分 类 号_____ 学号 M201172605

学校代码 10487 密级_____

華中科技大學

硕士学位论文

龙芯 3A 环境下的 EJTAG 调试技术

学位申请人： 钟 逸

学 科 专 业： 计算机技术

指 导 教 师： 李国徽 教授

答 辩 日 期： 2013 年 5 月 29 日

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree for the Master of Engineering

EJTAG Debugging In Loongson 3A System

Candidate : Zhong Yi

Major : Computer technology

Supervisor : Prof. Li Guohui

Huazhong University of Science & Technology

Wuhan 430074, P.R.China

May 2013

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于　保密□，　在_____年解密后适用本授权书。
　　不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日 期 ：　　年　　月

摘 要

为了建立能够让军队、政府、国有企业和科研机构用的放心的信息系统，保障国家信息网络的安全，2001 年，中科院计算所成立了一支专门开发设计我国自主 CPU 的课题小组，取名“龙芯”。配合开源的 Linux 系统，龙芯计算机将大大减小国家信息网络的风险。在新系统的开发过程中，调试是一个非常重要的环节，一个优秀的调试器可以大大加快研发的进程。然而，现有的大多数调试器都是针对应用程序，对系统内核的调试无能为力，而少数的内核调试系统却也不是为国产的龙芯平台而打造。

本项目研究了如何使用 EJTAG 接口，来对龙芯 3A 处理器平台下的 Linux 内核进行源码级调试。

本文先介绍了国内外内核调试的研究现状，比较了 EJTAG 调试和其他内核调试方案的优缺点。随后，文章对 EJTAG 调试所使用的核心知识技术一一进行说明，并介绍了 EJTAG 调试系统的结构。接下来，文章介绍的指令级调试讲述了如何从并口操作起一步步实现发送 CPU 指令，进而达到读写内存、寄存器的目的。文章随后介绍的源码级调试则解析内核符号表，使调试系统能够直接对 Linux 源码进行调试，从而达到打印变量，设置断点等目的。

关键词：交叉调试，内核调试，MIPS，EJTAG，龙芯

ABSTRACT

In 2001, Institute Of Computing Technology Chinese Academy Of Sciences established a research team named "Loongson" to develop China's own CPU. The team is established to assure the security of the information systems used by the military, government, state-owned enterprises and research institutions. With the help of the open-source Linux system , Loongson computer will greatly reduce the risk of the national information network. Debugging plays a very important role in the development of the new system, and a good debugger can greatly speed up the process of the research. Unfortunately, most available debuggers are designed for applications other than kernels. What's more, the few kernel debuggers can not work well on the loogson 3A processor platform.

This project examines how to use EJTAG to do source-level debugging of the Linux kernel under the Loongson 3A processor platform.

This article first introduces domestic and foreign kernel debugging Research , compares the advantages and disadvantages of EJTAG debug and other kernel debug methods. Some basic concepts are introduced later to make preparation for the debug system. Subsequently, the article describes the structure of the EJTAG debug system. Next, the article describes the instruction-level debugging, explains how to implement a CPU instruction from the parallel port operation , thus achieving the read-write memory and registers purposes. Then the debugger works with libdwarf library to implement source-level debugging. You can print variable or set breakpoints with it.

Key words：Cross-debugging，Kernel-debugging，MIPS，EJTAG，Loongson

目 录

1 绪论

1.1 课题的背景及意义

自第三次工业革命以来，以美国为首的西方国家无可争议的走在了高新技术的前沿。大至能够进行星际畅游的航天飞船，小至我们身边随处可见的芯片、传感器，美国都处于绝对领先的位置。以人们日常生活中离不开的电子计算机为例，其核心部件 CPU，全球使用的几乎都来源于美国的 Intel 和 Amd 两家公司[1] 。个人用户或许可以不太在意自己的电脑部件是哪里制造，然而对于国家而言，使用他国提供的芯片来构建自己的信息系统，就好像是雇佣外国人来管理自己国家的机密信息，其危险性不言而喻。

为了建立能够让军队、政府、国有企业和科研机构用的放心的信息系统，保障国家信息网络的安全，2001 年，中科院计算所成立了一支专门开发设计我国自主 CPU 的课题小组，取名“龙芯”[2] 。采用龙芯 CPU，将有助于消除我国在电子政务、国防等方面的安全困惑，改变在信息安全领域的被动局面[3] 。

冰冻三尺，非一日之寒。龙芯作为国产的 CPU，不可避免的面临着起步较晚，经验较少的困难。无论是软件还是硬件，每一个新的设计，每一个新的尝试，都新鲜却又可能碰到难以预料的问题与故障。系统方面龙芯选用开源的 Linux，虽然 Linux 系统开源且相对稳定，然而这系统配合龙芯的 CPU 能够有着怎样的表现，却也还是一个未知数。这些只能够在不断的严格测试和调试的过程中，一步步得到完善。现有的普通调试系统，多是专门针对特定系统下对于应用软件的调试，虽然能够提供很丰富的功能、很人性化的界面，却大多需要操作系统的支持，故只工作于上层系统，而对底层的系统内核方面的问题无能为力。系统启动前，启动时，各个寄存器，各段内存中究竟储存着怎样的数据，究竟 CPU 在执行怎样的指令，都难以得知。系统就像一个黑匣子，你可以从源码看到匣子中装着的物品清单，却不知道当前这些物品的状态。

著名的计算机科学家 Brian Kernighan 曾经说过，软件调试要比编写代码困难一

倍[4]，而缺少工具的调试就愈发的困难。本课题的目标，便是建立起一个能够供龙芯机器使用的内核调试系统。若是能将本系统真正投入实用，那么我们就是为国产CPU 事业的发展做出不小的贡献。

1.2 国内外研究状况

调试器调试程序时，会将调试目标中断到调试器中。而要对系统内核进行调试，意味操作系统的内核中断，而内核负责整个系统的调度和执行，一旦它被停止，系统中的所有进程和线程也都停止运行，此时整个系统系统相当于进入了停滞模式。而这样的一个停滞的内核应该如何与调试系统进行通信？当前，全球范围内存在两种较为主流的方法。其一是使用硬件调试器，硬件调试器通过特定的硬件接口与 CPU 进行通信，能够独立于 CPU 的正常工作模式之外与 CPU 进行互动，比如 ARM 的 JTAG 接口、MIPS 的 EJTAG 接口。其二是通过“给内核打补丁”的方式，在系统内核中加入调试支持，是一种基于插桩的调试方式[5]，也被称作为基于代理的调试方式[6]。除了这两种方法，也有人使用类似于 QEMU 的虚拟软件进行系统仿真，模拟出一台特定平台的计算机，再使用虚拟软件提供的方式对其模拟平台中的系统内核进行调试[7]。然而此种方式只是模拟，其结果完全依赖于模拟软件对于特定平台的认识，不足以代表真正的内核调试。另一方面，龙芯 3A 当前也并不被 QEMU 支持，因此下面并不考虑这种“内核调试”。

kgdb 采取了上述的第二种方法[8]。kgdb 给待调试的内核打上补丁（插桩），让目标机运行打过补丁的内核。运行 gdb 的开发机则使用串口或网线连接上目标机。开发机调试内核，就像调试应用程序一样，可以查看变量、寄存器，也可以设置断点，进行单步调试等操作。

Intel 的全资子公司——风河公司提出的针对基于 JTAG 的 Linux 调试解决方案，则是采取的硬件调试方式，对插桩模式内核调试起到补充的作用[6]。基于 JTAG 的调试解决方案采用的是与目标硬件非侵入式的连接。它可以连接到已经运行、遇到故障的系统。这种连接无需改变处理器寄存器的状态和同步 Linux 内核及应用程序的上下文就可以进行调试。例如，当基于 Linux 的系统处于死锁状态时。

通过 JTAG 解决方案，开发人员可以在不改变系统状态的情况下连接到目标系统。然后开发人员就可以查看 Linux 内核对象、应用程序上下文，从而确定引起故障的线程、系统调用、和系统调用使用的参数。这是一种端到端的解决方案，尤其适用于基于代理的开发解决方案无法使用的情况。JTAG 调试简化了开发工具，从而缩短了调试时间。

国内的内核调试工具，相对而言不如国外的影响那么大，但也不乏一些优秀产品。科银京成公司开发的交叉调试器 LambdaGDB 是专门应用于嵌入式实时应用系统开发的调试工具，开发人员可以通过一个图形界面完成调试。LambdaGDB 支持两种调试代理，一种为存储器监控调试代理——LambdaTRA ，另一种是片上调试代理——LambdaODA 。LambdaTRA 运行在目标机的内存中，而 LambdaODA 是则是一个设备，通过 JTAG 接口与目标机相连，充当调试代理[9] 。

1.3 论文研究内容

作为研究生阶段工作的结尾，本论文描述了我在龙芯机内核调试项目上完成的各项工作。

论文第二章主要是对 EJTAG 的相关理论技术进行简要的介绍，为之后的系统实现作铺垫；

第三章描述了整个项目的系统结构及 EJTAG 调试部分在其中所处的作用及地位；

第四章主要讲述了如何使用 EJTAG 来对内核进行指令级调试，这也是全文的重点；

第五章则简要描述了如何将用户的源码级调试指令转换为 EJTAG 底层指令，并通过程序间的通信来实现源码级的调试。

2 基础理论

上一章介绍了课题的意义，并对国内外类似课题的研究情况进行了简要的介绍。本章将介绍部分相关基础理论，为之后的系统实现作铺垫。

2.1 龙芯 3A 介绍

龙芯处理器主要包括三个系列。龙芯 1 号处理器及其 IP 系列主要面向嵌入式应用，龙芯 2 号超标量处理器及其 IP 系列主要面向桌面应用，龙芯 3 号多核处理器系列主要面向服务器和高性能机应用[10] 。

龙芯 3A 是龙芯 3 号多核处理器系列的第一款产品，是一个配置为单节点 4 核的处理器，采用 65nm 工艺制造，最高工作主频为 1GHz，主要技术特征如下[11] ：

(1) 片内集成 4 个 64 位的四发射超标量 GS464 高性能处理器核；
(2) 片内集成 4 MB 的分体共享二级 Cache(由 4 个体模块组成，每个体模块容量为 1MB) ；
(3) 通过目录协议维护多核及 I/O DMA 访问的 Cache 一致性；
(4) 片内集成 2 个 64 位 400MHz 的 DDR2/3 控制器；
(5) 片内集成 2 个 16 位 800MHz 的 HyperTransport 控制器；
(6) 每个 16 位的 HT 端口拆分成两个 8 路的 HT 端口使用；
(7) 片内集成 32 位 100MHz PCIX/66MHz PCI；
(8) 片内集成 1 个 LPC、2 个 UART、1 个 SPI、16 路 GPIO 接口。

其集成的 GS464 是一款实现 64 位 MIPS64 指令集的通用 RISC 处理器 IP，同时其支持 MIPS 公司的 EJTAG 调试规范[12] ，这使得 EJTAG 调试龙芯 3A 成为可能。事实上，龙芯 3A 处理器同时提供了符合 MIPS 标准的 EJTAG 接口和 ARM 标准的 JTAG 接口。在本项目中，我们使用的是 MIPS 标准的 EJTAG 接口。EJTAG 部分的相关知识，可参见 2.2 节。

2.2 EJTAG 介绍

EJTAG(Enhanced Joint Test Action Group)是 MIPS 公司根据 IEEE 1149.1 协议的基本构造和功能扩展而制定的规范，是一个硬件/软件子系统，在处理器内部实现了一套基于硬件的调试特性，用于支持片上调试[13] 。

EJTAG 有意义的引脚只有五个，其定义如下[14] ：

TCK ：测试时钟输入；

TDI ：测试数据输入，数据通过 TDI 输入；

TDO ：测试数据输出，数据通过 TDO 输出；

TMS ：测试模式选择，TMS 用来设置 EJTAG 处于某种特定的测试模式，模式间的转换可参见图 2-3；

TRST ：测试复位，输入引脚，低电平有效（可选）。由于可以使用 TMS 发送软复位达到同样的功能，一般并不实现 TRST 引脚。实际上，本项目也并未使用此引脚。

EJTAG 的针脚如图 2-1 所示：

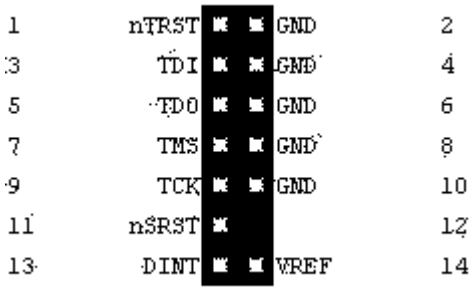


图 2-1 EJTAG 针脚

EJTAG 由 ARM 的 JTAG 发展而来[15] ，而 JTAG 的设计，原意便是对制作好的 CPU 进行测试。因此，它“够底层”，能够在不依赖于上层操作系统的前提下，直接对 CPU 进行操作，令 CPU 执行指定的机器码指令。而能够控制 CPU 执行特定指令，理论就可以让电脑执行任意操作，当然也就可以读写寄存器，读写内存，设置断点。

2.3 调试模式

在 gdb 里调试程序时，执行 r 命令，程序才会运行起来。而直到程序遇到事先设置的断点，它才会停下。内核调试却不能遵循这个流程，因为系统启动后，内核就已经在运行了。要想让内核停止下来，只有使用 EJTAG 进入调试模式。调试模式下，内核停止运行，系统可供 EJTAG 观察和调试，类似于 gdb 调试应用程序时进入断点的情况。

要想使用 EJTAG 进入调试模式，比较常见的方式有这么几种[16]：CPU 执行 SDBBP 指令、EJTAG 断点寄存器置位、单步调试等。这里让正在运转的内核进入调试模式，一般采用将 EJTAG 断点寄存器置位的办法。

其实在 gdb 中，也可以通过信号使程序暂停运行以供调试。而较为成熟的 IDE，一般也都提供了调试时暂停的功能，比如开源跨平台的 Eclipse[17]。

总之，调试模式是一种程序停止运行的状态。在调试模式这一时间停止的状态下，才能对内核进行各种状态的观察，才能控制内核的运行情况。若无特殊说明，下文中各种 EJTAG 调试均是在调试模式下完成的。

2.4 边界扫描

边界扫描是 EJTAG 调试中一个很重要的概念[18]，其特点是在待测试芯片的每个引脚上添加一个移位寄存器，如图 2-2 所示。由于移位寄存器在芯片的边界上，所以被称作是边界扫描寄存器（Boundary-Scan Register Cell）。

正常工作的时候，这些移位寄存器是透明的，并不会对芯片的输入输出造成任何影响；而在调试的时候，配合 TAP 状态机的各种状态，移位寄存器可起到直接控制引脚的输入输出[19]。简单的说，Capture-DR/Capture-IR 状态和 Update-DR/Update-IR 状态，可以将引脚上的数据捕获到移位寄存器，或是从移位寄存器更新引脚上的数据。而在 Shift-DR/Shift-IR 状态下，这些移位寄存器可以通过 TDI 和 TDO 进行数值的移位输入输出，从而设置或是读取芯片各引脚的电平[20]。2.5 节对各种 Tap 状态进行了较为详细的描述，另外 2.7 节讲述了其在多核处理中的应

用。

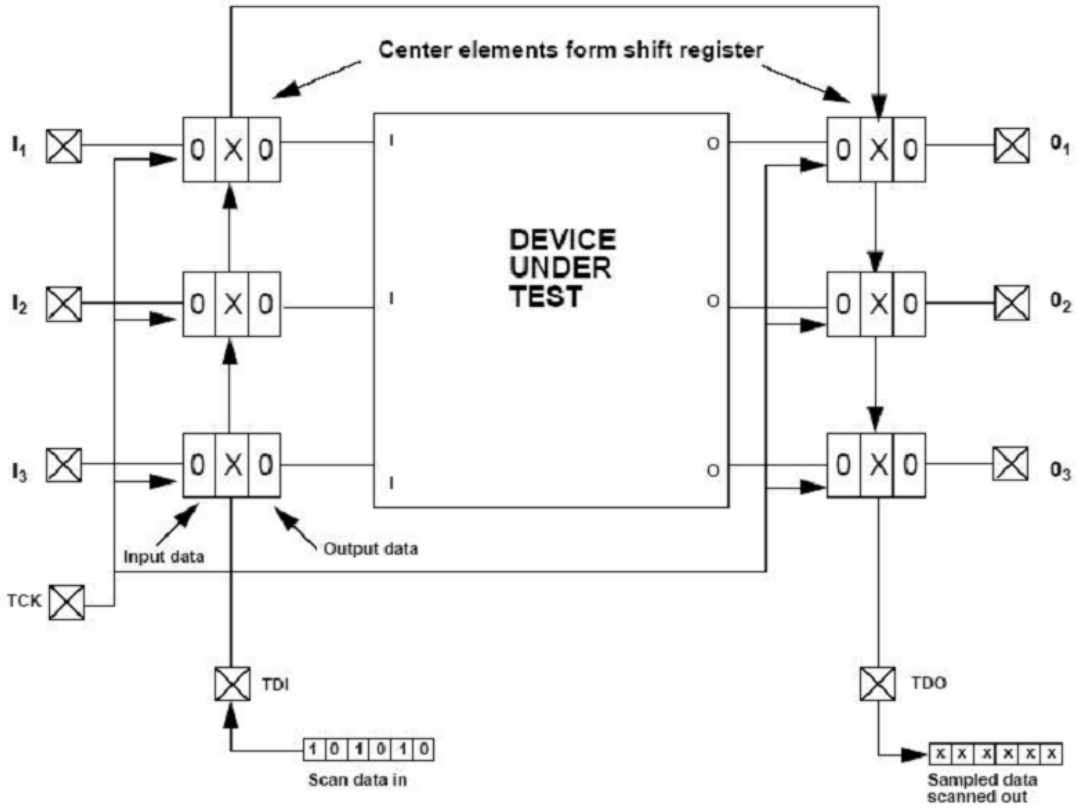


图 2-2 边界扫描链

2.5 TAP 原理

对边界扫描链的控制主要是通过 TAP（Test Access Port）控制器来完成的。EJTAG 标准中定义了 TAP 控制器，可以使 EJTAG 通过状态机实现相对比较复杂的功能。通过给 TMS 引脚输入不同的信号，即可让 TAP 控制器进入不同的状态，实现特定的功能。

TAP 控制器的状态转化图如图 2-3 所示[21] 。可以看到图中有许多以 DR 和 IR 为后缀的状态，这些状态表示当前正对数据寄存器（Data Register）或指令寄存器（Instruction Register）进行操作。由于这两类状态的原理类似，只是面向的操作对象不同，因此下面对状态转换图中的重要状态进行介绍时，若状态以 DR/IR 为后缀，仅以数据寄存器（Data Register）为例来进行描述。

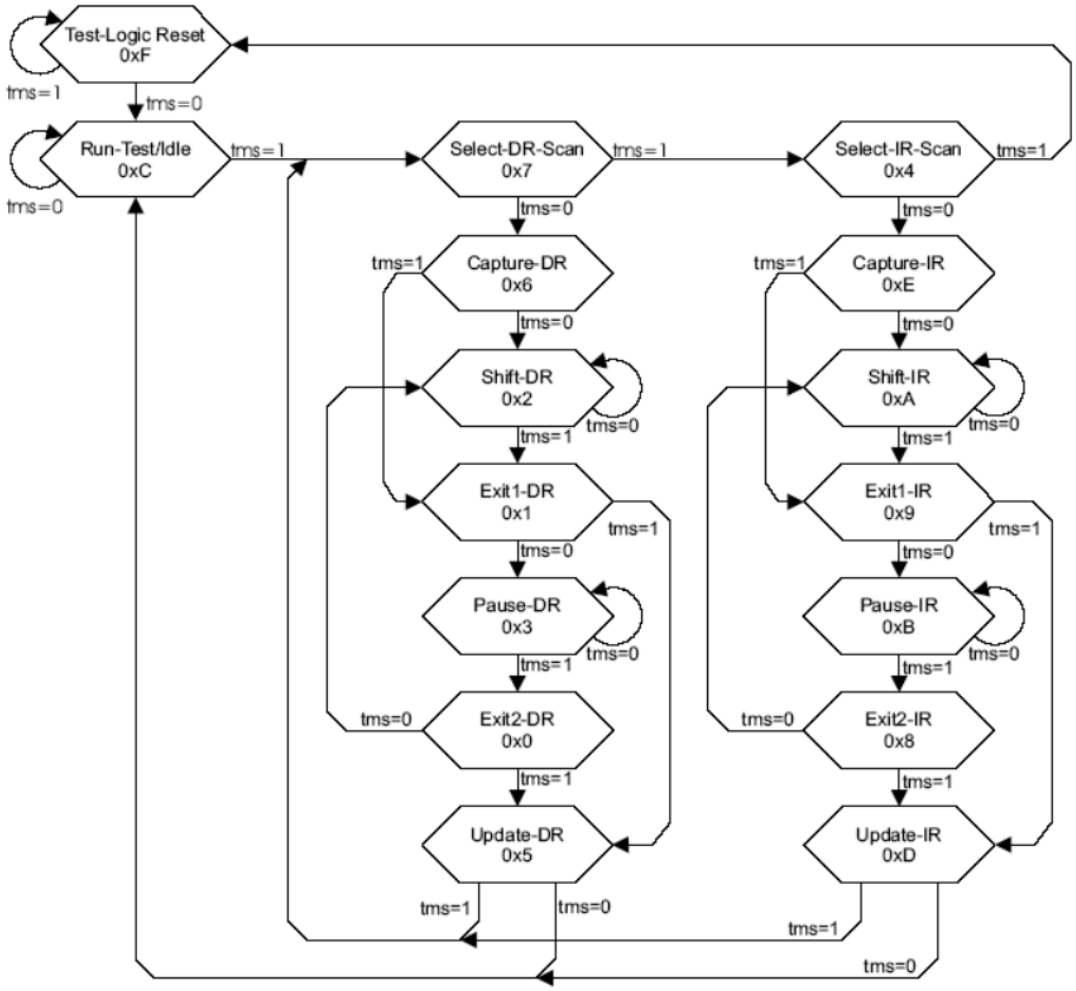


图 2-3 TAP 状态转化

(1) Test-Logic Reset

这是 TAP 控制器在开机后自动进入的状态，也是接收到 TRST 信号后进入的状态。从图 2-3 可以看到，任意一个状态下，只要连续接收五次 TMS=1，TAP 控制器都会进入到此状态，这就是 TRST 不是必需信号的原因。

(2) Run-Test/Idle

这是 TAP 控制器的一个中间状态，一般而言只用于状态的过渡。但在我们系统的实现中，这是一个比较重要的状态，所有的 EJTAG 高级功能（详见 4.2 节）所封装的函数都在此状态下调用，且在调用完毕后返回此状态。因此，对于 EJTAG 高级功能而言，TAP 状态是透明的，这大大方便了上层程序的编写。

(3) Select-DR-Scan

作为一个中间状态，Select-DR-Scan 所起到的作用类似于高级程序设计中的 if 语句。如果要对 EJTAG 的数据寄存器进行操作，那么随后 TMS=0，若是其他操作，随后便输入 TMS=1。

(4) Capture-DR

正如名字里的意思，此状态下，数据寄存器将“捕获”芯片管脚上的信号。

(5) Shift-DR

Shift-DR 和 Shift-IR 在我看来是 TAP 状态机最为核心的两个状态，数据的输入和输出都是在这个状态下完成。此状态下，只要 TMS=0 保持不变，就可以一直维持这个状态。此状态下每一个时钟周期，数据寄存器都会从 TDI 接收一比特的数据，而将最后一比特数据自 TDO 移出。这些移出的数据，可能就是程序关心的输出。直到 TMS=1 时，状态机进入下一状态，留在数据寄存器中的数据，便是真正输入的数据。这些真正输入的数据，也将在之后的 Update-DR 状态下被提交，进而生效。

(6) Update-DR

本状态下，之前在 Shift-DR 中数据寄存器保留下来的数据会加载到对应的芯片管脚上，可看作是 Capture-DR 的逆过程。

简单的说来，操作 TAP 控制器一般是如下流程（以 Run-Test/Idle 起，Run-Test/Idle 止为例）：先通过设置 TMS 进入 Shift-IR 状态（TMS 依次输入 1100 即可，下同），设置五位的 IR 寄存器来选择指令（详见 2.6 节）；然后经过 Update-IR 返回 Run-Test/Idle。再之后进入 Shift-DR 来设置 DR 寄存器来设置刚刚所选指令的值，最后经过 Upadate-IR 返回 Run-Test/Idle。比如，在 Shift-IR 状态下，TDI 输入 01010，可选中 EJTAG 控制寄存器；随后在 Shift-DR 状态下，TDI 即可设置 EJTAG 控制寄存器的值。

2.6 TAP 指令

EJTAG 在 Shift-IR 下能够选择的指令如表 2-1 所示，这里的所有指令都是 5bit

长度[22] 。

表 2-1 TAP 指令

Code	Instruction	Function
All 0's	(Free for other use)	Free for other use, such as JTAG boundary scan
0x01	IDCODE	Selects Device Identification (ID) register
0x02	(Free for other use)	Free for other use, such as JTAG boundary scan
0x03	IMPCODE	Selects Implementation register
0x04 - 0x07	(Free for other use)	Free for other use, such as JTAG boundary scan
0x08	ADDRESS	Selects Address register
0x09	DATA	Selects Data register
0x0A	CONTROL	Selects EJTAG Control register
0x0B	ALL	Selects the Address, Data and EJTAG Control registers
0x0C	EJTAGBOOT	Makes the processor take a debug exception after reset
0x0D	NORMALBOOT	Makes the processor execute the reset handler after reset
0x0E	FASTDATA	Selects the Data and Fastdata registers
0x0F	(EJTAG reserved)	Reserved for future EJTAG use
0x10	TCBCONTROLA	Selects the control register *TCBTraceControl* in the Trace Control Block
0x11	TCBCONTROLB	Selects another trace control block register
0x12	TCBDATA	Used to access the registers specified by the $TCBCONTROLB_{REG}$ field and transfers data between the TAP and the TCB control register
0x13	TCBCONTROLC	Selects another trace control block register
0x14	PCSAMPLE	Selects the PCsample register
0x15 - 0x1B	(EJTAG reserved)	Reserved for future EJTAG use
0x1C - All 1's	(Free for other use)	Free for other use, such as JTAG boundary scan
All 1's	BYPASS	Select Bypass register

其中，比较常用的有 0x08、0x09、0x0A，即设置 EJTAG 的地址、数据和控制寄存器。虽然 0x0B 可以同时对地址、数据和控制寄存器进行控制，但是项目中出于效率考虑并没有采用。另外，通过 0x1f 设置 bypass 状态，也在控制执行核心上起了极为重要的作用。

2.7 多核处理

由于龙芯 3A 是四核处理器，而四核是可以相互独立进行调试和运行的。因此，需要有一种机制来对多核进行管理。

采用多条边界扫描链或是个好方法，然而却会增加一定的成本和复杂性。龙芯

3A 只使用了一条边界扫描链，采用菊型链[23] 的方式对多核进行操作。所谓菊型链，即一个接一个内核顺序连接，如图 2-4 所示：

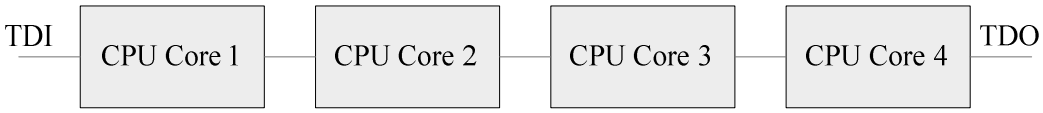


图 2-4 菊型链

边界扫描链能够控制何种寄存器，取决于当前 TAP 状态：如果是 Shift-IR 状态，边界扫描链连接在指令寄存器上，四核长度均为 5bit；如果是 Shift-DR，则四核所连接寄存器取决于之前指令寄存器的设置，可能长度不同。

当从扫描链输入数据时，操作有些类似于数据结构中队列的入队和出队，其队列长度一定，且只在入队同时出队（反之要出队也必须入队）。输入从核心 1 进入（入队），同时将核心 1 内原有数据向后“挤”。TDI 输入了几位数据，就会有几位数据被挤出核心 1（出队）。这些被挤出的数据同时相当于核心 2 的输入，核心 2 中的数据也同时后移……这种情况下，即使只有一位的输入，也会导致所有核心所有的数据向后移动一位，且核心 4 原有最后一位数据从 TDO 中移出。

以上提到的只是边界扫描的应用。而要针对特定核心设置调试与否，还需要看到上节提到的 TAP 指令。如希望在操作时跳过某特定内核，可在 Shift-IR 时将该内核设置为 Bypass（给该位 IR 设置为 11111）。设置为 Shift-IR 状态的核心在 Shift-DR 时只有一位的数据，且该位数据并无实际含义。这样，便可以将此核心“忽略”，达到选择特定核心的目的。

2.8 GDB 远程串行协议

由于 EJTAG 调试端最终需同 epdb 相联进行调试，因此两方需要约定通信协议。GDB 调试软件在行业里获得了广泛的使用和认可，因此，epdb 和 Server 端之间的通信也使用 GDB 远程串行协议。

GDB 远程串行协议中，所有的命令和相应都以包的形式发送[24] 。每个包由美元符“$”开始，紧接着是包的实际内容 packet-data，最后是结束符“#”和两位 16 进制的数字校验码 checksum。即

$packet-data#checksum

epdb 每发送一个包，Server 端也就相应的给出一次响应。每个包内容中可以包含除了“$”、“#”以外的任意字符，而包内容中的字段可以以“，”、“；”或“：”隔开。

表 2-2 是一份简明的列表，描述了项目相关的较为常用的一些指令包以及其响应包的格式。完整的列表可以参考文献[25] 的附录。

表 2-2 GDB 远程串行协议包描述简表

maddr,length	读取地址自 addr 起长度为 length 字节的内存。注意这地址 addr 可能并没有对齐。 响应： ‘XX...’：内存的内容，每个字节以两个 16 进制数表示。当服务器只能够获取其中一部分内存内容的时候，返回值可能没有请求的长度那么长。 ‘ENN’：NN 是错误码。
Xaddr,length:XX...	给内存写二进制数据。addr 参数是写入内存的地址，length 是写入字节长度，‘XX...’是二进制数据。 响应： ‘OK’：表示成功。 ‘ENN’：表示错误。
GXX...	写通用寄存器。XX 是待写数据。 响应： ‘OK’：表示成功。 ‘ENN’：表示错误。
g	读取通用寄存器。 响应： ‘XX...’：每字节寄存器数据用两位 16 进制数字表示。寄存器的字节序被转换为目标字节序。 ‘ENN’：表示错误。
s[addr]	单步调试。addr 是恢复地址。若未提供 addr 参数，则恢复为原地址。 响应： 参见表 2-3——停止应答报文（Stop Reply Packets）。
qC	返回当前线程 ID。 响应： ‘QCpid’：pid 为 16 进制的无符号数，表示线程 ID。 其他响应：表示老旧的 pid。
vCont[;action[:tid]]...	恢复前端。可以给不同的线程(tid)设定不同的动作(action)。如果某个动作没有指定线程，那么动作可以被任意线程执行；而若是没有指定默认动作，那么其他的线程需保持停止状态。指定多个动作会导致错误，不指定动作同样会导致错误。线程 ID 以 16 进制形式传递。当前支持的动作有：

续表 2-2

	‘c’：继续。 ‘Csig’：伴随着信号 sig 继续。sig 为 2 位 16 进制数。 ‘s’：单步。 ‘Ssig’：伴随着信号 sig 单步。sig 为 2 位 16 进制数。 响应： 参见表 2-3——停止应答报文（Stop Reply Packets）。
ztype,addr,length Ztype,addr,length	插入（Z）或删除（z）一个 type 类型的断点/观察点。addr 为断点/观察点地址，length 表示断点覆盖接下来字节长度。 type 可以为’0’/’1’/’2’/’3’/’4’。’0’代表内存断点，’1’代表硬件断点，’2’代表写观察点，’3’代表读观察点，’4’代表访问观察点。 响应： ‘OK’：表示成功。 ‘ENN’：表示错误。 ‘’：表示不支持。
qNonStop:1 qNonStop:0	进入 non-stop (‘QNonStop:1’) 或 all-stop (‘QNonStop:0’) 模式。

上表表示了部分常用的 GDB 远程串行协议包，停止应答报文则参见表 2-3。

表 2-3 停止应答报文（Stop Reply Packets）

SAA	程序接收到信号“AA”（二位十六进制数字）。
TAAn1:r1;n2:r2;...	程序接收到信号“AA”（二位十六进制数字）。 若不使用‘n:r’对来在停止应答报文中直接传递额外的重要寄存器值或是其它重要信息，它和 S 报文效果是一样的。
WAA	进程退出，AA 是退出状态。只对特定的目标应用。
XAA	进程以信号 AA 结束。
OXX...	‘XX...’是 16 进制编码的 ascii 数据，将写往程序的命令行输出。当程序在运行，而调试器等待‘W’‘T’等报文时，随时都有可能发生。
Fcall-id,parameter...	call-id 用来标识究竟是哪个宿主系统调用应该得到响应。

同 GDB 远程串行协议包描述简表相同，更多的信息可参考[25] ，在本文稍后会看到这些报文的应用。

2.9 本章小结

本章介绍了 EJTAG 调试的相关基础理论，为下文的系统实现打下基础。除了 GDB 远程串行协议，本章的其他小节都是和指令级调试相关的。这也从一个侧面反映了 EJTAG 是一种底层硬件调试方式。

3 系统结构

3.1 调试系统模块结构

项目组最终的目标是对基于龙芯 3A CPU 搭建的龙芯计算机平台进行调试，其调试对象不仅包含上文中叙述较多的 Linux 内核调试部分，也包含了龙芯 3A 平台上运行的应用程序。调试系统模块结构如图 3-1 所示。

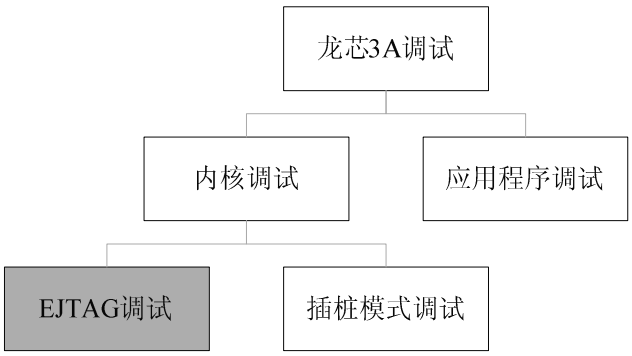


图 3-1 调试模块结构

应用调试部分，其基本功能是对龙芯 3A 平台上运行的应用程序进行调试。然而和常见的应用调试软件不同，本项目的应用调试将重心放在多线程调试上，其错误回放、检查点等功能使得对多线程应用程序的调试代价大幅度降低。另外，将要研究的并行度分析系统等功能，也跳出了传统的调试范畴，通过对应用程序进行诊断分析，来帮助用户更容易的对代码进行优化。

内核调试部分，调试系统提供了 EJTAG 调试和插桩模式调试两种方法。正如 1.2 节中所描述的，插桩模式有着极为明显的速度优势，而 EJTAG 调试能够在系统尚未完全启动的时候就开始调试。因此，这两种调试方案都有其价值，不可替代。

鉴于我个人的主要工作主要集中在 EJTAG 调试部分，下面进一步阐述 EJTAG 部分的结构及设计思路。

3.2 EJTAG 调试模块结构

EJTAG 的调试部分可如图 3-2 搭建，这种开发机与目标机之间跨平台进行的调

试便是交叉调试[26] 。

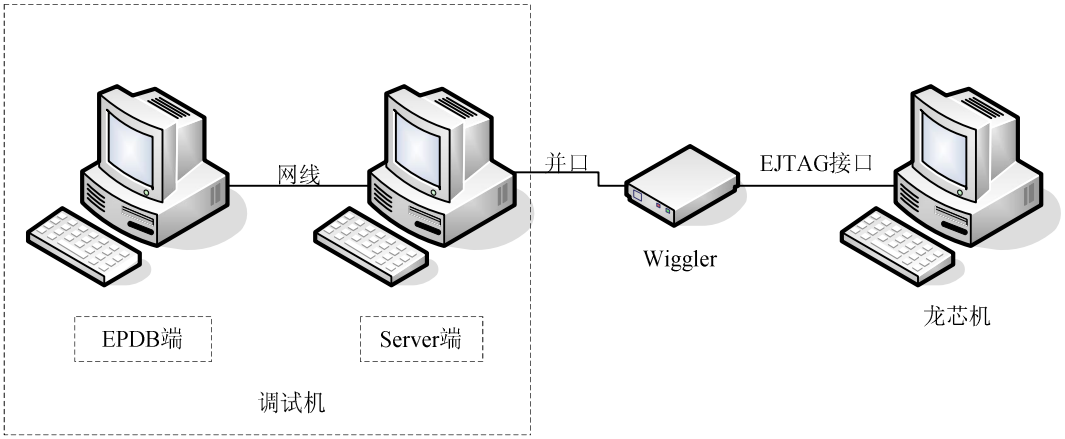


图 3-2 EJTAG 调试部分结构图

图中共有三台 PC，分别为 epdb 端，Server 端和龙芯机端。龙芯机端为待调试机器，也被称为目标机；epdb 端是一个调试软件的前端，直接与用户进行交互，其框架已由应用程序调试小组基本实现。epdb 可以以 GDB 远程串行协议[25] 将用户的需求以包的形式通过 Socket 发送出来，实际上，epdb 实现了源码级调试到指令级调试的翻译。

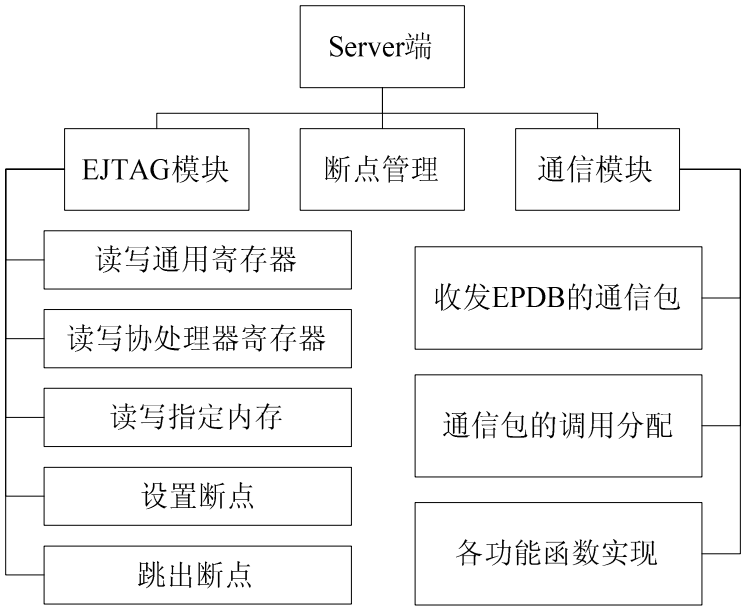


图 3-3 Server 端结构框架

Server 端是将要实现的主要部分，一方面，它需要能够通过 Wiggler 和龙芯机相连，使用 EJTAG 对龙芯机进行指令级调试；另一方面，它需要能够与 epdb 端进行

通信，将 epdb 端的指令解释执行，并将指令调试的结果加以组织后返回。除此之外，由于 EJTAG 处理断点的方式和 epdb 的不尽相同，因此 Server 还需要对断点进行一些管理。Server 端的设计结构如图 3-3 所示。

虽然 epdb 端和 Server 端可以是物理上分开的两台主机，仅使用 Socket 传递消息，但为方便起见，本项目中这两个部分运行于同一台计算机，但依旧使用 Socket 进行通信。

系统正常运行时，epdb 端面向用户，将用户输入的指令转化为一个个 GDB 远程串行协议包发送给 Server 端。Server 端将远程串行协议包解析，通过 EJTAG 对龙芯端进行调试，并将龙芯端的调试结果以 GDB 远程串行协议包返回给 epdb 端。最终 epdb 端解析 Server 端的返回包，经过处理后将结果返回给用户，其流程如图 3-4 EJTAG 源码级调试流程图 3-4 所示。

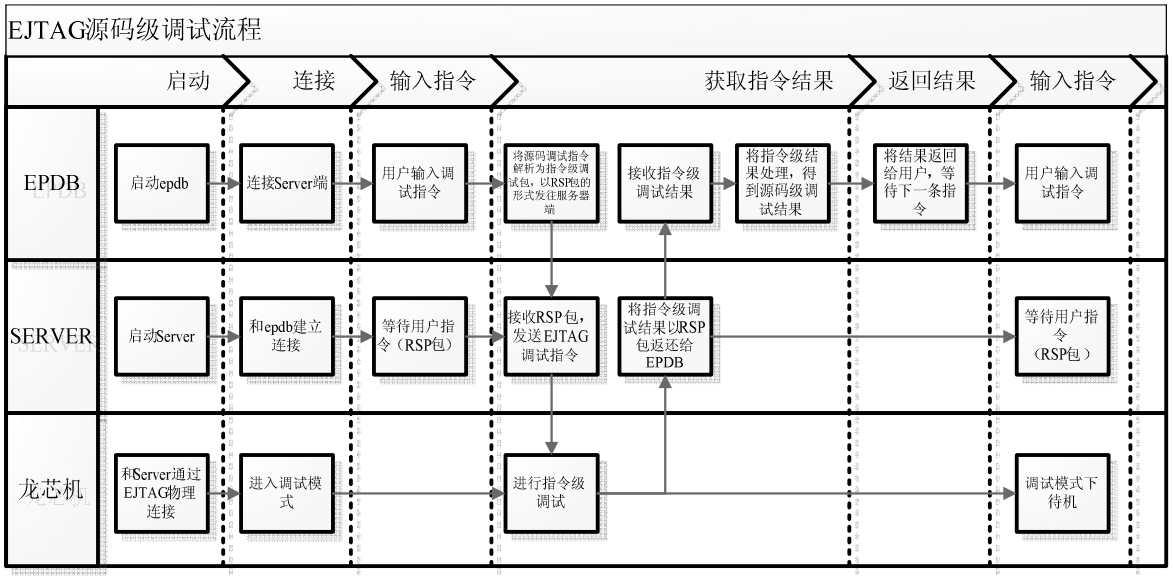


图 3-4 EJTAG 源码级调试流程

3.3 本章小结

本章先介绍了整个调试系统的模块结构，随后进一步介绍了 EJTAG 调试模块及其地位。下面两章，将先介绍指令级调试，随后在其基础上实现源码级调试。

4 指令级调试

为了使 PC 能与龙芯机的 EJTAG 接口通信，项目中采用 Wiggler 将 PC 的并口同 EJTAG 连接起来。Wiggler 其实质可以看作是一个接口转换器，负责将并口引脚映射到 EJTAG 引脚。

图 4-1 是一个市售 JTAG 用 Wiggler 电路，实际实验时，EJTAG 端由 20 针 JTAG 接口修正为 14 针 EJTAG 接口。

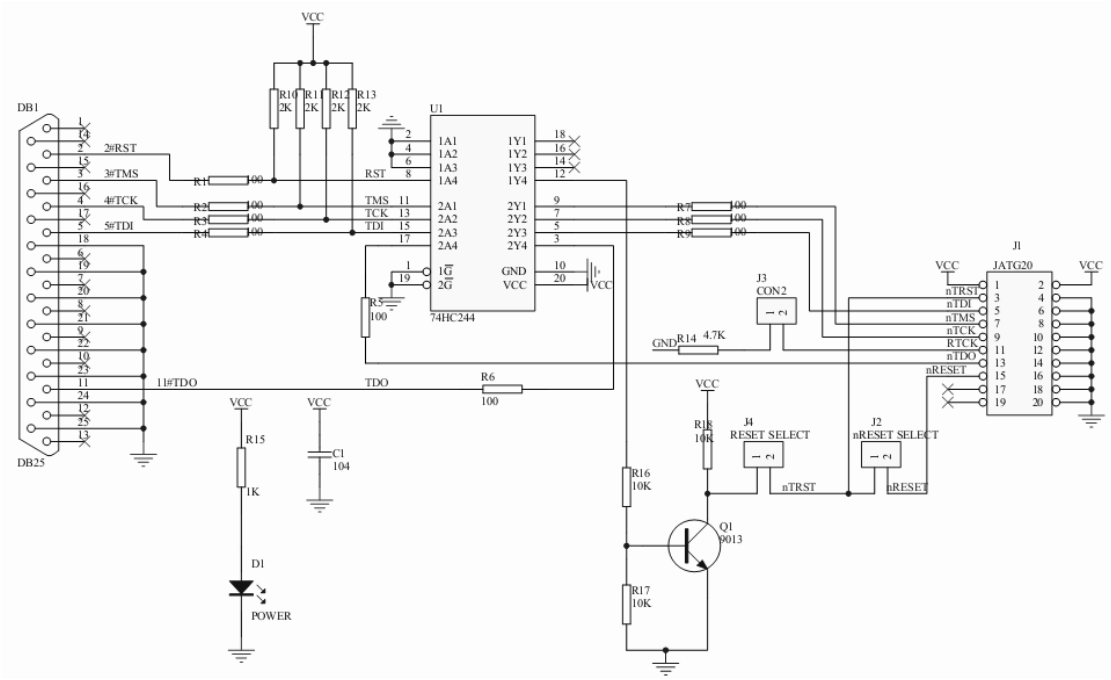


图 4-1 市售 JTAG 用 Wiggler 电路图

从功能上而言，为了实现读写寄存器等高级功能，需要先实现让 CPU 执行一条二进制指令（即发起处理器访问）。因此可以此为界，分别完成 EJTAG 的底层实现和高级功能。底层功能通过在不同的 TAP 状态间进行跳转，来逐步实现发起一条处理器访问；高级功能只需要调用发起处理器访问的函数来进行具体设计，不必考虑底层 EJTAG 的运行情况。这样一来，TAP 状态对高级功能而言成为了透明的，大大方便了高级功能的编写。

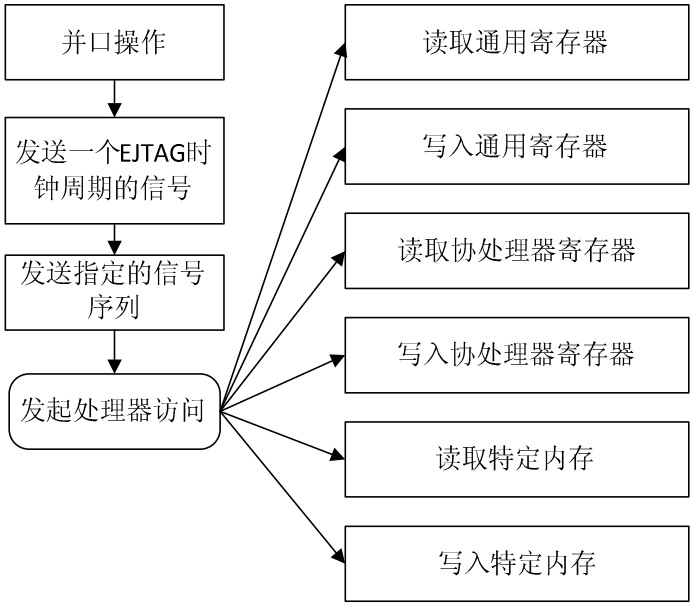


图 4-2 EJTAG 部分设计框架

以上是本机 EJTAG 指令级调试所需要实现的功能。由于最终需要与 epdb 之间通过 GDB 远程串行协议进行通信，所以还编写了对寄存器的批量读写函数，以提高工作效率。

4.1 EJTAG 底层设计

EJTAG 底层的目标是执行一条二进制 CPU 指令。由上文的 EJTAG 基本原理可知，执行一条指令可分层逐步实现：

(1) 给并口（映射到 EJTAG 口）发送一个时钟周期的信号；

(2) 给并口发送指定的信号序列；

(3) 发起处理器访问。

4.1.1 发送一个 EJTAG 时钟周期的信号

由于并口和 EJTAG 通过 Wiggler 相连，在 Linux 下直接对并口的操作即可通过 Wiggler 转换成对 EJTAG 信号的操作。

Linux 下有 outb 函数，可在 Root 权限下直接控制并口输出[27] 。PC 并口如图 4-3 所示[28] ：

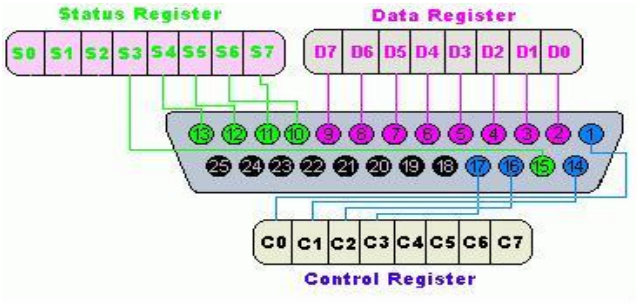


图 4-3 PC 并口

根据 Wiggler 电路，可知 TCK 连接并口 4 号针脚，对应数据寄存器的 d2 位；TMS 连接并口 3 号针脚，对应数据寄存器 d1 位；TDI 连接并口 5 号针脚，对应数据寄存器 d3 位；TDO 连接并口 11 号针脚，对应状态寄存器的 S7 位。

因此，要输出 TAP_TDI，可向并口数据寄存器输出 1<<3，对 LPT1，并口数据寄存器地址为 0x378。类似的，TAP_TMS 输出 1<<1，TAP_TCK 输出 1<<2。为求清晰，代码中可以使用宏来定义 TAP_TDI 等量。同理，TAP_TDO 是状态寄存器（地址为 0x379）的 S7 位，输入 1<<7（即 0x80）。

EJTAG 的四个有效引脚之中，TAP_TDO 是输出信号，TAP_TCK 是时钟信号。TAP_TMS 控制 TAP 状态的切换，而 TAP_TDI 控制 Shift-IR 和 Shift-DR 下的数据移位输入。

由于对 EJTAG 而言，每次信号生效时刻是在 TAP_TCK 上跳沿，因此，为输出一次 TAP_TDI/TAP_TMS，可以如下操作：

```
outb(value, 0x378);
outb(value | TAP_TCK, 0x378);
outb(value, 0x378);
```

其中，value 只会是 TAP_TDI 和 TAP_TMS 的笛卡尔积。尽管理论上向 EJTAG 的输入只在时钟上跳沿生效，但是实验中若不加上述第三条语句则不能正常运行。这或许是 Wiggler 转换电路所引起的细微差异。

另外，由边界扫描的特性，可以在写入数据的同时，使用 Linux 库函数 inb 将 TAP_TDO 的值读取出来。

上述过程可以封装成一个函数：

```
//向 TAP 输入一次，返回 TDO 输出
int output(uint8_t value);
```

4.1.2 发送指定的信号序列

有了 output 函数，其实就可以对 TAP 控制器进行操作了。不过为了方便函数调用，精简代码，可以编写一个函数，在 Shift-IR 和 Shift-DR 下对 TAP_TDI 以字符串的方式进行输入输出。

```
//向 TDI 发送移位指令，返回从 TDO 取出挤出的数据，挤出的数据为正序。
//注意只能在 SHIFT-DR、SHIFT-IR 下使用。
/*
 * code:           发送指令序列
 * length:         指令长度
 * bLastWithTMS:   表示最后一次 TDI 输入是否需要与 TMS 相或
 */
char* instruction(const char* code, int length, int bLastWithTMS);
```

code 参数是此时由 TDI 口传入的 0/1 序列，length 为序列长度，而 bLastTMS 则表示最后一次 TDI 输入是否需要与 TMS 相或，这在控制 TAP 流程时有效。返回值则是从 TAP_TDO 挤出的 EJTAG 输出。

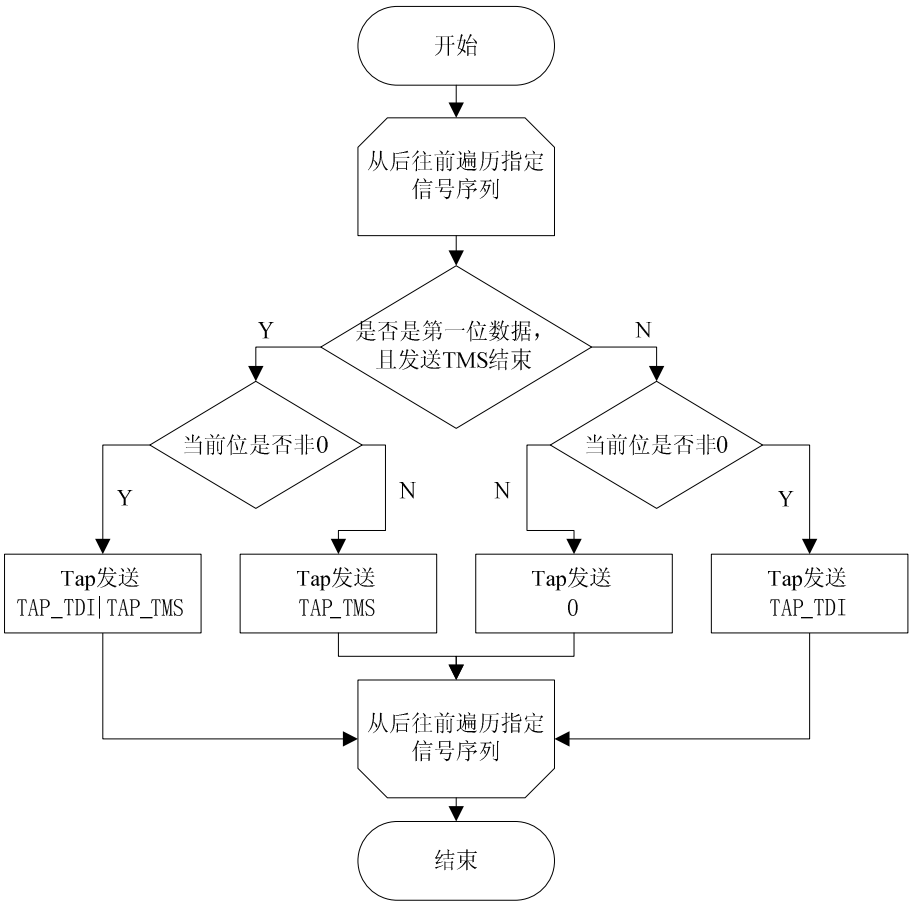


图 4-4 发送指定的信号序列

如此，可将 output 输出 TAP_TDI 的功能封装。在此基础上编写的代码，output 可以只担负切换 TAP 状态的职责。而在 Shift 状态下的位移，则交予 instruction 操作。instruction 函数的流程如图 4-4 所示。

4.1.3 发起处理器访问

EJTAG 机制中，针对处理器访问，有着较为固定的处理流程[29]，如图 4-5 所示：

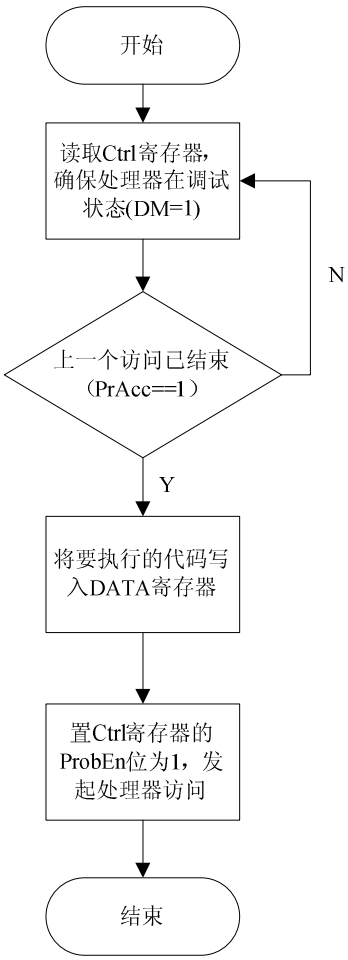


图 4-5 处理器访问

(1) 首先读取 EJTAG 控制寄存器（ECR），确保 ECR 的 DM 位为 1，若 PrAcc 位为 0，则重复读取 ECR，直至其为 1；

(2) 然后进入 Shift-IR 状态，选择 DATA 寄存器；

(3) 随后进入 Shift-DR 状态，设置 DATA 寄存器为二进制指令；

(4) 再读取 ECR，将 PrAcc 位写 0 压回，即可完成处理器的访问。

此函数的定义如下：

```
//在 Run-Test/Idle 状态下，向 select 的核心发送 32 位 cpu 指令，
//执行完毕返回 Run-Test/Idle 状态
/*
 *  code    :   cpu 指令，比如 DERET，比如其他经由汇编得到的四字节代码。
 *              指令可以以 2 进制或是 16 进制。
 *  select  :   本质为一个 0~15 的整数，写成二进制为四位数字。
 *              四位分别说明了四核处理器是否为 select 状态，是则为 1。
 *              比如 1010 说明一、三核为 select 状态，0000 则都不在 select 状态。
 */
void send_cpu_ins(const char* code, int select);
```

一般而言，上述操作足以发起处理器访问，不过上述语句只负责了向 EJTAG 写入数据，对于 store 类语句（mips64 中主要是 sd 语句，可将寄存器写入指定的 dmseg 内存之中），则还需要进行额外的操作，才能将这些“内存”中的数据读出。这些额外的操作，封装在 things_after_sd 函数中，此函数需紧接 send_cpu_ins 函数使用。

```
//在进行 SW 之类的写内存操作之后，需要从 TAP 中把它取回来。
//本函数应当在调用 send_cpu_ins（SD）之后调用，
//执行完毕返回 Run-Test/Idle 状态。
/*
 * select  :选择执行 CPU
 * return  :返回存入内存的数据
 */
char** things_after_sd(int select);
```

此函数先进入 Shift-IR，选中 DATA 寄存器，随后在 Shift-DR 中，将数据读出。此即为 store 类指令写入内存的数据。最后，函数将 ECR 的 PrAcc 位置 0，表示写操作结束。函数返回存入内存中的数据。

4.2 EJTAG 高级功能设计

高级功能，只是相对于 EJTAG 底层功能而言，但也还属于指令级调试的范畴。其主要包括：读写特定通用寄存器、读写所有通用寄存器、读写特定协处理器寄存器、读写所有协处理器寄存器、读写特定内存、给指定地址设置断点、跳出断点继续运行。

4.2.1 读取特定通用寄存器

读取通用寄存器，其基本原理是将需要读取的寄存器使用 sd 指令写入 dmseg 内存中，随后通过 things_after_sd 函数将其读取出来。注意此处的汇编指令并不是常见的 x86 汇编，而是 MIPS64 汇编[30]，而这也正是交叉调试的体现。以 k1 为例，其执行的处理器访问代码如下（参数 select 代表选中执行代码的 CPU 核心，下同）：

```
send_cpu_ins("40baf800",select); // dmtc0 k0,$31(DESAVE)
send_cpu_ins("3c1aff20",select); // lui k0,0xff20
send_cpu_ins("375a7000",select); // ori k0,k0,0x7000
send_cpu_ins("ff5b0000",select); // sd k1,0(k0)
send_cpu_ins("403af800",select); // dmfc0 k0,$31
```

从代码可以看出，除去对于临时寄存器 k0 的数据保存与恢复，总共只有三句二进制代码，而其中两句还是给临时寄存器 k0 赋 dmseg 值的。真正的核心代码就是一句 sd 语句，以及与它相关的 things_after_sd。SD 语句负责将寄存器的值写入 EJTAG 的 DATA 寄存器，而 things_after_sd 负责将 EJTAG 的 DATA 寄存器中的值读取出来。

不过，鉴于读取的寄存器本身可变，因此 sd 语句的二进制码是可变量。由表 4-1 可算出其具体值。

表 4-1 SD 机器指令

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
63	b	t		offset				sd t,o(b)	MIPS64

这样，以 sd k1,0(k0)为例，写成二进制就是：

“11111111010110110000000000000000”，

也就是十六进制的“FF5B0000”。

另外要注意的是，读取 k0 寄存器时，不能使用其自身作为临时变量。此时应当选取其他寄存器作为临时值。

最终，可以得到函数 read_gen_reg：

```
//读取通用寄存器
/*
 * number  :   通用寄存器序号，自 0 至 31
 * select  :   本质为一个 0~15 的整数，写成二进制为四位数字。
```

```
 *                四位分别说明了四核处理器是否为 select 状态，是则为 1。
 *                比如 1010 说明一、三核为 select 状态，0000 则都不在 select 状态。
 * return  :      返回读取出的寄存器数据
 */
char** read_gen_reg(int number, int select);
```

其处理流程如图 4-6 所示：

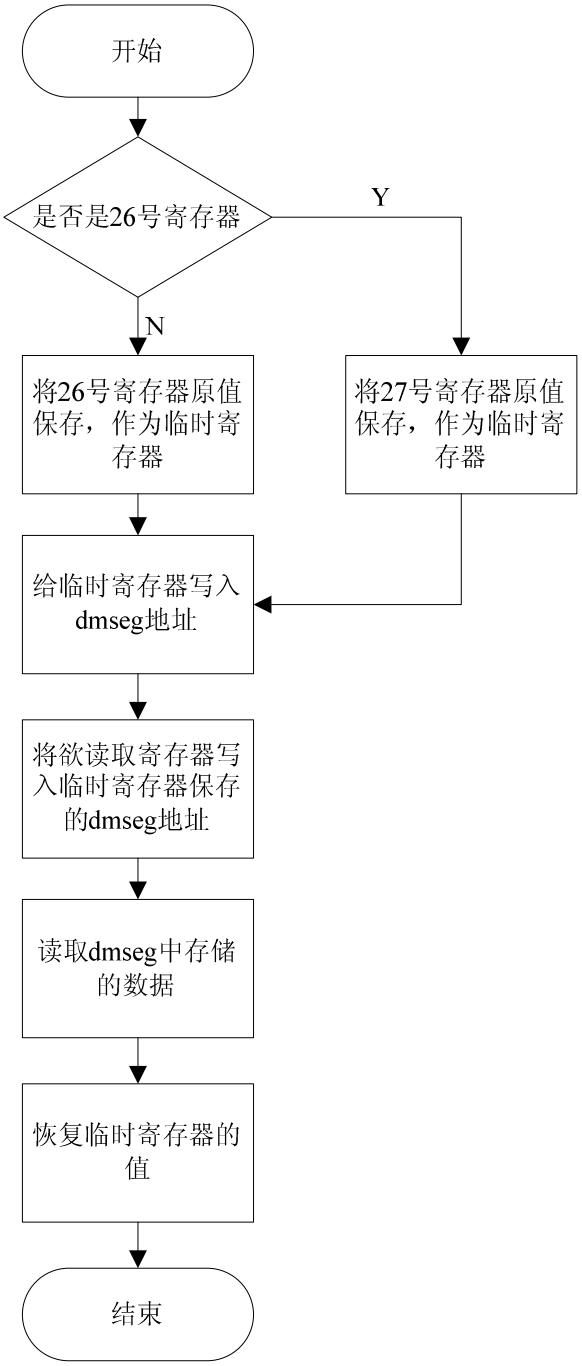


图 4-6 读取特定通用寄存器

4.2.2 写入特定通用寄存器

原本，和读取特定通用寄存器对应，写入特定通用寄存器应当使用 ld 汇编指令，将内存中的数值载入到寄存器中。然而，由于本身函数是将输入参数写入指定寄存器，若采用由内存载入的方式就绕圈子了。因此，可以直接通过 lui 等汇编指令，直接将立即数给“赋值”给指定寄存器。

以写入 k1 寄存器 0x1234567890abcdef 为例，其执行的处理器访问代码如下：

```
send_cpu_ins("3c1b1234",select); //    lui k1,0x1234
send_cpu_ins("377b5678",select); //    ori k1,k1,0x5678
send_cpu_ins("001bdc38",select); //    dsll k1,k1,16
send_cpu_ins("377b90ab",select); //    ori k1,k1,0x90ab
send_cpu_ins("001bdc38",select); //    dsll k1,k1,16
send_cpu_ins("377bcdef",select); //    ori k1,k1,0xcdef
```

而将一个 16 位 16 进制字符串更细化分配给每句 CPU 访问，则是函数的工作。另一方面，和 read_gen_reg 中的 sd 类似，lui、ori 以及 dsll 都是和寄存器相关的。因此，对于不同的寄存器，有不同的指令。

表 4-2 LUI、ORI、DSLL 机器指令

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
15	0	d		(unsigned) const				**lui d,s,const**	
13	s	d		(unsigned) const				**ori d,s,const**	
0	0	w	d	shf			56	**dsll d,w,shf**	MIP64

其处理流程如图 4-7 所示：

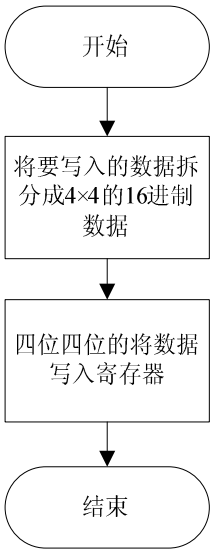


图 4-7 写入特定通用寄存器

最终可以得到函数 write_gen_reg：

```
//写入通用寄存器
/*
 * number  :   通用寄存器序号，自 0 至 31
 * data    :   写入寄存器的值，16 进制形式，要求长度为 16
 * select  :   本质为一个 0~15 的整数，写成二进制为四位数字。
 *             四位分别说明了四核处理器是否为 select 状态，是则为 1。
 *             比如 1010 说明一、三核为 select 状态，0000 则都不在 select 状态。
 */
void write_gen_reg(int number, const char* data, int select);
```

4.2.3 读取所有通用寄存器

读取所有通用寄存器，理所当然的可以遍历调用读取指定的通用寄存器的函数。但是由于遍历所有寄存器在性能上还有较大的优化余地，而且此函数比起读取指定通用寄存器更加通用，因此还是有必要单独写一个，进行性能上的优化。

之所以遍历会比较慢，主要是因为每读一个寄存器，都将中途使用的临时寄存器进行了进出栈和赋值操作，而这些并不是每次读取寄存器都必须单独进行的。而实践也证明，进行优化之后的读取所有通用寄存器函数执行时间只为遍历的一半不到。优化效果还是比较显著的。

读取某个寄存器需要一个临时寄存器，而临时寄存器不能为待读取的寄存器本身。因此，采取这样的策略：读取 0～26 号寄存器时，使用 27 号寄存器（k1 寄存器）作为临时寄存器；读取 27～31 号寄存器时，使用 26 号寄存器（k0 寄存器）作为临时寄存器。两次分别遍历，每次遍历前压栈临时寄存器，设置临时寄存器，遍历后出栈临时寄存器，恢复临时寄存器原值。

于是有了读取所有通用寄存器的函数 read_all_gen_regs：

```
//读取所有寄存器，所读数据放入全局变量 core 中
/*
 * select  :   本质为一个 0~15 的整数，写成二进制为四位数字。
 *             四位分别说明了四核处理器是否为 select 状态，是则为 1。
 *             比如 1010 说明一、三核为 select 状态，0000 则都不在 select 状态。
 */
void read_all_gen_regs(int select);
```

其处理流程如图 4-8 所示：

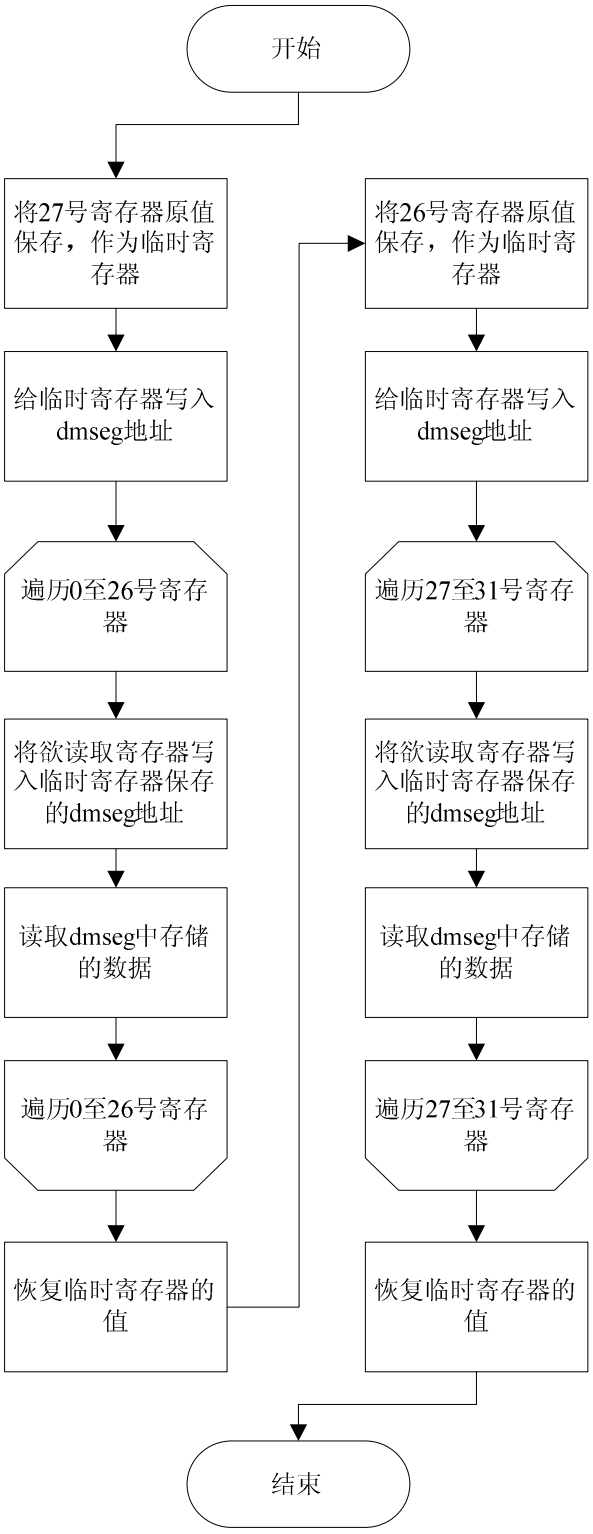


图 4-8 读取所有通用寄存器

为了方便对数据进行操作和管理，设立了全局变量来管理部分核心数据，其中就有通用寄存器。因此，此函数读取的所有通用寄存器写入该全局量。该全局量的

定义如下：

```
typedef struct coreStatus
{
//是否跳过该 CPU 核心，0 为不跳过，1 为跳过
int bypass;

//通用寄存器，32 个，每个 16 位 16 进制数
char gen_regs[32][17];

/*
* cp0 寄存器，需集体获取的共如下 6 个，每个 16 位 16 进制数
* "cp0_status","lo"(entrylo),"hi"(entryhi),
* "cp0_badvaddr","cp0_cause","cp0_epc"
* 另外有不向外输出，却起到控制作用的，依次有
* "cp0_debug","cp0_depc"
*/
char cp0_regs[8][17];

/*
* cp1 寄存器，共 33 个，每个 16 位 16 进制数
* 前 32 个依次为"f0"~"f31"，最后一个为"fcr31"（现称 fcsr）
*/
char cp1_regs[33][17];

//Ejtag 控制寄存器，32 位二进制数
char ejtag_ctrl_reg[33];

//Ejtag 地址寄存器，16 位 16 进制数
char ejtag_addr_reg[17];
} coreStatus;

//四个核心的状态
coreStatus core[4];
```

4.2.4 写入所有通用寄存器

由于写入通用寄存器没有使用临时寄存器，因此并不像读取所有通用寄存器般能够进行大幅度优化。而为了实现函数集的完整，也单独给其封装一个函数来写入所有通用寄存器，不过其本质还是对单个通用寄存器的遍历。如此，可以得到write_all_gen_regs：

```
//写入所有通用寄存器
/*
 * data    :   一个(char (*)[17])的数组，储存要写入的数据。
 * select  :   本质为一个 0~15 的整数，写成二进制为四位数字。
 *             四位分别说明了四核处理器是否为 select 状态，是则为 1。
 *             比如 1010 说明一、三核为 select 状态，0000 则都不在 select 状态。
```

```
 */
void write_all_gen_regs(const char* data[], int select);
```

其处理流程如图 4-9 所示：

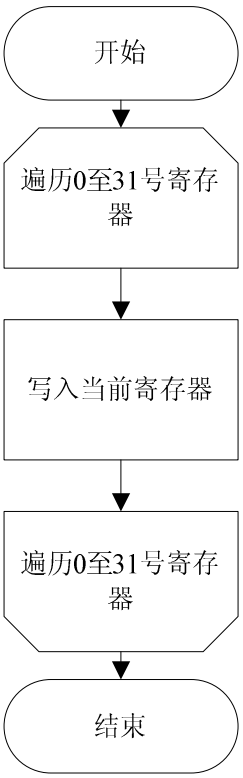


图 4-9 写入所有通用寄存器

4.2.5 读取特定协处理器寄存器

因为有着对 CPU 的控制和浮点运算的需求，MIPS 设计了一系列的协处理器[31]。此处设计的函数需要能够使用 EJTAG 来读取部分 cp0 寄存器、所有 cp1 寄存器以及一个 cp1 控制寄存器。

类似于通用寄存器，协处理器寄存器也记载在全局量 core 中。因此，读取出来的协处理器直接放入全局量 core 中，不再以指针返回。

读取协处理器的寄存器，主要操作原理是将协处理器寄存器的值取出到通用寄存器，再按照读取通用寄存器的读取方式将协处理器寄存器值读取出来。而相对复杂的地方在于，对于不同位数的寄存器，或是对于控制寄存器，都需要使用不同的机器指令来将它取出到通用寄存器中。

一方面，这些协处理器寄存器有的 32 位，有的 64 位。对于 32 位的寄存器，cp0 和 cp1 分别使用 mfc0/mfc1 指令将其移入通用寄存器；相应的，64 位则使用 dmfc0/dmfc1 指令。另一方面，根据需求，还需要读取一个 cp1 控制寄存器。由于它是 32 位的，因此对它而言，此处应当使用 cfc1 指令。

表 4-3 取协处理器寄存器指令

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
16	0	t		cs	0		0	**mfc0 t,cs**	
16	1	t		cs	0		0	**dmfc0 t,cs**	MIP64
17	0	t		fs	0		0	**mfc1 t,fs**	
17	1	t		fs	0		0	**dmfc1 t,fs**	MIPS64
17	2	t		cs	0		0	**cfc1 t,cs**	

首先考虑 cp0 寄存器。需要读取的 cp0 寄存器共有 8 个："cp0_status","lo"(entrylo),"hi"(entryhi),"cp0_badvaddr","cp0_cause","cp0_epc","cp0_debug","cp0_depc"，依次储存于 core 的 cp0_regs[8][17]中。由于协处理器号并不直接就是其储存的顺序号，因此要进行序号转换。

读取时，首先根据读取协处理器寄存器编号确定所读取的寄存器，获取其对应的真正 cp 编号（协处理器内部编号），同时确定其寄存器位数。随后，根据 cp 编号和位数，确定是使用 mfc0 指令还是 dmfc0 指令，再配合 cp 编号，完成指令。以读取 cp0_regs[0]为例：

(1) 由协处理器寄存器编号为 0，得知是 cp0_status 寄存器；

(2) 设置寄存器位数为 32，寄存器内部编号为 12；

(3) 以 12 为参数，使用 mfc0 指令；

(4) 最后，读取该通用寄存器即可。

对于 cp1 寄存器，情况有些不同，因为 cp1 并不总是有效的。若在 cp1 无效的时候尝试读取 cp1 寄存器，会导致调试中断。因此在读取 cp1 寄存器前，需要判断 cp1 的有效性。cp0_status 的第 29 位就是用来指示 cp1 的使能状况的。因此，在将 cp1 寄存器移入通用寄存器前，需要检查 cp0_status 寄存器。如果无效，则读取出来的值为“0xffffffffffffffff”。

Cp1 寄存器共 33 个，以 cp1_regs[33][17]的形式储存在 core 中。其中，前 32 个

寄存器依次为"f0"~"f31"，均为 64 位，最后一个是控制寄存器的"fcr31"（现称 fcsr），32 位。其处理方法和 cp0 类似，不再赘述。

最终，可以得到 read_cp_reg 函数：

```
//读取协处理器寄存器，读取的内容更新全局变量中对应值
/*
 * cp_num  :   协处理器序号，可为 0 或 1。即 cp0/cp1。
 * reg_num :   协处理器寄存器序号，超出则报错。
 * select  :   选择执行 CPU。
 */
void read_cp_reg(int cp_num, int reg_num, int select);
```

其处理流程如图 4-10 所示：

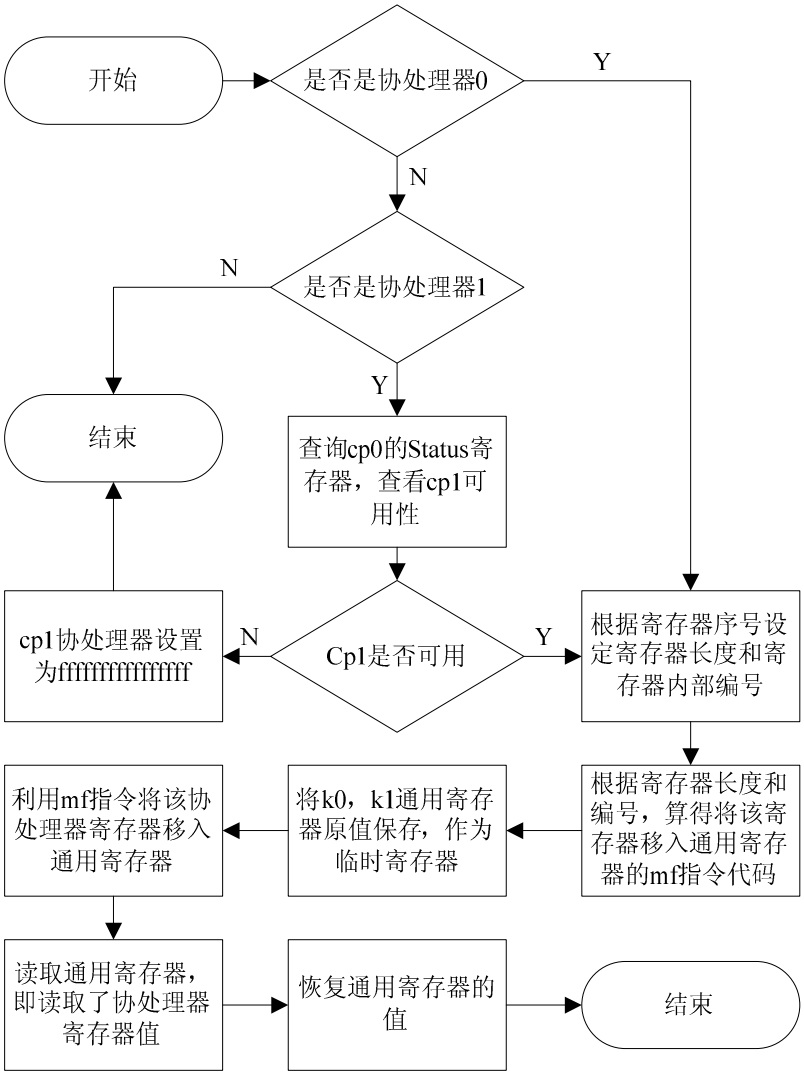


图 4-10 读取特定协处理器寄存器

4.2.6 写入特定协处理器寄存器

和读取特定协处理器寄存器思路类似，要写入协处理器寄存器，可以先将待写入的值写入通用寄存器，再将通用寄存器中的值传入协处理器寄存器。相对于mfc0/mfc1/dmfc0/dmfc1/cfc1，写入时相应的使用 mtc0/mtc1/dmtc0/dmtc1/ctc1。

表 4-4 写协处理器寄存器指令

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
16	4	t		cd	0		0	**mtc0 t,cd**	
16	5	t		cd	0		0	**dmtc0 t,cs**	MIP64
17	0	t		fs	0		0	**mfc1 t,fs**	
17	1	t		fs	0		0	**dmfc1 t,fs**	MIPS64
17	2	t		cs	0		0	**cfc1 t,cs**	

同样，对 cp1 的操作需要先检查其有效性，只在 cp1 使能的前提下才能对其进行写操作，否则不向 cp1 寄存器写值。不过由于对 cp1 寄存器的写操作往往通过写全部 cp1 寄存器来调用，故而把检查有效性的操作放在写所有协处理器寄存器函数里能够大大提升效率。下面以 cp1_regs[3]写值为例叙述函数运作过程：

(1) 寄存器序号不是 33，则设置长度为 64，序号为 3；

(2) 设置好 dmtc1 语句；

(3) 将要写入的值放入 k0 通用寄存器（任意寄存器皆可）；

(4) 调用配置好的 dmtc1 语句，将 k0 寄存器写入 cp1_regs[3]；

(5) 还原临时通用寄存器。

如此，可得函数 write_cp_reg：

```
//写入协处理器寄存器
/*
 * cp_num  :   协处理器序号，可为 0 或 1。即 cp0/cp1。
 * reg_num :   协处理器寄存器序号，超出则报错。
 * data    :   写入寄存器的值，16 进制形式，要求长度为 16
 * select  :   本质为一个 0~15 的整数，写成二进制为四位数字。
 *             四位分别说明了四核处理器是否为 select 状态，是则为 1。
 *             比如 1010 说明一、三核为 select 状态，0000 则都不在 select 状态。
 */
void write_cp_reg(int cp_num, int reg_num, const char* data, int
select);
```

其处理流程如图 4-11 所示：

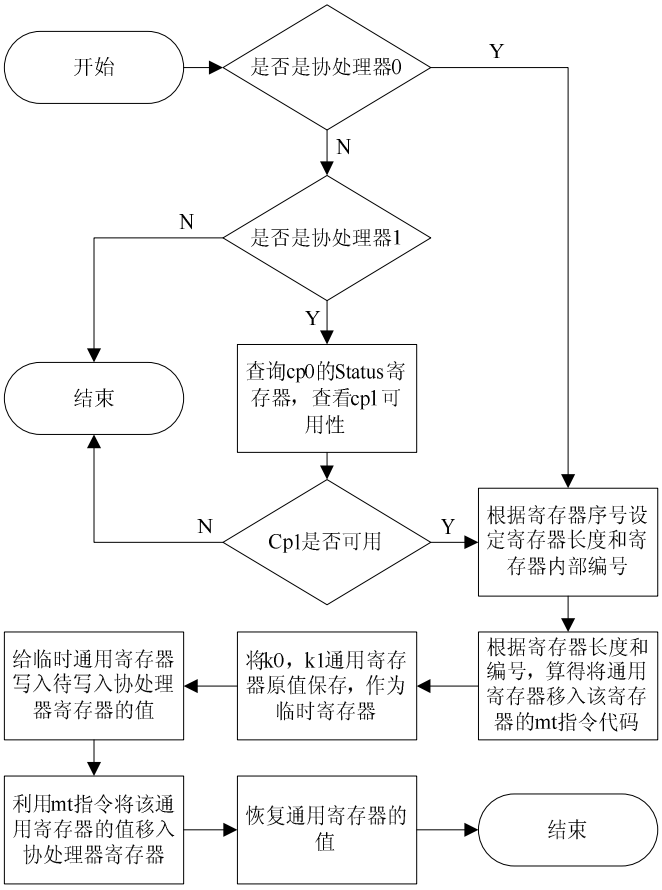


图 4-11 写入特定协处理器寄存器

4.2.7 读取所有协处理器寄存器

和读取所有通用寄存器情况类似，读取所有协处理器寄存器也是将一些读取各个寄存器所共用的操作减少执行次数，而将不能共用的操作遍历而成。此处需要注意下，读取所有通用协处理器寄存器函数并不真正意义上的读取“所有协处理器寄存器”，而只是同时读取部分要求的协处理器寄存器。这些要求了的寄存器包括 cp0 能读取的前 6 个寄存器，和 cp1 能够读取的全部 33 个寄存器。

以读取所有 cp1 寄存器为例，过程如下：

(1) 检查 cp0_status 的 cp1 使能位。不可访问则返回；

(2) 保存 k0/k1 寄存器作为临时变量，并设置 k0 为 dmseg 地址；

(3) 遍历 33 个 cp1 寄存器，分别设置其 dmfc1/cfc1；

(4) 通过 dmfc1/cfc1 将 cp1 寄存器读入 k1；

(5) 再读取 k1；

(6) 遍历完毕，恢复 k0 和 k1 的值。

对于 cp0 也采取类似的方法。可以得到 read_all_cp_regs：

```
//读取所有协处理器寄存器，所读数据放入全局变量 core 中
/*
 * cp_num  :   协处理器序号，可为 0 或 1。即 cp0/cp1。
 * select  :   本质为一个 0~15 的整数，写成二进制为四位数字。
 *             四位分别说明了四核处理器是否为 select 状态，是则为 1。
 *             比如 1010 说明一、三核为 select 状态，0000 则都不在 select 状态。
 */
void read_all_cp_regs(int cp_num, int select);
```

其处理流程如图 4-12 所示：

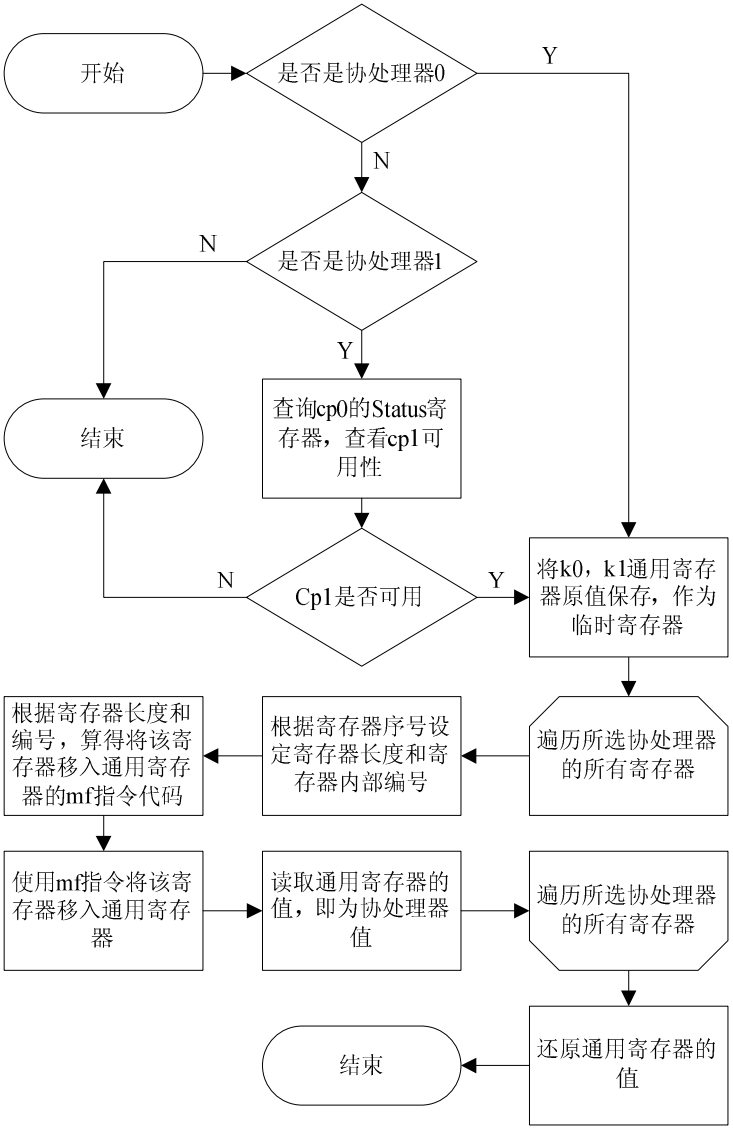


图 4-12 读取所有协处理器寄存器

4.2.8 写入所有协处理器寄存器

和读取所有协处理器寄存器类似，写入所有协处理器寄存器只操作 cp0 能读取的前 6 个寄存器，和 cp1 能够读取的全部 33 个寄存器。

和写入所有通用寄存器类似，写入所有协处理器寄存器也是遍历调用写入单个协处理器寄存器的函数；略有不同的是，遍历前，写入协处理器寄存器函数需要检查 cp1 的有效性。即检查 cp0_status 的第 29 位。若无效，则不写入该协处理器寄存器。

其处理流程如图 4-13 所示：

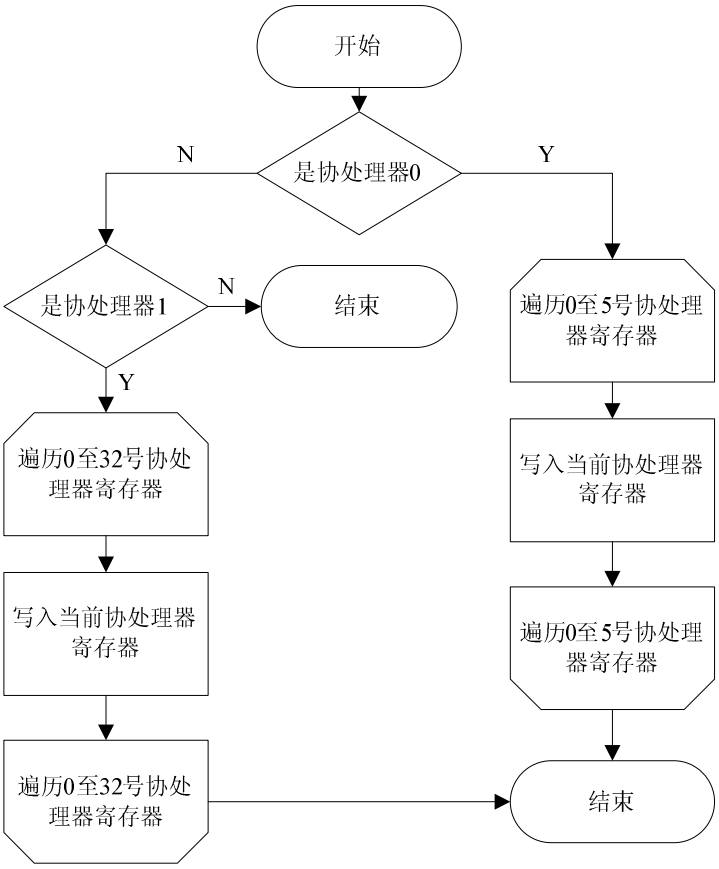


图 4-13 写入所有协处理器寄存器

如此，可得 write_all_cp_regs：

```
//写入所有协处理器寄存器
/*
 * cp_num  :   协处理器序号，可为 0 或 1。即 cp0/cp1。
 * data    :   一个(char (*)[17])的数组，储存要写入的数据。
```

```
 * select  :   本质为一个 0~15 的整数，写成二进制为四位数字。
 *             四位分别说明了四核处理器是否为 select 状态，是则为 1。
 *             比如 1010 说明一、三核为 select 状态，0000 则都不在 select 状态。
 */
void write_all_cp_regs(int cp_num, const char** data, int select);
```

4.2.9 读取指定地址的内存

要读取指定地址的内存，可以将其拆分为两个步骤：1）将指定地址指定长度的内存，分解为地址对齐的 8 字节小块，并在读取后合并返回；2）读取已经对齐地址的 8 字节小块。

之所以这样进行处理，是因为要对内存进行操作的 sd/sw/sh/sb 等指令，分别要求操作的内存 8 字节/4 字节/2 字节/1 字节对齐，而对于大段内存，采用 8 字节对齐的方式能够获得最高的效率。因此，才将内存分解为 8 字节对齐的小块。

这两个步骤，前者主要是对给入的地址和长度进行处理，而后者才是真正的对内存操作。实现上，也将这两部分为两个函数来实现，后者为 read_8byte_mem，前者为 read_mem，对分成对齐 8 字节的内存调用 read_8byte_mem。根据由下至上的逻辑顺序，先实现 read_8byte_mem，再来看 read_mem。

读取 8 字节内存的原理并不复杂。简单的说，就是将该地址内存读取到指定的通用寄存器，再将通用寄存器的值取出来。将指定地址内存读取到指定的通用寄存器，可以使用 ld 指令，一次载入 8 字节。

以读取 ffffffffa2345678 的 8 字节内存为例：

```
send_cpu_ins("3c1aff20", select); //   lui k0,0xff20
send_cpu_ins("375a7000", select); //   ori k0,k0,0x7000
send_cpu_ins("3c1bffff", select); //   lui k1,0xffff
send_cpu_ins("377bffff", select); //   ori k1,k1,0xffff
send_cpu_ins("001bdc38", select); //   dsll k1,k1,16
send_cpu_ins("377ba234", select); //   ori k1,k1,0xa234
send_cpu_ins("001bdc38", select); //   dsll k1,k1,16
send_cpu_ins("377b5678", select); //   ori k1,k1,0x5678
send_cpu_ins("df7c0000", select); //   ld gp,0(k1)
send_cpu_ins("ff5c0000", select); //   sd gp,0(k0)
```

再调用 things_after_sd 即可读取该处 8 字节内存。

其处理流程如图 4-14 所示：

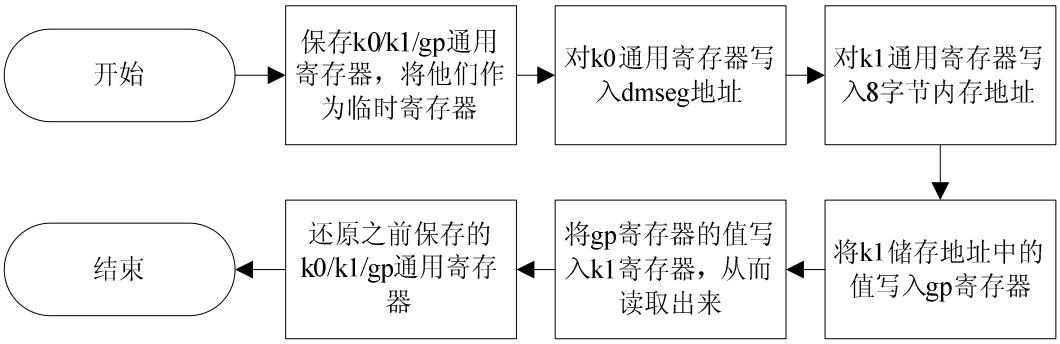


图 4-14 读取指定地址的 8 字节内存

而 read_mem 则是将内存分割为对齐的 8 字节小块，再将 8 字节小块的读取结果拼接起来返回给用户。这里值得注意的是内存不对齐的部分，比如从 ffffffffa2340003 读取到 ffffffffa23400013，按照 8 字节对齐，分为 0x03～0x07（块一），0x08～0x0f（块二），0x10～0x13（块三）。块二肯定是使用一次 ld/sd 来进行操作，没有问题。而块一读取五个字节，块三读取四个字节，是否应该使用 lw/sw/lh/sh/lb/sb 等来进行操作呢？如果是平时 CPU 直接执行机器指令，那么将类似读 5 个字节的操作分成读 4 个加读 1 个，或许比直接读 8 个来的快。然而现在的每一条机器指令都是通过 EJTAG 的处理器访问操作执行的，指令的实际花费时间，其主要部分并不在指令执行本身，而在于 EJTAG 的操作。因此，指令的条数尽可能少，才是提高效率的正确方法。

因此，无论某对齐的 8 字节块需要读取几个字节，都将那个 8 字节块完整的使用一次 ld/sd 将其读出。比如上面的地址，块一直接读取 0x00～0x07，块三直接读取 0x10～0x17。最终读取出来的内存，再根据实际地址来进行截取，连接，得到需要读取的内存值。由此可得函数 read_mem：

```
//读取内存
/*
 * paddr   :   要读取内存的地址。
 * pmem    :   要个读取内存数据分配的空间。
 * length  :   读取内存的长度，单位为字节。
 * select  :   选择执行 CPU。
 * return  :   返回 pmem。
 */
char* read_mem(const char* paddr, char* pmem, int length, int select);
```

其处理流程如图 4-15 所示：

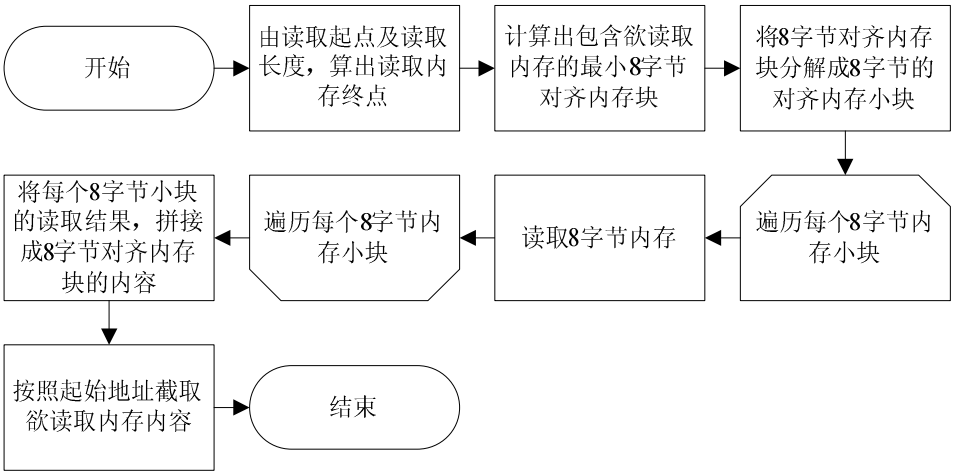


图 4-15 读取指定地址的内存

4.2.10 写入指定地址的内存

同样是对内存进行操作，因此，还是会遇到内存地址对齐的问题。同读取指定地址内存类似，写入指定地址内存，也将其分为两个步骤：1）将指定地址指定长度的内存，分解为地址对齐的 8 字节小块，分别处理；2）写入已经对齐地址的 8 字节小块。前者为 write_mem，后者为 write_8byte_mem。

和读取时原理相似，无论是只有 1 个字节还是有 7 个字节，对于对齐的 8 字节小块都采取同时写入 8 字节的方式进行处理。

同样还是先考虑 write_8byte_mem。对于将指定 8 字节数据写入特定对齐内存，其实就是 sd 指令的运用：先将要写入的数据放入通用寄存器 k1，再将要写入地址放入通用寄存器 k0，再使用 sd 指令将 k1 数据存入 k0 地址。注意，这里虽然调用了 sd 指令，但是并不需要调用 things_after_sd 函数。该函数仅用于向 dmseg 写内存之后，从 EJTAG 的 DATA 寄存器取数据。以将“1234567890abcdef”写入地址“ffffffffa2345678”为例：

```
send_cpu_ins("3c1affff", select); //   lui k0,0xffff
send_cpu_ins("375affff", select); //   ori k0,k0,0xffff
send_cpu_ins("001ad438", select); //   dsll k0,k0,16
send_cpu_ins("375aa234", select); //   ori k0,k0,0xa234
send_cpu_ins("001ad438", select); //   dsll k0,k0,16
send_cpu_ins("375a5678", select); //   ori k0,k0,0x5678

send_cpu_ins("3c1b1234", select); //   lui k1,0x1234
```

```
send_cpu_ins("377b5678", select); //   ori k1,k1,0x5678
send_cpu_ins("001bdc38", select); //   dsll  k1,k1,16
send_cpu_ins("377b90ab", select); //   ori k1,k1,0x90ab
send_cpu_ins("001bdc38", select); //   dsll  k1,k1,16
send_cpu_ins("377bcdef", select); //   ori k1,k1,0xcdef

send_cpu_ins("ff5b0000", select); //   sd k1,0(k0)
```

其处理流程如图 4-16 所示：

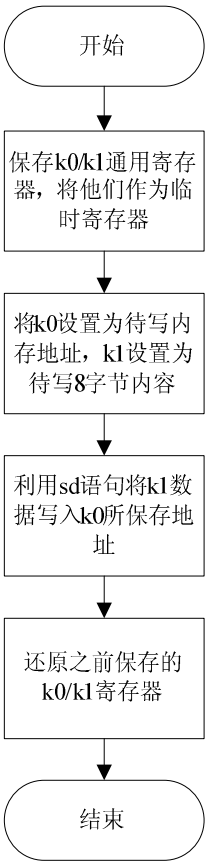


图 4-16 写入指定地址的 8 字节内存

而 write_mem 相较 read_mem 要稍微复杂些。由于头块和尾块并不全是要写入的字节，因此在写入前要先读取该 8 字节块。将读出的 8 字节内存修改部分字节内容，形成新的 8 字节，再写回头尾块内存。

可以得到 write_mem 函数：

```
//写入内存
/*
 * paddr   :   要写入内存的地址。
 * pdata   :   要写入的数据。这里的数据并不一定以'\0'结束。
 * length  :   写入数据长度，单位为字节。
 * select  :   选择执行 CPU。
 */
```

```
void write_mem(const char* paddr,const char* pdata,int length,int
select);
```

其处理流程如图 4-17 所示：

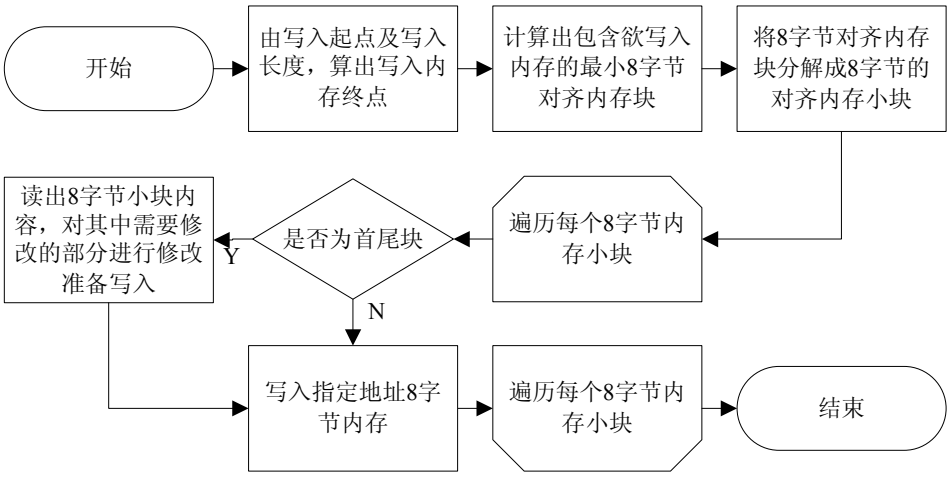


图 4-17 写入指定地址的内存

4.2.11 给指定地址设置断点

给指定地址设置断点，其本质就是让内核在执行到该地址代码的时候自动进入调试模式。为此，EJTAG 给 MIPS 的 CPU 指令集加入了两个额外的指令：SDBBP 和 DERET。

表 4-5 SDBBP 指令

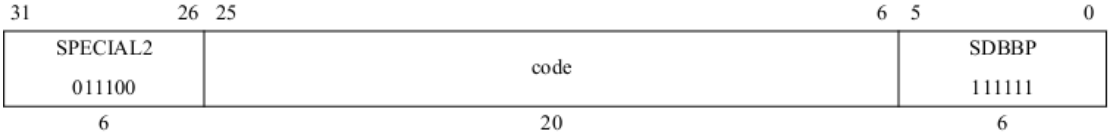
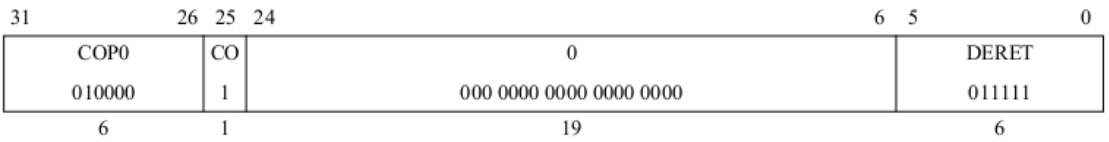


表 4-6 DERET 指令



前者让 CPU 在执行时进入调试模式，后者则可以让 CPU 退出调试模式，继续运行。那么，如果可以将指定地址的原有指令换成 SDBBP 指令，那么当 CPU 执行到该处时，则会进入调试模式，相当于在该地址设置了断点。因此，核心问题就变成了替换指定地址的一条 CPU 指令。

一条 MIPS 的 CPU 指令 32 位，4 字节。已有函数可以一次读取对齐的 8 字节内存，或是写入 8 字节内存。因此，只要找到包含该指令地址的 8 字节对齐块，将其取出改写四字节指令后写回，即可完成替换指令的操作。可得函数 replace_ins：

```
//替换内存指定处一条指令
/*
 * paddr   :   要设置指令的地址。
 * pNew    :   该地址替换指令。四个字节。
 * pOld    :   该地址原有指令。四个字节。可为 NULL，即不读取原有指令。
 * select  :   选择执行 CPU。
 */
void replace_ins(const char* paddr, const char* pNew, char* pOld, int
select);
```

其处理流程如图 4-18 所示：

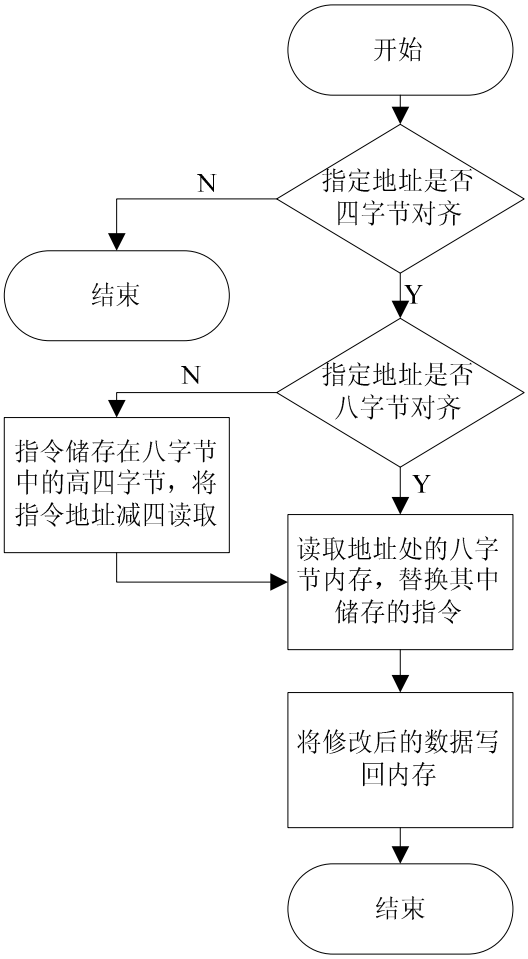


图 4-18 给指定地址替换一条指令

另外，可封装一个函数，直接调用 replace_ins 将代码替换为 SDBBP，从而可以减少一个参数的传递。得到函数 set_breakpoint：

```
//给指定地址设置断点
/*
 * paddr   :   要设置断点的地址。
 * pins    :   该地址原本执行指令。四个字节。可为 NULL，即不读取原有指令。
 * select  :   选择执行 CPU。
 */
void set_breakpoint(const char* paddr, char* pins, int select);
```

其处理流程如图 4-19 所示：

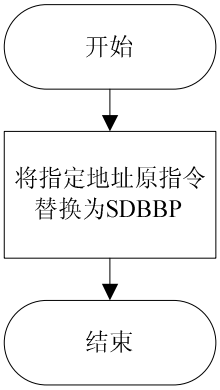


图 4-19 给指定地址设置断点

4.2.12 跳出断点继续运行

由上节得知，要退出断点（调试模式），需要 CPU 执行 DERET 指令。那么，使用处理器访问发送 DERET 指令，即可使系统跳出断点继续运行，即：

```
//退出调试模式
send_cpu_ins("01000010000000000000000000011111",select);
```

4.3 本章小结

本章以发送一条 CPU 指令为界，分别描述了 EJTAG 指令级调试的底层设计及高级功能实现。EJTAG 的底层设计，是指从计算机并口编程开始，实现 EJTAG 调试模式状态下的发送特定 CPU 指令；EJTAG 的高级功能，主要是指的通过 EJTAG 手段来读写内存及寄存器。通过对 EJTAG 功能进行这样的划分，可以在实现高级功能时忽略 TAP 状态机等底层细节，提高程序设计效率。

5 源码级调试

上一章介绍了如何使用 EJTAG 对龙芯 3A 计算机进行指令级调试。然而在更多情况下，用户更关心的，或许是如何在函数的特定行设置断点，又如何查看某个变量的内容。这些需求，其实可以归类于源码调试。也就是说，需要有一个软件来将源码级调试转换为指令级调试。在我们的方案中，这个软件就是 epdb。

5.1 Server 端与 epdb 的通信

由 EJTAG 调试模块结构（3.2）可知，server 端直接使用指令级调试函数来对龙芯机进行控制，同时也需要和 epdb 之间使用 Socket 方式进行通信。其通信包符合 GDB 远程串行协议。

一个典型的通信包片段如表 5-1 所示（报文由 server 端打印，send 和 recv 相对于 server 而言）：

表 5-1 典型的通信包片段

recv:	g
send:	$000000000000000092934200000000000000edc0000000000000edc00000 00008076115c0100009800b0917a010000988070115c010000980000edc0f fffffff00000000000000008a76115c010000983000000000000000482c41 0000000000000000000000000008000080fffffffff4f94180ffffffff58a cf27f000000000000edc0ffffffff8076115c010000989893420000000000 0000edc0ffffffff00b0917a010000980000000000000000889342 0000000 000000000000000000010000000000000000b055f92a0000000030aef27f00 000000e0ff8e7a0100009800c08e7a01000098c0fd8e7a010000980000420 000000000c8fb4180ffffffffe1c80010000000001f11fa0500000000fa00 8000000000c0e8028000000000c020880050000000000b48f3f80fffffffff fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff fffffffffffffffffffffffffffffffffffffff#66
recv:	z0,803f8fb4,1

续表 5-1

send:	$OK#9a
recv:	m980000015c117680,8
send:	$6264732f7665642f#ae
recv:	m980000015c117688,8
send:	$0000000000000031#04

上段代码演示了 epdb 查看寄存器、取消断点以及读取内存的动作。server 端收到 epdb 端发来的报文后，相应的调用指令级调试中已经完成的函数，将得到的结果返回给 epdb 端，再由 epdb 端解析收到的报文，将结果以正确形式返回给用户。

5.2 Server 端设计

EJTAG 调试模块结构（3.2）中的图 3-3 展示了 Server 端的模块结构。其中的 EJTAG 模块即为上文的指令级调试部分。下面来看看通信模块和断点模块的设计。

5.2.1 通信模块设计

同其他任何一个 Server 类似，通信模块的主函数会新建一个 socket 连接，在做完必要的初始化后，便不断的进行收包，随后对收到的包进行处理。

```
while (cmd = get_packet())
{
    handle_packet(cmd);
}
```

此处 cmd 自 get_packet()函数得到的是已经整理好的命令字符串，相当于 $packet-data#checksum 中的 packet-data，具体信息可参见表 2-2。为了能够自动解析传进来的命令，可以新建一个结构和一个对字符串进行“运行时识别”的标准结构数组。结构如下：

```
typedef struct tag_CMD
{
    char cmdname;
    void (*cmd_func)(char*);
} CMD;
```

这结构包含了一个字符的命令名，以及一个函数指针。如果能够让命令名关联

上函数指针，也就可以在收到特定的命令的时候自动让其执行特定的函数了。为了达到这个目的，还需要一个“指令数组”来储存命令名和执行函数之间的关系。一个符合要求的数组看上去像是这样：

```
CMD basic_cmd[] = { { 'm', readMemory }, { 'X', writeMemory }, { 'G',
writeRegister32 }, { 'g', readRegister32 }, { 's',singleStep }, { 'q',
queryInfo }, { 'H', defaulHandler }, { '?', signalInfo }, { 'v',
vcontHandler }, { 'z',inactiveBP }, { 'Z', setBP }, { 'c',
vcont_continue }, { 'k', quitServer }, { 'Q', Qhandler } };
```

handle_packet 于是可以这样的简单实现：

```
void handle_packet(char* cmd)
{
    int i = 0;
    while (i < sizeof(basic_cmd) / sizeof(CMD))
    {
        if (cmd[0] == basic_cmd[i].cmdname)
        {
            basic_cmd[i].cmd_func(cmd + 1);
            return;
        }
        i++;
    }
    ignore_cmd();
}
```

以 cmd 接收到读内存命令 maddr,length 为例。在 handle_packet 的遍历中，发现 cmd[0]和 basic_cmd[0]（即{ 'm', readMemory }）匹配。那么将 cmd+1（即字符串“addr,length”）作为参数传入 readMemory 函数中。剩下的，就是一个个去实现具体操作的函数了。

basic_cmd 中涉及到的各函数，其实质基本上还是调用的是 EJTAG 模块中直接对 EJTAG 接口进行操作的函数。下面仅以 readMemory 为例，来看看这类函数的实现。

```
long readMemory(char* parg)
{
    //do ejtag operations then send packet
    char paddr[17] = { 0 };
    long memlen = 0;
    int length1 = 0, length2 = 0;
    while (*(parg + length1) != ',')
        length1++;
    int i = 0;
    for (; i < 16 - length1; i++)
        paddr[i] = 'f';
    for (i = 0; i < length1; i++)
        paddr[i + 16 - length1] = parg[i];
    length2 = strlen(parg + 1 + length1);
```

```
    if (-1 == chartolong2(&memlen, parg + length1 + 1, length2))
    {
        printf("read memory wrong \n");
        return -1;
    }
    //invoke ejtag function
    printf("paddr is %s\n", paddr);
    char* pascii = (char*) malloc(memlen);
    pascii = read_mem(paddr, pascii, memlen, 15);

    ascii2hex(pascii, message_out, memlen);
    free(pascii);
    put_packet(message_out);
    return 0;
}
```

由于需要通过函数指针调用，这些函数的传入参数类型都保持一致，即字符指针。函数的实现上，主要还是将传入的参数转换成合适的类型来调用 EJTAG 操作函数，之后再将得到的值以报文的形式发送给客户端。

5.2.2 断点模块设计

比起另外两个模块，断点管理模块的工作相对简单。断点管理模块其实可以看作是通信模块和 EJTAG 模块关于断点问题的一个中间层。通信模块可以直接调用 EJTAG 端提供的大多数底层接口，然而断点方面，服务器端调用断点模块提供的函数，而断点模块则可以调用底层断点操作。

之所以这么设计，主要是因为底层的断点处理所能提供的服务和上层的断点需求之间有着较大的差异。对服务器端而言，可能会同时操作多个断点，而每个断点都可能有各种状态。另外，由于断点的底层实现是将其内存中的原有指令替换成断点指令，因此对于每个断点，还需要将其原有指令保存以便之后恢复。这一块功能相对独立，因此，可将其作为一个单独的模块，不仅便于管理，逻辑上也更加清晰。

Server 端的断点模块维护一个断点数组，来保存所有设置断点的信息，包括了断点的地址，断点是否活跃。当 Server 端收到报文“Z0,addr,type”时，服务程序会将该断点保存在断点数组中，然后调用 EJTAG 操作函数进行设置断点操作。设置断点的步骤具体如下：

(1) 读取 addr 地址处的内存值；

(2) 将该处的原来的指令保存起来；

(3) 进行 EJTAG 写操作，将 addr 的指令替换成 sdbbp 调试陷阱指令。

当 Server 端收到断点的取消报文“z0,addr,type”，则是通过以下几个步骤来将设置过的断点取消：

(1) 将保存在断点数组中的断点取消；

(2) 调用 EJTAG 操作函数取消断点；

(3) 对 addr 处进行写操作，恢复保存的 addr 处的原来的指令。

5.3 epdb 端的设计

epdb 端并非 EJTAG 调试组所设计，因此我并不清楚其中各处细节。然而为了论文的逻辑完整性，在此还是简要的对 epdb 进行一些基本的描述。下面将首先对 epdb 进行简要的介绍，随后将以打印字符串变量为例简单的描述 epdb 的工作原理。

5.3.1 epdb 简要介绍

epdb 是一个参考 gdb 来实现的命令行式的调试软件。除了上文中提到的对 GDB 远程串行协议的兼容，epdb 的使用方式及使用习惯也和 gdb 相类似[32] 。epdb 的代码结构比较复杂，其源码文件如表 5-2 所示。

表 5-2 epdb 源码文件

xmltok_ns.c	ansidecl.h	archures.c	arch-utils.c
arch-utils.h	array.c	ascii.h	asciitab.h
block.c	block.h	blockframe.c	breakpoint.c
breakpoint.h	callback.c	charset.c	cli-cmds.c
cli-cmds.h	cli-decode.h	cli-interp.c	cli-out.c
cli-out.h	cli-utils.c	cli-utils.h	command.h
defs.h	dwarf_api.c	dwarf_api.h	epdb_assert.h
epdb_curses.h	epdb_string.h	epdbarch.c	epdbarch.h
epdb-host.c	epdbloop.c	epdbthread.h	epdbtypes.h
event-loop.c	event-loop.h	event-top.c	event-top.h
exceptions.c	exceptions.h	expat.h	expat_external.h
findvar.c	frame.c	frame.h	frame-base.h

续表 5-2

frame-unwind.c	frame-unwind.h	GDB.c	iasciitab.h
infcmd.c	inferior.c	inferior.h	inf-loop.c
inf-loop.h	infrun.c	inline-frame.c	inline-frame.h
inputcmd.c	internal.h	interps.c	interps.h
latin1tab.h	linespec.c	linespec.h	loop.c
main.c	main.h	memattr.c	memattr.h
mem-break.c	mips-linux-tdep.c	mips-tdep.c	mips-tdep.h
moz_extensions.c	nametab.h	obstack.c	obstack.h
osabi.c	osabi.h	printcmd.c	printcmd.h
progspace.c	progspace.h	record.h	regcache.c
regcache.h	reggroups.c	reggroups.h	remote.c
remote.h	sentinel-frame.c	sentinel-frame.h	ser-base.c
ser-base.h	serial.c	serial.h	signals.c
signals.h	solib.c	solib-svr4.c	solib-svr4.h
source.c	source.h	stack.c	stack_1.c
stack_1.h	symtab.c	symtab.h	target.c
target.h	target-descriptions.c	target-descriptions.h	terminal.c
thread.c	threadid.c	threadofmy.c	threads2.c
threads.c	top.c	top.h	tui.c
tui.h	tui-data.c	tui-data.h	tui-file.c
tui-interp.c	tui-io.c	tui-io.h	tui-out.c
tui-win.c	ui-file.c	ui-file.h	ui-out.c
ui-out.h	user-regs.c	utf8tab.h	utils.c
valprint.c	valprint.h	value.c	value.h
vartest.c	vec.c	vec.h	xmlparse.c
xmlrole.c	xmlrole.h	xml-support.c	xml-support.h
xml-tdesc.c	xml-tdesc.h	xmltok.c	xmltok.h
xmltok_impl.c	xmltok_impl.h		

共计 158 个文件。大量的代码在没有文档的帮助下难免令人不知所措。为了寻找突破口，下面先来看下一段典型的 epdb 运行指令及结果，或许能够从中得到启发。需要在命令行中输入的语句将下划线表示。

```
epdb vmlinux-mips
Reading symbols from /home/cozy/workspace/epdb_local/epdb/vmlinux-
mips...done.
(epdb)target
[New inferior 42000]
(epdb)b fs/namespace.c:2014
src is fs/namespace.c
Breakpoint 1 at 0x803f8fb4: file namespace.c, line 2014.
(epdb)c
Continuing.
[New Thread 2]
[Switching to Thread 2]
thread 2 hit Breakpoint 1 at 0x803f8fb4: file namespace.c, line 2014.
in print_pc_source
pc is 803f8fb4
in_CU_die 1in_CU_die 2in src linecount is 1606 find lineno is 2014
```

```
"namespace.c":2014   if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
(epdb)si
0x803f8fb8 in print_pc_source
pc is 803f8fb8
in_CU_die 1in_CU_die 2in src linecount is 1606 find lineno is 2014
"namespace.c":2014   if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
(epdb)s
in src linecount is 1606 in src linecount is 1606 in print_pc_source
pc is 803f8fbc
in_CU_die 1in_CU_die 2in src linecount is 1606 find lineno is 2015
"namespace.c":2015       flags &= ~MS_MGC_MSK;
(epdb)p flags
$1 = 18446744072651341824
(epdb)p dev_name
$2 = 0x980000017bc80280 "/dev/sdb1"
(epdb)info-r
zero       0x0000000000000000
at         0x0000000000429392
v0         0x00000000c0ed0000
v1         0x00000000c0ed0000
a0         0x980000017bc80280
a1         0x980000017fc46000
a2         0x980000017bc81400
a3         0xffffffffc0ed0000
t0         0x0000000000000000
t1         0x980000017bc8028a
t2         0x0000000000000030
t3         0x0000000000412c48
t4         0x0000000000000000
t5         0xffffffff80000008
t6         0xffffffff8041f9f4
t7         0x000000007f9a64e8
s0         0xffffffffc0ed0000
s1         0x980000017bc80280
s2         0x0000000000000000
s3         0x980000017bc81400
s4         0x980000017fc46000
s5         0x0000000000000000
s6         0x0000000000429388
s7         0x0000000000000000
t8         0x0000000000000010
t9         0x000000002ba955b0
k0         0x000000007fef8080
k1         0x980000017ed33fe0
gp         0x980000017be34000
sp         0x980000017be37dc0
s8         0x0000000000420000
ra         0xffffffff8041fbc8
status     0x000000001000f8e1
lo         0x0000000005d0aa1e
hi         0x00000000000000a3
badvaddr   0x000000002ac09a50
cause      0x0000000050008020
pc         0xffffffff803f8fbc
f0         0xffffffffffffffff
```

```
f1          0xffffffffffffffff
f2          0xffffffffffffffff
f3          0xffffffffffffffff
f4          0xffffffffffffffff
f5          0xffffffffffffffff
f6          0xffffffffffffffff
f7          0xffffffffffffffff
f8          0xffffffffffffffff
f9          0xffffffffffffffff
f10         0xffffffffffffffff
f11         0xffffffffffffffff
f12         0xffffffffffffffff
f13         0xffffffffffffffff
f14         0xffffffffffffffff
f15         0xffffffffffffffff
f16         0xffffffffffffffff
f17         0xffffffffffffffff
f18         0xffffffffffffffff
f19         0xffffffffffffffff
f20         0xffffffffffffffff
f21         0xffffffffffffffff
f22         0xffffffffffffffff
f23         0xffffffffffffffff
f24         0xffffffffffffffff
f25         0xffffffffffffffff
f26         0xffffffffffffffff
f27         0xffffffffffffffff
f28         0xffffffffffffffff
f29         0xffffffffffffffff
f30         0xffffffffffffffff
f31         0xffffffffffffffff
fcsr        0xffffffffffffffff
fir         0x0000000000000000
restart     0x0000000000000000
(epdb)l 2014
in src linecount is 1606 file: /home/pan/loongson/linux-loongson-
all/fs/namespace.c
2009        struct path path;
2010        int retval = 0;
2011        int mnt_flags = 0;
2012
2013        /* Discard magic */
2014        if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
2015            flags &= ~MS_MGC_MSK;
2016
2017        /* Basic sanity checks */
2018
2019        if (!dir_name || !*dir_name || !memchr(dir_name, 0,
PAGE_SIZE))
(epdb)info-break
Num     Type           Disp   Enb    Address        What
1       breakpoint     keep   y      0x803f8fb4     namespace.c:2014
(epdb)c
Continuing.
```

上述示例是使用 epdb 通过 ejtag 来调试内核代码的常用操作，它展示了使用

epdb 选择调试程序、连接 server 端、设置断点、进行指令级单步、源码级单步、查看寄存器、打印整形和字符型变量、查看指定行代码、查看断点信息以及继续执行等功能。

在第一次运行 c 指令后，目标机恢复运行状态。此时，需在目标机端输入

```
mount /dev/sdb1 /media/usb
```

指令，方可触发 fs/namespace.c 文件中 2014 行处设下的断点。该行位于 do_mount 函数，而该函数正是在输入 mount 指令时所被调用[33] 。

从上述运行结果可以看出，epdb 在选定调试程序后（上文中为 vmlinux-mips，即 Linux 内核代码），调试器就进入了一个循环等待用户命令的状态。此时会有“(epdb)”来提醒用户输入下一条命令。因此，很可能 epdb 会采取这么一种工作机制：等待用户输入指令，随后根据不同的指令将其分配到不同的函数进行处理，有些类似于 server 端对于不同包的处理。

事实的确如此。在 epdb 中输入的每一条指令，都会辗转进入 loop.c 的 execute_command 函数，在这里，程序对接收到的不同指令进行分配，随后根据指令，转入各自不同的文件来进行处理。截取 execute_command 函数的部分代码如下：

```
else if((0==strcmp(t_tempcmd,"print"))||(0==strcmp(t_tempcmd,"p")))
    {
        print_command(t_arg,1);
    }
else if((0==strcmp(t_tempcmd,"x"))||(0==strcmp(t_tempcmd,"examine")))
    {
        x_command(t_arg,1);
    }
else if((0==strcmp(t_tempcmd,"info-break"))||
(0==strcmp(t_tempcmd,"info-b"))||
(0==strcmp(t_tempcmd,"info-breakpoints")))
    {
        if(t_arg[0] == 0)
          breakpoints_info(NULL,1);
        else
          breakpoints_info(t_arg,1);
    }
```

不同指令被转交给不同的函数，比如 print 指令转入了 Printcmd.c 中的 print_command 函数，info-b 指令转入了 Breakpoint.c 中的 breakpoints_info 函数。下一节，以“p dev_name”为例，一步步来看看 epdb 是怎样从头至尾处理一条指令。

5.3.2 print 指令的处理

在 loop.c 文件中，“p dev_name”指令被识别为打印指令，于是被辗转交给了 Printcmd.c 中的 print_command_1 函数，“dev_name”也作为参数传进来。

print_command_1 函数调用了两个关键函数，其一是 variable_query 函数，主要负责查询变量所处的物理存储位置及其变量类型；另一个则是 print_formatted 函数，主要负责将该变量获取并打印出来。

variable_query 函数位于 Dwarf_api.c 文件中。这是一个看上去有些奇怪的文件名，实际上，Dwarf 却几乎是当今最为流行的调试信息格式。关于 Dwarf 以及更加详细的符号表知识，并不在本文的讨论范围之内，感兴趣的读者可参见[34] [35] [36] 。这里只需明白，对于加了调试参数而编译出来的内核文件，除了包含程序执行的指令，还包含有用以调试的信息。这些信息以 Dwarf 格式存储着各变量函数等的实际类型和存储位置。

然而，即便 Dwarf 已经是较为成熟的调试信息格式，但手动解析 Dwarf 格式也并不是一件容易的事。业界比较常用的 Dwarf 解析库主要有两个[37] ：

(1) BFD（libbfd），GNU binutils 就是使用的它，包括 Linux 下大名鼎鼎的工具 objdump，ld（GNU 链接器），以及 as（GNU 汇编器）；

(2) libdwarf —— 同它的老大哥 libelf 一样，为 Solaris 以及 FreeBSD 系统上的工具服务。

epdb 选择了 libdwarf，或许是因为 libdwarf 的版权更加自由（LGPL，BFD 是 GPL）。与 Dwarf 类似，本文也并未给 libdwarf 库准备太多篇幅，更多的信息可参见其官方文档[38] [39] 。

在 libdwarf 库的帮助下，variable_query 首先通过当前 pc 值得到当前函数的调试信息项[36] ，再根据变量名找到变量的调试信息项。随后，再在变量调试信息项中找到其 DW_AT_location 属性，从而获得其位置属性，进一步计算得到其存储的物理地址（或是寄存器）。类似的，可以找到变量的类型。

知晓存储地址和变量类型，便可以去获取具体数据并返回给用户了。这一过程

主要由 print_formatted 函数来完成。再看看“p dev_name”指令。在 variable_query 执行过后，已经明确知道 dev_name 存储在内存中，地址为 0x980000015c117680（见 5.1 节通信包），变量类型为字符串。于是，print_formatted 向 server 端发送包来要求读内存数据。

由于并不清楚字符串长度，每次要求 server 读取 8 字节内存（主要是考虑到效率），直到遇到“\0”。另外字符串的打印需要考虑 Big Endian 和 Little Endian 转换的问题[40] 。最终，终端里得到了预期的结果：

```
(epdb)p dev_name
$1 = 0x980000015c117680 "/dev/sdb1"
```

5.4 本章小结

在指令级调试的基础上，本章实现了 EJTAG 的源码级调试。源码级调试能够在相应源码处设置断点、打印变量、单步运行，像调试应用程序一般来调试 Linux 内核。

6 总结与展望

本文先介绍了国内外内核调试的研究现状，比较了 EJTAG 调试和其他内核调试方案的优缺点。随后文章介绍了使用 EJTAG 方式进行内核硬件调试的相关知识理论，为后文的系统实现打下铺垫。接下来，文章对整个调试系统的模块结构进行了简要的说明，并阐述了 EJTAG 硬件调试在其中所处的作用和地位。随后的两章分别实现了 EJTAG 的指令级调试和源码级调试，成功构建了 EJTAG 内核调试系统，这也是全文和课题的重点。

然而在实践中，发现还有以下几点存在改进空间，值得进一步的探究。

(1) 当前所进行的内核调试，是在 epdb 端向 server 端发送 target 指令时开始，调试的起点也就是执行 target 指令的时刻。然而，用户往往希望能够在内核载入的起始处便进入调试。因此，可以研究下如何使系统在载入内核的同时进入调试模式。

(2) 由于编译内核时的优化，导致了内核带有的符号表信息不完整，部分变量在特定行时可能不能通过符号表找到，从而无法打印。如何编译内核而不丢失符号表信息，也是需要考虑的一个问题。

(3) 虽然在指令级调试阶段，能够对特定内核进行完全控制而不影响其它内核，然而在源码级调试时，由于操作系统参与了内核的任务分配，对特定内核进行调试会让操作系统的行为混乱。要如何在多核环境下进行源码级调试，将 EJTAG 单独控制 CPU 的能力发挥出来，也是一个需要研究的课题。

(4) 当前，使用 EJTAG 进行调试的速度较慢，用户进行操作时能够感受到相当明显的延迟。如何提高 EJTAG 调试的性能也是值得关注的话题。

致 谢

经过两年研究生阶段的学习和研究，在论文完成之际，我特向指导和帮助过我的老师、同学、朋友及关心支持我的家人，致以最衷心的感谢。

首先要感谢我的导师李国徽教授。本论文是在李老师的精心指导下完成的，从论文的选题、设计方案直至完成论文的整个过程，我都得到了李老师耐心细致的指导。李老师严谨的治学态度、渊博的学识、一丝不苟的工作作风、热情待人的品质，给我留下了极其深刻的印象。另外要感谢朱虹教授、袁凌副教授。在论文初稿完成后，两位老师给出了大量有价值的建议，帮助我修正了论文的不少问题。

接下来我要感谢我的父母、室友、同学和朋友们。他们在我的整个研究生生涯中，一直陪伴着我，给了我生活和学习上的帮助。尤其要感谢的是同为 EJTAG 调试小组成员的潘源斌和姜俊，项目中的每一些微小的进步，同样凝聚着他们的汗水。

最后，再次对关心、帮助我的老师和同学表示衷心地感谢。

参考文献

[1] 李磊.主流 CPU 的历史发展与趋势分析.中国西部科技,2004,(18):6-7,11.DOI:10.3969/j.issn.1671-6396.2004.18.003

[2] 刘品阳.一种多处理器异构系统设计与实现.计算机技术与发展,2011,21(5):179-182.DOI:10.3969/j.issn.1673-629X.2011.05.047

[3] 孙蔷,姜靖. 从拿来主义到自主创“芯”. 科技日报: 2009

[4] 张银奎. 软件调试：Software Debugging : 电子工业出版社. 2008. 3

[5] 李红卫. 嵌入式 Linux 远程调试工具 gdbstub 的剖析与改进. 哈尔滨工业大学, 2004

[6] Wind River. 通过 JTAG 芯片级调试加速嵌入式 Linux 设备的开发

[7] M. Tim Jones. System emulation with QEMU--The machine within the machine, IBM developerWorks Linux Technical library

[8] 李树雷, 陈渝. Linux 系统内核的调试. IBM developerWorks 中国 Linux 文档库

[9] 科银京成. Lambda 技术白皮书

[10] 中国科学院计算技术研究所. 龙芯 3A 处理器用户手册之多核处理器架构、寄存器描述与系统软件编程指南

[11] 何颂颂,顾乃杰,朱海涛等. 面向龙芯 3A 体系结构的 BLAS 库优化. 小型微型计算机系统. 2012

[12] 中国科学院计算技术研究所. 龙芯 3A 处理器用户手册之 GS464 处理器核

[13] 朱建培.基于龙芯一号 IP 核的 EJTAG 调试.单片机与嵌入式系统应用, 2008

[14] 金辉,华斯亮,张铁军等. 基于 JTAG 标准的处理器片上调试的分析和实现.微电子学与计算机,2007,24(6):116-119,122.DOI:10.3969/j.issn.1000-7180.2007. 06.034

[15] MIPS Technologies. EJTAG Specification

[16] 葛宏. 基于 EJTAG 的交叉调试器开发. 清华大学. 2003

[17] Steve Holzner. Eclipse Cookbook, June 2004

[18] 许建荣,姚国良,胡晨. 并口 JTAG 仿真器的设计与实现. 电子器件,2007

[19] IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std 1149.1-2001

[20] OPEN-JTAG 开发小组. ARM JTAG 调试原理

[21] 刘旭伟. FPGA 可配置端口电路的设计, 西安电子科技大学, 2008

[22] 乔崇. mips ejtag 原理和实现, June 2012

[23] 韩青. 多核调试新方法探讨.电子产品世界,2007,(12):114-115.DOI: 10. 3969 /j.issn.1005-5517.2007.12.019

[24] 盛建忠,王胜,张庆文. GDB RSP 协议与 USB 通信在嵌入式调试系统中的应用. 电子与封装:2013

[25] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with GDB 10th

[26] 吴疆, 田金兰与张素琴. 面向多目标机的交叉调试器的研究与设计. 清华大学学报(自然科学版), 2003. 43(1): 第 101-104 页

[27] Tim Waugh. The Linux 2.4 Parallel Port Subsystem

[28] 占维,杨根头.基于 Linux 的并口驱动程序.电脑编程技巧与维护,2006,(10):72-76.DOI:10.3969/j.issn.1006-4052.2006.10.017

[29] 蔡蓓. 基于 GNU、并行口和 EJTAG 的 MIPS 调试系统, 复旦大学, 2005

[30] Robert Britton. MIPS Assembly Language Programming

[31] D. Sweetman, See MIPS run Linux (2nd Edition)

[32] 王聪. 学习使用 GNU GDB Debugger

[33] 毛德操, 胡希明. Linux 内核源代码情景分析, 浙江大学出版社, 2001

[34] UNIX International Programming Languages Special Interest Group, DWARF Debugging Information Format

[35] 李宝丹. 嵌入式系统辅助调试环境的开发.北京邮电大学, 2009

[36] Michael J. Eager, Eager Consulting, Introduction to the DWARF Debugging FormatApril, 2012

[37] Eli Bendersky . How debuggers work: Part 3 - Debugging information

[38] David Anderson. A Consumer Library Interface to DWARF

[39] UNIX International Programming Languages Special Interest Group. A Producer Library Interface to DWARF

[40] Kyle Aubrey, Ashan Kabir. Data Movement Between Big-Endian and Little-Endian Devices