Lab0.5 比麻雀更小的麻雀（最小可执行内核）

练习1 :使用GDB验证启动流程

问题

> 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。

回答

1.复位

通常RISC-V计算机上电时，硬件会进行复位，所有寄存器被清零，系统状态初始化，程序计数器被设置为特定地址。

现在我们进入到riscv64-ucore-labcodes下的lab0中，同时打开两个终端，分别运行下面的命令，结合gdb和qemu源码级调试ucore：

`make debug`

`make gdb`

结果如下：

```
xxxx@xxxx-virtual-machine:~/Downloads/riscv64-ucore-labcodes/lab0$ make gdb
riscv64-unknown-elf-gdb \
    -ex 'file bin/kernel' \
    -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 bin/kernel...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb)
```

RISC-V计算机加电运行后停在了`0x1000`处，这说明QEMU模拟的这款RISC-V处理器的复位地址是`0x1000`，PC被初始化为`0x1000`。

接着使用gdb调试程序，通过命令`x/10i $pc`查看0x1000处的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
```

下面对指令进行分析：

- `auipc t0, 0x0`：将当前程序计数器的值加上立即数 0x0，结果存储到寄存器 t0 中。由于立即数为 0x0，t0 将被设置为当前 PC 的值，即 0x1000。
- `addi a1,t0,32`：将寄存器 t0 的值（0x1000）与立即数 32 相加，结果存储到寄存器 a1 中。此时，a1 的值将为 `0x1000 + 0x20 = 0x1010`。
- `csrr a0,mhartid`: 从控制和状态寄存器（CSR）中读取 mhartid 的值，并将其存储到寄存器 a0 中。mhartid 通常用于标识当前硬件线程的 ID。

我们可以通过命令`info r mhartid`查看mhartid的值：

```
(gdb) info r mhartid
mhartid         0x0        0
```

如图，mhartid的值为`0x0`，说明当前线程id为0，那么a0此时的值也为`0x0`。

`ld t0, 24(t0)`

从地址 `t0 + 24` 处加载一个64位数据到寄存器 `t0` 中。由于 `t0` 的当前值为 `0x1000`，因此将从 `0x1000 + 0x18 = 0x1018` 处加载数据，根据指令的内容可知，此时，t0此时的值为`0x80000000`。

也可以通过命令`si`单步执行这条指令，再通过命令`info t t0`查看其值：

```
0x000000000000100c in ?? ()
(gdb) si
0x0000000000001010 in ?? ()
(gdb) info r t0
t0             0x80000000       2147483648
```

`unimp`表示未实现的指令。

`jr t0`表示无条件跳转到寄存器 `t0` 中存储的地址，也就是`0x80000000`，程序将继续执行该地址处的指令。

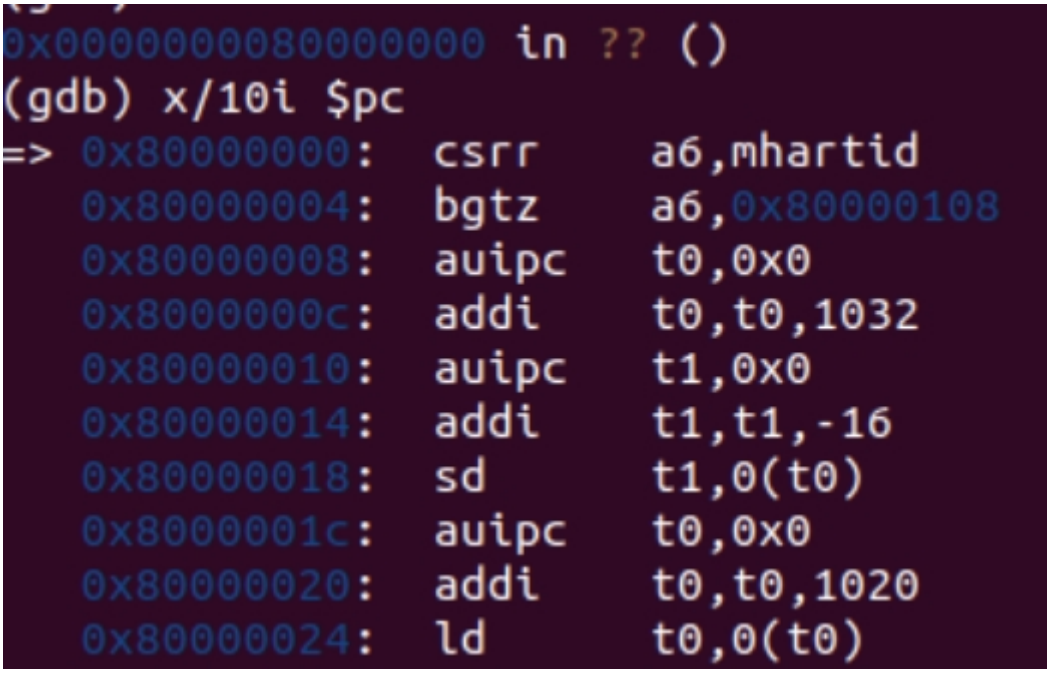
接下来通过`si`单步执行`jr t0`指令，程序跳转到`0x80000000`处继续执行，我们预先已知这个地址处加载qemu自带的`bootloader OpenSBI.bin`，启动OpenSBI固件（根据实验指导书）。

```
(gdb) si
0x0000000080000000 in ?? ()
(gdb)
```

以上是RISC-V硬件加电后的几条指令及其功能，加电后的几条指令开始于`0x1000`，总体完成的功能就是加载数据，并将程序跳转到`0x80000000`处，来启动`OpenSBI`。

2.启动OpenSBI

同样的，通过命令`x/10i $pc`查看`0x80000000`处的10条汇编指令：

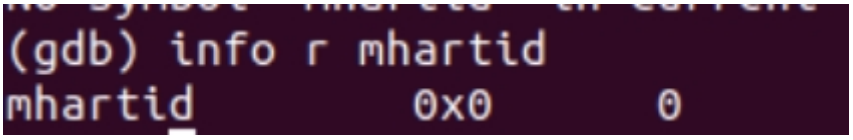


下面对指令进行分析：

`csrr a6, mhartid`

从控制和状态寄存器中读取`mhartid`的值，并将其存储到寄存器`a6`中。

可以通过命令`info r mhartid`查看`mhartid`的值：



如图，`mhartid`的值为`0x0`，当前硬件线程ID为0，因此`a6`的值也为`0x0`。

`bgtz a6, 0x80000108`

如果寄存器`a6`的值大于零，则跳转到地址`0x80000108`。这通常用于检查当前硬件线程是否有效。

之后的`auipc`、`addi`、`sd`和`ld`指令则涉及初始化和寄存器的设置，这些指令过后，`t0`的值为`0x80000418`处加载的双字，t1的值为`0x80000000`。

3.内核启动

从这里开始实际上已经是执行应用程序的第一条指令（即跳转到`0x80200000`）这个阶段后的内容，主要用于内核的初始化。

我们预先已知内核镜像`os.bin`被加载到以物理地址`0x80200000`开头的区域上，通过命令`break *0x80200000`在`0x80200000`处设置断点：

可以看到输出了一个文件路径kern/init/entry.S，在这里会进行内核栈的分配，然后转入C语言编写的内核初始化函数。

接下来我们打开`entry.S`，下面是`entry.S`里的代码内容：

```
#include <mmu.h>
#include <memlayout.h>

    .section .text,"ax",%progbits
    .globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

接下来对其中的关键代码进行分析：

`.globl kern_entry`:这里的`kern_entry`标签，是一个全局的符号，也是定义的内核的入口点。

`kern_entry`:

- `la sp, bootstacktop`:将标签bootstacktop的地址加载到寄存器sp中，bootstacktop是内核栈的顶部，内核栈从高地址向低地址增长。这一句是为了在内核启动时初始化内核栈，以便接下来的函数调用和局部变量能够正常工作。
- `tail kern_init`: 使用tail指令调用kern_init，跳转到kern_init函数执行，而不保留当前函数的栈帧，并且会将返回地址设置为跳转后的第一条指令的地址。
- `.space KSTACKSIZE`:这条指令用于在数据段中预留KSTACKSIZE字节的空间，KSTACKSIZE就是内核栈的大小。

总体来看，`entry.S`就负责定义内核的入口点，设置内核栈，并跳转到`kern_init`进行后续的初始化工作。

我们还可以通过命令`x/10i 0x80200000`查看`0x80200000`处的10条汇编指令，发现确实在`0x80200008`处跳转到`kern_init`。

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

接下来，通过命令`continue`继续执行程序，直到我们设置的断点处`0x80200000`：

```
(gdb) continue
Continuing.

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7           la sp, bootstacktop
```

可以发现，执行`make debug`的终端有新的输出：

```
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)
```

证明OpenSBI此时已经启动。

前面我们分析出，内核的入口点会跳转到`kern_init`继续执行，因此，我们可以通过命令`break kern_init`在`break_init`处设置断点，得到输出：

```
(gdb) break kern_init
Breakpoint 2 at 0x8020000a: file kern/init/init.c, line 8.
```

可以看到输出了一个文件路径kern/init/init.c，这是C语言编写的内核入口点，从kern/entry.S跳转过来完成其他初始化工作。

进入`init.c`查看c代码：

```c
#include <stdio.h>
#include <string.h>
#include <sbi.h>
int kern_init(void) __attribute__((noreturn));

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)
        ;
}
```

这段代码里面包含`kern_init()`函数，这个函数使用了`__attribute__((noreturn))`属性，表明这个函数不会正常返回。

- `memset(edata, 0, end - edata)`:这行代码使用memset函数将从edata 到 end 之间的内存区域全部清零。这是为了初始化 .bss 段，以确保未初始化的数据部分开始时是零，避免读取垃圾值。
- `while (1)` :内核初始化时，会进入一个无限循环，保证了操作系统在启动时处于一个已知的、可控的状态，准备进行后续的操作。

总体来说，在内核初始化时，它清空数据段，打印加载信息，最后进入一个无限循环以阻止程序流的继续。

我们还可以通过命令`x/20i kern_init`查看`kern_init`处的20条汇编指令，发现在`0x8020003c`处的指令`<kern_init+48>:  j  0x8020003a <kern_init+48>`跳回了自己开始的地址，所以代码一直循环下去。

```
(gdb) x/20i 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
   0x80200020 <kern_init+22>:   sd      ra,8(sp)
   0x80200022 <kern_init+24>:   jal     ra,0x802004b6 <memset>
   0x80200026 <kern_init+28>:   auipc   a1,0x0
   0x8020002a <kern_init+32>:   addi    a1,a1,1186
   0x8020002e <kern_init+36>:   auipc   a0,0x0
   0x80200032 <kern_init+40>:   addi    a0,a0,1210
   0x80200036 <kern_init+44>:   jal     ra,0x80200056 <cprintf>
   0x8020003a <kern_init+48>:   j       0x8020003a <kern_init+48>
   0x8020003c <cputch>: addi    sp,sp,-16
   0x8020003e <cputch+2>:       sd      s0,0(sp)
   0x80200040 <cputch+4>:       sd      ra,8(sp)
   0x80200042 <cputch+6>:       mv      s0,a1
   0x80200044 <cputch+8>:       jal     ra,0x8020008c <cons_putc>
```

继续通过`continue`命令执行程序，观察debug窗口，发现有新输出，打印了init.c中的常量字符串message中的内容：`”(THU.CST) os is loading ...”`

```
(THU.CST) os is loading ...
```

QEMU启动完成

本实验中的重要知识点

QEMU启动流程

加电 -> OpenSBI启动 -> 跳转到`0x80200000 (kern/init/entry.S)`->进入`kern_init()`函数（kern/init/init.c) ->调用`cprintf()`输出一行信息->结束

Bootloader

- Bootloader的定义：它是一个小型程序，负责在设备开机时初始化硬件，加载操作系统与引导系统到内存中，并转交控制权给操作系统。
- Bootloader的基本功能：
    1. 硬件初始化：设置处理器和外设，配置内存，准备系统运行。
    2. 加载操作系统：从指定的存储介质（如硬盘、闪存等）中读取操作系统映像，将其加载到内存中。
    3. 转交控制权：完成内核加载后，将执行权转交给内核。

在QEMU模拟的riscv计算机里，我们使用QEMU自带的bootloader: OpenSBI固件，它是 RISC-V 生态系统中的一个开源软件包，提供了一个标准的接口，用于在 RISC-V 处理器上实现超级用户模式和特权操作。

本次实验中，在 Qemu 开始执行任何指令之前，作为 bootloader 的 OpenSBI.bin 被加载到物理内存以物理地址 `0x80000000` 开头的区域上，同时内核镜像 os.bin 被加载到以物理地址 `0x80200000` 开头的区域上。

在计算机中，固件(firmware)是一种特定的计算机软件，它为设备的特定硬件提供低级控制，也可以进一步加载其他软件。

复位地址

复位地址的定义

复位地址指的是CPU在上电的时候，或者按下复位键的时候，PC被赋的初始值。

RISC-V 架构中的复位地址

在 RISC-V 架构中，允许芯片的实现者自主选择复位地址复位地址，通常是 `0x80000000` 或 `0x00000000`，具体取决于系统的配置和实现。在此实验中，QEMU模拟的riscv处理器复位地址为 `0x1000`

复位后，处理器会从该地址开始执行指令，主要是将计算机系统的各个组件（包括处理器、内存、设备等）置于初始状态，并且会启动Bootloader，在本实验QEMU的复位代码指定加载Bootloader的位置为 `0x80000000`，Bootloader将加载操作系统内核并启动操作系统的执行。

Lab1 断,都可以断

练习1：理解内核启动中的程序入口操作

问题

> 阅读 kern/init/entry.S内容代码，结合操作系统内核启动流程，说明指令 `la sp, bootstacktop` 完成了什么操作，目的是什么？ `tail kern_init` 完成了什么操作，目的是什么？

解答

1. `la sp, bootstacktop` 这条指令的目的是在系统启动的早期阶段，将堆栈指针 `sp` 设置为指向 `bootstack` 的顶部，为系统启动和初始化过程中的函数调用提供了必要的堆栈空间，并确保了系统行为的可预测性和安全性。我们在entry.S中观察到后续指令定义了 `bootstack`，来用于栈分配一定的字节空间，并在后续完成了栈顶的定义。通过将 `bootstacktop` 的地址加载到 `sp` 寄存器

中，将栈顶指针初始化为内核栈的顶部，从而指示栈从高地址向低地址增长。因此该指令的作用是将栈指针 `sp` 放置于栈顶，后续内核栈从高地址向低地址增长。

2. `tail` 是一种函数约定，表明该函数调用是尾递归的,所以 `tail kern_init` 是一个尾调用指令，将控制权传递给 `kern_init` 函数。`kern_init` 是内核的初始化入口点，它负责完成操作系统的初始化工作，包括初始化硬件、建立内存映射、初始化设备驱动、创建进程等。通过执行该指令，使程序首先执行真正的入口点 `kern_init`，从而启动操作系统的内核。
3. 总体来看,kern/init/entry.S代码主要功能是设置内核的启动堆栈，并将控制权传递给内核的初始化函数 `kern_init`。这是操作系统启动过程中的一个关键步骤，它确保了内核在初始化过程中有足够的堆栈空间来执行其任务，并且能够正确地开始执行初始化代码。

练习2：完善中断处理

问题

> 请编程完善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。
>
> 要求完成问题1提出的相关函数实现，提交改进后的源代码包（可以编译执行），并在实验报告中简要说明实现过程和定时器中断中断处理的流程。实现要求的部分代码后，运行整个系统，大约每1秒会输出一次”100 ticks”，输出10行。

回答

完善代码：

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

- **实现过程**：首先调用 `clock_set_next_event` 函数，设置下次的中断时间，为当前时间加上100000。接着 `ticks` 自增1，如果 `ticks` 为100，调用 `print_ticks` 打印 `ticks`，将 `ticks` 置0，`num` 自增1。如果 `num` 增加到10，调用 `sbi_shutdown` 关机。
- **定时器中断处理流程**：

  中断触发：当定时器中断发生时，CPU 将中断信号传递给处理程序，进入 `interrupt_handler` 函数。

  识别中断类型：处理函数首先检查中断原因，通过 `tf->cause` 获取当前的中断类型。定时器中断的类型为 `IRQ_S_TIMER`。

  设置下次定时器中断：在 `IRQ_S_TIMER` 分支中，首先调用 `clock_set_next_event()` 函数，设置下一个定时器中断的时间。这是为了确保系统能够继续接收定时器中断。

计数器增量：每当定时器中断被处理时，`ticks` 计数器增加 1。该计数器用于跟踪自上次打印以来发生的中断次数。

检查计数器：代码检查 `ticks` 是否达到 100。如果达到了 100，则调用 `print_ticks()` 函数，打印出 100 个时钟中断的消息，并重置 `ticks` 为 0。同时，将 `num` 计数器加 1，以记录已打印的次数。

检查打印次数并关机：代码随后检查 `num` 是否达到了 10。如果 `num` 达到 10，表示已经处理了 1000 次定时器中断（10 次打印），然后调用 `sbi_shutdown()` 函数进行关机处理。

结束处理：完成所有处理后，`interrupt_handler` 函数结束。控制权将返回到上层代码，继续执行被中断的程序。

Challenge1：描述与理解中断流程

问题

> 描述ucore中处理中断异常的流程（从异常的产生开始）

回答

异常的产生与检测

- 内部中断（异常）：由CPU内部产生，指在执行一条指令的过程中发生了错误，此时我们通过中断来处理错误。长江的有除零错误、地址访问越界、发生缺页等。他们中有的可以恢复(如缺页)，有的不可恢复(如除零)，只能终止程序执行。
- 陷入(Trap)，指我们主动通过一条指令停下来，并跳转到处理函数。常见的形式有通过ecall进行系统调用(syscall)，或通过ebreak进入断点(breakpoint)。
- 外部中断（Interrupt）：由外设（如键盘、鼠标、磁盘等）产生，当外设需要CPU的注意或处理时，会通过中断控制器向CPU发送中断请求。

由于中断处理需要进行较高权限的操作，中断处理程序一般处于内核态。在RISC-V里，中断(interrupt)和异常(exception)统称为"trap"。

我们以时钟中断为例：时钟中断通过调用 `openSBI` 固件中提供的 `SBI_SET_TIMER` 接口实现。通过使用 `sbi_call` 方法调用 `openSBI` 的接口。封装一个 `sbi_set_timer()` 函数，通过调用这个函数并传参，即可实现按照指定时间对 cpu 发送一个时钟中断。随后，cpu 会在硬件层面检测到这个时钟中断信号，触发中断的处理流程。

中断请求的处理与跳转

中断控制器接收：中断控制器将接收外设的中断请求（IRQ），并将其转换为中断号（INT）。RISC-V架构有个CSR叫做stvec，即所谓的”中断向量表基址”。中断向量表的作用就是把不同种类的中断映射到对应的中断处理程序。如果只有一个中断处理程序，那么可以让stvec直接指向那个中断处理程序的地址。

CPU响应中断：CPU暂停当前正在执行的程序，保存当前程序的执行现场（包括程序计数器PC、状态寄存器、通用寄存器等），以便之后恢复执行。

```
//kern/trap/trap.c
void idt_init(void) {
  extern void __alltraps(void);
  /* Set sscratch register to 0, indicating to exception vector that we are
   \* presently executing in the kernel */
    //在内核态将sscratch置零
  write_csr(sscratch, 0);
  /* Set the exception vector address */
  //我们保证__alltraps的地址是四字节对齐的，将__alltraps这个符号的地址直接写到stvec寄存器
  write_csr(stvec, &__alltraps);
}
```

异常处理向量表有两种模式，即直接模式和向量模式，直接模式，即 stvec 处只有一个异常处理程序。向量模式则是在 `stvec` 处存在一个向量表，需要根据异常的类型进行映射到对应的异常处理程序。

> 在发现异常后，`pc` 会切换到 `stvec` 的值。本次实验中，在trap.c代码中使用的是直接模式，即直接跳转到中断处理函数的入口点，即 `__alltrap` 处。接下来对异常与中断进行初步的处理，保存cpu先前的各个寄存器状态，保证cpu在中断处理后 能够正常恢复。

保存与恢复被中断的程序执行

异常或中断发生后，将所有的寄存器保存到栈中，包括程序的返回值、程序计数器、基指针等，在异常恢复后，还要从内存（栈）恢复CPU的各个寄存器值

本次代码中，通过 `__alltrap` 中的宏定义 `SAVE_ALL` 实现现场信息的封装，以此来完成上下文切换机制。为了方便我们组织上下文的数据（几十个寄存器），我们定义一个结构体 `trapFrame`。一个 `trapFrame` 结构体占据36个 `uintptr_t` 的空间，里面依次排列通用寄存器x0到x31,然后依次排列4个和中断相关的CSR, 我们希望中断处理程序能够利用这几个CSR的数值，利用栈顶指针 `sp` 把一个 `trapFrame` 结构体放到了栈顶。

而当中断处理结束后，需要重新读取这块内存上存储的寄存器值和 `sstatus`、`sepc` 两个和恢复状态相关的寄存器，这里通过进入类似于 `__alltrap` 的 `__trapret` 入口点实现，其中的宏定义 `RESTORE_ALL` 即为读取这段栈空间的值到寄存器上，恢复中断处理前的现场，并调用 sret 从内核态返回。

```
  .globl __alltraps
.align(2)

__alltraps:
  SAVE_ALL
  move  a0, sp
  jal trap
  # sp should be the same as before "jal trap"
  .globl __trapret
__trapret:
  RESTORE_ALL
   # return from supervisor call
  sret
```

异常与终端在trap函数的具体处理

在trap.c代码中，是将中断处理、异常处理的任务分别分配给了`interrupt_handler()`，`exception_handler()`，上述的函数将会根据中断或异常的不同类型来具体处理。

```
/* trap_dispatch - dispatch based on what type of trap occurred */
static inline void trap_dispatch(struct trapframe *tf) {
    //scause的最高位是1，说明trap是由中断引起的
  if ((intptr_t)tf->cause < 0) {
 // interrupts
interrupt_handler(tf);
  } else {
// exceptions
exception_handler(tf);
  }
}
void trap(struct trapframe *tf) { trap_dispatch(tf); }
```

可以看到，进入`trap_dispatch()`函数后，将会进行判断，然后再进入`interrupt_handler()`或`exception_handler()`函数中执行对应中断的处理内容。`interrupt_handler()`和`exception_handler()`主要是依据scause的数值更仔细地条件判断，然后进行一些输出便直接返回。switch里的各种case, 如`IRQ_U_SOFT`,`CAUSE_USER_ECALL`,是riscv ISA 标准里规定的。

问题

> mov a0，sp的目的是什么?

回答

在`mov a0，sp`指令中，将栈顶指针sp的值传给了a0,而a0存放的就是汇编宏`SAVE_ALL`得到的`trapFrame`结构体的首地址，即将此次中断新生成的`trapFrame`结构体作为参数，传递给trap.c中的`trap()`函数，从而实现中断。

问题

> SAVE_ALL中寄寄存器保存在栈中的位置 是什么确定的?

回答

在`SAVE_ALL`中，寄存器保存在栈中的位置是通过栈指针`sp`和偏移量得出。在C语言结构体中，若干个变量在内存中都是直线排列，所以一个`trapFrame`结构体占据36个`uintptr_t`的空间，其中依次排列通用寄存器 x0到 x31（`trapFrame`还有4个和中断相关的控制状态寄存器）。综上，寄存器保存在栈中的位置便可先通过栈顶指针`sp`，以及固定的偏移量，再乘以索引号便可确定，来找到我们想要的寄存器。
对于一些特殊寄存器的 CSR，按照`sstatus`、`sepc`、`sbadaddr`、`scause`的顺序存储在通用寄存器的高地址处，所以同样可以由固定的偏移量乘以索引号来找到。

问题

> 对于任何中断，`__alltraps`中都需要保存所有寄存器吗?

回答

产生异常或中断后，大部分情况下`__alltraps`中都需要保存所有寄存器。主要是因为中断或异常随时会发生，从而导致处理器的状态可能被修改，因此需要保存所有寄存器的值保证能正确的返回到被中断的程序。由于不同的中断类型可能需要保存的寄存器不同，因此在`__alltraps`中保存所有寄存器可以保证处理器状态的完整性。
但是小部分情况下具体的异常或中断处理程序的设计和要求可能会有所不同，可以选择只保存必要的寄存器以减小处理程序的开销。

Challenge2：理解上下文切换机制

问题

> 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？

回答

问题：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？

`csrw sscratch, sp`：

- 这条指令的作用是将当前栈指针`sp`的值写入`sscratch`。`sscratch`是一个用于保存特定临时值的寄存器，通常用于异常或中断处理时存储必要的信息。

`csrrw s0, sscratch, x0`：

- 这条指令的作用是将`sscratch`中的值读出并存入寄存器`s0`，同时将`x0`（即零寄存器）写入`sscratch`。结果是将`sscratch`清空，并将之前存储在其中的值（即`sp`的值）加载到`s0`中。

这些操作的主要目的是在处理中断或异常时，保存和恢复当前的执行上下文。通过将栈指针保存到`sscratch`中，可以在处理完中断或异常后恢复到正确的执行状态。

不还原那些`csr`，是因为异常已经由`trap`处理过了，没有必要再去还原。它们包含有关导致异常或中断的信息，这些信息在处理异常或中断时可能仍然需要。在异常或中断处理程序中，这些`csr`可能需要被读取以确定异常的原因或其他相关信息。保存这些寄存器的值允许在处理异常后能够进行调试，查看导致异常的原因。某些情况下，异常处理可能会导致状态的改变。即使不恢复之前的状态，保存的值可以用于后续决策或处理。在处理中断或异常时，保存所有相关寄存器状态有助于保持系统的一致性，以便于进行正确的异常管理。

Challenge3：完善异常中断

问题

> 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”。

回答

在本challenge中，需要在kern/trap/trap.c的异常处理函数中捕获非法指令异常与断点异常，输出指令异常类型，输出异常指令地址并更新tf->epc寄存器：

完善后的代码如下：

```
void exception_handler(struct trapframe *tf)
{
    switch (tf->cause)
    {
    case CAUSE_MISALIGNED_FETCH:
        break;
    case CAUSE_FAULT_FETCH:
        break;

    case CAUSE_ILLEGAL_INSTRUCTION:
        // 非法指令异常处理
        /* LAB1 CHALLENGE3   2213906 : */
       /*(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 exception at 0x%016llx\n", tf->epc);
        tf->epc += 4;
        break;
    case CAUSE_BREAKPOINT:
        //断点异常处理
        /* LAB1 CHALLLENGE3   2213906 : */
        /*(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%016llx\n", tf->epc);
        tf->epc += 4;
        break;
        ……其他代码
        }
}
```

下面对完善的代码进行分析：

异常处理进行以下工作：

- **输出异常类型：**

在捕获到非法指令异常时，调用`cprintf`函数输出`"Exception type:Illegal instruction"`，在捕获到断点异常时，调用cprintf函数输出`"Exception type: breakpoint "`。

- **输出异常类型和异常指令触发地址：**

`epc`寄存器是异常发生时的程序计数器，保存发生异常的指令地址。在遇到异常指令时想要输出异常指令触发地址，需要将`epc`的值以16进制的形式输出，即可得到相应地址。

`0x%016llx``是一个格式化字符串，在输出时以十六进制格式显示一个 64 位整数。0x是一个固定的字符串，表示后面的数字是以16进制格式显示的。%标记了格式控制的开始。0表示用零填充空位，16则指定输出的总宽度为16个字符。ll是长度修饰符，表示要打印long long类型的整数。x是格式控制符，表示以十六进制格式输出整数。

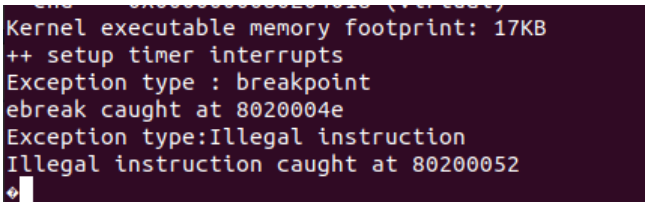
- **更新epc的值，跳过异常指令：**

epc寄存器是程序计数器，此时保存引起异常的指令的地址，在RISC-V中，所有标准的原始指令（包括非法指令和ebreak指令）均为32位长，即4个字节，因此，epc的值要加4，即可跳过异常指令，正常执行后面的指令。

然后我在`kern/init/init.c`文件中的`kern_init()`函数中插入了两条汇编指令`asm volatile("ebreak")`和`asm volatile(".word 0xFFFFFFFF")`来触发两种异常，对异常处理进行测试：

```
int kern_init(void)

{

  extern char edata[], end[];

  memset(edata, 0, end - edata);

  cons_init(); // init the console

  const char *message = "(THU.CST) os is loading ...\n";

  cprintf("%s\n\n", message);

  print_kerninfo();

  // grade_backtrace();

  idt_init(); // init interrupt descriptor table

  // rdtime in mbare mode crashes

  clock_init(); // init clock interrupt

  intr_enable(); // enable irq interrupt

  **asm volatile("ebreak"); // 断点异常**

  **asm volatile(".word 0xFFFFFFFF"); // 非法指令异常**

  while (1)

   ;

}
```

然后在lab1下打开终端，执行make qemu命令，运行ucore，结果如下：



中断被正常处理。

本实验重要的知识点：

异常与中断处理:

- 异常(Exception)，指在执行一条指令的过程中发生了错误，此时我们通过中断来处理错误。最常见的异常包括：访问无效内存地址、执行非法指令(除零)、发生缺页等。他们有的可以恢复(如缺页)，有的不可恢复(如除零)，只能终止程序执行。
- 外部中断(Interrupt)，简称中断，指的是 CPU 的执行过程被外设发来的信号打断，此时我们必须先停下来对该外设进行处理。典型的有定时器倒计时结束、串口收到数据等。