

A Pipelined CRC Calculation Using Lookup Tables

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—We present a fast cyclic redundancy check (CRC) algorithm that performs CRC computation for any length of message in parallel. Traditional CRC implementations have feedbacks, which make pipelining problematic. In the proposed approach, we eliminate feedbacks and implement a pipelined calculation of 32-bit CRC in the SMIC 0.13 μm CMOS technology. For a given message, 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 by performing XOR operations. The simulation results show that our proposed pipelined CRC is more efficient than existing CRC implementations.

Index Terms—CRC, lookup table, pipelining

I. INTRODUCTION

Communication networks use protocols with ever increasing demands on speed. Meeting the speed requirement is crucial because packets will be dropped if the jobs are 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. In order to support CRC (Cyclic Redundancy Check) calculation for these high throughput standards at a reasonable frequency, processing multiple bits in parallel and pipelining the processing path are desirable. Although there have been algorithms for parallelism in CRC calculation in recent years, they increase the length of the worst case timing path as well as the required area and power consumption. Therefore, we seek an alternative way to implement CRC hardware to speed up the CRC calculation with reasonable area and power consumption.

II. PROPOSED CRC ALGORITHM

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

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

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

Both theorems can be easily proved using the properties of $\text{GF}(2)$.

Theorem 1 implies that we can split a message $A(x)$ into multiple blocks, $A_1(x)$, $A_2(x)$, \dots , $A_N(x)$, and compute

each block's CRC independently. For example, suppose that $A(x) = a_{l-1}x^{l-1} + a_{l-2}x^{l-2} + a_{l-3}x^{l-3} + \dots + a_0$ 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 i th block of the message is a b -bit binary string from the $(i-1)b + 1$ st bit to the ib th bit in the original message, followed by $l - ib$ zeros, and thus we get $A_i(x) = \sum_{k=l-ib}^{l-(i-1)b-1} a_k x^k$.

Theorem 2 is critical in the pipelined calculation of $\text{CRC}[A_i(x)]$. $A_i(x)$ 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 $A_i(x)$ including the b -bit substring of the original message, and then update the CRC later when the number of following zeros is known.

The first step of our algorithm 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.

In our implementation, we perform four CRC calculations simultaneously. $A_1(x)$ 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 $\text{CRC}[x^{4 \cdot 8 \cdot n} \text{CRC}_1]$ is the CRC for the first eight blocks because of Theorem 2. It is implemented in Fig. 1. 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. 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

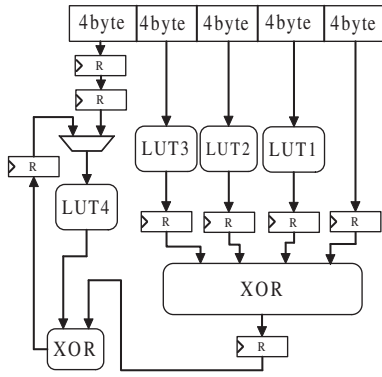


Fig. 1. Proposed Pipelined CRC Architecture

TABLE I
COMPARISON BETWEEN OUR ALGORITHM AND EXISTING ALGORITHMS

Algorithms	Frequency(MHz)	Area	Throughput(Gbps)
Algorithm in [1]	347.8	3576	20.68
Algorithm in [2]	561.1	16494	35.28
Proposed algorithm	878.3	14587	56.21

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. 1 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, 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.

We need to calculate the CRC of every block followed by zeros, whose length varies from 4-byte to 16-byte. For faster calculation, our algorithm employs lookup tables which contain pre-calculated CRCs, and 4 lookup tables are needed. 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. As an alternative, we maintain four small lookup tables, 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 k th block in one iteration, $4k$ byte zeros should be added in addition, where $k = 0, 1, 2, 3$. The outputs of lookups should also be combined together using XOR.

When the message size is not a multiple of $4n$, the number of bytes processed in each iteration, the architecture in Fig. 1 fails to calculate the correct CRC. If the last block has less

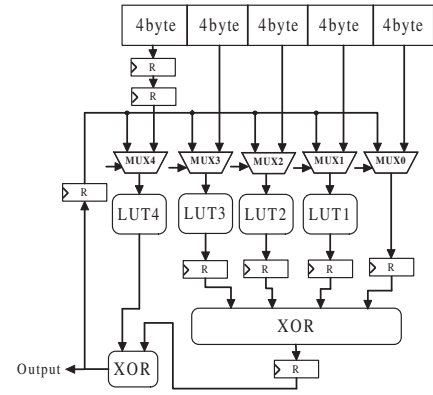


Fig. 2. Pipelined CRC Architecture with Packet Size Unknown

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. 2.

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 .

III. EVALUATION

A Verilog implementation of the proposed algorithm has been created for CRC32. We compared our algorithm with an ordinary CRC algorithm [1] and a pipelined CRC algorithm [2] when the size of input is 64 bits. Each of the algorithms was implemented with 1.2 V power supply using the SMIC 0.13 μm CMOS technology. The synthesis results are shown in Table I. 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 over the compared pipelined algorithm is that our algorithm does not require LFSR logic or inversion operations.

IV. CONCLUSION

We presented a fast CRC algorithm that uses lookup tables and implements a pipelined architecture to perform CRC computation for any length of message in parallel. Given more space, it achieves considerably higher throughput than existing 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] A. Simionescu, "CRC tool computing CRC in parallel for ethernet," in <http://www.nobugconsulting.ro/crcdetails.htm>, 2001, pp. 1–5.
- [2] M. Walma, "Pipelined cyclic redundancy check (CRC) calculation," in *ICCCN'07: Proceedings of 16th International Conference on Computer Communications and Networks*, Aug. 2007, pp. 365–370.