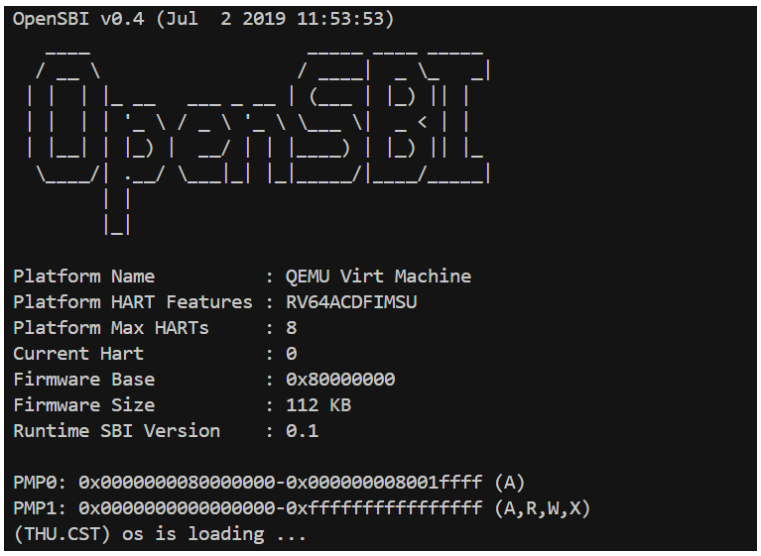
lab01实验报告

任务二

1. 调试过程

根据任务指导书的内容，我们首先使用 `make qemu` 进行测试，得到如下结果，说明环境配置完成。可以看到这里已经成功输出字符串 `(THU.CST) os is loading ...\n` ，同时陷入 `while(1)` 的死循环。



我们接下来尝试对其进行GDB调试。首先，我们打开两个终端，分别输入 `make debug` 和 `make gdb` ，在GDB调试的界面，我们可以看到语句 `0x0000000000001000 in ?? ()` ，也就是说现在的PC在复位地址 `0x1000` 地址处。

我们接下来使用语句 `x/10i $pc` 来获取将要执行的10条指令，得到如下结果。

```
0x1000:      auipc   t0,0x0          # 将PC的值加载到t0中
0x1004:      addi    a1,t0,32        # a1 = t0 + 32
0x1008:      csrr    a0,mhartid      # 获取当前 CPU 核心的 ID，并将其值存入 a0
0x100c:      ld      t0,24(t0)       # 在 t0 + 24 所在地址处读取一个64位的值，加载入 t0
0x1010:      jr      t0              # 跳转t0存储的地址。
0x1014:      unimp
0x1016:      unimp
0x1018:      unimp
0x101a:      0x8000
0x101c:      unimp
```

我们可以看到这里最后跳转的相当于是 `0x1018` 的值，我们可以尝试读取一下，看其具体是什么。使用指令 `x/1x 0x1018` 进行读取，发现该地址上存着值 `0x80000000` ，也就是说PC直接跳转到了 `0x80`

`000000` ，与我们的bootloader的初始加载地址相同，我们可以使用 `b *0x80000000` 在该地址打上断点，然后直接输入指令 `c` 运行到该地址。

接下来使用和刚刚相同的 `x/10i $pc` 来获取将要执行的10条指令，得到如下结果。

```
0x80000000:  csrr    a6,mhartid
0x80000004:  bgtz    a6,0x80000108
0x80000008:  auipc   t0,0x0
0x8000000c:  addi    t0,t0,1032
0x80000010:  auipc   t1,0x0
0x80000014:  addi    t1,t1,-16
0x80000018:  sd      t1,0(t0)
0x8000001c:  auipc   t0,0x0
0x80000020:  addi    t0,t0,1020
0x80000024:  ld      t0,0(t0)
```

这里主要执行的是区分主/从核，仅让主核（Hart 0）执行初始化，并准备跳转到内核阶段。至于具体在哪里进入内核，我们马上会看到。

使用指令 `b* kern_entry` ，在内核的入口函数 `kern_entry` 处设置一个断点。然后输入 `c` 继续执行，程序便会停在 `kern/init/entry.S` 中，这证明 OpenSBI 已经完成了它的引导任务，并将控制权成功移交给了我们的内核。此时再次使用 `x/10i $pc` 查看，可以看到我们自己编写的内核汇编代码：

```
0x80200000 <kern_entry>:     auipc   sp,0x3
0x80200004 <kern_entry+4>:   mv      sp,sp
0x80200008 <kern_entry+8>:   j       0x8020000a <kern_init>
0x8020000a <kern_init>:      auipc   a0,0x3
0x8020000e <kern_init+4>:    addi    a0,a0,-2
0x80200012 <kern_init+8>:    auipc   a2,0x3
0x80200016 <kern_init+12>:   addi    a2,a2,-10
0x8020001a <kern_init+16>:   addi    sp,sp,-16
0x8020001c <kern_init+18>:   li      a1,0
0x8020001e <kern_init+20>:   sub     a2,a2,a0
```

前两条指令实际上是汇编代码中的 `la sp, bootstacktop` ，也就是加载 bootstacktop 的地址到栈指针 sp，设置了初始的栈指针。之后的一条命令则代表调用 `kern_init` 函数。且此时OpenSBI界面出现，也就是说内核正式启动。

接下来就是 `init.c` 的具体实现，整体相对“白盒”。主要是通过 `memset` 清空 `BSS` 段，确保全局变量的初始值为 0。再调用我们创造的 `cprintf` 函数，同时进入待机死循环状态。

```
int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);
    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message);
    while (1);
}
```

2. 练习回答

1. RISC-V 硬件加电后最初执行的几条指令位于什么地址?
   加电后最初执行的指令位于地址 `0x1000` 到地址 `0x1010` 。
2. 它们主要完成了哪些功能?
   i. **准备参数**:
      - 通过 `csrr a0, mhartid` 获取当前 CPU 核心的 ID，并存入 `a0` 寄存器。
      - 通过 `auipc` 和 `addi` 指令计算出的地址，并存入 `a1` 寄存器。
        这两个参数会传递给后续的 OpenSBI 和操作系统内核。

   ii. **加载目标地址**:
      - 通过 `ld t0, 24(t0)` 指令，从一个约定好的内存地址 `0x1018` 加载下一阶段的入口地址（即 OpenSBI 的入口地址 `0x80000000`）到 `t0` 寄存器。

   iii. **移交控制权**:
      - 执行 `jr t0` 指令，将 CPU 的控制权无条件地跳转到 `t0` 寄存器中的地址，即 OpenSBI 的入口，从而完成第一阶段的引导任务。