第4季

RISC-V异常处理

本节课主要内容

- 本章主要内容
  - 异常处理

技术手册：

1. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203
2. SiFive U74-MC Core Complex Manual, 21G2.01.00

本节课主要讲解书上第8章内容

异常类型

- 什么是异常？为什么需要异常？
- 异常可以理解为：**处理器硬件 主动请求 与软件交互的一种接口**
- 异常类型
  - ✓ 异常
  - ✓ 中断
  - ✓ 系统调用
- 同步异常和异步异常

异常入口

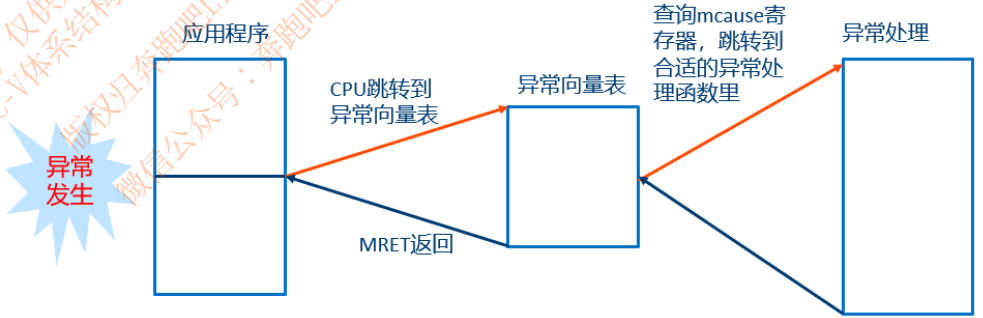
- 所有的异常（包括中断）都在M模式下处理
- CPU自动做如下事情：
  ① PC值 -> mepc寄存器
  ② 异常的类型 -> mcause寄存器。
  ③ 异常虚拟地址 -> mtval寄存器。
  ④ MIE字段 -> MPIE字段。
  ⑤ 处理器模式 -> MPP字段。
  ⑥ MIE字段 -> 0
  ⑦ 设置处理器模式为M模式。
  ⑧ 跳转到异常向量表里执行，即PC -> mtvec寄存器的值

- 操作系统需要做的事情：
  ① 保存异常发生时的上下文，所有通用寄存器以及部分M模式的系统寄存器
  ② 查询mcause寄存器中的异常以及中断编号，跳转到合适的异常处理程序中。

异常返回

- 操作系统需要做的事情：
  ① 异常处理完成，恢复保存在栈里的上下文。
  ② 执行mret指令

- 执行mret后，CPU自动做如下实现：
  ① MPIE -> MIE
  ② 从MPP中恢复处理器模式
  ③ mpec的值 写入到 PC



异常返回地址

- ra寄存器保存了函数返回地址
- 发生异常时的PC值，CPU会自动保存到mepc寄存器里。异常返回时，CPU会把mepc/sepc寄存器的值恢复到PC寄存器中

- 异常返回地址是指向发生异常时的指令还是下一条指令呢？
  - ✓ 对于中断，它的返回地址是第一条还没执行或由于中断没有成功执行的指令。
  - ✓ 对于不是系统调用的同步异常，比如数据异常、访问了没有映射的地址等等，那么它返回的是触发同步异常的那条指令。
  - ✓ 系统调用返回的是系统调用指令（例如ECALL指令）的下一条指令。

异常返回的处理器模式

- 异常返回要不要切换处理模式看mstatus寄存器中MPP字段：
  ① MPP 为0时，表示触发异常时CPU正运行在U模式（用户模式），那么异常处理结束后会返回到U模式。

  ② MPP 为1时，表示触发异常时CPU正运行在S模式（特权模式），那么异常处理结束后会返回到S模式

  ③ MPP为2时，表示是在M模式触发的异常，异常之后还是返回M模式

栈的选择

- 有些处理器架构，每个处理器模式都有一个专用的SP寄存器，如Armv8
  - ✓ EL0 -> SP_EL0
  - ✓ EL1 -> SP_EL1
  - ✓ EL2 -> SP_EL2
  - ✓ EL3 -> SP_EL3

- RISC-V处理器，所有的处理器模式只有一个SP寄存器。
  - ✓ 当跳转到另外一个处理器模式时，SP寄存器还是指向上一个处理器模式的栈地址
  - ✓ 软件需要把上一个处理器模式的SP指针保存起来，然后再设置当前处理器的栈到SP

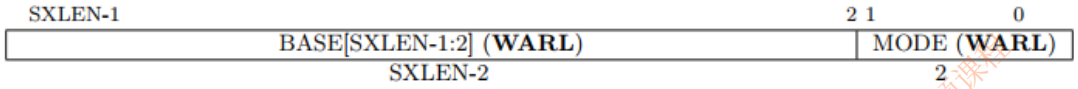
与M模式相关的异常寄存器

➢ 与M模式相关的异常寄存器有：mstatus、mtvec、mie、mip以及mcause寄存器。

表 8.1 mstatus 寄存器中与异常/中断相关的字段

字段	位	说明
SIE	Bit[1]	使能S模式下的中断
MIE	Bit[3]	使能M模式下的中断
SPIE	Bit[5]	临时保存的中断使能状态（S模式下）
MPIE	Bit[7]	临时保存的中断使能状态（M模式下）
SPP	Bit[8]	中断之前的特权模式（发生在S模式下的中断）
MPP	Bit[12:11]	中断之前的特权模式（发生在M模式下的中断）

异常向量寄存器mvtec



MODE字段：用来设置向量模式。

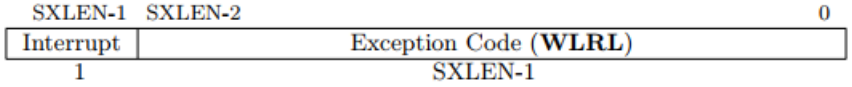
- ✓ 0：表示直接访问模式。
- ✓ 1：表示向量访问模式。

BASE字段：异常向量表的基地址

- 直接访问模式：异常向量基地址4字节对齐
  - ① 先跳转到BASE字段设置的基地址中。
  - ② 读取mcause寄存器来查询异常或者中断触发的原因
  - ③ 再跳转到对应的异常（中断）处理函数中。

- 向量访问模式：异常向量基地址必须256个字节对齐
  - ① 每个向量占4个字节，即“BASE + 4 × exception code”

mcause和mtval寄存器



- Interrupt字段：为1时表示触发的异常类型为中断类型，否则为同步异常类型
- EC字段：异常编码

- mtval（mbadaddr）寄存器记录了记录了发生异常的虚拟地址

表 8.2 mcause 寄存器

Interrupt 字段	EC 字段	说明
1	0	保留
1	1	S模式下的软件中断（software interrupt）
1	2	保留
1	3	M模式下的软件中断
1	5	S模式下时钟中断
1	6	保留
1	7	M模式下的时钟中断
1	8	保留
1	9	S模式下的外部中断
1	10	保留
1	11	M模式下的外部中断
1	12-13	保留
1	>=16	预留给芯片设计使用
0	0	指令地址没对齐（instruction address misaligned）
0	1	指令访问异常（instruction access fault）
0	2	非法指令（illegal instruction）
0	3	断点（breakpoint）
0	4	加载地址没对齐（load address misaligned）
0	5	加载访问异常（load access fault）
0	6	存储/AMO地址没对齐（store/AMO address misaligned）
0	7	存储/AMO访问异常（store/AMO access fault）

详见RISC-V架构手册Table 3.6

mie/mip寄存器

表 8.3 mie 寄存器

字段	位	说明
SSIE	Bit[1]	使能S模式下的软件中断
MSIE	Bit[3]	使能M模式下的软件中断
STIE	Bit[5]	使能S模式下的时钟中断
MTIE	Bit[7]	使能M模式下的时钟中断
SEIE	Bit[9]	使能S模式下外部中断
MEIE	Bit[11]	使能M模式下的外部中断

表 8.4 mip 寄存器

字段	位	说明
SSIP	Bit[1]	S模式下的软件中断处于等待响应状态
MSIP	Bit[3]	M模式下的软件中断处于等待响应状态
STIP	Bit[5]	S模式下的时钟中断处于等待响应状态
MTIP	Bit[7]	M模式下的时钟中断处于等待响应状态
SEIP	Bit[9]	S模式下外部中断处于等待响应状态

委托寄存器mideleg和medeleg

表 8.5 mideleg 寄存器

字段	位	说明
SSIP	Bit[1]	把软件中断委托给S模式
STIP	Bit[5]	把时钟中断委托给S模式
SEIP	Bit[9]	把外部中断委托给S模式

表 8.6 medeleg 寄存器

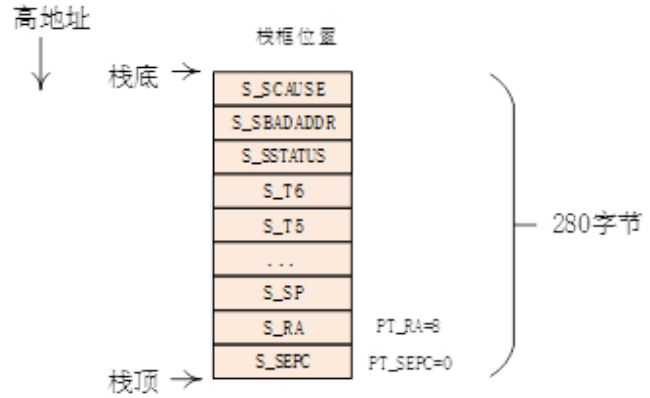
位	说明
Bit[0]	把未对齐的指令访问异常委托给S模式
Bit[1]	把指令访问异常委托给S模式
Bit[2]	把无效指令异常委托给S模式
Bit[3]	把断点异常委托给S模式
Bit[4]	把未对齐加载访问异常委托给S模式
Bit[5]	把加载访问异常委托给S模式
Bit[6]	把未对齐存储/AMO访问异常委托给S模式
Bit[7]	把存储/AMO访问异常委托给S模式
Bit[8]	把来自用户模式的系统调用处理委托给S模式
Bit[9]	把来自管理员特权模式的系统调用处理委托给S模式
Bit[12]	把指令缺页异常委托给S模式
Bit[13]	把加载缺页异常委托给S模式
Bit[15]	把存储/AMO缺页异常委托给S模式

异常上下文

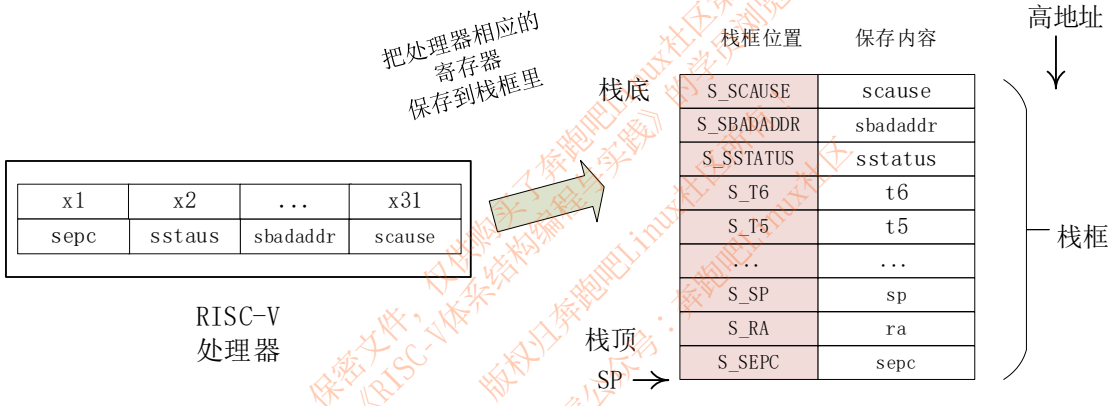
- 在异常发生时需要保存发生异常的现场，以免破坏了异常发生前正在处理的数据和程序状态
- 以发生在S模式的异常为例，我们需要保存如下内容到栈空间里。
  - ✓ x1 ~ x31 通用寄存器
  - ✓ spec寄存器
  - ✓ sstatus寄存器
  - ✓ sbadaddr/stval寄存器
  - ✓ scause寄存器
- 这个栈空间指的是发生异常时进程的内核态的栈空间。

pt_regs栈框

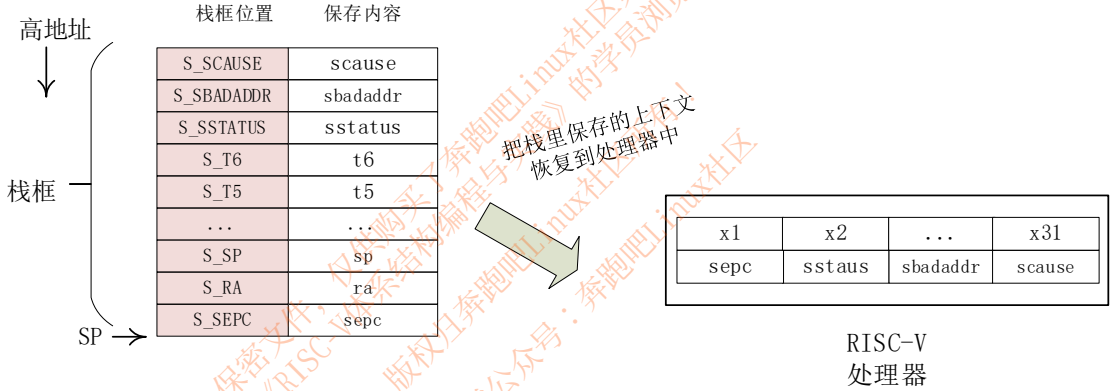
```
struct pt_regs {
    /* 31个通用寄存器 + sepc + sstatus */
    unsigned long sepc;
    unsigned long ra;
    unsigned long sp;
    unsigned long gp;
    unsigned long tp;
    unsigned long t0;
    unsigned long t1;
    unsigned long t2;
    unsigned long s0;
    unsigned long s1;
    unsigned long a0;
    unsigned long a1;
    unsigned long a2;
    unsigned long a3;
    unsigned long a4;
    unsigned long a5;
    unsigned long a6;
    unsigned long a7;
    unsigned long s2;
    unsigned long s3;
    unsigned long s4;
    unsigned long s5;
    unsigned long s6;
    unsigned long s7;
    unsigned long s8;
    unsigned long s9;
    unsigned long s10;
    unsigned long s11;
    unsigned long t3;
    unsigned long t4;
    unsigned long t5;
    unsigned long t6;
    /* S模式下的寄存器 */
    unsigned long sstatus;
    unsigned long sbadaddr;
    unsigned long scause;
};
```

保存异常上下文



恢复异常上下文



案例分析1：实现SBI系统调用

- 要求：运行在S模式下的BenOS可以通过ECALL指令来陷入到M模式的MySBI固件，然后在MySBI固件中实现串口打印功能
- ECALL指令：RISC-V提供的系统调用指令，U -> S, S -> M

<benos/include/asm/sbi.h>

```
1    #define SBI_CALL(which, arg0, arg1, arg2) ({           \
2        register unsigned long a0 asm ("a0") = (unsigned long)(arg0); \
3        register unsigned long a1 asm ("a1") = (unsigned long)(arg1); \
4        register unsigned long a2 asm ("a2") = (unsigned long)(arg2); \
5        register unsigned long a7 asm ("a7") = (unsigned long)(which);\
6        asm volatile ("ecall"                  \
7                : "+r" (a0)                \
8                : "r" (a1), "r" (a2), "r" (a7)       \
9                : "memory");               \
10       a0;                        \
11   })
```

<benos/include/asm/sbi.h>

```
/*
 * 陷入到M模式，调用M模式提供的服务。
 */
#define SBI_CALL_0(which) SBI_CALL(which, 0, 0, 0)
#define SBI_CALL_1(which, arg0) SBI_CALL(which, arg0, 0, 0)
#define SBI_CALL_2(which, arg0, arg1) SBI_CALL(which, arg0, arg1, 0)
```

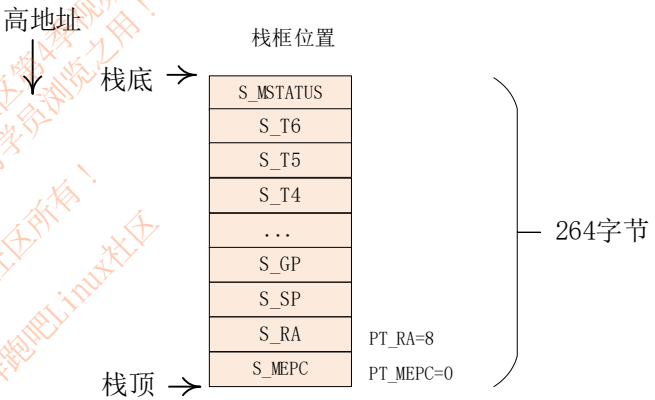
which参数用于SBI扩展ID（SBI extension ID, EID）
arg0是要传递的第一个参数
arg1是要传递的第二个参数
arg2是要传递的第三个参数

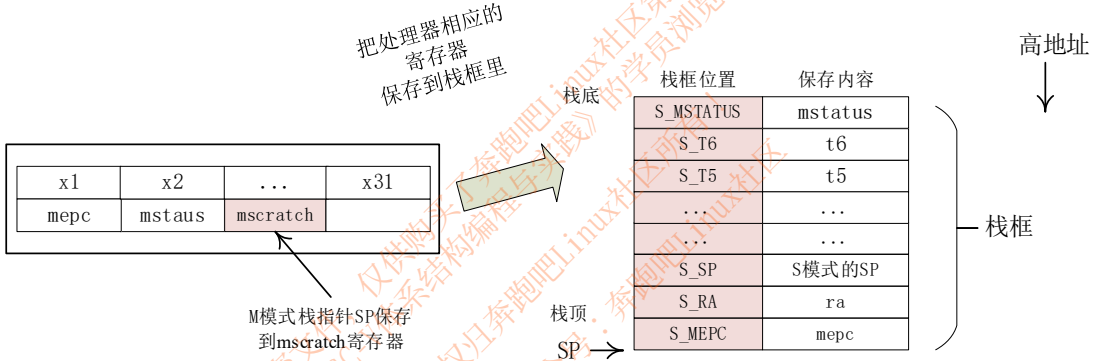
<benos/include/asm/sbi.h>

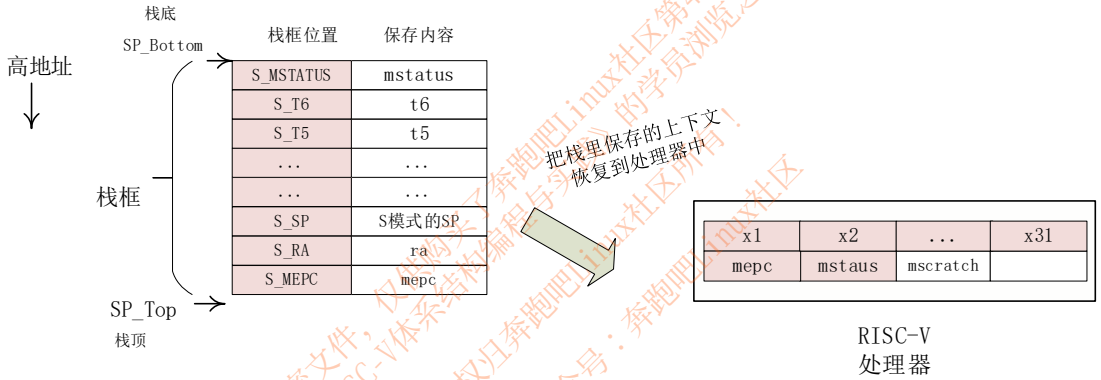
```
#define SBI_CONSOLE_PUTCHAR 0x1
#define SBI_CONSOLE_GETCHAR 0x2

static inline void sbi_putchar(unsigned char c)
{
    SBI_CALL_1(SBI_CONSOLE_PUTCHAR, c);
}

static inline void sbi_put_string(char *str)
{
    int i;

    for (i = 0; str[i] != '\0'; i++)
        sbi_putchar((char) str[i]);
}
```

SBI栈处理

```
<benos/sbi/sbi_trap_regs.h>
struct sbi_trap_regs {
    /* sepc + 31个通用寄存器 */
    unsigned long mepc;
    unsigned long ra;
    unsigned long sp;
    unsigned long gp;
    unsigned long tp;
    unsigned long t0;
    unsigned long t1;
    unsigned long t2;
    unsigned long s0;
    unsigned long s1;
    unsigned long a0;
    unsigned long a1;
    unsigned long a2;
    unsigned long a3;
    unsigned long a4;
    unsigned long a5;
    unsigned long a6;
    unsigned long a7;
    unsigned long s2;
    unsigned long s3;
    unsigned long s4;
    unsigned long s5;
    unsigned long s6;
    unsigned long s7;
    unsigned long s8;
    unsigned long s9;
    unsigned long s10;
    unsigned long s11;
    unsigned long t3;
    unsigned long t4;
    unsigned long t5;
    unsigned long t6;
    /* mstatus寄存器 */
    unsigned long mstatus;
};
```







案例分析2：BenOS的异常处理

- 要求：在S模式下的BenOS里制造一个加载访问异常，然后在异常处理中输出：异常类型、出错地址等日志信息。

```
do_exception, scause:0x5
Oops - Load access fault
sepc: 00000000802019ec ra : 00000000802018fc sp : 0000000080203ff0
 gp : 0000000000000000 tp : 0000000000000000 t0 : 0000000000000005
 t1 : 0000000000000005 t2 : 0000000080200020 s0 : 0000000080017f20
 s1 : 0000000080200010 a0 : 0000000070000000 a1 : 000000000000000a
 a2 : 0000000000000006 a3 : 0000000080203ef0 a4 : 0000000000000031
 a5 : 0000000000000031 a6 : 0000000000000002 a7 : 0000000000000061
 s2 : 8000000000006800 s3 : 0000000080200000 s4 : 0000000082200000
 s5 : 0000000000000000 s6 : 0000000000000000 s7 : 00000000800120e8
 s8 : 000000008020002e s9 : 000000000000007f s10: 0000000000000000
 s11: 0000000000000000 t3 : 0990106f91166285 t4: 0000000080017ee0
 t5 : 0000000000000027 t6 : 0000000000000000
sstatus:0x8000000000006100 sbadaddr:0x0000000070000000
scause:0x0000000000000005
Kernel panic
```

实验1：在SBI中实现串口输入功能

1．实验目的

加深对异常处理流程的理解。

2．实验要求

在MySBI固件中实现SBI_CONSOLE_GETCHAR的服务接口并测试。

实验2：在BenOS中触发非法指令异常

1．实验目的

加深对异常处理流程的理解。

2．实验要求

在BenOS中触发一个非法指令异常。

提示：触发非法指令异常可以有如下两种方式。

在S模式下访问M模式下的寄存器，如mstatus寄存器。

通过篡改代码段里的指令代码触发一个非法指令访问异常。

例如，下面的代码把trigger_load_access_fault()汇编函数的第1行代码篡改了。

```
void create_illegal_intr(void)
{
    int *p = (int *)trigger_load_access_fault;

    *p = 0xbadbeef;
}
```

```
do_exception, scause:0x2
Oops - Illegal instruction
sepc: 00000000802dc2a2 ra : 0000000080201078 sp : 0000000080203ff0
 gp : 0000000000000000 tp : 0000000000000000 t0 : 0000000000000005
 t1 : 0000000000000005 t2 : 0000000080200020 t3 : 0000000000000000
 s1 : 0000000080200010 a0 : 0000000000000031 a1 : 0000000000000000
 a2 : 0000000000000000 a3 : 000000000802008a0 a4 : 000000000badbeef
 a5 : 0000000802011e8 a6 : 0000000000000000 a7 : 0000000000000001
 s2 : 0000000000000000 s3 : 0000000000000000 s4 : 0000000000000000
 s5 : 0000000000000000 s6 : 0000000000000000 s7 : 0000000000000000
 s8 : 0000000080200034 s9 : 0000000000000000 s10: 0000000000000000
 s11: 0000000000000000 t3 : 00510133000012b7 t4: 0000000802011ec
 t5 : 0000000000000000 t6 : 0000000000000000
sstatus:0x0000000000000100  sbadaddr:0x0000000000000000  scause:0x0000000000000002
Kernel panic
```

实验3：输出触发异常时函数栈的调用过程

1．实验目的

加深对异常处理流程的理解。

2．实验要求

在BenOS中触发一个异常之后，输出函数栈的调用过程（calltrace）

```
do_exception, scause:0x5
Oops - Load access fault
Call Trace:
[<0x0000000080201da8>] trigger_load_access_fault+0x4/0xc
[<0x0000000080201b80>] test_fault+0x10/0x28
[<0x0000000080201c0c>] kernel_main+0x74/0xa4
sepc: 0000000080201da8 ra : 0000000080201b5c sp : 0000000080205fc0
 gp : 0000000080206800 tp : 0000000000000000 t0 : 0000000000000005
 t1 : 0000000000000005 t2 : 0000000080200020 t3 : 0000000080205fd0
 s1 : 0000000080200010 a0 : 0000000070000000 a1 : 0000000000000000
 a2 : 0000000000000000 a3 : 0000000080201214 a4 : 0000000000000005
 a5 : 000000000000000f a6 : 0000000000000000 a7 : 0000000000000001
 s2 : 0000000000000000 s3 : 0000000000000000 s4 : 0000000000000000
 s5 : 0000000000000000 s6 : 0000000000000000 s7 : 0000000000000000
 s8 : 000000008020003c s9 : 0000000000000000 s10: 0000000000000000
 s11: 0000000000000000 t3 : 00510133000012b7 t4: 0000000000000000
 t5 : 0000000000000000 t6 : 0000000000000000
sstatus:0x0000000000000100  sbadaddr:0x0000000070000000  scause:0x0000000000000005
Kernel panic
```