# COSC2473

# Data Error Handling

**Intro and Gray Codes**

**<u>Parity and Hamming Codes</u>**

**SECDED and Templates**

# Data Error Detection: Parity

- A simple way to handle bit errors is to count the ‘1’ bits and force the correct number to always be even (or odd), by adding a parity bit if needed to force this.

- Redundant bits can be added to data, to detect errors
  - e.g. adding a parity bit to 7-bit ASCII
  - parity bit is  for ASCII MSB

- ASCII with even parity
  - total number of 1s in a (8 bit) character is even
    - ‘A’ = _100 0001,  parity 0,  code = 0100 0001
    - ‘C’ = _100 0011,  parity 1,  code = 1100 0011
- ASCII with odd parity
  - parity bit is set so total number of 1s is odd

# Parity

- Any 1 bit error will change the number of 1s and 0s, so the parity bit will no longer be correct
  - error is thus *detected*
  - but not enough information to *correct* error (which bit has been flipped?)

- Flipping a single bit of an ASCII character produces another valid ASCII character
  - there is a 1-bit ‘distance’ between valid ASCII symbols (characters)

- If we could increase the ‘distance’ (in bits) between valid symbols, then a single bit error would turn a valid symbol into an invalid symbol

# Examples of Parity Error Detection

- Assuming even parity for 5 bits, we get:

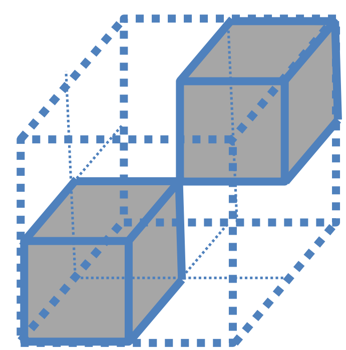
| | | | |
|---|---|---|---|
| 00000 | 10100 | 11000 | 01100 |
| 10001 | 00101 | 01001 | 11101 |
| 10010 | 00110 | 01010 | 11110 |
| 00011 | 10111 | 11011 | 01111 |

  - Notice that the three bits in the middle determine whether the parity bit matches the bottom bit, or is the opposite.
  - The parity MSB and LSB can be said to <u>frame</u> the bits in between and provide an error detection mechanism.
  - In this case a fixed frame of 3 bits is used.

- You can see how a 1 bit redundancy can improve reliability

# Data Error Correction: Hamming Distance

- The ‘bit distance’ between valid symbols in a code is called the Hamming distance
- e.g. say a code only had the symbols 000 and 111
  - everything else is illegal
  - so the Hamming distance is 3 (need 3 flips to move from one valid symbol to another)
- If 1 bit is flipped, and we receive 001:
  - we know an error has occurred
  - we know the correct code must have been 000
  - such a code allows error *correction*



- This is generalisation of Gray codes

** David Hamming was a Mathematician during WW2*

# Hamming Code

- Hamming codes are a generalisation of the parity and Gray code ideas, to cover many bits, not just one.

- A Hamming distance of 3 can correct for 1 bit error
  – if 2 bit flips are possible, and we receive 011, we don't know if it should be 111 (one bit flipped) or 000 (two bits flipped)

- For detecting a single bit error, we use the formula:

$$2^p >= m + p + 1$$

m = number of data bits

p = number of check bits

to implement a *Hamming Code.* with m data bits and p check bits

•the above example (000 and 111) would have m = 3, p = 3

•So we need 7 bits to encode 3 bits of data

# Hamming code – Check Bits

- e.g. assume we have 4 data bits, find the number of check bits in order to detect and correct a single bit error
- How many check bits for m=4 data bits?
  - try p = 2. So $2^p = 2^2 = 4$,
    - but m + p + 1 = 4 + 2 + 1 = 7 and 4 is not >= 7
  - try p = 3. So $2^p = 2^3 = 8$
    - m + p + 1 = 4 + 3 + 1 = 8
    - so 3 check bits are needed to encode 4 bits of data
- Check bits are usually interspersed with data bits in a set pattern
  - if they were all at the front (or end), we'd have to know in advance how long the original data bit string was

# Hamming code

- For example, it is common to place check bits at positions that are powers of 2 (1,2,4,8,16 etc) from LSB to MSB

| **Bit Position** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** |
|---|---|---|---|---|---|---|---|---|---|---|
| Data/Parity | D6 | D5 | P8 | D4 | D3 | D2 | P4 | D1 | P2 | P1 |

  - Note that P1,P2,P4,P8 is sometimes written as P1,P2,P3,P4 (ie powers of 2, corresponding to their bit position)
  - – This placement pattern allows a mathematical trick to be used to determine which parity bits protect which data bits
  - – Consider P1: what binary numbers have a 1 at bit position 1?
    - 1, 11, 101, 111, 1001, 1011, 1101, 1111, 10001, 10011, …
  - – Read those binary numbers as base 10, those are the bits being protected by P1
    - bit positions: 1, 3, 5, 7, 9, 11, 13, 15, 17, …

# Hamming code

- example cont.
  - Consider P2: what binary numbers have a 1 at bit position 2?
    - 10, 11, 110, 111, 1010, 1011, 1110, 1111, 10010, …
  - Read those binary numbers as base 10, those are the bits being protected by P2
    - bit positions: 2, 3, 6, 7, 10, 11, 14, 15, 18

  - Consider P4: what binary numbers have a 1 at bit position 3?
    - 100, 101, 110, 111, 1100, 1101, 1110, 1111, 10100
  - Read those binary numbers as base 10, those are the bits being protected by P4
    - bit positions: 4, 5, 6, 7, 12, 13, 14, 15, 20, …

# Hamming code

- example cont.
  - Consider P8: what binary numbers have a 1 at bit position 4?
    - 1000,1001,1010…
  - Read those binary numbers as base 10, those are the bits being protected by P2
    - bit positions: 8,9,10…

- So for example, bit position 5 (i.e. D2) is protected by both P1 and P4
  - if we were using even parity, and bit position 5 was 0, then P1 and P4 should both be 1
  - if bit position 5 was 1, then one of P1 or P4 would need to be 1

# Hamming code – Correct

- e.g. determine the single bit error-correcting Hamming code required for the data $1011_2$, using even parity
  - 1011 is 4 data bits, so from the formula we need $2^p$=8, so =p3

| **Bit Position** | **7** | **6** | **5** | **4** | **3** | **2** | **1** |
|---|---|---|---|---|---|---|---|
| Data/Parity | D4 | D3 | D2 | P4 | D1 | P2 | P1 |
| Data | 1 | 0 | 1 | | 1 | | |
| P1 | 1 | | 1 | | 1 | | ? |
| P2 | 1 | 0 | | | 1 | ? | |
| P4 | 1 | 0 | 1 | ? | | | |

  - P1 checks bits 1,3,5,7 = ?111. For even parity, P1 must be 1
  - P2 checks bits 2,3,6,7 = ?101 P2 =must be 0
  - P4 checks bits 4,5,6,7 = ?101 P4 must be 0

- So the final code for $1011_2$ is $1010101_2$

# Hamming code – Error in bit

- e.g. assume we have a Hamming 4-bit (data) code with even parity. We have received code $1000100_2$.
  - find and repair the error (if any)

| Bit Position | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| Data/Parity | D4 | D3 | D2 | P4 | D1 | P2 | P1 |
| Data | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| P1 | 1 | | 0 | | 1 | | ? |
| P2 | 1 | 0 | | | 1 | ? | |
| P4 | 1 | 0 | 0 | ? | | | |

  - P1 checks bits 1,3,5,7 = **0**101 so P1 is correctly = 0.. Correct.
  - P2 checks bits 2,3,6,7 = **0**101 so P2 is correctly = 0. Correct.
  - P4 checks bits 4,5,6,7 = **0**001 so having P4 = 0 . Error
    - error must be one of bits 4,5,6,7
      - bit 5 is correct (via P1 being correct).
      - bits 6 & 7 must be correct (as P1 and P2 are correct)
      - error must be bit 4 (a check bit) – corrected code is 1001100
- correct data = $1001_2$

# Hamming code – Error in bit

- e.g. assume we have a Hamming 4-bit (data) code with even parity. We have received code 1000101.
  - find and repair the error (if any)

| Bit Position | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| Data/Parity | D4 | D3 | D2 | P4 | D1 | P2 | P1 |
| Data | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| P1 | 1 | | 0 | | 1 | | ? |
| P2 | 1 | 0 | | | 1 | ? | |
| P4 | 1 | 0 | 0 | ? | | | |

  - P1 checks bits 1,3,5,7 = **1**101 and P1=1. Error
  - P2 checks bits 2,3,6,7 = **0**101 so P2 is 0 Correct
  - P4 checks bits 4,5,6,7 = **0**001 so P4 is 1 Error
    - P1 and P4 both check bits 5 and 7
    - but P2 also checks bit 7, and P2 is OK
    - error must be bit 5 – corrected code is 1010101
    - corrected data = 1101
- Due to where the check bits are positioned, there is a shortcut
  - Simply add the bit value for the P's in error. More later…..

# Hamming code – Error in Check bit

- e.g. assume we have a Hamming 4-bit (data) code with even parity. We have received code 1010100.
  - find and repair the error (if any)

| **Bit Position** | **7** | **6** | **5** | **4** | **3** | **2** | **1** |
|---|---|---|---|---|---|---|---|
| Data/Parity | D4 | D3 | D2 | P4 | D1 | P2 | P1 |
| Data | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| P1 | 1 | | 1 | | 1 | | ? |
| P2 | 1 | 0 | | | 1 | ? | |
| P4 | 1 | 0 | 1 | ? | | | |

  - P1 checks bits 1,3,5,7 = **0**111 so P1 = 0. Error
  - P2 checks bits 2,3,6,7 = **0**101 so P2 = 0 Correct
  - P4 checks bits 4,5,6,7 = **0**101 so P4 is 0 Correct
    - P1 checks bits 1,3,5,7 and P1 is in error
    - P2 checks bits 2,3,6,7, and P2 is OK
    - P4 checks bits 4,5,6,7, and P4 is Ok
  - Bit 1 is the only bit not covered by the correct Ps
    - error must be bit 1 – corrected code is 1010101
- corrected data = $1011_2$

# Parity Bits - Summary

- Parity codes can be used to confirm the number of 1s in a bit pattern
  - Parity can be even or odd, which means that the number of 1s in the bit string including the parity itself, must be even or odd
  - A single bit error will invalidate the parity bit and is thus detectable
    - But not WHICH bit is in error

- Hamming introduced the idea of having a parity bit for each bit position in the bitstring
  - This means that there are at least n parity bits for an n bit stream.
  - These bit-position based parity bits can be used to isolate single-bit errors