P7 - 简单的MIPS微系统 - 设计文档

思考题

1. 请查阅相关资料，说明鼠标和键盘的输入信号是如何被 CPU 知晓的?

   简言之，是向CPU传递对应的中断信号；

   复杂一点的话，不妨参阅PS/2标准是怎么实现的，可以参考：
   https://nju-projectn.github.io/dlco-lecture-note/exp/07.html#ps-2
   （要是CO跟别人家的CO一样该多好，23333）
2. 请思考为什么我们的 CPU 处理中断异常必须是已经指定好的地址？如果你的 CPU 支持用户自定义入口地址，即处理中断异常的程序由用户提供，其还能提供我们所希望的功能吗？如果可以，请说明这样可能会出现什么问题？否则举例说明。（假设用户提供的中断处理程序合法）

   对于根据16位立即数（branch），26位立即数（jump）的跳转，我们执行的都是相对跳转，对跳转地址的范围有限制；此时，若程序规模足够大，则会不可避免地出现目标跳转地址，即我们的中断程序地址，超出跳转范围的问题。（不过后者的可能性不大）

   那对于32位立即数跳转（e.g. jal）呢？确实是绝对跳转，但这不就是根据指定好的地址跳转了吗?

   用户定义当然是可以的（假设用户编写的异常处理程序十分完备）。但我们此时的用户自定异常程序，是存在IM里的；由此会引起程序正常执行到处理程序入口前，顺序进入异常程序的问题（e.g. 0x417c -> 0x4180），这是我们不希望的。

   更何况，这占据了本就有限的IM空间（笑）
3. 为何与外设通信需要 Bridge?

   我们通过将外设的寄存器映射到内存空间内，从而实现对外设的访问。

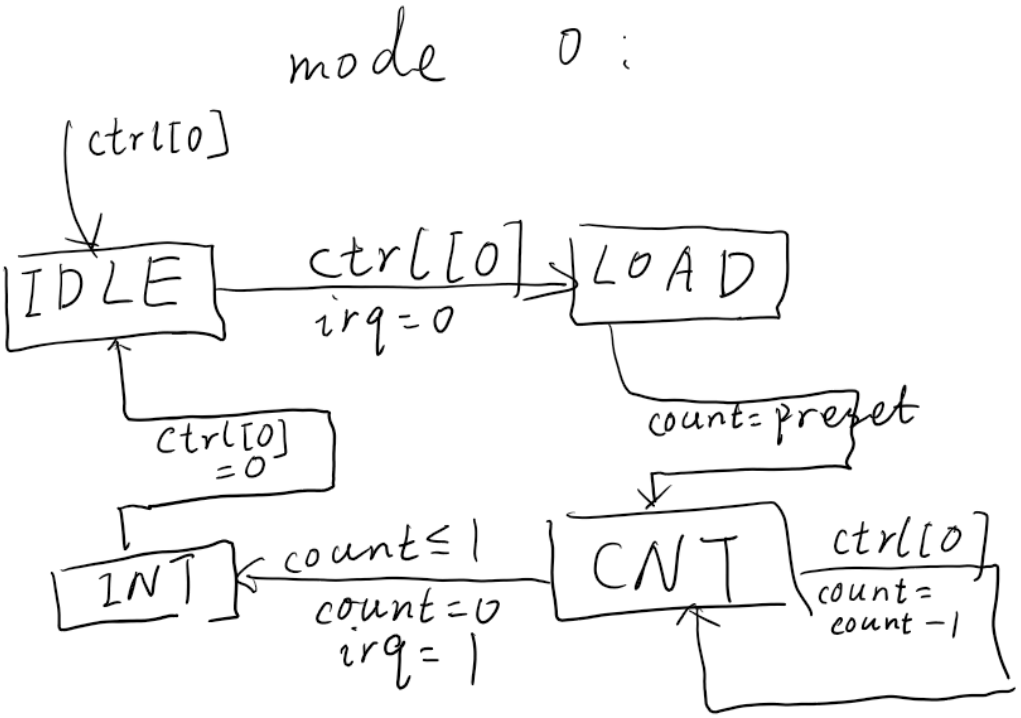
   显然，外设对应的“内存地址”跟DM的内存地址范围不一样，不同外设的地址范围也不一样。

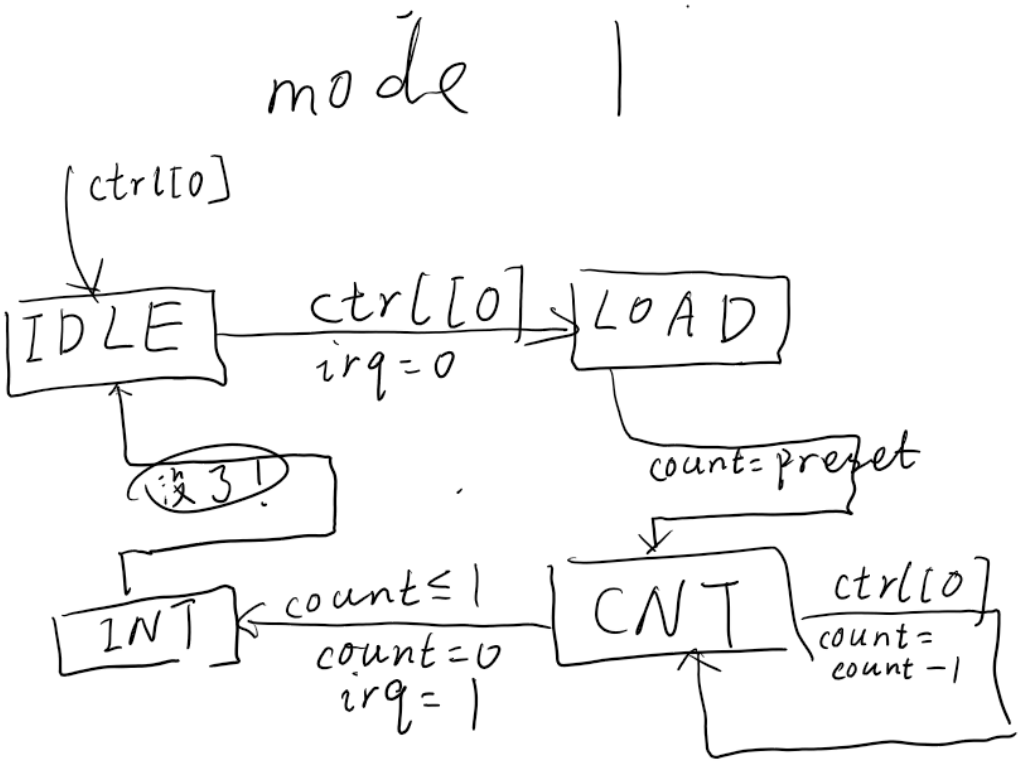
   因此，我们便需要Bridge，通过地址找到当前操作的究竟是DM还是哪一个外设，同时产生对应的控制信号。
4. 请阅读官方提供的定时器源代码，阐述两种中断模式的异同，并分别针对每一种模式绘制状态移图。

模式0要求手动对Enable进行操作来重启计时操作，**在重启前持续产生中断信号**，与中断发生器的逻辑颇为相似；

模式1则是持续工作，在倒数到1时抛出一周期的中断信号，随后从头再来；产生的中断信号是**周期性脉冲信号。**

二者相同之处：计数逻辑，和在数到何值时产生中断。





5. 倘若中断信号流入的时候，在检测宏观 PC 的一级如果是一条空泡（你的 CPU 该级所有信息均为空）指令，此时会发生什么问题？在此例基础上请思考：在 P7 中，清空流水线产生的空泡指令应该保留原指令的哪些信息？

写入VPC的地址是无效地址，执行 `eret` 时会出问题。

这部分不妨直接把PC的传播逻辑完整解释一遍：

- 在重置时，各级传递的PC复位到0x3000；
- 在 `Req` 有效时，各级传递的PC赋值为处理程序地址；
- 在发生暂停时：对CD间寄存器，保持PC值不变；对DE间寄存器，**按正常逻辑，传递来自CD级的PC**
- 正常情况：各级传递来自之前一级的PC。

除了各级的PC外，**还要注意CD，DE级处的延迟槽指示，只在重置和进入处理程序时重置为0。**

6. 为什么 `jalr` 指令为什么不能写成 `jalr $31, $31`？

   若在 `jalr` 的延迟槽指令处发生了中断，我们在执行完处理程序后，会回到 `jalr` 上；此时，`$31` 已被改写，这会导致 `jalr` 的行为异常。

   实在想这么干也可以，改改设计即可：对 `link` 类指令，在GRF的使能端口前，根据是否产生事件进行一下判断即可。

7. [P7 选做] 请详细描述你的测试方案及测试数据构造策略。
   我在这一部分，只针对P7添加的功能做了简单的测试。
   - 中断：利用课程组 `testbench` 中在0x3010处的中断进行测试。在其之前手动用 `mtc0` 配置CP0_SR，使得中断能被接收。
   - 非延迟槽异常：我在这一部分，主要对AdEL（除PC相关异常）， AdES与Syscall进行了测试。前两者通过构造加法溢出（DM地址溢出，ALU加法溢出），未对齐地址（DM），向Timer执行非Word指令，向计时器Count寄存器执行store指令，地址范围外地址（DM,Bridge），而后者直接syscall一下即可...
   - 延迟槽异常：在一条进行跳转的语句后，构造上面的异常。
   - eret的跳转：在eret后跟了数条指令，通过查看输出结果+波形的方法检查是否立即跳转。
   - 暂停时中断：在0x3010前，利用add + beq构造出0x3010时的暂停。这么构造，是因为配置中断已经耗掉了两条指令，欲在3010处产生中断，必须在2条指令中产生暂停；上述方法最方便。
   - 众所周知，.ktext没法直接导出；为此，我们通过将导出的.text（ `code.txt` ）用 `nop` 填充到0x4180（4480行），再在其后紧接.ktext对应处理程序的机器码。在测试时，我们最开始不通过 `mtc0` 配置中断，这样在0x3010时便不会进入中断；在处理程序中也不出现 `mtc0`，以确保中断不会产生。**用户程序最后以 `beq` 死循环结束，避免误入处理程序。** 这样就可以利用MARS，根据已编写的测试程序，自动测试异常。

设计草稿

PC值的流水

- 在重置时，各级传递的PC复位到0x3000；
- 在 `Req` 有效时，各级传递的PC赋值为处理程序地址；
- 在发生暂停时：对CD间寄存器，保持PC值不变；对DE间寄存器，**按正常逻辑，传递来自CD级的PC**
- 正常情况：各级传递来自之前一级的PC。

CP0 与异常

CP0位置，进入异常相关控制

考虑到目前要求的所有异常都在M级及之前生成，我们决定将CP0置于M级。

根据**高内聚，低耦合**的要求，我们利用先前的NPC设计，进行异常处理程序的跳转。具体实现是，在NPC的IMM32，NPCOp，NPCCond前加入按异常跳转使能进行选择的MUX；IMM32指向处理程序的地址，NPCOp指向按IMM32跳转，NPCCond指向无条件；

同时，**我们需要清除流水线寄存器；在发生异常的时候，直接无脑清空CD,DE,EM,MW三级流水线寄存器即可。**

> 我们之前推出来不清CD，但为什么最后还是决定要清呢？这跟我们PC+NPC的实现方法有关；PC内含一个寄存器。我们此时指示NPC将Handler传给PC，PC会先让**我们已经决定不再执行的指令继续流水，再写入我们的handler**，与我们的预期不符；要取消这条继续流水的指令，自然要清空CD。

当异常信号传入CP0后，CP0记下产生异常的指令PC值，存入EPC中，再按照异常类型对应进行跳转，将对应的处理程序地址传给NPC。

注意：**写入EPC的地址，跟是否是延迟槽指令有关！否则跳转后的指令无法正确执行！**

- 对于正常指令，我们直接写入PC即可；
- 而对于延迟槽指令，我们则需要写入PC-4，指向延迟槽之前的跳转语句，从而进行正确的跳转！

在出现异常/中断时，我们需要对MDU，DM/Bridge的使能进行修改，防止事件产生时，对应指令对DM/Bridge的写入无效，且该指令之后的指令不对MDU的状态进行更改：

- MDU：对Start，Ovrd两个信号输入前加入与门，将原有信号与 `Req` 进行与操作；MDU的实现请见P6设计文档。
- DM/Bridge：对写使能加入与门，将原有信号与 `Req` 进行与操作。

eret的跳转

对于`eret`的无延迟槽跳转，由于我们的解码在D级进行，我们自然不可能依靠D级控制器进行解码，再进行跳转的决定，不然一定会导致`eret`的`I+1`指令进入流水。

这会引起什么问题呢？ 假设`eret`的下一条指令时产生了中断，且我们采用了清空延迟槽的思路进行`eret`的跳转处理。此时，**CP0的VPC将被复写**；按照处理流程处理完中断后，我们回到了`eret`，那么，`eret`向哪跳转呢？**完辣！死循环了！！！**

因此，我们在C级，根据IFU读出的指令，直接进行`eret`的解码；若是，则指示NPC直接引用CP0读出的VPC进行跳转，从而规避这个问题。

eret的转发

这个问题的成因，是`mtc0`类指令有可能对CP0的VPC进行复写，从而导致`eret`的跳转出现异常。

注意到，在CP0以外，只有`eret`指令需要直接读取CP0内容，**且只读EPC（CRF[14]）**；且当前只有`mtc0`指令对CP0进行读写。分析可知，我们只需要构造转发即可，不用暂停...?

要暂停！考虑`lw,mtc0,eret`这个序列，**为了能够从流水线寄存器转发，我们不得不暂停两个周期！！**

数据来源都是转发后的GRF_RD2。

中断 / 支持外设

中断其实没什么好说的。

对于Bridge，把Timer例化两个，对应Timer1，Timer0，再在Brigde的端口处接好中断发生器要接的线，最后把这三者的中断接入HwInt，没了...

对于CP0，把HwInt读进来，若符合终端条件则产生中断，即可。

对于DM与外设寄存器的读写，我们统一交由Bridge进行管理，由Bridge根据地址范围，对应生成写使能信号。

此时我们可以保留原有的DM写使能，将之视为桥整体的写使能，对外设的写入也视为对Memory的Load操作；细分的写使能交给Bridge就可以了。

你问外设寄存器怎么读写？它被视为内存，也被分配了一个内存地址。

~~Timer你就把它当作插USB口上的一个神奇设备就可以了，别想复杂...~~

Timer是为了执行多任务而实现的，系统内部固有的中断发生器；而Interrupt Generator才是我们虚线划掉部分内容提到的设备。

也能注意到，我们需要从桥，向CP0发送中断信号；根据教程里面的图，Timer的线是直接向CP0接的，但我觉得不妥；既然外设交互交给了Bridge进行，这一部分也应该由Bridge交给CP0才对。

模块定义

直接RTFSC吧，没人看，且不想写，明显源码更直观...

相较P6，增加了Bridge，CP0，修改了CTRL，并在CPU内添加了对应的控制信号与连接

测试方案

见思考题有关部分。

测试程序如下：

```
# AdEL
        # lw align
        ori $2,$0,2
        lw $5,0($2)
        # lh align
        ori $2,$0,1
        lh $5,0($2)
        # lh,lb to Timer
        ori $2,$0,0x7F00
        lh $5,0($2)
        lb $5,0($2)
        ori $2,$0,0x7F10
        lh $5,0($2)
        lb $5,0($2)
        # DM_ADDR Overflow
        ori $2,$0,0xffffffff
        ori $3,$0,2
        lw $5,0($2)
        lh $5,0($2)
        lb $5,0($2)
        # Out Of Range
        ori $2,$0,0x7f24
        lw $5,0($2)
        lh $5,0($2)
        lb $5,0($2)
# AdES
```

```
        # sw align
        ori $2,$0,2
        sw $5,0($2)
        # sh align
        ori $2,$0,1
        sh $5,0($2)
        # sh,sb to Timer
        ori $2,$0,0x7F00
        sh $5,0($2)
        sb $5,0($2)
        ori $2,$0,0x7F10
        sh $5,0($2)
        sb $5,0($2)
        # DM_ADDR Overflow
        ori $2,$0,0xffffffff
        ori $3,$0,2
        sw $5,0($2)
        sh $5,0($2)
        sb $5,0($2)
        # Save to count
        ori $2,$0,0x7f08
        sw $5,0($2)
        sh $5,0($2)
        sb $5,0($2)
        ori $2,$0,0x7f18
        sw $5,0($2)
        sh $5,0($2)
        sb $5,0($2)
        # Out Of Range
        ori $2,$0,0x7f25
        sw $5,0($2)
        sh $5,0($2)
        sb $5,0($2)

# Syscall
        ori $2,$0,32
        syscall
# Ov
        ori $2,$0,0x7fffffff
        ori $3,$0,1
```

```
        add $5,$2,$3
        addi $5,$2,1

        ori $2,$0,0xffffffff
        ori $3,$0,0xffffffff
        sub $5,$2,$3
# RI
        j next
        nop
        next:
        ori $2,$0,0xffffffff

# Special: PC associated
        ori $20,$0,1
        ori $21,$0,0x00002fff
        jal next_1
        nop
        next_1:
        jr $21
        nop
        ori $20,$0,0

        ori $20,$0,1
        ori $21,$0,0x00003001
        jal next_2
        nop
        next_2:
        jr $21
        nop
        ori $20,$0,0

ori $31,$0,0xffffffff
end:
beq $0,$0,end

.ktext 0x4180
        mfc0 $6,$12
        mfc0 $7,$13
        mfc0 $8,$14
        ori $2,$0,0
```

```
mfc0 $k0, $13
ori $k1, $0, 0x7c
and $k0, $k0, $k1

ori $25,$0,0x00000020
beq $k0,$25,handle_syscall
nop
ori $25,$0,0x00000028
beq $k0,$25,handle_syscall
nop
bne $20,$0,handle_pc
nop

return:
mfc0 $k0, $14
addi $k0, $k0, 4
sw $k0,0($0)
lw $18,0($0)
mtc0 $18, $14
eret

handle_syscall:
mfc0 $k0, $14
addi $k0, $k0, 4
mtc0 $k0, $14
beq $0,$0,return
nop

handle_pc:
add $21,$31,8
mtc0 $31,$14
beq $0,$0,return
nop
```