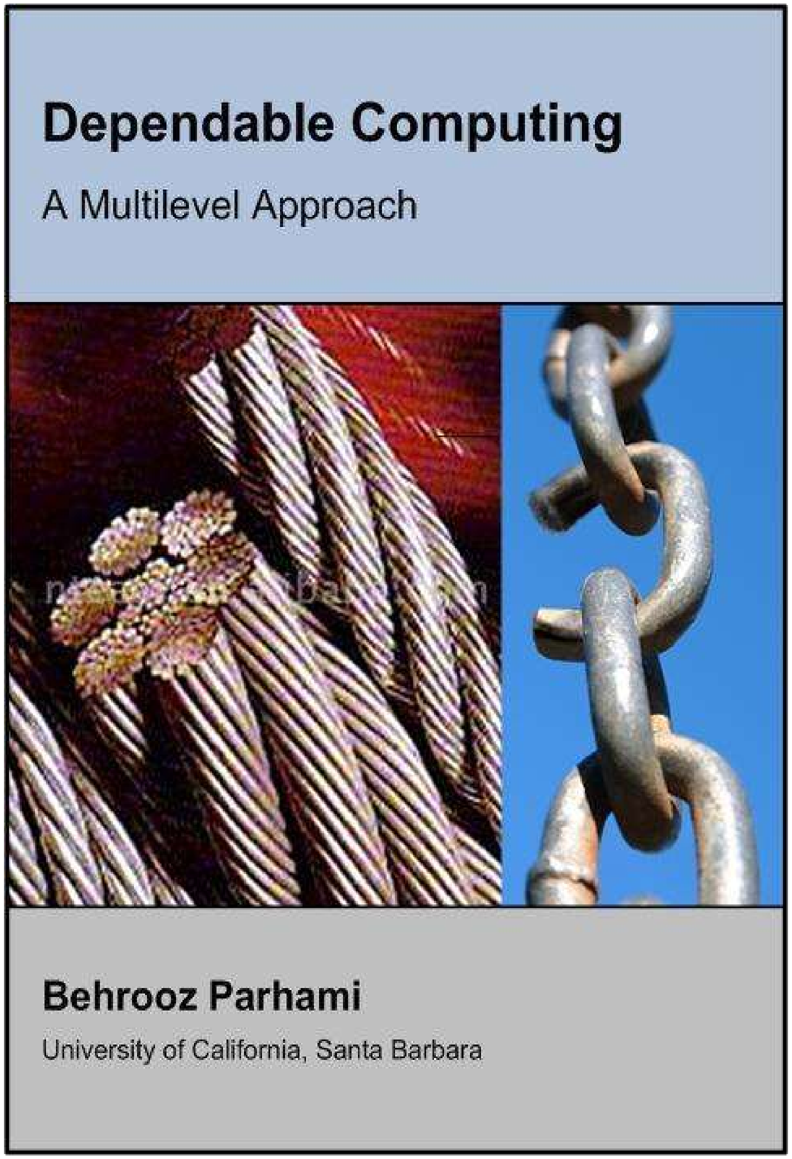


Dependable Computing

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

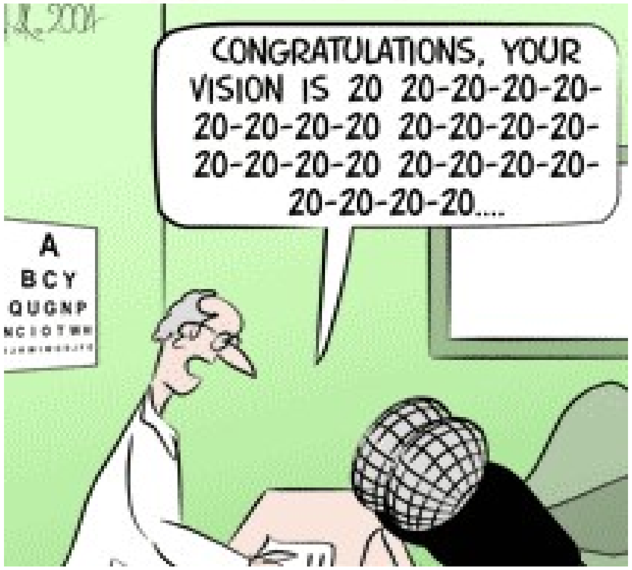
This presentation is intended to support the use of the textbook *Dependable Computing: A Multilevel Approach* (traditional print or on-line open publication, TBD). It is updated regularly by the author as part of his teaching of the graduate course ECE 257A, Fault-Tolerant Computing, at Univ. of California, 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. © Behrooz Parhami

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

9 Fault Testing

45 METRES OF ROPE AND YOU'RE FINDING FAULT WITH THIS LITTLE BIT ?

PEAS ARE COOKED
POTATOES ARE COOKED
BISCUITS ARE COOKED
TURKEY IS COOKED
CARROTS ARE COOKED
BUILT-IN SELF TEST

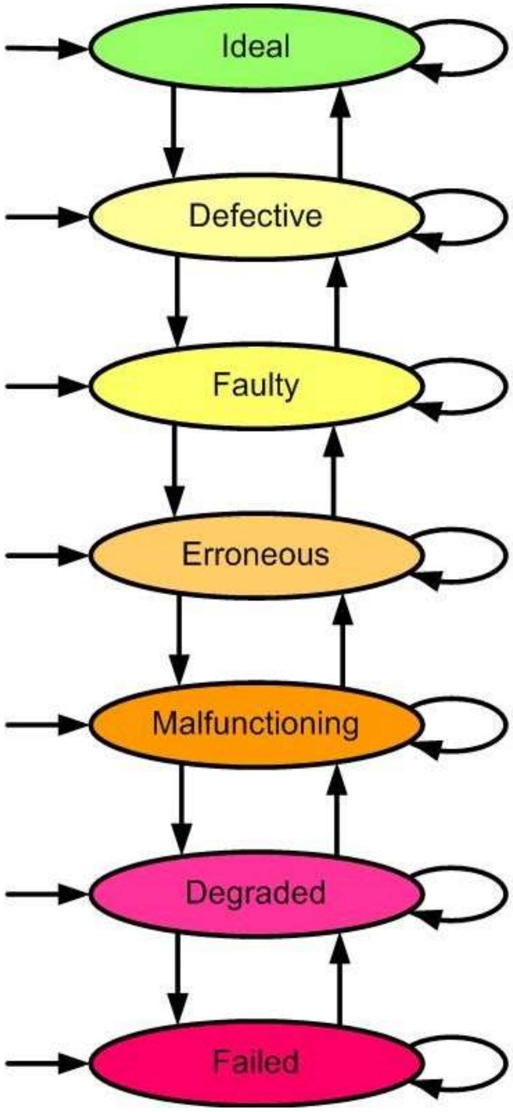

CONGRATULATIONS, YOUR VISION IS 20 20-20-20-20-20-20-20-20 20-20-20-20-20-20-20-20 20-20-20-20-20-20-20-20....
A
BCY
QUGNP

The good news is that the tests don't show any other problems

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

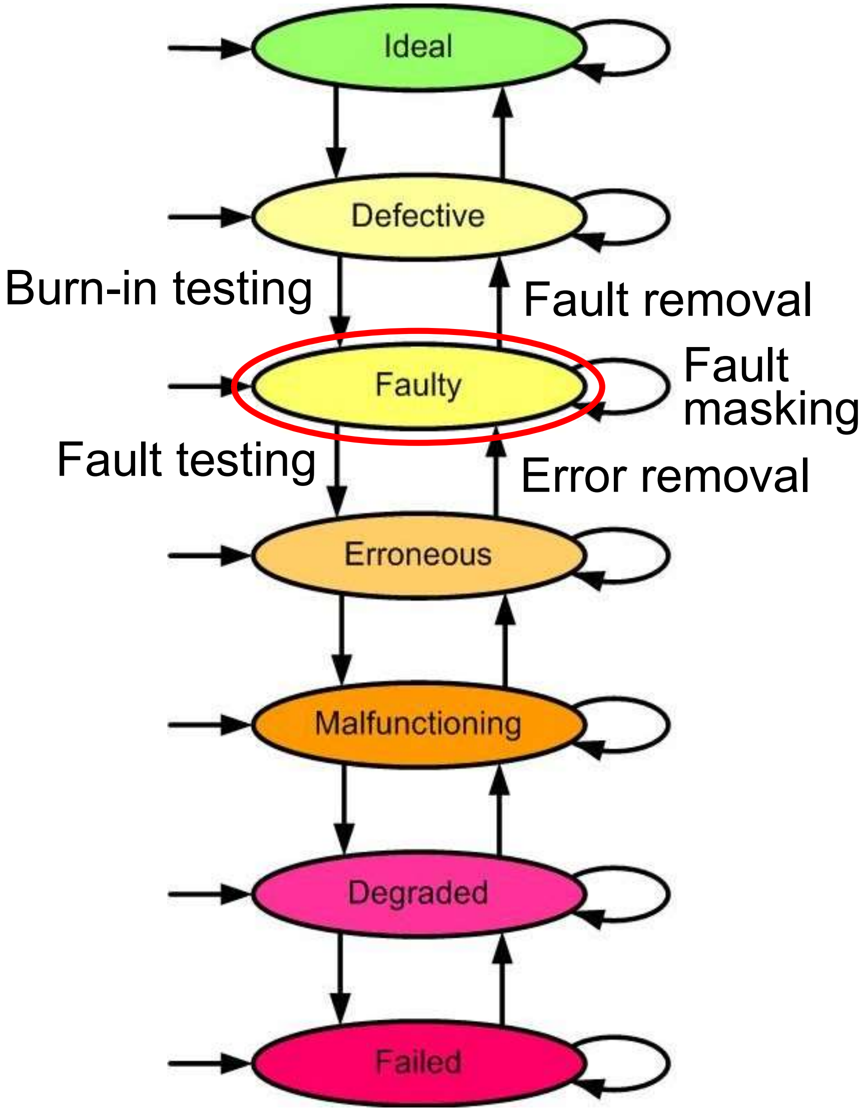
Appendix: Past, Present, and Future



9.1 Overview and Fault Models

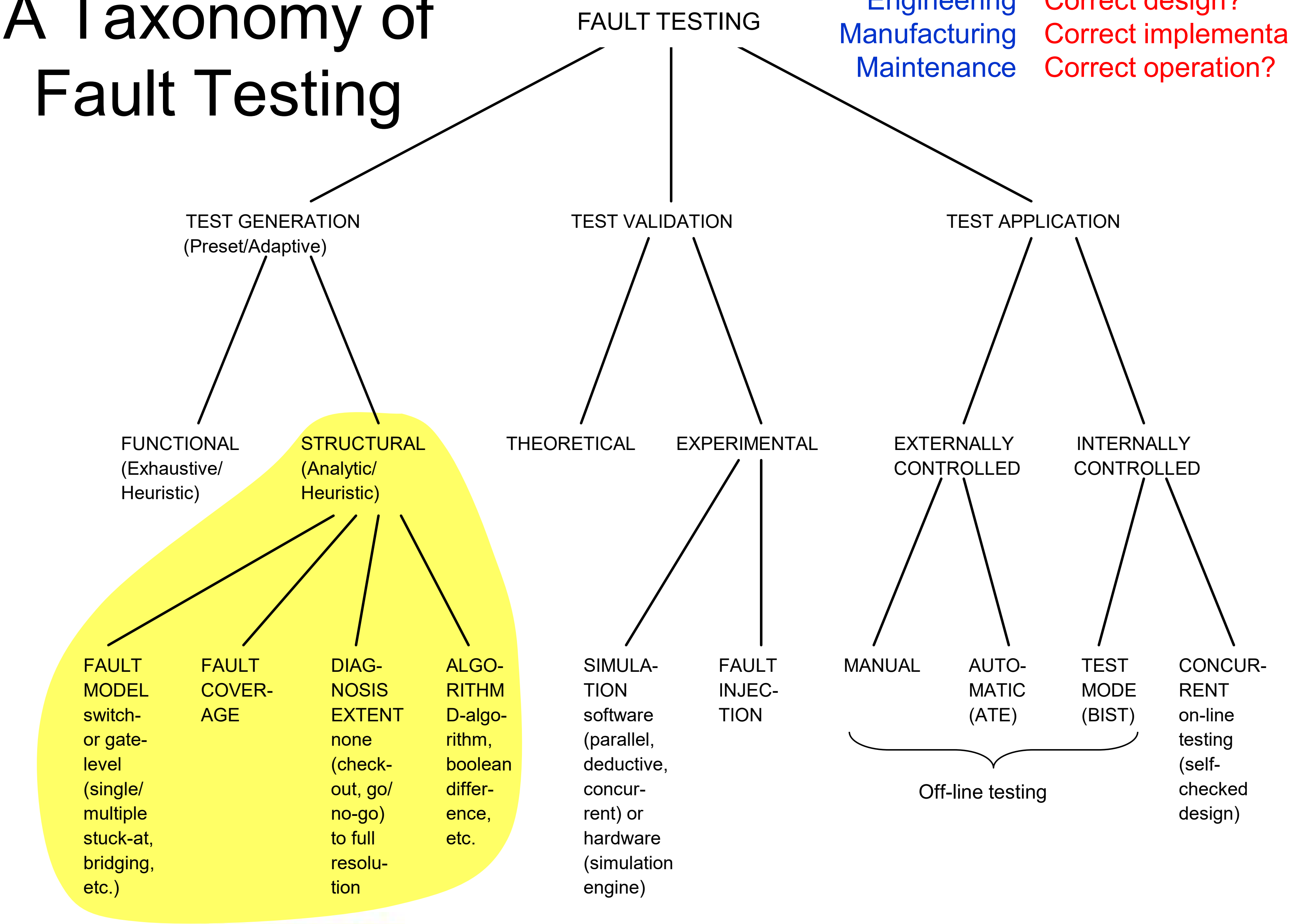
The faulty state

and transitions into and out of it



A Taxonomy of Fault Testing

Engineering	Correct design?
Manufacturing	Correct implementation?
Maintenance	Correct operation?



FAULT TESTING

TEST GENERATION (Preset/Adaptive)

FUNCTIONAL (Exhaustive/ Heuristic)

STRUCTURAL (Analytic/ Heuristic)

FAULT MODEL switch- or gate-level (single/ multiple stuck-at, bridging, etc.)

FAULT COVERAGE

DIAGNOSIS EXTENT none (check-out, go/ no-go) to full resolution

ALGORITHM D-algorithm, boolean difference, etc.

TEST VALIDATION

THEORETICAL

EXPERIMENTAL

SIMULATION software (parallel, deductive, concurrent) or hardware (simulation engine)

FAULT INJECTION

TEST APPLICATION

EXTERNALLY CONTROLLED

MANUAL

AUTOMATIC (ATE)

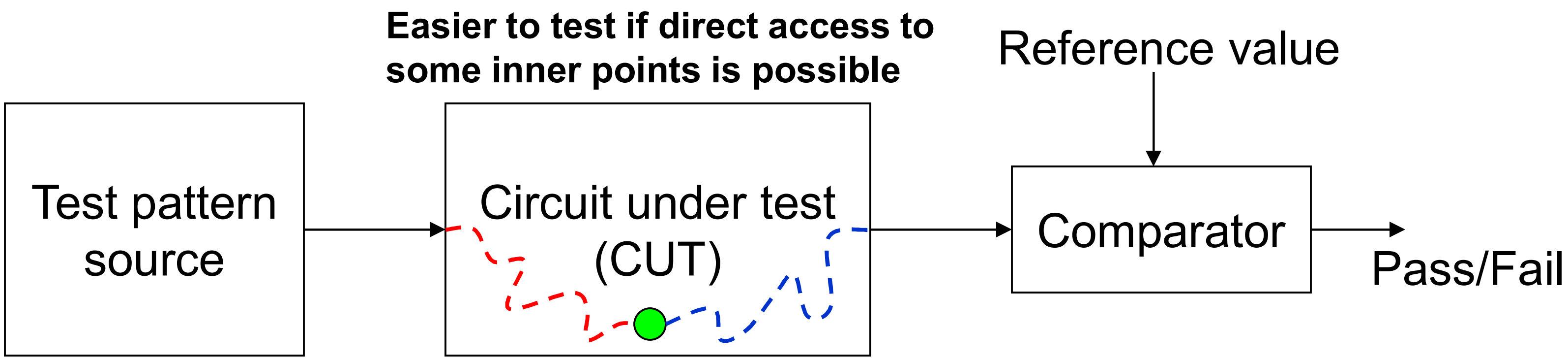
INTERNALLY CONTROLLED

TEST MODE (BIST)

CONCURRENT on-line testing (self-checked design)

Off-line testing

Requirements and Setup for Testing



Testability requires **controllability** and **observability**
(redundancy may reduce testability if we are not careful; e.g., TMR)

Reference value can come from a “gold” version or from a table

Test patterns may be randomly generated, come from a preset list, or be selected according to previous test outcomes

Test results may be compressed into a “signature” before comparing

Test application may be off-line or on-line (concurrent)

Importance and Limitations of Testing

Important to detect faults as early as possible
Approximate cost of catching a fault at various levels

Component	$1
Board	$10
System	$100
Field	$1000

Test coverage may be well below 100% (model inaccuracies and impossibility of dealing with all combinations of the modeled faults)

“Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often.”
Steve C. McConnell

“Program testing can be used to show the presence of bugs, but never to show their absence!” Edsger W. Dijkstra

Fault Models at Different Abstraction Levels

Fault model is an abstract specification of the types of deviations in logic values that one expects in the circuit under test

Can be specified at various levels: transistor, **gate**, function, system

Transistor-level faults

- Caused by defects, shorts/opens, electromigration, transients, . . .
- May lead to high current, incorrect output, intermediate voltage, . . .
- Modeled as stuck-on/off, bridging, delay, coupling, crosstalk faults
- Quickly become intractable because of the large model space

Function-level faults

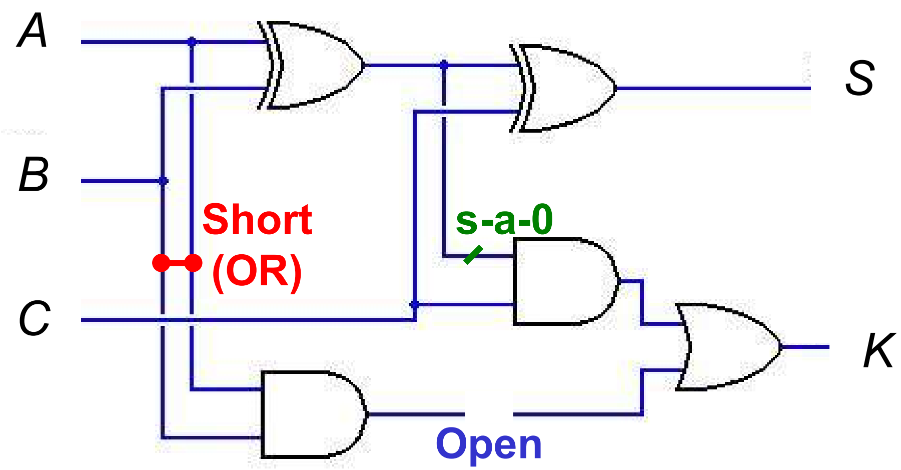
- Selected in an ad hoc manner based on the function of a block (decoder, ALU, memory)

System-level faults (malfunctions, in our terminology)

- Will discuss later in Part V

Gate- or Logic-Level Fault Models

Most popular models (due to their accuracy and relative tractability)



Line stuck faults
- Stuck-at-0 (s-a-0)
- Stuck-at-1 (s-a-1)

Line bridging faults
- Unintended connection (wired OR/AND)

Line open faults
- Often can be modeled as s-a-0 or s-a-1

Delay faults (less tractable than the previous fault types)
- Signals experience unusual delays

Other faults
- Coupling, crosstalk

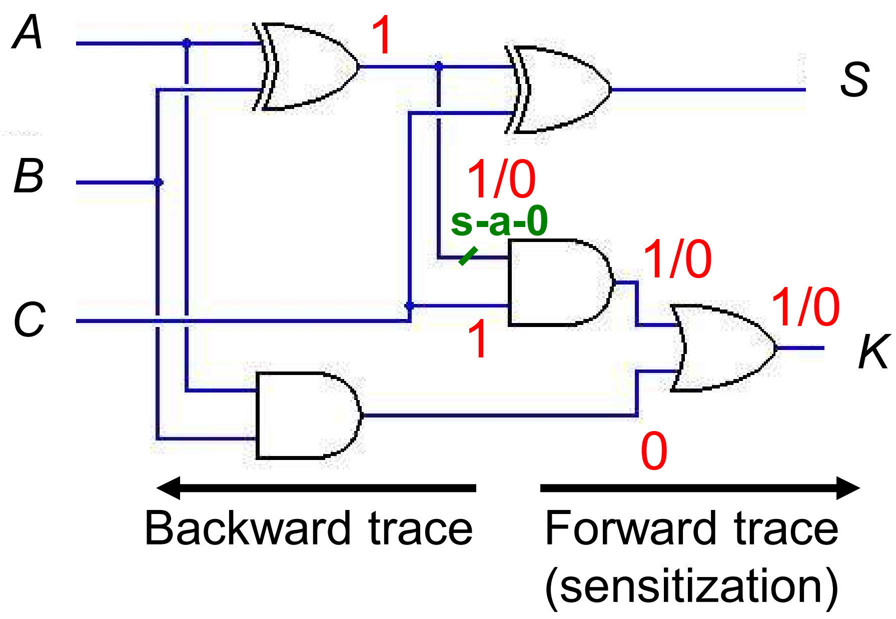
9.2 Path Sensitization and D-Algorithm

The main idea behind test design: control the faulty point from inputs and propagate its behavior to some output

Example: s-a-0 fault
Test must force the line to 1

Two possible tests
(*A*, *B*, *C*) = (0 1 1) or (1 0 1)

This method is formalized in the *D*-algorithm

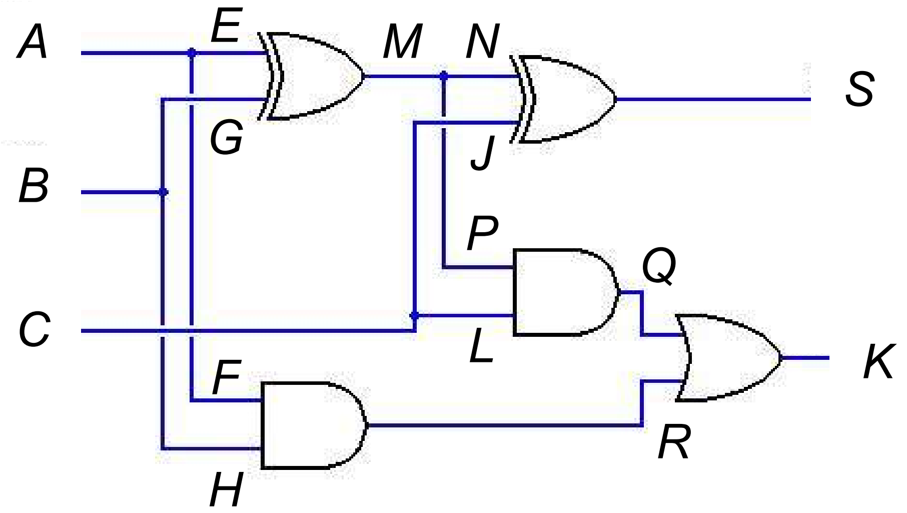


***D*-calculus**
1/0 on the diagram above is represented as *D*
0/1 is represented as $\overline{D}$
Encounters difficulties with XOR gates (PODEM algorithm fixes this)

Selection of a Minimal Test Set

Each input pattern detects a subset of all possible faults of interest (according to our fault model)

A	*B*	*C*	*P*		*Q*	
			s-a-0	s-a-1	s-a-0	s-a-1
0	0	0	-	-	-	x
0	0	1	-	x	-	x
0	1	1	x	-	x	-
1	0	1	x	-	x	-



Choosing a minimal test set is a covering problem

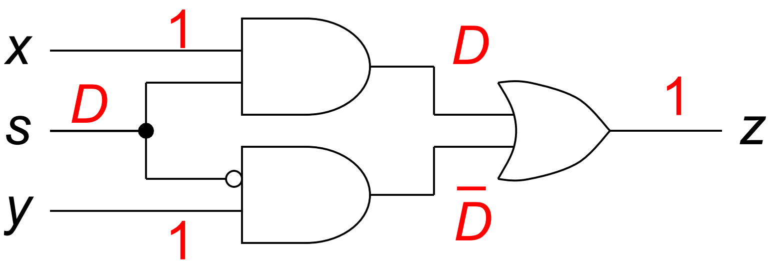
Equivalent faults: e.g., *P* s-a-0 ≡ *L* s-a-0 ≡ *Q* s-a-0
Q s-a-1 ≡ *R* s-a-1 ≡ *K* s-a-1

Capabilities and Complexity of D-Algorithm

Reconvergent fan-out

Consider the *s* input s-a-0

Simple path sensitization does not allow us to propagate the fault to the primary output *z*



PODEM solves the problem by setting *y* to 0

Worst-case complexity of D-algorithm is exponential in circuit size

Must consider all path combinations

XOR gates cause the behavior to approach the worst case

Average case is much better; quadratic

PODEM: <u>P</u>ath-<u>o</u>riented <u>de</u>cision <u>m</u>aking

Developed by Goel in 1981

Also exponential, but in the number of circuit inputs, not its size

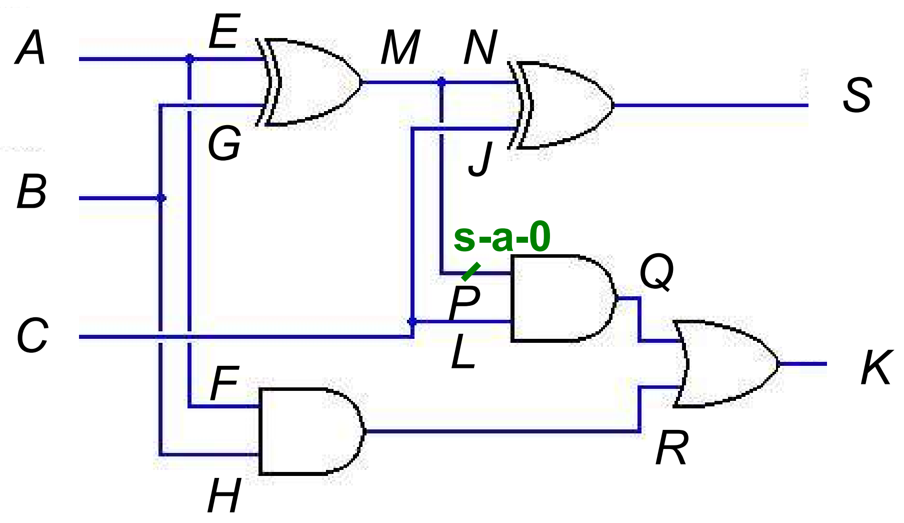
9.3 Boolean Difference Methods

$K = f(A, B, C) = AB \vee BC \vee CA$

$$\begin{aligned} dK/dB &= f(A, 0, C) \oplus f(A, 1, C) \\ &= CA \oplus (A \vee C) \\ &= A \oplus C \end{aligned}$$

$K = PC \vee AB$

$dK/dP = AB \oplus (C \vee AB) = C(\overline{AB})$



Tests that detect P s-a-0 are solutions to the equation $P\, dK/dP = 1$

$(A \oplus B)\, C(\overline{AB}) = 1 \quad \Rightarrow \quad C = 1, A \neq B$

Tests that detect P s-a-1 are solutions to the equation $\bar{P}\, dK/dP = 1$

$(\overline{A \oplus B})\, C(\overline{AB}) = 1 \quad \Rightarrow \quad C = 1, A = B = 0$

9.4 The Complexity of Fault Testing

The satisfiability problem (SAT)
Decision problem: Is a Boolean expression satisfiable?
(i.e., can we assign values to the variables to make the result 1?)

Theorem (Cook, 1971): SAT is NP-complete
In fact, even restricted versions of SAT remain NP-complete

Theorem (Cook, 1971): 3SAT is NP-complete
In 3SAT, the logic expression is a product of 3-term OR clauses

According to the Boolean difference formulation, fault detection can be converted to SAT (find the solutions to $P\ dK/dP = 1$)

To prove the NP-completeness of fault detection, we need to show that SAT (or another NP-complete problem) can be converted to it

Proof of NP-completeness is due to Ibarra and Sahni [Ibar75]
A simple alternate proof by Fujiwara [Fuji82] is in the textbook

Proof that Fault Detection is NP-Complete

Theorem (Cook, 1971): 3SAT is NP-complete

Theorem: Clause-monotone SAT (CM-SAT) is NP-complete

CM-SAT has OR clauses each of which consists entirely of complemented or uncomplemented variables, but not both

3SAT can be converted to CM-SAT by replacing each mixed OR clause with the product of two clauses involving a new variable
Example: $(x_i \vee x_j \vee x'_k)$ is replaced by $(x_i \vee x_j \vee v_k)(v'_k \vee x'_k)$

Clause-monotone SAT can be converted to fault detection in a circuit

First level has ANDs for all clauses with complemented variables

Second level has ORs for all clauses with uncomplemented variables, plus an OR gate with level-1 outputs as its inputs (one input to this gate is *y*)

Third level has one AND gate that receives all level-2 outputs as its inputs
A test for y s-a-1 satisfies the original clause-monotone expression

9.5 Testing of Units with Memory

The presence of memory expands the number of required test cases

To test a sequential machine, we may need to apply different input sequences for each possible initial state

Exponentially many possible input sequences

Exponentially many possible machine states

Testing of Memory

Simple-minded approach: Write 000 . . . 00 and 111 . . . 11 into every memory word and read out to verify proper storage and retrieval

Problems with the simple-minded approach:

- Does not test access/decoding mechanism – How do you know the intended word was written into and read from?
- Many memory faults are pattern-sensitive, where cell operation is affected by the values stored in nearby cells
- Modern high-density memories experience dynamic faults that are exposed only for specific access sequences

Memory testing continues to be an active research area

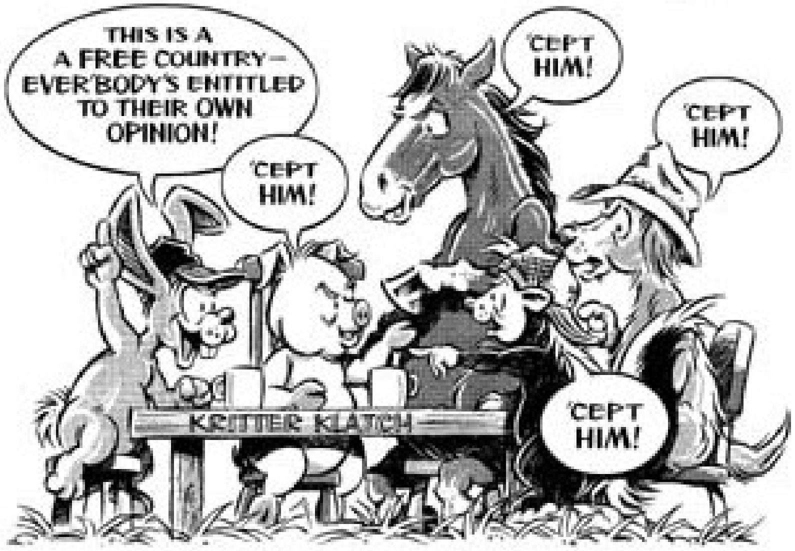
Built-in self test is the only viable approach in the long term

Challenge: Any run time testing consumes some memory bandwidth

9.6 Off-Line vs. Concurrent Testing

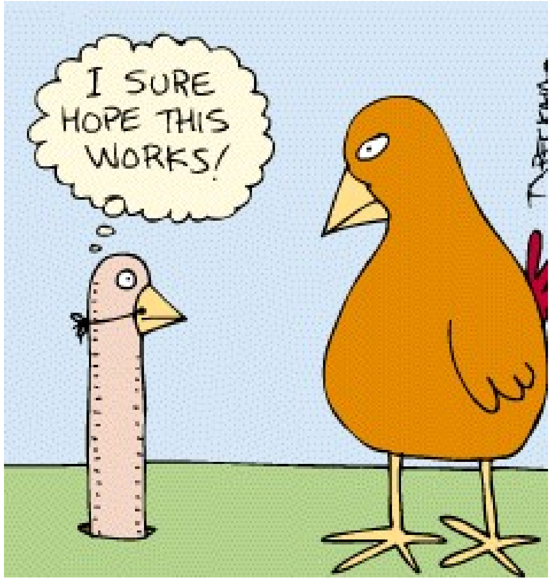
This section will be forthcoming.

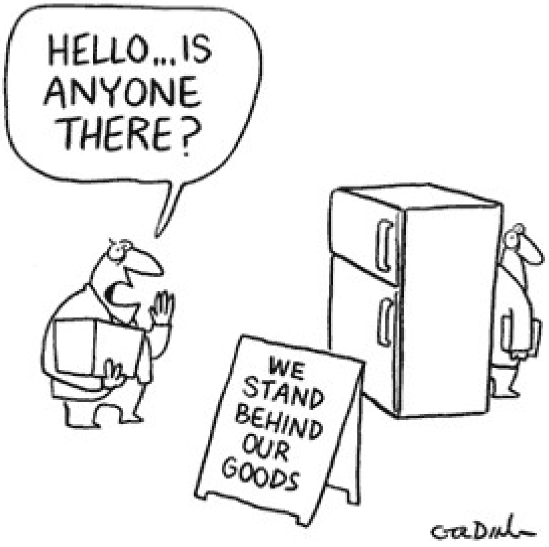
10 Fault Masking

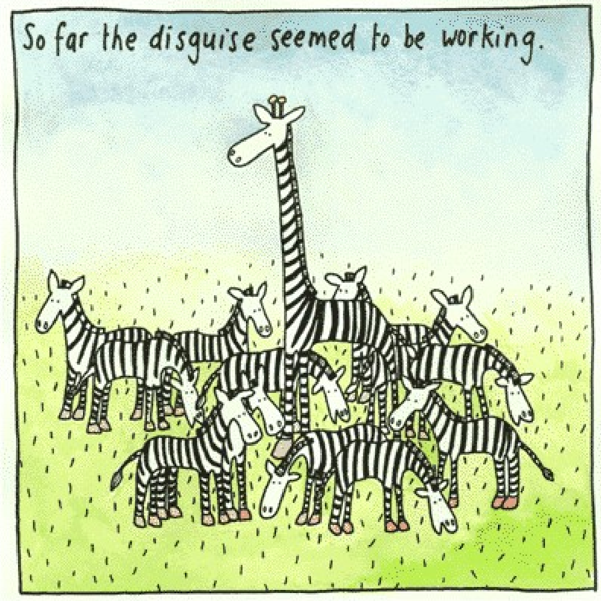

THIS IS A A FREE COUNTRY— EVER'BODY'S ENTITLED TO THEIR OWN OPINION!
'CEPT HIM!
'CEPT HIM!
'CEPT HIM!
'CEPT HIM!
KRITTER KLATCH

ZERO TOLERANCE ON INTOLERANCE...

off the mark by Mark Parisi
www.offthemark.com
I THINK I'VE ISOLATED THE REASON YOUR COMPUTER REFUSES TO RECOGNIZE THE PRINTER...



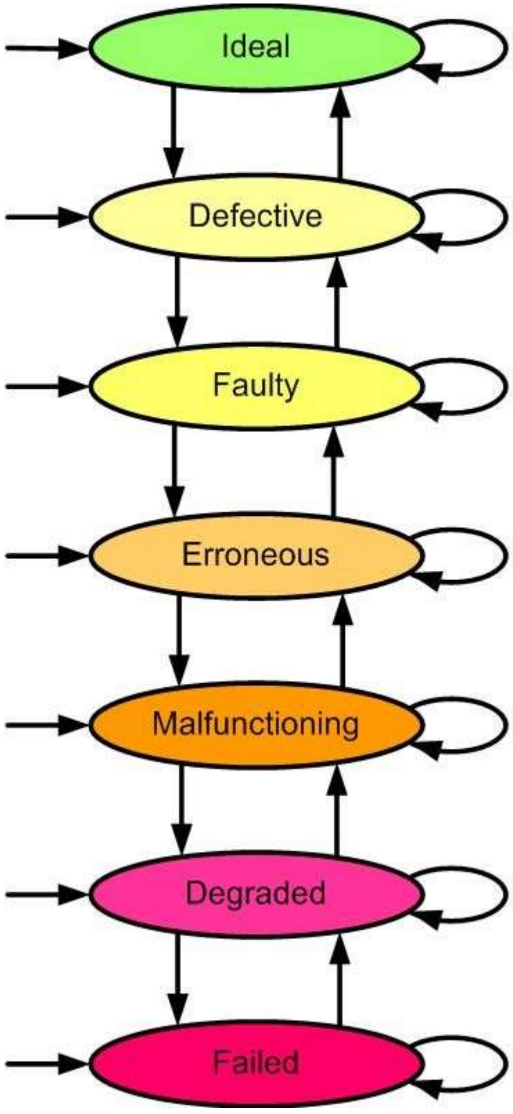




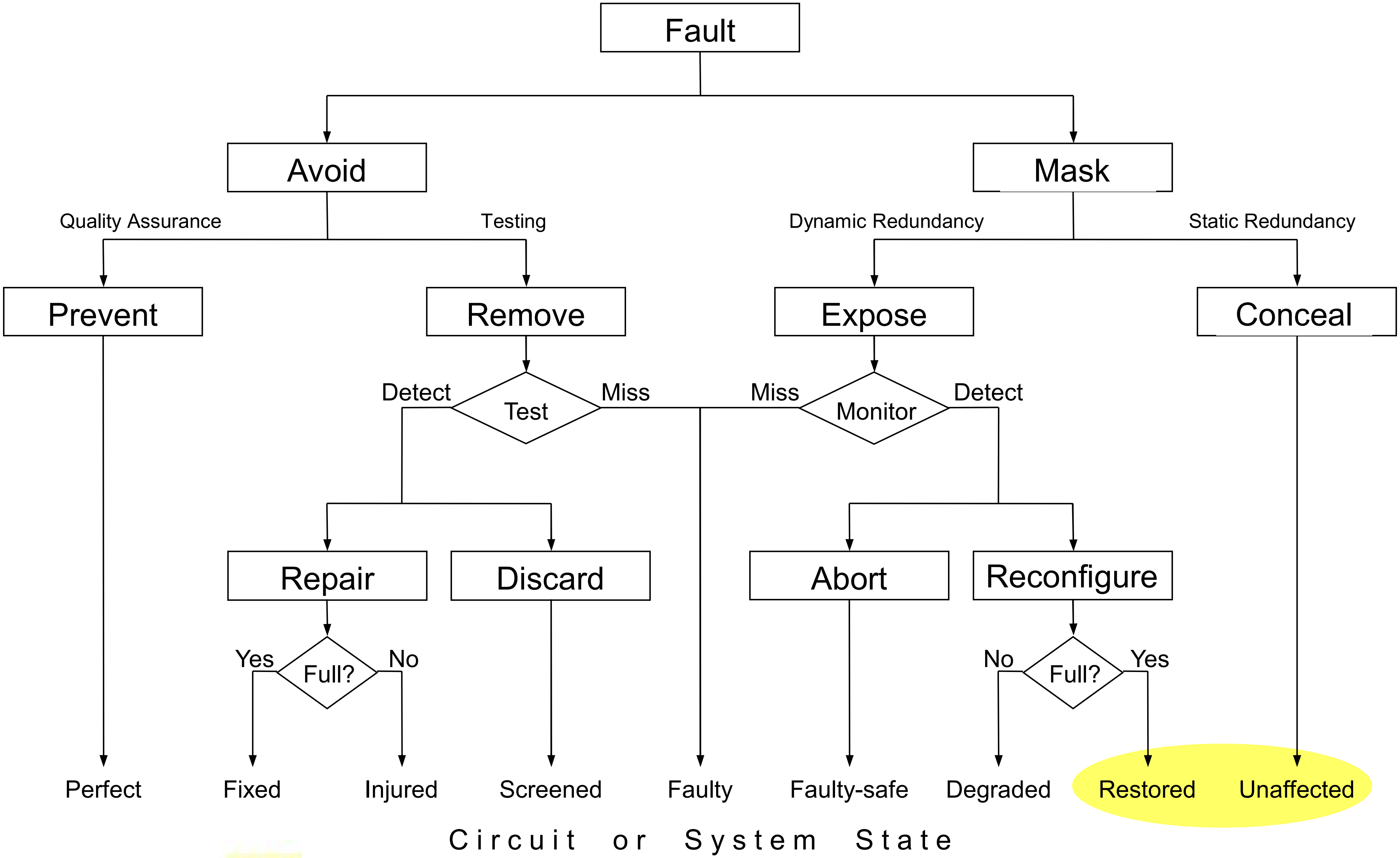
STRUCTURE AT A GLANCE

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

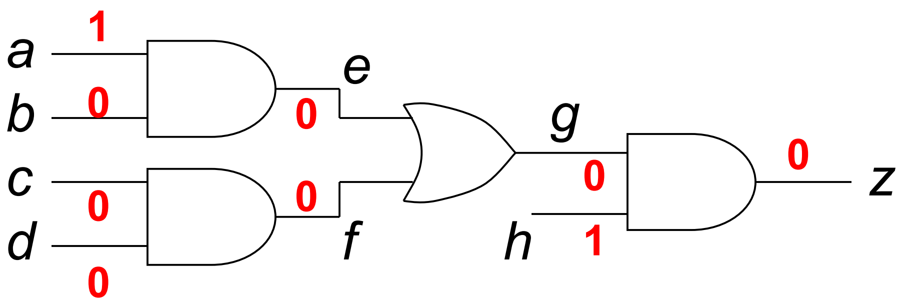
Appendix: Past, Present, and Future



10.1 Fault Avoidance vs. Masking



10.2 Interwoven Redundant Logic



0 → 1 fault in *b* is critical

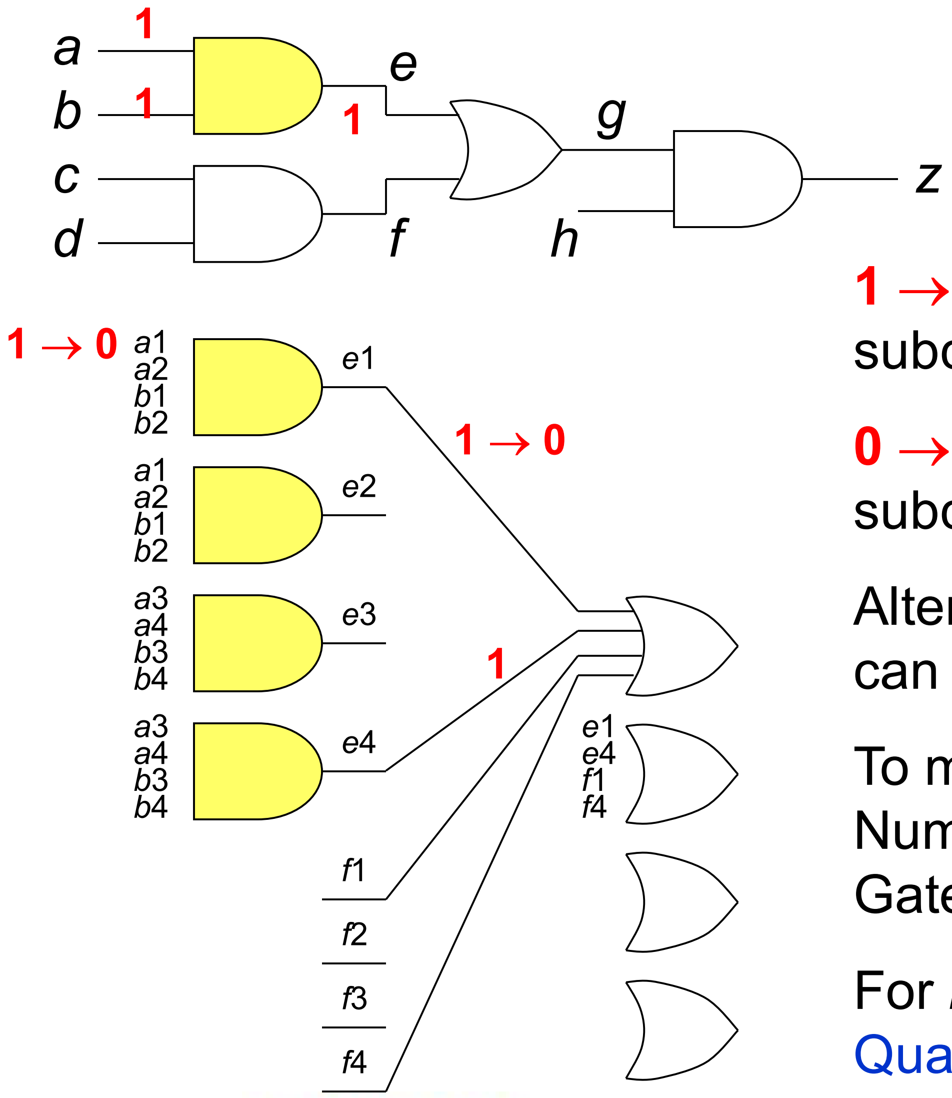
0 → 1 fault in *c* or *d* is not critical (it is masked)

1 → 0 fault in *a* or *h* is not critical (it is masked)

Even nonredundant circuits have some masking capability

Is there a way to exploit the inherent masking capabilities of logic gates to achieve general fault masking?

How Interwoven Logic Works



Let *x*1, *x*2, *x*3, and *x*4 be 4 copies of the signal *x*

1 → 0 change is critical for AND, subcritical for OR

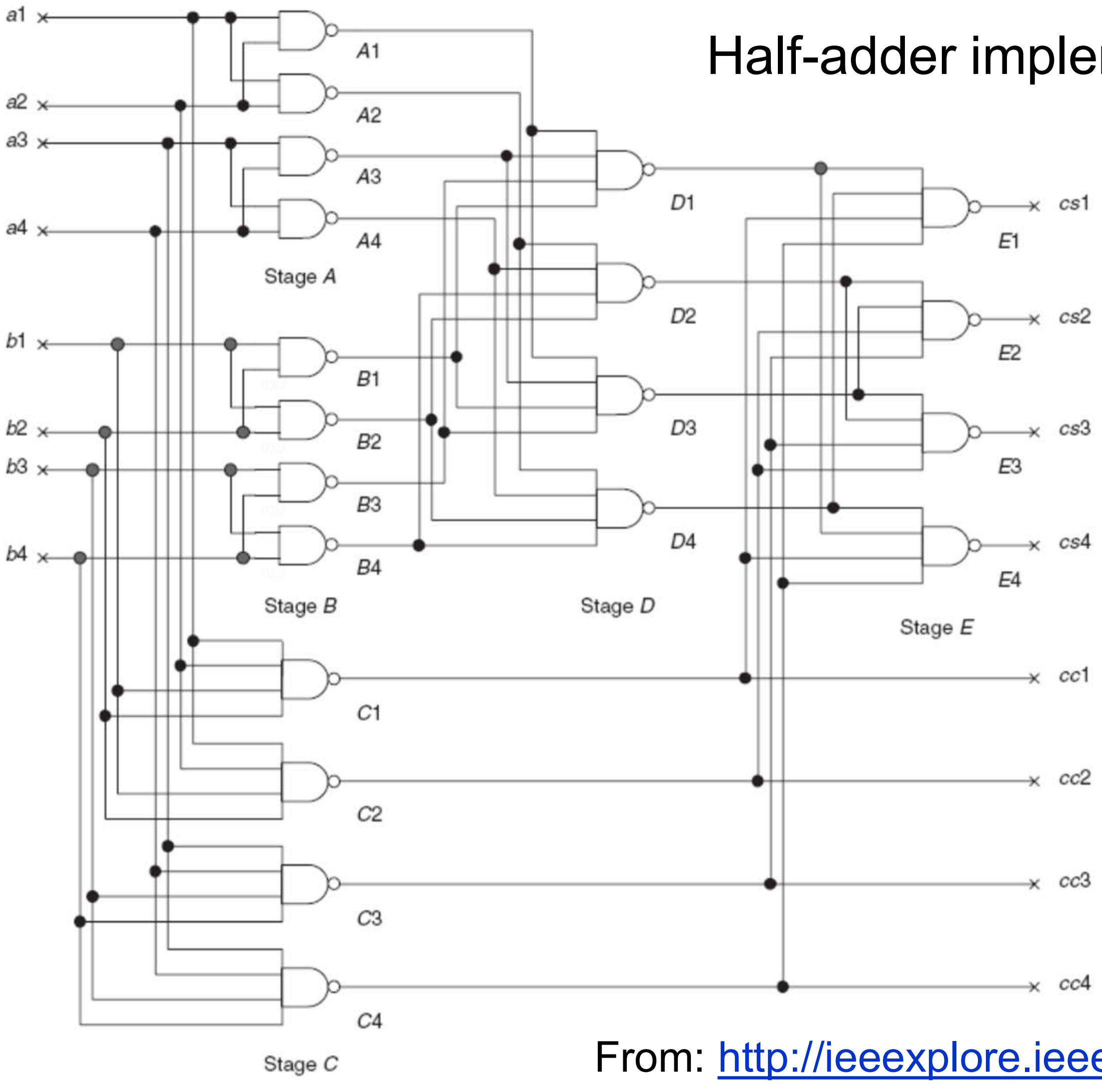
0 → 1 change is critical for OR, subcritical for AND

Alternating layers of ANDs and ORs can mask each other's critical faults

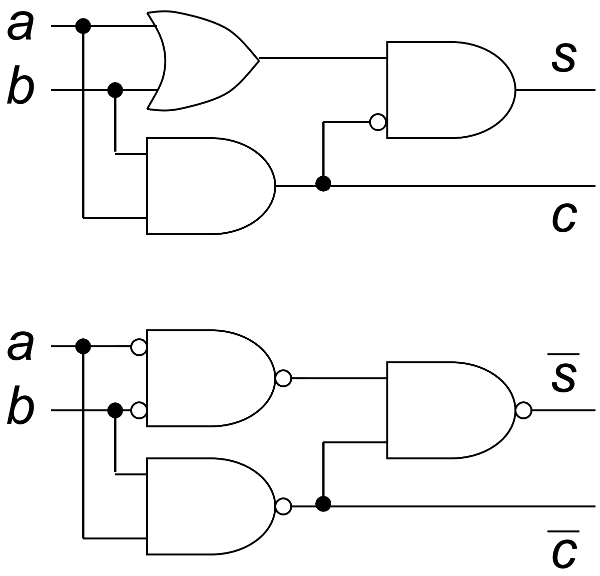
To mask *h* critical faults:
Number of gates multiplied by $(h + 1)^2$
Gate inputs multiplied by $h + 1$

For *h* = 1, the scheme is known as Quadded logic

Interwoven Logic for Nanoelectronics



Half-adder implemented in quadded logic



IEEE D&T
July-Aug. 2005
pp. 328-339

From: http://ieeexplore.ieee.org/iel5/54/32070/01492293.pdf

Highly Reliable Logic with “Crummy” Relays

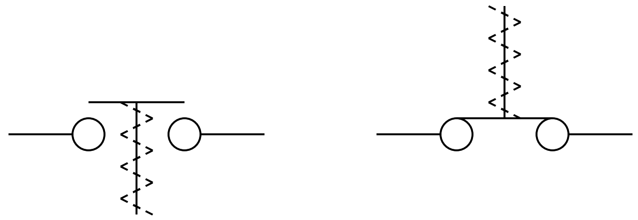
Moore & Shannon, 1956

a: prob [contact made | energized]
1 – *a*: prob [contact open | energized]
c: prob [contact made | not energized]
1 – *c*: prob [contact open | not energized]

No matter how crummy the relays (i.e., how close the values of *a* and *c*), one can interconnect many of them in a redundant series-parallel structure to achieve arbitrarily high reliability

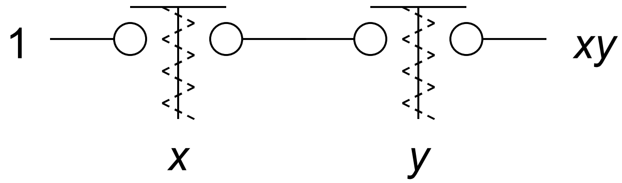
prob [connection made | energized] =
$2a^2 - a^4$ (> *a* if *a* > 0.62)

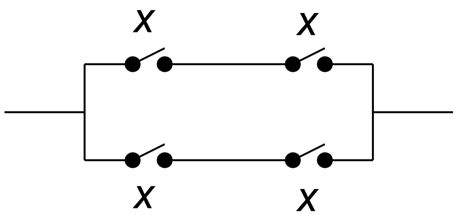
prob [connection made | not energized] =
$2c^2 - c^4$ (always < *c*)



“Make” contact (normally open)
$a > c$

“Break” contact (normally closed)
$a < c$

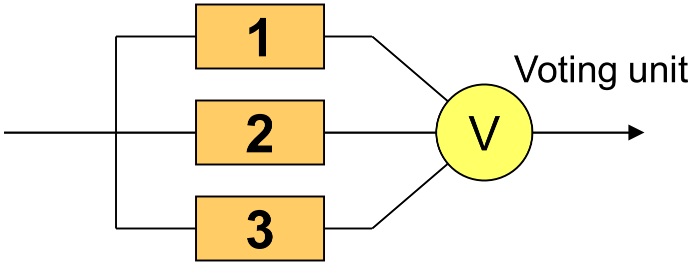




$a > 0.5$, $c < 0.5$

10.3 Static Redundancy with Replication

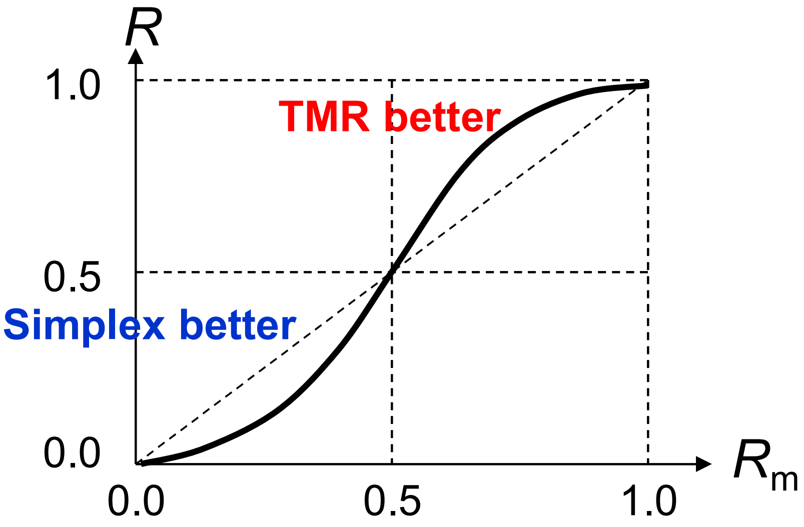
TMR: $R = 3R_m^2 - 2R_m^3 \overset{?}{>} R_m$

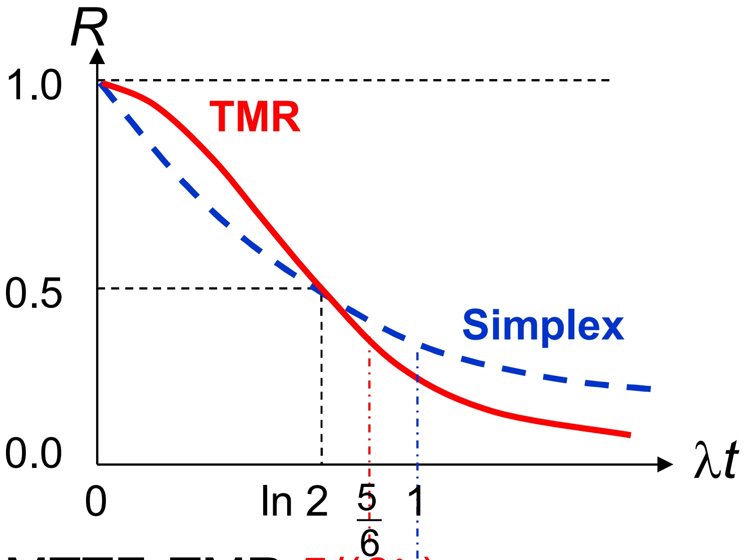


Condition on the module reliability:

$R = R_m [1 + (1 - R_m)(2R_m - 1)]$

$(1 - R_m)(2R_m - 1) > 0 \quad \Rightarrow \quad R_m > 1/2$





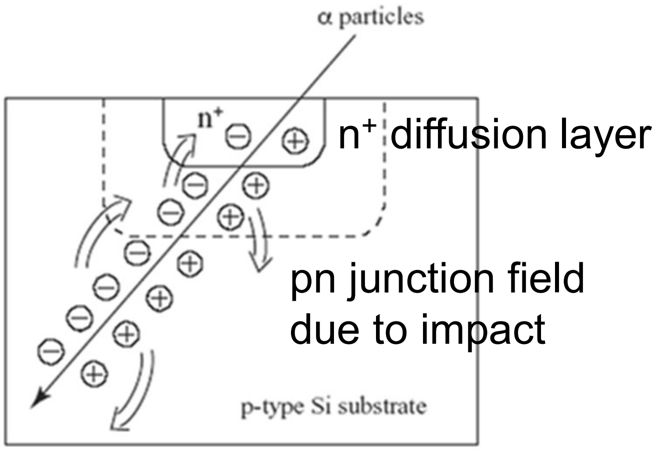
$\text{RIF}_{\text{TMR/Simplex}} = (1 - R_m)/(1 - R)$
$= 1/[1 - R_m(2R_m - 1)]$

MTTF: TMR $5/(6\lambda)$

Simplex $1/\lambda$

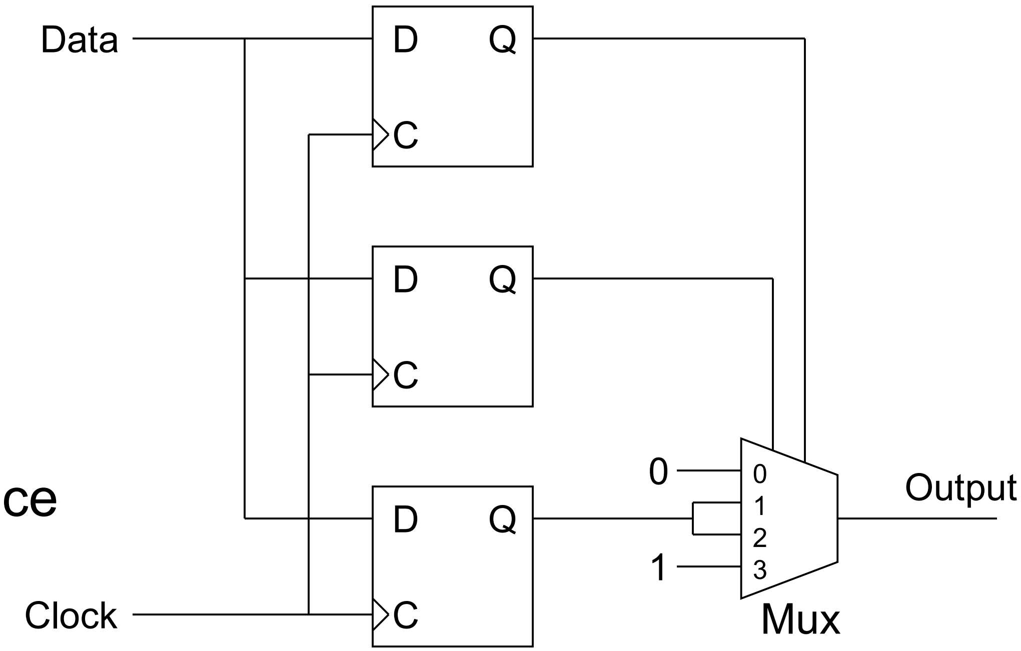
A TMR Application and Its Bit-Voting Unit

Single-event upset (SEU) = Soft error
Change of state caused by a high-energy particle strike

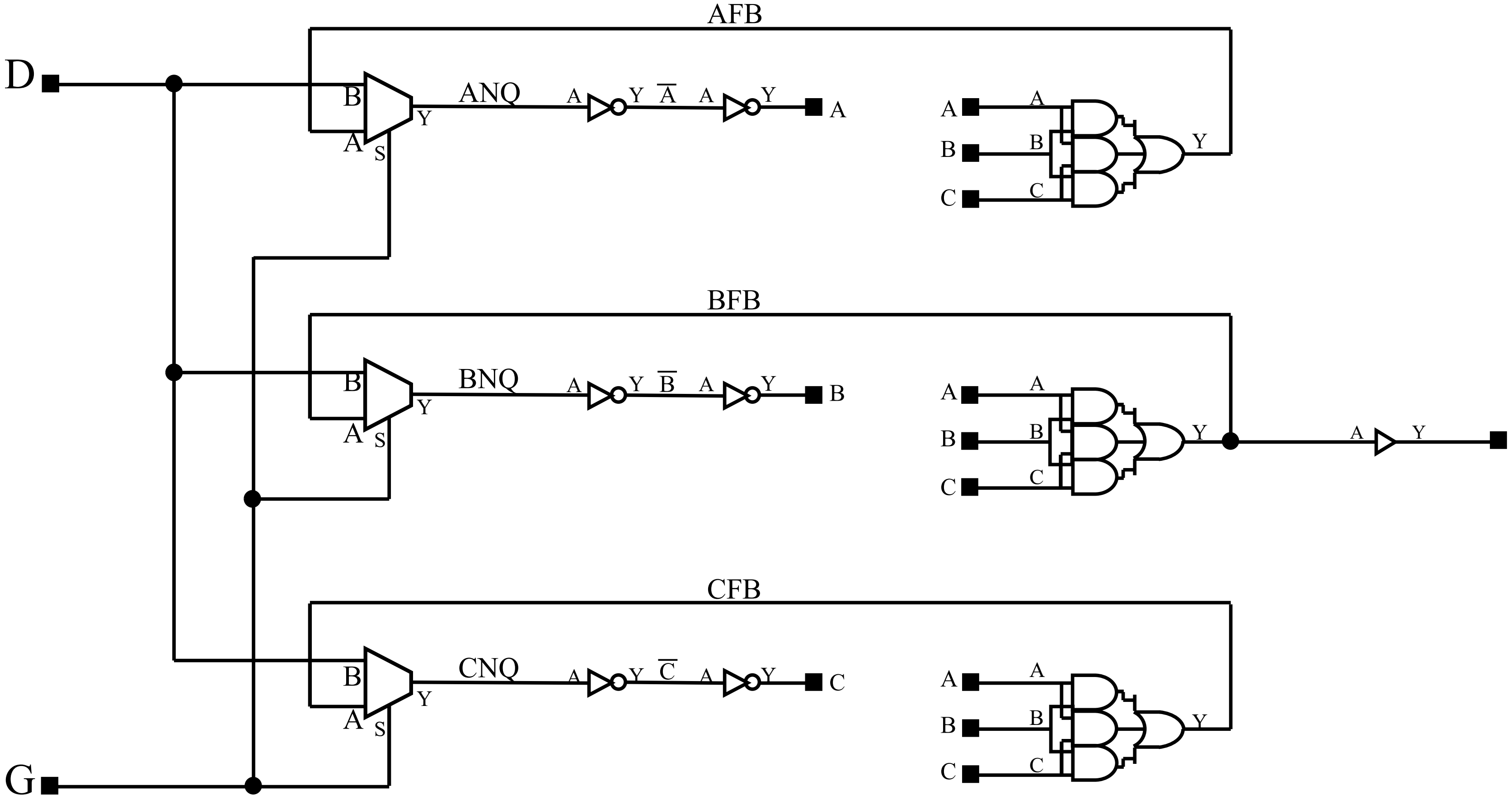


SEU effect on DRAMs
(from SANYO website)

TMR flip-flop for SEU tolerance



Example: SEU Hardened Flip-Flop

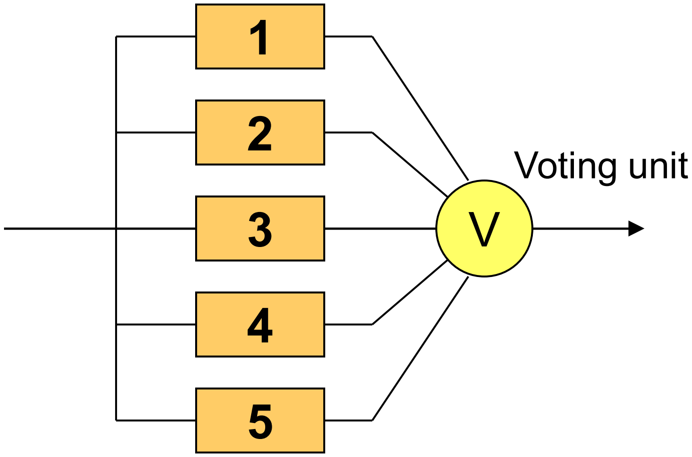


For list of flip-flop hardening methods and their comparison, see:
http://klabs.org/richcontent/fpga_content/pages/notes/seu_hardening.htm

N-Modular Redundancy (NMR)

Triple-modular redundancy (TMR) can be generalized to *N* units

N-modular redundancy (NMR) uses *N* modules along with a voter, with *N* usually being odd



Example: 5MR
Operates correctly as long as
3 of the 5 modules are healthy

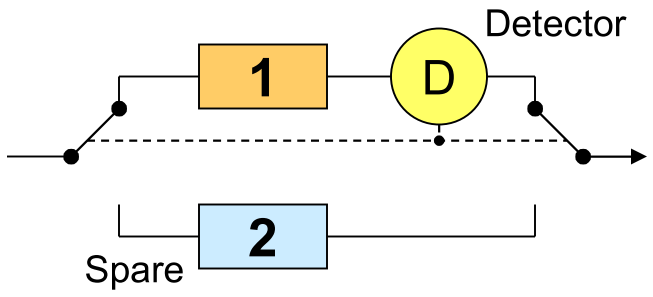
Voter complexity rises rapidly
with increasing *N*

Even values of *N* are also feasible

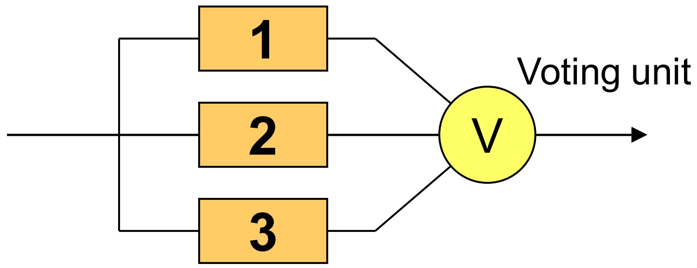
Example: 4MR, with 3-out-of-4 voting
Voter masks single faults; can be designed to detect double faults

10.4 Dynamic and Hybrid Redundancy

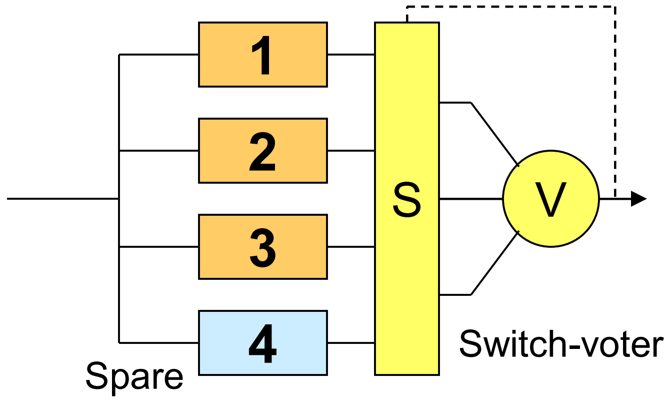
1. **Detect and replace**
   Dynamic redundancy (cold/hot standby)
   Detection via
   -- coding, watchdog timer, self-checking
   -- duplication (pair-and-spares)



2. **Mask in place**
   Static redundancy
   May revert to simplex instead of duplex
   Design challenges include
   -- synchronization for voting
   -- voting on imprecise results

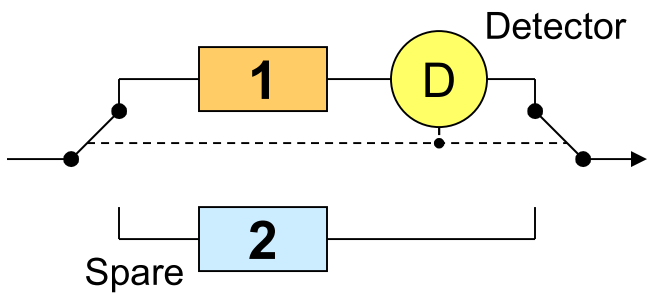


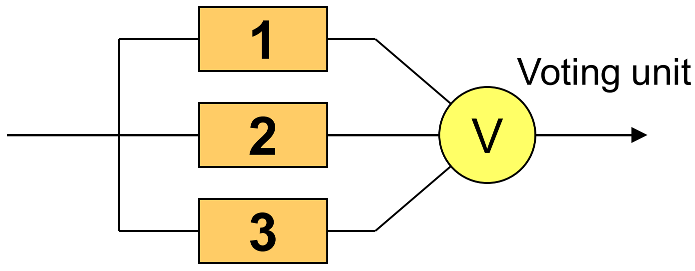
3. **Mask, diagnose, and reconfigure**
   Hybrid redundancy
   Fault masked at output, but diagnosed
   -- e.g., via comparison with voter output
   Faulty circuit is replaced by spare
   Becomes static upon spare exhaustion

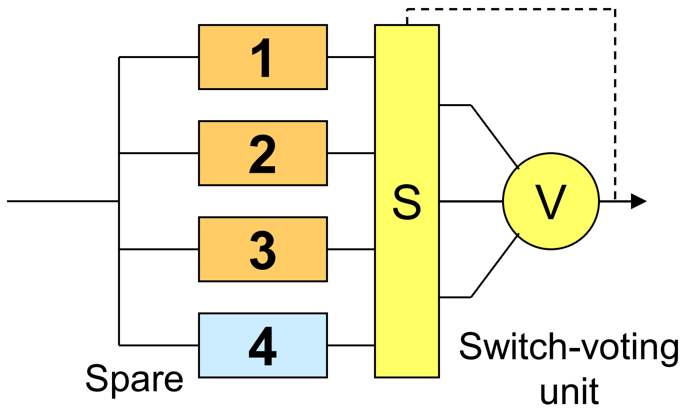


Comparing Replication Schemes

Advantages	**Drawbacks**
Less power (cold standby)	Coverage factor
Long life (just add spares)	Tolerance latency
Immediate masking	Power/area penalty
High safety	Voting critical
Immediate masking	Power/area penalty
Long life and high safety	Switch-voting critical





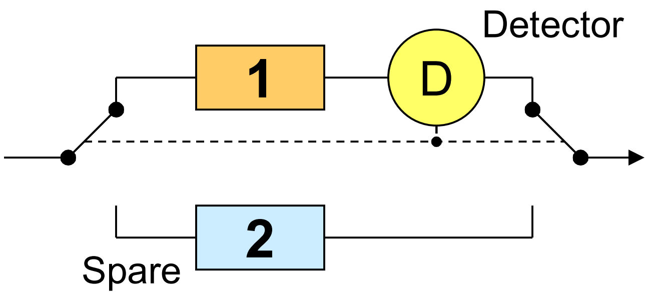


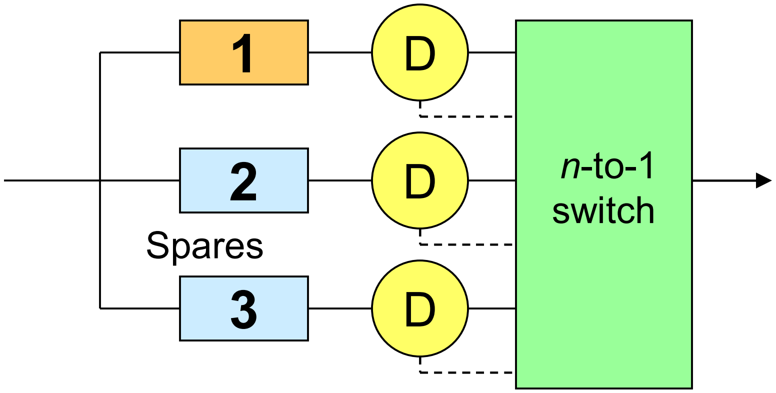
Switch for Standby Redundancy

Standby redundancy requires an *n*-to-1 switch to select the output of the currently active module

The detectors use various info to deduce fault conditions
-- Error coding
-- Reasonableness checks
-- Watchdog timer

Once a fault has been detected, the switch reconfigures the system by flagging the faulty unit and activating next spare in sequence





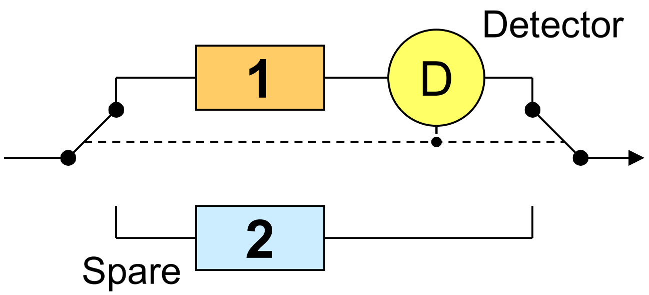
If we use an *n*-to-2 switch and compare the two selected outputs, the configuration is known as “pair-and-spares”

Fault Detection in Standby Redundancy

Activity monitoring

Duplication and comparison

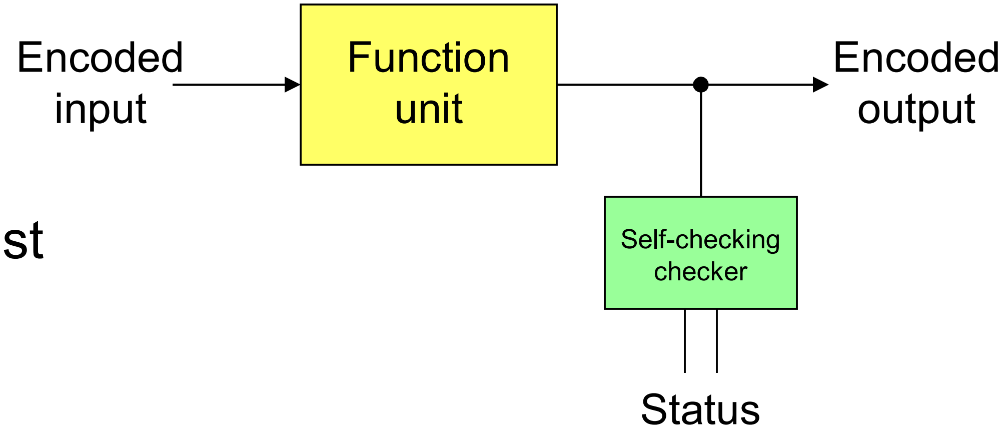
Self-checking design

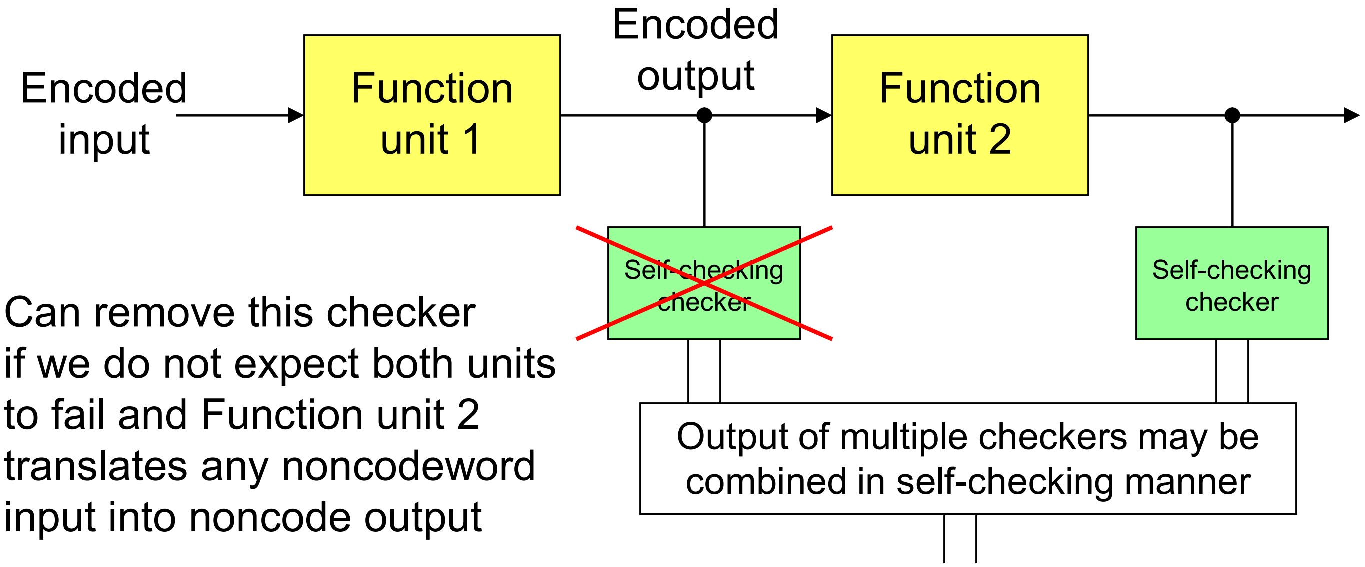


Preview of Self-Checking Design

Covered in Chapter 15

Function unit designed
such that internal faults manifest
themselves as invalid outputs





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

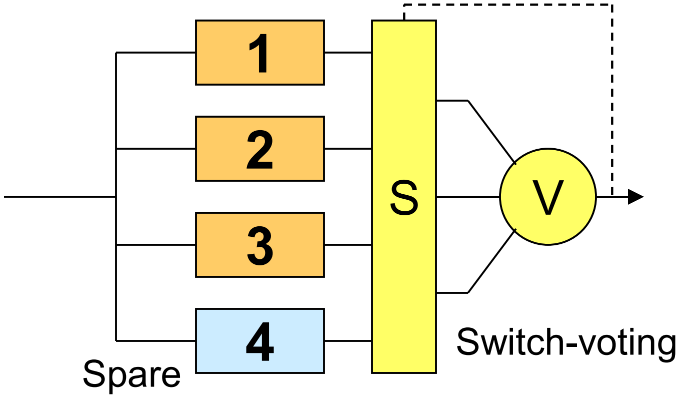
Output of multiple checkers may be
combined in self-checking manner

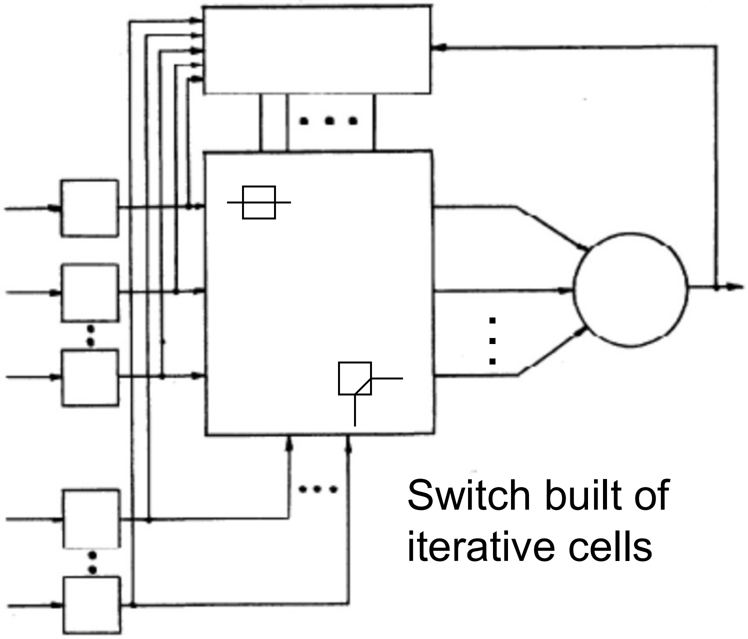
Switch for Hybrid Redundancy

Hybrid redundancy with *n* active and *s* spare modules requires an (*n* + *s*)-to-*n* switch to select the outputs of the active modules

Self-purging redundancy is a variant of hybrid redundancy in which all modules are active at the outset, but they are purged as they disagree with the majority output

Voting unit in self-purging redundancy is a threshold voter that considers the inputs with weights of 1 (active) or 0 (purged)





Switch built of iterative cells

10.5 Time Redundancy

Retry upon a detected fault: particularly useful for transient faults

Recomputation not useful with permanent faults

Can make recomputation work by slightly changing the operands, but this is not always applicable

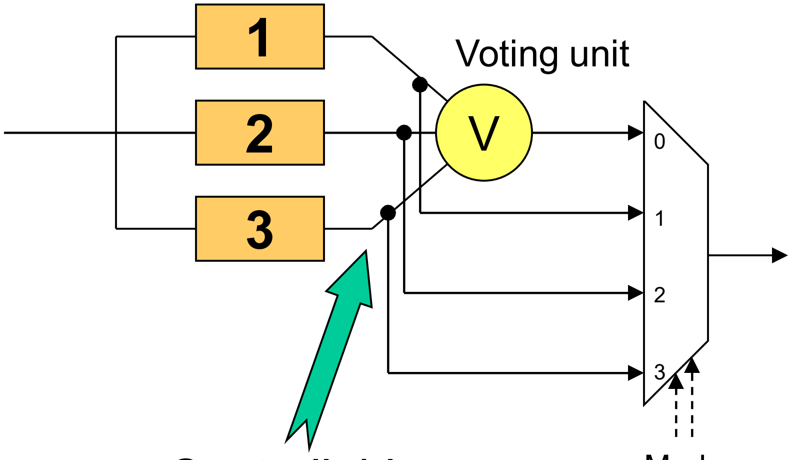
Compute $a \times (2b)$ instead of $(2a) \times b$

Compute $b + a$ or $-(-a - b)$ instead of $a + b$

10.6 Variations and Complications

Static redundancy makes fault testing more challenging

For static redundancy to be effective, we must ensure that initially all redundant components are fault-free



Controllable, but not observable

Applications of NMR and Hybrid Redundancy

NASA's Space Shuttle (retired in 2012):

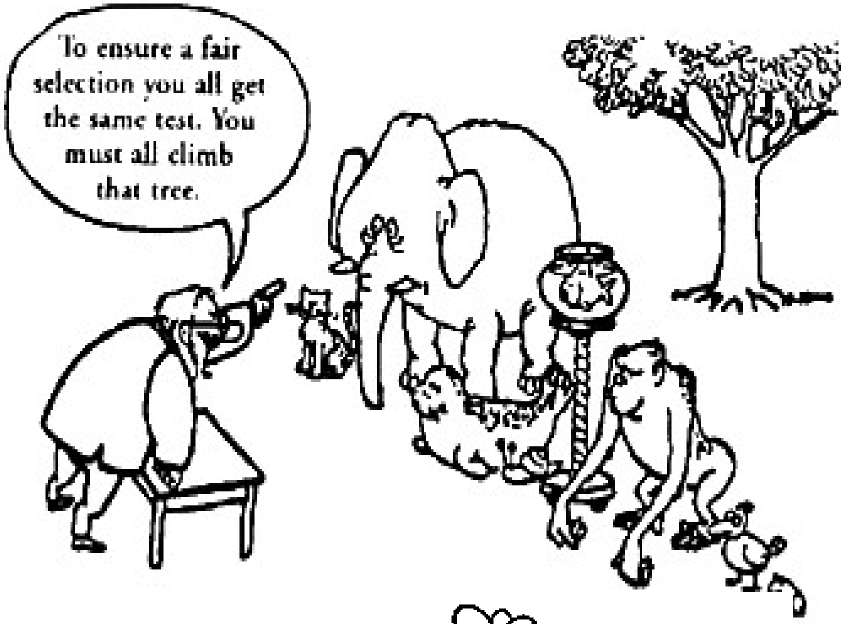
Used 5-way redundancy in hardware
Originally, 3 operational units + 2 spares (one warm, one cold)
More recently, 4 operational + 1 spare

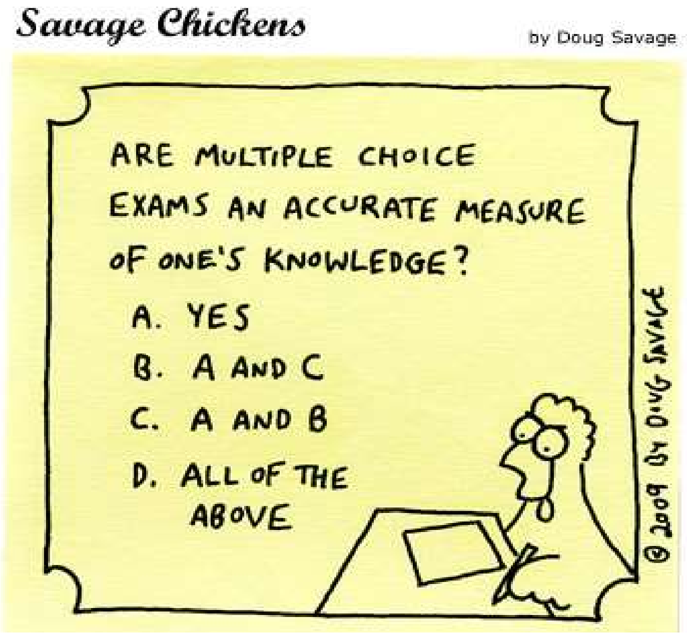
Also, uses 2 independently developed software systems (Design diversity)

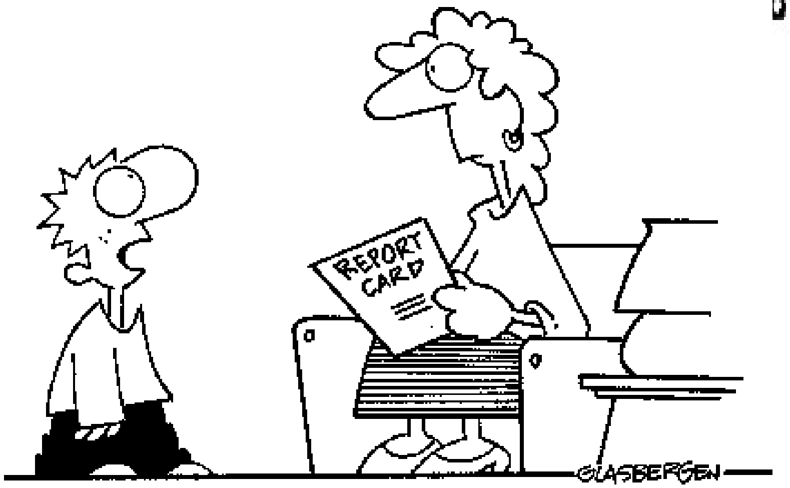
Japanese Shinkansen "Bullet" Train

Triple-duplex system (6-fold redundancy)

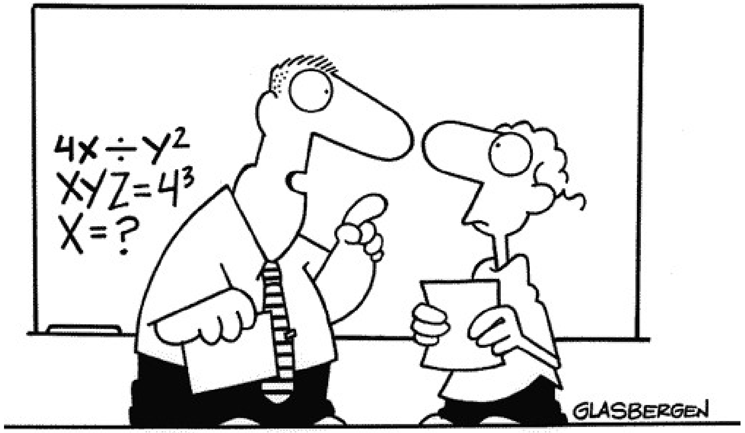
11 Design for Testability


To ensure a fair selection you all get the same test. You must all climb that tree.

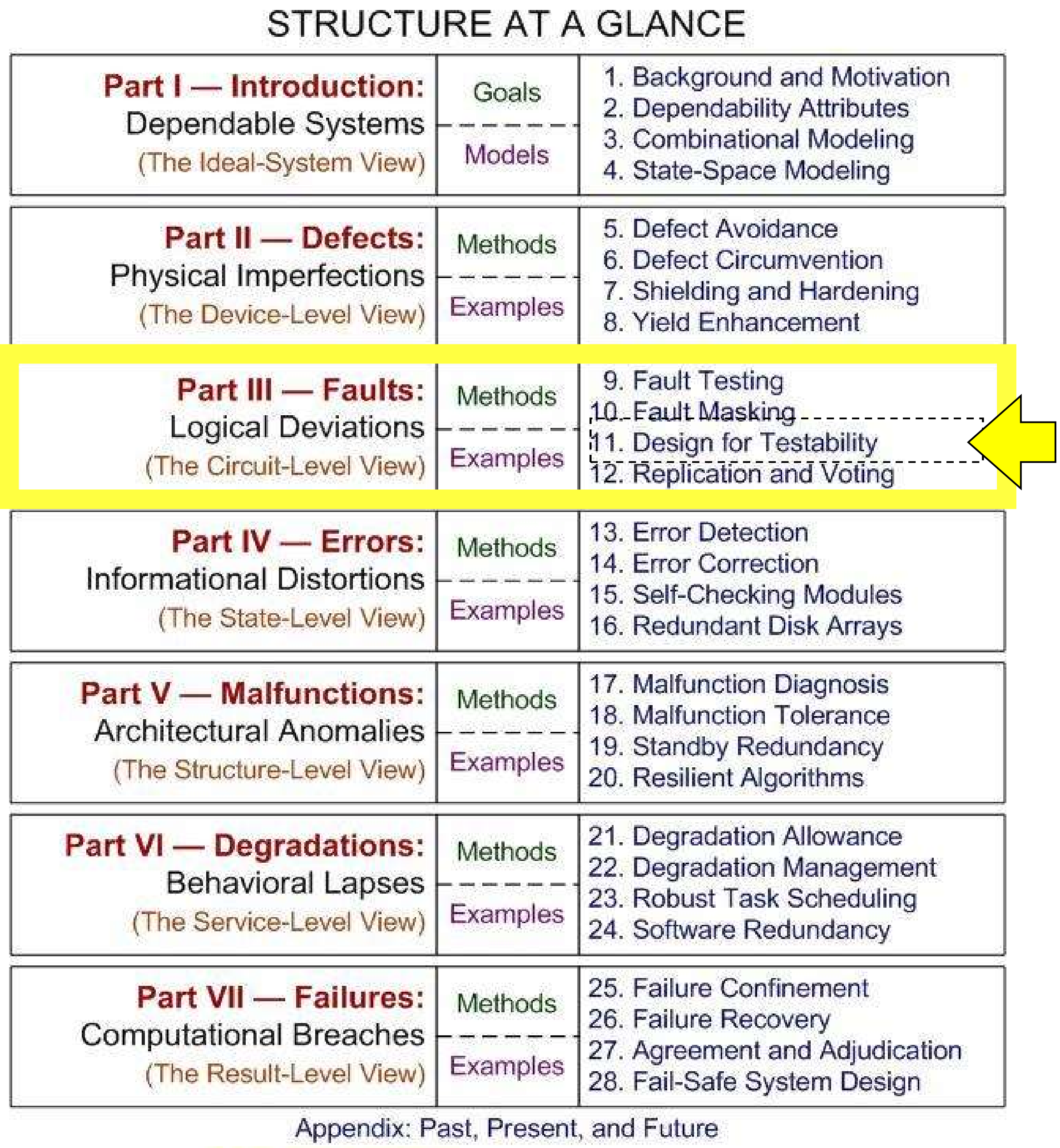




"Someone in this house flunked his earth science test because someone else in this house told him that love makes the world go around!"



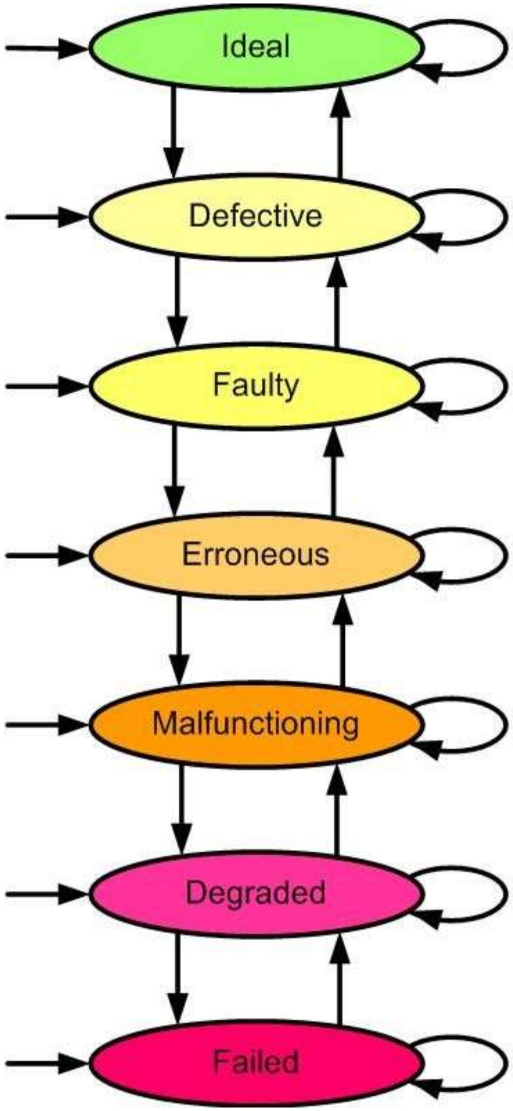
"Algebra class will be important to you later in life because there's going to be a test six weeks from now."



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



11.1 The Importance of Testability

A small circuit with a limited number of inputs and outputs can be tested with a reasonable amount of effort and time

A complex unit, such as a microprocessor, cannot be tested solely based on its input/output behavior

Hence, the need for provisions in the design to facilitate testing

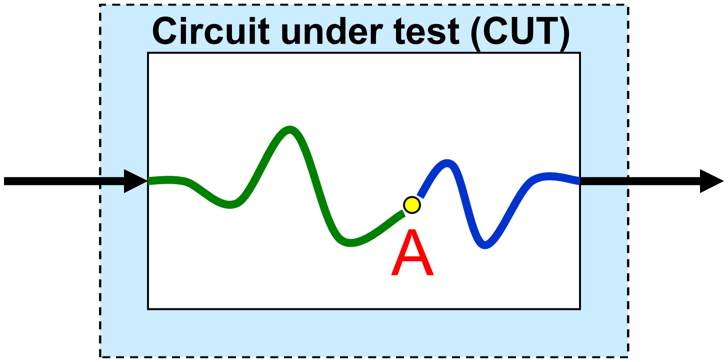
11.2 Testability Modeling

To allow detection of a fault in point A of a logic circuit, we need to:

Be able to control that point from the primary inputs

Be able to observe that point from the primary outputs

Thus, good **testability** requires
good **controllability** and
good **observability**
for every node in the circuit



Quantifying Controllability

Controllability C of a line has a value between 0 and 1

Derive C values by proceeding from inputs (C = 1) to outputs

Controllability transfer factor

$$CTF = 1 - \left| \frac{N(0) - N(1)}{N(0) + N(1)} \right|$$

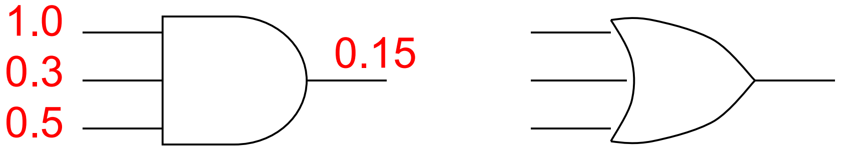
$$C_{\text{output}} = (\Sigma_i C_{\text{input } i} / k) \times CTF$$

f-way fan-out

A line with very low controllability is a good test point candidate

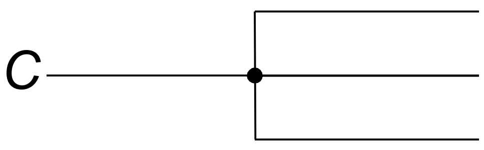
$N(0)$: # input patterns leading to 0 output
$N(1)$: # input patterns leading to 1 output

k-input, 1-output components

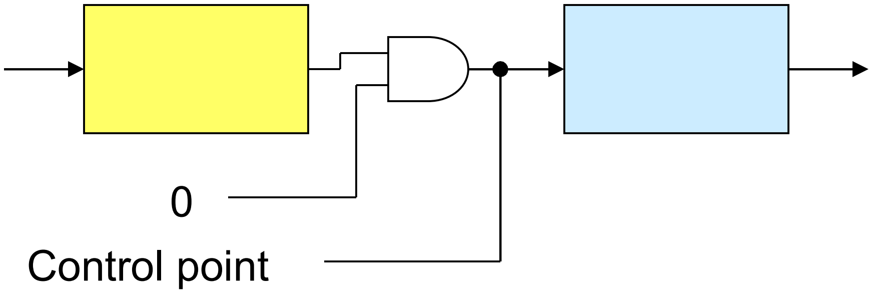


$N(0) = 7$
$N(1) = 1$
$CTF = 0.25$

$N(0) = 1$
$N(1) = 7$
$CTF = 0.25$



$C / (1 + \log_2 f)$ for each of f fan-out lines



Quantifying Observability

Observability *O* of a line has a value between 0 and 1

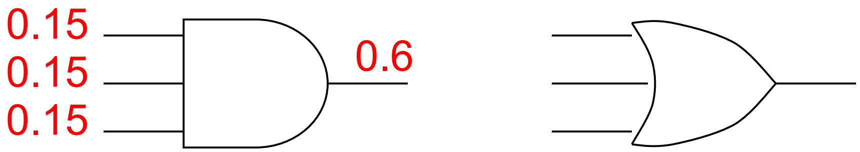
Derive *O* values by proceeding from outputs (*O* = 1) to inputs

Observability transfer factor

$$OTF = \frac{N(sp)}{N(sp) + N(ip)}$$

$$O_{\text{input } i} = O_{\text{output}} \times OTF$$

k-input, 1-output components

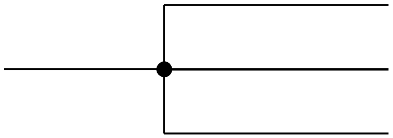


$N(sp) = 1$
$N(ip) = 3$
$OTF = 0.25$

$N(sp) = 1$
$N(ip) = 3$
$OTF = 0.25$

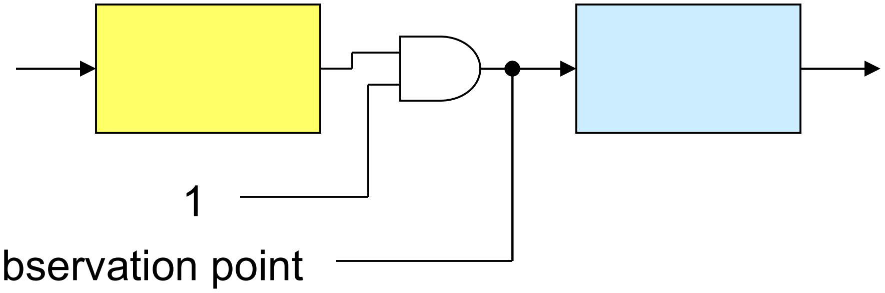
f-way fan-out

$1 - \Pi_j(1 - O_j)$



O_j for line *j*

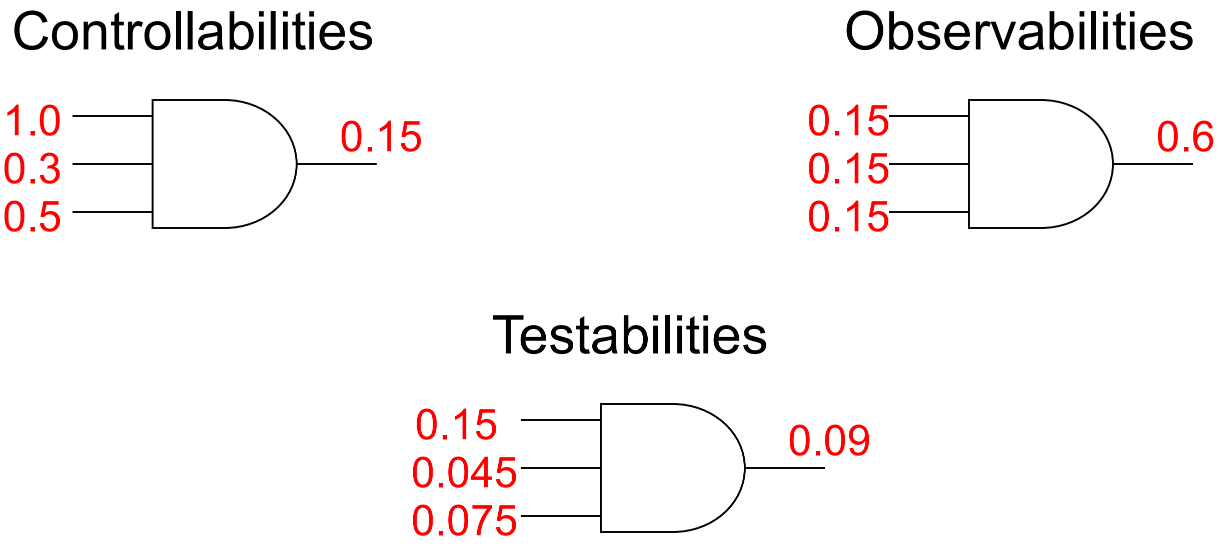
A line with very low observability is a good test point candidate



N(*sp*): # ways of sensitizing a path to output
N(*ip*): # ways of inhibiting a path to output

Quantifying Testability

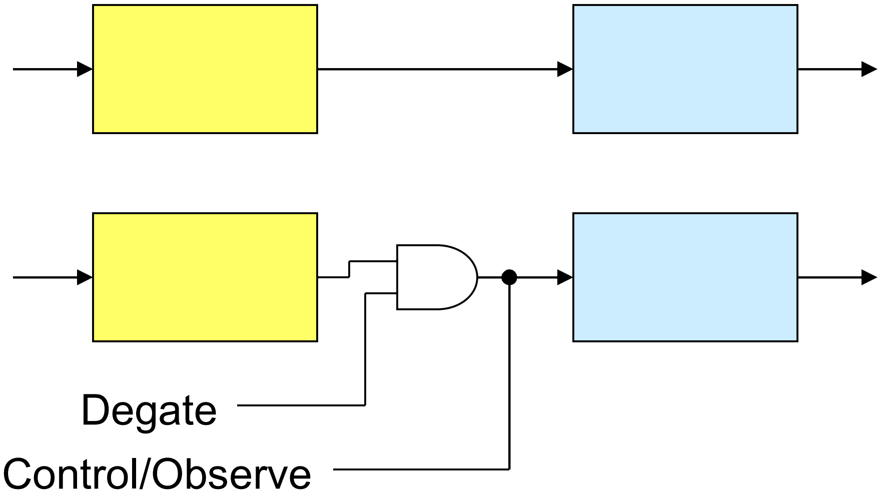
Testability = Controllability × Observability



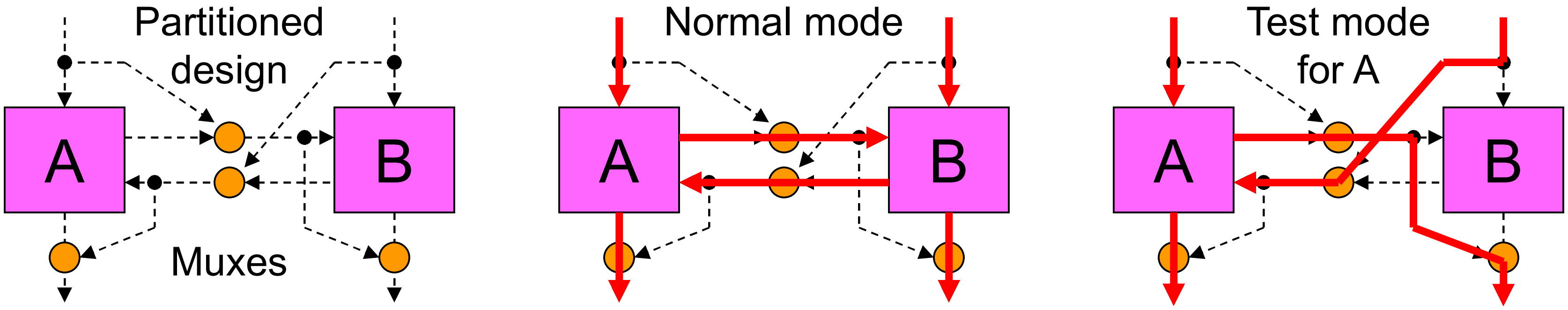
Overall testability of a circuit = Average of line testabilities

11.3 Testpoint Insertion

Increase controllability and observability via the insertion of degating mechanisms and control points



Design for dual-mode operation
  Normal mode
  Test mode

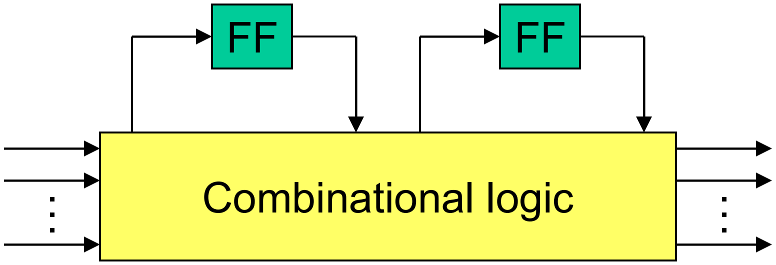


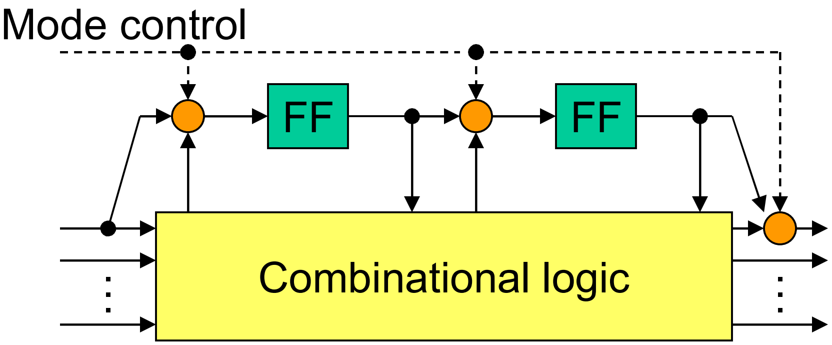
11.4 Sequential Scan Techniques

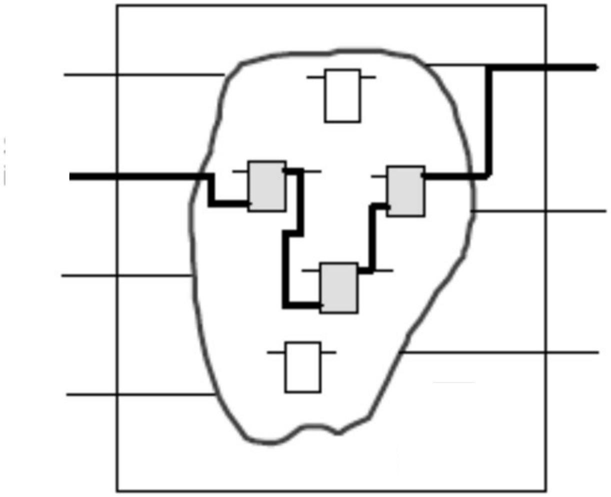
Increase controllability and observability via provision of mechanisms to set and observe internal flip-flops

Scan design

- Shift desired states into FF
- Shift out FF states to observe



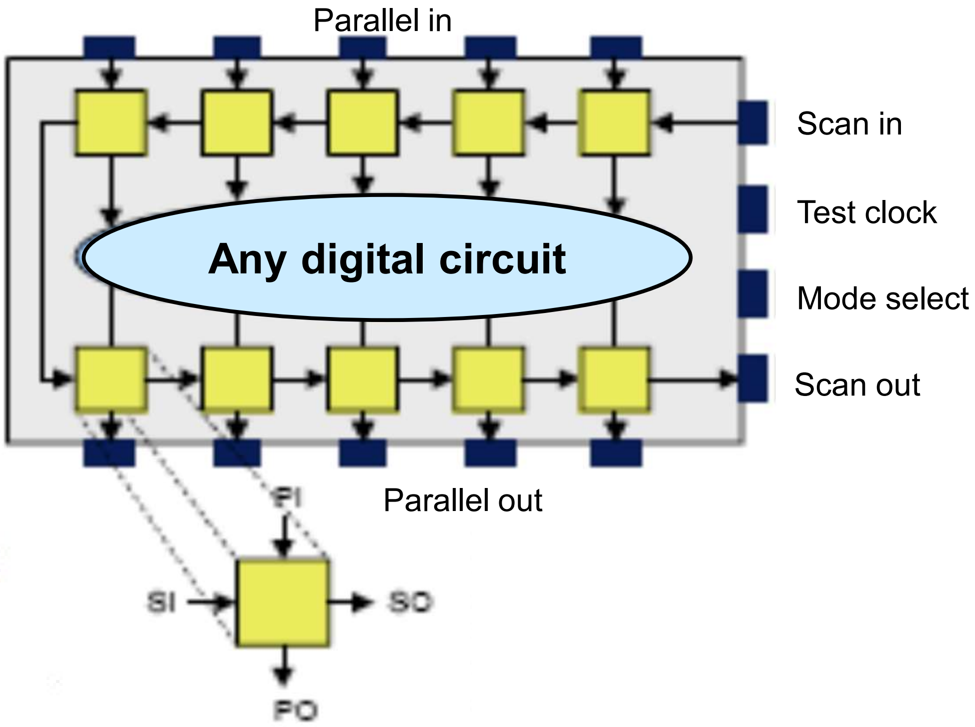




Partial scan design:
Mitigates the excessive overhead of a full scan design

11.5 Boundary Scan Design

Allows us to apply arbitrary inputs to circuit parts whose inputs would otherwise not be externally accessible



Parallel in

Scan in

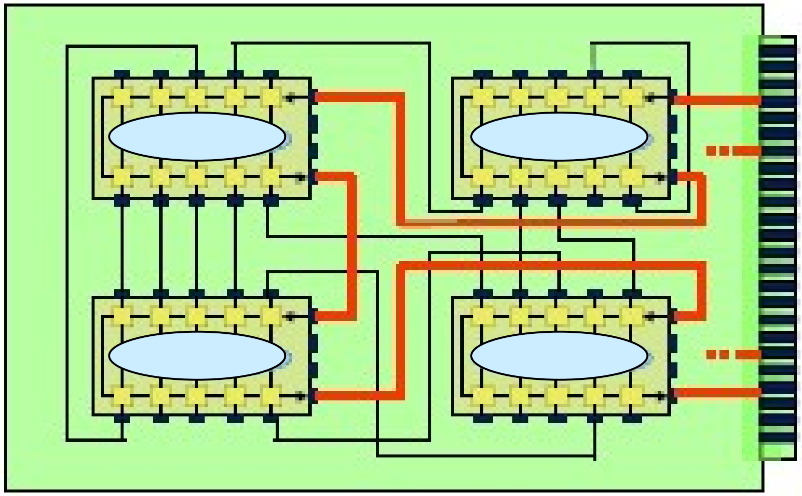
Test clock

Any digital circuit

Mode select

Scan out

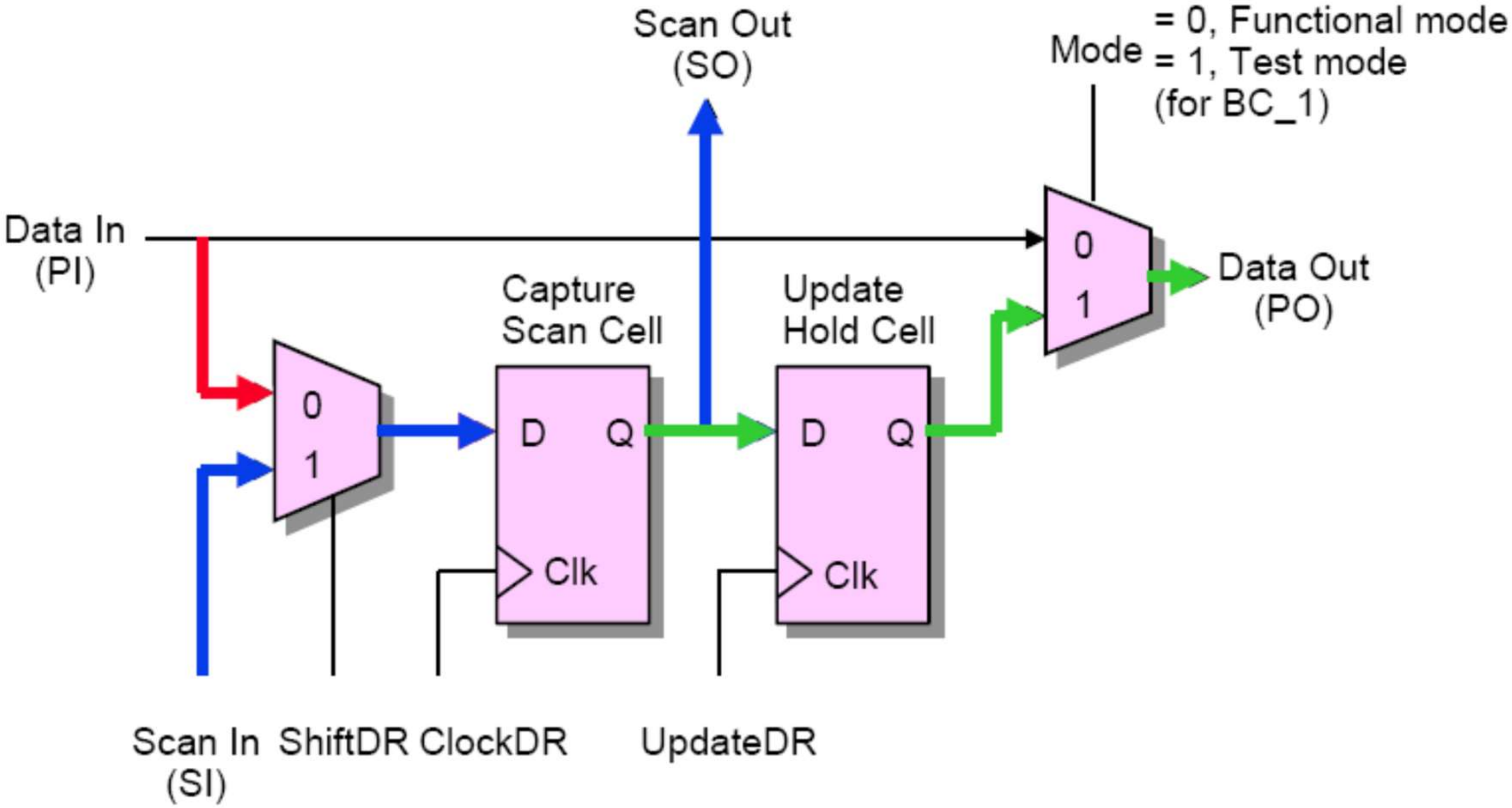
Parallel out



Boundary scan elements of multiple parts are cascaded together into a scan path

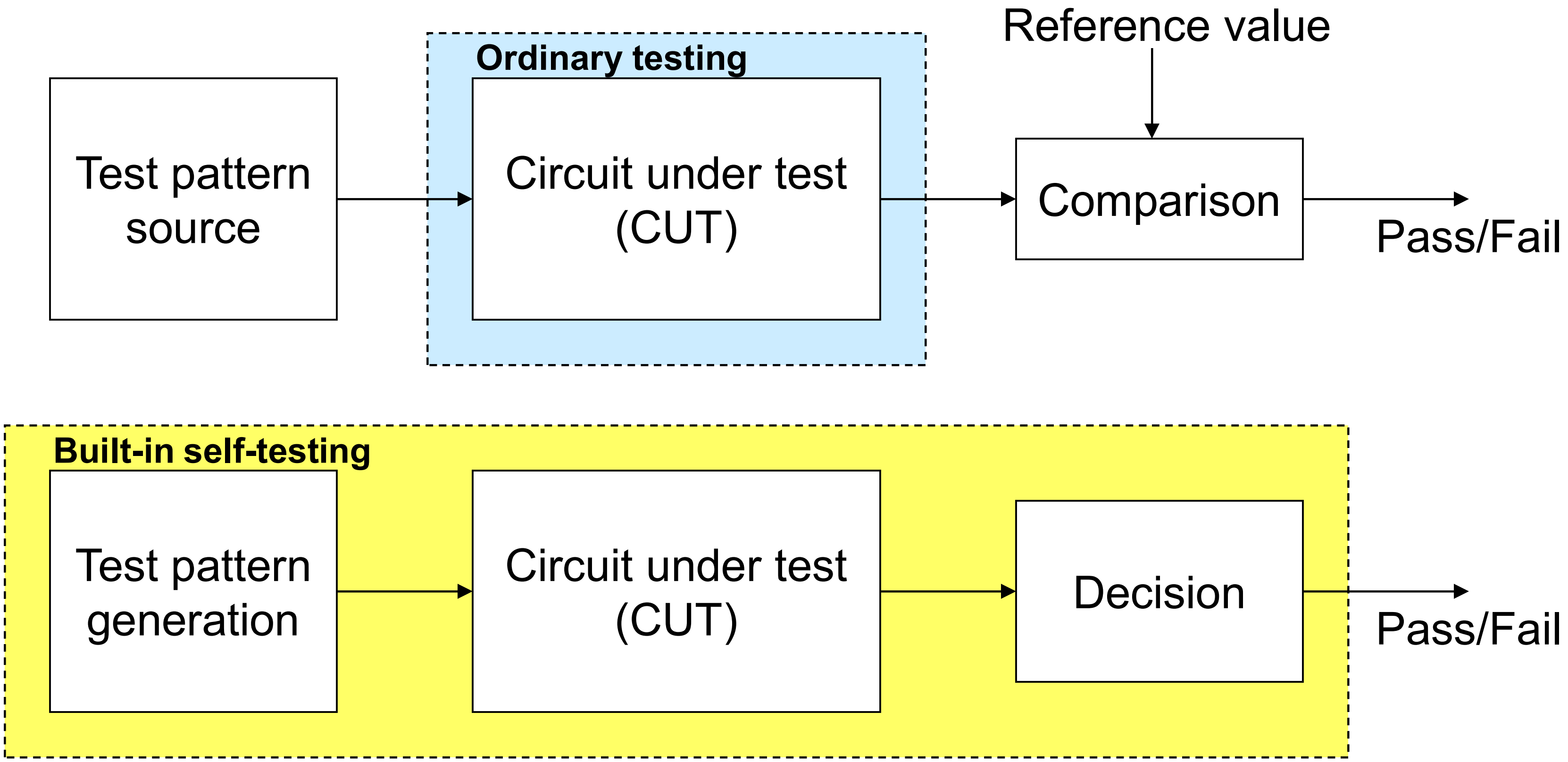
From: http://www.asset-intertech.com/pdfs/boundaryscan_tutorial.pdf

Basic Boundary Scan Cell



From: http://www.asset-intertech.com/pdfs/boundaryscan_tutorial.pdf

11.6 Built-in Self-Test (BIST)

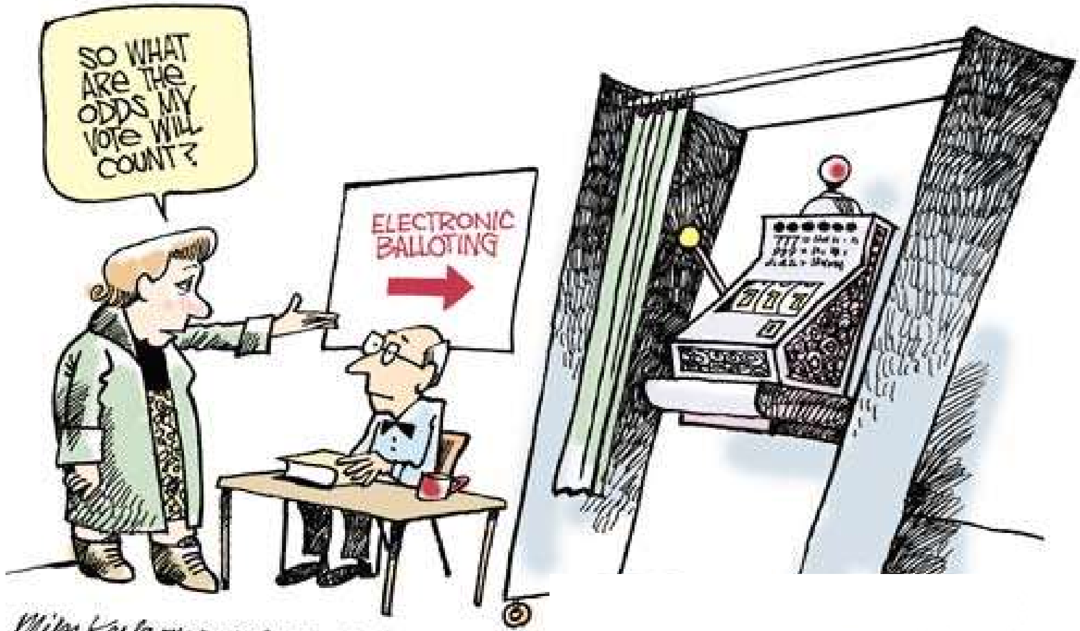


Test patterns may be generated (pseudo)randomly – e.g., via LFSRs

Decision may be based on compressed test results

12 Replication and Voting

"Let's try voting for the greater of the two evils this time and see what happens."



Mike Keefe THE DENVER POST 12/20/07 www.caglecartoons.com

© 2000 Randy Glasbergen.
www.glasbergen.com

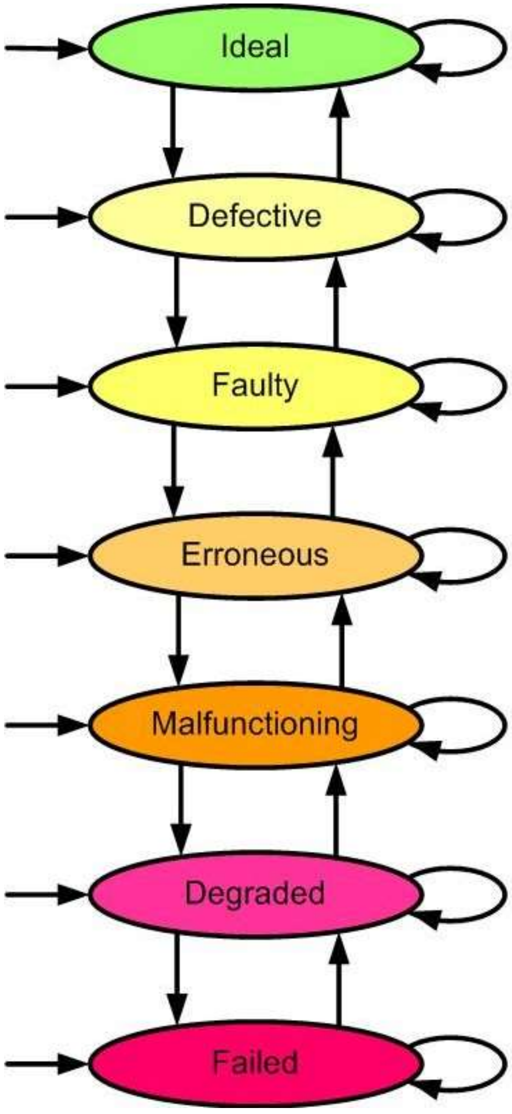
"AFTER MONTHS OF SPEECHES, PROMISES, AND ACCUSATIONS, I'VE DECIDED TO JUST VOTE FOR THE GUY WITH THE COOLEST WEB SITE."

"Fire. Bad. Those in favour?"

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



12.1 Hardware Redundancy Overview

Data path methods:

Replication in space (costly)
- Duplicate and compare
- Triplicate and vote
- Pair-and-spare
- NMR/hybrid

Replication in time (slow?)
- Recompute and compare
- Recompute and vote
- Alternating logic
- Recompute after shift
- Recompute after swap
- Replicate operand segments

Mixed space-time replication

Monitoring (imperfect coverage)
- Watchdog timer
- Activity monitor

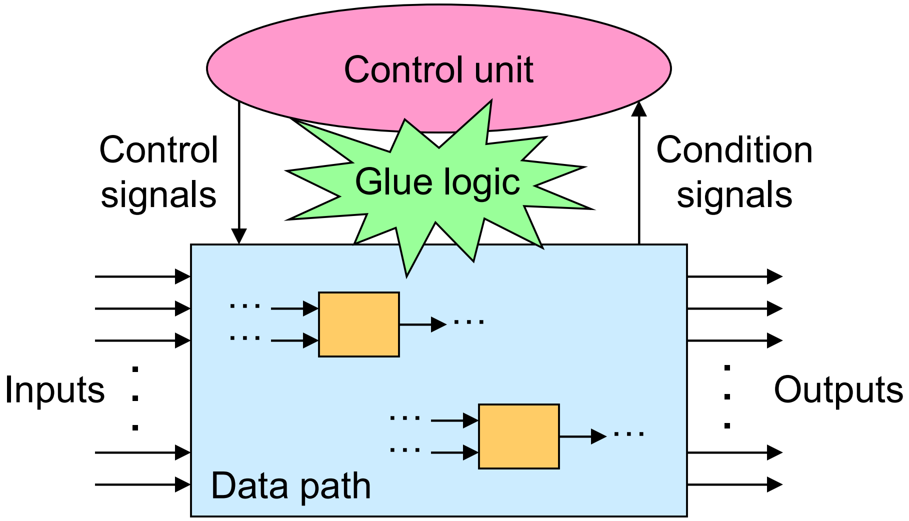
Low-redundancy coding
- Parity prediction
- Residue checking
- Self-checking design

Control unit methods:

Coding of control signals
Control-flow watchdog
Self-checking design

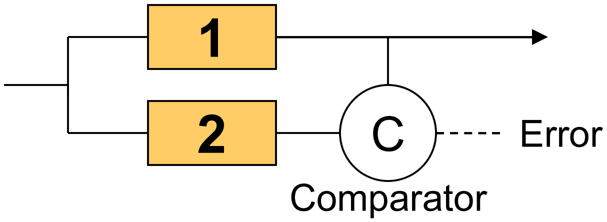
Glue logic methods:

Self-checking design

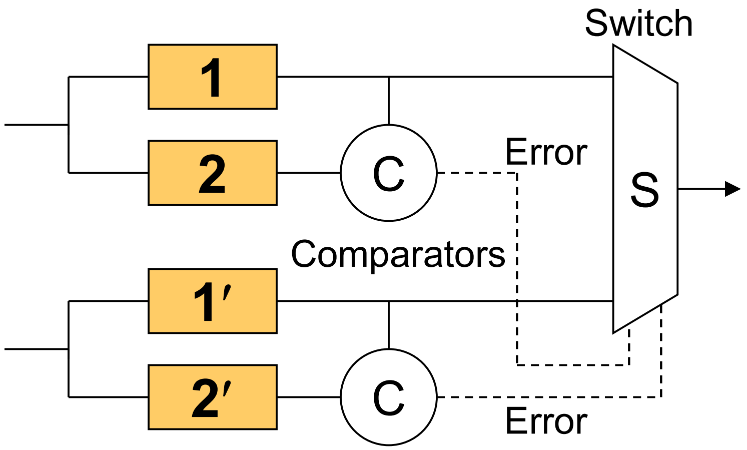


12.2 Replication in Space

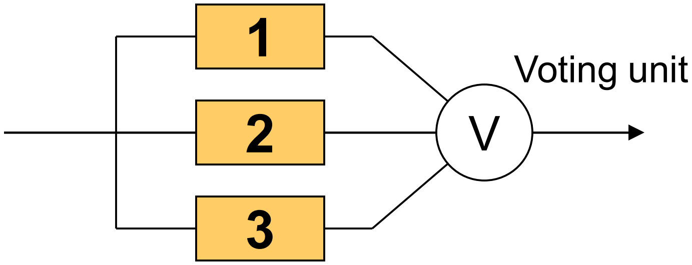
The following schemes have already been discussed in connection with fault masking



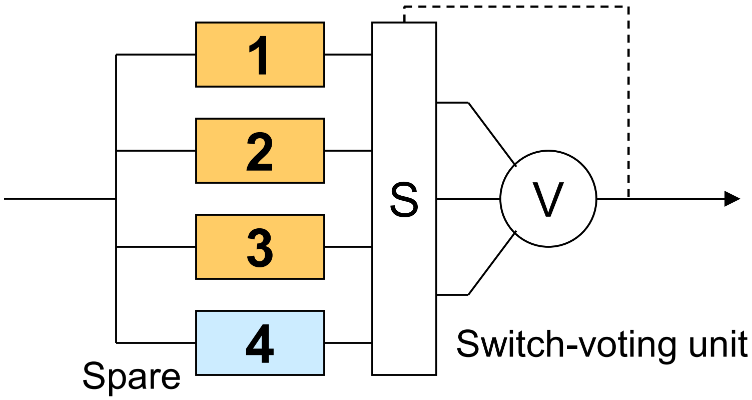
Duplicate and compare



Pair-and-spare



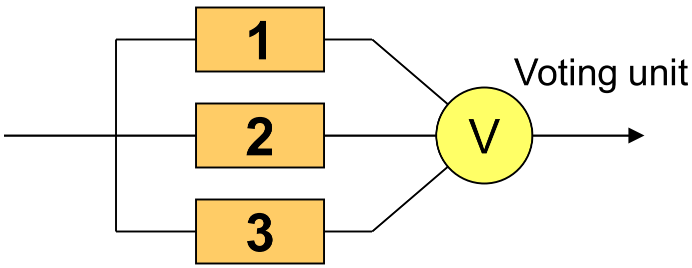
Triplicate and vote



NMR/Hybrid

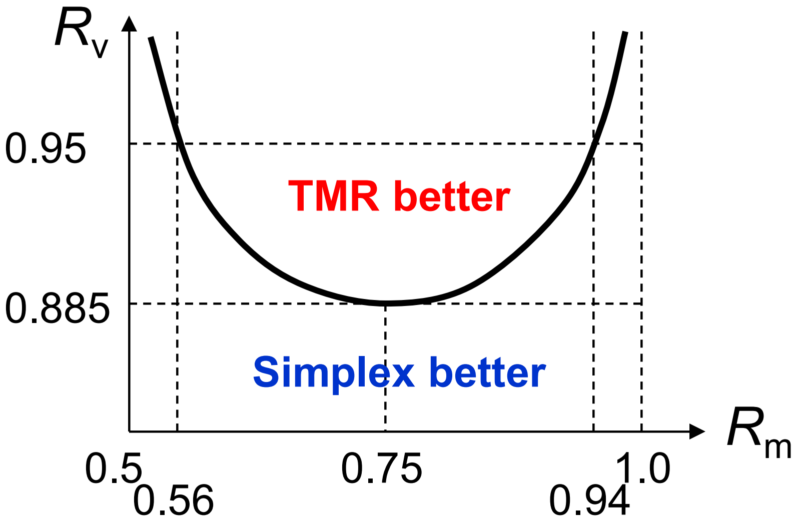
TMR with Imperfect Voting Unit

$R = R_v(3R_m^2 - 2R_m^3) \overset{?}{>} R_m$



Condition on the voting unit reliability
$R_v > 1/[3R_m - 2R_m^2]$

$dR_v^{\min}/dR_m = (-3 + 4R_m)/(3R_m - 2R_m^2)^2$



Condition on the module reliability

$$\frac{3 - \sqrt{9 - 8/R_v}}{4} < R_m < \frac{3 + \sqrt{9 - 8/R_v}}{4}$$

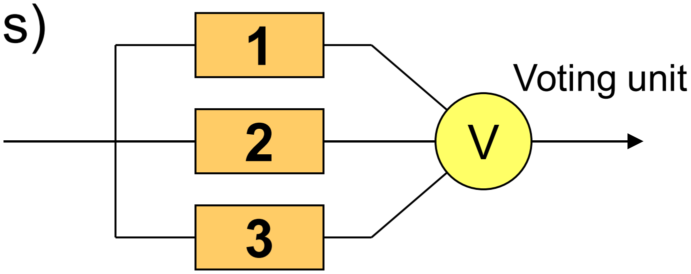
Example: $R_v = 0.95$ requires that
$0.56 < R_m < 0.94$

When $R_v = 1 - \varepsilon$ is close to 1, we have
$1/R_v \approx 1 + \varepsilon$ and $(1 - 8\varepsilon)^{0.5} \approx 1 - 4\varepsilon$, leading to $0.5 + \varepsilon < R_m < 1 - \varepsilon$

TMR with Compensating Faults

$R_m = 1 - p_0 - p_1$ (0- and 1-fault probabilities)

$R = (3R_m^2 - 2R_m^3) + 6p_0p_1R_m$



Example: $R_m = 0.998$, $p_0 = p_1 = 0.001$

$R = \underbrace{0.999{,}984}_{\text{Basic TMR}} + \underbrace{0.000{,}006}_{\text{Compensation}} = 0.999{,}990$

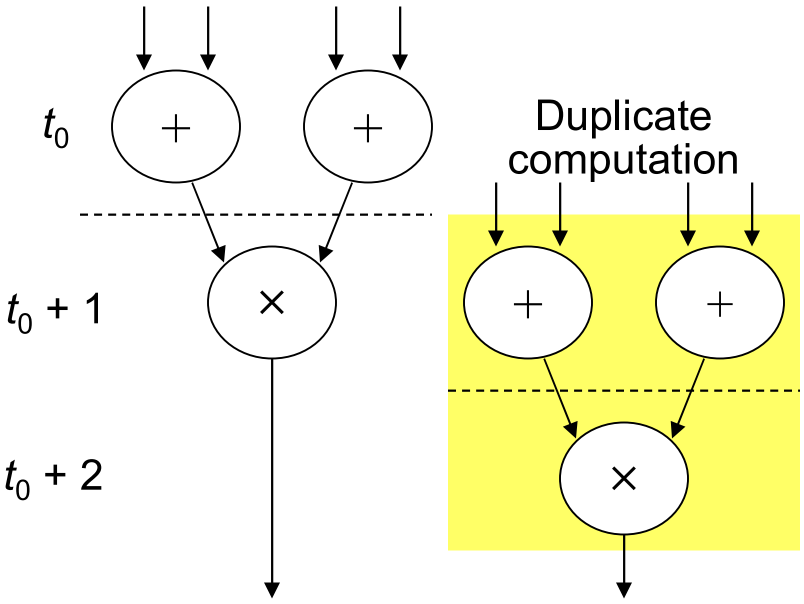
$\text{RIF}_{\text{TMR/Simplex}} = 0.002 / 0.000{,}016 = 125$

$\text{RIF}_{\text{Compen/TMR}} = 0.000{,}016 / 0.000{,}010 = 1.6$

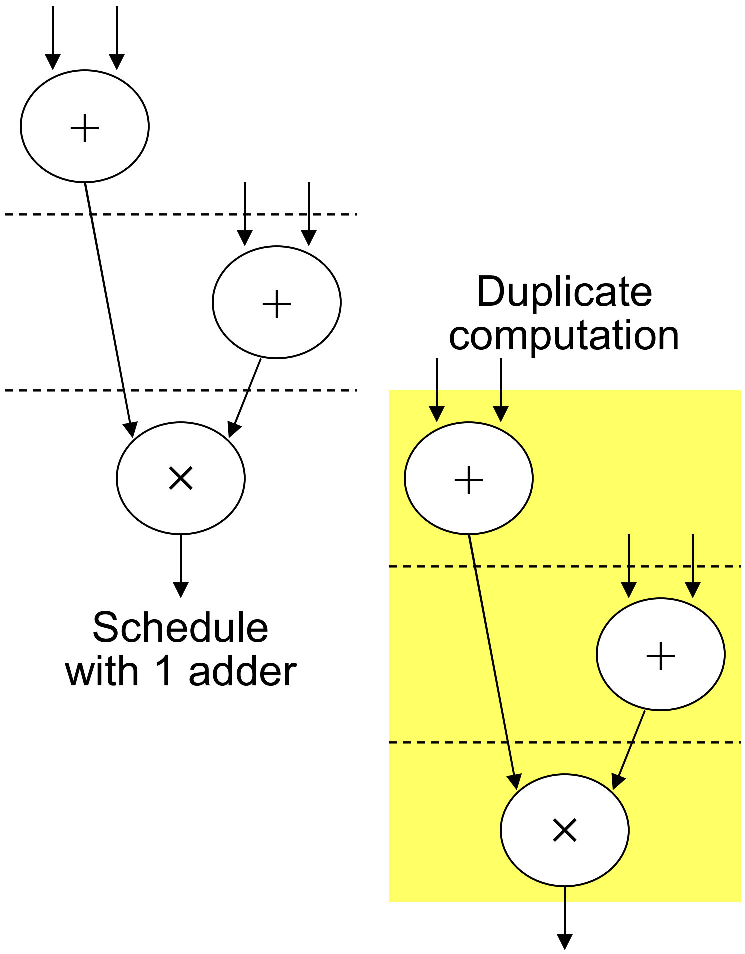
12.3 Replication in Time

Can be slow, but in many control applications, extra time is available

Interleaving of the primary and duplicate computations saves time



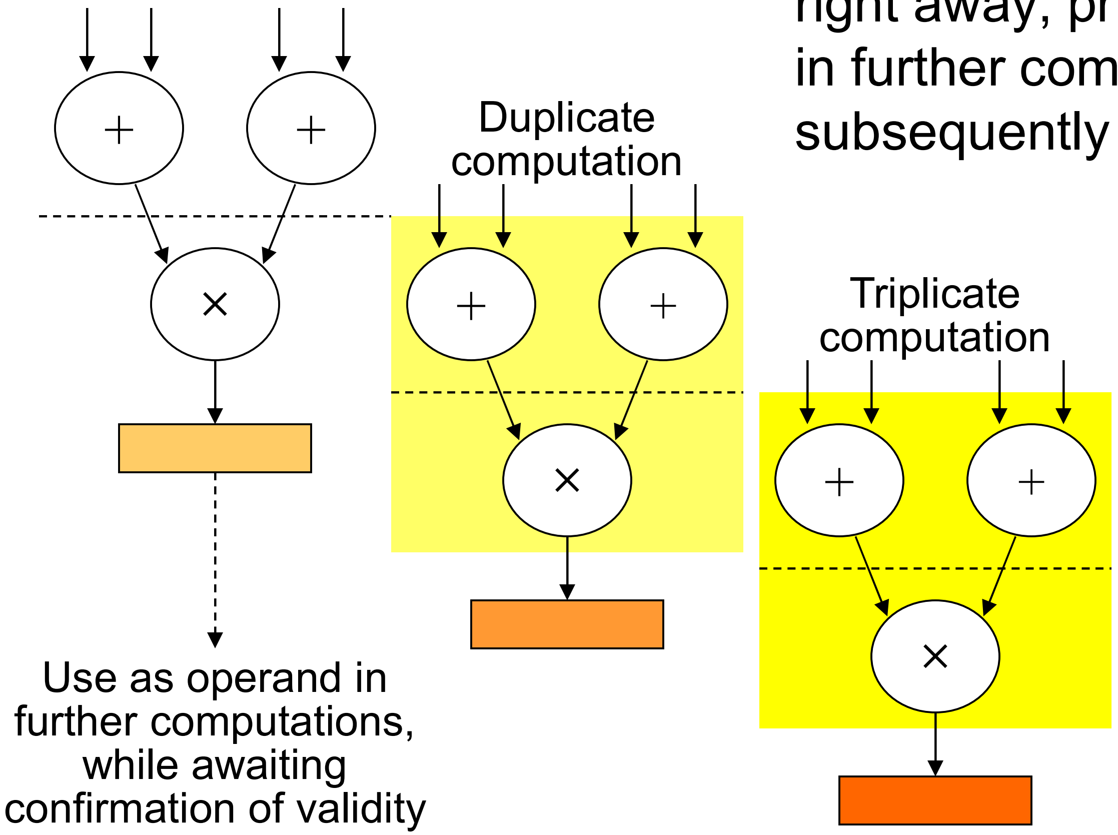
Computation flowgraph, and schedule with 2 adders



Schedule with 1 adder

Recompute and Compare/Vote

Repeat computation and store the results for comparison or voting


+
+
×
Duplicate computation
+
+
×
Triplicate computation
+
+
×
Use as operand in further computations, while awaiting confirmation of validity

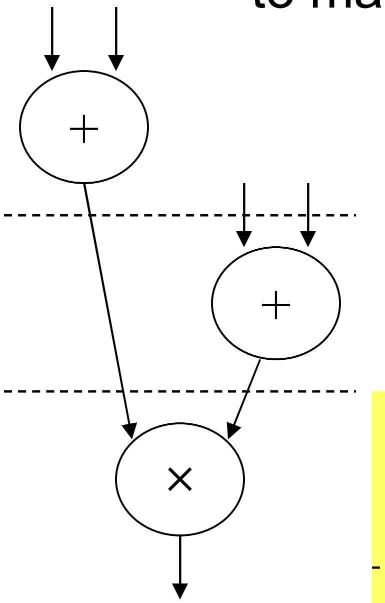
Comparison or voting need not be done right away; primary result may be used in further computations, with the result subsequently validated, if appropriate

On a simultaneous multithreading architecture, multiple instruction streams may be interspersed

Some Cray machines take advantage of extensive hardware resources to execute instructions twice

12.4 Mixed Space/Time Replication

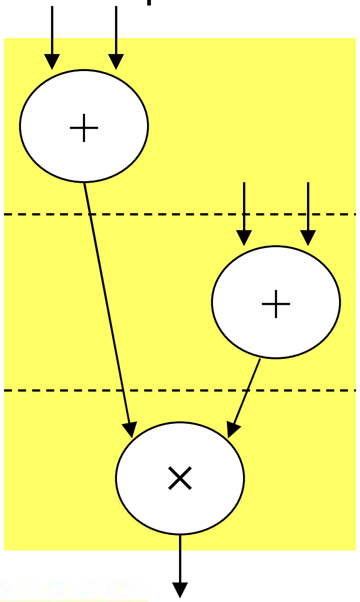
Instead of duplicating the computation with no hardware change (slow) or duplicating the entire hardware (costly), we can add some hardware to make the interleaved recomputations more efficient



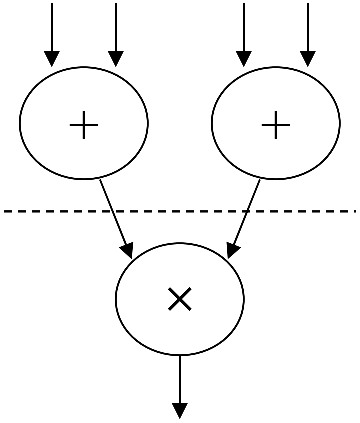
Original computation (T = 3)

Recomputation with same hardware resources (T = 5, excluding compare time)

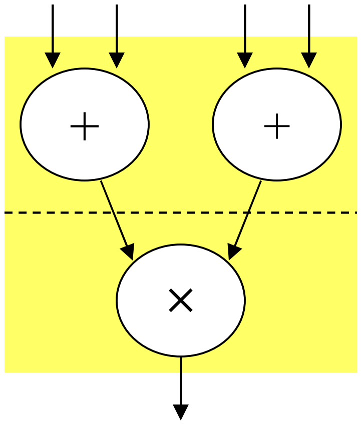
Duplicate computation



Consider the effect of including a second adder



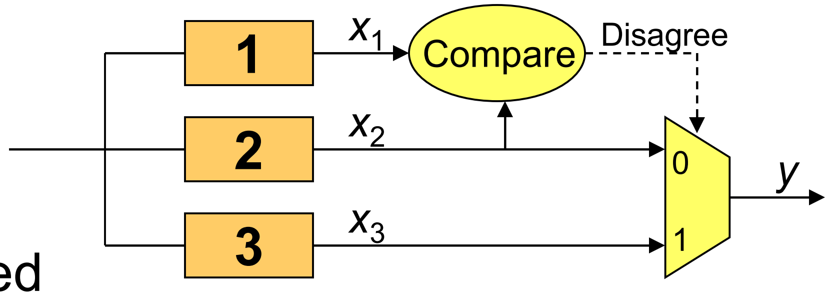
Recomputation with the inclusion of an extra adder (T = 3, excluding compare time)



12.5 Switching and Voting Units

We begin with some simple voting unit designs:

If in the case of 3-way disagreement any of the inputs can be chosen, then a simple design is possible



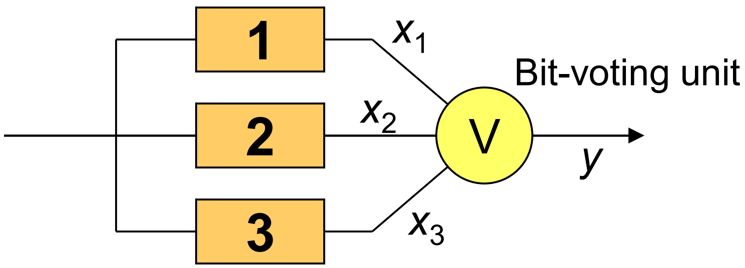
This design can be readily generalized to a larger number of inputs

One can perform pseudo voting that yields the median of 3 analog signals (Dennis, N.G., *Microelectronics and Reliability*, Aug. 1974)

Median and mean voting are also possible with digital signals

Implementing a Bit-Voting Unit

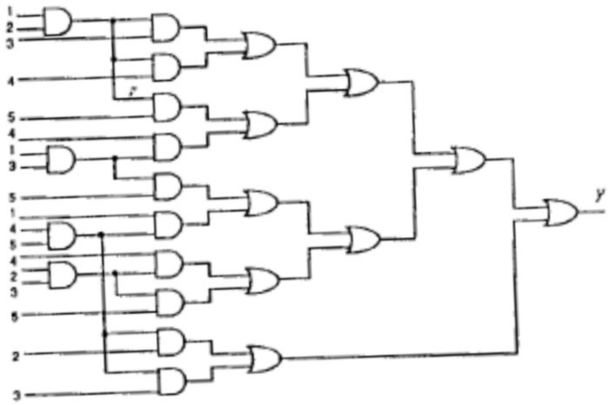
TMR bit-voting: $y = x_1x_2 \vee x_2x_3 \vee x_3x_1$
(carry output of a single-bit full-adder)
What about 5MR, 7MR?



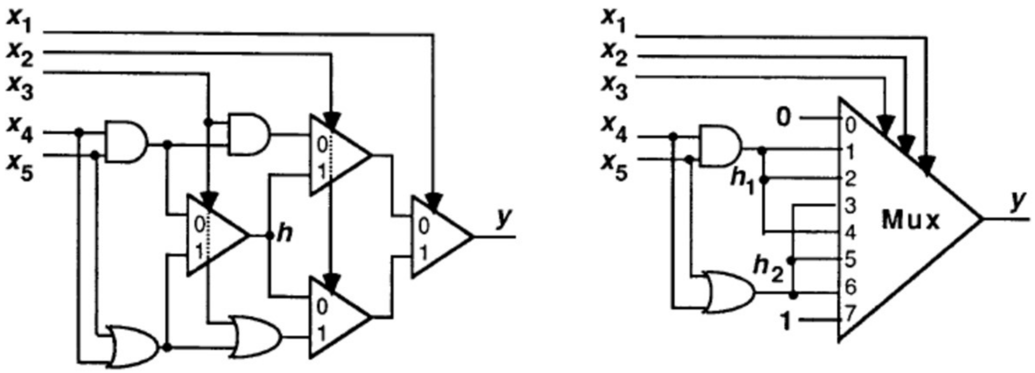
Gate-level design quickly explodes in size

Other designs are also possible
- Arithmetic: add the bits, compare to threshold
- Mux-based
- Selection-based (majority of bit values is their median)

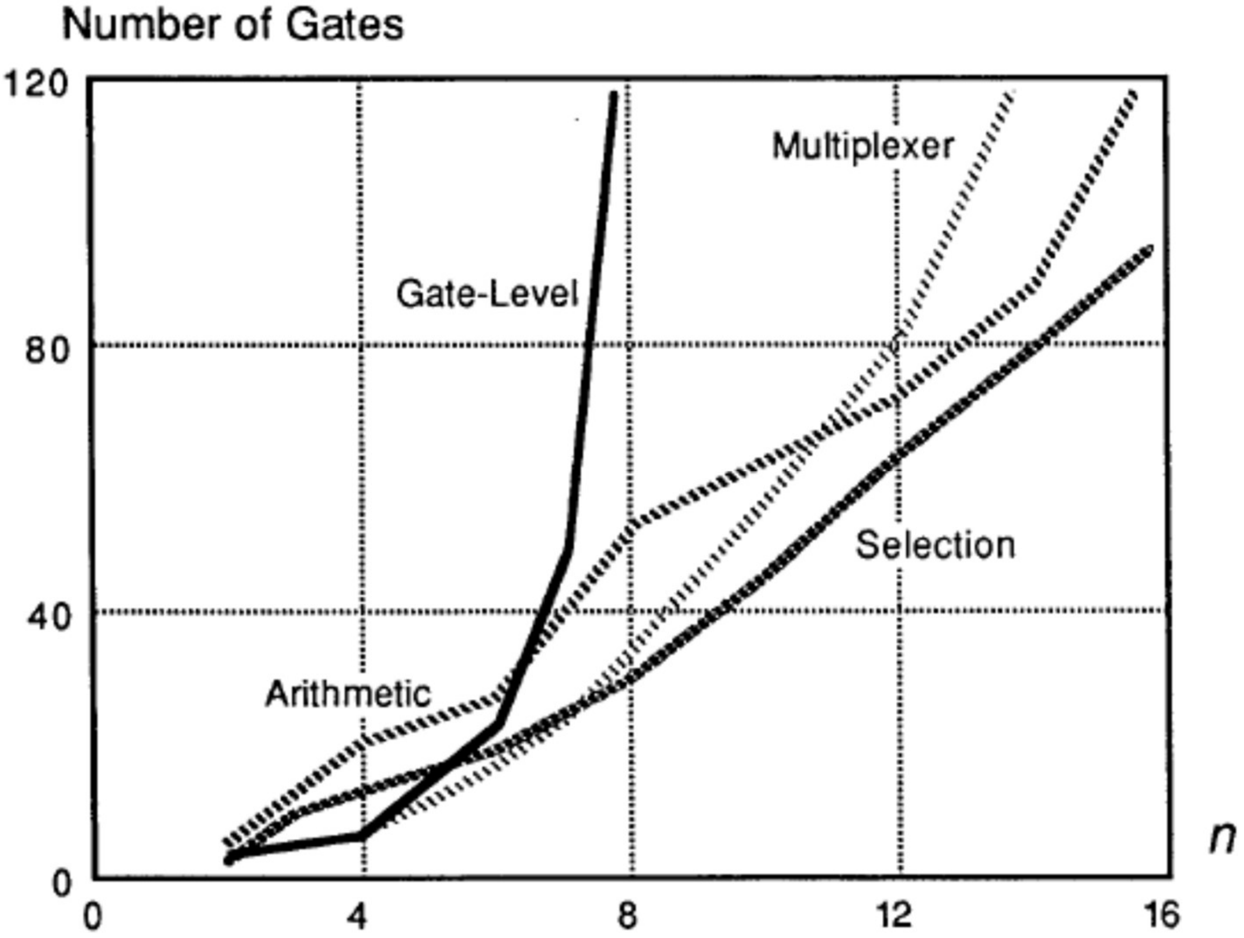
3-out-of-5 voting unit built of 2-input gates



Two mux-based designs for a 3-out-of-5 bit-voting unit



Complexity of Different Bit-Voting Unit Designs

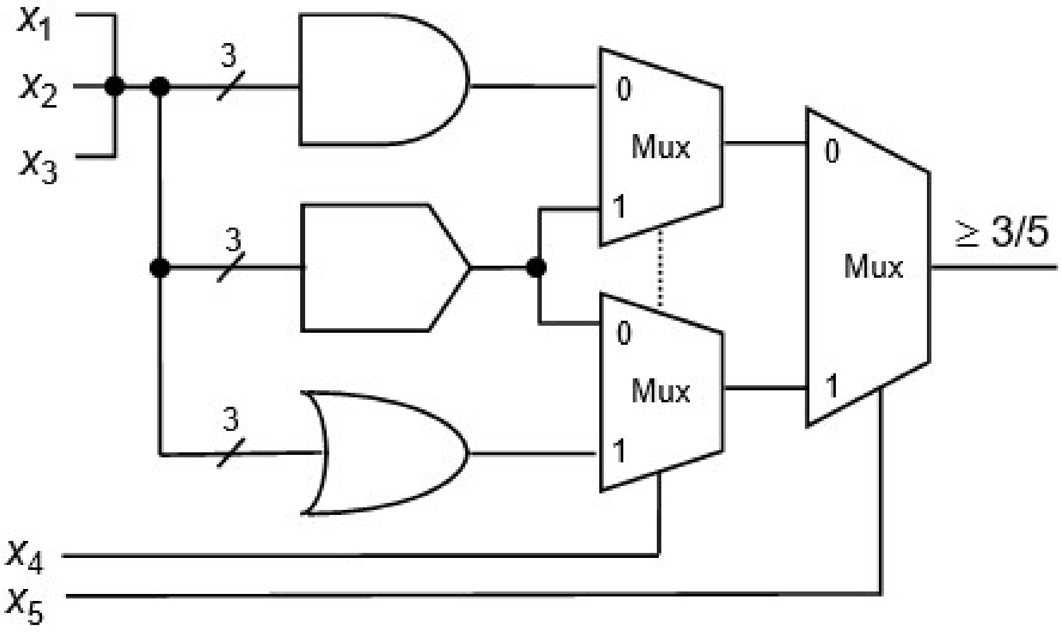


Cost of majority bit-voting units as a function of the number *n* of inputs

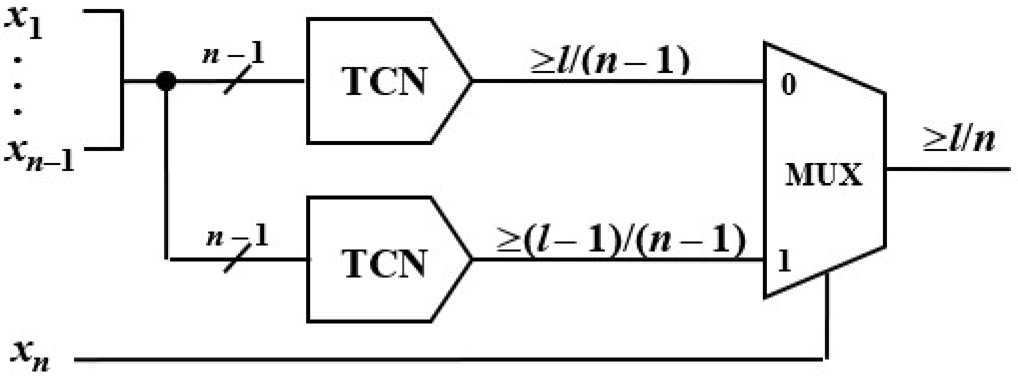
Majority-Friendly Nanotechnologies

Certain new nanotechnologies offer efficient majority gates

Can we use majority gates as building-blocks in realizing voters?

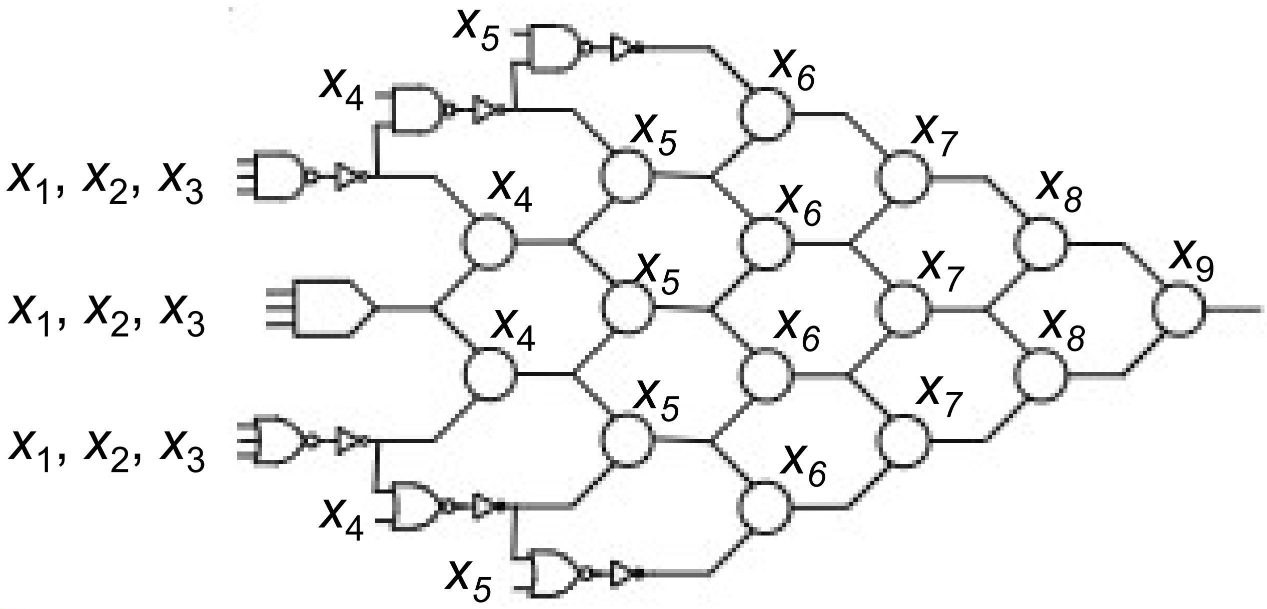


Recursive Construction of Large Voters



At-least-*l*-out-of-*n* threshold counting network built from a multiplexer and two smaller threshold counting networks

Recursively-built 5-out-of-9 voter



Voting at the Word Level

Using bit-by-bit voting may be dangerous

$$\begin{array}{l} x_1 = 0\ \ 0 \\ x_2 = 1\ \ 0 \\ \underline{x_3 = 1\ \ 1} \\ y\ = 1\ \ 0 \end{array}$$

One might think that in this example, any of the module outputs could be correct, so that producing 1 0 at the output isn't all that wrong

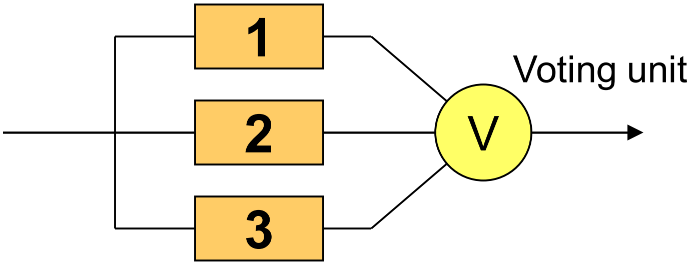
However, with bit-by-bit voting, the output may be different from all inputs

$$\begin{array}{l} x_1 = 0\ \ 0\ \ 0 \\ x_2 = 1\ \ 0\ \ 1 \\ \underline{x_3 = 1\ \ 1\ \ 0} \\ y\ = 1\ \ 0\ \ 0 \end{array}$$

Design of bit- and word-voting networks discussed in:
Parhami, B., "Voting Networks," *IEEE TR*, Aug. 1991

12.6 Variations and Design Issues

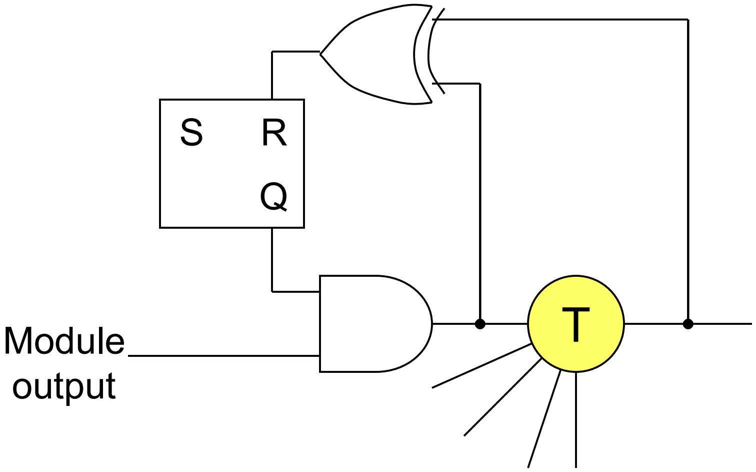
NMR/simplex: Voting unit is replaced with a unit that can also detects disagreements



When a faulty unit is detected, that unit and one other unit are removed from service

This makes all votes unambiguous and also improves systems lifetime

Self-purging redundancy: Modules purged when they disagree with the output and the threshold of the voting unit is adjusted accordingly (purged modules produce 0 outputs)



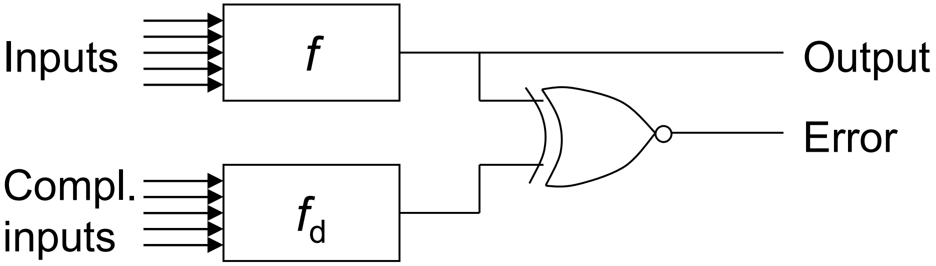
Alternating Logic: Basic Ideas

Transmission of data over unreliable wires or buses

- Send data; store at receiving end
- Send bitwise complement of data
- Compare the two versions
- Detects wires s-a-0 or s-a-1, as well as many transients

The *dual* of a Boolean function $f(x_1, x_2, \ldots, x_n)$ is another function $f_\mathrm{d}(x_1, x_2, \ldots, x_n)$ such that $f_\mathrm{d}(x_1', x_2', \ldots, x_n') = f'(x_1, x_2, \ldots, x_n)$

Fact: Obtain the dual of f by exchanging AND and OR operators in its logical expression. For example, the dual of $f = ab \vee c$ is $f_\mathrm{d} = (a \vee b)c$

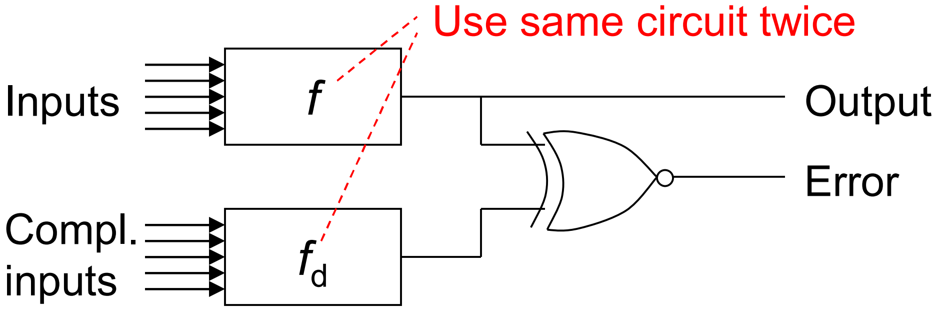


Advantages of this approach compared with duplication include a smaller probability of common errors

Alternating Logic: Self-Dual Functions

A function f is self-dual if $f(x_1, x_2, \ldots, x_n) = f_d(x_1, x_2, \ldots, x_n)$

For example, both the sum $a \oplus b \oplus c$ and carry $ab \vee bc \vee ca$ outputs of a full-adder are self-dual functions



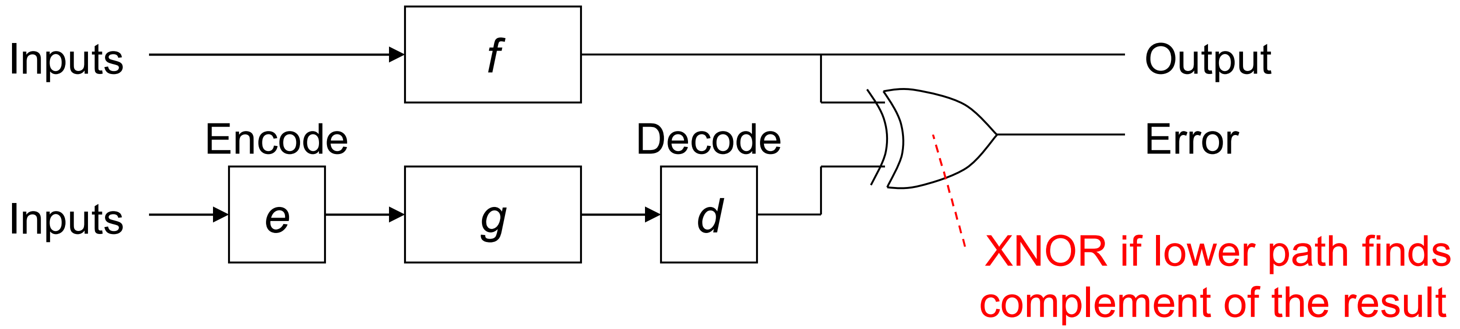
With a self-dual function f, the functions f and f_d in the diagram above can be computed by using the same circuit twice (time redundancy)

Many functions of practical interest are self-dual

Examples (proofs left as exercise)
A k-bit binary adder, with $2k + 1$ inputs and $k + 1$ outputs, is self-dual
So are 1's-complement and 2's-complement versions of such an adder

Recomputing with Transformed Operands

Alternating logic is a special case of the following general scheme, with its encoding and decoding functions being bitwise complementation



XNOR if lower path finds complement of the result

Recompute after shift

When *f* is binary addition, we can use shifts for encoding and decoding

Shifting causes the adder circuits to be exercised differently each time

Originally proposed for ALUs with bit-slice organization

Recompute after swap

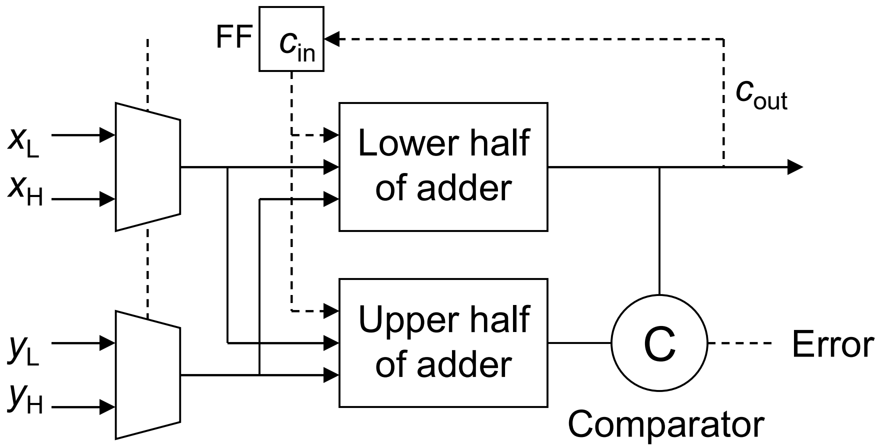
When *f* is binary addition, we can use swaps for encoding and decoding

Swap the two operands; e.g., compute $b + a$ instead of $a + b$

Swap upper and lower halves of the two operands (modified adder)

Time-Redundant, Segmented Addition

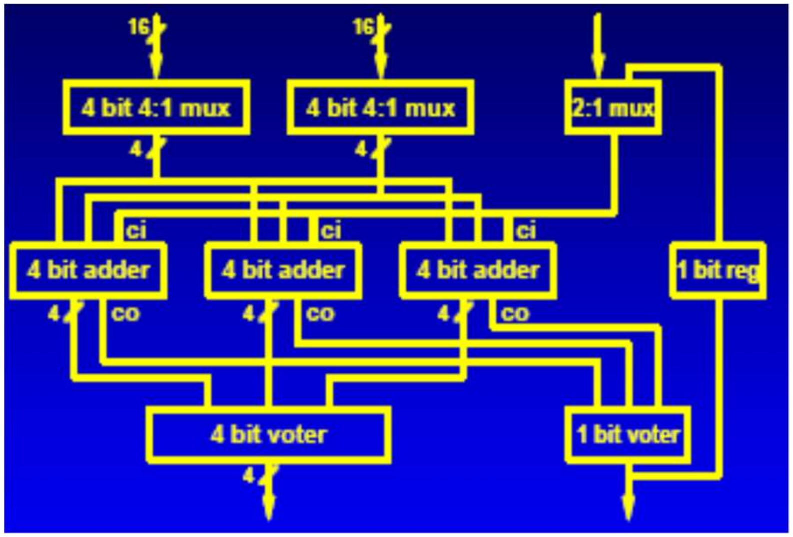
Instead of using a *k*-bit adder twice for error detection or 3 times for error correction, one can segment the operands into 2 or 3 parts and similarly segment the adder; perform replicated addition on operand segments and use comparison/voting to detect/correct error



Sum computed in two cycles: The lower half in cycle 1, and the upper half in cycle 2

Various other segmentation schemes have been suggested

Example: 16-bit adder with 4-way segmentation and voting



Townsend, Abraham, and Swartzlander, 2003