Chapter 9 Memory Testing

EDAC for RAMs

- ✯ A fault-tolerant technique.
- ✯ All kinds of FT require redundancy. In EDAC for RAMs, redundancy is in the form of a *code*.
- ✯ Extra bits (check bits) are stored along with the message (information) bits.

Definition 1
The *(Hamming) weight* of a vector $X = x_1 x_2 \cdots x_n$, denoted as $w(X)$, is the number of nonzero components in X. Each component x_i is an element of $GF(q)$ and the vector X is also called an n-tuple or a word.

- ☞ Usually $q = 2^b$ for some positive integer b. Unless otherwise indicated, we assume $b = 1$.

Definition 2
The *(Hamming) distance* between 2 vectors (words) X and Y, denoted as $d(X, Y)$, is the Hamming weight of $X - Y$, i.e., the number of components in which these two vectors differ.

- ☞ $d(1000, 0101) = 3$; $d(37AE, 40A5) = 3$
- ☞ $d(X, X) = 0$; $d(X, Y) > 0$ if $X \neq Y$ (positive definiteness)
- ☞ $d(X, Y) = w(X - Y) = w(Y - X) = d(Y, X)$ (symmetry)
- ☞ $d(X, Y) + d(Y, Z) \geq d(X, Z)$ (triangle inequality)

Definition 3
The *minimum (Hamming) distance* of a code **C**, denoted as d_m, is the minimum of the distances between all pairs of codewords in **C**.

Example 1
The binary code for $n = 3$:

```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

In a binary code (for any n), $d_m = 1$. If we add an overall parity bit to each vector (codeword), then we obtain a parity code with $d_m = 2$. □

Theorem 1
A code **C** is d-error detectable iff $d_m \geq d + 1$, i.e., it is necessary and sufficient that $d_m \geq d + 1$ for the code **C** in order to detect any error pattern of weight d or less.

- For single-error detection, we need $d_m \geq 2$.
- In a d_m-2 code of n bits, we have a max of 2^{n-1} codewords. For example, consider the single-parity code: $x_1x_2x_3x_4x_5p$, where $p = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5$: 5 message bits $\Rightarrow$ $2^{6-1} = 2^5 = 32$ valid codewords and 32 non-codewords (with bad parity).
- To do single-error correction, we require a greater distance. For example, we need a distance of 4 to detect doubles while correcting singles.

Theorem 2
A code **C** is t-error correctable iff $d_m \geq 2t + 1$, i.e., it is necessary and sufficient that $d_m \geq 2t + 1$ for the code **C** in order to detect and correct any error pattern of weight t or less.

Theorem 3
A code **C** is t-error correctable and d-error detectable ($d \geq t$) iff $d_m \geq t + d + 1$.

*Hamming Code

Consider m message bits and k check bits. The codeword of $m + k$ bits will be written into the channel (RAM). We need to determine the smallest k and specify the locations of the check bits so that when an error is detected we know *which* bit is wrong.

① Determine k: k should be large enough to uniquely identify any of the $m+k$ bits $\Rightarrow\ k = \lceil \log(m + k + 1) \rceil$.

② Determine cb locations: place the check bits (cb's) in bit positions corresponding to powers of 2, i.e.,

$$b_1 b_2 \cdot b_4 \cdots b_8 \cdot \cdots \cdot b_{16} \cdot \cdots \cdot$$

And each is set to establish *even* parity of some bits as follows:

$$\begin{aligned} cb \text{ for } 1 &= b_3 \oplus b_5 \oplus b_7 \oplus b_9 \cdots \\ cb \text{ for } 2 &= b_3 \oplus b_6 \oplus b_7 \oplus b_{10} \cdots \\ cb \text{ for } 4 &= b_5 \oplus b_6 \oplus b_7 \oplus b_{12} \cdots \\ &\vdots \\ cb \text{ for } 2^{k-1} &= \cdots \text{ (bits with } 2^{k-1} \text{ set in their bit \#)} \end{aligned}$$

☞ In this case, $d_m = 3$.

Example 2
BCD code ⇒ cb's go in positions 1, 2, & 4.

① When writing into RAM, generate cbs & write both mbs and cbs.

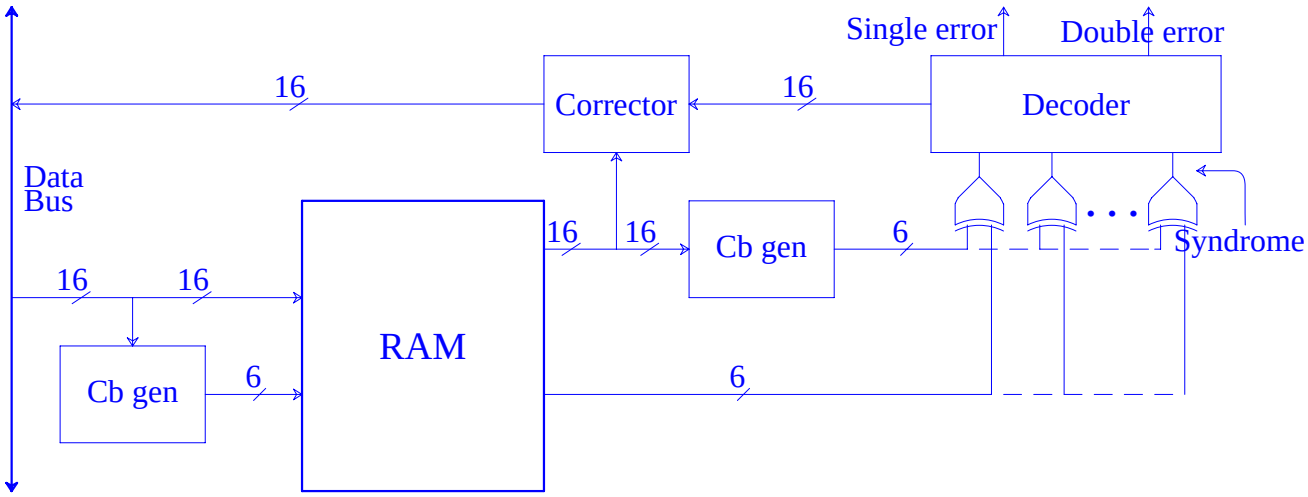
	b_1	b_2	b_3	b_4	b_5	b_6	b_7
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1
+	0	1	1	1	0	1	0
–	1	0	1	0	0	1	1

② When reading from RAM, generate cbs based on mbs received and compare received cbs with generated cbs:

$$\text{error syndrome} \leftarrow \text{received cbs} \oplus \text{generated cbs}.$$

The error syndrome gives the position of the faulty bit. It is 0 if there is no error.

③ Suppose the received cbs from RAM are $b_1 b_2 b_4 = 100$, and the generated cbs are $b_1 b_2 b_4 = 110$, then the error syndrome is $100 \oplus 110 = 010 = 2_{10}$, which implies that bit 2 is faulty. □



☞ $m = 16 \Rightarrow k = \lceil \log(16 + k + 1) \rceil = 5 \Rightarrow$ real memory is 21-bit wide.

☞ Add an overall parity bit $\Rightarrow d_m = 4 \Rightarrow$ SEC-DED.

Parity Check Table:

bit #	7	6	5	4	3	2	1
cb 4	X	X	X	⊗			
cb 2	X	X			X	⊗	
cb 1	X		X		X		⊗

☞ Each row corresponds to a cb (even parity). The cb is formed by taking $\oplus$ of all bits marked X. For example, $\frac{7\ 6\ 5}{1\ 0\ 1} \Rightarrow \frac{4}{0}$.

#	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
16	X	X	X	X	X	⊗															
8							X	X	X	X	X	X	X	⊗							
4	X	X					X	X	X	X					X	X	X	⊗			
2			X	X			X	X			X	X			X	X			X	⊗	
1	X		X		X		X		X		X		X		X		X		X		⊗

☞ Implementation presents some problems:

❶ Number of bits XORed not same for each cb.

❷ Number of bits checked ranges from 5 to 10.

➨ The propagation delay through a 10-input XOR tree is much larger than that for a 5-input one.

❸ Two bits in error can cause a *correction* to occur when in fact the answer is wrong ($\because d_m = 3$).

➨ For example, cb 1 & cb 2 in error $\Rightarrow b_3 \oplus 1$ (but in fact we should do $b_1 \oplus 1$ and $b_2 \oplus 1$).

*Modified Hamming Code

- ✯ Take straight HC, and add an *overall* parity bit.
- ✯ More typical in real systems: SEC-DED.

Four Cases on Read-Out:

① $p = 0$ & syndrome = 0 $\Rightarrow$ good data.

② $p = 1$ & syndrome = 0 $\Rightarrow$ single error in p, ignore.

③ $p = 0$ & syndrome $\neq 0 \Rightarrow$ double error.

④ $p = 1$ & syndrome $\neq 0 \Rightarrow$ correctable single error.

Theorem 4
A parity check table represents an SEC-DED (d_m-4) code iff there is no set of 3 or fewer columns with an even number (i.e., 0 or 2) of X's in all rows.

Counter example:

X		X		X		X	no change on cb
X	X				X	X	no change on cb $\Rightarrow$ $d_m = 3$
	X		X	X		X	no change on cb

Theorem 5
If a parity check table has a) no all-zero column, b) no 2 identical columns, and c) an odd number of X's in each column, then it represents a d_m-4 code.

Exercise 1
Prove the above theorems.

Memory Functional Test

① Characterize the device: determine the most likely failure modes.

② Select a set of tests to detect these faults (failure modes).

- Classical fault models are not sufficient to represent all important failure modes in a RAM.
- Checking experiments or unrolling are impossible for RAMs.
- Use functional fault models:
  - **Memory cell faults**
    - Stuck-at fault (SAF): cell or line s-a-0 or s-a-1
    - Stuck-open fault (SOF): open cell or broken line
    - Transition fault (TF): cell fails to transit
    - Data retention fault (DRF): cell fails to retain its logic value after some specified time due to, e.g., leakage, resistor opens, or feedback path opens
    - Coupling fault (CF)
      - Inversion coupling fault (CFin): a transition in one cell (aggressor) inverts the content of another cell (victim)
      - Idempotent coupling fault (CFid): a transition in one cell forces a fixed logic value into another cell
      - State coupling fault (CFst): a cell/line is forced to a fixed state only if the coupling cell/line is in a given state (a.k.a. pattern-sensitivity fault (PSF))
    - Bridging fault (BF): short between cells (can be AND type or OR type)
    - Neighborhood Pattern Sensitive Fault (NPSF)
      - Active (Dynamic) NPSF
      - Passive NPSF
      - Static NPSF
  - **Address decoder faults** (AFs)

- No cell accessed by certain address
- Multiple cells accessed by certain address
- Certain cell not accessed by any address
- Certain cell accessed by multiple addresses

☆ Dynamic Faults

- **Recovery faults**: when some part of the memory cannot recover fast enough from a previous state.
  - **Sense amplifier recovery**: sense amplifier saturation after reading/writing a long string of 0s or 1s.
  - **Write recovery**: a write followed by a read or write at a different location resulting in reading or writing at the same location due to slow address decoder.
- **Disturb faults**: victim cell forced to 0 or 1 if we read or write aggressor cell (may be the same cell).
- **Data Retention faults**: memory loses its content spontaneously, not caused by read or write.
  - DRAM refresh fault: Refresh-line stuck-at fault
  - DRAM leakage fault: Sleeping sickness—loose data in less than specified hold time (typically hubdreds of μs to tens of ms); caused by charge leakage or environment sensitivity; usually affects a row or a column
  - SRAM leakage fault: Static data losses—defective pull-up device inducing excessive leakage currents which can change the state of a cell
  - Checkerboard pattern triggers max leakage

RAM Functional Test Algorithm

Definition 4

A RAM *test algorithm* (or simply *test*) is a finite sequence of *test elements*. A test element contains a number of memory operations (access commands), data patterns (backgrounds) specified for the operations, and address (sequence) specified for the operations.

Table 1: Test time as a function of memory size.

Size	Complexity			
n	n	$n \log n$	$n^{3/2}$	n^2
1K	0.0001s	0.001s	0.0033s	0.105s
4K	0.0004s	0.0048s	0.0262s	1.7s
16K	0.0016s	0.0224s	0.21s	27s
64K	0.0064s	0.1s	1.678s	7.17m
256K	0.0256s	0.46s	13.4s	1.9h
1M	0.102s	2.04s	1.83m	1.27d
4M	0.41s	9.02s	14.3m	20.39d
16M	1.64s	39.36s	1.9h	326d
64M	6.56s	2.843m	15.25h	14.3y
256M	26.24s	12.25m	5.1d	229y
1G	1.75m	52.48m	40.8d	3659y

Definition 5

A *march test* consists of a finite sequence of march elements, while a *march element* is a finite sequence of operations applied to every cell in the memory array before proceeding to the next cell. An *operation* can consist of writing a 0 into a cell (w0), writing a 1 into a cell (w1), reading an expected 0 from a cell (r0), and reading an expected 1 from a cell (r1).

- The simplest tests which detect SAFs, TFs and CFs are part of a family of tests called *marches* (march tests).
- A march element can be done in either one of two *address orders*: $\Uparrow$ or $\Downarrow$. When the address order is irrelevant, then the symbol $\Updownarrow$ is used.

- **Zero-one algorithm (MSCAN)**:

```
Procedure ZERO-ONE
{
1: write 0 in all cells;
2: read all cells;
3: write 1 in all cells;
4: read all cells;
}
```

- ☆ The march notation: $\{\Uparrow (w0); \Uparrow (r0); \Uparrow (w1); \Uparrow (r1)\}$.
- ☆ The minimal test—$O(4n)$.
- ☆ A.k.a. memory scan (MSCAN).
- ☆ Not all $\downarrow /1$ TFs are covered; not all CFs are covered.
- ☆ SAFs are covered if the address decoder is correct (not all AFs are covered).
- ☆ Note: A test detects all AFs if it contains the march elements $\Uparrow (rx, \ldots, w\bar{x})$ and $\Downarrow (r\bar{x}, \ldots, wx)$, and the memory is initialized to the proper value before each march element.

✯ **Checkerboard pattern**: Writes 1's and 0's into alternate memory locations in a checkerboard pattern. Wait for several seconds and read. Repeat for complementary patterns.

```
Procedure Checkerboard
{ while(i is odd && j is even)
   { write 0 in cell[i]; write 1 in cell[j];
     pause; read all cells;
     complement all cells;
     pause; read all cells; } }
```

- ☆ Time complexity is $O(4n)$.
- ☆ For shorts between cells, data retention of SRAMs, SAFs, and half of the TFs.
- ☆ The starting point for pattern sensitivity test, but some CFs cannot be detected.
- ☆ Not good for AFs.
- ☆ Must create true physical checkerboard, not logical checkerboard (the engineer must obtain design information about the actual layout and then modify the test addressing accordingly).

- **Galloping (ping-pong) pattern (GALPAT)**: The base cell (BC) is read alternately with every other cell in its set.

```
Procedure GALPAT
{ 1: write 0 in all cells;
  2: for i = 0 to n-1
        { complement cell[i];
          for j = 0 to n-1, j != i
                { read i; read j; }
          complement cell[i]; }
  3: write 1 in all cells;
  4: replay Step 2; }
```

  - $O(4n^2)$, very long sequence (for characterization, not for production tests).
  - A strong test for most faults.
  - All AFs, TFs, CFs, and SAFs are detected and located.
  - Set may be a column, a row, a diagonal, or all cells.

- **Walking pattern (WALPAT)**: It is similar to galloping except that the BC is read only after all others are read.
  - WALPAT (walking 1/0): all cells in the RAM, $O(2n^2)$.

- **Galloping diagonal/row/column**: It is based on GALPAT. Instead of shifting a 1 through the memory, a complete diagonal of 1s is shifted. The whole memory is read after each shift.
  - Complexity is $O(4n^{3/2})$.
  - Detects all faults as GALPAT, except for some CFs.
  - Variations are *galloping row* and *galloping column*.

- **Sliding diagonal/row/column**: As galloping diagonal/row/column, but only those cells which are supposed to contain 1 are read after each shift.
  - Complexity is $O(4n)$.

- ☆ Some CFs and TFs are not covered.
- ☆ More CFs and all TFs can be covered if repeated with a complemented background.

✯ **Butterfly**: This test is also modified from GALPAT, with the purpose to find only AFs and SAFs.

```
Procedure Butterfly
{ 1: write 0 in all cells;
  2: for i = 0 to n-1
        { complement cell i;
          dist = 1;
          while dist <= maxdist   /* maxdist < 0.5*col/row
            {
            read cell at dist north from cell[i];
            read cell at dist east from cell[i];
            read cell at dist south from cell[i];
            read cell at dist west from cell[i];
            read cell[i];
            dist *= 2;           /* or dist += skip */
            }
          complement cell[i]; }
  3: write 1 in all cells;
  4: replay Step 2; }
```

- ☆ Complexity is $O(5n \log n)$.
- ☆ All SAFs and some AFs are detected.

✯ **Moving inversion(MOVI)**: The memory is initialized to contain all 0s, then this string of 0s is successively inverted to become all 1s, and vice versa. MOVI was designed as a shorter alternative for GALPAT.

- ☆ Complexity is $O(12n \log n)$.
- ☆ Both a functional test and an AC parametric test.

- ☆ Functional test—ensures that no cell is disturbed by a read/write on another unrelated cell. It detects all AFs and SAFs.
- ☆ Parametric test—allows for the determination of the best and worst access times together with the address changes imposing these times.

✯ **Surround disturb**: The pattern attempts to examine how the cells in a particular row are affected when complementary data are written into adjacent cells of other rows.

```
Procedure Surround-Disturb
{
1: write a 0 in cell[p,q-1]; /* row p, column q-1 */
2: write a 0 in cell[p,q];
3: write a 0 in cell[p,q+1];
4: write a 1 in cell[p-1,q];
5: verify that cell[p,q+1] contains a 0;
6: write a 1 in cell[p+1,q];
7: verify that cell[p,q-1] contains a 0;
8: verify that cell[p,q] contains a 0;
}
```

- ☆ Done for all cells, and repeated with complementary data.
- ☆ Designed on the premise that DRAM cells are most susceptible to interference from their nearest neighbors (∴ eliminate global sensitivity checks).

March Tests

✯ **Modified algorithmic test sequence (MATS)**: $\{\Updownarrow (w0); \Updownarrow (r0, w1); \Updownarrow (r1)\}$ [Nair, 1979]

- ☆ The shortest march test for (unlinked) SAFs in the cell array and the read/write logic.

- ☆ Detects all AFs for the OR-type technology (the result of reading multiple cells is assumed to be the OR function of the contents of those cells).
- ☆ For AFs of the AND-type technology, use MATS-AND: $\{\Updownarrow\ (w1); \Updownarrow (r1, w0); \Updownarrow (r0)\}$
- ☆ Can detect some CFs and TFs.
- ☆ Number of (read/write) operations = $4n$, which is the same as those for Zero-One and Checkerboard, but MATS has a much better fault coverage.
- ☆ Can be used for word-oriented memory ($4 \cdot 2^N$ operations), with lower detectability for CFs and TFs.

★ **MATS+**: $\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0)\}$ [Abadir & Reghbati, 1983]

- ☆ Detects all SAFs and AFs, used instead of MATS when technology is unknown.
- ☆ The suggested test for unlinked SAFs.
- ☆ Number of operations = $5n$.

★ **Marching 1/0**: $\{\Uparrow (w0); \Uparrow (r0, w1, r1); \Downarrow (r1, w0, r0); \Uparrow (w1); \Uparrow (r1, w0, r0); \Downarrow (r0, w1, r1)\}$

- ☆ The marching-1 pattern begins by writing a background of zeros, then read and write back complement (and read again to verify) values for all cells (from cell[0] to cell[$n-1$], and then from cell[$n-1$] to cell[0]), in $O(7n)$ time.
- ☆ The marching-0 pattern follows exactly the same pattern, with the data reversed.
- ☆ Number of operations = $14n$.
- ☆ Detects all AFs, SAFs and TFs.
- ☆ Can detect only part of the CFs.
- ☆ It is a *complete test*, i.e., all faults that should be detected are covered by the test.
- ☆ It however is a *redundant test*, because only the first three march elements are necessary.

Exercise 2
Prove that Marching 1/0 is a redundant test for AFs, SAFs and TFs. □

- ✯ **MATS++**: $\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0, r0)\}$
  - ☆ Optimized marching-1/0 scheme—complete and irredundant.
  - ☆ Similar to MATS+, but allow for the coverage of TFs.
  - ☆ The suggested test for unlinked SAFs & TFs.
  - ☆ Number of operations = $6n$.

Exercise 3
Prove that March++ is complete and irredundant for AFs, SAFs and TFs. □

- ✯ **March X**: $\{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)\}$
  - ☆ Detects unlinked AFs, SAFs, TFs and inversion CFs (CFins).
  - ☆ Called March X because the test has been used without being published.
  - ☆ Number of operations = $6n$.
- ✯ **March C**: $\{\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Updownarrow (r0); \Downarrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)\}$ [Marinescu, 1982]
  - ☆ For unlinked inversion (CFin), idempotent 2-coupling faults (CFid2) and disturb 2-coupling faults (CFdi2).
  - ☆ It is semi-optimal (redundant).
- ✯ **March C-**: $\{\Updownarrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r1, w0); \Updownarrow (r0)\}$
  - ☆ Detects unlinked AFs, SAFs, TFs and CFs (including CFins, CFids, CFsts and CFdis).
  - ☆ Number of operations = $10n$.
- ✯ **March A**: $\{\Updownarrow (w0); \Uparrow (r0, w1, w0, w1); \Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0)\}$ [Suk & Reddy, 1981]
  - ☆ Detects any combination of *linked* CFids.
  - ☆ The shortest test for AFs, SAFs, linked CFids, TFs not linked with CFids, and certain CFins linked with CFids.

- ☆ It is complete and irredundant.
- ☆ Number of operations = $15n$.

✯ **March Y**: $\{\Updownarrow (w0); \Uparrow (r0, w1, r1); \Downarrow (r1, w0, r0); \Updownarrow (r0)\}$

- ☆ Detects AFs, SAFs, CFins, and TFs linked with CFins.
- ☆ An extension of March X—detects all faults detectable by March X.
- ☆ Called March Y because the test has been used without being published.
- ☆ Number of operations = $8n$.

✯ **March B**: $\{\Updownarrow (w0); \Uparrow (r0, w1, r1, w0, r0, w1); \Uparrow (r1, w0, w1); \Downarrow (r1, w0, w1, w0); \Downarrow (r0, w1, w0)\}$ [Suk & Reddy, 1981]

- ☆ For linked TFs and CFids.
- ☆ An extension of March A.
- ☆ Detects AFs, SAFs, linked CFids, TFs linked with CFids or CFins.
- ☆ It is complete and irredundant.
- ☆ Number of operations = $17n$.

Exercise 4

```
Procedure My-March
{ for(i=0; i<n; i++) write 0 in cell[i];
  pause; /* detects retention of 0---typically 100 ms */
  for(i=0; i<n; i++) read cell[i];
  for(i=0 && j=n-1; i<n/2 && j>(n/2-1); i++ && j--)
    { write 1 in cell[i];
      read cell[i];
      write 1 in cell[j];
      read cell[j]; }
  pause; /* detects retention of 1---typically 100 ms */
  for(i=0; i<n; i++) read cell[i];
  for(i=(n/2-1) && j=n/2; i>=0 && j<n; i-- && j++)
    { write 0 in cell[i];
      read cell[i];
      write 0 in cell[j];
```

```
        read cell[j]; } }
```

Determine the march element type in this procedure. What faults can it detect? □

Exercise 5

Show that the following four march tests are identical in terms of fault coverage:

1. $\{\Uparrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0, r0)\}$
2. $\{\Uparrow (w1); \Uparrow (r1, w0); \Downarrow (r0, w1, r1)\}$
3. $\{\Downarrow (w0); \Downarrow (r0, w1); \Uparrow (r1, w0, r0)\}$
4. $\{\Downarrow (w1); \Downarrow (r1, w0); \Uparrow (r0, w1, r1)\}$

□

Fault	MATS++	March X	March Y	March C$^-$
SAF	100%	100%	100%	100%
TF	100%	100%	100%	100%
SOF	100%	0.2%	100%	0.2%
AF	100%	100%	100%	100%
CFin	75.0%	100%	100%	100%
CFid	37.5%	50.0%	50.0%	100%
CFst	50.0%	62.5%	62.5%	100%

Test Pattern	AF	SAF	TF	CF	Others	Complexity
Zero-One	N	L	N	N		$4n$
Checkerboard	N	L	N	N	Refresh	$4n$
WALPAT	L	L	L	L	Sense amp. rec.	$2n^2$
GALPAT	L	L	L	L	Write rec.	$4n^2$
Galloping Diagonal	LS	L	L	N		$4n^{1.5}$
Butterfly	L	L	N	N		$5n \log n$
MATS	DS	D	N	N		$4n$
MATS+	D	D	N	N		$5n$
Marching 1/0	D	D	D	N		$14n$
MATS++	D	D	D	N		$6n$
March X	D	D	D	D	Unlinked CFin	$6n$
March C-	D	D	D	D	Unlinked CFin	$10n$
March A	D	D	D	D	Unlinked CF	$15n$
March Y	D	D	D	D	Linked TF	$8n$
March B	D	D	D	D	Linked CF	$17n$
MOVI	D	D	D	D	Read access time	$12n \log n$

N='no'; L='locate'; D='detect'; LS='locate some'; DS='detect some'

Word-Oriented Memory

☞ For m-bit-wide word-oriented memory, $\log m$ additional backgrounds should be used. For example, if $m = 8$, we add backgrounds 10101010, 11001100, and 11110000.

Exercise 6

Can we save test time by testing a bit-oriented memory as a word-oriented memory? □