내장된 메모리 테스트를 위한 랜덤 BIST의 비교분석

김태형, 윤수문, 김국환, 박성주
한양대학교 전자계산학과

An Analysis of Random Built-In Self Test Techniques for Embedded Memory Chips

Taehyung Kim, Sumun Yun, Kukhwan Kim, and Sungju Park
Dept. of Computer Science & Engineering
Hanyang University

요 약

메모리 테스트는 Built-In Self Test(BIST)와 같이 메모리에 내장된 회로를 통하여 자체 점검하는 방법과 테스터를 통하여 생성된 패턴을 주입하는 방법이 있다. 테스트 패턴 생성방법으로는 각각의 고장모델에 대한 테스트 패턴을 deterministic하게 생성해주는 방법과 Pseudo Random Pattern Generator(PRPG)를 이용하여 생성하는 경우로 구분할 수 있다. 본 연구에서는 PRPG를 패턴 생성기로 사용하여 여러 가지 메모리으 결함을 대표한다고 볼 수 있는 Static 및 Dynamic Neighborhood Pattern Sensitive Fault(NPSF) 등 다양한 종류의 고장을 점검할 수 있도록 메모리 BIST를 구성하였다. 기존의 Linear Feedback Shift Register(LFSR)보다 본 연구에서 제안하는 Linear Hybrid Cellular Automata (LHCA)를 이용한 PRPG가 높고 안정된 고장 점검도를 나타내었다.

1. 서론

메모리 내부에 구현되는 BIST 회로는 테스트 패턴 생성기와 결과 압축회로로 구성되어 있으며 대용량 메모리의 점검을 위하여 여러 개의 메모리 블록을 동시에 병렬로 테스트 할 수 있도록 설계된다[1]. 각각의 블록 내부에서도 여러 개의 비트선들을 동시에 처리할 수 있도록 설계함으로써[2] 테스터 및 BIST의 테스트 속도를 더욱 더 증진시키고 있다. PRPG는 Primitive Polynomial로 Linear Hybrid Cellular Automata (LHCA)를 구현하여 각 셀에서 '0'과 '1'을 균등한 확률로 생성하게 한다[3,4,5,6]. 이러한 LHCA에 의한 메모리 테스트는 Off-Chip과 On-Chip으로 사용할 수 있으며 Markov-Chain을 통한 분석적인 방법으로 NPSF 고장 점검도를 확인하였다[7,8].

본 연구에서는 기존의 PRPG를 메모리 테스트에 적합하도록 재 설계하여 NPSF의 점검도를 극대화시킨다. 개선된 PRPG에 의한 테스트 패턴을 Checkerboard 번지생성 방법[9]으로 주입하는 BIST 회로를 Synopsys CAD tool에서 VHDL로 구현하고 합성 및 시뮬레이션을 수행하였다.

본 논문은 다음과 같이 구성되어 있다. 제 2장에서는 메모리 고장모델과 테스트 패턴 생성 알고리듬을 살펴본다. 제 3장에서는 LHCA에 대하여 살펴보고 제 4장에서는 PRPG 및 개선된 PRPG를 BIST의 패턴 생성기로 사용하는 방법에 대하여 설명한다. 제 5장에서는 시뮬레이션 결과를 보여주며 결론 및 향후 연구계획을 마지막 장에서 기술한다. 메모리 내부회로로서 구현되는 BIST는 테스트패턴 생성기와 결과 압축회로로 구성되어 있으며 대용량메모리의 점검을 위하여 여러 개의 메모리 블록을 동시에 병렬로 테스트할 수 있도록 설계된다[1]. 각각의 메모리 블록내부에서도 여러 개의 비트선들을 동시에 처리할 수 있도록 설계함으로써[2] 테스터 및 BIST의 점검속도를 더욱 더 증진시키고 있다. PRPG는 LFSR을 Primitive Polynomial로 구현하여 각 셀에서 '0'과 '1'을 균등한 확률로 생성하게 한다. 이러한 LFSR에 의한 메모리테스트는 분석적인 방법인 Markov Chain을 통해 NPSF 고장 점검을 위한 랜덤 테스트패턴의 개수를 확인할 수 있다[7,8].

2. 메모리 고장모델

메모리는 데이터를 저장하는 비트 셀들과 row / column address decoder, read / write controlfhf 구성되어 있으며 이러한 메모리에서 발생 가능한 고장의 종류는 다음과 같다[10].

1) Stuck-At-Fault(SAF)
: 셀 혹은 선이 논리적으로 0 혹은 1로 고정됨.
2) Transition Fault(TF)
: 셀 혹은 선에서 0→1 (혹은 1→0) 천이가 불가능함.
3) Coupling Fault(CF)
: 하나의 셀에 0→1 (혹은 1→0) 쓰기를 할 때 다른 셀의 내용이 바뀜. k-1개의 다른 셀의 내용이 바뀔 때 k-coupling fault로 일반화하며 0→1 천이가 다른 셀의 내용을 역으로 바꾸는 경우와 일정한 값으로 바꾸는 경우에 따라서 Inversion coupling fault와 Idempotent coupling fault로 구분됨.
4) Address Decoder Fault(ADF)
: 어떤 주소로는 access 되는 셀이 하나도 없거나 이와는 반대로 어떤 셀은 access 할 수 있는 주소가 없는 경우. 또는 어떤 주소로는 여러 개의 셀이 access 되거나 이와는 반대로 어떤 셀은 여러 주소로 access 되는 경우.
5) Neighborhood Pattern Sensitive Fault(NPSF)
: 일정한 이웃 셀의 패턴이 기본 셀의 천이(0→1 혹은 1→0)에 영향을 미치는 Static NPSF와 이웃 셀의 천이에 따라서 기본 셀의 내용이 바뀌게 되는 Dynamic NPSF로 분류됨.

메모리칩의 양산시 수율을 높이기 위하여 공정 시에 발생하는 문제점을 정밀 분석한다. Mask layout 수준에서 공정 시에 가장 자주 발생하는 오점을 분류한 결과를 보면[11] 전체의 65% 정도가 'Junction leakage between storage and substrate' 결함이며 나머지도 대부분이 'leakage' 결함임을 알 수 있다. 이러한 물리적인 오점을 기능적인 고장으로 모델링 하여 보면 Coupling 혹은 Pattern Sensitive Fault(PSF)로 구분할 수 있을 것이다. 본 연구에서는 공정상의 물리적인 오점을 가장 잘 모델링 한다고 볼 수 있는 Pattern Sensitive Fault를 점검하는 테스트 패턴을 생성하는데 주안점을 두었다.

3. Linear Hybrid Cellular Automata(LHCA)

CA는 pseudo-random-pattern-generator처럼 다양한 방법으로 built-in-self-test를 설계하는데 사용되는 linear feedback shift register(LFSR)과 같은 기능을 수행 할 수 있다[3]. 어떤 특정한 cell의 다음 상태는 그 특정한 cell의 현재 상태와 그 좌우 cell의 현재 상태의 조합으로 결정된다는, 즉 독립적인 linear-finite-state-machine을 구현하는 방법이 linear-hybrid-cellular-automata(LHCA)이다[4].

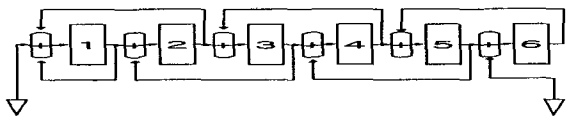
이 CA는 Wolfram에 의해서 연구되었고, 각 cell의 다음 상태를 결정하는 규칙은 다음과 같이 명명된다[5]. 규칙에 대해 서술하면, 3비트로 구성할 수 있는 8가지의 조합을 오른쪽에서 왼쪽으로의 오름차순에 따라서 배치를 한다. 3비트 중 둘째 비트가 특정 cell의 현재 상태를 나타내고, 나머지 두 비트가 그 좌우 cell의 현재 상태를 나타낸다. 그리고 3비트 조합의 결과는 각각의 3비트 배치 밑에 써넣는다. 이 써넣어진 8비트의 나열을 '0'에서부터 '255'사이의 2진수 값으로 하여 각 규칙의 이름으로 명명하게 된다. [그림 1]에서는 가능한 조합

	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

[그림 1] CA Naming of Rule

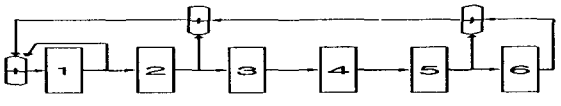
들 중에서 2가지의 예를 든 것이다. Rule 90에서의 다음 상태는 좌우의 현재 상태를 modulo-2 sum에 의해서 얻어진 것이고, Rule 150에서의 다음 상태는 cell의 현재 상태와 좌우의 현재 상태를 modulo-2 sum에 의해서 얻어진 것이다.

여기서 기존의 LFSR과 CA를 비교하여 보면 다음과 같다. CA와 LFSR은 같은 characteristic polynomial에 대하여 설계를 할 수 있고, 따라서 비슷한 성능의 pseudo random pattern generator를 구현할 수 있다. 예를 들어,



[그림 2] Linear Hybrid Cellular Automata

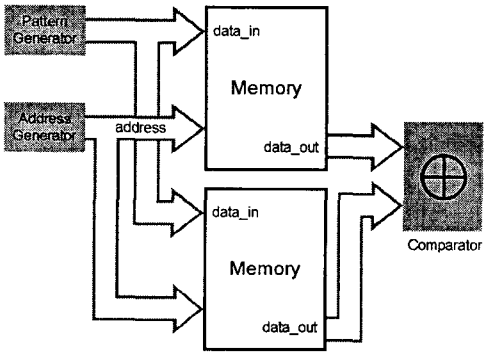
$p(x) = x^6+x^5+x^4+x+1$와 같은 characteristic polynomial에 대하여 구현을 하면 CA는 [그림 2]의 경우이고, LFSR은 [그림 3]의 경우이다.



[그림 3] Linear Feedback Shift Register

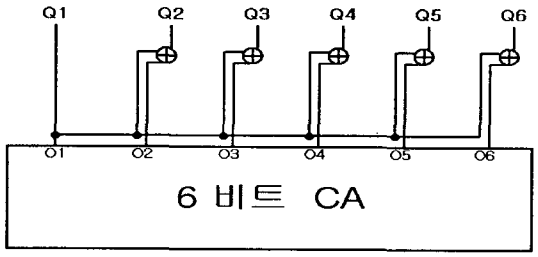
4. CA를 이용한 메모리 BIST의 구현

메모리 BIST는 크게 메모리, 패턴 생성기, 번지 생성기 및 결과 비교기로 나눌 수 있다. [그림 4]는 메모리 BIST의 블록도를 보여주고 있다. 여러 개의 메모리에 병렬로 패턴을 주입하고 비교기들을 통해 고장을 점검한다. 패턴 생성기는 [9]에서의 LFSR과 유사하게 LHCA와 Randomly Inversed-CA를 이용하였다. 번지 생성기는 Checkerboard 번지 생성 방법을 이용하였다. Checkerboard 번지 생성 방식은 메모리 셀을 두 그룹으로 나누어 한 그룹씩 쓰고 전체를 읽는 동작을 반복한다. 메모리는 Synopsys에서 제공하는 라이브러리를 이용하였다.



[그림 4] : 메모리 BIST 블럭도

기존의 PRPG를 이용한 BIST[3,4]에서는 고장 점검도가 메모리 크기 및 생성되는 패턴의 크기에 따라 크게 달라질 수 있다는 문제점이 있다. 그래서 기존의 PRPG의 임의의 한 출력 단을 다른 출력 단들과 서로 XOR하여 비 주기적으로 출력 단의 값을 반전시키도록 설계된 Randomly Inversed를 추가적으로 적용하여 BIST의 PRPG로 사용하였다. [그림 5]에



[그림 5] 6 비트 RI-CA

6 비트 RI-CA의 블록도를 보여주고 있다. [표 1]은 6 비트 CA와 6 비트 RI-CA의 각각 독립적인 64개 출력을 보여주고 있다.

CA-6	RI-CA-6
000001 001010 110000 011110	000001 001010 101111 011110
000010 010001 011000 110011	000010 010001 011000 101100
000101 101010 111100 011111	000101 110101 100011 011111
001000 100001 000110 110001	001000 111110 000110 101110
010100 110010 001111 011010	010100 101101 001111 011010
100010 011101 011001 111001	111101 011101 011001 100110
110101 110100 111110 001110	101010 101011 100001 001110
010000 010010 000011 011011	010000 010010 000011 011011
101000 101101 000111 111011	110111 110010 000111 100100
100100 101100 001101 001011	111011 110011 001101 001011
111010 101110 011100 010011	100101 110001 011100 010011
001001 101011 110110 101111	001001 110100 101001 110000
010110 100011 010111 101001	010110 111100 010111 110110
100111 110111 100101 100110	111000 101000 111010 111001
111101 010101 111000 111111	100010 010101 100111 100000
000100 100000 001100 000000	000100 111111 001100 000000

[표 1] 6 비트 CA와 6 비트 RI-CA의 출력

$$\begin{bmatrix} c_1 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 1 & c_2 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & c_3 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & c_4 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 1 & c_5 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & c_{n-1} & 1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & c_n \end{bmatrix}$$

[배열 1] *Cellular Automata*

$$\begin{bmatrix} c_1 & 1 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 1 & c_2 & \cdots & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & c_{n/2} & 0 & 0 & \cdots & 1 \\ 1 & 0 & \cdots & 0 & c_1' & 1 & \cdots & 0 \\ 1 & 0 & \cdots & 0 & 1 & c_2' & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & c_{n/2}' \end{bmatrix}$$

[배열 2] $2-by-n$ *CA*

LHCA로 구현하는 방법은 복잡하기 때문에 재귀적인 관계를 추가하여 CA를 구현하게 되면 최소한의 비용으로 maximum length cycle이 가능하게 된다[6]. 즉, 구현 방법에서 [배열 1]에서 보이는 기본적인 배열 형식을 [배열 2]에서 보이는 것처럼 배열 형식을 바꾸어 주어 구현을 하게 된다. 더불어 [배열 3]은 LFSR의 배열 형식을 보여준다.

$$\begin{bmatrix} c_1 & c_2 & c_3 & c_4 & c_5 & \cdots & c_{n-1} & c_n \\ 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

[배열 3] *LFSR*

5. 고장 시뮬레이션 결과

고장 시뮬레이션 결과, 본 연구에서 구현한 CA는 PRPG를 변화 없이 적용한 것(NI-CA)과 Randomly Inversed를 이용하여 적용한 것(RI-CA) 모두가 static 및 dynamic NPSF에서 기존의 LFSR보다 변화가 거의 없는 안정된 점검도를 보여 주었다.

메모리 크기	16×16		32×32		64×64	
	Stat.	Dyn.	Stat.	Dyn.	Stat.	Dyn.
NI-LFSR	73.98	91.75	49.95	74.10	98.30	98.23
RI-LFSR	90.56	89.76	91.80	92.23	91.25	90.71
NI-CA	90.53	90.11	93.37	93.07	91.26	90.72
RI-CA	90.69	90.38	90.16	90.65	91.27	90.72

[표 2] Static 및 Dynamic NPSF 점검도(%)

[표 2]는 다양한 크기의 메모리에 대한 static 및 dynamic NPSF 점검도를 보여주고 있다. RI-LFSR과 RI-CA는 각각 LFSR과 CA에 Randomly Inversed를 적용한 것을 나타내고, NI-LFSR과 NI-CA는 Randomly Inversed를 적용하지 않은 것을 나타낸다.

6. 결론 및 향후 연구계획

본 연구에서는 PRPG를 이용하여 내장된 메모리 테스트를 위한 BIST 패턴 생성기를 개발하였고, Checkerboard 번지 생성에 의한 랜덤 메모리 BIST 기술의 성능을 비교분석 하였다. 테스트 패턴 생성기에 이용된 CA는 각 셀의 구성이 LFSR보다 독립적으로 구성되어 있기 때문에 static 및 dynamic NPSF에 대하여 변화가 거의 없는 안정된 고장 점검도를 가질 수 있었다.

LFSR은 홀수와 짝수로만 Tapping이 가능하지만, CA는 보다 다양한 방법으로 Tapping을 할 수가 있다. 그렇기 때문에 Complete한 CA를 만들어 생성된 패턴을 Tapping하여 deterministic 하지 않는 랜덤 메모리 BIST 번지를 자동 생성하는데 이용할 수 있을 것이다.

현재 CA를 이용한 번지 생성 방법에 의한 랜덤 메모리 BIST 및 번지 디코딩 고장 시뮬레이션에 관한 연구를 진행하고 있다.

참고문헌

[1] A. J. Van de Goor, "Testing Semiconductor Memories," Theory and Practice, John Wiley and Sons, 1991.

[2] P. Mazumder and J. H. Patel, "An Efficient Built-In Self Testing For Random Access Memory," IEEE Int'l Test Conf., pp. 1072-1077, 1987.

[3] J.v. Sas, F. Catthoor, Hugo De Man, "Celular Automata Based Self-Test for Programmable Data Paths," Proc. IEEE Int'l Test Conf., pp. 769-778, 1990.

[4] P.H. Bardell, "Analysis of Cellular Automata as Pseudorandom Pattern Generators," Proc. IEEE Int'l Test Conf., pp. 762-767, 1990.

[5] C. Chen, S.K. Gupta, "BIST Test Pattern Generators for Two-Pattern Testing - Theory and Design Algorithms," IEEE Trans. Comput., vol. 45, no. 3, pp. 257-269, 1996.

[6] K. Cattell, S. Zhang, M. Serra, J.C. Muzio, "2-by-n Hybrid Cellular Automata with Regular Configuration: Theory and Application," IEEE Trans. Comput., vol. 48, no. 3, pp. 285-295, 1999.

[7] J. Savir, et al., "Testing for Coupled Cells in Random Access Memories," Proc. IEEE Int'l Test Conf., Washington D.C., pp. 439-451, 1989.

[8] P. Mazumder, "An Efficient Design of Embedded memories and their Testability Analysis using Markov Chain," Proc. Int'l Conf. on Wafer Scale Integration, pp. 389-400, 1989.

[9] 박종욱, 박경택, 박성주, "고밀도 메모리 테스트를 위한 개선된 랜덤 BIST의 구현," 대한전자공학회 하계종합학술대회 논문집 제20권, 제 1호, pp. 755-758, 1997년 6월.

[10] A. J. Van de Goor, "Using March Test to Test RAMs," IEEE Design and Test of Computer, vol. 10, no. 1, pp. 8-14, 1993.

[11] S. Oh, et. al., "Automatic Failure Analysis System for High Density DRAM," IEEE Int'l Test Conf., pp. 526-530, 1994.