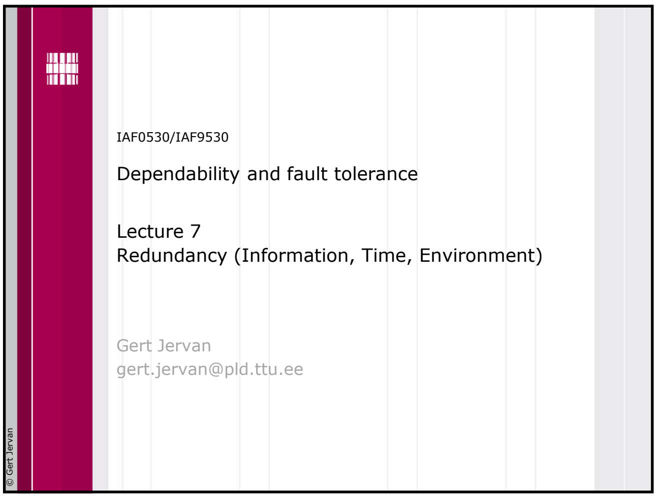
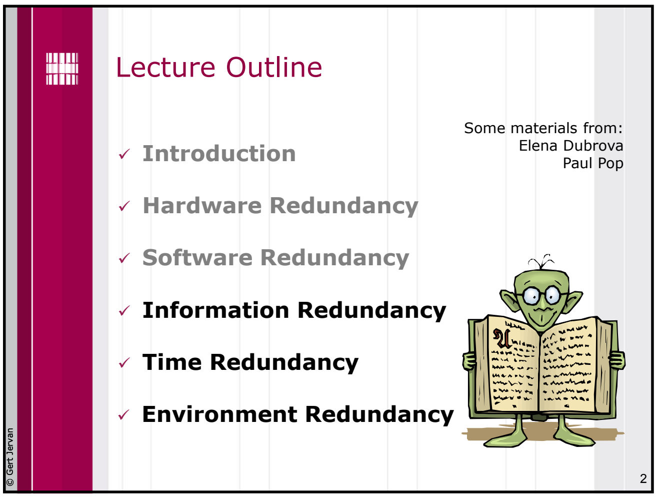


IAF0530/IAF9530

Dependability and fault tolerance

Lecture 7
Redundancy (Information, Time, Environment)

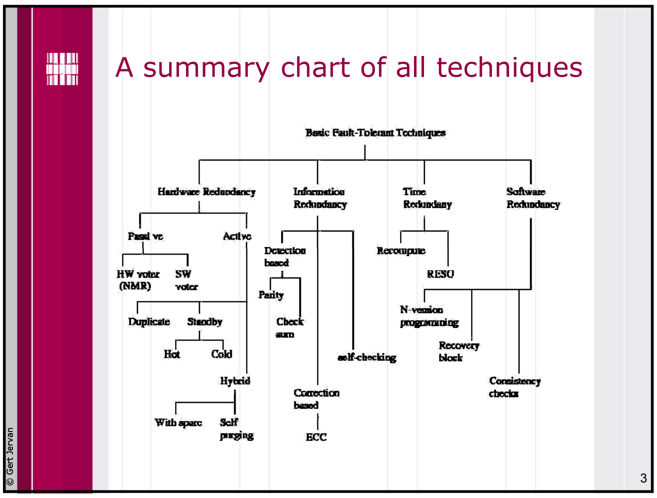
Gert Jervan
gert.jervan@pld.ttu.ee



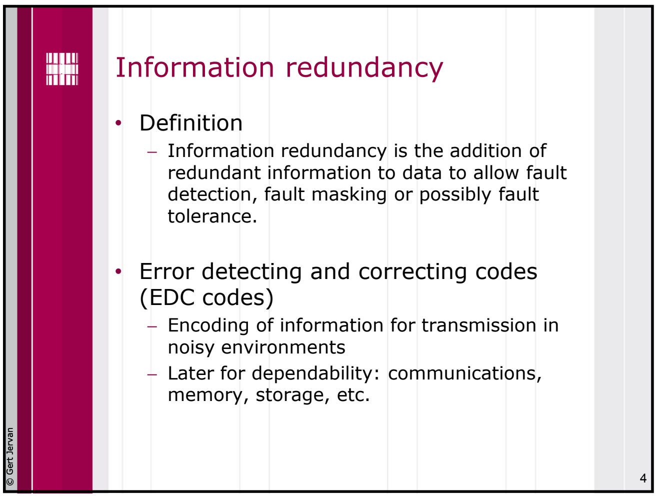
Lecture Outline

Some materials from:
Elena Dubrova
Paul Pop

- ✓ Introduction
- ✓ Hardware Redundancy
- ✓ Software Redundancy
- ✓ **Information Redundancy**
- ✓ **Time Redundancy**
- ✓ **Environment Redundancy**

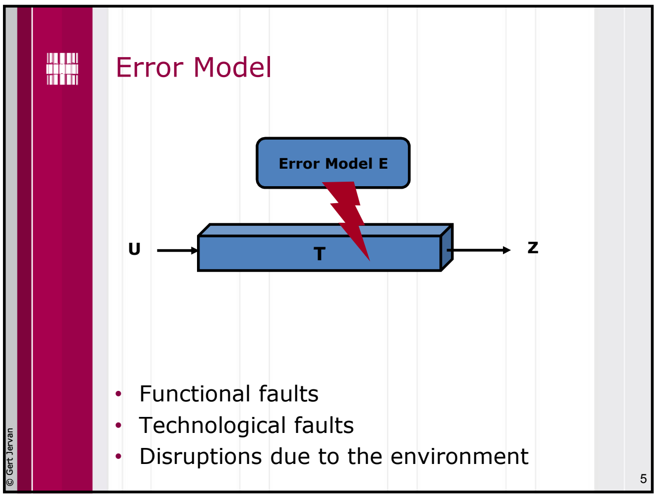


A summary chart of all techniques



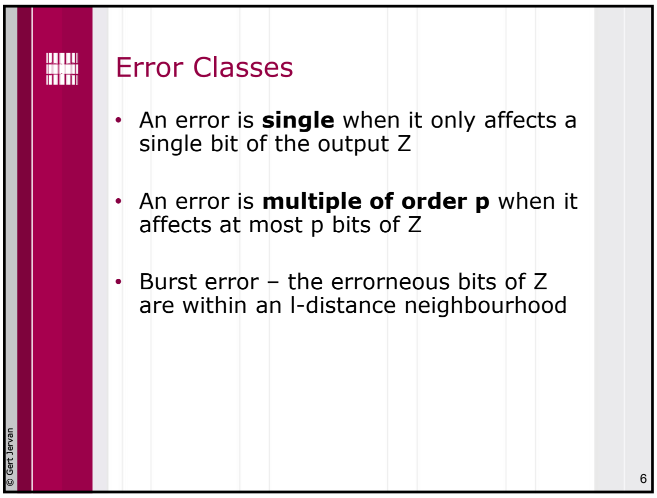
Information redundancy

- Definition
  - Information redundancy is the addition of redundant information to data to allow fault detection, fault masking or possibly fault tolerance.
- Error detecting and correcting codes (EDC codes)
  - Encoding of information for transmission in noisy environments
  - Later for dependability: communications, memory, storage, etc.



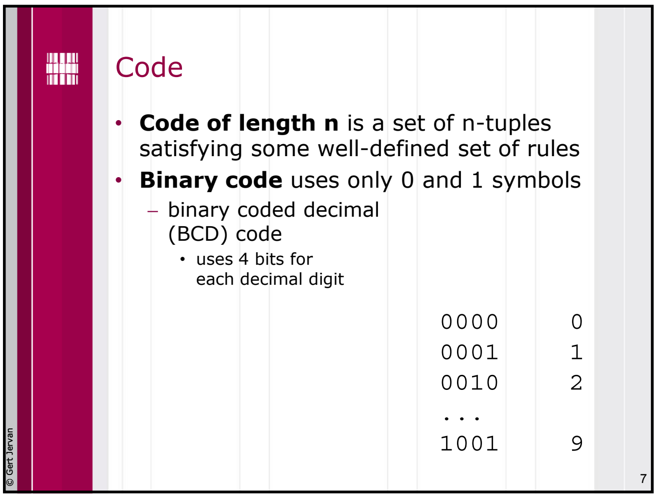
Error Model

- Functional faults
- Technological faults
- Disruptions due to the environment



Error Classes

- An error is **single** when it only affects a single bit of the output Z
- An error is **multiple of order p** when it affects at most p bits of Z
- Burst error – the errorneous bits of Z are within an l-distance neighbourhood



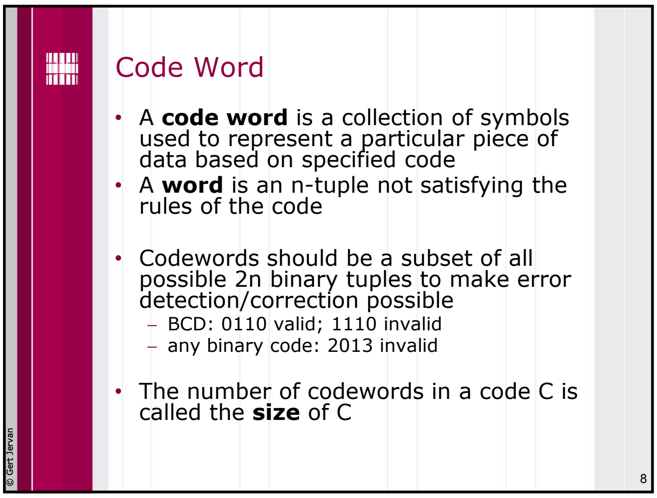
Code

- **Code of length n** is a set of n-tuples satisfying some well-defined set of rules
- **Binary code** uses only 0 and 1 symbols
  - binary coded decimal (BCD) code
    - uses 4 bits for each decimal digit

0000	0
0001	1
0010	2
...	
1001	9

© Gert Jervan

7

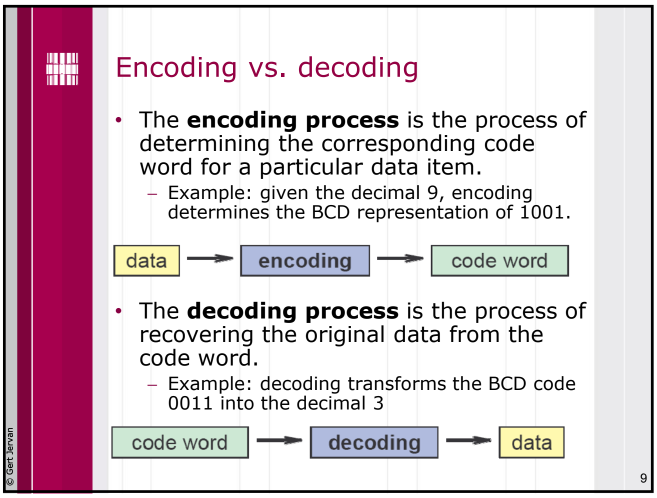


Code Word

- A **code word** is a collection of symbols used to represent a particular piece of data based on specified code
- A **word** is an n-tuple not satisfying the rules of the code
- Codewords should be a subset of all possible 2n binary tuples to make error detection/correction possible
  - BCD: 0110 valid; 1110 invalid
  - any binary code: 2013 invalid
- The number of codewords in a code C is called the **size** of C

© Gert Jervan

8



Encoding vs. decoding

- The **encoding process** is the process of determining the corresponding code word for a particular data item.
  - Example: given the decimal 9, encoding determines the BCD representation of 1001.

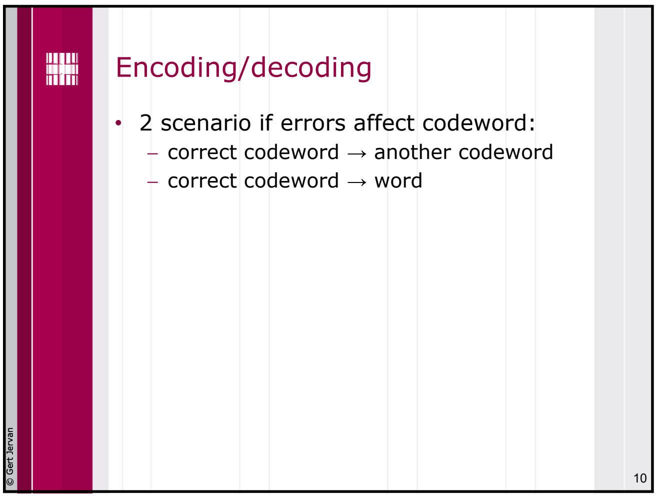
data
encoding
code word

- The **decoding process** is the process of recovering the original data from the code word.
  - Example: decoding transforms the BCD code 0011 into the decimal 3

code word
decoding
data

© Gert Jervan

9

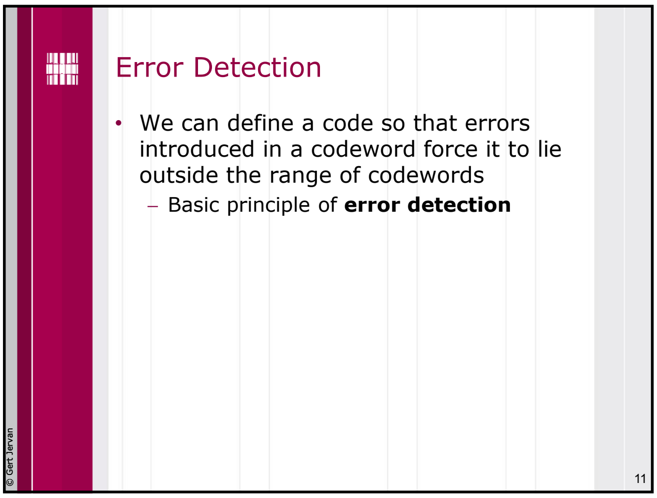


Encoding/decoding

- 2 scenario if errors affect codeword:
  - correct codeword → another codeword
  - correct codeword → word

© Gert Jervan

10

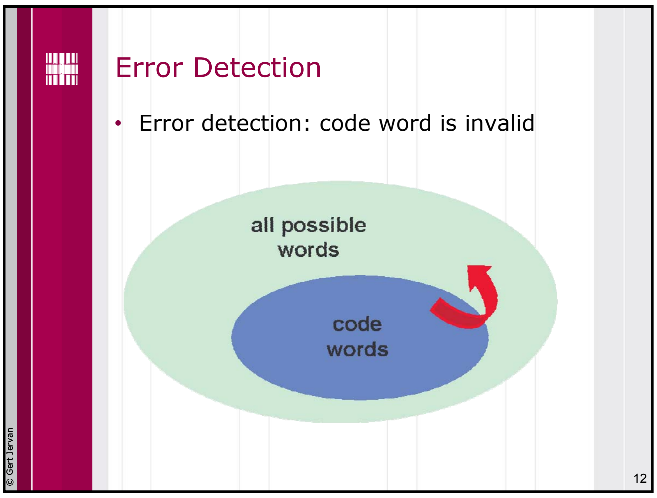


Error Detection

- We can define a code so that errors introduced in a codeword force it to lie outside the range of codewords
  - Basic principle of **error detection**

© Gert Jervan

11



Error Detection

- Error detection: code word is invalid

© Gert Jervan

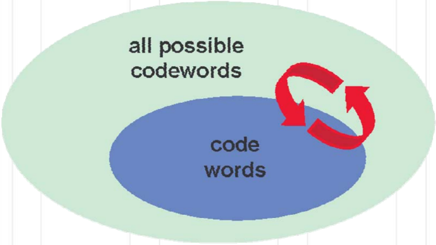
12

Error Correction

- We can define a code so that it is possible to determine the correct code word from the erroneous codeword
  - Basic principle of **error correction**

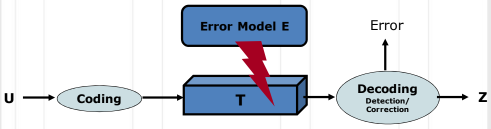
Error Correction

- Error correction: correct word can be identified from the corrupted word



EDC/ECC

- Error Detecting and Correcting Codes

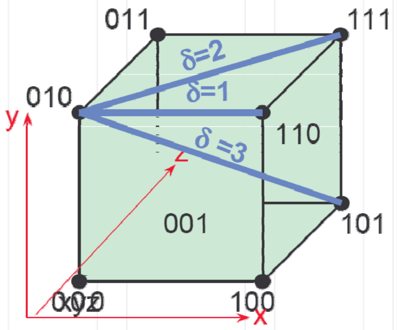


- Separable/non-separable codes
  - Separable: original information is appended with new information

EDC/ECC

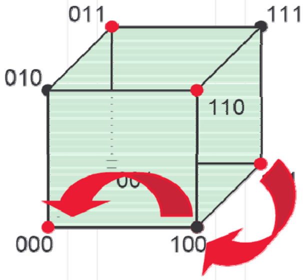
- Characterized by the number of bits that can be corrected
  - double-bit detecting code can detect two single-bit errors
  - single-bit correcting code can correct one single-bit error
- Hamming distance gives a measure of error detecting/correcting capabilities of a code
  - Number of bit positions in which the two words differ
    - Hamming distance of 1: 0000 to 0001; 2: 0000 to 0101
  - Code distance:
    - Minimum Hamming distance between any two valid code words

3-dimensional space (3-bit words)



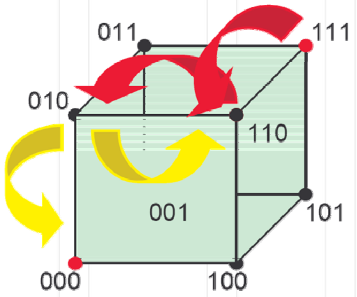
Error Detection

- If codewords are on distance ≥ 2, we can detect single-bit errors



Error Correction

- If codewords are on distance ≥ 3, we can correct single-bit errors



Code Distance

- Code distance is the minimum Hamming distance between any two distinct codewords

$C_d = 2$ code detects all single-bit errors
code: 00, 11
invalid code words: 01 or 10

$C_d = 3$ code corrects all single-bit errors
code: 000, 111
invalid code words: 001, 010, 100, 101, 011, 110

Code Capabilities

- To correct ε-bit errors a code should have the code distance Cd ≥ 2ε +1
- To be able to detect ε-bit errors a code should have the code distance Cd ≥ ε +1
- A code can correct up to c bit errors and detect up to d additional bit errors if and only if:

$$2c + d + 1 \le Cd$$

Separable/non-separable code

- Separable code
  - codeword = data + check bits
  - e.g. parity: 11011 = 1101 + 1
- Non-separable code
  - codeword = data mixed with check bits
  - e.g. cyclic: 1010001 -> 1101
- Decoding process is much easier for separable codes (remove check bits)

Information Rate

- The ratio k/n, where
  - k is the number of data bits
  - n is the number of data + check bits

  is called the information rate of the code
- Example: a code obtained by repeating data three times has the information rate 1/3

Code Characterization

- Cost: number of bits n that it needs
- Power of expression (cardinality): number of codewords N that it is able to represent
- Error model: defining the errors detected and/or corrected
  - Redundancy rate: rr=r/k (r: added bits)
  - Density of a code: d=N/2n
  - Coverage rate

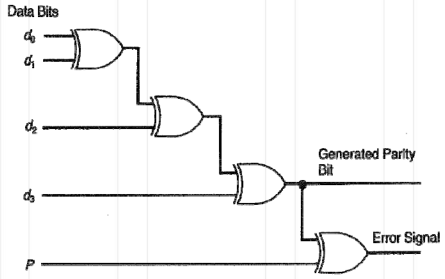
Parity codes

TABLE 3.3 Odd and even parity codes for BCD data

Decimal digit	BCD	BCD odd parity	BCD even parity
0	0000	0000 1	0000 0
1	0001	0001 0	0001 1
2	0010	0010 0	0010 1
3	0011	0011 1	0011 0
4	0100	0100 0	0100 1
5	0101	0101 1	0101 0
6	0110	0110 1	0110 0
7	0111	0111 0	0111 1
8	1000	1000 0	1000 1
9	1001	1001 1	1001 0
		↑ Parity bit	↑ Parity bit

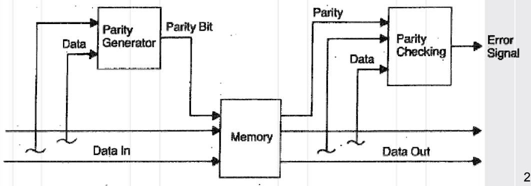
- Addition of an extra bit to a binary word such that the resulted code word has either an even number or an odd number of 1s.
- Separable code
- Hamming distance of 2
- Can detect any single bit error
  - Even parity
    - One bit flip: odd number of bits
  - Odd parity
    - One bit flip: even number of bits
- Application: bus, memory, transmission

Generation and checking circuit for parity codes: XOR



Memory with parity coding

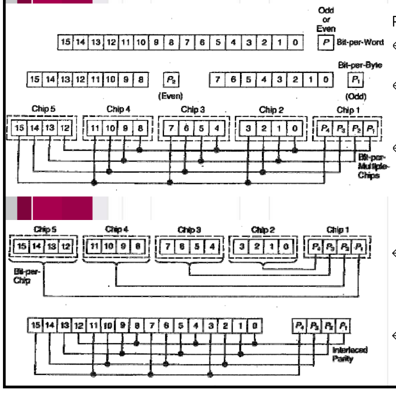
- 32-bit word + 4 parity bits = 36 bits stored
- 64-bit word + 8 parity bits = 72 bits stored



Application

- Memories are built of individual chips
  - 4-bit chips
  - 98% of all memory errors are single-bit errors
    - 1GByte DRAM with parity code: 0,7 failures per year
- If one chip fails: multiple-bit errors
  - Parity codes cannot detect multiple bit errors
- Modifications of the basic parity scheme
  - Bit-per-word parity
  - Bit-per byte parity
  - Bit-per chip parity
  - Bit-per multiple chips parity
  - Interlaced parity

Forms of parity code



Problems

← All bits are changed to 1 or 0

← Better, but still not able to detect a 4-bit chip failure

← Each bit from a chip forms a parity word with other bits
- Cannot locate the problem

← Can detect and locate the error
- Problem with chip-failure

← Used when errors in adjacent bits are of concern

Forms of parity code

Code	Advantages	Disadvantages
Even parity Bit-per-word	Detects single-bit errors	Certain errors go undetected, e.g., if a word, including the parity bit, becomes all is
Bit-per-byte parity	Detects the all is and the all 0s conditions	Ineffective in detection of multiple errors
Bit-per-multiple-Chips parity	Detects failure of entire chip	Failure of a complete chip is detected, but it is not located
Bit-per-chip parity	Detects single error and identifies the chip that contains the erroneous bit	Susceptible to the whole-chip failure
Interlaced parity	Detects errors in adjacent bits; does not take into account the physical memory organization	Parity groups not based on the physical memory organization

Overlapping parity code

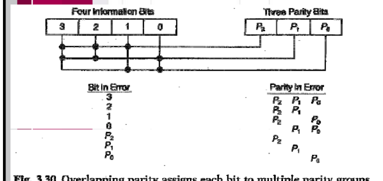


Fig. 3.30 Overlapping parity assigns each bit to multiple parity groups.

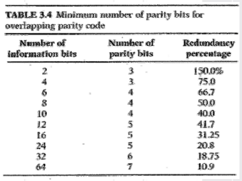


TABLE 3.4 Minimum number of parity bits for overlapping parity code

Number of information bits	Number of parity bits	Redundancy percentage
2	3	150.0%
4	3	75.0
6	4	66.7
8	4	50.0
10	4	40.0
12	5	41.7
16	5	31.25
24	5	20.8
32	6	18.75
64	7	10.9

- The impact of an erroneous bit is unique
  - Example: flip of data bit 2 affects parity bits P1 and P2
  - Bit can be detected and corrected
- The required redundancy decreases as the numbers of bits increase

Error correction using overlapping parity

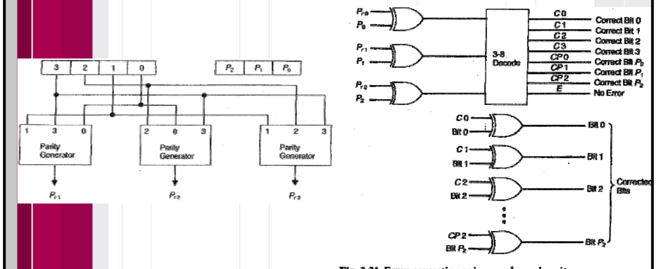


Fig. 3.31 Error correction using overlapped parity.

Syndrome and Costs

- Output of 3-8 decoder called syndrome bits
  - Indicates which of several possible bit flips occurred
- High Cost – 75% redundancy (3 parity bits for a 4-bit word). Cost goes down as information content increases.

m-of-n codes

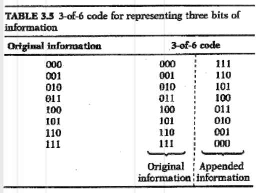


TABLE 3.5 3-of-6 code for representing three bits of information

Original information	3-of-6 code	
000	000	111
001	001	110
010	010	101
011	011	100
100	100	011
101	101	010
110	110	001
111	111	000
	Original information	Appended information

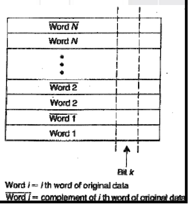
✓ m-of-n codes define code words that are n bits in length and contain exactly m 1s

✓ Example: 3-of-6

- Single Bit Errors – Erroneous word has m+1 or m-1 ones -- conceptually simple
- Disadvantage – Encoding, decoding, detection of fault difficult and complex
- Easiest Implementation:
  - i-of-2i code:
    - Take the original i-bits of information and append i bits such that the resulted code word has exactly i 1s

Duplication codes

- Duplication codes are based on the concept of completely duplicating the original information to form the code word.
  - Simply append the original i bits of information to itself: code of length 2i
  - If an error occurs, the two halves disagree
- Advantage: simple to encode/decode and detect errors
- Disadvantage: 100% redundancy
- Application
  - Memory systems
  - Communication systems


$\overline{\text{Word } N}$
Word N
$\overline{\text{Word 2}}$
Word 2
$\overline{\text{Word 1}}$
Word 1
Bit k
Word i = i th word of original data
$\overline{\text{Word } i}$ = complement of i th word of original data

Complemented duplication in a communication system

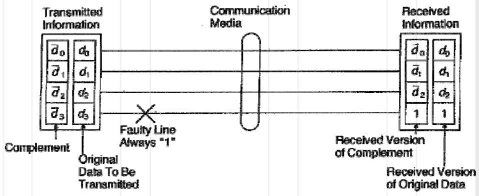
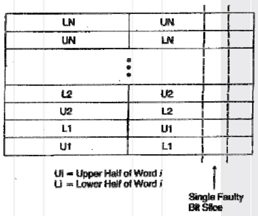


Fig. 3.33 Example of complemented duplication for error detection in a communication system.

Swap and compare duplication codes

- Maintain two copies of the original information, but swap the upper and lower halves of the second copy
- A single bit slice that is faulty affects the upper half of one copy of the information and the lower half of the other copy
  - By comparing the appropriate halves, the error can be detected



Checksum codes

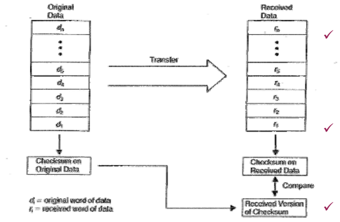

Original Data
d_n
d_5
d_4
d_3
d_2
d_1
Transfer
Received Data
r_n
r_5
r_4
r_3
r_2
r_1
Checksum on Original Data
Checksum on Received Data
Compare
Received Version of Checksum
d_i = original word of data
r_i = received word of data

Fig. 3.35 In checksum coding, the sum of the original data words is appended to the block of data.

- ✓ **Checksum**: sum of the original data, appended to the block of data
- ✓ Separable code applicable when blocks of data are transferred from one point to another
- ✓ Checksums cannot correct errors
- ✓ Four types of checksums
  1. Single-precision
  2. Double-precision
  3. Honeywell
  4. Residue checksums

Single-precision checksum

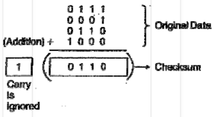


Fig. 3.36 A single-precision checksum is formed by adding the data words and ignoring any overflow.

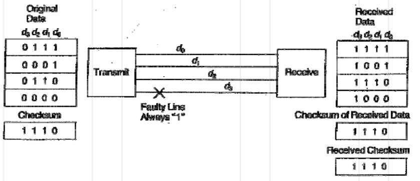


Fig. 3.37 The single-precision checksum is unable to detect certain types of errors. The received checksum and the checksum of the received data are equal, so no error is detected.

Double-precision checksums

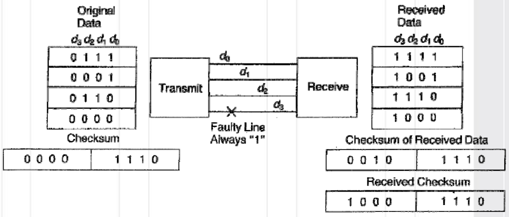


Fig. 3.38 A double-precision checksum is formed by adding the data using double-precision arithmetic. The received checksum and the checksum of the received data are not equal, so the error is detected.

Honeywell checksum

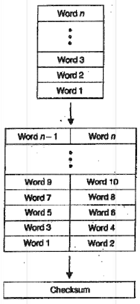


Fig. 3.39 In the Honeywell checksum, adjacent data words are concatenated prior to forming the checksum.

- Concatenate consecutive words to form a collection of double-length words.
  - The checksum is formed over the new double-length words
- If a complete column of the original data is erroneous, the modified data structure has two erroneous columns

Error detection using Honeywell checksums

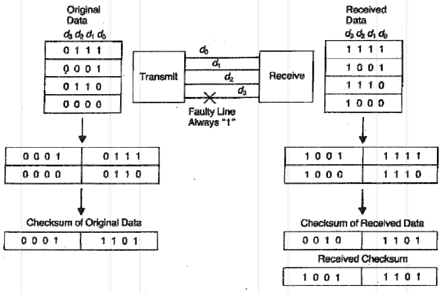


Fig. 3.40 Illustration of the error detection capability of the Honeywell checksum. The received checksum and the checksum of the received data are not equal, so the error is detected.

Residue checksums

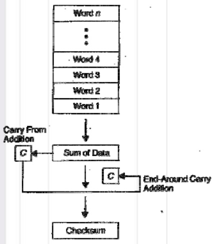


Fig. 3.41 The residue checksum is formed using end-around carry addition so that information in the carry bit is not lost.

- Residue checksums are similar with single-precision checksums, except that the most significant bit position is not ignored but is added back to the checksum in an end-around carry fashion.

Error detection using residue checksums

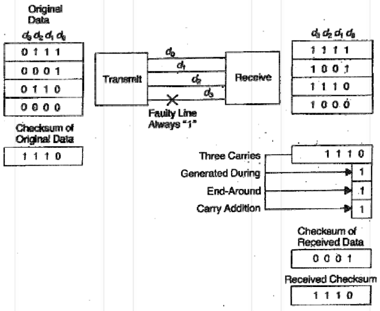


Fig. 3.42 Illustration of the error detection capability of the residue checksum. The checksum of the received data and the received checksum are not equal, so the error is detected.

Cyclic Codes

- Cyclic codes are special class of linear codes
- Used in applications where burst errors can occur
  - a group of adjacent bits is affected
  - digital communication, storage devices (disks, CDs)
- Important classes of cyclic codes:
  - Cyclic redundancy check (CRC)
    - Used in modems and network protocols
      - North-America T-carrier standard uses Extended-SuperFrame (ESF) cyclic code. Coding frame of 4632 bits. Detects 98,4% single or multiple errors
  - Reed-Solomon code
    - Used in CD and DVD players
      - CDs: up to 4000 consecutive errors can be corrected (2,5 mm of track)

Cyclic codes

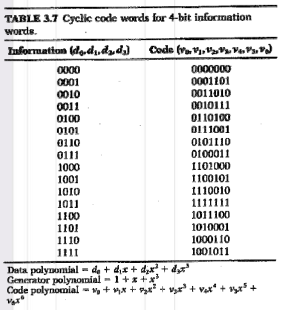


TABLE 3.7 Cyclic code words for 4-bit information words.

Information (d_0, d_1, d_2, d_3)	Code $(v_0, v_1, v_2, v_3, v_4, v_5, v_6)$
0000	0000000
0001	0001101
0010	0011010
0011	0010111
0100	0110100
0101	0111001
0110	0101110
0111	0100011
1000	1101000
1001	1100101
1010	1110010
1011	1111111
1100	1011100
1101	1010001
1110	1000110
1111	1001011

Data polynomial = $d_0 + d_1x + d_2x^2 + d_3x^3$
Generator polynomial = $1 + x + x^3$
Code polynomial = $v_0 + v_1x + v_2x^2 + v_3x^3 + v_4x^4 + v_5x^5 + v_6x^6$

- Cyclic code: the end-around shift of a code word will produce another code word
  - n bits in the original, k in the code
    - Called (n, k) cyclic code
- Simple encoding
  - Shift registers with feedback connections
- Can detect n-k adjacent errors
  - Burst errors

Cyclic code polynomial

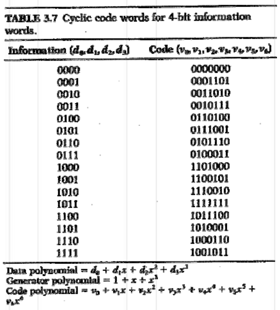


TABLE 3.7 Cyclic code words for 4-bit information words.

Information (d_0, d_1, d_2, d_3)	Code $(v_0, v_1, v_2, v_3, v_4, v_5, v_6)$
0000	0000000
0001	0001101
0010	0011010
0011	0010111
0100	0110100
0101	0111001
0110	0101110
0111	0100011
1000	1101000
1001	1100101
1010	1110010
1011	1111111
1100	1011100
1101	1010001
1110	1000110
1111	1001011

Data polynomial = $d_0 + d_1x + d_2x^2 + d_3x^3$
Generator polynomial = $1 + x + x^3$
Code polynomial = $v_0 + v_1x + v_2x^2 + v_3x^3 + v_4x^4 + v_5x^5 + v_6x^6$

- Data polynomial
  - $D(X) = d_0 + d_1x + d_2x^2 + d_3x^3$
- Generator polynomial
  - $G(X) = 1 + x + x^3$
- Cyclic code generation:
  - $V(X) = D(X) * G(X)$
  - Modulo-2 addition!
- V(X) is the code polynomial
  - $V(X) = v_0 + v_1x + v_2x^2 + v_3x^3 + v_4x^3 + v_5x^5 + v_6x^6$

Encoding

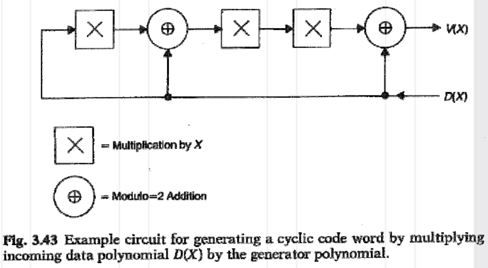


Fig. 3.43 Example circuit for generating a cyclic code word by multiplying an incoming data polynomial D(X) by the generator polynomial.

Encoding, cont.

TABLE 3.8 The encoding process for the circuit of Fig. 3.44

Clock period	Register values 1	2	3	$D(x)$	$V(x)$
0	0	0	0		
				1	1
1	1	0	1		
				1	0
2	1	1	1		
				0	1
3	0	1	1		
				1	0
4	1	0	0		
				0	0
5	0	1	0		
				0	0
6	0	0	1		
				0	1
7	0	0	0		

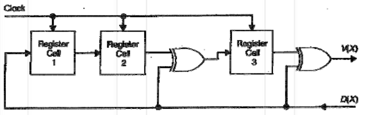


Fig. 3.44 Circuit for generating cyclic code words for the generator polynomial $G(X) = 1 + X + X^3$.

Decoding and error detection

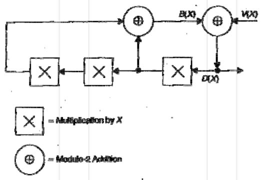


Fig. 3.45 A division circuit for use in decoding cyclic code words.

Decoding is done through division

- Data is obtained by dividing V(X) / G(X)
- The circuit in the figure implements the division

Syndrome polynomial S(X) is zero in case of no error

- R(X) = D(X)G(X) + S(X)

Decoding and error detection, cont.

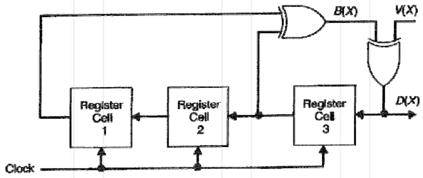


Fig. 3.46 Decoding circuit for the cyclic code with generator polynomial, $G(X) = 1 + X + X^3$.

Decoding and error detection, cont.

TABLE 3.9 The decoding process for the circuit of Fig. 3.46

Clock period	Register values 1	2	3	$V(x)$	$B(x)$	$D(x)$
0	0	0	0			
				1	0	1
1	0	0	1			
				0	1	1
2	0	1	1			
				1	1	0
3	1	1	0			
				0	1	1
4	1	0	1			
				0	0	0
5	0	1	0			
				0	0	0
6	1	0	0			
				1	1	0
7	0	0	0	↑ Code word		↑ Original information
	Syndrome					

TABLE 3.10 The decoding process with erroneous information

Clock period	Register values 1	2	3	$V(x)$	$B(x)$	$D(x)$
0	0	0	0			
				1	0	1
1	0	0	1			
				0	1	1
2	0	1	1			
				1	1	0
3	1	1	0			
				1	1	0
4	1	0	0			
				0	1	1
5	0	0	1			
				0	1	1
6	0	1	1			
				1	1	0
7	1	1	0	↑ Received word		
	Nonzero syndrome					

Reed-Solomon Code

- Reed-Solomon (RS) codes are a class of separable cyclic codes used to correct errors in a wide range of applications including
  - storage devices (tapes, compact disks, DVDs, bar-codes)
  - wireless communication (cellular telephones, microwave links)
  - satellite communication, digital television, high-speed modems (ADSL, xDSL)

Reed-Solomon Code

- The encoding for Reed-Solomon code is done the using the usual procedure
  - codeword is computed by shifting the data right n-k positions, dividing it by the generator polynomial and then adding the obtained reminder to the shifted data
- A key difference is that groups of m bits rather than individual bits are used as symbols of the code.
  - usually m = 8, i.e. a byte

Encoding

- An encoder for an RS code takes k data symbols of s bits each and computes a codeword containing n symbols of m bits each
- A Reed-Solomon code can correct up to n-k/2 symbols that contain errors

Example: RS(255,223) code

- A popular Reed-Solomon code is RS(255,223)
  - symbols are a byte (8-bit) long
  - each codeword contains 255 bytes, of which 223 bytes are data and 32 bytes are check symbols
  - n = 255, k = 223, this code can correct up to 16 bytes containing errors
  - each of these 16 bytes can have multiple bit errors.

Decoding

- Decoding of Reed-Solomon codes is performed using an algorithm designed by Berlekamp
  - popularity of RS codes is due to efficiency this algorithm to a large extent.
    - This algorithm was used by Voyager II for transmitting pictures of the outer space back to Earth
    - Basis for decoding CD in players

Summary of Cyclic Codes

- Any end-around shift of a codeword produce another codeword
- Code is characterized by its generator polynomial g(x), with a degree (n-k), n = bits in codeword,
  k = bits in data word
- Detect all single errors and all multiple adjacent error affecting
  (n-k) bits or less

Arithmetic codes

- Arithmetic codes are invariants to (a set of) arithmetic operations
  - A(b <op> c) = A(b) <op> A(c)
- Application
  - Checking arithmetic operations
- Examples of arithmetic codes
  - AN codes
  - Residue codes
  - Inverse-residue codes
  - Residue number system

AN codes

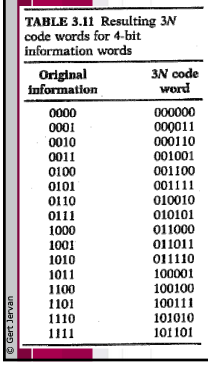


TABLE 3.11 Resulting 3*N* code words for 4-bit information words

Original information	3*N* code word
0000	000000
0001	000011
0010	000110
0011	001001
0100	001100
0101	001111
0110	010010
0111	010101
1000	011000
1001	011011
1010	011110
1011	100001
1100	100100
1101	100111
1110	101010
1111	101101

- AN code is formed by multiplying each data word N by some constant A
  - The magnitude of A determines the number of extra bits required to represent the code words and the error detection capability
- AN codes are invariant to addition and subtraction but not multiplication and division
- Example code: 3N
  - The constant must not be a power of 2
    - Power of 2 is a shift, difficult to detect the error
  - 3N requires n+2 bit words

AN codes encoding: 3N = 2N + N

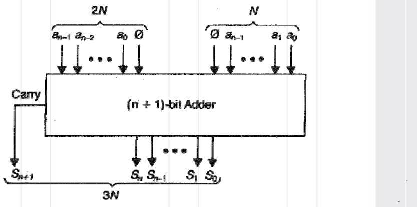


Fig. 3.48 Illustration of the use of an $(n + 1)$-bit adder to create $3N$ code words.

Error detection using AN codes

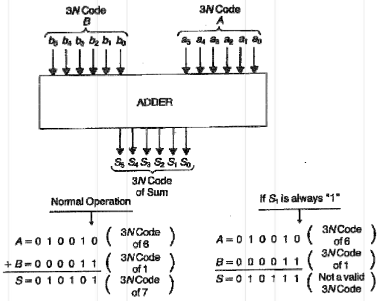
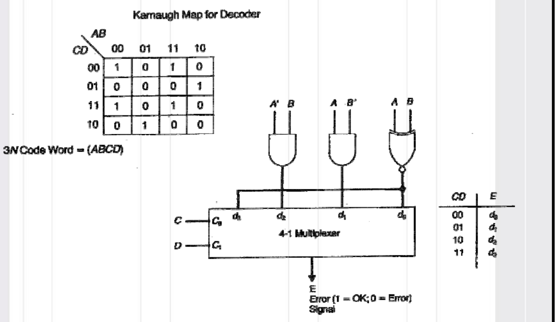

3N Code
B
b_5 b_4 b_3 b_2 b_1 b_0
3N Code
A
a_5 a_4 a_3 a_2 a_1 a_0
ADDER
S_5 S_4 S_3 S_2 S_1 S_0
3N Code of Sum
Normal Operation
If S_1 is always "1"
A = 0 1 0 0 1 0 (3N Code of 6)
+ B = 0 0 0 0 1 1 (3N Code of 1)
S = 0 1 0 1 0 1 (3N Code of 7)
A = 0 1 0 0 1 0 (3N Code of 6)
B = 0 0 0 0 1 1 (3N Code of 1)
S = 0 1 0 1 1 1 (Not a valid 3N Code)

Fig. 3.47 Illustration of the error detection capabilities of the $3N$ arithmetic code. The presence of the fault results in the sum being an invalid $3N$ code.

Error detection circuit for AN codes



CD \ AB	00	01	11	10
00	1	0	1	0
01	0	0	0	1
11	1	0	1	0
10	0	1	0	0

CD	E
00	d_0
01	d_1
10	d_2
11	d_3

Fig. 3.49 A simple error detection circuit for the $3N$ code can be constructed using combinational logic.

Residue codes

TABLE 3.12 Residue code words for 4-bit information words using a modulus of three

Information	Residue	Code word
0000	0	0000 | 00
0001	1	0001 | 01
0010	2	0010 | 10
0011	0	0011 | 00
0100	1	0100 | 01
0101	2	0101 | 10
0110	0	0110 | 00
0111	1	0111 | 01
1000	2	1000 | 10
1001	0	1001 | 00
1010	1	1010 | 01
1011	2	1011 | 10
1100	0	1100 | 00
1101	1	1101 | 01
1110	2	1110 | 10
1111	0	1111 | 00

- A residue code is a separable arithmetic code created by appending the residue (reminder) of a number to that number
  - Code = D | R, where R is the reminder of the division with an integer m
  - The number of bits in R depend on m

Residue codes are invariant to addition

Low-cost residues:
m = 2b – 1, b at least 2

  - R requires b bits
  - Easy to encode: division is Modulo-3 addition

Decoding simply removes R

Adder using residue codes

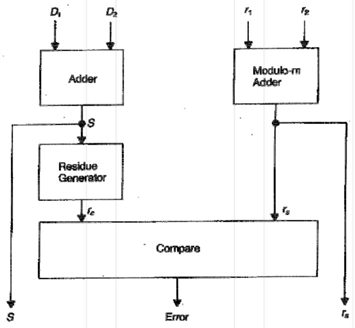


Fig. 3.50 The structure of an adder designed using the separable residue code.

Residue calculation for low-cost residues

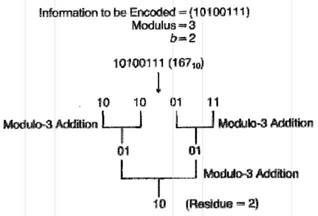
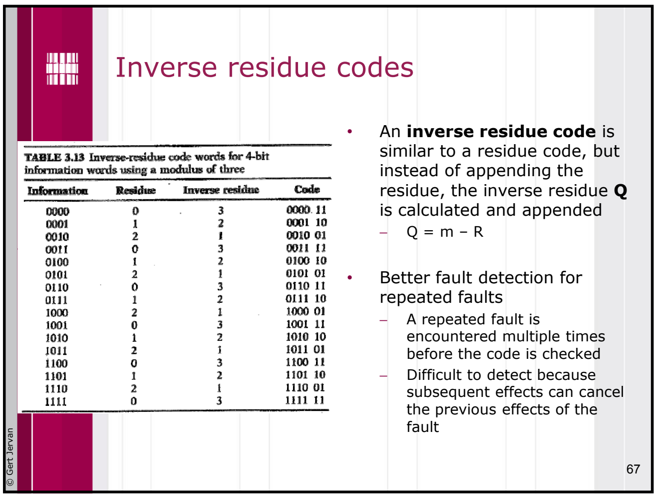


Fig. 3.51 The residue calculation for a low-cost residue code can be performed using successive additions.



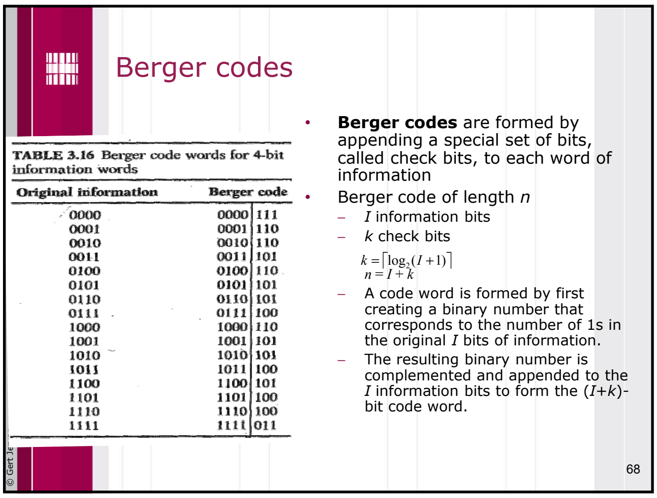
Inverse residue codes

TABLE 3.13 Inverse-residue code words for 4-bit information words using a modulus of three

Information	Residue	Inverse residue	Code
0000	0	3	0000 11
0001	1	2	0001 10
0010	2	1	0010 01
0011	0	3	0011 11
0100	1	2	0100 10
0101	2	1	0101 01
0110	0	3	0110 11
0111	1	2	0111 10
1000	2	1	1000 01
1001	0	3	1001 11
1010	1	2	1010 10
1011	2	1	1011 01
1100	0	3	1100 11
1101	1	2	1101 10
1110	2	1	1110 01
1111	0	3	1111 11

- An **inverse residue code** is similar to a residue code, but instead of appending the residue, the inverse residue **Q** is calculated and appended
  - Q = m – R
- Better fault detection for repeated faults
  - A repeated fault is encountered multiple times before the code is checked
  - Difficult to detect because subsequent effects can cancel the previous effects of the fault

67



Berger codes

TABLE 3.16 Berger code words for 4-bit information words

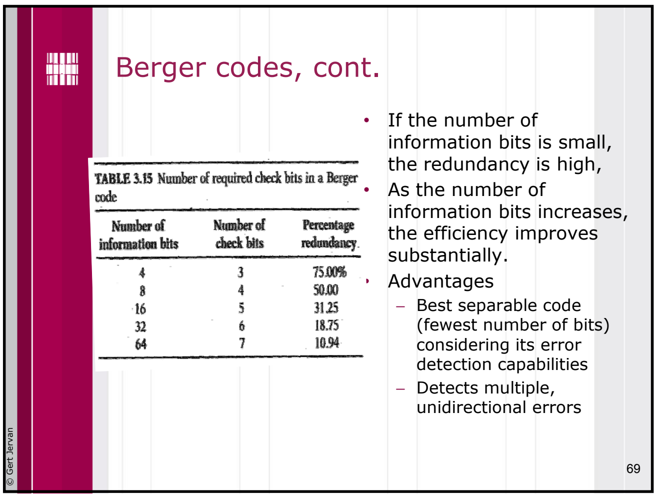
Original information	Berger code
0000	0000 111
0001	0001 110
0010	0010 110
0011	0011 101
0100	0100 110
0101	0101 101
0110	0110 101
0111	0111 100
1000	1000 110
1001	1001 101
1010	1010 101
1011	1011 100
1100	1100 101
1101	1101 100
1110	1110 100
1111	1111 011

- **Berger codes** are formed by appending a special set of bits, called check bits, to each word of information
- Berger code of length *n*
  - *I* information bits
  - *k* check bits

$$k = \lceil \log_2(I+1) \rceil$$
$$n = I + k$$

  - A code word is formed by first creating a binary number that corresponds to the number of 1s in the original *I* bits of information.
  - The resulting binary number is complemented and appended to the *I* information bits to form the (*I*+*k*)-bit code word.

68



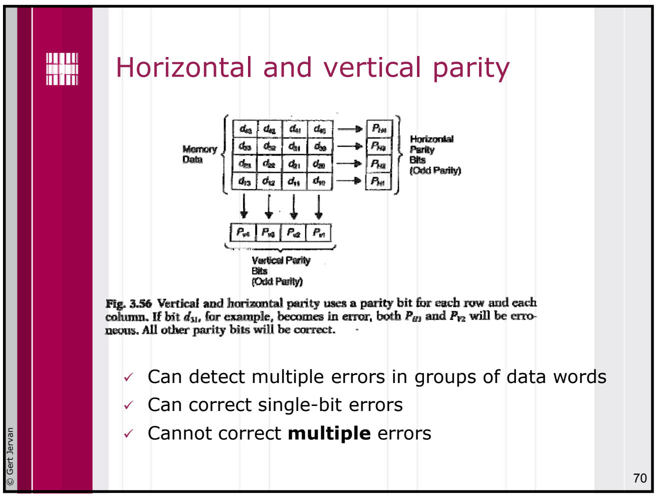
Berger codes, cont.

TABLE 3.15 Number of required check bits in a Berger code

Number of information bits	Number of check bits	Percentage redundancy
4	3	75.00%
8	4	50.00
16	5	31.25
32	6	18.75
64	7	10.94

- If the number of information bits is small, the redundancy is high,
- As the number of information bits increases, the efficiency improves substantially.
- Advantages
  - Best separable code (fewest number of bits) considering its error detection capabilities
  - Detects multiple, unidirectional errors

69



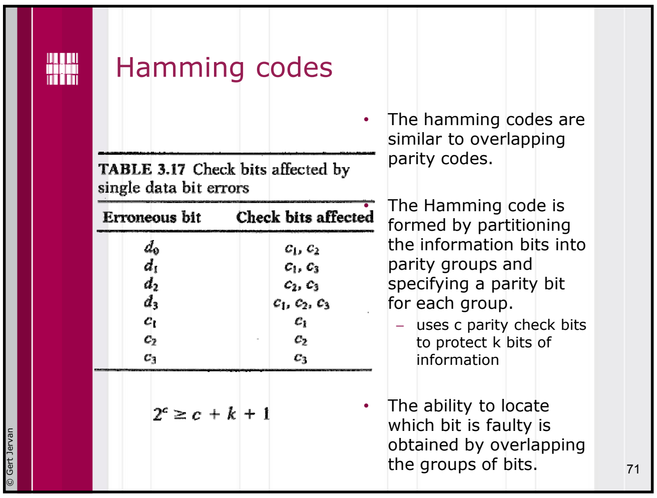
Horizontal and vertical parity

Memory Data
d_{43} d_{42} d_{41} d_{40}
d_{33} d_{32} d_{31} d_{30}
d_{23} d_{22} d_{21} d_{20}
d_{13} d_{12} d_{11} d_{10}
P_{H4} P_{H3} P_{H2} P_{H1}
Horizontal Parity Bits (Odd Parity)
P_{V4} P_{V3} P_{V2} P_{V1}
Vertical Parity Bits (Odd Parity)

Fig. 3.56 Vertical and horizontal parity uses a parity bit for each row and each column. If bit d_{31}, for example, becomes in error, both P_{H3} and P_{V2} will be erroneous. All other parity bits will be correct.

- ✓ Can detect multiple errors in groups of data words
- ✓ Can correct single-bit errors
- ✓ Cannot correct **multiple** errors

70



Hamming codes

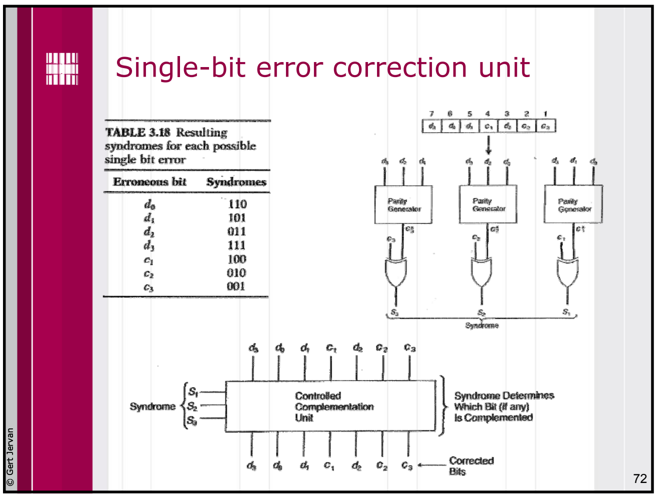
TABLE 3.17 Check bits affected by single data bit errors

Erroneous bit	Check bits affected
d_0	c_1, c_2
d_1	c_1, c_3
d_2	c_2, c_3
d_3	c_1, c_2, c_3
c_1	c_1
c_2	c_2
c_3	c_3

$$2^c \geq c + k + 1$$

- The hamming codes are similar to overlapping parity codes.
- The Hamming code is formed by partitioning the information bits into parity groups and specifying a parity bit for each group.
  - uses c parity check bits to protect k bits of information
- The ability to locate which bit is faulty is obtained by overlapping the groups of bits.

71



Single-bit error correction unit

TABLE 3.18 Resulting syndromes for each possible single bit error

Erroneous bit	Syndromes
d_0	110
d_1	101
d_2	011
d_3	111
c_1	100
c_2	010
c_3	001

7 6 5 4 3 2 1
d_3 d_0 d_1 c_1 d_2 c_2 c_3
d_3 d_2 d_1
d_3 d_2 d_0
d_3 d_1 d_0
Parity Generator
Parity Generator
Parity Generator
c_3^* c_3
c_2^* c_2
c_1^* c_1
S_3 S_2 S_1
Syndrome
d_3 d_0 d_1 c_1 d_2 c_2 c_3
Syndrome S_1 S_2 S_3
Controlled Complementation Unit
Syndrome Determines Which Bit (if any) Is Complemented
d_3 d_0 d_1 c_1 d_2 c_2 c_3
Corrected Bits

72

Hamming correction in memories

- Memory: 60 to 70% of the faults in a system
  - Transient faults are becoming much more prevalent as memory chips become denser
- Many memory designs use hamming error correction
  - Relatively inexpensive: 10-40% redundancy
  - Encoding and the decoding are fast
  - Error correction circuit is readily available on inexpensive chips

Hamming correction in memories, cont.

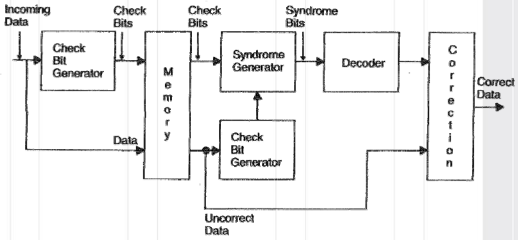
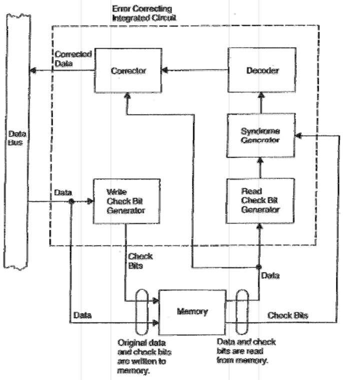


Fig. 3.59 Basic structure of a memory using Hamming single error correcting code.

Error correction in commercial circuits


Error Correcting Integrated Circuit
Corrected Data
Corrector
Decoder
Data Bus
Syndrome Generator
Data
Write Check Bit Generator
Read Check Bit Generator
Check Bits
Data
Data
Memory
Check Bits
Original data and check bits are written to memory.
Data and check bits are read from memory.

Code selection issues

- Fulfills the desired error detection/correction while maintaining costs at an acceptable level
  - Cost of a code
    - Redundancy required; time redundancy is also considered
  - Effectiveness of a code
    - Number of bit errors detected/corrected
- Key design decisions
  - Separable or not
  - What is required? Detection, correction, both?
  - Number of bit errors to be detected/corrected

Information redundancy

- Self-Checking
  - This is a form of hardware redundancy but often it is closely related to ECC techniques, therefore I have chosen to include it here
  - Assumptions: inputs are coded and outputs are coded
  - Objective: in the presence of a fault the circuit should either continue to provide correct output(s) or indicate by providing an error indication that there is a fault.
    - Clearly error indication can not be 1-bit output (why?)
    - With 2-bits output, 00 and 11 may indicate no failure
    - other output combinations (10, 01) may indicate a failure

Self-Checking (contd.)

- Example application
  - two devices produce identical outputs and we compare these outputs to check their equality
  - checker has two outputs encoded as follows
    - 00 equal
    - 11 unequal
    - 01 or 10 possible fault in the circuit
    - (we will discuss input encoding when we discuss an example of a 2-rail 1-bit checker)

Self-Checking (contd.)

- Definitions
  - a circuit is fault secure if in the presence of a fault, the output is either always correct, or not a code word for valid input code words
  - a circuit is self-testing if only valid inputs can be used to test it for the faults
  - a circuit is totally self-checking if it is fault secure and self-testing
- Example: a totally self-checking 2-rail 1-bit comparator
  - assumptions
    - 2 inputs and each input x is available as x and its complement
    - x and its complement are independently generated
    - note with these assumption the input space is encoded (4 valid inputs out of 16 possible inputs)
    - single stuck-at fault model

Time redundancy

Time redundancy

- Disadvantages of hardware and information redundancy
  - Require large amounts of extra hardware for their implementation.
- Time redundancy: reduce the extra hardware at the expense of using additional time.
  - Hardware is a physical entity that impacts weight, size, power consumption and cost.
  - Time may be readily available in some applications.

Time redundancy

- Key Concept - do a job more than once over time
  - examples
    - re-execution
    - re-transmission of information
  - different faults and capabilities of different schemes
    - transient faults
      - re-execution and re-transmission can detect such faults provided we wait for transient to subside
    - permanent faults
      - simple re-execution or re-transmission will not work. Possible solutions
        - send or process shifted version of data
        - send or process complemented data during second transmission

Time redundancy

- Different faults and capabilities of different schemes (contd.)
  - faults in ALU
    - re-execution with complement or shifted version can detects permanent and transient faults
    - (RESO concept - re-computation with shifted operands)
  - multiple re-computations
    - can detect and possibly correct transient and permanent faults if properly employed/designed

Time redundancy: transient fault detection

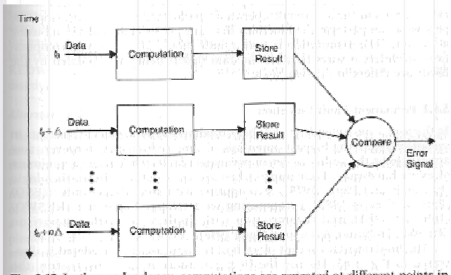
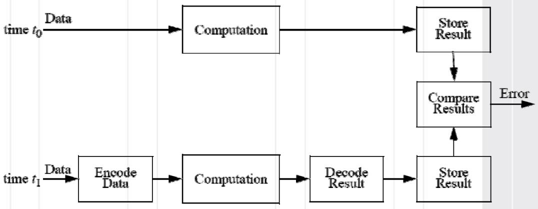


Fig. 3.62 In time redundancy, computations are repeated at different points in time and then compared.

Time redundancy: permanent fault detection



© Gert Jervan

85

Alternating logic

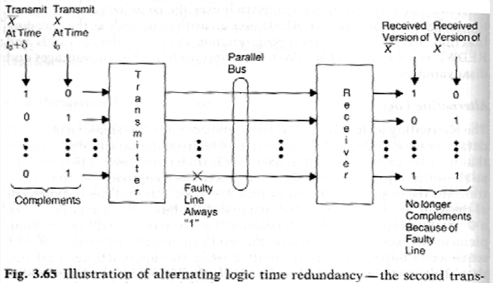

Transmit $\overline{X}$ At Time $t_0+\delta$
Transmit X At Time t_0
Complements
Transmitter
Parallel Bus
Faulty Line Always "1"
Receiver
Received Version of $\overline{X}$
Received Version of X
No longer Complements Because of Faulty Line

Fig. 3.65 Illustration of alternating logic time redundancy — the second transmission is the complement of the first.

© Gert Jervan

86

Error correction using time-redundancy

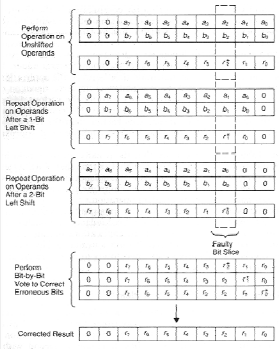

Perform Operation on Unshifted Operands
Repeat Operation on Operands After a 1-Bit Left Shift
Repeat Operation on Operands After a 2-Bit Left Shift
Faulty Bit Slice
Perform Bit-by-Bit Vote to Correct Erroneous Bits
Corrected Result

Fig. 3.75 Example of error correction using time redundancy.

© Gert Jervan

87

Environment Diversity

- To diversify the software operating circumstance temporarily.
- The typical examples of environment diversity technique are progressive retry, rollback, rollforward, recovery with checkpointing, restart, hardware reboot, etc.

© Gert Jervan

88

Adjudicators

- Voter
- AT
- Hybrid, other

© Gert Jervan

89

What's new

- Modifications to Traditional Techniques
  - Adaptive N-version Systems
  - Fuzzy Voting
  - Abstraction
  - Parallel Graph Reduction
- New Concepts (Not Classifiable in Data or Design Diversity)
  - Rejuvenation

© Gert Jervan

90

Why Adaptive

- Defective components should be removed from the voting process
- The voting procedure can be adaptively modified and tailored to the fault state of the overall system

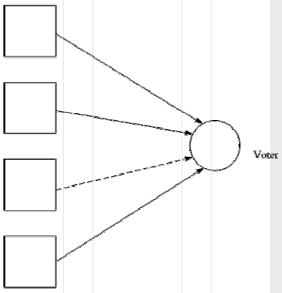
An Adaptive Approach for n-Version Systems

- Model and manage different quality levels of the versions by introducing an individual weight factor to each version of the n-version system.
- This weight factor is then included in the voting procedure, i.e. the voting is based on a weighted counting.

Voting schemes

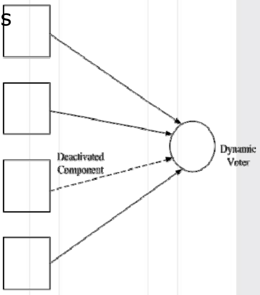
- Static voting:

  fixed number of versions



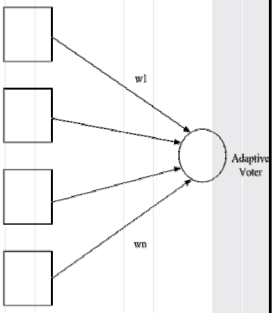
Voting schemes

Dynamic voting: defective components are removed from the voting process



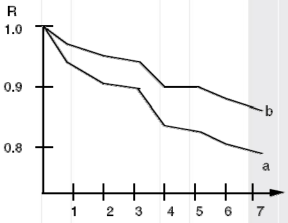
Voting schemes

Adaptive voting: versions have dynamically changeable weight factors



Example

- Cumulated reliability of 7 components of a software system for a satellite control application
  - a) classical 3-version system
  - b) 3-version system with adaptive weight factors



Why Fuzzy Voting

- In traditional voting, equality relation regards two real numbers as equal if their difference is smaller than fixed tolerance ε. For different version outputs that are "closer" to each other than the fixed threshold there is no gradual comparison. As a result, certain interconnection of faults could incur incorrect selection.
- Fuzzy equivalence relation results in more reliable systems

© Gert Jervan

Fuzzy Equality Equation

- Traditional Equality Equation

$$r_{i,j} = \begin{cases} 1, & if\ |x_i - x_j| \le \varepsilon \\ 0, & otherwise \end{cases}$$

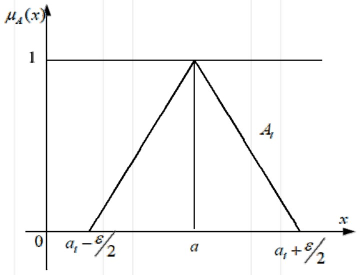
- Fuzzy Equality Equation

$$\mu_{A_i}(x_i) = \begin{cases} 1 - \frac{|a_i - x_i|}{\varepsilon/2}, & if\ |x_i - a_i| \le \varepsilon/2 \\ 0, & otherwise \end{cases}$$

© Gert Jervan

Output of Fuzzy Sets (Triangular Shape)

- The fuzzy logic maps the input vector into an output nonlinearly



© Gert Jervan

Why Abstraction Improve Fault Tolerance

- Reduce the cost of fault tolerance
- Improve its ability to mask software errors.

© Gert Jervan

An Example of Abstraction: BASE

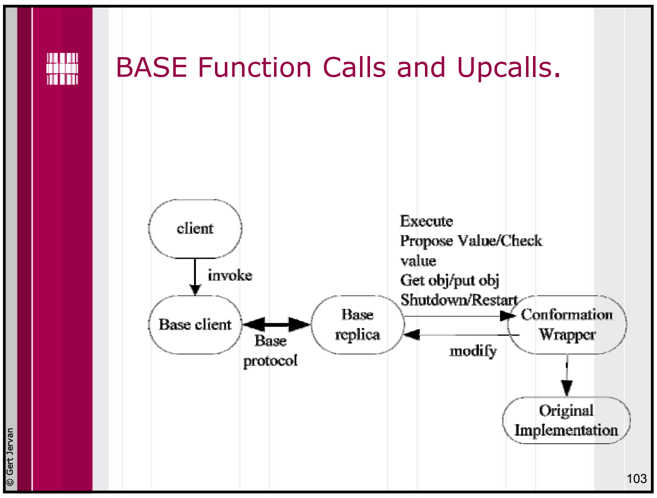
- Byzantine fault tolerance (BFT) with Abstract Specification Encapsulation (BASE)

© Gert Jervan

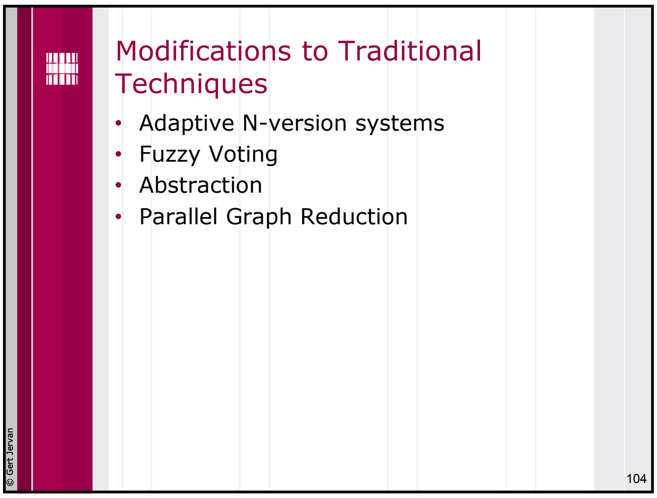
How BASE Works

- BASE reduces cost because it enables reuse of off-the-shelf service implementations.
- It improves availability because each replica can be repaired periodically using an abstract view of the state stored by correct replicas, and because each replica can run distinct or nondeterministic service implementations, which reduces the probability of common mode failures.

© Gert Jervan

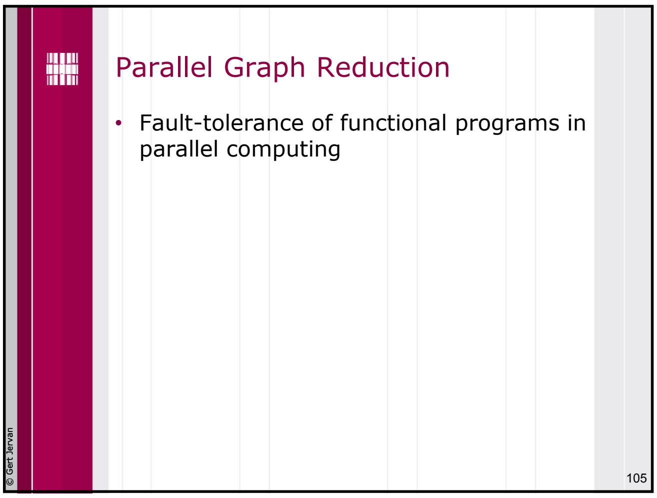


BASE Function Calls and Upcalls.



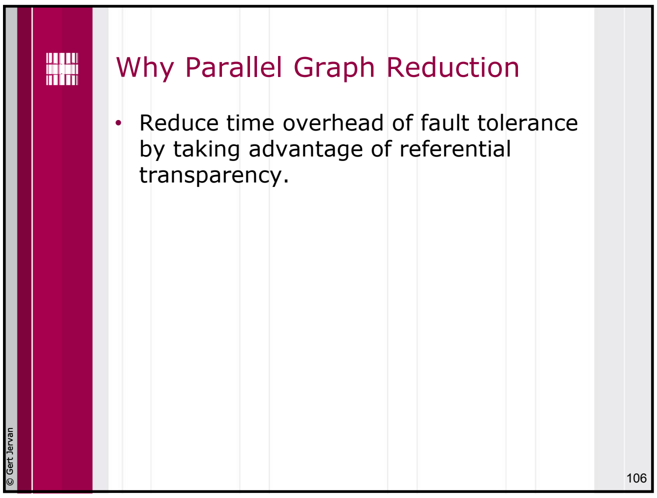
Modifications to Traditional Techniques

- Adaptive N-version systems
- Fuzzy Voting
- Abstraction
- Parallel Graph Reduction



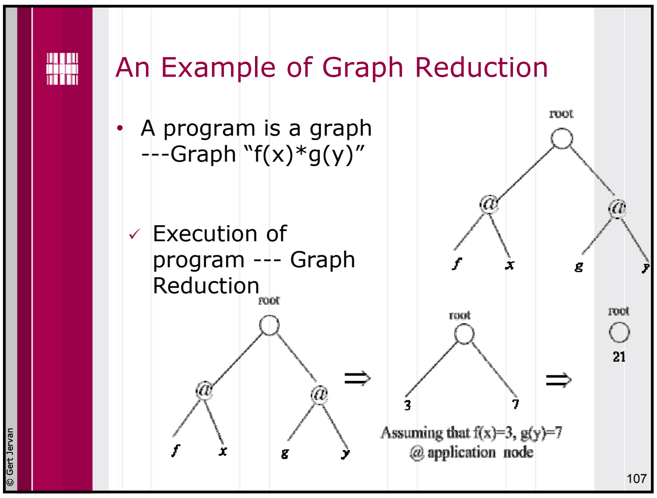
Parallel Graph Reduction

- Fault-tolerance of functional programs in parallel computing



Why Parallel Graph Reduction

- Reduce time overhead of fault tolerance by taking advantage of referential transparency.

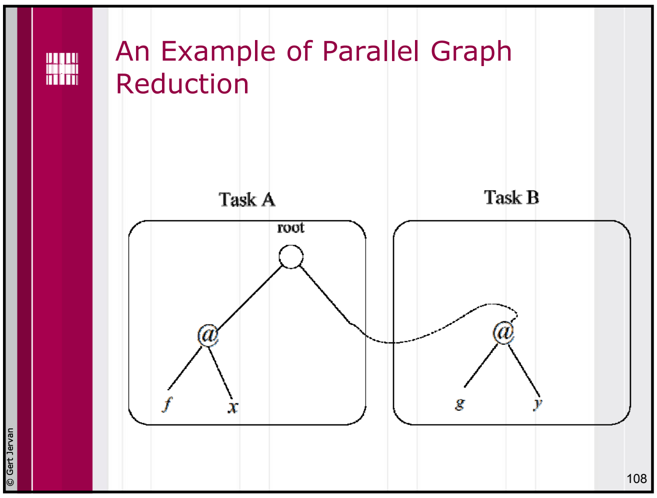


An Example of Graph Reduction

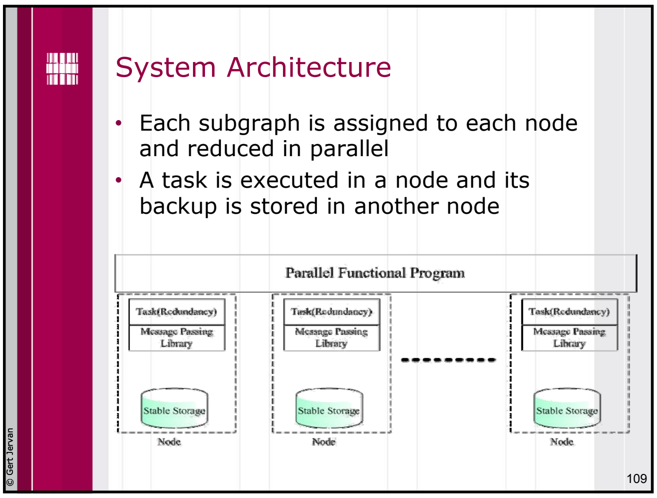
- A program is a graph ---Graph “f(x)*g(y)”

✓ Execution of program --- Graph Reduction

Assuming that f(x)=3, g(y)=7
@ application node

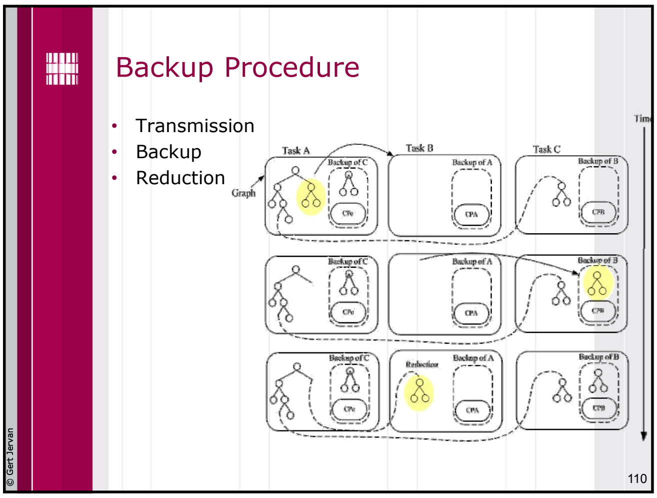


An Example of Parallel Graph Reduction



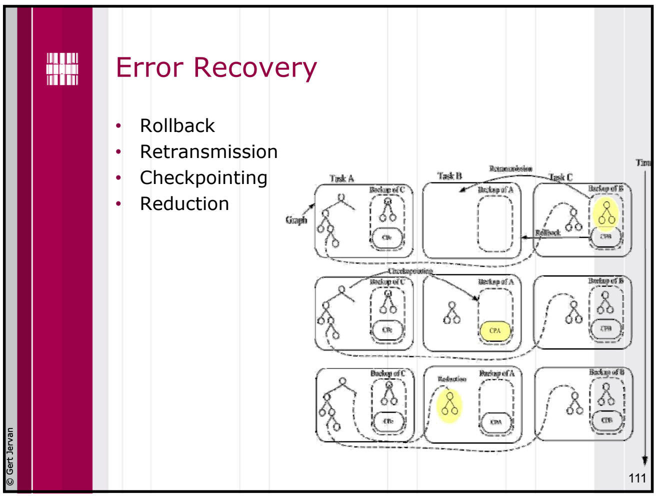
System Architecture

- Each subgraph is assigned to each node and reduced in parallel
- A task is executed in a node and its backup is stored in another node



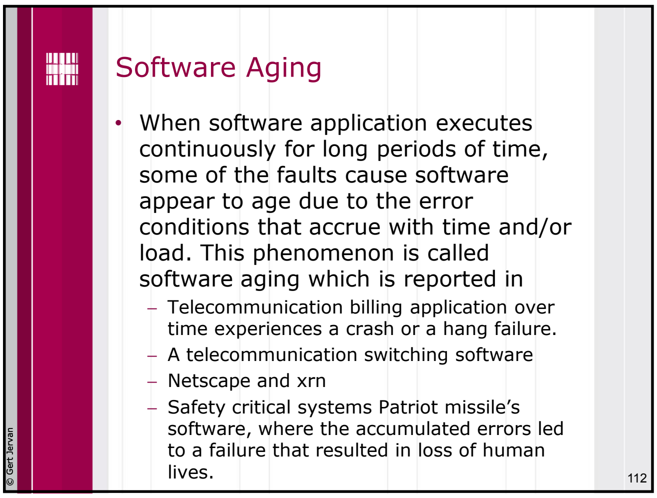
Backup Procedure

- Transmission
- Backup
- Reduction



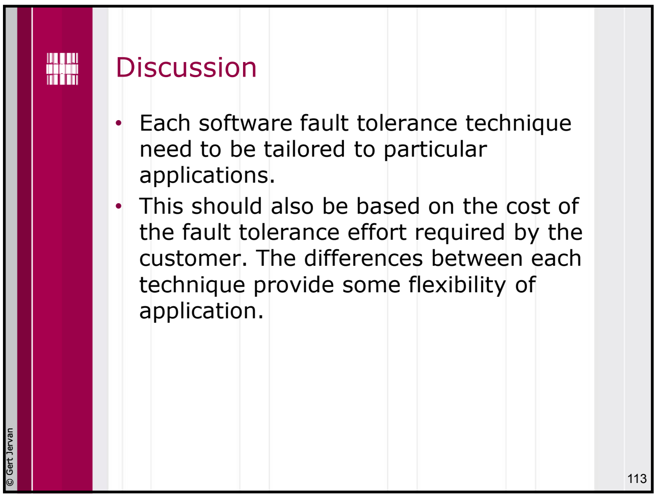
Error Recovery

- Rollback
- Retransmission
- Checkpointing
- Reduction



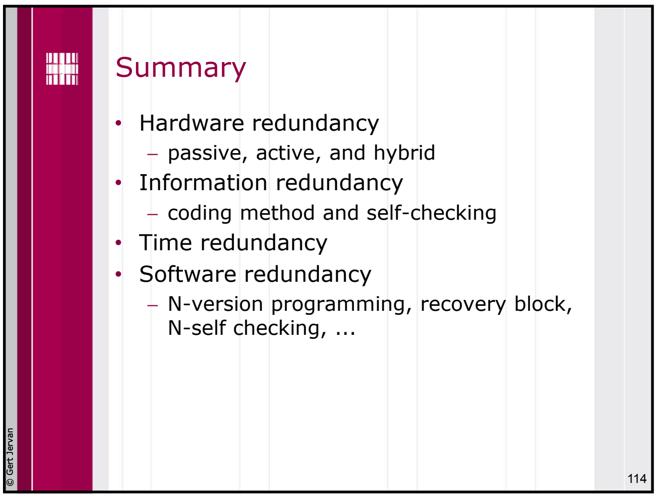
Software Aging

- When software application executes continuously for long periods of time, some of the faults cause software appear to age due to the error conditions that accrue with time and/or load. This phenomenon is called software aging which is reported in
  - Telecommunication billing application over time experiences a crash or a hang failure.
  - A telecommunication switching software
  - Netscape and xrn
  - Safety critical systems Patriot missile's software, where the accumulated errors led to a failure that resulted in loss of human lives.



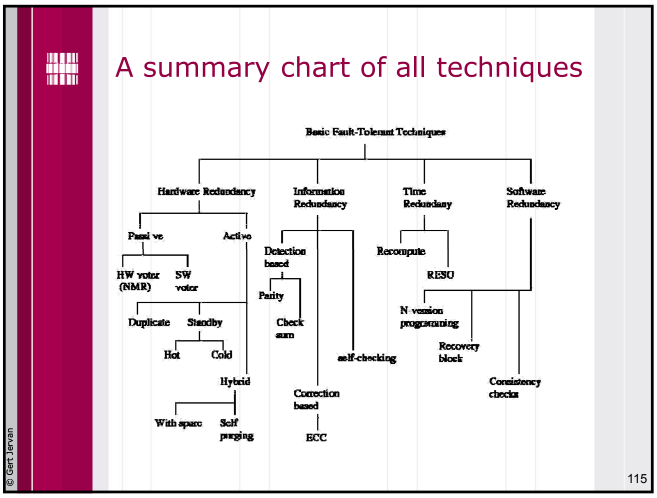
Discussion

- Each software fault tolerance technique need to be tailored to particular applications.
- This should also be based on the cost of the fault tolerance effort required by the customer. The differences between each technique provide some flexibility of application.



Summary

- Hardware redundancy
  - passive, active, and hybrid
- Information redundancy
  - coding method and self-checking
- Time redundancy
- Software redundancy
  - N-version programming, recovery block, N-self checking, ...



A summary chart of all techniques

Basic Fault-Tolerant Techniques

Hardware Redundancy

Passi ve

HW voter (NMR)

SW voter

Active

Duplicate

Standby

Hot

Cold

Hybrid

With spare

Self purging

Information Redundancy

Detection based

Parity

Check sum

self-checking

Correction based

ECC

Time Redundany

Recompute

RESO

Software Redundancy

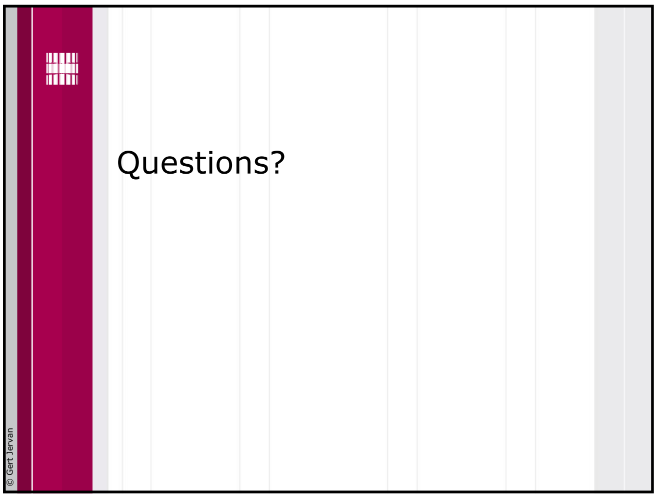
N-version programming

Recovery block

Consistency checks

© Gert Jervan

115



Questions?

© Gert Jervan