Lecture #25

Error Detection

18-348 Embedded System Engineering

Philip Koopman

Monday, 29-April-2013

Electrical & Computer ENGINEERING

Carnegie Mellon

What About Errors In Digital Data?

- **Noise on serial buses is a fact of life**
  - In embedded systems, can easily be one bit error per 10^5 (or 10^6) bits
    - Does that matter?
  - At 9600 bps x 24 hours
    - 86,400 seconds/day; 829,440,000 bits per day ➔ ~8300 errors per day
  - CAN (serial network in cars) might run at 1Mbps ➔ ~ 1 million errors/day
    - Many will be single-bit errors, but many others will be multi-bit errors.

- **Is parity enough?**
  - All even numbers of bit errors are undetected by parity
  - AND, it costs too much for what you get (~10% bandwidth penalty)

- **Want a more general approach**
  - In case a noise burst creates multiple bit errors close together
  - In case network has periods of high noise, or otherwise sees many errors

Error Coding For Poets (who know a little discrete math)

- **The general idea of an error code is to mix all the bits in the data word to produce a condensed version (the check sequence)**
  - Ideally, every bit in the data word affects many check sequence bits
  - Ideally, bit errors in the code word have high probability of being detected
  - Ideally, a small number of bit errors is detected 100% of the time
  - At a hand-wave, similar to desired properties of a pseudo-random number generator
    - The Data Word is the seed value, and the Check Sequence is the pseudo-random number

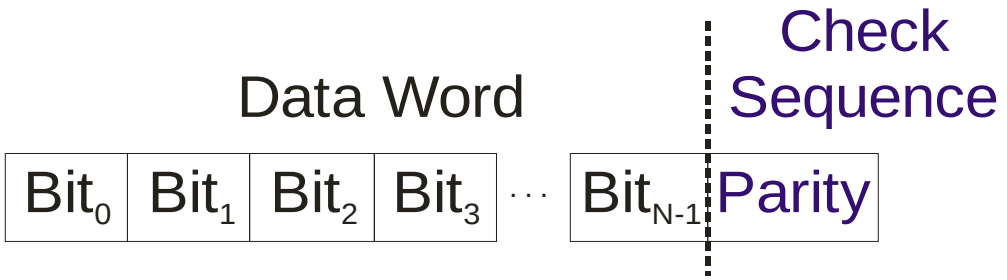
- **The ability to do this will depend upon:**
  - **The size of the data word**
    - Larger data words are bigger targets for bit errors, and are harder to protect
  - **The size of the check sequence**
    - More check sequence bits makes it harder to get unlucky with multiple bit errors
  - **The mathematical properties of the "mixing" function**
    - Thorough mixing of data bits lets the check sequence detect simple error patterns
  - **The type of errors you expect to get** (patterns, error probability)

Brute Force Replication

- **Duplicate values**
  - Put two copies of every value in memory
    - "Mirroring:" store value and also one's complement of value in two locations
  - Send every message twice
  - Detects errors, but if two values mismatch how to you know which is correct?
- **Triplicate values**
  - Put three copies of every value in memory
  - If two match and the third doesn't, assume the two that match are correct
- **How good are these at detecting/preventing corruption?**
  - A one-bit error in the same place in two copies will go undetected

Example: Parity As An Error Detection Code

- **Example error detection function:** Parity
  - XOR all the bits of the data word together to form 1 bit of parity



$$Parity = Bit_0 \oplus Bit_1 \oplus Bit_2 \oplus Bit_3 \oplus \ldots \oplus Bit_{N-1}$$

Note: ⊕ is eXclusive OR (XOR)
Parity = 0 for even number of "1" bits
Parity = 1 for odd number of "1" bits

XOR Facts:
$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$
(Think of it as Boolean addition and subtraction.)

- **How good is this at error detection?**
  - Only costs one bit of extra data; all bits included in mixing
  - Detects all odd number of bit errors (1, 3, 5, 7, … bits in error)
  - Detects NO errors that flip an even number of bits (2, 4, 6, … bits in error)
  - Performance: detects up to 1 bit errors; misses all 2-bit errors
  - Not so great – can we do better?

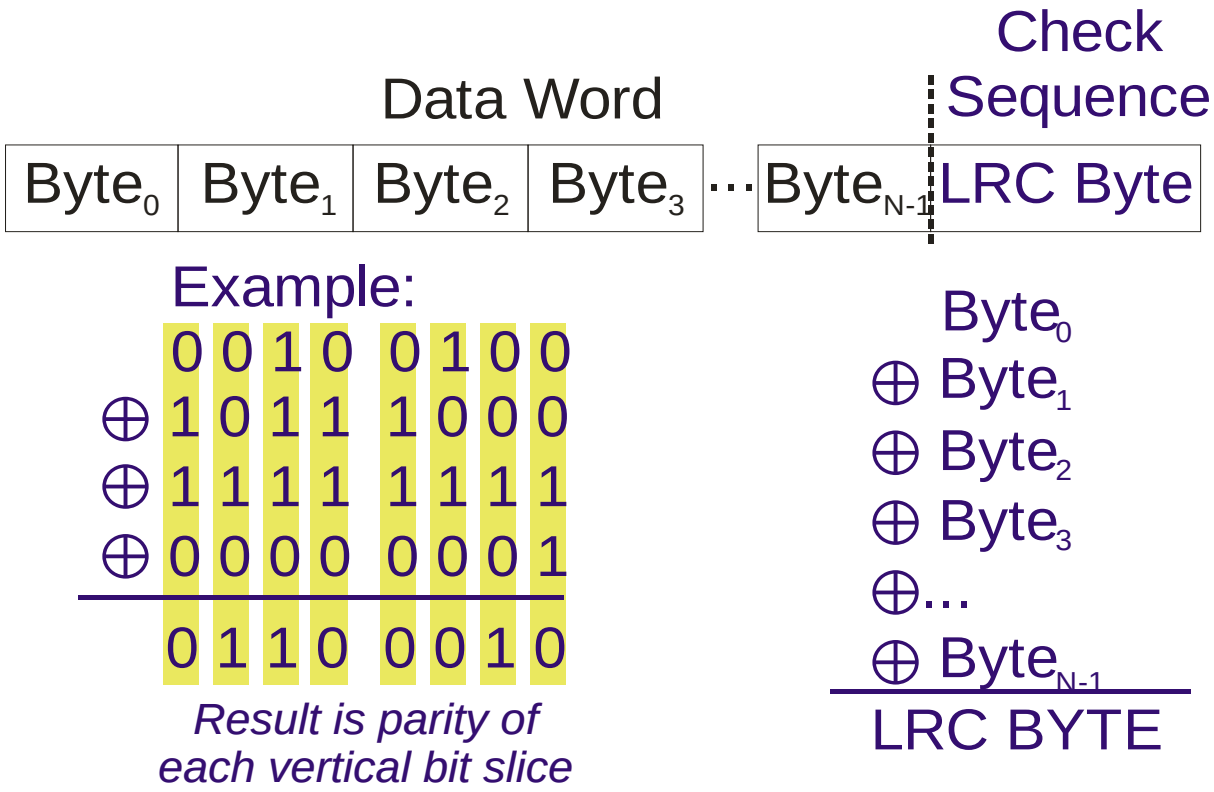
SECMED

- **Single Error Correction Multiple Error Detection (HD=4)**
  - A pattern of XORs across the data word that is carefully designed to identify the location of any single bit error
    - Essentially, a lot of parity bits across different subsets of the data word
    - Pattern is tricky and I don't expect you to know it – (Wikipedia explains it)
  - If you know the location, you can flip the bad bit back ➔ error correction
  - Also detects any odd number of bit errors (includes a classical parity bit)
  - May detect many other error patterns depending upon design
    - For example, might design to detect 4 bit errors in a single x4 DRAM chip
  - Cost is ~Log_2N bits of error code for an N-bit code word (includes SECMED code size)

- **Typical uses:**
  - DRAM memory for servers and high-dependability systems
    - If data is corrupted you need to recover it – this is a classic hardware technique
  - Used in very low-end wireless transmission (e.g., remote keyless entry)
    - If transmission is garbled don't want to spend power on a retry
  - Usually NOT used for wired data or software RAM techniques
    - If you can retry easily, don't need error correction
    - Computationally too expensive for protecting in-RAM data

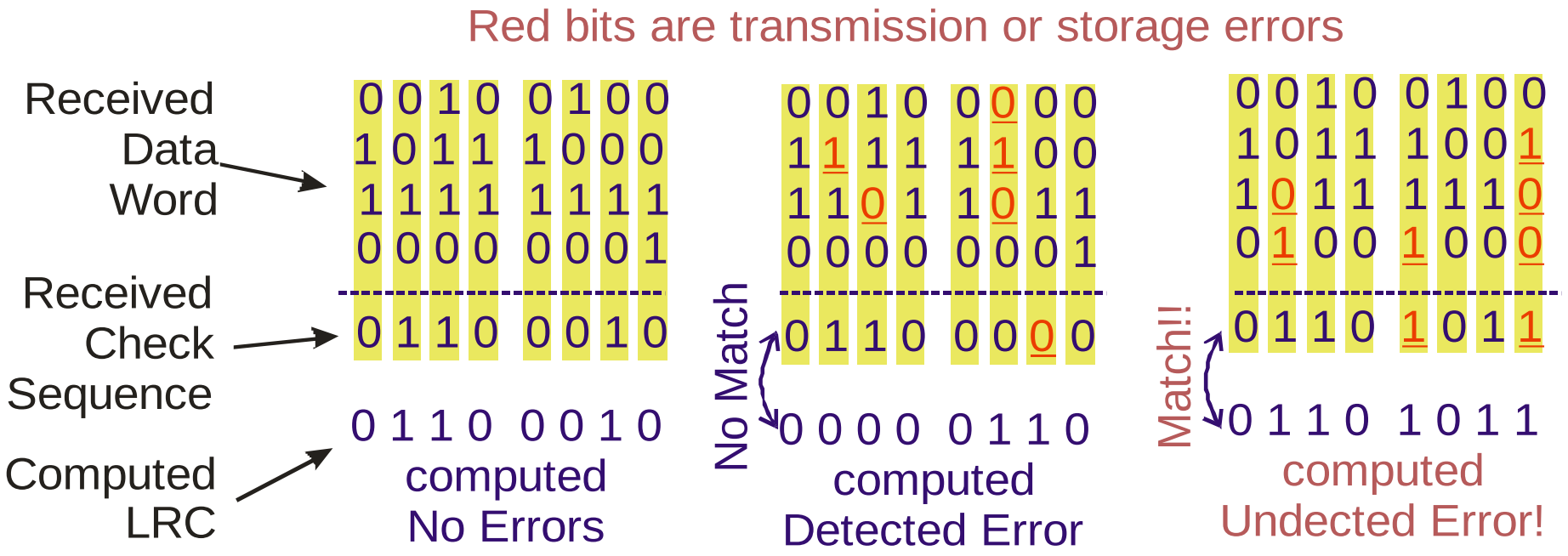
Example: Longitudinal Redundancy Check (LRC)

- **LRC is a byte-by-byte parity computation**
  - XOR all the bytes of the data word together, creating a one-byte result
  - (This is sometimes called an "XOR checksum" but it isn't really addition, so it's not really a sum)



How Good Is An LRC?

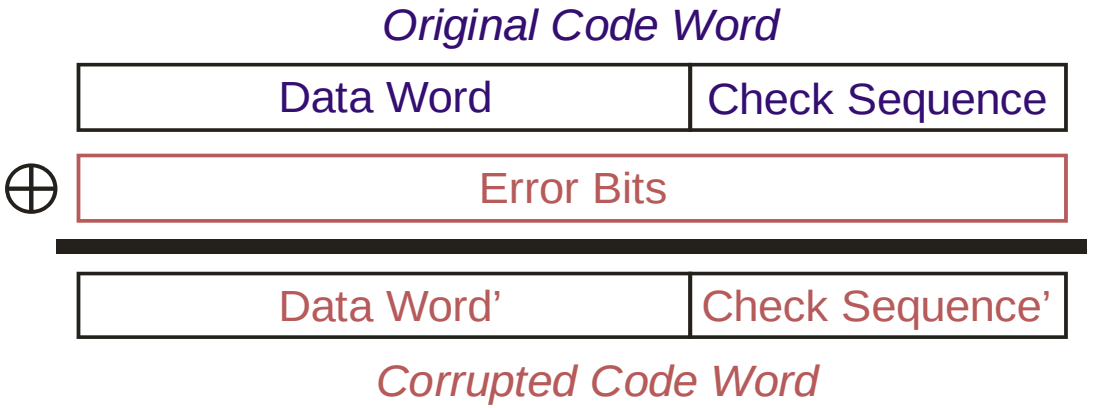
- **Parity is computed for each bit position (vertical stripes)**
  - Note that the **received copy of check sequence** can be corrupted too!



- **Detects all odd numbers of bit errors in a vertical slice**
  - Fails to detect even number of bit errors in a vertical slice
  - Detects all 1-bit errors
  - Detects many 2-bit errors, **<u>but not *all* 2-bit errors</u>**
    - Any 2-bit error in same vertical slice is undetected

Basic Model For Data Corruption

- **Data corruption is "bit flips" ("bisymmetric inversions")**
  - Each bit has some probability of being inverted
  - **"Weight"** of error word is number of bits flipped (number of "1" bits in error)



- **Error detection works if the corrupted Code Word is invalid**
  - In other words, if corrupted Check Sequence doesn't match the Check Sequence that would be computed based on the Data Word
  - If corrupted Check Sequence just happens to match the Check Sequence computed for corrupted data, you have an **undetected error**
  - All things being equal (which they are **not!!!**) probability of undetected error is 1 chance in 2^k for a k-bit check sequence

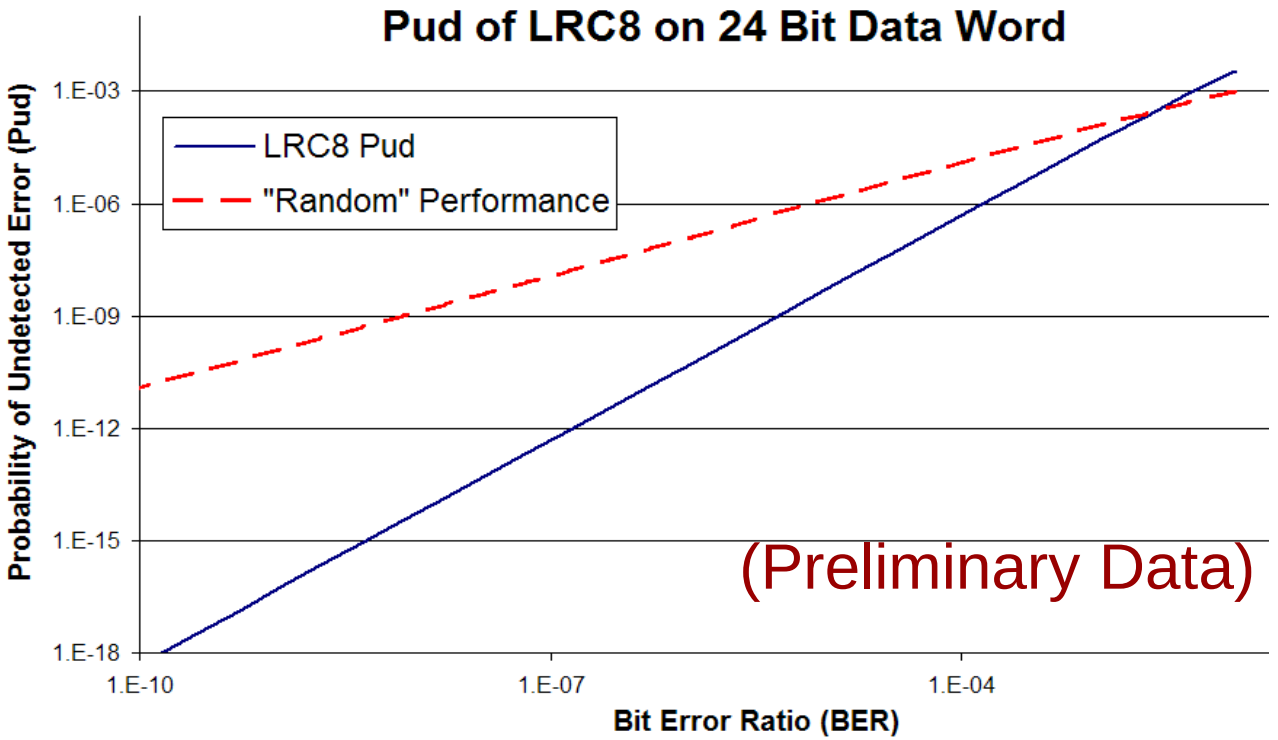
Error Code Effectiveness Measures

- **Metrics that matter depend upon application, but usual suspects:**
  - **Maximum weight** of error word that is 100% detected
    - Hamming Distance (HD) is lowest weight of any undetectable error
    - For example, HD=4 means all 1, 2, 3 bit errors detected
  - **Fraction of errors** undetected for a given number of bit flips
    - Hamming Weight (HW): how many of all possible m-bit flips are undetected?
      - » E.g. HW(5)=157,481 undetected out of all possible 5-bit flip Code Word combinations
  - **Fraction of errors** undetected at a given random probability of bit flips
    - Assumes a **Bit Error Ratio (BER)**, for example 1 bit out of 100,000 flipped
    - Small numbers of bit flips are most probable for typical BER values
  - **Special patterns** 100% detected, such as adjacent bits
    - Burst error detection – e.g., all possible bit errors within an 8 bit span
  - Performance usually depends upon data word size and code word size

- **Example for LRC8 (8 bit chunk size LRC)**
  - HD=2 (all 1 bit errors detected, not all 2 bit errors)
  - Detects all 8 bit bursts (only 1 bit per vertical slice)
  - Other effectiveness metrics coming up…

LRC8 Error Detection Effectiveness (Pud)

- **Assume random independent bit flips**
  - For a given BER, can determine Probability of UnDetected error (Pud)

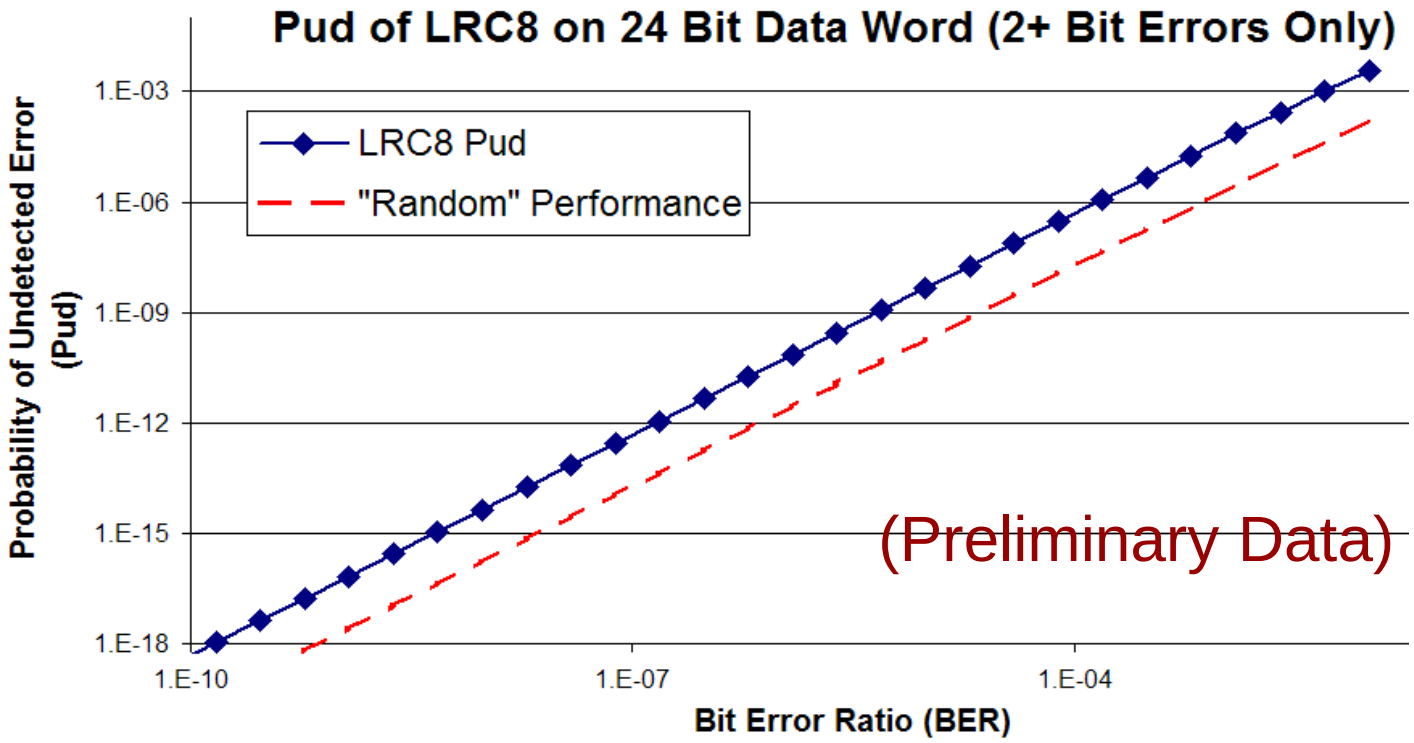


DOWN IS GOOD

- **Most networks won't work at all for BER > 10^{-2} so we're OK, right?** WAIT!!!
- **Consider at BER of 10^{-6} (Pud = $4.8 * 10^{-11}$)**
  - 1MBps ➔ 8.64 * 10^{10} 32-bit data chunks per day
    ➔ **4 undetected errors/day**

LRC8 With 2 Or More Bit Errors

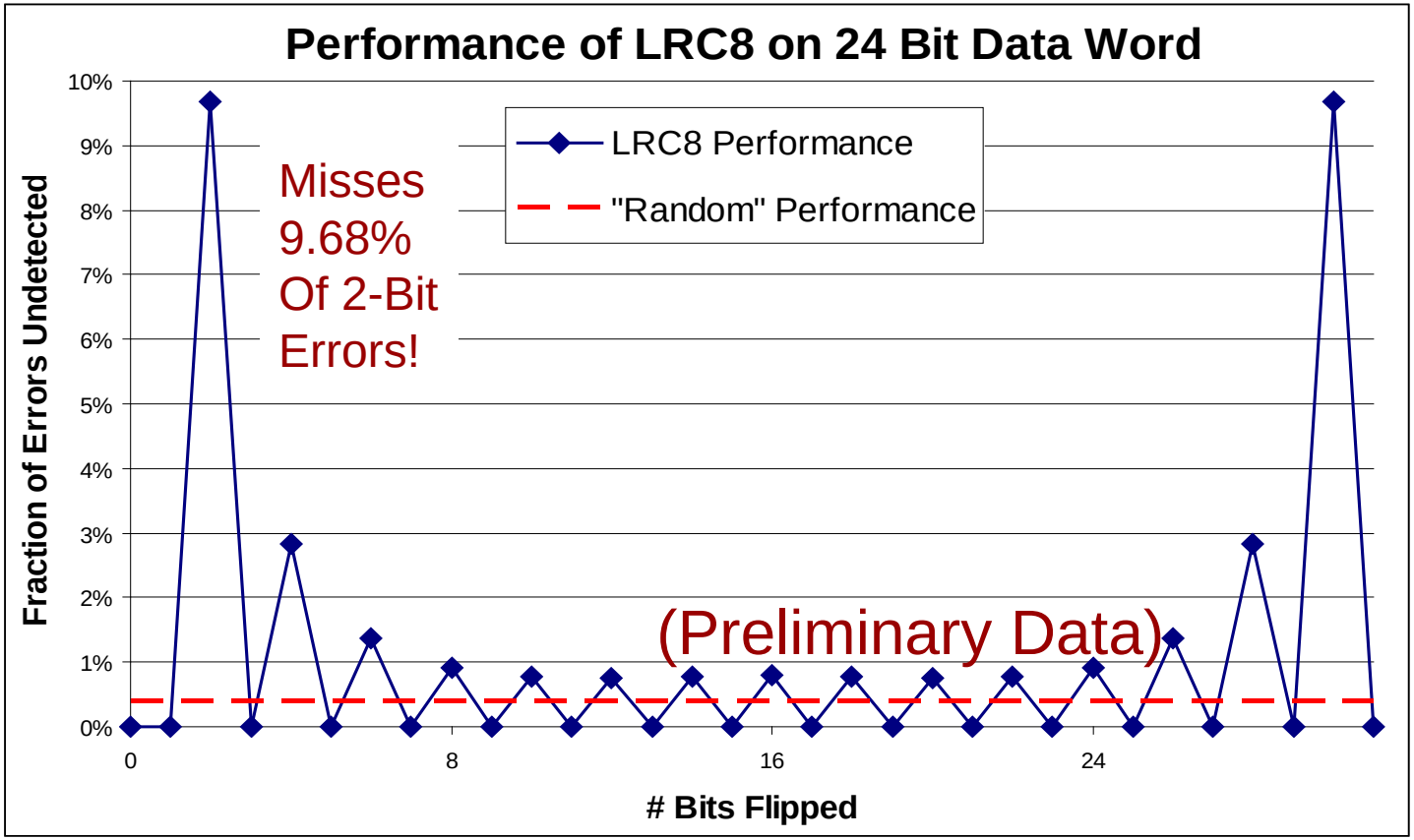
- **At low BER almost all errors involve 1 bit**
  - LRC8 catches all 1 bit errors, so makes things look great
  - Those 4 failures/day are mostly coming from undetected 2-bit errors
  - LRC8 is **a factor of 25 worse** than "random" for 2 bit errors!



DOWN IS GOOD

LRC8 Error detection Effectiveness (Fraction)

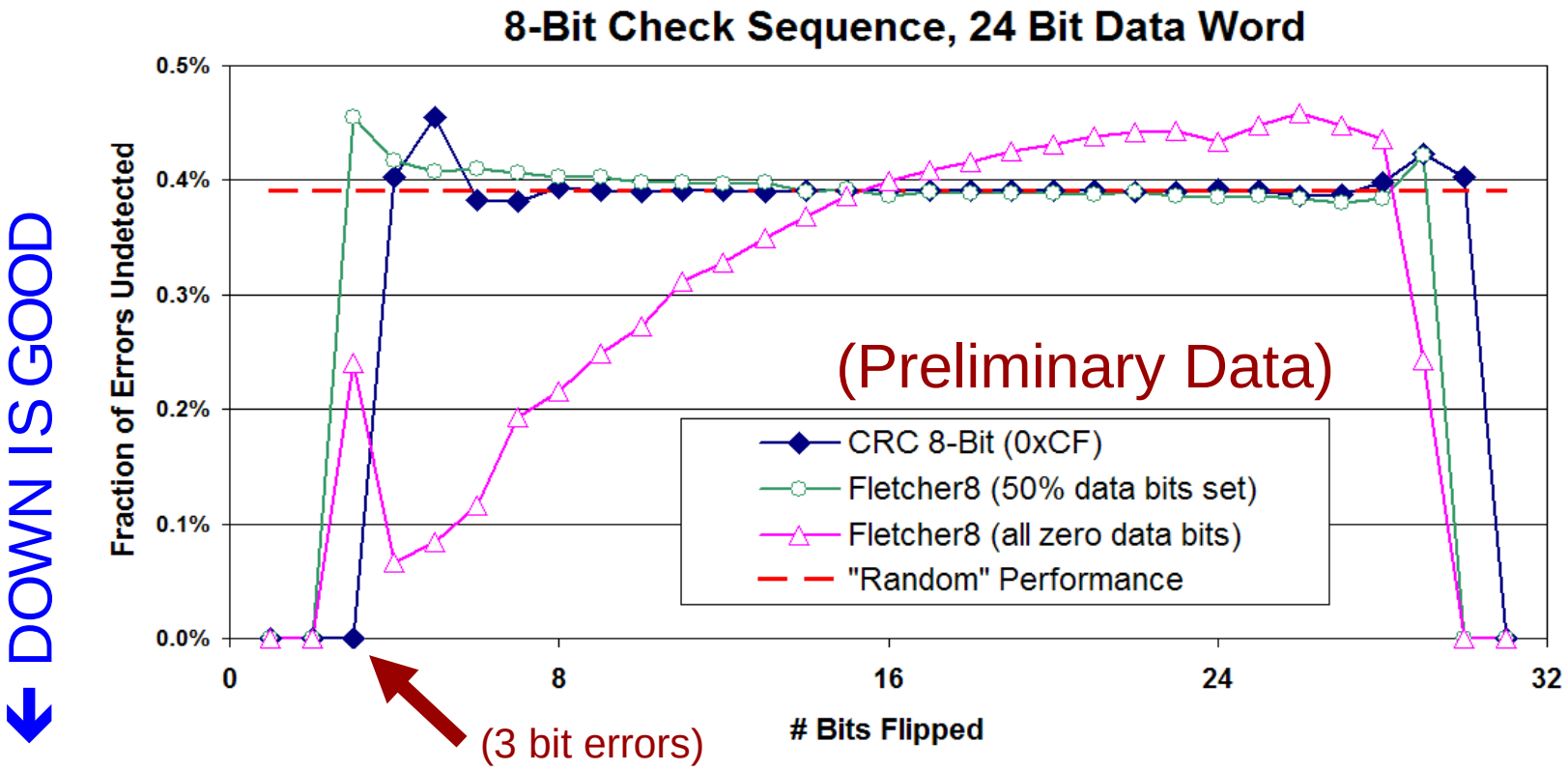
- **Each data point is # undetected / # total bit error patterns**
  - Assumes every bit error pattern is equally likely for given HW
  - 2-bit errors are especially vulnerable – and they may be quite likely!
  - **General rule: what happens for small # of bit flips is what matters most**



DOWN IS GOOD

Can We Do Better?

- YES – A whole lot better!
- **8-bit Fletcher Checksum (performance varies based on data)**
- **Optimal 8-bit CRC (detects all 1-, 2-, 3-bit errors at this length)**
  - Detecting all 3-bit errors ***dramatically*** improves Pud



Checksum (review)

- **Checksums (add up all the bytes) are better than parity (HD=2 or 3)**
  - XOR checksum is just individual parity for each bit in message
  - ADD costs same as XOR, but gives better mixing due to carry bits

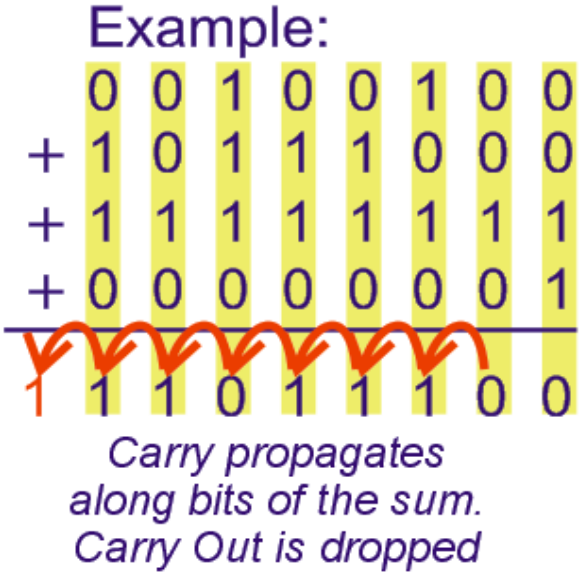
```
unit8 lrc = SEED;
for( int i = 0; i < length; i++) { lrc = lrc + data[i]; }
data[length] = lrc;
```

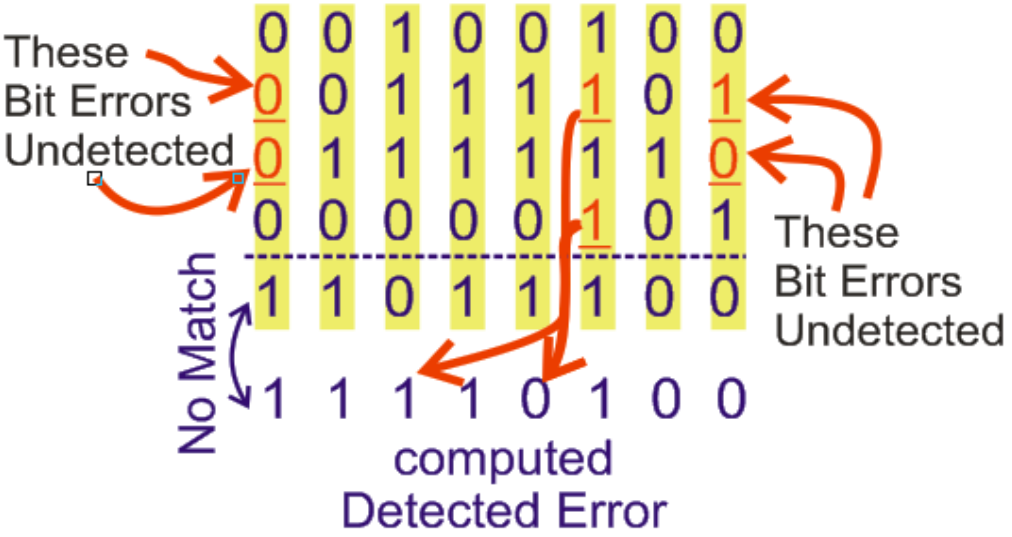
  - Still, in worst case can miss small errors that hit just the wrong way

- **If you use a checksum, use 1's complement addition**
  - Gives better coverage than 2's complement addition; catches carry-out bits
    - 2's complement misses double-bit errors on top bit position; 1's complement doesn't
    - 1's complement about 12.5% better for 8-bit checksum
  - About twice as good as XOR-based checksum
  - About the same speed (now that you know the ADC wrap-around trick from the optimization lectures!)

Integer Addition Checksum

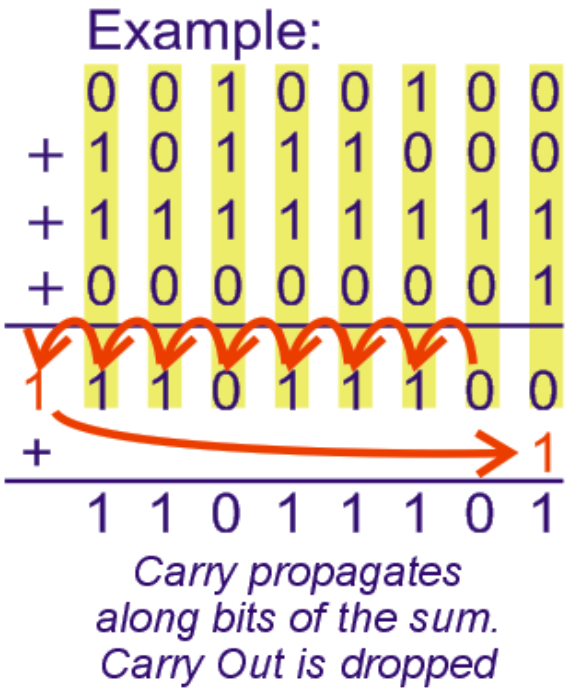
- **Same as LRC, except use integer "+" instead of XOR**
  - The carries from addition promote bit mixing between adjacent columns
    - Can detect errors that make two bits go 0 ➔ 1 or 1 ➔ 0 (except top-most bits)
    - Cannot detect compensating errors (one bit goes 0 ➔1 and another 1 ➔ 0)
  - Carry out of the top bit of the sum is discarded
    - No pairs of bit errors are detected in top bit position

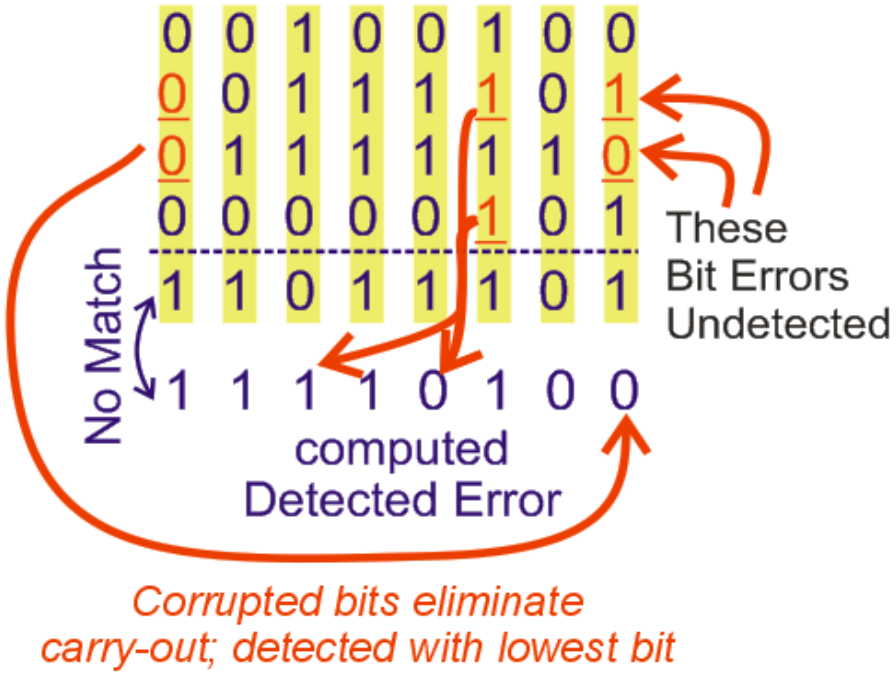




One's Complement Addition Checksum

- **Same as integer checksum, but add Carry-Out bits back**
  - Plugs error detection hole of two top bits flipping with the same polarity
  - But, doesn't solve problem of compensating errors
  - Hamming Distance 2 (HD=2); some two-bit errors are undetected





Advanced Checksums

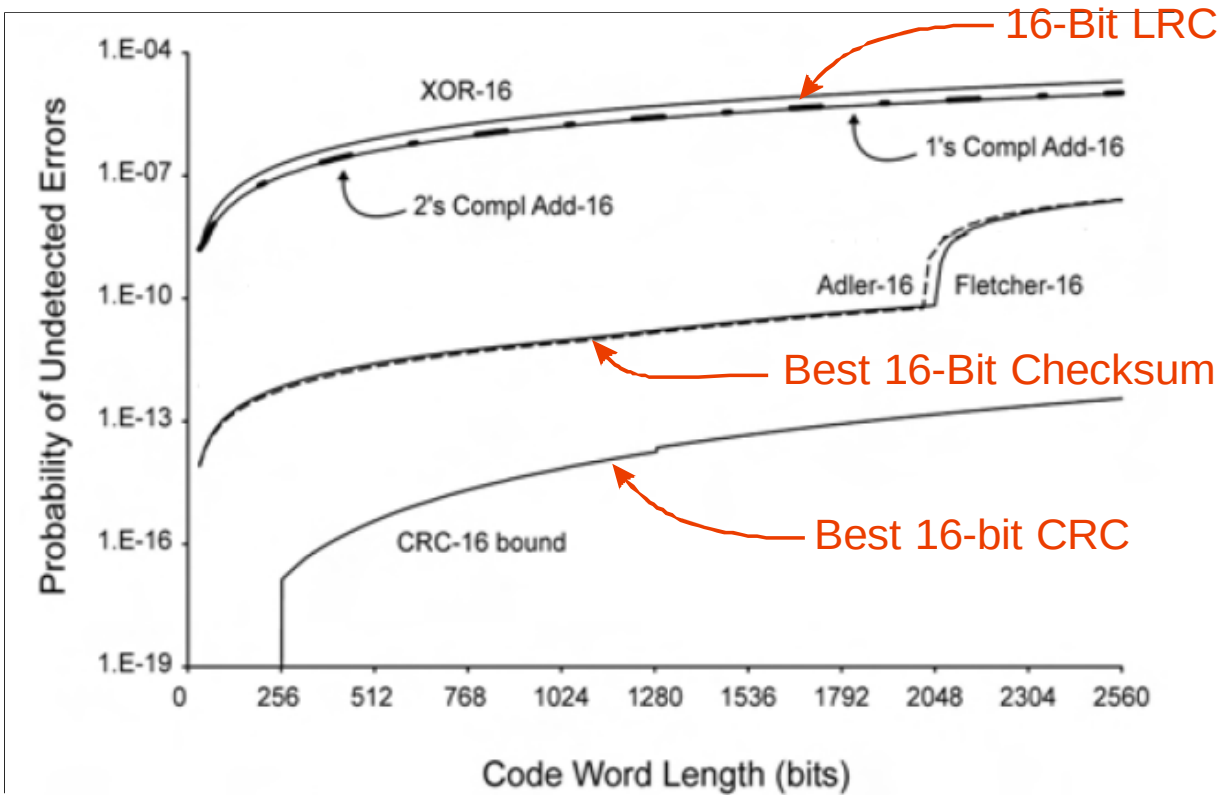
- **Fletcher Checksum – use two running one's complement sums**
  - HD = 3 for short codewords; HD=2 for long codewords
  - This example generates a 16-bit Fletcher Checksum on 8-bit chunks

```
unit8 a = 0; unit8 b = 0;
for( int i = 0; i < length; i++)
{ a = OnesCompAdd(a,data[i]);
  b = OnesCompAdd(a,b);
}
data[length] = a;    data[length+1] = b;
```

- **Properties**
  - Better bit mixing – less vulnerable to same bit-position errors
    - Detects all 1 and 2-bit errors until the sum a rolls over (at almost 256 bytes)
  - Significant improvement comes from the running sum b
    - b = data[length-1] + 2*data[length-2] + 3*data[length-3]…
    - This allows it to catch byte values that are out of order, which are missed by regular checksums
  - Another similar variant, Adler checksums, aren't worth the trouble; use Fletcher

Can We Do Better Than A Checksum?

- **Can often get HD=6 (detect all 1, 2, 3, 4, 5-bit errors) with a CRC**
- **Note that this is a different graph that happens to be previously published**
  - 16 bit Check Sequences; longer code word lengths
- **For this graph, assume Bit Error Rate (BER) = 10^{-5} flip probability per bit**



DOWN IS GOOD

Source:

Maxino, T., & Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," IEEE Trans. on Dependable and Secure Computing, Jan-Mar 2009, pp. 59-72.

CRC – Better Bit Mixing

- **Error detection is all about mixing together message bits**
  - Hopefully in a way so that lots of errors have to hit just the wrong way to go undetected!

- **CRC – Cyclic Redundancy Code**
  - Shifts bits into an XOR-based mixing register
  - Can often guarantee detection of multiple bit errors
  - Slower than checksum, but still useful

Table 1. Bitwise Left-Shift CRC Algorithm

```
for (i=0; i<sizeof(data); i++) {
  if (msb(data) ^ msb(crc)) {
    crc = (crc << 1) ^ (poly);
  } else {
    crc = (crc << 1);
  }
  data <<= 1;
}
```

- **Caution!**
  - Much of the published lore about CRCs is *incorrect*
  - One size does not fit all (there is no single best feedback polynomial)
  - Some published polynomials have bugs in them (incorrect values)
    - Even in Numerical Recipes in C 2nd Ed. (newest edition fixed based our feedback)
    - Even in scholarly journal papers

Mathematical Basis of CRCs

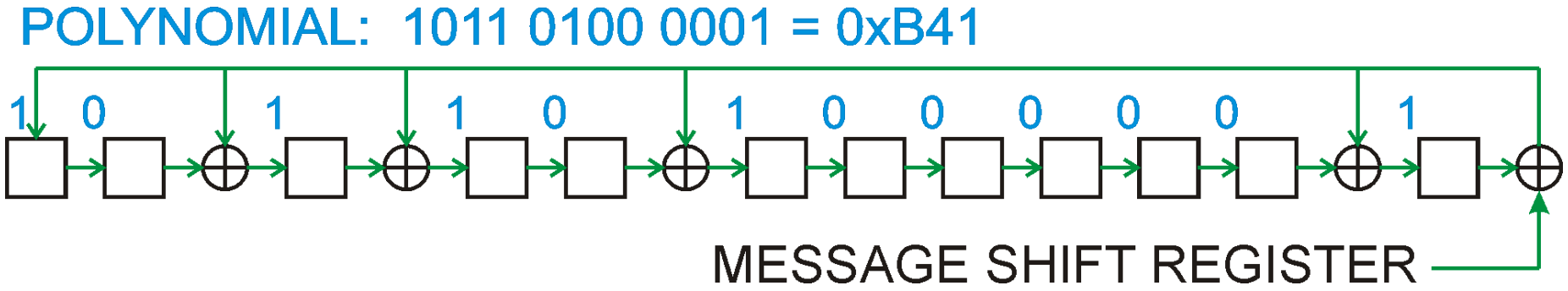
- **Use polynomial division over Galois Field(2) (remember that from high school?) (this is a mathematician thing)**
  - At a hand-waving level this is division using Boolean Algebra
    - "Add" and "Subtract" used by division algorithm both use XOR

```
11010011101100 000 <--- Data Word left shifted by 3 bits
1011               <--- 4-bit divisor is 1011  x³ + x + 1
01100011101100 000 <--- result of first conditional subtraction
 1011              <--- divisor
00111011101100 000 <--- result of second conditional subtraction
  1011             <--- continue shift-and-subtract ...
00010111101100 000
   1011
00000001101100 000
       1011
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
-----------------   Remainder is the Check Sequence
00000000000000 100 <--- Remainder (3 bits)
```

[Wikipedia]

Classical CRC Overview

- **Cyclic Redundancy Code operation**
  - Computes a (non-secure) message digest using shift and XOR
  - This is a hardware implementation of polynomial division



$0xB41 = x^{12}+x^{10}+x^{9}+x^{7}+x+1$ (the "+1" is implicit in the hex value)

$= (x+1)(x^{3}+x^{2}+1)(x^{8}+x^{4}+x^{3}+x^{2}+1)$

MESSAGE PAYLOAD	CRC Remainder

- Detected error if received digest doesn't match CRC Remainder of payload

Aren't CRCs Really Slow?

- **Speedup techniques have been known for years**
  - Important to compare best implementations, not slow ones
- **256-word lookup table provides about 4x CRC speedup**
  - Careful polynomial selection gives 256-byte table and ~8x speedup
  - Intermediate space/speedup approaches can also be used
  - Ray, J., & Koopman, P. "Efficient High Hamming Distance CRCs for Embedded Applications," DSN06, June 2006.
- **In a system with cache memory, CRCs are probably not a lot more expensive than a checksum**
  - Biggest part of execution time will be getting data bytes into cache
  - We are working on a more definitive speed tradeoff study

Is Using A CRC Worth It?

- **Checksums can be faster** **(although this is usually overstated)**
  - **But give far worse error performance**
    - Most checksum folklore is based on comparing to a *bad* CRC or with *non-representative* fault types

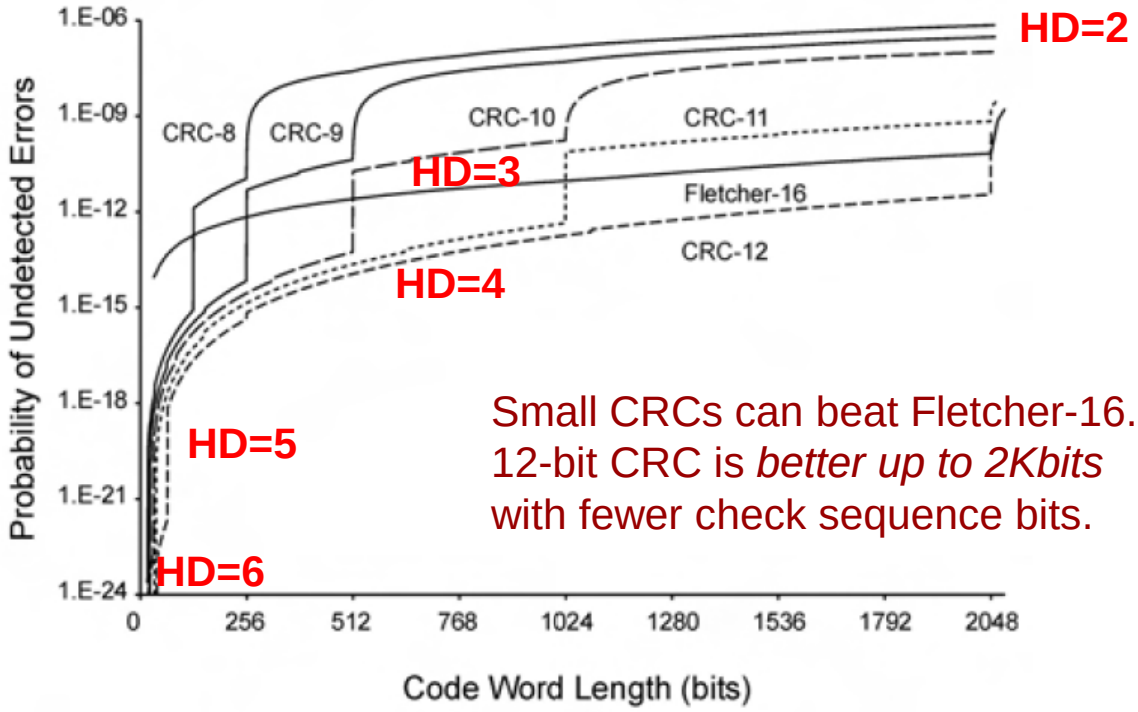


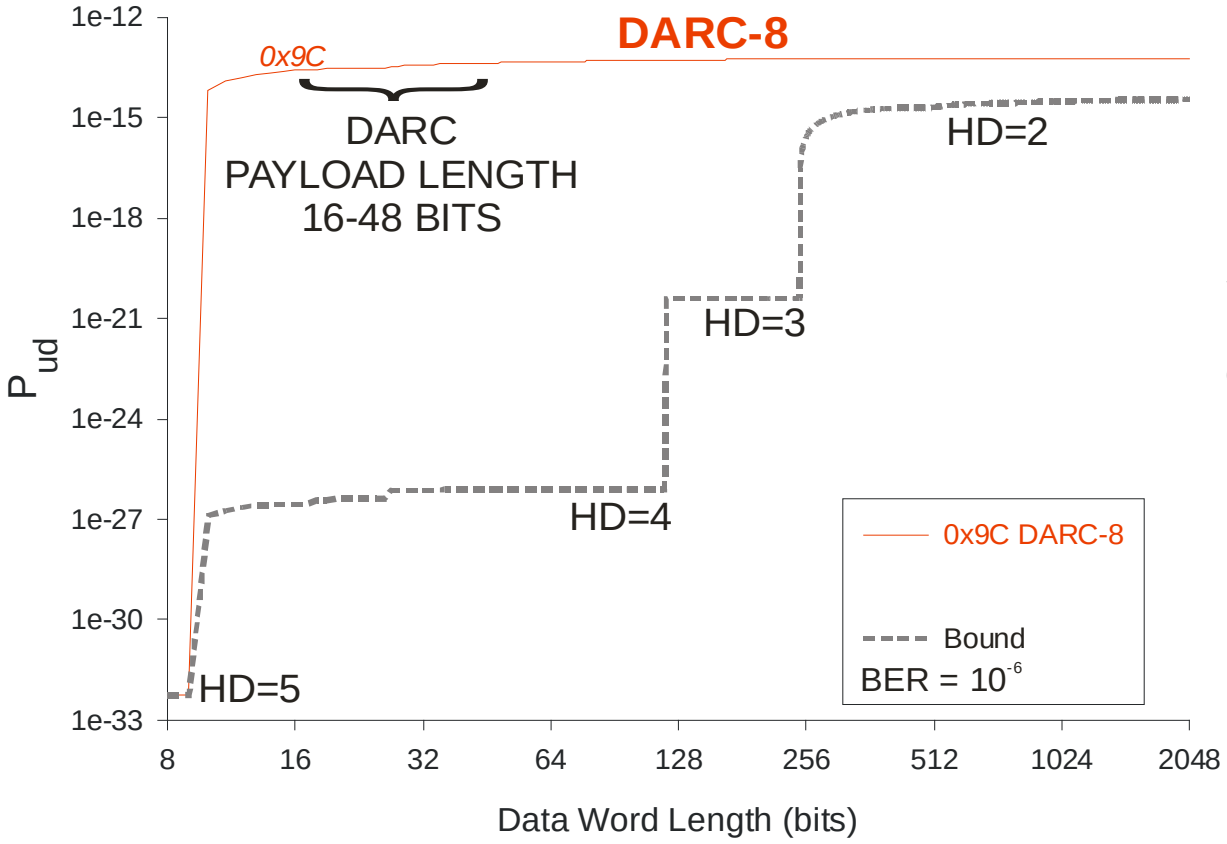
Fig. 12. Probability of undetected errors for Fletcher-16 and CRC bounds for different CRC widths at a BER of 10^{-5}. Data values for Fletcher-16 are the mean of 10 trials using random data.

Source:

Maxino, T., & Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," IEEE Trans. on Dependable and Secure Computing, Jan-Mar 2009, pp. 59-72.

What Happens When You Get The CRC Wrong?

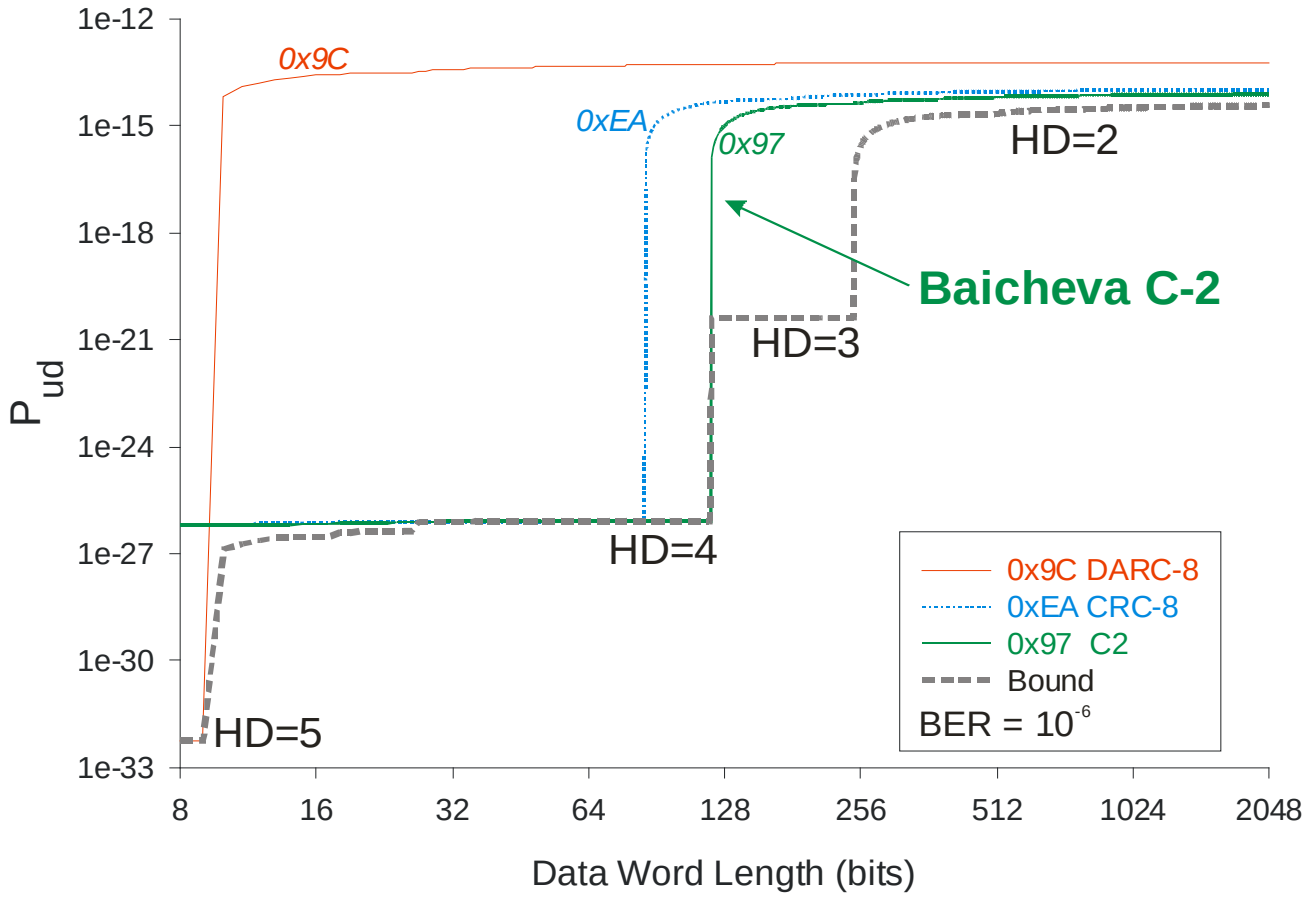
- **DARC (Data Radio Channel), ETSI, October 2002**
  - DARC-8 polynomial is optimal for 8-bit payloads
  - BUT, DARC uses 16-48 bit payloads, and misses some 2-bit errors
  - Could have detected all 2-bit and 3-bit errors with same size CRC!



Source:
Koopman, P. & Chakravarty, T., "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks," DSN04, June 2004

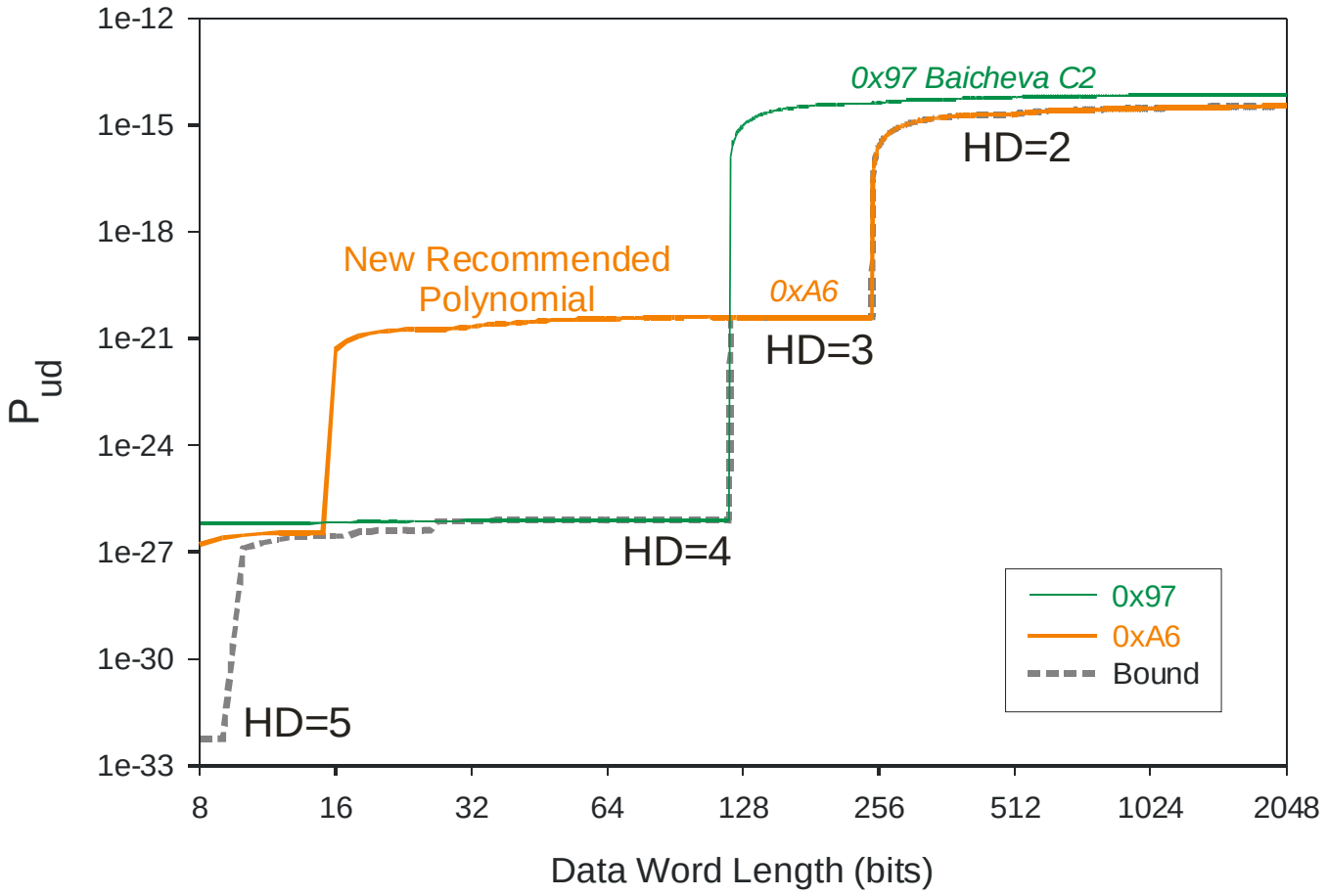
Baicheva's Polynomial C2

- **[Baicheva98] proposed polynomial C2, 0x97**
  - Recommended as good polynomial to length 119
  - Dominates 0xEA which is the "standard" 8-bit CRC (better P_{ud} at every length)



But What If You Want the HD=3 Region?

- **We found that 0xA6 has good performance**
  - Better than C2 and near optimal at all lengths of 120 and above



Source: Koopman, P. & Chakravarty, T., "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks," DSN04, June 2004

How To Pick A Good CRC Polynomial

- HD "Hamming Distance" – polynomial is guaranteed to detect all errors with *fewer* than HD bits flipped (so, it misses some with HD bits flipped)
- CRC size – number of bits in CRC field
- Length – number of bits in data payload (excluding CRC)
- Polynomial – hex value of feedback "poly" for bit mixing

Table 3. "Best" polynomials for HD at given CRC size and data word length. Underlined polynomials have been previously published as "good" polynomials. [Koopman04]

Max length at HD / *Polynomial*	CRC Size (bits) 3	4	5	6	7	8	9	10	11	12	13	14	15	16
HD=2	2048+ 0x5	2048+ 0x9	2048+ 0x12	2048+ 0x21	2048+ 0x48	2048+ 0xA6	2048+ 0x167	2048+ 0x327	2048+ 0x64D	–	–	–	–	–
HD=3		11 0x9	26 0x12	57 0x21	120 0x48	247 0xA6	502 0x167	1013 0x327	2036 0x64D	2048 0xB75	–	–	–	–
HD=4			10 0x15	25 0x2C	56 0x5B	119 0x97	246 0x14B	501 0x319	1012 0x583	2035 0xC07	2048 0x102A	2048 0x21E8	2048 0x4976	2048 0xBAAD
HD=5						9 0x9C	13 0x185	21 0x2B9	25 0x5D7	53 0x8F8	none	113 0x212D	136 0x6A8D	241 0xAC9A
HD=6							8 0x13C	12 0x28E	22 0x532	27 0xB41	52 0x1909	57 0x372B	114 0x573A	135 0xC86C
HD=7									12 0x571	none	12 0x12A5	13 0x28A9	16 0x5BD5	19 0x968B
HD=8										11 0xA4F	11 0x10B7	11 0x2371	12 0x630B	15 0x8FDB

Good Polynomial Examples:

- **What is the best HD you can get for:**
  - 112 bit data word length
  - 15-bit CRC
  - (what polynomial should you use?)

- **What is the smallest CRC size you need to attain:**
  - 2015 bit data word length
  - HD=4
  - (what polynomial should you use?)

- **Given polynomial 0x167**
  - What is the longest data word for HD=1?
  - What is the longest data word for HD=2?
  - What is the longest data word for HD=3?

- **Resource under construction: http://checksumcrc.blogspot.com/**

Where Are We Now?

- **Where we've been:**
  - Various flavors of I/O
- **Where we're going today:**
  - Error detection codes
  - Therac 25 – a case study of why you need to get actuator settings right
- **Where we're going next:**
  - Bluetooth & CAN embedded networks
  - System resets & robustness
  - Test #2 on Wed April 25
  - Lab 11 final demos on or before Wed May 9
  - Lab 11 write-ups due Friday May 11 by 9:00 PM.