PA 3

201300035 方盛俊

1. 阶段一: 穿越时空的旅程

1.1 设置异常入口地址

要想正确实现 `csrrw` 指令, 就需要先加入 `mtvec`, `mcause`, `mstatus`, `mepc` 这四个寄存器, 然后再使用这些寄存器实现第一条 CSR 指令.

1. 加入 CSR 寄存器
    1. 在 `isa-def.h` 加入 CSR 的四个寄存器 `mtvec`, `mcause`, `mstatus`, `mepc`
    2. 在 `reg.h` 中加入 `csr(idx)` 全局宏
    3. 在 `reg.c` 中修改寄存器有关的函数, 如 `isa_reg_display()`
2. 加入 `csrrw` 指令
    1. 在 `isa-all-instr.h` 之类的添加就不过多赘述
    2. 在 `decode.c` 里加入了 `def_DopHelper(csr)` 辅助函数
        1. 用于将 12 位的 csr 地址映射到只有 4 个的 csr 寄存器中 (暂时)
        2. 这步踩了很多坑...
    3. 在 `csr.h` 加入 `def_EHelper(csrrs)` 实现对应指令

做完这些之后, 终于能开始写自陷操作了...

1.2 触发自陷操作

1. 完成 `ecall` 指令
    1. `rtl_j(s, isa_raise_intr(11, cpu.pc + 4)); // 异常号 11, 代表 Environment call from M-mode`
    2. 注意不要与 `mret` 重合.
2. 完成 `csrrs` 指令
3. 完成 `mret` 指令, 返回到 `mepc` 寄存器所保存的地址

这里踩了很多坑, difftest 还出问题了...

1.3 重新组织结构体

通过观察 `trap.S` 的内容, 对 `Context` 重新整理如下:

```
struct Context {
  // fix the order of these members to match trap.S
  uintptr_t gpr[32], mcause, mstatus, mepc;
  void *pdir;
};
```

1.4 实现正确的事件分发

1. 在 `__am_irq_handle()` 加入了 `case -1: ev.event = EVENT_YIELD; break;`, 识别异常号 `-1`, 并打包为 `EVENT_YIELD` 事件.

2. 在 `do_event()` 加入了 `case EVENT_YIELD: printf("Event: Yield\n"); break;`, 识别出自陷事件 `EVENT_YIELD`, 然后输出 `Event: Yield`.

1.5 理解上下文结构体的前世今生 & 理解穿越时空的旅程

从 Nanos-lite 调用 `yield()` 开始, 到从 `yield()` 返回的期间, 这一趟旅程具体经历了什么?

前置工作: 初始化 `init_irq(void)`:

1. `init_irq(void)` 调用了 `cte_init(do_event)`, 将 `do_event()` 这个函数传入;
2. `cte_init()` 调用 `asm volatile("csrw mtvec, %0" : : "r"(__am_asm_trap))` 将 `__am_asm_trap()` 函数地址保存在 `mtvec` 中;
3. `cte_init()` 调用 `user_handler = handler` 将 `do_event` 保存在全局变量中, 以便后续回调;

正式工作: 调用 `yield()`:

1. `yield()` 调用了 `asm volatile("li a7, -1; ecall")`, 使用 `ecall` (设置异常号 `11`) 跳转到 `mtvec` 寄存器的 `__am_asm_trap()` 函数中;
2. `__am_asm_trap()` 使用 `addi sp, sp, -CONTEXT_SIZE` 在堆栈区初始化了 `CONTEXT_SIZE` 大小的上下文结构体 `c`, **这是上下文结构体生命周期的开端**;
3. `__am_asm_trap()` 使用 `MAP(REGS, PUSH)` 的宏展开式函数映射编程法, 类似于 `PUSH(REGS)` 这样, 将 `32` 个通用寄存器保存到了 `c` 中相应位置;
4. `__am_asm_trap()` 使用类似于 `csrr t0, mcause; STORE t0, OFFSET_CAUSE(sp)` 的汇编语句将 `mcause`, `mstatus` 和 `mepc` 三个寄存器保存到了 `c` 中相应位置;
5. `__am_asm_trap()` 将 `mstatus.MPRV` 置位, 以便通过 difftest;
6. `__am_asm_trap()` 使用 `mv a0, sp; jal __am_irq_handle` 将位于堆栈区的 `c` 上下文结构体保存到函数传参寄存器 `a0` 中, 作为函数参数调用并传给 `__am_irq_handle()` 函数;
7. `__am_irq_handle()` 通过 `c->mcause` 判别异常号, 并创建对应 `Event ev`, 调用 `user_handler(ev, c)`, 即调用上文提到的 `do_event(e, c)`;
8. `do_event()` 对异常或中断做完相应处理后, 返回到 `__am_irq_handle()` 中;
9. `__am_irq_handle()` 也做完了相应处理, 返回到 `__am_asm_trap()` 中;
10. `__am_asm_trap()` 使用类似于 `LOAD t1, OFFSET_STATUS(sp); csrw mstatus, t1` 的汇编语句将 `c` 中相应位置保存到 `mstatus` 和 `mepc` 两个寄存器中;
11. `__am_asm_trap()` 使用 `MAP(REGS, POP)` 将 `c` 中相应位置数据复原回 `32` 个通用寄存器中;
12. `__am_asm_trap()` 使用 `addi sp, sp, CONTEXT_SIZE` 将堆栈区复原, 相当于将 `c` 释放, **这是上下文结构体生命周期的结束**;
13. `__am_asm_trap()` 使用 `mret`, 将 `mtvec` 寄存器内保存的数据取出, 并跳转到该位置, 即回到了调用中断代码的 `yield()` 函数中;
14. `yield()` 处理完所有事情, 便返回了, 进而调用了 `panic("Should not reach here")`.

此外, 我还在 `trap.S` 的 `csrw mepc, t2` 指令前加入了 `addi t2, t2, 4`, 来实现自陷指令 `ecall` PC 加 4 的效果.

1.6 异常处理的踪迹 - etrace

修改 Kconfig, 并在 `intr.c` 的 `isa_raise_intr(word_t NO, vaddr_t epc)` 中加入

```
#ifdef CONFIG_ETRACE
log_write("[etrace] mcause: %d, mstatus: %x, mepc: %x\n", cpu.csr[1]._32, cpu.csr[2]._32, cpu.csr[
#endif
```

即可.