实验二：RISC-V CPU 中断处理机制设计与实现

朱梓涵

学号：24325356

2025 年 12 月 21 日

1 实验目的

本实验旨在为已实现的单周期 RISC-V CPU 增加中断与异常处理功能，深入理解现代处理器如何响应并处理非顺序控制流事件。实验目标包括：

1. 设计并实现控制状态寄存器模块，支持 CSR 指令的读写操作。
2. 设计并实现核本地中断控制器，使其能够正确处理外部硬件中断和内部软件中断以及中断返回指令。
3. 理解中断处理流程中 `mstatus`, `mepc`, `mcause`, `mtvec` 等关键寄存器的作用与变化。
4. 学习通过 Chisel 测试用例和波形图分析，验证复杂的中断处理逻辑的正确性。

2 实验环境

- **操作系统**: Windows 11
- **开发工具**: IntelliJ IDEA
- **构建工具**: SBT
- **仿真与测试**: Verilator, GTKWave, MSYS2

3 模块实现与分析

本次实验的核心是新增 `CSR` 和 `CLINT` 模块，并对 `Execute` 模块进行扩展以支持 CSR 指令。

3.1 CSR 模块

CSR 模块是 CPU 的状态管理中心，负责存储和更新如 `mstatus`, `mepc` 等关键状态寄存器。

- **实现要点：**

  1. **独立寄存器与查找表：**将重要的 CSR（如 `mstatus`）实现为独立的物理寄存器，并通过一个查找表 `regLUT` 响应读请求，设计清晰且高效。
  2. **写操作优先级：**写入逻辑必须处理来自 CLINT（中断/异常事件）和 Execute（CSR 指令）的并发写请求。设计中，CLINT 的写入拥有最高优先级，确保了中断处理的原子性。
  3. **数据旁路：**为解决数据冒险，设计了旁路机制。CSR 模块会预计算出下一拍寄存器的值并立即提供给 CLINT，使其能在当前周期就基于最新的 CPU 状态做出正确决策。

3.1.1 代码实现

```
val mstatus_next = Mux(io.reg_write_enable_id &&
  io.reg_write_address_id === CSRRegister.MSTATUS,
  io.reg_write_data_ex, mstatus)
val mepc_next = Mux(io.reg_write_enable_id &&
  io.reg_write_address_id === CSRRegister.MEPC,
  io.reg_write_data_ex, mepc)

io.clint_access_bundle.mstatus := mstatus_next
io.clint_access_bundle.mepc := mepc_next

when(io.clint_access_bundle.direct_write_enable) {
  mstatus := io.clint_access_bundle.mstatus_write_data
  mepc := io.clint_access_bundle.mepc_write_data
  mcause := io.clint_access_bundle.mcause_write_data
}.elsewhen(io.reg_write_enable_id) {
  when(io.reg_write_address_id === CSRRegister.MSTATUS) {
    mstatus := io.reg_write_data_ex
  }
}
```

Listing 1: CSR 模块的写入优先级与旁路逻辑

3.2 CLINT 模块

CLINT 是中断处理的决策中心，负责监视 CPU 状态，判断中断与异常事件，并生成控制信号来改变 CPU 的执行流和状态。

- **实现要点：**

  1. **事件检测:**通过组合逻辑判断外部中断信号 (`interrupt_flag`) 和特定指令（`mret`, `ecall`, `ebreak`）的发生。
  2. **状态更新计算：**根据发生的事件类型，精确计算 `mepc`, `mcause`, `mstatus` 这三个 CSR 寄存器需要更新成的新值。
  3. **PC 重定向：**当陷阱 (trap) 发生或 `mret` 执行时，置位 `interrupt_assert` 标志，并提供新的 PC 地址（中断时为 `mtvec`，返回时为 `mepc`）。
  4. **处理优先级：**通过一个 `when-elsewhen-otherwise` 结构明确了事件处理的优先级：外部硬件中断 > `mret` > `ecall` > `ebreak`。

3.2.1 代码实现

```
val interrupt_enable = io.csr_bundle.mstatus(3)
val instruction_address = Mux(
  io.jump_flag,
  io.jump_address,
  io.instruction_address + 4.U
)
val mstatus = io.csr_bundle.mstatus
val mie = mstatus(3)

when(io.interrupt_flag =/= InterruptCode.None && interrupt_enable) {
  io.interrupt_assert := true.B
  io.csr_bundle.direct_write_enable := true.B
  io.interrupt_handler_address := io.csr_bundle.mtvec
  io.csr_bundle.mepc_write_data := instruction_address
  io.csr_bundle.mcause_write_data := "h80000007".U
```

```
    val new_mpie = mie << 7
    io.csr_bundle.mstatus_write_data :=
      (mstatus & (~(1.U << 3)).asUInt) | new_mpie | (3.U << 11)
  }.elsewhen(io.instruction === InstructionsRet.mret) {
  }
```

Listing 2: CLINT 模块处理硬件中断的核心逻辑

4 测试与结果分析

4.1 CLINTCSRTest：硬件中断测试分析

4.1.1 测试机制简述

该测试旨在验证 CPU 对外部硬件中断的响应是否正确。测试用例通过 `chiseltest` 框架，模拟外部定时器中断信号，并验证 CLINT 模块在不同场景下的中断处理行为。

输入信号：

- `interrupt_flag`: 外部中断标志，测试中使用值为 `0x1` 的定时器中断
- `instruction`: 当前执行的指令
- `instruction_address`: 当前指令地址，表示被中断时的 PC 值
- `jump_flag`: 指示当前指令是否为跳转指令
- `jump_address`: 跳转目标地址
- `csr_bundle.mtvec`: 预设的中断向量表基地址 `0x1144`
- `csr_bundle.mstatus`: 初始值 `0x1888`，MIE=1, MPIE=1，使能中断

测试的 CLINT 功能：

1. **中断检测:**检测外部中断信号 (`interrupt_flag`) 并判断是否应该响应（检查 `mstatus.MIE` 位）
2. **上下文保存：**将中断发生时的关键状态保存到 CSR 寄存器：
   - `MEPC` ← PC + 4（非跳转）或跳转目标地址（跳转）
   - `MCAUSE` ← 中断原因编码（`0x80000007` 表示定时器中断）

- MSTATUS ← 更新状态（MIE ← 0, MPIE ← 原 MIE 值）

3. **中断跳转**：跳转到中断处理程序（mtvec 中存储的地址 0x1144）
4. **中断返回**：执行 mret 指令时恢复现场（PC ← MEPC，MIE ← MPIE）
5. **跳转与非跳转指令的差异**：验证在跳转指令执行期间发生中断时，MEPC 保存的是跳转目标地址而非 PC + 4

4.1.2 波形图分析：非跳转指令下的硬件中断处理

本次测试选用 CLINTCSRTest.scala 中的硬件中断测试（handle external interrupt）。测试通过手动向 CPU 输入一个外部硬件中断标志 io_interrupt_flag，并观察 CLINT 是否正确生成中断、是否能在非跳转指令下按照 RISC-V 标准流程完成一次完整的中断处理。

测试输入信号及作用：

信号	作用
io_interrupt_flag = 1	触发一次硬件中断（定时器中断）
io_instruction = 0x13	当前执行指令（NOP，0x00000013）
io_instruction_address = 0x1900	当前 PC 值
io_jump_flag = 0	非跳转指令标志
mtvec、mstatus 初始写入	配置中断入口（0x1144）及打开 MIE

表 4.1: 硬件中断测试输入信号

此测试用来验证 CLINT 是否能完成以下功能：

- 在非跳转指令下正确响应硬件中断
- 正确生成中断断点（mepc = PC + 4）
- 正确写入中断原因（mcause = 0x80000007）
- 自动清除 MIE 并保存到 MPIE（mstatus 更新）
- 正确跳转到中断处理入口（mtvec）

波形图关键信号说明：

- io_interrupt_flag[31:0]: 外部中断标志输入
- io_instruction[31:0]: 当前执行的指令

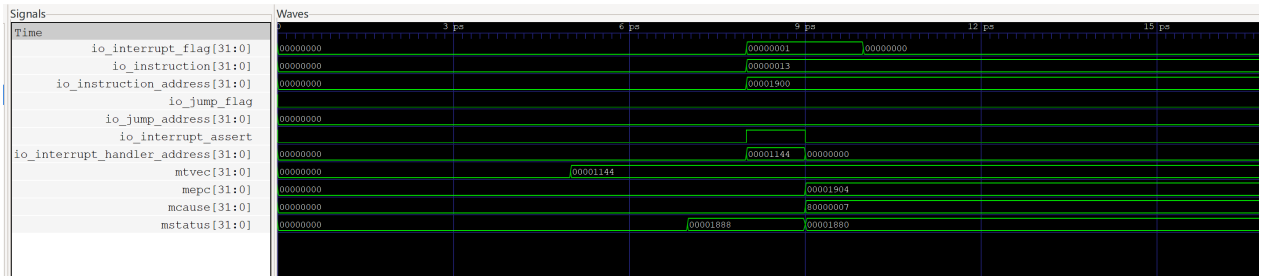

图 4.1: 硬件中断处理过程波形图 - 非跳转指令场景（8ps ∼ 12ps）

- `io_instruction_address[31:0]`: 当前指令地址（PC）
- `io_jump_flag`: 跳转指令标志
- `io_jump_address[31:0]`: 跳转目标地址
- `io_interrupt_assert`: CLINT 输出的中断响应信号
- `io_interrupt_handler_address[31:0]`: 中断跳转目标地址
- `mtvec[31:0]`: 中断向量表基地址
- `mepc[31:0]`: 中断返回地址
- `mcause[31:0]`: 中断原因寄存器
- `mstatus[31:0]`: 机器状态寄存器

如图所示，本测试一次完整的硬件中断处理过程（波形截图区间约为 8ps ∼ 12ps）。以下挑选关键信号说明中断发生与处理的整个过程：

(1) 初始化阶段（约 6ps） 在中断发生前，测试代码已通过 CSR 写指令完成初始化：

- `mtvec = 0x00001144`：中断处理程序入口地址已配置
- `mstatus = 0x00001888`：全局中断已使能
  - 二进制表示为 `...0001_1000_1000_1000`
  - bit[3] MIE = 1（全局中断使能）
  - bit[7] MPIE = 1（中断前的 MIE 备份）

(2) 中断发生时刻（约 9ps） 此时 CPU 正在执行一条普通的 NOP 指令，外部中断请求到来：

- `io_instruction_address = 0x00001900`：当前 PC
- `io_instruction = 0x00000013`：NOP 指令
- `io_jump_flag = 0`：非跳转指令
- `io_interrupt_flag`：从 `0x00000000` → `0x00000001`（Timer0 中断）

(3) 同周期中断响应（约 9ps） CLINT 模块检测到 `interrupt_flag = 1` 且 `mstatus.MIE = 1` 后，立即通过组合逻辑响应中断：

- `io_interrupt_assert = 1`：触发中断信号
- `io_interrupt_handler_address = 0x00001144`：指示 CPU 跳转到 `mtvec`
- 波形图上可观察到 `io_interrupt_assert` 出现一个脉冲
- `io_interrupt_handler_address` 短暂输出 `0x1144` 后恢复为 `0`

这表示 CPU 将在下一时钟周期跳转到中断处理程序入口。

(4) CSR 自动更新(约 10ps） 在时钟上升沿,CSR 模块根据 CLINT 的 `direct_write_enable` 信号自动更新关键寄存器：

- `mepc`：从 `0x00000000` → `0x00001904`
  - 保存的是 PC + 4（`0x1900 + 4 = 0x1904`）
  - 这是被中断指令的下一条指令地址，中断返回后将从此处继续执行
- `mcause`：从 `0x00000000` → `0x80000007`
  - bit[31] = 1：表示这是异步硬件中断（而非同步异常）
  - 低位 = 7：对应 Timer 中断编码
- `mstatus`：从 `0x00001888` → `0x00001880`
  - `0x1888` = `...0001_1000_1000_1000`（MIE=1, MPIE=1）
  - `0x1880` = `...0001_1000_1000_0000`（MIE=0, MPIE=1）

- MIE 被清 0：关闭全局中断，防止中断嵌套
- MPIE 保存了先前的 MIE 值（=1）

这些变化完全符合 RISC-V 特权架构手册中定义的中断进入流程。

(5) 中断标志清除（约 12ps 后） 波形图显示 `io_interrupt_flag` 在约 12ps 后从 1 恢复为 0，这是测试代码模拟中断处理程序清除外设中断标志的行为。

波形分析总结：

通过以上波形图分析，可以验证 CLINT 模块在非跳转指令场景下正确实现了以下功能：

- `mepc` 正确保存了 PC + 4（`0x1904`），确保中断返回后从下一条指令继续执行
- `mcause` 正确记录了中断原因（`0x80000007`），bit[31]=1 标识为异步中断
- `mstatus` 自动完成 MIE 清零和 MPIE 备份，实现中断嵌套保护机制
- 从中断检测到 CSR 更新的整个过程由硬件自动完成，无需软件干预

测试还包括跳转指令场景的验证：当 `jump_flag=1`, `jump_address=0x1990` 时发生中断，`mepc` 应保存跳转目标 `0x1990` 而非 `PC+4`，且 `mcause=0x8000000B`。这确保了 CLINT 能正确处理各种执行场景下的中断。

4.2 CPUTest: SimpleTrapTest 分析

4.2.1 测试目的

本测试通过执行 `csrc/simpletest.c` 中的测试程序，验证 CPU 是否能够按照 RISC-V 标准正确处理中断，包括中断触发、保存现场、跳转到中断处理程序、执行处理逻辑以及返回主程序等完整流程。

4.2.2 测试程序的中断验证机制

`simpletest.c` 通过以下过程验证 CPU 的中断处理正确性：

```
extern void enable_interrupt();

void trap_handler(void *epc, unsigned int cause){
    *((int*)0x4) = 0x2022;
}
```

```
int main(){
    *((int*)0x4) = 0xDEADBEEF;
    enable_interrupt();
    for(;;);
}
```

Listing 3: simpletest.c 测试程序源码

主程序初始化阶段 程序首先向内存地址 0x4 写入标记值 0xDEADBEEF，用于表示” 尚未处理中断” 的初始状态。随后调用 enable_interrupt() 函数配置中断环境：

- 将 trap_handler 的地址写入 mtvec 寄存器
- 设置 mstatus.MIE = 1，使能全局中断

等待中断触发 程序进入无限循环 for(;;)，持续等待中断到来。测试框架 TestTopModule 在仿真过程中通过 io.interrupt_flag.poke(0x1) 向 CPU 注入外部中断请求。

中断处理程序执行 当中断触发后，CPU 自动跳转到 trap_handler 函数。该函数执行唯一的操作：将内存地址 0x4 的值修改为 0x2022。这一修改作为中断处理程序成功执行的关键证据。

中断返回与验证 trap_handler 执行完毕后，通过 mret 指令返回主程序。测试代码随后读取内存地址 0x4 和相关 CSR 寄存器，验证：

- 内存值已从 0xDEADBEEF 变为 0x2022
- mstatus 恢复为 0x1888（中断返回后状态）
- mcause 保持 0x80000007（记录中断原因）

该验证机制的核心在于：若无限循环能够被中断打断，且内存值发生预期变化，则证明 CPU 确实跳转到了 trap_handler，中断处理程序正确执行，且 mret 返回机制正常工作。这种通过可观测副作用验证复杂流程的方法，是嵌入式系统测试的典型手段。

4.2.3 波形图分析

本实验从波形图中截取了两段关键片段，分别展示中断触发与进入处理、以及中断处理程序修改内存的核心过程。

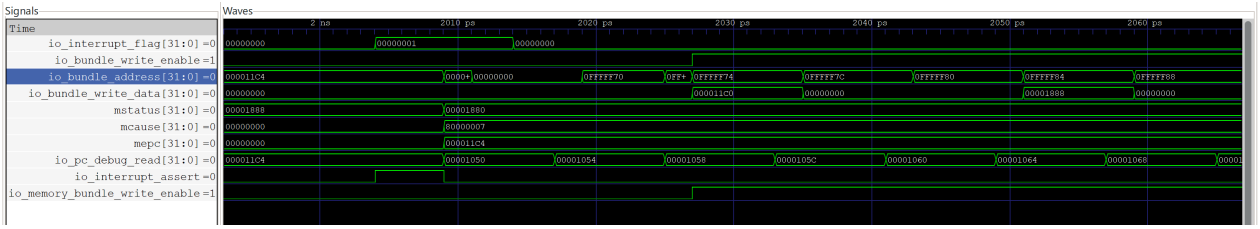

图 4.2: SimpleTrapTest 中断触发与进入处理过程波形图（约 2000ps）

(1) 中断触发与进入中断处理 如图所示，展示了中断触发至 CPU 进入中断处理程序的完整时序。波形图中的关键信号及其变化如下：

- `io_interrupt_flag`: 在约 2000ps 从 0 → `0x00000001`，表示外部中断请求到来，随后恢复为 0
- `io_interrupt_assert`: 随即产生一个脉冲，说明 CLINT 已检测到中断并产生中断响应信号
- `mepc`: 更新为 `0x000011C4`，即被中断时刻 PC 的下一条指令地址，保存了中断返回点
- `mcause`: 被写入 `0x00000007`，记录中断原因为定时器中断（外部中断编码）
- `mstatus`: 从 `0x00001888` → `0x00001880`，MIE 位（bit[3]）被自动清零，MPIE 位（bit[7]）保存了先前的 MIE 值，符合 RISC-V 中断进入时的标准行为
- `io_pc_debug_read`: 从原执行地址跳转到 `0x00001050` 附近，该地址为 `mtvec` 指向的 `trap_handler` 函数入口，随后 PC 逐条递增执行中断处理程序指令

这一段波形完整展示了中断触发、CSR 自动保存、PC 跳转到 `trap_handler` 的全过程。

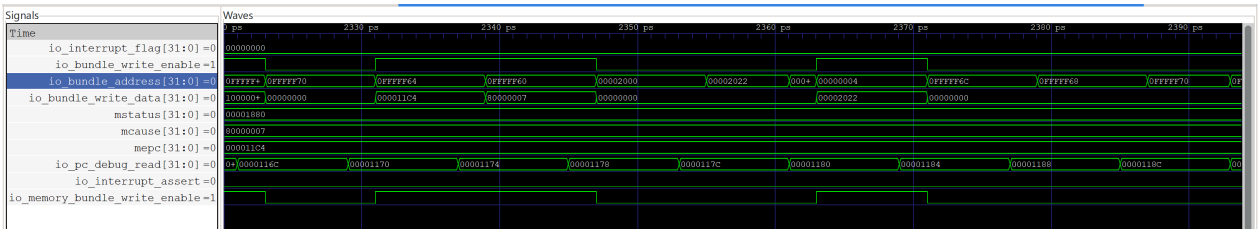

图 4.3: SimpleTrapTest 中断处理程序修改内存标志（约 2330ps ∼ 2390ps）

(2) 中断处理程序修改内存值 如图所示，展示了验证程序成功执行的最关键时刻。该时段内 CPU 正在执行 `trap_handler` 函数体，波形图清晰记录了以下信号变化：

- io_bundle_write_enable = 1: CPU 正在执行内存写操作
- io_bundle_address = 0x00000004: 写入的目标地址正是 simpletest.c 中指定的内存地址 0x4
- io_bundle_write_data = 0x00002022: 写入的数据为 0x2022，即 trap_handler 函数中期望写入的新值
- mstatus: 保持 0x00001880，说明此时仍处于中断处理状态，MIE 位为 0
- mcause: 保持 0x80000007，中断原因记录未变
- mepc: 保持 0x000011C4，中断返回地址保持不变
- io_pc_debug_read: 在 trap_handler 函数内部的指令地址间递增，表明 CPU 正在顺序执行中断处理程序

这一瞬间表明 CPU 已成功进入 trap_handler，并执行了中断处理中最核心的操作——将内存地址 0x4 的值从 0xDEADBEEF 替换为 0x2022。这是 SimpleTrapTest 验证的直接依据，也是实验要求中明确指出的”程序成功执行”的关键信号。

波形分析总结 通过以上两段波形图分析，可以验证：

- CPU 能够在外部中断到来时正确进入中断处理流程
- CSR 寄存器（mepc、mcause、mstatus）均按 RISC-V 标准顺序自动更新
- PC 正确跳转到 mtvec 指向的 trap_handler 函数
- 中断处理函数成功执行，完成了将内存地址 0x4 的值从 0xDEADBEEF 修改为 0x2022 的操作
- 测试代码后续验证了 mret 返回后 mstatus 恢复为 0x1888，中断返回机制正常工作

综上所述，本次测试成功验证了 CPU 中断处理机制的完整性与正确性。

4.3 CPU 与操作系统协作处理定时器中断的机制

假设本实验设计的 CPU 上运行着一个简单的操作系统（如嵌入式实时操作系统），当定时器中断发生时，硬件（CPU 与 CLINT）和软件（操作系统内核）将协同完成中断处理的全过程。

4.3.1 操作系统初始化阶段

操作系统在启动过程中，执行特权指令（如 `csrrw`）来初始化中断处理机制：

- 向 `mtvec` 寄存器写入中断分发程序 (interrupt dispatcher) 的入口地址
- 配置定时器硬件，设置定时周期和中断使能位
- 设置 `mstatus` 寄存器的 MIE 位为 1，使能全局中断

4.3.2 硬件自动响应

定时器硬件在倒计时结束后，向 CPU 发送中断信号。CPU 的 CLINT 模块检测到该信号且 `mstatus.MIE = 1` 时，硬件自动执行以下原子操作：

1. 将 `mstatus.MIE` 位的值备份到 `mstatus.MPIE` 位，然后将 `mstatus.MIE` 清零，防止中断嵌套
2. 将当前 PC 的下一条指令地址存入 `mepc` 寄存器
3. 根据中断源，在 `mcause` 寄存器中写入原因码（定时器中断为 `0x80000007`）
4. 读取 `mtvec` 寄存器的值，将 PC 强制设置为该地址，跳转到中断分发程序

4.3.3 操作系统软件处理

CPU 跳转到操作系统预设的中断分发程序，该程序执行以下操作：

1. **保存上下文**：将所有通用寄存器（`x1`∼`x31`）压栈，保存到内核栈中
2. **识别中断源**：读取 `mcause` 寄存器，判断中断类型
3. **分发处理**：根据中断类型调用相应的中断服务例程。对于定时器中断，调用定时器中断服务例程
4. **执行中断服务例程**：定时器中断服务例程执行核心任务：
   - 更新系统时钟计数器
   - 检查并唤醒睡眠超时的任务
   - 执行任务调度算法，决定是否需要任务切换
   - 重新配置定时器，设置下一次中断
5. **恢复上下文**：从内核栈中恢复所有通用寄存器
6. **执行 `mret`**：返回被中断的程序

4.3.4 硬件恢复

执行 `mret` 指令时，CPU 硬件自动执行以下操作：

1. 将 `mstatus.MPIE` 的值恢复到 `mstatus.MIE`，重新使能全局中断
2. 将 `mepc` 中保存的地址加载到 PC，返回被中断的程序

至此，CPU 无缝返回到被中断的用户程序继续执行。

5 实验结论

本次实验，我成功地为单周期 CPU 添加了完整的中断处理功能。通过设计 CSR 和 CLINT 模块，我深入学习了 RISC-V 的特权架构和中断处理流程，对 `mstatus`, `mepc`, `mcause` 等核心 CSR 的作用有了实践层面的深刻理解。解决 Windows 环境配置难题和调试复杂中断逻辑的过程，极大地锻炼了我分析问题和解决问题的能力。通过将理论知识与硬件实现、软件测试相结合，我不仅验证了 CPU 设计的正确性，也对操作系统与硬件的交互机制有了更具体的认识，为未来更深入的系统级学习打下了坚实的基础。