《计算机组成原理实验》

实验报告

（实验二）

学 院 名 称 ：	计算机学院
专业（班级）：	21 计教学 1 班
学 生 姓 名 ：	凌国明
学　　号 ：	21307077
时　　间 ：	2022　年　1　月　14　日

成 绩 ：

实验二 ： 中断

一. 实验目的

1. 学习中断控制器的原理，并根据原理进行设计
2. 使CPU有处理外部中断的能力，并能保护现场，返回现场继续执行程序。

二. 实验内容

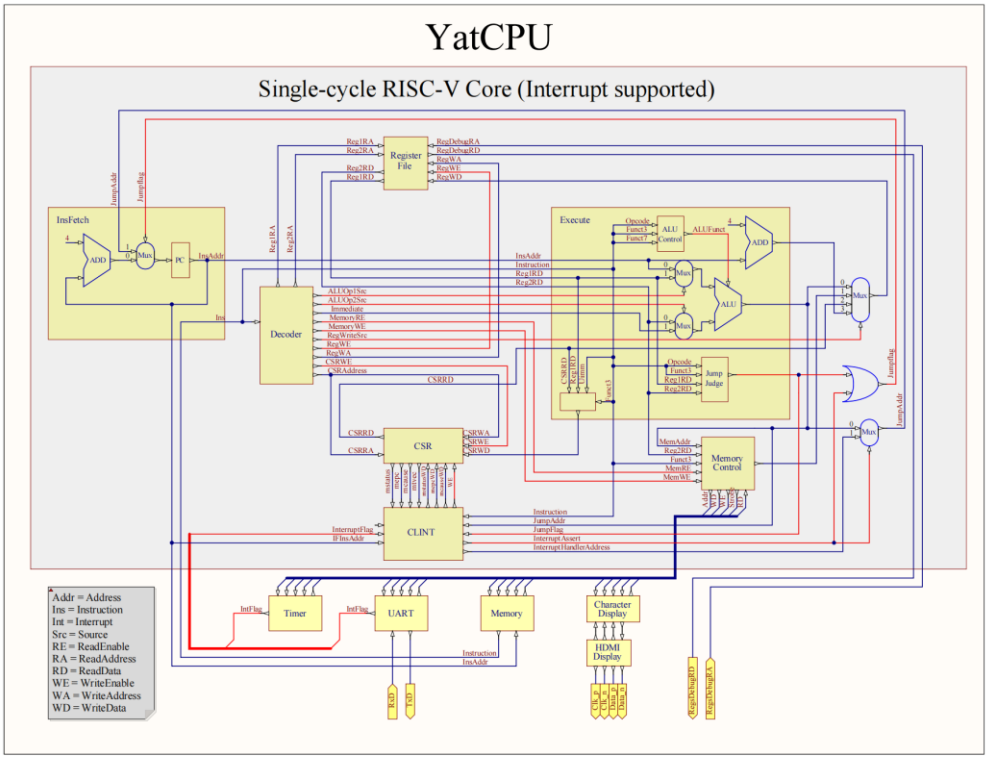
1、使ex执行单元在处理 CSR 指令时能够正确地得到写入 CSR 寄存器的数据。

2、使CSR寄存器组可以正确支持CLINT和来自CSR指令的读写操作。

3、使中断发生器可以正确产生中断信号，并且实现 Timer 寄存器的 MMIO。

4、使CLINT 能够正确的响应中断并且在中断结束后回到原来的执行流。

三. 实验原理



CSR 寄存器组：CSR 寄存器是一组类似于 RegisterFile 寄存器组的，地址空间大小为 4096 字节，独立编址的寄存器。 从指令手册可以看到对 CSR 寄存器的操作都是原子读写的，CSR 指令具体的语义请查阅手册。 CSR 寄存器组需要根据 ID 模块译码后给出的控制信号和 CSR 寄存器地址，来对内部寄存器进行寻址，获取其内容并且修改。

ID 译码单元：ID 译码单元需要识别 CSR 指令，根据手册里描述的指令语义和编码规范，产生相应的传给其它模块的控制信号与数据。

EX 执行单元：CSR 指令都是原子读写的，即一条指令的执行结果中，既要把目标 CSR 寄存器原来的内容写入到目标通用寄存器中，还要按指令语义把从目标 CSR 寄存器读出来的内容修改之后再写回给该 CSR 寄存器。此时 EX 里面的 ALU 单元是空闲的，要得到写入 CSR 寄存器的值，可以复用 ALU，也可以不复用。

WB 写回单元：支持 CSR 相关操作指令后，写回到目标通用寄存器的数据来源就多了一个从目标 CSR 寄存器读出来的修改前的值。

中断控制器：

中断控制器要完成的任务就是检测外部中断，在中断到来并且中断使能时，会中断 CPU 目前的执行流，设置好相关 CSR 寄存器信息后跳转到中断处理程序中执行中断处理程序。

关键就是该保存哪些信息到对应的CSR寄存器中，答案就是 CPU 执行完当前指令后的下一个状态。比如当前指令是跳转指令，那么 mepc 保存的应该是当前跳转指令的跳转目标地址。

还有一些特殊情况。我们知道外部中断使能由 mstatus 内容决定，那么要思考如果当前指令如果是修改 mstatus 执行结果是关中断的指令执行时，外部中断到来了，那么下个周期是否应该响应中断？ 为了统一起见，我们认为这种情况不应该响应中断，并且我们认为应该让当前指令执行完后，再跳转到中断处理程序。

中断返回：

与响应中断对应的中断处理完成后（mret），需要恢复 CPU 执行流程，这时候其实干的事与响应中断是大致相同的，只不过需要写入的寄存器只有 mstatus，跳转的目标地址则是从 mepc 获取。对于 mret 时 mstatus 的要写入的值，我们为了简单起见，就把 MIE 位置为 MPIE 位，那么 MPIE 为 1 的话 mret 就会恢复中断，如果 MPIE 为 0 的话，mret 则不改变 mstatus 的值，这也导致了我们不支持中断嵌套。

CLINT的实现：

CLINT 具体的实现方法很多，为了简单起见，我们采用纯组合逻辑实现这个中断控制器。由于基于单周期 CPU 且 CLINT 是组合逻辑，所以外部中断到来时，CLINT 会马上响应。

CLINT 需要一个周期就把多个寄存器的内容修改的功能，而正常的 CSR 指令只能对一个寄存器读-修改-写（Read-Modify-Write，RMW）。所以 CLINT 和 CSR 之间有独立的优先级更高的通路，用来快速更新 CSR 寄存器的值。

定时中断发生器：

我们要实现一个 MMIO 的定时中断发生器——Timer。

MMIO 简单来说就是：该外设用来和 CPU 交互的寄存器是与内存一起编址的，所以 CPU 可以通过访存指令（load/store）来修改这些寄存器的值，从而达到 CPU 和外设交互的目的。

而 MMIO 的实现目前在没有实现总线的情况下，使用多路选择器即可达到目的。原因主要是我们把取指令的操作和 load/store 访存操作分开了，让 Memory 有单独的一个通路进行取指令操作。因此我们的模型还是一个 CPU 对多个外围设备，不会出现 CPU 的取指操作与访存操作冲突争抢外设的情况。

所以我们CPU发出的逻辑地址要发送到哪个设备，就由逻辑地址的高位作为外围设备的位选信号即可，低位则用于设备内部的寻址。

此外还有定时中断发生器的内部逻辑：两个控制寄存器 `enable` 寄存器和 `limit` 寄存器。

- `enable` 寄存器用来控制定时中断发生器的使能，为 `false` 则不产生中断， 映射到地址空间的逻辑地址为 0x80000008。
- `limit` 寄存器用来控制定时器的中断发生间隔，映射到地址空间的逻辑地址为 0x80000004。中断发生器内部有一个加一计数器，当计数器的值到达 `limit` 为标准的界限时，定时器会发生一次中断信号（`enable` 使能情况下）。注：产生中断信号的时长没有太大关系，但是至少应该大于一个 CPU 时钟周期，确保 CPU 能够正确捕捉到该信号即可。

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

所有的CSR指令都会对CSR进行读——改——写操作。CSR指令中共有12位（20-31位）用来指示被“读改写”的是哪一个寄存器。

CSRRW指令可以原子性地读写CSR，先将指定的CSR的值存入目的通用寄存器，再将源通用寄存器的值存入目的CSR。CSRRWI和CSRRW指令类似，只不过将32位的源寄存器替换为了5位的无符号立即数，5位的立即数在赋值到CSR之前会在立即数前补0，使其整体为32位

CSRRS指令可以原子性地读取并置位CSR的指定位（Atomic Read and Set Bits in CSR）。此指令会读取指定CSR的值到目的寄存器；然后将源通用寄存器作为掩码，写bit 1到CSR——也就是源通用寄存器中所有为1的位所对应的CSR中的那一位都会被写1（如果那一位可写的话），源通用寄存器中所有为0的位所对应的CSR中的位都保持原来的值。CSRRSI指令和CSRRS类似，只不过将源寄存器替换为了5位的无符号立即数，0-拓展方式和CSRRWI一样

CSRRC指令可以原子性地读取并清零CSR的指定位（Atomic Read and Clear Bits in CSR）。此指令和CSRRS指令类似，需要注意的是源通用寄存器中**为1**的位在CSR中所对应的位会被**清零**。CSRRCI和CSRRC类似，只不过将源寄存器替换为了5位的无符号立即数，0-拓展方式和CSRRWI一样

```
// lab2(CLINTCSR)
val uimm = io.instruction(19, 15)
io.csr_reg_write_data := MuxLookup(
  funct3,
  0.U,
  IndexedSeq(
    InstructionsTypeCSR.csrrw -> io.reg1_data,
    InstructionsTypeCSR.csrrc -> (io.csr_reg_read_data & (~io.reg1_data).asUInt),
    InstructionsTypeCSR.csrrs -> (io.csr_reg_read_data | io.reg1_data),
    InstructionsTypeCSR.csrrwi -> (0.U(27.W) ## uimm),
    InstructionsTypeCSR.csrrci -> (io.csr_reg_read_data & (~(0.U(27.W) ## uimm)).asUInt),
    InstructionsTypeCSR.csrrsi -> (io.csr_reg_read_data | 0.U(27.W) ## uimm),
  )
)
```

根据以上知识，我们可以写出CSR寄存器组的空缺代码

```
//lab2(CLINTCSR)
io.clint_access_bundle.mstatus :=  Mux(io.reg_write_enable_id && io.reg_read_address_id === CSRRegister.MSTATUS, io.reg_write_data_ex, mstatus)
io.clint_access_bundle.mtvec := Mux(io.reg_write_enable_id && io.reg_read_address_id === CSRRegister.MTVEC, io.reg_write_data_ex, mtvec)
io.clint_access_bundle.mcause := Mux(io.reg_write_enable_id && io.reg_read_address_id === CSRRegister.MCAUSE, io.reg_write_data_ex, mcause)
io.clint_access_bundle.mepc := Mux(io.reg_write_enable_id && io.reg_read_address_id === CSRRegister.MEPC, io.reg_write_data_ex, mepc)
//lab2end
```

注意根据输入使能与当下的指令判断写入的是什么数据。

CLINT操作：

1、无中断时，写使能和中断使能为低电平

2、mstaus第四位M模式下全局中断

3、中断信号有效时，mstatus修改值，mepc存储跳转指令的跳转地址或当前pc+4，修改CSR寄存器的值，写使能赋值为高电平，中断使能赋值为高电平，从mtevc中获取中断处理程序的地址

4、处理完中断程序返回时，恢复msttaus的值，mepc、mcause、写使能、中断使能不变，下一个指令地址为中断前指令的下一条指令的地址

Timer操作：

```
io.bundle.read_data := MuxLookup(
  io.bundle.address,
  0.U,
  IndexedSeq(
    0x4.U -> limit,
    0x8.U ->enabled.asUInt,
  )
)
```

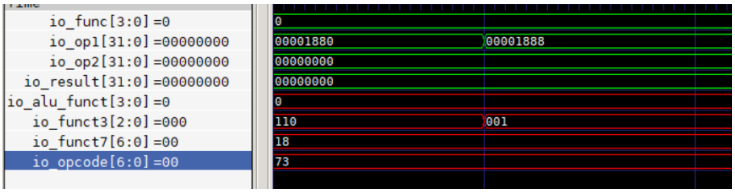
```
when(io.bundle.write_enable){
  when(io.bundle.address === 0x4.U){
    limit := io.bundle.write_data
    count := 0.U
  }.elsewhen(io.bundle.address === 0x8.U){
    enabled := io.bundle.write_data =/= 0.U
  }
}
```

在bundle中读取相应的值，并在timer里写。

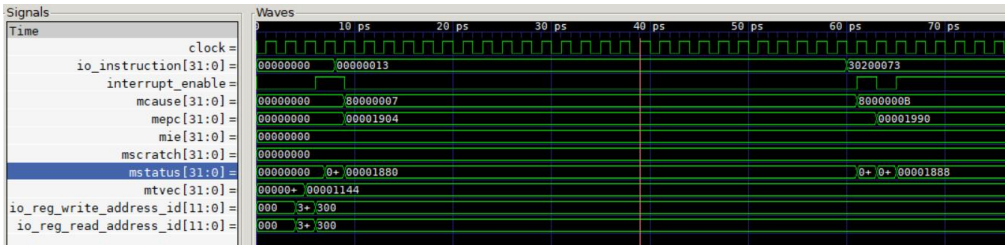
```
io.signal_interrupt := enabled && (count >= (limit - 10.U))
when(count >= limit){
  count := 0.U
}.otherwise{
  count := count + 1.U
}
```

判断是否满足中断条件，在timer计数到特定时间后触发中断。

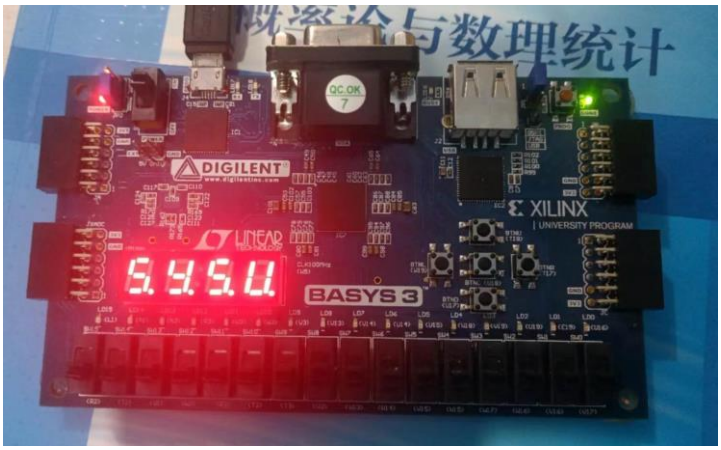
测试波形如下：



如图所示，给出执行阶段的ALU相关信号以及funct3、funct7、opcode字段，符合预期。



CLINT：CSR的值，id的读写地址，阻塞地址都没有问题



六. 实验心得

中断部分的逻辑较为复杂，导致分析实验原理和已有代码的过程十分漫长。在阅读实验文档的过程中遇到许多不懂的问题，于是在网上搜索，逐个击破。

经过漫长的学习过程，大概明白了RISCV中断的原理，于是开始阅读代码，将代码的每个部分与我学习到的原理对应起来理解，如执行阶段的各种CSR指令格式。

在CLINT中，保存哪些信息到对应的CSR寄存器中是一个关键的问题，在这部分我思考了很久。写CLINT空缺代码时，我参考了网络上关于CLINT的阐述。网上资料有很多，质量参差不齐，这个查阅资料的过程锻炼了我筛选信息和整合信息的能力。