P7 拓展研究报告

14231010 武洋阳 2015.12.31

一、为什么会在 P7 中取消异常的处理。

(一)中断、异常

在 MIPS 系统中，中断、自陷、系统调用统称为异常，中断是由 CPU 之外的事件，即来自于真实“连线”上的输入信号，产生的信号。

常见的异常：

表 3.2: ExcCode：异常的不同类型

ExcCode	助记符	描述
0	Int	中断
1	Mod	存储操作时，该页在 TLB 中被标记为只读。
2	TLBL	没有 TLB 转换（读写分别）。也就是 TLB 中没有和程序地址匹配的有效入口。
3	TLBS	当根本没有匹配项（连无效的匹配项都没有）并且 CPU 尚未处于异常模式—— SR(EXL) 置位——即 TLB 失效时，为高效平滑处理这种常见事件而采用的特殊异常入口点。
4	AdEL	（取数、取指或者存数时）地址错误：这要么是在用户态试图存取 kuseg 以外的空间，或者是试图从未对齐的地址读取一个双字、字或者半字。
5	AdES	
6	IBE	总线错误（取指或者读取数据）：外部硬件发出了某种出错信号；具体该怎么做与系统有关。因存储而导致的总线错，只能作为一个为了获取要写入的高速缓存行而执行的高速缓存读操作的结果间接出现。
7	DBE	
8	Syscall	执行了一条 syscall 指令。
9	Bp	执行了一条 break 断点指令，由调试程序使用。
10	RI	不认识的（或者非法的）指令码。

11	CpU	试图运行一条协处理器指令，但是在 SR(CU3–0) 中并没有使能相应的协处理器。 具体说，就是当 FPU 可用位 SR(CU1) 没有置位时从浮点操作得到的异常；因而是浮点仿真开始的地方。
12	Ov	自陷形式的整数算术指令（比如说 add 但 addu 不会）导致的溢出。C 语言程序不使用溢出-自陷指令。
13	TRAP	符合了 teq 等条件自陷指令的某一条。
14		目前未用。在有些拥有 L2 高速缓存的老式的 CPU 上，当硬件探测到可能的高速缓存重影时使用这位，在 4.12 节对此有解释。
15	FPE	浮点异常。（在某些很老的 CPU 上，浮点异常以中断形式出现。）
16–17	-	定制的异常类型，与具体实现相关。
18	C2E	来自协处理器 2 的异常（如果有的话，就是对指令集的一种定制的扩展）。
19–21	-	保留给未来扩展使用
22	MDMX	试图运行 MDMX 指令，但是 SR(MX) 位没有置位（很可能该 CPU 没有实现 MDMX）。
23	Watch	load/store 的物理地址匹配了使能的 WatchLo/WatchHi 寄存器。

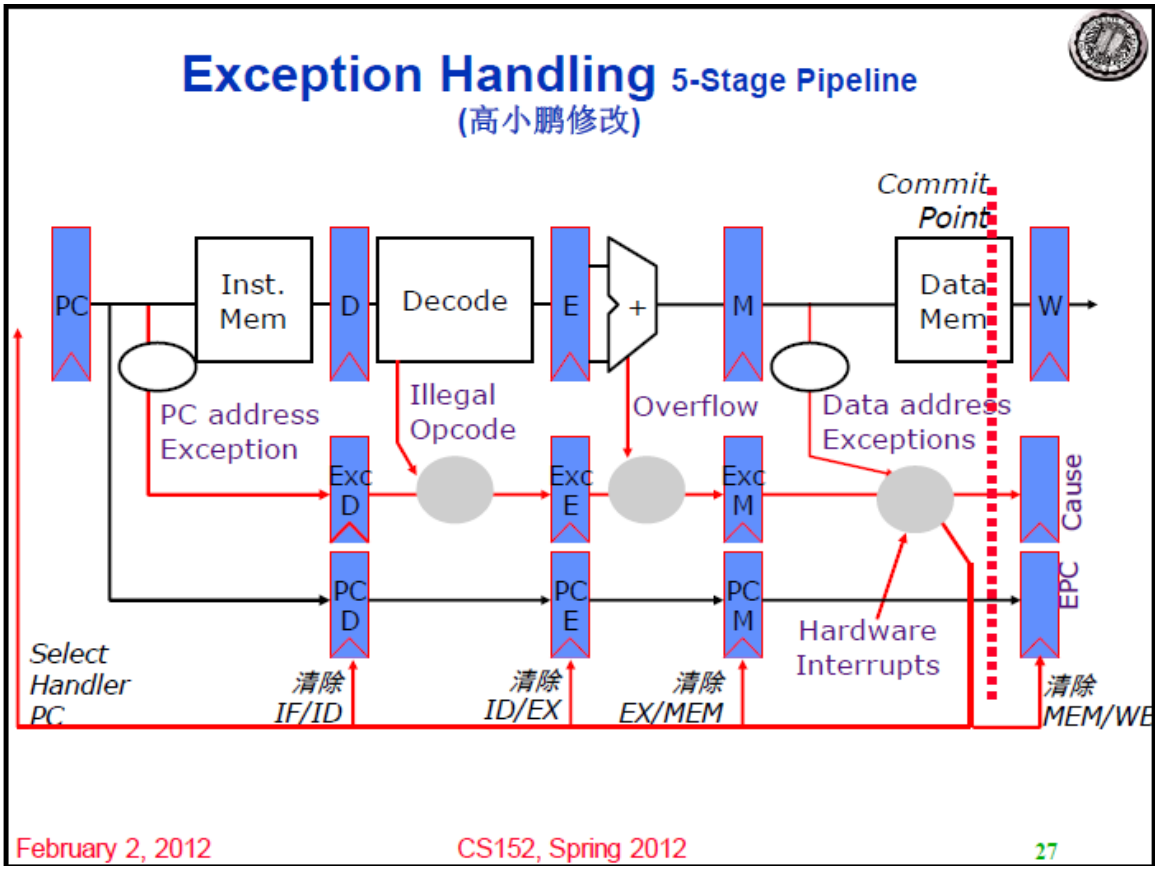
表 3.2 续

ExcCode	助记符	描述
24	MCheck	机器检查——CPU 监测到了 CPU 控制系统中的灾难性的错误。MIPS 公司的有些核当 TLB 中加载了匹配同一个程序地址的第二个转换项时发出该异常。
25	Thread	线程相关的异常，这在附录 A 中有描述。还有一个寄存器域，VPEControl(EXCPT)，提供有关线程异常的更多细节。
26	DSP	试图运行一个 DSP ASE 指令，但是要么该 CPU 不支持 DSP 指令，要么SR(MMX) 没有设置成使能 DSP。
27–29	-	保留给将来的扩展。
30	CacheErr	在取指、读数、或者高速缓存填充时，核内某个地方发生奇偶校验码/ECC 纠错码错误。这种错误有它们自己的（位于非缓存空间的）异常入口点。事实上，在 Cause(ExcCode) 中从来看不到这个值；但是这个表中的某些编码，包括这个在内，在 EJTAG 调试单元的调试模式下是可见的——参见第 12.1 节，特别是该节关于 Debug 寄存器的说明。
31	-	现在未用，但是历史上用过，跟上面的位 14 差不多。

要回答为什么会在 P7 中取消异常这个问题，我将它转化为：如果 P7 要实现异常处理，会遇到什么问题。

如果加入异常处理的话，规范问题将是最大的问题。

首先，是包含哪些异常的问题。同学们首先应该统一：自己的 CPU 应该包含对于哪些异常的处理。从上述常见的异常来看， 我们只能处理其中取址错误、溢出、不认识的指令码这三个异常。在 MIPS 体系结构指导书中，我找到了这几个异常的处理。



在这张图中，PC address Exception, Illigal Opcode, Overflow, Data address Exceptions, Hardware Interrupts 分别对应了 5 中不同异常。这五种异常在上图中的表 3.2 中对应的 ExcCode 分别是：4(AdEL);10(RI);12(Ov);5(AdES);0(IntReq).

如何实现对异常的处理，在第二个问题的回答中会有详细的描述。我们先讨论对每一个异常的解决。我认为，异常解决的规范问题是 P7 中没有涉及异常的最主要的问题。

每个异常的处理程序是由高级的程序设计语言写成的。在 Mars 中，如果出现了异常，原程序会终止然后跳到 Exception Handler, 操作系统层面需要在遇到这样的异常的情况下有一种应对机制，所以在 Exception Handler(下文简称”eh”)中的程序是必不可少的。相比于中断来说，异常实际上是某条指令在流水线的某个阶段出现了问题。他们的本质区别是：一个是 CPU 内部传出的信号，一个是外部。在所有的资料中都指出通过异常向量，将不同的异常指向不同的地址来处理，而并未明确指出不同的异常的具体处理办法。猜测原因是：不同的系统对问题的处理方式确实有差别，且这方面的问题设计过多操作系统等方面的知识，不宜在计算机组成原理书中过多描述。因此，如果在 P7 中加入异常处理，全班同学就需要编写不同的，对于不同异常的处理程序，这就会使得 P7 最后的输出结果十分不便于检查。

另外一个很原始的原因就是：在处理中断时，已经新增了 CP0 模块并且对整个 CPU 作了很大调整，如果要加入异常处理的话，同学们可能会压力过大。而且中断已经使同学们对 MIPS 的体系结构和输入输出的一些基本思想有了认识，加入异常的性价比比较低，涉及的技术层面问题远远大于思想层面问题，得不偿失。

二、在 P7 基础上做何改动能够支持异常？

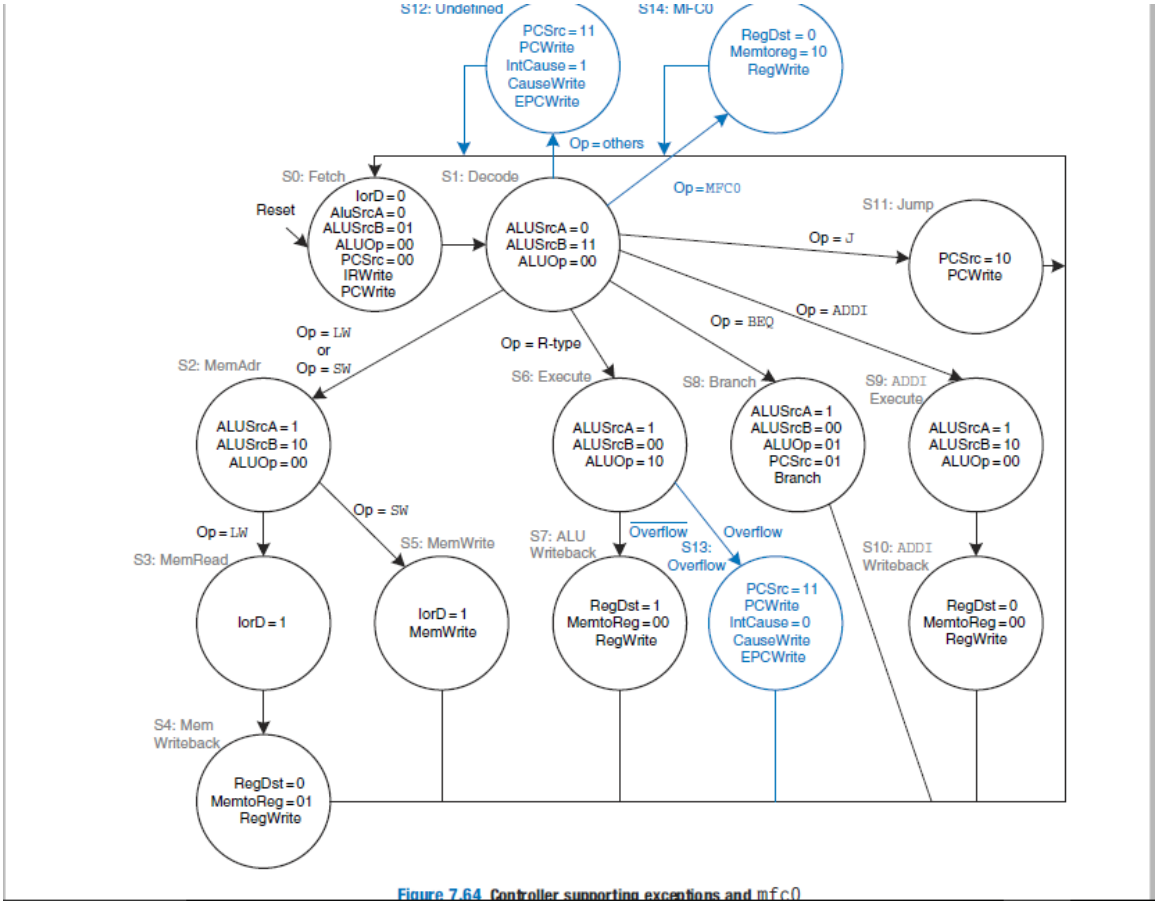


Figure 7.64 Controller supporting exceptions and mfc0

（一） 整体框架

1. 在每一个控制模块中加入判断异常的语句。
2. 在每一级流水线中增加 ExcCode，传递到 W 级流水线判断异常。
3. 根据异常的优先级，进入优先级最高的异常的处理环节：保留现场、执行异常处理程序、恢复现场。

（二） 图示 MIPS_CPU 顶层架构的更改

（三） 细节

1. PC 值如果超过了 IM 的内存，取出的指令都是 X 未知值，流水线会有一段时间充满了未知数值。对于寄存器和 DM 中存的数值不会有影响。在异常处理的程序中可以考虑将 CPU 初始化一下（软件层面解决的问题）。
2. Op 码如果不存在，那么在后续指令的执行过程中，Controller 的输出会是未知，每个模块的输出也会是未知，在异常处理的程序中也要考虑将 CPU 初始化。
3. 如果 ALU 或者 Mul/Div 模块中发生溢出，此时计算出的结果是错误的，会逐级向下传递，此时需要考虑以下情况：ALU 计算出的值是将要存入寄存器的，在 W 阶段数据即将写入寄存器

时，如果不加更改，在清空寄存器之前，错误的计算结果就已经被写入寄存器了。解决办法：将 Cause 寄存器中的 ExcCode 传给 W 阶段的 Controller,如果 ExcCode 的值是 12(0v)，且由 Op 码和 Function 码得知该指令时 cal_r 或者 cal_i 或者 jalr 类型的指令时，RegWriteW 的值便是 0，不再写入寄存器。二、ALU 计算出的结果是 DM 的地址。因为 DM 将输入的地址当作无符号数，在 ALU 中已经溢出，则这个数已经超过了 DM 的地址范围。此时只需小心判断 Overflow 和 Data address Exceptions 这两种异常的优先级即可。

4. 如果 DM 的地址出现了异常，DM 内已经存储的值不会受到影响。

（四） 部分代码

```
assign EPC = epc;
assign IntReq1=(HWInt[15] & im[15] | HWInt[14]&im[14] | HWInt[13]&im[13] | HWInt[12]&im[12] | HWInt[11]&im[11] | HWInt[10]&im[10]) & ie & !exl;
assign Exception = ExcCode!=0 & ie &!exl;//异常信号
assign IntReq = IntReq1 | Exception;//生成最终的中断信号，控制当前PC保存至EPC；传入Hazard模块，在Hazard模块中执行清除寄存器。


assign PCOut = Exception & IntReq1 ? 32'h0000_4180:
               !Exception & IntReq1 ? 32'h0000_4180:
                Exception &!IntReq1 ? `Exception_Handling_Code;
```

在 CP0 中的更改。PCOut 中体现了中断的优先级高于异常。

除此之外，各级流水线中还增加了 ExcCode 寄存器，在此不必赘述。

在图示中还展示了每一级 ExcCode 寄存器前增加的 Mux 多路选择器，Mux 多路选择器的选择信号提供源是每一级的控制器（分布式控制器），控制器根据优先级决定是保留上一级提供的 ExcCode 还是将当前流水线阶段产生的新异常编号覆盖至之前的。

将异常信号归并至 IntReq，则当从外部程序返回时遇到的冲突也都可以与中断时的情况类似解决。

三、在 P7 基础上做何改动能够支持中断嵌套？

在《See Mips Run》这本书中对于嵌套异常的描述：

在许多情况下，你可能想要在你的异常处理例程中允许（或者无法避免）进一步的异常；这称为嵌套异常。

如果处理不慎，这可能导致混沌；被中断的程序的要害状态保存在 **EPC** 和 **SR** 中，你必须预计到其它的异常可能会冲掉其值。除出了一个非常特殊的嵌套异常之外，使能其它异常之前都必须保存这些寄存器的内容。此外，一旦重新使能异常，你就不能再依赖为异常处理保留的通用寄存器 **k1** 和 **k2** 了。

一个能够允许嵌套异常的异常处理程序，必须使用某些内存区域来保存寄存器值。所用的数据结构常常叫做 异常帧；嵌套的多个异常帧通常安排在一个堆栈上。

每次异常都要消耗堆栈资源，所以不能容忍任意嵌套深度的异常。大多数系统给每种异常一个优先级，并且安排成当正在处理某个异常时，只允许更高优先级的异常。这样的系统只需要有跟优先级同样个数的异常帧就行了。

你可以避免一切异常；中断可以用软件单个屏蔽掉以满足优先级规则，可以用 **SR(IE)** 一次性屏蔽全部中断，或者可以通过（后来的 CPU 才有的）异常级位隐式屏蔽。其它类型的异常可以通过适当的软件约束来避免。例如，在核心态（为大多数异常处理软件所用）不可能发生特权级违反异常，程序可以避免寻址错误和 TLB 未命中的可能性。当处理高优先级异常时这样做是绝对必要的。

在 P7 基础上实现异常嵌套分为两部分：一、分清各种异常的优先级。二、设置堆栈存储 EPC 的值。

一、分清各种异常的优先级：在 CP0 中设置每一种异常的优先级，如果遇到的异常的优先级大于当前正在处理的异常，则将当前的 EPC 保存到堆栈中，执行下一级异常处理程序。

二、关于堆栈：这里的堆栈是由软件实现的，即每次进入异常处理程序的时候将当前的 EPC 保存至堆栈，下一级的 EPC 的值会覆盖当前的 EPC 的值。在异常处理程序 ERET 之前，要将堆栈中的 PC 值通过 LW 和 MTC0 语句转移到 EPC 中然后返回。

综合上述思考，至少在这种嵌套的实现方法中，只需要在 CPU 中赋予不同的异常不同的优先级，其他的事情都是软件层面需要解决的问题。CPU 在每一次的异常处理过程中都和 P7 的处理过程完全相同。