kendryte K210 startup_xxx.S文件分析

源码

```
//Part 1
#include "cpuport.h"

  .section      .text.entry
  .align 2                    //四个字节对齐
  .global trap_entry          //符号对链接器可见，可以供其他链接对象模块使用

//Part 2
trap_entry:
#ifdef ARCH_RISCV_FPU
    addi    sp, sp, -32 * FREGBYTES

    FSTORE  f0, 0 * FREGBYTES(sp)
    FSTORE  f1, 1 * FREGBYTES(sp)
    FSTORE  f2, 2 * FREGBYTES(sp)
    FSTORE  f3, 3 * FREGBYTES(sp)
    FSTORE  f4, 4 * FREGBYTES(sp)
    FSTORE  f5, 5 * FREGBYTES(sp)
    FSTORE  f6, 6 * FREGBYTES(sp)
    FSTORE  f7, 7 * FREGBYTES(sp)
    FSTORE  f8, 8 * FREGBYTES(sp)
    FSTORE  f9, 9 * FREGBYTES(sp)
    FSTORE  f10, 10 * FREGBYTES(sp)
    FSTORE  f11, 11 * FREGBYTES(sp)
    FSTORE  f12, 12 * FREGBYTES(sp)
    FSTORE  f13, 13 * FREGBYTES(sp)
    FSTORE  f14, 14 * FREGBYTES(sp)
    FSTORE  f15, 15 * FREGBYTES(sp)
    FSTORE  f16, 16 * FREGBYTES(sp)
    FSTORE  f17, 17 * FREGBYTES(sp)
    FSTORE  f18, 18 * FREGBYTES(sp)
    FSTORE  f19, 19 * FREGBYTES(sp)
    FSTORE  f20, 20 * FREGBYTES(sp)
    FSTORE  f21, 21 * FREGBYTES(sp)
    FSTORE  f22, 22 * FREGBYTES(sp)
    FSTORE  f23, 23 * FREGBYTES(sp)
    FSTORE  f24, 24 * FREGBYTES(sp)
    FSTORE  f25, 25 * FREGBYTES(sp)
    FSTORE  f26, 26 * FREGBYTES(sp)
    FSTORE  f27, 27 * FREGBYTES(sp)
    FSTORE  f28, 28 * FREGBYTES(sp)
    FSTORE  f29, 29 * FREGBYTES(sp)
    FSTORE  f30, 30 * FREGBYTES(sp)
    FSTORE  f31, 31 * FREGBYTES(sp)

#endif

    /* save thread context to thread stack */
```

```
    addi sp, sp, -32 * REGBYTES

    STORE x1,   1 * REGBYTES(sp)

    csrr  x1, mstatus
    STORE x1,   2 * REGBYTES(sp)

    csrr  x1, mepc
    STORE x1,   0 * REGBYTES(sp)

    STORE x4,   4 * REGBYTES(sp)
    STORE x5,   5 * REGBYTES(sp)
    STORE x6,   6 * REGBYTES(sp)
    STORE x7,   7 * REGBYTES(sp)
    STORE x8,   8 * REGBYTES(sp)
    STORE x9,   9 * REGBYTES(sp)
    STORE x10, 10 * REGBYTES(sp)
    STORE x11, 11 * REGBYTES(sp)
    STORE x12, 12 * REGBYTES(sp)
    STORE x13, 13 * REGBYTES(sp)
    STORE x14, 14 * REGBYTES(sp)
    STORE x15, 15 * REGBYTES(sp)
    STORE x16, 16 * REGBYTES(sp)
    STORE x17, 17 * REGBYTES(sp)
    STORE x18, 18 * REGBYTES(sp)
    STORE x19, 19 * REGBYTES(sp)
    STORE x20, 20 * REGBYTES(sp)
    STORE x21, 21 * REGBYTES(sp)
    STORE x22, 22 * REGBYTES(sp)
    STORE x23, 23 * REGBYTES(sp)
    STORE x24, 24 * REGBYTES(sp)
    STORE x25, 25 * REGBYTES(sp)
    STORE x26, 26 * REGBYTES(sp)
    STORE x27, 27 * REGBYTES(sp)
    STORE x28, 28 * REGBYTES(sp)
    STORE x29, 29 * REGBYTES(sp)
    STORE x30, 30 * REGBYTES(sp)
    STORE x31, 31 * REGBYTES(sp)

//Part 3
    /* switch to interrupt stack */
    move  s0, sp

    /* get cpu id */
    csrr  t0, mhartid

    /* switch interrupt stack of current cpu */
    la    sp, __stack_start__
    addi  t1, t0, 1
    li    t2, __STACKSIZE__
    mul   t1, t1, t2
    add   sp, sp, t1 /* sp = (cpuid + 1) * __STACKSIZE__ + __stack_start__ */

    /* handle interrupt */
    call  rt_interrupt_enter
```

```
    csrr  a0, mcause
    csrr  a1, mepc
    mv    a2, s0
    call  handle_trap
    call  rt_interrupt_leave

//Part 4
#ifdef RT_USING_SMP
    /* s0 --> sp */
    mv  sp, s0
    mv  a0, s0
    call rt_scheduler_do_irq_switch
    tail   rt_hw_context_switch_exit

#else

    /* switch to from_thread stack */
    move  sp, s0

    /* need to switch new thread */
    la    s0, rt_thread_switch_interrupt_flag
    lw    s2, 0(s0)
    beqz  s2, spurious_interrupt
    sw    zero, 0(s0)

    la    s0, rt_interrupt_from_thread
    LOAD  s1, 0(s0)
    STORE sp, 0(s1)

    la    s0, rt_interrupt_to_thread
    LOAD  s1, 0(s0)
    LOAD  sp, 0(s1)

#endif

spurious_interrupt:
    tail rt_hw_context_switch_exit
```

下面是cpuport.h源码

```
#ifndef CPUPORT_H__
#define CPUPORT_H__

#include <rtconfig.h>

/* bytes of register width  */
#ifdef ARCH_CPU_64BIT
#define STORE                   sd
#define LOAD                    ld
#define REGBYTES                8
#else
#define STORE                   sw
#define LOAD                    lw
#define REGBYTES                4
```

```
#endif

#ifdef ARCH_RISCV_FPU
#ifdef ARCH_RISCV_FPU_D
#define FSTORE                  fsd
#define FLOAD                   fld
#define FREGBYTES               8
#define rv_floatreg_t           rt_int64_t
#endif
#ifdef ARCH_RISCV_FPU_S
#define FSTORE                  fsw
#define FLOAD                   flw
#define FREGBYTES               4
#define rv_floatreg_t           rt_int32_t
#endif
#endif

#endif
```

我们现在开始逐行分析整个代码。

Part1

```
//Part 1
#include "cpuport.h"

  .section      .text.entry
  .align 2                    //四个字节对齐
  .global trap_entry          //符号对链接器可见，可以供其他链接对象模块使用
```

四字节对齐

Part2

```
//Part 2
trap_entry:
#ifdef ARCH_RISCV_FPU
    addi    sp, sp, -32 * FREGBYTES

    FSTORE  f0, 0 * FREGBYTES(sp)
    FSTORE  f1, 1 * FREGBYTES(sp)
    FSTORE  f2, 2 * FREGBYTES(sp)
    FSTORE  f3, 3 * FREGBYTES(sp)
    FSTORE  f4, 4 * FREGBYTES(sp)
    FSTORE  f5, 5 * FREGBYTES(sp)
    FSTORE  f6, 6 * FREGBYTES(sp)
    FSTORE  f7, 7 * FREGBYTES(sp)
    FSTORE  f8, 8 * FREGBYTES(sp)
    FSTORE  f9, 9 * FREGBYTES(sp)
    FSTORE  f10, 10 * FREGBYTES(sp)
    FSTORE  f11, 11 * FREGBYTES(sp)
    FSTORE  f12, 12 * FREGBYTES(sp)
    FSTORE  f13, 13 * FREGBYTES(sp)
    FSTORE  f14, 14 * FREGBYTES(sp)
```

```
    FSTORE  f15, 15 * FREGBYTES(sp)
    FSTORE  f16, 16 * FREGBYTES(sp)
    FSTORE  f17, 17 * FREGBYTES(sp)
    FSTORE  f18, 18 * FREGBYTES(sp)
    FSTORE  f19, 19 * FREGBYTES(sp)
    FSTORE  f20, 20 * FREGBYTES(sp)
    FSTORE  f21, 21 * FREGBYTES(sp)
    FSTORE  f22, 22 * FREGBYTES(sp)
    FSTORE  f23, 23 * FREGBYTES(sp)
    FSTORE  f24, 24 * FREGBYTES(sp)
    FSTORE  f25, 25 * FREGBYTES(sp)
    FSTORE  f26, 26 * FREGBYTES(sp)
    FSTORE  f27, 27 * FREGBYTES(sp)
    FSTORE  f28, 28 * FREGBYTES(sp)
    FSTORE  f29, 29 * FREGBYTES(sp)
    FSTORE  f30, 30 * FREGBYTES(sp)
    FSTORE  f31, 31 * FREGBYTES(sp)

#endif

    /* save thread context to thread stack */
    addi sp, sp, -32 * REGBYTES

    STORE x1,   1 * REGBYTES(sp)

    csrr  x1, mstatus
    STORE x1,   2 * REGBYTES(sp)

    csrr  x1, mepc
    STORE x1,   0 * REGBYTES(sp)

    STORE x4,   4 * REGBYTES(sp)
    STORE x5,   5 * REGBYTES(sp)
    STORE x6,   6 * REGBYTES(sp)
    STORE x7,   7 * REGBYTES(sp)
    STORE x8,   8 * REGBYTES(sp)
    STORE x9,   9 * REGBYTES(sp)
    STORE x10, 10 * REGBYTES(sp)
    STORE x11, 11 * REGBYTES(sp)
    STORE x12, 12 * REGBYTES(sp)
    STORE x13, 13 * REGBYTES(sp)
    STORE x14, 14 * REGBYTES(sp)
    STORE x15, 15 * REGBYTES(sp)
    STORE x16, 16 * REGBYTES(sp)
    STORE x17, 17 * REGBYTES(sp)
    STORE x18, 18 * REGBYTES(sp)
    STORE x19, 19 * REGBYTES(sp)
    STORE x20, 20 * REGBYTES(sp)
    STORE x21, 21 * REGBYTES(sp)
    STORE x22, 22 * REGBYTES(sp)
    STORE x23, 23 * REGBYTES(sp)
    STORE x24, 24 * REGBYTES(sp)
    STORE x25, 25 * REGBYTES(sp)
    STORE x26, 26 * REGBYTES(sp)
    STORE x27, 27 * REGBYTES(sp)
```

```
    STORE x28, 28 * REGBYTES(sp)
    STORE x29, 29 * REGBYTES(sp)
    STORE x30, 30 * REGBYTES(sp)
    STORE x31, 31 * REGBYTES(sp)
```

存储浮点数：

调整线程栈指针`sp`，预留出32个FREGBYTES的空间，用于保存浮点数寄存器。REGBYTES是4字节，即IEEE754标准浮点数，即32位。整个栈空间为32*4字节。

存储整数：

调整线程栈指针`sp`，预留出32个REGBYTES的空间，用于保存浮点数寄存器。REGBYTES是8字节，即64位。整个栈空间为32*8字节。

这里存储了`x1，x4~x31`整数寄存器，`x0`硬件接为0，不需要存储，`x2`是`sp`，`x3`是`gp`，都不需要存储。

Part3

```
//Part 3
    /* switch to interrupt stack */
    move  s0, sp

    /* get cpu id */
    csrr  t0, mhartid

    /* switch interrupt stack of current cpu */
    la    sp, __stack_start__
    addi  t1, t0, 1
    li    t2, __STACKSIZE__
    mul   t1, t1, t2
    add   sp, sp, t1 /* sp = (cpuid + 1) * __STACKSIZE__ + __stack_start__ */

    /* handle interrupt */
    call  rt_interrupt_enter
    csrr  a0, mcause
    csrr  a1, mepc
    mv    a2, s0
    call  handle_trap
    call  rt_interrupt_leave
```

现在开始处理异常或者中断了，这时先把线程栈的栈顶指针暂存到`s0`中，等异常或中断处理完成后再恢复。

首先获取cpu id号码，然后计算中断栈的栈顶指针。

然后，使用`rt_interrupt_enter`和`rt_interrupt_leave`成对记录中断嵌套计数。此外，将设置参数`mcause`，`mepc`和`s0`，并调用中断处理函数`handle_trap`。

- mcause，即machine cause register寄存器。当一个trap发生，陷入M-mode时，mcause将记录发生trap的原因。下图列出了machine-level的异常编码。异常代码是一个 WLRL 字段，因此只能保证包含支持的异常代码。`mcause`用来表明发生了何种trap，其最高位为1表示中断，最高位是0表示异常，低63位为不同中断类型或异常类型的编码。
  - 软件中断`IRQ_M_SOFT`：`IRQ_M_SOFT`的值是3，由上表可知是machine software interrupt，用

来触发软件中断进行调度。
- 外部中断`IRQ_M_EXT`：来自中断管理平台PLIC的中断。
- 定时器中断`IRQ_M_TIMER`：用于产生操作系统所需要的时钟节拍，一般设置软件中断频率为1000Hz。
- 异常：打印异常类型，打印栈帧，打印epc和当前线程。

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

Table 3.6: Machine cause register (`mcause`) values after trap.

- `mepc`机器模式异常程序计数器

- s0存放的是sp栈指针，s0即x8寄存器。

在`libcpu/risc-v/k210/interrupt.c`中定义了`handle_trap`的实现。

```
uintptr_t handle_trap(uintptr_t mcause, uintptr_t epc, uintptr_t * sp)
{
    int cause = mcause & CAUSE_MACHINE_IRQ_REASON_MASK;

    if (mcause & (1UL << 63))
    {
        switch (cause)
        {
            case IRQ_M_SOFT:
                {
                    uint64_t core_id = current_coreid();

                    clint_ipi_clear(core_id);
                    rt_schedule();
                }
                break;
            case IRQ_M_EXT:
                handle_irq_m_ext(mcause, epc);
                break;
            case IRQ_M_TIMER:
                tick_isr();
                break;
        }
    }
    else
    {
        rt_thread_t tid;
        extern long list_thread();

        rt_hw_interrupt_disable();

        tid = rt_thread_self();
        rt_kprintf("\nException:\n");
        switch (cause)
        {
            case CAUSE_MISALIGNED_FETCH:
                rt_kprintf("Instruction address misaligned");
                break;
            case CAUSE_FAULT_FETCH:
                rt_kprintf("Instruction access fault");
                break;
            case CAUSE_ILLEGAL_INSTRUCTION:
                rt_kprintf("Illegal instruction");
                break;
            case CAUSE_BREAKPOINT:
                rt_kprintf("Breakpoint");
                break;
            case CAUSE_MISALIGNED_LOAD:
                rt_kprintf("Load address misaligned");
                break;
            case CAUSE_FAULT_LOAD:
```

```
                rt_kprintf("Load access fault");
                break;
            case CAUSE_MISALIGNED_STORE:
                rt_kprintf("Store address misaligned");
                break;
            case CAUSE_FAULT_STORE:
                rt_kprintf("Store access fault");
                break;
            case CAUSE_USER_ECALL:
                rt_kprintf("Environment call from U-mode");
                break;
            case CAUSE_SUPERVISOR_ECALL:
                rt_kprintf("Environment call from S-mode");
                break;
            case CAUSE_HYPERVISOR_ECALL:
                rt_kprintf("Environment call from H-mode");
                break;
            case CAUSE_MACHINE_ECALL:
                rt_kprintf("Environment call from M-mode");
                break;
            default:
                rt_kprintf("Uknown exception : %08lX", cause);
                break;
        }
        rt_kprintf("\n");
        print_stack_frame(sp);
        rt_kprintf("exception pc => 0x%08x\n", epc);
        rt_kprintf("current thread: %.*s\n", RT_NAME_MAX, tid->name);
#ifdef RT_USING_FINSH
        list_thread();
#endif
        while(1);
    }

    return epc;
}
```

`handle_irq_m_ext` 用来处理来自PLIC的中断，定义在 `libcpu/risc-v/k210/interrupt.c` 中。此外，程序中 `if (mcause & (1UL << 63))` 感觉这里用逻辑与 `&&` 更适合。

```
uintptr_t handle_irq_m_ext(uintptr_t cause, uintptr_t epc)
{
    /*
     * After the highest-priority pending interrupt is claimed by a target
     * and the corresponding IP bit is cleared, other lower-priority
     * pending interrupts might then become visible to the target, and so
     * the PLIC EIP bit might not be cleared after a claim. The interrupt
     * handler can check the local meip/heip/seip/ueip bits before exiting
     * the handler, to allow more efficient service of other interrupts
     * without first restoring the interrupted context and taking another
     * interrupt trap.
     */
    if (read_csr(mip) & MIP_MEIP)
    {
        /* Get current core id */
```

```
        uint64_t core_id = current_coreid();
        /* Get primitive interrupt enable flag */
        uint64_t ie_flag = read_csr(mie);
        /* Get current IRQ num */
        uint32_t int_num = plic->targets.target[core_id].claim_complete;
        /* Get primitive IRQ threshold */
        uint32_t int_threshold = plic-
>targets.target[core_id].priority_threshold;
        /* Set new IRQ threshold = current IRQ threshold */
        plic->targets.target[core_id].priority_threshold = plic-
>source_priorities.priority[int_num];

        /* Disable software interrupt and timer interrupt */
        clear_csr(mie, MIP_MTIP | MIP_MSIP);

        if (irq_desc[int_num].handler ==
(rt_isr_handler_t)rt_hw_interrupt_handle)
        {
            /* default handler, route to kendryte bsp plic driver */
            plic_irq_handle(int_num);
        }
        else if (irq_desc[int_num].handler)
        {
            irq_desc[int_num].handler(int_num, irq_desc[int_num].param);
        }

        /* Perform IRQ complete */
        plic->targets.target[core_id].claim_complete = int_num;
        /* Set MPIE and MPP flag used to MRET instructions restore MIE flag */
        set_csr(mstatus, MSTATUS_MPIE | MSTATUS_MPP);
        /* Restore primitive interrupt enable flag */
        write_csr(mie, ie_flag);
        /* Restore primitive IRQ threshold */
        plic->targets.target[core_id].priority_threshold = int_threshold;
    }

    return epc;
}
```

在`rt_hw_interrupt_init`中会把各个中断源对应的中断服务程序初始化为`rt_hw_interrupt_handle`，然后通过`rt_hw_interrupt_install`来安装实际的中断服务程序来替换`rt_hw_interrupt_handle`。
查询当前中断号对应的中断服务程序是不是`rt_hw_interrupt_handle`，如果是则调用默认的中断服务程序，否则调用安装的中断服务程序。

Part4

```
//Part 4
#ifdef RT_USING_SMP
    /* s0 --> sp */
    mv  sp, s0
    mv  a0, s0
    call rt_scheduler_do_irq_switch
    tail   rt_hw_context_switch_exit
```

```
#else

    /* switch to from_thread stack */
    move  sp, s0

    /* need to switch new thread */
    la    s0, rt_thread_switch_interrupt_flag
    lw    s2, 0(s0)
    beqz  s2, spurious_interrupt
    sw    zero, 0(s0)

    la    s0, rt_interrupt_from_thread
    LOAD  s1, 0(s0)
    STORE sp, 0(s1)

    la    s0, rt_interrupt_to_thread
    LOAD  s1, 0(s0)
    LOAD  sp, 0(s1)

#endif

spurious_interrupt:
    tail rt_hw_context_switch_exit
```

中断处理的操作执行完后，还有两件事情要做，**一是确定是否要进行线程切换，二是要恢复现场。**`mv sp, s0`是把先前保存的线程栈指针恢复到`sp`中。

切换新线程。

```
    la    s0, rt_interrupt_from_thread
    LOAD  s1, 0(s0)
    STORE sp, 0(s1)

    la    s0, rt_interrupt_to_thread
    LOAD  s1, 0(s0)
    LOAD  sp, 0(s1)
```

这一部分就是线程的切换核心代码。其中`rt_interrupt_from_thread`是切换前线程，而`rt_interrupt_to_thread`是切换后的线程

下面是三段线程切换相关的函数：

1）rt_interrupt_from_thread

2）rt_interrupt_to_thread

3）rt_hw_context_switch_exit

```
.global rt_hw_context_switch_exit
rt_hw_context_switch_exit:
#ifdef RT_USING_SMP
#ifdef RT_USING_SIGNALS
    mv a0, sp

    csrr  t0, mhartid
```

```
    /* switch interrupt stack of current cpu */
    la    sp, __stack_start__
    addi  t1, t0, 1
    li    t2, __STACKSIZE__
    mul   t1, t1, t2
    add   sp, sp, t1 /* sp = (cpuid + 1) * __STACKSIZE__ + __stack_start__ */

    call rt_signal_check
    mv sp, a0
#endif
#endif
    /* resw ra to mepc */
    LOAD a0,   0 * REGBYTES(sp)
    csrw mepc, a0

    LOAD x1,   1 * REGBYTES(sp)

    li    t0, 0x00007800
    csrw  mstatus, t0
    LOAD a0,   2 * REGBYTES(sp)
    csrs mstatus, a0

    LOAD x4,   4 * REGBYTES(sp)
    LOAD x5,   5 * REGBYTES(sp)
    LOAD x6,   6 * REGBYTES(sp)
    LOAD x7,   7 * REGBYTES(sp)
    LOAD x8,   8 * REGBYTES(sp)
    LOAD x9,   9 * REGBYTES(sp)
    LOAD x10, 10 * REGBYTES(sp)
    LOAD x11, 11 * REGBYTES(sp)
    LOAD x12, 12 * REGBYTES(sp)
    LOAD x13, 13 * REGBYTES(sp)
    LOAD x14, 14 * REGBYTES(sp)
    LOAD x15, 15 * REGBYTES(sp)
    LOAD x16, 16 * REGBYTES(sp)
    LOAD x17, 17 * REGBYTES(sp)
    LOAD x18, 18 * REGBYTES(sp)
    LOAD x19, 19 * REGBYTES(sp)
    LOAD x20, 20 * REGBYTES(sp)
    LOAD x21, 21 * REGBYTES(sp)
    LOAD x22, 22 * REGBYTES(sp)
    LOAD x23, 23 * REGBYTES(sp)
    LOAD x24, 24 * REGBYTES(sp)
    LOAD x25, 25 * REGBYTES(sp)
    LOAD x26, 26 * REGBYTES(sp)
    LOAD x27, 27 * REGBYTES(sp)
    LOAD x28, 28 * REGBYTES(sp)
    LOAD x29, 29 * REGBYTES(sp)
    LOAD x30, 30 * REGBYTES(sp)
    LOAD x31, 31 * REGBYTES(sp)

    addi sp,  sp, 32 * REGBYTES

#ifdef ARCH_RISCV_FPU
    FLOAD   f0, 0 * FREGBYTES(sp)
```

```
    FLOAD   f1, 1 * FREGBYTES(sp)
    FLOAD   f2, 2 * FREGBYTES(sp)
    FLOAD   f3, 3 * FREGBYTES(sp)
    FLOAD   f4, 4 * FREGBYTES(sp)
    FLOAD   f5, 5 * FREGBYTES(sp)
    FLOAD   f6, 6 * FREGBYTES(sp)
    FLOAD   f7, 7 * FREGBYTES(sp)
    FLOAD   f8, 8 * FREGBYTES(sp)
    FLOAD   f9, 9 * FREGBYTES(sp)
    FLOAD   f10, 10 * FREGBYTES(sp)
    FLOAD   f11, 11 * FREGBYTES(sp)
    FLOAD   f12, 12 * FREGBYTES(sp)
    FLOAD   f13, 13 * FREGBYTES(sp)
    FLOAD   f14, 14 * FREGBYTES(sp)
    FLOAD   f15, 15 * FREGBYTES(sp)
    FLOAD   f16, 16 * FREGBYTES(sp)
    FLOAD   f17, 17 * FREGBYTES(sp)
    FLOAD   f18, 18 * FREGBYTES(sp)
    FLOAD   f19, 19 * FREGBYTES(sp)
    FLOAD   f20, 20 * FREGBYTES(sp)
    FLOAD   f21, 21 * FREGBYTES(sp)
    FLOAD   f22, 22 * FREGBYTES(sp)
    FLOAD   f23, 23 * FREGBYTES(sp)
    FLOAD   f24, 24 * FREGBYTES(sp)
    FLOAD   f25, 25 * FREGBYTES(sp)
    FLOAD   f26, 26 * FREGBYTES(sp)
    FLOAD   f27, 27 * FREGBYTES(sp)
    FLOAD   f28, 28 * FREGBYTES(sp)
    FLOAD   f29, 29 * FREGBYTES(sp)
    FLOAD   f30, 30 * FREGBYTES(sp)
    FLOAD   f31, 31 * FREGBYTES(sp)

    addi    sp, sp, 32 * FREGBYTES
#endif

    mret
```

如果定义了`RT_USING_SMP`和`RT_USING_SIGNALS`，则先切换到中断栈，执行完`rt_signal_check`后再切回线程栈。
最后的`mret`指令会把PC值设置为`mepc`中保存的值。

参考

[1]. 自制操作系统: risc-v Machine寄存器mscratch/mepc/mcause/mtval

[2]. RT-Thread在Kendryte K210上的trap处理过程