논문 2004-41SD-7-7

마이크로프로세서를 위한 효율적인 기능 검증 환경 구현

(An Implementation of Efficient Functional Verification Environment for Microprocessor)

권 오 현*, 이 문 기**

(Oh-Hyun kwon and Moon-Key Lee)

요 약

본 논문은 마이크로프로세서의 설계과정 중, 중요도가 크게 부각되고 있는 기능 검증을 좀더 효율적으로 할 수 있는 검증 환경을 제안한다. 본 검증 환경은 테스트 벡터 생성부분, 시뮬레이션 부분, 결과 비교 부분으로 구성되어 있다. 기존에 사용되던 검증 방법보다 좀더 효율적인 기능 검증이 가능하도록 하기 위해 바이어스 랜덤 테스트 벡터 생성기를 사용하였고, 참조 모델로 재정의 가능 명령어 수준 시뮬레이터를 사용하였다. 본 검증 환경에서 수행된 결과를 비교함으로써 일반적인 테스트 벡터에서 발견하기 어려운 오류 유형을 발견하고 새로운 오류 유형의 기준을 제시하는 효과를 지닌다.

Abstract

This paper proposes an efficient functional verification environment of microprocessor. This verification environment consists of test vector generator part, simulator part, and comparator part. To enhance efficiency of verification, it use a bias random test vector generator. In a part of simulation, retargetable instruction level simulator is used for reference model. This verification environment is excellent to find error which is not detected by general test vector and will become a good guide to find new error type

Keywords: 마이크로프로세서, 기능 검증, 랜덤 테스트 벡터 생성기, 명령어 수준 시뮬레이터

I. 서 론

전체 시스템을 하나의 칩으로 구현하는 SoC (System On a Chip) 설계 기술이 발달하고, 이에 프로세싱 코어들이 포함됨으로 해서 하나의 시스템이 아주 복잡해졌다. 이로 인해, 내부에 포함되어 있는 마이크로프로세서의 기능 검증은 집적회로가 복잡해지고 설계기간과 설계비용이 증가하면서 매우 중요해졌다.[5]

일반적으로 HDL을 이용하여 기술된 마이크로프로세서에 대한 검증 방법은 정규검증과 시뮬레이션에 기초한 디자인 검증으로 크게 나눌 수 있다.[2]

정규 검증은 추상화된 디자인의 수학적인 기호를 이용하여 올바름을 증명하는 방법이고 게이트 수준 모델과 RTL 모델의 일치성을 확인하는 것과 같이 하위 수준의 추상화 모델에 적합하다. 이 방법은 아키텍처의 다른 구현 레벨 사이에 기능적 차이가 없음을 보여주는 equivalence checking과 디자인의 특정 기능의 올바름을 검증하는 model checking을 포함한다. 이 방법은 대부분의 경우에 수작업으로 진행되며 디자인이 상위 수준 스펙을 만족시키는지 보증하기 어렵고 검증할 수 있는 회로나 상태 머신의 크기에 제한이 있다.[3]

시뮬레이션에 기초한 방법은 실제 하드웨어 시스템을 모델링한 환경에서 디자인의 소프트웨어 모델을 적용하여 검증하는 과정이다. 마이크로프로세서는 명령어에 의해 개별적인 동작이 정의되는 범용 칩으로서 최종적인 소프트웨어 응용 형태는 마이크로프로세서 설계자가 아닌 프로그래머가 명령어를 조합하여 결정한다. 따라서 프로세서 설계자는 개개 명령어의 고유 기능뿐만 아니라 임의의 명령어 조합에 대해 발생 가능한 모든 측면에 대해 동작의 안정성을 보장해야 한다. 이러한

* 정회원, 삼성전자 DM연구소
(Samsung Electronics Digital Media R&D Center)
** 종신회원, 연세대학교 전기전자공학과
(Dept. of Electrical and Electronic Eng., Yonsei Univ.)
접수일자: 2004년1월19일, 수정완료일: 2004년6월16일

이유로 시뮬레이션에 기초한 방법은 마이크로프로세서의 설계 과정에서 설계자가 디자인의 기능 검증과 타이밍 검증을 위해 사용하는 검증 방법으로 적용되고 있다.[4][6]

그러나 시뮬레이션을 통한 기능 검증 방식은 검증 결과의 신뢰성이 기능 검증을 위해 사용되는 존재 가능한 모든 명령어 조합인 워크로드의 수행 량에 따라 결정된다. 따라서 이론적으로 완벽한 신뢰성을 얻기 위해서는 존재 가능한 모든 명령어 유형에 대한 시뮬레이션을 수행해야 하지만, 마이크로프로세서와 같은 복잡한 시스템의 경우 적용이 불가능하다. 따라서 효율적인 기능 검증을 위해서는 검증 대상 마이크로프로세서와 참조 모델인 시뮬레이터와의 연동과 발생 가능한 모든 오류 상황(코너 케이스)을 대표할 수 있는 적절한 양의 표본 테스트 벡터가 요구된다.[1]

또한, 마이크로프로세서를 포함하고 있는 SoC 시스템은 소프트웨어를 포함하고 있어야 되며, 이는 개발과정에서도 하드웨어와 소프트웨어가 같이 설계되어야 하고 검증되어야 함을 의미한다. 그러므로 마이크로프로세서 구현과 소프트웨어 개발을 동시에 진행하면서, 동시에 통합 검증을 수행함으로서 빠른 시간 내에 마이크로프로세서의 기능 검증을 할 수 있는 환경이 필요하다.

본 논문에서는 효율적인 기능 검증을 위해서 검증 대상 마이크로프로세서와 참조 모델인 명령어 수준 시뮬레이터와의 연동 관계와 모든 코너 케이스를 대표할 수 있는 적절한 양의 표본 테스트 벡터를 이용하여 마이크로프로세서의 기능을 검증하게 된다. 테스트 벡터에는 바이어스 랜덤 벡터 생성기를 이용한 테스트 벡터와 소프트웨어와의 통합 검증을 위한 C 소스 파일로 구성된다.

본 논문에서 제안한 기능 검증 환경에서 연세대학교에서 개발한 32비트 RISC 마이크로프로세서인 SMA-RT 7을 이용하여 성능과 효율성을 측정하였다.

본 논문의 체계는 다음과 같다. 제 Ⅱ장에서 검증 환경의 구성 방안에 대해 논하고, 제 Ⅲ장에서는 제안한 검증 환경의 성능 및 검증 결과에 대해 논하고 마지막으로 제 Ⅳ장에서 결론을 도출한다.

Ⅱ. 검증 환경의 구성

본 논문에서 제안한 마이크로프로세서의 효율적인 기능 검증을 위한 검증 환경은 크게 테스트 벡터 생성 부분, 마이크로프로세서와 참조 모델의 시뮬레이션 부분, 그리고 각각의 시뮬레이션 결과를 비교하는 부분으로 나누어진다.

마이크로프로세서의 기능 검증환경에서 테스트 벡터 생성 부분을 구성하고 있는 것은 마이크로프로세서의 효율적인 기능 검증을 위한 랜덤 벡터 생성기 부분, 인터럽트 처리를 검증하기 위한 테스트 벡터 부분, 그리고 소프트웨어와의 통합 검증을 위한 소프트웨어 프로그램 부분으로 이루어진 검증 테스트 벡터부분과 소프트웨어 개발 도구로 나누어진다.

시뮬레이션 부분은 테스트 벡터 생성 부분에서 생성된 테스트 벡터를 검증하고자 하는 마이크로프로세서와 참조 모델에 인가하여 시뮬레이션을 수행하는 부분이다. 본 검증환경에서는 연세대학교에서 개발한 SMA-RT 7 마이크로프로세서를 검증 목적 마이크로프로세서로 설정하고, 참조 모델로 재정의 가능 명령어 수준 시뮬레이터를 채택하였다.

마지막으로 시뮬레이션 결과 비교 부분은 검증 목적 마이크로프로세서와 명령어 수준 시뮬레이터의 결과를 비교하여 그 결과를 보이고, 기능 검증 수행 중 발생된 오류를 보고하는 역할을 하는 부분이다.

그림 1은 본 논문에서 제안한 검증 환경의 전체 블록 다이어그램을 나타낸다.

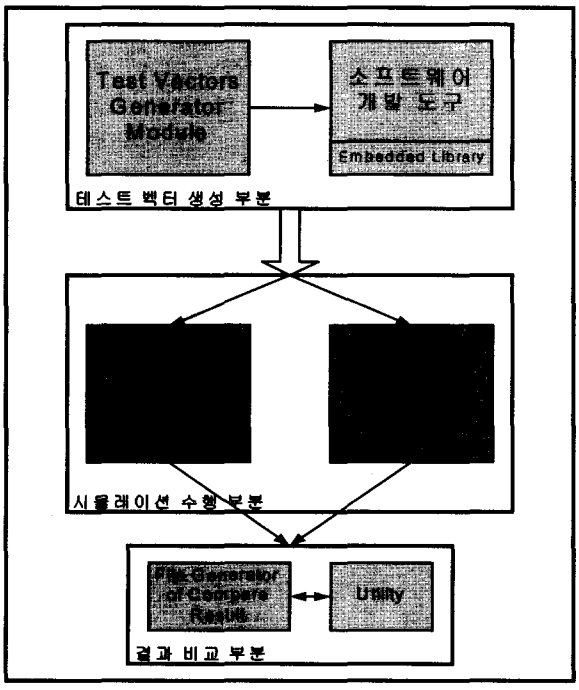

그림 1. 마이크로프로세서 검증 환경의 전체 블록 다이어그램

Fig. 1. A block diagram of verification environment for microprocessor.

2.1 테스트 벡터 생성

본 검증환경에서 사용한 시뮬레이션을 통한 기능 검증 방식은 검증 결과의 신뢰성이 워크로드의 수행 량에 비례하는 귀납적인 특성을 지닌다. 따라서 이론적으로 완벽한 신뢰성을 얻기 위해서는 존재 가능한 모든 명령어 조합 유형에 대한 시뮬레이션을 수행해야 하지만, 마이크로프로세서와 같은 복잡한 시스템의 경우, 적용이 불가능하다. 그러므로 효율적인 기능 검증을 위해서는 검증 대상 시스템의 모든 코너 케이스를 대표할 수 있는 적절한 양의 표본 테스트 벡터와 검증 방법이 어려워 쉽게 할 수 없는 내부 혹은 외부 인터럽트의 처리에 관한 검증에 있어서 효율적으로 검증 할 수 있는 테스트 벡터가 요구된다. 기존에는 설계자가 미리 발생 가능한 오류 상황을 예측하여 직접 작성하는 방식을 주로 사용하였으나 이러한 방식은 오류 검출에 유리한 방향으로 명령어 조합을 조절하기 힘들기 때문에 부적절하다. 반면 이에 대한 대안으로 사용되는 랜덤 테스트 벡터는 작성이 매우 간편하며 설계자가 미처 간파하지 못한 부분에 대한 오류 발견에 매우 탁월하고 새로운 오류 유형의 정립 지침을 제공하기도 한다. 그러나 실제 오류를 검출하는 테스트 벡터 표본은 모집합에 대하여 균일하게 분포하는 것이 아니라 특정 유형과 강한 상관관계를 지니는 경우가 대부분이다. 따라서 순수 랜덤 벡터의 경우 이러한 상관관계를 제대로 반영하지 못하기 때문에 효율성이 매우 떨어지는 단점이 있다.[1]

따라서, 본 논문에서는 위의 단점을 보완하기 위해 설계자 의도에 따른 테스트 벡터와 랜덤 테스트 벡터의 장점을 포괄한 새로운 형태의 바이어스 랜덤 벡터 생성기를 설계하였다. 바이어스 랜덤 벡터 생성기는 어셈블리 언어 수준의 워크로드 소스 코드에 대해 구문 단위로 무작위성을 제어하는 기능을 제공한다. 따라서 설계자는 검증하고자 하는 기능에 대한 상관관계를 미리 설

표 1. 랜덤 벡터 생성기 전처리 명령어
Table 1. Pre-processor instruction of the random vector generator.

구문	동작
%DEVAR `identifier lower : upper : step	[lower, upper] 범위의 step에 대한 등차수열 형태를 지닌 정수 랜덤 마크로 변수를 정의
%DEVAR `identifier string	string을 값으로 하는 문자열 랜덤 마크로 변수 정의
%DEVAR `identifier $`pre_defined_identifier	이미 정의된 마크로 변수를 포함하는 새로운 랜덤 마크로 변수 정의
%DEVAR `identifier (first, ..., last)	값 집합을 갖는 램덤 마크로 변수 정의, 값 집합 요소는 정수, 문자열, 그리고 랜덤 마크로 변수를 포함
%DEVAR `identifier [width] (first_value [first_bit_range], ... last_value [last_bit_range])	비트열 랜덤 마크로 변수 정의, width는 비트열의 길이, value, range 쌍은 각각 랜덤하지 않은 부분 비트 열의 값과 위치를 나타낸다.
%DEVAR `identifier [UNIQUE]	할당 가능한 마크로 변수 정의
%DEFUN `identifier [GUARD] `statements %ENDFUN	마크로 함수 정의, 함수 인자의 프로토타입 검사를 하지 않는다.
%SELECT `first_statements %OR `second_statements %OR ..... %ENDSEL	무작위로 하나의 구역만을 선택하여 출력한다.
%REPEAT number [GUARD] `statements %ENDREP	무작위로 구역을 number만큼 반복하여 출력한다.
$`identifier	정의된 마크로 변수의 쓰기/읽기 접근
$`identifier (first, ..., last)	정의된 마크로 함수의 호출
$`number	호출된 마크로 함수 내에서의 함수 인자를 지칭, 숫자는 함수 호출 시 인자의 순서를 나타낸다.
%EVAL `identifier `express	마크로 변수와 상수로 이루어진 식을 계산하여 마크로 변수에 할당

정하고 부분적으로 무작위성을 부여함으로써 랜덤 벡터의 효율성을 극대화 시킨다.

2.1.1 바이어스 랜덤 벡터 생성기의 구조

2.1.1.1 랜덤 벡터 시뮬레이션의 고려사항

본 검증 환경에서 사용된 랜덤 벡터 시뮬레이션은 정상적인 동작을 위해서 다음과 같은 특수성을 고려해야 한다.

랜덤 벡터는 어셈블리 언어 수준의 워크로드 소스 코드에 대한 구문 단위의 무작위성을 제어 하여 생성된 테스트 벡터라는 특성상 코드 분기 주소나 데이터 메모리 주소가 정의되지 않거나 접근 불가인 메모리 영역을 임의로 참조할 수 있다. 이로 인해 랜덤 벡터 시뮬레이션이 비정상적인 상태가 되어 더 이상의 시뮬레이션을 수행할 수 없는 경우가 발생하거나, 잘못된 명령어를 수행하여 잘못된 결과를 야기할 수도 있다. 이러한 특수성을 해결하기 위해 본 논문에서 제안하는 방법은 코어에서 인출되는 주소를 디코딩하여, 위와 같이 접근이 불가능한 주소나 메모리 영역을 참조할 경우, 마이크로프로세서로 Abort 신호를 내보내 마이크로프로세서의 동작을 일시적으로 멈추게 한 후, Exception Routine을 수행하게 하는 방법이다. 이 방법은 별도의 명령어 시뮬레이터나 명령어 변환기가 필요 없고, 임의의 마이크로프로세서에서도 사용이 가능하며 수행될 Exception Routine에 해당하는 프로그램을 작성해서 간단히 해결할 수 있다.

2.1.1.2 바이어스 랜덤 벡터 생성기의 사양

C/C++과 같은 상위 수준 언어는 구문 합성을 위해 사용되는 기계어의 종류와 조합이 제한되므로 기능 검증용 워크로드로는 어셈블리 언어가 더 적합하다. 따라서 본 랜덤 벡터 처리기는 기본적으로 어셈블리 수준 소스 코드의 전처리기로서 동작하도록 설계하였다.

사용자는 어셈블리 소스 코드에 전처리기 명령어를 삽입함으로써 무작위성을 부여하고 벡터 생성기는 전처리기 명령어를 해석 수행하여 최종적인 어셈블리 소스 코드를 생성한다.

본 검증 환경에서 사용된 바이어스 랜덤 벡터 생성기가 지원하는 전처리 명령어는 크게 3가지 부류로 분류된다.

첫 번째 유형은 무작위 하게 문자열이나 정수, 비트열을 발생시키는 랜덤 마크로 변수이다. 랜덤 마크로 변수는 일반적인 프로그램 언어의 마크로 변수와 유사하나 차이점은 후자가 한 가지 값을 가지는 반면 전자는 미리 정의된 값 집합에서 무작위로 선택된 요소를 취한다는 점이다.

두 번째 유형은 구문 생성을 제어하는 구문 제어 문으로 특정 구역을 정해진 횟수만큼 반복 생성하는 REPEAT문과 여러 구문 중 무작위로 선택된 하나의 구문을 생성하는 SELECT문으로 구분된다.

세 번째 유형은 마크로 함수 구문으로 일반적인 프로그램 언어의 마크로 함수와 동일하다. 전처리기 명령어의 자세한 사양은 표 1에 정리하였다.

2.1.1.3 바이어스 랜덤 벡터 생성기 구현

바이어스 랜덤 벡터 생성기의 내부 구조는 그림 3과 같다. 바이어스 랜덤 벡터 생성기는 전처리기 명령어가 포함된 어셈블리 소스 코드를 입력 받아 중간 코드를 합성한다. 중간 코드는 최종 출력을 수행하기 위한 전단계 원시 코드로서 변수 값 생성, 출력 생성을 위한 코드 제어 및 최종 문자열 출력 등을 수행한다. 정의된 중간 코드는 표 2와 같다.

바이어스 랜덤 벡터 생성기는 REPEAT에 의한 반복문이나 마크로 함수 호출 수행 시, 동일한 레이블이 반복 출력되는 문제가 발생한다. 이 경우 랜덤 벡터 생성기 자체는 오류가 아니나 이후 어셈블러에 의해 레이블 중복 정의 오류가 발생한다. 본 바이어스 랜덤 벡터 생성기는 이러한 문제점을 해결하기 위해 두 가지 방법을 제안한다.

```
// 랜덤 변수 정의 및 치환
%DEVAR OPCODE (STR, MOV)
$OPCODE R3, [R4, R5]        ->    STR R3, [R4, R5]
                            or    MOV R3, [R4, R5]
%DEVAR INDEX 0:15:2
MOVEQ R$INDEX, R5           ->    MOVEQ R6, R5

// 반복 구문
%REPEAT 2
MUL R$INDEX, [R7]           ->    MUL R2, [R7]
LDM R$INDEX, R9                   LDM R2, R9
%ENDREP                           ----------
                                  MUL R4, [R7]
                                  LDM R4, R9

// 랜덤 선택 구문과 반복 구문
%REPEAT 4
%   SELECT
      ASR R0, R1            ->    ASR R0, R1
%   OR                              ----------
      MOV R1, R2                    NOP
%   OR                              ----------
      NOP                           MOV R1, R2
%   ENDSEL                          ----------
%ENDREP                           MOV R1, R2

// 매크로 함수 정의 및 호출
%DEFUN F
      MOV $1, $2
%ENDFUN

%REPEAT 2                   ->    MOV R0, R2
    $F(R$INDEX, R2)                 ----------
%ENDREP                           MOV R8, R2
```

그림 2. 랜덤 벡터 생성기 전처리 명령어 사용 예
Fig. 2. A example of pre-processor instruction in the random vector generator.

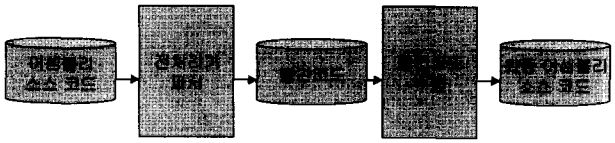

그림 3. 랜덤 벡터 생성기 내부 동작 흐름도
Fig. 3. An implementation flow in the random vector generator.

표 2. 랜덤 벡터 생성기의 중간 코드
Table 2. A middle code of the random vector generator.

코드 니모닉	동작
EXIT	종료
JA *L*	*L* 레이블로 무조건 분기
JC [*C*] *L*	카운트 *C*가 0이 되면 *L*로 분기
JR *LA*	레이블 집합 *LA* 중 하나로 무작위 분기
AC CONST	어큐뮬레이터에 상수 저장
AC RAND	어큐뮬레이터에 랜덤 변수 저장
ENTER	마크로 함수 프레임 생성
BL *L*	마크로 함수 호출
LEAVE	마크로 함수 프레임 제거 및 리턴
PUSH	마크로 함수 인자 스택에 저장
AC ARG	어큐뮬레이터에 함수 인자 저장
OUT RAND	랜덤 변수를 출력
OUT ARG	함수 인자를 출력
OUT BUFFER	버퍼를 출력, 버퍼는 전처리기 명령어를 제외한 구문이 저장됨

```
// GUARD 키워드, LTEMP는 지역 레이블
%REPEAT 2 GUARD
LTEMP : MOV R0, R1        ->    REPEAT_10_0:
%ENDREP                           LTEMP : MOV R0, R1
                                  ENDREP_10_0:
                                  REPEAT_10_1:
                                    LTEMP : MOV R0, R1
                                  ENDREP_10_1:

// UNIQUE 키워드
%DEVAR X UNIQUE

%REPEAT 4                 ->    X_0 : MOV R0, R1
$X : MOV R0, R1                 X_1 : MOV R0, R1
%ENDREP                         X_2 : MOV R0, R1
                                X_3 : MOV R0, R1
```

그림 4. 반복문이나 함수 호출 시 레이블 중복 방지
Fig. 4. A preventive measure of label overlap in an iteration statement.

첫 번째 방법은 REPEAT 문이나 마크로 함수 정의문에 GUARD라는 키워드를 사용하는 방식이다. 이 방식은 지역 레이블을 지원하는 어셈블러에만 적용 가능하다. GUARD 속성은 구역 경계에 광역 레이블을 삽입한다. 다라서 구역 내부에 동일한 지역 레이블을 사용해도 이름 공간 충돌은 방지할 수 있다.

```
        AREA test, CODE, READONLY
        EXPORT main

main
        BXVS R2

        MOVGE R7, R10, LSL #4

        TSTEQ R6, R13
        CMPHI R7, R1, ASL R8

        RSCGT R7, R0, R5
        ADDVS R9, R2, R7, ASL R1
        SUBGE R1, R11, R8, LSR #5

        MRSCS R9, SPSR

        MSRMI CPSR, R11

        MULEQ R4, R11, R12

        MLALT R12, R10, R1, R6
        MLAVCS R4, R12, R7, R10

        UMULLGT R3, R4, R5, R6

        LDRMI R6, [R1]
        STRVC R3, [R11, #8]

        STRCCB R13, [R5, #3]
        LDRGEB R3, [R4, #1]!

        LDMMIDB R10, {R11, R13, R2}
        STMEQDA R10, {R7, R13, R14}^

        SWIPL 70

        CDPHI p4, 4, c7, c1, c4

        LDCGT p8, c4, [R2]

        MCRVC p2, 5, R3, c5, c6

        SWPHI R1, R9, [R9]

END
```

그림 5. 랜덤 벡터 생성기를 이용한 랜덤 테스트 벡터 예
Fig. 5. Random test vector using random vector generator.

두 번째 방법은 UNIQUE라는 키워드를 사용하는 방식이다. 마크로 변수에 UNIQUE 속성을 부여할 경우, 접근할 때마다 카운트를 수행하여 고유의 키를 발생시킨다. 따라서 이를 레이블로 이용하여 동일한 이름이 발생하는 것을 방지할 수 있다.

2.1.1.4 소프트웨어 비동기적 기능 검증

SoC 코어용 임베디드 프로세서의 응용 분야는 시간 제약이 엄격하기 때문에 빠른 응답시간을 보장해야만 한다. 따라서 고속의 정확한 인터럽트 및 컨텍스트 스

위칭 구현이 필수적이다. 또한 대부분의 임베디드 프로세서는 저전력 관리를 위한 클락 동작 모드를 지원하며 JTAG을 이용한 에뮬레이션 기능 첨가가 보편화되고 있다. 그러나 이러한 동작들은 전체 모듈과 폭넓게 연관되는 경향을 보이기 때문에 소프트웨어 수행 시 설계 오류가 발생하기 쉽다. 따라서 설계자는 이러한 비동기적 동작에 대한 다양한 시나리오를 수립하고 시나리오가 성립된 상황에서 소프트웨어를 수행시켜야 한다. 그러나 이러한 과정은 많은 수작업을 요구하는 매우 번거로운 일이다. 따라서 본 검증 환경에서는 이러한 소프트웨어 비동기적 기능 검증을 위한 효율적인 방안을 제시한다.

본 검증 환경에서 제시하는 방안은 특정 테스트 벡터가 수행될 때에 검증 대상인 소프트웨어 비동기적인 동작이 발생하도록 한다. 테스트 벡터는 PC 값과 연관되어 있으므로 특정 PC 값에 이벤트가 발생하여 신호가 구동되게 함으로써 위와 같은 기능의 구현이 가능하다.

본 방안은 특정 테스트 벡터의 PC 값을 설계자가 직접 계산하는 대신 원하는 하드웨어 동작을 어셈블리 소스 파일에 첨부하면 전처리기가 자동으로 PC 값을 구하고 설계자가 명시한 하드웨어 동작에 대한 HDL 모델을 합성한다. 그림 6은 이에 대한 예를 도시한 것이다.

전처리기 명령어는 크게 2가지 종류로 나뉘며 문법은 각각 다음과 같다.

Type 1
//{%
STATEMENT
%}//

Type 2
//stimulus: {SIGNAL_NAME} = [{DELAY}, {DURATION}] {VALUE0} : {VALUE1}

첫 번째 유형은 기준 시점에서 STATEMENT에 해당하는 하드웨어 동작이 발생한다. STATEMENT는 오류 없는 HDL 코드여야 한다. 한편 변수 앞의 $표시는 이름 공간이 충돌하는 것을 방지하기 위한 것으로써 변수 이름 생성 시 PC 값을 접미사로 첨가한다.

두 번째 유형은 기준 시점부터 DELAY 이후에 DURATION 동안 SIGNAL_NAME을 VALUE0으로 할당하고 이후 VALUE1로 할당한다. 이 유형은 외부 인터럽트 발생, 캐시 미스 발생으로 인한 코어홀딩 등 하드웨어 동작이 소수 특정 신호에 의해 제어될 경우 매우 편리하다.

```
LINE 8 PC 100: LD R0 [R1]
LINE 9 PC 100: /*%
LINE 10 PC 100:    DBUS=$Y+$Z
LINE 11 PC 100: %*/
LINE 9 PC 104: LD R2 [R1, 4]
LINE 10 PC 108: ADD R3, R0, R2
LINE 11 PC 108: //stimulus: nIRQ = [4, 2] 0 : 1
LINE 12 PC 10C: ST R3, [R1, 8]
```

(a) 하드웨어 동작 삽입

```
integer Y_100, Z_100;
always@( posedge CLK )
begin
  if( PC=='h100 )
    begin
      DBUS=Y_100+Z_100;
    end
  end

  always@( posedge CLK )
  begin
    integer Y_100, Z_100;
    if( PC=='h108 )
      begin
        #(4*PERIOD)
          nIRQ=0;
        #(2*PERIOD)
          nIRQ=1;
      end
  end
```

(b) 전처리기에 의해 생성된 HDL 파일

그림 6. 전처리기 명령어의 사용 일례
Fig. 6. An example of pre-processor instruction in non-sequence of software.

2.2 목적 마이크로프로세서의 설계 사양

구축된 검증 환경에서 기능 검증을 하게 되는 SMART7 마이크로프로세서는 저전력을 고려한 범용으로 사용할 수 있는 32비트 RISC 마이크로프로세서로 연세대학교에서 설계한 것이고 HDL로 기술되었다. SMART7 마이크로프로세서는 내장형 응용에 적합한 작은 코드 크기와 빠른 컨텍스트 스위칭 속도를 제공한다. 명령어 군은 11개의 기본 형태로 구성되며 세 종류의 코프로세서 인터페이스를 위한 명령과 여섯 개의 인터럽트 처리를 위한 모드를 가지고 있고, 3단 파이프라인을 통해 한 사이클 당 하나의 명령어를 처리할 수 있다. 표 3은 SMART7 마이크로프로세서의 설계 사양을 나타낸다.

2.3 재정의 가능 명령어 수준 시뮬레이터

SoC 설계 기법으로 대표되는 최근 반도체 설계 동향은 복잡하고 방대한 시스템을 단축된 기간 안에 칩으로 구현 개발하는데 주력하고 있다. 설계자들은 고도로 전문화된 CAD 툴을 활용하여 게이트 이하 하위 수준은

표 3. SMART7 마이크로프로세서 설계 사양
Table 3. A design specification of the SMART7 Micropro -cessor.

구분	설계 사양
비트	32비트 RISC 마이크로프로세서 (32bit Data Bus, 32bit Address Bus)
Endian	Big and Little Endian operating modes
파이프라인	3단 (Fetch - Decode - Execution)
인터럽트	실시간 처리를 위한 Fast 인터럽트 Response
메모리	가상 메모리 시스템 지원
Software 개발도구	Excellent high-level Language support
레지스터	37개의 Banked Register 구조
명령어 수	11가지 유형의 34개의 명령어로 구성
Cache/MMU	Cache와 MMU 지원
코프로세서	명령어를 이용 코프로세서와 인터페이스 지원

설계 자동화에 의존하고 대신 알고리즘 및 동작 수준의 상위 수준 단계에 설계 초점을 둠으로써 전체적인 개발 기간이 단축된다. 따라서 상위 수준 기능 검증의 중요성이 매우 커진다. 또한 SoC는 코어 및 주변 장치와 연동하는 컴파일러 및 시스템 소프트웨어에 대한 의존성이 매우 크다. 반면 소프트웨어는 하드웨어 설계가 종료된 시점에서 시작 가능한 종속성이 매우 강하므로 소프트웨어/하드웨어 동시 설계가 매우 어렵고 이것이 전체적인 개발 기간지연의 중요한 요인으로 작용하였다. 반면 근래에는 이를 해결하기 위한 Co-design 기법이 주목 받고 있는데 이를 위해서는 효율적인 상위 수준 시뮬레이터 개발이 중요하다. 그래서 본 검증 환경에서는 재정의 가능 명령어 수준 시뮬레이터를 참조 모델로 사용하여 검증 결과의 신뢰성을 높이는데 이용하였다.

기존의 명령어 수준 시뮬레이터는 전용 개발 방법이 없이 C/C++과 같은 범용 프로그래밍 언어를 이용하여 직접 설계하거나 HDL등을 이용하였다. 그러나 C/C++과 같은 범용 프로그래밍 언어는 속도 면에서는 탁월하나 모델링의 체계성이 없기 때문에 설계 변경이 매우 어렵다. 그리고 HDL을 이용한 것은 설계 변경이 용이하나 수행 속도가 느리다. 따라서 본 검증 환경에서 사용된 재정의 가능 명령어 수준 시뮬레이터는 이러한 단점을 절충한 새로운 형태의 명령어 수준 시뮬레이터로 이를 상위 수준 기능 검증용 준거 모델로서 채택하였다.

본 재정의 가능한 명령어 수준 시뮬레이터는 명령어 셋을 기반으로 하는 임의의 파이프라인 구조 머신의 명령어 수준 시뮬레이션을 클락 사이클 기반으로 수행한다.

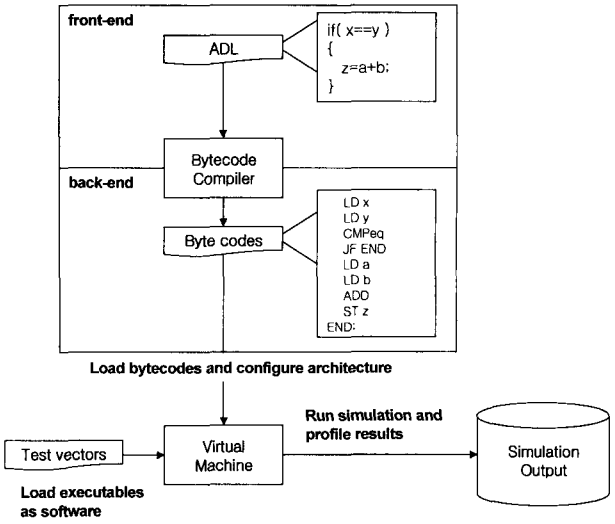

그림 7. 명령어 수준 시뮬레이션 절차
Fig. 7. An implementation flow of instruction level simulator.

본 시뮬레이터는 크게 바이트코드 컴파일러와 가상머신으로 이루어졌으며 HDL과 유사한 형태로 자체 정의한 ADL(Architecture Description Language)을 지원한다. 바이트코드 컴파일러는 ADL을 컴파일하여 바이트코드를 생성하고 이를 가상머신에서 수행함으로써 시뮬레이션을 수행한다. 그림 7은 이러한 과정을 나타낸 것이다.

2.4 시뮬레이션 결과 비교

테스트 벡터 생성 부분에서 생성된 테스트 벡터를 이용하여 시뮬레이션을 수행한 검증 목적 마이크로프로세서와 참조 모델의 시뮬레이션 결과를Hardware Result Compare 부분이 종합하여 결과를 보여주게 된다. 만일 각각에 해당하는 결과 값들이 다를 경우 에러 메시지와 함께, report file이 생성된다. 이때 생성되는 output 파일은 bic라는 확장자를 가지게 되는데, 이 report file은 내부에 오류가 발생된 위치를 표시하게 된다. 이에 표시된 오류 위치와 시뮬레이션 결과 파형을 이용하여 오류가 발생한 부분을 쉽게 찾을 수 있다.

2.5 전체적인 검증 과정

본 논문에서 제안한 기능 검증 환경은 그림 8에서 보이는 것과 같은 과정을 통해 이루어진다. 본 검증 방법은 바이어스 랜덤 벡터 생성기를 이용하여 랜덤 벡터 어셈블리 소스 파일을 생성하고 이를 전처리기와 소프

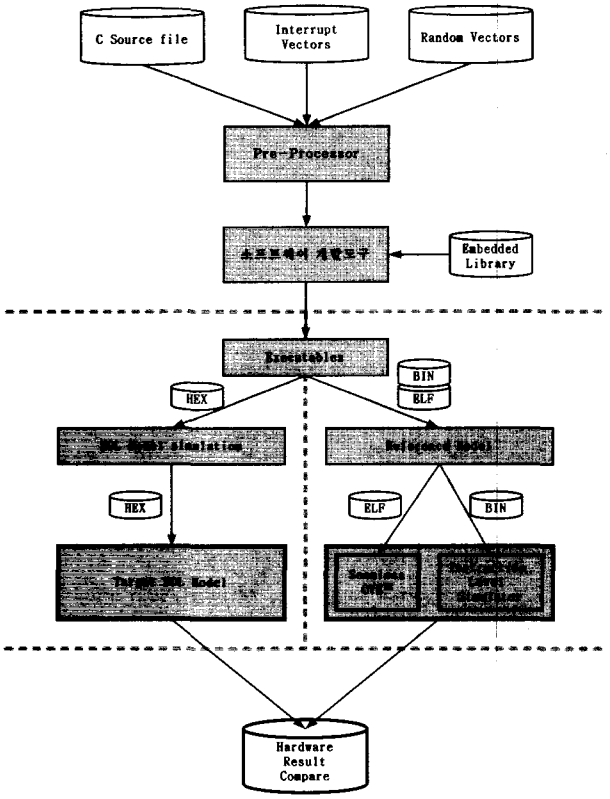

그림 8. 기능 검증 환경의 전체 검증 과정
Fig. 8. An implementation flow in the functional verification environment.

표 4. 기능 검증 환경에서 이루어지는 검증 대상
Table 4. A verification target in the functional verification environment.

검증대상	검증 방법
Register	생성된 테스트 벡터를 이용하여 목적 마이크로프로세서와 참조 모델들의 레지스터 값 비교
Memory	C 소스 파일을 통한 시뮬레이션의 메모리 결과 값을 비교
Code Trace	랜덤 테스트 벡터와 C 소스파일을 순차적으로 수행해가면서 Code의 위치 파악

트웨어 개발 도구를 이용하여 검증 목적 마이크로프로세서의 시뮬레이션에 필요한 워크로드를 생성한다. 이렇게 생성된 워크로드를 검증 목적 마이크로프로세서와 참조모델에서 수행하여 결과를 얻게 되는데, 각각에서 생성된 시뮬레이션 결과를 마지막 부분인 하드웨어 결과 비교 부분에서 비교하여 오류 여부를 판명하게 된다. 이때 수행되는 검증 대상은 표 4에 나타내었다.

III. 결과 및 고찰

제안한 기능 검증 환경의 단계별 처리 속도는 표 5와 같다. 전체 기능 검증 처리 시간 중 목적 마이크로프로세서의 시뮬레이션을 제외한 나머지 과정이 차지하는 비율은 불과 3% 미만으로 참조 모델의 시뮬레이션이 초래하는 부담은 무시할 수 있을 정도로 극히 미미하다. 그리고 본 검증 환경은 과거에 사람이 직접 찾아야 했던 기능상의 오류를 소프트웨어를 이용하여 찾을 수 있도록 함으로서, 마이크로프로세서의 설계 시 가장 큰 비중을 차지하던 검증 시간을 단축할 수 있게 하여, 마이크로프로세서의 전체 설계 기간을 단축시켜주는 이점을 가지고 있다.

표 5. 기능 검증 환경의 단계 별 수행 속도
Table 5. An implementation speed of the functional verification environment at each level.

검증 과정	속도(IPS)	점유율(%)
랜덤 벡터 생성	12,000	1.2
목적 마이크로프로세서의 시뮬레이션	150	97.3
재정의 가능 명령어 수준 시뮬레이터의 시뮬레이션	22,000	0.7
결과 비교	18,000	0.8

표 6. SMART7 마이크로프로세서의 오류 유형의 분포
Table 6. The distribution of error patterns in the SMART7 Microprocessor.

명령어 유형	개수	하드웨어 유형	개수
예외 처리	7	상태 레지스터	6
다중 사이클 명령어	1	코드 버스	2
단일 사이클 명령어	2	종속성 검사	1
포괄적인 경우	1	기타	2
총합	**11**	**총합**	**11**

SMART7 마이크로프로세서는 본 논문에서 제안한 검증 환경에서 검증을 수행하기 전에 이미 HDL 모델에 대해 1차 수준인 단일 명령어에 대한 기능 검증은 모든 경우에 대해 완벽하게 수행하였으며2차 수준인 소수 명령어 조합의 경우, 동작 유형별로 분류되어 구성 명령어 총합이 10만개인 워크로드에 대해 검증을 완료하였다. 그러나 이후 대략 36만개의 명령어를 이용한 랜덤 벡터 기능 검증을 수행한 결과 총 11개의 오류가 추가로 관찰되었다. 발견된 오류 유형의 분포는 표 6과 같다.

명령어 유형 중 가장 오류가 빈번한 경우는 예외 처리 과정이며 특히 명령어 비트 필드의 비정형성에 의해 비정의 명령어를 인식하지 못하는 오류가 빈번하였다. 하드웨어 유형 중 가장 오류가 빈번한 부분은 상태 래

지스터로서 예외 처리 과정과 맞물린 특권 모드 비트는 물론 일반 ALU 명령어와 관련된 상태 비트 변환 오류도 관찰되었다. 또한 다중 바이트 즉치 저장 명령어, LDC 계열 명령어와 명령어 패치의 공유에 의한 명령어 버스 오류 발생률도 높은 수치를 보였다.

랜덤 벡터에 의해 검출된 오류 유형의 특성은 오류 발생 빈도가 높은 영역이 상태 레지스터나 명령어 버스처럼 서로 다른 명령어에 의한 공유가 빈번하다는 점이다. 또한 이러한 오류는 단일 명령어가 아닌 명령어 조합에 의해서만 발견되나 명령어 조합이 2개 이상만 되어도 모든 경우를 대처하기에 워크로드 용량이 너무 크다. 따라서 앞의 결과에서 볼 수 있듯이 본 검증 환경은 랜덤 벡터를 이용하여 유형별로 최적화된 2차 수준 워크로드가 수용하지 못한 오류 검출을 효율적으로 보완함으로써 마이크로프로세서의 기능 검증의 효율성을 극대화하였다.

IV. 결 론

본 논문에서는 마이크로프로세서의 기능 검증을 위한 효율적인 기능 검증 환경을 제안하였다. 본 검증 환경은 테스트 벡터 생성부분, 시뮬레이션 부분, 그리고 결과 비교 부분으로 나누어져 있으며, 테스트 벡터 생성부분에서 생성된 랜덤 테스트 벡터를 이용하여 검증 목적 마이크로프로세서와 참조 모델의 시뮬레이션 결과를 비교함으로써 오류를 검출한다. 본 검증 환경의 성능을 측정한 결과 HDL 시뮬레이션을 제외한 나머지 과정이 차지하는 비율은 3% 미만으로 참조 모델의 시뮬레이션이 초래하는 부담은 무시할 수 있을 정도로 극히 미미하였다. 검증 목적 마이크로프로세서로 사용된 SMART7 마이크로프로세서의 검증 결과 랜덤 테스트 벡터를 사용하지 않고 검증했을 때 발견되지 않았던 오류가 11개 발견되었다. 이로 인해, 랜덤 테스트 벡터는 유형별로 최적화된 워크로드가 수용하지 못한 오류 검출을 효율적으로 보완함으로써 기능 검증의 효율성을 극대화 한다는 것을 알 수 있었다. 프로세싱 코어를 포함하는 SoC 시스템에서 마이크로프로세서의 기능 검증은 집적회로가 복잡해지고 설계기간과 설계 비용이 증가하면서 매우 중요해진 시점에서 본 기능 검증 환경을 이용하여 마이크로프로세서의 기능 검증을 수행할 경우, 좀더 빠른 시간과 적은 비용을 들여 기능 검증을 수행할 수 있다.

참 고 문 헌

[1] 권오현, 양훈모, 이문기, "마이크로프로세서 기능 검증을 위한 바이어스 랜덤 벡터 생성기 설계," 대한전자공학회 하계종합학술대회 Vol.25, No.1, pp.121~124, June 2002.

[2] C. Pixley, N. Strader, W. Bruce, J. Park, M. Kaufmann, K. Shultz, M. Burns, J. Kumar, J. Yuan, and J. Nguyen, "Commercial Design Verification : Methodology and Tools," Proc. IEEE Int. Test Conf., pp.839~848, 1996.

[3] P.J. Windley, "Formal modeling and verification of microprocessors," IEEE Transactions on Computers, Vol. 44, No. 1, pp.54~72, Jan. 1995.

[4] M. Kantrowitz and L.M. Noack, "I'm Done Simulating : Now What? Verification Coverage Analysis and Correctness Checking of the DEC chip 21164 Alpha Microprocessor," Proc. Design Automation Conf.,pp.325~330, 1996.

[5] 기안도, "단일 칩 시스템 설계검증을 위한 가상프로토타이핑," 대한전자공학회 전자공학회지, 제30권, 제9호, pp.59~69, Sep. 2003.

[6] Ta-Chung Chang, "A Biased Random Instruction Generation Environment for Architectural Verification of Pipelined Processor," in Journal of Electronic Testing : Theory and Applications 16, pp.13~27, 2000.

[7] M.S. Abadir, J. Ferguson, and T.E. Kirkland, "Logic Design Verification via Test Generation," IEEE Trans. Computer-Aided Design, Vol.7, No.1, pp.138~148, Jan. 1988.

[8] H. Iwashita, T. Nakata, and F. Hirose, "Integ-rated Design and Test Assistance for Pipeline Controllers,"IEICE Trans. Information and Systems, Vol.E76-D, No.7, pp.747~754, 1993.

[9] D. Moundanos, J.A. Abraham, and Y.V. Hoskote, "Abstraction Techniques for Validation Coverage Analysis and Test Generation," IEEE Trans. Computers, Vol.47, No.1, pp.2~13, Jan. 1998.

[10] J. Freeman, R. Duerden, C. Taylor, and M. Miller, "The 68060 Microprocessor Function Design and Verification Methodology," Proc. On-Chip Systems Design Conf., pp.10.1~10.14, 1995.

저 자 소 개

권 오 현(정회원)
2002년 울산대학교 전자공학과 공학사.
2004년 연세대학교 전자공학과 공학석사.
현재 삼성전자 디지털미디어연구소 Home Solution Lab. 연구원
<주관심분야: SoC, FPGA 기능 검증, 마이크로프로세서 구조 및 설계>

이 문 기(종신회원)
1965년 연세대학교 전기공학과 공학사.
1967년 연세대학교 전자공학과 공학석사.
1973년 연세대학교 전자공학과 공학박사.
1980년 Oklahoma Univ. 전자공학과 공학박사.
현재 연세대학교 전기전자공학과 교수
<주관심분야: SoC, 마이크로프로세서 구조 및 설계, 지문인식 칩 설계>