순차 회로를 위한 효율적인 혼합 고장 진단 알고리듬

An Efficient Hybrid Diagnosis Algorithm for Sequential Circuits

김지혜, 이주환, 강성호

JiHye Kim, JooHwan Lee and Sungho Kang

사단법인 대한전자공학회

논문 2004-41SD-5-7

순차 회로를 위한 효율적인 혼합 고장 진단 알고리듬

(An Efficient Hybrid Diagnosis Algorithm for Sequential Circuits)

김 지 혜*, 이 주 환**, 강 성 호***

(JiHye Kim, JooHwan Lee, and SungHo Kang)

요 약

반도체 기술의 발달로 회로의 집적도와 복잡도가 증가함에 따라 칩의 생산 과정에서 고장이 발생하는 빈도가 높아지게 되었다. 칩의 수율을 향상시키고, 생산 단가를 절감시키기 위해서 고장의 원인을 찾아내고 분석하는 과정은 매우 중요하다. 그러나 고장의 원인을 분석하는 과정 중 고장의 위치를 찾아내는 데는 많은 시간이 소요된다. 게이트 수준에서의 고장 위치 진단은 물리적 수준에서의 고장 범위를 한정해 줌으로써 고장 위치를 찾는 데 소요되는 시간을 줄 일 수 있다는 데 의미를 갖는다. 본 논문에서는 새로운 방식의 고장 딕셔너리 방식과 추가적인 고장 시뮬레이션 방식을 혼합하여, 메모리의 소비를 최소화하면서도 시뮬레이션 수행 시간을 단축시킴으로써 효과적으로 고장 진단을 수행할 수 있는 고장 진단 알고리듬을 제안한다.

Abstract

Due to the improvements in circuit design and manufacturing technique, the complexity of a circuit is growing. Since the complexity of a circuit causes high frequency of faults, it is very important to locate faults for improvement of yield and reduction of production cost. But unfortunately it takes a long time to find sites of defects by e-beam proving in the physical level. A fault diagnosis algorithm in the gate level has meaning to reduce diagnosis time by limiting fault sites. In this paper, we propose an efficient fault diagnosis algorithm in the logical level. Our method is hybrid fault diagnosis algorithm using a new fault dictionary and additional fault simulation which minimizes memory consumption and simulation time.

Keywords : 순차회로, 고장 진단, 고장 딕셔너리, 고장 시뮬레이션

I. 서 론

반도체 설계와 공정 기술의 발달로 회로의 복잡도가 증가하고 하나의 칩이 수천만 게이트로 이루어질 정도로 고집적화 되었다. 따라서 칩의 생산 과정에서 발생하게 되는 고장이 늘어나게 되었다. 이렇듯 고장의 발생이 빈번해짐에 따라 칩의 수율을 향상시키기 위해서는 고장에 대한 분석을 통해 설계나 공정상의 고장 원인을 찾아내어 이를 개선하는 과정을 수행해야 한다. 이를 위해 고장 진단 과정이 필요하다. 회로가 고집적화 됨에 따라 고장의 위치를 찾아내는 과정은 더욱 복잡해지고 많은 시간을 필요로 하게 되었다. 따라서 효과적으로 고장의 위치를 찾아내기 위한 고장 진단 알고리듬에 대한 연구가 중요하게 여겨지고 있다.[1,2]

회로의 고장 위치를 진단하기 위한 방식은 크게 cause-effect 방식과 effect-cause 방식으로 나눌 수 있다. cause-effect 방식은 대상 회로에 고장을 삽입하여 시뮬레이션을 수행한 후 그 결과를 바탕으로 고장의 위치를 찾아가는 방식이다. 고장 시뮬레이션이나 고장 딕셔너리 방식 등이 이에 해당된다. 고장 시뮬레이션 방식은 각각의 칩을 진단 할 때마다 모든 고장에 대한 시뮬레이션을 수행해 주어야 하므로 진단 시간이 오래 걸리는 단점을 가지고 있다. 고장 딕셔너리 방식은 고장 시뮬레이션을 고장 진단 과정 이전에 수행한 후 그 결과를 저장하여 고장 진단 시 사용하는 방식으로 고장 진단 시간이 짧은 반면 많은 메모리의 소비로 인해 큰

* 정회원, 삼성전자 반도체 총괄 시스템 LSI 사업부
(Samsung Electronics, System LSI Division)
** 학생회원, ***정회원, 연세대학교 전기전자공학부
(Dept. of Electrical and Electronic Engineering, Yonsei Univ.)

※ 본 연구는 정보통신부 대학 IT연구센터 육성 지원 사업의 연구결과로 수행되었습니다.
접수일자: 2003년12월16일, 수정완료일: 2004년5월3일

회로에 대해서는 사용하기 힘들다. effect-cause 방식은 회로 내에 존재하는 고장의 영향을 받아 나타난 결과물인 고장 출력을 시작으로 하여 고장의 후보가 될 수 있는 경로를 추적해 나감으로써 고장의 위치를 찾아가는 방식이다. back-tracing이나 critical-path-tracing 방식 등이 이에 해당한다. 이 방식은 고장이 발견된 출력을 시작으로 고장이 발생 가능한 경로를 입력 단 까지 추적해 나가는 방식으로 여러 가지 고장 모델에 대한 고려가 용이하고 메모리 소비가 적은 반면, 알고리듬이 복잡하고 이를 수행하는 데 많은 시간이 걸리는 단점을 가지고 있다.

조합 회로의 경우 입력으로 가해지는 테스트 패턴의 개수가 한정적이고 플립플롭을 가지고 있다고 하더라도 완전 스캔 구조로 이루어지기 때문에 회로 내부에 필요한 데이터를 만들어주고 이를 출력 단에서 확인하는 과정이 순차 회로에 비해 용이하다. 그러므로 대부분의 경우 회로의 특성에 적합한 고장 딕셔너리를 이용하여 고장 진단을 하는 방식이 주로 제안되어 왔다[1-8].

그러나 순차회로의 경우, 플립플롭의 영향으로 회로 내부에 필요한 데이터를 넣어주는 것이 어려우며, 이를 출력 단으로 전파시키는 과정도 조합 회로보다 복잡하다. 그러므로 다양한 고장 진단 방식의 적용이 어려워 기존의 방식 중 대부분의 경우 effect-cause 방식을 적용시키고 있다[9, 10]. 그러나 critical-path-tracing이나 back-tracing 방식은 알고리듬이 복잡하고 수행시간이 많이 소요되며 고장 후보의 개수가 많아지는 등의 단점을 가지고 있다. 또한 순차회로의 경우 테스트 패턴의 수가 많으며 unknown value(x)의 표기로 인해 하나의 데이터를 2비트로 표기해야 한다. 따라서 딕셔너리의 크기가 너무 커져 기존의 방식대로는 큰 순차 회로의 경우 딕셔너리의 생성이 거의 불가능 하게 되며, 모든 고장에 대해 시뮬레이션을 수행하는 것도 많은 시간을 요하므로 cause-effect 방식을 사용할 수 없게 된다. 따라서 본 논문에서는 순차 회로를 위한 새로운 방식의 혼합 고장 진단 알고리듬을 제안한다. 제안하는 고장 진단 알고리듬은 정적(static) 고장 진단 방식과 동적(dynamic) 고장 진단 방식을 혼합한 방식이다. 이 방식은 새로운 고장 딕셔너리를 제안하여 아주 작은 크기의 딕셔너리로 고장 후보의 개수를 1차적으로 줄인다. 그 후, 걸러진 고장 후보에 대해 고장 시뮬레이션을 통하여 세부적으로 구분해 내는 방식이다. 이 방식은 메모리의 소비를 최소화 하면서도 단시간에 고장 진단 성능을 향상시켜 큰 회로를 가진 스캔이 없는 칩에서도 적용 가능한 단일 고착 고장을 위한 효과적인 고장 진단 방식이다. 본 논문은 다음과 같은 구성으로 이루어 졌다. Ⅱ장에서는 고장 진단을 위한 기존연구를 소개하고 Ⅲ장에서는 제안하는 고장 딕셔너리와 이를 이용한 고장 진단 알고리듬에 대해 소개한다. Ⅳ장에서는 실험 결과를 제시하고 Ⅴ장에서 결론을 맺는다.

Ⅱ. 기존 연구

조합 회로와 순차 회로에 대한 고장 진단은 근본적으로는 cause-effect 방식과 effect-cause 방식을 사용할 수 있다는 것에 대해서는 비슷하다고 할 수 있다. 조합 회로의 경우, 회로 내부에 원하는 데이터를 만들어주고 그 데이터를 출력 단에서 확인하기가 순차회로에 비해 쉽기 때문에 칩의 테스트가 쉬울 뿐 아니라 테스트 결과의 분석과 진단이 용이하다. 그러나 순차회로의 경우, 모든 플립플롭에 원하는 데이터를 넣어주는 것이 쉽지 않다. 플립플롭은 현재의 입력 값이 아닌 이전의 입력 값을 저장하고 있으므로 이전 입력 값의 결과가 현재의 출력 값으로 나타나게 된다. 따라서 현재 입력에 대한 결과뿐만 아니라 이전에 가해졌던 입력 값에 대한 정보도 고려해야 하기 때문에 테스트 과정이나 테스트 결과의 분석을 통한 고장 진단 과정이 복잡해진다. 또한 플립플롭에 워하는 데이터를 넣어 주기 위해 더 많은 테스트 패턴을 필요로 하게 되는데 이것 또한 고장 진단을 어렵게 하는 요인이 된다. 플립플롭의 단이 여러 단으로 이루어졌을 경우에는 여러 타임 프레임(time frame) 이전의 입력 데이터까지도 모두 알 수 있어야 한다. 또한 회로 내부에 원하는 값을 넣어주고 이를 출력 단으로 전파하여 확인하고자 할 때에는 여러 패턴을 순차적으로 가하여 플립플롭을 원하는 값으로 먼저 바꾸어 놓은 후에 회로에 값을 인가하여 주어야 한다. 따라서 순차 회로의 테스트와 고장 진단 시에는 여러 타임 프레임에 대하여 고려해야 한다.

2.1 조합 회로에 대한 고장 진단

조합 회로에서는 주로 고장 딕셔너리를 이용한 고장 진단 방식이 제안되었다[1-8]. 고장 진단 과정은 고장 테스트 이후에 수행되지만 고장 딕셔너리는 고장 테스트 이전에 생성된다. 대표적인 고장 딕셔너리의 종류로는 완전 딕셔너리와 Output-compaction(OC) 딕셔너리, 그리고 pass/fail 딕셔너리가 있다. 완전 딕셔너리는 고장 시뮬레이션 결과인 고장 출력 값을 모두 저장하고 있는

형태의 딕셔너리로 고장 진단 능력이 가장 좋은 딕셔너리이지만 많은 저장 공간을 필요로 하므로 큰 회로에서는 사용하기 힘들다. pass/fail 딕셔너리는 각각의 고장에 대해 고장 시뮬레이션을 수행한 후 각각의 패턴에 대해 고장의 검출 여부만을 0과 1로 나타내어 저장한 방식으로 기존의 딕셔너리 중 가장 작은 크기를 가지고 있다. OC 딕셔너리는 크기는 작지만 단독으로 쓰일 경우 고장 진단 성능이 떨어지므로 pass/fail 딕셔너리와 함께 쓰이게 되면 완전 딕셔너리와 비슷한 고장 진단 성능을 갖게 된다.[8]

2.2 순차회로에 대한 고장 진단

순차 회로의 경우 테스트 패턴의 수가 많고 unkno-wn value의 저장 등의 문제로 고장 딕셔너리 보다는 critical-path-tracing 방식을 주로 사용한다. [9, 10]에서는 4단계로 이루어진 회로 추출 방식을 제안하였다. 이들은 고장의 위치를 추적하는 방식이므로 고장 모델이 고려되지 않아 여러 고장에 대해서 고장 진단이 가능하며 많은 메모리를 소비하지 않는다는 장점을 가지고 있다. 그러나 복잡한 수행 과정으로 인해 고장 진단 과정에서 소요되는 시간이 길다는 단점을 가지고 있다. 또한 딕셔너리와 같이 한번의 시뮬레이션으로 모든 칩에 적용 가능한 것이 아니라 각각의 칩마다 위의 고장 진단 과정을 모두 수행해주어야 하기 때문에 크고 많은 양의 칩을 진단하는 데는 적합하지 않다.

[11]에서는 고장 응답 값들의 가짓수와 각각의 종류마다 모든 패턴에 대해 나타나는 횟수를 저장하여 고장을 진단하는 순차 회로를 위한 고장 딕셔너리가 제안되었다. 이 방식은 정보들을 각각의 고장마다 저장을 하게 되면 고장 진단 성능은 완전 딕셔너리와 거의 비슷하며 크기는 평균 절반정도로 줄어들게 된다. 또한 이 방식은 순차 회로에 대해 적용시킨 고장 딕셔너리이며 고장 딕셔너리를 저장할 때 unknown value가 모두 없어질 때까지 테스트 패턴을 가한 후에 나타나는 고장 응답만을 저장하게 되므로 1비트로 값의 표현이 가능하게 하였다. 그러나 이 방식 역시 회로의 크기가 커지게 되면 너무 많은 메모리를 필요로 하게 되므로 사용 불가능 하게 된다.

[12]에서는 cause-effect와 effect-cause 방식을 혼합하여 순차 회로를 위한 고장 진단 알고리듬을 제안하였다. 이 방식은 effect-cause 방식을 사용하여 모든 고장에 대한 시뮬레이션의 25%에서 50%정도로 줄였으며 모든 과정을 동적 방식으로 수행하여 메모리의 소비가 거의 없는 반면 고장 딕셔너리를 사용했을 때 보다 수행 시간이 길어지는 것이 단점이다.

Ⅲ. 제안하는 고장 딕셔너리와 고장 진단 알고리듬

고장 딕셔너리는 테스트 이전에 생성되므로 테스트 결과에 상관없이 같은 회로를 갖는 칩에 대해서는 동일하게 사용 가능하다. 따라서 모든 고장과 모든 패턴에 대해 시뮬레이션하고 그 결과를 저장하여, 많은 메모리를 필요로 하지만 하나의 회로에 대해서 한 번만 생성하면 되고 고장 진단 과정에서는 단순한 데이터의 비교 과정만이 필요하므로 고장 진단 시간을 줄일 수 있다는 장점을 가지고 있다.

3.1 고장 딕셔너리

고장 딕셔너리는 어떠한 정보를 저장하느냐와 어떤 방식으로 저장하느냐에 따라 성능과 크기가 달라진다. 본 논문에서는 저장하고자 하는 정보를 변화시키며 효과적인 딕셔너리를 찾고자 하였으므로 딕셔너리의 저장 방식은 고려하지 않도록 하겠다. 순차 회로는 플립플롭이 어떤 데이터를 가지고 있느냐에 따라 가해질 테스트 패턴에 대한 출력 응답이 달라진다. 따라서 순차 회로의 테스트 패턴은 플립플롭의 데이터를 원하는 데이터로 만들어 주기 위한 테스트 패턴들을 포함하여 여러 테스트 패턴들이 하나의 테스트 패턴 셑(set)을 형성하게 된다. 따라서 순차회로를 테스트하기 위한 테스트 패턴의 수는 비슷한 크기의 조합회로에 비해 매우 많다. 조합회로의 경우 가해지는 모든 패턴에 대해 예상 응답을 저장하는 완전 딕셔너리(full dictionary)를 생성하여 별도의 고장 시뮬레이션이 없이 정적인 방법으로 고장 진단을 하여 수행 시간을 단축하기도 한다. 정적인 방식의 고장 진단이란 테스트 결과에 따라 고장 진단 과정에 필요한 정보를 모아야 하는 동적인 방식과 달리, 테스트 결과에 상관없이 모든 칩에 대해 동일하게 사용할 수 있는 방식을 말한다. 그러나 조합회로의 경우도 100만 게이트급 이상의 큰 회로에 대해서는 완전 딕셔너리를 생성하기 위해서는 너무 많은 메모리를 필요로 하기 때문에 생성이 어렵다. 이에 비해 테스트 패턴의 수가 더 많은 순차 회로의 경우는 이런 메모리의 소비가 더 큰 문제점이 된다. 게다가 플립플롭의 초기화 값인 unknown value가 출력 단으로 전파 될 경우, 이에 대한 저장을 위해 모든 값을 1비트가 아닌 2비트로 저장해야 하기 때문에 전체적인 딕셔너리의 크기

가 두 배가 된다. 따라서 어느 정도 작은 회로에 대해서는 완전 딕셔너리를 생성할 수 있지만, 회로의 크기가 커지게 되면 완전 딕셔너리 뿐만 아니라, 패턴의 수가 많기 때문에 기존의 딕셔너리 중 가장 작은 사이즈로 여겨지는 pass-fail 딕셔너리 조차 생성하지 못하게 된다. 그렇기 때문에 순차 회로를 위한 딕셔너리에서 모든 패턴에 대해 테스트 결과를 저장한다는 것은 아주 작은 회로를 제외하고는 거의 불가능 한 일이 된다. 따라서 사이즈를 최소화 하면서 고장 딕셔너리를 생성하기 위해서는 적은 수의 비트수로 가장 효율적으로 고장을 진단할 수 있도록 하는 요소를 찾아 이를 저장해야 한다.

순차회로에서의 테스트 패턴의 대부분을 차지하는 결정 패턴에 의해 검출되는 고장들은 또 다시 같은 조건의 패턴이 가해져야만 다시 검출되게 된다. 그러므로 같은 패턴에서 한 번 검출 된 고장들은 또다시 검출될 때 또 같은 패턴에서 함께 검출되는 경우가 많게 된다. 그러므로 고장이 검출될 때마다 고장 패턴을 저장하면 더 좋은 성능을 가지는 딕셔너리가 되겠지만 그림 1과 같이 각 고장에 대해서 가해질 테스트 패턴의 리스트 중 가장 처음으로 그 고장을 검출하게 되는 테스트 패턴만을 저장하여도 매우 작은 딕셔너리 사이즈로 10% 이하로 고장의 후보를 줄일 수 있는 효과를 가져 올 수 있다. 테스트 초기에 무작위 패턴에 의해 검출 된 고장의 경우에도 그 개수는 다소 많아지더라도 처음 고장을 검출 한 패턴의 저장만으로 상당한 고장 구별의 효과를 얻을 수 있다.

이와 같은 형태의 딕셔너리는 기존의 딕셔너리 중에 가장 크기가 작은 것으로 여겨지는 pass/fail 딕셔너리와 비교하여 고장 구분 능력은 다소 떨어지지만 딕셔너리의 크기는 상당히 작아지게 된다. 그러므로 처음 고장을 검출한 패턴의 번호를 저장한 딕셔너리는 100만 게이트 이상의 매우 큰 회로에 대해서도 사용 가능하다는 장점을 가지게 된다. 100만 게이트 급 회로의 경우, 회로의 크기가 커지면서 고장의 개수와 이를 검출하기 위한 테스트 패턴, 그리고 출력 핀의 수까지 모두 늘어나게 되기 때문에 가장 작은 크기의 pass/fail 딕셔너리 조차도 생성하기 어렵게 된다. 그러나 제안하는 딕셔너리를 생성할 경우, 예를 들어 고장의 개수가 3백만 개 정도이며 이를 검출하기 위한 테스트 패턴의 수가 4백만 개 가량이 된다고 하더라도, 같은 경우 pass/fail 딕셔너리는 120기가비트의 메모리를 필요로 하는 반면, 제안된 딕셔너리의 경우, 메모리 소비량이 66메가비트 즉 8메가바이트 정도면 가능해진다. 이와 같은 딕셔너리는 회로가 크고, 테스트 패턴의 수가 많은 경우에 사용하기 적합하다. 1000만 게이트의 회로라고 하여도 고장이 3천만개정도 되고, 패턴이 4천만개정도 된다면 딕셔너리의 사이즈는 780메가비트, 약 100메가바이트 정도일 것으로 예상된다. 순차회로의 경우, 고장 검출율이 낮기 때문에 실제 검출할 수 있는 고장의 수는 예상치보다 적을 것이고 그렇게 되면, 딕셔너리의 크기는 더욱 작아지게 되므로 1000만 게이트급 이상의 회로에서도 제안하는 딕셔너리의 사용이 가능하다.

그림 2는 s349.bench회로의 고장 시뮬레이션 결과 중의 일부를 보여주는 그림이다. 그림 2는 테스트 패턴을 순차적으로 가하여 각 패턴별로 전체 고장 리스트 중 처음으로 검출한 고장을 기록한 것이다. 그림 2를 제안하는 딕셔너리 방법으로 나타내면 표 1과 같이 된다. 여기에서 저장하게 될 테스트 패턴이라 함은 테스트 패턴 자체를 말하는 것이 아니라, 해당 테스트 패턴이 가해진 순서를 저장해 두는 것을 말한다. 이는 동일한 데이터의 테스트 패턴이 가해지는 경우가 많이 때문이다.

그림 1. 새로운 고장 딕셔너리 방식의 기본 아이디어
Fig. 1. Main idea of a new fault dictionary.

그림 2. s349 벤치회로의 테스트 결과의 일부 예시
Fig. 2. An example of the test result of s349.

표 1. s349에 대한 제안하는 딕셔너리 방식의 예
Table 1. An example of the fault dictionary of s349.

고장 이름	테스트 패턴 번호						
AX3-> AMVG5VG1VAD1NF /1	0	1	1	0	0	0	->24
BMVS0N-> BMVG4VG1VAD1NF /1	0	1	1	0	1	0	->30
BMVS0N-> BMVG3VG1VAD1NF /1	0	1	1	0	1	1	->31
BMVG5VG1VAD2NF /0	0	1	1	0	1	1	->31
SMVG2VS0P /1	0	1	1	0	1	1	->31
AMVS0N-> AMVG2VG1VAD1NF /1	1	0	0	0	0	0	->32

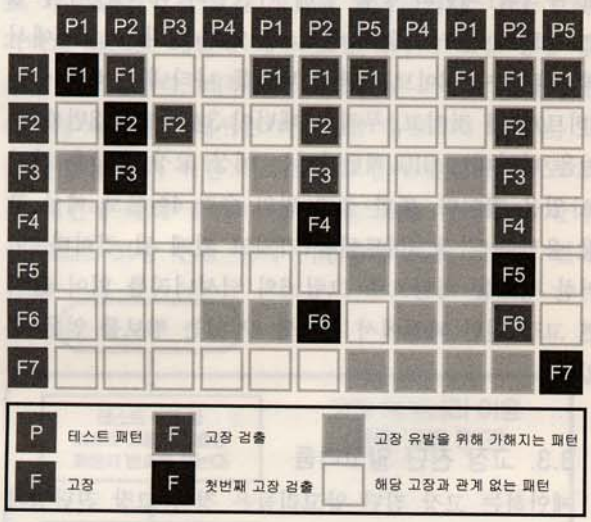

그림 3. 순차 회로의 고장 검출 과정
Fig. 3. Fault detection process of sequential circuits.

이렇게 고장을 처음으로 검출한 테스트 패턴을 저장 할 경우 고장 딕셔너리를 생성하는 데는 $F*\log_2(P)$ 비트가 필요하게 된다. (F: 고장 개수, P: 테스트 패턴의 개수)

그림 3은 11개의 테스트 패턴이 가해져 7개의 고장을 검출하는 테스트 결과를 보여주는 예시이다. 테스트 패턴은 P1, P2, P3, P4, P1, P2, P5, P4, P1, P2, P5의 순으로 가해진다. 고장 F1은 테스트 패턴 P1, P2, P5가 가해지면 항상 검출되는 고장이고, F2는 P2와 P3이 가해지면 항상 검출되는 고장이다. F3은 P1과 P2가 순차적으로 가해질 때마다 검출되는 고장이며 F4와 F6는 P4, P1 그리고 P2가 순서대로 가해질 때 검출되는 고장이다. F5는 P5, P4, P1, P2가 순차적으로 가해질 때, F7은 P5, P4, P1, P2, P5가 순차적으로 가해질 때 각각 검출할 수 있는 고장이다. 첫 번째 테스트 패턴에 대해 F1이 검출될 것이고, 두 번째 패턴 P2에 의해 F1, F2 F3이 검출될 것이며, 이 중 F2와 F3은 처음으로 검출 된 고장이다. 6번째 테스트 패턴인 P2에서 F4, F6이 처음으로 검출 될 것이며, 10번째 패턴에서 F5가, 마지막 패턴에서 F7이 검출되게 된다. F5가 검출 될 때에는 반드시 F4와 F6도 검출될 것이며, F4와 F6이 검출될 때에는 반드시 F3이 검출될 것이고, F3이 검출될 때에는 반드시 F2도 검출되게 된다. 이들은 이와 같이 어떠한 조건에서는 함께 검출이 되고, 또 어떠한 상황에서는 함께 검출되지 않을 수도 있다. 패턴들이 어떠한 순서로 가해지느냐에 따라 고장이 검출될 수도 아닐 수도 있게 된다. F4와 F6같은 경우에는 항상 같이 고장이 검출되므로 출력 응답 정보가 없으면 서로 구별되지 않게 되며, 각각 처음으로 검출 된 패턴의 번호를 저장한 딕셔너리의 경우, {F1}, {F2, F3}, {F4, F6}. {F5}, {F7}의 고장 후보 그룹으로 분류된다.

3.2 최초 검출 패턴에 대한 고장 출력 값의 저장

테스트 초기에 검출 된 많은 고장들은 처음 검출된 패턴의 번호를 통해 어느 정도 구분이 되긴 하지만 회로에 따라 이렇게 구분 된 고장 그룹의 크기가 커지게 되어 고장 진단 성능이 좋지 못한 경우가 발생할 수 있다. 따라서 테스트 초반에 검출 된 고장들을 더 세부적으로 구분하는데 고장을 구분할 수 있는 정보를 추가적으로 저장하게 되면 좀 더 좋은 효과를 얻을 수 있다. 그러므로 무작위 패턴에 의해 검출되는 고장에 대해서만 처음으로 고장을 검출한 테스트 패턴의 번호와 함께 그때의 예상 고장 응답 값을 저장하게 되면, 추가되는 메모리 소비량은 출력 핀 수와 무작위 패턴에 의해 검출 된 고장 개수의 곱이며 이로 인해 고장 후보의 그룹이 다시 세분화 되므로 평균적으로 구분되는 고장 후보의 수가 매우 줄어들게 된다. 이러한 예상 고장 응답 값의 저장은 출력 핀의 수가 적은 경우에 사용하는 것이 좋다. 출력 핀의 수가 많을 경우, 진단 성능 향상에 비해 추가되는 메모리의 소비가 너무 많이 늘어나게 될 수 있으므로, 작은 회로에 대해 이와 같은 방식을 사용하면 적은 메모리의 추가로 성능을 증대시킬 수 있다.

그림 4에서 처음 고장을 검출한 테스트 패턴만을 고장했을 경우, 고장 1과 고장 2가 서로 구분되지 않고, 고장 3과 고장 4가 서로 구분되지 않지만, 처음 고장을 검출한 테스트 패턴에 의한 고장 응답 값을 저장하게 되면, 구분 되지 않던 고장들이 세부적으로 구분되어 고장 진단 성능을 높이게 된다. 모든 고장에 대해 이러

고장	처음 검출한 패턴 번호					예상 고장 응답		
고장 1	0	0	1	1		0	0	1
고장 2	0	0	1	1	+	0	1	0
고장 3	1	0	1	1		1	1	0
고장 4	1	0	1	1		0	1	1

그림 4. 예상 고장 응답 값의 추가
Fig. 4. Addition of expected faulty responses.

고장	패턴 번호			예상 고장 응답				딕셔너리 정보
고장 1	0	0	1	0	1	1	0	입력 핀수 : 4
고장 2	0	0	1	1	0	1	1	출력 핀수 : 4
고장 3	0	1	0	0	1	0	1	고장 수 : 10
고장 4	1	0	0	Fault list				테스트 패턴수 : 8
고장 5	1	0	0	1 : A /0				무작위 패턴수 : 3
고장 6	1	0	0	2 : A /1				Pattern list
고장 7	1	1	0	3 : B /0				0 : 0010
고장 8	1	1	1	4 : B /1				1 : 0011
고장 9	1	1	1	5 : C /0				2 : 1010
고장 10	1	1	1	6 : C /1				3 : 1101
				7 : A->C /0				4 : 0111
				8 : A->C /1				5 : 1101
				9 : D /0				6 : 0010
				10 : D /1				7 : 0100

그림 5. 고장 딕셔너리와 딕셔너리 관련 정보
Fig. 5. Fault dictionary and its information.

한 예상 고장 응답 값을 저장하게 되면, 고장 진단 성능이 높아지겠지만, 그렇게 되면, 그만큼 메모리의 소비가 늘어나기 때문에, 가장 효과를 증대시킬 수 있는 부분인 무작위 패턴에 의해 검출되는 고장들에 대해서만 이와 같은 예상 고장 응답 값을 저장한 고장 딕셔너리를 생성하면, 무작위 패턴에 의해 검출되는 고장들은 그 개수가 많은 만큼 처음 고장을 검출한 테스트 패턴의 번호에 의해 한번 구분이 되고, 다시 예상 고장 응답을 비교하여 또 한번 구분이 되므로 처음 고장을 검출한 테스트 패턴의 번호만을 가지고 있을 때보다 고장 구별력이 더 좋아진다. 무작위 패턴에 의해 검출되지 않은 나머지 고장들은 여러 패턴들에 의해 분산되어 검출되므로 처음 고장을 검출한 테스트 패턴의 번호만으로 고장 후보를 구분한다.

생성된 고장 딕셔너리를 실제로 이용하려면 고장 딕셔너리에 대한 정보를 가지고 있어야 한다. 총 패턴수와 무작위 패턴의 수, 그리고 출력 핀의 수를 알고 있어야 딕셔너리를 사용할 수 있다. 그림 5는 고장 딕셔너리와 딕셔너리에 관한 정보, 그리고 고장 리스트와 테스트 패턴 리스트의 예를 보여주는 그림이다.

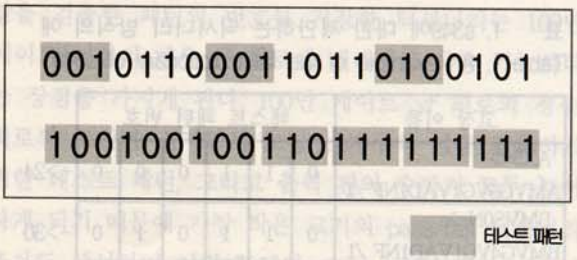

그림 6. 고장 딕셔너리의 실제 데이터 저장 방식
Fig. 6. Actual data storing method of a fault dictionary.

실제의 딕셔너리는 그림 6과 같이 0과 1의 이진수로만 표현된 형태의 값을 갖게 된다. 따라서 이용하고자 하는 딕셔너리의 테스트 패턴의 수와 출력 핀의 수, 그리고 무작위 패턴의 수를 통해 이진수의 나열을 의미 있는 비트로 나누어 읽어 들일 수 있게 된다. 그림 5에서의 패턴 수는 8이므로 패턴 번호를 나타내는 비트 수는 3비트가 될 것이고, 무작위 패턴이 3개이므로 3번째 패턴을 나타내는 010 패턴까지는 예상 고장 응답을 가지고 있을 것이다. 또한 출력 핀의 수가 4이므로 예상 고장 응답은 각각 4비트씩을 가지고 있게 될 것이다. 이러한 정보를 바탕으로 그림 6의 딕셔너리를 읽어 들이면 고장 진단 과정에서 사용할 수 있는 정보를 얻을 수 있다.

3.3. 고장 진단 알고리듬

제안하는 고장 진단 알고리듬은 정적 고장 진단법과 동적 고장 진단법을 혼합하여 사용하는 방식이다. 다시 말해 정적 고장 진단법을 사용했을 때 고장 진단 소요 시간을 줄일 수 있다는 점과 동적 고장 진단을 사용했을 때 메모리 소비를 줄일 수 있다는 점을 이용하여 메모리의 소비를 최소화 하면서 고장 진단 시간을 줄일 수 있는 방식이다. 정적 고장 진단법으로는 제안된 고장 딕셔너리를 통해 1차적으로 고장 후보를 구분해 내면, 구분된 고장 후보들 중에 최종 고장 후보를 가려내기 위한 동적 고장 진단법인 고장 시뮬레이션이 수행된다. 처음부터 고장 시뮬레이션을 수행하는 것 보다 고장 딕셔너리를 통해 걸러진 고장들에 대해서만 고장 시뮬레이션을 수행하므로 수행 시간이 매우 단축된다.

고장 진단과정은 다음과 같이 진행된다. 먼저 회로 네트 리스트와 테스트 패턴, 그리고 테스트 패턴을 통해 검출할 수 있는 고장 리스트를 입력으로 받아들인다. 고장 드롭핑 방식을 사용하여 고장 시뮬레이션을 수행하면서 가장 처음으로 고장을 검출하는 테스트 패턴의 번호를 기록한다. 동시에 새로운 고장을 검출하지

못하는 테스트 패턴이 나타나기 이전의 테스트 벡터에 대해서는 처음 고장을 검출하는 테스트 패턴에 대한 예상 고장 응답을 저장한다. 이 때 예상 고장 응답은 응답 자체를 그대로 저장해도 되지만, 고장 응답 리스트를 따로 저장하여 그 번호를 저장하면 좀 더 메모리를 적게 소비할 수도 있다. 무작위 패턴에 의해 검출되는 고장에 대한 딕셔너리 기록이 끝나면 결정 패턴에 의해 검출되는 고장에 대해 처음으로 고장을 검출하는 테스트 패턴의 번호를 기록한다. 모든 고장에 대해 위의 과정을 수행하게 되면 고장 딕셔너리가 생성된다. 고장 딕셔너리의 생성이 완료 된 후에는 각각의 칩에 대한 고장 진단 과정을 수행하게 된다. 먼저 고장 칩의 테스트 과정에서 얻게 되는 정보를 받아들인다. 다시 말해, 각각의 테스트 패턴에 대한 고장 유무나 고장 응답 값 등을 받아들인다. 대부분의 테스트는 고장이 검출되면, 이후의 패턴에 대해서는 테스트를 수행하지 않지만, 고장 진단을 하기 위한 테스트는 고장이 발견되더라도 모든 패턴에 대해 끝까지 고장 테스트를 수행해야 한다. 그러므로 고장 진단 시에 이용하게 되는 테스트 결과

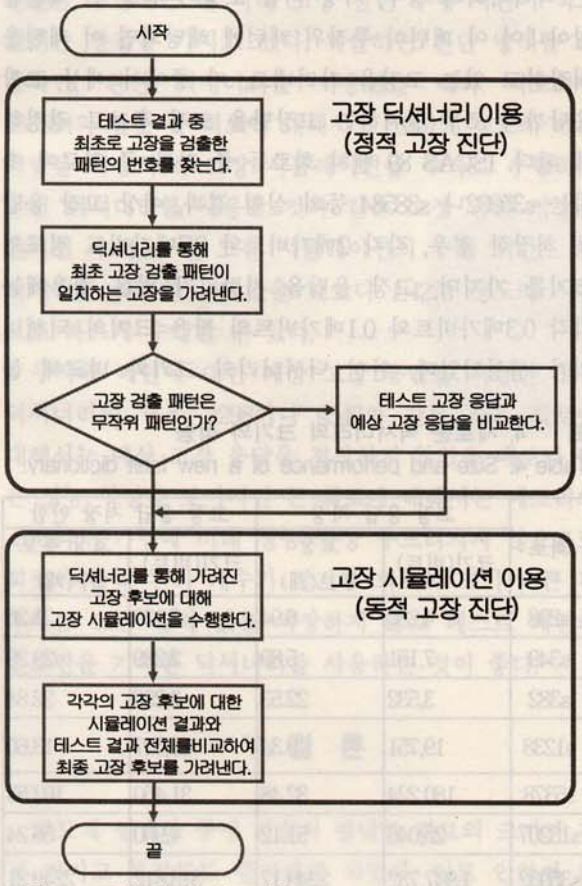

그림 7. 고장 진단 알고리듬
Fig. 7. A fault diagnosis algorithm.

정보는 테스트 대상 회로의 모든 테스트 패턴에 의한 고장 응답 값이다. 가장 먼저 테스트 결과 가장 처음으로 고장이 발견 된 테스트 패턴의 번호를 찾아낸다. 그리고 찾아낸 테스트 패턴의 번호가 무작위 패턴에 속하는 번호인지 확인하고, 무작위 패턴에 속하는 번호라면 해당 패턴에 대한 테스트 응답 값을 저장한다. 결정 패턴에 속하는 테스트 패턴이라면 1차적으로 고장 딕셔너리를 사용하기 위해 필요한 정보는 테스트 패턴의 번호만이 필요로 하게 된다. 딕셔너리를 사용하여 고장 후보를 줄이는 과정은 다음과 같다. 먼저 처음으로 고장을 검출한 테스트 패턴의 번호를 가지고, 고장 딕셔너리 첫 부분부터 비교를 시작한다. 이후에 출력 핀의 수만큼 고장 예상 응답 값이 저장되어 있으므로 고장 딕셔너리는 한번의 생성으로 같은 회로 네트 리스트를 갖는 모든 칩에 사용 가능하다. 그러나 이후의 고장 시뮬레이션은 모든 칩에 대해 각각 수행해 주어야 한다. 고장 진단 과정의 흐름도는 그림 7과 같다. 정적 고장 진단법인 고장 딕셔너리를 이용하여 처음으로 고장 후보를 걸러낸 후 동적 고장 진단법인 고장 시뮬레이션을 수행하며, 테스트 결과와 시뮬레이션 결과를 비교하여 최종 고장 후보를 결정하게 된다.

Ⅳ. 실험 결과

제안된 딕셔너리와 고장 진단에 대한 실험은 Hope 고장 시뮬레이터[13]를 기반으로 수행되었다. Hope는 병렬 고장 단일 패턴 고장 시뮬레이터(parallel fault single pattern simulator)로 하나의 테스트 패턴에 대해 여러 고장들을 병렬적으로 시뮬레이션 할 수 있는 고장 시뮬레이터이다. 테스트와 고장 진단을 실험하기 위한 테스트 패턴은 HITEC 테스트 패턴 생성기를 사용하였으며 대상 회로는 ISCAS 89회로에 대해 실험을 수행하였다.

표 2. ISCAS 89 벤치 회로에 대한 테스트 정보
Table 2. A test information of ISCAS 89 bench circuits.

회로	테스트 패턴 수(개)	고장 리스트 수(개)	고장 검출율 (%)
s298	220	265	86.039
s349	102	327	93.429
s382	1359	296	74.185
s1238	472	1283	94.686
s5378	894	3146	68.347
s13207	135	620	6.317
s35932	300	34868	89.190
s38584	2998	8445	23.263

표 2는 시뮬레이션 수행 시 가해지는 테스트 패턴의 수와 이 테스트 패턴으로 검출되는 고장 리스트의 개수, 그리고 이에 대한 고장 검출율을 나타낸 것이며 좀 더 정확한 실험을 위해 실험 대상은 고장 검출율이 높은 회로를 대상으로 하였다.

표 3은 각각의 회로에 대한 완전 딕셔너리의 크기와 pass/fail 딕셔너리의 크기 및 성능을 나타내는 것이다. 딕셔너리를 사용하지 않고 시뮬레이션을 수행할 경우, 각각의 회로가 가지는 고장의 개수만큼 고장 시뮬레이션을 수행해야 하므로 딕셔너리가 없을 경우의 고장 후보의 수는 고장 리스트의 수와 동일하다. pass/fail 딕셔너리를 사용했을 경우, 고장 후보의 수는 대부분 1% 정도로 줄어드는 것을 확인할 수 있다. 순차 회로에서 pass/fail 딕셔너리의 효과가 좋은 이유는 테스트 패턴의 수가 많기 때문에 이에 대한 정보를 모두 저장하였으므로 고장 진단에 사용할 수 있는 정보가 많아지기 때문이다. 그러나 딕셔너리의 성능이 좋아지는 대신 딕셔너리의 크기가 매우 커지게 된다. s35932 회로의 경우 게이트의 수와 플립플롭을 합한 수가 2만 개 안팎임에도 각각 10메가비트, 25메가비트 정도의 크기를 가지고 있다. s35932회로의 경우 회로의 s382나 s1238, s5378 등 크기가 훨씬 작은 회로의 테스트 패턴보다 적은 수인 300개의 매우 적은 테스트 패턴으로 테스트가 수행되었으며 s38584는 검출 가능한 고장 리스트의 수가 적었던 점을 생각할 때 보통의 경우, 비슷한 크기의 pass/fail 딕셔너리의 크기는 더 커질 것이며 회로의 크기가 10만 게이트 이상이 되면 기가비트 단위의 크기를 가지게 될 것으로 예상된다. 따라서 100만 게이트 급 이상의 큰 회로에 대해서는 사용이 힘들게 된다.

표 3. 완전 딕셔너리와 pass/fail 딕셔너리의 성능
Table 3. Performance of full dictionaries and pass/fail dictionaries.

회로	고장 수(개)	완전 딕셔너리 크기(비트)	pass/fail 딕셔너리	
			크기 (비트)	고장 후보 수(개)
s298	265	607,380	50,615	3.49
s349	327	733,788	33,354	3.34
s382	296	4,827,168	402,264	13.88
s1238	1283	16,956,128	605,576	1.28
s5378	3146	275,627,352	2,812,524	3.05
s13207	620	10,127,700	83,700	18.96
s35932	34868	6,694,656,000	10,460,400	431.98
s38584	8445	7,038,434,580	25,318,110	9.90

표 4는 제안하는 두 가지 방식의 딕셔너리의 크기와 딕셔너리의 성능을 나타내는 표이다. 이는 무작위 패턴에 대한 예상 고장 응답을 저장하는 경우와 저장하지 않는 경우로 나누어 실험한 결과이며 각각의 딕셔너리의 크기와 딕셔너리를 사용하였을 때의 고장 후보의 평균 개수를 나타내었다. 이때, 고장 후보의 평균 개수는 각 회로별로 100개의 고장을 임의로 뽑아 이에 대하여 고장 딕셔너리를 사용할 때의 결과를 나타낸 것이다. 실제 테스트를 통해 실험을 수행하기 위해서는 회로에 임의의 고장을 삽입하여 테스트 패턴을 가하고 나타나는 결과를 이용해야 하지만, 실험에서는 고장 리스트 중 100개의 고장을 임의로 선택하고 각각에 대하여 고장 시뮬레이션을 수행한 결과인 모든 패턴에 대한 고장 응답을 저장하여 테스트 결과 대신 사용하였다. 딕셔너리를 이용한 1차 고장 진단 과정에서는 먼저 테스트 응답과 고장이 없을 때의 시뮬레이션 결과를 비교하여 가장 처음으로 고장이 검출 된 패턴을 알아내고 이 패턴을 저장하고 있는 고장을 딕셔너리를 통해 가려내어 1차적인 고장 후보를 생성하게 된다. 고장 응답이 저장되어 있는 딕셔너리의 경우, 가장 처음 고장이 검출 된 패턴을 알아내어 이 패턴이 무작위 패턴에 해당되면 이 패턴을 저장하고 있는 고장을 가려내고, 이 중에서 예상 고장 응답까지 모두 일치하는 고장만을 고장 후보로 결정하게 된다. ISCAS 89 벤치 회로들 중 가장 큰 회로에 속하는 s35932나 s38584 등의 실험 결과, 예상 고장 응답을 저장한 경우, 각각 2메가비트와 0.7메가비트 정도의 크기를 가지며, 고장 응답을 저장하지 않은 경우에는 각각 0.3메가비트와 0.1메가비트의 작은 크기의 딕셔너리가 생성되었다. 완전 딕셔너리의 크기와 비교해 볼

표 4. 새로운 딕셔너리의 크기와 성능
Table 4. Size and performance of a new fault dictionary.

회로	고장 응답 저장		고장 응답 저장 안함	
	크기(비트)	고장 후보(개)	크기(비트)	고장 후보 수(개)
s298	4,280	6.94	2,120	21.26
s349	7,151	5.89	2,289	29.29
s382	3,532	22.53	3,256	22.84
s1238	19,751	9.34	11,547	13.60
s5378	180,224	32.48	31,460	102.53
s13207	25,046	51.12	4,960	56.24
s35932	1,947,732	2249.17	313,812	2249.21
s38584	697,372	94.64	101,340	120.09

때, 고장 응답을 저장하지 않은 경우 완전 딕셔너리에 대해 각각 0.005%와 0.001%의 매우 작은 딕셔너리를 통해 전체 고장의 6%와 1%대의 개수로 고장 후보를 줄일 수 있는 성능을 가지는 딕셔너리를 생성할 수 있었다. 따라서 기존의 딕셔너리가 큰 회로에서 사용할 수 없었기 때문에 고장 시뮬레이션 시간을 줄일 수가 없었던 반면, 제안하는 고장 딕셔너리는 매우 큰 회로에서도 적용 가능하여 1000만 게이트급 이상의 회로에서도 사용 가능하다. 고장 구분 능력이라 할 수 있는 고장 후보 수의 경우도 딕셔너리를 사용하지 않을 경우와 비교하여 1%에서 9% 정도의 수준으로 줄일 수 있게 된다. 또한 제안된 딕셔너리는 딕셔너리 사용 후에 고장 후보에 대한 고장 시뮬레이션을 전제로 하고 있기 때문에 고장 시뮬레이션을 수행하여 2차적으로 고장 진단을 수행하면, 모든 패턴과 모든 고장에 대해 시뮬레이션을 수행해준 결과, 즉 완전 딕셔너리를 사용했을 때의 성능과 같은 결과를 얻을 수 있게 된다. 또한 각각의 고장 후보와 패턴에 대해 한번씩의 고장 시뮬레이션만 수행하면 되므로 effect-cause 방식을 이용하여 고장 진단을 수행하는 데 비해 고장 진단 수행 시간이 적게 걸리는 장점을 가지고 있다. 제안하는 진단 방식을 이용하면, 완전 딕셔너리를 사용했을 때와 비교하여 s38584와 같은 큰 회로에 대해서도 0.001% 정도의 메모리만을 사용하고, 고장 시뮬레이션을 추가로 수행하여 완전 딕셔너리를 사용할 것과 같은 고장 진단 성능을 얻어낼 수 있으며, 고장 시뮬레이션의 수행 시간도 딕셔너리를 사용하지 않았을 때보다 1.42% 정도의 시간으로 빠르게 수행할 수 있다.

무작위 패턴에 대한 예상 고장 응답을 저장한 고장 딕셔너리의 경우, s298이나 s349와 같은 작은 회로에 대해서는 예상 고장 응답을 저장하지 않았을 때보다 많은 성능 향상을 보이지만 큰 회로에 대해서는 메모리의 소비의 증가량에 비해 성능 향상 두드러지지 않으므로 회로의 출력 핀의 개수가 많거나 회로의 크기가 큰 경우에는 고장 출력 값을 저장하지 않고 테스트 패턴의 번호만을 기록한 딕셔너리를 사용하는 것이 좋다.

V. 결 론

반도체 설계와 공정 기술의 발달로 회로의 크기가 점점 커지고 복잡도도 증가하게 되었다. 이로 인하여 회로 내의 고장은 더욱 빈번히 발생하게 되었고, 칩의 수율을 증가시키기 위해서 이러한 고장에 대한 진단 과정은 필수적이 되었다. 게이트 수준에서의 고장 위치를 찾아내는 과정은 물리적 수준에서의 고장의 서치 스페이스를 최소화 시킬 수 있다는 점에서 의미를 갖는다. 따라서 본 논문에서는 고장 딕셔너리를 사용하여 효율적으로 고장을 진단하는 방식을 제안하였다. 순차 회로에 대한 기존의 고장 진단 방법인 critical-back-tracing에서의 가장 큰 문제점인 고장 진단 시간을 줄이면서도 정확한 고장 진단 성능을 보이기 위해 고장 딕셔너리와 고장 시뮬레이션 방식을 이용한 고장 진단 알고리듬을 제안하였다. 고장 딕셔너리 방식은 고장 진단 시간을 최소화 할 수 있는 반면 메모리 소비량이 많아지는 단점을 가지고 있다. 그러므로 딕셔너리의 성능은 다소 낮더라도 딕셔너리의 크기를 100만 또는 1000만 게이트급 이상의 큰 회로에서도 적용 가능하도록 최소화 하는 것을 목표로 하여 아주 작은 크기의 딕셔너리 방식을 제안하였다. 이러한 딕셔너리를 사용했을 경우 다소 낮아진 성능은 1차적으로 고장 딕셔너리를 사용한 정적 고장 진단 과정 이후에 얻어지는 고장 후보에 대해서만 동적 고장 진단 과정인 고장 시뮬레이션을 수행하여 보완한다. 고장 딕셔너리는 각각의 고장을 처음으로 검출하는 테스트 패턴의 번호를 저장하는 방식이며, 이를 이용하여 테스트 수행 시 처음으로 고장이 검출 된 패턴을 비교하여 고장 후보를 1차적으로 줄인 후, 고장 후보에 대하여 모든 테스트 패턴에 대한 시뮬레이션을 수행하고 이에 대한 응답을 모두 비교하여 최종 고장 후보를 찾아내는 알고리듬이다. 제안하는 고장 딕셔너리를 이용하면, ISCAS 89 회로 중 가장 큰 회로의 경우, 완전 딕셔너리의 0.001%정도의 메모리를 소비하게 되며 회로의 출력 핀 수의 증가에 따른 영향을 받지 않으므로 회로의 크기가 더 커질수록 완전 딕셔너리와의 크기의 비율은 더 작아질 것으로 예상된다. 또한 딕셔너리 사용 이후, 고장 시뮬레이션을 통해 완전 딕셔너리와 동일한 성능을 갖게 되며, 딕셔너리를 이용함으로써 1차적으로 고장 후보를 1.42%로 줄임으로써 모든 고장에 대한 고장 시뮬레이션으로 고장 진단을 하는 방식보다 그만큼의 시뮬레이션 시간을 단축할 수 있다. 따라서 고장 딕셔너리와 고장 시뮬레이션의 혼합 방식을 사용하여 크기가 큰 순차 회로에 대해 빠르게 고장 진단을 수행할 수 있게 된다.

참 고 문 헌

[1] V. Boppana and W. K. Fuchs "Fault Dictionary compaction by Output Sequence Removal," *Proc. of IEEE ACM Intl. Conf.* pp. 576-579, 1994.

[2] A. W. John and L. Eric, "Failure Diagnosis of Structured VLSI," *Proc. of IEEE Design & Test of Computers,* pp 49 - 60, 1989.

[3] I. Pomeranz and S. M. Reddy, "On the Generation of Small Dictionaries for Fault Location." *Proc. of IEEE Intl. Conf. on Computer Aided Design,* pp.272-279, 1992.

[4] I. Pomeranz and S.M, Reddy, "On Dictionary-Based Fault Location in Digital Logic Circuits," *Transactions on IEEE Computers,* pp. 48-59, 1997.

[5] P. G. Ryan and W. K. Fuchs, "Dynamic Fault Dictionaries and Two-Stage Fault Isolation," *Transactions on IEEE VLSI Systems,* pp. 176-180, 1998.

[6] B. Chess and T. Larrabeee, "Creating Small Fault Dictionaries," *Transactions on IEEE Computer-Aided Design,* pp. 346-356, 1999.

[7] D. B. Lavo and Tracy Larrabee, "Making Cause-Effect Cost Effective : Low-Resolution Fault Dictionaries," Proc. of IEEE Intl Test Conf, pp. 278-286, 2001.

[8] V. Boppana, I.Hartanto and W. K. Fuchs. "Full Fault Dictionary Storage Based on Labeled Tree Encoding," *Transactions on IEEE Computer -Aided Design,* pp. 255-268, 1998.

[9] K. Shigeta, T. Ishiyama, An Improved Fault Diagnosis Algorithm Based on Path Tracing with Dynamic Circuit Extraction、" *Proc. of IEEE International Test Conference,* pp. 235-244, 2000.

[10] K. Shigeta, T. Ishiyama, "A Nnew Path Tracing Algorithm with Dynamic Circuit Extraction for Sequential Circuit Fault Diagnosis," *Proc. of VLSI Test Symposium,* pp. 48-53, 1998.

[11] I. Pomeranz, S. M. Reddy, " Fault Diagnosis Based on Parameters of Output Responses," *Proc. of Pacific Rim International Symposium,* pp. 139-147, 2000.

[12] S. Venkataraman, I. Hartanto, W. Kent Fuchs, " Dynamic Diagnosis of Sequential Circuits Based on Stuck-at Faults," *Proc. of VLSI Test Symposium,* pp. 198-203, 1996.

[13] H. Lee, D. Ha, " Hope: An Efficient Parallel Fault Simulator," *Proc. of Design Automation Conference,* pp 336-340, 1992.

저 자 소 개

김 지 혜(정회원)
2002년 연세대학교 기계전자공학부 전기전자 전공 학사 졸업
2004년 연세대학교 전기전자공학과 석사 졸업
2004년 현재 삼성전자 반도체 총관 System LSI 사업부
<주관심분야 : DFT, CAD, VLSI, Testing>

강 성 호(정회원)
1986년 서울대학교 제어계측공학과 학사 졸업
1988년 The University of Texas at Austin 전기 및 컴퓨터 공학과 석사 졸업
1992년 The University of Texas at Austin 전기 및 컴퓨터공학과 박사 졸업
미국 Schlumberger 연구원, Motorola 선임 연구원
현재 연세대학교 전기전자공학과 부교수
<주관심분야 : SoC 설계 및 SoC 테스트>

이 주 환(학생회원)
2003년 연세대학교 공과대학 전기전자공학과 학사 졸업
2004년 현재 연세대학교 전기전자공학과 석사 과정
<주관심분야 : Logic Diagnosis>