논문 2010-47SD-4-8

Core-A를 위한 효율적인 On-Chip Debugger 설계 및 검증

(Design and Verification of Efficient On-Chip Debugger for Core-A)

허 경 철*, 박 형 배*, 정 승 표*, 박 주 성**

(Jingzhe Xu, Hyungbae Park, Seungpyo Jung, and Jusung Park)

요 약

최근 SoC 가 주목받으면서 검증이 더욱 중요해졌다. SoC 설계 추세는 구조 및 RTL(Register Transistor Logic) 레벨의 HW(Hardware) 설계 및 내장형 프로세서에서 수행 될 SW(Software) 개발을 동시에 진행하는 HW/SW 통합 설계이다. 테크놀로지가 DSM(Deep-Submicron)으로 가면서 SoC 내부 상태를 확인하는 것은 매우 어려운 일이 되었다. 이와 같은 이유 때문에 SoC 디버거는 매우 어려운 분야이며 디버깅에 매우 많은 시간이 소모된다. 즉 신뢰성이 있는 디버거 개발이 필요하다. 본 논문에서는 JTAG을 기반으로 하는 하드웨어 디버거 OCD를 개발하였다. OCD는 Core-A를 대상으로 하여 개발 된 것이다. 개발된 OCD는 Core-A에 내장하여 SW 디버거와 연동하여 검증까지 마치고 디버거로서의 기능 및 신뢰성을 확인하였다. Core-A에 내장한 OCD는 약 14.7%의 오버헤드를 보이며 OCD의 2% gate count를 차지하는 DCU를 수정함으로써 다른 프로세서에도 쉽게 적용할 수 있는 디버거 유닛으로 사용할 수 있다.

Abstract

Nowadays, the SoC is watched by all over the world with interest. The design trend of the SoC is hardware and software co-design which includes the design of hardware structure in RTL level and the development of embedded software. Also the technology is toward deep-submicron and the observability of the SoC's internal state is not easy. Because of the above reasons, the SoC debug is very difficult and time-consuming. So we need a reliable debugger to find the bugs in the SoC and embedded software. In this paper, we developed a hardware debugger named OCD. It is based on IEEE 1140.1 JTAG standard. In order to verify the operation of OCD, it is integrated into the 32bit RISC processor - Core-A (Core-A is the unique embedded processor designed by Korea) and is tested by interconnecting with software debugger. When embedding the OCD in Core-A, there is 14.7% gate count overhead. We can modify the DCU which occupies 2% gate count in OCD to adapt with other processors as a debugger.

Keywords: 디버거, JTAG, halt-mode debugging, Core-A

I. 서 론

최근 반도체 설계 및 공정기술의 발달로 인하여 SoC(System on Chip)가 보편화되고 있다. SoC에서 가장 핵심적인 부분은 내장형 프로세서이다. 그 이유는 수많은 SoC칩이 프로세서를 내장하고 있고 프로세서가 두뇌 역할을 하기 때문이다. 내장형 프로세서의 디버거는 프로세서뿐만 아니라 시스템에 내장된 주변장치도 디버깅 및 제어할 수가 있다. 그러므로 SoC 를 디버깅함에 있어서 프로세서를 디버깅할 수 있는 디버거가 매우 중요하다.

요즘의 SoC 설계 추세는 구조 및 RTL(Register Transistor Logic) 레벨의 HW(Hardware) 설계 및 내장형 프로세서에서 수행 될 SW(Software) 개발을 동시에 진행하는 HW/SW 통합 설계의 과정으로 이루어진다. HW/SW 통합 설계로 인하여 time to market이 짧아지는 것은 사실이다. 그렇지만 설계된 고집적도의 SoC와 SoC의 성능을 최대로 발휘하여 효율적으로 구

* 학생회원, ** 평생회원, 부산대학교 전자전기공학과
(School of Electrical Engineering, Pusan National University)

※ 이 논문은 부산대학교 자유과제 학술연구비에 의하여 연구되었으며, 특허청 및 삼성전기의 지원으로 이루어졌습니다.
접수일자: 2009년12월3일, 수정완료일: 2010년3월12일

동시키는 어플리케이션(application) 검증에 대한 요구가 더욱 높아져가고 있다. 즉 SoC를 디버깅하는데 소모되는 비용과 시간은 점점 많아지고 있는 것이다. 이런 원인 때문에 SoC를 설계함에 있어서 보다 빠르고 정확한 디버거를 구현하는 것이 중요한 문제로 부각되고 있다.

대부분 SoC에서 포함하고 있는 프로세서는 반도체 기술이 발달함에 따라서 동작 주파수가 높아지고 따라서 디버깅은 더욱더 어렵게 되었다. 이와 같은 기술의 발전과 고성능 디버거의 필요성에 의해서 디버거는 기존에 Logic Analyzer와 In-Circuit Emulation 등과 같이 프로세서 외부에서 디버깅 기능을 구현하는 간접적인 접근 방식에서, 지금은 프로세서 내부에 디버깅 기능을 담당하는 로직을 내장하여 디버깅 환경을 구축하는 직접적인 접근 방식으로 발전하였다. 이와 같이 프로세서 내부에 내장되어서 프로세서와 연동하여 동작하면서 다양한 디버깅 기능을 지원하는 블록을 OCD(On-Chip Debugger)라 한다[1].

칩 개발 툴(EDA: Electric Design Automation)과 설계 기술의 발전으로 고성능의 칩 설계가 가능해 짐에 따라 막대한 자본력을 가지고 있는 거대한 칩 벤더에서만 개발이 가능했던 프로세서 설계가 저 자본의 설계가들에게도 가능 하게 되었다. 특정 목적의 시스템을 설계할 경우에 불필요한 블록까지 포함하고 있는 상용 프로세서 IP(Intellectual Property)를 사용할 필요가 없이 시스템에 최적화된 프로세서를 자체적으로 설계하여서 내장할 수 있다. 그러나 독자적인 구조의 코어를 설계하였을 때 제일 먼저 부딪히는 문제가 디버거라고 할 수 있다. 오늘날 디버깅의 어려움으로 인해서 하드웨어 엔지니어 뿐만 아니라 소프트웨어엔지니어도 적절한 디버거 없이는 아무리 좋은 성능의 프로세서라고 할지라도 사용하려고 하지 않을 것이다[1].

특허청 사업을 통하여 한국형 임베디드(embedded) 프로세서인 Core-A 국산 프로세서가 개발되었다. 본 논문에서는 Core-A를 타깃으로 하는 OCD 설계구현 및 검증에 대하여 구체적으로 소개하고자 한다.

OCD는 BP(Break-Point), WP(Watch-Point) 지정 및 감지, 타깃 프로세서 정지 및 재개, 레지스터/메모리 읽기 및 쓰기, single-step 등 디버깅 기능을 지원하며 SW 디버거와 연동되어 타깃을 디버깅할 수 있는 HW 디버거이다. BP는 해당 명령어의 실행 결과가 업데이트 되지 않는 것이고 WP는 해당 명령어의 실행 결과가 업데이트 되는 것이다.

본 논문의 구성은 다음과 같다. Ⅱ장에서는 OCD를 포함하는 전반적인 디버그 시스템에 대하여 다루고 Ⅲ장에서는 제안된 OCD 에 대하여 소개한다. Ⅳ장에서는 OCD 구현에 관하여 설명하고 Ⅴ장에서는 검증 및 결과에 대해 기술하고 마지막 Ⅵ장에서는 결론을 맺는다.

Ⅱ. 디버그 시스템

프로세서를 내장하는 시스템을 디버깅함에 있어서 OCD만으로 타깃을 디버깅할 수는 없으며 OCD를 포함하는 전반적인 디버그 시스템이 구축되어 있어야 된다. 디버그 시스템은 아래의 그림 1과 같이 OCD를 내장하게 되는 타깃 시스템(Core-A로 구성된 시스템), 타깃을 디버깅하고 상태를 확인할 수 있는 SW 디버거 및 OCD와 SW 디버거를 연동하는 Emulator Board로 구성되어 있다.

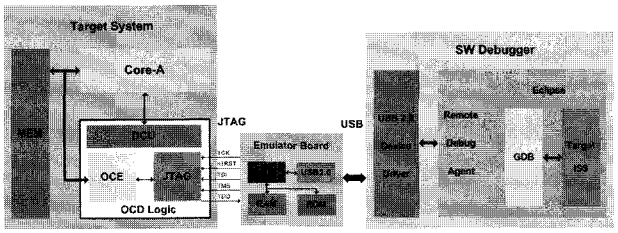


그림 1. 디버그 시스템
Fig. 1. Debug system.

1. Core-A

Core-A 프로세서는 RISC(Reduced Instruction Set Computer) 타입, Harvard architecture, 5단 파이프라인 구조의 32-bit 프로세서이면서 간단한 하드웨어 구조를 통해 적은 게이트 카운트(gate count)를 가지면서도 code density 및 효율적인 DSP(Digital Signal Processing) 프로그램의 처리를 위한 구조로 되어 있다. FPU(Floating Point Unit)와 같은 Coprocessor를 위한 인터페이스를 가지고 있으며 아래의 그림 2와 같이 Core-B Lite, AMBA, Wishbone과 같은 On-Chip High-Speed Bus를 통해서 SoC 내의 다른 IP들과 연결된다. 여러 가지 버스와 인터페이스 할 수 있는 wrapper들이 IP로 제공되고 있다. 그리고 Core-A도 완전 합성 가능한 soft-IP로 제공된다[2].

Core-A는 Fetch, Decode, Execute, Memory-access,

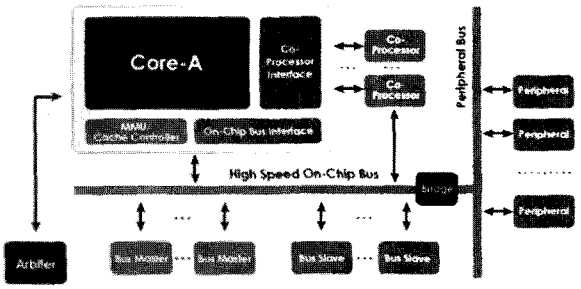


그림 2. Core-A SoC 구조
Fig. 2. Core-A SoC architecutre.

Write-back의 5단 파이프라인으로 구성되어 있고 Harvard 구조를 가지고 있으며 0.18㎛ 공정에서 약 200~250MHz급의 동작 속도를 가진다.

2. SW 디버거

SW 디버거는 ISS(Instruction Set Simulator)를 이용한 호스트 컴퓨터에서 동작하는 로컬 디버깅(local debugging) 모드와 소프트웨어 디버거를 실제 타깃 시스템에 연결해서 OCD블록을 제어하면서 디버깅하는 원격 디버깅(remote debugging) 기능을 지원한다.

OCD와 연동하여 디버깅할 때에는 SW 디버거가 원격 디버깅을 수행하여 타깃 시스템에서 수행되는 프로그램의 동작을 제어하는 기능을 갖추고 있으며, 프로그램의 소스코드(C/C++, 어셈블리어)와 변수, 시스템의 메모리, 레지스터 등을 손쉽게 볼 수 있는 GUI(Graphic User Interface)를 지원한다.

SW 디버거는 크게 타깃 지정 기능을 이용하여 다양한 시스템을 지원하는 범용 SW 디버거와 특정 시스템에 특화된 내장형 SW 디버거로 구분 할 수 있다. 본 논문의 검증에는 범용 SW 디버거인 GNU의 GDB를 사용하였다.

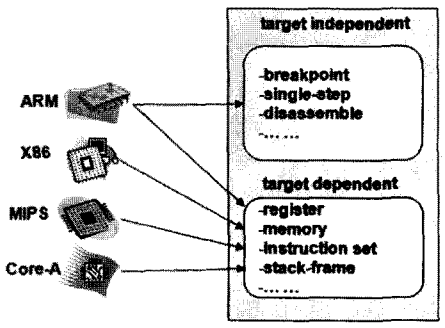


그림 3. GDB 구현
Fig. 3. Implementation of GDB.

GDB에는 위의 그림 3과 같이 타깃에 종속적인 파트와 독립적인 파트가 있다. GDB의 타깃에 종속적인 파트에 레지스터, 메모리, 명령어와 같은 대상 프로세서 아키텍처와 관련된 정보를 추가하면 해당 프로세서용 SW 디버거로 구축할 수 있다. 즉 GDB를 이용한 디버거는 새로 개발한 프로세서용 SW 디버거 설계에 적합하다. 그리하여 본 논문에서 사용한 Core-A SW 디버거는 GDB에 Core-A를 추가하고 Core-A ISS 및 toolchain(complier, assembler, linker 등을 포함한다.)을 연동하여 구성하였다. 그리고 GUI는 프리웨어인 Eclipse를 사용하였다.

3. Emulator Board

OCD는 JTAG 기반으로 설계되었기에 JTAG 포트를 통하여 통신하여야 한다. 하지만 타깃 시스템은 JTAG 포트가 없기에 때문에 서로 다른 인터페이스를 연결해 주는 장비 즉 Emulator Board가 필요하게 된다. Emulator Board는 소프트웨어 디버거를 원격 디버깅 모드로 동작 할 때 소프트웨어 디버거와 타깃 시스템에 내장된 OCD를 연결해주는 역할을 한다. 그리고 Emulator Board는 디버깅에 필요한 정보를 얻기 위해서 OCD 블록을 제어하고 디버깅 정보를 SW 디버거로 전달하는 방식으로 동작한다. 프로세서의 데이터 처리 양과 속도가 높아짐으로 인해서 디버깅해야 할 데이터의 양 또한 증가 하였다. 고속 디버깅을 위해서 Emulator Board의 처리 속도도 또한 높아 져야 할 필요가 있게 되었다. 이와 같은 이유로 인해서 Emulator Board는 SW 디버거와는 고속으로 통신하기 위해서 USB2.0 프로토콜을 사용하였고 OCD 블록을 고속으로 제어하기 위해서 고성능의 마이크로프로세서를 사용하였다.

USB 케이블로 SW 디버거가 수행되는 호스트 컴퓨터와 Emulator Board를 연결하고 JTAG cable로 Emulator Board와 OCD가 내장된 타깃 시스템을 연결하면 그림 1과 같은 전반적인 디버그 시스템이 구축된다. 디버깅을 시작하면 호스트 컴퓨터에서 수행되는 SW 디버거는 타깃 프로세서에서 실행 할 프로그램 코드를 편집, 컴파일하고 타깃 시스템에 내장된 OCD를 제어하여 디버깅 과정을 제어하며 Eclipse GUI를 통하여 상태 및 결과를 일목요연하게 사용자한테 보여준다.

Ⅲ. 제안된 On-Chip Debugger

1. 메커니즘

OCD는 프로세서를 디버깅하기 위하여 시스템 모드와 디버그 모드를 정의하였다. 시스템 모드는 프로세서가 시스템 클록(Clock)에 의하여 동작하여 디버거의 간섭이 없이 자체적으로 기존의 프로그램을 수행하는 모드이다. 반대로 디버그 모드는 프로세서가 SW 디버거의 제어 하에 있는 OCD와 연동되어 디버그 클록으로 동작하는 모드이다.

제안된 OCD가 프로세서를 제어하는 메커니즘은 아래의 그림 4에서 자세히 보여주고 있다. 사용자가 프로세서를 원하는 프로그램 실행 위치에서 디버깅하기 위하여 PC(Program Counter), 명령어, 데이터 주소, 데이터 값, 프로세서 상태 등을 이용하여 BP 혹은 WP를 지정하면 OCD는 제어 메커니즘에 따라 동작한다.

프로세서가 시스템 모드에서 프로그램을 수행할 때 OCD는 사용자가 원하는 BP 혹은 WP가 인가되었는지를 감지하고 종류를 확인하게 된다. BP이면 유효성을

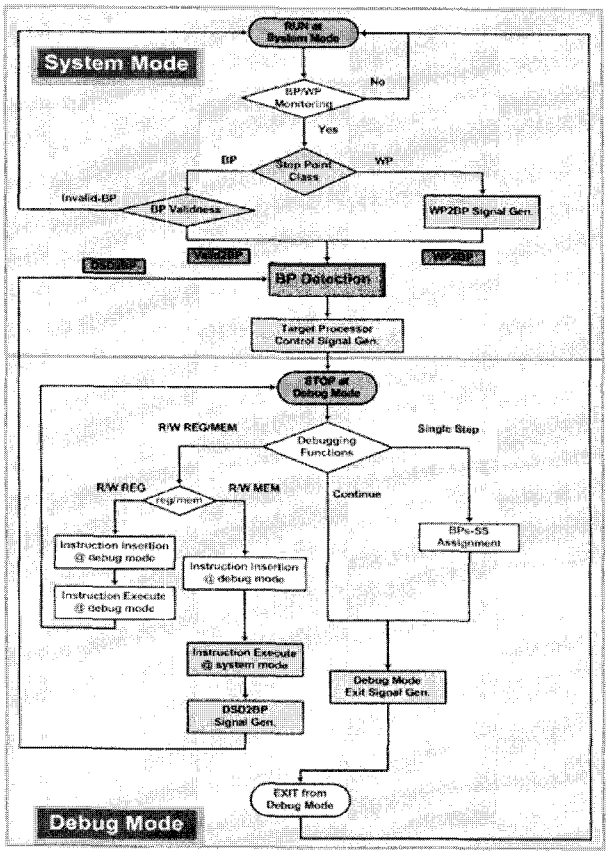


그림 4. 제안된 디버깅 메커니즘
Fig. 4. Debugging mechanism.

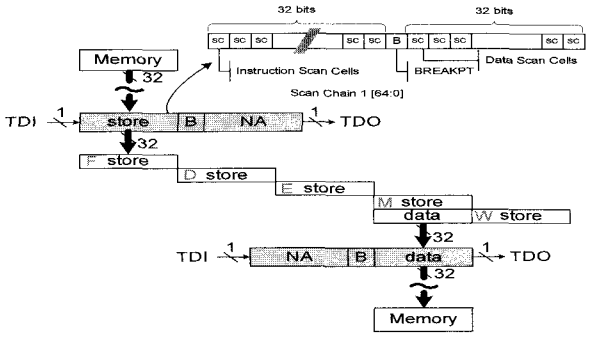


그림 5. 명령어 삽입 및 실행 과정
Fig. 5. Procedure of instruction insertion & executing.

확인하여 Valid2BP를 생성한다. BP는 branch, jump 명령어, 예외처리(exception) 등으로 무효화될 수 있다. WP이면 해당 명령어의 다음 명령어에 BP를 지정하였을 때 인가되는 BP신호와 같은 WP2BP(Watch-Point to Break-Point)를 생성한다. 다시 말하여 WP에 해당하는 명령어의 다음 명령어에 BP 지정된 것과 같기에 WP된 명령어는 실행되어 결과가 업데이트되고 그 다음 명령어는 실행되지 않는다. 이와 같이 제안한 디버깅 메커니즘은 BP, WP를 같은 BP 감지 및 제어 메커니즘으로 수행할 수 있게 된다. 만약 BP, WP가 감지되지 않으면 프로세서는 계속하여 시스템 모드에서 프로그램을 실행한다. BP Detection 과정을 거치면 OCD는 프로세서의 동작을 제어하는 신호를 생성하여 프로세서로 하여금 디버그 모드로 진입하게 한다.

디버그 모드에서 OCD는 여러 가지 디버깅 기능에 따라 프로세서를 제어한다. OCD가 지원하는 디버깅 기능에는 BP/WP 지정 및 감지, 타깃 프로세서 정지 및 재개, 레지스터/메모리 읽기 및 쓰기, single-step 등이 있다.

레지스터 읽기/쓰기와 메모리 읽기/쓰기는 프로세서 파이프라인에 명령어를 삽입하고 실행시키는 방법으로 수행된다. 명령어 삽입 및 실행 과정은 위의 그림 5에서 자세히 보여주고 있다. 먼저 명령어 버스에 연결되는 instruction scan-chain을 통하여 명령어를 삽입하고 다음엔 삽입된 명령어를 파이프라인 수행 단계에 따라 디버그 클록을 인가함으로써 실행시킨다. 마지막으로 버스에 실린 명령어 실행 결과를 데이터 버스에 연결되는 data scan-chain에 업데이트하여 출력하는 형태로 구현된다. 하나의 예로 레지스터 읽기는 다음과 같은 과정을 거친다.

- store register to memory 명령어를 instruction 스

캔 체인(scanchain)에 삽입한다.

- 스캔 체인에 실린 store명령어를 파이프라인에 업데이트한다.
- 디버그 클록을 인가하면 store 명령어는 파이프라인 단계에 따라 수행된다.
- 명령어가 실행되면 Memory-access 단계 시에 실행결과 즉 레지스터 값이 데이터 버스에 실리게 된다.
- 데이터 버스의 값을 data 스캔 체인으로 캡쳐하고 출력한다.

스캔 체인으로부터 출력된 값은 읽으려는 레지스터 값이다. 이 과정에서 스캔 체인은 메모리와 프로세서사이의 버스 연결을 차단하기 때문에 store 명령어가 실행되었다 하더라도 수행 결과는 메모리로 업데이트 되지 않고 디버깅하는 사용자가 원하지 않는 상황을 초래하지 않는다. 또한 이 원인 때문에 레지스터 읽기/쓰기와 메모리 읽기/쓰기의 수행과정이 조금 다르게 된다. 레지스터 읽기/쓰기의 명령어 삽입과 수행은 모두 디버그 모드에서 실행되지만 메모리의 경우에는 명령어 수행이 시스템 모드에서 실행된다. 시스템 모드에서 명령어가 실행되면 메모리와 프로세서가 연결되기 때문에 메모리 액세스(access)가 가능하여 메모리 읽기/쓰기가 가능하게 된다. 때문에 메모리 읽기/쓰기를 위해서 OCD는 프로세서를 디버그 모드에서 시스템 모드로 탈출시켜 명령어를 실행시키고 다시 디버그 모드로 다시 진입시켜야 한다. 이를 위하여 생성하는 신호가 DSD2BP(Debug-System-Debug mode change to Break-Point)신호이다.

프로세서로 하여금 디버그 모드에서 탈출하여 원래의 프로그램을 재개시키는 기능과 한 스텝(c언어에서는 하나의 statement, assembler에서는 하나의 명령어)씩 실행하는 싱글스텝 기능은 BP 설정과 디버그 모드 탈출로 수행된다. 디버그 모드 탈출 시에는 원하는 PC의 값에 해당하는 명령어부터 실행되게끔 프로세서의 파이프라인을 제어한다.

이 모든 디버깅 기능의 수행을 위하여 OCD는 해당 기능에 따라 프로세서의 디버그 모드 진입, 탈출을 제어해야 할 뿐만 아니라 메모리 읽기/쓰기를 위한 디버그 모드 탈출-재진입도 제어하여야 한다.

제안한 OCD는 모든 디버깅 기능의 수행을 Valid2BP, WP2BP 및 DSD2BP 이 3가지 신호 생성으로 구현하고 3가지 신호는 하나의 BP 감지 블록으로 처리한다. 이렇게 함으로써 1bit의 파이프라인 제어신호 및 1bit의 디버그 상태신호로 프로세서를 제어할 수 있게 된다. 즉 제안한 OCD는 단지 2-bit의 신호로 프로세서를 제어할 수 있는 장점이 있을 뿐만 아니라 적은 게이트 카운트(gate count)로 구현된다.

2. 구조

제안한 OCD를 구현하기 위하여 구조적으로 아래의 그림 6과 같이 Emulator Board를 통하여 SW 디버거와의 통신을 제공하는 JTAG블록, BP 혹은 WP를 지정하는 OCE(On-Chip Emulator) 및 디버깅 기능을 제어하는 DCU(Debug Control Unit)으로 나뉘어져 있다. 매 블록의 자세한 내용 및 설계는 Ⅳ장에서 구체적으로 설명한다.

BIU(Bus Interface Unit)는 해당 버스(Local, AMBA, Core-B, Wishbone)에 따라 서로 다르지만 제안한 OCD는 BIU에 관계없이 프로세서에만 종속함으로 임의의 버스, BIU와 손쉽게 연결할 수 있다.

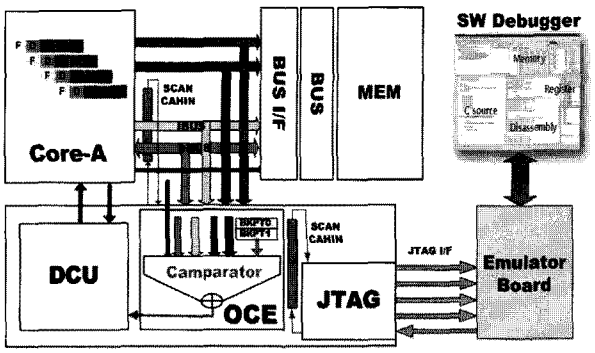


그림 6 OCD 블록도
Fig. 6. OCD block diagram.

Ⅳ. On-Chip Debugger 구현

본 논문에서 구현하는 OCD는 크게 시스템제어와 레지스터 및 메모리 제어기능을 지원한다. 시스템제어는 타깃에서 수행되고 있는 프로그램 흐름을 멈추게 하거나 다시 재개시키고, 싱글 스텝(single step)을 지원하는 기능이다. 이 기능은 OCD에서 제공하는 2개의 HW BP 혹은 WP의 지정 및 감지로 구현된다. 레지스터 및 메모리 제어 기능은 레지스터나 메모리 값을 읽고 변경 가능한 기능이다. 이런 디버깅 기능을 위하여 JTAG 블록, OCE, DCU는 체계적으로 연동되어 동작하고 있다. OCD를 이용하여 타깃을 디버깅할 때 데이터 흐름은

위의 그림 6에서 보이는 바와 같다. OCD는 JTAG 블록의 JTAG 포트를 통하여 Emulator Board와 연결되고 SW 디버거와 통신한다. SW 디버거에서 디버깅 명령을 내리면 Emulator Board가 그 명령어를 해석하고 JTAG 신호로 변환한다. 변환된 JTAG 신호는 OCD의 JTAG 블록으로 전달되고 JTAG은 스캔 체인 제어를 통하여 OCE에 BP/WP를 지정하거나 프로세서의 버스에 값을 인가한다. 이때 DCU는 OCE에서 생성되는 BP/WP 신호나 스캔 체인의 BREAKPT신호를 참조하여 해당 디버깅 기능에 따라 프로세서를 제어하게 되고 디버깅 결과는 최종적으로 스캔 체인을 통하여 SW 디버거에 전송된다. 아래에 매개 블록의 자세한 구현에 대하여 설명한다.

1. JTAG 블록

JTAG블록은 OCD와 외부의 다리작용을 하며 말 그대로 JTAG 표준을 따른다. JTAG(Joint Test Action Group)[4]은 IEEE 1149.1 "Standard Test Access Port and Boundary Scan Architecture" 표준이다. JTAG은 4개의 입력 포트 TDI(Test Data In), TMS(Test Mode Select), nTRST(not Test Reset) 및 TCK(Test Clock)와 1개의 출력 포트 TDO(Test Data Out)로 이루어져 있어 적은 핀으로 SW 디버거와 통신하여 HW 디버거를 제어할 수 있는 장점이 있다[4]. 때문에 OCD는 JTAG을 기반으로 하여 설계하였다.

설계된 JTAG 블록은 위의 그림 7과 같이 TAP(Test Access Port), TAP 제어기, 2개의 스캔 체인, 명령어

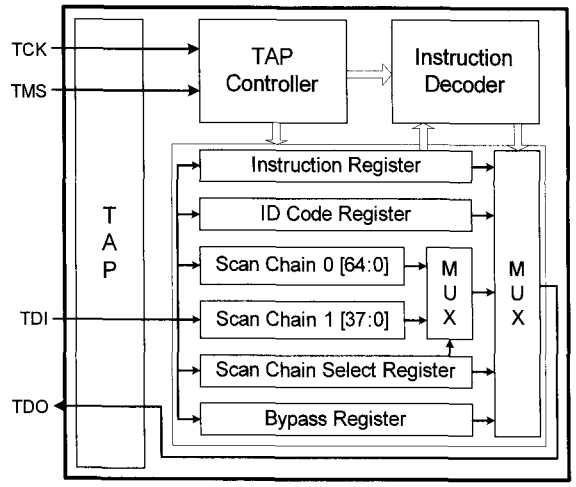


그림 7. JTAG 블록도
Fig. 7. JTAG block diagram.

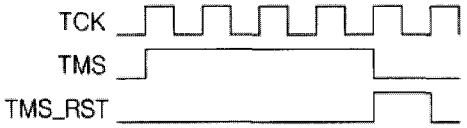


그림 8. TMS_RST 생성
Fig. 8. Generation of TMS_RST.

Instruction	IR value	register
INTEST	b'1100	Scan-chain0 or 1
EXTEST	b'0000	Scan-chain0 or 1
IDCODE	b'1110	IDCODE register
BYPASS	b'1111	BYPASS register
SCAN_N	b'0010	Scan-chain select register
RESTART	b'0100	BYPASS register

그림 9. JTAG 명령어
Fig. 9. JTAG instruction.

레지스터 및 여러 개의 특수목적 레지스터(register)로 구성되어 있다.

TAP은 5개의 포트를 지원하지만 구현된 TAP은 3개의 입력 포트 TDI, TMS, TCK와 1개의 출력 포트 TDO 즉 nTRST 포트를 제외한 4개의 포트로 구성되어 있다. nTRST는 JTAG 블록을 초기화하는 포트이지만 TAP 제어기에 nTRST와 같은 동작을 하는 TMS_RST신호를 구현하여 nTSRT 포트를 제거하였다. TMS_RST신호는 그림 8과 같이 TCK 4 사이클(cycle)동안 TMS를 인가하면 생성된다.

TAP 제어기는 JTAG 블록의 모든 동작을 제어하며 구현은 JTAG 표준 그대로 하였다. 스캔 체인 0은 65bit로, 프로세서의 32bit 명령어 버스, BREAKPT 신호 및 32bit 데이터 버스와 와 연결되어 있어 OCD와 프로세서간의 데이터 교환을 수행한다. 즉 스캔 체인 0을 통하여 OCD는 프로세서의 명령어 버스에 명령어를, 데이터 버스에 데이터를 인가할 수도 있고 데이터 버스에 실린 명령어의 수행 결과를 출력할 수도 있다. 이런 기능 때문에 스캔 체인 0은 디버그 모드에서만 동작하고 시스템 모드에서는 된다. 그리고 스캔 체인(scan chain) 1은 38bit로, OCE의 레지스터와 연결되어 있어 BP 혹은 WP를 지정하거나 디버그 상태 레지스터를 제어하는 역할을 한다.

구현된 JTAG 블록이 지원하는 JTAG 명령어와 해당 명령어에 따라 TDI, TDO사이에 연결되는 레지스터는 그림 9와 같다

SCAN_N 명령어는 스캔 체인 선택을 위한 명령어로

서 Scan Chain Select Register를 TDI와 TDO사이에 위치하게 하고 스캔 체인 0과 스캔 체인 1을 선택하는 기능을 수행한다. INTEST와 EXTEST 명령어는 선택된 스캔 체인에 값을 인가하는 기능을 수행하고 IDCODE와 BYPASS는 각각 IDCODE 레지스터와 BYPASS 레지스터를 TDI와 TDO사이에 위치하게 한다. RESTART 명령어는 프로세서를 디버그 모드에서 탈출하게 하는 명령이다.

2. OCE(On-Chip Emulator)

OCE는 BP 혹은 WP를 지정 및 감지하는 기능을 수행하고 DSR(Debug State Register)를 포함하고 있다. OCE는 아래의 그림 10과 같이 2개의 BP 레지스터 셋(set), DSR과 비교기로 구성되어 있다.

BP 레지스터 셋은 스캔 체인 1과 연결되어 BP 혹은 WP 지정이 가능하다. 시스템을 디버깅할 때 C/C++ 및 어셈블리 레벨에서 한 문장씩 수행하는 싱글 스텝, 프로세서의 PC값이 하나의 주소 포인트가 아닌, 일정한 범위의 주소영역에 진입하였을 때 디버그 모드로 진입하게 하는 BP 지정, 그리고 SW BP와 HW BP를 동시에 지정 가능하게 하는 기능이 매우 유용하다. 이런 기능들을 가능하게 하려면 BP 레지스터 셋 1개로는 불가능하고 적어도 2개 이상이어야 한다. 때문에 구현한 OCE에는 BP 레지스터 셋 2개를 설계하였다.

비교기는 프로세서의 데이터 및 주소 버스와 2개의 BP 레지스터 셋의 값을 비교하여 BP 혹은 WP를 감지한다. 생성된 BP/WP 신호는 디버깅 기능을 제어하는

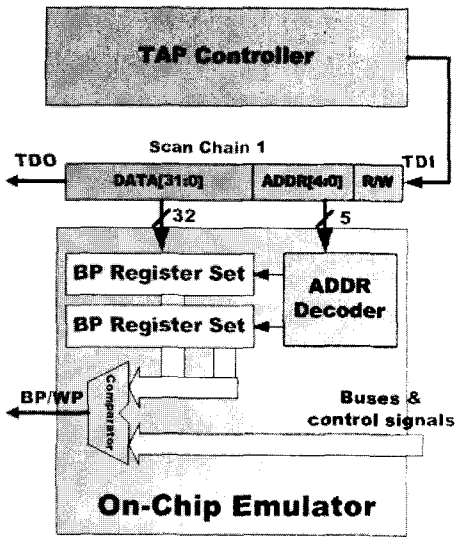


그림 10. OCE 블록도
Fig. 10. OCE block diagram.

그림 11. DSR 포맷
Fig. 11. Format of DSR

DCU로 전달된다.

DSR은 그림 11과 같이 3 bit로 구성되어 있다. DBGACK는 프로세서가 디버그 모드인지 시스템 모드인지를 나타내며 값이 1이면 디버그 모드이다. DBGRQ는 BP/WP 지정 없이 프로세서를 디버그 모드로 진입하게 하는 bit이다. SW 디버거로 DBGRQ 값을 1로 인가하면 외부에서 강제적으로 프로세서를 정지시킬 수 있다. 그리고 DBGSRC는 프로세서가 디버그 모드로 진입한 원인을 나타내는 bit이다. DBGSRC 값이 1이면 BP/WP 혹은 DBGRQ에 의하여 디버그 모드로 진입한 것을 나타내며 값이 0이면 스캔 체인 0의 BREAKPT 신호에 의하여 디버그 모드로 진입한 것을 표현한다.

3. DCU(Debug Control Unit)

DCU는 OCD의 핵심 블록이다. DCU는 프로세서를 디버그 모드로의 진입 및 탈출을 제어함으로써 OCD의 전반적인 디버깅 기능의 수행을 제어한다. OCE가 BP 혹은 WP를 감지하였을 때 프로세서가 디버그 모드에 진입하는 타이밍은 서로 다르다. 프로세서는 BP일 때, BP된 명령어가 Execute단계에 있을 때, 즉 명령어 실행결과가 업데이트 되지 않았을 때 디버그 모드로 진입한다. WP일 때에는 WP된 명령어가 실행을 끝마치면 디버그 모드로 진입하게 된다. 그리고 메모리 읽기/쓰기를 위한 디버그 모드 탈출-재진입할 때에도 BP, WP와 서로 다른 타이밍을 가진다. 즉 BP, WP 및 디버그 모드 탈출-재진입 이 3가지 경우에 대한 제어 모듈은 서로 다르게 된다. 하지만 본 논문에서 제안한 메커니즘은 WP 신호를 WP2BP신호로 변환하여 사용하고 디버그 모드 탈출-재진입도 DSD2BP신호를 생성하여 사용하기에 아래의 그림 12와 같이 BP에 대한 제어 모듈로만 모든 제어가 가능하다. 즉 BP를 제외한 나머지 2가지 경우에 대한 실질적인 제어 모듈만큼 gate count가 줄어든다.

제어를 위하여 DCU는 프로세서의 명령어가 유효함을 나타내는 IV(Instruction Valid), 명령어의 실행이 끝났음을 나타내는 IE(Instruction End), BP된 명령어의 앞 명령어가 브랜치 및 점프 명령어임을 나타내는 BR,

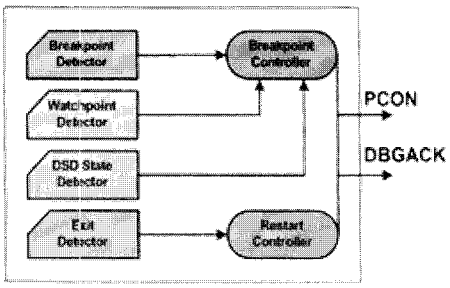


그림 12. DCU 기능 블록도
Fig. 12. Function block diagram of DCU.

멀티 사이클(multi-cycle) 명령어를 나타내는 MI (Multi-cycle Instruction) 및 EC(Exception) 등 5가지 신호를 참조한다. IV는 BP된 명령어와 BR 명령어의 유효성을 나타낸다. BP된 명령어가 유효하지 않으면 해당 명령어에 지정된 BP는 무시되고 디버그 모드로 진입하지 않는다. 또한 BR 명령어가 유효하면 BP된 명령어는 실행되지 않기 때문에 마찬가지로 BP가 무시되고 디버그 모드로 진입하지 않는다. DCU는 IE와 MI를 참조하여 프로세서의 파이프라인 단계를 추정할 수 있고 이로써 프로세서가 디버그 모드로 진입하는 타이밍을 제어한다. DCU는 EC를 참조하여 프로세서의 예외처리 진입과 디버그 모드 진입 타이밍 충돌을 피하고 예외처리를 보다 높은 우선순위를 가지게끔 구현하였다.

디버깅 동작을 제어하는 DCU는 OCD의 여러 블록 중 가장 적은 게이트 카운트를 차지하지만 OCD의 가장 중요한 블록이며 대상 프로세서에 가장 의존성이 높은 블록이다. 때문에 DCU를 프로세서의 파이프라인 제어에 맞게 수정하면 다른 RISC 타입의 프로세서에도 같은 메커니즘을 이용하여 손쉽게 OCD를 접목할 수 있다.

OCD의 제어에 의하여 BP의 경우에 프로세서가 디버그 모드 진입, 탈출하는 과정은 위의 그림 13에서 보여주고 있다. 먼저 JTAG을 통하여 스캔 체인 1을 제어함으로써 OCE에 원하는 BP 주소 "N"을 지정한다. 프로세서가 프로그램을 수행하다가 주소 버스에 "N"이란 값이 실리면 OCE 내부의 비교기가 BP를 감지하고 BREAKPOINT 신호를 1로 인가한다. BREAKPOINT 신호가 DCU에 전달되면 DCU는 IV, IE, BR, INT 신호를 참조하여 프로세서의 상태를 체크하고 BP된 명령어부터의 실행 결과가 업데이트 되지 않게 프로세서를 제어한다. 주소 버스에 "N+12"가 실리는 타이밍에 실행 클록을 시스템 클록에서 디버그 클록으로 변경하고 프로세서를 디버그 모드로 진입시킨다. 디버그 모드에서는 SW 디버거에서 원하는 임의의 디버깅 기능을 수행하다가 디버깅을 완료시키면 프로세서의 모든 레지스터 값을 복원시킨다. 그 다음에 프로세서는 디버그 모드 진입하기 전의 주소 "N"에 해당하는 명령어부터 실행하게 된다. 이러면 프로세서는 기존에 실행하던 프로그램의 흐름에 따라 수행되어 디버깅 과정이 수행 결과에 영향을 주지 않게 된다.

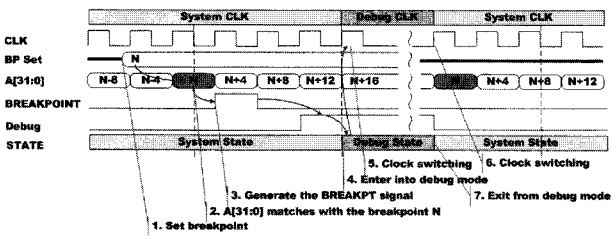


그림 13. 디버깅 타이밍도
Fig. 13. Debugging timing diagram.

V. 검증 및 결과

설계된 OCD를 검증하기 위해 본 논문에서는 아래의 그림 14에서 보이는 바와 같이 3 단계의 검증 과정 즉 행위수준 시뮬레이션 (functional simulation)을 통한 검증, PLI(Program Language Interface)를 이용한 SW 디버거 연동 검증 및 FPGA 레벨에서의 SW 디버거 연동 검증을 진행하였다. 행위수준 시뮬레이션을 이용한 검증은 속도는 느리지만 원하는 다양한 방식과 알고리즘을 소프트웨어적으로 적용할 수 있다는 이점이 있다. 반면에 FPGA를 이용한 검증은 실시간에 가까운 속도로 응용 프로그램들을 테스트할 수 있기에 타이밍에 민감한, 행위수준 시뮬레이션에서 검증하지 못하는 프로

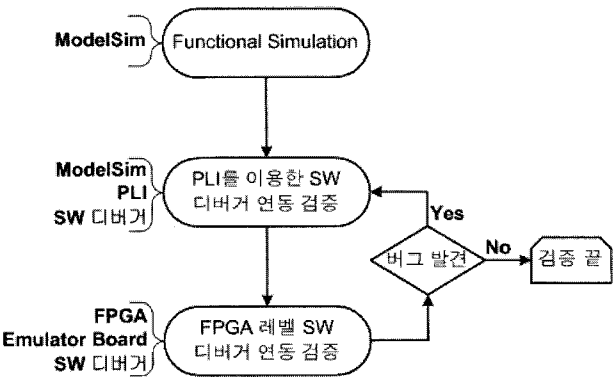


그림 14. OCD 검증 과정
Fig. 14. Procedure of OCD verification.

그램도 테스트할 수가 있다. 하지만 본 논문에서 구현한 OCD는 컴퓨터에서 수행되는 SW 디버거와 연동하여 동작하여야 하기에 FPGA 레벨 검증에서 오류가 발생한다 하더라도 원인 분석이 매우 어렵다. 때문에 검증 과정에 PLI를 이용한 SW 디버거 연동 검증을 추가하였다.

1. 행위수준 시뮬레이션 기능 검증

행위수준 시뮬레이션 검증은 프로세서의 단일 명령어, 조합 명령어 및 간단한 응용 알고리즘을 실행시키면서 진행하였다. 이 과정 중 OCD의 추가가 프로세서의 수행에 영향을 주는지를 먼저 확인하였다. 그 다음에 OCD의 매개 기능 블록별로 해당 기능을 수행하는지 검증하였다. JTAG 블록이 스캔 체인을 제어하는지, OCE에 BP/WP가 지정되는지, BP가 발생하였을 때 DCU가 프로세서를 정확하게 제어하는지 등을 검증하였다. 그리고 마지막엔 OCD의 모든 블록을 통합하여 레지스터 읽기/쓰기, 메모리 읽기/쓰기 등 모든 디버깅 기능을 검증하여 디버거가 정확하게 동작하는 것을 확인하였다. 검증한 디버깅 기능에는 BP/WP 지정 및 감지, 싱글 스텝, 레지스터/메모리 제어 및 디버그 모드 진입, 탈출 등이 있다.

행위수준 시뮬레이션 검증의 하나의 예로 OCE BP 레지스터 셋에 BP를 지정하는 디버깅 기능 검증 과정을 아래의 그림 15에서 보여주고 있다. 먼저 C 언어를 이용하여 OCD 제어 메커니즘을 모델링하고 이 프로그램으로 텍스트 벡터(Test Vector)를 생성하였다. 텍스트 벡터는 디버깅 기능에 따라 생성되는 TDI, TCK, TMS 3가지 신호로 구성되고 이 신호를 테스트 벤치(Test-bench)를 통하여 DUT(Design Under Test)에 인가하면 그림 15에서의 시뮬레이션 파형이 생성된다. 파형을 보면서 원하는 디버깅 기능이 정상적으로 동작하는지 확인하면서 모든 디버깅 기능을 검증하였다.

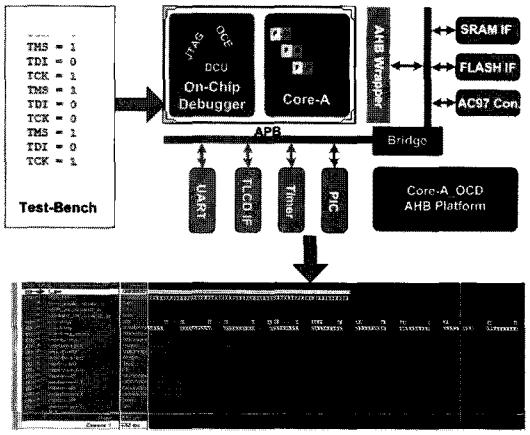


그림 15. 행위수준 시뮬레이션 검증
Fig. 15. Verification by functional simulation.

2. PLI를 이용한 SW 디버거 연동 검증

PLI를 통한 SW 디버거와 ModelSim 시뮬레이터 연동 검증하였다. PLI를 이용한 검증 환경은 아래의 그림 16과 같다. SW 디버거에서 PLI 연동을 위한 원격 디버깅 RDA를 선택하고 Core-A 테스트 벤치와 소켓 통신을 진행하면서 모델심과 연동 및 통신을 진행한다. 이렇게 연동되면 SW 디버거의 디버깅 명령어가 RDA_PLI를 통하여 테스트 벤치에 전달하게 되고 다시 TDI, TMS, TCK, nTRST 포트를 통하여 OCD를 내장한 Core-A에 전달된다. OCD는 위 4개 신호를 받아들여 해당 디버깅 기능을 수행하고 Core-A를 제어한다. 그리고 디버깅 기능에 따라 생성된 디버깅 결과는 다시 TDO를 통해 테스트 벤치, PLI를 거쳐 최종적으로 SW 디버거에 전달된다. SW 디버거는 디버깅 결과를 Eclipse GUI를 통하여 사용자에게 보여준다. 만약 SW 디버거에서 보여준 디버깅 결과가 사용자가 원하는 결과와 일치하지 않으면 시뮬레이터 파형을 통하여 원인 분석할 수 있다.

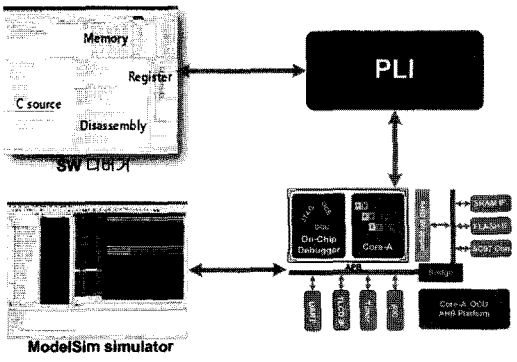


그림 16. PLI를 이용한 검증 환경
Fig. 16. Verification environment by PLI.

3. FPGA레벨 SW 디버거 연동 검증

행위수준 시뮬레이션 검증과 PLI를 이용한 검증은 모두 gate delay를 고려하지 않고 기능만 검증하게 된다. 그리고 실시간 검증이 아니기 때문에 추가적으로 FPGA 레벨에서의 검증이 필요하다.

설계된 OCD를 아래 그림 17의 타깃 시스템과 같이

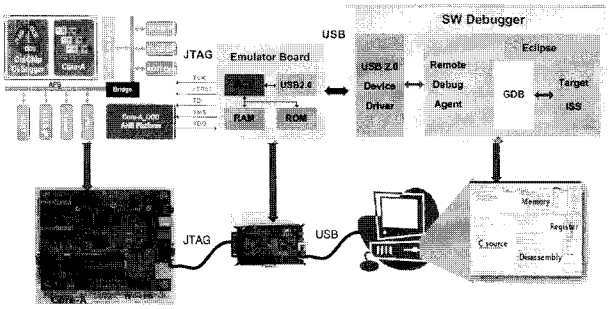


그림 17. 디버깅 검증 환경
Fig. 17. Verification environment for debugging.

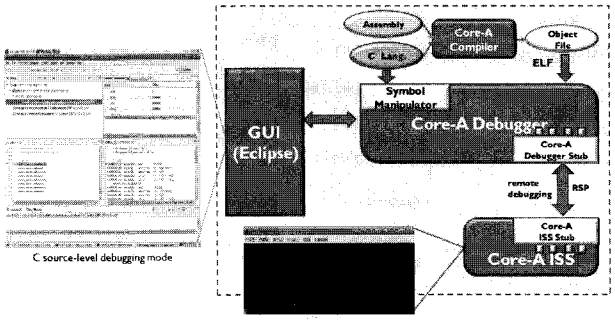


그림 18. SW 디버거 내부 구조
Fig. 18. Internal structure of SW debugger.

Core-A SoC에 내장하였다. Core-A SoC에 포함되는 주변장치에는 SRAM, NOR Flash, UART, AC'97 등의 제어 IP와 타이머(timer), PIC(Programmable Interrupt Controller), GPIO(General Purpose Input/Output) 등 IP가 있다. Core-A SoC는 이러한 IP들을 AMBA 버스를 이용하여 시스템을 꾸몄다. 그리고 타깃 시스템을 Xilinx ISE 툴로 합성, P&R(Place and Route)과정을 거쳐 최종적으로 netlist 파일과 타이밍 정보를 포함하는 SDF(Standard Delay Format)파일을 추출하였다. 추출된 파일과 SRAM, Flash, UART 등 주변장치의 시뮬레이션 모델을 이용하여 post layout 타이밍 시뮬레이션을 진행하여 행위수준 시뮬레이션 검증과 마찬가지로 OCD의 추가로 인하여 프로세서가 오동작을 하지 않는가를 먼저 확인하였다. 구체적으로 단일 명령어, 조합 명령어 및 여러 가지 알고리즘 명령어 조합을 수행시켜 ISS model의 실행 결과와 일치한지 비교 검증하였다. 검증 결과 OCD를 내장한 Core-A SoC가 정상적으로 동작하는 것을 확인하였다. 그리고 Core-A SoC를 Xilinx FPGA에 다운로드(download)하여 아래 그림 17과 같은 디버그 시스템을 구축하여 FPGA 레벨에서의 SW 디버거 연동 검증을 진행하였다.

타깃 시스템은 설계된 OCD를 내장한 Core-A SoC 플랫폼이다. 타깃 시스템과 Emulator Board를 JTAG cable로 연결하고 Emulator Board와 호스트 컴퓨터는 USB cable로 연결한다. 그리고 컴퓨터에는 GDB 기반의 Core-A SW 디버거를 실행시키면 전반적인 디버그 시스템이 구축된다.

SW 디버거와의 연동 검증 과정은 아래의 그림 18을 가지고 설명하면 다음과 같다. 먼저 프로세서가 수행할 임의의 프로그램(C언어와 assembly 언어)을 SW 디버거가 포함하고 있는 Core-A 컴파일러(compiler)로 컴파일하여 binary 파일과 디버깅 정보를 포함하는 ELF(Executable and Linking Format)파일을 생성한다. 그리고 SW 디버거로 타깃 시스템의 OCD를 인식하고 원격 디버깅을 시작한다. SW 디버거가 OCD와 연동되면 프로세서는 디버그 모드로 진입하게 되고 이때 SW 디버거로 binary 파일과 프로그램 수행에 필요한 데이터 파일을 Core-A의 프로그램 및 데이터 메모리에 다운로드한다. 다음에 SW 디버거로 타깃 실행(run) 명령을 OCD에 전달하면 Core-A SoC는 메모리에 다운로드된 프로그램을 수행하게 된다. 그리고 원하는 타이밍에 BP 혹은 WP 지정 및 감지, 실행, 정지, C/C++ 및 어셈블리 레벨에서의 싱글 스텝, 레지스터, 메모리 및 변수값을 읽고/쓰기 등 OCD의 디버깅 기능을 FPGA 레벨에서 SW 디버거와 실시간 연동하면서 검증하였다. 또한 SW 디버거로 Core-A ISS를 연동한 로컬 디버깅을 이용하여 프로그램 수행과정과 실제 FPGA에서 실행과정을 step by step으로 비교하면서 검증을 진행하였다.

검증에는 Core-A SoC 플랫폼에 포함하는 여러 주변장치 제어 프로그램, 그리고 ADPCM, MP3 디코더와 같은 음성응용알고리즘 등을 사용하였다. ADPCM, MP3 디코더 알고리즘 검증할 때에 주변장치를 아래와 같은 용도로 사용하였다. SRMA엔 프로그램 binary 파일, Flash엔 음원 데이터를 다운로드하고 UART와 GPIO는 외부와의 통신 역할, PIC는 코덱(codec)의 인터럽트를 감지하고 Core-A에 전달하는 역할, 그리고 AC'97은 음성을 스피커(speaker)로 출력하는 역할로 사용하였다.

FPGA 레벨에서의 SW 디버거 연동 검증에서 OCD의 오동작을 발견하면 PLI 연동 검증에서 최대한 FPGA와 비슷한 환경을 조성하여 원인을 찾고 수정하여 다시 FPGA 검증을 진행하였다. 이렇게 FPGA 검증과 PLI 검증을 반복 수행하면서 OCD의 디버깅 기능 검증을 수행하였다. 그리고 Core-A SoC에서 버스를

표 1. 디버깅 기능 수행 시간
Table 1. Executing time for debugging functions.

Halt mode debugging		ARM920T	Core-A
Debugging function	SW debugger view window		
Single Step	without any view	156ms	263ms
	with register view	158ms	266ms
	with memory view	203ms	344ms
BP Setting & Run	without any view	442ms	723ms
	with register view	443ms	730ms
	with memory view	527ms	823ms

Core-B Lite와 Wishbone버스로 변경하여서도 검증하였다. 이로써 설계된 OCD의 기능뿐만 아니라 Core-A에만 종속적인 OCD가 인터페이스 수정 없이 여러 종의 버스에 바로 연결하여 사용할 수 있다는 것도 검증하였다.

FPGA 레벨에서 OCD를 상용 ARM9 프로세서의 디버거와 성능비교를 진행하였다. ARM9은 32비트 RISC 타입의 프로세서이며 Harvard architecture, 5단 파이프라인 구조를 가지고 있다. 즉 ARM9은 Core-A와 비슷한 프로세서이다.

성능비교는 SW 디버거에서 레지스터 View창, 메모리 View창을 close 혹은 open하였을 때에 싱글스텝, BP 지정 및 실행 등 디버깅 기능을 수행하는데 걸리는 시간을 대상으로 하여 진행하였다. SW 디버거에서 특정 디버깅 기능을 한번 수행하는데 걸리는 시간을 측정할 수 없기 때문에 연속으로 1000번 수행하여 걸리는 시간을 평균하여 오차를 최소한으로 하였다. 디버깅 기능을 1000번 수행하는 방법은 수동으로 한 번씩 클릭하여 수행하는 것이 아니라 SW 디버거에서 명령어 스크립트(script)를 실행시켜 시간적 오차를 줄였다. 성능 비교할 때 사용된 Core-A와 ARM9이 실행하는 프로그램은 같은 bubble sort 알고리즘을 채택하였다. 그리고 두 프로세서의 컴파일러 성능차이가 있을 수 있기 때문에 C/C++ 레벨이 아닌, 어셈블리 레벨에서의 명령어 수를 똑같게 하여 디버깅 기능을 수행시켰다. ARM9의 디버그 시스템에서 Emulator Board는 Multi ICE를, SW 디버거로는 AXD를 사용하였다. 그리고 Core-A와 ARM9의 TCK의 주기는 400KHz로 하였다. 성능 비교 결과는 위의 표 1에서 자세히 보여주고 있으며 Core-A 디버그 시스템이 상용 ARM9 디버그 시스템보다 최적화되어 있지 않은 점을 감안하면 halt-mode 디버깅에 있어서 Core-A 디버거가 상용 디버거와 견주어 보아도 많이 부족하지 않다는 것을 확인할 수 있다.

표 2. 게이트 카운트
Table 2. Gate count.

Module name	Gate count
JTAG	3233
On-Chip Emlator (OCE)	3830
Debug Control Unit (DCU)	153
On-Chip Debugger (OCD)	7216
Target processor with OCD	48963

검증된 OCD를 Megnachip/Hynix 0.35μm CMOS 셀 라이브러리(cell library)로 합성결과는 표 2와 같다. 여기서 2-input NAND를 gate count 1로 간주한다. 그 결과 설계된 OCD는 약 14.7%의 오버헤드를 보인다. 그리고 stand-alone Core-A의 최대 합성 주파수가 95MHz이고 OCD를 포함한 Core-A의 주파수는 90MHz이다. 즉 OCD를 Core-A에 내장하는 것이 Core-A의 동작에 큰 영향을 주지 않는 것을 확인할 수 있다.

VI. 결 론

본 논문에서는 Core-A 프로세서를 디버깅 할 수 있는 OCD를 설계하였다. 구현한 OCD는 Ⅲ장에서 소개한 OCD 메커니즘에 따라 설계되어 적은 핀으로 프로세서를 제어할 수 있는 장점이 있다. 그리고 버스의 종류에 따라 따로 수정작업이 필요 없이 바로 OCD를 내장하여 사용할 수 있다. 또한 새로운 프로세서가 개발되었을 때 IE, IV, BR, MI, EC 신호만 해당 프로세서가 지원해주면 가장 적은 gate count를 가지고 있는 DCU를 파이프라인 제어에 맞게 조금만 수정하면 바로 프로세서의 디버깅 유닛(unit)으로 사용할 수 있다.

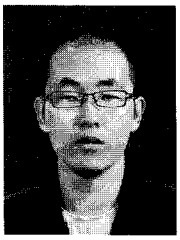
설계된 OCD를 Core-A SoC 플랫폼에 내장하여 GDB 기반의 Core-A SW 디버거와 연동하여 FPGA 레벨까지 3단계 검증을 진행하였다. 이로써 설계된 OCD는 디버거로서의 기능 및 신뢰성을 확인하였다.

참 고 문 헌

[1] 박형배, 지정훈, 허경철, 우균, 박주성, "GNU 디버거를 이용한 온칩 디버깅 시스템 설계", 전자공학

회논문지, 제46권 SD편, 제1호, 24-38쪽, 2009년 1월.
[2] www.core-a.net
[3] Richard Stallman, Roland Pesch, Stan Shebs, "GDB User Manual: Debugging With GDB(The GNU Source-Level Debugger)", GDB version 6.4. Technical report, Free Software Foundation, Cambridge, MA.
[4] C. MacNamee and D. Heffernan, "Emerging On-Chip Debugging Techniques for Real-Time Embedded Systems", IEEE Computing & Control Eng. J., vol. 11, no. 6, pp. 295-303, Dec. 2000.
[5] IEEE Std. 1149.1a-1993, "Test Access Port and Boundary-Scan Architecture", IEEE, Piscataway, N.J., 1993.
[6] ARM7TDMI Specification, DDI0210B, http://www.arm.com.
[7] Ing-Jer Huang, Chung-Fu Kao, Hsin-Ming Chen, Ching-Nan Juan, Tai-An Lu, "A retargetable embedded in-circuit emulation module for microprocessors", Design & Test of Computers, IEEE, Volume 19, Issue 4, pp. 28-38, July-Aug. 2002.
[8] Jundi, K., Moon, D., "Monitoring techniques for RISC embedded systems", Aerospace and Electronics Conference, 1993, vol.1, pp. 542-550, May 1993.
[9] C166S On Chip Debug Support, August 2001. http://www.infineon.com
[10] Jonathan B. Rosenberg, "How Debuggers Work-Algorithms, Data Structures, and Architecture", Wiley Computer Publishing, 1996.
[11] Hubert H¨ogl,Dominic Rath, "Open On-Chip Debugger", http://openocd.berlios.de/web/
[12] Yue-li Hu, Ke-xin Zhang, "Design of On-Chip Debug Module based on MCU", High Density packaging and Microsystem Integration, 2007. pp. 1-4, June 2007.
[13] G.R. Alves and J.M. Martins Ferreira, "From Design-for-Test to Design-for-Debug-and-Test: Analysis of Requirements and Limitations for 1149.1," Proc. 17th IEEE VLSI Test Symp. (VTS99), IEEE CS Press, Los Alamitos, Calif., pp.473-480, 1999.
[14] Jonathan B. Rosenberg, "How Debuggers Work-Algorithms, Data Structures, and Architecture", Wiley Computer Publishing, 1996.

저 자 소 개



허 경 철(학생회원)
2005년 중국 연변과학기술대학 전자통신학과 학사 졸업.
2008년 부산대학교 전자공학과 석사 졸업.
2010년 현재 부산대학교 전자전기공학과 박사과정.
<주관심분야 : 프로세서 설계, 디버거 설계, SoC 설계, 멀티 프로세서 플랫폼 설계, 오디오 알고리즘 구현>

박 형 배(정회원)
2004년 동서대학교 전자공학과 학사 졸업.
2006년 부산대학교 전자공학과 석사 졸업.
2010년 현재 부산대학교 전자전기공학과 박사과정.
<주관심분야 : 프로세서 설계, 디버거 설계, 디지털 시스템 설계, SoC 설계>

정 승 표(학생회원)
2007년 부산대학교 전자공학과 학사 졸업.
2009년 부산대학교 전자공학과 석사 졸업.
2010년 부산대학교 전자공학과 박사 과정.
<주관심분야 : 신호처리, SoC 플랫폼 설계>

박 주 성(평생회원)-교신저자
1976년 부산대학교 전자공학과 학사 졸업.
1978년 한국과학기술원(KAIST) 전자공학과 석사 졸업.
1989년 University of Florida 전자공학과 박사 졸업.
1998년~2007년 부산대학교 IDEC 센터장
1991년~현재 부산대학교 전자공학과 교수
2008년~2009년 부산대학교 공과대학 학장
<주관심분야 : DSP 설계, ASIC 설계, 반도체 소자 모델링, 음성/사운드 신호처리 및 구현, SoC 설계>