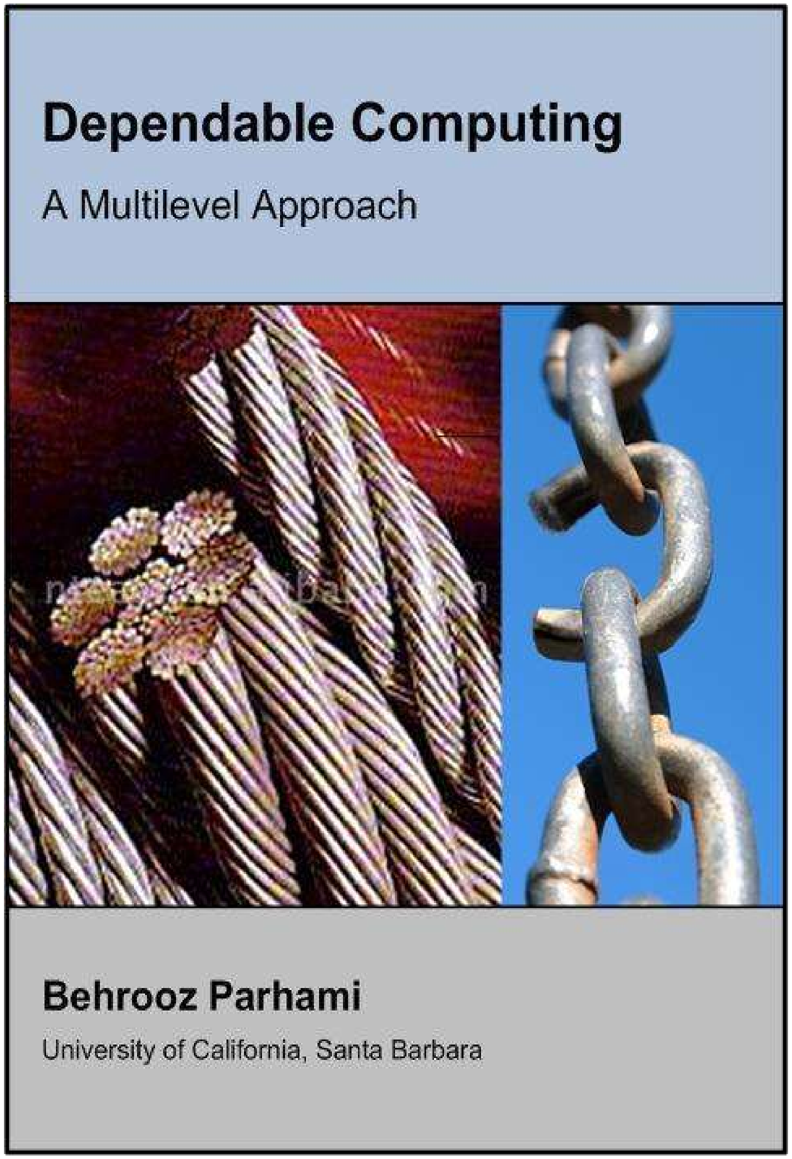


Dependable Computing

A Multilevel Approach

Behrooz Parhami

University of California, Santa Barbara

STRUCTURE AT A GLANCE

Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Resilient Algorithms
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods Examples	21. Degradation Allowance 22. Degradation Management 23. Robust Task Scheduling 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

Appendix: Past, Present, and Future

About This Presentation

This presentation is intended to support the use of the textbook *Dependable Computing: A Multilevel Approach* (traditional print or on-line open publication, TBD). It is updated regularly by the author as part of his teaching of the graduate course ECE 257A, Fault-Tolerant Computing, at Univ. of California, Santa Barbara. Instructors can use these slides freely in classroom teaching or for other educational purposes. Unauthorized uses, including distribution for profit, are strictly prohibited.

Edition	Released	Revised	Revised	Revised	Revised
First	Sep. 2006	Oct. 2007	Oct. 2009	Oct. 2012	Oct. 2013
		Feb. 2015	Oct. 2015	Oct. 2018	Oct. 2019
		Nov. 2020			

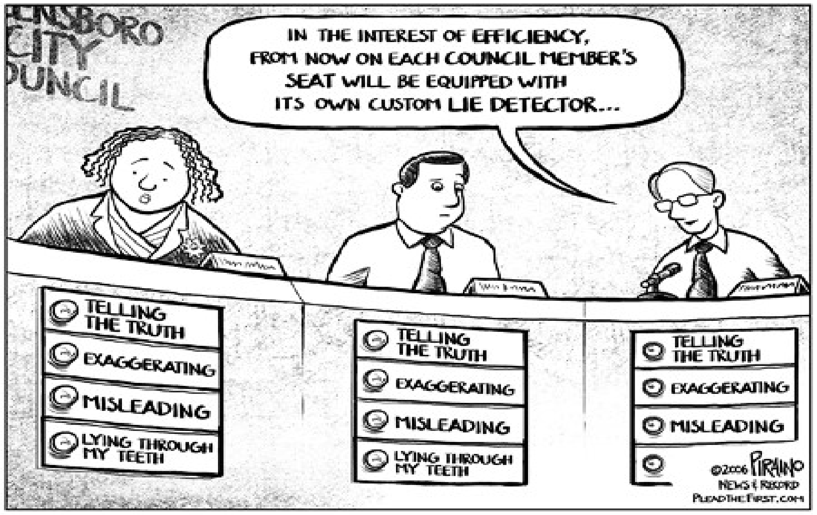
Error Detection

A Runtime Error has occurred.
Do you wish to Debug?

Line: 1
Error: Syntax error

"Just among us we goofed. But officially it will go down as computer error."

No results for Honalulu
No results for Honoloulou
No results for Hawaai
OH, FORGET IT. LET'S JUST GO VISIT MY MOTHER IN FARGO.

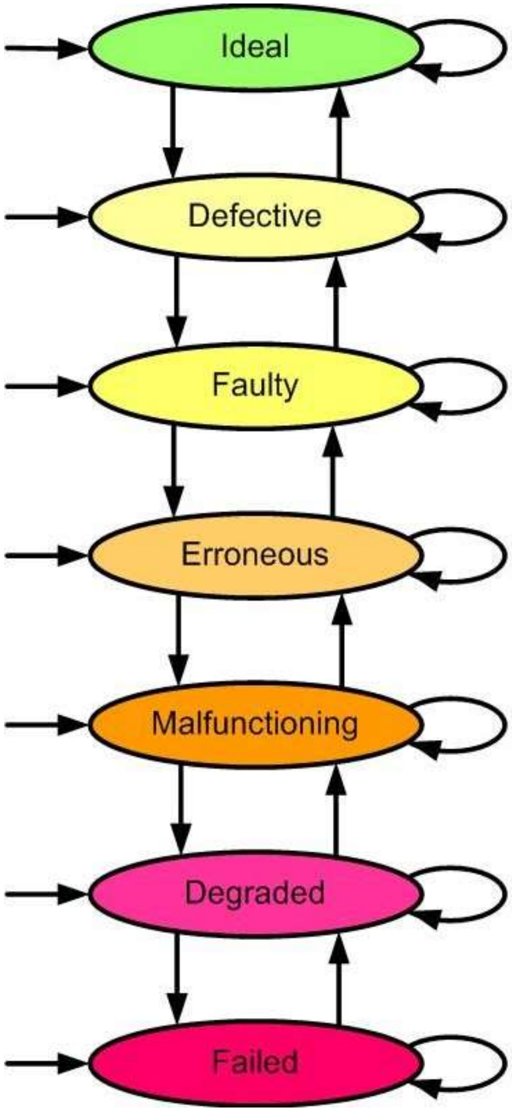


"We rarely back up our data. We'd rather not keep a permanent record of everything that goes wrong around here!"

STRUCTURE AT A GLANCE

Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Resilient Algorithms
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods Examples	21. Degradation Allowance 22. Degradation Management 23. Robust Task Scheduling 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

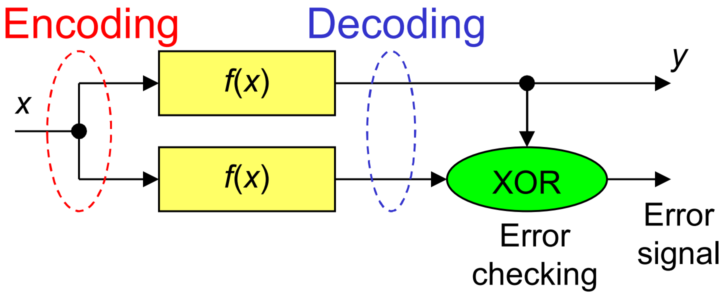
Appendix: Past, Present, and Future



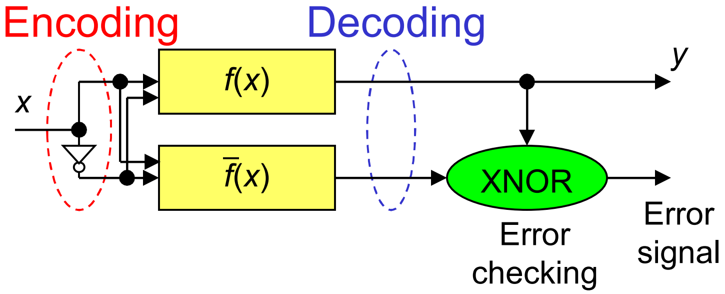
13.1 Basics of Error Detection

High-redundancy codes

Duplication is a form of error coding:
x represented as xx (100% redundancy)
Detects any error in one version

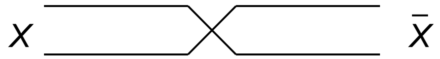


Two-rail encoding
x represented as $x\bar{x}$ (100% redundancy)
e.g., 0 represented as 01; 1 as 10
Detects any error in one version
Detects all unidirectional errors



Two-rail logic, with each input having a true bit and a complement bit

AND: $(t_1, c_1)\,(t_2, c_2) = (t_1 t_2,\ c_1 \vee c_2)$
OR: $(t_1, c_1) \vee (t_2, c_2) = (t_1 \vee t_2,\ c_1 c_2)$
NOT: $(t, c)' = (c, t)$
XOR: $(t_1, c_1) \oplus (t_2, c_2) = (t_1 c_2 \vee t_2 c_1,\ t_1 t_2 \vee c_1 c_2)$

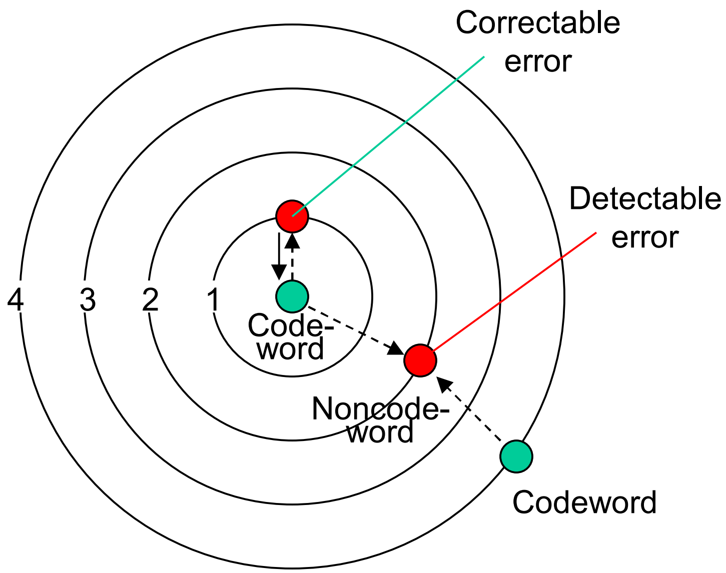


Hamming Distance

Definition: Hamming distance between two bit-vectors is the number of positions in which they differ

A distance-2 code:

00011
00101
00110 → 00111 (0→1 error)
       → 00100 (1→0 error)
01001
01010
01100
10001
10010
10100
11000



Min H-dist	Code capability
2	d = 1; SED
3	c = 1; SEC or (d = 2; DED)
4	c = 1 and d = 2; SEC/DED
5	c = 2 or (c = 1 and d = 3; SEC/3ED)
h	cEC/dED such that $h = c + d + 1$

$d \geq c$, so that $d - c$ represents the add'l detection capability

Error Classification and Models

Goal of error tolerance methods:

Allow uninterrupted operation despite presence of certain errors
Error model – Relationship between errors and faults (or other causes)

Errors are detected/corrected through:

Encoded (redundant) data, plus code checkers
Reasonableness checks, activity monitoring, retry

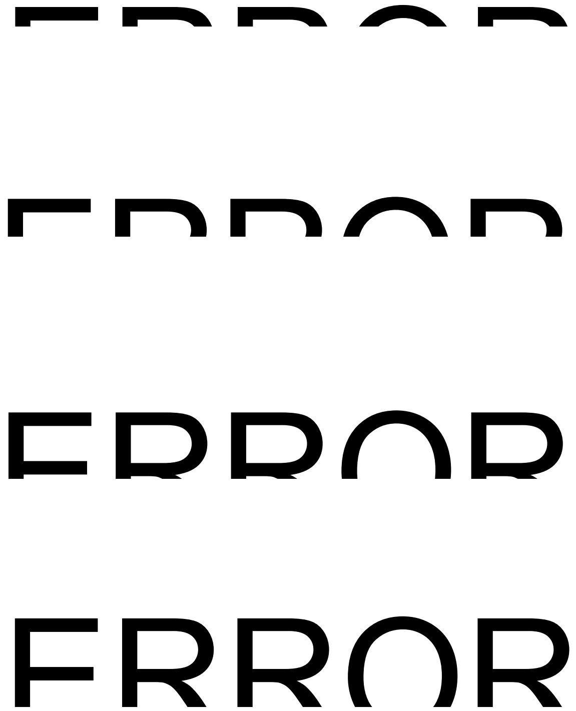
Errors are classified as:

Single or Multiple (according to the number of bits affected)
Inversion or Erasure (symbol or bit changed or lost)*
Random or Correlated (correlation in the form of byte or burst error)
Symmetric or Asymmetric (regarding 0 → 1 and 1 → 0 inversions)

* Nonbinary codes have substitution rather than inversion errors
Also of interest for nonelectronic systems are transposition errors

Errors are permanent by nature; transient faults, not transient errors

Error Detection in Natural Language Texts



ERROR

E□R□R

Erasure errors

EQROR

Substitution error

ERORR

Transposition error

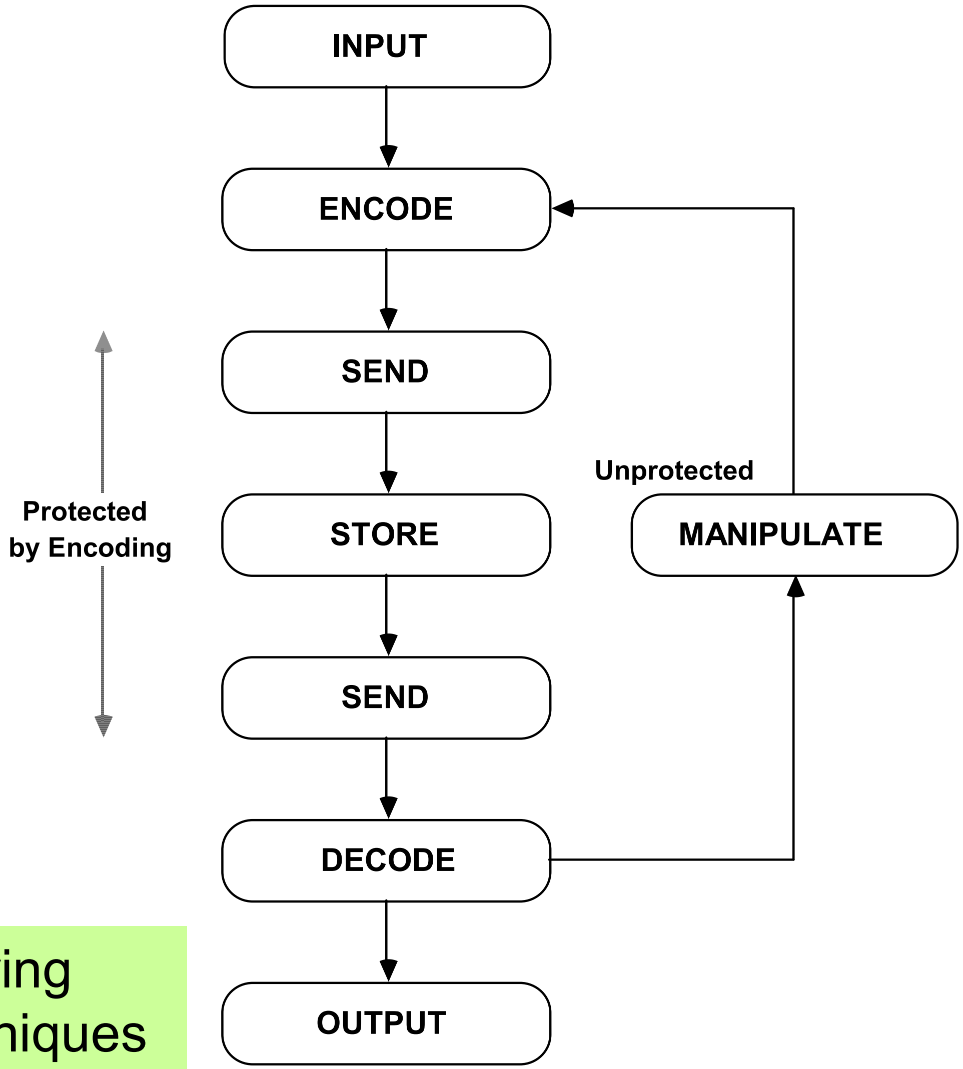
Application of Coding to Error Control

Ordinary codes can be used for storage and transmission errors; they are not closed under arithmetic / logic operations

Error-detecting, error-correcting, or combination codes (e.g., Hamming SEC/DED)

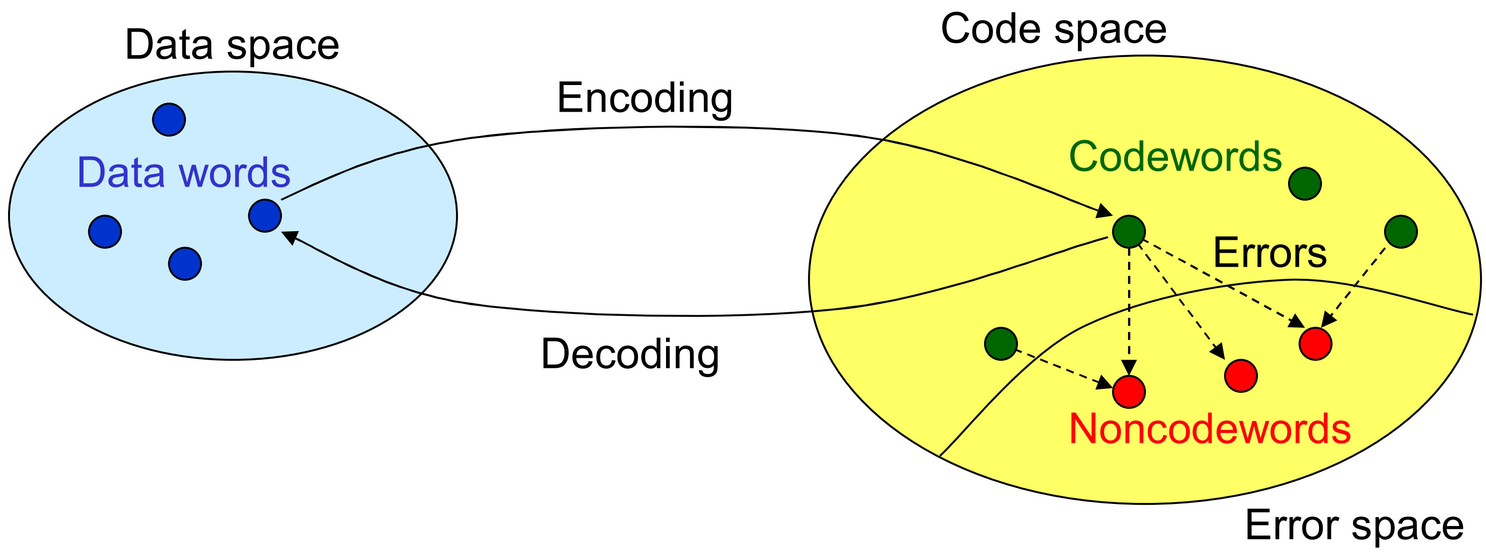
Arithmetic codes can help detect (or correct) errors during data manipulations:

1. Product codes (e.g., 15*x*)
2. Residue codes (*x* mod 15)



A common way of applying information coding techniques

The Concept of Error-Detecting Codes



The simplest possible error-detecting code:

Attach an even parity bit to each *k*-bit data word

Check bit = XOR of all data bits

Data space: All 2^k possible *k*-bit words

Code space: All 2^k possible even-parity (*k* + 1)-bit codewords

Error space: All 2^k possible odd-parity (*k* + 1)-bit noncodewords

Detects all single-bit errors

0 0 1 0 1 0 $\overset{1}{\cancel{0}}$ 0 1 1

Evaluation of Error-Detecting Codes

Redundancy: k data bits encoded in $n = k + r$ bits (r redundant bits)

Encoding: Complexity (cost / time) to form codeword from data word

Decoding: Complexity (cost / time) to obtain data word from codeword
Separable codes have computation-free decoding

Capability: Classes of error that can be detected
Greater detection capability generally involves more redundancy
To detect d bit-errors, a minimum code distance of $d + 1$ is required

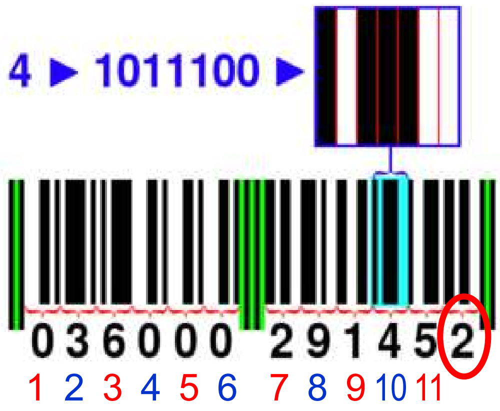
Examples of code detection capabilities:
Single, double, *b*-bit burst, byte, unidirectional, . . . errors

Closure: Arithmetic and other operations done directly on codewords
(rather than in 3 stages: decode, operate, and encode)

13.2 Checksum Codes

Ex.: 12-digit UPC-A universal product code—Computing the check digit:
Add the odd-indexed digits and multiply the sum by 3
Add the sum of even-indexed digits to previous result
Subtract the total from the next higher multiple of 10

Example:
Sum odd indexed digits: 0 + 6 + 0 + 2 + 1 + 5 = 14
Multiply by 3: 14 × 3 = 42
Add even-indexed digits: 42 + 3 + 0 + 0 + 9 + 4 = 58
Compute check digit: 60 – 58 = 2



Bar code uses 7 bits per digit, with different encodings on the right and left halves and different parities at various positions

Checking:
Verify that weighted mod-10 sum of all 12 digits is 0

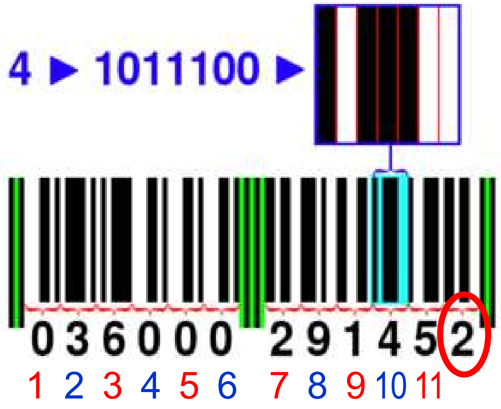
Capabilities:
Detects all single-digit errors
Detects most, but not all, transposition errors

Characterization of Checksum Codes

Given a data vector $x_1, x_2, \ldots, x_n$, encode the data by attaching the checksum x_{n+1} to the end, such that $\Sigma_{j=1 \text{ to } n+1}\, w_j x_j = 0 \bmod A$

The elements w_j of the weight vector w are predetermined constants

Example:
For the UPC-A checksum scheme, we have
$w = 3, 1, 3, 1, 3, 1, 3, 1, 3, 1, 3, 1$
$A = 10$



Checking:
Verify that weighted mod-A sum of all elements is 0

Capabilities:
Detects all errors adding an error magnitude that is not a multiple of A

Variant: Vector elements may be XORed rather than added together

13.3 Weight-Based and Berger Codes

Constant-weight codes

Definition: All codewords have the same number of 1s

A weight-2 code:
00011
00101
00110
01001
01010
01100
10001
10010
10100
11000

Can detect all unidirectional errors

Maximum number of codewords obtained when weight of *n*-bit codewords is *n*/2

Berger Codes

Definition: Separable code that has the count of 0s within the data part attached as a binary number that forms the check part

Alternative – attach the 1's-complement of the number of 1s

A Berger code:

000000	110
000001	101
000010	101
000011	100
. . .	
100111	010
101000	100
. . .	
111110	001
111111	000

Check part

Can detect all unidirectional errors

$\lceil \log_2(k + 1) \rceil$ check bits for k data bits

Jay M. Berger
(IBM)

13.4 Cyclic Codes

Definition: Any cyclic shift of a codeword produces another codeword

A k-bit data word corresponds to a polynomial of degree $k - 1$

Data = 1101001: $D(x) = 1 + x + x^3 + x^6$ (addition is mod 2)

The code has a generator polynomial of degree $r = n - k$

$G(x) = 1 + x + x^3$

To encode data (1101001), multiply its associated polynomial by $G(x)$

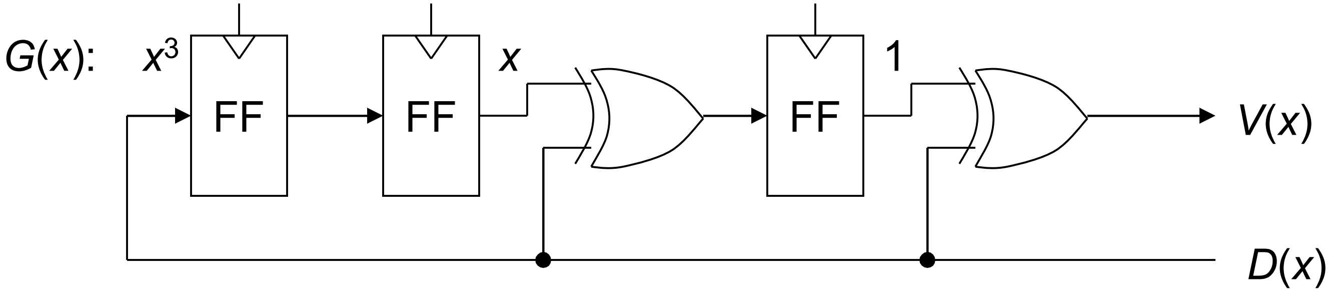
$$\begin{array}{ll} & 1 + x + x^3 + x^6 \\ \times & \underline{1 + x + x^3} \\ & 1 + \cancel{x} + \cancel{x^3} + \cancel{x^6} + \cancel{x} + x^2 + \cancel{x^4} + x^7 + \cancel{x^3} + \cancel{x^4} + \cancel{x^6} + x^9 \\ & 1 + x^2 \quad + \quad x^7 + x^9 \\ & 1\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 1 \end{array}$$

Detects all burst errors of width less than $n - k$

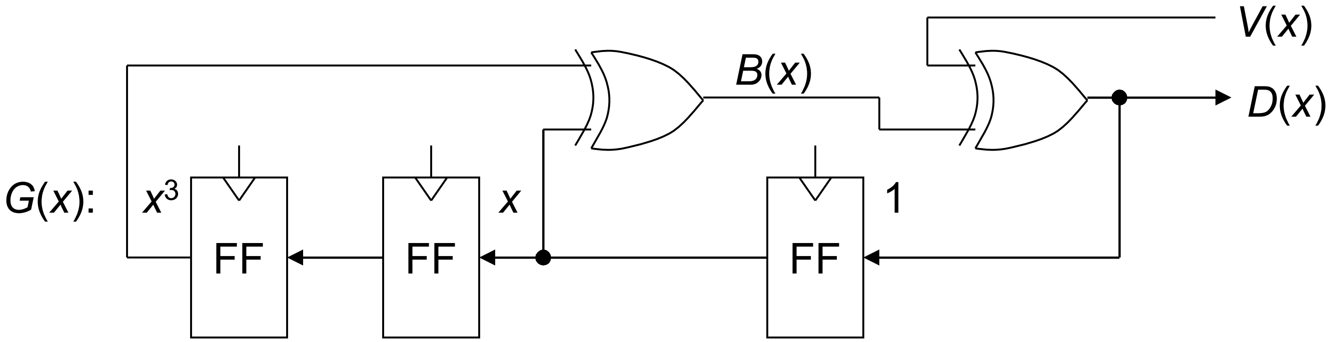
Burst error polynomial $x^j E(x)$, where $E(x)$ is of degree less than $n - k$

Cyclic Codes: Encoding and Decoding

Encoding: Multiplication by the generator polynomial *G*(*x*)



Decoding: Division by the generator polynomial *G*(*x*)



$B(x) = (x + x^3)\, D(x)$ $V(x) = D(x) + B(x) = (1 + x + x^3)\, D(x)$

Separable Cyclic Codes

Let $D(x)$ and $G(x)$ be the data and generator polynomials

Encoding:

Multiply $D(x)$ by x^{n-k} and divide the result by $G(x)$ to get the remainder polynomial $R(x)$ of degree less than $n - k$

Form the codeword $V(x) = R(x) + x^{n-k}D(x)$, which is divisible by $G(x)$

Example: 7-bit code with 4 data bits and 3 check bits, $G(x) = 1 + x + x^3$

Data = 1 0 0 1, $D(x) = 1 + x^3$

$x^3D(x) = x^3 + x^6 = (x + x^2) \bmod (1 + x + x^3)$

$V(x) \quad = \quad x + x^2 + x^3 \quad + \quad x^6$

$$\text{Codeword} = \underbrace{0 \quad 1 \quad 1}_{\text{Check part}} \quad \underbrace{1 \quad 0 \quad 0 \quad 1}_{\text{Data part}}$$

aka CRC = cyclic redundancy check

Single parity bit:
$G(x) = x + 1$

13.5 Arithmetic Error-Detecting Codes

```
Unsigned addition        0010 0111 0010 0001
                       + 0101 1000 1101 0011
                         -------------------
Correct sum              0111  1111  1111  0100
Erroneous sum            1000 0000 0000 0100
                                       ↑
                             Stage generating an
                             erroneous carry of 1
```

How a single carry error can lead to an arbitrary number of bit-errors (inversions)

The *arithmetic weight* of an error: Min number of signed powers of 2 that must be added to the correct value to turn it into the erroneous result (contrast with Hamming weight of an error)

	Example 1	Example 2
Correct value	0111 1111 1111 0100	1101 1111 1111 0100
Erroneous value	1000 0000 0000 0100	0110 0000 0000 0100
Difference (error)	16 = 2^4	–32752 = $-2^{15} + 2^4$
Min-weight BSD	0000 0000 0001 0000	$^{-}$1000 0000 0001 0000
Arithmetic weight	1	2
Error type	Single, positive	Double, negative

Codes for Arithmetic Operations

Arithmetic error-detecting codes:

Are characterized by arithmetic weights of detectable errors

Allow direct arithmetic on coded operands

We will discuss two classes of arithmetic error-detecting codes, both of which are based on a check modulus A (usually a small odd number)

Product or AN codes
Represent the value N by the number AN

Residue (or inverse residue) codes
Represent the value N by the pair (N, C),
where C is N mod A or (N – N mod A) mod A

Product or *AN* Codes

For odd *A*, all weight-1 arithmetic errors are detected

Arithmetic errors of weight ≥ 2 may go undetected

e.g., the error 32 736 = $2^{15} - 2^{5}$ undetectable with *A* = 3, 11, or 31

Error detection: check divisibility by *A*

Encoding/decoding: multiply/divide by *A*

Arithmetic also requires multiplication and division by *A*

Product codes are *nonseparate* (*nonseparable*) codes
Data and redundant check info are intermixed

Low-Cost Product Codes

Use low-cost check moduli of the form $A = 2^a - 1$

Multiplication by $A = 2^a - 1$: done by shift-subtract
$(2^a - 1)N = 2^a N - N$

Division by $A = 2^a - 1$: done a bits at a time as follows

Given $y = (2^a - 1)x$, find x by computing $2^a x - y$

. . . xxxx 0000	–	. . . xxxx xxxx	=	. . . xxxx xxxx
Unknown $2^a x$		Known $(2^a - 1)x$		Unknown x

Theorem: Any unidirectional error with arithmetic weight of at most $a - 1$ is detectable by a low-cost product code based on $A = 2^a - 1$

Arithmetic on *AN*-Coded Operands

Add/subtract is done directly: $Ax \pm Ay = A(x \pm y)$

Direct multiplication results in: $Aa \times Ax = A^2ax$

The result must be corrected through division by A

For division, if $z = qd + s$, we have: $Az = q(Ad) + As$

Thus, q is unprotected
Possible cure: premultiply the dividend Az by A
The result will need correction

Square rooting leads to a problem similar to division

$\lfloor \sqrt{A^2x} \rfloor = \lfloor A\sqrt{x} \rfloor$ which is not the same as $A\lfloor \sqrt{x} \rfloor$

Residue and Inverse Residue Codes

Represent N by the pair $(N, C(N))$, where $C(N) = N \bmod A$

Residue codes are *separate* (*separable*) codes

Separate data and check parts make decoding trivial

Encoding: Given N, compute $C(N) = N \bmod A$

Low-cost residue codes use $A = 2^a - 1$

To compute $N \bmod (2^a - 1)$, add a-bit segments of N, modulo $2^a - 1$ (no division is required)

Example: Compute 0101 1101 1010 1110 mod 15
0101 + 1101 = 0011 (addition with end-around carry)
0011 + 1010 = 1101
1101 + 1110 = 1100 The final residue mod 15

Arithmetic on Residue-Coded Operands

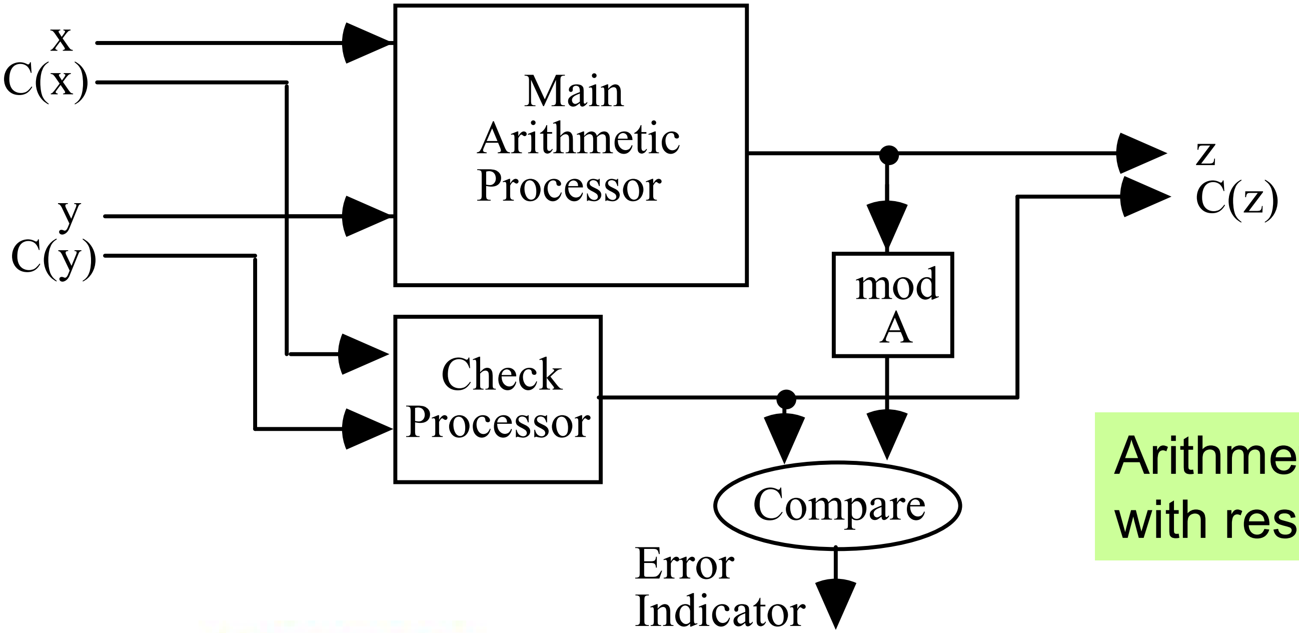
Add/subtract: Data and check parts are handled separately

$$(x, C(x)) \pm (y, C(y)) \;=\; (x \pm y, (C(x) \pm C(y)) \bmod A)$$

Multiply

$$(a, C(a)) \times (x, C(x)) \;=\; (a \times x, (C(a)\times C(x)) \bmod A)$$

Divide/square-root: difficult



Arithmetic processor with residue checking

13.6 Other Error-Detecting Codes

Codes for erasure errors

Assume *n* total symbols, *k* info symbol, *n* – *m* erasures allowed
Info can be recovered from any *m* symbols in an *n*-symbol codeword
When *m* = *k*, the erasure code is optimal

Codes for byte errors

Bytes are common units of data representation, storage, transmission
So, it makes sense to tie our error detection capability to bytes
Example: Single-byte-error-correcting, double-byte-error-detecting code

Codes for burst errors

With serial data or scratched disk surface, adjacent bits can be affected
Example: Single-bit-error-correcting, 6-bit-burst-error-detecting code

Higher-Level Error Coding Methods

We have applied coding to data at the bit-string or word level

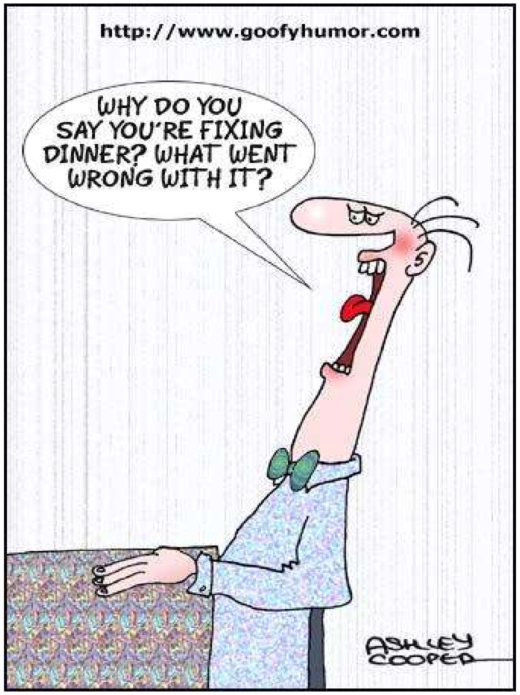
It is also possible to apply coding at higher levels

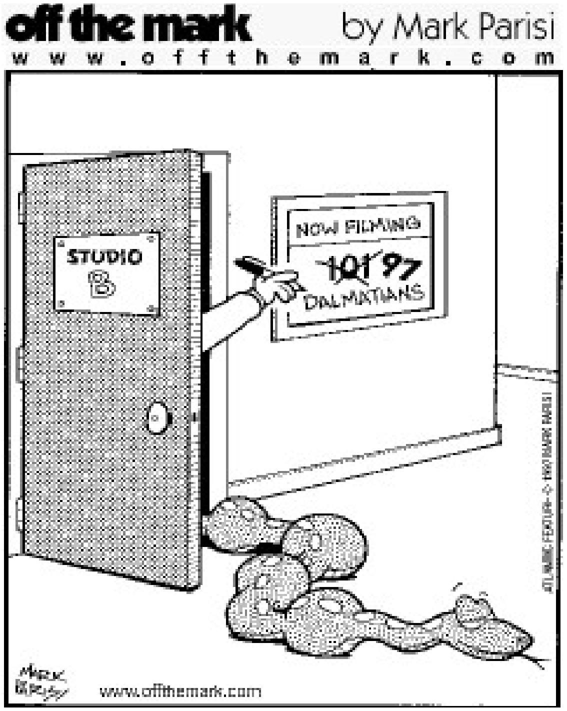
Data structure level – Robust data structures

Application level – Algorithm-based error tolerance

Error Correction

off the mark by Mark Parisi
www.offthemark.com
AH, HERE'S THE PROBLEM! WE'RE USING THE "VIRTUAL DENTIST" HARDWARE WITH THE "VIRTUAL OPTOMETRIST" SOFTWARE...
BETA TEST
MarkParisi@aol.com
offthemark.com
©2004 MARK PARISI DIST. BY UFS, INC.


http://www.goofyhumor.com
WHY DO YOU SAY YOU'RE FIXING DINNER? WHAT WENT WRONG WITH IT?
ASHLEY COOPER



"Dear, your boss justcalled to tell you there was a slight mistake in your paycheck."

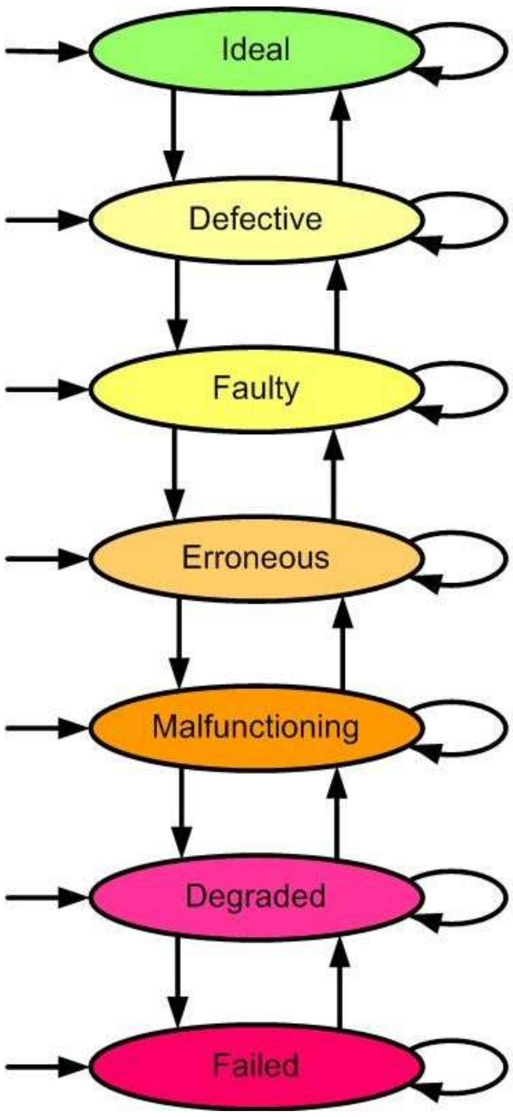
"We found the problem. You called your computer a moron and it's waiting for an apology."

"THERE'S NOTHING WRONG WITH YOUR IPOD, DAD. IT'S JUST TOO EMBARRASSED TO PLAY THE KIND OF MUSIC YOU LIKE!"

STRUCTURE AT A GLANCE

Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Resilient Algorithms
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods Examples	21. Degradation Allowance 22. Degradation Management 23. Robust Task Scheduling 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

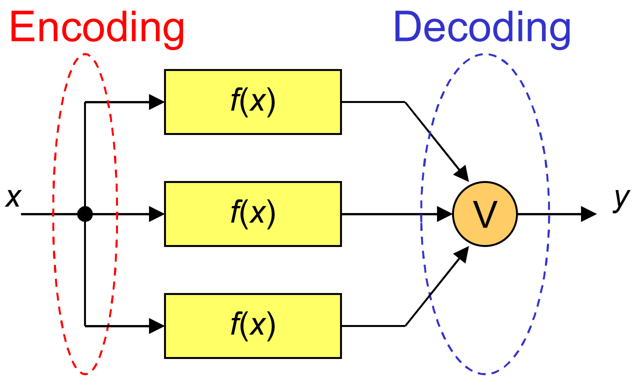
Appendix: Past, Present, and Future



14.1 Basics of Error Correction

High-redundancy codes

Triplication is a form of error coding:
x represented as *xxx* (200% redundancy)
Corrects any error in one version
Detects two nonsimultaneous errors



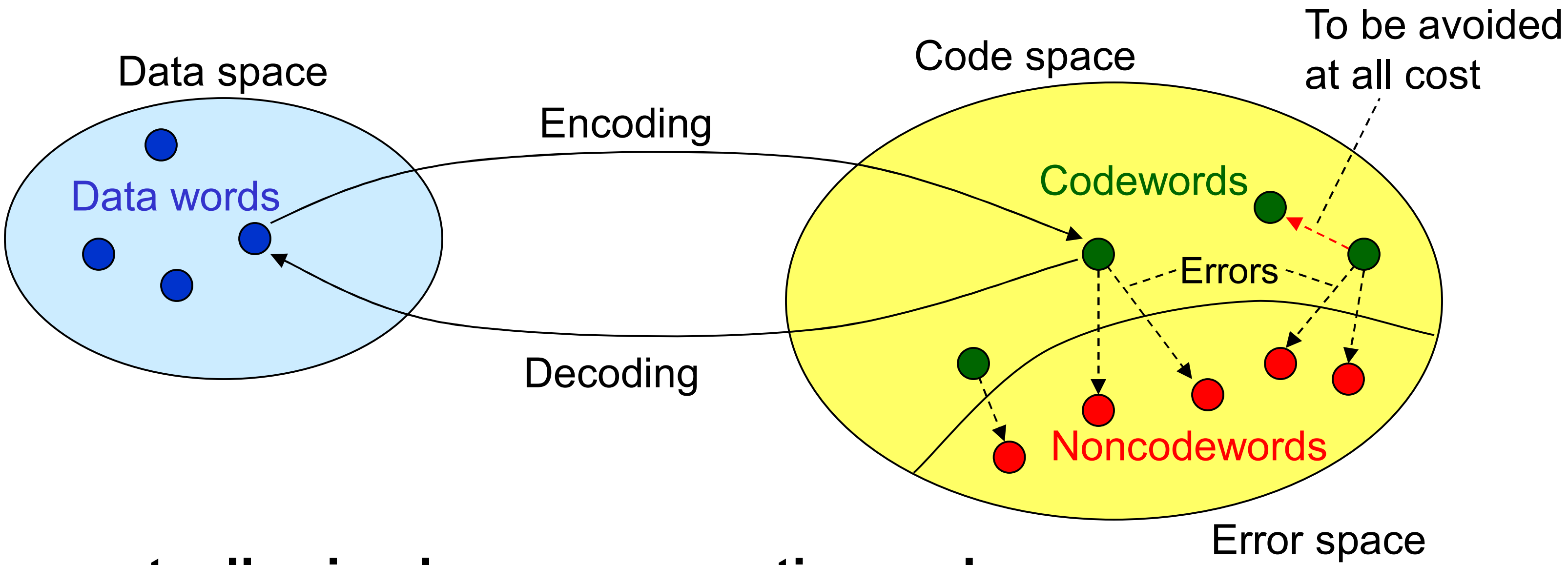
If we triplicate the voting unit to obtain 3 results, we are essentially performing the operation $f(x)$ on coded inputs, getting coded outputs

With a larger replication factor, more errors can be corrected

Our challenge here is to come up with strong correction capabilities, using much lower redundancy (perhaps an order of magnitude less)

To correct all single-bit errors in an *n*-bit code, we must have $2^r > n$, or $2^r > k + r$, which leads to about $\log_2 k$ check bits, at least

The Concept of Error-Correcting Codes



A conceptually simple error-correcting code:

Arrange the k data bits into a $k^{1/2} \times k^{1/2}$ square array

Attach an even parity bit to each row and column of the array

Row/Column check bit = XOR of all row/column data bits

Data space: All 2^k possible k-bit words

Redundancy: $2k^{1/2} + 1$ check bits for k data bits

Corrects all single-bit errors (lead to distinct noncodewords)

Detects all double-bit errors (some triples go undetected)

		0	
0	1	~~1~~	0
0	1	0	1
1	0	1	0
1	0	0	1

Evaluation of Error-Correcting Codes

Redundancy: k data bits encoded in $n = k + r$ bits (r redundant bits)

Encoding: Complexity (circuit / time) to form codeword from data word

Decoding: Complexity (circuit / time) to obtain data word from codeword

Capability: Classes of error that can be corrected
Greater correction capability generally involves more redundancy
To correct c bit-errors, a minimum code distance of $2c + 1$ is required

Examples of code correction capabilities:
Single, double, byte, *b*-bit burst, unidirectional, . . . errors

Combined error correction/detection capability:
To correct c errors and additionally detect d errors ($d > c$),
a minimum code distance of $c + d + 1$ is required

Example: Hamming SEC/DED code has a code distance of 4

Hamming Distance for Error Correction

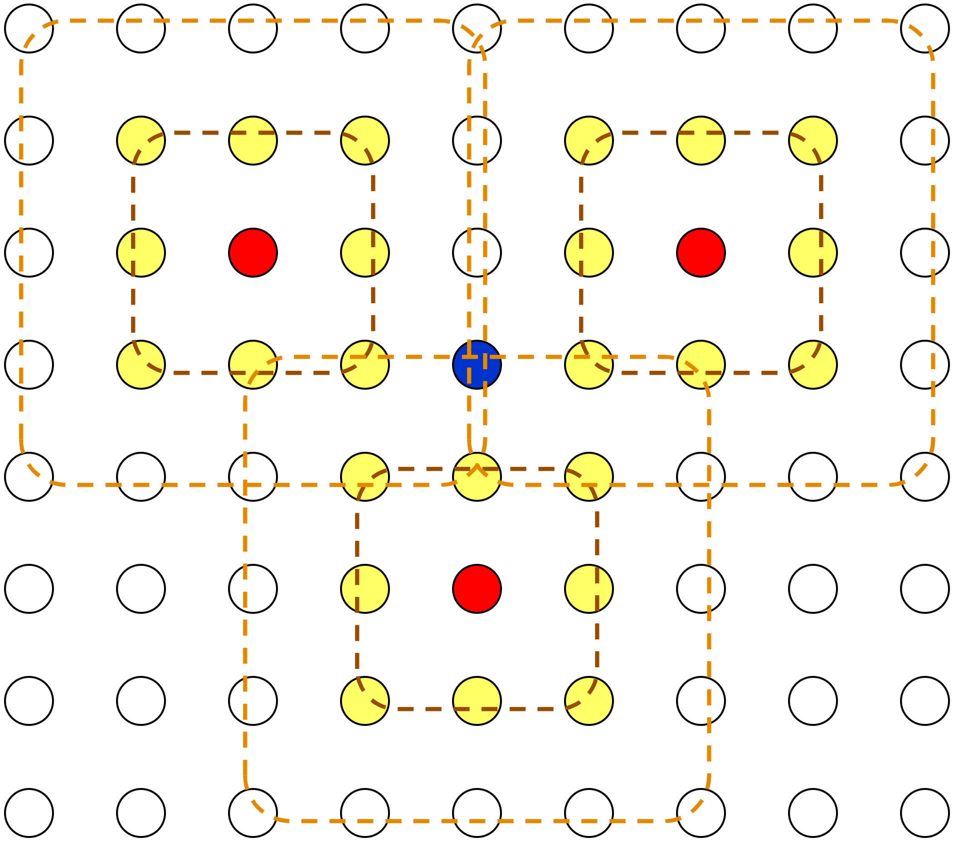
The following visualization, though not completely accurate, is still useful

Red dots represent codewords

Yellow dots, noncodewords within distance 1 of codewords, represent correctable errors

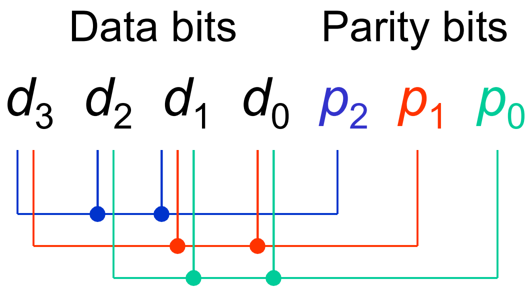
Blue dot, within distance 2 of three different codewords represents a detectable error

Simultaneous single error correction and double error detection requires that there not be points within distance 2 of some codewords that are also within distance 1 of another



14.2 Hamming Codes

Example: Uses multiple parity bits, each applied to a different subset of data bits



Encoding: 3 XOR networks to form parity bits

Checking: 3 XOR networks to verify parities

Decoding: Trivial (separable code)

Redundancy: 3 check bits for 4 data bits
Unimpressive, but gets better with more data bits
(7, 4); (15, 11); (31, 26); (63, 57); (127, 120)

Capability: Corrects any single-bit error

$s_2\ s_1\ s_0$	Error
0 0 0	None
0 0 1	p_0
0 1 0	p_1
0 1 1	d_0
1 0 0	p_2
1 0 1	d_2
1 1 0	d_3
1 1 1	d_1

$$s_2 = d_3 \oplus d_2 \oplus d_1 \oplus p_2$$
$$s_1 = d_3 \oplus d_1 \oplus d_0 \oplus p_1$$
$$s_0 = d_2 \oplus d_1 \oplus d_0 \oplus p_0$$

$s_2\ s_1\ s_0$
Syndrome

Matrix Formulation of Hamming SEC Code

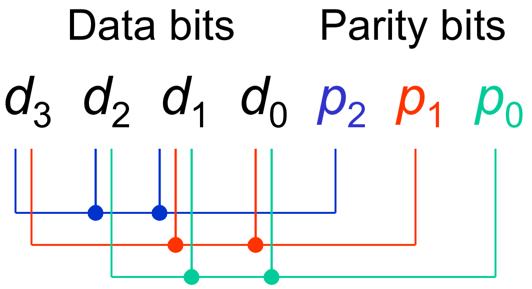
Data bits Parity bits

$$\begin{array}{ccccccc} d_3 & d_2 & d_1 & d_0 & p_2 & p_1 & p_0 \\ \hline 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array} \times \begin{pmatrix} d_3 \\ d_2 \\ d_1 \\ d_0 \\ p_2 \\ p_1 \\ p_0 \end{pmatrix} = \begin{pmatrix} s_2 \\ s_1 \\ s_0 \end{pmatrix}$$

Parity check matrix

Received word

Syndrome



s_2 s_1 s_0	Error
0 0 0	None
0 0 1	p_0
0 1 0	p_1
0 1 1	d_0
1 0 0	p_2
1 0 1	d_2
1 1 0	d_3
1 1 1	d_1

Syndrome matches the p_2 column in the parity check matrix

Matrix-vector multiplication is done with AND/XOR, instead of ×/+

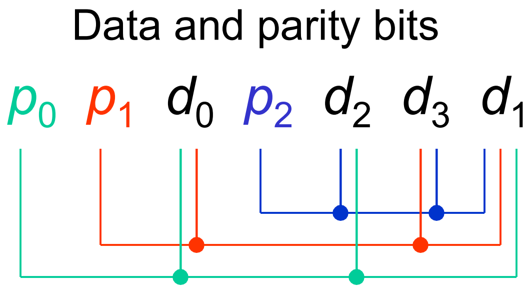
Matrix Rearrangement for Simpler Correction

Data and parity bits

$$\begin{array}{c} \begin{array}{ccccccc} p_0 & p_1 & d_0 & p_2 & d_2 & d_3 & d_1 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \\ \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \end{array} \times \begin{pmatrix} p_0 \\ p_1 \\ d_0 \\ p_2 \\ d_2 \\ d_3 \\ d_1 \end{pmatrix} = \begin{pmatrix} s_2 \\ s_1 \\ s_0 \end{pmatrix}$$

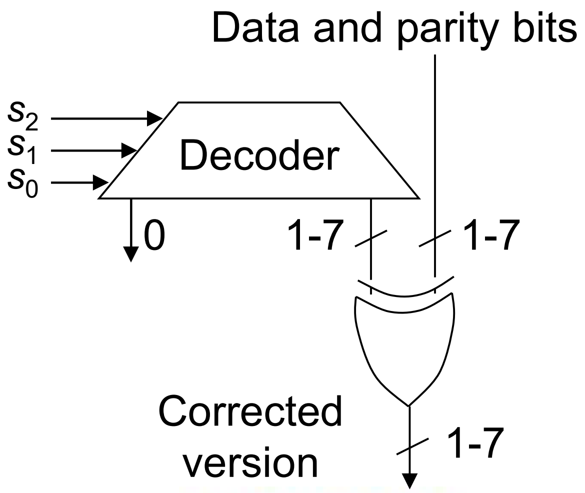
Position number

Matrix columns are binary rep's of column indices



s_2 s_1 s_0	Error
0 0 0	None
0 0 1	p_0
0 1 0	p_1
0 1 1	d_0
1 0 0	p_2
1 0 1	d_2
1 1 0	d_3
1 1 1	d_1

Syndrome indicates error in position 4

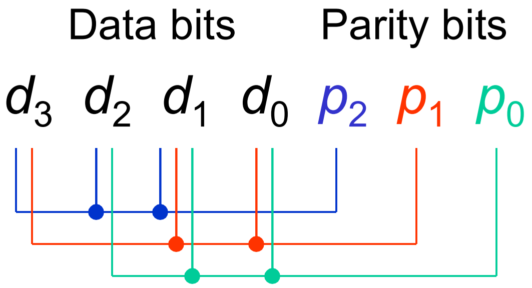


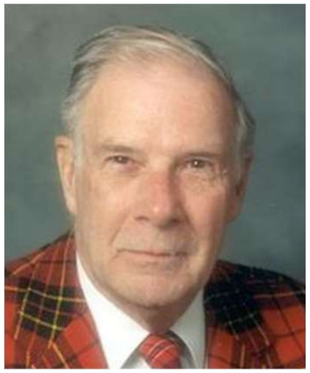
Hamming Generator Matrix

Data bits

$$\begin{array}{c} \begin{array}{cccc} d_3 & d_2 & d_1 & d_0 \end{array} \\ \left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right) \end{array} \times \begin{pmatrix} d_3 \\ d_2 \\ d_1 \\ d_0 \end{pmatrix} = \begin{pmatrix} d_3 \\ d_2 \\ d_1 \\ d_0 \\ p_2 \\ p_1 \\ p_0 \end{pmatrix}$$

Generator matrix Data word Codeword





Richard W. Hamming (Bell Labs, Naval Postgraduate School)

Recall that matrix-vector multiplication is done with AND/XOR, instead of ×/+

Generalization to Wider Hamming SEC Codes

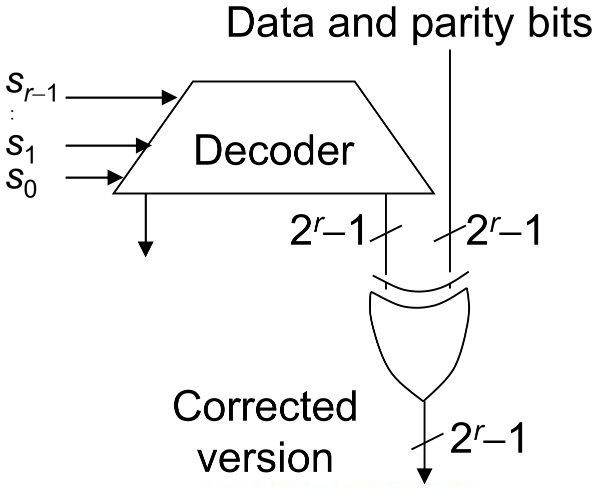
Data and parity bits

$$\begin{array}{c} \begin{array}{ccccccc} p_0 & p_1 & d_0 & p_2 & . & . & . \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & \ldots & 1 & 1 & 1 \\ \vdots & \vdots & \vdots & \ldots & \vdots & \vdots & \vdots \\ 0 & 1 & 1 & \ldots & 0 & 1 & 1 \\ 1 & 0 & 1 & \ldots & 1 & 0 & 1 \end{pmatrix} \\ \begin{array}{ccccccc} 1 & 2 & 3 & & & & 2^r-1 \end{array} \end{array} \times \begin{pmatrix} p_0 \\ p_1 \\ d_0 \\ p_2 \\ . \\ . \\ . \end{pmatrix} = \begin{pmatrix} s_{r-1} \\ \vdots \\ s_1 \\ s_0 \end{pmatrix}$$

Position number

Condition for general Hamming SEC code:
$n = k + r = 2^r - 1$

n	$k = n - r$
7	4
15	11
31	26
63	57
127	120
255	247
511	502
1023	1013



Matrix columns are binary rep's of column indices

A Hamming SEC/DED Code

Data and parity bits

$$\begin{array}{cccccccc} p_0 & p_1 & d_0 & p_2 & . & . & . & \\ 1 & 1 & 1 & \dots & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & \dots & 1 & 1 & 1 & 0 \\ : & : & : & \dots & : & : & : & : \\ 0 & 1 & 1 & \dots & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & \dots & 1 & 0 & 1 & 0 \\ 1 & 2 & 3 & & & & 2^r-1 & \end{array} \times \begin{pmatrix} p_0 \\ p_1 \\ d_0 \\ p_2 \\ . \\ . \\ . \\ p_r \end{pmatrix} = \begin{pmatrix} s_r \\ s_{r-1} \\ : \\ s_1 \\ s_0 \end{pmatrix}$$

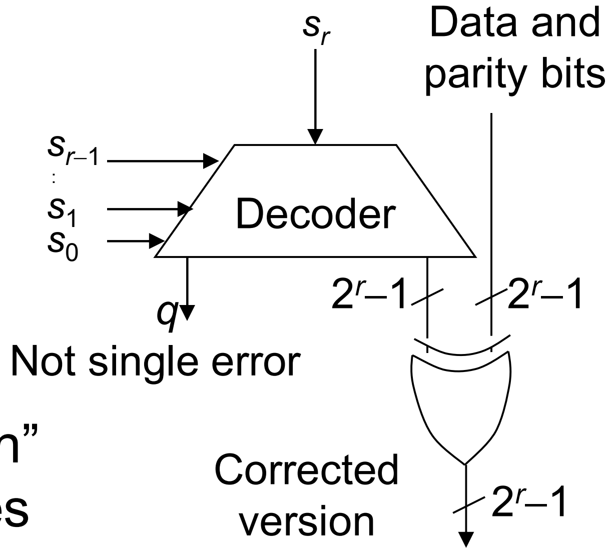
Position number

Parity check matrix

Received word

Syndrome

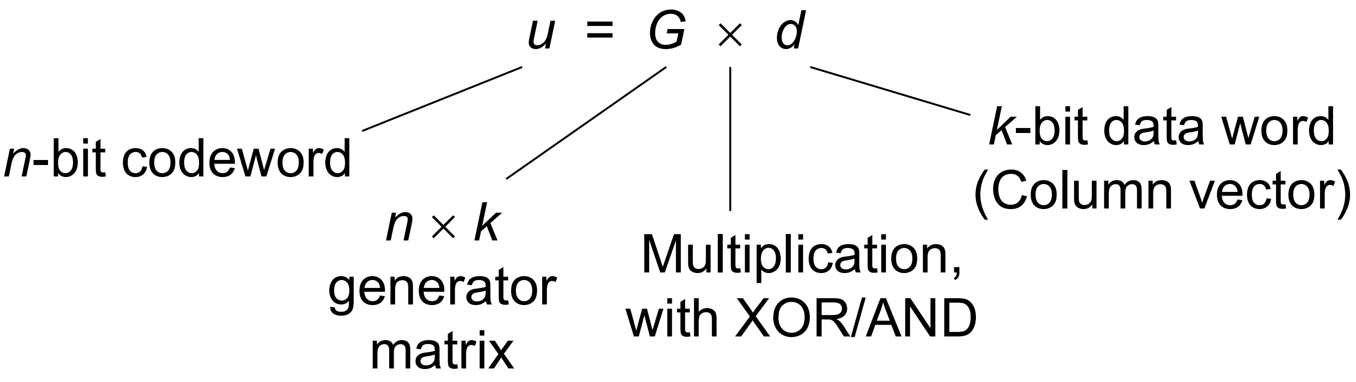
Add an extra row of all 1s and a column with only one 1 to the parity check matrix



Easy to verify that the appropriate "correction" is made for all 4 combinations of (s_r, q) values

14.3 Linear Codes

Hamming codes are examples of linear codes
Linear codes may be defined in many other ways



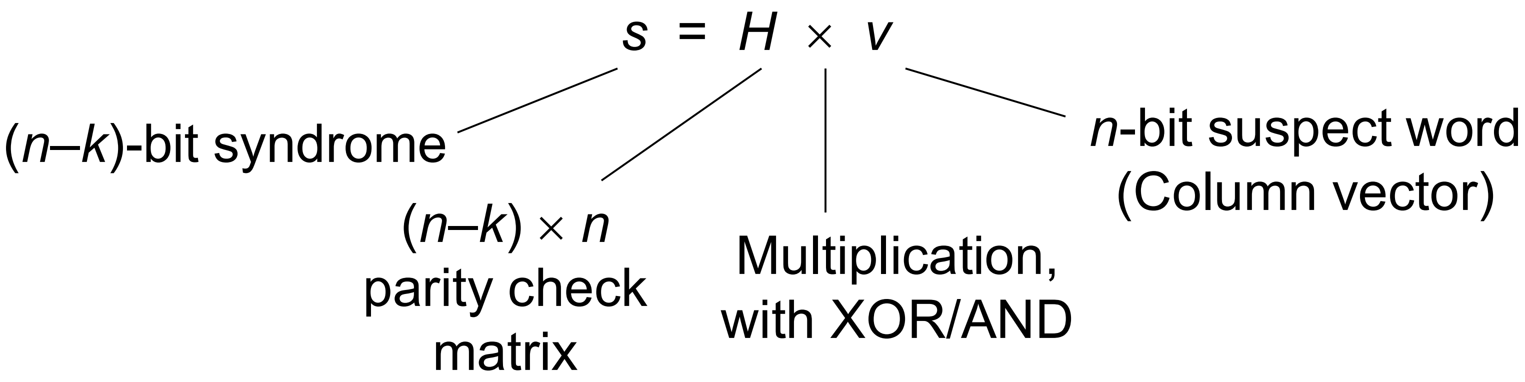
$u = G \times d$

n-bit codeword

$n \times k$ generator matrix

Multiplication, with XOR/AND

k-bit data word (Column vector)



$s = H \times v$

(*n*–*k*)-bit syndrome

$(n–k) \times n$ parity check matrix

Multiplication, with XOR/AND

n-bit suspect word (Column vector)

14.4 Reed-Solomon and BCH Codes

BCH codes: Named in honor of Bose, Chaudhuri, Hocquenghem

Reed-Solomon codes: Special case of BCH code

Example: A popular variant is RS(255, 223) with 8-bit symbols
223 bytes of data, 32 check bytes, redundancy ≈ 14%
Can correct errors in up to 16 bytes anywhere in the 255-byte codeword
Used in CD players, digital audio tape, digital television

Reed-Solomon Codes

With k data symbols, require $2t$ check symbols, each s bits wide, to correct up to t symbol errors; hence, RS($k + 2t$, k) has distance $2t + 1$
The number k of data symbols must satisfy $k \le 2^s - 1 - 2t$ (s grows with k)

k data symbols	2*t* check symbols

Example: RS(6, 2) code, with 2 data and $2t = 4$ check symbols (7-valued)
→ up to $t = 2$ symbol errors correctable; hence, RS(6, 2) has distance 5

Generator polynomial: $g(x) = (x - \alpha)(x - \alpha^2)(x - \alpha^3)(x - \alpha^4)$;
α is a primitive root mod 7 $\Rightarrow$ integers from 1 to 6 are powers of α mod 7
$3^1 = 3$; $3^2 = 2$; $3^3 = 6$; $3^4 = 4$; $3^5 = 5$; $3^6 = 1$

Pick $\alpha = 3 \rightarrow g(x) = (x - 3)(x - 3^2)(x - 3^3)(x - 3^4)$
$= (x - 3)(x - 2)(x - 6)(x - 4) = x^4 + 6x^3 + 3x^2 + 2x + 4$

As usual, the codeword is the product of $g(x)$ and the info polynomial; convertible to matrix-by-vector multiply by deriving a generator matrix G

Elements of Galois Field GF(2^3)

A primitive element α of GF(2^3) is one that generates all nonzero elements of the field by its powers

Here are three different representation of the elements of GF(2^3)

Power	Polynomial	Vector
--	0	000
1	1	001
α	α	010
α^2	α^2	100
α^3	$\alpha + 1$	011
α^4	$\alpha^2 + \alpha$	110
α^5	$\alpha^2 + \alpha + 1$	111
α^6	$\alpha^2 + 1$	101

BCH Codes

Correct the deficiency of Reed-Solomon code; have a fixed alphabet
We usually choose the alphabet {0, 1}

BCH(15, 7) code: Capable of correcting any two errors

Generator polynomial: $g(x) = 1 + x^4 + x^6 + x^7 + x^8$

$$[0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 0] \times \begin{pmatrix} 1000\ 1000 \\ 0100\ 0001 \\ 0010\ 0011 \\ 0001\ 0101 \\ 1100\ 1111 \\ 0110\ 1000 \\ 0011\ 0001 \\ 1101\ 0011 \\ 1010\ 0101 \\ 0101\ 1111 \\ 1110\ 1000 \\ 0111\ 0001 \\ 1111\ 0011 \\ 1011\ 0101 \\ 1001\ 1111 \end{pmatrix} = [x\ x\ x\ x\ x\ x\ x\ x]$$

Received word

Syndrome

Parity check matrix

BCH(511, 493) used as DEC code in a video coding standard for videophones

BCH(40, 32) used as SEC/DED code in ATM

14.5 Arithmetic Error-Correcting Codes

Positive	Syndrome		Negative	Syndrome	
error	mod 7	mod 15	error	mod 7	mod 15
1	1	1	–1	6	14
2	2	2	–2	5	13
4	4	4	–4	3	11
8	1	8	–8	6	7
16	2	1	–16	5	14
32	4	2	–32	3	13
64	1	4	–64	6	11
128	2	8	–128	5	7
256	4	1	–256	3	14
512	1	2	–512	6	13
1024	2	4	–1024	5	11
2048	4	8	–2048	3	7
4096	1	1	–4096	6	14
8192	2	2	–8192	5	13
16,384	4	4	–16,384	3	11
32,768	1	8	–32,768	6	7

Error syndromes for weight-1 arithmetic errors in the (7, 15) biresidue code

Because all the syndromes in this table are different, any weight-1 arithmetic error is correctable by the (mod 7, mod 15) biresidue code

Properties of Biresidue Codes

Biresidue code with relatively prime low-cost check moduli $A = 2^a - 1$ and $B = 2^b - 1$ supports $a \times b$ bits of data for weight-1 error correction

Representational redundancy = $(a + b)/(ab) = 1/a + 1/b$

a	*b*	*n=k+a+b*	*k=ab*
3	4	19	12
5	6	41	30
7	8	71	56
11	12	143	120
15	16	271	240

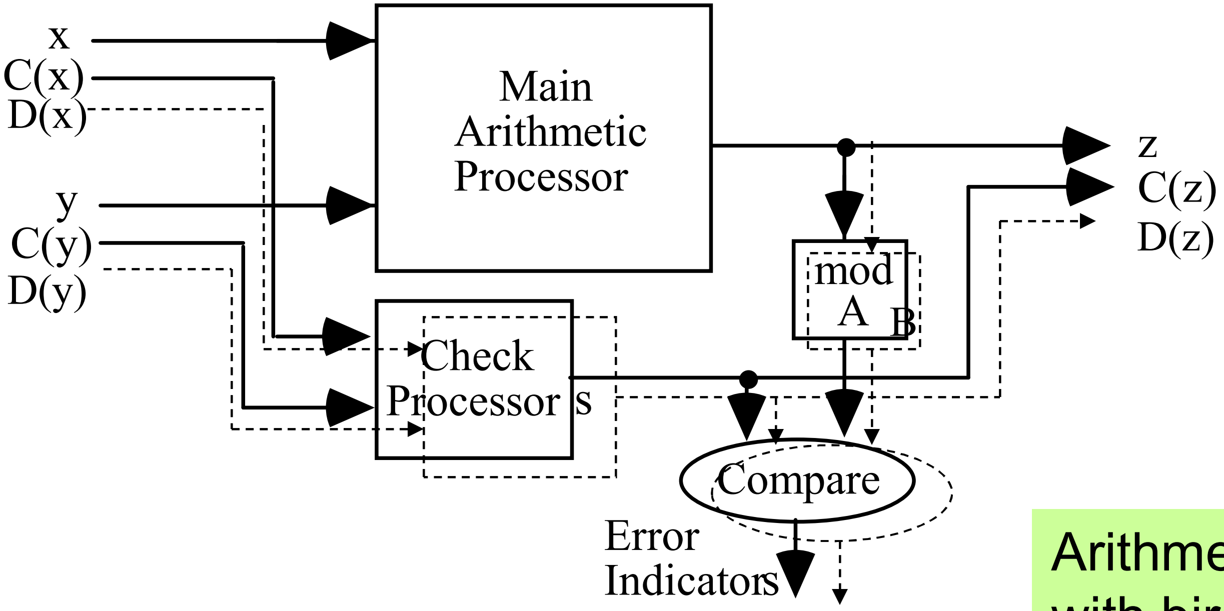
Compare with Hamming SEC code →

n	*k*
7	4
15	11
31	26
63	57
127	120
255	247
511	502
1023	1013

Arithmetic on Biresidue-Coded Operands

Similar to residue-checked arithmetic for addition and multiplication, except that two residues are involved

Divide/square-root: remains difficult

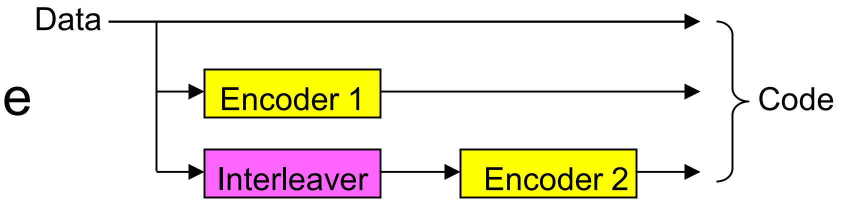


Arithmetic processor with biresidue checking

14.6 Other Error-Correcting Codes

Reed-Muller codes: Have a recursive construction, with smaller codes used to build larger ones

Turbo codes: Highly efficient separable codes, with iterative (soft) decoding



Low-density parity check (LDPC) codes: Each parity check is defined on a small set of bits, so error checking is fast; correction is more difficult

Information dispersal: Encoding data into n pieces, such that any k of the pieces are adequate for reconstructing the data

Higher-Level Error Coding Methods

We have applied coding to data at the bit-string or word level

It is also possible to apply coding at higher levels

Data structure level – Robust data structures

Application level – Algorithm-based error tolerance

Preview of Algorithm-Based Error Tolerance

Error coding applied to data structures, rather than at the level of atomic data elements

Example: mod-8 checksums used for matrices

If $Z = X \times Y$ then $Z_f = X_c \times Y_r$

In M_f, any single error is correctable and any 3 errors are detectable

Four errors may go undetected

Matrix M

$$M = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \end{pmatrix}$$

Row checksum matrix

$$M_r = \begin{pmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \end{pmatrix}$$

Column checksum matrix

$$M_c = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \\ 2 & 6 & 1 \end{pmatrix}$$

Full checksum matrix

$$M_f = \begin{pmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \\ 2 & 6 & 1 & 1 \end{pmatrix}$$

Self-Checking Modules

Of *course* I have a grandiose sense of self-importance. Who doesn't?"

Earl checks his balance at the bank.

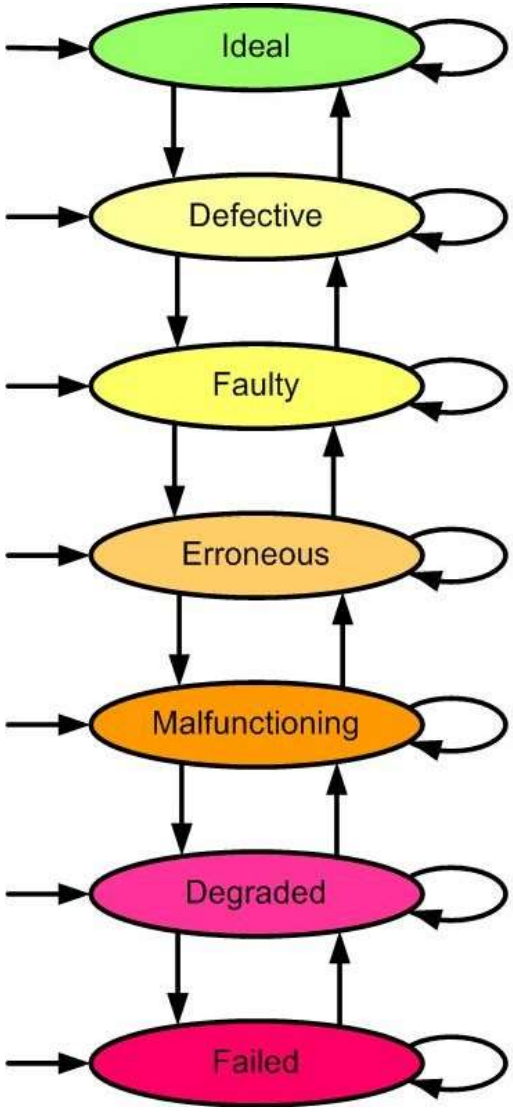
© MARK ANDERSON, ALL RIGHTS RESERVED WWW.ANDERTOONS.COM

"He's called Sir Lance-A-Lot because he's always checking his blood glucose."

STRUCTURE AT A GLANCE

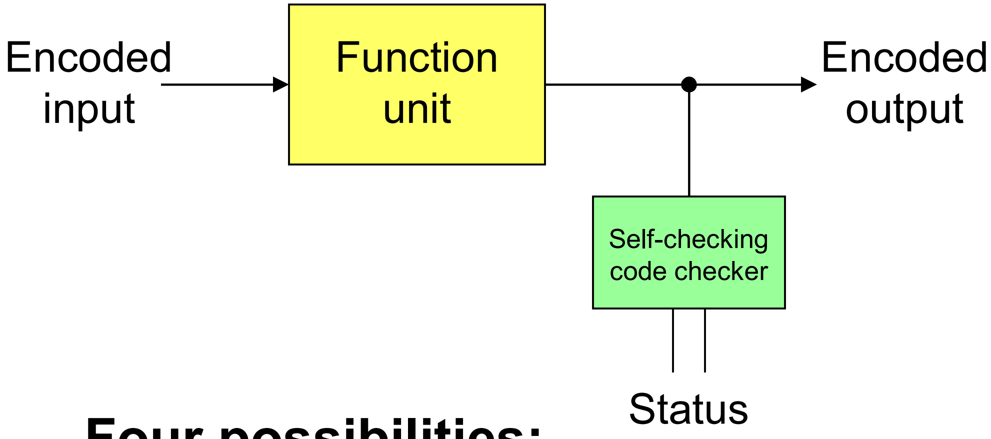
Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Resilient Algorithms
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods Examples	21. Degradation Allowance 22. Degradation Management 23. Robust Task Scheduling 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

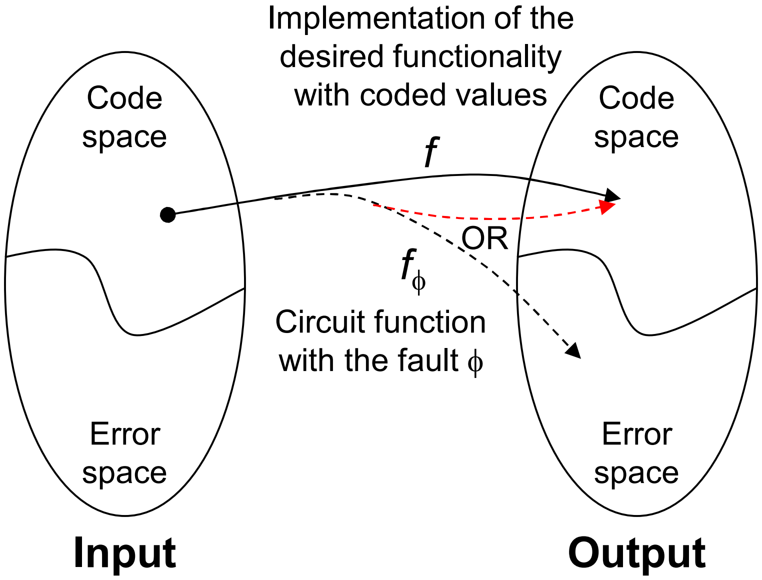
Appendix: Past, Present, and Future



15.1 Checking of Function Units

Function unit designed in a way that faults/errors/malfns manifest themselves as invalid (error-space) outputs, which are detectable by an external code checker




Implementation of the desired functionality with coded values
f
OR
f_ϕ
Circuit function with the fault ϕ
Code space
Error space
Code space
Error space
Input
Output

Four possibilities:

Both function unit and checker okay

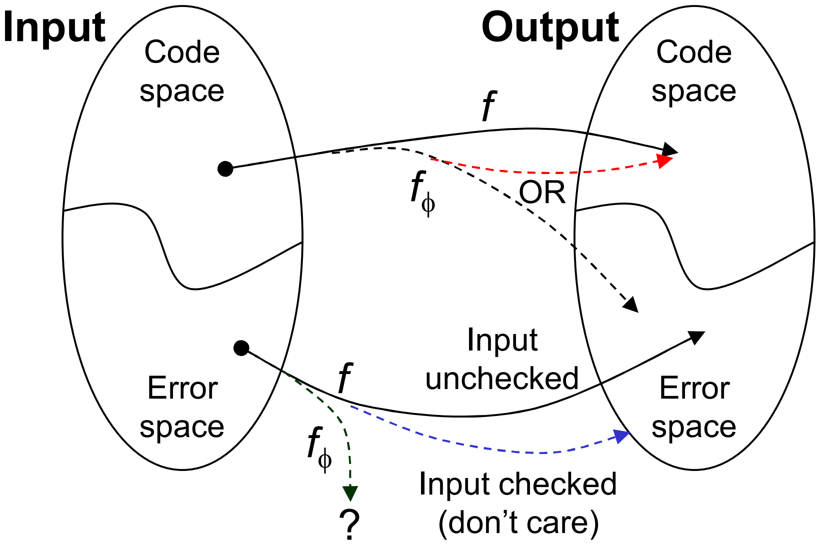
Only function unit okay (false alarm may be raised, but this is safe)

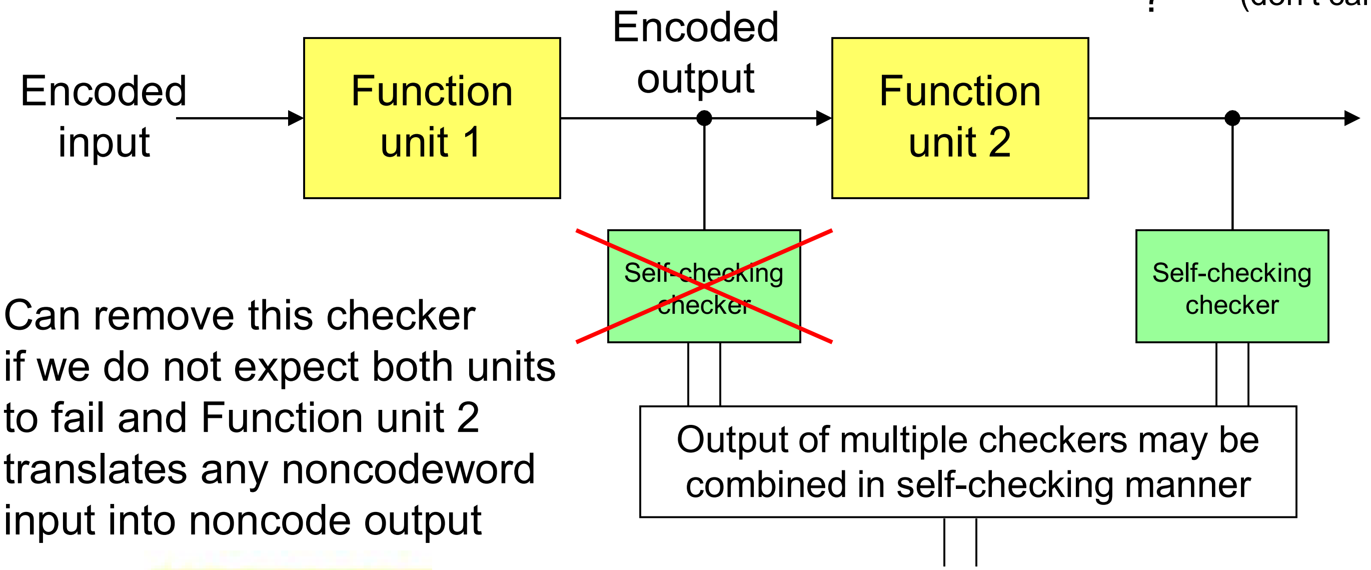
Only checker okay (we have either no output error or a detectable error)

Neither function unit nor checker okay (use 2-output checker; a single check signal stuck-at-okay goes undetected, leading to fault accumulation)

Cascading of Self-Checking Modules

Given self-checking modules that have been designed separately, how does one combine them into a self-checking system?

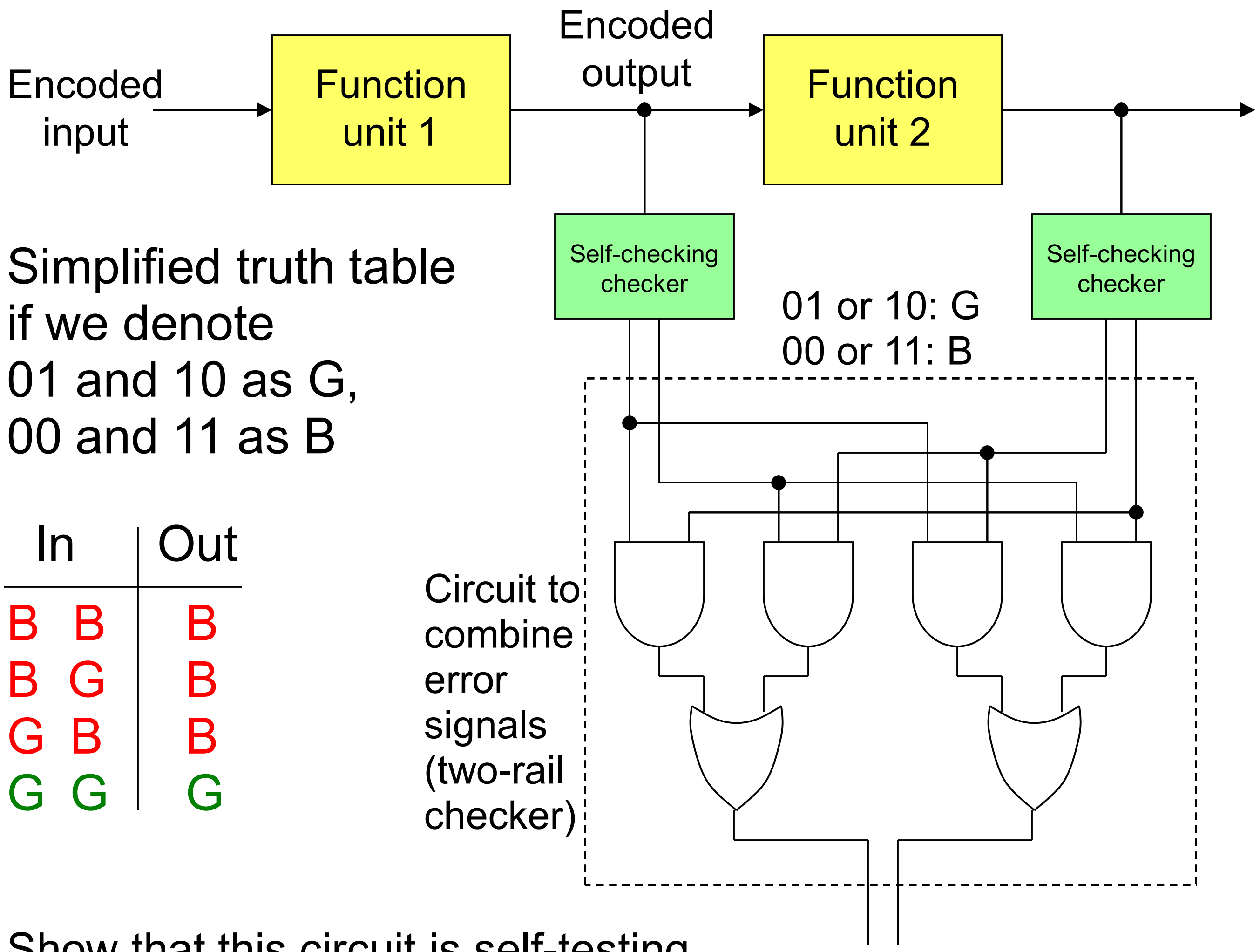




Output of multiple checkers may be combined in self-checking manner

Can remove this checker if we do not expect both units to fail and Function unit 2 translates any noncodeword input into noncode output

15.2 Error Signal and Their Combining



Simplified truth table if we denote 01 and 10 as G, 00 and 11 as B

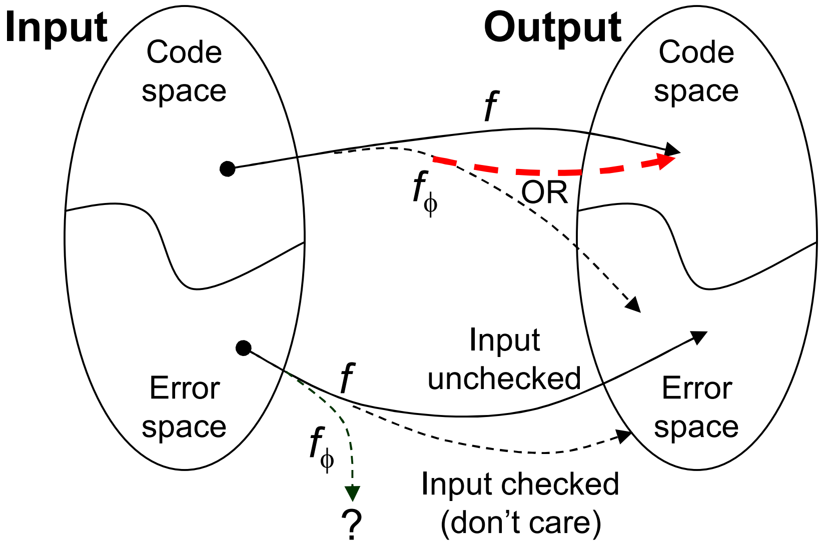
In		Out
B	B	B
B	G	B
G	B	B
G	G	G

In	Out
0 0 0 0	0 0
0 0 0 1	0 0
0 0 1 0	0 0
0 0 1 1	0 0
0 1 0 0	0 0
0 1 0 1	0 1
0 1 1 0	1 0
0 1 1 1	1 1
1 0 0 0	0 0
1 0 0 1	1 0
1 0 1 0	0 1
1 0 1 1	1 1
1 1 0 0	0 0
1 1 0 1	1 1
1 1 1 0	1 1
1 1 1 1	1 1

Show that this circuit is self-testing

15.3 Totally Self-Checking Design

A module is totally self-checking if it is self-checking and self-testing



If the dashed red arrow option is used too often, faults may go undetected for long periods of time, raising the danger of a second fault invalidating the self-checking design

A self-checking circuit is self-testing if any fault from the class covered is revealed at output by at least one code-space input, so that the fault is guaranteed to be detectable during normal circuit operation

Note that if we don't explicitly ensure this, tests for some of the faults may belong to the input error space

The self-testing property allows us to focus on a small set of faults, thus leading to more economical self-checking circuit implementations (with a large fault set, cost would be prohibitive)

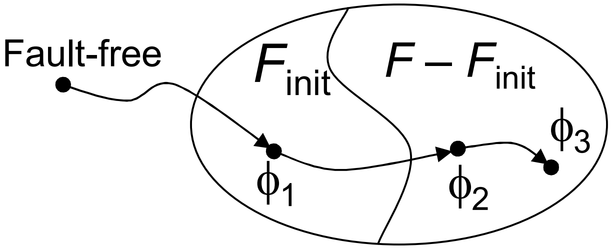
Self-Monitoring Design

A module is self monitoring with respect to the fault class *F* if it is

(1) Self-checking with respect to *F*, or

(2) Totally self-checking wrt the fault class $F_{\text{init}} \subseteq F$, chosen such that all faults in *F* develop in time as a sequence of simpler faults, the first of which is in F_{init}

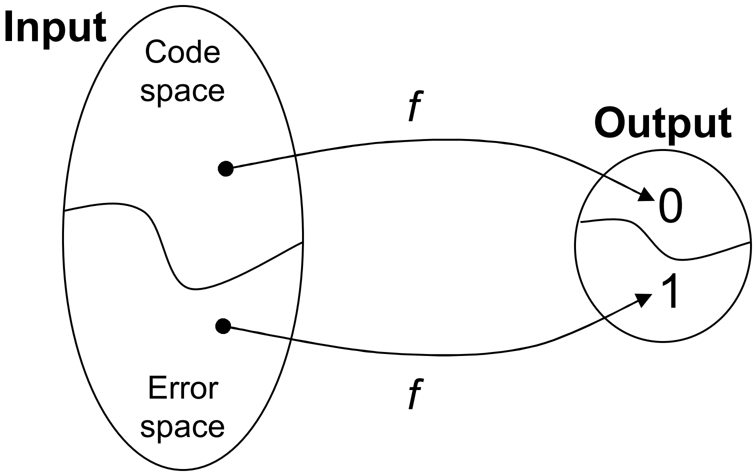
Example:
A unit that is totally-self-checking wrt single faults may be deemed self-monitoring wrt to multiple faults, provided that multiple faults develop one by one and slowly over time



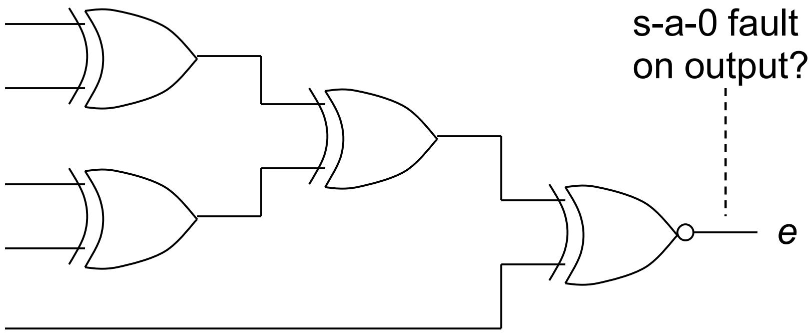
The self-monitoring design approach requires the more stringent totally-self-checking property to be satisfied for a small, manageable set of faults, while also protecting the unit against a broader fault class

15.4 Self-Checking Checkers

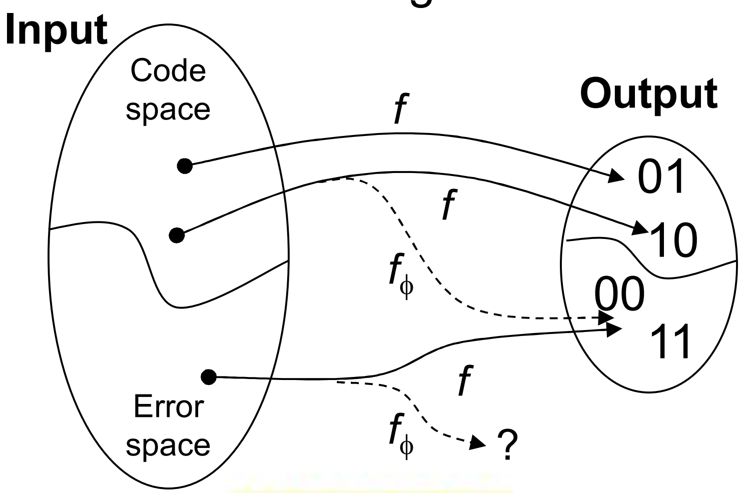
Conventional code checker



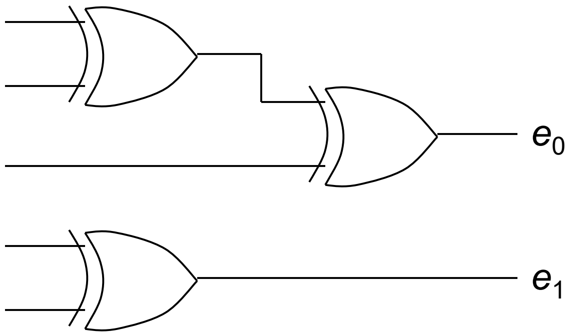
Example: 5-input odd-parity checker



Self-checking code checker



Example: 5-input odd-parity checker



Pleasant surprise: The self-checking version is simpler!

TSC Checker for *m*-out-of-2*m* Code

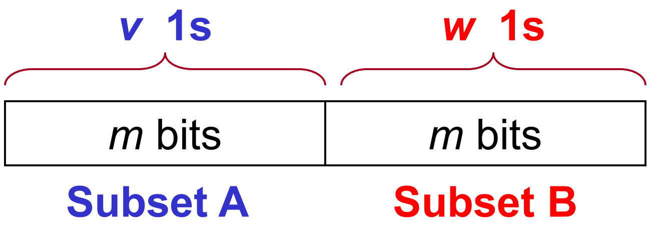
Divide the 2*m* bits into two disjoint subsets *A* and *B* of *m* bits each

Let *v* and *w* be the weight of (number of 1s in) *A* and *B*, respectively

Implement the two code checker outputs e_0 and e_1 as follows:

$$e_0 = \bigvee_{\substack{i = 0 \\ (i \text{ even})}}^{m} (v \geq i)(w \geq m - i)$$

$$e_1 = \bigvee_{\substack{j = 1 \\ (j \text{ odd})}}^{m} (v \geq j)(w \geq m - j)$$



Example: 3-out-of-6 code checker, $m = 3$, $A = \overbrace{\{a, b, c\}}^{v \text{ 1s}}$, $B = \overbrace{\{f, g, h\}}^{w \text{ 1s}}$

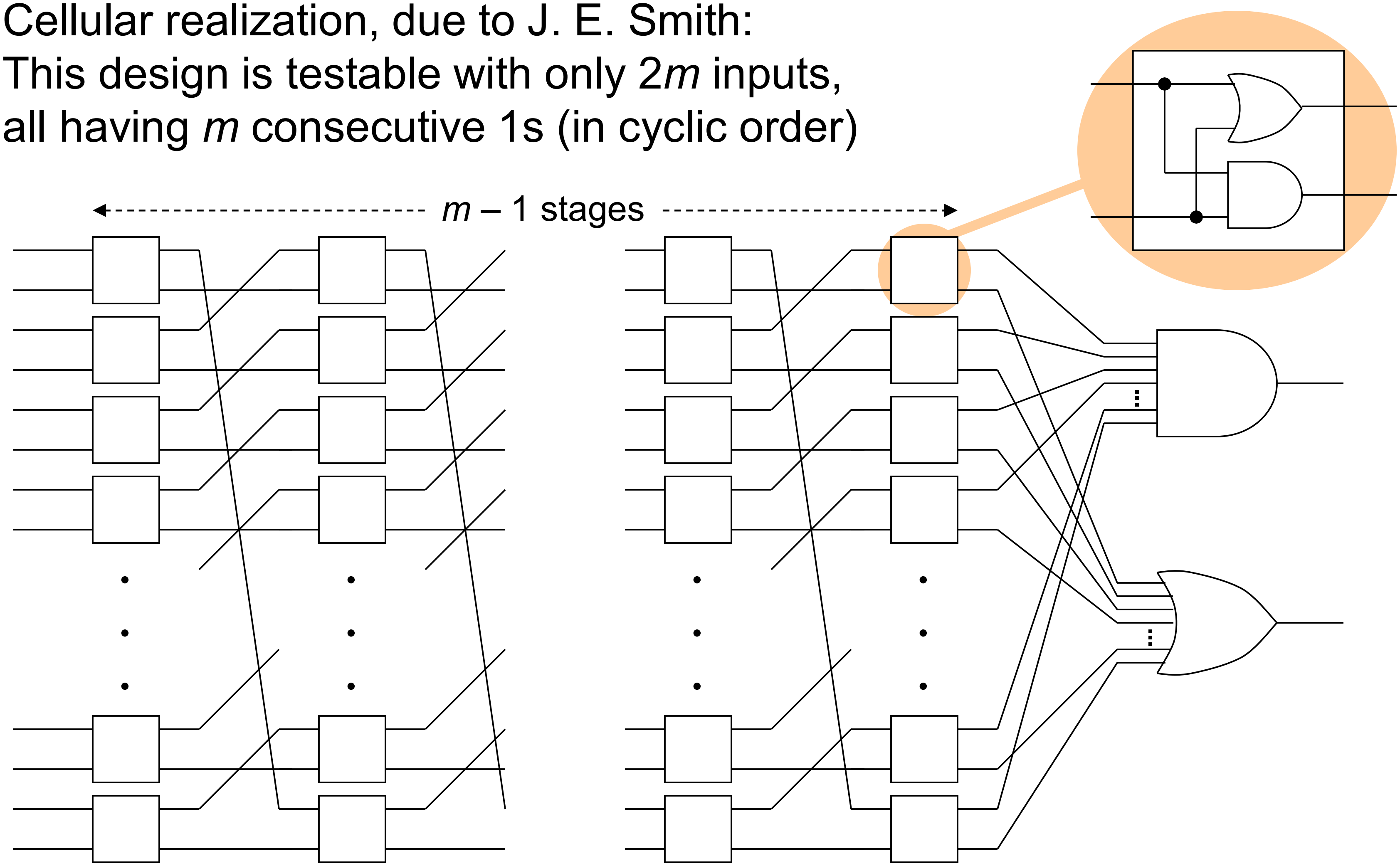
$e_0 = (v \cancel{\geq} 0)(w \geq 3) \vee (v \geq 2)(w \geq 1) = fgh \vee (ab \vee bc \vee ca)(f \vee g \vee h)$

$e_1 = (v \geq 1)(w \geq 2) \vee (v \geq 3)(w \cancel{\geq} 0) = (a \vee b \vee c)(fg \vee gh \vee hf) \vee abc$

Always satisfied

Another TSC *m*-out-of-2*m* Code Checker

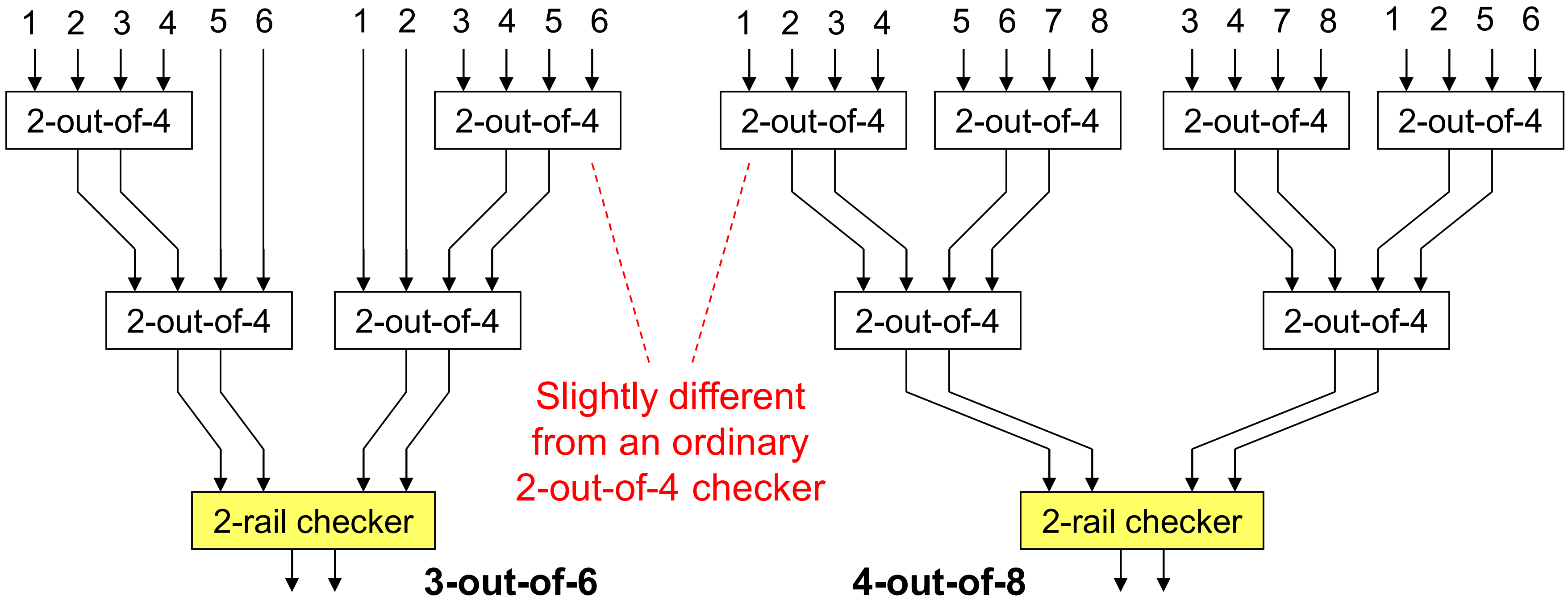
Cellular realization, due to J. E. Smith:
This design is testable with only 2*m* inputs,
all having *m* consecutive 1s (in cyclic order)



Using 2-out-of-4 Checkers as Building Blocks

Building *m*-out-of-2*m* TSC checkers, 3 ≤ *m* ≤ 6, from 2-out-of-4 checkers (construction due to Lala, Busaba, and Zhao):

Examples: 3-out-of-6 and 4-out-of-8 TSC checkers are depicted below (only the structure is shown; some design details are missing)

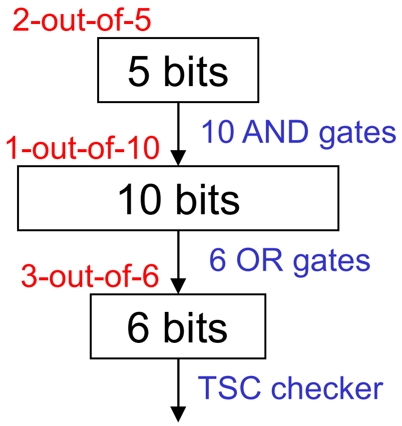


Slightly different from an ordinary 2-out-of-4 checker

TSC Checker for *k*-out-of-*n* Code

One design strategy is to proceed in 3 stages:
Convert the *k*-out-of-*n* code to a 1-out-of-$\binom{n}{k}$ code
Convert the latter code to an *m*-out-of-2*m* code
Check the *m*-out-of-2*m* code using a TSC checker

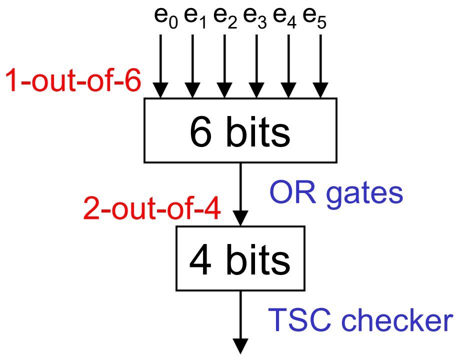
This approach is impractical for many codes



2-out-of-5
5 bits
10 AND gates
1-out-of-10
10 bits
6 OR gates
3-out-of-6
6 bits
TSC checker

A procedure due to Marouf and Friedman:
Implement 6 functions of the general form
(these have different subsets of bits as inputs and constitute a 1-out-of-6 code)
Use a TSC 1-out-of-6 to 2-out-of-4 converter
Use a TSC 2-out-of-4 code checker

$$e_0 = \bigvee_{\substack{j = 1 \\ (j \text{ even})}}^{m} (v \geq j)(w \geq m - j)$$

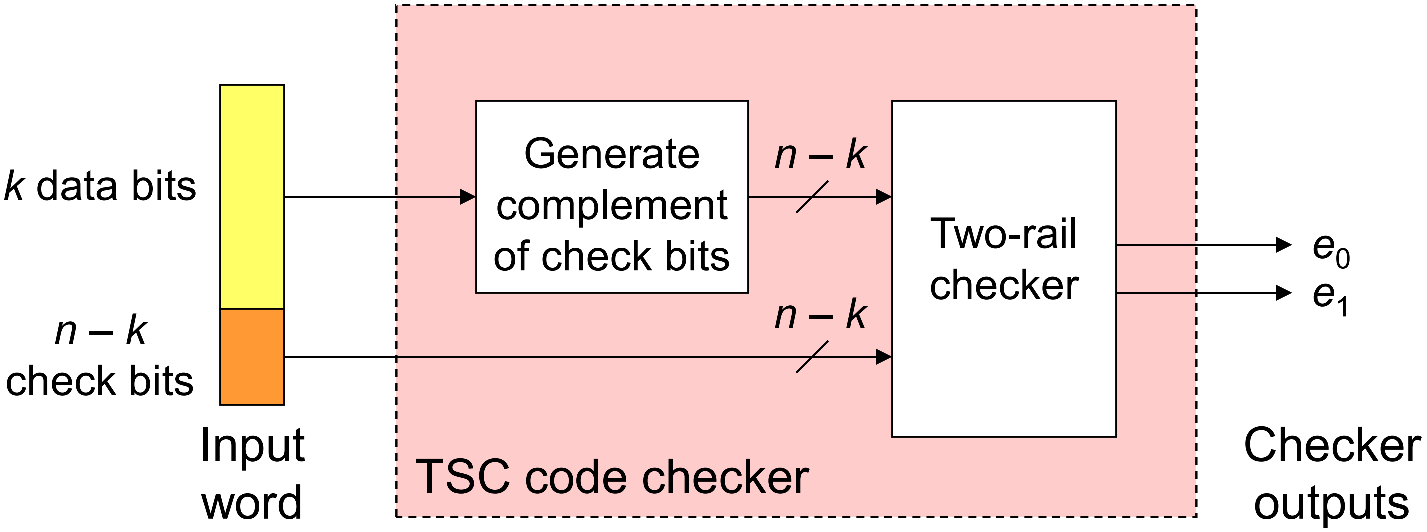


e_0 e_1 e_2 e_3 e_4 e_5
1-out-of-6
6 bits
OR gates
2-out-of-4
4 bits
TSC checker

The process above works for $2k + 2 \leq n \leq 4k$
It can be somewhat simplified for $n = 2k + 1$

TSC Checkers for Separable Codes

Here is a general strategy for designing totally-self-checking checkers for separable codes



For many codes, direct synthesis will produce a faster and/or more compact totally-self-checking checker

Google search for “totally self checking checker” produces 817 hits

15.5 Self-Checking State Machines

Design method for Moore-type machines, due to Diaz and Azema:

Inputs and outputs are encoded using two-rail code
States are encoded as $n/2$-out-of-n codewords

Fact: If the states are encoded using a k-out-of-n code, one can express the next-state functions (one for each bit of the next state) via monotonic expressions; i.e., without complemented variables

Monotonic functions can be realized with only AND and OR gates, hence the unidirectional error detection capability

	Input		Output
State	$x = 0$	$x = 1$	z
A	*C*	*A*	1
B	*D*	*C*	1
C	*B*	*D*	0
D	*C*	*A*	0

	Input		Output
State	$x = 01$	$x = 10$	z
0011	1010	0011	10
0101	1001	1010	10
1010	0101	1001	01
1001	1010	0011	01

15.6 Practical Self-Checking Design

Design based on parity codes

Design with residue encoding

FPGA-based design

General synthesis rules

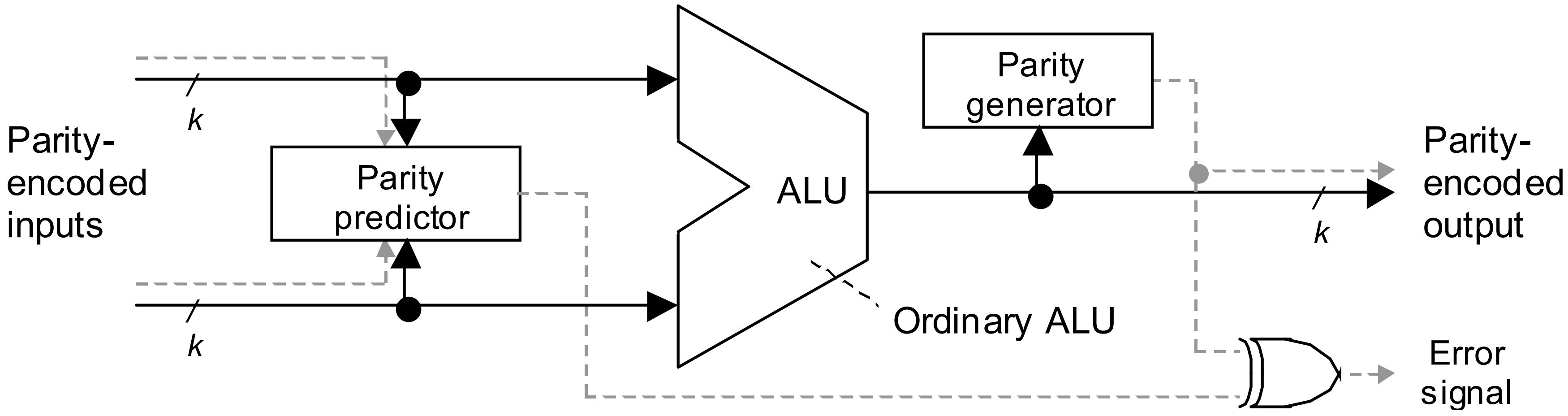
Partially self-checking design

Design with Parity Codes and Parity Prediction

Operands and results are parity-encoded
Parity is not preserved over arithmetic and logic operations

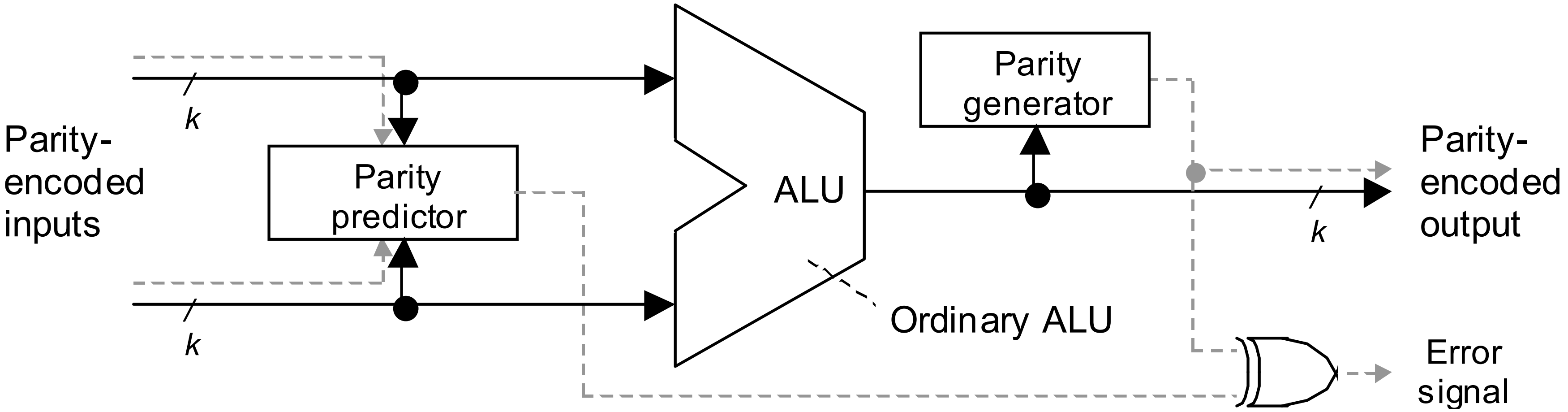
Parity prediction is an alternative to duplication

Compared to duplication:
Parity prediction often involves less overhead in time and space
The protection offered by parity prediction is not as comprehensive



TSC Design with Parity Prediction

Recall our discussion of parity prediction as an alternative to duplication

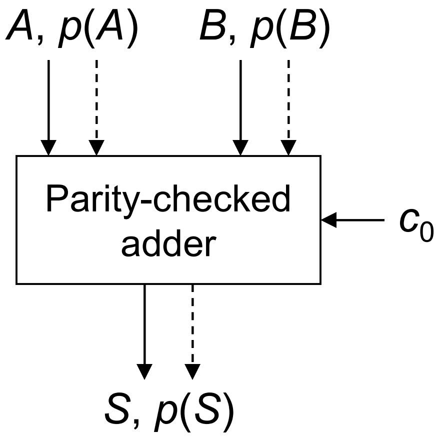


If the parity predictor produces the complement of the output parity, and the XOR gate is removed, we have a self-checking design

To ensure the TSC property, we must also verify that the parity predictor is testable only with input codewords

Parity Prediction for an Adder

Operand *A*:		1	0	1	1	0	0	0	1	Parity 0
Operand *B*:		0	0	1	1	1	0	1	1	Parity 1
$A \oplus B$		1	0	0	0	1	0	1	0	
Carries:	0	0	1	1	0	0	1	1		Parity 0
Sum *S*:		1	1	1	0	1	1	0	0	Parity 1

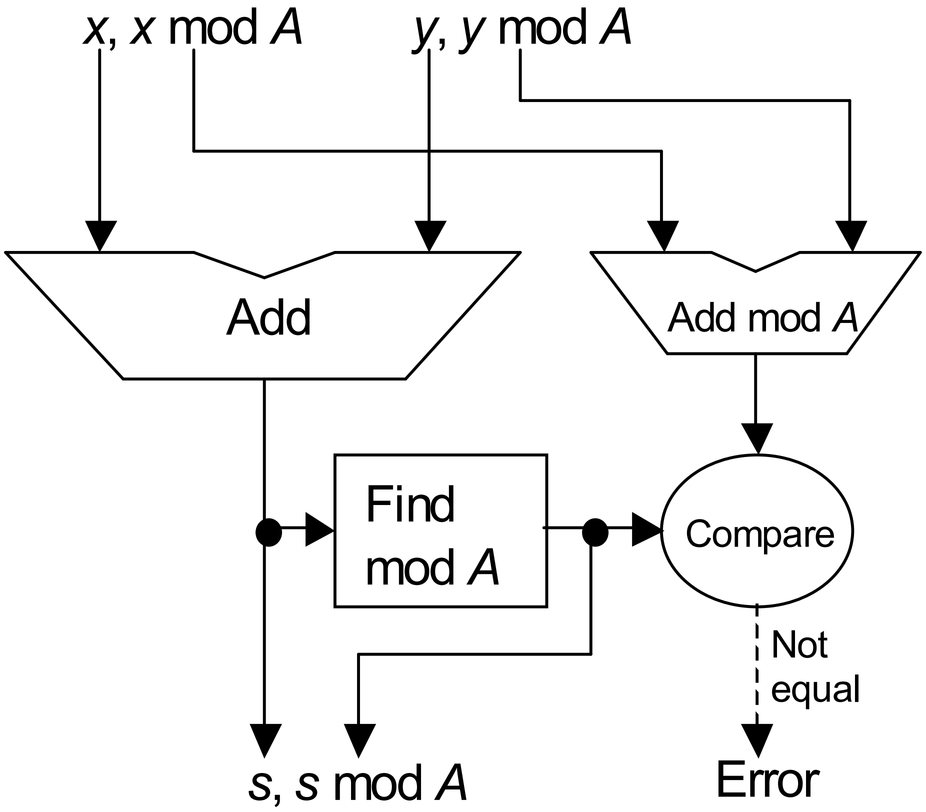


$$p(S) = \underbrace{p(A) \oplus p(B) \oplus c_0}_{\text{Inputs}} \oplus \underbrace{c_1 \oplus c_2 \oplus \; . \; . \; . \; \oplus c_k}_{\substack{\text{Must compute second} \\ \text{versions of these carries} \\ \text{to ensure independence}}}$$

Parity predictor for our adder consists of a duplicate carry network and an XOR tree

TSC Design with Residue Encoding

Residue checking is applicable directly to addition, subtraction, and multiplication, and with some extra effort to other arithmetic operations



To make this scheme TSC:

Modify the "Find mod A" box to produce the complement of the residue

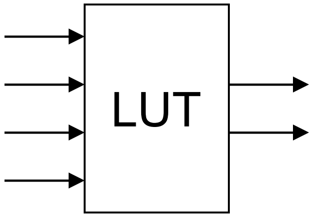
Use two-rail checker instead of comparator

Verify the self-testing property if the residue channel is not completely independent of the main computation (not needed for add/subtract and multiply)

Self-Checking Design with FPGAs

LUT-based FPGAs can suffer from the following fault types:

Single s-a faults in RAM cells
Single s-a faults on signal lines
Functional faults in a multiplexer within a single CLB
Functional faults in a D flip-flop within a single CLB
Single s-a faults in pass transistors connecting CLBs



Synthesis algorithm:

(1) Use scripts in the Berkeley synthesis tool SIS to decompose an SOP expression into an optimal collection of parts with 4 or fewer variables
(2) Assign each part to a functional cell that produces a 2-rail output
(3) Connect the outputs of a pair of intermediate functional cells to the inputs of a checker cell and find the output equations for that cell
(4) Cascade the checker cells to form a checker tree

Ref.: [Lala03]

Synthesis of TSC Systems from TSC Modules

System consists of a set of modules, with interconnections modeled by a directed graph

Theorem 1: A sufficient condition for a system to be TSC with respect to all single-module failures is to add checkers to the system such that if a path leads from a module M_i to itself (a loop), then it encounters at least one checker

Theorem 2: A sufficient condition for a system to be TSC with respect to all multiple module failures in the module set $A = \{M_i\}$ is to have no loop containing two modules in A in its path and at least one checker in any path leading from one module in A to any other module in A

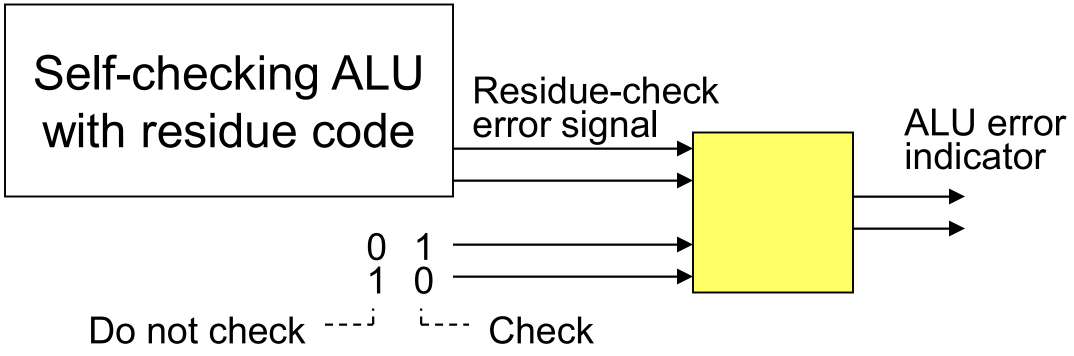
Optimal placement of checkers to satisfy these condition

Easily solved, when checker cost is the same at every interface

Partially Self-Checking Units

Some ALU functions, such as logical operations, cannot be checked using low-redundancy codes

Such an ALU can be made partially self-checking by circumventing the error-checking process in cases where codes are not applicable


Self-checking ALU with residue code
Residue-check error signal
ALU error indicator
0 1
1 0
Do not check
Check

01, 10 = G (top)
01 = D, 10 = C (bottom)
00, 11 = B

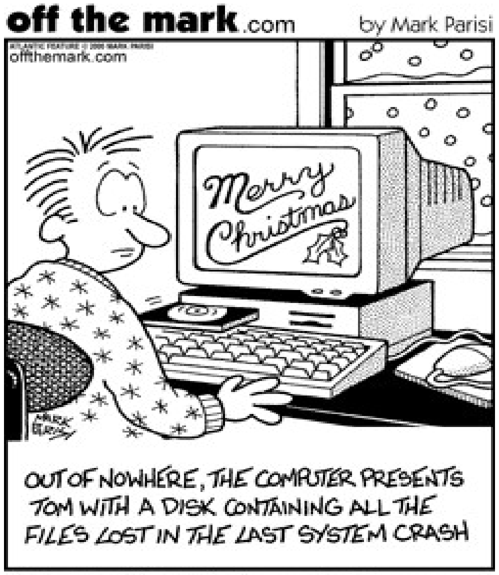
	In		Out	
	X	B	B	
	B	C	B	
	B	D	G	
Normal operation	G	C	G	01
	G	D	G	10

The check/do-not-check indicator is produced by the control unit

Redundant Disk Arrays

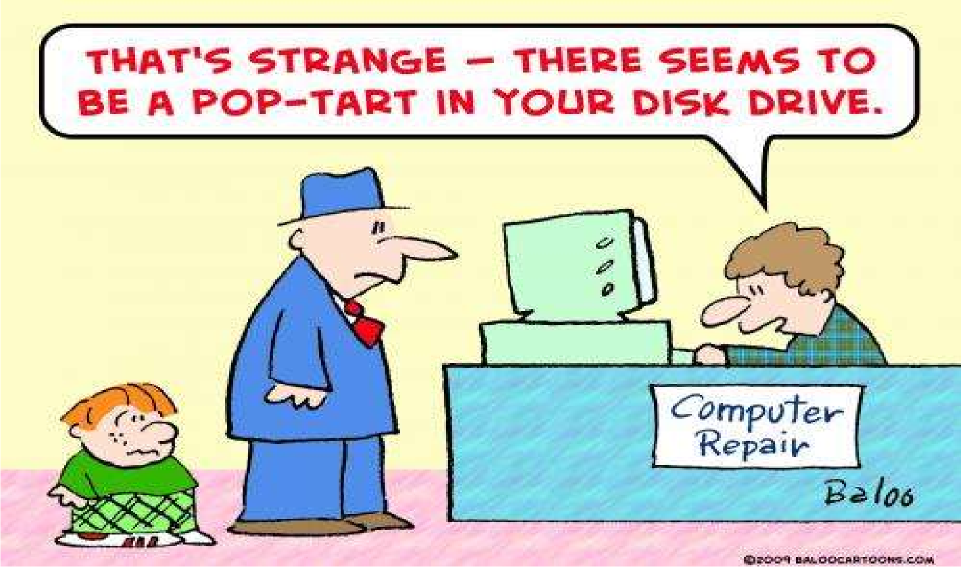
off the mark by Mark Parisi www.offthemark.com
OOPS, IT CRASHED. LUCKILY, I MADE A BACK-UP DISCUS...
ATLANTIC FEATURE SYND. ©2002 MARK PARISI
www.offthemark.com MarkParisi@aol.com 2-10
MARK PARISI

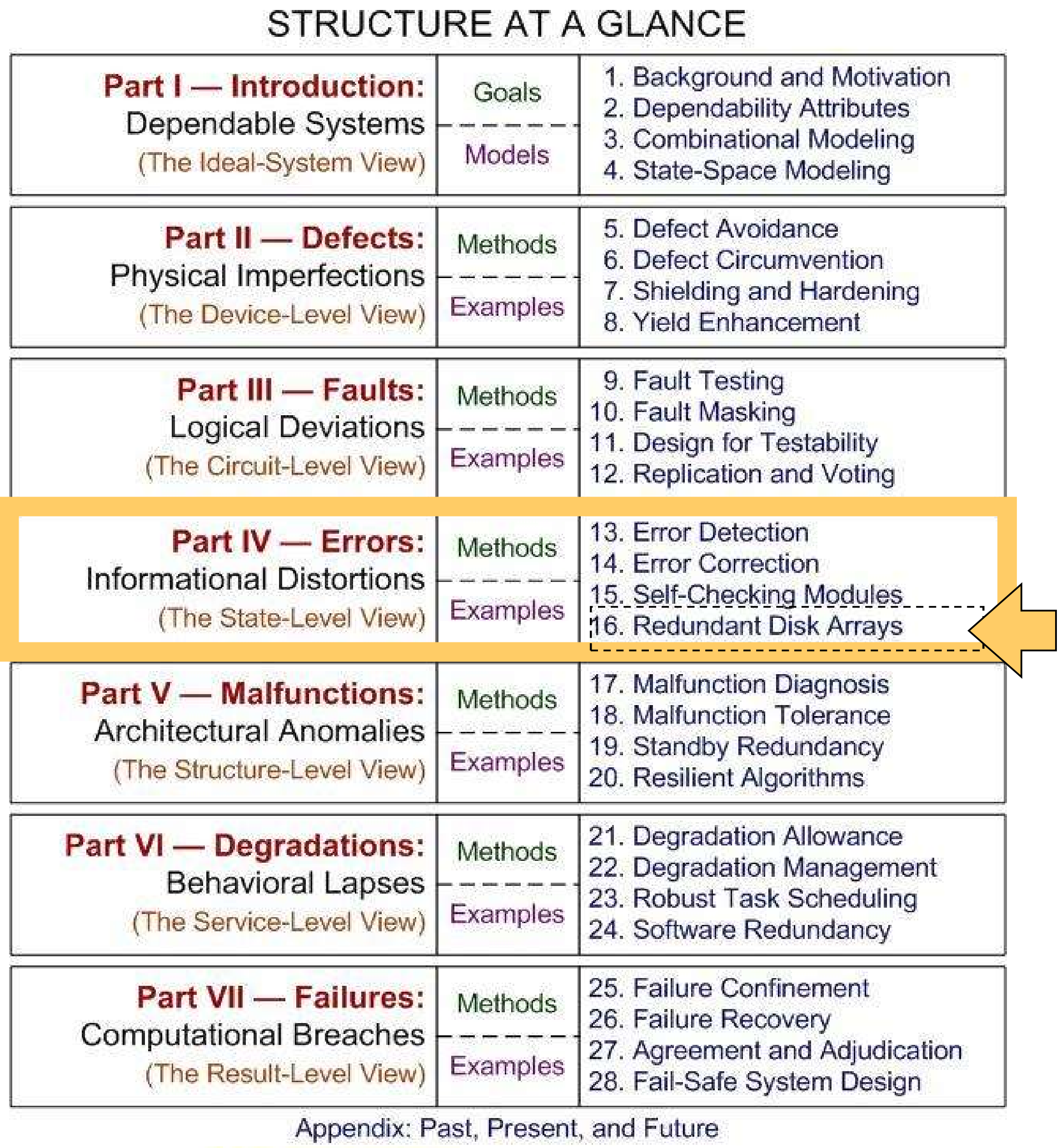
© Mark Parisi. Permission required for use.


off the mark.com by Mark Parisi
offthemark.com
Merry Christmas
OUT OF NOWHERE, THE COMPUTER PRESENTS TOM WITH A DISK CONTAINING ALL THE FILES LOST IN THE LAST SYSTEM CRASH

© Mark Parisi, Permission required for use.

"I want you to know that even if you pass away, I'll keep your hard drive forever."

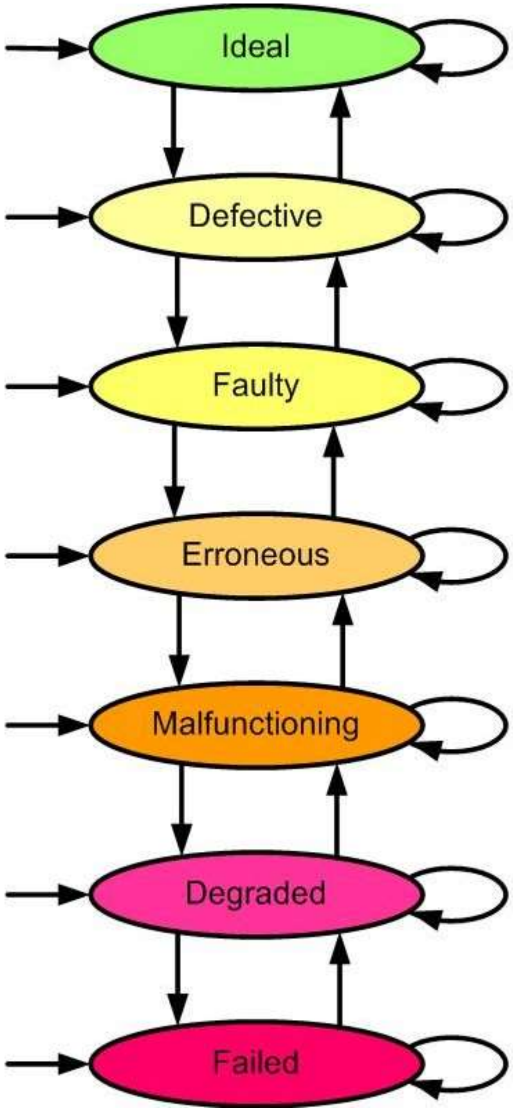

THAT'S STRANGE – THERE SEEMS TO BE A POP-TART IN YOUR DISK DRIVE.
Computer Repair
Baloo
©2009 BALOOCARTOONS.COM



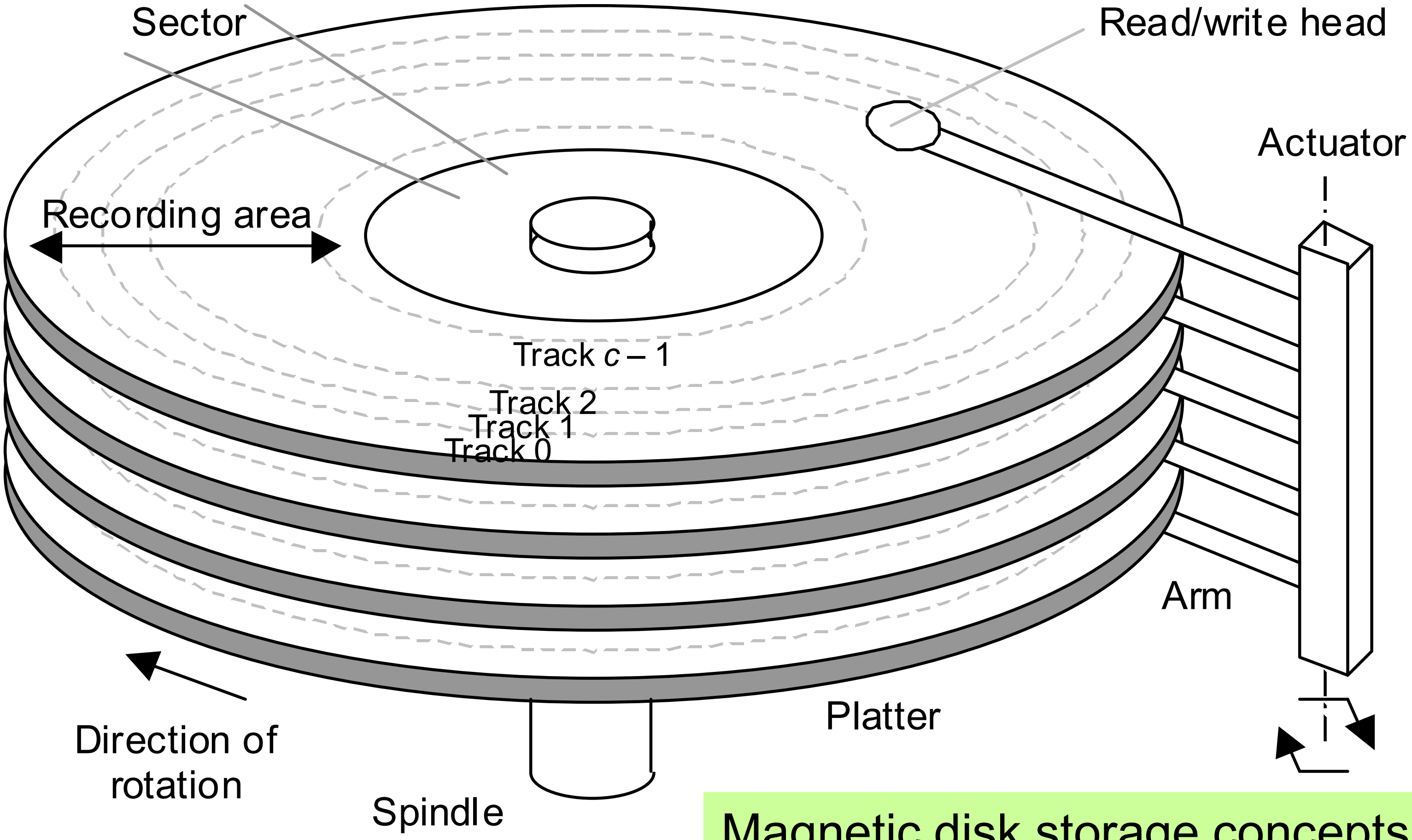
STRUCTURE AT A GLANCE

Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Resilient Algorithms
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods Examples	21. Degradation Allowance 22. Degradation Management 23. Robust Task Scheduling 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

Appendix: Past, Present, and Future



16.1 Disk Memory Basics



Magnetic disk storage concepts

Comprehensive info about disk memory: http://www.storagereview.com/guide/index.html

Typical Modern Hard-Disk Drives

Seagate BarraCuda
Toshiba X300
WD VelociRaptor
WD Blue Desktop
Seagate Firecuda Desktop
Seagate IronWolf NAS
Seagate FireCuda Mobile
WD My Book
G-Technology G-Drive

Price: Mostly under $100
Capacity: Mostly 1-20 TB
Cache: 64-256 MB
Data rate: ~ 6 Gbps

https://www.techradar.com/news/10-best-internal-desktop-and-laptop-hard-disk-drives-2016

Access Time for a Disk

Data transfer time = Bytes / Data rate

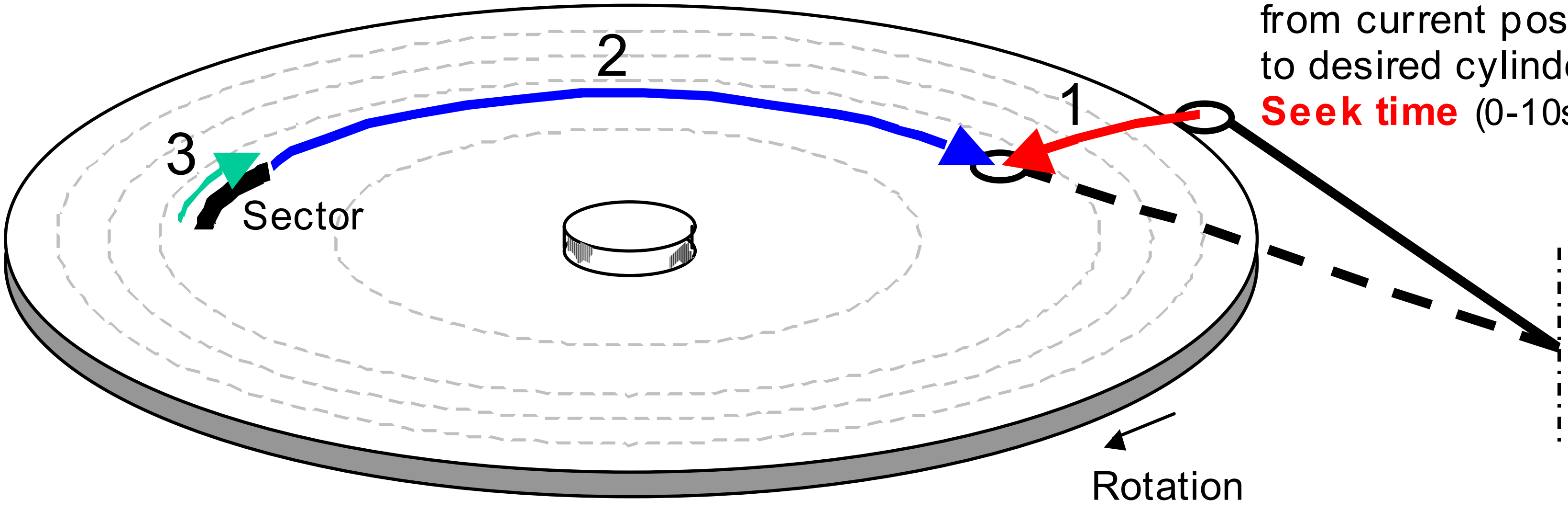
Average rotational latency = 30 000 / rpm (in ms)

Seek time = $a + b(c - 1) + \beta(c - 1)^{1/2}$

3. Disk rotation until sector has passed under the head: **Data transfer time** (< 1 ms)

2. Disk rotation until the desired sector arrives under the head: **Rotational latency** (0-10s ms)

1. Head movement from current position to desired cylinder: **Seek time** (0-10s ms)

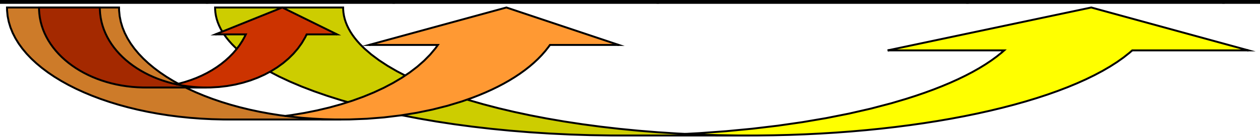


The three components of disk access time. Disks that spin faster have a shorter average and worst-case access time.

Amdahl’s Rules of Thumb for System Balance

The need for high-capacity, high-throughput secondary (disk) memory

Processor speed	RAM size	Disk I/O rate	Number of disks	Disk capacity	Number of disks
1 GIPS	1 GB	100 MB/s	1	100 GB	1
1 TIPS	1 TB	100 GB/s	1000	100 TB	100
1 PIPS	1 PB	100 TB/s	1 Million	100 PB	100 000
1 EIPS	1 EB	100 PB/s	1 Billion	100 EB	100 Million



1 RAM byte for each IPS

1 I/O bit per sec for each IPS

100 disk bytes for each RAM byte

G	Giga
T	Tera
P	Peta
E	Exa

Head-Per-Track Disks

Dedicated track heads eliminate seek time (replace it with activation time for a head)

Multiple sets of head reduce rotational latency

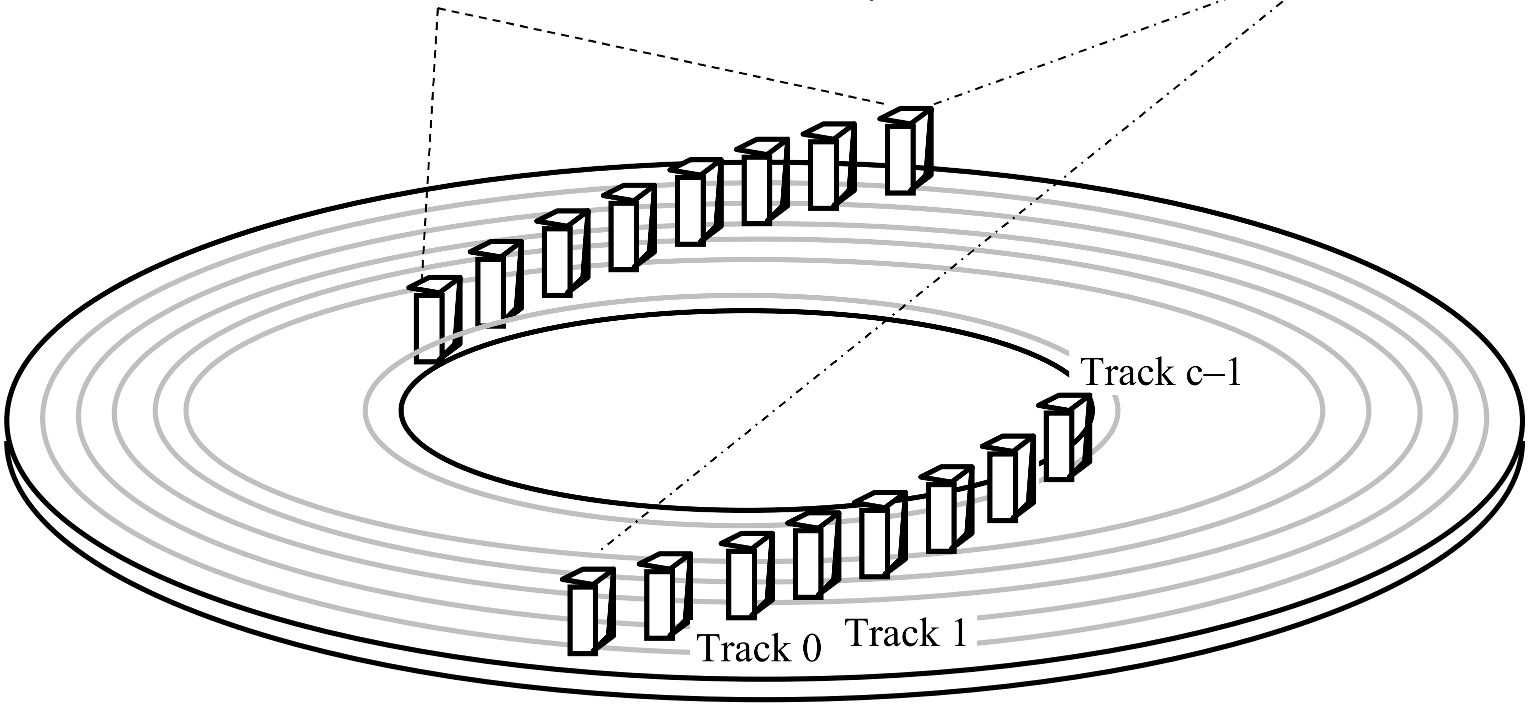
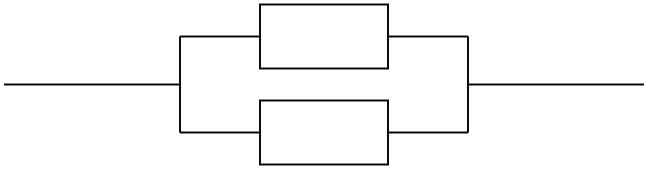


Fig. 18.7 Head-per-track disk concept.

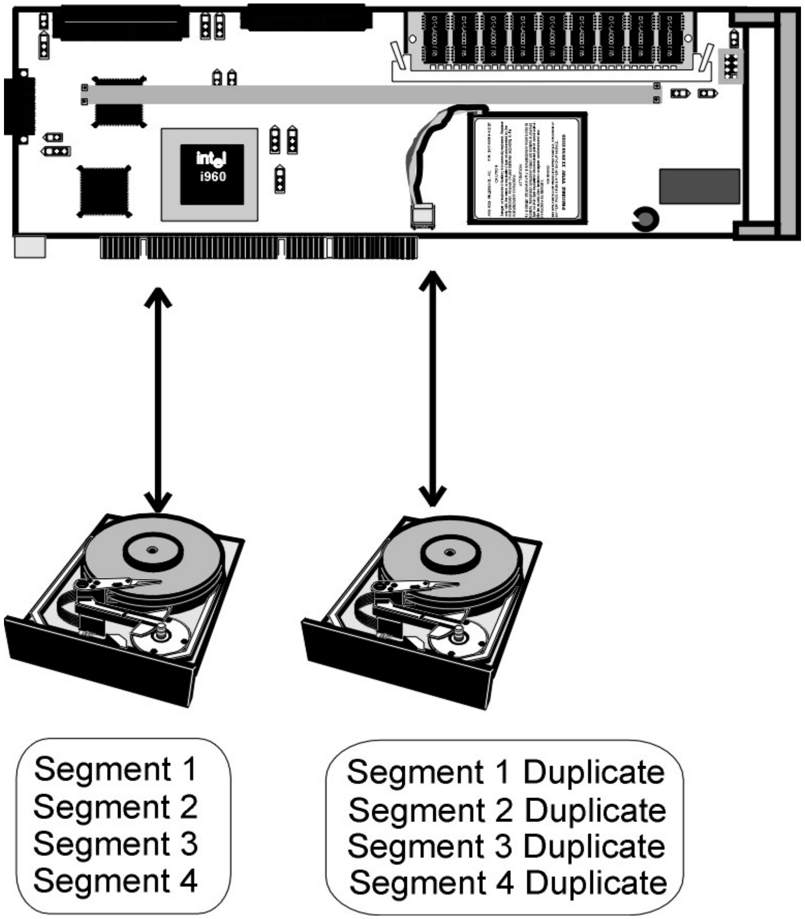
16.2 Disk Mirroring and Striping

Mirroring means simple duplication

Disadvantage:
No gain in performance
or bandwidth



Advantage:
Parallel system,
highly reliable

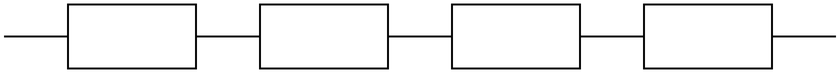


http://www.recoverdata.com/images/raid_mirror.gif

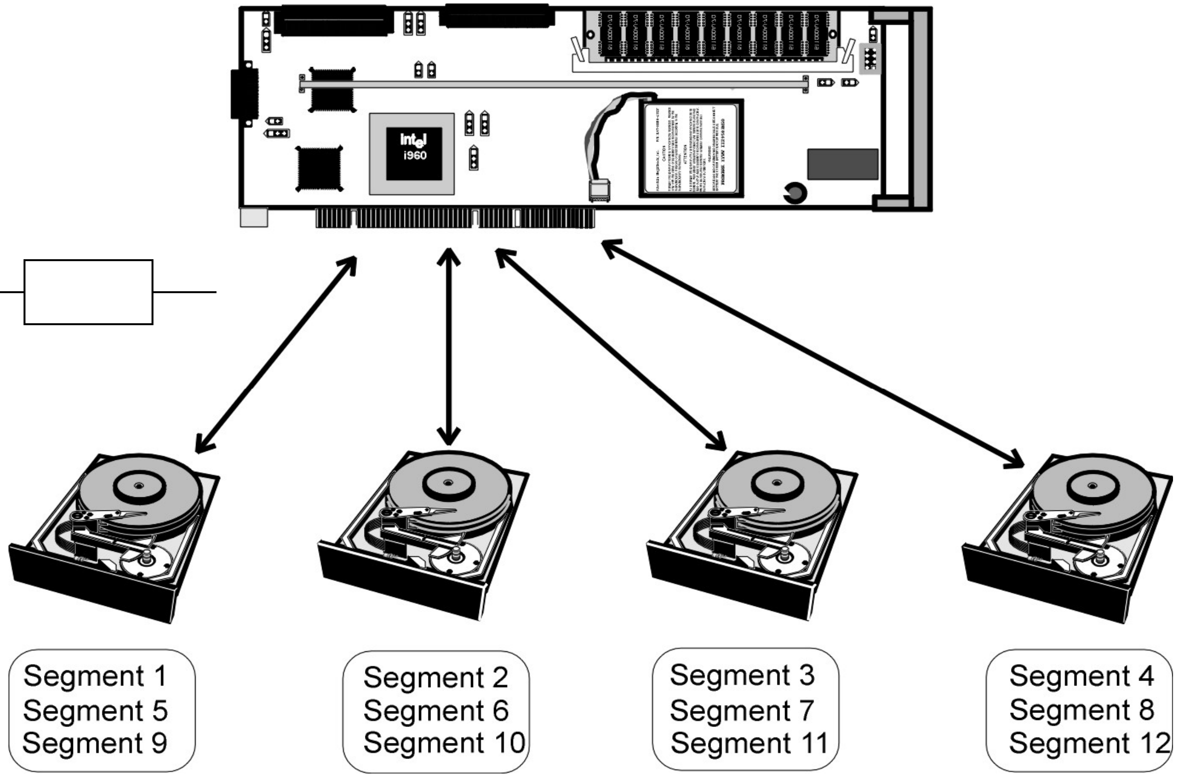
Disk Striping

Striping means dividing a block of data into smaller pieces (perhaps down to the bit level) and storing the pieces on different disks

Advantage:
Faster (parallel)
access to data



Disadvantage:
Series system,
less reliable



Segment 1
Segment 5
Segment 9

Segment 2
Segment 6
Segment 10

Segment 3
Segment 7
Segment 11

Segment 4
Segment 8
Segment 12

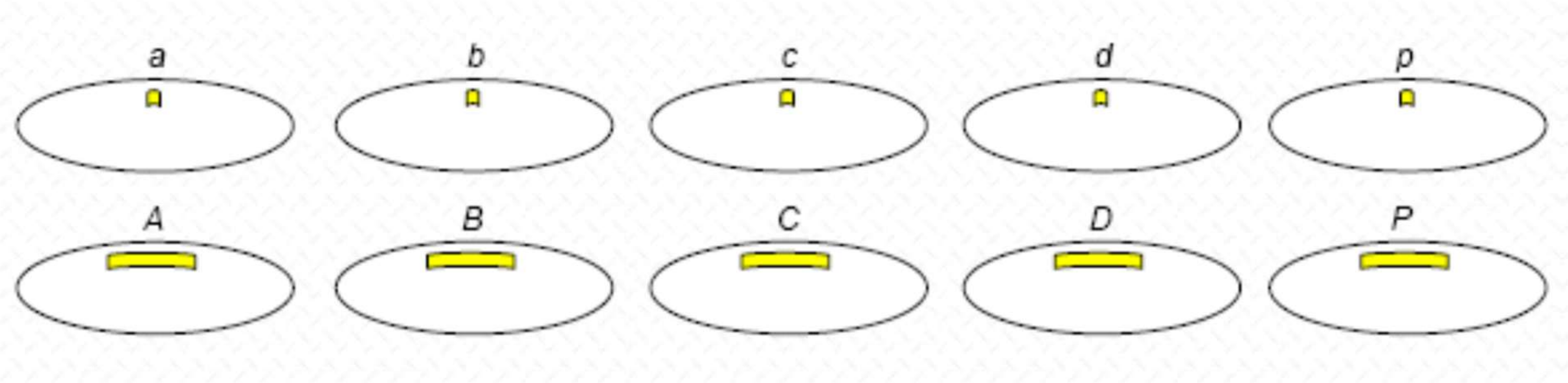
http://www.recoverdata.com/images/raid_striping.gif

16.3 Data Encoding Schemes

Simplest possible encoding: data duplication

Error-correcting code: An overkill, because disk errors are of erasure type (strong built-in error-detecting code indicates error location)

Parity, applied to bits or blocks: $P = A \oplus B \oplus C \oplus D$

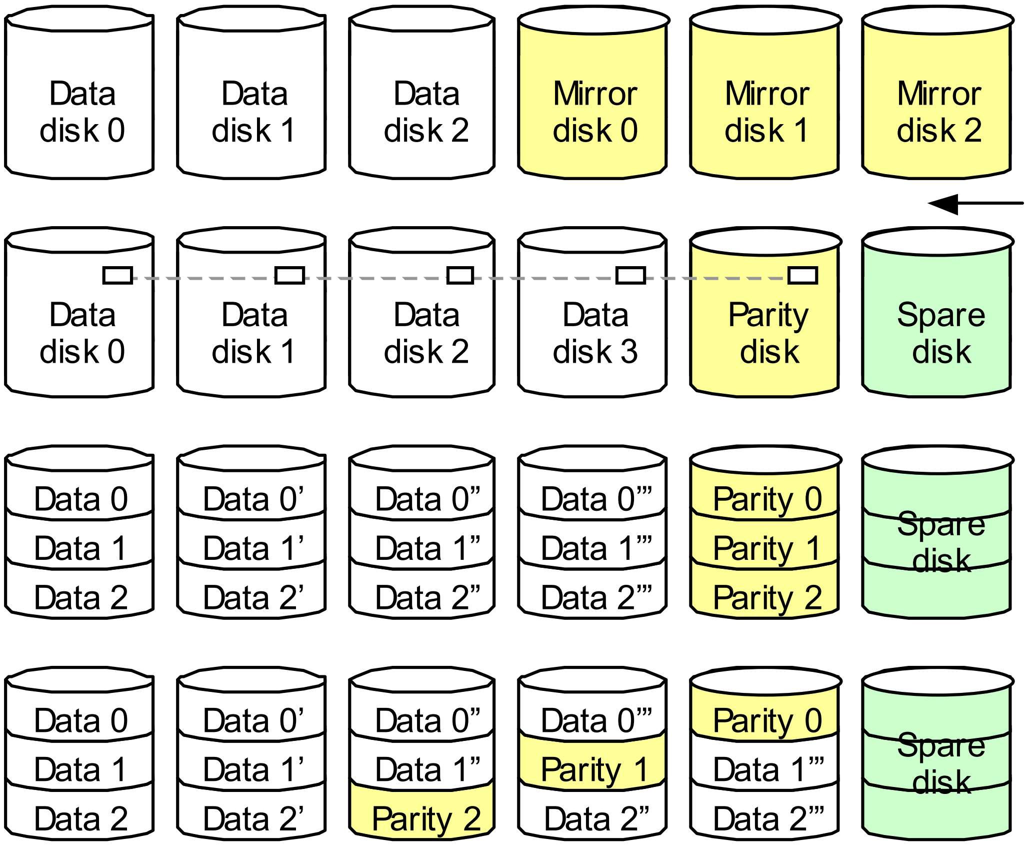


Data reconstruction: Suppose *B* is lost or erased

$B = A \oplus C \oplus D \oplus P$

16.4 The RAID Levels

Data organization on multiple disks



RAID0: Multiple disks for higher data rate; no redundancy

RAID1: Mirrored disks

RAID2: Error-correcting code

RAID3: Bit- or byte-level striping with parity/checksum disk

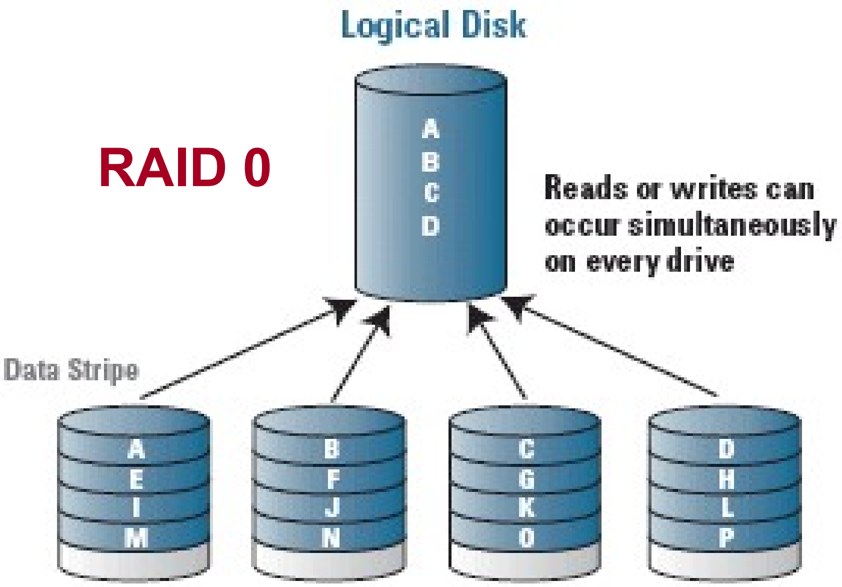
RAID4: Parity/checksum applied to sectors,not bits or bytes

RAID5: Parity/checksum distributed across several disks

RAID6: Parity and 2nd check distributed across several disks

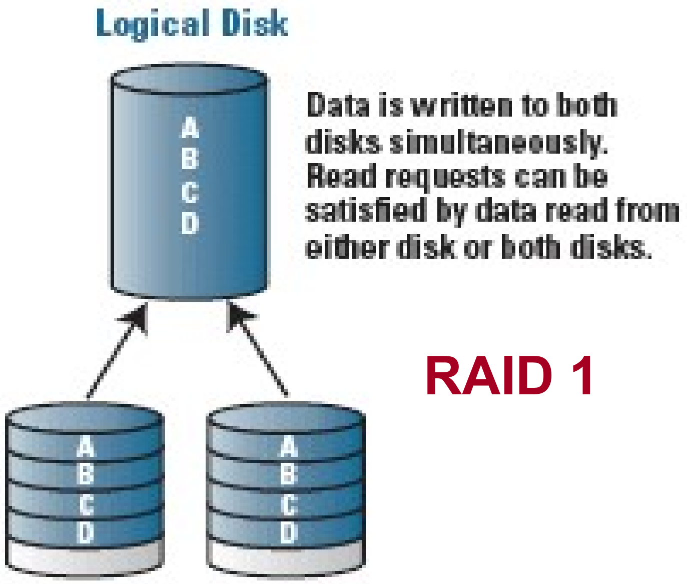
Alternative data organizations on redundant disk arrays.

RAID Levels 0 and 1



RAID 0

Reads or writes can occur simultaneously on every drive



Data is written to both disks simultaneously. Read requests can be satisfied by data read from either disk or both disks.

RAID 1

Structure: Striped (data broken into blocks & written to separate disks)

Advantages: Spreads I/O load across many channels and drives

Drawbacks: No error tolerance (data lost with single disk failure)

Structure: Each disk replaced by a mirrored pair

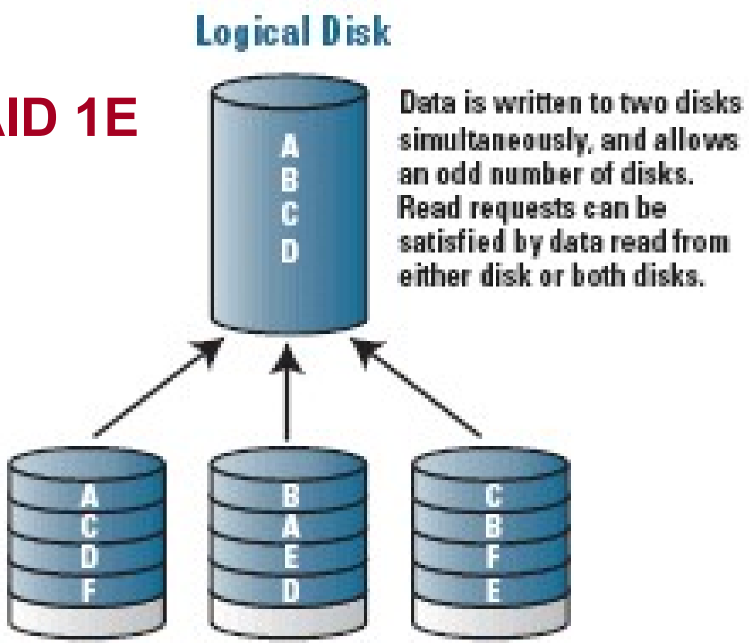
Advantages: Can double the read transaction rate; no rebuild required

Drawbacks: Overhead is 100%

Diagrams: http://ironraid.com/whatisraid.htm

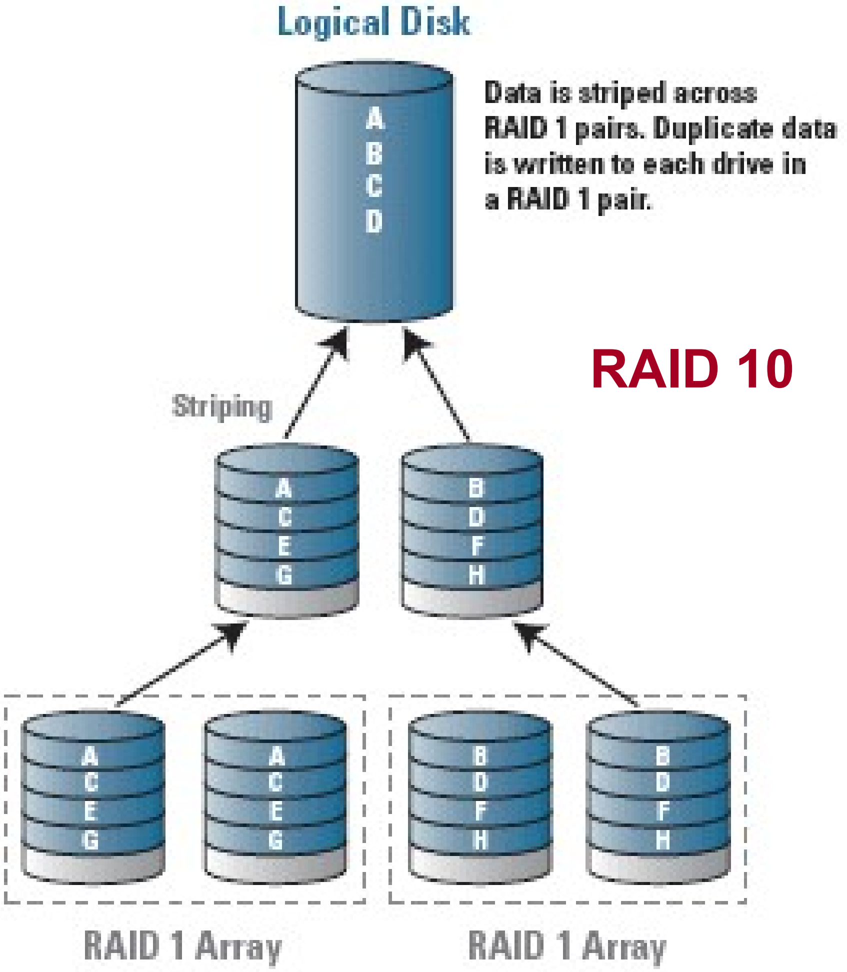
Combining RAID Levels 0 and 1

RAID 1E



Data is written to two disks simultaneously, and allows an odd number of disks. Read requests can be satisfied by data read from either disk or both disks.

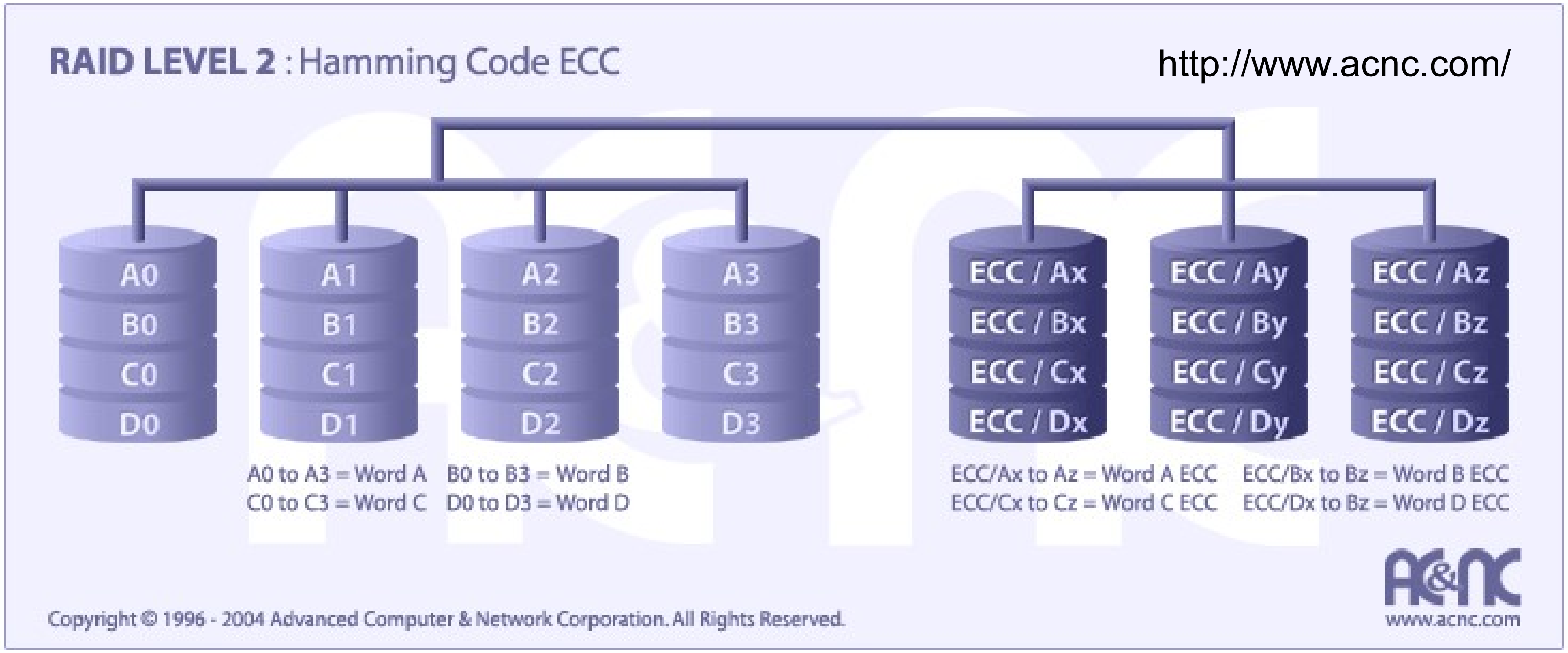
RAID 10



Data is striped across RAID 1 pairs. Duplicate data is written to each drive in a RAID 1 pair.

Diagrams: http://ironraid.com/whatisraid.htm

RAID Level 2



Structure:

Data bits are written to separate disks and ECC bits to others

Advantages:

On-the-fly correction

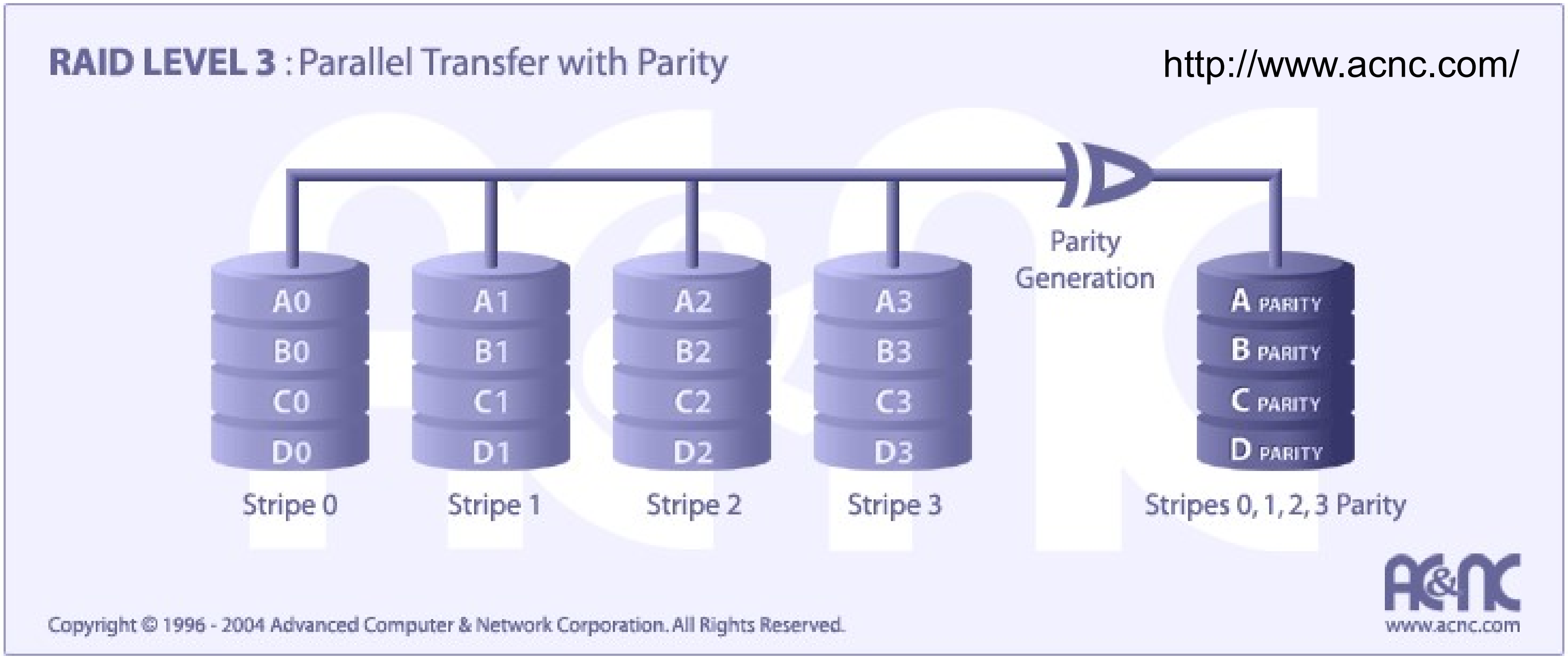
High transfer rates possible (w/ sync)

Drawbacks:

Potentially high redundancy

High entry-level cost

RAID Level 3



Structure:

Data striped across several disks, parity provided on another

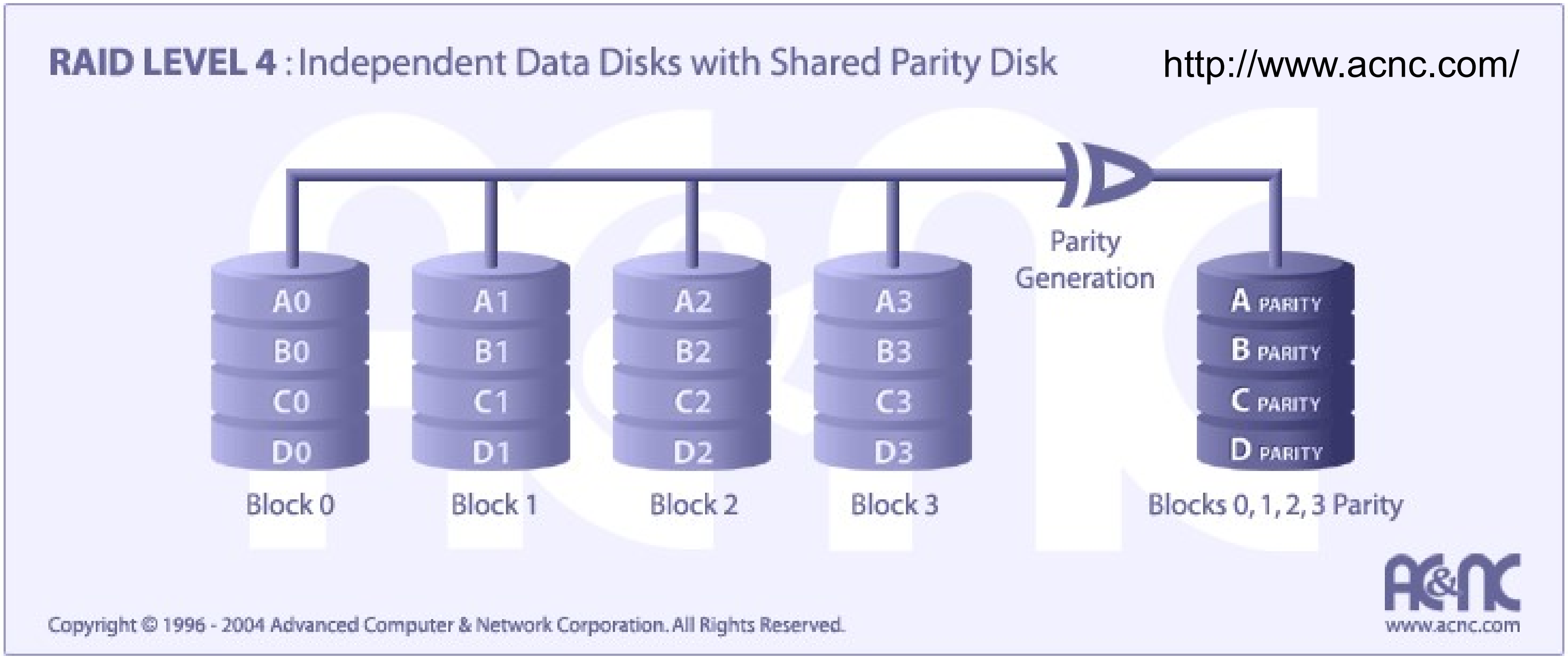
Advantages:

Maintains good throughput even when a disk fails

Drawbacks:

Parity disk forms a bottleneck
Complex controller

RAID Level 4



Structure:

Independent blocks on multiple disks share a parity disk

Advantages:

Very high read rate
Low redundancy

Drawbacks:

Low write rate
Inefficient data rebuild

RAID Level 5

Structure:

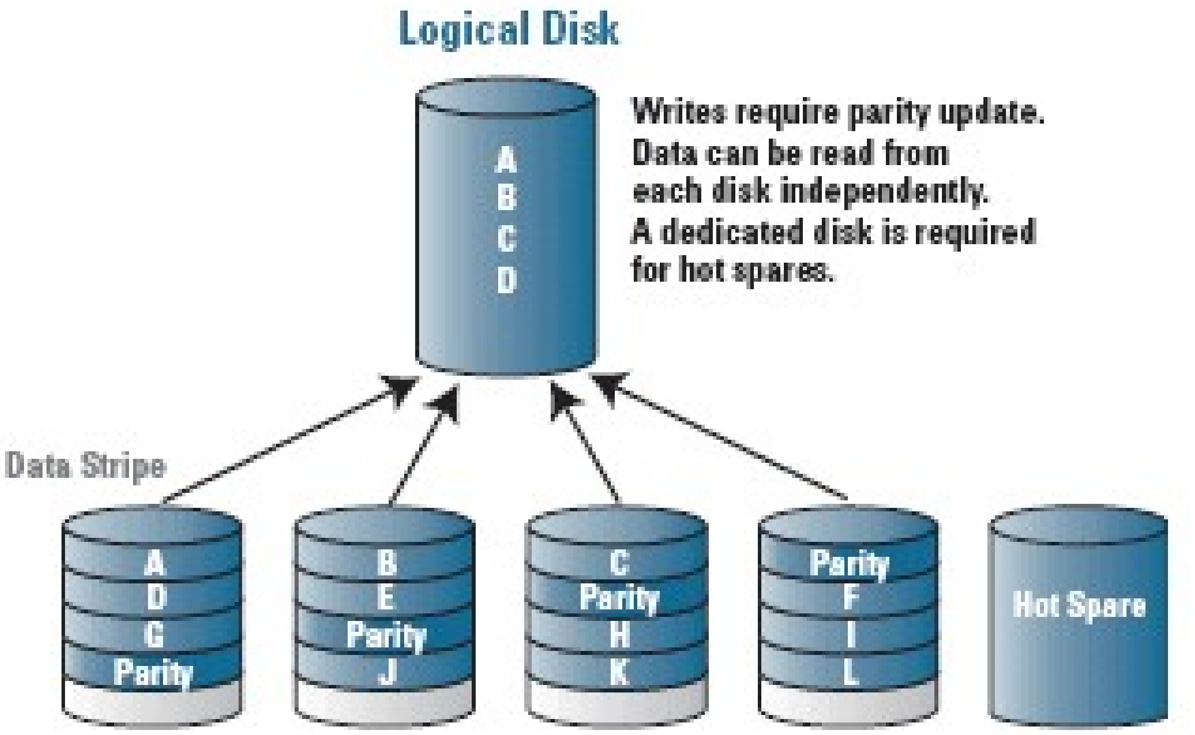
Parity and data blocks distributed on multiple disks

Advantages:

Very high read rate
Medium write rate
Low redundancy

Drawbacks:

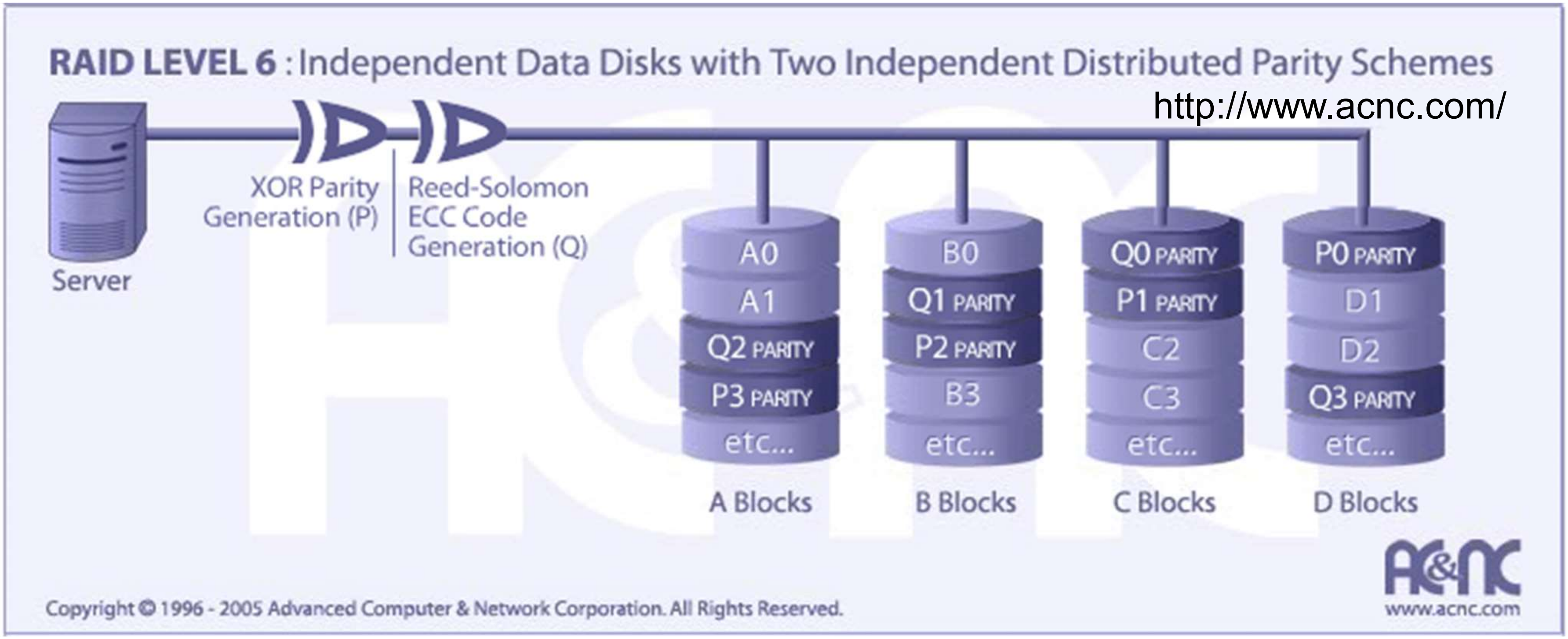
Complex controller
Difficult rebuild



Variant: The spare is also active and the spare capacity is distributed on all drives; particularly attractive with small arrays

Diagrams: http://ironraid.com/whatisraid.htm

RAID Level 6



http://www.acnc.com/

Structure:

RAID Level 5, extended with second parity check scheme

Advantages:

Tolerates 2 failures
Protected even during recovery

Drawbacks:

More complex controller
Greater overhead

16.5 Disk Array Performance

Disk array performance has two components:
1. Speed during normal read and write operations
2. Speed of reconstruction (also affects reliability)

Data reconstruction

$P = A \oplus B \oplus C \oplus D \quad \Rightarrow \quad B = A \oplus C \oplus D \oplus P$

To reconstruct B, we must read all other data blocks and the parity block

The reconstruction time penalty and the "small write" penalty have led some to reject all parity-based RAID schemes

BAARF = Battle Against Any RAID-F (Free, Four, Five): www.baarf.com

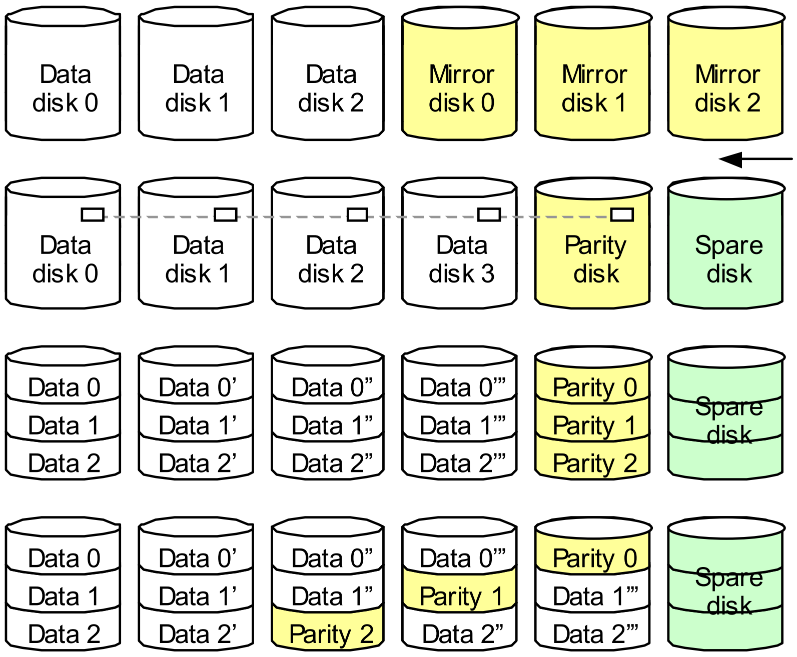
The Write Problem in Disk Arrays

Parity updates may become a bottleneck, because the parity changes with every write, no matter how small

Computing sector parity for a write operation:

New parity = New data ⊕ Old data ⊕ Old parity

Data organization on multiple disks



RAID0: Multiple disks for higher data rate; no redundancy

RAID1: Mirrored disks

RAID2: Error-correcting code

RAID3: Bit- or byte-level striping with parity/checksum disk

RAID4: Parity/checksum applied to sectors,not bits or bytes

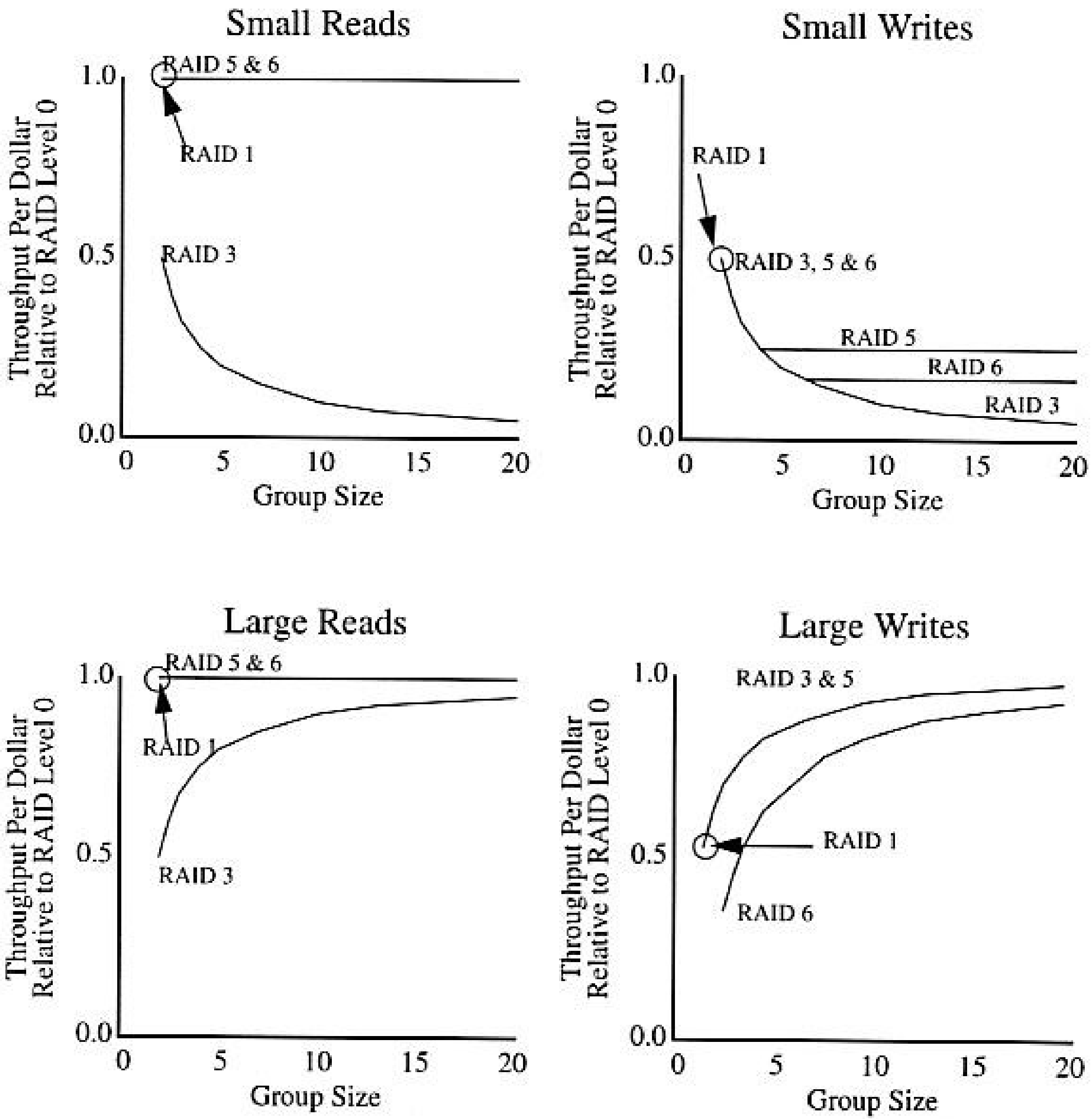
RAID5: Parity/checksum distributed across several disks

RAID6: Parity and 2nd check distributed across several disks

RAID Tradeoffs

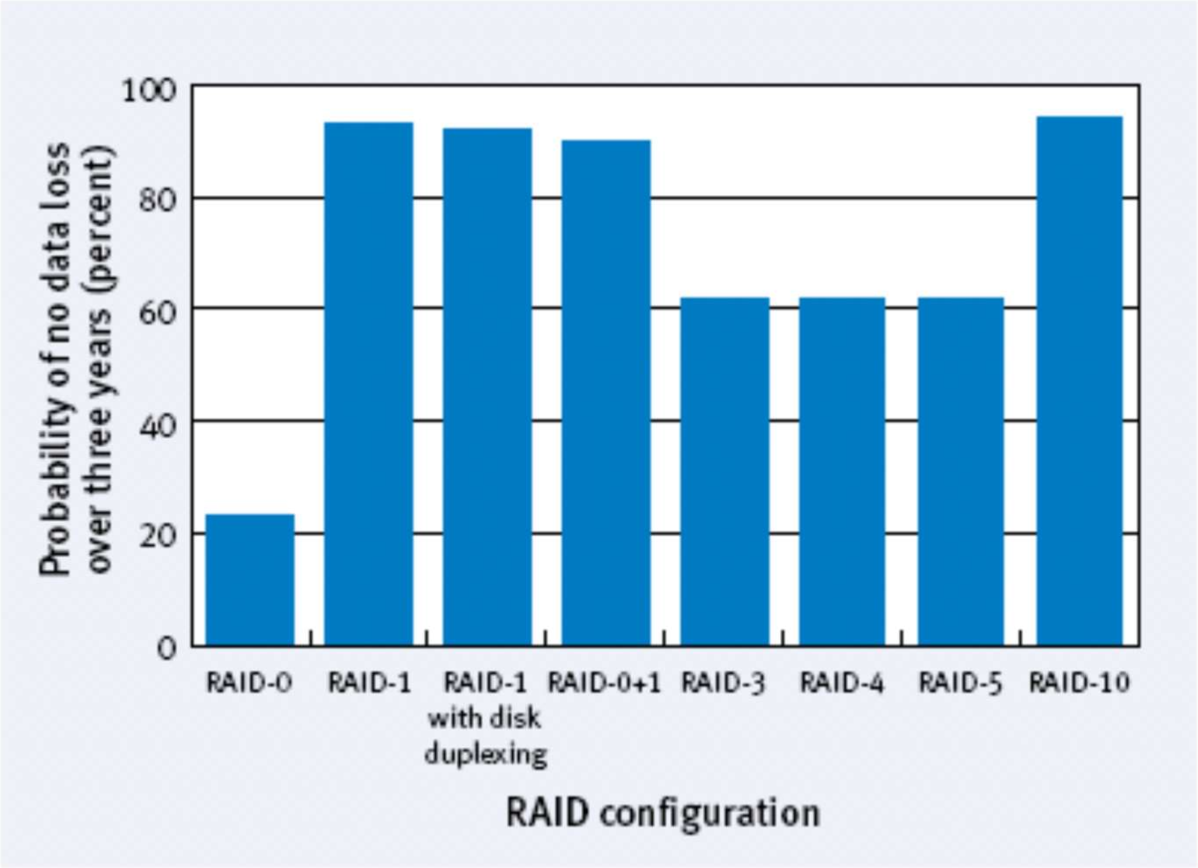
RAID5 and RAID 6 impose little penalty on read operations

In choosing the group size, balance must be struck between the increasing penalty for small writes vs. decreasing penalty for large writes



Figures from: [Chen94]

16.6 Disk Array Reliability Modeling



From: http://www.vinastar.com/docs/tls/Dell_RAID_Reliability_WP.pdf

MTTF Calculation for Disk Arrays

RAID1: $$\frac{\text{MTTF}^2}{2\ \text{MTTR}}$$

RAID5: $$\frac{\text{MTTF}^2}{N(G-1)\ \text{MTTR}}$$

RAID6: $$\frac{\text{MTTF}^3}{N(G-1)(G-2)\ \text{MTTR}^2}$$

Notation:

MTTF is for one disk
MTTR is different for each level
N = Total number of disks
G = Disks in a parity group

Caveat: RAID controllers (electronics) are also subject to failures and their reported MTTF is surprisingly small (on the order of 0.2 to 2 M hr). Also, must account for errors that go undetected by the disk's error code.

Actual Redundant Disk Arrays