PACKETS, FRAMES, AND ERROR DETECTION

Packets Versus Hardware Frames

- A *packet* is a small block of data
- There is no universal agreement on the exact format of a packet
  - Packet details (e.g., size and format) are hardware dependent
- The term *frame* is used to denote the definition of a packet used with a specific type of network
- Example:
  When using RS-232 to send a block of characters:
  - RS-232 is a character-oriented transmission scheme
  - It specifies how to encode bits and transmit an individual character, but it does not include a mechanism that allows the sender to signal the end of a block of characters
  - The sender and receiver must agree on how they will specify the start and the end of each frame, e.g., by adding to the data of frames some special characters to mark the start and the end of each frame
    - ∗ Start Of Header ("soh") ASCII 0x01
    - ∗ End Of Text ("eot") ASCII 0x04

- Advantage
  - ∗ The ability of dealing with several computer crash cases, for example
    - · Receiving computer is rebooted:
      - ▷ An "soh" is received first (normal case)
      - ▷ A non "soh" is received first, then neglect whatever it will be received until "eot" or "soh" is received
    - · Receiving computer gets "soh", but not "eot"; instead, it gets another "soh". This means, the sending computer is crashed before sending an "eot" and when it is rebooted, it starts sending again
- Disadvantage
  - ∗ Overhead

soh	block of data in frame	eot

A frame that uses "soh" and "eot" characters to mark the start and end of the frame

Byte Stuffing

- What if the data itself includes "soh" and/or "eot" characters
- In computer network systems
  - The sending side changes the data slightly before sending it
  - The receiving side restores the original data before passing it to the receiving application
- This technique is known as "*byte stuffing*" or "*character stuffing*"; and it is used with character-oriented hardware
- In byte stuffing technique, each
  - "soh" character in data is replaced by "esc" and "x" characters
  - "eot" character in data is replaced by "esc" and "y" characters
  - "esc" character in data is replaced by "esc" and "z" characters
  - Note that, the "soh" and "eot", which are added to the begin and the end of the each frame, will stay as they are without any replacement

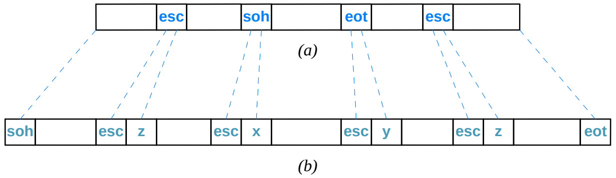


Illustration of byte stuffing
(a) data that include "esc", "soh", and "eot"; (b) A frame after byte stuffing

- Note that
  - The characters "soh" and "eot" do not occur in the data section of a frame; they only appear as a start and end of the frame
  - The "esc" character is always followed by x, y, or z character
  - The stuffed frame is longer than the original data
- There is a similar technique called "bit stuffing", but this technique is used with bit-oriented hardware

Handling Errors

- Data can be corrupted during transmission
  - Bits might be lost
  - Bits might be changed
  - Unsent bits might be appeared
- To detect errors, network systems usually send a small amount of additional information with the data, where
  - The sender computes the value of this additional information
  - The receiver performs the same computation to verify that the packet was transmitted without error
- Error detection techniques are characterized by
  - The size of the additional information (transmission overhead)
  - The computational complexity of the algorithm (computational overhead)
  - The number of detected bit errors (scheme efficiency)

Parity Checking

- A parity scheme adds an extra one additional bit for each n bits, (it usually adds one bit per character)
- There are two forms of parity checking
  - Even parity: Setting the parity bit to 0 or 1 so that the total number of 1's (including the parity bit) is an even number
  - Odd parity: Setting the parity bit to 0 or 1 so that the total number of 1's (including the parity bit) is an odd number
- Examples
  - The even parity bit for 0100100 is 0
  - The even parity bit for 0110100 is 1
  - The odd parity bit for 0100100 is 1
  - The odd parity bit for 0110100 is 0
- The parity checking scheme
  - Adds one bit per character
  - Is easy to calculate
  - Can not detect transmission errors that change an even number of bits

Checksum

- A checksum scheme groups the data into 16-bit segments and treat each of these segments as an integer number
- A checksum is computed as the arithmetic sum of these integers, where if the sum grows larger than 16 bits, the carry bits are added into the final (i.e., 1's complement arithmetic)

H	e	l	l	o		w	o	r	l	d	.
48	65	6C	6C	6F	20	77	6F	72	6C	64	2E

4865 + 6C6C + 6F20 + 776F + 726C + 642E + carry = 71FC

A 16-bit checksum computation for the "Hello world." string

- Note that: The sum + carry = 71FA + 2 = 71FC
- The checksum scheme
  - Adds 16 bits per frame
  - Is easy to calculate
  - Can detect a single bit error and some cases of multiple bit errors, but not all of them;
    For example: 0110 + 0001 = 0100 + 0011 = 0111

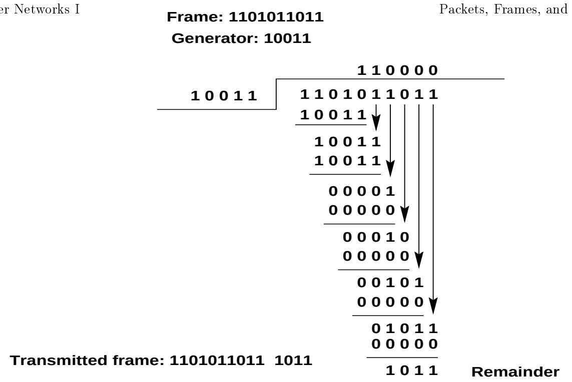
Cyclic Redundancy Checks (CRC)

- Basic Idea
  - Consider the data as a one long number
  - Divide this number by certain dividend
  - Send the remainder as a way of checking
- Example: in base 10
  - Data = 230,000
  - Dividend = 10,941
  - Remainder = 239

- CRC uses modulo 2 binary arithmetic, i.e., binary addition/subtraction with no carries/borrows

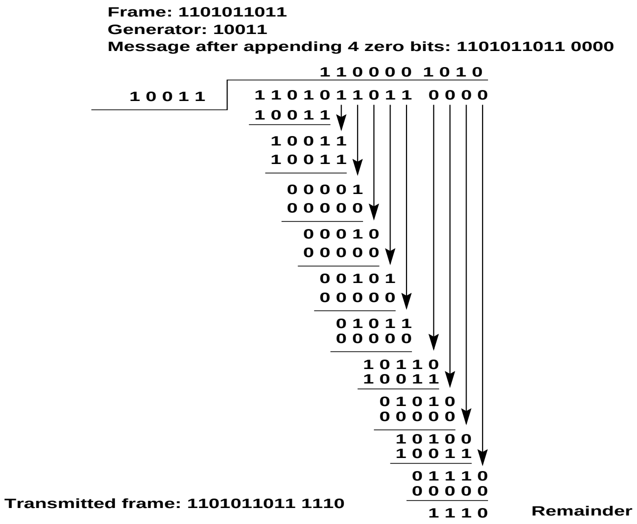
$$\begin{array}{r} 1010 \\ +1111 \\ \hline 0101 \end{array} \qquad \begin{array}{r} 1010 \\ -1111 \\ \hline 0101 \end{array}$$

- Addition and subtraction in modulo 2 binary arithmetic are identical to Exclusive OR (XOR)
- The dividend is called generator
- The first and last bits in the generator MUST BE 1
- If the generator is n bits, the generated CRC code is $n-1$ bits



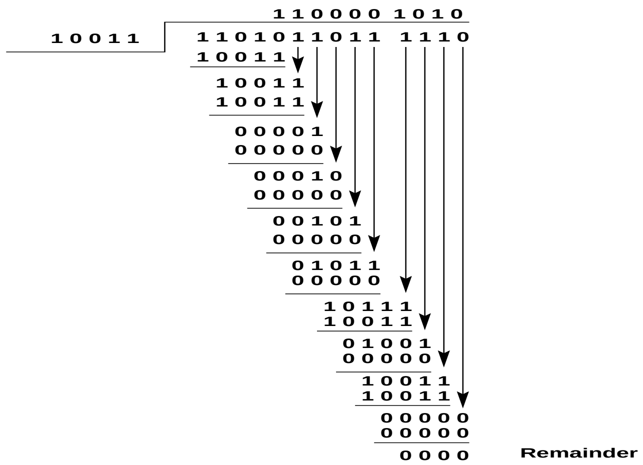
Generating/checking the CRC

- The receiver, as well, must perform the above calculation until getting the remainder, then checking it against the transmitted remainder
- In modulo 2 binary arithmetic, adding $n-1$ zero-bits to the right side of the frame during generating the CRC, means, mathematically, that if no error happened during transmission, the CRC checking process will produce zeros as a remainder



Generating the CRC

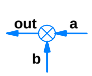
Frame: 1101011011 1110
Generator: 10011



Checking the CRC

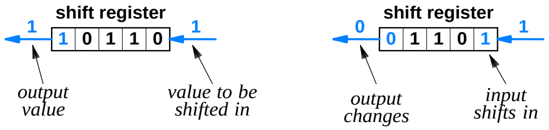
CRC Hardware Implementation

- CRC can be implemented using only XORs and shift registers



a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

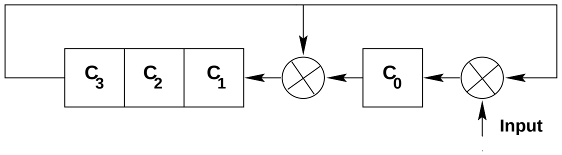
Exclusive OR (XOR)



Shift register

CRC Hardware Implementation Procedure

- Number of used shift registers = number of 1's in the generator − 1
- The length of each shift register = number of bits between consecutive ones in the generator + 1
- Number of XORs is equal to the number of shift registers
- An XOR is added before each shift register (i.e., to the right of each shift register)
- The output of each XOR is connected to the input of the shift register in front of it (i.e., to the left of each XOR)
- The last output of the left-most shift register is feedback to all XORs
- The input is fed to the right-most XOR.



Initial Value	C_3	C_2	C_1	C_0	input	C_3 XOR C_0	C_3 XOR input
Step 1	0	0	0	0	1	0	1
Step 2	0	0	0	1	1	1	1
Step 3	0	0	1	1	0	1	0
Step 4	0	1	1	0	1	0	1
Step 5	1	1	0	1	0	0	1
Step 6	1	0	0	1	1	0	0
Step 7	0	0	0	0	1	0	1
Step 8	0	0	0	1	0	1	0
Step 9	0	0	1	0	1	0	1
Step 10	0	1	0	1	1	1	1
Step 11	1	0	1	1	-		

Generating/checking CRC using shift registers and XORs
Generator = 10011

	C_3	C_2	C_1	C_0	input	C_3 XOR C_0	C_3 XOR input
Initial Value	0	0	0	0	1	0	1
Step 1	0	0	0	1	1	1	1
Step 2	0	0	1	1	0	1	0
Step 3	0	1	1	0	1	0	1
Step 4	1	1	0	1	0	0	1
Step 5	1	0	0	1	1	0	0
Step 6	0	0	0	0	1	0	1
Step 7	0	0	0	1	0	1	0
Step 8	0	0	1	0	1	0	1
Step 9	0	1	0	1	1	1	1
Step 10	1	0	1	1	0	0	1
Step 11	0	1	0	1	0	1	0
Step 12	1	0	1	0	0	1	1
Step 13	0	1	1	1	0	1	0
Step 14	1	1	1	0	-		

Generating CRC using shift registers and XORs (4 zeros were added to the input)
Generator = 10011

	C_3	C_2	C_1	C_0	input	C_3 XOR C_0	C_3 XOR input
Initial Value	0	0	0	0	1	0	1
Step 1	0	0	0	1	1	1	1
Step 2	0	0	1	1	0	1	0
Step 3	0	1	1	0	1	0	1
Step 4	1	1	0	1	0	0	1
Step 5	1	0	0	1	1	0	0
Step 6	0	0	0	0	1	0	1
Step 7	0	0	0	1	0	1	0
Step 8	0	0	1	0	1	0	1
Step 9	0	1	0	1	1	1	1
Step 10	1	0	1	1	1	0	0
Step 11	0	1	0	0	1	0	1
Step 12	1	0	0	1	1	0	0
Step 13	0	0	0	0	0	0	0
Step 14	0	0	0	0	-		

Checking CRC using shift registers and XORs
Generator = 10011

- The CRC scheme
  - Adds $n-1$ bits per frame, where n is the length of the generator
  - Is more complex than checksum and parity checking
  - Detects more errors than checksum and parity checking

Widely Used Generators

- A generator can be expressed as a polynomial of X, for example 10011 can be expressed as $X^4 + X + 1$
- The most widely used 17-bit generators are:
  - CRC-16: $X^{16} + X^{15} + X^2 + 1$, i.e., 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
  - CRC-CCITT: $X^{16} + X^{12} + X^5 + 1$, i.e., 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1

Frame format and Error Detection Mechanism

- Error detection information is associated with each frame

A frame format which include the CRC value

- Do we need byte stuffing for the CRC value?
  Without byte stuffing
  - If the CRC value damaged, but not lost, no problem
  - If one or two bytes from the CRC value are lost, the next frame will be rejected
- CRC can be applied before or after byte stuffing, as long as both sender and receiver agree upon it