计算机学院专业必修课

计算机组成

MIPS体系结构 ～异常与中断

高小鹏

北京航空航天大学计算机学院
系统结构研究所

提纲

- 输入输出
- 异常/中断
- 协处理器

Processor Checks Status Before Acting

- Path to a device generally has 2 registers:
  - *Status Register* says it’s OK to read/write (I/O ready)
  - *Data Register* contains data

1) Processor reads from status register in a loop, waiting for device to set *Ready bit* (0 → 1)
2) Processor then loads from (input) or writes to (output) data register
   - Resets Ready bit of status register (1 → 0)

- This process is called “*Polling*”

polling：轮询，查询

I/O Example (Polling in MIPS)

- **Input:** Read from keyboard into `$v0`

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

- **Output:** Write to display from `$a0`

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

- "Ready" bit is from processor's point of view!

I/O Device Examples and Speeds

- I/O speeds: *7 orders of magnitude* between mouse and LAN

Device	Behavior	Partner	Data Rate (KB/s)
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
Voice output	Output	Human	5.00
Floppy disk	Storage	Machine	50.00
Laser printer	Output	Human	100.00
Magnetic disk	Storage	Machine	10,000.00
Wireless network	Input or Output	Machine	10,000.00
Graphics display	Output	Human	30,000.00
Wired LAN network	Input or Output	Machine	125,000.00

- When discussing transfer rates, use SI prefixes (10^x)

orders of magnitude：数量级

Processor-I/O Speed Mismatch

- 1 GHz microprocessor can execute 1 billion load or store instr/sec (4,000,000 KB/s data rate)
  - **Recall:** I/O devices data rates range from 0.01 KB/s to 125,000 KB/s
- *Input:* Device may not be ready to send data as fast as the processor loads it
  - Also, might be waiting for human to act
- *Output:* Device not be ready to accept data as fast as processor stores it
- *What can we do?*

Cost of Polling?

- Processor specs: 1 GHz clock, 400 clock cycles for a polling operation (call polling routine, accessing the device, and returning)
- Determine % of processor time for polling:
  - **Mouse:** Polled 30 times/sec so as not to miss user movement
  - **Floppy disk:** Transferred data in 2-Byte units with data rate of 50 KB/sec. No data transfer can be missed.
  - **Hard disk:** Transfers data in 16-Byte chunks and can transfer at 16 MB/second. Again, no transfer can be missed.

% Processor time to poll

自行推导计算

- Mouse polling:
  - *Time taken:* 30 [polls/s] × 400 [clocks/poll] = 12K [clocks/s]
  - *% Time:* 1.2×10^4 [clocks/s] / 10^9 [clocks/s] = 0.0012%
  - Polling mouse little impact on processor
- Disk polling:
  - *Freq:* 16 [MB/s] / 16 [B/poll] = 1M [polls/s]
  - *Time taken:* 1M [polls/s] × 400 [clocks/poll] = 400M [clocks/s]
  - *% Time:* 4×10^8 [clocks/s] / 10^9 [clocks/s] = 40%
  - Unacceptable!
- **Problems:** polling, accessing small chunks

Alternatives to Polling?

- Wasteful to have processor spend most of its time "spin-waiting" for I/O to be ready
- Would like an unplanned procedure call that would be invoked only when I/O device is ready
- **Solution:** Use *exception* mechanism to help trigger I/O, then *interrupt* program when I/O is done with data transfer
  - This method is discussed next

提纲

- 输入输出
- 异常/中断
- 协处理器

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- *Exception*
  - Arises within the CPU
    (e.g. undefined opcode, overflow, syscall, TLB Miss)
- *Interrupt*
  - From an external I/O controller
- Dealing with these without sacrificing performance is difficult!

Handling Exceptions (1/2)

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: save in special register called *Exception Program Counter (EPC)*
- Save indication of the problem
  - In MIPS: saved in special register called *Cause* register
  - In simple implementation, might only need 2-bit (0 for undefined opcode, 1 for overflow)
- Jump to *exception handler code* at address 0x80000180

异常入口地址：不同系统不同，同一系统也可能不同

Handling Exceptions (2/2)

- Operating system is also notified
  - Can kill program (e.g. segfault)
  - For I/O device request or syscall, often switch to another process in meantime
    - This is what happens on a TLB misses and page faults

Exception Properties

- Re-startable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Re-fetched and executed from scratch
- PC+4 saved in EPC register
  - Identifies causing instruction
  - PC+4 because it is the available signal in a pipelined implementation
    - Handler must adjust this value to get right address

难题：到底返回到哪条指令？后面讨论

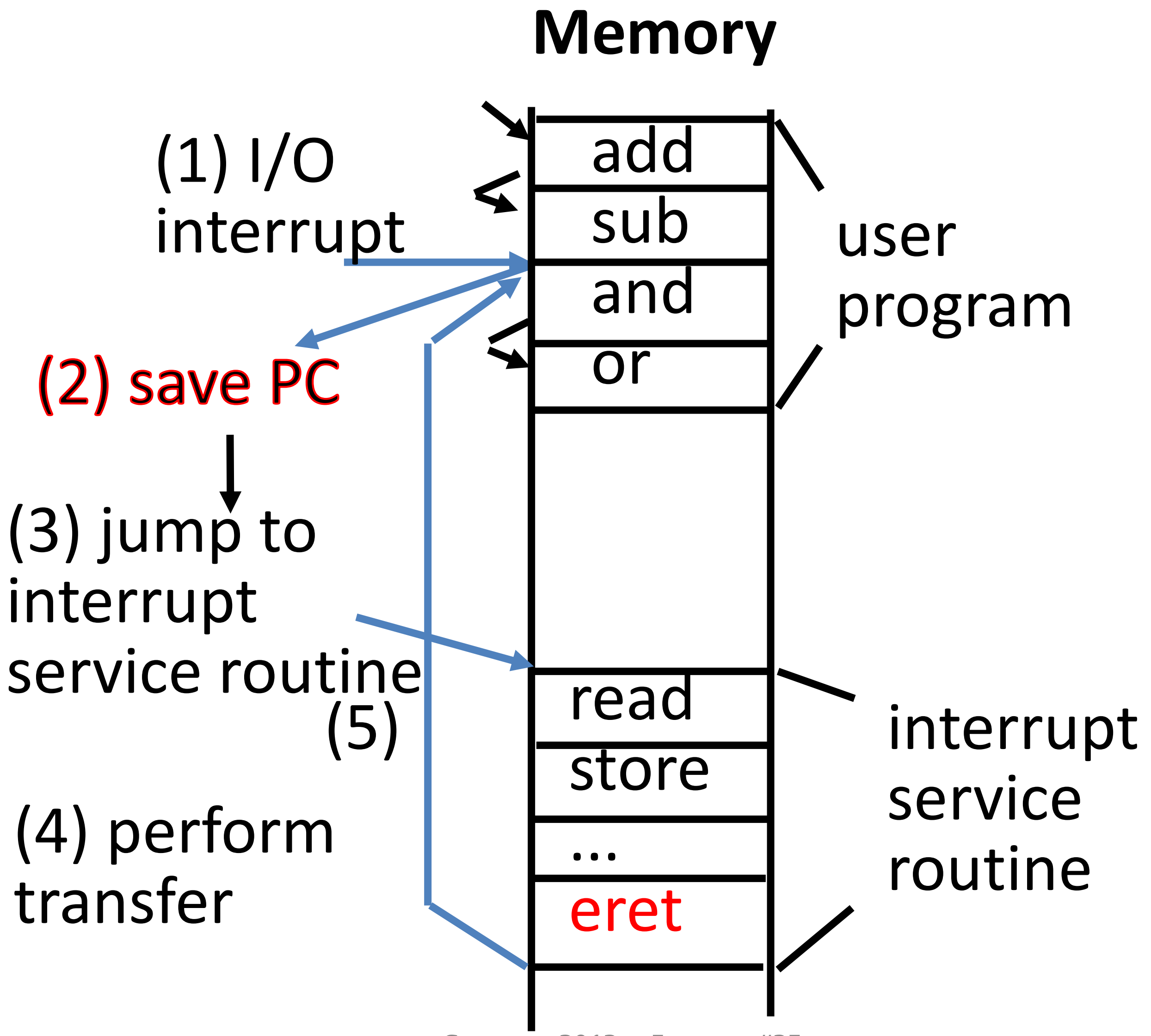
Handler Actions

- Read Cause register, and transfer to relevant handler
- OS determines action required:
  - If restartable exception, take corrective action and then use EPC to return to program
  - Otherwise, terminate program and report error using EPC, Cause register, etc.
    (e.g. our best friend the segfault)

I/O Interrupt

- An I/O interrupt is like an exception except:
  - An I/O interrupt is "asynchronous"
  - More information needs to be conveyed
- "Asynchronous" with respect to instruction execution:
  - I/O interrupt is not associated with any instruction, but it can happen in the middle of any given instruction
  - *I/O interrupt does not prevent any instruction from running to completion*

Interrupt-Driven Data Transfer



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
    - IP[7:2]：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
    - 可以用于实现个性的编码☺

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

EPC寄存器和eret指令

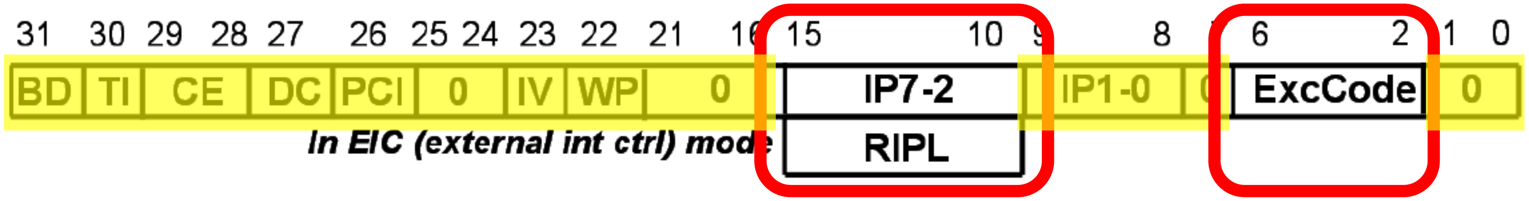
- EPC：保存中断/异常时的PC
  - 以便从中断/异常服务程序返回至被中断指令
- ERET：中断/异常服务程序返回指令

编码	31 26	25 21 20 16 15 11 10 6	5 0
	COP0 010000	80000 1000 0000 0000 0000 0000	eret 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0、算数溢出)时，PC+4将被保存在EPC中。		

CAUSE寄存器

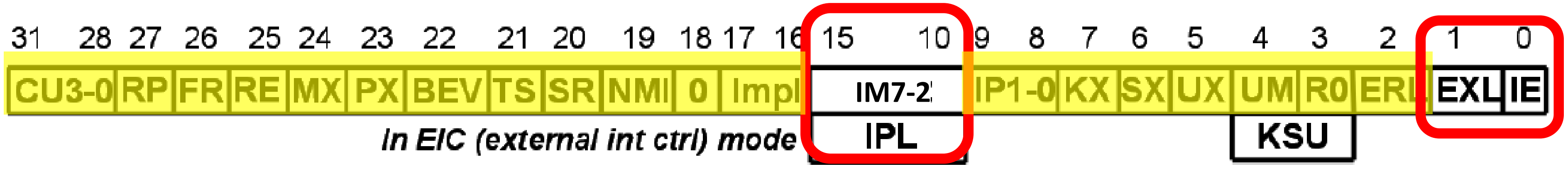
- IP[7:2]：6位待决的中断位，分别对应6个外部中断
  - 记录当前哪些硬件中断正在有效
  - 1-有中断；0-无中断
- ExcCode[6:2]：异常编码，记录当前发生的是什么异常
  - 共计32种
  - 与课程相关的主要异常类型

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

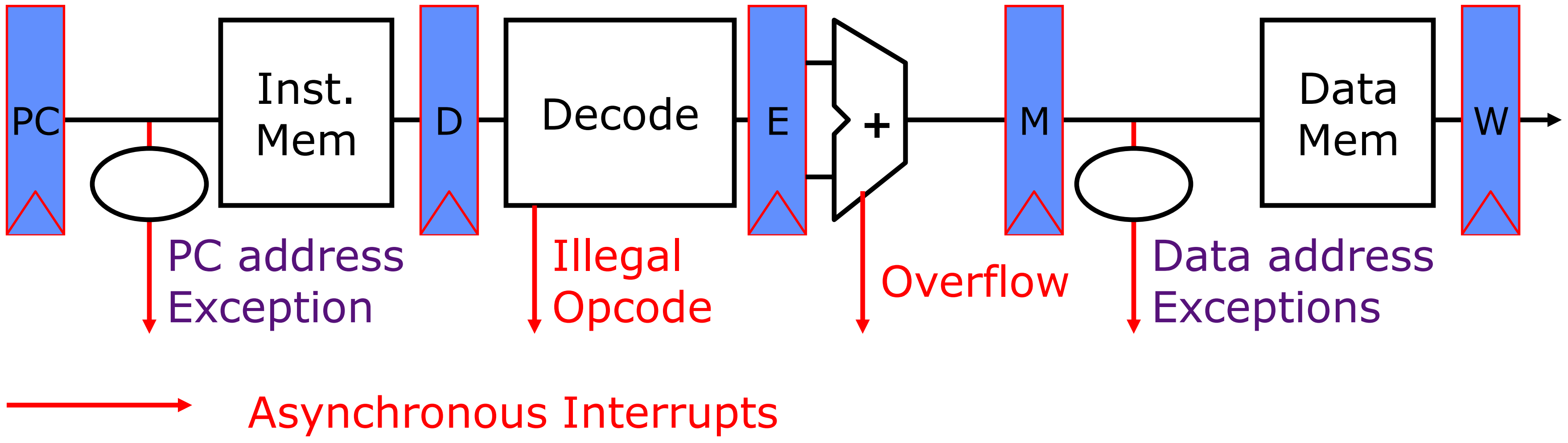


SR寄存器

- IM[7:2]：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
  - 注意：重入（在中断程序中仍然允许再次进行中断）需要OS的配合，重点是堆栈

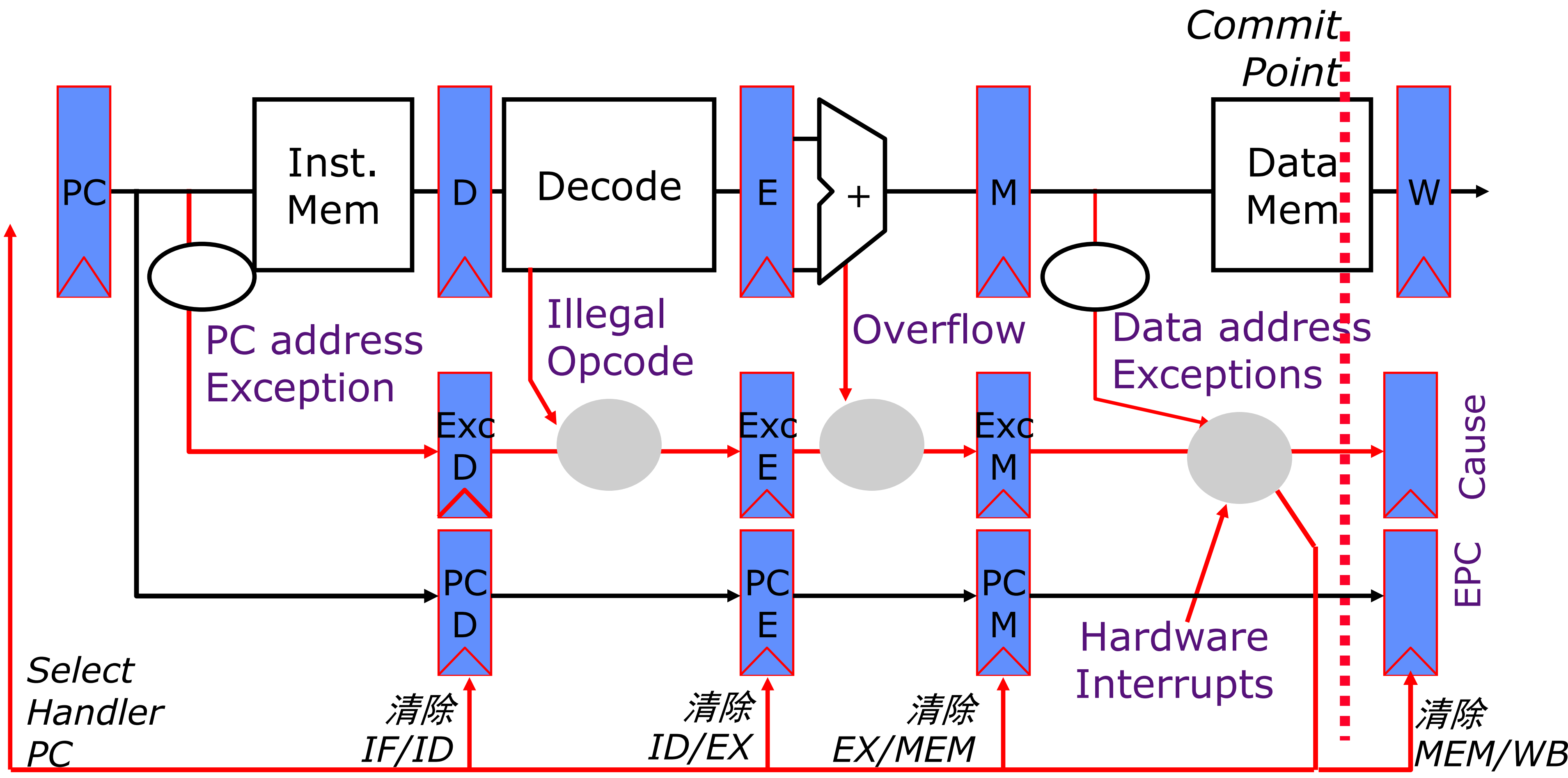


Exception Handling 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

Exception Handling 5-Stage Pipeline (高小鹏修改)

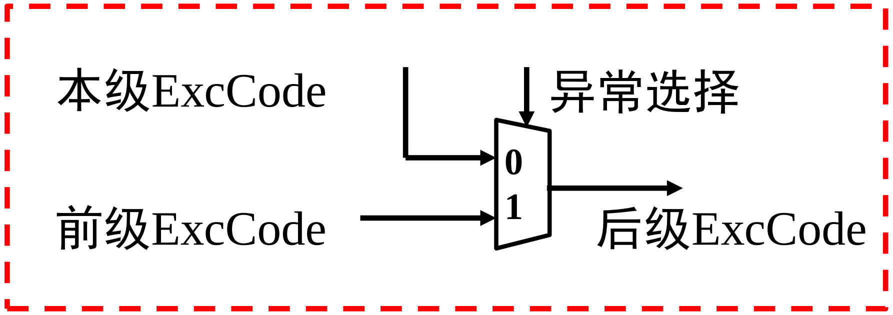


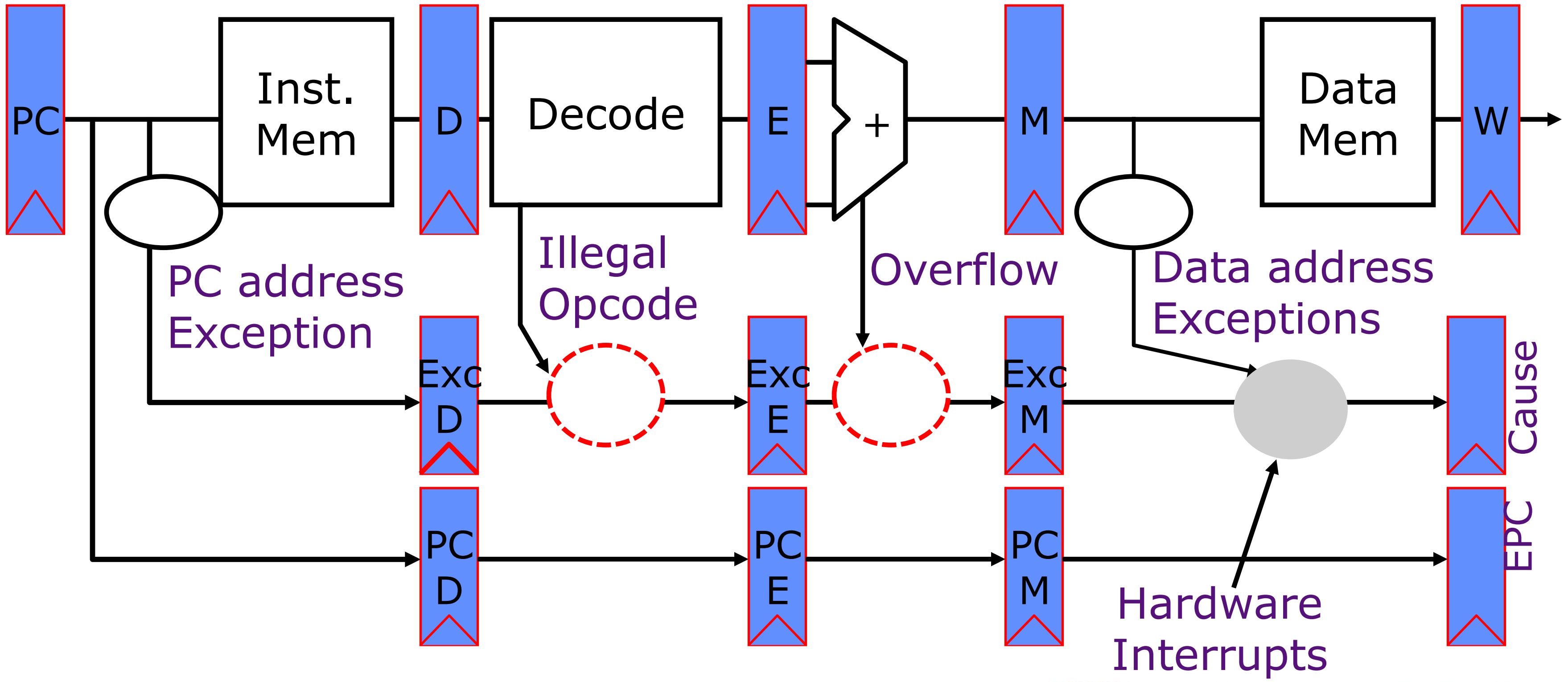
Exception Handling 5-Stage Pipeline

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

硬件实现：传递异常

- 当本级有异常时，则传递本级异常编码
- 硬件中断的优先级高于异常



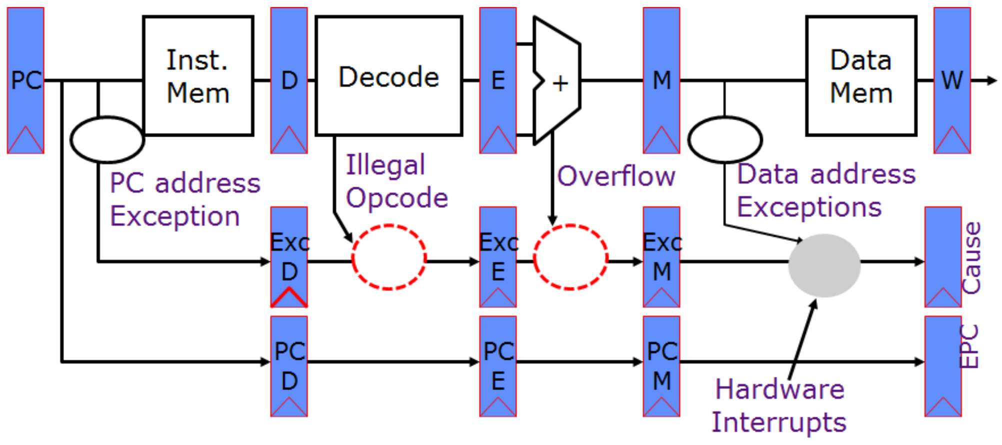


从EPC返回地址角度分析架构设计的合理性

- 架构时序特征：当M阶段检测到异常/中断后，下一个cycle完成的任务如下
  - W级指令：执行完
  - EPC：保存了M级指令的地址
  - PC：PC指向异常/中断地址
- 关于异常
  - 诉求：①异常指令不应执行完，②从handler退出后应返回异常指令
  - 分析：流水线最终处理的异常指令是M级指令，而EPC保存的正是M级指令地址，因此从handler退出后正好返回至M级指令

正确

Q：如果D级存入的是PC+4，那么EPC值与发生异常的指令是什么关系？在指令分析和异常返回时，handler应做哪些处理？

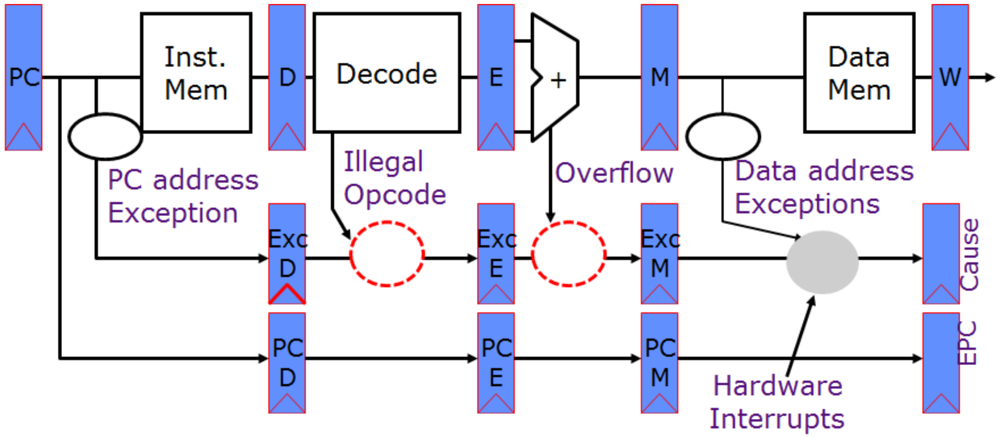


从EPC返回地址角度分析架构设计的合理性

- 架构时序特征：当M阶段检测到异常/中断后，下一个cycle完成的任务如下
  - W级指令：执行完
  - EPC：保存了M级指令的地址
  - PC：PC指向异常/中断地址
- 关于中断
  - 诉求：①任意指令都可以被中断，②从handler退出后应返回被中断指令
  - 分析：进入中断时，M级指令未能执行完，且EPC保存了M级指令地址，因此从handler退出后正好返回至M级指令（即被中断指令）

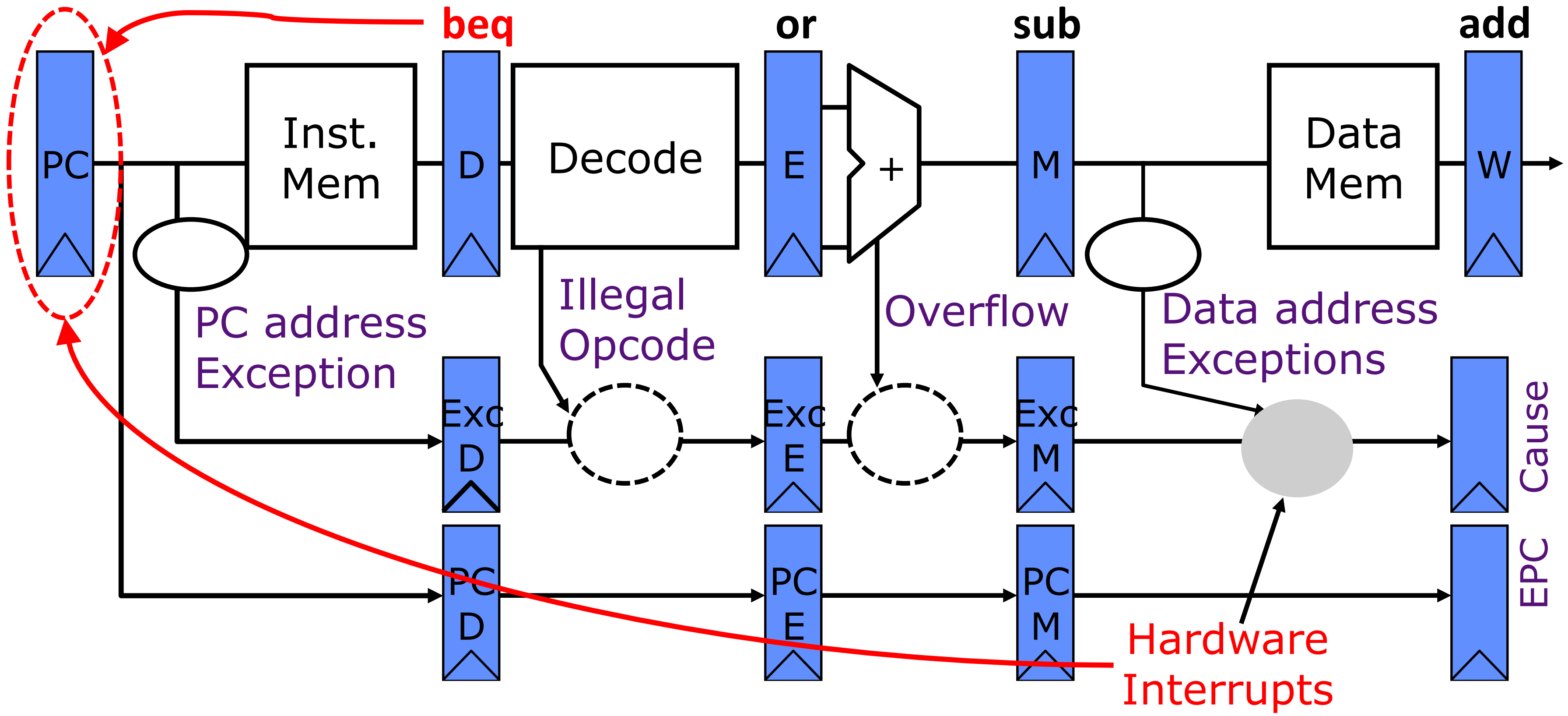
正确

Q：对于中断的分析，是否完备？
H：考虑b类指令的执行（b类指令前移至D阶段）



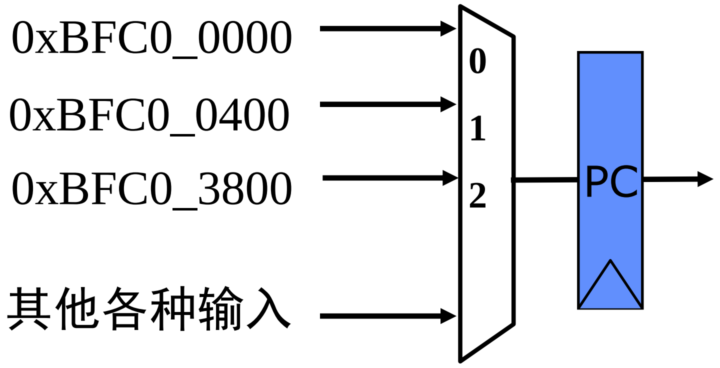
beq在D级时的指令流分析

- 场景：中断发生，同时beq在D级且转移执行
- 冲突：由于产生中断，因此PC在下一cycle应该修改为handler入口地址；由于beq转移，因此PC在下一cycle应该修改为转移地址
- 思路：handler地址的优先级必须高于beq转移地址的优先级
  - PC写入必须考虑优先级（建模PC的MUX控制信号时要注意顺序）



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
- MARS：2种地址
  - default：0x8000_0000
  - compact：0x0000_4180



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
- 1、保存现场
  - 将所有寄存器都保存在堆栈中
- 2、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）
- 3、恢复现场
  - 从堆栈中恢复所有寄存器
- 4、中断返回
  - 执行eret指令

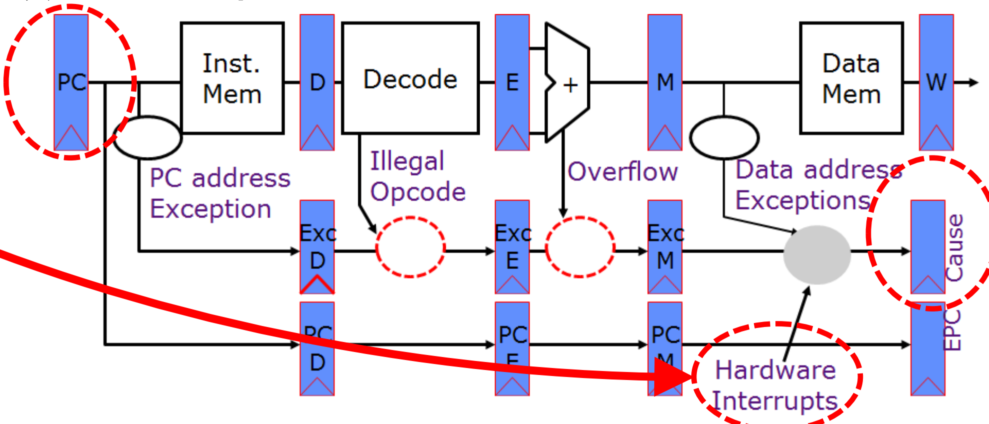
1、3、4：通用代码
2：针对特定设备

中断/异常响应机制：检测中断(1)

- 流水线响应中断的条件
  - 1）6个外部设备中断请求，至少有1个有效且未被屏蔽
    - HWInt[7:2]：来自6个设备，对应设备中断请求
  - 2）全局中断使能（CP0的SR的IE）有效
  - 3）当前不处于中断服务程序中（CP0的SR的EXL）
- 流水线控制器增加一个中断相关的输入信号：IntReq

```
assign IntReq = |(HWInt[7:2] & IM[7:2]) & IE & !EXL ;
```

- 注意：中断/异常同时发生时，中断优先级应高于异常优先级
  - PC写入/CAUSE写入：中断的优先级高于异常



中断/异常响应机制：控制器(2)

- 处理：保存PC/跳转/关中断
- ①保存：将M级指令的PC值和ExcCode保存在EPC和Cause中
- 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

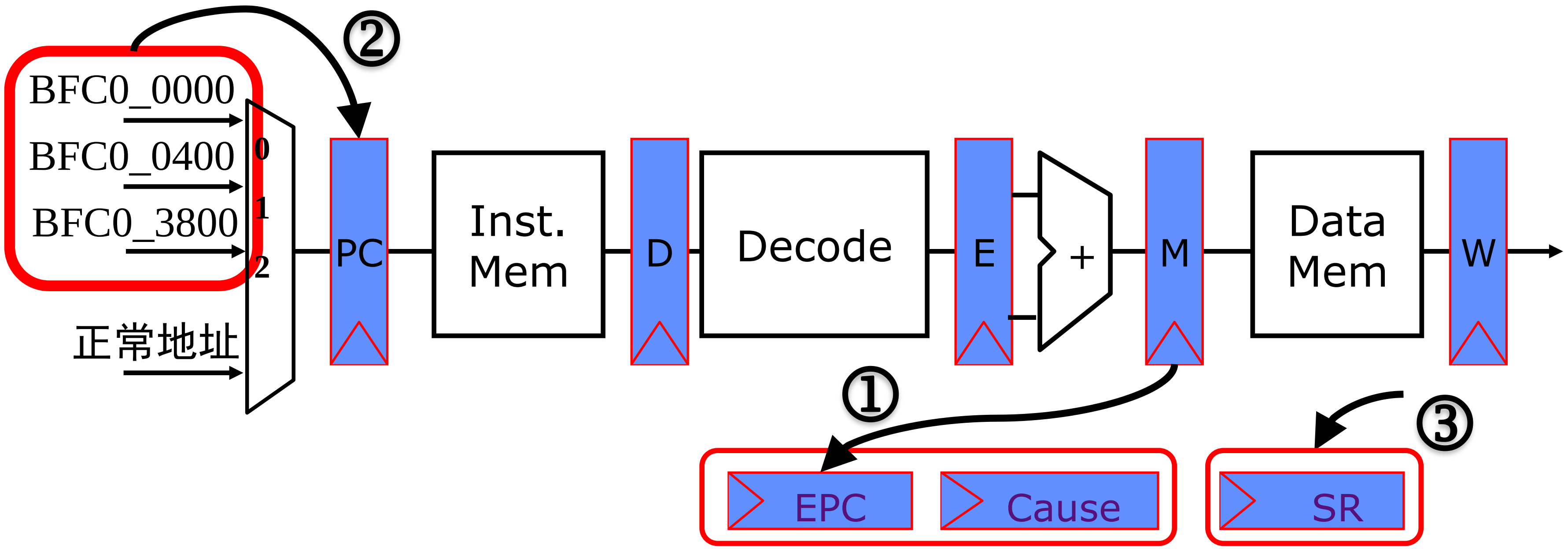
实现要点：

3个步骤在同一周期完成

实现方法：

PC/EPC/ExcCode/EXL

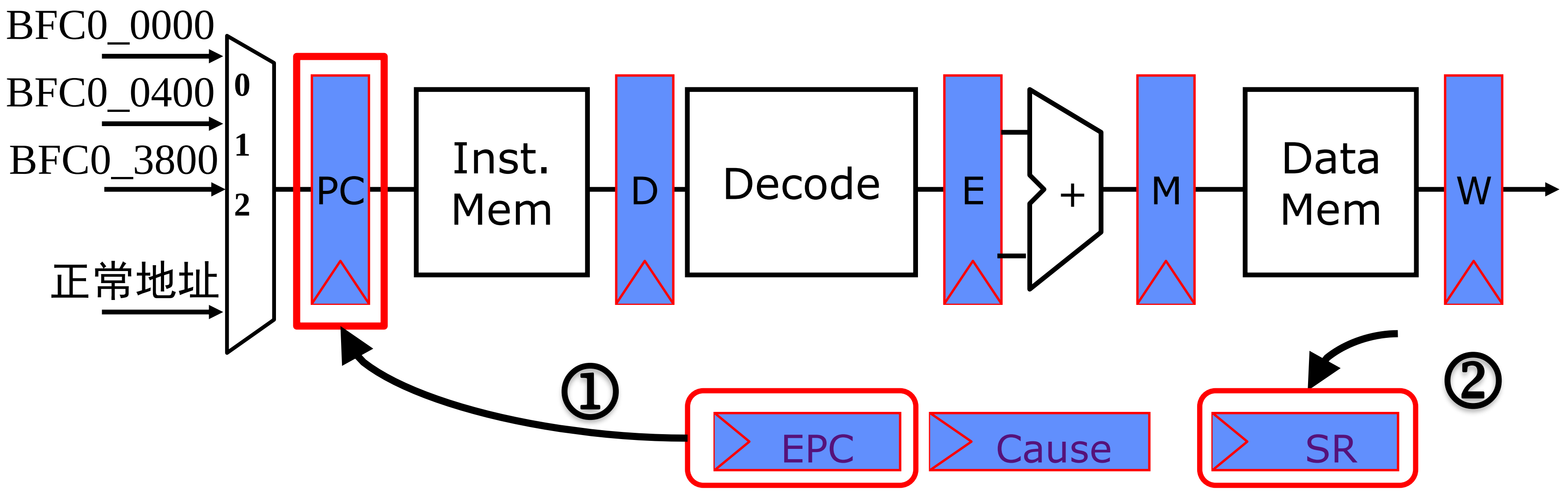
写使能同时产生



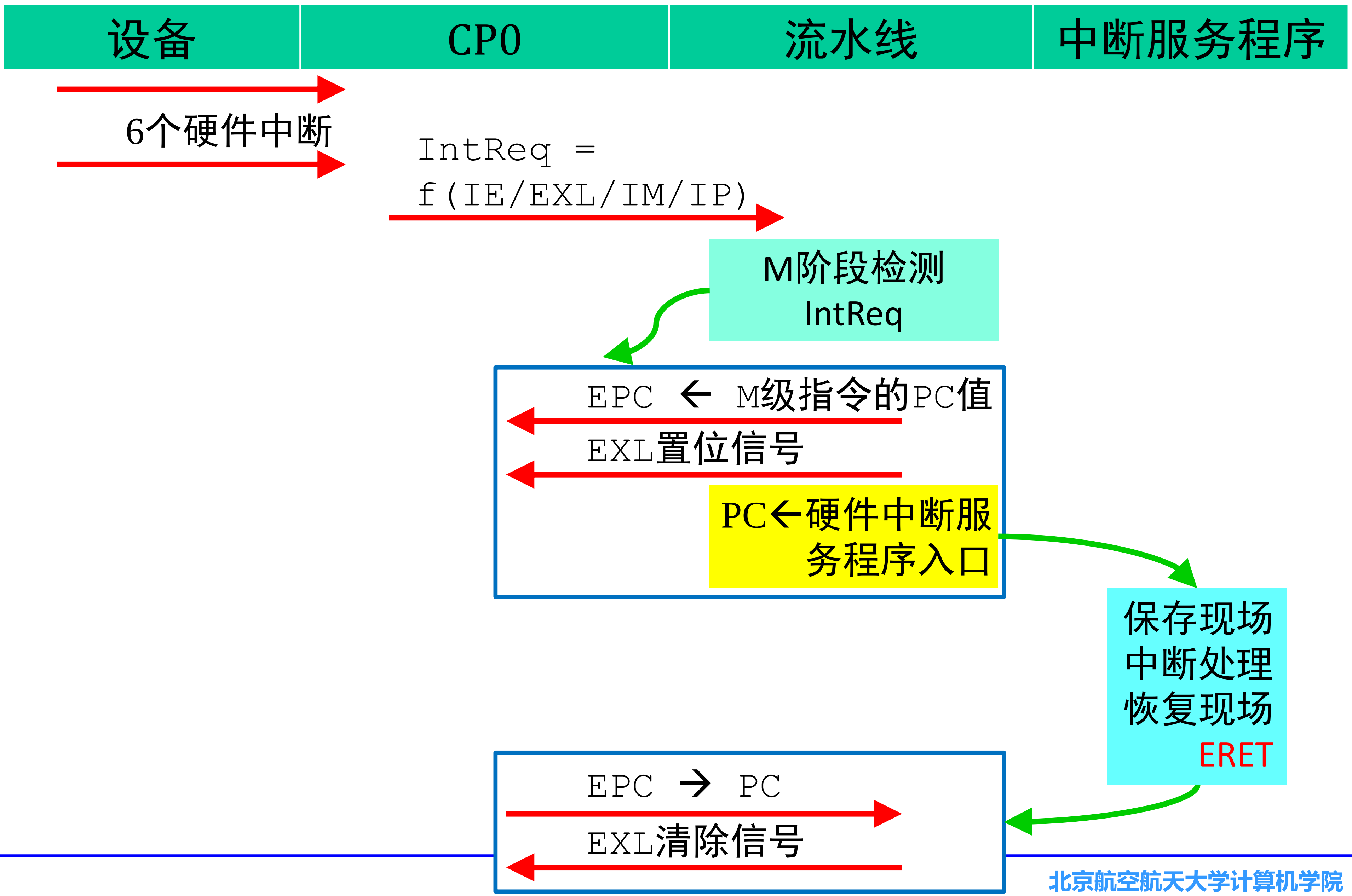
中断响应机制：ERET指令(3)

- 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
- 注意：防止eret后继指令被误加载进入流水线
  - 软件方法：eret后添加nop
  - 硬件方法：清空D级1次(相当于插入nop)

实现要点：
2个步骤在同一周期完成
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



Interrupt-Driven I/O Example (1/2)

- Assume the following system properties:
  - 500 clock cycle overhead for each transfer, including interrupt
  - Disk throughput of 16 MB/s
  - Disk interrupts after transferring 16 B
  - Processor running at 1 GHz
- If disk is active 5% of program, what % of processor is consumed by the disk?
  - 5% × 16 [MB/s] / 16 [B/inter] = 50,000 [inter/s]
  - 50,000 [inter/s] × 500 [clocks/inter] = 2.5×10^7 [clocks/s]
  - 2.5×10^7 [clocks/s] / 10^9 [clock/s] = **2.5% busy**

Interrupt-Driven I/O Example (2/2)

- 2.5% busy (interrupts) much better than 40% (polling)
- **Real Solution:** *Direct Memory Access (DMA)* mechanism
  - Device controller transfers data directly to/from memory without involving the processor
  - Only interrupts once per page (large!) once transfer is done

提纲

- 输入输出
- 异常/中断
- 协处理器

协处理器指令及用途

- 指令：MFC0、MTC0
  - 软件不能直接修改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于的状态
  - 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：操作系统中将用于多任务切换

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M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：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]} ;
```

- exl：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 ;
```

reg [5:0] im与reg [15:10] im是等价的，但后者可读性更好

设计CP0：Cause

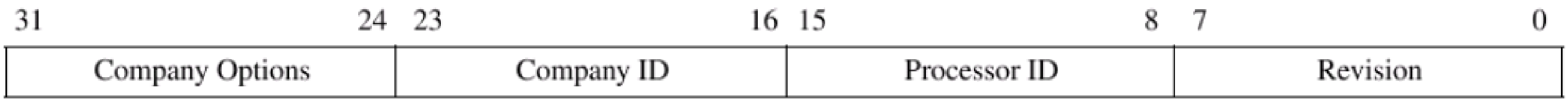
- Cause：只需定义6位寄存器，在clock上升沿不断的保存外部6个中断(HWInt[5:0])
  - `reg [15:10] hwint_pend ;`
- Cause整体表示为：
  - `{16’b0, hwint_pend, 10’b0}`

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

- 目前可以任意选则用一个4字节的编码值，如
  - 0x1234_5678

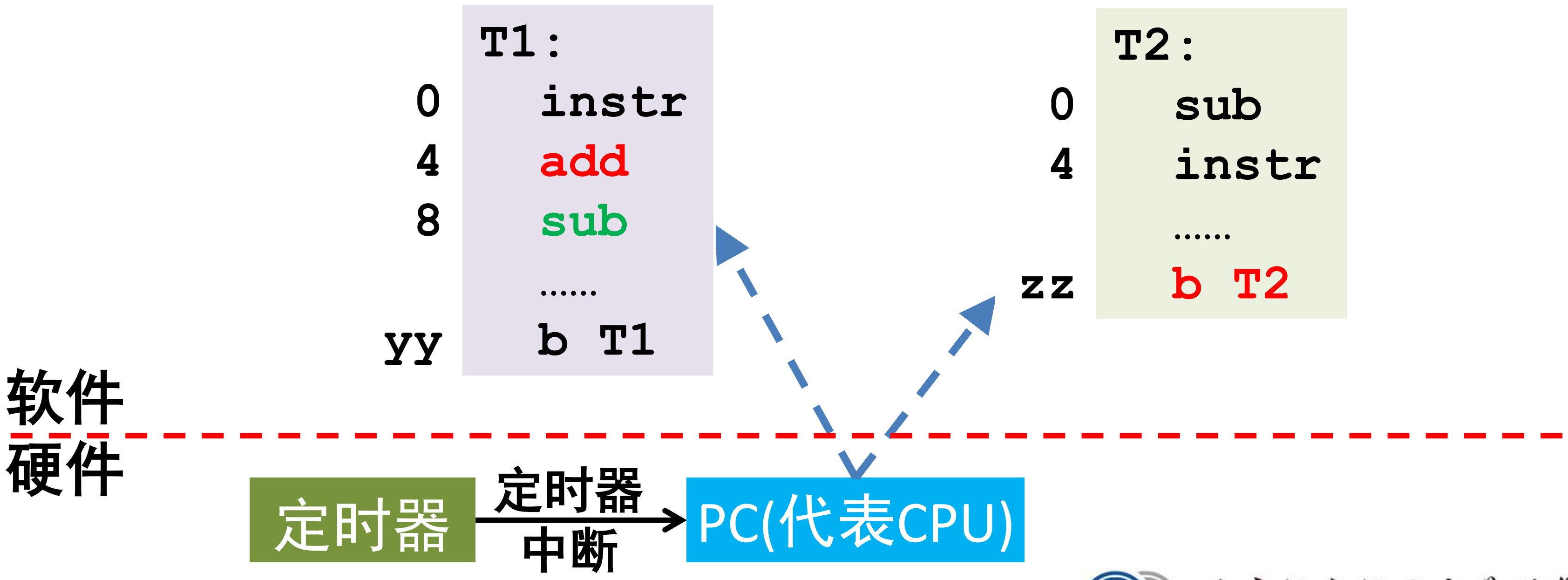
设计CP0：输出CP0寄存器

- 除了SR/Cause/EPC/PRId外，不用的寄存器一律输出0
- 可以设计5选1的MUX，或者用行为描述（如样例代码）

```
assign DOut = (Sel==12) ? {16'b0, im, 8'b0, exl, ie} :
              (Sel==13) ? {xxxxxxxxxxxxxxxxxxxxxxxxxx} :
              (Sel==14) ? EPC                          :
              (Sel==15) ? PrID
                          32'b0 ;
```

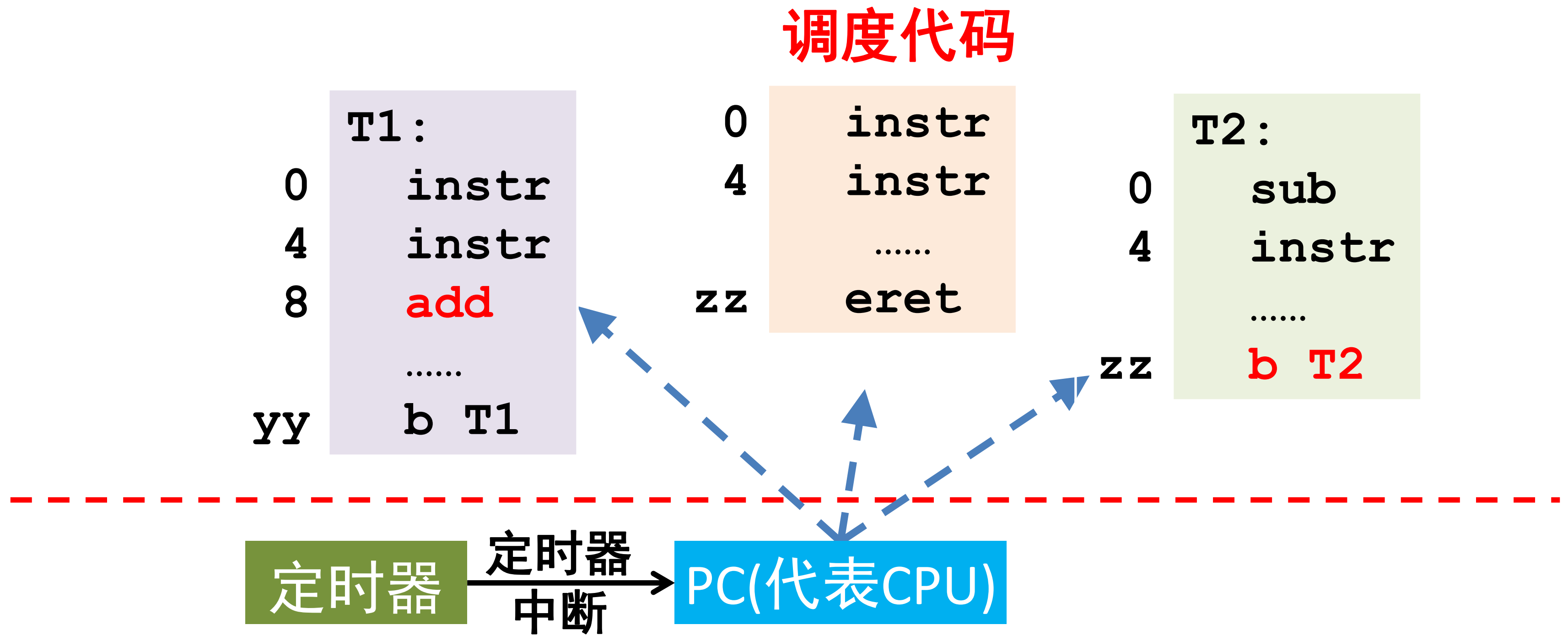
OS任务切换基本机制探讨

- 场景：T1和T2两个任务（均为无限循环）定时切换
  - 定时器定时产生中断（这就是时间片）
  - 中断产生后（时间片到了），如果PC当前在执行T1则切换至T2，反之亦然
- 切换的具体说明（以从T1切换至T2为例）：
  - 时间片到时，PC从指向T1的add被修改为指向T2的b，然后开始执行T2
  - 当下个时间片到来时，PC要恢复执行T1的add的后继指令（sub）



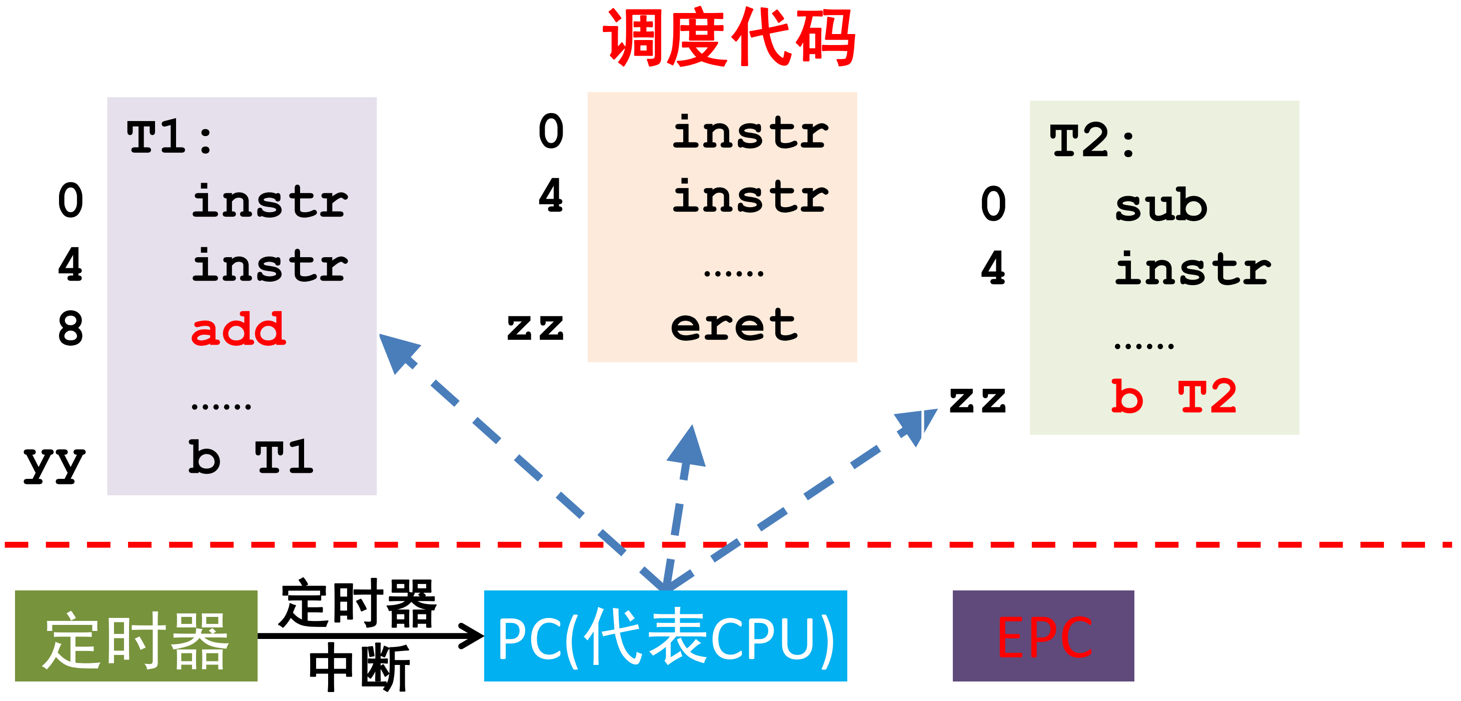
OS任务切换基本机制探讨

- 分析：系统中必须有独立的任务调度代码（第3段代码）
  - 这段代码必然与定时器中断服务程序相关（不妨假设就是）



OS任务切换基本机制探讨

- Q：如何在调度代码(中断服务程序)中实现T1与T2间的切换？
- H：关键要素在于利用EPC
  - EPC是任务切换的核心所在（其他寄存器都是次要因素）



调度代码
T1:
0 instr
4 instr
8 add
……
yy b T1
0 instr
4 instr
……
zz eret
T2:
0 sub
4 instr
……
zz b T2
定时器
定时器
中断
PC(代表CPU)
EPC