# Raport: Implementacja kodu wielomianowego CRC-32 – IEEE 802.3

**Autorzy:** Antoni Lenart, Bruno Banaszczyk, Jakub Kowalski, Filip Kaczor, Jakub Kogut
**Data:** 15 czerwca 2025

## 1. Wprowadzenie

Kody CRC (Cyclic Redundancy Check) są powszechnie stosowaną metodą wykrywania błędów w transmisji danych oraz przechowywaniu informacji w systemach cyfrowych. Projekt koncentruje się na implementacji kodu CRC-32 zgodnego ze standardem IEEE 802.3, wykorzystywanym m.in. w sieciach Ethernet. Niniejszy raport przedstawia szczegółowy opis podstaw matematycznych, algorytmu, implementacji w języku Python, dokumentację kodu, wyniki jego działania oraz miejsce na przedstawienie sprzętowej implementacji w programie MultiSim.

Celem projektu jest analiza i implementacja CRC-32, omówienie jego zastosowań, możliwości detekcyjnych oraz stworzenie wizualizacji procesu obliczania sumy kontrolnej. Projekt obejmuje zarówno aspekty teoretyczne, jak i praktyczne, w tym symulację działania algorytmu oraz potencjalną implementację sprzętową.

---

## 2. Teoretyczne podstawy CRC

### 2.1. Czym jest CRC?

CRC to technika wykrywania błędów polegająca na dołączaniu do bloku danych krótkiej wartości kontrolnej, będącej resztą z dzielenia wielomianowego zawartości danych. Po stronie odbiorczej obliczenia są powtarzane, a niezgodność wartości kontrolnych wskazuje na uszkodzenie danych. Nazwa CRC pochodzi od:

- **Redundancji**: wartość kontrolna zwiększa rozmiar wiadomości bez dodawania nowych informacji.
- **Cykliczności**: operacje są oparte na kodach wielomianowych.

CRC jest popularne ze względu na:

- Prostą implementację sprzętową.
- Łatwość analizy matematycznej.
- Wysoką skuteczność w wykrywaniu błędów spowodowanych szumami w kanałach transmisyjnych.

### 2.2. Zastosowania CRC

Kody CRC znajdują zastosowanie w wielu dziedzinach, m.in.:

- **Sieci komputerowe**: Ethernet (IEEE 802.3), WiFi (IEEE 802.11).
- **Systemy komunikacji**: CAN, USB, Bluetooth.
- **Przechowywanie danych**: dyski twarde, SSD.
- **Formaty plików i kompresja**: ZIP, RAR, Gzip, PNG.
- **Funkcje hashujące**: rzadziej, w specyficznych zastosowaniach.

## 2.3. Podstawy matematyczne CRC-32

CRC-32 opiera się na arytmetyce wielomianów w ciele skończonym GF(2), gdzie:

- Dodawanie i odejmowanie to operacja XOR.
- Mnożenie i dzielenie są specyficzne dla arytmetyki binarnej.

Wiadomości są reprezentowane jako wielomiany $M(x)$, gdzie każdy bit odpowiada współczynnikowi. Na przykład ciąg bitów `1011` odpowiada wielomianowi $x^3 + x + 1$.

#### Wielomian generujący

Standardowy wielomian generujący dla CRC-32 (IEEE 802.3) to:
$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
W postaci binarnej: `100000100110000010001110110110111` (33 bity, w tym bit najwyższego stopnia).

#### Proces obliczania CRC

1. **Dopełnienie danych**: Dane wejściowe są mnożone przez $x^{32}$, co odpowiada dopełnieniu 32 zerami na końcu.
   $M'(x) = M(x) \cdot x^{32}$
2. **Dzielenie wielomianowe**: Dopełnione dane są dzielone przez $G(x)$ w ciele GF(2). Reszta z dzielenia (32 bity) stanowi sumę kontrolną CRC. $M'(x) = Q(x) \cdot G(x) + R(x),\ deg(R(x)) < 32$
3. **Dołączanie sumy kontrolnej**: Reszta $R(x)$ jest dołączana do oryginalnych danych, tworząc słowo kodowe.
4. **Weryfikacja**: Odbiorca dzieli otrzymane słowo $M'(x)$ kodowe przez $G(x)$. Jeśli reszta wynosi 0, dane są poprawne.
   $M'(x)\ mod\ G(x) = 0$

#### Projektowanie wielomianu generującego

Wielomian $G(x)$ musi być starannie wybrany, aby zapewnić wysoką skuteczność wykrywania błędów. Kluczowe cechy:

- **Wielomian pierwotny**: ma on największy "cykl" wykrywalności. Wykrywa: wszystkie 1-bitowe błędy, wszystkie 2-bitowe błędy jeśli blok_danych $\leq 2^r - 1$, gdzie `r` to stopień wielomianu. Nie mamy natomiast gwarancji wykrycia wszystkich błędów 3-bitowych, 4-bitowych itd.
- **Wielomian typu** $g(x) = p(x) \cdot (x + 1)$ , $p(x) \rightarrow$ pierwiastek pierwotny stopnia `(r-1)`: Wykrywa wszystki błędy 1-bitowe, 2-bitowe oraz wszystkie błędy o nieparzystej liczbie błędów. Ma on natomiast krótszy maksymalny blok danych, w którym można wykryć

błąd: $2^{r-1} - 1$, czyli o połowe mniej niż w przypadku pierwotnego wielomianu generującego.

- **Wielomiany rozkładalne**: Wielomian jest rozkładalny, jeśli da się go zapisać w następujący sposób:
  $g(x) = f(x) \cdot h(x)$
  W takiej sytuacji pierścień resztkowy nie jest ciałem, tylko ma w sobie tzw. zero-dzielniki. Są to takie elementy $a(x)$, dla których istnieje element niezerowy $b(x)$, taki że:
  $a(x) \cdot b(x) = 0 \ mod \ g(x)$
  Jeżeli jakikolwiek błąd przyjmie taką właśnie postać, to nie będziemy w stanie go wykryć za pomocą CRC.

**Możliwości detekcyjne CRC**

Jeśli dane zostaną zmienione ( np. przez zakłócenia transmisji ), to odebrany ciąg będzie różnił się od oryginału. Tę różnicę zapisuje się jako wielomian błędu $E(x)$. CRC będzie w stanie wykryć błąd tylko wtedy, gdy $E(x)$ nie dzieli się przez $G(x)$.

Jakie jesteśmy w stanie wykryć z pomocą kodów cyklicznych CRC-x ?

- **Błędy pojedynczego bitu**: Zawsze, jeśli $G(x)$ ma co najmniej dwa niezerowe wyrazy.
  Wyjaśnienie:
  $E(x) = x^5 \ \rightarrow$ wielomian $x^k$ dzieli się tylko przez $x, x^2, x^3, \ldots$ (wielomiany tylko z jednym współczynnikiem)
- **Błędy dwóch bitów**: Jeśli odległość między błędnymi bitami jest mniejsza niż rząd wielomianu pierwotnego. Czym jest rząd wielomianu ?
  Jest to najmniejsza liczba `m`, dla której:
  $G(x) | (x^m + 1)$
  Wyjaśnienie:
  $E(x) = x^k \cdot (x^{i-k} + 1) \ \rightarrow$ widzimy, że $G(x)$ musi dzielić $x^{i-k} + 1$ żeby nie było możliwe wykrycie tego błędu.
- **Błędy o nieparzystej liczbie bitów**: Jeśli $G(x)$ jest podzielny przez $(x + 1)$. Wyjaśnienie:
  Dany wielomian $f(x)$ jest podzielny przez $(x + 1)$ jeśli jego wartość w punkcie 1 wynosi 0, czyli:
  $f(1) = 0 \ \leftrightarrow \ x + 1 \mid f(x)$
  Zgodnie z zasadami arytmetyki w ciele $GF(2)$ sumowanie nieparzystej liczby 1 daje nam wynik $\rightarrow$ 1 ( wystąpienie błędu ), natomiast parzysta liczba 1 w $E(x)$ daje pozorny brak błędu ( oczywiście jest to błędne wskazanie ).
- **Błędy burst**: Ciągłe sekwencje błędnych bitów o długości mniejszej niż stopień $G(x)$, jeśli najwyższy współczynnik i wyraz wolny są niezerowe. Wyjaśnienie:
  Błąd burst jest to taki błąd, który występuje w postaci ciągłej sekwencji błędnych bitów, w której:
  - pierwszy i ostatni bit są błędne
  - pomiędzy nimi może znajdować się dowolna liczba ( również 0 ) błędnych lub poprawnych błędów.
- **Wszystkie kombinacje błędów mniejszych niż minimalna odległość Hamminga**:
  Wyjaśnienie:
  Minimalna odległość Hamminga kodu $\rightarrow$ czyli najmniejsza liczba bitów, które trzeba

zmienić, aby otrzymać inne słowo kodowe.
Błędy o liczności < $d_{min}$ nie są w stanie przekształcić poprawnego słowa kodowego w inne poprawne słowo kodowe. Czyli takie słowo zostanie wykryte jako nieprawidłowe.

Problemy z wykrywaniem:

- **Błędy na początku sekwencji (zera na początku).**
- **Błędy o parzystej liczbie bitów, jeśli $G(x)$ nie zawiera $(x + 1)$.**

---

# 3. Algorytm CRC-32

Algorytm CRC-32 według standardu IEEE 802.3 (zdefiniowany w IEEE 802.3 - 2022 ) polega na:

1. Przyjęciu danych wejściowych jako ciągu bitów.
2. Dopełnieniu danych 32 zerami.
3. Wykonaniu dzielenia wielomianowego przez wielomian generujący.
4. Dołączeniu 32-bitowej reszty jako sumy kontrolnej.
5. Weryfikacji po stronie odbiorcy przez powtórzenie dzielenia.

802-3-CRC32

---

# 4. Implementacja w Pythonie

## 4.1. Dokumentacja kodu

### Spis treści



### Wprowadzenie

Dokumentacja opisuje implementację algorytmu CRC-32 w Pythonie, zgodnego ze standardem IEEE 802.3. Kod zawiera funkcje do obliczania sumy kontrolnej CRC, wizualizacji procesu dzielenia binarnego oraz weryfikacji poprawności danych. Każda funkcja jest opisana pod kątem jej przeznaczenia, parametrów, zwracanych wartości oraz kluczowych fragmentów kodu.

### Funkcje

#### toBin(num)

Konwertuje liczbę całkowitą na ciąg binarny, usuwając prefiks `0b`.

#### Parametry

- `num` (int): Liczba całkowita do konwersji.

#### Zwraca

- `str`: Ciąg binarny reprezentujący liczbę. Dla zera zwraca `"0"`.

#### Przykład

```
>>> toBin(10)
'1010'
>>> toBin(0)
'0'
```

#### Kluczowe linie kodu

```
return bin(num)[2:] if num != 0 else "0"
```

- **`bin(num)[2:]`**: Konwertuje liczbę na ciąg binarny i usuwa prefiks `0b`.
- **Warunek `if num != 0 else "0"`**: Zapewnia, że dla zera zwracany jest ciąg `"0"`.

---

#### toDec(bin_str)

Konwertuje ciąg binarny na liczbę całkowitą (dziesiętną).

#### Parametry

- `bin_str` (str): Ciąg binarny (np. `"1011"`).

#### Zwraca

- `int`: Wartość dziesiętna odpowiadająca ciągowi binarnemu. Dla pustego ciągu zwraca `0`.

#### Przykład

```
>>> toDec("1011")
11
>>> toDec("")
0
```

#### Kluczowe linie kodu

```
return int(bin_str, 2) if bin_str else 0
```

- **`int(bin_str, 2)`**: Konwertuje ciąg binarny na liczbę dziesiętną.

- **Warunek `if bin_str else 0`**: Obsługuje pusty ciąg, zwracając `0`.

---

#### CRC_visual(data, key)

Wizualizuje proces obliczania sumy kontrolnej CRC-32 poprzez dzielenie binarne w ciele GF(2). Dane są dzielone przez wielomian generujący, a wynik (słowo kodowe) zawiera dane wejściowe i dołączoną sumę kontrolną.

##### Parametry

- `data` (str): Ciąg binarny reprezentujący dane wejściowe (np. `"1011001"`).
- `key` (str): Ciąg binarny reprezentujący wielomian generujący (np. `"1001"` dla prostego przypadku lub 33-bitowy dla CRC-32).

##### Zwraca

- `str`: Słowo kodowe (dane + CRC) jako ciąg binarny.

##### Wyjątki

- `ValueError`: Jeśli klucz jest pusty.

##### Przykład

```
>>> CRC_visual("1000101", "101")
# Wyświetla kroki dzielenia binarnego i zwraca np.:
'100010101'
```

##### Kluczowe linie kodu

```
dividend = code << (n - 1)
```

- Przesuwa dane w lewo o $n - 1$ bitów, dopełniając je zerami, aby zarezerwować miejsce na sumę kontrolną.

```
portion = dividend >> current_shft
```

- Pobiera $n$ najbardziej znaczących bitów z dywidendy do operacji XOR.

```
rem = portion ^ gen
```

- Wykonuje operację XOR między fragmentem danych a wielomianem generującym, realizując dzielenie w ciele GF(2).

```
dividend = (dividend & ((1 << current_shft) - 1)) | (rem <<
current_shft)
```

- Symuluje dzielenie w słupku:
  - `(1 << current_shft) - 1`: Tworzy maskę bitową z jedynkami do pozycji `current_shft`.
  - `dividend & mask`: Usuwa *n*-bitowy fragment z przodu dywidendy.
  - `rem << current_shft`: Wstawia wynik XOR na miejsce usuniętego fragmentu.

```
toBin(dividend).zfill(total_bits)
```

- Zapewnia stałą długość ciągu binarnego w wizualizacji, dodając zera wiodące.

#### Wizualizacja

Funkcja wyświetla każdy krok dzielenia binarnego, podświetlając aktualnie przetwarzane bity (za pomocą kodów ANSI) oraz pokazując operacje XOR i przesunięcia.

---

### check_crc(codeword, key)

Sprawdza poprawność słowa kodowego (dane + CRC) przez ponowne dzielenie binarne przez wielomian generujący.

#### Parametry

- `codeword` (str): Słowo kodowe (dane + CRC) jako ciąg binarny.
- `key` (str): Wielomian generujący jako ciąg binarny.

#### Zwraca

- Brak. Funkcja wyświetla:
  - Resztę z dzielenia (jeśli `0`, brak błędów).
  - Komunikat o powodzeniu lub niepowodzeniu weryfikacji CRC.

#### Przykład

```
>>> check_crc("100010101", "101")
# Wyświetla:
Reszta po sprawdzeniu: 0
✅ CRC check passed: brak błędów.
```

#### Kluczowe linie kodu

```
current_shft = dividend.bit_length() - n
```

- Określa, czy dywidenda ma wystarczającą liczbę bitów do kolejnej operacji XOR.

```
rem = (dividend >> current_shft) ^ gen
```

- Pobiera fragment dywidendy i wykonuje XOR z wielomianem generującym.

```
dividend = (dividend & ((1 << current_shft) - 1)) | (rem <<
current_shft)
```

- Aktualizuje dywidendę po każdej iteracji, podobnie jak w `CRC_visual`.

```
if dividend == 0:
    print("✅ CRC check passed: brak błędów.")
else:
    print("❌ CRC check failed: wykryto błąd.")
```

- Sprawdza, czy reszta wynosi `0` (brak błędów) i wyświetla odpowiedni komunikat.

---

### 4.2 Kod źródłowy

Poniżej znajduje się pełny kod źródłowy programu w Pythonie, który wizualizuje proces obliczania CRC-32 oraz weryfikacji sumy kontrolnej.

```
def toBin(num):
    return bin(num)[2:] if num != 0 else "0"


def toDec(bin_str):
    return int(bin_str, 2) if bin_str else 0


def CRC_visual(data, key):
    print("🔍 Wizualizacja obliczania CRC:\n")

    n = len(key)
    if n == 0:
        print("Error: Key cannot be empty")
        return

    gen = toDec(key)
    code = toDec(data)
    data_len = len(data)
    max_shift = 0

    """Dodajemy n ( długość_wielomianu_generującego - 1 ) do ciągu
danych -> przygotowujemy miejsce pod reszte"""
    dividend = code << (n - 1)
    total_bits = data_len + n - 1

    print(f"Wejściowe dane:    {data}")
    print(f"Generator (key):   {key}")
    print(f"Dane + zera:       {toBin(dividend).zfill(total_bits)}")
    print(f"Rozpoczynam dzielenie binarne...\n")

    while True:
```

```
        """Sprawdzamy czy aktualny dividend jest dłuższy niż wartość
wielomiana generującego ( przez który jest on dzielony )"""
        current_shft = dividend.bit_length() - n
        if current_shft > max_shift:
            max_shift = current_shft
        if current_shft < 0:
            break

        """Bierzemy n najbardziej znaczących bitów z ciagu
informacyjnego w celu XOR z wielomianem generujacym"""
        portion = dividend >> current_shft
        """W celach wizualizacji"""
        portion_bin = toBin(portion).zfill(n)
        gen_bin = toBin(gen).zfill(n)
        """Wykonuje XOR n-najbardziej znaczacych bitow z ciagu
informacyjnego z wielomianem generujacym"""
        rem = portion ^ gen
        """W celach wizualizacji"""
        rem_bin = toBin(rem).zfill(n)

        dividend_bin = toBin(dividend).zfill(total_bits)

        start_idx = total_bits - dividend.bit_length() # gdzie
zaczynaja sie znaczace bity ( bez zer na początku )
        portion_start = start_idx
        portion_end = portion_start + n  # n bitów, które bierzemy do
XOR

        colored_dividend = (
            dividend_bin[:portion_start]
            +
f"\033[32m{dividend_bin[portion_start:portion_end]}\033[0m"
            + dividend_bin[portion_end:]
        )


        print(f"Divident bits : {colored_dividend}")
        print(f"  XORing:       {portion_bin}")
        print(f"          XOR   {gen_bin}")
        print(f"        =       {rem_bin}")
        print(f"Posuwam się dalej w prawo (shift =
{current_shft})\n")

        """Wstawiamy reszte z dzielenia na miejscu pierwszych n bitów
w ciagu informacyjnym ( umitacja dzielenia w słupku )"""
        dividend = (dividend & ((1 << current_shft) - 1)) | (rem <<
current_shft)

    """Ostateczne wyniki"""
    remainder = dividend
    codeword = (code << (n - 1)) | remainder

    print("🔚 Koniec dzielenia.")
    print(f" 🔸 Reszta (CRC):     {toBin(remainder).zfill(n - 1)}")
    print(f" 🔸 Codeword (dane + CRC):
```

```
{toBin(codeword).zfill(total_bits)}\n")

    return toBin(codeword)


def check_crc(codeword, key):
    print("🔍 Sprawdzanie poprawności odebranego kodu...\n")

    n = len(key)
    gen = toDec(key)
    code = toDec(codeword)
    dividend = code

    while True:
        current_shft = dividend.bit_length() - n
        if current_shft < 0:
            break
        rem = (dividend >> current_shft) ^ gen
        dividend = (dividend & ((1 << current_shft) - 1)) | (rem <<
current_shft)

    print(f"Reszta po sprawdzeniu: {toBin(dividend)}")
    if dividend == 0:
        print("✅ CRC check passed: brak błędów.")
    else:
        print("❌ CRC check failed: wykryto błąd.")


if __name__ == "__main__":
    data =
"11101000110010101110011011101001000111010001111001010000110 10"
    generator_hex = "0x04C11DB7"
    generator_bin = bin(int(generator_hex, 16))[2:].zfill(32)
    generator_bin = "1" + generator_bin  # upewniamy się, że jest 33-
bitowy

    """Do testow"""
    # data = "1000101"
    # generator_bin = "101"

    print("-" * 10 + " CRC wizualizacja z 802.3/802.11 polynomem " +
"-" * 10)
    print(f"Generator G(x): {generator_bin}")
    print("-" * 60)

    codeword = CRC_visual(data, generator_bin)
    check_crc(codeword, generator_bin)
```

---

### 4.3. Omówienie kodu

Kod składa się z czterech głównych funkcji:

1. **`toBin(num)`**: Konwertuje liczbę dziesiętną na ciąg binarny, usuwając prefiks `0b`. Zwraca `"0"` dla zera.
2. **`toDec(bin_str)`**: Konwertuje ciąg binarny na liczbę dziesiętną.
3. **`CRC_visual(data, key)`**: Implementuje algorytm CRC-32 z wizualizacją krok po kroku:
    - Przyjmuje dane wejściowe i wielomian generujący jako ciągi binarne.
    - Dopełnia dane zerami (przesunięcie bitowe w lewo o $n - 1$).
    - Wykonuje dzielenie binarne z operacjami XOR.
    - Wyświetla kolejne kroki dzielenia, podświetlając aktualnie przetwarzane bity.
    - Zwraca słowo kodowe (dane + CRC).
4. **`check_crc(codeword, key)`**: Weryfikuje poprawność słowa kodowego przez ponowne dzielenie. Jeśli reszta wynosi 0, dane są poprawne.

Program używa wielomianu generującego CRC-32 w formacie zgodnym z IEEE 802.3 (`0x04C11DB7` w postaci heksadecymalnej, z dołączonym bitem $x^{32}$).

## 4.3. Wyniki uruchomienia

Przykładowe dane wejściowe:

- Dane: `00111010001100101011100110111010010001110100011111001010001101000000000000`
- Wielomian generujący: `100000100110000010001110110110111` (33 bity)

**Wynik działania funkcji `CRC_visual`:**

```
---------- CRC wizualizacja z 802.3/802.11 polynomem ----------
Generator G(x): 100000100110000010001110110110111
------------------------------------------------------------
🔍 Wizualizacja obliczania CRC:

Wejściowe dane:    00111010001100101011100110111010010001110100011111001(
Generator (key):   100000100110000010001110110110111
Dane + zera:
00111010001100101011100110111010010001110100011111001010001101000000000000
Rozpoczynam dzielenie binarne...

Divident bits :
00111010001100101011100110111010010001110100011111001010001101000000000000
  XORing:       11101000110010101110011011101001
          XOR   10000010011000001000111011011011
        =       01101010101010100110100001100101
Posuwam się dalej w prawo (shift = 59)

Divident bits :
00011010101010101001101000001100101001110100011111001010001101000000000000
  XORing:       11010101010101001101000001100101
          XOR   10000010011000001000111011011011
        =       01010111001101000101111010111101
Posuwam się dalej w prawo (shift = 58)

Divident bits :
00001010111001101000101111010111110101110100011111001010001101000000000000
```

```
  XORing:       10101110011010001011101011111010
          XOR   10000010011000001000111011010111
        =       00101100000010000011001110100110
Posuwam się dalej w prawo (shift = 57)

Divident bits :
000000101100000010000011001110100110111101000111100101000110100000000000
  XORing:       10110000001000001100111010011011
          XOR   10000010011000001000111011010111
        =       00110010010000000100000001000000
Posuwam się dalej w prawo (shift = 55)

Divident bits :
000000001100100100000001000000010000000101000111100101000110100000000000
  XORing:       11001001000000010000000100000010
          XOR   10000010011000001000111011010111
        =       01001011011000011000111111010101
Posuwam się dalej w prawo (shift = 53)

Divident bits :
000000000100101101100001100011111101101011000111100101000110100000000000
  XORing:       10010110110000110001111110110101
          XOR   10000010011000001000111011010111
        =       00010100101000111001000101101100
Posuwam się dalej w prawo (shift = 52)

Divident bits :
000000000000101001010001110010001011011100000111100101000110100000000000
  XORing:       10100101000111001000101101110000
          XOR   10000010011000001000111011010111
        =       00100111011111000000010110101111
Posuwam się dalej w prawo (shift = 49)

Divident bits :
000000000000001001110111111000000010110101011111111001010001101000000000
  XORing:       10011101111100000001011010101111
          XOR   10000010011000001000111011010111
        =       00011111100100001001100001110100
Posuwam się dalej w prawo (shift = 47)

Divident bits :
000000000000000001111111001000010011000011101000110010100011010000000000
  XORing:       11111100100001001100001110100011
          XOR   10000010011000001000111011010111
        =       01111110111001000100110101111001
Posuwam się dalej w prawo (shift = 44)

Divident bits :
000000000000000000111111011100100010011010111100010101000110100000000000
  XORing:       11111101110010001001101011110001
          XOR   10000010011000001000111011010111
        =       01111111101010000001010000101010
Posuwam się dalej w prawo (shift = 43)

Divident bits :
```

```
00000000000000000001111111101010000001010000101010110100011010000000000
  XORing:       11111111010100000010100001010101
          XOR   10000010011000001000111011011 0111
        =       01111101001100001010011010001 1100
Posuwam się dalej w prawo (shift = 42)

Divident bits :
00000000000000000001111110100110000101001101000111000100011010000000000
  XORing:       11111010011000010100110100011 1000
          XOR   10000010011000001000111011011 0111
        =       01111000000000011100001111000 1111
Posuwam się dalej w prawo (shift = 41)

Divident bits :
00000000000000000000111100000000011100011110001111100011010000000000
  XORing:       11110000000000111000011110001 1111
          XOR   10000010011000001000111011011 0111
        =       01110010011000110000100101010 1000
Posuwam się dalej w prawo (shift = 40)

Divident bits :
00000000000000000000011100100110001100001001010101000000110100000000000
  XORing:       11100100110001100001001010100 0000
          XOR   10000010011000001000111011011 0111
        =       01100110101001101001110001110 0111
Posuwam się dalej w prawo (shift = 39)

Divident bits :
00000000000000000000001100110101001101001110001110011100110100000000000
  XORing:       11001101010011010011100011100 1110
          XOR   10000010011000001000111011011 0111
        =       01001111001011011011001111001 1001
Posuwam się dalej w prawo (shift = 38)

Divident bits :
00000000000000000000000010011110010110110110110001111001011010000000000
  XORing:       10011110010110110110110001111 0010
          XOR   10000010011000001000111011011 0111
        =       00011100001110111110001010100 0101
Posuwam się dalej w prawo (shift = 37)

Divident bits :
00000000000000000000000000011100011101111100010101000101110100000000000
  XORing:       11100011101111110001010100010 1110
          XOR   10000010011000001000111011011 0111
        =       01100011101111111001101111001 1001
Posuwam się dalej w prawo (shift = 34)

Divident bits :
00000000000000000000000000001100011101111111001101111001100110000000000
  XORing:       11000111011111110011011110011 0011
          XOR   10000010011000001000111011011 0111
        =       01000101000111111011100101000 0100
Posuwam się dalej w prawo (shift = 33)
```

```
Divident bits :
00000000000000000000000000000000100010100011111101110010100001000000000000
  XORing:       10001010001111110111001010000100
          XOR   10000010011000001000111011011 0111
        =       00001000010111111111100010111111
Posuwam się dalej w prawo (shift = 32)

Divident bits :
00000000000000000000000000000000100001011111111111000101111110000000000
  XORing:       10000101111111111100010111111 10000
          XOR   10000010011000001000111011011 0111
        =       00000111100111110100101100100 0111
Posuwam się dalej w prawo (shift = 28)

Divident bits :
00000000000000000000000000000000000011110011111010010110010001111000000
  XORing:       11110011111010010110010001110 0000
          XOR   10000010011000001000111011011 0111
        =       01110001100010011110101010101 0111
Posuwam się dalej w prawo (shift = 23)

Divident bits :
00000000000000000000000000000000000011100011000100111101010101011101
  XORing:       11100011000100111101010101011 0110
          XOR   10000010011000001000111011011 0111
        =       01100001011100110101101110001 1001
Posuwam się dalej w prawo (shift = 22)

Divident bits :
00000000000000000000000000000000000011000010111001101011011100011001
  XORing:       11000010111001101011011100011 0010
          XOR   10000010011000001000111011011 0111
        =       01000000100001100011100111000 0101
Posuwam się dalej w prawo (shift = 21)

Divident bits :
00000000000000000000000000000000000010000001000011000111001110000101
  XORing:       10000001000011000111001110000 1010
          XOR   10000010011000001000111011011 0111
        =       00000011011011001111110101011 1101
Posuwam się dalej w prawo (shift = 20)

Divident bits :
00000000000000000000000000000000000001101101100111111101010111
  XORing:       11011011001111110101011110100 0000
          XOR   10000010011000001000111011011 0111
        =       01011001010111111011001011110 0111
Posuwam się dalej w prawo (shift = 14)

Divident bits :
00000000000000000000000000000000000000000000000000101100101011111110110010
  XORing:       10110010101111111011001011110 1110
          XOR   10000010011000001000111011011 0111
        =       00110000110111110011110000101 1001
Posuwam się dalej w prawo (shift = 13)
```

```
Divident bits :
00000000000000000000000000000000000000000000000011000110111110011111000
  XORing:       11000011011111001111000101100100
          XOR   10000010011000001000111011011 0111
        =       01000001000111000111110011010011
Posuwam się dalej w prawo (shift = 11)

Divident bits :
00000000000000000000000000000000000000000000000010000100011100011111000
  XORing:       10000100011100011111100110100110
          XOR   10000010011000001000111011011 0111
        =       00000000010110000111001000001 0001
Posuwam się dalej w prawo (shift = 10)

Divident bits :
00000000000000000000000000000000000000000000000000000000101100001110000
  XORing:       10110000111001000001000100000000
          XOR   10000010011000001000111011011 0111
        =       00110010100001001001111111011 0111
Posuwam się dalej w prawo (shift = 1)

🔚 Koniec dzielenia.
🔸 Reszta (CRC):     11001010000100100111111101101110
🔸 Codeword (dane + CRC):
00111010001100101011100110111010010001110100011110010100011010110010100 0

🔍 Sprawdzanie poprawności odebranego kodu...

Reszta po sprawdzeniu: 0
✅ CRC check passed: brak błędów.
```

Wyniki potwierdzają poprawność implementacji – reszta z dzielenia słowa kodowego wynosi 0, co oznacza brak błędów.

### 4.4. Testy z błędami

Aby zweryfikować zdolność wykrywania błędów, zmodyfikowano słowo kodowe przez zmianę ostatniego bitu w reszcie. Wynik:

```
---------- CRC wizualizacja z 802.3/802.11 polynomem ----------
Generator G(x): 10000010011000001000111011011 0111
----------------------------------------------------------------
🔍 Sprawdzanie poprawności odebranego kodu...

Reszta po sprawdzeniu: 1
❌ CRC check failed: wykryto błąd.
```

Testy potwierdzają, że CRC-32 skutecznie wykrywa błędy pojedynczego bitu, zgodnie z teoretycznymi założeniami.

---

## 5. Implementacja sprzętowa w MultiSim

[Multisim_video]

## 6. Literatura


- [1] Cyclic_redundancy_check_wikipedia
- [2] Mathematics_of_CRC
- [3] IEEE 802.3 - 2022