64비트 RISC 마이크로프로세서의 기능 검증에 관한 연구

김연선, 서범수, 김수원, *박 경, *한우종
고려대학교 전자공학과 ASIC연구실
*한국전자통신연구원 컴퓨터시스템 연구부
Tel. 02-923-2081, Fax. 02-928-1216, Email. yskim@asic.korea.ac.kr

Functional Verification of 64bit RISC Microprocessor

Youn-Sun Kim, Bum-Soo Suh, Soo-Won Kim, Park Kyoung and Woo-Jong Hahn
ASIC Laboratory, Department of Electrical Engineering, Korea University
*Computer System Research Department, ETRI
Tel. 02-923-2081, Fax. 02-928-1216, Email. yskim@asic.korea.ac.kr

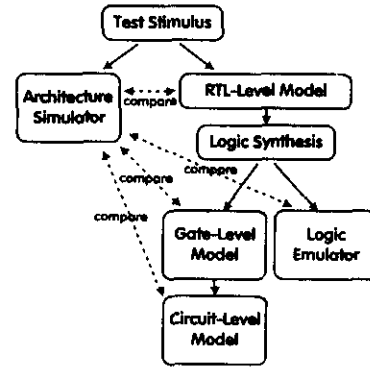
Abstract - As the performance of microprocessor improves, the design complexity grows exponentially. Therefore, it is very important to make the bug-free model as early as possible in a design life-cycle.

This paper describes the simulation-based functional verification methodology for the RTL level description model. It is performed by multi-stage verification methods using extensive hand-generated self-checking tests supplemented with random tests. This approach is applied to the functional verification of the GPU processor of Raptor and various bugs are detected.

I. 서론

공정 기술과 회로 기술의 급격한 발달은 고성능 마이크로 프로세서의 출현을 가능케 하였으나 프로세서의 성능이 향상될수록 하드웨어의 복잡도는 더욱 증가되었기 때문에, 하드웨어의 기능을 검증하는 일은 하드웨어를 설계하는 일보다 더욱 어렵고 복잡한 일이 되어가고 있다. 뿐만아니라 새로운 마이크로 프로세서 제품이 컴퓨터 시장에 나오는데 걸리는 시간인 타임-투-마켓(Time-to-Market)은 점점 감소되고 새로운 프로세서의 출현에 대한 컴퓨터 사용자의 기대도는 더욱 높아졌기 때문에 이러한 설계 현실을 뒷받침해 줄 수 있는 방안에 관한 연구들이 매우 중요한 과제로 여겨지게 되었으며, 이러한 노력의 일환으로 등장한 것이 설계의 병렬성 향상을 위한 기능 검증 방법 연구이다.

설계된 마이크로프로세서의 기능을 검증하는 것은 그림 1.1 와 같이 설계 과정 전체를 통해 이루어진다. 그러나 가장 핵심적인 단계가 되는 것은 설계의 초기 단계에 이루어지는 RTL(Register Transfer Level) 레벨의 기능 검증이다. RTL 레벨의 기능 검증은 게이트 레벨이나 회로 레벨의 검증에 비해 훨씬 적은 검증 시간을 가지고도 비교적 정확한 검증을 할 수 있으므로 하드웨어의 복잡도가 증가될수록 더욱 중요한 설계 요소가 되었다.



<그림 1.1> 마이크로 프로세서의 검증 과정

본 논문에서는 검증의 초기 단계에 설계된 마이크로프로세서 모델의 기능을 보다 효율적이고 체계적으로 검증하기 위한 방법들을 모색해보고 이를 RTL 레벨로 기술된 RAPTOR 의 내부 프로세서 유닛인 GPU 에 적용하였다.

II. 검증 방법의 종류

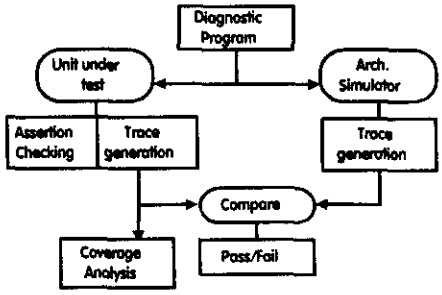
2.1 유닛 레벨 검증

유닛 레벨의 기능 검증은 프로세서를 구성하는 각각의 서브 모듈들에 대한 기능을 검증하는 것으로서 각 유닛의 정확한 동작을 시험할 수 있는 테스트 벡터를 구동함으로

이루어진다. 따라서 각 유닛이 지니게 되는 특성에 따라 매우 다양한 결과 값의 확인 방법이 존재하며 가능한 한 각 유닛에서 발생될 수 있는 다양한 경우의 테스트를 하는 것이 매우 중요하다. 그러나 이러한 검증은 하드웨어를 설계한 툴을 재사용하여 각 모듈에 대한 독립적인 검증을 수행하는 것이므로 검증할 수 있는 범위에 한계가 있으며 설계 엔지니어에 의해 이루어져야 하기때문에 설계의 병렬성을 향상시키기 위한 방법으로는 적합하지 않다.

2.2 통합 레벨 검증

통합 레벨에서의 기능 검증은 RTL 레벨과 게이트 레벨 그리고 스위치 레벨로 나뉘어 진행된다. 그러나 하위 레벨로 갈수록 검증을 위해 필요한 데이터의 양은 수십 배 혹은 수백 배 까지 증가하므로 마이크로 프로세서의 성능이 향상되고 그 기능이 복잡해질수록 프로세서의 기능을 검증하는 일은 게이트 레벨이나 스위치 레벨보다는 RTL 레벨로 집중되고 있으며, 이를 뒷받침하기 위한 다양한 연구들이 진행되고 있다. 통합 레벨에서의 기능 검증에서는 각 유닛의 기능 및 유닛 상호 간의 인터페이스들이 주어진 설계 사양대로 설계되었는가를 확인하는 과정이 이루어지며 따라서 독립적으로 설계된 모듈들이 하나의 모듈로 통합될 때 생겨나는 에러들이 주로 검출된다. 이러한 검증에는 다양한 방법들이 적용될 수 있지만 본 논문에서는 그림 2.1과 같은 시뮬레이션을 기반으로 한 방법만을 언급하였다.

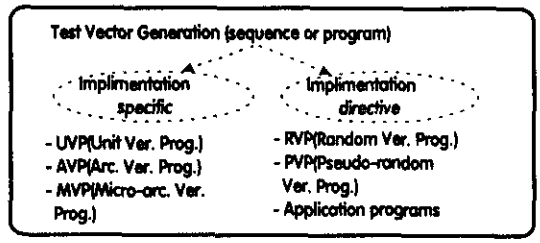


<그림 2.1> 시뮬레이션 기반의 기능 검증

시뮬레이션을 통해 이루어지는 기능 검증은 자기 진단 방법(Self Checking Method)과 기준 모델 비교법(Reference Model Comparison Method) 그리고 어썰션 체킹 방법(Assertion Checking Method) 등을 통해 이루어진다. 자기 진단 방법은 검증 모델에 버그의 리포팅(Reporting) 메커니즘이 포함된 프로그램을 입력 스티뮬러스(Input Stimulus)로 주고 검증하는 것이고, 어썰션 체킹 방법은 설계된 모델에 어썰션 체킹을 위한 루틴을 첨가하는 것이다. 기준 모델 비교 방법은 레퍼런스 모델로서 아키텍쳐 시뮬레이터를 사용하는 방법으로 동일한 테스트 벡터를 검증하고자 하는 하드웨어 모델과 레퍼런스 모델의 입력 스티뮬러스로 주고 시뮬레이션하는 것을 말한다.

III. 검증 프로그램의 종류

검증 모델과 기준 모델의 입력 스티뮬러스로 사용되는 테스트 프로그램은 그 사용 용도와 생성 방법에 따라 다양한 종류들이 있으며 여기에는 특정한 아키텍쳐의 기능 검증을 목적으로 작성되는 것(Implementation Specific Program)과 일반적인 아키텍쳐의 기능 검증을 목적으로 작성되는 것(Implementation Directive Program) 등이 있다.



<그림 3.1> 검증 프로그램의 종류

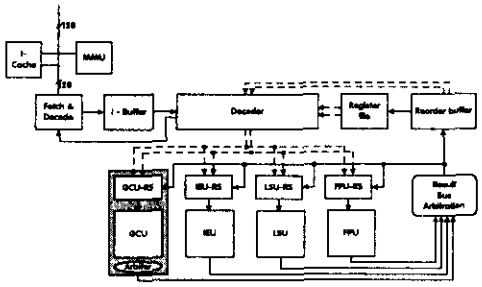
특정한 아키텍쳐의 기능 검증을 목적으로 사용되는 테스트 프로그램은 그림 3.1에서 볼 수 있는 바와 같이 주로 핸드 제너레이션되는데, 이들은 다시 유닛 검증 프로그램과 아키텍쳐 검증 프로그램 그리고 마이크로-아키텍쳐 검증 프로그램의 세 가지로 나뉘어 진다. 이와 같은 테스트 벡터들에는 자기 진단 메커니즘과 커버리지 분석(Coverage Analysis) 메커니즘들이 첨가 될 수 있기 때문에 설계된 하드웨어를 보다 더 효과적으로 검증할 수 있는 수단이 되기는 하지만 프로그램을 작성하는 데 많은 시간과 노력이 들게 된다는 것과 설계 모델의 모든 에러를 검출해낼 수 없다는 단점이 존재한다.

일반적인 아키텍쳐의 기능 검증을 목적으로 사용되는 테스트 벡터들에는 랜덤 검증 프로그램과 의사 랜덤 검증 프로그램 그리고 응용 프로그램들이 있다. 이러한 프로그램들은 보다 다양한 명령어들의 조합을 가지고 프로세서의 기능을 검증할 수 있기 때문에 특정한 아키텍쳐의 기능 검증을 위한 검증 프로그램들로는 검증해내기 어려운 에러들을 검출해 낼 수 있으며 따라서 핸드 제너레이션되는 검증 프로그램과 상호 보완적으로 사용된다..

IV. 기능 검증 모델

본 논문에서 검증하고자 하는 모델은 RAPTOR의 내부 프로세서인 GPU(General Processor Unit)이다. GPU는 하바드(Harvard) 구조의 1차 캐쉬를 내장하는 64비트 RISC 프로세서로서 SPARC Architecture Version 9 명령어 셋을 지원한다. GPU는 그림 4.1에서 볼 수 있는 바와 같이 윈도우 형태의 대용량 정수 레지스터 파일과 부동 소수점 레지스

터 파일 그리고 그래픽 레지스터 파일을 모두 가지고 있으며 정수 연산 유닛, 부동 소수점 연산 유닛, 로드 스토어 유닛 그리고 그래픽 명령어 처리 유닛을 명령어 실행 유닛으로 가지고 있다. 또한 명령어 페치 유닛과 디코더 유닛 그리고 효과적인 명령어 처리를 위한 리오더 버퍼(Reorder Buffer) 유닛과 레져베이션 스테이션(Reservation Station) 유닛 그리고 결과 버스 중재 유닛 등도 포함되어 있다.



<그림 4.1> GPU 의 블록 다이어그램

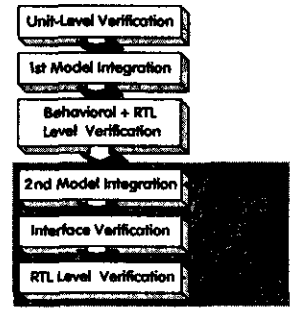
GPU 의 가장 큰 특징은 명령어의 버퍼링(Buffering)과 트랩 처리에 있다. 명령어의 버퍼링은 리오더 버퍼와 레져베이션 스테이션을 통해 이루어지며, 트랩은 SPARC Architecture Version9 의 특징인 다단계 트랩 처리와 리오더 버퍼를 통한 프리사이즈 트랩(Precise Trap) 처리를 통해 이루어진다. 이 중에서 특히 리오더 버퍼는 비순차적인 명령어 완료(Out-of-Order Completion)에 의해 발생되는 문제점들을 해결하기 위해 사용되었으며 레져베이션 스테이션은 피연산자의 종속성 문제로 인한 성능 저하를 감소시키기 위해 사용되었다.

V. 기능 검증 구현

5.1 시뮬레이션 방법

GPU 를 검증하는 데에는 그림 5.1 과 같은 다단계의 검증 방법이 사용되었다. 검증의 초기 단계에는 각각의 서브 모듈에 대한 스탠드 얼론(Stand Alone) 테스트를 수행하였고 다음에는 각각의 모듈들을 통합하는 과정을 진행하였으며 그 다음 단계에는 행위 레벨(Behavioral Level)로 기술된 모듈을 포함한 통합 검증을 수행하였다. 이를 위해서는 먼저 각 단계별로 요구되는 각각의 서브 모듈들을 행위 레벨로 기술하는 작업이 이루어져야 하는데, 본 논문에서는 명령어 실행 유닛이나 명령어 페치 유닛처럼 설계하는데 많은 시간과 노력이 요구되는 유닛들에 적용함으로써 설계와 검증의 병렬성을 최대화하도록 하였다.

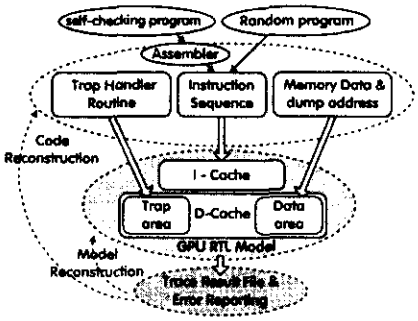
따라서 이 단계에는 RTL 레벨로 기술된 나머지 유닛들에 대한 디버깅이 진행되었는데, 이 때에는 개별적인 명령어의 수행 확인을 통해 어떤 한 명령어에 잠재될 수 있는 오류들을 다른 명령어와 격리시켜 에러를 검출해낸 다음 다양한 명령어들의 조합에 의해 생겨날 수 있는 에러들을 다시 검출해내는 방법을 사용하였다.



<그림 5.1> 다단계의 기능 검증

검증 엔지니어가 위와 같은 과정을 통해 설계된 모델의 기능을 어느 정도 정립시켜 나가는 동안 설계 엔지니어는 행위 레벨로 기술된 모듈을 RTL 레벨로 변환하는 작업을 진행하고 검증 엔지니어는 RTL 모델 통합을 위한 간단한 인터페이스 모델과 매뉴얼을 작성하여 일차적인 인터페이스 검증을 수행하였다. 마지막 단계의 검증은 인터페이스 검증이 완료된 후에 진행되었으며 이 때에는 통합 단계에서 발생될 수 있는 에러 검출을 위한 집중적인 테스트가 수행되었다. 그러나 이러한 검증만으로는 설계 모델에 대한 높은 신뢰도를 얻기 어려우므로 약 150 개의 자기 진단 프로그램을 통한 포커스드-테스팅(Focused Testing)과 기준 모델 비교법을 병행하였으며, 비교적 간단한 응용 프로그램들을 실행시켜 보았다.

5.2 시뮬레이션 환경



<그림 5.2> 시뮬레이션 환경

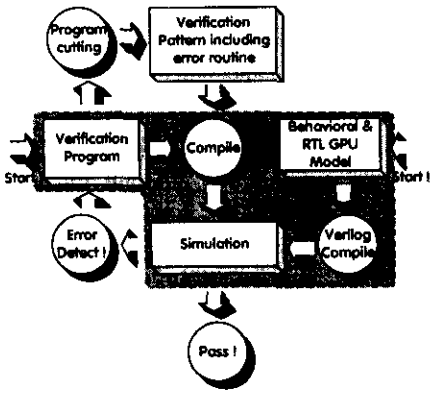
GPU 의 기능 검증을 위한 시뮬레이션은 그림 5.2 와 같은 과정으로 진행되었다. GPU 프로세서는 Verilog-HDL 언어로 기술된 모델이므로 시뮬레이터로는 Verilog-XL 로직

시뮬레이터를 사용하였으며, 어셈블리 언어로 작성된 자기 진단 검증 프로그램과 인-하우스(In House) Tool인 RandGen에 의해 생성된 랜덤 검증 프로그램을 입력 스티뮬러스로 사용하였다. GPU의 시뮬레이션에는 입력 스티뮬러스 이외에도 트랩처리를 위한 트랩 테이블과 핸들러 루틴 그리고 로드 스토어 연산에 필요한 메모리 데이터들을 시뮬레이션 모델에 제공해주는 일이 필요하므로 그림 5.2에 나타낸 바와 같이 GPU 프로세서에 내장된 명령어 캐쉬와 데이터 캐쉬를 통하여 구현하였다.

5.3 디버깅 과정

마이크로프로세서와 같이 복잡한 구조로 설계되는 하드웨어 모델은 검출해내기 어려운 에러를 포함하고 있기 때문에 이러한 에러를 디버깅하는 과정은 매우 중요하다. 특히 칩에서 발생될 수 있는 다양한 논리적인 상호 작용을 테스트하는 것은 매우 어려운 일이므로 보다 효과적인 에러 검출 과정과 디버깅 과정이 요구되는데, 본 논문에서는 다음의 그림 5.3과 같은 과정을 통해 에러의 검출과 디버깅을 수행하였다.

먼저 시뮬레이션은 검증 모델과 검증 프로그램의 컴파일로부터 시작된다. 이 때 사용되는 검증 프로그램은 어셈블리 언어로 작성된 명령어의 시퀀스들이거나 랜덤한 명령어 시퀀스들이므로 에러를 검출해내는 과정은 레지스터 또는 메모리 내용의 리포팅 루틴이나 결과 값의 비교 분석을 위한 프로그램의 수행을 통해 이루어졌다. 에러가 검출된 경우 디버깅은 검증에 사용된 전체 프로그램 중에서 에러를 발생시킨 명령어를 포함하는 프로그램을 다시 작성하여 재시뮬레이션하는 방법을 사용하였다.



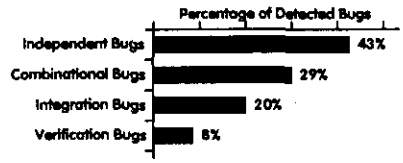
<그림 5.3> 에러 검출과 디버깅 과정

VI. 기능 검증 결과

본 논문에 제안된 기능 검증은 그림 6.1과 같은 에러 리포팅 메커니즘을 통하여 이루어졌으며, 검출된 에러의 분포는 그림 6.2과 같다.

```
< Register File Write Stage >
[W0]    INS              LOCAL            OUT              GLOBAL
  0: 0000000000001011 0000000000000000 0000000000000000 0000000000000000
  1: 8000000000000000 0000000000000000 0000000000000000 8000000000000000
  2: 7fffffff00000000 0000000000000000 0000000000000000 7fffffffffffffff
  3: 0000000000000000 0000000000000000 0000000000000000 fffffffffffffffe
  4: ffffffffffffffff 0000000000000000 0000000000000000 ffffffffffffffff
  5: 8000000000000000 0000000000000000 0000000000000000 0000000000000001
  6: xxxxxxxxxxxxxxxx 0000000000000000 0000000000000000 0000000080000000
  7: xxxxxxxxxxxxxxxx 0000000000000000 0000000000000000 xxxxxxxxxxxxxxxx
[W7]    INS              LOCAL            OUT              GLOBAL
  0: 0000000000000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx 0000000000000000
  1: 0000000000000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx 8000000000000000
  2: 0000000000000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx 7fffffffffffffff
  3: 0000000000000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx fffffffffffffffe
  4: 0000000000000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx ffffffffffffffff
  5: 0000000000000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx 0000000000000001
  6: 0000000000000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx 0000000080000000
  7: 0000000000000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
=> Display the Contents of Nonprivileged Registers:
    31245 PC:0000000000000aa0 nPC:0000000000000aa4 CCR:b8
    (XCC:NZVC=1011 ICC:NZVC=1000)
=> Display the Contents of Privileged Registers:
    31245 PSTATE:000
=> Display the contents of Window State Registers:
    31245 CWP:00
L75 "rap_reg_sim.s": $finish at simulation time 31275
```

<그림 6.1> 에러 리포팅 메커니즘



<그림 6.2> 검출된 에러 분포

VII. 결론

본 논문에서는 설계의 초기 단계에 마이크로프로세서의 기능을 보다 효율적이고 체계적으로 검증하기 위한 방법들을 모색하고 이를 RTL 레벨로 기술된 GPU 프로세서에 적용해보았다. 그 결과 제안된 검증 방법은 설계 시간의 단축과 신뢰도의 향상을 가져올 수 있었으며, 설계와 검증의 병렬성을 최대화할 수 있었다.

참고문헌

[1] M. Kantrowitz et al., "I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor", in Proc. DAC, pp.325-330, 1996.

[2] J. Monaco et al., "Functional Verification Methodology for the PowerPC 604™ Microprocessor", in Proc. DAC, pp. 319-324, June 1996.

[3] Scott Taylor et al., "Functional Verification for a Multiple-issue, Out-of-Order, Superscalar Alpha Processor-The DEC Alpha 21264 Microprocessor", in Proc. DAC, pp.638-643, 1998.