트랜잭션 중심의 인터페이스 프로토콜 기술로부터 트랜잭션 모니터 모듈의 생성

(The Generation of Transaction Monitor Modules from a Transaction-Oriented Interface Protocol Description)

윤 창 렬 † 장 경 선 †† 조 한 진 †††

(Chang-Ryul Yun) (Kyoung-Son Jhang) (Han-Jin Cho)

요 약 SoC 설계의 검증 비용이 전체 설계 비용의 70%를 차지한다. 이런 검증을 위한 노력과 시간을 줄이기 위해서는 SoC 설계 검증 수준을 시그널 수준 또는 사이클 수준에서 트랜잭션 수준으로 높여야 할 필요성이 있으며, 또한 그렇게 하는 것이 바람직하다. 이 논문에서는 인터페이스 신호를 모니터하고, 트랜잭션의 수행을 로그 파일에 기록하고, 트랜잭션 오류를 보고하는 트랜잭션 모니터 모듈의 생성 방법에 대해 기술한다. 인터페이스 프로토콜에 대한 기술을 입력으로 모니터 모듈을 생성한다.

키워드 : 인터페이스 프로토콜 모니터링, IP 재사용

Abstract The verification portion of SoC design consumes about 70% of total design effort. To reduce the verification effort and time, it is necessary and desirable to raise the level of SoC design verifications level from the signal or cycle level to the transaction. This paper describes a generation method of transaction monitor modules that monitor interface signals, logging the transaction executions, and report transaction errors. The input of the generation method is a transaction-oriented interface protocol description.

Key words : interface protocol monitoring, IP reuse

1. 서 론

VLSI 공정 기술의 발달로 한 칩에 점점 더 많은 기능을 넣을 수 있게 됨에 따라, 그 이전에 PCB 보드 상에 있던 많은 칩들을 한 칩에 집적화시킬 수 있게 되었으며 이것을 SoC(System-on-a-Chip)라고 한다. 이와 같이, ASIC이 100만 게이트 이상을 수용할 수 있을 정도가 되면서, 완전한 프로세서들을 코어(Core)로 포함하는 설계들이 급속도로 많아지고 있다. 그러나, 미국 반도체 산업 협회인 SIA는 매년 칩의 집적도는 58% 정도 증가하지만, 설계의 생산성은 매년 21% 정도만 증가한다고 보고하고 있다. 따라서, 점점 작아지는 TTM (Time-To-Market)을 지키고, 시스템 온 칩 설계의 복잡도를 극복하기 위해서는 설계의 재사용이 매우 중요하고, 유일한 해결책일 것이다. 설계의 재사용 기술이란 이미 검증되고, 설계에 사용된 적이 있는 컴포넌트나 블록들을 가능한 많이 재 사용되도록 하는 방법이다. 그런 코어들이 지적 재산권이나 특허로 보호 받을 경우에 IP (Intellectual Property)라고 부른다.

SoC 설계에서 검증 부분에 드는 노력이 전체 설계노력의 70%를 차지한다. 설계팀은 대개 검증을 전담하는 기술자들이 있고, 일반적으로 이런 기술자가 전체 설계자들의 두 배에 이른다. 실제로 프로젝트가 끝났을 때, 전체 코드 중에 테스트 벤치에 해당하는 코드가 80%까지 이른다. 이런 사실로 볼 때 프로젝트에서 설계 검증이 임계경로에 있음을 알 수 있다[1].

검증시간을 줄이는 방법은 잘 정의된 인터페이스를 통해 설계와 검증 작업을 동시에 수행 하는 것이다. 또

· 이 논문은 2001년도 한국전자통신연구원의 지원(계약번호:2001-28)에 의해 연구되었음.

† 학생회원 : 충남대학교 컴퓨터공학과
yun@ce.cnu.ac.kr
†† 종신회원 : 충남대학교 정보통신공학부 교수
ksjhang@computer.org
††† 비 회 원 : 한국전자통신연구원 연구원
hjcho@erti.re.kr

논문접수 : 2001년 10월 12일
심사완료 : 2002년 8월 6일

다른 방법은 검증수준을 0과 1을 다루는 하위 수준에서 더 높은 수준 - 트랜잭션 또는 버스 사이클 수준으로 추상화 하는 방법이다[1]. 사이클 수준 또는 하위수준으로 트랜잭션 자체를 기술하는 것과, 트랜잭션 수준으로 테스트 벤치를 개발하는 것을 분리 함으로써, 많은 새롭고 복잡한 테스트 벤치를 빠르게 개발 할 수 있게 될 것이다. 상세한 프로토콜에 관해서는 트랜잭션 기술 [2]에 포함되어 있다. 물론 이런 방식을 통해 테스트 벤치 코드의 재사용성을 높일 수 있다.

IP 또는 코어 기반 설계 [3]에서, 보통 IP는 읽기, 쓰기 등과 같은 고정 인터페이스 프로토콜을 갖기 때문에, IP의 인터페이스 프로토콜은 타이밍 다이아그램과 같은 방식으로 기술될 수 있다. 그러나 타이밍 다이어그램 형태로 인터페이스를 기술하는 것은 설계자에게는 유용하지만 컴퓨터로 처리하기에는 어렵다. 타이밍 다이어그램을 대신하면서, 모니터 모듈 생성기, 테스트 벤치 생성기, IP 래퍼 합성기의 입력으로도 사용될 수 있는 형태, 즉 컴퓨터로 처리할 수 있는 형태로 인터페이스를 기술하는 것이 필요하다.

대부분의 사람들은 논문 [4]에서 제시된 것과 비슷한 방법을 적용하면 IP의 RTL HDL 기술에서 컴퓨터가 인식할 수 있는 형태의 인터페이스 프로토콜을 쉽게 추출할 수 있을 것이라고 생각 할 수 있다. 그래서 별도의 방법으로 인터페이스를 기술하는 방법은 필요하지 않다고 주장할 수도 있다. 그러나 RTL HDL 코드는 항상 가용한 것이 아니다. 논문 [4]에 명시된 추출 방법은 출력(입력) 포트가 유효한 데이타를 보내는(받는) 경우에 근거한다. 결론적으로 이런 추출 방법은 아래 문장처럼 조건 없이 출력포트 신호에 값이 할당된다면 잘못된 추출이 발생할 것이다. 예를 들어, 어떤 DES 코어는 다음과 같은 할당문을 갖는다. 여기서 d_out은 모든 클럭 사이클에 유효한 값을 갖는다. 그러나 이 값이 항상 올바른 값은 아니다. 적어도 출력포트에서 유효성(validness)이 항상 올바름(rightness)을 의미하지는 않는다는 것을 알 수 있다. 따라서 IP의 인터페이스는 설계자 또는 개발자에 의해 기술되는 것이 바람직하다. 타이밍 다이어그램을 포함하는 데이타 시트가 IP의 전달물의 하나이듯이, IP 또는 VC(Virtual Component)의 인터페이스 기술이 전달물 중에 하나로 포함되어야 할 것이다.

```
d_out <=  internal_signal;
```

설계에서 인터페이스를 분리하는 설계 방법론은 논문 [5]에 이미 언급되었다. ICL(International Computers Limited)에서 제안된 VHDL+[6,7]과 UCI(Univ. of California, Irvine)의 CADLAB에서 설계된 SpecC[7,8] 같은 시스템 수준의 설계 기술 언어들은 컴포넌트의 기술과 인터페이스 부분의 기술을 분리하여 기술하는 방식에 부합하기 위해 만들어진 언어이다. 이런 언어를 통해 컴포넌트가 재사용되는 것처럼 인터페이스도 재사용될 수 있다. Synopsys와 Coware 등 몇몇 CAD 회사에서 개발한 System C는 C++ 클래스 정의를 사용하여 채널과 인터페이스를 기술하는 위와 같은 방식을 채택하고 있다. 최근에 제안된 시스템 수준의 기술 언어는 C++에서의 객체지향 개념을 도입하여 모든 수준으로 설계를 기술할 수 있다고 한다. 그러나 이런 언어를 이용하여 낮은 수준의 설계를 기술한다면 많은 비효율성과 복잡성을 초래할 수 있다. 예를 들어, VHDL이나 Verilog로 기술된 RTL 설계는 효과적으로 합성되는 반면, 동일한 수준에서의 C++ 하드웨어 기술은 비효율적인 논리게이트 수준의 설계가 될 수 있다. 그러므로 모든 사람들이 이런 접근[9] 방식에 동의하는 것은 아니다. 이런 인터페이스에 대한 기술은 타이밍 다이아그램으로 사용될 수 있을 뿐만 아니라, 설계자동화 툴의 입력으로도 사용될 수 있기 때문에 인터페이스를 간단하고, 간결하고, 처리하기 쉽도록 기술하는 것이 필요하다.

인터페이스 프로토콜을 간략하게 기술하는 방법들이 인터페이스 합성의 입력으로 사용된 적이 있다. 이벤트 그래프 방법은 IP의 입출력 포트에서 발생하는 이벤트들간의 순서를 기술하는 방식을 채택했다[10]. 이때 노드(node)는 이벤트(신호 값의 변화)를 나타내고 에지(edge)는 노드 내의 이벤트 간의 정확한 지연시간이나 순서관계를 의미한다. 이벤트 그래프는 사이클 수준이 아니고 게이트 지연시간 수준(gate propagation accurate level)이다. 또 다른 방법은 제한된 VHDL 코드를 이용하여 IP의 인터페이스를 기술하는 방법이다[11]. 제한된 VHDL 코드의 기본 동작은 입력제어신호에서 이벤트 기다리기, 출력 제어신호에 값 할당하기, 입력 데이타로부터 값 읽기, 출력 데이타 신호에 값 할당하기, 고정된 시간동안 대기하기의 다섯 가지 동작이다. 그러나 이런 동작들 만으로는 분기가 있는 인터페이스 프로토콜을 기술하기 곤란하다. PFG(Protocol Flow Graph)는 언제, 어떻게 데이타를 제공하고, 서버(곱셈기, 덧셈기 등)로부터 데이타를 가져와야 하는지에 대해 기술한다[12]. PFG는 트랜잭션 형태로 인터페이스를 기술하는 것으로 간주할 수 있지만, 테스트 벤치 개발에서 필요한 트랜잭션 수준에서 프로그래밍 언어와의 인터페이스를 위한 데이타 유형의 정의가 허용되지 않는다는 제한이 있다. PIG(Protocol Interface generation)의 인

터페이스 프로토콜 기술은 정규식(regular expression)에 근거한다[13]. 이때 정규식의 심볼 또는 알파벳은 한 클럭 사이클에서 모든 포트 값의 집합을 의미한다. '*'(Kleene-closure), '|'(alternative), ';'(Sequential) 등의 연산자와 심볼의 조합으로 프로토콜을 나타낸다. 이런 방법은 트랜잭션 수준에서 프로그램 언어와의 인터페이스를 위해, 데이타 유형의 정의가 허용되고, 프로토콜의 전달 인수가 정의된 유형을 갖도록 할 수 있다. 그러나 PIG의 기술은 프로토콜당 하나의 트랜잭션만을 허용한다.

본 논문에서는 사이클 수준(cycle-accurate level)에서 IP의 포트들을 통해 수행되는 트랜잭션들을 기술하는 인터페이스 기술 언어(IPL)를 제안한다. PIG의 기술 방법을 확장하여 다중 트랜잭션, 내부 레지스터, 고정 또는 가변 버스트 전송의 기술, 조합회로 출력포트(combinational output port) 등을 허용한다. 이런 IPL을 입력으로 하여 VHDL로 기술된 트랜잭션 모니터 모듈을 생성하는 방법을 기술한다. 해당 신호들에 연결된 모니터 모듈은, 연결된 신호를 통해 수행되는 트랜잭션을 모니터 하고, 그 결과를 파일에 기록하고, 오류가 발생하면 그 정보를 기록하게 된다. 이런 모니터 모듈을 이용하여, 인터페이스 프로토콜의 오류 위치를 알아내는 시간을 줄이고, 시뮬레이션 결과를 트랜잭션 수준에서 알 수 있으므로, 시뮬레이션 결과(타이밍 다이아그램 등)를 분석하는 시간을 줄일 수 있다.

2절에서는 SDRAM 컨트롤러의 예를 통해 IPL의 구조를 설명한다. 3절에서 IPL로부터 트랜잭션 모니터 모듈의 자동생성 방법을 기술한다. 4절은 생성 결과에 대한 설명이고 마지막으로 논문을 요약하고 향후 과제를 기술한다.

2. 인터페이스 프로토콜 기술 언어(IPL)

논리 합성 툴들이 일반화 되고, 많은 코어가 RTL수준으로 기술되고 있기 때문에, IPL은 RTL 수준의 인터페이스 프로토콜 언어를 목표로 한다. IPL은 테스트 벤치 자동 생성, 트랜잭션 모니터 모듈 생성기, 인터페이스 합성기 등과 같은 인터페이스 관련 설계자동화 툴의 입력으로 사용될 수 있도록 구성되었다.

자동으로 테스트 벤치를 생성하기 위해서 IPL은 트랜잭션 수준에서 C/C++ 등 프로그래밍 언어와의 인터페이스가 필요하다[7,14]. 그러기 위해서는 데이타 유형의 정의가 가능해야 한다. IP는 여러 개의 트랜잭션이 있고, 각 트랜잭션은 트랜잭션 수행시에 전달되는 인수를 갖는다. 많은 IP나 온칩 버스 프로토콜에서 종종 사용되는 버스트 전송에 대한 기술이 가능하도록 설계되었다.

IPL에서는 3상태(tri-state), 양방향, don't care조건, 그리고 포트 값의 유효시간 등 포트와 관련된 특징을 기술 할 수 있다.

IPL에서 설계자는 리셋(reset)과 클럭에 대해 매개변수(parameter)를 이용하여 표현한다. IPL은 단일 클럭 도메인(single clock domain)으로 가정한다.

다음 그림 1은 IPL로 SDRAM 제어기가 프로세서와 연결되는 쪽의 인터페이스를 기술한 것이다. 데이타 유형의 정의는 키워드 'type'이 오고, 이어 새로운 유형과 유형의 이름 'basic'이 온다(①). 이런 문법은 PIG[13]에서와 같은 방법을 사용했다. 데이타 유형 'mode'(②)는 버스트 길이와 읽기 지연시간 등을 초기화 시키는 'initiate' 트랜잭션의 전달 인수의 데이타 유형이다.

```
type basic bit[31:0];  /* ① */
type Bsize bit[2:0];
type twobit bit[1:0];
type SADD bit[19:0];
type mode {              /* ② */
        Bsize N;  /* Burst length */
        Bsize CAS;  /* latency */
}
protocol SDRAMctr {
master bit data_addr_n;
master bit we_rn;
master bit rst;
maslave basic AD;                /* ⑥ */
parameter clock clkp double;       /* ③ */
parameter reset rst asynchronous positive;
register Bsize n(7), cas(3);  /* ④ */
type block {
        SADD Addr;          /* Address */
        basic Data[0:n];   /* DATA */
}
term int_0() { 0, 1, 0, ["--10----------"] > }  /* ⑤ */

term int_1(Bsize a, twobit b) {1, 1, 0, ["--00----------
"&a&"-"&b&"-"] >  }
term Write_0(SADD a) { 0, 1, 0, ["--00------"&a&"--"]> }
term Write_1(basic b){ 1, 1, 0, b>}
term Read_0(SADD c){ 0, 0, 0, ["--00-------"&c&"--"]> }
term Read_1(){ 1, 0, 0, -}
term Read_2(basic d){ 1, 0, 0, d<}
term ready() { 1, 0, 0, -}
transaction initiate(mode mdset) /* ⑦ */
  : n <= mdset.N, cas <= mdset.CAS; /* ⑧ */
{ ready()*, int_0(), int_1(mdset.n, mdset.cas) }
transaction Read(block Imp) /* ⑨ */
{ ready()*, Read_0(Imp.Addr),Read_1()^(9+cas),
Read_2(Imp.Data[0])...Read_2(Imp.Data[n-1]) } /* ⑪ */
transaction Write(block Imp)                    /* ⑩ */
{ ready()*, Write_0(Imp.Addr), Write_1(Imp.Data[0])...
        Write_1(Imp.Data[n-1]) }
SDRAMctr = initiate | Read | Write; /* ⑩ */
}
```

그림 1 IPL로 기술한 SDRAM 제어기의 인터페이스 프로토콜

PIG에서는 프로토콜당 하나의 트랜잭션만을 가정했기 때문에, 프로토콜 이름은 전달되는 데이타 다음에 오게 된다. 그러나 IPL에서는 다중 트랜잭션이 기술 될 수 있기에 트랜잭션의 이름은 IPL의 마지막 부분에 열거된다(⑩). 또한 각 트랜잭션은 전달되는 데이타 또는 주소의 전달 인자를 갖는다(⑨).

클럭에 대한 매개변수는 키워드 'parameter clock'이 오고, 클럭의 이름, 클럭 동작 특성이 온다(③). 리셋의 경우도 비슷하게, 키워드 'parameter reset'이 오고, 리셋 신호의 이름, 리셋 동작 특성(synchronous, asynchronous, positive, negative)의 순서로 기술한다.

내부 레지스터는 IP의 내부 변수를 변경하는 이전의 트랜잭션에 의해 영향받는 트랜잭션을 기술하기 위해 제안했다. 예로 SDRAM 제어기의 initiate 트랜잭션(⑦)은 버스트 길이와 읽기 지연시간 등을 변경한다. 이것은 읽기 쓰기의 트랜잭션의 동작에 영향을 준다. 레지스터의 초기값은 내부 레지스터의 선언부에서 괄호 안에 기술 한다(④). 트랜잭션에 의한 내부 레지스터의 변화는 트랜잭션 기술 전에 나타나는 할당문을 사용하여 표현한다(⑧).

텀의 기술은 PIG에서와 같은 의미를 갖는다. 텀은 한 클럭 사이클 내에 각 포트 값의 집합이다. 전달 인수와 상수 값의 조합으로 포트의 값을 구성하기 위해 결합 연산자를 추가 하였다(⑤).

마스터 또는 슬래이브는 양방향 포트의 값을 구동 할 수 있다. 이 경우 IPL에서 텀의 표현에서 값 뒤에 오는 '>'('<')는 해당 클럭 사이클에서 마스터(슬레이브)가 해당 포트를 구동함을 의미한다.

반복적 구조를 표현하는 연산자 '*', '+' 이외에도 두 가지 연산자를 추가하였다. '^' 연산자는 간단한 수식 다음에 오고 상수 또는 레지스터를 갖는다. 이것은 주어진 클럭 사이클 수 동안 해당 텀의 반복을 의미한다(⑪). '#' 연산자는 간단한 수식 뒤에 오고, 주어진 수만큼의 클럭 사이클 안에 연관된 텀 조합의 반복이 끝나야 함을 의미한다. 연산자 '#'는 제한된 수의 클럭 사이클 동안 타겟에서 응답을 기다리는 PCI initiator와 같이 어떤 시간 제약을 나타내는데 사용된다.

고정된 길이 또는 가변 길이의 버스트 전송은 텀 전달인수의 인덱스를 제외하고 같은 구조를 갖는다면, '…' 연산자로 표현할 수 있고, 'Read_2' 텀은 0부터 n-1번 반복함을 의미한다. 이때 'n'은 내부 레지스터 값이다.

조합회로의 출력을 갖는 포트(combinational output ports)의 동작을 표현하기 위해 텀 선언부에 마스터 포트와 슬래이브 포트간의 부분순서관계(partial order relationship)을 기술하도록 추가하였다. 예를 들어 다음 기술은 슬래이브 포트 'done'이 조합회로의 출력을 갖는 포트이고, 마스터 포트 'start'의 값이 '0'이 될 때, 'done' 신호는 '1'이 됨을 나타낸다. 마스터 포트 'start'는 순차적(sequential) 또는 레지스터드(registered) 출력이고, 마스터는 해당 클럭 사이클의 시작에서부터 값을 구동해야 한다.

```
master bit start;
slave bit done combinational;
...
term a() { 1, 0 } { start -> done }
term b() { 0, 1 } { start -> done }
```

3. 트랜잭션 모니터 모듈의 생성

트랜잭션 모니터 모듈은 그림 2와 같이 두 개의 연결된 IP 사이의 신호들을 통해 실행되는 트랜잭션을 기록하는 역할을 한다. 모니터 모듈은 오류가 발생한다면 오류를 기록한다. 모니터 모듈은 인터페이스 프로토콜 오류의 위치를 찾는 시간을 줄일 수 있고, 트랜잭션 수준에서 시뮬레이션 결과를 보여줌으로써, 타이밍 다이아그램의 파형을 보는 것과 같은 시뮬레이션 결과에서 트랜잭션을 찾는 시간을 줄여준다. 각 트랜잭션에 대한 기록은 트랜잭션의 시작과 종료시간, 실행된 트랜잭션의 이름, 전달된 데이타 또는 주소, 오류와 그 원인 등이 파일에 저장된다.

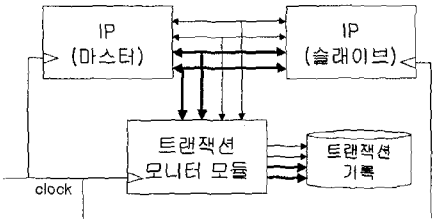


그림 2 두 IP 사이에 연결된 트랜잭션 모니터 모듈

그림 3은 트랜잭션 모니터 모듈의 생성 흐름이다. 추상적 FSM의 생성은 derivative construction[13,15] 알고리즘을 이용했다. IPL은 마스터쪽의 기술도 아니고 슬래이브쪽의 기술도 아니다. IPL은 마스터를 위한 FSM, 슬래이브를 위한 FSM, 모니터 모듈을 위한 FSM으로 변환 될 수 있기 때문에, 생성된 FSM은 '추

상적(abstract)'으로 불린다. 생성흐름의 다음 단계는 추가된 반복 연산자 '^', '#', '…'를 처리하는 후처리 단계이다. 마지막으로 IP들간의 특정 신호들에 연결되어서, 연결된 신호들을 통해 수행되는 트랜잭션을 모니터 할 있는 모듈을 만드는 코드생성 단계이다.

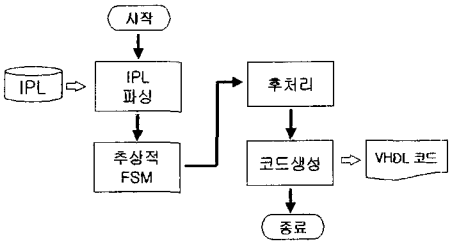


그림 3 트랜잭션 모니터 모듈 생성 흐름

첫째 IPL 파싱 단계에서는 IPL을 입력으로 받아 정의한 IPL의 문법에 맞는 기술인지 파악하고, 정해진 자료 구조에 해당 데이타를 삽입한다. IPL을 파싱하여 얻은 인터페이스의 각 트랜잭션은 내부적으로 정규식 형태의 트리와 추가된 데이타구조를 통해 표현된다. 다른 트랜잭션에 속한 정규식의 트리를 이용하여 하나의 추상적인 FSM을 생성한다. 트랜잭션을 시작하기 전, 대기상태를 하나의 출발 상태로 하여 하나의 FMS으로 만들었다. 이 생성된 FSM의 상태는 다음에 오는 텀의 집합이 되고, 전이의 조건은 텀이 된다. FSM을 생성하는 알고리즘은 다음과 같이 요약된다.

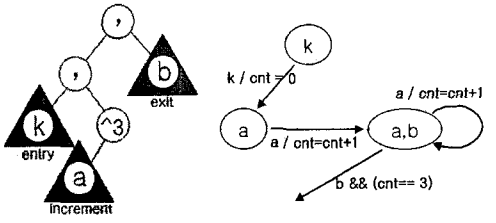
1. 처음 오는 텀으로 초기상태 's0'을 구성한다.
2. 's0'의 각 텀 't'에 대해서
3.
   A. 텀 't' 다음에 오는 텀의 집합 's1' 생성
   B. 만약 집합 s1이 존재하지 않는다면, 상태 's1'을 만들고, 's0'로 부터 's1'으로 전이 't'를 만든다.
   C. 만약 's1'이 존재한다면, 's0'로 부터 's1'으로 전이 't'를 만든다.

   더 이상 새로운 상태가 만들어지지 않을 때까지, 단계 2를 반복하여 새로운 상태를 추가한다.

모니터 모듈 생성시, 각 상태에 에러 발생 경우를 고려한 'else' 전이를 추가하였다.

추상적 FSM 생성과정에서 직접 반복연산자를 다루는 것이 쉽지 않다. 따라서 추상적 FSM 생성단계에서 반복연산자 '#'는 '*' 연산자와 동일하게 취급하고, '^'와 '…' 연산자는 '+' 연산자와 동일하게 취급하였다. 그리고 후처리 단계를 통해 각 연산자와 관련 있는 전이에 특별한 표시를 한다. 예로, 그림 4(a)는 'k', 'a', 'b'가 텀인 정규식 'k, a^3, b'의 트리이다. 그림 4(b)는 정규식의 올바른 FSM이다. 정규식은 'k'의 인식한 후에 세번의 'a' 텀이 오고, 'b'텀이 옴을 의미한다. 그래서 'k'가 인식되었을 때 카운터의 초기화가 필요하다. 그리고 매번 'a'가 인식되었을 때, 카운터를 증가시켜야 하고, 'b'가 인식되고 카운터가 3과 같을 때, 반복 동작 'a^3'가 올바르게 마치는 것이다. 이 예에서 알 수 있듯이, 'entry', 'increment', 'exit'의 세 타입의 표시가 필요하다. 그림 4에서 텀 'k'는 'entry' 전이가 되고, 'a'는 'increment' 전이, 'b'는 'exit' 전이가 된다.

반복 연산자 '^'의 전이 타입을 찾기 위해서 연산자 '^' 텀의 세 집합을 계산해야 한다. 첫째로 연산자 '^'를 루트로 하는 정규식의 서브 트리에서 가능한 모든 스트링의 첫번째 심볼(텀) Si(^)의 집합이다. 예로 정규식 '(a | b)^3'에서 'aaa', 'aab', 'aba', 'baa' 등이 올 수 있기 때문에 Si(^3)는 {a, b}가 될 수 있다. 두번째로 집합 Si(^) 전에 올 수 있는 모든 가능한 문자열의 마지막 심볼의 집합 Sp(^)이다. 예로, '(x,y | z), a^3, b'서 Sp(^3)는 {y, z}가 된다. 마지막으로 Sn(^)는 '^'노드를 루트로 하는 정규식 서브 트리의 바로 다음에 올 수 있는 가능한 문자열의 첫 심볼의 집합이다.



(a) (b)

그림 4 정규식 'k, a^3, b'의 올바른 트리 구성과 추상적 FSM

텀 'k'로 표시된 전이는 목표상태가 '^'를 루트로 하는 정규식 서브 트리에 포함되어 있는 스트링을 이끌어 내면 'entry' 전이가 된다. 리셋 전이는 FSM의 초기상태를 구성하는 텀의 집합이 부분집합으로 Si(^)를 포함하면 'entry' 전이가 될 수 있다.

$$S \rightarrow^{k} S_i(^) \cup \{k\}, \text{ where } k \in S_p(^)$$

'k'로 이름 붙여진 전이가 다음과 같은 조건을 만족한다면, 'increment' 전이가 될 수 있다.

$$S_1 \rightarrow^{k} S_2, \text{ where } k \in S_i(\hat{}\,)$$

텀 'k'로 이름 붙여진 전이가 다음과 같은 조건을 만족하면 'exit' 전이가 된다.

$$S_1 \rightarrow^{k} S_2, \text{ where } k \in S_n(\hat{}\,)$$

derivative construction 알고리즘은 반드시 종료를 보장하기 때문에 심볼 또는 텀의 부분집합이 2N 만큼 존재한다. 이때 N은 정규식의 심볼 또는 텀의 수이다. 비록 최악의 복잡도는 지수함수이지만, 부분집합의 구성면에서 관련 있는 부분집합만 고려하면, 일반적으로 최소 상태 다이아그램(minimal state diagram)이다 [16].

노드 'op'로부터 시작하여 $S_i(op)$의 집합을 찾는 동작은, 트리의 'op' 노드의 특성에 따라 상위 또는 하위 트리를 탐색하고, 마지막으로 찾은 결과를 합하여 반환한다. 다음의 개략적인 재귀 과정으로 알고리즘의 기본 동작을 요약했다. '|'는 'OR' 또는 선택적임을 의미한다.

```
Si(op) =
    ① op =  term → Si(op) = op
    ② op =  ',' → Si(op) = Si(op→left)
    ③ op =  '|' → Si(op) = Si(op→left) U Si(op→right)
    ④ op =  '+' / '*' → Si(op) = Si(op→left)
    ⑤ op =  '^' | '#' | '...'  → Si(op) = Si(op→left)
```

탐색의 영역이 'op'를 루트로 하는 정규식의 서브트리로 제한되고, 탐색의 동작 범위가 중첩되지 않기 때문에 S_i('op')의 계산 시간은 O(N)이내 이다. 비슷한 재귀적 방식이 $S_p(\hat{}\,)$와 $S_n(\hat{}\,)$를 찾을 때에도 사용된다. 그러나 이런 집합을 찾기 위한 영역은 전체 정규식의 트리에 제한된다. 따라서, S_p('op')와 S_n('op')의 계산 시간도 역시 O(N)이다. 어느 한 연산자의 $S_i()$, $S_p()$, $S_n()$의 세 집합은 공통된 텀을 갖지 않는다. 따라서 반복 연산자를 위한 후처리 과정은 O(N)이 된다.

다음 단계는 코드 생성 단계이다. 그림 5는 그림 1의 SDRAM 컨트롤러의 인터페이스의 기술을 이용하여 생성된 트랜잭션 모니터 모듈의 일부이다. 모니터 모듈은 하나의 프로세스 문으로 구성된다. 내부 레지스터는 프로세서의 선언 부분에서 초기화 되고(⑤), 그 값은 'init' 트랜잭션의 한 상태 S2의 전이에서 변수 값 할당에 의해 변화 된다. 클럭 또는 리셋과 연관된 매개변수는 에 표시된 것처럼 구현된다. 그림 1의 'Read' 트랜잭션에서 '^(9+cas)' 연산자는 ⑦에서 'entry' 전이이고, ⑧에서 'increment' 전이가 되고, ⑨에서 'exit' 전이가 된다.

```
use STD.TEXTIO.all;
    ...
entity SDRAMctr_monitor is  /*  protocol name */
 generic ( Filename : string := "/user1/yun/SDRAM/vhdl/
          func_sim/SDRAMctr_monitor.log");
    port (  clkp : in  std_logic;  /* ② */
            AD  : in  std_logic_vector( 31 downto 0 );
             ...
            data_addr_n : in  std_logic );
end SDRAMctr_monitor;
architecture monitor of SDRAMctr_monitor is
  type StateType is (S0, S2, S3, S5, S6, S7, S9, S10);
                    /* ③ state name */
            ...
  FILE outputfile: TEXT open APPEND_MODE is Filename;
begin
monit:      process(clkp, rst)   /*  asynchronous reset */
  variable count_clk : integer := 0 ;
  variable reg_cas : integer := 3; /* initialization for
                                    internal register cas */
  variable reg_n : integer := 7;  /* for internal register n */
  variable cnt_11 : integer range 0 to 1000 := 0;
                                  /* for special operator */
            . . .
  begin
  if ( rst = '1' ) then   /* positive asynchronous reset */
      CurrentState <= S0;
      cnt_11 := 0;
            . . .
  elsif clkp'event then   /* ④ double edge clocking */
      count_clk := count_clk + 1;
            . . .
  case CurrentState is
  when S0 =>
  if ( we_rn = '0' and data_addr_n = '1' ) then
                  CurrentState <= S0;
                  cnt_11 := 0;        /* ⑦ entry */
                        . . .
  elsif ( AD(29 downto 28) = "00"  and we_rn = '0'  and
        data_addr_n = '0'  ) then
                  CurrentState <= S5;
                  cnt_11 := 0;        /*  entry */
            . . .
    else
                  CurrentState <= S0;
            . . .
    end if;
  when S2 =>
    if ( AD(29 downto 28) = "00" and we_rn = '1' and
        data_addr_n = '1'  ) then
                  CurrentState <= S3;
                  reg_n := BIN2INT(e_reg_n);
                              /*  internal registers */
                  reg_cas := BIN2INT(e_reg_cas);
                          /* assign internal register */
            . . .
    else
                  CurrentState <= S0;
            . . .
    end if;
            . . .
  when S6 =>
    if (we_rn = '0'  and data_addr_n = '1'  and
       cnt_11 < reg_cas + 9 ) then
                  cnt_11 := cnt_11 + 1; /* ⑥ increment */
                  CurrentState <= S6;
```

```
    elsif (and we_n = '0'  and data_addr_n = '1' and
          cnt_15 < reg_n and cnt_11 = reg_cas + 9 ) then
                cnt_15 := cnt_15 + 1;      /* ⑨ exit */
                CurrentState <= S7;
          . . .
    end if;
          . . .
  when others =>
                CurrentState <= S0;
  end case;
 end if;
 end process;
end monitor;
```

그림 5 VHDL로 기술된 SDRAM 제어기의
트랜잭션 모니터 모듈의 일부

4. 실험 결과

전체 프로그램의 구현은 약 4000라인의 C 코드로 Sun Ultra 10에서 수행되었다. 모니터 모듈의 정확성을 검증하는 데에는 ModelSim 시뮬레이터를 사용하였다. 예로, SDRAM 컨트롤러 인터페이스가 그림 6과 같은 환경에서 실험되었다. 그림 6에서 SYSTEM이 SDRAM 제어기를 구동하는 마스터 모듈이고 SDRAM 제어기가 슬레이브 모듈이 된다. IP 모니터 모듈은 이 두 모듈 사이에 연결되어서 두 IP 사이에 발생하는 데이타 전달과 동작을 트랜잭션 수준으로 모니터링 하고, 그 결과를 기록한다. 그림 7(a)는 모니터링 결과로 생성된 파일(로그)의 일부분이다. 그림 7(a)에서 보듯이, 87번째(696ns) 사이클에서 'initiate' 트랜잭션이 수행되어 n(버스트 동작 길이), CAS(읽기 지연시간)를 초기화 했고, Write' 트랜잭션이 150번째 사이클에서 주소 80000번지를 할당하면서 시작되었고, 158번째 사이클에서 'Write' 트랜잭션이 종료되었다. 'Read' 트랜잭션은 189 사이클에서 주소 00000번지를 할당하면서 시작되었고, 209 사이클에서 'Read' 트랜잭션이 종료되었다. 271번째 사이클에서 시작된 'Read' 트랜잭션은 284번째 사이클에서 트랜잭션 프로토콜 오류가 발생 했음을 알 수 있고, 원인도 기술되어 있다. 그림 7(b)는 'Write' 트랜잭션의 타이밍 다이아그램이다. 트랜잭션 모니터링 결과 파일과 타이밍 다이아그램의 결과가 일치함을 알 수 있다. waveform 형태에서 위와 같은 정보를 얻고, 수행된 트랜잭션을 찾는 작업은 쉽지 않다. 그러나 모니터 모듈을 이용하면, 주소는 몇 비트를 사용하는지, 데이타의 전달은 언제 이루어지는지에 대한 정보 등의 시뮬레이션 결과를 쉽게 알 수 있다.

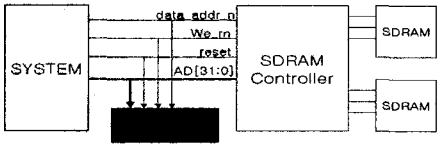
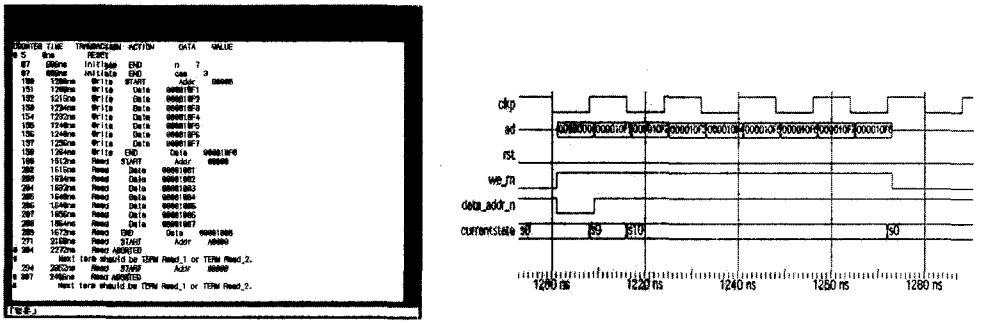


그림 6 SDRAM 컨트롤러 모니터의 실험 환경

복잡한 프로토콜을 갖는 IP들의 인터페이스 부분을 이해하는 것은 쉽지 않다. 그러나 이 모니터링 모듈을 이용하면, 쉽게 IP의 인터페이스 프로토콜을 이해하고, 테스트 벤치 생성에도 이용될 수 있으며, 모니터 모듈을 연결하여 실험 하면 프로토콜 상의 잘못된 원인을 쉽게 찾을 수 있다. 즉, 테스트 시간을 단축하게 되고 설계의 시간을 줄일 수 있다. 또한 IP를 시스템 버스에 연결할 때에, IPL로 시스템 버스 프로토콜을 기술 후, 모니터 모듈과 함께 시뮬레이션을 하면, IP가 시스템 버스 프로토콜을 제공하는지도 쉽게 알 수 있다.

다음 표는 실험한 예제 IP들의 인터페이스 기술 언어에서 트랜잭션, 팀, 입출력 포트의 수와 IPL의 줄 수 그리고 생성된 VHDL 모니터 모듈의 줄 수가 나타나 있다.



(a) (b)

그림 7 SDRAM 컨트롤러의 모니터링 결과와 타이밍 다이아 그램

표 1 IPL 기술의 예

프로토콜 이름	트랜잭션 수	텀 수	입출력 포트 수	IPL 줄 수	모니터 모듈 줄 수
SDRAM Controller	3	8	4	56	422
PCI Target Module	4	21	11	76	654
DES(64 bit)	2	7	7	41	372
DES(8 bit)	3	10	7	57	578
DES interface (to PIBUS)	3	8	13	63	729

이렇게 생성된 모니터 모듈은 연관된 인터페이스 신호에 연결되어서 올바르게 동작함을 검증했다.

5. 요약과 향후 과제

이 논문은 트랜잭션 중심으로 IP간 인터페이스 동작을 모니터링 하는 모니터 모듈 자동 생성방법을 기술하였다. 이 모니터 모듈의 입력은 트랜잭션 중심으로 인터페이스 프로토콜을 기술한 언어를 사용하였다. 인터페이스 모니터링의 예로 SDRAM 컨트롤러, PCI Target, DES 코어들의 인터페이스를 IPL로 기술하고, 이 IPL로부터 모니터 모듈을 생성하였다. 생성된 모니터 모듈이 연관된 인터페이스와 연결되어서 올바르게 동작함을 검증하였다.

시뮬레이션 과정에서 눈으로 Waveform 형태의 결과물을 보는 것보다, 모니터 모듈을 이용하여 트랜잭션 수준의 로그 결과를 통해 관찰하는 방법이, 결과 분석을 용이하게 할 것이다. 이런 모니터링 모듈을 이용하면 인터페이스를 통해 이루어 지는 동작에 대한 검증 시간을 줄일 수 있다. 앞으로 사용자의 테스트 시나리오에 맞는 테스트 벤치 자동 생성, 인터페이스 자동 합성 등의 자동 생성 툴의 개발을 계속할 계획이다.

참 고 문 헌

[1] J. Bergeron, "Writing Testbenches," Kluwer Academic Publishers, 2000.

[2] D.S. Brahme, et. al, "Transaction-Based Verification Methodology," Cadence Berkeley Labs, Technical Report #CDNL-TR-2000-0825, Aug. 2000.

[3] J.A. Rowson, A. L. Sangiovanni-Vincentelli, "Interface-Based Design," Proc. Of DAC'97, June 1997, pp.178-183.

[4] J. Smith and G. De Micheli, "Automated composition of hardware components," Proc. Of DAC 98, 1998.

[5] F.S. Eory, "A Core-Based System-to-Silicon Design Methodology," IEEE Design & Test of Computers, Vol. 14, No. 4, October/December 1997, pp.36-41.

[6] "VHDL+ LRM extentions to VHDL for System Specification," Internaltional Computers Limited, Mar. 1999.

[7] A. Gerstlauer, S. Zhao, D. D. Gajski, "VHDL+/SpecC Comparisons: A Case Study," Technical Report ICS-98-23, May 19, 1998, Dept. of Information and Computer Science, Univ. of California, Irvine.

[8] D. D. Gajski, et. al "SpecC: Specification Language and Methodology," Kluwer Academic Publishers, Mar. 2000.

[9] "System-on-Chip specification and modeling using C++: challenges and opportunities," Roundtable of IEEE Design & Test of Computers, Vol.18, No. 3, May-June 2001, pp.115-123.

[10] G. Borriello, and R.H. Katz, "Synthesis and optimization of interface transducer logic," Proc. ICCAD '87, pp.274-277.

[11] S. Narayan and D. D. Gajski, "Interfacing incompatible protocols using interface process generation," in Proc. of DAC, 1995, pp.468-473.

[12] J. Madsen and B. Hald, "An approach to interface synthesis," in Proc. of ISSS, 1995, pp.16-21.

[13] R. Passerone, J. A. Rowson, A. Sangiovanni-Vincentelli, "Automatic synthesis of interfaces between incompatible protocols," in Proc. of DAC '98, 1998 pp.8-13.

[14] "Funtional Specification for System C 2.0," version 2.0-M, Jan. 2001, available on the web site www.systemc.org.

[15] J. A. Brzozowski, "Derivatives of regular expressions," Journal of the Association for Computing Machinery, vol. 11, pp. 481-494, Oct. 1964.

[16] R. Passerone, "Automatic synthesis of interfaces between incompatible protocols," M.S. Thesis, University of California at Berkeley, 1997.

윤 창 렬

2000년 한남대학교 컴퓨터공학과 졸업(학사). 2002년 한남대학교 컴퓨터공학 석사 졸업(석사). 2002년 ~ 현재 충남대학교 컴퓨터공학과 박사 과정. 관심분야는 VLSI 설계 자동화, 컴퓨터 구조

장 경 선

1986년 서울대학교 전자계산기공학과 졸업(학사). 1988년 서울대학교 대학원 컴퓨터공학과 졸업(석사). 1995년 서울대학교 대학원 컴퓨터공학과 졸업(박사). 1996년~현재 충남대학교 정보통신공학부 조교수. 관심분야는 VLSI 설계자동화, 컴퓨터 구조

조 한 진

1982년 한양대학교 전자공학과 학사. 1987년 New Jersey Institute of Technology 전기공학과 석사. 1992년 University of Florida 전기공학과 박사. 1992년~현재 한국전자통신연구원 시스템 IC 설계팀 팀장. 관심분야는 SOC 설계 방법론, 무선통신, 멀티미디어 설계