Cyclic Redundancy Checks (CRC) in Detail

November 2, 2025

Contents

1 Introduction

Cyclic Redundancy Checks (CRCs) are among the most widely used error–detection mechanisms in digital communication and storage systems. They provide a lightweight, mathematically rigorous means of detecting accidental data corruption caused by noise, interference, or physical layer imperfections. CRCs are ubiquitous in networking protocols (e.g., Ethernet, Wi-Fi), storage devices (hard drives, SSDs), embedded systems (CAN bus, automotive ECUs), and peripheral/data interfaces (USB, PCIe, SATA), where reliability and efficiency are paramount.

At a high level, a CRC augments a data frame with a short checksum computed from the message bits using polynomial arithmetic over $\mathbb{F}_2$ (the binary finite field). This checksum enables the receiver to verify data integrity by recomputing and checking for consistency. Unlike naive checksums that simply sum bytes or XOR blocks, CRCs are designed using algebraic properties of polynomials to maximize the probability of detecting real–world error patterns, including burst errors and structured bit flips that commonly occur in digital channels.

Motivation for CRCs in Communication Systems

Modern communication systems must transmit data efficiently while preserving correctness in the presence of noise, timing drift, electromagnetic interference, and imperfect media. CRCs are motivated by the need for a mechanism that:

- Detects a wide class of error patterns with mathematically provable guarantees.
- Is computationally efficient to implement in hardware (shift registers, XOR gates) or software (bitwise operations).
- Adds minimal redundancy relative to the data payload.
- Operates without requiring retransmission or complex decoding steps unless an error is detected.

Because CRC computation reduces to modular polynomial arithmetic over $\mathbb{F}_2$, the circuitry required to implement it is simple and fast, making CRCs ideal for protocols that require extremely high throughput and low latency.

Error Detection vs. Error Correction

CRCs are *error–detecting codes*, meaning they identify whether an error occurred but do not directly correct the erroneous bits. This distinguishes CRCs from *error–correcting codes* (ECC) such as Reed–Solomon or LDPC codes, which can recover corrupted data at the cost of greater computational and bandwidth overhead.

In many systems, detection alone is sufficient and efficient:

- Networking stacks can request retransmission (ARQ / TCP reliability).
- Filesystems or storage controllers can retry physical reads.
- Embedded networks (e.g., automotive CAN) treat corrupted messages as invalid and discard them.

Thus, CRCs play a complementary role alongside error–correcting codes, balancing complexity and reliability depending on system constraints.

Practical Applications

CRCs appear in virtually every modern digital communication and storage system. Representative applications include:

- **Networking:** Ethernet frames, Wi-Fi frames, PPP, HDLC.
- **Storage:** Disk sectors, NAND flash, RAID integrity checks.

- **Embedded/Automotive:** CAN bus, LIN bus, FlexRay, industrial fieldbuses.
- **Peripheral Interfaces:** USB, PCI Express, SATA, NVMe.
- **Digital media transmission:** MPEG transport streams, DSL, DOCSIS.

The ubiquity of CRCs stems from their efficiency, strong mathematical foundations, and suitability for hardware pipelines, making them a cornerstone primitive for reliable digital systems.

2 Mathematical Background

2.1 Binary Arithmetic and Bitwise Operations

Digital systems represent information in binary form, using bits that take values in $\{0, 1\}$. Computations over bitstrings are performed using Boolean logic operations such as AND, OR, and XOR. These operations directly correspond to algebraic rules when working over the finite field $\mathbb{F}_2$, which makes them a natural foundation for CRC computations.

Basic Bitwise Operations

The fundamental bitwise operations used in CRC computation are:

- **AND** ($\wedge$): Produces 1 only if both input bits are 1.

$$0 \wedge 0 = 0, \quad 0 \wedge 1 = 0, \quad 1 \wedge 0 = 0, \quad 1 \wedge 1 = 1$$

- **OR** ($\vee$): Produces 1 if either input bit is 1.

$$0 \vee 0 = 0, \quad 0 \vee 1 = 1, \quad 1 \vee 0 = 1, \quad 1 \vee 1 = 1$$

- **Exclusive OR (XOR)** ($\oplus$): Produces 1 if the input bits are different.

$$0 \oplus 0 = 0, \quad 0 \oplus 1 = 1, \quad 1 \oplus 0 = 1, \quad 1 \oplus 1 = 0$$

The XOR operation is central in CRC arithmetic. It mirrors addition in $\mathbb{F}_2$, where:

$$0 + 0 = 0, \qquad 1 + 0 = 1, \qquad 0 + 1 = 1, \qquad 1 + 1 = 0$$

Since $1 + 1 = 0$ in $\mathbb{F}_2$, XOR also implicitly performs subtraction:

$$a \oplus b = a + b = a - b \qquad (\text{in } \mathbb{F}_2)$$

Thus, XOR implements both addition and subtraction modulo 2 without carries or borrows.

Why XOR Corresponds to Addition Modulo 2

Binary arithmetic within CRC computation is performed over the field $\mathbb{F}_2 = \{0, 1\}$, where all operations are reduced modulo 2. In this algebraic system, addition is defined by:

$$1 + 1 \equiv 0 \pmod 2, \qquad 1 + 0 \equiv 1, \qquad 0 + 0 \equiv 0$$

This is identical to the behavior of XOR. Therefore:

$$a \oplus b \equiv a + b \mod 2$$

This equivalence is crucial. It means that when CRC algorithms perform XOR across aligned bit sequences, they are carrying out polynomial addition in $\mathbb{F}_2[x]$ (polynomials with coefficients in $\mathbb{F}_2$). Because no carries propagate across bit positions, the arithmetic is simple and hardware-friendly.

Implication for CRC Computation

CRC division consists of repeated XOR operations between the shifted message and the generator polynomial. Each XOR operation corresponds to subtracting a multiple of the generator polynomial from the working dividend, exactly as in classical long division, but in the binary field where subtraction equals addition.

This is why CRC circuits can be implemented efficiently using only shift registers and XOR gates, without the need for full multipliers or adders.

2.2 Galois Field $\mathbb{F}_2$

Cyclic Redundancy Checks are built on arithmetic performed in the finite field $\mathbb{F}_2$, the simplest possible field. Understanding CRCs requires recognizing that binary sequences are treated not merely as bit patterns, but as coefficients of polynomials over $\mathbb{F}_2$. This algebraic structure underlies the error–detection guarantees of CRCs and explains why XOR operations implement polynomial arithmetic.

Definition and Properties

The Galois Field $\mathbb{F}_2$ (also denoted GF(2)) is the field consisting of two elements:

$$\mathbb{F}_2 = \{0, 1\}$$

with arithmetic operations defined modulo 2. The addition and multiplication tables are:

$+$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

Key properties:

- $1 + 1 = 0$ (addition modulo 2)
- Each element is its own additive inverse:

$$a + a = 0 \quad \text{for all } a \in \mathbb{F}_2$$

- Multiplication behaves like logical AND, addition like XOR
- $\mathbb{F}_2$ is a field: addition, subtraction, multiplication, and division (by nonzero elements) are all defined

This field structure enables well–defined polynomial arithmetic, which is essential to CRC computation.

Polynomial Representation over $\mathbb{F}_2$

A bitstring can be interpreted as a polynomial whose coefficients lie in $\mathbb{F}_2$. For example, the bitstring

$$110101$$

corresponds to the polynomial

$$1 \cdot x^5 + 1 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x + 1 = x^5 + x^4 + x^2 + 1.$$

In general, a bitstring $b_{n-1}b_{n-2}\dots b_1b_0$ maps to

$$\sum_{i=0}^{n-1} b_i x^i, \qquad b_i \in \mathbb{F}_2.$$

Important observations:

- Bit shifting corresponds to multiplication by x.
- Appending k zeros corresponds to multiplying by x^k.
- CRC division is polynomial division over $\mathbb{F}_2[x]$.

No Carries: Addition = Subtraction = XOR

Arithmetic over $\mathbb{F}_2$ has no carries. For example,

$$1 + 1 = 0 \quad (\text{not } 10_2)$$

Thus:

$$a + b = a - b = a \oplus b$$

for $a, b \in \mathbb{F}_2$. Polynomial addition operates coefficient–wise with this rule, meaning:

$$(x^3 + x + 1) + (x^3 + x^2) = x^2 + x + 1$$

since the x^3 terms cancel. Operationally, CRC division repeatedly performs this kind of cancellation—implemented in hardware or software by XORing bit sequences.

Implications for CRCs

- No carries means that arithmetic is simple and bitwise.
- XOR replaces addition and subtraction.
- Polynomial long division becomes repeated XOR on aligned shifted copies of the generator.
- Efficient hardware implementation using only shift registers and XOR gates.

This algebraic model—bit sequences as polynomials mod 2—is what allows CRCs to detect structured errors far more effectively than ordinary checksums.

2.3 Polynomials Over $\mathbb{F}_2$

In CRC computation, bitstrings are not treated merely as binary numbers, but as polynomials with coefficients in the finite field $\mathbb{F}_2$. This interpretation enables the use of algebraic division to derive a remainder that serves as a compact integrity check value. Understanding this polynomial viewpoint is fundamental to the derivation and guarantees of CRCs.

Mapping Bitstrings to Polynomials

Given a bitstring

$$b_{n-1}b_{n-2}\dots b_1b_0, \qquad b_i \in \{0,1\},$$

we associate it with the polynomial

$$M(x) = \sum_{i=0}^{n-1} b_i x^i \in \mathbb{F}_2[x].$$

Example:

$$1101011011 \quad \longleftrightarrow \quad x^9 + x^8 + x^6 + x^4 + x^3 + x + 1.$$

Important notes:

- The leftmost bit corresponds to the highest power of x.
- Coefficients are added and subtracted modulo 2.
- Each bit position corresponds to a monomial.

This perspective transforms binary manipulation into algebra on polynomials over a field, allowing well-defined division.

Bit Shifts as Multiplication by x^k

Appending k zeros to a bitstring corresponds to multiplication by x^k:

$$b_{n-1}\dots b_0 \to b_{n-1}\dots b_0\,\underbrace{00\dots0}_{k\text{ zeros}}$$

$$\iff \quad x^n + \dots + 1 \mapsto x^k(x^n + \dots + 1)$$

Example:

$$1011 \quad \mapsto \quad 10110000$$
$$(x^3 + x + 1) \cdot x^4 = x^7 + x^5 + x^4.$$

This operation is central to CRC generation: the message is multiplied by x^k, where k is the degree of the generator polynomial.

Long Division in $\mathbb{F}_2$

CRC computation performs polynomial long division:

$$T(x) = M(x)x^k \quad \text{divided by} \quad G(x),$$

where $G(x)$ is the generator polynomial. The remainder $R(x)$ becomes the CRC.

Key properties of division in $\mathbb{F}_2[x]$:

- Subtraction = addition = XOR
- No carries propagate across bit positions
- Division operates by repeatedly canceling highest-degree terms

Operationally, long division over $\mathbb{F}_2$ looks like binary subtraction, but using XOR:

$$(x^5 + x^4 + x^2 + 1) - (x^5 + x + 1) = x^4 + x^2 + x,$$

since identical powers cancel:

$$x^5 - x^5 = 0, \qquad x - x = 0, \qquad 1 - 1 = 0.$$

In bit form, this corresponds to XORing aligned bit sequences during the division process.

Implication for CRC Implementation

This algebraic formulation yields the following implementation principles:

- CRC hardware uses shift registers (for x multiplications) and XOR gates (for polynomial subtraction).
- Software implementations mimic polynomial long division using bit shifts and XORs.
- The CRC remainder is the remainder of a division in $\mathbb{F}_2[x]$, not a numeric remainder in $\mathbb{Z}$.

Thus, the polynomial model is not merely an abstraction: it directly dictates the bitwise mechanics of CRC generation and verification.

3 CRC Fundamentals

3.1 Generator Polynomial

The core of a CRC scheme is its generator polynomial, typically denoted $G(x)$. This polynomial determines the structure of the CRC code, the length of the checksum, and the classes of errors that can be detected. CRC computation consists of dividing the message polynomial (shifted by an appropriate number of bits) by this generator and taking the remainder.

Degree and Structure

A generator polynomial $G(x)$ for an n-bit CRC has degree n, and thus consists of $n+1$ coefficients:

$$G(x) = g_n x^n + g_{n-1}x^{n-1} + \cdots + g_1 x + g_0, \quad g_i \in \mathbb{F}_2.$$

- The degree n determines the number of CRC bits appended to the message.
- $g_n = g_0 = 1$ so that the polynomial is monic and divisible polynomials align correctly.
- The bitstring representation of the generator has length $n+1$ and always begins and ends with 1.

Example (CRC-4 generator):

$$G(x) = x^4 + x + 1 \quad \longleftrightarrow \quad 10011.$$

Example (CRC-32 generator used in Ethernet and many protocols):

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

This corresponds to the bitstring:

`0x04C11DB7`

in hexadecimal (the leading x^{32} term is implicit).

Primitive and Irreducible Polynomials

The performance of a CRC in detecting errors depends critically on the algebraic properties of the generator.

Irreducible Polynomials A polynomial $G(x) \in \mathbb{F}_2[x]$ is *irreducible* if it cannot be factored into nontrivial polynomials over $\mathbb{F}_2$:

$$G(x) = A(x)B(x) \quad \Rightarrow \quad A(x) = 1 \text{ or } B(x) = 1.$$

Using an irreducible polynomial ensures that the resulting code has no short cycles and offers strong detection of many error patterns, including all single-bit errors and many burst errors.

Primitive Polynomials A polynomial is *primitive* if it is irreducible and if x generates the multiplicative group of the field extension $\mathbb{F}_{2^n}$ it defines. Primitive polynomials provide the strongest possible cyclic properties, maximizing the period of the associated linear feedback shift register (LFSR).

In simpler terms:

$$\text{primitive} \Rightarrow \text{irreducible}, \quad \text{but irreducible} \not\Rightarrow \text{primitive}.$$

Why These Matter for CRCs

- Irreducibility ensures no obvious algebraic collisions.
- Primitivity maximizes burst error detection and pseudo-random properties.
- Many widely used CRC generator polynomials are not primitive but are chosen by exhaustive search and analysis to optimize detection of real-world error patterns.

Error Detection Capabilities When $G(x)$ contains factors corresponding to $(x+1)$, it guarantees detection of all odd numbers of bit errors. When $G(x)$ is primitive or specifically structured, it guarantees detection of:

- All single-bit and double-bit errors
- All odd-weight errors (if $(x+1)$ divides $G(x)$)
- Any burst error of length $\leq n$ (where $n = \deg G$)
- Most longer burst and random errors with probability $\approx 2^{-n}$

Thus the generator polynomial is the mathematical mechanism by which CRCs achieve their error detection guarantees.

3.2 CRC Computation Process

The computation of a Cyclic Redundancy Check consists of three primary stages: shifting the message by the degree of the generator polynomial, performing polynomial long division in $\mathbb{F}_2$, and appending the resulting remainder to form the transmitted codeword. These steps ensure that the transmitted polynomial is divisible by the generator polynomial, enabling error detection at the receiver.

Step 1: Append Zeros (Shift)

Let the message bitstring be represented by the polynomial $M(x)$, and let the generator polynomial $G(x)$ have degree n. To prepare the message for CRC computation, we multiply it by x^n:

$$M(x) \longrightarrow M(x)x^n$$

In binary form, this corresponds to appending n zeros to the message:

$$b_{k-1}b_{k-2}\dots b_1b_0 \;\mapsto\; b_{k-1}b_{k-2}\dots b_1b_0\underbrace{00\dots0}_{n\text{ zeros}}$$

This step reserves space for the n-bit remainder that will serve as the CRC value.

Step 2: Polynomial Long Division via XOR

We divide the shifted message $M(x)x^n$ by the generator polynomial $G(x)$ using polynomial long division over $\mathbb{F}_2$:

$$\frac{M(x)x^n}{G(x)}.$$

Key properties of division in $\mathbb{F}_2[x]$:

- Subtraction and addition are identical, both implemented via XOR.
- No carries propagate during addition.
- Division proceeds by canceling the highest-degree term of the current dividend.

Operationally:

- Align $G(x)$ with the highest remaining 1 in the dividend.
- XOR the generator polynomial with the aligned bits.
- Shift the generator to the next position where the leading bit of the working dividend is 1.
- Continue until fewer than n bits remain.

The result of this process is a quotient (discarded) and a remainder $R(x)$, with:

$$\deg(R) < \deg(G).$$

Step 3: Remainder as CRC

The remainder $R(x)$ constitutes the CRC value. The transmitted codeword is formed by appending this remainder to the original (unshifted) message:

$$T(x) = M(x)x^n + R(x)$$

By construction,

$$T(x) \equiv 0 \pmod{G(x)},$$

which means $T(x)$ is exactly divisible by the generator polynomial.

At the receiver, the same division is performed. If no transmission error occurs, the remainder will again be zero. Any nonzero remainder indicates that the received polynomial is not divisible by $G(x)$, signaling data corruption.

Summary

- Append n zeros to reserve space for the remainder.
- Divide by $G(x)$ using XOR to simulate subtraction in $\mathbb{F}_2$.
- Append the n-bit remainder as the CRC.

This procedure ensures that CRC-protected messages belong to a structured algebraic code with well-understood and strong error-detection capabilities.

3.3 Why CRCs Detect Errors

CRCs detect errors by treating corrupted data as a polynomial perturbation and using algebraic divisibility to verify integrity. The key observation is that the transmitted codeword is constructed so that it is divisible by the generator polynomial $G(x)$. Any transmission error results in adding an error polynomial, and the receiver checks whether the resulting polynomial remains divisible by $G(x)$.

Error Polynomial $E(x)$

Suppose the transmitted codeword is $T(x)$ and the received word is $R(x)$. Transmission errors can be modeled as adding a nonzero error polynomial $E(x)$:

$$R(x) = T(x) + E(x).$$

The polynomial $E(x)$ has coefficients in $\mathbb{F}_2$ and indicates which bit positions were flipped during transmission. For example:

$$E(x) = x^7 + x^2$$

means the 7th and 2nd bits (counting from zero) were inverted.

If $E(x) = 0$, no errors occurred.

Divisibility Argument

By construction of the CRC code,

$$T(x) = M(x)x^n + R(x)$$

is divisible by $G(x)$.

At the receiver, the remainder check reduces to examining whether the received polynomial is divisible by $G(x)$:

$$R(x) \equiv 0 \pmod{G(x)}.$$

Substituting $R(x) = T(x) + E(x)$ gives:

$$T(x) + E(x) \equiv E(x) \pmod{G(x)}.$$

Since $T(x)$ is divisible by $G(x)$, the receiver detects an error exactly when:

$$E(x) \not\equiv 0 \pmod{G(x)}.$$

Thus, a CRC fails to detect an error if and only if the error polynomial $E(x)$ is itself divisible by $G(x)$. This means CRC error detection performance depends entirely on the algebraic properties of $G(x)$.

Burst Error Detection Guarantees

A *burst error* of length L is an error that affects L or fewer consecutive bits; its polynomial representation has degree less than L.

If the generator polynomial has degree n, then CRCs guarantee detection of:

- All burst errors of length $\leq n$
- Most burst errors longer than n (with probability $1 - 2^{-n}$ for random errors)

Reason: a burst of length $\leq n$ yields an error polynomial $E(x)$ with degree $< n$, and $G(x)$ (degree n) cannot divide such a polynomial unless $E(x) = 0$.

More generally:

- If $G(x)$ contains $(x + 1)$ as a factor, it detects all errors that flip an odd number of bits (parity check property).
- A primitive or well-chosen $G(x)$ detects:
  - All single-bit and double-bit errors
  - All odd-weight errors (if $(x + 1)$ divides $G(x)$)
  - Any burst error of length $\leq n$
  - Almost all longer bursts

In practice, generator polynomials for CRC standards (CRC-16, CRC-32, etc.) are selected based on exhaustive search and formal analysis to maximize detection of real-world error patterns, such as short bursts and random multi-bit flips.

Summary

CRCs detect errors because:

- Transmitted codewords are divisible by $G(x)$.
- Errors add a polynomial $E(x)$.
- Detection reduces to checking whether $G(x)$ divides $E(x)$.

This algebraic structure gives CRCs strong, provable guarantees for detecting burst and random errors, far exceeding the reliability of simple checksums or parity checks.

4 Worked Example

4.1 Example Setup

To illustrate the CRC computation process concretely, we walk through a complete example using a small message and a simple generator polynomial. Although real systems often use longer polynomials (e.g., CRC-16 or CRC-32), a compact example makes the arithmetic easier to follow by hand while demonstrating the same underlying principles.

Message Bitstring

Let the message bitstring be:

$$M = 1101011011.$$

Interpreted as a polynomial over $\mathbb{F}_2$, this corresponds to:

$$M(x) = x^9 + x^8 + x^6 + x^4 + x^3 + x + 1.$$

This will be our input message to protect with a CRC.

Generator Polynomial

We select a generator polynomial of degree 4:

$$G(x) = x^4 + x + 1,$$

which in bitstring form is:

$$G = 10011.$$

This is a commonly used illustrative CRC generator, often referred to as the CRC-4 example polynomial. Its degree determines that the resulting CRC value will be 4 bits long.

Preparation for Division

Because $\deg(G) = 4$, we append 4 zeros to the original message (equivalently multiplying by x^4):

$$M' = 1101011011 \qquad \underbrace{0000}_{\text{4 zeros appended}} \quad .$$

Thus,

$$M'(x) = M(x) \cdot x^4.$$

This prepared bitstring M' will be divided by G using polynomial long division over $\mathbb{F}_2$ to obtain the CRC remainder.

4.2 Bitwise Polynomial Division Walkthrough

We now perform the CRC division of the prepared message by the generator, showing both the algebraic (polynomial) view and the concrete bitwise XOR process. Recall:

$$M = \texttt{1101011011}, \qquad G = \texttt{10011}\ (x^4 + x + 1), \qquad M' = \texttt{1101011011}\ \underbrace{\texttt{0000}}_{\deg G = 4}\ .$$

Step-by-Step Table (Polynomial + Bit Views)

At each step i, if the leading bit of the current dividend slice is 1, we XOR (subtract) G shifted by i bits. Over $\mathbb{F}_2$, subtraction = addition = XOR.

Step	Polynomial Action	Bitwise Action (Before → After)
0	$x^{13}+\cdots \ominus x^{13}+x^{10}+x^9$	`11010110110000` $\xrightarrow[\text{XOR at pos 0}]{\oplus\ 10011}$ `01001110110000`
1	$x^{12}+\cdots \ominus x^{12}+x^{9}+x^8$	`01001110110000` $\xrightarrow[\text{XOR at pos 1}]{\oplus\ 10011}$ `00000010110000`
6	$x^{7}+\cdots \ominus x^{7}+x^{4}+x^3$	`00000010110000` $\xrightarrow[\text{XOR at pos 6}]{\oplus\ 10011}$ `00000000101000`
8	$x^{5}+\cdots \ominus x^{5}+x^{2}+x^1$	`00000000101000` $\xrightarrow[\text{XOR at pos 8}]{\oplus\ 10011}$ `00000000001110`

At the end, fewer than $\deg G = 4$ actionable bits remain; the last 4 bits are the remainder.

XOR Windows and Alignment Diagrams

We now show the generator "sliding" under the dividend at each position. XOR is applied only when the leftmost bit of the current window is 1.

Position 0

```
Dividend (before):  11010110110000
Generator:          10011      (aligned under bits [0..4])
XOR result:         01001110110000
```

Position 1

```
Dividend (before):   01001110110000
Generator (shift 1):  10011     (aligned under bits [1..5])
XOR result:          00000010110000
```

Position 6

```
Dividend (before):   00000010110000
Generator (shift 6):       10011  (aligned under bits [6..10])
XOR result:          00000000101000
```

Position 8

```
Dividend (before):   00000000101000
Generator (shift 8):         10011 (aligned under bits [8..12])
XOR result:          00000000001110
```

Remainder Extraction and Codeword

The final working register is:

$$\texttt{0000000000}\underline{\texttt{1110}}$$

so the remainder (CRC) is

$$\boxed{\texttt{1110}}.$$

The transmitted codeword appends this remainder to the *original* message (not the zero-extended one):

$$\texttt{1101011011} \parallel \texttt{1110} = \texttt{11010110111110}.$$

Sanity Check (Receiver)

The receiver divides `11010110111110` by `10011`. Because we constructed the codeword as

$$T(x) = M(x)x^{\deg G} + R(x),$$

the remainder is identically zero if no errors occurred. Algebraically, $T(x) \equiv 0 \pmod{G(x)}$.

Key Rules in $\mathbb{F}_2$ (for reference)

- Addition = subtraction = XOR; no carries or borrows.
- Multiplication by x^k is a left shift by k bits.
- Division proceeds by canceling the highest-degree term via aligned XOR with the generator.

4.3 Receiver Verification

Once a CRC-protected frame is received, the receiver must verify integrity using the same generator polynomial $G(x)$ and the same conventions (bit order, initial register preset, final XOR, reflection) as the transmitter. There are two equivalent verification procedures.

Method A: Recompute-and-Compare

1. Parse the received frame into the payload (message) $M(x)$ and the appended n-bit CRC value $R(x)$, where $n = \deg G$.
2. Locally recompute the CRC over the received message: compute the remainder

$$R_{\text{local}}(x) \equiv M(x)\, x^n \bmod G(x).$$

3. **Accept** if and only if $R_{\text{local}}(x) = R(x)$ (after applying the same post-processing conventions, e.g., bit reflection or final XOR, if used).

Method B: Syndrome (Divisibility) Test

1. Treat the entire received bitstring (message plus CRC) as the polynomial

$$\widetilde{T}(x) \;=\; M(x)\, x^n + R(x).$$

2. Perform a single division by $G(x)$:

$$S(x) \;\equiv\; \widetilde{T}(x) \bmod G(x).$$

3. **Accept** if and only if the *syndrome* $S(x) = 0$.

These methods are equivalent because the transmitted codeword $T(x) = M(x)\, x^n + R(x)$ is constructed to be exactly divisible by $G(x)$. If an error polynomial $E(x)$ corrupts the transmission, the receiver sees $\widetilde{T}(x) = T(x)+E(x)$ and the remainder becomes $S(x) \equiv E(x) \bmod G(x)$, which is nonzero unless $G(x) \mid E(x)$.

Worked Check on the Running Example

With

$$M = \texttt{1101011011}, \quad G = \texttt{10011}\ (x^4 + x + 1), \quad \text{CRC} = \texttt{1110},$$

the transmitted codeword is

$$\texttt{1101011011} \,\|\, \texttt{1110} \;=\; \texttt{11010110111110}.$$

Dividing this 14-bit word by G (bitwise XOR long division in $\mathbb{F}_2$) yields remainder `0000`, so the syndrome $S(x) = 0$ and the frame is accepted.

Operational Notes

- **Conventions must match.** Many standardized CRCs use implementation conventions (initial register preset, input/output reflection, final XOR). In those cases, the receiver either
  (a) mirrors the transmitter's process and compares computed CRCs (Method A), or
  (b) checks against a known nonzero *residue* instead of literal zero in the syndrome test (Method B).
- **Pass/Fail policy.** If the remainder (or syndrome) is nonzero, the frame is declared corrupt and typically discarded or retried by higher layers (e.g., ARQ/TCP).
- **LFSR view.** In hardware, verification is naturally implemented by clocking the entire received codeword through the same LFSR. A zero (or known residue) state at the end indicates success.

5 Choosing CRC Polynomials

The effectiveness of a CRC depends critically on the choice of the generator polynomial $G(x)$. Not all polynomials offer equally strong error detection properties, and the polynomials used in practice are the result of theoretical analysis, exhaustive search, and decades of empirical testing in real communication systems.

Practical CRC Standards

Widely used CRCs employ standardized generator polynomials with proven detection guarantees. A few representative examples:

- **CRC-8** (ATM HEC, 8-bit CRC)

$$G(x) = x^8 + x^2 + x + 1 \quad \leftrightarrow \quad \texttt{100000111}$$

- **CRC-16-IBM** (CRC-16 used in USB, X.25)

$$G(x) = x^{16} + x^{15} + x^2 + 1 \quad \leftrightarrow \quad \texttt{11000000000000101}$$

- **CRC-32 (Ethernet, ZIP, PNG)**

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

$$\leftrightarrow \texttt{0x04C11DB7}$$

- **CRC-32C** (Castagnoli, used in iSCSI, SSE4.2 CRC instruction)

$$\leftrightarrow \texttt{0x1EDC6F41}$$

  This polynomial has been shown to outperform CRC-32 in detecting certain multi-bit patterns and is now widely used in modern protocols and CPUs.

These standardized choices strike a balance between implementation efficiency and strong algebraic error detection.

Criteria and Error Guarantees

A good CRC polynomial is chosen to maximize the minimum "distance" between valid codewords and to guarantee detection of common error types. Desired properties include:

- Detection of all single-bit errors
- Detection of all double-bit errors
- Detection of all odd numbers of bit errors (if $(x + 1) \mid G(x)$)
- Detection of all burst errors up to length n (where $n = \deg G$)
- Low probability of undetected random errors:

$$P_{\text{undetected}} \approx 2^{-n}$$

For example:

- CRC-16 detects all burst errors ≤ 16 bits
- CRC-32 detects all burst errors ≤ 32 bits
- CRC-32C increases detection strength for short patterns vs. CRC-32

In practice, real-world bit-error processes are *not random.* Patterns arise from EMI, clock recovery slips, crosstalk, and burst noise. Published CRC polynomials have been selected to excel against such structured corruption.

Why Not Use Random Polynomials?

Although any polynomial could be used as a CRC generator, random choices perform poorly in practice. Reasons include:

- Many polynomials fail to detect simple patterns (e.g., two-bit errors)
- Some lack the $(x + 1)$ factor needed to detect all odd-weight errors
- Real communication channels produce burst errors, not uniform random flips
- Cyclic structure matters: non-primitive or reducible polynomials may produce poor cycle lengths and weak burst detection
- Exhaustive search and algebraic construction outperform trial-and-error

A random polynomial of degree n is overwhelmingly unlikely to match the carefully vetted performance of standardized CRC polynomials. Modern CRCs (e.g. Castagnoli) were selected via computational analysis testing billions of patterns.

Summary

- CRC performance depends critically on the generator polynomial.
- Industry-standard CRC polynomials are used because they have strong, proven detection guarantees.
- Random polynomials are almost always inferior and should not be used.
- Modern CRC development combines algebraic theory and exhaustive testing.

6 Reference Implementations

6.1 Pseudocode

This section presents clear pseudocode for CRC generation and verification. It covers both the *bitwise long-division* method (portable, simple) and the *table-driven* method (fast, byte-at-a-time). Conventions (init, reflection, final XOR) are parameterized.

Parameters and Conventions

- W: CRC width in bits (e.g., $8, 16, 32$), $n \leftarrow W$.
- `POLY`: generator polynomial without the top bit (for non-reflected algorithms the implicit x^W term is omitted from `POLY` but assumed).
- `INIT`: initial register preset.
- `XOROUT`: value XORed with the register at the end ("final XOR").
- `REFIN` / `REFOUT`: whether to reflect input bytes / final register.
- `MSB_MASK` $\leftarrow 1 \ll (W-1)$,   `REG_MASK` $\leftarrow (1 \ll W) - 1$.

Helper: Bit reflection

```
function REFLECT(v, bits):
    r := 0
    for i in 0 .. bits-1:
        if (v >> i) & 1 == 1: r := r | (1 << (bits-1-i))
    return r
```

A. Bitwise (Long Division) – Non-Reflected Form

Matches the polynomial model used in the math sections: left-shift register, test MSB, XOR with `POLY` (without implicit top bit), process from most-significant bit to least in each byte.

```
function CRC_BITWISE_NONREF(data[0..N-1], W, POLY, INIT, XOROUT, REFIN=false, REFOUT=false):
    reg := INIT & REG_MASK
    for each byte b in data:
        if REFIN: b := REFLECT(b, 8)
        reg := reg ^ (b << (W-8))          # inject next 8 bits at top
        for i in 0 .. 7:                   # process 8 bits, MSB-first
            if (reg & MSB_MASK) != 0:
                reg := ((reg << 1) ^ POLY) & REG_MASK
            else:
                reg := (reg << 1) & REG_MASK
    if REFOUT: reg := REFLECT(reg, W)
    return (reg ^ XOROUT) & REG_MASK
```

Notes:

- `POLY` is the W-bit polynomial with the top bit dropped (implicit x^W).
- Many standards set `REFIN=false`, `REFOUT=false` for this style; check the spec.

B. Bitwise – Reflected Form

Common in software (e.g., CRC-32/IEEE with `REFIN=true`, `REFOUT=true`). Here the register shifts right, the LSB is tested, and `POLY` is the *reflected* polynomial.

```
function CRC_BITWISE_REFLECTED(data[0..N-1], W, POLY_REFLECTED, INIT, XOROUT):
reg := INIT & REG_MASK
for each byte b in data:
# When reflected, bytes are fed LSB-first (no per-byte reflect needed)
reg := reg ^ b
for i in 0 .. 7:                   # process 8 bits, LSB-first
if (reg & 1) != 0:
reg := (reg >> 1) ^ POLY_REFLECTED
else:
reg := reg >> 1
return (reg ^ XOROUT) & REG_MASK
```

Notes:

- For CRC-32 (IEEE), `W=32`, `POLY_REFLECTED=0xEDB88320`, `INIT=0xFFFFFFFF`, `XOROUT=0xFFFFFFFF`.
- Equivalently, `POLY_REFLECTED = REFLECT(0x04C11DB7, 32)` >> 1 (aligned for LSB tests).

C. Verification (Syndrome) Check

Either recompute and compare, or clock the entire (message ‖ CRC) through the same core and check for the known residue (often 0, but may be nonzero for some conventions).

```
function CRC_VERIFY(data, expected_crc, params):
# Option 1: recompute-and-compare
computed := CRC_xxx(data, params)
return (computed == expected_crc)

function CRC_SYNDROME_ZERO(codeword, params):
# Feed message bytes then CRC bytes through the same routine but without final XOR/reflect
reg := CRC_CORE_NO_FINALS(codeword, params)
return (reg == KNOWN_RESIDUE)         # often 0 for pure polynomial model
```

D. Table-Driven (Byte-at-a-Time) – Reflected Form

Precompute a 256-entry table to accelerate the inner loop from 8 bit-steps to 1 table lookup.

```
function MAKE_TABLE_REFLECTED(W, POLY_REFLECTED):
table[256]
for i in 0 .. 255:
r := i
for k in 0 .. 7:
if (r & 1) != 0:
r := (r >> 1) ^ POLY_REFLECTED
else:
r := r >> 1
table[i] := r & REG_MASK
return table

function CRC_TABLE_REFLECTED(data, W, POLY_REFLECTED, INIT, XOROUT):
```

```
T := MAKE_TABLE_REFLECTED(W, POLY_REFLECTED)   # usually cached globally
reg := INIT & REG_MASK
for each byte b in data:
idx := (reg ^ b) & 0xFF
reg := (reg >> 8) ^ T[idx]
return (reg ^ XOROUT) & REG_MASK
```

E. Table-Driven – Non-Reflected Form

Equivalent speedup for MSB-first conventions.

```
function MAKE_TABLE_NONREF(W, POLY):
table[256]
top := 1 << (W-1)
for i in 0 .. 255:
r := i << (W-8)
for k in 0 .. 7:
if (r & top) != 0:
r := ((r << 1) ^ POLY) & REG_MASK
else:
r := (r << 1) & REG_MASK
table[i] := r
return table

function CRC_TABLE_NONREF(data, W, POLY, INIT, XOROUT, REFIN=false, REFOUT=false):
T := MAKE_TABLE_NONREF(W, POLY)
reg := INIT & REG_MASK
for each byte b in data:
if REFIN: b := REFLECT(b, 8)
idx := ((reg >> (W-8)) ^ b) & 0xFF
reg := ((reg << 8) & REG_MASK) ^ T[idx]
if REFOUT: reg := REFLECT(reg, W)
return (reg ^ XOROUT) & REG_MASK
```

F. Practical Notes

- Always match the standard's tuple (`POLY`, `INIT`, `REFIN`, `REFOUT`, `XOROUT`).
- For verification via syndrome, some standards require checking against a fixed nonzero residue.
- For performance, prefer reflected table-driven (many CPUs provide CRC intrinsics, e.g., CRC32C).
- For pedagogical clarity or portability (no tables), use the bitwise version.

6.2 Hardware LFSR Interpretation

Although CRCs are often taught through polynomial long division, real high-speed systems (Ethernet MACs, storage controllers, buses such as CAN/FlexRay, FPGA datapaths) implement CRC generation using a *linear feedback shift register* (LFSR). The LFSR is a direct hardware realization of polynomial arithmetic over $\mathbb{F}_2$.

Conceptual View

An LFSR maintains a W-bit register (where $W = \deg G$) and shifts one bit of input at a time. Certain bit positions (taps) XOR into the least significant bit (or the most significant bit, depending on reflected vs.

non-reflected implementation). These taps correspond exactly to the nonzero coefficients of the generator polynomial $G(x)$.

$$G(x) = x^W + g_{W-1}x^{W-1} + \cdots + g_1 x + g_0, \qquad g_i \in \{0, 1\}$$

If $g_i = 1$, then bit position i in the register participates in the feedback XOR path.

LFSR Structure (Non-Reflected / MSB-First)

For a generator polynomial of degree W, the canonical LFSR for MSB-first operation:

$$\text{input bit} \to \boxed{\text{XOR}} \to \boxed{b_{W-1}} \to \boxed{b_{W-2}} \to \cdots \to \boxed{b_0}$$

Feedback taps return from selected register stages to the XOR at the input according to $G(x)$.

- The leftmost flip-flop corresponds to x^W and always feeds back.
- Each coefficient $g_i = 1$ creates a tap from stage i into the XOR.
- On each clock: shift left, XOR feedback into LSB.

LFSR Structure (Reflected / LSB-First)

Reflected CRCs (like CRC-32 IEEE) shift right instead. The LSB is tested and taps XOR into the MSB:

$$\boxed{b_{W-1}} \leftarrow \cdots \leftarrow \boxed{b_1} \leftarrow \boxed{b_0} \leftarrow \boxed{\text{XOR}} \leftarrow \text{input bit}$$

Where b_0 is the least significant bit and is AND-tested to decide feedback.

Example: CRC-4 Generator $G(x) = x^4 + x + 1$

Polynomial: `10011`. Degree 4, taps at positions 4 and 1 and 0.

MSB-first LFSR logic:

$$b_4 \oplus b_1 \oplus b_0 \longrightarrow \text{feedback into } b_0$$

Diagram (conceptual):

$$\text{in} \to \oplus \to b_3 \to b_2 \to b_1 \to b_0$$

with taps from b_3 and b_0 feeding the XOR (implicit x^4 term, x^1, and x^0).

Equivalence to Polynomial Division

LFSR hardware and polynomial long division are equivalent:

- Shifting $\leftrightarrow$ multiplying by x
- XOR taps $\leftrightarrow$ subtracting (=) adding $G(x)$ when the leading bit is 1
- Final register contents $\leftrightarrow$ the remainder $R(x)$

Thus, the LFSR is a compact, pipelined hardware engine performing $\mathbb{F}_2$ division in real time.

Why LFSRs Are Ideal for CRCs

- Minimal hardware: only flip-flops and XOR gates
- One bit processed per clock (or multiple in parallel with unrolled logic)
- Very high throughput—suitable for gigabit-speed links
- Directly reflects polynomial structure and error-detection guarantees

Design Notes

- Bit order (MSB-first vs. LSB-first) must match CRC convention.
- Taps come from the nonzero coefficients of $G(x)$ excluding the highest bit.
- Some CRCs incorporate initial presets, bit reflection, or final XOR stages; these can also be implemented as part of the LFSR pipeline.
- Parallel LFSR designs exist for byte- or word-at-a-time computation inside FPGAs/ASICs.