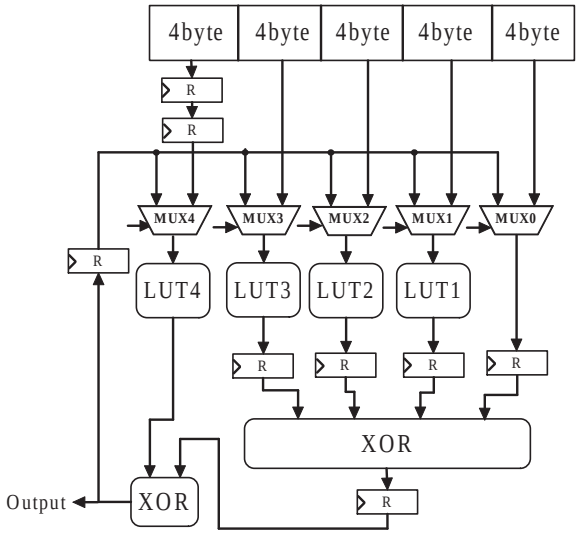


Fig. 6. Pipelined CRC architecture for unknown packet size in advance

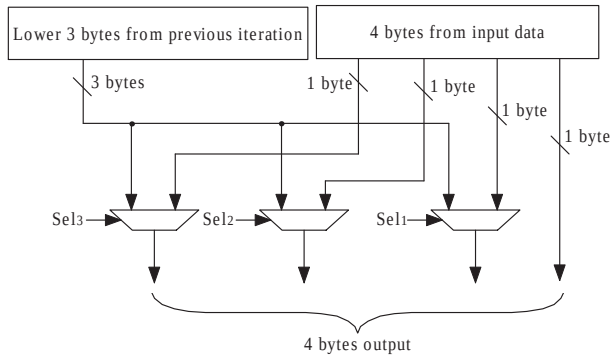


Fig. 7. Structure of MUX0

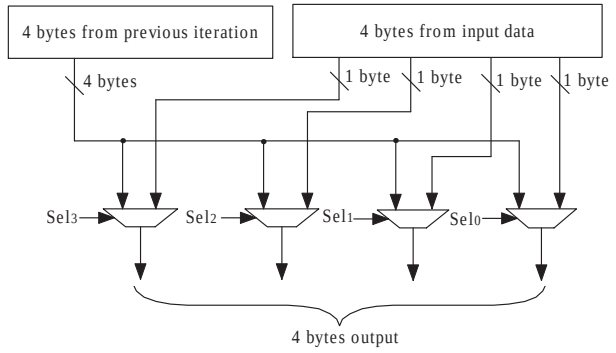


Fig. 8. Structure of MUX1–MUX3

C. Last Iteration

If there remain less than $4n$ bytes in the last iteration, the architecture in Fig. 4 fails to calculate the correct CRC. In such a case, it should load only those remaining bytes in the last iteration and no more. They occupy the rightmost bytes of the input. The remaining input up to 4 bytes should be filled with the CRC from the previous iteration, and the rest with zeros. To implement this, we introduce four multiplexers in Fig. 6, MUX0–3; MUX4 remains the same as in Fig. 4. Their structures are shown in Fig. 7 and Fig. 8.

Since the number of remaining bytes in the last iteration 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, the remaining bytes are loaded into the rightmost bytes of the input. Then the multiplexers select between the input buffer and the value from the previous iteration according to w. For example, the rightmost multiplexer must select the last three byte from the previous iteration if the number of remaining bytes is one, namely $w = 0000$. Otherwise, it must select the byte from the input buffer. To demonstrate this, the internals of MUX0 is shown in Fig. 7. Because the rightmost byte always contains a new input value, it need not go through a multiplexer. The second rightmost byte, however, has a new input value only when $w \geq 1$. Therefore, Sel1 should be set to select the input buffer if and only if $w \geq 1$. Similarly, Sel2 and Sel3 must be set to select the input buffer if and only if $w \geq 2$ and $w \geq 3$, respectively.

This needs to be slightly modified for MUX1–3, as shown in Fig. 8. The ith rightmost multiplexer in MUXk selects the new input when $w \geq 4k + i - 1$.

Before the last iteration, all the multiplexers in MUX0–3 select the new input.

V. Evaluation

To demonstrate the effectiveness of our proposed algorithm, we evaluate it and compare with previous ones using three metrics: the overall throughput, memory consumption, and logic area.

A Verilog implementation of the proposed algorithm has been created for CRC32. We have selected two previous algorithms for comparison: a typical parallel CRC algorithm [14] we refer to as the "Parallel CRC" below, and Walma's pipelined CRC algorithm [19] as the "Pipelined CRC." We select these two hardware-based approaches because they are more comparable: the "Parallel CRC" approach is commonly used parallel CRC calculation algorithm and the "Pipelined CRC" is one of the most efficient CRC calculation algorithms proposed recently. All these algorithms are implemented with 1.2 V power supply using the SMIC 0.13 μm CMOS technology. Note that the "Pipelined CRC" approach allows independent scaling of circuit frequency and data throughput by varying the data width and the number of pipeline stages. In order to compare with "Pipelined CRC", we implemented "Pipelined CRC" with the same number of stages as in our approach.

TABLE I shows general characteristics of the proposed algorithm including the amount of memory used by lookup tables, the length of the critical path, and the throughput as we vary the degree of parallelism, i.e., the number of blocks that can be processed in parallel. In the "Critical path length" column, D_{lookup} and D_{XOR} represent the delays of a single lookup table and an XOR operation, respectively. It is the multiplexer that causes the delay of $5D_{\text{XOR}}$. As expected, the lookup table size and throughput are proportional to the degree of parallelism,

TABLE I
SIMULATION RESULTS OF THE PROPOSED CRC WITH DIFFERENT DEGREES OF PARALLELISM

Parallelism	Lookup table size (KB)	Critical path length	Throughput(bytes/cycle)
2	8	$D_{\text{lookup}} + 5D_{\text{XOR}}$	8
4	16	$D_{\text{lookup}} + 5D_{\text{XOR}}$	16
8	32	$D_{\text{lookup}} + 5D_{\text{XOR}}$	32
16	64	$D_{\text{lookup}} + 5D_{\text{XOR}}$	64

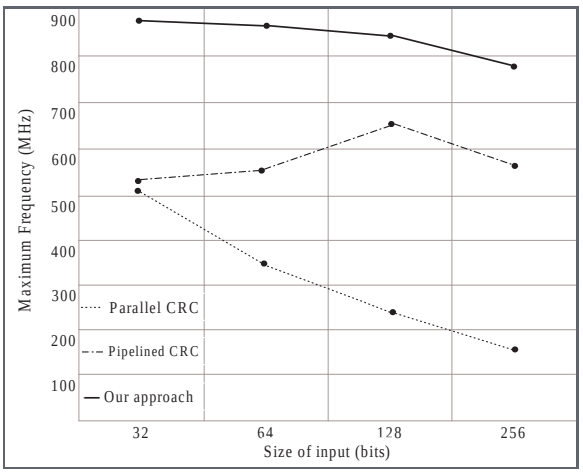


Fig. 9. The maximum frequency comparison under different input size

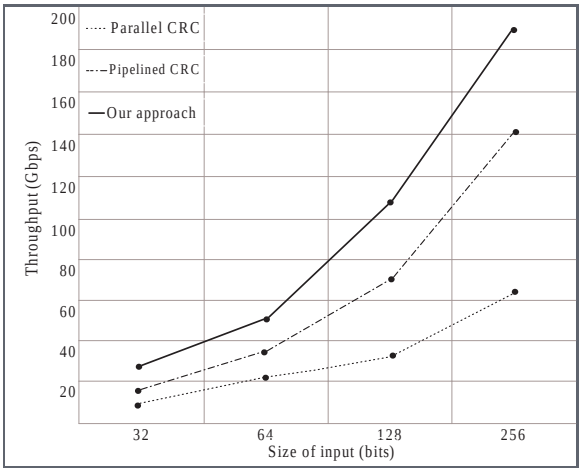


Fig. 10. The throughput comparison under different input size

while the critical path remains constant. Note that some compilers may perform optimization on lookup tables based on the content stored in the lookup tables to reduce the hardware resource. In order to make our approach more general and support different polynomial, we disable such optimizations.

Below, we present the synthesis results of the three CRC algorithms: the Parallel CRC, the Pipelined CRC, and our table-based approach.

The maximum clock frequency that each approach can achieve is shown in Fig. 9 for different input sizes between 32 bits and 256 bites. The clock frequency of the Parallel CRC drops rapidly as the input size grows, while those of the other two remain relatively constant. In all cases, our approach achieves the highest frequency,

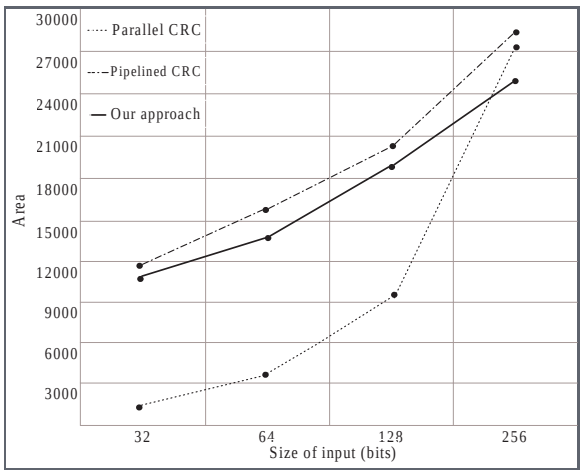


Fig. 11. The hardware area comparison under different input size

because of the short critical path.

This difference in frequencies clearly affects the throughput, as demonstrated in Fig. 10. The order of the three approaches follows the order in Fig. 9. All of them have higher throughput with a larger input size. Note that our approach constantly outperforms the Pipelined CRC by more than 35% and the Parallel CRC by 200%.

Obviously, all three approaches increase throughput at the cost of space. Fig. 11 demonstrates how much area is used by each approach. In the case of the Parallel CRC, the area is linearly proportional to the input size. On the other hand, the areas of the Pipelined CRC and our approach grow sublinearly as the input grows. With a 256-bit input size, our approach occupies a less area than the others. Based on these results, we see that our approach achieves better throughput than the previous algorithms without increasing the hardware area, especially with a large input size. Besides, another advantage over the Pipelined CRC is that our approach does not require the LFSR logic or inversion operations because all these are encoded in the lookup tables.

VI. CONCLUSION

We presented a fast cyclic redundancy check (CRC) algorithm, which uses lookup tables instead of linear feedback shift registers. The proposed algorithm is based on a pipelined 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 previous parallel or pipelined approaches while consuming less space most of the time. With little

delay increase in the critical path, the throughput can be improved further by increasing parallelism. Furthermore, our approach doesn't require the packet length initially, which makes our approach do not need to wait until it receives the whole data.

ACKNOWLEDGMENT

We thank the reviewers for their detailed comments.

REFERENCES

[1] Y. Sun and M. S. Kim, "A table-based algorithm for pipelined CRC calculation," in *Proceedings of IEEE International Conference on Communications*, May 2010.

[2] 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.

[3] 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, Dec. 1975, pp. 8–21 to 8–25.

[4] 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, Sep. 2007.

[5] Y. Sun and M. S. Kim, "A pipelined crc calculation using lookup tables," in *Proceedings of IEEE Consumer Communications and Networking Conference (CCNC)*, Jan. 2010.

[6] T. V. Ramabadran and S. S. Gaitonde, "A tutorial on CRC computations," *IEEE Micro*, vol. 8, no. 4, pp. 62–75, Aug. 1988.

[7] 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, Oct. 2006.

[8] 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, Jul. 2005.

[9] 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, Jul. 2004.

[10] G. Campobello, G. Patane, and M. Russo, "Parallel CRC realization," *IEEE Transactions on Computers*, vol. 52, no. 10, pp. 1312–1319, Oct. 2003.

[11] J. Satran, D. Sheinwald, and I. Shimony, "Out of order incremental CRC computation," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1178–1181, Sep. 2005.

[12] D. Feldmeier, "Fast software implementation of error detection codes," *IEEE/ACM Transactions on Networking*, vol. 3, no. 6, pp. 640–651, Dec. 1995.

[13] S. Joshi, P. Dubey, and M. Kaplan, "A new parallel algorithm for CRC generation," in *Proceedings of IEEE International Conference on Communications*, Jun. 2000, pp. 1764–1768.

[14] A. Simionescu, "CRC tool computing CRC in parallel for Ethernet," http://space.ednchina.com/upload/2008/8/27/5300b83c-43ea-459b-ad5c-4dc377310024.pdf, 2001.

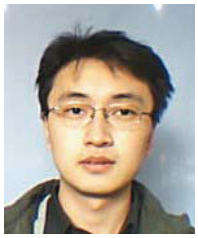
[15] M. E. Kounavis and F. L. Berry, "Novel table lookup-based algorithms for high-performance CRC generation," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1550–1560, Nov. 2008.

[16] M. Braun, J. Friedrich, T. Grn, and J. Lembert, "Parallel CRC computation in FPGAs generation," *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, vol. 1142, pp. 156–165, 1996.

[17] C. Anton, L. Ionescu, I. Tutanescu, A. Mazare, and G. Serban, "FPGA-implemented CRC algorithm," in *Proceedings of Applied Electronics*, Sep. 2007, pp. 25–29.

[18] R. Ahmad, O. Sidek, and S. Mohd, "Development of the CRC block for Zigbee standard on FPGA," in *Proceedings of International Conference for Technical Postgraduates*, Dec. 2009.

[19] M. Walma, "Pipelined cyclic redundancy check (CRC) calculation," in *Proceedings of the 16th International Conference on Computer Communications and Networks*, Aug. 2007, pp. 365–370.



Yan Sun received the B.S. degree in applied physics in 2005 from University of Science and Technology, Beijing, China, the M.S. degree in 2008 in microelectronics from the University of Science and Technology of China, Hefei, China, and the Ph.D. degree in computer science from Washington State University in 2011. He is currently a staff scientist in Broadcom. His research interests include network security, high-performance VLSI systems and computer architectures.

Min Sik Kim received the B.S. degree in computer engineering from Seoul National University, Seoul, Korea, in 1996, and the Ph.D. degree in computer science from the University of Texas at Austin in 2005. At present, he is an Assistant Professor of computer science with the School of Electrical Engineering and Computer Science, Washington State University, Pullman. Prior to joining Washington State University, he cofounded and served as Chief Technology Officer of Infnis, Inc., from 2002 to 2004. His research interests include overlay networks, network monitoring, and network traffic analysis.

1

APPENDIX A
PROOF OF THEOREM 1

Proof: In GF(2),

$$(X(x)+Y(x)) \bmod Z(x) = X(x) \bmod Z(x)+Y(x) \bmod Z(x)\,.$$

Therefore,

$$\begin{aligned} CRC[A(x)] &= A(x)x^m \bmod G(x) \\ &= (A_1(x) + A_2(x) + \ldots + A_N(x))x^m \bmod G(x) \\ &= A_1(x)x^m \bmod G(x) + A_2(x)x^m \bmod G(x) + \ldots + A_N(x)x^m \bmod G(x) \\ &= CRC[A_1(x)] + CRC[A_2(x)] + \ldots + CRC[A_N(x)]\,. \end{aligned}$$

■

APPENDIX B
PROOF OF THEOREM 2

Proof: In GF(2),

$$\begin{aligned} CRC[x^k B(x)] &= x^k B(x)x^m \bmod G(x) \\ &= x^k [B(x)x^m \bmod G(x)] \bmod G(x) \\ &= x^k\, CRC[B(x)] \bmod G(x)\,. \end{aligned}$$

■