A Very Fast Linear CRC Routine: Loop-Free, Table-Free Error Detection in 8086 Assembly

Ivars Vilums and Aubrey McIntosh

Original implementation: 1984

Abstract

In 1984, we developed a cyclic redundancy check (CRC) computation routine that achieved real-time performance for synchronous communications on 4 MHz Intel 8086 processors. The implementation—written entirely in assembly language—required neither lookup tables nor iterative loops. Instead, it exploited the processor's hardware parity flag and conditional branch instructions to achieve linear, constant-time performance per data byte. Processing an 8-bit byte required approximately 10 machine instructions, enabling 34 KB/s data rates—sufficient for BiSync protocol communications on early IBM PC hardware. The method represents an early example of architecture-aware algorithmic optimization, exploiting specific processor features (parity flag, conditional branching) to achieve performance previously thought to require table lookups or bit-serial processing.

1. Introduction

Cyclic redundancy checks (CRCs) provide efficient error detection for serial communications. By the early 1980s, CRC computation was well-established in protocols such as IBM's Binary Synchronous (BiSync) communications. However, implementations on early personal computers faced severe performance constraints. A 4 MHz Intel 8086 processor provided approximately 0.25 million instructions per second (MIPS), and real-time communications protocols demanded that all processing—including character conversions, protocol handling, and error checking—execute within tight timing windows.

Standard CRC implementations of the era employed either bit-serial algorithms (processing one bit at a time through repeated shifts and conditional XORs) or table-lookup methods (using 256-entry tables to process bytes). Bit-serial approaches required 8 iterations per byte. Table lookups, while faster, consumed precious memory and cache resources on machines with limited RAM.

This paper describes a novel approach developed by the authors in 1984 that eliminated both loops and tables, achieving constant-time performance per byte through direct exploitation of processor architecture features.

2. Technical Background

2.1 CRC Fundamentals

A CRC is computed by treating a bit stream as the coefficients of a polynomial and dividing by a generator polynomial, retaining the remainder. For the BiSync protocol, the generator polynomial is $x^{16} + x^{15} + x^2 + 1$, represented by the 16-bit constant 0x8003.

Mathematically, CRC computation is linear: $CRC(A \oplus B) = CRC(A) \oplus CRC(B)$, where $\oplus$ denotes XOR. This linearity property is fundamental to our approach.

2.2 The 8086 Parity Flag

The Intel 8086 processor maintains a parity flag (PF) indicating whether the low byte of the most recent result contains an even (PF=1) or odd (PF=0) number of set bits. The JP (Jump on

Parity Even) instruction branches if PF=1. This hardware-computed parity is equivalent to XORing all bits of a byte—precisely the operation needed for certain CRC computations.

3. Algorithm Description

3.1 Core Insight

The key innovation exploits three observations: (1) CRC computation is linear, allowing decomposition into independent operations. (2) The processor's parity flag computes the XOR of all bits in a byte at no additional cost. (3) Bit reversal—required by BiSync's LSB-first transmission order—can be precomputed for nibbles using a 16-byte table, much smaller than a full 256-byte CRC table.

3.2 Structural Decomposition

The algorithm decomposes CRC update into three structural transformations:

- **Structure 1:** Conditional XOR with generator polynomial based on parity
- **Structure 2:** Byte swap of current CRC value
- **Structure 3:** Positioning and XOR of input byte

These operations, executed in fixed sequence without branching (except the single parity-based conditional), produce the updated CRC in constant time.

4. Implementation Details

4.1 Bit Reversal

BiSync protocol transmits bits LSB-first, while computation proceeds MSB-first. The REVBYTE procedure reverses bit order using a 16-entry table for nibbles:

```
;Reverse bit order of byte in AL
REVBYTE PROC NEAR
    MOV  AH,AL          ;save original
    AND  AL,0FH         ;mask low nibble
    XLAT                ;table lookup reverses nibble
    XCHG AL,AH          ;get high nibble
    SHR  AL,1           ;shift to low position
    SHR  AL,1
    SHR  AL,1
    SHR  AL,1
    XLAT                ;reverse second nibble
    SHL  AH,1           ;recombine
    SHL  AH,1
    SHL  AH,1
    SHL  AH,1
    OR   AL,AH
    RET
REVBYTE ENDP
```

4.2 CRC Update Core

The ADDTOCRC procedure updates the 16-bit CRC with a new byte:

```
ADDTOCRC PROC NEAR
    CALL REVBYTE        ;reverse input byte
    XOR  AH,AH          ;clear high byte
    MOV  DH,CRCHI       ;load current CRC
    MOV  DL,CRCLO
    ;Core algorithm begins
    XCHG DH,DL          ;swap bytes (structure 2)
    XOR  AL,DL          ;XOR with input
    XCHG DL,AH          ;clear DL, save AL in AH
    JP   PDONE          ;parity test
    XOR  DX,PCONSTANT   ;conditional XOR (structure 1)
PDONE:
    XOR  AH,AH          ;clear again
    SHL  AX,1           ;position input (structure 3)
    XOR  DX,AX
    SHL  AX,1
    XOR  DX,AX
    MOV  CRCLO,DL       ;store result
    MOV  CRCHI,DH
    RET
ADDTOCRC ENDP
```

5. Performance Analysis

On a 4 MHz 8086 processor, the core ADDTOCRC routine executes in approximately 10-12 machine instructions per byte, totaling roughly 30-40 clock cycles including the bit reversal. This enabled sustained data rates of 34 KB/s, meeting real-time requirements for BiSync communications at 9600 baud (approximately 1.2 KB/s) with substantial margin for protocol overhead, character conversion, and other processing.

Compared to contemporary approaches:

- **Bit-serial CRC:** 8-16 instructions per bit × 8 bits = 64-128 instructions per byte
- **Table lookup:** ~15-20 instructions per byte but required 256-byte table
- **This algorithm:** ~10-12 instructions per byte with only 16-byte table

6. Historical Context and Application

This algorithm was developed for a BiSync communications module in a 3741 Data Entry Station emulator for the IBM PC. The emulator had to handle full BiSync protocol processing, including on-the-fly ASCII/EBCDIC conversion, all in real time on 4 MHz hardware.

The algorithm was first implemented on the Z80 processor (where it was actually shorter due to richer 8-bit operations) and subsequently translated to 8086 assembly. While the technique was proprietary at the time, similar approaches have since been independently rediscovered in

various optimized CRC implementations, though the specific exploitation of the parity flag appears to have remained unique.

6.1 Development Narrative

The collaboration began on a Friday morning in 1984 when Ivars Vilums telephoned Aubrey McIntosh and asked whether he knew what a CRC-16 was. McIntosh, then holding a B.A. in mathematics and chemistry, did not but agreed to research the topic and visited the university library that day. The following Saturday, he went to Vilums's home, bringing with him a small program written in IBM Pascal 1.0 that traced the CRC output bit patterns for each input bit. This exploratory code revealed a recurring linear structure suggesting a simpler computational path.

During the same session, McIntosh—new to the Intel 8086 instruction set—examined the processor documentation and asked whether the architecture provided a parity bit and a conditional branch based on parity, terminology both participants knew from Motorola 6809 assembler conventions. Vilums, whose background combined music and computer science and who was already experienced with 8086 assembly, confirmed that such instructions existed. Together, they recognized that the parity flag could replace explicit bit-counting logic.

By Sunday evening, the CRC routine was implemented, verified against reference data, and integrated into a BiSync communications package. The customer acceptance test on Monday morning succeeded without modification.

Both authors have since pursued distinct but complementary paths—McIntosh later earning a Ph.D. in chemistry and Vilums continuing in software, systems development, and other persuits. They remain active collaborators and, through The Eastjesus Company, occasionally accept special-interest projects under a best-effort retainer model, favoring technically challenging work that has been abandoned by others.

7. Generalization and Adaptation

The algorithm is readily adaptable to other CRC polynomials by changing the PCONSTANT value (0x8003 for BiSync). The approach works for any processor with a parity flag and conditional branch instruction. For polynomials with different bit orders or applications not requiring bit reversal, the REVBYTE step can be omitted, further improving performance.

The term "Magic Constant" in the context of the FastCRC.html snippet refers to the pre-calculated, hardcoded value that represents the CRC generator polynomial. In the code, this constant is PCONSTANT=08003H for the x16+x15+x2+1 (Bisync) polynomial.

The key to the very fast, loop-less implementation is that this constant is used in a conditional XOR operation (JP PDONE / XOR DX, PCONSTANT) which simulates the effect of the CRC feedback over multiple bit shifts at once (an entire byte).

To implement this style of loop-less CRC for other polynomials, you would need to calculate a corresponding "magic constant." This constant is simply the hexadecimal representation of the generator polynomial itself, sometimes bit-reversed, depending on the implementation's endianness and bit order convention.

Below are the Magic Constants (generator polynomials) for several commonly used CRC standards. The values are typically presented in their Normal (most-significant-bit first) or

Reversed (least-significant-bit first) form, as the assembly code in your example uses a reversed-bit approach.

Common CRC Polynomials and their Constants

CRC Name	Polynomial (g(x))	Normal Hex Value (MSB-first)	Reversed Value (LSB-first)	Typical Use
CRC-16-CCITT	x16+x12+x5+1	0x1021	0x8408	XMODEM, V.41, Bluetooth, PPP
CRC-16-IBM (or ANSI)	x16+x15+x2+1	0x8005	0xA001	USB, Modbus
CRC-16-Bisync (Your example)	x16+x15+x2+1	0x8003	0xC002	Bisync Protocol
CRC-32-IEEE 802.3	x32+x26+x23+x22+x16+x12+x11+x10+x8+x7+x5+x4+x2+x+1	0x04C11DB7	0xEDB88320	Ethernet, ZIP, PNG, zlib
CRC-32C (Castagnoli)	x32+x28+x27+x26+x25+x23+x22+x20+x19+x18+x14+x13+x11+x10+x9+x8+x6+1	0x1EDC6F41	0x82F63B78	iSCSI, Btrfs, ext4

Notes on the "Magic Constant"

Bit Ordering is Crucial: The value 08003H in the original assembly code is slightly unusual because it is the Normal representation (0x8003) that has been truncated by dropping the highest x16 term, but then bit-reversed differently than the standard convention. For a general, loop-less implementation like the one in FastCRC.html, the PCONSTANT must be a carefully pre-calculated value that models the full 8-bit block transformation.

For the standard x16+x15+x2+1 polynomial, the standard Reversed constant (used in many modern implementations) is 0xC002, which is the bit-reversed 0x8003.

The value 08003H used in the original code is the Normal representation of the polynomial x15+x2+1, which is effectively x16+x15+x2+1 with the highest x16 term dropped (as is common). The fact that the code uses 08003H for the Bisync polynomial indicates the implementation is tailored specifically to its processor and bit-reversal/shift logic.

Implementation Dependence: The specific value of the constant depends entirely on the implementation details:

Bit-Order: Is the data shifted Most-Significant-Bit (MSB) first (Normal convention) or Least-Significant-Bit (LSB) first (Reversed convention)?

Final XOR: Is the result XORed with a final value (e.g., 0xFFFF or 0xFFFFFFFF)?

Initial Value: Does the CRC register start at zero or an initial non-zero value (e.g., 0xFFFF)?

Modern Fast Implementations: Modern fast CRC calculations typically use a table-driven approach (Sarwate or slice-by-8 algorithms) rather than the pure loop-less, table-less bit-twiddling of the 8086 code. These table-driven methods are often much faster on modern CPUs and, while they still use the generator polynomial (e.g., 0xEDB88320 for CRC-32), they use it to generate a 256-entry lookup table rather than as a single PCONSTANT in a conditional jump.

Extension to 8-bit CRCs is straightforward and would be even faster. The fundamental principle—exploiting hardware-computed parity to eliminate conditional branches in the core loop—remains applicable to modern processors, though the relative performance advantage has diminished as processor speeds have far outpaced I/O rates.

8. Conclusion

The 1984 Fast CRC routine demonstrates how deep understanding of processor architecture can yield substantial performance improvements. By recognizing that the parity flag computes the XOR of all bits, and that CRC operations decompose into simple structural transformations, we achieved constant-time performance without the memory overhead of table lookups.

This work exemplifies architecture-aware algorithm design—a principle that remains relevant today as developers seek to exploit SIMD instructions, GPU parallelism, and other specialized hardware features. The specific technique described here, while perhaps of primarily historical interest given modern processor speeds, illustrates the creative possibilities when algorithm design and hardware architecture are considered together.

Acknowledgments

This work was developed during the authors' collaboration in the early 1980s. The algorithm was originally implemented for the BiSync module for the emulation of the IBM 3741 Data Entry Station on the original IBM PC. We thank the early IBM PC developer community for creating an environment where such optimization was both necessary and rewarding.

References

IBM Corporation. Binary Synchronous Communications Protocol Reference Manual. IBM Document GA27-3004.

Intel Corporation (1981). The 8086 Family User's Manual. Intel Corporation, Santa Clara, CA.

Peterson, W. W., & Brown, D. T. (1961). Cyclic codes for error detection. Proceedings of the IRE, 49(1), 228-235.

McIntosh, A. (1998). Fast CRC without loops or tables. Usenet posting to comp.arch.embedded, November 7, 1998.

Vilums, I. (1997). Fast CRC Routine. http://www.underware.com/ijv/crc.htm (archived at Internet Archive).

Historical Note: This paper documents work originally performed in 1984. The algorithm was posted to the web in 1997 and discussed in Usenet groups in 1998. This formal writeup was prepared in 2025 to preserve the technical details and historical context of this early optimization work.