COMP3721 Week Seven Lab Synopsis

The Data-Link Layer

- whereas the Physical layer was concerned about getting bits onto a link, the Data Link layer is concerned with:
  - packetization of the bits – we don't discuss this in detail, but from this layer up we tend to group bits into groups called frames (at the data link layer) or packets (at the Network layer and above).
  - line discipline – access to the channel. This refers to how a sender gets their bits onto the line. ENQ/ACK and Poll/Select were covered in lecture, but we will spend more time (in lecture and lab) discussing contention-based schemes commonly used in LANs (such as CSMA/CD).
  - Error Control – the Data-link layer generally adds some measure of reliability to data transfer by adding error checking in some form.

Error Control

- Error control involves adding redundancy bits to the data transmitted to allow for either:
  - Forward Error Correction (FEC), or
  - Automatic Repeat ReQuest (ARQ) error control

Forward Error Correction

- Additional (parity) bits are added to allow the receiver to **correct** errors when they occur
  - in general, for this to be effective across real channels requires a large number of redundant bits be provided
    - this is expensive (in terms of bandwidth) and is generally only considered effective (cost-effective) when
      - the error rate is high
      - there is no means of communications back to the transmitter to indicate errors (i.e. CD-ROM, digital TV signal)
    - Examples of FEC codes: Hamming, Reed-Solomon
      - You are responsible for understanding Hamming code. Note that the current textbook labels the rightmost bit as least significant (opposite of Tanenbaum from architecture). Any answers for this course must be stated in this form (least significant bit to the right).

Automatic Repeat Request (ARQ) and Error Detection

- The most common form of error control involves adding redundancy bits for **error detection** rather than correction

- When an error is detected, the receiver asks the transmitter to retransmit the packet in error
- Several different forms of error detection are commonly used:
  - Simple parity
  - Two-dimensional parity – be sure to review this in the textbook
  - Cyclical Redundancy Checks (CRC)
  - Checksums
- While some of the above codes may have some error correcting capability, it is generally much more limited than their error detecting capability and they are thus more often used in this latter context.

Cyclical Redundancy Checks (CRC)

- CRC is an error detecting code based on a different algebraic system, GF(2) or Modulo-2 math.
  - numbers are expressed as polynomials
    - $x^6 + x^5 + 1 \quad = 1x^6 + 1x^5 + 0x^4 + 0x^3 + 0x^2 + 0x^1 + 1x^0$
      $= 1\ 1\ 0\ 0\ 0\ 0\ 1$
    - the *order* of a number is the order of its polynomial. For example, the above value has an order of 6.
  - addition and subtraction are simply the XOR operator. Note how this makes sense as subtraction should 'undo' addition and XOR undoes itself.
    - no carries or borrowing

+/-	***0***	***1***
0	0	1
1	1	0

  - Given two numbers (polynomials), A divides into B (not necessarily evenly) if the order of A is less than or equal to B. This is counter-intuitive in some cases; for example, 10001 divides into 10000 once (as they are both order 4 terms) with a remainder of 1.
- As with other error control systems, parity bits will be calculated and added to the message to form a *codeword*.
- The parity bits in the case of CRC are calculated such that the codeword is evenly divisible by a number known as the generator.

- Start with an example in base 10 using normal algebra to understand the idea:
  - Generator, G(x) = 9.
  - Message, M(x) = 73
  - Divide G(x) into M(x):

```
                     ____8
Generator --> 9 ) 73 <--Message
                  72
                   1 <--Remainder (CRC)
```

  - The codeword by definition must be divisible by the generator – here then we could create a codeword by subtracting the remainder from the message.
    - Codeword, C(x) = M(x) – CRC = 73 – 1 = 72
    - This is problematic in the above case as the message is altered – we would like the message bits to be separate of the CRC or parity bits.
    - In the actual CRC calculation this problem is avoided by padding the message with enough space to hold the remainder. The principle is essentially the same as we have seen above though.
- A proper CRC calculation:
  - G(x) = $x^5 + x^4 + 1$ = 1 1 0 0 0 1
    - Given the algebraic system used for CRC calculations, the remainder can never exceed *n* bits for an order *n* generator.
    - Here the generator is order 5, then the remainder cannot be more than 5 bits
      - **In calculating the codeword then, the message bits will be padded with 5 zeros (*n* zeros in the general case)**
      - **This way the message bits will be unaltered by the subtraction**
  - M(x) = $X^7 + X^1 + 1$ = 1 0 0 0 0 0 1 1

- Divide:

```
              11111000
110001 ) 1000001100000 <--M(x) padded with 5 zeros
         110001
          100011
          110001
           100101
           110001
            101000
            110001
             110010
             110001
                 11000 <--CRC
```

- Subtract the CRC from the padded message to create the evenly-divisible codeword:

```
        1000001100000
       -        11000
C(x) =  1000001111000
```

  - Note that since subtraction is just XOR, and the padding bits are all zeros, the above calculation is identical to simply replacing the padding bits with the CRC bits.

- The above calculation would occur *on the transmitter*. The codeword generated, C(x), is transmitted to the receiver.
- During transmission, some bits may get flipped (an error). These errors can be described by a number, similar to the codeword.
  - For an error, a 0 in a particular position means the bit was not flipped; a 1 indicates the bit in that position was flipped.
  - For example, the error, E(x) = 0000000100000 means that the sixth bit from the right of a 13-bit codeword was toggled.
- The received codeword, C'(x), is then just the sum of the transmitted codeword and the error from the channel (using modulo-2/XOR math):

```
  C(x) =  1000001111000
+ E(x) = +0000000100000
 C'(x) =  1000001011000
```

- The receiver can detect an error by dividing the received codeword, C'(x), by the generator, G(x).

- The expected answer, in the case where there is no error (or equivalently E(x) is all zeros), is that the remainder = 0.
- A remainder indicates that an error must have been added (C'(x) ≠C(x))
  - then the message is rejected and the receiver must ask for re-transmission
- From the example above – test G(x) | C'(x):

```
              11111001
110001 ) 1000001011000
         110001
          100011
          110001
           100100
           110001
            101011
            110001
             110101
             110001
                100000
                110001
                 10001 <-- non-zero remainder, error
```

Checksum and the Internet Checksum

- Checksum calculations are a simpler (and less effective) means of error detection.
- In general you are simply calculating a sum over a set of data words
  - the data words are a fixed-width. That is, the sum is calculated over a set of ***n*-bit** words
  - the checksum is then also *n* bits. To ensure the sum does not exceed n-bits, the calculation is normally done modulo 2^n or modulo 2^n-1

Checksum Math

- Depending on the checksum to be calculated, you will need to use any of following operations:
  - addition modulo M
    - where M = 2^n, or
    - where M = 2^n - 1
  - modulo negation: -A mod M

- When doing the checksum calculations, you may want to work in any one of the following three bases based on the following criterion:
  - decimal – addition is easy, modulo calculation is hard (relative to

other methods), negation is easy

- Modulo is calculated as follows:
  S mod P = (S/P – floor(S/P)) * P

  - Example, calculate 76 modulo 14:

    ```
    76 mod 14     = (76/14 – floor( 76/14)) * 14
                  = (5.428571429 – 5) * 14
                  = 6
    ```

- Negation: for a value *A* in a modulo *M* field:
  -A = M – A

  - Example, calculate -15 (modulo 32)

    ```
    -15           = 32 – 15
                  = 17
    ```

- binary – addition can be tricky (potential multi-column carries), modulo calculation is trivial, negation is easy
  - modulo 2^n is calculated by simply dropping any carryout from the most significant bits
    - Example:

      ```
      1101 + 1000 (mod 2⁴)  = 10101 (mod 2⁴)
                            =  0101
      ```

  - modulo (2^n-1) is calculated by adding the carry from the most significant bits back into the value
    - Example:

      ```
      1101 + 1000 (mod 2⁴-1)     = 10101 (mod 2⁴-1)
                                 =  0101 + 1
                                 =  0110
      ```

  - One's complement negation: -A (mod 2^N-1):
    - simple complement all of the bits in A: A = ~A
    - Example:

      ```
      -0110 = 1001
      ```

- hex – addition can be tricky, modulo calculation is trivial, negation can be tricky
  - modulo 2^n is calculated by simply dropping any carryout from the most significant hex digit
    - Example:

```
   AC
+ 6E
 11A (mod 2^8) = 1A
```

- modulo (2^n-1) is calculated by adding the carry from the most significant bits back into the value
  - Example:

```
   AC
+ 6E
 11A (mod 2^8-1) = 1A + 1 = 1B
```

- Negation: for a value *A* in a modulo *M* field:
  -A = M – A

  `-1B (mod 2^8-1) = -1B (mod FF_16) = FF – 1B = E3`

- The easiest solution to these calculations (and several to come in the last portion of the course) is to get a calculator that does:
  - base conversions
  - calculations in binary, decimal and hexadecimal

Simple Checksum (Aman's)

- All calculations are done modulo 2^n (two's complement)
- Checksum is simply the sum of the *m* data words:

$$Checksum = \sum_{i=1}^{m} d_i (modulo\, 2^n)$$

- The receiver calculates the checksum over the data words as well and compares it to the transmitted checksum.
  - If the checksums do not match, an error has occurred and retransmission is necessary.

- Example: calculate the checksum for the three data words {F8, B3, 2A}. Express the answer in hexadecimal.

```
n = 8                    ;all the words given are 8-bit values
m = 3                    ;three data values specified
field size = 2^n = 256   ;calculations are modulo 256

        F8      1111 1000        248
        B3      1011 0011        179
      + 2A     +0010 1010       + 42
```

```
                 1D5       11101 0101          469
             (mod 256)      (mod 256)        mod(256)
  Checksum=       D5 <--    1101 0101 <--      213
```

Internet Checksum (same as checksum in the Forouzan Textbook)

- This is the most common checksum calculation in practice. It is often simply referred to as checksum (as in the text).[1]
- For the Internet (the IP protocol), 16-bit words are used; that is, n= 16
- All calculations are done modulo $2^n - 1$ (one's complement)
- Internet checksum is calculated (by the transmitter) as the negation of the sum of the *m* data words:

$$Internet\ checksum = -\sum_{i=1}^{m} d_i (modulo\ 2^n - 1)$$

- The receiver adds up the data words as well as the checksum value
  - The sum should be zero, if so, then no error has occurred
  - If the sum is not zero, an error has been detected

- Example: calculate the Internet checksum for the three data words {F81A, B301, 2A2A}. Express the answer in hexadecimal.

  n = 16 ;all the words given are 8-bit values
  m = 3 ;three data values specified
  field size = $2^n - 1$ = 65535 ;calculations are modulo 65535

```
                  F81A    1111 1000 0001 1010        63514
                  B301    1011 0011 0000 0001        45825
                + 2A2A   +0010 1010 0010 1010      + 10794
  CS = -1 *      1D545   11101 0101 0100 0101       120133
             (mod 65535)            (mod 65535)  mod(65535)
                  D545    1101 0101 0100 0101
                    +1                     +1
  CS = -1 *       D546    1101 0101 0100 0110        54598
     =            2AB9 <--0010 1010 1011 1001<--     10937
```

Practice Problems

1. For the following set of 16-bit values: {2B1C, 11CA, EEFB, FF01}
   1. Calculate the checksum
   2. Calculate the Internet Checksum

1 For this course, the term checksum on its own will refer to the simple checksum (Aman's); the term Internet checksum will be used when referring to the more common checksum calculation.

2. You receive the following values {2B1C, 11CA, EEFB, 1234}, where the final value is the Internet Checksum calculated over the three preceding data values. Determine whether an error has occurred in transmission.
3. You receive the message 11001101 and CRC 1011. The generator used by the system is x^4+1. Determine whether an error has occurred in transmission. If so, how is the error corrected?

Answers to Practice Problems