논문 2008-45SD-9-6

명령어 분석기를 이용한 고속 메모리 테스트를 위한 병렬 ALPG

(An Effective Parallel ALPG for High Speed Memory Testing Using Instruction Analyzer)

윤 현 준*, 양 명 훈*, 김 용 준*, 박 영 규*, 박 재 석*, 강 성 호**

(Hyunjun Yoon, Myung-Hoon Yang, Yongjoon Kim, Youngkyu Park, Jaeseok Park, and Sungho Kang)

요 약

메모리의 속도가 빠르게 향상됨에 따라, 고속 메모리를 테스트하기 위한 테스트 장비가 요구되고 있다. 특히 고속 메모리를 사용자가 원하는 명령어를 그대로 사용하여 효율적으로 테스트할 수 있도록 패턴을 만들어 내는 알고리즘 패턴 생성기(ALPG)가 필요하다. 본 논문에서는 고속 메모리 테스트를 위한 새로운 병렬 ALPG를 제안한다. 제안하는 ALPG는 명령어 분석기를 통해 사용자가 실행하고자 하는 명령어를 그대로 사용하여 고속 메모리 테스트를 위한 패턴을 생성할 수 있다.

Abstract

As the speed of memory is improved very fast, the advanced test equipments are needed to test the ultra-high speed memory devices efficiently. It is necessary to develop the Algorithmic Pattern Generator (ALPG) that tests fast memory devices effectively using the instructions that testers want to use. In this paper, we propose a new parallel ALPG for the ultra-high speed memory testing. The proposed ALPG can generate patterns for fast memory devices at high speed using manual instructions by the Instruction Analyzer.

Keywords : ALPG, Algorithmic Pattern Generator, Memory test, Instruction Analyzer

I. 서 론

최근 메모리 반도체 설계 기술의 급속한 발전에 따라, 메모리의 동작 속도가 빠르게 높아지고 있다. 그리고 반도체 공정 기술이 발전하여 메모리 회로의 집적도가 점점 증가하고 있다. 이러한 고속의 고집적 메모리는 메모리 반도체 산업의 비약적인 발전을 이끌어내고 있다. 그러나 고속의 고집적 메모리는 고속으로 동작함과 동시에 회로의 복잡도를 크게 증가시켜, 반도체 생산 비용 대비 테스트 비용 또한 크게 증가시키고 있다. 이에 따라 테스트 비용을 줄여 메모리 반도체 제품의 경쟁력을 높이기 위한 효율적인 테스트 기술은 매우 중요한 이슈가 되었다.

메모리 반도체를 테스트하는 방법 중, 대표적인 방법으로 Automatic Test Equipment (ATE)를 사용한 방법이 있다. ATE를 통한 메모리 반도체 테스트 방법은 Built-In Self Test (BIST)와 같은 방법과는 달리 메모리 반도체 내부에 테스트 회로를 삽입하지 않고 보다

* 학생회원, ** 정회원, 연세대학교 공과대학 전기전자공학과
(Department of Electrical and Electronic Engineering, Yonsei University)
※ 본 논문은 IDEC(IC Design Education Center)의 CAD tool 지원을 받은 것임.
접수일자: 2008년4월30일, 수정완료일: 2008년8월26일

다양한 기능을 테스트할 수 있는 장점이 있다. 하지만 ATE는 비용이 많이 들뿐만 아니라 최신의 고속 메모리의 속도에 대한 완벽한 검증을 하기 어렵다. 이러한 문제점을 해결하기 위해 기존의 ATE를 사용하되 ATE 내부의 ALPG의 성능을 향상시키기 위한 연구가 계속 되어 왔다. 그러므로 고속의 고집적 메모리 반도체를 테스트하기 위해서는, 다양하고 복잡한 고장을 잡아낼 수 있는 다양한 종류의 패턴을 고속으로 생성할 수 있는 ALPG가 필요하다.

본 논문은 고속 메모리를 테스트하는 동시에, 사용자가 원하는 다양한 알고리즘을 사용할 수 있는 ALPG를 소개한다. 이 ALPG는 기존에 소개된 것과는 달리, 미리 명령어 리스트를 만들어 합성하지 않고, 사용자가 원하는 다양한 명령어를 그대로 사용할 수 있는 하드웨어 구조를 가지고 있다. 본 논문의 구성은 다음과 같다. 우선 Ⅱ장에서는 기존 연구에서의 ALPG의 장단점을 살펴본다. Ⅲ장에서는 제안하는 ALPG의 구조에 대해 설명한다. Ⅳ장에서는 제안하는 ALPG를 실험하여 구현한 결과에 대해 살펴본 후, Ⅴ장에서 결론을 맺는다.

Ⅱ. 기존 연구

고속 메모리를 테스트하기 위한 ALPG에 대한 기존의 연구 중 가장 대표적인 연구로, 네 개의 Pattern Generator (PG)를 병렬로 연결하여 만든 연구를 꼽을 수 있다[1~2]. PG는 메모리 테스트를 위한 패턴을 생성하는 부분으로, 기본적으로 패턴의 순서를 정하는 sequence controller, 주소를 만드는 address generator, 주소에 쓸 데이터를 만드는 data generator, 메모리에 들어가는 신호를 만드는 control signal generator로 구성되어 있다. 이 기본적인 구조는 여러 연구에서 비슷한 형태로 사용되고 있다[1~4]. 이 중, 네 개의 PG를 병렬로 연결하고, 각 PG가 미리 합성된 명령어를 수행함으로써 하나의 PG 보다 네 배의 속도를 내는 ALPG가 현재 사용되어지는 ALPG의 기본 구조이다[2]. 그러나 네 배의 속도를 내기 위해서는 반드시 주어진 명령어를 합성한 후 각 PG에 입력해주어야만 한다. 즉, 첫 번째 PG는 하나의 명령어를 수행하고, 두 번째, 세 번째, 네 번째 PG는 각각 두 개, 세 개, 네 개의 명령어가 미리 합성된 명령어를 한 사이클 내에 수행해야만 한다. 예를 들어, X를 하나의 레지스터라고 하고 X의 값에 1을 더하여 X에 저장하는 명령어인 X < X + 1을 네 번 수행하는 과정이라고 가정한다. 이때, 첫 번째 PG에서는 X < X + 1의 명령어만 수행하면 되지만, 두 번째 PG에서는 X < X + 2, 세 번째 PG에서는 X < X + 3, 네 번째 PG에서는 X < X + 4의 명령어를 수행해야 한다.

이러한 방법으로 미리 명령어를 합성하여 만들 경우, 합성해야 하는 명령어의 개수가 셀 수 없을 정도로 많다는 문제점이 있다. 즉, 모든 경우의 수를 따져서 합성된 명령어를 미리 정해놓는다는 것은 한계가 있다. 많이 사용하는 알고리즘에서 주로 사용하는 명령어의 리스트를 만들어서 미리 명령어를 합성해 놓으면 어느 정도의 명령어는 적용할 수 있다. 하지만, 사용자가 직접 만든 명령어를 사용할 경우, 명령어 리스트 이외의 다른 명령어를 합성해야 되므로 이 방법은 사용할 수 없다는 문제점이 있다.

위의 문제점을 해결하기 위해 명령어를 미리 합성하지 않고 두 개의 PG로 ALPG를 구현한 연구가 제안되었다[5]. 이 연구에서는 각 명령어를 분석하여 두 개의 PG에 개개의 명령어를 입력하고, 하나의 PG로부터 나오는 결과를 다른 PG에 넣어주어 마치 하나의 PG가 파이프라인 구조를 형성하는 것과 같은 구조를 가지고 있다. 이와 같은 구조는 사용자가 직접 만드는 복잡한 명령어도 사용할 수 있다는 장점이 있지만, 두 개의 PG로 구현했기 때문에 앞에서 설명한 방법보다 속도가 훨씬 느린 단점이 있다.

Ⅲ. 본 론

제안하는 ALPG의 구조는 그림 1과 같다. 이 ALPG는 네 개의 PG를 병렬로 연결한 구조이다. 전체적인 구조는 Instruction Memory, Instruction Analyzer, 4개의 PG, Pattern Selector, Delay-Locked Loop (DLL)로 이루어져 있다. 그리고, Memory Unter Test (MUT)는 테스트되는 메모리이다.

1. 세부 Block 설명

그림 1에서 ALPG를 구성하는 각 block의 세부 설명은 다음과 같다. Instruction Memory에는 사용자가 메모리를 테스트하는 명령어가 바이너리 형태로 저장된다. Instruction Analyzer는 Instruction Memory로부터 바이너리 형태의 명령어를 받아 어떤 명령어인지 분석하고, 명령어에 따라 패턴의 생성 순서를 결정하는 부분이다. PG는 패턴을 생성하는 부분이고, Pattern Selector는 네 개의 PG에서 생성되는 패턴 중 하나를 선택하는 부분이다. DLL은 각 block에 필요한 clock을

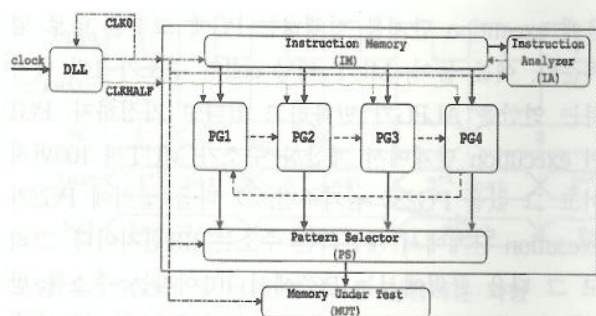

그림 1. 네 개의 PG를 사용한 병렬 ALPG의 구조
Fig. 1. Architecture of the parallel ALPG using four PGs.

만들어주는 부분으로 원래의 clock과 같은 주파수를 가지는 CLK0과, CLK0의 1/2의 주파수를 가지는 CLKHALF를 생성한다. CLK0은 Instruction Memory, Instruction Analyzer, Pattern Selector, Memory Under Test에 입력되고, CLKHALF는 네 개의 PG에 입력된다.

2. Instruction Analyzer에 의한 명령어 펼치기

Instruction Memory에 저장되어 있는 바이너리 형태의 명령어는 sequence, address, data, control 부분으로 구성되어 있다. Instruction Analyzer로 하나의 명령어가 입력되면, Instruction Analyzer는 명령어의 sequence 부분만을 파싱한다. Sequence 부분은 명령어의 종류와 패턴의 순서를 결정하는 데 필요한 정보를 가지고 있다. 명령어에는 No operation (NOP), Loop, Jump, End 등이 있으며, Instruction Analyzer는 이 명령어 중 Loop와 Jump 명령어만을 처리하게 된다. 즉, Loop 또는 Jump 명령어에 의해 반복되거나 branch되는 순서에 따라, 모든 명령어를 NOP로 만든다. 그림 2는 Instruction Analyzer가 Loop 명령어를 분석하여 처리하는 과정을 보여준다. Cell_num은 메모리 cell의 총 개수이며, ST0은 Loop 명령어가 반복을 시작할 부분을 의미한다. ST0부터 해당 Loop 명령어까지 Inst_A와 Inst_B 두 명령어를 10번 반복하면 되므로, 그림 2의 오른쪽과 같이 Inst_A와 Inst_B 한쌍을 10번 반복하는 NOP 명령어만의 리스트가 된다.

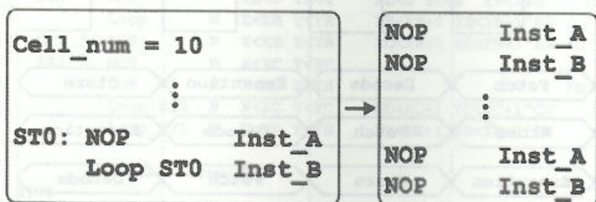

그림 2. Instruction Analyzer에 의한 명령어 펼치기
Fig. 2. Instruction Unrolling method by Instruction Analyzer.

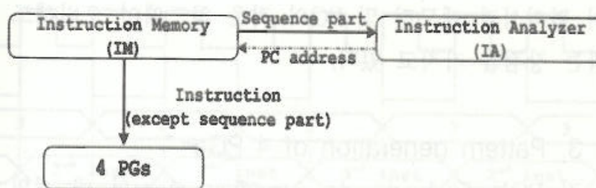

그림 3. PC address에 해당하는 명령어가 PG에 들어가는 과정
Fig. 3. Procedure of inputting the instruction of PC address into the PG.

이러한 명령어 펼치기 (Instruction Unrolling) 방법으로 Instruction Memory에 저장되어 있는 모든 명령어의 주소를 나타내는 PC address를 생성하여 Instruction Memory에 전달한다. Instruction Memory는 PC address에 따라 그 주소의 명령어를 Instruction Analyzer와 네 개의 PG에 전달한다. 그림 3과 같이, Instruction Analyzer에는 명령어의 sequence 부분만 전달하고, 네 개의 PG에는 sequence 부분을 제외한 다른 부분의 명령어를 전달한다.

Instruction Analyzer에 의한 명령어 펼치기 방법은 세 가지 장점을 가지고 있다. 첫 번째는 사용자가 명령어를 만들기 쉽다는 것이다. 기존의 ALPG의 경우 두 개 또는 네 개의 PG를 병렬로 연결하여 사용할 경우 하나의 PG를 사용할 때보다 많은 수의 명령어를 작성하여 Instruction Memory에 입력해주어야 한다. 그리고 단순히 명령어의 줄 수가 많아지는 것뿐만 아니라, 매우 복잡하게 명령어를 구성해야하는 어려움이 있어서 잘 알려진 알고리즘이 아닌 사용자가 원하는 새로운 알고리즘의 경우 명령어 자체를 만들어내기가 쉽지 않다. 제안하는 ALPG의 경우는 네 개의 PG를 사용함에도 불구하고, 하나의 PG를 사용할 때와 같은 수의 명령어를 Instruction Memory에 입력하여 사용자가 원하는 알고리즘을 쉽게 구현할 수 있다는 장점이 있다. 두 번째로 Instruction Memory의 크기를 줄일 수 있다는 것이다. 하나의 PG를 사용할 때와 같은 수의 명령어만 사용하면 되므로, 작은 크기의 Instruction Memory가 필요하다. 그러므로 메모리에 의한 하드웨어 오버헤드는 크지 않다. 세 번째로 Instruction Analyzer가 따로 존재하므로 PG 내부에 sequence controller가 필요 없다는 것이다. 그만큼 하나의 PG 당 하드웨어 오버헤드가 줄어들게 된다. 이와 같이 Instruction Analyzer에 의한 명령어 펼치기 방법은 사용자의 명령어 작성에 있어서

의 편의성과 메모리 및 PG의 적은 하드웨어 오버헤드라는 장점을 가지고 있다.

3. Pattern generation of 4 PGs

각 PG는 fetch, decode, execution, store의 네 개의 단계로 동작한다. Fetch 단계에서는 PC address에 해당하는 하나의 명령어에서 sequence 부분을 제외한 address, data, control 부분이 해당 PG로 입력된다. Decode 단계에서는 address, data, control 부분이 각각 세부적으로 파싱되어 PG 내부의 address generator, data generator, control signal generator로 입력된다. 예를 들어, 명령어 중 address 부분에서 메모리의 주소를 연산하기 위해 필요한 코드와 데이터가 파싱되어 address generator로 입력된다. Execution 단계에서는 address, data 및 control signal generator에서 해당 코드가 실행된다. 예를 들어, address generator에서는 입력받은 코드와 데이터를 통해, 데이터를 쓰거나 읽을 MUT의 한 cell의 주소를 만든다. 마지막 단계인 store 단계에서는 세 개의 generator에서 만든 데이터를 MUT로 전달하기 위해 레지스터에 저장한다. 주소를 예를 들면, address generator에서 만든 MUT의 주소를 그 다음 클럭에 사용할 수 있도록 store 단계에서 레지스터에 저장한다. 이와 같이 각 PG는 네 개의 단계로 동작하여 패턴을 생성한다.

네 개의 PG는 DLL에서 생성하는 CLKHALF의 한 클럭마다 한 개의 단계를 교대로 실행한다. 그림 4와 같이, 각 PG는 매 클럭마다 서로 다른 단계를 실행하여 겹치는 부분이 없도록 함으로써, 파이프라인과 비슷한 원리로 동작하게 구성하였다. Excution 단계를 예로 들면, PG1이 execution을 마치면, PG2가 execution 단계를 시작하고, 그 다음에는 PG3와 PG4 순으로 execution 단계를 실행한다. 이때, 다른 단계와는 달리 execution 단계에서는 현재 execution 단계를 실행하는 PG에서 연산을 끝내서 주소 및 데이터가 나오면, 그 다음에 execution 단계를 실행하는 PG에 그것을 바로 넘겨준다. 예를 들어, MUT 해당 cell의 주소가 1씩 증가하는 연산을 ALPG가 반복하고 있다고 가정하자. PG1이 execution 단계에서 계산한 주소가 MUT의 100번지이고 그 값을 PG2로 넘겨주면, 그 다음 클럭에 PG2가 execution 단계에서 계산하는 주소는 101번지이다. 그리고 그 다음 클럭에서는 PG3에서 101이라는 주소를 받아 execution 단계에서 102로 만들어 PG4로 전달한다. 이와 같이 네 개의 PG가 순차적으로 네 개의 단계를 각각 실행하여 패턴을 생성한다.

4. CLK0 & CLKHALF of Delay-Locked Loop

제안하는 ALPG에서 사용하는 클럭은 원래의 클럭과 같은 주파수를 가지는 CLK0과 원래 클럭의 1/2의 주파수를 가지는 CLKHALF 등 두 가지이다. 네 개의 PG에서만 CLKHALF를 사용하고 나머지 부분에서는 CLK0을 사용한다. 이와 같이 PG에만 CLKHALF를 사용하는 이유는 다음과 같이 두 가지가 있다. 첫 번째는, ALPG의 모든 블록에 같은 clock을 사용한다면, critical path를 가지는 부분은 ALPG의 다른 부분이 아닌 PG의 address generator에 존재한다. 즉, ALPG의 속도가 address generator의 execution 단계의 속도로 결정된다. 그러므로 address generator가 동작하는 시간을 두 배로 늘리면 address generator에서의 critical path가 없어질 것이다. 두 번째, Instruction Analyzer에서 PC address가 두 CLK0 마다 하나씩 출력된다. 첫 번째 CLK0의 rising edge에서는 Instruction Memory에서 명령어를 읽어 오고, 두 번째 CLK0의 rising edge에서는 명령어의 sequence 부분을 분석하여 PC address를 정한다. 즉, 두 CLK0 마다 PC address를 업데이트한다. 그림 5를 보면 CLK0보다 주기가 두 배 큰 CLKHALF의 falling edge마다 PC address가 바뀌게 된다. 그리고 PC address가 바뀐 후, 한 CLK0 후인 CLKHALF의 rising edge에 PC address 값을 받아 PG가 명령어를

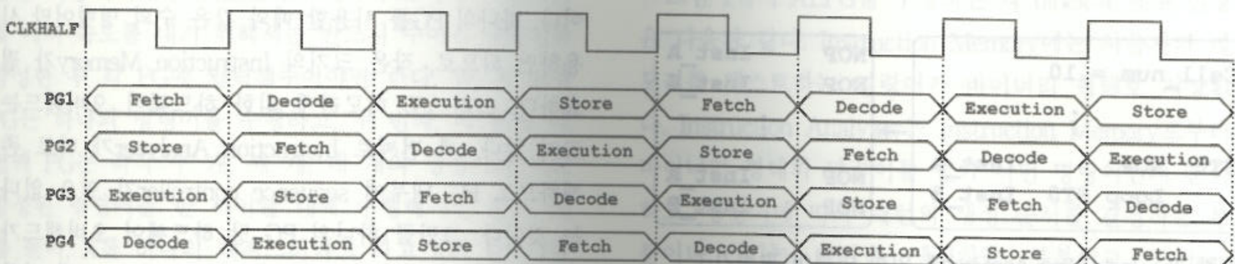

그림 4. 네 개의 PG가 동작하는 과정
Fig. 4. Operatng procedure of four PGs.

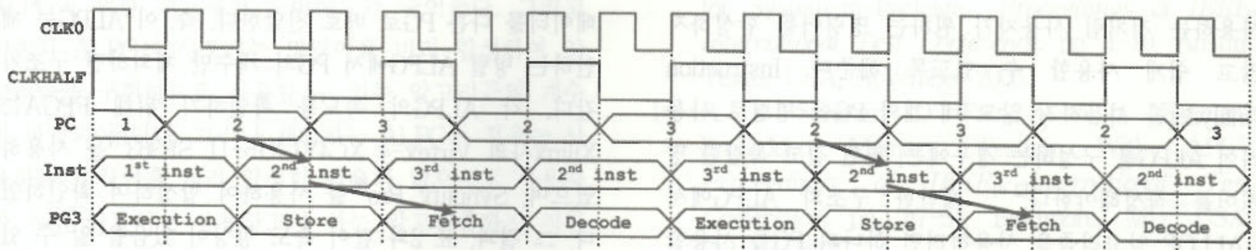

그림 5. PG3이 두 번째 명령어를 실행하는 과정
Fig. 5. Procedure of executing second instruction in PG3.

입력받으면 그 후에 PG의 네 단계가 시작되면서 패턴을 만들어낸다. 이와 같이 두 CLK0마다 PC address를 업데이트하는 Instruction Analyzer의 구조상, PG는 CLKHALF를 기본 clock으로 동작하도록 설계하였다.

IV. 실험 결과

제안하는 병렬 ALPG를 사용하여 GALLOP 알고리즘을 실행했을 때의 예를 살펴보자. 먼저, 그림 6에서 GALLOP 알고리즘을 구현하는 명령어를 나타내었다. 이 명령어는 [6]에서 사용하는 명령어와 비슷한 형식으로 표기되었다. 위의 명령어를 간단히 설명하면 다음과 같다. X와 Y는 각각 메모리의 열과 행의 주소이다. XB와 XC는 X의 주소 레지스터, YB와 YC는 Y의 주소 레지스터이며, TP는 데이터 레지스터이다. 그리고 TPH는 초기 값으로 0이 저장되어 있다. XB<XB+1은 XB값을 1만큼 증가시키는 명령어이고, YB<YB+1^BX는 XB값이 한 행의 마지막 열의 주소가 되었을 때 그 다음 행의 주소가 YB값, 첫 번째 열의 주소가 XB값이 되도록 하는 명령어이다. /D는 D의 반전된 값이며, Goto&Inverse_ST0은 ST0으로 돌아가 다시 시작하되, TP값을 반전시킨다. 이와 같이 명령어를 사용하면 GALLOP 알고리즘을 구현할 수 있다.

제안하는 병렬 ALPG에 그림 6의 명령어를 적용했을 때의 결과를 보면 다음과 같다. 실험 결과를 시뮬레이션했을 때, 각 명령어가 그림 7과 같이 펼쳐지는 것을 확인할 수 있었다. 그림에서 생략을 나타내는 점들은 각 명령어가 메모리의 cell의 개수만큼 반복되는 것을 의미한다. 예를 들어, 5번에서 9번까지의 명령어가 포함된 블록을 설명하면 다음과 같다. 여기에는 6번에서 8번까지의 명령어를 포함하는 작은 블록과 5번에서 9번까지의 명령어를 포함하는 큰 블록 등 두 개의 블록이 있다. 먼저 5번의 명령어를 실행한 후, 작은 블록의 명령어를 cell의 개수만큼 반복한다. 그 후에 9번의 명령어를 실행한 후, 다시 5번 명령어로 돌아가게 된다. 이와 같이 5번에서 9번까지의 큰 블록의 명령어를 cell의 개수만큼 반복한다.

이와 같이 Instruction Analyzer를 사용한 명령어 펼치기 방법은 네 개의 PG를 사용하더라도 하나의 PG를

```
START #0
ST0  : NOP                 XB<0 YB<0       XC<0 YC<0   TP<TPH
       Loop          W     X<XB Y<YB       XB<XB+1 YB<YB+1^BX
ST1  : NOP           W     X<XB Y<YB       XC<XB+1 YC<YB+1^BX /D
ST2  : NOP           R     X<XC Y<YC
       NOP           R     X<XB Y<YB                          /D
       Loop_ST2      R     X<XC Y<YC       XC<XC+1 YC<YC+1^CY
       Loop_ST1      W     X<XB Y<YB       XB<XB+1 YB<YB+1^BX

       GoTo&Inverse_ST0
STOP
END
```

그림 6. GALLOP 알고리즘 명령어
Fig. 6. Instructions for GALLOP algorithm.

```
 1  START #0
 2  NOP        XB<0 YB<0 XC<0 YC<0 TP<0
 3  NOP   W   X<XB Y<YB    XB<XB+1 YB<YB+1^BX
                 :
 4  NOP   W   X<XB Y<YB    XB<XB+1 YB<YB+1^BX

 5  NOP   W   X<XB Y<YB    XC<XB+1 YC<YB+1^BX /D
 6  NOP   R   X<XC Y<YC
 7  NOP   R   X<XB Y<YB                      /D
 8  NOP   R   X<XC Y<YC    XC<XC+1 YC<YC+1^CY
                 :
 9  NOP   W   X<XB Y<YB    XB<XB+1 YB<YB+1^BX

10  NOP        XB<0 YB<0 XC<0 YC<0 TP<1
11  NOP   W   X<XB Y<YB    XB<XB+1 YB<YB+1^BX
                 :
12  NOP   W   X<XB Y<YB    XB<XB+1 YB<YB+1^BX

13  NOP   W   X<XB Y<YB    XC<XB+1 YC<YB+1^BX /D
14  NOP   R   X<XC Y<YC
15  NOP   R   X<XB Y<YB                      /D
16  NOP   R   X<XC Y<YC    XC<XC+1 YC<YC+1^CY
                 :
17  NOP   W   X<XB Y<YB    XB<XB+1 YB<YB+1^BX
                 :
18  STOP
19  END
```

그림 7. Instruction Analyzer에 의해 만들어진 GALLOP 알고리즘 명령어 리스트
Fig. 7. GALLOP algorithm Instruction list made by Instruction Analyzer.

사용하는 것처럼 사용자가 원하는 명령어를 합성하지 않고 쉽게 사용할 수 있도록 해준다. Instruction Analyzer를 사용하지 않고 네 개의 PG를 병렬로 사용하여 ALPG를 구성하는 경우에는, 훨씬 길고 복잡한 명령어를 작성해야한다[2, 6]. 제안한 구조의 ALPG에서 GALLOP 알고리즘을 사용하려면 하나의 PG를 사용할 때와 같이, 그림 6의 명령어에 파라미터 정의를 합한 18줄의 명령어가 필요하지만, Instruction Analyzer를 사용하지 않은 병렬 ALPG[2, 6]에서는 120줄의 명령어가 필요하다. 그러므로 제안하는 병렬 ALPG의 경우가 더 간단하고 짧은 명령어를 사용하여 사용자가 원하는 알고리즘을 쉽게 구현할 수 있다. 표 1에서 제안하는 병렬 ALPG와 기존 ALPG에서 네 개의 알고리즘을 구현할 때 사용하는 명령어 줄 수를 비교해놓았다.

제안하는 ALPG에서 사용한 네 개의 PG 중, 한 개의 PG를 사용한 ALPG, 두 개의 PG를 사용한 ALPG, 네 개의 PG를 병렬로 연결한 제안하는 ALPG의 속도를 표 2와 같이 비교하였다. 세 경우 모두 같은 PG를 사용하여 비교함으로써, 제안하는 병렬 ALPG의 구조의 효과를 정확히 확인하도록 하였다. 이때, 두 개의 PG를 사용한 ALPG의 경우는 [5]의 논문에서 구현했던 ALPG와 구조가 다르다. [5]의 논문에서는 두 PG 사이에 Common Register를 사용하여 각 PG간 이동하는 데이터를 매 클록마다 번갈아가며 저장한 뒤 다른 PG에서 사용하였으나, 표 2의 두 개의 PG를 사용한 ALPG는 Common Register를 없애고 각 PG에서 만들어내는

표 1. 기존 ALPG와의 사용하는 명령어 줄 수 비교
Table 1. Comparisons of the number of the instruction line between the previous ALPG and proposed ALPG.

Algorithm	기존 ALPG[2]	제안하는 ALPG
March	31	16
Checker Board	44	16
Gallop	120	18
Moving Inversion	99	29

표 2. PG 수에 따른 ALPG의 속도 비교
Table 2. Comparisons of the speed of the ALPGs with different numbers of PG.

ALPG	ALPG (1 PG)	ALPG (2 PG)[5]	제안하는 ALPG (4 PG)
ALPG 속도(MHz)	103.8	171.6	321.4
ALPG 속도 비율	1	1.65	3.09

데이터를 다른 PG로 바로 전달한다. 즉, 이 ALPG는 제안하는 병렬 ALPG에서 PG의 개수만 제외하면 구조가 같다. 각 ALPG의 속도를 확인하기 위해 FPGA는 Xilinx사의 Virtex-4 XC4VLX15-11 SF363[7]을 사용하였으며, Synplify Pro[8]를 사용하여 합성하여 확인하였다. 그 결과, 표 2와 같이 속도 향상이 있음을 알 수 있었다. 이와 같이, 제안하는 ALPG의 구조는 PG를 하나 사용할 때보다 큰 성능 향상을 가져다준다.

지금까지의 내용을 정리하여 기존의 ALPG[2, 6]와 제안하는 ALPG를 표 3에서 비교하였다. 먼저 기존의 ALPG는 ASIC, 제안하는 ALPG는 FPGA에서 구현하였으므로, 속도를 직접적으로 비교할 수 없다. 그러나 제안하는 ALPG는 하나의 PG를 사용했을 때보다 속도 면에서 성능이 향상되는 것을 확인할 수 있다. 그러므로 하나의 PG를 사용할 때보다 훨씬 빠른 속도로 패턴을 생성할 수 있다. 그리고 기존의 ALPG와 제안하는 ALPG를 구성하는 PG를 비교하면, 기존의 ALPG의 각 PG는 sequence controller를 포함하고 있지만 제안하는 ALPG의 각 PG는 sequence controller를 포함하고 있지 않다. 대신에, 앞에서 설명한 것과 같이 제안하는 ALPG에서는 Instruction Analyzer가 sequence controller의 역할을 한다. 그러므로 제안하는 ALPG의 경우, 기존의 ALPG보다 각 PG에 의한 하드웨어 오버헤드가 작으므로 전체 ALPG의 하드웨어 오버헤드도 줄어들게 된다. 또한 기존의 ALPG와 제안하는 ALPG를 사용할 때의 명령어의 줄 수가 다르다. 많이 사용하는 몇 가지 알고리즘을 살펴본 결과, 제안하는 ALPG의 경우에 줄 수도 적고 간단한 명령어를 사용할 수 있음을 확인할 수 있었다. 명령어를 간단하게 구현할 경우, 사용자가 원하는 명령어를 쉽게 적용하여 사용할 수 있다는 장점이 있다. 또한 명령어 줄 수가 적으므로 명령어를 저장하는 Instruction Memory의 크기가 작으므로,

표 3. 기존 ALPG와 제안하는 ALPG 비교
Table 3. Comparisons between the previous ALPG and proposed ALPG.

ALPG	기존 ALPG[2]	제안하는 ALPG
칩 구현 종류	ASIC	FPGA
동작 속도	500MHz	321.4MHz
하드웨어 오버헤드	크다	작다
명령어 줄 수	많다	적다
명령어 복잡도	크다	작다
명령어 메모리	크다	작다
알고리즘 제한	크다	작다

ALPG에 의한 하드웨어 오버헤드가 줄어든다. 그리고 기존의 ALPG는 사용하는 명령어를 미리 합성하여 알고리즘을 구현하므로 사용할 수 있는 알고리즘의 개수가 한정되어 있다. 그러나 제안하는 ALPG의 경우는 사용하는 명령어를 미리 합성할 필요가 없으므로 사용자가 직접 만들어서 사용하고자 하는 알고리즘까지 지원한다.

V. 결 론

본 논문에서는 네 개의 PG를 병렬 연결하여 구성한 병렬 ALPG의 구조를 제안했다. Instruction Analyzer를 사용한 제안하는 병렬 ALPG는, 사용자가 원하는 명령어를 간단하게 작성하여 패턴을 생성할 수 있도록 하고, 명령어를 실행할 때 미리 명령어를 합성할 필요가 없다. 그리고 각 PG와 Instruction Memory에 의한 하드웨어 오버헤드가 줄어든다. 또한 네 개의 PG를 사용하여 한 개의 PG를 사용할 때보다 빠른 속도로 패턴을 생성할 수 있도록 하였다. 네 개의 PG를 병렬로 연결하되 Instruction Analyzer를 사용하지 않은 기존의 ALPG와 비교했을 때, 한 개의 PG를 사용했을 때보다 패턴 생성 속도를 향상시킬 수 있다는 면에서는 같지만, 사용자가 원하는 알고리즘을 위한 명령어를 보다 쉽고 간단하게 사용할 수 있다는 강점이 있다. 이러한 명령어 사용의 편의성은, 테스트 알고리즘을 쉽게 만들어 사용하여 점점 다양해지고 복잡해지는 메모리 고장을 찾아낼 수 있게 함으로써, 메모리 테스트의 효율을 향상시킬 것으로 기대된다.

참 고 문 헌

[1] S. Kikuchi, Y. Hayashi, T. Matsumoto, R. Yoshino, R. Takagi, "A 250 MHz shared-resource VLSI test system with high pin count and memory test capability", *Proceedings of IEEE International Test Conference*, pp. 558-566, Washington, D.C., USA, August 1989.

[2] H. Imada, K. Fujisaki, T. Ohsawa, M. Tsuto, "Generation technique of 500 MHz ultra-high speed algorithmic pattern", *Proceedings of IEEE International Test Conference*, pp. 677-684, Washington, D.C., USA, October 1996.

[3] K. Yamasaki, I. Suzuki, A. Kobayashi, K. Horie, Y. Kobayashi, H. Aoki, H. Hayashi, K. Tada, K. Tsutsumida, K. Higeta, "External Memory BIST for System-in-Package", *Proceedings of IEEE International Test Conference*, pp. 1-10, Austin, Texas, USA, November 2005.

[4] A. T. Sivaram, D. Fan, A. Yiin, "Efficient Embedded Memory Testing With APG", *Proceedings of IEEE International Test Conference*, pp. 47-54, Baltimore, MD, USA, October 2005.

[5] 윤현준, 양명훈, 김용준, 박영규, 이대열, 강성호, "고속 메모리 테스트를 위한 파이프라인 ALPG", *테스트 학술대회*, 서울, 대한민국, 2007년 6월

[6] ATL-51 Pattern Program Reference Manual. Release 2004. 04, Advantest Corporation, Tokyo, Japan.

[7] Virtex-4 User Guide. Release 2007. 04, Xilinx Inc., San Jose, CA.

[8] Synplify Userguide. Release 2004. 02, Synplicity Inc., Sunnyvale, CA.

저 자 소 개

윤 현 준(학생회원)
2004년 2월 연세대학교 공과대학 전기전자공학과 학사졸업.
2008년 현재 연세대학교 공과대학 전기전자공학과 석사과정.
<주관심분야 : SoC 설계, SoC 테스트>

김 용 준(학생회원)
2002년 2월 연세대학교 공과대학 전기공학과 학사졸업.
2004년 2월 연세대학교 공과대학 전기전자공학과 석사졸업.
2008년 현재 연세대학교 공과대학 전기전자공학과 박사과정.
<주관심분야 : SoC 설계, SoC 테스트>

박 재 석(학생회원)
2008년 2월 연세대학교 전자공학과 학사 졸업.
2008년 현재 연세대학교 전기전자공학과 석박통합과정.
<주관심분야 : Compression, SoC 테스트>

양 명 훈(학생회원)
1996년 2월 연세대학교 공과대학 전기공학과 학사 졸업.
1998년 연세대학교 공과대학 전기전자공학과 석사졸업.
2004년 삼성전자 시스템 LSI 선임 연구원.
2008년 현재 연세대학교 공과대학 전기전자공학과 박사과정.
<주관심분야 : SoC 설계, SoC 테스트>

박 영 규(학생회원)
2004년 2월 호서대학교 공과대학 전자공학과 학사졸업.
2007년 2월 연세대학교 공과대학 전기전자공학과 석사졸업.
2008년 현재 연세대학교 공과대학 전기전자공학과 박사과정.
<주관심분야 : SoC 설계, 테스트>

강 성 호(정회원)
1986년 2월 서울대학교 공대 제어계측공학과 학사졸업.
1988년 5월 The University of Texas at Austin 전기 및 컴퓨터 공학과 석사졸업.
1992년 5월 The University of Texas at Austin 전기 및 컴퓨터공학과 박사 졸업.
1992년 미국 Schlumberger, Research Scientist.
1994년 미국 Motorola, Senior Staff Engineer.
2008년 현재 연세대학교 전기전자공학과 교수.
<주관심분야 : SoC 설계, SoC 테스트>