Error Detection

Notes for COMP28411 - 2010

- Error Detection Methods
  - Checksum
  - Cyclic Redundancy Check (CRC)
- Error Correction Methods
  1. Tell the sender who may or may not react.
     - Works well when errors are rare or when correction fails.
     - On Ethernet use a 32 bit CRC for 12,000 bits = 1500 Bytes and gives quite strong protection (but not perfect).
  2. Try and correct the error at the receiver.
     - Needs extra information to aid in reconstruction of the correct message.
       - Could send message twice.
       - If both identical then OK
       - Overhead is 2N.
       - This simple scheme <u>fails to spot occurrences of the same error in both copies</u>
         - **Forward Error Correction** (**FEC**) codes do much better than this!
         - But with a significant overhead.
- Insight used by most methods:
  - XOR is equivalent to detecting a flipped bit.

```
    1010 Original
XOR
    1010 no changes
    ----
    0000 XOR zero!
```

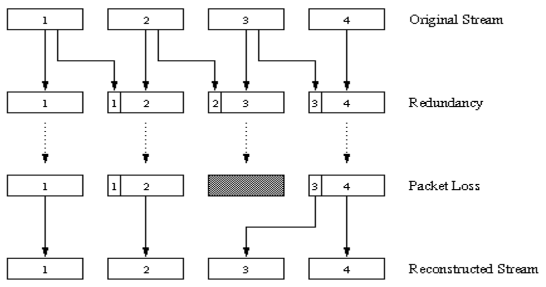
```
    1010 Original
XOR
    1110 1 bit changed
    ----
    0100 XOR non zero!
```

Forward Error Correction (FEC): Simple Scheme

- ***For every group of n chunks create redundant chunk by exclusive OR-ing n original chunks***
- ***Send out n+1 chunks, increasing bandwidth by factor 1/n.***
- ***Can reconstruct original n chunks if at most one lost chunk from n+1 chunks***
- ***Play out delay: Enough time to receive all n+1 packets***
- ***Tradeoff:***
    - ***increase n, less bandwidth waste***
    - ***increase n, longer playout delay***
    - ***increase n, higher probability that 2 or more chunks will be lost***

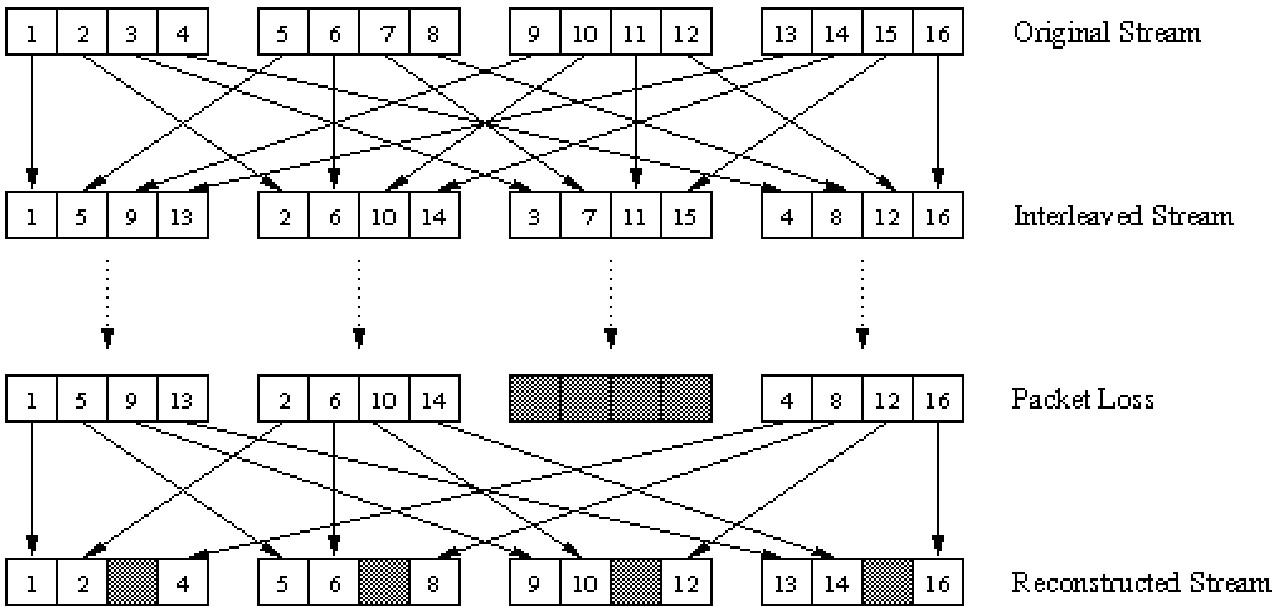
Try it: Have 3 chunks/packets : 01 10 11 do XOR so 4 chunks/packets sent. Loose 1 of the 4 packets and show can reproduce it from other 3!
My answer is at the end of this document.

- ***2nd FEC scheme***
  - ***"piggyback lower quality stream"***
  - ***Send lower resolution audio stream as redundant information***
  - ***e.g. nominal stream PCM at 64 kbps and redundant stream GSM at 13 kbps.***



- ***Whenever there is non-consecutive loss, receiver can conceal the loss.***
- ***Can also append (n-1)st and (n-2)nd low-bit rate chunk***

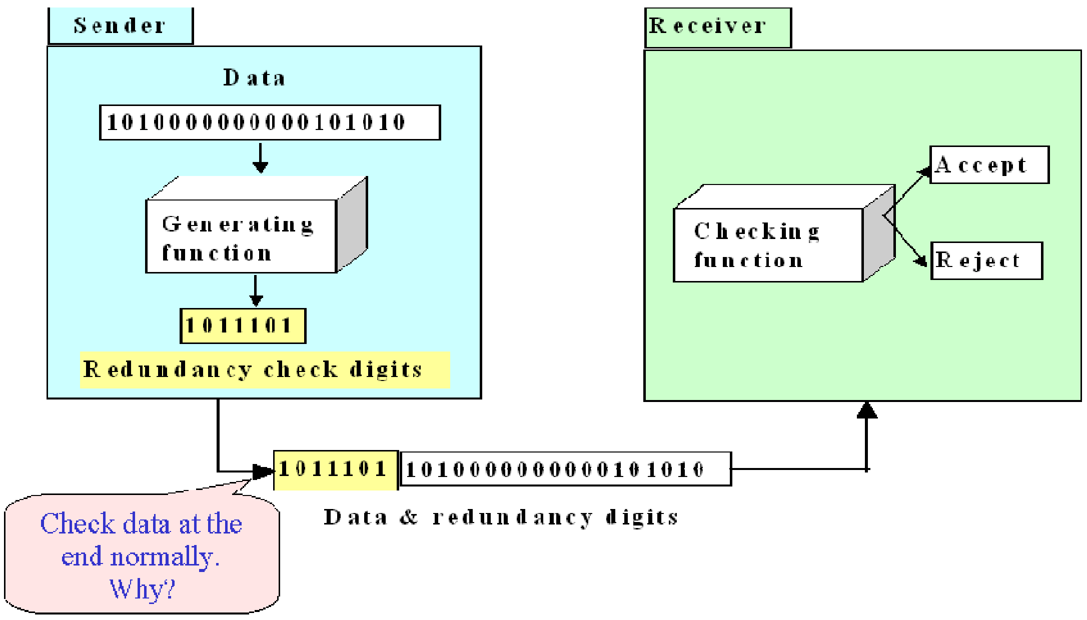
3rd FEC scheme



Interleaving

- Chunks divided into smaller units
- For example, four 5ms units per chunk
- Packet contains small units from different chunks
- If packet lost, still have most of every chunk
- No redundancy overhead, but increases play out delay!
- The standard FEC scheme used in most systems today is **Convolution Coding** with a **Viterbi Decoder**. Latest systems have a range of FEC schemes including Turbo Codes.

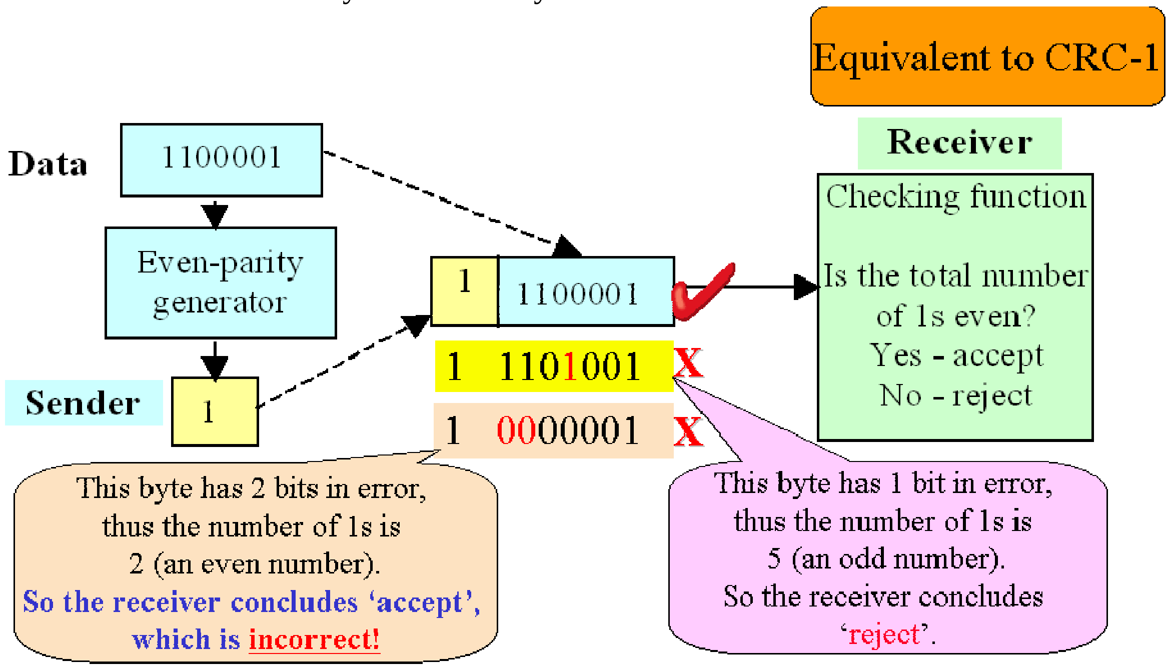
Error Detection - Schematic



- At sender side:
  - Once the data has been generated, it passes through a device that analyses it, calculates check digits and append the check digits on to the data stream.
- In transmission:
  - The data together with the check digits are now being transmitted.
- At receiver side:
  - It puts the entire stream through a checking function. If the bit stream passes the checking criteria the data portion of the data unit is accepted and the check digits are discarded. Otherwise, the data is said to be 'corrupt' and is discarded.
- The check data can go anywhere. Normally sent last as it can be calculated on the fly before sending. The receiver calculates its check on arrival then compares with the following check data in the stream. But, no reason it should not be sent earlier if it is available.

1 Bit Parity – Helps but Easily Fooled

- This shows "even" parity. Can also use "odd" parity.
    - But the story would be very similar

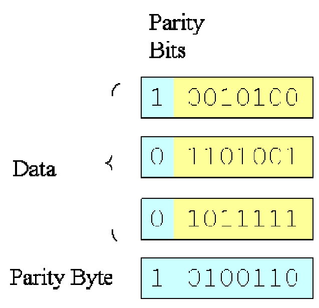

Equivalent to CRC-1
Data
1100001
Even-parity generator
Sender
1
1 1100001
1 1101001
1 0000001
Receiver
Checking function
Is the total number of 1s even?
Yes - accept
No - reject
This byte has 2 bits in error, thus the number of 1s is 2 (an even number). So the receiver concludes 'accept', which is incorrect!
This byte has 1 bit in error, thus the number of 1s is 5 (an odd number). So the receiver concludes 'reject'.

This is a simple scheme that was used on serial connections using RS232 for many years. It is fairly easy to fool if, for example, clusters of errors occur. But if errors are rare it can work well and has a fairly low overhead.

If only 1 error occurs we cannot correct it as we have no idea where the error occurred!

2D Parity

- Use all rows and columns to generate parity.
- Can catch all 1-, 2-, 3- and some 4-bit errors.
- N X M (3 * 7) bits need N + M + 1 (11) parity bits.



Is it using Odd or Even parity?
Are there sny errors?
If so, can you fix them?

Much more powerful than 1 bit parity and offers considerable protection but the overheads are quite large

Using 2D parity, the receiver can detect errors in the parity bits themselves. If the parity bits are all correct or can be corrected, then the 2D nature allows the location of errors to be determined in many cases and therefore the data can be corrected.

The IP (Internet) Checksum

- This is used by IP, TCP, UDP, … above/after the protection given by Data link and Physical layer mechanisms.
- It is relatively weak but is quick and easy to calculate.
- Works on 16 bit integers (pairs of bytes) to generate a 16 bit checksum.
- Simply add the data broken up into a sequence of 16 bit values using **1s-compliment** arithmetic.
  - A carry out from the most significant bit must be added to the result.
  - -ve numbers are simply the compliment of +ve numbers.
  - `+8 = 1000 -8 = 0111`

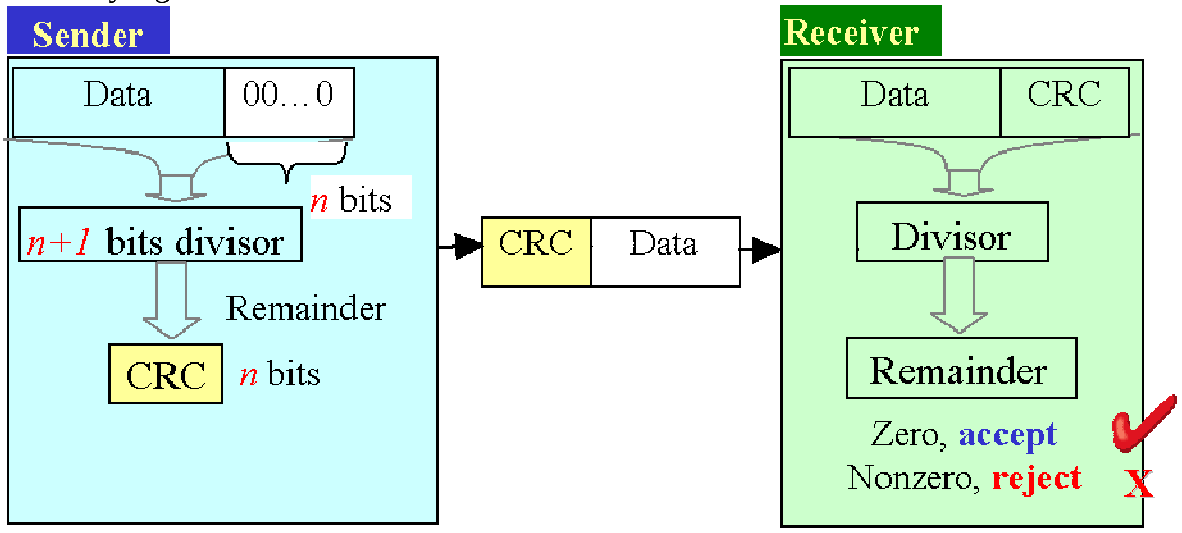
```
   -7    11000        (+7 = 00111)
 + -3    11100        (+3 = 00011)
 ====    =====
  -10 (1)10100 (but = -11)
                so add the carry 10101 = -10
          ^
           Carry bit!
```

- Calculating the checksum is a simple iterative process.
- It can be done incrementally which is very useful for changes needed for e.g.:
  - Network Address Translation (NAT)
  - Time To Live (TTL)

  For these we need to remove some packet fields and then replace the removed fields with new values so we subtract from the checksum the removed portion and then add the inserted portion.
  - To make changes to the checksum:
    - **We simply subtract the sum of the “from” section of a packet.**
    - **Then add the sum for the “to” (replacement) section of the packet.**
    - **Thus, changes are simple which is good.**
    - **But hackers can also make changes leaving no way for their change to be detected! 🡸 Security hole!**
- Adds only 16 bits for any length message.
- Can be fooled by, for example a pair of single bit errors:
  1. Increments a word by an amount
  2. Decrements another word by the same amount.
- OK to use because errors are rare, is a secondary defence.
  - Link-Layer usually provides more robust protection if needed!

Cyclic Redundancy Check (CRC)

- Typically used as primary defence mechanism in the Link-Layer. Backed up by e.g. an IP checksum.

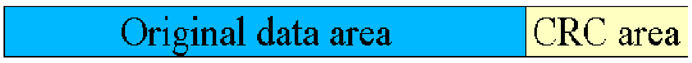


CRC32 – How good is it?

"CRC's are based on polynomial arithmetic, base 2.

CRC-32 is a 32-bit polynomial with several useful error detection properties:

- It will detect all errors that span less than 32 contiguous bits within a packet and all 2-bit errors less than 2048 bits apart.
- It will also detect all cases where there are an odd number of errors.
- "For other types of errors, if they occur in data which has uniformly distributed values, the chance of not detecting an error is 1 in 2^{32} ."
  From: "Performance of Checksums and CRC's over Real Data", Jonathan Stone, Michael Greenwald, Craig Partridge and James Hughes, IEEE/ACM TRANSACTIONS On Networking, Vol. 6, No. 5, October 1998, pp 529-543.
- Divisors are usually internationally standardised, and use polynomial representation, e.g.
  - 110101 => $1x^5 + 1x^4 + 0x^3 + 1x^2 + 0x^1 + 1$ or
    - We can leave out the zero terms giving 110101 => $1x^5 + 1x^4 + 1x^2 + 1 = G(X)$
  - Get them from a book – known good choices!

- Sender: 

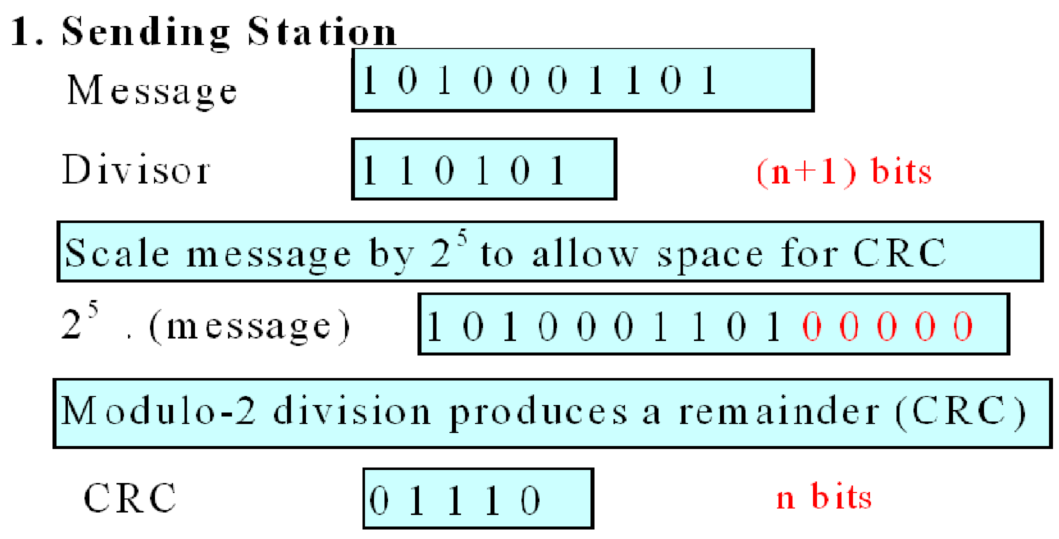
  - A string of *n* 0s is appended to the data unit. The number *n* is one less than the number of bits in the pre-determined divisor, which is *n+1* bits. (6 bit divisor so 5 in example). **This is multiplication.**
  - The newly elongated data unit is divided by the divisor using a process called modulo-2 division (or XOR operation).

  - The *n* bits remainder (i.e. the CRC code) replaces the appended 0s at the end of the data unit, and is transmitted with the data.
    - This is achieved by **XOR** with the added zeros ≡ subtract
- Receiver:
  - The data arrives together with the CRC code. The receiver divides the whole string using the same divisor and the same operation.
  - **If there is no error in the stream, the CRC checker yields a <u>remainder of zero</u>.**
  - **Otherwise, if the <u>remainder is not zero</u>, the check fails and the data is rejected.**
- The theory of this detection technique is straightforward.
- The only complexity is the **Polynomial Arithmetic Modulo-2** division.

Special CRC Polynomial Arithmetic Rules

- All coefficients are either **0** or **1**.
- The sum **A ÷ B** or **A / B** can be calculated if the length (degree) of **B** is less than or equal to the length (degree) of **A**.
  - So, for example,
    - Two 6 bit values can always be divided.
    - A 6 bit value can be divided by any 5 bit value.
    - But a 5 bit value cannot be divided by a 6 bit value!
- The **remainder** (modulo - after integer division) **of A ÷ B** can be calculated by **subtracting B from A**. So **A mod B ≡ B – A**.
- We can do this by simply taking the XOR of A and B.
- **1001** mod **1101** = **0100**
  - (**9** / **13** = **0** remainder **4** – do not care about/need the **0**)
- **1101** mod **1001** = **0100**
  - (**13** / **9** = **1** remainder **4** – do not care about/need the **1**)

Now a worked example



1. Sending Station

Message 1 0 1 0 0 0 1 1 0 1

Divisor 1 1 0 1 0 1 (n+1) bits

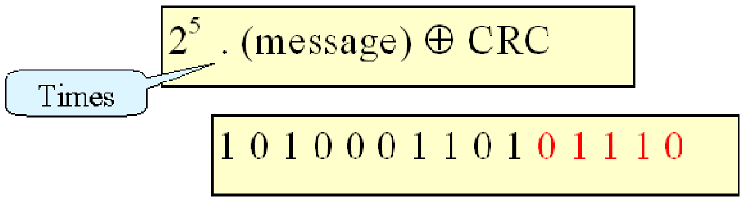
Scale message by 2^5 to allow space for CRC

2^5 . (message) 1 0 1 0 0 0 1 1 0 1 0 0 0 0 0

Modulo-2 division produces a remainder (CRC)

CRC 0 1 1 1 0 n bits

2. Transmission

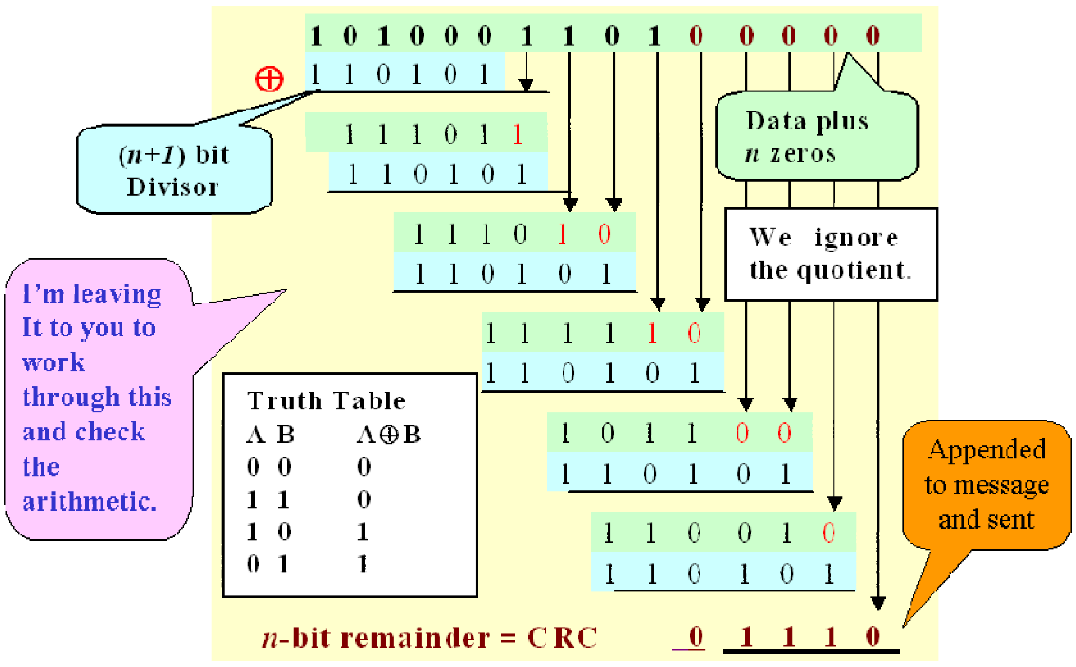


3. Receive Station

Modulo-2 division produces a remainder (check)

Check 0 0 0 0 0

Doing the sums by hand on paper (Transmitter):



Truth Table	
A B	A⊕B
0 0	0
1 1	0
1 0	1
0 1	1

I recommend you work through this convincing yourself it is correct. It has been a past favourite for examinations.

The first bit to get you started:

We have a 6 bit polynomial divisor so we tae the first 6 bits of the message to XOR with the divisor.

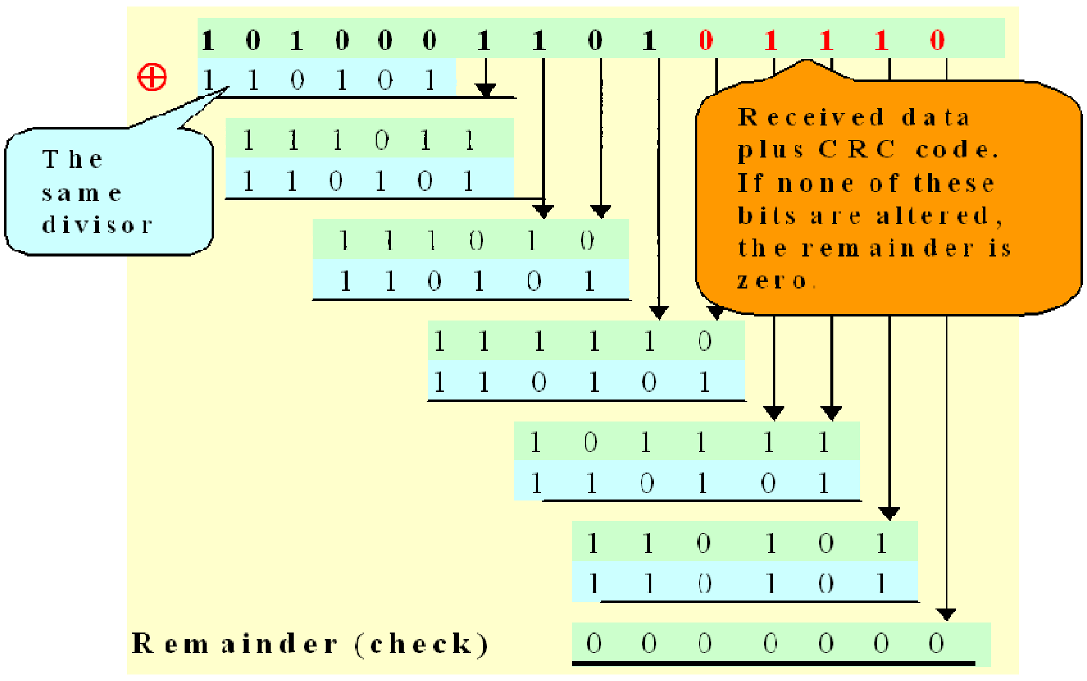
```
101000
```
– from message

```
110101
```
– Divisor

```
======
```

011101 – The answer. Note, we drop the leading zero as it plays no useful part if the resulting calculations and has no effect on the answer!

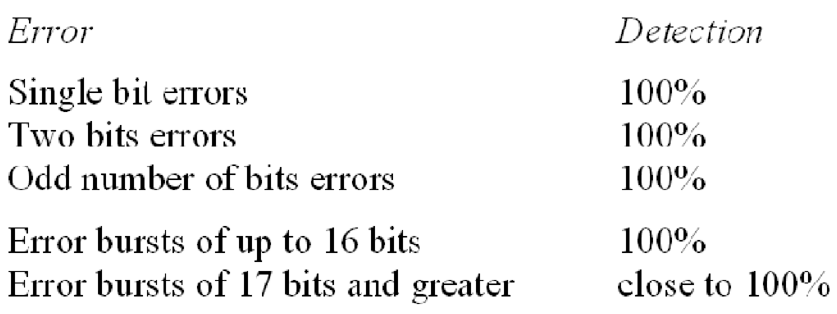
Doing the sums by hand on paper: (receiver):



Again I suggest you work through this to check I have got it right!

CRC is very effective at detecting errors. We do not often use it to correct errors but it can be used for this to some extent. However, we try and ensure that very few frames with errors are passed up to the Data-link layer for checking from the Physical layer. The Physical layer often has strong data protection methods in hardware which make local corrections. These local corrections, however, can allow incorrect data through in some cases so the CRC protects against this but it is not perfect! – nothing is!

- An example real CRC is the CRC-CCITT – $x^{16} + x^{12} + x^{5} + 1$



Error	*Detection*
Single bit errors	100%
Two bits errors	100%
Odd number of bits errors	100%
Error bursts of up to 16 bits	100%
Error bursts of 17 bits and greater	close to 100%

- This is pretty good for a 16 bit redundant value.

Some Common Polynomials for Reference

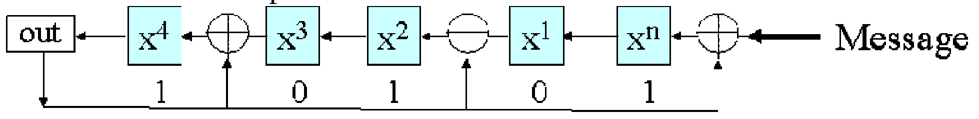
CRC	C(x)
CRC-8 ATM	$x^8 + x^2 + x^1 + 1$
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-CCITT	$x^{16} + x^{12} + x^5 + 1$
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4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

- What binary pattern does each polynomial represent? Fill in the empty boxes.
- I think the first box should be: $1\ 000\ 0111_2 = 107_{16}$

CRC	C(x)	In Binary/Hex
CRC-8 ATM	$x^8 + x^2 + x^1 + 1$	
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$	
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + 1$	
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 Hardware

- Very simple – shift register and XOR gates.
- Start with all zero's.
- An XOR gate is positioned in front of bit n if xn is a term in polynomial. For 1-10101 in our example:



- Each shift, shifts a bit out of the register from the most significant bit (MSb).
- When we shift out a 1, we exclusive-OR the polynomial with the value in the register.

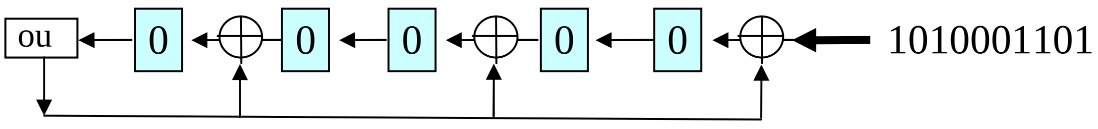
- Because our polynomial (the magic number) always contains a high-order bit, which always forces the shifted-out bit back to a logical 0, we need not actually operate on the high-order bit.
    - There is therefore no x^5 XOR.
- So only zeros shift out, keeping the mod 2 polynomial remainder in the register.

See: http://www.macs.hw.ac.uk/~pjbk/nets/crc/ for an example.

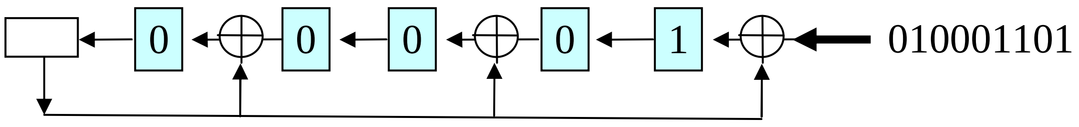
A short demo of the CRC hardware in action.

This is new December 2009 so please report any mistakes I have made to me.
Also, I have not taken it through to completion; you should try and do it.

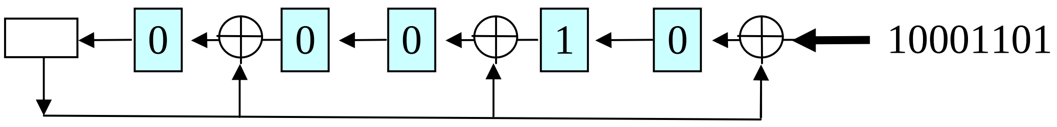
Uses the same worked example as earlier.



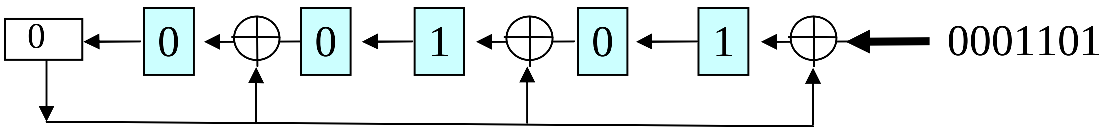
Step 1: Transmit leading 1 and shift result of XOR 1 with 0 (from register RH position) = 1into register



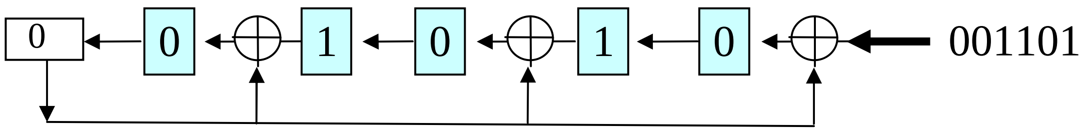
Step 2: Transmit 1 + 0. Shift result of XOR 0 with 0 = 0 into register.



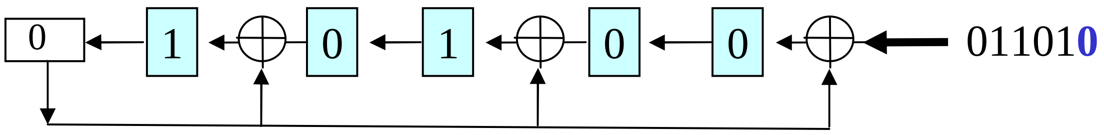
Step 3: Transmit 1 + 0 + 1. Shift result of XOR 1 with 0 = 1 into register.



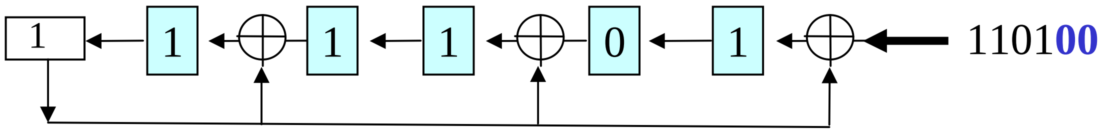
Step 4: Transmit 1 + 0 + 1 + 0.
Shift result of XOR 0 with 0 = 0 into register.



Step 5: Transmit 1 + 0 + 1 + 0 + 0.
Shift result of XOR 0 with 0 = 0 into register.



Step 5: Transmit 1 + 0 + 1 + 0 + 0 + 0.
Shift result of XOR 0 with 0 = 0 into register.



Step 6: Transmit 1 + 0 + 1 + 0 + 0 + 0 + 1.
Shift result of XOR 0 with 1 = 1 into register. Also XOR the other two shift positions with 1.

Now I suggest you check my reasoning, I did it late at night! Also, carry on the calculation so you know that it operates correctly. You can easily see the mapping from polynomial to hardware.

Summary

- We can detect most transmission errors at the receiver quite cheaply using CRCs.
- It would be even nicer to correct the errors but when errors are rare it is not worth the overheads.
- When errors are common, we cannot prevent all errors.
- We can use stronger detection and correction codes.
- We send the information to fix the data with the data.
- There is therefore no re-transmission provided we can fix the received data. Commonly use **convolution codes** and **Viterbi decoder**.

- Called **Forward Error Correction** (**FEC**). Just mentioned in this course so you are aware of it and have some very basic knowledge..
- Even with FEC, some frames will get through and fail the CRC check in the data link layer. We must discard these frames (Ethernet) or we might keep them to compare with re-transmissions .giving a much stronger checking mechanism because normally:
    - A re-transmission will be correct in most cases?
    - If the re-transmission is wrong, it is likely to be incorrect in different places to where the errors occurred in previous incorrect transmissions. This can make fixing the error easier.
- A reliable link level protocol must recover from lost frames.
- Transport protocols such as TCP, will recover any frames the link layer fails to recover so we have multiple layers of protection using different techniques.
- We vary the techniques for detection and correction using stronger methods for mediums likely to have more errors and weaker protection for reliably media.

Worked Simple XOR Example from near the top.

```
pkt a:      01
pkt b:  XOR 10
       ======
          11 as in both cases a zero
pkt c:      11 and a 1 per column
       ======
Gen pkt d:  00
```

Now **if loose pkt b** get 01 and 11 and 00.

```
pkt a:      01
pkt c:      11
        =====
          10
pkt d:      00
        =====
```

Recreate b: 10