Cyclic Redundancy Check

Team members:
Atabek Mykyev: FYQWAV
Alidin Abylkasym uulu: NRUDXE
Begimai Kadyrova: JQ70KI
Myktybek Sattarov: W4OJEE

Introduction:

What is CRC?

CRC stands for Cyclic Redundancy Check. It is a method for checking errors in digital data. It attaches a small, calculated "check value" to the data, which is used to confirm the integrity of the data when it is retrieved later. If the check value does not match the original data, it indicates that the data has been corrupted, and the error can be corrected.

How does the CRC work? (The way it checks for errors)

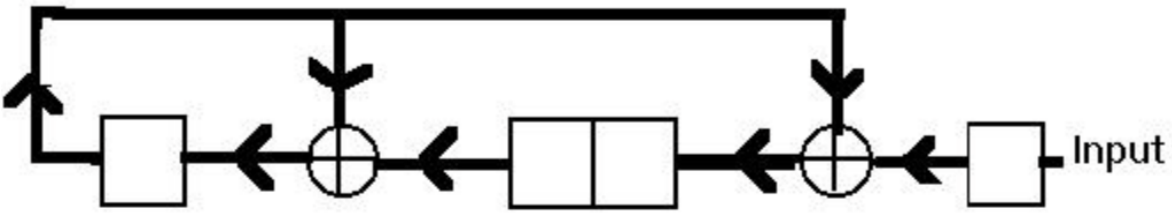
On the sender side, check bits are calculated based on the data being sent, and are appended to the data before it is transmitted.

When the receiver receives the data, it performs the same calculation on the received data, and compares the resulting check bits to the check bits that were appended to the data. If the two sets of check bits match, it means that the data has been transmitted without errors. If they do not match, it indicates that an error has occurred during transmission and the data should be retransmitted.

CRCs can be implemented using different polynomials, leading to different variants such as CRC-16, CRC-32, etc.
Each step of CRC algorithm is shown below:

The data is divided into a series of fixed-length blocks.

- A generator polynomial is chosen for the calculation.
- A shift register is initialized with a specific value, known as the initial value.



Shift Register for x^3+x^2+1

- The data blocks are processed one at a time. For each block:
- The block is divided into a series of bits, with the least significant bit being processed first.
- The bits are shifted through the shift register, one at a time.
- As the bits are shifted, they are also processed by an exclusive OR (XOR) gate. The XOR gate is connected to a specific tap in the shift register, which is determined by the specific polynomial being used. The output of the XOR gate is fed back into the input of the shift register, causing the bits in the shift register to be modified as they are shifted.
- Once all of the data bits have been processed, the contents of the shift register will be the CRC remainder.
- The remainder is appended to the original data and transmitted.

On the receiver side, the process is similar but with the received data and the appended remainder.

- The received data and the appended remainder are processed in the same way as the original data was processed at the sender side.
- The contents of the shift register at the end of this process are compared to the remainder that was appended to the data.
- If the contents of the shift register match the remainder, it means that the data has been transmitted without errors.
- If they do not match, it indicates that an error has occurred during transmission and the data should be retransmitted.

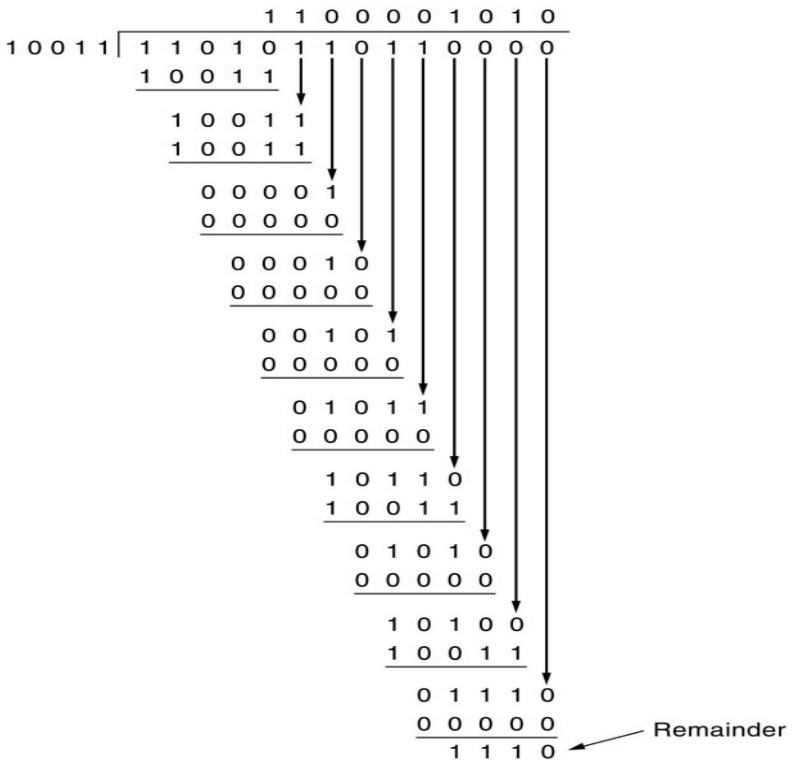
It can also be done in a simpler way with modulo-2 division method

This method is based on the mathematical concept of modular arithmetic and involves the use of a binary division algorithm to calculate the check bits.
The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. Just that instead of subtraction, we use XOR here.

n => Number of bits in (ex: 01010110)
k => Number of bits in the key obtained from generator polynomial.(ex: 101 ($1*x^2+0*x^1+1*x^0$))

- In every step a copy of the divisor is XORed with the k bits of the key.

- The result of the XOR operation (remainder) is (n-1) bits, which is used for the next step after 1 extra bit is pulled down to make it n bits long.
- In the end, we are left with (n-1)-bit remainder which we append to the original data and send.



What are the different implementations/variations?

There are several variations and implementations of the Cyclic Redundancy Check (CRC) algorithm, each with its own specific properties and characteristics. Some of the most common implementations and variations include:

1. CRC-8: This is a 8-bit version of the CRC algorithm that is commonly used in memory systems and other applications where space is at a premium.
2. CRC-16: This is a 16-bit version of the CRC algorithm that is commonly used in communication systems such as modems and X.25 networks.
3. CRC-32: This is a 32-bit version of the CRC algorithm that is commonly used in file transfer and other applications that require high levels of data integrity.

Each implementation has its own generator polynomial, which leads to a different number of check bits and different error detection properties. The choice of implementation depends on the specific requirements of the application and the trade-off between error detection capabilities and complexity.

Bit-oriented algorithm
A straightforward way to calculate a CRC is to examine the input data bit by bit, and add or subtract the coefficients of the CRC polynomial (pCRC) as needed. This is repeated until only a shorter number of bits remain, which are the remainder coefficients and therefore the desired CRC. This approach is effective because the goal is to only obtain the remainder, not the whole quotient, of the division.

A practical approach to calculate the CRC is to use a bit register with a width of N, where the input data bits are shifted in from the right. As soon as a 1 bit is shifted out from the left side, the contents of the register are XORed with the coefficients of the pCRC polynomial. This results in the same outcome as the previous method. The bit mask used in the XOR operation is called the CRCPOLY. Additionally, this approach is slightly improved by using the register to store only the effects of the XOR operations on the data stream, rather than the actual result. This means that the process starts with a register of all zeroes and shifts in 0s from the right. The register is XORed with the CRCPOLY when the bit shifted out is different from the bit seen in the data stream. This eliminates the need for "augmentation" with N 0-bits as they are not used in any operation.

For technical reasons, the process of calculating the CRC does not start with an empty register, but rather with a specific pattern called INITXOR. This pattern is used to detect and compensate for errors such as added or missing leading zeroes that would otherwise go unnoticed. The INITXOR pattern is usually set to all 1s. Similarly, after the computation is completed, a second pattern called FINALXOR is added to the CRC register. It is important to keep this in mind when using the result as a CRC value. However, these additional steps do not change the overall structure of the algorithm.

Table-driven algorithm
The bit-oriented algorithm for calculating the Cyclic Redundancy Check (CRC) is a simple and straightforward method, but it has some weaknesses. One of the main weaknesses is that it is not very efficient. It operates at the bit-level, resulting in one loop for each bit, regardless of the word-width of the machine that is being used. This means that it can be slow for large amounts of data or for machines with wide word-widths.

A table-driven algorithm is a way to improve the efficiency of the bit-oriented algorithm. The basic idea is to process units of multiple bits at once, rather than one bit at a time. This is done by shifting not only one bit at a time, but M bits at a time. The result is a pattern of M bits shifted out of the register, which have to be compared to corresponding M bits of the data stream. After that, a pattern (xor-mask) has to be applied to the M bits just shifted out and the N bits of the CRC register so that the first M bits match the M bits of the data stream.

The process of finding such a sum of CRCPOLYs is equivalent to calculating the bit-oriented approach, but it only has to be done once in advance. For every pattern of M bits shifted out and xored with the data bits, the corresponding mask of N bits that has to be applied to the CRC register can be stored in a table. This table is called the "CRC table" and it has 2^M lines with N bits each. The typical size for a CRC32 (N = 32) is M = 8, so that the units to be processed are bytes and the table has 2^8 = 256 entries. With M = 16 the process of calculating the CRC32 would be twice as fast, but the table would be 2^8 times larger.

In summary, the table-driven algorithm is more efficient than the bit-oriented algorithm because it processes multiple bits at a time and uses a precomputed table to quickly look up the necessary xor-masks, rather than calculating them on the fly. This reduces the number of operations needed to calculate the CRC, making the process faster.

Problem statement:

Given the data calculate its CRC value, append it to the data and check it for the errors. Implement the bit-oriented and table-driven algorithms of CRC calculation.

Solution Description:

Auxiliary functions and fields

In [1]:
```
CRC_POLY = 0x04C11DB7
CRC_INV = 0x5B358FD3
INIT_XOR = 0xFFFFFFFF
FINAL_XOR = 0xFFFFFFFF
```

INIT_XOR - a pattern which is meant to compensate for errors like erroneously added or left-out leading zeros which would otherwise remain undetected and it's usually all 1
FINAL_XOR - same as INIT_XOR, which is added after the computation.

Bit-oriented algorithm of calculation of the CRC

In [2]:
```
def crc32_bo(data):
    crc = INIT_XOR
    for byte in data:
        crc ^^= byte
        for _ in range(8):
            if crc & 1:
                crc = (crc >> 1) ^^ CRC_POLY
            else:
                crc = (crc >> 1)
    return crc ^^ FINAL_XOR
```

Explanation:

- The above given code does following:
- It iterates over data byte by byte.
- In each step, we XOR 'crc' with the current byte of 'data'.
- Then do the following 8 times:

- Check the least significant byte of 'crc' and
  - - if it is 1, then shift 'crc' with 1 and XOR it with a polynomial generator.
  - - otherwise we just shift it by 1.
- And finally we XOR 'crc' with FINAL_XOR and return it.

Table-driven algorithm of calculation of the CRC

In [3]:
```
def crc32_td(data):
    table = [0] * 256
    #part 1
    for i in range(256):
        crc = i
        for j in range(8):
            if crc & 1:
                crc = (crc >> 1) ^^ CRC_POLY
            else:
                crc = crc >> 1
        table[i] = crc

    #part 2
    crc = INIT_XOR
    for byte in data:
        crc = (crc >> 8) ^^ table[(crc & 0xFF) ^^ byte]

    return crc ^^ FINAL_XOR
```

Explanation:

PART 1

- At the beginning we create a 256 sized table with 0 initial value.
- The we iterate it and in each step we do the following:
  - -initialize 'crc' to the current index i of the loop
- -iterate 8 times and do the following:
- Check the least significant byte of 'crc' and
  - - if it is 1, then shift 'crc' with 1 and XOR it with a polynomial generator.
  - - otherwise we just shift it by 1.
- -Assign the value to the table[i].
-

PART 2

- We initialize 'crc' with pre-defined INIT_XOR.
- Then we iterate data byte by byte doing the following:
  - -a. Perform an XOR operation between 'crc' shifted right by 8, and the value of the table at index = XOR operation between 'crc' masked with 0xFF and the current byte.
  - -b. Assign the result of this operation to crc.
- Finally we return XOR of 'crc' and FINAL_XOR.

After getting CRC keys, we append them to the “data” with append_crc32_bo() and append_crc32_td() functions.

```
In [5]: # Bit-oriented algorithm of calculation of the CRC
def append_crc32_bo(data):
    data=int(data, 2)
    crc = crc32_bo(data.to_bytes(4, byteorder='big'))
    data = data.to_bytes(4, byteorder='big') + int(crc).to_bytes(4, byteorder='little')
    return data

# Table-driven algorithm of calculation of the CRC
def append_crc32_td(data):
    data=int(data, 2)
    crc = crc32_td(data.to_bytes(4, byteorder='big'))
    data = data.to_bytes(4, byteorder='big') + int(crc).to_bytes(4, byteorder='little')
    return data
```

Explanation:

- First we convert binary data to an integer
- Then we get CRC key of the data in big-endian format, with appropriate function
- We convert data back to byte format, then append the CRC key after converting it from int to byte in little-endian format
- The concatenated data is returned as the final output of the function.

And Finally our main function

```
def check_crc32(data):
    # Recalculate the CRC32 value of the data with the appended CRC32
    calculated_crc = crc32_td(data[:-4])
    appended_crc = int.from_bytes(data[-4:], byteorder='little')

    if calculated_crc != appended_crc:
        return None
    else:
        return data
```

Explanation:

- we calculate ‘crc’ with appropriate function removing the last 4 bytes (data[:-4]). This will give us the calculated CRC32 value.
- next we get appended CRC value.
- the compare the original data with calculated one and return whether it is true or not

REFERENCES:

https://web.archive.org/web/20110719042902/http://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2006-05/SAR-PR-2006-05_.pdf — Book dedicated to CRC. It contains different implementations of CRC and explains the CRC algorithm itself in detail.

http://www.ciphersbyritter.com/ARTS/CRCMYST.HTM — Article about the CRC (Brief explanation of different features)

https://zlib.net/crc_v3.txt - Painless guide to CRC Error Detection Algorithms

https://web.archive.org/web/20170720165847/http://www.drdobbs.com/an-algorithm-for-error-correcting-cyclic/184401662 — Error correcting CRC implementation

https://www.geeksforgeeks.org/modulo-2-binary-division/ - Modulo-2 division method