

CSE 3231

Computer Networks

Chapter 3

The Data Link Layer

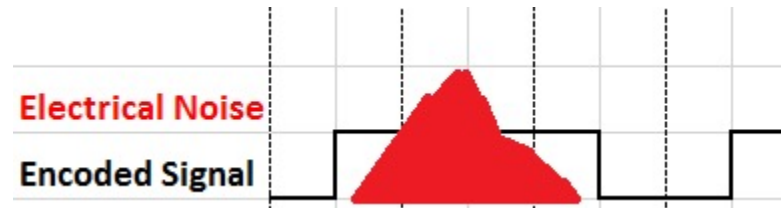
part 2 - Error Detection

William Allen, PhD
Spring 2022

Error Detection

- *Bit errors* can be introduced into frames
 - they are often caused by random electrical interference and/or thermal noise

What can we
do about this?



- There are two ways to handle errors:
 - Detect the error and reject the frame
 - Detect and correct the error in the frame

Error Detection and Correction

- Error *detection* refers to the ability to determine that a frame contains an error
 - This usually requires that some *additional information* is transmitted with the frame
- Error *correction* refers to the ability to correct the detected error
 - First, we must be able to detect the error
 - Then, correct it by repairing or replacing the data in the frame that was affected

Error Detection and Correction

- Two approaches can be used when the receiver detects an error
 - Notify the sender that the message was corrupted, so the sender can **send again**
 - If errors in the network are rare, then the retransmitted message will usually be error-free
 - By using an **error detection and correction** algorithm, the receiver can **reconstruct** the message without waiting for a retransmission
 - this avoids the delay for retransmission and the impact on throughput from sending duplicate frames

Error Correction

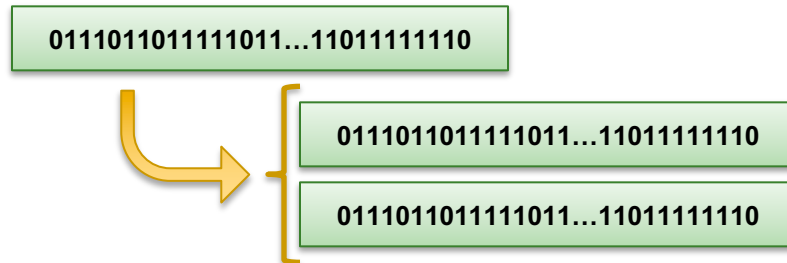
- There are several error correction algorithms:
 - Hamming codes
 - Binary convolutional codes
 - Reed-Solomon codes
 - Low-Density Parity Check codes
- Most protocols *do not* use these approaches
 - For successful error correction, even more information must be sent with the original data so that the errors can be detected *and corrected*
 - This decreases the amount of data that will be delivered over time; in a network with few errors it is more efficient to just resend corrupted frames

Error Detection

- The goal of error detection is to provide a **high probability of detecting errors**, with a relatively **low number of redundant bits**
 - The additional bits are referred to as *redundant* because they do not add information to the message and are just used for error detection

Error Detection

- The extreme case is to transmit two copies of the data



If both copies are identical, there may be no error, otherwise there was an error in the transmission

Clearly not practical as it reduces throughput to $\frac{1}{2}$ as much data per second

- Instead, we should provide error detection using up to “ k ” redundant bits for an n -bit message (hopefully with $k \ll n$).

Error Detection

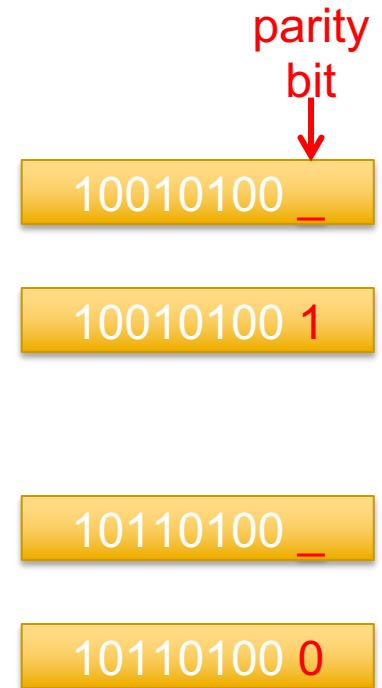
- Common techniques for detecting transmission errors
 - CRC (Cyclic Redundancy Check)
 - Used in Ethernet, HDLC, DDCMP, CSMA/CD, Token Ring
 - Other approaches
 - Two Dimensional Parity (BISYNC)
 - Checksum
 - MD5, SHA-1, Modulus (e.g., Luhn algorithm)
 - Often used in network protocol headers

Simple (1-bit) parity

- Simple (one-dimensional) parity, usually involves **adding one extra bit** to group of bits to balance the number of 1's in the group.
- **Odd parity** – expects an odd number of 1's
 - if the number of 1's in a group is even, set the extra bit to 1 to give an odd number of 1's overall
 - if the number of 1's is odd, set the extra bit to 0
- **Even parity** – expects an even number of 1's
 - if the number of 1's in a byte is odd, set the extra bit to 1 to give an even number of 1's overall, otherwise set the extra bit to 0

One-dimensional Parity

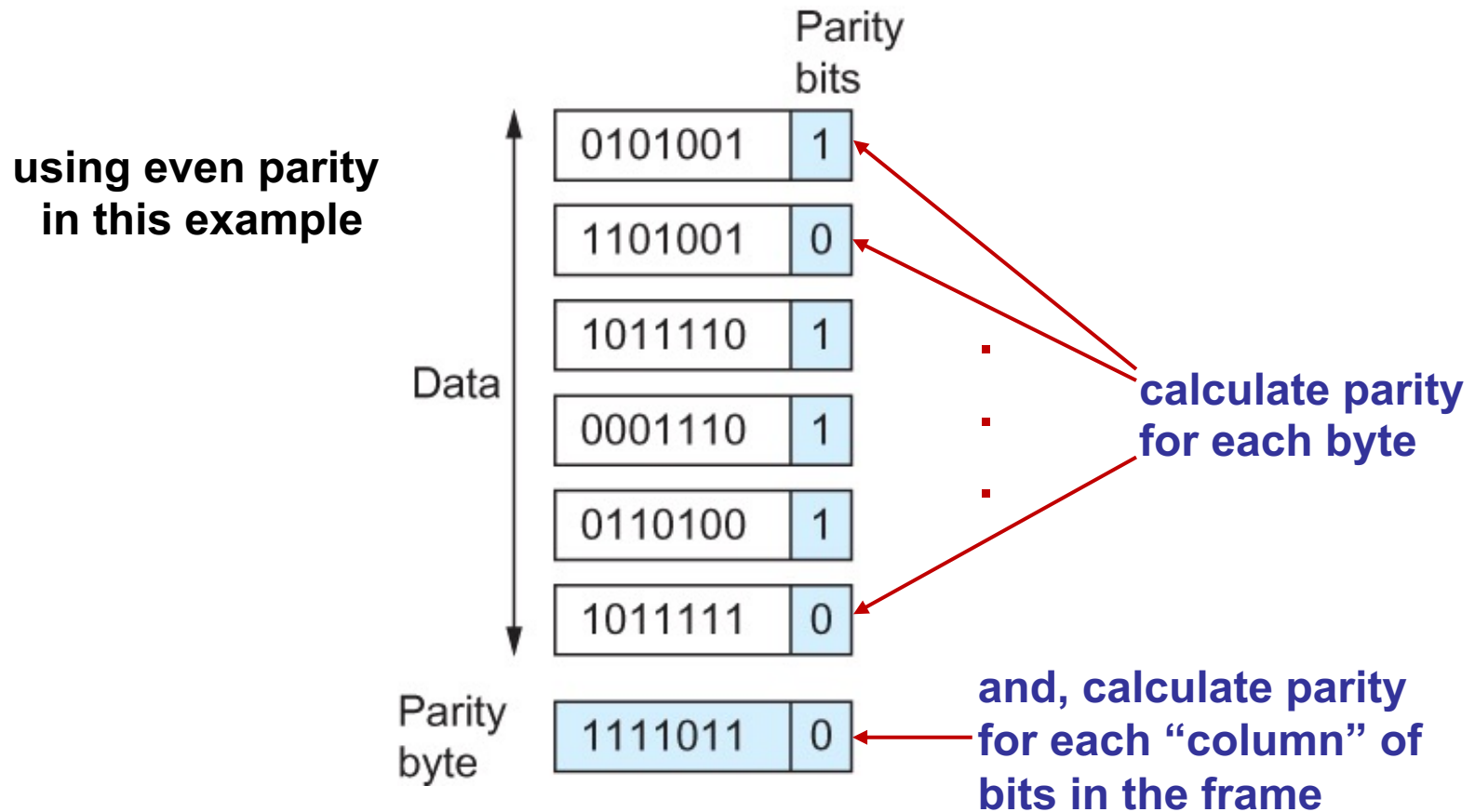
- Assume 8-bit data, **even** parity
 - if the number of 1's is odd, set the parity bit to 1, making the overall number of 1's even
 - if the number of 1's is even, set the parity bit to 0 so that the overall number of 1's stays even
- Odd parity is similar, but the goal is an odd number of 1's



Two-dimensional parity

- However, one-dimensional parity does not always detect errors of more than one bit
 - even parity says: 00101100 p1 == 01001100 p1
- Two-dimensional parity organizes the data in a two-dimensional array and does a parity calculation for each of the "rows" and each of the "columns" in the array
 - This results in an extra parity row for the entire frame, in addition to a parity bit for each row
 - Two-dimensional parity can detect all 1-, 2-, and 3-bit errors and most 4-bit errors

Two-dimensional parity

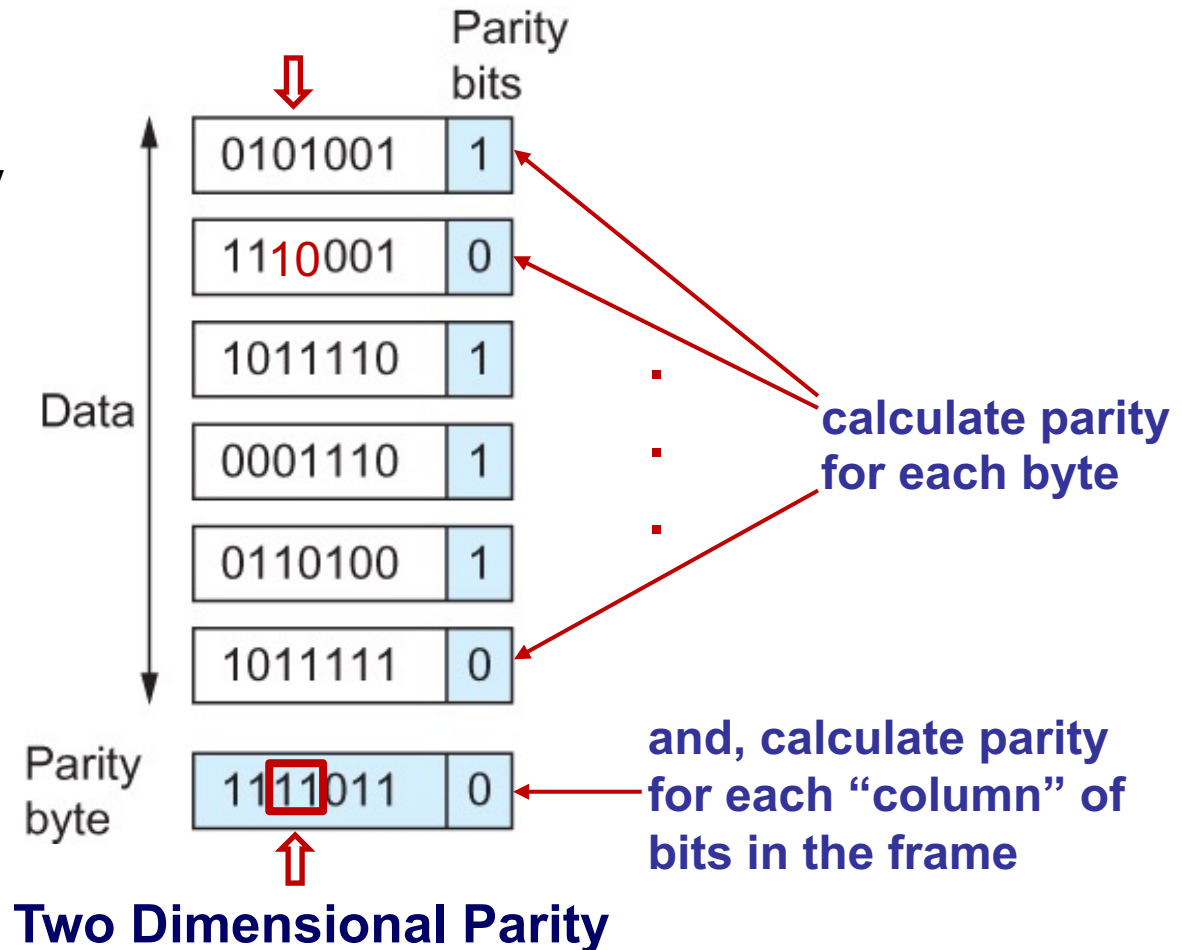


Two Dimensional Parity

Two-dimensional parity

using even parity
in this example

If two bits in a row
were flipped, that
row's parity is
still correct, but
the columns
where those bits
are located will no
longer be correct.



Checksum Algorithm

- Perform some calculation on the bytes that will be transmitted and add it to the end of the frame
 - The calculation result is called the *checksum*
- The receiver performs the same calculation on the received data and compares that result with the checksum received from the sender
- If any transmitted data, including the checksum itself, is corrupted, the results will not match and the receiver "knows" that an error occurred
 - one example is the Cyclic Redundancy Check (CRC)

Error Detection – Checksums

A checksum algorithm performs a calculation on a collection of data and produces a remainder value that is used for error detection:

- The **Luhn Algorithm** detects errors in:
 - International Mobile Equipment Identity (IMEI)
 - ID numbers in some countries/companies
 - Some Credit Card numbers
- The **IP Header** checksum has the following benefits
 - Improved error detection over parity bits
 - Detects bursts of up to N sequential errors
 - Detects random errors with probability $1-2^N$

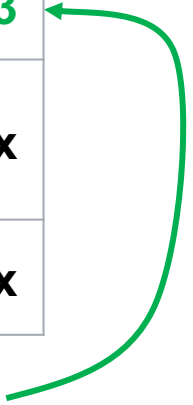
Luhn Algorithm

Calculates a checksum for verification of credit card numbers, IMEI numbers, etc.

Starting from the right, double every other digit and then **sum the digits**, **multiply the sum by 9** and **calculate the remainder** ($\%10$)

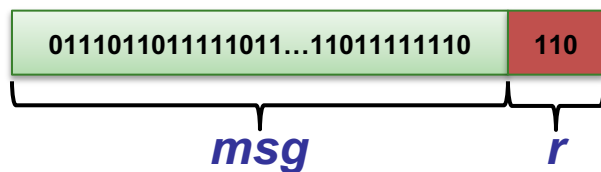
Account number	7	9	9	2	7	3	9	8	7	1	3
Double every other number	7	18	9	4	7	6	9	16	7	2	x
Sum digits of each number	7	1+8	9	4	7	6	9	1+6	7	2	x

Sum = 67, **9x67 = 603**, **603 % 10 = 3**

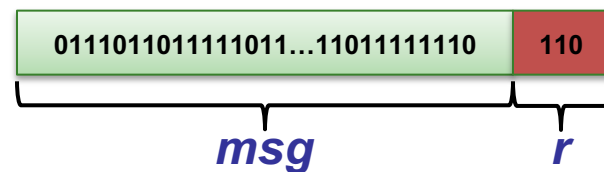
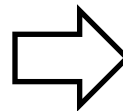


Error Detection

- The extra bits are *redundant*
 - They add no new information to the message
 - They are derived from the original message by using some algorithm known by both the sender and receiver



Sender calculates *r* from *msg* and sends *r* with *msg*

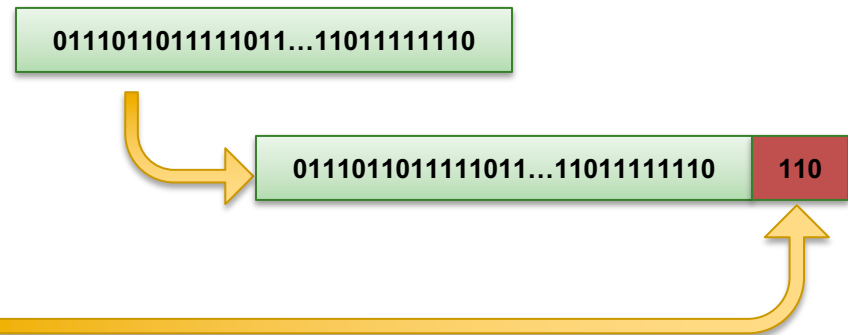


Receiver computes *r'* from the *msg* it received. If *r'* matches *r*, no error occurred

Error Detection

- With Ethernet, for instance, each frame carrying 1500 bytes (12,000 bits) includes an additional 32-bits for **error detection**.

For Ethernet, the error detection algorithm is referred to as **CRC-32** (where CRC stands for Cyclic Redundancy Check)



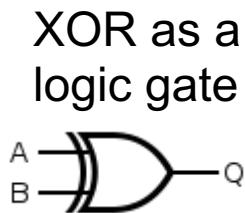
Polynomial Arithmetic

- A polynomial can represent a series of binary values where each *coefficient* represents a bit
 - for example: $1 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x^1 + 1$ represents **1101**
- We can **divide** a polynomial $M(x)$ by another polynomial $C(x)$, if the *degree* of $M(x)$ is higher than the degree than $C(x)$
 - the degree of $M(x)$ is the highest power of x in $M(x)$
- The remainder when $M(x)$ is divided by $C(x)$ is obtained by modular division: $M(x) \% C(x)$
 - modular division uses the **exclusive-OR (XOR)** operation on each corresponding pair of coefficients

XOR

XOR acts on two (or more) inputs

- the output is true *if one and only one* of the inputs are true
- can be expressed as a combination of AND, OR and NOT: $(A \cdot \sim B) + (\sim A \cdot B)$
- symbol for XOR: $A \oplus B$



input A	input B	output $A \oplus B$
T	T	F
T	F	T
F	T	T
F	F	F

XOR

XOR of:

10110100

01100010

11010110

When the two bits in a column are the same, the result is 0, when they are not, it is 1

A Practical Checksum Approach

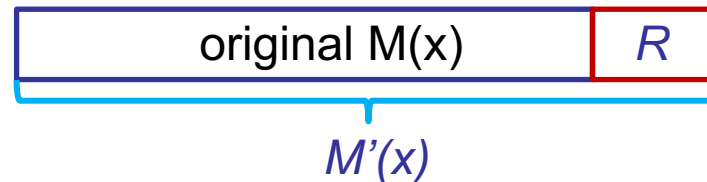
- Assume $M(x)$ is a *message* polynomial and $C(x)$ is a *divisor* polynomial, the operation $M(x) \% C(x)$ will produce a remainder R
- When $M(x)$ is received, the receiver computes $M(x) \% C(x)$ and if the remainder is equal to R , then no error has occurred
- The sender and the receiver need to share the *divisor* polynomial $C(x)$ and R to detect errors
 - If we can append R to the message before it is sent and use a standard polynomial for $C(x)$, then the receiver has all it needs to check for errors in $M(x)$

Choosing the Polynomial $C(x)$

- A checksum algorithm can be designed to reduce the number of redundant bits and maximize the range of errors detected
 - depending on the polynomial chosen, certain types of errors can reliably be detected
 - we can choose the number of bits needed for the checksum by balancing the types of errors we need to detect with the size of the polynomial $C(x)$ we use
 - as stated before, this will not detect all errors unless $C(x) = M(x)$ (i.e., they are identical), but a reasonable range of errors can be detected with 16 to 32 bits

How does this work?

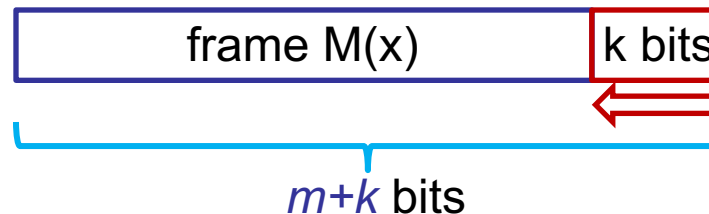
- We find the remainder from $M(x) \% C(x)$ and then shift $M(x)$ to the left by k bits so we can append that remainder R onto the end of $M(x)$ to create a new binary string $M'(x)$



- This means that $M'(x)$ can be divided by $C(x)$ to produce a result of 0 since the remainder from $M(x) \% C(x)$ is now included in $M'(x)$
 - this is the approach used by Ethernet for its CRC (Cyclic Redundancy Check) algorithm

Cyclic Redundancy Check (CRC)

- Assume $M(x)$ is a frame with m bits
- Let the divisor polynomial $C(x)$ have k bits ($k < m$), thus the degree of $C(x)$ is k
 - shift $M(x)$ left by k , raising the power of each bit of $M(x)$ by x^k and appending k zero bits to the low-order end of the frame
 - the result is a polynomial $x^k M(x)$ that contains $m+k$ bits



Example

- Assume $M(x)$ is a 4-bit message: **1101**

$$M(x) = 1*x^3 + 1*x^2 + 0*x^1 + 1*x^0 = 1101$$

- Assume $k = 2$

- When we multiply each number in the polynomial $M(x)$ by x^k , it increases the exponent of each x by 2

$$x^k M(x) = 1*x^{3+2} + 1*x^{2+2} + 0*x^{1+2} + 1*x^{0+2}$$

$$x^k M(x) = 1*x^5 + 1*x^4 + 0*x^3 + 1*x^2 + 0*x^1 + 0*x^0$$

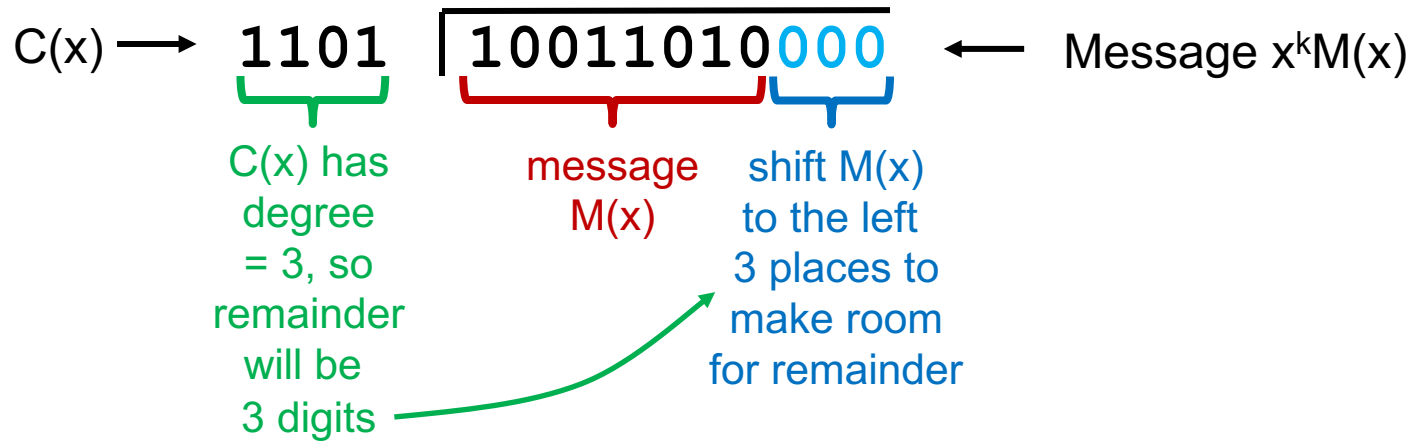
$$x^k M(x) = 110100$$

Cyclic Redundancy Check (CRC)

- Divide the bit string $x^kM(x)$ by the bit string $C(x)$ using modulo 2 division
 - modulo 2 division is similar to long division, except it uses modulo 2 subtraction to get the remainder
 - modulo 2 subtraction uses the XOR (eXclusive OR) operation instead of integer subtraction
- Insert the remainder (which is k or fewer bits) into the last k bit positions in the string $x^kM(x)$ to produce $M'(x)$ (add leading zeros, if needed)
 - $M'(x)$ includes both the original frame and the k -bit checksum value

Cyclic Redundancy Check (CRC)

- $M(x)$ is an 8-bit message: **10011010**
 - $C(x)$ is a polynomial of degree 3 (i.e., $k=3$):
$$x^3 + x^2 + 1 = 1101$$
 - First, shift $M(x)$ left by appending 3 zeros ($k=3$) to the right end of the message
10011010000
 - Next, divide $C(x)$ into $M(x)$, using bitwise XOR on each bit position to produce the remainder
 - Then, copy the remainder into the 3 zeros at the right end of $M(x)$, creating $M'(x)$
 - Transmit $M'(x)$ to the receiving node



$$\begin{array}{r}
 \text{C(x)} \longrightarrow 1101 \quad \overline{) \begin{array}{c} 1 \\ 10011010000 \end{array}} \longleftarrow \text{Message } x^k M(x) \\
 \begin{array}{r}
 \text{to subtract, we do an} \\
 \text{exclusive OR on the bits}
 \end{array}
 \quad \begin{array}{r}
 \underline{1101} \\
 1001
 \end{array}
 \quad \begin{array}{l}
 \downarrow \\
 \text{then we carry down the} \\
 \text{next bit, just like division}
 \end{array}
 \end{array}$$

$$\begin{array}{r}
 \text{C(x)} \longrightarrow 1101 \quad \overline{) \begin{array}{c} 11 \\ 1001101000 \end{array}} \quad \longleftarrow \text{Message } x^k M(x) \\
 \underline{1101} \\
 1001 \\
 \underline{1101} \\
 1000
 \end{array}$$

then we carry down the next bit, just like division

to subtract, we do an eXclusive OR on the bits

$$\begin{array}{r}
 \text{C(x)} \longrightarrow 1101 \quad \overline{) \begin{array}{c} 111 \\ 10011010000 \end{array}} \quad \longleftarrow \text{Message } x^k M(x) \\
 \begin{array}{r}
 1101 \\
 \hline
 1001 \\
 1101 \\
 \hline
 1000 \\
 1101 \\
 \hline
 1011
 \end{array}
 \end{array}$$

then we carry down the next bit, just like division

to subtract, we do an eXclusive OR on the bits

$$\begin{array}{r}
 \text{C(x)} \longrightarrow 1101 \overline{) 1111 \text{ } 1001101 \text{ } 0000} \longleftarrow \text{Message } x^k M(x) \\
 \underline{1101} \\
 1001 \\
 \underline{1101} \\
 1000 \\
 \underline{1101} \\
 1011 \\
 \underline{1101} \\
 1100
 \end{array}$$

then we carry down the next bit, just like division

to subtract, we do an eXclusive OR on the bits

$$\begin{array}{r}
 \text{C(x)} \longrightarrow 1101 \quad \overline{) \begin{array}{l} 1111100 \\ 10011010000 \end{array}} \quad \longleftarrow \text{Message } x^k M(x) \\
 \begin{array}{r}
 1101 \\
 \hline
 1001 \\
 1101 \\
 \hline
 1000 \\
 1101 \\
 \hline
 1011 \\
 1101 \\
 \hline
 1100 \\
 1101 \\
 \hline
 1000
 \end{array}
 \end{array}$$

then we carry down the next bit, just like division

since it's not divisible, we carry down a 2nd bit

since it's not divisible, we carry down a 3rd bit

to subtract, we do an eXclusive OR on the bits

we can ignore this result

$$\begin{array}{r}
 11111001 \\
 \overline{) 10011010000} \\
 \underline{1101} \\
 1001 \\
 \underline{1101} \\
 1000 \\
 \underline{1101} \\
 1011 \\
 \underline{1101} \\
 1100 \\
 \underline{1101} \\
 1000 \\
 \underline{1101} \\
 0101
 \end{array}$$

$C(x) \rightarrow 1101$ Message $x^k M(x)$

to subtract, we do an eXclusive OR on the bits

the Remainder is the CRC value we need but, we only use the last k-bits of the result

What Does the Receiver See?

- The remainder **101** is inserted into the last k ($k=3$) bit positions to create $M'(x)$ where $M'(x) = 10011010**101**$
- Then, $M'(x)$ is transmitted to the receiver
- The receiver can check for errors by dividing the same $C(x)$ into $M'(x)$
 - If no errors, the remainder should be **0**

$C(x) \rightarrow 1101$
 $\overline{11111001}$
 $\overline{1001101010101}$
 $\leftarrow M'(x)$

1101

1001

1101

1000

1101

1011

1101

1100

1101

1101

1101

0000

Since these bits were the remainder from $M(x) \% C(x)$, they will negate the remainder from $M'(x) \% C(x)$

Remainder:
 No Error

What if an error occurred?

- Assume one bit of $M'(x)$ was flipped:
 10011010 becomes 10011011
while in transit to the receiver
- Can we detect the error?

$$\begin{array}{r}
 \begin{array}{l} C(x) \longrightarrow 1101 \end{array} \quad \begin{array}{r} 1111100 \\ \hline 1001101 \\ 1101 \\ \hline 1001 \\ 1101 \\ \hline 1000 \\ 1101 \\ \hline 1011 \\ 1101 \\ \hline 1100 \\ 1101 \\ \hline 0101 \end{array}
 \end{array}$$

The degree is less than 3 ($0101 = x^2 + 1$) so we can't divide by $C(x)$ and must stop here

We have a non-zero remainder, which indicates that an error occurred

Error

$M'(x)$ with an error

Error Detection Properties of $C(x)$

- The following types of errors can be detected by a $C(x)$ with these properties:
 - All **single-bit errors**, as long as the x^k and x^0 terms have nonzero coefficients (i.e., the first & last terms)
 - All **double-bit errors**, as long as $C(x)$ has a factor with at least three terms
 - Any **odd number of errors**, if $C(x)$ contains $(x+1)$
 - Any **“burst” error** (i.e., sequence of consecutive error bits) where the length of the burst is **less than k bits**.
 - for example, Ethernet’s CRC-32 field contains $k=32$ bits
 - most burst errors of larger than k bits can also be detected

Cyclic Redundancy Check (CRC)

- Six generator polynomials that have become international standards are:
 - CRC-8 = x^8+x^2+x+1
 - CRC-10 = $x^{10}+x^9+x^5+x^4+x+1$
 - CRC-12 = $x^{12}+x^{11}+x^3+x^2+x+1$
 - CRC-16 = $x^{16}+x^{15}+x^2+1$
 - CRC-CCITT = $x^{16}+x^{12}+x^5+1$
 - CRC-32 (used by Ethernet) =
 $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

Is CRC over the whole frame?

- Not entirely. For example, in Ethernet the CRC checks the header fields and the data, but not the Preamble section
 - The Preamble bits are used for synchronization and if that part is corrupted the frame is ignored anyway

