

Lecture 4: Reliable Transmission

CSE 123: Computer Networks
Alex C. Snoeren



Project 1a due one week from today



Lecture 4 Overview



- Finishing Error Detection
 - ◆ Checksums
 - ◆ Cyclic Remainder Check (CRC)
- Handling errors
 - ◆ Automatic Repeat Request (ARQ)
 - ◆ Acknowledgements (ACKs) and timeouts
 - ◆ Stop-and-Wait

Checksums



- Simply sum up all of the data in the frame
 - ♦ Transmit that sum as the EDC
- Extremely lightweight
- Also easy to modify if frame is modified in flight
 - ♦ Happens a lot to packets on the Internet
- IP packets include a 1's complement checksum

What is the Hamming Distance of a checksum-based code?

- A. 1**
- B. 2**
- C. It depends**
- D. I don't know**



IP Checksum Example

- 1's complement of sum of 16-bit *words* (not bytes)

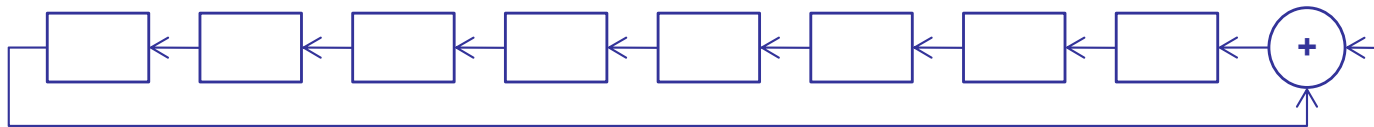
```
u_short cksum(u_short *buf, int count) {  
    register u_long sum = 0;  
    while (count-->0) {  
        sum += *buf++;  
        if (sum & 0xFFFF0000) {  
            /* carry occurred, so wrap around */  
            sum &= 0xFFFF;  
            sum++;  
        }  
    }  
    return ~(sum & 0xFFFF);  
}
```

**What is the IP
checksum of an all-
zero packet?**

- A. 0x0000**
- B. 0xFFFF**
- C. 0x1111**
- D. I don't know**



Checksum in Hardware



- Compute checksum in Modulo-2 Arithmetic
 - ♦ Addition/subtraction is simply XOR operation
 - ♦ Equivalent to vertical parity computation
- Need only a word-length shift register and XOR gate
 - ♦ Assuming data arrives serially
 - ♦ All registers are initially 0

Modulo-2 Arithmetic



- Addition & subtraction are XOR
 - ◆ $1 + 1 = 0$; $0 - 1 = 1$ (no carries!)

- Multiplication

```
  1101
   110
-----
  0000
 11010
110100
-----
101110
```

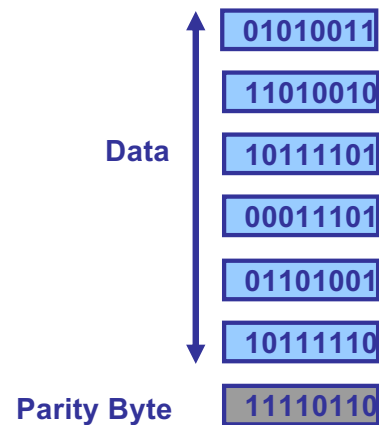
- Division

```
      1101
110 ) 101110
     110
     ---
      111
      110
      ---
       011
       000
       ---
        110
```

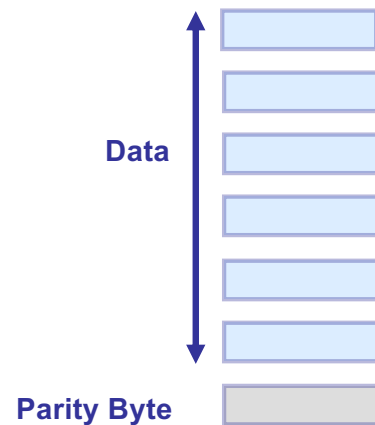
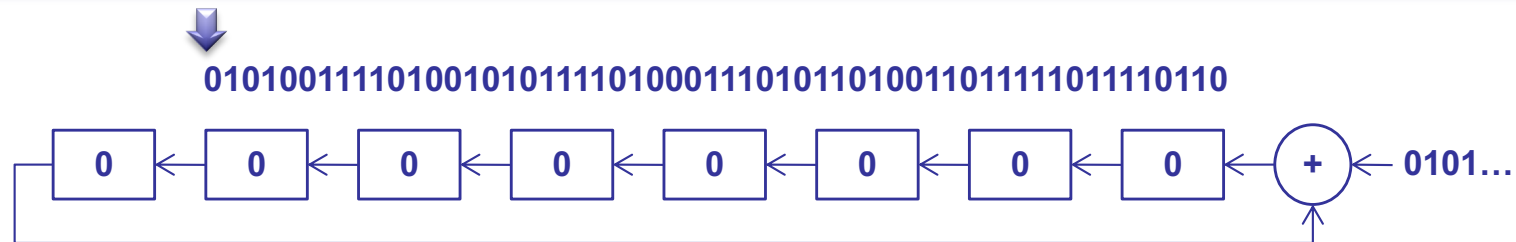
Checksum Example



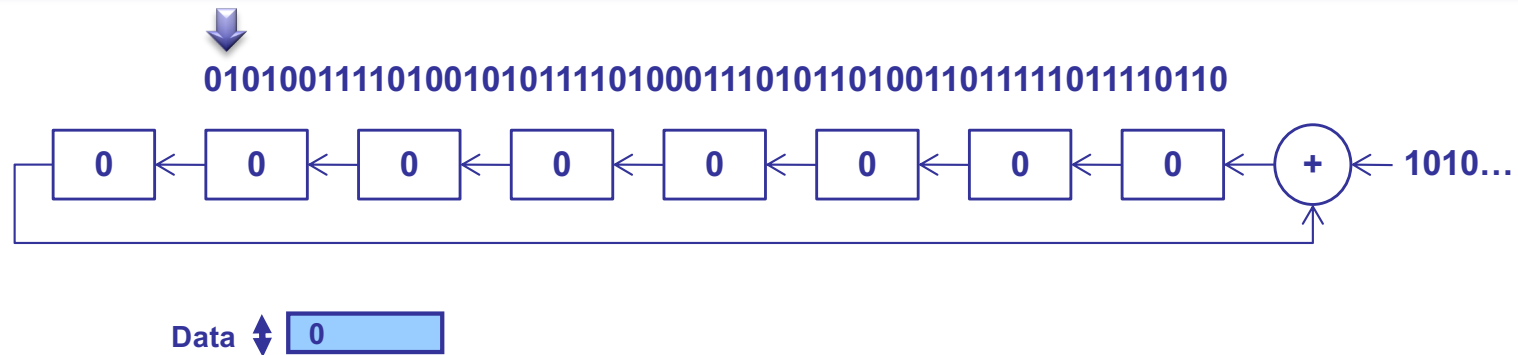
01010011110100101011110100011101011010011011111011110110



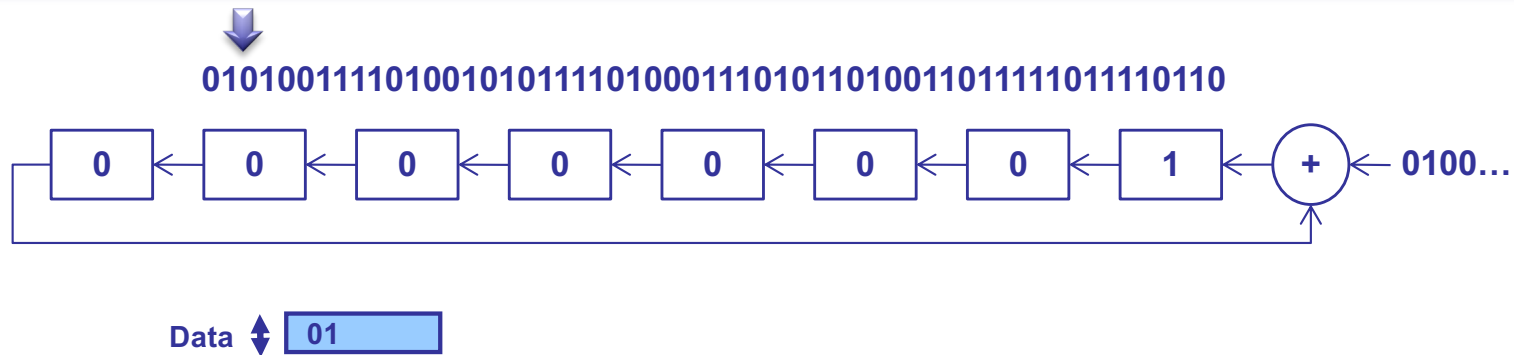
Checksum Example



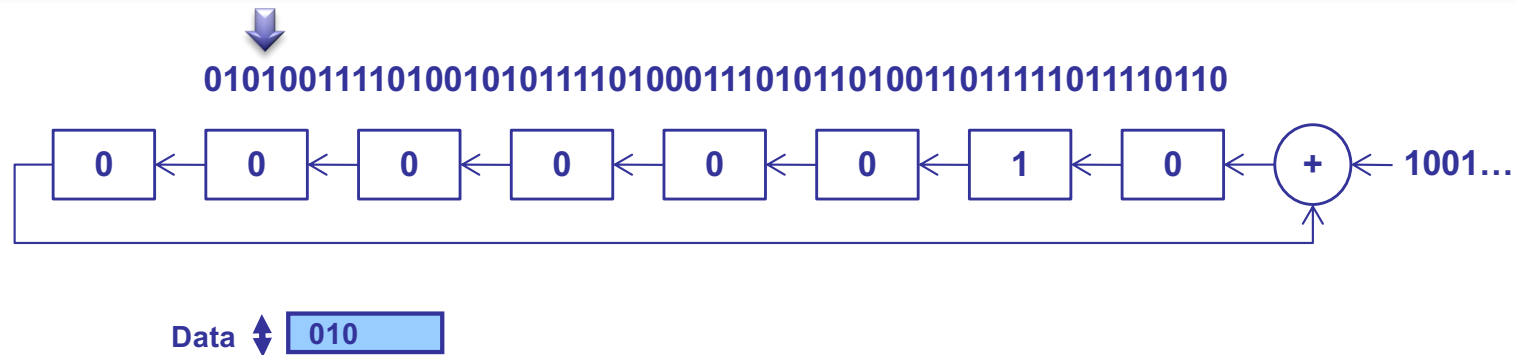
Checksum Example



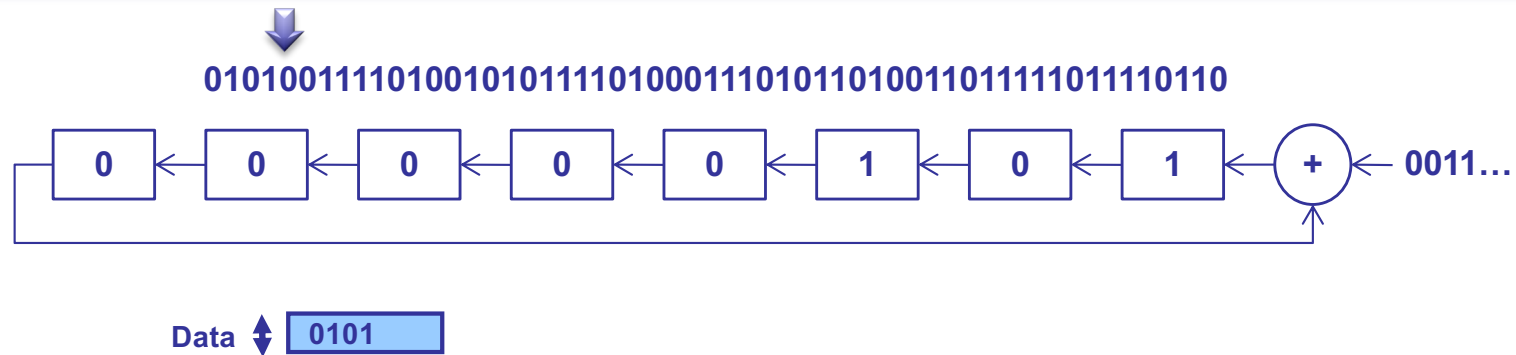
Checksum Example



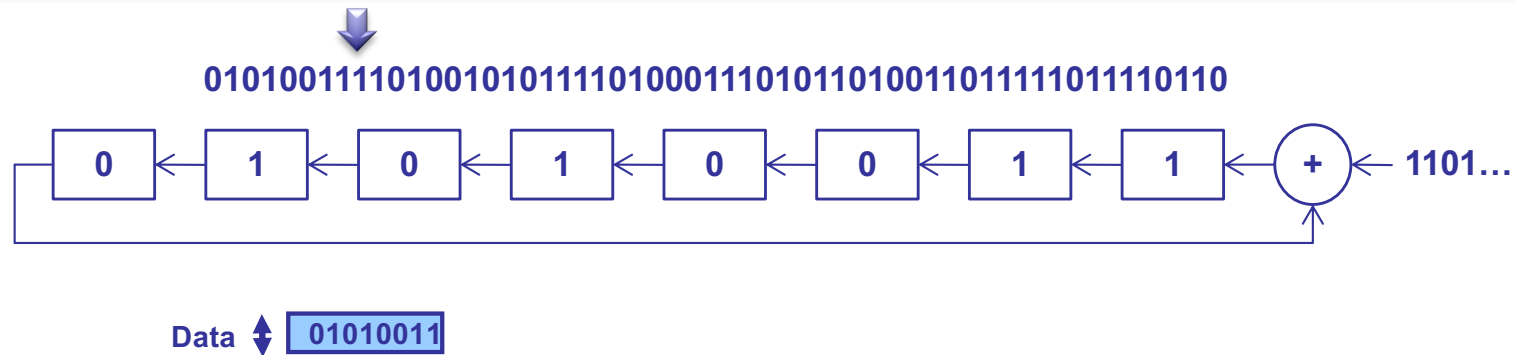
Checksum Example



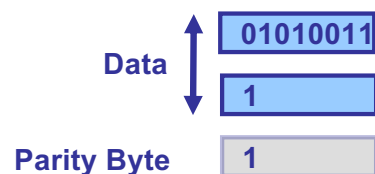
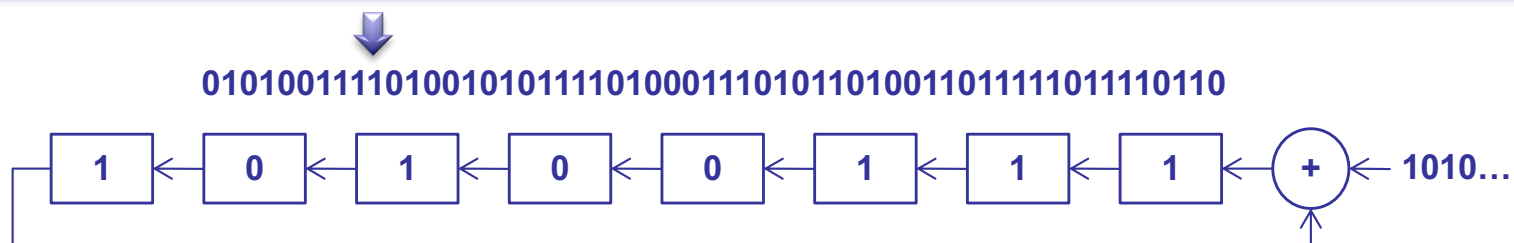
Checksum Example



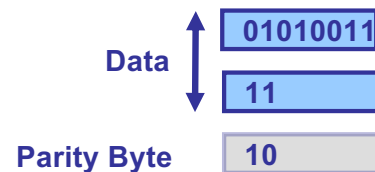
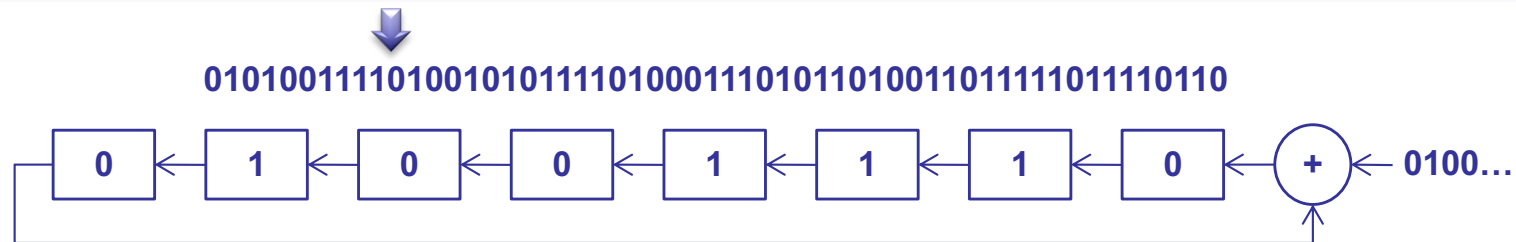
Checksum Example



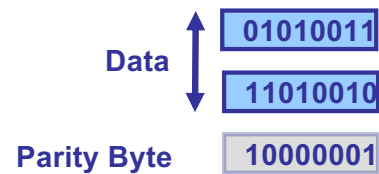
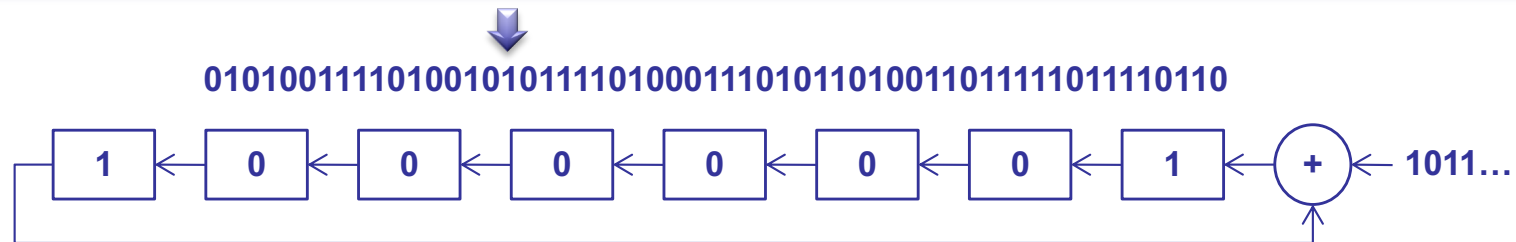
Checksum Example



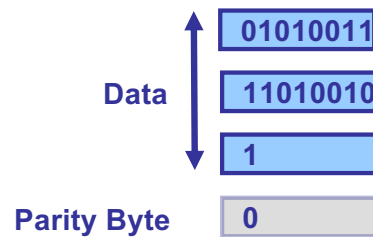
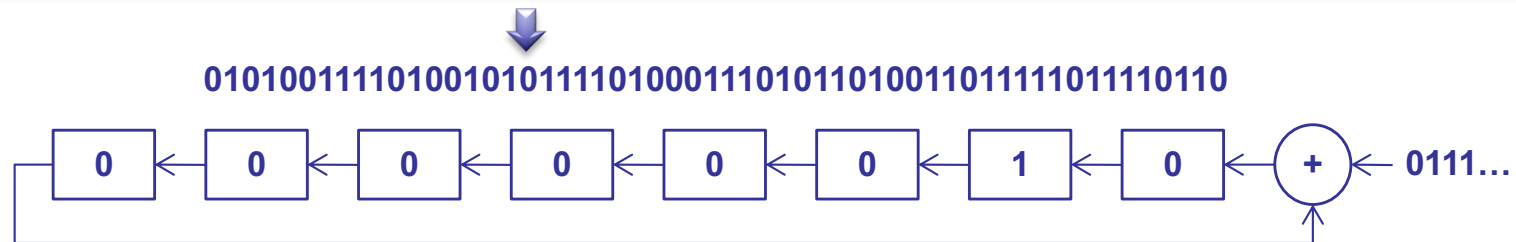
Checksum Example



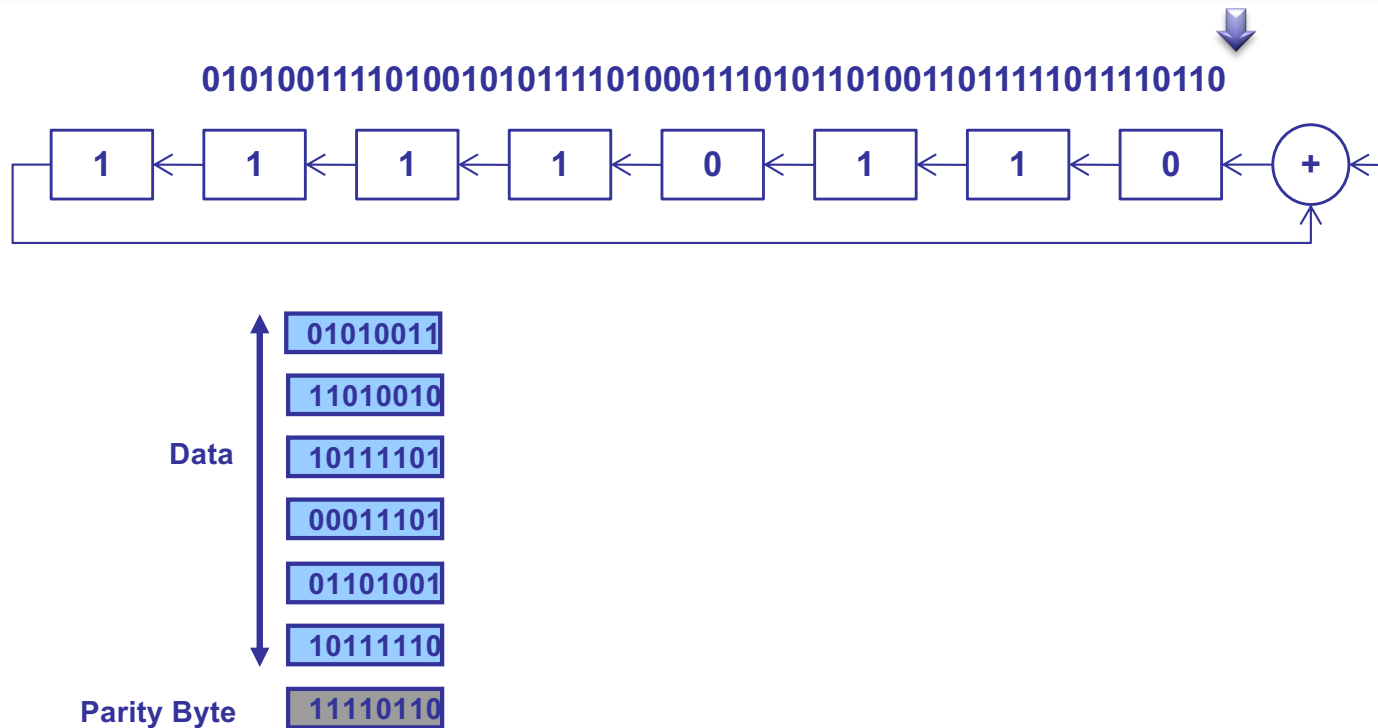
Checksum Example



Checksum Example



Checksum Example





From Sums to Remainders

- Checksums are easy to compute, but very fragile
 - ◆ In particular, **burst** errors are frequently undetected
 - ◆ We'd rather have a scheme that "smears" parity
- Need to remain easy to implement in hardware
 - ◆ All we need are shift registers and an XOR gate
- We'll stick to Modulo-2 arithmetic
 - ◆ Multiplication and division are XOR-based as well
 - ◆ Let's do some examples...



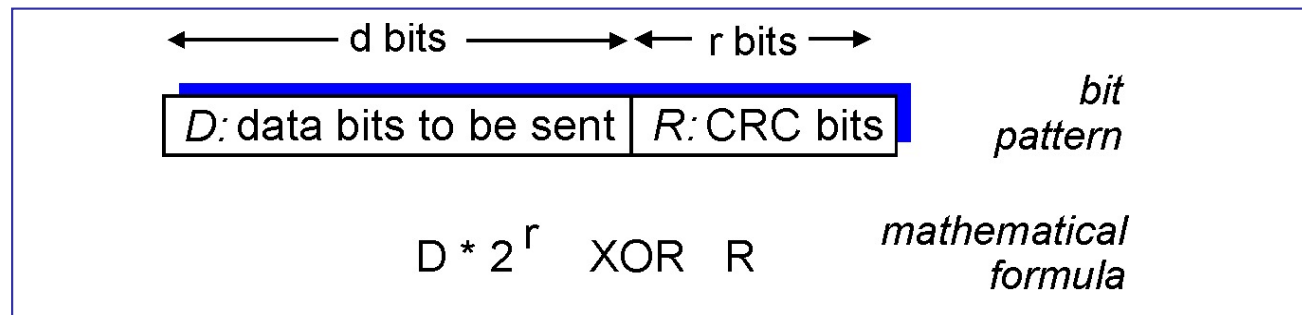
Cyclic Remainder Check

- Idea is to *divide* the incoming data, D , rather than add
 - ♦ The divisor is called the **generator**, g
- We can make a CRC resilient to k -bit burst errors
 - ♦ Need a generator of $k+1$ bits
- Divide $2^k D$ by g to get remainder, r
 - ♦ Remainder is called **frame check sequence**
- Send $2^k D - r$ (i.e., $2^k D \text{ XOR } r$)
 - ♦ Note $2^k D$ is just D shifted left k bits
 - ♦ Remainder must be at most k bits
- Receiver checks that $(2^k D - r)/g = 0$



Error Detection – CRC

- View data bits, **D**, as a binary number
- Choose $r+1$ bit pattern (generator), **G**
- Goal: choose r CRC bits, **R**, such that
 - ♦ $\langle D \mid R \rangle$ exactly divisible by G (modulo 2)
 - ♦ Receiver knows G , divides $\langle D \mid R \rangle$ by G . If non-zero remainder: error detected!
 - ♦ Can detect all burst errors less than $r+1$ bits
- Widely used in practice (Ethernet, FDDI, ATM)





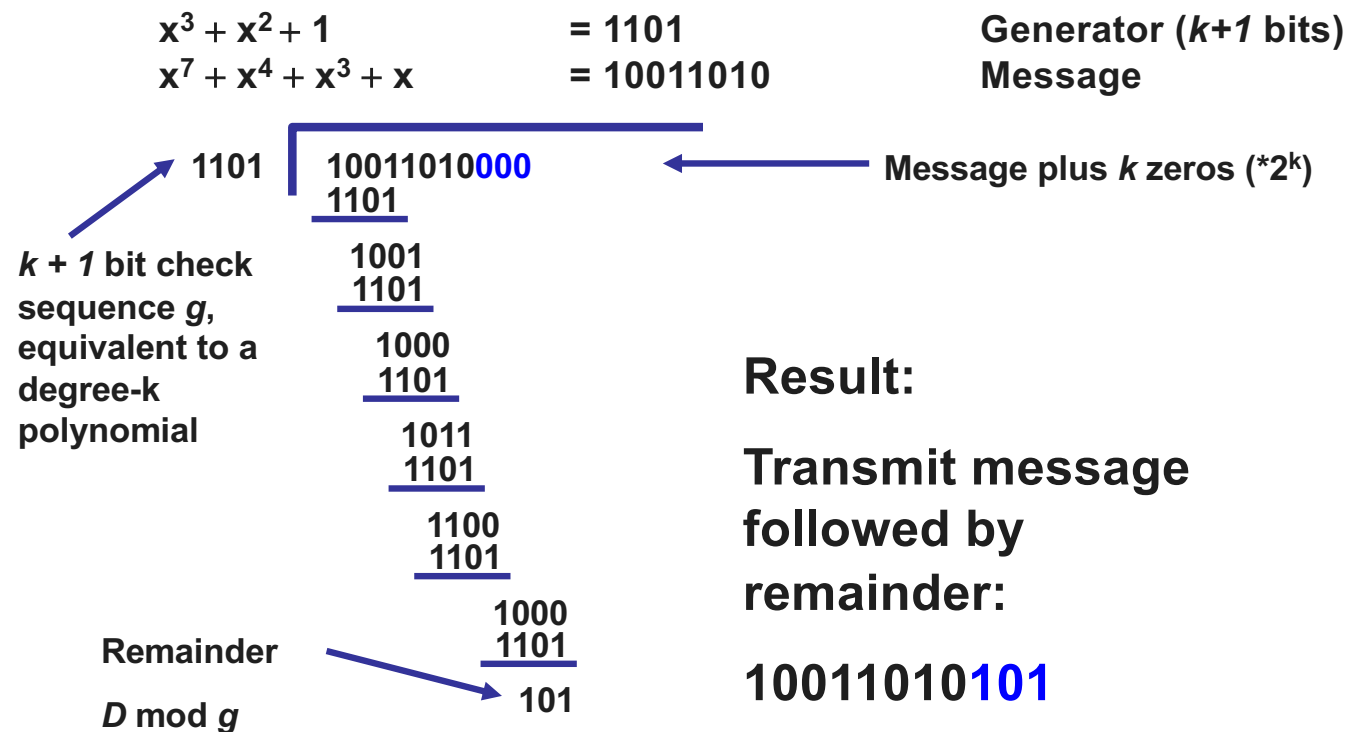
CRC: Rooted in Polynomials

- We're *actually* doing polynomial arithmetic
 - ♦ Each bit is actually a coefficient of corresponding term in a k^{th} -degree polynomial

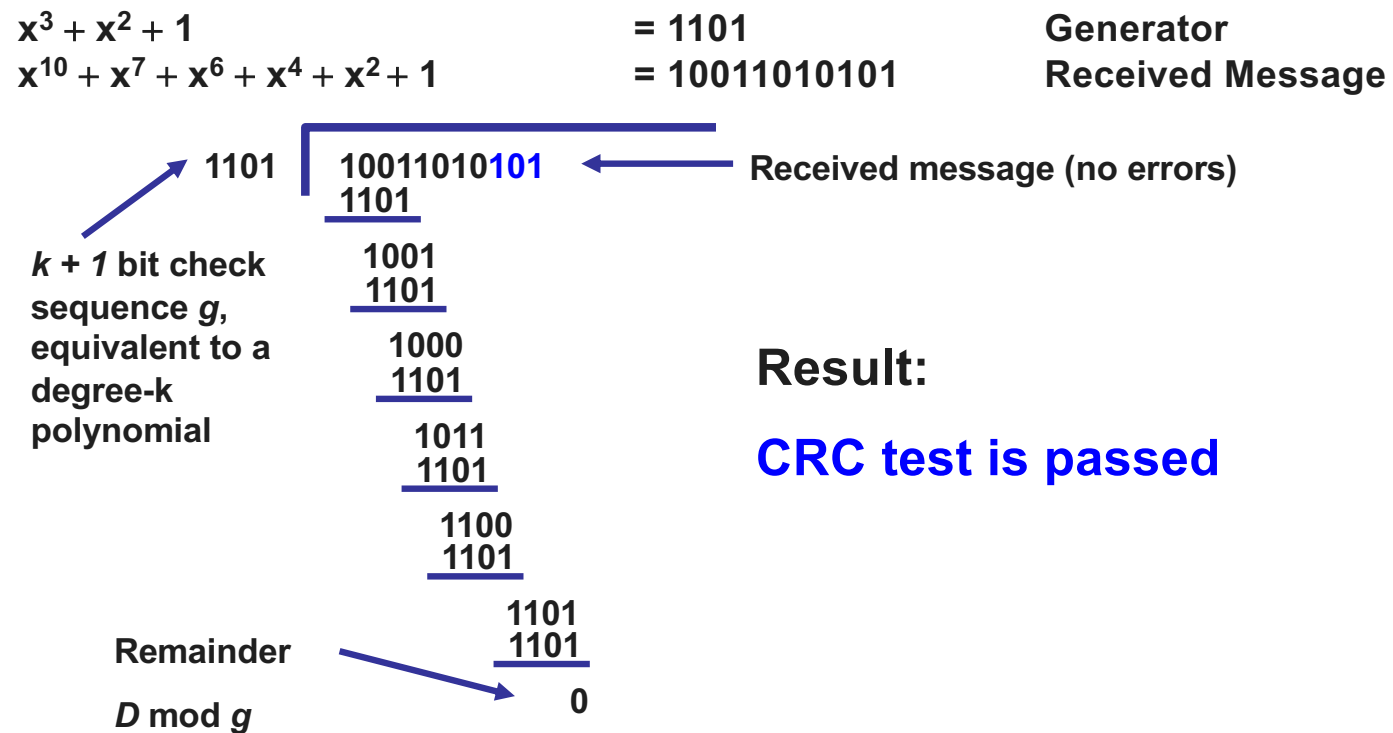
$$1101 \text{ is } (1 * X^3) + (1 * X^2) + (0 * X^1) + (1 * X^0)$$

- Why do we care?
 - ♦ Can use the properties of finite fields to analyze effectiveness
 - ♦ E.g., says any generator with two terms catches single bit errors

CRC Example Encoding



CRC Example Decoding





CRC Example Failure

$x^3 + x^2 + 1$
 $x^{10} + x^7 + x^5 + x^4 + x^2 + 1$

$= 1101$
 $= 10010110101$

Generator
Received Message

Will this be caught?

A. Yes, because 2 bits is less than four
B. Yes, because CRC always works
C. No, CRC can't catch even numbers of bit errors
D. No, the errors are in the message, not the remainder

$k + 1$ bit check sequence g , equivalent to a degree- k polynomial

Remainder $D \bmod g$

Received message

Two bit errors

Result:
CRC test failed

1101

10010110101

1101

1000

1101

1011

1101

1101

1101

0101

Common Generators



CRC-8	$x^8 + x^2 + x^1 + 1$
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-CCITT	$x^{16} + x^{12} + x^5 + 1$
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$



Error Handling Summary

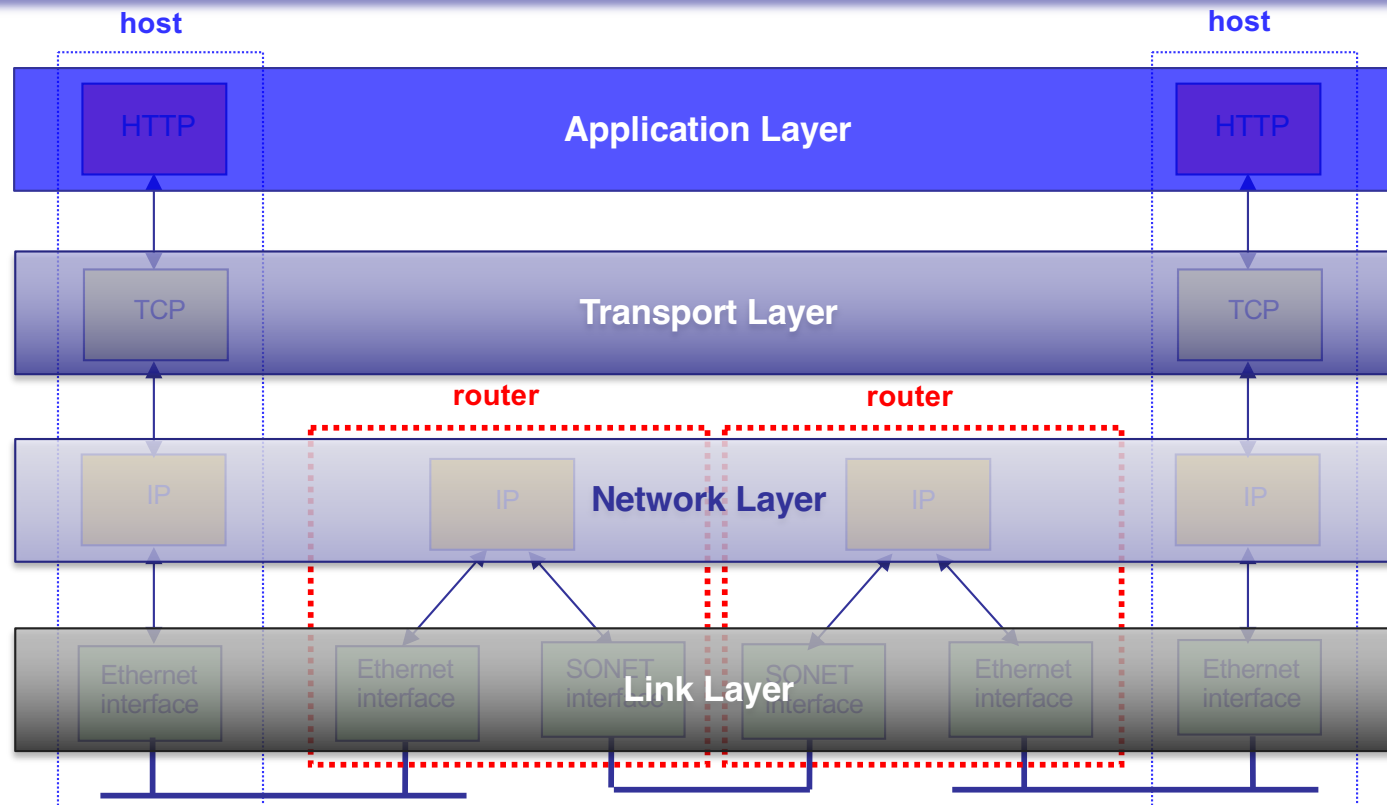
- Add redundant bits to detect if frame has errors
 - ◆ A few bits can detect errors
 - ◆ Need more to *correct* errors
- Strength of code depends on Hamming Distance
 - ◆ Number of bitflips between codewords
- Checksums and CRCs are typical methods
 - ◆ Both cheap and easy to implement in hardware
 - ◆ CRC much more robust against burst errors



Picking up the Pieces

- Link layer is lossy
 - ♦ We deliberately throw away corrupt frames!
 - ♦ Infrequent bit errors still lead to occasional frame errors
 - » 10,000+ bits in each frame
- Things get even harrier if we consider multiple links
 - ♦ In a few lectures, we'll start sending frames on long trips
 - ♦ Each intermediate stop might lose, corrupt, *reorder*, etc.
 - ♦ Regardless of cause, we'll call loss events **drops**
- We want to provide reliable, in-order delivery
 - ♦ Can—and will—do this at multiple layers

Moving up the Stack

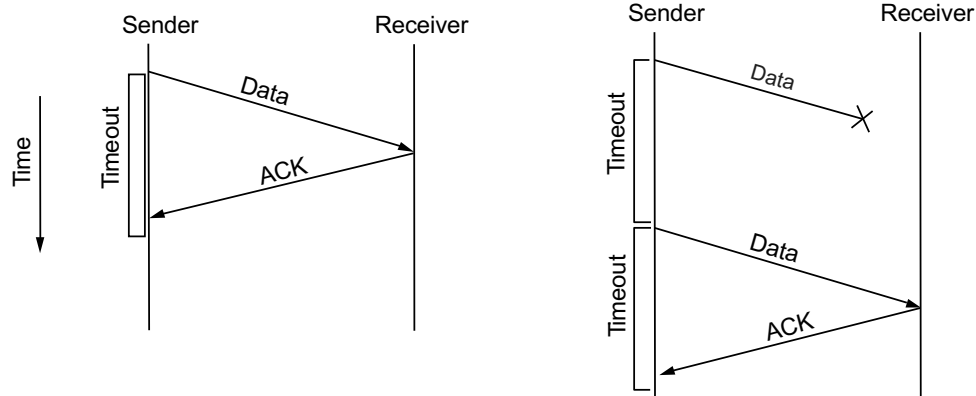


Reliable Transmission



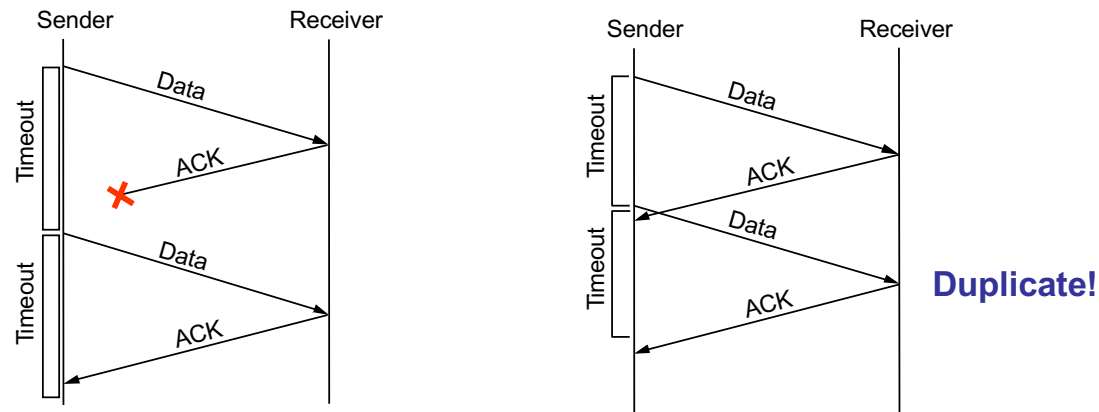
- The data networking version of the same problem
 - ◆ How do we reliably send a message when packets can be lost/corrupted in the network?
- Two options
 - ◆ **Detect a loss/corruption and retransmit**
 - ◆ Send data redundantly to tolerate loss/corruption

Simple Idea: ARQ



- Receiver sends **acknowledgments** (ACKs)
 - ◆ Sender “times out” and retransmits if it doesn’t receive them
- Basic approach is generically referred to as **Automatic Repeat Request** (ARQ)

Not So Fast...



- Loss can occur on ACK channel as well
 - ◆ Sender cannot distinguish data loss from ACK loss
 - ◆ Sender will retransmit the data frame
- ACK loss—or early timeout—results in duplication
 - ◆ The receiver thinks the retransmission is new data

For Next Time



- Project 1a due one week from today: 1/17
- HW1 due on Friday
- Read 2.5 in P&D