最小可执行内核的构建与启动全流程深度验证

操作系统 lab1 实验报告

梁景铭　王一诺　强博
南开大学计算机学院

2025 年 10 月 9 日

摘要

本报告旨在对操作系统实验 Lab1 的全部内容进行一份极致详尽的记录与分析。实验的核心任务是构建一个能够在 QEMU RISC-V 模拟器上运行的最小可执行内核，并深入理解其完整的启动链。报告严格遵循“原理学习-编码实践-GDB 调试-结果验证”的科学流程，不仅记录了学习和操作步骤，同时将每一个环节与操作系统核心理论进行了深度绑定。

报告通过对 `entry.S` 的逐行剖析，阐明了内核进入 C 语言环境前手工建立栈空间的必要性。最为核心的是，报告利用 GDB 工具，完整复现了从 QEMU 加电（复位向量 `0x1000`）到 OpenSBI 固件（`0x80000000`），再到内核第一行代码（`0x80200000`）的完整“接力”过程。所有 GDB 日志和 QEMU 控制台输出均被作为一手证据收录，并配以详尽解析。

表 1: 启动阶段-地址-证据快速对照

阶段	关键地址	证据
MROM 复位	0x1000	`x/10i 0x1000` 显示 `csrr mhartid; jr t0`
OpenSBI 入口	0x80000000	`x/8i $pc`，QEMU 控制台 Next Address=0x80200000
内核入口	0x80200000	`x/8i $pc` 见 `auipc/mv`（`la sp,...` 等价）
C 入口	`kern_init`(0x8020000a)	`disassemble` 见 `memset; cprintf; j .`

目录

1 实验任务分工

本次实验不涉及代码编写，核心在于环境配置与 GDB 调试。因此，小组成员均独立完成了全部实验内容，以确保个人对启动流程有扎实的理解。在此基础上，全体成员共同对实验难点进行研讨，并协作完成了本报告的撰写与答辩准备工作。

2 实验内容与目标

2.1 实验内容概述

本次实验的核心任务是构建一个最小化的可执行内核，并深入理解其在 QEMU RISC-V 模拟器上的完整启动流程。这要求我们必须掌握内核与 QEMU 平台的对接机制，包括程序的内存布局、交叉编译链接过程，以及如何通过 OpenSBI 固件提供的底层服务与硬件进行交互。

2.2 核心学习目标

通过本次实验，我们将掌握以下关键技能：

- **使用链接脚本**: 学习如何精确描述内核的内存布局。
- **进行交叉编译**: 掌握生成内核可执行文件及镜像的全过程。
- **使用 OpenSBI 与 QEMU**: 理解 Bootloader 加载内核的原理并进行模拟。
- **调用 SBI 服务**: 学会在裸机环境下通过固件服务在屏幕上进行格式化输出，为后续调试打下基础。

3 ucore 实验文档学习

3.1 现代计算机启动流程对比

本次实验中的“OpenSBI → 内核”启动流程，是真实世界计算机启动过程的一个精炼缩影。现代笔记本电脑（通常是 x86 架构）的启动虽然更复杂，但其核心思想是共通的，同样遵循一个“接力”模式：

1. **UEFI 固件** : 当按下电源键，CPU 首先执行的并非硬盘上的操作系统，而是主板芯片上固化的 UEFI 固件。它扮演了类似 OpenSBI 的角色，负责硬件自检、初始化基本硬件，并作为第一级引导程序。
2. **引导加载程序**: UEFI 根据预设的启动顺序，从硬盘的一个特殊分区（EFI 系统分区）加载并执行第二级引导加载程序，如 GRUB (用于 Linux) 或 Windows Boot Manager。
3. **操作系统内核**: 最后，由 GRUB 或 Windows Boot Manager 负责将完整的操作系统内核（如 `vmlinuz` 或 `ntoskrnl.exe`）加载到内存，并将控制权移交给它。

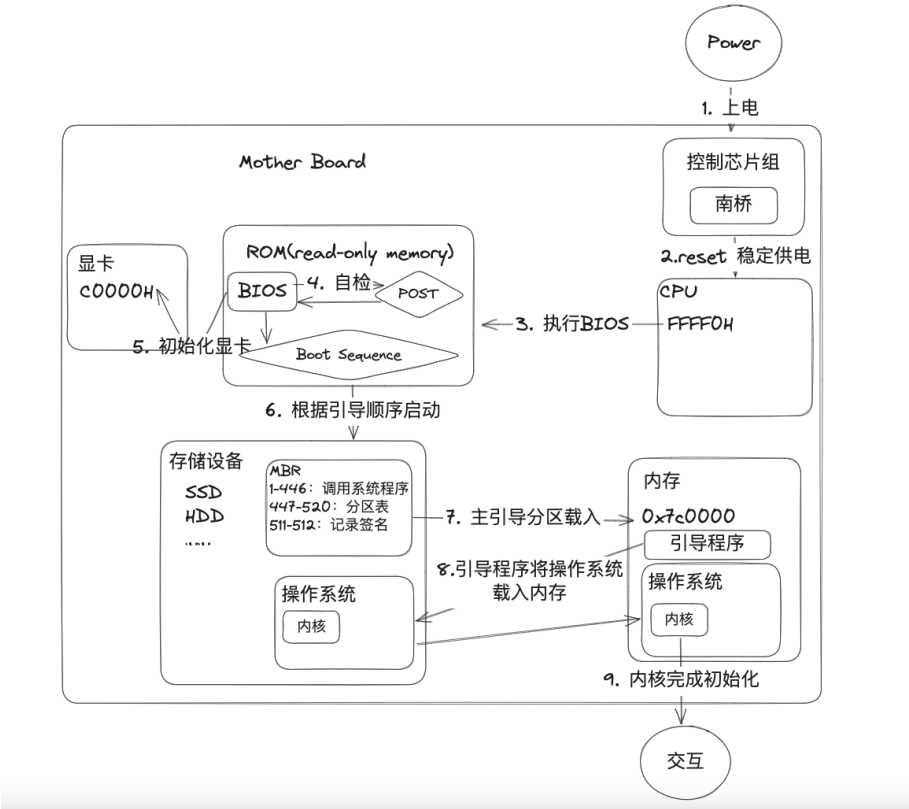

图 1: 现代计算机启动流程

对比结论: 无论是实验中的 "**OpenSBI → 内核**"，还是现代 PC 的 "**UEFI → GRUB → Linux 内核**"，其本质都是一个从底层固件到高级操作系统的、分工明确、层层递进的引导链。理解了实验中的简化模型，就 grasp 了所有现代操作系统启动的共通设计哲学。

3.2 内核执行流详解

最小可执行内核的完整启动流程可以概括为一个清晰的、线性的 "接力" 过程:

加电复位 → CPU 从 0x1000 进入 MROM → 跳转到 0x80000000(OpenSBI) → OpenSBI 初始化并加载内核到 0x80200000 → 跳转到 entry.S → 调用 kern_init() → 输出信息 → 结束

整个执行流可以精确地划分为以下三个核心阶段:

第一步: 硬件初始化与固件启动 QEMU 模拟器启动后，会首先模拟计算机的加电复位过程。此时，处理器的程序计数器（PC）被硬件强制设置为一个固定的复位地址 0x1000。CPU 从该地址开始执行一小段被固化在机器只读存储器（MROM, Machine ROM）中的代码。这段代码的功能非常基础，其核心任务是完成最基本的硬件环境准备（例如设置初始的栈指针或寄存器状态），然后将控制权无条件地移交给下一阶段的引导加载程序——OpenSBI。为了实现这一跳转，OpenSBI 固件的二进制镜像已被预先加载到物理内存的 0x80000000 处。

第二步：OpenSBI 初始化与内核加载 CPU 跳转到 0x80000000 地址后，开始执行 OpenSBI 固件。OpenSBI 运行在 RISC-V 架构的最高特权级——机器模式（M-Mode），这使其拥有对所有硬件资源的完全访问权限。在此阶段，OpenSBI 会负责一系列关键的初始化工作，包括但不限于：设置中断和异常的委托机制、初始化定时器、以及配置物理内存保护等（可以说是自检吧）。在完成这些必要的底层设置后，OpenSBI 的核心使命便是准备加载并启动操作系统内核。它会将我们预先编译好的内核镜像文件（ucore.img）从模拟的存储设备加载到指定的物理内存地址 0x80200000 处。

第三步：内核启动执行 当 OpenSBI 完成所有初始化和加载工作后，它会执行最后一次跳转，将 PC 设置为 0x80200000，正式将计算机的控制权移交给我们的内核。由于我们在链接脚本（kernel.ld）中精确地将 entry.S 的代码段放在了内核镜像的最开始位置，因此 0x80200000 地址上存放的正是 entry.S 编译后的第一条机器指令。entry.S 作为进入内核的第一道关卡，其职责明确而关键：首先，它设置好内核栈指针（sp），为后续的 C 语言函数调用提供必须的栈空间；然后，它遵循 RISC-V 的调用约定，跳转到 C 语言编写的内核入口函数 kern_init()。最后，kern_init() 调用 cprintf() 函数，通过之前建立的 SBI 服务在控制台上输出一行信息，标志着内核已成功启动并运行，随后进入一个无限循环。

3.3 引导加载程序与文件格式

3.3.1 Bootloader 的必要性

操作系统自身也是一个程序，它无法“自己加载自己”到内存中运行，就像人不能提着自己的头发离开地面。因此，在操作系统运行之前，必须有一个先驱程序——**引导加载程序 (Bootloader)**——来完成这项工作。它的核心职责就是将操作系统内核从硬盘等非易失性存储中加载到内存，然后跳转到内核的入口点，将 CPU 的控制权交给操作系统。

3.3.2 OpenSBI：固件与引导加载程序

在本次实验的 QEMU 模拟环境中，**OpenSBI** 扮演了固件和 Bootloader 的双重角色。

- **固件**: 作为固件，它为硬件提供最底层的控制，并为上层软件（内核）提供标准化的服务接口 (SBI)。为了直接访问硬件，它运行在 RISC-V 架构的最高特权级——**机器模式 (M-Mode)**。
- **引导加载程序**: 作为 Bootloader，它负责在 QEMU 启动时，将我们的内核镜像加载到指定的内存地址 0x80200000，并最终跳转过去。

3.3.3 文件格式：ELF vs. BIN

在编译链接和引导加载的过程中，我们接触到了两种关键的文件格式：

- **ELF**: 这是由编译器和链接器生成的一种复杂、功能强大的可执行文件格式。它包含了丰富的元信息，如符号表（用于 GDB 调试）、段信息（如 .text, .data, .bss）以及它们在虚拟内存中的布局指令。对于未初始化的数据段（.bss），ELF 文件只记录其大小而不存储实际的零，从而节省了文件体积。

- **BIN**: 这是一种“扁平”的二进制镜像文件，它不包含任何元信息，仅仅是程序代码和数据的原始字节流。Bootloader 更喜欢这种简单的格式，因为它只需将文件内容原封不动地复制到内存的指定位置即可。与 ELF 不同，BIN 文件会包含 `.bss` 段的所有零字节，因此文件通常更大。

我们的编译流程遵循 **ELF → BIN** 的转换：首先，通过交叉编译工具链生成一个包含所有调试和链接信息的 ELF 文件 (`bin/kernel`)；然后，使用 `objcopy` 工具从中提取出纯粹的程序镜像，生成一个扁平的 BIN 文件 (`ucore.img`)，以供 OpenSBI 加载。

3.3.4 代码的地址相关性

内核必须被加载到链接脚本中指定的 `0x80200000` 地址，这是由“地址相关性”决定的。在编译链接时，对全局变量和函数的引用会被解析为绝对地址。例如，一条指令可能会被编译为‘load r1, 0x80201234‘。如果内核被加载到其他地址，这条指令就会访问错误的位置，导致程序崩溃。因此，Bootloader 和内核之间必须遵守严格的地址约定。

3.4 内存布局、链接脚本与程序入口点

3.4.1 程序内存分段

一个编译后的程序在内存中并非浑然一体，而是按照功能划分为不同的“段”（Section），主要包括：

- `.text`: 代码段，存放编译后的机器指令，通常是只读和可执行的。
- `.rodata`: 只读数据段，存放常量字符串等只读数据。
- `.data`: 已初始化数据段，存放程序中已赋初值的全局变量和静态变量。
- `.bss`: 未初始化数据段，存放所有未赋初值或初值为零的全局变量和静态变量。在可执行文件中，该段只记录大小而不占空间，以节省体积。内核启动时必须手动将其清零。

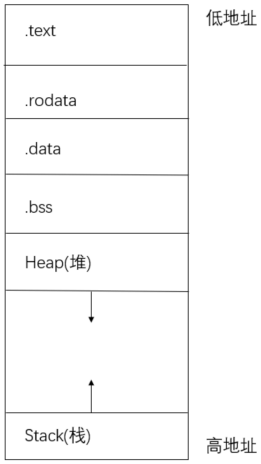

图 2: 内存布局示意图

3.4.2 链接脚本的角色与必要性

链接器（‘ld‘）的作用是将多个目标文件（‘.o‘）合并成一个最终的可执行文件（ELF）。它通过一个**链接脚本**来决定如何组织和布局这些段。对于用户态应用程序，链接器使用一个默认脚本，使其能在现有操作系统下运行。但对于操作系统内核，我们必须提供一个自定义的链接脚本，原因在于：

- **对接引导加载程序**: 内核必须被加载到引导加载程序（OpenSBI）预期的特定物理地址。
- **定义入口点**: 必须明确告诉链接器，哪一条指令是内核的第一条指令。

3.4.3 ‘kernel.ld‘ 脚本解析

我们的链接脚本 `tools/kernel.ld` 通过几个关键指令完成了上述任务：

Listing 1: tools/kernel.ld 关键指令解析

```
OUTPUT_ARCH(riscv)
ENTRY(kern_entry)

BASE_ADDRESS = 0x80200000;

SECTIONS
{
    . = BASE_ADDRESS;
    .text : {
        *(.text.kern_entry) /* 确保入口代码在最前面 */
        *(.text .stub .text.*)
    }
    ...
    .data : { ... }
    ...
    .bss : { ... }
    ...
}
```

- `BASE_ADDRESS = 0x80200000`: 定义了内核的加载基地址，与 OpenSBI 的跳转目标完全一致。
- `ENTRY(kern_entry)`: 指定 `kern_entry` 符号为程序的唯一入口点。
- `.text : { *(.text.kern_entry) ... }`: 这条规则至关重要，它指示链接器将所有输入文件中名为 `.text.kern_entry` 的段（即我们在 ‘entry.S‘ 中定义的）放在整个 `.text` 代码段的**最开始**。

这三条指令共同确保了当 OpenSBI 跳转到 `0x80200000` 时，执行的正是 ‘kern_entry‘ 的第一条指令。

3.4.4 从汇编到 C

链接脚本搭好了舞台，而 ‘entry.S‘ 和 ‘init.c‘ 则上演了从汇编到 C 的控制权交接：

1. **汇编入口 (‘entry.S‘)**: 作为物理上的第一站，它定义了 `kern_entry` 符号，并执行了两项不可或缺的任务：
   - `la sp, bootstacktop`: 为即将运行的 C 代码设置好栈指针。
   - `tail kern_init`: 以无返回的方式，将控制权彻底移交给 C 函数。
2. **C 入口 (‘init.c‘)**: 作为逻辑上的“真正”入口，`kern_init` 函数开始执行内核的核心初始化任务。例如，它使用链接脚本提供的符号 `edata` 和 `end` 来定位并清零 `.bss` 段：

```
1 // extern char edata[], end[]; 声明由链接器定义的符号
2 memset(edata, 0, end - edata); // 清零 .bss 段
```

这个从链接脚本 → 汇编入口 → C 入口的精密配合，构成了内核自举并进入高级语言环境的基础。

3.5 自底向上构建输出：从 SBI 到 cprintf

3.5.1 问题的提出：“鸡生蛋”的困境

在 ‘init.c‘ 中，我们希望调用一个类似 ‘printf‘ 的函数来输出启动信息。然而，在裸机环境下，我们不能简单地 ‘#include <stdio.h>‘。因为标准库函数（如 ‘glibc‘ 中的 ‘printf‘）本身依赖于底层操作系统提供的系统调用来完成 I/O 操作。在一个正在开发的操作系统内核中依赖另一个操作系统，这在逻辑上是矛盾的。因此，我们必须“自力更生”，从最底层构建自己的输出能力。

3.5.2 OpenSBI 与 ecall

解决问题的起点是运行在 M-Mode 的 OpenSBI 固件。它通过 SBI 标准，为运行在 S-Mode 的内核提供了一系列底层服务，其中就包括“在控制台输出一个字符”的功能。

- **服务请求机制**: 内核通过执行 `ecall` 指令来请求这些服务。当 S-Mode 代码执行 `ecall` 时，会触发一个受控的陷阱，使 CPU 切换到 M-Mode 并跳转到 OpenSBI 的处理程序。
- **SBI 调用约定**: 为了让 OpenSBI 知道需要哪种服务，我们必须遵循 RISC-V SBI 的调用约定，即在执行 `ecall` 前，将参数设置到指定的寄存器中（例如，‘a7‘ 存放功能号，‘a0‘-‘a2‘ 存放参数）。

3.5.3 内联汇编封装

由于 C 语言本身无法直接执行 `ecall` 指令或精确控制寄存器，我们必须借助内联汇编来搭建一座桥梁。‘sbi_call‘ 函数就是这个桥梁的核心：

Listing 2: libs/sbi.c: 通用 SBI 调用封装

```
uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, ...) {
    uint64_t ret_val;
    __asm__ volatile (
        "mv x17, %[sbi_type]\n" // a7 = sbi_type
        "mv x10, %[arg0]\n"     // a0 = arg0
        ...
        "ecall\n"              // 切换到 M-Mode
        "mv %[ret_val], x10"    // ret_val = a0 (返回值)
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), ...
        : "memory"
    );
    return ret_val;
}

void sbi_console_putchar(unsigned char ch) {
    sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0);
}
```

通过这个封装，我们得到了一个 C 函数 `sbi_console_putchar`，它能够输出单个字符。

3.5.4 从单个字符到格式化字符串

有了输出单个字符的能力，我们就可以像搭积木一样，逐层构建更高级的功能：

1. **驱动层 (‘console.c‘)**: 提供一个简单的驱动接口 ‘cons_putc‘,它直接调用 ‘sbi_console_putchar‘。
2. **基础 I/O 库 (‘stdio.c‘)**: 在驱动层之上，实现更通用的函数，如 ‘cputchar‘ (输出单个字符) 和 ‘cputs‘ (输出字符串)。
3. **格式化库 (‘printfmt.c‘)**: 实现 ‘cprintf‘ 的核心逻辑，即解析格式化字符串（如 ‘%d‘, ‘%s‘），处理可变参数，并循环调用 ‘cputchar‘ 来输出最终的字符序列。

这个 `cprintf` → `cputs/cputchar` → `cons_putc` → `sbi_call` → `ecall` 的调用链，完美展示了软件工程中分层抽象的设计思想，使我们能在 “一无所有” 的环境中，最终拥有一个功能强大的格式化输出函数。

3.6 Make 与 Makefile

3.6.1 自动化构建的必要性

编译和运行一个内核涉及一系列复杂的步骤：编译所有 C 和汇编源文件、将目标文件链接成一个 ELF 可执行文件、从中提取出二进制镜像、最后启动 QEMU 模拟器并加载该镜像。手动执行这些命令不仅繁琐且容易出错。为此，我们使用 ‘make‘ 工具来自动化整个流程。

3.6.2 Make 与 Makefile 的工作原理

‘make‘ 是一个强大的构建自动化工具，它根据一个名为 ‘Makefile‘ 的规则文件来工作。‘Makefile‘ 的核心是定义文件之间的依赖关系和生成规则。其基本语法如下：

```
target: prerequisites
    command
```

- **target**: 目标文件，如 ‘kernel.elf‘ 或 ‘ucore.img‘。
- **prerequisites**: 生成目标文件所依赖的源文件或中间文件。
- **command**: 生成目标文件的 Shell 命令。

当执行 ‘make target‘ 时，‘make‘ 会检查 ‘prerequisites‘ 是否比 ‘target‘ 更新。如果是，它就会执行 ‘command‘ 来重新生成 ‘target‘。这种智能的增量构建机制极大地提高了开发效率。

3.6.3 编译与运行 ucore

在本项目中，我们只需执行一个简单的命令，即可完成从编译到运行的全过程：

```
$ make qemu
```

这个命令会触发 ‘Makefile‘ 中一系列的依赖规则，最终执行 QEMU 启动命令，并将我们的内核加载进去。‘Makefile‘ 中相关的 QEMU 启动命令如下：

```
qemu: $(UCOREIMG)
    $(QEMU) \
        -machine virt \
        -nographic \
        -bios default \
        -device loader,file=$(UCOREIMG),addr=0x80200000
```

执行 ‘make qemu‘ 后，终端会显示编译、链接和运行的全过程，最终输出内核的启动信息：

Listing 3: make qemu 的执行输出

```
+ cc kern/init/init.c
+ cc libs/sbi.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img

OpenSBI v0.6
   ____                   _____ ____ _____
  / __ \                 / ____|  _ \_   _|
...
(THU.CST) os is loading ...
```

这个输出清晰地展示了从源文件编译、链接生成 ‘bin/kernel‘ (ELF)，到转换成 ‘ucore.img‘ (BIN)，再到 QEMU 启动 OpenSBI 并最终运行我们内核的全过程。

4 实验练习解答

4.1 练习 1：理解内核启动中的程序入口操作

4.1.1 问题重述

阅读kern/init/entry.S内容代码,结合操作系统内核启动流程,说明指令`la sp, bootstacktop`完成了什么操作，目的是什么？`tail kern_init`完成了什么操作，目的是什么？

4.1.2 源码定位

Listing 4: kern/init/entry.S 关键代码

```
.section .text,"ax",%progbits
.globl kern_entry
kern_entry:
    la sp, bootstacktop
    tail kern_init
```

4.1.3 指令深度解析

`la sp, bootstacktop`

- **完成的操作**: 这是一条RISC-V伪指令,其全称为“加载地址”。它的作用是将符号`bootstacktop`的绝对内存地址计算出来，并加载到栈指针寄存器`sp`(即`x2`)中。在底层，汇编器会将其展开为一条或两条实际的机器指令，以实现PC相对的地址加载。
- **目的与原理**:
  1. **为C环境建立基础**: C语言的运行严重依赖于一个合法的栈。函数调用时，返回地址、调用者保存的寄存器以及函数自身的局部变量都需要在栈上分配空间。在OpenSBI跳转到我们的内核入口时，虽然CPU已经处于S-Mode，但`sp`寄存器的值是未知的或继承自前一阶段，不能直接用于C函数。
  2. **初始化内核栈**: 此指令的目的就是在进入任何C代码之前，手动建立起内核的第一个栈。‘bootstacktop‘是我们在链接脚本和汇编中预先定义好的、一块静态分配的内存区域的顶部（高地址）。将`sp`指向它，后续的C函数调用就可以在这片内存上安全地创建和销毁栈帧了。

`tail kern_init`

- **完成的操作**: 这同样是一条伪指令，表示“尾调用”。它的作用是执行一个无条件的跳转，直接将程序计数器(PC)设置为`kern_init`函数的地址。与常规的‘call‘或‘jal‘指令不同，它**不会**在返回地址寄存器‘ra‘(即‘x1‘)中保存下一条指令的地址。
- **目的与原理**:

1. **彻底移交控制权**: `entry.S` 的使命在设置好栈之后就已完成。其存在的唯一目的就是将环境配置好，然后把舞台完全交给 C 代码。使用 'tail' 跳转，明确地表示了这种 "永不返回" 的意图。
2. **优化调用栈**: 因为没有保存返回地址，所以当 `kern_init` 函数返回时（尽管在本实验中它不会返回），它不会回到 `entry.S`，而是会返回到调用 `entry.S` 的地方（即 OpenSBI）。这避免了在调用栈上创建一个无用的栈帧，使得调用关系更清晰，也略微节省了资源。
3. **定义逻辑入口**: 此操作将 `kern_init` 确立为内核的**逻辑入口点**，而 `kern_entry` 则是**物理入口点**。物理入口负责最底层的硬件准备，逻辑入口负责高层的软件初始化。

4.2 练习 2：使用 GDB 验证启动流程

4.2.1 问题重述

使用 GDB 跟踪 QEMU 模拟的 RISC-V 从加电开始，直到执行内核第一条指令的整个过程。通过调试，请思考并回答：RISC-V 硬件加电后最初执行的几条指令位于什么地址？它们主要完成了哪些功能？

4.2.2 调试环境与流程

- **终端 A (被调试目标)**: 执行 'make debug'，启动 QEMU 并使其在端口 1234 等待 GDB 连接。
- **终端 B (调试器)**: 执行 'make gdb'，启动 'gdb-multiarch' 并自动连接到 QEMU，同时加载 'bin/kernel' 的符号。

4.2.3 GDB 跟踪与观察记录

阶段一：硬件复位与 MROM @ `0x1000` GDB 成功连接后，程序立即暂停。我们查看此时的 PC 和内存，确认了硬件的起点。

Listing 5: GDB: 连接并查看复位向量

```
(gdb) target remote :1234
Remote debugging using :1234
0x0000000000001000 in ?? ()
(gdb) x/10i 0x1000
=> 0x1000:      auipc   t0,0x0
   0x1004:      addi    a2,t0,40
   0x1008:      csrr    a0,mhartid
   0x100c:      ld      a1,32(t0)
   0x1010:      ld      t0,24(t0)
   0x1014:      jr      t0
   0x1018:      unimp
   ...
```

观察与结论: GDB 显示 PC 的初始值为 0x1000。此地址上的代码是固化在 MROM 中的复位序列，负责读取 CPU ID (‘mhartid‘) 等最基础的初始化，并最终通过 ‘jr t0‘ 跳转到下一阶段的引导程序。

阶段二：OpenSBI 固件接管 @ 0x80000000 我们在 OpenSBI 的标准入口地址设置断点，以观察控制权的第一次主要移交。

Listing 6: GDB: 在 OpenSBI 入口中断

```
(gdb) b *0x80000000
Breakpoint 1 at 0x80000000
(gdb) c
Continuing.

Breakpoint 1, 0x0000000080000000 in ?? ()
(gdb) x/8i $pc
=> 0x80000000:  add     s0,a0,zero
   0x80000004:  add     s1,a1,zero
   0x80000008:  add     s2,a2,zero
   0x8000000c:  jal     0x80000580
   ...
```

同时，QEMU 控制台输出的 OpenSBI 日志也证实了它已接管，并指明了下一个目标:

Listing 7: QEMU 控制台输出：OpenSBI 启动信息

```
OpenSBI v1.3
Firmware Base             : 0x80000000
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Mode         : S-mode
```

观察与结论: 控制权成功转移到 0x80000000，OpenSBI 开始在 M-Mode 下执行。它会完成一系列平台初始化，并根据配置，准备将控制权移交给位于 0x80200000 的 S-Mode 内核。

阶段三：内核入口与关键验证 @ 0x80200000 最后，我们在内核的物理入口点 kern_entry 设置断点，见证内核的诞生。

Listing 8: GDB: 在内核入口中断并验证栈初始化

```
(gdb) b *kern_entry
Breakpoint 2 at 0x80200000: file kern/init/entry.S, line 7.
(gdb) c
Continuing.

Breakpoint 2, kern_entry () at kern/init/entry.S, line 7.
7           la sp, bootstacktop
(gdb) x/8i $pc
```

```
=> 0x80200000 <kern_entry>:     auipc   sp,0x3
   0x80200004 <kern_entry+4>:   mv      sp,sp
   0x80200008 <kern_entry+8>:   j       0x8020000a <kern_init>
(gdb) info address bootstacktop
Symbol "bootstacktop" is at 0x80203000 in a file compiled without debugging
    .
(gdb) si
0x0000000080200004      7           la sp, bootstacktop
(gdb) info registers sp
sp             0x80203000       0x80203000 <bootstacktop>
```

观察与结论: 程序准确地停在了 0x80200000 的 kern_entry。单步执行第一条指令后，sp 寄存器的值被成功设置为 0x80203000，与符号 bootstacktop 的地址完全一致。这证明了内核栈已成功建立。

4.2.4 问题解答

- **RISC-V 硬件加电后最初执行的几条指令位于什么地址？**
  答：最初的指令位于物理地址 0x1000 附近。这是 QEMU 模拟的 RISC-V 'virt' 平台的硬件复位向量地址。而最初级条指令就位于第一阶段，让我们逐行解读：

Listing 9: GDB: 加电后前几行指令

```
(gdb) x/10i 0x1000
0x 0x1000:  auipc   t0,0x0      # t0 = PC + 0x0000 = 0x1000
0x 0x1004:  addi    a1,t0,32    # a1 = t0 + 32 = 0x1020 (设备树地址参数)
0x 0x1008:  csrr    a0,mhartid  # a0 = mhartid (读取硬件线程ID)
0x 0x100c:  ld      t0,24(t0)   # t0 = *(0x1000 + 24) = *(0x1018)
0x 0x1010:  jr      t0          # 跳转到 t0 指向的地址
0x1014:  unimp
0x1016:  unimp
0x1018:  unimp
0x101a:  0x8000
0x101c:  unimp
```

可以分析这几行指令的主要功能：

1. **计算设备树地址 @ 0x1000-0x1004**: 使用 auipc 获取当前 PC 值到 t0，然后计算设备树（DTB）的地址，存入 a1 寄存器，为后续的 OpenSBI 初始化准备参数。
2. **读取硬件线程 ID @ 0x1008**: 从 mhartid 读取当前硬件线程的 ID，存入 a0 寄存器，作为参数传递给后续代码。
3. **获取跳转目标地址 @ 0x100c**: 从地址 0x1018 处加载一个地址值，根据调试信息，0x101a 处的值是 0x8000，实际跳转地址应该是 0x80000000（OpenSBI 的入口）。

4. **跳转到 OpenSBI @** `0x1010`: 执行 jr t0，跳转到 OpenSBI 固件的入口地址，将控制权移交给 OpenSBI 进行系统初始化。

可以总结，这段代码是 QEMU 内置的最小化启动代码 ROM Bootloader，主要职责是：准备调用参数（hart ID 和设备树地址）并跳转到真正的固件 OpenSBI。

- **它们主要完成了哪些功能？**
  答：整个启动过程主要完成了三个阶段的功能传递：

  1. **MROM @** `0x1000`: 执行最基础的、固化在硬件中的初始化代码，其唯一目标是将控制权移交给功能更全面的固件。
  2. **OpenSBI @** `0x80000000`: 作为运行在 M-Mode 的固件和引导加载程序，它负责初始化整个硬件平台，加载操作系统内核到指定内存位置，并为内核提供底层的运行时服务（SBI）。
  3. **内核 @** `0x80200000`: 在 OpenSBI 完成使命后，它最终跳转到内核的入口点，将系统的控制权彻底交给操作系统。

5 实验与理论对应

5.1 实验中出现相关知识点

- **系统上电执行流程**
  实验课上讲解的系统上电的过程涉及到底层硬件架构，实验过程的环境是虚拟化和架构化的。

- **内存布局与程序分段**

5.2 实验中未涉及相关知识点

- **进程管理和调度**
  主要包含进程控制块、进程状态转换、进程调度算法等相关知识点，实验一关注的是系统启动的过程。

- **虚拟内存管理**

6 实验所遇问题

在实验过程中，组员遇到了 make 失败的问题：

6.1 问题描述

在 riscv64-unknown-elf-gcc 和 opensbi 都验证正常的情况下，终端运行 make qemu 返回了以下错误信息：

Listing 10: 终端: make qemu 报错

```
+ cc kern/init/entry.S
Assembler messages:
Fatal error: invalid -march= option: `rv64imafodc'
make: *** [Makefile:112: obj/kern/init/entry.o] Error 1
```

6.2 问题解决

与助教交流后尝试修改 makefile 和.bashrc 均未改正，最终在助教的建议下重装环境。重装环境后仍然 make 失败，怀疑是交叉编译工具链文件缺损。这时的工具链是从浏览器网站上下载后转移到 Ubuntu 系统文件夹里并解压的，很有可能在这个下载-转移-解压的过程中发生了缺损。于是尝试改用终端命令下载与解压，完成后尝试 make qemu，成功执行，问题解决。