Karlsruhe Institute of Technology

Reliable Computing I

Lecture 6: Information Redundancy

Instructor: Mehdi Tahoori

INSTITUTE OF COMPUTER ENGINEERING (ITEC) – CHAIR FOR DEPENDABLE NANO COMPUTING (CDNC)

Today's Lecture

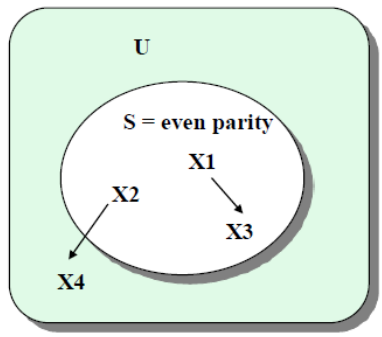
- Code, codeword, binary code
- Error detecting and correcting codes
- Hamming distance and codes
- Parity prediction
  - Odd/even parity
  - Basic parity approaches

Error Detection through Encoding

- At logic level, codes provide means of masking or detecting errors
- Formally, code is a subset S of universe U of possible vectors
- A noncode word is a vector in set U-S

X1 is a codeword
<10010011>
due to multiple bit error,
becomes
X3 = <10011100>
not detectable

X2 is a codeword,
becomes X4 noncode
detectable



Basic Idea

- Start with k-bit data word
- Add r check bits
- Total = n-bit codeword (n=k+r)
- Map 2^k data words to 2^n sized codeword space
- Overhead = r/n (sometimes computed as r/k)
  - E.g., for (single-bit) parity, the overhead is 1/n

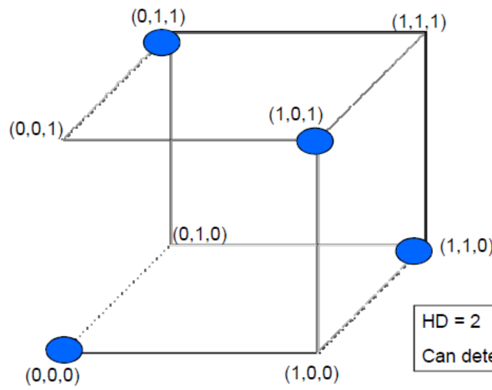
Basic Concepts

- Code, codeword, encoding, decoding error detection code, error correcting code
- Hamming distance properties:
  - The Hamming weight of a vector x (e.g., codeword), w(x), is number of nonzero elements of x.
  - Hamming distance between two vectors x and y, d(x,y) is number of bits in which they differ.
  - Distance of a code is a minimum of Hamming distances between all pairs of code words.
  - Example: x = (1011), y = (0110)
    w(x) = 3, w(y) = 2, d(x, y) = 3

Hamming Distance

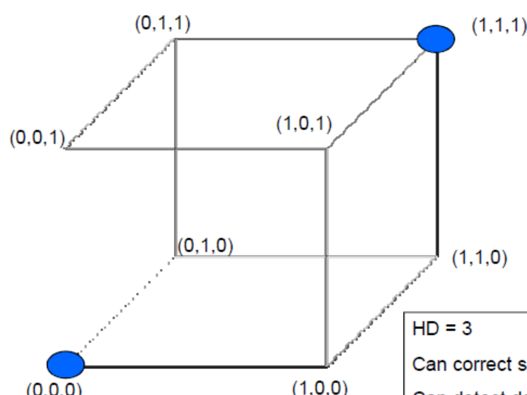
- Hamming distance (HD): number of bits in which two words differ from each other
  - E.g., 0010 and 1110 have a Hamming distance of ??
- For a group of codewords, the minimum HD between any two codewords determines the code's ability to detect and/or correct errors
  - This is a fundamental rule, not just some ad-hoc reasoning

Hamming Distance Visual: HD=2



HD = 2
Can detect single errors

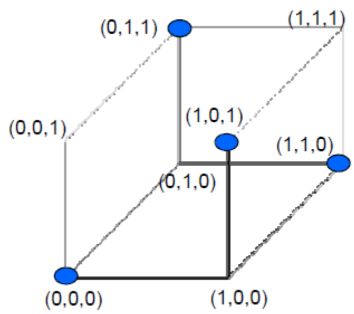
Hamming Distance Visual: HD=3



HD = 3
Can correct single errors
Can detect double& single errors

Hamming Distance and Error Detection

- Can detect up to t-bit errors if HD >= t + 1

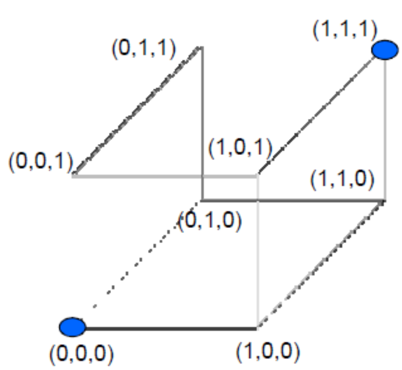


HD=2, detects 1-bit errors

- **What if we receive 111?**
  - **Could've been 011**
  - **Could've been 101**
  - **Could've been 110**
  - **What about 001? Or 000??**
- **What if we receive 011?**
  - **Could it have been 001???**

Hamming Distance and Error Detection

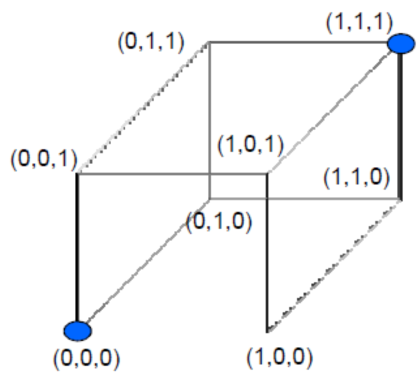
- Can detect up to t-bit errors if HD >= t + 1



HD=3, detects 1,2-bit errors

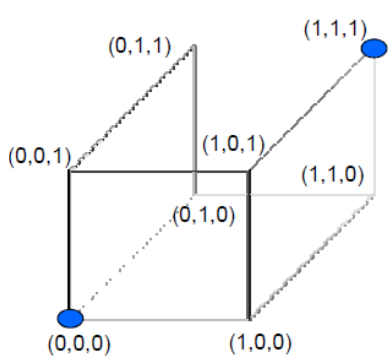
Hamming Distance and Error Correction

- Can correct up to t-bit errors if HD >= 2t+1



Hamming Distance and Error Correction

- Can correct up to t-bit errors if HD >= 2t+1



- **What if we receive 011?**
  - **More likely to have been 111**
  - **But could've been 000**
  - **Guess that it was 111**
- **What if we receive 111?**
  - **Could it have been 000?**

Summary: Hamming Distance Properties

- To detect all error patterns of Hamming distance ≤ d,
  code distance must be ≥ d+1
  - e.g., code with distance 2 can detect patterns with distance 1 (i.e., single bit errors)
- To correct all error patterns of Hamming distance ≤ c,
  code distance must be ≥ 2c + 1
- To correct all patterns of Hamming distance c and detect up to d additional errors ,
  code distance must be ≥ 2c + d + 1
  - e.g., code with distance 3 can detect and correct all single-bit errors

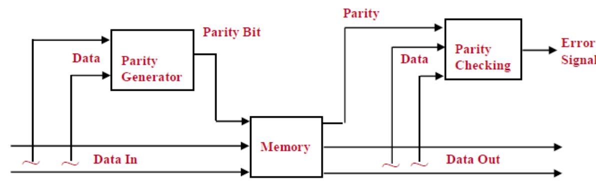
Single-bit Parity

- Simplest error detection code
  - Adds one bit of redundancy to each data word
- Even (odd) parity: add bit such that total number of ones in codeword is even (odd)
  - E.g., 001010 gets a parity bit of 0 for even parity (1 for odd)
- Can detect all single-bit errors
  - Hamming distance ≥ 2
  - Could be greater than 2 if data words don't use all bit combinations
- Drawbacks:
  - Can't detect anything except single-bit errors

Parity Codes - Example

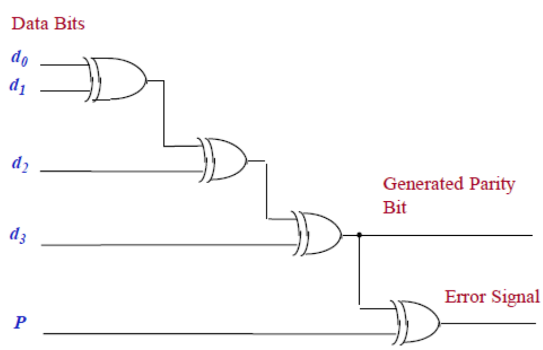
Odd and even parity codes for BCD data

Decimal Digit	BCD	BCD odd parity	BCD even parity
0	0000	00001	00000
1	0001	00010	00011
2	0010	00100	00101
3	0011	00111	00110
4	0100	01000	01001
5	0101	01011	01010
6	0110	01101	01100
7	0111	01110	01111
8	1000	10000	10001
9	1001	10011	10010
		↑ Parity Bit	↑ Parity Bit

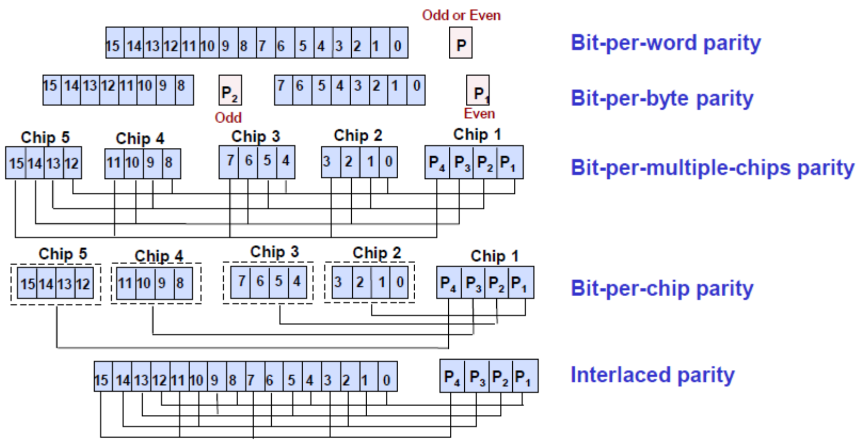


Organization of a memory that uses single-bit parity. The parity bit is generated when data is written to memory and checked when data is read.

XOR Tree for Parity Generation



Codes for RAMs



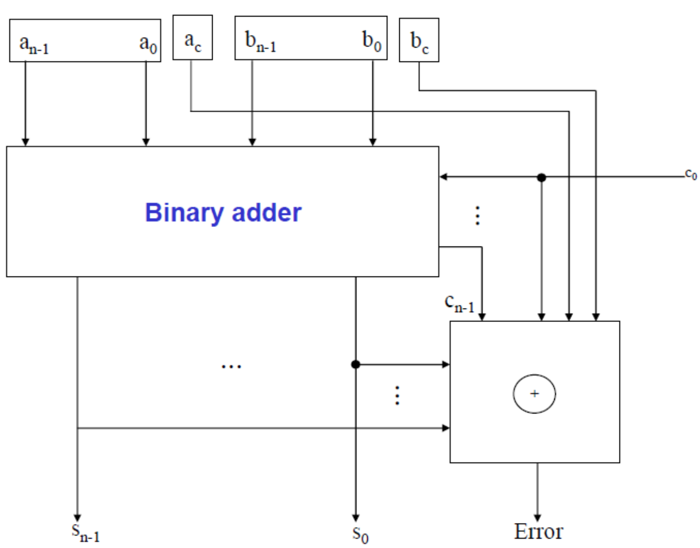
Parity Codes for Memory - Comparison

Parity Code	Advantages	Disadvantages
Bit-per-word: one parity bit per data word	Detects all single-bit errors	Certain errors undetected, e.g., a word, including parity bit becomes all 1s, due to a failure of a bus or a set of data buffers.
Bit-per-byte: each data portion (e.g., a byte) is protected by a separate parity bit; the parity of one group should be even and the parity of the other group should be odd	Detects all-1s and all-0s conditions	Ineffective for multiple errors, e.g., the whole-chip failure
Bit-per-multiple-chips: one bit from each chip is associated with a single parity bit	Detects failure of entire chip	Cannot locate failure of complete chip
Bit-per-chip: each parity bit is associated with one chip of the memory	Detects single-bit errors and identifies chip with erroneous bit	Susceptible to whole-chip failure, i.e., a single chip error can result in multiple bits to be corrupted and this may go undetected.
Interlaced: similar to the bit-per-multiple-chips; must ensure that no two adjacent bits are from the same parity group	Detects errors in adjacent bits	Parity groups are not based on physical organization of the memory

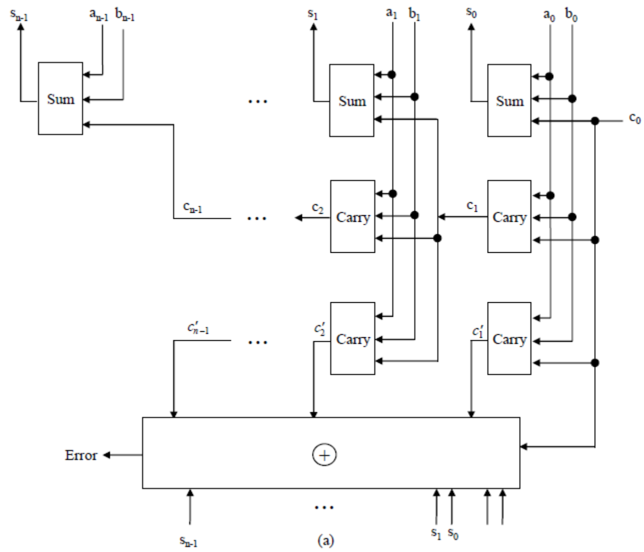
Parity Prediction in Arithmetic Circuits

- Binary Adder
  - Two inputs: $a = (a_{n-1} \dots a_0\ a_c)$ and $b = (b_{n-1} \dots b_0\ b_c)$
  - Two operands to be added: $(a_{n-1} \dots a_0)$ and $(b_{n-1} \dots b_0)$
  - a_c and b_c are check bits of a and b respectively
  - Encoded output will be $s = (s_{n-1} \dots s_0\ s_c)$ where $(s_{n-1} \dots s_0)$ are determined by the ordinary binary addition of $(a_{n-1} \dots a_0)$ to $(b_{n-1} \dots b_0)$ and s_c is the check bit for $(s_{n-1} \dots s_0)$
  - Then $s_c = \sum_{i=0}^{n-1} s_i = \sum_{i=0}^{n-1} a_i \oplus \sum_{i=0}^{n-1} b_i \oplus \sum_{i=0}^{n-1} c_i$
  - Reduces to $s_c = a_c \oplus b_c \oplus \sum_{i=0}^{n-1} c_i$

Parity Prediction in Binary Adder



Parity-Checked Binary Adder



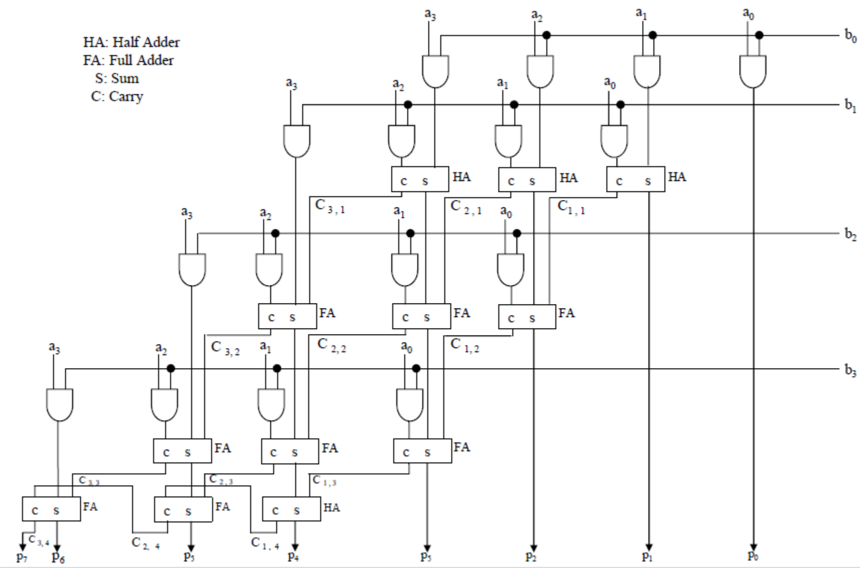
Binary Multiplier

$$p_0 = a_0 b_0$$
$$p_1 = a_0 b_1 \oplus a_1 b_0$$
$$p_2 = a_0 b_2 \oplus a_1 b_1 \oplus a_2 b_0 \oplus c_{1,1}$$
$$p_3 = a_0 b_3 \oplus a_1 b_2 \oplus a_2 b_1 \oplus a_3 b_0 \oplus c_{2,1} \oplus c_{1,2}$$
$$p_4 = a_1 b_3 \oplus a_2 b_2 \oplus a_3 b_1 \oplus c_{3,1} \oplus c_{2,2} \oplus c_{1,3}$$
$$p_5 = a_2 b_3 \oplus a_3 b_2 \oplus c_{3,2} \oplus c_{2,3} \oplus c_{1,4}$$
$$p_6 = a_3 b_3 \oplus c_{3,3} \oplus c_{2,4}$$
$$p_7 = c_{3,4}$$

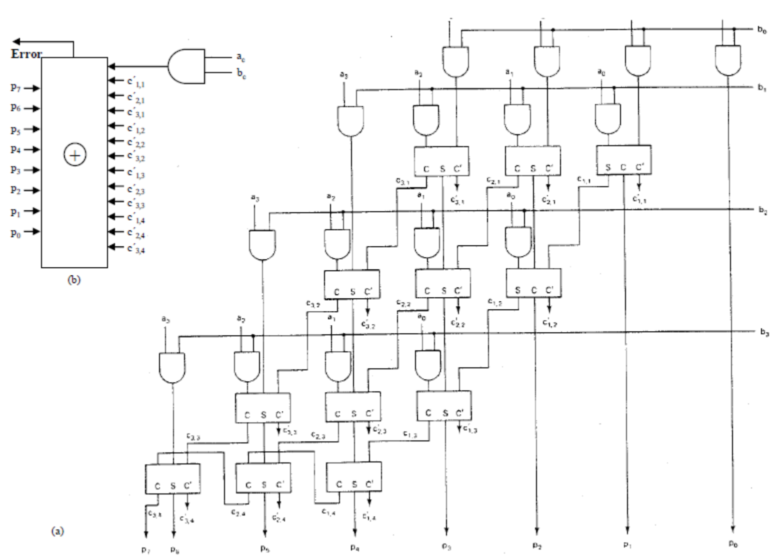
- Therefore, denoting the check bit for ($p_7 \ldots p_0$) by p_c

$$p_c = \sum_{i=0}^{7} p_i$$
$$= (\sum_{i=0}^{3} a_i)(\sum_{i=0}^{3} b_i) \oplus \sum_{i=1}^{3}\sum_{j=1}^{4} c_{i,j}$$
$$= a_c b_c \oplus \sum_{i=1}^{3}\sum_{j=1}^{4} c_{i,j}$$

Multiplier Using Array of Full-Half Adders

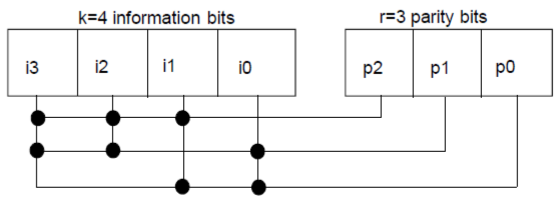


Parity Checked Multiplier



Overlapping Parity (for single-bit errors)

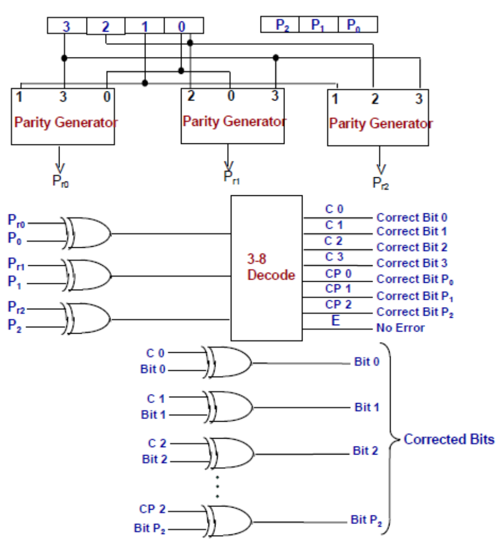
- Parity groups are formed with each bit appearing in more than one parity group
- Errors can be detected and located
- Erroneous bit can be corrected by a simple complementation



Which bit has error?	Parity bits affected
i3	p2, p1, p0
i2	p2, p1
i1	p2, p0
i0	p1, p0
p2	p2
p1	p1
p0	p0

When receiving codeword, re-compute 3 parity bits and compare to those that were sent. If different, can diagnose error (and correct it)!

Error Correction with Overlapped Parity



Generalized Overlapping Parity Codes

- The previous slide showed how to use overlapping parity to detect and diagnose single-bit errors
- For single-bit errors, there are k+r possible errors
  - Therefore, we need $2^r \geq k + r + 1$ to uniquely diagnose errors
- In general, can extend this scheme to detect and diagnose more than single-bit errors
  - General approach called "Hamming Codes"

Hamming Error-Correcting Code

- Require from 10% to 40% redundancy
- Best thought of as overlapping parity
- The Hamming single-error correcting code uses c parity check bits to protect k bits of information:
  - $2^c \geq c + k + 1$
- Example:
  - suppose four information bits (d3, d2, d1, d0) and as a result three parity bits (p1, p2, p3)
  - the bits are partitioned into groups as (d3, d1, d0, p1), (d3, d2, d0, p2) and (d3, d2, d1, p3)
    - the grouping of bits can be determine from a list of binary numbers from 0 to 2^k - 1.
  - each check bit is specified to set the parity, either even or odd, of its respective group

7	6	5	4	3	2	1
d_3	d_2	d_1	p_3	d_0	p_2	p_1

Hamming Error-Correcting Code

Determining the bit groups (three parity bits)

0 0 0			
0 0 1	1		
0 1 0		2	
0 1 1	3	3	
1 0 0			4
1 0 1	5		5
1 1 0		6	6
1 1 1	7	7	7

Parity bits calculation
p1 = XOR of bits (3, 5, 7)
p2 = XOR of bits (3, 6, 7)
p3 = XOR of bits (5, 6, 7)

Parity checking
c1 = XOR of bits (1,3, 5, 7)
c2 = XOR of bits (2,3, 6, 7)
c3 = XOR of bits (4,5, 6, 7)

- Observe that each group of bits for parity checking starts with a number that is a power of 2, e.g., 1, 2, 4.

(7,4) Hamming Code

- Class of (n,k) Hamming codes, e.g., (7,4) [r= n-k =3]
- Let i1, i2, i3, i4 be the information bits
- Let p1, p2, p4 be the check bits
- p1 = i1 xor i2 xor i4
- p2 = i1 xor i3 xor i4
- p4 = i2 xor i3 xor i4
- Let H be the Parity Check Matrix
- If C is a codeword, then H C = 0 (mult modulo 2!)
- Else, H C = S, where S is the syndrome
  - Syndrome identifies where error occurred (i.e., which bit)
  - This works out like magic because of some cute math

(7,4) Hamming Code

- H =

p1	p2	i1	p4	i2	i3	i4
1	0	1	0	1	0	1
0	1	1	0	0	1	1
0	0	0	1	1	1	1

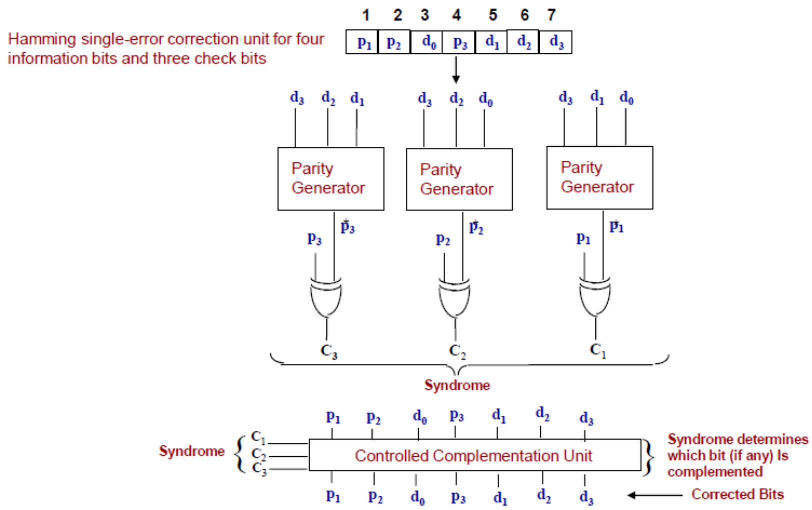
- Info word: 0101: p1 = 0, p2 = 1, p4 = 0
  - codeword is 0100101
- Example1:
  - received error-free codeword R = 0100101
  - Compute syndrome: S = H R = 0 = [0 0 0]
- Example 2:
  - received R =0110101 (i.e., error in bit position 3)
  - Compute syndrome: S = H R = [1 1 0]
    - Read backwards this is 011 = 3

Check Bits and Syndromes for Single-Bit Errors

- The original data is encoded by generating a set C_g, of parity bits.
- To check correctness, the encoding process is repeated and a set C_c, of parity bits is generated.
- If C_g and C_c agree, the information is correct.
- If C_g and C_c disagree, the information is incorrect and must be corrected.
- To aid the correction, a *syndrome* is defined:
  - The syndrome is a binary word that has 1 in each bit position in which C_g and C_c disagree; the syndrome points directly to the erroneous bit.

Erroneous bits	Check bits affected	Syndromes
d_0	p_1, p_2	110
d_1	p_1, p_3	101
d_2	p_2, p_3	011
d_3	p_1, p_2, p_3	111
p_1	p_1	100
p_2	p_2	010
p_3	p_3	001

Hamming Single-Error Correction Unit



Single Error Correction and Double Error Detection Hamming Code (SEC-DED)

	p_1	p_2	d_0	p_3	d_1	d_2	d_3	p_4
	1	1	1	1	1	1	1	1
	1	0	1	0	1	0	1	0
	0	1	1	0	0	1	1	0
	0	0	0	1	1	1	1	0

c_1	c_2	c_3	c_4	
0	0	0	0	No errors
x_3	x_2	x_1	1	Single error in a position ($x_1x_2x_3$)
y_3	y_2	y_1	0	Double error
0	0	0	1	Error in bit p_4

- Consider a data word consisting of four information bits
- Three parity bits are needed to provide single error correction
- Adding an extra parity bit, the Hamming code can be used to **correct single bit errors and to detect double errors**

Check bits computation

P_1 = XOR (3, 5, 7)
P_2 = XOR (3, 6, 7)
P_3 = XOR (5, 6, 7)
P_4 = parity over the first 7 bits of the code word

Syndromes computation

C_1 = XOR (1, 3, 5, 7)
C_2 = XOR (2, 3, 6, 7)
C_3 = XOR (4, 5, 6, 7)
C_4 = parity over all 8 bits of the code word

Single Error Correction and Double Error Detection Hamming Code (SEC-DED) Example

Initial daa
d_0 d_1 d_2 d_3
0 1 1 0

Failure scenarios

	p_1	p_2	d_0	p_3	d_1	d_2	d_3	p_4
1	1	1	0	0	1	1	0	0
2	1	1	1	0	1	1	0	0
3	1	1	0	0	1	0	0	0
4	0	1	0	0	0	1	0	0
5	1	1	0	0	1	1	0	1

Corresponding Syndromes

	c_1	c_2	c_3	c_4	
1	0	0	0	0	No errors
2	1	1	0	1	Single error in position 3
3	0	1	1	1	Single error in position 6
4	0	0	1	0	Double error
5	0	0	0	1	Error in bit p_4