A Fast CRC Update Implementation

Weidong Lu and Stephan Wong
Computer Engineering Laboratory,
Electrical Engineering Department,
Delft University of Technology,
Delft, The Netherlands
{wlu, stephan}@dutepp0.tudelft.nl
http://ce.et.tudelft.nl

Abstract— **In networking environments, the cyclic redundancy check (CRC) is widely utilized to determine whether errors have been introduced during transmissions over physical links. In this paper, we focus on the CRC calculation that is performed during the routing of the Ethernet packets by encapsulating the packets into Ethernet frames, adding a frame header and adding a frame trailer. The CRC code is placed within the frame trailer. In our investigation, we observed that only specific fields located at the beginning of a frame are changed when it is passing through interconnecting devices. Based on this observation, we propose a novel implementation of the CRC update and we present the proof of correctness of our implementation. Our approach calculates the intermediate results of the changed fields based on the parallel CRC calculation and performs a single step update afterwards. Consequently, the number of cycles utilized to recalculate the CRC codes is dramatically reduced. Furthermore, an estimation on the maximum throughput is made based on synthesis results of our implementation and under the assumption that the CRC operation is the only bottleneck. In this case, we have estimated that the theoretical throughput can reach about 56 Gbps assuming realistic frame size distributions.**

Keywords— **Cyclic redundancy check, parallel CRC calculation, CRC update.**

I. Introduction

The cyclic redundancy check (CRC) is an error detection technique that is widely utilized in digital data communication and other fields such as data storage, data compression, etc.. Traditionally, the hardware implementation of CRC computations is based on the linear feedback shift registers (LFSRs), which handle the data in a serial manner. However, the serial calculation of the CRC codes can not achieve a high throughput. On the other hand, parallel CRC calculation [7] [2] can dramatically increase the throughput of CRC computations. For example, the throughput of the 32-bit parallel calculation of CRC-32 can reach several gigabits per second [5]. However, that is still not enough for higher than 10 Gigabit Ethernet networks. A possible solution is to process more bits in parallel, but that is not an efficient solution.

In this paper, we propose an efficient fast CRC update method to reduce the CRC calculation time and thereby to increase the throughput. The proposed approach is based on the observation that only a small portion residing in the beginning of a frame is changed when Ethernet frames are forwarded at routers or interconnecting devices. Our fast CRC update method is extended from the parallel CRC calculation and can adapt to any number of bits processed in parallel. The method can also reduce the data traffic and power consumption of the CRC calculation unit. In this paper, the proof of our fast CRC update approach for correctness is also presented. Furthermore, our approach is described in VHDL and the VHDL code has been synthesized, placed and routed by Xilinx ISE to evaluate its performance from the static timing report. Based on the synthesis results and realistic assumptions, we estimate that the theoretical throughput can reach 56 Gbps.

This paper is organized as follows. In Section II, the serial and parallel implementation of CRC calculation are discussed. In Section III, our fast CRC update method is introduced and proved for correctness. In Section IV, the synthesis results are presented and an estimation of the theorectical maximum throughput is given. Finally, we conclude this paper in Section V.

II. Background

In this section, a brief description of the CRC calculations is presented. The detailed theory of the CRC code and its calculation is presented in [9]. CRC calculations can be described by binary divisions using modulo 2 arithmetic. For different lengths of CRC codes, such as 16 bits (CRC-16) or 32 bits (CRC-32), a generator polynomial $G(x)$ is assigned to each of them acting as a divisor. CRC-32, which is utilized as the CRC code in Ethernet frames, is selected as an example, and we will also focus on CRC-32 in the remainder of this paper. The generator

polynomial for CRC-32 is as follows:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + x^0,$$

the highest order or called the degree of the generator polynomial (denoted by m in this paper) is 32. $m = 32$ is also the length of the CRC code. We can extract the coefficients of $G(x)$ and represent it in binary form as:

$$\begin{aligned} P &= \{p_{32}, p_{31}, \ldots, p_1, p_0\} \\ &= \{10000010011000001000111011011 0111\}. \end{aligned}$$

The most significant bit of P, p_{32}, corresponds to the coefficient of x^{32}, the highest order of $G(x)$. Similarly, p_{31} corresponds to the coefficient of x^{31}, which is 0 in this case. The ensuing bits follow the coefficients in $G(x)$ at their corresponding positions. P is called the *generator*, and uniquely coincides with the generator polynomial $G(x)$.

The basic processes of CRC calculation are the following. First, the to be protected message is perceived as an unsigned binary number D, which can be hundreds of bits long. The length of D is denoted by k in this paper. Second, m 0s are appended to D. This operation can be perceived as a multiplication with 2^m resulting in $D \times 2^m$. Third, the binary number $D \times 2^m$ is divided by generator P using modulo 2 arithmetic. The remainder of this operation is the m bits CRC code represented by C. Fourth, this CRC code C is added to the extended message resulting in $D \times 2^m + C$, which will be transmitted. Actually, no real addition needs to be performed since adding C is the same as replacing the appended m 0s with C. Finally, at the receiver's end, when performing the CRC calculation on $D \times 2^m + C$, the remainder should be 0, if no transmission errors are occurred.

Based on above discussions, the main operation of CRC calculations is nothing more than the binary divisions. Binary divisions generally can be performed by a sequence of shifts and subtractions. Furthermore, in modulo 2 arithmetic, addition and subtraction are equivalent to bitwise XORs (denoted by "$\oplus$" in this paper) and multiplication is equivalent to AND (denoted by " $\cdot$ " in this paper). Therefore, in modulo 2 arithmetic, binary divisions can be accomplished by shifts and bitwise XORs. Linear feedback shift registers (LFSRs) are designed to perform shifts and bitwise XORs, namely, binary divisions. The implementations of the serial and parallel CRC calculation based on LFSRs are described in the following sections.

A. Serial CRC Calculation

Figure 1 illustrates the basic architecture of LFSRs for serial calculation of CRC. Shift operations are performed

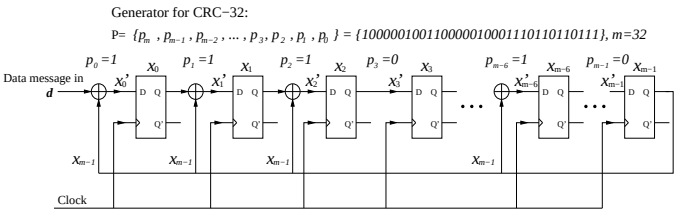


Fig. 1. Linear Feedback Shift Registers.

by a sequence of flip-flops called shift registers, which store the temporal result after every subtraction (bitwise XOR). The number of shift registers equals m, the length of the CRC code. Bitwise XORs are performed by XORs in between the shift registers. The data message shifts in from the left, beginning with the most significant bit and ending with the m 0s that is appended to the data message. When all the messages have been shifted in, the final value in the shift registers is the remainder of the division, namely, the CRC code. The LFSRs can be regarded as a discrete-time time-invariant linear system. If $X = [x_{m-1} \ldots x_1 \ x_0]^T$ is utilized to denote the state of the shift registers, in linear system theory, the state equation for LFSRs can be expressed in modular 2 arithmetic as follows (see [4] [7]):

$$X(i+1) = F \cdot X(i) \oplus G \cdot d \tag{1}$$

where $X(i)$ represents the ith state of the registers, namely, the remainder after ith subtraction;
$X(i+1)$ denotes the $(i+1)$th state of the registers, the remainder after $(i+1)$th subtraction;
d denotes the one-bit shift-in serial input;
F is an $m \times m$ matrix and G is a $1 \times m$ matrix.

If the generator is written as $P = \{p_m, p_{m-1}, ..., p_0\}$, F and G matrices can be denoted as follows.

$$F = \begin{bmatrix} p_{m-1} & 1 & 0 & \cdots & 0 \\ p_{m-2} & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ p_1 & 0 & 0 & \cdots & 1 \\ p_0 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{2}$$

$$G = [0\,0 \cdots 0\,1]^T \tag{3}$$

Consequently, if $X = [x_{m-1} \ldots x_1 \ x_0]^T$ is used to denote the current state: ith state and $X' = [x'_{m-1} \ldots x'_1 \ x'_0]^T$ is employed to denote the next state: $(i+1)$th state, the system state equation can thus be written as:

$$X' = F \cdot X \oplus G \cdot d \tag{4}$$

Furthermore, if F and G are substituted by Equations (2) and (3), we can write Equation (4) in the matrix form as:

$$\begin{bmatrix} x'_{m-1} \\ x'_{m-2} \\ \cdots \\ x'_1 \\ x'_0 \end{bmatrix} = \begin{bmatrix} p_{m-1} & 1 & 0 & \cdots & 0 \\ p_{m-2} & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ p_1 & 0 & 0 & \cdots & 1 \\ p_0 & 0 & 0 & \cdots & 0 \end{bmatrix} \cdot \begin{bmatrix} x_{m-1} \\ x_{m-2} \\ \cdots \\ x_1 \\ x_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 0 \\ 1 \end{bmatrix} \cdot d \tag{5}$$

If Equation (5) is expanded, the equations can be expressed as follows[1]:

$$\begin{cases} x'_{m-1} = (p_{m-1} \cdot x_{m-1}) \oplus x_{m-2} \\ x'_{m-2} = (p_{m-2} \cdot x_{m-1}) \oplus x_{m-3} \\ \quad \vdots \\ x'_1 = (p_1 \cdot x_{m-1}) \oplus x_0 \\ x'_0 = (p_0 \cdot x_{m-1}) \oplus d \end{cases} \tag{6}$$

Equation (6) exactly coincides with the LFSRs depicted in Figure 1. If $p_i = 1$, we have $x'_i = x_{m-1} \oplus x_{i-1}$ ($i = 1, 2, ..., m-1$), which means there is an XOR operation between register x_i and register x_{i-1}. If $p_i = 0$, we have $x'_i = x_{i-1}$, which means there is no XOR operation between these two registers, only shift operation is performed. If $x_{m-1} = 0$, we have $x_0 = d$ and $x'_i = x_{i-1}$ for $i = 1, 2, ..., m-1$, which are still shift operations.

B. Parallel CRC Calculation

A 32-bit parallel CRC implementation of CRC calculation is described here based on paper [2] and [4]. We extend the notation introduced in Section II-A in order to describe the parallel CRC implementation. The parallel CRC calculation equation is derived from Equation (4).

The solution for the system state equation Equation (4) can be written as (see [4]):

$$X(i) = F^i \cdot X(0) \oplus [F^{i-1}G, \ldots, FG, G] \cdot [d_{i-1}, \ldots, d_0]^T \tag{7}$$

where $X(i)$ denotes the ith state of the shift registers, F^i is the F matrix to the power of i, which is still an $m \times m$ matrix, and $X(0)$ is the initial state of the shift registers. Since

$$[F^{i-1}G,\ \ldots,\ FG,\ G] \cdot [d_{i-1},\ \ldots,\ d_1,\ d_0]^T = [0,\ \ldots,\ 0, d_{i-1},\ \ldots,\ d_0]^T$$

Equation (7) can be simplified as:

$$X(i) = F^i \cdot X(0) \oplus [0,\ \ldots,\ 0, d_{i-1},\ \ldots,\ d_0]^T \tag{8}$$

Equation (8) demonstrates how the *i*th state of the shift registers $X(i)$ is related to the initial states $X(0)$ and the vector $\mathbf{D} = [0, \ldots, 0, d_{i-1}, \ldots, d_0]$. $\mathbf{D}$ is an m-bit long vector with the input vector $[d_{i-1}, \ldots, d_0]$ at the lower significant bits and all zeros at the other bit positions. The idea of parallel CRC calculation is originated from this equation, that is, to calculate $X(i)$ directly from $X(0)$ and the vector $\mathbf{D}$ in one clock cycle. Now, assuming that *w* multiple bits of the data unit are processed in parallel in one clock cycle ($i = w \leq m$, where m is the length of the CRC code). The m-bit vector $\mathbf{D}$ can be represented as $\mathbf{D} = [0, \ldots, 0, d_{w-1}, \ldots, d_0]$.

Therefore, the Equation (8) can be rewritten as:

$$X(w) = F^w \cdot X(0) \oplus \mathbf{D}(0)$$

This is the equation that is utilized to calculate the first w bits of the Ethernet frame, namely, **D**(0). For the second w bits, the similar equation can be drawn:

$$X(2w) = F^w \cdot X(w) \oplus \mathbf{D}(1)$$

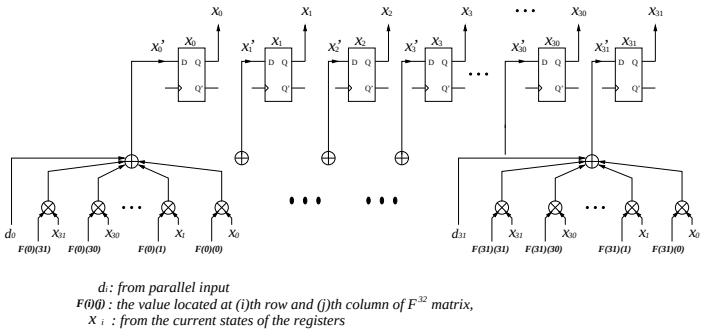
Similar recursive equations can be derived until the input data unit ($D \times 2^m$) is finished, and the final value of X is the CRC code. We describe these recursive equations utilized to calculate CRC code in a unified form as follow.

$$X' = F^w \cdot X \oplus \mathbf{D} \tag{9}$$

From this equation, it is notable that the parallel calculation of CRC code is a recursive calculation of the next state of the registers X' by the current states X, the predetermined matrix F^w and the parallel input vector $\mathbf{D}$ using Equation (9). The recursive calculation is terminated when the input bits are finished.

An example of parallel CRC calculation, e.g., 32-bit parallel calculation of CRC-32 can be drawn based on above equations ($w = m = 32$). The block diagram is depicted in Figure 2. Only the inputs of the register x_0 and register x_{31} have been illustrated, the other inputs of the registers are in the similar cases.

[1]Note: the multiplication of matrix also follows the modulo 2 arithmetic, that is, multiplication is performed by AND and addition is performed by XOR.



d_i : from parallel input
F(i)(j) : the value located at (i)th row and (j)th column of F^{32} matrix,
X_i : from the current states of the registers

Fig. 2. Block diagram for parallel calculation of CRC-32.

III. FAST CRC UPDATE

When we investigate how the CRC is calculated and is verified during data transmission process over the Internet, we observed that the following steps are performed in sequence when reaching an interconnecting device (See Figure 3):

1. CRC verification is performed in order to verify the correctness of the frame. If the verification fails, the frame is discarded.
2. The frame is passed through upper layers. Upper layer operations, such as finding the next hop according to the destination address, updating the TTL (Time To Live) field and checksum field in the IP packet header and so on. are performed.
3. The reassembled frame without the CRC code is forwarded to perform CRC32 calculation.
4. The calculated CRC code will be appended to the frame and the frame will be forwarded through physical layer to its next hop.

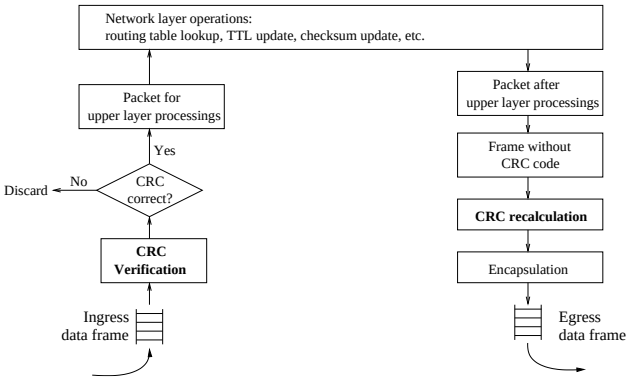


Fig. 3. CRC verifi cation and recalculation.

Furthermore, the difference between an incoming frame and an outgoing frame (after being updated by higher layer operations) is small and the differences usually occur at fixed bit-positions [1] [8]. Figure 4 depicts a detailed Ethernet frame format. The fields in gray are the most likely to be changed fields. The TTL field in the IP header is decreased by 1 every time when a packet passes through a router. The Header Checksum in the IP header also changes accordingly. The source Ethernet address and the destination Ethernet address are different as well. All of them will be modified when the frame is sent out. The changed fields, 15 bytes in total, are only a small portion of the whole frame, which may range from 64 bytes to 1518 bytes. Furthermore, we observe that the changed fields lies in the first 26 bytes of the frame and the remainder of the frame stays intact. Therefore, it is not necessary to recalculate the CRC code over the entire frame, instead, a fast CRC update, which calculates only the changed fields, performs a single step update, and exors the updated value with the old CRC, is more preferable. Our fast CRC update design is based on above observations and Equation (9) discussed in Section II-B. 32-bit parallel calculation of CRC-32 is again utilized as an example, Equation (9) can be rewritten as:

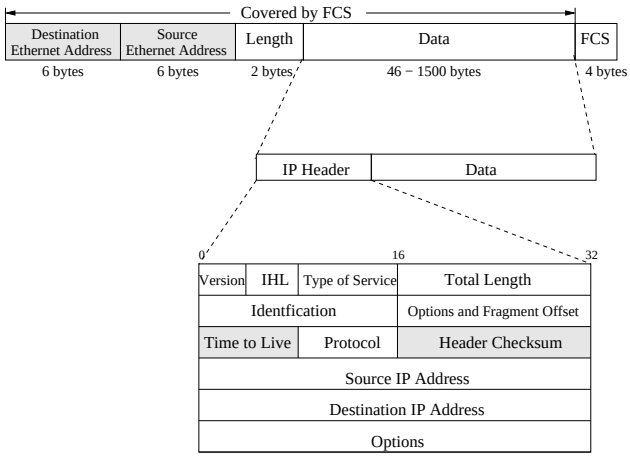


Fig. 4. Detailed Ethernet frame structure.

$$X' = F^{32} \cdot X \oplus \mathbf{D} \tag{10}$$

where F^{32} is a 32×32 matrix for CRC32, X' and X is the next and current states of the shift registers respectively, and $\mathbf{D}$ is the m-bit parallel input vector. Furthermore, the associative and distributive laws apply for the modulo 2 arithmetic, that is,

$$\begin{aligned} a \cdot (b \oplus c) &= (a \cdot b) \oplus (a \cdot c) \\ a \oplus (b \oplus c) &= (a \oplus b) \oplus c \end{aligned}$$

Now, assuming that the states of the shift register follow the sequence $X(0), X(1), X(2), \ldots, X(n)$ and the parallel inputs follow the sequence $\mathbf{D}(0), \mathbf{D}(1), \mathbf{D}(2), \ldots, \mathbf{D}(n-1)$, where $n = \frac{k+m}{w}$ is the number of cycles to get the final CRC code. From the recursive equation

Equation (10), we have:

$$\begin{array}{rcl} X(1) & = & F^{32} \cdot X(0) \oplus \mathbf{D}(0) \\ X(2) & = & F^{32} \cdot X(1) \oplus \mathbf{D}(1) \\ & \cdots & \\ X(7) & = & F^{32} \cdot X(6) \oplus \mathbf{D}(6) \\ \hline X(8) & = & F^{32} \cdot X(7) \oplus \mathbf{D}(7) \\ X(9) & = & F^{32} \cdot X(8) \oplus \mathbf{D}(8) \\ & = & F^{32} \cdot \Big(F^{32} \cdot X(7) \oplus \mathbf{D}(7)\Big) \oplus \mathbf{D}(8) \\ & = & \Big((F^{32})^2 \cdot X(7)\Big) \ \oplus \ \Big(F^{32} \cdot \mathbf{D}(7)\Big) \ \oplus \ \mathbf{D}(8) \\ & \cdots & \end{array}$$

The equations above the line, are calculated from the changed fields, i.e., from $\mathbf{D}(0)$ to $\mathbf{D}(6)$, the first 28 bytes of a frame. And the equations below the line are calculated from the unchanged fields and they are all denoted by the intermediate state $X(7)$. Finally, the final register states $X(n)$, $n = \frac{k+m}{w}$, namely the CRC code, can be denoted by the intermediate states $X(7)$ and the unchanged fields.

$$\begin{aligned} CRC &= X(n) \\ &= \big((F^{32})^{n-7} \cdot X(7)\big) \oplus \ \big((F^{32})^{n-8} \cdot \mathbf{D}(7)\big) \\ &\quad \oplus \ldots \oplus \mathbf{D}(n-1) \end{aligned} \tag{11}$$

For the old CRC code before frame update and the new CRC code after the update, Equation (11) can be described as follows:

$$\begin{aligned} CRC_{old} = &\big((F^{32})^{n-7} \cdot X(7)_{old}\big) \oplus \big((F^{32})^{n-8} \cdot \mathbf{D}(7)\big) \\ &\oplus \ldots \oplus \mathbf{D}(n-1) \end{aligned}$$

and

$$\begin{aligned} CRC_{new} = &\big((F^{32})^{n-7} \cdot X(7)_{new}\big) \oplus \big((F^{32})^{n-8} \cdot \mathbf{D}(7)\big) \\ &\oplus \ldots \oplus \mathbf{D}(n-1) \end{aligned}$$

Since all differences between the old and new frame locate in the first 26 bytes of the frame, for $i > 6$, the parallel input $\mathbf{D}(i)$ remains unchanged. If we XOR two CRC codes, the two set of unchanged fields are eliminated:

$$\begin{aligned} & CRC_{new} \oplus CRC_{old} \\ &= \Big((F^{32})^{n-7} \cdot X(7)_{new}\Big) \oplus \Big((F^{32})^{n-7} \cdot X(7)_{old}\Big) \\ &= (F^{32})^{n-7} \cdot \Big(X(7)_{new} \oplus X(7)_{old}\Big) \end{aligned}$$

therefore, the new CRC code can be represented in the following form:

$$CRC_{new} = (F^{32})^{n-7} \cdot \Big(X(7)_{new} \oplus X(7)_{old}\Big) \ \oplus \ CRC_{old} \tag{12}$$

Consequently, the block diagram of the CRC update circuit can be depicted in Figure 5. The CRC recalculation is only performed over the first 28 bytes of the frame. The intermediate state $X(7)$ is collected, and compared with the old one which can be obtained through CRC verification, and perform the update. Finally, the new CRC code is obtained by adding the update result to the old CRC code.

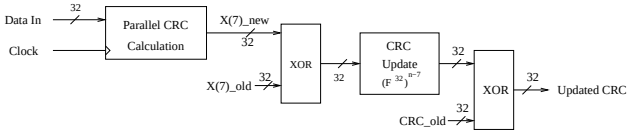


Fig. 5. Block diagram of fast CRC update circuit.

In Equation (12), for a fixed frame length, the matrix $(F^{32})^{n-7}$ ($n = \frac{k+m}{w}$) can be pre-calculated, $X(7)_{old}$ can be obtained from the CRC verification, $X(7)_{new}$ is calculated through parallel CRC circuit. For the smallest frame size, 64 bytes, we have $n = \frac{64\text{bytes} \times 8\text{bits}}{32} = 16$. Equation (12) can be written as

$$CRC_{new} = (F^{32})^{9} \cdot \Big(X(7)_{new} \oplus X(7)_{old}\Big) \ \oplus \ CRC_{old} \tag{13}$$

where $(F^{32})^9$ is a fixed 32×32 matrix, $X(7)_{new}$ and $X(7)_{old}$ can be calculated from parallel CRC circuit. CRC_{old} is the original CRC code and CRC_{new} is the updated CRC code. Therefore, for CRC fast update of a 64-byte frame, we only need to calculate the first 28 bytes of the frame, namely get intermediate state X(7) and then perform the CRC update to get the new CRC code.

For the calculation of the first 28 bytes CRC, 7 cycles are needed (32-bit parallel calculation) and for update, only one cycle is needed. In total, only 8 cycles are needed for the fast update of the 64-byte frame with $(F^{32})^9$ precalculated.

Furthermore, as can be observed on the real Internet, there are three predominant frame sizes: 64 bytes, 596 bytes and 1518 bytes. They make up $35\%, 11.5\%$ and 10% of the total, respectively [6]. Similar results can also be found in [3] and [10]. It is advantageous to find the corresponding $(F^{32})^{n-7}$ for these frame sizes and implement them as CRC update units as well in order to further improve the performance. Accordingly, for the frames of these sizes, only 8 cycles are needed to calculate CRC code when compared with the $n = \frac{k+m}{w}$ cycles in the parallel CRC calculation circuit. In Figure 6, the number of cycles needed to calculate the CRC code utilizing the parallel method and our fast CRC update method is depicted for different frame sizes. The performance gain of our approach becomes increasingly more significant for longer frames because our method recalculate the new CRC in a constant number of cycles.

Once the $(F^{32})^{n-7}$ is determined, for example, $(F^{32})^9$ in Equation (13), the CRC update scheme can be extended

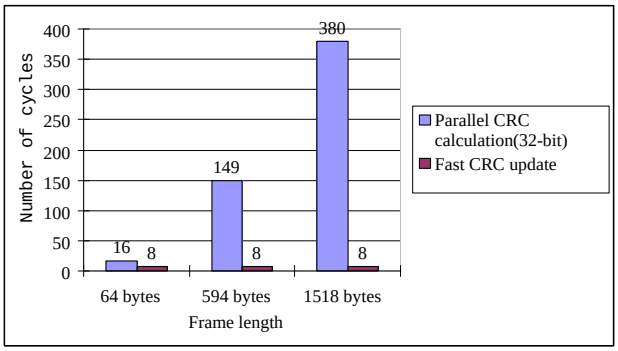


Fig. 6. Comparison of the number of cycles.

to update the CRC code for the frames with arbitrary length. Suppose h cycles are needed to calculate the CRC code over a frame with random length using 32-bit parallel CRC calculation. Obviously $h \geq 16$, which is the cycles for the smallest 64-byte frame. From Equation (11), let $n = h$, the final CRC code is

$$\begin{aligned} X(h) &= \Big((F^{32})^9 \cdot X(h-9)\Big) \\ &\quad \oplus \Big((F^{32})^8 \cdot \mathbf{D}(h-9)\Big) \\ &\quad \oplus \Big((F^{32})^7 \cdot \mathbf{D}(h-8)\Big) \\ &\quad \oplus \cdots \oplus \mathbf{D}(h-1) \end{aligned} \tag{14}$$

Since $h \geq 16$, $h - 9 \geq 7$, from $\mathbf{D}(h-9)$ on, the inputs are surely those intact fields in the frame. Equation (14) is equivalent to Equation (11) except the state is changed to $X(h-9)$. Therefore, Equation (13) can be rewritten as:

$$CRC_{new} = (F^{32})^9 \cdot \Big(X(h-9)_{new} \oplus X(h-9)_{old}\Big) \oplus CRC_{old} \tag{15}$$

Equation (15) differs with Equation (13) only in $X(h-9)$, which depends on the size of the frame. We stop the parallel CRC calculation at the stage $X(h-9)$, and perform one step CRC update. Therefore, for any frame size, in fast CRC update scheme, at least 8 cycles is reduced compared with the parallel CRC calculation. If we can build more circuits for $(F^{32})^{n-7}$, for example, $(F^{32})^{n-7}$ for 64 bytes, 128 bytes, 256 bytes etc., much more cycles will be reduced. Equation (15) then becomes:

$$\begin{aligned} &CRC_{new} \\ &= (F^{32})^{n-7} \cdot \Big(X(h-(n-7))_{new} \oplus X(h-(n-7))_{old}\Big) \\ &\quad \oplus CRC_{old} \end{aligned}$$

In this paper, we only build and test the circuits for 64 bytes, 594 bytes and 1518 bytes. Their corresponding $(F^{32})^{n-7}$ matrices are pre-calculated in MATLAB.

IV. Synthesis Results

In this section, we describe the synthesis, place and route results for of three CRC update blocks: 64 bytes ($(F^{32})^9$), 594 bytes ($(F^{32})^{142}$) and 1518 bytes($(F^{32})^{373}$) of our design after having modeled it in VHDL.

The design is described in VHDL and synthesis, placement and route are performed for three CRC update blocks: 64 bytes ($(F^{32})^9$), 594 bytes ($(F^{32})^{142}$) and 1518 bytes($(F^{32})^{373}$), respectively. The area utilization and the performance from post place and route static timing report on the Xilinx Virtex II 2v250fg256-6 device are demonstrated as follows: With these results, the performance of

CRC Update blocks	Utilization	Performance
$(F^{32})^9$	283 slices	196 MHZ
$(F^{32})^{142}$	239 slices	236 MHZ
$(F^{32})^{373}$	290 slices	217 MHZ

TABLE I

Fast CRC update area and performance report for different processing unit.

the fast CRC update is evaluated as follows. For the frames with the length 64 bytes, 594 bytes and 1518 bytes, the time utilized to obtain the CRC code by parallel CRC calculation and fast CRC update calculation is compared and demonstrated in Table II. As can be observed, there is a great reduction of the calculating time. For larger frame size, the reduced ratio is much higher.

Frame (bytes)	CRC calculation time comparisons		
	Parallel	Fast update	Time reduced
64	65 ns	41 ns	37%
594	605 ns	34 ns	94%
1518	1543 ns	37 ns	97%

TABLE II

Clock cycles and calculation time comparisons on different frame sizes.

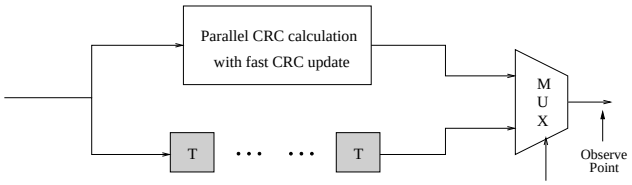


Fig. 7. The block diagram for the transmission of the frame with CRC code.

In order to evaluate the throughput of CRC update, the manner in which frames with CRC codes are transmitted

is first analyzed. As depicted in Figure 7, the CRC code is calculated while the frame is being transmitted. The lower path in the figure is the data path, where the frame without the CRC code goes through. The same frame without CRC code also is the input of the upper path to calculate the CRC code using either parallel CRC calculation with fast CRC update. Several delay elements (depicted in gray and named T) are inserted in the data path in order to synchronize with the CRC calculation so that the CRC code is obtained and transmitted exactly after the frame has been transmitted. Therefore, the throughput of this block is highly depended on the time consumed in CRC calculations. The shorter the CRC calculation time, the faster the frame is transmitted.

If we assume that the CRC calculation is the only bottleneck of a data transmission system, we can estimate the maximum throughput that the parallel calculation of CRC with CRC update can support. The observation of the throughput is performed from the observe point depicted in Figure 7. Assume that the three processing units are only applied to their corresponding frame length, that is, 64 bytes, 594 bytes, 1518 bytes, respectively and the clock rates for these three circuits are obtained from Table IV. For the 64-byte frames on the Internet, 8 cycles running at 196 MHz are needed. Therefore, the throughput for 64-byte frames is:

$$(64 \times 8) \text{ bits}/(8 \text{ cycles}/196 \text{ MHz}) = 12.54 \text{ Gbps}$$

Similarly, the throughput for 594-byte frame and 1518-byte frame can be calculated and the results are listed in Table III. Therefore, the total throughput of the CRC parallel calculation with fast update could be estimated as follows. For simplicity, assume that the rest of the frames other than 64 bytes, 594 bytes, 1518 bytes still use parallel CRC calculation (7.88 Gbps [5]).

frame length	frame distribution	calculation time	throughput (Gbps)
64 bytes	35%	41 ns	12.54
594 bytes	11.5%	34 ns	139.76
1518 bytes	10%	37 ns	328.22

TABLE III
CRC UPDATE THROUGHPUT FOR 64 BYTES, 594 BYTES AND 1518 BYTES.

$$\begin{array}{rrcl} & 12.48 \text{ Gbps} \times 35\% & = & 4.37 \\ + & 139.76 \text{ Gbps} \times 11.5\% & = & 16.07 \\ + & 328.22 \text{ Gbps} \times 10\% & = & 32.82 \\ + & 7.88 \text{ Gbps} \times 43.5\% & = & 3.43 \\ \hline & & Total & 56.7 \text{ Gbps} \end{array}$$

This throughput is a rough estimation of the throughput that the parallel calculation of CRC with fast CRC update can support. Furthermore, if the CRC fast update scheme is also utilized to calculate CRC code of the frames other than 64 bytes, 594 bytes and 1518 bytes, the throughput could be much higher. If a simple pipeline is built on parallel CRC and CRC update processing unit, one extra cycle can be saved, and the throughput can be further improved.

Therefore, if the other bottlenecks are excluded, the parallel calculation of CRC with CRC update can reach a throughput of 56.7 Gbps. Actually, the throughput is only a theoretical estimation, the real throughput will be much lower due to other bottlenecks. However, with this design, it is obvious that CRC calculation can no longer be considered as a bottleneck for data transmission.

The CRC update implementation offers several additional advantages. First, only 28 bytes data plus 8 bytes original CRC code and intermediate states are needed to be copied and sent to calculate the new CRC instead of the total data message which may have hundreds of bits. Therefore, the time used for data moving is reduced. Second, only 8 cycles are required to obtain the new CRC, the corresponding power consumption is also reduced compared with the CRC recalculation of the overall frame. A disadvantage is that some extra buffer may be needed to store the intermediate state $X(7)$, or $X(h-(n-7))$, and the old CRC code. The entry of the extra buffer also need to point to the frame body which may be stored in the main buffer of the system. And thanks to the modern techniques for memory, a buffer with 64-byte entries is not an expensive implementation. It is a small sacrifice compared with the great throughput.

In summary, the fast CRC update scheme can dramatically increase the throughput of the CRC recalculation. CRC fast update can be applied to any other number of bits processed in parallel and any other CRC algorithms such as CRC-16, CRC-8. Furthermore, the fast CRC update scheme is not only suitable for Ethernet, but also hold true for other protocols such as ATM (AAL-5 frames). And the fast CRC update may have a broader usage because of the wide utilizations of the CRC error detecting techniques.

V. CONCLUSIONS

In this paper, we presented a novel method to update the CRC code when packets are passing through interconnecting devices. The manner in which CRC code is generated was first introduced. Second, serial and parallel calculations of CRC code based on Linear Feedback Shift Registers (LFSRs) were discussed. The linear system state equation of LFSRs was used to provide the mathematic background for the fast CRC update. Subsequently, the fast

CRC update scheme was proposed and proved for correctness based on the linear system state equations. The idea of our fast CRC update is based on the observation that only a small portion residing in the beginning of an Ethernet frame is changed when it is forwarded by the interconnecting devices. Therefore, the fast CRC update only calculates the changed portion of a frame and performs a single step update afterwards to obtain the new CRC code. This method dramatically improves the throughput of CRC recalculation. Based on synthesis result of our design and realistic assumptions, we can safely estimate that our design can support a throughput of about 56 Gbps.

REFERENCES

[1] F. Braun and M. Waldvogel, *Fast Incremental CRC Updates for IP over ATM Networks*, IEEE Workshop on High Performance Switching and Routing (2001).

[2] M. Braun, J. Friedrich, F. Grün, and J. Lembert, *Parallel CRC computation in FPGAs*, Field-Programmable Logic: Smart Applications, New Paradigms and Compilers (Reiner W. Hartenstein and Manfred Glesner, eds.), Springer-Verlag, Berlin, 1996, pp. 156–165.

[3] Memik et al., *Evaluating Network Processors using Netbench*, ACM Transactions on Embedded Computing System (2002).

[4] G. Patane G. Campobello and M. Russo, *Parallel CRC Realization*, 2003.

[5] W. Lu, *Designing TCP/IP Functions in FPGAs*, Master's thesis, Delft University of Technology, Computer Engineering Group, 2003.

[6] M. Miyazaki, *Workload Characterization and Performance for a Network Processor*, Master's thesis, Princeton University, Department of Electrical Engineering, 2002.

[7] T. Pei and C. Zukowski, *High-Speed Parallel CRC Circuits in VLSI*, IEEE Transactions on Communications (1992).

[8] A. Rijsinghani, *Computation of the Internet Checksum via Incremental Update*, IETF RFC (1994).

[9] S.Lin and Jr D.J.Costello, *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Inc., 1983.

[10] *Modeling Tomorrow's Internet*, www.lightreading.com.