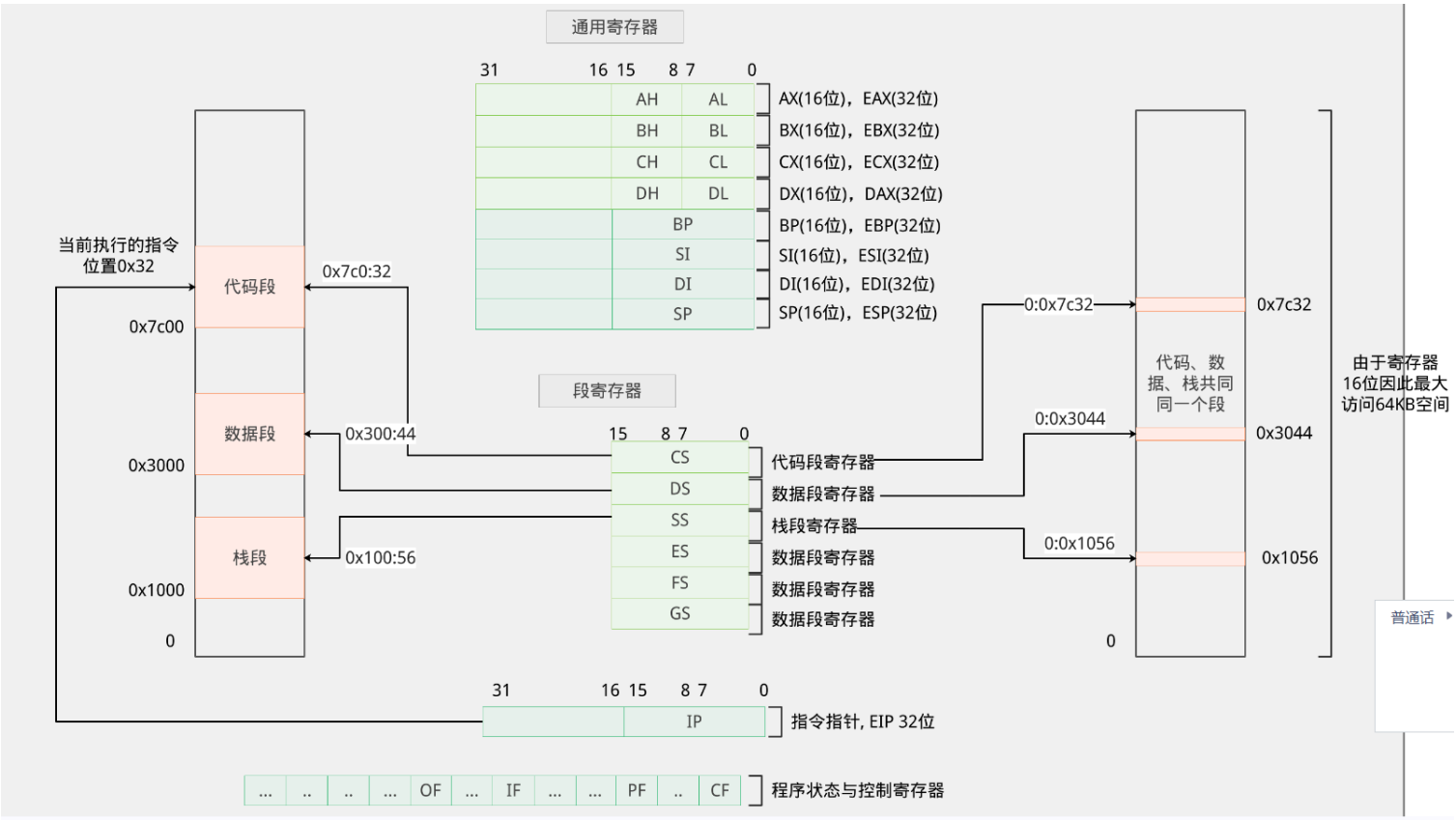
1.boot实现

采用了二级引导boot只完成loader的加载工作，再由loader完成具体的初始化工作和内核加载。

实模式

CPU启动后，自动进入所谓的实模式。可以理解其为最早期的8086芯片的工作模式。这种模式无任何保护机制，只能运行16位代码、不支持虚拟内存、不支持访问1MB以上的内存。

x86处理器编程模型



通用寄存器：

- **EAX**：累加寄存器，通常用于算术运算、返回值和累加操作。
- **EBX**：基址寄存器，通常用于存储数组或表格的基址。
- **ECX**：计数寄存器，通常用于循环计数和字符串操作。
- **EDX**：数据寄存器，通常用于存储除法和乘法操作的高32位。
- **ESI**：源索引寄存器，通常用于字符串处理，指向源数据。
- **EDI**：目标索引寄存器，通常用于字符串处理，指向目标数据。
- **EBP**：基指针寄存器，通常用于存储栈帧的基址。
- **ESP**：栈指针寄存器，指向当前栈顶。

段寄存器

段寄存器用于 **定义内存段的基址**，使得 CPU 可以通过 **段选择子** 定位到内存中的具体区域。

- **CS**（Code Segment）：代码段寄存器，指向当前执行的代码所在的内存区域。
- **DS**（Data Segment）：数据段寄存器，指向数据存储的内存区域。
- **SS**（Stack Segment）：栈段寄存器，指向栈的内存区域。
- **ES**（Extra Segment）：额外数据段寄存器，通常用于某些数据存储或字符串操作。
- **FS** 和 **GS**（额外的段寄存器）：用于特定的应用，尤其是在多任务操作系统中，用于指向额外的数据或任务上下文。

EIP（Extended Instruction Pointer）

- **EIP**：指令指针寄存器，用于存储 **下一条将被执行的指令的内存地址**。
- 在 **实模式** 和 **保护模式** 下，EIP 都用于跟踪程序的执行位置。

EFLAGS程序状态控制寄存器

控制寄存器（Control registers）

- **CR0**：控制寄存器 0，控制 CPU 的基本操作模式，如启用保护模式。
- **CR3**：控制寄存器 3，存储页目录表的物理地址，用于分页。
- **CR4**：控制寄存器 4，控制其他特性，如虚拟化支持等。

段寄存器

为什么提出段寄存器

最初的 x86 处理器（例如 8086），并没有直接支持 32 位的线性内存地址，最多访问64KB的内存空间。

16 位实模式段寄存器寻址

在 x86 的 **16 位实模式** 下，每个地址是一个 **段地址(16位) + 偏移地址（4位）**。
段寄存器（如 `CS`，`DS`，`SS` 等）定义了一个 **段** 的 **基地址**，**偏移量** 是指 **段内的地址**，即相对于段基址的位置。

CPU 能够访问最大 1MB 的内存空间。

寄存器保存了所访问的内存段的起始地址 >> 4。在使用时，其会和偏移量共同形成最终的地址：段寄存器值 << 4 + 偏移。

保护模式段寄存器寻址

段寄存器不再直接表示物理内存地址的起始位置，存储的是一个**段选择子**，选择**段描述符**GDT或 LDT。段描述符是一个 8 字节的结构，包含了段的基地址、大小、权限信息等。

段选择子（Segment Selector）

16 位
Index高13位：段描述符的索引，指示选择子在 GDT 或 LDT 中的索引位置（8的倍数）。
TI（Table Indicator）1位：选择子所指向的是 **GDT** 还是 **LDT**。
RPL（Request Privilege Level）：请求特权级，2 位，表示请求该段的特权级别（如 0 表示内核级，3 表示用户级）。

段寄存器的寻址过程

- **段寄存器提供段选择子**，它指向 **GDT 或 LDT** 中的 **段描述符**。
- **从段描述符中提取段的基地址**（Base Address）和 **段的界限**（Limit）。
- **偏移量**：通过段寄存器存储的 **偏移量** 与基地址结合，得到最终的物理地址。
  - 公式：`物理地址 = Base + Offset`
- **`EIP`**：**指令的偏移量**，并与段寄存器中的基地址（如 `CS`）结合计算物理地址。
- **`ESP`**：当前栈帧的偏移量。与 `SS`（栈段寄存器）结合，计算当前栈帧的物理地址。

假设我们有一个段寄存器 `DS = 0x10`，表示它指向 GDT 中的第 2 个段描述符，且该段描述符的基地址为 `0x00100000`（16MB），偏移量为 `0x2000`。则物理地址为：

```
物理地址 = 基地址 + 偏移量
         = 0x00100000 + 0x2000
         = 0x00102000
```

平坦模型

没有使用其复杂的分段模式，即采用平坦模式，所有的段寄存器全部指向0。

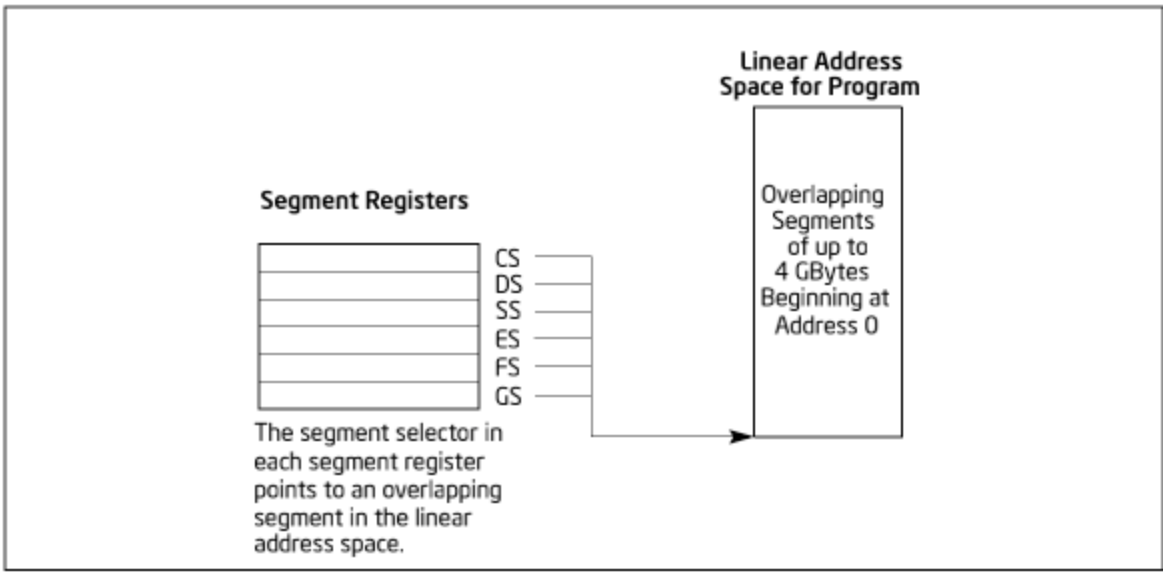
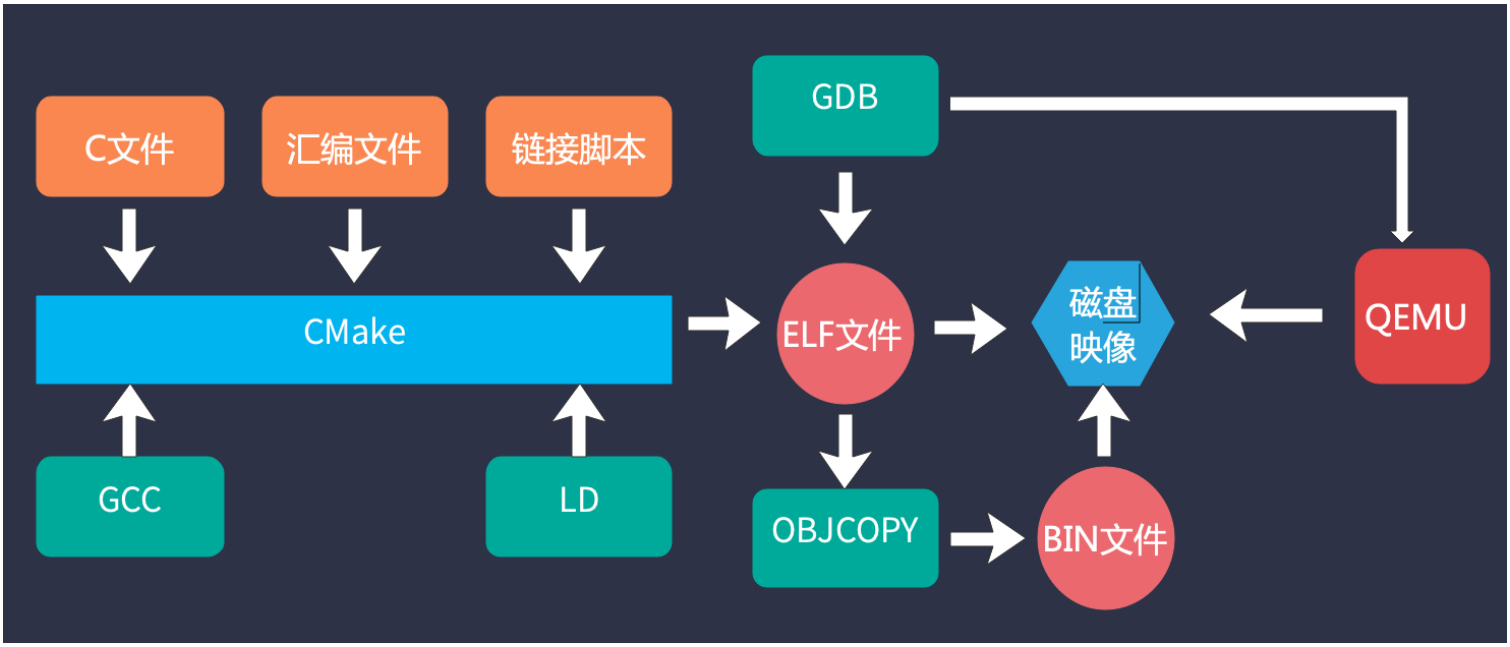


Figure 3-6. Use of Segment Registers for Flat Memory Model

存储映射

实模式模式只支持访问1MB以内的内存，具体的内存映射如下。BIOS会自动将磁盘的第0扇区加载到0x7c00地址处。

工程模板介绍



1. 在VIsual Studio Code中编写C源文件、汇编文件、链接脚本
2. CMake根据配置脚本，调用GCC对源文件进行编译和汇编，调用LD进行链接生成可执行的ELF文件
3. CMake还会调用OBJCOPY将ELF文件进行缩小，或者转换成BIN文件。
4. 将BIN文件写入磁盘映像
5. 调试时，QEMU加载磁盘映像文件，然后等待GDB连接。
6. GDB连接上QEMU，必须正确设置elf，否则没有调试信息，开始正式的调试过程。

> 磁盘映像具体是什么(.bin二进制文件只包含代码和数据)
> GDB调试需要加载elf

elf 文件和bin文件

链接器生成elf文件，elf文件生成bin写入磁盘映像，elf文件借助工具生成反汇编和elf详细信息文件方便调试分析。

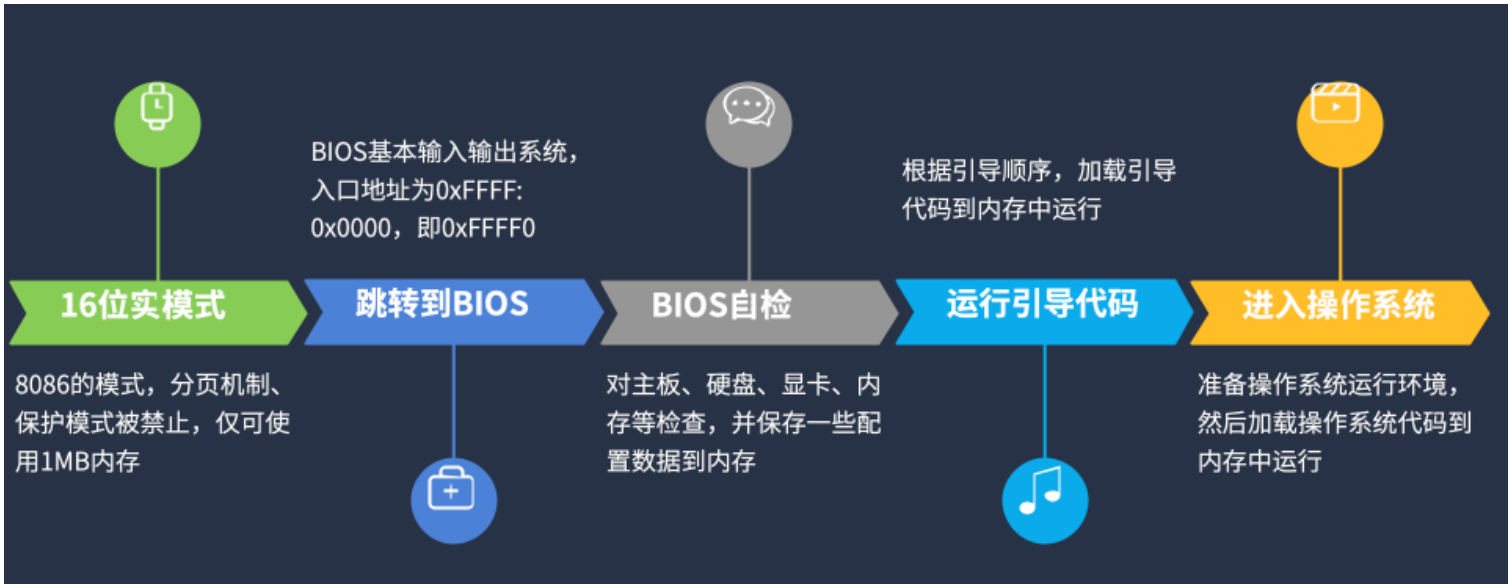
elf文件

ELF Header：包含文件的元数据，描述了文件的类型、体系结构、入口点等信息。
Program Header Table（程序头）：描述如何将程序加载到内存中的信息，告诉操作系统每个段的内存映射方式（地址、大小、权限、对齐方式）
Section Header Table（节头）：节通常包含代码（.text）、数据(.rodata .data .bss)、符号（.symtab）、字符串（.strtab）等。
符号表和调试信息：存储程序的符号信息（如函数名、变量名等）以及调试所需的信息。

bin 文件格式

`.bin` 文件没有任何信息来指导操作系统如何加载它，只包含原始二进制数据，通常是代码或数据。没有调试信息，通常不能直接调试。

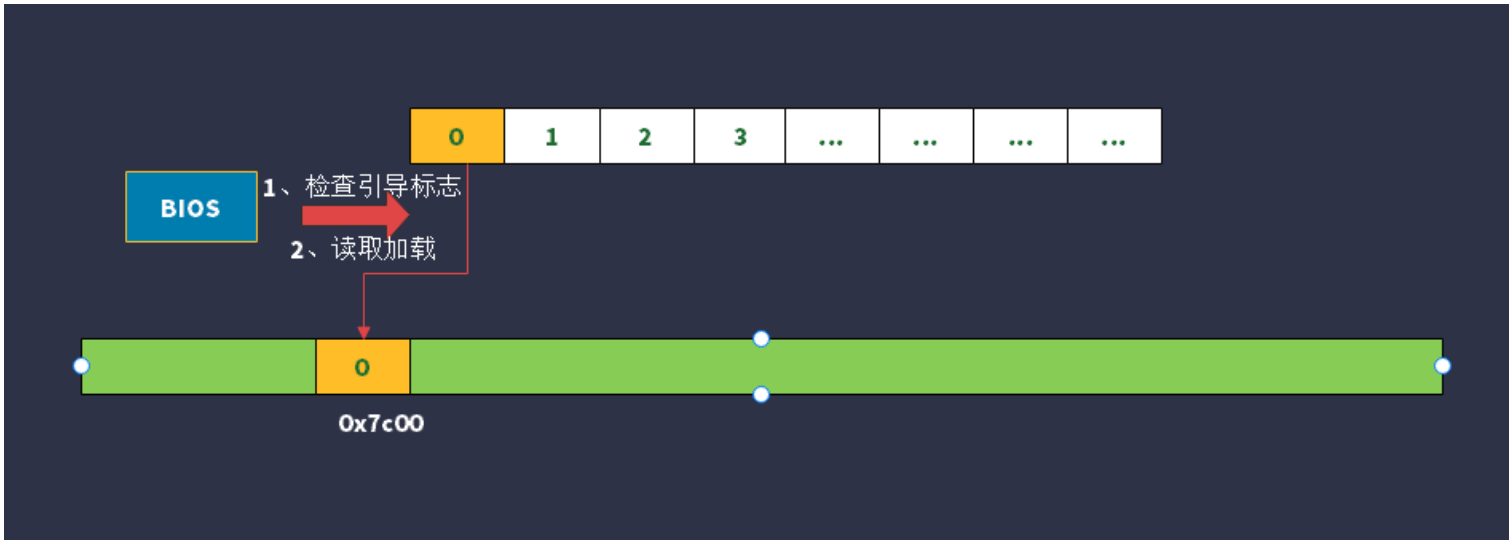
启动流程



1. 上电启动后，CPU处于16位运行的实模式，分页机制禁止，此时只有1MB内存可用，没有特权级
2. CPU跳转到BIOS的入口（地址为0xFFFF0）开始执行
3. BIOS进行硬件自检（POST），主板、硬盘、显卡、内存等自检，并保存一些配置数据到特定内存地址处（如硬盘的数量）
4. 根据配置的启动顺序（光驱、U盘、硬盘等），加载引导代码运行。例如，从硬盘启动时，将硬盘的第1扇区(主引导纪录)读取到0x7c00处并且跳转到该地址处运行
5. 引导代码对操作系统的运行初始环境进行配置，并加载操作系统到内存中
6. 跳转到操作系统运行
   我们的主要工作是完成最后两步，实现自己的引导代码和操作系统。

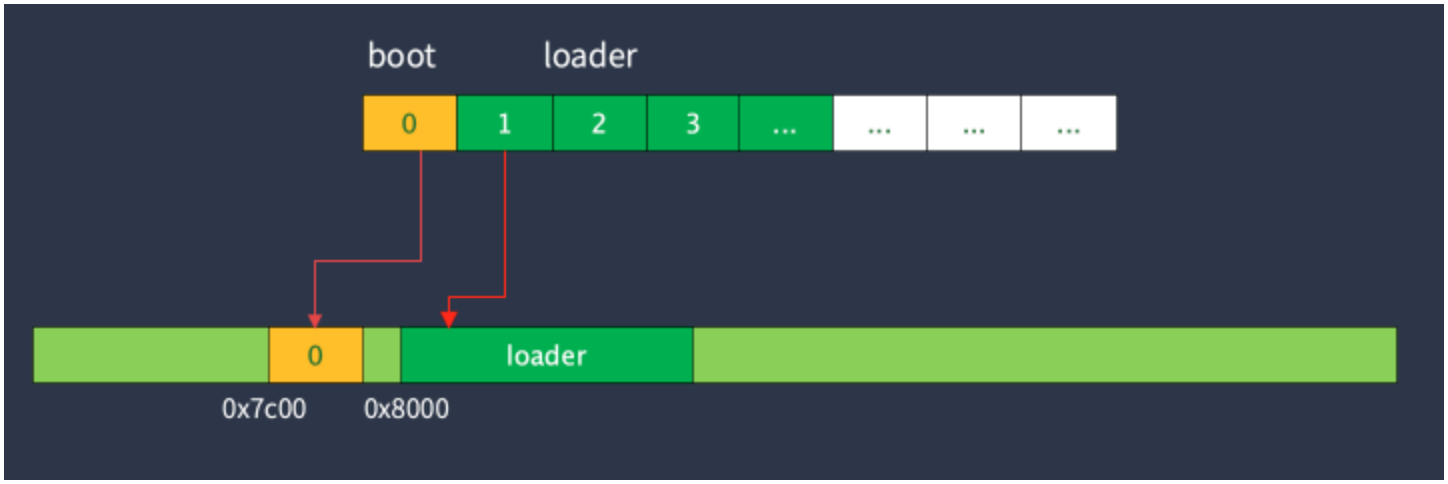
接管控制权

启动磁盘没有分区的情况，以简化学习。
BIOS在完成自检后，会检查第0扇区的最后两个字节是否是0x55, 0xaa，以此来判断是否包含有效的引导代码。如果是，则自动从引导磁盘的第0扇区加载引导程序到0x7c00处执行。



使用BIOS中断读取磁盘

BIOS提供了一些服务函数，方便开发操作系统使用。
在使用时并不需要知道特定函数的入口地址，因其内部通过向量表的方式去访问，向量表里保存了函数的入口地址。



INT13磁盘读取
调用BIOS中断从第1扇区加载loader到0x8000地址处，之后跳转到0x8000地址处运行。

进入C语言环境并跳到loader

从汇编语言进入到C语言环境中运行?
从boot进入到loader中运行?

从汇编语言跳到C语言

从汇编跳转到C语言执行，有两种方式：一是用JMP直接跳转过去；二是用CALL指令进行函数调用。
在本课时中，由于是从boot中的汇编跳转到C语言，无需返回，所以直接用JMP跳转。

在使用前，先用.extern boot_entry导入外部boot_entry符号，然后再用jmp boot_entry跳转。

跳转到指定loader运行

boot和loader分属两个工程，共生成两个bin文件。
从boot跳到loader，只知道loader的起始地址为0x8000，所以采用函数指针转换。
(void (*)(void) 为无参数、无返回值的函数类型。((void (*)(void))LOADER_START_ADDR)()
即认为在0x8000地址处存放了这种类型的函数的代码，通过调用函数函数进入到loader中运行。

> 注：无论是boot还是loader，其工程均已经配置好让start.S中的代码位于生成的bin文件开头。所以无论是boot还是loader，其最开头的指令总是程序的入口指令。