6 SC CPU 中断/异常处理设计

英语

1. *former* 前者
2. *latter* 后者
3. *be tied to* 被束缚
4. *reside* 驻留
5. *vectored* 矢量的
6. *pending* 待定的、未响应的
7. *correspond to* 对应
8. *drain* 菊花
1. *alter* 改变
2. *detect* 检测
3. *suspend* 悬挂
4. *polled* 轮询的
5. *hardwired* 硬连接的
6. *nesting* 嵌套 *nest* 巢
7. *simultaneously* 同时地

6.1 异常和中断

6.1.1 异常和中断的类型

一个**同步的内部异常**是由当前正在执行的程序中的一条指令的操作引起的。它是同步的，因为当程序运行时，假设其他情况是相同的，那么相同的异常将发生在程序中的同一位置。例如 TLB 不命中、段异常、算术溢出、除 0 等都是同步的内部异常

> *A synchronous internal exception is caused by the actions of an instruction in the currently executing pro gram. It is synchronous because the same exception will occur at the same place in the program whenever the program runs, assuming other circumstances are the same*

一个**异步的外部中断**是被额外的硬件设备生成的。它是异步的，因为它不与当前正在执行的程序的执行绑定，而且可以随时发生。例如键盘、鼠标输入、定时器等都是异步的外部中断

> *An asynchronous external interrupt is generated by external hardware devices. It is asynchronous because it is not tied to the execution of the currently executing program and can occur at any time.*

同步是内部的，异步是外部的。同步是只要其他情况相同，那么不管在什么地方执行这个指令，都会在同一个位置发生异常。而异步是发生中断的地方不确定，可以随时由外部设备发出中断请求

当一个异常或者中断发生时，CPU 会挂起当前正在执行的程序，然后转移控制权到一个事先准备好的异常或者中断服务程序去处理中断或者异常。这一过程是被称作“中断或者异常的确认”，这个事先准备好的程序也被称作“中断或者异常处理程序”，是提前驻留在操作系统内核（外设的中断处理程序通过驱动加入内核）中的。**中断/异常处理程序执行结束后，可能返回继续执行被挂起的现行程序，也可能取消挂起的现行程序，将控制权返回给操作系统**

> *When an exception or an interrupt occurs, the CPU suspends the current program and transfers control to a pre-prepared exception or interrupt service routine to deal with the exception or interrupt. This pro cedure is called an exception acknowledge or interrupt acknowledge. The service routine is also known as an exception handler or interrupt handler, which resides in the operating system kernel. Depending on the type of the exception/interrupt, the best result of executing the handler is to resume execution of the suspended program, after doing whatever it needs to, as shown in Figure 6.1.* Some exceptions or interrupts may cancel the continuous execution of the suspended program, display an error message on screen, and return to the operating system. The illegal instruction exception and control_c keystroke interrupt are examples that cancel the execution of the program.

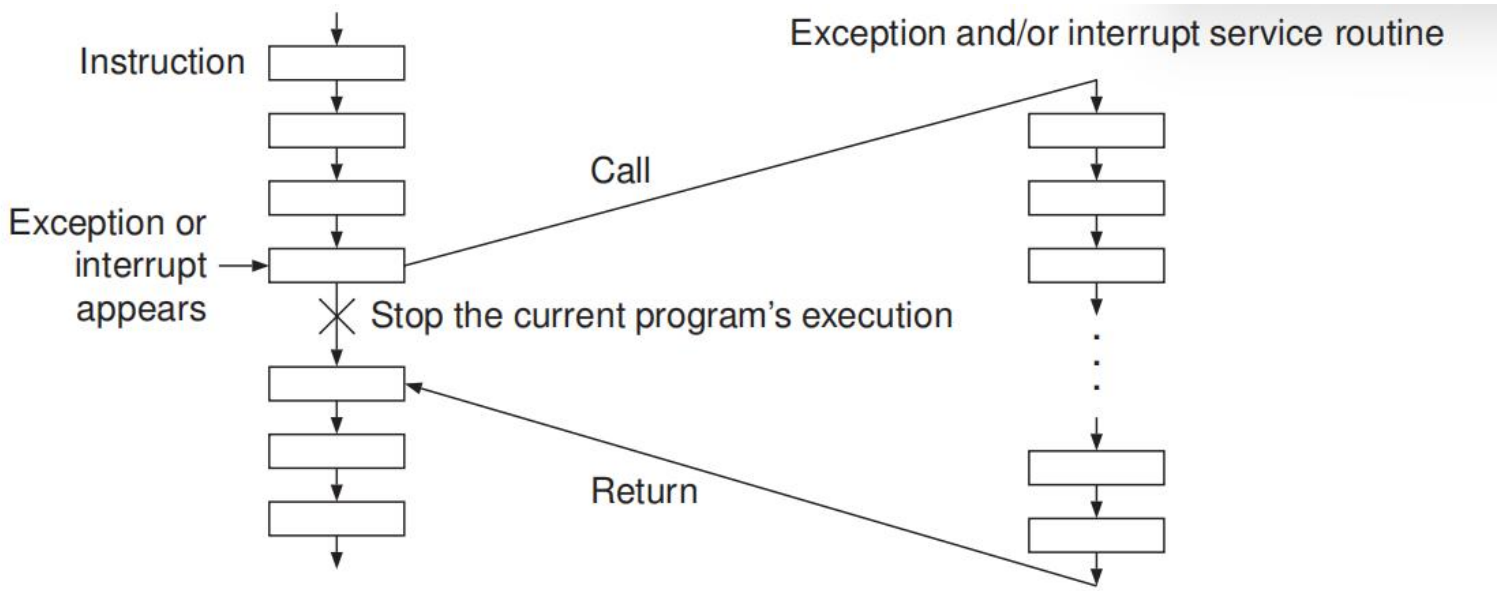


Figure 6.1 Dealing with exceptions and interrupts

6.1.2 轮询中断和矢/向量中断

轮询中断和矢量中断是两种用来将控制权交付给中断/异常处理程序的方式

当一个中断或者异常发生时，CPU 通过将 PC 值设为中断服务程序的入口地址实现控制权的交换。这个中断服务程序的入口地址可以是硬连接的（不可更改的）也可以是被存储在一个 CPU 可以用指令写的特别的寄存器中（可更改的）

> *When an exception or interrupt occurs, the CPU transfers control to the handler by jumping to a fixed address, which is the entry address of the handler. This address can be hardwired (not changeable) or stored in a particular register to which the CPU can write with an instruction (changeable).*

6.1.2.1 轮询中断

CPU 中的一个程序可以从某个地方读取一些信息去确定发生了什么异常或者哪一个设备发出了中断请求。在 MIPS ISA 中，这样的一个地方被定义为 **Cause 寄存器，Cause 寄存器是 0 号协处理器（CP0）的一个寄存器，包含着中断和异常源**，其与中断、异常相关的字段内容如下：

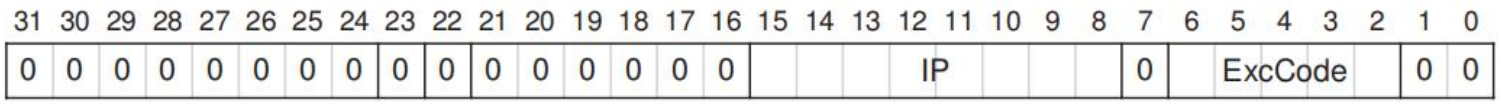


Figure 6.2 Cause register of CP0

1. ExcCode 字段表示异常源，其值序号就是

Table 6.1 ExcCode field of the MIPS Cause register

ExcCode	Mnemonic	Description
0	Int	Interrupt (`IP[7:0]` indicates the interrupt source)
1	Mod	TLB hit on store but memory was not yet modified
2	TLBL	TLB miss on instruction fetch or load
3	TLBS	TLB miss on store
4	AdEL	Address error when fetch or load
5	AdES	Address error when store
6	IBE	Bus error on instruction fetch
7	DBE	Bus error on load or store
8	Sys	Executing system call instruction
9	Bp	Executing break instruction
10	RI	Attempt to execute reserved instruction
11	CpU	Coprocessor is not available
12	Ov	Arithmetic overflow
13	Tr	Executing trap instruction
14	–	Reserved
15	FPE	Floating-point exceptions
16–22	–	Reserved
23	WATCH	Virtual address matches value in Watch register
24	MCheck	TLB multiple match but not consistent
25–29	–	Reserved
30	CacheErr	Cache error
31	–	Reserved

下面的程序可以获得异常处理程序的入口地址：

```
lui   $k0, %hi(jump_table_base) 取高半字        # jump table base address
mfc0  $k1, c0_cause   读cp0中的cause寄存器      # $27 <-- cause
andi  $k1, $k1, 0x7c                           # [6:2]: exccode
add   $k0, $k0, $k1                            # plus offset
lw    $k0, %low(jump_table_base)($k0) 取低半字 # get address from jump table
jr    $k0                                      # jump to the address
```

2. IP 字段表示中断源，当 ExcCode 全 0 时，表示发生中断

获取中断服务程序的入口地址也可以采用上面的方法：

mfc0 读取到 cp0 的 cause 寄存器后，判断是否是中断→andi $t1,$k1,0x7c

如果$t1 是全 0，那么就是发生了中断，得到中断号→andi $k1,$k1,0xff00

再按照表的构建方式得到中断服务程序的入口地址

轮询中断的方法通过将控制转移到一个公共的入口地址，然后在这个公共入口地址上查看中断或者异常的源信息，然后跳转到对应的中断服务程序的入口地址上

> *The polled interrupt method makes the control to be transferred to a common entry address and then to an individual entry address by checking the source information of the exception or interrupt*

6.1.2.2 矢量中断

矢量中断方法通过一个向量[注释1]直接或间接地将控制交给中断服务程序的入口地址，如下图：

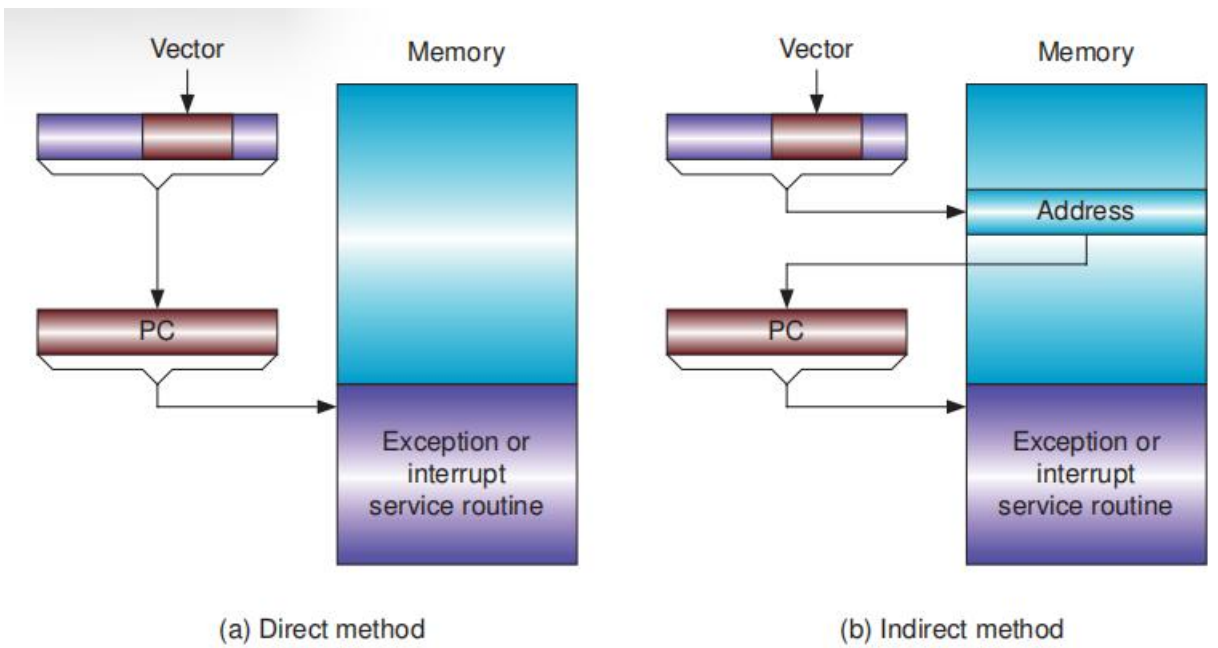


Figure 6.3 Mechanism of vectored exception/interrupt

当一个异常或者中断发生时，硬件会产生一个独一无二的向量。通过在这个向量左右两端加一些合适的值得到了一个地址。这个地址可能就是中断服务程序的入口地址，将其保存在某寄存器中，用 jr 指令写入 pc；也可能是一个存储器的地址，这个存储器单元中保存了中断服务程序的地址，先 load 加载到某寄存器中，然后 jr 写 PC（间接跳转）

> *When an exception or an interrupt occurs, a unique vector is generated by the hardware. By attaching suitable values to the left side and right side of the vector, we get a memory address. This address can be written to the PC, as shown in Figure 6.3(a), so that the control is transferred to the individual entry address directly. SUN SPARC adopts this method. Or, we can use the address to get a value from memory and write the value to the PC, as shown in Figure 6.3(b), so that the control is transferred to the individual entry address indirectly, just like what the Intel x86 does.*

6.1.3 从异常/中断返回

在将控制权交给中断服务程序时，需要保存现行程序的断点[注释2]，然后再更改 PC 为中断服务程序的入口地址

MIPS 采用写断点到特殊的寄存器——EPC 寄存器[注释3]中。对于异常来说，因为引起异常的指令在解决异常后可能需要重新执行，所以 EPC 包含的是所执行引起异常的指令的地址即 PC→EPC；而对于中断来说，中断是在当前指令执行周期结束后操作系统才响应中断，那么就是保存当前执行指令的下一条指令到 EPC 即 PC+4/PC+8[注释4]→EPC

> *In MIPS ISA, an exception program counter (EPC) is prepared for storing the return address. Refer ring to Figure 6.4(a), the EPS contains the address of the instruction that generates the exception (this instruction may need to be executed again, in the case of a TLB miss exception, for instance). In the case of an interrupt, when an external interrupt request comes, the current instruction will be executed before transferring control to the interrupt handler. Therefor , the EPC contains the address of the next instruction, as shown in Figure 6.4(b).*

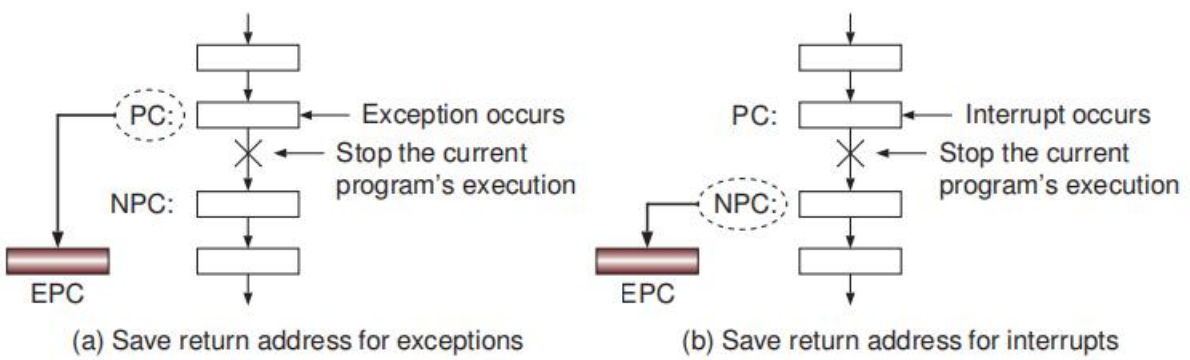


Figure 6.4 EPC contents for exceptions and interrupts

MIPS ISA 使用 eret 指令从中断/异常服务程序返回到被挂起的现行程序，eret 执行的功能是将 **EPC→PC**。但是因为异常是将引起异常的指令地址写 EPC，可能会存在一些异常不需要执行这条引起异常的指令，那么我们就需要对 EPC 手动+4/8，相对应的程序如下：

```
mfc0 $k0,c0_epc ;读取 CP0 协处理器的 epc 寄存器
addi $k0,$k0,4 ;加 4
mtc0 $k0,c0_epc ;写回 epc
eret ;返回断点
```

WebAssembly

6.1.4 中断屏蔽和中断嵌套

因为中断可以在任何时候发出，所以当 CPU 正在处理中断时会有进一步的中断请求。这些中断请求可以通过设置特定的位来实现屏蔽或者允许

> *Because interrupts can occur anytime, further interrupt requests may come when the CPU is dealing with a current interrupt. These requests can be enabled or disabled (masked) by setting or resetting the enable bits of a special control register.Because interrupts can occur anytime, further interrupt requests may come when the CPU is dealing with a current interrupt. These requests can be enabled or disabled (masked) by setting or resetting the enable bits of a special control register.*

在 MIPS 中这样的寄存器称为“*status*”，status 也是协处理器 CP0 的一个寄存器，它的一些字段如下图所示。IM[7:0]的第 i 位对应于第 i 个中断，为 1 表示允许响应当前中断，为 0 时表示禁止。最右侧的 IE 位为总控制，当 IE=0 时所有中断都不能被响应

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				IM					0	0	0	0	0	0	0	IE

Figure 6.5 Status register of CP0

> *MIPS ISA provides such a control register, named the Status register. Some bits of the Status register are shown in Figure 6.5. The ith bit in the field of IM[7:0] corresponds to the ith interrupt: if it is a 0, the interrupt is disabled; otherwise the interrupt is enabled. The rightmost bit, IE, is the master control bit for all the interrupts*

在 MIPS CPU 中默认是不允许中断嵌套的，所以当 CPU 响应一个中断时，会将 IE[7:0]清空

> *MIPS CPU clears all these bits when it acknowledges an interrupt. This prevents interrupt nesting automatically.*

6.1.5 中断优先级

中断优先级解决了当多个中断同时发出中断请求时，先响应哪个中断的问题

确定优先级的方法这里主要介绍两种：菊花链+优先级编码器

优先级编码器

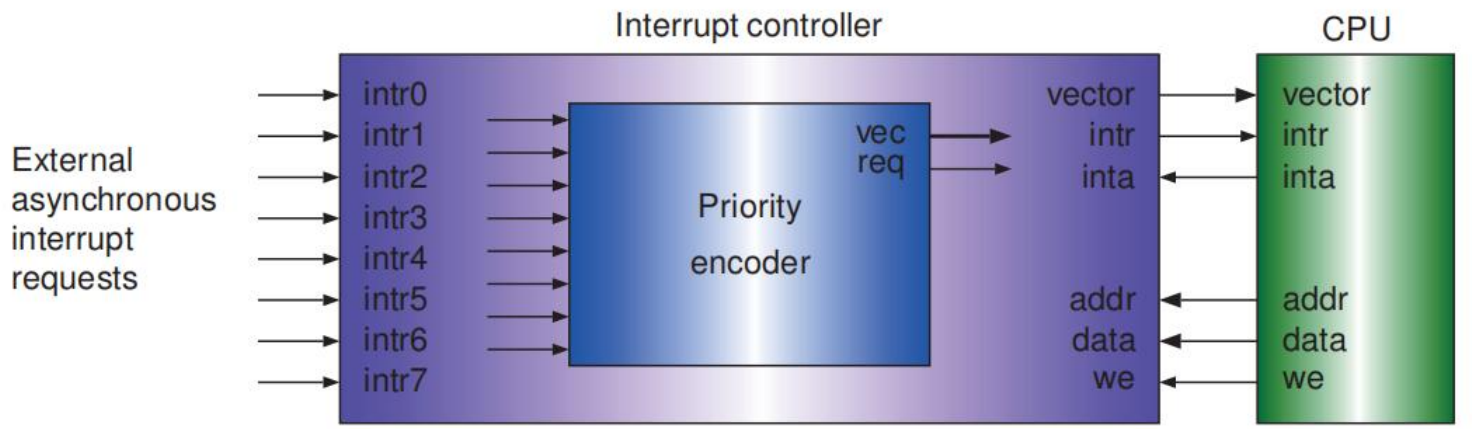


Figure 6.7 Priority interrupt controller

req 是接 CPU 的 intr，作为中断请求信号

vec 是编码结果，是采用 Vectored Iterupt 的 vector 的最右侧的 3 位

菊花链

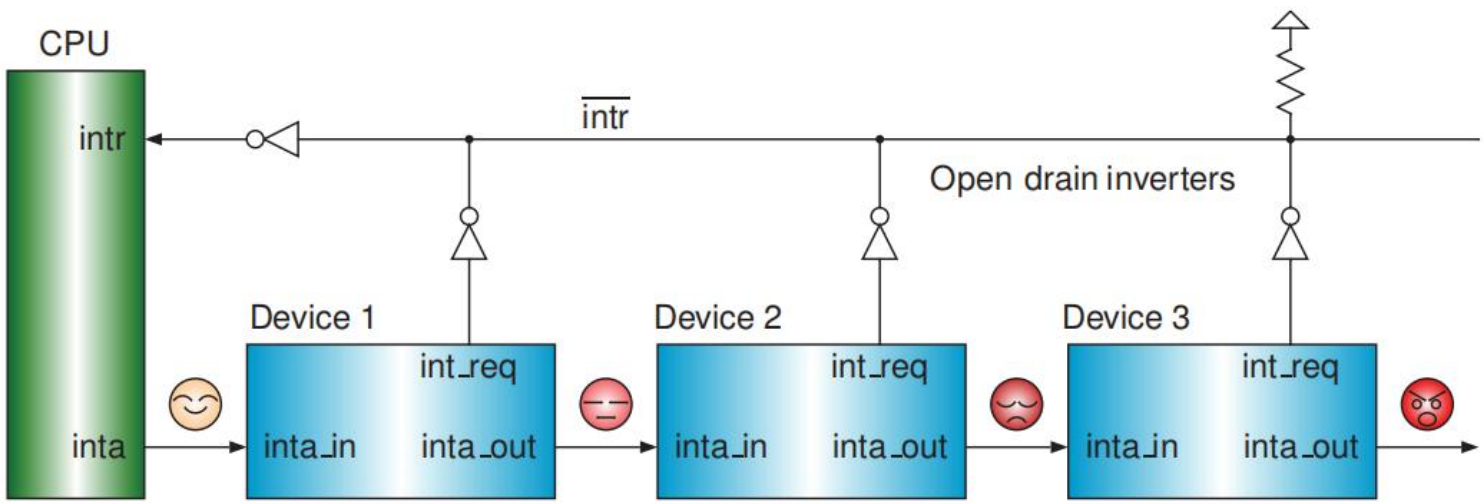


Figure 6.8 Daisy chain for priority interrupts

当设备发出中断请求时，会置 int_req 为高电平，并保持到该设备被服务

所有设备的中断请求 int_req **经过反相器处理后产生**$barintr$**，共同连接到一个反相器接 CPU 的 intr，因此每个设备的信号反相器必须是** open-drain **的或者** open-collector **的**

当 CPU 可以响应中断时，就会使 inta 信号有效，**按设备链连接顺序**输入给设备，当所流经的设备的 int_req 为高电平时，就会锁住 inta，使得其保存在当前设备处，只有当该设备中断处理完成 int_req 为低电平时，inta 才会传送到下一个设备

> *Daisy chain is another way to assign priorities to devices that generate level-sensitive interrupts, as shown in Figure 6.8. Any requesting device can take the interrupt line int_req high, and keep it asserted high until it is serviced. The inverted requests are wire-ORed to generate an active-low interrupt request signal (intr). Note that the three NOT gates in the figure must be the open-drain or open-collector inverters in case their outputs are wired together. The priority is assigned using the interrupt acknowledge signal, inta, generated by the CPU in response to an interrupt request. inta is passed through each device, from the highest priority device first, to the lowest priority device last. If device 1 generated the interrupt, it will block the inta. Otherwise, it will pass inta on to the next device in the chain. Other devices follow the same procedure.*

open-drain 反相器实现

1. 使用 opendrn 模块实现

先对每一个输入使用 not 模块取反，然后构建同样多的 opndrn 模块对所有的反相结果集成输出

```
module openDrain_invetorByopenDrain(
    input a,b,c,
    output r
```

```
);
    wire temp1,temp2,temp3;
    not not_1(temp1,a);
    not not_2(temp2,b);
    not not_3(temp3,c);

    opndrn o1(.in(temp1),.out(r));
    opndrn o2(.in(temp2),.out(r));
    opndrn o3(.in(temp3),.out(r));
endmodule
```

Verilog

2. 使用三态门

当三态门的控制端为 1 时，输出=输入；否则输出为 z。三态门模块定义为 bufif1(out,in,ctl)

构建等同设备数量个三态门，每个三态门的输入端为 0，控制端为当前设备的 int_req，输出端集成

```
module openDrain_invetorBytrigate(
    input a,b,c,
    output r
);
    bufif1 o1(r,0,a);
    bufif1 o2(r,0,b);
    bufif1 o3(r,0,c);
endmodule
```

Verilog

3. 使用相同于三态门的条件表达式

声明输出为 tri 型，使用多个 assign 对输出赋值：int1_req?0;1'bz;

```
module openDrain_invertorByBehav(
    input a,b,c,
    output tri r
);
```

```
    assign r=a?0:1'bz;
    assign r=b?0:1'bz;
    assign r=c?0:1'bz;
endmodule
```
Verilog

4. 对 3 的优化

```
//声明 r 为 wire 型
assign r=(a|b|c)?0:1'bz;
```
Verilog

6.2 使用异常/中断机制设计 CPU

因为 Yamin Li 的这本书并不是实现所有的 MIPS 指令，因此我们简化了中断和异常的处理器机制，一些实现也是不兼容于 MIPS 架构的。我们做以下假设以方便实现：

1. 系统中仅仅只有一个外部的异步中断
2. CPU 只处理三种异常：算术溢出、系统调用和未实现的指令
3. **对于异常来说，是写产生异常的指令的地址到 PC**

   对于中断来说，是写下一条指令的地址到 PC
4. 采用轮询中断的方式确定中断服务程序的地址，中断/异常程序的入口地址是 **0x00000008**
5. 默认不允许中断/异常嵌套，当中断响应时，左移 4 位状态寄存器保存状态寄存器内容
6. 当中断返回时右移状态寄存器 4 位恢复状态寄存器内容

> *1. There is only one external asynchronous interrupt request.*
>
> *2. The CPU deals with only three exceptions: arithmetic signed overflow, unimplemented instruction, and system call.*
>
> *3. For exceptions, the address of the current instruction that generates the exception is saved to EPC; for interrupt, the address of the next instruction is saved to EPC.*
>
> *4. The polled interrupt is adopted. The entry address of the interrupt/exception handler is 0x00000008.*
>
> *5. In response to an interrupt or an exception, the content of the Status register is shifted to the left by 4 bits in order to save previous settings of the Status register and disable further interrupts.*

6. When returning from the interrupt/exception handler, the content of the Status register is shifted to the right by 4 bits in order to restore the previous settings of the Status register.

6.2.1 中断/异常处理程序和相关的寄存器

基于 1-6 的假设，当 CPU 响应中断/异常时，有以下事项需要同时去做：

1. 硬件更新 Cause 寄存器的 ExcCode 位，以跳转到服务程序的入口地址
2. 左移 Status 寄存器 4 位，屏蔽其他中断/异常
3. 保存地址到 EPC——当前和下一个
4. 跳转到处理程序的入口地址

也基于我们的假设，Cause、Status、EPC 寄存器的格式如下：

Cause 寄存器的 ExcCode 字段 0、1，2，3 分别表示中断、系统调用、未实现的指令、算术溢出

Status 寄存器的 IM 字段，IM[0]、IM[1]、IM[2]、IM[3]分别与中断、系统调用、未实现的指令和算术溢出相关。S 字段存储响应服务后的 IM 字段

EPC 寄存器为 32 位，格式不变

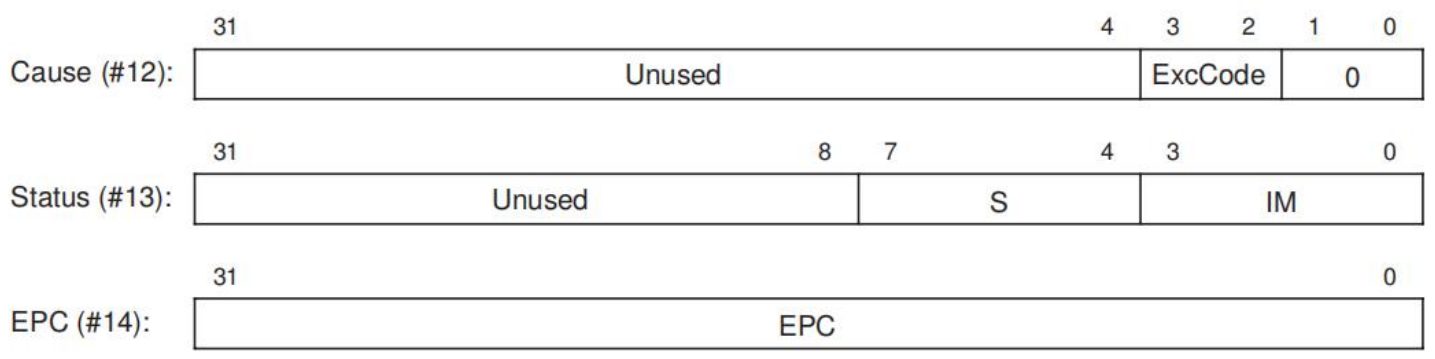


Figure 6.9 Three registers related to exceptions and interrupts

6.2.2 和中断/异常相关的指令

任何指令的执行过程中都可能发生中断，系统调用指令会发生系统调用异常，而未实现的指令是指那些“在 ISA 中是合法的，但是 CPU 并未实现的”，**而算术溢出则在我们的实现中只有 add、sub、addi 这三条指令会出现算术溢出**[注释 5]

算术溢出指令 add、sub、addi

下表为会出现算术溢出的几种情况：“+” + “+” = “−”，“−” + “−” = “+”，“+” − “−” = “−”，“−” − “+” = “+”

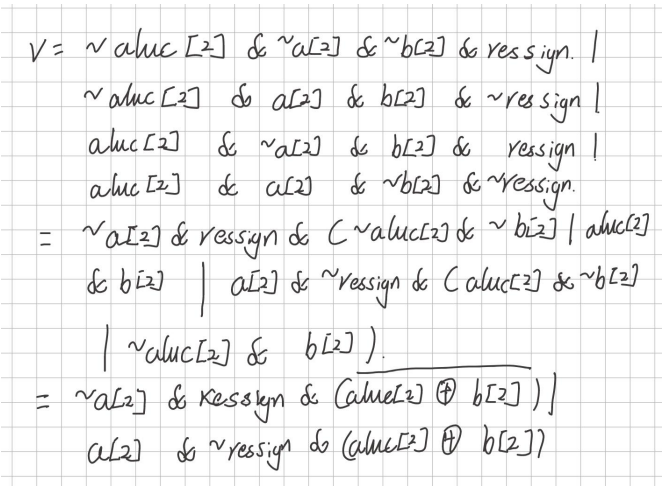
Table 6.3 Overflow on signed add and subtract

	aluc[3:0]	a[31]	b[31]	r[31]	v	Comment
ADD	x 0 0 0	0	0	1	1	Plus + plus, result is minus
ADD	x 0 0 0	1	1	0	1	Minus + minus, result is plus
SUB	x 1 0 0	0	1	1	1	Plus − minus, result is minus
SUB	x 1 0 0	1	0	0	1	Minus − plus, result is plus

则可以列出 v 的逻辑表达为：

$$v = (aluc == 4'bx000)\&(asign\&bsign\&barres\ \ sign|barasign\&barbsign\&res\ \ sign)|$$
$$(aluc == 4'bx100)\&(barasign\&bsign\&res\ \ sign|asign\&barbsign\&barres\ \ sign)$$

化简为：



mfc0 和 mtc0

在中断/异常机制中，也需要使用 Cause、Status、EPC 等寄存器的值，以及写这些寄存器。这就用到了 **mfc0 和 mtc0 指令，这两个指令的格式均是“助记符 rt,rd”，前者“move from CP0”后者“move to CP0”**

syscall 和 eret

syscall 指令实现系统调用，申请操作系统的服务

eret 指令则是从服务程序返回，将 EPC→PC

mfc0、mtc0、eret、syscall 四条指令的格式如下图：

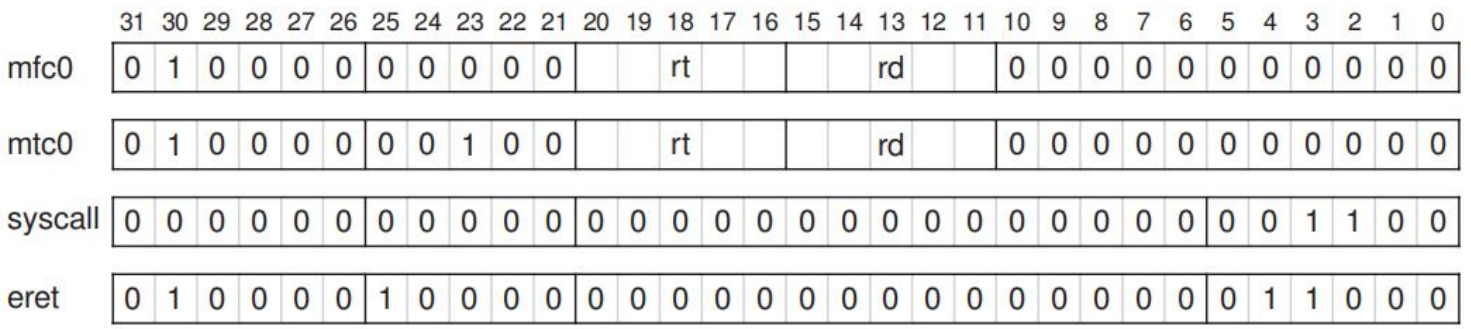


Figure 6.10 Format of exception- and interrupt-related instructions

6.2.3 具有中断/异常处理机制的 SC CPU 示意图

根据假设所实现的具有中断/异常处理机制的 SC CPU 示意图如下，红框内是比 C5 SC CPU 示意图多的部分

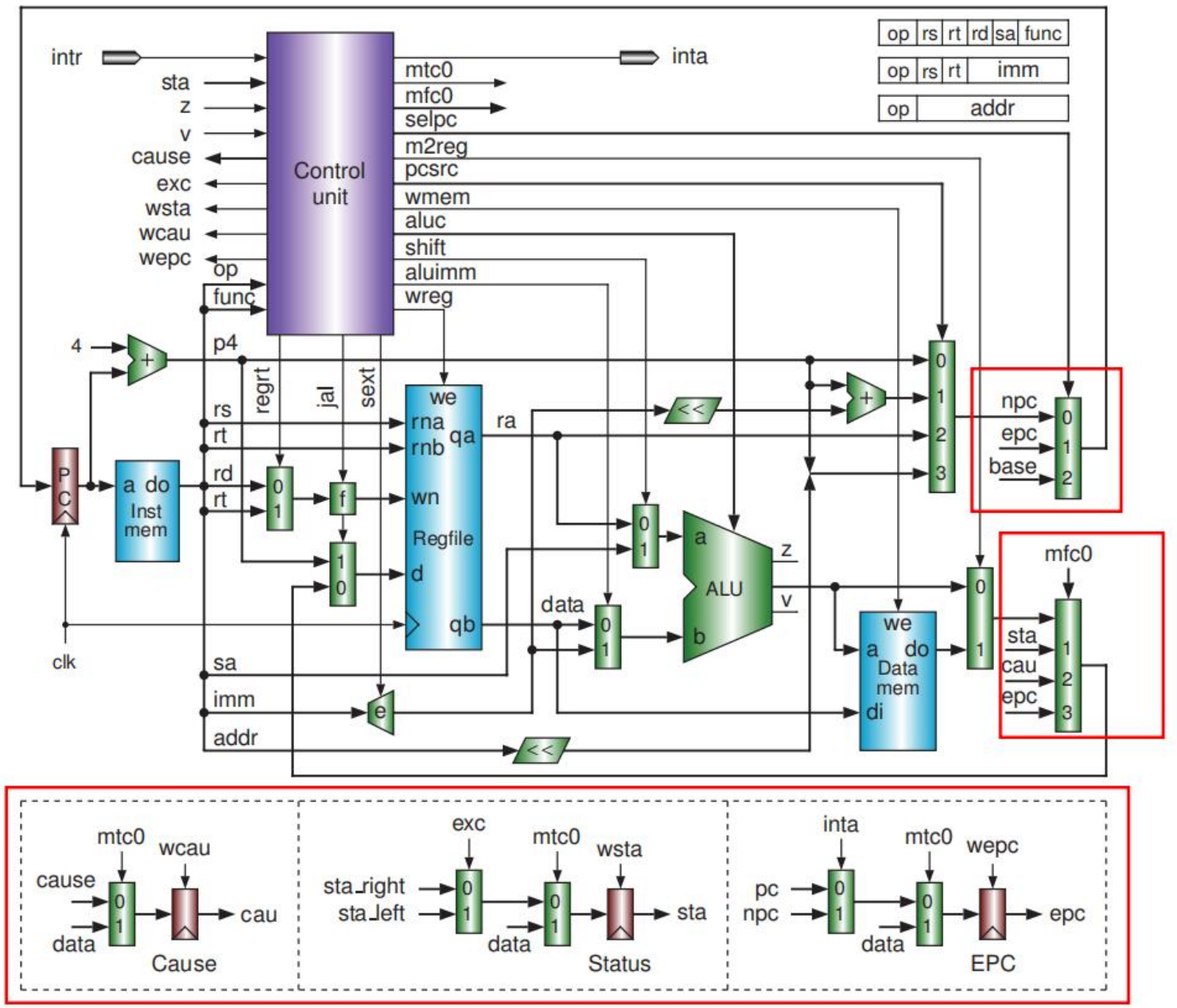


Figure 6.11 Block diagram of the single-cycle CPU with exceptions and interrupts

最下面的红框内是 Cause、Status、EPC 寄存器的实现，这三个寄存器的写信号分别是 wcau、wsta、wepc

mtc0 选择信号选择寄存器中的数据 data 写这三个寄存器，此外也可以写 data

Cause 寄存器可以写 CPU 响应中断时硬件生成的 exccode，也可以写寄存器数据，mtc0 为 1 时写后者

Status 寄存器可以写移位结果，也可以是写寄存器数据，mtc0 为 1 写后者。移位结果也需要 exc 位选择，当响应中断时左移结果写入 sta，中断返回时右移结果写入 sta

EPC 寄存器可以写 PC、NPC 也可以写寄存器数据，mtc0 为 1 时写后者。前者的 PC、NPC 需要 inta 是否是中断响应做选择

右侧下面的 4to1 多选器用于选择写寄存器的数据，需要**在“ALUout 和 memread 的选择结果”和 mfc0 读的“cause、status、epc”中四选一**

右侧上方的 3to1 多选器用于写 PC 的数据选择，需要在**“PCAdder4、BPC、RPC、JPC 的选择结果”和“EPC”和“服务程序基址 Base”中三选一**

6.2.4 Code

这里只列出有改变的模块

CU 控制器模块

CU 控制器模块增加的工作有：

1. 对 eret、mfc0、mtc0、syscall 指令译码

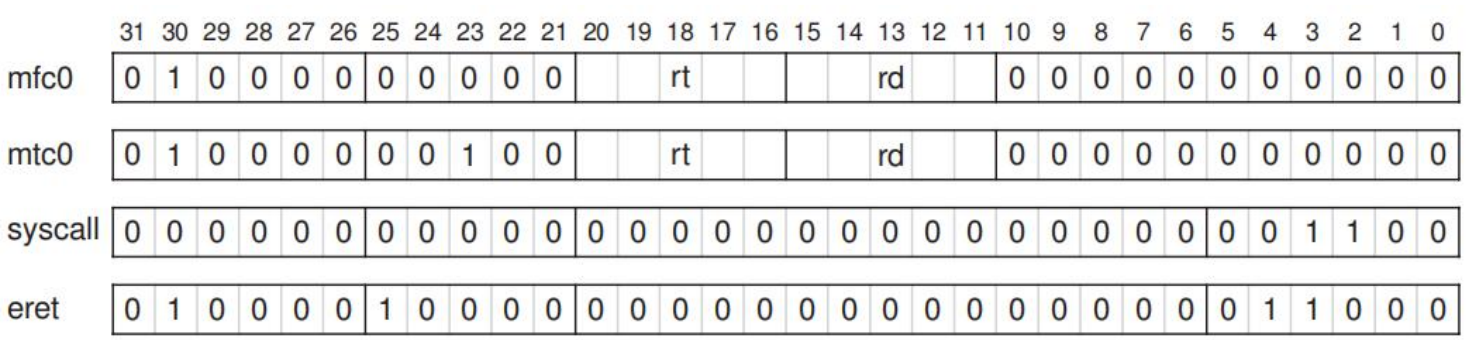


Figure 6.10 Format of exception- and interrupt-related instructions

mfc0、mtc0 根据 31:26 和 25:21 判断，syscall 根据 31:26 和 5:0 判断

eret 根据 31:26、25:21、5:0 判断

因此模块输入增加一个 25:21 端

2. 判断 mfc0 是取 CP0 哪个寄存器，mtc0 是写 CP0 哪个寄存器

指令格式“助记符 rt,rd”，根据 rd 的值判断是哪个寄存器：12→cau 13→sta 14→epc

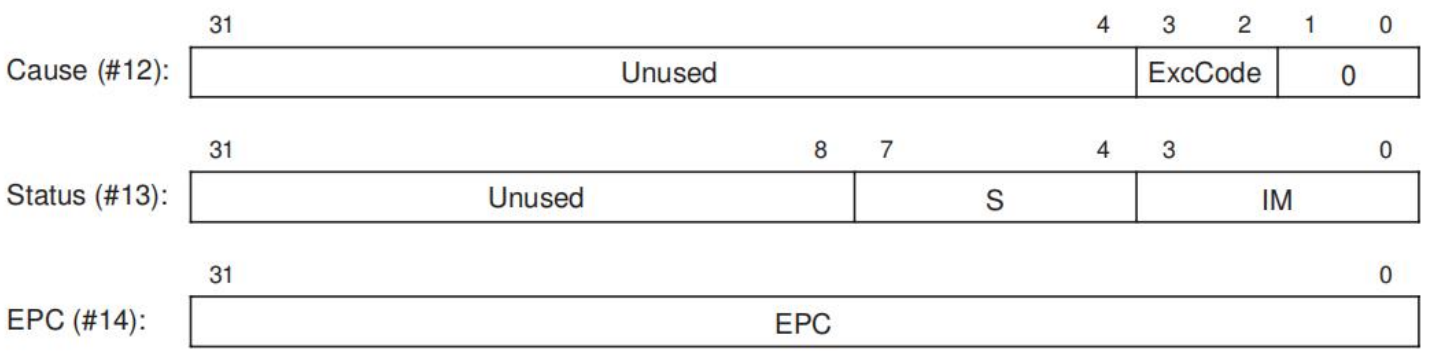


Figure 6.9 Three registers related to exceptions and interrupts

因此模块输入要把 rd 也输入进来

3. 判断是有异常还是有中断

系统调用异常➡i_syscall 有效

算术溢出➡执行阶段输入的 overflow 且是执行 i_add、i_sub、i_addi 指令

未实现的指令➡译码后没有一个指令信号有效

中断➡外部输入中断请求 intr

因此需要增加两个模块输入：overflow 和 intr

4. 判断是否中断/异常是否能发生——没有被屏蔽

3 产生的中断/异常信号分别与 sta 寄存器的 IM[3:0]相与，最后为 1 的表示产生这个中断/异常

因此需要增加一个模块输入：sta 寄存器的值

5. 中断没有被屏蔽就给出中断响应信号 inta

inta 即为有中断 intr 且未被屏蔽

inta 要输出，因此模块输入增加一个：inta

6. 进行响应工作

1. 写 exccode

exccode 的值：中断 00、系统调用 01、未实现的指令 10、算术溢出 11

Table 6.2 Definition of ExcCode and interrupt mask

ExcCode	Mnemonic	Mask	Description
0	Int	IM[0]	External interrupt
1	Sys	IM[1]	Executing the system call instruction
2	Unimpl	IM[2]	Executing an unimplemented instruction
3	Ov	IM[3]	Arithmetic signed overflow

2. 产生 sta、epc 的一级数据选择信号 exc、inta

exc 信号表示发生了中断或异常 exc=4 的信号相或，inta 为 5 的 inta

inta 在 npc 和 pc 中选择，这里的 npc 是一级 PC 选择结果(pcAdder4,bpc,rpc,jpc 四选一)产生的 npc

3. 产生 PC 的二级写信号

PC 写的二级信号 pcsel:00➡npc 01➡epc 10➡base

当 exc 为 1 时，选 base，当 eret 时选 epc

该二级选择器信号需要输出：pcsel

7. 产生 CP0 寄存器的写使能信号 wcau、wsta、wepc

   写 cause 的情况：mtc0 且寄存器是 cause、发生中断/异常 exc

   写 epc 的情况：mtc0 且寄存器是 epc、发生中断/异常 exc

   写 sta 的情况：mtc0 且寄存器是 sta、发生中断/异常 exc、中断返回

   三个输出信号：wcau、wsta、wepc

8. 产生写寄存器数据的二级选择信号 mfc0_sel

   00→一级选择的结果，01→sta，10→cau，11→epc

   mfc0_sel[0]=i_mfc0&(is_sta|is_epc),mfc0_sel[1]=i_mfc0&(is_cau|is_epc)

9. 已有信号增加源条件

   regdst、wreg

```
`timescale 1ns / 1ps

module cu(
    input [31:0]status_word,
    input [5:0]funct,op,
    input [4:0]decode_cp0,rd,
    input eq,intr,overflow,

    //第一类：选择器信号
    output regdst,alusrc,shift,m2reg,jal,exc,inta,mtc0_sel,
    output [1:0]pcsrc,pcsel,
    //第二类：存储器寄存器写使能
    output wmem,wreg,wcau,wsta,wepc,
    output [1:0]mfc0_sel,
    //第三类：alu
    output [3:0]aluc,
    //第四类
    output sext,
    output [31:0]cause_word
);
```

```
    wire rtype=~op[5]&~op[4]&~op[3]&~op[2]&~op[1]&~op[0];//op 000000
    wire i_add=rtype&funct[5]&~funct[4]&~funct[3]&~funct[2]&~funct[1]&~funct[0];//op 0 funct 100000
    wire i_sub=rtype&funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&~funct[0];//op 0 funct 100010
    wire i_and=rtype&funct[5]&~funct[4]&~funct[3]&funct[2]&~funct[1]&~funct[0];//op 0 funct 100100
    wire i_or=rtype&funct[5]&~funct[4]&~funct[3]&funct[2]&~funct[1]&funct[0];//op 0 funct 100101
    wire i_xor=rtype&funct[5]&~funct[4]&~funct[3]&funct[2]&funct[1]&~funct[0];//op 0 funct 100110
    wire i_sll=rtype&~funct[5]&~funct[4]&~funct[3]&~funct[2]&~funct[1]&~funct[0];//op 0 funct 000000
    wire i_srl=rtype&~funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&~funct[0];//op 0  funct 000010
    wire i_sra=rtype&~funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&funct[0];//op 0 funct 000011
    wire i_jr=rtype&~funct[5]&~funct[4]&funct[3]&~funct[2]&~funct[1]&~funct[0];//op 0 funct 001000
    wire i_addi=~op[5]&~op[4]&op[3]&~op[2]&~op[1]&~op[0];//op 001000
    wire i_andi=~op[5]&~op[4]&op[3]&op[2]&~op[1]&~op[0];//op 001100
    wire i_ori=~op[5]&~op[4]&op[3]&op[2]&~op[1]&op[0];//op 001101
    wire i_xori=~op[5]&~op[4]&op[3]&op[2]&op[1]&~op[0];//op 001110
    wire i_lw=op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];//op 100011
    wire i_sw=op[5]&~op[4]&op[3]&~op[2]&op[1]&op[0];////op 101011
    wire i_beq=~op[5]&~op[4]&~op[3]&op[2]&~op[1]&~op[0];//op 000100
    wire i_bne=~op[5]&~op[4]&~op[3]&op[2]&~op[1]&op[0];//op 000101
    wire i_lui=~op[5]&~op[4]&op[3]&op[2]&op[1]&op[0];//op 001111
    wire i_j=~op[5]&~op[4]&~op[3]&~op[2]&op[1]&~op[0];//op 000010
    wire i_jal=~op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];//op 000011

    //1.译码新增的四条指令，需要多输入一个 25:21 端数据 decode_cp0
    wire cpo_inst=~op[5]&op[4]&~op[3]&~op[2]&~op[1]&~op[0];//op 010000
    wire
i_mfc0=cpo_inst&~decode_cp0[4]&~decode_cp0[3]&~decode_cp0[2]&~decode_cp0[1]&~decode_cp0[0];//op
010000 decode_cp0 00000
    wire
i_mtc0=cpo_inst&~decode_cp0[4]&~decode_cp0[3]&decode_cp0[2]&~decode_cp0[1]&~decode_cp0[0];//op
010000 decode_cp0 00100
    wire
i_eret=cpo_inst&decode_cp0[4]&~decode_cp0[3]&~decode_cp0[2]&~decode_cp0[1]&~decode_cp0[0]&~funct[5]&
funct[4]&funct[3]&~funct[2]&~funct[1]&~funct[0];//op 010000 decode_cp0 10000 funct 011000
```

```
    wire i_syscall=rtype&~funct[5]&~funct[4]&funct[3]&funct[2]&~funct[1]&~funct[0];//op 000000 funct
001100

    //2.判断 mfc0/mtc0 是对 CP0 哪个寄存器操作,需要多一个 rd 输入
    wire is_cau=(rd==5'd12);
    wire is_sta=(rd==5'd13);
    wire is_epc=(rd==5'd14);

    //3.判断是否有中断异常请求发生,需要多一个 intr 外部的中断请求,overflow 溢出的输入
    wire intr_req=intr;
    wire overflow_req=overflow&(i_add|i_sub|i_addi);
    wire syscall_req=i_syscall;
    wire
unimplement_inst_req=~(i_add|i_sub|i_and|i_or|i_xor|i_sll|i_srl|i_sra|i_jr|i_addi|i_andi|i_ori|

i_xori|i_lw|i_sw|i_beq|i_bne|i_lui|i_j|i_jal|i_mfc0|i_mtc0|i_eret|i_syscall);

//4.判断中断/异常请求能否被响应，需要利用 sta 判断，因此需要多一个 sta 的输入
wire int_intr=intr_req&status_word[0];//00
wire exc_syscall=syscall_req&status_word[1];//01
wire exc_unimplement_inst=unimplement_inst_req&status_word[2];//10
wire exc_overflow=overflow_req&status_word[3];//11

//5.产生中断响应信号 inta,inta 需要输出
assign inta=int_intr;

//6.中断/异常响应
//6.1 写 Cause 寄存器，因此需要产生 exccode，并组合成 Cause 字，输出
assign cause_word={28'b0,exc_unimplement_inst|exc_overflow,exc_syscall|exc_overflow,2'b0};
//6.2 产生写 sta、epc 数据的一级选择信号 exc、inta，需要输出
```

```
    assign exc=int_intr|exc_syscall|exc_unimplement_inst|exc_overflow;//表示有中断/异常响应
    //6.3 产生 cause 写数据的选择信号和 sta、epc 写数据的二级选择信号,以及寄存器的写信号
    assign mtc0_sel=i_mtc0;
    assign wcau=exc|i_mtc0&is_cau;
    assign wsta=exc|i_mtc0&is_sta|i_eret;
    assign wepc=exc|i_mtc0&is_epc;
    //6.4 产生 NPC 的选择信号 00 NPC 01 epc 10 BASE
    assign pcsel[0]=i_eret;
    assign pcsel[1]=exc;

    //7 产生写寄存器数据的二级选择信号
    assign mfc0_sel[0]=i_mfc0&(is_sta|is_epc);
    assign mfc0_sel[1]=i_mfc0&(is_cau|is_epc);

    assign
wreg=i_add|i_sub|i_add|i_or|i_xor|i_sll|i_srl|i_sra|i_addi|i_andi|i_ori|i_xori|i_lw|i_lui|i_jal|i_mf
c0;
    assign regdst=i_addi|i_andi|i_ori|i_xori|i_lw|i_lui|i_mfc0;
    assign jal=i_jal;
    assign m2reg=i_lw;
    assign shift=i_sll|i_srl|i_sra;
    assign alusrc=i_addi|i_andi|i_ori|i_xori|i_lw|i_sw+i_lui;
    assign sext=i_addi|i_lw|i_sw|i_beq|i_bne;
    assign aluc[3]=i_sra;
    assign aluc[2]=i_sub|i_or|i_srl|i_sra|i_ori|i_lui;
    assign aluc[1]=i_xor|i_sll|i_srl|i_sra|i_xori|i_beq|i_bne|i_lui;
    assign aluc[0]=i_and|i_or|i_sll|i_srl|i_sra|i_andi|i_ori;
    assign wmem=i_sw;
    assign pcsrc[1]=i_j|i_jal|i_jr;
    assign pcsrc[0]=i_beq&eq|i_bne&~eq|i_j|i_jal;
endmodule
```

Verilog

数据通路

数据通路即为示意图所示：

需要增加三个寄存器 Cause、EPC、Status，构建一个寄存器模块，然后声明例化三次即可，如下：

```
 `timescale 1ns / 1ps


module reg_design(
    input clk,enable,
    input [31:0]i_data,
    output [31:0]o_data
);
    reg [31:0]cur_data;
    assign o_data=cur_data;
    always @(posedge clk) begin
        if (enable) begin
            cur_data<=i_data;
        end
    end
endmodule

/*
//增加的三个寄存器
reg_design Cause(clk,wcau,cause_write,cause_read);
reg_design Status(clk,wsta,status_write,status_word);
reg_design EPC(clk,wepc,epc_write,epc_read);*/
```

Verilog

然后再增加两个多路选择器，选择信号已由 CU 产生：

```
//分别是写寄存器数据的选择和写 PC 的数据的选择
```

Verilog

ALU 模块需要进行溢出判断

```
`timescale 1ns / 1ps

module alu(
    input [31:0]a,b,//a 端多路选择 pc(pc 的 adder 可以放在 IF),sa,reg_a;b 端多路选择
reg_b,imm,4(pc_adder),imm<<2;
    input [3:0]aluc,
    //x000 ADD,x100 SUB,x001 AND,x101 OR,x010 XOR,x110 LUI,0011 SLL,0111 SRL,1111 SRA
    output wire[31:0]res,
    output wire z,
    output wire overflow
);
    reg [31:0]s0,s1,s2,s3;

    always @(*) begin
        casex (aluc)//带有 x 的匹配
            4'bx000:s0=a+b;
            4'bx100:s0=a+~b+1'b1;
            4'bx001:s1=a&b;
            4'bx101:s1=a|b;
            4'bx010:s2=a^b;
            4'bx110:s2={b[15:0],16'b0};
            4'b0011:s3=b<<a[4:0];
            4'b0111:s3=b>>a[4:0];
            4'b1111:s3=$signed(b)>>>a[4:0];
        endcase
    end
    assign z=(a==b)?1:0;
    assign overflow=(aluc==4'bx000)&(a[31]&b[31]&~s0[31]|~a[31]&~b[31]&s0[31])|
```

```
                    (aluc==4'bx100)&(~a[31]&b[31]&s0[31]|a[31]&~b[31]&~s0[31]);
    mux4to1 mux4to1_inst(s0,s1,s2,s3,aluc[1:0],res);
endmodule
```

Verilog

更改后的数据通路代码如下：

```
`timescale 1ns / 1ps

module datapath(
    input clk,clrn,
    input regdst,alusrc,shift,m2reg,jal,mtc0_sel,exc,inta,
    input wreg,wcau,wsta,wepc,
    input sext,

    input [1:0]pcsrc,pcsel,mfc0_sel,
    input [3:0]aluc,

    input [31:0]memread,
    input [31:0]inst,
    input [31:0]cause_word,

    output eq,overflow,
    output [5:0]funct,op,
    output [4:0]rs,rd,
    output [31:0]pc,aluout,qb,status_word
);
    //pc 赋值
    wire [31:0]npc_temp,pcAdder4,bpc,jpc,rpc,epc_read,npc;
    wire [31:0]base=32'h0000_0008;
    mux4to1 npcTemp_init(pcAdder4,bpc,rpc,jpc,pcsrc,npc_temp);
    mux4to1 npc_init(npc_temp,epc_read,base,32'b0,pcsel,npc);
    pc pc_init(clk,clrn,npc,pc);
```

```
    adder pc_adder(pc,32'd4,pcAdder4);
    assign jpc={pcAdder4[31:29],inst[25:0],2'b00};

    //赋值寄存器相关
    wire[4:0]rt;
    assign rs=inst[25:21];
    assign rt=inst[20:16];
    assign rd=inst[15:11];
    assign funct=inst[5:0];
    assign op=inst[31:26];

    //产生写寄存器地址
    wire [31:0]tempA;
    wire [4:0]wn;
    mux2to1 mux2to1_wreg_A({27'b0,rd},{27'b0,rt},regdst,tempA);
    assign wn=tempA[4:0]|{5{jal}};

    //写寄存器数据选择
    wire [31:0]wregData_temp1,wregData_temp2;
    wire [31:0]cause_read;
    mux2to1 generate_wregData_temp1(aluout,memread,m2reg,wregData_temp1);
    mux4to1
generate_wregData_temp2(wregData_temp1,status_word,cause_read,epc_read,mfc0_sel,wregData_temp2);
    wire [31:0]wregData;
    mux2to1 generate_wregData(wregData_temp2,pcAdder4,jal,wregData);
    //寄存器模块
    wire [31:0]qa;
    regfile regfile_init(clk,wreg,clrn,rs,rt,wn,wregData,qa,qb);
    assign rpc=qa;
    assign eq=~|(qa^qb);
```

```
    //立即数扩展模块
    wire [15:0]imm=inst[15:0];
    wire [31:0]imm_sext={{16{sext&imm[15]}},imm};
    adder generate_bpc(pcAdder4,imm_sext<<2,bpc);

    //ALU 模块
    wire [31:0]a,b;
    wire z;
    mux2to1 generate_alua(qa,{26'b0,imm[10:6]},shift,a);
    mux2to1 generate_alub(qb,imm_sext,alusrc,b);
    alu alu_init(a,b,aluc,aluout,z,overflow);

    //Cause 输入数据的选择
    wire [31:0]cause_write;
    mux2to1 generate_causeWriteData(cause_word,qb,mtc0_sel,cause_write);
    //Status 输入数据的选择
    wire [31:0]status_temp1,status_write;
    wire [31:0]status_left={status_word[27:0],4'b0};
    wire [31:0]status_right={4'b0,status_word[31:4]};
    mux2to1 generate_statusTemp1(status_right,status_left,exc,status_temp1);
    mux2to1 generate_statusWrite(status_temp1,qb,mtc0_sel,status_write);
    //EPC 输入数据的选择
    wire [31:0]epc_temp1,epc_write;
    mux2to1 generate_epcTemp1(pc,npc_temp,inta,epc_temp1);
    mux2to1 generate_epcWrite(epc_temp1,qb,mtc0_sel,epc_write);
    //增加的三个寄存器
    reg_design Cause(clk,wcau,cause_write,cause_read);
    reg_design Status(clk,wsta,status_write,status_word);
    reg_design EPC(clk,wepc,epc_write,epc_read);
endmodule
```

Verilog

其他数据通路底层器件不需要改变

指令、数据存储器

数据存储器

```
`timescale 1ns / 1ps

module data_ram #(parameter LENGTH = 32)(
    input clk,we,

    input [31:0]addr,wdata,
    output [31:0]readData
);
    reg [31:0]ram[LENGTH-1:0];
    assign readData = ram[addr[6:2]]; // use word address to read ram
    always @(posedge clk) begin
        if (we) begin
            ram[addr[6:2]]<=wdata;
        end
    end

    //RAM 初始化
    integer i;
    initial begin // initialize memory
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        // ram[word_addr] = data // (byte_addr) item in data array
        ram[5'h08] = 32'h00000030; //08 是基址，0 是 int 的处理程序
        ram[5'h09] = 32'h0000003c; //08+1 是系统调用的处理程序
        ram[5'h0a] = 32'h00000054; //08+2 是没有实现的指令的处理程序
```

```
        ram[5'h0b] = 32'h00000068; //08+3 是溢出的处理程序
        ram[5'h12] = 32'h00000002; //12, 13 单元上的字是为了实现溢出测试 2+7fffffff
        ram[5'h13] = 32'h7fffffff;
        ram[5'h14] = 32'h000000a3; //之前用到的计算
        ram[5'h15] = 32'h00000027;
        ram[5'h16] = 32'h00000079;
        ram[5'h17] = 32'h00000115;
    end
endmodule
```

Verilog

指令存储器

```
`timescale 1ns / 1ps

module inst_rom #(parameter LENGTH = 64)(
    input [31:0]pc,
    output [31:0]inst
);
    wire [31:0]rom[LENGTH-1:0];
    assign rom[6'h00] = 32'h0800001d;
    assign rom[6'h01] = 32'h00000000;
    assign rom[6'h02] = 32'h401a6000;
    assign rom[6'h03] = 32'h335b000c;
    assign rom[6'h04] = 32'h8f7b0020;
    assign rom[6'h05] = 32'h00000000;
    assign rom[6'h06] = 32'h03600008;
    assign rom[6'h07] = 32'h00000000;
    assign rom[6'h0c] = 32'h00000000;
    assign rom[6'h0d] = 32'h42000018;
    assign rom[6'h0e] = 32'h00000000;
```

```
    assign rom[6'h0f] = 32'h00000000;
    assign rom[6'h10] = 32'h401a7000;
    assign rom[6'h11] = 32'h235a0004;
    assign rom[6'h12] = 32'h409a7000;
    assign rom[6'h13] = 32'h42000018;
    assign rom[6'h14] = 32'h00000000;
    assign rom[6'h15] = 32'h00000000;
    assign rom[6'h16] = 32'h08000010;
    assign rom[6'h17] = 32'h00000000;
    assign rom[6'h1a] = 32'h00000000;
    assign rom[6'h1b] = 32'h08000010;
    assign rom[6'h1c] = 32'h00000000;
    assign rom[6'h1d] = 32'h2008000f;
    assign rom[6'h1e] = 32'h40886800;//pdf 指令错误
    assign rom[6'h1f] = 32'h8c080048;
    assign rom[6'h20] = 32'h8c09004c;
    assign rom[6'h21] = 32'h01094020;
    assign rom[6'h22] = 32'h00000000;
    assign rom[6'h23] = 32'h0000000c;
    assign rom[6'h24] = 32'h00000000;
    assign rom[6'h25] = 32'h0128001a;
    assign rom[6'h26] = 32'h00000000;
    assign rom[6'h27] = 32'h34040050;
    assign rom[6'h28] = 32'h20050004;
    assign rom[6'h29] = 32'h00004020;
    assign rom[6'h2a] = 32'h8c890000;
    assign rom[6'h2b] = 32'h20840004;
    assign rom[6'h2c] = 32'h01094020;
    assign rom[6'h2d] = 32'h20a5ffff;
    assign rom[6'h2e] = 32'h14a0fffb;
    assign rom[6'h2f] = 32'h00000000;
    assign rom[6'h30] = 32'h08000030;
```

```
    assign inst = rom[pc[7:2]]; // use word address to read rom,pc 每次+4

endmodule
```

Verilog

顶层模块

加一个 intr 中断请求输入和 inta 中断响应输出

```
`timescale 1ns / 1ps

module top(
    input clk,clrn,intr,
    output inta,
    output [31:0]pc,
    output [31:0]inst,
    output [31:0]aluout,memread
);
    wire [5:0]funct,op;
    wire eq,overflow;
    wire regdst,alusrc,shift,m2reg,jal,exc,mtc0_sel;
    wire [1:0]pcsrc,pcsel;
    wire wmem,wreg,wcau,wsta,wepc;
    wire [1:0]mfc0_sel;
    wire [3:0]aluc;
    wire sext;
    wire [31:0]qb,status_word,cause_word;
    wire [4:0]rd,decode_cp0;

    datapath  datapath_inst (
        .clk(clk),
        .clrn(clrn),
        .regdst(regdst),
        .alusrc(alusrc),
```

```
        .shift(shift),
        .m2reg(m2reg),
        .jal(jal),
        .mtc0_sel(mtc0_sel),
        .exc(exc),
        .inta(inta),
        .wreg(wreg),
        .wcau(wcau),
        .wsta(wsta),
        .wepc(wepc),
        .sext(sext),
        .pcsrc(pcsrc),
        .pcsel(pcsel),
        .mfc0_sel(mfc0_sel),
        .aluc(aluc),
        .memread(memread),
        .inst(inst),
        .cause_word(cause_word),
        .eq(eq),
        .overflow(overflow),
        .funct(funct),
        .op(op),
        .rs(decode_cp0),
        .rd(rd),
        .pc(pc),
        .aluout(aluout),
        .qb(qb),
        .status_word(status_word)
    );
    cu  cu_inst (
        .status_word(status_word),
        .funct(funct),
        .op(op),
```

```
        .eq(eq),
        .overflow(overflow),
        .intr(intr),
        .decode_cp0(decode_cp0),
        .rd(rd),
        .regdst(regdst),
        .alusrc(alusrc),
        .shift(shift),
        .m2reg(m2reg),
        .jal(jal),
        .exc(exc),
        .inta(inta),
        .mtc0_sel(mtc0_sel),
        .pcsrc(pcsrc),
        .pcsel(pcsel),
        .mfc0_sel(mfc0_sel),
        .wmem(wmem),
        .wreg(wreg),
        .wcau(wcau),
        .wsta(wsta),
        .wepc(wepc),
        .aluc(aluc),
        .sext(sext),
        .cause_word(cause_word)
    );
    inst_rom inst_romInst(pc,inst);
    data_ram data_ramInst(clk,wmem,aluout,qb,memread);
endmodule
```

Verilog

6.2.5 测试仿真

```Verilog
`timescale 1ns / 1ps

module top_test();
    reg clk,clrn,intr;
    wire inta;
    wire [31:0]pc;
    wire [31:0]inst;
    wire [31:0]aluout,memread;
    initial begin
        clk=0;clrn=0;intr=0;
        #3;clrn=1;
        #1;clrn=0;
        #516;intr=1;
        #10;intr=0;
    end
    always #5 clk=~clk;
    top  top_inst (
        .clk(clk),
        .clrn(clrn),
        .intr(intr),
        .inta(inta),
        .pc(pc),
        .inst(inst),
        .aluout(aluout),
        .memread(memread)
    );
endmodule
```

实现过程中出现了很多粗心的问题：

1. 命名的问题，一个模块中和一个模块中的命名不对应：pcsel selpc

2. 有的模块实例化时偷懒，复制的之前的实例化，没有改信号或者改的不全

```
reg_design Cause(clk,wcau,cause_write,cause_read);
reg_design Status(clk,wsta,status_write,status_word);
reg_design EPC(clk,wepc,epc_write,epc_read);
```

3. CU 单元的译码也是，复制的 op 的，但是是 funct 译码

4. 还有信号的位宽问题

```
wire [31:0]status_left={status_word[27:0],4'b0};
wire [31:0]status_right={4'b0,status_word[31:4]};
mux2to1 generate_statusTemp1(status_right,status_left,exc,status_temp1);
mux2to1 generate_statusWrite(status_temp1,qb,mtc0_sel,status_write);
```

习题

1. *Design the interrupt controller shown in Figure 6.7 in Verilog HDL.*

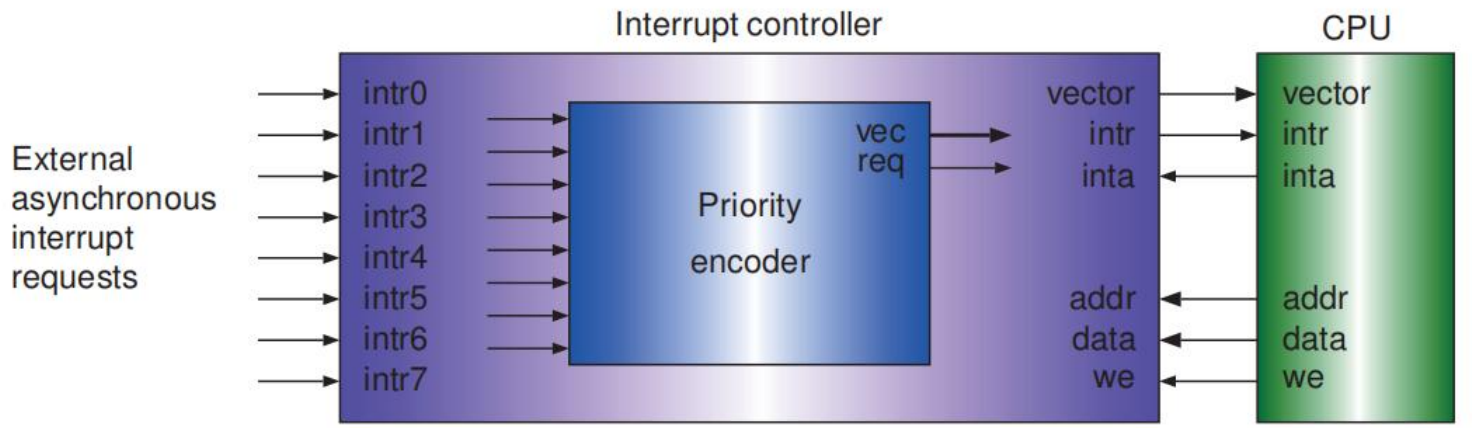


Figure 6.7 Priority interrupt controller

主要是实现优先级编码器，在已经实现过了优先级编码器

采用的是 Vectored interupt，编码得到的 Vector 就是 CPUvector 的最右侧 3 位

CPU 的 addr、data、we 主要是实现和中断控制器的通信，比如写方式字、控制字等

2. *Design the Daisy chain circuit shown in Figure 6.8 in Verilog HDL.*

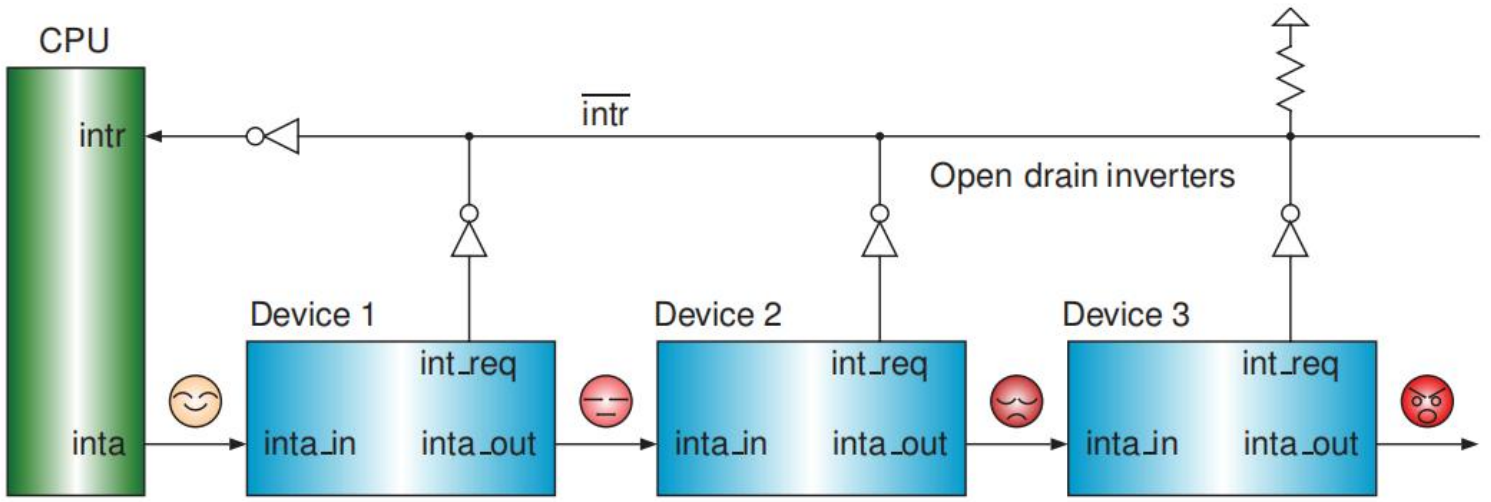


Figure 6.8 Daisy chain for priority interrupts

上面已经设计出了 open-drain 反相器，因此只需要实现设备之间的 inta 流动即可

当当前设备有中断响应时，会锁住 inta，使得 inta_out=0 流不到下一级，否则会使 inta_out=1。当设备被中断响应，int_req 即置为 0。中断返回后，又交给 CPU 控制去发出 inta 响应之后的设备

```Verilog

```

3. *Design a single-cycle CPU which can deal with multiple interrupts.*

   多个中断的话，则可以建立两个表，一个是异常处理程序的入口地址表，一个是中断处理程序的入口地址表➡寻找入口地址的汇编程序如下：

```
mfc0 $26,c0_cause
andi $27,$26,0xc
beq $27,$0,int_table
lw $27,exc_table($27)
nop
jr $27
int_table:
andi $27,$26,0x30//假设高两位为 IP
lw $27,int_table($27)
nop
jr $27
```

```
Verilog
```

[注释 1] 在这个向量前后附加一些值得到地址，直接->这个地址就是入口地址，"j 地址"直接给 PC；间接->这个地址是一个保存了中断服务程序的入口地址的寄存器地址，使用"jr 寄存器"来实现跳转

[注释 2] 保存的方法可以是保存在通用寄存器中，也可以保存在一个特殊的寄存器中，也可以保存在堆栈中

[注释 3] 也在协处理器 CP0 中

[注释 4] 采用分支延迟技术——延迟槽的流水线需要保存 PC+8

[注释 5] 无符号计算的 addu、subu 在任何情况下都不会发生算术溢出