Correction from Monday's Lecture

	0	*0*	*0*	*0*	*1*	*1*	*0*	*1*	*1*	*0*	*1*	*0*	*Count*	*Syn-drome*
Group 1	X		X		X		X		X		X		3	0
Group 2		X	X			X	X			X	X		2	1
Group 3				X	X	X	X					X	2	1
Group 4								X	X	X	X	X	3	0

- Groups 2 and **3** fail; the error bit is in **0110** = 6
- Correct the received data to: 000010011010
- Delete the parity bits: 01001010
- Which is the same as the original: 01001010

Error Detecting Checksums - Outline

- Introduction
- Fletcher Checksum
- Alder Checksum
- Polynomial Checksums
- CRC Checksums

Introduction

- Checksums are used to detect errors
- Send the message and append a checksum
- The receiver detaches the messages and performs the same calculations on the message.
  - Receiver then checks if the resulting checksum = the received checksum

Fletcher's Checksum

- Another variant of 1s complement checksum
  - Slightly slower, and more complicated to implement
  - Can overcome the problems of simply summing up all the bytes
    - Can detect inserting/deleting zero byte values
    - Incrementing/decrementing of bytes in opposite directions
    - Reordering of bytes

Fletcher's Checksum Ctd.

- Calculated over sequences of 2 octets
- Has sums modulus 255
- Uses two sums, *s*1 and *s*2
  - A straight 1s complement sum
    - $s1 = (s1 + c_i) \bmod 255$
  - A higher order sum of running sums
    - $s2 = (s1 + s2) \bmod 255$
- Checksum is the 16-bit concatenation of *s*1 and *s*2
  - Checksum=(256*s1+s2);
- The result is correct if *either* *s*1 *or* *s*2 are *zero*

Fletcher's Checksum

- Stated to be nearly as powerful as CRC-16 checks
- It detects
  - All single bit errors
  - All double bit errors
  - All but 0.000019% of burst errors up to length 16
  - All but 0.0015% of longer burst errors

Fletcher's Checksum Code

```
int s1 = 0, s2 = 0;      // initialize checksums
for (int i = 0; i < nChars; i++)    // scan the
                                    // characters
{
   s1 += c[i];           // add in the character
   while (s1 >= 255){    // reduce modulo 255
      s1 -= 255;
   }
   s2 += s1;             // get the sum of sums
   while (s2 >= 255){    // modulo 255
      s2 -= 255;
   }
}
```

Alder-32 Checksum

- Almost as reliable as CRC-32
- But can be forged easily and is not ideal against intentional modification
- Modification on the Fletcher checksum
- Not very good on short messages shorter than a few 100 bytes as checksums for these messages have poorer coverage of the 32 available bits

Alder-32 Algorithm

- Same as Fletchers checksum but uses prime number modulo and 16-bit sums
- Values are concatenated into a 32-bit integer
- Calculate two 16-bit checksums, *s*1 and *s*2
  - $s1 = (s1 + c_i) \bmod 65521$
  - $s2 = (s1 + s2) \bmod 65521$
- sums are done modulo 65521
  - The largest prime number < 2^{16}
- The checksum is the 32 bits $65536*s1 + s2$

Modulus Checks

- This important class of checks is an extension of simple parity. It can be demonstrated by some decimal examples
- Example
  - k = 23145

Modulus Checks - Example

- Choose an agreed modulus, *m*
  - A prime number
  - Just less than a power of 10
    - *m = 7*
    - *m = 97*
- Extend the number by zeroes as many as digits in the modulus
- Divide the *extended* number by *m* and get the remainder *r*
  - 231450 mod 7 =2
  - 2314500 mod 97 = 80
- Replace the zero extension with (*m-r)*, to give
  - (m – 2) = 7 – 2 = 5 = 231455
  - (m – 80) = 97 – 80 = 17 = 2314517

Modulus Checks - Example

- Encode or transmit this new extended number
- To check on reception, divide the received number by the agreed modulus
  - The remainder should be zero
- Discard the last (extension or remainder) digits and deliver the preceding digits as the verified number

Polynomials involved in checking

- The following checks work on bit vectors, and regards the bits as coefficients of polynomial in some arbitrary variable *x*
  - *i(x)* Information The information or data to be checked
  - *g(x)* Generator The system-defined divisor polynomial
  - *c(x)* Codeword What is transmitted; *i(x)* with (*i(x)* mod *g(s))* appended
  - *e(x)* Error The error vector; $e(x)=x^i$ for a single bit error
  - *v(x)* Received What is received; *v(x)=e(x)+c(x)*

Polynomials involved in checking

- Calculate the syndrome
  - $S(x) = v(x) \bmod g(x)$
    $= [e(x) + c(x)] \bmod g(x)$
    $= e(x) \bmod g(x) + c(x) \bmod g(x)$
    $= e(x) \bmod g(x) + 0$ by construction
- The syndrome $s(x)$ is a function only of the error vector $e(x)$

Cyclic Redundancy Checks (CRC)

- Most checking now uses Cyclic Redundancy Checks, which treats the bits of the message as coefficients of an "*information polynomial" i(x),* divides it by another "*generator"* polynomial *g(x)* and sends the remainder as the check digits at the end of the message
- The *principles* are identical to the previous slide, but *details* are different

Cyclic Redundancy Checks

- Does error checking based on polynomial division
- Each bit in the string is interpreted as a polynomial
  - The set of polynomials where each coefficient is 1 bit
  - 100101 $= x^5 + x^2 + 1$
  - 100000111 $= x^8 + x^2 + x + 1$

CRC Algorithm

- Given a bit string, append several 0s and call it *b*
  - Let *b(x)* be the polynomial corresponding to *b*
- Divide *b(x)* by an agreed on polynomial *g(x)* – *the generator polynomial* and determine the remainder *r(x)*
- Define *t*(x) = *b(x)* – *r(x)*
- Transmit *t*, the bit string corresponding to *t(x)*
- Let *t'* represent the bit stream the receiver gets and *t'(x)* the associated polynomial. The receiver divides *t'(x)* by *g(x)*
  - If there is a 0 remainder *t* = *t'*
  - Otherwise there was an error

Polynomial Division

- Like conventional polynomial division but we use modulo 2 arithmetic
  - Addition
    - 0 + 0 = 0
    - 1 + 0 = 1
    - 0 + 1 = 1
    - 1 + 1 = 0
  - Subtraction
    - 0 – 0 = 0
    - 1 – 0 = 1
    - 0 – 1 = 1
    - 1 – 1 = 0
  - Modulo 2 addition and subtraction correspond to XOR

Example of Modulo 2 Polynomial Division

$$
\begin{array}{rl}
 & X^6 + \quad X^4 + \quad X^2 + x \\
\hline
X^3 + x + 1 \;\big|\; X^9 + \quad X^6 + \quad X^3 & \\
\quad X^9 + X^7 + X^6 & \\
\hline
\quad X^7 + \quad X^3 & \\
\quad X^7 + \quad X^5 + X^4 & \\
\hline
\quad X^5 + X^4 + X^3 & \\
\quad X^5 + \quad X^3 + X^2 & \\
\hline
\quad X^4 + \quad x^2 & \\
\quad X^4 + \quad X^2 + x & \\
\hline
\quad x & \\
\hline
\end{array}
$$

CRC Example

- Can use synthetic division by substituting a bit string for the polynomial
  - $x^3 + x + 1 = 1011$
  - $x^9 + x^6 + x^4 + x^2 = 10010\mathbf{101}00$
- Get CRC *i(x)*=1001001, with generator *g(x)*=1011

CRC Example

```
                    1 0 1 0 1 1 0
          ┌─────────────────────────
1 0 1 1   │ 1 0 0 1 0 0 1 0 0 0
        - 1 0 1 1
          ---------
            0 1 0 0
              1 0 0 0
            - 1 0 1 1
              ---------
                0 1 1 1 0
              - 1 0 1 1
                ---------
                  1 0 1 0
                - 1 0 1 1
                  ---------
                      0 1 0
```

- Add the remainder to the original *i(x)*
- Message with checksum is 1001001010
  - Data = 1001001
  - Remainder = 010
  - Message = 1001001010

Checking the CRC

- Divide $c(x)$=1001001010 by $g(x)$ = 1011
- Divide the received codeword $v(x)$ by $g(x)$, *without* extending by 4 zeros

Correction from handouts - Checking the CRC

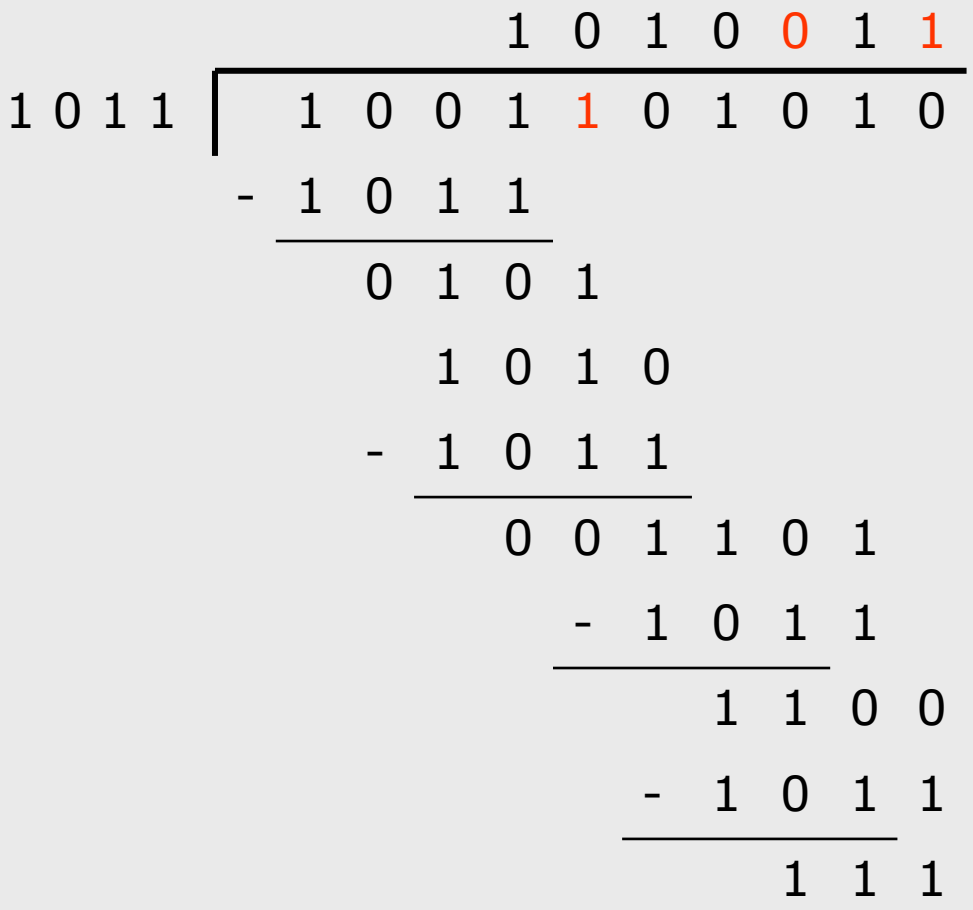
```
                1 0 1 0 1 1 0
          ┌─────────────────────
1 0 1 1   │ 1 0 0 1 0 0 1 0 1 0
        - 1 0 1 1
          ---------
          0 1 0 0
            1 0 0 0
          - 1 0 1 1
            -------
            0 1 1 1 0
              - 1 0 1 1
              ---------
              1 0 1 1
            - 1 0 1 1
            ---------
                0 0 0
```

- Remainder = 0
- Message is Correct
- The quotient is discarded; only the fact that the remainder is zero or non-zero is important

Checking the CRC

- Now assume the received codeword
  - $v(x)$ = 1001101010
- Here $v(x) = c(x) + e(x)$; where $e(x) \neq 0$; $e(x)$ = 0000100000

Checking the CRC



- Remainder is non-zero – checksum failed

Construction of a generator polynomial

- A single bit error has $e(x) = x^i$. If $g(x)$ has two or more terms it will never divide $e(x)$ and all single-bit errors will be detected
- Two isolated single bit errors have $e(x)=x^i (x^k + 1)$. Find $g(x)$ which does not divide $x^k + 1$, for k up to message length. Use a computer search; for example $x^{15} + x^{14} + 1$ does not divide $x^k + 1$ for $k < 32768$
- If there are an odd number of bits in error, then $e(x)$ has an odd number of bits. As no polynomial with an odd number of bits has $(x + 1)$ as a factor, make $g(x) = (x+1)(\ldots\ \ldots)$

Construction of generator polynomial

- A code with *r* check bits will detect all burst errors of length < *r*
- If the burst has length *r* + *1*, *r(x)* will be zero only if *r(x)* = *g(x)*, and with the probability of $1/2^{r-1}$
- For longer bursts the probability of an undetected error is $1/2^{r}$

Standard CRC polynomials

- These are all predefined and are agreed between sender and receiver
  - CRC-12 $x^{12}+x^{11}+x^3+x+1$ Old banking etc codes (6 bit)
  - CRC-16 $x^{16}+x^{15}+x^2+x+1$ North American SDLC (8 bit)
  - CRC-ITU $x^{16}+x^{12}+x^5+x+1$ ITU standard HDLC (8 bit)
  - IEEE 802 $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$ IEEE 802 LAN standards and others
  - ATM HEC x^8+x^2+x+1 ATM cell headers
  - ATM AAL3/4 $x^{10}+x^9+x^5+x^4+x+1$ some ATM traffic
- The CRC-16 and CRC-ITU checks can detect
  - All bursts ≤ 16 bits
  - All odd-bit errors
  - 99.998% of error bursts exceeding 17 bits

Misconceptions

Alleged disadvantage	Comment
A disadvantage of CRC-16, Fletcher and 1s complement sums is that they cannot correct an error	They are not designed to correct errors, but give good error detection
A 1s complement check can detect only a single-bit error	A 16-bit additive checksum is about as good as a 10-bit CRC
A disadvantage of CRC-16 is that you must send also the generator polynomial	The generator is agreed between the sender and receiver and is not transmitted

Misconceptions

Alleged disadvantage	**Comment**
A disadvantage of CRC-16 is that its generator is difficult to make	It is hard to make, but is then defined in the standard
Some of them are inefficient because we must send the checksum with the message	All of them must send the checksum with the message!

Required Reading

- Understanding Data Communications and Networks
  - Pages 231 - 240