Fault-Tolerant Computing

Hardware Design Methods
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

<table>
<thead>
<tr>
<th>Edition</th>
<th>Released</th>
<th>Revised</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>Oct. 2006</td>
<td>Nov. 2007</td>
</tr>
</tbody>
</table>
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

Legend:
- Entry
- Deviation
- Remedy
- Tolerance

Level → Component Logic Information System Service Result

Ideal → Defective → Faulty → Erroneous → Malfunctioning → Degraded → Failed

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

Nov. 2007 UCSB Self-Checking Modules
Main Ideas of Self-Checking Design

Function unit designed in a way that faults/errors/malfunctions 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

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>G</td>
</tr>
</tbody>
</table>

Circuit to combine error signals (two-rail checker)

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.

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).

Note that if we don't explicitly ensure this, tests for some of the faults may belong to the input error space.
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 $F_{\text{init}}$

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

Input
Code space

Error space

Output

f

0

1

Self-checking code checker

Input
Code space

Error space

Output

f

01

10

00

11

Example: 5-input odd-parity checker

s-a-0 fault on output?

e

Pleasant surprise: The self-checking version is simpler!
TSC Checker for $m$-out-of-$2m$ Code

Divide the $2m$ 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:

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

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

**Example:** 3-out-of-6 code checker, $m = 3$, $A = \{a, b, c\}$, $B = \{f, g, h\}$

- $e_0 = (v \geq 0)(w \geq 3) \lor (v \geq 2)(w \geq 1) = fgh \lor (ab \lor bc \lor ca)(f \lor g \lor h)$
- $e_1 = (v \geq 1)(w \geq 2) \lor (v \geq 3)(w \geq 0) = (a \lor b \lor c)(fg \lor gh \lor hf) \lor abc$

Always satisfied
Another TSC \( m \)-out-of-\( 2m \) Code Checker

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

\( m - 1 \) stages
Using 2-out-of-4 Checkers as Building Blocks

Building $m$-out-of-$2m$ TSC checkers, $3 \leq m \leq 6$, 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-$2m$ code
- Check the $m$-out-of-$2m$ 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

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:

1. **Input Word**
   - $k$ data bits
   - $n - k$ check bits

2. **TSC Code Checker**
   - Generate complement of check bits

3. **Two-rail Checker**
   - Outputs $e_0$ and $e_1$ for $n - k$ inputs

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.
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

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 conditions is 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.

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

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(x = 0)</td>
<td>(x = 1)</td>
</tr>
<tr>
<td>(A)</td>
<td>(C)</td>
<td>(A)</td>
</tr>
<tr>
<td>(B)</td>
<td>(D)</td>
<td>(C)</td>
</tr>
<tr>
<td>(C)</td>
<td>(B)</td>
<td>(D)</td>
</tr>
<tr>
<td>(D)</td>
<td>(C)</td>
<td>(A)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(x = 01)</td>
<td>(x = 10)</td>
</tr>
<tr>
<td>(A)</td>
<td>(0011)</td>
<td>(1010)</td>
</tr>
<tr>
<td>(B)</td>
<td>(0101)</td>
<td>(1001)</td>
</tr>
<tr>
<td>(C)</td>
<td>(1010)</td>
<td>(0101)</td>
</tr>
<tr>
<td>(D)</td>
<td>(1001)</td>
<td>(1010)</td>
</tr>
</tbody>
</table>