EEDG/CE 6303: Testing and Testable Design

Mehrdad Nourani

Dept. of ECE
Univ. of Texas at Dallas

Session 08

Memory Testing

Key Issues

- Motivation for testing memories
- Modeling memory chips
- Reduced functional fault models
- Traditional tests
- March tests
- Pseudorandom memory tests

Classical Tests

Functional RAM Chip Testing

- Purpose
  1. Cover traditional tests
     - Zero-One (MSCAN)
     - Checkerboard
     - GALPAT and Walking 1/0
  2. Cover tests for stuck-at, transition and coupling faults
     - MATS and MATS+
     - March C-
     - March A and March B
  3. Comparison of march tests

Fault Coverage of Tests

- When a test detects faults of a particular type, it detects:
  - *all subtypes* of that type; e.g., if it detects TFs it has to detect *all* <↑/0> *and* <↓/1> TFs
  - *all positions* of each subtype (addr. a-cell < or > v-cell)
- A *complete test* detects all faults it is designed for

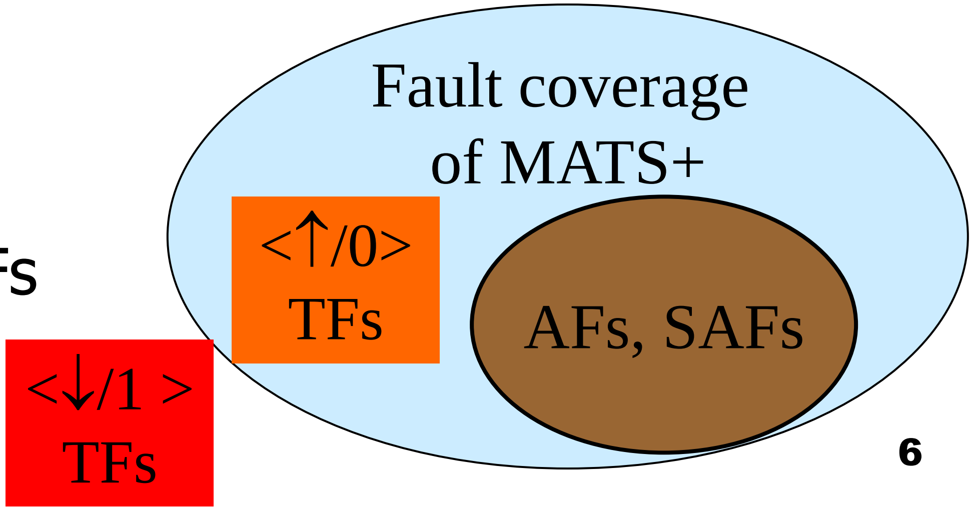
  It may, additionally, and unintentionally, detect also other faults

  But not all subtypes and not all positions of each of these faults

<u>Example</u>: MATS+ : {M0:⇕(w0); M1:⇑(r0,w1); M2:⇓(r1,w0)}

- Detects all AFs
- Detects all SAFs
- Detects all <↑/0> TFs
- Does not detect all <↓/1> TFs

⇒ MATS+ does **not** detect TFs



Traditional Tests

- Traditional tests are older tests
  - Usually developed without explicitly using fault models
  - Usually they also have a relatively long test time
  - Some have special properties in terms of:
    - detecting *dynamic faults*
    - *locating* (rather than only *detecting*) faults
- Many traditional tests exist:
  1. Zero-One (Usually referred to as *Scan Test* or *MSCAN*)
  2. Checkerboard
  3. GALPAT and Walking 1/0
  4. Sliding Diagonal
  5. Butterfly
  6. Surround Disturb
  7. Many, many others

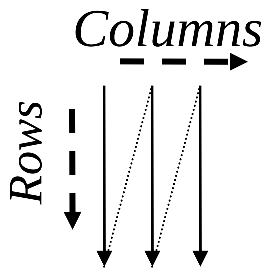
Zero-One Test (Scan Test, (M)SCAN)

- Minimal test, consisting of writing & reading 0s and 1s
  - **Step 1: write 0 in all cells**
  - **Step 2: read all cells**
  - **Step 3: write 1 in all cells**
  - **Step 4: read all cells**
- March notation for Scan test: $\{\Updownarrow(w0);\Updownarrow(r0);\Updownarrow(w1);\Updownarrow(r1)\}$
- Test length: $4n$ operations; which is $O(n)$

Zero-One Test (cont.)

- MSCAN: {M0:⇳(w0); M1:⇳(r0); M2:⇳(w1); M3:⇳(r1)}
- Fault detection capability: AFs not detected
  - *Condition AF* not satisfied: 1. ⇑(r*x*,…,w*x**) 2. ⇓(r*x**,…,w*x*). So, not all AFs are detected.
    - If address decoder maps all addresses to a *single cell*, then it can only be guaranteed that *one cell* is fault free
  - Not all TFs are detected. E.g. Not all <↓/1> TFs are detected because not all ↓ transitions are generated.
  - Not all CFs are detected because not all ↓ transitions are generated.
    - <↑;↑> CFids are not detected because in M3, the expected value is the same as the value induced by CFs.
    - <↑;↓> and <↑;↕> CFs are detected only if a-cell has lower address than v-cell (otherwise, w1 in M2 will mask the fault)
  - Special property: Stresses read/write & precharge circuits when *Fast X addressing is used and sequence of write/read 0101…. data in a column!*

Fast X addressing

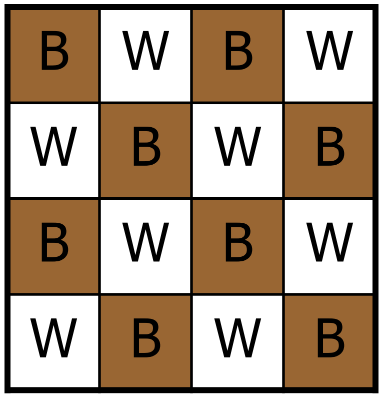


Row stripe	000000
	111111
	000000
	111111

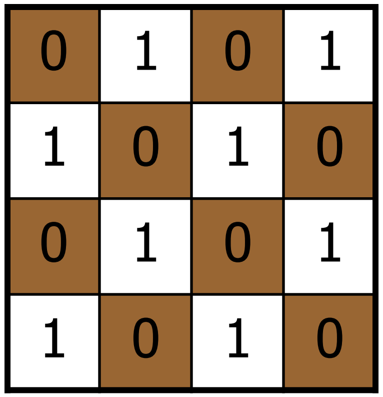
Checker board	010101
	101010
	010101
	101010

Checkerboard

- It is a SCAN test, using *checkerboard* data background pattern:
  - *Step 1*: w1 in all cells-W
    w0 in all cells-B
  - *Step 2*: read all cells
  - *Step 3*: w0 in all cells-W
    w1 in all cells-B
  - *Step 4*: read all cells



Checkerboard data background

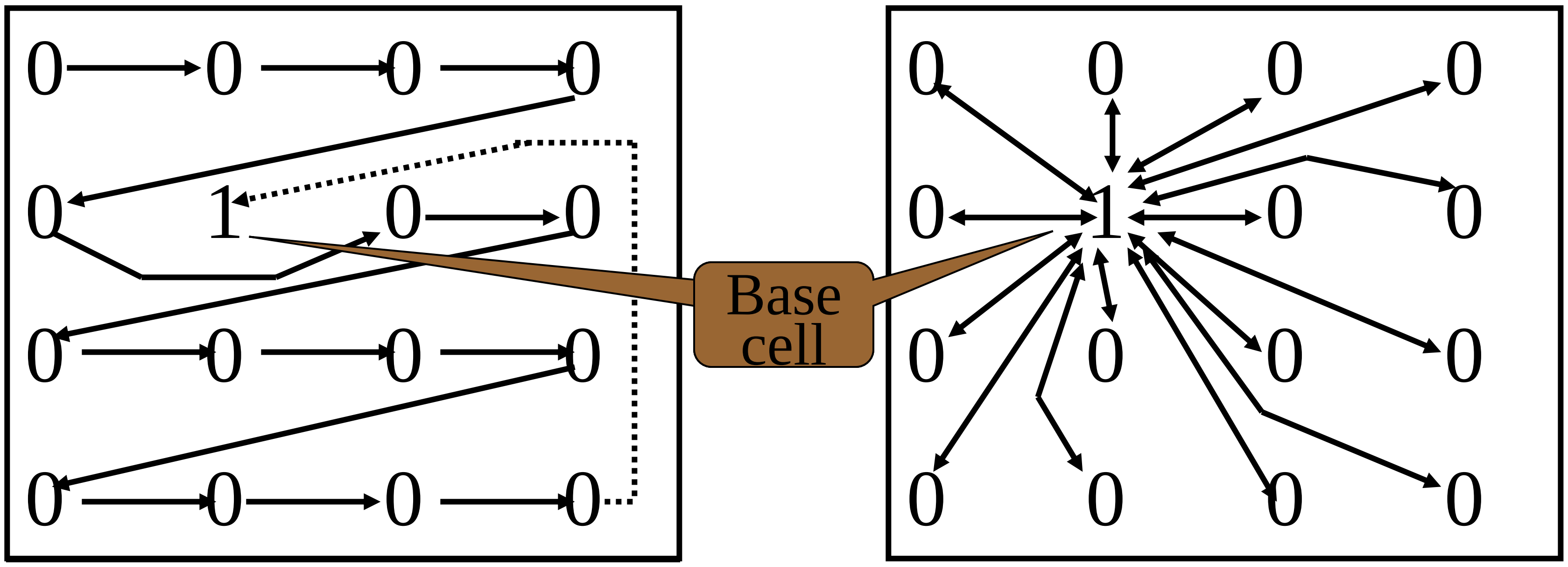


Step1 pattern

- Test length: 4n operations; which is O(*n*)
- Fault detection capability:
  - Condition *AF* not satisfied : 1. ⇑(r*x*,…,w*x**); 2. ⇓(r*x**,…,w*x*). So, not all AFs are detected.
    - If address decoder maps all cells-W to one cell, and all cells-B to another cell, then only 2 cells guaranteed fault free
  - SAFs are detected if it can be guaranteed (through other tests) that the address decoder functions correctly. Otherwise, only two cells can be guaranteed to be free of SAFs.
  - Similar to Zero-One test, not all TFs and CFs are detected.
  - *Special property: Maximizes leakage between physically adjacent cells. Used for DRAM retention test!!*

GALPAT and Walking 1/0

- GALPAT (GALloping PATterns) and Walking 1/0 are similar algorithms
  - They *walk* a *base-cell* through the memory
  - The memory cell is filled with 0s (or 1s) except the based cell which contains a 1 (or 0).
  - After each step of the base-cell, the contents of all other cells is verified, followed by verification of the base-cell
  - Difference between GALPAT and Walking 1/0 is when, and how often, the base-cell is read

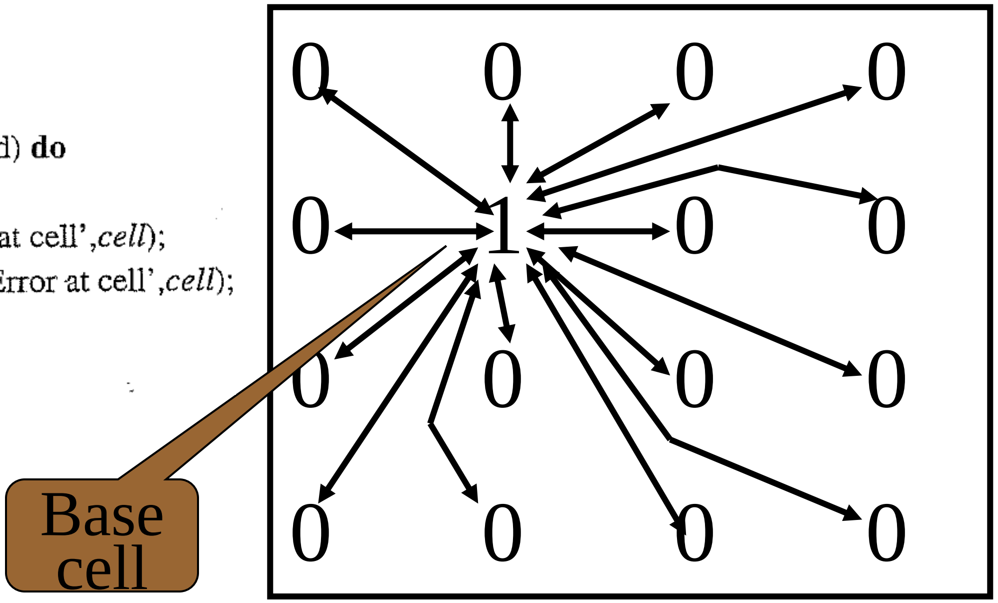


Walking 1/0

GALPAT

GALPAT Algorithm

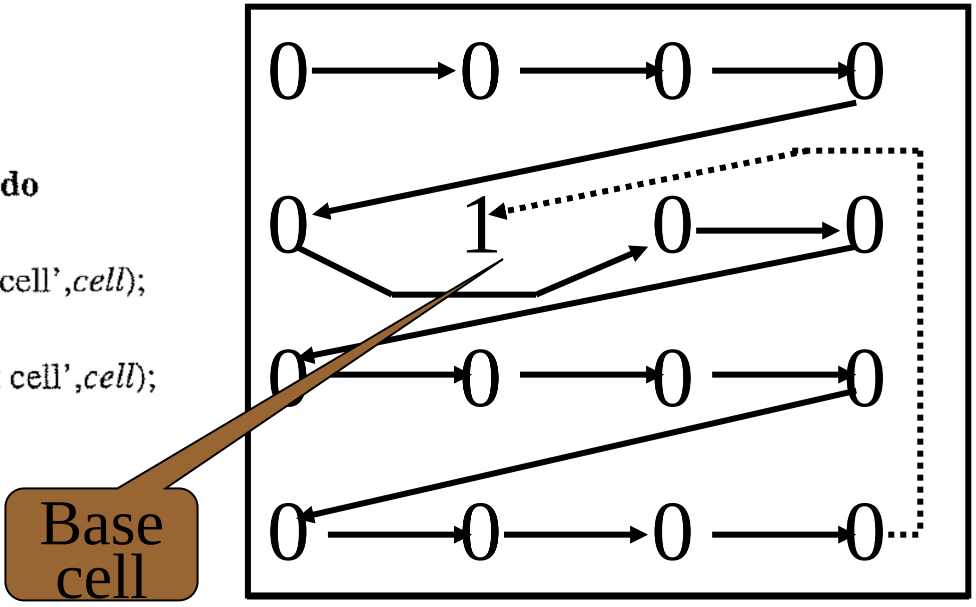
```
Step 1:  for $d$ := 0 to 1 do
         begin
             for $i$ := 0 to $n-1$ do
                 A[$i$] := $d$;
             for $base\text{-}cell$ := 0 to $n-1$ do
             begin
Step 2:          A[$base\text{-}cell$] := $\overline{d}$;
                 perform READ ACTION;
Step 3:          A[$base\text{-}cell$] := $d$;
             end;
         end;

         READ ACTION for GALPAT:
         begin
             for $cell$ := 0 to $n-1$ ($base\text{-}cell$ excluded) do
             begin
Step 4:          if (A[$cell$] $\neq d$) then output('Error at cell',$cell$);
Step 5:          if (A[$base\text{-}cell$] $\neq \overline{d}$) then output('Error at cell',$cell$);
             end;
         end;
```



GALPAT

Walking 1/0 Algorithm

```
Step 1:  for $d$ := 0 to 1 do
         begin
             for $i$ := 0 to $n-1$ do
                 A[$i$] := $d$;
             for $base\text{-}cell$ := 0 to $n-1$ do
             begin
Step 2:          A[$base\text{-}cell$] := $\overline{d}$;
                 perform READ ACTION;
Step 3:          A[$base\text{-}cell$] := $d$;
             end;
         end;
         READ ACTION for Walking 1/0:
         begin
             for $cell$ := 0 to $n-1$ ($base\text{-}cell$ excluded) do
             begin
Step 6:          if (A[$cell$] $\neq d$) then output('Error at cell',$cell$);
             end;
Step 7:      if (A[$base\text{-}cell$] $\neq \overline{d}$) then output('Error at cell',$cell$);
         end;
```



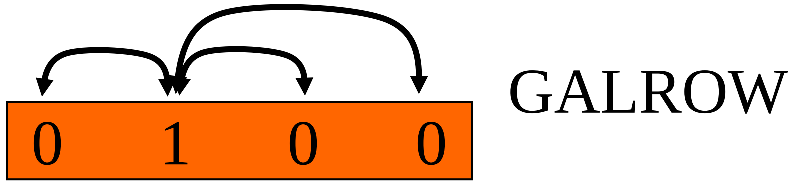
Walking 1/0

GALPAT and Walking 1/0: Properties

- All AFs are detected and *located*.
  - Step 5 (or 7) locates the problem if it is in the base-cell
  - Step 4 (or 6) locates the problem if it is in the other cells.
- All SAFs will be located because the base-cell is written (Step 2) and read (Step 5 and 7) with values 0 and 1.
- All TFs are located because the base-cell will make a ↑ and a ↓ transitions (Step 2) after which it is read (Step 5 and 7).
- CFids are located. In Step 2, <↑;↑> and <↓;↓> CFs may be sensitized (depending on the value of d in Step 1 to be 1 or 0, respectively) and located in Step 4 or Step 6. In Step 3, <↓;↑> and <↑;↓> CFids may be sensitized and located in Step 4 or Step 6.

GALPAT and Walking 1/0: Properties (cont.)

- GALPAT detects *write recovery faults* (*Cause*: slow addr. decoders)
- Test length: O(n^2): Not acceptable for practical purposes
- Most coupling faults in a memory are due to sharing
  - a WL and the column decoder: cells in the *same row*
  - BLs and row decoder: cells in the *same column*
- Subsets of GALPAT and Walking I/O used (*BC*= *B*ase-*C*ell)
  - GALROW and WalkROW: Read Action on cells in row of BC



  - GALCOL and WalkCOL: Read Action on cells in column of BC

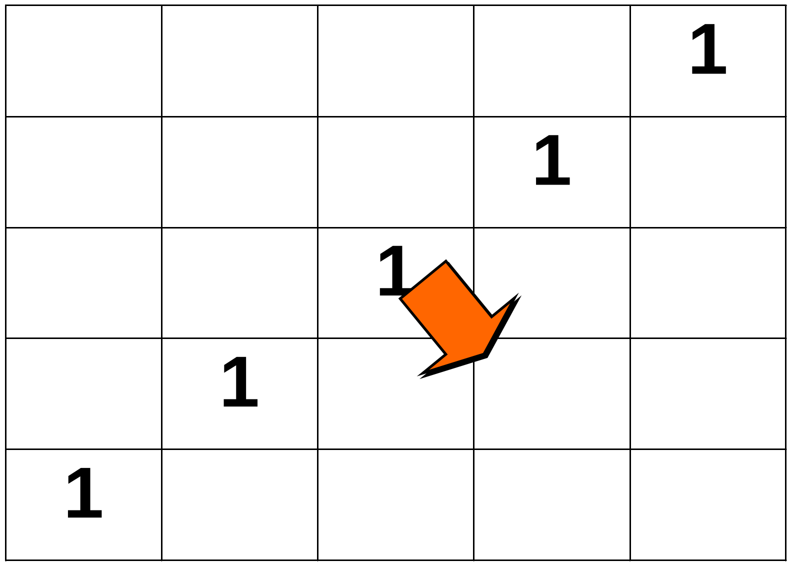
  Test length (assuming $n^{1/2}$ rows and $n^{1/2}$ columns): O($n^{3/2}$)

Note: Test time for 4Mb 150 ns memory

  - For O(n^2) test = O(20 days), and for O($n^{3/2}$) test = O(14 sec.)

Sliding Diagonal Algorithm

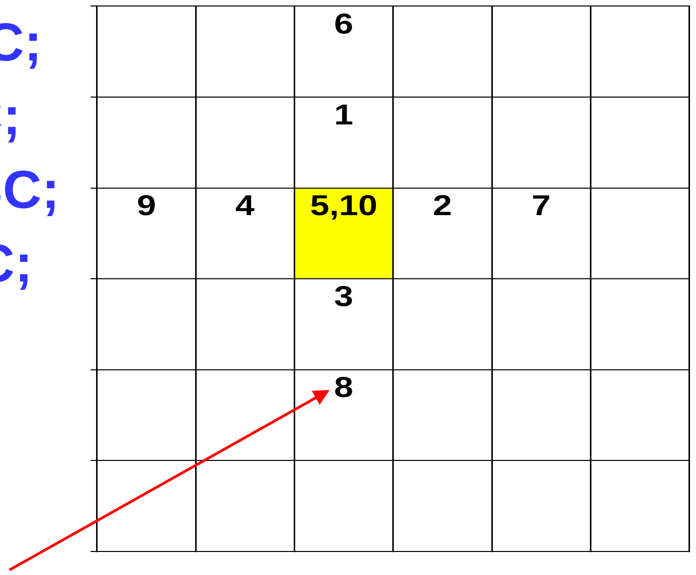
- Sliding (Galloping) Row/Column/Diagonal
  - Based on GALPAT, but instead of shifting a 1 through the memory, a complete diagonal of 1s is shifted:
    - The whole memory is read after each shift
  - Detects all faults as GALPAT, except for some CFs
  - Complexity is $4n^{1.5}$.



Butterfly Algorithm

- Butterfly Algorithm
  - Complexity is 5nlogn
  - All SAFs and some AFs are detected

```
1. Write background 0;
2. For BC = 0 to n-1
   { Complement BC; dist = 1;
     While dist <= mdist     /* mdist < 0.5 col/row length */
       {  Read cell @ dist north from BC;
          Read cell @ dist east from BC;
          Read cell @ dist south from BC;
          Read cell @ dist west from BC;
          Read BC;  dist *= 2; }
     Complement BC; }
3. Write background 1;  repeat Step 2;
```

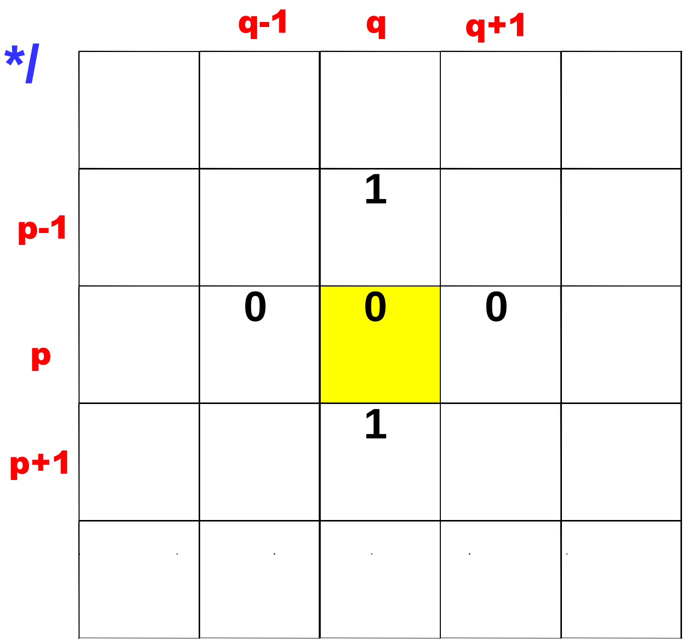


Numbers show order of read in the "while" loop.

Surround Disturb Algorithm

- Surround Disturb (SD) Algorithm
  - Examine how the cells in a row are affected when complementary data are written into adjacent cells of neighboring rows.
  - Designed on the premise that DRAM cells are most susceptible to interference from their nearest neighbors (eliminates global sensitivity checks).

```
1. For each cell[p,q]   /* row p and column q */
   { Write 0 in cell[p,q-1];
     Write 0 in cell[p,q];
     Write 0 in cell[p,q+1];
     Write 1 in cell[p-1,q];
     Read 0 from cell[p,q+1];
     Write 1 in cell[p+1,q];
     Read 0 from cell[p,q-1];
     Read 0 from cell[p,q]; }
2. Repeat Step 1 with complementary data;
```



Moving Inversion Algorithm

- Moving Inversion (MOVI) Algorithm
  - For functional and AC parametric test
    - Functional (13n): for AF, SAF, TF, and most CF

      $\{\Downarrow(w0); \Uparrow(r0, w1, r1); \Uparrow(r1, w0, r0); \Downarrow(r0, w1, r1); \Downarrow(r1, w0, r0)\}$
    - Parametric (12nlogn): for Read access time
      - 2 successive Reads @ 2 different addresses with different data for all 2-address sequences differing in 1 bit
      - Repeat M2~M5 for each address bit
      - GALPAT---all 2-address sequences

March Tests

March Tests

- The simplest, and most efficient tests for detecting AFs, SAFs, TFs and CFs are *march tests*
- The following march tests are covered:
  - MATS+
    - Detects AFs and SAFs
  - March C-
    - Detects AFs, SAFs, TFs, and unlinked CFins, CFsts, CFids
  - March A
    - Detects AFs, SAFs, TFs, CFins, CFsts, CFids, linked CFids (but not linked with TFs)
  - March B
    - Detects AFs, SAFs, TFs, CFins, CFsts, CFids, linked CFids

History of MATS Tests

- ATS: Algorithmic Test Sequence
  - By Knaizuk and Hartmann (1977)
  - It requires 4×2^n memory accesses.
  - Notations:
    - Addresses:

Let A_μ be the memory address μ,

$$0 \le \mu < 2^n.$$

Let

$$\pi_0 = \{A_\mu | \mu \equiv 0 (\text{modulo } 3)\},$$
$$\pi_1 = \{A_\mu | \mu \equiv 1 (\text{modulo } 3)\},$$
$$\pi_2 = \{A_\mu | \mu \equiv 2 (\text{modulo } 3)\}.$$

- Tabulated algorithm
  - Wr: Write
  - R: Read

Algorithm

Step 1: Write the all 0 word, W_0, at all locations

$$A_j \in \pi_1 \text{ and } A_k \in \pi_2.$$

Step 2: Write the all 1 word, W_1, at all locations

$$A_i \in \pi_0.$$

Step 3: Read all locations $A_j \in \pi_1$:

$$\text{if output} \begin{cases} = W_0; & \text{no fault indicated;} \\ \ne W_0; & \text{RAM fault indicated.} \end{cases}$$

Step 4: Write the all 1 word W_1 at all locations

$$A_j \in \pi_1.$$

Step 5: Read all locations $A_k \in \pi_2$:

$$\text{if output} \begin{cases} = W_0; & \text{no fault indicated;} \\ \ne W_0; & \text{RAM fault indicated.} \end{cases}$$

Step 6: Read all locations $A_i \in \pi_0$ and $A_j \in \pi_1$:

$$\text{if output} \begin{cases} = W_1; & \text{no fault indicated;} \\ \ne W_1; & \text{RAM fault indicated.} \end{cases}$$

Step 7: Write and then read the all 0 word W_0 at all locations

$$A_i \in \pi_0.$$

$$\text{If output} \begin{cases} = W_0; & \text{no fault indicated;} \\ \ne W_0; & \text{RAM fault indicated.} \end{cases}$$

Step 8: Write and then read the all 1 word W_1 at all locations

$$A_k \in \pi_2.$$

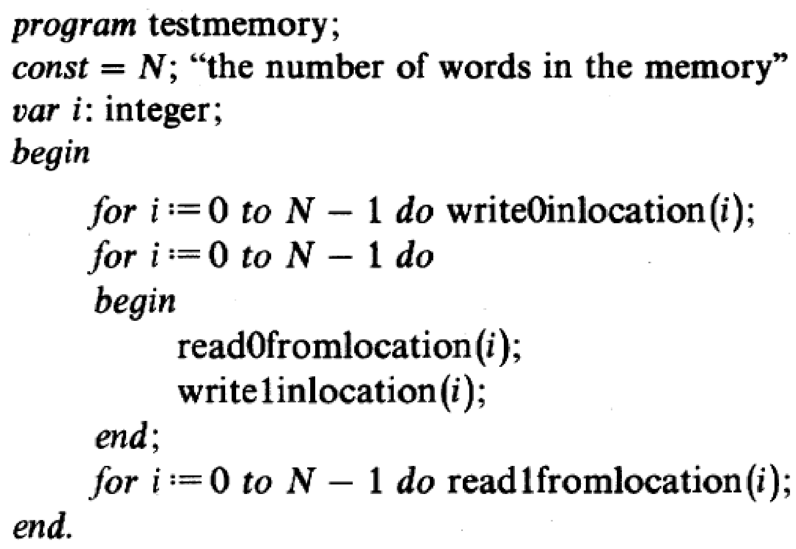
$$\text{If output} \begin{cases} = W_1; & \text{no fault indicated;} \\ \ne W_1; & \text{RAM fault indicated.} \end{cases}$$

END.

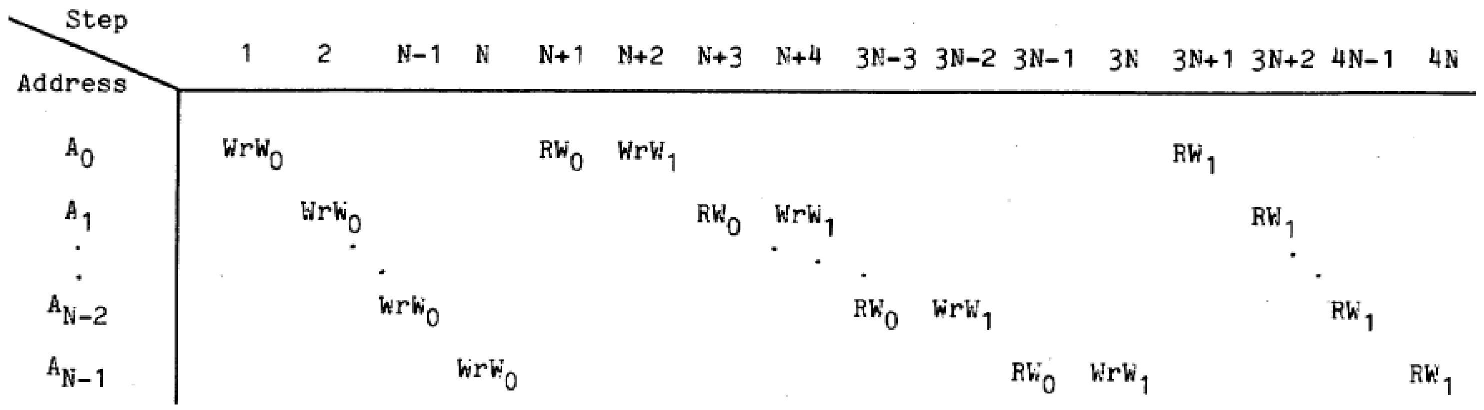
Step / Partition	1	2	3	4	5	6	7	8
π_0		$Wr\ W_1$				$R\ W_1$	$Wr\ W_0$, $R\ W_0$	
π_1	$Wr\ W_0$		$R\ W_0$	$Wr\ W_1$		$R\ W_1$		
π_2	$Wr\ W_0$				$R\ W_0$			$Wr\ W_1$, $R\ W_1$

History of MATS Tests (cont.)

- MATS: Modified Algorithmic Test Sequence
  - By Nair (1979)
  - MATS {⇑(w0);⇑(r0,w1); ⇑(r1)}
  - Complexity is 4n.
  - Tabulated algorithm
    - Wr: Write
    - R: Read



```
program testmemory;
const = N; "the number of words in the memory"
var i: integer;
begin
      for i := 0 to N − 1 do write0inlocation(i);
      for i := 0 to N − 1 do
      begin
            read0fromlocation(i);
            write1inlocation(i);
      end;
      for i := 0 to N − 1 do read1fromlocation(i);
end.
```



Step / Address	1	2	N-1	N	N+1	N+2	N+3	N+4	3N-3	3N-2	3N-1	3N	3N+1	3N+2	4N-1	4N
A_0	WrW_0				RW_0	WrW_1							RW_1			
A_1		WrW_0					RW_0	WrW_1						RW_1		
A_{N-2}			WrW_0						RW_0	WrW_1					RW_1	
A_{N-1}				WrW_0							RW_0	WrW_1				RW_1

MATS+

- MATS+ algorithm: {M0:⇕(w0); M1:⇑(r0,w1); M2:⇓(r1,w0)}
- Fault coverage
  - AFs detected because MATS+ satisfies *Cond. AF*
    (When reads, accessing multiple cells, return a *random* value)
    Cond. AF: **1. ⇑(r*x*,...,w*x**) and 2. ⇓(r*x**,...,w*x*)**
    (1) satisfied by: M1:⇑(*r0,w1*) and (2) by: M2:⇓(*r1,w0*)
  - SAFs are detected: from each cell the value 0 and 1 is read
  - SAFs on Read/Write logic will be detected as both 0 and 1 are written and read.
- Test length: 5*n*

Note: If fault model is *symmetric* with respect to 0/1, ↑/↓, **⇑/⇓**

and with respect to address a-cell < v-cell and address a-cell > v-cell,

then **each march tests has 3 equivalent tests**

0s ⇔ 1s: {⇕(w0);⇑(r0,w1);⇓(r1,w0)}⇒ {⇕(w1);⇑(r1,w0);⇓(r0,w1)}

⇑s ⇔ ⇓s: {⇕(w0);⇑(r0,w1);⇓(r1,w0)}⇒ {⇕(w0);⇓(r0,w1);⇑(r1,w0)}

0s⇔1s,⇑s⇔⇓s: {⇕(w0);⇑(r0,w1);⇓(r1,w0)}⇒{⇕(w1);⇓(r1,w0);⇑(r0,w1)}

March C-

- March C (Marinescu,1982): an 11*n* algorithm

  {⇕(w0);⇑(r0,w1);⇑(r1,w0);**⇕(r0);**⇓(r0,w1);⇓(r1,w0);⇕(r0)}

  It can be shown that middle '⇕(r0)' march element is *redundant*
- March C- (van de Goor,1991): a 10*n* algorithm

  {⇕(w0);⇑(r0,w1);⇑(r1,w0);⇓(r0,w1);⇓(r1,w0);⇕(r0)}

  M0 M1 M2 M3 M4 M5

- Fault coverage of March C- (Summary)
  - AFs: *Cond. AF* satisfied by M1 and M4, or by M2 and M3
  - SAFs: Detected by M1 (SA1 faults) and M2 (SA0 faults)
  - TFs: <↑/0> TFs sensitized by M1, detected by M2 (and M3+M4)

    <↓/1> TFs sensitized by M2, detected by M3 (and M4+M5)
  - CFins (<↑;↕>, <↓;↕>) detected
  - CFsts (<1;0>, <1;1>, <0;0>, <0;1>) detected
  - CFids (<↑;0>=<↑;↓>, <↑;1>=<↑;↑>, <↓;0>=<↓;↓>, <↓;1>=<↓;↑>) detected

March C- Detects SAF, TF, AF

March C-: {⇕(w0);⇑(r0,w1);⇑(r1,w0); ⇓(r0,w1);⇓(r1,w0);⇕(r0)}

M0 M1 M2 M3 M4 M5

Fault	Condition	Sensitizing	Detection	Comments
SAF <∀/0>		M1 (when operating on a cell, it tries to write 1)	M2 (when operating on a cell, it reads and expects 1)	M3+M4 also sensitizes and detects SA0.
SAF <∀/1>		M0 (when operating on a cell, it tries to write 0)	M1 (when operating on a cell, it reads and expects 0)	M2+M3 also sensitizes and detects SA1.
TF <↑/0>		M1	M2	M3+M4 together do the same.
TF <↓/1>		M2	M3	M4+M5 together do the same.
AFs				M1+M4 together satisfy Condition for detecting AFs M2+M3 together satisfy Condition for detecting AFs

March C- Detects CFids

March C-: {⇕(w0);⇑(r0,w1);⇑(r1,w0); ⇓(r0,w1);⇓(r1,w0);⇕(r0)}
M0 M1 M2 M3 M4 M5

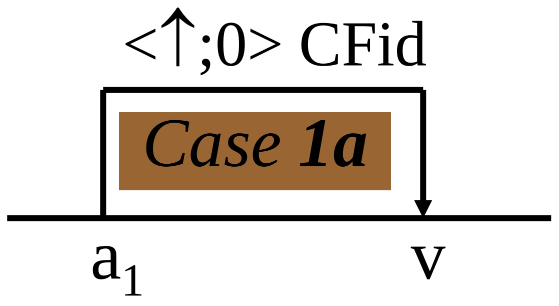
Proof for detecting CFs are all similar. **Analyze all cases**:

- Relative positions of a-cell and v-cell

  1. address of a-cell < v-cell;

  2. address of a-cell > v-cell
- Fault subtype

  a. CFid <↑;0>; ***b***. CFid <↑;1>; ***c***. CFid <↓;0>; ***d***. CFid<↓;1>



Consider *Case* ***1a***: a-cell < v-cell and CFid <↑;0>

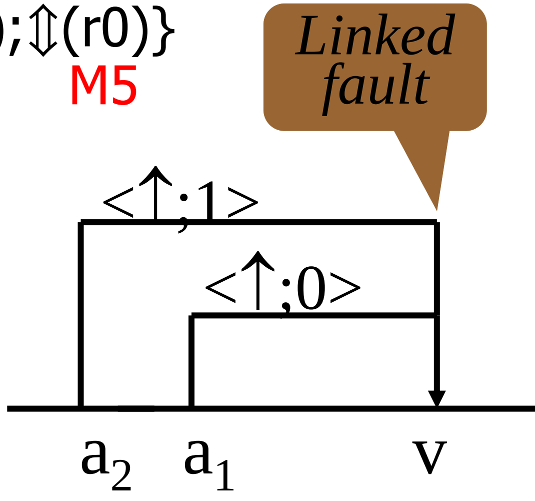
- Fault sensitized by M3 and detected by M4
- Other cases need to be argued similarly one by one.
- Can be summarized like this:

Fault	Condition	Sensitizing	Detection	Comments
<↑;0>	a-cell < v-cell	M3 (when operating on a changes v to faulty value [v=0])	M4 (when operating on v expecting 1 but reading 0)	

March C- Cannot Detect Linked CFids

March C-: $\{\Updownarrow(w0);\Uparrow(r0,w1);\Uparrow(r1,w0);\Downarrow(r0,w1);\Downarrow(r1,w0);\Updownarrow(r0)\}$

M0 M1 M2 M3 M4 M5



- If the CFid $<\uparrow;0>a_1$ (a-cell is a_1) is linked to CFid $<\uparrow;1>a_2$, **and** address of $a_2 < a_1$ then linked fault will *not* be detected

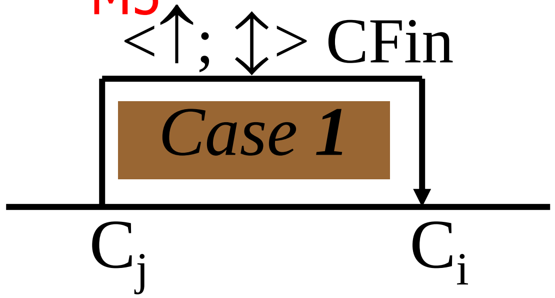
  Reason: M3 will sensitize both faults, such that masking occurs

- Can be summarized like this:

Fault	**Condition**	**Sensitizing**	**Detection**	**Comments**
$<\uparrow;0>a_1$ linked to $<\uparrow;1>a_2$	a_2-cell<a_1-cell<v-cell	M3 (when operating on a_1 changes v to faulty value [v=0], but when operating on a_2 changes it back to fault-free value [v=1])	None	It cannot be detected

March C- Detects Unlinked CFin

March C-: {⇕(w0);⇑(r0,w1);⇑(r1,w0); ⇓(r0,w1);⇓(r1,w0);⇕(r0)}
M0 M1 M2 M3 M4 M5



- Case 1: j<i
  - Let C_i be coupled to any number of cells with addresses lower than i and let C_j be the highest of those cells (j<i)
    - (a) C_i is <↑; ↕> coupled to C_j; then M1 will sensitize and detect the fault, as well as M3 followed by M4.
    - (b) C_i is < ↓; ↕> coupled to C_j; then M2 will sensitize and detect the fault, as well as M4 followed by M5. In summary:

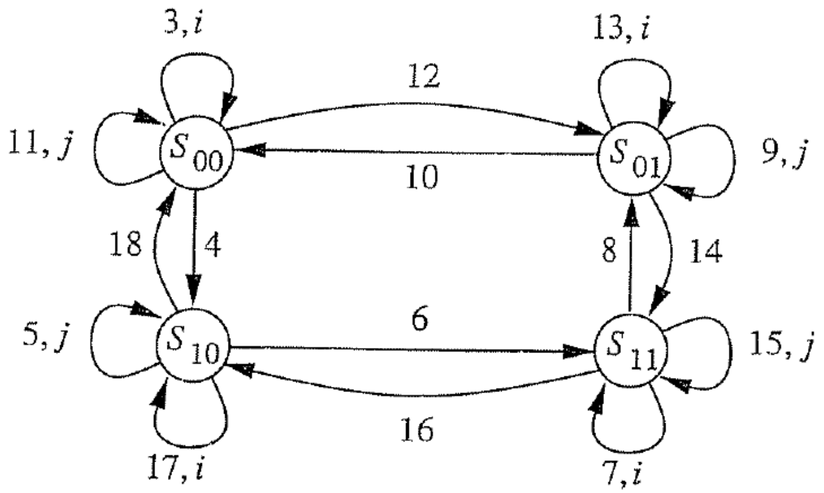
Fault	**Condition**	**Sensitizing**	**Detection**	**Comments**
<↑; ↕>	a-cell (C_j)<v-cell (C_i)	M1 (when operating on C_j changes C_i to faulty value [C_i=1]	M1 (when operating on C_i expecting 0 but reading 1)	Similarly M3 can sensitize the fault and M4 can detect it
<↓; ↕>	a-cell (C_j)<v-cell (C_i)	M2 (when operating on C_j changes C_i to faulty value [C_i=0]	M2 (when operating on C_i expecting 1 but reading 0)	Similarly M4 can sensitize the fault and M5 can detect it

- Case 2: j>i
  - The proof is similar to Case 1.

March C- Detects State Coupling Faults

March C-:{⇕(w0);⇑(r0,w1);⇑(r1,w0);⇓(r0,w1);⇓(r1,w0);⇕(r0)}
M0 M1 M2 M3 M4 M5

- All CFst are detected as the four states of any two cells i and j are reached.
  - Example here assumes i<j (C_i is victim)
  - All 4 states are generated and verified because in each state the values of cell C_i and C_j are read. For example, in state S_{00} cell C_i is read in Step 3 and cell C_j in step 11.



Step	March element	State S_{ij} before operation	Operation	State S_{ij} after operation
1	M_0	–	w0 into *i*	–
2		–	w0 into *j*	S_{00}
3	M_1	S_{00}	r0 from *i*	S_{00}
4		S_{00}	w1 into *i*	S_{10}
5		S_{10}	r0 from *j*	S_{10}
6		S_{10}	w1 into *j*	S_{11}
7	M_2	S_{11}	r1 from *i*	S_{11}
8		S_{11}	w0 into *i*	S_{01}
9		S_{01}	r1 from *j*	S_{01}
10		S_{01}	w0 into *j*	S_{00}
11	M_3	S_{00}	r0 from *j*	S_{00}
12		S_{00}	w1 into *j*	S_{01}
13		S_{01}	r0 from *i*	S_{01}
14		S_{01}	w1 into *i*	S_{11}
15	M_4	S_{11}	r1 from *j*	S_{11}
16		S_{11}	w0 into *j*	S_{10}
17		S_{10}	r1 from *i*	S_{10}
18		S_{10}	w0 into *i*	S_{00}

March A & March B

- March A algorithm (Suk,1981)
  {⇕(w0);⇑(r0,w1,w0,w1);⇑(r1,w0,w1);⇓(r1,w0,w1,w0);⇓(r0,w1,w0)}
  M0 M1 M2 M3 M4
- March A (Test length: 15**n)* detects
  - AFs, SAFs, TFs, CFins, CFsts, CFids
  - Linked CFids, but **not linked with TFs**. For example, M1 detects <↑;↑> linked with <↓;↓> - Odd number of transitions prevents masking.
  - March A is *complete*: detects all intended faults
  - March A is *irredundant*: no operation can be removed
- March B algorithm (Test length: 17**n*)

{⇕(w0);⇑(r0,w1,r1,w0,r0,w1);⇑(r1,w0,w1);⇓(r1,w0,w1,w0);⇓(r0,w1,w0)}
M0 M1 M2 M3 M4

- Detects all faults of March A
- Detects **CFids linked with TFs**, because M1 detects *all* TFs (e.g. <↑/0> or <↓/1>)

March A Detects CFin

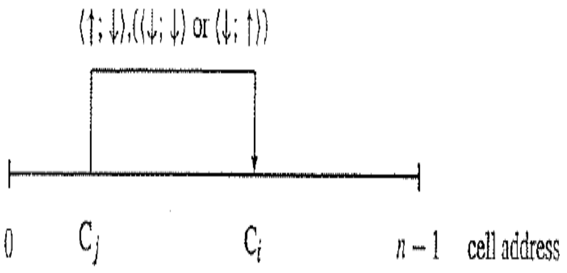
- March A algorithm (Suk,1981)
  {⇕(w0);⇑(r0,w1,w0,w1);⇑(r1,w0,w1);⇓(r1,w0,w1,w0);⇓(r0,w1,w0)}
  M0 M1 M2 M3 M4

- Case 1: j<i (j: aggressor's address and i for victim's address)
  - Let C_i be coupled to any odd number of cells with addresses lower than i and let C_j be the highest of those cells (j<i)
    - (a) C_i is <↑; ↕> coupled to C_j; then M2 will sensitize and detect the fault.
    - (b) C_i is <↓; ↕> coupled to C_j; then M1 or M2 will sensitize and detect the fault.
    - (c) C_i is <↑; ↕> and <↓; ↕> coupled to C_j; then M1 will sensitize and detect the fault.

- Case 2: j>i
  - The proof is similar to Case 1.
  - (a) will be sensitized and detected by M3, (b) by M4 and (c) by M3.

March A Detects CFid

- March A algorithm (Suk,1981)
  {⇕(w0);⇑(r0,w1,w0,w1);⇑(r1,w0,w1);⇓(r1,w0,w1,w0);⇓(r0,w1,w0)}
  M0 M1 M2 M3 M4
- The proof that March A can detect linked CFids and certain CFins linked with CFids follows.
- Case 1: j<i (j: aggressor's address and i for victim's address)
  — Let C_i be coupled to any number of cells with addresses lower than i and let C_j be the highest of those cells (j<i). There are 4 cases (for 4 CFids):
    – (a) C_i is <↑;↓> coupled to C_j (and possibly **also** <↓;↓> or <↓;↑> coupled to C_j); then M2 will detect the <↑;↓> fault because:



Fault: CFid <↑;↓> (unlinked)

M2 oper. on C_j/Ci	C_j	C_i	Comment
r1 (C_j)	1	1	Initial state
w0 (C_j)	0	1	
w1 (C_j)	1	0	
r1 (C_i)	1	0	<↑; ↓> detected

When M2 operates on C_i, the fault <↑;↓> will be detected

Faults: Linked (<↑;↓> , <↓;↓>)

M2 oper. on C_j/Ci	C_j	C_i	Comment
r1 (C_j)	1	1	Initial state
w0 (C_j)	0	0	
w1 (C_j)	1	0	
r1 (C_i)	1	0	Linked faults detected

When M2 operates on C_i, the linked faults will be detected

Faults: Linked (<↑;↓> , <↓;↑>)

M2 oper. on C_j/Ci	C_j	C_i	Comment
r1 (C_j)	1	1	Initial state
w0 (C_j)	0	1	
w1 (C_j)	1	0	
r1 (C_i)	1	0	Linked faults detected

When M2 operates on Ci, the linked faults will be detected

March A Detects CFid (cont.)

- March A algorithm (Suk,1981)
  {⇕(w0);⇑(r0,w1,w0,w1);⇑(r1,w0,w1);⇓(r1,w0,w1,w0);⇓(r0,w1,w0)}
  M0 M1 M2 M3 M4
- Case 1: j<i (cont.)
  - (b) C_i is <↑;↑> coupled to C_j (and possibly **also** <↓;↓> or <↓;↑> coupled to C_j); then M1 will detect the <↑;↑> fault.
  - (c) C_i is <↓;↓> coupled to C_j (and **not** <↑;↓> or <↑;↑> coupled to C_j); then M2 will detect the <↓;↓> fault.
  - (d) C_i is <↓;↑> coupled to C_j (and **not** <↑;↓> or <↑;↑> coupled to C_j); then M1 will detect the <↓;↑> fault.
- Case 2: (j>i)
  - The proof is similar to Case 1, using M3 and M4 instead of M1 and M2.

March B

- March A algorithm (Test length: 15n)
  $\{\Updownarrow(w0);\Uparrow\mathbf{(r0,w1,w0,w1);}\Uparrow(r1,w0,w1);\Downarrow(r1,w0,w1,w0);\Downarrow(r0,w1,w0)\}$
  M0 M1 M2 M3 M4
- March B algorithm (Test length: 17*n*)

$\{\Updownarrow(w0);\Uparrow\mathbf{(r0,w1,r1,w0,r0,w1);}\Uparrow(r1,w0,w1);\Downarrow(r1,w0,w1,w0);\Downarrow(r0,w1,w0)\}$
M0 M1 M2 M3 M4

  - Detects all faults of March A
  - Detecting SoPF (open fault) required (…,r*x*,…,r*x**). M1 satisfies this condition.
  - Detects **CFids linked with TFs**, because M1 detects *all* TFs.
  - Two extra reads in M1 are intended to prevent TFs to be masked by CFs because no write operations to other cells, which may be potential coupling cells, take place.
    - For TF<↑/0>: The first w1 stimulates it and r1 will detect it.
    - For TF<↓/1>: w0 stimulates it and r0 will detect it.

Other March Tests

- Marching 1/0:

$\{\Uparrow(w0); \Uparrow(r0,w1,r1); \Downarrow(r1,w0,r0); \Uparrow(w1); \Uparrow(r1,w0,r0); \Downarrow(r0,w1,r1)\}$

- MATS++:

$\{\Updownarrow(w0); \Uparrow(r0,w1); \Downarrow(r1,w0,r0)\}$

- March X:

$\{\Updownarrow(w0); \Uparrow(r0,w1); \Downarrow(r1,w0); \Updownarrow(r0)\}$

- March Y:

$\{\Updownarrow(w0); \Uparrow(r0,w1,r1); \Downarrow(r1,w0,r0); \Updownarrow(r0)\}$

Test Requirements for Detecting SOpFs

- An SOpF is caused by an open WL which makes the cell inaccessible
- To detect SOpFs, assuming a non-transparent sense amplifier, a march test has to verify that a 0 and a 1 has to be read from every cell.
- This will be the case when the march test contains the March Element 'ME' of the form: (…, r*x*, …, r*x**, …), for *x* = 0 and *x* = 1.

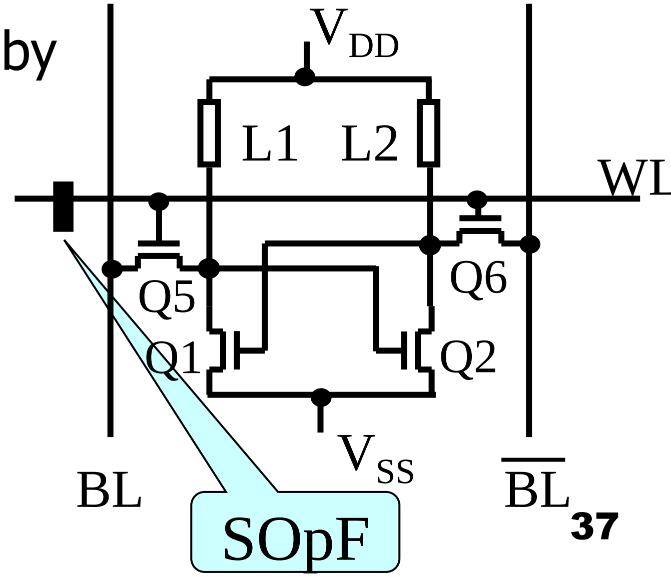
  Example: The ME "⇑(r0,w1,r1,w0,r0,w1)" satisfies the above requirement

  — This ME may be broken down into two MEs of the form: (…,r*x*,…) & (…, r*x**,…), for *x* = 0 and *x* = 1.

  Example: Two MEs "⇑(r0,w1,r1); ⇓(r1,w0,r0)" satisfy the above requirement

Note: Any test can be changed to detect SOpFs by making sure that the above requirement is satisfied by possibly adding a *rx* and/or a r*x** operation to a ME

Example: MATS+ {⇕(w0);⇑(r0,w1);⇓(r1,w0)} becomes {⇕(w0);⇑(r0,w1,r1); ⇓(r1,w0,r0)}

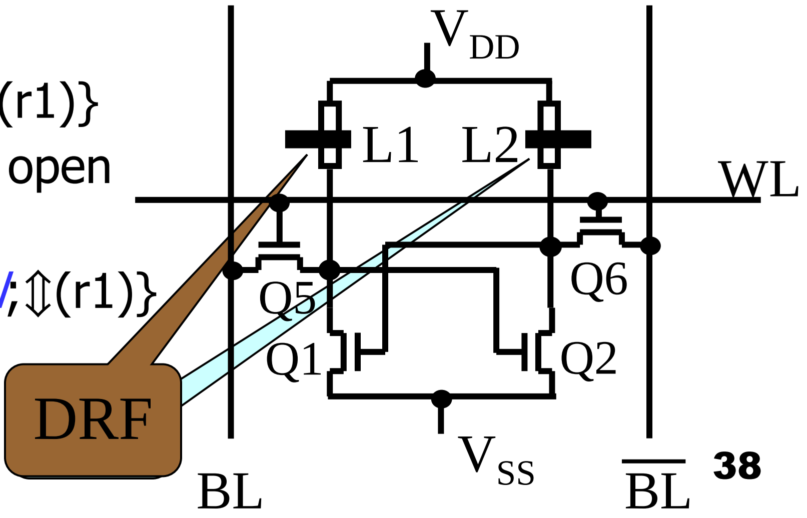


Test Requirements for Detecting DRFs

Any march test can be extended to detect DRFs

1. Every cell has to be brought into one state
2. A time period (*Del*) has to be waited for the fault to develop
   - Note: The time for *Del* is typically between 100 and 500 ms
3. The cell contents has to be verified (should not be changed)

- Above three steps to be done for both states of every cell
  - Example: MATS+ {⇕(w0);⇑(r0,w1);⇓(r1,w0)} becomes {⇕(w0);*Del*;⇑(r0,w1);*Del*;⇓(r1,w0)}
  - Example: Assuming the existing test ends with all cells in state 0: {Existing March Test;*Del*;⇕(r0,w1);*Del*;⇕(r1)}
  - Example: If both pull-up devices may be open DRF behaves like an SOpF: {Existing March Test;*Del*;⇕(r0,w1,r1);*Del*;⇕(r1)}

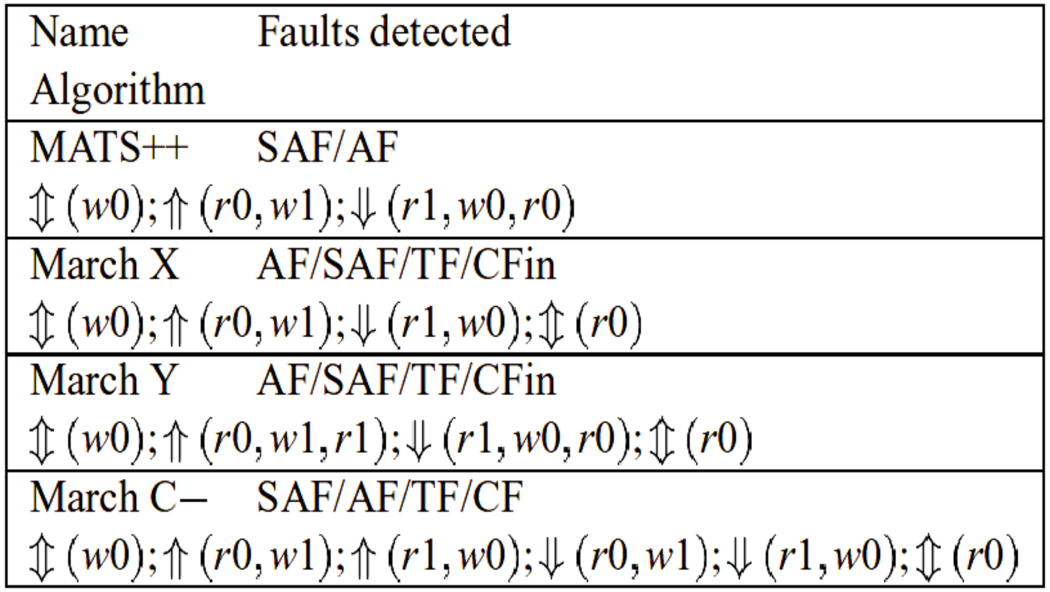


How to Analyze March Tests

- For each fault in the following list, find out:
  1. which element(s) sensitizes the fault
  2. which element(s) detects the fault

  (For coupling faults (victim C_i and aggressor C_j), consider two cases i<j and i>j).
- List of faults include:
  - Stuck at faults [**2** cases]: SA0 (<∀/0>), SA1 (<∀/1>)
  - Transition faults [**2**]: <↑/0>, <↓/1>
  - Inversion coupling faults [**2**]: <↑;↕>, <↓;↕>
  - Idempotent coupling faults [**4**]: <↑;↓>, <↑;↑>, <↓;↓>, <↓;↑>
  - State coupling faults [**4**]: <0;0>, <0;1>, <1;0>, <1;1>
  - Two linked faults within the same group: CFin-CFin [**3**], CFid-CFid [**6**], CFst-CFst [**6**]
  - Two linked faults across the groups: CFin-CFid [**8**], CFin-CFst [**8**], CFid-CFst [**16**]
  - Stuck-open faults (SopF)
  - Data retention fault (DTR)
  - …

Comparison of March Tests



Name / Algorithm	Faults detected
MATS++ $\Updownarrow(w0);\Uparrow(r0,w1);\Downarrow(r1,w0,r0)$	SAF/AF
March X $\Updownarrow(w0);\Uparrow(r0,w1);\Downarrow(r1,w0);\Updownarrow(r0)$	AF/SAF/TF/CFin
March Y $\Updownarrow(w0);\Uparrow(r0,w1,r1);\Downarrow(r1,w0,r0);\Updownarrow(r0)$	AF/SAF/TF/CFin
March C– $\Updownarrow(w0);\Uparrow(r0,w1);\Uparrow(r1,w0);\Downarrow(r0,w1);\Downarrow(r1,w0);\Updownarrow(r0)$	SAF/AF/TF/CF

	MATS++	March X	March Y	March C-
SAF	✓	✓	✓	✓
TF	✓	✓	✓	✓
AF	✓	✓	✓	✓
SOF	✓		✓	
CFin		✓	✓	✓
CFid				✓
CFst				✓

Test Algorithm Generation by Simulation (TAGS)

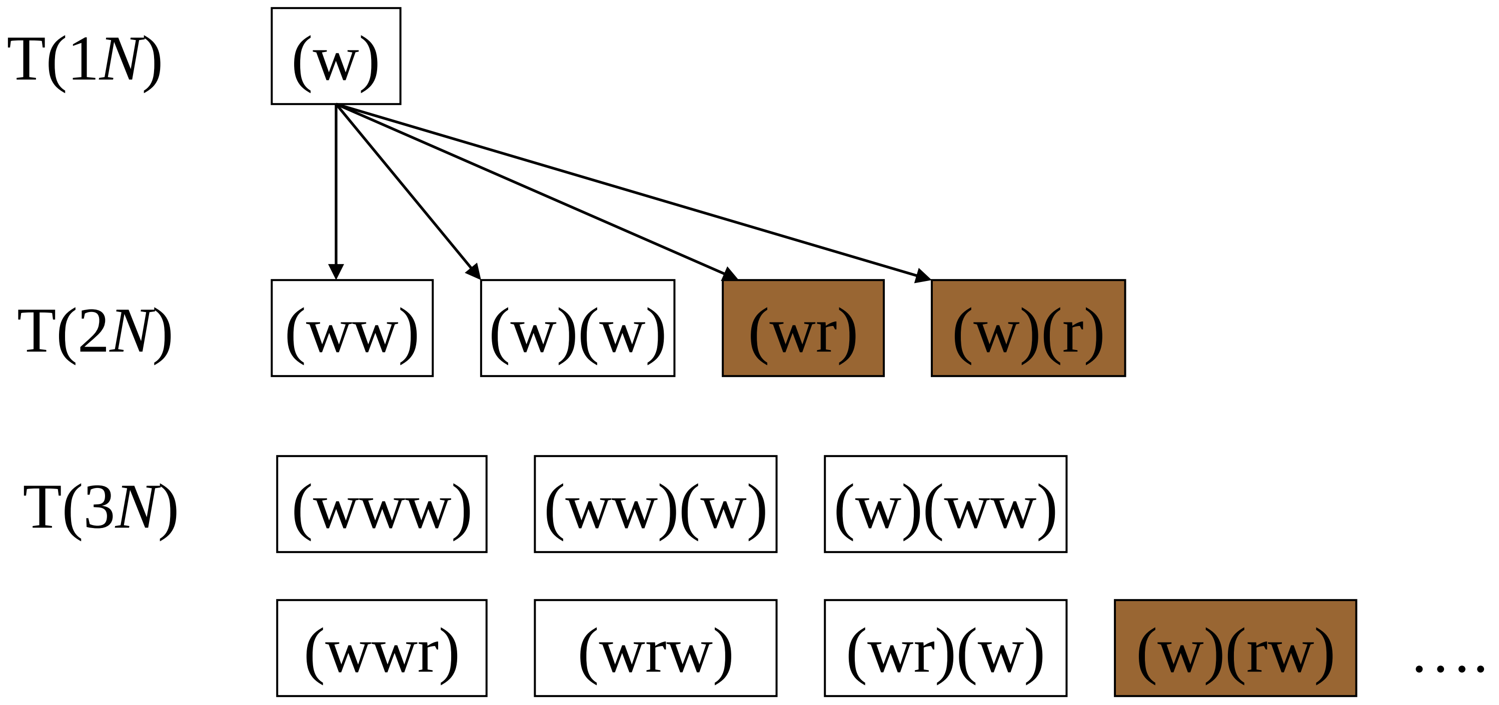
- Target fault models (SAF, TF, AF, SOF, CFin, CFid, CFst), time constraints ∞.
  - Given a set of target fault models, generate a test with 100% fault coverage
  - Given a set of target fault models and a test length constraint, generate a test with the highest fault coverage
- Priority setting for fault models
  - Test length/test time can be reduced
- Diagnostic test generation
  - Need longer test to distinguish faults
- March template abstraction

TAGS Template and Heuristics

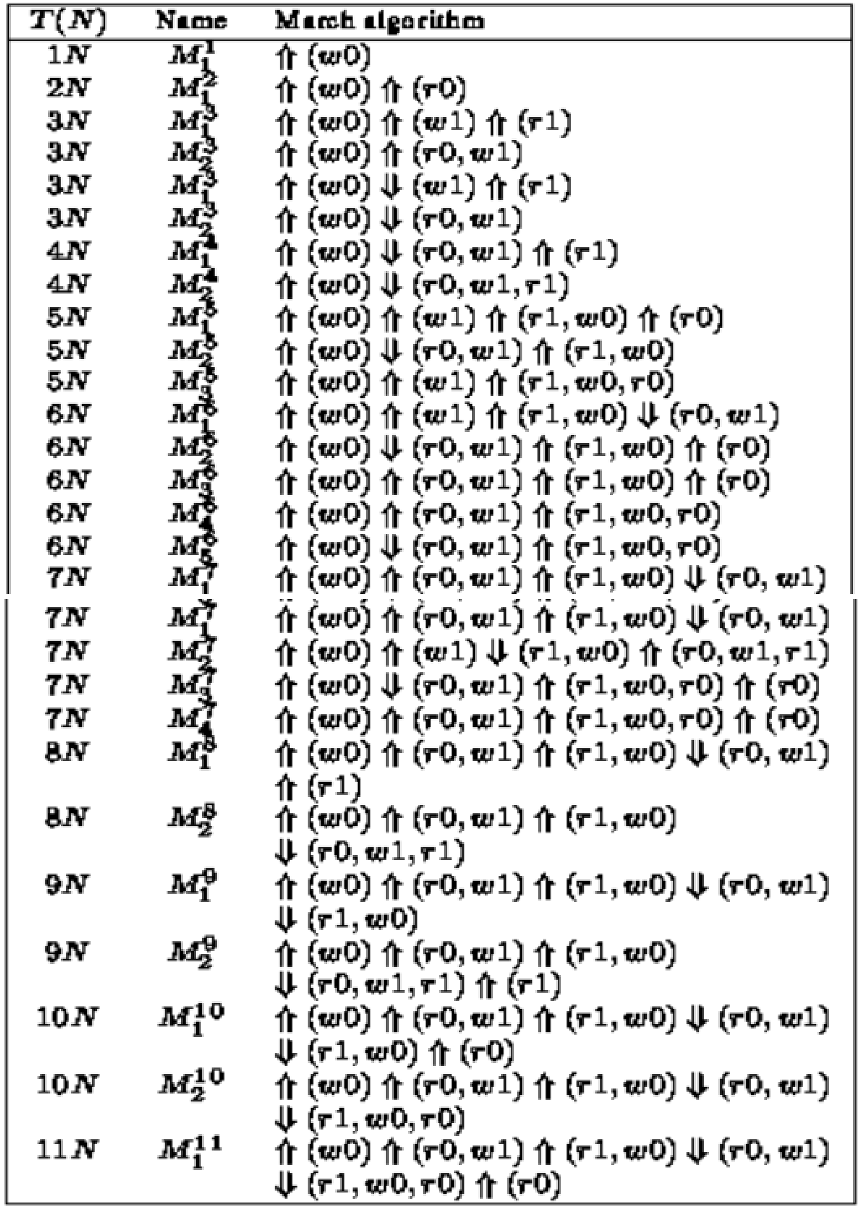
- March template abstraction:

$$\begin{tikzcd}
\boxed{\uparrow(w0);\ \uparrow(r0,w1);\ \downarrow(r1,w0,r0)} \arrow[d] \\
\boxed{\uparrow(w)\ \uparrow(r,w);\ \downarrow(r,w,r)} \arrow[d] \\
\boxed{(w)(rw)(rwr)}
\end{tikzcd}$$

- Exhaustive generation: complexity is very high, e.g., 6.7 million templates when *N* = 9
- Heuristics should be developed to select useful templates

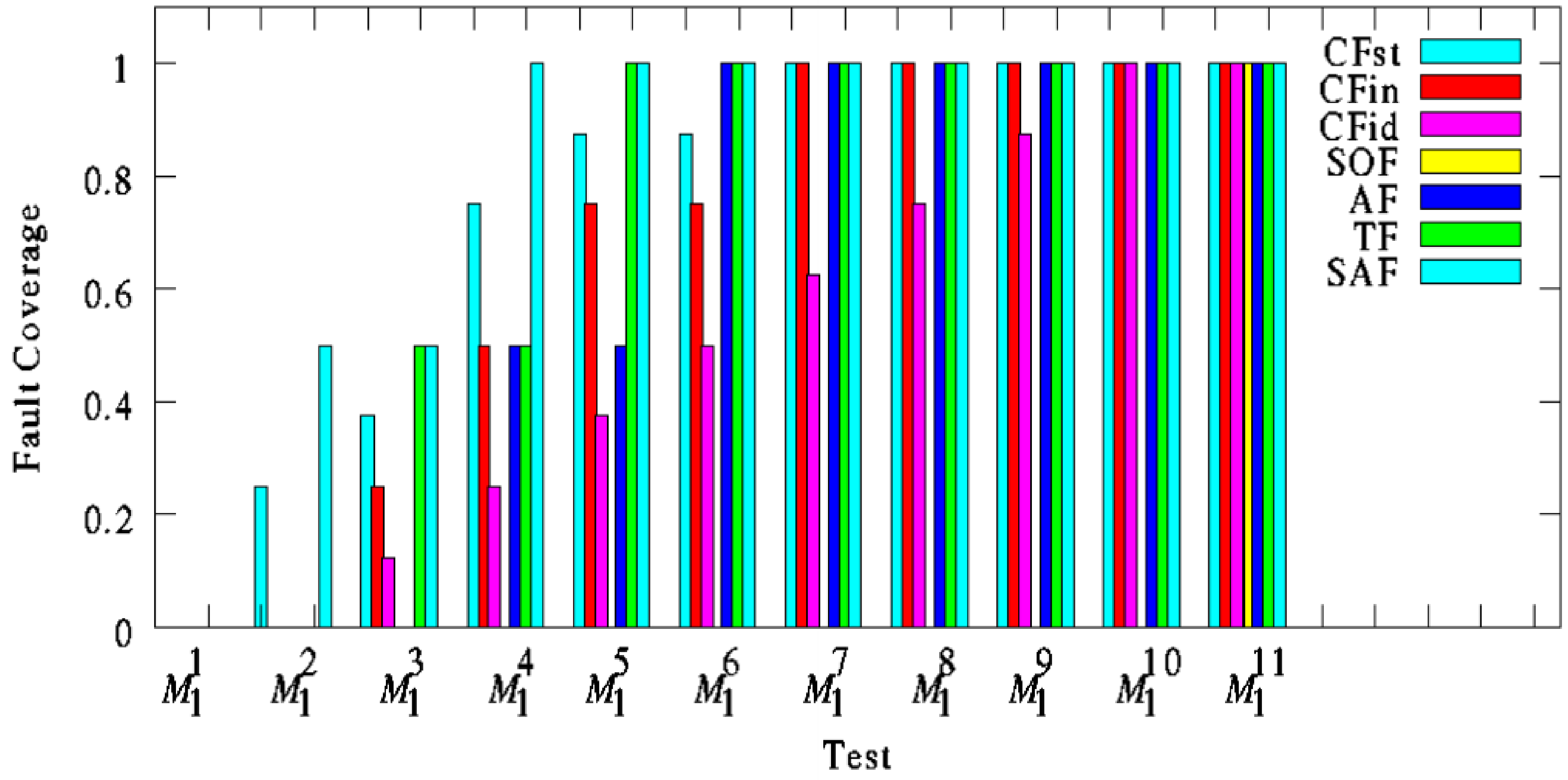


TAGS Results

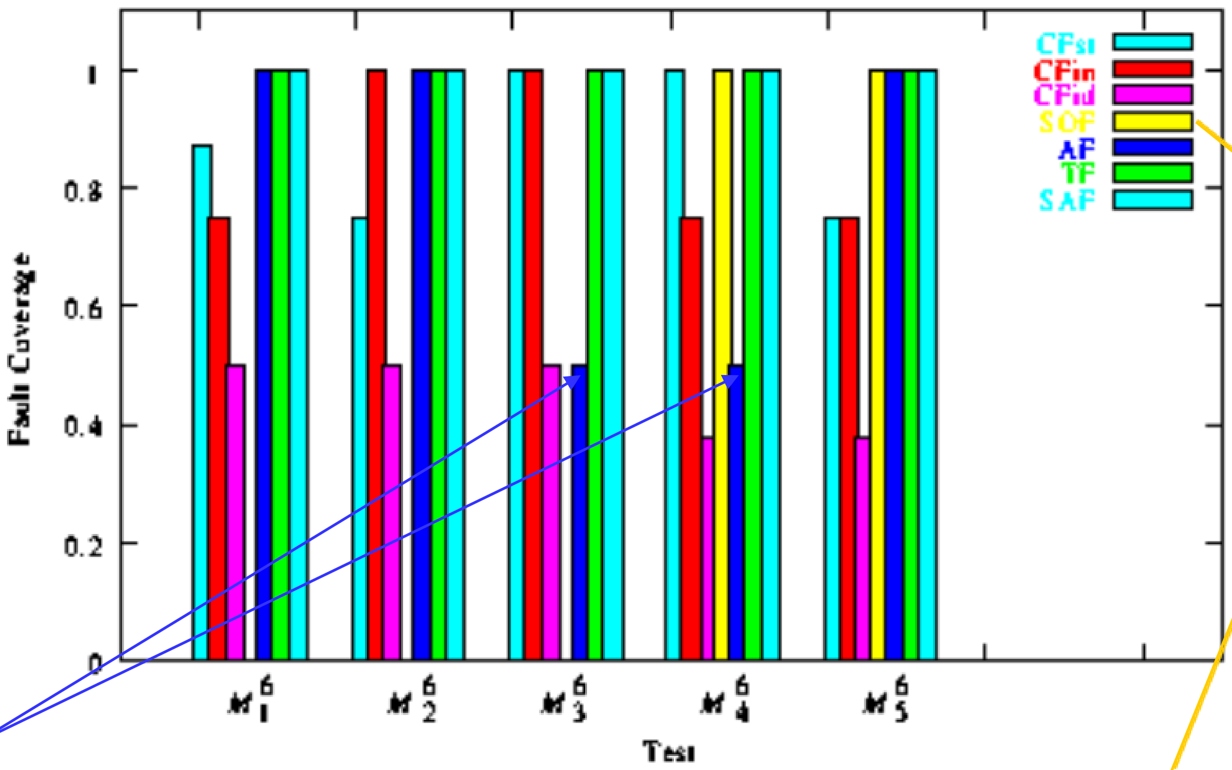


$T(N)$	Name	March algorithm
$1N$	M_1^1	$\Uparrow(w0)$
$2N$	M_1^2	$\Uparrow(w0)\ \Uparrow(r0)$
$3N$	M_1^3	$\Uparrow(w0)\ \Uparrow(w1)\ \Uparrow(r1)$
$3N$	M_2^3	$\Uparrow(w0)\ \Uparrow(r0,w1)$
$3N$	M_1^3	$\Uparrow(w0)\ \Downarrow(w1)\ \Uparrow(r1)$
$3N$	M_2^3	$\Uparrow(w0)\ \Downarrow(r0,w1)$
$4N$	M_1^4	$\Uparrow(w0)\ \Downarrow(r0,w1)\ \Uparrow(r1)$
$4N$	M_2^4	$\Uparrow(w0)\ \Downarrow(r0,w1,r1)$
$5N$	M_1^5	$\Uparrow(w0)\ \Uparrow(w1)\ \Uparrow(r1,w0)\ \Uparrow(r0)$
$5N$	M_2^5	$\Uparrow(w0)\ \Downarrow(r0,w1)\ \Uparrow(r1,w0)$
$5N$	M_3^5	$\Uparrow(w0)\ \Uparrow(w1)\ \Uparrow(r1,w0,r0)$
$6N$	M_1^6	$\Uparrow(w0)\ \Uparrow(w1)\ \Uparrow(r1,w0)\ \Downarrow(r0,w1)$
$6N$	M_2^6	$\Uparrow(w0)\ \Downarrow(r0,w1)\ \Uparrow(r1,w0)\ \Uparrow(r0)$
$6N$	M_3^6	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0)\ \Uparrow(r0)$
$6N$	M_4^6	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0,r0)$
$6N$	M_5^6	$\Uparrow(w0)\ \Downarrow(r0,w1)\ \Uparrow(r1,w0,r0)$
$7N$	M_1^7	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0)\ \Downarrow(r0,w1)$
$7N$	M_1^7	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0)\ \Downarrow(r0,w1)$
$7N$	M_2^7	$\Uparrow(w0)\ \Uparrow(w1)\ \Downarrow(r1,w0)\ \Uparrow(r0,w1,r1)$
$7N$	M_3^7	$\Uparrow(w0)\ \Downarrow(r0,w1)\ \Uparrow(r1,w0,r0)\ \Uparrow(r0)$
$7N$	M_4^7	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0,r0)\ \Uparrow(r0)$
$8N$	M_1^8	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0)\ \Downarrow(r0,w1)$ $\Uparrow(r1)$
$8N$	M_2^8	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0)$ $\Downarrow(r0,w1,r1)$
$9N$	M_1^9	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0)\ \Downarrow(r0,w1)$ $\Downarrow(r1,w0)$
$9N$	M_2^9	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0)$ $\Downarrow(r0,w1,r1)\ \Uparrow(r1)$
$10N$	M_1^{10}	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0)\ \Downarrow(r0,w1)$ $\Downarrow(r1,w0)\ \Uparrow(r0)$
$10N$	M_2^{10}	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0)\ \Downarrow(r0,w1)$ $\Downarrow(r1,w0,r0)$
$11N$	M_1^{11}	$\Uparrow(w0)\ \Uparrow(r0,w1)\ \Uparrow(r1,w0)\ \Downarrow(r0,w1)$ $\Downarrow(r1,w0,r0)\ \Uparrow(r0)$

Simulation Results for TAGS (1N to 11N)

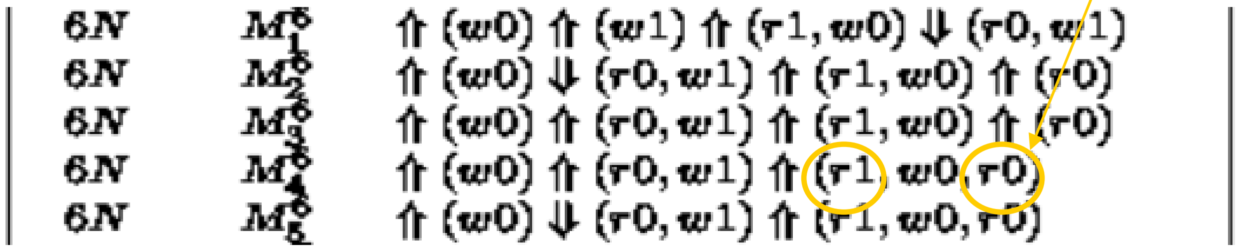


Simulation Results for TAGS (6N)



Detecting SopF requires (…rx…rx'…)

Detecting AF requires ⇑ and ⇓ addresses that do not exist here



6N	M_1^6	⇑ (w0) ⇑ (w1) ⇑ (r1, w0) ⇓ (r0, w1)
6N	M_2^6	⇑ (w0) ⇓ (r0, w1) ⇑ (r1, w0) ⇑ (r0)
6N	M_3^6	⇑ (w0) ⇑ (r0, w1) ⇑ (r1, w0) ⇑ (r0)
6N	M_4^6	⇑ (w0) ⇑ (r0, w1) ⇑ (r1, w0, r0)
6N	M_5^6	⇑ (w0) ⇓ (r0, w1) ⇑ (r1, w0, r0)

Word-Oriented Memory Test

- A word-oriented memory has Read/Write operations that access the memory cell array by a word instead of a bit.
- Word-oriented memories can be tested by applying a bit-oriented test algorithm repeatedly with a set of different data backgrounds:
  - The repeating procedure multiplies the testing time

Word-Oriented Memory Test (cont.)

- Background bit is replaced by background word
  - Bit level MATS++: {⇕(w0); ⇑(r0,w1); ⇓(r1,w0,r0)}
  - Word level MATS++: {⇕(wa); ⇑(ra,wb); ⇓(rb,wa,ra)}
    (***a*** is complement of ***b***)
- Conventional method is to use logm+1 different backgrounds for m-bit words
  - Called *standard backgrounds*
  - m=8 bits: 00000000, 01010101, 00110011, and 00001111
  - Apply the test algorithm logm+1=4 times, so complexity is 4*6N/8=3N

Cocktail-March Algorithm

- Motivation:
  - —Repeating the same algorithm for all logm+1 backgrounds is redundant as far as intra-word coupling faults are concerned
  - —Different algorithms target different faults.
- Approaches:
  1. Use multiple backgrounds in a single algorithm run
  2. Merge and forge different algorithms and backgrounds into a single algorithm
- Good for word-oriented memories

Cocktail-March Algorithm (cont.)

- Algorithm (by Wu et al. – TCAD 04/2002):
  - March C- (complexity of 10N) for solid background (0000)
  - Then a 5N March for each of other standard backgrounds (0101, 0011): ⇕(wa,wb,rb,wa,ra)
- Results:
  - Complexity is (10+5logm)N, where m is word length and N is word count
  - Test time is reduced by 39% if m=4, as compared with extended March C-
  - Improvement increases as m increases

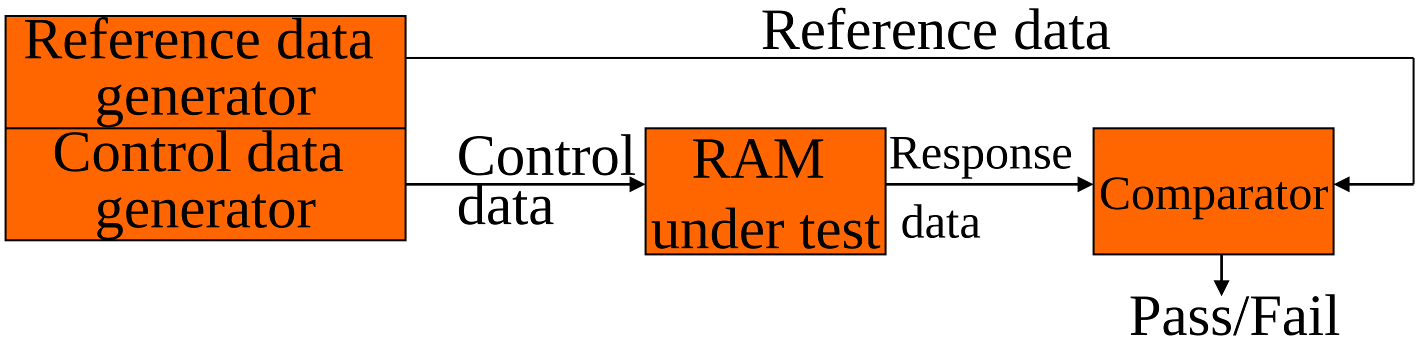
Pseudorandom Memory Tests

Pseudo-Random ‘PR’ Memory Tests

- Purpose
  - Explain concept of pseudo-random (PR) testing
  - Compute test length of PR tests for SAFs and *k*-CFs
  - Evaluation of PR tests
  - PR pattern generators and test response evaluators
- Sources of material
  - Mazumder, P. and Patel, J.H. (1992). *An Efficient Design of Embedded Memories and their Testability Analysis using Markov Chains*. JETTA, Vol. 3, No. 3; pp. 235-250
  - Krasniewski, A. and Krzysztof, G. (1993). *Is There Any Future for Deterministic Self-Test of Embedded RAMs?* In Proc. ETC’93; pp. 159-168
  - van de Goor, A.J. (1998). *Testing Semiconductor Memories, Theory and Practice*. ComTex Publishing, Gouda, The Netherlands
  - van de Goor, A.J. and de Neef, J. (1999*). Industrial Evaluation of DRAM Tests.* In Proc. Design and Test in Europe (DATe’99), March 8-13, Munich; pp. 623-630
  - van de Goor, A.J. and Lin, Mike (1997). *The Implementation of Pseudo-Random Tests on Commercial Memory Testers*. In Proc. IEEE Int. Test Conf., Washington DC, 1997, pp. 226-235

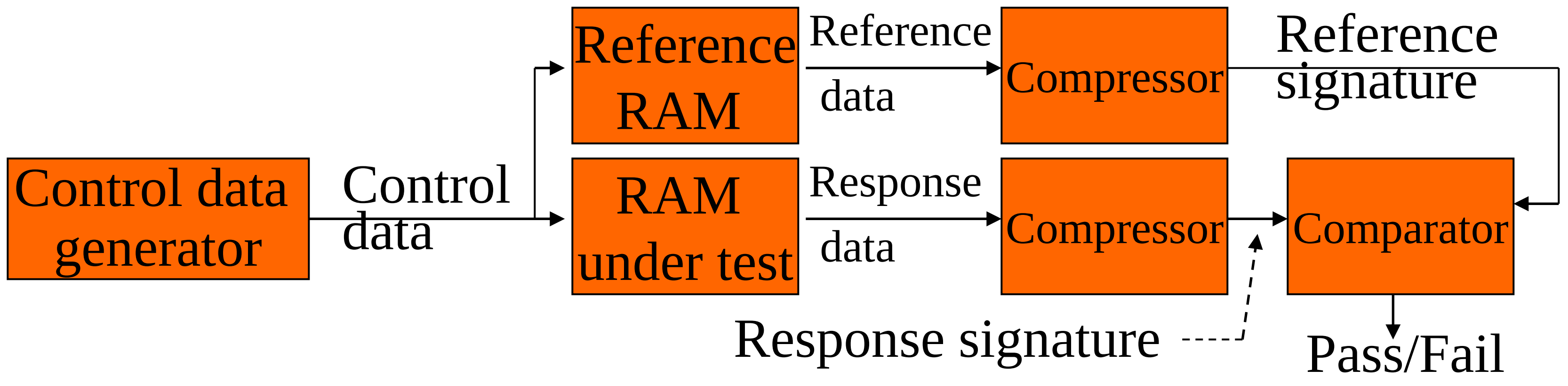
Concepts of PR Memory Testing

- Deterministic tests
  - Control & Reference data for the RAM under test have predetermined values
  - The Response data of the RAM under test is compared with the expected data, in order to make Pass/Fail decision

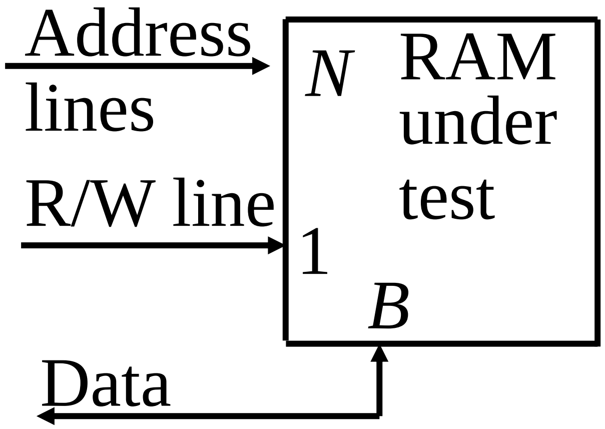


Concepts of PR Memory Testing (cont.)

- Pseudo-random tests
  - Control data on some or all inputs established pseudo-randomly
  - Reference data can be obtained from a Reference RAM or, as shown, from a compressor



Concepts of PR Memory Testing (cont.)



- Memory tests use
  — control values for:
    – Address lines (*N*)
    – R/W line (1)
  — write data values (*B*)
- *Deterministic* test method

  Uses deterministic control and write data values
- In a test the following can be *Deterministic* (**D**) or *PR* (**R**)
  — The Address (**A**): DA or RA
  — The Write (**W**) operation: DW or RW
  — The Data (**D**) to be written: DD or RD
  ⇒ MATS+ is a DADWDD (Det. Addr, Det. Write, Det. Data) test
- In a PR test at least ONE component has to be PR
  — This can be: A (Address) &/or W (Write operation) &/or D (Data)
  — *PR tests* are preferred over *random* tests: PR tests are *repeatable*

Pseudo-Random Tests for SAFs

Some probabilities for computing the *test length* (*TL*)

- The TL is a function of the *escape probability 'e'*

- p: probability that a *line* has the value 1
- p_a: probability that an *address line* has the value 1
- p_d: probability that a *data line* has the value 1
- p_w: probability that the *write line* has the value 1
- p_A: probability of selecting address A (with z 0s and N-z 1s)

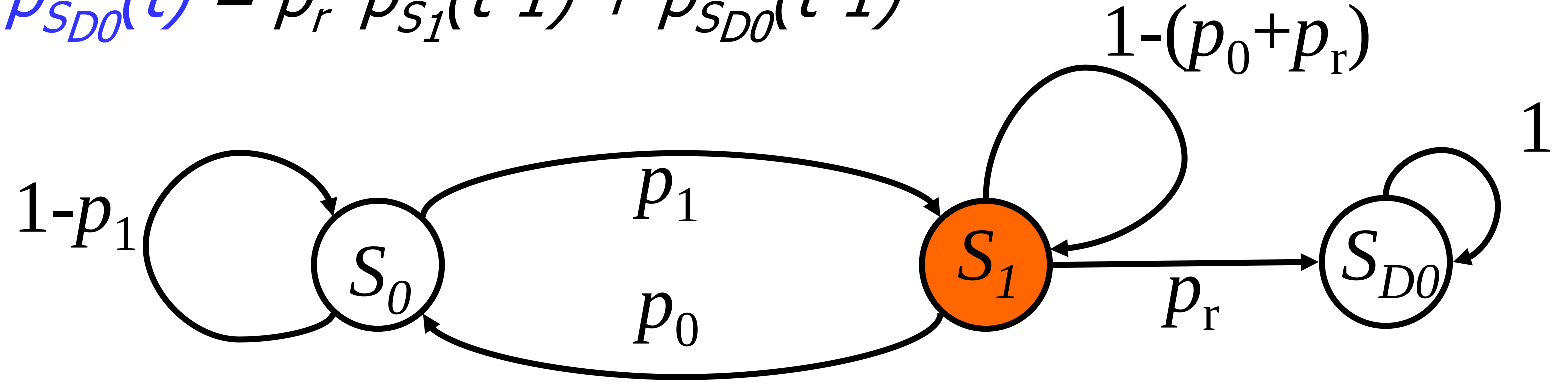
$$p_A = (1- p_a)^z * p_a^{(N-z)}$$

- p_1: probability of *writing 1* to address A; $p_1 = p_d * p_w * p_A$
- p_0: prob. of *writing 0* to address A; $p_0 = (1- p_d) * p_w * p_A$
- p_r: probability of *reading* address A; $p_r = (1 - p_w) * p_A$
- Given that a particular address A has been selected, the operation will either be a w1, w0 or r. Therefore, $p_A = p_1 + p_0 + p_r$

Test Length of PR Test for SAFs

Markov chain for detecting a SA0 fault (SA1 fault is similar)

- S_0 : state in which a 0 is stored in the cell
- S_1 : state in which a 1 should be in the cell
- S_{D0} : state in which SA0 fault is detected (absorbing state)
- $p_{S_0}(t)$: probability of being in state S_0 at time t
- Initial conditions: $p_{S_0}(0)=1-p_{I1}$, $p_{S_1}(0)=p_{I1}$, $p_{S_{D0}}(0)=0$
- $p_{S_0}(t) = (1- p_1)*p_{S_0}(t-1) + p_0* p_{S_1}(t-1)$
- $p_{S_1}(t) = p_1*p_{S_0}(t-1) + (1- p_0 -p_r)* p_{S_1}(t-1)$
- $p_{S_{D0}}(t) = p_r*p_{S_1}(t-1) + p_{S_{D0}}(t-1)$


$$\begin{tikzcd} S_0 \arrow[loop left, "1-p_1"] \arrow[r, bend left, "p_1"] & S_1 \arrow[l, bend left, "p_0"] \arrow[loop above, "1-(p_0+p_r)"] \arrow[r, "p_r"'] & S_{D0} \arrow[loop right, "1"] \end{tikzcd}$$

Test Length of PR Test for SAFs (cont.)

- With *deterministic testing* fault detected with *certainty*
- With *PR testing* fault detected with an *escape probability 'e'*
  - SA0 fault is detected when: $p_{S_{D0}}(t) \geq 1\text{-}e$
  - $T_0(e)$ is TL (test length) for SA0 faults
- $T(e)$: The test length for SAFs is: $T(e) = max\,(T_0(e), T_1(e))$

$$T_0(e) = \left\lceil \frac{ln\left(\frac{2.\alpha.e}{1+\alpha-2\cdot(1-p_w)\cdot p_{I1}}\right)}{ln\left(1-\frac{(1-\alpha)\cdot p_A}{2}\right)} \right\rceil \qquad T_1(e) = \left\lceil \frac{ln\left(\frac{2.\beta.e}{1+\beta-2\cdot(1-p_w)\cdot(1-p_{I1})}\right)}{ln\left(1-\frac{(1-\beta)\cdot p_A}{2}\right)} \right\rceil$$

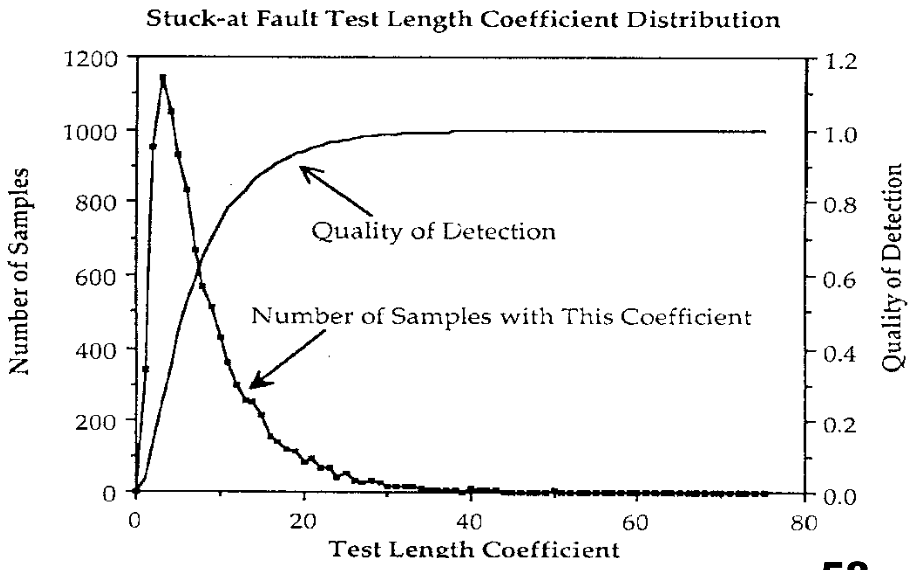
- $\beta = [1\text{-}4\,(1\text{-}p_d)\,p_w\,(1\text{-}p_w)]^{1/2}$
- $\alpha = [1\text{-}4\,p_d\,p_w\,(1\text{-}p_w)]^{1/2}$

Test Length of PR Test for SAFs (cont.)

- Test length is a function of the escape probability and memory size.
- *Test length coefficient =T(e)/n* (using $p_a=p_d=p_w=1/2$)
  - T(e): total number of operations
  - T(e)/n: test length in terms of number of operations per cell. It is independent of memory size *n* & proportional to ln(*e*)
- e.g. MATS+ test requires 5 operations per cell to detect all SAFs, while PR test requires 48+1(for initialization)=49 operations per cell to detect SAFs with an escape probability of e=0.001.

Test length coefficient

e	Memory size			
	n=32	*n*=1k	*n*=32k	*n*=1024k
0.1	17	17	17	17
0.01	33	33	33	33
0.001	48	48	48	48
0.0001	64	64	64	64
0.00001	80	80	80	80



Test Lengths: Deterministic -- PR Tests

Fault		Test length coefficient			
		Deterministic	Pseudo-random		
			e=0.01	e=0.001	e=0.000001
SAF		5*n (MATS+)	33*n	46*n	93*n
CFid		10*n (March C-)	145*n	219*n	445*n
ANPSF k=3		28*n	294*n	447*n	905*n
APSF k=3		$n+32*n*\log_2 n$	294*n	447*n	905*n
ANPSF k=5		195*n	1200*n	1805*n	3625*n
APSF k=5		?	1200*n	1805*n	3625*n

Observations

- Note: ANPSFs have *k*-2 cells in only *one* position
- For simple fault models deterministic tests more efficient
  - Detect *all faults* of ***some*** *fault models* with *e* = 0
- For complex fault models PR tests do exist
  - PR tests detect *all faults* of *all fault models, however* with *e* > 0

Strengths/Weaknesses of PR Tests

- Deterministic tests based on *a-priori* fault models
  - Models usually *restricted* to the memory cell array
  - 5% of real defects not explained (Krasniewski, ETC'93)
  - Tests detect 100% of *targeted* faults only
- Pseudo-random tests
  - Not targeted towards a particular fault model
    - PR tests detect faults of *all fault models*; however, with some $e > 0$
  - Long test time: Test length (TL) proportional to $\ln(e)$ and 2^{k-2}
    - For CFids: $445*n$ ($e = 10^{-5}$) versus $10*n$ (for March C-)
    - Less of a problem for SRAMs (e.g.,1 Mword, 1ns, $1000n$ test takes 1s)
  - Random pattern resistant faults
    - with a *large data state* (e.g., bit line imbalance)
    - requiring a *large address/operation state* (e.g., Hammer tests)
  - Cannot *locate faults* easily (For laser/dynamic repair)
  - Well suited for BIST
  - Very useful for verification purposes
  - Used for production SRAM testing (together with deterministic tests)
    - Unknown fault models, short time to volume, high speed SRAM