碩士學位論文

VHDL 구조적 레벨에서 효율적인 BIST 설계 기법

濟州大學校 大學院

通信工學科

金 英 愛

2001 年 12 月

VHDL 구조적 레벨에서 효율적인 BIST 설계 기법

指導敎授 林 載 允

金 英 愛

이 論文을 工學 碩士學位 論文으로 提出함

2001 年 12 月

金英愛의 工學 碩士學位 論文을 認准함

審査委員長 李 鎔 鶴 印

委 員 康 鎭 植 印

委 員 林 載 允 印

濟州大學校 大學院

2001年 12 月

Effective BIST(Built-In Self-Test) Design Method in VHDL Structural level

Young-ae Kim
(Supervised by professor Jea-yun Lim)

A thesis submitted in partial fulfillment of the requirement for the degree of Master of Science

2001. 12.

This thesis has been examined and approved.

Thesis director, Yong-hak Lee, Prof. of Telecom. Eng.

(Name and signature)

Date

DEPARTMENT OF TELECOMMUNICATION ENGINEERING
GRADUATE SCHOOL
CHEJU NATIONAL UNIVERSITY

목 차

Abstract

Development of semi-conductor manufacturing process technique is caused much cost and increase of time not only design of circuit but also test of circuit, so BIST(Built-In Self-Test) is widely used by solution method for this test cost and time. However, according as circuit is integrated gradually, it is a problem to increase test time and hardware overhead by data signal and many pin numbers. And existent test pattern generation method has much restriction. Therefore, it must be needed research that can perform test corresponding change of design method and test method.

In this thesis, proposed BIST is presented method of efficient test pattern generation and allocation to reduce test time about circuit and hardware overhead. The first step in this method is that VHDL structural level code convert to graph to analyze easily. The second step is finding node that node length amounts to 1 from nodes to primary input in graph and is making input sets after gathering inputs entering the node. And then it generates test pattern with decreased input number according to each input set. Finally it properly allocates generated test pattern to input sets and performs test.

BIST was performed in several object circuits by proposed algorithm. The applied BIST simulation result shows that test time was remarkably reduced after making input set and hardware overhead was reduced by the compared result of area after synthesis of proposed BIST.

The proposed BIST is expected to be utilized by efficient design and test methodology because there is advantage that reduced design time also.

Ⅰ. 서론

VLSI 회로 설계 기술의 발달로 회로의 집적도가 증가함에 따라 회로에 대한 검사는 점점 더 어려워지고 있으며 검사에 소요되는 비용도 커지고 있다. 따라서 효과적인 검사의 수행은 검사 비용의 절감뿐만 아니라 전체 회로의 개발 시간을 절약하여 양질의 회로를 만들게 하므로 점점 중요하게 여겨지고 있다. 이에 따라 회로의 최적 설계와 더불어 생산된 회로의 테스트가 매우 중요한 문제로 대두되고 있으며, 테스트를 고려한 설계 방법에 대한 연구가 활발히 진행되고 있다. 그 중 가장 널리 연구되고 응용되고 있는 분야는 내장형 자체 테스트(BIST, Built-In Self-Test) 방법이다.

BIST는 회로 내에 자체 테스트 기능을 포함하는 방법으로 자동 테스트 장비(ATE, Automatic Test Equipment)의 사용이나 테스트 응답에 의한 고장 여부 판단 등에 있어서 많은 비용을 절감할 수 있으며, 실제 회로가 동작하는 속도로 테스트를 수행할 수 있는 장점을 갖고 있다. 그러나 BIST를 사용하는데 크게 두 가지 단점을 가지고 있다. 첫 번째는 테스트를 수행하는 부분이 첨가됨으로써 부과 하드웨어 양이 증가한다는 것이다. 하지만 하드웨어의 상대적인 비용 감소와 집적도 증가로 인해 이 문제는 많은 부분 해결 방안이 발표되고 있다. 두 번째 문제는 점점 고집적이 되어감에 따라 입력 수가 증가하여 테스트 시간이 증가한다는 것이다.(Michael J, 1998)

멀티미디어 기술 및 통신 기술의 발달로 인해 현재 VLSI 칩은 여러 기능을 복합적으로 수행할 수 있도록 고기능 및 고집적이 되고 있으며, 칩의 구조도 매우 복잡하게 되고 있다. 따라서 회로 설계의 복잡도가 증가하고 설계 시간의 단축을 위하여 종전의 스캐메틱(Schematic) 설계를 통한 상향식(bottom-up) 설계 방식보다 하드웨어 기술 언어(HDL, Hardware Description Language)를 이용한 하향식(top-down) 설계 방식으로 변화하고 있다. 그리고 그 중 가장 대표적인 IEEE 표준 하드웨어 설계 언어 VHDL(Very high speed integrated circuit HDL)을 이용한 회로 설계가 점점 증가하고 있다. 또 HDL을 사용하여 회로를 설계할 경우 스

캐메틱 방법으로 구현하기 힘든 복잡한 알고리듬을 쉽고 빠르게 구현할 수 있으며, 설계 시간 및 설계 데이터의 관리, 교환 그리고 재사용 등에 훨씬 용이하게 사용된다.

VHDL 설계를 상위 수준 설계라고 하는데 그 중 동작적 레벨(Behavioral level)이나 데이터 흐름 레벨(Data flow level)로 설계할 경우 회로가 복잡해지며, 또한 그 효율이 급격히 떨어지게 된다. 이는 상위 수준에서 동작적 레벨이나 데이터 흐름 레벨로 회로를 설계할 경우 합성 소프트웨어마다 합성할 수 있는 게이트 수가 한정되어 있기 때문이다. 그리고 이러한 레벨 방법으로 회로를 설계하면 합성 시간도 많이 걸려 비효율적이 된다. 또한 합성이 끝난 다음 스캐메틱 sheet에 모든 회로가 펼쳐지기 때문에 회로를 수정할 때나 검증할 때에 곤란을 겪게 된다. 반면 VHDL 상위 수준 설계 방법 중 구조적 레벨(Structural level)로 설계하게 되면 회로가 모듈로 표현되고 하나의 모듈로 회로를 확인할 수 있다. 하지만 동작적 레벨이나 데이터 흐름 레벨을 취하면 전체 회로가 하나의 회로도에 모두 표현되므로 검증과 debugging하기가 어려워진다. 이러한 이유 등으로 상위 레벨 설계에서는 구조적 레벨 설계 방식을 유용하게 사용할 수 있다.(J. R. Armstrong, 1983)(J. R. Armstrong, 1984)

본 논문에서는 회로 설계 방법과 테스트 방법의 변화에 맞는 테스트가 이루어질 수 있도록 VHDL 상위 레벨 중 구조적 레벨로 이루어진 회로에 대해 테스트 시간을 감소시키기 위한 효율적인 테스트 패턴 생성 및 할당을 이루는 BIST를 설계하고자 한다. 또한 기존의 BIST 보다 테스트 패턴의 감소로, 테스트 패턴을 생성시키는 테스트 패턴 생성기의 구조가 간단하게 되어 테스트 회로가 첨가된 전체 회로의 하드웨어 오버헤드를 감소시킬 수 있는 이점을 지니게 된다.

본 논문의 구성을 살펴보면 Ⅰ장에서는 본 논문의 동기와 목적을 제시하고, Ⅱ장에서 일반적인 BIST에 대하여 기술한다. Ⅲ장에서는 VHDL 구조적 레벨로 설계된 회로에서 테스트 패턴을 생성, 할당하는 방법을 제시한다. Ⅳ장에서는 제안한 방법을 적용하기 위한 BIST 설계 기법에 대해 제시한다. Ⅴ장에서 시뮬레이션을 통해 제안한 방법이 개선되었음을 검증하고, 마지막 Ⅵ장에서는 본 논문의 결론을 맺는다.

Ⅱ. 내장형 자체 테스트

VHDL 상위 수준 설계 중 본 논문에서 제안하는 구조적 레벨에서 설계되는 BIST를 설명하기 전 일반적인 BIST에 대해 기술하고자 한다.

1. 일반적인 BIST의 구조

시스템이나 회로를 설계 시 시작할 때부터 테스트를 최대한 쉽게 할 수 있는 방법을 미리 고려하여 설계하는 것은 칩의 생산 기간을 단축하기 위한 좋은 방법이다. 이는 설계 단계에서부터 테스트에 필요한 테스트 용이성을 위한 설계(DFT, Design For Testability)를 미리 고려할 필요가 있다는 것이다. 따라서 시스템의 유용성(Availability)과 신뢰성(Reliability)을 가지고 필요한 기능을 이루기 위해서는 테스트 기능을 가지고 있어야 한다. 특히 시스템이 구성된 후에 테스트 기능을 첨가한다는 것은 아주 어려운 일이다. 따라서 시스템을 구성하기 전 시스템의 기능 중 테스트 기능을 포함시키는 방법이 사용되는데 하드웨어적으로 이러한 방법을 내장형 자체 테스트(BIST, Built-In Self-Test)라고 한다.(Abramovici, 1990)(F. F. Tusi, 1986)

BIST 방법은 회로의 한 부분이 회로 자체를 검사하는데 쓰이는 것으로, 검사 입력이 자체 내에서 생성이 되고 검사의 결과도 자체 내에서 평가되어지는 것이다. 따라서 외부에서 필요한 동작은 검사를 시작시킨 후 검사에 성공했는지 실패했는지 결과만을 확인하면 된다.

BIST의 일반적인 구조는 Fig. 1과 같다. 그림에서 보듯이 BIST를 위한 구조는 테스트 대상 회로(CUT, Circuit Under Test) 즉, 수행하고자 하는 논리 함수를 실현한 회로 이외에 테스트 패턴 발생기(TPG, Test Pattern Generator)와 CUT의 출력을 분석하기 위한 테스트 응답 분석기(TRP, Test Response Processor)로 이

루어진다.

BIST의 동작은 정규 동작 시에는 입력과 출력선을 따라 회로가 정상 동작을 하고 검사 동작 시에는 TPG에서 생성된 테스트 패턴으로 회로를 검사한 후 결과를 TRP에서 분석하여 회로의 검사 유무를 나타내게 된다. 따라서 이 기법은 외부의 비싼 검사기(Tester)를 필요로 하지 않으며, 정규 동작 속도로 회로를 검사할 수 있게 된다.

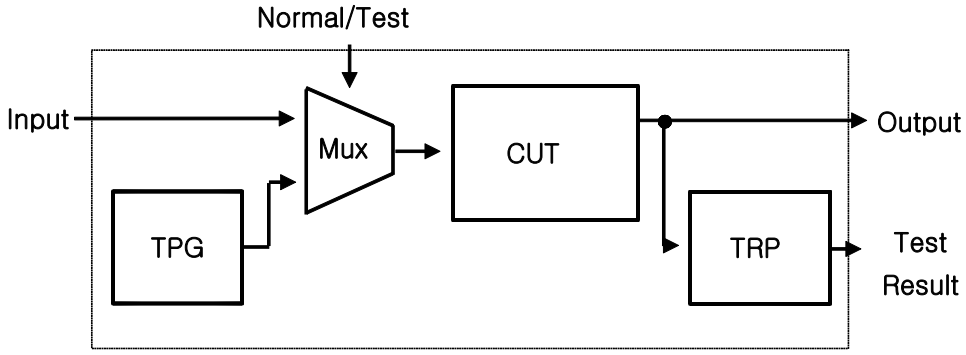


Fig. 1. General structure of BIST

2. 테스트 패턴 발생기

BIST를 이용하여 회로를 검사하기 위해서는 테스트 입력을 자체 생성해 내어야 한다. 이런 테스트 패턴을 칩 내부에서 생성해 내는 방법으로는 ROM 등의 메모리에 테스트 패턴을 저장하는 방법과 전 경우 테스팅(Exhaustive testing), 의사 전 경우 테스팅(Pseudo-exhaustive testing), 그리고 의사 무작위 테스팅(Pseudo-random testing) 방법 등이 있다.(Parag K, 1997)

그 중 ROM 등의 메모리에 테스트 패턴을 저장하는 방법은 과도한 메모리 양을 필요로 하기 때문에 부적합하다.

전 경우 테스팅 방법은 n개의 입력을 갖는 조합회로에 대해 2^n개의 모든 가능한 입력 조합을 인가하여 테스트하는 방법으로 이를 위하여 2진 카운터가 테스트 패턴 생성기로 사용될 수 있다. 이 방법은 순차적인 동작을 유발시키지 않고도 탐지할 수 있는 모든 고장들이 탐지되는 것을 보장한다. 단, n이 22보다 더 큰 경우에는 필요한 테스트 패턴이 너무 많기 때문에 이 방식은 적합하지 않다. 따라서 일반적으로 n이 큰 경우에는 다른 방식이 훨씬 더 효과적이다.

다음으로 의사 전 경우 테스팅은 전 경우 테스팅의 장점을 모두 갖으며 훨씬 작은 수의 테스트 패턴을 요구한다. 이 방식은 다양한 형태의 회로 분할에 의존하며 각 회로는 전 경우 테스트 방식에 의해 테스트된다. 예를 들어 n개의 입력을 갖는 회로를 의사 전 경우 테스트하고자 할 때 단지 m 개 (m<n)의 입력에 대한 의사 전 경우 테스트 패턴을 생성할 수 있도록 입력을 재배열할 수 있다.

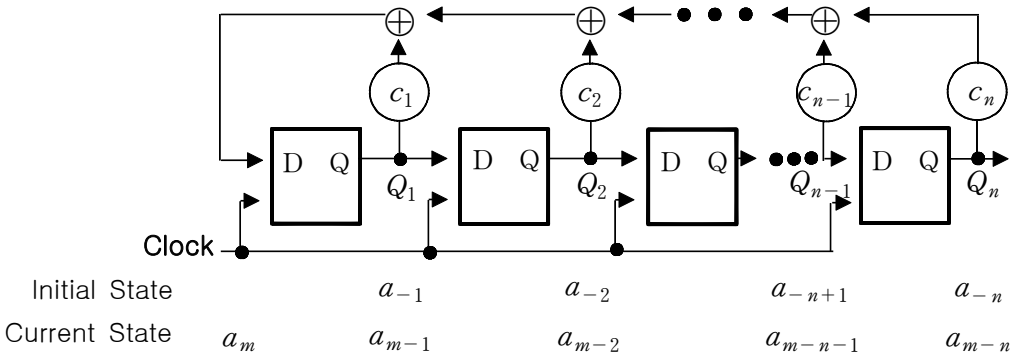
의사 무작위 테스팅은 무작위(random)한 성격을 갖는 테스트 패턴을 사용하는 방법으로 실제로는 패턴의 발생 순서가 정해져 있어 테스트 패턴이 일정 길이로 반복이 된다. 또한 모든 검사 입력이 같은 확률로 생성되며 같은 패턴이 반복되지 않는다.

때로는 만족할 만한 고장 검출률을 얻는데 있어서 2^n개의 모든 패턴의 조합을 생성할 필요는 없을 수도 있다. 즉, 회로의 여러 부분들은 특성에 따라 다양한 비균일 분포를 갖는 의사 무작위 패턴을 이용하여 보다 더 효과적으로 테스트할 수 있다. BIST에서 가장 널리 사용되는 테스트 패턴은 의사 무작위 패턴으로, 이는 패턴 자체의 우수성에 있다기보다는 하드웨어 구성의 편의성 때문이다. 그리고 생성된 패턴들을 나열하다 보면 마치 무작위 패턴처럼 일정한 순서가 없는 것 같이 보이지만 일정한 길이의 패턴들이 반복된다. 의사 무작위 테스트 알고리듬의 가장 대표적인 패턴 생성기는 선형 귀환 쉬프트 레지스터(LFSR, Linear Feedback Shift Register)로 본 논문에서는 테스트 패턴 생성기로 LFSR을 사용하였다.

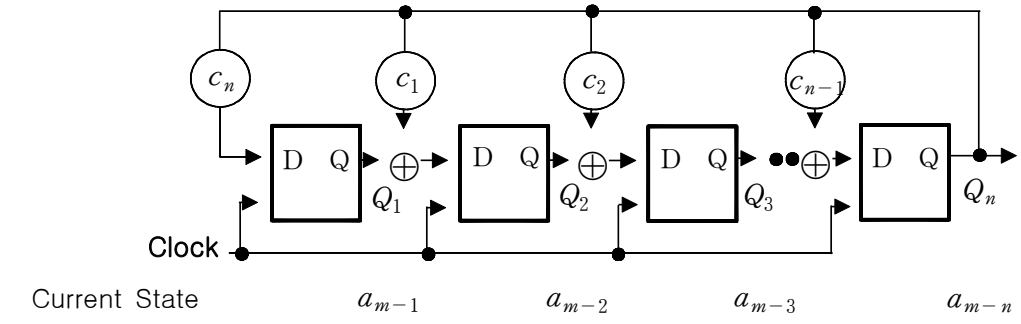
LFSR은 구조가 단순하고 구현하기가 쉽기 때문에 BIST에서 테스트 패턴을 발생시키기 위한 테스트 패턴 발생기로 많이 사용된다. 원래 LFSR은 부호 이론에서 가장 먼저 사용되었는데, 통신에서 사용되는 코드 이름은 CRC(Cyclic Redundant Code)라고 하며 BIST에서도 이 코드를 그대로 사용한다.(K. Kim, 1988)(Nur A.,

1994)(Nur A., 1996)

일반적으로 사용되는 LFSR의 형태는 Fig. 3의 Type 1의 외부형 LFSR과 Type 2의 내부형 LFSR의 두 가지로 분류될 수 있다.



(a)



(b)

Fig. 2. The types of LFSR

(a) Type 1. LFSR : External LFSR (b) Type 2. LFSR : Internal LFSR

Fig. 2에서 이진 상수 C_n는 그 값이 '0' 이면 연결되지 않음을 뜻하고, '1' 이면 연결됨을 뜻한다.

임의의 이진 숫자들의 수열 a_0, a_1, a_2, ..., a_m,...은 생성함수 $G(x)$라고 부르는 식 (1)과 같은 다항식으로 나타낼 수 있다. 이 때 수열 $\{a_m\} = a_0$, a_1, a_2, ...이 LFSR에 의해 생성되는 출력 값이라고 가정하면, 이 수열은 식 (2)과 같이 표시된다.

$$G(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m \ldots \tag{1}$$

$$G(x) = \sum_{m=0}^{\infty} a_m x^m \tag{2}$$

Fig. 2(a)와 같은 Type 1 형태를 갖는 LFSR의 경우에 Q_i 의 현재 상태를 a_{m-i} 라 하면, a_m 은 다음과 같이 표현된다.

$$a_m = \sum_{i=1}^{n} c_i a_{m-i} \tag{3}$$

LFSR의 초기값을 a_{-1}, a_{-2}, ..., a_{-n+1}, a_{-n} 이라 가정하면, 식 (2)는 식 (4)와 같이 표현된다.

$$\begin{aligned} G(x) &= \sum_{m=0}^{\infty} a_m x^m \\ &= \sum_{m=0}^{\infty} \sum_{i=1}^{n} c_i a_{m-i} x^m \\ &= \sum_{i=1}^{n} c_i x^i \sum_{m=0}^{\infty} a_{m-i} x^{m-i} \\ &= \sum_{i=1}^{n} c_i x^i [a_{-i} x^{-i} + \ldots + a_{-1} x^{-1} + G(x)] \end{aligned} \tag{4}$$

$$G(x) = \frac{\sum_{i=1}^{n} c_i x^i [a_{-i} x^{-i} + \ldots + a_{-1} x^{-1}]}{1 + \sum_{i=1}^{n} c_i x^i} \tag{5}$$

식 (5)의 $G(x)$는 LFSR의 초기값인 a_{-1}, a_{-2}, ..., a_{-n}과 귀환 상수 c_1, c_2, ..., c_n으로 이루어진 함수이다. $G(x)$의 분모는 식 (6)처럼 표시되며 이를 특성 다항식(Characteristic Polynomial, $P(x)$)라 한다.

$$P(x) = 1 + c_1 x + c_2 x^2 + \ldots + c_n x^n \tag{6}$$

특성 다항식은 LFSR의 구조에 따라 결정되며 플립플롭의 개수가 n 개일 때 LFSR에서 출력할 수 있는 반복되지 않은 테스트 패턴의 수는 ($2^n - 1$)개로 가장 긴 수열의 길이는 ($2^n - 1$)이 된다. 이러한 수열을 발생시킬 수 있는 특성 다항식을 기본 다항식(Primitive polynomial)이라 부른다. 이를 위해서 기본 다항식은 자기 자신과 1 이외에는 나누어지지 않는 형태를 가지고 있어야 한다.

Table 1. Examples of LFSR primitive polynomial

1 : 0	9 : 4 0	17 : 3 0
2 : 1 0	10 : 3 0	18 : 7 0
3 : 1 0	11 : 2 0	19 : 6 5 1 0
4 : 1 0	12 : 7 4 3 0	20 : 3 0
5 : 2 0	13 : 4 3 1 0	21 : 2 0
6 : 1 0	14 : 12 11 1 0	22 : 1 0
7 : 1 0	15 : 1 0	23 : 5 0
8 : 6 5 1 0	16 : 5 3 1 0	24 : 4 3 1 0

표 1은 n이 1과 24 사이에 존재할 때 그 각각에 대해 한 개씩의 기본 다항식을 보여주고 있다. 예를 들어, 12 : 7 4 3 0 은 $x^{12}+x^7+x^4+x^3+1$ 을 표현하며, 이 다항식에 의한 최대 수열 길이는 $(2^{12}-1)$이며, $(2^{12}-1)$개의 테스트 패턴을 생성한다.

Fig. 3 에서 4 bit LFSR의 예를 보여주고 있다.

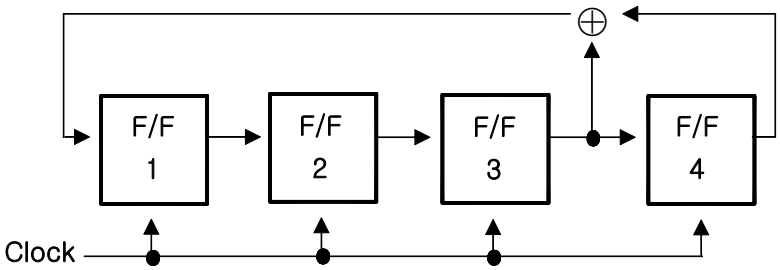


Fig. 3. 4 bit LFSR with primitive polynomial

Fig. 3에서 LFSR의 회로는 클럭 이외에 다른 입력은 없고 플립플롭과 EX-OR로만 이루어져 있으며, 표 2에서 보여주듯이 출력 값들은 일정 길이를 갖고 반복되는 특징을 갖는다. n 개의 플립플롭을 갖는 이진 카운터가 $0, 1, \ldots, 2^n-1$ 까지의 2^n 개의 상태를 계속해서 반복하듯이 n개의 플립플롭을 갖는 LFSR은 플립플롭의 초기값이 모두 0일 때는 플립플롭의 상태가 변하지 않으므로 최대 (2^n-1)개의 상태를 갖는다. 그림에서의 회로는 (2^4-1)개의 패턴을 생성해 내며 15개의 반복되는 상태를 갖는 경우를 보여주고 있다. 이 때 초기값이 모두 0인 경우에는 상태가 변하지 않아 항상 0 만을 출력하고 초기값이 0 이 아닌 경우에는 (2^n-1)개의 길이를 갖는 신호를 생성해 내는 회로를 최대 길이를 갖는 쉬프트 레지스터라고 한다.

그리고 LFSR에 의한 의사 무작위 패턴의 성질 중, n개의 플립플롭을 갖는 LFSR이 (2^n-1)개의 패턴을 가질 때, 그 중 (2^k-1)개의 의사 무작위 패턴을 선

택할 수 있는 $k(k \leq n)$ 개의 플립플롭을 포함하는 성질을 갖는다. 표 2에서 보면 4 비트 중 무작위로 두 개 또는 세 개의 비트를 선택하여도 선택된 비트에서 생성되는 패턴들 중 의사 무작위 패턴이 포함됨을 알 수 있다(Solomon, 1982). 즉, 표에서 4 비트의 의사 무작위 패턴 중 임의로 F/F 1, F/F 2, F/F 4 또는 F/F 2, F/F 3, F/F 4를 선택하였을 때 이 세 비트로 생성된 패턴 중 의사 무작위 패턴이 포함됨을 알 수 있다.

Table 2. Pattern generation by 4 bit LFSR with primitive polynomial

State	F/F 1	F/F 2	F/F 3	F/F 4
s_0	1	1	1	1
s_1	0	1	1	1
s_2	0	0	1	1
s_3	0	0	0	1
s_4	1	0	0	0
s_5	0	1	0	0
s_6	0	0	1	0
s_7	1	0	0	1
s_8	1	1	0	0
s_9	0	1	1	0
s_{10}	1	0	1	1
s_{11}	0	1	0	1
s_{12}	1	0	1	0
s_{13}	1	1	0	1
s_{14}	1	1	1	0
$s_{15} = s_0$	1	1	1	1

결국 기본 다항식에 의해 구현된 LFSR이 생성하는 수열을 의사 무작위 수열이라 부르며, 여기서 LFSR은 초기값만 설정하면 클럭에 의해 자동적으로 테스트 패

턴을 생성하므로 BIST의 TPG로 매우 적합하다는 것을 알 수 있다.

3. 테스트 응답 분석기

테스트 대상 회로를 검사하는 과정은 정확하게 동작할 때 기대되는 회로의 출력과 테스트 패턴을 입력으로 가했을 때의 출력을 비교하는 것이다.

BIST 방법으로 검사를 수행할 경우 정상 동작시의 출력 값을 ROM등에 저장해 두어야 하는데, 많은 수의 입력이 생성되는 경우 많은 기억 용량을 필요로 한다. 따라서 이렇게 출력되는 데이터의 양이 많기 때문에 압축 기술이 필요하다. (Vishwani D, 1993)

Fig. 1에서 TRP는 주로 테스트 데이터의 검사와 비교를 수행하고, 테스트 데이터를 압축하여 최종적으로 검사해야 하는 데이터 양을 줄이는 동작을 수행한다. 이와 같이 압축을 사용했을 때 오버헤드를 줄이는 장점이 있지만, 고장 시 부호가 정상 동작의 부호와 같아져서 고장이 있는 회로를 고장이 없는 회로로 판단하게 되는 에러 소실(Aliasing)이 생길 수 있다. 따라서 에러 소실이 생기는 확률을 예측하고, 이를 줄이는 것 또한 중요하다.

TRP는 압축을 수행하는 방법에 따라 여러 가지가 연구되어 왔다.(Parag K, 1997) 그 중 가장 간단한 방법은 출력에 나타나는 1의 개수를 부호로 이용하는 1계수 압축 기법으로 n개의 검사 입력을 가한다면 부호는 0과 n사이의 값이 되고 따라서 이를 위한 계수기가 필요하게 된다. 만일 정상 동작 시 1의 수가 p개라면 이는 p개의 1과 ($n-p$)개의 0으로 출력이 나오는데 이 중 순서가 바뀌는 경우 고장이 있지만 에러 소실이 된다. 따라서 정확한 p와 에러 소실되는 수열은 (${}_nC_p-1$)이다. 따라서 에러 소실의 확률은 다음과 같이 표시된다.(K. Kim, 1988)

$$P_{aliasing} = \frac{{}_nC_p - 1}{2^n - 1} \tag{7}$$

다음으로 신드롬(Syndrome) 검사 방법은 n개의 입력을 가진 조합 회로에 2^n개의 전수 입력을 가해서 그 때의 1의 개수가 나올 확률을 부호로 사용하게 되는데 이를 신드롬이라고 한다. 논리 함수에서의 신드롬은 다음과 같이 정의할 수 있다. 여기서 K는 논리 함수에서 모든 변수를 포함한 최소 논리 함수의 항의 수이고, n은 입력 수이다.

$$S = \frac{K}{2^n} \tag{8}$$

예를 들어, 3개의 입력을 갖는 AND 게이트의 신드롬은 $1/8$이고, 2개의 입력을 갖는 OR 게이트의 신드롬은 $1/4$이다. 또한 같은 함수 값을 나타내지만 다른 구현 방법으로 구성된 함수도 같은 신드롬을 가진다.

다른 테스트 응답 분석 방법에는 천이 카운터(Transition Counter) 방법이 있다. 이는 출력 순서에서 0에서 1 또는 1에서 0으로의 천이 수를 사용하여 부호로 사용하는 것이다. 예를 들어, 출력 시퀀스가 $Z = 100111010$ 라면 천이 카운터 $c(Z) = 5$ 가 된다. 따라서, 전체의 출력 시퀀스를 저장하는 대신 논리적으로 천이 수만 가지고 분석할 수 있게 된다. 천이 수가 고장이 발생하지 않은 상태의 천이 수와 다르다면, CUT에 고장이 발생하였다고 할 수 있다. 천이 카운터 방법은 완전한 출력 시퀀스나 정확한 출력 시퀀스를 알 필요가 없으므로 쉽게 데이터를 압축시킬 수가 있지만, 다른 종류의 고장 에러를 발생시킬 수가 있다. 즉, 고장이 발생한 시퀀스의 논리적인 천이 수가 고장이 발생하지 않은 상태의 천이 수와 같은 에러 소실이 생기게 되는 것이다.

정상 회로에서 천이 계수의 부호가 p라면 n개의 검사 입력을 가했을 때 부호는 최대 ($n-1$)까지 가능하고, 이 중에서 순서가 바뀌는 경우는 ${}_{n-1}C_p$ 이고, 0과 1이 바뀌어도 마찬가지이므로 모든 가능한 순서는 ${}_{2n-1}C_p$ 로 된다. 따라서 에

러 소실 확률은 다음과 같다.

$$P_{aliasing} = \frac{{}_{2n-1}C_p - 1}{2^n - 1} \tag{9}$$

TRP 압축 기술 중 현재 가장 널리 사용되고 있는 방법은 부호 분석기(Signature Register) 방법이다. 이 방법은 긴 데이터 열을 여러 비트의 부호(Signature)라 불리는 단일 코드로 압축시키는 방법으로, 부호는 긴 데이터 열을 n 비트의 LFSR에 입력시킴으로 얻을 수 있다. Fig. 5는 그 구성을 보였다. 데이터 열이 시리얼로 입력이 되면 부호 분석기의 쉬프트 레지스터에는 입력과 LFSR에서 하나의 레지스터 상태가 EX-OR 된 논리가 존재하게 된다. 부호 분석기의 최종 레지스터 상태는 데이터의 나머지를 갖게 되는데 이러한 나머지가 각 데이터 열의 특성을 표현하는 부호가 된다. 즉, 회로의 출력을 나눗셈의 입력으로 하고 LFSR을 제산 회로로 사용하는 방법으로 여기서 출력된 부호가 나눗셈의 몫이 되어 출력을 압축시키게 된다.

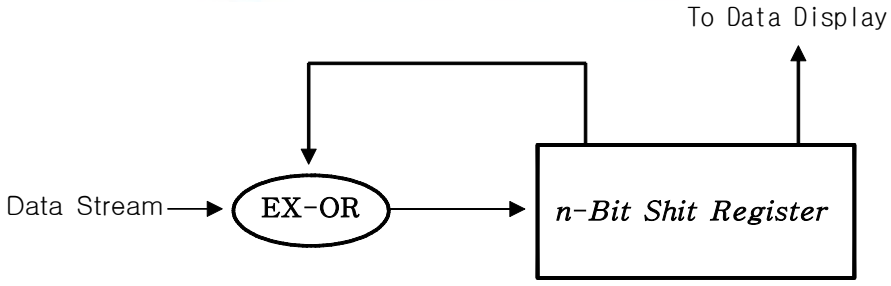


Fig. 4. Signature Register

Fig. 4의 회로를 사용할 경우 출력 단자에서 출력되는 값과 최종적으로 플립플롭에 남아 있는 값들을 압축된 테스트 응답으로 생각할 수도 있다. 하지만 출력 단자를 계속 검사하는 것은 번거로운 일이므로, 최종적으로 플립플롭에 남아 있는 나머지를 압축된 데이터로 간주하고 이를 검색하는 것으로 고장 진단을 수행한다.

이 때 플립플롭에 압축된 데이터를 부호(Signature)라고 하고, Fig. 4와 같은 회로를 부호 분석기(Signature Register)라고 한다.

Fig. 5는 4 bit 부호 분석기의 예를 나타내고 있다.

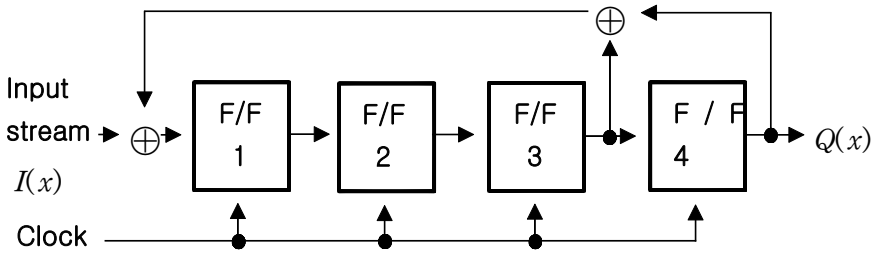


Fig. 5. 4 bit Single Input Signature Register(SISR)

그림에서 입력 데이터 열이 $I(x)$이고 출력 신호가 $Q(x)$로 나타내고, 초기 값이 0 이라 가정하자. 그리고 플립플롭의 마지막 상태를 $R(x)$라고 표시하면 식 (10)이 성립하게 된다.

$$\frac{I(x)}{P(x)} = Q(x) + \frac{R(x)}{P(x)} \tag{10}$$

식 (10)에서 $P(x)$는 LFSR의 기본 방정식이다. Fig. 5는 하나의 입력 단자를 갖는 부호 분석기이므로, 단일 입력 부호 분석기(SISR , Single Input Signature Register)이라고 부른다.

n개의 플립플롭이 사용되었을 때 에러 소실이 생길 확률을 보면, m개의 검사 입력이 사용될 경우 다음 식과 같이 에러 소실이 일어날 확률이 생긴다.

$$P_{aliasing} = \frac{2^{m-n}-1}{2^{m}-1} \tag{11}$$

만약 $m \gg n$이면 결국 아래 식과 같이 된다.

$$P_{aliasing} \approx \frac{1}{2^n} \tag{12}$$

Fig. 5의 SISR은 테스트 데이터 입력 단자를 하나만 갖고 있어서 여러 개의 출력 단을 갖는 회로로 그 응용이 적합하지 않다. 이를 보완하는 한 방법으로 CUT 출력 단의 병렬 응답을 각각 EX-OR를 통해 LFSR로 입력하는 방법을 사용한다. 이 때 사용되는 것을 Fig. 6의 다중 입력 부호 분석기(MISR, Multiple Input Signature Register)라고 한다. 이 구조는 단일 입력 구조에서 입력 단자를 적절히 옮긴 네 개의 LFSR를 중첩한 것과 같은 효과를 갖는다. 따라서 입력 데이터 열의 길이가 충분히 길다면, 식 (12)와 거의 같은 에러 소실 확률을 갖는다. 또한 이 회로를 사용하면 병렬로 데이터를 처리할 수 있다.

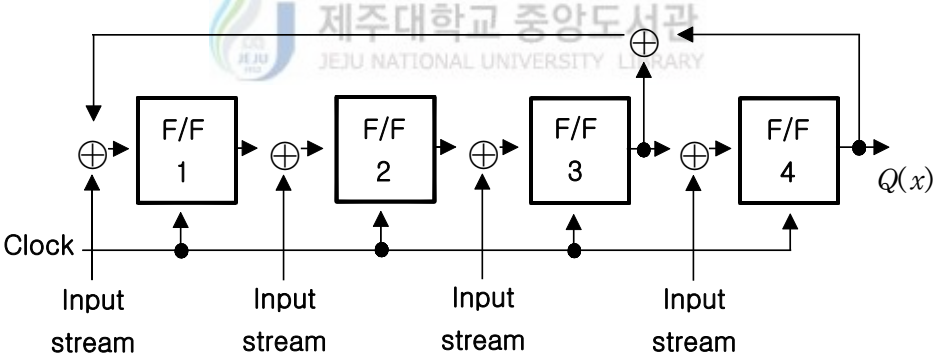


Fig. 6. 4 bit Multiple Input Signature Register

Ⅲ. 구조적 레벨에서의 테스트 패턴 생성 및 할당

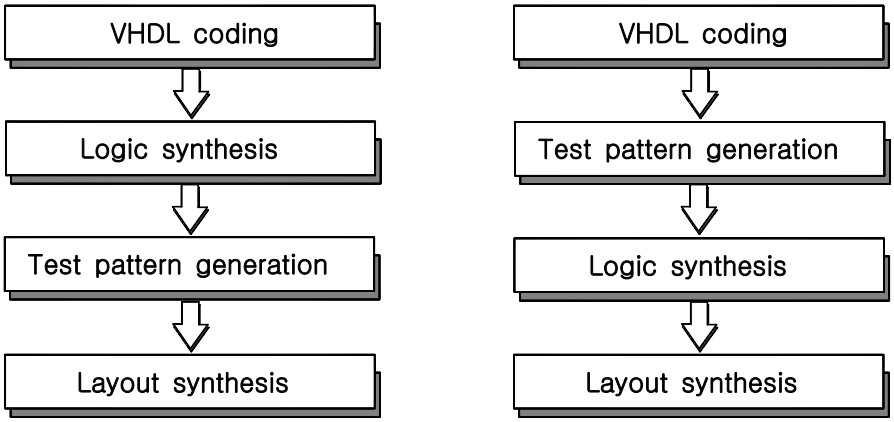
본 장에서는 2장에서 기술한 BIST를 이용하여 효율적인 테스트가 이루어지도록 VHDL 상위 수준 중 구조적 레벨로 설계된 회로에 효율적으로 적용될 수 있는 테스트 패턴 생성 및 할당 방법을 제안한다.

1. VHDL을 이용한 설계 순서

기존의 테스트 방법에서 VHDL을 이용하여 설계된 회로를 구현할 경우 합성 과정이 끝난 후에 테스트 패턴이 생성된다. 즉, VHDL을 이용하여 회로를 설계할 경우 설계 초기에는 테스트를 고려하기가 힘들게 되는 단점을 가지게 된다. 이런 단점을 극복하기 위해 VHDL로 기술된 회로를 가지고 테스트 패턴을 생성하면 설계 초기에 테스트를 고려할 수 있다.

Fig. 7은 본 논문에서 제안하는 테스트를 위한 설계 순서를 나타내는 그림이다. Fig. 7(a)에서 보듯이 기존의 설계 순서는 VHDL로 기술된 코드를 합성 Tool을 이용하여 합성을 시킨다. 그런 다음 테스트 패턴을 생성하고 레이아웃 합성을 통하여 칩을 만들게 되는 것이다. 하지만 회로가 복잡해질수록 VHDL 합성 결과인 게이트 수준 회로가 복잡하게 되어 테스트 패턴을 생성시키기도 힘들게 된다. 하지만 Fig. 7(b)과 같이 설계하고자 하는 회로를 VHDL 코드로 설계한 뒤 테스트 패턴을 VHDL로 설계, 회로에 추가하여 합성시키면 쉽게 테스트 패턴을 생성시킬 수 있게 된다. 그리고 합성 단계와 무관한 패턴을 생성할 수 있으므로 패턴 재사용이 가능하다.

결국 VHDL 설계 단계에서 테스트 패턴을 생성한다면 회로를 이해하고 테스트 패턴을 생성할 수 있으므로 보다 효율적으로 테스트 패턴 생성이 될 수 있다.



(a) (b)

Fig. 7. Design flow

(a) The classic design flow (b) The proposed design flow

2. 구조적 레벨 도표 변환

본 논문에서는 구조적 레벨로 설계된 VHDL 코드를 분석하기 편하게 그리고 효율적으로 테스트 패턴을 생성, 할당하기 위한 방법으로 먼저 VHDL의 구조적 레벨로 설계된 코드를 도표로 변환시킨다. 이렇게 코드를 도표로 변환시키면 전체적인 회로의 구조와 입출력 및 기능을 쉽게 분석할 수 있다.

VHDL 구조적 레벨 설계는 netlist 표현 방식과 유사하여 동작적 레벨 설계보다 하드웨어 구조에 가까운 설계 방식이다. 이는 이미 설계된 component들을 이용하여 그 component들을 서로 연결하여 시스템을 기술하려는 방식으로 component와 게이트와의 상호 연결을 나타낸다. 여기서 component란 단순히 게이트 또는 플립플롭이거나 동작적 레벨 내지 RTL 코드로 기술된 좀 더 큰 블록으로 이는 이전에 설계된 entity나 IP(Intellectual Property) 모듈을 말한다. 이러한 component들을 마치 논리 게이트처럼 여겨 이에 따른 입출력 관계를 연결하여 도표로 표현할 수 있다. 이 때 component들과 게이트들을 노드(Node)라고 하고 이

노드들을 연결하는 입/출력선을 신호선이라 할 수 있다.

VHDL 구조적 레벨 설계의 도표 변환 예를 살펴보면, 먼저 Fig. 8은 동작적 레벨로 설계된 Full adder 코드이다. 이는 Fig. 9의 Ripple-carry adder의 VHDL 구조적 레벨 설계 시 component로 이용된다.

```
Library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
  port( a, b, c : in std_logic;
        sum, ca : out std_logic);
end full_adder;
architecture behavioral of full_adder is
  begin
    sum <= a xor b xor c;
    ca <= (a and b) or (b and c) or (a and c);
end behavioral;
```


Fig. 8. VHDL code of Full adder

```
Library ieee;
use ieee.std_logic_1164.all;
entity ripple_carry_adder is
  port( cin : in std_logic;
        x, y : in std_logic_vector(3 downto 0);
        s : out std_logic_vector(3 downto 0);
        cout : out std_logic);
end ripple_carry_adder;
architecture structure of ripple_carry_adder is
    component full_adder
        port( a, b, c : in std_logic;
              s, cout : out std_logic);
    end component;
    signal c : std_logic_vector(3 downto 1);
begin
    stage0 : full_adder port map (x(0), y(0), cin, s(0), c(1));
    stage1 : full_adder port map (x(1), y(1), c(1), s(1), c(2));
    stage2 : full_adder port map (x(2), y(2), c(2), s(2), c(3));
    stage3 : full_adder port map (x(3), y(3), c(3), s(3), cout);
end structure;
```


Fig. 9. VHDL code of 4 bit Ripple-carry adder

Fig. 9의 구조적 설계는 Fig. 10에서 보는 것처럼 노드와 신호선, 입출력이 서로 연결된 도표로 변환할 수 있다.

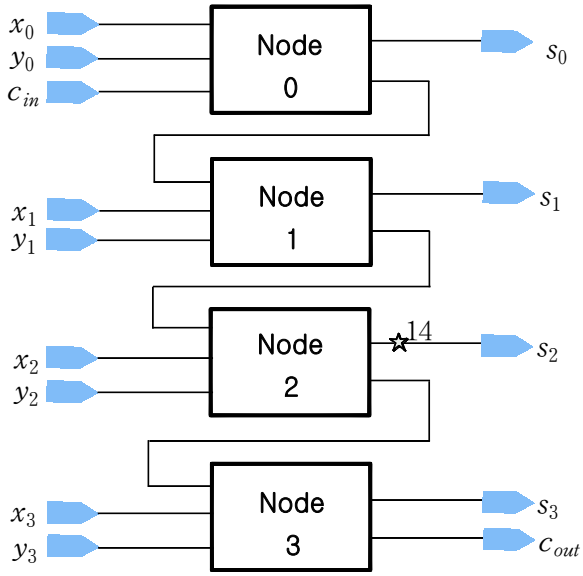


Fig. 10. Converted graph from the VHDL code of Fig. 9

3. 고장 모델

임의의 회로나 시스템이 원래 의도했던 동작에서 벗어날 때, 이를 고장이라고 한다. 회로에서의 고장은 신호선의 단선(Open), 전원이나 다른 신호선과 단락(Short)이 되었거나, 신호의 지연(Delay)등에 의해 발생한다. 이러한 것을 모델링하여 표현한 것을 고장 모델(Fault Model)이라 한다. 고장 모델을 구성하는 목적은 임의의 칩이나 회로의 일부가 고장이 발생하였을 때 어떤 영향을 미치는가를 알아보고, 이러한 고장을 논리함수로 모델링함으로써 여러 종류의 고장에 대한 검출이나 표현을 가능하게 하기 위함이다. 일반적인 고장 모델에서의 고장의 종류로는 Stuck-at 고장, Stuck-open 고장, Stuck-on 고장, Bridging 고장 그리고 Delay 고장 등이 있다.

이 중 게이트 레벨에서 주로 사용되는 고장 모델은 단일 고착 고장(Stuck-at

fault)이다. Stuck-at 고장은 임의의 노드의 입력이나 출력이 논리적으로 변하지 않고, 특정한 논리 값(0 또는 1)으로 고정되며 논리 값이 0이냐, 1이냐에 따라서 Stuck-at 0 고장과 Stuck-at 1 고장으로 나누어진다.

본 논문에서 제안하는 테스트 생성 및 할당 방법은 VHDL 상에서 수행되며, 원하는 논리 수행을 방해하는 고장을 검출하는 것을 목적으로 하고 있다. VHDL 상위 수준 설계에서는 게이트 레벨에서와 달리 IC의 물리적인 고장과 직접 연관된 고장 모델을 규정하기가 어렵다. 따라서 본 논문에서 사용되는 고장 모델은 게이트 레벨에서의 고장 모델을 응용한 VHDL에서의 고장 모델을 정의한다.

본 논문에서는 Fig. 11에서 보는 것과 같이 VHDL의 구조적인 레벨로 기술된 코드를 변환한 도표에서 도표상의 노드(component 또는 gate)들을 연결하는 신호선에 가상적인 고장을 고장 모델로 선택하였다. 게이트 레벨에서의 고착 고장을 이용하여, 신호선에서 특정 논리 값으로 고정되는 즉, 신호선상의 고장을 VHDL 상의 고장 모델로 정의하여 신호선의 고장으로 인해 데이터가 올바르게 전달되지 않는 것을 VHDL상의 고장으로 가정하였다. 가정한 VHDL 고장 모델에 대한 고장이 아닌 component 내부에서 고장이 발생하더라도 결국 component에서 나오는 고장이 component 출력 신호선에 전달되어 전체 출력으로도 전달되므로 그런 고장 또한 판단할 수 있게 된다.

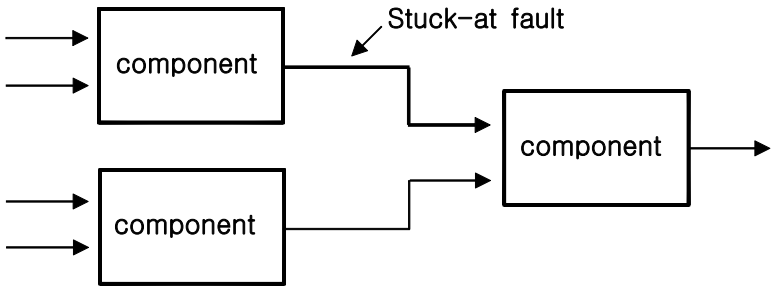


Fig. 11. VHDL fault model

4. 테스트 패턴 생성 및 할당 알고리듬

일반적인 BIST에서 입력 수 n 이 증가하면 테스트 길이가 (2^n-1)으로 증가하게 되어 테스트 시간이 매우 길어진다. 예를 들어 만약 50개의 입력을 갖는 회로에서 100 MHz의 테스트 패턴을 사용할 경우 약 130일의 테스트 시간이 걸리게 된다. 이런 BIST의 단점을 보완하기 위해 본 논문에서는 회로로 들어가는 입력들을 적절히 묶어 입력들을 집합화하고자 한다. 그런 다음 각 입력 집합에 속한 입력들에 대해 테스트 패턴을 생성하고 생성된 패턴을 효율적으로 각 입력 집합에 할당시키려 한다. 이렇게 생성 및 할당된 테스트 패턴을 이용하여 테스트하고자 하는 회로에 대해 BIST를 수행할 수 있다. 따라서 이런 방법을 이용하여 BIST에 대한 테스트 시간을 줄이고, 또한 회로 설계가 VHDL 상에서 이루어지므로 회로를 설계하는 시간을 감소시킬 수 있게 된다.

먼저 본 논문에서 사용되는 용어들을 정리하면 다음과 같다.

- 노드($Node$, N_j)

  VHDL의 구조적 레벨로 상위 수준 설계된 코드를 변환한 도표에서의 component 또는 gate

- 노드의 길이($Length\ PI_i(N_j)$)

  변환한 도표에서 특정 주 입력 (PI_i , $Primary\ Input$)에 대하여 PI_i 에 부터 기준 N_j까지 거쳐야 하는 노드 수

- 입력 집합 ($Input\ Set$, $IS(N_j)$)

  노드 길이가 1인 N_j들에 대하여 그 N_j로 들어가는 입력들의 집합, 만약 PI_i 이외의 입력 즉 중간 노드의 출력 값이 입력 집합의 입력이 되더라도 표기를 적절히 하여 중간 노드의 출력 값을 입력 집합의 입력으로 표기

1) 입력 집합 구성

많은 수의 테스트 패턴을 생성시키는 입력들을 줄이기 위해서는 서로 연관 있는 입력들을 묶어 집합으로 만든 후 원래의 입력 수보다는 줄어든 입력 수를 갖는 입력 집합을 여러 개 만들 수 있다. 연관 있는 입력들이란 같은 N_j로 들어가는 입력들을 말하는 것으로 같은 N_j로 들어가는 입력은 PI_i 뿐만 아니라 중간 N_j에서 나온 출력이 다시 N_j로 들어가게 되는 중간 입력도 갖는다. 이런 입력 중 PI_i이 입력으로 들어가는 N_j들에 대해 $Length\ PI_i(N_j)$을 구하여 찾을 수 있다. 각 N_j로 들어가는 모든 입력들에 대한 길이를 구하기보다는 PI_i에서 N_j로 들어가는 길이가 1이 되는 즉, 직접 N_j로 들어가는 PI_i들을 찾으면 된다. 이렇게 $Length\ PI_i(N_j)$가 1인 N_j들에 대해 입력들을 모아 $IS(N_j)$을 만들 수 있다.

Fig. 9에 있는 4 비트 Ripple-carry adder의 변환된 Fig. 10의 도표를 이용하여 $IS(N_j)$을 구하기 위해 먼저 Fig. 12처럼 각 PI_i에 대해 $Length\ PI_i(N_j)$가 1인 N_j을 찾을 수 있다.

$Length\ x_0(Node\ 0)=1$	$Length\ x_2(Node\ 2)=1$
$Length\ y_0(Node\ 0)=1$	$Length\ y_2(Node\ 2)=1$
$Length\ c_{in}(Node\ 0)=1$	$Length\ x_3(Node\ 3)=1$
$Length\ x_1(Node\ 1)=1$	$Length\ y_3(Node\ 3)=1$
$Length\ y_1(Node\ 1)=1$	

Fig. 12. Nodes with $Length\ PI_i(N_j)=1$ of Fig. 10

Fig. 12에서 보는 것처럼 $Length\ PI_i(N_j)$가 1인 N_j는 $Node\ 0$, $Node\ 1$, $Node\ 2$, 그리고 $Node\ 3$이다. 이렇게 찾은 N_j들에 대해 각 N_j로 들어가는 입력들을 묶으면 $IS(N_j)$을 만들 수 있게 되는데, Fig. 13은 각 N_j들에 대한 $IS(N_j)$들을 나타낸 것이다. 이 때 c_1, c_2, c_3는 중간 N_j의 출력 값으로 다시 N_j로 들어가는 입력들을 적절하게 표기한 것이다.

$$IS0(Node\ 0) = \{x_0,\ y_0,\ c_{in}\}$$
$$IS1(Node\ 1) = \{x_1,\ y_1,\ c_1\}$$
$$IS2(Node\ 2) = \{x_2,\ y_2,\ c_2\}$$
$$IS3(Node\ 3) = \{x_3,\ y_3,\ c_3\}$$

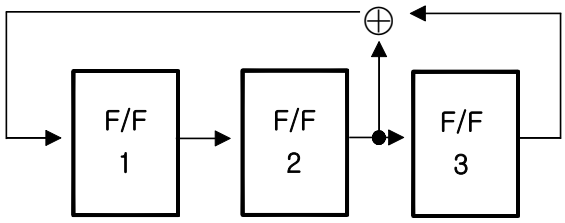
Fig. 13. Input Sets

2) 테스트 패턴 생성 및 할당

본 논문에서 생성되는 테스트 패턴은 LFSR에 의한 의사 무작위 패턴이다. 모든 PI_i 수에 맞는 LFSR을 이용하여 패턴을 생성할 필요 없이 각 집합에 해당하는 입력 수에 맞추어 따로 테스트 패턴을 생성하여 사용하면 된다. 하지만 각 $IS(N_j)$들에 대해 각각 테스트 패턴을 생성할 경우 TPG에 의한 하드웨어 오버헤드가 더욱 증가할 것이다. 따라서 이 때 분류된 집합 중 PI_i 수가 가장 많은 집합을 선정하여 이를 기준으로 의사 무작위 패턴을 생성하고, 이 패턴들을 나머지 집합의 입력들에 적절하게 할당시키면 된다. 만약 최대 입력 수를 가진 집합이 하나 이상 인 경우는 임의의 집합을 선택하여 테스트 패턴을 생성시키면 된다.

이렇게 생성, 할당된 패턴을 사용할 경우, 모든 PI_i들을 대상으로 의사 무작위

패턴을 생성하여 테스트를 수행하였을 때보다 입력들을 집합시킨 후 적은 수의 입력 수로 패턴을 생성하여 테스트를 수행하였을 때 걸리는 시간이 줄어들게 되는 이점이 있다. Fig. 14는 Fig. 13에서 찾은 집합 중 PI_i 수가 가장 많은 $IS0$에 대해 테스트 패턴을 생성하고, 나머지 각 집합들로 들어가는 입력들에 패턴을 할당하는 것을 보여주고 있다. 이 때 테스트 패턴을 할당할 때에는 $IS(N_j)$에 속하는 모든 입력에 할당하는 것이 아니라 PI_i에 대해서만 할당시킨다.



$IG0(Stage0)$	x_0	y_0	c_{in}
$IG1(Stage1)$		x_1	y_1
$IG2(Stage2)$	x_2	y_2	
$IG3(Stage3)$		x_3	y_2

Fig. 14. Test pattern assignment of the each input set

위에서 설명한 입력들을 연관 있는 것끼리 집합시키고 각 $IS(N_j)$에 대한 테스트 패턴의 생성 및 할당을 알고리듬으로 Fig. 15와 같이 정리할 수 있다.

```
begin
  Convert VHDL structural level to a graph;
  Find all nodes( N_j) with Length PI_i(N_j) = 1 from primary
  input( PI_i);
  Gather all inputs putting in node( N_j) with Length PI_i(N_j) = 1
  and then make IS(N_j);
    if input is not PI_i, then
       Name suitable name to input;
    end if
  Find IS(N_j) with the number of maximum PI_i of IS(N_j)s;
    if the number of maximum PI_i are several, then
       Select random IS(N_j) of them;
    end if
  Generate pseudo-random test pattern corresponding to
  the number of maximum PI_i;
  Assign properly test pattern to PI_i of each IS(N_j);
end
```


Fig. 15. Test pattern assignment algorithm of input set

Ⅳ. VHDL 구조적 레벨로 설계된 BIST 기법

3장에서 설명한 입력 집합에 따른 효율적인 테스트 패턴 할당 알고리듬에 의해서 VHDL 구조적 레벨로 설계된 대상 회로에 BIST 기능을 부과한 후 BIST 회로에 대해 시뮬레이션을 수행해 보기 위한 본 논문에서 제안하는 BIST의 구조 및 기능은 다음과 같다.

본 논문에서 제시하는 BIST는 정상 동작과 테스트 동작이 구분되는 테스트 방법으로, 3장에서 설명한 입력 집합에 따른 효율적인 테스트 패턴 생성 및 할당이 이루어지도록 하드웨어를 구성하였다.

1. 테스트 패턴 발생기

BIST 회로의 앞단에 있는 테스트 패턴 발생기(TPG)는 입력 집합 중 주 입력 수가 가장 많은 집합에 속한 주 입력 수와 같은 비트 수를 갖는 LFSR로 구성하였다. 이 때 회로의 입력을 통해 그 노드의 값을 조절할 수 있는 정도인 조절 용이도(controllability)와 회로의 출력을 관찰할 수 있는 정도인 관측 용이도(observability)를 높인 주사(Scan) 플립플롭을 사용하였다. 이는 효율적으로 테스트 패턴을 생성해 낼 수 있도록 하여 LFSR에 사용된 기억 소자들을 테스트 모드에서 외부로부터 쉽게 제어할 수 있도록 한다.

Fig. 16은 구현된 4 비트 TPG를 예로 든 것으로 그림을 보면, TPG는 Scan Select 신호와 Clock 신호에 의해 제어되며, Mux를 통해 제어되는 Scan Select 신호에 따라 Scan 동작과 LFSR 동작이 구분되어진다. Scan Select = '0' 일 때는 Scan In이 입력으로 들어가 주사 동작을 수행한 후 Scan 출력으로 나온다. 이렇게 나온 Scan 출력을 통해 TPG의 기억 소자들을 제어할 수 있게 된다. 그리고 Scan Select = '1' 일 때는 LFSR 기능을 수행하도록 하여 각 플립플롭에서

테스트 패턴이 생성된다. 이렇게 생성된 테스트 패턴이 다음 단에 연결되는 테스트 대상 회로의 입력에 들어가게 되는 것이다.

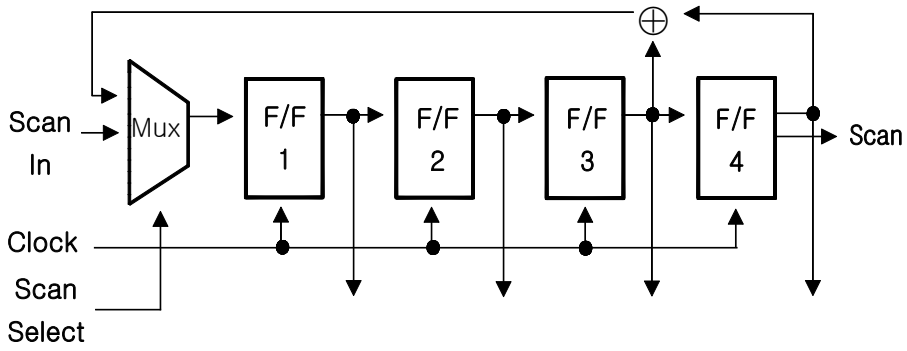


Fig. 16. 4 bit test pattern generator

2. 테스트 대상 회로

CUT는 정상 동작과 테스트 동작 시 각기 다른 개수의 입력이 들어가 논리 동작을 수행하므로, CUT 바로 앞단에 CUT 입력에 할당되는 테스트 패턴 입력과 정상 동작 시 들어가는 입력을 제어하기 위해 Mux를 두었다. 테스트 패턴은 입력 집합에 따라 줄어든 비트 수를 갖지만, 정상 동작 시 입력은 원래의 입력 비트 수를 갖기 때문에 Mux의 제어핀 Normal/Test에 따라 정상 동작과 테스트 동작을 구분하여 효율적으로 CUT 입력에 들어가도록 제어하였다. Normal/Test = '0' 이면 정상 동작이 수행되고, Normal/Test = '1'이면 테스트 동작이 수행되도록 하였다.

Fig. 17은 CUT와 Mux를 구현한 그림이다.

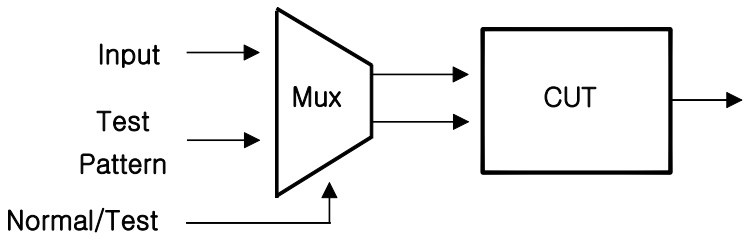


Fig. 17. Mux + CUT

3. 테스트 응답 분석기

테스트 응답 분석기(TRP)로는 LFSR과 마찬가지로 주사 출력을 위한 주사 경로를 포함하여야 하기 때문에 Mux를 포함하고 주사 기능을 갖는 MISR을 사용하였다. 그리고 무 고장일 때의 응답과 고장이 발생했을 때의 응답을 비교하기 위해 마지막 단에 비교기를 설치하여 Pass/Fail 핀만으로 고장의 유무를 알 수 있도록 하였다.

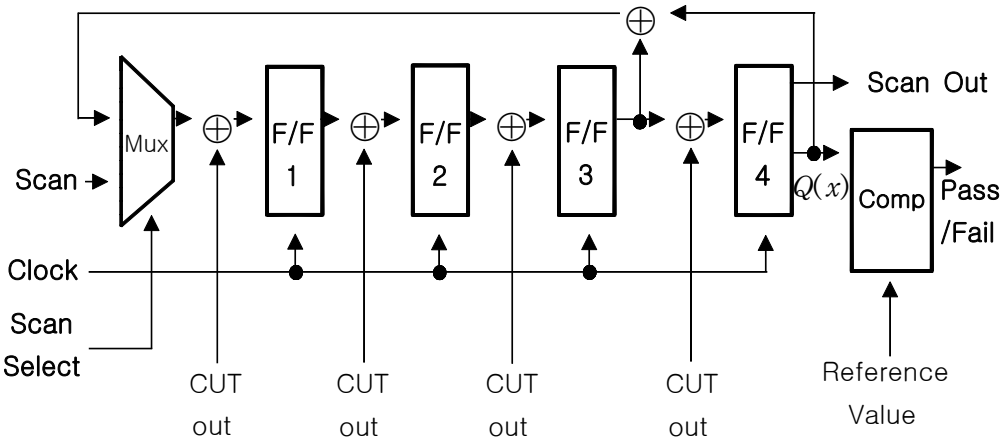


Fig. 18. 4 bit test response processor

표 3은 본 논문에서 제안한 BIST에 부가된 입출력 핀과 각 핀들의 동작을 보여주고 있다. Clock은 CUT가 클럭을 사용할 경우 그 Clock을 사용해도 된다.

Tabel 3. Operation of pins and additional I/O pins for the proposed BIST

Pin Name	Operation
Clock	Clock generation on Test
Scan In	Scan input
Scan Out	Scan output from MISR
Scan Select	Selection between Scan function and LFSR function
Normal/Test	Selection between Normal operation and Test operation
Pass/Fail	Judgment fault

Fig. 19은 Fig. 9의 대상 회로에 대해서 BIST를 실현하기 위해 합성 Tool을 이용하여 합성한 결과이다. 테스트 대상 회로의 입력 단에 테스트 패턴 발생기가 부가되고, 출력 단에 테스트 응답 분석기가 부가가 된 전체적인 BIST 회로이다.

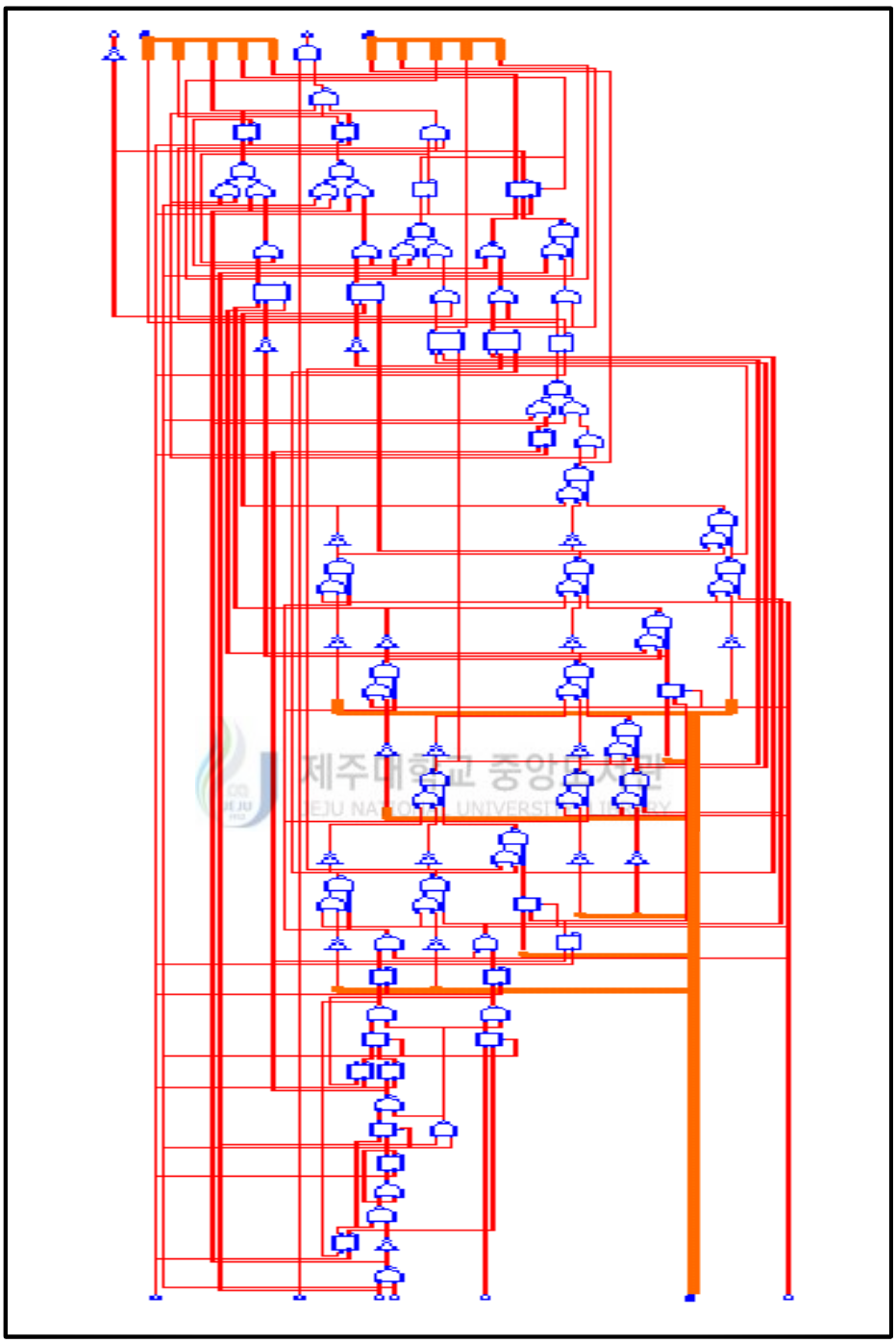


Fig. 19. Synthesis Result of the proposed BIST of Fig. 9 VHDL code

Ⅴ. 시뮬레이션 결과 및 고찰

본 논문에서 제안한 BIST 및 입력 집합에 따른 테스트 패턴 생성 및 할당을 실험하기 위해 VHDL 상위 수준의 구조적 레벨로 설계된 대상 회로를 가지고 수행하였다.

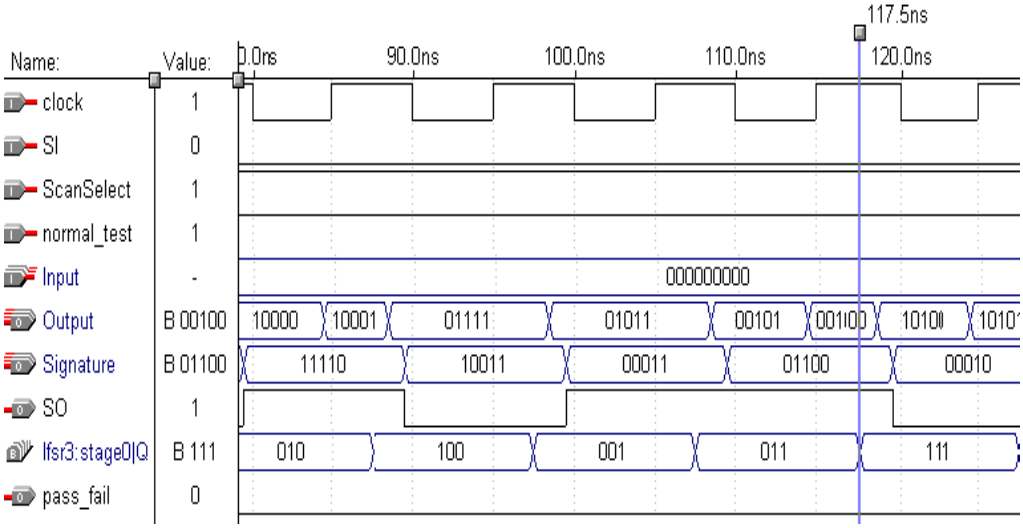
VHDL 상에서 구조적 레벨로 설계된 회로를 도표로 변환, 변환된 도표에 의해 입력 집합을 수행한 후 입력 집합에 따라 테스트 패턴을 생성, 할당하여 테스트를 수행하였다. 그리고 본 논문에서의 고장은 앞에서 가정된 VHDL 고장 모델을 사용하였다.

테스트 시간에 대한 시뮬레이션을 수행하기 위해 MAX-PLUS Ⅱ를 이용하여 시뮬레이션을 수행하였고, 하드웨어 오버헤드를 알아보기 위해 IDEC Standard Cell Library (IDEC-C631 based on LG 0.6 μm three-metal 3.3 V CMOS technology)을 이용하여 합성 Tool인 Synopsys에서 합성을 수행하였다.

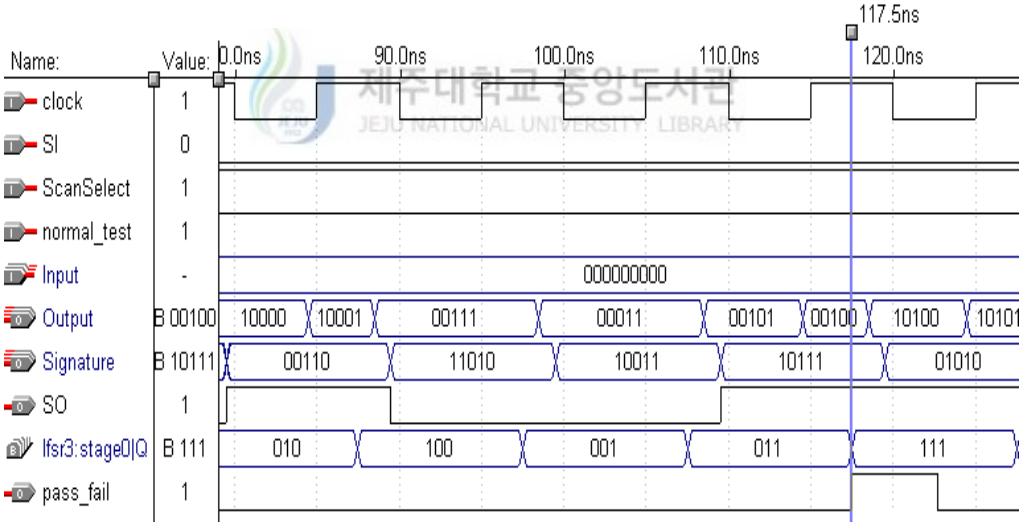
먼저 제안한 알고리듬에 의해 실현되는 BIST 회로가 제대로 동작하는지 알아보기 위해 실험하였다. 제안한 알고리듬에 의해 줄어든 테스트 패턴 수를 이용하여 테스트를 수행하기 위한 대상 회로는 Fig. 9의 회로를 사용하였고, Fig. 19와 같이 BIST가 실현된 회로에 대한 시뮬레이션 결과는 Fig. 20에 나타내었다.

테스트 동작에서는 모든 테스트 입력이 인가된 후 테스트 응답 분석기의 값을 보고 고장 유무를 알아볼 수 있는데 본 논문에서는 모든 테스트 패턴이 인가된 후 BIST 회로의 후단에 설치되는 비교기의 출력인 Pass_Fail 핀의 결과를 비교하면 된다. 모든 패턴이 인가 된 후 Pass_Fail = '0' 이면 고장이 발생하지 않음을 나타내고, Pass_Fail = '1' 이면 고장이 발생함을 알 수 있어 간단하게 고장 여부를 판단할 수 있었다.

Fig. 20의 시뮬레이션 결과를 살펴보면, Fig. 10의 내부 신호선 14에 Stuck-at 0 고장이 발생하였을 때 Pass_Fail의 논리 값이 '1'이 되는 것으로 보아 고장이 검출됨을 알 수 있었다. 따라서 본 논문에서 제안한 BIST 회로가 제대로 동작함을 또한 알 수 있었다.



(a)



(b)

Fig. 20. BIST simulation result of Fig. 9 VHDL code

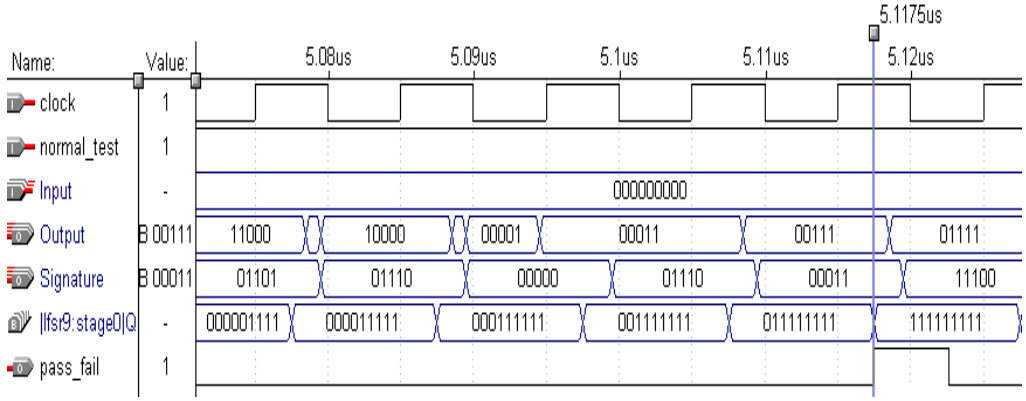
(a) Fault-free case (b) Faulty case

그리고 테스트 대상 회로를 검사하기 위해 주 입력에 대해 입력 집합을 한 후 제안한 BIST를 구현하여 테스트 시간이 입력 집합을 하기 전의 BIST 회로에 비해 테스트 시간이 감소되었는가를 실험하였다. 테스트 대상 회로는 Fig. 9의 회로이며, 표 3의 제어 핀들에 의해 BIST를 구현한 회로에 대한 시뮬레이션을 수행한 결과를 Fig. 21에 나타내었다.

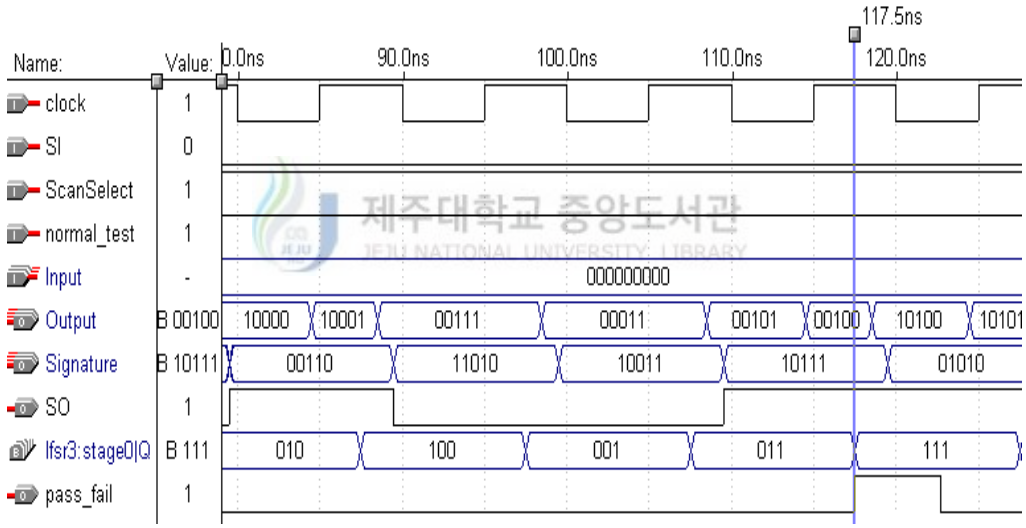
먼저 시뮬레이션을 수행하기 전 Fig. 9의 회로를 도표로 변환한 Fig. 10의 내부 신호선 14에 Stuck-at 0 고장이 발생하여 테스트 대상 회로가 고장 회로임을 가정하였다. 이렇게 한 회로에 대해 시뮬레이션을 수행한 결과 테스트 대상 회로에 대한 기존의 입력 집합을 하지 않은 BIST는 입력 비트 수가 9개이므로 $2^9-1=511$개의 테스트 패턴 후에 고장 여부를 판단할 수 있었다. 따라서 Fig. 21(a)에서처럼 $100MHz$의 클럭에 대해 LFSR을 수행한 시간 후인 $5.1175us$ 후에 테스트 결과를 볼 수 있었다. 하지만 입력 집합한 후 테스트 대상 회로에 대한 BIST의 패턴 입력 비트 수가 3개로 줄어들어 $2^3-1=7$개의 테스트 패턴 후에 고장을 판단할 수 있어 Fig. 21(b)에서 보는 것과 같이 주사 테스트와 더불어 LFSR에 의한 테스트를 수행한 후인 $0.1175us$ 후에 테스트 결과를 관찰할 수 있었다. 따라서 본 논문에서 제안한 알고리듬을 이용하여 테스트를 수행한 결과 테스트 시간이 감소함을 알 수 있었다.

VHDL 구조적 레벨로 기술된 대상 회로를 통해, 본 논문에서 제안한 방법이 테스트 시간 감소와 하드웨어 오버헤드 감소에 어떤 영향을 미치는지를 시뮬레이션을 통해 실험하였으며 그 결과를 표 4에 나타내었다.

표에서 보는 바와 같이 실험 대상은 테스트 시간과 하드웨어 오버헤드, 그리고 고장 검출률이다. 여기서 테스트 시간은 클럭 수로 정의하였고, 하드웨어 오버헤드는 대상 회로에 BIST 회로를 첨가한 후 Library를 이용하여 합성한 다음 전체 셀 면적을 평가하였다. 그리고 고장 검출률은 전체 고장 수에 대해 검출된 고장 수를 백분율로 나타내었다.



(a)



(b)

Fig. 21. Simulation result of Fig. 19 BIST circuit

(a) Before algorithm application (b) After algorithm application

Table 4. Experimental results from applying the proposed BIST implementation for various circuits

CUT	Test Time (기준 clock 수)		Test Time Rate(%)
	Pseudo-random Pattern	proposed	
4 bit RCA	511	12	2.35
4×16 Decoder	31	23	74.19
C17	31	5	16.13
8×1 Mux	2,047	134	6.55
8 bit ALU	1,048,575	2,058	0.20
16 bit CLA	4,294,967,295	272	6.33×10^{-6}

CUT	Hardware overhead (Total cell area)		H/O rate (%)	Fault coverage (%)
	Pseudo-random Pattern	proposed		
4 bit RCA	294.6	231.1	21.6	100
4×16 Decoder	363.5	342.0	5.9	98.57
C17	140.2	109.9	21.6	97.22
8×1 Mux	242.1	174.2	28.0	86.84
8 bit ALU	620.6	482.8	22.2	100
16 bit CLA	1026.8	682.3	33.6	100

표 4에서 보는 바와 같이 제안한 알고리듬에 의한 BIST는 본 논문에서 사용한 대상 회로에 대해 기존의 테스트 시간에 비해 최소 6.33×10^{-6} (%) 의 테스트 시간만으로 테스트가 수행됨을 알 수 있었다. 그리고 테스트 시간이 감소할 뿐만 아니라 하드웨어 오버헤드 면에서도 본 논문에서 제안하는 알고리듬을 적용하지 않은 기존의 BIST 합성 회로의 면적에 비해 최대 33.6 (%)가 줄어드는 좋은 결과를 보여주고 있다. 또한 고장 검출률 면에서도 가정한 고장에 대해 대체로 높은 검출률을 보이고 있다. 단, 8×1 Mux인 경우 고장 검출률이 비교적 낮은 결과를 보이고 있는데, 이는 입력의 개수에 비해 출력의 개수가 상대적으로 적어 노드의 고장에 의한 오류가 출력까지 전달될 수 있는 확률이 매우 적기 때문인 것으로 생각된다.

따라서 본 논문에서 제안한 방법은 BIST의 문제점인 테스트 시간의 증가와 하드웨어 오버헤드의 증가에 대해 효과적으로 감소시킬 수 있는 방법을 제시한다고 볼 수 있다.

Ⅵ. 결론

본 논문에서는 VHDL의 상위 수준 중 구조적 레벨로 기술된 회로에 대해 테스트 수행 시 테스트 시간을 줄이고 또한 하드웨어 오버헤드를 줄일 수 있는 BIST 실현 방법을 제안하였다.

제안한 방법은 이전에 만들어진 entity를 이용하여 구조적 레벨로 만들어진 VHDL 회로를 하드웨어적으로 분석하기 편하도록 도표로 변환한 후 각 노드로 들어가는 입력들을 노드 길이에 따라 연관 있는 입력끼리 묶어 입력들을 집합시킨다. 이 때 노드 길이가 1이 되는 입력들을 집합시켜 입력 집합들 중 최대 주 입력 수를 갖는 노드를 선정한다. 이렇게 선정된 노드의 최대 주 입력 수에 맞는 테스트 패턴을 주사 기능을 갖는 LFSR을 이용하여 의사 무작위 패턴으로 생성한다. 생성된 테스트 패턴은 우선 최대 입력 수를 갖는 노드로 선정된 노드에서 주 입력들에 할당하고, 나머지 입력 집합에 대해서 주 입력들에 대해서만 생성된 테스트 패턴을 적절히 할당시킨다. 이렇게 할당된 테스트 패턴을 이용하여 테스트 하고자 하는 대상 회로에 대해 제안한 BIST 회로를 설계, 테스트를 수행할 수 있게 하였다.

제안한 방법은 VHDL의 구조적 레벨로 설계된 여러 대상 회로에 대해 실험을 통해 테스트 시간의 감소 및 하드웨어 오버헤드의 감소, 그리고 높은 고장 검출률을 확인하였다. 실험 결과 테스트 시간은 본 논문에서 사용된 대상 회로에서 기존의 BIST 회로에 비해 최소 6.33×10^{-6} % 정도의 테스트 시간을 보였고, 기존의 긴 테스트 시간을 갖는 회로에 더욱 효과적으로 테스트 시간의 감소를 보였다. 그리고 하드웨어 오버헤드 또한 최대 33.6%의 감소를 보였다.

따라서 본 논문에서 제안한 방법은 기존의 BIST에 대한 문제점으로 있는 테스트 시간의 증가와 하드웨어 오버헤드의 증가에 대해 효율적으로 감소시킬 수 있는 방법으로 사용될 수 있을 것이다. 또한 최근 회로 설계의 흐름에 맞추어 기존의 상향식 설계 방식이 아닌 하향식 설계에 맞추어 테스트가 수행될 수 있도록 하여 회로 설계 시간 또한 줄일 수 있을 것으로 기대된다.

참고 문헌

- Abramovici, Breuer, Frideman, 1990, Digital systems testing and testable design, Computer science Press,
- A Magdolen, J. Bexakova, E. Gramatova, M. Fischerove, September 1994, REGENT-Test Pattern Generation on Register Transfer Level, Euro-DAC, pp.446-451.
- C. H. Cho, J. R. Armstrong, 1994, B-Algorithm : A Behavioral Test Generation Algorithm", IEEE International Test Conference, pp.968-979.
- David W. Knapp, 1996, Behavioral synthesis, Prentice-Hall.
- Douglas J. Smith, 1996, HDL Chip Design, Doone Publications.
- F. F Tusi, 1986, LSI/VLSI Testability Design, McGrow-Hill.
- Gabriel M. Silberman and Ilan Spillinger, 1991, RIDDLE : A Foundation for Test Generation on a High-Level Design Description, IEEE Transactions on computers, Vol.40, NO.1.
- James R. Armstrong F. Gail Gray, 1993, Structured Logic Design with VHDL, Prentice-Hall.
- J. R. Armstrong, 1983, Chip-Level Modeling and Simulation, IEEE Trans. CAD of Integrated Circuits and Systems, pp. 141-148.
- J. R. Armstrong, 1984, Chip-Level Modeling of LSI Devices, IEEE Trans. CAD of Integrated Circuits and Systems, pp. 288-297.
- K. Kim, D. S. Ha, J. G. Tront, 1988, On using Signature Registers as Pseudo random Pattern Generators in Built-In Self-testing", IEEE Trans. on CAD, vol.7, No.8, pp. 919-928.
- Manzer Masud and Maddumage Karunarate, 1994, Test Generation based on Synthesizable VHDL Description", Euro-DAC.
- McCabe, T. J., 1982, Structured Testing, COMPSAC82 Tutorial Notes, IEEE Computer Society.

- Michael J. Riezenman, 1998, Technology 1998, Test & Measurement, IEEE spectrum, pp 65-69.
- Nur A. Touba, Edward J. McClusky, 1994, Transformed Pseudo-random pattern for BIST, CRC Technical Report No.94-10.
- Nur A. Touba, 1996, Synthesis Techniques for Pseudo-random Built-In Self-Test, Stanford University.
- Parag K, Lala, 1997, Digital Circuit Testing and Testability, Academic Press.
- Parag K, Lala, 1996, Practical Digital Logic Design and Testing, Prentice-Hall.
- Solomon W. Golomb, 1982, Shift Register Sequence, Aegean Park Press, Laguna Hills, CA.
- Stephen Brown, 2000, Fundamentals of Digital Logic with VHDL design, McGraw-Hill.
- Vishwani D, Agrawal, Charles R. Kimw, and Kewal K. Saluja, 1993, A Tutorial on Built-In Self Test, Part 2 : Application", IEEE Design & Test, pp.69-77.

부 록

▷ 4bit Ripple-carry adder VHDL code

```
library IEEE;
use IEEE.Std_logic_1164.all;
entity RCA is
  port (x, y : in std_logic_vector(3 downto 0);
        cin : in std_logic;
        s : out std_logic_vector(3 downto 0);
        cout : out std_logic);
end RCA;
architecture structure of RCA is
  component fulladder
     port(a, b, c : in std_logic;
                  sum, ca : out std_logic);
  end component;
  signal c : std_logic_vector(3 downto 1);
  begin
    stage0 : fulladder port map (x(0), y(0), cin, s(0), c(1));
    stage1 : fulladder port map (x(1), y(1), c(1), s(1), c(2));
    stage2 : fulladder port map (x(2), y(2), c(2), s(2), c(3));
    stage3 : fulladder port map (x(3), y(3), c(3), s(3), cout);
  end structure;
```

▷ 4×16 Decoder VHDL code

```
library ieee;
use ieee.std_logic_1164.all;
entity 4to16decoder is
  port ( w : in std_logic_vector(3 downto 0);
         En : in std_logic;
         y : out std_logic_vector(15 downto 0));
end 4to16decoder;
architecture structure of 4to16decoder is
  component 2to4decoder
     port ( w : in std_logic_vector(1 downto 0);
```

```
            En : in std_logic;
            y : out std_logic_vector(3 downto 0)) ;
  end component;
  signal m : std_logic_vector(3 downto 0);
begin
    stage0 : 2to4decoder port map (w(2 to 3), En, m );
    stage1 : 2to4decoder port map (w(0 to 1), m(0), y(0 to 3));
    stage2 : 2to4decoder port map (w(0 to 1), m(1), y(4 to 7));
    stage3 : 2to4decoder port map (w(0 to 1), m(2), y(8 to 11));
    stage4 : 2to4decoder port map (w(0 to 1), m(3), y(12 to 15));
end structure;
```

▷ C17 VHDL code

```
library ieee;
use ieee.std_logic_1164.all;
entity c17 IS
  port ( INP : in std_logic_vector(4 downto 0);
         OUTP : out std_logic_vector(1 downto 0));
end c17;
architecture structure of c17 is
  component NAND_2
    port ( inp1, inp2 : in std_logic;
           out1 : out std_logic );
  end component;
  signal INTERP : std_logic_vector(3 downto 0);
  signal OUTPI : std_logic_vector(1 downto 0);
begin
  stage0 : NAND_2 port map (INP(0), INP(2), INTERP(0));
  stage1 : NAND_2 port map (INP(2), INP(3),INTERP(1));
  stage2 : NAND_2 port map (INP(1), INTERP(1), INTERP(2));
  stage3 : NAND_2 port map (INTERP(1), INP(4), INTERP(3));
  stage4 : NAND_2 port map (INTERP(0), INTERP(2), OUTPI(0));
  stage5 : NAND_2 port map (INTERP(2), INTERP(3), OUTPI(1));
  OUTP <= OUTPI;
end structure ;
```

▷ 8×1 Mux VHDL code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith. all;
entity mux81 is
  port ( ii : in std_logic_vector( 7 downto 0);
         a, b, en : in std_logic;
         dout : out std_logic);
end mux81;
architecture structure of mux81 is
  component mux41
    port ( I : in std_logic_vector(3 downto 0);
           a1, b1, en : in std_logic;
           Dout : out std_logic);
  end component;
  signal en1, r1, r2 : std_logic;
begin
  stage0 : mux41 port map (ii(7 downto 4), a, b, en, r1);
    en1 <= not (en);
  stage1 : mux41 port map (ii(3 downto 0), a, b, en1, r2) ;
    dout <= r1 or r2;
end structure;
```


▷ 8 bit ALU VHDL code

```
library Ieee;
use Ieee.std_logic_1164.all;
use Ieee.std_logic_arith.all;
entity ALU8 is
  port ( A : in Std_Logic_Vector ( 7 downto 0 );
         B : in Std_Logic_Vector ( 7 downto 0 );
         Cin : in Std_Logic;
         Sel : in Std_Logic_Vector ( 2 downto 0 );
         DataOut : out Std_Logic_Vector ( 7 downto 0 ));
end ALU8;
```

```
architecture structure of ALU8 is
  component Transfer
      port ( Trans : in Std_Logic_Vector ( 7 downto 0 );
             Sel: in Std_Logic_Vector ( 2 downto 0 );
             Result : out Std_Logic_Vector ( 7 downto 0 ));
  end component;
  component Transfer2
      port( Trans : in Std_Logic_Vector ( 7 downto 0 );
            Sel : in Std_Logic;
            Result : out Std_Logic_Vector ( 7 downto 0 ));
  end component;
  component Adder8
      port ( A : in STD_LOGIC_VECTOR( 7 downto 0 );
             B : in STD_LOGIC_VECTOR( 7 downto 0 );
             C : in STD_LOGIC;
             SUM : out STD_LOGIC_VECTOR ( 7 downto 0 ));
  end component;
  component adder2
      port ( A, B : in Std_Logic;
             S : out Std_Logic);
  end component;
  signal Data_A, Data_B : Std_Logic_Vector ( 7 downto 0 );
  signal C, notSel : Std_Logic;
  begin
    stage0 : Transfer port map ( B, Sel, Data_B );
             notSel <= not (Sel(2));
    stage1 : Transfer2 port map (A, notSel, Data_A);
    stage2 : adder2  port map (notSel , cin, c);
    stage3 : Adder8 Port Map ( Data_A, Data_B, c, DataOut );
end structure;
```

▷ 16 bit CLA VHDL code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity cla16 is
  port ( ain, bin : in std_logic_vector(15 downto 0);
```

```
            cin : in std_logic;
            sum : out std_logic_vector(15 downto 0);
            cout : out std_logic);
end cla16;
architecture structure of cla16 is
  component cla
      port(a, b : in std_logic_vector(3 downto 0);
            cin : in std_logic;
            sum : out std_logic_vector(3 downto 0);
            cout : out std_logic);
  end component;
  signal c : std_logic_vector(2 downto 0);
begin
 stage0 : cla port map (ain(3 downto 0), bin(3 downto 0), cin,
                        sum(3 downto 0), c(0));
 stage1 : cla port map (ain(7 downto 4), bin(7 downto 4), c(0),
                        sum(7 downto 4), c(1));
 stage2 : cla port map (ain(11 downto 8), bin(11 downto 8), c(1),
                        sum(11 downto 8), c(2));
 stage3 : cla port map (ain(15 downto 12), bin(15 downto 12), c(2),
                        sum(15 downto 12), cout);
end structure;
```

감사의 글

많은 다짐과 기대를 가지고 대학원 생활을 시작한 지 엊그제 같은데 벌써 졸업을 눈앞에 두게 되었습니다. 시작할 때는 2년이라는 대학원 시간 동안 많은 것을 다 할 수 있을 것이라고 생각했는데 아직도 해야 할 일이 많이 남은 것 같아 아쉽기도 합니다.

2년 동안 사회 생활을 한 후 대학원에 진학하게 되어 늦은 공부를 시작하는데 많은 지도를 주시느라 애쓰셨던 임재윤 교수님께 감사드립니다. 처음 연구실 문을 들어섰을 때 생각지도 못한 학생이라 조금은 당황해 하시며 저를 받아주셨던 일도 너무나도 감사드립니다. 그리고 바쁜데도 논문을 쓰는데 많은 도움을 주신 이용학 교수님과 강진식 교수님께 깊은 감사를 드립니다. 그리고 대학 때부터 저에게 학문의 길을 열어주신 문건 교수님, 김홍수 교수님, 지금은 다른 나라에 계신 양두영 교수님께도 감사드립니다.

대학원 선배이자 좋은 오빠들인 정동성 선배님, 강부식 선배님, 홍성욱 선배님 그리고 이권익 선배님께도 진심으로 감사드립니다. 그리고 연구실 선배로 부족한 저를 꾸짖으며 더 열심히 하라며 조언을 아끼지 않았던 향진 오빠, 대학원 생활을 처음 시작하는 저의 투정을 다 받아줬던 형준 오빠에게도 감사드립니다. 또 2년 동안 고락을 같이 해온 봉수 오빠에게는 박사 과정 잘 하기를, 재필 오빠, 진경 오빠, 광삼 오빠, 그리고 원률이에게는 사회 생활 잘 하기를 바랍니다. 후배이긴 하지만 대학 졸업 동기인 창윤 오빠, 많은 도움을 줘야 하는데 그러지 못해 미안한 진숙, 연구실에서 같이 떠들고 웃으며 행복하게 같이 지내왔던 영배, 현미, 재오에게도 고맙다는 말을 하고 싶습니다. 또한 학과 사무실에서 나를 보며 반가이 맞아주던 은진, 언제나 든든한 철우, 비록 한 학기를 쉬어 동기들보다 조금 늦은 출발을 하는 수미에게도 남은 대학원 생활 열심히 하라고 전하고 싶습니다.

마지막으로 항상 곁에서 든든한 버팀목이 되어주었던 미르 아빠와 사랑스럽게 나를 보며 예쁜 웃음을 보여주는 미르에게 사랑한다고 전하고 싶습니다.