소프트웨어 검사방법을 이용한 VHDL 설계에서의 테스트 패턴 생성

박 승 규, 김 종 현, 김 동 욱
광운대학교 전자재료공학과
서울시 노원구 월계동 447-1
E-mail : vlsicad@daisy.kwangwoon.ac.kr

Test Pattern Generation in VHDL Design using Software Testing Method

Seung Kyu Park , Jong Hyun Kim , Dong Wook Kim
Dept. of Electronic Materials Engineering Kwangwoon University
447-1 Wolgye-Dong Nowon-Gu Seoul, 139-701
E-mail : vlsicad@daisy.kwangwoon.ac.kr

Abstract

This paper proposes a new test generation method. Most of the test generation methods are gate-level based, but our scheme is VHDL based, especially in other word, behavioral-level based. Our test pattern generation method uses software test method. And we generate deterministic test pattern with this method. The purpose of our method is to reduce the time and effort to generate the test patterns for the end-product test of IC.

I. 서론

반도체 기술의 급속한 발전으로 반도체 칩의 고집적화가 보다 용이해 짐에 따라 고밀도 회로 설계를 통해 생산성과 기술 경쟁력의 우위를 차지하려는 산업계의 노력이 가속되고 있다. 회로의 복잡성에 기인하는 칩 고장율의 증가는 반도체 제조 시 또는 칩을 사용한 제품 개발 시 테스트에 소요되는 작업 공정의 비용과 시간의 비중을 크게 차지하게 하였다.

또한 전자 시스템 개발의 모든 과정에서 쓰여질 수 있도록 ANSI/IEEE 표준안으로 제정된 VHSIC Hardware Description Language (VHDL)등의 HDL은 고집적화 되고 복잡해진 전자시스템의 하드웨어 설계 방법이 상향식(bottom-up)방식에서 하향식(top-down)방식으로 변화함에 따라 그 사용이 급격히 확산되고 있다. 이러한 HDL에 의한 설계는 합성(synthesis)라는 과정을 지나 원하는 형태(timing, area, low power ..)로 회로가 생성된다.

HDL에서 그 기능을 테스트 할 수 있는 효율적인 패턴을 생성할 수 있다면, 그 설계가 어떤 형태로 합성이 되든 한가지 패턴으로 테스트를 할 수 있기 때문에 그 때마다 테스트 패턴을 설계하는 노력(시간적, 물질적)이 따로 필요하지 않게 된다. 또, 행위-레벨(behavioral level)이라고도 하는 HDL은 그 자체가 그 칩의 기능을 기술하고 있는 것이기 때문에 칩의 기능을 가장 쉽게 알 수 있다.

이러한 이유에서 HDL에서 테스트 패턴을 자동적으로 생성할 수 있다면 고집적화가 되고 복잡해진 칩에서 한계를 느끼고 있는 지금까지의 ATPG(Automatic Test Pattern Generation)방법보다 더 나은 방법이 될 수 있다.

이 논문에서는 HDL이 software의 프로그래밍 언어와 유사한 점이 많다는 것에 착안 software testing 방법을 참조하여 칩을 테스트하기 위한 테스트 패턴을 HDL의 high-level에서 생성하는 방법에 대해 수행한 연구를 하고자 한다.[1][6]

II. 소프트웨어 검사방법

Software testing은 프로그램의 오류를 찾는데 목적이 있다. 알파 버전, 베타 버전등을 사용하는 것은 많이 알려진 프로그램의 오류(bug)를 찾기 위한 소프트웨어 테스트 방법이다. 이러한 테스트는 프로그램 설계상의 잘못을 찾아 고치는 검증(validation) 또는 검토(verification)이라고 할 수 있다. 그러나 우리는 설계 상의 잘못을 찾아 고치는 것이 아니라 만들어진 칩이 설계대로 동작하는 칩인지 아닌지를 찾는(detect) 테스트이다. 우리는 소프트웨어 테스팅 방법을 사용하여 fault의 유무를 판정할 수 있는 테스트 패턴을 생성하는 방법을 설명한다.

소프트웨어 테스트 방법은 크게 두 가지로 분류할 수 있는데, 한 가지는 소프트웨어 내면을 알 수 없는 블랙박스로 규정하고 외부에서 그 기능, 성능 등을 시험하는 블랙 박스 시험(black box test)이고, 다른 방법은 소프트웨어 내면의 세계를 파헤치는 화이트 박스 시험(white box test)으로 나뉘어 질 수 있다.[1][2][7]

우리는 화이트 박스 시험 방법을 사용하여 HDL 설계를 분석하여 효율적인 테스트 패턴을 생성한다.

화이트 박스 시험 기법

화이트박스 시험은 시험사례들을 만들기 위해 소프트웨어 형상의 구조(structure)를 이용하는 것이다. 즉 프로그램 상에 허용되는 모든 논리적 경로(logical paths)를 파악하거나 경로들의 복잡도(complexity)를 계산하여 시험사례들을 만드는 기본 토대로 삼자는 것이다.

(1) 시험영역에 대한 이해

시험영역(test coverage)이란 시험하고자 하는 원시코드의 범위를 말하며 아래와 같은 척도(metrics)로 분류될 수 있다.

a) 문장영역(statement coverage) : 프로그램내의 각 원시코드 라인이 시험과정에서 단 한번이라도 수행되도록 시험사례들을 설계하는 것이다.

b) 물리적 경로영역(branch coverage) : 프로그램의 모든 경로(flow graph 에서 화살표)가 단 한번이라도 수행되도록 시험사례들을 설계하는 것이다.

c) 논리적 경로영역(logical path coverage) : 물리적 경로의 순서가 결과에 영향을 미친다는 가정아래 모든 논리적 경로들이 단 한번이라도 수행되도록 시험사례를 설계하는 것이다.

그림 1 은 약 100 라인의 프로그램 구조를 나타내는 예이다. 이 구조의 논리 경로(logical path) 수를 따지면 100 조 가량 된다. 이렇게 많은 논리 경로를 고려하려면 기술적인 노력과 시간적인 노력이 아주 많이 필요할 것이다.

이와 같은 이유로 논리적 경로영역에서의 test pattern 생성은 힘드므로, 이 논문에서는 물리적 경로영역에서 테스트 패턴을 생성하는 방향을 택한다.

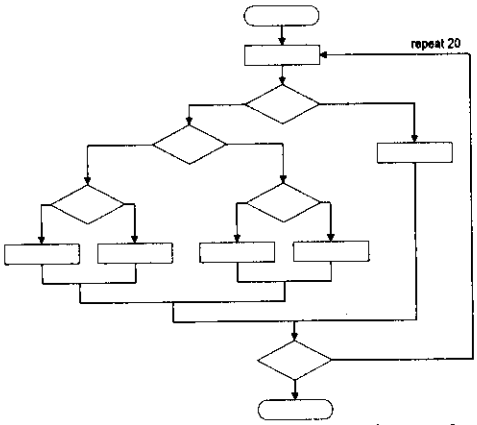

그림 1. 100 line 정도의 프로그램의 구조의 예

(2) 구조 시험

구조 시험(structural testing)은 T.McCabe[9]에 의해 제안된 가장 대표적인 화이트박스 시험 기법으로서 프로그램의 논리적 복잡도를 측정한 후 이 척도에 따라 수행시킬 기본 경로들의 집합(basis set of execution paths)을 정의한다.

논리적 복잡도를 측정하기 위해서는 프로그램 수행의 경로(execution path)를 프로그램 라인을 표시하는 노드(node)와 수행방향을 표시할 화살표(arrow)의 그래프로 나타낸 후 다음과 같은 복잡도 공식을 적용시키고 있다.

복잡도(v) = 화살표의 수(A) - 노드의 수(N) + 2 (1)

구조시험은 다음과 같은 절차를 거친다.

a) 상세설계나 원시코드를 기초로 논리흐름도(flow graph)를 그린다. 코드에 노드 번호를 부여하는 작업이 선행되어야 할 것이다.

b) 논리적 복잡도를 위 공식을 적용하여 계산한다. 이 복잡도 수치는 프로그램 구조에서 독립된 경로(independent paths)의 수를 제시한다.

III. 독립경로 탐색 알고리듬

그림 2 는 이 논문의 실제 방법에 대한 전체 흐름도이고 그림 3 은 독립경로를 찾는 알고리듬이다. 이 알고리듬에 의해서 path 들을 결정하고 이 path 들에 대해 test pattern 을 생성한다. 그림 3 에서 path 를 찾는 규칙에 대한 흐름도는 그림 4 에 나타내었다.

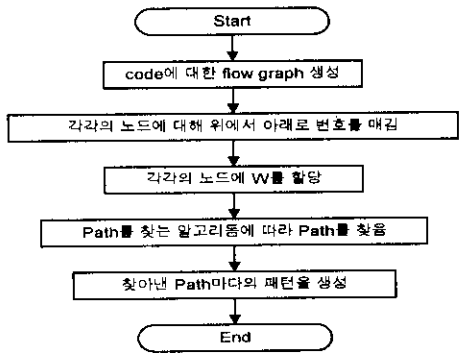

그림 2. High-level test pattern 생성 flow graph

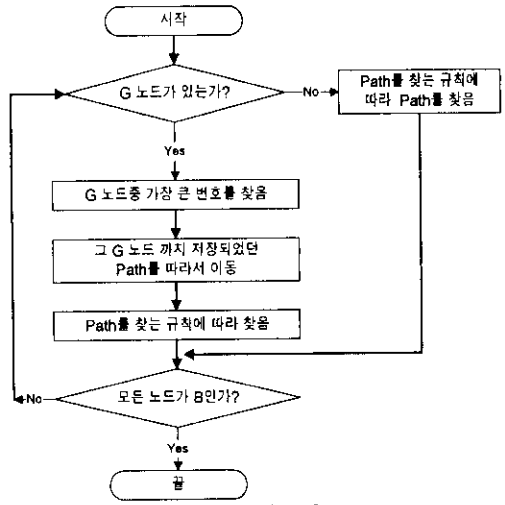

그림 3. Path 를 찾는 알고리듬

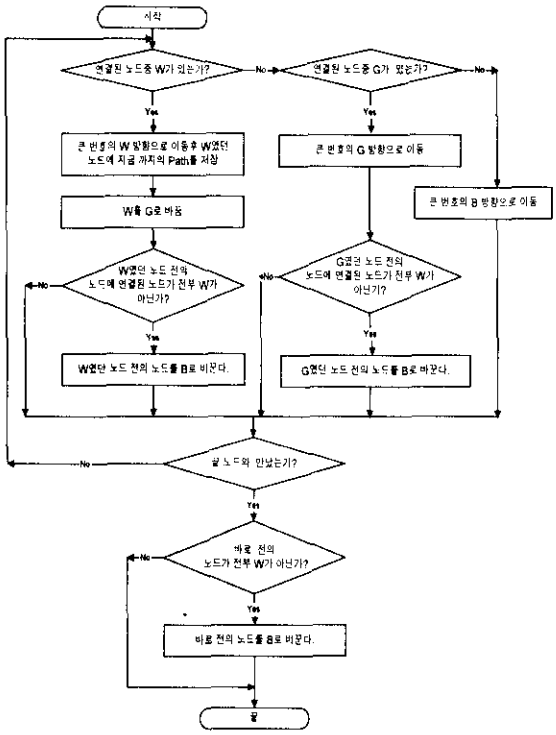

그림 4. Path 를 찾는 규칙의 flow graph

IV. 구조적 시험의 장점

독립경로를 자동적으로 찾아낼 수 있는 알고리듬 때문에 자동적인 패턴생성을 자동적으로 수행할 수 있다. 논리적 경로 영역을 모두 찾는다는 것은 거의 불가능하기 때문에 현실적인 범위에서는 최대의 시험영역이다. 또한 루프에 대해 또 다른 방법을 생각할 필요가 없이 이 구조시험에서는 루프에 대해서도 독립경로가 생길 수 있는 알고리듬이기 때문에 간단하게 pattern을 만들 수 있다. 이런 이유에서 구조적 시험을 이 논문에서 다룬다.

V. 고장검출률을 높이기 위한 방법

좀더 많은 test pattern 을 생성한다면 fault coverage 는 증가할 것이다.

1. Don't care 조건이 생기는데 패턴을 두 가지로 만든다. 한 패턴은 don't care 를 0 으로 만드는 것이고, 다른 패턴은 don't care 를 1 로 만드는 것이다. don't care 가 전부 0 일경우 stuck-at 1 을 검출하기 좋게 한 패턴이라면 전부 1 인 경우는 stuck-at 0 을 검출하기 좋은 패턴인 경우가 많다. 이것으로 인하여 패턴 수는 복잡도의 약 두 배에 해당한다.
2. Flow graph 를 만들 때 bit 가 단위라는 것을 생각하여 만든다. 비교연산자, 산술연산에 관해 어떠한 규칙을 만들어 look up table 등을 만든다. 예를 들어 2 bit 비교를 할 경우 1bit 비교연산을 연결하여 flow graph 를 만든다. (그림 5)
3. 각각의 출력에 0 과 1 이 적어도 하나는 나오도록 pattern 을 고려할 수 있게 flow graph 를 만든다. 예를 들어 Flow graph 의 출력 부분을 "out <= a; "인 경우, 화살표를 둘로 만들어서 out <= 1 인 경우와 out <= 0 인 경우 두 가지로 만든다.

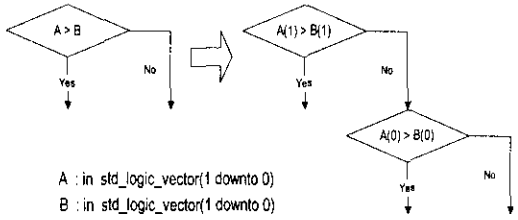

그림 5. 한 bit 씩 고려한 flow graph

VI. 시뮬레이션

대상 VHDL 코드는 Synopsys 를 사용하여 합성하여 회로를 생성하였다. 합성된 회로에는 stuck-at 고장이 삽입되며, C-언어로 구현된 테스트 생성 알고리듬으로부터 생성된 패턴을 적용하여 그 고장이 검출되는지를 검사한다.

본 논문의 알고리듬을 수행하는 방법을 보이기 위해 비교기를 예로 들겠다.

비교기에 대한 VHDL code 는 그림 6 에 나타내었고, 이 코드에 대한 flow graph 는 그림 7, 합성후의 논리회로는 그림 8 에 각각 나타내었다.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity comp2b is
port (  en   : in std_logic;
        a , b : in std_logic_vector(1 downto 0);
          o    : out std_logic_vector(1 downto 0));
end;

architecture behave of comp2b is
begin
 process
  begin
  if en = '1' then
   if a > b then
    o <= "100";
   else if a < b then
            o <= "001";
           else
            o <= "010";
           end if;
   end if;
 else
   o <= "000";
 end if;
 end process;
end behave;
```

그림 6. 비교기의 VHDL 코드

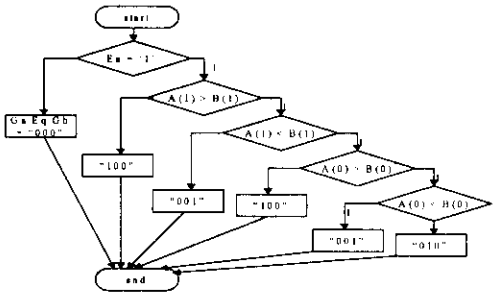

그림 7. 비교기의 flow graph

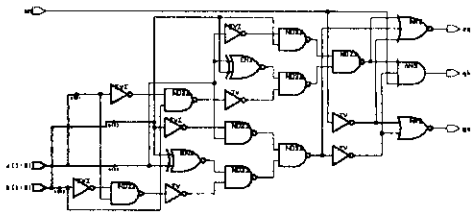

그림 8. 비교기의 논리회로

그림 7 을 사용하여 path 를 찾은 후 필요한 테스트 패턴을 생성한다. 그 pattern 은 그림 9 의 왼쪽과 같이 don't care 조건을 포함하고 있는데, 그 don't care 를 0, 1 을 규칙에 맞도록 대입시켜서 pattern 을 완성시킨다(우측의 두 패턴 세트).

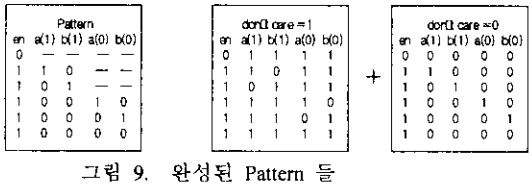

그림 9. 완성된 Pattern 들

이 패턴 세트로 53 개 노드의 106 개 stuck-at fault 에 대해 적용한 결과 101 개의 fault 를 검출할 수 있었다. 즉 95.2%의 검출율을 얻을 수 있었다. 이러한 방법으로 몇 가지 조합회로에 적용한 결과를 표 1 에 나타내었다.

회로	게이트	노드수	PI/PO	패턴	검출율
Full Adder	6	18	3/2	5	100%
1bit 비교기	8	25	3/3	5	96%
2bit 비교기(1)	18	55	5/3	12	94%
2bit 비교기(2)	21	52	5/3	12	95.2%
2bit Adder	10	31	5/3	5	95%

표 1. 시뮬레이션 결과

표 1 은 2bit 비교기를 2 개 포함되어 있는데, 코딩을 다르게 하여 하나는 1bit 비교기를 연결시킨 구조적 형태(1)로 작성하고, 다른 하나는 행위적 형태(2)로 작성하여 합성한 것이다. 패턴은 둘 다 같은 패턴이 생성되었고, 거의 비슷한 검출율을 보였다. 이러한 이유로 high-level 에서 패턴이 만들어진다면 그것이 최적의 시간적 합성이거나 최적의 면적을 갖기 위한 합성일지라도 이 패턴들이 적용될 수 있을 것이다.

이 시뮬레이션에서는 fault collapsing 을 하지 않았다. 만약 fault collapsing 을 수행한다면 검출율(fault coverage)은 달라질 것이다. 검출하지 못한 노드들을 검출하는 패턴을 찾아본 결과 검출하지 못한 노드들의 기능이 거의 비슷한 기능을 하고 있는 것을 알 수 있었다. 2bit 비교기의 경우 검출되지 않은 노드들은 'enable' 신호에 관련된 것이었다. 이러한 이유로 fault collapsing 을 할 경우 검출율은 더 높아질 것으로 예상한다.

Test 를 할 때 VHDL 코드에서 deterministic pattern 을 생성하여 적용한다면 chip 을 test 를 할 때, 현재보다 적은 시간과 노력을 들일 수 있을 것이라 사료된다.

VII. 결론

이 논문에서는 상위 수준 설계에서 test 에 필요한 deterministic pattern 을 생성하는 방법에 대해 기술하였고, 그 결과 높은 fault coverage 를 얻을 수 있었다. 이러한 방법으로 test pattern 을 만든다면 복잡한 회로의 test pattern 도 간단한 알고리듬에 의해 만들 수 있다.

이 후 연구할 내용으로 VHDL code 가 더 자동적인 규칙에 의해 flow graph 로 변화 시키는 방법과 독립경로에 대해 더 자동적으로 패턴이 생기도록 하는 연구가 되어야 할 것이다. 또, fault coverage 를 높이기 위한 연구 또한 병행되어야 할 것이다.

Reference

[1] Mark W. Johnson, "High level test generation using software metrics", M.S. thesis, Department of Electrical and Computer Engineering, Technical Report CRHC-95-06/UILU-ENG-95-2204, University of Illinois, 1995.

[2] Paul C. Jorgensen, "Software Testing", CRC Press, 1995.

[3] Hideo Fujiwara, "Logic Testing and Design for Testability", MIT Press, 1985.

[4] Abramovici, Breuer, Friedman, "Digital Systems Testing and Testable Design", IEEE Press, 1990.

[5] Douglas L.Perry, "VHDL", McGrawHill, 1993.

[6] Jong Hyun Kim, Dong Wook. Kim , "High Level Test Generation in Behavioral Level Design for Hardware Faults Detection", Proceedings of IEEK Summer Conference, 1998

[7] 최은만, "소프트웨어 공학", 희중당, 1996

[8] 김영철 외, "디지털 시스템 설계를 위한 VHDL", 홍릉과학출판사, 1998.

[9] 이주헌, ""소프트웨어 공학론", 법영사, 1993