GDB 기반의 재목적 소프트웨어 디버거 설계 및 구현

지정훈[0], 이건우, 우균
부산대학교 컴퓨터공학과
e-mail:jhji@pusan.ac.kr

Design and Implementation of Retargetable Software Debugger based on GDB

Jung-Hoon Ji[0], Gun-Woo Lee, Gyun Woo
Dept of Computer Engineering, Pusan National University

요 약

최근 SoC 기술이 발달하면서, 내장형 시스템을 위한 프로세서 개발이 활발해졌다. 새로운 프로세서가 개발되면, 운영체제 및 소프트웨어 개발을 위해 컴파일러 및 디버거가 필요하다. 컴파일러는 소스코드를 타겟 프로세서에서 실행 가능한 목적파일로 변환하고, 디버거는 프로그램의 개발에서 오류를 찾기 위한 도구로 소프트웨어 개발에 매우 중요한 도구들이다. 본 논문에서는 KAIST에서 개발하는 32bit 프로세서인 Core-A를 위한 소프트웨어 디버거를 설계 및 구현한다. Core-A용 디버거는 공개 소스 디버거 시스템인 GDB를 참조모델로 했으며, 레지스터와 메모리 맵과 같은 프로세서 종속적인 부분을 확장하고 외부 인터페이스 모듈과 같은 프로세스 독립적인 모듈은 재사용함으로써 개발기간을 단축시켰다. 그리고 Core-A용 디버거의 검증을 위해 상용 디버거 시스템인 ARM용 AXD 디버거와 비교 실험을 진행하였다.

1. 서론

최근 SoC(System-on-Chip) 기술에서 가장 핵심적인 분야는 바로 내장형 DSP(Digital Signal Processor)와 마이크로프로세서 코어 분야이다. 그 이유는 대부분의 SoC 형 칩은 DSP나 마이크로세서를 내장하고 있기 때문이다. SoC 설계는 구조 및 RTL 레벨설계, 내장형 프로세서에서 수행될 소프트웨어 개발, 하드웨어 소프트웨어 통합, 설계 검증으로 이루어진다. SoC 성능은 탑재된 소프트웨어에 의하여 결정되기 때문에 내장형 프로세서를 효율적으로 사용하게 하는 지원시스템을 잘 구축하는 것이 SoC 개발의 성패를 결정한다. 하드웨어 측면에서 아무리 성능이 우수한 프로세서 일지라도 프로그램의 소스를 검증하는 디버거, 실제 타겟 시스템(target system)에서 프로세스의 동작을 검증하는 ICE(In Circuit Emulator) 없이는 제 성능을 제대로 발휘하는지 검증 할 수 없을 뿐만 아니라 설계에도 많은 기간이 소요된다.

소프트웨어 디버거는 크게 타겟 지정 기능을 이용하여 다양한 시스템을 지원하는 범용 소프트웨어 디버거와 특정 타겟 시스템에 특화된 내장형 소프트웨어 디버거로 구분 지을 수 있다. 범용 소프트웨어 디버거는 GNU GDB가 대표적인 예이다. 소프트웨어 디버거의 기본적인 기능으로 프로그램의 동작을 정지시키는 breakpoint, 한 명령어씩 수행하는 single step, 레지스터와 메모리를 read/write 기능, 특정 조건일 때 프로그램 수행을 정지시키는 watch-point 기능 등이 있다. 그리고 프로그램 수행에 따른 프로세서 동작을 손쉽게 볼 수 있는 GUI(Graphic User Interface)를 갖추고 있다.

기술 발달로 인해 새롭게 제작되는 프로세스를 위해 매번 새로운 소프트웨어 디버거를 개발한다면 이는 너무 많은 시간과 비용이 소모된다. 이를 해결할 수 있는 방법은 기존에 개발된 디버거를 재목적하는 것이다.

본 논문에서는 성공적인 SoC 개발을 위해 소프트웨어 검증을 위한 재목적 디버거 개발 방법에 대해서 살펴본다. 이를 바탕으로 KAIST에서 개발한 32bit RISC 프로세서인 Core-A를 위한 GDB 기반의 어셈블리 언어 및 C 언어 수준의 디버깅을 지원하는 Core-A 디버거를 설계 및 구현하였다. 그리고 본 논문에서는 구현한 디버거의 검증을 위해 ARM사의 상용 디버거인 AXD 디버거와 비교를 통해 성능을 검증하였다.

본 논문의 구성은 다음과 같다. 2장에서는 디버거 개발에 대한 관련연구에 대해서 살펴보고, 3장에서는 Core-A 디버거를 설계하고 구현한다. 4장에서는 개발된 Core-A 디버거의 검증을 위해서 C 언어로 작성된 ADPCM 코드를 AXD 디버거와 Core-A 디버거로 디버깅하여 그 결과를 비교한다. 그리고 5장에서는 향후 연구와 결론을 기술한다.

2. 관련연구

2.1. GDB 개요 및 구조

GDB[1]는 GNU(GNU's Not Unix) 프로젝트 팀에서 개발한 고성능 디버거로 1988년부터 오픈소스로 운영되고 있으며 오랜 시간동안 안정성과 신뢰성을 인정받은 범용 소프트웨어 디버거이다. GDB는 현재 linux, unix, windows등 다양한 플랫폼에서 사용이 가능하며 C, C++, Object-C 등 많은 언어와 ARM, I386, MPIS등의 프로세스들을 지원하고 있다. GDB의 주요 기능은 다음과 같다[2].

- 프로그램 실행 및 제어 흐름 조절

- 스택, 소스 파일, 데이터, 심볼테이블 검사
- 실행 변경, 디버깅 타겟 지정, 원격 디버깅

GDB는 크게 사용자 인터페이스, 심볼 처리부, 타겟 시스템 처리부 3개의 주요 서브시스템으로 구성된다[3]. GDB의 CLI(Command Line Interface)와 MI(Machine Interface)를 지원한다. DDD나 XXGDB 소프트웨어는 CLI를 기본적으로 사용하며, 이클립스의 경우는 MI를 사용한다. 타겟 시스템 처리부에서 사용하는 주요 요소들은 심볼 처리부에서도 사용된다.

타겟 시스템 처리부는 GDB에서 사용하는 라이브러리 libiberty와 타겟 시스템 정보가 기술된 bfd(binary file descriptor), opcode에 대한 프로세싱을 담당하는 opcodes로 구성된다. 특히 bfd 부분은 타겟 시스템에 종속적인 부분이며 심볼 처리부에서도 사용되는 핵심 부분이다. 그림 1은 GDB에 대한 최상위 수준(top-level) 구조를 보여준다.

[그림 1] GDB 최상위 수준 구조도

2.2. 재목적 디버거(Retagetable Debugger)

과거에는 주로 컴파일러, 에디터와 같은 요소들에만 재목적이 적용되었지만 최근에는 재목적 기술의 발달로 디버거에서 적용되고 있다. ldb는 대표적인 재목적 디버거로 재목적 컴파일러인 lcc와 함께 사용되고 있다[4]. ldb는 gdb와 같은 C 언어 수준 디버거로 MIPS R3000, Motorola 68020, SPARC, VAX 시스템을 지원한다. ldb에서는 재목적을 위한 세 가지 요소를 가지고 있다.

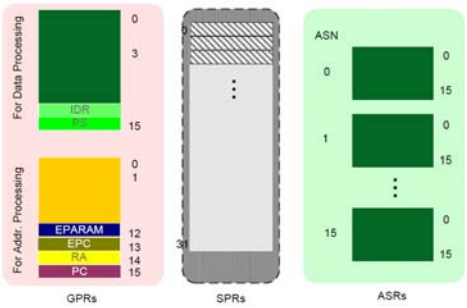
- 컴파일러와의 연동
- 프로세서 종속적인 내장 인터프리터 사용
- 추상화를 통한 디버거 크기 최소화 및 프로세서 종속적인 코드 분리

ldb는 프로세서에 종속적인 부분에 대한 부분을 lcc와 함께 사용할 수 있게 개발되었다. PostScript를 사용하여 lcc가 프로세서에 종속적인 심볼 테이블을 생성하고 이를 ldb에서 사용한다.

ldb와 마찬가지로 GDB도 프로세서에 종속적인 주요 핵심부인 bfd가 GCC에서도 사용되는 부분이기 때문에 이를 응용하여 기존의 GDB를 재목적이 가능하다.

2.3. Core-A 프로세스

Core-A 프로세스는 32bit RISC 구조의 프로세서로 General Purpose Register(GPR), Special Purpose Register(SPR), Application Specific Register(ASP)으로 구성된다. 그림 2는 Core-A 프로세서의 3가지 타입에 대한 레지스터들 구조이다.



[그림 2] Core-A 프로세서 레지스터

GPR은 데이터 및 어드레스 프로세싱을 위해 사용된다. SPR은 프로세서 외부에 위치할 수 있으며, 캐쉬(cache) 또는 메모리 관리 유닛(MMU)에 대한 제어 레지스터들과 맵핑될 수 있다.

3. 디버거 설계 및 구현

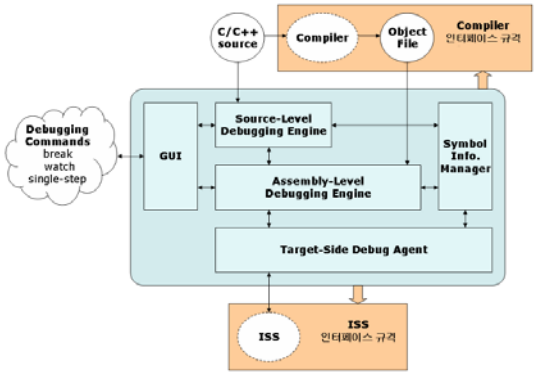
3.1 디버거 요구사항

본 논문에서 구현하는 Core-A용 소프트웨어 디버거 시스템은 크게 시스템 제어, 메모리 추적, 타겟 지정, 원격 디버깅 기능을 지원한다. 시스템 제어는 프로그램이 수행될 때, 사용자가 디버거를 통해 프로그램의 제어 흐름을 멈추게 하거나, 다시 재게 시키는 기능이다. 시스템 제어 기능은 런타임 오류를 검출하기 위한 중요 기능이다. 메모리 추적 기능은 시스템이 사용하는 메모리를 분석하여 사용자에게 보여주는 기능이다. 변수 값 조회 또는 메모리 주소를 통한 데이터 조회 등의 기능을 지원한다. 타겟 지정은 다양한 내장형 시스템 지원을 위한 기능이다. 이 기능은 디버거에서 프로세서 종속적인 코드를 분리함으로써 지원 가능하다. 마지막으로 타겟 지정 기능을 좀 더 발전시킨 기능으로 원격 디버깅 기능을 지원한다. 원격 디버깅은 물리적으로 다른 수행환경과 디버깅 환경 사이에 통신 프로토콜을 이용하여 디버깅 작업을 수행하는 것이다.

그리고 소프트웨어 디버거는 구조적으로 컴파일러, 타겟 시스템과 연동해서 동작한다. 컴파일러와의 연동을 위해 Core-A에서는 ELF(Executable and Linking Format) 형식의 목적파일을 사용한다. 컴파일 과정을 거쳐 생성된 목적파일은 타겟 시스템과 디버거의 입력파일로 사용된다. 그리고 디버거와 타겟 시스템은 JTAG을 통해 연결된다. 디버거는 사용자의 입력을 JTAG용 디버깅 명령어로 변환하고, 디버깅 결과를 GUI를 통해 사용자에게 보고한다. 원격 디버깅의 경우 RSP(Response Serial Protocol)을 이용한다.

3.2 시스템 구조

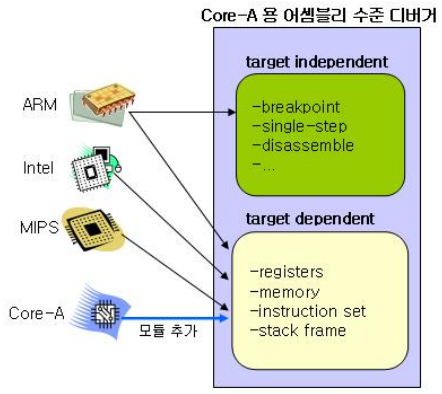
Core-A용 소프트웨어 디버거는 그림 3과 같이 디버깅 엔진(Assembly/Source-Level Debugging Engine), 심볼 관리자(Simbol Info Manager), 타겟 시스템 디버거 에이전트(Target-Side Debug Agent)로 이루어진다.



[그림 3] Core-A용 소프트웨어 구조도

어셈블리 및 C 언어로 작성된 프로그램은 어셈블리 수준 및 소스 수준 디버깅 엔진에 의해 디버깅 된다. 디버깅 엔진은 심볼 관리자를 통해 변수 및 코드에 접근하고 GUI를 통해 사용자로부터 디버깅 명령을 전달 받는다. 그리고 심볼 관리자는 컴파일러가 생성한 ELF 형식의 목적 파일을 분석하여 데이터, 코드, 디버깅 심볼 정보를 디버깅 엔진에 알려준다. 마지막으로 타겟 시스템 디버거 에이전트는 디버거와 프로그램이 실제 수행되는 타겟 시스템을 연결한다. 프로그램 데이터 변경, 레지스터 값 변경, 메모리 감시 등의 디버깅 수행은 디버깅 에이전트를 통해 타겟 시스템에 전달된다.

3.3 디버거 개발방향



[그림 4] Core-A용 디버거 개발 방향

Core-A 용 디버거는 GDB를 참조모델로 하였다. 그리고 프로세서 종속적인 코드와 독립적인 코드를 분리하여 프로세서 종속적인 코드 부분을 Core-A에 맞도록 확장하였다. 새로 개발된 프로세서에 대해 처음부터 컴파일러 및 디버거를 개발한다는 것은 많은 개발비용과 시간이 필요하다. Core-A용 디버거 개발에서는 GDB에서 프로그래밍 언어와 타켓 시스템에 독립적인 부분은 재사용하고 종속적인 코드만을 새로 작성한다. 그림 4는 Core-A용 디버거 개발 방향을 나타낸다.

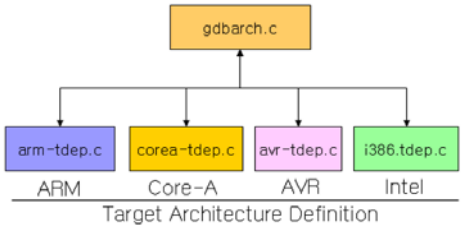
3.4 디버거 구현

1) 타겟 시스템 인식

먼저 Core-A 타겟 정보를 디버거에 추가해 타겟 시스템을 인식시켜야 한다. 타겟에 대한 주요 정보로는 레지스터・메모리 맵, 프로그램 카운터, 스택포인터 등이 있다. 이를 위해 GDB의 Target Architecture Definition 모듈을 확장한다. GDB의 모듈을 기능적으로 구분하면 Handling, UI, libgdb, Symbol, Target Vector Definition, Target Architecture Definition의 6가지 모듈로 나눌 수 있다. GDB에는 ARM, MIPS, Intel과 같이 많이 사용되는 프로세서에 대한 타겟 구조에 대하여 정의되어 있다.

GDB에서 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. 타겟 아키텍처 정보를 추가하기 위해서는 TDEPFILES, TDEPLIBS, SIM에 대한 파일을 맵핑시켜주면 된다. 그리고 GDB를 확장하기 위해서는 프로세서 독립적인 부분과 종속적인 부분을 연결시켜주는 GDBARCH에 정의되어 있는 인터페이스 함수들을 제정의 해야 한다. 그림 5는 GDBARCH와 Target Architecture Definition 모듈에 대한 연결구조이다.



[그림 5] GDB 모듈 연결구조

2) 명령어 셋(instruction set) 인식

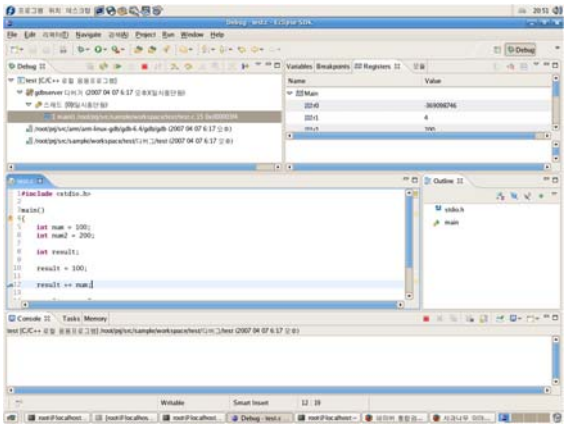
Core-A용 디버거에 명령어 셋을 등록하기 위해서는 opcode 관련 모듈을 수정해야 한다. opcode 모듈은 “~/gdb/opcodes”에 위치한다.

Core-A 명령어 셋은 ARM 명령어 셋과 구조가 비슷하다. ARM의 opcode는 32비트와 16비트 두 구분된다. 하지만 Core-A는 현재 32비트 명령어만을 지원하고 있으므로 32비트 opcode를 사용한다. opcode는 총 4가지 필드(arch, value, mask, assembler)로 이루어진다. arch는 명령어가 사용되는 아키텍처를 나타내고, value는 opcode의 기계어 코드 값을 나타낸다. 그리고 mask는 opcode에 대

한 마스크 값이고, assembler는 명령어에 대한 어셈블러 형식이다.

3) 이클립스를 이용한 GUI 지원

Core-A 용 디버거는 GUI 환경으로 이클립스를 지원한다. 이클립스와 디버거의 인터페이스에는 2.1절에서 언급한 것처럼 MI 방식을 사용한다. MI는 기계 중심적인 구조로 되어 있으며, 디버깅 결과를 파싱하기가 용이하다. 그림 6은 이클립스를 이용한 Core-A용 프로그램 디버깅 화면이다.

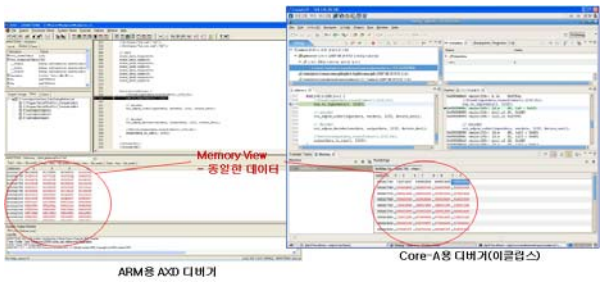


[그림 6] 이클립스를 이용한 디버깅

4. 검증

Core-A 용 디버거를 검증하기 위해 본 논문에서는 ARM용 디버거인 AXD 디버거[5]와 디버깅 과정을 비교하는 실험을 진행하였다. 실험에는 C언어로 작성된 MP3 디코더 프로그램과 ADPCM 코드를 이용하였다. 디버거 검증을 위해 ADPCM 코드를 빌드하고, Core-A용 디버거와 연결된 이클립스에서 breakpoint를 설정한 뒤, 프로그램이 멈추었을 때, 변수 값 및 메모리에 저장되는 결과 값을 확인하였다. 그리고 AXD 디버거를 이용하여 동일한 과정을 반복한 뒤 결과를 비교하였다.

이 실험은 상용 디버거 시스템과 디버깅 결과를 비교함으로써 Core-A용 디버거가 잘 동작하는지 확인하였다. 두 디버거 시스템의 타겟은 다르지만, 디버깅 중간과정에서 저장되는 변수 값 및 변경되는 메모리 값은 동일하게 변해야 한다. 그림 7은 Core-A용 디버거와 AXD 디버거를 이용하여 ADPCM 코드를 디버깅한 스크린 샷이다.



[그림 7] 디버거 검증

Core-A용 디버거와 AXD용 디버거를 이용하여 동일한 주소의 메모리 값을 조회했을 때, 두 디버거에서 같은 값을 나타내었다. 이는 두 디버거 시스템의 동작이 동일하다는 것을 나타낸다.

5. 결론 및 향후 연구

본 논문에서는 새로 개발되는 내장형 프로세서에 대한 GDB 기반의 어셈블리 및 C 소스 수준의 디버거를 설계 및 구현하였다. 내장형 프로세서의 빠른 개발주기에 맞추어 디버거를 개발하기 위해 디버거에서 프로세서 독립적인 모듈과 종속적인 모듈로 나누고, 독립적인 모듈은 재사용하고 종속적인 모듈을 프로세서에 맞게 레지스터 및 메모리 맵과 같은 Core-A에 대한 타겟을 인식시켰으며, 명령어 셋을 추가하였다. 그리고 사용자 인터페이스를 위한 GUI로 이클립스를 사용하였다. 마지막으로 디버거의 검증을 위하여 상용 디버거 시스템인 AXD 시스템과 비교 실험을 진행하였다. GDB는 공개 소스용 디버거 시스템이기 때문에 개발 비용을 줄일 수 있으며, 또한 확장이 용이하므로 새로운 프로세서를 위한 디버거의 참조모델로 적합하다.

향후 연구로는 JTAG을 이용한 원격 디버깅 모듈 개발을 진행할 것이다. JTAG용 하드웨어 디버거와 연동하여 원격에서 타겟 시스템의 부트로더와 운영체제, 그리고 간단한 프로그램을 디버깅 할 수 있는 소프트웨어 디버거 개발을 진행할 것이다.

참 고 문 헌

[1] John Gilmore, Stan Shebs, Cygnus Solution, "GDB Internal Manual: A guide to the internals of the GNU debugger", GDB version 6.4. Technical report, Free Software Foundation, Cambridge, MA.

[2] 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.

[3] J. Arceneaux, M. Tiemann, D. V. Henkel-Wallace, "The portability of GNU software" Proceedings of the Spring 1992 EurOpen/USENIX Workshop, pp. 89-103, EurOpen, 1992.

[4] Norman Ramsey, David R. Hanson, "A Retargetable Debugger", ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, in SIGPLAN Notices, 27(7):22 - 31, July 1992

[5] RealView Development Suite AXD Debugger, "http://www.arm.com/products/DevTools/AXDDebugger.html"