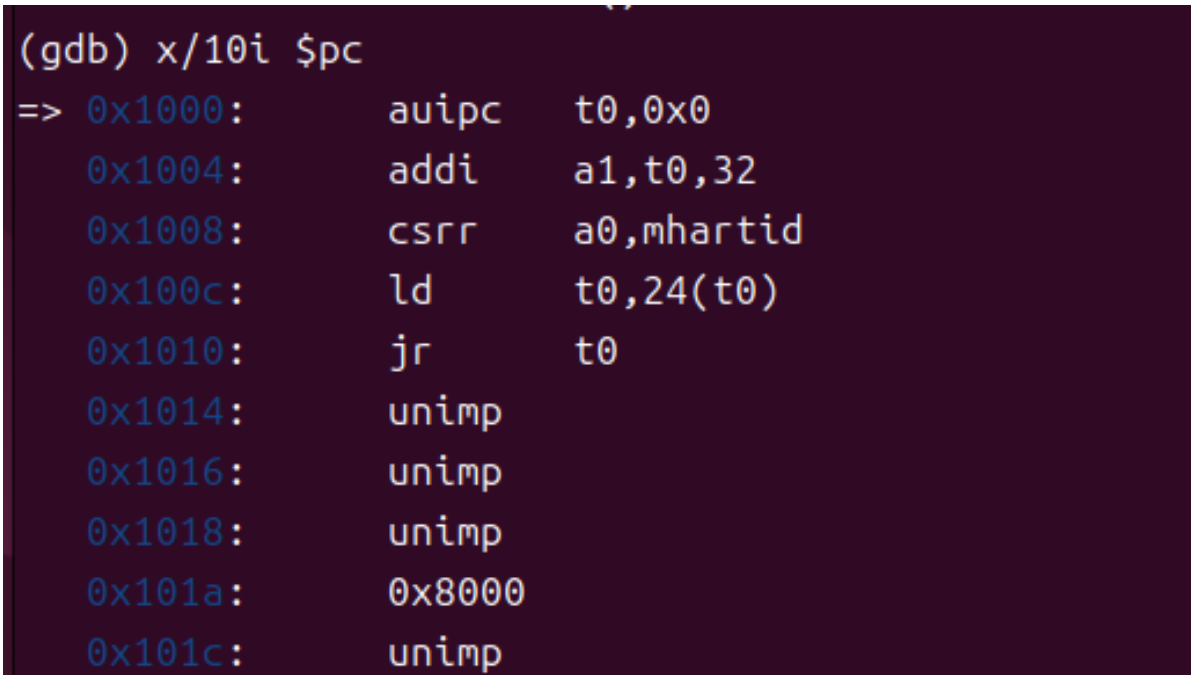
lab0.5

题目：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？要求在报告中简要写出练习过程和回答。

实验过程

启动qemu之后使用make gdb指令进入调试，执行x/10i $pc，输出即将执行的10条指令，这十条指令就是RISC-V加电后的几条指令，如下图所示：



- `auipc`（Add Upper Immediate to PC）指令将当前程序计数器（PC）的高20位与立即数`0x0`相加，并将结果存储到寄存器`t0`中。`t0`中保存的数据是`(pc)+(0<<12)`。用于PC相对寻址。`auipc`用于生成全局地址，因此这个指令主要是将pc值加载到t0中。
- `addi`（Add Immediate）指令将寄存器`t0`的值与立即数`32`相加，并将结果存储到寄存器`a1`中。
- `csrr`（Control and Status Register Read）指令从控制和状态寄存器`mhartid`中读取唯一标识符，并将其存储到寄存器`a0`中，以便访问使用线程id，`mhartid`通常用于标识当前的硬件线程。
- `ld`（Load Doubleword）指令从内存中加载一个双字（64位8个字节）到寄存器`t0`中，内存地址是`t0`寄存器的值加上偏移量`24`。
- `jr`（Jump Register）指令根据寄存器`t0`的值跳转到指定地址。输入`si`单步执行，使用`info r t0`的指令查看涉及到的寄存器结果：

```
(gdb) si
0x0000000000001004 in ?? ()
(gdb) info r t0
t0             0x1000   4096
(gdb) si
0x0000000000001008 in ?? ()
(gdb) info r t0
t0             0x1000   4096
(gdb) si
0x000000000000100c in ?? ()
(gdb) info r t0
t0             0x1000   4096
(gdb) si
0x0000000000001010 in ?? ()
(gdb) info r t0
t0             0x80000000       2147483648
```

可以观察到跳转到了0x80000000处，即跳转到bootloader处开始运行。bootloader（引导加载程序）是计算机系统中的一段小型软件，它负责在计算机开机或复位时初始化系统并加载操作系统。

输入x/10i 0x80000000，显示0x80000000处的10条数据。

```
(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)
```

- 该地址处加载的是作为bootloader的`OpenSBI.bin`，该处的作用为加载操作系统内核并启动操作系统的执行。即跳转到一段汇编代码kern/init/entry.S，引导操作系统内核初始化的起始点，完成栈的初始化以及跳转到kern_init处开始执行。

接着输入指令`break kern_entry`，在目标函数kern_entry的第一条指令处设置断点，输出如下：

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

- 我们可以看见在0x80200000有一个breakpoint，是一个加载地址（即BASE_ADDRESS），指定了操作系统内核将要加载到内存中的起始地址。当操作系统启动时，引导加载程序（bootloader会将内核的二进制文件加载到指定的地址（`BASE_ADDRESS`）中。
- `kern_entry` 是在链接脚本中定义的，表示内核的入口点。通是汇编程序第一次执行的实际指令位置。在启动时，控制权将转移到这个位置以开始内核的初始化过程。

输入指令`x/5i 0x80200000`，查看汇编代码：

```
(gdb) x/5i 0x80200000
   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
```

- 可以看到在`kern_entry`之后，紧接着就是`kern_init`是内核的初始化函数`kern_init`。这个函数负责执行更高层次的初始化工作，例如设置内存管理、初始化设备驱动、启动调度器等。
- 接着输入输入`continue`执行直到断点，debug输出如下：

```
hyf@hyf-VMware-Virtual-Platform: ~/labs/lab0
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
```

此时opensbi已经启动。

输入`disassemble kern_init`查看反汇编代码：

```
(gdb) disassemble kern_init
Dump of assembler code for function kern_init:
=> 0x000000008020000a <+0>:     auipc   a0,0x3
   0x000000008020000e <+4>:     addi    a0,a0,-2 # 0x80203008
   0x0000000080200012 <+8>:     auipc   a2,0x3
   0x0000000080200016 <+12>:    addi    a2,a2,-10 # 0x80203008
   0x000000008020001a <+16>:    addi    sp,sp,-16
   0x000000008020001c <+18>:    li      a1,0
   0x000000008020001e <+20>:    sub     a2,a2,a0
   0x0000000080200020 <+22>:    sd      ra,8(sp)
   0x0000000080200022 <+24>:    jal     ra,0x802004b6 <memset>
   0x0000000080200026 <+28>:    auipc   a1,0x0
   0x000000008020002a <+32>:    addi    a1,a1,1186 # 0x802004c8
   0x000000008020002e <+36>:    auipc   a0,0x0
   0x0000000080200032 <+40>:    addi    a0,a0,1210 # 0x802004e8
   0x0000000080200036 <+44>:    jal     ra,0x80200056 <cprintf>
   0x000000008020003a <+48>:    j       0x8020003a <kern_init+48>
End of assembler dump.
```

函数最后一个指令是`j 0x8020003c <kern_init+48>`，也就是跳转到自己，所以代码会在这里一直循环下去，此时输入`continue`，debug窗口出现以下输出：

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
```

题目回答

上电后的几条指令及其功能如下：

1. `auipc`（Add Upper Immediate to PC）指令将当前程序计数器（PC）的高20位与立即数`0x0`相加，并将结果存储到寄存器`t0`中。`t0`中保存的数据是`(pc)+(0<<12)`。用于PC相对寻址。`auipc`用于生成全局地址，因此这个指令主要是将pc值加载到t0中。
2. `addi`（Add Immediate）指令将寄存器`t0`的值与立即数`32`相加，并将结果存储到寄存器`a1`中。
3. `csrr`（Control and Status Register Read）指令从控制和状态寄存器`mhartid`中读取唯一标识符，并将其存储到寄存器`a0`中，以便访问使用线程id，`mhartid`通常用于标识当前的硬件线程。
4. `ld`（Load Doubleword）指令从内存中加载一个双字（64位8个字节）到寄存器`t0`中，内存地址是`t0`寄存器的值加上偏移量`24`。
5. `jr`（Jump Register）指令根据寄存器`t0`的值跳转到指定地址0x80000000。

重要知识点

- 使用 `make` 工具编译和调试程序时的一个重要步骤，具体涉及到两个命令：`make debug` 和 `make gdb`。`make debug` 是一个命令，用于构建包含调试信息的二进制文件。这意味着在编译过程中，添加了调试符号，以便在调试器（如 GDB）中能够追踪源代码、变量和其他调试信息。

  `make gdb` 的命令用于启动 GDB（GNU Debugger），使用它来调试 `make debug` 所生成的二进制文件。`make gdb` 依赖之前的 `make debug` 的执行结果，也就是说，它需要一个已经构建好的包含调试信息的二进制文件供 GDB 使用。

- 启动操作系统整体过程分析：

  硬件初始化

  计算机加电后，各个硬件组件开始初始化。在此过程中，处理器需要为其寄存器分配初始值，并设置合适的控制状态，这些任务通常由硬件固件完成。随之，计算机会加载复位向量地址，指向启动代码的入口点（在本实验中为0x1000）。在QEMU模拟的RISC-V处理器中，复位向量地址被初始化为0x1000，程序计数器（PC）也设置为该地址，以便处理器从此处执行复位代码，进而将系统各组件（包括处理器、内存和设备）置于初始状态。

  引导加载程序（Bootloader）

  一旦跳转到复位向量指定的入口点，Bootloader便开始执行。QEMU的复位代码将Bootloader位置设定为0x80000000（OpenSBI.bin）。此时，Bootloader的任务是加载操作系统内核，并开始操作系统的执行流程。具体来说，它会跳转到 `kern/init/entry.S`，这段汇编代码负责分配内核栈，并最终跳转到 `kern_init` 函数，作为内核初始化的起始点。

  加载操作系统内核

  接下来，系统会跳转到OpenSBI固件预先导入的ucore内核（os.bin）地址0x80200000。内核的入口点为 `kern/init/init.c`，它是C语言编写的，主要包含 `kern_init()` 函数，该函数在从 `kern/entry.S` 跳转过来后进行进一步的初始化工作，标志着内核的实际启动。

lab1

练习1：理解内核启动中的程序入口操作

指令分析

1. `la sp, bootstacktop`
   - **操作**：该指令用于加载一个地址到寄存器中，将 `bootstacktop` 的地址加载到堆栈指针寄存器 `sp` 中。
   - **目的**：初始化内核的堆栈指针，为后续函数调用和中断处理准备堆栈环境。正确设置堆栈指针非常重要，因为堆栈会被用来保存函数调用中的返回地址、局部变量等。
2. `tail kern_init`
   - **操作**：该指令进行无条件的跳转到另一个函数，且不保存返回地址，即尾调用优化。
   - **目的**：开始内核的初始化过程，通过直接跳转到 `kern_init` 函数，系统开始执行更高层次的初始化代码，如设置虚拟内存、初始化设备、准备进程调度等。

扩展练习 Challenge1：描述与理解中断流程

中断异常的处理流程

1. **异常的产生**：
   - 当 CPU 检测到需要立即处理的情况时，自动停止当前执行的指令序列，然后跳转到寄存器 `stvec` 保存的地址执行指令。由于内核初始化时将该寄存器设置为 `__alltraps`，所以跳转到 `trapentry.S` 中的 `__alltraps` 标签处执行。
2. **保存所有寄存器**：
   - 将所有寄存器保存到栈中，以便异常处理完成后能够恢复到原始状态，包括程序的返回值、程序计数器、基指针等。
3. **处理异常**：
   - 异常处理函数执行具体的处理动作，可能包括修复错误、处理中断请求或执行系统调用等。
4. **恢复上下文**：
   - 一旦异常处理完成，通过 `RESTORE_ALL` 宏从栈中恢复之前保存的寄存器值，以便能够准确地恢复到异常发生前的状态。
5. **返回到原始程序**：
   - 使用 `iret` 指令从异常处理程序返回，使 CPU 从栈中恢复程序计数器、栈指针等关键寄存器，并继续执行被中断的程序。

具体指令分析

- `mov a0, sp`：将当前栈指针（`sp`）的值移动到寄存器 `a0` 中，以便在后续操作中使用 `a0` 寄存器来访问和管理栈上的数据。
- `SAVE_ALL` **中寄存器保存位置**：保存位置由结构体 `trapframe` 和 `pushregs` 中的定义顺序决定，因为后续这些寄存器将作为函数 `trap` 的参数。
- `__alltraps` **需要保存所有寄存器**：中断可能在任何时刻发生，必须确保被中断的程序能够恢复到完全相同的状态，这是实现可靠操作系统的重要基础。所有寄存器也将用于函数 `trap` 参数的一部分，如果不保存所有寄存器，函数参数不完整。

练习2：完善中断处理

练习要求

编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写kern/trap/trap.c函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print_ticks子程序，向屏幕上打印一行文字”100 ticks”，在打印完10行后调用sbi.h中的shut_down()函数关机。

实验过程

trap函数代码完善部分如下：

```
case IRQ_S_TIMER:
            // "All bits besides SSIP and USIP in the sip register are
            // read-only." -- privileged spec1.9.1, 4.1.4, p59
            // In fact, Call sbi_set_timer will clear STIP, or you can clear it
            // directly.
            // cprintf("Supervisor timer interrupt\n");
            /* LAB1 EXERCISE2   YOUR CODE :  */
            /*(1)设置下次时钟中断- clock_set_next_event()
             *(2)计数器（ticks）加一
             *(3)当计数器加到100的时候，我们会输出一个`100ticks`表示我们触发了100次时钟中断，同时打印次数（num）加一
             *(4)判断打印次数，当打印次数为10时，调用<sbi.h>中的关机函数关机
```

```
            */
            clock_set_next_event();
            ticks++;
            if (ticks == 100) {
                print_ticks();
                num++;
                ticks = 0;
            }
            if (num == 10) {
                sbi_shutdown();
            }
            break;
```

实现过程：

当遇到时钟中断（IRQ_S_TIMER）时，首先设置下次时钟中断，即调用clock.h中的clock_set_next_event函数，clock_set_next_event函数代码如下：

```
void clock_set_next_event(void)
{
    sbi_set_timer(get_cycles() + timebase);
}
```

然后增加ticks计数器，当ticks达到 100 时，调用print_ticks函数打印 “100 ticks”，print_ticks函数代码如下：

```
#define TICK_NUM 100
static void print_ticks() {
    cprintf("%d ticks\n", TICK_NUM);
#ifdef DEBUG_GRADE
    cprintf("End of Test.\n");
    panic("EOT: kernel seems ok.");
#endif
}
```

接下来增加num计数器，并将ticks重置回0，便于下一次判断。最后，当num达到 10 时，调用sbi.h中的sbi_shutdown函数关机，sbi_shutdown函数代码如下：

```
void sbi_shutdown(void)
{
    sbi_call(SBI_SHUTDOWN,0,0,0);
}
```

定时器中断处理流程：

1. 当定时器中断发生时，系统进入interrupt_handler函数。
2. 根据中断原因判断是否为时钟中断。
3. 如果是时钟中断，执行以下操作：设置下次中断，更新ticks和num计数器，并根据条件判断进行打印和关机操作。
4. 处理完中断后，继续执行程序。

测试

如下图所示，运行整个系统，大约每1秒会输出一次”100 ticks”，共输出了10行，测试通过。

拓展练习2：理解上下文切换机制

练习要求

回答：在trapentry.S中汇编代码 csrw sscratch, sp；csrrw s0, sscratch, x0实现了什么操作，目的是什么？save all里面保存了stval scause这些csr，而在restore all里面却不还原它们？那这样store的意义何在呢？

实验过程

分析trapentry.S代码的大致含义如下：

该代码通过保存和恢复上下文，确保在异常发生后程序能够正确地继续执行。

SAVE_ALL宏用于在发生异常时保存当前程序的状态，以便在异常处理完成后能够恢复程序的执行。它首先将当前栈指针sp存入特殊控制状态寄存器sscratch；然后调整栈指针，为保存寄存器腾出空间；接着保存通用寄存器x0到x31的值到栈上特定位置；最后读取并保存一些特殊控制状态寄存器（sstatus、sepc、sbadaddr、scause）的值到栈上。

RESTORE_ALL宏用于在异常处理完成后恢复程序的上下文，使程序能够继续从异常发生的地方继续执行。它首先从栈上读取之前保存的特殊控制状态寄存器的值，并写回到相应的寄存器中，恢复状态和异常程序计数器；然后从栈上恢复通用寄存器x1到x31的值；最后恢复栈指针sp的值。

__alltraps是异常处理的入口点。当发生异常时，程序跳转到这里。首先调用SAVE_ALL宏保存当前上下文；然后将栈指针的值赋给a0寄存器，最后调用trap函数进行具体的异常处理。

__trapret是异常处理返回的入口点。它首先调用RESTORE_ALL宏恢复之前保存的上下文；然后发出从监督模式返回的指令sret，使程序恢复执行。

问题1：csrw sscratch, sp；csrrw s0, sscratch, x0实现了什么操作，目的是什么？
回答：
（1）csrw sscratch, sp将当前栈指针sp的值写入特殊控制状态寄存器sscratch。sscratch寄存器通常被用作暂存器，可以存储一些临时数据，将栈指针存入sscratch寄存器能够在后续的异常处理过程中快速访问到当前栈的位置，以便保存和恢复上下文信息。
（2）csrrw s0, sscratch, x0先将sscratch寄存器的值读入临时寄存器s0，然后将x0写入sscratch，即将sscratch寄存器清零。这样可以获取之前存入sscratch的栈指针值，保存到s0中以供后续使用，而将sscratch清零，则能在一些异常发生时，很容易地识别出它来自内核，方便后续进行异常处理。

问题2：save all里面保存了stval scause这些csr，而在restore all里面却不还原它们？那这样store的意义何在呢？
回答：
（1）scause寄存器记录的是异常发生的原因，stval寄存器存储的是与特定异常相关的地址或数据，这些寄存器在异常处理返回后并不需要恢复到异常发生前的状态。这样做可以减少恢复操作的时间和复杂性。
（2）在不同的异常处理场景下，可能需要根据具体情况决定哪些寄存器需要保存和恢复，因此存储时需要将所有寄存器状态都进行存储。

拓展练习3：完善异常中断

编程完善在触发一条非法指令异常 mret和，在 kern/trap/trap.c的异常处理函数中捕获，并对其进行处理，简单输出异常类型和异常指令触发地址，即“Illegal instruction caught at 0x(地址)”，“ebreak caught at 0x（地址）”与“Exception type:Illegal instruction"，“Exception type: breakpoint”。

编程实现如下：

```
case CAUSE_ILLEGAL_INSTRUCTION:
     // 非法指令异常处理
     /* LAB1 CHALLENGE3   YOUR CODE :  */
    /*(1)输出指令异常类型（ Illegal instruction）
     *(2)输出异常指令地址
     *(3)更新 tf->epc寄存器
    */
    cprintf("Exception type:Illegal instruction\n");
    cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
    tf->epc += 4;
    break;
case CAUSE_BREAKPOINT:
    //断点异常处理
    /* LAB1 CHALLLENGE3   YOUR CODE :  */
    /*(1)输出指令异常类型（ breakpoint）
     *(2)输出异常指令地址
     *(3)更新 tf->epc寄存器
    */
    cprintf("Exception type: breakpoint\n");
    cprintf("ebreak caught at 0x%08x\n", tf->epc);
    tf->epc += 2; //这里由于ebreak占2个字节，所以下一条指令偏移为2
    break;
```

在kern_init中加入两语句：

```
asm("mret");
asm("ebreak");
```

得到如下结果：

```
Kernel executable memory footprint: 17KB
++ setup timer interrupts
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Exception type:Illegal instruction
Illegal instruction caught at 0x8020004e
Exception type: breakpoint
ebreak caught at 0x80200052
```