异步图书 www.epubit.com

RISC-V

RISC-V®

体系结构编程与实践

奔跑吧Linux社区◎编著

香山处理器团队　龙蜥社区RISC-V SIG　进迭时空◎审校

基于64位RISC-V处理器，系统介绍RISC-V体系结构，总结常见陷阱与项目经验

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

RISC-V®

体系结构编程与实践

奔跑吧Linux社区◎编著

香山处理器团队　龙蜥社区RISC-V SIG　进迭时空◎审校

人民邮电出版社

北京

图书在版编目（CIP）数据

RISC-V体系结构编程与实践 / 奔跑吧Linux社区编著
. -- 北京 : 人民邮电出版社, 2023.3(2023.3重印)
ISBN 978-7-115-60360-9

Ⅰ. ①R… Ⅱ. ①奔… Ⅲ. ①微处理器－系统设计－
教材 Ⅳ. ①TP332.021

中国版本图书馆CIP数据核字(2022)第222752号

内 容 提 要

本书旨在介绍 RISC-V 体系结构的设计和实现。本书首先介绍 RISC-V 体系结构的基础知识、实验环境搭建、常用指令、函数调用规范与栈，然后讲述 GNU 汇编器、链接器、链接脚本和 GCC 内嵌汇编代码，接着讨论 RISC-V 体系结构中的异常处理、中断、内存管理、高速缓存、缓存一致性、TLB 管理、原子操作、内存屏障指令，最后阐述 RSIC-V 体系结构中的压缩指令扩展、虚拟化扩展等。

本书不仅适合软件开发人员阅读，还可以作为计算机相关专业和相关培训机构的教材。

◆ 编　　著　奔跑吧 Linux 社区
责任编辑　谢晓芳
责任印制　王　郁　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　https://www.ptpress.com.cn
固安县铭成印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：29　　2023 年 3 月第 1 版
字数：745 千字　　2023 年 3 月河北第 2 次印刷

定价：129.80 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东市监广登字 20170147 号

本书约定

为了帮助读者更好地阅读本书以及完成本书的实验，我们对本书的术语、实验环境做了一些约定。

1. RISC-V 与 RV64 术语

目前 RISC-V 体系结构主要包括 RV32 以及 RV64，即 32 位体系结构以及 64 位体系结构。本书重点介绍与 RV64 相关的内容。除在寄存器位宽、地址映射模式等方面有区别之外，RV32 和 RV64 还有其他方面的细微差异，不过本书不会详细介绍它们之间的区别。要了解关于 RV32 的内容，请读者自行阅读 RISC-V 官方手册。

在 RISC-V 体系结构手册中使用哈特（Hart）来描述一个处理器硬件线程，这个术语类似于 x86 体系结构定义的超线程（Simultaneous Multi-Threading，SMT）以及 ARMv8 体系结构定义的处理机（Processing Element，PE）。不过截至 2022 年还没有商用的 RISC-V 处理器实现超线程技术，因此，本书对哈特和中央处理器（Central Processing Unit，CPU）两个概念不做严格区分，继续沿用 CPU 这个通用的概念来描述一个处理器执行单元或者硬件线程（如果硬件实现了 SMT）。

2. 实现案例

本书基于 Linux 内核以及小型操作系统（BenOS）进行讲解。Linux 内核采用 Linux 5.15。本书大部分实验以 BenOS 为基础，使读者可从最简单的裸机程序不断进行扩展，最终完成一个具有内存管理、进程调度、系统调用等基本功能的小型操作系统，从而学习和掌握 RV64 体系结构的相关知识。我们在实验的设计过程中参考了 Linux 内核等开源代码的实现，在此对开源社区表示感谢。

3. 实验环境

本书推荐的实验环境如下。

- 主机硬件平台：Intel x86-64 处理器兼容主机。
- 主机操作系统：Ubuntu Linux 20.04[①]。
- GCC 版本：9（riscv64-linux-gnu-gcc）。
- QEMU 版本：4.2.1。
- GDB 版本：gdb-multiarch。

读者在安装完 Ubuntu Linux 20.04 系统后，通过如下命令安装本书需要的软件包。

```
$ sudo apt update -y
$ sudo apt install net-tools libncurses5-dev libssl-dev build-essential openssl qemu-
  system-misc libncurses5-dev gcc-riscv64-linux-gnu git bison flex bc vim universal-
  ctags cscope gdb-multiarch libsdl2-dev libreadline-dev
```

我们基于 VMware/VirtualBox 镜像搭建了全套开发环境，读者可以通过“奔跑吧 Linux 社区”微信公众号获取下载地址。使用本书配套的 VMware/VirtualBox 镜像可以减少配置开发环境带来的麻烦。

---

① 推荐读者使用我们提供的 VMware/VirtualBox 镜像。

4. 实验平台

读者无须额外购买开发板，推荐在如下两个免费的模拟器实验平台上完成本书所有实验。

- QEMU。
- 香山模拟器 NEMU[1]。

5. 关于实验参考代码和配套资料下载

为了节省篇幅，本书中大部分实验只列出了实验目的和实验要求，希望读者能独立完成实验。

在 GitHub 网站搜索“runninglinuxkernel/riscv_ programming_ practice”即可找到本书中大部分实验的参考代码。

本书有如下配套资料。

- 大部分实验的参考代码。
- VMware/VirtualBox 镜像。
- 免费视频。

读者可以通过关注微信公众号“奔跑吧 Linux 社区”，输入“risc-v”获取下载地址。

6. 芯片资料

作者在编写本书的过程中参考了 RISC-V 官方的技术资料以及与 GNU 工具链相关的文档。下面是涉及的技术手册，这些技术手册都是公开发布的，读者可以在 RISC-V 基金会官网、SiFive 官网以及 GNU 官网上下载。

- “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 20191213”：RISC-V 指令集手册。
- “The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20211203”：RISC-V 体系结构手册。
- SiFive FU740-C000 Manual, v1p6：SiFive 公司的 FU740 处理器内核技术手册。
- Using the GNU Compiler Collection, v9.3：GCC 官方手册。
- Using as, the GNU Assembler, v2.34：汇编器（AS）官方手册。
- Using ld, the GNU Linker, v2.34：链接器（LD）官方手册。
- RISC-V “V” Vector Extension, Version 1.0：RVV 手册。
- RISC-V Base Cache Management Operation ISA Extensions, Version 1.0：RISC-V 高速缓存维护指令扩展手册。
- RISC-V Platform-Level Interrupt Controller Specification, Version 1.0：PLIC 手册。
- RISC-V ABIs Specification, Version 0.01：RISC-V ABI 接口手册。
- RISC-V Supervisor Binary Interface Specification, Version 1.0.0：RISC-V SBI 手册。
- SiFive TileLink Specification, Version 1.8.1：TileLink 总线手册。
- 香山官方文档：香山处理器手册。

7. 汇编代码大小写

RISC-V 指令、寄存器的书写方式约定如下。

- 关于 RISC-V 指令，在汇编代码中用小写，在正文中用大写。

---

① 有部分实验在 NEMU 上还没实现，不过可以在 QEMU 上完成。

- 关于通用寄存器和系统寄存器，在汇编代码和正文中都用小写。若使用大写形式书写通用寄存器和系统寄存器，汇编器会报错。

8. 汇编代码说明

在 RISC-V 汇编代码中，本书有如下约定。

- (x*n*)：直接寻址模式，表示以 x*n* 寄存器的值为基地址进行寻址，在本书中简称为 **x*n* 地址**，示例如下。

```
ld x2, (x1)   //从 x1 地址中加载 8 字节数据到 x2 寄存器中
sd x2, (x1)   //把 x2 寄存器的值存储到 x1 地址中
```

- offset(x*n*)：偏移寻址模式，表示以 x*n* 寄存器的值为基地址，然后偏移 offset 字节进行寻址。

```
ld x2, 8(x1)   //从 x1+8 地址中加载 8 字节数据到 x2 寄存器中
sd x2, 8(x1)   //把 x2 寄存器的值存储到 x1+8 地址中
```

推荐序

非常荣幸应笨叔的邀请为本书写序。这也是我第一次为书写序，我感到非常荣幸。

我认识笨叔好多年，他与我是广东老乡，他身上带着广东人特有的韧劲、谦虚与阳光。在2021年第一届RISC-V中国峰会上听他说准备写一本关于RISC-V的小册子，没想到一年时间，一本厚厚的《RISC-V体系结构编程与实践》就放在我面前。我们知道写一本书需要付出太多，他身上这种韧劲与乐于分享的精神很让我佩服。他常常说自己是草根（“佛山蓝翔”常挂在他嘴边），并且是国内某小企业的一线工程师。然而，他一直活跃在开源社区，分享知识和经验，从出书到视频课程，他都受到了广大读者的喜爱。

出于共同的爱好和热情，我们在计算机底层系统软件领域的很多观点非常相似。这次写序时才知道他有一个很酷的英文名字——Figo，也许他想像著名的欧洲足球天才路易斯·菲戈（Luis Figo）一样在自己擅长的领域不停地奔跑。我一直有一个强烈的感觉——总有一天我们会有合作的机会，但是没有想到机会这么快就来了。

2022年是RISC-V的元年，我们的产品线都全部转到RISC-V上，我们也盼了好久才等到这个时刻，我一直都和笨叔说RISC-V一定可以和ARM、x86三足鼎立，因为它的开放性和国际性，以及它在雏形阶段非常多的机会。这让我回想起20世纪90年代在英国读本科时Linux系统面对各大UNIX系统的竞争的场景。本书的内容与巧妙的编排方式深深吸引了我，因此我非常愿意尽一点绵力，帮笨叔审阅书稿，让本书变得更加完美。另外，我们把本书作为ROMA系列产品（全球首款RISC-V笔记本计算机）附送的图书之一，让本书在RISC-V社区贡献一份非常重要的力量。

如作者之前的“奔跑吧Linux内核”系列作品一样，本书也有自己的独特之处，特别是RISC-V体系结构方面的讲解，这里就不详细介绍了。建议读者带着疑问去学习本书，特别是对比ARM体系结构的实现，相信本书将使读者受益匪浅。

祝笨叔和“奔跑吧Linux内核”系列图书在开源软件与RISC-V开源硬件的蓝天上飞得更高，在地上奔跑得更快、更远！

“奔跑吧，学习者和跟随者！”希望读者在RISC-V领域有更大的收获。

梁宇宁

鉴释科技创始人

RISC-V 体系结构自测题

在阅读本书之前，请读者尝试完成以下自测题，从而了解自己对 RISC-V 体系结构的掌握程度（以下假定处理器体系结构为 RV64，编译器为 GCC）。一共有 20 道题，每道题 5 分，总分 100 分。

1．RISC-V 指令集支持 64 位宽的数据和地址寻址，为什么指令的编码宽度只有 32 位？为什么不使用 64 位指令编码？请说明使用 64 位指令编码的优缺点。

2．下面的指令中，假设当前 PC 值为 0x8020 0000，那么 a5 和 a6 寄存器的值是多少？

```
auipc   a5,0x2
lui     a6, 0x2
```

3．请解析下面这条指令的含义。

```
csrrw tp, sscratch, tp
```

4．假设函数调用关系为 main()→func1()→func2()，请画出 RISC-V 体系结构下函数栈的布局。

5．下面是 my_entry 宏的定义。

```
.macro my_entry, rv, label
       j    rv\()\rv\()_\label
 .endm
```

下面的语句调用 my_entry 宏，请解释该宏是如何展开的。

```
my_entry   1, irq
```

6．关于链接器，请解释链接地址、虚拟地址以及加载地址。当一个程序的代码段的链接地址与加载地址不一致时，我们应该怎么做才能让程序正确运行？

7．什么是加载重定位和链接重定位？RISC-V 为什么要做松弛链接优化？RISC-V 是如何做的？

8．在 RISC-V 处理器中，异常发生后 CPU 自动做了哪些事情？操作系统需要做哪些事情？在异常返回时，CPU 是返回发生异常的指令还是下一条指令？什么是异常现场？对于 RISC-V 处理器来说，应该保存异常现场中的哪些内容？异常现场保存到什么地方？

9．为什么页表要设计成多级页表？直接使用一级页表是否可行？多级页表又引入了什么问题？请简述 Sv39 页表映射模式下的映射过程。

10．在使能 MMU 时，为什么需要建立恒等映射？

11．请简述直接映射、全相联映射以及组相联映射的高速缓存的区别。什么是高速缓存的重名问题？什么是高速缓存的同名问题？VIPT 类型的高速缓存会产生重名问题吗？

12．假设系统中有 4 个 CPU，每个 CPU 都有各自的一级高速缓存，处理器内部实现的是 MESI 协议，它们都想访问相同地址的数据 *a*，大小为 64 字节，这 4 个 CPU 的高速缓存在初始状态下都没有缓存数据 *a*。在 *T*0 时刻，CPU0 访问数据 *a*。在 *T*1 时刻，CPU1 访问数据 *a*。在 *T*2 时刻，CPU2 访问数据 *a*。在 *T*3 时刻，CPU3 想更新数据 *a* 的内容。请依次说明 *T*0～*T*3 时刻 4 个 CPU 中高速缓存行的变化情况。

13．DMA 缓冲区和高速缓存容易产生缓存一致性问题。从 DMA 缓冲区向设备的 FIFO 缓

冲区搬运数据时，应该如何保证缓存一致性？从设备的 FIFO 缓冲区向 DMA 缓冲区搬运数据时，应该如何保证缓存一致性？

14. 为什么操作系统在切换（或修改）页表项时需要先刷新对应的 TLB 项后切换页表项？

15. 在 RISC-V 体系结构中，刷新所有处理器的 TLB 是如何实现的？

16. 假设内存模型为 RVWMO，在下面的执行序列中，CPU0 先执行 a=1 和 b=1，接着 CPU1 一直循环判断 b 是否等于 1，如果等于 1，则跳出 while 循环，最后执行“assert (a == 1)”语句，判断 a 是否等于 1，那么 assert 语句有可能会出错吗？如果会出错，请指出如何修复，并分析其原因。

```
CPU0                          CPU1
-------------------------------------------------------------
void func0()                 void func1()
{                            {
     a = 1;                     while (b == 0) continue;
     b = 1;                     assert (a == 1)
}                             }
```

17. 如果多个核同时使用 LR 与 SC 指令对同一个内存地址进行访问，如何保证数据的一致性？为什么原子内存操作指令比独占内存访问指令要高效很多？

18. 什么是进程上下文切换？对于 RISC-V 处理器来说，进程上下文切换需要保存哪些内容？保存到哪里？新创建的进程第一次执行时，第一条指令在哪里？

19. 如果调度器通过 switch_to()把进程 A 切换到进程 B，那么进程 B 是否在切换完成之后，马上执行进程 B 的回调函数？在下面的代码片段中，printf()函数能输出正确的 data 值吗？为什么？

```
void schedule(void)
{
  long data;

  //读取当前进程的私有数据
  data = xxx;

  ...

  //切换到进程 B
  switch_to(A, B);

  //输出 data
  printf(data);
}
```

20. 请简述在虚拟化场景中两阶段地址映射的过程。在 RISC-V 虚拟化扩展中，VMM 如何进入虚拟机？VM 有哪些途径可以陷入 VMM 中？

以上题目的答案都分布在本书的各章中。

前　言

站在 2023 年来看处理器的发展，x86-64 与 ARM64 体系结构依然是目前市场上的主流处理器体系结构。不过一个可喜的变化是，RISC-V 受到学术界和工业界越来越多的关注，RISC-V 有可能成为第三大体系结构。最近几年，国内出现了一批优秀的芯片公司和科研机构，它们基于 RISC-V 体系结构来打造国产芯片，包括 MCU（Micro-Controller Unit，微控制单元）芯片、高性能处理器芯片。开源高性能处理器的优秀代表有香山处理器等。

RISC-V 体系结构以开放性、先进性、简洁性以及开源性等众多优点，得到了国内外高校的青睐。越来越多的高校采用 RISC-V 体系结构作为蓝本进行讲授“计算机系统基础”“计算机组成原理”“操作系统”“嵌入式系统”等计算机专业的核心课程。国内高校涌现一批基于 RISC-V 体系结构的创新性和开拓性教学项目，如中国科学院大学的“一生一芯”项目。

RISC-V 是一个崭新的体系结构，软硬件生态也在逐步完善。为了帮助读者快速和深入了解 RISC-V 体系结构，我们以工程师的视角，结合大学课程特点、企业新员工培训需求以及实际工程项目经验，精心设计了几十个有趣的实验，读者可以通过实验逐步深入学习和理解 RISC-V 体系结构与编程。

本书特色

本书有如下一些特色。

- 突出动手实践。学习一门新技术，动手实践是非常有效的方法。本书基于 QEMU 以及香山模拟器 NEMU 介绍了几十个有趣的实验。我们可以采用搭积木的方法，从编写第一行代码开始，通过一个个实验逐步深入学习 RISC-V 体系结构，最终可以编写一个能在 QEMU 或者 NEMU 上运行的简易小型操作系统，它具有存储管理单元（Memory Management Unit，MMU）以及进程调度等功能。
- 以问题为导向。问题导向式的学习方法有利于提高学习效率。本书在大部分章前面列举了一些思考题，用于激发读者探索未知知识的兴趣。这些思考题也是各大公司的高频面试题，相信仔细解答这些问题对读者的面试大有裨益。
- 基于 64 位处理器讲述 RISC-V 体系结构。本书不仅介绍开源的 64 位香山处理器的微架构实现，还介绍 RV64 指令集、GCC、链接器、链接脚本、异常处理、中断处理、内存管理、TLB 管理、内存屏障指令、可伸缩矢量计算与优化以及虚拟化扩展等方面的知识。本书把 RISC-V 体系结构中难理解的部分通过通俗易懂的语言呈现给读者，并通过有趣的案例分析使读者加深理解。
- 总结常见陷阱与项目经验。本书总结了众多一线工程师在实际项目中遇到的陷阱，如使用指令集时的陷阱等，这些宝贵的项目经验会对读者有所帮助。

本书主要内容

本书主要介绍 RISC-V 体系结构的相关内容。本书重点介绍 RV64 指令集、GNU 汇编器、

链接器、链接脚本、内存管理、高速缓存、可伸缩矢量计算与优化以及虚拟化扩展等。

本书一共有 20 章，包含如下内容。

第 1 章主要介绍 RISC-V 体系结构基础知识。

第 2 章介绍如何使用 QEMU 以及香山模拟器 NEMU 搭建实验环境。

第 3 章讨论 RV64 指令集中常见指令以及常见陷阱等内容。

第 4 章介绍函数调用规范、RISC-V 栈的布局等内容。

第 5 章介绍 GNU 汇编器的汇编语法、常用的伪指令、RISC-V 依赖特性等内容。

第 6 章介绍链接器、链接脚本、加载重定位、链接重定位与链接器松弛优化等内容。

第 7 章介绍 GCC 内嵌汇编代码基本用法以及常见陷阱等内容。

第 8 章介绍 RISC-V 体系结构中异常处理的触发与返回、异常向量表、异常现场等内容。

第 9 章介绍 RISC-V 体系结构里中断处理的基本概念和流程，包括 CLINT、平台级别的中断控制器、保存和恢复中断现场的方法等内容。

第 10 章介绍 RISC-V 体系结构中的内存管理，包括页表、页表项属性、页表遍历过程、内存属性以及恒等映射等内容。

第 11 章介绍高速缓存的基础知识，包括高速缓存的工作原理、映射方式、虚拟高速缓存、物理高速缓存等相关内容。

第 12 章介绍缓存一致性，包括缓存一致性的分类、MESI 协议、高速缓存伪共享等内容。

第 13 章介绍 TLB 管理，包括 TLB 基础知识、ASID、TLB 管理指令等内容。

第 14 章介绍原子操作，包括原子操作基本概念、独占内存访问工作原理、原子内存访问操作指令等内容。

第 15 章介绍内存屏障指令，包括内存屏障指令产生的原因、RISC-V 约束条件以及 RISC-V 中的内存屏障指令等内容。

第 16 章介绍如何合理使用内存屏障指令等内容。

第 17 章介绍与操作系统相关的内容，包括 64 位编程下常见的 C 语言陷阱、简易进程调度器、系统调用等内容。

第 18 章不仅介绍 RVV 指令，还分析如何使用 RVV 指令进行矢量优化。

第 19 章介绍 RISC-V 中的压缩指令扩展。

第 20 章介绍 RISC-V 中的虚拟化扩展，包括 CPU 虚拟化、内存虚拟化、中断虚拟化等内容。

在编写本书的过程中，作者得到了中国科学院计算技术研究所的包云岗老师以及香山处理器团队成员的大力支持，他们为本书提供了香山处理器微架构的实现细节以及香山模拟器 NEMU，让读者有机会深入了解一款开源高性能处理器的实现细节。余子濠博士为在香山模拟器 NEMU 上完成本书的实验做了大量适配和优化工作，让本书的实验变得更加丰富和有趣。另外，香山处理器团队的成员还帮忙审阅了本书的书稿，并给出了宝贵的意见和建议。在此，衷心感谢包云岗老师以及余子濠、王凯帆、张梓悦。

作者在编写本书的过程中还得到鉴释科技有限公司的梁宇宁和进迭时空的陈志坚的鼎力支持，梁老师一直鼓励着我们，审阅了本书并给出了许多宝贵的意见和建议。另外，深度数智科技有限公司的工程师孙彦邦和张猛，鉴释科技有限公司的工程师赖建新、李隆以及肖林杰也帮忙审阅了部分书稿，在此表示感谢。

本书在编写过程中也得到了龙蜥社区运营委员会主席陈绪与技术委员会主席杨勇的大力支持和帮助。龙蜥社区 RISC-V SIG（Special Interest Group，特别兴趣小组）成员也帮忙审阅了本书，并给出了宝贵的反馈和改进意见。他们是平头哥的熊健、郭任、尚云海和阿里云操作系统

团队的陈健康、田瑞冬、王江波、王宝林、宋卓。

本书由笨叔策划和主编，奔跑吧 Linux 社区中一群热爱开源的热心工程师也参与了本书的编写和审阅工作中，他们是张毅峰、胡茂留、牛立群、胡进、何花、黄山、郑律。另外，还有一些热心工程师帮忙审阅了部分书稿，他们是贾献华、钱为、孙少策、曹文辉、钟居哲、陆亚涵、王乐、杨明辉、杨勇、张诺方、李朋、潘建亨、方盛洲、朱信冉、张志鹏等。

由于作者知识水平有限，书中难免存在不妥之处，敬请各位读者批评指正。欢迎通过电子邮箱 runninglinuxkernel@126.com 与作者交流。要下载本书配套的实验环境、实验参考代码，请关注“奔跑吧 Linux 社区”微信公众号并输入“risc-v”。

笨叔

服务与支持

本书由异步社区出品，社区（https://www.epubit.com）为您提供后续服务。

提交勘误信息

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，单击“发表勘误”，输入相关信息，单击“提交勘误”按钮即可，如下图所示。本书的作者和编辑会对您提交的信息进行审核，确认并接受后，您将获赠异步社区的 100 积分。积分可用于在异步社区兑换优惠券、样书或奖品。

与我们联系

我们的联系邮箱是 contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区投稿（直接访问 www.epubit.com/contribute 即可）。

如果您所在的学校、培训机构或企业想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接通过邮件发送给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下 IT 专业图书社区，致力于出版精品 IT 图书和相关学习

产品，为作译者提供优质出版服务。异步社区创办于 2015 年 8 月，提供大量精品 IT 图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网 https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品 IT 专业图书的品牌，依托于人民邮电出版社几十年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的 Logo。异步图书的出版领域包括软件开发、大数据、人工智能、测试、前端、网络技术等。

异步社区　　微信服务号

目　录

第 1 章　RISC-V 体系结构基础知识 ……1
1.1　RISC-V 介绍 ……1
1.1.1　RISC-V 指令集优点 ……1
1.1.2　RISC-V 指令集扩展 ……2
1.1.3　RISC-V 商业化发展 ……2
1.2　RISC-V 体系结构介绍 ……3
1.2.1　RISC-V 体系结构 ……3
1.2.2　采用 RISC-V 体系结构的常见处理器 ……3
1.2.3　RISC-V 体系结构中的基本概念 ……4
1.2.4　SBI 服务 ……5
1.3　RISC-V 寄存器 ……6
1.3.1　通用寄存器 ……6
1.3.2　系统寄存器 ……7
1.3.3　U 模式下的系统寄存器 ……8
1.3.4　S 模式下的系统寄存器 ……9
1.3.5　M 模式下的系统寄存器 ……11
1.4　香山处理器介绍 ……15
1.4.1　香山处理器体系结构 ……15
1.4.2　香山处理器的前端子系统 ……16
1.4.3　香山处理器的后端子系统 ……18
1.4.4　香山处理器的访存子系统 ……20
1.4.5　香山处理器的 L2/L3 高速缓存 ……25
第 2 章　搭建 RISC-V 实验环境 ……29
2.1　实验平台 ……29
2.1.1　QEMU ……29
2.1.2　NEMU ……30
2.2　搭建实验环境 ……31
2.2.1　实验 2-1：输出“Welcome RISC-V!” ……31
2.2.2　实验 2-2：单步调试 BenOS 和 MySBI ……32
2.3　BenOS 和 MySBI 基础实验代码解析 ……34
2.3.1　MySBI 基础代码分析 ……34
2.3.2　BenOS 基础代码分析 ……37
2.3.3　合并 BenOS 和 MySBI ……41
2.4　QEMU + RISC-V + Linux 实验平台 ……41
第 3 章　基础指令集 ……44
3.1　RISC-V 指令集介绍 ……44
3.2　RISC-V 指令编码格式 ……45
3.3　加载与存储指令 ……46
3.4　PC 相对寻址 ……49
3.5　移位操作 ……53
3.6　位操作指令 ……55
3.7　算术指令 ……56
3.8　比较指令 ……57
3.9　无条件跳转指令 ……58
3.10　条件跳转指令 ……59
3.11　CSR 指令 ……61
3.12　寻址范围 ……62
3.13　陷阱：为什么 ret 之后就进入死循环 ……62
3.14　实验 ……64
3.14.1　实验 3-1：熟悉加载指令 ……64
3.14.2　实验 3-2：PC 相对地址寻址 ……64
3.14.3　实验 3-3：memcpy()函数的实现 ……65
3.14.4　实验 3-4：memset()函数的实现 ……65
3.14.5　实验 3-5：条件跳转指令 1 ……65
3.14.6　实验 3-6：条件跳转指令 2 ……66
3.14.7　实验 3-7：子函数跳转 ……66
3.14.8　实验 3-8：在汇编中实现串口输出功能 ……66
第 4 章　函数调用规范与栈 ……67
4.1　函数调用规范 ……67

4.2 入栈与出栈 …… 70
4.3 RISC-V 栈的布局 …… 72
4.3.1 不使用 FP 的栈布局 …… 72
4.3.2 使用 FP 的栈布局 …… 74
4.3.3 栈回溯 …… 76
4.4 实验 …… 78
4.4.1 实验 4-1：观察栈布局 …… 78
4.4.2 实验 4-2：观察栈回溯 …… 78
第 5 章 GNU 汇编器 …… 79
5.1 编译流程与 ELF 文件 …… 79
5.2 一个简单的汇编程序 …… 82
5.3 汇编语法 …… 84
5.3.1 注释 …… 84
5.3.2 符号 …… 84
5.4 常用的伪指令 …… 85
5.4.1 对齐伪指令 …… 85
5.4.2 数据定义伪指令 …… 86
5.4.3 与函数相关的伪指令 …… 87
5.4.4 与段相关的伪指令 …… 87
5.4.5 与宏相关的伪指令 …… 89
5.4.6 与文件相关的伪指令 …… 91
5.5 RISC-V 依赖特性 …… 91
5.5.1 RISC-V 特有的命令行选项 …… 91
5.5.2 RISC-V 特有的伪指令 …… 92
5.6 实验 …… 92
5.6.1 实验 5-1：汇编语言练习——查找最大数 …… 92
5.6.2 实验 5-2：汇编语言练习——通过 C 语言调用汇编函数 …… 92
5.6.3 实验 5-3：汇编语言练习——通过汇编语言调用 C 函数 …… 92
5.6.4 实验 5-4：使用汇编伪操作实现一张表 …… 92
5.6.5 实验 5-5：汇编宏的使用 …… 93
第 6 章 链接器与链接脚本 …… 94
6.1 链接器 …… 94
6.2 链接脚本 …… 95
6.2.1 一个简单的链接程序 …… 95
6.2.2 设置入口点 …… 96
6.2.3 基本概念 …… 97
6.2.4 符号赋值与引用 …… 97
6.2.5 当前位置计数器 …… 98
6.2.6 SECTIONS 命令 …… 99
6.2.7 常用的内置函数 …… 101
6.3 加载重定位 …… 103
6.3.1 BenOS 重定位 …… 103
6.3.2 OpenSBI 和 Linux 内核重定位 …… 105
6.4 链接重定位与链接器松弛优化 …… 108
6.4.1 链接重定位 …… 108
6.4.2 函数跳转优化 …… 112
6.4.3 符号地址访问优化 …… 114
6.5 实验 …… 116
6.5.1 实验 6-1：分析链接脚本 …… 116
6.5.2 实验 6-2：输出每个段的内存布局 …… 116
6.5.3 实验 6-3：加载地址不等于运行地址 …… 117
6.5.4 实验 6-4：设置链接地址 …… 117
6.5.5 实验 6-5：链接器松弛优化 1 …… 117
6.5.6 实验 6-6：链接器松弛优化 2 …… 117
6.5.7 实验 6-7：分析 Linux 5.15 内核的链接脚本 …… 117
第 7 章 内嵌汇编代码 …… 118
7.1 内嵌汇编代码基本用法 …… 118
7.1.1 基础内嵌汇编代码 …… 118
7.1.2 扩展内嵌汇编代码 …… 118
7.1.3 内嵌汇编代码修饰符 …… 120
7.1.4 使用汇编符号名字 …… 121
7.1.5 内嵌汇编代码与宏结合 …… 122
7.1.6 使用 goto 修饰词 …… 122
7.1.7 小结 …… 123
7.2 案例分析 …… 124
7.3 注意事项 …… 128
7.4 实验 …… 128
7.4.1 实验 7-1：实现简单的 memcpy()函数 …… 128
7.4.2 实验 7-2：使用汇编符号名写内嵌汇编代码 …… 128

7.4.3 实验 7-3：使用内嵌汇编代码完善 memset()函数……129
7.4.4 实验 7-4：使用内嵌汇编代码与宏的结合……129
7.4.5 实验 7-5：实现读和写系统寄存器的宏……129
7.4.6 实验 7-6：goto 模板的内嵌汇编代码……129

第 8 章　异常处理……130
8.1 异常处理基本概念……130
8.1.1 异常类型……130
8.1.2 同步异常和异步异常……131
8.1.3 异常入口和返回……131
8.1.4 异常返回地址……132
8.1.5 异常返回的处理器模式……133
8.1.6 栈的选择……133
8.2 与 M 模式相关的异常寄存器……133
8.2.1 mstatus 寄存器……134
8.2.2 mtvec 寄存器……134
8.2.3 mcause 寄存器……135
8.2.4 mie 寄存器……135
8.2.5 mtval 寄存器……136
8.2.6 mip 寄存器……136
8.2.7 mideleg 和 medeleg 寄存器……136
8.2.8 中断配置……137
8.3 与 S 模式相关的异常寄存器……137
8.3.1 sstatus 寄存器……137
8.3.2 sie 寄存器……137
8.3.3 sip 寄存器……138
8.3.4 scause 寄存器……138
8.3.5 stvec 寄存器……138
8.3.6 stval 寄存器……139
8.4 异常上下文……139
8.4.1 保存异常上下文……141
8.4.2 恢复异常上下文……141
8.5 案例分析 8-1：实现 SBI 系统调用……142
8.5.1 调用 ECALL 指令……142
8.5.2 实现 SBI 系统调用……143
8.6 案例分析 8-2：BenOS 的异常处理……148
8.6.1 设置异常向量表……148
8.6.2 保存和恢复异常上下文……149
8.6.3 异常处理……151
8.6.4 委托中断和异常……153
8.6.5 触发异常……153
8.7 实验……154
8.7.1 实验 8-1：在 SBI 中实现串口输入功能……154
8.7.2 实验 8-2：在 BenOS 中触发非法指令异常……155
8.7.3 实验 8-3：输出触发异常时函数栈的调用过程……155
8.7.4 实验 8-4：在 MySBI 中模拟实现 RDTIME 伪指令……155

第 9 章　中断处理与中断控制器……156
9.1 中断处理基本概念……156
9.1.1 中断类型……156
9.1.2 中断处理过程……157
9.1.3 中断委派和注入……158
9.1.4 中断优先级……158
9.2 CLINT……159
9.3 案例分析 9-1：定时器中断……160
9.3.1 访问 mtimer……160
9.3.2 在 MySBI 中实现定时器服务……160
9.3.3 定时器中断处理……161
9.3.4 打开中断总开关……163
9.3.5 小结……164
9.4 PLIC……164
9.4.1 中断号……165
9.4.2 中断优先级……166
9.4.3 中断使能寄存器……166
9.4.4 中断待定寄存器……166
9.4.5 中断优先级阈值寄存器……167
9.4.6 中断请求/完成寄存器……167
9.5 案例分析 9-2：串口中断……167
9.5.1 初始化 PLIC……168
9.5.2 使能串口 0 的接收中断……169

9.5.3 处理中断 …… 169
9.6 实验 …… 171
9.6.1 实验 9-1：定时器中断 …… 171
9.6.2 实验 9-2：使用汇编函数保存和恢复中断现场 …… 171
9.6.3 实验 9-3：实现并调试串口 0 中断 …… 171
第 10 章 内存管理 …… 172
10.1 内存管理基础知识 …… 172
10.1.1 内存管理的“远古时代” …… 172
10.1.2 地址空间的抽象 …… 174
10.1.3 分段机制 …… 175
10.1.4 分页机制 …… 175
10.2 RISC-V 内存管理 …… 178
10.2.1 页表分类 …… 179
10.2.2 Sv39 页表映射 …… 180
10.2.3 Sv48 页表映射 …… 182
10.2.4 页表项描述符 …… 183
10.2.5 页表属性 …… 185
10.2.6 与地址转换相关的寄存器 …… 186
10.3 物理内存属性与物理内存保护 …… 187
10.3.1 物理内存属性 …… 187
10.3.2 物理内存保护 …… 188
10.4 案例分析 10-1：在 BenOS 里实现恒等映射 …… 190
10.4.1 页表定义 …… 191
10.4.2 页表数据结构 …… 193
10.4.3 创建页表 …… 193
10.4.4 打开 MMU …… 198
10.4.5 测试 MMU …… 198
10.4.6 图解页表创建的过程 …… 200
10.5 内存管理实验 …… 204
10.5.1 实验 10-1：建立恒等映射 …… 204
10.5.2 实验 10-2：为什么 MMU 无法运行 …… 205
10.5.3 实验 10-3：实现一个 MMU 页表的转储功能 …… 205
10.5.4 实验 10-4：修改页面属性 …… 205
10.5.5 实验 10-5：使用汇编语言来建立恒等映射 …… 206
10.5.6 实验 10-6：在 MySBI 中实现和验证 PMP 机制 …… 206
第 11 章 高速缓存 …… 207
11.1 为什么需要高速缓存 …… 207
11.2 高速缓存的访问延时 …… 208
11.3 高速缓存的工作原理 …… 210
11.4 高速缓存的映射方式 …… 212
11.4.1 直接映射 …… 212
11.4.2 全相联映射 …… 213
11.4.3 组相联映射 …… 213
11.4.4 组相联的高速缓存的例子 …… 214
11.5 虚拟高速缓存与物理高速缓存 …… 215
11.5.1 物理高速缓存 …… 215
11.5.2 虚拟高速缓存 …… 215
11.5.3 VIPT 和 PIPT …… 215
11.6 重名和同名问题 …… 216
11.6.1 重名问题 …… 217
11.6.2 同名问题 …… 217
11.6.3 VIPT 产生的重名问题 …… 218
11.7 高速缓存策略 …… 220
11.8 高速缓存的维护指令 …… 221
11.8.1 高速缓存管理指令 …… 221
11.8.2 高速缓存预取指令 …… 222
第 12 章 缓存一致性 …… 224
12.1 为什么需要缓存一致性 …… 224
12.2 缓存一致性的分类 …… 225
12.2.1 缓存一致性协议发展历程 …… 225
12.2.2 缓存一致性分类 …… 226
12.2.3 系统缓存一致性问题 …… 227
12.3 缓存一致性的解决方案 …… 227
12.3.1 关闭高速缓存 …… 228
12.3.2 使用软件维护缓存一致性 …… 228
12.3.3 使用硬件维护缓存一致性 …… 228
12.4 MESI 协议 …… 228
12.4.1 MESI 协议简介 …… 229
12.4.2 本地读写与总线操作 …… 230
12.4.3 MESI 状态转换 …… 230

12.4.4 初始状态为 I ……231
12.4.5 初始状态为 M ……233
12.4.6 初始状态为 S ……234
12.4.7 初始状态为 E ……234
12.4.8 小结与案例分析 ……235
12.4.9 MOESI 协议 ……237
12.5 高速缓存伪共享 ……237
12.6 两种缓存一致性控制器 ……239
12.6.1 CCI 缓存一致性控制器 ……239
12.6.2 CCN 缓存一致性控制器 ……240
12.7 案例分析 12-1：伪共享的避免 ……241
12.8 案例分析 12-2：DMA 和高速缓存的一致性 ……242
12.8.1 从内存到设备的 FIFO 缓冲区 ……243
12.8.2 从设备的 FIFO 缓冲区到内存 ……243
12.9 案例分析 12-3：自修改代码的一致性 ……244
12.10 实验 ……245
12.10.1 实验 12-1：高速缓存伪共享 ……245
12.10.2 实验 12-2：使用 Perf C2C 发现高速缓存伪共享 ……245
第 13 章 TLB 管理 ……246
13.1 TLB 基础知识 ……247
13.2 TLB 重名与同名问题 ……249
13.2.1 重名问题 ……249
13.2.2 同名问题 ……250
13.3 ASID ……251
13.4 TLB 管理指令 ……253
13.4.1 TLB 维护指令介绍 ……253
13.4.2 TLB 广播 ……254
13.4.3 SFENCE.VMA 指令使用场景 ……256
13.5 TLB 案例分析 ……256
13.5.1 TLB 在 Linux 内核中的应用 ……256
13.5.2 ASID 在 Linux 内核中的应用 ……257
13.5.3 Linux 内核中的 TLB 维护操作 ……257
13.5.4 BBM 机制 ……259
第 14 章 原子操作 ……261
14.1 原子操作介绍 ……261
14.2 保留加载与条件存储指令 ……262
14.3 独占内存访问工作原理 ……263
14.3.1 独占监视器 ……264
14.3.2 独占监视器与缓存一致性 ……265
14.4 原子内存访问操作指令 ……266
14.4.1 原子内存访问指令工作原理 ……266
14.4.2 原子内存访问指令与 LR/SC 指令的效率对比 ……267
14.4.3 RISC-V 中的原子内存访问指令 ……268
14.5 比较并交换操作 ……270
第 15 章 内存屏障指令 ……275
15.1 内存屏障指令产生的原因 ……275
15.1.1 顺序一致性内存模型 ……276
15.1.2 处理器一致性内存模型 ……277
15.1.3 弱一致性内存模型 ……277
15.1.4 释放一致性内存模型 ……278
15.1.5 MCA 模型 ……279
15.2 RISC-V 约束条件 ……280
15.2.1 全局内存次序与保留程序次序 ……280
15.2.2 RVWMO 的约束规则 ……281
15.3 RISC-V 中的内存屏障指令 ……284
15.3.1 使用内存屏障的场景 ……284
15.3.2 FENCE 指令 ……285
15.3.3 内置获取和释放屏障原语的指令 ……285
15.3.4 FENCE.I 指令 ……286
15.3.5 SFENCE.VMA 指令 ……286
15.4 RISC-V 内存屏障指令移植指南 ……286
15.4.1 从 RISC-V 到 x86 体系结构 ……286
15.4.2 从 RISC-V 到 ARM 体系结构 ……287

15.4.3 Linux 内核常用的内存屏障 API 函数 …… 287
15.5 案例分析 …… 288
15.5.1 消息传递问题 …… 288
15.5.2 单方向内存屏障与自旋锁 …… 289
15.5.3 邮箱传递消息 …… 290
15.5.4 关于 DMA 的案例 …… 291
15.5.5 在 Linux 内核中使指令高速缓存失效 …… 291
15.6 模拟和测试内存屏障故障 …… 291
15.6.1 使用 Litmus 测试工具集 …… 292
15.6.2 编写 C 程序来模拟 …… 295
15.7 实验 …… 297
15.7.1 实验 15-1：编写 Litmus 脚本并测试内存一致性 1 …… 297
15.7.2 实验 15-2：编写 Litmus 脚本并测试内存一致性 2 …… 298
第 16 章　合理使用内存屏障指令 …… 299
16.1 存储缓冲区与写内存屏障指令 …… 300
16.2 无效队列与读内存屏障指令 …… 305
16.3 内存屏障指令总结 …… 307
16.4 案例分析：Linux 内核中的内存屏障指令 …… 308
16.4.1 第一次使用内存屏障指令 …… 309
16.4.2 第二次使用内存屏障指令 …… 310
16.4.3 第三次使用内存屏障指令 …… 313
16.4.4 第四次使用内存屏障指令 …… 314
16.4.5 小结：内存屏障指令的使用 …… 315
16.5 实验 …… 315
16.5.1 实验 16-1：验证和测试内存一致性 1 …… 315
16.5.2 实验 16-2：验证和测试内存一致性 2 …… 315
16.5.3 实验 16-3：验证和测试内存一致性 3 …… 315
第 17 章　与操作系统相关的内容 …… 316
17.1 C 语言常见陷阱 …… 317
17.1.1 数据模型 …… 317
17.1.2 数据类型转换与整型提升 …… 318
17.1.3 移位操作 …… 320
17.2 创建进程 …… 320
17.2.1 进程控制块 …… 320
17.2.2 0 号进程 …… 321
17.2.3 do_fork()函数的实现 …… 323
17.2.4 进程上下文切换 …… 324
17.2.5 新进程的第一次执行 …… 326
17.3 简易进程调度器 …… 327
17.3.1 扩展进程控制块 …… 327
17.3.2 就绪队列 …… 327
17.3.3 调度类 …… 328
17.3.4 简易调度器的实现 …… 329
17.3.5 自愿调度 …… 330
17.3.6 抢占调度 …… 331
17.3.7 测试用例 …… 333
17.3.8 关于调度的思考 …… 333
17.4 让进程运行在用户模式 …… 335
17.5 系统调用 …… 338
17.5.1 系统调用介绍 …… 338
17.5.2 在用户模式下调用 SVC 指令 …… 338
17.5.3 在内核模式下对系统调用的处理 …… 339
17.5.4 系统调用表 …… 340
17.6 实现 clone 系统调用 …… 341
17.7 实验 …… 343
17.7.1 实验 17-1：进程创建 …… 343
17.7.2 实验 17-2：进程调度 …… 343
17.7.3 实验 17-3：让进程运行在用户模式 …… 343
17.7.4 实验 17-4：新增一个 malloc() 系统调用 …… 343
17.7.5 实验 17-5：新增一个 clone() 系统调用 …… 344
第 18 章　可伸缩矢量计算与优化 …… 345
18.1 矢量计算基本概念 …… 345
18.1.1 SISD 与 SIMD …… 345
18.1.2 定长计算与可变长矢量计算 …… 347

18.1.3 通道……347
18.1.4 矢量与标量……347
18.2 RVV 寄存器……348
18.2.1 矢量寄存器……348
18.2.2 mstatus 寄存器中的矢量上下文状态……348
18.2.3 vtype 寄存器……348
18.2.4 vl 寄存器……350
18.2.5 vlenb 寄存器……351
18.2.6 vstart 寄存器……351
18.3 配置编译和运行环境……351
18.3.1 搭建编译环境……351
18.3.2 运行第一个“hello RVV!”程序……352
18.3.3 单步调试汇编程序……353
18.3.4 单步调试 C 语言与汇编混合程序……355
18.4 RVV 指令格式……357
18.5 配置指令……358
18.6 加载和存储指令……360
18.6.1 单位步长模式……360
18.6.2 任意步长模式……363
18.6.3 聚合加载/离散存储……364
18.6.4 打包数据的加载与存储……365
18.6.5 首次异常加载指令……367
18.6.6 加载和存储全部矢量数据……368
18.7 矢量掩码指令……369
18.7.1 逻辑操作指令……369
18.7.2 VCPOP.M 指令……369
18.7.3 VFIRST.M 指令……370
18.7.4 VMSBF.M 指令……370
18.7.5 VMSIF.M 指令……371
18.7.6 VMSOF.M 指令……372
18.8 矢量整型算术指令……372
18.8.1 加宽和变窄算术指令……372
18.8.2 加法和减法指令……373
18.8.3 加宽模式的加法和减法指令……374
18.8.4 位操作指令……376
18.8.5 移位操作指令……376
18.8.6 比较指令……376
18.8.7 数据搬移指令……377
18.9 案例分析 18-1：使用 RVV 指令优化 strcmp()函数……377
18.9.1 使用纯汇编方式……378
18.9.2 测试……379
18.10 案例分析 18-2：RGB24 转 BGR24……380
18.10.1 使用 C 语言实现 RGB24 转 BGR24……380
18.10.2 使用 RVV 指令优化……380
18.10.3 测试……381
18.11 案例分析 18-3：4×4 矩阵乘法运算……382
18.11.1 使用 C 语言实现 4×4 矩阵乘法运算……382
18.11.2 使用 RVV 指令优化……383
18.11.3 测试……387
18.12 案例分析 18-4：使用 RVV 内置函数……388
18.13 案例分析 18-5：自动矢量优化……388
18.14 术语……390
18.15 实验……391
18.15.1 实验 18-1：RGB24 转 BGR32……391
18.15.2 实验 18-2：8×8 矩阵乘法运算……391
18.15.3 实验 18-3：使用 RVV 指令优化 strcpy()函数……391
18.15.4 实验 18-4：使用 RVV 内置函数优化……391
18.15.5 实验 18-5：使用 RVV 优化转置矩阵的求法……392
第 19 章 压缩指令扩展……393
19.1 RISC-V 指令集的特点……393
19.2 RVC 支持的指令格式与指令编码……394
第 20 章 虚拟化扩展……396
20.1 虚拟化技术介绍……396
20.1.1 虚拟化技术的发展历史……396

20.1.2 虚拟机管理程序的分类 …… 398
20.1.3 内存虚拟化 …… 398
20.1.4 I/O 虚拟化 …… 399
20.2 RISC-V 虚拟化扩展 …… 399
20.2.1 CPU 虚拟化扩展 …… 399
20.2.2 M 模式下系统寄存器的扩展 …… 400
20.2.3 HS 模式下的系统寄存器 … 402
20.2.4 VS 模式下的系统寄存器 … 404
20.3 RISC-V 内存虚拟化 …… 404
20.4 RISC-V 虚拟化扩展中的新增指令 …… 406
20.4.1 加载与存储虚拟机内存指令 …… 406
20.4.2 虚拟化内存屏障指令 …… 406
20.5 进入和退出虚拟机 …… 407
20.5.1 异常陷入 …… 408
20.5.2 异常返回 …… 408
20.5.3 新增的中断与异常类型 …… 409
20.6 中断虚拟化 …… 410
20.6.1 虚拟中断注入 …… 410
20.6.2 陷入与模拟 …… 411
20.7 案例分析 20-1：进入和退出虚拟机 …… 412
20.7.1 进入虚拟机 …… 413
20.7.2 退出虚拟机 …… 414
20.8 案例分析 20-2：建立虚拟化两阶段地址映射 …… 415
20.8.1 建立第二阶段的地址映射 …… 416
20.8.2 建立第一阶段的地址映射 …… 418
20.8.3 测试 …… 419
20.9 案例分析 20-3：在虚拟机中实现虚拟定时器 …… 420
20.10 案例分析 20-4：在 VMM 中加载和存储虚拟机内存地址 …… 422
20.11 案例分析 20-5：在 VMM 中模拟串口设备 …… 424
20.12 实验 …… 429
20.12.1 实验 20-1：加载虚拟机 1 …… 429
20.12.2 实验 20-2：加载虚拟机 2 …… 430
20.12.3 实验 20-3：虚拟化地址映射 …… 430
20.12.4 实验 20-4：解析虚拟机陷入的指令 …… 430
20.12.5 实验 20-5：在 VMM 中模拟实现 vPLIC …… 430
20.12.6 实验 20-6：在虚拟机中加载并运行 Linux 内核 …… 430
附录 A 关于 RISC-V 体系结构自测题的参考答案与提示 …… 431
附录 B RV64I 指令速查表 …… 433
附录 C RV64M 指令速查表 …… 437
附录 D RV64 常用伪指令速查表 …… 439

第 1 章　RISC-V 体系结构基础知识

本章思考题

1. RISC-V 体系结构有什么特点？
2. RISC-V 体系结构处理器包含多少个通用寄存器？
3. RISC-V 体系结构包含几种处理器模式？它们分别有什么作用？
4. 在 RISC-V 体系结构中，Hart 代表什么意思？
5. 在 RISC-V 体系结构中，什么是 SBI？
6. 在香山处理器体系结构中，前端子系统包括哪些模块？
7. 在香山处理器体系结构中，后端子系统包括哪些模块？
8. 在香山处理器体系结构中，如何解决高速缓存别名问题？

本章主要介绍 RISC-V 体系结构基础知识。

1.1 RISC-V 介绍

RISC（Reduced Instruction Set Computer）表示精简指令集计算机，而 RISC-V 源自美国加州大学伯克利分校，其中 V 表示第 5 代，因为在此之前加州大学伯克利分校已经完成了 4 代 RISC 体系结构的设计。

早在 2010 年，加州大学伯克利分校的研究人员在对比市面上所有的 RISC 体系结构后发现，指令集越来越复杂和臃肿，而且授权费昂贵，于是他们打算自行设计一套全新的开源指令集。

2015 年，RISC-V 基金会成立，旨在维护指令集以及体系结构规范的标准化和完整性。此外，他们还成立了 SiFive 公司，以推动 RISC-V 商业化落地。

RISC-V 基金会吸引了全球高校、学术机构以及芯片设计、软件工具等商业公司加入，负责维护 RISC-V 指令集标准规范以及各种体系结构和扩展特性的规范文档的审阅与发布。

1.1.1 RISC-V 指令集优点

作为一个后起之秀，RISC-V 指令集在设计时充分借鉴了其他商业化指令的优点，并且吸取了它们的经验和教训。RISC-V 指令集具有如下优点。

- 设计简洁。RISC-V 指令集和基础体系结构设计都相当简洁，特别是 RISC-V 的指令集文档和基础体系结构设计文档一共才几百页，而一些商业 RISC 芯片的文档多达上万页。
- 模块化。RISC-V 指令集采用模块化设计思想。它具有一个最小的指令集，这个指令集可以完整地实现一个软件栈，然后通过模块化的方式实现其他扩展功能的指令，如浮点数乘法和除法指令、矢量指令等。模块化的 RISC-V 体系结构使得用户能够灵活选

择不同的模块组合，以适应不同的应用场景，从嵌入式设备到服务器都可以使用 RISC-V 体系结构进行设计。

- 开源。RISC-V 指令集采用 BSD 开源协议授权。使用 RISC-V 进行教学、学术研究、商业化都不需要授权费。
- 具有丰富的软件生态。目前大部分开源软件支持 RISC-V 指令集，包括 Linux 内核、GCC 等。

1.1.2　RISC-V 指令集扩展

RISC-V 针对 32 位处理器的最小指令集是 RV32I，表示 32 位基础整型指令集，它大约包含 40 条指令，可以在 32 位处理器上实现一个完整的软件栈。RV32I 是 RISC-V 指令集中固定不变的最小指令集。RISC-V 针对 64 位处理器的最小指令集是 RV64I，它表示 64 位基础整型指令集。RV64I 在 RV32I 的基础上添加了对字（word）、双字（double word）和长整型（long）版本指令的支持，并且将所有寄存器扩展到 64 位。本章主要介绍 RV64I 指令集。本书中把 32 位处理器使用的指令集简称为 **RV32 指令集**，把 64 位处理器使用的指令集简称为 **RV64 指令集**。

在 RV32I 和 RV64I 的基础上，RISC-V 指令集还支持模块化扩展，以支持更多特性，如表 1.1 所示。

表 1.1　RISC-V 扩展指令集

扩展指令集	说明
F	单精度浮点数扩展指令集
D	双精度浮点数扩展指令集
Q	4 倍精度浮点数扩展指令集
M	整型乘法和除法扩展指令集
C	压缩指令集（见第 19 章）
A	原子操作指令集（见第 14 章）
B	位操作指令集
E	为嵌入式设计的整型指令集
H	虚拟化扩展指令集（见第 20 章）
K	密码运算扩展指令集
V	可伸缩矢量扩展指令集（见第 18 章）
P	打包 SIMD（packed-SIMD）扩展指令集
J	动态翻译语言（dynamically translated language）扩展指令集
T	事务内存（transactional memory）指令集
N	用户态中断指令集

芯片设计人员可以根据项目需求和成本选择不同的扩展指令集，以适应不同的应用场景。RISC-V 为芯片设计人员提供了一个稳定的指令集组合，称为 RV32G/RV64G。其中，G 表示 IMAFD，实现基础指令集、整型乘法/除法扩展指令、单/双精度浮点数扩展指令以及原子操作指令。

1.1.3　RISC-V 商业化发展

一个全新的指令集要蓬勃发展需要上游与下游公司都参与进来。为了验证和推广 RISC-V，RISC-V 创始人成立了 SiFive 公司，为 RISC-V 的商业化推广起到示范作用。除开源基于 RISC-V 的 CPU 及片上系统（System on Chip，SoC）实现之外，SiFive 还提供基于 RISC-V 体系结构的商业化处理器 IP、开发工具和芯片解决方案等服务。

在国内也有许多大公司和初创公司加入 RISC-V 芯片设计的队伍中。此外，国内的一些研究机构（如中国科学院计算技术研究所的香山处理器团队）还发布了开源的香山处理器。

1.2 RISC-V 体系结构介绍

1.2.1 RISC-V 体系结构

RISC-V 体系结构包括指令集规范和体系结构规范，以及众多的扩展规范，这些都由 RISC-V 基金会审阅与发布。RISC-V 体系结构具有如下特点。

- 对学术界和工业界完全开放。
- 真正适合硬件实现的指令集体系结构，而不是一个模拟或者二进制翻译的指令集体系结构。
- 属于一个通用的指令集体系结构，而不是针对某个特定微体系结构的实现。
- 实现最小的整数指令集，作为基础指令集，可以用于教学。在此基础上还实现众多可选扩展指令，以支持通用软件开发。
- 支持 IEEE 754 浮点标准。
- 支持众多扩展指令集。
- 支持 32 位及 64 位地址空间。
- 支持多核及异构体系结构。
- 支持可选的压缩指令编码，以提高性能和能源效率并优化静态代码。
- 支持虚拟化扩展。
- 支持可伸缩矢量指令扩展（参见 GitHub 上的“Working Draft of the Proposed RISC-V V Vector Extension”）。

RISC-V 支持 32 位指令集和 64 位指令集。本书重点介绍 64 位指令集。

1.2.2 采用 RISC-V 体系结构的常见处理器

下面介绍市面上常见的采用 RISC-V 体系结构的处理器。

1. SiFive FU740 处理器

SiFive 公司基于 RISC-V 体系结构开发了商业化 64 位处理器芯片。广受 RISC-V 开发者喜爱的 HiFive Unmatched 开发板基于该处理器芯片。

FU740 内置 1 个 S7 处理器内核以及 4 个 U7 处理器内核。其中，S7 处理器内核用于系统监控，U7 处理器内核用于计算和应用，如图 1.1 所示。

FU740 处理器具有如下特性。

- 完全兼容 RISC-V 体系结构规范。
- 属于 64 位 RISC-V 处理器内核。
- 具有 32KB L1 指令高速缓存。
- 具有 32KB L1 数据高速缓存。
- 支持 8 个物理内存保护区。
- 支持 128KB 大小的 L2 高速缓存。
- 内置处理器内核本地中断控制器（Core-Local Interrupt Controller，CLINT）。
- 支持平台级别的中断控制器（Platform-Level Interrupt Controller，PLIC），最多支持 128 个外设中断和 7 级中断优先级。

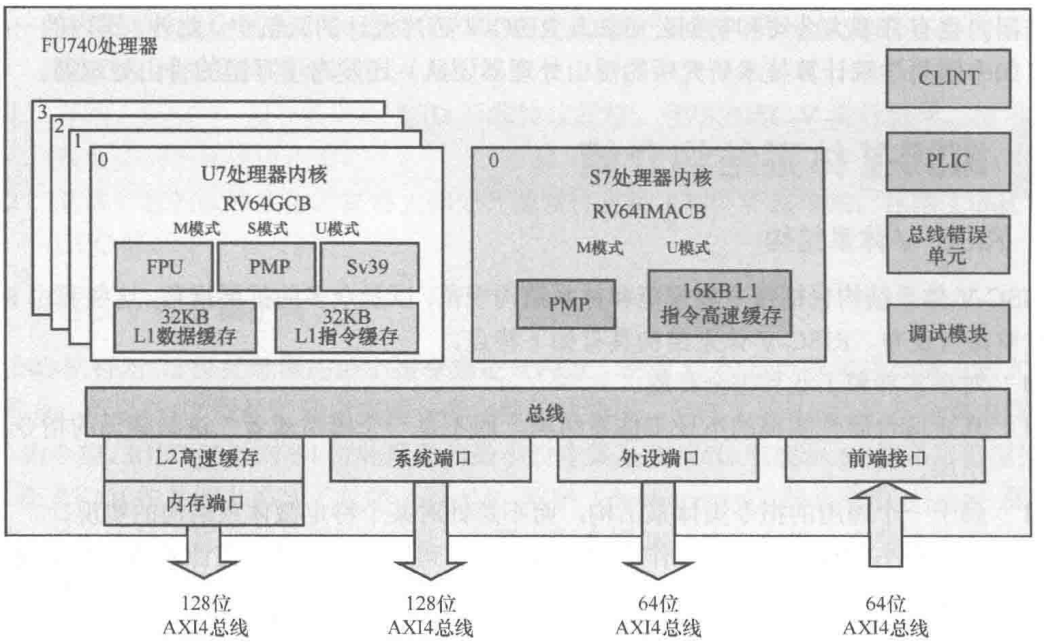


图 1.1　FU740 处理器框架

2. 香山处理器

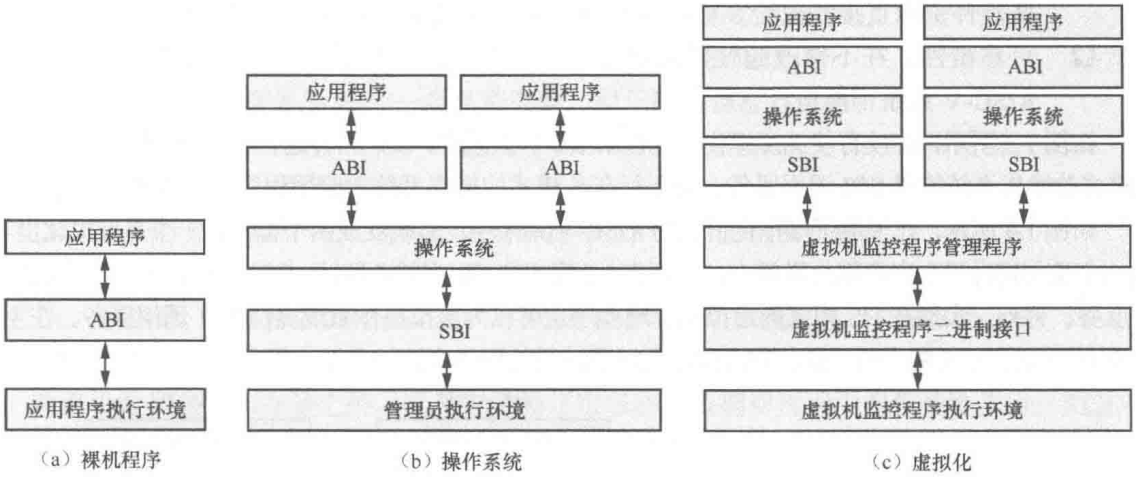
香山处理器是由中国科学院计算技术研究所的香山处理器团队推出的高性能 64 位开源处理器。它的体系结构代号以湖命名，第一版体系结构代号是“雁栖湖”，第二版体系结构代号是“南湖”，详见 1.4 节。

1.2.3　RISC-V 体系结构中的基本概念

本节介绍 RISC-V 体系结构中的一些基本概念和定义。

- 执行环境接口（Execution Environment Interface，EEI）：包括程序的初始状态、CPU 的类型和数量、支持的 S 模式、内存和 I/O 区域的可访问性与属性、在每个 CPU 上执行指令的行为以及任何异常（包括中断、系统调用）的处理等。常见的 RISC-V 执行环境接口包括 Linux 应用程序二进制接口（Application Binary Interface，ABI）以及 RISC-V 管理员二进制接口（Supervisor Binary Interface，SBI）。一个 RISC-V 执行环境接口可以是由纯硬件、纯软件或者软硬件结合实现的，示例如下。
  - 裸机程序：程序直接通过 ABI 访问硬件资源，如图 1.2（a）所示。
  - 操作系统：提供 U（User）模式的执行环境并复用到物理处理器线程中，同时提供虚拟内存机制。在操作系统与管理员执行环境（Supervisor Execution Environment，SEE）之间提供一层 SBI，它对所有 RISC-V 硬件平台中共性的功能做了抽象，为操作系统访问 M 模式的硬件资源提供服务，如图 1.2（b）所示。
  - 虚拟化：为客户操作系统提供多个特权模式的执行环境。如图 1.2（c）所示，每个客户操作系统通过 SBI 访问虚拟化管理程序。此外，虚拟化管理程序可以通过虚拟机监控程序二进制接口（Hypervisor Binary Interface，HBI）访问虚拟机监控程序执行环境（Hypervisor Execution Environment，HEE）中的硬件资源。
- 模拟器：在主机上模拟 RISC-V 的用户模式以及特权模式的执行环境。
- 硬件线程（Hart）：一个处理器执行线程，在执行环境中自主获取和执行 RISC-V 指令资源的硬件单元，这个术语类似于 x86 体系结构中的超线程以及 ARMv8 体系结构中

的处理机（processing element）。SMT 技术让同一个处理器内核上的多个线程同步执行并共享处理器的执行资源。假设一个 RISC-V 处理器实现了超线程技术，那么一个处理器内核就有多个硬件线程，不过目前商用的 RISC-V 处理器还没有实现超线程技术。



（a）裸机程序　（b）操作系统　（c）虚拟化

图 1.2　执行环境

特权级别（privilege level）包括 RISC-V 处理器提供的 3 种模式。

- 机器模式（M 模式）。以 M 模式运行的代码通常在本质上是可信的，因为它具有对机器实现的全部访问权限。M 模式通常可用于管理 RISC-V 上的安全执行环境。通常在 M 模式下运行 SBI 固件，为操作系统提供服务。
- 特权模式（S 模式）。S 模式通常用来运行操作系统的内核，为应用程序提供服务。
- 用户模式（U 模式）。U 模式的特权级别最低，通常运行应用程序。

使能了虚拟化扩展后，新增如下特权模式。

- HS 模式。把原有的 S 模式扩展为 HS 模式，用来运行虚拟化管理程序。
- VS 模式。VS 模式通常用来运行虚拟机操作系统内核。
- VU 模式。VU 模式通常运行虚拟机操作系统中的应用程序。

所有的硬件实现必须提供 M 模式，因为这是唯一可以不受限制地访问整个机器的资源。简单的 RISC-V 系统（如嵌入式系统）只提供 M 模式，如表 1.2 所示。

表 1.2　特权级别使用场景

特权级别个数	支持的模式	使用场景
1	M 模式	嵌入式系统
2	M 模式和 U 模式	具有安全特性的嵌入式系统
3	M 模式、S 模式和 U 模式	通用操作系统
5	M 模式、HS 模式、VS 模式、VU 模式和 U 模式	虚拟化操作系统

1.2.4 SBI 服务

在 RISC-V 软件生态中，RISC-V 规范定义了一个 SBI 规范。SBI 对所有 RISC-V 硬件平台中共性的功能做了抽象，为运行在 S 模式的操作系统或者虚拟机监控程序扩展的特权（Hypervisor-extended Supervisor，HS）模式的虚拟化管理软件提供统一的服务接口。

在现代操作系统体系结构中，内核空间和用户空间之间多了一个中间层，即系统调用层，它为用户空间中的应用程序提供硬件抽象接口。SBI 非常类似于操作系统中的系统调用层，它有如下几个优点。

- 它为运行在低级别的处理器模式提供访问 M 模式下硬件资源（如 S 模式下的操作系统或者 U 模式下的应用程序）的抽象接口。
- 保证系统稳定和安全。因为 M 模式下处理器具有对系统资源的全部访问权限，如果有些硬件资源直接开放给 S 模式和 U 模式，就会造成系统的安全问题。
- 可移植性。在不修改源代码的情况下，让不同的操作系统或者拥有不同厂商设计的 RISC-V 系统都能运行。

如图 1.3 所示，在没有使能虚拟化扩展的 RISC-V 系统中，SBI 固件运行在 M 模式，它为运行在 S 模式的操作系统提供 SBI 调用服务，而运行在 S 模式的操作系统为应用程序提供系统调用服务。

如图 1.4 所示，在使能了虚拟化扩展的 RISC-V 系统中，系统变成两个世界：一个是虚拟化世界，另一个是主机世界。在虚拟化世界中，实现在 M 模式的 SBI 固件可以为虚拟化管理程序提供 SBI 调用服务。另外，实现在 HS 模式的虚拟化管理程序也可以为虚拟操作系统提供 SBI 调用服务。在主机世界中，运行在 U 模式的主机的应用程序可以通过系统调用来访问虚拟化管理程序提供的服务。

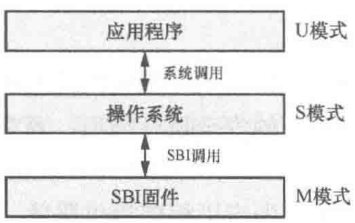


图 1.3　没有使能虚拟化扩展的 RISC-V 系统

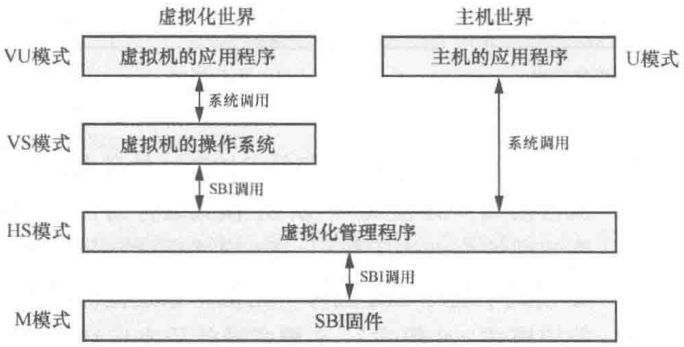


图 1.4　使能了虚拟化的 RISC-V 系统

1.3 RISC-V 寄存器

1.3.1　通用寄存器

64 位的 RISC-V 体系结构提供 32 个 64 位的整型通用寄存器，分别是 x0～x31 寄存器，而 32 位的 RISC-V 体系结构提供 32 个 32 位的整型通用寄存器，如图 1.5 所示。对于浮点数运算，64 位的 RISC-V 体系结构也提供 32 个浮点数通用寄存器，分别是 f0～f31 寄存器。

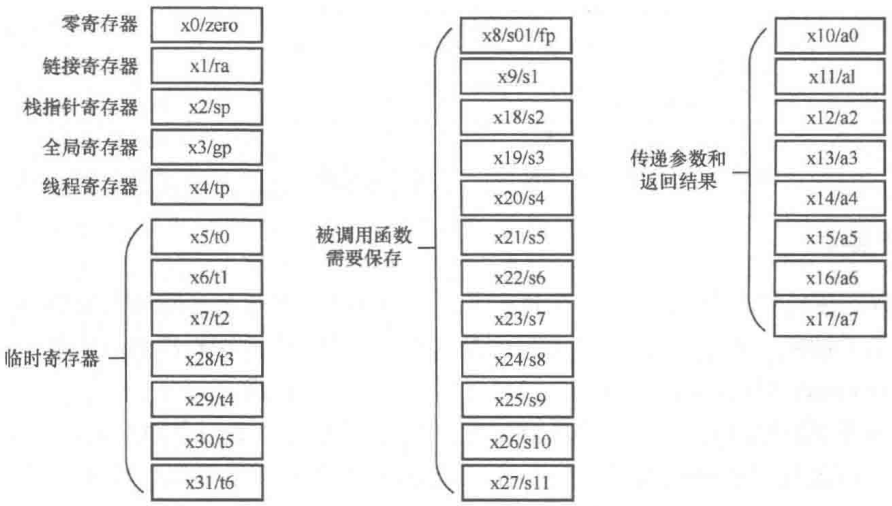


图 1.5　RISC-V 整型通用寄存器

RISC-V 的通用寄存器通常具有别名和特殊用途，在书写汇编指令时可以直接使用别名。

- x0 寄存器的别名为 zero。寄存器的内容全是 0，可以用作源寄存器，也可以用作目标寄存器。
- x1 寄存器的别名为 ra——链接寄存器，用于保存函数返回地址。
- x2 寄存器的别名为 sp——栈指针寄存器，指向栈的地址。
- x3 寄存器的别名为 gp——全局寄存器，用于链接器松弛优化。
- x4 寄存器的别名为 tp——线程寄存器，通常在操作系统中保存指向进程控制块——task_struct 数据结构的指针。
- x5～x7 以及 x28～x31 寄存器为临时寄存器，它们的别名分别是 t0～t6。
- x8～x9 以及 x18～x27 寄存器的别名分别是 s0～s11。如果在函数调用过程中使用这些寄存器，需要保存到栈里。另外，S0 寄存器可以用作栈指针（Frame Pointer，FP）。
- x10～x17 寄存器的别名分别为 a0～a7，在函数调用时传递参数和返回值。

除用于数据运算和存储之外，通用寄存器还可以在函数调用过程中起到特殊作用。RISC-V 体系结构的函数调用规范对此有所约定，详见第 4 章。

1.3.2 系统寄存器

除上面介绍的通用寄存器之外，RISC-V 体系结构还定义了很多的系统控制和状态寄存器（Control and Status Register，CSR），通过访问和设置这些系统寄存器完成对处理器不同的功能配置。

RISC-V 体系结构支持如下 3 类系统寄存器：

- M 模式的系统寄存器；
- S 模式的系统寄存器；
- U 模式的系统寄存器。

程序可以通过 CSR 指令（如 CSRRW 指令）访问系统寄存器，详见 3.11 节。

在 CSR 指令编码中预留了 12 位编码空间（csr[11:0]）用来索引系统寄存器，如图 1.6 中的 csr 字段，即指令编码中的 Bit[31:20]。

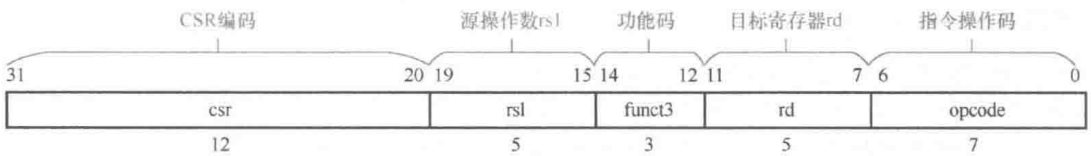


图 1.6 CSR 指令编码

RISC-V 体系结构对 12 位 CSR 编码空间继续做了约定。其中，Bit[11:10]用来表示系统寄存器读写属性，0b11 表示只读，其余表示可读可写。Bit[9:8]表示允许访问该系统寄存器的处理器模式，0b00 表示 U 模式，0b01 表示 S 模式，0b10 表示 HS/VS 模式，0b11 表示 M 模式。剩余的位用作寄存器的索引。使用 CSR 地址的最高位对默认的访问权限进行编码，简化了硬件中的错误检查，并提供了更大的 CSR 编码空间，但限制了 CSR 到地址空间的映射。CSR 地址空间映射如表 1.3 所示。

表 1.3 CSR 地址空间映射

地址范围	CSR 编码			访问模式	访问权限
	Bit[11:10]	Bit[9:8]	Bit[7:4]		
0x000～0x0FF	00	00	XXXX	U	读写
0x400～0x4FF	01	00	XXXX	U	读写
0x800～0x8FF	10	00	XXXX	U	读写（用户自定义系统寄存器）

续表

地址范围	CSR 编码			访问模式	访问权限
	Bit[11:10]	Bit[9:8]	Bit[7:4]		
0xC00～0xC7F	11	00	0XXX①	U	只读
0xC80～0xCBF	11	00	10XX	U	只读
0xCC0～0xCFF	11	00	11XX	U	只读
0x100～0x1FF	00	01	XXXX	S	读写
0x500～0x57F	01	01	0XXX	S	读写
0x580～0x5BF	01	01	10XX	S	读写
0x5C0～0x5FF	01	01	11XX	S	读写（用户自定义系统寄存器）
0x900～0x97F	10	01	0XXX	S	读写
0x980～0x9BF	10	01	10XX	S	读写
0x9C0～0x9FF	10	01	11XX	S	读写（用户自定义系统寄存器）
0xD00～0xD7F	11	01	0XXX	S	只读
0xD80～0xDBF	11	01	10XX	S	只读
0xDC0～0xDFF	11	01	11XX	S	只读（用户自定义系统寄存器）
0x300～0x3FF	00	11	XXXX	M	读写
0x700～0x77F	01	11	0XXX	M	读写
0x780～0x79F	01	11	100X	M	读写
0x7A0～0x7AF	01	11	1010	M	读写（用于调试寄存器）
0x7B0～0x7BF	01	11	1011	M	读写（只能用于调试寄存器）
0x7C0～0x7FF	01	11	11XX	M	读写（用户自定义系统寄存器）
0xB00～0xB7F	10	11	0XXX	M	读写
0xB80～0xBBF	10	11	10XX	M	读写
0xBC0～0xBFF	10	11	11XX	M	读写（用户自定义系统寄存器）
0xF00～0xF7F	11	11	0XXX	M	只读
0xF80～0xFBF	11	11	10XX	M	只读
0xFC0～0xFFF	11	11	11XX	M	只读（用户自定义系统寄存器）

① CSR 编码中的“X”可以是 0 或 1。

下面的访问行为会触发非法指令异常。

- 访问不存在或者没有实现的系统寄存器。
- 尝试写入只具有只读属性的系统寄存器。
- 在低级别的处理器模式下访问高级别的处理器模式的系统寄存器，例如，在 S 模式下访问 M 模式的系统寄存器。

1.3.3　U 模式下的系统寄存器

U 模式下的系统寄存器如表 1.4 所示。

表 1.4　　U 模式下的系统寄存器

地址	CSR 名称	属性	说明
0x001	fflags	URW	浮点数累积异常（accrued exception）
0x002	frm	URW	浮点数动态舍入模式（dynamic rounding mode）
0x003	fcsr	URW	浮点数控制和状态寄存器
0xC00	cycle	URO	读取时钟周期，映射到 RDCYCLE 伪指令
0xC01	time	URO	读取 time 系统寄存器的值，映射到 RDTIME 伪指令
0xC02	instret	URO	执行指令数目，映射到 RDINSTRET 伪指令
0xC03～0xC1F	hpmcounter3～hpmcounter31	URO	性能监测寄存器

RDCYCLE 伪指令读取 cycle 系统寄存器的值，返回处理器内核执行的时钟周期数。注意，它返回的是物理处理器内核而不是处理器硬件线程的时钟周期数。RDCYCLE 伪指令的主要目的是进行性能监控和调优。

RDTIME 伪指令读取 time 系统寄存器的值，获取系统的实际时间。系统每次启动时读取互补金属-氧化物-半导体（Complementary Metal-Oxide-Semiconductor，CMOS）上的 RTC（实时时钟）计数，当时钟中断到来时，更新该计数。

RDINSTRET 伪指令读取 instret 系统寄存器的值，返回处理器执行线程已经执行的指令数量。

hpmcounter3～hpmcounter31 为 29 个用于系统性能监测的寄存器，这些计数器的计数记录平台的事件，并通过额外的特权寄存器进行配置。

RDCYCLE、RDTIME 以及 RDINSTRET 伪指令在有些处理器上是通过 SBI 固件进行软件模拟实现的。例如，在 U 模式下使用 RDTIME 伪指令会触发非法指令异常，处理器陷入 M 模式。在 M 模式下的异常处理程序会读取 time 系统寄存器的值，然后返回 U 模式。

1.3.4 S 模式下的系统寄存器

S 模式下的系统寄存器如表 1.5 所示。

表 1.5　　S 模式下的系统寄存器

地址	CSR 名称	属性	说明
0x100	sstatus	SRW	S 模式下的处理器状态寄存器
0x104	sie	SRW	S 模式下的中断使能寄存器
0x105	stvec	SRW	S 模式下的异常向量表入口地址寄存器
0x106	scounteren	SRW	S 模式下的计数使能寄存器
0x10A	senvcfg	SRW	S 模式下的环境配置寄存器
0x140	sscratch	SRW	用于异常处理的临时寄存器
0x141	sepc	SRW	S 模式下的异常模式程序计数器（Program Counter，PC）寄存器
0x142	scause	SRW	S 模式下的异常原因寄存器
0x143	stval	SRW	S 模式下的异常向量寄存器
0x144	sip	SRW	S 模式下的中断待定寄存器
0x180	satp	SRW	S 模式下的地址转换与保护寄存器
0x5A8	scontext	SRW	S 模式下的上下文寄存器（用于调试）

接下来，介绍表 1.5 中的部分寄存器。

1. sstatus 寄存器

sstatus 寄存器表示 S 模式下的处理器状态，如图 1.7 所示。

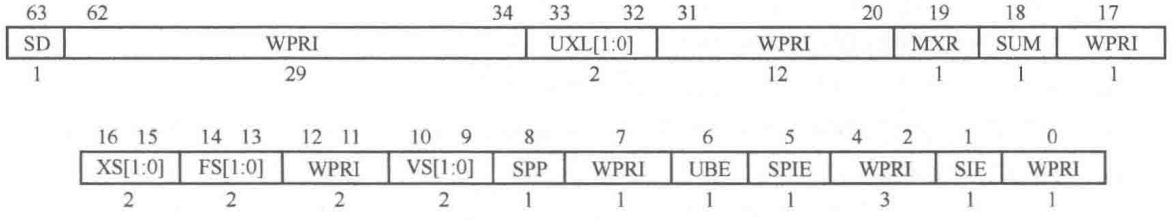


图 1.7　sstatus 寄存器

图 1.7 中的 WPRI 表示这些字段是保留的，软件应该忽略从这些字段读取的值，并且在向同一寄存器的其他字段写入值时，应该保留这些字段中保存的值。通常，为了向前兼容，硬件会将这些字段设为只读的零值。sstatus 寄存器中其他字段的含义如表 1.6 所示。

表 1.6　　sstatus 寄存器中其他字段的含义

字段	位段	说明
SIE	Bit[1]	中断使能位，用来使能和关闭 S 模式中所有的中断
SPIE	Bit[5]	中断使能保存位。当一个异常陷入 S 模式时，SIE 的值保存到 SPIE 中，SIE 设置为 0。当调用 SRET 指令返回时，从 SPIE 中恢复 SIE，然后 SPIE 设置为 1
UBE	Bit[6]	用来控制 U 模式下加载和存储指令访问内存的大小端模式。 0：小端模式。 1：大端模式
SPP	Bit[8]	陷入 S 模式之前 CPU 的处理模式。 0：表示从 U 模式陷入 S 模式。 1：表示在 S 模式触发的异常
VS	Bit[10:9]	用来使能 RVV
FS	Bit[14:13]	用来使能浮点数单元
XS	Bit[16:15]	用来使能其他 U 模式下扩展的状态
SUM	Bit[18]	设置在 S 模式下能否允许访问 U 模式下的内存。 0：在 S 模式下访问 U 模式的内存时会触发异常。 1：在 S 模式下可以访问 U 模式的内存
MXR	Bit[19]	访问内存的权限。 0：可以加载只读页面。 1：可以加载可读和可执行的页面
UXL	Bit[33:32]	用来表示 U 模式的寄存器长度，通常是一个只读字段，并且 U 模式下寄存器的长度等于 S 模式下寄存器的长度
SD	Bit[63]	用来表示 VS、FS 以及 XS 中任意一个字段已经设置

2. sie 寄存器

sie 寄存器用来使能和关闭 S 模式下的中断，详见第 8 章。

3. stvec 寄存器

stvec 寄存器用来在 S 模式下配置异常向量表入口地址和异常访问模式，详见第 8 章。

4. scounteren 寄存器

scounteren 寄存器是一个 32 位寄存器，用来使能 U 模式下的硬件性能监测和计数寄存器，如图 1.8 所示。

31	30	29	28 … 6	5	4	3	2	1	0
HPM31	HPM30	HPM29	…	HPM5	HPM4	HPM3	IR	TM	CY
1	1	1	23	1	1	1	1	1	1

图 1.8　scounteren 寄存器

其中，字段的含义如下。

- CY 字段：使能 U 模式下的 cycle 系统寄存器。
- TM 字段：使能 U 模式下的 time 系统寄存器。
- IR 字段：使能 U 模式下的 instret 系统寄存器。
- HPM3～HPM31 字段：使能 U 模式下的 hpmcounter3～hpmcounter31 系统寄存器。

5. sscratch 寄存器

sscratch 寄存器是一个专门给 S 模式使用的临时寄存器，当处理器运行在 U 模式时，它用来保存 S 模式下进程控制块（例如，BenOS 中的 task_struct 数据结构）的指针。

在操作系统中，当一个进程从 S 模式返回 U 模式时，通常使用 sscratch 寄存器来保存该进程的 task_struct 数据结构的指针。当该进程需要重新返回 S 模式时，读取 sscratch 寄存器可以得到 task_struct 数据结构。有人认为，进程可能在 U 模式下调度，这是不正确的，一个进程要调

度，它必须返回 S 模式下的调度器里才能调度（见 17.3.8 节）。

6. sepc 寄存器

当处理器陷入 S 模式时，把中断现场或触发异常时的指令对应的虚拟地址会写入 sepc 寄存器中。

7. scause 寄存器

scause 寄存器用于保存 S 模式下的异常原因，详见第 8 章。

8. stval 寄存器

当处理器陷入 S 模式时，stval 寄存器记录了发生异常的虚拟地址。

9. sip 寄存器

sip 寄存器用来表示哪些中断处于待定（pending）状态，详见第 8 章。

10. satp 寄存器

satp 寄存器用于地址转换和保护，详见第 10 章。

1.3.5 M 模式下的系统寄存器

M 模式下的系统寄存器如表 1.7 所示。

表 1.7　　M 模式下的系统寄存器

地址	CSR 名称	属性	说明
0xF11	mvendorid	MRO	机器厂商 ID 寄存器
0xF12	marchid	MRO	体系结构 ID 寄存器
0xF13	mimpid	MRO	实现编号寄存器
0xF14	mhartid	MRO	处理器硬件线程 ID 寄存器
0xF15	mconfigptr	MRO	配置数据结构寄存器
0x300	mstatus	MRW	M 模式下的处理器状态寄存器
0x301	misa	MRW	指令集体系结构和扩展寄存器
0x302	medeleg	MRW	M 模式下的异常委托寄存器
0x303	mideleg	MRW	M 模式下的中断委托寄存器
0x304	mie	MRW	M 模式下的中断使能寄存器
0x305	mtvec	MRW	M 模式下的异常向量入口地址寄存器
0x306	mcounteren	MRW	M 模式下的计数使能寄存器
0x340	mscratch	MRW	用于异常处理的临时寄存器
0x341	mepc	MRW	M 模式下的异常模式 PC 寄存器
0x342	mcause	MRW	M 模式下的异常原因寄存器
0x343	mtval	MRW	M 模式下的异常向量寄存器
0x344	mip	MRW	M 模式下的中断待定寄存器
0x34A	mtinst	MRW	M 模式下的陷入指令（用于虚拟化）
0x34B	mtval2	MRW	M 模式下的异常向量寄存器（用于虚拟化）

接下来，介绍表 1.7 中常用的寄存器。

1. misa 寄存器

misa 寄存器用来表示处理器支持的体系结构和扩展，如图 1.9 所示。

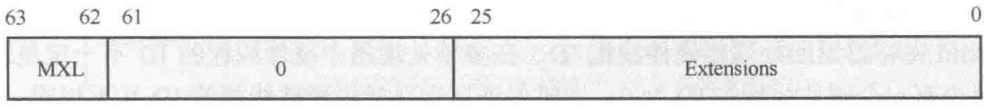


图 1.9　misa 寄存器

其中，字段的含义如下。

- Extensions：用来表示处理器支持的扩展，如表 1.8 所示。
- MXL：表示 M 模式下寄存器的长度。
  - 1：表示 32 位。
  - 2：表示 64 位。
  - 3：表示 128 位。

表 1.8　misa 寄存器支持的扩展

位	名称	说明
0	A	原子操作扩展
1	B	位操作扩展
2	C	压缩指令扩展
3	D	双精度浮点数扩展
4	E	RV32E 指令集
5	F	单精度浮点数扩展
6	G	保留
7	H	虚拟化扩展
8	I	RV32I/RV64I/RV128I 基础指令集
9	J	动态翻译语言扩展
10	K	保留
11	L	保留
12	M	整数乘/除扩展
13	N	用户中断扩展
14	O	保留
15	P	SIMD 扩展
16	Q	4 倍精度浮点数扩展
17	R	保留
18	S	支持 S 模式
19	T	保留
20	U	支持 U 模式
21	V	可伸缩矢量扩展
22	W	保留
23	X	非标准扩展
24	Y	保留
25	Z	保留

2. mvendorid 寄存器

mvendorid 寄存器是一个 32 位只读寄存器，遵循 JEDEC 制造商 ID 规范。

3. marchid 寄存器

marchid 寄存器返回处理器体系结构 ID，该 ID 由 RISC-V 基金会统一分配。

4. mimpid 寄存器

mimpid 寄存器返回处理器的实现版本 ID。

5. mhartid 寄存器

mhartid 寄存器返回处理器硬件线程 ID。在多核处理器中硬件线程的 ID 不一定是连续编号的，但至少有一个硬件线程的 ID 为 0，同时保证运行环境中硬件线程的 ID 互不相同。

6. mstatus 寄存器

mstatus 寄存器表示 M 模式下的处理器状态，如图 1.10 所示。

63	62 38	37	36	35 34	33 32	31 23	22	21	20	19	18
SD	WPRI	MBE	SBE	SXL[1:0]	UXL[1:0]	WPRI	TSR	TW	TVM	MXR	SUM
1	25	1	1	2	2	9	1	1	1	1	1

17	16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
MPRV	XS[1:0]	FS[1:0]	MPP[1:0]	VS[1:0]	SPP	MPIE	UBE	SPIE	WPRI	MIE	WPRI	SIE	WPRI
1	2	2	2	2	1	1	1	1	1	1	1	1	1

图 1.10 mstatus 寄存器

mstatus 寄存器中部分字段的含义如表 1.9 所示。

表 1.9 mstatus 寄存器中部分字段的含义

字段	位段	说明
SIE	Bit[1]	中断使能位，用来使能和关闭 S 模式下所有的中断
MIE	Bit[3]	中断使能位，用来使能和关闭 M 模式下所有的中断
SPIE	Bit[5]	中断使能保存位。当一个异常陷入 S 模式时，SIE 的值保存到 SPIE 中，SIE 设置为 0。当调用 SRET 指令返回时，从 SPIE 中恢复 SIE，然后 SPIE 设置为 1
UBE	Bit[6]	用来控制 U 模式下加载和存储指令访问内存的大小端模式。 0：小端模式。 1：大端模式
MPIE	Bit[7]	中断使能保存位。当一个异常陷入 M 模式时，MIE 的值保存到 MPIE 中，MIE 设置为 0。当调用 MRET 指令返回时，从 MPIE 中恢复 MIE，然后 MPIE 设置为 1
SPP	Bit[8]	陷入 S 模式之前 CPU 的处理模式。 0：表示从 U 模式陷入 S 模式。 1：表示在 S 模式触发的异常
VS	Bit[10:9]	用来使能可伸缩矢量扩展
MPP	Bit[12:11]	陷入 M 模式之前 CPU 的处理模式。 ❑ 0：从 U 模式陷入 M 模式。 ❑ 1：从 S 模式陷入 M 模式。 ❑ 2：在 M 模式触发的异常
FS	Bit[14:13]	用来使能浮点数单元
XS	Bit[16:15]	用来使能 U 模式下扩展的其他状态
MPRV	Bit[17]	用来修改有效特权模式。 ❑ 0：加载和存储指令按照当前的处理器模式进行地址转换与内存保护。 ❑ 1：加载和存储指令按照 MPP 字段中存储的处理器模式的权限进行内存保护与检查
SUM	Bit[18]	指定在 S 模式下是否允许访问 U 模式的内存。 ❑ 0：在 S 模式下访问 U 模式下的内存时会触发异常。 ❑ 1：在 S 模式下可以访问 U 模式下的内存
MXR	Bit[19]	指定访问内存的权限。 ❑ 0：可以加载只读页面。 ❑ 1：可以加载可读和可执行的页面
TVM	Bit[20]	支持拦截 S 模式下的虚拟内存管理操作。 ❑ 0：在 S 模式下可以正常访问 satp 系统寄存器或者执行 SFENCE.VMA/ SINVAL.VMA 指令。 ❑ 1：在 S 模式下访问 satp 系统寄存器或者执行 SFENCE.VMA/ SINVAL.VMA 指令会触发一个非法指令异常
TW	Bit[21]	支持拦截 WFI 指令。 ❑ 0：WFI 指令可以在低权限模式下执行。 ❑ 1：如果 WFI 指令以任何低特权模式执行，并且它没有在特定实现中约定的有限时间内完成，就会触发一个非法指令异常
TSR	Bit[22]	支持拦截 SRET 指令。 ❑ 0：在 S 模式下正常执行 SRET 指令。 ❑ 1：在 S 模式下执行 SRET 指令会触发一个非法指令异常

续表

字段	位段	说明
UXL	Bit[33:32]	用来表示 U 模式下寄存器的长度
SXL	Bit[35:34]	用来表示 S 模式下寄存器的长度
SBE	Bit[36]	用来控制 S 模式下加载和内存访问的大小端模式。 ❑ 0：小端模式。 ❑ 1：大端模式
MBE	Bit[37]	用来控制 M 模式下加载和内存访问的大小端模式。 ❑ 0：小端模式。 ❑ 1：大端模式
SD	Bit[63]	用来表示 VS、FS 以及 XS 中任意一个字段已经设置

7. medeleg 寄存器

medeleg 寄存器用于把异常委托到 S 模式来处理，详见第 8 章。

8. mideleg 寄存器

mideleg 寄存器用于把中断委托到 S 模式来处理，详见第 8 章。

9. mie 寄存器

mie 寄存器用来使能和关闭 M 模式下的中断，详见第 8 章。

10. mtval 寄存器

当处理器陷入 M 模式时，mtval 寄存器记录发生异常的虚拟地址。

11. mcounteren 寄存器

mcounteren 寄存器是一个 32 位寄存器，用来使能 S 模式或者 U 模式下的硬件性能监测和计数寄存器，如图 1.11 所示。

31	30	29	28 … 6	5	4	3	2	1	0
HPM31	HPM30	HPM29	…	HPM5	HPM4	HPM3	IR	TM	CY
1	1	1	23	1	1	1	1	1	1

图 1.11　mcounteren 寄存器

其中，相关字段的含义如下。

- ❑ CY 字段：使能 S 模式或者 U 模式下的 cycle 系统寄存器。
- ❑ TM 字段：使能 S 模式或者 U 模式下的 time 系统寄存器。
- ❑ IR 字段：使能 S 模式或者 U 模式下的 instret 系统寄存器。
- ❑ HPM3～HPM31 字段：使能 S 模式或者 U 模式下的 hpmcounter3～hpmcounter31 系统寄存器。

12. mscratch 寄存器

mscratch 寄存器是一个专门给 M 模式使用的临时寄存器，当处理器运行在 S 模式或者 U 模式时，它用来保存 M 模式上下文数据结构的指针，例如，在 MySBI 固件中用来保存 M 模式的栈指针（Stack Pointer，SP），在 OpenSBI 中用来保存 M 模式下的 sbi_scratch 数据结构。

13. mepc 寄存器

当处理器陷入 M 模式时，中断或遇到异常的指令的虚拟地址会写入 mepc 寄存器中。

14. mcause 寄存器

mcause 寄存器是 M 模式下的异常原因寄存器，详见第 8 章。

15. mip 寄存器

mip 寄存器用来表示哪些中断处于待定状态，详见第 8 章。

1.4 香山处理器介绍

在 2019 年，中国科学院计算技术研究所发起了“香山” 高性能开源 RISC-V 处理器项目，研发出了目前国际上性能最高的开源高性能 RISC-V 处理器核之一 ——“香山”，并成为国际上最受关注的开源硬件项目之一，得到国内外企业的广泛支持。

香山开发团队致力于研究面向世界的体系结构创新开源平台，满足工业界、学术界、个人爱好者等的体系结构研究需求，探索高性能处理器的敏捷开发流程，建立一套基于开源工具的高性能处理器设计、实现、验证流程，大幅提高处理器开发效率，降低处理器开发门槛。

目前香山处理器开发团队已经发布了两个版本的处理器体系结构核心。

- 香山处理器第一版（雁栖湖体系结构）：支持 RV64GC 指令集，已在 2021 年 7 月投片成功，在 28 nm 的工艺节点下达到 1.3 GHz 的频率。
- 香山处理器第二版（南湖体系结构）：支持 RV64GCBK 指令集，已完成 RTL 代码冻结和后端设计验证流程，计划在 14 nm 的工艺节点下频率达到 2 GHz。

本节介绍的香山处理器是采用南湖体系结构的版本。香山处理器采用 Chisel 语言进行开发，并且采用宽松的开源协议——木兰宽松许可证第 2 版。

1.4.1 香山处理器体系结构

香山处理器具有如下特性。

- 支持 RV64GCBK 指令集。
- 采用超标量处理器设计，乱序六发射结构设计。
- 支持分支预测、指令缓冲、顺序取指、译码、重命名、重定序缓冲、保留站、整数/浮点数寄存器堆、整型/浮点运算等硬件单元。
- 具有 L1 指令 TLB（Translation Lookaside Buffer）：支持 32 个 4 KB 页面的表项以及 8 个大页表项。
- 具有 L1 数据 TLB：支持 128 个 4 KB 页面的表项以及 8 个 2 MB/1 GB 大页表项，图 1.12 中简称 DTLB。
- 具有 L2 TLB&PTW：支持 2048 个表项。
- 具有 128 KB 的 L1 指令高速缓存以及 128 KB 的 L1 数据高速缓存。
- 具有 1 MB 大小的 L2 高速缓存，采用 8 路组相联映射方式。
- 具有 6 MB 大小的 L3 高速缓存，采用 6 路组相联映射方式。

香山处理器体系结构如图 1.12 所示。它主要分成前端、后端以及访存子系统。

- 前端子系统：包括分支预测单元、取指目标队列、指令预取单元、指令缓冲区等硬件单元。
- 后端子系统：包括译码单元、重命名单元、重定序缓冲区、保留站、整数/浮点数寄存器堆、整型/浮点运算单元等。
- 访存子系统：包括两条加载流水线、两条存储地址流水线、两条存储数据流水线，以及独立的加载队列、存储队列、存储缓冲区、各级存储单元等。存储单元包括 L1 指令高速缓存、L1 数据高速缓存、L2/L3 高速缓存、TLB 以及 BOP（Best Offset Prefectcher，最佳偏移量预取器）等硬件单元。

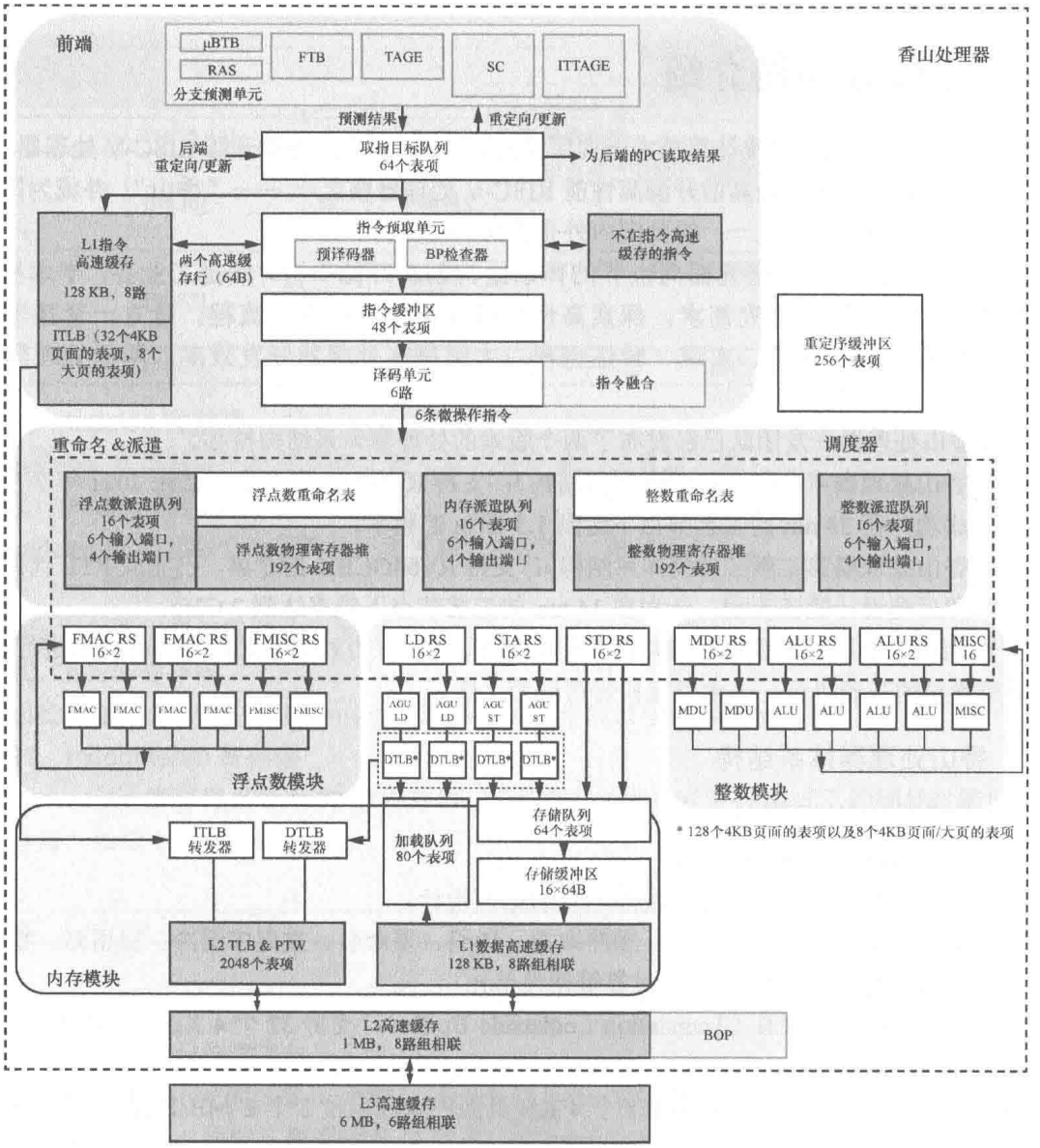


图 1.12　香山处理器体系结构

1.4.2　香山处理器的前端子系统

香山处理器的前端子系统采用一种分支预测和指令缓存解耦的取指体系结构，如图 1.13 所示。分支预测单元响应取指令请求。取指令请求被写入取指目标队列中，然后发送到指令预取单元。从指令高速缓存取出的指令首先通过预译码器的初步检查，如发现分支预测错误，则冲刷（flush）预测流水线。通过检查后的指令送到指令缓冲区以及译码模块，最终形成后端的指令流。

1. 分支预测单元

在超标量处理器中，使用流水线提高性能。在取指令阶段，处理器不仅要从指令高速缓存中取出多条指令，还需要决定下一个周期取指令的地址。当遇到条件跳转指令时，取指模块不能确定是否需要跳转。处理器会使用分支预测单元试图猜测每条跳转指令是否会执行。当它猜测的准确率很高时，流水线中充满了指令，以实现高性能。当它猜测错误时，处理器会丢弃为这次猜测所做的所有工作，然后从正确的分支位置取指令和填充流水线。因此，一次错误的分支预测会导致严重的惩罚，并使程序性能下降。通常现代处理器的分支预测准确率都很高。

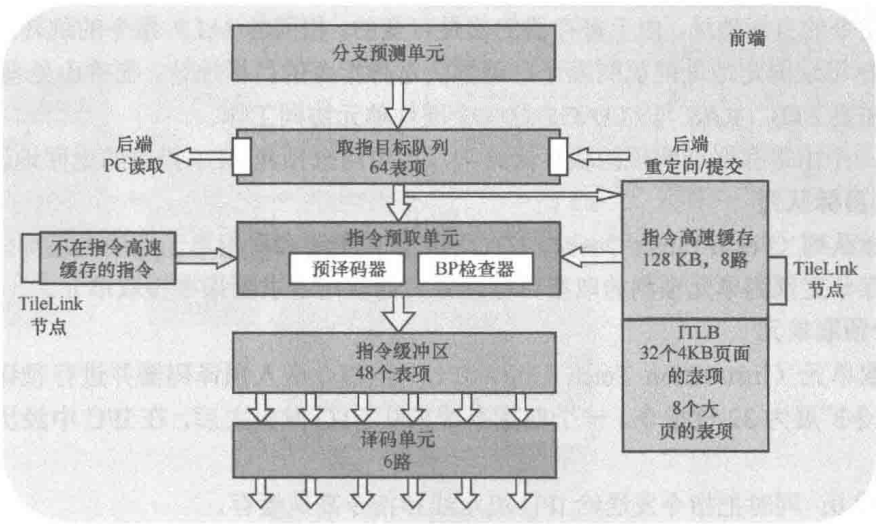


图 1.13 香山处理器的前端子系统

分支预测一般有两种情况。

- 预测方向。一个分支语句通常有两个可能方向，一是发生跳转，二是不发生跳转。对于条件分支语句（如 if 语句），这就意味着要预测是否需要执行分支语句。
- 预测目标地址。如果分支指令的方向是发生跳转，那么需要知道它跳转到哪里，即跳转的目标地址。根据目标地址，跳转又分成两种形式。
  - 直接跳转。在指令中使用立即数的形式给出一个相对于 PC 的偏移量值。这种形式的跳转容易预测，因为立即数一般是固定的。
  - 间接跳转。分支指令的目标地址是来自一个通用寄存器的值，并且是其他指令的计算结果，这会导致需要等到流水线的执行阶段才能确定目标地址，加大了分支预测失败时惩罚的概率。大部分程序中使用的间接跳转指令是用于调用子程序的 call/return 类型的指令，这类指令有着很强的规律性，容易预测。

香山处理器的分支预测单元采用一种多级混合预测的体系结构，其主要组成部分包括下一行预测器（Next Line Predictor，NLP）和精确预测器（Accurate PreDictor，APD）。

NLP 旨在用较小的存储开销提供一个无空泡的快速预测流。它的功能主要由微型分支目标缓冲器（Micro-Branch Target Buffer，μBTB）提供。对于给定的起始地址 PC，μBTB 对从 PC 开始的一个预测块做出整体预测。

μBTB 利用分支历史和 PC 的低位异或索引存储表，根据从表中读出的内容直接提供最精简的预测，包括下一个预测块的起始地址、这个预测块是否发生分支指令跳转、跳转指令（如果发生跳转）相对于起始地址的偏移量等信息。另外，它还提供分支指令的相关信息以更新分支历史，包括是否在条件分支跳转，以及块内包含的分支指令的数目。

为提高总体预测准确率，减少预测错误带来的流水线冲刷，香山处理器的 APD 模块包括取指目标缓冲区（Fetch Target Buffer，FTB）、条件分支方向预测器[①]、间接跳转预测器（位于分支预测单元中）和返回地址栈（Return Address Stack，RAS）。

FTB 是分支预测的核心部件，它除提供预测块内分支指令的信息之外，还提供预测块的结束地址。TAGE-SC 是条件分支的主预测器。ITTAGE 是一种准确率很高的间接跳转预测器，主要用于 RISC-V 中的间接跳转指令 JALR。JALR 指令支持以寄存器取值加立即数的方式来指定

① 包括 TAGE（TAgged GEometric history length predictor，标记几何历史长度预测器）和统计校正器（Statistical Corrector，SC）。

无条件跳转指令的目标地址。由于寄存器的值是可变的，相同的 JALR 指令的跳转地址可能不相同，因此 FTB 记录固定地址的机制难于准确预测这种指令的目标地址。在香山处理器中，JALR 指令的预测需要 FTB、RAS 与 ITTAGE 这 3 个硬件单元协同工作。

RAS 是一个由寄存器堆实现的栈存储结构，主要用来预测 RET 指令的返回地址。

2. 取指目标队列

取指目标队列（Fetch Target Queue，FTQ）是分支预测和取指单元之间的缓冲队列，它的主要职能是暂存分支预测单元预测的取指目标，并发送取指请求到指令预取单元。

3. 指令预取单元

指令预取单元（Instruction Fetch Unit，IFU）把指令送入预译码器并进行预译码，同时将 16 位压缩指令扩展为 32 位指令。一个取指令请求从 FTQ 发出之后，在 IFU 中经历了下面 4 个阶段。

- 阶段 0：同时把指令发送给 IFU 流水线和指令高速缓存。
- 阶段 1：对指令做简单计算。
- 阶段 2：等到指令高速缓存返回最多两个高速缓存行的数据后，先做指令切分，并把取指令地址之外的指令抛弃，然后送入预译码器并进行预译码。另外，把 16 位压缩指令扩展为 32 位指令。
- 阶段 3：将预译码结果发送到分支预测检查器并检查。如果发现错误，就会在下一节拍中刷新 IFU 流水线并把信息发送给 FTQ，让其刷新分支预测器并重新取指令。如果未发现错误，则缓存在指令缓冲单元里等待译码。

4. 指令缓冲区

指令缓冲区（instruction buffer）的工作流程包括如下 3 个阶段。

- 阶段 0：把 FTQ 取指令请求的地址发送到指令高速缓存以及 ITLB 中。由于指令缓冲区采用的是虚拟索引物理标记（Virtual Index Physical Tag，VIPT）的映射方式，因此需要在比较标记（tag）之前将虚拟地址翻译为物理地址。
- 阶段 1：判断指令高速缓存是否命中。
- 阶段 2：如果指令高速缓存命中，则直接返回数据给 IFU。如果指令高速缓存未命中，暂停流水线，并将请求发送给缺失处理单元。缺失处理单元会向 L2 高速缓存发送请求。等缺失处理单元回填完成后将数据返回给 IFU。这个阶段还会把 L1 指令 TLB 翻译出的物理地址发送给 PMP（Physical Memory Protection，物理内存保护）模块并进行访问权限的查询。如果权限错误会触发指令访问异常。

在指令缓冲区中产生的异常主要包括两种：

- ITLB 报告的页面权限不足或者指令缺页异常（instruction page fault）；
- ITLB 以及 PMP 报告的指令访问异常（instruction access fault）。

这些异常信息直接报告给 IFU，而请求的数据则被视为无效数据。

5. 译码单元

指令从指令高速缓存中取出后，暂存在指令缓冲区里，然后以每个周期 6 条指令的速度送入译码单元并进行译码。

1.4.3　香山处理器的后端子系统

香山处理器的后端子系统主要负责指令的重命名（rename）与乱序执行。如图 1.14 所示，后端模块可以分成控制模块、整数模块、浮点数模块、访存子系统。

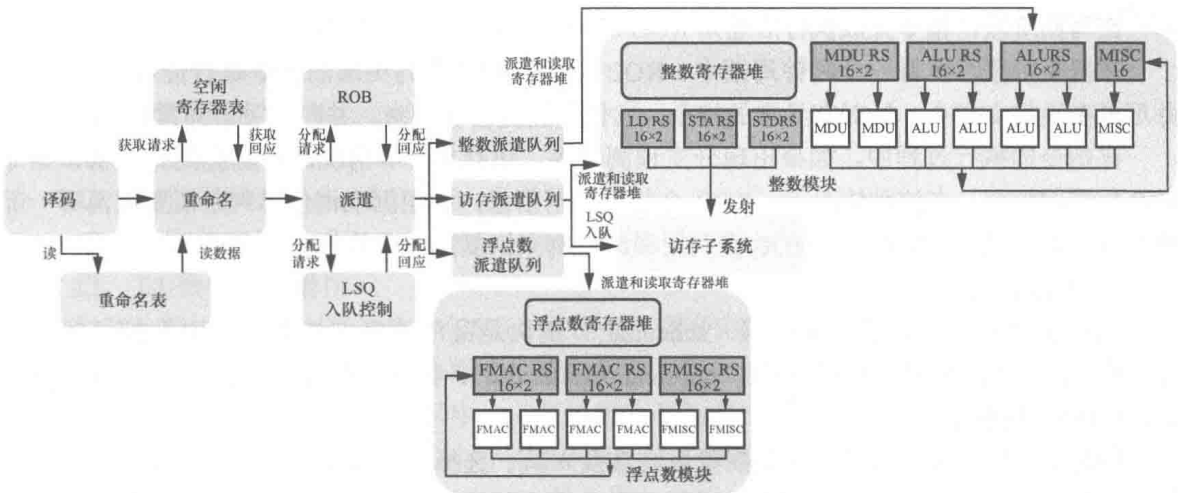


图 1.14 香山处理器的后端子系统

1. 重命名模块

在乱序处理器中，重命名模块负责管理和维护逻辑寄存器与物理寄存器之间的映射，通过对逻辑寄存器的重命名，实现指令间依赖的消除，并完成乱序调度。重命名模块包括如下 3 个子模块。

- 重命名的流水线级：更新指令的物理寄存器信息。由于指令间依赖的存在，对于在同一时钟节拍进行重命名的指令，重命名模块需要负责在它们之间进行物理寄存器号的旁路，如当指令 1 需要使用指令 0 的结果时，指令 1 的对应源操作数需要来自指令 0 的目的寄存器。
- 空闲寄存器列表：记录所有的空闲寄存器状态，本质上是一个队列，由入队指针、出队指针和队列存储组成。
- 重命名表：维护逻辑寄存器与物理寄存器之间的映射关系。重命名表内部包含 32 个寄存器，它们的宽度与物理寄存器的宽度相同，分别代表 32 个逻辑寄存器对应的物理寄存器地址。目前，整数和浮点数寄存器堆各有一套重命名表，根据未来的指令扩展情况，重命名表支持记录更多内容的模式（如条件寄存器堆、向量寄存器堆等重命名状态）。

2. 派遣模块

在香山处理器中，派遣（dispatch）模块的工作流程包括两级流水线。

- 第一级负责将指令分类并发送至整数、浮点数与访存这 3 类派遣队列（dispatch queue）中。在发送之前需要对指令的类型、是否能够进入下一级进行判断。在香山处理器中，出于时序的考虑简化了指令是否可以继续进入下一级的条件。目前，当且仅当所有资源都是充足的（如派遣队列有足够空闲表项、重定序缓冲区（Re-Order Buffer，ROB）有足够表项等）时，指令才能够进入下一级。整型指令可能会因为浮点派遣队列满而被阻塞。
- 第二级负责将对应类型的指令根据运算操作类型派遣至不同的保留站中。

3. 重定序缓冲区

在乱序处理器中，ROB 的作用是给指令定序，使程序正常执行的结果能够保留下来。按照一条指令的执行流程，ROB 会依次影响指令的派遣、写回、提交等流程，同时指令还可能在任何时刻被冲刷和丢弃掉。

在派遣阶段，为指令分配一个 ROB 表项，并将一些需要保存的信息存入 ROB 表项中，这些信息包括指令的重命名信息、类型信息、对应的 FTQ 指针等。在指令进入 ROB 之后，会更新一些状态位（如有效位等）。

在写回阶段，指令完成执行后通知 ROB 对应的运算操作已经完成。

在提交阶段，在每一个时钟周期中，ROB 会依次检查队列头部的指令是否能够正常提交，并尽量多地提交指令。针对有异常的指令，它们的提交会被阻塞，并向外发出异常信息。

在指令的执行过程中，如果出现分支预测错误、访存异常等情况，可能会需要冲刷该指令以及后续的指令。在这种情况下，ROB 会收到取消信息，并根据取消信息判断需要取消哪一部分指令。对于取消掉的指令，ROB 会利用回滚机制来恢复重命名等信息。

4. 乱序调度器

在香山处理器中，乱序调度器（scheduler）模块是乱序调度的核心，其中维护了保留站（reservation station）、物理寄存器堆（physical register file）、物理寄存器状态表（physical register status table）等模块。

保留站也称为发射队列，在发射阶段它涉及入队、选择、读数据、出队等操作，同时保留站还需要负责对指令写回的监听以及对等待指令的唤醒等操作。香山处理器目前采用了发射前读寄存器堆的设计方案。

5. 执行单元

在香山处理器中，执行单元包括整数和浮点数执行单元。整数执行单元包括如下部件。

- 算术逻辑单元（Arithmetic Logic Unit，ALU）：支持加、减、逻辑、分支、部分位扩展等指令。
- 乘法（MUL）单元：香山处理器的乘法器默认为 3 级流水线的华莱士树乘法器。
- 除法（DIV）单元：香山处理器使用 SRT16 定点除法器。

香山处理器还实现了对 CSR 和 JUMP 指令的支持。

1.4.4　香山处理器的访存子系统

香山处理器的访存子系统如图 1.15 所示。

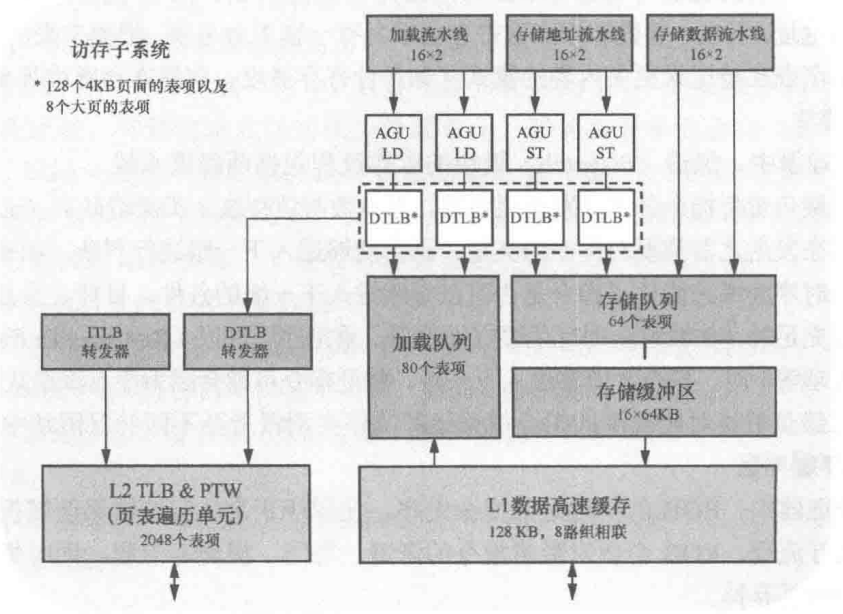


图 1.15　访存子系统

访存子系统包括如下部件。

- 两条加载流水线；

- 两条存储地址（STA）流水线；
- 两条存储数据（STD）流水线；
- 加载队列（load queue）；
- 存储队列（store queue）；
- 存储缓冲区（store buffer）；
- 内存管理单元（Memory Management Unit，MMU）；
- L1 数据高速缓存。

加载队列的作用是，当加载指令在 L1 数据高速缓存中缺失时负责监听后续的重填结果并执行写回操作。存储队列的作用是在指令提交之前暂存存储指令的数据。在存储指令提交之后，存储指令把数据搬运到存储缓冲区中，并且以缓存行为单位对存储操作的写请求进行合并。当存储缓冲区接近满时将合并后的多个存储请求写入 L1 数据高速缓存中。L1 数据高速缓存采用 TileLink 缓存一致性协议。

MMU 负责将虚拟地址翻译成物理地址，然后用物理地址访问内存，同时会进行地址权限检查，如检查是否具有可写、可执行权限等。在香山处理器中，MMU 包括 TLB、L2 TLB、TLB 转发器、PMP（Physical Memory Protection，物理内存保护）和 PMA（Physical Memory Attribute，物理内存属性）等部件。

1. 加载流水线

香山处理器中包含两条加载流水线，其工作流程包含阶段 0、阶段 1、阶段 2 和阶段 3。

在阶段 0，完成如下工作。

（1）从保留站读出指令和操作数。

（2）加法器将操作数与立即数相加，计算虚拟地址。

（3）把虚拟地址送入 TLB 并进行查询。

（4）把虚拟地址送到 L1 数据高速缓存。

在阶段 1，完成如下工作。

（1）TLB 输出物理地址。

（2）完成快速异常检查。

（3）使用虚拟地址在 L1 数据高速缓存中进行索引。

（4）物理地址被送进 L1 数据高速缓存并进行物理标记比较。

（5）把虚拟/物理地址送进存储队列/存储缓冲区。

（6）根据 L1 数据高速缓存返回的命中向量以及初步异常，判断结果。产生提前唤醒信号并发送给保留站。如果在这一级就出现了会导致保留站重发的事件，则通知保留站这条指令需要重发。

在阶段 2，完成如下工作。

（1）完成异常检查。

（2）根据 L1 高速缓存返回的结果，选择数据。

（3）根据加载指令的要求，对返回的结果做裁剪操作。

（4）更新加载队列中对应项的状态。

（5）把结果（整数）写回公共数据总线。

（6）把结果（浮点数）送到浮点模块。

在阶段 3，完成如下工作。

（1）根据 L1 数据高速缓存的反馈结果，更新加载队列中对应项的状态。

（2）根据 L1 数据高速缓存的反馈结果，输出到保留站，通知保留站这条指令是否需要重发。

2. 存储流水线

香山处理器采用数据和地址分离的存储流水线，包括两条 STA 流水线和两条 STD 流水线。每条 STA 流水线分成 4 个流水级。前两个流水级负责将存储指令的控制信息和地址传递给存储队列，后两个流水级负责等待访存依赖检查完成。每条 STD 流水线从保留站取出存储的数据，写入存储队列的对应项中。

STA 流水线包括阶段 0、阶段 1、阶段 2 和阶段 3。

在阶段 0，完成如下工作。

（1）计算虚拟地址。

（2）把虚拟地址送入 TLB。

在阶段 1，完成如下工作。

（1）查询 TLB，输出物理地址。

（2）快速检查异常。

（3）检查访存依赖。

（4）把物理地址送入存储队列。

在阶段 2，完成如下工作。

（1）检查访存依赖。

（2）完成全部异常检查和 PMA 查询，根据结果更新存储队列。

在阶段 3，完成如下工作。

（1）完成访存依赖检查。

（2）通知 ROB 可以提交指令。

STD 流水线完成如下工作。

（1）从保留站取出待存储数据。

（2）将存储数据写入存储队列。

3. 加载队列

加载队列是离 L1 数据高速缓存最近的硬件单元。在加载流水线中，已经把数据请求发送到 L1 数据高速缓存并把状态信息更新到加载队列，加载队列的作用是监听 L1 数据高速缓存的缺失回填结果并执行写回操作。香山处理器的加载队列是一个具有 80 个表项的循环队列，每个表项包括：

- 物理地址；
- 虚拟地址；
- 回填数据项；
- 状态位。

4. 存储队列

存储队列的作用是在指令提交之前暂存存储指令的数据，然后写入 L1 数据高速缓存中。存储队列是一个具有 64 个表项的循环队列，每个表项包括：

- 物理地址；
- 虚拟地址；
- 数据；
- 数据有效掩码；
- 状态位。

一条存储地址指令在 STA 流水线的阶段 1 和阶段 2 中写入存储队列中。一条存储数据指令

从保留站发出后会立即写入存储队列中。

在指令提交后，ROB 根据存储指令提交的数量发送信号给存储队列，并通知这些存储指令已经写入存储队列了。存储队列收到信号后，把这些指令按顺序写入存储缓冲区。

5. 存储缓冲区

在香山处理器中，存储缓冲区由 16 个表项组成，每个表项以高速缓存行为单位组织数据，每个表项包括：

- 虚拟地址；
- 物理地址；
- 数据；
- 状态位；
- 数据有效掩码。

从存储队列向存储缓冲区写入的过程如下。

（1）从存储队列中接收存储指令，读取数据和地址。

（2）与存储缓冲区已有的地址进行对比。如果地址不匹配，则分配新表项并写入；如果地址匹配，则合并成一个表项并写入已有表项中。如果地址匹配但正在写入 L1 数据高速缓存，则阻塞等待。如果没有空闲表项，则阻塞等待。

从存储缓冲区写入 L1 数据高速缓存的过程如下。

（1）选出需要写入的表项，读出数据，更新表项的状态位。

（2）发送写入请求到 L1 数据高速缓存中，并等待 L1 数据高速缓存写入结果。

6. MMU

为了实现进程隔离，每个进程都会有自己的地址空间，使用的地址都是虚拟地址。MMU 的作用是将虚拟地址翻译成物理地址，然后用这个物理地址访问内存，同时会进行权限检查，如检查是否具有可写、可执行等权限。

香山处理器目前仅支持 Sv39 分页机制，即虚拟地址的有效位宽为 39 位，实现三级页表，这意味着遍历页表需要 3 次内存访问。香山处理器的 MMU 由 ITLB、DTLB、L2 TLB、TLB 转发器（TLB repeater）、PMP 和 PMA 等硬件单元组成。

当 ITLB 和 DTLB 发生未命中时会发送请求给 L2 TLB。如果 L2 TLB 也发生未命中，则通过页表遍历单元（page table walker）访问内存中的页表。L2 TLB 主要考虑如何提高并行度并过滤重复的请求。TLB 转发器是 L1 TLB 到 L2 TLB 的请求缓冲区。PMP 和 PMA 需要对所有的物理地址访问进行权限检查。

SFENCE.VMA 指令执行时会先清空存储缓冲区的全部内容（即写回 L1 数据高速缓存中），然后发出刷新信号到 MMU 的各个部件。指令最后会刷新整个流水线，从取指开始重新执行。SFENCE.VMA 指令会取消所有正在处理的请求，包括 TLB 转发器和过滤器以及 L2 TLB 中正在处理的请求，并且根据地址和地址空间标识符（Address Space IDentifier，ASID）刷新 TLB 和 L2 TLB 中缓存的页表。全相联部分根据是否命中进行刷新，组相联或者直接相联部分根据索引刷新。

在香山处理器中添加了对 ASID 的支持，默认长度为 16 位，也可以参数化配置。

在香山处理器中采用触发缺页异常（page fault）的方式报告页表的 A（access）和 D（dirty）位的更新状态，最终由软件进行更新。

7. TLB

在香山处理器中，TLB 的配置如下。

- ITLB 支持 32 个 4 KB 页面的表项和 4 个大页表项，采用全相联映射方式，以及伪 LRU

替换策略。

- DTLB 支持 128 个 4 KB 页面的表项和 8 个大页表项，其中 4KB 页面采用直接相联的映射方式，大页采用全相联映射方式，以及伪 LRU 替换策略。
- L2 TLB 为更大的页表缓存，支持 2048 个表项。

L1 TLB 包括前端的 ITLB 以及后端的 DTLB。前端取指令时对 ITLB 的访问方式为阻塞式，即当 TLB 未命中时，TLB 不立即返回结果，而进行页表遍历，取回页表项后返回。而后端访存模块对 DTLB 的访问方式为非阻塞式，即当 TLB 发生未命中时，TLB 也需要立即返回结果，无论是命中还是未命中。

L2 TLB 是更大容量的页表缓存，它包括如下几个部件。

- 页表缓存（page cache）：用于分别缓存 Sv39 中的三级页表。
- 页表遍历单元（page walker）：遍历内存中的页表。
- 未命中队列（miss queue）：查询页表缓存中的未命中请求。
- 预取器（prefetcher）。

从 L1 TLB 发来的请求会先访问页表缓存。如果命中，直接返回结果给 L1 TLB。如果未命中，则进入页表遍历单元或者未命中队列。如果在页表遍历单元中，则遍历页表。如果在未命中队列中，要么再次访问页表缓存，要么访问内存中的页表。访问页表结束后，直接返回结果给 L1 TLB。预取器会监控页表缓存的查询结果，产生预取请求，预取的结果会存到页表缓存中，不会返回给 L1 TLB。

页表缓存分别缓存了三级页表的表项。根据虚拟地址，判断是否命中，获得最靠近叶子节点的结果。

页表遍历单元根据虚拟地址访问页表，访存的结果会存到页表缓存中。它同一时刻只能够处理一个请求，并且最多只访问前两级页表。当访问到 2MB 大小的页表节点时，返回给未命中队列，由未命中队列进行最后一级页表的访问，因此整个页表遍历过程需要与未命中队列配合。

未命中队列的目的是提高 L2 TLB 的访存能力。未命中队列接收来自页表缓存的未命中请求以及页表遍历单元的结果以访问最后一级页表。未命中队列和页表遍历单元分工合作，共同完成页表遍历的全过程。

预取器监控页表缓存的查询结果，如果未命中或者命中预取项，预取器产生预取请求。预取请求会发给未命中队列但是不会发送给页表遍历单元。把预取结果存到页表缓存中，不返回给 L1 TLB。

8. L1 数据高速缓存

在香山处理器体系结构中，L1 数据高速缓存与访存流水线紧耦合，对外通过 TileLink 总线协议和 L2 高速缓存直接交互。L1 数据高速缓存大小为 128 KB，采用 8 路组相联映射方式以及伪 LRU 替换策略。L1 数据高速缓存包含的模块如表 1.10 所示。

表 1.10　L1 数据高速缓存包含的模块

模块	说明
加载流水线（load pipeline）	与加载访存流水线紧耦合
主流水线（main pipeline）	处理存储、探测、原子指令和替换操作
回填流水线（refill pipeline）	负责将 L2 高速缓存的回填数据写回 L1 数据高速缓存
原子单元（atomics unit）	调度原子请求
未命中队列（miss queue）	向 L2 高速缓存请求缺失的块
探测队列（probe queue）	接收 L2 高速缓存的一致性请求
回写队列（writeback queue）	负责将替换块写回 L2 高速缓存或应答探测请求

与 L1 数据高速缓存相关的请求有加载请求、存储请求、探测请求以及原子操作请求。

加载请求的处理流程如下。

（1）如果在 L1 数据高速缓存中命中来自加载队列的加载请求，则直接返回；如果未命中，则进入未命中队列。

（2）在未命中队列中分配一项空闲的表项，并记录相关信息。

（3）如果需要替换缓存行，则向主流水线发送替换（replace）请求。

（4）发送替换请求的同时向 L2 高速缓存发送获取（acquire）请求。

（5）等待 L2 高速缓存返回数据。

（6）向回填流水线发送回填请求，把数据回填到 L1 数据高速缓存。

（7）把回填数据转发给加载队列。

（8）如果缓存行需要替换，则在回写队列中把替换的缓冲行回写到 L2 高速缓存中。

存储请求主要处理来自存储缓冲区的请求，在主流水线中访问 L1 数据高速缓存。

- 如果命中，直接写入 L1 数据高速缓存。
- 如果未命中并且未命中队列已满，存储缓冲区会稍后重复该请求。
- 如果未命中并且未命中队列成功接收了该请求，则由未命中队列继续处理该请求。在完成后通知存储缓冲区，并通过回填流水线更新 L1 数据高速缓存。

探测请求主要接收来自 L2 高速缓存的请求。

（1）接收来自 L2 高速缓存的探测请求。

（2）在主流水线中修改已探测的缓存行权限。

（3）返回应答，同时回写脏数据。

在香山处理器中，对原子操作指令的处理是在 L1 数据高速缓存中完成的，它主要在主流水线执行原子指令。

（1）如果在主流水线中发现原子指令未命中，则会向未命中队列发起请求。

（2）未命中队列从 L2 高速缓存中取得数据，执行原子指令。

（3）如果原子指令执行时未命中队列已满，则不断尝试重发原子指令，直到可以接收未命中的回填请求为止。

上述过程保证原子指令的目标地址已经在 L1 数据高速缓存中。LR 指令会在主流水线中注册一个保留集（reservation set），记录 LR 的地址，并阻塞对该地址的探测请求。为了避免带来死锁，主流水线会在等待 SC 指令一定时间后不再阻塞探测请求，此时再收到的 SC 指令则均被视为失败的。因此，在等待 SC 指令配对的时间里，需要阻塞这个该地址的探测请求。

1.4.5 香山处理器的 L2/L3 高速缓存

香山处理器的 L2/L3 高速缓存采用“非严格包含”（non-inclusive）的设计方案，具体来说就是目录是“包含目录”（inclusive directory），而数据是“非严格包含数据”（non-inclusive data）。这里说的目录指的是元数据和标记等。

在传统的多级高速缓存设计中，在考虑设计难度的情况下，通常的设计方案是使多级高速缓存都有相同数据的备份，即采用“包含”的设计方案。随着处理器内核数量增多，L3 高速缓存的容量必须远远大于 L1 与 L2 高速缓存的总容量才有意义。因此现在的设计大多采用“非严格包含”的设计方案，即各级高速缓存不会同时保存着数据的备份，下级的高速缓存是上级高速缓存的回收站。例如，当 L1 高速缓存的缓存行被替换时会写入 L2 高速缓存中。

L2/L3 高速缓存的内部结构如图 1.16 所示。

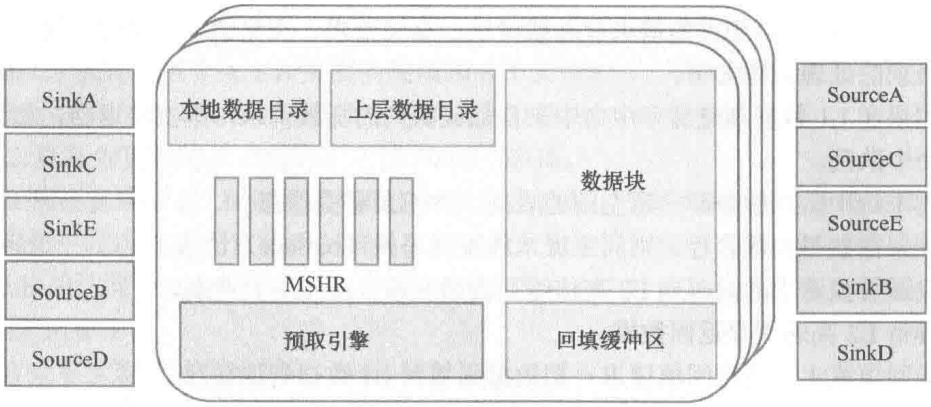


图 1.16　L2/L3 高速缓存的内部结构

- 根据请求的地址的索引来分片（slice）以提升并发度。每个片内部的 MSHR（Miss Status Holding Register，未命中状态保存寄存器）数量可配，负责具体的任务管理。
- 数据块（data bank）：负责存储具体数据，可以通过参数配置块数量从而提升读写并行度。
- 回填缓冲区（refill buffer）：负责暂存回填的数据，直接透传到上层高速缓存而不需要经过静态随机存储器（Static Random Access Memory，SRAM）。
- Sink/Source：与 TileLink 协议相关的通道控制，负责与 TileLink 接口进行交互，一方面将外部请求转换为高速缓存内部信号，另一方面接收高速缓存内部请求并转换为 TileLink 请求，发送到接口上。
- 本地数据目录（self directory）：当前高速缓存所对应的目录。
- 上层数据目录（client directory）：上一级高速缓存对应的目录。
- 预取引擎（prefetch engine）：预取数据。

L2/L3 高速缓存总体工作流程如下。

（1）通道控制模块接收 TileLink 请求，将其转换为高速缓存内部请求。

（2）MSHR 为内部请求分配一个 MSHR 表项。

（3）MSHR 根据请求发起不同的任务，任务类型包括数据读写、向上下层高速缓存发送新请求或返回响应以及触发或更新预取器等。

（4）当一个请求所需的全部操作在 MSHR 中完成时，MSHR 被释放，等待接收新的请求。

1. 目录设计

除数据之外，高速缓存还包括目录。目录包括元数据和标记等信息。在香山处理器的多级高速缓存设计中，目录数据采用包含设计方案，而数据采用非严格包含设计方案。在结构组织上将上层数据的目录与本地数据的目录分开存储，两者的结构类似。在 VIPT 映射机制的高速缓存中，目录中的每个表项是按照高速缓存行的物理地址进行记录的。

本地数据目录中的每一项如下。

- state：保存数据块的权限。
- dirty：表示数据块是否脏的。
- clientStates：保存数据块在上层的权限情况。
- prefetch：指示数据块是否是预取的。

上层数据目录中的每一项如下。

- state：保存上层数据块的权限。
- alias：保存上层数据块的虚地址末位，用于修复高速缓存别名问题。

2. MSHR

MSHR 的作用是，当接收到上层高速缓存发来的获取/释放（acquire/release）请求或者接收到下层高速缓存的探测请求时，会为该请求分配一个 MSHR 表项，同时通过读取目录获取该地址在本层及以上一级高速缓存中的权限信息。MSHR 根据这些信息决定以下内容。

- 权限控制：表示如何更新本地数据目录和上层数据目录中的权限。
- 请求控制：表示是否需要向上层、下层高速缓存发送子请求，并等待这些子请求的响应。

3. 高速缓存别名问题

在香山处理器中，L1 数据高速缓存和 L1 指令高速缓存的大小均为 128 KB，采用 8 路组相联的映射方式。每路（way）的大小为 16 KB，索引域的范围为 Bit[13:0]，这已经超过了 4 KB 页面偏移量，其中页面着色（page color）可以取整数 0～3，如图 1.17 所示，因此产生了 VIPT 别名问题。关于 VIPT 别名问题产生的原因，详见 11.6.3 节。

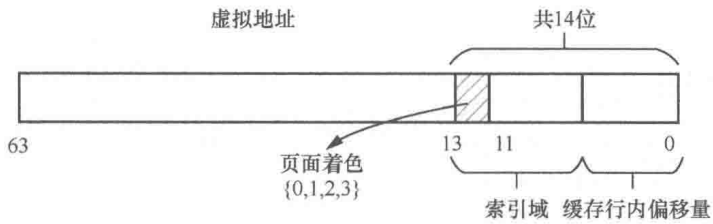


图 1.17 索引域

RISC-V 体系结构规范约定处理器不允许出现类似于高速缓存别名（aliasing）的缓存一致性问题，因此香山处理器的高速缓存系统采用硬件方式解决 VIPT 别名问题。具体的解决方式是由 L2 高速缓存保证一个物理缓存行在上层高速缓存中最多只有一种别名位，并且回填正确的数据到上层高速缓存行中。

假设虚拟页面 Page_0 和虚拟页面 Page_1 同时映射到物理页面 Page_P 上，其中虚拟页面 Page_0 的起始地址为 0x0，虚拟页面 Page_1 的起始地址为 0x1000，如图 1.18 所示。

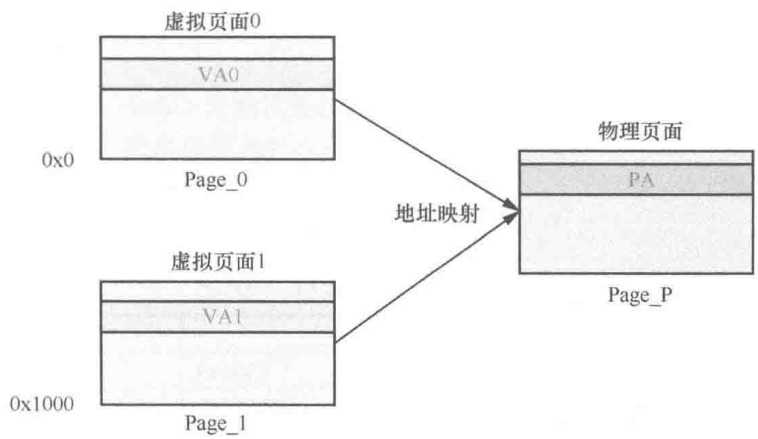


图 1.18 两个虚拟页面同时映射到物理页面

L2 高速缓存解决 VIPT 别名问题的过程如图 1.19 所示。需要注意的是，在香山处理器中，目录中的每个表项是按照高速缓存行的物理地址进行记录的。

- 若 L1 数据高速缓存访问虚拟页面 Page_1 时发生未命中，向 L2 高速缓存发起获取请求。在这个请求中记录的别名为 0x1。

- 若 L2 高速缓存命中，但是发现获取请求中的别名（0x1）和目录中的别名（0x0）不同，L2 高速缓存发起探测请求，探测请求中 data 域记录的别名为 0x0。

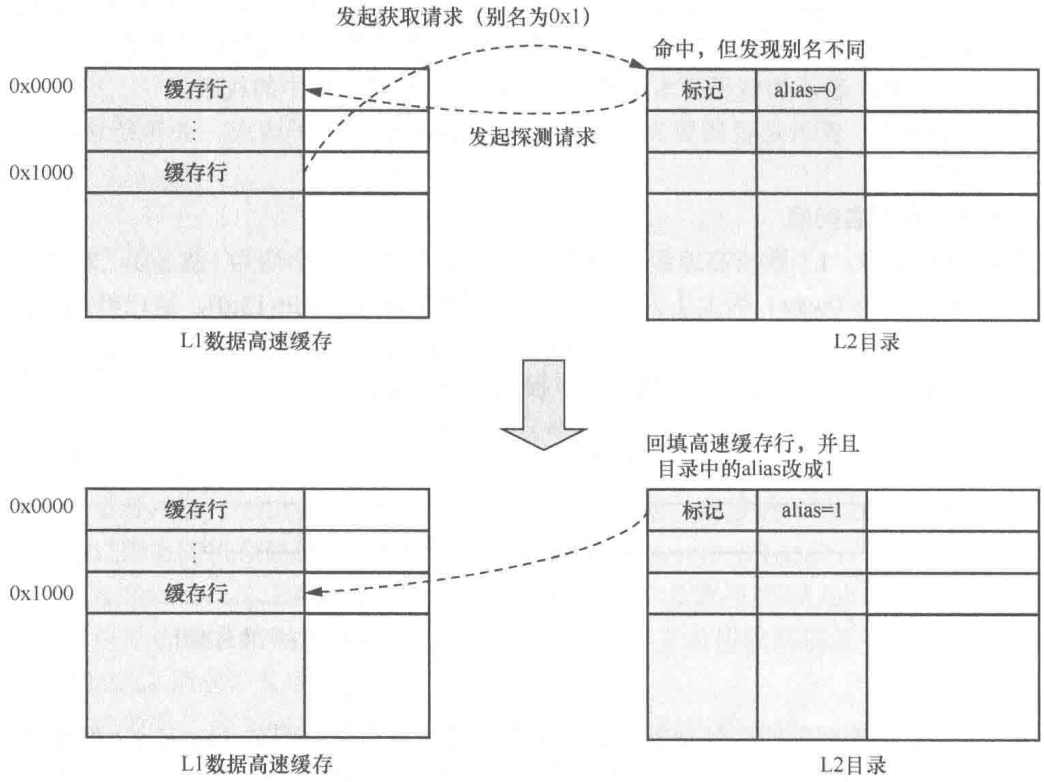


图 1.19　别名修复

侦测请求完成之后，L2 高速缓存把数据返回给虚拟页面 Page_1 对应的 L1 数据高速缓存，然后在 L2 高速缓存中的上层数据目录把对应表项的别名改成 0x1。

第 2 章　搭建 RISC-V 实验环境

本书中大部分实验是基于 BenOS 的。BenOS 是一个简单的小型操作系统实验平台，可以在 QEMU 或者在香山采用的指令集模拟器（NEMU）上运行。我们可以通过在最简单的裸机程序上慢慢添加功能，实现一个具有任务调度功能的小型操作系统，通过动手实践加深对 RISC-V 体系结构的理解。

2.1 实验平台

2.1.1 QEMU

QEMU Virt 实验平台（是 QEMU 模拟器中内置的一块虚拟开发板）模拟的是一款通用的 RISC-V 开发板，其中包括内存布局、中断分配、CPU 配置、时钟配置等信息，这些信息目前都实现在 QEMU 的源代码中，具体文件是 qemu/hw/riscv/virt.c。QEMU Virt 实验平台具有如下硬件特性：

- 最多支持 8 个 RV32GC/RV64GC 通用处理器内核；
- 支持 CLINT；
- 支持平台级别的中断控制器；
- 支持 NOR Flash 存储器；
- 支持兼容 NS16550 的串口；
- 支持 RTC；
- 支持 8 个 VirtIO-MMIO 传输设备；
- 支持 1 个 PCIe 主机桥接设备；
- 支持 fw_cfg，用于从 QEMU 获取固件配置信息。

QEMU Virt 开发板的地址空间布局如表 2.1 所示。

表 2.1　　QEMU Virt 开发板的地址空间布局

内存类型	起始地址	大小/字节
ROM	0x1000	0xF000
RTC	0x10 1000	0x1000
CLINT	0x200 0000	0x10000
ACLINT_SSWI	0x2F0 0000	0x4000
PCIe PIO	0x300 0000	0x10000
PLIC	0xC00 0000	—
UART0	0x1000 0000	0x100
VirtIO	0x1000 1000	0x1000
FW 配置信息	0x1010 0000	0x18

续表

内存类型	起始地址	大小/字节
Flash 存储器	0x2000 0000	0x400 0000
PCIe ECAM	0x3000 0000	0x1000 0000
PCIe MMIO	0x4000 0000	0x4000 0000
RAM	0x8000 0000	用户指定

QEMU Virt 模拟的开发板的中断号分配如表 2.2 所示。

表 2.2　QEMU Virt 模拟的开发板的中断号分配

中断类型	中断号
VIRTIO_IRQ	1～8
UART0	10
RTC	11
PCIe	32～35

2.1.2　NEMU

NEMU（NJU Emulator）最早是由南京大学实现的一个用于教学的计算机指令集体系结构（Instruction Set Architecture，ISA）模拟器，香山处理器团队基于 2019 版的 NEMU 进行增强和维护，用于香山处理器前期指令集和体系结构的模拟。本章中把 NEMU 简称为香山模拟器。NEMU 支持 x86、RV32 和 RV64 等指令集体系结构。

RV64 版本 NEMU 具有如下硬件特性：

- 支持 1 个 RV64GC 通用处理器内核；
- 支持 CLINT；
- 支持 NOR Flash 存储器；
- 支持兼容 NS16550 的串口；
- 支持 Xilinx UartLite 串口控制器；
- 支持 RTC；
- 支持单步调试。

RV64 版本的 NEMU Virt 开发板的地址空间布局如表 2.3 所示。

表 2.3　NEMU Virt 开发板的地址空间布局

内存类型	起始地址	大小/字节
CLINT	0x200 0000	0x10000
UART	0x1000 0000	0x100
Flash	0x2000 0000	0x400 0000
RAM	0x8000 0000	用户指定

表 2.1 的信息都可以在 NEMU 的 Configure 菜单中配置。

在使用 NEMU 之前需要自行编译。

```
$ sudo apt install libsdl2-dev libreadline-dev
$ git clone https://github.com/runninglinuxkernel/NEMU.git
$ cd NEMU
$ export NEMU_HOME=$(pwd)
$ make riscv64-benos_defconfig
$ make -j$(nproc)
$ sudo cp build/riscv64-nemu-interpreter /usr/local/bin
```

2.2 搭建实验环境

2.2.1 实验 2-1：输出“Welcome RISC-V!”

1. 实验目的

了解并熟悉如何在 NEMU 和 QEMU 模拟器上运行最简单的 BenOS 程序。

2. 实验详解

首先，在 Linux 主机中安装相关工具。

```
$ sudo apt-get install qemu-system-misc libncurses5-dev gcc-riscv64-linux-gnu build-esse
ntial git bison flex libssl-dev
```

1）在 QEMU 上运行 BenOS

在 Linux 主机上使用 make 命令编译 BenOS。

```
$ cd benos
$ export board=qemu
$ make clean
$ make
 CC   build_src/uart_c.o
 CC   build_src/kernel_c.o
 AS   build_src/boot_s.o
 LD build_src/benos.elf
 OBJCOPY benos.bin
 CC   build_sbi/sbi_main_c.o
 AS   build_sbi/sbi_boot_s.o
 AS   build_sbi/sbi_payload_s.o
 LD build_sbi/mysbi.elf
 OBJCOPY mysbi.bin
 LD build_sbi/benos_payload.elf
 OBJCOPY benos_payload.bin
```

要编译能在 QEMU 模拟器上运行的可执行二进制文件，需要设置“board=qemu”环境变量。

直接输入“make run”命令来运行。

```
rlk@master:benos$ make run
qemu-system-riscv64 -nographic -machine virt -m 128M  -bios mysbi.bin  -device loader,fi
le=benos.bin,addr=0x80200000  -kernel benos.elf

Welcome RISC-V!
```

QEMU 输出“Welcome RISC-V!”。

2）在 NEMU 上运行 BenOS

在 Linux 主机上使用“make”命令编译 BenOS。

```
$ cd benos
$ export board=nemu
$ make clean
$ make
 CC   build_src/uart_c.o
 CC   build_src/kernel_c.o
 AS   build_src/boot_s.o
 LD build_src/benos.elf
 OBJCOPY benos.bin
 CC   build_sbi/sbi_main_c.o
 AS   build_sbi/sbi_boot_s.o
 AS   build_sbi/sbi_payload_s.o
 LD build_sbi/mysbi.elf
 OBJCOPY mysbi.bin
 LD build_sbi/benos_payload.elf
 OBJCOPY benos_payload.bin
```

要编译能在 NEMU 模拟器上运行的可执行二进制文件，需要设置“board=nemu”环境变量。如果想查看更多编译日志，可以使用“make V=1”命令。编译完成之后会生成 5 个文件。

- benos.bin：BenOS 可执行文件。
- benos.elf：BenOS 带调试信息的可执行与可链接格式（Executable and Linkable Format，ELF）文件。
- mysbi.bin：MySBI 固件的可执行文件。
- mysbi.elf：MySBI 带调试信息的 ELF 文件。
- benos_payload.bin：把 benos.bin 和 mysbi.bin 整合到一个可执行二进制文件中。

直接输入“make run”命令来运行。

```
rlk@master:benos$ make run
riscv64-nemu-interpreter -b benos_payload.bin

Welcome to riscv64-NEMU!
For help, type "help"
Welcome RISC-V!
```

NEMU 输出“Welcome RISC-V!”，如图 2.1 所示。

```
rlk@master:benos$ make run
riscv64-nemu-interpreter -b benos_payload.bin
[src/device/io/mmio.c:18,add_mmio_map] Add mmio map 'clint' at [0x0000000002000000, 0x000000000200ffff]
[src/isa/riscv64/init.c:69,init_isa] NEMU will start from pc 0x80000000
[src/monitor/image_loader.c:53,load_img] Loading image (bbl/bare metal app) from cmdline: benos_payload.bin

[src/monitor/image_loader.c:82,load_img] Read 2105352 bytes from file benos_payload.bin to 0x80000000
[src/device/io/port-io.c:15,add_pio_map] Add port-io map 'serial' at [0x0000000000000000, 0x0000000000000007]
[src/device/io/mmio.c:18,add_mmio_map] Add mmio map 'serial' at [0x0000000010000000, 0x0000000010000007]
[src/device/io/port-io.c:15,add_pio_map] Add port-io map 'rtc' at [0x0000000000000000, 0x0000000000000007]
[src/device/io/mmio.c:18,add_mmio_map] Add mmio map 'rtc' at [0x0000000002000000, 0x0000000002000007]
[src/monitor/monitor.c:28,welcome] Debug: ON
[src/monitor/monitor.c:29,welcome] If debug mode is on, a log file will be generated to record every instruction NEMU
executes. This may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c:33,welcome] Build time: 12:57:33, May  3 2022
Welcome to riscv64-NEMU!
For help, type "help"
Welcome RISC-V!
```

图 2.1　NEMU 的输出结果

2.2.2　实验 2-2：单步调试 BenOS 和 MySBI

本节提供两种单步调试 BenOS 的方法。一是使用 QEMU 与 GDB 工具，二是使用 NEMU 内置的单步调试功能。

1. 使用 QEMU 的 gdbserver

本节以实验 2-1 为例，在终端启动 QEMU 虚拟机的 gdbserver。使用“make debug”命令启动 gdbserver。

```
$ cd benos
$ make debug
```

或者直接使用如下命令启动 gdbserver。

```
$ qemu-system-riscv64 -nographic -machine virt -m 128M -bios mysbi.bin  -device loader,fi
le=benos.bin,addr=0x80200000  -kernel benos.elf -S -s
```

在另一个终端，输入如下命令启动 GDB。

```
$ gdb-multiarch --tui benos.elf
```

在 GDB 命令行中，输入如下命令。

```
(gdb) target remote localhost:1234
(gdb) b _start
Breakpoint 1 at 0x80200000: file src/boot.S, line 6.
(gdb) c
```

此时，使用 GDB 单步调试，如图 2.2 所示。

另外，我们也可以通过 GDB 调试 MySBI 固件。为了在终端启动 QEMU 虚拟机的 gdbserver，使用“make debug”命令。

```
$ cd benos
$ make debug
```

在另一个终端输入如下命令启动 GDB，这时候我们需要加载 MySBI 固件的 ELF 文件。

```
$ gdb-multiarch --tui mysbi.elf
```

在 GDB 命令行中输入如下命令。

```
(gdb) target remote localhost:1234
(gdb) b _start
Breakpoint 1 at 0x80000000: file sbi/sbi_boot.S, line 6.
(gdb) c
```

此时，使用 GDB 命令进行单步调试，如图 2.3 所示。

图 2.2 使用 GDB 调试 BenOS

图 2.3 单步调试

2. NEMU 单步调试机制

NEMU 内置了简易调试器（Simple Debugger，SDB）。SDB 是 NEMU 中一个非常重要的基础设施，可以获取程序的执行信息，以协助调试。SDB 支持的命令如表 2.4 所示。

表 2.4 SDB 支持的命令

命令	格式	说明
继续运行	c	继续运行暂停的程序
退出	q	退出 NEMU
单步执行	si [*N*]	让程序单步执行 *N* 条指令后暂停执行，当 *N* 没有给出时，默认为 1
输出程序状态	info SUBCMD	info r：输出寄存器信息。 info w：输出监视点信息
扫描内存	x *N* EXPR	例如，x 10 $sp 用于求出表达式$sp 的值，将结果作为起始内存地址，以十六进制形式输出连续的 10 个 4 字节
表达式求值	p EXPR	求出表达式（EXPR）的值
设置监视点	w EXPR	当表达式的值发生变化时，暂停程序
删除监视点	d *N*	删除序号为 *N* 的监视点

在 Linux 主机终端，使用“make debug”命令启动 SDB。

```
$ cd benos
$ make debug
riscv64-nemu-interpreter benos_payload.bin
Welcome to riscv64-NEMU!
For help, type "help"
(nemu)
```

在 SDB 命令行中，输入“help”，查看帮助信息，如图 2.4 所示。

输入“si”，进行单步调试，如图 2.5 所示。

```
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
si - step
info - info r - print register values; info w - show watch point state
x - Examine memory
p - Evaluate the value of expression
w - Set watchpoint
d - Delete watchpoint
detach - detach diff test
attach - attach diff test
save - save snapshot
load - load snapshot
q - Exit NEMU
(nemu)
```

图 2.4　查看帮助信息

```
(nemu) si
0x0000000080000000:   17 11 00 00                          auipc  0x1000,sp
[src/cpu/cpu-exec.c:74,monitor_statistic] host time spent = 10 us
[src/cpu/cpu-exec.c:76,monitor_statistic] total guest instructions = 1
[src/cpu/cpu-exec.c:77,monitor_statistic] simulation frequency = 100,000 instr/s
(nemu)
```

图 2.5　进行单步调试

另外，我们还可以通过监视点命令设置断点，不过目前只支持使用地址来设置断点。例如，在 0x8020 0000 地址处设置断点。

```
$ make debug

(nemu) w $pc==0x80200000   //设置断点
Set watchpoint #0
(nemu) c                   //继续执行，此时 NEMU 停在断点处
Hint watchpoint 0 at address 0x00000000800000a8, expr = $pc==0x80200000
old value = 0x0000000000000000
new value = 0x0000000000000001
(nemu) si                  //单步执行
0x0000000080200000:   73 10 40 10                    csrrw  $0,0x104,$0


Hint watchpoint 0 at address 0x0000000080200000, expr = $pc==0x80200000
old value = 0x0000000000000001
new value = 0x0000000000000000
```

2.3 BenOS 和 MySBI 基础实验代码解析

本书中大部分的实验代码是基于 BenOS 实现的。本书的实验会从最简单的裸机程序开始，逐步对其进行扩展和丰富，让其具有进程调度、系统调用等现代操作系统的基本功能。

BenOS 基础实验代码包含 MySBI 和 BenOS 两部分，其中 MySBI 是运行在 M 模式下的固件，为运行在 S 模式下的操作系统提供引导和统一的接口服务。BenOS 基础实验代码的结构如图 2.6 所示。

其中，sbi 目录包含 MySBI 的源文件；src 目录包含 BenOS 的源文件；include 目录包含 BenOS 和 MySBI 共用的头文件。

```
rlk@master:benos$ tree
.
├── include
│   ├── asm
│   │   ├── csr.h
│   │   └── uart.h
│   ├── io.h
│   └── uart.h
├── Makefile
├── sbi
│   ├── sbi_base.ld
│   ├── sbi_boot.S
│   ├── sbi_linker.ld
│   ├── sbi_linker_payload.ld
│   ├── sbi_main.c
│   ├── sbi_payload
│   └── sbi_payload.S
└── src
    ├── boot.S
    ├── kernel.c
    ├── linker.ld
    └── uart.c
```

图 2.6　BenOS 基础实验代码的结构

2.3.1　MySBI 基础代码分析

本书的实验并没有采用业界流行的 OpenSBI 固件，我们从零开始编写一个小型可用的 SBI 固件，以便从底层深入学习 RISC-V 体系结构。

系统上电后，RISC-V 处理器运行在 M 模式。通常 SBI 固件运行在 M 模式，为运行在 S 模

式的操作系统提供引导服务以及 SBI 服务。不过本节介绍的 MySBI 代码仅仅提供引导服务，我们会在后续的实验中逐步添加 SBI 服务。

MySBI 本质上是一个裸机程序，因此我们先从链接脚本（Linker Script，LS）开始分析。

任何一种可执行程序（不论是.elf 还是.exe 文件）都是由代码（.text）段、数据（.data）段、未初始化的数据（.bss）段等段（section）组成的。链接脚本最终会把大量编译好的二进制文件（.o 文件）综合成二进制可执行文件，也就是把所有二进制文件链接到一个大文件中。这个大文件由总的.text/.data/.bss 段描述。下面是 MySBI 中的一个链接脚本，名为 sbi_linker.ld。

```
<benos/sbi/sbi_linker.ld>

1  OUTPUT_ARCH(riscv)
2  ENTRY(_start)
3
4  SECTIONS
5  {
6      INCLUDE "sbi/sbi_base.ld"
7  }
```

在第 1 行中，OUTPUT_ARCH 说明这个链接脚本对应的处理器体系结构为 RISC-V。

在第 2 行中，指定程序的入口地址为_start。

在第 4 行中，SECTIONS 是链接脚本语法中的关键命令，用来描述输出文件的内存布局。SECTIONS 命令告诉链接脚本如何把输入文件的段映射到输出文件的各个段，如何将输入段整合为输出段，以及如何把输出段放入虚拟存储器地址（Virtual Memory Address，VMA）和加载存储器地址（Load Memory Address，LMA）。

在第 6 行中，通过 INCLUDE 命令引入 sbi/sbi_base.ld 脚本。

```
<benos/sbi/sbi_base.ld>

1      /*
2       * 设置 SBI 的加载入口地址为 0x8000 0000
3       */
4
5     . =0x80000000,
6
7     .text.boot : { *(.text.boot) }
8     .text : { *(.text) }
9     .rodata : { *(.rodata) }
10    .data : { *(.data) }
11    . = ALIGN(0x8);
12    bss_begin = .;
13    .bss : { *(.bss*) }
14    bss_end = .;
```

在第 5 行中，“.”非常关键，它代表当前位置计数器（Location Counter，LC），这里把.text 段的链接地址设置为 0x8000 0000，这里链接地址指的是加载地址（load address）。

在第 7 行中，输出文件的.text.boot 段由所有输入文件（其中的“*”可理解为所有的.o 文件，也就是二进制文件）的.text.boot 段组成。

在第 8 行中，输出文件的.text 段由所有输入文件（其中的“*”可理解为所有的.o 文件，也就是二进制文件）的.text 段组成。

在第 9 行中，输出文件的.rodata 段由所有输入文件的.rodata 段组成。

在第 10 行中，输出文件的.data 段由所有输入文件的.data 段组成。

在第 11 行中，设置对齐方式为按 8 字节对齐。

在第 12～14 行中，定义了一个.bss 段。

因此，上述链接脚本定义了如下几个段。

- .text.boot 段：该段包含系统启动时首先要执行的代码，即把_start 函数链接到 0x8000 0000 地址。
- .text 段：代码段。
- .rodata 段：只读数据段。
- .data 段：数据段。
- .bss 段：包含未初始化的全局变量和未初始化的局部静态变量。

下面开始编写 MySBI 启动用的汇编代码，将代码保存为 sbi_boot.S 文件。

```
<benos/sbi/sbi_boot.S>

1     .section ".text.boot"
2
3     .globl _start
4     _start:
5         /*关闭 M 模式的中断*/
6         csrw mie, zero
7
8         /*设置栈，栈的大小为 4 KB*/
9         la sp, stacks_start
10        li t0, 4096
11        add sp, sp, t0
12
13        /*跳转到 C 语言的 sbi_main()函数*/
14        tail sbi_main
15
16    .section .data
17    .align  12
18    .global stacks_start
19    stacks_start:
20            .skip 4096
```

启动 MySBI 的汇编代码不长，下面进行简要分析。

在第 1 行中，把 sbi_boot.S 文件编译、链接到.text.boot 段。我们可以在链接脚本 sbi_linker.ld 中把.text.boot 段链接到这个可执行文件的开头，这样当程序执行时将从这个段开始执行。此时，处理器运行在 M 模式。

在第 4 行中，_start 为程序的入口点。

在第 6 行中，关闭 M 模式的所有中断。

在第 9～11 行中，初始化栈指针，为栈分配 4 KB 的空间。

在第 14 行中，跳转到 C 语言的 sbi_main()函数。

sbi_main.c 源文件如下。

```
<benos/sbi/sbi_main.c>

1     #include "asm/csr.h"
2
3     #define FW_JUMP_ADDR 0x80200000
4
5     /*
6      * 运行在 M 模式，并且切换到 S 模式
7      */
8     void sbi_main(void)
9     {
10        unsigned long val;
11
12        /*设置跳转模式为 S 模式 */
13        val = read_csr(mstatus);
14        val = INSERT_FIELD(val, MSTATUS_MPP, PRV_S);
15        val = INSERT_FIELD(val, MSTATUS_MPIE, 0);
16        write_csr(mstatus, val);
```

```
17
18          /*设置 M 模式的异常程序计数器，用于 mret 跳转 */
19          write_csr(mepc, FW_JUMP_ADDR);
20          /*设置 S 模式的异常向量表入口地址*/
21          write_csr(stvec, FW_JUMP_ADDR);
22          /*关闭 S 模式的中断*/
23          write_csr(sie, 0);
24          /*关闭 S 模式的页表转换*/
25          write_csr(satp, 0);
26
27          /*切换到 S 模式*/
28          asm volatile("mret");
29      }
```

sbi_main()函数的主要目的是把处理器模式从 M 模式切换到 S 模式，并跳转到 S 模式的入口地址处。对于 QEMU Virt 平台，S 模式的入口地址为 0x8020 0000。

在第 13～16 行中，设置 mstatus 寄存器中的 MPP 字段为 S 模式，并且把中断使能保存位 MPIE 也清除。

在第 19 行中，当处理器陷入 M 模式时，mepc 寄存器记录了陷入时的异常地址。因此，这里设置 M 模式的跳转地址为 0x8020 0000，执行 mret 指令会跳转到 0x8020 0000 地址处。

在第 28 行中，执行 mret 指令，完成模式切换。

2.3.2 BenOS 基础代码分析

本节介绍 BenOS 的代码体系结构，目前它只有串口输出功能，类似于裸机程序。BenOS 的链接脚本参见 benos/src/linker.ld 文件。

```
<benos/src/linker.ld>

SECTIONS
{
    . =0x80200000,

    .text.boot : { *(.text.boot) }
    .text : { *(.text) }
    .rodata : { *(.rodata) }
    .data : { *(.data) }
    . = ALIGN(0x8);
    bss_begin = .;
    .bss : { *(.bss*) }
    bss_end = .;
}
```

上述链接脚本与 benos/sbi/sbi_linker.ld 类似，唯一的区别在于链接地址不一样。BenOS 的入口地址为 0x8020 0000。

下面开始编写启动 BenOS 用的汇编代码，将代码保存为 boot.S 文件。

```
<benos/src/boot.S>

.section ".text.boot"

.globl _start
_start:
    /*关闭中断*/
    csrw sie, zero

    /*设置栈*/
    la sp, stacks_start
    li t0, 4096
    add sp, sp, t0

```

```
13        call  kernel_main
14
15    hang:
16        wfi
17        j hang
18
19    .section .data
20    .align  12
21    .global stacks_start
22    stacks_start:
23            .skip 4096
```

启动用的汇编代码不长，下面进行简要分析。

在第 1 行中，把 boot.S 文件编译、链接到.text.boot 段。我们可以在链接脚本 link.ld 中把.text.boot 段链接到这个可执行文件的开头，这样当程序执行时将从这个段开始执行。

在第 4 行中，_start 为程序的入口点。此时，处理器模式运行在 S 模式。

在第 6 行中，屏蔽所有的中断源。

在第 9～11 行中，初始化栈指针，为栈分配 4 KB 的空间。

在第 13 行中，跳转到 C 语言的 kernel_main()函数。

总之，上述汇编代码还是比较简单的，我们只做了一件事情——设置栈，跳转到 C 语言入口。

接下来，编写 C 语言的 kernel_main()函数。本实验的目的是输出一条欢迎语句，因此这个函数的实现比较简单，将代码保存为 kernel.c 文件。

```
<benos/src/kernel.c>

1    #include "uart.h"
2
3    void kernel_main(void)
4    {
5        uart_init();
6        uart_send_string("Welcome RISC-V!\r\n");
7
8        while (1) {
9            ;
10       }
11   }
```

上述代码很简单，主要操作是初始化串口和往串口里输出欢迎语句。

接下来，实现简单的串口驱动代码。QEMU 使用兼容 16550 规范的串口控制器。16550 串口控制器内部的寄存器如表 2.5 所示。这些寄存器的偏移地址由芯片的 A0～A2 引脚确定。另外，预分频寄存器的高/低字节与其他寄存器复用，可以通过线路控制寄存器（LCR）的 DLAB 字段区分。

表 2.5　16550 串口控制器内部的寄存器

DLAB 字段	寄存器偏移地址	寄存器说明
0	0b000	数据寄存器，接收数据或者发送数据
0	0b001	中断使能寄存器
1	0b000	预分频寄存器中的低 8 位
1	0b0001	预分频寄存器中的高 8 位
—	0b010	中断标识寄存器或者先进先出（First In First Out，FIFO）控制寄存器
—	0b011	线路控制寄存器
—	0b100	MODEN 控制寄存器
—	0b101	线路状态寄存器
—	0b110	MODEN 状态寄存器

下面是 16550 串口的初始化代码。

```
<benos/src/uart.c>

static unsigned int uart16550_clock = 1843200; //串口时钟
#define UART_DEFAULT_BAUD  115200

void uart_init(void)
{
    unsigned int divisor = uart16550_clock / (16 * UART_DEFAULT_BAUD);

    /*关闭中断*/
    writeb(0, UART_IER);

    /*打开 DLAB 字段，用来设置波特率分频*/
    writeb(0x80, UART_LCR);
    writeb((unsigned char)divisor, UART_DLL);
    writeb((unsigned char)(divisor >> 8), UART_DLM);

    /*设置串口数据格式*/
    writeb(0x3, UART_LCR);

    /*使能 FIFO 缓冲区，清空 FIFO 缓冲区，设置 14 字节阈值*/
    writeb(0xc7, UART_FCR);
}
```

上述代码关闭中断，设置串口的波特率，设置串口数据格式（一个起始位、8 个数据位以及 1 个停止位），使能 FIFO 缓冲区，清空 FIFO 缓冲区。

接下来，实现如下几个函数来发送字符串。

```
<benos/src/uart.c>

void uart_send(char c)
{
    while((readb(UART_LSR) & UART_LSR_EMPTY) == 0)
        ;

    writeb(c, UART_DAT);
}

void uart_send_string(char *str)
{
    int i;

    for (i = 0; str[i] != '\0'; i++)
        uart_send((char) str[i]);
}
```

uart_send()用于在 while 循环中判断是否有数据需要发送，这里只需要判断 UART_LSR 寄存器的发送移位寄存器即可。

接下来，编写 Makefile 文件。

```
<benos/Makefile 文件>

GNU ?= riscv64-linux-gnu

COPS += -save-temps=obj -g -O0 -Wall -nostdlib -nostdinc -Iinclude -mcmodel=medany
-mabi=lp64 -march=rv64imafd -fno-PIE -fomit-frame-pointer

board ?= qemu

ifeq ($(board), qemu)
COPS += -DCONFIG_BOARD_QEMU
else ifeq ($(board), nemu)
COPS += -DCONFIG_BOARD_NEMU
```

```
11  endif
12
13  ##############
14  #  build benos
15  ##############
16  BUILD_DIR = build_src
17  SRC_DIR = src
18
19  all : clean benos.bin mysbi.bin benos_payload.bin
20
21  #检查进程的冗余功能是否开启
22  CMD_PREFIX_DEFAULT := @
23  ifeq ($(V), 1)
24      CMD_PREFIX :=
25  else
26      CMD_PREFIX := $(CMD_PREFIX_DEFAULT)
27  endif
28
29  clean :
30      rm -rf $(BUILD_DIR) $(SBI_BUILD_DIR) *.bin  *.map *.elf
31
32  $(BUILD_DIR)/%_c.o: $(SRC_DIR)/%.c
33      $(CMD_PREFIX)mkdir -p $(BUILD_DIR); echo " CC   $@" ; $(GNU)-gcc $(COPS) -c $< -o $@
34
35  $(BUILD_DIR)/%_s.o: $(SRC_DIR)/%.S
36      $(CMD_PREFIX)mkdir -p $(BUILD_DIR); echo " AS   $@"; $(GNU)-gcc $(COPS) -c $< -o $@
37
38  C_FILES = $(wildcard $(SRC_DIR)/*.c)
39  ASM_FILES = $(wildcard $(SRC_DIR)/*.S)
40  OBJ_FILES = $(C_FILES:$(SRC_DIR)/%.c=$(BUILD_DIR)/%_c.o)
41  OBJ_FILES += $(ASM_FILES:$(SRC_DIR)/%.S=$(BUILD_DIR)/%_s.o)
42
43  DEP_FILES = $(OBJ_FILES:%.o=%.d)
44  -include $(DEP_FILES)
45
46  benos.bin: $(SRC_DIR)/linker.ld $(OBJ_FILES)
47      $(CMD_PREFIX)$(GNU)-ld -T $(SRC_DIR)/linker.ld -o $(BUILD_DIR)/benos.elf
        $(OBJ_FILES) -Map benos.map; echo " LD $(BUILD_DIR)/benos.elf"
48      $(CMD_PREFIX)$(GNU)-objcopy $(BUILD_DIR)/benos.elf -O binary benos.bin;
        echo " OBJCOPY benos.bin"
49      $(CMD_PREFIX)cp $(BUILD_DIR)/benos.elf benos.elf
50
51  ##############
52  #  build SBI
53  ##############
54  # 此处省略，建议读者查看本书配套源代码
```

上述 Makefile 文件使用 board 变量来选择支持 NEMU 或者 QEMU。

GNU 用来指定编译器，这里使用 riscv64-linux-gnu-gcc。

COPS 用来在编译 C 语言和汇编语言时指定编译选项。

- -g：表示编译时加入调试符号表等信息。
- -Wall：表示打开所有警告信息。
- -nostdlib：表示不连接系统的标准启动文件和标准库文件，只把指定的文件传递给链接器。这个选项常用于编译内核、bootloader 等程序，它们不需要标准启动文件和标准库文件。
- -nostdinc：表示不包含 C 语言的标准库的头文件。
- -mcmodel=medany：目标代码模型，主要表示符号地址的约束。编译器可以利用这些约束来生成更有效的代码。RISC-V 上主要有两个选项。
  - medlow：表示程序及符号必须介于绝对地址减 2 GB 和绝对地址加 2 GB。
  - medany：表示程序及符号能访问 PC − 2 GB 到 PC + 2 GB 这个地址空间。
- -mabi=lp64：表示支持的数据模型。
- -march=rv64imafdc：表示处理器的指令集。

- -fno-PIE：PIE（position independent executables）表示与位置无关的可执行程序。在GCC 中，“-fpic”与“-fPIE”类似，只不过“-fpic”适用于编译动态库，“-fPIE”适用于编译可执行程序。

上述文件会编译和链接两个可执行的 ELF 文件——benos.elf 和 mysbi.elf。这些.elf 文件包含调试信息，最后使用 objcopy 命令把.elf 文件转换为可执行的二进制文件 benos.bin 和 mysbi.bin 文件。

2.3.3 合并 BenOS 和 MySBI

NEMU 运行环境要求使用一个完整的二进制可执行文件，即把 benos.bin 和 mysbi.bin 合并。我们可以利用链接脚本实现这个功能。下面是 sbi_linker_payload.ld。

```
<benos/src/sbi_linker_payload.ld>

1    SECTIONS
2    {
3        INCLUDE "sbi/sbi_base.ld"
4
5        . = 0x80200000;
6
7        .payload :
8        {
9            PROVIDE(_payload_start = .);
10           *(.payload)
11           . = ALIGN(8);
12           PROVIDE(_payload_end = .);
13       }
14   }
```

在第 3 行中，同样使用 INCLUDE 命令引入 sbi/sbi_base.ld 脚本。

在第 5 行中，把当前的链接地址设置为 0x8020 0000。

在第 7～13 行中，新建一个名为.payload 的段，这个段的起始地址为 0x8020 0000，这个地址是 BenOS 的入口地址。

在 sbi_payload.S 汇编文件中使用.incbin 伪指令把 benos.bin 二进制数据嵌入.payload 段中，完成合并工作。

```
<benos/sbi/sbi_payload.S>

1        .section .payload, "ax"
2        .globl payload_bin
3    payload_bin:
4        .incbin  "benos.bin"
```

在 Makefile 中还需要使用 LD 命令进行链接，最后生成 benos_payload.elf 文件以及 benos_payload.bin。

2.4 QEMU + RISC-V + Linux 实验平台

本书中少部分实验(如指令集实验和部分高速缓存伪共享实验)可以在基于 RISC-V 的 Linux 主机上完成。基于 RISC-V 的 Linux 主机可以通过两种方式获取。一种是购买 RISC-V 开发板，另一种是使用 QEMU+RISC-V 实验平台。

下面介绍 QEMU+RISC-V 实验平台。Linux 主机使用 Ubuntu 20.04 系统。

1）安装工具

首先，在 Linux 主机中安装相关工具。

```
$ sudo apt-get install qemu-system-misc libncurses5-dev gcc-riscv64-linux-gnu
build-essential git bison flex libssl-dev opensbi
```

然后，在 Linux 主机系统中默认安装 RISC-V GCC 编译器的 9.3 版本。

```
$ riscv64-linux-gnu-gcc -v
gcc version 9.3.0 (Ubuntu 9.3.0-8ubuntu1)
```

2）下载仓库

下载 runninglinuxkernel_5.15 的 GIT 仓库并切换到 master 分支。

```
$ git clone https://github.com/runninglinuxkernel/runninglinuxkernel_5.15.git
```

3）编译内核并创建文件系统

runninglinuxkernel_5.15 目录中有一个 rootfs_debian_riscv.tar.xz 文件，这个文件采用 Ubuntu Linux 20.04 系统的根文件系统制作而成。

注意，该脚本会使用 dd 命令生成一个 2 GB 大小的镜像文件，因此主机系统需要保证至少 10 GB 的空余磁盘空间。读者如果需要生成更大的根文件系统镜像文件，那么可以修改 run_rlk_riscv.sh 脚本。

首先，编译内核。

```
$ cd runninglinuxkernel_5.15
$ ./run_rlk_riscv.sh build_kernel
```

执行上述脚本需要几十分钟时间，具体依据主机的计算能力。

然后，编译根文件系统。

```
$ sudo ./run_rlk_riscv.sh build_rootfs
```

注意，编译根文件系统需要管理员权限，而编译内核则不需要。执行完上述命令后，将会生成名为 rootfs_debian_riscv.ext4 的根文件系统。

4）运行刚才编译好的 RISC-V 版本的 Linux 系统

要运行 run_rlk_riscv.sh 脚本，输入 run 参数即可。

```
$./run_rlk_riscv.sh run
```

运行结果如下。

```
rlk@ runninglinuxkernel_5.15 $ ./run_rlk_riscv.sh run
[   0.000000] OF: fdt: Ignoring memory range 0x80000000 - 0x80200000
[   0.000000] No DTB passed to the kernel
[   0.000000] Linux version 5.15.0+ (rlk@master) (gcc version 9.3.0 (Ubuntu
9.3.0-17ubuntu1~20.04)) #4 SMP Thu Sep 9 19:14:52 CST 2021
[   0.000000] initrd not found or empty - disabling initrd
[   0.000000] Zone ranges:
[   0.000000]   DMA32    [mem 0x0000000080200000-0x00000000bfffffff]
[   0.000000]   Normal   empty
[   0.000000] Movable zone start for each node
[   0.000000] Early memory node ranges...
...
rlk login:
```

登录系统时使用的用户名和密码如下。

- 用户名：root。
- 密码：123。

5）在线安装软件包

QEMU 虚拟机可以通过 VirtIO-Net 技术生成虚拟的网卡，并通过网络地址转换（Network Address Translation，NAT）技术和主机进行网络共享。下面使用 ifconfig 命令检查网络配置。

```
root@ubuntu:~# ifconfig
enp0s1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fec0::ce16:adb:3e70:3e71  prefixlen 64  scopeid 0x40<site>
        inet6 fe80::c86e:28c4:625b:2767  prefixlen 64  scopeid 0x20<link>
        ether 52:54:00:12:34:56  txqueuelen 1000  (Ethernet)
        RX packets 23217  bytes 33246898 (31.7 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4740  bytes 267860 (261.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

可以看到，这里生成名为 enp0s1 的网卡设备，分配的 IP 地址为 10.0.2.15。

可通过 apt update 命令更新 Debian 系统的软件仓库。

```
root@ubuntu:~# apt update
```

如果更新失败，有可能因为系统时间比较旧，可以使用 date 命令设置日期。

```
root@ubuntu:~# date -s 2020-03-29 #假设最新日期是 2020 年 3 月 29 日
Sun Mar 29 00:00:00 UTC 2020
```

使用 apt install 命令安装软件包。比如，在线安装 GCC 等软件包。

```
root@ubuntu:~# apt install gcc build-essential
```

6）在主机和 QEMU 虚拟机之间共享文件

主机和 QEMU 虚拟机可以通过 NET_9P 技术进行文件共享，这需要 QEMU 虚拟机和主机的 Linux 内核都使能 NET_9P 的内核模块。本实验平台已经支持主机和 QEMU 虚拟机的共享文件，可以通过如下简单方法来测试。

复制一个文件到 runninglinuxkernel_5.15/kmodules 目录中。

```
$ cp test.c  runninglinuxkernel_5.15/kmodules
```

启动 QEMU 虚拟机之后，检查一下/mnt 目录中是否有 test.c 文件。

```
root@ubuntu:/# cd /mnt
root@ubuntu:/mnt # ls
README    test.c
```

我们在后续的实验（例如，第 7 章以及第 14～16 章的部分实验）中会经常利用这个特性，如把编写好的代码文件复制到 QEMU 虚拟机。

第3章 基础指令集

本章思考题

1. RISC-V 指令集有什么特点？
2. RISC-V 指令编码格式分成几类？
3. 什么是零扩展和符号扩展？
4. 什么是 PC 相对寻址？
5. 假设当前 PC 值为 0x8020 0000，分别执行如下指令，那么 a5 和 a6 寄存器的值是多少？

```
auipc   a5,0x2
lui     a6, 0x2
```

6. 在下面的指令中，a1 和 t1 寄存器的值是多少？

```
li t0, 0x8000008a00000000
srai a1, t0, 1
srli t1, t0, 1
```

7. 假设执行如下各条指令时当前 PC 值为 0x8020 0000，下面指令哪些是非法指令？

```
jal a0, 0x800fffff
jal a0, 0x80300000
```

8. 请解析下面这条指令的含义。

```
csrrw tp, sscratch, tp
```

9. 在 RISC-V 指令集中，如何实现大范围和小范围内跳转？

3.1 RISC-V 指令集介绍

指令集是处理器体系结构设计的重点部分之一，作为软硬件的接口，它紧密地连接软件和硬件。目前市面上比较成功的指令集采用增量式的设计方法，目的是保持向后的二进制兼容性。在新增指令的同时，开发人员还必须保留遗留的并且几乎不怎么使用甚至是设计错误的指令，这导致指令集越来越庞大和复杂。而 RISC-V 在指令集设计过程中吸取了这些经验教训，采用模块化的设计方法。所谓的模块化设计就是设计一个最小集合和最基础的指令集，这个最小的指令集可以完整地实现一个软件栈，其他特殊功能的指令集可以在最小指令集的基础上通过模块化的方式叠加实现，用于支持浮点数运算指令、乘法和除法指令、矢量指令等。1.1 节已经介绍了 RISC-V 支持的扩展指令集。

【例 3-1】 在 QEMU+RISC-V 平台上通过“cpuinfo”查看节点的信息。

```
root:~# cat /proc/cpuinfo
processor     : 0
hart     : 0
```

```
isa     : rv64imafdcsu
mmu     : sv48
```

从“isa”可知该系统支持的扩展为 rv64imafdcsu，即支持 64 位的基础整型指令集 I、整型乘法和除法扩展指令集 M、原子操作指令集 A、单精度浮点数扩展指令集 F、双精度浮点数扩展指令集 D、压缩指令集 C、特权模式指令集 S 以及用户模式指令集 U。

3.2 RISC-V 指令编码格式

RISC-V 的每条指令宽度为 32 位（不考虑压缩扩展指令），包括 RV32 指令集以及 RV64 指令集。指令格式大致可分成 6 类。

- R 类型：寄存器与寄存器算术指令。
- I 类型：寄存器与立即数算术指令或者加载指令。
- S 类型：存储指令。
- B 类型：条件跳转指令。
- U 类型：长立即数操作指令。
- J 类型：无条件跳转指令。

RISC-V 指令集编码格式如图 3.1 所示。

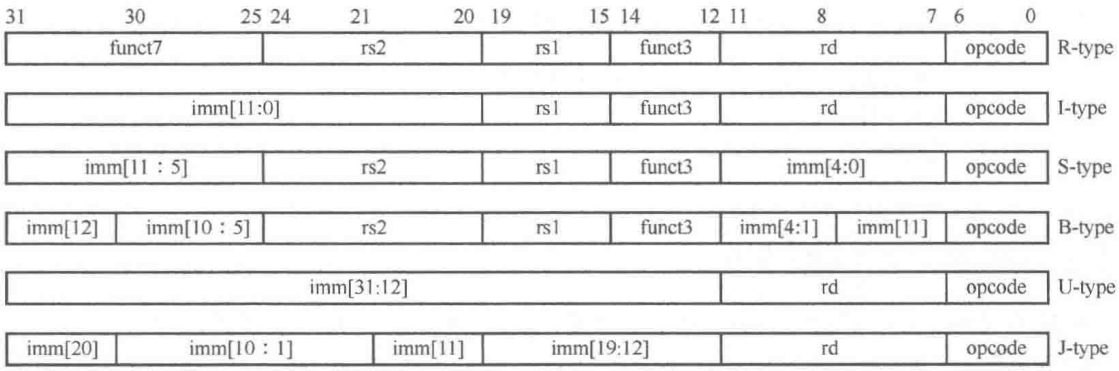


图 3.1 RISC-V 指令集编码格式

指令编码可以分成如下几个部分。

- opcode（操作码）字段：位于指令编码 Bit[6:0]，用于指令的分类。
- funct3 和 funct7（功能码）字段：常常与 opcode 字段结合在一起定义指令的操作功能。
- rd 字段：表示目标寄存器的编号，位于指令编码的 Bit[11:7]。
- rs1 字段：表示第一源操作寄存器的编号，位于指令编码的 Bit[19:15]。
- rs2 字段：表示第二源操作寄存器的编号，位于指令编码的 Bit[24:20]。
- imm：表示立即数。在 RISC-V 中使用的立即数大部分是符号扩展（sign-extended）的立即数。

RV64 指令集支持 64 位宽的数据和地址寻址，为什么指令的编码宽度只有 32 位[①]？

因为 RV64 指令集是基于寄存器加载和存储的体系结构设计，所有的数据加载、存储以及处理都是在通用寄存器中完成的。RISC-V 一共有 32 个通用寄存器，即 x0～x31，例如，x0 寄存器的编号为 0，以此类推。因此，在指令编码中使用 5 位宽（$2^5=32$），即索引 32 个通用寄存器。

① RISC-V 通常使用 32 位定长指令，不过 RISC-V 为了减少代码量，也支持 16 位的压缩扩展指令。

lw 加载指令的编码如图 3.2 所示。

图 3.2　lw 加载指令的编码

- 第 0～6 位为 opcode 字段，用于指令分类。
- 第 7～11 位为 rd 字段，用来描述目标寄存器 rd，它可以从 x0～x31 通用寄存器中选择。
- 第 12～14 位为功能码字段，在加载指令中表示加载数据的位宽。
- 第 15～19 位为基地址 rs1，可以从 x0～x31 通用寄存器中选择。
- 第 20～31 位为 offset 字段，表示偏移量。

RV64 指令集中常用的符号说明如下。

- rd：表示目标寄存器，可以从 x0～x31 通用寄存器中选择。
- rs1：表示源寄存器 1，可以从 x0～x31 通用寄存器中选择。
- rs2：表示源寄存器 2，可以从 x0～x31 通用寄存器中选择。
- ()：通常用来表示寻址模式，例如，(a0)表示以 a0 寄存器的值为基地址进行寻址。这个前面还可以加 offset，表示偏移量，可以是正数或负数。例如，8(a0)表示以 a0 寄存器的值为基地址，然后偏移 8 字节进行寻址。
- { }：表示可选项。
- imm：表示有符号立即数。

3.3 加载与存储指令

和其他 RISC 体系结构一样，RISC-V 体系结构也基于加载和存储的体系结构设计理念。在这种体系结构下，所有的数据处理都需要在通用寄存器中完成，而不能直接在内存中完成。因此，首先把待处理数据从内存加载到通用寄存器，然后进行数据处理，最后把结果写回内存中。

1. 加载指令

加载指令的格式如下。

```
l{d|w|h|b}{u} rd,  offset(rs1),
```

其中，相关选项的含义如下。

- {d|w|h|b}：表示加载的数据宽度。加载指令如表 3.1 所示。
- {u}：可选项，表示加载的数据为无符号数，即采用零扩展方式。如果没有这个选项，表示加载的数据为有符号数，即采用有符号扩展方式。
- rd：表示目标寄存器。
- rs1：表示源寄存器 1。
- (rs1)：表示以 rs1 寄存器的值为基地址进行寻址，简称 **rs1 地址**。
- offset：表示以源寄存器的值为基地址的偏移量。offset 是 12 位有符号数，取值范围为 [−2048, 2047]。

表 3.1 加载指令

加载指令	数据位宽/位	说明
lb rd, offset(rs1)	8	以 rs1 寄存器的值为基地址，在偏移 offset 的地址处加载一字节数据，经过符号扩展之后写入目标寄存器 rd 中
lbu rd, offset(rs1)	8	以 rs1 寄存器的值为基地址，在偏移 offset 的地址处加载一字节数据，经过零扩展之后写入目标寄存器 rd 中
lh rd, offset(rs1)	16	以 rs1 寄存器的值为基地址，在偏移 offset 的地址处加载两字节数据，经过符号扩展之后写入目标寄存器 rd 中
lhu rd, offset(rs1)	16	以 rs1 寄存器的值为基地址，在偏移 offset 的地址处加载两字节数据，经过零扩展之后写入目标寄存器 rd 中
lw rd, offset(rs1)	32	以 rs1 寄存器的值为基地址，在偏移 offset 的地址处加载 4 字节数据，经过符号扩展之后写入目标寄存器 rd 中
lwu rd, offset(rs1)	32	以 rs1 寄存器的值为基地址，在偏移 offset 的地址处加载 4 字节数据，经过零扩展之后写入目标寄存器 rd 中
ld rd, offset(rs1)	64	以 rs1 寄存器的值为基地址，在偏移 offset 的地址处加载 8 字节数据，写入寄存器 rd 中
lui rd, imm	64	先把 imm（立即数）左移 12 位，然后进行符号扩展，把结果写入 rd 寄存器中

上述加载指令的编码如图 3.3 所示，其中字段 opcode 都是一样的，唯一不同的是 funct3 字段。

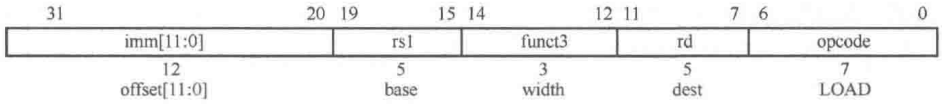


图 3.3 加载指令的编码

【例 3-2】 下面的代码使用了加载指令。

```
1    li t0, 0x80000000
2
3    lb t1, (t0)
4    lb t1, 4(t0)
5    lbu t1, 4(t0)
6    lb t1, -4(t0)
7    ld t1, (t0)
8    ld t1, 16(t0)
```

第 1 行是一条伪指令，它把立即数加载到 t0 寄存器中。

在第 3 行中，从以 t0 寄存器的值为基地址的内存中加载一字节的数据到 t1 寄存器中，对这一字节的数据会进行符号扩展。符号扩展是计算机系统中把小字节转换成大字节的规则之一，它将符号位扩展至所需要的位数。例如，一个 8 位的有符号数为 0x8A，它的最高位（第 7 位）为 1，因此在做符号扩展的过程中，高字节部分需要填充为 0xFF，如图 3.4 所示，符号扩展到 64 位的结果为 0xFFFF FFFF FFFF FF8A。

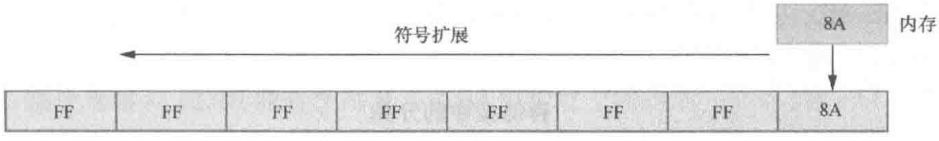


图 3.4 符号扩展

在第 4 行中，以 t0 寄存器的值为基地址再加上 4 字节的偏移量为内存地址（0x8000 0004），从这个内存地址中加载一字节的数据到 t1 寄存器中，对该字节的数据会进行符号扩展。

第 5 行中的指令与第 4 行中的指令基本类似，不同之处在于对该字节的数据不会做符号扩展，即按照无符号数来处理，因此高字节部分填充为 0，称为零扩展，如图 3.5 所示。

在第 6 行中，以 t0 寄存器的值为基地址再减去 4 字节的偏移量为内存地址（0x7FFF FFFC），从这个内存地址中加载一字节的数据到 t1 寄存器中，对该字节的数据会进行符号扩展。

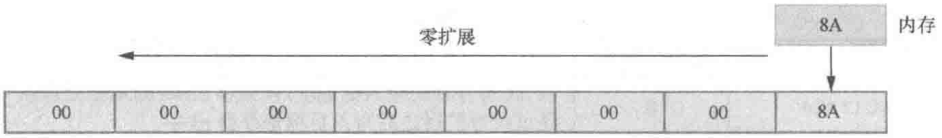


图 3.5　零扩展

在第 7 行中，从以 t0 寄存器的值为基地址的内存中加载 8 字节的数据到 t1 寄存器中。

在第 8 行中，以 t0 寄存器的值为基地址再加上 16 字节的偏移量为内存地址（0x8000 0010），从这个内存地址中加载 8 字节的数据到 t1 寄存器中。

【例 3-3】　下面的代码使用 LUI 加载立即数。

```
lui t0, 0x80200
lui t1, 0x40200
```

在第 1 行中，首先把 0x80200 左移 12 位得到 0x8020 0000，然后进行符号扩展，最后结果等于 0xFFFF FFFF 8020 0000。

在第 2 行中，首先把 0x40200 左移 12 位得到 0x4020 0000，然后进行符号扩展，因为最高位为 0，所以最后结果为 0x4020 0000。

【例 3-4】　下面的代码有错误。

```
lb a1, 2048(a0)
lb a1,-2049(a0)
```

上述指令的偏移量已经超过了取值范围，汇编器会报错。

```
AS   build_src/boot_s.o
src/boot.S: Assembler messages:
src/boot.S:6: Error: illegal operands 'lb a1,-2049(a0)'
src/boot.S:7: Error: illegal operands 'lb a1,2048(a0)'
make: *** [Makefile:28: build_src/boot_s.o] Error 1
```

2. 存储指令

存储指令的格式如下。

```
s{d|w|h|b} rs2,  offset(rs1),
```

其中，相关选项的含义如下。

- {d|w|h|b}：表示存储的数据宽度。根据数据的位宽，存储指令的分类如表 3.2 所示。
- rs1：表示源寄存器 1，用于表示基地址。
- (rs1)：表示以 rs1 寄存器的值为基地址进行寻址，简称 **rs1 地址**。
- rs2：表示源寄存器 2，用来表示源操作数。
- offset：表示以源寄存器的值为基地址的偏移量。offset 是 12 位有符号数，取值范围为 [−2048, 2047]。

表 3.2　存储指令的分类

存储指令	数据位宽/位	说明
sb rs2, offset(rs1)	8	把 rs2 寄存器的低 8 位宽的值存储到以 rs1 寄存器的值为基地址加上 offset 的地址处
sh rs2, offset(rs1)	16	把 rs2 寄存器的低 16 位宽的值存储到以 rs1 寄存器的值为基地址加上 offset 的地址处
sw rs2, offset(rs1)	32	把 rs2 寄存器的低 32 位宽的值存储到以 rs1 寄存器的值为基地址加上 offset 的地址处
sd rs2, offset(rs1)	64	把 rs2 寄存器的值存储到以 rs1 寄存器的值为基地址加上 offset 的地址处

3.4 PC 相对寻址

程序计数器（Program Counter，PC）用来指示下一条指令的地址。为了保证 CPU 正确地执行程序的指令代码，CPU 必须知道下一条指令的地址，这就是程序计数器的作用，通常程序计数器是一个寄存器。例如，在程序执行之前，把程序的入口地址（即第一条指令的地址）设置到 PC 寄存器中。CPU 从 PC 寄存器指向的地址取值，然后依次执行。当 CPU 执行完一条指令后会自动修改 PC 的内容，使其指向下一条指令的地址。

RISC-V 指令集提供了一条 PC 相对寻址的指令 AUIPC。AUIPC 指令的格式如下。

```
auipc rd, imm
```

这条指令把 imm（立即数）左移 12 位并带符号扩展到 64 位后，得到一个新的立即数，这个新的立即数是一个有符号的立即数，再加上当前 PC 值，然后存储到 rd 寄存器中。由于新的立即数表示的是地址的高 20 位部分，并且是一个有符号的立即数，因此这条指令能寻址的范围为基于当前 PC 偏移量±2 GB，如图 3.6 所示。另外，由于这个新的立即数的低 12 位都是 0，因此它只能寻址到与 4 KB 对齐的地址。对于 4 KB 内部的寻址，需要结合其他指令（如 ADDI 指令）来完成。

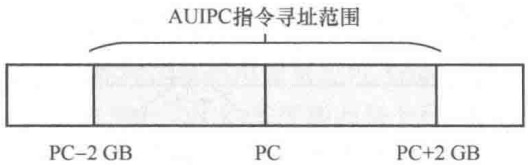


图 3.6 AUIPC 指令寻址范围

另外，还有一条指令（即 LUI 指令）与 AUIPC 类似。不同在于 LUI 指令不使用 PC 相对寻址，它仅仅把立即数左移 12 位，得到一个新的 32 立即数，带符号扩展到 64 位，并存储到 rd 寄存器中。AUIPC 和 LUI 指令的编码如图 3.7 所示。

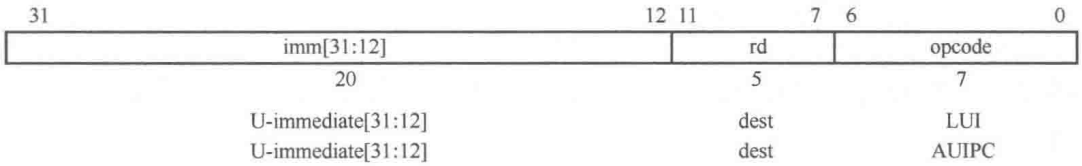


图 3.7 AUIPC 和 LUI 指令的编码

【例 3-5】 假设当前 PC 值为 0x8020 0000，分别执行如下指令，那么 a5 和 a6 寄存器的值是多少？

```
auipc   a5,0x2
lui     a6,0x2
```

a5 寄存器的值为 PC + sign_extend(0x2 << 12) = 0x8020 0000 + 0x2000 = 0x8020 2000。

a6 寄存器的值为 0x2 << 12 = 0x2000。

AUIPC 指令通常和 ADDI 联合使用来实现 32 位地址空间的 PC 相对寻址。AUIPC 指令可以寻址与被访问地址按 4 KB 对齐的地方，即被访问地址的高 20 位。ADDI 指令可以在[−2048, 2047]范围内寻址，即被访问地址的低 12 位。

如果知道了当前 PC 值和目标地址，如何计算 AUIPC 和 ADDI 指令的参数呢？如图 3.8 所示，offset 为地址 B 与当前 PC 值的偏移量，地址 B 与 4 KB 对齐的地方为地址 A，地址 A 与地址 B 的偏移量为 lo12。lo12 是有符号数的 12 位数值，取值范围为[−2048, 2047]。

根据上述信息，我们得出计算 hi20 和 lo12 的公式

```
hi20 = (offset >> 12) + offset[11]
lo12 = offset & 0xfff
```

这里特别需要注意如下几点。

- hi20 表示地址的高 20 位，用在 AUIPC 指令的 imm 操作数中。
- lo12 表示地址的低 12 位，用于 ADDI 指令的 imm 操作数中。
- 当计算 hi20 时需要加上 offset[11]，用于抵消低 12 位有符号数的影响，见例 3-6。
- lo12 是一个 12 位有符号数，取值范围为[−2048, 2047]。

下面我们使用 AUIPC 和 ADDI 指令对地址 B 进行寻址。

```
auipc a0, hi20
addi a1, a0, lo12
```

【例 3-6】 假设 PC 值为 0x8020 0000，地址 B 为 0x8020 1800。地址 B 正好在 4 KB 的正中间，地址 B 与地址 A 的偏移量为 2048 字节，而与地址 C 的偏移量为−2048 字节，如图 3.9 所示。

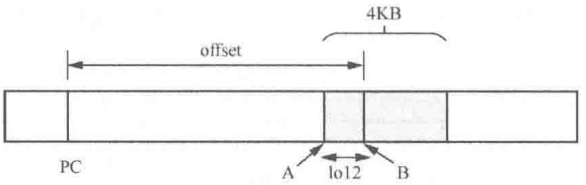


图 3.8　使用 AUIPC 和 ADDI 指令寻址

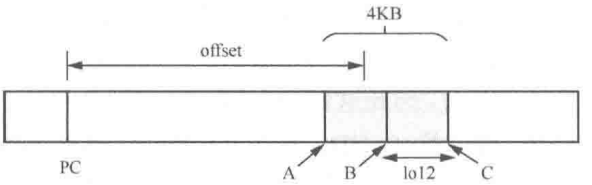


图 3.9　地址之间的关系

那我们应该使用地址 A 还是地址 C 来计算 lo12 呢？

我们应该使用地址 C 来计算 lo12。因为 lo12 是一个 12 位的有符号数，取值范围为[−2048, 2047]。若使用地址 A 来计算偏移量，lo12 就会超过取值范围。

地址 B 与 PC 值的偏移量为 0x1800。根据前面介绍的计算公式，计算 hi20 和 lo12。

```
hi20 = (0x1800 >> 12) + offset[11] = 2
lo12 = 0x800
```

因为 lo12 为 12 位有符号数，所以 0x800 表示的十进制数为−2048。下面是访问地址 B 的汇编指令。

```
auipc a0, 2
addi a1, a0, -2048
```

如果把 ADDI 指令写成如下形式，汇编器将报错，因为汇编器把字符“0x800”当成 64 位数值（即 2048）解析，它已经超过了 ADDI 指令中立即数的取值范围。

```
addi a1, a0, 0x800
```

报错日志如下。

```
AS   build_src/boot_s.o
src/boot.S: Assembler messages:
src/boot.S:6: Error: illegal operands 'addi a1,a0,0x800'
make: *** [Makefile:28: build_src/boot_s.o] Error 1
```

我们通常很少直接使用 AUIPC 指令，因为编写汇编代码时不知道当前 PC 值是多少。计算上述 hi20 和 lo12 的过程通常由链接器在重定位时完成。不过 RISC-V 定义了几条常用的伪指令，这些伪指令是基于 AUIPC 指令的。伪指令是对汇编器发出的命令，它在源程序汇编期间由汇编器处理。伪指令可以完成处理器选择、定义程序模式、定义数据、分配存储区、指示程序结束等功能。总之，伪指令可以分解为几条指令的集合。与 PC 相关的加载与存储伪指令如表 3.3 所示。

表 3.3 与 PC 相关的加载和存储伪指令

伪指令	指令组合	说明
la rd, symbol（非 PIC）	auipc rd, delta[31 : 12] + delta[11] addi rd, rd, delta[11:0]	加载符号的绝对地址。 其中 delta = symbol−pc
la rd, symbol（PIC）	auipc rd, delta[31 : 12] + delta[11] l{w|d} rd, rd, delta[11:0]	加载符号的绝对地址。 其中 delta = GOT[symbol]−pc
lla rd, symbol	auipc rd, delta[31 : 12] + delta[11] addi rd, rd, delta[11:0]	加载符号的本地地址（local address）。 其中 delta = symbol−pc
l{b|h|w|d} rd, symbol	auipc rd, delta[31 : 12] + delta[11] l{b|h|w|d} rd, delta[11:0](rd)	加载符号的内容
s{b|h|w|d} rd, symbol, rt	auipc rt, delta[31 : 12] + delta[11] s{b|h|w|d} rd, delta[11:0](rt)	存储内容到符号中。 其中 rt 为临时寄存器
li rd, imm	根据情况扩展为多条指令	加载立即数（imm）到 rd 寄存器中

表 3.3 中的 PIC 表示生成与位置无关的代码（Position Independent Code），GOT 表示全局偏移量表（Global Offset Table）。GCC 有一个“-fpic”编译选项，它在生成的代码中使用相对地址，而不是绝对地址。所有对绝对地址的访问都需要通过 GOT 实现，这种方式通常运用在共享库中。无论共享库被加载器加载到内存什么位置，代码都能正确执行，而不需要重定位（relocate）。若没有使用“-fpic”选项编译共享库，当多个程序加载此共享库时，加载器需要为每个程序重定位共享库，即根据加载到的位置重定位，这中间可能会触发写时复制机制。

【例 3-7】 观察 LA 和 LLA 指令在 PIC 与非 PIC 模式下的区别。下面是 main.c 文件和 asm.S 文件。

```
<main.c>

extern void asm_test(void);

int main(void)
{
    asm_test();

    return 0;
}


<asm.S>
.globl my_test_data
my_test_data:
    .dword 0x12345678abcdabcd

.global asm_test
asm_test:
    la t0, my_test_data
    lla t1, my_test_data

    ret
```

首先，观察非 PIC 模式。在 QEMU+RISC-V 平台上编译，使用“-fno-pic”选项关闭 PIC。

```
# gcc main.c asm.S -fno-pic -O2 -g -o test
```

通过 OBJDUMP 命令反汇编。

```
root:example_pic# objdump -d test

00000000000005f4 <my_test_data>:
 5f4:abcd                j  be6 <__FRAME_END__+0x53e>
 5f6:abcd                j  be8 <__FRAME_END__+0x540>
```

```
 5f8:5678                lw a4,108(a2)
 5fa:1234                addi   a3,sp,296

00000000000005fc <asm_test>:
 5fc:00000297            auipc  t0,0x0
 600:ff828293            addi   t0,t0,-8 # 5f4 <my_test_data>
 604:00000317            auipc  t1,0x0
 608:ff030313            addi   t1,t1,-16 # 5f4 <my_test_data>
 60c:8082                ret
```

通过反汇编可知，在非 PIC 模式下，LA 和 LLA 伪指令都是 AUIPC 与 ADDI 指令，并且都直接获取了 my_test_data 符号的绝对地址。

接下来，使用“-fpic”选项重新编译 test 程序。

```
# gcc main.c asm.S -fpic -O2 -g -o test
```

然后，通过 objdump 命令反汇编。

```
root:example_pic# objdump -d test

0000000000000634 <my_test_data>:
 634:abcd                j   c26 <__FRAME_END__+0x53e>
 636:abcd                j   c28 <__FRAME_END__+0x540>
 638:5678                lw  a4,108(a2)
 63a:1234                addi    a3,sp,296

000000000000063c <asm_test>:
 63c:00002297            auipc  t0,0x2
 640:9f42b283            ld  t0,-1548(t0) # 2030 <_GLOBAL_OFFSET_TABLE_+0x10>
 644:00000317            auipc  t1,0x0
 648:ff030313            addi   t1,t1,-16 # 634 <my_test_data>
 64c:8082                ret
```

通过反汇编可知，在 PIC 模式下，LA 伪指令是 AUIPC 和 LD 指令的集合，它会访问 GOT，然后从 GOT 中获取 my_test_data 符号的地址。而 LLA 伪指令是 AUIPC 和 ADDI 指令的集合，直接获取 my_test_data 符号的绝对地址。

总之，在非 PIC 模式下，LLA 和 LA 伪指令的行为相同，获取符号的绝对地址；而在 PIC 模式下，LA 指令从 GOT 中获取符号的地址，而 LLA 伪指令获取符号的绝对地址。

【例 3-8】　在例 3-7 的基础上修改 asm.S 汇编文件，目的是观察 LI 伪指令。

```
<asm.S>

.global asm_test
asm_test:

    li t0, 0xffffff080200000
    ret
```

在 QEMU+RISC-V 平台上编译。

```
# gcc main.c asm.S -O2 -g -o test
```

通过 OBJDUMP 命令反汇编。

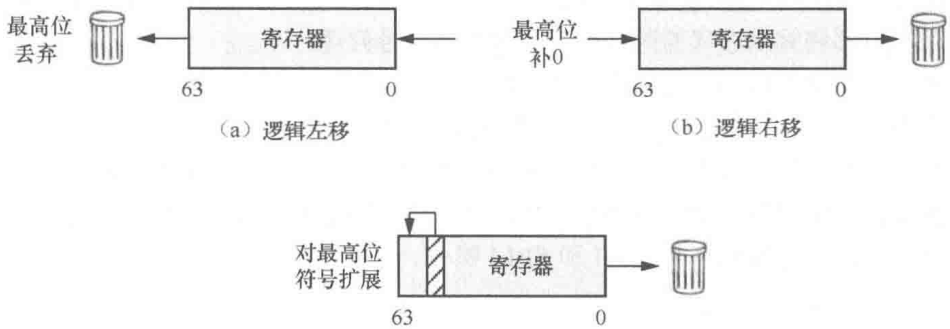
```
root:example_pic# objdump -d test

00000000000005fc <asm_test>:
5fc:72e1                lui  t0,0xffff8
5fe:4012829b            addiw  t0,t0,1025
602:02d6                slli  t0,t0,0x15
604:8082                ret
```

从上面的反汇编结果可知，上述的 LI 伪指令由 LUI、ADDIW 和 SLLI 这 3 条指令组成。

3.5 移位操作

常见的移位操作如下。

- sll：逻辑左移（shift left logical），最高位会丢弃，最低位补 0，如图 3.10（a）所示。
- srl：逻辑右移（shift right logical），最高位补 0，最低位会丢弃，如图 3.10（b）所示。
- sra：算术右移（shift right arithmetic），最低位会丢弃，最高位会按照符号进行扩展，如图 3.10（c）所示。



（a）逻辑左移　（b）逻辑右移

（c）算术右移

图 3.10　移位操作

常见的移位指令如表 3.4 所示。

表 3.4　常见的移位指令

指令	指令格式	说明
sll	sll rd, rs1, rs2	逻辑左移指令。 把 rs1 寄存器左移 rs2 位，结果写入 rd 寄存器中
slli	slli rd, rs1, shamt	立即数逻辑左移指令。 把 rs1 寄存器左移 shamt 位，结果写入 rd 寄存器中
slliw	slliw rd, rs1, shamt	立即数逻辑左移指令。 截取 rs1 寄存器的低 32 位作为新的源操作数，然后左移 shamt 位，根据结果进行符号扩展后写入 rd 寄存器
sllw	sllw rd, rs1, rs2	逻辑左移指令。 截取 rs1 寄存器的低 32 位作为新的源操作数，然后左移 rs2 位（取 rs2 寄存器低 5 位的值），根据结果进行符号扩展后写入 rd 寄存器
sra	sra rd, rs1, rs2	算术右移指令。 把 rs1 寄存器右移 rs2 位，根据 rs1 寄存器的旧值，进行符号扩展后写入 rd 寄存器中
srai	srai rd, rs1, shamt	立即数算术右移指令。 把 rs1 寄存器右移 shamt 位，进行符号扩展后写入 rd 寄存器中
sraiw	sraiw rd, rs1, shamt	立即数算术右移指令。 截取 rs1 寄存器的低 32 位作为新的源操作数，然后右移 shamt 位，根据新的源操作数进行符号扩展后写入 rd 寄存器中
sraw	sraw rd, rs1, rs2	算术右移指令。 截取 rs1 寄存器的低 32 位作为新的源操作数，然后右移 rs2 位（取 rs2 寄存器低 5 位的值），根据新的源操作数进行符号扩展后写入 rd 寄存器中
srl	srl rd, rs1, rs2	逻辑右移指令。 把 rs1 寄存器右移 rs2 位，进行零扩展后写入 rd 寄存器中
srli	srli rd, rs1, shamt	立即数逻辑右移指令。 把 rs1 寄存器右移 shamt 位，进行零扩展后写入 rd 寄存器中

续表

指令	指令格式	说明
srliw	srliw rd, rs1, shamt	立即数逻辑右移指令。 截取 rs1 寄存器的低 32 位作为新的源操作数，然后右移 shamt 位，进行符号扩展后写入 rd 寄存器中
srlw	srlw rd, rs1, rs2	逻辑右移指令。 截取 rs1 寄存器的低 32 位作为新的源操作数，然后右移 rs2 位（取 rs2 寄存器低 5 位的值），进行符号扩展后写入 rd 寄存器中

关于移位操作指令有三点需要注意。

- RISC-V 指令集里没有单独设置一个算术左移的指令，因为 sll 指令会把最高位丢弃。
- 逻辑右移和算术右移的区别在于是否考虑符号问题。

  例如，源操作数为二进制数 10 1010 1010。

  逻辑右移一位，变成[0]1 0101 0101（最高一位永远补 0）。

  算术右移一位，变成[1]1 0101 0101（对于算术右移，需要按照源操作数进行符号扩展）。
- 在 RV64 指令集中，SLL、SRL 以及 SRA 指令只使用 rs2 寄存器中低 6 位数据做移位操作。

【例 3-9】　如下代码使用了 SRAI 和 SRLI 指令。

```
li t0, 0x8000008a00000000
srai a1, t0, 1
srli t1, t0, 1
```

在上述代码中，SRAI 是立即数算术右移指令，把 0x8000 008A 0000 0000 右移一位并且根据源二进制数的最高位需要进行符号扩展，最后结果为 0xC000 0045 0000 0000。SRLI 是立即数逻辑右移指令，把 0x8000 008A 0000 0000 右移一位并且在最高位补 0，最后结果为 0x4000 0045 0000 0000。

【例 3-10】　如下代码使用了 SRAIW 和 SRLIW 指令。

```
1   li t0, 0x128000008a
2   sraiw a2, t0, 1
3   srliw a3, t0, 1
4
5   li t0, 0x124000008a
6   sraiw a4, t0, 1
```

在第 2 行中，使用立即数算术右移指令，截取 t0 寄存器低 32 位的值（0x8000 008A）作为新的源操作数，然后右移一位等于 0x4000 0045，根据新的源二进制数的最高位需要进行符号扩展，最后结果为 0xFFFF FFFF C000 0045。

在第 3 行中，使用立即数逻辑右移指令，截取 t0 寄存器低 32 位的值（0x8000 008A）作为新的源操作数，然后右移一位并且进行符号扩展，最后结果为 0x4000 0045。

在第 6 行中，使用立即数算术右移指令，截取 t0 寄存器低 32 位的值（0x4000 008A）作为新的源操作数，然后右移一位等于 0x2000 0045，根据新的源二进制数的最高位需要进行符号扩展，最后结果为 0x2000 0045。

【例 3-11】　下面的示例代码使用了 SLLIW 指令。

```
1   li t0, 0x128000008a
2   slliw a3, t0, 1
3
4   li t0, 0x122000008a
5   slliw a4, t0, 1
6
7   li t0, 0x124000008a
8   slliw a5, t0, 1
```

在第 2 行中，使用立即数逻辑左移指令，截取 t0 寄存器低 32 位的值（0x8000 008A），然后左移一位，得到 0x114，最后结果为 0x114。

在第 5 行中，截取 t0 寄存器低 32 位的值（0x2000 008A），左移一位后等于 0x4000 0114，最后结果为 0x4000 0114。

在第 8 行中，截取 t0 寄存器的低 32 位的值（0x4000 008A），左移一位后等于 0x800 00114，由于最高位为 1，需要进行符号扩展，最后这条 SLLIW 指令的执行结果为 0xFFFF FFFF 8000 0114。

3.6 位操作指令

RV64I 指令集提供与（and）、或（or）以及异或（xor）3 种位操作指令，如表 3.5 所示。

表 3.5 位操作指令

指令	指令格式	说明
and	and rd, rs1, rs2	与操作指令。 对 rs1 和 rs2 寄存器按位进行与操作，把结果写入 rd 寄存器中
andi	andi rd, rs1, imm	与操作指令。 对 rs1 寄存器和 imm（立即数）按位进行与操作，把结果写入 rd 寄存器中
or	or rd, rs1, rs2	或操作指令。 对 rs1 寄存器和 rs2 寄存器按位进行或操作，把结果写入 rd 寄存器中
ori	ori rd, rs1, imm	或操作指令。 对 rs1 寄存器和 imm 按位进行或操作，把结果写入 rd 寄存器中
xor	xor rd, rs1, rs2	异或操作指令。 对 rs1 寄存器和 rs2 寄存器按位进行异或操作，把结果写入 rd 寄存器中
xori	xori rd, rs1, imm	异或操作指令。 对 rs1 寄存器和 imm 按位进行异或操作，把结果写入 rd 寄存器中
not	not rd, rs	按位取反指令。 对 rs 寄存器按位进行取反操作，把结果写入 rd 寄存器中。该指令是伪指令，内部使用“xori rd, rs, -1”

异或操作的真值表如下。

0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0

从上述真值表可以发现 3 个有意思的特点。

- 0 异或任何数 = 任何数。
- 1 异或任何数 = 任何数取反。
- 任何数异或自己都等于 0。

利用上述特点，异或操作有如下几个非常常用的场景。

- 使某些特定的位翻转。例如，若想把 0b1010 0001 的第 1 位和第 2 位翻转，则可以将该数与 0b0000 0110 进行按位异或运算。

```
10100001 ^ 00000110 = 10100111
```

- 交换两个数。例如，要交换两个整数 a=0b1010 0001 和 b=0b0000 0110 的值，可通过下列语句实现。

```
a = a^b;      //a=1010 0111
b = b^a;      //b=1010 0001
a = a^b;      //a=0000 0110
```

- 在汇编代码里把变量设置为 0。

```
xor x1, x1
```

- 判断两个数是否相等。

```
bool is_identical(int a, int b)
{
    return ((a ^ b) == 0);
}
```

3.7 算术指令

RV64I 指令集只提供基础的算术指令，即加法和减法指令，如表 3.6 所示。

表 3.6　基础的算术指令

指令	指令格式	说明
add	add rd, rs1, rs2	加法指令。 将 rs1 寄存器的值与 rs2 寄存器的值相加，把结果写入 rd 寄存器中
addi	addi rd, rs1, imm	加法指令。 将 rs1 寄存器与 imm 相加，把结果写入 rd 寄存器中
addw	addw rd, rs1, rs2	加法指令。 截取 rs1 和 rs2 寄存器的低 32 位数据作为源操作数并且相加，结果只截取低 32 位，最后进行符号扩展并写入 rd 寄存器中
addiw	addiw rd, rs1, imm	加法指令。 截取 rs1 寄存器的低 32 位数据作为源操作数，加上 imm，对结果进行符号扩展并写入 rd 寄存器中
sub	sub rd, rs1, rs2	减法指令。 将 rs1 寄存器的值减去 rs2 寄存器的值，把结果写入 rd 寄存器中
subw	subw rd, rs1, rs2	减法指令。 截取 rs1 和 rs2 寄存器的低 32 位数据作为源操作数，然后新的 rs1 值减去新的 rs2 值，结果只截取低 32 位，最后进行符号扩展并写入 rd 寄存器中

ADD 指令的编码如图 3.11 所示。

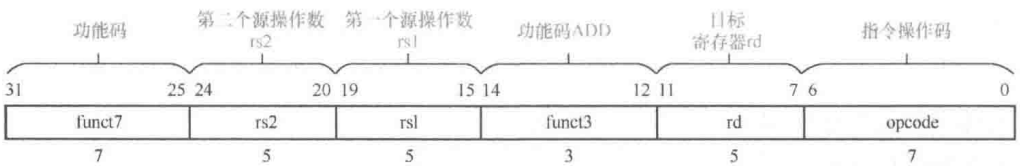


图 3.11　ADD 指令的编码

ADDI 指令的编码如图 3.12 所示。其中，imm 是一个 12 位的带符号扩展立即数，取值范围为[−2048, 2047]。

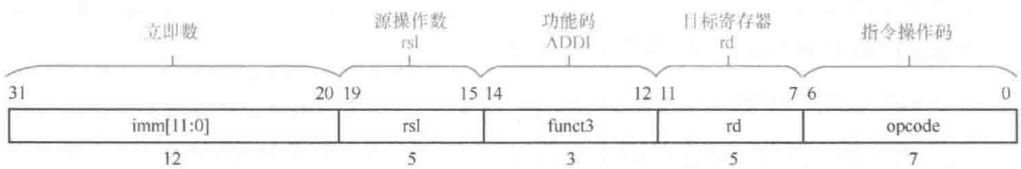


图 3.12　ADDI 指令的编码

【例 3-12】 从图 3.12 可知，ADDI 指令中的立即数是 12 位有符号数，那么下面两条指令哪一条是非法指令？

```
addi a1, t0, 0x800
addi a1, t0, 0xfffffffffffff800
```

上述两条指令中，第一条指令为非法指令。编译器会提示如下警告。

```
src/asm_test.S: Assembler messages:
src/asm_test.S:12: Error: illegal operands 'addi a1,t0,0x800'
```

在第一条指令中，我们想传递数值−2048 给 ADDI 指令。既然 ADDI 指令中的立即数为带符号扩展的 12 位立即数，0x800 表示 Bit[11]为 1，那么它为什么是一个非法的立即数呢？

其实，在 GNU AS 中，0x800 被看作一个数值为 2048 的 64 位无符号数，而不是 12 位宽的带符号扩展的立即数。如果想表示“−2048”立即数，我们需要使用 0xFFFF FFFF FFFF F800，因为汇编器中的立即数是按照处理器的位宽解析的。例如，64 位处理器使用 64 位数据，32 位处理器使用 32 位数据，而不是按照指令编码的 12 位数据。

综上所述，ADDI 指令中的立即数取值范围为[−2048, 2047]。

【例 3-13】 下面的代码使用了 ADD 指令。

```
1   li t0, 0x140200000
2   li t1, 0x40000000
3
4   addi a1, t0, 0x80
5   addiw a2, t0, 0x80
6
7   add a3, t0, t1
8   addw a4, t0, t1
```

在第 4 行中，ADDI 指令把 t0 寄存器的值与 0x80 相加，把结果写入 a1 寄存器中，a1 寄存器的值为 0x140200080。

在第 5 行中，ADDIW 指令首先截取 t0 寄存器的低 32 位，得到 0x4020 0000，然后与立即数 0x80 相加，最后做符号扩展并存入 a2 寄存器中，a2 寄存器的值为 0x4020 0080。

在第 7 行中，ADD 指令把 t0 寄存器的值加上 t1 寄存器的值，把结果写入 a3 寄存器，a3 寄存器的值为 0x18020 0000。

在第 8 行中，首先，截取 t0 寄存器的低 32 位，得到 0x4020 0000；接着，截取 t1 寄存器的低 32 位，得到 0x4000 0000；然后，相加，结果为 0x8020 0000；最后，做符号扩展并写入 a4 寄存器中，a4 寄存器的值为 0xFFFF FFFF 8020 0000。

【例 3-14】 下面的代码使用了 SUB 指令。

```
1   li t0, 0x180200000
2   li t1, 0x200000
3
4   sub a0, t0, t1
5   subw a1, t0, t1
```

在第 4 行中，SUB 指令直接用 t0 寄存器的值减去 t1 寄存器的值，结果为 0x18000 0000。

在第 5 行中，SUBW 指令，首先，截取 t0 寄存器的低 32 位，得到 0x8020 0000；然后，减去 t1 寄存器的值，得到 0x8000 0000；最后，做符号扩展，结果为 0xFFFF FFFF 8000 0000。

3.8 比较指令

RV64I 指令集支持 4 条基本比较指令，如表 3.7 所示。

表 3.7　　基本比较指令

指令	指令格式	说明
slt	slt rd, rs1, rs2	有符号数比较指令。 比较 rs1 和 rs2 寄存器的值，如果 rs1 寄存器中的值小于 rs2 寄存器中的值，向 rd 寄存器写入 1；否则，写入 0
sltu	sltu rd, rs1, rs2	等同于 slt 指令，区别在于 rs1 寄存器中的值和 rs2 寄存器中的值为无符号数
slti	slti rd, rs1, imm	比较指令。 比较 rs1 寄存器的值与 imm，如果 rs1 寄存器中的值小于 imm，向 rd 寄存器写入 1；否则，写入 0
sltiu	sltiu rd, rs1, imm	无符号数与立即数比较指令。 如果 rs1 寄存器中的值小于 imm，向 rd 寄存器写入 1；否则，写入 0

RV64I 指令集只支持小于比较指令，为了方便程序员编写汇编代码，RISC-V 提供几条常用的比较伪指令，如表 3.8 所示。

表 3.8　　比较伪指令

伪指令	伪指令格式	说明
sltz	sltz rd, rs1	小于 0 则置位指令。 如果 rs1 寄存器的值小于 0，向 rd 寄存器写入 1；否则，写入 0
snez	snez rd, rs1	不等于则置位指令。 如果 rs1 寄存器的值不等于 0，向 rd 寄存器写入 1；否则，写入 0
seqz	seqz rd, rs1	等于 0 则置位指令。 如果 rs1 寄存器的值等于 0，向 rd 寄存器写入 1；否则，写入 0
sgtz	sgtz rd, rs1	大于 0 则置位指令。 如果 rs1 寄存器的值大于 0，向 rd 寄存器写入 1；否则，写入 0

3.9 无条件跳转指令

RV64I 指令集支持的无条件跳转指令如表 3.9 所示。

表 3.9　　无条件跳转指令

指令	指令格式	说明
jal	jal rd, offset	跳转与链接指令。 跳转到数值为 PC + offset 的地址中。然后，把返回地址（PC + 4）保存到 rd 寄存器中。offset 是 21 位有符号数。跳转范围是大约当前 PC 值偏移±1 MB，即 PC−0x10 0000～PC+0xFF FFFE
jalr	jalr rd, offset(rs1)	使用寄存器的跳转指令。 跳转到以 rs1 寄存器的值为基地址且偏移 offset 的地址处，然后把返回地址（PC + 4）保存到 rd 寄存器中。offset 是 12 位有符号数。偏移范围为−2048～2047

JAL（Jump And Link，跳转与链接）指令使用 J 类型的指令编码，如图 3.13 所示。其中，操作数 offset[20:1]由指令编码的 Bit[31:12]构成，它默认是 2 的倍数，因此它的跳转范围为当前 PC 值偏移±1 MB。另外，把返回地址（即 PC + 4）存储到 rd 寄存器中。根据 RISC-V 函数调用规则，如果把返回地址存储到 ra 寄存器中，则可以实现函数返回。

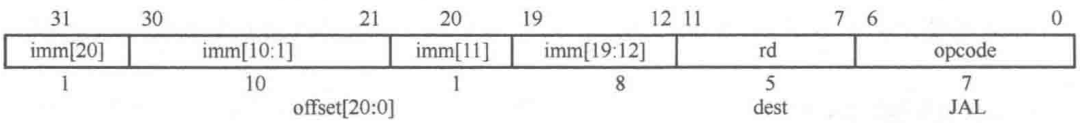


图 3.13　JAL 指令的编码

JALR（Jump And Link Register，跳转与链接寄存器）指令使用 I 类型指令编码，如图 3.14 所

示。要跳转的地址由 rs1 寄存器和 offset 操作数组成。其中，offset 是一个 12 位的有符号立即数。

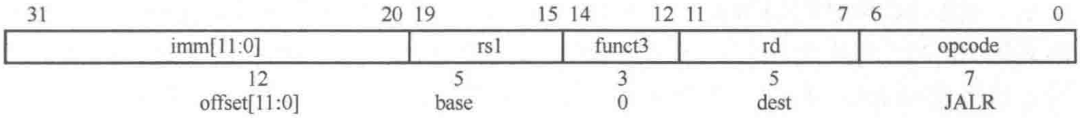


图 3.14 JALR 指令的编码

【例 3-15】 假设执行如下各条指令时当前 PC 值为 0x8020 0000，下面指令哪些是非法指令？

```
1   jal a0, 0x800fffff
2   jal a0, 0x80300000
```

编译上述两条指令都会出错，汇编器会输出如下错误消息。

```
AS   build_src/boot_s.o
build_src/boot_s.o: in function '.L0 ':
/home/rlk/rlk/riscv_trainning/benos/src/boot.S:10:(.text.boot+0x0): relocation truncated
 to fit: R_RISCV_JAL against '*UND*'
```

上述两条指令都超过了 JAL 指令的跳转范围。以 PC 值为 0x8020 0000，JAL 指令的跳转范围为[0x8010 0000, 0x802F FFFE]。

为了方便程序员编写汇编代码，RISC-V 根据 JAL 和 JALR 指令扩展了多条无条件跳转伪指令，如表 3.10 所示。

表 3.10 无条件跳转伪指令

伪指令	指令组合	说明
j label	jal x0, offset	跳转到 label 处，不带返回地址
jal label	jal ra, offset	跳转到 label 处，返回地址存储在 ra 寄存器中
jr rs	jalr x0, 0(rs)	跳转到 rs 寄存器中的地址处，不带返回地址
jalr rs	jalr ra, 0(rs)	跳转到 rs 寄存器中的地址处，返回地址存储在 ra 寄存器中
ret	jalr x0, 0(ra)	从 ra 寄存器中获取返回地址，并返回。常用于子函数返回
call func	auipc ra, offset[31:12]+offset[11] jalr ra, offset[11:0](ra)	调用子函数 func，返回地址保存到 ra 寄存器中
tail func	auipc x6, offset[31:12]+offset[11] jalr x0, offset[11:0](x6)	调用子函数 func，不保存返回地址

表中的 label 和 func 表示汇编符号，在链接重定位过程中由当前 PC 值与符号地址来共同确定指令编码中 offset 字段的值。关于汇编符号，见 5.3.2 节，关于链接重定位，见 6.4 节。

3.10 条件跳转指令

RV64I 支持的条件跳转指令如表 3.11 所示。

表 3.11 条件跳转指令

指令	指令格式	说明
beq	beq rs1, rs2, label	如果 rs1 和 rs2 寄存器的值相等，则跳转到 label 处
bne	bne rs1, rs2, label	如果 rs1 和 rs2 寄存器的值不相等，则跳转到 label 处
blt	blt rs1, rs2, label	如果 rs1 寄存器的值小于 rs2 寄存器的值，则跳转到 label 处
bltu	bltu rs1, rs2, label	与 blt 指令类似，只不过 rs1 寄存器的值和 rs2 的值为无符号数
bgt	bgt rs1, rs2, label	如果 rs1 寄存器的值大于 rs2 寄存器的值，则跳转到 label 处
bgtu	bgtu rs1, rs2, label	与 bgt 指令类似，只不过 rs1 寄存器的值和 rs2 的值为无符号数
bge	bge rs1, rs2, label	如果 rs1 寄存器的值大于或等于 rs2 寄存器的值，则跳转到 label 处
bgeu	bgeu rs1, rs2, label	与 bge 指令类似，只不过 rs1 寄存器和 rs2 寄存器的值为无符号数

上述条件跳转指令都采用 B 类型的指令编码，如图 3.15 所示，其中操作数 offset 表示 label 的地址基于当前 PC 地址的偏移量。操作数 offset 是 13 位有符号立即数。其中，offset[12:1]由指令编码的 Bit[31:25]以及 Bit[11:7]共同构成，offset[0]默认为 0，offset 默认是 2 的倍数，它的最大寻址范围是−4 KB～4 KB，因此上述指令只能跳转到当前 PC 地址±4 KB 的范围。若跳转地址大于上述范围，编译器不会报错，因为链接器在链接重定位时会做链接器松弛优化，选择合适的跳转指令。指令编码中的 offset 值是在链接重定位过程中由当前 PC 值与 label 的地址共同来确定的。为了编程方便，我们通常使用汇编符号完成条件跳转指令。

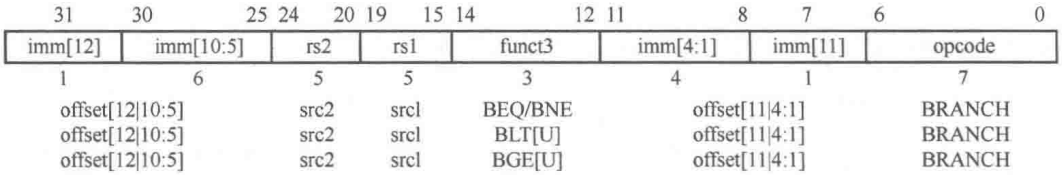


图 3.15　条件跳转指令编码

【例 3-16】　在下面的汇编代码中，当 x1 与 x2 寄存器的值不相等时，跳转到 L1 标签处。

```
main:
    addi x1, x0, 33
    addi x2, x0, 44
    bne x1, x2, .L1
.L0:
    li      a5,-1
.L1:
    mv      a0,a5
    ret
```

使用 riscv64-linux-gnu-as 和 riscv64-linux-gnu-objdump 工具编译反汇编代码。

```
$ riscv64-linux-gnu-as my_asm.s -o my_asm
$ riscv64-linux-gnu-objdump -d my_asm

Disassembly of section .text:

0000000000000000 <main>:
   0:02100093          li  ra,33
   4:02c00113          li  sp,44
   8:00209463          bne  ra,sp,10 <.L1>
   c:fff00793          li  a5,-1

0000000000000010 <.L1>:
  10:00078513          mv  a0,a5
      14:00008067          ret
```

从反汇编结果可知，BNE 指令的编码值为 0x0020 9463，指令地址为 0x8，L1 标签处的地址为 0x10，对照图 3.15 从指令编码值可计算出 offset 值为 8，这符合我们的预期。

为了方便程序员编写汇编代码，RISC-V 又扩展了多条伪指令，如表 3.12 所示。

表 3.12　　条件跳转伪指令

伪指令	指令组合	判断条件
beqz rs, label	beq rs, x0, label	rs == 0
bnez rs, label	bne rs, x0, label	rs != 0
blez rs, label	bge x0, rs, label	rs <= 0
bgez rs, label	bge rs, x0, label	rs >= 0
bltz rs, label	blt rs, x0, label	rs < 0
bgtz rs, label	blt x0, rs, label	rs > 0
bgt rs, rt, label	blt rt, rs, label	rs > rt

续表

伪指令	指令组合	判断条件
ble rs, rt, label	bge rt, rs, label	rs <= rt
bgtu rs, rt, label	bltu rt, rs, label	rs > rt（无符号数比较）
bleu rs, rt, label	bleu rs, rt, label	rs <= rt（无符号数比较）

3.11 CSR 指令

RISC-V 体系结构不仅提供了一组系统寄存器（详见 1.3.2 节），还提供了一组指令来访问这些系统寄存器。CSR 指令的编码如图 3.16 所示。

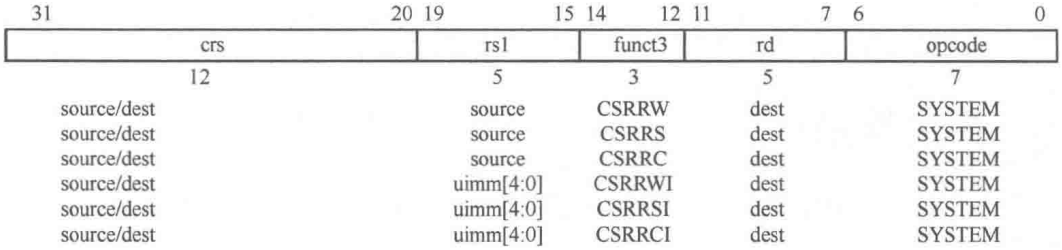


图 3.16 CSR 指令的编码

常用的 CSR 指令如表 3.13 所示。

表 3.13 常用的 CSR 指令

CSR 指令	指令格式	说明
csrrw	csrrw rd, csr, rs1	原子地交换 CSR 和 rs1 寄存器的值。 读取 CSR 的旧值，将其零扩展到 64 位，然后写入 rd 寄存器中。与此同时，rs1 寄存器的旧值将被写入 CSR 中
csrrs	csrrs rd, csr, rs1	原子地读 CSR 的值并且设置 CSR 中相应的位。 指令读取 CSR 的旧值，将其零扩展到 64 位，然后写入 rd 寄存器中。与此同时，以 rs1 寄存器的值作为掩码，设置 CSR 相应的位
csrrc	csrrc rd, csr, rs1	原子地读 CSR 的值并且清除 CSR 中相应的位。 指令读取 CSR 的旧值，将其零扩展到 64 位，然后写入 rd 寄存器中。与此同时，以 rs1 寄存器的值作为掩码，清除 CSR 中相应的位
csrrwi	csrrwi rd, csr, uimm	作用与 csrrw 指令类似，区别在于使用 5 位无符号立即数替代 rs1
csrrsi	csrrsi rd, csr, uimm	作用与 csrrs 指令类似，区别在于使用 5 位无符号立即数替代 rs1
csrrci	csrrci rd, csr, uimm	作用与 csrrc 指令类似，区别在于使用 5 位无符号立即数替代 rs1

【例 3-17】 下面的代码使用了 CSR 指令。

```
1   csrrw t0, sscratch, tp
2   csrrw tp, sscratch, tp
3   csrrs t0, sstatus, t1
```

在第 1 行中，交换 tp 和 sscratch 寄存器的值，即读取 sscratch 寄存器的旧值并写入 t0 寄存器中。与此同时，把 tp 寄存器的旧值写入 sscratch 寄存器中。这用 C 语言伪代码 t0 = sscratch，sscratch = tp 表示。

在第 2 行中，源寄存器和目标寄存器是同一个寄存器，这很容易迷惑人。这条指令先读取 sscratch 寄存器的旧值并写入 tp 寄存器，与此同时，把 tp 寄存器的旧值写入 sscratch 寄存器。这用 C 语言伪代码 tp = sscratch，sscratch = tp 表示。

在第 3 行中，把 sstatus 寄存器的旧值读取并写入 t0 寄存器中。与此同时，以 t1 寄存器的值

为掩码，设置 sstatus 寄存器中相应的位。这用 C 语言伪代码 t0 = sstatus，sstatus |= t1 表示。

CSR 指令中的目标寄存器和源寄存器可以使用 x0 寄存器，从而组合成常用的 CSR 伪指令，如表 3.14 所示。

表 3.14　　常用的 CSR 伪指令

伪指令	指令组合	说明
csrr rd, csr	csrrs rd, csr, x0	读取 CSR 的值
csrw csr, rs	csrrw x0, csr, rs	写 CSR 的值
csrs csr, rs	csrrs x0, csr, rs	设置 CSR 的字段（csr |= rs）
csrc csr, rs	csrrc x0, csr, rs	清除 CSR 的字段（csr &=~rs）
csrwi csr, imm	csrrwi x0, csr, imm	把 imm 写入 CSR 中
csrsi csr, imm	csrrsi x0, csr, imm	设置 CSR 的字段（csr |= imm）
csrci csr, imm	csrrci x0, csr, imm	清除 CSR 的字段（csr &=~imm）

3.12 寻址范围

在使用 RISC-V 汇编指令编写代码时需要特别注意指令的寻址范围。RISC-V 支持长距离寻址和短距离寻址。

- 长距离寻址：通过 AUIPC 可以实现基于当前 PC 偏移量±2 GB 范围的寻址，这种寻址方式叫作 PC 相对寻址，不过 AUIPC 指令只能寻址到按 4 KB 对齐的地方。
- 短距离寻址：有些指令（如 ADDI 指令、加载和存储指令等）可以实现基于基地址短距离寻址，即寻址范围为-2048～2047，这个范围正好是 4 KB 大小内部的寻址范围。

长距离寻址和短距离寻址结合可以实现基于当前 PC 偏移量±2 GB 范围的任意地址的寻址。

对于跳转指令，RISC-V 也支持长跳转模式和短跳转模式，这些模式在链接器松弛优化中会用到。

- 长跳转模式：通过 AUIPC 与 JALR 指令实现基于当前 PC 偏移量±2 GB 的范围跳转。
- 短跳转模式：JAL 指令可以实现基于当前 PC 偏移量±1 MB 的范围跳转。

3.13 陷阱：为什么 ret 之后就进入死循环

在汇编代码里使用 call 伪指令来调用子函数，不过处理不当会导致程序崩溃。因为使用 call 伪指令跳转到子函数时会修改 ra 寄存器（返回地址）的值，把当前 PC+4 写入 ra 寄存器中。这就把父函数的返回地址给修改了，导致父函数调用 ret 指令返回时崩溃。

【例 3-18】 C 语言中的 asm_test()函数调用 branch_test()汇编函数，branch_test()汇编函数调用 add_test()子函数。

汇编文件 asm.S 如下。

```
<asm.S>

1    add_test:
2        add a0, a0, a1
3        nop
4        ret
5
6    .globl branch_test
7    branch_test:
8        li a0, 1
```

```
9        li a1, 2
10       /*调用 add_test()子函数*/
11       call add_test
12       nop
13       ret
```

asm_test()函数如下。

```
1   void asm_test(void)
2   {
3       /*调用汇编函数*/
4       branch_test();
5   }
```

在上述例子中，add_test()函数通过 RET 指令返回时就陷入死循环。

下面是分析的过程。

在第 6 行中，假设程序执行到 branch_test()函数时，上一级函数 asm_test()的返回地址（ra 寄存器的值）为 0x8020 0160。

在第 11 行中，调用子函数 add_test()，此时 PC 值为 0x8020 0192。

在第 1 行中，程序执行 add_test()子函数，此时返回地址（ra 寄存器的值）被改写为 0x8020 0196。因为 ra 寄存器的值为调用子函数的 call 指令的 PC 值加上 4。

子函数 add_test()执行 RET 返回第 11 行，此时 ra 寄存器的值为 0x8020 0196。而对于父函数 branch_test()来说，它的返回地址应该为 0x8020 0160，这样才能正确返回，而 0x8020 0196 地址对应 branch_test()汇编函数的 NOP 指令（第 12 行），因此程序进入死循环（见图 3.17）。

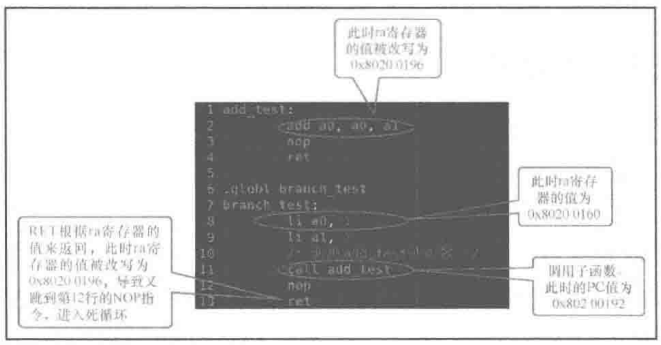


图 3.17 CALL 指令导致程序死循环

下面是通过 riscv64-linux-gnu-objdump 命令得到的反汇编代码以及对应的地址。

```
000000080200154 <asm_test>:
    80200154:1141                    addi  sp,sp,-16
    80200156:e406                    sd  ra,8(sp)
    80200158:e022                    sd  s0,0(sp)
    8020015a:0800                    addi  s0,sp,16
    8020015c:032000ef                jal  ra,8020018e <branch_test>
    80200160:0001                    nop
    80200162:60a2                    ld  ra,8(sp)
    80200164:6402                    ld  s0,0(sp)
    80200166:0141                    addi  sp,sp,16
    80200168:8082                    ret

0000000080200188 <add_test>:
    80200188:952e                    add  a0,a0,a1
    8020018a:0001                    nop
    8020018c:8082                    ret

000000008020018e <branch_test>:
    8020018e:4505                    li  a0,1
    80200190:4589                    li  a1,2
```

```
    80200192:ff7ff0ef                jal  ra,80200188 <add_test>
=>  80200196:0001                    nop
    80200198:8082                    ret
```

解决办法是在遇到嵌套调用函数时需要在父函数里把 ra 寄存器保存到一个临时寄存器。在父函数 ret 返回之前，先从临时寄存器中恢复 ra 寄存器的值，再执行 RET 以返回。如图 3.18 所示，使用 MV 指令先把 ra 寄存器的值存储到 x18 寄存器中，然后在父函数返回之前从 x18 寄存器中取回 ra 寄存器的内容。

总之，这里涉及 ra 寄存器的保存，比较通用的做法是在函数入口把 ra 寄存器的值都保存到栈中，在函数返回时从栈中恢复 ra 寄存器的值，如图 3.19 所示。

```
1  add_test:
2          add a0, a0, a1
3          nop
4          ret
5
6  .globl branch_test
7  branch_test:
8          /*把ra寄存器的值保存临时寄存器x18中
9            以免调用子函数时被破坏*/
10         mv x18, ra
11
12         li a0, 1
13         li a1, 2
14         /* 调用add_test子函数 */
15         call add_test
16         nop
17
18         /* 从x18中恢复ra返回地址*/
19         mv ra, x18
20         ret
```

图 3.18　把返回地址保存到临时寄存器中

```
.globl branch_test
branch_test:
        /*把ra寄存器的值保存到栈里*/
        addi    sp,sp,-8
        sd      ra,(sp)

        li a0, 1
        li a1, 2
        /*调用add_test子函数 */
        call add_test
        nop

        /*从栈中恢复ra*/
        ld      ra,(sp)
        addi    sp,sp,8
        ret
```

图 3.19　把返回地址保存到栈里

3.14 实验

3.14.1　实验 3-1：熟悉加载指令

1. 实验目的

熟悉加载指令的使用。

2. 实验要求

请在 BenOS 里做如下练习，以新建一个汇编文件。

（1）使用 LI 指令把 0x8020 0000 加载到 a0 寄存器。使用 LI 指令把立即数 16 加载到 a1 寄存器。

（2）从 0x8020 0000 地址中读取 4 字节的数据。

（3）从 0x8020 0010 地址中读取 8 字节的数据。

（4）下面的 LUI 指令最后的执行结果是多少？

```
lui t0, 0x8034f
lui t1, 0x400
```

编写汇编代码，并使用 GDB 单步调试。

提示：在 GDB 中，使用 x 命令读取内存地址的值，然后和寄存器的值进行比较以验证是否正确。

3.14.2　实验 3-2：PC 相对地址寻址

1. 实验目的

熟悉 PC 相对地址寻址的加载和存储指令。

2. 实验要求

在 BenOS 里做如下练习。

在汇编文件中输入如下代码。

```
#define MY_OFFSET -2048

auipc t0, 1
addi t0, t0, MY_OFFSET
ld t1, MY_OFFSET(t0)
```

t0 和 t1 寄存器的值分别是多少？

请使用 GDB 单步调试并观察对应内存单元与寄存器的变化。

3.14.3 实验 3-3：memcpy()函数的实现

1. 实验目的

熟悉加载与存储指令。

2. 实验要求

在 BenOS 里做如下练习。

实现一个小的 memcpy()汇编函数，从 0x802 0000 地址复制 32 字节到 0x8021 0000 地址处，并使用 GDB 比较数据是否复制正确。

3.14.4 实验 3-4：memset()函数的实现

1. 实验目的

熟悉 RISC-V 指令。

2. 实验要求

在 BenOS 里做如下练习。

memset()函数的 C 语言原型如下。假设内存地址 s 按 16 字节对齐，count 也按 16 字节对齐。请使用 RISC-V 汇编指令实现这个函数。

（1）memset()函数的原型如下。

```
void *memset(void *s, int c, size_t count)
{
    char *xs = s;

    while (count--)
        *xs++ = c;
    return s;
}
```

（2）假设内存地址 s 以及 count 不按 16 字节对齐，如 memset(0x80210004, 0x55, 102)，请继续优化 memset()函数。

3.14.5 实验 3-5：条件跳转指令 1

1. 实验目的

熟练掌握条件跳转指令。

2. 实验要求

在 BenOS 里做实验。编写一个汇编函数，实现比较与返回功能。下面是该函数的 C 语言伪代码。

```
unsigned long compare_and_return(unsigned long a, unsigned long b)
{
      if (a >= b)
          return 0;
```

```
        else
            return 0xffffffffffffffff;
}
```

3.14.6　实验 3-6：条件跳转指令 2

1. 实验目的

进一步掌握条件跳转指令。

2. 实验要求

请在 BenOS 里做如下练习。

请使用条件选择指令实现如下 C 语言函数。

```
unsigned long sel_test(unsigned long a, unsigned long b)
{
    if (a == 0)
            return b+2;
    else
            return b-1;
}
```

3.14.7　实验 3-7：子函数跳转

1. 实验目的

熟练汇编中的子函数跳转。

2. 实验要求

请在 BenOS 里做如下练习。

（1）创建 bl_test()汇编函数，在该汇编函数里使用 CALL 指令跳转到实验 3-6 中实现的 sel_test()汇编函数。

（2）在 kernel.c 文件中，用 C 语言调用 bl_test()汇编函数。

3.14.8　实验 3-8：在汇编中实现串口输出功能

1. 实验目的

熟练使用 RISC-V 汇编指令。

2. 实验要求

在实际项目开发中，如果没有硬件仿真器（如 J-Link 仿真器），那么可以在汇编代码中利用下面几个常见的调试技巧。

- 利用 LED 实现一个跑马灯。
- 使用串口输出。

请在 BenOS 上用汇编代码实现串口的输出功能，并输出“Booting at asm”。

第 4 章　函数调用规范与栈

本章思考题

1. 请阐释 RISC-V 体系结构下的函数调用规范。
2. 在函数调用过程中，如果函数传递的参数大于 8，如何传递参数？
3. 假设函数调用关系为 main()→func1()→func2()，请画出 RSIC-V 体系结构下函数栈的布局。
4. 请画出 RISC-V 体系结构下函数调用过程中的入栈和出栈过程。
5. 请阐释在 RISC-V 体系结构下如何通过 FP 回溯整个栈。

本章主要介绍 RISC-V 函数调用规范和栈的基础知识，结合 LP64 数据模型介绍函数调用规范。

4.1 函数调用规范

函数调用规范（calling convention）用来描述父/子函数是如何编译和链接的，特别是父函数和子函数之间调用关系的约定，如栈的布局、参数的传递等。每个处理器体系结构都有不同的函数调用规范。本章重点介绍 RISC-V 的函数调用规范。

RISC-V 整型通用寄存器在函数调用规范中的使用情况如表 4.1 所示。

表 4.1　RISC-V 整型通用寄存器在函数调用规范中的使用情况

名称	ABI 别名	描述	调用过程中是否需要保存
x0	zero	内容一直为 0 的寄存器	否
x1	ra	保存返回地址	否
x2	sp	保存栈指针	是
x3	gp	保存全局指针	否
x4	tp	保存线程指针	否
x5～x7	t0～t2	临时寄存器	否
x8～x9	s0～s1	被调用者需要保存的寄存器	是
x10～x17	a0～a7	用于传递子程序的参数和结果	否
x18～x27	s2～s11	被调用者需要保存的寄存器	是
x28～x31	t3～t6	临时寄存器	否

总之，关于函数调用规范，有如下规则。

- 函数的前 8 个参数使用 a0～a7 寄存器传递。
- 如果函数参数多于 8 个，除前 8 个参数使用寄存器来传递之外，后面的参数使用栈传递。

- 如果传递的参数小于寄存器宽度（64 位），那么先按符号扩展到 32 位，再按符号扩展到 64 位。如果传递的参数为寄存器宽度的 2 倍（128 位），那么将使用一对寄存器来传递该参数。
- 函数的返回参数保存到 a0 和 a1 寄存器中。
- 函数的返回地址保存在 ra 寄存器中。
- 如果子函数里使用 s0～s11 寄存器，那么子函数在使用前需要把这些寄存器的内容保存到栈中，使用完之后再从栈中恢复内容到这些寄存器里。
- 栈向下增长（向较低的地址），sp 寄存器在进入函数时要对齐到 16 字节边界上。传递给栈的第一个参数位于 sp 寄存器的偏移量 0 处，后续的参数存储在相应的较高地址处。
- 如果 GCC 使用“-fno-omit-frame-pointer”编译选项，那么编译器使用 s0 作为栈帧指针（Frame Pointer，FP）。

【例 4-1】 请使用汇编语言实现下面的 C 语言程序。

```
#include <stdio.h>

int main(void)
{
    int a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, g = 7, h =8, i = 9, j = -1;

    printf("data: %d %d %d %d %d %d %d %d %d %d\n",
            a, b, c, d, e, f, g, h, i, j);

    return 0;
}
```

上面的 C 语言程序使用 printf()函数输出 10 个参数的值。根据函数调用规则，前 8 个参数使用 a0～a7 寄存器传递，后面的参数只能使用栈来传递。下面是使用汇编语言编写的代码。

```
<test.S>

1        .section  .rodata
2        .align  3
3    .string:
4        .string  "data: %d %d %d %d %d %d %d %d %d %d\n"
5
6    data:
7        .word 1, 2, 3, 4, 5, 6, 7, 8, 9, -1
8
9        .text
10       .align  2
11
12   .global  main
13   main:
14       /*栈往下扩展 48 字节*/
15       addi  sp,sp,-48
16
17       /*保存 main()函数的返回地址到栈里*/
18       sd  ra,40(sp)
19
20       /*a0 传递第一个参数——.string*/
21       la  a0, .string
22
23       /*a1 ~ a7 寄存器传递 printf()函数的前 7 个参数*/
24       li  a1,1
25       li  a2,2
26       li  a3,3
27       li  a4,4
28       li  a5,5
29       li  a6,6
30       li  a7,7
```

```
31
32        /*printf()函数的第 8~10 个参数通过栈来传递*/
33        li  t0,8
34        sd  t0,0(sp)
35        li  t0,9
36        sd  t0,8(sp)
37        li  t0,-1
38        sd  t0,16(sp)
39
40        /*调用 printf()函数*/
41        call printf
42
43        /*从栈中恢复返回地址*/
44        ld  ra,40(sp)
45        /*设置 main()函数的返回值为 0*/
46        li  a0,0
47        /*SP 回到原点*/
48        addi  sp,sp,48
49        ret
```

下面在 QEMU+RISC-V 平台上编译和运行。

```
# gcc test.S -o test
# ./test
data: 1 2 3 4 5 6 7 8 9 -1
```

下面是上述汇编代码的分析。

在第 1～7 行中，定义了一个只读数据段，其中 string 是 printf()函数输出的字符串，data 是一组数据。

在第 15 行中，把栈空间往下生长（扩展）48 字节，如图 4.1（a）所示。

在第 18 行中，把 main()函数的返回地址（ra 寄存器的值）保存到 SP+40 的位置上（s_ra），如图 4.1（b）所示。

在第 21 行中，加载字符串.string，并且把.string 的地址作为第 1 个参数传递给 printf()函数，这里使用了 a0 寄存器。

在第 24～30 行中，使用 a1～a7 寄存器传递 printf()函数的前 7 个参数。

在第 33～38 行中，剩余的参数需要通过栈传递，把 printf()函数的第 8 个参数存储到 SP+0 的位置上，第 9 个参数存储在 SP+8 的位置上，第 10 个参数存储到 SP+16 的位置上，如图 4.1（c）所示。

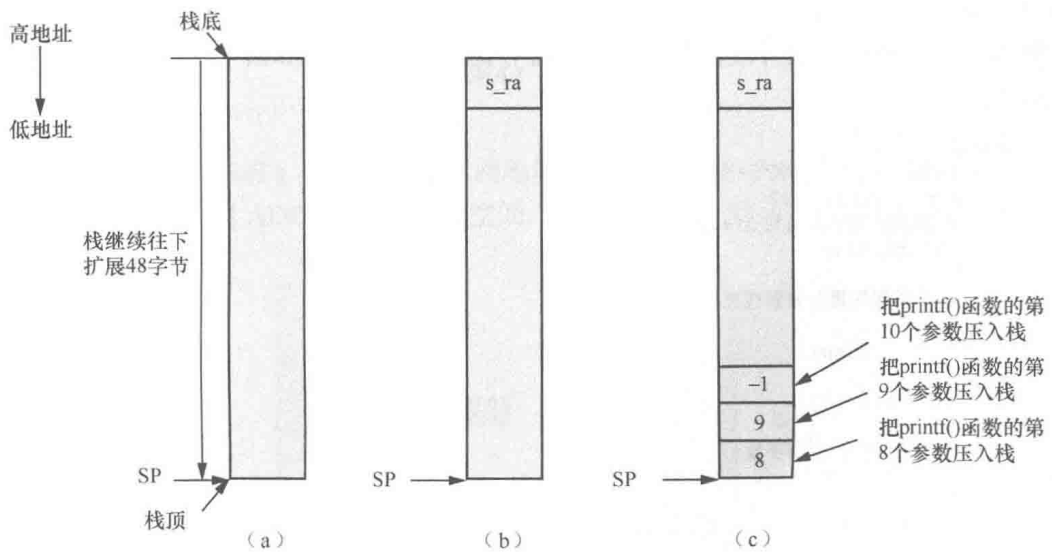


图 4.1 参数调用过程

在第 44 行中，从栈里恢复 main()函数的返回地址。

在第 46 行中，main()函数的返回值为 0。

在第 48 行中，释放栈空间。

在第 49 行中，通过 RET 指令返回。

4.2 入栈与出栈

栈（stack）是一种后进先出的数据存储结构。栈的作用如下。

- 保存临时存储的数据，如局部变量等。
- 在函数调用过程中，如果传递的参数少于或等于 8 个，那么使用 a0～a7 通用寄存器传递。当参数多于 8 个时，则需要使用栈传递参数。

通常栈是指一种从高地址往低地址扩展的数据存储结构。栈的起始地址称为栈底，栈从高地址往低地址延伸到的某个点称为栈顶。栈需要一个指针来指向栈最新分配的地址，即指向栈顶。这个指针是栈指针（SP）。把数据往栈里存储称为入栈或者压栈（push），从栈中移出数据称为出栈（pop）。当数据入栈时，SP 指向的地址减小，栈空间扩大；当数据出栈时，SP 指向的地址增大，栈空间缩小。另外，还有一个指针，叫作栈帧指针（FP），用来指向栈底。在 RISC-V 中使用 s0 寄存器作为 FP。

栈在函数调用过程中起到非常重要的作用，包括存储函数使用的局部变量、传递参数等。在函数调用过程中，栈是逐步生成的。为单个函数分配的栈空间，即从该函数栈底（高地址）到栈顶（低地址）的这段空间称为栈帧。

尽管有不少处理器的指令集提供了入栈和出栈的专用指令，例如，ARM 的 A32 指令集提供了 POP 和 PUSH 指令来实现入栈与出栈，但是在 RISC-V 指令集中并没有提供专门的入栈和出栈指令。

在 RISC-V 体系结构中，每个栈帧的大小至少为 16 字节。sp 寄存器指向栈顶，并且必须按 16 字节对齐。从栈底开始的 16 字节用来存储函数的返回地址和 FP 的值（假设 GCC 打开“-fno-omit-frame-pointer”编译选项），如图 4.2 所示。其中，s_ra 用来存储函数返回地址，s_fp 用来存储 FP 的值。s_fp 的用法在 4.3.2 节介绍。

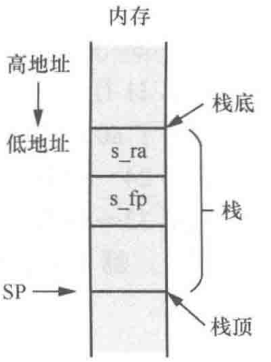


图 4.2　栈

【例 4-2】　下面的代码片段使用加法与加载/存储指令实现入栈和出栈操作。

```
1    func1:
2        /*栈往下扩展 32 字节*/
3        addi  sp,sp,-32
4        /*把返回地址存储到 SP+24*/
5        sd  ra,24(sp)
6
7        /*把局部变量存储到栈里*/
8        li  a5,1
9        sd  a5,8(sp)
10       li  a5,2
11       sd  a5,0(sp)
12
13       /*从栈里取出局部变量*/
14       ld  a1,0(sp)
15       ld  a0,8(sp)
16       /*调用子函数*/
17       call add_c
18
```

```
19        /*从栈里恢复返回地址*/
20        ld  ra,24(sp)
21        /*释放栈空间，SP 回到原点*/
22        addi  sp,sp,32
23        /*返回*/
24        ret
```

上述 func1()汇编函数演示了入栈和出栈的过程。

在第 1 行中，还没有为栈申请空间，如图 4.3（a）所示。

在第 3 行中，使用 ADDI 指令从 SP 指向的值减去 32 字节，相当于把栈空间往下扩展 32 字节，如图 4.3（b）所示。

在第 5 行中，把返回地址保存到 SP+24 位置上（s_ra），如图 4.3（c）所示。

在第 8～11 行中，把局部变量 *a* 和 *b* 分别存储到 SP 与 SP+8 位置上，如图 4.4（d）所示，其中 *a* 的值为 1，*b* 的值为 2。

在第 14 和 15 行中，从栈中重新把局部变量 *a* 和 *b* 取出。

在第 17 行中，通过 CALL 指令调用 add_c()函数。

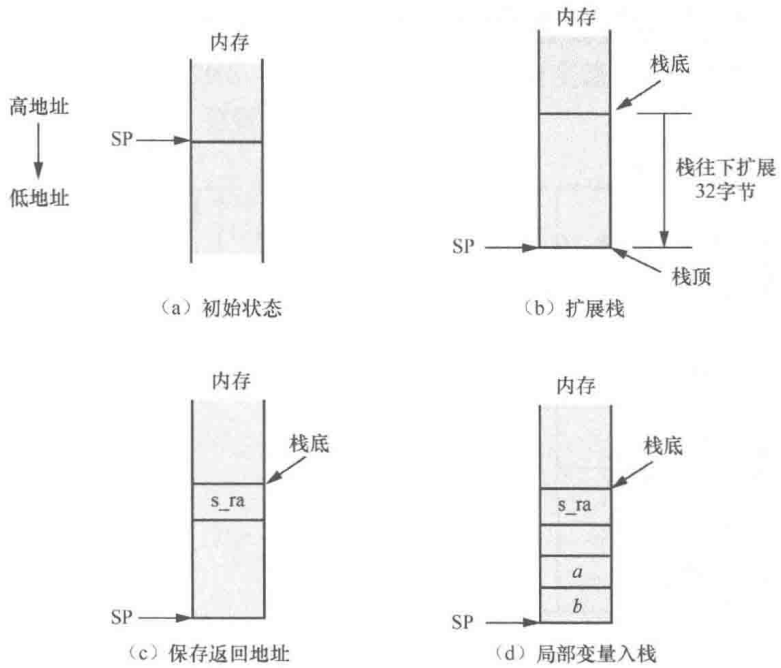


图 4.3　入栈

接下来是出栈操作。

在第 20 行中，从栈的 s_ra 位置恢复返回地址到 ra 寄存器中，如图 4.4（a）所示。

在第 22 行中，使用 ADDI 指令释放栈空间，如图 4.4（b）所示。

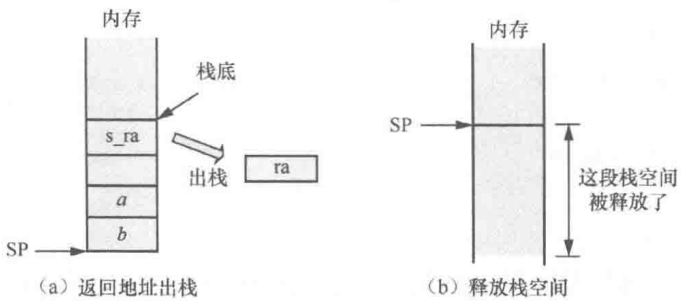


图 4.4　出栈

4.3 RISC-V 栈的布局

在函数调用过程中，栈是逐步生成的。对于栈的布局，我们需要分两种情况来考虑。

- 不使用 FP。若在 GCC 中使用“-fomit-frame-pointer”编译选项，可以不使用 FP，在函数入栈和出栈时减少访问内存的指令，从而提高程序性能。
- 使用 FP。在 GCC 中通过“-fno-omit-frame-pointer”编译选项使用 FP，目的是在调试过程中可以方便地计算每个栈的大小并方便回溯栈帧。

4.3.1 不使用 FP 的栈布局

下面以一个例子来说明。请确保 GCC 使用“-fomit-frame-pointer”编译选项。

【例 4-3】 在 BenOS 中，kernel_main()调用子函数 func1()，然后在 func1()函数中继续调用 add_c()函数。

首先，在 boot.S 汇编文件中分配栈空间，假设 SP 指向 0x8020 3000，然后，跳转到 C 语言的 kernel_main()函数中。

```
<boot.S>

1     .section ".text.boot"
2     .globl _start
3     _start:
4         /*分配栈空间，设置 SP */
5         la sp, stacks_start
6         li t0, 4096
7         add sp, sp, t0
8
9         tail  kernel_main
10
11    .section .data
12    .align  12
13    .global stacks_start
14    stacks_start:
15        .skip 4096
```

下面是 kernel.c 文件。

```
<kernel.c>

1     int add_c(int a, int b)
2     {
3         return a + b;
4     }
5
6     int func1(void)
7     {
8         int a = 1;
9         int b = 2;
10
11        return add_c(a, b);
12    }
13
14    void kernel_main(void)
15    {
16        func1();
17    }
```

从 boot.S 汇编文件通过 tail 指令跳转到 C 语言的 kernel_main()函数时，假设 SP 指向的初始值为 0x8020 3000，编译器会自动完成通过如下操作创建 kernel_main()函数的栈。

（1）使用 addi 指令扩展栈空间，SP 会向低地址延伸一段空间，为 kernel_main()函数创建一个栈帧，大小为 16 字节，如图 4.5（a）所示，此时 SP 指向的值为 0x8020 2FF0。

（2）把 kernel_main()函数的返回地址（ra 寄存器）存储到 SP+8 的位置上（s_ra）。

创建栈之后，使用 JAL 指令跳转到 func1()函数。

下面是 kernel_main()的反汇编代码（在 GDB 下使用 DISASSEMBLE 命令查看反汇编代码）。

```
<kernel_main()函数的反汇编代码>

(gdb) disassemble
Dump of assembler code for function kernel_main:
   0x0000000080200160 <+0>:     addi   sp,sp,-16
   0x0000000080200162 <+2>:     sd     ra,8(sp)
=> 0x0000000080200164 <+4>:     jal    ra,0x8020013e <func1>
   0x0000000080200168 <+8>:     nop
   0x000000008020016a <+10>:    ld     ra,8(sp)
   0x000000008020016c <+12>:    addi   sp,sp,16
   0x000000008020016e <+14>:    ret
```

当跳转到 func1()函数时，SP 会向低地址延伸 32 字节，为 func1()函数创建一个栈帧，如图 4.5（b）所示，此时 SP 指向的值为 0x8020 2FD0。这时候需要把 func1()函数的返回地址存储到 SP+24 位置（s_ra）上。另外，func1()函数有两个临时变量 *a* 和 *b*，它们也需要存储到栈里，分别存储到 SP+12 和 SP+8 位置上。

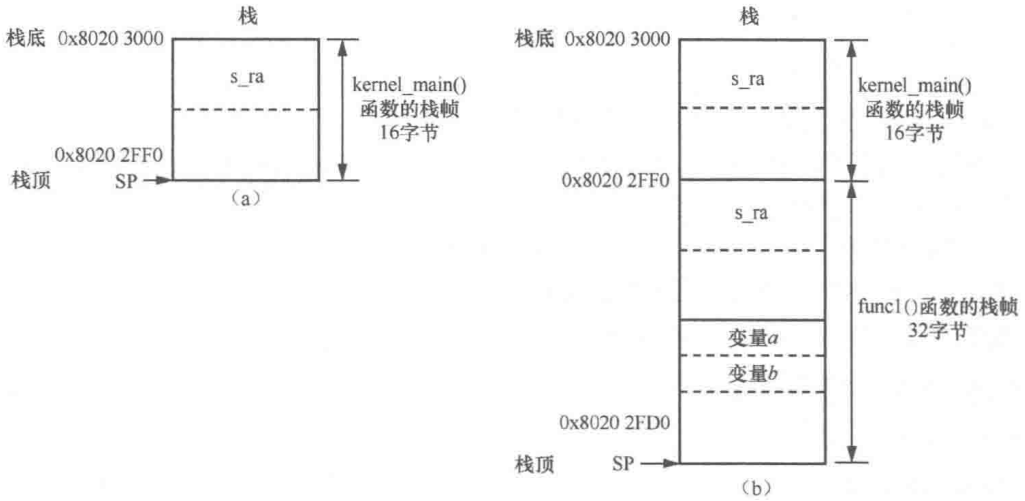


图 4.5 栈扩展过程 1

下面是 func1()函数的反汇编代码，使用 ADDI 指令扩展栈空间，然后把返回地址保存到栈中。把局部变量 *a* 和 *b* 先存储到栈里，然后从栈中取出变量的值，最后通过 JAL 指令跳转到 add_c()函数。

```
<func1()函数的反汇编代码>

(gdb) disassemble
Dump of assembler code for function func1:
   0x000000008020013e <+0>:     addi    sp,sp,-32
   0x0000000080200140 <+2>:     sd      ra,24(sp)
=> 0x0000000080200142 <+4>:     li      a5,1
   0x0000000080200144 <+6>:     sw      a5,12(sp)
   0x0000000080200146 <+8>:     li      a5,2
   0x0000000080200148 <+10>:    sw      a5,8(sp)
   0x000000008020014a <+12>:    lw      a4,8(sp)
   0x000000008020014c <+14>:    lw      a5,12(sp)
   0x000000008020014e <+16>:    mv      a1,a4
   0x0000000080200150 <+18>:    mv      a0,a5
```

```
   0x0000000080200152 <+20>:    jal     ra,0x80200124 <add_c>
   0x0000000080200156 <+24>:    mv      a5,a0
   0x0000000080200158 <+26>:    mv      a0,a5
   0x000000008020015a <+28>:    ld      ra,24(sp)
   0x000000008020015c <+30>:    addi    sp,sp,32
   0x000000008020015e <+32>:    ret
```

当跳转到 add_c()函数执行时，SP 会向低地址延伸 16 字节，为 add_c()函数创建一个栈帧，如图 4.6 所示，此时 SP 指向的值为 0x8020 2FC0。由于 add_c()函数是末端的函数，因此不用担心 ra（返回地址）被破坏。即使不存储 ra 和局部变量，我们也会为 add_c()函数分配 16 字节的栈空间。

add_c()函数的反汇编代码如下。

```
<add_c()函数的反汇编代码>

(gdb) disassemble
Dump of assembler code for function add_c:
   0x0000000080200124 <+0>:     addi    sp,sp,-16
   0x0000000080200126 <+2>:     mv      a5,a0
   0x0000000080200128 <+4>:     mv      a4,a1
   0x000000008020012a <+6>:     sw      a5,12(sp)
   0x000000008020012c <+8>:     mv      a5,a4
   0x000000008020012e <+10>:    sw      a5,8(sp)
=> 0x0000000080200130 <+12>:    lw      a4,12(sp)
   0x0000000080200132 <+14>:    lw      a5,8(sp)
   0x0000000080200134 <+16>:    addw    a5,a5,a4
   0x0000000080200136 <+18>:    sext.w  a5,a5
   0x0000000080200138 <+20>:    mv      a0,a5
   0x000000008020013a <+22>:    addi    sp,sp,16
   0x000000008020013c <+24>:    ret
```

当 add_c()函数执行完时，调用 RET 指令返回，它会做两件事情。

- 根据 ra 寄存器的返回地址，跳转到上一级函数（即 func1()函数通过 JAR 指令调用 add_c()函数的下一条指令）。
- 释放空间栈，即 SP 指向 func1()函数的栈顶。

下面是 func1()函数返回父函数的反汇编代码。首先把返回地址从栈（s_ra 的位置）中取出，然后释放栈空间，最后调用 ret 指令返回。

```
<func1()函数的反汇编代码>

000000008020015a <func1>:
    ...
    80200172:60e2              ld    ra,24(sp)
    80200174:6105              addi  sp,sp,32
    80200176:8082              ret
```

关于 RISC-V 中函数栈布局的关键点如下。

- 所有的函数调用栈从高地址向低地址扩展。
- SP 指向栈顶（栈的最低地址处）。
- 如果调用了子函数，函数的返回地址需要保存到栈里，即 s_ra 位置处。
- 栈的大小为 16 字节的倍数。
- 函数返回时需要先把返回地址从栈（s_ra 位置处）中恢复到 ra 寄存器，然后执行 RET 指令。

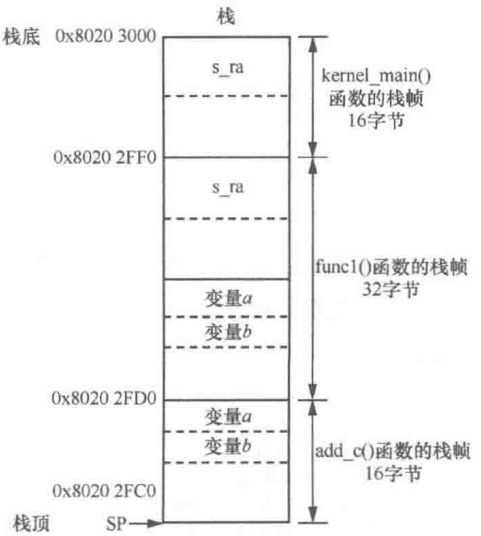


图 4.6　栈扩展过程 2

4.3.2　使用 FP 的栈布局

为了使用例 4-3 中的代码，首先需要修改 BenOS 的 Makefile 文件，把编译选项从“-fomit-

frame-pointer”改成“-fno-omit-frame-pointer”，然后重新编译。

当从 boot.S 汇编文件通过 tail 指令跳转到 C 语言的 kernel_main()函数时，假设 SP 和 FP 指向的初始值均为 0x8020 3000，编译器会自动完成如下操作来创建 kernel_main()函数的栈。

（1）使用 addi 指令扩展栈空间，SP 会向低地址延伸一段，为 kernel_main()函数创建一个栈帧，大小为 16 字节，如图 4.7（a）所示，此时 SP 指向的值为 0x8020 2FF0。

（2）把 kernel_main()函数的返回地址（ra 寄存器）存储到 SP+8 的位置（s_ra 位置）上。

（3）把 fp 寄存器的值存储到 SP 的位置（s_fp）上。

（4）更新 fp 寄存器的值，其值对应 kernel_main()函数的栈底，即 0x8020 3000。

创建栈之后，使用 JAL 指令跳转到 func1()函数。

下面是 kernel_main()的反汇编代码（在 GDB 下使用 disassemble 命令查看反汇编代码）。

```
(gdb) disassemble
Dump of assembler code for function kernel_main:
   0x00000000802001a4 <+0>:     addi    sp,sp,-16
   0x00000000802001a6 <+2>:     sd      ra,8(sp)
   0x00000000802001a8 <+4>:     sd      s0,0(sp)
   0x00000000802001aa <+6>:     addi    s0,sp,16
=> 0x00000000802001ac <+8>:     jal     ra,0x80200174 <func1>
   0x00000000802001b0 <+12>:    nop
   0x00000000802001b2 <+14>:    ld      ra,8(sp)
   0x00000000802001b4 <+16>:    ld      s0,0(sp)
   0x00000000802001b6 <+18>:    addi    sp,sp,16
   0x00000000802001b8 <+20>:    ret
```

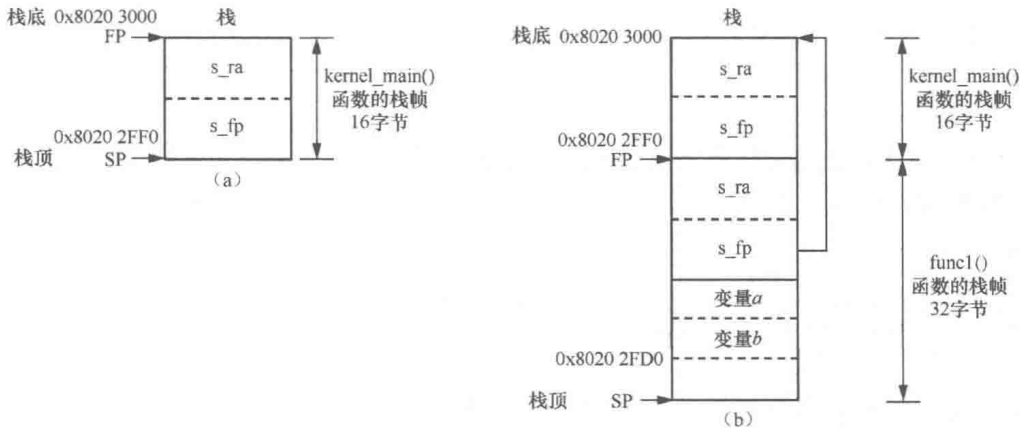


图 4.7 栈的布局（使用 FP）

当跳转到 func1()函数时，编译器自动按如下实现来创建栈帧。

（1）SP 会向低地址延伸 32 字节，为 func1()函数创建一个栈帧，此时 SP 指向的值为 0x8020 2FD0。

（2）把 func1()函数的返回地址存储到 SP+24 位置上。

（3）把 FP 指向的值存储到 SP+16 位置（s_fp）上，此时 s_fp 上存储的值为上一个栈帧的底部。

（4）更新 FP 指向的值为 func1()函数的栈底。

另外，func1()函数有两个临时变量 *a* 和 *b*，它们也需要存储到栈里，把它们分别存储到 SP+12 和 SP+8 位置上，最终结果如图 4.7（b）所示。

```
<func1()函数的反汇编代码>

Dump of assembler code for function func1:
   0x0000000080200174 <+0>:     addi    sp,sp,-32
```

```
   0x0000000080200176 <+2>:    sd      ra,24(sp)
   0x0000000080200178 <+4>:    sd      s0,16(sp)
   0x000000008020017a <+6>:    addi    s0,sp,32
=> 0x000000008020017c <+8>:    li      a5,1
   0x000000008020017e <+10>:   sw      a5,-20(s0)
   0x0000000080200182 <+14>:   li      a5,2
   0x0000000080200184 <+16>:   sw      a5,-24(s0)
   0x0000000080200188 <+20>:   lw      a4,-24(s0)
   0x000000008020018c <+24>:   lw      a5,-20(s0)
   0x0000000080200190 <+28>:   mv      a1,a4
   0x0000000080200192 <+30>:   mv      a0,a5
```

同理，当跳转到 add_c()函数时，处理器也会做类似的操作来创建栈帧。

综上所述，在本案例中，假设函数调用关系是 kernel_main()→func1()→add_c()，图 4.8 所示为栈的布局。

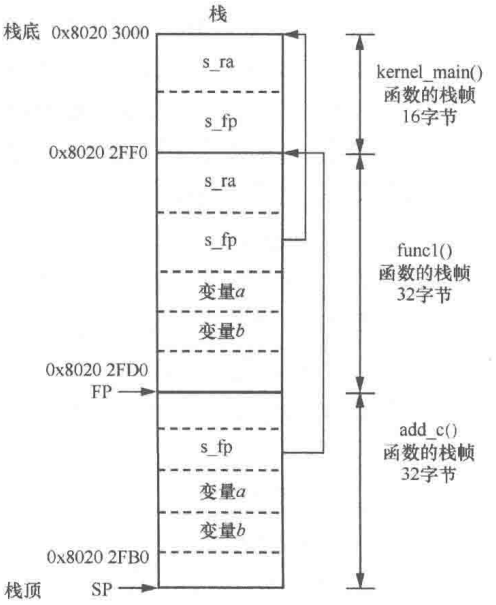


图 4.8　栈的布局（使用 FP）

关于 RISC-V 体系结构中函数栈布局的关键点如下。

- 所有的函数调用栈都会组成一个单链表。
- 每个栈使用两个地址来构成这个链表，这两个地址都是 64 位宽的，并且它们都位于栈底。
  - s_fp 的值指向上一个栈帧（父函数的栈帧）的栈底。
  - s_ra 保存当前函数的返回地址，也就是父函数调用该函数时的地址。
- 当函数返回时，RISC-V 处理器先把返回地址从栈的 s_ra 位置处载入当前 ra 寄存器，然后执行 RET 指令。

4.3.3　栈回溯

操作系统常用的输出栈信息等技术手段是通过 FP 完成的。例如，下面的错误日志输出了发生异常时的函数栈信息，如函数名称等，并且通过栈的回溯技术输出函数调用关系。

```
Oops - Store/AMO page fault
Call Trace:
[<0x0000000080202edc>] test_access_unmap_address+0x1c/0x42
[<0x0000000080202f12>] test_mmu+0x10/0x1a
[<0x000000008020329a>] kernel_main+0xb4/0xb6
```

下面通过一个示例分析如何通过 FP 回溯整个栈。

【例 4-4】　在例 4-3 的基础上，输出每个栈的范围，以及调用该函数时的 PC 值，如下面的日志信息所示。

```
Call Frame:
[0x0000000080202fa0 - 0x0000000080202fb0]  pc 0x0000000080200f32
[0x0000000080202fb0 - 0x0000000080202fd0]  pc 0x000000008020114a
[0x0000000080202fd0 - 0x0000000080202ff0]  pc 0x0000000080201184
[0x0000000080202ff0 - 0x0000000080203000]  pc 0x00000000802011a4
```

如果想把 PC 值对应的符号名称（函数名称）显示出来，需要建立一个符号名称与地址的对应表，然后查表，本示例中，我们没有满足这个需求。不过，读者可以通过查看 benos.map 文件的符号表信息确定 PC 值对应的函数名称。

下面是实现栈回溯的示例代码。

```
<stacktrace.c>

1    struct stackframe {
2        unsigned long s_fp;
```

```
3         unsigned long s_ra;
4     };
5
6     extern char _text[], _etext[];
7     static int kernel_text(unsigned long addr)
8     {
9         if (addr >= (unsigned long)_text &&
10            addr < (unsigned long)_etext)
11            return 1;
12
13        return 0;
14    }
15
16    static void walk_stackframe(void )
17    {
18        unsigned long sp, fp, pc;
19        struct stackframe *frame;
20        unsigned long low;
21
22        const register unsigned long current_sp __asm__ ("sp");
23        sp = current_sp;
24        pc = (unsigned long)walk_stackframe;
25        fp = (unsigned long)__builtin_frame_address(0);
26
27        while (1) {
28            if (!kernel_text(pc))
29                break;
30
31            /*检查 FP 是否有效*/
32            low = sp + sizeof(struct stackframe);
33            if ((fp < low || fp & 0xf))
34                break;
35
36            /*
37             *FP 指向上一级函数的栈底
38             *减去 16 字节，正好指向结构体 stackframe
39             */
40            frame = (struct stackframe *)(fp - 16);
41            sp = fp;
42            fp = frame->s_fp;
43
44            pc = frame->s_ra - 4;
45
46            if (kernel_text(pc))
47                printk("[0x%016lx - 0x%016lx]  pc 0x%016lx\n", sp, fp, pc);
48        }
49    }
50
51    void dump_stack(void)
52    {
53        printk("Call Frame:\n");
54        walk_stackframe();
55    }
```

在第 1～4 行中，定义一个 stackframe 数据结构来描述栈结构中的 s_fp 和 s_ra。

在第 6～14 行中，检查地址是否在代码段中。

在第 22 行中，通过内嵌汇编方式直接获取 SP 的值。

在第 24 行中，PC 值为当前函数的地址。

在第 25 行中，通过__builtin_frame_address()来获取 FP 的值。

在第 32～34 行，对 FP 做检查。首先，FP 和 SP 的差值要大于 16 字节。其次，FP 需要按 16 字节对齐。

在第 40 行中，由于 FP 指向上一级函数的栈底，而栈底正好存放了 stackframe 数据结构。根据 4.3.2 节总结的规律，我们可以实现栈的回溯。

最后，我们需要在 add_c()函数里调用 dump_stack()。

```
int add_c(int a, int b)
{
    dump_stack();
    return a + b;
}
```

4.4 实验

4.4.1 实验 4-1：观察栈布局

1. 实验目的

熟悉 RISC-V 的栈布局。

2. 实验要求

请在 BenOS 里做如下练习。

在 BenOS 里实现函数调用 kernel_main()→func1()→func2()，然后使用 GDB 观察栈的变化情况，并画出栈布局。

4.4.2 实验 4-2：观察栈回溯

1. 实验目的

熟悉 RISC-V 的栈回溯。

2. 实验要求

请在 BenOS 里做如下练习。

在 BenOS 里实现函数调用 kernel_main()→func1()→func2()，并实现一个栈回溯功能，输出栈的地址范围和大小，并通过 GDB 观察栈是如何回溯的。

第5章 GNU汇编器

本章思考题

1. 什么是汇编器？
2. 如何给汇编代码添加注释？
3. 什么是符号？
4. 什么是伪指令？
5. 在RISC-V汇编中，“.align 3”表示什么意思？
6. 下面这条伪指令表示什么意思？

```
.section ".my.text","awx"
```

7. 在汇编宏里，如何使用参数？
8. 下面是my_entry宏的定义。

```
.macro my_entry, rv, label
      j    rv\()\rv\()_\label
 .endm
```

下面的语句调用my_entry宏，请解释该宏是如何展开的。

```
my_entry   1, irq
```

9. 请阐释.section和.previous伪指令的作用。

本章主要介绍与GNU汇编器相关的内容。汇编器是将汇编代码翻译为机器目标代码的程序。通常，汇编代码通过汇编器生成目标代码，然后由链接器链接成最终的可执行二进制程序。对于RISC-V的汇编语言来说，常用的汇编器是GCC提供的AS。AS采用AT&T格式。AT&T格式源自贝尔实验室，是为开发UNIX系统而产生的汇编语法。

GNU工具链提供了一个名为as的命令。如图5.1所示，as命令的版本为2.37，汇编目标文件配置成“riscv64-linux-gnu”，即汇编后的文件为RV64体系结构的。

```
root@benshushu:/mnt/riscv# as --version
GNU assembler (GNU Binutils for Debian) 2.37
Copyright (C) 2021 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or later.
This program has absolutely no warranty.
This assembler was configured for a target of `riscv64-linux-gnu'.
root@benshushu:/mnt/riscv#
```

图5.1 as命令

5.1 编译流程与ELF文件

本节以一个简单的C语言程序为例。

```
<test.c>

#include <stdio.h>
```

```
int data = 10;

int main(void)
{
    printf("%d\n", data);

    return 0;
}
```

GCC 的编译流程主要分成如下 4 个步骤。

（1）预处理（pre-process）。GCC 的预处理器（CPP）对各种预处理命令进行处理，例如，对头文件的处理、宏定义的展开、条件编译的选择等。预处理完成之后，会生成 test.i 文件。另外，我们也可以通过如下命令生成 test.i 文件。

```
gcc -E test.c -o test.i
```

（2）编译（compile）。C 语言的编译器（CC）首先对预处理之后的源文件进行词法、语法以及语义分析，然后进行代码优化，最后把 C 语言代码翻译成汇编代码。编译完成之后，生成 test.s 文件。另外，我们也可以通过如下命令生成汇编文件。

```
gcc -S test.i -o test.s
```

（3）汇编（assemble）。汇编器（AS）把汇编代码翻译成机器语言，并生成可重定位目标文件。汇编完成之后，生成 test.o 文件。另外，我们可以通过如下命令生成 test.o 文件。

```
as test.s -o test.o
```

（4）链接（link）。链接器（LD）会把所有生成的可重定位目标文件以及用到的库文件组合成一个可执行二进制文件。另外，我们可以通过如下命令手动生成可执行二进制文件。

```
ld -o test  test.o -lc
```

图 5.2 所示的是编译 test.c 源代码的过程。

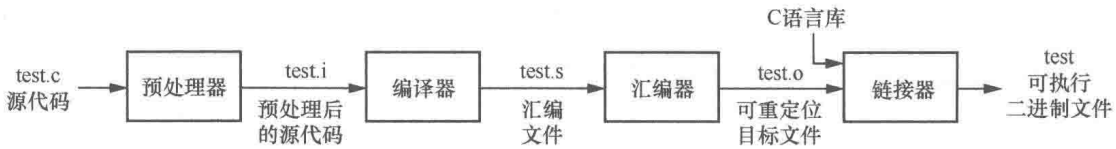


图 5.2　编译 test.c 源代码的过程

汇编阶段生成的可重定位目标文件以及链接阶段生成的可执行二进制文件都是按照一定文件格式（如 ELF 格式）组成的二进制文件。在 Linux 系统中，应用程序常用的可执行文件格式是 ELF，它是对象文件的一种格式，用于定义不同类型的对象文件中都放了什么内容，以及以什么格式存放这些内容。ELF 文件的结构如图 5.3 所示。

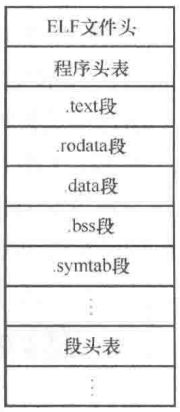


图 5.3　ELF 文件的结构

ELF 最开始的部分是 ELF 文件头（ELF header），它包含描述整个文件的基本属性，如 ELF 文件版本、目标计算机型号、程序入口地址等信息。程序头表（program header table）描述如何创建一个进程的内存镜像。程序头表后面是各个段[①]（section），包括代码（.text）段、只读数据（.rodata）段、数据（.data）段、未初始化的数据（.bss）段等。段头表（section header table）用于描述 ELF 文件中包含的所有段信息，如每个段的名字、段的长

① 有的中文教材使用“节”表示。

度、在文件中的偏移量、读写权限以及段的其他属性等。

下面介绍常见的几个段。

- 代码段：存放程序源代码编译后生成的机器指令。
- 只读数据段：存储只能读取不能写入的数据。
- 数据段：存放已初始化的全局变量和已初始化的局部静态变量。
- 未初始化的数据段：存放未初始化的全局变量以及未初始化的局部静态变量。
- 符号表（.symtab）段：存放函数和全局变量的符号表信息。
- 可重定位代码（.rel.text）段：存储代码段的重定位信息。
- 可重定位数据（.rel.data）段：存储数据段的重定位信息。
- 调试符号表（.debug）段：存储调试使用的符号表信息。

我们可以通过 READELF 命令（例如，读取 test 文件的 ELF 文件头信息）了解一个目标二进制文件的组成。

```
root:riscv# readelf -h test
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x560
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6688 (bytes into file)
  Flags:                             0x5, RVC, double-float ABI
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

从上面的信息可知，test 文件是一个 ELF64 类型的可执行文件（executable file）。test 程序的入口地址为 0x560。段头（section header）的数量是 27，程序头（program header）的数量是 9。

下面通过 READELF 命令读取段头表信息。下面是 test 可执行二进制文件的段头表的信息片段。

```
root@benshushu:/mnt/riscv# readelf -S test
There are 27 section headers, starting at offset 0x1a20:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align

  [12] .text             PROGBITS         0000000000000560  00000560
       000000000000014a  0000000000000000  AX       0     0     4
  [13] .rodata           PROGBITS         00000000000006b0  000006b0
       000000000000000c  0000000000000000   A       0     0     8

  [21] .got              PROGBITS         0000000000002010  00001010
       0000000000000048  0000000000000008  WA       0     0     8
  [22] .bss              NOBITS           0000000000002058  00001058
       0000000000000008  0000000000000000  WA       0     0     1
  [24] .symtab           SYMTAB           0000000000000000  00001078
       0000000000000648  0000000000000018          25    45     8
```

```
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)
```

从上面的信息可知，test 文件一共有 27 个段，段头表从 0x1a20 地址开始。这里除我们常见的代码段、数据段以及只读数据段之外，还包括其他的一些段。以代码段为例，它的起始地址为 0x560，偏移量为 0x560，大小为 0x14a，属性为可分配（A）和可执行（X）属性。

汇编阶段生成的可重定位目标文件和链接阶段生成的可执行二进制文件的主要区别在于，可重定位目标文件的所有段的起始地址都是 0，读者可以通过“readelf -S test.o”命令查看 test.o 文件的段头表信息；而链接器在链接过程中根据链接脚本的要求会把所有可重定位目标文件中相同的段（在链接脚本中称为输入段）合并生成一个新的段（在链接脚本中称为输出段）。合并的输出段会根据链接脚本的要求重新确定每个段的虚拟地址和加载地址。

在默认情况下，链接器使用自带的链接脚本，读者可以通过如下命令查看自带的链接脚本。

```
$ ld --verbose
```

符号表（symbol table）最初在生成可重定位目标文件时创建，存储在符号表段中。不过，此时的符号还没有一个确定的地址，所有符号的地址都是 0。符号表包括全局符号、本地符号以及外部符号。链接器在链接过程中对所有输入可重定位目标文件的符号表进行符号解析和重定位，每个符号在输出文件的相应段中得到一个确定的地址，最终生成一个符号表。

5.2 一个简单的汇编程序

编译和运行一个简单的汇编程序有两种方式：一是在 RISC-V 处理器的 Linux 系统中编译和运行汇编程序，如运行 RISC-V Linux 的 QEMU 系统；二是编写一个裸机的汇编程序，如本书的实验平台 BenOS。本节的例子采用第一种方式。

【例 5-1】　下面是一段用汇编指令写的程序，文件名为 test.S。

```
1    # 测试程序：往终端中输出 my_data1 数据与 my_data2 数据之和
2     .section .data
3     .align  3
4
5     my_data1:
6         .word  100
7
8    my_data2:
9         .word  50
10
11   print_data:
12        .string "data: %d\n"
13
14   .align  3
15   .section .text
16
17   .global main
18   main:
19        addi sp, sp, -16
20        sd ra, 8(sp)
21
22        lw t0, my_data1
23        lw t1, my_data2
24        add a1, t0, t1
25
```

```
26        la a0, print_data
27        call printf
28
29        li a0, 0
30
31        ld ra, 8(sp)
32        addi sp, sp, 16
33        ret
```

首先，把上述代码文件复制到 QEMU+RISC-V+Linux 实验平台中。使用 as 命令编译 test.S 文件。

```
# as test.S -o test.o
```

其中 as 为 GNU 汇编器命令，test.S 为汇编源文件，-o 选项告诉汇编器编译后输出的目标文件为 test.o。目标文件 test.o 是基于机器语言的文件，还不是可执行二进制文件，我们需要使用链接器把目标文件合并与链接成可执行文件。关于链接器的更多内容，请参考第 6 章。

```
# ld test.o -o test -Map test.map -lc --dynamic-linker
/lib/ld-linux-riscv64-lp64d.so.1
```

ld 为 GNU 链接器命令。其中，test.o 是输入文件，-o 选项告诉链接器最终链接后输出的二进制文件为 test，-Map 输出的符号表可用于调试，-lc 表示链接 libc 库。

运行 test 程序。

```
# ./test
data: 150
```

可执行二进制文件由代码段、数据段以及未初始化的数据段等组成。代码段存放程序执行代码，数据段存放程序中已初始化的全局变量等，未初始化的数据段包含未初始化的全局变量和未初始化的局部静态变量。此外，可执行二进制文件还包含符号表，这个表里包含程序中定义的所有符号的相关信息。

下面分析这个 test.S 汇编文件。

第 1 行以“#”字符开始，是注释。

在第 2 行中，以“.”字符开始的指令是汇编器能识别的伪操作，它不会直接被翻译成机器指令，而由汇编器来预处理。“.section .data”用来表明数据段的开始。程序中需要用的数据可以存储在数据段中。在第 15 行中，“.section .text”表示接下来的代码为代码段。

在第 3 行中，.align 是对齐伪操作，参数为 3，因此对齐的字节大小为 2^3，即接下来的数据所在的起始地址能被 8 整除。

在第 5～9 行中，“.word”是数据定义的伪指令，用来定义数据元素，数据元素的标签为 my_data1/ my_data2，它存储了一个 32 位的数据。在汇编代码中，任何以“:”符号结束的字符串都被视为标签（label）或者符号（symbol）。

在第 11～12 行中，“.string”是数据定义伪指令，用来定义字符串。

在第 17 行中，“.global main”表示把 main 设置为全局可以访问的符号。main 是一个特殊符号，用来标记该程序的入口地址。“.global”是用来定义全局符号的伪指令，该符号可以是函数的符号，也可以是全局变量的符号。

在第 18 行中，定义 main 标签。标签是一个符号，后面跟着一个冒号。标签定义符号的值，当汇编器对程序进行编译时会为每个符号分配地址。标签的作用是告诉汇编器以该符号的地址作为下一条指令或者数据的起始地址。

第 19～33 行是这个程序代码段的主体。

在第 19 行中，申请 16 字节大小的栈空间。

在第 20 行中，把返回地址存储到栈中 SP+8 的位置上。

在第 22 和 23 行中，读取 my_data1 以及 my_data2 标签存储的数据。

在第 24 行中，使用 add 指令相加。

在第 26 行中，加载 print_data 标签的地址到 a0 寄存器。

在第 27 行中，通过 call 指令来调用 C 库的 printf()函数。其中，a0 是第一个参数，a1 是第二个参数。

在第 29 行中，设置 main()函数的返回值。

在第 31 行中，从栈中恢复返回地址到 ra 寄存器中。

在第 32 行中，释放栈空间。

在第 33 行中，通过 ret 指令返回。

我们可以通过 readelf 命令获取 test 程序的符号表。readelf 命令通常用于查看 ELF 格式的文件信息。其中，-s 选项用来显示符号表的内容。

```
root:riscv# readelf -s test

Symbol table '.symtab' contains 37 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    26: 0000000000002040     0 NOTYPE  GLOBAL DEFAULT   14 __BSS_END__
    27: 0000000000002040     0 NOTYPE  GLOBAL DEFAULT   14 _edata
    28: 0000000000002040     0 NOTYPE  GLOBAL DEFAULT   14 __SDATA_BEGIN__
    29: 0000000000002000     0 NOTYPE  GLOBAL DEFAULT   13 __DATA_BEGIN__
    30: 0000000000002000     0 NOTYPE  GLOBAL DEFAULT   13 my_data1
    31: 0000000000002040     0 NOTYPE  GLOBAL DEFAULT   14 _end
    32: 0000000000000320     0 NOTYPE  GLOBAL DEFAULT   11 main
    33: 0000000000002800     0 NOTYPE  GLOBAL DEFAULT   ABS __global_pointer$
    34: 0000000000002040     0 NOTYPE  GLOBAL DEFAULT   14 __bss_start
    35: 0000000000002004     0 NOTYPE  GLOBAL DEFAULT   13 my_data2
```

从上面的日志可知，test 程序的符号表包含 37 项，其中 my_data1 标签的地址为 0x2000，my_data2 标签的地址为 0x2004，而 main 符号的地址为 0x320。

5.3 汇编语法

下面介绍 AS 汇编器中常见的语法。

5.3.1 注释

汇编代码可以通过如下方式来注释。

- “//”或者“#”：如果出现在一行的开始，表示注释整行；如果出现在一行中间，表示注释后面的内容。
- “/* */”：可以跨行注释。

5.3.2 符号

符号是一个核心概念。程序员使用符号命名事物，链接器使用符号链接，调试器使用符号调试。符号一般用来标记程序或数据的位置，而不用内存地址来标记它们。如果使用内存地址来标记，那么程序员必须记住每行代码或者数据的内存地址，这将是一件很麻烦的事情。

符号可以由下面几种字符组合而成：

- 所有字母（包括大写和小写）；

- 数字；
- “_”“.”以及“$”这3个字符。

符号可以代表它所在的地址，也可以当作变量或者函数使用。

全局符号（global symbol）可以使用.global声明。全局符号可以被其他模块引用，例如，C语言可以引用全局符号。

本地符号（local symbol）主要在本地汇编代码中引用。在ELF文件中，通常使用“.L”前缀定义本地符号。本地符号不会出现在符号表中。

本地标签（local label）可供汇编器和程序员临时使用。标签通常使用0～99的整数作为编号，和f指令与b指令一起使用。其中，f表示汇编器向前搜索，b表示汇编器向后搜索。

我们可以重复定义相同的本地标签，不过在汇编器内部本地标签会编译成更有意义的本地符号，例如，第一个“1:”本地标签可能编译成“.L1C-B1”，第二个“1:”本地标签可能编译成“.L1C-B2”。例如，使用相同的数字 *N*。跳转指令只能引用（向后引用或者向前引用）最近定义的本地标签。

【例5-2】 下面的汇编代码使用数字定义标签的编号。

```
1:
    j 1f
2:
    j 1b
1:
    j 2f
2:
    j 1b
```

上述汇编代码等同于如下汇编代码。

```
label_1:
        j label_3
label_2:
        j label_1
label_3:
        j label_4
label_4:
        j label_3
```

5.4 常用的伪指令

伪指令是对汇编器发出的命令，它在源程序汇编期间由汇编器处理。伪指令是由汇编器预处理的指令，它可以分解为几条指令的集合。另外，伪指令仅仅在汇编器编译期间起作用。当汇编结束时，伪指令的使命也就结束了。伪操作可以实现如下功能：

- 符号定义；
- 数据定义和对齐；
- 汇编控制；
- 汇编宏；
- 段描述。

5.4.1 对齐伪指令

.ALIGN伪指令用来对齐或者填充数据等。.ALIGN伪指令通常有3个参数。第一个参数表示对齐的要求。第二个参数表示要填充的值，它可以省略。如果省略，填充的字节通常为零。

在大多数系统上，如果需要在代码段中填充第二个参数，则用 nop 指令来填充。第三个参数表示对齐指令应该跳过的最大字节数。如果为了对齐需要跳过比指定的最大字节数更多的字节，则不会执行对齐操作。通常我们只使用第一个参数。在 RISC-V 中，第一个参数表示 2^n 字节。

【例 5-3】 下面的.ALIGN 伪指令表示按照 4 字节对齐。

```
.align 2
```

【例 5-4】 下面是使用 3 个参数的.ALIGN 伪指令。

```
.align 5,0,100

.align 5,0,8
```

上述两条伪指令都要求按照32字节对齐。其中，第一条伪指令设置最多跳过的字节数为100，填充的值为 0，而第二条伪指令设置最多跳过的字节数小于对齐的字节数，因此该伪指令有可能不会执行，不推荐这么使用。

5.4.2 数据定义伪指令

下面是汇编代码中常用的数据定义伪指令。

- .byte：把 8 位数当成数据插入汇编代码中。
- .hword 和.short：把 16 位数当成数据插入汇编代码中。
- .long 和.int：这两条伪指令的作用相同，都把 32 位数当成数据插入汇编代码中。
- .word：把 32 位数当成数据插入汇编代码中。
- .quad：把 64 位数当成数据插入汇编代码中。
- .float：把浮点数当成数据插入汇编代码中。
- .ascii 和.string：把 string 当作数据插入汇编代码中，对于 ascii 伪操作定义的字符串，需要自行添加结尾字符'\0'。
- .asciz：类似于 ascii，在 string 后面自动插入一个结尾字符'\0'。
- .rept 和.endr：重复执行伪操作。
- .equ：给符号赋值。

【例 5-5】 下面的代码片段使用数据定义伪指令。

```
.rept 3
.long 0
.endr
```

上述的.REPT 伪操作会重复“.long 0”指令 3 次，等同于下面的代码片段。

```
.long 0
.long 0
.long 0
```

.EQU 伪指令给符号赋值，其指令格式如下。

```
.equ symbol, expression
```

【例 5-6】 使用.EQU 伪指令来改写例 5-1。

```
.equ my_data1, 100   #为 my_data1 符号赋值 100
.equ my_data2, 50    #为 my_data2 符号赋值 50
.global main
main:
     ...
     li x2, =my_data1
     li x3, =my_data2
```

```
add x1, x2, x3
...
```

5.4.3 与函数相关的伪指令

下面是汇编代码中与函数相关的伪指令。

- .global：定义一个全局的符号，可以是函数的符号，也可以是全局变量的符号。
- .include：引用头文件。
- .if, .else, .endif：控制语句。
- .ifdef symbol：判断 symbol 是否定义。
- .ifndef symbol：判断 symbol 是否没有定义。
- .ifc string1,string2：判断字符串 string1 和 string2 是否相同。
- .ifeq expression：判断 expression 的值是否为 0。
- .ifeqs string1,string2：等同于.ifc。
- .ifge expression：判断 expression 的值是否大于或等于 0。
- .ifle expression：判断 expression 的值是否小于或等于 0。
- .ifne expression：判断 expression 的值是否不为 0。

5.4.4 与段相关的伪指令

1. .SECTION 伪指令

.SECTION 伪指令表示接下来的汇编会链接到某个段，如代码段、数据段等。.SECTION 伪指令的格式如下。

```
.section name, "flags"
```

其中，name 表示段的名称；flags 表示段的属性，如表 5.1 所示。

表 5.1　段的属性

属性	说明
a	段具有可分配属性
d	具有 GNU_MBIND 属性的段
e	段被排除在可执行和共享库之外
w	段具有可写属性
x	段具有可执行属性
M	段具有可合并属性
S	段包含零终止字符串
G	段是段组（section group）的成员
T	段用于线程本地存储（thread-local-storage）

.SECTION 伪指令定义的段会以段的名称开始。该段的结束有两个位置，一是下一个段的开始处，二是文件结尾处。

【例 5-7】 下面的.SECTION 伪指令用来声明一个新的段。

```
.section .fixup,"ax"
```

这表示接下来的代码在. fixup 段里，具有可分配以及可执行的属性。

.SECTION 伪指令可以用于一个汇编文件中定义多个不同的段。图 5.4 中的代码定义了两个不同的段。

在第 1 行中，使用.SECTION 伪指令来定义一个数据段，该段从第 1 行开始到第 8 行结束。

在第 10 行中，使用.SECTION 伪指令来定义一个代码段，该段从 10 行开始到文件结束。

2. .PUSHSECTION 和.POPSECTION 伪指令

.PUSHSECTION 和.POPSECTION 伪指令通常需要配对使用，用于把代码链接到指定的段，而其他代码还保留在原来的段中。

【例 5-8】 下面的代码片段使用了.PUSHSECTION 和.POPSECTION 伪指令。

```
1     .section .text
2     .global my_memcpy_test
3     my_memcpy_test:
4         ...
5         ret
6
7     .pushsection ".my.text", "awx"
8
9     .global compare_and_return
10    compare_and_return:
11        bltu  a0,a1,.L2
12        li  a5,0
13        j  .L3
14    .L2:
15        li  a5,-1
16    .L3:
17        mv  a0, a5
18        ret
19
20    .popsection
21
22    ...
```

```
1  .section .data
2  .align  3
3  my_data1:
4          .word  100
5  my_data2:
6          .word  50
7  print_data:
8          .string "data: %d\n"
9
10 .section .text
11 .align  2
12 .global main
13 main:
14         addi sp, sp, -16
15         sd ra, 8(sp)
16
17         lw t0, my_data1
18         lw t1, my_data2
19         add a1, t0, t1
20
21         la a0, print_data
22         call printf
23
24         li a0, 0
25
26         ld ra, 8(sp)
27         addi sp, sp, 16
28         ret
```

数据段

代码段

图 5.4　定义两个段

在第 1 行中，使用.SECTION 伪指令来定义一个代码段。

在第 2～5 行中，my_memcpy_test 函数会链接到代码段中。

在第 7～20 行中，使用.PUSHSECTION 和.POPSECTION 伪指令把 compare_and_return 函数链接到.my.text 段。

3. .SECTION 和.PREVIOUS 伪指令

通常.SECTION 和.PREVIOUS 两条伪指令是配对使用的，用来把一段汇编代码链接到特定的段。这里.SECTION 伪指令表示开始一个新的段，.PREVIOUS 伪指令表示恢复到.SECTION 定义之前的那个段，以那个段作为当前段。

【例 5-9】 下面的代码片段使用了.SECTION 和.PREVIOUS 伪指令。

```
1     .section ".text.boot"
2
3     .globl _start
4     _start:
5         /*关闭中断*/
6         csrw sie, zero
7
8         /*设置栈，栈的大小为 4KB*/
9         la sp, stacks_start
10        li t0, 4096
11        add sp, sp, t0
12
13        .section .fixup, "ax"
14        .balign 4
15        li a2, -1
16        li a1, 0
17        .previous
18
19        .section .init_boot,"ax"
20        li a0, -1
21        mv a1, a2
```

```
22        .previous
23
24        call kernel_main
```

在第 1 行中，声明一个.text.boot 段。

在第 13～17 行中，通过.SECTION 和.PREVIOUS 两条伪指令把这里面的汇编代码链接到.fixup 段。

在第 19～22 行中，通过.SECTION 和.PREVIOUS 两条伪指令把这里面的汇编代码链接到.init_boot 段。

在第 24 行中，CALL 指令恢复到之前的.SECTION 定义之前的那个段作为当前段，即.text.boot 段。

所以，第 6～11 行以及第 24 行汇编代码会链接到.text.boot 段，而第 14～16 行汇编代码链接到.fixup 段，第 20 和 21 行汇编代码链接到.init_boot 段中。

我们也可以通过 riscv64-linux-gnu-objdump 命令查看段的信息。

```
$ riscv64-linux-gnu-objdump -d benos.elf

benos.elf:     file format elf64-littleriscv


Disassembly of section .text.boot:

0000000080200000 <_start>:
    80200000:10401073            csrw  sie,zero
    80200004:00001117            auipc  sp,0x1
    80200008:ffc10113            addi  sp,sp,-4 # 80201000 <stacks_start>
    8020000c:000012b7            lui  t0,0x1
    80200010:00510133            add  sp,sp,t0
    80200014:174000ef            jal  ra,80200188 <kernel_main>
Disassembly of section .fixup:

00000000802001a4 <.fixup>:
    802001a4:fff00613            li  a2,-1
    802001a8:00000593            li  a1,0

Disassembly of section .init_boot:

00000000802001ac <.init_boot>:
    802001ac:fff00513            li  a0,-1
    802001b0:00060593            mv  a1,a2
```

5.4.5 与宏相关的伪指令

.MACRO 和.ENDM 伪指令可以用来组成一个宏。.MACRO 伪指令的格式如下。

```
.macro macname macargs ...
```

.MACRO 伪指令后面依次是宏名称与宏的参数。

1. 宏的参数使用

在宏里使用参数，需要添加前缀“\”。

【例 5-10】 下面的代码片段在宏里使用参数。

```
.macro add_1 p1 p2
add x0,   \p1, \p2
.endm
```

另外，在定义宏参数时还可以设置一个初始化值，如下面的代码片段所示。

```
.macro reserve_str p1=0 p2
```

在上述的 reserve_str 宏中，参数 p1 有一个默认值 0。当使用“reserve_str a,b”来调用该宏时，宏里面\p1 的值为 a，\p2 的值为 b。同时，如果省略第一个参数，即使用“reserve_str ,b”来调用该宏，宏参数\p1 使用默认值 0，\p2 的值为 b。

【例 5-11】　下面的代码使用了宏的参数。

```
1     .macro add_data p1=0 p2
2     mv a5, \p1
3     mv a6, \p2
4     add a1, a5, a6
5     .endm
6
7     .globl main
8     main:
9         mv a2, #3
10        mv a3, #3
11
12        add_data a2, a3
13        add_data , a3
```

第 1～5 行实现了 add_data 宏，其中参数 p1 有一个默认值 0。

在第 12 行中，调用 add_data 宏，它会把 a2 和 a3 的值传递给宏的参数 p1 和 p2，最终 a1 的值为 6。

在第 13 行中，同样调用 add_data 宏，但是没有传递第一个参数，此时，add_data 宏会使用 p1 的默认值 0，最终 a1 的计算结果为 3。

如果设置宏的参数有默认值，调用该宏时可以省略这个参数。此时，这个参数会使用默认值，如第 13 行中的参数 1。

在宏参数后面加“:req”表示在宏调用过程中必须传递一个值，否则在编译时会报错。

【例 5-12】　下面的代码片段有问题。

```
1    .macro add_data_1 p1:req p2
2    mv a5, \p1
3    mv a6, \p2
4    add a1, a5, a6
5    .endm
6
7    .globl main
8    main:
9        add_data_1 , a3
```

通过 as 命令编译上述汇编代码，会得到如下编译错误，说明在第 9 行调用 add_data_1 宏时缺失了 p1 参数。

```
root:riscv# as test.S -o test.o
test.S: Assembler messages:
test.S:27: Error: Missing value for required parameter 'p1' of macro 'add_data_1'
test.S:27: Error: illegal operands 'mv a5,'
```

2. 宏的特殊字符

在一些场景下需要把宏的多个参数作为字符串连接在一起。

【例 5-13】　下面的代码使用了宏的多个参数。

```
.macro opcode base length
\base.\length
.endm
```

在这个例子中，opcode 宏想把两个参数串成一个字符串，如 base.length，但是上述的代码是错误的。例如，当调用 opcode store l 时，它并不会生成 store.l 字符串。因为汇编器不知道如何解析参数 base，它不知道 base 参数的结束字符在哪里。

我们可以使用“\()”来告知汇编器，宏的参数什么时候结束，例如，在下面的代码片段中，\base 后面加了“\()”，因此汇编器就知道字母 e 为参数的最后一个字符。

```
.macro opcode base length
\base\().\length
.endm
```

【例 5-14】 下面的代码使用“\()”特殊字符。

```
.macro my_entry, rv, label
        j     rv\()\rv\()_\label
 .endm
```

上述的 j 指令比较有意思，这里出现了两个“rv”和 3 个“\”。其中，第一个“rv”表示 rv 字符，第一个“\()”在汇编宏实现中可以用来表示宏参数的结束字符，第二个“\rv”表示宏的参数 rv，第二个“\()”也用来表示结束字符，最后的“\label”表示宏的参数 label。

假设通过下面的方式调用 my_entry 宏。

```
my_entry  1, irq
```

宏展开之后，上述的 j 指令变成 j rv1_irq。

5.4.6 与文件相关的伪指令

.INCBIN 伪指令可以把文件的二进制数据嵌入当前位置。

【例 5-15】 在下面的代码中，把 benos.bin 的二进制数据嵌入.payload 段中。

```
    .section .payload, "ax", %progbits
     .globl payload_bin
payload_bin:
     .incbin  "benos.bin"
```

.INCLUDE 伪指令可以在汇编代码中插入另外一个文件的汇编代码。例如，下面的汇编代码把 sbi/sbi_payload.S 嵌入当前汇编代码中。

```
.include "sbi/sbi_payload.S"
```

5.5 RISC-V 依赖特性

为了支持几十种处理器体系结构，GNU 汇编器提供一些与特定体系结构相关的额外的伪指令或命令行选项。本节介绍 RISC-V 体系结构中一些特有的命令行选项和伪指令。

5.5.1 RISC-V 特有的命令行选项

GNU 汇编器中 RISC-V 特有的命令行选项如表 5.2 所示。

表 5.2 GNU 汇编器中 RISC-V 特有的命令行选项

选项	说明
-fpic/-fPIC	生成与位置无关的代码
-fno-pic	不生成与位置无关的代码（AS 汇编器默认配置）
-mabi=ABI	指定源代码使用哪个 ABI。可识别的参数是 ilp32 和 lp64，它们分别决定生成 ELF32 或者 ELF64 格式的对象文件
-march=ISA	用来指定目标体系结构，如-march=rv32ima。如果没有指定这个选择，那么 AS 汇编器会读取默认的配置–with-arch=ISA
-misa-spec=ISAspec	选择目标指令集的版本
-mlittle-endian	生成小端的机器码
-mbig-endian	生成大端的机器码

5.5.2　RISC-V 特有的伪指令

RISC-V 特有的伪指令如下。

- .bss：设置当前段为未初始化的数据段。
- .half：把 16 位数当成数据插入汇编代码中。
- .word：把 32 位数当成数据插入汇编代码中。
- .dword：把 64 位数当成数据插入汇编代码中。
- .option：修改特定汇编代码的汇编选项。

5.6　实验

5.6.1　实验 5-1：汇编语言练习——查找最大数

1. 实验目的

通过本实验了解和熟悉 RISC-V 汇编语言。

2. 实验要求

使用 RISC-V 汇编语言来实现如下功能：在给定的一组数中查找最大数，通过 printf()函数输出这个最大数。程序可使用 GCC（RISC-V 版本）工具来编译，并且可在 QEMU+RISC-V+Linux 实验平台上运行。

5.6.2　实验 5-2：汇编语言练习——通过 C 语言调用汇编函数

1. 实验目的

通过本实验了解和熟悉在 C 语言中如何调用汇编函数。

2. 实验要求

使用汇编语言实现一个汇编函数，用于比较两个数的大小并返回较大值，然后用 C 语言代码调用这个汇编函数。程序可使用 GCC（RISC-V 版本）工具来编译，并且可在 QEMU+RISC-V+Linux 实验平台上运行。

5.6.3　实验 5-3：汇编语言练习——通过汇编语言调用 C 函数

1. 实验目的

通过本实验了解和熟悉在汇编语言中如何调用 C 函数。

2. 实验要求

使用 C 语言实现一个函数，用于比较两个数的大小并返回较大值，然后用汇编代码调用这个 C 函数。程序可使用 GCC（RISC-V 版本）来编译，并且可在 QEMU + RISC-V 实验平台上运行。

5.6.4　实验 5-4：使用汇编伪操作实现一张表

1. 实验目的

熟悉常用的汇编伪操作。

2. 实验要求

使用汇编的数据定义伪指令，可以实现表的定义。Linux 内核使用.quad 和.asciz 来定义一个名为 kallsyms 的表，用于表示地址和函数名的对应关系，例如：

```
0x800800 -> func_a
0x800860 -> func_b
0x800880 -> func_c
```

请在 BenOS 里做如下练习。在汇编里定义一个类似的表，然后在 C 语言中根据函数的地址查找表，并且输出函数的名称，如图 5.5 所示。

图 5.5 输出函数名称

5.6.5 实验 5-5：汇编宏的使用

1. 实验目的

熟悉汇编宏的使用。

2. 实验要求

请在 BenOS 里做如下练习。

在汇编文件中通过一个宏实现如下两个汇编函数。

```
long add_1(a, b)
long add_2(a, b)
```

该宏的定义如下。

```
.macro op_func,label, a, b
    //这里调用 add_1()或者 add_2()函数，label 等于 1 或者 2
.endm
```

第 6 章　链接器与链接脚本

本章思考题

1. 什么是链接器？为什么链接器简称 LD？
2. 链接脚本中的输入段和输出段有什么区别？
3. 什么是加载地址和虚拟地址？
4. 在链接脚本中定义如下符号。

```
foo = 0x100
```

foo 和 0x100 分别代表什么？

5. 在 C 语言中，如何引用链接脚本定义的符号？
6. 为了构建一个基于 ROM 的镜像文件，常常会设置输出段的虚拟地址和加载地址不一致，在一个输入段中，如何表示一个段的虚拟地址和加载地址？
7. 什么是链接地址？
8. 当一个程序的代码段的链接地址与加载地址不一致时，我们应该怎么做才能让程序正确运行？
9. 什么是与位置无关的代码？什么是与位置有关的代码？请举例说明在 RISC-V 指令集中哪些指令是与位置无关的指令，哪些是与位置有关的指令。
10. 什么是加载重定位和链接重定位？
11. OpenSBI 和 Linux 内核是如何实现重定位的？
12. 在 Linux 内核中，打开 MMU 之后如何实现重定位？
13. 什么是链接器松弛优化？

本章主要介绍链接器和链接脚本的相关内容。

6.1 链接器

在现代软件工程中，一个大的程序通常由多个源文件组成，其中包含以高级语言编写的源文件以及以汇编语言编写的汇编文件。在编译过程中会分别对这些文件进行编译或者汇编，并生成目标文件。这些目标文件包含代码段、数据段、符号表等内容。而链接指的是把这些目标文件（也包括用到的标准库函数目标文件）的代码段、数据段以及符号表等内容收集起来，并按照某种格式（如 ELF）组合成一个可执行二进制文件的过程。而链接器（linker）用来完成上述链接过程。在操作系统发展的早期并没有链接器的概念，操作系统的加载器（Loader，LD）做了所有的工作。后来操作系统越来越复杂，慢慢出现了链接器，所以 LD 成为链接器的代名词。

链接器采用 AT&T 链接脚本语言，而链接脚本最终会把大量编译（汇编）好的二进制文件（.o

文件）综合成最终可执行二进制文件，也就是把每一个二进制文件整合到一个可执行二进制文件中。这个可执行二进制文件有一个总的代码段/数据段，这就是链接的过程。

GNU 工具链提供了一个名为 ld 的命令，如图 6.1 所示。

下面是 ld 命令简单的用法。

```
$ ld -o mytest  test1.o test2.o -lc
```

```
root:riscv# ld --version
GNU ld (GNU Binutils for Debian) 2.37
Copyright (C) 2021 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or (at your option) a later version.
This program has absolutely no warranty.
```

图 6.1 ld 命令

上述命令把 test1.o、test2.o 以及库文件 libc.a 链接成名为 mytest 的可执行文件。其中，-lc 表示把 C 语言库文件也链接到 mytest 可执行文件中。若上述命令没有使用-T 选项来指定链接脚本，则链接器会默认使用内置的链接脚本。读者可以通过 ld --verbose 命令查看内置链接脚本的内容。

不过，在操作系统实现中常常需要编写一个链接脚本来描述最终可执行文件的代码段/数据段等布局。

【例 6-1】 使用本书的 BenOS，下面的命令可链接、生成 benos.elf 可执行文件，其中 linker.ld 为链接脚本。

```
$ riscv64-linux-gnu-ld -T src/linker.ld  -Map benos.map -o build/benos.elf
 build/printk_c.o build/irq_c.o build/string_c.o
```

ld 命令的常用选项如表 6.1 所示。

表 6.1　　ld 命令的常用选项

选项	说明
-T	指定链接脚本
-Map	输出一个符号表文件
-o	输出最终可执行二进制文件
-b	指定目标代码输入文件的格式
-e	使用指定的符号作为程序的初始执行点
-l	把指定的库文件添加到要链接的文件清单中
-L	把指定的路径添加到搜索库的目录清单中
-S	忽略来自输出文件的调试器符号信息
-s	忽略来自输出文件的所有符号信息
-t	在处理输入文件时显示它们的名称
-Ttext	使用指定的地址作为代码段的起始点
-Tdata	使用指定的地址作为数据段的起始点
-Tbss	使用指定的地址作为未初始化的数据段的起始点
-Bstatic	只使用静态库
-Bdynamic	只使用动态库
-defsym	在输出文件中定义指定的全局符号

6.2 链接脚本

链接器在链接过程中需要使用一个链接脚本，当没有通过-T 选项指定链接脚本时，链接器会使用内置的链接脚本。链接脚本控制如何把输入文件的段整合到输出文件的段里，以及这些段的地址空间布局等。本节主要介绍如何编写一个链接脚本。

6.2.1 一个简单的链接程序

任何一种可执行程序（不论是 ELF 文件还是 EXE 文件）都是由代码段、数据段、未初始化

的数据段等组成的。链接脚本最终会把大量编译好的二进制文件合并为一个可执行二进制文件，也就是把每一个二进制文件整合到一个大文件中。这个大文件有总的代码段、数据段、未初始化的数据段。在 Linux 内核中链接脚本是 vmlinux.lds.S 文件，这个文件有点复杂。我们先看一个简单的链接脚本。

【例 6-2】 如下是一个简单的链接脚本。

```
1   SECTIONS
2   {
3           . = 0x80200000,
4           .text : { *(.text) }
5           . = 0x80210000;
6           .data : { *(.data) }
7           .bss : { *(.bss) }
8   }
```

在第 1 行中，SECTIONS 是链接脚本语法中的关键命令，它用来描述输出文件的内存布局。SECTIONS 命令告诉链接脚本如何把输入文件的段映射到输出文件的各个段，如何将输入段整合为输出段，如何把输出段放入程序地址空间和进程地址空间中。SECTIONS 命令的格式如下。

```
SECTIONS
{
  sections-command
  sections-command
  ...
}
```

sections-command 有如下几种。

- ENTRY 命令，用来设置程序的入口。
- 符号赋值语句，用来给符号赋值。
- 输出段的描述语句。

在第 3 行中，“.”代表当前位置计数器（Location Counter，LC），用于把代码段的链接地址设置为 0x8020 0000。

在第 4 行中，输出文件的代码段由所有输入文件（其中“*”表示所有的.o 文件，即二进制文件）的代码段组成。

在第 5 行中，链接地址变为 0x8021 0000，即重新指定后面的数据段的链接地址。

在第 6 行中，输出文件的数据段由所有输入文件的数据段组成。

在第 7 行中，输出文件的未初始化的数据段由所有输入文件的未初始化的数据段组成。

6.2.2　设置入口点

程序执行的第一条指令称为入口点（entry point）。在链接脚本中，使用 ENTRY 命令设置程序的入口点。例如，设置符号 symbol 为程序的入口点。

```
ENTRY(symbol)
```

除此之外，还有几种方式来设置入口点。链接器会依次尝试下列方法来设置入口点，直到成功为止。

- 使用 GCC 工具链的 LD 命令和-e 选项指定入口点。
- 在链接脚本中通过 ENTRY 命令设置入口点。
- 通过特定符号（如 start 符号）设置入口点。
- 使用代码段的起始地址。
- 使用地址 0。

6.2.3 基本概念

通常链接脚本用来定义如何把多个输入文件的段合并成一个输出文件，描述输入文件的布局。输入文件和输出文件指的是汇编或者编译后的目标文件，它们按照一定的格式（如 ELF）组成，只不过输出文件具有可执行属性。这些目标文件都由一系列的段组成。段是目标文件中具有相同特征的最小可处理信息单元，不同的段用来描述目标文件中不同类型的信息以及特征。

在链接脚本中，我们把输入文件中的一个段称为输入段（input section），把输出文件中的一个段称为输出段（output section）。输出段告诉链接器最终的可执行文件在内存中是如何布局的。输入段告诉链接器如何将输入文件映射到内存布局中。

输出段和输入段包括段的名字、大小、可加载（loadable）属性以及可分配（allocatable）属性等。可加载属性用于在运行时加载这些段的内容到内存中。可分配属性用于在内存中预留一个区域，并且不会加载这个区域的内容。

链接脚本中还有两个关于段的地址，它们分别是加载地址和虚拟地址。加载地址是加载时段所在的地址，虚拟地址是运行时段所在的地址，也称为运行地址。通常情况下，这两个地址是相同的。不过，它们也有可能不相同。例如，一个代码段被加载到只读存储器（Read-Only Memory，ROM）中，在程序启动时被复制到随机存储器（Random Access Memory，RAM）中。在这种情况下，ROM 地址将是加载地址，RAM 地址将是虚拟地址。

6.2.4 符号赋值与引用

在链接脚本中，符号可以像 C 语言一样进行赋值和操作，允许的操作包括赋值、加法、减法、乘法、除法、左移、右移、与、或等。

```
symbol = expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
symbol <<= expression ;
symbol >>= expression ;
symbol &= expression ;
symbol |= expression ;
```

高级语言（如 C 语言）常常需要引用链接脚本定义的符号。链接脚本定义的符号与 C 语言中定义的符号有本质的区别。例如，在 C 语言中定义全局变量 foo 并且赋值为 100。

```
int foo = 100
```

当在高级语言（如 C 语言）中声明一个符号时，编译器在程序内存中保留足够的空间来保存符号的值。另外，编译器在程序的符号表中创建一个保存该符号地址的条目，即符号表包含保存符号值的内存块的地址。因此，编译器会在符号表中存储 foo 符号。这个符号保存在某个内存地址里，这个内存地址用来存储初始值 100。当程序再一次访问 foo 变量时，例如，设置 foo 为 1，程序就在符号表中查找符号 foo，获取与该符号关联的内存地址，然后把 1 写入该内存地址。而链接脚本定义的符号仅仅在符号表中创建了一个符号，并没有分配内存来存储这个符号。也就是说，它有地址，但是没有存储内容。所以链接脚本中定义的符号只代表一个地址，而链接器不能保证这个地址存储了内容。例如，在链接脚本中定义一个 foo 符号并赋值。

```
foo = 0x100;
```

链接器会在符号表中创建一个名为 foo 的符号，0x100 表示内存地址的位置，但是地址 0x100 没有存储任何特别的东西。换句话说，foo 符号仅仅用来记录某个内存地址。

在实际编程中，我们常常需要访问链接脚本中定义的符号。例 6-3 在链接脚本中定义 ROM 的起始地址 start_of_ROM、ROM 的结束地址 end_of_ROM 以及 FLASH 的起始地址 start_of_FLASH，这样在 C 语言程序中就可以访问这些地址。例如，把 ROM 的内容复制到 FLASH 中。

【例 6-3】 下面是链接脚本。

```
start_of_ROM = .ROM;
end_of_ROM = .ROM + sizeof (.ROM);
start_of_FLASH = .FLASH;
```

在上述链接脚本中，ROM 和 FLASH 分别表示存储在 ROM 与闪存中的段。在 C 语言中，我们可以通过如下代码片段把 ROM 的内容搬移到 FLASH 中。

```
extern char start_of_ROM, end_of_ROM, start_of_FLASH;
memcpy (& start_of_FLASH, & start_of_ROM, & end_of_ROM - & start_of_ROM);
```

上面的 C 语言代码使用“&”符号来获取符号的地址。这些符号在 C 语言中也可以看成数组，所以上述 C 语言代码改写成如下代码。

```
extern char start_of_ROM[], end_of_ROM[], start_of_FLASH[];

memcpy (start_of_FLASH, start_of_ROM, end_of_ROM - start_of_ROM);
```

一个常用的编程技巧是在链接脚本里为每个段都设置一些符号，以方便 C 语言访问每个段的起始地址和结束地址。例 6-4 中的链接脚本定义了代码段的起始地址（start_of_text）、代码段的结束地址（end_of_text）、数据段的起始地址（start_of_data）以及数据段的结束地址（end_of_data）。

【例 6-4】 下面是一个链接脚本。

```
SECTIONS
{
    start_of_text = . ;
    .text: { *(.text) }
    end_of_text = . ;

    start_of_data = . ;
    .data: { *(.data) }
    end_of_data = . ;
}
```

在 C 语言中，使用以下代码可以很方便地访问这些段的起始地址和结束地址。

```
extern char start_of_text[];
extern char end_of_text[];
extern char start_of_data[];
extern char end_of_data[];
```

6.2.5　当前位置计数器

有一个特殊的符号“.”，它表示当前位置计数器。下面举例说明。

【例 6-5】 下面的链接脚本使用了当前位置计数器。

```
1     floating_point = 0;
2     SECTIONS
3     {
4         .text :
5         {
6         *(.text)
7         _etext = .;
8         }
9         _bdata = (. + 3) & ~ 3;
10        .data : { *(.data) }
11    }
```

上述链接脚本中，第 7 行和第 9 行使用了当前位置计数器。在第 7 行中，_etext 设置为当前位置，当前位置为代码段结束的地方。在第 9 行中，设置_bdata 的起始地址为当前位置后下一个与 4 字节对齐的地方。

6.2.6 SECTIONS 命令

SECTIONS 命令告诉链接器如何把输入段映射到输出段，以及如何在内存中存放这些输出段。

1. 输出段

输出段的描述格式如下。

```
section [address] [(type)] :
  [AT(lma)]
  [ALIGN(section_align)]
  [constraint]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
```

其中，部分内容的含义如下。

- section：段的名字，如.text、.data 等。
- address：虚拟地址。
- type：输出段的属性。
- lma：加载地址。
- ALIGN：对齐要求。
- output-section-command：描述输入段如何映射到输出段。
- region：特定的内存区域。
- phdr：特定的程序段（program segment）。

一个输出段有两个地址，分别是虚拟内存地址（Virtual Memory Address，VMA）和加载内存地址（Load Memory Address，LMA）。

如果没有通过“AT”指定 LMA，那么 LMA = VMA，即加载地址等于虚拟地址。但在嵌入式系统中，经常存在加载地址和虚拟地址不同的情况，如将镜像文件加载到开发板的闪存中（由 LMA 指定），而 BootLoader 将闪存中的镜像文件复制到同步动态随机存储器（Synchronous Dynamic Random Access Memory，SDRAM）中（由 VMA 指定）。

2. 输入段

输入段用来告诉链接器如何将输入文件映射到内存布局。输入段包括输入文件以及对应的段。通常，使用通配符来包含某些特定的段，例如：

```
*(.text)
```

这里的“*”是一个通配符，可以匹配任何文件名的代码段。另外，如果想从所有文件中剔除一些文件，可以使用“EXCLUDE_FILE”列出哪些文件是需要剔除的，剩余的文件用作输入段，例如：

```
EXCLUDE_FILE (*crtend.o *otherfile.o) *(.ctors)
```

在上面的代码中，除 crtend.o 和 otherfile.o 文件之外，把剩余文件的 ctors 段加入输入段中。

下面两条语句是有区别的。

```
*(.text .rodata)
*(.text) *(.rodata)
```

第一条语句按照加入输入文件的顺序把相应的代码段和只读数据段加入；而第二句条语句先加入所有输入文件的代码段，再加入所有输入文件的只读数据段。

```
*(EXCLUDE_FILE (*somefile.o) .text .rodata)
```

如果“EXCLUDE_FILE”后面跟着一串段列表，那么只有第一个段起到剔除的作用。例如在上述语句中，除 somefile.o 文件的代码段之外，把其他文件的代码段加入输入段里。另外，只读数据段不在剔除之列，即所有文件的只读数据段都加入输入段里。

如果你想同时剔除 somefile.o 文件的代码段和只读数据段，可以这么写。

```
*(EXCLUDE_FILE (*somefile.o) .text EXCLUDE_FILE (*somefile.o) .rodata)
```

或者使用以下语句。

```
EXCLUDE_FILE (*somefile.o) *(.text .rodata)
```

要指定文件名中特定的段，例如，把 data.o 文件中的数据段加入输入段里，使用以下代码。

```
data.o(.data)
```

下面结合例子来说明输入段的作用。

【例 6-6】　下面实现了一个链接脚本。

```
1     SECTIONS {
2         outputa 0x10000 :
3         {
4             all.o
5             foo.o (.input1)
6         }
7         outputb :
8         {
9             foo.o (.input2)
10            foo1.o (.input1)
11        }
12        outputc :
13        {
14            *(.input1)
15            *(.input2)
16        }
17    }
```

这个链接脚本一共有 3 个输出段——outputa、outputb 和 outputc。outputa 输出段的起始地址为 0x10000，首先在这个起始地址里存储 all.o 文件中所有的段，然后存储 foo.o 文件的 input1 段。outputb 输出段包括 foo.o 文件的 input2 段以及 foo1.o 文件的 input1 段。outputc 段包括所有文件的 input1 段和所有文件的 input2 段。

3. 例子

通常，为了构建一个基于 ROM 的镜像文件，要设置输出段的虚拟地址和加载地址不一致。镜像文件存储在 ROM 中，运行程序时需要把镜像文件复制到 RAM 中。此时，ROM 中的地址为加载地址，RAM 中的地址为虚拟地址，即运行地址。在例 6-7 中，链接脚本会创建 3 个段。其中，.text 段的虚拟地址和加载地址均为 0x1000，.mdata（用户自定义的数据）段的虚拟地址设置为 0x2000，但是通过 AT 符号指定了加载地址是代码段的结束地址，而符号_data 指定了.mdata 段的虚拟地址为 0x2000。.bss 段的虚拟地址是 0x3000。

【例 6-7】　创建 3 个段。

```
SECTIONS
  {
  .text 0x1000 : { *(.text) _etext = . ; }
  .mdata 0x2000 :
    AT ( ADDR (.text) + SIZEOF (.text) )
```

```
    { _data = . ; *(.data); _edata = . ;  }
  .bss 0x3000 :
    { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

.mdata 段的加载地址和链接地址（虚拟地址）不一样，因此程序的初始化代码需要把.mdata 段从 ROM 中的加载地址复制到 SDRAM 中的虚拟地址。如图 6.2 所示，.mdata 段的加载地址在_etext 起始的地方，.mdata 段的虚拟地址在_data 起始的地方，.mdata 段的大小为_edata－_data。下面这段代码把.mdata 段从_etext 起始的地方复制到从_data 起始的地方。

```
<程序初始化>

extern char _etext, _data, _edata, _bstart, _bend;
char *src = &_etext;
char *dst = &_data;

/*ROM 中包含.mdata 段，.mdata 段位于.text 段的结束地址处，把.mdata 段复制到.mdata 段的虚拟地址处*/
while (dst < &_edata)
  *dst++ = *src++;

/*清除.bss 段*/
for (dst = &_bstart; dst< &_bend; dst++)
  *dst = 0;
```

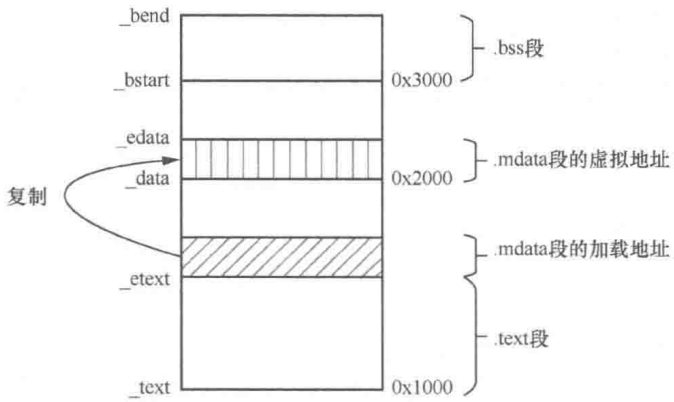


图 6.2　复制数据到虚拟地址处

6.2.7　常用的内置函数

链接脚本语言包含一些内置函数。

1. ABSOLUTE(exp)

ABSOLUTE(exp)返回表达式的绝对值。它主要用于在段定义中给符号赋绝对值。

【例 6-8】　下面的链接脚本使用了 ABSOLUTE()内置函数。

```
SECTIONS
{
     . = 0xb0000,
     .my_offset : {
          my_offset1 = ABSOLUTE(0x100);
          my_offset2 = (0x100);
     }
}
```

上述链接脚本定义了一个名为.my_offset 的段。其中，符号 my_offset1 使用了 ABSOLUTE()内置函数，它把数值 0x100 赋值给符号 my_offset1；而符号 my_offset2 没有使用内置函数，因此符号 my_offset2 属于.my_offset 段里的符号，于是符号 my_offset2 的地址为 0xB0000 + 0x100。下面是通过链接器生成的符号表信息。

```
my_offset      0x00000000000b0000                0x0
               0x0000000000000100                my_offset1 = ABSOLUTE (0x100)
               0x00000000000b0100                my_offset2 = 0x100
```

2. ADDR(section)

ADDR(section)返回段的虚拟地址。

3. ALIGN(align)

ALIGN(align)返回下一个与 align 字节对齐的地址，它是基于当前的位置来计算对齐地址的。

【例 6-9】 下面是一个使用 ALIGN(align)的链接脚本。

```
SECTIONS {
     ...
     .data ALIGN(0x2000): {
     *(.data)
     variable = ALIGN(0x8000);
     }
     ...
}
```

上述链接脚本的.data 段会设置在下一个与 0x2000 字节对齐的地址上。另外，定义一个 variable 变量，这个变量的地址是下一个与 0x8000 字节对齐的地址。

4. SIZEOF(section)

.SIZEOF(section)返回一个段的大小。

【例 6-10】 在下面的代码中，symbol_1 和 symbol_2 都用来返回.output 段的大小。

```
SECTIONS{
     ...
     .output {
         .start = . ;
         ...
         .end = . ;
     }
     symbol_1 = .end - .start ;
     symbol_2 = SIZEOF(.output);
     ...
}
```

5. PROVIDE

PROVIDE 内置函数从链接脚本中导出一个符号，不过只有当这个符号在其他地方没有定义并且没有链接时才会使用。

【例 6-11】 在下面的链接脚本里定义 my_label 符号。

```
SECTIONS
{

  . = 0x80500000,
  PROVIDE (my_label = .);

  . = 0x80200000,
  .text : { *(.text.boot) }

  ...
}
```

my_label 符号在链接脚本中定义的地址为 0x8050 0000，然后在汇编代码中也定义一个 my_label 符号。

```
1    .global my_label
2    my_label:
3        .byte 8
4
```

```
5    test_provide:
6            la a0, my_label
7            ret
```

在这个场景下，LA 指令会使用第 2 行定义的 my_label 符号的地址，而不会使用链接脚本中定义的 my_label 符号。假设汇编代码中没有定义 my_label 符号（见第 1～3 行），那么 LA 指令将加载链接脚本中的 my_label 符号的地址（0x8050 0000）到 a0 寄存器中。

6. INCLUDE

INCLUDE 函数可以引入另外的链接脚本。例如，下面的链接脚本直接引入 sbi/sbi_base.ld 链接脚本。

```
SECTIONS
{
  INCLUDE "sbi/sbi_base.ld"
}
```

7. 其他内置函数

其他内置函数如下。

- LOADADDR(section)：返回段的加载地址。
- MAX(exp1, exp2)：返回两个表达式中的最大值。
- MIN(exp1, exp2)：返回两个表达式中的最小值。

6.3 加载重定位

我们首先要知道下面几个重要概念。

- 加载地址：存储代码的物理地址，在 GNU 链接脚本里称为 LMA。例如，RISC-V 处理器上电复位后是从异常向量表开始取第一条指令的，所以通常这个地方存放代码最开始的部分，如异常向量表的处理代码。
- 运行地址：程序运行时的地址，在 GNU 链接脚本里称为 VMA。
- 链接地址：在编译、链接时指定的地址，编程人员设想将来程序要运行的地址。程序中所有标号的地址在链接后便确定了，不管程序在哪里运行都不会改变。当使用 riscv64-linux-gnu-objdump（简称 objdump）工具进行反汇编时，查看的就是链接地址。

链接地址和运行地址可以相同，也可以不同。那运行地址和链接地址什么时候不相同？什么时候相同呢？本节介绍**加载重定位**（load relocation）。我们分别以 BenOS 和 U-Boot/Linux 为例说明。

6.3.1 BenOS 重定位

QEMU Virt 平台上电之后，首先运行在 M 模式，跳转到 0x8000 0000 地址处。此时运行的是 SBI 固件，如 OpenSBI 或者 MySBI。经过 SBI 固件的初始化，然后切换到 S 模式，并且跳转到 0x8020 0000 地址处，启动 BenOS。

1. 链接地址与运行地址和加载地址相同的情况

【例 6-12】 BenOS 的一个链接脚本如下所示。

```
1    SECTIONS
2    {
3         . = 0x80200000;
4        .text.boot : { *(.text.boot) }
5        .text : { *(.text) }
6        .rodata : { *(.rodata) }
```

```
7         .data : { *(.data) }
8         . = ALIGN(0x8);
9         bss_begin = .;
10        .bss : { *(.bss*) }
11        bss_end = .;
12    }
```

链接地址就是从 0x8020 0000 开始的，此时加载地址也是 0x8020 0000，运行地址也是 0x8020 0000。我们打开 benos.map 文件来查看链接地址，如图 6.3 所示，.text.boot 的链接地址为 0x8020 0000。

```
rlk@master:benos$ cat benos.map

Memory Configuration

Name             Origin             Length             Attributes
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

                 0x0000000080200000                 . = 0x80200000

.text.boot       0x0000000080200000       0x3c
 *(.text.boot)
 .text.boot      0x0000000080200000       0x3c build/benos/boot_s.o
                 0x0000000080200000                _start

.text            0x0000000080200040      0x894
 *(.text)
 .text           0x0000000080200040      0x170 build/benos/uart_c.o
                 0x0000000080200040                uart_send
                 0x0000000080200094                uart_send_string
                 0x00000000802000f4                uart_init
```

图 6.3　链接地址为 0x80200000

2. 加载地址与链接地址不相同的情况

【例 6-13】 BenOS 的另一个链接脚本如下所示。

```
1     TEXT_ROM = 0x80300000;
2
3     SECTIONS
4     {
5
6         . = 0x80200000,
7
8         _text_boot = .;
9         .text.boot : { *(.text.boot) }
10        _etext_boot = .;
11
12        _text = .;
13        .text : AT(TEXT_ROM)
14        {
15            *(.text)
16        }
17        _etext = .;
18
19        ...
20    }
```

在第 13 行里使用 AT 表明代码段的加载地址为 TEXT_ROM（0x8030 0000），此时代码段的加载地址就与链接地址不一样，而链接地址与运行地址一样。代码段的链接地址可以通过 benos.map 来查看。如图 6.4 所示，代码段的起始链接地址可以通过符号表中的_text 符号获得，地址为 0x8020 0060，结束链接地址可以通过_etext 符号来获取。

```
rlk@master:benos$ cat benos.map

Memory Configuration

Name             Origin             Length             Attributes
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

                 0x0000000080300000                 TEXT_ROM = 0x80300000
                 0x0000000080200000                 . = 0x80200000
                 0x0000000080200000                 _text_boot = .

.text.boot       0x0000000080200000       0x5c
 *(.text.boot)
 .text.boot      0x0000000080200000       0x5c build/benos/boot_s.o
                 0x0000000080200000                 _start
                 0x000000008020005c                 _etext_boot = .
                 0x000000008020005c                 _text = .

.text            0x0000000080200060      0x894 load address 0x0000000080300000
 *(.text)
 .text           0x0000000080200060      0x170 build/benos/uart_c.o
                 0x0000000080200060                 uart_send
                 0x00000000802000b4                 uart_send_string
                 0x0000000080200114                 uart_init
```

图 6.4　代码段的链接地址

在这种情况下，如果想要 BenOS 正常运行，我们需要把代码段从加载地址复制到链接地址。

```
.globl _start
_start:
        /*
            假设代码段存储在 ROM 中，而 ROM 的地址在 0x8030 0000
```

```
            我们需要把代码段从加载地址复制到链接地址
         */
        la t0, TEXT_ROM
        la t1, _text
        la t2, _etext
.L0:
     ld  a5, (t0)
     sd a5, (t1)
     addi t1, t1, 8
     addi t0, t0, 8
     bltu t1, t2, .L0
```

3. 运行地址与链接地址不相同的情况

当 BenOS 的 MMU 使能之后，我们可以把双倍数据速率（Double Data Rate，DDR）内存映射到内核空间。

【例 6-14】 使用下面的链接脚本。

```
1       SECTIONS
2       {
3           /*
4           * 设置 BenOS 的链接地址 0xFFFF 0000 0000 0000
5           */
6           . = 0xffff000000000000,
7
8           _text_boot = .;
9           .text.boot : { *(.text.boot) }
10          _etext_boot = .;
11
12          . = ALIGN(8);
13          _text = .;
14          .text :
15          {
16                      *(.text)
17          }
18          . = ALIGN(8);
19          _etext = .;
20
21              ...
22          }
```

在第 6 行中，当前位置计数器把代码段的链接地址设置在 0xFFFF 0000 0000 0000，这是内核空间的一个地址。QEMU 上电复位后，benos.bin 加载并跳转到 0x8020 0000 地址处，此时，运行地址和加载地址都为 0x8020 0000，而链接地址为 0xFFFF 0000 0000 0000，如图 6.5 所示。

我们需要在汇编代码里初始化 MMU，并且把物理 DDR 内存映射到内核空间里，然后做一次重定位操作，让 CPU 的运行地址重定位到链接地址处，这种做法在 U-Boot 和 Linux 内核中很常见。

```
rlk@master:benos$ cat benos.map

Memory Configuration

Name             Origin             Length             Attributes
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

                 0xffff000000000000                 . = 0xffff000000000000
                 0xffff000000000000                 _text_boot = .

.text.boot       0xffff000000000000        0x5c
 *(.text.boot)
 .text.boot      0xffff000000000000        0x5c build/benos/boot_s.o
                 0xffff000000000000                 _start
                 0xffff00000000005c                 _etext_boot = .
                 0xffff00000000005c                 _text = .
```

图 6.5 链接地址与运行地址不一样

6.3.2 OpenSBI 和 Linux 内核重定位

我们以一块 RISC-V 开发板为例，假设芯片内部有 SRAM，起始地址为 0x0，DDR 内存的起始地址为 0x8000 0000。SBI 固件（如 OpenSBI）运行在 M 模式，起始地址为 0x8000 0000，U-Boot 和 Linux 内核运行在 S 模式，起始地址为 0x8020 0000。下面是芯片启动的一般流程。

（1）芯片上电，运行 BOOT ROM 的程序。

（2）BOOT ROM 程序会初始化 Nor Flash 等外部存储介质，把 OpenSBI 加载到 DDR 内存

中，并跳转到 DDR 内存中。

（3）OpenSBI 切换到 S 模式，并运行 S 模式下的 U-Boot。

（4）U-Boot 初始化硬件和启动环境，跳转到 Linux 内核并运行。

在第（2）步中，由于 OpenSBI 的镜像太大了，SRAM 放不下，因此必须要把镜像放在 DDR 内存中。通常 OpenSBI 在编译时把链接地址都设置到 DDR 内存中，也就是 0x8000 0000 地址处，于是运行地址和链接地址就不一样了。既然运行地址为 0x0，链接地址变成 0x8000 0000，那么程序为什么还能运行呢？

这就涉及汇编编程的一个重要问题，就是位置无关的代码和位置有关的代码。

- 位置无关的代码：从字面意思看，指令的执行是与内存地址无关的；无论运行地址和链接地址相同或者不相同，该指令都能正常运行。在汇编语言中，像 J、JAL、MV 等指令属于位置无关指令，不管程序在哪个位置，它们都能正确地运行，指令的地址属性是基于 PC 值相对寻址的，相当于[PC+offset]。
- 位置有关的代码：从字面意思看，指令的执行是与内存地址有关的，和当前 PC 值无关。在 RISC-V 汇编语言里，通过修改 ra 寄存器的值实现相对跳转，示例如下。

```
li a1, PAGE_OFFSET
add ra, ra, a1
```

因此，如果通过修改返回地址为链接地址，当函数返回时，就会跳转到链接地址处。这个过程叫作重定位。在重定位之前，程序只能执行和位置无关的一些汇编代码。

为什么要刻意设置加载地址、运行地址以及链接地址不一样呢？

如果所有代码都在 ROM（或 NOR Flash 存储器）中执行，那么链接地址可以与加载地址相同。而在实际项目应用中，往往想要把程序加载到 DDR 内存中，DDR 内存的访问速度比 ROM 的要快很多，而且容量也大，所以设置链接地址到 DDR 内存中，而程序的加载地址设置到 ROM 中，这两个地址是不相同的。如何让程序能在链接地址上运行呢？常见的思路就是让程序的加载地址等于 ROM 起始地址（或者片内 SRAM 地址），而链接地址等于 DDR 内存中某一处的起始地址（暂且称为 ram_start）。程序先从 ROM 中启动，最先启动的部分要实现代码复制功能（把 ROM 中的全部代码复制到 DDR 内存中），并通过位置有关的指令跳转到 DDR 内存中，也就是在链接地址里运行（J 指令没法实现这个跳转）。上述重定位过程在 OpenSBI 中实现，如图 6.6 所示。

从 OpenSBI 跳转到 U-Boot 是一次模式切换的过程，由处理器完成模式切换与跳转。当从 U-Boot 跳转到 Linux 内核中时，U-Boot 需要把 Linux 内核镜像内容复制到 DDR 内存中，然后跳转到内核入口地址处（_start）。当跳转到内核入口地址时，程序运行在运行地址，即 DDR 内存中的地址。但是我们从 vmlinux 看到的_start 的链接地址是虚拟地址（假设为 0xFFFF FFE0 0000 0000，用 PAGE_OFFSET 宏来表示）。内核启动汇编代码也需要一个重定位过程。这个重定位过程在 relocate 汇编函数中完成，relocate 汇编函数的主要功能是实现重定位。启动 MMU 之后，通过 LI 指令把 PAGE_OFFSET 宏的值加载到 a1 寄存器，然后修改返回地址（ra 寄存器的值）来跳转到内核空间的链接地址处，从而实现重定位，如图 6.7 所示。relocate 汇编函数简化后的代码片段如下。

```
<linux5.15/arch/riscv/kernel/head.S>
1    relocate:
2        /* 重定位返回地址 */
3        li a1, PAGE_OFFSET
4        la a0, _start
5        sub a1, a1, a0
6        add ra, ra, a1
7
8        ...
9        ret
```

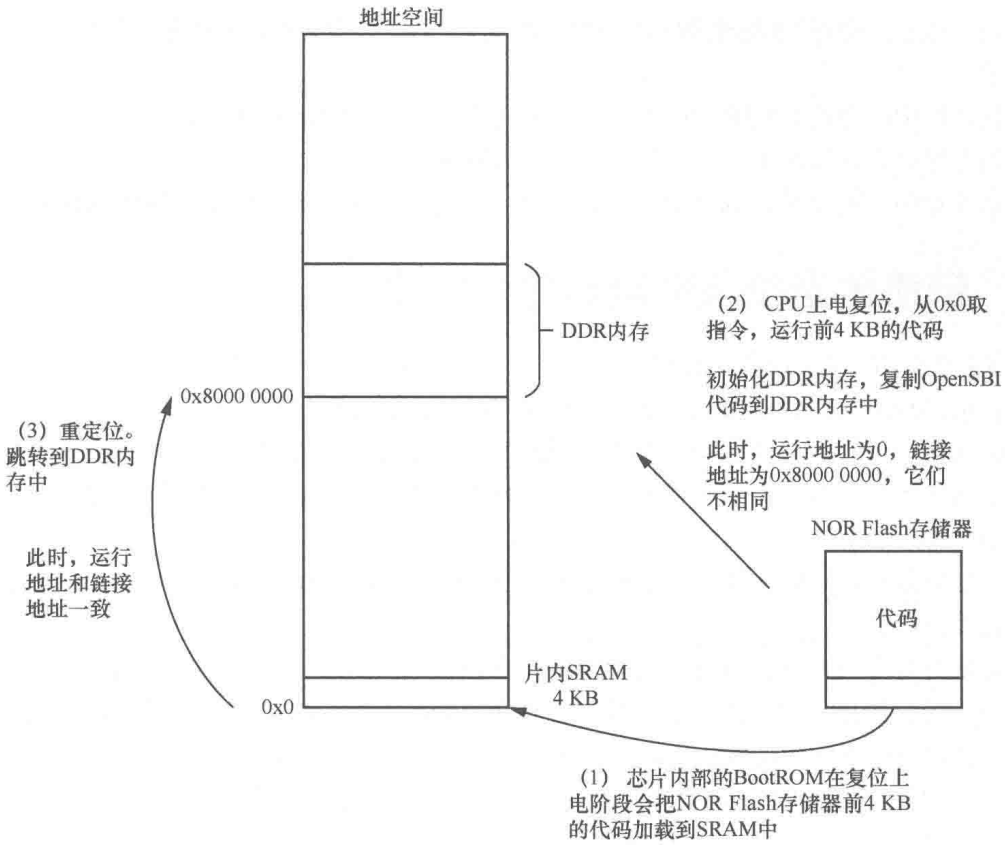


图 6.6 OpenSBI 启动时的重定位过程

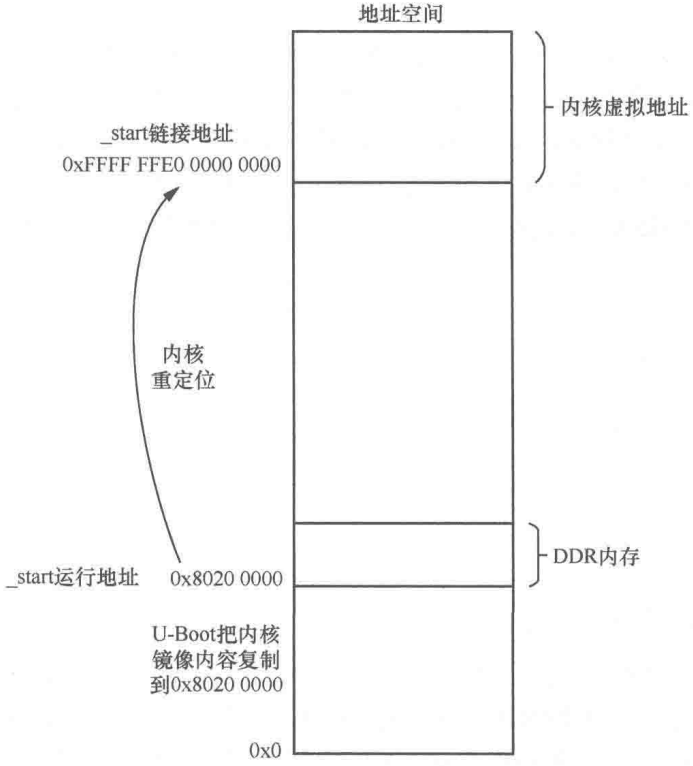


图 6.7 Linux 内核镜像地址重定位

在第 3 行中，通过 LI 加载 PAGE_OFFSET 宏的值到 a1 寄存器，PAGE_OFFSET 为_start 的链接地址。

在第 4 行中，通过 LA 指令加载_start 的运行地址，它为 0x8020 0000。

在第 5 行中，计算上述两个值的差值，即偏移量。

在第 6 行中，把当前 ra 寄存器加上偏移量，就等于该函数返回地址的链接地址了。

6.4 链接重定位与链接器松弛优化

从 5.1 节介绍可知，源代码编译成可执行二进制文件要经过预处理、编译、汇编以及链接等阶段。在编译阶段，编译器是无法确定每个符号的最终链接地址的，因此编译阶段生成的可重定位目标文件中所有的符号都暂时设定为 0x0 地址。链接器在链接最终可执行二进制文件时才具有全局内存地址布局图，这些符号的最终地址由链接器来分配和确定，这个过程称为**链接重定位**（link relocation）。

在链接阶段有一种优化技术——链接器松弛优化（linker relaxation optimization），它旨在减少不必要的指令。

通常精简指令处理器体系结构需要两条指令来实现对一个符号地址的访问。一条指令处理符号地址的高位部分，另一条指令处理符号地址的低位部分。但是，在链接阶段，通常我们可以使用一条指令完成上述操作从而达到优化目的，这就是链接器松弛优化的作用。

对于 RISC-V 处理器来说，链接器松弛优化技术主要涉及两方面。

- 函数跳转优化。
- 符号地址访问优化。

6.4.1 链接重定位

在介绍链接器松弛优化技术之前，我们先简单介绍一下链接重定位，本节关注的是静态链接。链接的目的是把多个可重定位目标文件链接成一个统一的可执行二进制文件，这个过程包括地址和空间分配、符号解析、重定位等。链接器分配地址和空间之后，就进入符号解析和重定位阶段。我们结合一个简单的例子说明链接器是如何做重定位的。

【例 6-15】 下面是 test.c 的代码，它是怎么引用全局变量 *a* 以及如何调用 foo()函数的？

```
<test.c>

int a = 5;

int foo(void)
{
   return a;
}

int main(void)
{
    foo();
}
```

首先，编译 test.c 文件。

```
$ riscv64-linux-gnu-gcc test.c -o test --save-temps -mno-relax
```

- --save-temps 表示保留编译过程中所有产生的中间文件，例如，test.i 是预处理后的文件，test.s 是编译后的汇编文件，test.o 是汇编后的可重定位目标文件，test 为最终编译的可执行二进制文件。

- -mno-relax 表示关闭链接器松弛优化。GCC 默认打开链接器松弛优化，本节中，暂时关闭该优化。

编译后的汇编文件 test.s 如下。

```
<test.s>

1    a:
2        .word  5
3        .text
4
5    foo:
6        ...
7        lla  a5,a
8        lw  a5,0(a5)
9        ...
10
11   main:
12       ...
13       call  foo
14       ...
```

从上述汇编代码可知，对全局变量 *a* 的地址的访问采用了 LLA 伪指令（见第 7 行），采用 CALL 伪指令调用 foo()函数（见第 13 行）。LLA 和 CALL 伪指令在汇编阶段会解析成多条汇编指令。

使用 riscv64-linux-gnu-objdump 命令查看可重定位目标文件 test.o 对应的反汇编代码。

```
rlk@master:riscv_example$ riscv64-linux-gnu-objdump -d  -r test.o

test.o:     file format elf64-littleriscv


Disassembly of section .text:

0000000000000000 <foo>:
   0: 1141              addi sp,sp,-16
   2: e422              sd   s0,8(sp)
   4: 0800              addi s0,sp,16
   6: 00000797          auipc    a5,0x0    //此处需要重定位
          6: R_RISCV_PCREL_HI20  a
   a: 00078793          mv   a5,a5
          a: R_RISCV_PCREL_LO12_I  .L0
   e: 439c              lw   a5,0(a5)
  10: 853e              mv   a0,a5
  12: 6422              ld   s0,8(sp)
  14: 0141              addi sp,sp,16
  16: 8082              ret

0000000000000018 <main>:
  18: 1141              addi sp,sp,-16
  1a: e406              sd   ra,8(sp)
  1c: e022              sd   s0,0(sp)
  1e: 0800              addi s0,sp,16
  20: 00000097          auipc    ra,0x0    //此处需要重定位
          20: R_RISCV_CALL  foo
  24: 000080e7          jalr ra # 20 <main+0x8>
  28: 4781              li   a5,0
  2a: 853e              mv   a0,a5
  2c: 60a2              ld   ra,8(sp)
  2e: 6402              ld   s0,0(sp)
  30: 0141              addi sp,sp,16
  32: 8082              ret
```

LLA 伪指令被解析成如下指令。

```
auipc  a5,0x0
mv  a5,a5
```

CALL 伪指令被解析成如下指令。

```
auipc  ra,0x
jalr  ra
```

因为编译器在编译成可重定位目标文件时并不知道全局变量 *a* 和 foo()函数的最终地址，所以这里暂时使用 0x0 作为 PC 相对地址的偏移量。链接器完成地址和空间分配之后，就确定了符号的最终地址。链接器需要根据符号的最终地址来修正上述指令，以实现重定位。链接器怎么知道哪些指令需要进行重定位与修正呢？其实，在 ELF 文件规范里定义了一个重定位表(relocation table)，它们存储在重定位段里。以代码段为例，如果代码段中有需要重定位的地方，那么会有一个叫作“.rela.text”的重定位段。

使用 riscv64-linux-gnu-objdump 命令查看重定位段的信息。

```
rlk@master:riscv_example$ riscv64-linux-gnu-objdump -r test.o

test.o:     file format elf64-littleriscv

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000006 R_RISCV_PCREL_HI20    a
000000000000000a R_RISCV_PCREL_LO12_I  .L0
0000000000000020 R_RISCV_CALL          foo
```

- RELOCATION RECORDS FOR [.text]表示下面的信息是代码段的重定位表。OFFSET 表示要重定位的相对地址。例如，6 表示代码段偏移 0x6 的地方。
- TYPE 表示重定位的类型。
- VALUE 表示要重定位的符号。

每个处理器体系结构的指令集都不一样，所以各自有一套独立的重定位类型。RISC-V 常用的重定位类型如表 6.2 所示，完整的重定位类型见 RISC-V ABIs Specification，Version 0.01。

表 6.2　　RISC-V 常用的重定位类型

编号	重定位类型	说明	计算公式
18	R_RISCV_CALL	函数调用，用于 CALL 和 TAIL 伪指令	$S+A-P$
23	R_RISCV_PCREL_HI20	PC 相对寻址（高 20 位部分）	$S+A-P$
24	R_RISCV_PCREL_LO12_I	PC 相对寻址（低 12 位部分），用于加载指令	$S-P$
25	R_RISCV_PCREL_LO12_S	PC 相对寻址（低 12 位部分），用于存储指令	$S-P$
26	R_RISCV_HI20	绝对地址寻址（高 20 位部分）	$S+A$
27	R_RISCV_LO12_I	绝对地址寻址（低 12 位部分），用于加载指令	$S+A$
28	R_RISCV_LO12_S	绝对地址寻址（低 12 位部分），用于存储指令	$S+A$
51	R_RISCV_RELAX	表示指令会被链接器松弛优化	—

表 6.2 中，计算公式中变量的含义如下。

- *S*：表示符号的最终链接地址。
- *A*：表示需要额外附加（appended）的字节数。
- *P*：重定位的位置，即重定位的那条指令的 PC 值。

我们先来看对全局变量 a 的重定位是如何修正的。它需要重定位的地方是偏移量为 0x6 的位置，重定位类型为 R_RISCV_PCREL_HI20，表示 PC 相对寻址。从表 6.2 中的计算公式可知，重定位需要修正的值 offset = $S+A-P$。下面是 *S*、*A* 和 *P* 的计算过程。

首先，通过 riscv64-linux-gnu-readelf 命令读取 test 可执行二进制文件的符号。

```
$ riscv64-linux-gnu-readelf -s test

Symbol table '.symtab' contains 66 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
```

```
    ...
    56: 00000000000005fc    24 FUNC    GLOBAL DEFAULT   12 foo
    60: 0000000000002008     4 OBJECT  GLOBAL DEFAULT   19 a
```

从上述日志可知，全局变量 *a* 的地址为 0x2008。因此，计算公式中的 *S* 为 0x2008。

然后，查看 test 可执行二进制文件的反汇编代码。

```
$ riscv64-linux-gnu-objdump -d test

00000000000005fc <foo>:
...
602: 00002797          auipc  a5,0x2
606: a0678793          addi  a5,a5,-1530 # 2008 <a>
60a: 439c              lw  a5,0(a5)
...

0000000000000614 <main>:
...
61c: 00000097          auipc  ra,0x0
620: fe0080e7          jalr  -32(ra) # 5fc <foo>
...
```

从上述反汇编结果可知，foo()函数的地址为 0x5FC，需要做重定位的偏移量为 0x6，因此需要做重定位的位置是 0x5FC + 0x6 = 0x602，即计算公式中的 *P* 为 0x602。

接着，通过 riscv64-linux-gnu-readelf 命令读取 test.o 目标文件的重定位表信息。

```
 $ riscv64-linux-gnu-readelf -r test.o

Relocation section '.rela.text' at offset 0x1c8 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000006  000800000017 R_RISCV_PCREL_HI2 0000000000000000  a + 0
```

从上述重定位表信息可知，重定位需要附加的字节数为 0，因此，计算公式中的 *A* 为 0。

把上述值代入计算公式，offset = *S* + *A*–*P* = 0x2008 + 0–0x602 = 0x1A06。

接下来，计算 AUIPC 指令和 ADDI 指令中的操作数 hi20 和 lo12。读者可以根据 3.4 节的公式计算（参考例 3-6）。

```
hi20 = (offset >> 12) + offset[11] = (0x1a06 >> 12) + 1 = 2
lo12 = offset & 0xfff = 0xa06
```

由于 lo12 是 12 位有符号数，因此转换成 64 位有符号的十进制数后变成–1530。

因此我们可以把访问变量 *a* 的指令修正成如下指令。

```
auipc a5, 0x2
addi  a5, a5, -1530
```

对比 test 可执行二进制文件的反汇编代码可知，上述推导结果是正确的。

我们接下来看调用 foo()函数是如何重定位修订的。它需要重定位的地方是偏移量为 0x20 的位置，重定位的类型为 R_RISCV_CALL，其偏移量计算公式为 offset = *S*–*P*。

通过 riscv64-linux-gnu-readelf 和 riscv64-linux-gnu-objdump 工具，我们可知 *S* 为 foo()函数的地址，即 0x5FC；*P* 为需要重定位的位置，即 0x61C。

代入计算公式，offset = *S*–*P* = 0x5FC − 0x61C = −32。

这个地址范围满足 JALR 指令的寻址范围，所以可以把调用 foo()的指令修改成如下形式。

```
 auipc ra,0
 jalr  ra, -32(ra)
```

第一条指令把当前 PC 值设置到 ra 寄存器中，然后通过 JALR 指令直接跳转到（ra–32）地址中，就完成 foo()函数的调用。

6.4.2　函数跳转优化

在 RISC-V 中我们常常使用 CALL 指令实现函数跳转。CALL 指令是一条伪指令，它主要由两条指令组成。

CALL 伪指令可分解成 AUIPC 指令和 JALR 指令，它们配对使用可以实现长跳转模式。

CALL FUN 指令等价于以下两条指令。

```
auipc ra, 0
jalr  ra, ra, 0
```

CALL 指令的跳转范围为 32 位有符号数的地址区间，即以当前 PC 值为基地址的±2 GB 区间。另外，RISC-V 指令集还支持一种短跳转模式，JAL 是短跳转指令，只能在 21 位有符号数的地址区间中跳转，这对应以当前 PC 值为基地址的±1 MB 区间。

在编译阶段，编译器无法确定 fun()函数的最终地址，因此编译器默认把 CALL 伪指令解析为 AUIPC 和 JALR 指令。在链接阶段，链接器分配和确定了 fun()函数的地址，从而可以根据偏移量确定是否可以选择使用短跳转指令实现，从而达到优化的目的。

```
jal offset
```

其中，offset 为调用 fun()函数时 PC 值与 fun()函数地址之间的偏移量。

【例 6-16】　结合一个简单的 test 程序来说明函数跳转优化，其代码如下。

```
<test.c>

int foo(void )
{
}

int main(void)
{
    foo();
}
```

首先，对 test.c 文件进行预编译。

```
$ riscv64-linux-gnu-gcc -c test.c -o test.o
```

上述命令编译成可重定位目标文件 test.o。可重定位目标文件的所有段的起始地址都是 0。接下来，使用 riscv64-linux-gnu-objdump 命令反汇编可重定位目标文件。

```
$ riscv64-linux-gnu-objdump -d -t -r test.o
test.o:     file format elf64-littleriscv

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 g    F  .text  0000000000000010 foo
0000000000000010 g    F  .text  000000000000001c main

Disassembly of section .text:

0000000000000000 <foo>:
   0: 1141              addi  sp,sp,-16
   ...
   e: 8082              ret

0000000000000010 <main>:
  10: 1141              addi  sp,sp,-16
  12: e406              sd  ra,8(sp)
```

```
  14: e022              sd  s0,0(sp)
  16: 0800              addi  s0,sp,16
  18: 00000097          auipc  ra,0x0
          18: R_RISCV_CALL  foo
          18: R_RISCV_RELAX  *ABS*
  1c: 000080e7          jalr  ra # 18 <main+0x8>
  ...
  2a: 8082                  ret
```

从 main()函数的反汇编结果可知，编译器在编译阶段无法确定 main()函数的地址与 foo()函数的地址的偏移量，所以编译器默认使用长跳转指令组合，AUIPC 指令使用 0x0 作为临时操作数，并且把这个问题转交给链接器来解决。

R_RISCV_CALL 和 R_RISCV_RELAX 是 ELF 规范里定义的重定位类型。R_RISCV_CALL 与 R_RISCV_RELAX 告诉链接器，在链接阶段可以进行链接器松弛优化。

接下来，编译成可执行二进制文件，然后继续观察反汇编结果。

```
$ riscv64-linux-gnu-gcc test.c -o test
$ riscv64-linux-gnu-objdump -d -r test

00000000000005ea <foo>:
 5ea: 1141              addi  sp,sp,-16
 ...
 5f8: 8082              ret

00000000000005fa <main>:
 5fa: 1141              addi  sp,sp,-16
 5fc: e406              sd  ra,8(sp)
 5fe: e022              sd  s0,0(sp)
 600: 0800              addi  s0,sp,16
 602: fe9ff0ef          jal  ra,5ea <foo>
 ...
 610:  8082             ret
```

从上述反汇编结果可知，链接器在链接可执行二进制文件时进行了链接器松弛优化，因为它知道 foo()函数的地址与 main()函数的地址的偏移量，这个偏移量正好在 JAL 指令的寻址范围内，所以这里使用短跳转指令 JAL 即可，从而省略了一条指令。

RISC-V 版本的 GCC 有与松弛链接相关的编译选项。

- -mrelax：使能链接器松弛优化，GCC 默认使能该选项。
- -mno-relax：关闭链接器松弛优化。

我们使用-mno-relax 编译选项编译 test 程序并观察反汇编结果。

```
$ riscv64-linux-gnu-gcc test.c -o test -mno-relax
$ riscv64-linux-gnu-objdump -d -r test
00000000000005fc <foo>:
 5fc: 1141              addi  sp,sp,-16
 ...
 60a: 8082              ret

000000000000060c <main>:
 60c: 1141              addi  sp,sp,-16
 60e: e406              sd  ra,8(sp)
 610: e022              sd  s0,0(sp)
 612: 0800              addi  s0,sp,16
 614: 00000097          auipc  ra,0x0
 618: fe8080e7          jalr  -24(ra) # 5fc <foo>
 ...
 626: 8082              ret
```

从上述日志可知，链接器并没有做优化，依然使用默认的两条指令来实现函数跳转，比链接器松弛优化的情况下要多一条指令，效率要低一些。

6.4.3　符号地址访问优化

RISC-V 在访问 32 位 PC 相对地址或者符号（如全局变量）时常常使用如下两条指令的组合。

```
auipc a0, %pcrel_hi(sym)
addi a0, a0, %pcrel_lo (sym)
```

链接器松弛优化的目的是减少访问内存的指令。

【例 6-17】　在 BenOS 中添加 data()函数，分别对全局变量 *a* 和 *b* 进行访问。

```
<benos/src/kernel.c >

long a = 5;
long b = 10;

long data(void) {
    return a | b;
}

void kernel_main(void)
{

    ...
    data();

    while(1)
        ;

}
```

编译 BenOS。可重定位目标文件的所有段的起始地址都是 0。接下来，使用 riscv64-linux-gnu-objdump 命令反汇编可重定位目标文件。

```
$ riscv64-linux-gnu-objdump -d -r build_src/kernel_c.o

build_src/kernel_c.o:     file format elf64-littleriscv


Disassembly of section .text:

0000000000000000 <data>:
   0: 00000797          auipc  a5,0x0
          0: R_RISCV_PCREL_HI20  a
          0: R_RISCV_RELAX  *ABS*
   4: 00078793          mv  a5,a5
          4: R_RISCV_PCREL_LO12_I  .L0
          4: R_RISCV_RELAX  *ABS*
   8: 6398              ld  a4,0(a5)
   a: 00000797          auipc  a5,0x0
          a: R_RISCV_PCREL_HI20  b
          a: R_RISCV_RELAX  *ABS*
   e: 00078793          mv  a5,a5
          e: R_RISCV_PCREL_LO12_I  .L0
          e: R_RISCV_RELAX  *ABS*
  12: 639c              ld  a5,0(a5)
  14: 8fd9              or  a5,a5,a4
  16: 853e              mv  a0,a5
  18: 8082              ret
```

从上述反汇编结果可知，我们使用两条 AUIPC 指令访问两个全局变量。这两条 AUIPC 指令都用于计算全局变量地址，但是我们知道这些全局变量在内存中是连续的，我们可以通过优化手段来减少对 AUIPC 指令的使用。

R_RISCV_PCREL_HI20 与 R_RISCV_PCREL_LO12_I 都是 ELF 规范里定义的重定位类型，

它们主要用于 32 位 PC 相对寻址。R_RISCV_PCREL_HI20 表示符号地址的高 20 位，R_RISCV_PCREL_LO12_I 表示符号地址的低 12 位。R_RISCV_RELAX 用来告诉链接器这里需要做链接器松弛优化。

如前所述，RISC-V 指令集中没有单独的一条指令能完成符号地址的加载操作，通常需要两条指令（即 AUIPC 和 ADDI 指令）的组合。不过，RISC-V 指令集提供另外一种巧妙的优化手段，即使用全局指针（Global Pointer，GP）寄存器，它指向数据段（.sdata 段）中的一个地址。如果全局变量存储在以这个 GP 寄存器的值为基地址的±2 KB 范围内，那么链接器就能够优化对全局变量的访问，AUIPC 与 ADDI 指令的组合可以替换成对以 GP 寄存器的值为基地址的相对寻址，只需要一条 LW 或者 ADDI 指令即可达到优化目的。

```
auipc rd, symbol[31:12]
addi rd, rd, symbol[11:0]

优化=>
addi rd, gp, offset
```

之所以设定±2 KB，这正对应 RISC-V 中的 LW 和 ADDI 等指令支持最大的寻址范围，即 12 位有符号数对应的地址范围。

下面我们对 benos.elf 程序进行反汇编。

```
$ riscv64-linux-gnu-objdump -d -r benos.elf
...
0000000080200d74 <data>:
    80200d74: 00002797          auipc    a5,0x2
    80200d78: 29c78793          addi a5,a5,668 # 80203010 <a>
    80200d7c: 6398              ld   a4,0(a5)
    80200d7e: 00002797          auipc    a5,0x2
    80200d82: 29a78793          addi a5,a5,666 # 80203018 <b>
    80200d86: 639c              ld   a5,0(a5)
    80200d88: 8fd9              or   a5,a5,a4
    80200d8a: 853e              mv   a0,a5
    80200d8c: 8082              ret
```

从反汇编结果可知，链接器并没有对 AUIPC 和 ADDI 指令进行优化。要在 BenOS 里使能全局变量访问优化，通过如下步骤初始化 GP 寄存器。

（1）修改 linker.ld，新增.sdata 段和__global_pointer$。

```
<benos/src/linker.ld>

/* 新增 sdata 段和设置__global_pointer$ */
.sdata : {
      __global_pointer$ = . + 0x800;
      *(.sdata)
      *(.sbss*)
  }
```

（2）在启动汇编代码中初始化 GP 寄存器。

```
<benos/src/boot.S>

_start:

      ...

.option push
.option norelax
  la gp, __global_pointer$
.option pop

    ...
```

（3）重新编译 BenOS。

（4）重新对 benos.elf 程序进行反汇编。

```
$ riscv64-linux-gnu-objdump -d -r benos.elf
...
    0000000080200d7c <data>:
    80200d7c:  81018793          addi a5,gp,-2032 # 80203010 <a>
    80200d80:  6398              ld   a4,0(a5)
    80200d82:  81818793          addi a5,gp,-2024 # 80203018 <b>
    80200d86:  639c              ld   a5,0(a5)
    80200d88:  8fd9              or   a5,a5,a4
    80200d8a:  853e              mv   a0,a5
    80200d8c:  8082              ret
```

从上述反汇编结果可知，对全局变量 *a* 和 *b* 的访问已经优化成一条 ADDI 指令。链接器在链接阶段以 GP 寄存器的值为基地址进行寻址，把 AUIPC 和 ADDI 指令的组合给优化成一条 ADDI 指令。

查看 benos.map 文件，全局变量 *a* 和 *b* 都存储在.sdata 段中。

```
<benos/benos.map>
...
.sdata          0x0000000080203000    0x20
                0x0000000080203800          __global_pointer$ = (. + 0x800)
 *(.sdata)
 .sdata         0x0000000080203000    0x8 build_src/printk_c.o
 .sdata         0x0000000080203008    0x4 build_src/uart_c.o
.sdata          0x0000000080203010    0x10 build_src/kernel_c.o
                0x0000000080203010          a
                0x0000000080203018          b
...
```

6.5 实验

6.5.1 实验 6-1：分析链接脚本

1. 实验目的

熟悉 GNU 链接脚本的语法。

2. 实验要求

分析图 6.8 所示的链接脚本中每一条语句的含义。

```
s/linker.ld+
1  SECTIONS
2  {
3          . = 0x80200000,
4
5          .text.boot : { *(.text.boot) }
6          .text : { *(.text) }
7          .rodata : { *(.rodata) }
8          .data : { *(.data) }
9          . = ALIGN(0x8);
10         bss_begin = .;
11         .bss : { *(.bss*) }
12         bss_end = .;
13 }
```

图 6.8　链接脚本

6.5.2 实验 6-2：输出每个段的内存布局

1. 实验目的

熟悉链接脚本里符号的使用。

2. 实验要求

（1）在 C 语言中输出 BenOS 镜像文件的内存布局，如图 6.9 所示，即每个段的起始地址和结束地址，以及段的大小。与 benos.map 文件进行对比，确认输出的内存布局是否正确。

```
BenOS image layout:
  .text.boot: 0x80200000 - 0x80200032 (    50 B)
       .text: 0x80200038 - 0x80201188 (  4432 B)
     .rodata: 0x80201188 - 0x80201478 (   752 B)
       .data: 0x80201478 - 0x80203000 (  7048 B)
        .bss: 0x80203010 - 0x80223420 (132112 B)
```

图 6.9　内存布局

（2）修改链接脚本，把.data 段的 VMA 修改成 0x8020 9000，然后输出内存布局看是否有变化。

（3）编写 C 语言函数来把.bss 段的内容清零。

6.5.3 实验 6-3：加载地址不等于运行地址

1. 实验目的

熟悉链接脚本的运行地址和加载地址。

2. 实验要求

假设代码段存储在 ROM 中，ROM 的起始地址为 0x8030 0000，而运行地址在 RAM 里面，起始地址为 0x8020 0000，即 LMA=0x8030 0000，VMA=0x8020 0000。其他段（如.text.boot、.data、.rodata 以及.bss 段）的加载地址和运行地址都在 RAM 中。请修改 BenOS 的链接脚本以及汇编源代码，让 BenOS 可以正确运行。

6.5.4 实验 6-4：设置链接地址

1. 实验目的

熟悉链接脚本的运行地址和链接地址。

2. 实验要求

修改 BenOS 的链接脚本，让其链接地址为 0xFFFF 0000 0000 0000。查看 benos.map 文件，指出运行地址和链接地址的区别。

6.5.5 实验 6-5：链接器松弛优化 1

1. 实验目的

熟悉 RISC-V 中的链接器松弛优化技术。

2. 实验要求

在 BenOS 中，构造一个无法使用函数跳转优化（见 6.4.2 节）的场景。

6.5.6 实验 6-6：链接器松弛优化 2

1. 实验目的

熟悉 RISC-V 中的链接器松弛优化技术。

2. 实验要求

在 BenOS 中，使能和测试符号地址访问优化（见 6.4.3 节）。

6.5.7 实验 6-7：分析 Linux 5.15 内核的链接脚本

1. 实验目的

熟悉链接脚本的语法和使用。

2. 实验要求

Linux 5.15 的链接脚本是 arch/riscv/kernel/vmlinux.lds.S，请详细分析该链接脚本，写一份分析报告。

第 7 章　内嵌汇编代码

本章思考题

1. 在内嵌汇编代码中，关键字“asm”“volatile”“inline”以及“goto”分别代表什么意思？
2. 在内嵌汇编代码的输出部分里，“=”和“+”分别代表什么意思？
3. 在内嵌汇编代码中，如何表示输出部分和输入部分的参数？
4. 在内嵌汇编代码与 C 语言宏结合时，“#”与“##”分别代表什么意思？

本章主要介绍 RISC-V 体系结构的 C 语言内嵌汇编代码，本章使用的编译器为 GCC。

7.1 内嵌汇编代码基本用法

内嵌汇编代码指的是在 C 语言中嵌入汇编代码。其作用是对于特别重要和时间敏感的代码进行优化，同时在 C 语言中访问某些特殊指令（如内存屏障指令）来实现特殊功能。

内嵌汇编代码主要有两种形式。

- 基础内嵌汇编代码（basic asm）：不带任何参数。
- 扩展内嵌汇编代码（extended asm）：可以带输入/输出参数。

7.1.1 基础内嵌汇编代码

基础内嵌汇编代码是不带任何参数的，其格式如下。

```
asm（“汇编指令”）
```

其中，asm 关键字是一个 GNU 扩展。汇编指令是汇编代码块，它有如下几个特点。

- GCC 把汇编代码块当成一个字符串。
- GCC 不会解析和分析汇编代码块。
- 当汇编代码块包含多条汇编指令时需要使用“\n”来换行。

基础内嵌汇编代码最常见的用法是调用一条汇编指令。

【例 7-1】 SFENCE.VMA 指令常用于刷新 TLB 以及同步页表读写操作。

```
__asm__ __volatile("sfence.vma");
```

7.1.2 扩展内嵌汇编代码

扩展内嵌汇编代码是常用的形式，它可以使用 C 语言中的变量作为参数，其格式如下。

```
asm 修饰词(
            指令部分
            : 输出部分
```

```
        : 输入部分
        : 损坏部分)
```

内嵌汇编代码在处理变量和寄存器的问题上提供了一个模板与一些约束条件。

常用的修饰词如下。

- volatile：用于关闭 GCC 优化。
- inline：用于内联，GCC 会把汇编代码编译成尽可能短的代码。
- goto：用于在内嵌汇编代码里跳转到 C 语言的标签处。

在指令部分中，数字前加上%（如%0、%1 等）表示需要使用寄存器的样板操作数。指令部分用到了几个不同的操作数，就说明有几个变量需要和寄存器结合。

指令部分后面的输出部分用于描述在指令部分中可以修改的 C 语言变量以及约束条件。注意以下两点。

- 输出约束（constraint）通常以“=”或者“+”开头，然后是一个字母（表示对操作数类型的说明），接着是关于变量结合的约束。“=”表示被修饰的操作数只具有可写属性，“+”表示被修饰的操作数具有可读、可写属性。
- 输出部分可以是空的。

输入部分用来描述在指令部分只能读取的 C 语言变量以及约束条件。输入部分描述的参数只具有只读属性，不要试图修改输入部分的参数内容，因为 GCC 假定输入部分的参数在内嵌汇编之前和之后都是一致的。注意以下两点。

- 在输入部分中不能使用“=”或者“+”约束条件，否则编译器会报错，如下面的错误日志。

```
atomic_add.c: In function'my_atomic_add':
atomic_add.c:16:8: error: input operand constraint contains'+'
   16 |         );
      |         ^
```

- 输入部分可以是空的。

损坏部分一般以“memory”结束。注意以下 3 点。

- “memory”告诉 GCC，如果内嵌汇编代码改变了内存中的值，那么让编译器做如下优化：在执行完汇编代码之后重新加载该值，目的是防止编译乱序。
- “cc”表示内嵌代码修改了状态寄存器的相关标志位。
- 当输入部分和输出部分显式地使用了通用寄存器时，应该在损坏部分明确告诉编译器，这些通用寄存器已经被内嵌汇编代码使用了，编译器应该避免使用这些寄存器，以免发生冲突。

对于指令部分，在内嵌汇编代码中使用%0 来表示输出部分和输入部分的第一个参数，使用%1 表示第二个参数，以此类推。

【例 7-2】 图 7.1 展示了一段内嵌汇编代码。

第 5～14 行是内嵌汇编代码。

第 5 行，volatile 用来关闭 GCC 优化。

第 6～10 行是指令部分，这里包含 5 条汇编语句，每一条汇编语句都必须使用引号括起来，并且使用“\n”来换行，因为 GCC 会把指令部分当成一个字符串，并不

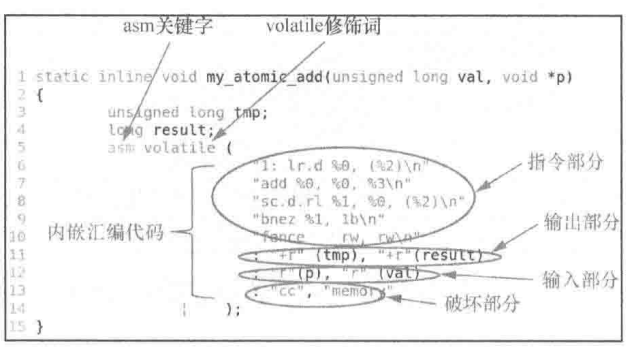


图 7.1 内嵌汇编代码

会解析汇编语句。

第 11 行是输出部分，这里有两个参数 tmp 和 result。

第 12 行是输入部分，一共有两个参数，分别是 p 和 val。

第 13 行是破坏部分。

【例 7-3】　下面展示了__raw_writel()函数的实现。

```
static inline void __raw_writel(u32 val, volatile void __iomem *addr)
{
    asm volatile("sw %0, 0(%1)" : : "r" (val), "r" (addr));
}
```

先看输入部分，%0 操作数对应"r" (val)，即 val 参数，r 表示使用一个通用寄存器，%1 操作数对应"r" (addr)，即 addr 参数。在上述例子中，输出部分和损坏部分是空的。

在输出部分和输入部分使用%来表示参数的序号，如%0 表示第 1 个参数，%1 表示第 2 个参数。

7.1.3　内嵌汇编代码修饰符

内嵌汇编代码常用的修饰符如表 7.1 所示。

表 7.1　　内嵌汇编代码常用的修饰符

修饰符	说明
=	被修饰的操作数具有只写属性
+	被修饰的操作数具有可读、可写属性
&	用于输出限定符。这个操作数在输入参数的指令执行完成之后才能写入。这个操作数不在指令所读的寄存器里，也不作为任何内存地址的一部分

内嵌汇编代码常用的操作数约束符如表 7.2 所示。

表 7.2　　内嵌汇编代码常用的操作数约束符

操作数约束符	说明
p	内存地址
m	内存变量
r	通用寄存器
o	内存地址，使用基地址寻址
i	立即数
V	内存变量，不允许偏移的内存操作数
n	立即数

RISC-V 架构中特有的操作数约束符如表 7.3 所示。

表 7.3　　RISC-V 架构中特有的操作数约束符[①]

操作数约束符	说明
f	表示浮点数寄存器
I	表示 12 位有符号的立即数
J	表示值为 0 的整数
A	表示存储到通用寄存器中的一个地址
K	表示 5 位无符号的立即数，用于 CSR 访问指令

在上述约束符中，“A”约束符常常使用在原子操作中。

① 详见 GCC 官方文档 Using the GNU Compiler Collection, v 9.3.0 的 6.47.3 节。

【例 7-4】 下面是一个原子加法计算函数。

```
void my_atomic_add(unsigned long val, void *p)
```

my_atomic_add()函数是把 val 的值原子地加到指针变量 p 指向的变量中。下面的内嵌汇编代码没有使用约束符“A”。

```
1    static inline void my_atomic_add(unsigned long val, void *p)
2    {
3        unsigned long tmp;
4        long result;
5        asm volatile (
6                "1: lr.d %0, (%2)\n"
7                "add %0, %0, %3\n"
8                "sc.d.rl %1, %0, (%2)\n"
9                "bnez %1, 1b\n"
10               "fence    rw, rw\n"
11               : "+r" (tmp), "+r"(result)
12               : "r"(p), "r" (val)
13               : "memory"
14                );
15   }
```

在第 6 行中，加载指针变量 p 的值到 tmp 变量中，这里使用“()”表示访问内存地址的内容。在破坏部分中，需要使用 memory 告诉编译器，上述内嵌汇编代码改变了内存中的值，在执行完汇编代码之后重新加载该值，否则会出错。

另外，我们可以使用约束符“A”来改写上面的汇编代码。

```
1    static inline void my_atomic_add(unsigned long val, void *p)
2    {
3        unsigned long tmp;
4        int result;
5        asm volatile (
6                "1: lr.d %0, %2\n"
7                "add %0, %0, %3\n"
8                "sc.d.rl %1, %0, %2\n"
9                "bnez %1, 1b\n"
10               : "+r" (tmp), "+r"(result), "+A"(*(unsigned long *)p)
11               : "r" (val)
12               : "memory"
13                );
14   }
```

在第 10 行中，输出部分的第 3 个参数使用了约束符“A”，并且参数变成了*(unsigned long *)p，第 6 行指令中后半部分也变成了 lr.d %0, %2。这种用法在 Linux 5.15 内核的原子操作函数里很常见，见 arch/riscv/include/atomic.h 文件。

7.1.4 使用汇编符号名字

在输出部分和输入部分，使用“%”来表示参数的序号，如%0 表示第 1 个参数，%1 表示第 2 个参数。为了增强代码可读性，我们还可以使用汇编符号名字来替代以“%”表示的操作数。

【例 7-5】 分析 add()的功能。

```
1    int add(int i, int j)
2    {
3        int res = 0;
4
5        asm volatile (
6        "add %[result], %[input_i], %[input_j]"
```

```
7         : [result] "=r" (res)
8         : [input_i] "r" (i), [input_j] "r" (j)
9         );
10
11        return res;
12    }
```

上述是一段很简单的 GCC 内嵌汇编代码，主要功能是把参数 *i* 的值和参数 *j* 的值相加，并返回结果。

先看输出部分，其中只定义了一个操作数。[result]定义一个汇编符号操作数，符号名字为 result，它对应"=r" (res)，使用函数中定义的 res 参数，在汇编代码中对应%[result]。

再看输入部分，其中定义了两个操作数。同样使用汇编符号操作数的方式来定义。第一个汇编符号操作数是 input_*i*，对应的是函数形参 *i*；第二个汇编符号操作数是 input_*j*，对应的是函数形参 *j*。

7.1.5　内嵌汇编代码与宏结合

内嵌汇编代码与 C 语言宏可以结合起来使用，让代码变得高效和简洁。我们可以巧妙地使用 C 语言宏中的“#”以及“##”符号。

若在宏的参数前面使用“#”，预处理器会把这个参数转换为一个字符串。

“##”用于连接参数和另一个标识符，形成新的标识符。

【例 7-6】 图 7.2 所示的是 ATOMIC_OP 宏，它在 Linux 5.15 内核里实现，代码路径为 arch/riscv/include/asm/atomic.h。

第 74～78 行通过调用 ATOMIC_OP 宏实现了多个函数，如 atomic_add()函数、atomic_or()、atomic_xor()。

在第 62 行中，使用“##”，把 atomic_与宏的参数 op 拼接在一起，构成函数名。

在第 65 行中，使用“#”，把参数 asm_op 转换成一个字符串。例如，假设 asm_op 参数为 add，那么第 65 行就变成"amoadd.w zero, %1, %0\n"。

另外，我们还可以通过一个宏实现多个类似的函数，这也是 C 语言中常用的技巧。

以第 74 行为例，该宏展开后变成如下代码。

```
static __always_inline
void atomic_add(int i, atomic_t *v)
{
    __asm__ __volatile__ (
        "amoadd.w zero, %1, %0"
        : "+A" (v->counter)
        : "r" (i)
        : "memory");
}
```

```
60 #define ATOMIC_OP(op, asm_op, I, asm_type, c_type, prefix)          \
61 static __always_inline                                               \
62 void atomic##prefix## ##op(c_type I, atomic##prefix##_t *v)          \
63 {                                                                    \
64         __asm__ __volatile__ (                                       \
65                 "        amo" #asm_op "." #asm_type " zero, %1, %0"  \
66                 : "+A" (v->counter)                                  \
67                 : "r" (I)                                            \
68                 : "memory");                                         \
69 }                                                                    \
70
71 #define ATOMIC_OPS(op, asm_op, I)                                    \
72         ATOMIC_OP (op, asm_op, I, w,  int,   )
73
74 ATOMIC_OPS(add, add,  i)
75 ATOMIC_OPS(sub, add, -i)
76 ATOMIC_OPS(and, and,  i)
77 ATOMIC_OPS( or,  or,  i)
78 ATOMIC_OPS(xor, xor,  i)
```

图 7.2　ATOMIC_OP 宏

7.1.6　使用 goto 修饰词

内嵌汇编代码还可以从指令部分跳转到 C 语言的标签处，这需要使用 goto 修饰词。goto 模板的格式如下。

```
asm goto (
                指令部分
                : /*输出部分是空的*/
                : 输入部分
                : 破坏部分
                : GotoLabels)
```

goto 模板与常见的内嵌汇编代码模板有如下不一样的地方。

- 输出部分必须是空的。
- 新增一个 GotoLabels，里面列出了 C 语言的标签，即允许跳转的标签。

【例 7-7】 下面的代码使用了 goto 模板。

```
1    static int test_asm_goto(int a)
2    {
3        asm goto (
4                "addi %0, %0, -1\n"
5                "beqz %0, %l[label]\n"
6                :
7                : "r" (a)
8                : "memory"
9                : label);
10
11       return 0;
12
13   label:
14       printf("%s: a = %d\n", __func__, a);
15       return 1;
16   }
```

这个例子比较简单，判断参数 *a* 是否为 1。如果为 1，跳转到 label 处；否则，就直接返回 0。

7.1.7 小结

下面对内嵌汇编代码做一下总结。

- GDB 不能单步内嵌汇编代码，所以建议使用纯汇编的方式验证过之后，再移植到内嵌汇编代码中。
- 认真仔细检查内嵌汇编代码的参数，这里很容易搞错。
- 输出部分和输入部分的修饰符不能用错，否则程序会出错。

【例 7-8】 下面的代码实现简单的 memcpy()。

```
<memcpy_test.c>

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void my_memcpy_asm_test(unsigned long src, unsigned long dst,
        unsigned long size)
{
    unsigned long tmp = 0;
    unsigned long end = src + size;

    asm volatile (
            "1: ld %1, (%2)\n"
            "sd %1, (%0)\n"
            "addi %0, %0, 8\n"
            "addi %2, %2, 8\n"
            "blt %2, %3, 1b"
            : "+r" (dst), "+r" (tmp), "+r" (src)
            : "r" (end)
            : "memory");
}

#define SIZE 8*100

int main()
{
    int i;

    char *src = malloc(SIZE);
```

```
    char *dst = malloc(SIZE);

    if (!src || !dst)
        return 0;

    printf("0x%lx 0x%lx\n", (unsigned long)src, (unsigned long)dst);

    for (i = 0; i < SIZE; i+=8)
        *(unsigned long *)(src + i) = 0x55;

    my_memcpy_asm_test((unsigned long)src, (unsigned long)dst, SIZE);

    for (i = 0; i < SIZE; i+=8) {
        if (*(unsigned long *)(dst + i) != 0x55) {
            printf("data error %lx, i %d\n", (unsigned long)(dst + i), i);
            goto free;
        }
    }

    printf("test done\n");

free:
    free(src);
    free(dst);
    return 0;
}
```

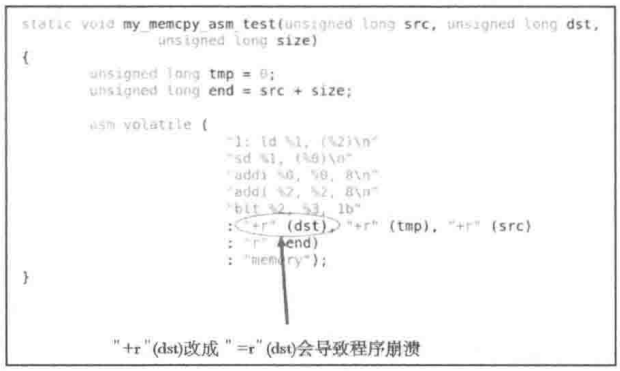


图 7.3　出错的程序

如图 7.3 所示，若输出部分的 dst 参数从"+r"(dst)改成"=r"(dst)，就会导致程序崩溃。原因是参数 dst 在 SD/ADDI 指令中，它既要读取，又要写入。

下面是程序出错的日志。

```
benshushu:riscv# ./memcpy_test
0x2aaaaad2a0 0x2aaaaad5d0
[14495.097157] memcpy_test[639]: unhandled signal 11 code 0x1 at
0x0000002aaaace000 in memcpy_test[2aaaaaa000+1000]
[14495.109473] CPU: 1 PID: 639 Comm: memcpy_test Not tainted 5.0.0+ #79
[14495.111006] sepc: 0000002aaaaaa736 ra : 0000002aaaaaa7e4 sp : 0000003fffdbda60
[14495.112621]  gp : 0000002aaaaac800 tp : 000000155557d310 t0 : 0000003fffdbd568
[14495.113105]  t1 : 0000000000000001 t2 : 0000000000000010 s0 : 0000003fffdbdaa0
[14495.113603]  s1 : 0000000000000000 a0 : 0000002aaaaad2a0 a1 : 0000002aaaaad5d0
[14495.115639]  a2 : 0000000000000320 a3 : 0000002aaaace000 a4 : 0000000000000055
[14495.116265]  a5 : 0000002aaaacdce0 a6 : fffffffffffffff6 a7 : 0000000000000040
[14495.117037]  s2 : 0000002aaabe9f90 s3 : 0000000000000000 s4 : 0000002aaabea0e0
[14495.117576]  s5 : 00000015555722c8 s6 : 0000002aaabbc3d0 s7 : 0000002aaabea080
[14495.118206]  s8 : 0000002aaabea0e0 s9 : 0000002aaab97850 s10: 0000000000000000
[14495.119701]  s11: 0000002aaab977c0 t3 : 0000000000000000 t4 : 0000002aaaaad5d0
[14495.120595]  t5 : 0000003fffdbd590 t6 : 000000000000002a
[14495.120978] sstatus: 8000000000006020 sbadaddr: 0000002aaaace000 scause:
000000000000000f
Segmentation fault
```

7.2 案例分析

【例 7-9】　下面的代码实现两字节交换的功能，其中有两处明显的错误，请认真阅读并找出来。

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    #define SIZE 10
5
```

```
6    static void swap_data(unsigned char *src, unsigned char *dst,unsigned int size)
7    {
8        unsigned int len = 0;
9        unsigned int tmp;
10
11       asm volatile (
12           "1: lhu a5, (%[src])\n"
13           "sll a6, a5, 8\n"
14           "srl a7, a5, 8\n"
15           "or %[tmp], a6, a7\n"
16           "sh %[tmp], (%[dst])\n"
17           "addi %[src], %[src], 2\n"
18           "addi %[dst], %[dst], 2\n"
19           "addi %[len], %[len], 2\n"
20           "bltu %[len], %[size], 1b\n"
21           : [dst] "+r" (dst), [len] "+r"(len), [tmp] "+r" (tmp)
22           : [src] "r" (src), [size] "r" (size)
23           : "memory"
24       );
25   }
26
27   int main(void)
28   {
29       int i;
30       unsigned char *bufa = malloc(SIZE);
31       if (!bufa)
32           return 0;
33
34       unsigned char *bufb = malloc(SIZE);
35       if (!bufb) {
36           free(bufa);
37           return 0;
38       }
39
40       for (i = 0; i < SIZE; i++) {
41           bufa[i] = i;
42           printf("%d ", bufa[i]);
43       }
44       printf("\n");
45
46       //printf("%p \n", bufa);
47       swap_data(bufa, bufb, SIZE);
48       //printf("%p \n", bufa);
49
50       for (i = 0; i < SIZE; i++)
51           printf("%d ", bufb[i]);
52       printf("\n");
53
54       free(bufa);
55       free(bufb);
56
57       return 0;
58   }
```

在 QEMU+RISC-V+Linux 平台上编译和运行。

```
# gcc in_test.c -O2 -o in_test
# ./in_test
0  1 2 3 4 5 6 7 8 9
in_test[283]: unhandled signal 11 code 0x1 at 0x0000000000000100 in
in_test[2aaaaaa000+1000]
CPU: 2 PID: 283 Comm: in_test Not tainted 5.0.0+ #3
sepc: 0000002aaaaaa6a4 ra : 0000002aaaaaa68c sp : 0000003fff950ab0
     gp : 0000002aaaaac800 tp : 000000155557b2f0 t0 : 00000015555831a8
     t1 : 00000015555d5986 t2 : 0000002aaaaac5d0 s0 : 000000000000000a
     s1 : 0000002aaaaad2a0 a0 : 000000000000000a a1 : 0000002aaaaad2e0
     a2 : 000000000000000a a3 : 0000000000010001 a4 : 0000000000000000
     a5 : 0000000000000100 a6 : 0000000000010000 a7 : 0000000000000001
     s2 : 000000000000000a s3 : 0000002aaaaaa828 s4 : 0000002aaaaad2c0
```

```
    s5 : 0000001555570810 s6 : 0000002aaabbba60 s7 : 0000002aaabc4f60
    s8 : 0000002aaabe5f00 s9 : 0000002aaab97850 s10: 0000000000000000
    s11: 0000002aaab977c0 t3 : 0000000000059986 t4 : 0000000000000009
    t5 : 0000000000000003 t6 : 000000000000002a
    sstatus: 8000000000006020 sbadaddr: 0000000000000100 scause: 000000000000000f
Segmentation fault
```

从上述日志可知，程序出现了段错误。经过分析，在第 12～15 行中显式地使用了 a5、a6 以及 a7 这 3 个通用寄存器，因此需要在破坏部分声明这些寄存器已经被内嵌汇编代码使用了，编译器在为内嵌汇编参数安排通用寄存器的时候避免使用这 3 个通用寄存器。修改方法是将第 23 行修改成如下。

```
: "memory", "a5", "a6", "a7"
```

在 QEMU+RISC-V+Linux 平台上重新编译和运行。

```
# gcc in_test.c -O2 -o in_test
# ./in_test
0 1 2 3 4 5 6 7 8 9
1 0 3 2 5 4 7 6 9 8
munmap_chunk(): invalid pointer
Aborted
```

从上述日志可知，字节交换功能实现了，但是在释放 bufa 指针时出现错误，错误日志为“munmap_chunk(): invalid pointer”。请读者认真阅读上面示例代码并找出错误的原因。

我们可以在第 47 行的 swap_data()函数前后都添加 printf("%p \n", bufa)，用于输出 bufa 指针的地址，重新编译并运行。

```
# gcc in_test.c -O2 -o in_test
# ./in_test
0 1 2 3 4 5 6 7 8 9
0x2aaaaad2a0
0x2aaaaad2aa
1 0 3 2 5 4 7 6 9 8
munmap_chunk(): invalid pointer
Aborted
```

在 C 语言中，指针形参 src 会自动生成一个副本 src_p，然后函数里的 src 指针会自动替换成副本 src_p 并参与运算。在第 17 行中，用 ADDI 指令实现 src_p = src_p + 2。从日志可知，bufa 指向的地址在 swap_data()函数前后发生了变化，可是一级指针 bufa 作为形参传递给 swap_data()函数是不会修改形参（指针参数）的指向的，那在本案例中，为什么一级指针 bufa 的指向发生了变化呢？

这里的主要原因是当使用 GCC 的 O2 优化选项时，GCC 会打开-finline-small-functions 优化选项，它会把一些短小和简单的函数集成（inline）到它们的调用者中。本案例中，swap_data()函数在编译阶段会被集成到 main()函数中。此外，如果 GCC 发现 src 形参在函数内只参与读操作，那么 GCC 在把 swap_data()函数的集成到 main()函数的过程中不会为 src 形参单独生成一个 src_p 的副本。当执行第 17 行的 ADDI 指令时，直接修改了 src 指针的指向，从而导致 bufa 指向的地址发生了改变，释放内存时出错。

我们仔细分析 swap_data()的内嵌汇编代码，发现参数 src 指定的属性不正确。参数 src 应该具有可读、可写属性，因为第 17 行修改 src 指针的指向。在第 22 行中，src 参数放在输入部分，输入部分用来描述在指令部分只能读取的 C 语言变量以及约束条件。

综上所述，swap_data()函数正确的写法如下。

```
1    static void swap_data(unsigned char *src, unsigned char *dst,unsigned int size)
2    {
```

```
3        unsigned int len = 0;
4        unsigned int tmp;
5
6        asm volatile (
7            "1: lhu a5, (%[src])\n"
8            "sll a6, a5, 8\n"
9            "srl a7, a5, 8\n"
10           "or %[tmp], a6, a7\n"
11           "sh %[tmp], (%[dst])\n"
12           "addi %[src], %[src], 2\n"
13           "addi %[dst], %[dst], 2\n"
14           "addi %[len], %[len], 2\n"
15           "bltu %[len], %[size], 1b\n"
16           : [dst] "+r" (dst), [len] "+r"(len), [tmp] "+r" (tmp),
17             [src] "+r" (src)
18           : [size] "r" (size)
19           : "memory", "a5", "a6", "a7"
20       );
21   }
```

主要改动见第 16～17 行。在第 17 行中把参数 src 放到了输出部分，并且指定它具有可读、可写属性。

读者还可以通过反汇编对比修改前后的区别。通过如下命令来得到反汇编文件 in_test.s。

```
# gcc in_test.c -S -O2
```

下面是修改前 in_test.c 源代码的反汇编代码片段。

```
1    call  malloc@plt
2    beq  a0,zero,.L2
3    mv  s1,a0
4
5    #APP
6    # 11 "in_test.c" 1
7    1: lhu a5, (s1)
8    sll a6, a5, 8
9    srl a7, a5, 8
10   or a2, a6, a7
11   sh a2, (a4)
12   addi s1, s1, 2
13   addi a4, a4, 2
14   addi a3, a3, 2
15   bltu a3, a1, 1b
16
17   # 0 "" 2
18   #NO_APP
19
20   mv  a0,s1
21   call  free@plt
```

在第 3 行中，s1 寄存器保存 bufa 指针的地址。

第 5～18 行是内嵌汇编代码。在第 12 行中，直接增加 s1 寄存器的值。

在第 20 行和第 21 行中，使用修改后的 s1 寄存器作为地址来调用 free()，导致出现“munmap_chunk(): invalid pointer”问题。

下面是修改后 in_test.c 源代码的反汇编代码片段。

```
1    call  malloc@plt
2    beq  a0,zero,.L2
3    mv  s1,a0
4    mv  a1,s1
5
6    #APP
7    # 11 "in_test_fix.c" 1
8        1: lhu a5, (a1)
9    sll a6, a5, 8
```

```
10    srl a7, a5, 8
11    or a2, a6, a7
12    sh a2, (a4)
13    addi a1, a1, 2
14    addi a4, a4, 2
15    addi a3, a3, 2
16    bltu a3, a0, 1b
17
18    # 0 "" 2
19    #NO_APP
20
21
22    mv  a0,s1
23    call  free@plt
```

对比上述两段汇编代码可以发现，最大的区别是第 4 行，GCC 为 bufa 指针（s1 寄存器的值）分配了一个临时寄存器 a1，相当于为形参 src 分配了一个副本 src_p，然后在内嵌汇编代码（见第 8 行和第 13 行）中使用该临时寄存器，从而避免第 13 行的 ADDI 指令修改 bufa 指针的指向。

7.3 注意事项

使用内嵌汇编代码时常见的注意事项如下。

- 需要明确每个 C 语言参数的约束条件，例如，指定这个参数是应该在输出部分还是输入部分。
- 正确使用每个 C 语言参数的约束符，使用错误的读写属性会导致程序出错。
- 当输入部分和输出部分显式地使用了通用寄存器时应该在损坏部分明确告诉编译器。
- 如果内嵌汇编代码修改了内存地址的值，则需要在破坏部分使用 memory 参数。
- 如果内嵌汇编代码隐含了内存屏障语义，如获取/释放屏障，则需要在破坏部分使用 memory 参数。
- 如果内嵌汇编代码使用 LR.D 以及 SC.D.RL 等原子操作指令，建议使用“A”约束符来实现寻址。

7.4 实验

7.4.1 实验 7-1：实现简单的 memcpy()函数

1. 实验目的

熟悉内嵌汇编代码的使用。

2. 实验要求

使用内嵌汇编代码实现简单的 memcpy()函数：从 0x8020 000 地址复制 32 字节到 0x8021 000 地址处，并使用 GDB 验证数据是否复制正确。

7.4.2 实验 7-2：使用汇编符号名写内嵌汇编代码

1. 实验目的

熟悉汇编符号名的使用。

2. 实验要求

在实验 7-1 的基础上尝试使用汇编符号名编写内嵌汇编代码。

7.4.3 实验 7-3：使用内嵌汇编代码完善 memset()函数

1. 实验目的

熟悉内嵌汇编代码的使用。

2. 实验要求

使用内嵌汇编代码完成__memset_16bytes()汇编函数。

7.4.4 实验 7-4：使用内嵌汇编代码与宏的结合

1. 实验目的

熟悉使用宏写内嵌汇编代码。

2. 实验要求

实现一个宏 MY_OPS(ops, instruction)，它可以对某个内存地址实现 or、xor、and、sub 等操作。

7.4.5 实验 7-5：实现读和写系统寄存器的宏

1. 实验目的

熟悉使用宏读与写系统寄存器。

2. 实验要求

实现 read_csr(csr)宏以及 write_csr(val, csr)宏，用于读取 RSIC-V 中的系统寄存器。

7.4.6 实验 7-6：goto 模板的内嵌汇编代码

1. 实验目的

熟悉基于 goto 模板的内嵌汇编代码的使用。

2. 实验要求

使用 goto 模板来实现一个内嵌汇编函数，判断函数的参数是否为 1。如果为 1，则跳转到 label，并且输出参数的值；否则，直接返回。

```
int test_asm_goto(int a)
```

第8章 异常处理

本章思考题

1. 在 RISC-V 处理器中，异常有哪几类？
2. 同步异常和异步异常有什么区别？
3. 在 RISC-V 处理器中，异常发生后 CPU 自动做了哪些事情？软件需要做哪些事情？
4. 当返回时，异常是返回到发生异常的指令还是下一条指令？
5. 当返回时，异常如何选择处理器的执行状态？
6. 请简述 RISC-V 体系结构的异常向量表。
7. 异常发生后，软件需要保存异常上下文，异常上下文包括哪些内容？
8. 异常发生后，软件如何知道异常类型？

本章主要介绍 RISC-V 体系结构中与异常处理相关的知识。

8.1 异常处理基本概念

在 RISC-V 体系结构中，异常处理和中断处理都属于异常。

8.1.1 异常类型

本节介绍异常的类型。

1. 中断

通常系统级芯片内部会有一个中断控制器，众多的外部设备的中断引脚会连接到中断控制器，由中断控制器负责中断优先级调度，然后发送中断信号给 RISC-V 处理器。中断模型如图 8.1 所示。

在外设中发生了重要的事情之后，它需要通知处理器。中断发生的时刻和当前正在执行的指令无关，因此中断的发生时间点是异步的。对于处理器来说，不得不停止当前正在执行的程序来处理中断。中断属于异步模式的异常。

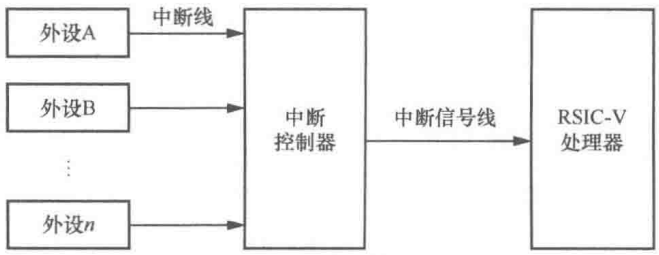


图 8.1 中断模型

2. 异常

异常主要有指令访问异常（instruction access fault）和数据访问异常（data access fault）两种。数据访问异常包括加载访问异常（load access fault）和存储访问异常（store access fault）。

当处理器打开 MMU 后，若访问内存地址时发生了错误（如缺页等），处理器内部的 MMU 捕获这些错误并且报告给处理器，从而触发缺页异常（page fault）。在 RISC-V 体系结构中常见

的缺页异常有加载缺页异常（load page fault）和存储缺页异常（store page fault）。当处理器尝试执行某条指令时发生了错误，会触发指令缺页异常。

3. 系统调用

RISC-V 体系结构提供软件触发的异常，用于系统调用。系统调用允许软件主动通过特殊指令请求更高特权模式的程序所提供的服务。例如，运行在 U 模式的应用程序可以通过系统调用来请求 S 模式的操作系统提供的服务，运行在 S 模式的操作系统可以通过系统调用来请求运行在 M 模式的 SBI 固件提供的服务。RISC-V 指令集提供 ECALL 指令来实现系统调用。

8.1.2 同步异常和异步异常

异常分成同步异常和异步异常两种。同步异常是指处理器执行某条指令而直接导致的异常，往往需要在异常处理程序里处理完该异常之后，处理器才能继续执行。例如，如果一个运行在 S 模式下的处理器内核访问了一个只在 M 模式下有访问权限的寄存器，那么会立即触发一个非法指令访问异常，操作系统的异常处理程序需要报告这个进程的错误访问或者终止（terminate）该进程。

常见的同步异常如下。

- 尝试执行非法指令（illegal instruction）。
- 使用没有对齐的 SP。
- 尝试执行一条 PC 指针没有对齐的指令。
- 软件产生的异常，如执行 ECALL 指令。
- 地址翻译或者权限等原因导致的数据异常。
- 地址翻译或者权限等原因导致的指令异常。
- 调试导致的异常，如断点异常、观察点异常、软件单步异常等。

而异步异常是指触发的原因与处理器当前正在执行的指令无关的异常，中断属于异步异常的一种。因此，指令和数据异常称为同步异常，而中断称为异步异常。

8.1.3 异常入口和返回

当异常（中断）发生时，我们把处理过程分解成 3 部分。

1. 异常发生时 CPU 做的事情

异常发生时 CPU 做的事情如下。

当一个异常发生时，在默认情况下，所有的异常（包括中断）都在 M 模式下处理。CPU 内核能检测到异常的发生，并把异常向量表的入口地址设置到 PC 寄存器中。CPU 会自动做如下一些事情。

- 保存当前 PC 值到 mepc 寄存器中。
- 把异常的类型更新到 mcause 寄存器。
- 把发生异常时的虚拟地址更新到 mtval 寄存器中。
- 保存异常发生前的中断状态，即把异常发生前的 MIE 字段保存到 mstatus 寄存器的 MPIE 字段中。
- 保存异常发生前的处理器模式（如 U 模式、S 模式等），即把异常发生前的处理器模式保存到 mstatus 寄存器的 MPP 字段中。
- 关闭本地中断，即设置 mstatus 寄存器中的 MIE 字段为 0。
- 设置处理器模式为 M 模式。
- 跳转到异常向量表，即把 mtvec 寄存器的值设置到 PC 寄存器中。

上述是 RISC-V 处理器检测到异常发生后自动做的事情。异常发生之后，还有两个寄存器与

异常相关。

- mtval：记录发生异常的虚拟地址。
- mpec：记录发生异常的指令地址。

另外，在 S 模式处理异常的流程也是类似的，前提是异常需要委派给 S 模式处理。

2. 操作系统需要做的事情

操作系统需要做的事情是从异常向量表开始的。RISC-V 支持两种异常向量表处理模式，一种是直接跳转模式，另一种是异常向量模式。下面是操作系统需要做的事情。

- 保存异常发生时的上下文，上下文包括所有通用寄存器的值和部分 M 模式下的寄存器的值。上下文需要保存到栈里。
- 查询 mcause 寄存器中的异常以及中断编号，跳转到合适的异常处理程序中。
- 异常或者中断处理完成之后，恢复保存在栈里的上下文。
- 执行 MRET 指令，返回异常现场。

3. 异常返回

当操作系统的异常处理完成后，执行一条 MRET 指令即可从异常返回。这条指令会自动完成如下工作。

- 恢复设置 MIE 字段。把 mstatus 寄存器中的 MPIE 字段设置到 mstatus 寄存器的 MIE 字段，恢复触发异常前的中断使能状态，通常这相当于使能了本地中断。
- 将处理器模式设置成之前保存到 MPP 字段的处理器模式。
- 把 mepc 寄存器保存的值设置到 PC 寄存器中，即返回异常触发的现场。

中断处理过程是关闭中断的情况下进行的，中断处理完成后什么时候把中断打开呢？

当中断发生时，CPU 会把 mstatus 寄存器中的 MIE 字段保存到 MPIE 字段中，并且把 MIE 字段清零，这相当于把本地 CPU 的中断关闭了。

当中断处理完成后，操作系统调用 MRET 指令返回中断现场，于是 mstatus 寄存器中的 MPIE 字段设置到 MIE 字段中，这就相当于把中断打开了。

异常触发与返回的流程如图 8.2 所示。

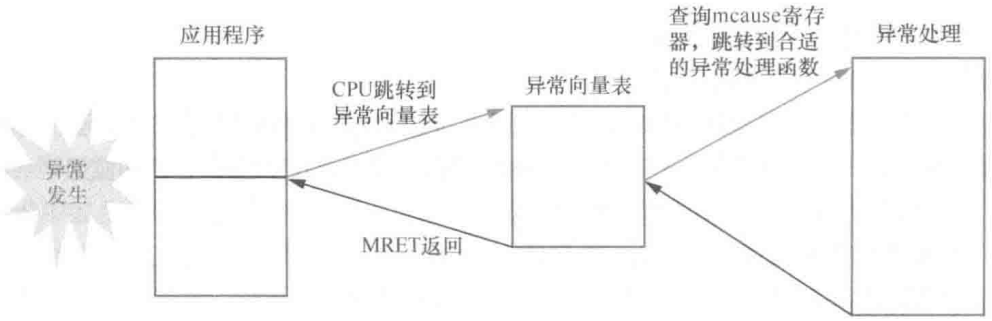


图 8.2　异常触发与返回的流程

8.1.4　异常返回地址

在 RISC-V 中，ra 寄存器用来存放子函数的返回地址，一般是用于函数调用，即可以使用 RET 指令返回到父函数里。但在异常处理中，发生异常时的 PC 值（指令地址）会自动保存到 mepc/sepc 寄存器里。在执行 mret/sret 指令返回到异常现场时，处理器会把 mepc/sepc 寄存器的值恢复到 PC 寄存器中。

既然 mepc/sepc 寄存器保存了异常返回地址，那么这个返回地址是指向发生异常时的指令还

是下一条指令呢？我们需要区分不同的情况。

- 对于异步异常（中断），它的返回地址是第一条还没执行或由于中断没有成功执行的指令。
- 对于不是系统调用的同步异常，如数据异常、访问没有映射的地址等，它返回的是触发同步异常的那条指令。例如，通过 LD 指令访问一个地址，对于这个地址，没有建立地址映射。CPU 访问这个地址时触发了一个加载访问异常，陷入内核模式。在内核模式，操作系统把这个地址映射建立起来，然后返回异常现场。此时，CPU 会继续执行这条 LD 指令。刚才因为地址没有映射而触发异常，异常处理中修复了这个映射关系，所以 LD 可以访问这个地址了。
- 系统调用返回的是系统调用指令（如 ECALL 指令）的下一条指令。在操作系统的异常处理程序中需要正确处理 mepc/sepc 寄存器的值，通常需要额外加 4 字节，这样才能返回系统调用指令的下一条指令，详见本章后面的 sbi_ecall_handle()函数。

8.1.5 异常返回的处理器模式

异常处理结束之后，当调用 MRET 指令返回时，要不要切换处理器模式呢？这里需要看 mstatus 寄存器中 MPP 字段的值。

- 如果 MPP 字段为 0，表示触发异常时 CPU 正运行在 U 模式，那么异常处理结束后会返回 U 模式。
- 如果 MPP 字段为 1，表示触发异常时 CPU 正运行在 S 模式，那么异常处理结束后会返回 S 模式。
- 如果 MPP 字段为 2，表示触发异常时 CPU 正运行在 M 模式，那么异常处理结束后会继续返回 M 模式。

8.1.6 栈的选择

在有些处理器体系结构（如 ARMv8 体系结构）中，每个处理器模式或者异常等级（exception level）都有对应的 SP 寄存器。在 ARMv8 体系结构中，在 EL0 有一个对应的 SP 寄存器 SP_EL0，同理，在 EL1 也有一个对应的 SP 寄存器 SP_EL1。当异常发生时，CPU 会跳转到目标异常等级。此时，CPU 会自动选择 SP_EL*x*。

在有些处理器体系结构（如 RISC-V 体系结构）中，所有的处理器模式只有一个 SP 寄存器。在 RISC-V 体系结构中，当处理器从一个模式跳转到另外一个模式时，SP 寄存器还指向上一个处理器模式的栈地址。此时，软件需要把上一个处理器模式的 SP 指针保存起来，可以保存到栈里或者某个特殊的寄存器，然后从这个特殊的寄存器中把当前处理器模式对应的栈地址加载到 SP 寄存器中。例如，RISC-V 体系结构提供一个 mscratch/sscratch 寄存器，用来保存 M 模式/S 模式的一些核心数据，如 SP 等。

对比上述两种设计，前者对软件比较友好，软件不需要额外处理 SP 问题，不过微体系结构设计可能会复杂一些；后者需要软件来处理 SP 问题，微体系结构的设计简洁一些。

8.2 与 M 模式相关的异常寄存器

与 M 模式相关的异常寄存器有 mstatus、mtvec、mcause、mie、mtval、mip 以及 mideleg 和 medeleg 等寄存器。

8.2.1　mstatus 寄存器

mstatus 寄存器记录了处理器内核的当前运行状态，包括是否使能了中断等信息。表 8.1 列出了与异常/中断相关的字段。

表 8.1　　mstatus 寄存器中与异常/中断相关的字段

字段	位	说明
SIE	Bit[1]	使能 S 模式下的中断
MIE	Bit[3]	使能 M 模式下的中断
SPIE	Bit[5]	临时保存的中断使能状态（S 模式下）
MPIE	Bit[7]	临时保存的中断使能状态（M 模式下）
SPP	Bit[8]	中断之前的特权模式（发生在 S 模式下的中断）
MPP	Bit[12:11]	中断之前的特权模式（发生在 M 模式下的中断）

mstatus 寄存器中的 MIE/SIE 字段是中断的总开关。而 mie 寄存器则是每一类具体中断的开关。在使能中断时，建议先使能 mie/sie 寄存器，后重置 mstatus 寄存器中的 MIE/SIE 字段。

8.2.2　mtvec 寄存器

当异常发生时，处理器必须跳转到并执行与异常处理相关的指令。与异常处理相关的指令通常存储在内存中，这个存储位置称为异常向量。在 RISC-V 体系结构中，异常向量存储在一个表中，称为异常向量表。RISC-V 体系结构在 S 模式和 M 模式下都有自己的异常向量表。

RISC-V 体系结构的异常向量表支持两种模式。

- 直接访问模式。
- 向量访问模式。

在 M 模式下，有一个异常向量表基地址寄存器——mtvec（machine trap vector base address）寄存器。它不仅用来设置异常向量表的基地址，还用来设置向量支持模式。mtvec 寄存器如图 8.3 所示。

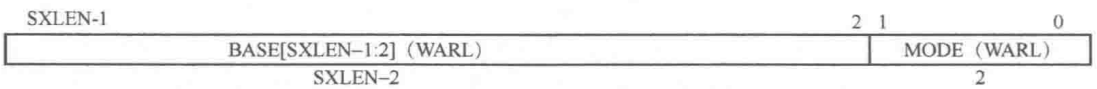


图 8.3　mtvec 寄存器

其中，BASE 字段用来设置异常向量表的基地址，MODE 字段用来设置向量模式。

- 0：表示直接访问模式。
- 1：表示向量访问模式。

1. 直接访问模式

当设置为直接访问模式时，所有陷入 M 模式的异常或者中断会自动跳转到 BASE 字段设置的基地址中。在中断处理函数中读取 mcause 寄存器来查询异常或者中断触发的原因，然后跳转到对应的异常（中断）处理函数中。

在直接访问模式下，异常向量基地址必须按 4 字节对齐。

2. 向量访问模式

当设置为向量访问模式时，中断触发后会跳转到 BASE 字段指向的异常向量表中，每个向量占 4 字节，即“BASE + 4(exception code)”，这个“exception code”是通过查询 mcause 寄存器得到的。例如，在 M 模式下，时钟中断触发后会跳转到 BASE+0x1C 地址处。

在向量访问模式下，异常向量基地址必须按 256 字节对齐。

8.2.3 mcause 寄存器

RISC-V 体系结构还提供一个查询异常原因的寄存器——mcause（machine cause）寄存器。mcause 寄存器如图 8.4 所示。

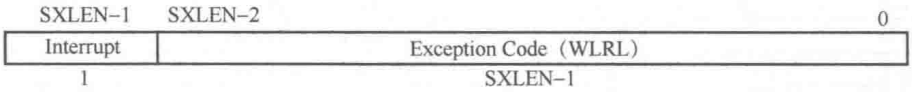


图 8.4 mcause 寄存器

其中，最高位为 Interrupt 字段，其余字段为异常编码（Exception Code，EC）字段。当 Interrupt 字段为 1 时，表示触发的异常类型为中断类型；否则，为同步异常类型。mcause 寄存器支持的异常和中断类型如表 8.2 所示。

表 8.2 mcause 寄存器支持的异常和中断类型

Interrupt 字段	EC 字段	说明
1	0	保留
1	1	S 模式下的软件中断
1	2	保留
1	3	M 模式下的软件中断
1	5	S 模式下的时钟中断
1	6	保留
1	7	M 模式下的时钟中断
1	8	保留
1	9	S 模式下的外部中断
1	10	保留
1	11	M 模式下的外部中断
1	12 与 13	保留
1	大于或等于 16 的整数	预留给芯片设计使用
0	0	指令地址没对齐
0	1	指令访问异常
0	2	非法指令
0	3	断点
0	4	加载地址没对齐
0	5	加载访问异常
0	6	存储/AMO 地址没对齐
0	7	存储/AMO 访问异常
0	8	来自 U 模式的系统调用
0	9	来自 S 模式的系统调用
0	11	保留
0	12	指令缺页异常
0	13	加载缺页异常
0	14	保留
0	15	存储/AMO 缺页异常
0	16～23 的整数	保留
0	24～31 的整数	预留给芯片设计使用
0	32～47 的整数	保留
0	48～63 的整数	预留给芯片设计使用
0	大于或等于 64 的整数	保留

8.2.4 mie 寄存器

mie（machine interrupt enable）寄存器用来单独使能和关闭 M 模式下的中断。mie 寄存器的字段如表 8.3 所示。

表 8.3　　mie 寄存器的字段

字段	位	说明
SSIE	Bit[1]	使能 S 模式下的软件中断
MSIE	Bit[3]	使能 M 模式下的软件中断
STIE	Bit[5]	使能 S 模式下的时钟中断
MTIE	Bit[7]	使能 M 模式下的时钟中断
SEIE	Bit[9]	使能 S 模式下的外部中断
MEIE	Bit[11]	使能 M 模式下的外部中断

8.2.5　mtval 寄存器[①]

当处理器陷入 M 模式时，mtval 寄存器记录发生异常的虚拟地址。

8.2.6　mip 寄存器

mip（machine interrupt pending）寄存器用来指示哪些中断处于等待响应状态。mip 寄存器的字段如表 8.4 所示。

表 8.4　　mip 寄存器的字段

字段	位	说明
SSIP	Bit[1]	S 模式下的软件中断处于等待响应状态
MSIP	Bit[3]	M 模式下的软件中断处于等待响应状态
STIP	Bit[5]	S 模式下的时钟中断处于等待响应状态
MTIP	Bit[7]	M 模式下的时钟中断处于等待响应状态
SEIP	Bit[9]	S 模式下的外部中断处于等待响应状态

8.2.7　mideleg 和 medeleg 寄存器

在默认情况下，所有的异常和中断都在 M 模式下处理。但是，RISC-V 体系结构提供了一种委托机制，即把部分异常和中断可以全盘委托给 S 模式来处理，这可以消除额外模式切换带来的性能损失，从而提高性能。

RISC-V 体系结构实现了两个委托寄存器，用来把异常以及中断委托给 S 模式来处理。运行在 M 模式的软件（如 OpenSBI）可以通过设置 mideleg 和 medeleg 寄存器相应的位，有选择地将中断和异常委托给 S 模式来处理。

mideleg 寄存器用来设置中断委托。mideleg 寄存器的字段如表 8.5 所示。

表 8.5　　mideleg 寄存器的字段

字段	位	说明
SSIP	Bit[1]	把软件中断委托给 S 模式
STIP	Bit[5]	把时钟中断委托给 S 模式
SEIP	Bit[9]	把外部中断委托给 S 模式

medeleg 寄存器用来设置异常委托。medeleg 寄存器的位如表 8.6 所示。

表 8.6　　medeleg 寄存器的位

位	说明
Bit[0]	把未对齐的指令访问异常委托给 S 模式
Bit[1]	把指令访问异常委托给 S 模式
Bit[2]	把无效指令异常委托给 S 模式
Bit[3]	把断点异常委托给 S 模式

① 在早期的 RISC-V 规范中，该寄存器称为 mbadaddr 寄存器。

续表

位	说明
Bit[4]	把未对齐加载访问异常委托给 S 模式
Bit[5]	把加载访问异常委托给 S 模式
Bit[6]	把未对齐存储/AMO 访问异常委托给 S 模式
Bit[7]	把存储/AMO 访问异常委托给 S 模式
Bit[8]	把来自用户模式的系统调用处理委托给 S 模式
Bit[9]	把来自管理员特权模式的系统调用处理委托给 S 模式
Bit[12]	把指令缺页异常委托给 S 模式
Bit[13]	把加载缺页异常委托给 S 模式
Bit[15]	把存储/AMO 缺页异常委托给 S 模式

8.2.8 中断配置

在 M 模式下，配置中断的基本步骤如下。

（1）配置 mtvec 寄存器来设置异常向量入口地址以及异常向量的访问模式。

（2）配置和初始化中断，例如，配置 PLIC 来初始化外设中断。

（3）配置 mie 寄存器来使能中断，例如，使能 M 模式以及 S 模式下的中断，包括软件中断、时钟中断以及外部中断等。

（4）配置 mstatus 寄存器来打开处理器的中断总开关。

8.3 与 S 模式相关的异常寄存器

与 S 模式相关的异常寄存器有 sstatus、sie、sip、scause 以及 stvec 等寄存器。

8.3.1 sstatus 寄存器

与 M 模式一样，在 S 模式下也有一个记录处理器内核的运行状态的寄存器，即 sstatus 寄存器。sstatus 寄存器可以看成 mstatus 寄存器的一个部分限制访问的镜像，它不包含 M 模式下的字段。对 sstatus 寄存器所做的更改会反映在 mstatus 寄存器中，对 mstatus 寄存器所做的更改会反映在 sstatus 寄存器中。

表 8.7 展示了 sstatus 寄存器中与异常/中断相关的字段。

表 8.7　　sstatus 寄存器中与异常/中断相关的字段

字段	位	说明
SIE	Bit[1]	使能 S 模式下的中断
SPIE	Bit[5]	之前的中断使能状态（S 模式下）
SPP	Bit[8]	中断之前的特权模式（发生在 S 模式下的中断）

8.3.2 sie 寄存器

sie（supervisor interrupt enable）寄存器用来单独使能和关闭 S 模式下的中断。sie 寄存器中的字段如表 8.8 所示。

表 8.8　　sie 寄存器中的字段

字段	位	说明
SSIE	Bit[1]	使能 S 模式下的软件中断
STIE	Bit[5]	使能 S 模式下的时钟中断
SEIE	Bit[9]	使能 S 模式下的外部中断

8.3.3　sip 寄存器

sip（supervisor interrupt pending）寄存器用来指示哪些寄存器处于等待响应状态。sip 寄存器中的字段如表 8.9 所示。

表 8.9　　sip 寄存器中的字段

字段	位	说明
SSIP	Bit[1]	S 模式下的软件中断处于等待响应状态
STIP	Bit[5]	S 模式下的时钟中断处于等待响应状态
SEIP	Bit[9]	S 模式下的外部中断处于等待响应状态

8.3.4　scause 寄存器

RISC-V 体系结构提供一个查询 S 模式下异常原因的寄存器——scause（supervisor cause）寄存器。scause 寄存器如图 8.5 所示。

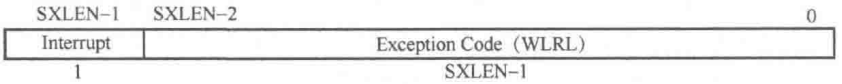


图 8.5　scause 寄存器

其中，最高位为 Interrupt 字段，其余位为 EC 字段，如表 8.10 所示。当 Interrupt 字段为 1 时，表示触发的异常类型为中断类型；否则，为同步类型异常。

表 8.10　　scause 寄存器中的字段

Interrupt 字段	EC 字段	说明
1	0	保留
1	1	S 模式下的软件中断
1	2～4 的整数	保留
1	5	S 模式下的时钟中断
1	6～8 的整数	保留
1	9	S 模式下的外部中断
1	大于或等于 10 的整数	保留
0	0	指令地址没对齐
0	1	指令访问异常
0	2	无效指令
0	3	断点
0	4	保留
0	5	加载访问异常
0	6	存储/AMO 地址没对齐
0	7	存储/AMO 访问异常
0	8	来自用户模式的系统调用
0	9～11 的整数	保留
0	12	指令缺页异常
0	13	加载缺页异常
0	14	保留
0	15	存储/AMO 缺页异常
0	大于或等于 16 的整数	保留

8.3.5　stvec 寄存器

与 M 模式一样，在 S 模式下也是需要配置异常向量基地址和向量访问模式的。在 S 模式下也支持两种异常向量的访问模式——直接访问模式和向量访问模式。

在 S 模式下，有一个用来设置异常向量基地址的管理者异常向量表基地址寄存器——stvec（supervisor trap vector base address）寄存器。stvec 寄存器有两个作用：一是用来设置异常向量的基地址，二是用来设置异常向量访问模式。stvec 寄存器如图 8.6 所示。

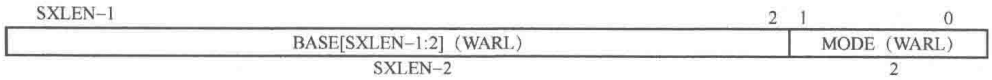


图 8.6 stvec 寄存器

其中，BASE 字段用来设置异常向量表的基地址，MODE 字段用来设置异常向量访问模式。

- 0：表示直接访问模式。
- 1：表示向量访问模式。

1）直接访问模式

在直接访问模式下，所有陷入 S 模式的异常或者中断会跳转到 BASE 字段设置的基地址中。在异常处理函数中读取 scause 寄存器来查询异常或者中断触发的原因，然后跳转到对应的异常（中断）处理函数中。

在直接访问模式下，异常向量基地址必须按 4 字节对齐。

2）向量访问模式

在向量访问模式下，异常触发后会跳转到以 BASE 字段对应的异常向量表中，每个向量占 4 字节，即“BASE + 4(exception code)”，这个“exception code”是通过查询 scause 寄存器得到的。例如，在 S 模式下的时钟中断触发后会跳转到 BASE+0x14 地址处。

在向量访问模式下，异常向量基地址必须按 256 字节对齐。

8.3.6 stval 寄存器

当处理器陷入 S 模式时，stval 寄存器记录发生异常的虚拟地址。

8.4 异常上下文

在异常发生时需要保存发生异常的现场，以免破坏异常发生前正在处理的数据和程序状态。以发生在 S 模式的异常为例，我们需要在栈空间里保存如下内容：

- x1～x31 通用寄存器的值；
- spec 寄存器的值；
- sstatus 寄存器的值；
- sbadaddr 寄存器[①]的值；
- scause 寄存器的值。

这个栈空间指的是发生异常时进程的内核模式的栈空间。在操作系统中，每个进程都有一个内核模式的栈空间。异常包括同步异常和异步异常（中断）。

为了编程方便，我们可以使用一个栈框数据结构来描述需要保存的中断现场。例如，使用结构体 pt_regs 来描述一个栈框。

```
struct pt_regs {
    /*31 个通用寄存器 + sepc + sstatus */
    unsigned long sepc;
    unsigned long ra;
    unsigned long sp;
```

① 在早期的 RISC-V 规范中，stval 寄存器称为 sbadaddr 寄存器。

```
    unsigned long gp;
    unsigned long tp;
    unsigned long t0;
    unsigned long t1;
    unsigned long t2;
    unsigned long s0;
    unsigned long s1;
    unsigned long a0;
    unsigned long a1;
    unsigned long a2;
    unsigned long a3;
    unsigned long a4;
    unsigned long a5;
    unsigned long a6;
    unsigned long a7;
    unsigned long s2;
    unsigned long s3;
    unsigned long s4;
    unsigned long s5;
    unsigned long s6;
    unsigned long s7;
    unsigned long s8;
    unsigned long s9;
    unsigned long s10;
    unsigned long s11;
    unsigned long t3;
    unsigned long t4;
    unsigned long t5;
    unsigned long t6;
    /*S 模式下的寄存器 */
    unsigned long sstatus;
    unsigned long sbadaddr;
    unsigned long scause;
};
```

pt_regs 栈框的大小为 280 字节。我们在保存异常上下文时，按照从栈顶到栈底的方向依次保存数据。以 sepc 寄存器为例，我们使用 S_SEPC 表示在栈中保存 sepc 寄存器中值的位置，使用 PT_SEPC 表示 S_SEPC 与栈顶的偏移量，如图 8.7 所示。

```
/*pt_regs 数据结构中每个字段的偏移量*/

#define PT_SIZE 280 /* (struct pt_regs)的大小 */
#define PT_SEPC 0 /* (struct pt_regs, sepc) 的偏移量*/
#define PT_RA 8 /* struct pt_regs, ra) 的偏移量*/
#define PT_SP 16 /* (struct pt_regs, sp) 的偏移量*/
#define PT_GP 24 /* (struct pt_regs, gp) 的偏移量*/
#define PT_TP 32 /* (struct pt_regs, tp) 的偏移量*/
#define PT_T0 40 /* (struct pt_regs, t0) 的偏移量*/
#define PT_T1 48 /* (struct pt_regs, t1) 的偏移量*/
#define PT_T2 56 /* (struct pt_regs, t2) 的偏移量*/
#define PT_FP 64 /* (struct pt_regs, s0) 的偏移量*/
#define PT_S1 72 /* (struct pt_regs, s1) 的偏移量*/
#define PT_A0 80 /* (struct pt_regs, a0) 的偏移量*/
#define PT_A1 88 /* (struct pt_regs, a1) 的偏移量*/
#define PT_A2 96 /* (struct pt_regs, a2) 的偏移量*/
#define PT_A3 104 /* (struct pt_regs, a3) 的偏移量*/
#define PT_A4 112 /* (struct pt_regs, a4) 的偏移量*/
#define PT_A5 120 /* (struct pt_regs, a5) 的偏移量*/
#define PT_A6 128 /* struct pt_regs, a6) 的偏移量*/
#define PT_A7 136 /* (struct pt_regs, a7) 的偏移量*/
#define PT_S2 144 /* (struct pt_regs, s2) 的偏移量*/
#define PT_S3 152 /* (struct pt_regs, s3) 的偏移量*/
#define PT_S4 160 /* (struct pt_regs, s4) 的偏移量*/
#define PT_S5 168 /* (struct pt_regs, s5) 的偏移量*/
#define PT_S6 176 /* (struct pt_regs, s6) 的偏移量*/
#define PT_S7 184 /* (struct pt_regs, s7) 的偏移量*/
#define PT_S8 192 /* (struct pt_regs, s8) 的偏移量*/
#define PT_S9 200 /* (struct pt_regs, s9) 的偏移量*/
#define PT_S10 208 /* (struct pt_regs, s10) 的偏移量*/
#define PT_S11 216 /* (struct pt_regs, s11) 的偏移量*/
#define PT_T3 224 /* (struct pt_regs, t3) 的偏移量*/
#define PT_T4 232 /* (struct pt_regs, t4) 的偏移量*/
```

```
#define PT_T5 240 /* (struct pt_regs, t5) 的偏移量*/
#define PT_T6 248 /* (struct pt_regs, t6) 的偏移量*/
#define PT_SSTATUS 256 /*(struct pt_regs, sstatus) 的偏移量*/
#define PT_SBADADDR 264 /*(struct pt_regs, sbadaddr) 的偏移量*/
#define PT_SCAUSE 272 /*(struct pt_regs, scause) 的偏移量*/
```

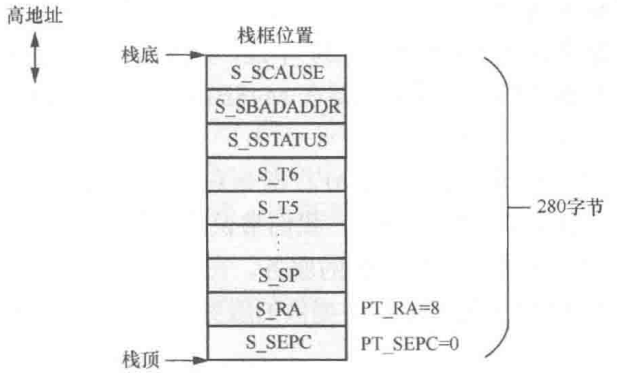


图 8.7 pt_regs 栈框

8.4.1 保存异常上下文

当异常发生时，我们需要把异常现场保存到当前进程的内核栈里，如图 8.8 所示。

- 栈框里的 S_SEPC 保存发生异常时 sepc 寄存器的内容。
- 栈框里的 S_RA 保存发生异常时 ra 寄存器的内容。
- 栈框里的 S_SP 保存发生异常时 sp 寄存器的内容。

……

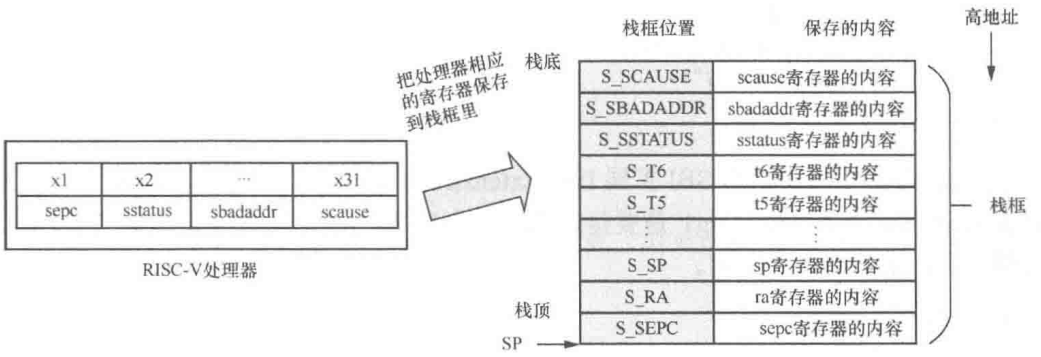


图 8.8 保存异常上下文

8.4.2 恢复异常上下文

当异常返回时，从进程的内核栈中恢复异常现场到 CPU，如图 8.9 所示。

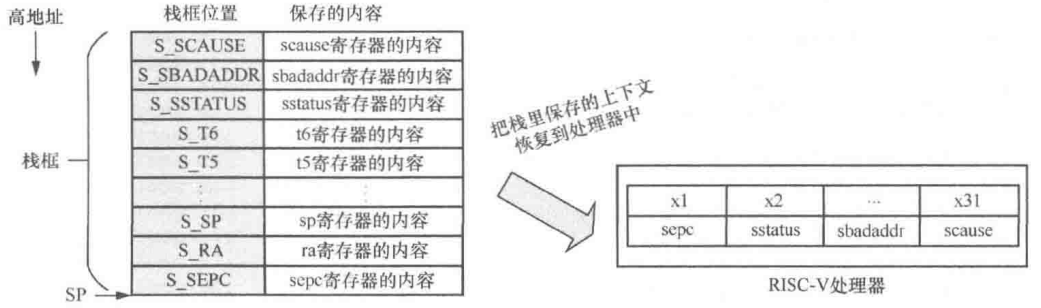


图 8.9 恢复异常上下文

8.5 案例分析 8-1：实现 SBI 系统调用

本案例中，我们需要实现一个 SBI 服务来实现串口输出功能。在第 2 章中，BenOS 是在 S 模式下通过配置串口控制器来实现串口输出的。不过，在 RISC-V 体系结构中，通常 SBI 提供了串口 I/O 的服务接口。在本案例中，我们需要在 MySBI 里实现对串口 I/O 的服务接口，让 S 模式的 BenOS 可以直接调用。

运行在 S 模式下的 BenOS 可以通过 ECALL 指令来陷入 M 模式的 MySBI 固件。ECALL 指令是 RISC-V 体系结构提供的系统调用指令。我们常说的系统调用指的是运行在 U 模式的应用程序主动调用运行在 S 模式的操作系统提供的服务。在 RISC-V 体系结构中，运行在 S 模式的操作系统主动调用运行在 M 模式的 SBI 固件提供的服务也算一种系统调用。

8.5.1 调用 ECALL 指令

我们首先要在 BenOS 里实现调用 SBI 的接口函数。SBI_CALL 宏直接调用 ECALL 指令来实现系统调用。

```
<benos/include/asm/sbi.h>

#define SBI_CALL(which, arg0, arg1, arg2) ({            \
    register unsigned long a0 asm ("a0") = (unsigned long)(arg0);  \
    register unsigned long a1 asm ("a1") = (unsigned long)(arg1);  \
    register unsigned long a2 asm ("a2") = (unsigned long)(arg2);  \
    register unsigned long a7 asm ("a7") = (unsigned long)(which);  \
    asm volatile ("ecall"                    \
              : "+r" (a0)                 \
              : "r" (a1), "r" (a2), "r" (a7)      \
              : "memory");                \
    a0;                          \
})
```

其中，which 参数用于表示 SBI 扩展 ID（Extension ID，EID）。EID 可以理解为系统调用号。arg0 是要传递的第一个参数，arg1 是要传递的第二个参数，arg2 是要传递的第三个参数。为了调用方便，我们实现如下 3 个宏。

```
<benos/include/asm/sbi.h>

/*
 * 陷入 M 模式，调用 M 模式提供的服务
*/
#define SBI_CALL_0(which) SBI_CALL(which, 0, 0, 0)
#define SBI_CALL_1(which, arg0) SBI_CALL(which, arg0, 0, 0)
#define SBI_CALL_2(which, arg0, arg1) SBI_CALL(which, arg0, arg1, 0)
```

根据 SBI 规范，定义串口的调用号。

```
<benos/include/asm/sbi.h>

#define SBI_CONSOLE_PUTCHAR 0x1
#define SBI_CONSOLE_GETCHAR 0x2

static inline void sbi_putchar(unsigned char c)
{
    SBI_CALL_1(SBI_CONSOLE_PUTCHAR, c);
}

static inline void sbi_put_string(char *str)
{
    int i;
```

```
    for (i = 0; str[i] != '\0'; i++)
        sbi_putchar((char) str[i]);
}
```

在 BenOS 里输出串口日志只需要调用上述 sbi_putchar()或者 sbi_put_string()函数。

8.5.2 实现 SBI 系统调用

如前所述，系统调用也是同步异常的一种。当处理器从 S 模式陷入 M 模式时，处理器首先会跳转到 M 模式的异常向量表。因此，对于运行在 M 模式的 SBI 固件，需要正确设置异常向量表并处理异常。在 M 模式下，若发生异常，也需要保存发生异常的现场，以免破坏异常发生前正在处理的数据和程序状态。

下面的代码用于设置 M 模式下的异常向量表。

```
<benos/sbi/sbi_trap.c>

1    extern void sbi_exception_vector(void);
2
3    void sbi_trap_init(void)
4    {
5        /*设置异常向量表的地址 */
6        write_csr(mtvec, sbi_exception_vector);
7        /*关闭所有中断 */
8        write_csr(mie, 0);
9    }
```

sbi_exception_vector()为异常向量的入口地址。在第 6 行中，把异常向量表的入口地址设置到 mtvec 寄存器中。sbi_exception_vector()函数的地址是按 4 字节对齐的，这相当于把 mtvec 寄存器中 Mode 字段设置为 0，即异常访问方式采用直接访问模式。

在第 8 行中，暂时先关闭 M 模式下的所有中断。

在 M 模式下，我们需要在栈空间里保存如下内容：

- x1～x31 通用寄存器的值；
- spec 寄存器的值。

MySBI 使用一个 sbi_trap_regs 结构体实现一个栈框，用来保存发生异常时的寄存器。

```
<benos/sbi/sbi_trap_regs.h>

struct sbi_trap_regs {
    /* sepc + 31 个通用寄存器 */
    unsigned long mepc;
    unsigned long ra;
    unsigned long sp;
    unsigned long gp;
    unsigned long tp;
    unsigned long t0;
    unsigned long t1;
    unsigned long t2;
    unsigned long s0;
    unsigned long s1;
    unsigned long a0;
    unsigned long a1;
    unsigned long a2;
    unsigned long a3;
    unsigned long a4;
    unsigned long a5;
    unsigned long a6;
    unsigned long a7;
    unsigned long s2;
    unsigned long s3;
    unsigned long s4;
```

```
    unsigned long s5;
    unsigned long s6;
    unsigned long s7;
    unsigned long s8;
    unsigned long s9;
    unsigned long s10;
    unsigned long s11;
    unsigned long t3;
    unsigned long t4;
    unsigned long t5;
    unsigned long t6;
    /*mstatus 寄存器 */
    unsigned long mstatus;
};
```

sbi_trap_regs 栈框一共需要保存 33 个寄存器的值，共 264 字节。

在保存异常上下文时，我们是按照从栈顶到栈底的方向来依次保存数据的。为了编程方便，我们使用 PT_MEPC 表示栈框中的 mepc 在栈顶的偏移量，S_MEPC 表示在栈里用来保存 mepc 寄存器的值的位置，以此类推，如图 8.10 所示。

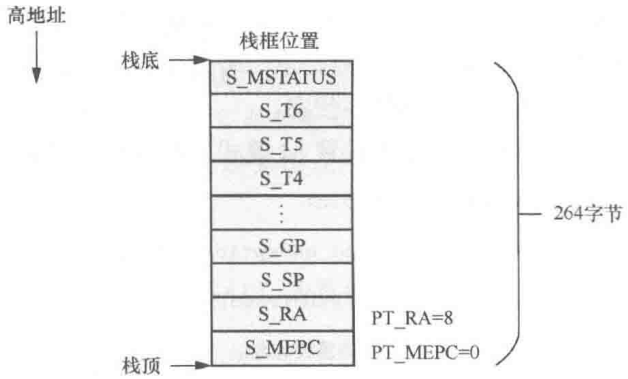


图 8.10　sbi_trap_regs 栈框

```
<benos/sbi/sbi_asm_offsets.h>

/* sbi_trap_regs 结构体中每个字段的偏移量*/

#define PT_SIZE 264 /* sizeof(struct sbi_trap_regs) */
#define PT_MEPC 0 /* offsetof(struct sbi_trap_regs, mepc) */
#define PT_RA 8 /* offsetof(struct sbi_trap_regs, ra) */
#define PT_SP 16 /* offsetof(struct sbi_trap_regs, sp) */
#define PT_GP 24 /* offsetof(struct sbi_trap_regs, gp) */
...
#define PT_T6 248 /* offsetof(struct sbi_trap_regs, t6) */
#define PT_MSTATUS 256 /* offsetof(struct sbi_trap_regs, mstatus) */
```

接下来，在 MySBI 固件里保存异常现场上下文。

```
<benos/sbi/sbi_entry.S>

/*
   sbi_exception_vector
   M 模式下的异常向量表的入口
   按 8 字节对齐
 */
.align 3
.global sbi_exception_vector
sbi_exception_vector:
    /*从 mscratch 获取 M 模式的 SP，把 S 模式的 SP 保存到 mscratch 寄存器中*/
    csrrw sp, mscratch, sp

    addi sp, sp, -(PT_SIZE)

    sd x1,  PT_RA(sp)
    sd x3,  PT_GP(sp)
    sd x5,  PT_T0(sp)
    sd x6,  PT_T1(sp)
    sd x7,  PT_T2(sp)
    sd x8,  PT_S0(sp)
    sd x9,  PT_S1(sp)
    sd x10, PT_A0(sp)
    sd x11, PT_A1(sp)
    sd x12, PT_A2(sp)
    sd x13, PT_A3(sp)
```

```
25        sd x14, PT_A4(sp)
26        sd x15, PT_A5(sp)
27        sd x16, PT_A6(sp)
28        sd x17, PT_A7(sp)
29        sd x18, PT_S2(sp)
30        sd x19, PT_S3(sp)
31        sd x20, PT_S4(sp)
32        sd x21, PT_S5(sp)
33        sd x22, PT_S6(sp)
34        sd x23, PT_S7(sp)
35        sd x24, PT_S8(sp)
36        sd x25, PT_S9(sp)
37        sd x26, PT_S10(sp)
38        sd x27, PT_S11(sp)
39        sd x28, PT_T3(sp)
40        sd x29, PT_T4(sp)
41        sd x30, PT_T5(sp)
42        sd x31, PT_T6(sp)
43
44        /*保存 mepc 寄存器的值*/
45        csrr t0, mepc
46        sd t0, PT_MEPC(sp)
47
48        /*保存 mstatus 寄存器的值*/
49        csrr t0, mstatus
50        sd t0, PT_MSTATUS(sp)
51
52        /*保存 sp*/
53        addi t0, sp, PT_SIZE
54        csrrw   t0, mscratch, t0
55        sd t0, PT_SP(sp)
56
57        /*调用 C 语言的 sbi_trap_handler */
58        mv a0, sp /* sbi_trap_regs */
59        call sbi_trap_handler
```

由于我们设置的异常访问模式是直接访问模式，因此当异常发生时处理器会直接跳转到 sbi_exception_vector()汇编函数里。

在第 10 行中，由于 mscratch 寄存器已经保存了 M 模式下 SP 的值，因此 CSRRW 指令从 mscratch 寄存器获取 M 模式下的 SP，与此同时，把 S 模式下的 SP 保存到 mscratch 寄存器中。

在 MySBI 固件入口函数_start 中，需要把 M 模式下的 SP 保存到 mscratch 寄存器中。

```
<benos/sbi/sbi_boot.S>

.globl _start
_start:
    ...
    /*
       把 M 模式下的 SP 设置到 mscratch 寄存器中，
       下次陷入 M 模式时可以获取 SP
     */
    csrw mscratch, sp
    ...
```

在第 12 行中，为 sbi_trap_regs 栈框分配栈空间，此时 SP 指向栈顶，即 S_MEPC 的位置。

在第 14～42 行中，保存 x1、x3、x5～x31 通用寄存器的值到栈中。

在第 45～46 行中，读取 mepc 寄存器的值，写入栈的 S_MEPC 位置中。

在第 49～50 行中，读取 mstatus 寄存器的值，写入栈的 S_MSTATUS 位置中。

在第 53～55 行中，有如下两个目的。

- 把 S 模式下的 SP 保存到 sbi_trap_regs 栈框的 S_SP 位置上。
- 把 M 模式下栈底的地址保存到 mscratch 寄存器中，以便下次陷入 M 模式时可以得到 SP。

在第 58 和 59 行中，以 sbi_trap_regs 栈框作为参数，调用 C 语言的 sbi_trap_handler()函数来

继续处理异常。

图 8.11 展示了如何保存异常现场。

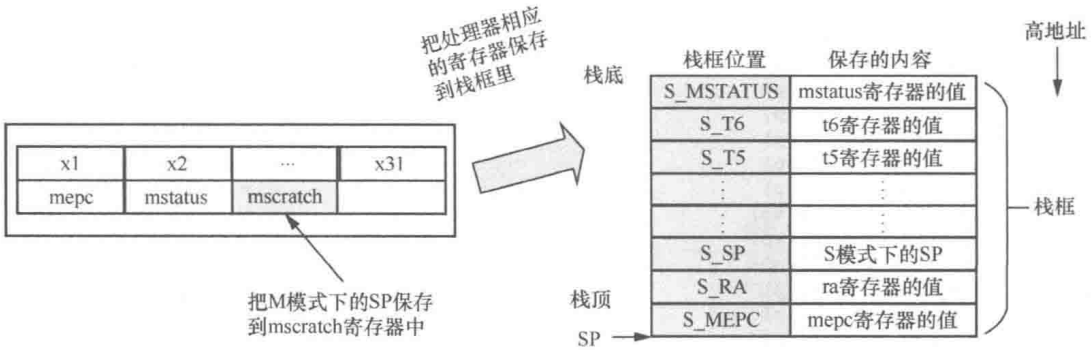


图 8.11　保存异常现场

接下来，在 sbi_trap.c 文件中实现 sbi_trap_handler()函数。

```
<benos/sbi/sbi_trap.c>

1    void sbi_trap_handler(struct sbi_trap_regs *regs)
2    {
3        unsigned long mcause = read_csr(mcause);
4        unsigned long ecall_id = regs->a7;
5
6        switch (mcause) {
7        case CAUSE_SUPERVISOR_ECALL:
8            sbi_ecall_handle(ecall_id, regs);
9            break;
10       default:
11           break;
12       }
13   }
```

sbi_trap_handler()函数的第一个参数是 sbi_trap_regs 栈框，通过这个栈框我们可以很方便地读取到发生异常时通用寄存器的值。

在第 3 行中，读取 mcause 寄存器，这个寄存器记录了发生异常的原因。

在第 4 行中，a7 寄存器记录了系统调用号。

在第 6～11 行中，通过 mcause 寄存器读取发生异常的类型，这里暂时忽略了中断的影响，第 9 章会介绍中断处理。如果判断异常类型为系统调用（CAUSE_SUPERVISOR_ECALL），那么调用 sbi_ecall_handle()函数来处理。

```
<benos/sbi/sbi_trap.c>

1    static int sbi_ecall_handle(unsigned int id, struct sbi_trap_regs *regs)
2    {
3        int ret = 0;
4
5        switch (id) {
6        case SBI_CONSOLE_PUTCHAR:
7            putchar(regs->a0);
8            ret = 0;
9            break;
10       }
11
12       /*系统调用返回的是系统调用指令（如 ECALL 指令）的下一条指令 */
13       if (!ret)
14           regs->mepc += 4;
15
16       return ret;
17   }
```

sbi_ecall_handle()函数的实现比较简单，在第 5～10 行中，如果系统调用号是 SBI_CONSOLE_PUTCHAR，直接调用串口输出函数 putchar()把字符写入串口设备中。

在第 13～14 行中，对于系统调用，异常返回后，PC 寄存器应该存放系统调用指令的下一条指令的地址，所以这里 mepc 寄存器的值需要加 4 字节。

异常处理完成之后，我们需要恢复异常上下文，并从异常现场返回。

```
<benos/sbi/sbi_entry.S>

60        /*恢复 mstatus 寄存器的值*/
61        ld t0, PT_MSTATUS(sp)
62        csrw mstatus, t0
63
64        ld t0, PT_MEPC(sp)
65        csrw mepc, t0
66
67        ld x1,  PT_RA(sp)
68        ld x3,  PT_GP(sp)
69        ld x5,  PT_T0(sp)
70        ld x6,  PT_T1(sp)
71        ld x7,  PT_T2(sp)
72        ld x8,  PT_S0(sp)
73        ld x9,  PT_S1(sp)
74        ld x10, PT_A0(sp)
75        ld x11, PT_A1(sp)
76        ld x12, PT_A2(sp)
77        ld x13, PT_A3(sp)
78        ld x14, PT_A4(sp)
79        ld x15, PT_A5(sp)
80        ld x16, PT_A6(sp)
81        ld x17, PT_A7(sp)
82        ld x18, PT_S2(sp)
83        ld x19, PT_S3(sp)
84        ld x20, PT_S4(sp)
85        ld x21, PT_S5(sp)
86        ld x22, PT_S6(sp)
87        ld x23, PT_S7(sp)
88        ld x24, PT_S8(sp)
89        ld x25, PT_S9(sp)
90        ld x26, PT_S10(sp)
91        ld x27, PT_S11(sp)
92        ld x28, PT_T3(sp)
93        ld x29, PT_T4(sp)
94        ld x30, PT_T5(sp)
95        ld x31, PT_T6(sp)
96
97        ld sp,  PT_SP(sp)
98        mret
```

在第 61 和 62 行中，把栈中 S_MSTATUS 位置的内容恢复到 mstatus 寄存器。

在第 64 和 65 行中，把栈中 S_MEPC 位置的内容恢复到 mepc 寄存器中。

在第 67～95 行中，分别从栈中恢复 x1、x3、x5～x31 寄存器的值。

在第 97 行中，恢复 sp 寄存器的值，即把 sbi_trap_regs 栈框空间释放，使 SP 指向 S 模式下的栈。

在第 98 行中，调用 MRET 指令实现异常返回，处理器会切换回 S 模式，如图 8.12 所示。

最后，重新编译和运行 BenOS。

```
rlk@master:benos$ make
rlk@master:benos$ make run
qemu-system-riscv64 -nographic -machine virt -m 128M -bios mysbi.bin  -device
loader,file=benos.bin,addr=0x80200000  -kernel benos.elf
Booting at asm
Welcome RISC-V!
```

```
printk init done
lab3-5: compare_and_return ok
lab3-5: compare_and_return ok
...
```

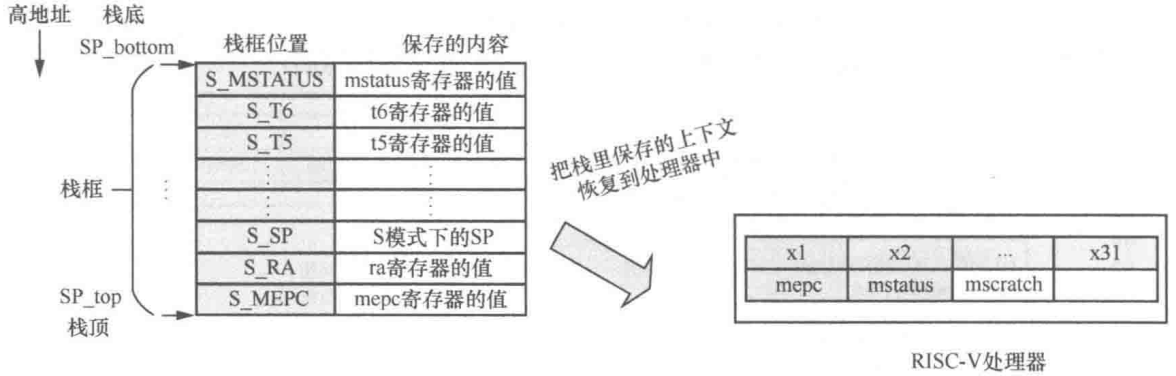


图 8.12　恢复异常上下文

运行在 S 模式下的 BenOS 可以正确输出串口信息。

8.6 案例分析 8-2：BenOS 的异常处理

我们在上一个案例中实现了 M 模式下的异常处理。在本案例中，我们需要在 S 模式下的 BenOS 里制造一个加载访问异常，然后在异常处理中输出异常类型、出错地址等日志信息。这个过程涉及 S 模式下的异常处理，包括保存异常上下文、恢复异常上下文等。

8.6.1 设置异常向量表

首先，编写一个处理异常的入口函数。

```
<benos/src/entry.S>

1     /*
2        do_exception_vector()必须按 4 字节对齐
3        否则写入 stvec 寄存器会不成功
4      */
5     .align 2
6     .global do_exception_vector
7     do_exception_vector:
8         kernel_entry
9
10        la ra, ret_from_exception
11
12        mv a0, sp /* pt_regs */
13        mv a1, s4
14        tail do_exception
15
16    ret_from_exception:
17    restore_all:
18        kernel_exit
19        sret
```

do_exception_vector()为 S 模式下的异常处理入口函数，它必须按 4 字节对齐，否则会出错。

在第 8 行中，kernel_entry 宏用来保存异常上下文。

在第 10 行中，设置返回地址为 ret_from_exception。

在第 12～14 行中，SP 指向 pt_regs 栈框。把 pt_regs 栈框作为参数传递给 do_exception()函数。s4 寄存器保存 scause 寄存器的值，该值同时作为第二个参数传递给 do_exception()函数，最

后调用 do_exception()函数来处理异常。

在第 16～19 行中，当异常处理完成后会跳转到 ret_from_exception 标签处，因为在第 10 行中设置 ret_from_exception 标签为返回地址。kernel_exit 宏恢复异常上下文。

在第 19 行中，处理器执行 SRET 指令，从异常中返回。

接下来，我们需要设置异常向量表。

```
<benos/src/trap.c>

1    void trap_init(void)
2    {
3        /*设置异常向量表的地址*/
4        write_csr(stvec, do_exception_vector);
5        printk("stvec=0x%x, 0x%x\n", read_csr(stvec), do_exception_vector);
6        /*关闭所有中断*/
7        write_csr(sie, 0);
8    }
```

在第 4 行中，通过 write_csr()函数把 do_exception_vector()汇编函数的首地址设置到 stvec 寄存器中。因为 do_exception_vector()汇编函数的首地址是按 4 字节对齐的，所以 stvec 寄存器的 MODE 字段为 0，即异常向量表的访问模式为直接访问模式。

8.6.2 保存和恢复异常上下文

在 do_exception_vector()函数中使用 kernel_entry 和 kernel_exit 宏来保存与恢复异常上下文。

kernel_entry 宏的实现如下。

```
<benos/src/entry.S>

.macro kernel_entry
    addi sp, sp, -(PT_SIZE)

    sd x1,  PT_RA(sp)
    sd x3,  PT_GP(sp)
    sd x5,  PT_T0(sp)
    sd x6,  PT_T1(sp)
    sd x7,  PT_T2(sp)
    sd x8,  PT_S0(sp)
    sd x9,  PT_S1(sp)
    sd x10, PT_A0(sp)
    sd x11, PT_A1(sp)
    sd x12, PT_A2(sp)
    sd x13, PT_A3(sp)
    sd x14, PT_A4(sp)
    sd x15, PT_A5(sp)
    sd x16, PT_A6(sp)
    sd x17, PT_A7(sp)
    sd x18, PT_S2(sp)
    sd x19, PT_S3(sp)
    sd x20, PT_S4(sp)
    sd x21, PT_S5(sp)
    sd x22, PT_S6(sp)
    sd x23, PT_S7(sp)
    sd x24, PT_S8(sp)
    sd x25, PT_S9(sp)
    sd x26, PT_S10(sp)
    sd x27, PT_S11(sp)
    sd x28, PT_T3(sp)
    sd x29, PT_T4(sp)
    sd x30, PT_T5(sp)
    sd x31, PT_T6(sp)

    /*保存 sepc 寄存器的值*/
    csrr s2, sepc
    sd s2, PT_SEPC(sp)
```

```
37
38          /*保存 sbadaddr 寄存器的值*/
39          csrr s3, sbadaddr
40          sd s3, PT_SBADADDR(sp)
41
42          /*保存 scause 寄存器的值*/
43          csrr s4, scause
44          sd s4, PT_SCAUSE(sp)
45
46          /*保存 ssratch 寄存器的值*/
47          csrr s5, sscratch
48          sd s5, PT_TP(sp)
49
50          /*保存 sp*/
51          addi s0, sp, PT_SIZE
52          sd s0, PT_SP(sp)
53      .endm
```

在第 1 行中，使用“.macro”伪指令来声明一个汇编宏，“.endm”表示汇编宏的结束。

在第 2 行中，使用 ADDI 指令在进程的内核栈中为 pt_regs 栈框开辟一段空间，此时 sp 寄存器指向栈顶。

在第 4～32 行中，保存 RISC-V 通用寄存器的值到栈框里。

在第 35～36 行中，读取 sepc 寄存器的值，并保存到栈框的 S_SEPC 位置上。

在第 39～40 行，读取 sbadaddr 寄存器的值，并保存到栈框的 S_SBADADDR 位置上。

在第 43～44 行中，读取 scause 寄存器的值，并保存到栈框的 S_SCAUSE 位置上。

在第 47～48 行中，读取 sscratch 寄存器的值，并保存到栈框的 S_TP 位置上。

在第 51～52 行中，把栈底（SP_bottom）保存到栈框的 S_SP 位置上，如图 8.13 所示。

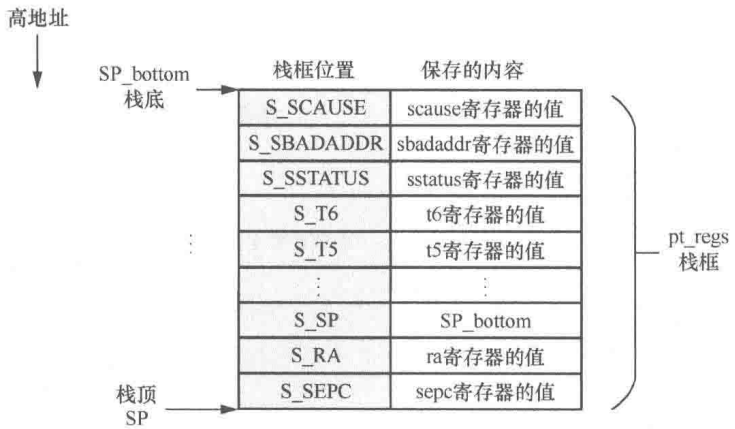


图 8.13　保存异常现场时的栈框

注意，在本场景中，因为目前仅在 S 模式下触发异常并且还没有实现进程控制块（Process Control Block，PCB），所以暂时不需要使用 sscratch 寄存器来保存 SP 等内容。

kernel_exit 宏的代码如下。

```
<benos/src/entry.S>

1     .macro kernel_exit
2         ld a0, PT_SSTATUS(sp)
3         csrw sstatus, a0
4
5         ld a2, PT_SEPC(sp)
6         csrw sepc, a2
7
8         ld x1,  PT_RA(sp)
9         ld x3,  PT_GP(sp)
```

```
10        ld x4,  PT_TP(sp)
11        ld x5,  PT_T0(sp)
12        ld x6,  PT_T1(sp)
13        ld x7,  PT_T2(sp)
14        ld x8,  PT_S0(sp)
15        ld x9,  PT_S1(sp)
16        ld x10, PT_A0(sp)
17        ld x11, PT_A1(sp)
18        ld x12, PT_A2(sp)
19        ld x13, PT_A3(sp)
20        ld x14, PT_A4(sp)
21        ld x15, PT_A5(sp)
22        ld x16, PT_A6(sp)
23        ld x17, PT_A7(sp)
24        ld x18, PT_S2(sp)
25        ld x19, PT_S3(sp)
26        ld x20, PT_S4(sp)
27        ld x21, PT_S5(sp)
28        ld x22, PT_S6(sp)
29        ld x23, PT_S7(sp)
30        ld x24, PT_S8(sp)
31        ld x25, PT_S9(sp)
32        ld x26, PT_S10(sp)
33        ld x27, PT_S11(sp)
34        ld x28, PT_T3(sp)
35        ld x29, PT_T4(sp)
36        ld x30, PT_T5(sp)
37        ld x31, PT_T6(sp)
38
39        ld x2,  PT_SP(sp)
40    .endm
```

恢复异常现场的顺序正好和保存中断现场的相反，从栈底开始依次恢复数据。

在第 2～3 行中，从栈框里恢复 sstatus 寄存器的值。

在第 5～6 行，从栈框里恢复 sepc 寄存器的值。

在第 8～37 行中，从栈框中依次恢复通用寄存器的值。

在第 39 行中，从栈框中恢复 sp 寄存器的值。

8.6.3 异常处理

异常根据类型细分为多种不同的异常。定义一个 fault_info 结构体来描述每一个具体的异常类型。

```
struct fault_info {
    int (*fn)(struct pt_regs *regs, const char *name);
    const char *name;
};
```

其中，fn 表示具体需要执行的异常处理函数，name 表示异常的名称。

我们基于 fault_info 结构体和表 8.10 初始化一个异常处理表（fault_info[]数组）。

```
1    static const struct fault_info fault_info[] = {
2        {do_trap_insn_misaligned, "Instruction address misaligned"},
3        {do_trap_insn_fault, "Instruction access fault"},
4        {do_trap_insn_illegal, "Illegal instruction"},
5        {do_trap_break, "Breakpoint"},
6        {do_trap_load_misaligned, "Load address misaligned"},
7        {do_trap_load_fault, "Load access fault"},
8        {do_trap_store_misaligned, "Store/AMO address misaligned"},
9        {do_trap_store_fault, "Store/AMO access fault"},
10       {do_trap_ecall_u, "Environment call from U-mode"},
11       {do_trap_ecall_s, "Environment call from S-mode"},
12       {do_trap_unknown, "unknown 10"},
13       {do_trap_unknown, "unknown 11"},
```

```
14        {do_page_fault, "Instruction page fault"},
15        {do_page_fault, "Load page fault"},
16        {do_trap_unknown, "unknown 14"},
17        {do_page_fault, "Store/AMO page fault"},
18    };
```

do_exception()函数首先会根据 scause 寄存器的最高位（Bit[63]）来判断是中断还是异常。若最高位为 1，表示中断；若最高位为 0，表示异常。is_interrupt_fault()函数做这个判断。

```
<benos/src/trap.c>

1     void do_exception(struct pt_regs *regs, unsigned long scause)
2     {
3         const struct fault_info *inf;
4
5         printk("%s, scause:0x%lx\n", __func__, scause);
6
7         if (is_interrupt_fault(scause)) {
8             /*TODO: 处理中断*/
9         } else {
10            inf = ec_to_fault_info(scause);
11
12            if (!inf->fn(regs, inf->name))
13                return;
14        }
15    }
```

在第 7～9 行中，处理中断，我们会在第 9 章中实现这里的代码。

在第 10～14 行中，ec_to_fault_info()函数查询异常处理表，然后调用表中 fn()处理函数。使用异常处理表中的处理函数，统一输出错误类型，然后就调用 panic()函数。

```
<benos/src/trap.c>

void panic()
{
    printk("Kernel panic\n");
    while(1)
        ;
}

static void do_trap_error(struct pt_regs *regs, const char *str)
{
    printk("Oops - %s\n", str);
    printk("sstatus:0x%016lx  sbadaddr:0x%016lx  scause:0x%016lx\n",
            regs->sstatus, regs->sbadaddr, regs->scause);
    panic();
}

#define DO_ERROR_INFO(name)                 \
int name(struct pt_regs *regs, const char *str)                \
{                                   \
    do_trap_error(regs, str);  \
    return 0;              \
}

DO_ERROR_INFO(do_trap_unknown);
DO_ERROR_INFO(do_trap_insn_misaligned);
DO_ERROR_INFO(do_trap_insn_fault);
DO_ERROR_INFO(do_trap_insn_illegal);
DO_ERROR_INFO(do_trap_load_misaligned);
DO_ERROR_INFO(do_trap_load_fault);
DO_ERROR_INFO(do_trap_store_misaligned);
DO_ERROR_INFO(do_trap_store_fault);
DO_ERROR_INFO(do_trap_ecall_u);
DO_ERROR_INFO(do_trap_ecall_s);
DO_ERROR_INFO(do_trap_break);
DO_ERROR_INFO(do_page_fault);
```

8.6.4 委托中断和异常

一般情况下，当中断和异常触发后，处理器会切换到 M 模式下。不过为了避免频繁在 S 模式和 M 模式之间来回切换，我们通常把常用的中断和异常委托给 S 模式来处理。于是，当中断和异常触发之后，处理器直接跳转到 S 模式下的异常向量表。

```
<benos/sbi/sbi_trap.c>

void delegate_traps(void)
{
    unsigned long interrupts;
    unsigned long exceptions;

    interrupts = MIP_SSIP | MIP_STIP | MIP_SEIP;
    exceptions = (1UL << CAUSE_MISALIGNED_FETCH) |
                 (1UL << CAUSE_FETCH_PAGE_FAULT) |
                 (1UL << CAUSE_BREAKPOINT) |
                 (1UL << CAUSE_LOAD_PAGE_FAULT) |
                 (1UL << CAUSE_STORE_PAGE_FAULT) |
                 (1UL << CAUSE_USER_ECALL) |
                 (1UL << CAUSE_LOAD_ACCESS) |
                 (1UL << CAUSE_STORE_ACCESS);

     write_csr(mideleg, interrupts);
     write_csr(medeleg, exceptions);
}
```

delegate_traps()函数把如下中断委托给 S 模式：

- 软件中断；
- 时钟中断；
- 外设中断。

另外，delegate_traps()函数把如下异常给 S 模式：

- 指令地址没对齐异常；
- 指令缺页异常；
- 加载缺页异常；
- 存储/AMO 缺页异常；
- 系统调用异常；
- 加载访问异常；
- 存储/AMO 访问异常；
- 断点异常。

8.6.5 触发异常

接下来，我们通过一个简单的汇编函数触发一个异常。

```
.global trigger_fault
trigger_fault:
    li a0, 0x70000000
    ld a0, (a0)
    ret
```

trigger_fault()汇编函数访问了一个非法的内存地址 0x7000 0000。它会触发一个加载访问异常。

下面是运行 BenOS 的结果。

```
do_exception, scause:0x5
Oops - Load access fault
```

```
sstatus:0x8000000000006100  sbadaddr:0x0000000070000000  scause:0x0000000000000005
Kernel panic
```

从日志可知，trigger_fault()函数导致系统触发了加载访问异常，并且 scause 寄存器显示异常类型编码为 5。

为了进一步帮助开发人员定位问题，我们可以把 pt_regs 栈框保存的内容都输出。

```
<benos/src/trap.c>

void show_regs(struct pt_regs *regs)
  {
      printk("sepc: %016lx ra : %016lx sp : %016lx\n",
          regs->sepc, regs->ra, regs->sp);
      printk(" gp : %016lx tp : %016lx t0 : %016lx\n",
          regs->gp, regs->tp, regs->t0);
      printk(" t1 : %016lx t2 : %016lx s0 : %016lx\n",
          regs->t1, regs->t2, regs->s0);
      printk(" s1 : %016lx a0 : %016lx a1 : %016lx\n",
          regs->s1, regs->a0, regs->a1);
      printk(" a2 : %016lx a3 : %016lx a4 : %016lx\n",
          regs->a2, regs->a3, regs->a4);
      printk(" a5 : %016lx a6 : %016lx a7 : %016lx\n",
          regs->a5, regs->a6, regs->a7);
      printk(" s2 : %016lx s3 : %016lx s4 : %016lx\n",
          regs->s2, regs->s3, regs->s4);
      printk(" s5 : %016lx s6 : %016lx s7 : %016lx\n",
          regs->s5, regs->s6, regs->s7);
      printk(" s8 : %016lx s9 : %016lx s10: %016lx\n",
          regs->s8, regs->s9, regs->s10);
      printk(" s11: %016lx t3 : %016lx t4: %016lx\n",
          regs->s11, regs->t3, regs->t4);
      printk(" t5 : %016lx t6 : %016lx\n",
          regs->t5, regs->t6);
  }
```

最后，BenOS 输出的结果如下。

```
do_exception, scause:0x5
Oops - Load access fault
sepc: 00000000802019ec ra : 00000000802018fc sp : 0000000080203ff0
 gp : 0000000000000000 tp : 0000000000000000 t0 : 0000000000000005
 t1 : 0000000000000005 t2 : 0000000080200020 s0 : 0000000080017f20
 s1 : 0000000080200010 a0 : 0000000070000000 a1 : 000000000000000a
 a2 : 0000000000000006 a3 : 0000000080203ef0 a4 : 0000000000000031
 a5 : 0000000000000031 a6 : 0000000000000002 a7 : 0000000000000061
 s2 : 8000000000006800 s3 : 0000000080200000 s4 : 0000000082200000
 s5 : 0000000000000000 s6 : 0000000000000000 s7 : 00000000800120e8
 s8 : 000000008020002e s9 : 000000000000007f s10: 0000000000000000
 s11: 0000000000000000 t3 : 0990106f91166285 t4: 0000000080017ee0
 t5 : 0000000000000027 t6 : 0000000000000000
sstatus:0x8000000000006100  sbadaddr:0x0000000070000000  scause:0x0000000000000005
Kernel panic
```

8.7 实验

8.7.1　实验 8-1：在 SBI 中实现串口输入功能

1. 实验目的

加深对异常处理流程的理解。

2. 实验要求

在 MySBI 固件中实现 SBI_CONSOLE_GETCHAR 的服务接口并测试。

8.7.2 实验 8-2：在 BenOS 中触发非法指令异常

1. 实验目的

熟悉异常处理的流程。

2. 实验要求

在 BenOS 中触发一个非法指令异常，如图 8.14 所示。

提示：可以使用如下两种方式触发非法指令异常。

- 在 S 模式下访问 M 模式下的寄存器，如 mstatus 寄存器。
- 通过篡改代码段里的指令代码触发一个非法指令访问异常。

例如，下面的代码把 trigger_load_access_fault()汇编函数的第 1 行代码篡改了。

```
void create_illegal_intr(void)
{
    int *p = (int *)trigger_load_access_fault;

    *p = 0xbadbeef;
}
```

8.7.3 实验 8-3：输出触发异常时函数栈的调用过程

1. 实验目的

熟悉异常处理的流程。

熟悉栈的布局。

2. 实验要求

在 BenOS 中触发一个异常之后，输出函数栈的调用过程，如图 8.15 所示。

```
do_exception, scause:0x2
Oops - Illegal instruction
sepc: 00000000802dc2a2 ra : 0000000080201078 sp : 0000000080203ff0
 gp : 0000000000000000 tp : 0000000000000000 t0 : 0000000000000005
 t1 : 0000000000000005 t2 : 0000000080200020 t3 : 0000000000000000
 s1 : 0000000080203010 a0 : 0000000000000031 a1 : 0000000000000000
 a2 : 0000000000000000 a3 : 00000000802008a0 a4 : 000000000badbeef
 a5 : 0000000000201fe8 a6 : 0000000000000000 a7 : 0000000000000001
 s2 : 0000000000000000 s3 : 0000000000000000 s4 : 0000000000000000
 s5 : 0000000000000000 s6 : 0000000000000000 s7 : 0000000000000000
 s8 : 0000000080200034 s9 : 0000000000000000 s10: 0000000000000000
 s11: 0000000000000000 t3 : 0510133908012b7 t4: 0000000080201fec
 t5 : 0000000000000000 t6 : 0000000000000000
sstatus:0x0000000050000100  sbadaddr:0x0000000000000000  scause:0x0000000000000002
Kernel panic
```

图 8.14 触发非法指令异常

```
do_exception, scause:0x5
Oops - Load access fault
Call Trace:
[<0x00000000802016ec>] walk_stackframe+0x0/0xc0
[<0x00000000802017d0>] show_stack+0x24/0x32
[<0x00000000802017e8>] dump_stack+0xa/0x18
[<0x0000000080201177>] do_trap_error+0x20/0x62
[<0x00000000802012a0>] do_trap_load_fault+0x18/0x28
[<0x0000000080201418>] do_exception+0x50/0x5c
[<0x00000000802020d6>] do_exception_vector+0x72/0x76
[<0x000000008020204c>] kernel_main+0x44/0x4e
```

图 8.15 函数栈的调用过程

8.7.4 实验 8-4：在 MySBI 中模拟实现 RDTIME 伪指令

1. 实验目的

熟悉异常的处理流程以及 CSR。

2. 实验要求

在 BenOS 中使用如下代码来读取系统的实际时间。在 S 模式下使用 RDTIME 伪指令会触发非法指令异常，处理器陷入 M 模式。在 M 模式下的异常处理程序中需要识别和处理该非法指令，读取 time 系统寄存器的值，然后返回给 S 模式。请在 MySBI 中实现上述模拟过程。

```
static inline unsigned long get_cycles(void)
{
    unsigned long n;

    asm volatile (
        "rdtime %0"
        : "=r" (n));
    return n;
}
```

第 9 章　中断处理与中断控制器

本章思考题

1. 请简述中断处理的一般过程。
2. 什么是中断现场？对于 RISC-V 处理器来说，中断现场应该保存哪些内容？
3. 中断现场保存到什么地方？

本章主要介绍与 RISC-V 处理器相关的中断处理的知识。

9.1 中断处理基本概念

在 RISC-V 架构里，中断属于异步异常的一种，其处理过程与异常处理很类似。许多 RISC-V 处理器同时支持 M 模式和 S 模式下的中断。在默认情况下，中断会在 M 模式下处理。如果处理器支持 S 模式，那么可以有选择地把部分中断委托给 S 模式来处理。

9.1.1 中断类型

RISC-V 架构把中断分成 4 类：

- 软件中断（software interrupt）；
- 定时器中断（timer interrupt）；
- 外部中断（external interrupt）；
- 调试中断（debug interrupt）。

软件中断指的是由软件触发的中断，通常用于处理器内核之间的通信，即处理器间中断（Inter-Processor Interrupt，IPI）。

定时器中断指的是来自定时器的中断，通常用于操作系统的时钟中断。在 RISC-V 体系结构中，在 M 模式和 S 模式下都有定时器。RISC-V 体系结构规定处理器必须有一个定时器，通常实现在 M 模式。RISC-V 体系结构还为定时器定义了两个 64 位的寄存器 mtime 和 mtimecmp。它们通常实现在 CLINT 中。

外部中断通常是指来自处理器外部设备（如串口设备等）的中断。RISC-V 体系结构在 M 模式和 S 模式下都可以处理外部中断。为了支持更多的外部中断源，处理器一般采用中断控制器来管理，例如，RISC-V 体系结构定义了一个平台级别的中断控制器（Platform-Level Interrupt Controller，PLIC），用于外部中断的仲裁和派发功能。

调试中断一般用于硬件调试功能。

在 RISC-V 处理器中，中断按照功能又可以分成如下两类。

- 本地（local）中断：直接发送给本地处理器硬件线程（Hart），它是一个处理器私有的

中断并且有固定的优先级。本地中断可以有效缩短中断延时，因为它不需要经过中断控制器的仲裁以及额外的中断查询。软件中断和时钟中断是常见的本地中断。本地中断一般由处理器内核本地中断器（Core-Local Interruptor，CLINT）来产生。

- 全局（global）中断：通常指的是外部中断，经过 PLIC 的路由，送到合适的处理器内核。PLIC 支持更多的中断号、可配置的优先级以及路由策略等。

中断框图如图 9.1 所示。

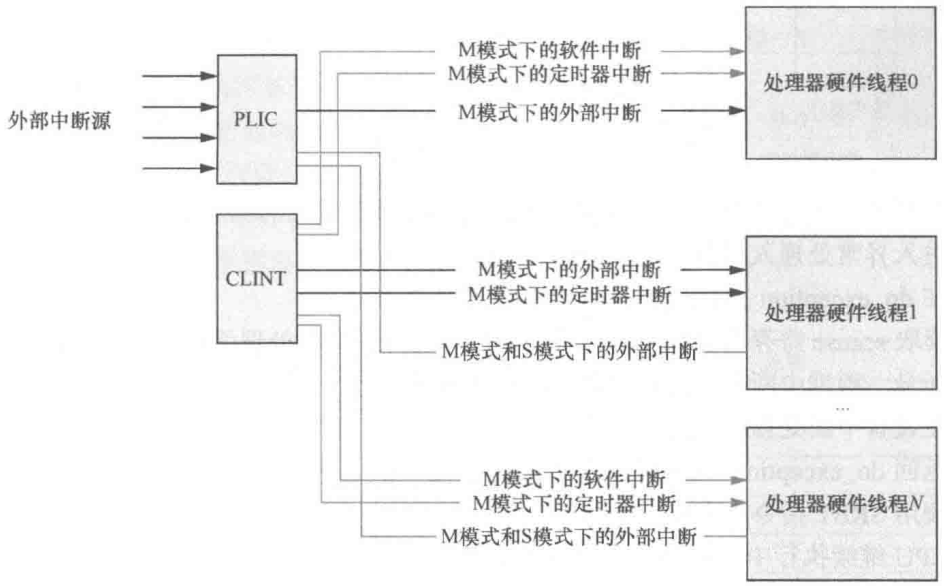


图 9.1 中断框图

9.1.2 中断处理过程

触发中断后，默认情况下由 M 模式响应和处理。处理器所做的事情与异常处理类似。这里假设中断已经委派并由 S 模式来处理。处理器做如下事情。

（1）保存中断发生前的中断状态，即把中断发生前的 SIE 位保存到 sstatus 寄存器中的 SPIE 字段。

（2）保存中断发生前的处理器模式状态，即把异常发生前的处理器模式编码保存到 sstatus 寄存器的 SPP 字段中。

（3）关闭本地中断，即设置 sstatus 寄存器中的 SIE 字段为 0。

（4）把中断类型更新到 scause 寄存器中。

（5）把触发中断时的虚拟地址更新到 stval 寄存器中。

（6）当前 PC 保存到 sepc 寄存器中。

（7）跳转到异常向量表，即把 stvec 寄存器的值设置到 PC 寄存器中。

操作系统软件需要读取以及解析 scause 寄存器的值来确定中断类型，然后跳转到相应的中断处理函数中。

中断处理完成之后，需要执行 SRET 指令来退出中断。SRET 指令会执行如下操作。

（1）恢复 SIE 字段，该字段的值从 sstatus 寄存器中的 SPIE 字段获取，这相当于使能了本地中断。

（2）将处理器模式设置成之前保存到 SPP 字段的模式编码。

（3）设置 PC 为 sepc 寄存器的值，即返回异常触发的现场。

下面以一个例子来说明中断处理的一般过程，如图 9.2 所示。假设有一个正在运行的程序，这个程序可能运行在内核模式，也可能运行在用户模式，此时，一个外设中断发生了。

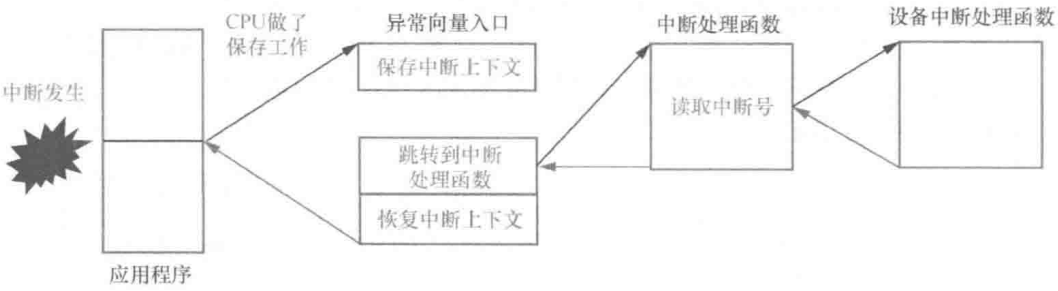


图 9.2　中断处理过程

（1）CPU 会自动做上文所述的事情，并跳转到异常向量表的基地址。

（2）进入异常处理入口函数，如 do_exception_vector()。

（3）在 do_exception_vector()汇编函数里保存中断现场。

（4）读取 scause 寄存器的值，解析中断类型，跳转到中断处理函数里。例如，在 PLIC 驱动里读取中断号，根据中断号跳转到设备中断处理程序。

（5）在设备中断处理程序里处理这个中断。

（6）返回 do_exception_vector()汇编函数，恢复中断上下文。

（7）调用 SRET 指令来完成中断返回。

（8）CPU 继续执行中断现场的下一条指令。

9.1.3　中断委派和注入

在 RISC-V 体系结构中，与异常一样，中断默认情况下由 M 模式来响应和处理。运行在 M 模式的软件（如 OpenSBI）可以通过在 mideleg 寄存器中设置相应的位，有选择地将中断委托给 S 模式。mideleg 寄存器用来设置中断委托。mideleg 寄存器中的字段如表 9.1 所示。

表 9.1　mideleg 寄存器中的字段

字段	位	说明
SSIP	Bit[1]	把软件中断委托给 S 模式
STIP	Bit[5]	把时钟中断委托给 S 模式
SEIP	Bit[9]	把外部中断委托给 S 模式

RISC-V 体系结构提供一种中断注入方式（例如，使用 M 模式下的 mtimer 定时器）把 M 模式特有的中断注入 S 模式。mip 寄存器用来向 S 模式注入中断，例如，设置 mip 寄存器中的 STIP 字段相当于把 M 模式下的定时器中断注入 S 模式，并由 S 模式的操作系统处理，详见 9.3.3 节。

9.1.4　中断优先级

RISC-V 体系结构支持的多种不同类型的中断是有优先级的。如果多个不同类型中断同时触发，RISC-V 处理器会优先处理优先级高的中断类型。下面的中断类型按优先级从高到低排序：

- M 模式下的外部中断；
- M 模式下的软件中断；
- M 模式下的定时器中断；
- S 模式下的外部中断；

- S 模式下的软件中断；
- S 模式下的定时器中断。

例如，假设 M 模式下的外部中断和 S 模式下的外部中断同时触发，那么处理器优先处理 M 模式下的外部中断。如果有多个来自 S 模式的外部中断，那么优先级由 PLIC 来管理。

9.2 CLINT

RISC-V 处理器一般支持软件中断、时钟中断这两种本地中断，它们属于处理器内核私有的中断，直接发送到处理器内核，而不需要经过中断控制器的路由，如图 9.3 所示。

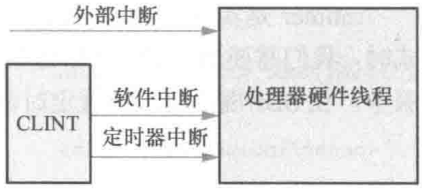


图 9.3 CLINT

CLINT 支持的中断采用固定优先级策略，高优先级的中断可以抢占低优先级的中断。CLINT 支持的中断如表 9.2 所示。中断号越大，优先级越高。

表 9.2 CLINT 支持的中断

名称	中断号	说明
ssip	1	S 模式下的软件中断
msip	3	M 模式下的软件中断
stip	5	S 模式下的时钟中断
mtip	7	M 模式下的时钟中断
seip	9	S 模式下的外部中断
meip	11	M 模式下的外部中断

FU740 处理器的 CLINT 中的寄存器如表 9.3 所示。在 CLINT 控制器中，没有设置专门的寄存器来使能每个中断，不过可以使用 mie 寄存器来控制每个本地中断。另外，还可以使用 mstatus 寄存器中 MIE 字段来关闭和打开全局中断。

表 9.3 CLINT 中的寄存器

名称	地址	属性	位宽	描述
MSIP	0x200 0000	RW	32	机器特权模式下的软件触发寄存器，用于处理器硬件线程 0
MSIP	0x200 0004	RW	32	机器特权模式下的软件触发寄存器，用于处理器硬件线程 1
MSIP	0x200 0008	RW	32	机器特权模式下的软件触发寄存器，用于处理器硬件线程 2
MSIP	0x200 000C	RW	32	机器特权模式下的软件触发寄存器，用于处理器硬件线程 3
MSIP	0x200 0010	RW	32	机器特权模式下的软件触发寄存器，用于处理器硬件线程 4
MTIMECMP	0x200 4000	RW	64	定时器比较寄存器，用于处理器硬件线程 0
MTIMECMP	0x200 4008	RW	64	定时器比较寄存器，用于处理器硬件线程 1
MTIMECMP	0x200 4010	RW	64	定时器比较寄存器，用于处理器硬件线程 2
MTIMECMP	0x200 4018	RW	64	定时器比较寄存器，用于处理器硬件线程 3
MTIMECMP	0x200 4020	RW	64	定时器比较寄存器，用于处理器硬件线程 4
MTIME	0x200 BFF8	RW	64	定时器寄存器

其中 MSIP 寄存器主要用来触发软件中断，用于多处理器硬件线程之间的通信，如 IPI。MTIMECMP 和 MTIME 是 M 模式下与定时器相关的寄存器。MTIME 寄存器返回系统的时钟周期数。MTIMECMP 寄存器用来设置时间间隔，当 MTIME 返回的时间大于或者等于 MTIMECMP 寄存器的值时，便会触发定时器中断。

9.3 案例分析 9-1：定时器中断

在这个案例中，我们利用 M 模式下的 mtimer 作为中断源来分析要完整实现一个中断处理需要做哪些事情。

9.3.1 访问 mtimer

mtimer 是实现在 M 模式下的定时器，它位于 CLINT 控制器内部。当操作系统运行在 S 模式时，我们需要通过 ECALL 指令陷入 M 模式以初始化 mtimer。我们需要在 MySBI 中实现这个服务。在 SBI 规范中，设置定时器的系统调用号为 0，定义的接口为 SBI_SET_TIMER。

```
<benos/include/asm/sbi.h>

1     /*
2      * SBI 提供 mtimer 服务
3      */
4     #define SBI_SET_TIMER 0
5
6 static inline void sbi_set_timer(unsigned long stime_value)
7 {
8     SBI_CALL_1(SBI_SET_TIMER, stime_value);
9 }
```

接下来，使用 sbi_set_timer()函数来初始化 mtimer。

```
<benos/src/timer.c>

1     #define CLINT_TIMEBASE_FREQ 10000000
2     #define HZ 1000
3
4     static inline unsigned long get_cycles(void)
5     {
6         return readq(VIRT_CLINT_TIMER_VAL);
7     }
8
9     void reset_timer()
10    {
11        sbi_set_timer(get_cycles() + CLINT_TIMEBASE_FREQ/HZ);
12        csr_set(sie, SIE_STIE);
13    }
14
15
16    void timer_init(void)
17    {
18        reset_timer();
19    }
```

在第 4 行中，get_cycles()函数读取 MTIME 寄存器来获取当前的时间。

在第 11 行中，通过 sbi_set_timer()函数调用 MySBI 固件提供的服务来给定时器设置一个初始值。初始值为当前时间再加上一个未来时间间隔，CLINT_TIMEBASE_FREQ 宏表示 mtimer 的频率，HZ 表示每秒触发多少次时钟中断。

在第 12 行中，打开 sie 寄存器中 S 模式下的定时器中断控制位。

9.3.2 在 MySBI 中实现定时器服务

我们需要在 MySBI 固件中实现定时器的系统调用服务并设置中断委托。下面在 sbi_ecall_handle()函数中增加对 SBI_SET_TIMER 系统调用的处理。

```
<benos/sbi/sbi_trap.c>

1     static int sbi_ecall_handle(unsigned int id, struct sbi_trap_regs *regs)
```

```
2    {
3        int ret = 0;
4
5        switch (id) {
6        case SBI_SET_TIMER:
7            clint_timer_event_start(regs->a0);
8            ret = 0;
9            break;
10       ...
11       }
12       ...
13       return ret;
14   }
```

在第 6～9 行中，当系统调用号为 SBI_SET_TIMER 时，调用 clint_timer_event_start()函数进行处理。

```
<benos/sbi/sbi_timer.c>

void clint_timer_event_start(unsigned long next_event)
{
        writeq(next_event, VIRT_CLINT_TIMER_CMP);
}
```

clint_timer_event_start()完成的处理流程比较简单，把下一次预设的定时器值 next_event 设置到 MTIMECMP 寄存器中。

9.3.3 定时器中断处理

触发定时器中断之后，处理器首先在 M 模式响应该中断，我们需要做的是关闭 M 模式下的定时器中断，然后委托给 S 模式来处理。

```
<benos/src/sbi_trap.c>

1    void sbi_trap_handler(struct sbi_trap_regs *regs)
2    {
3        unsigned long mcause = read_csr(mcause);
4        unsigned long ecall_id = regs->a7;
5        int rc = SBI_ENOTSUPP;
6        const char *msg = "trap handler failed";
7
8        /*处理中断*/
9        if (mcause & MCAUSE_IRQ) {
10           mcause &=~MCAUSE_IRQ;
11           switch (mcause) {
12           case IRQ_M_TIMER:
13               sbi_timer_process();
14               break;
15           default:
16               msg = "unhandled external interrupt";
17               goto trap_error;
18           }
19           return;
20       }
21
22       switch (mcause) {
23       /*处理异常*/
24       }
25
26   trap_error:
27       if (rc) {
28           sbi_trap_error(regs, msg, rc);
29       }
30   }
```

在第 9～20 行中，处理中断。若 mcause 寄存器的最高位为 1，表示中断。当该中断为定时

器中断时，调用 sbi_timer_process()来处理中断。

```
<benos/sbi/sbi_timer.c>

void sbi_timer_process(void)
{
    csr_clear(mie, MIP_MTIP);
    csr_set(mip, MIP_STIP);
}
```

sbi_timer_process()函数比较简单，它把 M 模式下的定时器中断关闭，然后通过设置 mip 寄存器中的 STIP 字段把定时器的等待中断状态注入 S 模式。于是，定时器中断就可以委托给 S 模式来处理。另外，我们还需要设置 mideleg 寄存器把 M 模式下的时钟中断委托给 S 模式。

```
<benos/sbi/sbi_trap.c>

1   void delegate_traps(void)
2   {
3       unsigned long interrupts;
4
5       interrupts = MIP_SSIP | MIP_STIP | MIP_SEIP;
6       write_csr(mideleg, interrupts);
7   }
```

当中断转发到 S 模式后，CPU 自动跳转到 S 模式下的异常向量表入口地址，即 do_exception_vector()汇编函数。在 do_exception_vector()汇编函数里，首先需要保存中断现场，然后跳转到中断处理函数 do_exception()。do_exception()函数的实现如下。

```
<benos/src/trap.c>

1    #define INTERRUPT_CAUSE_SOFTWARE    1
2    #define INTERRUPT_CAUSE_TIMER       5
3    #define INTERRUPT_CAUSE_EXTERNAL    9
4
5    void do_exception(struct pt_regs *regs, unsigned long scause)
6    {
7        const struct fault_info *inf;
8
9        if (is_interrupt_fault(scause)) {
10           switch (scause &~SCAUSE_INT) {
11           case INTERRUPT_CAUSE_TIMER:
12               handle_timer_irq();
13               break;
14           case INTERRUPT_CAUSE_EXTERNAL:
15           /* todo: 处理 IRQ */
16               break;
17           case INTERRUPT_CAUSE_SOFTWARE:
18           /* todo: 处理 IPI */
19               break;
20           default:
21               printk("unexpected interrupt cause");
22               panic();
23           }
24       } else {
25           inf = ec_to_fault_info(scause);
26
27           if (!inf->fn(regs, inf->name))
28               return;
29       }
30   }
```

第 10～23 行为新增加的代码，主要目标是通过解析 scause 寄存器来确定中断类型。当发现本次中断为定时器中断时，调用 handle_timer_irq()函数来处理。

handle_timer_irq()函数的实现如下。

```
<benos/src/timer.c>

void handle_timer_irq(void)
{
    csr_clear(sie, SIE_STIE);
    reset_timer();
    jiffies++;
    printk("Core0 Timer interrupt received, jiffies=%lu\r\n", jiffies);
}
```

在上述代码中，首先关闭 S 模式下的定时器中断，然后调用 reset_timer()函数来重新给定时器设置初始值，最后输出“Core0 Timer interrupt received”。

handle_timer_irq()函数执行完毕之后会返回 do_exception_vector()汇编函数，调用 kernel_exit 宏来恢复中断现场，最后调用 SRET 指令返回中断现场。

9.3.4 打开中断总开关

为了使能某个中断源，我们需要做两件事情。

- 使能 sie 寄存器中相应的中断类型。
- 使能 CPU 的中断总开关，即 sstatus 寄存器中的 SIE 字段。

下面的 arch_local_irq_enable()函数用来打开 CPU 的中断总开关，arch_local_irq_disable()函数用来关闭 CPU 的中断总开关。

```
<benos/include/asm/irq.h>

/*使能中断总开关*/
static inline void arch_local_irq_enable(void)
{
    csr_set(sstatus, SR_SIE);
}

/*关闭中断总开关*/
static inline void arch_local_irq_disable(void)
{
    csr_clear(sstatus, SR_SIE);
}
```

最后，我们需要在 kernel_main()函数里调用 timer_init()和 arch_local_irq_enable()。

```
<benos/src/kernel.c>

void kernel_main(void)
{
    ...
    timer_init();
    arch_local_irq_enable();
    ...

    while (1) {
        ;
    }
}
```

下面是这个案例的运行结果。

```
BenOS image layout:
  .text.boot: 0x80200000 - 0x80200032 (    50 B)
       .text: 0x80200038 - 0x80202488 (  9296 B)
     .rodata: 0x80202488 - 0x80203d00 (  6264 B)
       .data: 0x80203d00 - 0x80205000 (  4864 B)
        .bss: 0x80205010 - 0x80225488 (132216 B)
Core0 Timer interrupt received, jiffies=1
Core0 Timer interrupt received, jiffies=2
```

```
Core0 Timer interrupt received, jiffies=3
Core0 Timer interrupt received, jiffies=4
Core0 Timer interrupt received, jiffies=5
```

9.3.5　小结

在 S 模式下使用定时器的流程如下。

（1）初始化定时器。

① 通过 sbi_set_timer()函数给定时器设置一个初始值。

② 使能定时器中断。设置 sie 寄存器中 STIE 字段。

③ 通过 arch_local_irq_enable()函数打开中断总开关。

（2）处理定时器中断。

① 触发定时器中断。

② 跳转到异常向量表入口地址——do_exception_vector()汇编函数。

③ 保存中断上下文（使用 kernel_entry 宏）。

④ 跳转到 do_exception()中断处理函数。

⑤ 读取 scause 寄存器来获取中断类型。

⑥ 判断是否为定时器中断源触发的中断。如果是，重新设置定时器。

⑦ 返回 do_exception_vector()汇编函数。

⑧ 恢复中断上下文。

⑨ 返回中断现场。

9.4　PLIC

PLIC 主要用来管理外部中断，最多可以支持 1024 个中断源与 15 872 个中断硬件上下文（context），不过具体还需要看处理器的设计与实现。这里的中断硬件上下文指的是在一个处理器硬件线程里具有处理中断能力的特权模式，如 M 模式以及 S 模式。假设某款处理器支持 SMT，即支持硬件超线程，一个处理器内核支持两个硬件线程，那么一个双核处理器就拥有 4 个处理器硬件线程，每个处理器硬件线程都支持 M 模式和 S 模式，这个处理器一共支持 8 个中断硬件上下文，如图 9.4 所示。

以 FU740 处理器为例，它支持 5 个处理器内核，并且不支持 SMT。其中处理器内核 0 用于系统监控，仅支持 M 模式，其余 4 个处理器内核都支持 M 模式和 S 模式，因此它一共支持 9 个中断硬件上下文。

- S7 处理器硬件线程 0 的 M 模式；
- U7 处理器硬件线程 1 的 M 模式；
- U7 处理器硬件线程 1 的 S 模式；
- U7 处理器硬件线程 2 的 M 模式；
- U7 处理器硬件线程 2 的 S 模式；
- U7 处理器硬件线程 3 的 M 模式；
- U7 处理器硬件线程 3 的 S 模式；
- U7 处理器硬件线程 4 的 M 模式；
- U7 处理器硬件线程 4 的 S 模式。

FU740 处理器的中断硬件上下文如图 9.5 所示。

图 9.4　某处理器的中断硬件上下文

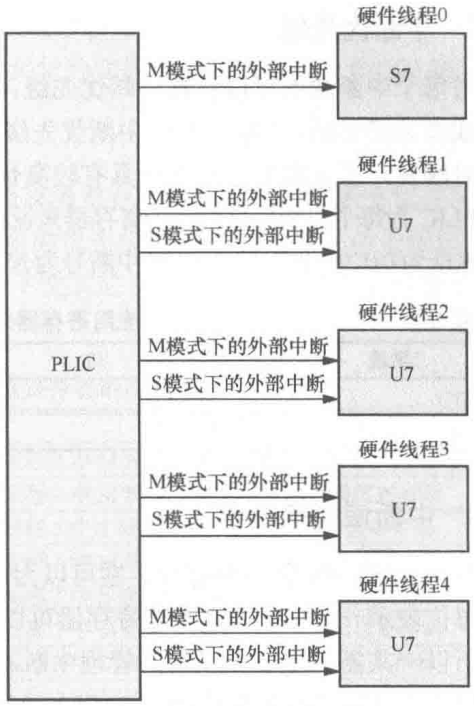


图 9.5　FU740 处理器的中断硬件上下文

9.4.1　中断号

FU740 处理器中的 PLIC 支持 69 个中断源以及 7 级中断优先级，其中第 0 号中断源是保留的。中断号分配如表 9.4 所示。

表 9.4　FU740 处理器的中断号分配

中断号	描述
1～10	MSI
11～18	DMA
19	L2 高速缓存目录错误
20	L2 高速缓存目录失败
21	L2 高速缓存数据错误
22	L2 高速缓存数据失败
23～38	GPIO
39	串口 0
40	串口 1
41	SPI 0
42	SPI 1
43	SPI 2
44～47	PWM 0
48～51	PWM 1
52	I2C 0
53	I2C 1
54	DDR
55	MAC
56～64	PCIE
65～69	总线错误

QEMU Virt 开发板的中断号分配见 2.1.1 节。

9.4.2　中断优先级

对每个中断源都可以设置中断优先级，PLIC 支持 7 级中断优先级。其中，0 表示不会触发中断或者关闭中断，1 表示最低中断优先级，7 表示最高中断优先级。如果两个相同优先级的中断同时触发，那么编号小的中断具有较高优先级。

PLIC 为每个中断号提供一个寄存器来设置相应中断的优先级，如表 9.5 所示。中断优先级寄存器的基地址为 0xC0 0000。例如，当中断号为 N 时，中断优先级寄存器的地址为 0xC0 0000 + 4 N。

表 9.5　　使用寄存器来设置相应中断的优先级

字段	位	属性	说明
Priority	Bit[2:0]	RW	设置中断优先级
Reserved	Bit[31:3]	RO	保留

9.4.3　中断使能寄存器

PLIC 提供中断使能寄存器，它可以为每个处理器内核使能和关闭中断源。中断使能寄存器中的每位表示一个中断源。一个寄存器可以表示 32 个中断源。因为 PLIC 最多支持 1024 个中断源，所以一共需要 128 个字节来管理中断源。以处理器上下文（硬件线程 0）的 M 模式为例，中断使能寄存器的地址范围是从 0xC00 2000 到 0xC00 2080。

另外，PLIC 最多支持 15 872 个处理器上下文，所以一共需要 128 × 15 872 = 2 031 616 字节。以 FU740 处理器为例，它最多支持 9 个处理器上下文，所以一共需要 1152 字节。

在 FU740 处理器中，由于只支持 53 个有效的中断源，因此使用两个寄存器就能实现一个处理器上下文中所有中断源的使能和关闭功能。其中断使能寄存器的布局如表 9.6 所示。

表 9.6　　FU740 处理器中 PLIC 的中断使能寄存器的布局

地址	位宽/位	属性	说明
0xC00 2000	4	RW	处理器硬件线程 0 中 M 模式下的中断使能寄存器 0
0xC00 2004	4	RW	处理器硬件线程 0 中 M 模式下的中断使能寄存器 1
0xC00 2080	4	RW	处理器硬件线程 1 中 M 模式下的中断使能寄存器 0
0xC00 2084	4	RW	处理器硬件线程 1 中 M 模式下的中断使能寄存器 1
0xC00 2100	4	RW	处理器硬件线程 1 中 S 模式下的中断使能寄存器 0
0xC00 2104	4	RW	处理器硬件线程 1 中 S 模式下的中断使能寄存器 1
0xC00 2180	4	RW	处理器硬件线程 2 中 M 模式下的中断使能寄存器 0
0xC00 2184	4	RW	处理器硬件线程 2 中 M 模式下的中断使能寄存器 1
0xC00 2200	4	RW	处理器硬件线程 2 中 S 模式下的中断使能寄存器 0
0xC00 2204	4	RW	处理器硬件线程 2 中 S 模式下的中断使能寄存器 1
0xC00 2280	4	RW	处理器硬件线程 3 中 M 模式下的中断使能寄存器 0
0xC00 2284	4	RW	处理器硬件线程 3 中 M 模式下的中断使能寄存器 1
0xC00 2300	4	RW	处理器硬件线程 3 中 S 模式下的中断使能寄存器 0
0xC00 2304	4	RW	处理器硬件线程 3 中 S 模式下的中断使能寄存器 1
0xC00 2380	4	RW	处理器硬件线程 4 中 M 模式下的中断使能寄存器 0
0xC00 2384	4	RW	处理器硬件线程 4 中 M 模式下的中断使能寄存器 1
0xC00 2400	4	RW	处理器硬件线程 4 中 S 模式下的中断使能寄存器 0
0xC00 2404	4	RW	处理器硬件线程 4 中 S 模式下的中断使能寄存器 1

9.4.4　中断待定寄存器

中断待定寄存器中的每一位表示一个中断源。一个寄存器可以表示 32 个中断源。因为 PLIC

最多支持 1024 个中断源，所以一共需要 128 字节来管理中断源。

在 FU740 处理器中，中断待定寄存器一共有两个，地址分别是 0xC00 1000 以及 0xC00 1004。

9.4.5 中断优先级阈值寄存器

PLIC 提供了一个处理器上下文级别的中断优先级阈值寄存器，它用来屏蔽所有优先级小于或等于阈值的中断源，只有当中断优先级大于该中断源的优先级阈值时，PLIC 才会处理该中断。

对于 FU740 处理器来说，它有 9 个中断硬件上下文。它的中断优先级阈值寄存器的布局如表 9.7 所示。

表 9.7　FU740 的中断优先级阈值寄存器的布局

地址	位宽/位	属性	说明
0xC20 0000	4	RW	处理器超线程 0 中 M 模式下的中断优先级阈值寄存器
0xC20 1000	4	RW	处理器超线程 1 中 M 模式下的中断优先级阈值寄存器
0xC20 2000	4	RW	处理器超线程 1 中 S 模式下的中断优先级阈值寄存器
0xC20 3000	4	RW	处理器超线程 2 中 M 模式下的中断优先级阈值寄存器
0xC20 4000	4	RW	处理器超线程 2 中 S 模式下的中断优先级阈值寄存器
0xC20 5000	4	RW	处理器超线程 3 中 M 模式下的中断优先级阈值寄存器
0xC20 6000	4	RW	处理器超线程 3 中 S 模式下的中断优先级阈值寄存器
0xC20 7000	4	RW	处理器超线程 4 中 M 模式下的中断优先级阈值寄存器
0xC20 8000	4	RW	处理器超线程 4 中 S 模式下的中断优先级阈值寄存器

9.4.6 中断请求/完成寄存器

中断请求（claim）寄存器和中断完成（complete）寄存器是同一个寄存器。

- 当触发中断时，软件通过读取中断请求寄存器可知哪个中断源是当前待定中断源中优先级最高的，软件可以在中断处理程序中处理该中断。
- 中断处理程序执行完后，软件可以把中断号写入中断完成寄存器中，PLIC 便知道软件已经完成了中断处理，PLIC 完成最后的中断处理工作。

对于 FU740 处理器来说，它有 9 个中断硬件上下文。它的中断请求/完成寄存器的布局如表 9.8 所示。

表 9.8　FU740 处理器的中断请求/完成寄存器的布局

地址	位宽/位	属性	说明
0xC20 0004	4	RW	处理器超线程 0 中 M 模式下的中断请求/完成寄存器
0xC20 1004	4	RW	处理器超线程 1 中 M 模式下的中断请求/完成寄存器
0xC20 2004	4	RW	处理器超线程 1 中 S 模式下的中断请求/完成寄存器
0xC20 3004	4	RW	处理器超线程 2 中 M 模式下的中断请求/完成寄存器
0xC20 4004	4	RW	处理器超线程 2 中 S 模式下的中断请求/完成寄存器
0xC20 5004	4	RW	处理器超线程 3 中 M 模式下的中断请求/完成寄存器
0xC20 6004	4	RW	处理器超线程 3 中 S 模式下的中断请求/完成寄存器
0xC20 7004	4	RW	处理器超线程 4 中 M 模式下的中断请求/完成寄存器
0xC20 8004	4	RW	处理器超线程 4 中 S 模式下的中断请求/完成寄存器

9.5 案例分析 9-2：串口中断

在 QEMU Virt 平台中，串口 0 设备的中断是通过 PLIC 来管理的，其中断号为 10。下面以串口 0 设备为例介绍 PLIC 的使用。

9.5.1　初始化 PLIC

在 QEMU Virt 平台中有多个处理器上下文，所以在访问 PLIC 的寄存器时需要特别小心。下面通过宏定义 PLIC 的寄存器。

```
<benos/include/asm/plic.h >

/*设置每个中断的优先级*/
#define PLIC_PRIORITY(hwirq) (PLIC_BASE + (hwirq) * 4)

/*每个中断的 pending 位，一位表示一个中断*/
#define PLIC_PENDING(hwirq) (PLIC_BASE + 0x1000 + ((hwirq) / 32) * 4)

/*中断使能位：每个处理器内核都有对应的中断使能位*/
#define PLIC_MENABLE(hart) (PLIC_BASE + 0x2000 + (hart) * 0x80)

/*设置每个处理器中每个中断的优先级阈值，当中断优先级大于阈值才会触发中断*/
#define PLIC_MTHRESHOLD(hart) (PLIC_BASE + 0x200000 + (hart) * 0x1000)

/*请求/完成寄存器*/
#define PLIC_MCLAIM(hart) (PLIC_BASE + 0x200004 + (hart) * 0x1000)
```

上述宏参数 hwirq 表示中断号，hart 表示处理器超线程或者处理器上下文。

接下来的 plic_init()函数用来初始化 PLIC。

```
<benos/src/plic.c>

1    int plic_init(void)
2    {
3        int i;
4        int hwirq;
5
6        for (i = 0; i < MAX_CPUS; i++) {
7            /*设置 M 模式下所有处理器内核的中断优先级阈值为 0*/
8            writel(0, PLIC_MTHRESHOLD(CPU_TO_HART(i)));
9
10           for (hwirq = 1; hwirq <= MAX_PLIC_IRQS; hwirq++) {
11               /*关闭 PLIC 中所有外部中断*/
12               plic_enable_irq(i, hwirq, 0);
13
14               /*预先设置所有中断号的优先级为 1*/
15               plic_set_prority(hwirq, 1);
16           }
17       }
18
19       csr_set(sie, SIE_SEIE);
20
21       return 0;
22   }
```

在第 19 行中，设置 sie 寄存器中的 SIE_SEIE 字段，打开 S 模式的外部中断。

plic_set_prority()函数和 plic_enable_irq()函数的实现如下。

```
<benos/src/plic.c>

1    #define CPU_TO_HART(cpu) ((2*cpu) + 1)
2
3    void plic_set_prority(int hwirq, int pro)
4    {
5        unsigned int reg = PLIC_PRIORITY(hwirq);
6
7        writel(pro, reg);
8    }
9
10   void plic_enable_irq(int cpu, int hwirq, int enable)
11   {
12       unsigned int hwirq_mask = 1 << (hwirq % 32);
```

```
    int hart = CPU_TO_HART(cpu);
    unsigned int reg = PLIC_MENABLE(hart) + 4*(hwirq / 32);

    printk("reg: 0x%x, hwirq:%d\n", reg, hwirq);

    if (enable)
        writel(readl(reg) | hwirq_mask, reg);
    else
        writel(readl(reg) & ~hwirq_mask, reg);
}
```

9.5.2 使能串口 0 的接收中断

16550 串口控制器中的 UART_IER 为中断使能寄存器，其中 Bit[0]为接收缓冲区满中断，我们需要在串口初始化时打开该中断。

```
<benos/src/uart.c>

void uart_init(void)
{
    ...
    /*使能接收缓冲区满中断*/
    writeb(0x1, UART_IER);
}
```

最后，我们还需要使能 PLIC 中串口 0 对应的中断源。

```
<benos/src/uart.c>

#define UART0_IRQ (10)

void enable_uart_plic()
{
    /*TODO: 使用 CPU0*/
    int cpu = 0;

    plic_enable_irq(cpu, UART0_IRQ, 1);
}
```

9.5.3 处理中断

当串口 0 的中断触发后，我们从 scause 寄存器读取到的中断类型为外部中断，即 INTERRUPT_CAUSE_EXTERNAL。

```
<benos/src/trap.c>

void do_exception(struct pt_regs *regs, unsigned long scause)
{
    const struct fault_info *inf;

    if (is_interrupt_fault(scause)) {
        switch (scause &~SCAUSE_INT) {
        case INTERRUPT_CAUSE_TIMER:
            handle_timer_irq();
            break;
        case INTERRUPT_CAUSE_EXTERNAL:
            plic_handle_irq(regs);
            break;
        case INTERRUPT_CAUSE_SOFTWARE:
        /*处理 IPI*/
            break;
        default:
            printk("unexpected interrupt cause");
            panic();
        }
```

```
20        } else {
21            ...
22        }
23    }
```

在第 11 行中，调用 plic_handle_irq()函数来处理外部中断。

plic_handle_irq()函数的实现如下。

```
<benos/src/plic.c>

1     void plic_handle_irq(struct pt_regs *regs)
2     {
3         int hwirq;
4         /*TODO: 仅 CPU0 处理*/
5         int hart = CPU_TO_HART(0);
6
7         unsigned int claim_reg = PLIC_MCLAIM(hart);
8
9         csr_clear(sie, SIE_SEIE);
10
11        while((hwirq = readl(claim_reg))) {
12            if (hwirq == UART0_IRQ)
13                handle_uart_irq();
14
15            writel(hwirq, claim_reg);
16        }
17        csr_set(sie, SIE_SEIE);
18    }
```

在第 9 行中，通过 SEIE 字段关闭 sie 寄存器中的外部中断。

在第 11 行中，读取中断请求寄存器，获取需要处理的中断号。

在第 13 行中，如果待处理的中断号为 UART0_IRQ，则跳转到 handle_uart_irq()函数中。

在第 15 行中，中断处理完成之后，把中断号写入中断请求/完成寄存器中，告诉 PLIC 软件已经处理完毕。

在第 17 行中，通过 SEIE 字段重新打开 sie 寄存器中的外部中断。

handle_uart_irq()函数的实现如下。

```
<benos/src/uart.c>

1     void handle_uart_irq(void)
2     {
3         char c;
4
5         c = uart_get();
6         if (c < 0)
7             return;
8         else if (c == '\r') {
9             printk("%s occurred\n", __func__);
10        }
11    }
```

在串口的中断处理函数中，先读取串口的内容，如果发现串口接收的数据为“回车符”，则输出“handle_uart_irq occurred”。下面是这个案例的运行结果。

```
BenOS image layout:
  .text.boot: 0x80200000 - 0x80200032 (    50 B)
       .text: 0x80200038 - 0x80202818 ( 10208 B)
     .rodata: 0x80202818 - 0x80204300 (  6888 B)
       .data: 0x80204300 - 0x80206000 (  7424 B)
        .bss: 0x80206010 - 0x80226488 (132216 B)
sstatus:0x8000000000006002
sstatus:0x8000000000006002, sie:0x222
handle_uart_irq occurred
handle_uart_irq occurred
```

9.6 实验

9.6.1 实验 9-1：定时器中断

1. 实验目的

熟悉 RISC-V 处理器的中断流程。

2. 实验要求

（1）在 QEMU 虚拟机中实现 mtimer 中断处理，并且在 QEMU 虚拟机上单步调试和观察。

（2）在汇编函数里实现保存中断现场的 kernel_entry 宏以及恢复中断现场的 kernel_exit 宏。

9.6.2 实验 9-2：使用汇编函数保存和恢复中断现场

1. 实验目的

熟悉如何保存和恢复中断现场。

2. 实验要求

（1）在实验 9-1 的基础上，把 kernel_entry 和 kernel_exit 两个宏修改成使用汇编函数实现。当修改成用汇编函数实现时，需要注意什么地方?

（2）请使用 QEMU 虚拟机和 GDB 或者 Eclipse 来单步调试中断处理过程，重点观察保存中断现场和恢复中断现场的寄存器的变化以及栈的变化情况。

9.6.3 实验 9-3：实现并调试串口 0 中断

1. 实验目的

熟悉 PLIC。

2. 实验要求

（1）在 BenOS 中实现案例 9-2 的串口中断，并用 GDB 单步调试中断处理过程。

（2）在案例 9-2 的基础上使用中断方式实现串口发送。

第 10 章　内存管理

本章思考题

1. 在计算机发展历史中，为什么会出现分段机制和分页机制？
2. 为什么页表要设计成多级页表？直接使用一级页表是否可行？多级页表又引入了什么问题？
3. 为什么页表存放在主内存中而不是存放在芯片内部的寄存器中？
4. MMU 查询页表的目的是找到虚拟地址对应的物理地址，页表项中有指向下一级页表基地址的指针，那它指向的是下一级页表基地址的物理地址还是虚拟地址？
5. 在 PMP 机制中，NAPOT 模式如何表示地址的范围？
6. 请简述在 RISC-V 处理器中 Sv39 页表映射模式的三级页表的映射过程。
7. 在打开 MMU 时，为什么需要建立恒等映射？
8. 在 RISC-V 处理器中，页表项属性中有一个 A 访问字段，它有什么作用？

本章主要介绍 RISC-V 体系结构中与内存管理相关的内容。

10.1 内存管理基础知识

10.1.1 内存管理的“远古时代”

在操作系统还没有出来之前，程序存放在卡片上，计算机每读取一张卡片就运行一条指令，这种从外部存储介质上直接运行指令的方法效率很低。后来出现了内存存储器，也就是说，程序要运行，首先要加载，然后执行，这就是所谓的“存储程序”。这一概念开启了操作系统快速发展的道路，直至后来出现的分页机制。在以上演变历史中，出现了两种内存管理思想。

- 单道编程的内存管理。所谓“单道”，就是整个系统只有一个用户进程和一个操作系统，形式上类似于 Unikernel 系统。这种模型下，用户程序始终加载到同一个内存地址并运行，所以内存管理很简单。实际上，不需要任何的 MMU，程序使用的地址就是物理地址，也不需要保护地址。但是缺点也很明显：其一，系统无法运行比实际物理内存大的程序；其二，系统只运行一个程序，会造成资源浪费；其三，程序无法迁移到其他的计算机中。
- 多道编程的内存管理。所谓“多道”，就是系统可以同时运行多个进程。内存管理中出现了固定分区和动态分区两种技术。

对于固定分区，在系统编译阶段，内存被划分成许多静态分区，进程可以装入大于或等于自身大小的分区。固定分区实现简单，操作系统的管理开销比较小。但是缺点也很明显：一是程序大小和分区的大小必须匹配；二是活动进程的数目比较固定；三是地址空间无法增长。

动态分区的思想就是在一整块内存中划出一块内存供操作系统本身使用，剩下的内存空间供用户进程使用。当进程 A 运行时，先从这一大片内存中划出一块与进程 A 大小一样的内存空间供进程 A 使用。当进程 B 准备运行时，从剩下的空闲内存中继续划出一块和进程 B 大小相等的内存空间供进程 B 使用，以此类推。这样进程 A 和进程 B 以及后面进来的进程就可以实现动态分区了。

如图 10.1 所示，假设现在有一块 32 MB 大小的内存，一开始操作系统使用了底部 4 MB 大小的内存，剩余的内存要留给 4 个用户进程使用，如图 10.1（a）所示。进程 A 使用了操作系统往上的 10 MB 内存，进程 B 使用了进程 A 往上的 6 MB 内存，进程 C 使用了进程 B 往上的 8 MB 内存。剩余的 4 MB 内存不足以装载进程 D（因为进程 D 需要 5 MB 内存），于是这块内存的末尾就形成了第一个空洞，如图 10.1（b）所示。假设在某个时刻操作系统需要运行进程 D，但系统中没有足够的内存，那么需要选择一个进程来换出，以便为进程 D 腾出足够的空间。假设操作系统选择进程 B 来换出，进程 D 就加载到原来进程 B 的地址空间里，于是产生了第二个空洞，如图 10.1（c）所示。假设操作系统在某个时刻需要运行进程 B，这也需要选择一个进程来换出，假设进程 A 被换出，于是系统中又产生了第三个空洞，如图 10.1（d）所示。

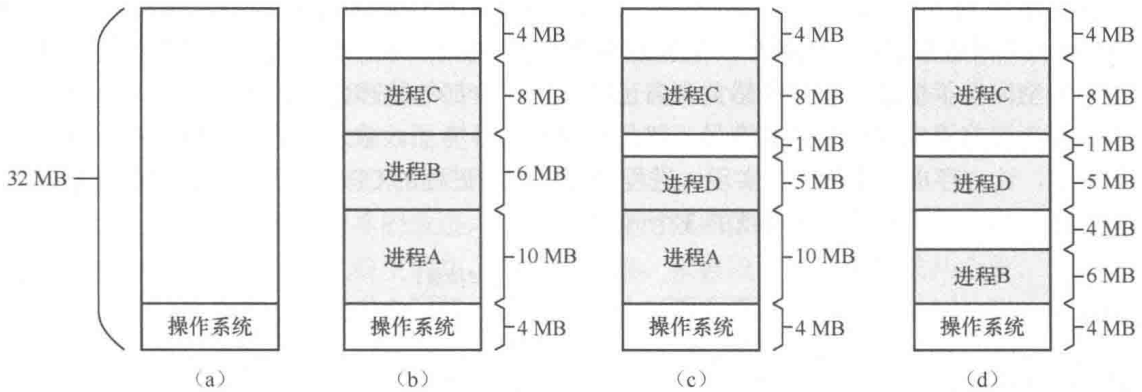


图 10.1 动态分区示意

这种动态分区方法在系统刚启动时效果很好，但是随着时间的推移会出现很多内存空洞，内存的利用率随之下降，这些内存空洞便是我们常说的内存碎片。为了解决内存碎片化的问题，操作系统需要动态地移动进程，使进程占用的空间是连续的，并且所有的空闲空间也是连续的。整个进程的迁移是一个非常耗时的过程。

总之，不管是固定分区还是动态分区，都存在很多问题。

- **进程地址空间保护问题**。所有的用户进程都可以访问全部的物理内存，所以恶意程序可以修改其他程序的内存数据，这使进程一直处于危险的状态下。即使系统里所有的进程都不是恶意进程，进程 A 也可能不小心修改了进程 B 的数据，从而导致进程 B 崩溃。这明显违背了“进程地址空间需要保护”（也就是地址空间要相对独立）的原则。因此，每个进程的地址空间都应该受到保护，以免被其他进程有意或无意地损坏。
- **内存使用效率低**。如果即将运行的进程所需要的内存空间不足，就需要选择一个进程并整体换出，这种机制导致大量的数据需要换出和换入，效率非常低下。
- **程序运行地址重定位问题**。从图 10.1 可以看出，进程在每次换出、换入时使用的地址都是不固定的，这给程序的编写带来一定的麻烦。因为访问数据和指令跳转时的目标地址通常是固定的，所以就需要使用重定位技术了。

由此可见，上述 3 个重大问题需要一个全新的解决方案，而且这个方案在操作系统层面已

经无能为力，必须在处理器层面才能解决，因此产生了分段机制和分页机制。

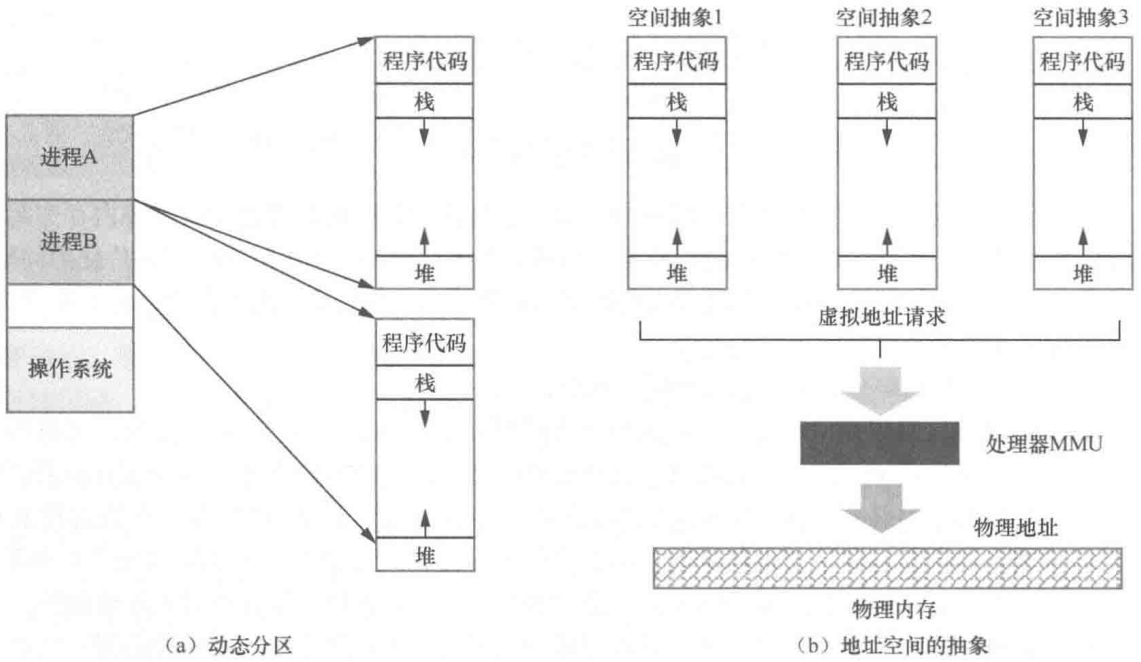
10.1.2　地址空间的抽象

站在内存使用的角度看，进程可能在 3 个地方需要用到内存。

- 在进程中。比如，代码段以及数据段用来存储程序本身需要的数据。
- 在栈空间中。程序运行时需要分配内存空间来保存函数调用关系、局部变量、函数参数以及函数返回值等内容，这些也是需要消耗内存空间的。
- 在堆空间中。程序运行时需要动态分配程序需要使用的内存，比如，存储程序需要使用的数据等。

不管是刚才提到的固定分区还是动态分区，进程、栈、堆都需要使用内存空间，如图 10.2（a）所示。但是，如果我们直接使用物理内存，在编写这样一个程序时，就需要时刻关心分配的物理内存地址是多少、内存空间够不够等问题。

后来，设计人员对内存进行了抽象，把上述用到的内存抽象成进程地址空间或虚拟内存。进程不用关心分配的内存在哪个地址，它只管使用。最终由处理器处理进程对内存的请求，经过转换之后把进程请求的虚拟地址转换成物理地址。这个转换过程称为地址转换（address translation），而进程请求的地址可以理解为虚拟地址，如图 10.2（b）所示。我们在处理器里对进程地址空间做了抽象，让进程感觉到自己可以拥有全部的物理内存。进程可以发出地址访问请求，至于这些请求能不能完全满足，就是处理器的事情了。总之，进程地址空间是对内存的重要抽象，让内存虚拟化得到了实现。进程地址空间、进程的 CPU 虚拟化以及文件对存储地址空间的抽象，共同组成了操作系统的 3 个元素。



（a）动态分区　　（b）地址空间的抽象

图 10.2　动态分区和地址空间的抽象

把进程地址空间的概念引入了虚拟内存后，基于这种思想，我们可以解决刚才提到的 3 个问题。

虚拟内存机制可以提供隔离性。因为每个进程都感觉自己拥有了整个地址空间，可以随意访问，然后由处理器转换到实际的物理地址，所以进程 A 没办法访问进程 B 的物理内存，也没

办法做破坏。

后来出现的分页机制可以解决动态分区中出现的内存碎片化和效率问题。

进程换入和换出时访问的地址变成相同的虚拟地址。进程不用关心具体物理地址在什么地方。

10.1.3 分段机制

基于进程地址空间这个概念，人们最早想到的一种机制叫作分段（segmentation）机制，其基本思想是把程序所需的内存空间的虚拟地址映射到某个物理地址空间。

分段机制可以解决地址空间保护问题，进程 A 和进程 B 会被映射到不同的物理地址空间，它们在物理地址空间中是不会有重叠的。因为进程看的是虚拟地址空间，不关心实际映射到哪个物理地址。如果一个进程访问了没有映射的虚拟地址空间，或者访问了不属于该进程的虚拟地址空间，那么 CPU 会捕捉到这次越界访问，并且拒绝此次访问。同时，CPU 会发送异常错误给操作系统，由操作系统处理这些异常情况，这就是我们常说的缺页异常。另外，对于进程来说，它不再需要关心物理地址的布局，它访问的地址位于虚拟地址空间，只需要按照原来的地址编写程序并访问地址，程序就可以无缝地迁移到不同的系统上。

基于分段机制解决问题的思路可以总结为增加虚拟内存（virtual memory）。进程在运行时看到的地址是虚拟地址，然后需要通过 CPU 提供的地址映射方法把虚拟地址转换成实际的物理地址。当多个进程在运行时，这种方法就可以保证每个进程的虚拟内存空间是相互隔离的，操作系统只需要维护虚拟地址到物理地址的映射关系。

虽然分段机制有了比较明显的改进，但是内存使用效率依然比较低。分段机制对虚拟内存到物理内存的映射通常采用粗粒度的块，即以代码段、数据段以及堆来分成几个段。当物理内存不足时，以段为单位换出到磁盘，因此会有大量的磁盘访问，进而影响系统性能。站在进程的角度看，以段为单位进行换出和换入的方法还不太合理。在运行进程时，根据局部性原理，只有一部分数据一直在使用。若把那些不常用的数据交换出磁盘，就可以节省很多系统带宽，而把那些常用的数据驻留在物理内存中也可以得到比较好的性能。另外，大小不一的段很容易引起物理内存的外碎片化（external fragmentation）问题，即引入了很多离散的空洞，导致难以分配新的段空间，从而增加额外的管理负担。因此，人们在分段机制之后又发明了一种新的机制，这就是分页（paging）机制。

10.1.4 分页机制

程序运行所需要的内存往往大于实际物理内存，采用分段机制会把程序的段交换到交换磁盘，这不仅费时费力，而且效率很低。后来出现了分页机制，分页机制引入了虚拟存储器的概念。分页机制的核心思想是把程序中一部分不使用的内存存放到交换磁盘中，而把程序正在使用的内存继续保留在物理内存中。因此，当一个程序运行在虚拟存储器空间中时，它的寻址范围由处理器的位宽决定，比如，32 位处理器的位宽是 32 位，地址范围是 0～4 GB。假设 64 位处理器的虚拟地址位宽是 48 位，程序员可以访问 0x0000 0000 0000 0000～0x0000 FFFF FFFF FFFF 以及 0xFFFF 0000 0000 0000～0xFFFF FFFF FFFF FFFF 这两段空间。在使能了分页机制的处理器中，我们通常把处理器能寻址的地址空间称为虚拟地址空间。和虚拟存储器对应的是物理存储器（physical memory），它对应系统中使用的物理存储设备的地址空间。在没有使能分页机制的系统中，处理器直接寻址物理地址，把物理地址发送到内存控制器；而在使能了分页机制的系统中，处理器直接寻址虚拟地址，这个地址不会直接发给内存控制器，而先发送给 MMU。

MMU 负责虚拟地址到物理地址的转换和翻译工作。在虚拟地址空间里，可按照固定大小来分页，典型的页面粒度为 4 KB，现代处理器都支持大粒度的页面，如 16 KB、64 KB 甚至 2 MB 的巨页。而在物理内存中，空间也分成和虚拟地址空间大小相同的块，称为页帧（page frame）。程序可以在虚拟地址空间里任意分配虚拟内存，但只有当程序需要访问或修改虚拟内存时，操作系统才会为其分配物理页面，这个过程叫作请求调页（demand page）或者缺页异常。

虚拟地址[31:0]可以分成两部分：一部分是虚拟页面内的偏移量，以 4 KB 页为例，VA[11:0]是虚拟页面偏移量；另一部分用来寻找属于哪个页，这称为虚拟页帧号（Virtual Page frame Number，VPN）。物理地址中，PA[11:0]表示物理页帧的偏移量，剩余部分表示物理页帧号（Physical page Frame Number，PFN）。MMU 的工作内容就是把虚拟页帧号转换成物理页帧号。处理器通常使用一张表来存储 VPN 到 PFN 的映射关系，这张表称为页表（Page Table，PT）。页表中的每一项称为页表项（Page Table Entry，PTE）。若将整张页表存放在寄存器中，则会占用很多硬件资源，因此通常的做法是把页表放在主内存里，由页表基地址寄存器指向这种页表的起始地址。如图 10.3 所示，处理器发出的地址是虚拟地址，通过 MMU 查询页表，处理器便得到了物理地址，最后把物理地址发送给内存控制器。

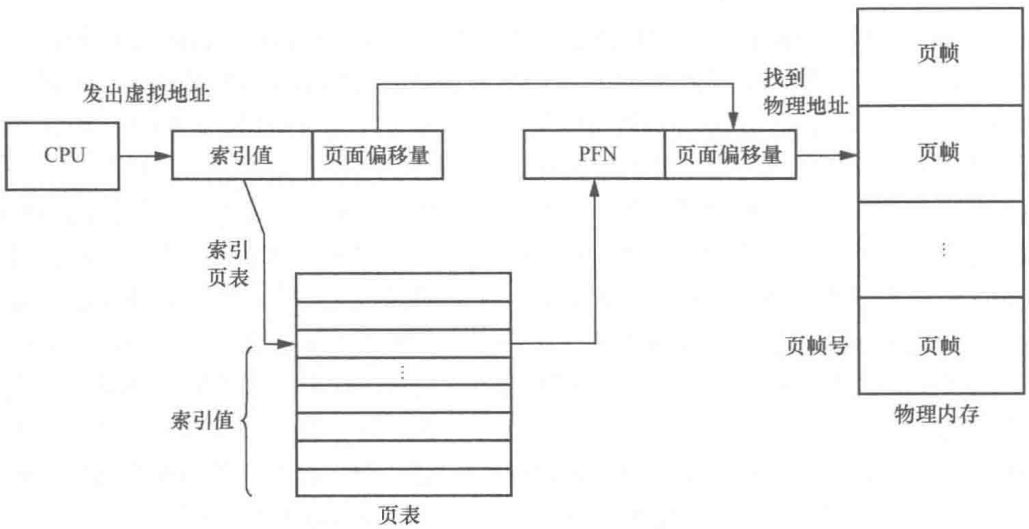


图 10.3　页表查询过程

下面以最简单的一级页表为例。如图 10.4 所示，处理器采用一级页表，虚拟地址空间的位宽是 32 位，寻址范围是 0～4 GB，物理地址空间的位宽也是 32 位，最多支持 4 GB 物理内存。另外，页面的大小是 4 KB。为了能映射整个 4 GB 地址空间，需要 4 GB/4 KB=2^{20} 个页表项，每个页表项占用 4 字节，需要 4 MB 大小的物理内存来存放这张页表。VA[11:0]是页面偏移量，VA[31:12]是 VPN，可作为索引值在页表中查询页表项。页表类似于数组，VPN 类似于数组的下标，用于查找数组中对应的成员。页表项包含两部分：一部分是 PFN，它代表页面在物理内存中的帧号（即页帧号），页帧号加上 VA[11:0]页内偏移量就组成了最终物理地址（PA）；另一部分是页表项的属性，比如图 10.4 中的 V 表示有效位。若有效位为 1，表示这个页表项对应的物理页面在物理内存中，处理器可以访问这个页面的内容；若有效位为 0，表示这个页表项对应的物理页面不在内存中，可能在交换磁盘中。如果访问该页面，那么操作系统会触发缺页异常，可在缺页异常中处理这种情况。当然，实际的处理器中还有很多其他的属性位，如描述这个页面是否为脏页，是否可读、可写等。

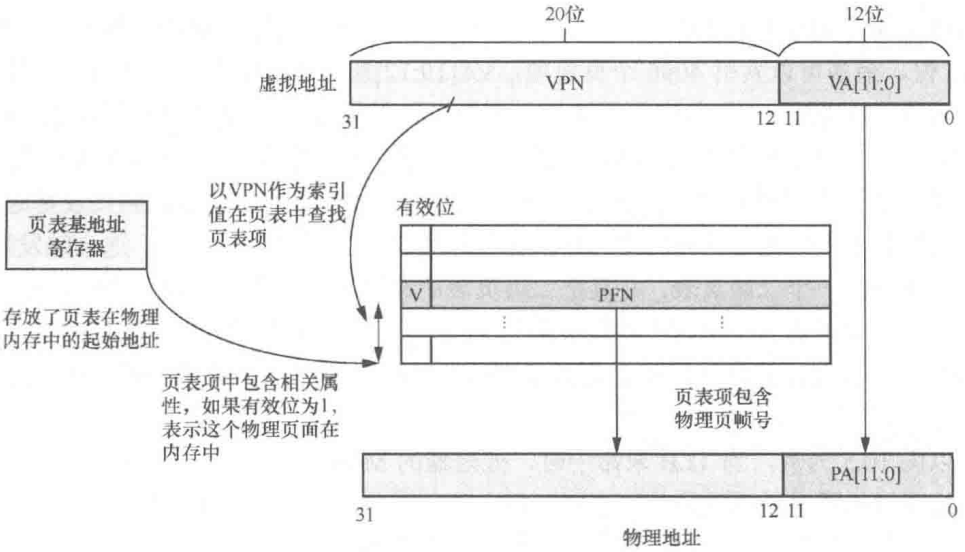


图 10.4 一级页表

通常操作系统支持多进程，进程调度器会在合适的时间（比如，当进程 A 使用完时间片时）从进程 A 切换到进程 B。另外，分页机制也让每个进程都感觉到自己拥有了全部的虚拟地址空间。为此，每个进程拥有一套属于自己的页表，在切换进程时需要切换页表基地址。比如，对于上面的一级页表，每个进程需要为其分配 4 MB 的连续物理内存，这是无法接受的，因为这太浪费内存了。为此，人们设计了多级页表来减少页表占用的内存空间。如图 10.5 所示，把页表分成一级页表和二级页表，页表基地址寄存器指向一级页表的基地址，一级页表的页表项里存放了一个指针，指向二级页表的基地址。当处理器执行程序时，只需要把一级页表加载到内存中，并不需要把所有的二级页表都加载到内存中，而根据物理内存的分配和映射情况逐步创建和分配二级页表。这样做有两个原因：一是程序不会马上使用完所有的物理内存；二是对于 32 位系统来说，通常系统配置的物理内存小于 4 GB，如仅有 512 MB 内存等。

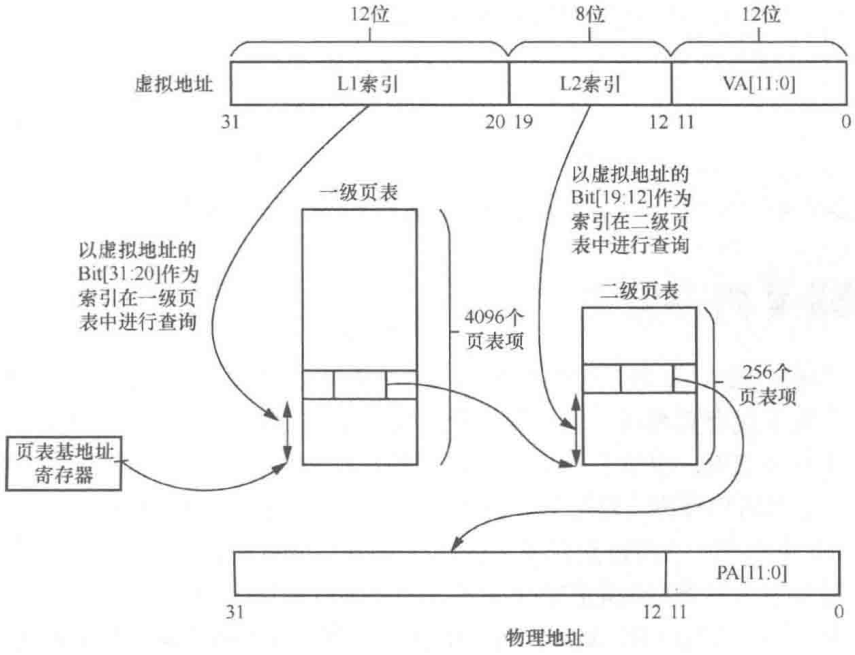


图 10.5 二级页表查询过程

图 10.5 展示了通用处理器体系结构的二级页表查询过程，VA[31:20]被用作一级页表的索引，一共有 12 位，最多可以索引 4096 个页表项；VA[19:12]被用作二级页表的索引，一共有 8 位，最多可以索引 256 个页表项。当操作系统复制一个新的进程时，首先会创建一级页表，分配 16 KB 页面。在本场景中，一级页表有 4096 个页表项，每个页表项占 4 字节，因此一级页表一共有 16 KB。当操作系统准备让进程运行时，会设置一级页表在物理内存中的起始地址到页表基地址寄存器中。进程在执行过程中需要访问物理内存，因为一级页表的页表项是空的，这会触发缺页异常。在缺页异常里分配一个二级页表，并且把二级页表的起始地址填充到一级页表的相应页表项中。接着，分配一个物理页面，把这个物理页面的 PFN 填充到二级页表的对应页表项中，从而完成页表的填充。随着进程的执行，需要访问越来越多的物理内存，于是操作系统逐步地把页表填充并建立起来。

我们以图 10.5 为例，当 TLB 未命中时，处理器的 MMU 页表查询过程如下。

（1）处理器的页表基地址控制寄存器（每个处理器体系结构都有类似的页表基地址寄存器，在 RISC-V 体系结构中是 satp 寄存器）存放着一级页表的基地址。

（2）处理器以虚拟地址的 Bit[31:20]作为索引值，在一级页表中找到页表项，一级页表一共有 4096 个页表项。

（3）一级页表的页表项中存放二级页表的物理基地址。处理器使用虚拟地址的 Bit[19:12]作为索引值，在二级页表中找到相应的页表项，二级页表有 256 个页表项。

（4）二级页表的页表项里存放了 4 KB 大小页面的物理基地址。这样，处理器就完成了页表的查询和翻译工作。

图 10.6 展示了 4 KB 映射的一级页表的页表项。Bit[31:10]指向二级页表的物理基地址。

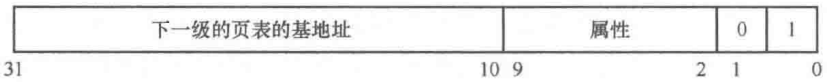


图 10.6　4 KB 映射的一级页表的页表项

图 10.7 展示了 4 KB 映射的二级页表的页表项。Bit[31:12]指向 4 KB 大小页面的页帧号，页帧号加上低 12 位地址组成最终的物理基地址。

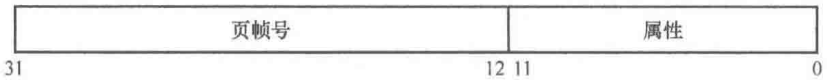


图 10.7　4 KB 映射的二级页表的页表项

对于 RISC-V 处理器来说，通常会使用三级或者四级页表，但是原理和二级页表是一样的。

10.2 RISC-V 内存管理

如图 10.8 所示，RISC-V 处理器内核的 MMU 包括 TLB 和页表遍历单元两个部件。TLB 是一个高速缓存，用于缓存页表转换的结果，从而缩短页表查询的时间。一个完整的页表翻译和查找的过程叫作页表查询。页表查询的过程由硬件自动完成，但是页表的维护由软件来完成。页表查询是一个较耗时的过程，理想的状态下，TLB 里应存放页表的相关信息。当 TLB 未命中时，MMU 才会查询页表，从而得到翻译后的物理地址，而页表通常存储在内存中。得到物理地址之后，首先需要查询该物理地址的内容是否在高速缓存中有最新的副本。如果没有，则说明高速缓存未命中，需要访问内存。MMU 的工作职责就是把输入的虚拟地址翻译成对应的物理地址以及相应的页表属性和内存访问权限等信息。另外，如果地址访问失败，那么会触发一个与

MMU 相关的缺页异常。

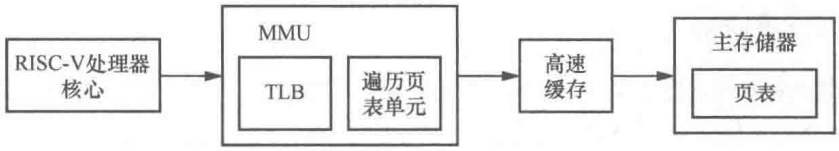


图 10.8 RISC-V 处理器的内存管理体系结构

对于多任务操作系统，每个进程都拥有独立的进程地址空间。这些进程地址空间在虚拟地址空间内是相互隔离的，但是在物理地址空间中可能映射到同一个物理页面。这些进程地址空间是如何映射到物理地址空间的呢？这就需要处理器的 MMU 提供页表映射和管理的功能。图 10.9 所示为进程地址空间和物理地址空间的映射关系，左边是进程地址空间，右边是物理地址空间。进程地址空间又分成内核空间（kernel space）和用户空间（user space）。无论是内核空间还是用户空间，都可以通过处理器提供的页表机制映射到实际的物理地址。

在 SMP（Symmetric MultiProcessor，对称多处理器）系统中，每个处理器内核内置了 MMU 和 TLB 硬件单元。如图 10.10 所示，CPU0 和 CPU1 共享物理内存，而页表存储在物理内存中。CPU0 和 CPU1 中的 MMU 与 TLB 硬件单元也共享同一份页表。

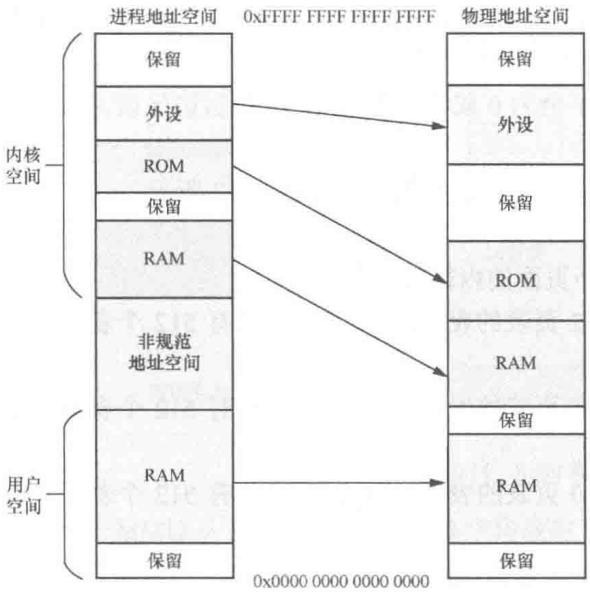


图 10.9 进程地址空间和物理地址空间的映射关系

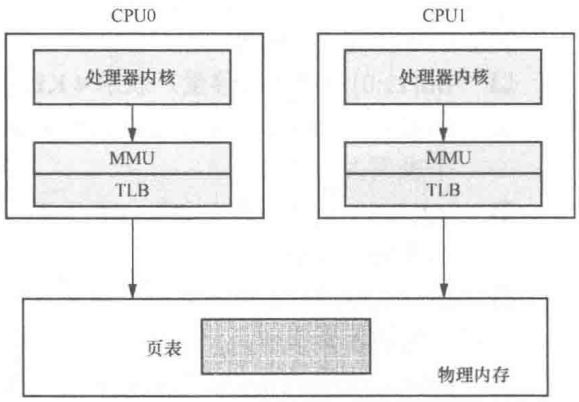


图 10.10 SMP 系统与 MMU

10.2.1 页表分类

在 RISC-V 体系结构中，根据处理器的虚拟地址位宽，提供了多种地址转换机制。

- Sv32：仅支持 32 位 RSIC-V 处理器，是一个二级页表结构，支持 32 位虚拟地址转换。
- Sv39：支持 64 位 RSIC-V 处理器（其中虚拟地址的低 39 位用作页表索引），是一个三级页表结构，支持 39 位虚拟地址转换。
- Sv48：支持 64 位 RSIC-V 处理器（其中虚拟地址的低 48 位用作页表索引），是一个四级页表结构，支持 48 位虚拟地址转换。

目前 RISC-V 体系结构通常支持 4 KB 大小的页面（page）粒度，也支持 2 MB、1 GB 大小

的块（block）粒度，也称为大页（huge page）。

10.2.2　Sv39 页表映射

目前大多数 64 位的 RISC-V 处理器采用 Sv39 模式的地址转换机制。Sv39 模式表示 64 位的虚拟地址中只有低 39 位用于页表索引，剩余的高位必须和第 38 位相等，否则处理器会触发缺页异常。在 Sv39 模式下，64 位的虚拟地址中只有低 39 位为有效位，即用于页表地址转换，并最终映射到 56 位的物理地址上。当虚拟地址的第 38 位为 0 或者 1 时，整个虚拟地址空间被划分成 3 个区域，如图 10.11 所示。

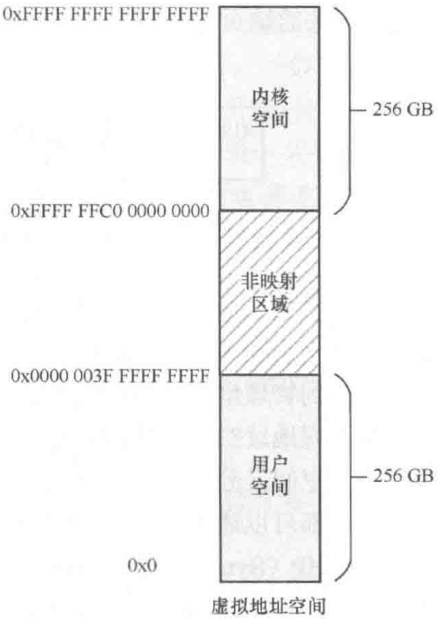


图 10.11　虚拟地址空间的划分

- 当第 38 位为 0 时，剩余的高位也为 0，这样组成了低位的虚拟地址空间，它位于 0x0000 0000 0000 0000 到 0x0000 003F FFFF FFFF，大小为 256 GB。操作系统通常把该虚拟地址区间用作用户空间。
- 当第 38 位为 1 时，剩余的高位也为 1，这样组成了高位的虚拟地址空间，它位于 0xFFFF FFC0 0000 0000 到 0xFFFF FFFF FFFF FFFF，大小为 256 GB。操作系统通常把该虚拟地址区间用作内核空间。
- 中间部分为非映射区域，即 Bit[63:38]不全为 0 或者不全为 1。处理器访问该区域会触发缺页异常。

Sv39 模式采用三级页表映射模式，64 位虚拟地址分成 5 部分，如图 10.12 所示。

其中，每一部分的说明如下。

- Bit[11:0]为页面偏移量，表示 4 KB 大小页面的内部偏移量。
- Bit[20:12]为 L2 页表索引，用于索引 L2 页表的表项。L2 页表一共有 512 个表项，每个表项占 8 字节。
- Bit[29:21]为 L1 页表索引，用于索引 L1 页表的表项。L1 页表一共有 512 个表项，每个表项占 8 字节。
- Bit[38:30]为 L0 页表索引，用于索引 L0 页表的表项。L0 页表一共有 512 个表项，每个表项占 8 字节。
- Bit[63:39]为保留位，但是这些位必须与第 38 位保持相同。

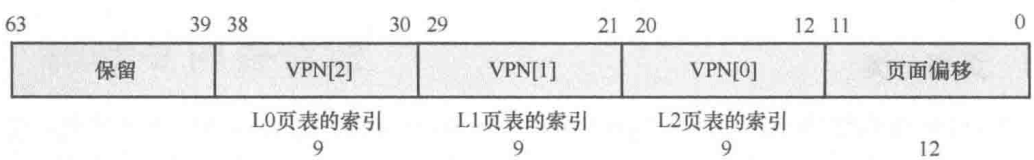


图 10.12　虚拟地址的划分

图 10.13 展示了三级页表的映射，satp 寄存器中的 PPN 字段存储了页表基地址的页帧号，它指向 L0 页表的基地址。在 L0 页表中有许多页表项，页表项通常分成页表类型页表项和块类型页表项。页表类型页表项包含下一级页表的基地址，用来指向下一级页表，而块类型页表项包含大块物理内存的基地址，如 1 GB、2 MB 等大块物理内存。最后一级页表由页表项组成，每个页表项指向一个物理页面，物理页面大小可以是 4 KB。

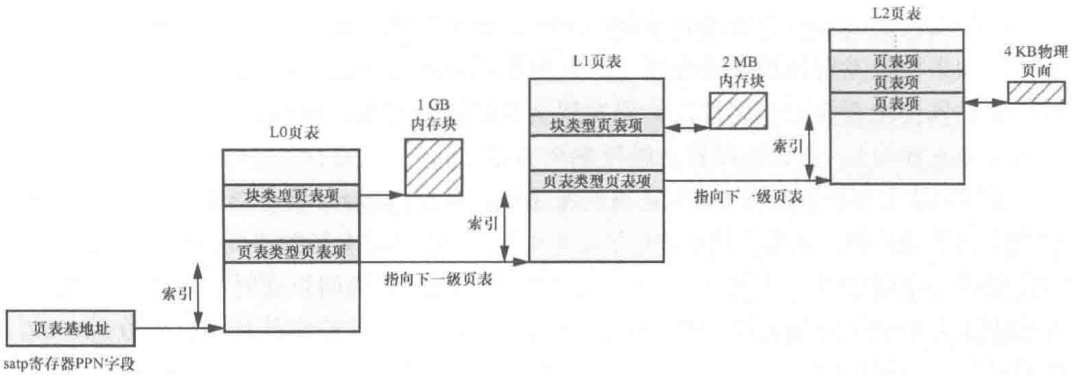


图 10.13　三级页表的映射

如图 10.14 所示，当 TLB 未命中时，MMU 查询页表的过程如下。

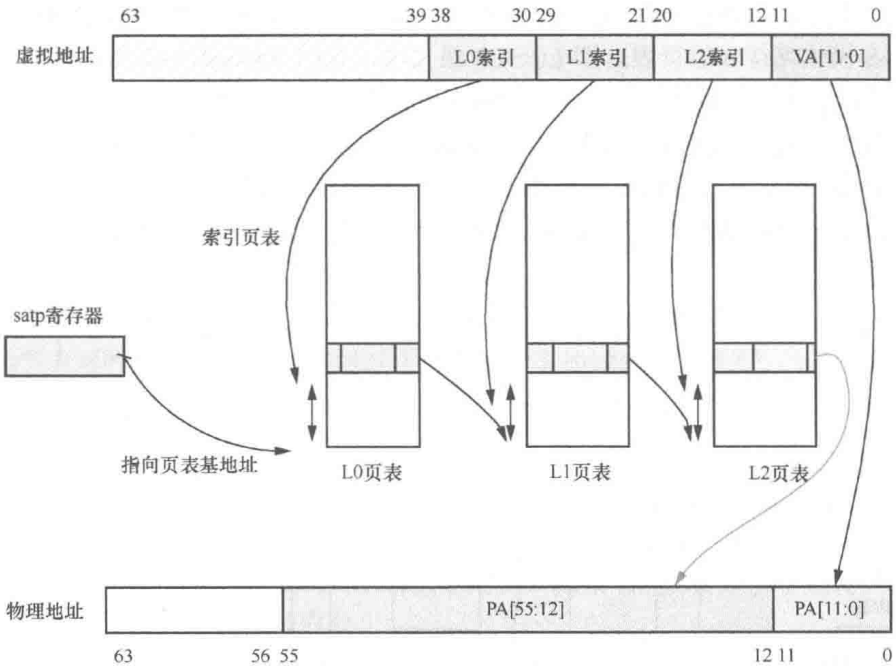


图 10.14　Sv39 模式的页表映射过程

（1）MMU 从 satp 寄存器的 PPN 字段获取页表基地址的页帧号，从而得到物理地址。

（2）MMU 将 VA[38:30]作为 L0 索引，根据 L0 索引在 L0 页表中查找表项，L0 页表有 512 个表项。

（3）在 L0 页表中找到的表项存放着 L1 页表的物理基地址的页帧号，MMU 得到了 L1 页表的基地址。

（4）MMU 将 VA[29:21]作为 L1 索引，根据 L1 索引在 L1 页表中查找表项，L1 页表有 512 个表项。

（5）在 L1 页表中找到的表项存放着 L2 页表的物理基地址的页帧号，MMU 得到了 L2 页表基地址。

（6）MMU 以 VA[20:12]作为 L2 索引，在 L2 页表中找到相应的表项，L2 页表有 512 个表项。

（7）L2 页表的表项里存放着 4 KB 页面的页帧号，加上 VA[11:0]，二者就构成了新的物理地址，因此 MMU 就完成了页表的查询和翻译工作。

上述页表查询过程是理想状态下的查询过程。实际上，处理器会做如下检查。

（1）处理器在访问各级页表项时会使用 PMA 或者 PMP（见 10.3 节）机制做与内存属性相关的检查。如果检查发现违规，处理器会触发内存访问异常（access-fault exception）。

（2）处理器会检查页表项属性，如果发现页表项是无效的，例如，V=0 或者保留的访问权限（如 R=0 && W=1），那么处理器会触发缺页异常。

（3）处理器需要根据子叶页表项描述符的属性以及 mstatus/status 寄存器中相关的字段（如 SUM 以及 MXR）做权限检查，如果不满足请求的访问权限，则处理器会触发缺页异常，详见 10.2.5 节。

假设处理器采用软件方式处理 A 和 D 标志位，当处理器访问页面时，如果该页面对应的子叶页表项描述符中的访问标志位为 0（A=0）或者该访问是存储操作并且脏位为 0（D=0），则会触发缺页异常，详见 10.2.5 节。

10.2.3　Sv48 页表映射

Sv48 模式表示 64 位的虚拟地址中只有低 48 位地址用于页表索引，剩余的高位必须和第 47 位相等。Sv48 模式支持 4 级页表，即 L0～L3 级页表，支持 4 KB 大小页面。

如图 10.15 所示，当 TLB 未命中时，MMU 查询页表的过程如下。

（1）MMU 从 satp（supervisor address translation and protection，监管地址转换与保护）寄存器的 PPN 字段获取 L0 页表基地址的页帧号，从而得到物理地址。

（2）MMU 以 VA[47:39]作为 L0 索引，根据 L0 索引在 L0 页表中查找页表项，L0 页表有 512 个页表项。

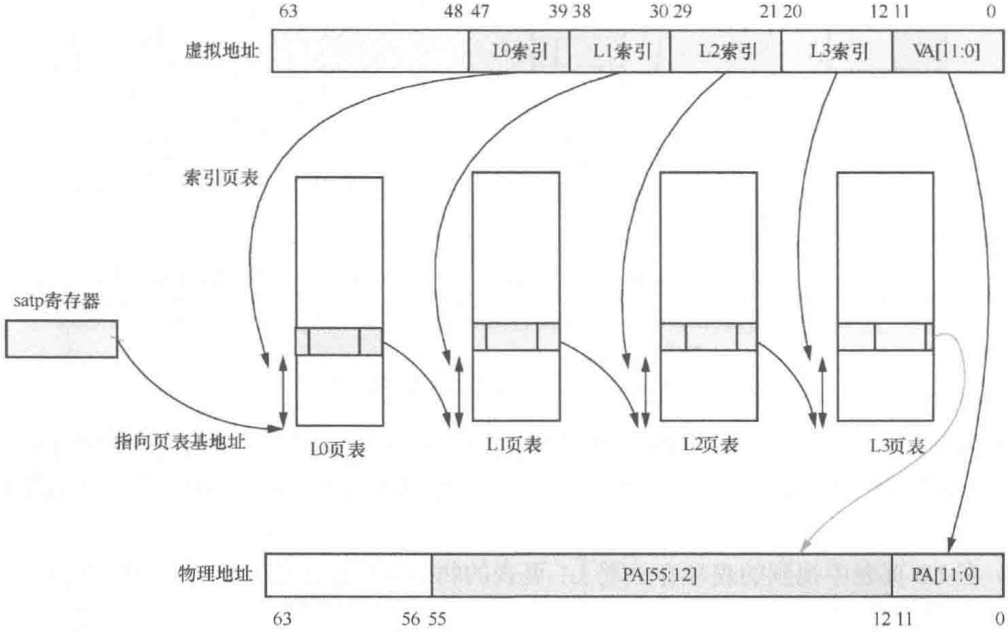


图 10.15　Sv48 模式的页表映射过程

（3）在 L0 页表中找到的表项存放着 L1 页表的物理基地址的页帧号，MMU 得到了 L1 页表的基地址。

（4）MMU 以 VA[38:30]作为 L1 索引，根据 L1 索引在 L1 页表中查找页表项，L1 页表有 512 个页表项。

（5）在 L1 页表中找到的表项存放着 L2 页表的物理基地址的页帧号，MMU 得到了 L2 页表基地址。

（6）MMU 以 VA[29:21]作为 L2 索引，在 L2 页表中找到相应的页表项，L2 页表有 512 个页表项。

（7）在 L2 页表中找到的表项存放着 L3 页表的物理基地址的页帧号，MMU 得到了 L3 页表基地址。

（8）MMU 以 VA[20:12]作为 L3 索引，在 L3 页表中找到相应的页表项，L3 页表有 512 个页表项。

（9）L3 页表的页表项里存放着 4 KB 页面的页帧号，加上 VA[11:0]，二者就构成了新的物理地址，因此 MMU 就完成了页表的查询和翻译工作。

10.2.4 页表项描述符

Sv39 模式以及 Sv48 模式包含多级页表，每一级页表都由多个页表项组成，我们把它们称为页表项描述符，每个页表项描述符占 8 字节。这些页表项描述符的格式都一样，但是内容不完全一样。

1. 非子叶页表项描述符

从图 10.14 和图 10.15 可知，Sv39 模式以及 Sv48 模式包含多级页表，如果页表项描述符中的 Bit[3:1]都为 0，该页表项描述符包含指向下一级页表基地址的页帧号，我们把该页表称为非子叶页表（non-leaf page table）。我们使用一个描述符来描述它对应的页表项，称为非子叶页表项描述符。页表项描述符中的 Bit[1]为 R 字段，Bit[2]为 W 字段，Bit[3]为 X 字段。

2. 子叶页表项描述符

在 Sv39 和 Sv48 模式下，如果页表项描述符中的 Bit[3:1]不为 0，则该页表项描述符包含指向最终物理地址的字段，我们把该页表称为子叶页表（leaf page table）。我们使用一个描述符来描述它的页表项，称为子叶页表项描述符。

子叶页表可以是末级的页表，也可以是中间某一个级页表。如果中间某一级页表为子叶页表，那么该页表项为块映射页表项描述符，用来描述大块连续物理内存，如 2 MB 或者 1 GB 的物理内存。如果末级页表为子叶页表，那么该页表项为页映射页表项描述符，该页表项描述符指向 4 KB 大小的物理页面。

以 Sv39 为例，所有级别的页表都可以是子叶页表。如图 10.14 所示，如果末级页表（L2 页表）为子叶页表，那么该页表项描述符指向 4 KB 大小的物理页面。如果 L0 页表为子叶页表，那么该页表项描述符指向 1 GB 的大块连续物理内存。如果 L1 页表为子叶页表，那么该页表项描述符指向 2 MB 的大块连续物理内存。

同理，在 Sv48 模式下，所有级别的页表都可以是子叶页表。如果末级页表（L2 页表）为子叶页表，那么该页表项描述符指向 4 KB 大小的物理页面。如果其他级别的页表为子叶页表，则对应的页表项描述符可以分别指向 512 GB、1 GB 以及 2 MB 的大块连续物理内存。

3. 页表项分类

根据类型，页表项分成 3 类：一是无效的页表项；二是页表（table）类型的页表项，三是子叶页表类型的页表项。

对于无效的页表项，当页表项描述符 Bit[0]为 0 时，表示无效的描述符；当 Bit[0]为 1 时，表示有效的描述符，如图 10.16 所示。

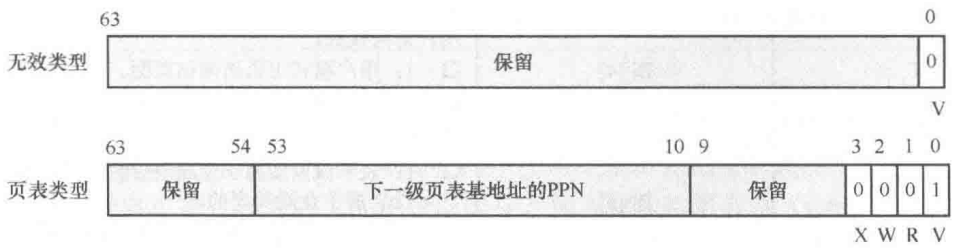


图 10.16 无效类型与页表类型的页表项描述符

对于页表类型的页表项，当页表项描述符 Bit[3:1]都为 0 时，表示指向下一级页表的描述符，该描述符包含指向下一级页表基地址的 PPN（Physical Page Number），如图 10.16 所示。

对于子叶页表类型的页表项，当页表项描述符 Bit[3:1]不为 0 时，表示子叶页表类型的页表项描述符，该描述符包含最终物理地址以及页面属性。页表项描述符可以指向 4 KB 大小的页面，也可以指向块映射的大块内存。块映射通常用来描述大块连续物理内存，如 2 MB 或者 1 GB 大小的物理内存。

子叶页表类型的页表项分成两种：一种是页映射页表项描述符，另一种是块映射页表项描述符。子叶页类型的页表项描述符如图 10.17 所示，它由三部分组成。

- 低位属性：由 Bit[9:0]组成的低位属性，如表 10.1 所示。
- 页帧号：由 Bit[53:10]组成的物理页面的页帧号。如果子叶页表类型的描述符是页映射页表项描述符，即该子叶页表是末级的页表（如图 10.14 中的 L2 页表），则物理页面是 4 KB 的物理页面；如果是块映射页表项描述符，即该子叶页表可以是中间级别的页表（如图 10.14 中的 L0～L1 页表），则物理页面可以是 2 MB 或者 1 GB 的大块连续物理内存。
- 高位属性：由 Bit[63:54]组成的高位属性。

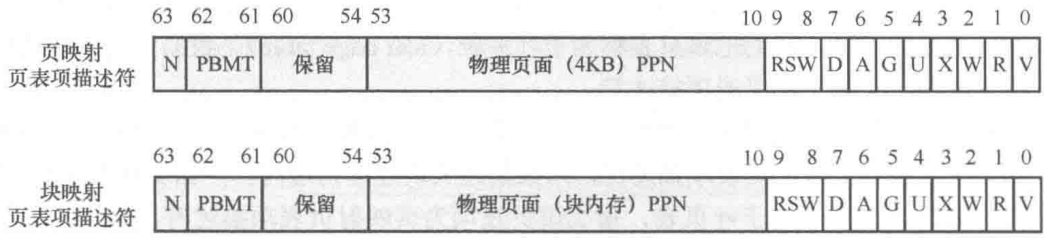


图 10.17　子叶页类型的页表项描述符

页表项描述符中的低位属性如表 10.1 所示。

表 10.1　　页表项描述符中的低位属性

名称	位	描述
V	Bit[0]	有效位。 ❑ 1：表示页表项有效。 ❑ 0：表示页表项无效
R	Bit[1]	可读属性。 ❑ 1：表示页面内容具有可读属性。 ❑ 0：表示页面内容不具有可读属性
W	Bit[2]	可写属性。 ❑ 1：表示页面内容具有可写属性。 ❑ 0：表示页面内容不具有可写属性
X	Bit[3]	可执行属性。 ❑ 1：表示页面内容具有可执行属性。 ❑ 0：表示页面内容不具有可执行属性
U	Bit[4]	用户访问模式。 ❑ 1：用户模式可以访问该页面。 ❑ 0：用户模式不能访问该页面
G	Bit[5]	全局映射，常用于 TLB，详见 13.3 节。 ❑ 1：表示该页面属于全局映射的页面，该页面对应的 TLB 属于全局类型的。 ❑ 0：表示该页面属于非全局映射的页面，该页面对应的 TLB 属于进程独有的，使用 ASID 来标识

续表

名称	位	描述
A	Bit[6]	访问标志位。 ❑ 1：表示处理器访问过该页面。 ❑ 0：表示处理器没有访问过该页面
D	Bit[7]	脏位。 ❑ 1：表示页面被修改过。 ❑ 0：表示页面是干净的
RSW	Bit[9:8]	预留给系统管理员使用

页表项描述符中的高位属性如表 10.2 所示，这个属于 RISC-V 体系结构中的“Svpbmt”扩展，将来用于替代物理内存属性（Physical Memory Attribute，PMA）机制。

表 10.2　　页表项高位属性

名称	位	描述
PBMT	Bit[62:61]	用来表示映射页面的内存属性。 ❑ 0：无。 ❑ 1：表示普通内存，关闭高速缓存，支持弱一致性内存模型。 ❑ 2：表示 I/O 内存，关闭高速缓存，支持强一致性内存模型。 ❑ 3：保留
N	Bit[63]	连续块表项

关于页表项属性，有两点需要注意。

- ❑ 由于非叶页表类型的页表项中的 A、D 以及 U 标志位预留给未来使用，因此软件需要把这些位设置为 0。
- ❑ 在使能了虚拟内存管理机制与 A 扩展的处理器中，LR/SC 指令的保留集（reservation set）必须在一个页面中。关于 LR/SC 保留集，见 14.2 节。

10.2.5 页表属性

本节介绍页表项中常见的属性。

1. 访问权限

页表项属性通过 R、W 以及 X 字段来控制处理器对这个页面的访问，例如，指定页面是否具有可读、可写权限等。它们可以组合在一起来使用，如表 10.3 所示。

表 10.3　　指定访问权限的字段

X 字段	W 字段	R 字段	说明
0	0	0	页表项指向下一级页表项描述符
0	0	1	只读属性页面
0	1	0	保留
0	1	1	可读、可写页面
1	0	0	只可执行的页面
1	0	1	可读、可执行页面
1	1	0	保留
1	1	1	可读、可写、可执行页面

在没有相应权限的页面中进行读、写或者执行代码等操作会触发缺页异常。

- ❑ 如果在没有可执行权限的页面中预取指令，触发预取缺页异常（fetch page fault）。
- ❑ 如果在没有读权限的页面加载数据，触发加载缺页异常（load page fault）。

- 如果在没有写权限的页面里写入数据，触发存储缺页异常（store page fault）。

另外，当页表项属性中的 U 字段为 1 时，表示用户态程序可以访问该页面。如果 sstatus 寄存器中的 SUM 字段也设置为 1，那么 S 模式下的软件也可以访问该页面。不过，S 模式下的软件不应该执行页表属性中 U 字段为 1 的页面里的程序代码。

2. 访问标志位与脏标志位

页表项属性中有一个访问字段 A（access），用来指示页面是否被 CPU 访问过。

- 如果 A 字段为 1，表示页面已经被 CPU 访问过。
- 如果 A 字段为 0，表示页面还没有被 CPU 访问过。

页表项属性中的脏标志位（D）表示页面内容被写入或者修改过。目前 RISC-V 体系结构处理器使用软件或硬件方式来维护访问标志位和脏标志位。

若以软件方式更新 A 和 D 标志，有以下两种方法。

- 当 CPU 尝试访问页面并且该页面对应的 A 标志位为 0 时，会触发缺页异常，然后软件就可以设置 A 标志位为 1。
- 当 CPU 尝试修改或者写入页面并且 D 标志位为 0 时，会触发缺页异常，然后软件就可以设置 D 标志位为 1。

当采用硬件方式时，页表项（PTE）的更新必须是原子的，即 CPU 会原子地更新整个页表项，而不是仅仅更新某个标志位。另外，CPU 在更新 A 标志位时允许预测（speculation）执行，而 D 标志位的更新必须是精确的，即不允许预测执行，并且在本地 CPU 里是按照程序次序执行的。操作系统通常使用 A 和 D 标志位实现页面回收（page reclaim）。D 标志位更新的精确性会影响页面回收的正确性。

如果以硬件方式更新 A 和 D 标志位，方法如下。

- 当 CPU 尝试访问页面并且该页面的 A 标志位为 0 时，CPU 自动设置 A 标志位。
- 当 CPU 修改或者写入页面并且该页面的 D 标志位为 0 时，CPU 自动设置 D 标志位。

操作系统使用访问标志位有如下好处。

- 用来判断某个已经分配的页面是否被操作系统访问过。如果访问标志位为 0，说明这个页面没有被操作系统访问过。
- 用于实现操作系统中的页面回收机制。

3. 连续块页表项

RISC-V 体系结构在页表设计方面考虑了 TLB 的优化，即利用一个 TLB 项来完成多个连续的虚拟地址到物理地址的映射。子叶页表项描述符中的 N 字段就用来实现 TLB 优化功能。

使用连续块页表项位的条件如下。

- 连续的页面必须有相同的内存属性，即子叶页表项描述符中 Bit[5:0]必须相同。
- 必须有 2^N 个连续的页面。

10.2.6　与地址转换相关的寄存器

与地址转换相关的寄存器主要是 satp 寄存器，如图 10.18 所示。

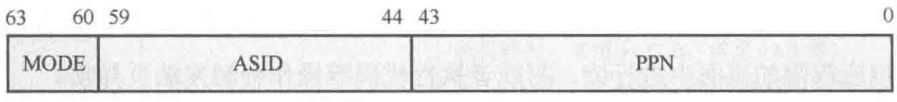


图 10.18　satp 寄存器

satp 寄存器中一共有 3 个字段。

- PPN 字段：存储了 L0 页表基地址的页帧号。
- ASID 字段：进程地址空间标识符（Address Space IDentifier，ASID），用于优化 TLB。
- MODE 字段：用来选择地址转换的模式。对于 32 位 RISC-V 处理器，MODE 字段如表 10.4 所示。对于 64 位 RISC-V 处理器，MODE 字段如表 10.5 所示。

表 10.4　32 位 RISC-V 处理器的 MODE 字段

MODE 字段	值	说明
Bare	0	没有实现地址转换功能
Sv32	1	实现 32 位虚拟地址转换（分页机制）

表 10.5　64 位 RISC-V 处理器的 MODE 字段

MODE 字段	值	说明
Bare	0	没有实现地址转换功能
保留	1～7 的整数	保留
Sv39	8	实现 39 位虚拟地址转换（分页机制）
Sv48	9	实现 48 位虚拟地址转换（分页机制）
Sv57	10	保留，用于将来实现 57 位虚拟地址转换（分页机制）
Sv64	11	保留，用于将来实现 64 位虚拟地址转换（分页机制）

当 MODE 字段设置为 Bare 时，表示没有实现地址转换功能以及保护功能，处理器访问的地址为物理地址。在 64 位 RISC-V 处理器中，当 MODE 字段为 8 时，表示使能 39 位虚拟地址转换机制。

10.3 物理内存属性与物理内存保护

RISC-V 体系结构提供两种机制来对物理内存访问进行检查与保护，它们分别是物理内存属性（Physical Memory Attributes，PMA）和物理内存保护（Physical Memory Protection，PMP）。在 RISC-V 处理器中，这两种机制可以同时发挥作用。

10.3.1 物理内存属性

在一个完整的系统中，系统内存映射包含各种不同访问属性的地址空间，例如，有些用于普通读写的内存空间，有些用于内存映射输入/输出（Memory Mapped Input/Output，MMIO）的寄存器空间，有些可能不支持原子操作，有些可能不支持缓存一致性，有些支持不同的内存模型。在 RISC-V 体系结构中，使用 PMA 来描述内存映射中的每个地址区域访问的属性。这些属性包含访问类型（如是否可执行、可读或可写），以及与访问相关的其他可选属性，例如，支持访问的内存单元的大小、对齐、原子操作和可缓存性等。

PMA 一般是在芯片设计阶段就固定下来的，有些（例如，连接到不同的芯片片选或者总线等）则在硬件开发板设计阶段固定下来。在系统执行阶段很少修改它。在 RISC-V 处理器中通常实现了一个 PMA 检测器，当 ITLB、DTLB 以及页表遍历单元获得物理地址之后，PMA 检测器会做物理地址权限和属性检查。检查到违规后，触发指令/加载/存储访问异常。

与页表属性不同，系统内存区域的 PMA 通常是固定的，或者只能在特定于平台的控制寄存器中修改，不过大部分 RISC-V 处理器不具备修改 PMA 的能力。PMA 取决于不同处理器的设计，本节以 SiFive 的 U74 处理器为例，它采用固定 PMA 的方式，支持的 PMA 内存端口如表 10.6 所示。

U74 处理器的部分内存映射如表 10.7 所示，详细的内存映射见“SiFive U74-MC Core Complex Manual”文档。

表 10.6　　U74 处理器支持的 PMA 内存端口

内存端口	访问权限	支持属性
普通内存端口	可读、可写、可执行	支持原子内存操作和 LR/SC 指令、数据高速缓存、指令高速缓存以及指令预测
外设端口	可读、可写、可执行	支持原子内存操作、指令高速缓存
系统端口	可读、可写、可执行	指令高速缓存

表 10.7　　U74 处理器的部分内存映射

起始地址	结束地址	PMA	说明
0x200 0000	0x200 FFFF	RWA	CLINT
0x201 0000	0x0201 3FFF	RWA	L2 高速缓存控制器
0xC00 0000	0xFFF FFFF	RWA	PLIC
0x2000 0000	0x3FFF FFFF	RWXIA	外设端口（512 MB）
0x4000 0000	0x5FFF FFFF	RWXI	系统端口（512 MB）
0x8000 0000	0x10 7FFF FFFF	RWXIDA	普通内存端口（64 GB）

其中，R 表示可读，W 表示可写，X 表示可执行，A 表示原子内存操作，I 表示指令高速缓存，D 表示数据高速缓存。

10.3.2　物理内存保护

在 RISC-V 体系结构中，M 模式具有最高特权，拥有访问系统全部资源的权限。为了安全，默认情况下 S 模式和 U 模式对内存映射的任何区域没有可读、可写或可执行权限，除非配置 PMP 以允许它们访问。

如果处理器运行在 M 模式，只有当 L 字段被设置时才会去做 PMP 检查。当有效的处理器模式为 S 模式或者 U 模式时，会对每次的访问做 PMP 检查。另外，根据 mstatus 寄存器中的 MPRV 字段的值，在如下两种情况下，也需要做 PMP 检查。

- MPRV 为 0 并且处于 U 模式或者 S 模式下的指令预取和数据访问。
- 当 MPRV 为 1 并且 MPP 为 S/U 模式时，在任意处理器模式下，对数据访问都需要做 PMP 检查。

另外，在 MMU 遍历页表的过程中也会做 PMP 检查。

RISC-V 体系结构使用配置表项（8 位宽的字段，用 pmp*N*cfg 表示，*N* 表示表项数）以及对应的 64 位地址寄存器来描述一段地址空间的 PMP 配置属性。RISC-V 体系结构最多支持 64 个 PMP 表项。在芯片设计阶段，根据需求，实现 0 个、16 个以及 64 个 PMP 表项。PMP 配置寄存器如图 10.19 所示，每个寄存器包括 8 个 PMP 表项，如 pmp0cfg、pmp1cfg。

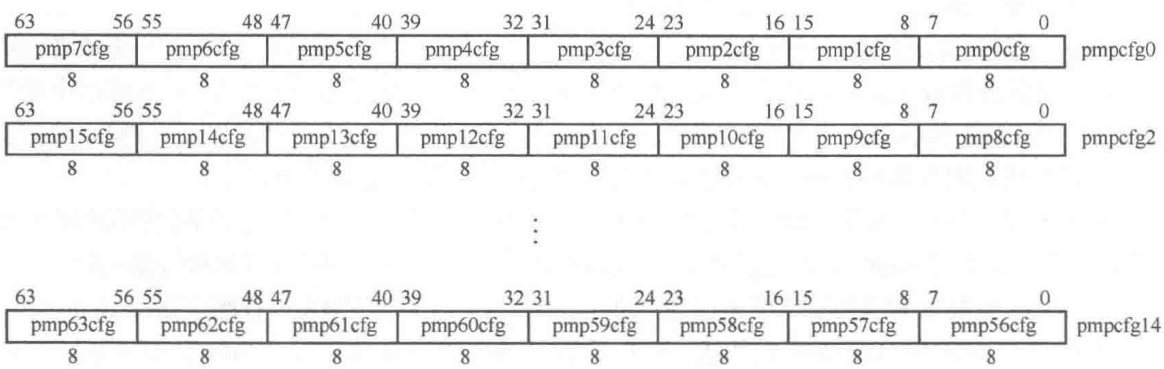


图 10.19　RV64 体系结构中的 PMP 配置寄存器

PMP 地址寄存器如图 10.20 所示，其中 Bit[53:0]用来存储地址的 Bit[55:2]。

PMP 表项如图 10.21 所示。每个表项的说明如表 10.8 所示。

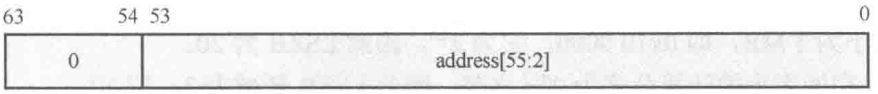


图 10.20 PMP 地址寄存器

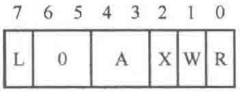


图 10.21 PMP 表项

表 10.8 每个表项的说明

字段	位	说明
R	Bit[0]	可读权限。 ❑ 0：没有读权限。 ❑ 1：具有读权限
W	Bit[1]	可写权限。 ❑ 0：没有写权限。 ❑ 1：具有写权限
X	Bit[2]	可执行权限。 ❑ 0：没有执行权限。 ❑ 1：具有执行权限
A	Bit[4:3]	地址匹配模式。 ❑ 0：表示关闭 PMP 表项对应的检查。 ❑ 1：TOR 模式。 ❑ 2：NA4 模式，即表示 PMP 表项对应的地址范围仅为 4 字节。 ❑ 3：NAPOT 模式
L	Bit[7]	锁定状态。 ❑ 0：表示 PMP 表项没有锁定，对 M 模式不起作用，仅对 S 模式和 U 模式起作用。 ❑ 1：表示 PMP 表项锁定，对所有处理器模式（包括 M 模式）都起作用

表中 TOR 模式的地址范围计算方法如下：由前一个 PMP 表项的地址寄存器代表的起始地址（假设为 pmpaddr($i-1$)）和当前 PMP 表项的地址寄存器代表的起始地址（假设为 pmpaddr$i$）共同决定，因此当前 PMP 表项代表的地址范围为

$$\text{pmpaddr}(i-1) \leqslant y < \text{pmpaddr}(i)$$

一种特殊情况是如果当前 PMP 表项是第 0 个表项并且 A 字段为 TOR，那么地址空间的下界被视为 0。此时，当前 PMP 表项代表的地址范围为

$$0 \leqslant y < \text{pmpaddr}(i)$$

NAPOT（Naturally Aligned Power-Of-Two region）模式采用 2^n 自然对齐的方式，其地址范围计算方式是从 PMP 地址寄存器第 0 位开始计算连续为 1 的个数 n，地址的长度为 2^{n+3} B。我们采用 LSZB（Least Significant Zero Bit，最低有效零位）来表示从第 0 位开始计算连续为 1 的个数。

如果 PMP 地址寄存器的值为 $yyyy\cdots yyy0$，即 LSZB 个数为 0，则该 PMP 表项所控制的地址空间为从 $yyyy\cdots yyy0$ 开始的 2^3 B，即 8 B。

如果 PMP 地址寄存器的值为 $yyyy\cdots yy01$，即 LSZB 个数为 1，则该 PMP 表项所控制的地址空间为从 yyyy…yy00 开始的 2^{1+3} B，即 16 B。

如果 PMP 地址寄存器的值为 $yy01\cdots 1111$，即 LSZB 个数为 n，则该 PMP 表项所控制的地址空间为从 $yy00\cdots 0000$ 开始的 2^{n+3} B。

【例 10-1】 假设一个地址区间的起始地址为 0x4000 0000，大小为 1 MB，这个地址区间的 PMP 属性为可读、可写、可执行，请计算 pmpaddr0 寄存器的值以及 pmpcfg0 寄存器的值（假设目前只有一个 PMP 表项）。

如下是计算步骤。

（1）由于 PMP 地址寄存器记录的是地址的 Bit[55:2]，因此地址需要右移 2 位，即 0x4000 0000 >> 2 = 0x1000 0000。

（2）地址区间的大小为 1 MB，即 0x10 0000，它为 2^{20}，因此 LSZB 为 20。

（3）由于 PMP 地址空间大小的计算公式为 2^{n+3} 字节，因此 LSZB 要减去 3，即 17。

（4）pmpaddr0 = 0x1000 0000 | 0b01 1111 1111 1111 1111 = 0x1001 FFFF。

（5）由于 PMP 属性为可读、可写、可执行，并且采用 NAPOT 模式，因此 pmpcfg0 寄存器的值为 0x1F。

以 0x4000 0000 为基地址，不同 PMP 地址大小对应的 PMP 地址寄存器的值的计算过程如表 10.9 所示。

表 10.9　　不同 PMP 地址大小对应的 PMP 地址寄存器的值的计算过程

基地址	PMP 地址的长度	LSZB	PMP 地址寄存器的值
0x4000 0000	8 B	0	(0x1000 0000 | 0B0)
0x4000 0000	32 B	2	(0x1000 0000 | 0B011)
0x4000 0000	4 KB	9	(0x1000 0000 | 0B01 1111 1111)
0x4000 0000	64 KB	13	(0x1000 0000 | 0B01 1111 1111 1111)
0x4000 0000	1 MB	17	(0x1000 0000 | 0B01 1111 1111 1111 1111)

如果同一个地址对应多个 PMP 表项，那么编号最小的 PMP 表项优先级最高。

【例 10-2】　假设同一个地址 0x8000 0000 在 pmp0cfg 和 pmp1cfg 表项中有重叠。

```
pmp0cfg: 0x8000 0000-0x8004 0000
pmp1cfg: 0x0000 0000-0xffff fffff
```

pmp0cfg 表项对应的属性是不可读和不可写，而 pmp1cfg 对应的属性为可读和可写。若运行在 S 模式的软件访问 0x8000 0000 地址，会触发加载/存储访问异常。

注意，PMP 只能在 M 模式下配置。另外，PMP 检查是基于地址范围的，例如，假设一个 PMP 表项设置了 4 字节地址区域[0xc–0xf]的访问属性并且这个 PMP 表项具有最高优先级，一个 8 字节的加载/存储指令访问 0x8 地址，则 PMP 检查会失败，触发加载/存储访问异常。

10.4　案例分析 10-1：在 BenOS 里实现恒等映射

恒等映射指的是虚拟地址映射到同等数值的物理地址上，即虚拟地址 = 物理地址，如图 10.22 所示。在操作系统的实现中，恒等映射是非常实用的技巧，特别是在系统初始化阶段使能 MMU 时。

从 OpenSBI 跳转到操作系统（如 Linux 内核）入口时，MMU 是关闭的。关闭 MMU 意味着不能利用高速缓存的性能。因此，我们在初始化的某个阶段需要把 MMU 打开并且使能数据高速缓存，以获得更高的性能。但是，如何打开 MMU？我们需要小心，否则会发生意想不到的问题。

在关闭 MMU 的情况下，处理器访问的地址都是物理地址。当 MMU 打开时，处理器访问的地址变成了虚拟地址。

现代处理器大多采用多级流水线体系结构，处理器会预取多条指令到流水线中。当打开 MMU 时，处理器已经预取了多条指令，并且这些指令是通过物理地址进行预取的。打开 MMU 的指令运行完之后，处理器的 MMU 立即生效，于是之前预取的指令会通过虚拟地址访问，再到 MMU 中查找对应的物理地址。因此，这是为了保证处理器在开启 MMU 前后可以连续取指令。

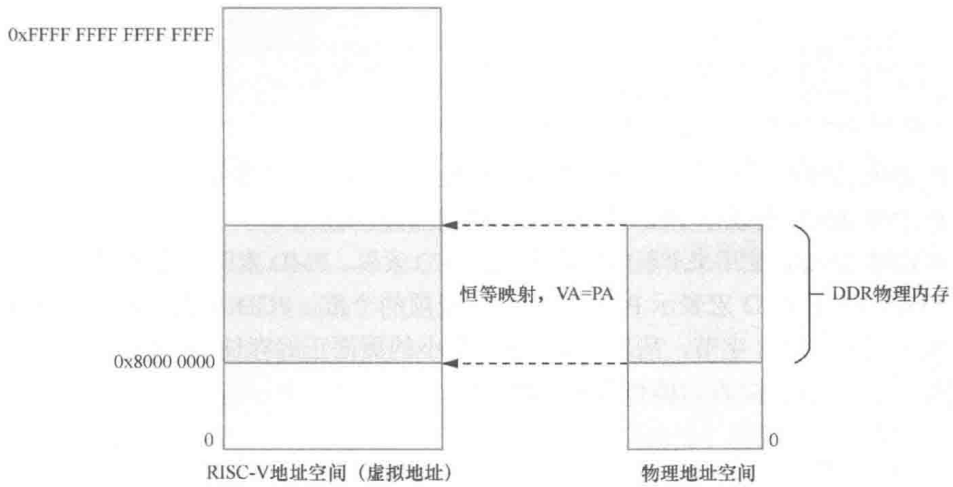


图 10.22 恒等映射

在本案例中，我们在 BenOS 上创建一个恒等映射，把低 8 MB 内存映射到虚拟地址空间里。在 QEMU Virt 平台上，BenOS 的代码段起始地址为 0x8020 0000。我们采用 4 KB 大小的页面和 Sv39 页表映射来创建这个恒等映射。

10.4.1 页表定义

我们采用与 Linux 内核类似的页表定义方式，即采用以下 3 级分页模型：

- 页全局目录（Page Global Directory，PGD）；
- 页中间目录（Page Middle Directory，PMD）；
- 页表（Page Table，PT）。

上述 3 级分页模型分别对应 RISC-V 体系结构中 Sv39 页表的 L0～L2 页表。上述 3 级分页模型在 64 位虚拟地址上的划分如图 10.23 所示。

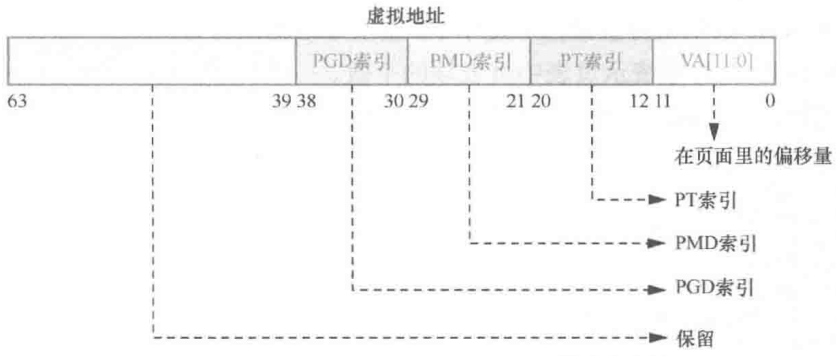


图 10.23 3 级分页模型在 64 位虚拟地址上的划分

64 位的虚拟地址分成如下几个部分。

- Bit[63:39]：保留。
- Bit[38:30]：表示 PGD 索引，即 L0 索引。
- Bit[29:21]：表示 PMD 索引，即 L1 索引。
- Bit[20:12]：表示 PT 索引，即 L2 索引。
- Bit[11:0]：表示页面内的偏移量。

从图 10.23 可知，PGD 的偏移量为 30，从中可以计算 PGD 页表的大小和 PGD 页表项数量。

```
/* PGD */
#define PGDIR_SHIFT     30
#define PGDIR_SIZE      (1UL << PGDIR_SHIFT)
#define PGDIR_MASK      (~(PGDIR_SIZE - 1))
#define PTRS_PER_PGD    (PAGE_SIZE / sizeof(pgd_t))
```

- PGDIR_SHIFT 宏表示 PGD 页表在虚拟地址中的起始偏移量。
- PGDIR_SIZE 宏表示 PGD 页表项所能映射的区域大小。
- PGDIR_MASK 宏用来屏蔽虚拟地址中的 PUD 索引、PMD 索引以及 PT 索引字段的所有位。
- PTRS_PER_PGD 宏表示 PGD 页表中页表项的个数。PGD 页表一共有 512 个页表项，每个页表项占 8 字节，所以一个 4 KB 大小的页面正好存储一个 PGD 页表。

接下来，计算 PMD 页表的偏移量和大小。

```
/* PMD */
#define PMD_SHIFT       21
#define PMD_SIZE        (1UL << PMD_SHIFT)
#define PMD_MASK        (~(PMD_SIZE - 1))
#define PTRS_PER_PMD (1 << (PGD_SHIFT - PMD_SHIFT))
```

- PMD_SHIFT 宏表示 PMD 页表在虚拟地址中的起始偏移量。
- PMD_SIZE 宏表示一个 PMD 页表项所能映射的区域大小。
- PMD_MASK 宏用来屏蔽虚拟地址中 PT 索引字段的所有位。
- PTRS_PER_PMD 宏表示 PMD 页表中页表项的个数。

最后是页表。由于设置页面粒度为 4 KB，因此页表的偏移量是从第 12 位开始的。

```
/* PTE */
#define PTE_SHIFT 12
#define PTE_SIZE (1UL << PTE_SHIFT)
#define PTE_MASK (~(PTE_SIZE-1))
#define PTRS_PER_PTE (1 << (PMD_SHIFT - PTE_SHIFT))
```

- PTE_SHIFT 宏表示页表在虚拟地址中的起始偏移量。
- PTE_SIZE 宏表示一个页表项所能映射的区域大小。
- PTE_MASK 宏用来屏蔽虚拟地址中 PT 索引字段的所有位。
- PTRS_PER_PTE 宏表示页表中页表项的个数。

页表项描述符包含丰富的属性，它们的定义如下。

```
#define _PAGE_PRESENT    (1 << 0)
#define _PAGE_READ       (1 << 1)
#define _PAGE_WRITE      (1 << 2)
#define _PAGE_EXEC       (1 << 3)
#define _PAGE_USER       (1 << 4)
#define _PAGE_GLOBAL     (1 << 5)
#define _PAGE_ACCESSED   (1 << 6)
#define _PAGE_DIRTY      (1 << 7)
#define _PAGE_SOFT       (1 << 8)
```

根据上述的页表项属性，我们又通过组合方式定义多种页面属性，如表 10.10 所示，例如只读页面。

```
#define _PAGE_BASE  (_PAGE_PRESENT | _PAGE_ACCESSED | _PAGE_USER)

#define PAGE_NONE         __pgprot(_PAGE_PROT_NONE)
#define PAGE_READ         __pgprot(_PAGE_BASE | _PAGE_READ)
#define PAGE_WRITE        __pgprot(_PAGE_BASE | _PAGE_READ | _PAGE_WRITE)
#define PAGE_EXEC         __pgprot(_PAGE_BASE | _PAGE_EXEC)
#define PAGE_READ_EXEC    __pgprot(_PAGE_BASE | _PAGE_READ | _PAGE_EXEC)
#define PAGE_WRITE_EXEC   __pgprot(_PAGE_BASE | _PAGE_READ |  \
                       _PAGE_EXEC | _PAGE_WRITE)
```

表 10.10 页面属性

页面属性	对应的字段
可读	V、A、U、R 字段
可读、可写	V、A、U、R、W 字段
可执行	V、A、U、X 字段
可读、可执行	V、A、U、R、X 字段
可读、可写、可执行	V、A、U、R、W、X 字段

在 BenOS 里，我们又可以根据内存属性划分页面类型。

- PAGE_KERNEL：操作系统内核中的普通内存页面。
- PAGE_KERNEL_READ：操作系统内核中只读的普通内存页面。
- PAGE_KERNEL_READ_EXEC：操作系统内核中只读的、可执行的普通页面。
- PAGE_KERNEL_EXEC：操作系统内核中可执行的普通页面。

```
#define _PAGE_KERNEL    (_PAGE_READ \
                | _PAGE_WRITE \
                | _PAGE_PRESENT \
                | _PAGE_ACCESSED \
                | _PAGE_DIRTY \
                | _PAGE_GLOBAL)

#define PAGE_KERNEL        __pgprot(_PAGE_KERNEL)
#define PAGE_KERNEL_READ   __pgprot(_PAGE_KERNEL & ~_PAGE_WRITE)
#define PAGE_KERNEL_EXEC   __pgprot(_PAGE_KERNEL | _PAGE_EXEC)
#define PAGE_KERNEL_READ_EXEC     __pgprot((_PAGE_KERNEL & ~_PAGE_WRITE) \
                      | _PAGE_EXEC)
```

10.4.2 页表数据结构

由于 L0～L2 页表的页表项都是 64 位宽的，因此它们可以使用 C 语言的 unsigned long long 类型来描述。

```
typedef unsigned long long u64;

typedef u64 pteval_t;
typedef u64 pmdval_t;
typedef u64 pgdval_t;

typedef struct {
    pteval_t pte;
} pte_t;
#define pte_val(x) ((x).pte)
#define __pte(x) ((pte_t) { (x) })

typedef struct {
    pmdval_t pmd;
} pmd_t;
#define pmd_val(x) ((x).pmd)
#define __pmd(x) ((pmd_t) { (x) })

typedef struct {
    pgdval_t pgd;
} pgd_t;
#define pgd_val(x) ((x).pgd)
#define __pgd(x) ((pgd_t) { (x) })
```

上面的代码中，pgd_t 表示一个 PGD 页表项，pmd_t 表示一个 PMD 页表项，pte_t 表示一个页表项。

10.4.3 创建页表

页表存储在内存中，页表的创建是由软件来完成的，页表的遍历则是由 MMU 自动完成的。

在打开 MMU 之前，软件需要手动把整个页表的相关页表项建立并填充好。

我们首先在链接脚本的数据段中预留 4 KB 大小的内存空间给 PGD 页表。

```
SECTIONS
{
    /*
     * 设置 BenOS 的加载入口地址为 0x8020 0000
     */
    . = 0x80200000,

    ...

    /*
     * 数据段
     */
    _data = .;
    .data : { *(.data) }
    . = ALIGN(4096);
    idmap_pg_dir = .;
    . += 4096;
    _edata = .;

    ...
}
```

idmap_pg_dir 指向的地址空间正好是 4 KB，用于 PGD 页表。接下来，使用__create_ pgd_mapping()函数逐步创建页表。

```
void __create_pgd_mapping(pgd_t *pgdir, unsigned long phys,
        unsigned long virt, unsigned long size,
        unsigned long prot,
        unsigned long (*alloc_pgtable)(void),
        unsigned long flags)
```

其中，pgdir 表示 PGD 页表的基地址；phys 表示要映射物理内存的起始地址；virt 表示要映射的虚拟内存的起始地址；size 表示要创建的映射的总大小；prot 表示要创建的映射的内存属性；alloc_pgtable 用来分配下一级页表的内存分配函数，PGD 页表在链接脚本里预先分配好了，剩下的页表则需要在动态创建过程中分配内存；flags 传递给页表创建过程中的标志位。

在本案例中，根据内存属性，我们将创建不同的恒等映射。

- 因为代码段具有只读、可执行属性，所以代码段必须映射到 PAGE_KERNEL_READ_EXEC 属性。
- 数据段以及剩下的内存属于普通内存，可以映射到 PAGE_KERNEL 属性。

```
static void create_identical_mapping(void)
{
    unsigned long start;
    unsigned long end;

    /*为代码段创建恒等映射*/
    start = (unsigned long)_text_boot;
    end = (unsigned long)_etext;
    __create_pgd_mapping((pgd_t *)idmap_pg_dir, start, start,
            end - start, PAGE_KERNEL_READ_EXEC,
            early_pgtable_alloc,
            0);

    printk("map text done\n");

    /*为内存创建恒等映射*/
    start = PAGE_ALIGN((unsigned long)_etext);
    end = DDR_END;
    __create_pgd_mapping((pgd_t *)idmap_pg_dir, start, start,
            end - start, PAGE_KERNEL,
            early_pgtable_alloc,
            0);
    printk("map memory done\n");
}
```

第一段代码为代码段创建恒等映射，它的起始地址是_text_boot，结束地址为_etext。第二段创建的恒等映射的起始地址为_etext，结束地址为内存的结束地址 DDR_END。

通过查看 benos.map 文件可知，_text_boot 地址是固定的入口地址，即 0x8020 0000，而_etext 的地址会随着代码段的大小而变化。

接下来，分析__create_pgd_mapping()函数的实现。

```
1    static void __create_pgd_mapping(pgd_t *pgdir, unsigned long phys,
2            unsigned long virt, unsigned long size,
3            pgprot_t prot,
4            unsigned long (*alloc_pgtable)(void),
5            unsigned long flags)
6    {
7        /*由 PGD 基地址和虚拟地址，找到对应 PGD 页表项*/
8        pgd_t *pgdp = pgd_offset_raw(pgdir, virt);
9        unsigned long addr, end, next;
10
11       phys &= PAGE_MASK;
12       addr = virt & PAGE_MASK;
13       end = PAGE_ALIGN(virt + size);
14
15       do {
16           /*找到 PGD 页表项管辖的范围*/
17           next = pgd_addr_end(addr, end);
18           alloc_init_pmd(pgdp, addr, next, phys,
19                   prot, alloc_pgtable, flags);
20           phys += next - addr;
21       } while (pgdp++, addr = next, addr != end);
22   }
```

在第 15～21 行中，以 PGDIR_SIZE 为步长遍历内存区域[virt, virt+size]，然后通过调用 alloc_init_pmd()分配 PMD 页表并填充 PGD 页表项。pgd_addr_end()以 PGDIR_SIZE 为步长。

pgd_addr_end()会计算一个 PGD 页表项的边界。如果 end 在管辖范围之内，则返回 end；否则，返回 PGD 页表项的边界。其定义如下。

```
#define pgd_addr_end(addr, end)                                          \
   ({      unsigned long __boundary = ((addr) + PGDIR_SIZE) & PGDIR_MASK;  \
            (__boundary - 1 < (end) - 1) ? __boundary : (end);            \
   })
```

alloc_init_pmd()函数的实现如下。

```
1    static void alloc_init_pmd(pgd_t *pgdp, unsigned long addr,
2            unsigned long end, unsigned long phys,
3            pgprot_t prot,
4            unsigned long (*alloc_pgtable)(void),
5            unsigned long flags)
6    {
7        pgd_t pgd = *pgdp;
8        pmd_t *pmdp;
9        unsigned long next;
10
11       /*若 PGD 页表项内容是空的，说明对应的 PMD 页表还没建立*/
12       if (pgd_none(pgd)) {
13           unsigned long pmd_phys;
14
15           /*分配一个页面，用于存放 PMD 页表*/
16           pmd_phys = alloc_pgtable();
17           /*用 PMD 基地址来回填 pgdp 页表项 */
18           set_pgd(pgdp, pfn_pgd(PFN_DOWN(pmd_phys), PAGE_TABLE));
19           pgd = *pgdp;
20       }
21
22       /*由 PGD 页表项和虚拟地址，找到对应的 PMD 页表项*/
23       pmdp = get_pmdp_from_pgdp(pgdp, addr);
24       do {
25           next = pmd_addr_end(addr, end);
26
27           alloc_init_pte(pmdp, addr, next, phys,
```

```
28                          prot,  alloc_pgtable, flags);
29
30              phys += next - addr;
31          } while (pmdp++, addr = next, addr != end);
32      }
```

alloc_init_pmd()函数会做如下事情。

在第 12～20 行中，通过 pgd_none()判断当前 PGD 页表项的内容是不是空的。如果 PGD 页表项的内容是空的，说明下一级页表还没创建，那么需要动态分配下一级页表。首先，使用 alloc_pgtable()函数分配一个 4 KB 页面，用于存放 PMD 页表。PMD 页表的基地址 pmd_phys 与相关属性 PAGE_TABLE 组成一个 PGD 的页表项，然后通过 set_pgd()函数设置到相应的 PGD 页表项中。

创建 PGD 页表项内容的过程如图 10.24 所示。

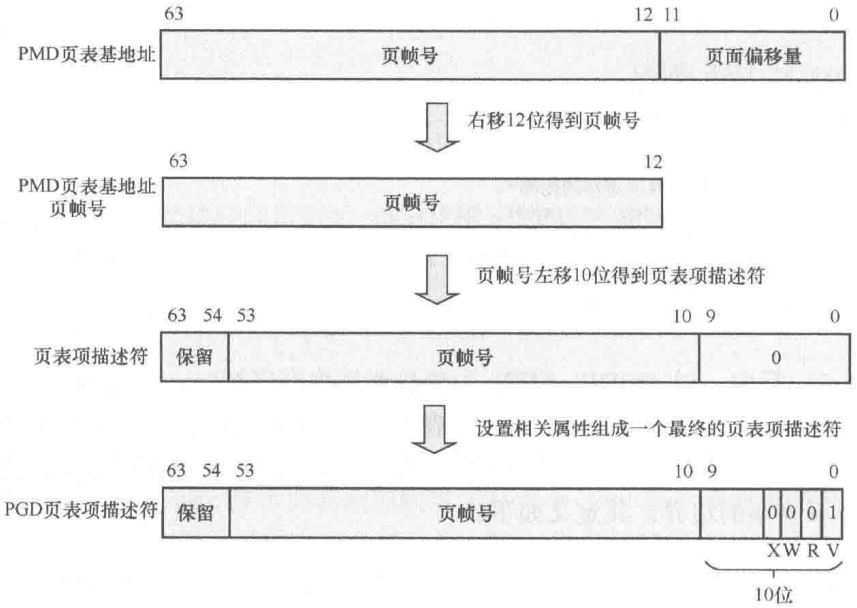


图 10.24　创建 PGD 页表项

主要步骤如下。

（1）计算 PMD 页表的基地址 pmd_phys 对应的页帧号 pmd_pfn。把 pmd_phys 地址右移 12 位，得到页帧号。

（2）把 pmd_pfn 左移 10 位，得到页表项描述符。

（3）设置相关的页表属性，例如，V 位设置为 1。另外，R 位、W 位以及 X 位设置为 0，表示指向下一级页表的页表项描述符。

在第 23 行中，get_pmdp_from_pgdp()函数通过 addr 和 PGD 页表项找到对应的 PMD 页表项。

在第 24～31 行中，以 PMD_SIZE 为步长，通过 while 循环设置下一级页表，依次调用 alloc_init_pte()函数来创建下一级页表。

```
1    static void alloc_init_pte(pmd_t *pmdp, unsigned long addr,
2            unsigned long end, unsigned long phys,
3            pgprot_t prot,
4            unsigned long (*alloc_pgtable)(void),
5            unsigned long flags)
6    {
7        pmd_t pmd = *pmdp;
8        pte_t *ptep;
9
10       /*若 PMD 页表项内容是空的，说明对应的页表还没建立*/
11       if (pmd_none(pmd)) {
12           unsigned long pte_phys;
```

```
13
14          /*分配一个页面，用于存放页表*/
15          pte_phys = alloc_pgtable();
16          /*用 PTE 基地址来回填 pmdp 表项 */
17          set_pmd(pmdp, pfn_pmd(PFN_DOWN(pte_phys), PAGE_TABLE));
18          pmd = *pmdp;
19      }
20
21      /*由 PMD 页表项和虚拟地址，找到对应的 PTE */
22      ptep = get_ptep_from_pmdp(pmdp, addr);
23      do {
24          /*设置 PTE*/
25          set_pte(ptep, pfn_pte(PFN_DOWN(phys), prot));
26          phys += PAGE_SIZE;
27      } while (ptep++, addr += PAGE_SIZE, addr != end);
28  }
```

页表是三级页表的最后一级，alloc_init_pte()函数用来配置页表项。

在第 11～19 行中，判断 PMD 页表项的内容是否为空的。如果为空的，说明下一级页表还没创建。使用 alloc_pgtable()来分配一个 4 KB 页面，用于存放页表的 512 个页表项。

PMD 页表项的内容包括页表的基地址页帧号 pte_pfn 以及页表项属性 PAGE_TABLE，创建的方法与 PGD 页表项类似，如图 10.24 所示。

在第 22 行中，get_ptep_from_pmdp ()函数通过 PMD 页表项和 addr 来获取页表项。

在第 23～27 行中，以 PAGE_SIZE（即 4 KB）为步长，通过 while 循环设置 PTE。调用 set_pte()来设置页表项。页表项包括物理地址的页帧号（phys >> PAGE_SHIFT）以及页表属性 prot，其中页表属性 prot 是通过__create_pgd_mapping()函数传递的，例如，代码段的恒等映射的页表属性为 PAGE_KERNEL_READ_EXEC，表示只读的、可执行的普通类型页面。

上述操作完成了一次建立页表的过程。读者需要注意如下两点。

- 页表的创建和填充是由操作系统来完成的，但是处理器则通过 MMU 来遍历页表。
- 除 PGD 页表是链接脚本中预留的之外，其他的页表是动态创建的。

在创建页表过程中经常使用到一个 early_pgtable_alloc()函数，它分配 4 KB 大小的页面，用于存放各级页表，如 PMD 页表、PD 页表等。

```
static unsigned long early_pgtable_alloc(void)
{
    unsigned long phys;

    phys = get_free_page();
    memset((void *)phys, 0, PAGE_SIZE);

    return phys;
}
```

注意，early_pgtable_alloc()分配的 4 KB 页面用于存放页表，最好把页面内容都清零，以免残留的数据干扰 MMU 遍历页面。

由于 BenOS 还没有实现伙伴分配系统，因此我们就使用数组分配和释放物理页面，而且我们直接从.bss 段的结束地址开始分配内存。

```
#define NR_PAGES (TOTAL_MEMORY / PAGE_SIZE)

static unsigned short mem_map[NR_PAGES] = {0,};

static unsigned long phy_start_address;

void mem_init(unsigned long start_mem, unsigned long end_mem)
{
    unsigned long nr_free_pages = 0;
    unsigned long free;

    start_mem = PAGE_ALIGN(start_mem);
    phy_start_address = start_mem;
```

```
    end_mem &= PAGE_MASK;
    free = end_mem - start_mem;

    while (start_mem < end_mem) {
        nr_free_pages++;
        start_mem += PAGE_SIZE;
    }

    printk("Memory: %uKB available, %u free pages\n", free/1024, nr_free_pages);
}

unsigned long get_free_page(void)
{
    int i;

    for (i = 0; i < NR_PAGES; i++) {
        if (mem_map[i] == 0) {
            mem_map[i] = 1;
            return phy_start_address + i * PAGE_SIZE;
        }
    }
    return 0;
}
```

10.4.4　打开 MMU

创建完页表之后需要设置 satp 寄存器来打开 MMU。

```
1    .global enable_mmu_relocate
2    enable_mmu_relocate:
3        la a2, idmap_pg_dir
4        srl a2, a2, PAGE_SHIFT
5        li a1, SATP_MODE_39
6        or a2, a2, a1
7        sfence.vma
8        csrw satp, a2
9        ret
```

在第 3 行中，加载 PGD 页表基地址 idmap_pg_dir。

在第 4 行中，获取 idmap_pg_dir 的页帧号。

在第 5 行中，设置分页机制的模式为 Sv39 模式。

在第 7 行中，SFENCE.VMA 指令是 RISC-V 体系结构中定义的一条内存屏障指令，用于同步页表写操作。

在第 8 行中，设置 satp 寄存器。

在第 9 行中，已经打开 MMU 功能。

下面是创建和打开 MMU 的整个过程。

```
1    void paging_init(void)
2    {
3        memset(idmap_pg_dir, 0, PAGE_SIZE);
4        create_identical_mapping();
5        create_mmio_mapping();
6
7        enable_mmu_relocate();
8    }
```

在第 3 行中，把 PGD 页表 idmap_pg_dir 清零。

在第 4～5 行中，动态创建恒等映射的页表，其中包括内核镜像、DDR 内存以及寄存器地址空间。

在第 7 行中，调用 enable_mmu_relocate()函数来打开 MMU。

10.4.5　测试 MMU

上面介绍了如何创建页表和打开 MMU，我们需要验证 MMU 是否正常工作。测试的方法很简

单，我们分别访问一个经过恒等映射和没有经过恒等映射的内存地址，观察系统会发生什么变化。

```
1    static int test_access_map_address(void)
2    {
3        unsigned long address = DDR_END - 4096;
4
5        *(unsigned long *)address = 0x55;
6
7        if (*(unsigned long *)address == 0x55)
8            printk("%s access 0x%x done\n", __func__, address);
9
10       return 0;
11   }
12
13   /*
14    * 访问一个没有建立映射的地址
15    *
16    * 存储/AMO 页面异常
17    */
18   static int test_access_unmap_address(void)
19   {
20       unsigned long address = DDR_END + 4096;
21
22       *(unsigned long *)address = 0x55;
23
24       printk("%s access 0x%x done\n", __func__, address);
25
26       return 0;
27   }
28
29   static void test_mmu(void)
30   {
31       test_access_map_address();
32       test_access_unmap_address();
33   }
```

在恒等映射中，我们映射了内核镜像的结束地址_ebss 到 DDR_END。test_access_ map_address()函数访问 DDR_END－4096 地址，这个地址是映射过的。test_access_unmap_address()函数访问 DDR_END＋4096 地址，这个地址还没有经过映射，CPU 访问这个地址会触发一个页表访问错误。

我们在 QEMU 上运行这个程序，结果如下。

```
rlk@master:benos$ make run
...
test_access_map_address access 0x87fff000 done
Oops - Store/AMO page fault
Call Trace:
[<0x0000000080202acc>] test_access_unmap_address+0x1c/0x42
[<0x0000000080202afe>] test_mmu+0xc/0x1a
[<0x0000000080202d1c>] kernel_main+0xa6/0xac
sepc: 0000000080202acc ra : 0000000080206f10 sp : 0000000080206fb0
 gp : 0000000000000000 tp : 0000000000000000 t0 : 0000000000000005
 t1 : 0000000000000005 t2 : 0000000080200020 t3 : 0000000080206fe0
 s1 : 0000000080200010 a0 : 0000000000000000 a1 : 0000000000000010
 a2 : ffffffffffffffff a3 : 0000000080206ed0 a4 : 0000000000000055
 a5 : 0000000088001000 a6 : 0000000000000000 a7 : 0000000000000061
 s2 : 8000000000006800 s3 : 0000000080200000 s4 : 0000000082200000
 s5 : 0000000000000000 s6 : 0000000000000000 s7 : 00000000800120e8
 s8 : 000000008020002e s9 : 000000000000007f s10: 0000000000000000
 s11: 0000000000000000 t3 : 45b0206f91166285 t4: 0000000080017ee0
 t5 : 0000000000000027 t6 : 0000000000000000
sstatus:0x8000000000006120  sbadaddr:0x0000000088001000  scause:0x000000000000000f
Kernel panic
```

从日志可以看到，BenOS 触发了一个写页面内容的地址转换异常，从 sbadaddr 寄存器可知出错地址为 0x8800 1000，test_access_unmap_address()函数访问 DDR_END＋4096 地址，这符合我们的预期，说明 MMU 已经正常工作。

10.4.6　图解页表创建的过程

本节以图解的方式来总结页表的创建过程。假设 PA=0x8020 0000，VA=0x8020 0000，映射大小为 4 KB，PGD 页表的基地址 idmap_pg_dir 为 0x8020 8000。

1. 填充和创建 PGD 页表项

通过虚拟地址查找 PGD 页表的索引值，从图 10.25 可知，虚拟地址 0x8020 0000 对应的 PGD 索引为 2。

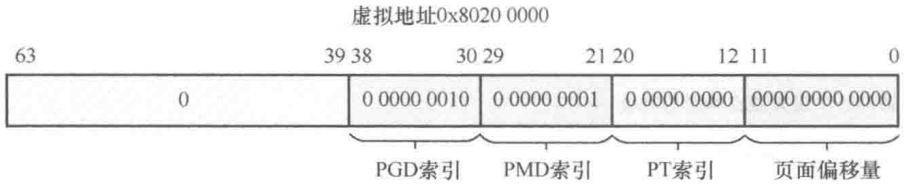


图 10.25　虚拟地址的布局

如图 10.26 所示，由 PGD 索引可以在 PGD 页表中找到页表项，即 PGD 页表项 2。

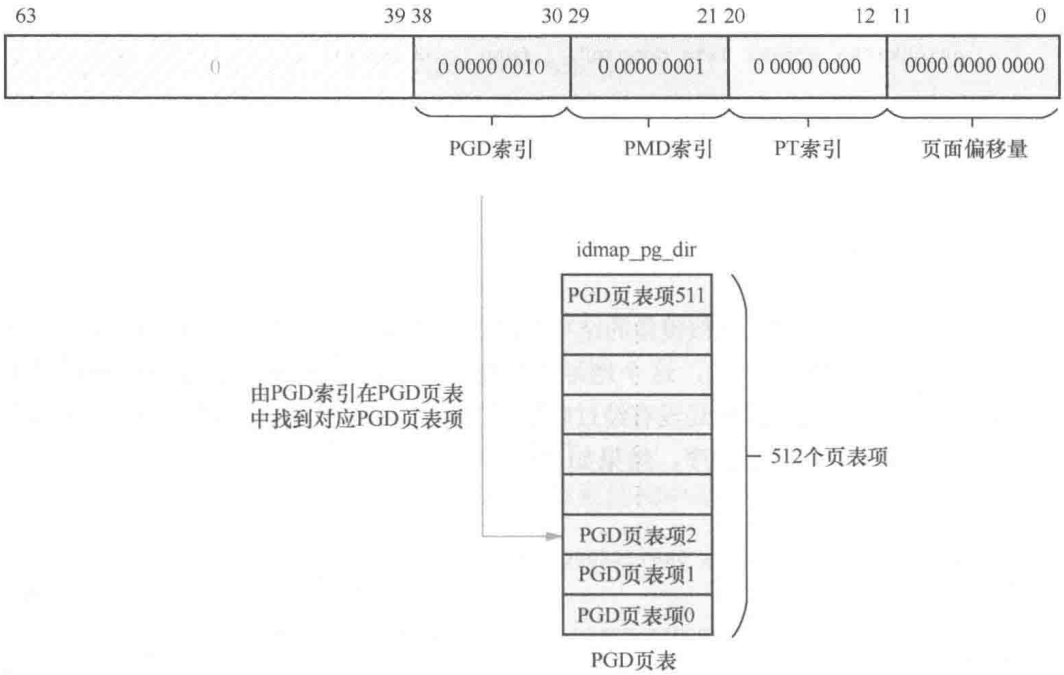


图 10.26　查找 PGD 页表项

由于 PGD 页表项 2 已空，因此需要构建页表项内容并填充该页表项。新创建一个下一级页表（即 PMD 页表），然后根据 PMD 页表的基地址可以构建 PGD 页表项的内容。

为 PMD 页表分配一个 4 KB 页面，假设这个页面的物理地址 pmd_phys 为 0x8022 C000，如图 10.27 所示。

PMD 页表基地址 pmd_phys 右移 12 位得到页帧号 pmd_pfn，然后把 pmd_pfn 左移 10 位，再加上页表属性（V 位），最后得到 PGD 页表项描述符，该描述符的内容为 0x2008 B001。注意，pmd_pfn 左移 10 位相当于页表属性都设置为 0，即 R 位、W 位、X 位都设置为 0，说明这个页表项为指向下一级页表的页表项，如图 10.28 所示。

把这个页表项描述符的内容写入 PGD 页表项 2 中，完成对 PGD 页表项 2 的填充，如图 10.29 所示。

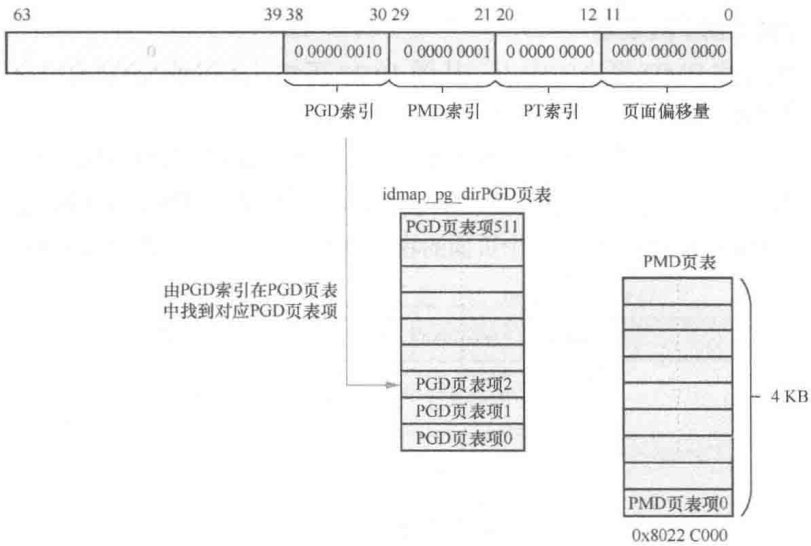


图 10.27　分配一个页面，用于存放 PMD

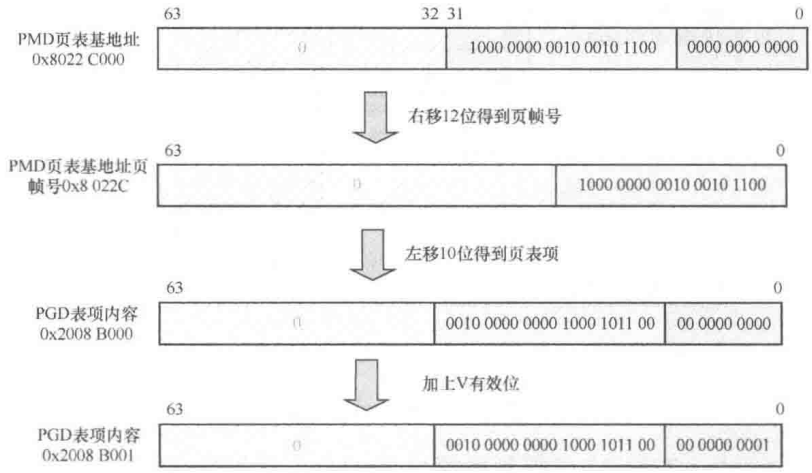


图 10.28　构造 PGD 页表项描述符

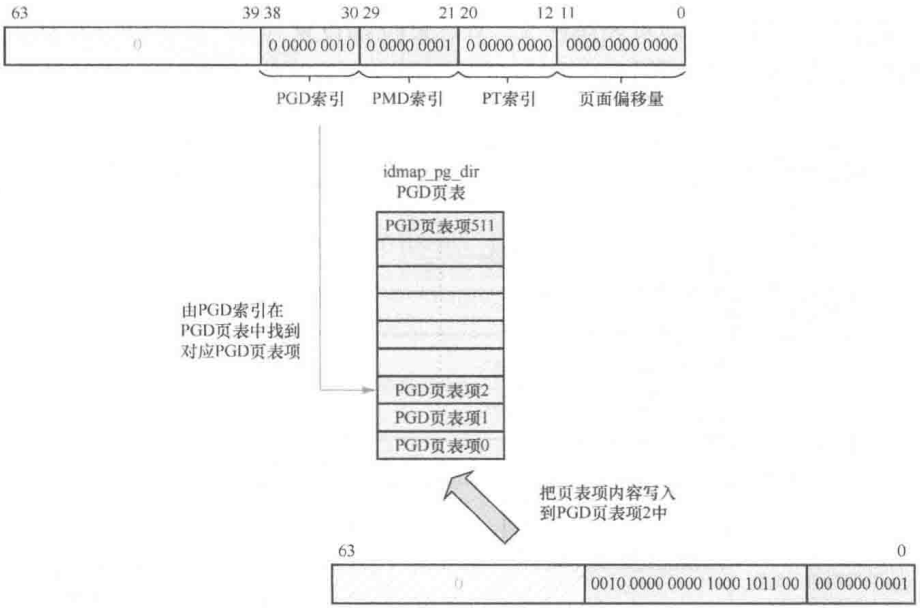


图 10.29　填充 PGD 页表项 2

2. 填充和创建 PMD 页表项

通过虚拟地址查找 PMD 页表的索引，从图 10.25 可知，虚拟地址 0x8020 0000 对应的 PMD 索引为 1，对应的页表项为 PMD 页表项 1。

由于 PMD 页表项 1 的内容是空的，因此需要构建一个 PMD 页表项描述符并填充该页表项。新创建一个下一级页表（即 PT），然后根据页表的基地址构建 PMD 页表项描述符的内容。

为页表分配一个 4 KB 页面，假设这个页面的物理地址 pte_phys 为 0x8022 D000，如图 10.30 所示。

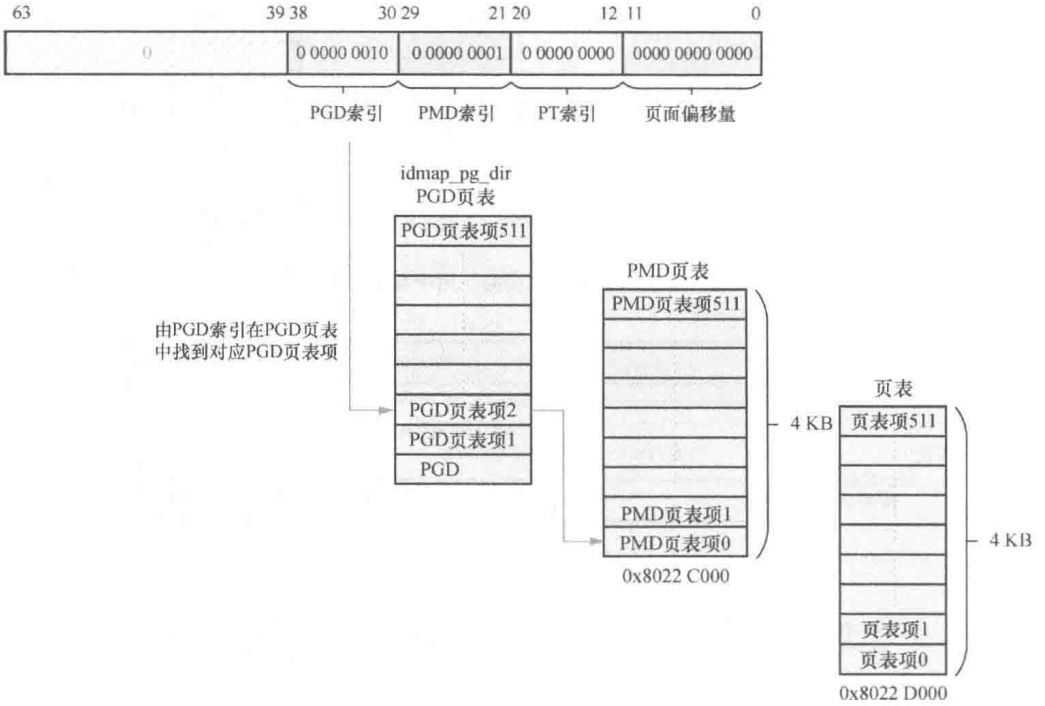


图 10.30　分配 PTE 页表

页表基地址 pte_phys 右移 12 位得到页帧号 pte_pfn，然后把 pte_pfn 左移 10 位，再加上页表属性 V 位，最后得到 PMD 表项描述符，它的内容为 0x2008 B401。注意，pte_pfn 左移 10 位相当于页表属性都设置为 0，即页表属性 R、W、X 位都设置为 0，说明这个页表项为指向下一级页表的页表项，如图 10.31 所示。

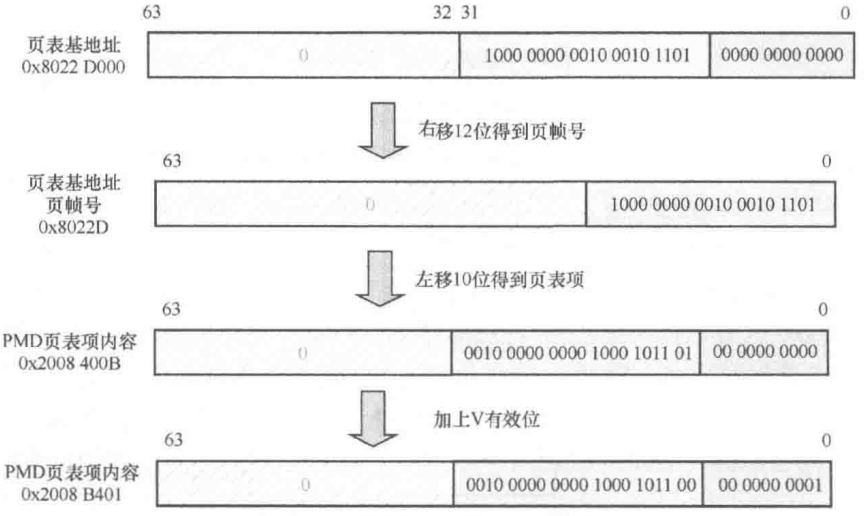


图 10.31　构建 PMD 页表项描述符

把这个页表项描述符内容写入 PMD 页表项 1 中，完成对 PMD 页表的填充。

3. 填充和创建页表项

通过虚拟地址查找页表的索引，从图 10.25 可知，虚拟地址 0x8020 0000 对应的页表项索引为 0，对应的页表项为页表项 0。

页表项 0 的内容为空的，需要根据物理地址（0x8020 0000）创建一个页表项描述符，如图 10.32 所示。

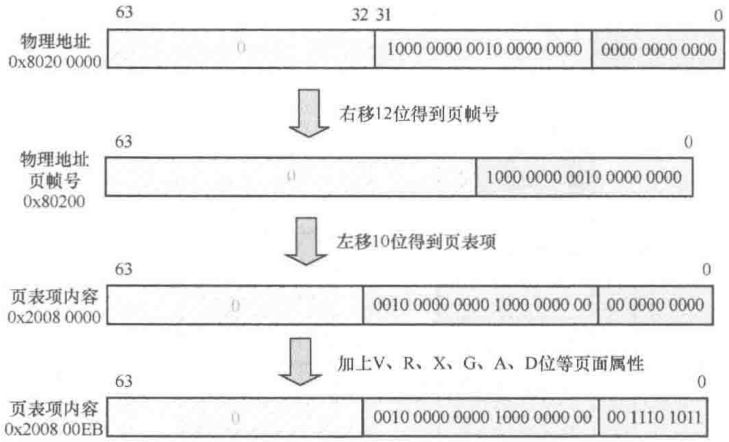


图 10.32 创建页表项描述符

把这个页表项的内容写入页表项 0 中，完成对页表的填充，如图 10.33 所示。

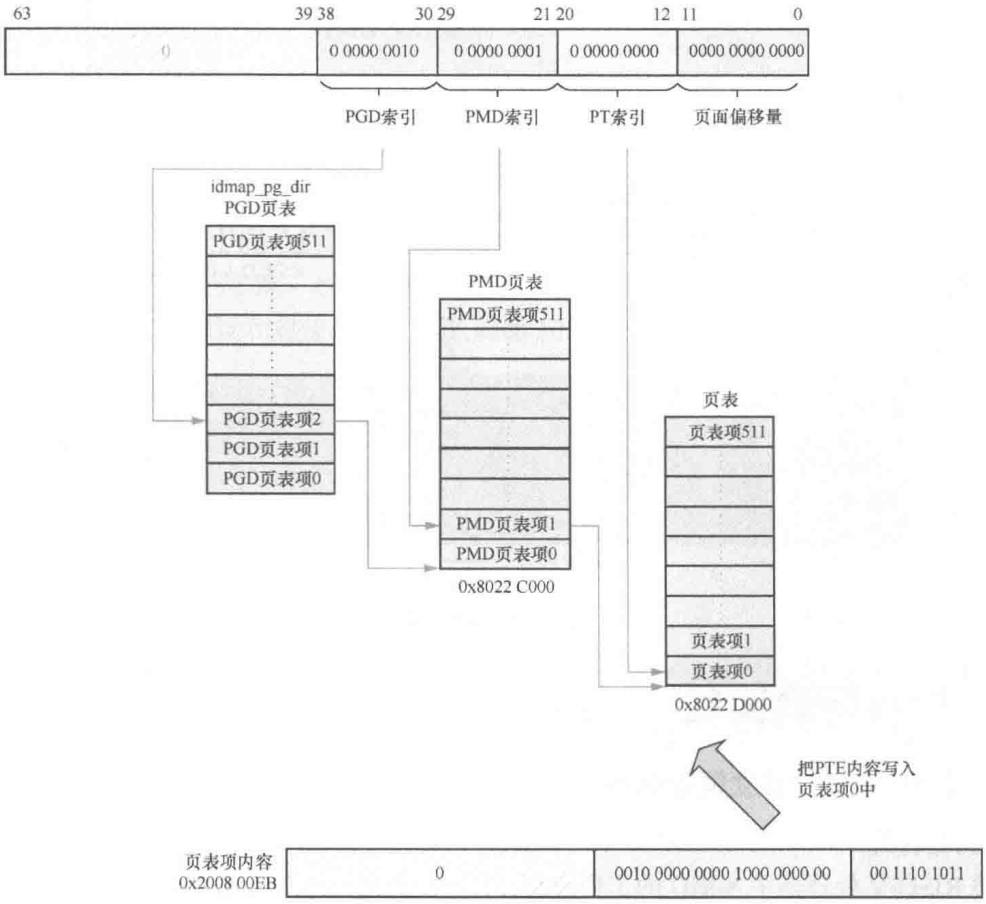


图 10.33 写入页表项 0 中

4. 最终映射图

虚拟地址到物理地址的映射过程如图 10.34 所示。

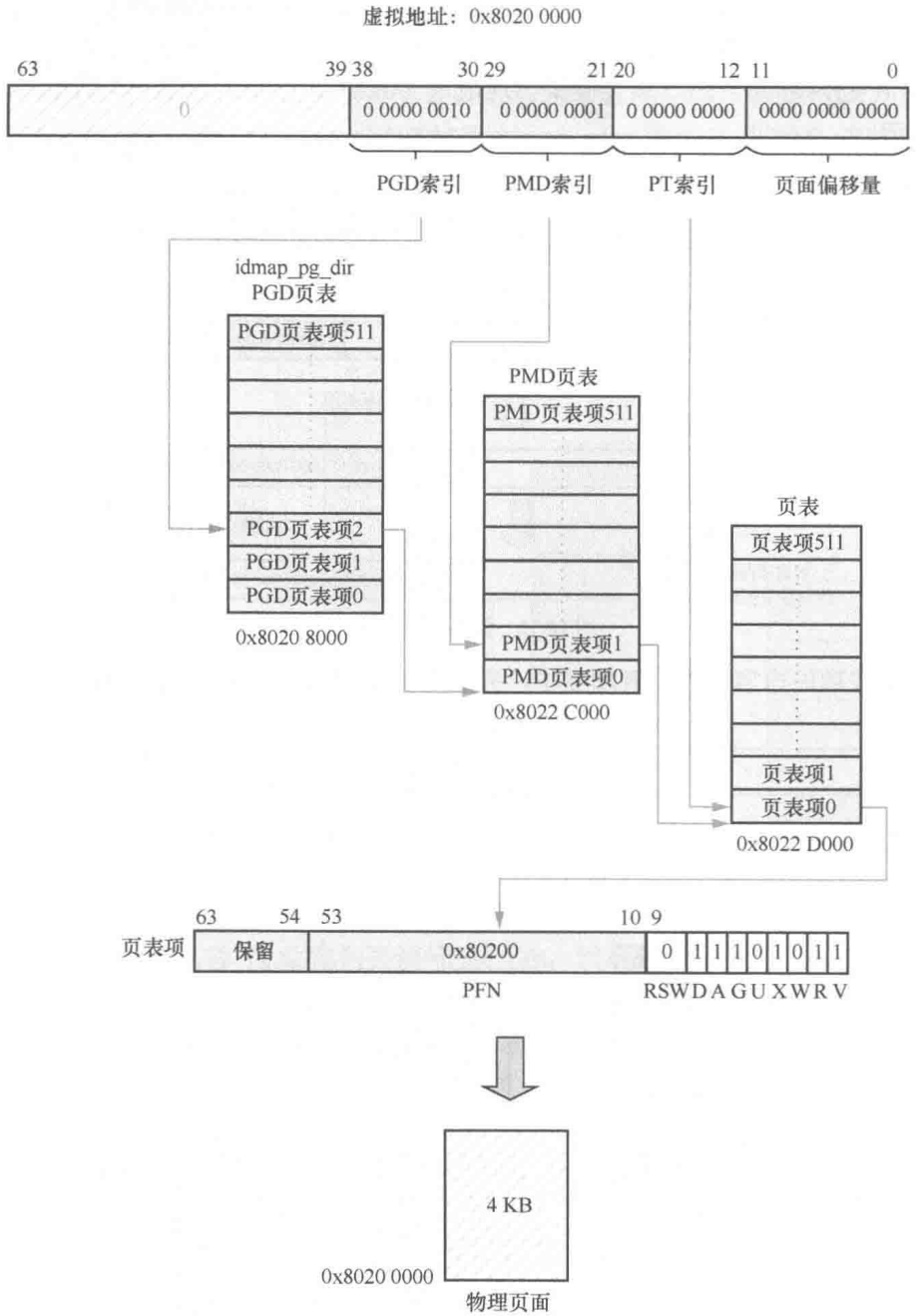


图 10.34　映射过程

10.5 内存管理实验

10.5.1　实验 10-1：建立恒等映射

1. 实验目的

熟悉 RISC-V 处理器中 MMU 的工作流程。

2. 实验要求

（1）在 QEMU 上建立一个恒等映射的页表，即虚拟地址等于物理地址。

（2）在 C 语言中实现页表的建立和 MMU 的开启功能。

（3）写一个测试例子来验证 MMU 是否开启了。

10.5.2 实验 10-2：为什么 MMU 无法运行

1. 实验目的

（1）熟悉 RISC-V 处理器中 MMU 的工作流程。

（2）培养调试和解决问题的能力。

2. 实验要求

某同学把实验 10-1 中的 create_identical_mapping()函数写成图 10.35 所示的形式。

```
static void create_identical_mapping(void)
{
        unsigned long start;
        unsigned long end;

        /*map memory*/
        start = (unsigned long)_text_boot;
        end = DDR_END;
        __create_pgd_mapping((pgd_t *)idmap_pg_dir, start, start,
                        end - start, PAGE_KERNEL,
                        early_pgtable_alloc,
                        0);
        printk("map memory done\n");
}
```

图 10.35　create_identical_mapping()函数

他发现系统无法运行，这是什么原因导致的？请使用 QEMU 与 GDB 单步调试代码并找出是哪条语句出现了问题。为什么 MMU 无法运行？

10.5.3 实验 10-3：实现一个 MMU 页表的转储功能

1. 实验目的

熟悉 RISC-V 处理器中 MMU 的工作流程。

2. 实验要求

在实验 10-1 的基础上实现一个 MMU 页表的转储（dump）功能，输出页表的虚拟地址、页表属性等信息，以方便调试和定位问题，结果如图 10.36 所示。

图 10.36　页表转储功能

10.5.4 实验 10-4：修改页面属性

1. 实验目的

熟悉页面属性等相关知识。

2. 实验要求

在系统中找出一个只读属性的页面，然后把这个页面的属性设置为可读、可写，使用 memset()函数往这个页面写入内容。

本实验的步骤如下。

（1）从系统中找出一个 4 KB 的只读页面，其虚拟地址为 vaddr。

（2）遍历页表，找到 vaddr 对应的页表项。

（3）修改页表项，为它设置可读、可写属性。

（4）使用 memset()修改页面内容。

10.5.5　实验 10-5：使用汇编语言来建立恒等映射

1. 实验目的

（1）熟悉 RISC-V 处理器中 MMU 的工作流程。

（2）熟悉页表建立过程。

（3）熟悉汇编的使用。

2. 实验要求

（1）在实验 10-1 的基础上，在汇编阶段使用汇编语言创建恒等映射，即大小为 2 MB 的块映射，并且打开 MMU。

（2）写一个测试例子来验证 MMU 是否开启。

10.5.6　实验 10-6：在 MySBI 中实现和验证 PMP 机制

1. 实验目的

熟悉 RISC-V 处理器的 PMP 机制。

2. 实验要求

（1）在 MySBI 中实现 PMP 配置功能，配置页表的属性为可读、可写、可执行。

```
pmp0cfg: 0x0-0xffff ffff ffff ffff
```

（2）在 MySBI 中实现 PMP 配置功能，先配置页表的属性为不可读、不可写、不可执行，再配置页表的属性为可读、可写、可执行。

```
pmp0cfg: 0x8000 0000-0x8004 0000
pmp1cfg: 0x0000 0000-0xffff fffff ffff ffff
```

（3）在 BenOS 中访问地址 0x8000 0000，请观察现象。

第11章 高速缓存

本章思考题

1. 为什么需要高速缓存？
2. 请简述 CPU 访问各级内存设备的延时情况。
3. 请简述 CPU 查询高速缓存的过程。
4. 请简述直接映射、全相联映射以及组相联映射的高速缓存的区别。
5. 在组相联高速缓存里，组、路、高速缓存行、标记域的定义分别是什么？
6. 什么是虚拟高速缓存和物理高速缓存？
7. 什么是高速缓存的重名问题？
8. 什么是高速缓存的同名问题？
9. VIPT 类型的高速缓存会产生重名问题吗？
10. 高速缓存中的直写和回写策略有什么区别？

高速缓存是处理器内部一个非常重要的硬件单元，虽然对软件是透明的，但是合理利用高速缓存的特性能显著提高程序的效率。本章主要介绍高速缓存的工作原理、映射方式、虚拟高速缓存、物理高速缓存，以及高速缓存的访问延时、高速缓存的访问策略、共享属性、高速缓存维护指令等方面的基础知识。

11.1 为什么需要高速缓存

在现代处理器中，处理器的访问速度已经远远超过了主存储器的访问速度。一条加载指令需要上百个时钟周期才能从主存储器读取数据到处理器内部的寄存器中，这不仅会导致使用该数据的指令需要等待加载指令完成才能继续执行，处理器处于停滞状态，还会严重影响程序的运行速度。解决处理器访问速度和内存访问速度严重不匹配问题是高速缓存设计的初衷。在处理器内部设置一个缓冲区，该缓冲区的速度与处理器内部的访问速度匹配。当处理器第一次从内存中读取数据时，也会把该数据暂时缓存到这个缓冲区里。这样，当处理器第二次读时，直接从缓冲区中取数据，从而大大地提升读的效率。同理，后续读操作的效率也得到了提升。这个缓冲区的概念就是高速缓存。第二次读的时候，如果数据在高速缓存里，称为高速缓存命中（cache hit）；如果数据不在高速缓存里，称为高速缓存未命中（cache miss）。

高速缓存一般是集成在处理器内部的 SRAM，相比外部的内存条造价昂贵。因此，高速缓存的容量一般比较小，成本高，访问速度快。如果程序的高速缓存命中率比较高，那么不仅能提升程序的运行速度，还能降低系统功耗。当高速缓存命中时，就不需要访问外部的内存模块，这有助于降低系统功耗。

通常，在系统的设计过程中，需要在高速缓存的性能和成本之间权衡，因此现代处理器系统都采用多级高速缓存的设计方案。越靠近 CPU 内核的高速缓存速度越快，成本越高，容量越小。如图 11.1 所示，经典的处理器体系结构包含多级高速缓存。CPU 簇 0 和 CPU 簇 1 均包含两个 CPU 内核，每个 CPU 内核都有自己的 L1 高速缓存。L1 高速缓存采用分离的两部分高速缓存。图中的 L1 D 表示 L1 数据高速缓存，L1 I 表示 L1 指令高速缓存。这两个 CPU 内核共享一个 L2 高速缓存。L2 高速缓存采用混合的方式，不再区分指令高速缓存和数据高速缓存。在这个系统中，还外接了一个扩展的 L3 高速缓存，CPU 簇 0 和 CPU 簇 1 共享这个 L3 高速缓存。

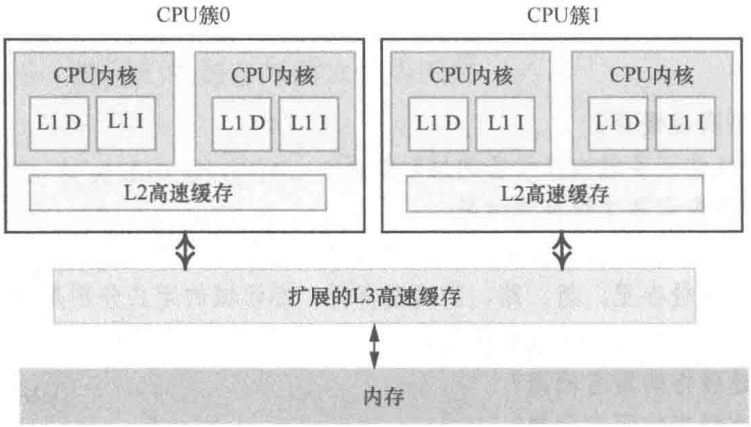


图 11.1　经典的高速缓存系统方案

高速缓存除带来性能的提升和功耗的降低之外，还会带来一些副作用。例如，高速缓存一致性，高速缓存伪共享，自修改代码导致的指令高速缓存和数据高速缓存的一致性等问题，本章会介绍这方面的内容。

11.2 高速缓存的访问延时

在现代广泛应用的计算机系统中，以内存为研究对象，体系结构可以分成两种：一种是均匀存储器访问（Uniform Memory Access，UMA）体系结构，另一种是非均匀存储器访问（Non-Uniform Memory Access，NUMA）体系结构。

- UMA 体系结构：内存有统一的结构并且可以统一寻址。目前大部分嵌入式系统、手机操作系统以及台式机操作系统采用 UMA 体系结构。如图 11.2 所示，该系统使用 UMA 体系结构，有 4 个 CPU，它们都有 L1 高速缓存。其中，CPU0 和 CPU1 组成一个簇（Cluster0），它们共享一个 L2 高速缓存。另外，CPU2 和 CPU3 组成另外一个簇（Cluster1），它们共享另外一个 L2 高速缓存。4 个 CPU 都共享同一个 L3 高速缓存。最重要的一点是，它们可以通过系统总线访问 DDR 物理内存。
- NUMA 体系结构：系统中有多个内存节点和多个 CPU 节点，CPU 访问本地内存节点的速度最快，访问远端内存节点的速度要慢一点。如图 11.3 所示，该系统使用 NUMA 体系结构，有两个内存节点。其中，CPU0 和 CPU1 组成一个节点（Node0），它们可以通过系统总线访问本地 DDR 物理内存。同理，CPU2 和 CPU3 组成另外一个节点（Node1），它们也可以通过系统总线访问本地的 DDR 物理内存。如果两个节点通过超路径互连（Ultra Path Interconnect，UPI）总线连接，那么 CPU0 可以通过这条内部总线访问远端内存节点的物理内存，但是访问速度要比访问本地物理内存慢很多。