5Mobile Systems transmit & receive data packets and bit-errors can occur due to noise affecting radio reception. We can try to achieve error-free packet transmission by:

1. Forward Error Correction (FEC)
2. Error detection and retransmission (ARQ)
3. A combination of 1 and 2

**Forward Error Correction**

This scheme relies on the correction of bit-errors at the receiver, based on information redundancy built into the transmission by:

* appending check bits or
* ‘coding‘ to produce larger packets.

Either block coding or convolutional coding may be used.

A FEC decoder tries to correct any bit-errors, and can check whether all bit-errors have been corrected. If all else fails, ARQ can be requested (retransmission) if bit-errors remain.

**Block & convolutional coding**

Block codes are used for *both* error detection and correction, whereas convolutional is generally used for bit-error correction.

Block codes require the whole block of data to be available before it can be coded at the *transmitter.* Then the complete block of data must be fully received at the receiver end before decoding can begin.

Convolutional coding can start as soon as a few bits are available, and can go on uninterrupted. A convolutional decoder can start producing an *error-corrected bit-stream* once around 50 bits have been received. Both processes (in principle) can continue for as long as the transmission and receipt of data is occurring.

A simple block coding idea: parity

* 1010 has even parity as the number of ones is even. XOR on 1,0,1 and 0 is 0.
* 1011 has odd parity, and XOR on the digits is 1.

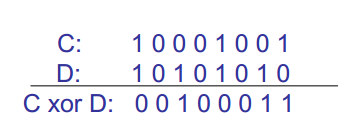
A transmitter can append the ‘parity bit’ to a 4-bit number to force parity to always be even, eg

1010**0** or 1011**1.**

Therefore the receiver calculates the parity of received messages using the **XOR of 5 bits**. If the parity is odd, a bit-error must have occurred somewhere within the 5 bits. If the parity is even, the data *may* be correct, or there may be an even number of bit-errors.

We could do the same but however test for odd parity instead, with even parity denoting a bit-error occurred.

**Hamming distance** is the number of bits that are different between two binary numbers. Obtained by XOR-ing & counting the number of 1s.

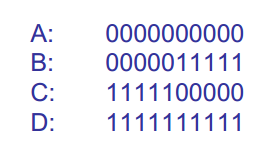


Here the hamming distance between C and D is 3. So clearly there are 3 bit-errors present.

Let the **minimum Hamming distance** between any 2 numbers be **d**.

Error *detection* is possible if the number of bit-errors < **d**.

Error *correction* is possible if the number of bit-errors <= **(d-1)/2**

The minimum Hamming distance is very important in error detection and correction. If we assume four binary numbers with a min Hamming distance of 5:

If we receive B with 2 bit-errors, it will be at distance of 2 from B and at least 3 from all others (similar situation with 1).

Taking the shortest distance, we can **correct** received number to B.

However for 3 or 4 bit-errors, we can **detect** that they are there *but* trying to correct by finding shortest distance to closest number will produce the wrong result.

**Hamming codes**

Assume if you are sending ‘m-bit’ messages, we will introduce r check-bits chosen to make **d** = 3. This allows the *detection* of single and double bit-errors, and the *correction* of a single bit-error. We must be aware that bit-errors can occur in check-bits as well as message bits.

Hamming codes of length 7, message length being 4 and number of check-bits being 3, are what we shall look at further. With m=4 and r=3, we get a (7,4) block code whose ‘rate’ is 4/7 (4 message bits per 7 bits sent).

Let message bits to transmit be B0 to B3.

Add 3 extra bits P0 to P2.



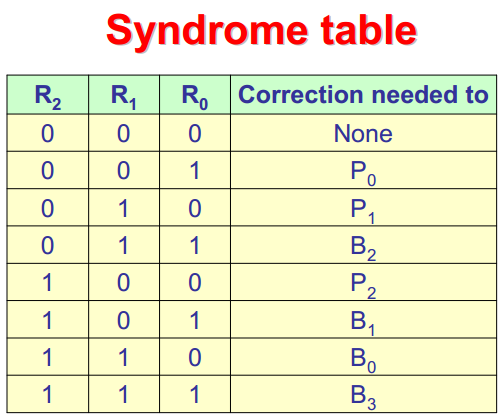
We make the following combinations all have even parity:

* B3, B2, B1, P0 (**miss out B0**) - set P0 to the XOR combo of B3, B2 and B1.
* B3, B2, B0, P1 (**miss out B1**) - set P1 to the XOR combo of B3, B2 and B0.
* B3, B1, B0, P2 **(miss out B2**) - set P2 to the XOR combo of B3, B1 and B0.

So at the receiver end we calculate the same “receiver parities” by missing out the relevant B bit.

* R0 = B3, B2,B1,P0 XORed
* R1 = B3, B2,B0,P1 XORed
* R2 = B3, B1,B0,P2 XORed

If no bit-errors, R0, R1 and R2 will be 0. We can check if either B0 up to B2 are in error if the other 2 receiver parities are 1. Eg. if B0 is in error, R1 & R2 will be 1, etc.



**Interleaving**

Radio links often suffer bursts of errors. Transmitting a block by column ensures only a single-bit error per codeword row.

**Cyclic redundancy check (CRC)**

Another example of a block code but only for bit-error detection.

Suppose we are transmitting a decimal number 139. We divide by a ‘generator’ 7 and take the remainder, and express it in binary. We get 19 with remainder 6, or 110 in binary.

We use 110 as the check-bits. The same division is done at the receiver, if we get a different number, a bit-error has occurred.

The ‘generator’ number 7 is agreed in advance and carefully chosen.

*However not all combinations of bit-errors can be detected by this method.*

* Any combo that adds or subtracts multiples of 7 are not detected.

**Real CRC checks and polynomials**

A decimal number may be expressed as a polynomial:

Binary numbers may be expressed as a polynomial:

10011001 =

Real CRC checks do not use normal arithmetic. They use a different way of ‘dividing’ based on XOR. It is modulo-2 /Galois field (base 2) arithmetic.

Summing or subtracting bits involves looking at each bit in each position and XORing the two.

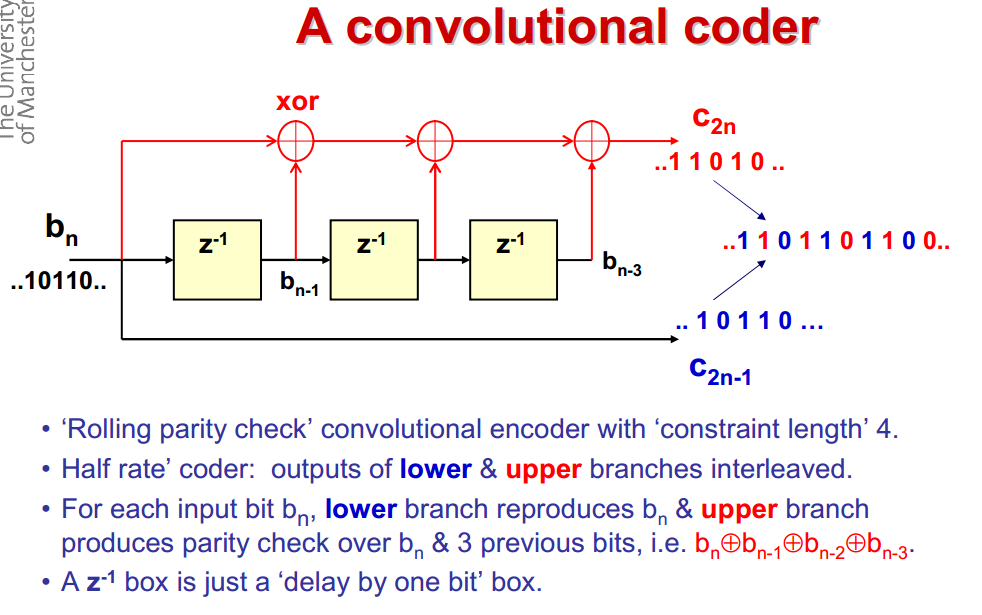
Limitations of CRC

Not all combinations of bit-errors are detectable by a CRC. Any combination of bit-errors that adds any ‘multiple’ of G(x) (our generator number) will not be detected. We can try to make this unlikely by making the order of G(x) large.

A G(x) of order **r** cause all error ‘bursts’ of length <= **r** to be detected.

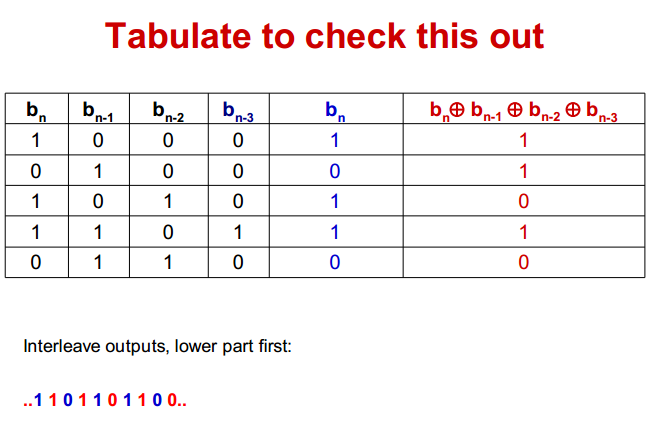
Standard CRC generators include CRC-8-ATM, CRC-16-IBM, and CRC-32-IEEE.

**Convolutional Coders**



The lower table works through bn (10110), left-to-right, and we assume missing bits are ‘0’.

**Lower** simply feeds bn through, **upper** performs XOR operations on the current bit and the 3 bits before it. Then we interleave the branches **(lower part first)**.



Decoder Strategy

The encoder generates ‘valid’ sequences where each upper bit is the XOR of all previous 3 lower bits. How ever bit-errors can make this received bit-sequence ‘invalid’.

We select the valid sequence with the *minimum Hamming distance* to the received sequence, as the error-corrected sequence. We can do this for short sequences of bits (eg 8), but it becomes infeasible for longer sequences(eg 1024).

A **Viterbi decoder** uses the [Viterbi algorithm](https://en.wikipedia.org/wiki/Viterbi_algorithm) for decoding a bitstream that has been encoded using a [convolutional code](https://en.wikipedia.org/wiki/Convolutional_code).

Conv. coders are used widely in mobile systems, and Viterbi efficiently decodes them.

Viterbi decoders use ‘soft decisions’, instead of just 1 and 0.

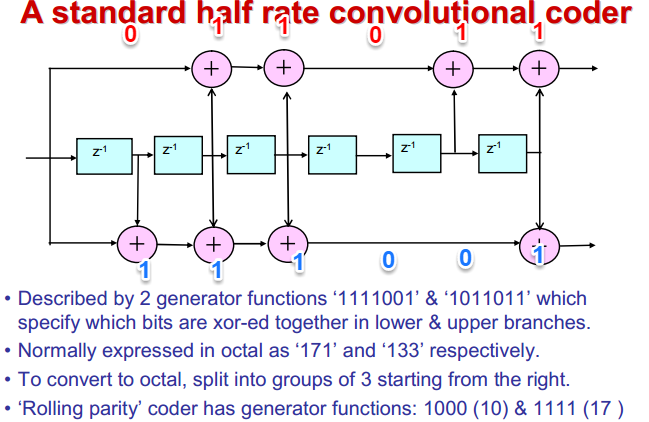
0.25 = ‘probably 0’, 0.5 = ‘don’t know’, 0.75 = ‘probably 1’

‘Rolling parity’ conv. coder is:

**Systematic** - as original code appears in coder output.

Of **constraint length 4** as there are three z^-1 output boxes.

The rolling parity coder was *presented for simplicity, this is the conv. coder used widely in practice.*



1 **011 011 -> 133 in octal**

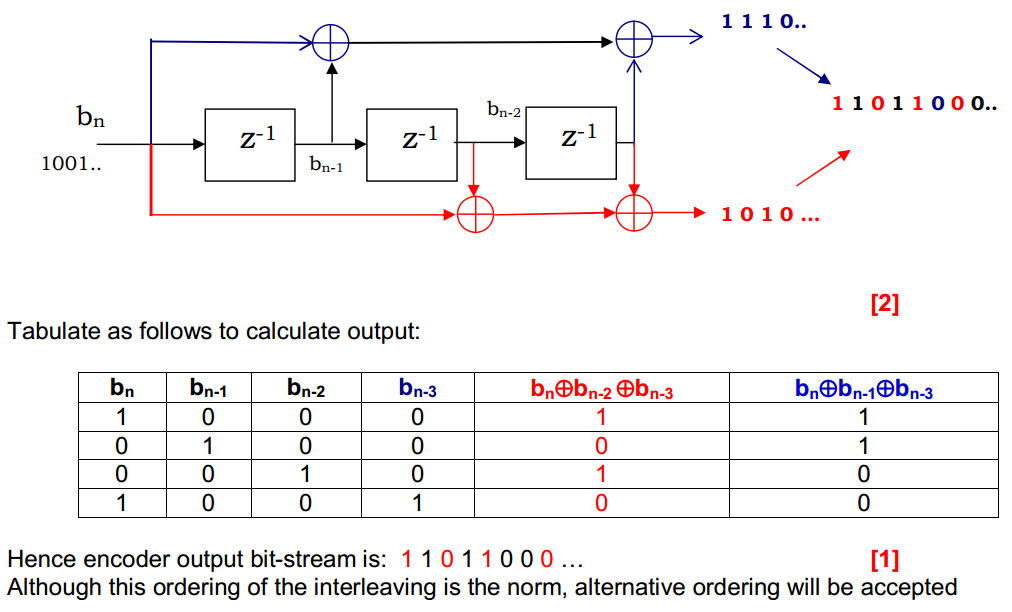
1 **111 001 -> 171 in octal**

The ‘generator’ functions denote when perform an XOR, 1 means XOR, 0 means do not (see above diagram). On the **red** path we take bn and XOR it with , not , as we encounter a ‘0’ (don’t XOR) and then a ‘1’ (XOR). After each XOR operation on a branch, we take the result and then use that as one of the inputs for the next XOR operation we encounter, and take the other input as the next bit in the input sequence of bits.

Funnily enough, this example is better explained in a *model answer* for last year’s past paper:

(obviously spoilers ahead if you want to attempt the question yourself first).

**NB. The diagram is wrong, blue output should be 1110 as in the table.**



Coder also of constraint length 4. (3 boxes).

Here we can see the **upper blue** branch performs XOR on and , then takes that result and performs XOR on the result on (because of the gap between the XOR operations).

Similar with the red, lower branch. Comparing this coder with the simpler, earlier coder may help your understanding.

**Advantages of FEC for mobile systems**

Use of FEC in cellular mobile systems increases energy efficiency and effectiveness.

Transmitting at a higher power is one way of making sure a signal is received with fewer errors, but high power signals cause and carry further interference over a wider range, and also quickly depletes a battery.

Solution is to reduce transmission power and deal with resulting increase in bit-error rate using FEC. Solves cellular frequency reuse problem and reduces power consumption.

**Shannon-Hartley Law**

The **Shannon–Hartley theorem** tells the maximum bit-rate at which information can be transmitted over a communications channel of a specified bandwidth in the presence of ‘additive white Gaussian noise’ and an arbitrarily small bit-error rate.

The channel capacity C is expressed as:

Where B is Bandwidth in Hz.

S/N = signal **power** / noise **power** ratio (**not** in dB).

SNR = 10 *signal-to-noise ratio* in dB.

If SNR >> 1, then we can use