

计算机组成原理 (2017级)



北京航空航天大学计算机学院中德所
栾钟治

I/O 设备及其速度

- I/O 的速度: 从鼠标到局域网可以跨越7个数量级

设备	行为	交互对象	数据率 (KB/s)
键盘	输入	人类	0.01
鼠标	输入	人类	0.02
音频输出	输出	人类	5.00
软盘	存储	机器	50.00
激光打印机	输出	人类	100.00
磁盘	存储	机器	10,000.00
无线网络	输入或输出	机器	10,000.00
图形化显示	输出	人类	30,000.00
有线局域网	输入或输出	机器	125,000.00

输入/输出 (I/O)

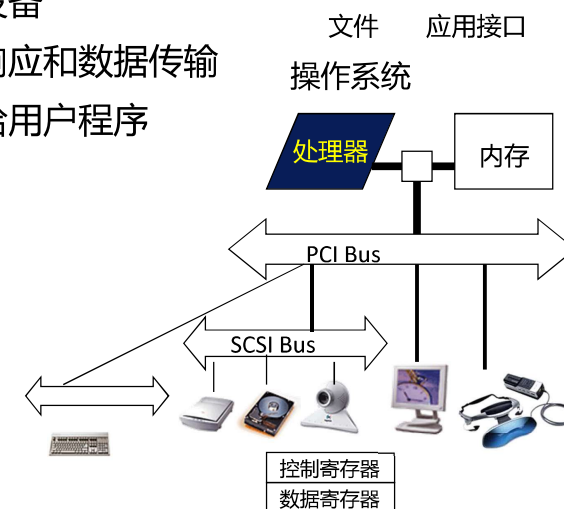
- ❖ 人类与计算机通过I/O交互
- ❖ 计算机需要通过I/O获得持久化的存储能力
- ❖ 计算机还可以通过I/O展现各种令人惊异的能力



MIT Media Lab
"Sixth Sense"

我们需要I/O做什么?

- ❖ 连接多种类型设备
- ❖ 设备的控制、响应和数据传输
- ❖ 提供相关能力给用户程序

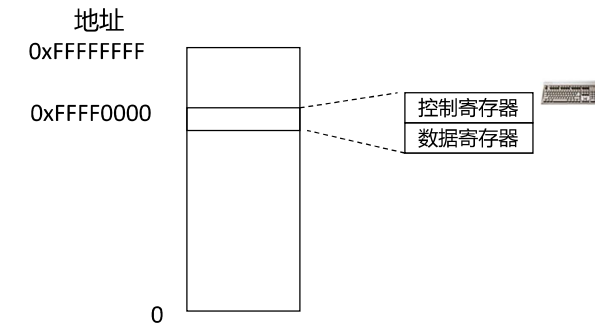


指令集体系结构 (ISA) 与 I/O

- ❖ 处理器要为 I/O 做什么?
 - 输入: 读字节序列
 - 输出: 写字节序列
- ❖ 有些处理器有专门的输入输出指令
- ❖ 另外的模式(MIPS):
 - 用load实现输入, 用store实现输出 (小规模)
 - *Memory Mapped Input/Output*
 - 贡献出一部分地址空间作为输入输出设备的通信通路

内存映射 I/O

- ❖ 某确定的地址段并不是常规的内存
- ❖ 它们被关联到设备的寄存器



处理器-I/O的速度不匹配

- ❖ 1 GHz 的微处理器每秒可以执行10亿条load或store指令
(4,000,000 KB/s 数据率)
 - I/O设备的数据率范围从 0.01 KB/s到125,000 KB/s
- ❖ 输入: 设备无法及时准备好数据供处理器load
 - 可能还需要等待人类响应
- ❖ 输出: 设备无法及时准备好接收处理器store的数据
- ❖ 怎么办?

处理器执行动作前先查询状态

- ❖ 通向设备的通信通路上通常有2个寄存器:
 - **控制寄存器** 用来确认是否允许读/写 (I/O ready)
 - **数据寄存器** 暂存数据
- ❖ 处理器周期性查询 (循环) 控制寄存器, 等待设备置位 **Ready bit** (0 → 1)
- ❖ 接着处理器load(输入)或者store(输出)数据寄存器
 - 重置控制寄存器 Ready bit (1 → 0)
- ❖ 这一过程称为“**Polling**” (轮询)

I/O 示例 (MIPS中的轮询)

❖ 输入: 从键盘读入 \$v0

```
Waitloop:    lui    $t0, 0xffff # ffff0000
             lw     $t1, 0($t0) # control reg
             andi   $t1, $t1, 0x1
             beq    $t1, $zero, Waitloop
             lw     $v0, 4($t0) # data reg
```

❖ 输出: 从 \$a0 写到显示器

```
Waitloop:    lui    $t0, 0xffff # ffff0000
             lw     $t1, 8($t0) # control reg
             andi   $t1, $t1, 0x1
             beq    $t1, $zero, Waitloop
             sw     $a0, 12($t0) # data reg
```

❖ “Ready” —处理器的视角!

轮询的开销

❖ 处理器规格: 1 GHz 时钟频率, 完成一个轮询操作需要400个时钟周期 (轮询程序, 读写设备, 返回)

❖ 轮询对CPU资源的占用:

- 鼠标: 每秒30次查询, 确保不会遗漏用户的动作
- 软盘: 以2字节为一个单元, 50 KB/秒的数据率传输, 没有遗漏
- 硬盘: 以16字节大小的块为单位, 16 MB/秒的数据传输率, 没有遗漏

轮询的处理器时间占比

❖ 鼠标轮询:

- 占用时间: $30 \text{ [轮询/秒]} \times 400 \text{ [时钟周期/轮询]} = 12\text{K} \text{ [时钟周期/秒]}$
- 时间百分比: $1.2 \times 10^4 \text{ [时钟周期/秒]} / 10^9 \text{ [时钟周期/秒]} = 0.0012\%$
- 鼠标轮询对处理器影响很小

❖ 磁盘轮询:

- 频率: $16 \text{ [MB/秒]} / 16 \text{ [B/轮询]} = 2^{20} \text{ [轮询/秒]}$
- 占用时间: $2^{20} \text{ [轮询/秒]} \times 400 \text{ [时钟周期/轮询]} \approx 419\text{M} \text{ [时钟周期/秒]}$
- 时间百分比: $4.19 \times 10^8 \text{ [时钟周期/秒]} / 10^9 \text{ [时钟周期/秒]} = 41.9\%$
- 不可接受!

❖ 问题: 轮询, 读写较小的块

替代方案?

- ❖ 浪费太多处理器时间用于“自旋等待” (spin-waiting) I/O就绪
- ❖ 当I/O设备就绪时调用相关的过程
- ❖ 方案: 使用异常机制触发I/O, 然后在I/O进行数据传输的时候中断程序

异常和中断

❖ “突发的” 事件需要改变控制流

- 不同的指令集体系结构会使用不同的术语

❖ 异常

- CPU内部产生
(例如未定义的opcode, 溢出, 系统调用, TLB 缺失)

❖ 中断

- 来自外部I/O控制器

❖ 需要牺牲性能

异常的特性

❖ 可重启的异常

- 流水线能够清空指令
- 处理程序的执行, 执行完毕后返回指令
 - 重新取指和执行

❖ PC+4 保存在EPC 寄存器

- 识别导致异常的指令
- 使用PC+4是因为它是流水线执行中可以获得的信号
 - 处理程序必须能够校准这个值以获得正确的地址

处理异常

❖ MIPS中异常由系统控制协处理器 (CP0) 处理

❖ 保存出问题 (或者被中断) 的指令的PC内容

- MIPS: 保存在特殊的寄存器中

Exception Program Counter (EPC)

❖ 保存问题的描述

- MIPS: 保存在特殊的寄存器中, *Cause* 寄存器
- 最简单的实现只需要1bit (0: 未定义的opcode, 1: 溢出)

❖ 转跳到异常处理代码(*exception handler code*), 起始地址: 0x80000180

❖ 通知操作系统

- 可以“杀”程序
- 对于 I/O 设备请求或系统调用, 通常同时切换到另一个进程
 - 比如当发生 TLB 缺失和页失效时

处理程序的动作

❖ 读 Cause 寄存器, 传递给相关的处理程序

❖ 操作系统确定需要采取的行动:

- 如果是可重启的异常, 执行正确的操作然后使用EPC返回程序
- 否则, 结束程序并使用EPC, Cause 寄存器等信息报告错误

I/O 中断

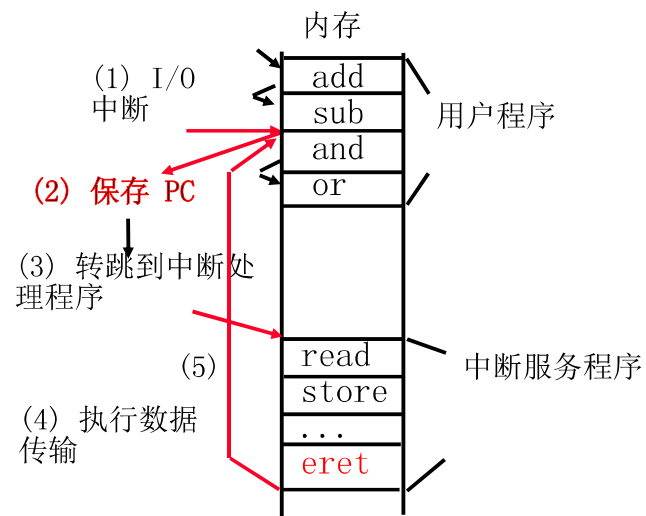
❖ I/O 中断与异常很相似，不同之处在于：

- I/O 中断是“异步”的
- 需要传递更多的信息

❖ “异步”是相对于指令执行的

- I/O 中断不与任何指令关联，但在任何指令执行的过程中发生
- I/O 中断不会阻止任何指令的运行

中断驱动的数据传输



协处理器0 (CP0)

❖ 4个寄存器：SR、Cause、EPC、PRId

- 阅读《See MIPS Run Linux》第3章
- 无关寄存器及无关位可以不阅读

❖ 理解要点：

- SR：用于对系统进行控制
 - 指令可读可写
- Cause：指令读取，硬件控制写入
 - IP7-2[15:10]：对应外部6个中断源
 - ExcCode[6:2]：异常/中断类型编码值
- EPC：用于保存异常/中断发生时的PC
 - 保存PC：硬件控制写入
 - 指令读取：中断服务程序
- PRId：处理器ID

寄存器号	寄存器
12	SR
13	CAUSE
14	EPC
15	PRID

EPC

❖ EPC：保存中断/异常时的PC

- 以便从中断/异常服务程序返回至被中断指令

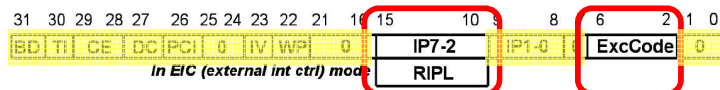
❖ ERET：中断/异常服务程序返回指令

编码	31	26	25	22	21	18	17	14	13	10	9	6	5	0
	CP0		80000										eret	
	010000		1000		0000		0000		0000		0000		011000	
	6		20										6	
格式	eret													
描述	eret将保存在CP0的EPC寄存器中的现场(被中断指令的下一条地址)写入PC，从而实现从中断、异常或指令执行错误的处理程序中返回。													
操作	PC ← CP0[epc]													
示例	eret													
其他	当程序被硬件中断、指令执行异常(如除以0、算术溢出)时，PC+4将被保存在EPC中。													

CAUSE寄存器

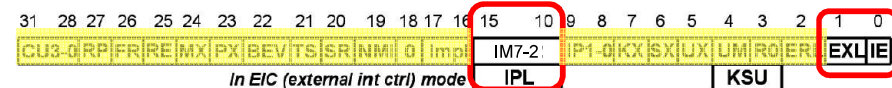
- ❖ IP7-2[15:10]：6位待决的中断位，分别对应6个外部中断
 - 记录当前哪些硬件中断正在有效
 - 1-有中断；0-无中断
- ❖ ExcCode[6:2]：异常编码，记录当前发生的是什么异常
 - 共计32种
 - 要求表中3种

ExcCode	助记符	描述
0	Int	中断
10	RI	不识别(非法)指令
12	Ov	算术指令导致的异常(如add)

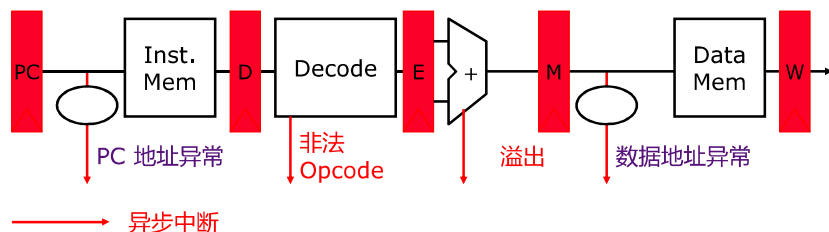


SR寄存器

- ❖ IM7-2[15:10]：6位中断屏蔽位，分别对应6个外部中断
 - 1-允许中断，0-禁止中断
- ❖ IE：全局中断使能
 - 1-允许中断；0-禁止中断
- ❖ EXL：异常级
 - 1-进入异常，不允许再中断；0-允许中断
 - 重入需要OS的配合，特别是堆栈



异常处理(5段流水线)



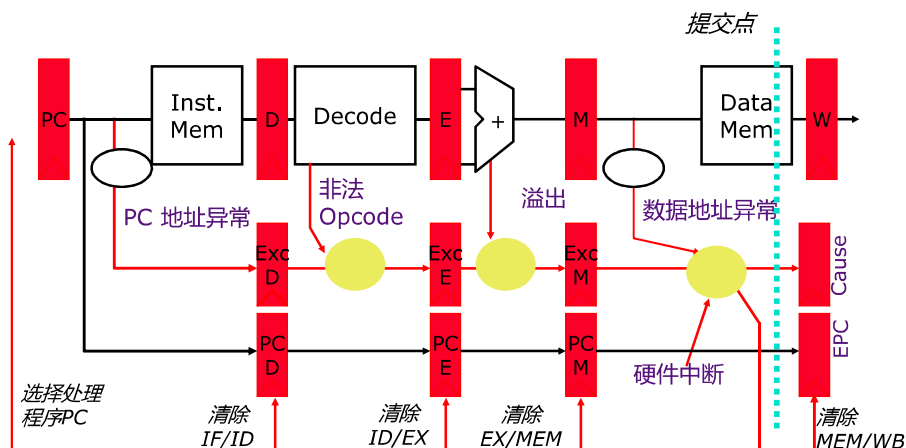
- ❖ 如何处理多个在不同流水段同时发生的异常？
- ❖ 如何以及在什么位置处理外部的异步中断？

异常：只要求实现非法指令和溢出

异常处理

- ❖ 流水线的收益来自于指令执行开销的重叠
 - 当指令流突然中断的时候（发生异常）会产生错误
- ❖ 许多的异常必须是精确的，当一个异常是精确的时
 - (同步并且可在断点处恢复)
 - 所有在出错指令之前的指令必须完成
 - 出错指令和它之后的所有指令必须被销毁（清空）
 - 异常处理程序必须开始执行
- ❖ 通常不适宜在异常发生的周期中立即处理异常，因为
 - 同一个周期内可能发生多个异常
 - 在不同的流水段处理异常会比较复杂
 - 异常处理必须按程序处理的序而不是时间序
- ❖ 通常，当异常发生时标记它并记录导致异常的原因，保持“安静”直到指令流水到写回阶段再处理

异常处理 (5段流水线)

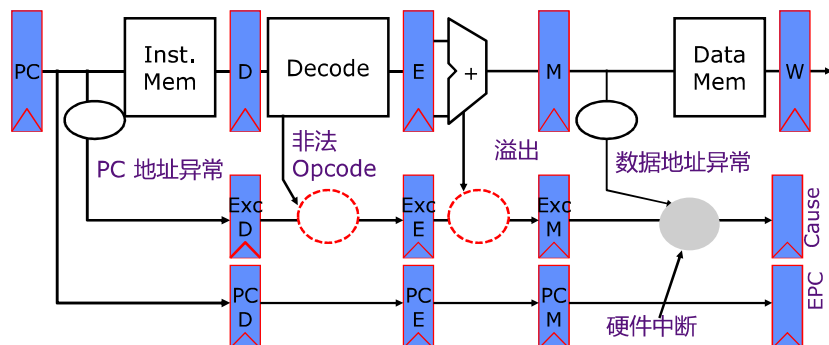
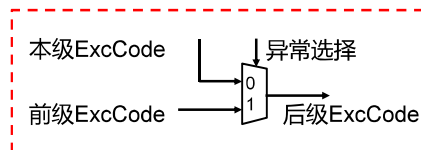


异常处理 (5段流水线)

- ❖ 在流水线中保持异常标志直到提交点 (MEM段)
- ❖ 对于给定的指令，在早期流水段的异常会覆盖后面的异常
- ❖ 在提交点注入外部中断 (覆盖其它的)
- ❖ 异常到达提交点: 更新Cause和EPC寄存器, 清空所有流水段, 向取指阶段注入处理程序的PC

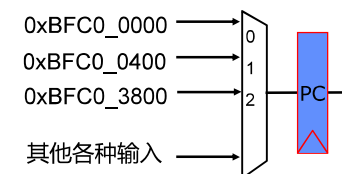
硬件实现: 传递异常

- ❖ 当前级有异常时，则传递前级异常编码
- ❖ 硬件中断的优先级高于异常



硬件实现: 修改PC

- ❖ PC需要增加: 异常处理程序的地址
 - 系统复位时输出: 0xBFC0_0000
 - 硬件中断时输出: 0xBFC0_0400
 - 其他异常时输出: 0xBFC0_0380



软件实现：中断服务程序

- ❖ 框架结构：保存现场、中断处理、恢复现场、中断返回
 - ❖ 保存现场
 - 将所有寄存器都保存在堆栈中
 - ❖ 中断处理
 - 读取特殊寄存器了解哪个硬件中断发生
 - 执行对应的处理策略（例如读写设备寄存器、存储器等）
 - ❖ 恢复现场
 - 从堆栈中恢复所有寄存器
 - ❖ 中断返回
 - 执行 **eret** 指令
- 1、3、4：通用
2：针对特定设备

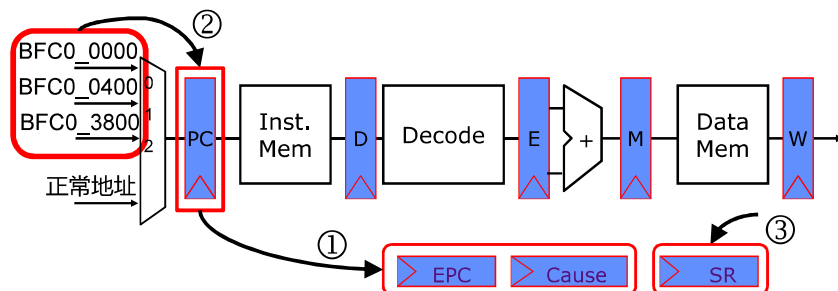
1、3、4：通用
2：针对特定设备

中断响应机制：控制器(2)

- ❖ **处理**：保存PC/跳转/关中断
- ❖ ①**保存**：将PC和ExcCode保存在EPC和Cause中
 - **注意**：PC存储的是PC+4
- ❖ ②**跳转**：产生中断处理程序入口地址并写入PC
- ❖ ③**关中断**：EXL置位，防止再次进入

实现要点：
①②③在**同一周期**完成

实现技巧：
PC/EPC/ExcCode/E
XL写使能**同时产生**



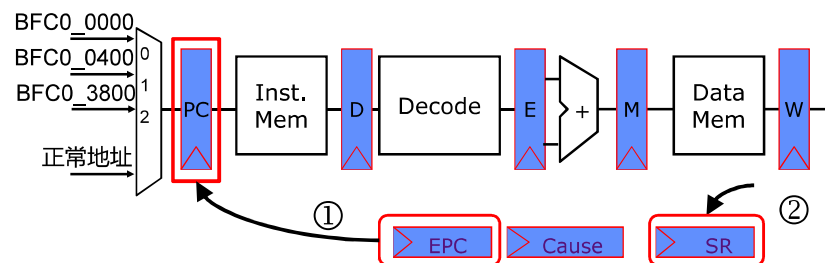
中断响应机制：检测异常与中断(1)

- ❖ 每条指令的WB阶段检测异常与中断
 - 最终异常：流水过来的前级异常
 - 是否有中断
- ❖ 中断检测时需要判断是否中断允许位
 - 解决方法：用HWINT/IM/IE/EXL产生中断请求
`assign IntReq = |(HWInt7-2 & IM7-2) & IE & !EXL ;`
- 注意：中断优先级高于异常
 - Q：怎么实现呢？
 - A：清除各级指令时，先判断中断再判断异常流水标志位

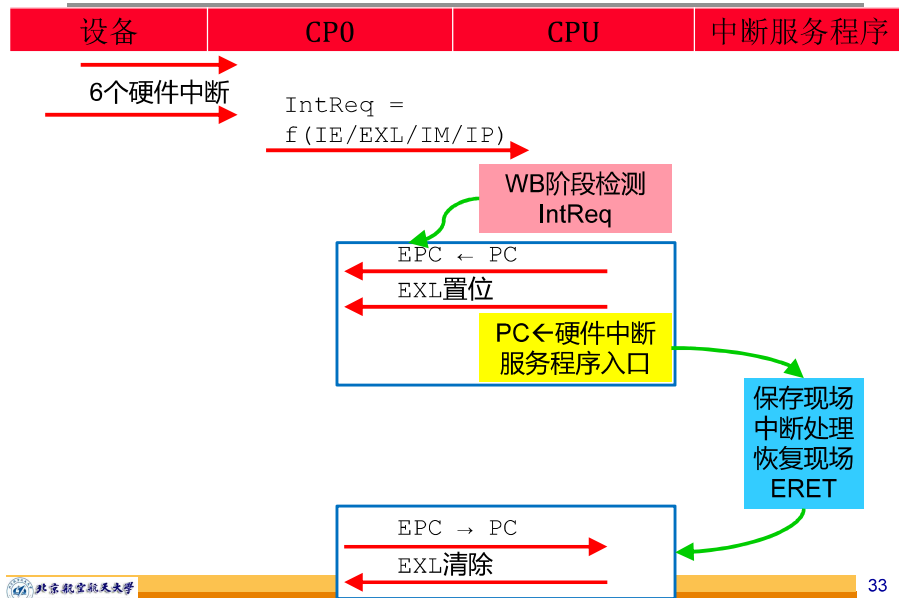
中断响应机制： ERET指令 (3)

- ❖ 恢复PC，开中断
 - ①恢复PC：将EPC写入PC
 - ②开中断：清除EXL，允许再次产生

实现要点：
①②在**同一周期**完成
实现技巧：
PC写使能/EXL清除
同时产生



中断响应机制分析：软硬件协同



中断驱动的 I/O 示例 (1/2)

❖ 假定以下系统特性:

- 每次传输包括中断在内需要500个时钟周期的开销
- 磁盘吞吐: 16 MB/秒
- 每传输16字节产生一个中断
- 处理器主频 1 GHz

❖ 如果磁盘的操作占程序执行的5%, 处理器被磁盘操作消耗的百分比是多少?

- $5\% \times 16 \text{ [MB/秒]} / 16 \text{ [B/中断]} = 52,428 \text{ [中断/秒]}$
- $52,428 \text{ [中断/秒]} \times 500 \text{ [时钟周期/中断]} = 2.62 \times 10^7 \text{ [时钟周期/秒]}$
- $2.62 \times 10^7 \text{ [时钟周期/秒]} / 10^9 \text{ [时钟周期/秒]} = 2.62\%$

中断驱动的I/O 示例 (2/2)

❖ 2.62% (中断) 远比41.9% (轮询)要好

❖ 真实的解决方案: 直接内存访问(DMA) 机制

- 设备控制器直接与内存交互完成数据传输, 不通过处理器
- 每页传输完成产生一次中断

协处理器指令及用途

❖ 指令: MFC0、MTC0

- 不能直接修改CP0寄存器, 必须借助通用寄存器

❖ MFC0: 读取CP0寄存器至通用寄存器

- SR: 获取处理器的控制信息
- Cause: 获取处理器当前所处的状态
- EPC: 获取被异常/中断的指令地址
- PRId: 读取处理器ID (可以读取你的个性签名)

❖ MTC0: 通用寄存器值写入CP0寄存器

- SR: 对处理器进行控制, 例如关闭中断
- EPC: 操作系统中用于多任务切换

设计CP0：模块接口

信号名	方向	用途	产生来源及机制
A1[4:0]	I	读CP0寄存器编号	执行MFC0指令时产生
A2[4:0]	I	写CP0寄存器编号	执行MTC0指令时产生
DIn[31:0]	I	CP0寄存器的写入数据	执行MTC0指令时产生 数据来自GPR
PC[31:2]	I	中断/异常时的PC	PC
ExcCode[6:2]	I	中断/异常的类型	异常功能部件
HWInt[5:0]	I	6个设备中断	外部硬件设备(如鼠标、键盘)
We	I	CP0寄存器写使能	执行MTC0指令时产生
EXLSet	I	用于置位SR的EXL(EXL为1)	流水线在WB阶段产生
EXLClr	I	用于清除SR的EXL(EXL为0)	执行ERET指令时产生
clk	I	时钟	
rst	I	复位	
IntReq	O	中断请求，输出至CPU控制器	是HWInt/IM/EXL/IM的函数
EPC[31:2]	O	EPC寄存器输出至NPC	
DOut[31:0]	O	CP0寄存器的输出数据	执行MFC0指令时产生，输出数据至GPR

设计CP0：SR

❖ 由于无用位较多，因此只定义有用位

```
>reg [15:10] im ;
>reg exl, ie ;
```

❖ SR整体表示为： $\{16'b0, im, 8'b0, exl, ie\}$

❖ im, ie的行为很简单

```
if (当Wen有效并且Sel为对应的寄存器编号)
    {im, exl, ie} <= {DIn[15:10], DIn[1], DIn[0]} ;
```

reg [5:0] im与reg [15:10] im
是等价的，但后者编码风格更好

设计CP0：SR

❖ exl要复杂一些：除了类似im/ie的行为外，还必须有置位和清除的功能。以置位为例：

```
if (EXLSet)
    exl <= 1'b1 ;
```

设计CP0：Cause

❖ Cause：只需定义6位寄存器，不断的锁存外部6个中断(HWInt7-2)

```
>reg [15:10] hwint_pend ;
```

❖ Cause整体表示为：

```
>{16'b0, hwint_pend, 10'b0}
```

设计CP0: EPC

❖ 定义30位寄存器

➤ `reg [32:2] epc;`

❖ 为什么不需要32位？

设计CP0: PRId

❖ 用于对公司/指令集版本等进行标识

➤ Intel处理器也有ID，CPU-Z就可以读取

31	24	23	16	15	8	7	0
Company Options		Company ID		Processor ID		Revision	

❖ 目前可以任意选择用一个4字节的编码值，如

➤ `0x1234_5678`

设计CP0: 输出CP0寄存器

❖ 除了SR/Cause/EPC/PRId外，一律输出0。

❖ 可以设计一个5选1的MUX。

❖ 也可以用行为描述，样例代码：

```
assign DOut = (Sel==12) ? {16'b0, im, 8'b0, exl, ie} :  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX :  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX :  
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX :  
32'b0 ;
```