PICO 프로세서를 위한 GDB 포팅[1]

이상희, 이호균, 나여울, 김선욱
고려대학교 전기전자전파공학과
e-mail: {lshron, hokyoon79, rapidsna, seon}@korea.ac.kr

GDB Porting for PICO Processor

Lee Sang Hee, Hokyoon Lee, Yeoul Na, Seon Wook Kim
School of Electrical Engineering, Korea University

요 약

GDB 는 프로그램의 버그를 찾고 수정할 수 있는 기능을 포함하고 있는 디버거로써 소프트웨어 개발자가 효과적인 프로그램 개발과 유지를 위해 꼭 필요하다. 그러나, PICO 프로세서와 같이 새로 개발된 프로세서는 그 위에서 동작하는 프로그램을 디버깅 하기 위한 환경을 갖추고 있지 않다. 이 논문은 PICO 프로세서를 위한 GNU 디버거인 GDB 와 그래픽 유저인터페이스를 제공하는 DDD 를 포팅하는 방법에 대해 소개하고 설명한다.

1. 서론

GDB[1]는 오픈 소스 디버거로 많은 소프트웨어 개발자들이 디버깅에 사용하는 개발도구이다. 최근 스마트폰을 포함한 임베디드 기기의 수요 급증, multi-core 에서 many-core 로의 전환, Green IT 로 인한 전력소비문제 등 프로세서 분야의 패러다임이 변화하고 있어, 이러한 각각의 요구에 최적화된 새로운 ISA 를 갖는 프로세서의 수요가 늘어날 것으로 보이며, 이에 따라 GDB 포팅 기술의 수요도 더욱 늘어날 전망이다. GDB 에 그래픽 유저 인터페이스 (GUI)를 지원한다면 더욱 편리한 디버깅 환경을 구축할 수 있다. 이 논문은 PICO 프로세서[2]와 같은 새로운 아키텍처를 위해 개발된 프로그램을 디버깅 하기 위한 GDB 포팅과, 그래픽 유저 인터페이스를 지원하기 위한 DDD[3] 포팅에 대한 방법을 설명하는 논문이다.

이 논문의 구성은 다음과 같다. 2 장은 GDB 포팅에 대한 구성요소에 대한 설명과 구현방법을 다룬다. 3 장에서는 DDD 포팅에 대한 설명과 구현을 다룬다. 장은 GDB 와 DDD 의 실제 동작화면을 제시하고, 5 장에서 결론을 맺는다.

2. GDB 포팅

새로운 target 아키텍처를 위하여 GDB 에 구현되거나 추가되어야 할 것은 BFD(binary file descriptor), disassembler, 시뮬레이터와 디버깅에 필요한 동작과 정보를 정의한 target dependent file 이다. BFD, disassembler 는 GNU Binutils[4] 포팅과정에서 개발이 완료되어 있기 때문에, 이 논문에서는 GDB 에 필요한 target dependent file 과 시뮬레이터를 구성하는 것에 대해 다룬다.

2.1 Target dependent file

Target dependent file 은 target dependent 한 환경에서 개발된 프로그램의 정보를 분석하기 위한 것으로, 레지스터 정보 제공, breakpoint 설정, frame analysis 하는 함수들과 그 밖에 target dependent 한 정보들로 이루어진다.

이러한 target dependent file 의 구성요소들은 GDB 가 시작할 때, GDB 아키텍처에 등록된다. GDB 아키텍처란 GDB 가 디버깅을 하는데 필요한 함수와 환경이 등록되는 자료구조이다.

(1) 레지스터 정보 제공

GDB 가 레지스터를 인식할 수 있도록 GDB 가 다루는 레지스터 번호와 일치하는 실제 이름, 자료형, 역할, 그리고 레지스터의 개수에 대한 정보를 GDB 아키텍처에 등록한다.

위와 같은 레지스터 정보를 GDB 에 제공하기 위하여, 우리는 PICO 프로세서의 레지스터 이름을 {“R0”, “R1”}과 같이 배열로 저장하였다. 총 등록한 레지스터의 개수는 30 개 이며, 자료형은 32bit unsigned 이다. 프로그램 카운터 레지스터의 번호는 23, 스택 포인터 레지스터의 번호는 16 으로 하였다.

GDB 는 이렇게 등록된 레지스터 정보를 이용하여 simulator 의 레지스터의 값들을 읽거나 쓸 수 있게 된다.

[1] 본 논문은 산업자원부가 지원하는 서울시 산학연협력사업(10560, 10920)의 지원 하에 개발된 결과임을 밝힙니다.

(2) Breakpoint 설정

PICO 디버거는 소프트웨어 breakpoint 를 사용한다. GDB 에서 소프트웨어 breakpoint 를 지원하도록 하기 위해 GDB 가 제공하는 함수를 이용하여, breakpoint address 에 삽입할 trap instruction 을 GDB 아키텍처에 등록한다. PICO 프로세서의 trap instruction 의 opcode 는 {0x6b, 0x00}이고, 이 opcode 가 시뮬레이터에 의해 실행되면 시뮬레이터는 동작을 멈추고 GDB 에 컨트롤이 넘어가게 된다.

(3) Frame analysis

Frame analysis 는 스택이 가지고 있는 정보를 분석할 수 있도록, 스택의 크기, caller 의 스택 포인터, 스택에 push 된 레지스터의 값을 GDB 에 제공한다.

Frame analysis 는 프레임의 prologue 에 위치한 push/add instruction 을 분석하는 것부터 시작한다. 이것은 prologue 의 instruction 들을 직접 읽어서 각 instruction 들의 operand 를 직접 분석하여 스택의 크기와 레지스터의 위치를 찾는 것이다. 분석 대상이 되는 instruction 은 push 와 adds 이다. 두 instruction 이 변화시키는 스택의 크기와 현재 스택 포인터를 이용하여 caller 의 스택 포인터를 찾고, 레지스터들이 push 된 상대적인 위치와 현재 스택 포인터를 이용하여 push 된 레지스터 값을 찾는다.

Prologue 의 분석이 끝나면 이 분석을 이용하여 스택의 크기, caller 의 스택 포인터, push 된 레지스터에 대한 정보로 이루어진 자료구조를 만들고, 이것을 이용하여 GDB 에 필요한 자료를 제공한다.

(4) 그 밖에 target dependent 한 정보

위의 정보 이외에 PICO 프로세서의 target dependent 한 정보는 PICO 아키텍처가 little endian 이고, 스택은 위에서 아래로 성장하며, 함수 리턴 방식으로 레지스터를 이용하는 것이다. 이 정보들은 앞에서 설명한 다른 target dependent 한 정보들과 마찬가지로 디스어셈블리 함수와 함께 GDB 아키텍처에 등록했다.

2.2 시뮬레이터

PICO 디버거는 시뮬레이터를 이용한다. PICO 디버거가 시뮬레이터를 이용하기 위해서 시뮬레이터가 소프트웨어 breakpoint 를 지원해야 하며, PICO 디버거가 시뮬레이터에 있는 레지스터와 메모리의 데이터를 read/write 할 수 있어야 한다.

(1) Breakpoint

소프트웨어 breakpoint 를 지원하기 위해서 시뮬레이터가 trap instruction 을 실행할 때, trap signal 을 발생시키는 trap instruction 함수를 구현했다. Trap signal 이 발생하면 시뮬레이터는 멈추고 컨트롤은 PICO 디버거로 넘어간다.

(2) 메모리 & 레지스터 read/write

시뮬레이터는 레지스터, 메모리의 데이터 구조를 포함하고 있으며, GDB 에서는 그 시뮬레이터의 레지스터와 메모리의 값을 읽고 쓰기 위한 함수가 필요하다. 이 기능을 지원하기 위해, 시뮬레이터에 추가로 구현한 함수는 다음과 같다.

sim_read/write(): 메모리의 주소와 크기를 입력 받아 메모리의 값을 buffer 를 이용하여 읽거나 쓰는 함수.
sim_fetch_register(): 레지스터의 번호를 입력 받아 해당 레지스터 값을 buffer 에 쓰는 함수.
sim_store(): 레지스터의 번호를 입력 받아 buffer 의 값을 해당 레지스터에 쓰는 함수.

3. DDD porting

DDD 는 그래픽 유저인터페이스를 제공하며, GNU 디버거를 실행한다. PICO 프로세서 DDD 포팅을 위해 아키텍쳐 이름을 등록하고, 2 장에서 개발한 PICO 프로세서 GDB 를 실행 할 수 있도록 ku32-elf-gdb 프로그램을 디폴트 디버거로 설정한다.

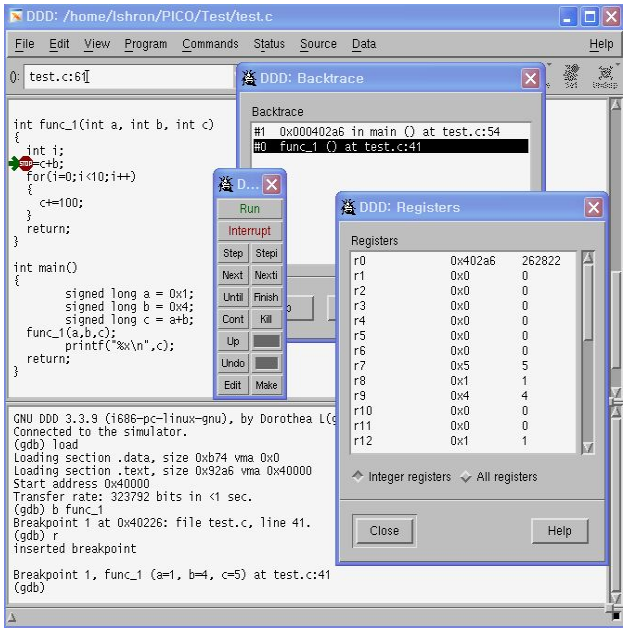
4. 구현 결과

```
[lshron@localhost Test]$ ku32-elf-gdb test.elf
GNU gdb 6.8
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=ku32-elf"...
(ku32-gdb) target sim
Connected to the simulator.
(ku32-gdb) load
Loading section .data, size 0xb78 vma 0x0
Loading section .text, size 0x92ee vma 0x40000
Start address 0x40000
Transfer rate: 324400 bits in <1 sec.
(ku32-gdb) b func_1
Breakpoint 1 at 0x402b2: file test.c, line 59.
(ku32-gdb) r
Starting program: /home/compiler/lshron/PICO/Test/test.elf
inserted breakpoint

Breakpoint 1, func_1 (a=1, b=3, c=4) at test.c:59
59              int i=0;
(ku32-gdb) where
#0  func_1 (a=1, b=3, c=4) at test.c:59
#1  0x00040250 in main () at test.c:46
(ku32-gdb) n
60              c=a*b;
(ku32-gdb) print b
$1 = 3
(ku32-gdb)
```

(그림 1) GDB 실행 화면.

그림 1 은 리눅스환경에서 PICO 프로세서 디버거를 실행한 결과이다. ‘target sim’명령을 통하여, 시뮬레이터와 GDB 를 연결하고, ‘load’ 명령으로 디버깅할 실행파일을 시뮬레이터의 메모리로 로딩하였다. 또한, break point 를 설정하고, ‘run’으로 시뮬레이터를 동작시켰을 때, break point 지점에서 멈추는 것을 확인할 수 있으며, 함수의 콜 스택을 backtrace 한 결과와 변수 값을 정상적으로 출력됨을 확인 할 수 있다.



(그림 2) DDD 실행화면.

그림 2 는 PICO 프로세서로 포팅한 DDD 를 실행한 결과이다. DDD 의 그래픽 인터페이스를 통해, breakpoint 의 위치, 함수의 호출 스택, 레지스터의 값을 사용자가 확인할 수 있다.

5. 결론

새로 개발된 프로세서에서 동작하는 프로그램을 디버깅하기 위한 환경을 가지고 있지 않기 때문에, 소프트웨어 개발자들은 프로그램을 개발하고 유지하는 것에 많은 어려움을 겪는다. 이 논문에서 우리는 PICO 프로세서의 GDB 와 DDD 를 포팅하는 방법을 소개하였다. 이 논문에서 제시한 방법으로 개발자들은 새로운 프로세서가 개발되었을 때 쉽게 디버거를 개발할 수 있으며, 그래픽 유저 인터페이스로 쉽고 편한 디버깅 환경을 구축 할 수 있다.

참고문헌

[1] GDB. http://www.sourceware.org/gdb/
[2] Jiho Cho, Yeoul Na, Jongwook Shoh, Moongi Seok, Jungwon Kang Seon Wook Kim, Implementation and Verification of PICO Processor, 제 16 회 반도체 학술대회, CDC-20, 2009.
[3] DDD. http://www.gnu.org/software/ddd/
[4] Binutils.
http://www.gnu.org/software/binutils/