Lab0.5

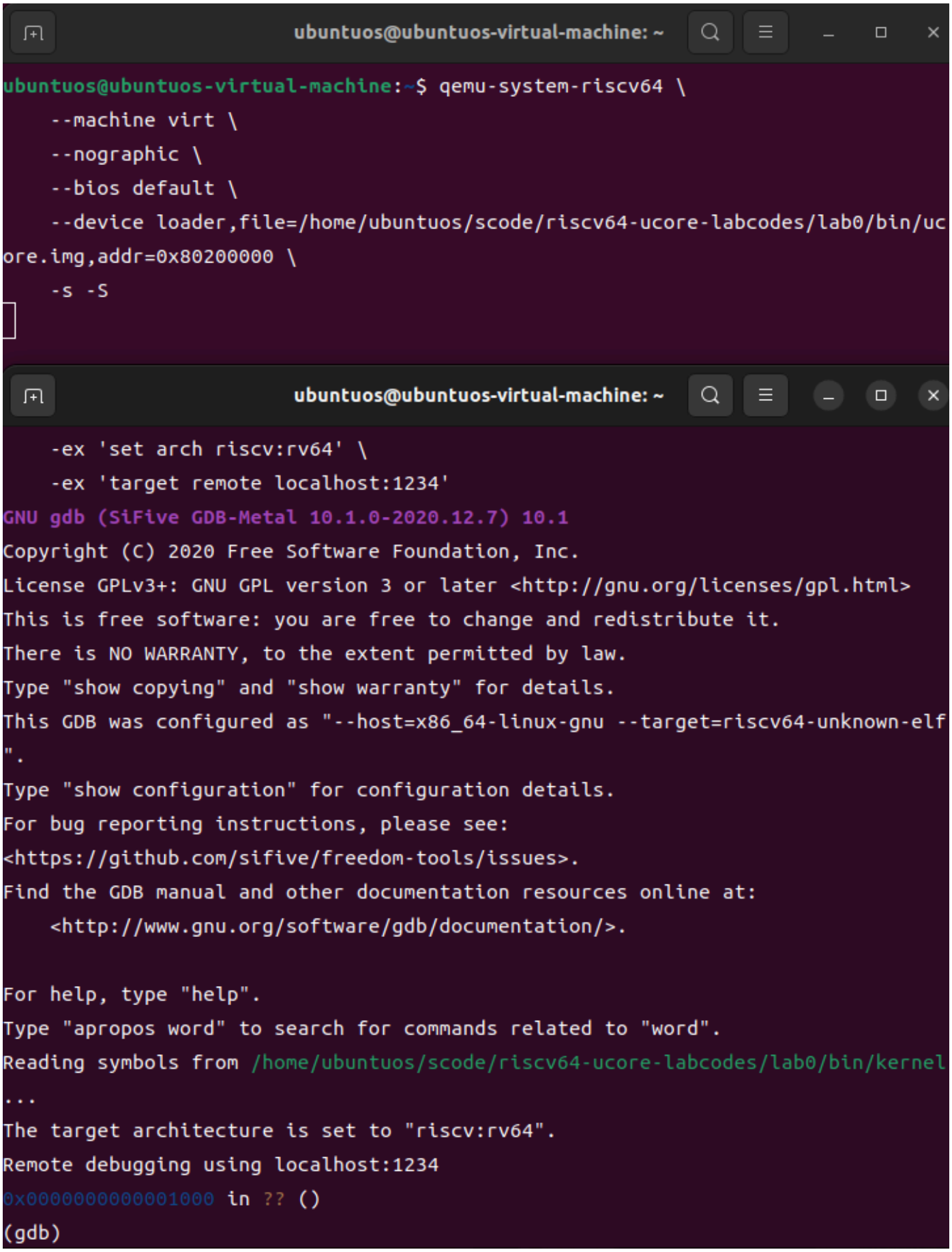
练习1: 使用GDB验证启动流程

为了熟悉使用qemu和gdb进行调试工作,使用gdb调试QEMU模拟的RISC-V计算机加电开始运行到执行应用程序的第一条指令（即跳转到0x80200000）这个阶段的执行过程，说明RISC-V硬件加电后的几条指令在哪里？完成了哪些功能？

一、GDB调试命令示例

1.启动调试会话

- 使用Makefile中的 `make debug` 或 `make gdb` 命令来启动带有GDB支持的QEMU实例。


ubuntuos@ubuntuos-virtual-machine: ~
ubuntuos@ubuntuos-virtual-machine:~$ qemu-system-riscv64 \
--machine virt \
--nographic \
--bios default \
--device loader,file=/home/ubuntuos/scode/riscv64-ucore-labcodes/lab0/bin/ucore.img,addr=0x80200000 \
-s -S
ubuntuos@ubuntuos-virtual-machine: ~
-ex 'set arch riscv:rv64' \
-ex 'target remote localhost:1234'
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 /home/ubuntuos/scode/riscv64-ucore-labcodes/lab0/bin/kernel...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb)

2.查看指定地址附近的指令

- `x/10i 0x80000000`：显示从地址 `0x80000000` 开始的10条汇编指令。

```
(gdb) x/10i 0x80000000
   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)
```

- `x/10i $pc`：显示当前程序计数器(`$pc`)指向位置之后的10条汇编指令。

```
(gdb) x/10i $pc
=> 0x1000:      auipc   t0,0x0
   0x1004:      addi    a1,t0,32
   0x1008:      csrr    a0,mhartid
   0x100c:      ld      t0,24(t0)
   0x1010:      jr      t0
   0x1014:      unimp
   0x1016:      unimp
   0x1018:      unimp
   0x101a:      0x8000
   0x101c:      unimp
```

3.查看内存内容

- `x/10xw 0x80000000`：以32位十六进制形式显示从 `0x80000000` 开始的10个内存单元的内容。

```
(gdb) x/10xw 0x80000000
0x80000000:     0xf1402873      0x11004263      0x00000297      0x40828293
0x80000010:     0x00000317      0xff030313      0x0062b023      0x00000297
0x80000020:     0x3fc28293      0x0002b283
```

4.查看寄存器状态

- `info register`：列出所有寄存器的当前值。
- `info r t0`：仅显示 `t0` 寄存器的值。

```
(gdb) info register
ra             0x0      0x0
sp             0x0      0x0
gp             0x0      0x0
tp             0x0      0x0
t0             0x0      0
t1             0x0      0
t2             0x0      0
fp             0x0      0x0
s1             0x0      0
a0             0x0      0
a1             0x0      0
a2             0x0      0
a3             0x0      0
a4             0x0      0
a5             0x0      0
a6             0x0      0
a7             0x0      0
s2             0x0      0
s3             0x0      0
s4             0x0      0
s5             0x0      0
s6             0x0      0
s7             0x0      0
--Type <RET> for more, q to quit, c to continue without paging--info r t0
s8             0x0      0
s9             0x0      0
s10            0x0      0
s11            0x0      0
t3             0x0      0
t4             0x0      0
t5             0x0      0
t6             0x0      0
pc             0x1000   0x1000
dscratch       Could not fetch register "dscratch"; remote failure reply 'E14'
mucounteren    Could not fetch register "mucounteren"; remote failure reply 'E14'
```

5.设置断点

- `break funcname`：在名为 `funcname` 的函数入口处设置断点。
- `break *0x80200000`：在地址 `0x80200000` 处设置断点。

```
(gdb) break *0x80200000
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
```

6.控制程序执行

- `continue`：继续执行程序，直到遇到下一个断点。
- `si`：执行单步操作，即执行下一条汇编指令。

```
(gdb) continue
Continuing.

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7           la sp, bootstacktop
(gdb) si
0x0000000080200004 in kern_entry () at kern/init/entry.S:7
7           la sp, bootstacktop
(gdb) si
9           tail kern_init
(gdb) si
kern_init () at kern/init/init.c:8
8           memset(edata, 0, end - edata);
(gdb)
```

二、RISC-V加电后的几条指令功能分析

RISC-V硬件加电后的几条指令位于地址0x80000000及其后续地址

1. **`csrr a6, mhartid`**
   - 读取当前硬件线程的 ID（mhartid 寄存器）到寄存器 a6。这是初始化过程的一部分，用于确认当前处理器核心的标识。
2. **`bgtz a6, 0x80000108`**
   - 如果 a6 中的值大于零，则跳转到 0x80000108。这条指令通常用于检查当前核心是否为主核心，如果不是，则跳过后续初始化。
3. **`auipc t0, 0x0`**
   - 将当前指令地址的高 20 位加载到 t0 中。这条指令常用于地址计算，通常是准备在后续的指令中使用。
4. **`addi t0, t0, 1032`**
   - 在 t0 中增加 1032。这通常用于计算栈指针或数据段的地址。
5. **`auipc t1, 0x0`**
   - 同样将当前指令地址的高 20 位加载到 t1 中，为后续的地址计算做准备。
6. **`addi t1, t1, -16`**
   - 在 t1 中减少 16。这条指令通常用于设置数据区域的末尾或栈的初始位置。
7. **`sd t1, 0(t0)`**

- 将 t1 中的值存储到 t0 指向的内存地址。这里可能是在设置某个数据结构的初始值或栈指针。

8. **`auipc t0, 0x0`**
   - 再次将当前指令地址的高 20 位加载到 t0 中，为后续使用准备。
9. **`addi t0, t0, 1020`**
   - 在 t0 中增加 1020，继续地址计算。
10. **`ld t0, 0(t0)`**
    - 从 t0 指向的内存地址加载数据到 t0 中。这通常是读取一些初始化数据或程序状态。

在这些指令中，主要完成了以下功能：

- 硬件线程ID读取: 确认当前线程的身份。
- 条件跳转: 确定是否继续初始化流程。
- 地址计算和数据存储: 设置必要的数据结构和内存区域，为后续的执行做好准备。