国科大操作系统研讨课任务书

RISC-V 版本

版本 2023

目录

第零章 准备知识 **1**

1 引言 . . . 1

2 开发环境快速搭建 . . . 1

2.1 Windows 实验环境搭建 . . . 2

2.2 导入虚拟机 . . . 3

2.3 Linux 实验环境搭建 . . . 4

3 开发板及工具介绍 . . . 5

3.1 开发板的启动 . . . 5

3.2 minicom 的配置 . . . 6

4 Linux 常用命令介绍 . . . 7

4.1 文件夹操作 . . . 7

4.2 文件命令 . . . 8

4.3 vim 操作 . . . 8

5 RISC V 架构介绍 . . . 9

5.1 寄存器说明 . . . 9

5.2 应用程序二进制接口 . . . 10

6 RISC V 汇编介绍 . . . 11

6.1 RISC V 汇编语言 . . . 11

6.2 RISC V 常用汇编指令 . . . 16

7 编译及相关工具介绍 . . . 19

7.1 从编译到执行 . . . 19

7.2 编译流程 . . . 19

7.3 编译优化 . . . 23

7.4 Makefile . . . 25

7.5 链接器脚本 . . . 27

7.6 objdump . . . 27

8 QEMU 和 gdb 调试 . . . 28

8.1 QEMU 的启动 . . . 28

8.2 gdb 调试 . . . 29

9 附录：开发板内部细节 . . . 29

9.1 概述 . . . 29

9.2 制作可启动的 SD 卡 29
9.3 地址空间情况 32

Project 0
准备知识

1 引言

操作系统是计算机系统的重要系统软件，也是计算机专业教学中的一个重要内容。这门课程内容复杂且抽象，涉及到从体系结构到应用软件等多个方面的知识。我们认为掌握操作系统原理的最好方法就是自己编写一个操作系统。对于多数同学来说，知识如果不去使用，很难理解得深，也很容易就会忘掉。因此在这门课中，我们希望的是同学们能够通过自己的努力去从头实现一个自己的操作系统，并在实现的过程中不断加深对操作系统的理解，而不是让大家只停留在理论课上的“纸上谈兵”。

通过该实验课，大家将由浅入深，从操作系统的引导到操作系统复杂的内部实现，一步一步，深刻理解操作系统中进程的管理、中断的处理、内存的管理、文件系统等相关知识，并将其实现出来。我们衷心希望每个选修完这门课的同学以后都能自豪的说：“我自己实现过一个完整的操作系统”。

最后，希望大家如果在实验的过程中发现有什么问题，请积极和老师、助教反映，提出自己的意见。对于学有余力的同学，非常欢迎家能够加入我们，一起将国科大的操作系统实验课做的更好。接下来，我们将从环境搭建开始，一步一步实现我们的操作系统 UCAS-OS 吧！

2 开发环境快速搭建

Project0 主要是为后面的实验准备环境和工具。我们会涉及到作为开发环境的 Linux 操作系统、交叉编译器 Gcc、虚拟机软件 QEMU，以及基于 FPGA 的 RISC-V 开发板。我们的开发语言以 C 语言为主，但绕不开一些基本的 RISC-V 汇编语言，调试工具除了 print，一定要熟悉 gdb 断点和寄存器查看的功能，整个工程要通过 git 工具进行管理等等。如果这些工具已经把你弄的有点晕头转向，那么一定要耐心的花点时间看看 Project0 的相关知识，因为这些工具是操作系统开发的标准配置，特别是调试工具等。如果不能用好工具，开发操作系统可能会寸步难行。

看到这么多要学工具和准备知识也不要害怕，边学边用是我们的法宝。让我们开始吧。

为了让大家快速完成开发环境的搭建，我们已经将开发所需的环境集成到我们给所给的 VirtualBox 虚拟机镜像中，同学们只需要安装完成 VirtualBox 后导入我们所给的镜像，并简单的配置一下即可。当然，我们也在附录中给出了环境的具体流程，有感兴趣的同学可以了解一下。

环境搭建所需的工具，我们已经拷贝到发给大家的 SD 卡中，此外我们将工具也上传到了百度云中，链接为：https://pan.baidu.com/s/1g3R3Gl4Pp5GOWill6SzYGQ，密码是：ucas。

2.1 Windows 实验环境搭建

这一节主要讲解 Windows 下环境的搭建，搭建实验环境所需文件及描述见P0-1。

文件名称	说明	对应网盘目录
VirtualBox-6.1.38-153438-Win.exe	VirtualBox windows 安装包	Windows_virtualbox
UCAS_OS_2023.zip	为 RISC-V 版本实验准备的 VirtualBox 虚拟机镜像	已有交叉编译环境的镜像，需要解压
Oracle_VM_VirtualBox_Extension_Pack-6.1.38.vbox-extpack	VirtualBox 拓展包	Windows_virtualbox

表 P0-1: Windows 实验环境所需文件

首先，需要安装 VirtualBox 虚拟机，安装包为 VirtualBox-6.1.38-153438-Win.exe。右键打开菜单，选择**以管理员身份运行**。按照安装向导的提示一步步完成安装即可。完成后打开虚拟机软件（如图P0-1所示）。

完成 VirtualBox 的安装后，需要再进一步安装 VirtualBox 的扩展包。VirtualBox 扩展包主要用于增强 VirtualBox 的 USB 2.0、USB 3.0、摄像头等设备的支持，可以提高部分设备的性能。安装方法为：运行 VirtualBox-> 管理-> 全局设定-> 扩展-> 选择扩展包目录-> 安装-> 重启。过程如图P0-2所示。

(a) (b)

图 P0-1: 安装 Virtual Box

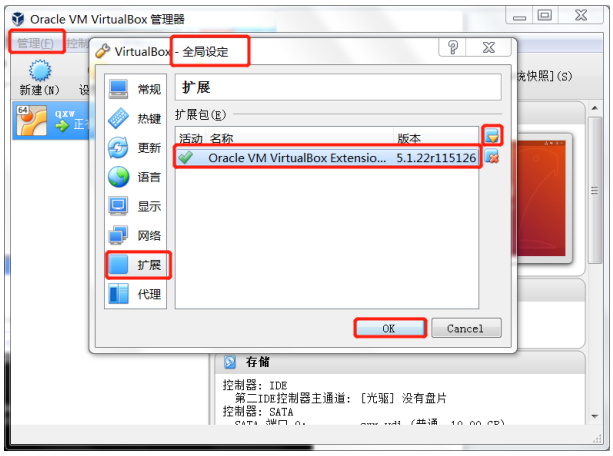


图 P0-2: 安装 Virtual Box

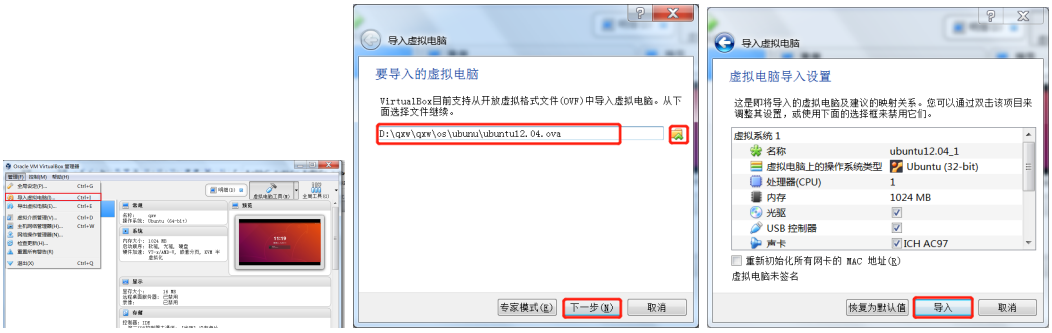
> **Note 2.1** VirtualBox 最早由 Sun Microsystems 公司开发并以 GPL 协议开源。在 Oracle 收购 Sun 后，VirtualBox 主体部分仍以 GPL 协议开源的，但一些增强功能以采用 PUEL 协议的 Extention Pack 的方式提供。该协议规定 [1]，可以出于个人和教育目的免费使用 Extention Pack。商业用途需付费购买许可。

之后，请按照2.2小节的说明，将虚拟机导入，导入后就可以使用我们预先制作好的 Ubuntu 虚拟机中的环境开始操作系统实验了。

2.2 导入虚拟机

VirtualBox 导入虚拟机的方式在不同平台上都是相同的，这里以 Windows 上导入虚拟机镜像为例。过程如图P0-3所示。RISC-V 版本实验环境采用 Ubuntu Server 20.04, 文件名为 UCAS_OS_2023.ova。请注意选择正确的虚拟机镜像进行导入。

我们预先配置好的镜像，用户名是 **stu**, 密码是 **123456**。



(a) 选择导入虚拟电脑 (b) 选择导入 Ubuntu23.08 (c) 点击导入

图 P0-3: 导入虚拟机

2.3 Linux 实验环境搭建

建议在 Linux 下也直接按照 VirtualBox, 然后按照2.2小节的内容直接导入我们准备好的虚拟机。我们准备的虚拟机上环境都已经配齐了。

当然，如果你不喜欢虚拟机的方式，我们也欢迎大家直接在 Linux 系统上配置整套。所以，下面的部分我们会介绍一下如何在 Linux 下自行构建交叉编译工具链，并配置相关开发环境。这也是我们为我们提供的虚拟机镜像配置相关环境的过程。

首先，安装 minicom。一般直接使用包管理器安装即可

```
在 Ubuntu/Debian/Deepin 等系统下
$ sudo apt-get install minicom
在 Fedora/CentOS 等系统下
$ sudo yum install minicom
```

接着，按照构建交叉编译工具链所需依赖

```
在 Ubuntu/Debian/Deepin 等系统下
$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev
$ sudo apt-get install libgmp-dev gawk build-essential bison flex texinfo gperf libtool
$ sudo apt-get install patchutils bc zlib1g-dev libexpat-dev
在 Fedora/CentOS 等系统下
$ sudo yum install autoconf automake libmpc-devel mpfr-devel gmp-devel gawk  bison flex
$ sudo yum install texinfo patchutils gcc gcc-c++ zlib-devel expat-devel
```

最后，解压我们提供的 riscv-gnu-toolchain.zip, 构建交叉编译工具链。

```
在解压出来 riscv-gnu-toolchain 同级目录下建立一个用于构建的目录
$ mkdir rv64-gnu-tools
进入新建立的目录
$ cd rv64-gnu-tools
构建并安装到/opt/riscv64-linux
$ ./configure --prefix=/opt/riscv64-linux
$ sudo make linux -j4
注意，-j4 代表 4 线程编译，你的机器几个核就设置使用几个线程，请自行调节该参数。
```

为了方便，可以将为工具链配置环境变量:

```
打开文件 bashrc
$ vi ~/.bashrc
按 i 进入编辑模式，在文件末尾添加语句：export PATH=/opt/riscv64-linux/bin:$PATH
按 ESC 退出编辑模式，按:wq 保存退出 vi。

重新刷新配置路径
$ source ~/.bashrc
```

测试一下，应该就可以找到交叉编译工具链了。

```
$ riscv64-unknown-linux-gnu-gcc -v
Using built-in specs.
COLLECT_GCC=riscv64-unknown-linux-gnu-gcc
COLLECT_LTO_WRAPPER=/opt/riscv64-linux/libexec/gcc/riscv64-unknown-linux-gnu/
8.3.0/lto-wrapper
Target: riscv64-unknown-linux-gnu
Configured with: /home/wangluming/os_course/riscv-gnu-toolchain/riscv-gcc/configure
--target=riscv64-unknown-linux-gnu --prefix=/opt/riscv64-linux --with-sysroot=
/opt/riscv64-linux/sysroot --with-system-zlib --enable-shared --enable-tls --en
```

```
able-languages=c,c++,fortran --disable-libmudflap --disable-libssp --disable-lib
quadmath --disable-nls --disable-bootstrap --src=/home/wangluming/os_course/risc
v-gnu-toolchain/riscv-gcc --enable-checking=yes --disable-multilib --with-abi=lp
64d --with-arch=rv64imafdc --with-tune=rocket 'CFLAGS_FOR_TARGET=-O2
-mcmodel=medlow' 'CXXFLAGS_FOR_TARGET=-O2  -mcmodel=medlow'
Thread model: posix
gcc version 8.3.0 (GCC)
```

编译 RISC-V 的交叉编译工具链有时候比较依赖某些库的特定版本。如果构建过程中发生编译错误，可能是版本不合适。具体哪个库有问题不好定位。笔者曾在 CentOS 7 和 Ubuntu 20.04 上试过，这两个发行版带的库的版本是可以正常编译的。webIDE 上的环境是 Ubuntu 的，所以可以正常安装。除了安装交叉编译工具链之外，为了方便调试，还推荐大家使用 QEMU 模拟器，安装的方法类似，使用我们提供的安装包编译 QEMU 和 U-boot 两个部分即可。需要注意的是，我们在 start-code 中给出了 Makefile，里面指定了一些默认的 QEMU 路径，所以需要大家在安装的时候也构造出相同的路径，或者根据自己的喜好，把 Makefile 里面的默认路径改掉。

3 开发板及工具介绍

实验采用 XILINX PYNQ Z2 开发板，开发板上有 ARM 和 FPGA。RISC V 核是烧写到 FPGA 里的。在开发板上电时由 ARM 核启动相关程序 (BOOT.BIN)，根据板卡上标有 SW1 的开关的状态，将 RISC V 处理器核烧入 FPGA。之后，RISC-V 核自动加载相关程序。

实验中使用的 RISC-V 核为升级后的双核 Nutshell。NutShell 为国科大第一届“一生一芯” 计划的产出，已在 Github 等网站上开源。在 PYNQ 板卡上，时钟主频为 60MHz。后续的实验中可以通过不同的命令启动单/双核 Nutshell。由于资源限制，我们提供的 RISC-V 核心均没有浮点模块。

3.1 开发板的启动

我们使用的开发板如图P0-4所示。请按照图示的顺序配置并使用开发板。

1. **将开发板设置为 SD 卡启动**图示中的 1 设置的是开发板的启动方式。PYNQ-Z2 支持从 SD、QSPI 和 JTAG 启动。我们将跳线插到最左侧，选择用 SD 启动。板子上也有标明，最左侧是 SD 启动。请保证跳线跳在了最左侧的两根针上面。

2. **设置电源选项**图示中的 2 设置的是开发板的供电方式。PYNQ 支持电源供电和 Micro-USB 供电。我们把跳线跳在 USB 这个选项上（靠上侧的两个针脚），用 Micro USB 直接供电。

3. **插入 SD 卡**插入根据9.2节所述制作好的 SD 卡。注意插入方向，不要插反。

4. **插入 Micro-USB** Micro-USB 一端连接图示 4 位置，一端连接笔记本的 USB 口。该线同时兼顾供电和传输数据的功能。

5. **打开电源开关**打开图示 5 位置的电源开关，开发板上红色指示灯亮起，之后绿色的 Done 信号灯亮起。之后代表 NutShell 的 LED 灯 LD0 亮起 1 秒后熄灭，随后 Done 信号灯重新亮起，表明开发板已开始工作。

6. **RESET 健**图示 6 位置为 reset 按键，板子上标记了 SRST 字样。按下该按键会让开发板重置。如果需要重新从开发板加电开始再执行一遍，可以按该按键。

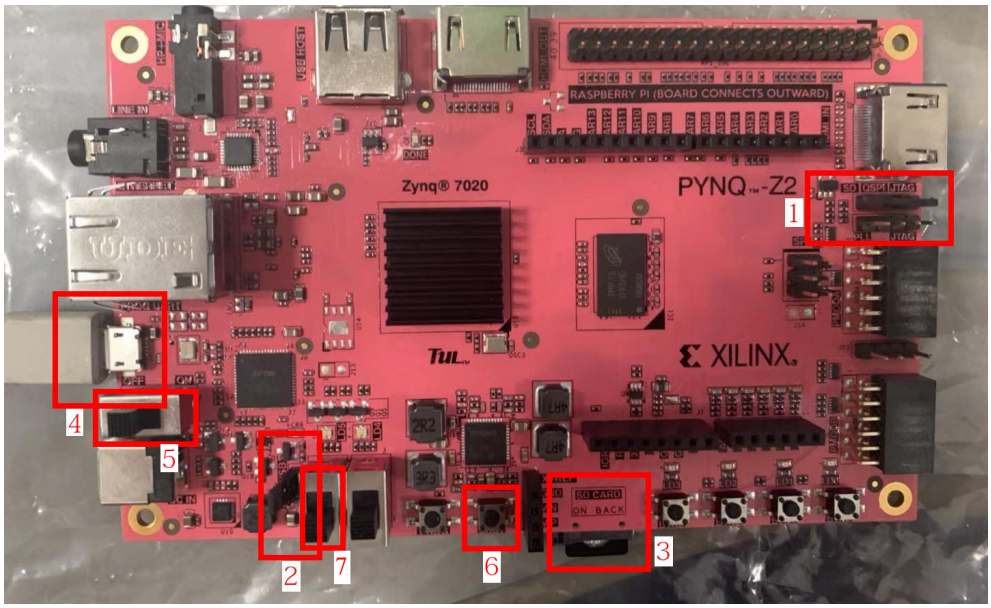


图 P0-4: PYNQ Z2 开发板

3.2 minicom 的配置

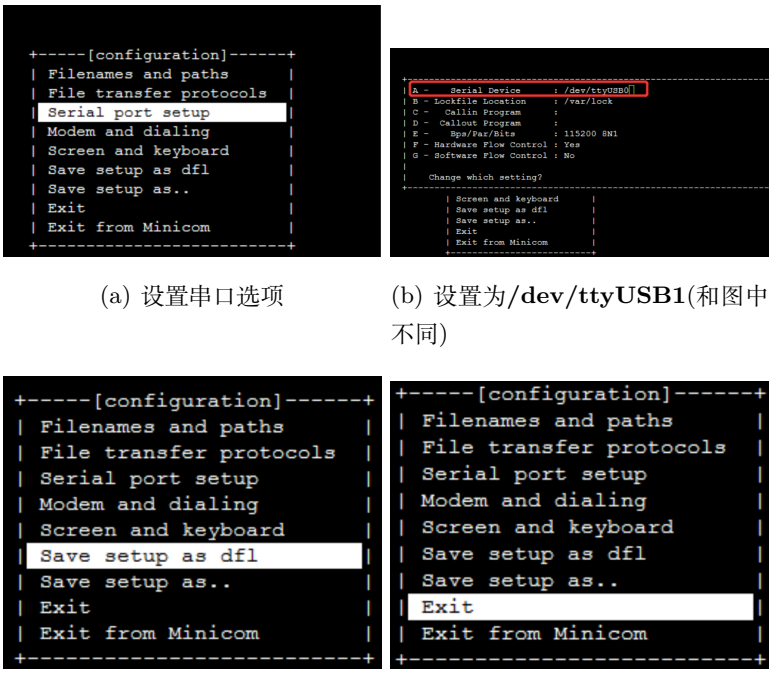
PYNQ-Z2 开发板和电脑通过 Micro-USB 线连接。连接好后，Micro-USB 线同时承担供电和通信的功能。在主机端，我们使用 minicom 和开发板通信。启动 minicom 的方法为：

```
$ sudo minicom -s
```

需要把 Serial port setup 中的 Serial Device 设为/dev/ttyUSB1。同时，确保 Bps/Par/Bits 那一项的值为 115200。过程如图P0-5所示。**注意：与图中不同，Serial Device 需要设置为/dev/ttyUSB1**。

将 SD 卡插入开发板（确保第一个分区中包含了 BOOT.BIN 等文件），在连接并启动开发板并打开 minicom 后，按下重置键（SRST）。如果设置正确，可以看到终端输出 RISC-V 启动环境的信息。

> **Note 3.1** 如果直接打开了 minicom，没有加-s 选项，也可以在 minicom 界面中，用键盘进行操作的方法是先按 Ctrl-A，再按相应按键。例如：Ctrl-A X 退出 minicom，Ctrl-A Z 显示帮助。



(a) 设置串口选项 (b) 设置为**/dev/ttyUSB1**(和图中不同)

(c) 保存设置 (d) 退出

图 P0-5: minicom 界面

4 Linux 常用命令介绍

在本课程中，我们使用 Linux 系统对代码进行编辑、编译，并用来连接开发板。通过学习熟悉 Linux 的基本使用方法，可以更高效的完成本课程的任务。

4.1 文件夹操作

表P0-2列出主要的 Linux 系统文件夹命令。

命令名称	命令作用
ls	显示当前目录下的所有文件夹和文件
mkdir	建立一个新的文件夹
cd	进入某一层文件夹
pwd	打印当前文件夹路径

表 P0-2: 文件夹操作命令

绝对路径和相对路径：文件夹命令的操作对象为指定路径的文件夹。在 Linux 系统中，有绝对路径和相对路径两种指定路径的方式。简单的来说，以/开头的路径会被 Linux 系统识别为绝对路径，否则为相对路径。因此，绝对路径指的是从根目录/开始的路径，而相对路径则代表当前路径下的路径。（使用 pwd 可以打印当前路径）需要注意的是，Linux 中有几个符号表示着特殊的路径，如表P0-3。

命令举例如表P0-4。

路径符合	含义
~	家路径，在本课程提供的镜像中，代表/home/stu
.	当前路径
..	上一级路径

表 P0-3: 特殊路径

命令内容	含义
cd ..	返回上一级目录
ls /home/stu	打印/home/stu 路径下的所有文件夹和文件
mkdir stu	在当前路径下创建名为 stu 的文件夹

表 P0-4: 文件夹命令举例

4.2 文件命令

表P0-5列出主要的 Linux 系统中进行文件操作的命令。-r 递归参数：Linux 系统的很多命令都支持-r 参数，例如 cp 命令，cp test.c /home/stu 是将 test.c 文件拷贝到路径/home/stu，cp -r ./ /home/stu 是将本文件夹全部拷贝至路径/home/stu。

| 管道符：管道符用于将管道符前面一个命令的输出作为后面一个命令的输入。例如 ls /home | grep ‘stu’ 的效果相当于在/home 路径下的文件夹和文件名称中查找是否有匹配 stu 字符串的内容。

命令名称	含义
cat	查看一个文件的内容
less	支持上下滚动行的查看文件内容的命令
grep	查找文件内容
chmod	改变文件权限
cp	copy，拷贝
rm	remove，删除

表 P0-5: 文件命令

文件命令使用举例如表P0-6所示：

4.3 vim 操作

vim 是一个功能强大的文本编辑器，本课程中，推荐在 Linux 系统中使用 vim 来编辑文件。vim 也可以用来直接创建新文件，只要后面的文件名并不存在，就会创建一个新的文件。例如 vim test.c。

命令名称	含义
less test.c	显示 test.c 文件的内容
rm -r stu	删除 stu
grep ‘hello’ test.c	在 test.c 中查找匹配字符串 hello 的内容
chmod +x test.sh	给 test.sh 文件赋予执行权限

表 P0-6: 文件命令使用示例

刚刚启动的 vim 程序处于命令模式，输入的字符都被 vim 视为命令而非编辑的文字。使用 i,a,s 可以进入编辑模式。在编辑模式中，任何字符的输入都会被认为普通输入的文字，除非使用 esc 键退出编辑模式回到命令模式。

在命令模式下，可以使用冒号键使屏幕下方出现一个冒号，在此时输入的任何字符都会出现在冒号后面。输入 w 然后回车可以保存当前编辑的文件，输入 q 然后回车可以退出 vim。类似的有 q! 和 wq 命令，分别是不保存退出和保存且退出。

在命令模式下还可以使用复制粘贴功能。分别是 d、y、p 键。

粘贴：p；小写 p 粘贴到当前游标之后，大写 P 粘贴到当前游标之前

剪切：d；dd 剪切当前行，d+ 数字 + 上下方向键剪切当前行和之前之后的 n 行

复制：y；yy 复制当前行，y+ 数字 + 上下方向键复制当前行和之前之后的 n 行

在命令模式下按/键会进入查找模式，通过输入要查找的字符或字符串可以查找，通过 n 或 N 查找下一个或上一个。

5 RISC V 架构介绍

5.1 寄存器说明

关于 RISC V 寄存器说明的详细内容，可以参阅 [2][3]。

我们的实验基于 RISC V 64 位架构。与大家在计算机组成原理实验课中接触到的 RISC V 32 位架构不同，在 RISC V 64 架构中，寄存器和指针的宽度都是 64 位。

RISC V 64 位架构总共有 32 个通用寄存器 (General-purpose register，简称 GPR) 和若干控制状态寄存器（Control Status Register，简称 CSR）。通用寄存器用于存储和操作通用数据, 它们是处理器中用于执行各种计算和数据传输操作的主要寄存器。

CSR 是一种特殊的寄存器，由处理器硬件定义和管理。它通常包含一组位字段(bits)，每个位字段代表不同的状态或控制标志。这些位字段可以存储和读取处理器的运行状态、中断状态、特权级别、运行模式等信息。CSR 的具体功能和位字段的含义会因不同的处理器体系结构而异。RISC V 架构中有一组约定的 CSR，用于管理和控制处理器的运行状态，如以下示例：

1. **sstatus**: 包含处理器的运行状态和特权级别信息。
2. **sie**: 包含处理器中断使能的标志位。
3. **scause**：用于存储最近的中断或异常原因。

4. **sepc**：存储异常程序计数器，指向中断或异常处理程序的地址。

通过读取和写入 CSR 的位字段，软件可以查询和修改处理器的运行状态和控制标志。例如，通过将位字段设置为特定的值，可以启用或禁用中断，修改特权级别，触发异常处理等。在 RISC V 架构中，只能使用控制状态寄存器指令（csrr、csrw 等）访问和修改 CSR。控制状态寄存器指令的使用与特权级相关，相关的内容将在后续的实验中介绍。

CSR 在处理器的内部实现中起着重要的作用，它提供了一种机制来管理和控制处理器的行为。同时，CSR 也是处理器与操作系统、编译器等软件之间的接口，用于进行状态传递和控制。

5.2 应用程序二进制接口

应用程序二进制接口（Application Binary interface，简称 ABI）定义了应用程序二进制代码中相关数据结构和函数模块的格式及其访问方式。这个约定是人为的，硬件上并不强制这些内容，自成体系的软件可以不遵循部分或者全部 ABI。为了和编译器以及其他的库配合，我们应该在写汇编代码时尽量遵循约定。ABI 包含但不限于以下内容：

1. 处理器基础数据类型的大小。布局和对齐要求
2. 寄存器使用约定。约定通用寄存器的使用方法、别名等。
3. 函数调用约定。约定参数调用、结果返回、栈的使用。
4. 可执行文件格式
5. 系统调用约定

下面主要介绍寄存器调用约定

通用寄存器使用约定

RISC V 中的通用寄存器分为两类，一类在函数调用的过程中不保留，称为**临时寄存器**。另一类寄存器则对应的称为 **保存寄存器**。表P0-7罗列出了寄存器的 RISC V 应用程序二进制接口（ABI）名称和它们在函数调用中是否保留的规定。除了保存寄存器之外，调用者需要保证用于存储返回地址的寄存器（ ra ）和存储栈指针的寄存器（ sp ）在函数调用前后保持不变。简而言之，如果某次函数调用需要改变保存寄存器的值，就需要采用适当的措施在退出函数时恢复保存寄存器的值。在第 6 节中将结合相应的汇编代码对寄存器使用约定和函数调用约定做进一步的介绍。

寄存器编号	助记符	用途	在调用中是否保留
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No
f0–7	ft0–7	FP temporaries	No
f8–9	fs0–1	FP saved registers	Yes
f10–11	fa0–1	FP arguments/return values	No
f12–17	fa2–7	FP arguments	No
f18–27	fs2–11	FP saved registers	Yes
f28–31	ft8–11	FP temporaries	No

表 P0-7: RISC V 通用寄存器[3]

6 RISC V 汇编介绍

6.1 RISC V 汇编语言

关于 RISC V 汇编语言的详细内容，可以参阅 [2][4][3]。

在此，我们讲述一些 C 语言和 RISC V 语言的对应关系，便于大家后面编写汇编代码。汇编语言可以理解为机器语言的直接翻译，是对于处理器的最直接的操作。RISC 类型的处理器提供 load/store 类指令将变量从内存载入到寄存器或从寄存器写回内存。除了 load/store 类的指令之外，其他指令都是在寄存器之间的操作。

为了便于大家理解汇编如何编写，下面我们演示一下，我们所熟悉的 C 语言是如何被转换为汇编的。作为例子，这里选用一个简单的选择排序来作为演示。为了演示到所有的情况，我们有意使用了循环、函数调用等元素，以观察这些是怎么被翻译到汇编的。

Listing 1: C 语言选择排序例程

```
1 #include <stdio.h>
```

```c
#include <stdlib.h>

#define MAX_N 100

int buf[MAX_N];

void do_sort(int a[], int n)
{
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (a[i] > a[j]) {
                // swap a[i],a[j]
                a[i] ^= a[j];
                a[j] ^= a[i];
                a[i] ^= a[j];
            }
        }
    }
}

int main()
{
    int n;
    scanf("%d", &n);

    int t = n;
    while (t--) {
        scanf("%d",&buf[t]);
    }

    do_sort(buf,n);

    for (int i = 0; i < n; ++i) {
        printf("%d ",buf[i]);
    }
    printf("\n");
    return 0;
}
```

那么，C 对应的汇编是什么样呢？首先，GCC 有一个特性：所有的循环，都会换成`do{}while();`的形式来实现。例如，GCC 翻译出来的循环用 C 语言形象地表示是这个样子的：

```c
for (int i = 0; i < n; ++i) {
   // ...
}
// 会被翻译成
int i = 0;
if (i >= n) goto END;
do {
   // ...
   ++i;
} while (i < n);
END:
// 进一步被翻译成
int i = 0;
goto L2;
L1:
```

```
  // ...
  ++i;
L2:
  if (!(i<n)) goto END;
  goto L1;
END:
```

函数调用会将第一个参数放在 a0 寄存器，第二个参数放在 a1 寄存器，依次类推，最后用 call 指令调用相应的函数。

```
ld a1, -24(s0); # 假设第二个参数位于-24(s0)
ld a0, -20(s0); # 假设第二个参数位于-20(s0)
call func # 相当于 func(a0,a1);
```

进入函数时，先分配栈空间。分配方法就是将栈指针减去需要的空间数（栈是向下增长的）。sp 到 sp-X 这 X 字节的空间就是当前函数运行所需的栈空间。将保存寄存器和返回地址寄存器（ra）的值存储在这部分栈空间中，退出时恢复。对于调用者来说，保存寄存器和返回地址寄存器（ra）的值在函数调用前后是不变的。栈指针（sp）的值也会在函数退出时进行恢复。此外，如果函数中使用了较多的局部变量，也会在栈空间上多开辟一部分空间用于存储局部变量。在内核编程中，内核栈的大小通常是有限的，如果函数中使用了大量的局部数据，很可能会爆栈而引发奇怪的问题。

```
func:
  addi sp,sp,-32
  sd   s0, 0(sp)
  sd   s1, 8(sp)
  sd   ra, 16(sp)
  # ...
  ld   ra, 16(sp)
  ld   s1, 8(sp)
  ld   s0, 0(sp)
  addi sp,sp,32
  jr ra
```

一般汇编里加载地址有两个常用方式：绝对地址加载和 PC 相对加载。

直接加载绝对地址的示例如下：

```
.section .text
.globl _start
_start:
        lui a0,       %hi(msg)       # load msg(hi)
        addi a0, a0,  %lo(msg)       # load msg(lo)
        jal ra, puts
2:      j 2b

.section .rodata
msg:
        .string "Hello World\n"
```

PC 相对的地址加载方式如下：

```
.section .text
.globl _start
_start:
1:      auipc a0,     %pcrel_hi(msg) # load msg(hi)
        addi  a0, a0, %pcrel_lo(1b)  # load msg(lo)
        jal ra, puts
2:      j 2b

.section .rodata
msg:
        .string "Hello World\n"
```

下面是 C 编译器翻译出来的代码1对应的 RISC V 汇编代码。

Listing 2: RISC-V 汇编选择排序例程

```
        .file        "riscv-example.c"
        .option nopic
        .text
        .comm        buf,400,8
        .align        1
        .globl        do_sort
        .type        do_sort, @function
# void do_sort(int a[], int n)
do_sort:
        addi        sp,sp,-48
        sd        s0,40(sp)
        addi        s0,sp,48
        sd        a0,-40(s0)
        mv        a5,a1
        sw        a5,-44(s0)
        sw        zero,-20(s0)
        j        .L2
        # ...
        # 未经优化的代码太长了，不再赘述
        # 下面做了个 a[i]^=a[j]
        lw        a5,-20(s0) # -20(s0) 是 i
        slli        a5,a5,2  # a5=i*4
        ld        a2,-40(s0) # 取出 a[] 的首地址
        add        a5,a2,a5 # a5 现在是 a[i] 的地址了
        xor        a4,a3,a4 # a[i]^a[j], a[i] 和 a[j] 在前面已经 load 进 a3 和 a4 了
        sext.w        a4,a4
        sw        a4,0(a5) # 把结果写回 a[i] 的地址上
        # ...
        # 恢复保留寄存器的值，返回
        ld        s0,40(sp)
        addi        sp,sp,48
        jr        ra
        .size        do_sort, .-do_sort
        .section        .rodata
        .align        3
.LC0:
        .string        "%d"
        .align        3
.LC1:
        .string        "%d "
```

```
        .text
        .align  1
        .globl  main
        .type   main, @function
# int main()
main:
        # 分配栈空间,sp 为栈顶，栈向下增长
        addi    sp,sp,-32
        # 保存保留寄存器
        sd      ra,24(sp)
        sd      s0,16(sp)
        # s0 用作帧指针，指向栈底
        addi    s0,sp,32
        # scanf("%d", &n);
        addi    a5,s0,-28 # -28(s0) 是 n 的位置
        mv      a1,a5     # 把 n 作为 scanf 的第二个参数
        lui     a5,%hi(.LC0)
        addi    a0,a5,%lo(.LC0) # 把"%d" 作为第一个参数
        call    __isoc99_scanf # 调用 scanf
        # int t = n;
        lw      a5,-28(s0) # -28(s0) 是 n
        sw      a5,-20(s0) # -20(s0) 是 t
        j       .L8 # goto .L8
        # do {
.L9:
        # scanf("%d",&buf[t]);
        lw      a5,-20(s0)
        slli    a4,a5,2 # a4=t*4,int 为 4 字节
        lui     a5,%hi(buf)
        addi    a5,a5,%lo(buf) # a5=buf
        add     a5,a4,a5 # a5=buf+t*4, 即 &buf[t]
        mv      a1,a5  # a5 作为 scanf 第二个参数
        lui     a5,%hi(.LC0)
        addi    a0,a5,%lo(.LC0) # "%d" 作为 scanf 第一个参数
        call    __isoc99_scanf  # 调用 scanf
.L8:    # } while ((t--) != 0);
        lw      a5,-20(s0)
        addiw   a4,a5,-1
        sw      a4,-20(s0)
        # 上面三句实现的是 t--,
        # 先把值取到 a5, 再把 a5-1 存回 t 的位置
        bne     a5,zero,.L9 # t!=0 时跳回.L9, 实现 while 的语义

        # sort(buf, n);
        lw      a5,-28(s0)
        mv      a1,a5 # n 作为第二个参数
        lui     a5,%hi(buf) # buf 作为第一个参数
        addi    a0,a5,%lo(buf)
        call    do_sort # 调用 do_sort

        # int i = 0; goto .L10
        sw      zero,-24(s0) # -24(s0) 是 i
        j       .L10
        # do {
.L11:
        # printf("%d ",buf[i])
        lui     a5,%hi(buf)
        addi    a4,a5,%lo(buf) # a4=buf
        lw      a5,-24(s0)
```

```
100         slli        a5,a5,2 # a5= t*4
101         add         a5,a4,a5 # a5=buf+t*4
102         lw          a5,0(a5) # a5 = buf[t]
103         mv          a1,a5 # buf[t] 作为第二个参数
104         lui         a5,%hi(.LC1)
105         addi        a0,a5,%lo(.LC1) # "%d " 作为第一个参数
106         call        printf # 调用 printf
107         # ++i;
108         lw          a5,-24(s0)
109         addiw       a5,a5,1
110         sw          a5,-24(s0)
111 .L10:   # } while (i < n);
112         lw          a4,-28(s0)
113         lw          a5,-24(s0)
114         sext.w      a5,a5
115         blt         a5,a4,.L11
116         # printf("\n"); 用 putchar('\n') 实现的
117         li          a0,10
118         call        putchar
119         li          a5,0
120         # return 0;
121         mv          a0,a5 # a0 放返回值
122         # 恢复 ra、s0、sp 原来的值
123         ld          ra,24(sp)
124         ld          s0,16(sp)
125         addi        sp,sp,32
126         jr          ra # 返回
127         .size       main, .-main
128         .ident      "GCC: (GNU) 8.3.0"
129         .section    .note.GNU-stack,"",@progbits
```

6.2 RISC V 常用汇编指令

RISC-V 的算术、逻辑运算等指令用法可以参考 [2] 一书。RISC 常用伪指令如表P0-8和表P0-9所示。伪指令是为了编写汇编方便所准备的指令，会被汇编器自动翻译成多条汇编指令。

伪指令	基础指令 (即被汇编器翻译后的指令)	含义
`fence`	`fence iorw, iorw`	Fence on all memory and I/O
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`ret`	`jalr x0, 0(x1)`	Return from subroutine
`call offset`	`auipc x1, offset`$[31:12]+$`offset`$[11]$ `jalr x1, offset[11:0](x1)`	Call far-away subroutine
`tail offset`	`auipc x6, offset`$[31:12]+$`offset`$[11]$ `jalr x0, offset[11:0](x6)`	Tail call far-away subroutine

表 P0-8: RISC-V 伪指令 [3]

伪指令	基础指令 (即被汇编器翻译后的指令)	含义
`la rd, symbol (non-PIC)`	`auipc rd, delta[31:12] + delta[11]` `addi rd, rd, delta[11:0]`	Load absolute address, where `delta = symbol − pc`
`la rd, symbol (PIC)`	`auipc rd, delta[31:12] + delta[11]` `l{w|d} rd, rd, delta[11:0]`	Load absolute address, where `delta = GOT[symbol] − pc`
`lla rd, symbol`	`auipc rd, delta[31:12] + delta[11]` `addi rd, rd, delta[11:0]`	Load local address, where `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)`	Load global
`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)`	Store global

The base instructions use `pc`*-relative addressing, so the linker subtracts* `pc` *from* `symbol` *to get* `delta`*. The linker adds* `delta[11]` *to the 20-bit high part, counteracting sign extension of the 12-bit low part.*

`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned

表 P0-9: RISC-V 伪指令 (续)[3]

7 编译及相关工具介绍

7.1 从编译到执行

我们即将用 C 语言来实现操作系统内核，但是 C 语言这样的高级语言离机器能够理解和运行还有不小的差距。对于一个不复杂的应用程序来说（例如 helloworld），让它运行起来的最简单的方式是使用 GCC 编译，输入以下命令：

```
$ gcc hello.c -o hello
```

在同级文件夹下将会生成一个可执行文件 hello。这里需要注意的是，这个 hello 文件目前是存放在硬盘上的，而计算机执行程序的时候需要从内存取指令，这就意味着我们在执行它的时候，这段代码首先被搬到了内存中。那么这个搬运的过程是不是原封不动地将代码从硬盘拷贝到内存呢？并没有这么简单，这个可执行文件实际上是 ELF 格式的，不仅仅是简单平铺的目标机器代码。关于 ELF 的具体内容会在未来的实验中进一步学习，在这里只需要知道，这个 ELF 文件中有许多不同作用的段，除了包含目标代码的代码段，还至少包括数据段、bss 段，还可能包括字符串表、符号表、动态链接信息、调试信息、只读数据段等各种各样的段。总结来说，所谓 ELF 文件只是 Linux 系统在硬盘中存放二进制程序的一种中间状态，仍然需要对它做进一步的解读才能让机器开始执行。

当我们有多个 C 文件时，事情发生了变化。一方面，我们可以和单文件一样，将多个文件名同时提供给 gcc 作为参数，直接产生可执行文件，例如：

```
$ gcc hello.c main.c -o hello
```

这样一个命令其实包含了多个行为，GCC 编译器首先会将每个 c 文件单独编译，各自产生一个 ELF 文件，这些文件包含了目标代码，称为目标文件（.o 文件）。但显然每个 ELF 文件都是不完整和无法运行的，main.c 里可能调用了一个位于 hello.c 的函数，这就需要进行一步链接操作，将多个 ELF 文件拼合起来。具体来说，链接过程需要合并各个 ELF 文件的段，合并符号表，并且对变量、函数等重定位，让不同文件中的函数能互相调用。上面这个命令的效果等同于以下操作：

```
$ gcc -c hello.c -o hello.o
$ gcc -c main.c -o main.o
$ gcc hello.o main.o -o hello
```

在我们的项目源文件数量比较少时，似乎看不出独立编译再链接这个行为的意义，直接将所有文件放到一起直接编译出完整的可执行文件好像更省事。随着工程规模的增加，模块化的设计越来越有必要，对整个工程进行完整编译的时间成本显著提高，并且对代码中一个小地方的改动要将所有文件从头重新编译，还伴随着库文件重复编译的问题等等。

7.2 编译流程

为了更好地理解和控制编译的行为，以帮助我们写出符合需要的代码，还需要进一步了解编译过程中的几个步骤：**预编译**、**编译**、**汇编**、**链接**（如图P0-6所示）。

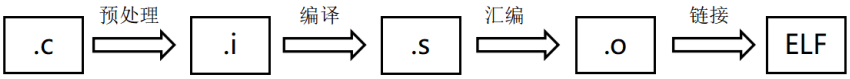


图 P0-6: 编译流程

预编译：识别文件中的一部分宏，如 #define、#include 等，完成文本级的替换，此时还不涉及代码的实际功能。

编译：将高级语言（C 语言等编程语言）转化为汇编语言，汇编语言仍然是文本形式的，无法直接运行，但是比 C 语言的文本结构简单的多，看上去基本就是机器语言的样子了。

汇编：将汇编语言转化为机器能识别的二进制目标代码，到这一步这个程序才初步具备了被执行的一些要素。

链接：将多个分离的二进制代码合并成最终的可执行文件。

下面将通过一个简单的例子来具体阐述各个阶段的行为。项目代码有 hello.h(Listing 3)、hello.c(Listing 4)、main.c(Listing 5) 三个文件。这 3 个文件主要完成输出"hello world"的简单工作。

Listing 3: hello.h

```h
#ifndef HELLO_H
#define HELLO_H

void hello_world();

#endif /* HELLO_H */
```

Listing 4: hello.c

```c
#include "hello.h"
#include "stdio.h"

void hello_world()
{
    printf("Hello World!\n");
}
```

Listing 5: main.c

```c
#include "hello.h"

int main()
{
    hello_world();
```

```
    return 0;
}
```

预编译

GCC 编译器可以分步执行上述编译过程中的每个步骤，我们只需要在 gcc 命令后面添加参数-E 就可以只进行预编译这第一步，现在我们在终端输入下列命令：

```
$ gcc -E hello.c -o hello.i
$ gcc -E main.c -o main.i
```

我们打开生成的文件，hello.i 和 main.i，可以发现，里面的内容如代码 6所示。这里只展示 main.i, 因为 hello.i 引用了标准输入输出，导致预编译完的文件很长，限于篇幅不再展示。

Listing 6: 预编译结果

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "main.c"
# 1 "hello.h" 1



void hello_world();
# 2 "main.c" 2

int main()
{
    hello_world();
    return 0;
}
```

可以发现，相对于预编译之前，生成的新文件只是简单的做了一下宏替换，将 include 的头文件的内容放进了文本中，而没有进行其他任何语言间的转化。

编译

在编译这一步骤，编译器要正式开始工作了。同理，我们将刚才生成的.i 文件继续编译，添加-S 参数，在终端输入以下命令：

```
$ gcc -S hello.i -o hello.s
$ gcc -S main.i -o main.s
```

我们打开生成的.s 文件，发现里面内容如下（同样只展示 main.s, 如代码7所示）：

Listing 7: 编译结果

```
        .file   "main.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, %eax
        call    hello_world@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Gentoo 9.1.0-r1 p1.1) 9.1.0"
        .section    .note.GNU-stack,"",@progbits
```

可以发现，原来的 **C 语言代码已经被转化成为了汇编代码**，这正是编译这一步所进行的工作。在对汇编语言有基本的了解后，就能大致阅读这段代码了，层次化、结构化的高级语言被转化成了线性的、连续排列的指令，每一行都有指令助记符和操作数，以及其他必要的提示语句。

你可能知道，编译器除了将高级代码按照语言规范逐条生成效果等价的汇编语言之外，还会对代码本身做一定程度的优化，例如削减变量、常量替换、循环展开等等，这些优化大部分是在这一步完成的。

汇编

汇编这一步骤，进一步将编译所生成的汇编代码转化成机器能识别的二进制机器代码，然后将其中的数据、代码等以 ELF 格式打包成目标文件。我们在终端里输入以下命令：

```
$ gcc -c hello.s -o hello.o
$ gcc -c main.s -o main.o
```

打开生成的.o 文件，里面的内容如图P0-7所示。是的！我们生成的文件已经是一个二进制文件，里面存放的数据都是只有机器才能识别的机器代码啦 除了少量的字符串作为数据保存，其余部分已经无法以文本的格式阅读了。

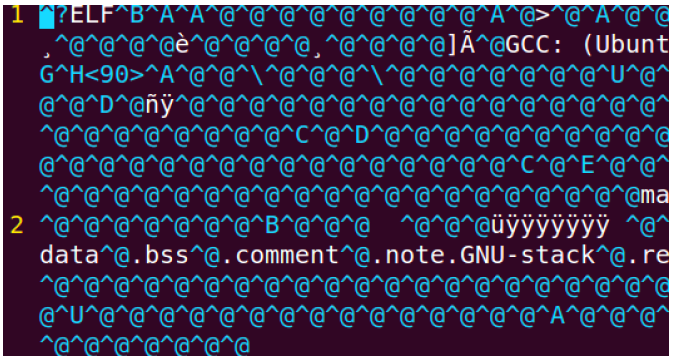


图 P0-7: 编译出的 main.o

链接

到目前，我们生成的文件有 main.o 和 hello.o 这 2 个二进制文件，但是它们现在还不能直接运行，因为它们是彼此分离的，所以我们需要通过链接这一重要的步骤去将彼此分离的二进制代码合并成最终的可执行文件。我们在终端输入以下命令：

```
$ gcc hello.o main.o -o main
$ ./main
Hello World!
```

最终，我们的 “Hello World” 项目从 3 个文件，合并成为了一个名为 main 的可执行文件，可以让操作系统去执行它并且打印出 “Hello World!”。是不是很神奇？在链接阶段，我们也经常会直接调用链接器 ld 代替 gcc，它们的效果是一致的。

7.3 编译优化

在内核编程中，熟知编译优化相关的知识非常重要。成熟的内核代码一定要经得起编译优化，保证行为的正确性。我们熟知的 Linux 内核就是使用 -O2 编译优化选项进行编译的。gcc 提供了不同级别的优化选项，可以在编译时添加 -OX 选项指定，其中’X’ 代表编译级别。

这里我们以往届学生在使用编译优化选项时出现的问题为例，提醒大家在编写代码一定要考虑不同的优化选项下生成的目标代码的行为。

案例分析

代码 8 展示了一段往届同学使用内联汇编进行系统调用的错误代码。这里大家不用深究系统调用的概念，只需要知道该段代码的功能是将 sysno 放置到 a7 寄存器中，arg0 ~arg4 分别按顺序放置到 a0 ~a4 寄存器中。

Listing 8: invoke_syscall.c 内敛汇编展示

```
static long invoke_syscall(long sysno, long arg0, long arg1, long arg2,
                           long arg3, long arg4)
{
```

```
    long res;
    asm volatile(
        "add a7, zero, a0\n\t"
        "add a0, zero, a1\n\t"
        "add a1, zero, a2\n\t"
        "add a2, zero, a3\n\t"
        "add a3, zero, a4\n\t"
        "add a4, zero, a5\n\t"
        "ecall\n\t"
        "mv %0, a0"
        :"=r"(res)
    );
}
```

可见该代码充分利用了 RISC V 函数调用 ABI 中指定的参数放置规定。因为在调用 invoke_syscall 函数时，sysno 已经放置到了 a0 寄存器中，剩余的参数也按照顺序保存到了 a1 ~a5 寄存器当中。于是在内联汇编代码中只做了简单的寄存器值交换动作。

代码 9 展示了 invoke_syscall 的一个简单封装调用。

Listing 9: -O0 优化选项下调用 invoke_syscall

```
int exhibit(int mbox_idx, void *msg, int msg_length)
{
    return invoke_syscall(SYS_EXHIBIT,\
                          (long)mbox_idx, \
                          (long)msg, \
                          msg_length, IGNORE, IGNORE);
}
```

这段代码在 -O0 优化选项下自然没有任何问题。因为 RISC V ABI 会正确的参数传递到对应的寄存中。但在 -O2 选项下，invoke_ syscall 函数会被优化为内联函数。同时 gcc 识别到传递给 invoke_syscall 的参数没有被使用到，因此直接将传参的步骤省去，优化后的代码等价于代码 10 所示。可见，直接丢失了原本要保存到 a7 寄存器中的 SYS_EXHIBIT 参数。并且参数也没有按照预期移动到 a0 ~a4 寄存器当中

Listing 10: -O2 优化选项下调用 invoke_syscall

```
int exhibit(int mbox_idx, void *msg, int msg_length)
{
    long res;
    asm volatile(
        "add a7, zero, a0\n\t"
        "add a0, zero, a1\n\t"
        "add a1, zero, a2\n\t"
        "add a2, zero, a3\n\t"
        "add a3, zero, a4\n\t"
        "add a4, zero, a5\n\t"
        "ecall\n\t"
        "mv %0, a0"
        :"=r"(res)
    );
```

```
15 }
```

这段原本在 -O0 选项下可以正常工作的代码在 -O2 下出现了问题。具体的原因在于我们需要在内联汇编中告诉编译器需要使用到具体的参数，使得编译器优化时保证参数的正确传递。如代码 11 所示

Listing 11: 修正后的 invoke_syscall

```
1  static long invoke_syscall(long sysno, long arg0, long arg1, long arg2,
2                             long arg3, long arg4)
3  {
4      long res;
5      asm volatile(
6          "mv a7, %[sysno]\n\t"
7          ...
8          :"=r"(res)
9          :[sysno] "r" (sysno)
10         :"r" (sysno) ...
11     );
12 }
```

经过了上述的流程，你应该已经对一个项目从编译到执行已经有了清楚的认识，但是仍然会有一些问题在困扰你：难道每编译一次项目都要手打这么多复杂命令吗？链接是通过什么规则将这些可执行文件合并在一起的呢？生成的可执行文件要如何在不运行的情况下做静态分析，以确定编译出的机器码符合我们的预期呢？

当然，这些问题我们都将在接下来的部分具体阐述，给介绍项目编译利器——Makefile，介绍如何通过链接器脚本控制我们的链接过程，以及一种方便快捷的反汇编命令——objdump。

7.4 Makefile

类似上面的例子，我们使用几行简单的命令就能够轻易地搞定一个小程序的编译过程，那么对于更大的程序呢？例如我们即将着手实现的内核可能会有数十个 C 文件需要编译，这些文件分散在以程序逻辑结构划分的诸多文件夹内，现实生活中一个项目的文件更是成百上千，显然不可能依靠人工手动逐个编译链接起来。

那么写一个 shell 脚本如何？shell 脚本允许我们将需要执行的编译命令预先写好，需要编译的时候只需要运行这个脚本就可以了。它没有手动编译繁琐，但是这种方法的可扩展性仍然很受限。一方面，编译的时候除了要关心编译哪个文件，还需要管理好头文件、静态链接库、预定义的参数、繁多的编译选项等等，要写出这样一个脚本就是繁重的重复劳动，一旦其中某个文件做了修改需要重新编译的时候，整个脚本的全部流程都要重新执行一遍；另一方面，如果项目功能调整，需要增加或者删除文件，或者文件目录结构发生了变化，就要仔细校对脚本中每个相关的地方，并且保证别的命令不能出错或者缺乏依赖，要维护这样一个脚本的成本过于高昂。

所幸在 Linux 下，有着 Makefile 这一利器，它能够帮助我们简化书写、执行编译过程。Makefile 仍然是一个脚本文件，在 shell 中能完成的指令同样都可以写入 Makefile

作为其自动化流程的一部分。与 shell 不同的是，你只需要设定好编译的规则，而不需要为每个单独的文件特意编写一条命令。一条规则一般包括目标、依赖、规则内容三部分，目标和依赖会用冒号分隔，具体的规则另起一行，例如我们在目录中创建一个名为“Makefile”的文件：

```
hello: hello.o main.o
    gcc hello.o main.o -o hello

hello.o: hello.c
    gcc -c hello.c -o hello.o

main.o: main.c
    gcc -c main.c -o main.o
```

想要完成 hello 这个目标，你只需要一条简单的命令：

```
$ make hello
```

或者 Makefile 默认会完成第一个目标：

```
$ make
```

Makefile 一个很大的优势在于，它能够自动管理规则之间的依赖，当你想完成 hello 这个目标的时候，它就会先将其依赖的 main.o 和 hello.o 完成；此外，Makefile 每次被执行的时候并不会从头开始，例如某个源文件 hello.c 被修改了需要重新编译，那么 make 命令在运行前扫描文件的时间戳，就会发现 hello.c 的最后修改时间发生了变化，从而识别出依赖 hello.c 的 hello.o 和 hello 这两个目标需要被重新执行，而不需要重新执行 main.o 这个目标。这就实现了我们想要的增量编译，在文件数量很多的情况下，能够大大节省每次编译花费的时间。

Makefile 的优点可以总结为：规则式管理（编译什么，按什么顺序，怎么编译）、增量式编译、一般不需要对 Makefile 做大幅修改。

在本次操作系统实验课中，大部分的项目代码我们都已经写好了 Makefile 供大家使用，只需要能够看懂简单的 Makefile 并且学会使用它。除了编译，许多其他需要执行的命令也会一并整合进 Makefile，例如启动 qemu 模拟器、拷贝镜像进 SD 卡等等，省去翻找和输入命令的时，只需要使用“make 目标”这样一行就可以一步执行对应的所有指令。如果你对某个操作具体使用了哪些命令感兴趣，可以参看这些 Makefile。

这里再给出一个实用的 make 参数-n，使用这个参数能够看到你即将运行的一次 make 将会具体执行什么样的命令，而不实际执行它们，例如在只修改了 hello.c 之后可以运行：

```
$ make -n hello
gcc -c hello.c -o hello.o
gcc hello.o main.o -o hello
```

到这里你可能会觉得上面那种 Makefile 写起来还是太麻烦，比 shell 脚本没有好多少，这是因为 Makefile 还有很多能提高生产力的写法和技巧，例如默认变量、自动变量、函数、默认规则等等。但这些规则很难一下子全部掌握，一个复杂的 Makefile 也有一定的理解门槛，如果你仍希望更加深入的理解 Makefile，熟悉掌握相关知识的话，可以查阅网上更多的资料 [5][6][7]，了解如何编写 Makefile。

7.5 链接器脚本

前几节中说过，在链接阶段，多个.o 文件会被合并成为一个 ELF 格式的可执行文件。ELF 里包含各种段，每个段包含的内容也不一样，有的包含数据，有的包含代码。在刚才的 demo 里我们直接使用了 gcc 命令进行链接，这个链接是使用了默认的规则。因此生成的 ELF 文件的布局我们都不是清楚的，比如代码段的位置，数据段的位置我们都不知道。但在内核编译的过程中，很多内容我们都需要将它放到固定的位置，比如内核的入口函数地址（清楚了入口地址我们才能跳到这里去运行内核代码），栈堆地址等等。因此，我们需要自己制定规则，去布置各个段在 ELF 文件中的位置，这就是链接器脚本的功能。

链接器脚本的书写是一个繁琐的过程，我们已经在大家以后的代码框架中都准备好了链接器脚本，配合 Makefile 一起使用，直接使用就可以了。如果想要在链接过程中使用自己写的链接器脚本（.lds 文件），可以在 ld 命令中加入-T 参数指定。

此外，在汇编或者 C 代码中，链接器脚本里面定义的符号是可以直接引用的，例如表示数据段起始位置的符号 ___DATA_BEGIN___。但是请注意，只有这个符号的地址是有意义的，因为这个地址里面并没有存放什么有意义的值。所以在汇编中，我们可以使用 la 指令来加载其地址；在 C 语言中，可以先声明 extern 之后再使用 & 获得其地址。

7.6 objdump

由于内核要在没有其他软件辅助的情况下管理机器底层的资源，需要最终生成的目标代码非常准确，很多时候 C 语言未必能精准地描述这些操作，在 C 语言编译、链接的过程中可能会产生各种各样的不确定性，我们有时需要去检查生成的目标文件中的指令是否符合预期，但是它本身是个二进制文件，无法以文本的形式展现出来，我们可以使用 xxd 命令来以 16 进制的文本打印这个文件：

```
1 $ xxd hello
```

屏幕上会看到如图P0-8的输出。

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 6010 0000 0000 0000  ..>.....`.......
00000020: 4000 0000 0000 0000 b839 0000 0000 0000  @........9......
00000030: 0000 0000 4000 3800 0d00 4000 1f00 1e00  ....@.8...@.....
00000040: 0600 0000 0400 0000 4000 0000 0000 0000  ........@.......
00000050: 4000 0000 0000 0000 4000 0000 0000 0000  @.......@.......
00000060: d802 0000 0000 0000 d802 0000 0000 0000  ................
00000070: 0800 0000 0000 0000 0300 0000 0400 0000  ................
00000080: 1803 0000 0000 0000 1803 0000 0000 0000  ................
00000090: 1803 0000 0000 0000 1c00 0000 0000 0000  ................
000000a0: 1c00 0000 0000 0000 0100 0000 0000 0000  ................
000000b0: 0100 0000 0400 0000 0000 0000 0000 0000  ................
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000d0: f805 0000 0000 0000 f805 0000 0000 0000  ................
000000e0: 0010 0000 0000 0000 0100 0000 0500 0000  ................
000000f0: 0010 0000 0000 0000 0010 0000 0000 0000  ................
```

图 P0-8: xxd 命令的输出

从这里你可以看到这个文件的全貌了，左侧是以 16 进制打印的内容，右侧是试图

以文本方式读取 16 进制的字符。能够基本看出不同的地址处有没有数据、是不是全 0，以及链接以后各个段的排布，在检查 ELF 文件排版的时候是很有用的。

如果想更细致地调试，这样的输出仍然十分冗长、不具备可读性，没有被逐条翻译成指令。objdump 反汇编工具能够帮助我们将目标文件 (.o 或者可执行文件) 翻译回汇编语言，从而能逐条指令地检查代码正确性，是我们定位问题的有力手段。

例如，如果我们想反汇编一个可执行文件 hello 包含代码的部分，可以输入命令：

```
$ objdump -d hello > hello.S
```

表示将反汇编出的汇编结果输出到 hello.S 文件中，能看到不同的段、不同的函数下具体的指令。使用不同的编译工具链中的 objdump 以反汇编不同指令集下的目标代码。进一步，如果在编译时加入了-g 选项，表示目标文件会包含调试信息，那么反汇编时可以使用-S 参数将反汇编结果和 C 代码对应起来，以及-l 参数将结果与源文件和行号对应起来。例如：

```
$ gcc -c hello.c -g -o hello.o
$ objdump -S -l hello.o > hello.S
```

如果只想看特定某个段的内容，还可以加入-j <section> 选项来指定。以上是最常用的 objdump 命令的使用方法。

8 QEMU 和 gdb 调试

本节将介绍本课程的软件模拟器调试工具 QEMU 模拟器，以及基于 QEMU 模拟器的 gdb 调试基本技巧。在我们提供的虚拟机环境中已经安装了可以完全模拟开发板功能的 QEMU 模拟器以及 gdb 工具链，大家可以善用于日常的开发调试中。

8.1 QEMU 的启动

同学们可以使用我们提供的写好的启动脚本来启动 QEMU 模拟器，QEMU 启动脚本的路径位于：/home/stu/OSLab-RISC-V/run-qemu.sh。使用 sh run-qemu.sh 这样的命令就可以启动 QEMU 了。

另外，QEMU 为了模拟开发板的从 sd 卡加载和启动操作系统的流程，也模拟了类似的 USB 设备，所以我们后续的实验调试之前，需要先制作一个足够大的空的 USB 镜像文件。制作镜像文件的命令是：dd if=/dev/zero of=disk bs=512 count=1M，这里的 disk 为命令的输出，也就是制作好的空镜像文件，因此请大家注意文件路径。bs=512 为设备文件的块大小是 512Byte，count=1M 代表一共 1M 个块，因此设备文件的总大小为 512MB，这个大小同学们可以随意调整，但是要足够大。

QEMU 的退出需要使用 ctrl+a x 这样的组合命令，注意是 ctrl+a 先一起按下去，然后按 x，就可以看到 QEMU 模拟器被关闭，退回到 Linux 系统命令行。请大家注意不要随意使用其他的方法退出 QEMU，可能会导致下一次 QEMU 启动失败。

8.2 gdb 调试

gdb 是功能强大的代码调试工具，RISC-V 版本的 gdb 命令为 riscv64-unknown-linux-gnu-gdb，已安装在我们的虚拟机环境中。输出该命令即可启动 gdb。

在 gdb 命令行，输入 target remote localhost:1234 即可与 QEMU 模拟器连接，连接成功之后 QEMU 模拟器的运行被暂停，需要在 gdb 这边手动继续运行。注意使用 target remote 连接 QEMU 之前请确保 QEMU 模拟器已经启动。通过 symbol-file main 命令可以载入符号表，main 为编译出来的文件（要求 gcc 编译时加上-g 选项）

gdb 的一些常用命令：

设置断点：b，后面跟上内存地址或代码中的行数，例：b *0xa0800000

继续运行：c

单步运行（单条汇编指令）：si

查看当前寄存器内容：i r

查看内存内容：x，命令格式：x/nfu [addr]

n 是内存单元个数，f 是显示格式，u 是内存单元大小

显示指定地址之后的 10 条汇编指令：x/10i addr

显示指定地址之后的 10 条数据单元：x/10x addr

退出：q

以上只是一些基础的命令和使用例子，请大家多自己搜索并使用 gdb 的各种功能，思考调试思路。

9 附录：开发板内部细节

本节将描述我们的实验平台的一些细节，帮助大家理解前面的一部分可能会使大家感到困惑的地方。

9.1 概述

PYNQ-Z2 上并没有真实的 RISC-V 芯片。它是一个载有 ARM 核和 FPGA 芯片的开发板。既然 PYNQ 上面是个 ARM，那么，我们为什么可以在其上使用 RISC-V 呢? 秘诀就在于它带的 FPGA 芯片。FPGA 是可编程门控阵列的简称，可以用来模拟各种数字电路。我们向 SD 卡的第一个分区拷贝了 BOOT.BIN 文件。该文件包含了一个 ARM 上的程序和 RISC-V 软核。开发板启动后，首先会运行 BOOT.BIN。BOOT.BIN 中的 ARM 程序会将 RISC-V 软核烧到 FPGA 芯片上，然后启动 RISC-V 核心。RISC-V 核心启动后，会执行 BBL，它会帮我们把 SD 卡上我们自己写的 bootloader(也就是第三个分区的头 512 字节) 载入到内存的指定位置，并将控制权移交，从而完成启动的过程。自行制作 SD 卡的流程参见9.2小节。

9.2 制作可启动的 SD 卡

在拿到裸的开发板和 SD 卡以后，需要将 SD 卡格式化为三个分区。第一个分区固定为 34MB，采用 fat32 文件系统；第二个分区建议设定为 100 MB，采用 ext4 文件系

统；剩余空间全部划入第三个分区，保持第三个分区为空分区。

```
// 非 root 用户请在前面加`sudo `
# fdisk /dev/sdb

Welcome to fdisk (util-linux 2.34).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help):
```

注意，如果你使用的 Linux 设置了中文，可能 fdisk 会输出中文提示信息。不影响操作。另外，根据机器上磁盘配置情况的不同，你可能需要自己选择正确的磁盘。比如，这里示例是/dev/sdb，但在你的机器上可能是/dev/sda 或/dev/sdc。sd 代表磁盘，一般的机械硬盘和 U 盘都会被列入这里。如果你的机器（或虚拟机）有一块机械硬盘，则新插入的 SD 卡可能是/dev/sdb（因为 sda 一般是机器自己的机械硬盘）。一个简单的方法是，先执行 ls /dev/sd* 命令，然后插上 U 盘再执行一遍这个命令，看多出来的是哪个。比如插上 U 盘后多出来了一个/dev/sdb，那么显然，U 盘就是这个/dev/sdb。

创建第一个分区的过程如下：

```
Command (m for help): o
Created a new DOS disklabel with disk identifier 0xae2c98d2.

Command (m for help): n
Partition type
   p   primary (0 primary, 0 extended, 4 free)
   e   extended (container for logical partitions)
Select (default p):

Using default response p.
Partition number (1-4, default 1):
First sector (2048-7626751, default 2048):
Last sector, +/-sectors or +/-size{K,M,G,T,P} (2048-7626751, default 7626751): +34M

Created a new partition 1 of type 'Linux' and of size 34 MiB.

Command (m for help): n
Partition type
   p   primary (1 primary, 0 extended, 3 free)
   e   extended (container for logical partitions)
Select (default p):

Using default response p.
Partition number (2-4, default 2):
First sector (71680-30253055, default 71680):
Last sector, +/-sectors or +/-size{K,M,G,T,P} (71680-30253055, default 30253055): +100M

Created a new partition 2 of type 'Linux' and of size 100 MiB.
```

在 fdisk 提示输入命令的时候按上面的示例输入。先输入 o 新建分区表，再输入 n 新建新分区，然后都默认就可以，只是大小必须为 34M。随后再输入 n 新建分区，然后默认，并设置大小为 100M。

接下来，创建第三个分区，并把第一个分区类型改为 fat32, 第二个分区改为 Linux，第三个分区设置为 empty。最后按 w 写入并退出 fdisk。

```
Command (m for help): n
Partition type
   p   primary (2 primary, 0 extended, 2 free)
   e   extended (container for logical partitions)
Select (default p):

Using default response p.
Partition number (3,4, default 3):
First sector (276480-30253055, default 276480):
Last sector, +/-sectors or +/-size{K,M,G,T,P} (276480-30253055, default 30253055):

Created a new partition 3 of type 'Linux' and of size 14.3 GiB.

Command (m for help): t
Partition number (1,2, default 2): 1
Hex code (type L to list all codes): b

Changed type of partition 'Linux' to 'W95 FAT32'.

Command (m for help): t
Partition number (1,2, default 2): 2
Hex code or alias (type L to list all): 83

Changed type of partition 'Linux' to 'Linux'.

Command (m for help): t
Partition number (1-3, default 3): 3
Hex code or alias (type L to list all): 0
Type 0 means free space to many systems. Having partitions of type 0 is probably unwise.

Changed type of partition 'Linux' to 'Empty'.

Command (m for help): w
```

创建好分区后，首先需要为第一个分区制作文件系统。制作方法是用 mkfs.vfat 格式化第一个分区：

```
# sudo mkfs.vfat -I /dev/sdb1 -n "BOOT"
```

随后为第二个分区制作 ext4 文件系统，制作方法是使用 mkfs.ext4

```
# sudo mkfs.ext4 -F /dev/sdb2 -L "ROOTFS"
```

以上两个分区的制作也请注意需要根据你自己的磁盘情况调整 sdb 这个参数。

制作完以后，将预先提供的 BOOT.BIN、boot.scr、image.ub 拷入到第一个分区；rootfs.tar.gz 压缩包的内容解压到第二个分区，然后插入到开发板上，上电后就可以看到效果了。

Note 9.1 本材料中出现的 shell 中的 $ 和 # 都代表命令行前面的提示符。$ 代表普通用户，# 代表需要 root 权限。root 权限可以用 sudo -i 获得，或者在命令前面

加 sudo。例如，如果想以 root 权限执行 ls, 则可以输入 sudo ls。前面没加提示符的内容代表命令的执行结果。

9.3 地址空间情况

地址空间情况如表P0-10所示。其中，最需要注意的是，BBL 所需的内存空间放置了 BBL 运行所需的数据和代码。请一定不要修改这段内存。BBL 为我们提供了读写 SD 卡和输出字符串的相关服务。如果不小心修改了它的数据或代码，可能导致相关功能异常。我们自己将要编写的内核可以使用的空间为 0x50200000-0x60000000 这一段地址。

地址范围	权限	作用
0x0-0x1000	ARWX	debug-controller
0x3000-0x4000	ARWX	error-device
0x10000-0x20000	RX	rom
0x2000000-0x2010000	ARW	clint
0xc000000-0x10000000	ARW	interrupt-controller
0xe0000000-0xe0001000	RWX	serial
0xe000b000-0xe000c000	RWX	ethernet
0xe0100000-0xe0101000	RWX	mmc
0xf8000000-0xf8000c00	RWX	SLCR
0x50000000-0x50200000	RWXC	memory(for BBL)
0x50200000-0x60000000	RWXC	memory

表 P0-10: 地址空间

另外一点需要注意的是，如果错误地读写了 0x0 或者其他非 memory 的地址，那么很有可能触发中断。由于前面的实验我们没有设置中断处理机制，所以一旦访问错误的地址，在开发板上看到的现象就是程序卡死，不再继续执行。建议在调试的时候，多使用 QEMU+gdb。或者分成小段一点一点调试，在出现内存相关的错误的情况下，试图直接找到大段代码中的错误很困难。一般应该一小段一小段逐步缩小范围，从而正确地找到错误的发生位置。

参考文献

[1] Oracle, "Virtualbox - licensing: Frequently asked questions." https://www.virtualbox.org/wiki/Licensing_FAQ, 2021. [Online; accessed 27-August-2021].

[2] A. W. David Patterson, *RISC-V Reader*. 2018. Available at http://riscvbook.com/chinese/RISC-V-Reader-Chinese-v2p1.pdf.

[3] "The risc-v instruction set manual volume i: User-level isa v2.2," 2017.

[4] A. B. Palmer Dabbelt, Michael Clark, "Risc-v assembly programmer's manual." https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md, 2019. [Online; accessed 27-August-2021].

[5] 阮一峰, "Make 命令教程." http://www.ruanyifeng.com/blog/2015/02/make.html, 2004. [Online; accessed 31-August-2021].

[6] 陈皓, "如何调试 makefile 变量." https://coolshell.cn/articles/3790.html, 2004. [Online; accessed 31-August-2021].

[7] 陈皓, "跟我一起写 makefile." https://blog.csdn.net/haoel/article/details/2886, 2004. [Online; 网友整理版: https://seisman.github.io/how-to-write-makefile/index.html].