논문 2009-46SD-1-4

GNU 디버거를 이용한 온칩 디버깅 시스템 설계

(Design of On-Chip Debugging System using GNU debugger)

박 형 배*, 지 정 훈**, 허 경 철*, 우 균***, 박 주 성****

(Hyungbae Park, Jeong-Hoon Ji, JINGZHE XU, Gyun Woo, and Jusung Park)

요 약

본 논문에서는 OCD(On-Chip Debugger)기반의 프로세서 디버거 구현한 것에 대해서 소개한다. 구현한 디버거는 프로세서 칩 내부에 내장에 내장해서 디버깅 기능을 하는 OCD로직과 심볼릭(Symbolic) 디버깅 기능을 지원하는 GNU 디버거 기반의 소프트웨어 디버거, 그리고 소프트웨어 디버거와 OCD를 연결해주고 고속 디버깅을 지원하는 인터페이스 & 컨트롤(Interface & Control) 블록으로 3개의 기능 블록으로 구성되어 있다. 디버거는 대상 프로세서에 OCD블록을 내장하여 소프트웨어 디버거를 이용해서 C/Assembly 레벨에서 디버깅이 가능하다. 디버깅 시스템(On-Chip Debugging System)은 FPGA로 구현된 32비트 RISC 타입 프로세서 코어에 OCD 블록을 내장해서 소프트웨어 디버거와 인터페이스 & 컨트롤 블록을 연동하여 동작을 검증하였다.

Abstract

In this paper, we implement processor debugger based on OCD(On-Chip Debugger). Implemented debugger consist of software debugger that supports a functionality of symbolic debugging, OCD integrated into target processor as a function of debugging, and Interface & Control block which interfaces software debugger and OCD at high speed rates. The debugger supports c/assembly level debugging using software debugger as OCD is integrated into target processor. After OCD block is interfaced with 32bit RISC processor core and then implemented with FPGA, the verification of On-Chip Debugging System is carried out through connecting OCD and Interface & Control block, and SW debugger.

Keywords: GDB, JTAG, On-Chip Debugger, Remote debugging, On-Chip Debugging System.

I. 서 론

오늘날 임베디드시스템 개발은 짧은 제품 주기에 맞추기 위해서 개발 기간이 줄어들고 있으며 고성능의 다양한 기능을 요구하게 됨에 따라서 시스템 복잡도는 점점 더 높아지고 있다. 따라서 점차 시스템을 디버깅하는데 드는 비용과 시간은 점점 더 늘어나고 있으며 개발하는 과정에서 디버거의 의존도와 중요성은 매우 높아지게 되었다. 시스템에서 가장 중요한 부분을 차지하고 있는 프로세서를 선택할 때에도 프로세서 자체의 성능뿐만 아니라 지원하는 디버거의 효율성과 성능 또한 중요한 선택 조건이 되었다.

대부분의 임베디드시스템이 포함하고 있는 프로세서는 반도체 기술이 발달함에 따라서 동작 주파수가 높아지고 프로세서 칩 내부에 여러 종류의 기능 블록들을 한 개의 칩에 내장 할 수 있는 SoC(System On Chip) 기술이 발전함에 따라서 디버깅은 더욱더 어렵게 되었

* 학생회원, **** 평생회원, 부산대학교 전자공학과
(Dept. of Electronics Engineering, Pusan National University)
** 학생회원, *** 정회원, 부산대학교 컴퓨터공학과
(Dept. of Computer Engineering, Pusan National University)
※ 이 논문은 부산대학교 자유과제 학술연구비(2년)에 의하여 연구되었음.
접수일자: 2008년8월4일, 수정완료일: 2009년1월5일

다. 이와 같은 기술의 발전과 고성능 디버거의 필요성에 의해서 디버거는 기존에 로직어날라이져와 In-Circuit Emulation등과 같은 칩 외부에서 디버깅 기능을 구현하는 간접적인 접근 방식에서, 지금은 칩 내부에 디버깅 기능을 담당하는 로직(OCD)을 내장해서 디버깅 환경을 구현하는 직접적인 접근 방식으로 발전하였다[1]. 이와 같이 프로세서 칩 내부에 내장되어서 코어와 연동해서 동작하면서 다양한 디버깅 기능을 지원하는 블록을 OCD(On-Chip Debugger)라 한다.

칩 개발 툴(EDA: Electric Design Automation)과 설계 기술을 발전으로 HDL 언어를 이용한 고성능의 칩 설계가 가능해 짐에 따라서 막대한 자본력을 가지고 있는 거대한 칩 벤더에서만 개발이 가능했던 프로세서 설계가 저 자본의 설계가들 에게도 가능 하게 되었다. 특정 목적의 칩을 설계할 경우에 불필요한 블록까지 포함하고 있는 상용 프로세서 IP를 사용할 필요가 없이 시스템에 최적화된 프로세서를 자체적으로 설계하여서 칩에 내장할 수 있다. 그러나 독자적인 구조의 코어를 설계하였을 때 제일 먼저 부딪히는 부분이 디버거라고 할 수 있다. 오늘날 디버깅의 어려움으로 인해서 하드웨어엔지니어 뿐만 아니라 소프트웨어엔지니어도 적절한 디버거 없이는 아무리 좋은 성능의 프로세서라고 할지라도 사용하려고 하지 않을 것이다. 현재 대부분의 상용 프로세서는 OCD를 내장하고 있으며 지원하는 많은 종류의 상용 소프트웨어 디버거들이 있다. 그리고 OCD 설계에 대해서 소개하거나[2~3] 소프트웨어 디버거를 소개하고는[4~5] 있으나 설계 자체에 대한 설명에 치우친 경향이 많고 실제 디버거를 구현하기 위한 디버거의 구성이나 동작 메커니즘, 각 기능 블록들의 인터페이스 방법 등에 대해서는 소개하고 있지 않다. 따라서 본 논문에서는 OCD기반의 프로세서 디버거에 대해서 소개하고자 한다. 구현한 디버거에 대해서 OCD블록을 설계하고 대상 프로세서에 내장하는 방법과 GDB(GNU 디버거)[6]를 이용한 소프트웨어 디버거 구현 방법, 그리고 OCD와 소프트웨어 디버거를 인터페이스 하는 방법 등 전체 디버거를 구현하는 것에 대해서 자세하게 기술 한다.

본 논문의 전체 순서는 먼저 Ⅱ장에서 OCD를 이용한 디버거의 기본적인 내용에 대해서 소개하고, Ⅲ장에서는 설계한 디버깅 시스템에 대해서 설명한다. 그리고 하드웨어 블록과 소프트웨어 블록을 포함하는 디버깅 시스템을 3개의 블록으로 나누어서 Ⅳ장과 Ⅴ장 그리고 Ⅵ장에서 각각 설명한다. 마지막으로 Ⅶ장과 Ⅷ장에서 설계한 디버깅 시스템의 검증과정을 소개하고 결론을 맺는다.

Ⅱ. 개요 : 온칩 디버깅

프로세서의 디버거는 프로세서의 동작 상태를 디버깅하기 위한 개발 툴로서 기본적으로 메모리 읽기/쓰기, 레지스터 읽기/쓰기, 한 명령어식 실행 시키는 싱글스텝(single-step)과 특정 위치에 동작을 멈추게 하는 브레이크 포인터의 기능을 가지고 있다. 이와 같은 4가지 기능을 바탕으로 해서 디버거는 프로파일링(Profiling), 모니터링(Monitoring), 트레이싱(Tracing), 성능 테스트(Performance analysis) 등과 같은 다양한 기능을 제공한다.

기존의 프로세서 디버거를 구현하는 방법에는 크게 3가지방법이 적용 되어 왔다. 먼저 프로세서가 실행하는 코드 내부에 디버깅을 위한 코드를 추가하는 형태의 디버깅 커널(Debugging kernel)을 이용한 방법과 PCB보드 상에 롬이 실장 될 소켓에 호스트 PC의 포트와 연결된 소켓과 연결해서 프로세서를 제어하는 방식으로 동작하는 ROM Monitor 방식이 있다. 그리고 프로세서 외부에 디버깅을 위해서 추가적인 회로를 구성해서 디버깅 기능을 구현하는 방법인 In-Circuit Emulator 방식 이 있다[7~8]. 그러나 32비트의 수백 MHz로 동작하는 고성능의 프로세서가 개발되고 SoC 기술이 발달하면서 프로세서 칩 내부에 단순히 프로세서 코어만 내장하는 것이 아니라 다양한 기능블록들이 칩 내부에 내장됨에 따라서 기존의 방법으로는 효율적인 디버거를 구현할 수 없게 되었다. 이와 같은 디버깅의 어려움을 해결하기 위해서 현재의 대부분의 프로세서는 디버거를 구현하기 위해서 디버깅 기능을 하는 블록을 칩 내부에 내장하는 형태의 접근 방식으로 프로세서의 디버깅 기능을 구현하고 있다[1]. 오늘날 저가의 8비트 프로서세부터 32비트 프로세서까지 대부분의 프로세서가 OCD를 포함하는 형태로 출시되고 있으며, 칩 벤더마다 독자적인 구조의 OCD 아키텍처를 내장하고 있다. 대표적인 상용 OCD 블록으로는 Embedded ICE, ETM (Embedded Trace Macrocell), EJTAG (Extended JTAG), PDTRACE(Program and Data Trace), OCDS(On Chip Debug Support) 등이 있다[9~13].

OCD 기반의 프로세서 디버거는 그림 1에서 보는 것

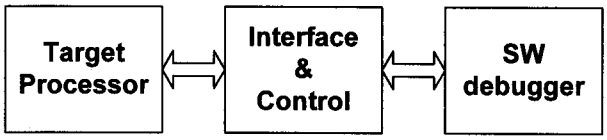


그림 1. 디버거 구조
Fig. 1. Structure of debugger.

과 같이 일반적으로 OCD를 내장한 대상 프로세서(Target processor)와 호스트 PC에서 동작하는 소프트웨어 디버거(SW debugger), 그리고 인터페이스 역할을 하는 인터페이스 & 컨트롤 블록으로 크게 3부분으로 구성되어 있다.

소프트웨어 디버거는 호스트 PC에서 동작하는 프로세서의 가상 시뮬레이션 모델인 ISS(Instruction Set Simulator)를 이용한 디버깅 모드와 소프트웨어 디버거를 대상 프로세서 칩 내부의 OCD블록에 연결해서 디버깅하는 원격 디버깅(Remote debugging)모드의 두 가지 동작 모드를 지원한다.

그리고 인터페이스 & 컨트롤 블록은 소프트웨어 디버거를 원격 디버깅 모드로 동작 할 때 소프트웨어 디버거와 대상 프로세서를 연결해주는 역할을 한다. 인터페이스 & 컨트롤 블록은 디버깅에 필요한 정보를 얻기 위해서 프로세서 칩 내부의 OCD 블록을 제어하고 디버깅 정보를 소프트웨어 디버거로 전달하는 방식으로 동작한다. 프로세서의 데이터 처리 양과 처리 속도가 높아짐으로 인해서 디버깅해야 할 데이터의 양 또한 증가 하였다. 고속 디버깅을 위해서 인터페이스 & 컨트롤 블록의 처리 속도도 또한 높아 져야 할 필요가 있게 되었다. 이와 같은 이유로 인해서 인터페이스 & 컨트롤 블록은 PC쪽의 소프트웨어 디버거와는 고속으로 통신하기 위해서 USB2.0 프로토콜을 사용하고, OCD 블록을 고속으로 제어하기 위해서 고성능의 전용 칩(Controller, FPGA)을 필요로 한다.

일반적인 소프트웨어 개발은 먼저 소프트웨어 디버거에 ISS 모델을 연동해서 코드를 개발하고 충분한 검증을 거치게 된다. 그리고 소프트웨어 디버거를 대상 프로세서에 연결하고 개발한 소프트웨어의 바이너리 코드를 메모리에 다운로딩해서 원격 디버깅모드에서 실제 프로세서의 동작을 디버깅하는 과정으로 진행된다.

III. 온칩 디버깅 시스템 : OCDS

본 논문에서 제안하는 OCDS(On-Chip Debugging System)은 프로세서 디버거를 위한 통합개발환경이다. OCDS는 프로세서의 ISS모델과 자체적으로 설계한 디버깅 로직인 OCD를 대상 프로세서에 내장해서 심볼릭 레벨(C/Assembly level)에서 디버깅이 가능하다. OCDS의 구성은 그림 2와 같이 칩 내부에 내장하는 OCD와 호스트 PC에서 동작하는 소프트웨어 디버거 그리고 인터페이스 블록으로 3개의 블록으로 나눌 수 있다. HDL 형태로 구현한 OCD 로직은 그림 2에서 보는 것과 같이 대상 프로세서 칩 내부에 내장되어서 코어 내부의 상태를 디버깅 할 수 있는 기능을 지원하는 블록이다. 소프트웨어 디버거는 오픈 소스형태로 개발되어서 널리 사용 되고 있는 GDB를 이용해서 프로세서의 아키텍처 의존적인 부분을 분석하여 수정하고 추가하는 방식으로 구현하였다. 그리고 GUI환경은 기본적으로 GDB와 연동 할 수 있는 기능을 지원하고 프로세서 디버깅을 위한 다양한 기능을 플러그인(Plug-in)형태로 포함하고 있는 Eclipse CDT를 사용하였다.

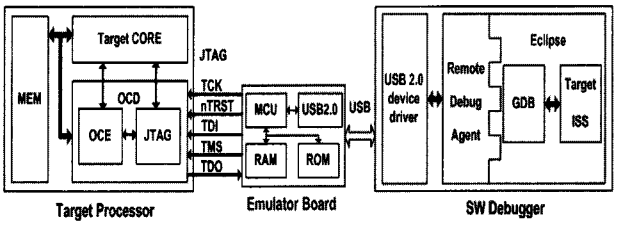


그림 2. 온칩 디버깅 시스템의 구조
Fig. 2. The structure of On-Chip Debugging System.

구현한 소프트웨어 디버거는 C 레벨과 Assembly 레벨에서 디버깅이 가능하고 원격 디버깅 및 다양한 디버깅 기능들을 제공한다. 그리고 소프트웨어 디버거와 대상 프로세서의 OCD 블록을 연결해주는 역할을 하는 인터페이스 & 컨트롤 블록은 그림 2에서 보는 것과 같이 소프트웨어 인터페이스 모듈인 RDA(Remote Debug Agent)과 하드웨어 인터페이스 모듈인 Emulator Board을 포함하고 있다. RDA는 소프트웨어 디버거의 디버깅 명령(메모리 읽기/쓰기, 레지스터 읽기/쓰기, 싱글스텝, 브레이크포인터 등)을 PC외부 하드웨어 인터페이스 모듈에 전달하고, Emulator Board 로부터 디버깅 데이터를 전달 받아서 소프트웨어 디버거로 전달하는 역할을 한다. Emulator Board는 디버깅 명령에 따라서 디버깅 정보를 프로세서로부터 얻기 위해서 OCD를 제어하게 되고 OCD로부터 수집한 디버깅 정보를 다시 호스트 PC의 RDA 모듈로 전달하는 기능을 한다. Emulator Board는 고속 디버깅을 위해서 USB2.0 칩과 고성능 프

로세서를 포함하고 있다. 다음 장부터는 구현한 OCDS의 각각의 기능 블록에 대해서 구조와 동작 메커니즘에 대해서 자세히 기술한다.

Ⅳ. 소프트웨어 디버거

소프트웨어 디버거는 심볼릭 레벨 디버깅을 지원하고 디버깅 시스템의 동작을 제어하는 역할을 한다. 또한 효율적인 디버깅을 위해서 모니터링, 트레이싱, 프로파일링 등의 다양한 기능을 제공한다. C 레벨의 코드는 컴파일 과정을 거치면서 그림 3과 같이 3개의 부분으로 구성된 실행파일(ELF:Executable Linker Format)로 변환된다[14]. 대상 프로세서는 컴파일 된 실행파일 중에 opcode로 만 구성된 binary part만 로딩하면 되지만, 소프트웨어 디버거는 심볼릭 레벨 디버깅 기능을 위해서 그림 3과 같이 Binary part 뿐만 아니라 실행파일의 구성을 나타내는 Header part와 디버깅 정보를 포함하고 있는 Symbol part도 로딩 하게 된다.

이번 장에서는 GDB를 이용해서 아키텍처 의존적인 부분을 수정 및 추가해서 소프트웨어 디버거를 구현하는 방법을 새롭게 설계한 32비트 RISC타입의 프로세서인 Core-A를 예로 중요한 부분을 간략히 설명하고자 한다.

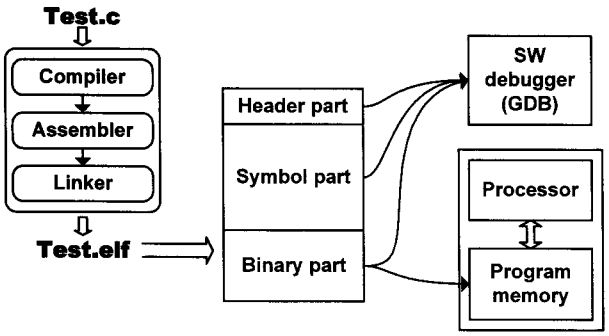


그림 3. 컴파일과 실행파일을 이용해서 디버깅하는 절차
Fig. 3. The procedure of compiling and debugging using executable file.

1. GDB를 이용한 소프트웨어 디버거 구현

GDB는 GNU 프로젝트 팀에서 오픈소스 형태로 개발한 고성능 디버거로 오랜 시간동안 안전성을 인정받은 범용 소프트웨어 디버거이다. GDB는 현재 리눅스(Linux) 및 유닉스(Unix), 윈도우즈(Windows) 등 다양한 플랫폼에서 사용 가능하며 C, C++, Assembly, Pascal 등 다양한 언어를 지원한다. 또한 GDB는 X86, ARM, MIPS, PowerPC 등 많은 종류의 프로세서들을 지원하고 있다.

새로운 프로세서 개발을 위해서는 어셈블러 및 컴파일러, 디버거 등의 프로그램 개발환경이 함께 개발되어야 한다. 프로세서를 위해 매번 새로 개발환경을 구현하는 것은 상당한 시간과 비용을 필요로 한다. 이러한 문제를 해결하기 위해 최근에는 재목적(retargetting) 기법을 이용하여 컴파일러 및 디버거를 개발하고 있다[15]. 재목적 기법은 이미 개발된 컴파일러 및 디버거에서 프로세서 종속적(target-dependent)인 모듈들을 새로운 프로세서에 맞게 수정하여 개발 비용과 시간을 단축시킨다[4]. GDB는 확장을 고려하여 설계되었으며, 주요 모듈은 프로세서 종속적인 부분과 프로세서 독립적인 부분으로 나누어져 있기 때문에 새로운 프로세서를 위한 소프트웨어 디버거 개발에 적합하다.

GDB는 크게 사용자 인터페이스, 심볼처리, 대상 시스템 지정의 3가지 모듈로 구성된다[16]. GDB는 사용자 인터페이스를 위해 CLI(Command Line Interface)와 MI(Machine Interface)를 제공한다. CLI는 명령형 입력 방식으로 사용자로부터 디버깅 명령을 전달받고 사용자가 해석하기 쉽도록 디버깅 결과를 출력한다. DDD와 XXGDB는 CLI 방식으로 GDB와 연결하여 GUI 디버깅 환경을 제공한다. MI 방식은 CLI 보다 구조화된 구조로 사용자보다 디버깅 도구들과의 인터페이스를 위해 설계되었다. GUI 디버거 인터페이스인 이클립스는 MI 방식을 이용하여 GDB와 연결된다. 심볼처리 모듈은 ELF, COFF 등의 실행파일을 로드하고, 디버거의 심볼 정보를 관리한다. 심볼처리 모듈에서는 주로 변수가 몇 번지의 어드레스에 위치하고, 함수의 시작 주소가 실제

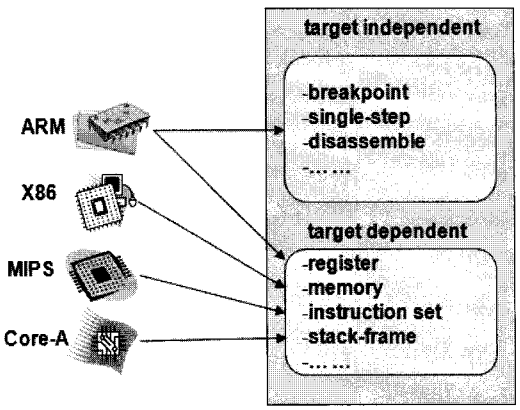


그림 4 재목적 디버거 구현
Fig. 4. Implementation of a retargetable debugger.

메모리의 몇 번지에 해당하는 지 등의 컴파일 되기 전의 C코드와 컴파일 된 후의 실행 파일 사이를 매칭 시켜주는 역할을 한다.

대상 시스템 지정 모듈은 레지스터, 메모리, 명령어와 같은 대상 프로세서 아키텍처와 관련된 정보를 지정하는 모듈이다. GDB를 이용해서 대상 프로세서용 소프트웨어 디버거를 개발하기 위해서는 그림 4와 같이 프로세서 의존적인 부분인 레지스터, 메모리 맵, 명령어 등을 수정하고 보완해야 한다.

2. 대상 프로세서 추가

GDB에 새로운 대상 프로세서를 추가하기 위해서는 먼저 그림 5와 같이 프로세서 종속적인 모듈에 해당하는 Target Architecture Definition 모듈에 대상 프로세서 정보를 등록해야 한다. 이 모듈에는 대상 프로세서의 레지스터 구조 및 메모리 구조, 스택 프레임, 명령어 셋 등의 정보가 등록된다. GDB에는 현재 X86, ARM, MIPS등과 같이 많이 사용되고 있는 프로세서들에 대한 모듈들이 이미 구현되어 있다. GDB의 Target Architecture Definition 모듈은 "*.mt", "gdbarch.c", "*-tdep.c" 파일들로 구성된다. 먼저, "*.mt"는 Target Architecture Definition이 정의되어 있는 소스파일에 대한 설정파일이다. GDB는 빌드 시에 "*.mt" 파일에 저장되어 있는 설정정보를 이용해 Target Architecture Definition에 대한 소스코드를 빌드 한다. "*.mt" 파일의 형식은 "옵션=파일"이다. 옵션에 대한 tag 명은 미리 정의되어 있다. 대상 프로세서의 아키텍처 정보를 추가하기 위해서는 TDEPFILES, TDEPLIBS, SIM에 대한 파일을 맵핑시켜주면 된다. 대상 프로세서에 대한 아키텍처 정보를 추가하기 위해서 "TDEPFILES=target_processor-tdep.c"를 추가한다. 이와 같이 설정한 상태에서 target_processor-tdep.c 파일에 프로세서에 대한 정보를 추가한다. GDB는 프로세서 종속적인 모듈과 독립적인 연결하기 위해 gdbarch.c 파일에 GDBARCH를 정의하고 있다.

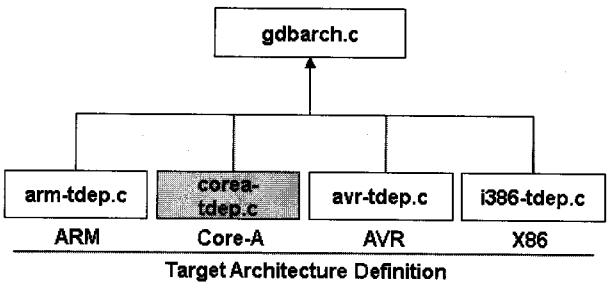


그림 5. 대상 프로세서 아키텍처 정의 모듈
Fig. 5. Target architecture definition module.

GDB를 확장하기 위해서는 GDBARCH에 정의되어 있는 인터페이스 함수들을 재 정의해야 한다. GDBARCH와 Target Architecture Definition 모듈에 대한 연결구조는 그림 5와 같다.

가. 레지스터 추가

대상 프로세스는 32bit RISC 구조로 그림 6에서와 같이 16개의 General Purpose Register(GPR)와 Intermediate Data Register(IDR), Program Count(PC), Processor Status Register(PS)의 Special Purpose Register(SPR)로 구성된다.

GDB에 레지스터 정보를 등록하기 위해서는 레지스터 개수와 레지스터 이름, 그리고 레지스터의 번호를 GDBARCH에 등록해야 한다. GDBARCH 인터페이스는 그림 7에서와 같이 "gdbarch.c"에 정의되어 있다.

대상 프로세서는 총 19개의 레지스터를 사용하며, 프

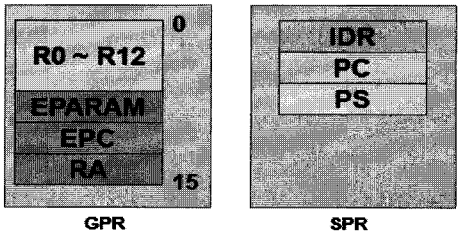


그림 6. 대상 프로세서 레지스터 맵
Fig. 6. Register map of target processor.

```
gdbarch.c
struct gdbarch {
    ... ...
    /* basic architectural information */
    const struct bfd_arch_info * bfd_arch_info;
    int byte_order;
    enum gdb_osabi osabi;
    ... ...
    int sp_regnum;
    int pc_regnum;
    int ps_regnum;
    int fp0_regnum;
    ... ...
    gdbarch_register_name_ftype *register_name;
    gdbarch_register_type_ftype *register_type;
    gdbarch_print_insn_ftype *print_insn;
    ... ...
};
```

그림 7. GDBARCH 구조체 정의
Fig. 7. Definition of GDBARCH structure.

corea-tdep.h

```
#ifndef __COREA_TDEP_H__
#define __COREA_TDEP_H__
#define NUM_COREA_REGS 19
#define COREA_FP_REGNUM 10
#define COREA_SP_REGNUM 12
#define COREA_PC_REGNUM 17
#define COREA_RA_REGNUM 15
#endif
```

그림 8. 대상프로세서의 레지스터 번호 정의
Fig. 8. Definition of traget processor register number.

corea-tdep.c

```
static struct gdbarch *
corea_gdbarch_init (struct gdbarch_info info, struct
gdbarch_list *arches)
{
    struct gdbarch_tdep *tdep;
    struct gdbarch *gdbarch;
    struct gdbarch_list *best_arch;
    ... ...
     /* Information about registers, etc. */
    set_gdbarch_deprecated_fp_regnum (gdbarch,
                                      COREA_FP_REGNUM);
    set_gdbarch_sp_regnum (gdbarch,
                                      COREA_SP_REGNUM);
    set_gdbarch_pc_regnum (gdbarch,
                                      COREA_PC_REGNUM);
    set_gdbarch_deprecated_register_byte (gdbarch,
                                      corea_register_byte);
    set_gdbarch_num_regs (gdbarch,
                                      NUM_COREA_REGS);
    ... ...
    return gdbarch
}
```

그림 9. 디버거 레지스터 초기화
Fig. 9. Debugger register initialization.

로그램 수행을 위한 주요 레지스터로 FP, SP, PC, RA 레지스터를 사용한다. GDB에 레지스터를 인식시키기 위해 주요 레지스터 번호를 그림 8과 같이 총 19개의 레지스터 중 16개의 GPR를 위해 0~15의 레지스터 번호를 할당하고, 3개의 SPR을 16~18사이에 각각 할당한다.

GDBARCH의 인터페이스 함수는 GDB가 실행될 때, 가장 먼저 실행되는 초기화 함수에서 등록된다. 인터페이스 함수를 통해 Core-A 프로세서의 레지스터 정보를 GDBARCH에 등록하는 부분은 그림 9와 같다.

GDB에 대상 프로세서의 레지스터 정보를 등록하는 것은 가장 중요한 과정 중에 하나이다. 특히, 대상 프로세서에 따라서 PC(Program Counter) 레지스터와, SP(Stack Pointer) 레지스터, 그리고 CALL 명령을 실행하면서 리턴 어드레스를 저장하는 레지스터 RA (Return Address)가 등록한 레지스터 중에 몇 번 레지스터 인지를 GDB에 올바르게 지정해야 GDB를 이용해서 올바르게 디버깅을 할 수 있다.

나. 명령어 셋 추가

대상 프로세서의 레지스터 정보를 GDBARCH에 등록하고 다음단계로 명령어 셋을 GDB에 등록한다. 대상 프로세서의 명령어 셋을 등록하기 위해서는 "~/gdb/opcodes"에 위치한 opcode와 관련된 모듈을 수정해야 한다. GDB는 대상 프로세서의 opcode에 관해서 그림 10과 같이 4가지 정보(arch, value, mask, assembler)를 필요로 한다. arch는 명령어의 아키텍처를 나타내고, value는 opcode의 기계어 코드 값을 나타낸다. 그리고 mask는 opcode에 대한 마스크 값이고, assembler는 명령어에 대한 디코딩 형식을 나타낸다. 명령어의 opcode와 mask 값을 and(&) 연산을 하면 코드의 value 값을 얻을 수 있다. 그리고 assembler는 GDB가 프로그램 메모리영역을 읽어서 Disassembly를 실행 할 때 opcode에 따른 뉴모닉(Mnemonic)을 나타낸다.

corea-dis.c

```
struct opcode32
{
    /* Architecture defining this insn. */
    unsigned long arch;
    unsigned long value, mask; /* if (op&mask)==value. */
    char *assembler;    /* How to disassemble this insn. */
};
... ...
static const struct opcode32 corea_opcodes[] =
{
    /* arch          value          mask    assembler */
    {COREA_EXT_V1, 0x26000000, 0x7e00001f,
                    "add%u%V\t%20-23r, %16-19r, %o"},
    {COREA_EXT_V1, 0x26000006, 0x7e00001f,
                    "adc%u%V\t%20-23r, %16-19r, %o"},
    {COREA_EXT_V1, 0x26000001, 0x7e00001f,
                    "sub%u%V\t%20-23r, %16-19r, %o"},
    {COREA_EXT_V1, 0x26000007, 0x7e00001f,
                    "sbb%u%V\t%20-23r, %16-19r, %o"},
    {COREA_EXT_V1, 0x26000004, 0x7e00001f,
                     "or%u%V\t%20-23r, %16-19r, %o"},
    ... ...
}
```

그림 10. 명령어 셋 정의
Fig. 10. Definition of instruction set.

3. GDB를 이용한 원격 디버깅

원격 디버깅은 호스트 시스템과 물리적으로 떨어져 있거나 다른 실행환경에서 동작하는 프로그램을 디버깅 하는 것을 의미한다. 일반적으로 호스트 시스템은 프로그램 개발환경(컴파일러와 디버거)이 설치되어 있는 시스템을 말하고, 타겟 시스템은 시뮬레이터(simulator)나 프로세서를 장착한 하드웨어 임베디드시스템을 말한다. 호스트 시스템과 타겟 시스템은 주로 시리얼 포트(serial port)나 네트워크를 통해 연결되고 이들 사이의 디버깅 작업은 사전에 정의된 디버깅 프로토콜에 의해 진행된다. 호스트 시스템에서 GDB로부터 디버깅 명령을 입력받아 타겟 시스템으로 전달하면 타겟 시스템의 원격 디버깅 서버가 명령어를 해석하여 타겟 시스템을 제어하는 형태로 디버깅을 하게 된다.

GDB는 원격 디버깅을 위해서 기본적으로 GDB 서버(gdb_server)라고 부르는 스텁(stub) 모듈을 지원한다. GDB 서버는 타겟 시스템에서 독립(standalone) 프로세스로 수행되며 GDB가 지원하는 디버깅 기능을 원격으로 사용 가능하게 해준다. 또한, GDB는 GDB 서버를 이용해서 타겟 시스템의 운영체제에 상관없이 디버깅 기능을 제공할 수 있다. GDB서버를 통해서 프로세서의 레지스터 값을 요청하는 과정은 그림 11과 같다.

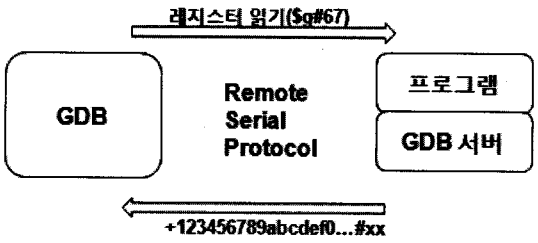


그림 11. 레지스터 값 요청
Fig. 11. Requesting all registers.

GDB는 원격 디버깅을 위해서 기본적으로 제공(remote.c)하는 RSP(Remote Serial Protocol) 프로토콜로 GDB 서버와 통신한다[17~19]. RSP는 ASCII 문자 기반의 프로토콜로 socket을 이용해서 통신을 한다. GDB와 GDB 서버 사이의 원격 디버깅은 GDB가 디버깅 명령 전달하고 그 결과가 도착할 때 기다리는 반이중 방식(half-duplex)으로 동작한다. 모든 GDB 명령과 GDB 서버의 응답은 RSP 프로토콜 패킷으로 구성되어 전달되며 프로토콜 형식은 다음과 같다.

- 명령 : $packet-data#checksum
- 응답 : +(-)$result-data#checksum

RSP 패킷은 '$'로 시작하여 #로 끝난다. 그리고 checksum은 통신오류를 체크하기 위해서 사용된다. packet-data는 한 바이트 이상의 ASCII 메시지로 디버깅 명령에 해당한다. 명령에 대한 응답의 시작은 '+' 또는 '-'로 시작한다. '+'는 packet-data가 정상적으로 전달되었으며 결과를 전송하는 것을 의미한다. '-'는 통신오류로 인해서 packet-data의 재전송 요구를 의미한다. 예를 들어, 메모리의 0x4015bc 번지로부터 두 바이트의 데이터 값을 요청할 경우, GDB는 GDB 서버로 RSP 패킷 "$m4015bc,2#5a"을 전달한다. 여기서 'm'은 메모리 읽기 명령어임을 나타낸다. 그리고 성공적으로 디버깅 명령을 수행했으면 GDB 서버는 '+'와 함께 요청한 데이터 메모리 값을 GDB로 RSP 패킷 형태로 전달한다. 메모리에 2f86이 저장되어 있을 경우 "+$2f86#06"을 패킷으로 전달한다.

GDB는 각기 다른 2종류의 GDB서버를 통해서 호스트 PC에서 동작하는 ISS 모델을 이용한 디버깅 모드와 실제 프로세서를 이용해서 원격 디버깅 모드를 지원할 수 있다. RSP 프로토콜에서 정의하고 있는 여러 종류의 디버깅 명령 중에서 표 1은 원격 디버깅을 위해서 GDB서버가 기본적으로 지원해야 하는 디버깅 명령을 나타내고 있다.

ISS모델을 이용한 디버깅 모드에서는 대상 프로세서의 ISS 모델은 프로세서 코어 모듈과 GDB 서버 모듈로 구성된다.

GDB 서버가 시작되면 원격 디버깅 모듈에서는 그림

표 1. 원격 디버깅을 위해서 필요한 디버깅 명령
Table 1. The necessary debugging command for remote debugging.

명령어	지원여부	기능
g	지원	레지스터 읽기
G	지원	레지스터 쓰기
m	지원	메모리 읽기
M	지원	메모리 쓰기
z	지원	브레이크포인터 설정
Z	지원	브레이크포인터 해제
s	지원	싱글스텝
c	지원	Continue
k	지원	프로세스 종료

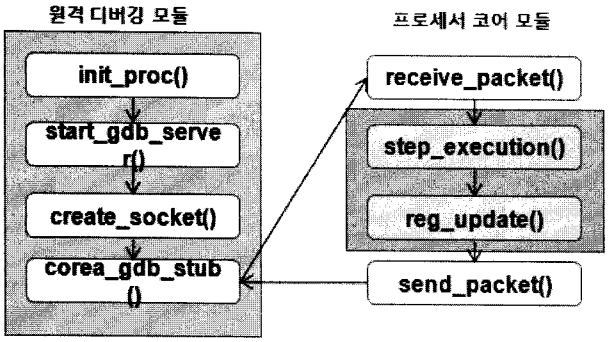


그림 12. GDB 서버 실행 과정
Fig. 12. Execution flow of GDB server.

12에서와 같이 init_proc() 함수를 통해 ISS 모델의 상태를 초기화하고 start_gdb_server() 함수에서 통신 소켓(socket)을 생성하고 GDB의 접속을 기다린다. GDB와 연결되면 corea_gdb_stub() 함수는 입력으로 받은 RSP 패킷 데이터를 해석하여 프로세서 코어 모듈에서 디버깅 명령을 수행한다.

원격 디버깅 모듈은 프로세서 코어 모듈에서 디버깅 동작에 해당하는 함수를 호출하는 방식으로 표 1에 기술한 디버깅 동작을 수행하게 된다.

V. 온칩 디버거

설계한 OCD 블록은 프로세서 칩 내부에서 코어와 연동해서 동작하면서 다양한 디버깅 기능을 지원하는 블록이다. OCD 블록의 구조는 코어의 특징과 필요로 하는 디버깅 기능과 성능에 따라서 다양한 접근 방법으로 구현이 가능하다[20~21]. 코어 내부에 디버깅용 명령어를 이용해서 구현하는 방법과 특정레지스터를 이용해서 구현하는 방법이 있고, 코어 외부에 별개의 블록을 구성해서 구현하는 방법 등이 있으며 상용 프로세서들도 다양한 접근 방식으로 OCD 기능을 구현하고 있다[9~13].

OCD 블록의 일반적인 구조는 그림 13에서 보는 것과 같이 코어와 연동해서 디버깅 기능을 하는 Debugger 블록과 통신채널(Communication channel)부분으로 구성된다.

통신채널은 프로세서 칩 내부의 Debugger 블록과 호스트 PC상의 소프트웨어 디버거 사이를 연결하는 역할을 한다. 기본적인 통신 채널은 소프트웨어 디버거의 디버깅 명령을 Debugger블록에 전달하고 OCD의 디버깅 결과를 소프트웨어 디버거에 전달하는 동작을 한다.

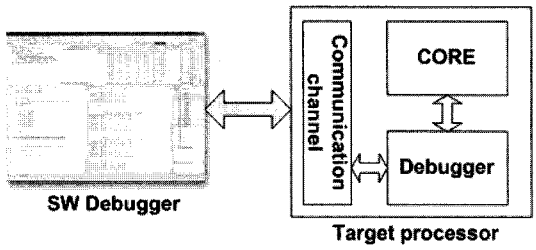


그림 13. 일반적인 OCD 구조
Fig. 13. General structure of OCD.

통신 채널로는 현재 JTAG 프로토콜을 가장 많이 적용하고 있다. JTAG 프로토콜은 단순한 구조와 적은 수의 핀 할당으로 구현이 가능하고, 사용 목적에 따라서 유연하게 구성이 가능한 장점이 있기 때문에 현재 대부분 프로세서는 통신 채널로 JTAG 프로토콜을 사용하고 있다[22~23].

1. 온칩 디버거 구조 및 동작 메커니즘

설계한 OCD 블록은 코어 외부에서 병렬적으로 동작하면서 코어와 메모리 사이의 어드레스/데이터 버스의 변화를 모니터링 하는 방식으로 디버깅 동작을 한다. OCD는 브레이크포인터/와치포인터 기능과 메모리 읽기/쓰기, 레지스터 읽기/쓰기, 싱글스텝의 디버깅 기능을 지원하며, 2개의 하드웨어 브레이크 포인터 레지스터를 가지고 있다. OCD를 내장한 프로세서는 실행 상태인 Run 모드와 디버깅을 위한 Debug 모드의 두 개의 동작 상태를 가지게 된다. Debug 모드에서 프로세서는 동작을 멈춘 정지 상태가 되고 외부 메모리와 차단된 상태가 된다. 그리고 OCD가 프로세서의 동작을 제어하게 되고 디버깅 정보(레지스터, 메모리 등)를 JTAG 포트를 통해서 칩 외부로 전달 할 수 있는 상태가 된다. OCD의 JTAG 구조는 그림 14에서 보는 것과

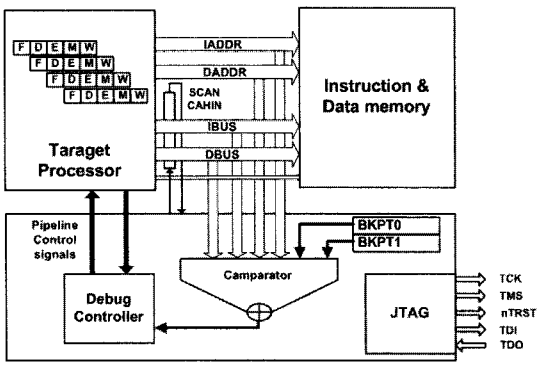


그림 14. OCD 구조
Fig. 14. Architecture of OCD.

같이 스캔체인이 프로그램/데이터 버스에 연결된 구조를 채택하고 있다.

프로그램/데이터 버스는 Run 모드에서는 코어 외부의 메모리와 연결되어 있지만, Debug 모드에서는 JTAG 블록의 Scan-chain을 통해서 연결된다. OCD의 구조는 그림 14에서와 같이 실질적인 디버깅 기능을 하는 Comparator와 Debug Controller로 구성된 OCE (On-Chip Emulator)부분과 통신채널 역할을 하는 스캔체인과 TAP을 포함하는 JTAG 부분으로 구성된다. OCE 블록은 세부적으로 브레이크포인터 상황을 감지하는 기능을 하는 Comparator 블록과 OCD가 코어와 연동해서 동작할 수 있도록 인터페이스 역할을 하는 Debug Controller블록으로 나눌 수 있다. Comparator 블록은 프로세서가 동작하고 있는 과정에서 OCD내부의 브레이크포인터 레지스터에 설정된 브레이크포인터 상황과 코어의 어드레스/데이터 버스와 메모리 제어신호를 비교해서 현재 프로세서의 상태가 브레이크포인터 상황인지 판단하고 그림 15에서와 같이 브레이크포인터 신호(breakpt)를 발생한다. OCD와 코어의 연결 관계를 보여주는 그림 15에서와 같이 Debug Controller블록은 코어의 동작 상태를 나타내는 3가지의 신호와 Comparator블록의 breakpt신호를 참고해서 브레이크포인터 걸린 상황에 따라서 적절한 디버그 모드 진입/탈출 타이밍을 판단하고, 코어의 입력으로 연결되어 있는 PD(Pipeline Disable)신호를 통해서 원하는 타이밍에 코어가 디버그 모드에 진입 및 탈출 할 수 있게 코어의 파이프라인 동작을 제어한다.

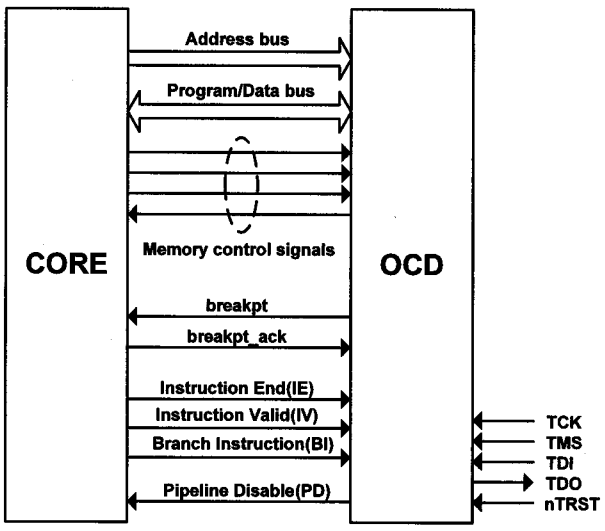


그림 15. OCD와 대상 프로세서의 인터페이스 신호
Fig. 15. Interfacing signals between OCD and processor.

적적한 타이밍을 파악하기 위해서 Debug Controller 블록이 참고하는 코어의 내부 상태를 나타내는 신호는 명령어 실행의 끝을 나타내는 IE(Instruction End)신호와 실행하는 명령어의 컨디션 조건이 참인지 거짓인지를 나타내는 IV(Instruction Valid), 그리고 브랜치 타입의 명령어를 나타내는 BI(Branch Instruction)신호이다.

Debug Controller블록은 브레이크포인터가 걸린 상황에서 브레이크포인터 어드레스의 명령어의 실행 결과가 업데이트되기 전에 디버그 모드에 진입할 것인지 아니면 업데이트 되고 난 후에 디버그 모드에 진입할 것인지를 판단하기 위해서 IE신호를 참고하게 된다. 그리고 브랜치 명령어 다음 어드레스에 브레이크포인터가 걸린 상황에서 Debug Controller블록은 BI신호를 이용해서 이전 명령어가 브랜치 명령인 것을 인지하고 IV신호를 이용해서 브랜치 명령의 컨디션이 참인지 거짓인지를 판단하게 된다. 코어의 파이프라인 동작을 제어하기 위한 PD신호는 브레이크 포인터 명령어의 Execute pipeline단에서 디버그 모드에 진입할 때 전 단의 파이파라인단에서 이미 실행하고 있는 명령어를 플러쉬(Flush)시키는 기능을 한다. 프로세서가 breakpt신호를 참고해서 Debug 모드에 진입하게 되면, OCD 블록은 프로세서가 디버그 모드에 진입했음을 나타내는 breakpt_ack신호를 참고해서 프로세서의 제어권을 넘겨받는다. 프로세서는 OCD의 요청으로 디버깅을 위해서 디버그 모드에 진입하게 되고, 그림 16에서와 것과 같이 프로그램 버스에 연결되어 있는 스캔체인을 통해서 디버깅 명령을 입력받는다. 그리고 디버깅 명령의 실행 단계에서 데이터 버스의 상태를 캡처하거나 로드하는 방식으로 프로세서 내부의 레지스터와 메모리의 값을

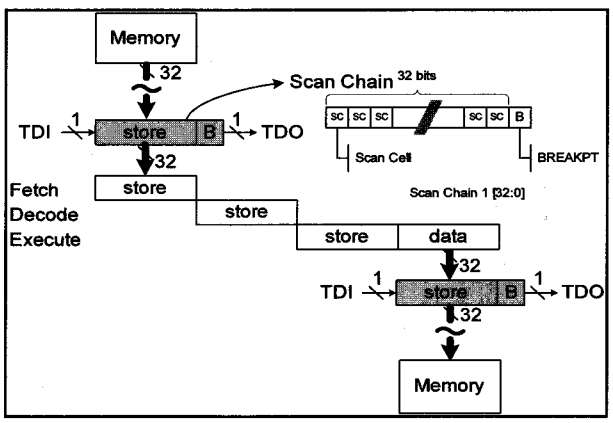


그림 16. OCD를 이용한 레지스터 읽기/쓰기 과정
Fig. 16. The process of reading/writing register using OCD block.

표 2. OCD블록 게이트 카운트
Table 2. Gate-counts of OCD block.

Module name	Gate counts
JTAG(scan-chain, TAP)	2,433
On-Chip Emulator(OCE)	2,730
Debug Controller(DCU)	193
Total	5,356

읽거나 쓸 수 있다. 디버깅 명령은 프로세서의 레지스터 읽기/쓰기(Store/Load register), 메모리 읽기/쓰기(Store/Load memory) 명령으로 구성되고, 디버그 모드에서 명령어 페치 파이프라인(Fetch pipeline) 단에서 스캔 체인을 통해서 프로그램버스에 로드된다[9].

설계한 OCD블록은 합성 가능한 Verilog HDL로 구현하였고, 0.18um 라이브러리로 합성했을 경우에 각 블록의 게이트 수는 표 2와 같다.

2. JTAG 프로토콜

JTAG는 디버깅 데이터의 통로 역할을 하는 스캔 체인과 스캔 체인의 동작을 제어하는 TAP컨트롤러 부분으로 구성된다. 스캔 체인은 기본적으로 직렬로 연결된 병렬 Load/Store가 가능한 쉬프트레지스터 형태의 구조를 가지고 있고 사용 목적에 따라서 다양한 구조로 구현이 가능하다. TAP 컨트롤러는 TAP 스테이트 다이어그램을 이용해서 5개의 TAP (TCK, TMS, TDI, TDO, nTRST)포트로 제어된다[20].

본 논문에서 제안하는 OCD블록은 통신 채널로

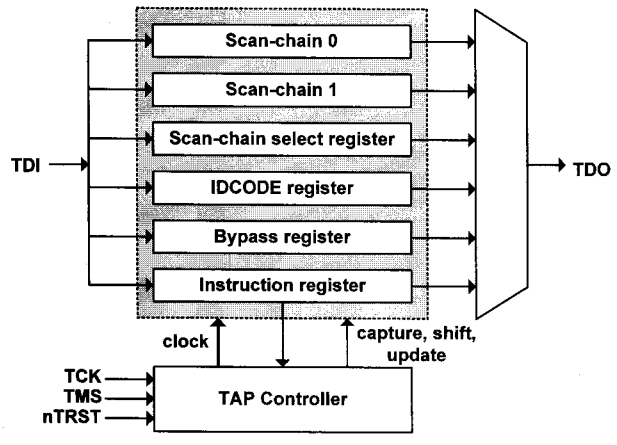


그림 17. Scan-chain 구조
Fig. 17. Scan-chain organization.

표 3. JTAG 명령어
Table 3. JTAG commands.

Instruction	IR value	register
INTEST	b'1100	Scan-chain0 or 1
EXTEST	b'0000	Scan-chain0 or 1
IDC0DE	b'1110	IDCODE register
BYPASS	b'1111	BYPASS register
SCAN_N	b'0010	Scan-chain select register
RESTART	b'0100	BYPASS register

JTAG 프로토콜을 적용하고 있다. 설계한 JTAG 블록은 그림 17에서와 같이 Instruction register와 2개의 스캔 체인을 포함하는 데이터 레지스터들, 그리고 TAP 컨트롤러로 구성된다.

지원하는 JTAG 명령어와 명령어에 따라서 TDI와 TDO사이에 연결되는 레지스터는 표 3과 같다. 스캔 체인0은 Debug 모드에서 디버깅명령을 실행시키기 위해서 코어의 명령어 버스와 데이터 버스에 연결되어 있고 스캔 체인1은 브레이크 포인터를 설정하기 위해서 OCE 블록 내부의 브레이크 포인터 레지스터에 연결되어 있다. scan-chain select register는 스캔 체인0와 스캔 체인1을 선택하는 기능을 하고, SCAN_N (b'0010) 명령어는 스캔 체인을 선택하기 위해서 Scan-chain select register를 TDI와 TDO사이에 연결시킨다. RESTART 명령어는 Debug mode에서 탈출하는 과정에서 OCD블록을 제어하기 위해서 사용되는 명령이다.

3. 온칩 디버거 제어 메커니즘

특정 프로그램 어드레스에서 브레이크 포인터가 걸린 상황에서 OCD 블록을 제어해서 디버깅을 하는 기본적인 제어 메커니즘은 그림 18과 같이 나타낼 수 있다. 설계한 OCD는 디버깅을 위해서 코어의 제어권을 OCD가 넘겨받는 상태인 Debug 모드라는 특정한 상태를 가지고 있다.

디버깅 과정에서 OCD는 Debug 모드 진입과 탈출을 반복하게 된다. 디버깅을 하기 위해서 Debug 모드에 진입하게 되고 디버깅이 끝나면 Debug 모드에서 탈출해서 다시 외부 메모리에 연결되어서 메인 클록으로 동작하는 Run 모드로 돌아간다. Debug 모드에서 디버깅 명령을 이용해서 디버깅 하는 방식이므로 OCD 블록이

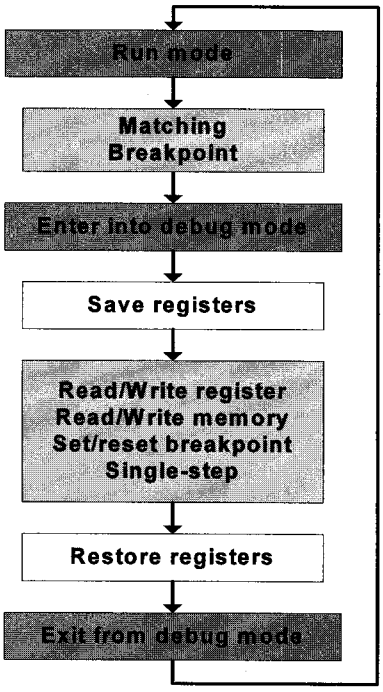


그림 18. OCD 제어 메커니즘
Fig. 18. The control mechanism of OCD.

레지스터 읽기/쓰기, 메모리 읽기/쓰기, 브레이크 포인터 설정, 싱글스텝 등과 같은 디버깅 동작을 하는 과정에서 프로세서 내부의 레지스터 값들이 바뀌게 된다. 따라서 프로세서 칩 외부에서 OCD를 제어하는 블록은 그림 18에서 나타내는 것과 같이 Debug 모드에 진입할 때 프로세서 내부 레지스터 상태를 저장하게 되고, 디버깅이 끝나고 Run 모드로 돌아가는 시점에 저장해 놓은 레지스터 값을 이용해서 레지스터 상태를 Debug 모드 진입 전의 상태로 다시 복귀키면서 Debug 모드를 탈출하게 된다.

VI. 인터페이스 & 컨트롤 블록

소프트웨어와 하드웨어 부분을 포함하는 인터페이스 & 컨트롤 블록은 소프트웨어 디버거와 프로세서 내부의 OCD블록 사이를 연결하는 기능을 하는 부분을 가리킨다. 인터페이스 & 컨트롤 블록은 소프트웨어 디버거가 필요한 디버깅 정보를 얻기 위해서 JTAG 신호를 이용해서 프로세서 내부의 OCD블록을 제어하고, OCD의 디버깅 정보를 소프트웨어 디버거에 전달하는 형태로 동작한다[19, 24]. 인터페이스 & 컨트롤 블록의 구성은 그림 19와 같이 3개의 기능 블록으로 분류할 수 있다.

Socket(read/write command)블록은 원격 디버깅을

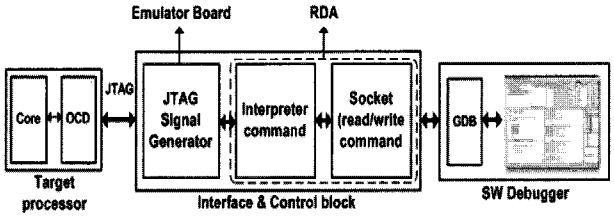


그림 19. 인터페이스와 컨트롤 블록
Fig. 19. The block of Interface and control.

위해서 GDB와 RSP프로토콜로 소켓 통신을 담당하는 블록으로 GDB로부터 디버깅 명령을 읽어 들이고 OCD 블록의 디버깅 정보를 다시 GDB로 전달하는 기능을 한다. Interpreter command 블록은 Socket 블록을 통해서 RSP 프로토콜로 전달된 디버깅 명령을 해석해서 JTAG Signal Generator 블록에 전달하는 기능과 OCD의 디버깅 정보를 다시 GDB로 전달하기 위해서 RSP 프로토콜 포맷으로 변환해서 Socket 블록에 전달하는 동작을 한다. JTAG Signal Generator 블록은 디버깅 명령에 해당하는 디버깅 동작을 하기 위해서 JTAG의 5개의 핀을 통해서 OCD블록을 제어하고 OCD의 디버깅 정보를 저장해서 Interpreter command블록으로 전달하는 기능을 한다.

소프트웨어 인터페이스 모듈인 RDA가 Socket(read/write command)블록과 Interpreter command블록의 기능을 하도록 구현하였고, 하드웨어 인터페이스 모듈인 Emulator Board가 JTAG Signal Generator 블록의 기능을 하도록 각각 구현하였다.

1. 소프트웨어 인터페이스 모듈

RDA는 GDB와 Emulator Board사이에서 GDB로부터 디버깅 명령을 받아서 Emulator Board에 전달하는 기능과 Emulator Board로부터 OCD의 디버깅 정보를 읽어서 GDB에 보내주는 기능을 한다. GDB와 RDA사이는 RSP 프로토콜로 디버깅 명령과 디버깅 데이터를 주고받는다. RDA의 기본적인 동작은 그림 20에서와 같이 RSP 패킷의 첫 번째 ASCII코드를 읽어 들여서 어떤 디버깅 명령인지 분류하고, 나머지 패킷에서 디버깅 정보를 포함하고 있는 packet-data부분을 디버깅 명령에 해당하는 동작을 하는 함수의 인자로 전달해서 호출하는 방식으로 동작한다. RDA에서 디버깅 명령에 따라 호출되는 각각의 함수는 그림 20에서와 같이 레지스터 읽기/쓰기, 메모리 읽기쓰기, 브레이크 포인터 설정, 싱글스텝 등의 함수들로 구성되어 있다. RDA는 고속 디

```
switch (packet[0])
{
 case 'g':
   gdb_get_registers_packet(packet, packet_size);
        break;
 case 'G':
   gdb_set_registers_packet(packet, packet_size);
        break;
 case 'p':
   gdb_get_register_packet(packet, packet_size);
        break;
 case 'P':
   gdb_set_register_packet(packet, packet_size);
        break;
 case 'm':
   gdb_read_memory_packet(packet, packet_size);
        break;
 case 'M':
   gdb_write_memory_packet(packet, packet_size);
        break;
 case 'z':
 case 'Z':
   gdb_breakpoint_watchpoint_packet(packet,
packet_size);
        break;
 case '?':
   gdb_last_signal_packet(packet, packet_size);
         break;
 case 'c':
 case 's':
   gdb_step_continue_packet(packet, packet_size);
        break;
 case 'D':
   gdb_resume(resume_type);
        break;
 default:
        break;
}
```

그림 20. RDA에서 디버깅 명령어에 따라서 디버깅 함수를 호출하는 부분

Fig. 20. The part of RDA calling related function according to debugging command.

버깅을 위해서 PC외부의 Emulator Board와 USB2.0 프로토콜로 통신을 한다.

2. 하드웨어 인터페이스 모듈

디버깅 시스템은 비동기적으로 동작함에 따라서 디버깅 속도가 느려질 수밖에 없는 구조이다. 그 중에서 동작하는데 가장 많은 시간이 걸리는 부분이 디버깅 명령에 따라서 OCD로직을 제어하는 부분인 JTAG Signal Generator 부분이다. 한 비트 씩 입출력하는 동기 직렬 통신 방식인 JTAG 프로토콜을 적용하는 OCD블록을 포함하는 32비트 프로세서의 경우에 고속으로 디버깅하기 위해서는 고속으로 JTAG 신호를 생성할 수 있는 고성능 프로세서를 포함하는 독립적인 블록이 필요하게

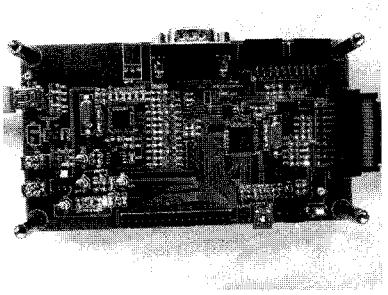


그림 21. 하드웨어 인터페이스 보드

Fig. 21. Hardware interface board.

된다. 실제 대부분의 상용 디버그 시스템에서도 이와 같은 한계점을 극복하기 위해서 Multi-ICE, TRACE32, OPENICE-A1000등과 같이 PC 외부에 제어블록을 따로 구성해서 디버그 시스템을 구현하고 있다.

소개하는 OCDS 또한 고속 디버깅을 위해서 그림 21과 같은 Emulator Board라는 고속의 JTAG 신호를 생성하는 보드를 구현하였다. Emulator Board는 PC와의 고속 데이터 통신을 위해서 USB2.0을 지원하는 USB칩과 고속의 JTAG 신호를 생성하기 위해서 고성능 프로세서를 포함하는 형태로 구현하였다. Emulator Board는 TCK 클록을 최대 50MHz까지 생성할 수 있다.

VII. 검 증

구현한 디버깅 시스템을 RTL형태로 구현된 32비트 RISC타입 프로세서(Core-A)에 적용해서 FPGA 레벨에서 검증하였다.

설계한 OCD블록을 그림 22와 같이 대상 프로세서에

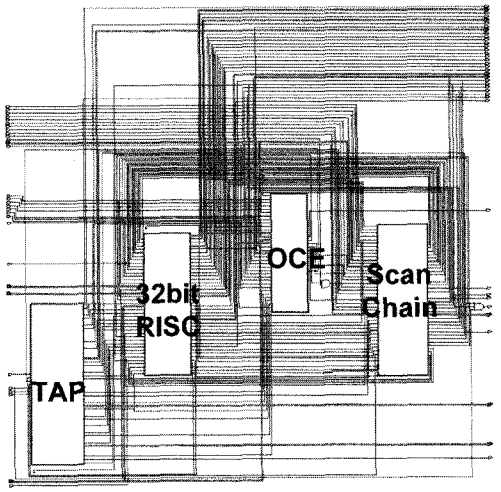


그림 22. OCD를 내장한 대상 프로세서

Fig. 22. Target processor integrating OCD.

인터페이스하여 시뮬레이션 검증을 하고 FPGA로 구현하였다. 그리고 대상 프로세서용으로 소프트웨어 디버거(GDB)와 인터페이스 블록을 구현하고 각각의 블록들을 인터페이스 하여 디버거를 구현하였다. ADPCM과 MP3등 다양한 C 코드를 구현한 디버거와 상용 ARM 디버거(AXD:ARM eXtended Debugger)에 각각 컴파일하고 다운로딩해서 구현한 디버거의 디버깅 동작이 심볼릭 레벨에서 상용 디버거의 동작과 일치하는 것을 검증하였다.

Ⅷ. 결 론

본 논문에서는 칩 내부에 디버깅 기능을 내장하는 방식(On-Chip Debugger)을 적용한 프로세서의 디버깅 시스템을 설계하였다. 설계한 디버깅 시스템은 HDL형태로 구현한 OCD블록과 GDB 기반의 소프트웨어 디버거, 그리고 소프트웨어 디버거와 OCD를 연동시켜서 칩 내부의 상태를 C/Assembly 레벨에서 디버깅이 가능하게 하는 인터페이스 & 컨트롤 블록으로 구성되어 있다.

구현한 디버깅 시스템은 프로세서 디버거가 만족해야 하는 재사용성 또한 만족하고 있다. 설계한 OCD블록은 코어의 어드레스버스와 데이터 버스를 공유하는 방식으로 코어 외부에서 독립적인 형태로 구성되기 때문에 다른 종류의 프로세서에도 적용 할 수 있는 구조이다. 소프트웨어 디버거와 인터페이스 & 컨트롤 블록 또한 아키텍처 의존적인 부분의 수정 및 보완으로 새로운 대상 프로세서의 디버거에 적용 될 수 있다.

제안하는 디버깅 시스템은 기본적으로 소프트웨어 개발을 위해서 사용될 수 있을 뿐만 아니라, 프로세서 개발하는 과정에서 프로세서 자체의 검증용으로도 사용 될 수 있다. 7장에서 소개하였듯이 FPGA로 구현한 OCD가 내장된 프로세서를 소프트웨어 디버거와 연동해서 C/Assembly 레벨에서 프로세서를 검증할 수 있었다. 또한 OCD가 내장된 프로세서에 내장된 상태에서 PLI(Prog ram Language Interface)를 이용하여 소프트웨어 디버거와 HDL 시뮬레이터를 연동하면 복잡한 시뮬레이션 창 대신에 소스레벨에서 프로세서를 검증할 수 있다.

설계한 디버깅 시스템은 실제 32비트 RISC 프로세서에 적용해서 디버깅 환경을 구현 하고 검증해 봄으로해서 설계의 타당성을 검증하였다.

참 고 문 헌

[1] MacNamee, C.; Heffernan, D., "Emerging on-ship debugging techniques for real-time embedded systems", Computing & Control Engineering Journal, Volume 11, Issue 6, pp.295 - 303, Dec. 2000.

[2] 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.

[3] Rainer Kress, Andreas Pyttel, "Debugging Application-Specific Programmable Products", Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications, Lecture Notes In Computer Science; Vol. 1673 pp.481-486, 1999.

[4] David R. Hanson and Mukund Raghavachari., "A machine-independent debugger. In Software-Practice and Experience, volume 26, pp. 1277-1299, November 1996.

[5] Kiyokuni Kawachiya and Takao Moriyama. "A Symbolic Debugger for PowerPC-Based Hardware, Using the Engineering Support Processor (ESP)", In IBM Research, Tokyo Research Laboratory. August, 1997.

[6] 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.

[7] Jundi, K., Moon, D., "Monitoring techniques for RISC embedded systems", Aerospace and Electronics Conference, 1993, vol.1, pp.542 - 550, May 1993.

[8] Eur Ing Chris Hills BSc(Hons), C. Eng., MIEE, FRGS, "Microcontroller Debuggers - Their Place In The Microcontroller Application Development Process" second edition, JAVA C & C++ Spring Conference Oxford Union, Oxford UK, April 1999. http://www.hitex.co.uk.

[9] ARM7TDMI Specification, DDI0210B, http://www.arm.com.

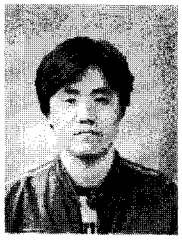
[10] ETM(Embedded Trace Marcocell) Specification, Architecture Specification IHI 0014N, http://www.arm.com.

[11] EJTAG Specification, MD00047, July 5, 2005. http://www.mips.com.

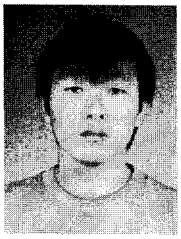
[12] PDtraceTM Interface Specification, MD00136, May

14, 2003. http://www.mips.com.
[13] C166S On Chip Debug Support, August 2001. http://www.infineon.com
[14] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format(ELF) Specification", Version 1.2, May 1995
[15] W. Qin and S. Malik. "Architecture Description Languages for Retargetable Compilation", In *The Compiler Design Handbook: Optimizations and Machine Code Generation.* CRC Press, 2002.
[16] J. Arceneaux, M. Tiemann, D. V. Henkel-Wallace, "The portability of GNU software", In *Proceedings of the Spring 1992 EurOpen/ USENIX Workshop*, pp.89-103, 1992.
[17] Daniel Jacobowitz, "Remote Debugging with GDB",http://www.kegel.com/linux/gdbserver.html, 2002.
[18] Minheng Tan, "A minimal GDB stub for embedded remote debugging", http://www1.cs.columbia.edu/~sedwards/classes /2002/w4995-02/tan-final.pdf, 2002.
[19] Bill Gatliff, "Embedding with GNU: The gdb Remote Serial Protocol." In Red Hat Developer Network (RHDN).
[20] Chen, H.-M., Kao, C.-F. et al, "Analysis of Hardware and Software Approaches to Embedded In-Circuit Emulation of Microprocessors", Proc. of ACSAC, 2002.
[21] Jonathan B. Rosenberg, "How Debuggers Work -Algorithms, Data Structures, and Architecture", Wiley Computer Publishing, 1996.
[22] IEEE Std. 1149.1a-1993, "Test Access Port and Boundary-Scan Architecture", IEEE, Piscataway, N.J., 1993.
[23] 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.
[24] Hubert H¨ogl,Dominic Rath, "Open On-Chip Debugger", http://openocd.berlios.de/web/

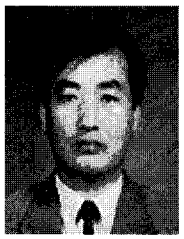
저 자 소 개



박 형 배(학생회원)
2004년 동서대학교 전자공학과 학사
2006년 부산대학교 석사
2008년 현재 부산대학교 전자공학과 박사과정
<주관심분야 : 프로세서 설계, 디버거 설계, 디지털 시스템 설계, SoC 설계>



허 경 철(학생회원)
2005년 중국 연변과학기술대학 전자통신학과 학사
2008년 부산대학교 전자공학과 석사
2008년 현재 부산대학교 전자공학과 박사
<주관심분야 : 프로세서 설계, 디버거 설계, SoC 설계>



박 주 성(평생회원)
1976년 부산대학교 전자공학과 학사
1978년 KAIST 석사
1989년 University of Florida 전자공학 박사
1998년~2006년 부산대학교 IDEC 센터장
1991년~현재 부산대학교 전자공학과 교수
2008년 현재 부산대학교 공과대학 학장
<주관심분야 : DSP 설계, ASIC 설계, 반도체 소자 모델링, 음성/사운드 신호처리 및 구현, SoC설계>

지 정 훈(학생회원)
2003년 경성대학교 컴퓨터공학 학사
2005년 경성대학교 컴퓨터공학 석사
2008년 현재 부산대학교 컴퓨터공학과 박사과정
<주관심분야 : 프로그래밍언어 및 컴파일러, 프로그램 표절검사, 자바가상기계>

우 균(정회원)
1991년 한국과학기술원 전산학 학사
1993년 한국과학기술원 전산학 석사
2000년 한국과학기술원 전산학 박사
2000년~2002년 동아대학교 컴퓨터공학과 전임강사
2002년~2004년 동아대학교 컴퓨터공학과 조교수
2004년~2007년 부산대학교 컴퓨터공학과 조교수
2007년~현재 부산대학교 컴퓨터공학과 부교수
<주관심분야 : 프로그래밍언어 및 컴파일러, 함수형 언어, 그리드컴퓨팅, 소프트웨어 메트릭>