Out of Order Incremental CRC Computation

Julian Satran Dafna Sheinwald Ilan Shimony

IBM Haifa Labs
February 25, 2003

Abstract

We consider a communication protocol where the sender takes an information message, appends a CRC, breaks the resulting message into small segments, and transmits them separately, possibly via different routes. Traditionally, the receiver reverses the sender operations. The receiver first assembles all the segments that make up the message, then computes a CRC for the message and verifies it against the arriving CRC, and finally delivers the information message to the upper layer protocol (ULP). We present an incremental CRC computation, to be carried out by the receiver, whereby each arriving segment contributes its share to the message's CRC without waiting for all the other segments to arrive in advance. This allows immediate delivery of the arriving segment to the ULP. We impose no constraint on the order of segment arrival. Yet, our scheme does not increase time complexity with respect to the classic computation, which assembles all the segments first, and it uses only a fixed and very small amount of extra memory. Our scheme is beneficial when the ULP can process the individual segments without first reading the entire message. We assume, to this end, that once all the segments arrive and the receiver compares the CRC it computed against the arriving message's CRC, it can, upon detecting an error, revert to the state it and the ULP had prior to the arrival of any segment of that message.

A practical application is the evolving protocol for Remote Direct Memory Access (RDMA) over TCP, where an overall CRC is added to a concatenation of data segments. However, our scheme also applies to a much wider set of network protocols.

1 Introduction

Braun and Waldvogel [2] consider packet CRC (Cyclic Redundancy Code) recreation at network routers which modify information at the packet header. The traditional approach of verifying the CRC (error detection) and then creating a new CRC for the modified packet can create a bottleneck at high network speeds. Braun and Waldvogel propose to eliminate most of the

duplicate calculations, namely the CRC calculation that is based on the non-modified part of the frame, and only compute the change in the CRC due to the actual modification done. In [1], the authors consider embedding their technique into a strictly layered, modular protocol stack.

We consider another type of incremental computation that speeds up CRC calculation in a different manner. In our case, the message, comprising a large data block and an associated CRC, is broken up by the sender into small segments, which travel independently to the receiver and arrive there in any order. Traditionally, the receiver handles this by reversing the sender operation. First it assembles all the segments composing the message, and then it computes a CRC based on the data block part and verifies it against the arriving CRC. Our objective is to eliminate the time spent waiting for all the segments to arrive ("reassembly time") before the CRC is computed. We suggest computing the message CRC incrementally, while accumulating individual contributions of the segments. This will allow faster delivery of an arriving segment to the upper layer protocol (ULP), having calculated its share in the message's CRC. Our incremental computation scheme consumes total calculation time over all segments which does not exceed the time consumed by the traditional "in order" computation, while using a very small extra memory of fixed size, independent of the message length or number of segments.

Our scheme is beneficial when the ULP can start processing individual segments without first reading the entire message. We assume, to this end, that once all the segments arrive and the receiver compares the CRC it has computed against the arriving message's CRC, it can, upon detecting an error, retrieve all the segments it has forwarded to the ULP.

In Section 2, we briefly review Cyclic Redundancy Codes. In Section 3, we present our incremental CRC computation scheme. In Section 4, we show an actual application, related to Remote Direct Memory Access (RDMA), where the segment conveys its target address within a very large address space. This space is much larger than the size of the including message, and the target address is where the segment is to be stored by the receiver ULP. Our incremental computation of the message CRC allows the receiver to forward the segments to the ULP, and have them stored, immediately following their arrival, without first reassembling them all. Section 5 concludes the paper.

2 Cyclic Codes - A Brief Overview

Detailed discussions of cyclic codes can be found in text books (e.g., [3] or [4]). For the sake of completeness, we present a short overview.

A codeword c of a binary (n, k) cyclic code consists of k data, or information bits, followed by $n - k = r$ parity bits, also called check bits or CRC bits. These CRC bits are computed from the block of data bits. Viewing codewords as polynomials $c(x)$, with the i-th bit of the codeword being the coefficient of x^i (the leading coefficients lead the codeword transmission and the 0-th bit

concludes it), the code is associated with a predetermined generator polynomial $g(x)$ of degree r that evenly divides all the codewords $c(x)$. Polynomial operations take place in $GF(2)$, the Galois Field of size two, with additions and subtractions being modulo 2. The code is linear, that is, the (bitwise) addition of any two codewords yields another codeword. We refer interchangeably to messages or codewords and the polynomials that correspond to them.

Given a binary information message m of length $|m| \leq k$, we compute r parity bits such that, when appended to m, the polynomial corresponding to that concatenation is evenly divisible by $g(x)$, and is thus a codeword. These r bits are the coefficients of the polynomial $C(m(x))$ that is computed from the polynomial $m(x)$ as follows:

$$C(m(x)) = (x^r m(x)) \mod g(x) \tag{1}$$

Subtracting this remainder from the dividend $x^r m(x)$ yields

$$c(x) = x^r m(x) - C(m(x))$$

which is evenly divisible by $g(x)$. Because $x^r m(x)$ has all its r least significant coefficients equal to 0, and the degree of $C(m(x))$ is at most $r-1$, and because in $GF(2)$, subtraction is the same as addition, we can see the coefficients of $c(x)$ as the concatenation of $m(x)$ followed by the r coefficients of $C(m(x))$. In the case where the degree of $C(m(x))$ is smaller than $r-1$, we extend $C(m(x))$ with leading 0 coefficients.

$c(x)$ is transmitted to the receiver. Due to errors in transmission, the polynomial $c'(x)$ arriving at the receiver might be corrupted, and will be different from $c(x)$. The receiver detects errors in transmission using the maximum likelihood principle, taking into account the very small probability for distortion at any specific bit position. It tests $c'(x)$ for being a codeword. If it is not a codeword, a transmission error is clearly detected. If it is a codeword, then the codeword most likely sent is $c'(x)$ itself, and the receiver assumes no error occurred in transmission. Testing $c'(x)$ for being a codeword (i.e., evenly divisible by $g(x)$) can be done by repeating the sender calculations: compute r CRC bits from the first $|c'| - r$ bits of $c'(x)$, as in Eq. (1), and compare them against the trailing r bits of $c'(x)$. A mismatch indicates that $c'(x)$ is not a codeword. Alternatively, $c'(x)$ is evenly divisible by $g(x)$ if and only if feeding the whole of it, as if it were an information message, into the computation of Eq. (1), yields r zero bits.

As mentioned above, if the corrupted word c' happens to be a codeword, the receiver can not detect that it is corrupted. By the linearity of the code, this happens if and only if the error word $e = c' - c$ (bitwise exclusive or) is a codeword by itself.

The classic hardware implementation of CRC computation and verification uses a binary linear feedback shift register (LFSR) of length r, wired according to the coefficients of $g(x)$. It takes $|m|$ clock cycles to feed message m through the LFSR, after which the contents of the LFSR are the r CRC bits for m. Hence, the time complexity of CRC computation and verification is linear in the length of the message. See Fig. 1 for an example of an LFSR wired according to the coefficients of $x^3 + x + 1$.

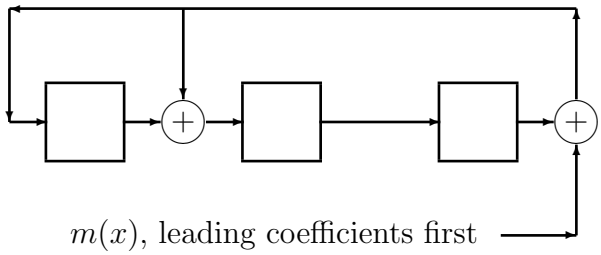

Figure 1: A Linear Feed Back Shift Register - when fed with polynomial $m(x)$ presents $(x^3\, m(x)) \mod (x^3 + x + 1)$, leading coefficient in the right bit register

Let a be the minimal integer such that $x^a - 1$ is evenly divisible by $g(x)$. It can be shown that $a \leq 2^r$. There is a number d, which depends on the choice of $g(x)$, such that any polynomial of degree smaller than a, which is evenly divisible by $g(x)$, has at least d non zero coefficients. (Typical numbers for commonly used generator polynomials are $d = 3$, or $d = 4$). Hence, any error word of less than d '1' bits is not a codeword by itself, and thus will be detected by the receiver. The association of being evenly divisible by $g(x)$ with having at least d non zero coefficients does not hold for polynomials of degree a or more. $x^a - 1$ is an example of such a polynomial. It only has two non zero coefficients, even for $g(x)$ with a larger d. It follows that for codewords longer than a, the division by $g(x)$ procedure can only ensure the detection of any error word with a single '1' bit, but will possibly fail to detect errors with two or more. To detect error words with only a single '1' bit, the simple mechanism of a single parity bit suffices; there is no need for a generator polynomial and r parity bits. Therefore, the length n of codewords is less than or equal to a. n does not need to be fixed in advance. We can view all the words whose length n does not exceed a, and which are evenly divisible by $g(x)$, as codewords of length a with all 0-s leading coefficients. We can then use the LFSR for any information messages of length not exceeding $a - r$.

3 Incremental CRC Calculation

As mentioned above, the traditional CRC computation for a message m of length $|m|$ takes $O(|m|)$ clock cycles, feeding the entire message through an LFSR of r bit registers, which is wired according to $g(x)$. In this section, we show how to calculate the CRC incrementally, by accumulating the individual contributions of the segments comprising m, where the order in which the segments are processed is arbitrary. The overall time complexity is $O(|m|)$, and the extra memory and logic used is fixed, a small function of r, independent of the message.

Breaking the message m into s segments S_i of respective lengths $|S_i|$, we can write

$$m(x) = \sum_{i=1}^{s} x^{l_i} S_i(x) \quad \text{with} \quad l_i = \sum_{j=1}^{i-1} |S_j| \tag{2}$$

l_i is the offset of segment S_i within the message m, measured by the number of bits from the least significant bit of m to the least significant bit of S_i. By the linearity of the $\mod g(x)$ function we have:

$$C(m(x)) = (x^r m(x)) \mod g(x) = \sum_{i=1}^{s} \left\{ \left(x^r \, x^{l_i} \, S_i(x) \right) \mod g(x) \right\} \tag{3}$$

That is, the CRC of $m(x)$ equals the sum of s CRC-s, a sum to which segment $S_i(x)$ contributes the CRC of $x^{l_i} S_i(x)$. We assume that when the receiver processes segment S_i, it knows l_i (e.g., the offset l_i is delivered in the segment header). In Section 4, we consider a different scenario. Direct calculation of (3) thus calls for feeding the arriving segment S_i, followed by a tail of l_i 0-s, into an LFSR wired according to $g(x)$. This will take $|S_i| + l_i$ clock cycles. Assuming the segments are of approximately same size, the total number of clock cycles, for the processing of all the segments, is $\frac{s}{2} \left(|m| + \frac{|m|}{s} \right) = O(|m| \, s)$. In scenarios where the length of the segments is limited by a fixed number of bits, which is independent of the message length, this means $O(|m|^2)$ clock cycles.

By simulating the feeding of each tail of zeros, using a process which takes only $O(r)$ clock cycles, our scheme reduces the time it takes to process segment S_i to $O(|S_i| + r)$ clock cycles. This accumulates to $O(|m| + s \, r)$ clock cycles for the processing of the entire message. Bearing in mind that r is usually 32 or 64, and is fixed in advance independently of the message or segment length, we have $O(|S_i|)$ clock cycles for the processing of segment S_i, and $O(|m|)$ clock cycles for the processing of the entire message, the same complexity as by the traditional computation.

Given an LFSR of r bit registers, let $\mathbf{T}$ be the $r \times r$ binary transformation matrix in which $T_{i,j} = 1$ iff bit register j directly feeds bit register i in the LFSR. Namely, there is a wire leading from bit register j to bit register i, possibly going through adders, but not through other bit registers. For the LFSR in Fig. 1, as an example, enumerating the bit registers from left to right, the corresponding transformation matrix is

$$\mathbf{T} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Let the column vector $\bar{\mathbf{p}}$ denotes the current contents of the bit registers, with the leftmost bit register at the top of the vector. If the input to the LFSR consists of 0-s only, then after one clock cycle the contents of the bit registers will be $\mathbf{T} \, \bar{\mathbf{p}}$, where matrix operations carried out in $GF(2)$ (i.e., modulo 2). Consequently, if $\bar{\mathbf{p}}$ denotes the current contents of the LFSR and the input consists of only 0-s, then after two clock cycles the contents of the LFSR will be $\mathbf{T} \, \mathbf{T} \, \bar{\mathbf{p}}$, and after l clock cycles, the contents will be $\mathbf{T}^l \, \bar{\mathbf{p}}$.

Transformation $\mathbf{T}$ is directly derived from our generator polynomial $g(x)$, and we can compute in advance the r matrices $\mathbf{T}^{2^i}$ for $i = 0, 1, \ldots, r-1$. For a number of clock cycles l, whose binary representation is $(b_{r-1} \ldots b_1\, b_0)$, $b_i \in \{0, 1\}$, we then have

$$\mathbf{T}^l = \prod_{b_i \neq 0} \mathbf{T}^{2^i} \tag{4}$$

Upon the arrival of a segment S, whose offset within its including message is l, we first feed S into the LFSR, in $|S|$ clock cycles. The LFSR thus takes the value of $C(S(x))$. Then, denoting these LFSR's contents by the column vector $\bar{\mathbf{p}} = \bar{\mathbf{p}}_0$, and the binary representation of l by $(b_{r-1} \ldots b_1\, b_0)$, we update iteratively as follows:

$$\text{for each } \; b_i \neq 0, \quad \bar{\mathbf{p}} \leftarrow \mathbf{T}^{2^i}\, \bar{\mathbf{p}} \; . \tag{5}$$

By (4), we end up with $\bar{\mathbf{p}} = \mathbf{T}^l \bar{\mathbf{p}}_0$, which would have been the LFSR's contents, had it been fed with S followed by a tail of l zeros.

Altogether, we compute the contribution of the arriving segment S to the message CRC, according to (3), in $|S|$ clock cycles, followed by at most r updates.

We now turn to investigate the time it takes to make these r updates. For $\mathbf{A}$, an $r \times r$ binary matrix, the multiplication $\bar{\mathbf{q}} \leftarrow \mathbf{A}\, \bar{\mathbf{p}}$, with $\bar{\mathbf{p}}$ and $\bar{\mathbf{q}}$ being dimension r vectors, can be calculated in one clock cycle. See, for example, Figure 2, where the horizontal busses are of width r: the first bus carries $A_{1,1}, A_{1,2}, \ldots, A_{1,r}$, with $A_{1,1}$ participating in the first **AND** gate on the left, $A_{1,2}$ participating in the second **AND** from left, etc. The vertical wires are of width 1, with the j-th line carrying p_j. Formally speaking, there are $r \times r$ **AND** gates $N_{i,j}$ making up the first layer, and r **XOR** gates X_i making up the second layer. The inputs to $N_{i,j}$ are $A_{i,j}$ and p_j, while the inputs of X_i are the outputs of $N_{i,1}, N_{i,2}, \ldots, N_{i,r}$, and its output is q_i.

Using this hardware, and the r matrices $\mathbf{T}^{2^i}$ which we prepare in advance, we can thus perform the iterative process (5) in r clock cycles.

Altogether, by precomputing r matrices $\mathbf{T}^{2^i}$, which depend only on $g(x)$, and using a rather standard hardware, whose size depends only on r, we can compute the CRC of a message m, by processing its segments, in a total time of $|m| + r * s$ clock cycles. Because r is only 32 or 64, and s can not exceed m, the total computation time is $O(|m|)$. That is, the time complexity of our incremental, out of order, CRC computation does not exceed the complexity of the original one pass computation. The extra memory needed is a small function of r.

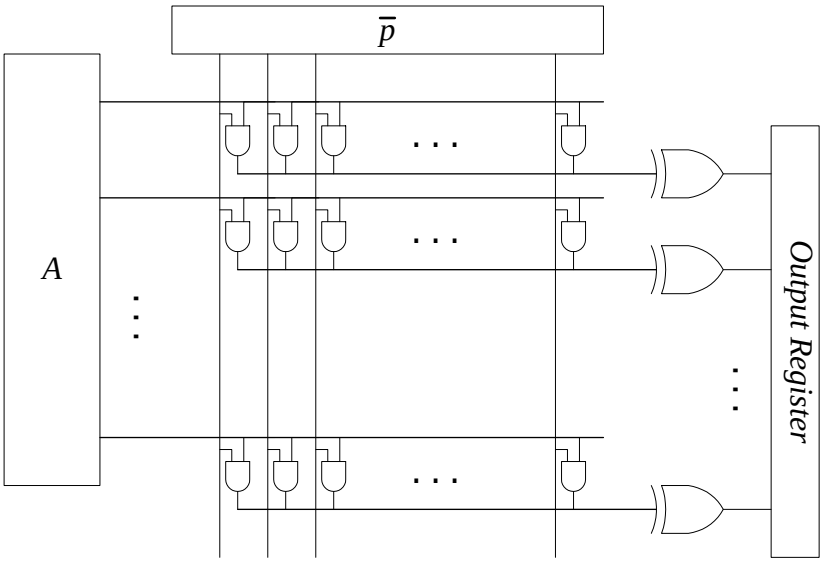

Figure 2: Parallel calculation of $\mathbf{A}\bar{\mathbf{p}}$

4 Incremental CRC Computation with Relative Segment Offsets

In this section, we consider an application, derived from the evolving RDMA (Remote Direct Memory Access) protocol, where the ULP can process each segment upon its arrival. As such, this application can benefit greatly from our scheme. The segments in this application do not convey their relative offset within their message. Nevertheless, the receiver can tell when all segments comprising a message have arrived, and it can identify the message's CRC as computed by the sender, and verify against it. The ULP at the receiver side posts an arriving segment in a large address space, called *the reassembly address space*, which is much larger than the size of a message. In its header, each segment conveys the address where it is to be posted, called its *target_address*.

Viewed as a polynomial, the leading bit of the segment corresponds to its leading coefficient. The segment's *target_address* is a bit address, specifying where that first bit of the segment should be stored, with the rest of the bits to be stored in consecutive increasing addresses. We assume the segments do not overlap when stored in the memory.

When a segment arrives, the receiver can not tell its offset within its including message. This information becomes available only when all of the segments comprising the message have arrived. While these segments are arriving, the receiver updates an *end_of_message* variable, which holds the largest bit address within the address space where the arriving segment are stored. When all

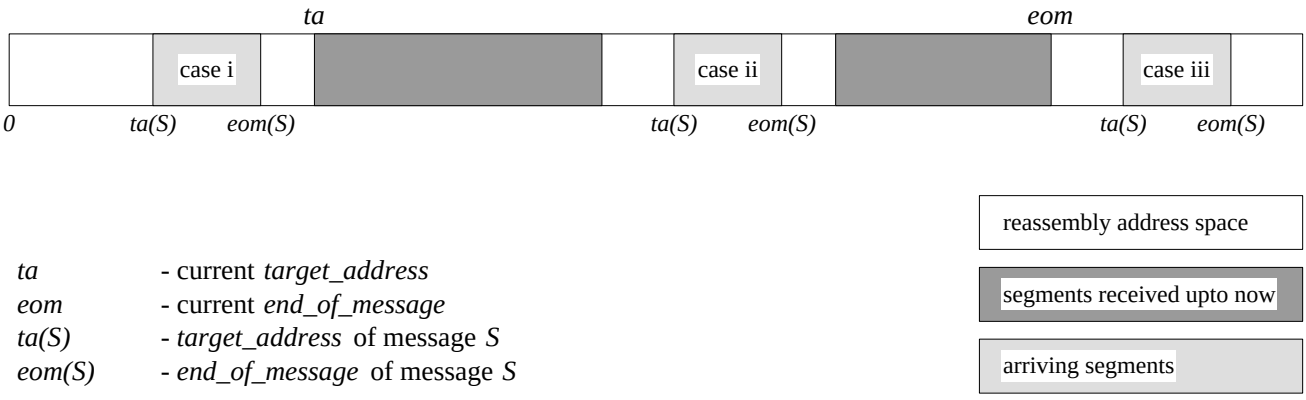

Figure 3: Segment arrival example

segments have arrived, $end\_of\_message$ holds the bit address of the end of the message, where the least significant coefficient of the polynomial corresponding to the message is stored. The offsets l_i, as in Eq. (2), are with respect to that address.

The receiver also maintains the r-bit variable crc, or $crc(x)$, depending on the context, which accumulates the contributions of the arriving segments to the message CRC. At any given moment, crc reflects the CRC value calculated for a message which spans from the smallest $target\_address$ received thus far down to the current $end\_of\_message$. Any holes of missing data are considered as zeros. Upon arrival of a new segment, crc is updated to reflect the new contribution, as follows.

Upon arrival of the first segment S of message m, whose $target\_address$ is $ta(S)$ and whose length is $|S|$, and thus the largest bit address it is going to occupy is $end\_of\_segment(S) = ta(S)+|S|-1$, the receiver sets

$$crc(x) \leftarrow C(S(x)), \quad \text{and} \quad end\_of\_message \leftarrow end\_of\_segment(S)$$

where $C(S(x))$ is the block of r CRC bits computed for S, as in Eq. (1).

For every other segment S that arrives, the receiver updates its variables as follows.

If $ta(S) > end\_of\_message$ (Figure 3, case iii), the message spans into larger addresses than thought thus far, and we need to increase the offsets l_i of all segments that arrived so far by $d = eos(S) - end\_of\_message$. By Equation (3), this means the updating

$$crc(x) \leftarrow (x^d\, crc(x)) \bmod g(x)$$

which, as described in Section 3, can be carried out in r clock cycles. The contribution of S itself

is calculated in $|S|$ clock cycles:

$$crc(x) \leftarrow crc(x) + C(S(x))$$

Finally, the receiver updates $end\_of\_message \leftarrow end\_of\_segment(S)$.

If $ta(S) < end\_of\_message$ (Figure 3, cases i and ii), which by our non-overlapping assumption also means $end\_of\_segment(S) < end\_of\_message$, there is no reason to update the largest address of the message, nor the offsets of the segments that arrived thus far. The contribution of S is added in $|S| + r$ clock cycles as in Equation (3), with $d = end\_of\_message - end\_of\_segment(S)$ being the offset from the current $end\_of\_message$:

$$crc(x) \leftarrow crc(x) + \left(x^d\, C(S(x))\right) \bmod g(x)$$

Once all the segments have arrived and have been processed, their contributions reflect their correct offsets within the message (which is now clear), and by Eq. (3), *crc* holds the message's CRC. Altogether, the total time for processing all the segments comprising a message m of length $|m|$, which is broken up to s segments, is $O(|m| + r\, s) = O(|m|)$, as with the traditional, one pass computation.

The hardware implementation data path (Figure 4) follows the above logic. Once the segment CRC, $C(S(x))$, is calculated, the segment *target_address* is compared with *end_of_message*. This sets the *select* line that controls the muxes and determines which value is to be corrected by the x^d CRC unit. The corrected results – *crc* or $C(S(x))$ – is added to $C(s(x))$ or *crc* respectively, and stored back into the *crc* register.

5 Conclusion

We presented a scheme by which a message CRC is computed at the receiver side by the accumulated individual contributions of its constituent segments. The segments arrive at the receiver side independently, in any order. The processing time of an arriving segment is linear in the segment length, and the total computation, for the entire message, is linear in the message length, as is the case with the traditional computation, which first assembles all the segments, and then makes one pass on their concatenation. The extra memory and logic needed is fixed, independent of the message or the segment length. It is a function of r, the degree of the generator polynomial of the cyclic redundancy code used.

In terms of [1], our scheme does lend itself to a strictly layered modular protocol stack. The layer that processes the individual segments can forward to the upper layer, the layer that processes whole messages, along with each segment the contribution of the segment to the message CRC. The upper layer will accumulate these contributions, thus computing the message CRC.

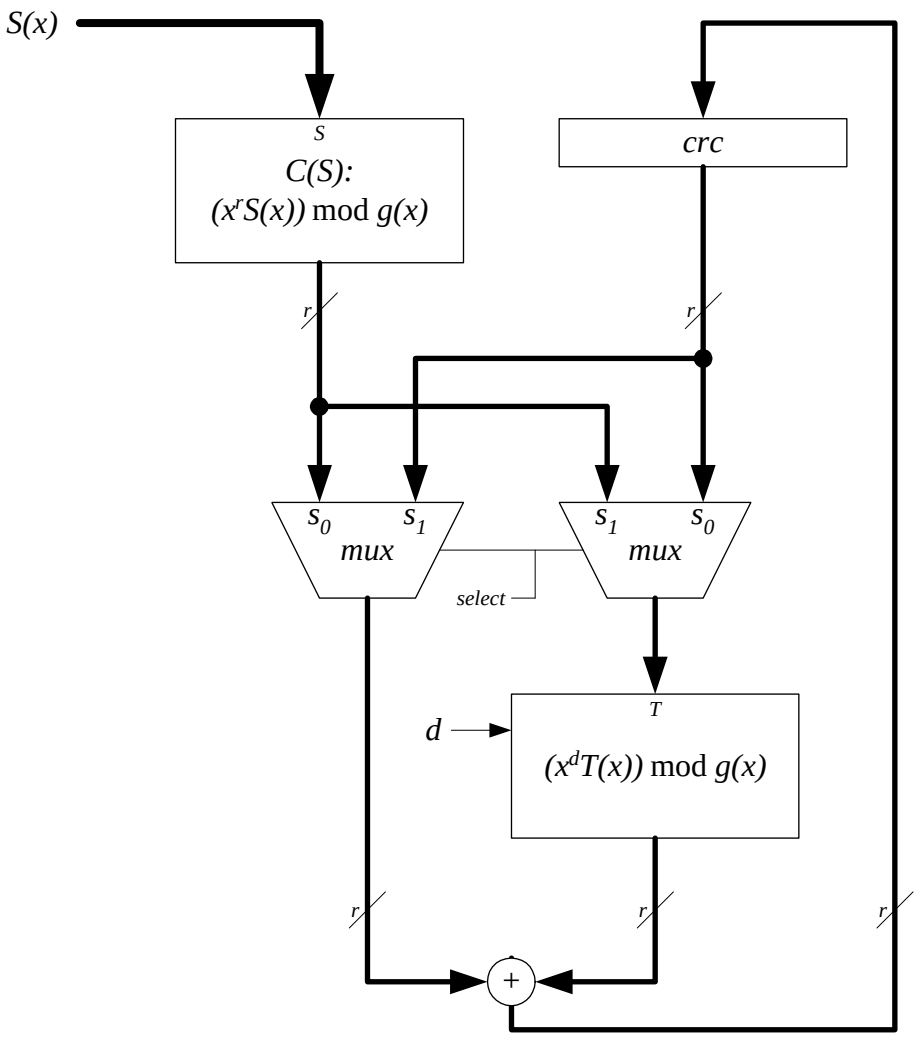

Figure 4: Implementation data path

The scheme we presented consumes $|S| + r$ clock cycles for the processing of an arriving segment S, updating iteratively as in (5). We would like to note here that with additional hardware, whose size, too, is a function of r, the number of clock cycles for the updates needed by the offset l of S can be reduced down to one clock cycle as follows. Upon the arrival of a segment S, whose offset within its including message is l, we launch two computing processes to run concurrently. The first process feeds S into the LFSR in order to obtain $C(S(x)) = \bar{\mathbf{p}}_0$, and the second obtains $\mathbf{T}^l$ by (4). In r or r^2 clock cycles, depending on the implementation used, we can produce the matrix $\mathbf{T}^l$. Upon completion of the two concurrent processes we feed $\mathbf{T}^l$ and $\bar{\mathbf{p}}_0$ into the hardware of Fig. 2, and in one clock cycle obtain the contribution of the arriving segment S to the message CRC.

References

[1] F. Braun, J. Lockwood, and M. Waldvogel. Protocol wrappers for layered network packet processing in reconfigurable networks. *IEEE Micro*, 22(1):66–74, January/February 2002.

[2] F. Braun and M. Waldvogel. Fast incremental crc updates for ip over atm networks. In *Proceeding of 2001 IEEE Workshop on High Performance Switching and Routing*, May 2001.

[3] F.J. McWilliams and N.J. Sloane. *The Theory of Error-Correcting Codes*. North Publishing Co., 1983.

[4] W. Peterson and E. Weldon. *Error-Correcting Codes*. MIT Press, 1972.