

UNIVERSITY OF MASSACHUSETTS
Dept. of Electrical & Computer Engineering

Fault Tolerant Computing
ECE 655

Part 4
Coding - I

Israel Koren
Fall 2006

ECE655/Koren Part.4 .1

Copyright 2006 Koren & Krishna

Information Redundancy - Coding

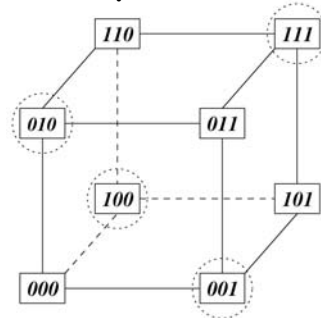
- ◆ A data word with d bits is encoded into a codeword with c bits - $c > d$
- ◆ Not all 2^c combinations are valid codewords
- ◆ To extract original data - c bits must be decoded
- ◆ If the c bits do not constitute a valid codeword an error is detected
- ◆ For certain encoding schemes - some types of 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

ECE655/Koren Part.4 .2

Copyright 2006 Koren & Krishna

Hamming Distance

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



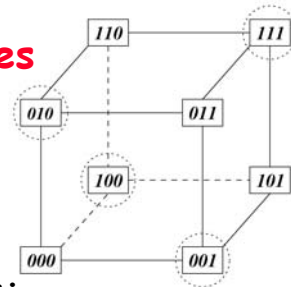
- ◆ Two words in this figure are connected by an edge if their Hamming distance is 1

ECE655/Koren Part.4 .3

Copyright 2006 Koren & Krishna

Hamming Distance - Examples

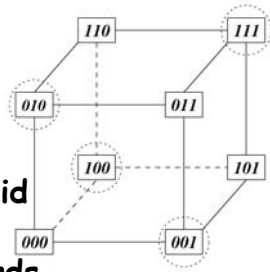
- ◆ 101 and 011 differ in two bit positions - Hamming distance of 2
 - * Need to traverse two edges to get from 101 to 011
- ◆ 101 and 100 differ by one bit position - a single error in the least significant bit in either of these two codewords will go undetected
- ◆ A Hamming distance of two between two codewords implies that a single bit error will not change one of the codewords into the other



ECE655/Koren Part.4 .4

Copyright 2006 Koren & Krishna

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

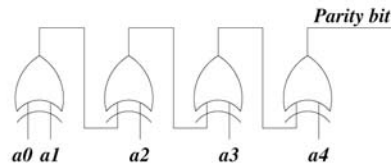
- ◆ The simplest **separable** codes are the **parity codes**
- ◆ A **parity-coded** word includes **n data bits** and an **extra bit** which holds the parity
- ◆ In even (odd) parity code - the extra bit is set so that the total number of **1's** in the **(d+1)**-bit word (including the parity bit)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

Properties of Parity Codes

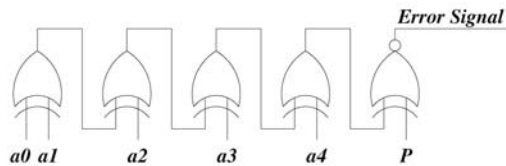
- ◆ A parity code has a **distance of 2** - will **detect** all single-bit errors
- ◆ If one bit flips from **0** to **1** (or vice versa) - the overall parity will not be the same - error can be **detected**
- ◆ Simple parity cannot **correct** any bit errors

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**



(a) Encoder



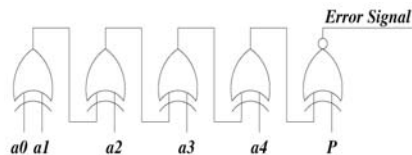
(b) Decoder

ECE655/Koren Part.4 .11

Copyright 2006 Koren & Krishna

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



ECE655/Koren Part.4 .12

Copyright 2006 Koren & Krishna

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 - $a_{63}, a_{62}, \dots, a_0$
- ◆ **Eight parity bits:**
- ◆ **First** - parity bit of $a_{63}, a_{55}, a_{47}, a_{39}, a_{31}, a_{23}, a_{15}, a_7$ - 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	0	1	0	1	1	0
1	1	0	0	0	0	0
0	0	0	1	1	1	1
1	1	1	1	1	1	0
1	0	0	1	0	0	0

- ◆ **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 (**a₃ a₂ a₁ a₀ p₂ p₁ p₀**), bit positions **0,1** and **2** are parity bits, the rest are data bits

State	Erroneous parity check(s)	Syndrome
No errors	None	000
Bit 0 (p ₀) error	p ₀	001
Bit 1 (p ₁) error	p ₁	010
Bit 2 (p ₂) error	p ₂	100
Bit 3 (a ₀) error	p ₀ , p ₁	011
Bit 4 (a ₁) error	p ₀ , p ₂	101
Bit 5 (a ₂) error	p ₁ , p ₂	110
Bit 6 (a ₃) error	p ₀ , p ₁ , p ₂	111

(7,4) Hamming Single Error Correcting (SEC) Code

State	Erroneous parity check(s)	Syndrome
No errors	None	000
Bit 0 (p_0) error	p_0	001
Bit 1 (p_1) error	p_1	010
Bit 2 (p_2) error	p_2	100
Bit 3 (a_0) error	p_0, p_1	011
Bit 4 (a_1) error	p_0, p_2	101
Bit 5 (a_2) error	p_1, p_2	110
Bit 6 (a_3) error	p_0, p_1, p_2	111

- ◆ If only p_0 check fails - bit 0 (p_0) is in error
- ◆ p_0 check fails also when bit 4 (a_1) is in error
- ◆ A parity bit covers all bits whose error it indicates
- ◆ p_0 covers positions 0,3,4,6 - $p_0 = a_0 \oplus a_1 \oplus a_3$
- ◆ p_1 covers positions 1,3,5,6 - $p_1 = a_0 \oplus a_2 \oplus a_3$
- ◆ p_2 covers positions 2,4,5,6 - $p_2 = a_1 \oplus a_2 \oplus a_3$

Definition - Syndrome

$$p_0 = a_0 \oplus a_1 \oplus a_3$$

$$p_1 = a_0 \oplus a_2 \oplus a_3$$

$$p_2 = a_1 \oplus a_2 \oplus a_3$$

- ◆ Example: $a_3 a_2 a_1 a_0 = 1100$ and $p_2 p_1 p_0 = 001$
- ◆ Suppose 1100001 becomes 1000001
- ◆ Recalculate $p_2 p_1 p_0 = 111$
- ◆ Difference (bit-wise XOR) is 110
- ◆ This difference is called **syndrome** - indicates the bit in error
- ◆ It is clear that a_2 is in error and the correct data is $a_3 a_2 a_1 a_0 = 1100$

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**

$$\begin{matrix} a_3 & a_2 & a_1 & a_0 & p_2 & p_1 & p_0 \\ \left[\begin{array}{ccccccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array} \right] & \begin{bmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \\ p_2 \\ p_1 \\ p_0 \end{bmatrix} & = & \begin{bmatrix} s_2 \\ s_1 \\ s_0 \end{bmatrix}
 \end{matrix}$$

$p_0 = a_0 \oplus a_1 \oplus a_3$

$p_1 = a_0 \oplus a_2 \oplus a_3$

$p_2 = a_1 \oplus a_2 \oplus a_3$

ECE655/Koren Part.4 .21

Copyright 2006 Koren & Krishna

Selecting Syndromes

State	Erroneous parity check(s)	Syndrome
No errors	None	000
Bit 0 (p_0) error	p_0	001
Bit 1 (p_1) error	p_1	010
Bit 2 (p_2) error	p_2	100
Bit 3 (a_0) error	p_0, p_1	011
Bit 4 (a_1) error	p_0, p_2	101
Bit 5 (a_2) error	p_1, p_2	110
Bit 6 (a_3) error	p_0, p_1, p_2	111

- ◆ 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 **$a_3 a_2 a_1 p_2 a_0 p_1 p_0$**
- ◆ 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

ECE655/Koren Part.4 .22

Copyright 2006 Koren & Krishna

Syndrome Generation for (8,4) Hamming Code

$$\begin{array}{c}
 a_3 a_2 a_1 a_0 p_3 p_2 p_1 p_0 \\
 \begin{bmatrix}
 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1
 \end{bmatrix}
 \begin{bmatrix}
 a_3 \\
 a_2 \\
 a_1 \\
 a_0 \\
 p_3 \\
 p_2 \\
 p_1 \\
 p_0
 \end{bmatrix}
 =
 \begin{bmatrix}
 s_3 s_2 s_1 s_0
 \end{bmatrix}
 \end{array}$$

- ◆ 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

ECE655/Koren Part.4 .25

Copyright 2006 Koren & Krishna

Example

$$\begin{array}{c}
 a_3 a_2 a_1 a_0 p_3 p_2 p_1 p_0 \\
 \begin{bmatrix}
 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1
 \end{bmatrix}
 \begin{bmatrix}
 a_3 \\
 a_2 \\
 a_1 \\
 a_0 \\
 p_3 \\
 p_2 \\
 p_1 \\
 p_0
 \end{bmatrix}
 =
 \begin{bmatrix}
 s_3 s_2 s_1 s_0
 \end{bmatrix}
 \end{array}$$

- ◆ 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

ECE655/Koren Part.4 .26

Copyright 2006 Koren & Krishna

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

a_3	a_2	a_1	a_0	p_3	p_2	p_1	p_0
0	1	1	1	1	0	0	0
1	0	1	1	0	1	0	0
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	1

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 -

$$\Phi(d,r) = 1 - (1-f)^{d+r} - (d+r)f(1-f)^{d+r-1}$$

$$\approx 0.5 (d+r)(d+r-1)f^2 \quad (\text{for } f \ll 1)$$

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

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

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

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

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

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 modulo 2^d addition
- ◆ Double-precision version - 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 modulo 2^{2d})
 - guards against errors in the same position

Comparing the Versions

0000	0000	0000	
0101	0101	0101	
1111	1111	1111	00000101
0010	0010	0010	11110010
0110	00010110	0111	11110111

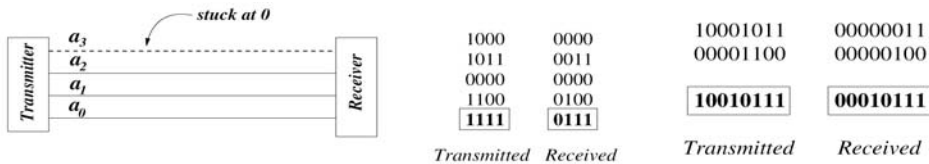
(a) Single-precision

(b) Double-precision

(c) Residue

(d) Honeywell

Comparison - Example



(a) Circuit

(b) Single-Precision

(c) Honeywell

- ◆ 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
- ◆ $\text{Overhead} = \lceil \log_2(d+1) \rceil / d$

- ◆ r - number of check bits

d	r	Overhead
8	4	0.5000
15	4	0.2667
16	5	0.3125
31	5	0.1613
32	6	0.1875
63	6	0.0952
64	7	0.1094
127	7	0.0551
128	8	0.0625
255	8	0.0314
256	9	0.0352