分类号 ________________ 密级 ________________

UDC ________________ 编号 ________________

南京大学
学士学位论文

GNU Debugger 的移植及原理分析

杨文博

指导教师 吴朝阳 副教授

南京大学数学系

申请学位级别 学士 学科专业名称 信息与计算科学

论文提交日期 2007年5月 论文答辩日期 2007年5月

培养单位 南京大学数学系

学位授予单位 南京大学

答辩委员会主席 ________________

Porting and Analysis of GNU Debugger

Wenbo Yang

Supervisor:

A/Prof. Zhaoyang Wu

Department of Mathematics
Nanjing University

May, 2007

*Submitted in total fulfilment of the requirements for the degree of B.S.
in Information and Computing Science*

摘　要

当前嵌入式系统领域处在一个飞速发展的阶段，各种各样功能和性能的芯片层出不穷，和芯片相关的跨平台开发工具链的研发也自然有着很迫切的需求。为了提升芯片相关软件设计的效率，并且降低开发上的成本，很多企业选择采用移植开源工具链来获得最好的效果。

而调试器是其中不可缺少的一环，当把工具链提供给程序设计人员使用时，有一个功能强大使用方便的调试器是保证程序员开发出工作正常的程序和快速找出程序中缺陷的必要条件。GNU project 提供了包括编译器、汇编器、链接器和调试器的跨平台开发工具链。GNU 的源码级调试器 GNU Debugger 可以工作在多种平台上，并且有通用的机器语言接口（MI,mechine interface），可以集成在 Eclipse，DDD，Insight 等多种集成开发环境中。

本文分析了 GDB 内部的算法逻辑，指出了移植的关键之处，并给出移植的详细方法。为了验证我们方法的正确性，我们对北京简约纳公司生产的基带通信和视频编解码芯片进行移植，结果显示我们的方法能快速准确地为嵌入式芯片开发一个正确的跨平台调试器。

关键词： 跨平台工具链，调试器，GDB，移植

Abstract

Embedded system industry is growing rapidly. More and more chips with different functions and performances are developed. So the related cross-platform development toolchains are under urgent demands. To upgrade the chip software design efficiency and cut the costs of development, many enterprises opt for porting the open-source toolchain to persue the maximum benefit.

Debugger is an indispensable part of the tools. When provide a cross-platform development toolchain to programmers, a powerful and easy-to-use debugger can help programmers to develop right programs and identify the bugs quickly. GNU project provids a toolkit with compiler, assembler, linker and debugger. GNU source-level debugger GDB can work in a variety of platforms, with a common MI(machine interface). GDB can be integrated into the Eclipse, DDD, Insight, and other IDEs(integrated development environment).

This thesis analyzes the GDB internal algorithms, points out the key parts and gives the detailed methodology for porting GDB. In order to verify the validity of our approach, we ported GDB to the base-band communication and video codec chips of Beijing Simplight Nanoelectronics. The results show that this method can quickly and accurately develop a correct cross-platform debugger for embedded chips.

Keywords: Cross-platform toolchain, Debugger, GDB, porting, retarget

目　　录

摘要 i

Abstract ii

目录 iii

第一章　GNU debugger 介绍 1

1.1 调试器 1

1.2 GNU Debugger 1

1.3 移植 GDB 的优点 2

第二章　关于 GDB 的一些重要概念 4

2.1 主机、目标机和跨平台开发工具链 4

2.2 符号和行信息 4

2.3 Call Stack and Stack Frame 6

2.4 远程调试 8

第三章　GDB 的内部算法 10

3.1 GDB 的启动 10

3.2 Frame 模型 10

3.3 Prologue 分析 11

3.4 断点处理 11

3.5 单步执行 12

3.6 信号处理 12

3.7 线程处理 13

3.8 内部函数调用 13

3.9 长跳转支持 14

3.10 观察点 14
3.11 检查点 15

第四章　移植 GDB 涉及到的文件 **16**

第五章　GDB 后端的编写 **18**

5.1 注册新后端到 GDB 中 18
5.2 设置目标架构基本信息 19
5.3 设置寄存器相关信息 19
5.4 设置 `stack frame` 相关信息 20
5.5 单步和断点 22
5.6 虚拟函数调用 22
5.7 屏幕输出相关 23
5.8 小结 23

参考文献 **24**

致谢 **25**

插　　图

2.1 Stack frame 6
2.2 Remote Debugging 8

第一章　GNU debugger 介绍

1.1　调试器

调试器是测试和调试计算机程序的一种软件，可以帮助程序员快速地定位程序错误并完成更改和验证。所以在为嵌入式芯片开发工具链时，调试器是不可缺少的一环，一个好的调试器能大幅度减少程序员修改程序错误的工作量，提高程序开发的效率和正确性。

在进行芯片设计和为它开发相应的工具软件的时候，一般情况下调试器的开发是在汇编器、连接器等二进制工具和编译器开发大致完成时进行。因为在编译工具开发初期可以分析二进制文件和使用模拟器对程序的正确性进行验证，但随着测试程序规模的增大和编译优化功能的完善，这时候仅仅靠分析二进制代码这种原始的方法验证程序的正确性已经无法满足程序开发的需要。而且调试器还有一个作用是可以验证编译工具的正确性。因为编译器在编译程序时一般都有调试选项和优化选项，在发布软件的时候会使用优化选项来得到精简的可执行文件，但在开发过程中，一般会使用调试选项来取得更多的调试信息以便寻找错误。使用调试选项时会向可执行文件中插入大量调试信息供调试器使用，所以这些调试信息的正确性只能由调试器进行检验。因此能够快速地为芯片开发一个功能完善行为正确的调试器是保证芯片设计和产业化进度的必要条件。

1.2　GNU Debugger

GNU Debugger，简称为 GDB，是一个功能强大且全面的 GNU 开源调试器。它使用字符界面，可以用来调试 C，C++，Pascal，Fortran 和一些其它语言。它可以根据不同的配置参数，在不同主机上编译生成不同的调试器以支持不同的计算机体系架构和目标文件格式。它支持 20 多家厂商的 90 多种芯片，其中包括 Intel，IBM，AMD，ARM，MIPS 和 SUN 等知名厂商的许多主流芯片。在目标文件格式方面，它支持 a.out，ECOFF，XCOFF，PE，ELF 和 SOM 等多种目标文件格式，以及 STABS，COFF，DWARF，DWARF2 和 SOM 等多种调试信息格式。

GDB 的当前版本是 6.6 版，官方网址是：http://sourceware.org/gdb，可以从官方网站下载到 GDB 各个版本的源代码和官方使用和开发文档。

1.3　移植 GDB 的优点

采用移植 GDB 的方法比从头开始写一个新的调试器有以下一些优点：

(1)采用移植 GDB 的方法可以利用 GDB 中与具体芯片无关的代码，如各种目标文件的支持代码，读取调试信息的代码，设置断点，观察点等相关的代码，还包括 GDB 的机器语言接口 (MI，Machine Interface) 等。这样能大大的减少开发一个新调试器的工作量，而且可以降低开发门槛。因为这样的话调试器开发者只用略微了解其它部分的功能，专注于某个方面的代码，而不用对各个方面的知识都得有很深的了解。

(2)从使用者角度来说，降低学习难度和周期，使用统一的命令接口。在一个计算机体系架构上使用 GDB 与在其它的计算机体系架构上的使用是完全类似的，甚至可以说是基本相同的。这些相同点包括各种调试命令，输出调试信息的格式等。这样能使程序员避免重新学习另外一个开发工具，提高开发效率。而且 GDB 也已经有了广泛的用户基础，并且能和多种集成开发环境 (IDE，Intergrated Development Environment) 集成，例如 DDD，Insight 和当前非常流行的 Eclipse，这样能够降低程序员的学习门槛，缩短学习周期，也能相应地降低客户公司培训程序员的成本。

(3) GDB 与 GNU 的其它开发工具，如 GNU 的编译器 GCC、汇编器 GAS、链接器 LD，能够紧密地结合在一起，形成完整的跨平台开发工具链。由于 GNU 标准的开放性，所有人都可以更改软件的源文件，就算在前端使用其它的编译器，比如 Open64 等，只要产生 GNU 能够识别的标准格式中间文件，结合 GNU 工具链的其它工具也可以非常好地完成整个工具链的构建。

(4)由于不需要重新写芯片无关的代码，为一个新的芯片写一个编译器的时间可以大大地缩短。重新编写一个调试器需要一个团队在相当长的时间内完成，而我们的实践表明移植 GDB 则完全可以由一个人在一个半月内完成，如果根据本文的实践经验再进行开发，整个开发工作可以缩短到一至两个星期，节省的时间和成本可以说是显而易见的。

目前，GDB 官方只有两个关键文档：《 Debugging with GDB 》[1]和《 GDB Internals 》[2]。前者是 GDB 的使用手册，主要介绍使用 GDB 调试程序的方法；

后者是给开发人员作为参考，介绍 GDB 内部工作机制的文档。但是后者的更新不能跟上 GDB 的发展，而且有很多关键章节是空的，所以不能完全依赖它来了解 GDB 所有的内部机制和解决移植 GDB 所遇到的难题。本文是对它的一个补充，着重描述如何基于 GDB 为某种芯片快速开发一个正确和稳定的调试工具。

第二章　关于 GDB 的一些重要概念

要想移植 GDB，必须得了解一些和移植相关的概念，下面我们介绍一些和 GDB 移植相关的重要概念：

2.1　主机、目标机和跨平台开发工具链

主机（host）和目标机（target）是跨平台编译和调试中非常重要的两个概念，是构建跨平台开发工具链的方法论的基础。在这里，主机和目标机都是指某种计算机体系架构。

在跨平台开发工具链中，目标机一般是指某种嵌入式系统（Embeded System），即工具链编译出的可执行程序运行的平台。但是在通常情况下嵌入式系统的性能都比较低，有的甚至连操作系统都不支持，所以大部分嵌入式系统都没有足够的处理能力和存储空间满足程序开发人员在其上开发程序的需要，就需要将程序在其它的平台上编译成功之后下载到目标机上运行。主机就是指这种有足够处理能力和存储空间，能够运行大规模开发工具进行软件开发的计算机，一般是工作站或者个人电脑（PC），但它的计算机体系架构和目标机又不一样。为了克服嵌入式系统资源的限制，又解决主机和目标机计算机体系架构不同的问题，通常情况下采用的方法是在主机上使用专门的开发工具生成可以在目标机上运行的二进制代码，编译成功之后再通过某些方式将可执行程序文件下载到目标机的内存上运行。调试的时候，通过主机上运行的调试器和目标机(一般是专门的开发板)进行交互式的通信来完成对程序状态的检测和对程序运行的控制。运行在主机上的开发工具集一般包括：编译器、汇编器、连接器、调试器还有库函数，它们与模拟器放在一起，被称为跨平台开发工具链。

2.2　符号和行信息

符号和行信息（SAL，symbol and line）是指二进制文件中为调试器提供的一些信息，在本文中可以看作是 GDB 能从程序二进制文件中直接得到的东西，即在程序不运行的情况下 GDB 对可执行文件了解多少。由于 GDB 是源码级调试器，在调试过程中，调试者可以直接在函数名，源程序行处设置断点，可以直接

打印源程序中的变量。这样 GDB 就必须知道函数入口的位置，各个符号的地址、偏移量和内容，源程序的行信息，所以 GDB 必须通过某些方式来取得这些信息。一般情况下，这些信息都是 GDB 通过读入可执行的或者链接的二进制文件，对二进制文件进行分析，把对应的结构放入 GDB 内部的一些结构体变量中得到的。然后在运行过程中有需求时，GDB 再从这些信息中寻找与被调试程序相关的部分。

GDB 中有专门的文件来读取具体的目标文件格式，例如："elfread.c/h，coffread.c/h，dwarfread.c/h，dwarf2read.c/h" 等。这些文件中提供了读取二进制文件中存储的对应该文件格式信息的函数接口。不过建立符号表的大部分工作都是通过调用 LIB BFD 中的工具函数来实现的。在 GDB 中定义的这些函数主要做的工作是给 GDB 提供一个接口去初始化、添加、重建和销毁符号表，以及在正确的符号列表里去寻找和解析要访问的符号。

对于使用动态链接库的程序，GDB 会根据目标文件中包含的信息来判断要载入哪些相关的库文件。GDB 中有一个 objfile 结构的列表 object_files 保存这些信息，已经被读入的目标文件会被添加到 symbol_objfile 列表中。GDB 并不总是一下读入所有的目标文件内容，甚至可以说在一般情况下 GDB 不会读入所有的目标文件信息，因为那样的话会增加 GDB 的存储空间消耗和运行时间。GDB 有三种类型的符号表：full symbol tables(symtabs)，partial symbol tables(psymtabs)，minimal symbol tables(msymtabs)。其中 symtabs 包含已经被完全读入的一些符号信息。在程序刚刚载入的时候，它包含的内容是很少的，会随着读入具体的符号信息变得越来越庞大；psymtabs 中包含一些最主要的符号信息，一般也是最重要的符号信息，这个列表以及包含的符号在第一次读取目标文件的时候就会被构建。GDB 在寻找一个 symbol 的时候，会首先在 symtabs 搜索，如果在 symtabs 找不到对应的符号入口，就会转而在 psymtabs 寻找。当在 psymtabs 中找到对应的符号入口之后，就调用 PSYMTAB_TO_SYMTAB 宏，如果 psymtabs 中对应的入口包含了到 symtabs 中对应入口的指针，此宏会直接返回该指针，否则调用相关函数将此符号的所有信息从可执行文件中读入到 symtabs 中去并把它注册到 psymtabs 中该符号对应入口上然后使用；msymtabs 主要包含那些与调试关联比较小的通用符号信息，比如：_init，_start，hlt 这些函数入口的地址，其作用比较简单。

行信息一般是伴随着相应的符号文件一起被读取的，行信息中包含了源程

序文件的位置，以及源程序行和机器指令的对应关系。行信息主要用于断点和单步时候对程序中断执行位置的判断，并提供给源码级调试器源代码打印的信息。

2.3　Call Stack and Stack Frame

在计算机科学中，Call stack 是指存放某个程序的正在运行的函数的信息的栈。Call stack 由 stack frames 组成，每个 stack frame 对应于一个未完成运行的函数。

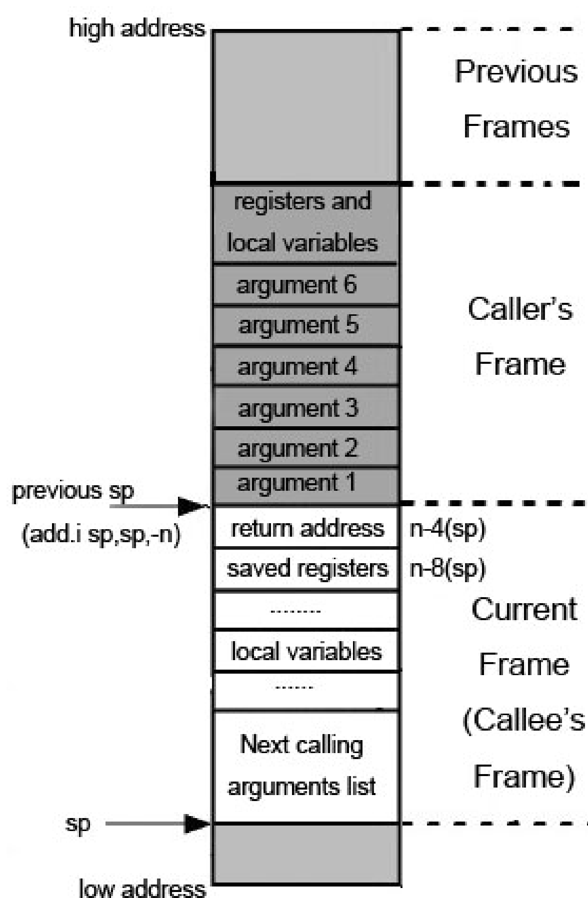

图 2.1: Stack frame

在当今流行的计算机体系架构中，大部分计算机的参数传递，局部变量的分配和释放都是通过操纵程序栈来实现的。栈用来传递函数参数，存储返回值

信息，保存寄存器以供恢复调用前处理机状态。每次调用一个函数，都要为该次调用的函数实例分配栈空间。为单个函数分配的那部分栈空间就叫做 stack frame ，也就是说，stack frame 这个说法主要是为了描述函数调用关系的。

Stack frame 组织方式的重要性和作用体现在两个方面：第一，它使调用者和被调用者达成某种约定。这个约定定义了函数调用时函数参数的传递方式，函数返回值的返回方式，寄存器如何在调用者和被调用者之间进行共享；第二，它定义了被调用者如何使用它自己的 stack frame 来完成局部变量的存储和使用。

图 2.1 描述的是一种典型的 (MIPS O32) 嵌入式芯片的 stack frame 组织方式[3]。在这张图中，计算机的栈空间采用的是向下增长的方式， SP(stack pointer) 就是当前函数的栈指针，它指向的是栈底的位置。Current Frame 所示即为当前函数(被调用者)的 frame ， Caller's Frame 是当前函数的调用者的 frame 。每个 frame 中所存放的内容和存放顺序，则由目标体系架构的调用约定 (calling convention) 定义。如图 2.1 所示， MIPS O32 调用约定规定了所占空间不大于 4 个比特的参数应该放在从 $4 到 $8 的寄存器中，剩下的参数应该依次放到调用者 stack frame 的参数域中，并且在参数域中需要为前四个参数保留栈空间；如果被调用者需要使用 $16 到 $23 这些保留寄存器 (saved register)，就必须先将这些保留寄存器的值保存在被调用者 stack frame 的保留寄存器域中，当被调用者返回时恢复这些寄存器值；当被调用者不是叶子函数时，即被调用者中存在对其它函数的调用，需要将 RA(return address) 寄存器 ($31) 值保存到被调用者 stack frame 的返回值域中；被调用者所需要使用的局部变量，应保存在被调用者 stack frame 的本地变量域中。

在没有 BP(base pointer) 寄存器的目标架构中，进入一个函数时需要将当前栈指针向下移动 n 比特，这个大小为 n 比特的存储空间就是此函数的 stack frame 的存储区域。此后栈指针便不再移动，只能在函数返回时再将栈指针加上这个偏移量恢复栈现场。由于不能随便移动栈指针，所以寄存器压栈和出栈都必须指定偏移量，这与 x86 架构的计算机对栈的使用方式有着明显的不同。

在 RISC 计算机中主要参与计算的是寄存器，saved registers 就是指在进入一个函数后，如果某个保存原函数信息的寄存器会在当前函数中被使用，就应该将此寄存器保存到堆栈上，当函数返回时恢复此寄存器值。而且由于 RISC 计算机大部分采用定长指令或者定变长指令，一般指令长度不会超过 32

个位。而现代计算机的内存地址范围已经扩展到 32 位，这样在一条指令里就不足以包含有效的内存地址，所以 RISC 计算机一般借助于一个返回地址寄存器 RA(return address) 来实现函数的返回。几乎在每个函数调用中都会使用到这个寄存器，所以在很多情况下 ra 寄存器会被保存在堆栈上以避免被后面的函数调用修改，当函数需要返回时，从堆栈上取回 RA 然后跳转。移动 SP 和保存寄存器的动作一般处在函数的开头，叫做 function prologue；恢复这些寄存器状态的动作一般放在函数的最后，叫做 function epilogue。

2.4 远程调试

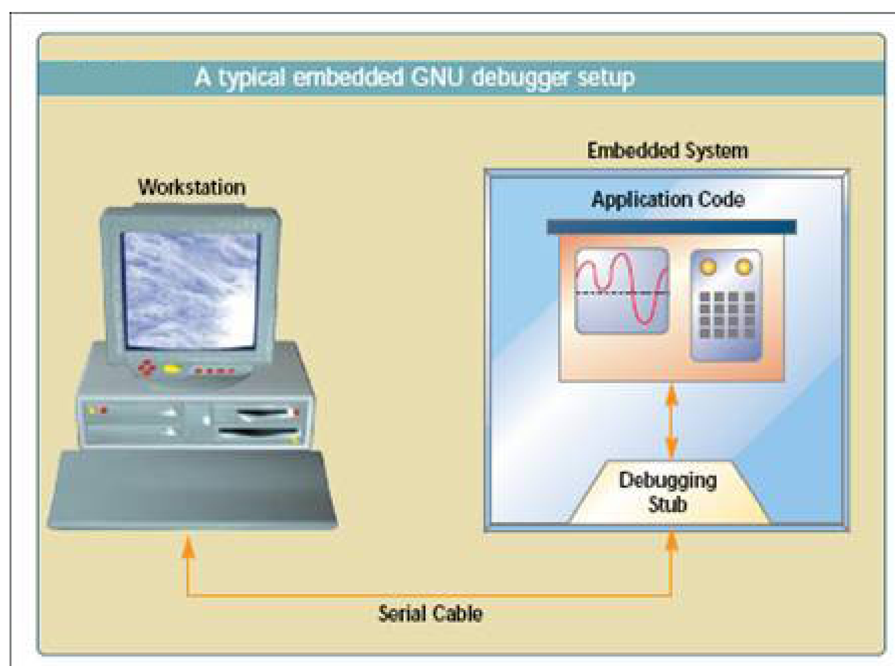

图 2.2: Remote Debugging

在 GDB 的许多特性中，远程调试 (remote debugging) 是其中有着非常重要应用的一项。远程调试是指 GDB 运行在工作站上，使用串行接口、网络接口或者其它接口与运行被调试程序的平台(目标机)进行通信，实现调试功能。[4]

当进行远程调试时，GDB 依赖于一个调试 stub 提供的功能，如图 2.2 所示。Stub 是指在目标机上运行的一小段代码，这一小段代码主要提供了对内存、寄存器的访问和修改以及与 GDB 进行通信的功能。GDB 通过 RSP(remote serial

protocol）与它进行交流，实现调试功能。[5]

值得注意的是，stub 的实现方式有多种：stub 可以放在 ROM 上，这样 stub 可以作为一个守护程序来处理处理机状态；stub 也可以与被调试程序编译到一起下载到内存中，但会随着应用程序的退出而退出；如果在编译时使用不同的连接脚本（link script），将 stub 放在不与被调试程序重叠的内存区域中，也可以实现让 stub 成为一个一直运行的守护程序。

第三章　GDB 的内部算法

GDB 有很多状态转移和内部执行逻辑的算法，下面简单介绍一下几个最基础的算法：

3.1　GDB 的启动

GDB 是一个用户交互软件，下面简单地看一下 GDB 在启动时做了些什么：

(1)处理命令行传入的参数，并由此设置一些对运行产生影响的变量。比如：GDB 标准输入输出和标准错误输出的文件描述符，用户界面，执行文件的参数，脚本的文件路径等等，可以使用 "gdb --help" 来查看所有可能的参数。

(2)调用各种初始化函数来对调试器进行初始化。初始化的内容包括：命令列表，目标处理机宏定义，工具函数，所有文件，当前计算机体系架构，命令行解释器，串行输入，信号处理函数，源程序语言，用户界面等。由于 GDB 本身的可扩展和可移植性，它定义很多通用的工具函数接口，需要开发人员把具体的实现注册到 GDB 内部的指针变量或者定义为相应的宏，这样 GDB 就可以采用同样的接口来使用这些函数和变量的不同实现。

(3)根据第一步设置的变量并使用第二步初始化后注册的函数来执行一些相应的动作，然后转移控制权。比如打印出 GDB 的版本信息，打印出当前时间，执行相应的脚本文件等等，执行完这些动作以后 GDB 会直接退出或者进入命令循环，等待用户的输入。

3.2　Frame 模型

Frame(stack frame) 模型是 GDB 用来确定函数的调用和被调用历史的采用的模型。它的理论基础已经在上文中讲过，其最初的设计是为了支持 DWARF 调试信息格式里的 CFI(call frame information) 。GDB 的 frame 模型是这样的：每个 frame 与一个函数对应，当前 PC(program counter) 所处函数的 frame 为第0层，此 frame 的上一层就是此函数调用者的 frame，就这样一级一级往上回滚 (unwind)，就可以构造出整个函数调用的历史。在 GDB 内部是通过构建链表数据结构来实现的。在 GDB 调试时查看函数的 backtrace 就是此模型

的一个外部表示。当然此模型还有很多用处，比如从被调用者中返回，从函数中跳出，步过一个函数，都需要此模型来判断程序的中断位置。在这里还有一个重要的特殊的 frame，叫做 sentinel frame，即第-1层 frame，定义此 frame 的目的是为了回滚得到当前(第0层) frame，这是整个 unwind frame 过程的开始，是回滚得以正常进行的关键。GDB 会使用 create_sentinel_frame 函数，根据 prologue 分析得到的寄存器信息来创建这个 frame。

3.3 Prologue 分析

Function prologue 指的是在目标文件中，位于每个函数开头的为该函数准备好栈空间和保存寄存器的机器代码，其行为主要包括将一些重要寄存器保存进栈，移动栈寄存器指针为当前函数留出堆栈空间。当函数执行完毕后，对应的 epilogue会做相反的事情以把处理机环境恢复到函数调用前的状态。prologue 分析主要是为构建 frame trace 服务的， GDB 必须知道各个 frame 的基址和偏移量才能决定某个地址位于哪个 stack frame 中，而且 GDB 必须知道在进入函数时都对哪些寄存器做了什么修改，才可以通过分析得到正确的寄存器内容，而且回滚到正确的 frame 。如果编译器能产生正确的 DWARF 调试信息，那么其中的 CFI 就能够给出足够的信息，只需读取相应的内容即可，否则就需要使用prologue 分析来得到这些信息。

Prologue 分析主要采用的方法是从函数入口开始检测一定范围的机器指令，分析其中每条指令对寄存器的修改。例如：寄存器内容压栈，转移到另外的寄存器等等，然后将这些修改进行汇总，以得出此函数的 prologue 对寄存器和堆栈的修改，并将这些修改保存在一个 register cache 结构中，以供构建 frame 列表的函数使用。有的还顺便返回分析得到的函数的第一条表达式语句的地址来达到 skip prologue 的目的。

3.4 断点处理

断点 breakpoint 是指用户指定程序中的某个位置，当程序运行到这个位置时会中断执行，把目标处理机控制权重新转交给调试者。实际使用中主要有两种方法来实现断点功能，一种叫做硬断点，另一种叫做软断点。硬断点是指目标处理机硬件支持断点功能(或者在目标处理机模拟器上支持断点功能)，可能的实现情况是目标处理机有一个专门存放断点的寄存器，当 PC 和这个寄存器

内容一样时，目标处理机会产生一个异常，异常处理函数会把当前处理机状态通报给 GDB ；软断点是指目标处理机不支持硬件断点，需要 GDB 采用其它方法来实现断点功能。一般在这种情况下，GDB 会把内存中需要中断位置的指令修改为一条中断指令或者其它可以使目标处理机产生某种异常的指令，当程序运行到这一位置，执行这条指令后，目标处理机会捕获异常，进入相应的异常处理函数，异常处理函数会把状态通报给 GDB 。需要注意的一点是，这种情况下必须要程序在目标处理机上的存储空间可写，否则无法实现软中断。

不过，上面仅仅是从理论上分析了 GDB 的断点机制，在当前版本的 GDB 中，尤其是在跨平台调试的过程中，GDB 可以把断点功能下放给芯片上的 GDB stub 或者 gdbserver 进行处理。GDB 仅仅告诉目标机上的程序在哪个地方设置断点，而由目标机上的程序实现断点功能。虽然有专门的协议格式规定哪些是硬件断点，哪些是软件断点，但具体的实现方式完全可以由 GDB stub 和 gdbserver 自己决定。

3.5 单步执行

这里的单步执行 (single stepping) 有两个方面的理解：单步跨过一行源程序和单步跨过一条机器指令，然后中断程序执行，将处理机控制权转交给调试者。单步跨过一行源程序是通过分析调试信息 SAL 来实现的， SAL 里标明了源程序的行和机器指令的对应关系，GDB 会在下一行源程序的开始设置断点来实现单步跨过源程序。单步跨过一条机器指令也有两种实现方式，类似于断点，分为硬单步和软单步。硬单步一般是目标处理机支持单步执行的功能；软单步是由 GDB 通过分析当前指令设置相应断点来实现单步的功能。GDB 首先读取 PC 所指向的指令，如果当前指令是一条一般指令，GDB 就在当前指令的下一条指令处插入断点；如果当前指令是一条分支或跳转类指令，GDB 通过指令集定义对指令解码，分析出它将要跳转的位置，然后在那个位置插入断点。

3.6 信号处理

信号是指在程序运行过程中发生的一些不可预料的事件。操作系统会将程序运行过程中可能产生的各种信号编号，并赋予有些信号相应的名字。GDB 可以检测到被调试程序运行过程中产生的信号，并可以由用户自定义特定信号的

处理方法；目标机也可以接受 GDB 传给它的信号。用户可以使用 info handle 命令查看各个信号的处理方法。

但是在一般情况下，目标机只会实现少量几种信号，并且这些信号和 GDB 的信号向量列表并不保证能一一对应。这样就需要目标机上的处理程序或者 GDB 自身进行转换，这个过程并不复杂。

3.7　线程处理

在拥有线程机制的操作系统上运行的程序，使用 GDB 调试时，可以在线程之间进行切换并访问对应资源。GDB 会根据目标机返回的信息来构建线程列表，并以线程编号为索引保存各个线程对应的信息。因为线程之间共享内存和系统资源，GDB 只需要保存进程对应的堆栈和PC等一些简单的信息，这些信息体现为 gdbthread.h 中的 thread_info 结构。所以这个处理也比较简单，GDB 源程序文件 thread.c 中定义了线程处理的相关函数。

由于 GDB 中的线程机制是指软件线程，在当前版本的 GDB 中并没有实现硬件线程，所以在向某些多线程处理器移植时，会采用欺骗 GDB，用软件线程模拟硬件线程的方法来实现硬件多线程调试。

3.8　内部函数调用

内部函数调用（inferior function call）是指 GDB 手动调用某个函数，而效果像是在程序的当前位置执行一条函数调用语句。为了实现完全仿真，内部函数调用必须使用到程序的堆栈和数据区而当返回后又不能影响程序的正常运行，所以 GDB 需要做到的是在调用前保存处理机和内存现场，然后给函数传入正确的参数和返回地址进行调用，在调用后恢复处理机和内存现场。它的实现方式是调用和目标机相关代码设置函数调用的参数和返回地址，在返回地址上设置内部断点，修改 PC 到函数的入口地址开始执行调用，当程序执行到此断点时得到函数返回值并将对寄存器的所有修改进行回滚。函数内部对数据区的修改是不可恢复的，因为 GDB 并不可知其操作，GDB 只恢复寄存器到原始状态，因为函数参数和返回值保存的位置都是寄存器和堆栈，而将堆栈指针恢复到原始位置就能取消所有对堆栈的修改。

不过内部函数调用还需要做一些其它的事情，例如处理函数参数。C 语言中会对参数类型进行自动转换，K&R 风格就需要将所有 float 类型的参数转

换成 double 类型,而 prototype(ANSI C) 风格就不做此转换。此外还需要找到函数的入口地址，确定返回地址。这些工作所依赖的函数和数据结构可以在 infcall.h/c 中找到。

3.9 长跳转支持

长跳转 (longjmp) 是 C 语言的一种机制，它绕过了一般函数的调用和返回规则。在 C 头文件 setjmp.h 中有一个依赖于目标机的结构体 jmp_buf。这个机制的体现是先定义一个 jmp_buf 类型变量 buf，调用 setjmp(buf) 函数时会把现场保存在变量 buf 里，当程序的继续执行遇到 longjmp(buf, value) 时候，程序会根据 buf 的内容跳转回最近的一个 setjmp 函数调用的位置，其效果相当于执行 setjmp() 这个函数，其返回值是 value。[6]

GDB 内部对长跳转的支持是通过设置一些特殊的内部断点来实现的。因为 jmp_buf 的结构和具体的计算机体系架构有关，GDB 先读取一些 symbol，即：longjmp，_longjmp，siglongjmp，_siglongjmp 的内容，然后用当前 CPU 体系架构依赖代码注册的 GET_LONGJMP_TARGET 宏或者 gdbarch_get_longjmp_target 函数来得到要跳转到的位置，在这些位置设置内部断点以达到对长跳转的支持。由于这些断点不是调试者设置的，因此在正常的调试过程中是不可见的，但可以用提供给 GDB 维护者的命令 "maintenance info breakpoints" 查看。

3.10 观察点

观察点 (watchpoints) 指的是一种设置在数据区的断点，当设置观察点的数据被访问时，程序会产生中断。和断点一样，观察点也分为硬观察点和软观察点。硬观察点和断点基本没有什么不同，当数据区的数据被访问时，CPU 自动产生中断并向 GDB 报告自己的状态；但软观察点和软断点就有很大不同了。因为 GDB 不应该去修改数据区的内容，那样会影响程序的执行，所以 GDB 只能采取一种很笨的做法，就是单指令运行，在每次停顿查看和比较一下设置观察点位置的数据是否被修改。但需要注意的一点是，对于读取和访问类型的观察点，GDB 需要硬件的支持，因为 GDB 在外部只能比较数据是否被修改，而不能得到数据是否被访问和读取的信息。

3.11 检查点

从概念上来讲，检查点 (checkpoints) 是 GDB 保存程序调试期状态的一种机制，调试者可以回到设置检查点的位置重新从那里开始调试程序。在支持进程机制的 Linux 系统下，一般设置检查点就是挂起当前进程，并根据当前进程状态创建一个新进程继续运行。但是在调试嵌入式系统的程序时，由于资源的限制性，不太容易做到这一点，比较简单的情况是嵌入式芯片的模拟器支持保存状态的功能，这样在进行模拟器调试的时候 GDB 就可以利用这些来方便的进行检查点的设置。如果目标机上不能运行支持进程机制的操作系统，GDB 还没有有效的方法来实现检查点机制。这个功能在 GDB 的当前发行版中还不够成熟。

第四章　移植 GDB 涉及到的文件

GDB 的内部结构设计和 GCC 类似，每个部分都分为前端和后端。其中和具体目标处理机有关的代码主要包括两个部分：特定格式二进制文件的读取和分析以及目标处理机体系结构依赖文件。

GDB 对二进制文件的读取是依赖于 LIB BFD(Binary File Descriptor Library) 的，对机器指令的反汇编依赖于 Opcode library 。这两个库都是跟随 GNU 的二进制工具集 GNU Binutils 发行的。 GDB 源代码中的 bfd 和 opcode 这两个目录下的内容和 GNU Binutils 对应版本的源代码里的这两个目录内容是完全相同的。如何移植 LIB BFD 和 Opcode 可以参考文献《GNU as 的移植》[8]和《Binary Utilities Generator for Application Specific Instruction Processor》[9]。本文主要讨论特定计算机体系架构的 target dependent 部分。

GDB 的 target dependent 主要包含两个部分，定义 GDB 的运行行为的头文件和初始化 gdbarch 结构的文件，这两个部分由不同的两个脚本文件来处理。下面我们用 SRC 表示 GDB 源程序的根目录：

在编译时， "SRC/gdb" 目录下的 configure 脚本会根据编译时候指定的目标处理机类型在 "SRC/gdb/config" 目录下寻找以对应目标处理机命名的目录，然后在相应目录下寻找 cpuname.mt 文件。根据 cpuname.mt 文件的内容找到对应于此目标处理机的定义 GDB 的头文件，然后 configure 会把此头文件转化为一个名为 tm.h 的头文件，configure 还会据此生成 config.h 文件。因为最常被包含的 defs.h 头文件中包含了这两个头文件，在下面的编译过程中，几乎所有的文件都会包含这两个头文件，因此在 "SRC/config/cpuname"目录下的头文件就能定义很多特定的宏来影响 GDB 的运行行为。

而 configure 文件还会根据 cpuname.mt 和 "SRC/gdb/Makefile.in"，生成指定目标处理机的 Makefile 文件，这个文件中指定了目标处理机对应的 target dependent 文件和编译命令，并根据所知的信息生成 "init.c" 文件。在 init.c 中的函数 initialize_all_files 为 GDB 的很多关键部分进行初始化操作，注册相关函数。

这里列出所有需要添加和修改的文件，假设我们正在为一块名为 BOX 的芯

片移植 GDB 。

需要添加的文件：

SRC/gdb/config/box/box.mt, SRC/gdb/config/box/tm-box.h;

SRC/gdb/box-tdep.h, SRC/gdb/box-tdep.c.

需要修改的文件：

SRC/gdb/Makefile.in, SRC/gdb/configure.host, SRC/gdb/configure.tgt;

SRC/configure.in, SRC/configure.sub, SRC/readline/support/config.sub.

第五章　GDB 后端的编写

移植 GDB 其实就是为 GDB 编写一个新的后端，这里的后端指和计算机体系结构密切相关的那部分代码。GDB 依赖这部分代码来实现对 CPU 行为的理解和操控，所以说这部分代码被称为 target depentdent code，包含这些代码的文件也被称为 target dependent file(tdep)。

GDB 内部对目标架构的抽象非常简单，它认为目标机就是一个拥有一些寄存器和一块存储空间的机器。对目标机如此简洁的抽象能让 GDB 得到很简单的运行时逻辑。在 GDB 的源代码中描述某种目标架构的抽象结构是一个 C 语言的结构体类型 struct gdbarch。这个结构体类型，以及 gdbarch.c/h 中的所有内容都是由脚本文件 gdbarch.sh 自动生成的。这个结构体类型中的成员变量和成员函数指针完全定义了 GDB 所需要知道的一个目标架构底层的一切。此结构体类型一个全局的指针变量 current_gdbarch 指向的结构体就是当前的目标架构相关的结构体实例。当需要使用这个结构体的成员变量和成员函数指针时，通过 gdbarch.c/h 中提供的函数接口进行调用。需要注意的一点是，在旧版本的 GDB 中，上面提到的很多方面都是通过定义宏到不同的函数上实现的，但是那样只能在编译时确定函数接口，不支持在运行时更改目标架构。

跟据 GDB 对计算机体系架构的抽象，我们通过几个部分来探讨 GDB 后端的编写方法：

5.1　注册新后端到 GDB 中

要想为 GDB 编写新的后端，首先要弄清楚如何将新后端注册到 GDB 中。由于在配置文件里已经注明哪些文件是当前目标架构的 tdep 文件，Makefile 会从对应的 tdep 文件中寻找以 "_initialize_" 开头的函数作为当前 target 的初始化函数。前面说过 GDB 启动时在 initialize_all_files() 中会调用这个初始化函数，所以我们需要做的就是在这个初始化函数中将新后端注册到 GDB 中。方法是在这个函数中调用 "gdbarch_register" 函数将当前目标架构的 bfd_architecture、gdbarch 的初始化和 dump 函数注册到 struct gdbarch_registration 类型的列表 gdbarch_registry的一个列表项上。这样

在需要时 GDB 就会调用注册的 gdbarch 的初始化函数对 current_gdbarch 分配内存和进行初始化。

由于 gdbarch 的初始化函数负责对 current_gdbarch 进行初始化，所以我们下面讨论的问题主要就和这个函数有关，即如何在这个初始化函数中为 current_gdbarch 设置正确的工作正常的成员变量和成员函数指针。

5.2 设置目标架构基本信息

描述某种目标架构来说，总应该提供一些非常基本的信息。这些信息其中有些已经在 LIB BFD 的某些变量中定义过了，比如计算机体系结构名、计算机字长、计算机地址长度、每比特位数等等， gdbarch 中有一个指针变量 bfd_arch_info 就直接指向 LIB BFD 中对应的 struct bfd_arch_info 类型的结构体实例；不过更多计算机体系架构的信息仍需要在 gdbarch 的初始化函数中设置。

在这些设置中比较重要的有计算机字节顺序 (byte order)、应用二进制接口 (ABI, Application Binary Interface)，还有一些与计算机体系结构特性和 C 语言标准密接相关的设置，如内建各种类型的比特数，内建浮点数类型的格式等等。这些以及随后所讲的成员变量和成员函数指针都可以通过 set_gdbarch_xxxx 来设置。其中 xxxx 指代 struct gdbarch 中的某个成员变量或成员函数指针，比如 set_gdbarch_short_bit(gdbarch, 16) 就是将 gdbarch 指向的成员变量 short_bit 置为16，意思就是指此 gdbarch 的 short 类型长度为 16 位。

5.3 设置寄存器相关信息

上面提到了，GDB 把一个目标机看成一个拥有指定数量寄存器和一块存储空间的机器，不过 GDB 并没有简单地认为所有机器都具有相同大小的而且顺序排列的寄存器。确实，在 GDB 看来，目标机的寄存器是以 current_gdbarch 的成员变量 num_regs 为上限，从小到大编好号的一个数组， GDB 在读取这些寄存器的时候也是以完全序列化的方法来读的，因为这样能够带来很大的方便。但是，由于这些寄存器在机器指令和调试信息中的编码并不一定与 GDB 完全一样，所以 GDB 中提供了一种机制使用户能将不同大小和使用方式的寄存器映射到一个线性表上去。在 gdbarch 结构体中有几个成员函数指针实现这个功能，

分别是 stab_reg_to_regnum 、stab_reg_to_regnum 、 dwarf_reg_to_regnum 、sdb_reg_to_regnum 、 dwarf2_reg_to_regnum ，这几个函数作用都是将寄存器在某种目标文件格式中的编号映射到 GDB 中的编号。如果用户需要这些寄存器编号的映射，就应该把对应的映射函数设置到 gdbarch 的成员函数指针上。由于在调试过程中，调试者需要用某种易记的名字来代替寄存器的编号，需要设置 gdbarch 的一个函数指针 register_name 指向一个函数来进行寄存器名与寄存器编号的转换。GDB 还需要知道寄存器的大小才能正确读写和为之分配空间，需要设置 gdbarch 的成员函数指针 register_type 指向某个可以返回寄存器的类型(用 GDB 的内部类型表示的大小)的函数，。

对于 GDB 来说，要想实现调试功能，需要知道几个非常重要的寄存器编号：pc(program counter)，sp(stack pointer)，ps(processer status) 和 fp0(floating pointer register 0)。pc 中存储下条指令的地址，sp 寄存器存储当前的栈指针，这两个寄存器是目前几乎所有 RISC 计算机都具备的。ps 寄存器存放处理机状态字，比如 i386 的 FLAGS 寄存器。fp0 为第一个浮点数寄存器，这两个寄存器是否存在视计算机体系结构而定。gdbarch 有对应的成员变量 pc_regnum，sp_regnum，ps_regnum，fp0_regnum 保存它们的寄存器编号。如果当前 target 包含上述寄存器，就应将 gdbarch 中的对应成员变量设为对应的寄存器编号，它们的默认值为 -1 ，即不存在。如果 GDB 不能直接读写 pc 或者不能读取 sp 寄存器，需要设置 read_pc，write_pc 和 read_sp 指向相关函数来实现这些功能，因为对这两个寄存器的读写是 GDB 控制 CPU 运行的关键。

gdbarch 的成员变量 num_regs 和 num_pseudo_regs 指定了目标架构的寄存器数和伪寄存器数，应该按照芯片的实际情况进行设置。寄存器即是指真实的寄存器，伪寄存器是指某些目标架构为了编译时方便而虚拟出的寄存器。对真实的寄存器， GDB 直接按照寄存器编号序列读取和写入，对于伪寄存器，应当设置 gdbarch 的两个成员函数指针：pseudo_register_read 和 pseudo_register_write指向负责处理伪寄存器的读取和写入的函数。GDB 还将寄存器分成许多组，如果需要的话，应当设置 gdbarch 的成员函数指针 register_reggroup_p 指向判断某个寄存器是否属于某个组的函数。

5.4 设置 stack frame 相关信息

首先，我们得确定目标架构的堆栈是向上还是向下增长的，可以设置

gdbarch 的成员函数指针 inner_than 指向两个内建的函数 core_addr_lessthan 和 core_addr_greaterthan 其中之一，具体选择哪个由 target 的实际情况决定。在 tdep 文件中，最重要的关于 frame 的信息是如何 unwind frame ，就是设置 gdbarch 中的成员函数指针 unwind_pc 指向行为正确的函数。

unwind pc 这个说法相当拗口，其实这个函数的主要作用就是为 unwind frame 提供足够的地址信息，简单地来说呢就是返回被 unwind 的 frame 所处函数的被调用地址，根据程序执行的顺序，一般情况下这个地址都是此函数的返回地址再减去调用指令长度。但是确定函数的返回地址在不同的计算机体系结构上并不相同，在 RISC 计算机中，一般情况下返回地址会放在一个专门的寄存器 ra 中。但是在 unwind 的时候 GDB 需要知道每个函数中 ra 存放的内容，仅仅取得当前 ra 的值是没有多大意义的。在前面我们也讲过，可执行程序也需要保存这个内容，当 ra 有可能被被调用函数修改时，被调用函数会把 ra 保存在栈上(如图 2.1)，这样通过分析 ra 在栈上的保存位置就能得到正确的返回地址。所以这个函数的通常做法是用 frame_unwind_register_unsigned 函数来取得 ra 的值。至于它如何获得正确的寄存器值，我们下面会继续讨论，不过在这里需要注意的一点是如果 unwind 的是 SENTINEL_FRAME ，那么就应该返回当前 pc 的值。

下面讨论 frame_unwind_register_unsigned 如何取得对应 frame 的寄存器值。在 frame.c 中定义了一个结构体类型 frame_info ，这个结构体可以保存每个函数的 frame 信息。 GDB 对 call frame 的分析就是自底向上构建一个 frame_info 的列表，最底层就是我们上面所说的 SENTINEL_FRAME 。frame_unwind_register_unsigned 对寄存器的读取就是通过读取这个列表中的信息来得到的。如果这个列表没有被创建， GDB 会调用一系列函数创建这个列表，而后端需要做的是实现 frame_sniffer 和 frame_base_sniffer 函数并使用 frame_unwind_append_sniffer 和 frame_base_append_sniffer 函数将其添加到对应的 sniffer 列表中去。GDB 会按顺序使用列表中的函数直到找到一个能正常工作的。简单的来说，这两个函数的作用就是告诉 GDB 参数 next_frame 的上一层 frame 对应的函数的入口地址和 prologue 中对寄存器的存储。如果编译器支持 dwarf2 调试信息格式，那么在 dwarf debugging information 里的 CFI(call frame information) 已经提供了足够的信息来，这时候只需要把 dwarf2.c/h 中的 dwarf2_frame_sniffer 和 dwarf2_frame_base_sniffer 添加到 sniffer 列表中即可。如果不支持调试信息，那么需要后端实现这两个函

数，具体的做法可见 mips-tdep.c 中的实现。

由于函数的 prologue 主要作用是保存处理机状态，是由编译器自动生成的，不对应于任何代码。所以对于调试者来说它们是没什么用处的，可以设置 gdbarch 的成员函数指针 skip_prologue 指向跨过 prologue 的函数。同分析 prologue 一样，如果有足够的调试信息，可以调用 skip_prologue_using_sal 得到跨过 prologue 后的第一条指令地址，即使用 SAL(symtab and line) 来确定 prologue 的范围，其实就是寻找函数源代码中的第一行语句对应的机器指令。如果没有足够的调试信息，也需要对机器指令进行分析来找到 prologue 的末尾。

5.5 单步和断点

单步和断点都是 GDB 控制 CPU 运行的手段。如果硬件直接支持这两个功能，在 tdep 文件里面不需要做什么事情。如果硬件不支持这两个功能，就需要设置相应的函数来处理这两个模块。breakpoint_from_pc 指向的函数作用是根据 pc 位置的不同来选择使用哪种 break 指令，主要为了方便使用变长指令的计算机。 memory_insert_breakpoint 和 memory_remove_breakpoint 负责插入和删除断点。但是由于大部分计算机都支持硬件断点，或者断点由目标机端的程序进行处理，所以一般的目标架构的 tdep 中都没有设置这两个函数。

由于单语句执行是从 SAL 中获得下条语句的指令地址然后插入断点，所以单步跨过语句完全依赖于断点功能和程序文件中的行信息。但软件单步跨过指令就需要对机器指令进行分析了，因为机器指令中有跳转、转移和返回指令，所以单步并不是简单地把断点设置在一条指令长度之后。一般地做法是读取当前机器指令，分析它是否属于转移指令，如果属于转移指令，就对指令进行解析，读取相应地寄存器，得到转移的地址，在转移的地址上设置断点。这个功能由 gdbarch 中的成员函数指针 software_single_step 指向的函数实现。如果这个函数没有被注册，则说明目标机支持硬件单指令运行。

5.6 虚拟函数调用

虚拟函数调用 (dummy function call) 就是指在 GDB 中使用命令："print foo(args...)" 来进行函数调用。这种调用就是前面算法分析中讲过的内部函数调用。关于设置函数参数和返回地址 GDB 是调用 current_gdbarch 的

成员函数指针 push_dummy_call 指向的函数实现的。这个函数的一个参数 bp_addr 是虚拟函数调用的返回地址，这个地址也是从 gdbarch 的一个成员变量 call_dummy_location 的设置计算出来的，这个整型变量有三个内建值：ON_STACK(1)，AT_ENTRY_POINT(4)，AT_SYMBOL(5)，分别指设置返回地址在堆栈上，在程序入口地址，在 symbol "__CALL_DUMMY_ADDRESS" 上。这三个宏分别定义和实现在 inferior.h 和 infcall.c 中。push_dummy_call 需要把返回地址赋给寄存器，把参数按照目标架构的调用约定放在寄存器和堆栈上，修改 sp 到正确的位置。

由于每个目标架构的返回值存放方式不同，需要定义 gdbarch 的成员函数指针 return_value 指向获得或者写入返回值的函数，应该按照目标架构调用约定中的定义来写。如果没有定义 return_value，就需要分别定义 store_return_value，extract_return_value 分别负责写入和获得返回值，否则就没有必要重新定义这两个函数。

5.7 屏幕输出相关

除上面的一些函数以外，还需要自行定义打印寄存器和反汇编指令的函数。打印寄存器函数在 gdbarch 中的指针是 print_registers_info，这个可以按照实际需要的打印格式自行定义；反汇编机器指令函数在 gdbarch 中的指针是 print_insn，这个一般是直接设置为 opcode library 中与目标架构相关函数，即是目标架构的二进制工具 objdump 反汇编代码时所需使用的函数。

5.8 小结

GDB 作为开源编译器，拥有非常优秀的程序结构设计并为移植提供了非常方便的接口，而且拥有着诸多优点。本文对如何移植 GDB 到一个新的 target 上做了非常清晰的描述，并且对简约纳公司生产的芯片进行了移植，结果显示本文的方法能够快速成功地将 GDB 移植到目标架构上。

参考文献

[1] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB–The GNU Source-Level Debugger, Ninth Edition*. Free Software Foundation, ISBN: 1-882114-77-9, Boston, USA, 2006.

[2] John Gilmore, Cygnus Solutions, Stan Shebs. *GDB Internals–A guide to the internals of the GNU debugger*. Free Software Foundation.

[3] Dominic Sweetman. *See MIPS Run*. Morgan Kaufmann Publishers Inc., ISBN: 1-55860-410-3, San Fransisco, USA, 1999.

[4] Bill Gatliff. *Embedding with GNU: GNU Debugger*. Embedded Systems Programming, September, 1999.

[5] Bill Gatliff. *Embedding with GNU: the gdb Remote Serial Protocol*. Embedded Systems Programming, November, 1999.

[6] ISO, IEC, ANSI, ITI. *Programming Languages – C*. American National Standards Institute, ISO/IEC 9899:1999(E), New York, USA, 1999.

[7] Steve Chamberlain, Cygnus Support *LIB BFD, the Binary File Descriptor Library*. Free Software Foundation.

[8] 陈文平. *GNU as 的移植*. 计算机工程, 第29卷, 第10期, 2003.

[9] 陈建智. *Binary Utilities Generator for Application Specific Instruction Processor*. 台湾大学资讯工程学研究所硕士学位论文, 台湾大学图书馆, 系统识别号: U0001-1307200517170400, 2005.

致　　谢

值此论文完成之际，向多年来给予我关心和帮助的老师、同学、朋友和家人表示衷心的感谢!

谨以本文献给我伟大的母亲!