

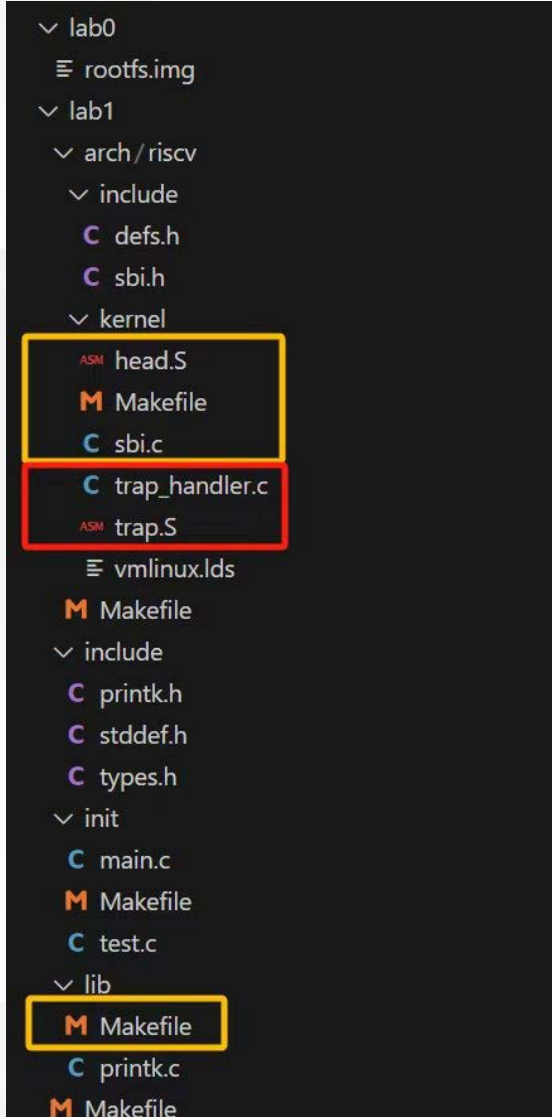
Lab1指南

Lab1 操作内容

- 学习 RISC-V 汇编，编写 head.S 实现跳转到内核运行的第一个 C 函数；
- 学习 OpenSBI，理解 OpenSBI 在实验中所起到的作用，并调用 OpenSBI 提供的接口完成字符的输出；
- 学习 Makefile 相关知识，补充项目中的 Makefile 文件，来完成对整个工程的管理；
- 学习 RISC-V 的 trap 处理相关寄存器与指令，完成对 trap 处理的初始化；
- 理解 CPU 上下文切换机制，并正确实现上下文切换功能；**
- 编写 trap 处理函数，完成对特定 trap 的处理；**
- 调用 OpenSBI 提供的接口，完成对时钟中断事件的设置。**

最后，最重要的是多动手，才能了解CPU体系结构的概念

Lab1 操作内容



·sbi.c
把RISCV ecall包装成C语言调用

·head.S
kernel的第一行代码

·Makefile
照猫画虎，完成make项目构建

+trap.S
进入内核前面向C语言对寄存器的包装

+trap_handler
内核入口函数，以及时钟中断相关实现

可以且建议另写clock.c

中断/异常/陷入 *Interrupt/Exception/Trap*

·中断/异常/陷入是什么？

在CPU执行一条指令时，PC会在这条指令执行完后，变为下一条指令所在的地址或是当前指令跳转到的目标地址，即执行流

为了使CPU增加处理能力，硬件工程师们给CPU设计了异常执行流的概念。异常执行流指CPU在收到软件异常/外部信号时会主动放弃当前执行流的正常跳转，进而改变CPU状态后，跳转到指定的地址执行中断处理程序

在lab1中实现的内核，需要在test代码的死循环下跳转并处理外部时钟中断，而RISC-V下的时钟中断会引起CPU的状态机改变，如下：

- PC <= stvec
- sepc <= PC
- scause <= ?
- sstatus <= ?
- stie <= ?

中断/异常/陷入 *Interrupt/Exception/Trap*

```
0x8020016c <test>      addi    sp,sp,-32
0x80200170 <test+4>     sd      ra,24(sp)
0x80200174 <test+8>      sd      s0,16(sp)
0x80200178 <test+12>    addi    s0,sp,32
0x8020017c <test+16>    sw      zero,-20(s0)
0x80200180 <test+20>    lw      a5,-20(s0)
0x80200184 <test+24>    mv      a1,a5
0x80200188 <test+28>    auipc   a0,0x1
0x8020018c <test+32>    addi    a0,a0,-352
0x80200190 <test+36>    jal     ra,0x80200700 <printk>
0x80200194 <test+40>    lw      a5,-20(s0)
0x80200198 <test+44>    addiw   a5,a5,1
0x8020019c <test+48>    sw      a5,-20(s0)
0x802001a0 <test+52>    sw      zero,-24(s0)
0x802001a4 <test+56>    j      0x802001c0 <test+84>
0x802001a8 <test+60>    lw      a5,-24(s0)
0x802001ac <test+64>    addiw   a5,a5,1
0x802001b0 <test+68>    sw      a5,-24(s0)
0x802001b4 <test+72>    lw      a5,-24(s0)
0x802001b8 <test+76>    addiw   a5,a5,1
> 0x802001bc <test+80>    sw      a5,-24(s0)
0x802001c0 <test+84>    lw      a5,-24(s0)
0x802001c4 <test+88>    sext.w  a4,a5
0x802001c8 <test+92>    lui     a5,0x800
0x802001cc <test+96>    blt     a4,a5,0x802001a8 <test+60>
0x802001d0 <test+100>   j      0x80200180 <test+20>
```

```
0x80200024 <_traps>      addi    sp,sp,-4
0x80200028 <_traps+4>    sd      zero,0(sp)
0x8020002c <_traps+8>    sd      zero,0(sp)
0x80200030 <_traps+12>   sd      zero,0(sp)
0x80200034 <_traps+16>   sd      zero,0(sp)
0x80200038 <_traps+20>   sd      zero,0(sp)
0x8020003c <_traps+24>   sd      zero,0(sp)
0x80200040 <_traps+28>   sd      zero,0(sp)
0x80200044 <_traps+32>   sd      zero,0(sp)
0x80200048 <_traps+36>   sd      zero,0(sp)
0x8020004c <_traps+40>   sd      zero,0(sp)
0x80200050 <_traps+44>   sd      zero,0(sp)
0x80200054 <_traps+48>   sd      zero,0(sp)
0x80200058 <_traps+52>   sd      zero,0(sp)
0x8020005c <_traps+56>   sd      zero,0(sp)
0x80200060 <_traps+60>   sd      zero,0(sp)
0x80200064 <_traps+64>   sd      zero,0(sp)
0x80200068 <_traps+68>   sd      zero,0(sp)
0x8020006c <_traps+72>   sd      zero,0(sp)
0x80200070 <_traps+76>   sd      zero,0(sp)
0x80200074 <_traps+80>   sd      zero,0(sp)
0x80200078 <_traps+84>   sd      zero,0(sp)
0x8020007c <_traps+88>   sd      zero,0(sp)
0x80200080 <_traps+92>   sd      zero,0(sp)
0x80200084 <_traps+96>   sd      zero,0(sp)
0x80200088 <_traps+100>    sd      zero,0(sp)
0x8020008c <_traps+104>  sd      zero,0(sp)
0x80200090 <_traps+108>  sd      zero,0(sp)
0x80200094 <_traps+112>  sd      zero,0(sp)
0x80200098 <_traps+116>  sd      zero,0(sp)
0x8020009c <_traps+120>  sd      zero,0(sp)
0x802000a0 <_traps+124>  sd      zero,0(sp)
0x802000a4 <_traps+128>  sd      zero,0(sp)
0x802000a8 <_traps+132>  sd      zero,0(sp)
0x802000ac <_traps+136>  sd      zero,0(sp)
0x802000b0 <_traps+140>  sd      zero,0(sp)
0x802000b4 <_traps+144>  addi    sp,sp,4
0x802000b8 <_traps+148>  sret
```

上下文 *Context*

·程序是什么？

Algorithms + Data Structures = Programs

尼古拉斯·沃斯 (Niklaus Wirth)

·CPU上下文是什么？

CPU上下文可以简单理解为当前的CPU状态及mem内容

CPU状态：通用寄存器内容、当前PC、跟进程相关的CSR寄存器内容

mem内容：内存内容（指令内容、变量内容、栈内容）

·为什么要学习CPU上下文的概念？

因为操作系统内核依靠中断/异常/系统调用运行，恢复执行流需要保存/恢复上下文

你也不想你的程序跑飞吧.jpg

·如何直观理解CPU上下文？

- 用gdb进入单步调试，使用layout命令查看寄存器与汇编指令在每次执行后的变化

- 在预期执行的位置设置断点

上下文 Context

The screenshot displays a QEMU debugger window with two main panels. The top panel, titled 'Register group: general', shows the current state of the CPU registers. The bottom panel shows the disassembled instructions at the current PC value.

Register Group: general

Register	Value	Register	Value	Register	Value	Register	Value
zero	0x0	ra	0x80078cc	sp	0x80202000	gp	0x0
t2	0x1000	fp	0x80017f20	s1	0x1	a0	0x0
a4	0x80	a5	0x1	a6	0x5	a7	0x800130b0
s5	0x0	s6	0x800000a000068	s7	0x800120e8	s8	0x80013100
t3	0x8001132e	t4	0x2	t5	0x27	t6	0x0
						pc	0x80200004

Disassembled Instructions:

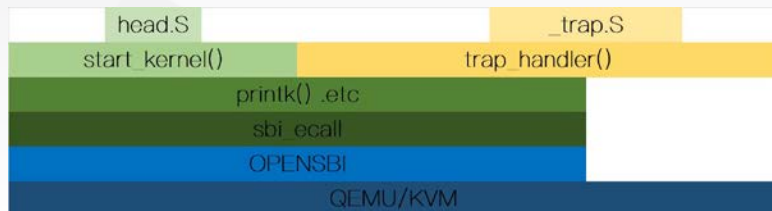
```
B+ 0x80200000 <_stext>      auipc  sp,0x2
> 0x80200004 <_stext+4>      ld      sp,8(sp)
0x80200008 <_stext+8>      jal      ra,0x802000b4 <start_kernel>
0x8020000c <_traps>         sret
0x80200010 <_sbi_ecall>      addi    sp,sp,-112
0x80200014 <_sbi_ecall+4>    sd      s0,104(sp)
0x80200018 <_sbi_ecall+8>    addi    s0,sp,112
0x8020001c <_sbi_ecall+12>   sd      a2,-64(s0)
0x80200020 <_sbi_ecall+16>   sd      a3,-72(s0)
0x80200024 <_sbi_ecall+20>   sd      a4,-80(s0)
0x80200028 <_sbi_ecall+24>   sd      a5,-88(s0)
0x8020002c <_sbi_ecall+28>   sd      a6,-96(s0)
0x80200030 <_sbi_ecall+32>   sd      a7,-104(s0)
0x80200034 <_sbi_ecall+36>   mv      a5,a0
0x80200038 <_sbi_ecall+40>   sw      a5,-52(s0)
0x8020003c <_sbi_ecall+44>   mv      a5,a1
0x80200040 <_sbi_ecall+48>   sw      a5,-56(s0)
0x80200044 <_sbi_ecall+52>   ld      a0,-64(s0)
0x80200048 <_sbi_ecall+56>   ld      a1,-72(s0)
0x8020004c <_sbi_ecall+60>   ld      a2,-80(s0)
0x80200050 <_sbi_ecall+64>   ld      a3,-88(s0)
0x80200054 <_sbi_ecall+68>   ld      a4,-96(s0)
0x80200058 <_sbi_ecall+72>   ld      a5,-104(s0)
```

remote Thread 1.1 In: _stext L9 PC: 0x80200004

图中的内容标识了目前qemuCPU的寄存器值，以及接下来的指令，可以简单理解为CPU的上下文

Lab1-lab7中，需要你将这些寄存器在发生中断时保存到一个特定的位置，然后从中断返回时将其恢复成中断前的样子

时钟中断?



右图的汇编中存在三个处理步骤

- 中断引起的CPU状态机改变，及上下文保存
- 调用trap_handler
- 中断后的上下文恢复，以及CPU状态机的改变

```
lab1 > init > C main.c > ...
1  #include "printk.h"
2  #include "sbi.h"
3
4  extern void test();
5
6  int start_kernel() {
7      printk("2022");
8      printk(" Hello RISC-V\n");
9
10     test(); // DO NOT DELETE !!!
11
12     return 0;
13 }
14
15 void test() {
16     while (1){
17         int i = 1;
18         i = 2;
19         i = 3;
20     };
21 }

rch > riscv > kernel > ASM trap.S
.section .text.entry
.align 2
.globl _traps
_traps:
# -----
# 1. save 32 registers and sepc to stack
sd x0,0(sp)
# -----
# 2. call trap_handler
# -----
# 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
# -----
# 4. return from trap
ret
# -----
```


CSR Control and Status Registers

·CSR寄存器是什么

CSR寄存器即**控制与状态寄存器**，与通用寄存器(a0-a7,s0-s10,ra,sp,gp,tp,zero.etc)加以区分

使用**csr**指令可以实现对指定CSR寄存器bit的更改/设1/设0的操作，在这些操作之后，CPU会发生一些副作用（side effect），例如允许外部时钟中断、指定中断向量、指定协处理器行为

在发生一些事件时，CSR寄存器的值会由CPU硬件主动改变，不同状态下CPU运行的代码允许读写的CSR寄存器有限制，低权限态可读写的寄存器为高权限态的子集（例如M态可以读写sstatus但是S态不能读写mstatus）

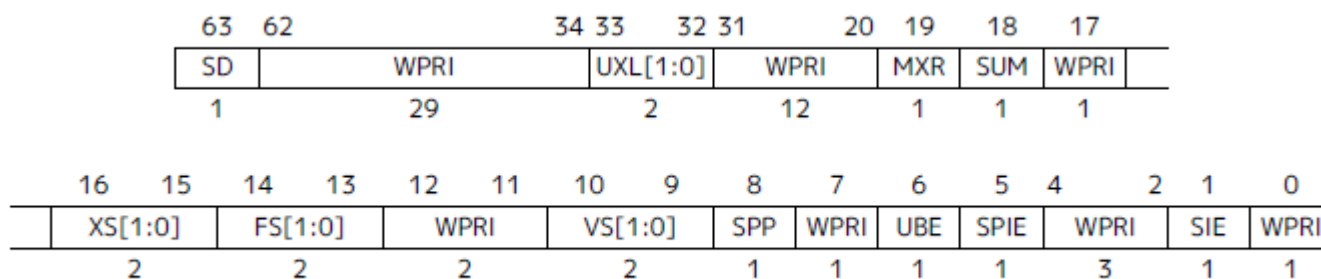


Figure 43. Supervisor-mode status register (sstatus) when SXLEN=64.

```
csrsi sstatus,(1<<1) #enable interrupt
```

CSR *Control and Status Registers*

✓ 10.1.1. Supervisor Status Register (sstatus)

10.1.1.1. Base ISA Control in sstatus Register

10.1.1.2. Memory Privilege in sstatus Register

10.1.1.3. Endianness Control in sstatus Register

·指定浮点协处理器的行为

·中断使能/中断返回恢复/大小端行为

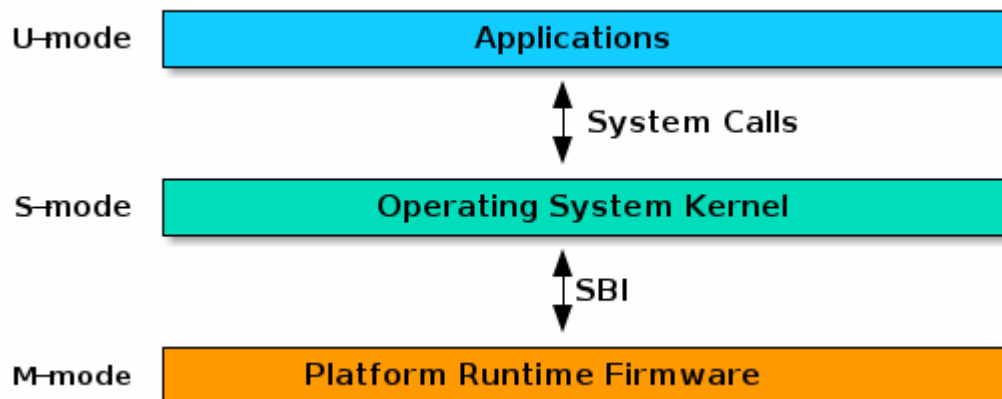
sstatus寄存器涉及到的功能就有两面单独的篇幅，功能之多可见一斑

SBI Supervisor Binary Interface

.SBI是什么

SBI是RISC-V平台的S-mode与M-mode之间的一套接口规范，内容涵盖了启动时如何加载内核、运行时的S态下的系统调用规范

openSBI是RISC-V官方实现的一套SBI，本学期的实验主要用到其中的三个功能（PUTCHAR,SETTIMER,GETCHAR）；大家需要在对应的接口处实现C转汇编，然后在S-mode的程序中通过ecall指令进入openSBI调用



```
struct sbiret sbi_ecall(int ext, int fid, uint64 arg0,
                        uint64 arg1, uint64 arg2,
                        uint64 arg3, uint64 arg4,
                        uint64 arg5)
```

```
0x80200040 <sbi_ecall+44> mv a5,a1
0x80200044 <sbi_ecall+48> sw a5,-56(s0)
B+> 0x80200048 <sbi_ecall+52> ld a0,-64(s0)
0x8020004c <sbi_ecall+56> ld a1,-72(s0)
0x80200050 <sbi_ecall+60> ld a2,-80(s0)
0x80200054 <sbi_ecall+64> ld a3,-88(s0)
0x80200058 <sbi_ecall+68> ld a4,-96(s0)
0x8020005c <sbi_ecall+72> ld a5,-104(s0)
0x80200060 <sbi_ecall+76> lw a6,-56(s0)
0x80200064 <sbi_ecall+80> lw a7,-52(s0)
0x80200068 <sbi_ecall+84> ecall
0x8020006c <sbi_ecall+88> mv a5,a0
0x80200070 <sbi_ecall+92> sd a5,-48(s0)
0x80200074 <sbi_ecall+96> mv a5,a1
0x80200078 <sbi_ecall+100> sd a5,-40(s0)
0x8020007c <sbi_ecall+104> ld a5,-48(s0)

remote Thread 1.1 In: sbi_ecall
(gdb) layout regs
(gdb) b sbi
sbi.c      sbi.h      sbi_ecall
(gdb) b sbi_ecall
Breakpoint 1 at 0x80200048: file sbi.c, line 14.
(gdb) c
Continuing.

Breakpoint 1, sbi_ecall (ext=1, fid=0, arg0=50, arg1=0, arg2=0, arg3=0, arg4=0, arg5=0) at sbi.c:14
(gdb) |
```

SBI Supervisor Binary Interface

·题外话

其实可以使用gdb打印mtvec的值，然后在那个位置打断点进入openSBI的固件
但如果按照实验中的命令做调试，在gdb中是看不到openSBI的源代码和符号表的，大家可以
想一想gdb需要什么条件才能使用C语言源代码来调试

```
0x80200040 <sbi_ecall+44> mv a5,a1
0x80200044 <sbi_ecall+48> sw a5,-56(s0)
B+> 0x80200048 <sbi_ecall+52> ld a0,-64(s0)
0x8020004c <sbi_ecall+56> ld a1,-72(s0)
0x80200050 <sbi_ecall+60> ld a2,-80(s0)
0x80200054 <sbi_ecall+64> ld a3,-88(s0)
0x80200058 <sbi_ecall+68> ld a4,-96(s0)
0x8020005c <sbi_ecall+72> ld a5,-104(s0)
0x80200060 <sbi_ecall+76> lw a6,-56(s0)
0x80200064 <sbi_ecall+80> lw a7,-52(s0)
0x80200068 <sbi_ecall+84> ecall
0x8020006c <sbi_ecall+88> mv a5,a0
0x80200070 <sbi_ecall+92> sd a5,-48(s0)
0x80200074 <sbi_ecall+96> mv a5,a1
0x80200078 <sbi_ecall+100> sd a5,-40(s0)
0x8020007c <sbi_ecall+104> ld a5,-48(s0)

remote Thread 1.1 In: sbi_ecall
(gdb) layout regs
(gdb) b sbi
sbi.c      sbi.h      sbi_ecall
(gdb) b sbi_ecall
Breakpoint 1 at 0x80200048: file sbi.c, line 14.
(gdb) c
Continuing.

Breakpoint 1, sbi_ecall (ext=1, fid=0, arg0=50, arg1=0, arg2=0, arg3=0, arg4=0, arg5=0) at sbi.c:14
(gdb) █
```

```
B+> 0x80000520 csrrw tp,mscratch,tp
0x80000524 sd t0,72(tp) # 0x48
0x80000528 csrr t0,mstatus
0x8000052c srli t0,t0,0xb
0x80000530 andi t0,t0,3
0x80000534 slti t0,t0,3
0x80000538 .2byte 0x12fd
0x8000053a xor sp,sp,tp
0x8000053e and t0,t0,sp
0x80000542 xor sp,sp,tp
0x80000546 xor t0,tp,t0
0x8000054a sd sp,-264(t0)
0x8000054e addi sp,t0,-280
0x80000552 ld t0,72(tp) # 0x48
0x80000556 .2byte 0xf416
0x80000558 csrrw tp,mscratch,tp

remote Thread 1.1 In:
(gdb) p $mtvec
$1 = 2147484960
(gdb) px $mtvec
Undefined command: "px". Try "help".
(gdb) i r $mtvec
mtvec      0x80000520      2147484960
(gdb) b * 0x80000520
Breakpoint 2 at 0x80000520
(gdb) si

Breakpoint 2, 0x0000000080000520 in ?? ()
```

汇编代码 *Assemble*

·GCC内嵌汇编

- 学习汇编与寄存器是如何工作的
- 但是实验指导的内容够用

·汇编调用C函数

- 要注意call trap_handler之前的寄存器数值如何赋值，才能在C的代码中读到正确的数值
- 关键词：RISCV call convention
- 但其实只要知道a0-a7是从前往后放参数就行（64位计算机yyds）

·C函数调用汇编

- lab3需要用到，建议提前了解

- 可视化编译：<https://godbolt.org/>

汇编代码 *Assemble*

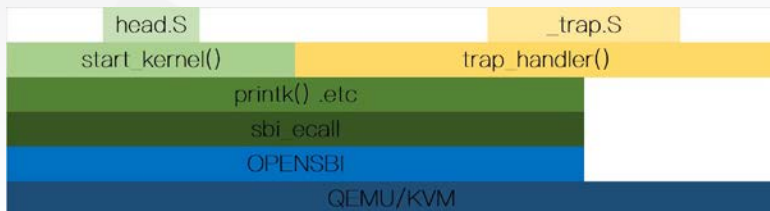
```
RISC-V (64-bits) gcc (trunk) (Editor #1) X
RISC-V (64-bits) gcc (trunk)  Compiler options...
A  Output...  Filter...  Libraries  Overrides  + Add new...  Add...
1  sbi_ecall(int, int, unsigned long, unsigned long, unsigned long, u
2      addi    sp,sp,-112
3      sd      ra,104(sp)
4      sd      s0,96(sp)
5      addi    s0,sp,112
6      sd      a2,-64(s0)
7      sd      a3,-72(s0)
8      sd      a4,-80(s0)
9      sd      a5,-88(s0)
10     sd      a6,-96(s0)
11     sd      a7,-104(s0)
12     mv      a5,a0
13     sw      a5,-52(s0)
14     mv      a5,a1
15     sw      a5,-56(s0)
16     ld      a0,-64(s0)
17     ld      a1,-72(s0)
18     ld      a2,-80(s0)
19     ld      a3,-88(s0)
20     ld      a4,-96(s0)
21     ld      a5,-104(s0)
22     lw      a6,-56(s0)
23     lw      a7,-52(s0)
24     ecall
25     mv      a5,a0
26     sd      a5,-48(s0)
27     mv      a5,a1
28     sd      a5,-40(s0)
29     ld      a5,-48(s0)
30     sd      a5,-32(s0)
31     ld      a5,-40(s0)
32     sd      a5,-24(s0)
33     ld      a4,-32(s0)
34     ld      a5,-24(s0)
35     mv      t1,a4
36     mv      t2,a5
37     mv      a4,t1
38     mv      a5,t2
39     mv      a0,a4
40     mv      a1,a5
41     ld      ra,104(sp)
42     ld      s0,96(sp)
43     addi    sp,sp,112
44     jr      ra
```

???



```
RISC-V (64-bits) gcc (trunk) (Editor #1) X
RISC-V (64-bits) gcc (trunk)  -O3
A  Output...  Filter...  Libraries  Overrides  + Add new...  Add...
1  ▾ sbi_ecall(int, int, unsigned long, unsigned long, unsigned long, u
2      mv      t1,a0
3      mv      t3,a1
4      addi    sp,sp,-16
5      mv      a0,a2
6      mv      a1,a3
7      mv      a2,a4
8      mv      a3,a5
9      mv      a4,a6
10     mv      a5,a7
11     mv      a6,t3
12     mv      a7,t1
13     ecall
14     addi    sp,sp,16
15     jr      ra
```

实验大纲



- 理解 CPU 上下文切换机制，并正确实现上下文切换功能；
- 编写 trap 处理函数，完成对特定 trap 的处理；
- 调用 OpenSBI 提供的接口，完成时钟中断/打印字符

```
lab1 > init > C main.c > ...
1  #include "printk.h"
2  #include "sbi.h"
3
4  extern void test();
5
6  int start_kernel() {
7      printk("2022");
8      printk(" Hello RISC-V\n");
9
10     test(); // DO NOT DELETE !!!
11
12     return 0;
13 }
14
15 void test() {
16     while (1){
17         int i = 1;
18         i = 2;
19         i = 3;
20     };
21 }
```

```
arch > riscv > kernel > ASM trap.S
.section .text.entry
.align 2
.globl _traps
_traps:
# -----
# 1. save 32 registers and sepc to stack
sd x0,0(sp)
# -----
# 2. call trap_handler
# -----
# 3. restore sepc and 32 registers (x2)
# -----
# 4. return from trap
ret
# -----
```

```
8 void trap_handler(uint64 scause,uint64 sepc){
9     printk("trap_handler\n");
10    sbi_ecall(1,1,4,5,1,4,0,0);
11    return ;
12 }
13
14 void sbi_ecall(int ext, int fid, uint64 arg0,
15                uint64 arg1, uint64 arg2,
16                uint64 arg3, uint64 arg4,
17                uint64 arg5);
```

课外指南

·多看文档，多找TA

Hot water plz

Linux社区很欢迎 I know its a stupid question

·压榨GPT

GPT is dead, so can I

ChatGPT本质上是个互联网养出来的克苏鲁

·抄代码，别抄作业

Same HASH???

抄代码能加快毕业，但抄作业能当场毙业

·写点函数库

#include <include/mini-stdlib.h>

你猜lib文件夹是用来干嘛的

推荐网站

- [RISCV SPEC](#)
- [可视化编译](#)
- [xv6中文文档](#), [xv6-riscv](#)
- [linux源代码在线查询](#)
- [FDU2023OSlab](#)
- [linux常用命令大全](#)

课外指南-多看文档

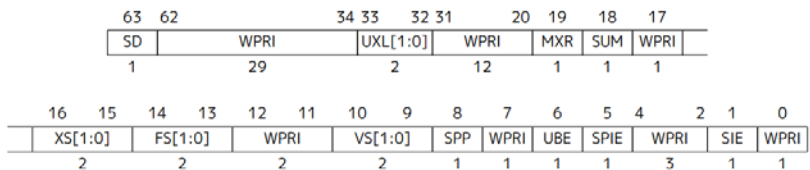


Figure 43. Supervisor-mode status register (**sstatus**) when SXLEN=64.

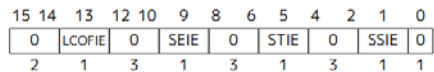


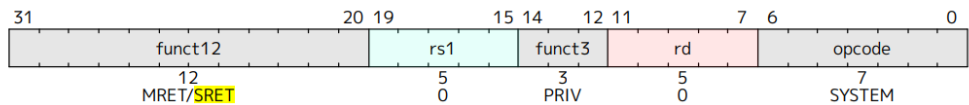
Figure 48. Standard portion (bits 15:0) of **sie**.

Bits **sip**.SEIP and **sie**.SEIE are the interrupt-pending and interrupt-enable bits for supervisor-level external interrupts. If implemented, SEIP is read-only in **sip**, and is set and cleared by the execution environment, typically through a platform-specific interrupt controller.

Bits **sip**.STIP and **sie**.STIE are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. If implemented, STIP is read-only in **sip**, and is set and cleared by the execution environment.

3.3.2. Trap-Return Instructions

Instructions to return from trap are encoded under the PRIV minor opcode.



To return after handling a trap, there are separate trap return instructions per privilege level, MRET and **SRET**. MRET is always provided. **SRET** must be provided if supervisor mode is supported, and should raise an illegal-instruction exception otherwise. **SRET** should also raise an illegal-instruction exception when TSR=1 in **mstatus**, as described in Section 3.1.6.5. An xRET instruction can be executed in privilege mode x or higher, where executing a lower-privilege xRET instruction will pop the relevant lower-privilege interrupt enable and privilege mode stack. In addition to manipulating the privilege stack as described in Section 3.1.6.1, xRET sets the **pc** to the value stored in the **xepc** register.

·RISCV privileged spec ([官网](#)最新为2024版， 2022版之后手册均可)

Lab1 Tips:

- sstatus.SIE/SPIE 中断使能
- sie.STIE 时钟中断使能
- stvec 中断向量
- stval 中断相关变量值
- scause 中断/异常来源号

课外指南-提问的艺术

Non-Canonical NaN Representation in Double-Precision Results from fmadd.d Instruction #2642

[New issue](#)

youzi27 opened this issue on Jan 15 · 8 comments



youzi27 commented on Jan 15

Describe the bug

When executing the `fmadd.d fa5, ft0, fa3, fa1, rtz` instruction in the XS, the resulting value in fa5 for a NaN outcome does not conform to the expected canonical NaN representation for double-precision floating-point numbers as specified in the RISC-V ISA. Instead of getting the canonical NaN value (0x7ff8000000000000), a different NaN value is observed.

To Reproduce

- Initialize the ft0 and fa3 registers with e.g., 0x00092afc56e7eae, and fa1 with e.g., 0xffffffff00000000.
- Execute the `fmadd.d fa5, ft0, fa3, fa1, rtz` instruction.
- Observe the value in the fa5 register.

Expected behavior

The expected result for a NaN outcome in double-precision floating-point operation, according to the IEEE 754 standard and RISC-V ISA, should be the canonical NaN value. For double-precision, this is 0x7ff8000000000000, which represents a NaN with a positive sign, all exponent bits set to 1, and the most significant bit of the significand set to 1 (quiet bit), with all other significand bits clear.

Screenshots

```
commit pc 00000000000000f58 inst 5ad017c3 wen 1 dst 15 data ffffffff idix 013 fmadd.d fa5, ft0, fa3, fa1 <--
different at fa5 freqd 90%, spike= 0x7ff8000000000000, xs = 0xffffffffff
```

Environment (optional, if necessary):

- OS: Ubuntu 22.04.3 LTS
- Compiler: gcc 11.4.0

Additional context

None



2

Assignees

huxuan0307

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

No branches or pull requests

Notifications

[Customize](#)

Subscribe

You're not receiving notifications from this thread.

5 participants



cebarobot assigned cebarobot and huxuan0307 and unassigned cebarobot on Jan 15

课外指南-压榨GPT

ChatGPT ▾



```
"qemu-system-riscv64 -nographic -machine virt -kernel  
path/to/linux/arch/riscv/boot/Image \  
-device virtio-blk-device,drive=hd0 -append "root=/dev/vda  
ro console=ttyS0" \  
-bios default -drive  
file=path/to/rootfs.img,format=raw,id=hd0"  
these are the command in my os lecture, plz explain the  
meaning to me, I'd appreciated if you could offer a different  
example command of qemu-system-riscv64 to me
```



Memory updated

Let's break down the QEMU command you provided step-by-step:

1. `qemu-system-riscv64`: This invokes the QEMU emulator for the RISC-V 64-bit architecture.
2. `-nographic`: Disables the graphical output, meaning only the serial console will be used (standard input/output is redirected to the terminal).
3. `-machine virt`: Specifies the type of machine QEMU will emulate. Here, `virt` refers to a generic virtual platform provided by QEMU for RISC-V.
4. `-kernel path/to/linux/arch/riscv/boot/Image`: Points to the kernel binary that you want QEMU to boot from. Replace `path/to/linux/arch/riscv/boot/Image` with the actual path of your compiled Linux kernel for RISC-V.



Message ChatGPT




ChatGPT can make mistakes. Check important info.

课外指南-错误示范

匿名D1CF84

别问我是谁



以下是引用21楼: 用户在2023-12-25 10:33:07的发言: >>查看原帖<<
如果是班判 0 的话, 本周二下午实验课可以去解释一下情况。

感觉老师也不太听, 所有人被抄, 不管啥都是扣一半🙄

发表于 2023-12-26 17:20:50

0

0

评分

引用

追踪

楼主

28

匿名D1CF84

别问我是谁



结果出来了, 先详细说一下背景, 之前也说了就那七行是几乎一模一样的, 也包括变量名(怪我教的太手把手了, 我寻思要是真抄谁没个心眼换一下啊) 然后那个文件一共三个函数, 另外一个函数里有一个变量名一样, 这个我之前都没注意过...思路的话确实也基本一样但是那就是上课讲的思路过程, 我看了看别人的思路也基本都一样。
另外可能之前没有说清楚, 挂0是单次lab, 不是整门课, 今天和老师助教解释了半天感觉也没啥用, 被“抄”的基本

都一视同仁扣一半, 我也喜提单次lab50%



唉,总评喜提-7.5...
谢谢大家关心了, 祝大家生活愉快, 努力不被辜负!

发表于 2023-12-26 17:27:55

2

2

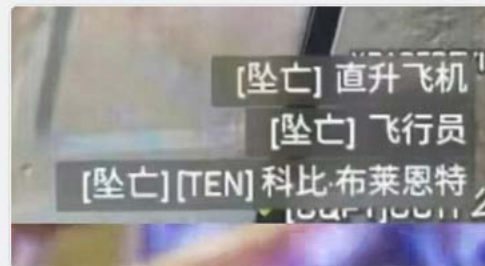
评分

引用

楼主

29

彩蛋：没有注释的warning必将汇聚成bug



已读

猜猜今天我花了多长时间帮人清理 🐛 山



已读

无奖竞猜



已读

遇上的问题有，没保存栈用完t0不恢复，不用csrr非得用csrrw结果把x0写进sstatus，栈指针腾了280结果往280(sp)写东西



已读

我真纳闷这哥们怎么通过前面的lab的



已读