論文2000-37SD-4-11

NPSFs를 고려한 수정된 March 알고리즘

(Modified March Algorithm Considering NPSFs)

金 泰 亨 *, 尹 秀 文 *, 朴 成 柱 *

(Tae-Hyung Kim, Su-Mun Yun, and Sung-Ju Park)

요 약

기존의 March 알고리즘으로는 내장된 메모리의 CMOS ADOFs(Address Decoder Open Faults)를 점검할 수 없다. 번지 생성 순서 및 데이터 생성을 달리 할 수 있다는 자유도(DOF: Degree of Freedom)에 근거한 수정된 March 알고리즘이 제안되었다. 본 논문에서는 번지생성기로 완전 CA(Cellular Automata)를, 데이터 생성기로 RI-LFSRs(Randomly Inversed LFSRs)을 사용하여 수정된 March 알고리즘을 개선하였다. 본 알고리즘은 기존의 March 알고리즘에서 점검할 수 있었던 SAF, ADF, CF, TF, 및 CMOS ADOF의 완점점검은 물론, NPSFs(Neighborhood Pattern Sensitive Faults)도 추가로 점검할 수 있으며, 알고리즘의 복잡도는 O(n)을 유지한다.

Abstract

The original March algorithms cannot detect CMOS ADOFs(Address Decoder Open Faults) which requires separate deterministic test patterns. Modified March algorithm using DOF(Degree of Freedom) was suggested to detect these faults in addition to conventional stuck faults. This paper augments the modified march test to further capture NPSFs(Neighborhood Pattern Sensitive Faults). Complete CA(Cellular Automata) is used for address generation and RI-LFSRs(Randomly Inversed LFSRs) for data generation. A new modified March algorithm can detect SAF, CF, TF, CMOS ADOFs, and part of NPSFs. Time complexity of this algorithm is still O(n).

Keyword : Memory test, March Alogorithm, CMOS ADOF, NPSF, DOF

Ⅰ. 서 론

반도체 소자의 크기가 지속적으로 줄어들면서[1], 수백 만개의 소자로 구성된 내장형 메모리가 SOC (Systems On a Chip)에서 중요한 요소가 되고있다. 칩의 상당 부분을 차지하고 있는, 대용량의 내장형 메모리를 포함하는 SOC는 메모리 테스터와 논리 테스터를 별도로 사용하여 테스팅 할 수 있다. 그러나 이는 테스트 시간 및 고장점검도 관점에서 현실적으로 적용하기 어렵기 때문에[2], BIST(Built-In Self-Test) 기법이 내장형 메모리 테스팅 방법으로 각광 받고 있다. BIST 구현에 적합한 알고리즘의 복잡도는 O(n)이며, 보통 March 알고리즘이 많이 쓰인다. March 알고리즘은 고전적인 고장(classical fault : ADF, SAF, CF, 및 TF)은 점검 할 수 있으나, NPSF 및 비 고전적인 고장인 CMOS ADOF를 점검 하기 어렵다[3-5].

주소 생성 방법과 데이터 생성방법을 달리한 수정된 March 알고리즘은 CMOS ADOF를 점검할 수 있는데[3], 본 논문에서는 LFSR보다 임의성이 더 큰 CA[6,7]를

* 正會員, 漢陽大學校 電子計算學科
(Dept. of Computer Science & Engineering, Hanyang Univ.)

※ 이 논문은 (1998)년 한국학술진흥재단의 학술연구비에 의하여 지원되었음.

接受日字:1999年12月3日, 수정완료일:2000年3月21日

이용하여 주소를 생성하였다. NPSF 고장을 점검하기 위하여, 5 비트와 16 비트의 RI-LFSR을 테스트 패턴 생성기로 사용하였다. 결정적인 방법으로 NPSF고장을 점검하기 위해서는 195N의 테스트 데이터가 필요한데[8], 본 논문에서 제안하는 수정된 March 알고리즘에서는 고전적인 고장, 비 고전적인 고장인 CMOS ADOFs 및 NPSF를 추가로 점검할 수 있다는 장점이 있다. 이 알고리즘은 여전히 O(n)의 복잡도를 가지므로 내장형 메모리의 BIST구현에 적합하다.

본 논문은 다음과 같이 구성되어 있다. Ⅱ장에서는 March 알고리즘과 메모리 고장에 대해 기술하였고, Ⅲ장에서는 주소 생성을 위해 사용된 완전 CA의 원리와 구현에 대해 기술하였고 Ⅳ 장에서는 데이터 생성을 위해 사용한 RI-LFSR의 특성 및 구현 방법에 대해서 기술하였다. Ⅴ 장에서는 실험 결과를 보이며, Ⅵ 장에서는 결론을 기술하였다.

Ⅱ. March 알고리즘과 메모리 고장

March 알고리즘은 알고리즘 1과 같이 March Element들로 구성된다. 알고리즘 2는 March Element

March Element

$$M_0\ \beta(w0)^{M_1} \uparrow (r0w1)^{M_2} \uparrow (r1w0)^{M_3} \downarrow (r0w1)^{M_4} \downarrow (r1w0)^{M_5}\ \beta\ (r0)$$

Address Sequence operator

알고리즘 1 : 10 N MARCH C 테스트

Algorithm 1 : 10 N March C test

```
{March element ⇑(r0, w1)}
For cell := 0 to n-1 do
begin
    read A[cell]; {Expected value = 0}
    write 1 to A[cell];
end;
{March element ⇓(r1, w0)}
For cell := 0 to n-1 do
begin
    read A[cell]; {Expected value = 1}
    write 0 to A[cell];
end;
```

알고리즘 2 : March 테스트 구성요소 {⇑(r0,w1), ⇓(r1,w0)}

Algorithm 2 : March test Element {⇑(r0,w1), ⇓(r1,w0)}

를 보여 주는데, 순방향(⇑) 혹은 역방향(⇓)으로 주소생성이 이루어지고, 0 혹은 1의 테스트 데이터를 읽고 씀으로, 메모리 셀 및 주소 해석기에 고장이 있는지 점검하는 알고리즘이다.

기존의 March 알고리즘으로 점검이 가능한 고전적인 고장은 다음과 같다.

1) Stuck-at-Fault(SAF) : 셀 혹은 선이 논리적으로 0 혹은 1로 고정됨.
2) Address Decoder Fault(ADF) : 어떤 주소로는 Access되는 셀이 하나도 없거나 이와는 반대로 어떤 셀은 Access할 수 있는 주소가 없는 경우. 또는 어떤 주소로는 여러 개의 셀이 Access되거나 이와는 반대로 어떤 셀은 여러 주소로 Access 되는 경우.
3) Transition Faults(TF) : 셀 혹은 선에서 0 -> 1 혹은 1 -> 0 천이가 불가능 함.
4) Coupling Faults(CF) : 하나의 셀에 0 -> 1(혹은 1 -> 0) 쓰기를 할 때 다른 셀의 내용이 바뀌는 경우.

일정한 방향으로의 주소생성 및 동일한 데이를 사용하는 기존의 March 알고리즘은 알고리즘의 특성상 NPSF고장 및 비 고전적 고장(Non-classical fault)은 점검하지 못하는 특징을 가진다. NPSF고장은 다음과 같다.

5) Neighborhood Pattern Sensitive Fault(NPSF) : 일정한 이웃셀의 패턴이 기본셀의 천이(0->1 혹은 1->0)에 영향을 미치는 정적 NPSF와 이웃셀의 천이에 따라서 기본셀의 내용이 바뀌게 되는 동적 NPSF로 분류함.
6) CMOS Address Decoder Open Fault(CMOS ADOF) : 메모리 주소 해석기의 CMOS 논리 게이트의 open 결함(defect)으로 야기되며, 순차적인 동작특성 때문에, 메모리 자체의 고장으로 mapping 할 수 없음.

다음은 CMOS 주소 생성기에서 발생한 open 고장에 대한 테스트 패턴 생성 방법에 대하여 상세히 살펴 보기로 하자[3,5].

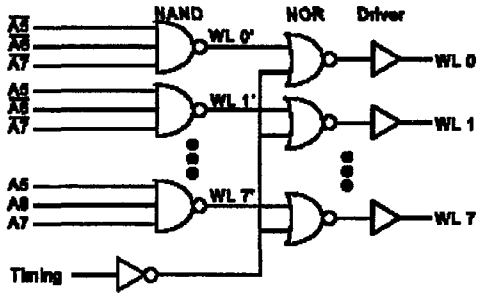


그림 1. NAND 게이트를 사용한 주소 디코더
Fig. 1. Address decoder using NAND gate.

그림 1은 [5]에서 사용된 주소해석기 이며, 그림 2는 주소 해석기에 사용되는 NAND 게이트의 트랜지스터 수준에서의 상세도이며, 오른쪽 상단에 open 결함이 나타나 있다. 알고리즘 3은 CMOS ADOF 고장시의 동작과 CMOS ADOF를 점검하기 위한 패턴을 나타낸다.

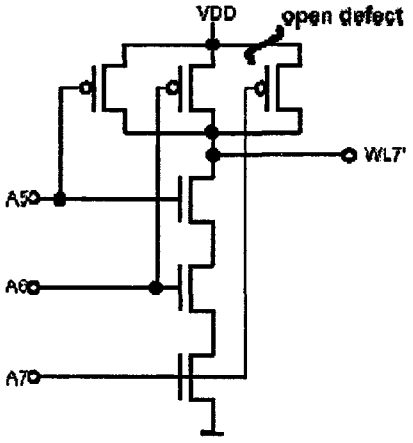


그림 2. NAND 게이트 P-channel 트랜지스터의 Open 결함
Fig. 2. Open defect of P-channel transistor of NAND gate.1

제안된 논문[3]에서는 주소 생성 순서 및 데이터 생성 방법을 변경할 수 있다는 자유도 I, IV(표 1)에 근거한 수정된 March 알고리즘으로 고전적인 고장 및 CMOS ADOF를 점검 할 수 있도록 하였다.

본 논문에서는, 수정된 March 알고리즘에서 사용된 주소 생성기인 LFSR보다 임의성이 더 큰 CA를 주소 생성기로 사용하였고, 테스트 데이터를 위해서는 NPSF를 점검하기 위해 RI-LFSR을 사용하여, 기존의 March 알고리즘에서 점검 할 수 있었던 고장외의 NPSF를 점검 하도록 하였다. 주소생성 및 테이터 생성에 대해서는 다음 장에서 상세히 설명하겠다

step 1 : 첫 번째 패턴으로 워드라인 WL7'을 활성화시키고, 선택된 메모리 워드에 데이터 D를 적는다.
pattern 1: {A7, A6, A5} = {1, 1, 1}; write data D
Effect 1 : 결함에 의해 디코더의 동작은 영향을 받지 않는다.
step 2 : 두 번째 패턴에서 A7 주소 비트만 바꾼다. WL3에 의해 활성화 되어진 메모리 워드에 $\overline{D}$ 를 기록한다.
pattern 2: {A7, A6, A5} = {0, 1, 1}; write data $\overline{D}$
Effect 2 : 워드라인 WL'7은 로직 값 '1'로 되지 않고, 이전의 '0' 값을 유지 한다. $\overline{D}$ 는 결함으로 인해 WL7에 의해 선택된 메모리에도 기록된다.
step 3 : 세 번째 패턴에서, 잘못된 데이터 $\overline{D}$ 가 WL7에 의해 선택된 메모리 워드에서 읽혀진다. pattern 3 : {A7, A6, A5} = {1, 1, 1}; read data $\overline{D}$
Effect 3 : 기대했던 데이터 D 대신 $\overline{D}$ 가 WL7에 의해 선택된 메모리 워드에서 읽혀지므로, CMOS ADOF 고장을 점검 할 수 있다.

알고리즘 3 : CMOS ADOF의 효과 및 CMOS ADOF를 점검하기 위한 패턴
Algorithm 3 : Effect of CMOS ADOF and test data to detect CMOS ADOF

표 1. 자유도 I, IV
Table 1. Degree of Freedom I, IV.

DOF I	모든 주소가 정확히 한 번 생성되면, 주소의 생성 순서가 변하여도 목표고장에 대한 점검도는 변화가 없다.
DOF IV	모든 주소의 메모리 각 셀에 0 또는 1의 일률적인 테스트 패턴을 사용하지 않아도 목표 고장 점검도에는 변화가 없다.

Ⅲ. 주소 생성을 위한 완전 CA

LHCA(linear hybrid CA)는 LCA(linear cellular automata)의 변환 행렬(transition matrix)과 LFSR를 이용하여 구현한다. 그렇기 때문에 LHCA는 PRPG (pseudo random pattern generator)처럼 다양한 방법으로 BIST를 설계하는데 사용되는 LFSR[9]과 같은 기능을 수행한다. 어떤 특정한 셀의 다음 상태는 그 특정한 셀의 현재 상태와 그 좌우 셀의 현재 상태의 조합으로 결정된다는, 즉 독립적인 LFSM(linear finite state machine)을 구현하는 방법이 LHCA이다. 이 LHCA는 보통 CA(Cellular Automata)로 불린다[6,7].

각 셀의 다음 상태를 결정하는 규칙은 다음과 같다. 먼저, CA의 규칙에 대해 서술하면, 3비트로 구성할 수 있는 8가지의 조합을 오른쪽에서 왼쪽으로의 오름차순에 따라서 배치를 한다. 3비트 중 둘째 비트가 특정 셀의 현재 상태를 나타내고, 나머지 두 비트가 그 좌우 셀의 현재 상태를 나타낸다. 그리고 3비트 조합의 결과는 각각의 3비트 배치 밑에 써넣는다. 이렇게 써넣은 8비트의 나열을 '0'에서부터 '255'사이의 2진수 값으로 하여 각 규칙의 이름으로 명명하게 된다.

표 2. CA Rule 의 명명
Table 2. Naming of CA Rule.

	7	6	5	4	3	2	1	0
	111	110	101	100	011	010	001	000
Rule 90	0	1	0	1	1	0	1	0
Rule 150	1	0	0	1	0	1	1	0
	128	64	32	16	8	4	2	1

표 2는 가능한 조합들 중에서 2가지의 예를 든 것이다. Rule 90에서의 다음 상태는 좌우의 현재 상태 값의 modulo-2 sum 연산에 의하여 얻은 것이고, Rule 150에서의 다음 상태는 셀의 현재 상태와 좌우의 현재 상태의 modulo-2 sum 연산에 의해서 얻은 것이다. 그리고 이 예는 CA를 구현하는 모든 경우에 대해서 주로 적용된다. 그림 3은 CA Rule 90을 MyCAD상에서 구현 한 것이다.

CA는 Rule 90과 Rule 150의 두 가지 규칙을 통해 변이 행렬로 나타낼 수 있다[6,7]. 이 행렬은 최대의 가지수를 가지는 회로의 조합을 나타낸 것인데, 이 조합으로 구현된 CA의 패턴을 살펴보면 2^n-1가지 경우의 패턴이 생성된다. 그러나 주소 생성을 위해서는 2^n가지 경우의 패턴이 필요하므로, 완전 CA(Complete Cellular Automata)를 구성해야만 한다.

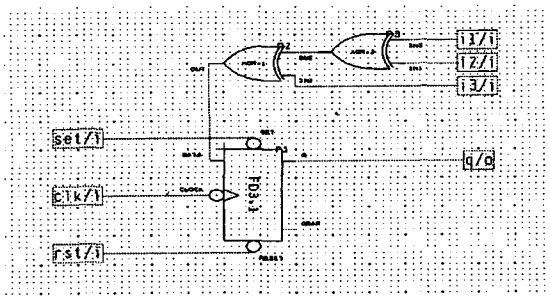


그림 3. CA Rule 150의 구현
Fig. 3. Implementation of CA Rule 150.

1. 2^n-1가지의 PRPG에서 회로 전체를 초기화하여 파형(wave form)을 관찰한다.
2. reset 직후의 패턴과 패턴이 다시 반복되는 마지막의 패턴을 기억한다.
3. 기억한 두 개의 패턴을 회로에 적용한다.
   1) 마지막의 패턴에 대해서만 1의 출력이 나오도록 회로를 구성한다.
   2) 1)에서 구성한 회로를 이용하여 reset 직후의 패턴이 1인 부분만 선택한다.
   3) 2)에서 선택된 stage의 입력에 1)에서의 입력을 주어 마지막의 패턴 다음에 모든 비트가 0이 나오도록 한다.

알고리즘 4 : 완전 CA를 만드는 과정
Algorithm 4 : Routine for Complete CA

주소 생성을 위한 완전 CA를 구성하기 위해서는 알고리즘 4의 과정을 적용해서 회로를 바꾸어 주어야 하는데, 알고리즘 4는 적용 예는 아래와 같다.

기존의 CA에서의 2^n-1가지의 패턴을 관찰하면, 6비트 CA의 예에서는 63가지의 패턴이 생성되게 되는데 '000000'의 패턴은 나오지 않는다. 그리고 '000000'의 패턴을 입력하여 적용한다면 '000000'의 패턴이 무한 반복된다. 그래서 '000000'일 때에, 그 다음의 패턴으로 '100000'을 회로에 적용하면 '000000'인 패턴은 처음의

한 주기에만 나오고 그 이후의 주기에는 반복되어 나오지 않는다. 그러나 '100000'인 패턴은 두 번째 주기 이후에서 주기의 처음 패턴으로 반복되어 생성된다. 모든 주기에서의 마지막 패턴의 다음 패턴으로 '000000'이 되도록 만들어 주어서 회로에 적용하면 모든 주기에서 '000000'인 패턴이 생성되므로 2^n가지의 패턴이 만들어진다.

완전 6비트 CA에서의 2^n가지의 패턴인 64가지의 패턴을 보면 '000000', '100000', '110100', '101100', ..., '101111', '011001'으로 하나의 주기를 구성하게 된다. CA의 규칙들을 이용하고, 알고리즘 4의 과정을 통해, 6비트 완전 CA를 구현하면, 그림 4와 같다.

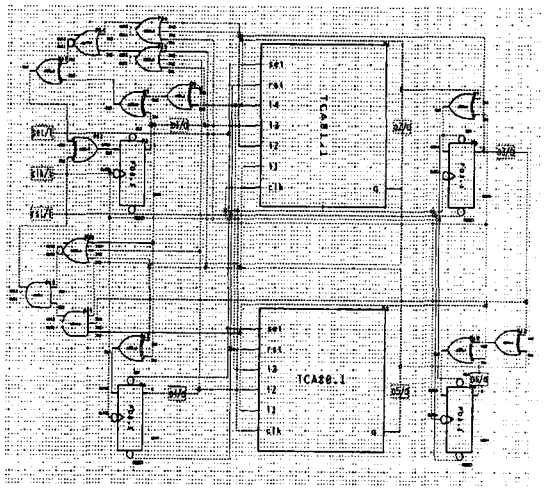


그림 4. 6 비트 완전 CA
Fig. 4. 6 bits Complete CA.

자유도 I에 의해 모든 주소가 정확히 한 번만 생성된다면, 주소 생성 순서는 자유롭게 선택 할 수 있는데, 본 논문에서는 2^n개의 주소를 생성해 줄 수 있는 완전 CA(Complete Cellular Automata)를 주소 생성기로 사용하였다. 그림 5와 6은 완전 CA와 완전 LFSR의 주소 생성 순서를 메모리 셀에 표시한 것이다. 완전 CA 및 완전 LFSR의 홀수 번째 비트는 열을 선택하고, 짝수 번째 비트는 행을 선택한다. 검게 표시한 부분은 바로 인접한 셀의 주소가 생성된 경우인데, 완전 CA가 완전 LFSR보다 인접 셀을 생성하는 경우가 적음을 알 수 있고, 이는 LFSR보다 임의성이 더 크다는 것을 의미한다. 그리고, 주소 생성 순서의 임의성이 클 때, NPSF점검도는 높게 나오리라고 예상되며, 실험을 통하여 확인하였다.

col / row	0	1	2	3	4	5	6	7
0	1	35	37	50	2	39	42	52
1	34	10	45	28	53	62	31	61
2	49	6	18	25	43	32	59	60
3	36	57	12	7	16	47	58	30
4	41	11	4	23	3	54	17	44
5	51	29	8	26	22	20	24	27
6	38	46	13	19	56	14	5	9
7	64	15	55	21	40	63	48	33

그림 5. 8×8 메모리에서 완전 CA의 주소 생성 순서
Fig. 5. Address sequence of complete CA in 8×8 memory.

col / row	0	1	2	3	4	5	6	7
0	1	39	37	52	2	35	42	50
1	64	63	55	33	40	15	48	21
2	38	14	13	9	56	46	5	19
3	34	62	45	61	53	10	31	28
4	41	54	4	44	3	57	17	23
5	36	47	12	30	16	11	58	7
6	49	32	18	60	43	6	59	25
7	51	20	8	27	22	29	24	26

그림 6. 8×8 메모리에서 완전 LFSR의 주소 생성 순서
Fig. 6. Address sequence of complete LFSR in 8×8 memory.

Ⅳ. 데이터 생성을 위한 RI-LFSR

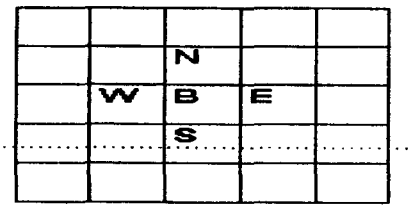


그림 7. 5 셀 NPSF
Fig. 7. Typel 5 cell NPSF.

자유도 IV에 근거해 목적 고장점검에 영향을 미치지 않으면, 데이터 값이 모두 일률적으로 같지 않아도 되므로, 본 논문에서는 랜덤한 패턴을 테스트 데이터로 사용하였다. 정적 NPSF는 그림 7에서 처럼 주위 4개의 셀('E', 'W', 'S', 'N')들이 특정한 상태일 때 기본 셀

('B')에 0 혹은 1 값을 쓸 수 없는 고장을 말한다. 이 NPSFs를 점검하기 위해서는 그림 8과 같이, 5개의 셀에 각각의 데이터 값의 상태에(2^5) 따라 32개의 패턴이 필요하다.

b	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
d_1	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
d_2	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
d_3	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
d_4	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
b	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
d_1	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
d_2	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
d_3	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
d_4	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

b = 기본 셀(Base Cell), d1, d2, d3, d4 = 인접셀(Deleted Neighbor Cell)

그림 8. 정적 NPSF의 가능한 모든 이웃 패턴
Fig. 8. All patterns of static NPSFs.

이 패턴의 생성을 위해서 5 비트 LFSR을 선택하였고, LFSR과 데이터 생성 능력은 갖지만, 생성 순서의 차이로 인해 좀 더 임의성이 큰 5비트 RI-LFSR을 테스트 패턴으로 사용하였다. RI-LFSR은 그림 10처럼, 1 비트를 천이 비트로 선택하고, 천이 비트 이외의 비트는 천이 비트와 Ex-OR연산을 통한 값을 출력한다. 그러므로, 천이된 패턴은 순서만 다를 뿐, 2^{k-1}개의 모든 패턴을 생성한다.

정리 1 : Primitive Polynomial로 구현한 길이가 K인 LFSR을 LFSR-K라고 하자. 한 비트(i)를 제외한 나머지 출력은 i의 값이 '1'일 때 반전되도록 설계한 RI-LFSR-K는 LFSR-K와 마찬가지로 2K-1개의 서로 다른 패턴을 출력한다. ■

증명 :

i 번째 출력이 다른 출력을 반전시키는 비트로 선택되었다고 가정하자. 비트 i가 '1'이 되는 패턴의 개수는 2K-1개이다. 비트 i가 '0'이면 다른 출력을 반전시키지 않으므로 '1'인 경우만 고려하면 된다. 반전된 2K-1개의 패턴은 반전되지 않았을 때의 패턴들과 순서만 다른 같은 패턴들이다. ■

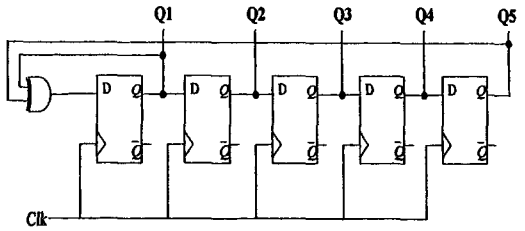


그림 9. 5 비트 LFSR
Fig. 9. 5 bits LFSR.

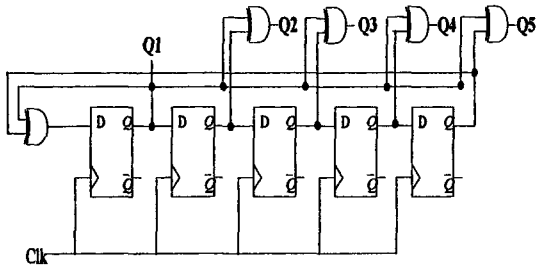


그림 10. 5 비트 RI-LFSR
Fig. 10. 5 bits RI-LFSR.

V. 메모리 고장 점검 특성 및 실험결과

자유도에 근거해, 주소생성 순서 및 데이터 생성을 달리 했을 때, SAF, AF, CF, 및 TF등의 고전적인 고장과 CMOS ADOF고장을 100% 점검하고, 나아가서 상당수준의 NPSF를 점검할 수 있었다. 각 고장에 대한 고장점검도는 C언어로 작성한 메모리 고장 시뮬레이터[10]를 이용하여 확인하였다. Tiling, Checkboard등의 결정적 및 완전 LFSR, 완전 CA등의 랜덤한 방법으로 생성된 번지에 5 비트 RI-LFSR에 의한 패턴을 주입한 결과 표 3과 같은 정적 NPSF 고장점검도를 얻을 수 있었다. NI는 LFSR을 RI는 Randomly-Inversed LFSR을 각각 나타낸다.

주소 생성 방법을 완전 CA로 하고, 테스트 데이터를 RI-LFSR을 사용하였을 때, SAF, ADF, CF, TF 및 CMOS ADOF등의 고장은 100% 점검하고, 추가로 NPSF의 24.48%를 점검할 수 있다(18 cycle일 경우). 참고로 Tiling과 Checkerboard는 고전적인 고장 점검을 보장 할 수도 없고, NPSF만 표 3의 수치만큼 점검할 수 있음을 나타낸다(18 cycle 일 경우, Tiling 27.26%, Checkerboard 21.96%, 완전 LFSR 23.52%).

표 3. 8×8 메모리에서 정적 NPSF 고장점검도
Table 3. Fault coverage of static NPSF in 8×8 memory.

Memory Size 8×8	Tiling		Checker board		Complete LFSR		CompleteCA	
	NI	RI	NI	RI	NI	RI	NI	RI
10 cycle	13.54	15.10	11.46	13.98	13.72	13.80	14.15	14.93
11 cycle	14.15	17.01	12.07	15.54	14.84	15.36	14.76	16.49
12 cycle	15.45	18.92	12.93	16.75	15.89	16.67	15.97	17.97
13 cycle	16.67	20.49	상동	17.53	17.27	17.88	16.84	18.92
14 cycle	17.53	21.61	13.54	18.23	18.58	18.75	18.32	19.53
15 cycle	18.32	22.74	14.06	19.44	19.27	19.88	18.84	20.49
16 cycle	19.01	24.74	14.15	21.18	19.79	21.88	19.62	22.31
17 cycle	19.70	26.22	14.58	21.96	20.05	22.92	20.05	23.61
18 cycle	21.09	27.26	15.10	상동	20.27	23.52	21.35	24.48
19 cycle	상동	상동	상동	상동	상동	상동	상동	상동

표 4는 결정적인 방법과 비결정적인 방법으로 주소를 생성 할 때, NPSF를 점검하기 위한 테스트 패턴의 수를 나타내는데(e는 탈출 확률, n은 메모리 셀의 수), PR(Rseudo-Random)방법이 결정적인 방법보다 더 많은 테스트 패턴을 필요로 함을 보여 준다. 표 3에서의 24.48%의 고장 점검도는 결정적인 방법인 Tiling, Checkerboard방법에 의한 고장점검도와 유사하며, 완전 LFSR보다 고장 점검도가 높음을 알 수 있다.

18주기 지나면, NPSF 점검도는 더 이상 증가하지 않는데, 이는 메모리 셀에 같은 패턴이 반복해서 들어가기 때문이다. NPSF 고장점검도를 높이기 위해 18주기까지만, 5비트 RI-LFSR을 사용하고, 그 이후에는 16비트의 RI-LFSR을 사용하였으며 표 5에 그 결과를 정리하였다.

표 4. 결정적인 테스트와 랜덤한 테스트의 비교
Table 4. Comparison of Deterministic test & Pseudo-Random test

Fault	Test Length			
	Deterministic	Pseudo-Random		
		e= 0.01	e= 0.001	e= 0.000001
ANPSF k=5	195 · n	1200 · n	1805 · n	3625 · n

표 5. 테스트 데이터로 RI-LFSR16을 사용할 때 8×8 메모리에서 정적 NPSF 고장 점검도
Table 5. Fault coverage of static NPSF in 8×8 memory using RI-LFSR16 as test data.

Memory Size 8×8	Tiling		Checker board		Complete LFSR		Complete CA	
	NI	RI	NI	RI	NI	RI	NI	RI
10 cycle	15.10	14.67	15.28	14.93	15.28	14.06	16.23	15.71
20 cycle	25.09	25.87	26.04	26.74	25.17	24.91	25.35	27.08
30 cycle	28.99	34.98	30.56	37.85	30.21	34.11	29.86	36.89
40 cycle	34.90	42.45	37.23	47.57	35.33	42.36	36.02	46.61
50 cycle	38.19	48.09	42.62	54.51	39.32	48.09	40.02	52.52
60 cycle	41.32	52.86	46.88	59.64	44.10	53.99	43.49	57.53
70 cycle	44.97	56.60	51.04	65.10	47.92	59.29	46.18	63.02
80 cycle	47.48	59.81	54.51	68.40	50.95	62.41	47.66	66.15
90 cycle	49.05	62.07	57.73	71.70	53.73	65.62	49.83	69.79
100 cycle	50.26	64.15	59.46	74.05	55.90	68.23	51.22	72.31
150 cycle	56.42	69.44	68.66	82.64	62.33	76.30	55.47	83.16
200 cycle	58.16	72.14	70.75	88.02	65.02	78.91	57.12	87.15
250 cycle	59.55	72.92	71.96	90.10	66.32	80.64	58.07	88.45

그림 11은 데이터 생성을 RI-LFSR16으로 했을 때의 그래프인데 , 100 주기 에서는 72.31%의 고장 점검도를 얻을 수 있고, 83%이상의 고장 점검도를 얻고 싶다면, 150주기 이상 테스트 패턴을 주입하면 된다.

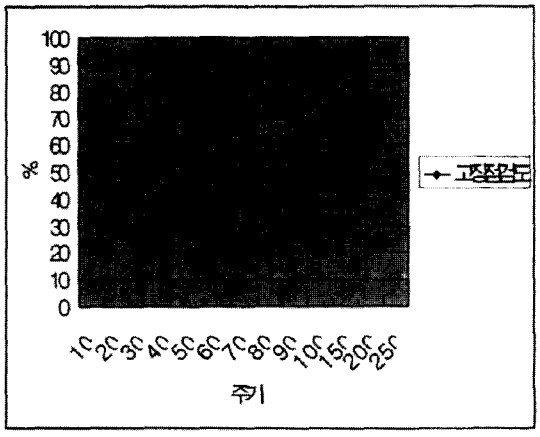


그림 11. NPSF 고장 점검도 그래프
Fig. 11. Graph for NPSF fault coverage.

Ⅵ. 결 론

March 알고리즘의 번지 생성 순서에 대한 자유도를 이용하여 본 논문에서는 완전 CA를 주소 생성을 위해 사용하였고, RI-LFSR을 데이터 생성기로 사용하였다. 완전 CA는 완전 LFSR보다 임의성이 더 크고, 천이 비트에 의한 반전이 있는 RI-LFSR 또한 LFSR보다 임의성이 크므로, NPSF 고장 점검도가 더 높게 나왔다.

본 논문에서 제안한 새로 수정된 March 알고리즘은 기존의 March 알고리즘이 점검 할 수 있던 SAF, AF, CF, 및 TF등의 고전적인 고장이외에, CMOS ADOF고장을 100% 점검하고, 상당 수준의 NPSF를 점검 할 수 있다.

본 논문에서 제안하는 개선된 March 알고리즘은 내장된 메모리의 BIST 구현에 추가영역 관점에서 적합하다고 사려된다.

참 고 문 헌

[1] Semiconductor Industry Association (SIA), The National Technology Readmap for Semiconductors, 1997.

[2] R. W. Bassett et al., "Boundary-scan design principles for efficient LSSD ASIC testing", IBM J. Res.. Develop., vol. 34, no. 2/3, pp.339-353, 1990.

[3] J. Otterstedt, D. Niggemeyer, & T.W. Williams, "Detection of CMOS Address Decoder Open Faults with March and Pseudo Random Memory Tests," ITC *conf., Oct. 1998, pp.53-62.*

[4] D. Niggemeyer, M. Redeker, and J. Otterstedt, "Intergration of Non-classical Faults in standard March Tests", Records of the IEEE Intl. WorkShop on Memory Technology, Design and Testing 1997, pp.27-32, San Jose, USA.

[5] M. Sachdev, "Open Defects in CMOS RAM Address Decoders," *IEEE Design & Test of Comp., vol. 14, no. 2, pp.26-33, 1997.*

[6] P. H. Bardell, "Analysis of Celluar Automata Used as Pseudorandom Pattern Generators," Proceedings of International Test Conference, 1990, pp. 762-768.

[7] K. Cattell, S. Zhang, M. Serra, and J, C. Muzio, "2-by-n Hybrid Cellular Automata with *Regular Configuration : Theory and* Application," IEEE Trans. *on Computers,* vol. 48, Mar. 1999, pp. 285-295.

[8] A. J. Van de Goor, "Testing Semiconductor Memories," Theory and practice, John Wiley and sons; Chichester : UK, 1991.

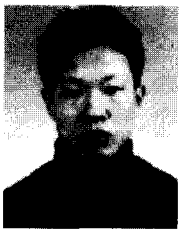
[9] E. J. McCluskey, S. Bozorgui-Nesbat, "Design for Autonomous Test," IEEE *Trans. on Computers, vol. 30, no. 11, 1981.*

[10] 박종욱, 박경택, 조상욱, 박성주, "고밀도 메모리 테스트를 위한 개선된 램덤 BIST의 비교분석," 한국정보과학회 가을 학술발표논문집, Vol. 24, NO. 2, 1997년 10월, pp.733-736

저 자 소 개

金 泰 亨(正會員)

1998년 동아대학교 컴퓨터공학과 학사. 2000년 2월 한양대학교 전자계산학과 석사. 2000년 3월~현재 한양대학교 전자계산학과 박사과정. 주관심분야는 설계자동화, VLSI 시스템 & 테스팅, ASIC 설계 등



尹 秀 文(正會員)

1999년 2월 한양대학교 전자계산학과 학사. 1999년 3월~현재 한양대학교 전자계산학과 석사과정. 주관심분야는 VLSI 시스템 & 테스팅, ASIC 설계, Device Driver 등

朴 成 柱(正會員)

1983년 한양대학교 전자공학과 학사. 1983년~1986년 금성사 소프트웨어개발. 1988년 Univ. of Massachusetts 전기 및 컴퓨터공학과 석사. 1992년 Univ. of Massachusetts 전기 및 컴퓨터공학과 박사. 1992년~1994년 IBM Microelectronics 연구스텝, 1995년~현재 한양대학교 전자컴퓨터공학부 부교수. 주관심분야는 테스트 합성, BIST, SCAN Design, ASIC 설계, 고속 신호처리 시스템 설계 등