Information Redundancy - Coding

♦ A data word with \( d \) bits is encoded into a codeword with \( c \) bits - \( c > d \)
♦ Not all \( 2^c \) combinations are valid codewords
♦ To extract original data - \( c \) bits must be decoded
♦ If the \( c \) bits do not constitute a valid codeword an error is detected
♦ For certain encoding schemes - some types of errors can also be corrected
♦ Key parameters: number of erroneous bits that can be detected as erroneous and number of erroneous bits that can be corrected
♦ Overhead:
  * additional bits required
  * time to encode and decode
Hamming Distance

♦ The Hamming distance between two codewords - the number of bit positions in which the two words differ

♦ Two words in this figure are connected by an edge if their Hamming distance is 1

Hamming Distance - Examples

♦ 101 and 011 differ in two bit positions - Hamming distance of 2
  * Need to traverse two edges to get from 101 to 011

♦ 101 and 100 differ by one bit position - a single error in the least significant bit in either of these two codewords will go undetected

♦ A Hamming distance of two between two codewords implies that a single bit error will not change one of the codewords into the other
Distance of a Code

♦ The Distance of a code - the minimum Hamming distance between any two valid codewords

♦ Example - The code with four codewords - \{001,010,100,111\} - has a distance of 2

♦ This code can detect any single bit error

♦ Example - The code with two codewords - \{000,111\} - has a distance of 3

♦ This code can detect any single or double bit error

♦ If double bit errors are not likely to happen - this code can be used to correct any single bit error

Detection vs. Correction

♦ To detect up to \(k\) bit errors, the code distance should be at least \(k+1\)

♦ To correct up to \(k\) bit errors, the code distance should be at least \(2k+1\)
Coding vs. Redundancy

♦ The code {000,111} can be used to encode a single data bit
♦ 0 can be encoded as 000 and 1 as 111
♦ This code is identical to TMR
♦ Many redundancy techniques can be considered as coding schemes
♦ Duplex - a code whose valid codewords consist of two identical data words
♦ For a single data bit - the codewords will be 00 and 11

Separability of a Code

♦ A code is separable if it has separate fields for the data and the code bits
♦ Decoding consists of disregarding the code bits
♦ The code bits can be processed separately to verify the correctness of the data
♦ A non-separable code has the data and code bits integrated together - extracting the data from the encoded word requires some processing
Parity Codes

♦ The simplest separable codes are the parity codes
♦ A parity-coded word includes n data bits and an extra bit which holds the parity
♦ In even (odd) parity code - the extra bit is set so that the total number of 1's in the (d+1)-bit word (including the parity bit) is even (odd)
♦ The overhead fraction of this parity code is 1/d

Properties of Parity Codes

♦ A parity code has a distance of 2 - will detect all single-bit errors
♦ If one bit flips from 0 to 1 (or vice versa) - the overall parity will not be the same - error can be detected
♦ Simple parity cannot correct any bit errors
Encoding and Decoding Circuitry for Parity Codes

The encoder: a modulo-2 adder - generating a 0 if the number of 1’s is even
The output is the parity signal

Parity Codes - Decoder

- The decoder generates the parity from the received data bits and compares it with the received parity bit
- If they match, the output of the exclusive-nor gate is a 1 - indicating no error has been detected
- If they do not match - the output is 0, indicating an error
- Double-bit errors can not be detected by a parity check
- All three-bit errors will be detected
Even or Odd Parity?

- The decision depends on which type of all-bits error is more probable
- For even parity - the parity bit for the all zeroes data word will be 0 and an all-0's failure will go undetected - it is a valid codeword
- Selecting the odd parity code will allow the detection of the all-0's failure
- If all-1's failure is more likely - the odd parity code must be selected if the total number of bits \((d+1)\) is even, and the even parity if \(d+1\) is odd

Parity Bit Per Byte

- A separate parity bit is assigned to every byte (or any other group of bits)
- The overhead increases from \(1/d\) to \(m/d\) \((m\) is the number of bytes or other equal-sized groups)
- Up to \(m\) errors will be detected if they occur in different bytes.
- If both all-0's and all-1's failures may happen - select odd parity for one byte and even parity for another byte
Byte-Interlaced Parity Code

♦ Example: d=64, data bits - a₆₃,a₆₂,...,a₀
♦ Eight parity bits:
♦ First - parity bit of a₆₃,a₅₅,a₄₇,a₃₉,a₃₁,a₂₃,a₁₅,a₇ - the most significant bits in the eight bytes
♦ Remaining seven parity bits - assigned so that the corresponding groups of bits are interlaced
♦ Scheme is beneficial when shorting of adjacent bits is a common failure mode (example - a bus)
♦ If parity type (odd or even) is alternated between groups - unidirectional errors (all-0's or all-1's) will also be detected

Error-Correcting Parity Codes

♦ Simplest scheme - data is organized in a 2-dimensional array
  0 0 0 1 1 1 1 1
  1 0 1 0 1 1 0 0
  1 1 0 0 0 0 1 0
  0 0 0 1 1 1 1 1
  1 1 1 1 1 1 0 0
  1 1 1 1 1 1 0 0
  1 0 0 0 1 0 0 1
  1 0 0 0 1 1 1 1

♦ Bits at the end of row - parity over that row
♦ Bits at the bottom of column - parity over column
♦ A single-bit error anywhere will cause a row and a column to be erroneous
♦ This identifies a unique erroneous bit
♦ This is an example of overlapping parity - each bit is covered by more than one parity bit
Overlapping Parity - General Model

♦ Purpose - identify every single erroneous bit
♦ d data bits and r parity bits - total of d+r bits
♦ Assuming single-bit errors - d+r error states + one no-error state - total of d+r+1 states
♦ We need d+r+1 distinct parity "signatures" (bit configurations) to distinguish among the states
♦ r parity checks generate $2^r$ parity signatures
♦ Hence, $r$ is the smallest integer that satisfies

$$2^r \geq d+r+1$$

♦ Question - how are the parity bits assigned?

Assigning Parity Bits - Example

♦ d=4 data bits - r=3 is the minimum number of parity bits - d+r+1=8 states that the word can be in
♦ Table shows a possible assignment of parity values to states - in $(a_3 \ a_2 \ a_1 \ a_0 \ p_2 \ p_1 \ p_0)$, bit positions 0, 1 and 2 are parity bits, the rest are data bits

<table>
<thead>
<tr>
<th>State</th>
<th>Erroneous parity check(s)</th>
<th>Syndrome</th>
</tr>
</thead>
<tbody>
<tr>
<td>No errors</td>
<td>None</td>
<td>000</td>
</tr>
<tr>
<td>Bit 0 (p_0) error</td>
<td>p_0</td>
<td>001</td>
</tr>
<tr>
<td>Bit 1 (p_1) error</td>
<td>p_1</td>
<td>010</td>
</tr>
<tr>
<td>Bit 2 (p_2) error</td>
<td>p_2</td>
<td>100</td>
</tr>
<tr>
<td>Bit 3 (a_3) error</td>
<td>p_0, p_1</td>
<td>011</td>
</tr>
<tr>
<td>Bit 4 (a_1) error</td>
<td>p_0, p_2</td>
<td>101</td>
</tr>
<tr>
<td>Bit 5 (a_2) error</td>
<td>p_1, p_2</td>
<td>110</td>
</tr>
<tr>
<td>Bit 6 (a_3) error</td>
<td>p_0, p_1, p_2</td>
<td>111</td>
</tr>
</tbody>
</table>
(7,4) Hamming Single Error Correcting (SEC) Code

- If only p₀ check fails - bit 0 (p₀) is in error
- p₀ check fails also when bit 4 (a₁) is in error
- A parity bit covers all bits whose error it indicates
- p₀ covers positions 0,3,4,6 - p₀ = a₀⊕a₁⊕a₃
- p₁ covers positions 1,3,5,6 - p₁ = a₀⊕a₂⊕a₃
- p₂ covers positions 2,4,5,6 - p₂ = a₁⊕a₂⊕a₃

<table>
<thead>
<tr>
<th>State</th>
<th>Errorous parity check(s)</th>
<th>Syndrome</th>
</tr>
</thead>
<tbody>
<tr>
<td>No errors</td>
<td>None</td>
<td>000</td>
</tr>
<tr>
<td>Bit 0 (p₀)</td>
<td>p₀</td>
<td>001</td>
</tr>
<tr>
<td>Bit 1 (p₁)</td>
<td>p₁</td>
<td>010</td>
</tr>
<tr>
<td>Bit 2 (p₂)</td>
<td>p₂</td>
<td>100</td>
</tr>
<tr>
<td>Bit 3 (a₃)</td>
<td>p₀, p₁</td>
<td>011</td>
</tr>
<tr>
<td>Bit 4 (a₁)</td>
<td>p₀, p₂</td>
<td>101</td>
</tr>
<tr>
<td>Bit 5 (a₂)</td>
<td>p₁, p₂</td>
<td>110</td>
</tr>
<tr>
<td>Bit 6 (a₃)</td>
<td>p₀, p₁, p₂</td>
<td>111</td>
</tr>
</tbody>
</table>

Definition - Syndrome

- Example: a₃a₂a₁a₀ = 1100 and p₂p₁p₀ = 001
- Suppose 1100001 becomes 1000001
- Recalculate p₂p₁p₀ = 111
- Difference (bit-wise XOR) is 110
- This difference is called syndrome - indicates the bit in error
- It is clear that a₂ is in error and the correct data is a₃a₂a₁a₀ = 1100
Calculating the Syndrome - (7,4) Hamming Code

- The syndrome can be calculated directly in one step from the bits \(a_3\ a_2\ a_1\ a_0\ p_2\ p_1\ p_0\)
- This is best represented by the following matrix operation where all the additions are mod 2

\[
\begin{bmatrix}
1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
a_3 \\
a_2 \\
a_1 \\
a_0 \\
p_0 \\
p_1 \\
p_2 \\
\end{bmatrix}
= 
\begin{bmatrix}
s_3 \\
s_2 \\
s_1 \\
\end{bmatrix}
\]

\[p_0 = a_0 \oplus a_1 \oplus a_3 \]
\[p_1 = a_0 \oplus a_2 \oplus a_3 \]
\[p_2 = a_1 \oplus a_2 \oplus a_3 \]

Selecting Syndromes

- Data and parity bits can be reordered so that: calculated syndrome minus 1 will be the index of the erroneous bit
- In Example - the order \(a_3a_2a_1p_2a_0p_1p_0\)
- In general - if \(2^r > d+r+1\) we need to select \(d+r+1\) out of the \(2^r\) binary combinations to be syndromes
- Combinations with many 1s should be avoided - less 1s in parity check matrix - simpler circuits for the encoding and decoding operations
Selecting Check Matrix

♦ Example: d=3 - r=3 but only seven out of the eight 3-bit binary combinations are needed
♦ Two possible parity check matrices:

\[
\begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}
\quad (a)
\]

\[
\begin{bmatrix}
0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}
\quad (b)
\]

♦ (a) uses 111 - (b) does not
♦ Encoding circuitry for (a) requires one XOR gate for p1 and p2 but two XOR gates for p0
♦ Encoding circuitry for (b) requires one XOR gate for each parity bit

Improving Detection

♦ Previous code can correct a single bit error but can not detect a double error
♦ Example - 1100001 becomes 1010001 - a2 and a1 are erroneous - syndrome is 011
♦ This indicates erroneously that bit a0 should be corrected
♦ One way of improving error detection capabilities - adding an extra check bit which is the parity bit of all the other data and parity bits
♦ This is an (8,4) single error correcting/double error detecting (SEC/DED) Hamming code
Syndrome Generation for (8,4) Hamming Code

- $p_3$ - parity bit of all data and check bits - a single bit error will change the overall parity and yield $s_3=1$
- The last three bits of the syndrome will indicate the bit in error to be corrected as before as long as $s_3=1$
- If $s_3=0$ and any other syndrome bit is nonzero - a double or greater error is detected

Example

- Single error - 11001001 becomes 10001001
- Syndrome is 1110 - indicating that $a_2$ is erroneous
- Two errors - 11001001 becomes 10101001
- Syndrome is 0011 indicating an uncorrectable error
Different (8,4) Hamming Code

- **Previous code** - calculating the additional check bit is the most time consuming in encoding and decoding.
- **Possible solution** - assign syndromes with an odd number of 1s.
- A double error will result in a syndrome with an even number of 1s - indicating an error that can not be corrected.
- Only $2^{p-1}$ out of the $2^p$ combinations used.
- An extra check bit is needed beyond the minimum for a SEC Hamming code.
- The total number of check bits is the same as that required for the original SEC/DED Hamming code.

Comparing Overlapping Parity Codes

- As $d$ increases, the parity overhead $r/d$ decreases.
- The probability of having more than one bit error in the $d+r$ bits increases.
- $f$ - probability of a bit error & assume bit errors occur independently of one another.
- Probability of more than one bit error in a field of $d+r$ bits -
  \[
  \Phi (d,r) = 1 - (1-f)^{d+r} - (d+r)(1-f)^{d+r-1} \\
  \approx 0.5 (d+r)(d+r-1)f^2 \quad \text{for } f \ll 1
  \]
Comparison - Cont.

♦ If we have a total of $D$ data bits, we can reduce the probability of having more than one bit error by partitioning the $D$ bits into $D/d$ equal slices, with each slice being encoded separately.

♦ We therefore have a tradeoff between the probability of undetected error and the overhead $r/d$.

♦ The probability that there is an uncorrectable error in at least one of the $D/d$ slices is

$$
\Psi(D, d, r) = 1 - [1 - \Phi(d, r)]^{D/d}
$$

$$
\approx (D/d) \Phi(d, r) \quad (\text{for } \Phi(d, r) << 1)
$$

Numerical Comparisons ($D=1024, f=10^{-11}$)

<table>
<thead>
<tr>
<th>$d$</th>
<th>$r$</th>
<th>Overhead $r/d$</th>
<th>$\Psi(D, d, r)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>1.5000</td>
<td>0.5120E-16</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0.7500</td>
<td>0.5376E-16</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0.5000</td>
<td>0.8448E-16</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>0.3125</td>
<td>0.1344E-15</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>0.1875</td>
<td>0.2250E-15</td>
</tr>
<tr>
<td>64</td>
<td>7</td>
<td>0.1094</td>
<td>0.3976E-15</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>0.0625</td>
<td>0.7344E-15</td>
</tr>
<tr>
<td>256</td>
<td>9</td>
<td>0.0352</td>
<td>0.1399E-14</td>
</tr>
<tr>
<td>512</td>
<td>10</td>
<td>0.0195</td>
<td>0.2720E-14</td>
</tr>
<tr>
<td>1024</td>
<td>11</td>
<td>0.0107</td>
<td>0.5351E-14</td>
</tr>
</tbody>
</table>
Checksum

- Primarily used to detect errors in data transmission on communication networks
- Basic idea - add up the block of data being transmitted and transmit this sum as well
- Receiver adds up the data it received and compares it with the checksum it received
- If the two do not match - an error is indicated

Versions of Checksums

- Data words - d bits long
- Single-precision version - checksum is a \( \text{modulo } 2^d \) addition
- Double-precision version - \( \text{modulo } 2^{2d} \) addition
- In general - single-precision checksum catches fewer errors than double-precision, since it only keeps the rightmost d bits of the sum
- Residue checksum takes into account the carry out of the d-th bit as an end-around carry - somewhat more reliable
- The Honeywell checksum concatenates words into pairs for the checksum calculation (done \( \text{modulo } 2^{2d} \)) - guards against errors in the same position
Comparing the Versions

- In single-precision checksum, transmitted checksum and computed checksum match.
- In Honeywell checksum, computed checksum differs from received checksum and error is detected.
- All checksum schemes allow error detection but not error location - entire block of data must be retransmitted if an error is detected.
Berger Code

♦ Separable code
  * counts the number of 1s in the word
  * expresses it in binary
  * complements it
  * appends this quantity to the data

♦ Example - encoding 11101
  * Four 1s
  * 100 in binary
  * 011 after complementing
  * the encoded word 11101011

♦ Detects all unidirectional bit errors - one or more 1s turn to 0s and no 0s turn to 1s (or vice versa)

♦ If the same number of bits flip from 0 to 1 as from 1 to 0 - the error will not be detected

Overhead of Berger Code

♦ d data bits - at most d 1s - up to \( \lceil \log_2(d+1) \rceil \) bits to describe

♦ Overhead = \( \lceil \log_2(d+1) \rceil / d \)

♦ r - number of check bits

<table>
<thead>
<tr>
<th>d</th>
<th>r</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>0.5000</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>0.2667</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>0.3125</td>
</tr>
<tr>
<td>31</td>
<td>5</td>
<td>0.1613</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>0.1875</td>
</tr>
<tr>
<td>63</td>
<td>6</td>
<td>0.0952</td>
</tr>
<tr>
<td>64</td>
<td>7</td>
<td>0.1094</td>
</tr>
<tr>
<td>127</td>
<td>7</td>
<td>0.0551</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>0.0625</td>
</tr>
<tr>
<td>255</td>
<td>8</td>
<td>0.0314</td>
</tr>
<tr>
<td>256</td>
<td>9</td>
<td>0.0352</td>
</tr>
</tbody>
</table>