Fault-Tolerant Computing

About This Presentation

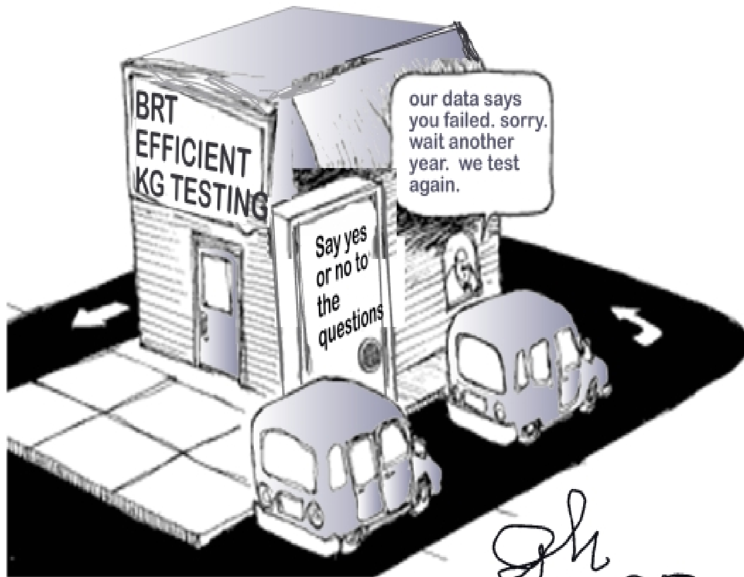
This presentation has been prepared for the graduate course ECE 257A (Fault-Tolerant Computing) by Behrooz Parhami, Professor of Electrical and Computer Engineering at University of California, Santa Barbara. The material contained herein can be used freely in classroom teaching or any other educational setting. Unauthorized uses are prohibited. © Behrooz Parhami

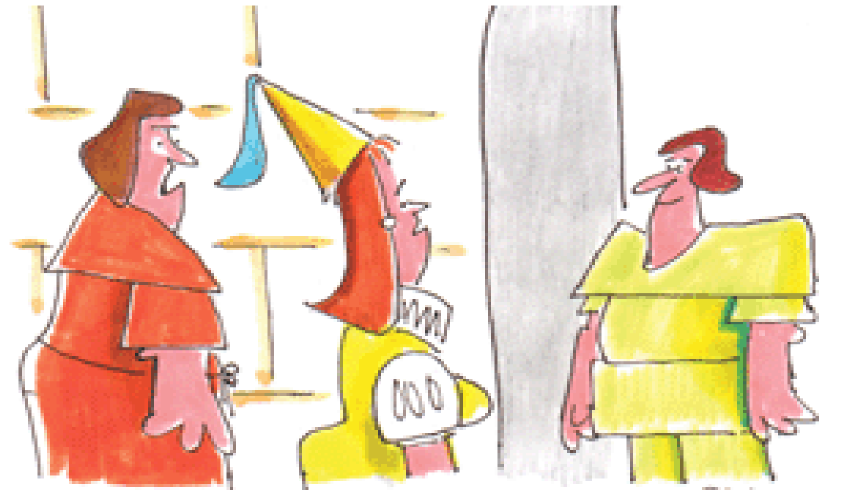
Edition	Released	Revised	Revised
First	Oct. 2006		

Self-Checking Modules

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

Earl checks his balance at the bank.





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

Multilevel Model of Dependable Computing

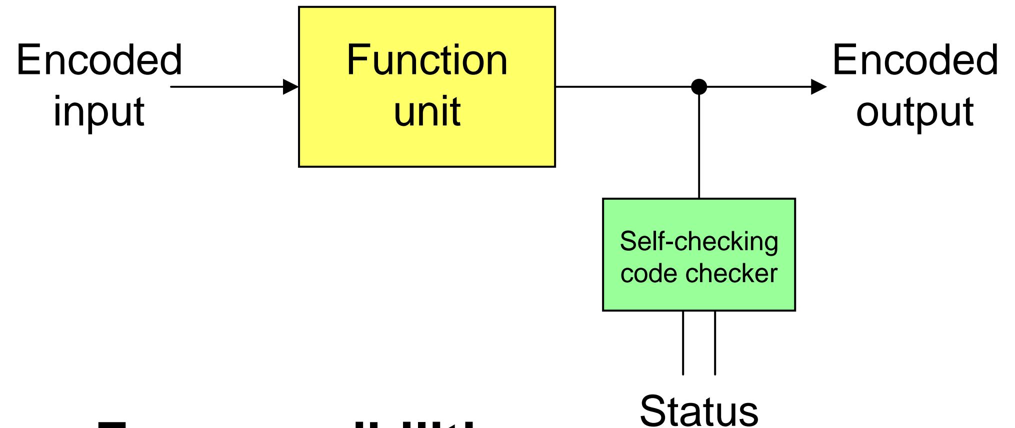
Level → Component Logic Information System Service Result

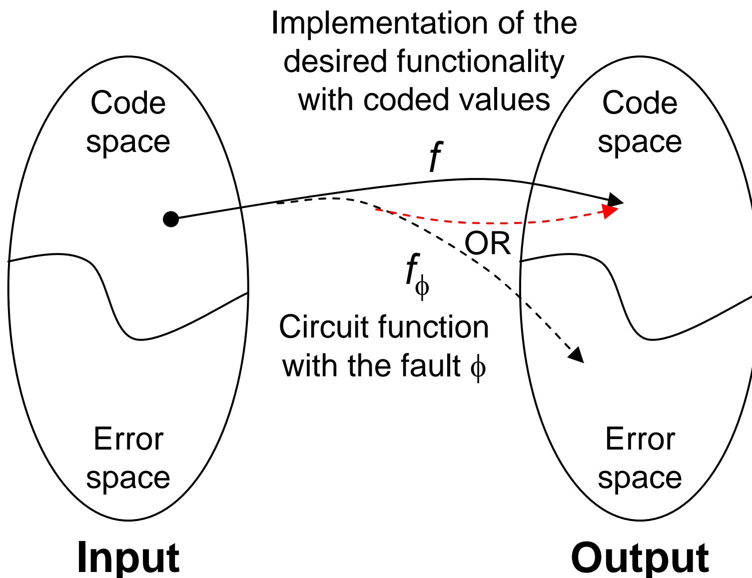
Ideal
Defective
Faulty
Erroneous
Malfunctioning
Degraded
Failed

Unimpaired
Low-Level Impaired
Mid-Level Impaired
High-Level Impaired

Main Ideas of Self-Checking Design

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





Four possibilities:

Both function unit and checker okay

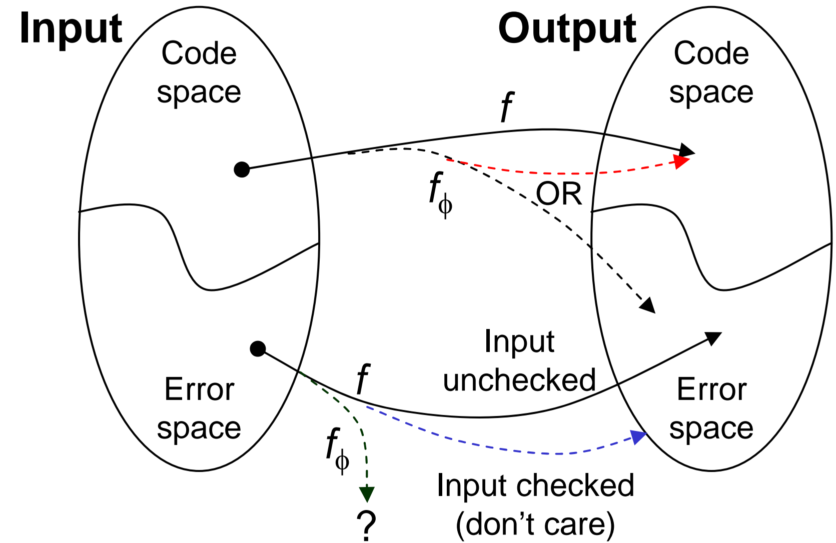
Only function unit okay (false alarm may be raised, but this is safe)

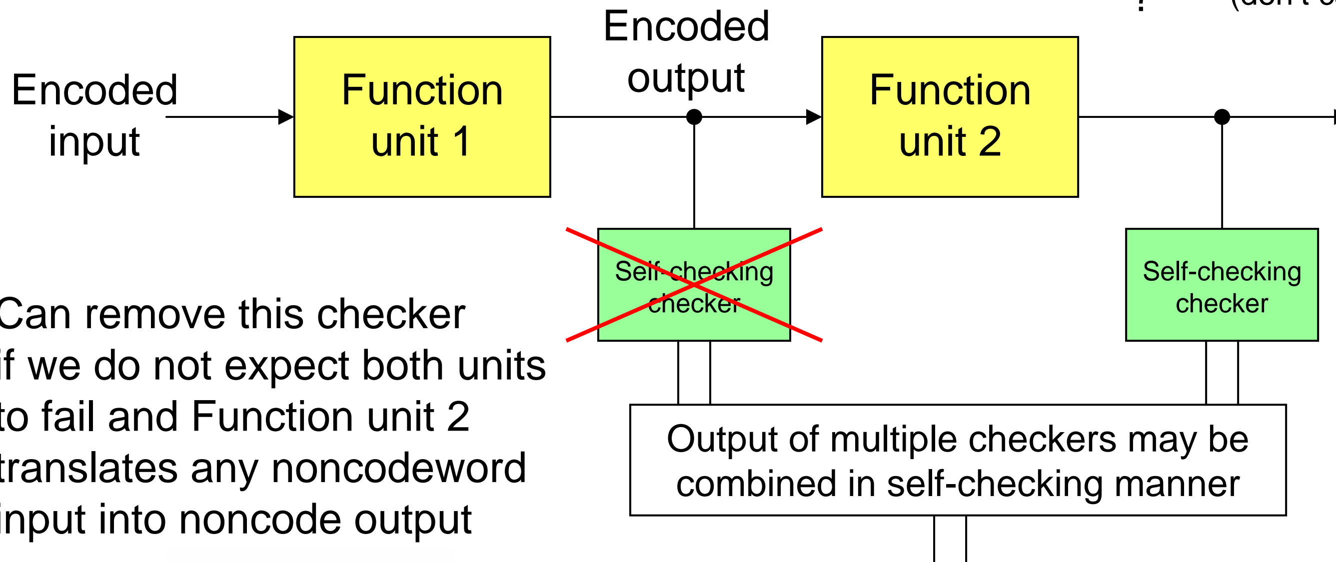
Only checker okay (we have either no output error or a detectable error)

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

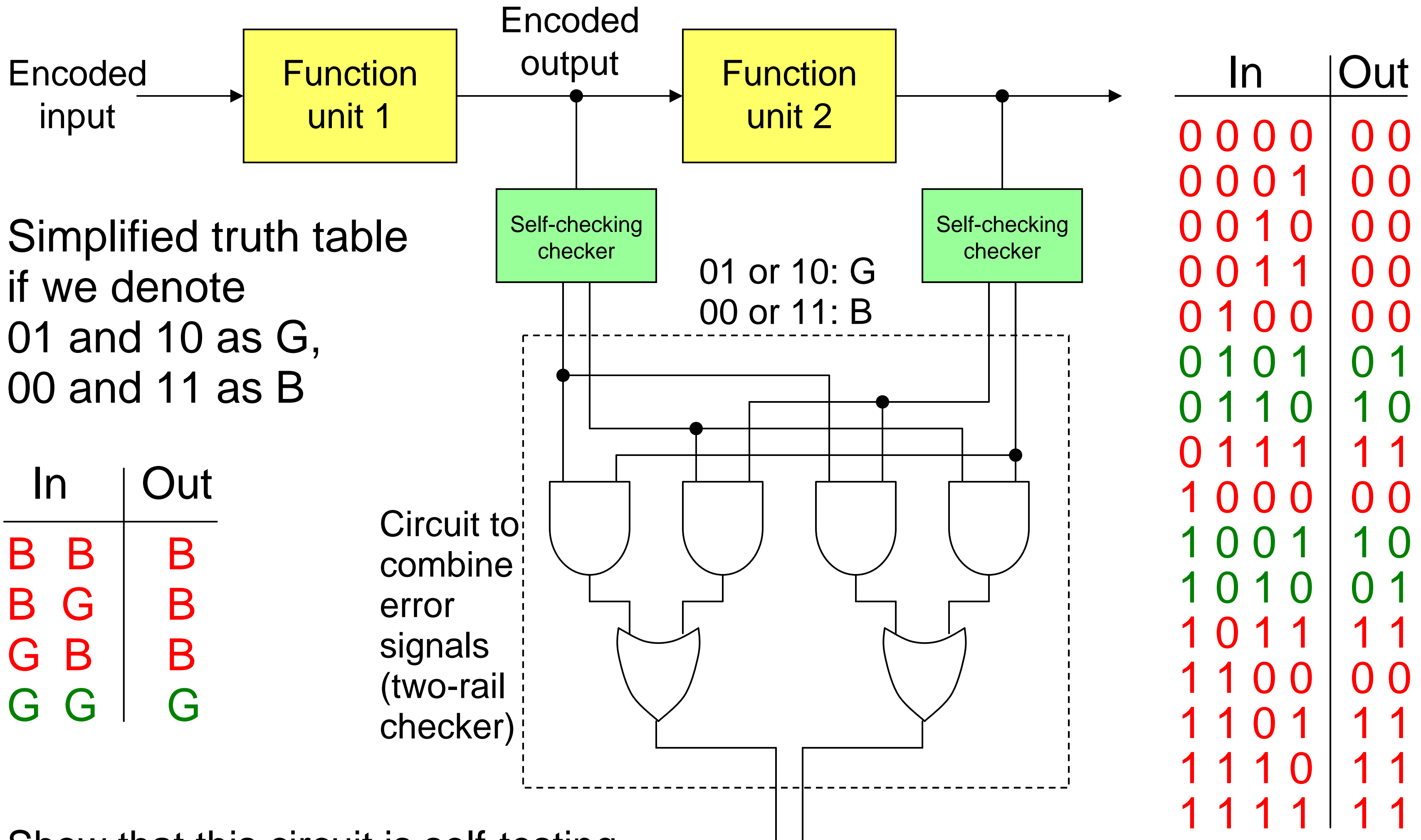
Given self-checking modules that have been designed separately, how does one combine them into a self-checking system?





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

Totally-Self-Checking Error Signal 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

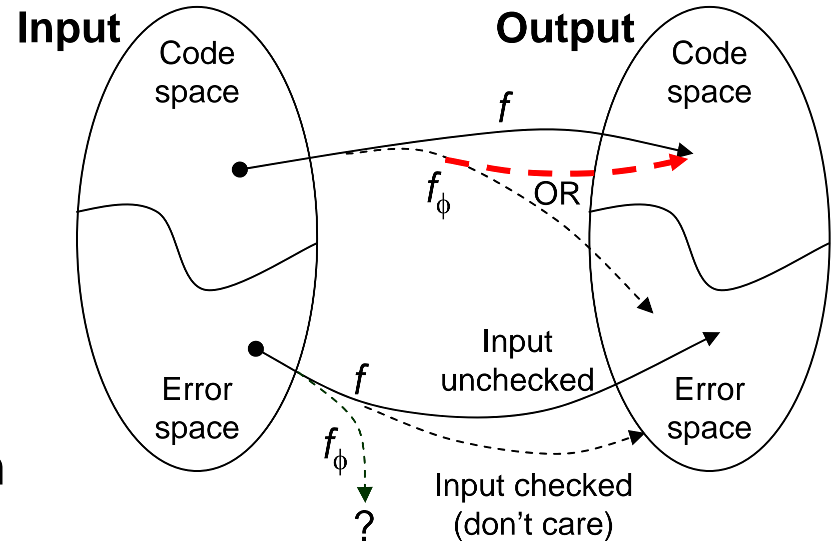
In	Out
0 0 0 0	0 0
0 0 0 1	0 0
0 0 1 0	0 0
0 0 1 1	0 0
0 1 0 0	0 0
0 1 0 1	0 1
0 1 1 0	1 0
0 1 1 1	1 1
1 0 0 0	0 0
1 0 0 1	1 0
1 0 1 0	0 1
1 0 1 1	1 1
1 1 0 0	0 0
1 1 0 1	1 1
1 1 1 0	1 1
1 1 1 1	1 1

Show that this circuit is self-testing

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)

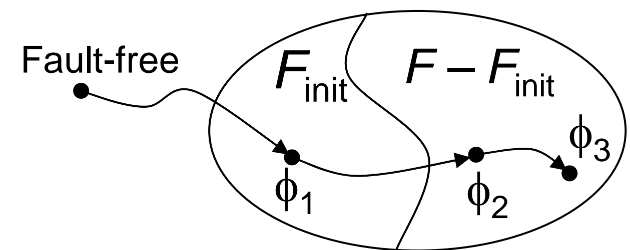
Self-Monitoring Design

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 $F_{init} \subseteq F$, chosen such that all faults in F develop in time as a sequence of simpler faults, the first of which is in F_{init}

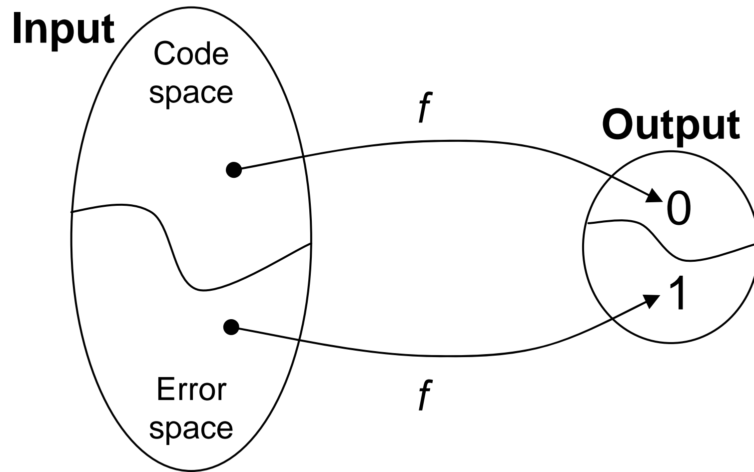
Example:
A unit that is totally-self-checking wrt single faults may be deemed self-monitoring wrt to multiple faults, provided that multiple faults develop one by one and slowly over time



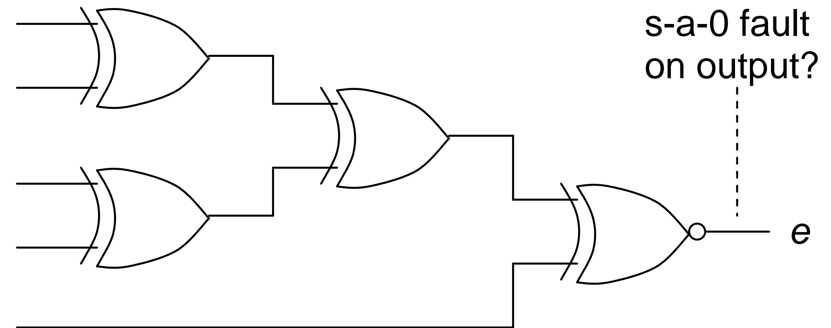
The self-monitoring design approach requires the more stringent totally-self-checking property to be satisfied for a small, manageable set of faults, while also protecting the unit against a broader fault class

Totally Self-Checking Checkers

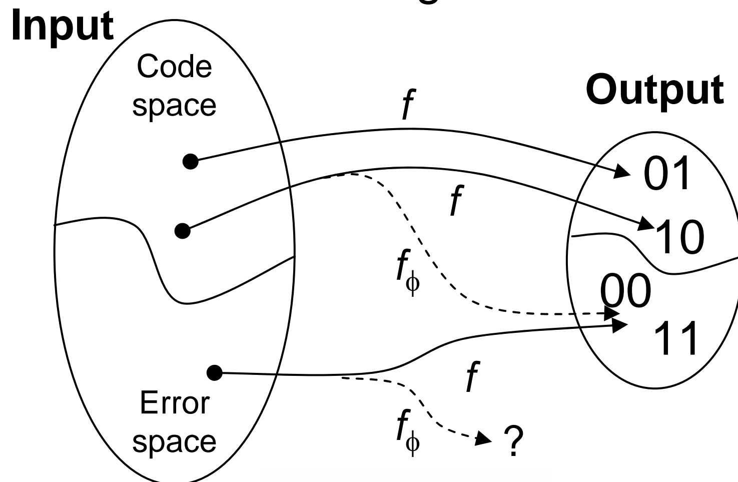
Conventional code checker



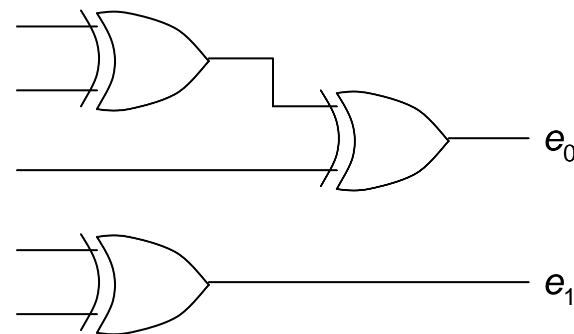
Example: 5-input odd-parity checker



Self-checking code checker



Example: 5-input odd-parity checker



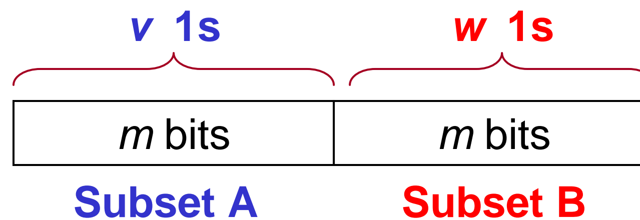
Pleasant surprise: The self-checking version is simpler!

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, $m = 3$, $A = \{a, b, c\}$, $B = \{f, g, h\}$

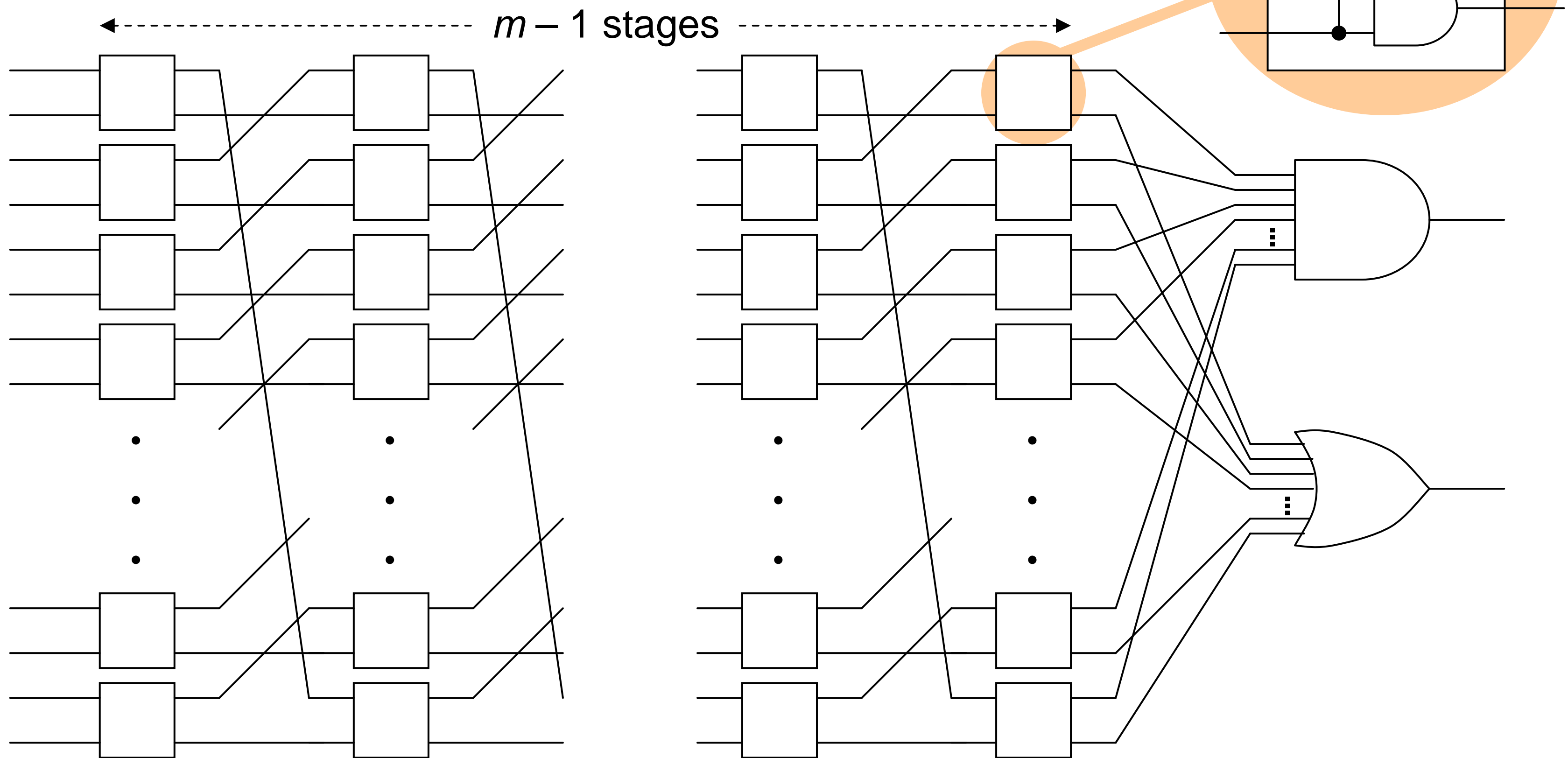
$$e_0 = (v \geq 0)(w \geq 3) \vee (v \geq 2)(w \geq 1) = fgh \vee (ab \vee bc \vee ca)(f \vee g \vee h)$$

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

Always satisfied

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

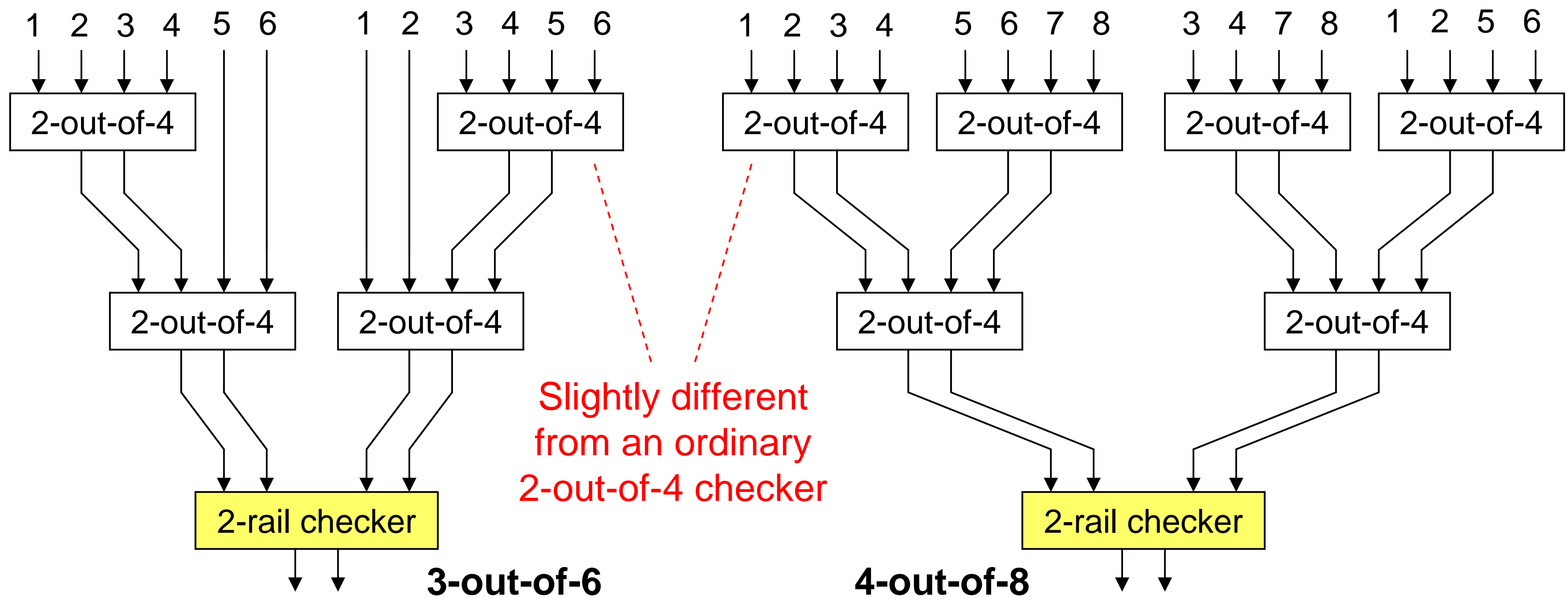
Cellular realization, due to J. E. Smith:
This design is testable with only 2*m* inputs, all having *m* consecutive 1s (in cyclic order)



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

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

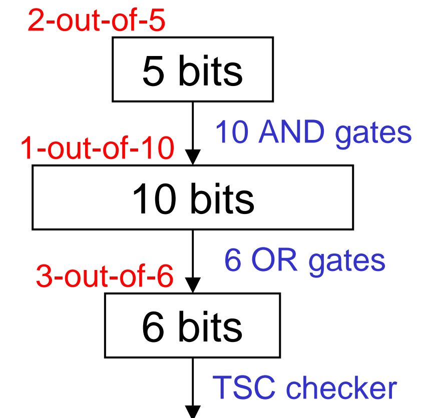
Examples: 3-out-of-6 and 4-out-of-8 TSC checkers are depicted below (only the structure is shown; some design details are missing)



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

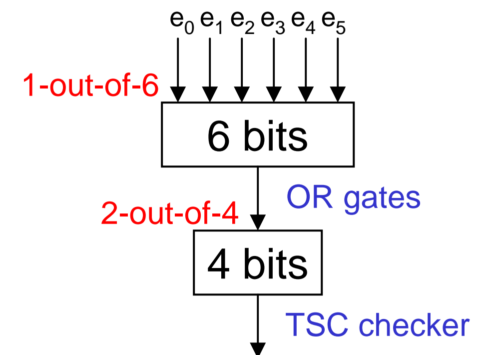
One design strategy is to proceed in 3 stages:
Convert the *k*-out-of-*n* code to a 1-out-of-$\binom{n}{k}$ code
Convert the latter code to an *m*-out-of-2*m* code
Check the *m*-out-of-2*m* code using a TSC checker

This approach is impractical for many codes



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

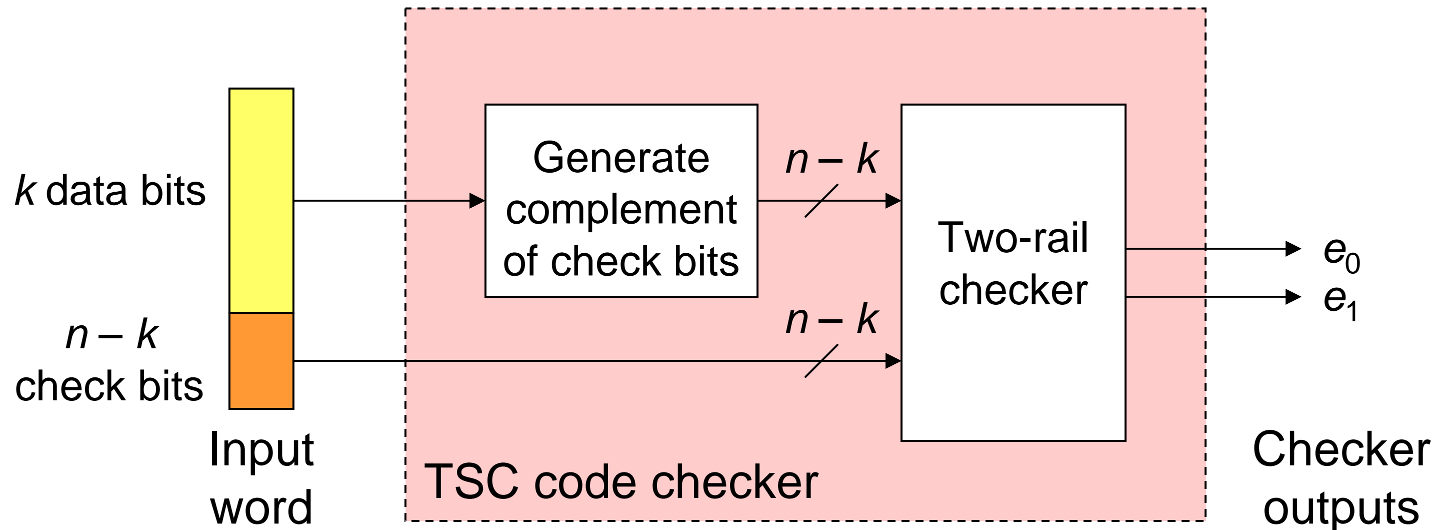
$$e_0 = \bigvee_{\substack{j=1 \\ (j \text{ even})}}^{m} (v \geq j)(w \geq m - j)$$



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

TSC Checkers for Separable Codes

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

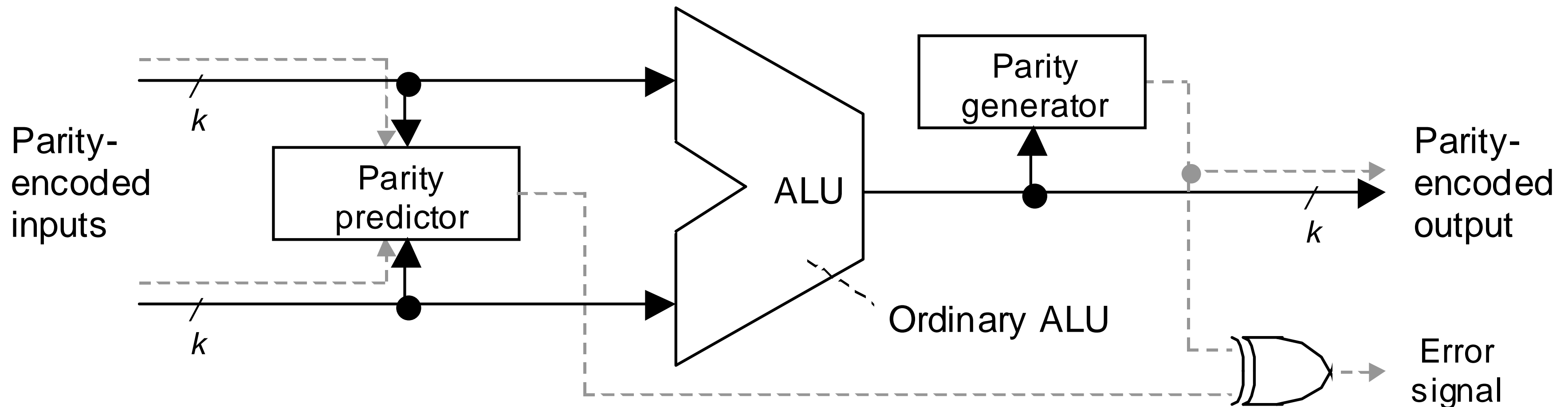


For many codes, direct synthesis will produce a faster and/or more compact totally-self-checking checker

Google search for “totally self checking checker” produces 442 hits

TSC Design with Parity Prediction

Recall our discussion of parity prediction as an alternative to duplication

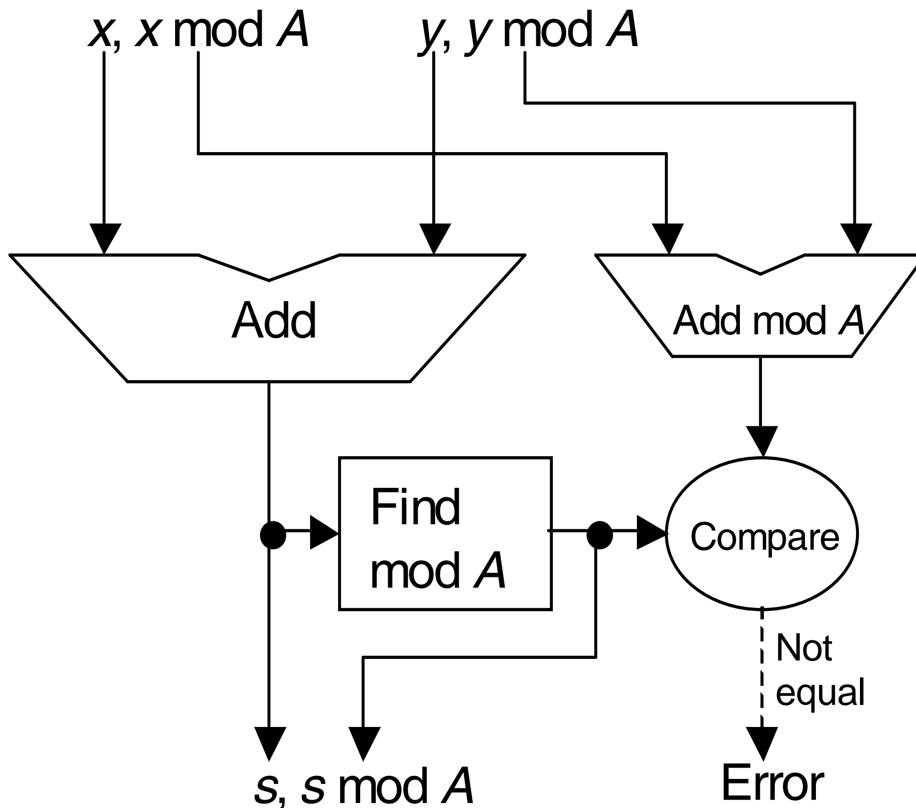


If the parity predictor produces the complement of the output parity, and the XOR gate is removed, we have a self-checking design

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

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)

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

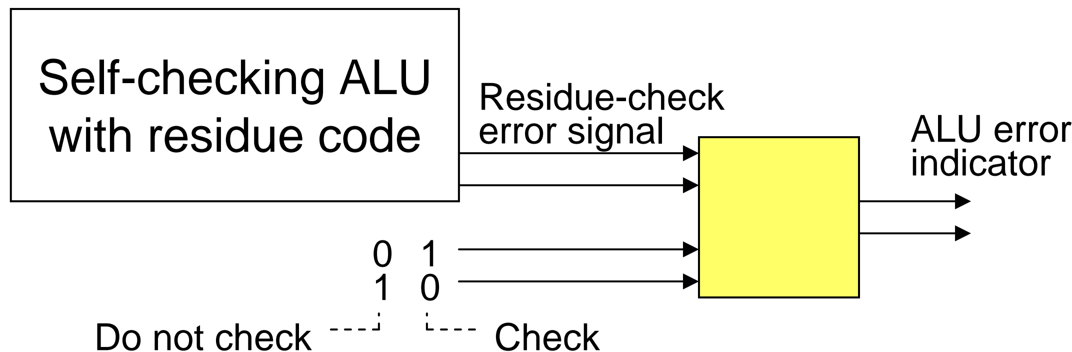
Optimal placement of checkers to satisfy these condition

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



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
Normal operation	G	D	G	10

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

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