論文99-36C-8-3

내장된 이중-포트 메모리의 효율적인 테스트 방법에 관한 연구

(A Study on Efficient Test Methodologies on Dual-port Embedded Memories)

韓載天*, 梁善雄*, 陳明究*, 張 勳*

(Jae-Cheon Han, Sun-Woong Yang, Myoung-Gu Jin, and Hoon Chang)

요 약

본 논문에서는 내장된 이중-포트 메모리를 위한 효율적인 테스트 알고리듬을 제안하였다. 제안된 테스트 알고리듬은 기존의 멀티-포트 메모리 테스트 알고리듬들보다 훨씬 빠르게 이중-포트 메모리를 테스트할 수 있으며, 고착 고장, 천이 고장 및 결합 고장을 완벽하게 검출할 수 있다. 또한, 본 연구에서는 제안된 알고리듬을 수행할 수 있는 BIST 회로를 Verilog-HDL을 이용하여 설계하고 시뮬레이션과 합성을 수행하였으며, BIST로 구현된 제안된 테스트 알고리듬의 높은 효율성을 다양한 크기의 내장 메모리에 대한 실험을 통하여 확인할 수 있었다

Abstract

In this paper, an efficient test algorithm for embedded dual-port memories is presented. The proposed test algorithm can be used to test embedded dual-port memories faster than the conventional multi-port test algorithms and can be used to completely detect stuck-at faults, transition faults and coupling faults which are major target faults in embedded memories. Also, in this work, BIST which performs the proposed memory testing algorithm is designed using Verilog-HDL, and simulation and synthesis for BIST are performed using Cadence Veirlog-XL and Synopsys Design-Analyzer. It has been shown that the proposed test algorithm has high efficiency through experiments on various size of embedded memories.

I. 서 론

최근 들어 하나의 칩에 대한 회로의 집적도는 시스템의 고성능화, 고기능화 및 소형화 요구와 함께 설계, 공정 기술의 발달에 힘입어 급속하게 증가되고 있다. 이에 따라, 칩에 집적시킬 수 있는 트랜지스터의 수는 제한된 면적 안에 더 많은 소자를 집적시킬 수 있게 되었으며, 칩의 기능을 더욱 향상시키기 위해 예전에는 칩 외부에 배치되었던 메모리 같은 모듈들도 이제는 하나의 칩에 내장되는 추세이다.

이와 같이 복잡해진 칩의 테스트는 갈수록 어려운 문제가 되어가고 있으며, 이로 인해 제품가격에서 테스트 비용이 차지하는 비율이 계속해서 증가하고 있다. 이렇게 고집적화된 칩의 테스트에 있어서 가장 어려운 부분 중에 하나로 여겨지는 것은 내장된 메모리(embedded memory)의 테스트이다. 메모리가 내장되기 전의 칩 테스트는 테스트 용이화 설계 기법(design for testability)과 경계주사(boundary scan) 기법을 사용하여, 칩의 회로가 더욱 복잡한 구조를 가지게 될지라도 어느 정도의 면적(area) 오버헤드를 감수하는 수준에서 만족할 만한 테스트 결과를 얻을 수 있었다.

* 正會員, 崇實大學校 電子計算學科

(School of Computing, Graduate School, Soongsil Univ.)

※ 이 연구는 99년도 한국과학재단 특정기초 연구비 지원으로 수행되었으며 지원에 감사드립니다.(과제번호 96-0102-16-01-3)

接受日字:1999年4月26日, 수정완료일:1999年7月29日

그러나, 내장된 메모리의 테스트는 메모리 자체의 기능적 특성으로 인하여 고착 고장(stuck-at fault) 모델만으로는 테스트하기 어려우며, 내장된 메모리의 입·출력 신호를 칩의 외부에서 제어(control)하거나 관찰(observe)하기 어렵기 때문에 기존의 방식으로는 테스트하기 어렵다[1, 2].

이러한 문제들을 해결하기 위해 가장 널리 사용되는 방법은 내장된 자체 테스트 기법(BIST: Built-In Self Test)이다. 자체 테스트 기법은 칩의 내부에 테스트 회로를 내장하여 자체적으로 테스트를 수행하는 기법이다.

자체 테스트 회로가 내장된 칩은 부수적으로 면적의 증가 등의 같은 오버헤드를 갖게 되지만, 다음과 같은 장점들을 갖게 된다[3].

- 각 모듈별로 자체적인 테스트가 수행되므로 전체 시스템의 테스트에 있어서 테스트의 복잡도가 크게 줄어든다.
- 각 모듈별로 적절한 BIST 회로가 내장되므로 모듈별로 가장 적합한 방식의 테스트가 가능하다.
- 고가의 외부 테스트 장치를 사용하지 않고도 빠른 시간에 테스트를 수행할 수 있다.

이런 장점들로 인하여 내장된 메모리의 테스트에 있어서는 BIST 회로의 사용이 확산되고 있다. 더욱이 내장된 메모리의 크기가 점차 커져감에 따라 자체 테스트 회로의 단점인 면적 오버헤드가 상대적으로 크게 감소하게 되므로 그 장점이 더욱 부각되고 있다.

내장된 메모리로는 리프레쉬가 필요 없는 다중-포트 비동기식(multi-port asynchronous) SRAM이 가장 많이 사용되고 있고[3], 대부분의 멀티 포트 메모리 테스트를 위한 알고리듬들은 멀티-포트 메모리를 여러 개의 단일 포트 메모리로 간주하고 각각의 포트에 대해 단일 포트 메모리 테스트 알고리듬을 적용하는 방식을 사용하고 있다[4].

본 논문에서는 이중 포트 메모리의 두개의 포트에서 동시에 테스트를 진행하여 기존의 방법들보다 더 효과적으로 이중 포트 메모리를 테스트할 수 있는 방법을 제안하고, 제안된 방법으로 Verilog HDL(Verilog Hardware Description Language)을 이용하여 자체 테스트 회로를 설계하여 Synopsys 사와 Cadence사의 툴을 사용하여 시뮬레이션 및 회로 합성을 하고 그 효율성을 검증하였다.

II. 메모리의 기능적 모델과 고장 모델

실제 메모리에서의 고장은 매우 다양한 상태로 나타나게 된다. 따라서, 메모리의 정상적인 동작에 영향을 미칠 수 있는 고장의 모든 경우에 대해서 테스트를 수행한다는 것은 실질적으로 불가능하다. 그러나 메모리 테스트의 목적은 특수한 경우를 제외하고는 고장의 유형이나 위치를 파악하기보다는, 단순히 고장의 발생유무를 파악하는 것이다[1, 4, 5, 6]. 그러므로 일반적인 메모리 테스트에서는 먼저 메모리의 구조를 기능(functional) 모델로 단순화시킨다. 이 모델에서는 메모리를 메모리 셀 배열(memory cell array), 주소 디코더(address decoder), 읽기/쓰기 회로(read/write logic)로 구성된 형태로 표현할 수 있다. 그림 1은 고장 검출만을 위해 사용되는 축소된 메모리의 기능적 모델을 보여 준다.

이러한 축소된 기능적 모델에서 발생 가능한 고장들은 발생위치에 따라 아래와 같이 나눌 수 있다. 단일 포트 메모리의 경우에는 주소 디코더와 읽기/쓰기 회로에서 발생되는 고장들은 고착-개방 고장(stuck-open fault)과 같은 일부 제한된 경우를 제외하고는 대부분 메모리 셀 배열의 고장으로 사상(mapping)시켜 테스트할 수 있다[1, 7, 8, 9, 10].

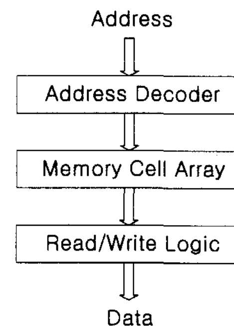

그림 1. 메모리의 축소된 기능적 모델
Fig. 1. Reduced functional model of memory.

1. 메모리 셀에서의 고장

그림 2는 SRAM 셀의 전기적 모델에서 발생할 수 있는 결함들을 보여준다[1]. 그림에서 BL(bit-line)은 비트 라인, WL(word-line)은 워드 라인을 나타내고, 발생 가능한 결함은 영문 소문자로 표시되어 있다.

- 결함 a, b c는 모두 메모리의 기능적 고장 중 stuck-at-0 고장을 일으키게 만든다.

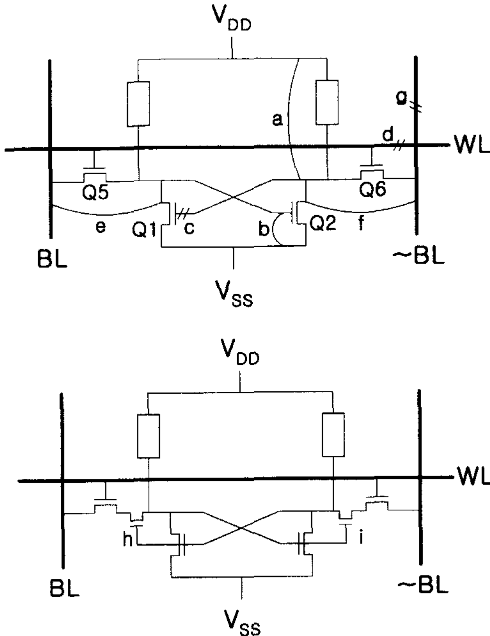

그림 2. SRAM 셀에서의 고장
Fig. 2. Fault in SRAM cell.

● 결함 d는 워드 라인이 단절된 것으로서 단절된 워드 라인 이후의 모든 셀들의 접근을 불가능하게 만든다. 이런 셀의 값을 읽어낼 때 Q5와 Q6 패스 트랜지스터가 열리지 않기 때문에 BL과 ~BL을 드라이브해 주는 것이 아무 것도 없게 되어 읽기 · 쓰기 회로의 구현 방법에 따라 0이나 1로 고정되게 된다. 따라서, 이 결함은 stuck-at-0 고장이나 stuck-at-1 고장으로 나타나게 된다.

● 논리값 0을 갖고 있는 셀에 결함 e가 발생하는 경우, Q1 트랜지스터의 패스가 열려 있기 때문에 BL에 인가되는 전하가 Vss로 풀다운(pull down)되어 이 셀에 논리값 1을 쓰는 것이 불가능하다. 만일 이 셀이 논리값 1을 갖고 있었다면 결함 e의 영향을 받지 않는다. 이 고장의 결과로 결함 e를 갖고 있는 셀이 논리값 0을 갖고 있을 때, 같은 BL 라인에 연결되어 있는 모든 셀들은 stuck-at-0 고장을 일으킨다. 또한, 이 결함을 갖고 있는 셀에는 상태 결합 고장 <0;0> 즉, 어떤 셀이 논리값 0을 갖고 있는 경우 계속 0을 갖게 되는 고장이 있는 것으로 볼 수도 있다.

● 결함 f는 같은 ~BL 라인에 연결된 모든 셀들에 stuck-at-1 고장을 발생시키게 되며, 이 결함을 갖고 있는 셀은 결함 e와 같이 결합 고장이 있는 것으로 볼 수 있다.

● 결함 g는 개방 고장 이후의 모든 셀들에 ~BL을 통해 논리값 0을 쓸 수 없음을 의미한다. 만일 개방 고장 이후의 셀이 논리값 0을 갖고 있다면, 이 값은 정확히 읽어낼 수 있다. 그러나 논리값 1을 갖고 있다면 읽기 회로의 구현 방법에 따라 읽혀지는 값이 바뀌게 된다.

● 결함 h와 i는 폴리실리콘 층의 고장으로 인해 새로운 트랜지스터가 생성된 것을 보여 주고 있다. 셀이 논리값 0을 갖고 있고 결함 h가 발생했을 때 이 셀은 BL에서 분리되어 상향 천이 고장을 일으킨다. 마찬가지로 결함 i는 하향 천이 고장을 일으킨다.

2. 주소 디코더에서의 고장

주소 디코더는 메모리 셀 어레이의 워드 라인을 선택하는 열 디코더(row decoder)와 메모리 셀 어레이(memory cell array)에서 출력되는 비트 라인 중 필요한 신호를 선택하는 행 디코더(column decoder)로 이루어져 있다. 이 부분에서 발생할 수 있는 일반적인 고장 모델은 다음과 같다[7, 8, 9, 11, 12].

1) 하나의 주소에 의해 접근되는 셀이 하나 이상이다.
2) 어떤 주소는 셀을 전혀 접근할 수 없다.

3. 읽기 · 쓰기 회로에서의 고장

읽기 · 쓰기 회로는 입출력 핀과 메모리 셀 어레이 사이에서 데이터를 넘겨주는 역할을 하는 회로로서 버스, 증폭기, 양방향 3상 버퍼 등으로 구성된다. 워드 기반의 SRAM에서 읽기 · 쓰기 회로에서 발생할 수 있는 고장은 다음과 같다[7, 8, 9, 12].

1) 하나 이상의 데이터 라인 비트가 고착(stuck-at) 된다.
2) 하나 이상의 데이터 라인 비트가 고착-개방(stuck-open)된다.
3) 데이터 라인 비트 쌍이 단락(short)된다.

Ⅲ. 이중 포트 메모리 테스트 알고리듬

이중 포트 메모리는 하나의 공통된 메모리 셀 어레이와 이를 접근할 수 있는 두 개의 입출력 회로로 구성된다. 그림 3은 2 Read / 2 Write가 가능한 이중 포트 메모리의 단순화된 내부 구조를 보여준다.

단일 포트에 비해 이중 포트 메모리에는 동시에 동일한 주소에 자료를 쓸 수 없도록 해주는 중재 회로(arbitration logic)가 추가될 수도 있으며, 중재 회로 내에서도 고장이 발생할 수 있다[1,6]. 본 논문에서는 중재 회로의 테스트는 고려하지 않는다.

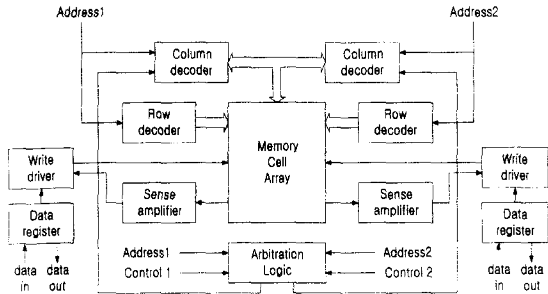

그림 3. 이중 포트 메모리의 구조
Fig. 3. Dual-port memory architecture.

대부분의 멀티-포트 메모리 테스트 알고리듬들은 멀티-포트 메모리를 여러 개의 단일-포트 메모리로 여기고 각각의 포트에 대해 독립적으로 기존의 단일-포트 메모리 테스트를 위한 알고리듬을 이용하여 테스트를 수행한다. 따라서, 기존의 멀티-포트 메모리 테스트 알고리듬을 이용하여 포트수가 P개인 멀티-포트 메모리를 시간 복잡도가 T인 단일-포트 메모리 테스트 알고리듬을 이용하여 테스트하는데는 $P \times T$의 시간이 소요된다. 한 워드의 크기가 m인 워드 단위의 멀티-포트 메모리라면 배경 데이터를 사용하여야 하므로 여기에 $\lceil \log_2 m \rceil + 1$을 곱한 만큼의 시간이 소요된다[12].

따라서, 기존의 멀티-포트 메모리 테스트 알고리듬들은 멀티 포트 메모리를 여러 개의 단일-포트 메모리의 조합으로 여기기 때문에 테스트 수행 시간이 포트 수에 비례하는 문제점을 갖고 있다.

본 연구에서는 이중-포트 메모리는 두 개의 포트를 이용하여 동시에 메모리를 접근할 수 있다는 점에 착안하여 그림 4와 같이 이중-포트에서 동시에 March 테스트를 진행하는 알고리듬을 제안한다. 그림에서 R(x)는 단일-포트 메모리에서는 0이나 1을 읽는다는 의미이며, 멀티-포트 메모리에서는 배경 데이터나 역전된 배경 데이터를 읽는다는 의미이다. 마찬가지로 W(x)는 단일-포트 메모리에서는 0이나 1을 쓴다는 의미이며, 멀티-포트 메모리에서는 배경 데이터나 역전된 배경 데이터를 쓴다는 의미이다.

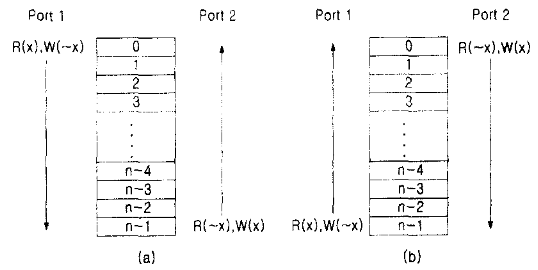

그림 4. 이중 포트를 동시에 사용하는 테스트 진행
Fig. 4. Testing the concurrently used Dual-port.

이중 메모리의 각 포트를 Port1, Port2라고 했을 때, Port 1과 Port 2는 항상 서로 반대되는 방향으로 반대되는 값을 이용하여 테스트를 진행한다. 예를 들어, Port 1에서 주소가 증가하는 방향으로 R(0), W(1)을 수행하고, 동시에 Port 2에서는 주소가 감소하는 방향으로 R(1), W(0)를 수행한다.

본 연구에서는 8비트 단위의 워드를 사용하는 이중-포트 메모리를 사용한다고 가정하였고, 테스트를 위하여 표 1과 같은 배경 데이터를 사용하여 워드 단위의 결합 고장까지도 완벽하게 검출할 수 있도록 구현하였다.

표 1. 배경 데이터
Table 1. Background data.

	W(0)	W(1)
P0	00000000	11111111
P1	01010101	10101010
P2	00110011	11001100
P3	00001111	11110000

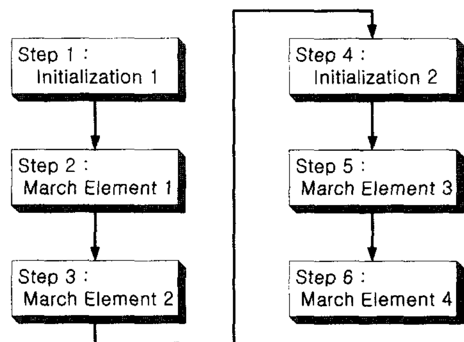

그림 5. 제안하는 March 요소
Fig. 5. Proposed March elements.

본 논문에서 제안하는 테스트 알고리듬은 그림 5와 같은 March 요소로 March 테스트를 수행한다. 제안된 알고리듬의 설명을 위해 2의 배수 크기를 갖는

n × 8 크기를 갖는 2의 배수 크기를 갖는 메모리 셀 어레이로 가정하여 이용하기로 한다.

각각의 March 요소가 하는 역할은 다음과 같다.

가) 1 단계: 초기화 1

테스트를 위해 메모리의 값을 초기화시킨다. 이 단계가 끝나면 메모리 셀 어레이는 그림 6의 (a)와 같은 값을 갖게 된다.

메모리 셀 어레이에서 0번지부터 {(n / 2) - 1} 번지까지는 Port 1을 사용하여 0으로 초기화시키고 (W(0)), {(n / 2)} 번지부터 {(n - 1)} 번지까지는 Port 2를 사용하여 1로 초기화시킨다(W(1)). 메모리를 테스트할 때 Port1과 Port2를 이용하여 서로 반대되는 값으로 테스트를 수행하기 때문에 같은 값으로 메모리 셀 어레이를 모두 초기화하지 않고 메모리를 나누어 서로 반대되는 값으로 초기화시킨다.

Port1	Add.	Memory Cell Array								Add.	Port2
W0	0	0	0	0	0	0	0	0	0	0	
	1	0	0	0	0	0	0	0	0	1	
	2	0	0	0	0	0	0	0	0	2	
	⋮				......					⋮	
	n/2-1	0	0	0	0	0	0	0	0	n/2-1	
	n/2	1	1	1	1	1	1	1	1	n/2	
	⋮				......					⋮	
	n-3	1	1	1	1	1	1	1	1	n-3	
	n-2	1	1	1	1	1	1	1	1	n-2	
	n-1	1	1	1	1	1	1	1	1	n-1	W1

(a) 초기화 1

Port1	Add.	Memory Cell Array								Add.	Port2
W1	0	1	1	1	1	1	1	1	1	0	
	1	1	1	1	1	1	1	1	1	1	
	2	1	1	1	1	1	1	1	1	2	
	⋮				......					⋮	
	n/2-1	1	1	1	1	1	1	1	1	n/2-1	
	n/2	0	0	0	0	0	0	0	0	n/2	
	⋮				......					⋮	
	n-3	0	0	0	0	0	0	0	0	n-3	
	n-2	0	0	0	0	0	0	0	0	n-2	
	n-1	0	0	0	0	0	0	0	0	n-1	W0

(b) 초기화 2

그림 6. 초기화
Fig. 6. Initialize.

나) 2 단계: March 요소 1

이 단계에서는 그림 7의 (a)와 같이 테스트를 진행한다. Port 1에서는 주소가 증가하는 방향으로 (R(0), W(1))을 수행하고, Port 2에서는 주소가 감소하는 방향으로 (R(1), W(0))을 수행한다.

Port1	Add.	Memory Cell Array								Add.	Port2
R0,W1	0	1	1	1	1	1	1	1	1	0	
	1	0	0	0	0	0	0	0	0	1	
	2	0	0	0	0	0	0	0	0	2	
	⋮				......					⋮	
	n/2-1	0	0	0	0	0	0	0	0	n/2-1	
	n/2	1	1	1	1	1	1	1	1	n/2	
	⋮				......					⋮	
	n-3	1	1	1	1	1	1	1	1	n-3	
	n-2	1	1	1	1	1	1	1	1	n-2	
	n-1	0	0	0	0	0	0	0	0	n-1	R1,W0

(a) March 요소 1

Port1	Add.	Memory Cell Array								Add.	Port2
R1,W0	0	0	0	0	0	0	0	0	0	0	
	1	1	1	1	1	1	1	1	1	1	
	2	1	1	1	1	1	1	1	1	2	
	⋮				......					⋮	
	n/2-1	1	1	1	1	1	1	1	1	n/2-1	
	n/2	0	0	0	0	0	0	0	0	n/2	
	⋮				......					⋮	
	n-3	0	0	0	0	0	0	0	0	n-3	
	n-2	0	0	0	0	0	0	0	0	n-2	
	n-1	1	1	1	1	1	1	1	1	n-1	R0,W1

(c) March 요소 3

Port1	Add.	Memory Cell Array								Add.	Port2
	0	1	1	1	1	1	1	1	1	0	R0,W1
	1	0	0	0	0	0	0	0	0	1	
	2	0	0	0	0	0	0	0	0	2	
	⋮				......					⋮	
	n/2-1	0	0	0	0	0	0	0	0	n/2-1	
	n/2	1	1	1	1	1	1	1	1	n/2	
	⋮				......					⋮	
	n-3	1	1	1	1	1	1	1	1	n-3	
	n-2	1	1	1	1	1	1	1	1	n-2	
R1,W0	n-1	0	0	0	0	0	0	0	0	n-1	

(b) March 요소 2

Port1	Add.	Memory Cell Array								Add.	Port2
	0	0	0	0	0	0	0	0	0	0	R1,W0
	1	1	1	1	1	1	1	1	1	1	
	2	1	1	1	1	1	1	1	1	2	
	⋮				......					⋮	
	n/2-1	1	1	1	1	1	1	1	1	n/2-1	
	n/2	0	0	0	0	0	0	0	0	n/2	
	⋮				......					⋮	
	n-3	0	0	0	0	0	0	0	0	n-3	
	n-2	0	0	0	0	0	0	0	0	n-2	
R0,W1	n-1	1	1	1	1	1	1	1	1	n-1	

(d) March 요소 4

그림 7. March 요소별 테스트 진행
Fig. 7. Testing the each of March elements.

그림 8은 테스트가 진행이 되면서 Port 1에서 {(n / 2) - 1} 번지를, Port 2에서 {(n / 2)} 번지를 접근한 바로 직후의 고장이 없는 메모리 셀 어레이의 값을 보여 준다.

Port1	Add.	Memory Cell Array	Add.	Port2
	0	1 1 1 1 1 1 1 1	0	
	1	1 1 1 1 1 1 1 1	1	
	2	1 1 1 1 1 1 1 1	2	
	⋮	......	⋮	
R0,W1	n/2-1		n/2-1	
	n/2		n/2	R1,W0
	⋮	......	⋮	
	n-3	0 0 0 0 0 0 0 0	n-3	
	n-2	0 0 0 0 0 0 0 0	n-2	
	n-1	0 0 0 0 0 0 0 0	n-1	

그림 8. March 요소 1 적용 중간 단계
Fig. 8. Progression of applying March element 1.

이 그림에서 알 수 있듯이 1 단계에서 {(n / 2)} 번지부터 {(n - 1)} 번지까지는 Port 2를 통해 1로 초기화되었지만 테스트가 Port 1에서 {(n / 2) - 1} 번지, Port 2에서 {(n / 2)} 번지까지 진행되면 {(n / 2)} 번지에서 {n - 1} 번지까지는 고장이 없을 경우에 모두 0을 갖게 되고, 마찬가지로, {0} 번지에서 {(n / 2) - 1} 번지까지는 모두 1을 갖게 되므로 Port 1과 Port 2에서 정해진 읽기 쓰기 방식으로 계속 테스트를 진행해 나갈 수 있다.

다) 3 단계: March 요소 2

이 단계에서는 그림 7의 (b)와 같이 Port 1에서는 주소가 감소하는 방향으로 (R(1), W(0))를 수행하고, Port 2에서는 주소가 증가하는 방향으로 (R(0),W(1))을 수행한다.

라) 4 단계: 초기화 2

3 단계가 끝난 후 메모리에 고장이 없다면 메모리 셀 어레이는 1 단계에 초기화시켰을 때와 같은 값을 갖게 되기 때문에 더 이상 테스트를 진행할 수 없고 결합 고장을 모두 검출할 수 없다. 예를 들어, March 요소 1과 March 요소 2에서는 Port 1에서 i 번지 셀의 상향 천이에 의한 j 번지 셀에서의 결합 고장을 활성화시키고($i \neq j$), Port 2에서 k 번지의 셀의 하향 천이에 의한 l 번지에서의 결합 고장을 활성화시켜($k \neq l$) 고장을 검출한다. 각각의 셀에서 천이가 발생할 때 다른 셀의 값은 항상 일정한 값을 갖고 있어서 모든 결합 고장을 검출할 수 없기 때문에 초기화 1과 반대되는 값으로 다시 초기화시켜 테스트를 진행한다. 이 단계가 끝나면 메모리 셀 어레이는 그림 6의 (b)와 같은 값을 갖게 된다.

마) 5 단계: March 요소 3

그림 7의 (c)는 March 요소 3에서의 테스트 진행을 보여준다. Port 1에서는 주소가 증가하는 방향으로 (R(1), W(0))를 수행한다. Port 2에서는 주소가 감소하는 방향으로 (R(0), W(1))를 수행한다.

바) 6 단계: March 요소 4

이 단계에서는 그림 7의 (d)와 같이 Port 1에서는 주소가 감소하는 방향으로 (R(0), W(1))를 수행하고, Port 2에서는 (R(1), W(0))를 수행한다.

Ⅳ. 제안된 방법에서의 고장 검출 및 성능 평가

그림 9는 본 논문에서 제안한 방법으로 내장된 메모리를 테스트할 때 메모리 셀 어레이의 값의 변화를 보여준다. 그림에서는 제안된 방법의 고장 검출 과정을 설명하기 위하여 8 X 1 크기, 즉 8개의 주소를 갖는 1비트 크기의 메모리 셀 어레이를 사용하였다.

그림 9에서 (a)는 초기화 1과 March 요소 1, March 요소 2가, (b)는 초기화 2와 March 요소 3, March 요소 4가 수행되는 동안 메모리 셀 어레이에서의 값의 변화를 보여주고 있다. 진하게 표시된 셀은 해당 번지가 Port 1를 통해서 접근되고 있음을 의미하고, 조금 옅게 표시된 셀은 Port 2 포트를 통해 접근되고 있음을 의미한다. 본 논문에서 제안한 방법은 Port 1과 Port 2에서 동시에 읽고 쓰며 테스트하기 때문에 각 단계에서는 두 개의 셀의 값이 동시에 변화한다. 그리고, 고장 검출 설명을 위해 각 March 요소별로 테스트가 진행되는 단계를 숫자로 표시하였다.

1. 메모리 셀에서의 고장 검출

앞에서 살펴보았듯이 메모리 셀 어레이에서는 고착 고장, 천이 고장, 결합 고장이 발생할 수 있다. 각각의 고장에 대한 검출은 다음과 같이 이루어진다[1].

가) 고착 고장

고착 고장을 검출하기 위한 테스트는 각각의 모든 셀에 0과 1을 읽고 쓸 수 있어야 한다. 본 논문에서 제안한 방법은 March 요소 1에서 Port 1과 Port 2

init1	Port 1↑ March Element 1 Port 2↓																Port 1↓ March Element 2 Port 2↑															
0	0		1		2		3		4		5		6		7		0		1		2		3		4		5		6		7	
W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W
0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
0			1	1	1	1	1	1	1	1	1	1	1	1	**1**	**0**	**0**	**1**	1	1	1	1	1	1	1	1	1	1	1	1		
0	0	0			1	1	1	1	1	1	1	1	**1**	**0**	0	0	0	0	**0**	**1**	1	1	1	1	1	1	1	1			0	0
0	0	0	0	0			1	1	1	1	**1**	**0**	0	0	0	0	0	0	0	0	**0**	**1**	1	1	1	1			0	0	0	0
0	0	0	0	0	0	0			**1**	**0**	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**			0	0	0	0	0	0
1	1	1	1	1	1	1	**1**	**0**			1	1	1	1	1	1	1	1	1	1	1	1			**0**	**1**	1	1	1	1	1	1
1	1	1	1	1	**1**	**0**	0	0	0	0			1	1	1	1	1	1	1	1			0	0	0	0	**0**	**1**	1	1	1	1
1	1	1	**1**	**0**	0	0	0	0	0	0	0	0			1	1	1	1			0	0	0	0	0	0	0	0	**0**	**1**	1	1
1	**1**	**0**	0	0	0	0	0	0	0	0	0	0	0	0					0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**
W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

(a)

init2	Port 1↑ March Element 3 Port 2↓																Port 1↓ March Element 4 Port 2↑															
0	0		1		2		3		4		5		6		7		0		1		2		3		4		5		6		7	
W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1			0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**0**	0	0	0	0	0	0	0	0	0	0	0	0		
1	1	1			0	0	0	0	0	0	0	0	**0**	**1**	1	1	1	1	**1**	**0**	0	0	0	0	0	0	0	0			1	1
1	1	1	1	1			0	0	0	0	**0**	**1**	1	1	1	1	1	1	1	1	**1**	**0**	0	0	0	0			1	1	1	1
1	1	1	1	1	1	1			**0**	**1**	1	1	1	1	1	1	1	1	1	1	1	1	**1**	**0**			1	1	1	1	1	1
0	0	0	0	0	0	0	**0**	**1**			0	0	0	0	0	0	0	0	0	0	0	0			**1**	**0**	0	0	0	0	0	0
0	0	0	0	0	**0**	**1**	1	1	1	1			0	0	0	0	0	0	0	0			1	1	1	1	**1**	**0**	0	0	0	0
0	0	0	**0**	**1**	1	1	1	1	1	1	1	1			0	0	0	0			1	1	1	1	1	1	1	1	**1**	**0**	0	0
0	**0**	**1**	1	1	1	1	1	1	1	1	1	1	1	1					1	1	1	1	1	1	1	1	1	1	1	1	**1**	**0**
W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W	R	W
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

(b)

그림 9. 메모리 셀 어레이의 값의 변화
Fig. 9. Changing status of memory cell array value.

을 통해 모든 셀들에 0과 1을 읽고 쓰므로 고착 고장을 완벽히 검출할 수 있다.

예를 들어, 그림 9의 (a)에서 March 요소 1을 살펴보면 Port 1에서 주소가 증가하는 방향으로 0을 읽고 1을 쓰고, Port 2에서 주소가 감소하는 방향으로 1을 읽고 0을 쓰므로 모든 셀들에 0과 1을 읽고 쓸 수 있다. 따라서, 메모리 셀 어레이에 존재하는 모든 고착 고장을 검출할 수 있다. 다른 March 요소들에서도 동일한 방법으로 Port 1과 Port 2에서 메모리 셀 어레이에 존재하는 고착 고장을 중복해서 검출할 수 있다

나) 천이 고장

천이 고장을 검출하기 위한 테스트는 각각의 모든 셀에 상향 천이와 하향 천이를 일으킬 수 있어야 하고, 다른 천이가 더 이상 발생되기 전에 셀의 값을 읽을 수 있어야 한다. 본 논문에서 제안한 방법은 March 요소 1과 March 요소 2에서 Port 1과 Port 2를 통해 모든 셀들에 상향 천이와 하향 천이를 발생시키고, 이를 읽기 때문에 천이 고장을 완벽히 검출할 수 있다.

예를 들어, 그림 9의 (a)를 살펴보면 March 요소 1의 0번 ~ 3번 단계에서 Port 1을 통해 {0} ~ {(n / 2) - 1} 번지의 셀들에 상향 천이가 발생하고, Port 2를 통해 {(n / 2)} ~ {(n - 1)} 번지의 셀들에 하향 천이가 발생한다. 이를 March 요소 1의 4번 ~ 7번 단계에서 각각 Port 1과 Port 2를 통해 읽어본다.

March 요소 1의 4번 ~ 7번 단계에서는 0번 ~ 3번 단계에서 발생시킨 천이에 대한 천이 고장을 확인할 뿐 만 아니라 각각의 셀에 0번 ~ 3번 단계에서 발생된 천이와 반대되는 천이를 발생시킨다. 즉, {0} ~ {(n / 2) - 1} 번지의 셀들에는 Port 2를 통해 하향 천이가 발생되고, {(n / 2)} ~ {(n - 1)} 번지의 셀들에는 Port 1을 통해 상향 천이가 발생된다. 이와 같이 March 요소 1의 4번 ~ 7번 단계에서 발생된 천이에 대한 천이 고장은 March 요소 2의 3단계까지 테스트가 진행되면 검출이 가능하고, 이와 같은 방법을 통해 메모리 셀 어레이의 모든 셀에서 발생되는 천이 고장의 완벽한 검출이 가능하다.

다) 결합 고장

결합 고장을 검출하기 위해서는 발생 가능한 모든 경우의 결합 고장을 활성화시키고, 결합된 셀에 어떤 값을 쓰기 전에 이를 읽어볼 수 있으면 된다. 그리고, 결합 고장의 특성상 한 셀에 결합 고장이 짝수개가 발생할 경우에는 검출해내지 못할 수도 있다[1]. 모든 경우의 결합 고장을 활성화시키기 위해서는 상향 천이와 하향 천이가 발생할 때 다른 메모리 셀들은 논리값 0을 갖고 있을 경우와 논리값 1을 갖고 있을 경우가 있어야만 한다.

본 논문에서 제안한 방법에서는 상향 천이와 하향 천이에 대해 천이가 발생하지 않는 모든 메모리 셀들이 논리값 0을 갖고 있는 경우와 논리값 1을 갖고 있는 경우가 모두 존재하고, 각각의 March 요소가 읽기를 수행한 뒤 쓰기를 수행하므로, 결합 고장이 있을 경우 결합된 셀의 값을 먼저 읽어 볼 수 있으므로 천이가 발생하는 셀과 발생하지 않는 셀간에 존재하는 완벽하게 결합 고장을 검출할 수 있다.

두 포트에서 접근하는 셀간에 결합 고장의 경우 결합 고장이 있다면 두 셀에 같은 논리값이 쓰여지게 된다[1,4]. 본 논문에서 제안한 방법에서는 두 포트에서 항상 서로 다른 논리값을 이용하여 테스트를 수행하므로 결합 고장이 발생된 두 셀 중 하나에는 잘못된 값이 쓰여지게 되어 검출이 가능하다. 따라서, 메모리 셀 어레이에 존재하는 결합 고장을 검출해 낼 수 있다.

먼저, {0} 번지 ~ {(n / 2) -1} 번지의 셀에 상향 천이가 발생했을 경우를 살펴보자. 그림 9를 살펴보면 March 요소 1의 0번 ~ 3번 단계와 March 요소 3의 4번 ~ 7번 단계에서 {0} 번지 ~ {(n / 2) -1} 번지에 상향 천이가 발생된다. 이때 두 포트를 통해 접근되는 셀을 제외한 다른 셀들의 값을 살펴보면 각각의 셀이 March 요소 1에서와 March 요소 3에서 서로 반대되는 값을 갖고 있음을 알 수 있다. 따라서, {0} 번지 ~ {(n / 2) -1} 번지에서의 상향 천이에 의한 결합 고장을 모두 발생시킬 수 있다. 이 고장들은 March 요소 1의 4번 ~ 7번 단계와 March 요소 4의 0번 ~ 3번 단계에서 검출이 가능하다.

두 번째로 {(n / 2)} ~ {(n - 1)} 번지의 셀에 상향 천이가 발생했을 경우를 살펴보자. {(n / 2)} ~ {(n - 1)} 번지의 셀들의 경우 March 요소 1의 4번 ~ 7번 단계에서 Port 1을 통하여 상향 천이가 발생되고, March 요소 3의 0번 ~ 3번 단계에서 Port 2를 통해 상향 천이가 발생된다. 이때 두 포트를 통해 접근되는 셀을 제외한 다른 셀들의 값을 살펴보면 각각의 셀이 March 요소 1에서와 March 요소 3에서 서로 반대되는 값을 갖고 있음을 알 수 있다. 따라서, {(n / 2)} ~ {(n - 1)} 번지에서의 상향 천이에 의한 결합 고장을 모두 발생시킬 수 있고, 이 고장들은 March 요소 2의 0번 ~ 3번 단계와 March 요소 3의 4번 ~ 7번 단계에서 검출이 가능하다.

세 번째로 {0} ~ {(n / 2) - 1} 번지의 셀에 하향 천이가 발생했을 경우를 살펴보자. {0} ~ {(n / 2) - 1} 번지의 셀들의 경우 March 요소 1의 4번 ~ 7번 단계에서 Port 2를 통해 하향 천이가 발생되고, March 요소 3의 0번 ~ 3번 단계에서 Port 1을 통해 하향 천이가 발생된다. 이때 두 포트를 통해 접근되는 셀을 제외한 다른 셀들의 값을 살펴보면 각각의 셀이 March 요소 1에서와 March 요소 3에서 서로 반대되는 값을 갖고 있음을 알 수 있다. 따라서, {0} ~ {(n / 2) - 1} 번지에서의 하향 천이에 의한 결합 고장을 모두 발생시킬 수 있고, 이 고장들은 March 요소 2의 0번 ~ 3번 단계와 March 요소 3의 4번 ~ 7번 단계에서 검출이 가능하다.

마지막으로 {(n / 2)} ~ {(n - 1)} 번지의 셀에 하향 천이가 발생했을 경우를 살펴보자. {(n / 2)} ~ {(n - 1)} 번지의 셀들의 경우 March 요소 1의 0번 ~ 3번 단계에서 Port 2를 통해 하향 천이가 발생되고, March 요소 3의 4번 ~ 7번 단계에서 Port 1을 통해 하향 천이가 발생된다. 이때 두 포트를 통해 접근되는 셀을 제외한 다른 셀들의 값을 살펴보면 각각의 셀이 March 요소 1에서와 March 요소 3에서 서로 반대되는 값을 갖고 있음을 알 수 있다. 따라서, {(n / 2)} ~ {(n - 1)} 번지에서의 하향 천이에 의한 결합 고장을 모두 발생시킬 수 있다. 이 고장들은 March 요소 1의 4번 ~ 7번 단계와 March 요소 3의 0번 ~ 3번 단계에서 검출이 가능하다.

2. 주소 디코더에서의 고장 검출

주소 디코더에서는 발생할 수 있는 고장을 본 논문에서 제안한 알고리듬은 다음과 같은 절차에 따라 모두 검출할 수 있다.

가) 하나의 주소에 의해 접근되는 셀이 하나 이상인 고장

하나의 주소에 의해 복수 개의 셀이 접근되는 고장은 메모리 셀 어레이에 존재하는 결합 고장과 동일하다. 예를 들어, i 번지의 셀에 x 값을 쓴다면 같이 접근되는 j 번지 셀에도 x 값이 쓰여진다($i \neq j$). 따라서, j 번지의 셀의 값을 읽어볼 때 이 고장을 검출할 수 있다.

나) 어떤 주소는 셀을 전혀 접근할 수 없는 고장

셀을 전혀 접근할 수 없는 고장은 메모리 셀 어레이에 존재하는 고착-개방 고장과 같다. 이 고장은 R(x), W(~x), R(~x)의 순서로 읽고 쓰기를 수행하면 검출이 가능하다[1,12]. 본 논문에서 제안하는 테스트 알고리듬에서는 각각의 셀에 R(x), W(~x), R(~x), W(x), W(~x), R(~x), W(x), R(~x), W(x)의 순서로 테스트를 진행하므로 이 고장을 검출할 수 있다.

3. 읽기·쓰기 회로에서의 고장 검출

읽기·쓰기 회로에서 발생할 수 있는 고장은 다음과 같은 과정을 통해 모두 검출된다.

가) 하나 이상의 데이터 라인 비트가 고착되는 고장

읽기·쓰기 회로에 고착 고장이 발생되면 고착된 데이터 라인 비트에 연결되는 모든 메모리 셀 에는 항상 고착된 값이 쓰여지고 읽혀지게 된다. 따라서, 각각의 메모리 셀에 0과 1을 쓰고 읽어볼 수 있으면 이 고장의 검출이 가능하다. 본 논문에서 제안한 알고리듬은 모든 March 요소에서 각각의 포트를 통하여 모든 셀에 대해 0과 1을 쓰고 읽어 볼 수 있으므로 이 고장을 검출할 수 있다.

나) 하나 이상의 데이터 라인 비트가 고착-개방되는 고장

읽기·쓰기 회로에 고착-개방 고장이 발생하면 고착-개방된 데이터 라인 비트에 연결되는 모든 메모리 셀에 고착-개방 고장이 발생되는 것과 같은 기능적 고장 증상을 보이므로 R(x), W(~x), R(~x) 의 순서로 읽고 쓰면 이 고장의 검출이 가능하다[1,12]. 본 논문에서 제안한 방법에서는 각각의 포트에서 모든 셀에 대하여 R(x), W(~x), R(x)의 순서로 테스트를 수행하므로 읽기·쓰기 회로에서 발생되는 stuck-open 고장을 검출할 수 있다.

다) 데이터 라인 비트 쌍이 상태 결합되는 고장

읽기·쓰기 회로에서 데이터 라인의 비트 쌍이 상태 결합되는 고장은 같은 WL을 공유하는 셀들 사이의 상태 결합 고장과 동일하다[12]. 3장에서 설명한 바와 같이 워드 단위의 메모리 테스트에서는 배경 데이터를 이용하여 워드 내에 존재하는 결합 고장을 검출한다[1]. 본 논문에서 제안하는 방법 역시 워드 단위의 테스트를 수행하므로 읽기·쓰기 회로의 데이터 비트 쌍이 상태 결합하는 고장을 검출할 수 있다.

2. 성능 평가

본 논문에서 제안한 알고리듬의 시간 복잡도는 표 2와 같다. 여기서 N은 메모리 테스트 알고리듬의 복잡도 계산에 사용되는 단위로서 메모리의 주소 크기를 나타낸다.

초기화 과정은 Port 1과 Port 2에서 동시에 메모리 셀 어레이의 절반씩만을 초기화하기 때문에 0.5 N의 시간이 소요된다. 각각의 March 요소에서는 각 주소에 대하여 총 4번의 읽기나 쓰기 작업을 각각 수행하지만 Port 1과 Port 2에서 동시에 처리되므로 각 요소에서는 2N이 소요된다. 따라서, 총 시간 복잡도는 9N이다.

표 2. 제안된 알고리듬의 시간 복잡도
Table 2. Time complexity of proposed algorithm.

	Time complexity
Initialization 1	0.5 N
March element 1	2 N
March element 2	2 N
Initialization 2	0.5 N
March element 3	2 N
March element 4	2 N
Total	**9 N**

본 논문에서 제안한 방법은 SRAM 테스트에 가장 많이 사용되는 9N March 테스트 알고리듬으로 워드 단위의 이중-포트 메모리를 테스트했을 때에 비해 2배의 성능향상을 얻을 수 있다. 예를 들어, 기존의 9N March 테스트 알고리듬을 이용하여 이중 포트 메모리를 각각의 포트에 적용하면 사용되는 배경 데이터의 수를 B라 했을 때 9N × 2 × B의 시간이 소요된다. 그러나, 본 논문에서 제안한 방법을 이용하면 절반의 시간인 9N × B 만이 소요된다.

테스트를 위하여 부가되는 BIST 회로는 면적 오버헤드가 적을수록 좋다. BIST 회로의 면적 오버헤드

문제에 있어서 또 하나 고려해야 할 사항은 메모리에 대한 BIST 회로의 상대적인 크기 문제이다. 일반적으로 테스트해야 할 메모리의 크기가 커짐에 따라 BIST 회로의 크기도 커지게 된다. 따라서 메모리의 크기에 따른 BIST 회로의 상대적인 크기를 최소화하는 것이 필요하다. 본 논문에서 제안한 방법을 구현한 BIST 회로의 면적 오버헤드는 NAND 게이트의 면적을 1로 봤을 때 표 3과 같다.

표 3에서 알 수 있듯이 2KByte(2K × 8bit) 크기의 내장 메모리에 대해 구현된 BIST 회로의 크기를 1로 보았을 때 4K 및 8K 바이트 메모리에 대해서 구현된 BIST 회로의 상대적인 크기는 각각 1.06과 1.10으로 상대적 크기의 증가가 매우 작음을 알 수 있었으며, 제안된 알고리듬이 실제 BIST 회로로 구현하기에도 매우 적합하다는 것을 알 수 있다.

표 3. 면적 오버헤드
Table 3. Area overhead.

	Area	Area ratio
BIST for 2K byte memory	1478.52	1
BIST for 4K byte memory	1560.99	1.06
BIST for 8K byte memory	1626.09	1.10

V. BIST 회로의 설계 및 경계 주사 기법과의 인터페이스

1. BIST 회로 설계

제안된 메모리 테스트 알고리듬을 이용하여 이중 포트 메모리를 대상으로 Verilog-HDL을 이용하여 BIST 회로를 다음과 같이 구현하였다. 구현된 BIST 회로는 다음의 기본기능을 수행한다.

- 메모리의 정상 동작과 테스트 동작사이의 전환 기능
- 테스트를 위한 주소 생성 및 테스트 데이터의 생성 기능
- 테스트 알고리듬 수행과 테스트 결과 분석 기능

이 기능들을 수행하는 BIST의 구조는 제어 회로(CONTROL: control logic), 주소 생성 회로(AGL: address generation logic), 데이터 생성 회로(DGL: data generation logic), 데이터 비교 회로(DCL: data comparison logic)로 구성된다. 그리고, 메모리 BIST는 경계 주사 기법을 이용하여 제어된다. 그림 10은 구현된 BIST의 블록 다이어그램을 보여주고 있다.

- 제어 회로(CONTROL): 제어 회로는 테스트 진행 과정 중에 자체 테스트 회로의 각 모듈의 동작을 제어

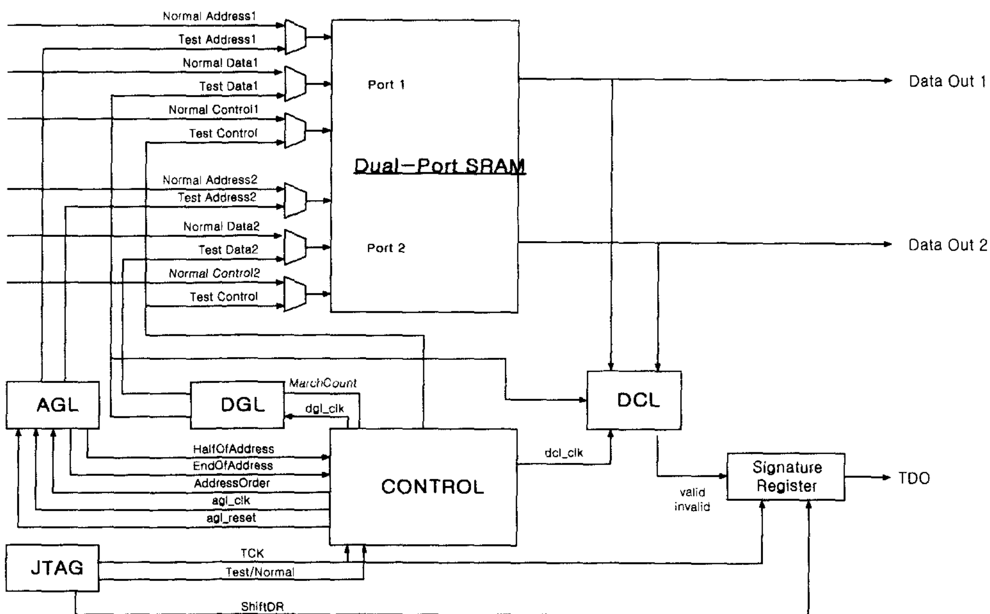

그림 10. 구현된 자체 테스트 회로의 블록 다이어그램
Fig. 10. Implemented self test logic block diagram.

하는 회로이다. 이 회로는 테스트의 시작과 종료를 판단하고, 테스트가 진행되는 동안 BIST 회로의 각 모듈로 적절한 제어 신호를 보낸다.

- 데이터 생성 회로(DGL): 데이터 생성 회로는 테스트 알고리듬의 진행 순서에 따라 테스트될 메모리에 배경 데이터를 공급해 주는 회로이다. 제어 회로로부터 제어 신호를 받아 각 테스트 단계별로 필요한 테스트 데이터를 생성한다. 본 논문에서 제안한 테스트 알고리듬에서는 이중 포트에서 동시에 테스트를 수행하고, 두 포트에서는 반대되는 배경 데이터를 이용해 테스트하기 때문에 Port 1에서 필요한 배경 데이터만을 생성하고, 이 값을 역전시켜(inverted) Port 2에서 사용하도록 설계하였다.
- 주소 생성 회로(AGL): 주소 생성 회로는 테스트될 워드를 지정하는데 사용되는 주소를 생성하는 회로이다. 본 논문에서 제안한 테스트 알고리듬은 주소가 증가하는 방향과 감소하는 방향으로 동시에 테스트를 수행하기 때문에 이진 증가·감소 카운터를 사용하여 구현하였다.
- 데이터 비교 회로(DCL): 데이터 비교 회로는 메모리에서 읽어들인 테스트 결과값과 예상되는 결과값을 비교하여 메모리의 고장 여부를 판단하는 회로이다. 비교된 결과는 압축치 레지스터(SR: signature register)에 따로 저장되며, 테스트 종료 후에 경계 주사 회로를 이용하여 외부에서 확인할 수 있다.

2. 경계 주사 기법과 메모리 BIST회로의 인터페이스

기판 수준의 테스팅 문제점을 해결하고, 기판 상의 여러 칩들의 테스팅을 용이하게 하기 위해 등장한 것이 1990년도에 IEEE에서 제정된 기판 수준 테스팅을 위한 IEEE 1149.1 표준안인 경계 주사(Boundary Scan) 기법이다[13, 14]. 경계 주사 기법은 기판 수준 테스팅뿐만 아니라 내장된 BIST와 같은 다양한 테스트 회로를 제어하는데 사용될 수도 있으며, 대부분의 상용 칩들은 경계 주사 기법을 이용하여 테스트 회로를 제어하고 있다[3, 13, 14].

그림 11은 본 연구에서 구현된 경계 주사 기법을 이용하여 내장된 메모리 BIST를 제어하는 것을 보여준다.

메모리 BIST를 구동시키기 위해서는 TDI 포트를 통해 미리 정의된 테스트 명령어를 경계 주사 회로의 명령어 레지스터에 인가한다. 인가된 명령어는 명령어 디코더에서 해석되고, 메모리 BIST에 Enable 신호가

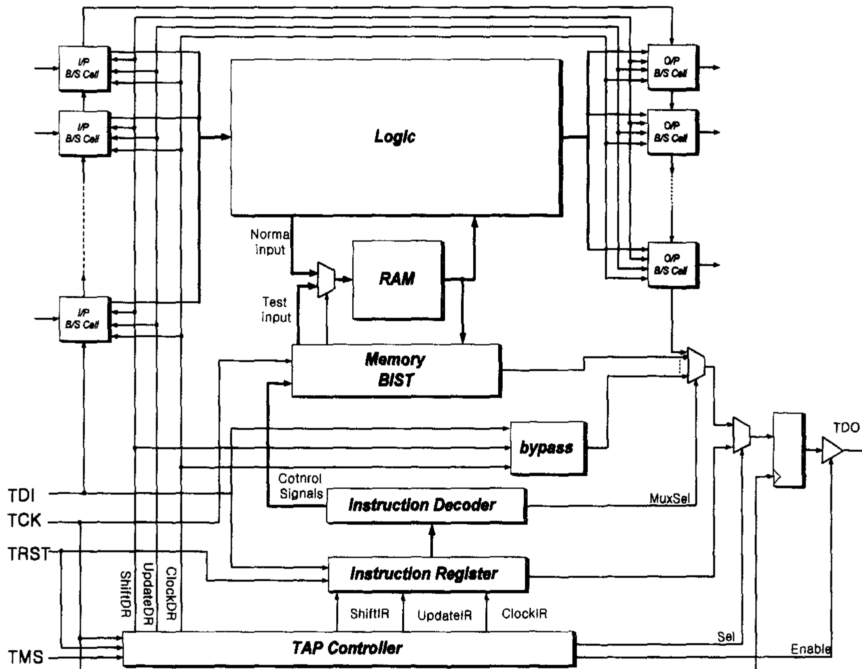

그림 11. 경계 주사 기법과 메모리 BIST 회로의 인터페이스
Fig. 11. Boundary scan and Memory BIST interface.

인가되고, 메모리의 입력에 테스트를 위한 데이터가 들어갈 수 있도록 메모리의 입력단에 위치한 멀티플렉서의 선택 신호를 조정해 준다. 또한, 테스트 결과를 TDO 포트로 출력시킬 수 있도록 메모리 BIST의 출력단에 위치한 멀티플렉서들의 선택 신호를 조정해 준다.

메모리 BIST는 Enable 신호가 인가된 직후에 제안된 알고리듬에 따라 테스트를 수행한다. 테스트의 결과는 자체 테스트 회로에 있는 레지스터에 저장되고, 테스트가 끝난 후 이미 설정된 경로를 통해 TDO 포트로 출력된다.

Ⅵ. 결론 및 향후 연구방향

최근에는 생산되는 마이크로 프로세서의 경우 메모리가 칩의 전체 트랜지스터의 대부분(70~90%)을 차지하고, 전체 다이 면적의 상당 부분(30~50%)을 차지하게 됨에 따라, 내장된 메모리에서 BIST의 중요성이 더욱 부각되고 있다[15]. 본 연구에서는 이중-포트 메모리에 대하여 기존의 멀티-포트 메모리 테스트 알고리듬들이 검출할 수 있는 모든 고장을 검출하면서 약 2배의 성능을 갖는 이중-포트 테스트 알고리듬을 제안하고, 이를 Verilog-HDL을 이용하여 구현하였다. 또한, IEEE 1149.1 표준안인 경계 주사 기법을 이용하여 메모리 자체 테스트 회로를 제어할 수 있도록 하였다. 본 연구의 결과는 효과적인 내장된 이중 포트 메모리의 테스트와 경계 주사 기법을 이용한 자체 테스트 회로의 제어를 가능하게 할 것이다.

향후 연구과제로는 자체 테스트 회로의 오버헤드 최적화, 멀티-포트 메모리 테스트를 위한 BIST 회로의 개선, 다수의 내장 메모리를 갖는 통신 및 멀티미디어 시스템에의 적용 등이며, 실제 시스템에의 적용을 통한 테스트 용이도 및 효용성 입증에 관한 연구를 수행할 예정이다.

참 고 문 헌

[1] A. J. Goor, *Testing Semiconductor Memories*, John Wiley & Sons Ltd., 1991.

[2] M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital systems testing and testable design*, Computer Science Press, 1990.

[3] *Memory BistCore™ User's Refernece Manual*, GeneSys TestWare, Revision 1.4, June, 1998.

[4] Yuejian Wu and Sanjay Gupta, "Built-In Self-Test for Multi-Port RAMs", *International Test Conference*, 1997.

[5] R. P. Treuer and V. K. Agarwal, "Fault Location Algorithms for Repairable Embedded RAMs," *International Test Conference*, 1993.

[6] Pinamki Mazumder and Kanad Chakraborty, *Testing and Testable Design of High-Density Random-Access Memories*, Kluwer Academic Publishers, 1996.

[7] Tom Chen and Glen Sunada, "A Self-Testing and Self-Repairing Structure for Ultra-Large Capacity Memories," *International Test Conference*, 1992.

[8] J. V. Sas, G. V. Wause, E. Huyskens and D. Rabaey, "BIST for Embeded Static RAMs with Coverage Calculation," *International Test Conference*, 1993.

[9] V. G. Mikitjuk, V. N. Yarmolik, A. J. van de Goor, "RAM Testing Algorithms for Detection Multiple Linked Faults," *International Test Conference*, 1996.

[10] R. Nair, S. M. Thatte and J. A. Abraham, "Efficient Algorithms for Testing Semiconductor Random-Access Memories," *IEEE Transactions on Computers*, Vol. C-28, No. 3, March 1979, pp. 258-261.

[11] Manoj Sachdev, "Test and Testability Techniques for Open Defects in RAM Address Decoder," *International Test Conference*, 1996.

[12] R. Dekker, F. Beenker, L. Thijseen, "Fault Modeling and Test Algorithm Development for Static Random Access Memories," *International Test Conference*, 1988.

[13] IEEE Std 1149.1-1990, *IEEE Standard Test Access Port and Boundary-Scan Architecture*, Published by the Institute

of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA.

[14] H. Bleeker, P. V. D. Eijnden, F. D. Jong, *BOUNDARY-SCAN TEST A Practical Approach*, Kluwer Academic Publishers, 1993.

[15] MICROPROCESSOR TEST SECTION, *Design & Test of Computers*, Vol. 15, No. 3, 1998, pp. 56-96.

저 자 소 개

韓 載 天(正會員)

1997년 숭실대학교 컴퓨터학부 졸업. 1999년 2월 동 대학원 전자계산학과(석사 과정) 졸업. 주관심분야는 컴퓨터 구조, VLSI 설계 및 V테스팅, 시스템 프로그래밍 등임

梁 善 雄(正會員)

1996년 숭실대학교 컴퓨터학부 졸업. 1998년 숭실대학교 대학원 전자계산학과 졸업(M.S). 1998년부터 숭실대학교 대학원 컴퓨터공학과 박사과정 재학중. 주관심분야는 컴퓨터 구조, VLSI 설계 및 테스팅, 디지털 신호처리 등

陣 明 究(正會員)

1998년 숭실대학교 컴퓨터학부 졸업. 1998 ~ 숭실대학교 대학원 컴퓨터공학과 석사과정 재학중. 주관심분야는 VLSI/CAD, 시스템 프로그래밍, 컴퓨터 구조, 디지털 신호처리 등

張 勳(正會員)

1987년 서울대학교 전자공학과 졸업(B.S.). 1989년 서울대학교 대학원 전자공학과 졸업(M.S.). 1993년 University of Texas at Austin 박사학위 취득. 1991년 IBM Inc. 1993년 Motorola Inc. Senior Member of Technical Staff. 1994 ~ 숭실대학교 컴퓨터학부 조교수. 주관심분야는 컴퓨터 시스템, VLSI 설계, VLSI 테스팅 등임