

Lecture 23 & 24: Error Handling

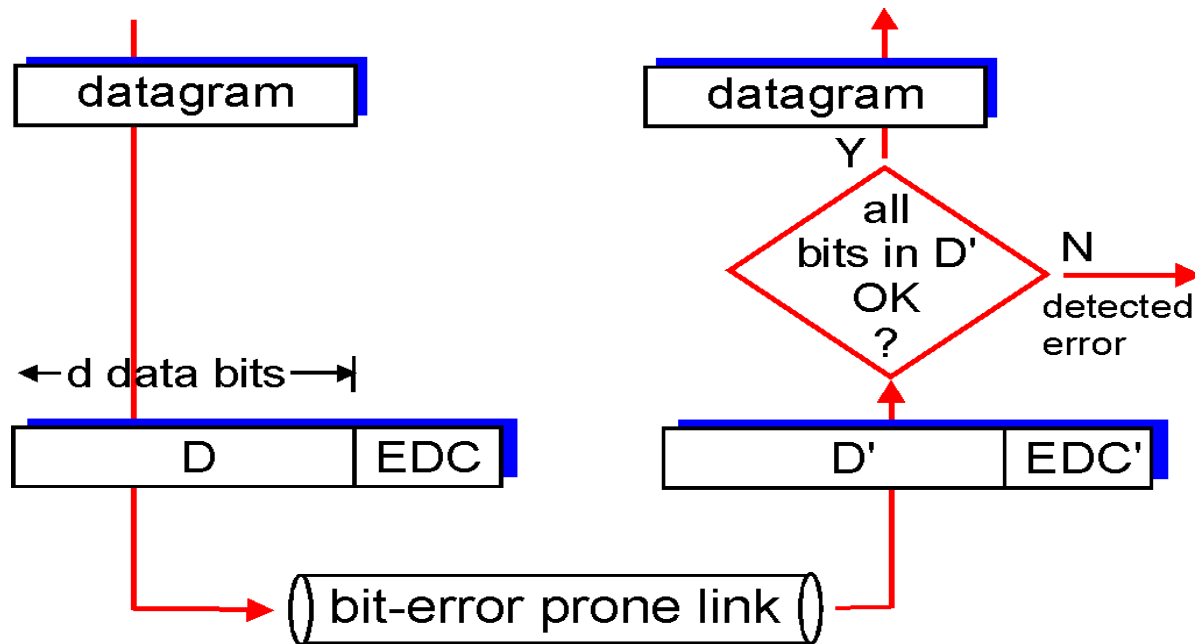
CSE 123: Computer Networks





Error Detection

- Implemented at many layers (end-to-end argument)
 - ◆ We'll mainly focus on link-layer techniques today





Basic Idea

- The problem is headers/payload is not self-verifying
 - ◆ Every string of bits is potentially legitimate
 - ◆ Hence, any errors/changes in a set of bits are equally legit
- The solution is to reduce the set of potential bitstrings
 - ◆ Not every string of bits is allowable
 - ◆ Receipt of a disallowed string of bits means the original bits were garbled in transit
- Key question: which bitstrings are allowed?

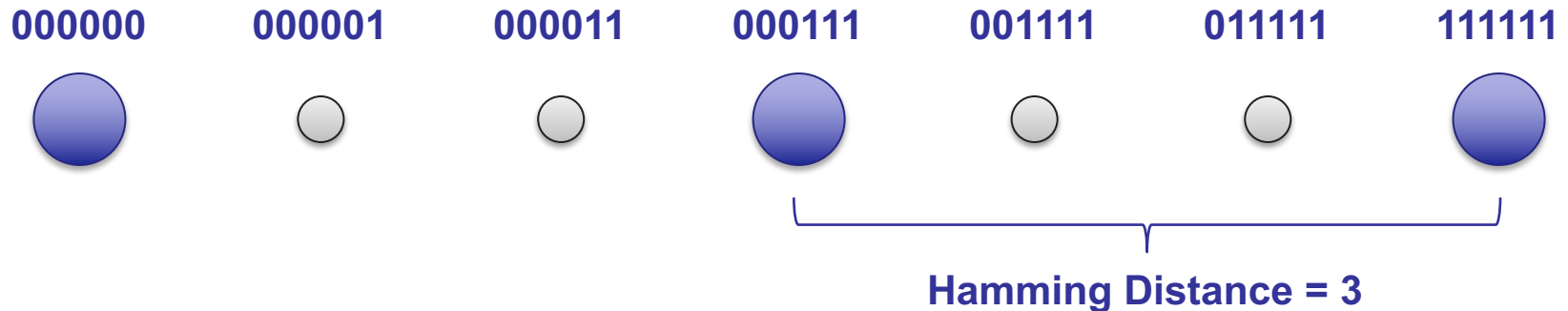


Codewords

- Let's start simple, and consider fixed-length bitstrings
 - ◆ Reduce our discussion to n -bit substrings
 - ◆ E.g., 7-bits at a time, or 4 bits at a time (4B/5B)
 - ◆ Or even a frame at a time
- We call an allowable sequence of n bits a **codeword**
 - ◆ Not all strings of n bits are codewords!
 - ◆ The remaining n -bit strings are “space” between codewords
 - ◆ Codewords represent shorter $< n$ bit sequences (add redundancy)
- What bit strings are allowed? → How many codewords with how much space between them?



Hamming Distance

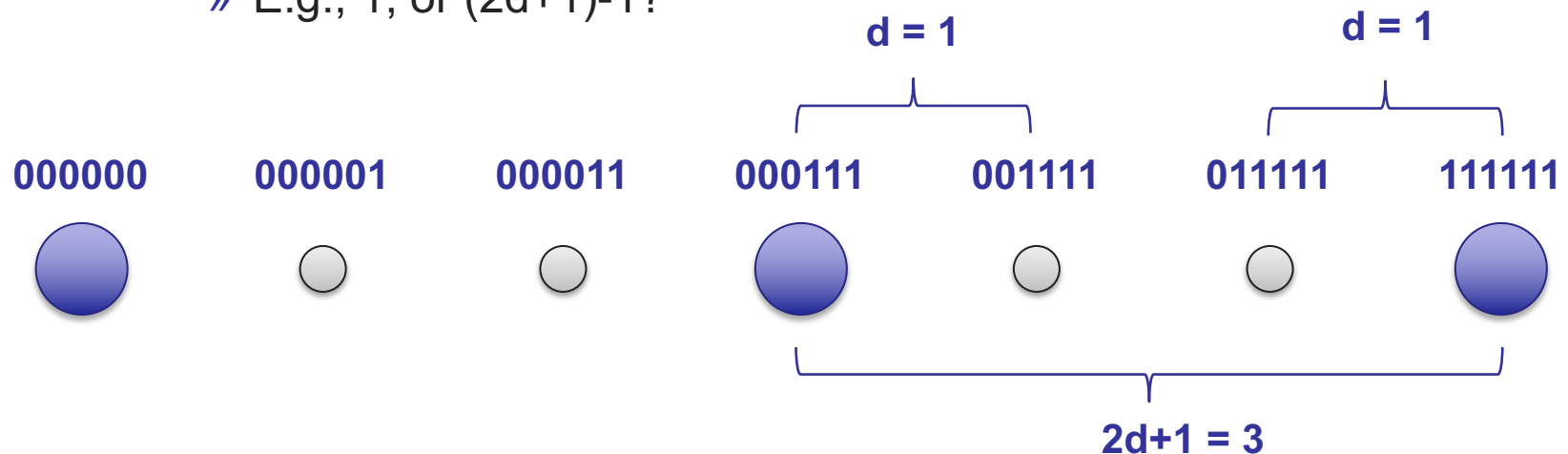


- Distance between legal codewords
 - ◆ Measured in terms of number of bit flips
- **Efficient** codes are of uniform Hamming Distance
 - ◆ All codewords are equidistant from their neighbors



$2d+1$ Hamming Distance

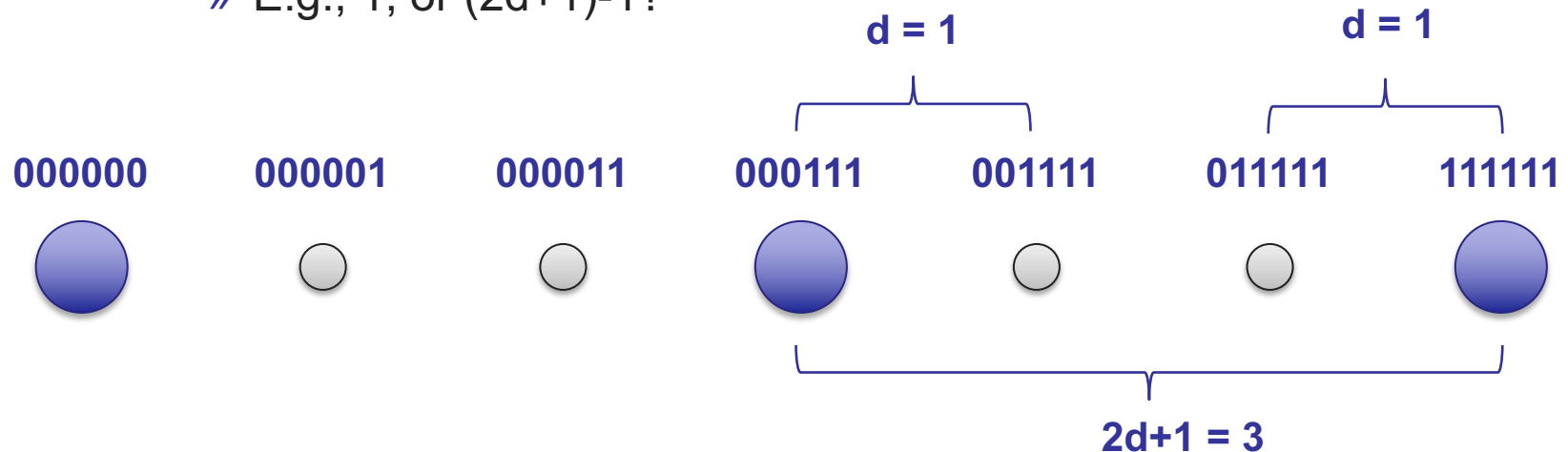
- Can **detect** up to $2d$ bit flips
 - ◆ The next codeword is always $2d+1$ bit flips away
 - ◆ Any fewer is guaranteed to land in the middle
- Can **correct** up to d bit flips
 - ◆ We just move to the closest codeword
 - ◆ Unfortunately, no way to tell how many bit flips
 - » E.g., 1, or $(2d+1)-1$?





$2d+1$ Hamming Distance

- Can **detect** up to $2d$ bit flips
 - ◆ The next codeword is always $2d+1$ bit flips away
 - ◆ Any fewer is guaranteed to land in the middle
- Can **correct** up to d bit flips
 - ◆ We just move to the closest codeword
 - ◆ Unfortunately, no way to tell how many bit flips
 - » E.g., 1, or $(2d+1)-1$?

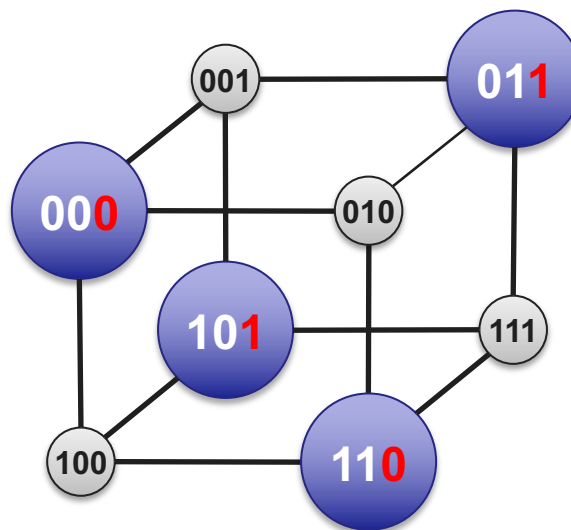
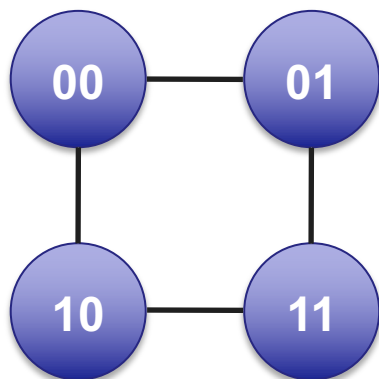


Encoding



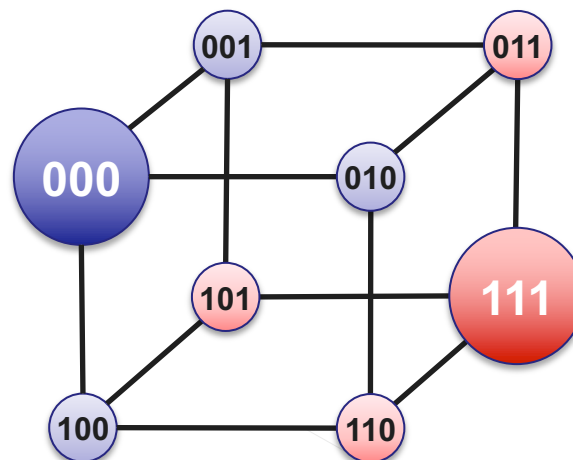
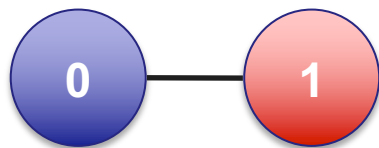
- We're going to send only codewords
 - ◆ Non-codewords indicate errors to receiver
- But we *want* to send any set of strings of bits
 - ◆ Need to embed arbitrary input into sequence of codewords
 - ◆ Adding redundancy: input represented as codewords is longer

Simple Embedding: Parity



- Essentially Code with Hamming Distance 2
 - ◆ Can detect one bit flip (no correction capability)
- Add extra bit to ensure odd(even) number of ones
 - ◆ Code is 66% efficient (need three bits to encode two)
 - ◆ Note: Even parity is simply XOR

Simple Correction: Voting



- Simply send each bit n (3 in this example) times
 - ◆ Code with Hamming Distance 3 ($d=1$)
 - ◆ Can detect 2 bit flips and correct 1
- Straightforward duplication is extremely inefficient
 - ◆ We can be much smarter about this

Per-Frame Detection Codes



- Want to add an error detection code per frame
 - ◆ Frame is unit of transmission; all or nothing.
 - ◆ Computed over the entire frame—including header! Why?
- Receiver checks EDC to make sure frame is valid
 - ◆ If frame fails check, throw it away
- We *could* use error-correcting codes
 - ◆ But they are less efficient, and *we expect errors to be rare*
 - ◆ Counter example: wireless communication



Two-Dimensional Parity

		Parity Bits
Data	0101001	1
	1101001	0
	1011110	1
	0001110	1
	0110100	1
	1011111	0
Parity Byte	1111011	0

- Start with normal parity
 - ◆ n data bits, 1 one parity bit
- Do the same across rows
 - ◆ m data bytes, 1 parity byte
- Can detect up to 3 bit errors
 - ◆ Even most 4-bit errors
- Can correct any 1 bit error
 - ◆ Why?



Checksums

- Simply sum up all of the data in the frame
 - ◆ Transmit that sum as the EDC
- Extremely lightweight
 - ◆ Easy to compute fast in hardware
 - ◆ Fragile: Hamming Distance of 2
 - So can only detect how many bit errors (poll)?
- Also easy to modify if frame is modified in flight
 - ◆ Happens a lot to packets on the Internet
- IP packets include a 1's compliment checksum

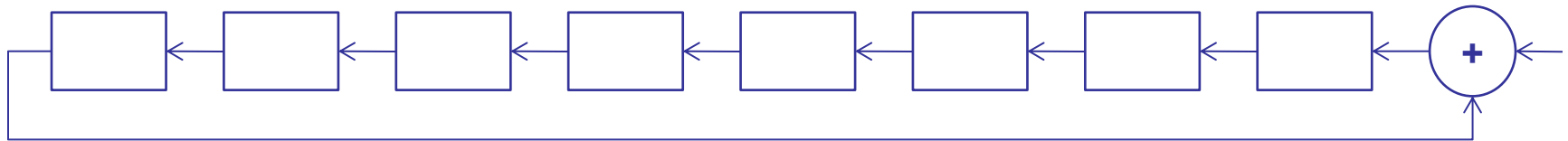
IP 16bit Checksum Example



- 1's compliment of sum of *words* (not bytes)
 - ◆ Final 1's compliment means all-zero frame is not valid

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

Checksum in Hardware

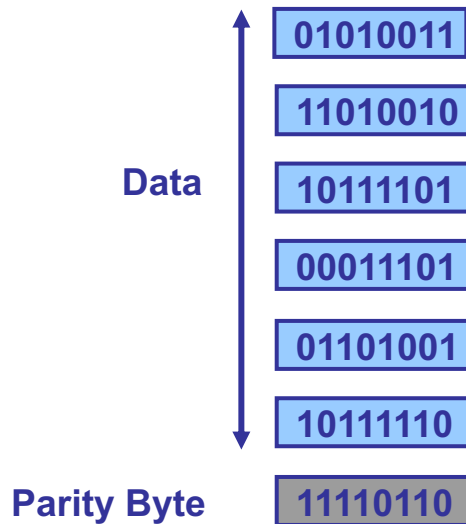


- Compute checksum in Modulo-2 Arithmetic
 - ◆ Addition/subtraction is simply XOR operation
 - ◆ **Equivalent to vertical parity computation**
- Need only a word-length shift register and XOR gate
 - ◆ Assuming data arrives serially
 - ◆ All registers are initially 0



Checksum Example

01010011110100101011110100011101011010011011111011110110



Checksum Example



01010011110100101011110100011101011010011011111011110110



0101...

+

0

0

0

0

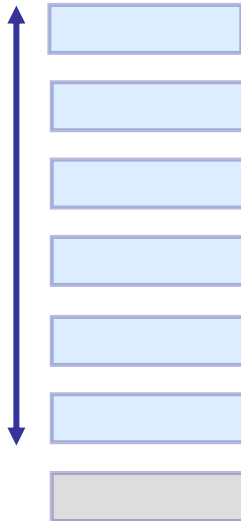
0

0

0

0

Data



Parity Byte

Checksum Example



01010011110100101011110100011101011010011011111011110110



1010...

+

0

0

0

0

0

0

0

0

Data



0

Checksum Example



01010011110100101011110100011101011010011011111011110110



0100...

+

1

0

0

0

0

0

0

0

Data



01

Checksum Example



01010011110100101011110100011101011010011011111011110110



1001...

+

1

0

0

0

0

0

0

0

Data



010

Checksum Example



01010011110100101011110100011101011010011011111011110110



0011...

+

1

0

1

0

0

0

0

0

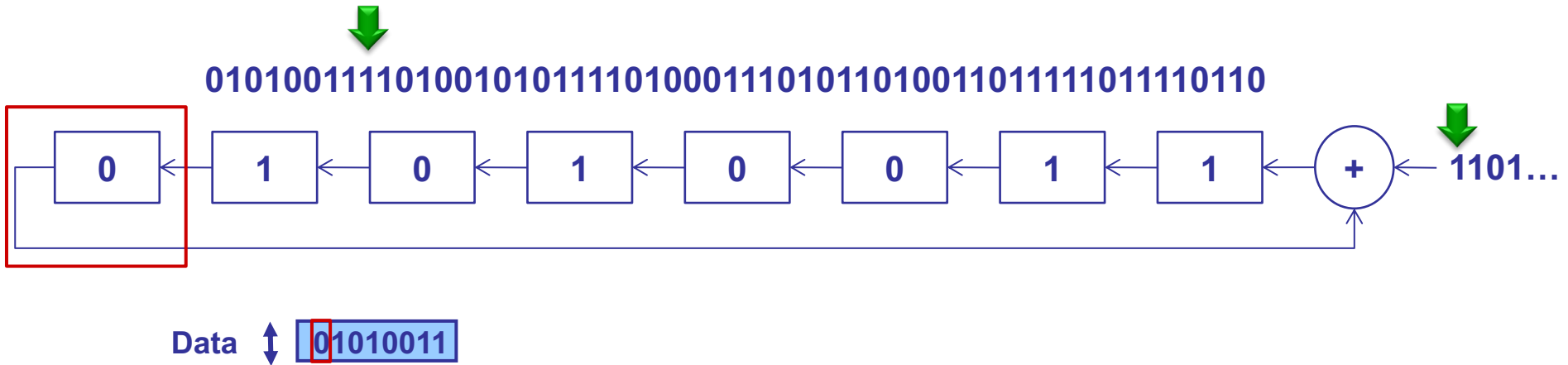
Data



0101

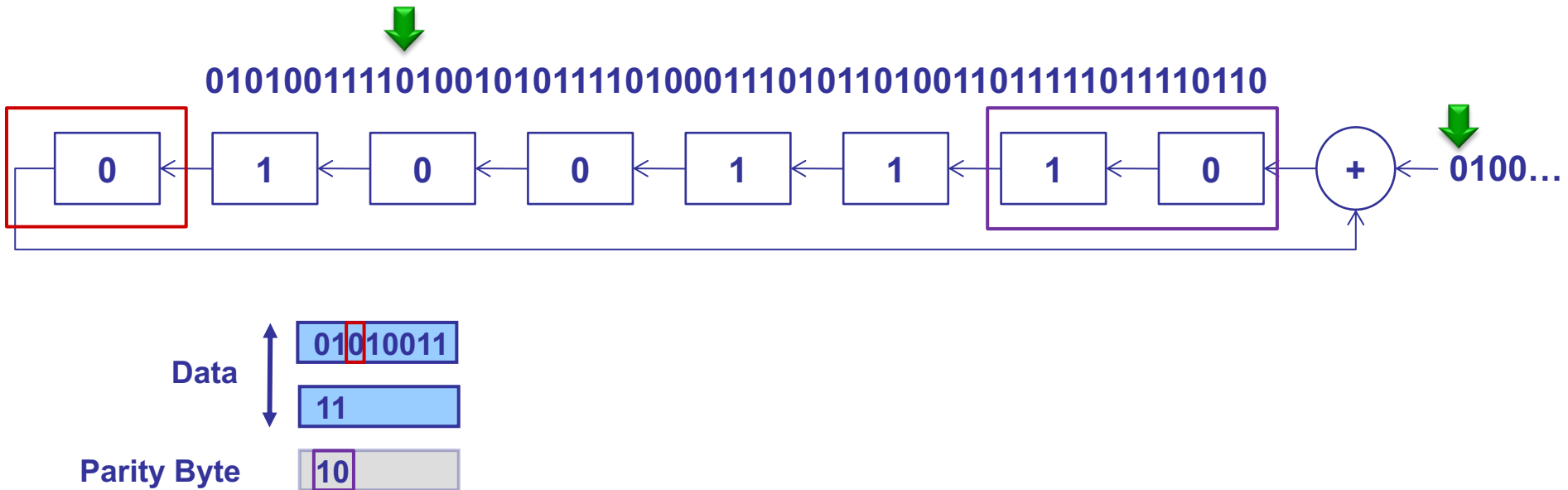


Checksum Example





Checksum Example



Checksum Example



01010011110100101011110100011101011010011011111011110110



1011...

+

1

0

0

0

0

0

0

1

Data



01010011

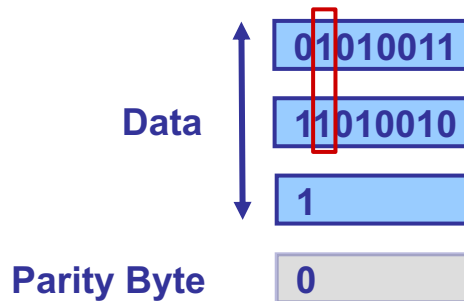
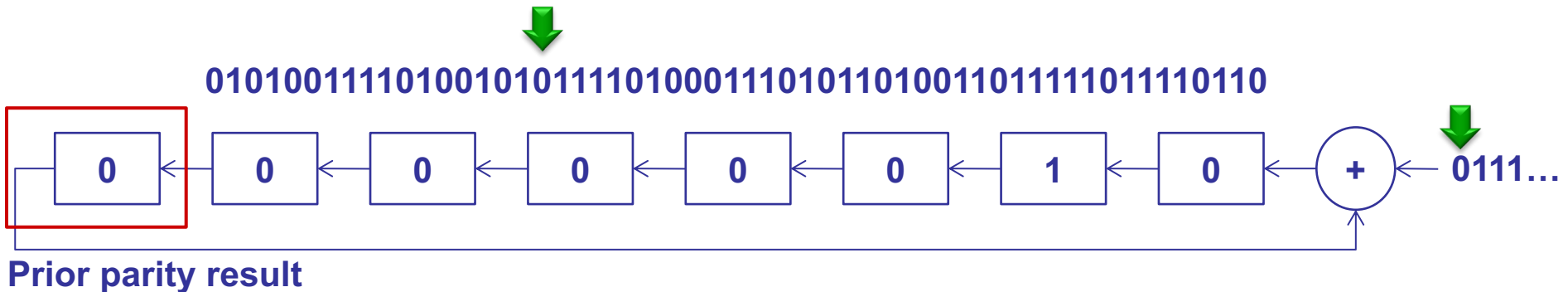
11010010

Parity Byte

10000001



Checksum Example

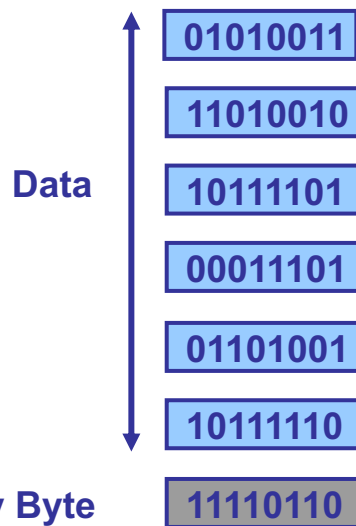
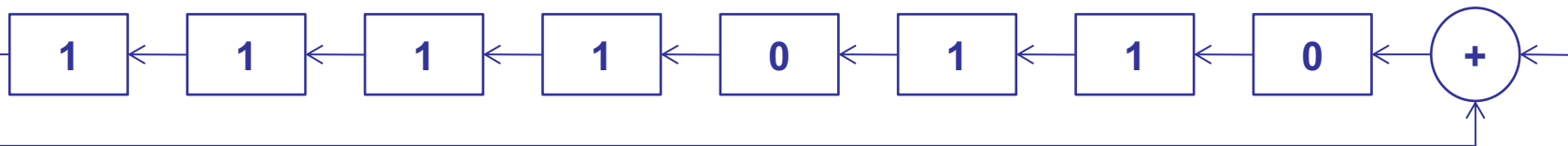




Checksum Example



01010011110100101011110100011101011010011011111011110110





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
 - ◆ So far just 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...



Modulo-2 Arithmetic

- Multiplication

$$\begin{array}{r} 1101 \\ 110 \\ \hline 0000 \\ 11010 \\ 110100 \\ \hline 101110 \end{array}$$

- Division

$$\begin{array}{r} 1101 \\ 110 \overline{) 101110} \\ \underline{110} \\ 111 \\ \underline{110} \\ 011 \\ \underline{000} \\ 110 \end{array}$$



Cyclic Remainder Check

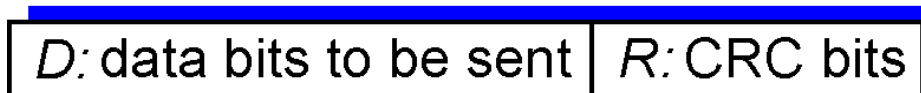
- Idea is to *divide* the incoming data, D , rather than add
 - ◆ The divisor is called the **generator**, g
- We can make a CRC resilient to k -bit burst errors
 - ◆ Need a generator of $k+1$ bits
- Divide $2^k D$ by g to get remainder, r
 - ◆ Remainder is called **frame check sequence**
- Send $2^k D - r$ (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
 - ◆ $\langle D, R \rangle$ exactly divisible by G (modulo 2)
 - ◆ Receiver knows G , divides $\langle D, R \rangle$ 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)

← d bits → ← r bits →



*bit
pattern*

$$D * 2^r \text{ XOR } R$$

*mathematical
formula*

CRC: Rooted in Polynomials



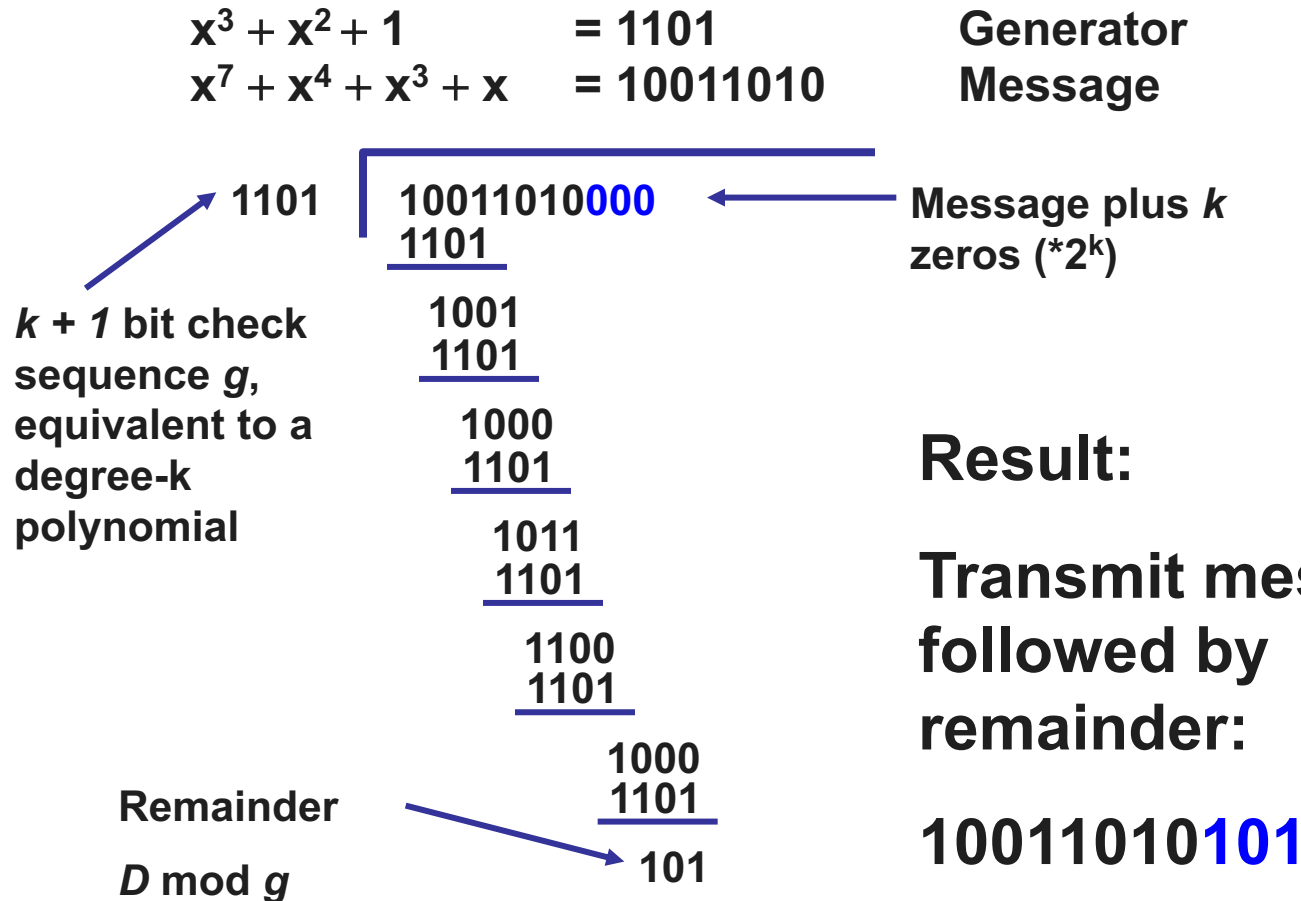
- We're *actually* doing polynomial arithmetic
 - ◆ Each bit is actually a coefficient of corresponding term in a k^{th} -degree polynomial

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

- Why do we care?
 - ◆ Can use the properties of finite fields to analyze effectiveness
 - ◆ Says any generator with two terms catches single bit errors



CRC Example Encoding





CRC Example Decoding

$$\begin{array}{lcl} x^3 + x^2 + 1 & = & 1101 \\ x^{10} + x^7 + x^6 + x^4 + x^2 + 1 & = & 10011010101 \end{array} \quad \begin{array}{l} \text{Generator} \\ \text{Received Message} \end{array}$$

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

Received message, no errors

Result:

CRC test is passed

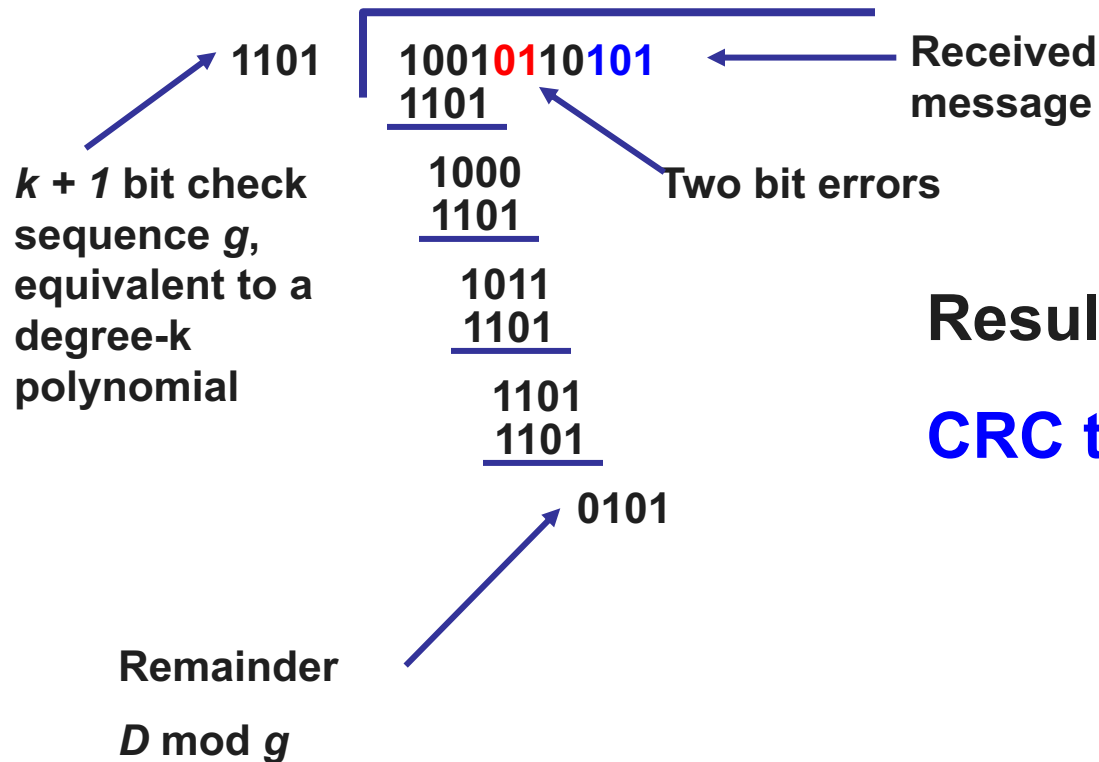
Remainder $D \bmod g$

$$\begin{array}{r} 1101 \overline{) 10011010101} \\ \underline{1101} \\ 1001 \\ \underline{1101} \\ 1000 \\ \underline{1101} \\ 1011 \\ \underline{1101} \\ 1100 \\ \underline{1101} \\ 1101 \\ \underline{1101} \\ 0 \end{array}$$



CRC Example Failure

$x^3 + x^2 + 1 = 1101$ Generator
 $x^{10} + x^7 + x^5 + x^4 + x^2 + 1 = 10010110101$ Received Message



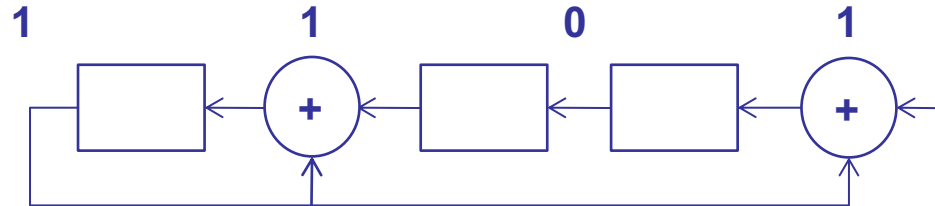


Common Generators

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	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$



CRC in Hardware



- Key observation is only subtract when MSB is one
 - ◆ Recall that subtraction is XOR
 - ◆ No explicit check for leading one by using as input to XOR
- Hardware cost very similar to most basic checksum
 - ◆ We're only interested in remainder at the end
 - ◆ Not a divider: Only need k registers as remainder is only k bits



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