要了解一个架构的所有内容，需要从以下几个方面去着手研究：

硬件部分：

1. 寄存器结构，包括通用数据和控制寄存器；
2. 指令集及其实现，特别是涉及到控制转移的指令，要着重分析；
3. 存储系统，一个架构对存储系统的要求是什么样的，有什么限制，其默认的存储器系统有哪些特点；

软件部分：

1. 内部陷阱，外部中断的种类，产生原因；对应的handler是怎么操作的？硬件在这个过程中都做了哪些工作？中断或陷阱程序的进入是由硬件来实现的（如51的硬件LCALL），还是软件？
2. 子程序调用的过程中，参数的传递和结果的返回，也就是堆栈的处理方式；以及在这个过程中，寄存器使用的规定，caller需要保护哪些寄存器，callee又需要保护哪些寄存器；
3. 架构对指令执行顺序的规定；
4. 在不同模式之间切换时的操作是怎么实现的，硬件和软件分别做出了怎样的操作？

整个系统：

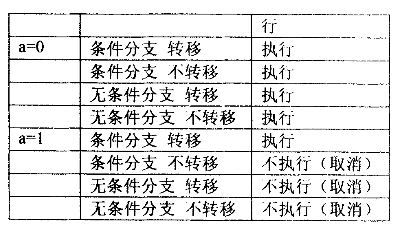
1. 对应架构下的开发工具链；编译-汇编-链接-加载，以及一些打包，反汇编等工具；
2. 操作系统相关的东西；

SPARC V8体系结构：

1. 最大的特点，寄存器窗口和延迟控制转移；前者优化子程序调用，后者优化流水线性能；

寄存器窗口：save，restore，rett，和trap进入的时候会改变寄存器窗口；

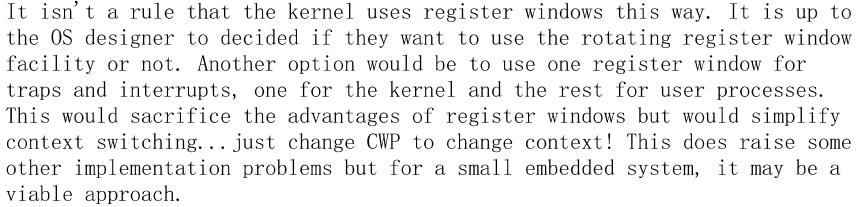
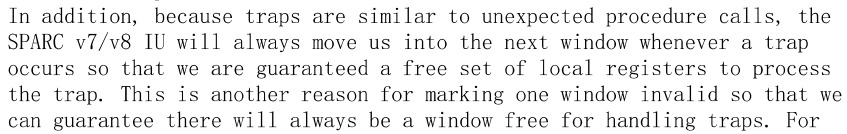
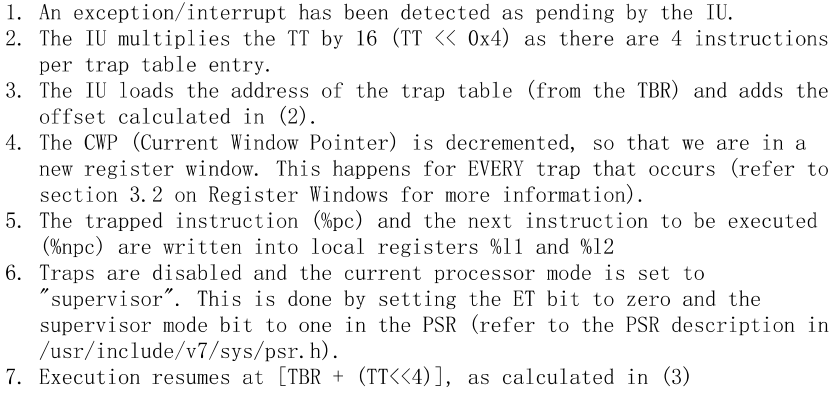
延迟控制转移：



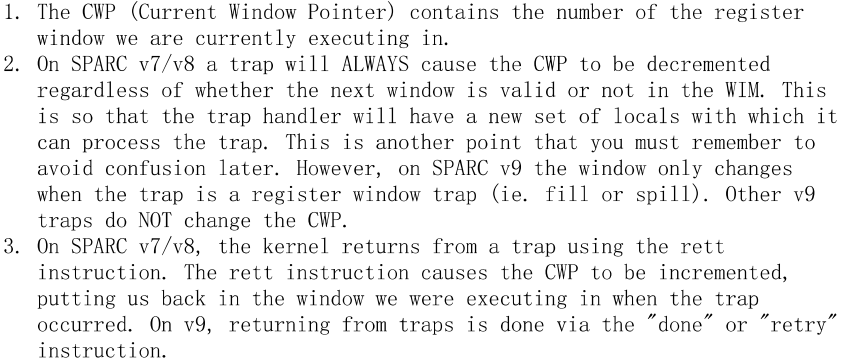
1. trap的类型及handler

三种，精确，延迟的和中断；其中，精确的是由指令执行产生，且没有任何cpu的状态被改变；

以下内容都是在SUNOS系统下的描述。Sparc v8的Trap的硬件处理过程如下：



在v8中的trap处理：

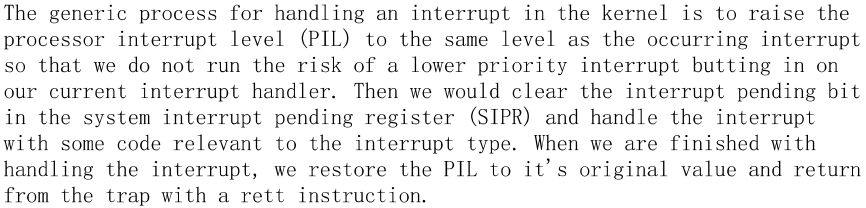


这里就有一个问题，也就是这里的cwp的decrease到底是硬件还是软件实现的？如果是硬件实现的，那么在trap的执行过程中的嵌套trap怎么处理？如果是软件实现的，也就是save指令，那么这条指令是在trap handler中调用的么？这条指令将触发window overflow trap，这又是怎么处理的？

看window overflow的handler程序，感觉trap的窗口切换是硬件实现的；

那么嵌套呢？可以由trap的硬件处理过程来进行解释；当然，这一部分内容，只要看到了任何一个trap的handler就可以明白了，在不同的硬件实现和操作系统下，可能会有所不同；

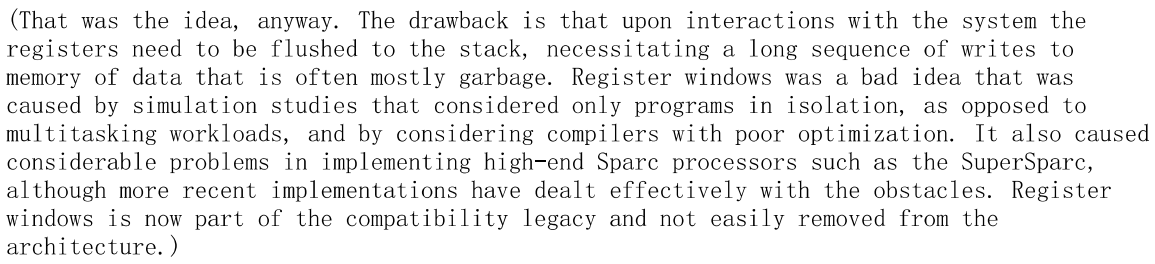
中断的处理大致如下：



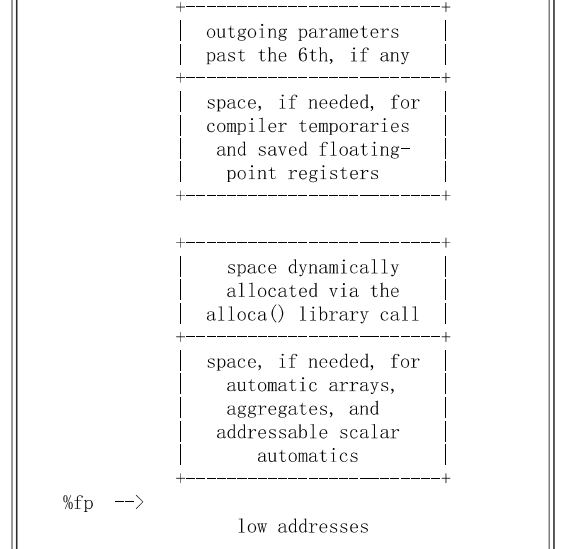
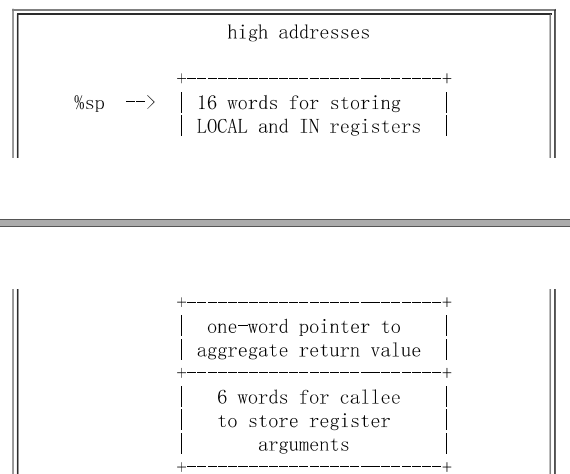
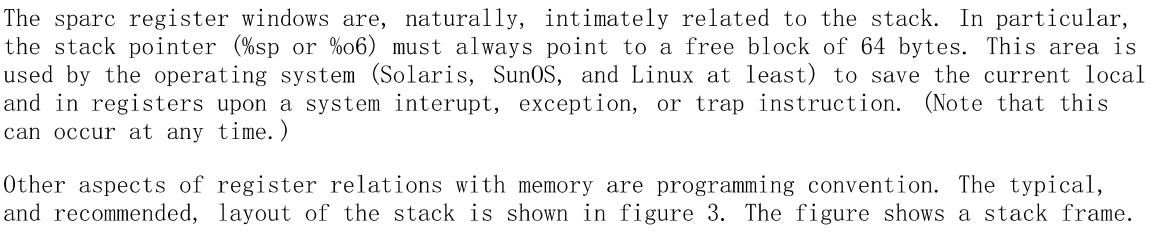
这种处理方式有一个中断lock的问题；同时，对每个中断，可以注册多个ISR，顺序调用，有专门的标志位用来表示该中断是否已经被服务了，在执行每个ISR之前，都会去查询该位，如果已经被服务，那么后序的ISR就不会被调用了；

不同指令之间如果存在数据依赖关系，那么在直接书写汇编程序的时候，就需要考虑到流水线中的问题，也就是数据什么时候能被后续指令处理；比如如果对wim寄存器进行修改，那么必须得有至少三个周期的延迟，才能调用restore或者是任何访问这个寄存器的指令；那么这三个周期的延迟，可由三条nop指令来实现；

寄存器窗口也可能在系统的其他方面带来一些问题：



栈桢的结构：



对于procedure的开头和结尾的代码结构：

