shots/electgon_logo_slogan_2020.png

Cyclic Redundancy Check

Mathematical and Hardware Overview

ELECTGON
www.electgon.com
contact@electgon.com

09.10.2020

Contents

Abstract

Although CRC technique is almost found in every digital system on chip, the theory and proof of this processing step is ambiguous for many engineers. This document is trying to declare how CRC works and how it can change and be used with different parameters to check accuracy of data. Dealing with mathematics maybe confusing for some readers but it is necessary to understand how CRC validates checksum of train of binary bits. The purpose of this document is not to explore deep background of the CRC but to explain its theory hopefully in understandable way to help engineers build or adapt their own CRC according to their system needs.

1 Introduction

In digital data processing, Data are handled as chunk of bytes, each byte is 8 bit. Thus any set of data can be considered as sequence of bytes or bits. Transferring these data between two points, requires then to transfer it in the right sequence. Since bits are considered to be independent and a bit sequence is limited by the size of data chunks (or frames, packets, datagrams, etc.), thus we consider the problem to check whether a finite sequence of bits is changed unintentionally, e.g. by transmission errors or faulty storage in memories.

To verify whether data are sent in right sequence or not, we add some redundant information to the bit sequence. Those redundancy information are extracted from transmitted data based on some operations on the bit sequence. The result of these operations is added as redundancy information to the actual data.

The receiver of a message, or the reader of memory input can make the same operations on the received data and compare against the stored redundancy information and decides if an error has occurred in the data or not.

2 Error Detection Techniques

Several techniques can be applied to extract redundancy information, the simplest technique is to append a single bit called the "parity bit" in which we count number of bits with level '1', if it is even number we append 0 to end of data (or vise-versa append 1 for odd Parity bit). Other techniques for computing a checksum are to form the exclusive or of all the bytes in the message, or to compute a sum with end-around carry of all the bytes. CRC (Cyclic Redundancy Check) is yet another efficient technique to detect error occurrence in bit sequence which performs a special operation that can be interpreted as polynomial division. The residual of this division is used as redundancy information. This is what we are going to discuss in this document.

3 CRC

Cyclic Redundancy Check is a process which is based on division between two binary vectors: first binary vector is the data (the dividend) which is divided by another fixed binary number (the divisor). The remainder of this division is the check value that we use to validate accuracy of the data. This operation is most used during transmission of data and when used, it is performed at both sides transmitter and receiver sides. In transmission side, the CRC (which is the remainder of the division) is calculated by dividing the data by the fixed binary number which is usually called the generator polynomial. After performing the division, the resulted CRC is appended to the data and transmitted to its destination.

At receiver side, the receiver either computes the CRC again for the data only (without appended CRC) or computes the CRC for the whole received message (original Data and its appended CRC). In case of computation of CRC for the data only, the receiver shall compare the computed CRC against appended CRC. If they both are the same, this means that the message has been received without errors. In case of computation of CRC for both data and appended CRC, the computation result should be zero. If not, it means that some error has be produced in received message. In fact, this last sentence is not practically correct, that is because the appended CRC is modified a little bit before transmission, it is ones-complemented then appended to the data then transmitted. In this case when calculating the CRC for received data and its appended CRC, the result will give another fixed binary number called residual constant. It is possible to explain CRC better in mathematical way to understand how CRC is calculated and we get at receiver side either constant value of zero. But before that, we have to introduce some arithmetic basics that are used in CRC computation.

4 Modular Arithmetic

Basic arithmetic is built around addition, subtraction, multiplication or division but there are other arithmetic that is derived from this basic arithmetic which is used to perform more developed operations like logarithms which is applied to one operand only but according to the base of the logarithm. Modulo operations is special set of operations also that are applied to one operand only but using the base of the modulo operation, we can define how to perform the Modulo operation.

Modulo operation means to find remainder of division operation between the operand and the base of the Modulo. For example Modulo-two of 5 means to find the remainder when dividing 5 by 2 (5 is the operand, 2 is the base of the modulo operation), the result then is 1. Modulo-three of 8 is 2, since dividing 8 by 3 gives remainder 2. Modulo-five of 15 is 0, since dividing 15 by 5 gives no remainder.

In Modular Arithmetic we perform also addition, subtraction and multiplication. Modular division is avoided. Modular addition means finding Modulo of two or more added numbers. Modular subtraction means to find Modulo of two or more subtracted numbers. Modular Multiplication means finding Modulo of two or more multiplied numbers. Modular division doesn't exist as it is ignored because we need to guarantee that the quotient (of divided numbers before Modulo operation) is an integer.

5 CRC Arithmetic

Since CRC is a digital signal processing scheme, it has its own rules when applying mathematical operations on group of digital bits. These rules are derived from Modular Arithmetic in a way that allows appending some bits or inverting some bits. Since we are talking about digital bits, the CRC Arithmetic is derived from modulo-2 arithmetic with the following rules:

- The binary numbers (dividend & divisor) are considered as polynomials. Therefore, when we add or subtract, we add or subtract each corresponding bits. When we multiply or divide, we multiply or divide polynomials together.
- Second rule is Addition or Subtraction is performed as modulo-2 addition or subtraction, and remember if we have a binary vectors, we add or subtract each corresponding bit together.

Lets now explain first rule which tells that binary numbers shall be considered as polynomials. Assume we have binary number 10111. This is 5 digits number, so it will be represented as a polynomial of the fourth order as $x^4 + x^2 + x + 1$. Then when we need to perform any arithmetic operation, it shall be handled as polynomial arithmetic.

The second rule tells us that addition or subtraction is performed as modulo-2 operation. Addition or subtraction in modulo-2 operation is the following

$0 \pm 1 = 1 \qquad 1 \pm 0 = 1 \qquad 0 \pm 0 = 0 \qquad 1 \pm 1 = 0$

which is exactly performing XOR logic between the two bits.

Example - Multiplication:

$$\begin{aligned}(x^3+x+1).(x^2+1) &= x^5+x^3+x^2+x^3+x+1\\ &= x^5+x^2+x+1\\ &= 100111\end{aligned}$$

Example - Addition:

$$\begin{aligned}(x^3+x+1)+(x^2+1) &= x^3+x^2+x+1+1\\ &= x^3+x^2+x\\ &= 1110\end{aligned}$$

Example - Division:

$$\begin{array}{r|cccccccc}
 & & & x^4 & + & 1 & & \\ \cline{2-8}
x^4+x+1 & x^8+ & x^5 & + & x^2 & +x & & \\ \cline{1-1}
 & x^8+ & x^5+ & x^4 & & & & \\ \cline{2-8}
 & & & x^4 & +x^2 & +x & & \\
 & & & x^4 & + & x & +1 & \\ \cline{4-8}
 & & & & x^2 & + & 1 & \Rightarrow 101
\end{array}$$

6 CRC Mathematics

We can understand that performing CRC computation on the message at receiver and transmitter side we will get the same CRC at both. But how do we get '0' (or constant value) if we computed the CRC for the transmitted message with appended CRC value? To answer that question we will demonstrate the CRC computation in a mathematical way. In general, an n-bit CRC is calculated by representing the data stream as a polynomial M(x). Then this M(x) is multiplied by x^n, where n is the degree of the Generator polynomial G(x). Then dividing the result by the Generator polynomial G(x). The resulting remainder is appended to the polynomial M(x) and transmitted. At the receiver side, The complete transmitted polynomial is then divided by the same Generator polynomial to get the resulting remainder which shall be a constant. Before we explain that mathematically we need to declare some concepts

Multiplication:

Multiplication of M(x) by x^n means trailing n-bits zeros to the end of M(x). For example assume that M(x) is 10111. Which is as a polynomial shall be written as $M(x) = x^4 + x^2 + x^1 + 1$. If we multiplied it by x^4 this will result in

$F(x) = M(x) * x^4$

$F(x) = (x^4 + x^2 + x^1 + 1) * x^4$

$F(x) = x^8 + x^6 + x^5 + x^4$

so the resulted F(x) is represented in binary as 101110000.

Complement:

It is know that to get ones-complement of a binary vector, you need just to invert every bit from 1 to 0 or 0 to 1. In digital logic, this can be performed by XORing each bit with '1'. If we used XOR with '0', no change will take place in the binary vector. This is important to note because we start CRC calculation with some initial value in the CRC registers. If this initial value is sequence of '1', this will result in ones-complementing the first part of the message. So lets assume now that the CRC registers are loaded initially with $L(x)$, which might be $[0\,0 \cdots 0]$ or $[1\,1 \cdots 1]$ but both of them has n bits which is the CRC width.

These two concepts were important before discussing the mathematics behind CRC. Now, as we mentioned previously, the CRC is eventually appended to end of the message. Therefore, first step we do is to shift up the message by n bits in order to prepare location of the CRC and note that n is the width of the CRC.

$$x^n.M(x)$$

If we have our message with length k, the previous multiplication will result in having message length of $k + n$. To start the CRC computation, first step will be XORing the Message with initial value of the CRC registers (why do we XOR first? this will be clear in section"XOR First Serial CRC"). But in order to XOR the initial value ($L(x)$) with the message, we must have same length in both. This means we need to increase length of $L(x)$ to be $k + n$ also

$$x^k L(x)$$

Then XORing both

$$x^n M(x) + x^k L(x) \tag{1}$$

Next Operation is to divide by the Generator polynomial G(x) as a modulo2 division, which can be described mathematically as follows

$$[x^n M(x) + x^k L(x)]/G(x) = Q(x) + R(x)/G(x) \tag{2}$$

where $Q(x)$ are a quotient. $R(x)$ are the remainder of the division operation. From that last equation we can understand the R(X) is the CRC value, but actually we will not take it barely because some implementations of the CRC requires to take ones-complement of the remainder. So we will consider the CRC as follows:

$$CRC_{tx} = L(x) + R(x) \tag{3}$$

$L(x)$value will be according to the implementation value. The transmitted message T(x) can be represented by

$$T(x) = x^n M(x) + CRC_{tx} \tag{4}$$

Note that here$T(x)$ is $k+n$ bits. substituting CRC as in 3 we can write T(x) as:

$$T(x) = x^n M(x) + L(x) + R(x) \tag{5}$$

At the receiver, we are interested in obtaining the remainder of the received message so that we make sure if it has been received correctly or not. So the same CRC technique is applied at the receiver.

$$x^n T(x) + x^{k+n} L(x) \tag{6}$$

then we intent to get remainder of modulo2 division by the Generator polynomial.

$$CRC_{rx} = rem\left\{[x^n T(x) + x^{k+n} L(x)]/G(x)\right\} \tag{7}$$

which can be rewritten as

$$CRC_{rx} = rem\left(x^n [T(x) + x^k L(x)]/G(x)\right) \tag{8}$$

Using 5 we will have:

$$CRC_{rx} = rem\left(x^n [x^n M(x) + x^k L(x) + L(x) + R(x)]/G(x)\right) \tag{9}$$

Taking the divisor inside

$$CRC_{rx} = rem\left(x^n [\{[x^n M(x) + x^k L(x)]/G(x)\} + \{L(x)/G(x)\} + \{R(x)/G(x)\}]\right) \tag{10}$$

Using equation 2 we can now rewrite as

$$CRC_{rx} = rem\left(x^n [\{Q(x) + R(x)/G(x)\} + \{L(x)/G(x)\} + \{R(x)/G(x)\}]\right) \tag{11}$$

Knowing that $rem(A+B+C) = rem(A) + rem(B) + rem(C)$

$$CRC_{rx} = rem\left(x^n \{Q(x) + R(x)/G(x)\}\right) + rem\left(x^n \{L(x)/G(x)\}\right) + rem\left(x^n \{R(x)/G(x)\}\right) \tag{12}$$

It should be clear that remainder of first term is $R(x)$. Last term should result also in $R(x)$ as remainder as $R(x)$ is a polynomial of degree $n-1$ (at maximum) while $G(x)$ is a polynomial of degree n.

$$CRC_{rx} = x^n R(x) + rem\left(x^n \{L(x)/G(x)\}\right) + x^n R(x) \tag{13}$$

Since we are using modulo2 arithmetic, the term $R(x)$ will cancel each other

$$CRC_{rx} = rem\left(x^n \{L(x)/G(x)\}\right) \tag{14}$$

This means that the CRC operation at the receiver side depends only on $L(x)$ and $G(x)$. The resulted remainder is the remainder of the following modulo2 division

$$x^n L(x)/G(x) \tag{15}$$

which is a constant value known as residual value. This explains why we always have a constant value that shall be obtained at receiver side no matter what are the content of the message. In appendix there is MATLAB or Octave code that can be used to make modulo2 division and to obtain this residual value

7 CRC Parameters

As seen in previous section, CRC computation scheme can be specified with some parameters for example the width of the CRC, width of the polynomial, residual value. There are other contexts discussing other parameter of a CRC method. These parameters depend actually on the implementation method of the CRC, for example ones-complemented initial value, reversed CRC. So basically we can say that the CRC has some basic parameters and other implementation parameters. All in all, the following parameters define a CRC scheme.

- **Width**: This is the basic identifier of a CRC scheme which expresses width of the CRC value, i.e. how many bits the CRC is.
- **Polynomial**: The value of the divisor determines also the CRC scheme which is the polynomial. If the CRC width is n-bit, then the polynomial width is (n+1) bit, or we can say it has n+1 coefficients and it is of degree n. Choosing the polynomial value is a big research topic, but the theoretical idea is based on the concept of prime number. Since prime numbers don't have underlying components, so dividing by prime numbers should give more common factors than non-prime numbers, this gives less probability to have same CRC for different numbers. In practice, it is not necessary to choose the polynomial to represent prime number, the main concern is to have better opportunity for error detection. Therefore, the polynomial is chosen to that purpose.
- **Residual Value**: At receiver side, in which we perform same CRC operation, we call the result of the operation then the residual value that gives indication if we have received the message correctly or not. For example the residual value for CCITT CRC-32 is C704DD7B, so any received message should give this value.
- **Initial Value**: When looking at implementation of the CRC procedure, we will find the we perform XOR operation between the message (dividend) and remaining of division (eventually the CRC). So in computation we perform repetitive XOR between Message variable and CRC variable until we finish all bits of the message. The CRC variable can be initialized to certain value (usually it is either 0x00 or 0xFF) before starting the computation.
- **Reverse Input**: We just mentioned that the CRC calculation is performed via iterative XOR operation. Usually each iteration step is performing the XOR operation byte by byte. Some computations requires that each byte of the message (dividend) shall be reversed. i.e. swap bit sequence to start with LSB instead of MSB.

- **Reverse CRC**: Some CRC computations require also that the final obtained CRC value shall be reversed also so that it can be ready for transmission.
- **Complement CRC**: Some computations require to ones-complement the obtained CRC value before transmission.

8 Hardware Implementation

Processing of binary data is usually done in hardware level, therefore the implementation of CRC processing will be discussed here in hardware point of view. The CRC processing as discussed before is a process of mathematical long division, however we are not interested in the quotient of this division. We are interested in the remainder only which is the CRC value that we intend to obtain. This will enable us to simplify the long division process i.e. we don't need to execute the long division at each step, we need only to execute steps where we apply XOR between the dividend and divisor. This can be discussed as follows, assume the long division example:

$$
\begin{array}{r|ccccccccc}
 & & & & & 1 & 0 & 0 & 0 & 1 \\ \cline{2-10}
10011 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
 & 1 & 0 & 0 & 1 & 1 & \downarrow & \downarrow & \downarrow & \downarrow \\ \cline{2-10}
 & & 0 & 0 & 0 & 1 & 0 & \downarrow & \downarrow & \downarrow \\
 & & 0 & 0 & 0 & 0 & 0 & & & \\ \cline{3-10}
 & & & 0 & 0 & 1 & 0 & 1 & \downarrow & \downarrow \\
 & & & 0 & 0 & 0 & 0 & 0 & & \\ \cline{4-10}
 & & & & 0 & 1 & 0 & 1 & 1 & \downarrow \\
 & & & & 0 & 0 & 0 & 0 & 0 & \\ \cline{5-10}
 & & & & & 1 & 0 & 1 & 1 & 0 \\
 & & & & & 1 & 0 & 0 & 1 & 1 \\ \cline{6-10}
 & & & & & & 0 & 1 & 0 & 1 \\ \cline{7-10}
\end{array}
$$

Procedure steps can be minimized into two steps only since we are not interested in the quotient.

$$
\begin{array}{r|ccccccccc}
\cline{2-10}
10011 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
 & 1 & 0 & 0 & 1 & 1 & \downarrow & \downarrow & \downarrow & \downarrow \\ \cline{2-10}
 & & & & & 1 & 0 & 1 & 1 & 0 \\
 & & & & & 1 & 0 & 0 & 1 & 1 \\ \cline{6-10}
 & & & & & & 0 & 1 & 0 & 1 \\ \cline{7-10}
\end{array}
$$

The key point in that is we look at the MSB after the XOR operation, if this MSB is 0 we drop it and seek for the next bit until we meet '1'. When we find this '1' we shift in n-bits from the original dividend, where n is the length of the CRC (or "length of divisor-1"). If there are no enough bits we stop the division because it means division is finished. But actually in real applications there will be enough bit as any dividend is appended by n 0-bits. In other words, after the first XOR operation we align the divisor (polynomial) with the first '1' we see in XOR result, then perform the next XOR operation.

To interpret previous description in hardware implementation, the process of seeking for the next ’1’ is done simply by shifting the dividend until we find that MSB of the dividend became ’1’. This means that the dividend is maintained in a shift register and XORed with the polynomial only when the MSB of the dividend is ’1’.

Dividend Register: $\boxed{1\quad 0\quad 0\quad 1\quad 0}$ $\dashleftarrow$ $\boxed{0110}$
$\oplus$
Polynomial Register: $\boxed{1\quad 0\quad 0\quad 1\quad 1}$

Dividend Register: $\boxed{0\quad 0\quad 0\quad 1\quad 0}$ $\dashleftarrow$ $\boxed{110\_}$
$\circ$
Polynomial Register: $\boxed{\cancel{1}\quad \cancel{0}\quad \cancel{0}\quad \cancel{1}\quad \cancel{1}}$

Dividend Register: $\boxed{0\quad 0\quad 1\quad 0\quad 1}$ $\dashleftarrow$ $\boxed{10\_\_}$
$\circ$
Polynomial Register: $\boxed{\cancel{1}\quad \cancel{0}\quad \cancel{0}\quad \cancel{1}\quad \cancel{1}}$

Dividend Register: $\boxed{0\quad 1\quad 0\quad 1\quad 1}$ $\dashleftarrow$ $\boxed{0\_\_\_}$
$\circ$
Polynomial Register: $\boxed{\cancel{1}\quad \cancel{0}\quad \cancel{0}\quad \cancel{1}\quad \cancel{1}}$

Dividend Register: $\boxed{1\quad 0\quad 1\quad 1\quad 0}$ $\boxed{\_\_\_\_}$
$\oplus$
Polynomial Register: $\boxed{1\quad 0\quad 0\quad 1\quad 1}$

Dividend Register: $\boxed{0\quad 0\quad 1\quad 0\quad 1}$ $\boxed{\_\_\_\_}$
$\circ$
Polynomial Register: $\boxed{1\quad 0\quad 0\quad 1\quad 1}$

9 Serial CRC

Previous steps show that to implement hardware for CRC calculation we need at least n-bit shift register, where n is the degree of the generator polynomial. If we tried to implement previous description literally, the hardware logic will be as in figure 1.

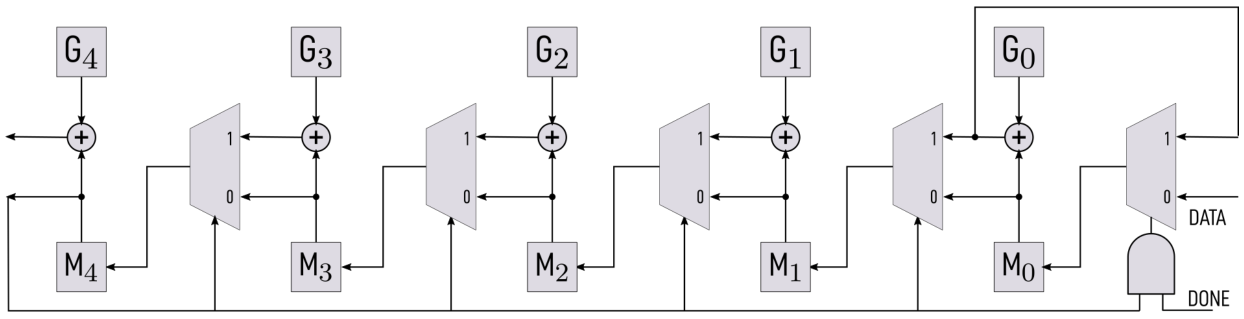

Figure 1: Initial CRC HW Circuit

Then, first thing we can conclude from figure 1 is output of last XOR operation between (G_4 and M_4) is not used anywhere, therefore we can definitely ignore this operation and dismiss G_4 totally from out calculation. This is explains why we always discard MSB bit of the polynomial equation although it is part of its definition. The new hardware logic will be as in figure 2.

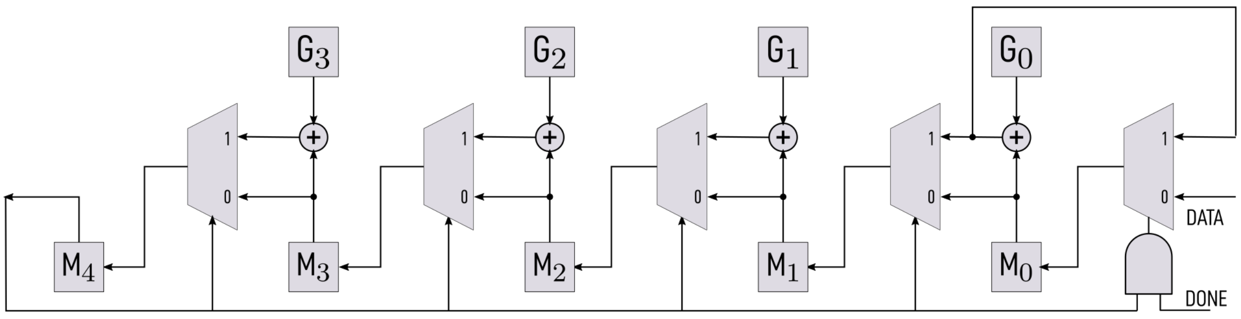

Figure 2: Remove MSB from Polynomial

More simplification can be done in the hardware implementation if we know the polynomial (and indeed we know). In our example here the polynomial is 10011. After discarding MSB we get 0011 which means G_3 is 0, G_2 is 0, G_1 is 1, G_0 is 1. Knowing that XORing any value with 0 gives the same value, we can simplify the hardware of G_3 and G_2 as the its multiplexer will result always in value of M_3 and M_2. This will result in hardware of figure 3.

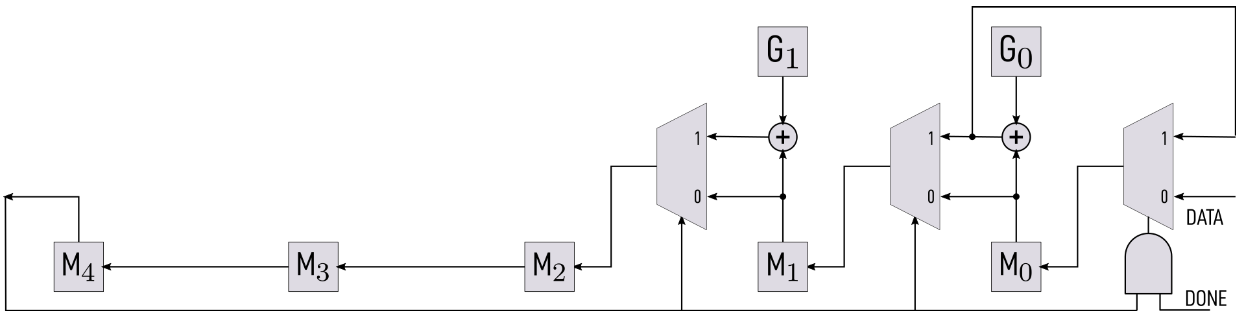

Figure 3: Simplify Polynomial Terms of Value 0

In previous step we made simplification based on the fact that XORing anything with 0 gives the same value. On the other side, XORing any value with 1 gives inverted value. This can help in simplifying the circuit to be as in figure 4 in which we don't need any registers for storing values of the polynomial.

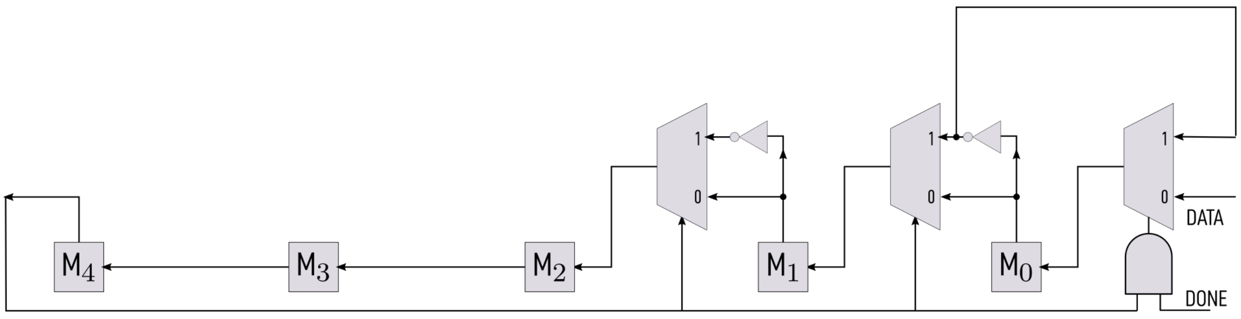

Figure 4: Simplify Polynomial Terms of Value 1

Looking into remaining Multiplexer units, we can see that they result in value of M_1 (or M_0) in case of M_4 is 0; and they result in inverted value of M_1 (or M_0) is case of M_4 is 1 which is exactly the logic of XOR operation. So we can replace these two multiplexers with simply two XOR as in figure 5.

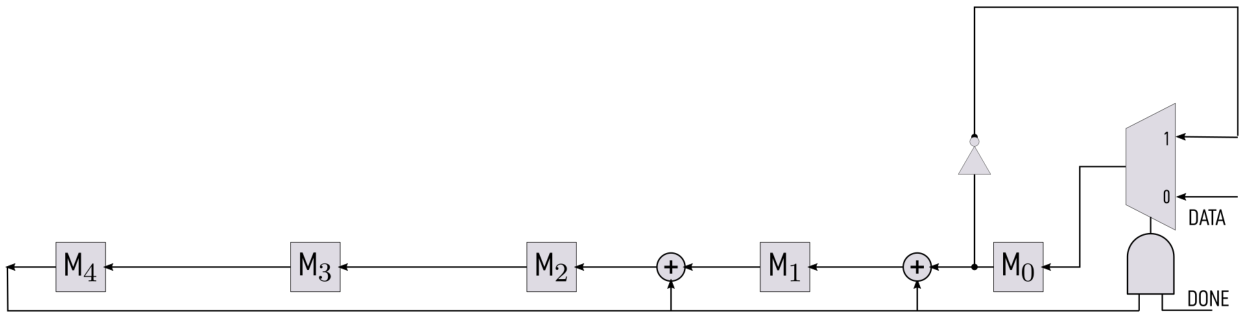

Figure 5: Replace MUX with XOR

This circuit seems not a practical one, since we always drop the MSB (M_4 in our example) of the registers to obtain the correct result.M_4 is used only to know when to shift only or to shift and XOR. In addition there is some more logic before M_0 which is used only at last step in which we don't shift and XOR but XOR only. There should be more simplification for this circuit.

10 XOR First Serial CRC

Previously we discussed basic CRC operation as the following

$$
\begin{array}{r|ccccccccc}
\underline{10011} & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
 & 1 & 0 & 0 & 1 & 1 & \downarrow & \downarrow & \downarrow & \downarrow \\
\hline
 & & & & & 1 & 0 & 1 & 1 & 0 \\
 & & & & & 1 & 0 & 0 & 1 & 1 \\
\cline{6-10}
 & & & & & & 0 & 1 & 0 & 1 \\
\cline{7-10}
\end{array}
$$

Because the MSB in all steps is trimmed and not used after the XOR operation, this operation can yet be enhanced to avoid need of the MSB as follows

$$
\begin{array}{r|cccccccccc}
\underline{10011} & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
 & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \\
 & 1 & 0 & 0 & 1 & 1 & \downarrow & \downarrow & \downarrow & \downarrow & \\
\hline
 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & \\
 & & & & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & & \\
 & & & & 1 & 0 & 0 & 1 & 1 & & \\
\cline{5-11}
 & & & & 0 & 0 & 1 & 0 & 1 & & \\
\cline{5-11}
\end{array}
$$

Which means that we don't need to shift the dividend until its MSB to align MSB of the dividend with the MSB divisor. Instead, we can predict that the MSB of the dividend is going to be '1' if the (MSB-1) bit was '1' in the preceding shift operation. In that way we can eliminate the need for the MSB of the dividend and we can align (MSB-1) of the dividend with the divisor. This means that we do the XOR operation before shifting operation. To know how this will save one more register in the circuit, lets describe the division operation according to our modified process.

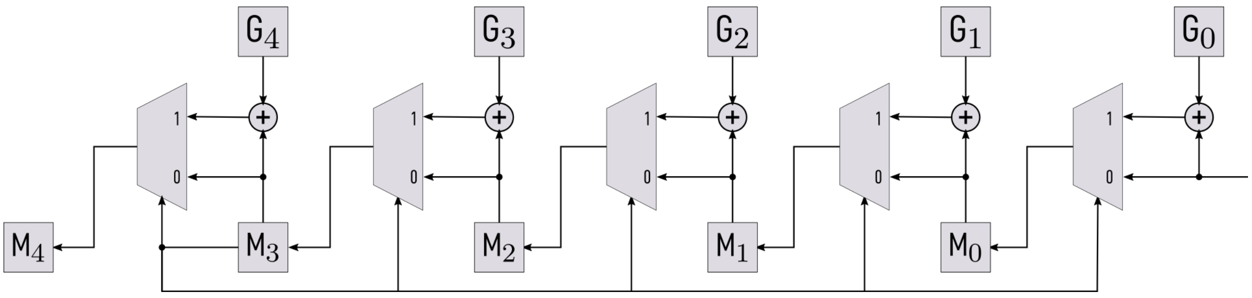

Figure 6: XOR Before Shift

As we can see in figure 6, M_4 is not affecting anything now in the circuit so we can remove it and G_4 as well.

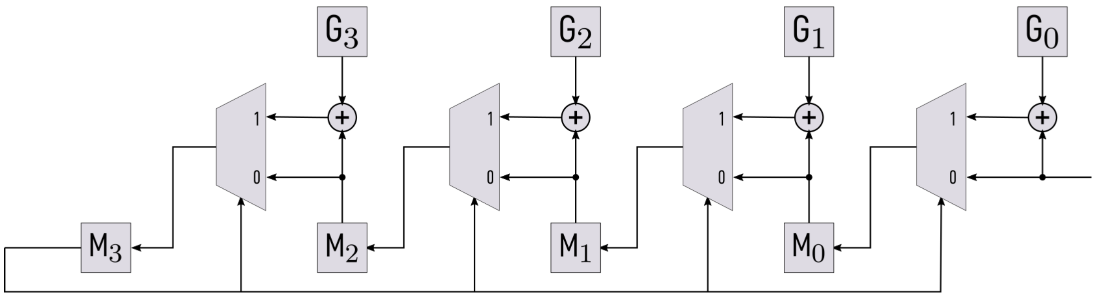

Figure 7: Trim MSB of Polynomial and Message

We can follow same previous simplification to reach eventually circuit in figure 8. The obtained circuit here is called Linear Feedback Shift Register (LFSR).

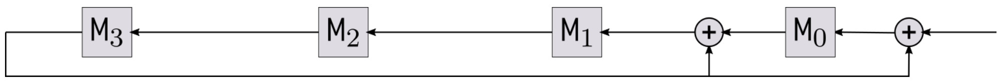

Figure 8: Internal LFSR

It is clear then that this circuit eliminates also the extra logic before M_0.

11 External Linear Feedback Shift Register

We have seen in previous analysis that when we predicted '1' at (MSB-1) position, we were able to enhance the circuit. The question now: what if we predicted coming '1' at LSB position, will this optimize the used hardware circuit?. The answer is yes, it will enhance the performance not in terms of used logic, but in terms of number of execution cycles.

Previous circuit of figure 8 is used to calculate CRC of chunk of data provided to this circuit serially. But in real applications, this chunk of data is appended with n Zeros (multiplied by x^n) where n is the degree of the polynomial as we have seen in CRC Mathematics section. Not only but also in real applications we don't compute the CRC of the Message at once, instead we divide the Message into chunks of sub-messages each of width w. So if we have Message $M(x)$, we can represent it as:

$$M(x) = \sum_{c=1}^{k/w} S_c(x) \tag{16}$$

Where $S_c(x)$ is sub-message of $M(x)$ each with specific width w (8-bit for example), k is length of message $M(x)$, so we will have number of chunks = $\frac{k}{w}$. But note that each $S_c(x)$ has specific location in $M(x)$. i.e. So it is better to decompose that and represent $M(x)$ as:

$$M(x) = \sum_{c=1}^{k/w} x^{(c-1).w}.s_c(x) \tag{17}$$

Where $x^{(c-1).w}$is shifting each sub-message to its correct position. But to add CRC to this message we multiply it with x^n

$$\begin{aligned} M(x).x^n &= \sum_{c=1}^{k/w} \{x^{(c-1).w}.s_c(x)\}.x^n \\ &= \sum_{c=1}^{k/w} x^{(c-1).w+n}.s_c(x) \end{aligned} \tag{18}$$

So if we are going to use circuit of figure 8 we have to take into consideration that this circuit will take $(c.w + n)$ cycles until it finishes. As a side note if we used circuit of figure 5 it will take $(c.w + n + 1)$ cycles to finish.

Since it is a LFSR circuit, we can use another form of this circuit which is called External (or Fibonacci) LFSR in which we try to find equivalent circuit to circuit of figure 8. The circuit in figure 8 is called Internal (or Galois) LFSR.

In External LFSR, we can provide the data chunk serially without the need to append n Zeros. The architecture of External LFSR results in the needed CRC as if we appended n Zeros to the data chunk. Therefore the External LFSR will take only (length of data chunk) cycles to complete CRC calculation. To understand that, lets use previous example to calculate its CRC after appending Zeros to the message.

$$\begin{array}{r|ccccccccccccc}
10011 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
 & 1 & 0 & 0 & 1 & 1 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\hline
 & & & & & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & 1 & 0 & 0 & 1 & 1 & \downarrow & \downarrow & \downarrow & \downarrow \\
\hline
 & & & & & & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
 & & & & & & & 1 & 0 & 0 & 1 & 1 & \downarrow & \downarrow \\
\hline
 & & & & & & & & 0 & 1 & 1 & 1 & 0 & 0 \\
 & & & & & & & & & 1 & 0 & 0 & 1 & 1 \\
\hline
 & & & & & & & & & & 1 & 1 & 1 & 1
\end{array}$$

Using circuit in figure 8 we can expect that it takes 13 cycles until we get the CRC as declared below

Step	Remainder				Serial Input Stream												
0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0
1	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	
2	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0		
3	0	1	0	0	1	0	0	1	1	0	0	0	0	0			
4	1	0	0	1	0	0	1	1	0	0	0	0	0				
5	0	0	0	1	0	1	1	0	0	0	0	0					
6	0	0	1	0	1	1	0	0	0	0	0						
7	0	1	0	1	1	0	0	0	0	0							
8	1	0	1	1	0	0	0	0	0								
9	0	1	0	1	0	0	0	0									
10	1	0	1	0	0	0	0										
11	0	1	1	1	0	0											
12	1	1	1	0	0												
13	1	1	1	1													

This can be enhanced if we consider External LFSR depicted in figure 9 which is very similar to Internal LFSR except the XOR gates is controlled by output of first XOR operation.

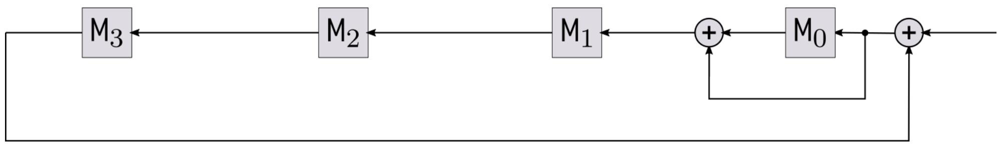

Figure 9: External LFSR

The design of this circuit can be understood from next section which is discussing parallel CRC and how to compute CRC for part of the message. But for the moment we need to know that this circuit can calculate the CRC in 9 cycles which is the length of our data as shown below

Cycle	Remainder				Serial Input Stream								
0	0	0	0	0	1	0	0	1	0	0	1	1	0
1	0	0	1	1	0	0	1	0	0	1	1	0	
2	0	1	1	0	0	1	0	0	1	1	0		
3	1	1	0	0	1	0	0	1	1	0			
4	1	0	0	0	0	0	1	1	0				
5	0	0	1	1	0	1	1	0					
6	0	1	1	0	1	1	0						
7	1	1	1	1	1	0							
8	1	1	1	0	0								
9	1	1	1	1									

This External LFSR is commonly used in case we need to provide the input stream bit by bit as it is optimized in terms of number of used registers and execution cycles. For example, CRC-32 used in Ethernet packets has the polynomial 104C11DB7. CRC circuit for such polynomial can be built using circuit in figure 10.

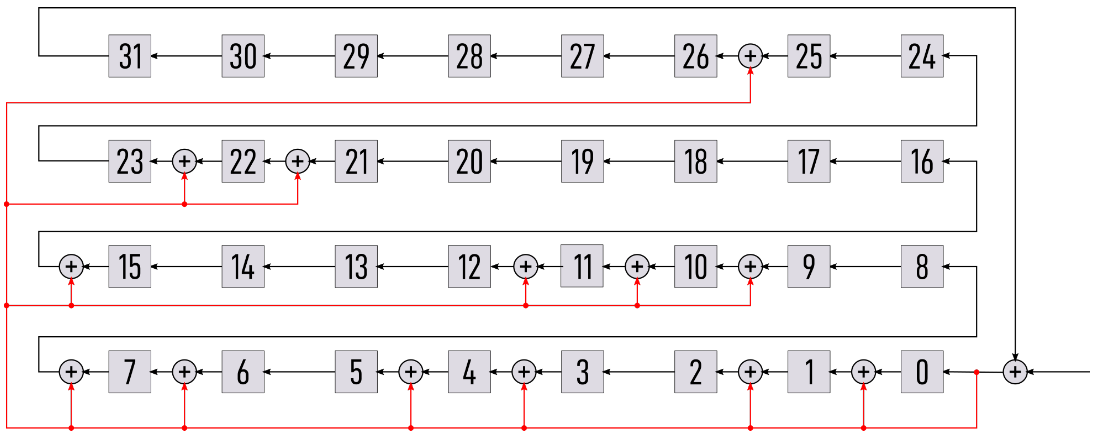

Figure 10: CRC32 LFSR

12 Parallel CRC

In real applications received messages packets are parsed and stored in internal FIFOs for further processing.This means that if we need to check CRC of these received messages, it will not be efficient if we provided this message to the CRC logic serially. Providing this as chunk of data will be more practical. Therefore, we need to have logic that can process chunk of data (for instance 8 bit of data) instantly.

Declaration of Parallel CRC is based considering the LFSR as discrete-time, linear time-invariant system which is modeled by certain equation but in our context here we will discuss this equation in a reverse way. i.e. we will reach to this equation starting from the proposed hardware depicted in figure 9.

Lets describe the state of circuit in figure 9 in a matrix form

$$M\,[t] \quad = \quad \begin{bmatrix} M_3[t] \\ M_2[t] \\ M_1[t] \\ M_0[t] \end{bmatrix}$$

Where $M_3[t], M_2[t], M_1[t], M_0[t]$ are state of each register at cycle t. Now lets derive next state with respect to this state t.

$$M_3[t+1] = M_2[t]$$

$$M_2[t+1] = M_1[t]$$

$$M_1[t+1] = M_0[t] \oplus M_3[t] \oplus d$$

$$M_0[t+1] = M_3[t] \oplus d$$

where d is serial input bit before first XOR gate. Writing $M[t+1]$ in form of matrix will be:

$$\begin{aligned} M[t+1] &= \begin{bmatrix} M_3[t+1] \\ M_2[t+1] \\ M_1[t+1] \\ M_0[t+1] \end{bmatrix} = \begin{bmatrix} M_2[t] \\ M_1[t] \\ M_0[t] \oplus M_3[t] \oplus d \\ M_3[t] \oplus d \end{bmatrix} \\ &= \begin{bmatrix} M_2[t] \\ M_1[t] \\ M_0[t] \oplus M_3[t] \\ M_3[t] \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ d \\ d \end{bmatrix} \end{aligned}$$

In last step we separated registers coefficients M from data coefficient d. This form is better represented as follows

$$\begin{aligned} M[t+1] &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} . \begin{bmatrix} M_3[t] \\ M_2[t] \\ M_1[t] \\ M_0[t] \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} .d \\ M[t+1] &= F.M[t] \oplus G.d \end{aligned} \tag{19}$$

Where F is a $n \times n$ matrix; n is degree of the generator polynomial (or CRC width). G is a $1 \times n$ matrix. So what we can understand from equation 19 that we can know next state with respect to current state. Equation 19 models our LFSR. In some contexts, F is called identity matrix which can be constructed for a LFSR if we know its generator polynomial. If the generator polynomial is described as $g_n, g_{n-1}, ..., g_1, g_0$, then we construct matrix F by

$$F = \begin{bmatrix} g_{n-1} & 1 & 0 & \cdots & 0 \\ g_{n-2} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & 0 & 0 & \cdots & 1 \\ g_0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

So first column represents the polynomial without its MSB, other columns will have value 1 diagonally. Last row consists of LSB of the polynomial then zeros. For sake of further simplification, we will denote here F as:

$$F = [A_0 A_1 A_2 \cdots A_{n-1}]$$

where

$$A_0 = \begin{bmatrix} g_{n-1} \\ g_{n-2} \\ \vdots \\ g_1 \\ g_0 \end{bmatrix} \tag{20}$$

$$A_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{21}$$

$$A_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{22}$$

$$A_{n-1} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix} \tag{23}$$

Lets define also A_n although it is not used in F

$$A_n = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{24}$$

Matrix G represents actually the polynomial if we consider External LFSR. So in case of External LFSR G is described by

$$G = \begin{bmatrix} g_{n-1} \\ g_{n-2} \\ \vdots \\ g_1 \\ g_0 \end{bmatrix} \tag{25}$$

In case of Internal LFSR, G is different and it should be

$$G = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{26}$$

So far we have concluded from equation 19 the register content at time $t+1$ with respect to time t. The difference between times t and $t+1$ is we have provided only 1 bit to the system. If we provided 2 bits, this means that we are now at time $t+2$. If we provided w bits, it means we are now at time $t+w$. Thus, if we got content of the system at time $t+w$ with respect to t, it means that we can provide w bits to the system at time t and we can calculate the CRC by substituting with the content of the system at time $t+w$. This can be explained mathematically as below:

$$M(t+1) = F.M(t) \oplus G.d_t \tag{27}$$

Since it is time-invariant system and only serial input is changing in each time, we can then write:

$$\begin{aligned} M(t+2) &= F.M(t+1) \oplus G.d_{t+1} \\ &= F.F.M(t) \oplus F.G.d_t \oplus G.d_{t+1} \\ &= F^2.M(t) \oplus (F.G.d_t \oplus G.d_{t+1}) \end{aligned} \tag{28}$$

Similarly at time $t+3$ we can get

$$\begin{aligned} M(t+3) &= F.M(t+2) \oplus G.d_{t+2} \\ &= F.F.F.M(t) \oplus F.F.G.d_t \oplus F.G.d_{t+1} \oplus G.d_{t+2} \\ &= F^3.M(t) \oplus (F^2.G.d_t \oplus F.G.d_{t+1} \oplus G.d_{t+2}) \end{aligned} \tag{29}$$

Equations 27, 28, 29 can be better formulated as

$$M(t+1) = F.M(t) \oplus ([G] \,.\, [d_t]^T) \tag{30}$$

$$M(t+2) = F^2.M(t) \oplus ([F.G\ G] \,.\, [d_t d_{t+1}]^T) \tag{31}$$

$$M(t+3) = F^3.M(t) \oplus (\left[F^2.G\ F.G\ G\right] . \, [d_t d_{t+1} d_{t+2}]^T) \tag{32}$$

from which we can generalize solution of equation 19 as:

$$M(i) = F^i.M(0) \oplus \left[F^{i-1}.G \cdots\ F.G\ G\right].\left[D(0)\ \cdots\ D(i-1)\right]^T \tag{33}$$

where $D(0)$ is first bit provided to the system(d_t), $D(1)$ second bit (d_{t+1}) and so on.

Equation 33 may look like complex and hard to calculate specially it has a lot of matrices. To simplify it lets consider equations 21, 22, 23 with which we will add trivial additions to equations 30, 31, 32 to become

$$M(t+1) = F.M(t) \oplus ([G \mid A_1\ A_2 \cdots\ A_{n-1}].[d_t \mid 0\ 0 \cdots\ 0]^T) \tag{34}$$

$$M(t+2) = F^2.M(t) \oplus ([F.G\ G \mid A_2 \cdots\ A_{n-1}].[d_t d_{t+1} \mid 0\ 0 \cdots\ 0]^T) \tag{35}$$

$$M(t+3) \quad = \quad F^3.M(t) \oplus (\left[F^2.G\ F.G\ G \mid A_3 \cdots\ A_{n-1}\right].[d_t d_{t+1} d_{t+2} \mid 0\ 0 \cdots\ 0]^T) \tag{36}$$

$$M(n-1) \quad = \quad F^{n-1}.M(0) \oplus (\left[F^{n-2}.G \cdots\ F.G\ G \mid A_{n-1}\right].[d_0 \cdots d_{n-3} d_{n-2} \mid 0]^T) \tag{37}$$

$$M(n) \quad = \quad F^n.M(0) \oplus (\left[F^{n-1}.G \cdots\ F.G\ G\right].[d_0 \cdots d_{n-1}]^T) \tag{38}$$

So what we did is adding A matrices (equations 21,22,23,...) then we added zeros in corresponding position in data $[d_0 d_1 ...]^T$ matrix so that it has no effect on the value. Then we can write equation 33 as:

$$M(i) \quad = \quad \begin{cases} F^i.M(0) \oplus (\left[F^{i-1}.G \cdots\ F.G\ G \mid A_i \cdots\ A_{n-1}\right].[d_0 d_1 \cdots d_{i-1} \mid 0\ 0 \cdots\ 0]^T) & i < n \\ F^i.M(0) \oplus (\left[F^{i-1}.G \cdots\ F.G\ G\right].[d_0 \cdots d_{n-1}]^T) & i = n \end{cases} \tag{39}$$

where n is the width of the CRC. To understand why we did that, lets calculateF^2, F^3 in our example

$$F = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$F^2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} . \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$F^3 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} . \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

According to [4], power of F can be calculated by

$$F^i = \left[F^{i-1} \otimes A_0 \mid first\, n-1\, columns\, of\, F^{i-1}\right] \tag{40}$$

where A_0 is first column of F as denoted in equation 20. If we tried to calculate$F^2.G$, $F^3.G$ this will depend on the considered value of G. If you consider External LFSR, then G will be as in equation 25. Then:

$$F.G = first\, column\, of\, F^2 = second\, column\, of\, F^3 \tag{41}$$

$$F^2.G = first\, column\, of\, F^3 = second\, column\, of\, F^4 \tag{42}$$

Then equation 39 can be simplified to be

$$M(i) = F^i.M(0) \oplus F^i D \tag{43}$$

where $D = [d_0\, d_1\, \cdots\, d_{n-1} \mid 0 \cdots 0]^T$. And when we provide w bits as parallel to the External LFSR, this equation will be:

$$\widehat{M} = F^w.(M \oplus D) \tag{44}$$

where $\widehat{M}$ is next state after M which is the CRC if we provided w bit in one cycle.

In case of Internal LFSR, then G will be as denoted in equation 26. Then:

$$F^{n-1}.G = A_1 \tag{45}$$

$$F.G = A_{n-1} \tag{46}$$

where A_{n-1}, A_1 are as denoted in equations 2321. Then we can simplify equation 33 directly to be

$$M(i) = F^i.M(0) \oplus [A_1 \cdots\, A_{i-1}\, A_i] . [D(0)\, \cdots\, D(i-1)]^T$$

and when we provide w bits as parallel to the Internal LFSR, this equation will be:

$$\widehat{M} = F^w.M \oplus \widetilde{D} \tag{47}$$

where $\widetilde{D} = [0 \cdots 0 \mid d_0\, d_1\, \cdots\, d_{n-1}]^T$.

12.1 Example

Lets now use our example to see how we can build parallel CRC instead of the serial LFSR. In our example we have $n = 4$. So if we going to calculate for $w = 4$ bit parallel data then we need to find F^4 which is:

$$F^4 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

with $\widehat{M} = [\widehat{m_3}\widehat{m_2}\widehat{m_1}\widehat{m_0}]^T$ and $M = [m_3 m_2 m_1 m_0]^T$

If we used External LFSR

$$\begin{aligned} \begin{bmatrix} \widehat{m_3} \\ \widehat{m_2} \\ \widehat{m_1} \\ \widehat{m_0} \end{bmatrix} &= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} m_3 + d_3 \\ m_2 + d_2 \\ m_1 + d_1 \\ m_0 + d_0 \end{bmatrix} \\ &= \begin{bmatrix} m_3 \oplus d_3 \oplus m_2 \oplus d_2 \\ m_2 \oplus d_2 \oplus m_1 \oplus d_1 \\ m_3 \oplus d_3 \oplus m_1 \oplus d_1 \oplus m_0 \oplus d_0 \\ m_3 \oplus d_3 \oplus m_0 \oplus d_0 \end{bmatrix} \end{aligned}$$

note here that order of the parallel data $D = [d_3 d_2 d_1 d_0]^T$ but you can choose any order you like, however you have to be careful of order of resulted CRC which will be discussed in next sub-section. What we can conclude from our example here that we can build the parallel CRC this combination of signals and XOR logic. You can try to build it also using Internal LFSR which will have different combination.

12.2 Input Data Order

Assume a generated LFSR logic with CRC width 8 bit and input parallel data 4 bit. The system maybe expressed then as:

$$\begin{bmatrix} \widehat{m_7} \\ \widehat{m_6} \\ \widehat{m_5} \\ \widehat{m_4} \\ \widehat{m_3} \\ \widehat{m_2} \\ \widehat{m_1} \\ \widehat{m_0} \end{bmatrix} = F^4 . \begin{bmatrix} m_7 + d_3 \\ m_6 + d_2 \\ m_5 + d_1 \\ m_4 + d_0 \\ m_3 \\ m_2 \\ m_1 \\ m_0 \end{bmatrix}$$

here we assumed that user will consider chunk of data as $D = [d_3 d_2 d_1 d_0]^T$. But what if provided chunk of data was considered as the inverse: $D = [d_0 d_1 d_2 d_3]^T$. The LFSR will be described by:

$$\begin{bmatrix} \widehat{m_7} \\ \widehat{m_6} \\ \widehat{m_5} \\ \widehat{m_4} \\ \widehat{m_3} \\ \widehat{m_2} \\ \widehat{m_1} \\ \widehat{m_0} \end{bmatrix} = F^4 . \begin{bmatrix} m_7 + d_0 \\ m_6 + d_1 \\ m_5 + d_2 \\ m_4 + d_3 \\ m_3 \\ m_2 \\ m_1 \\ m_0 \end{bmatrix}$$

which will give different combination of resulted XOR circuit which will give different CRC value accordingly.

12.3 Accumulated CRC Calculation

So far we got the idea of how to calculate CRC for parallel chunk of data. The question now; is this applicable in case of large amount of data (Ethernet packets for instance)? The answer is yes. We can divide for example Ethernet packet (which is usually about 1500 bytes) into data chunk each chunk is 8 bit for example, then calculate the CRC of each 8 bit but without resetting the CRC registers between each chunk. So CRC of each chunk shall be accumulated on the previous chunk CRC. In other words, First data chunk will have initial value in CRC registers as zeros or ones, but second data chunk will have previous CRC as initial value of the registers and so on. In case of External LFSR we can provide each data chunk directly. In case of Internal LFSR, we need to append each data chunk with n-bits of zeros.

Appendix

MATLAB/Octave code to perform polynomial division.

```
clear

pkg load communications

dividend = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, ...
            1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, ...
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, ...
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
divisor =  [1,0,0,0,0,0,1,0,0,1,1,0,0,0,0,0, ...
            1,0,0,0,1,1,1,0,1,1,0,1,1,0,1,1,1];

len_end = length(dividend);
len_sor = length(divisor);

if (len_sor <= len_end)
   len_diff = len_end - len_sor;
   for app_idx = len_sor+1 : len_sor+len_diff
       divisor(:, app_idx) = 0;
   end
else
    printf("Division is not possible, dividend must be longer than divisor\n")
end

if (len_diff >= 0)
    for div_idx = 1:len_diff+1
        if (dividend(div_idx) == 1)
            dividend = xor(dividend, divisor);
            div_res(div_idx) = 1;
        else
            div_res(div_idx) = 0;
        end
        if (div_idx == len_diff+1)
            break;
        end
        divisor = shift(divisor, 1);
        divisor(:,1) = 0;
    end
end

quotient  = dec2hex(bi2de(div_res, 'left-msb'))
remainder = dec2hex(bi2de(dividend, 'left-msb'))
divisor = dec2hex(bi2de(divisor, 'left-msb'))
```

Bibliography

[1] http://einstein.informatik.uni-oldenburg.de/papers/CRC-BitfilterEng.pdf.

[2] Warren, Henry S., Jr “CYCLIC REDUNDANCY CHECK” - http://www.hackersdelight.org/crc.pdf.

[3] https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[4] Giuseppe Campobello, Giuseppe Patane, Marco Russo “Parallel CRC Realization”, 2003.

[5] Wu Chuxiong, Shi Haifeng “Design and implementation of parallel CRC algorithm for fibre channel on FPGA”, 2019.

[6] Dawood Alnajjar, Mauricio Suguiy “A Comprehensive Guide for CRC Hardware Implementation”.

[7] https://www.cl.cam.ac.uk/research/srg/bluebook/21/crc/node2.html.