

# Lab1

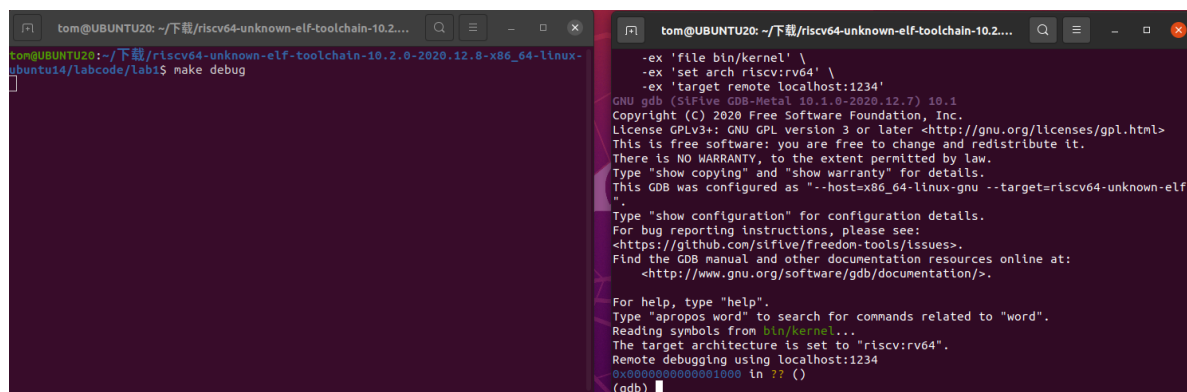
## 练习一

1. `la sp, bootstacktop`: 这一指令将栈顶地址`bootstacktop`加载到栈指针寄存器`sp`中, 为C代码的执行准备栈空间
2. `tail kern_init`: `tail`指令进行尾调用, 跳转到`kern_init`函数, 这一函数通常来说是C语言编写的主初始化函数, 也就是操作系统的入口。此外, 尾调用优化不会占用额外的栈帧。

## 练习二

### 实验过程

打开两个终端后, 在左边终端里输入`make debug`, 在右边终端里输入`make gdb`, 得到如下的界面, 说明成功进入调试阶段



```
tom@UBUNTU20: ~/下载/riscv64-unknown-elf-toolchain-10.2...
tom@UBUNTU20:~/下载/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14/labcode/lab1$ make debug

tom@UBUNTU20: ~/下载/riscv64-unknown-elf-toolchain-10.2...
GNU gdb (Sifive GDB-Metal 10.1.0-2020.12.7) 10.1
Copyright (c) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-linux-gnu --target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://github.com/sifive/freedom-tools/issues>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
Reading symbols from bin/kernel...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000000000 in ?? ()
(gdb)
```

输入`l(list)`可以显示当下的10行源代码, 如下所示:

```
1 1 #include <mmu.h>
2 2 #include <memlayout.h>
3 3
4 4     .section .text,"ax",%progbits
5 5     .globl kern_entry
6 6 kern_entry:
7 7     la sp, bootstacktop
8 8
9 9     tail kern_init
10 10
```

但是这些源码不够底层, 无法确定单步调试直到`0x80200000`需要的次数, 因此经过查阅资料, 可以使用`x/10i 0x1000`指令显示当下的10行汇编代码, 如下所示:



根据练习一可知这是将栈顶地址bootstacktop加载到栈指针寄存器sp中，接下来查看汇编指令：

```
1 (gdb) x/10i 0x80200000
2 0x80200000 <kern_entry>: auipc    sp,0x3                ;sp = PC + (0x3
<< 12) = 0x80200000 + 0x3000 = 0x80203000
3 0x80200004 <kern_entry+4>: mv     sp,sp                ;没有实际用途，可能
是用于调试
4 0x80200008 <kern_entry+8>: j      0x8020000a <kern_init> ;直接跳转到内核初始
化函数
5 0x8020000a <kern_init>: auipc    a0,0x3                ;a0 = PC + (0x3
<< 12) = 0x8020000a + 0x3000 = 0x8020300a
6 0x8020000e <kern_init+4>: addi    a0,a0,-2             ;a0 = a0 - 2 =
0x8020300a - 2 = 0x80203008
7 0x80200012 <kern_init+8>: auipc    a2,0x3                ;a2 = PC + (0x3
<< 12) = 0x80200012 + 0x3000 = 0x80203012
8 0x80200016 <kern_init+12>: addi    a2,a2,-10            ;a2 = a2 - 10 =
0x80200012 - 10 = 0x80203008
9 0x8020001a <kern_init+16>: addi    sp,sp,-16            ;在栈上分配16字节空
间
10 0x8020001c <kern_init+18>: li     a1,0                ;将 a1 寄存器清零
11 0x8020001e <kern_init+20>: sub     a2,a2,a0            ;计算内存区域大小
```

从上面汇编指令中可以发现在入口点之后就是初始化函数，因此在kern\_init处设置断点，得到如下结果：

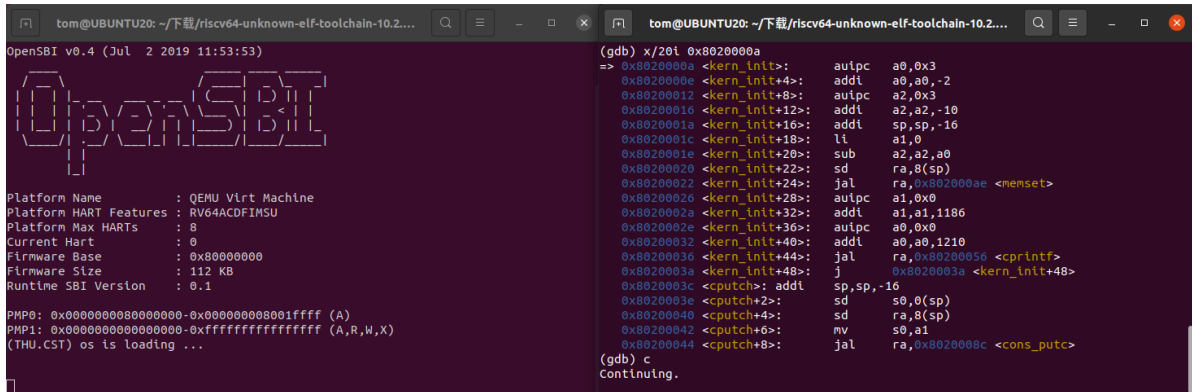
```
1 Breakpoint 2 at 0x8020000a: file kern/init/init.c, line 8.
```

这说明初始化函数的地址是0x8020000a，接着执行指令c，然后查看汇编代码，部分结果如下：

```
1 (gdb) x/20i 0x8020000a
2 0x8020000a <kern_init>: auipc    a0,0x3                ;a0 = PC + (0x3
<< 12) = 0x8020000a + 0x3000 = 0x8020300a
3 0x8020000e <kern_init+4>: addi    a0,a0,-2             ;a0 = a0 - 2 =
0x8020300a - 2 = 0x80203008
4 0x80200012 <kern_init+8>: auipc    a2,0x3                ;a2 = PC + (0x3
<< 12) = 0x80200012 + 0x3000 = 0x80203012
5 0x80200016 <kern_init+12>: addi    a2,a2,-10            ;a2 = a2 - 10 =
0x80200012 - 10 = 0x80203008
6 0x8020001a <kern_init+16>: addi    sp,sp,-16            ;在栈上分配16字节空
间
7 0x8020001c <kern_init+18>: li     a1,0                ;将 a1 寄存器清零
8 0x8020001e <kern_init+20>: sub     a2,a2,a0            ;计算内存区域大小
9 0x80200020 <kern_init+22>: sd      ra,8(sp)             ;保存返回地址到栈上
10 0x80200022 <kern_init+24>: jal     ra,0x802000ae <memset> ;调用memset函数清
零BSS段
11 0x80200026 <kern_init+28>: auipc    a1,0x0
12 0x8020002a <kern_init+32>: addi    a1,a1,1186          ;a1 = 0x80200026
+ 1186 = 0x802004c8
13 0x8020002e <kern_init+36>: auipc    a0,0x0
14 0x80200032 <kern_init+40>: addi    a0,a0,1210          ;a0 = 0x8020002e
+ 1210 = 0x802004e8
15 0x80200036 <kern_init+44>: jal     ra,0x80200056 <cprintf> ;调用cprintf输出内
核启动信息
```

```
16      0x8020003a <kern_init+48>:      j      0x8020003a <kern_init+48>;跳转到自身，进入无限循环
```

从上面的代码可知，该程序在执行j 0x8020003a指令后会跳转回自己的位置，因此操作系统会一直运行，也就是如下的显示：



The screenshot shows two windows. The left window displays the OpenSBI boot log, including platform information (QEMU Virt Machine, RV64ACDFIMSU) and firmware details. The right window shows a GDB disassembly of the kernel initialization code, starting from address 0x8020000a. The disassembly includes instructions like `auipc a0,0x3`, `addi a0,a0,-2`, `auipc a2,0x3`, `addi a2,a2,-10`, `addi sp,sp,-16`, `li a1,0`, `sub a2,a2,a0`, `sd ra,0(sp)`, `jal ra,0x802000ae <memset>`, `addi a1,a1,1186`, `auipc a0,0x0`, `addi a0,a0,1210`, `jal ra,0x80200056 <printf>`, `j 0x8020003a <kern_init+48>`, `addi sp,sp,-16`, `sd s0,0(sp)`, `sd ra,0(sp)`, `nv s0,a1`, and `jal ra,0x8020008c <cons_putc>`.

这样就完成了riscv从加电到执行内核第一条指令的全部过程！

## 题目回答

1. RISC-V 硬件加电后最初执行的几条指令在0x1000-0x1010的地址范围内
2. 这些指令主要是获取硬件信息并跳转到主程序，每条指令及功能如下

1	<code>auipc t0,0x0</code>	;将当前PC值的高20位加上立即数0x0，结果存入t0
2	<code>addi a1,t0,32</code>	;将 t0 + 32 存入 a1
3	<code>csrr a0,mhartid</code>	;读取 mhartid CSR（控制和状态寄存器）到 a0
4	<code>ld t0,24(t0)</code>	;从内存地址（t0 + 24）加载双字到 t0
5	<code>jr t0</code>	;跳转到 t0 寄存器指定的地址