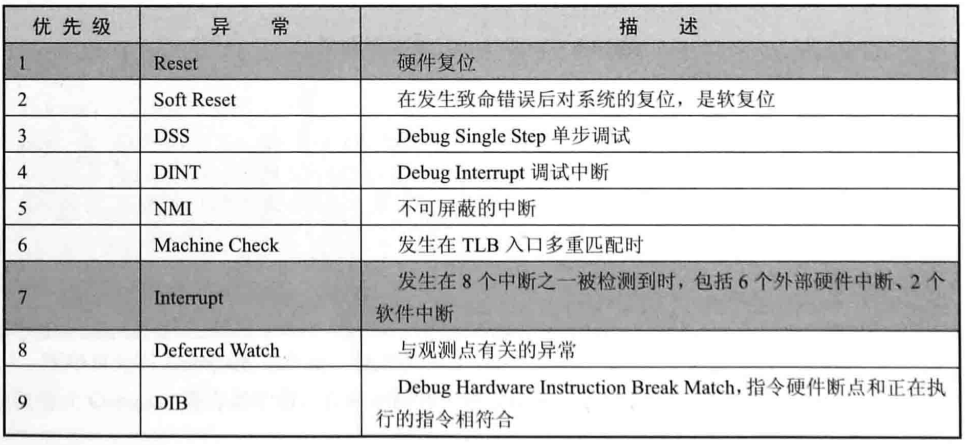
**异常相关指令的实现**

我们离OpenMIPS教学版的实现就差这一步啦~

# MIPS32架构中定义的异常类型

**异常**包括了中断（Interrupt）、陷阱（Trap）、系统调用（System Call）等一系列可以打断程序正常运行流程的情况。

异常类型及优先级表如下表所示，本节仅实现灰色的部分：





本OpenMIPS处理器只实现其中6中异常情况：

* 硬件复位
* 中断（包含软中断、硬中断）
* System系统调用
* 无效指令
* 溢出
* 自陷指令引发的异常

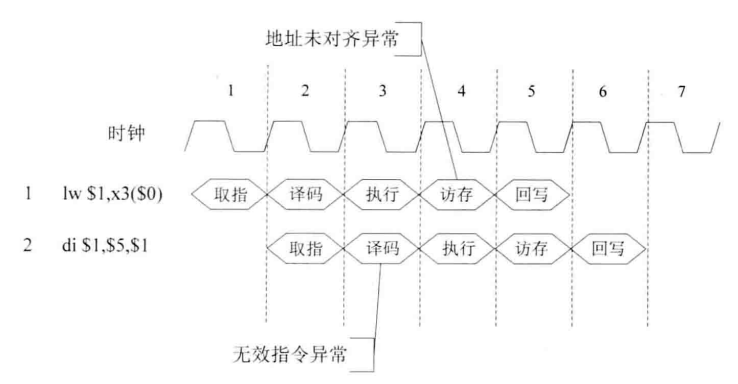
其中，硬件复位不需要考虑保护现场，只要将寄存器全部清零，从地址0x0处取指执行即可。

# 精确异常

异常发生时，要记住流水线上没有执行完的指令处于哪个阶段，以便异常处理结束后恢复执行，这就是精确异常。

在异常发生时，有一个被异常打断的指令，我们叫异常受害者（Exception Victim），有也就是异常发生的指令，该指令之前的所有指令都要被执行完毕，该指令后的指令都要被取消。

我们要保证发生异常的顺序和指令执行顺序一致，例如下图：



先进来的lw指令在访存发生异常，但是后进来的di（无效指令，未定义）却在译码阶段先发生异常。

为避免上述情况，先发生的异常并不立即处理，异常事件只是被标记，并继续运行流水线。一般会设计一个特殊的流水线阶段，只有当到达流水线的某个阶段（如访存阶段），才会进行异常的处理，也就是说上述的例子当di指令先在第三个时钟周期发生异常，但是做上标记，等他到访存阶段再执行，然而到第四个时钟周期lw发生了异常，也是在访存阶段，因此处理lw的异常。

# 异常处理过程

（1）检测CP0中Status寄存器的EXL字段，分两种情况

* 如果EXL为1，表示已经在异常处理过程中了，忽略异常，因为在异常处理过程中会禁止中断，若当前异常类型不是中断，将异常原因保存到CP0的Cause寄存器的ExcCode中，转步骤（4）。
* 如果EXL为0，将异常原因保存到Cause的ExcCode，进入步骤（2）

（2）检查异常发生的指令是否在延迟槽，如果在延迟槽，设置EPC寄存器的值为该指令地址减去4，同时设置Cause寄存器的BD字段为1，反之EPC寄存器的值就是该指令的地址，同时Cause的BD为0.

（3）设置Status寄存器EXL字段为1，进入异常处理过程，禁止中断。

（4）处理器转移到实现定义好的地址（有异常处理例程），这个地址叫异常处理例程入口地址。OpenMIPS定义的异常处理例程入口地址如下表所示。**地址可以自己设置。**



异常返回指令eret，要清除Status寄存器的EXL字段，还要将EPC寄存器保存的地址恢复到PC中。

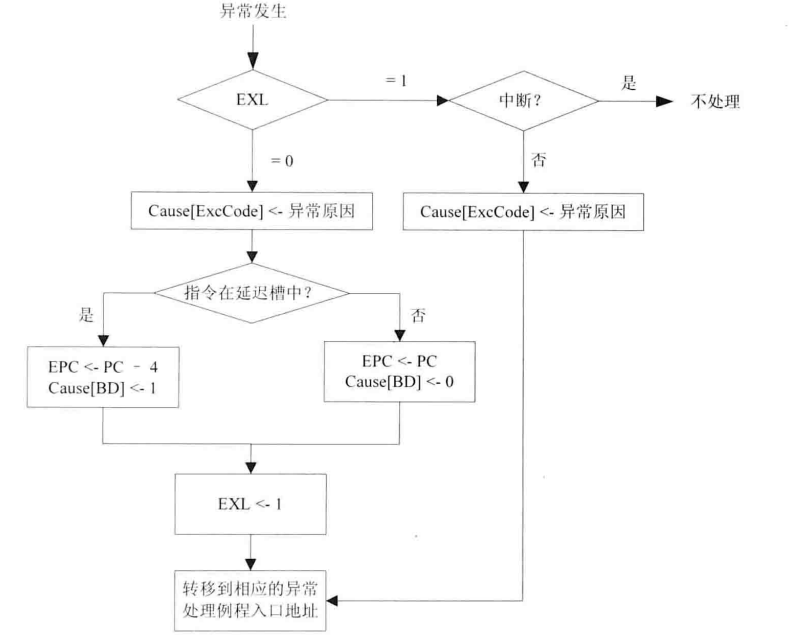
引入延迟槽之前，处理器执行转移指令的顺序：

**转移指令->转移目标地址的指令**

引入延迟槽后，处理器执行转移指令的顺序：

**转移指令->延迟槽指令->转移目标地址的指令**

下图表示了异常处理过程，很清晰。



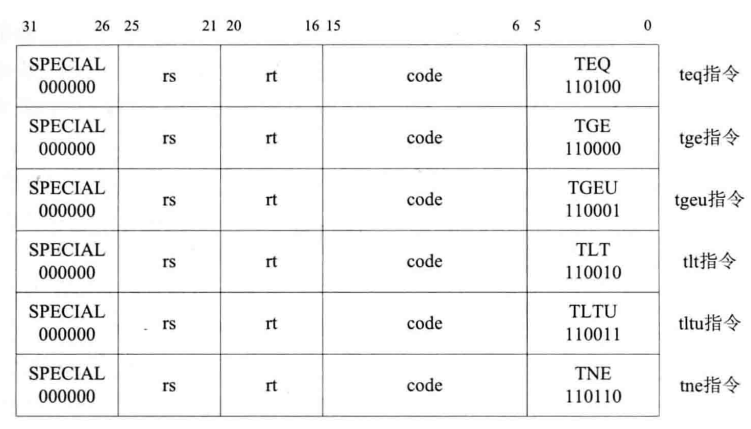
因此若延迟槽指令发生了异常，若保存该条指令，那么返回来执行延迟槽指令的下一条指令，跳转指令并没有执行，因此要保存延迟槽指令的前面一条指令的地址。

# 异常相关指令

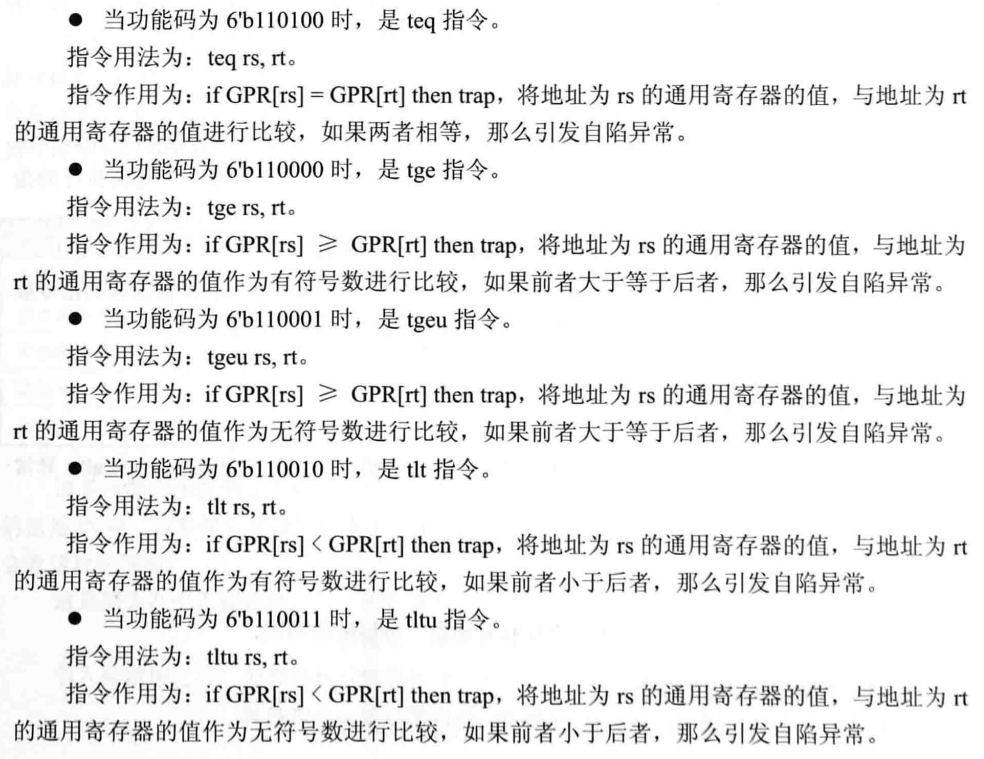
## 4.1 自陷指令

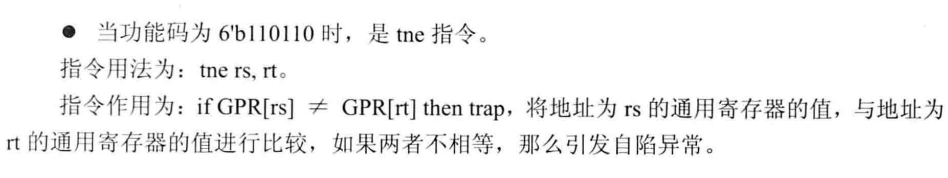
### 4.1.1 不包含立即数的自陷指令

指令格式如下表：



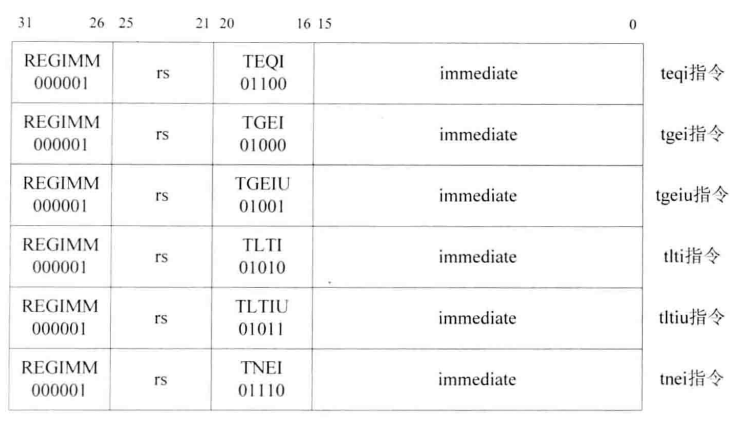
均是R类型指令，指令码都是SPECIAL，直接根据0~5bit功能码区分。另外其中code字段没有作用，忽略。

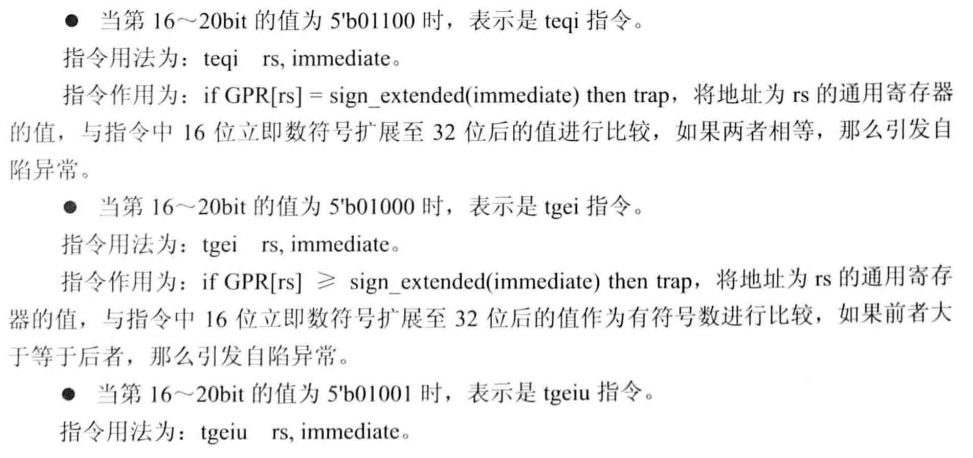


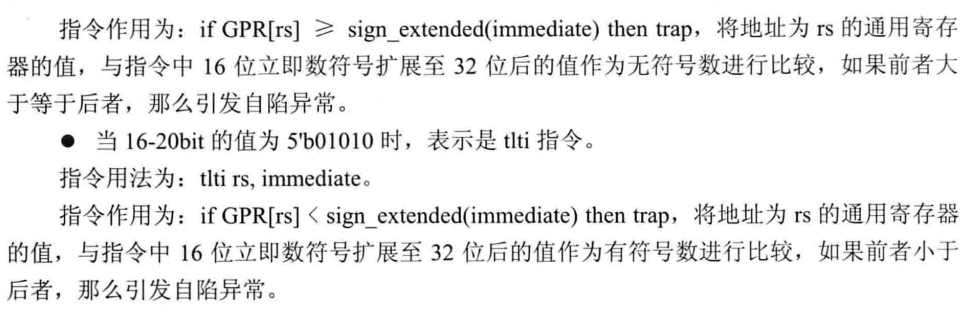


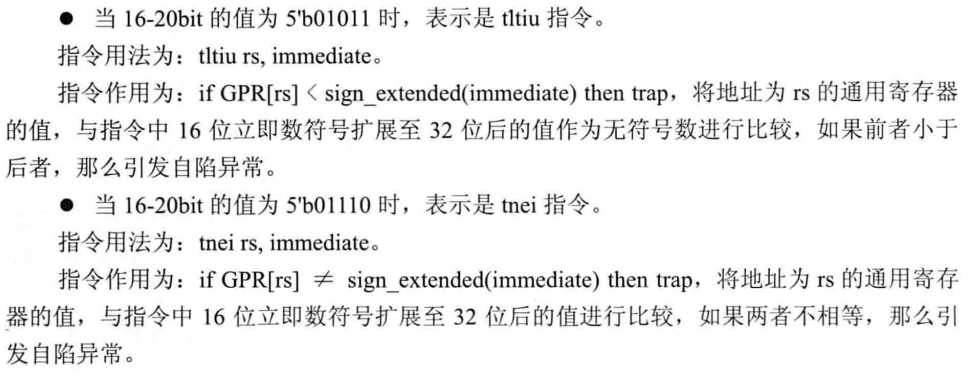
### 4.1.2 包含立即数的自陷指令

6条如下：



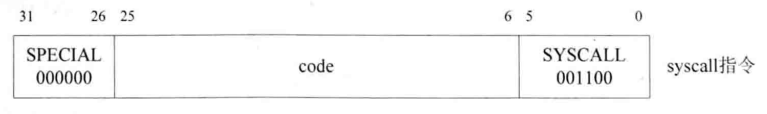
 均属于I类型指令，指令码都是REGIMM，是寄存器与立即数比较的结果。





## 4.2 系统调用指令syscall

格式如下图：



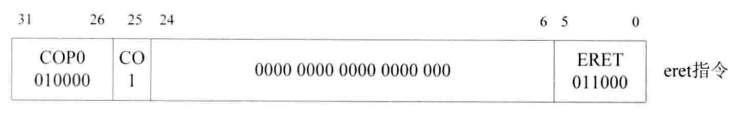
指令用法：syscall（无操作数）

指令作用：引发系统调用异常。MIPS架构定义了处理器两种工作模式：用户模式和内核模式，前一种操作受限，后一种主要用于处理异常和具有优先权的操作系统函数，包括管理协处理器CP0和I/O等。用户模式下的程序为了执行一些在内核模式下才能进行的操作，可以用syscall指令，引发系统调用异常，进入异常处理例程，从而进入内行人模式。用户模式和内核模式的状态标记为CP0种Status寄存器的UM字段。

但是OpenMIPS没有区分用户模式和内核模式。

## 4.3 异常返回指令eret

格式如图：



指令码为COP0，与mtc0和mfc0是一样的。

指令用法：从异常处理例程返回，执行该指令，进行如下操作：

1. 使EPC寄存器的值称为新的取指地址
2. 设置Status寄存器的EXL字段为0，表示不再处于异常级。

# 异常处理实现思路

## 5.1 实现思路

在流水线的各个阶段手机异常信息，传递到流水线访存阶段，在访存阶段统一处理异常信息。需要手机一下异常信息：

* 在流水线译码阶段判断是否是系统调用异常、是否是返回指令、返回指令。
* 在流水线执行阶段判断是否有自陷异常、溢出异常。
* 在流水线访存阶段检查是否有中断发生。  
  下面详细介绍一下该过程：

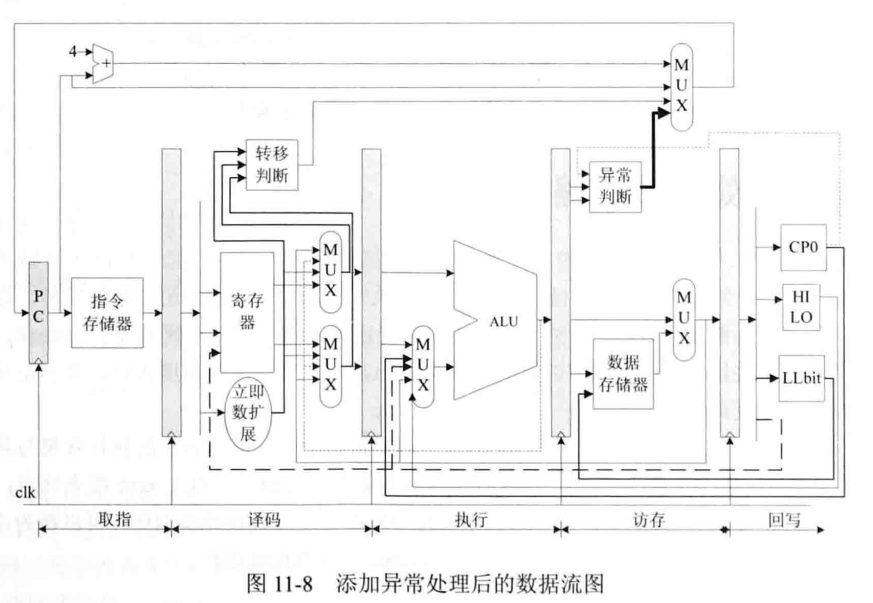
在流水线访存阶段，处理器根据CP0相关寄存器的值判断异常是否需要处理，如果需要处理，则转移到异常对应的处理例程入口地址，清除（reset即可）流水线除了**回写**阶段外的全部信息，同时修改CP0中相关寄存器的值。

如果是eret指令，转移到EPC寄存器保存的地址处，同时也要清除流水线上除了回写阶段外的全部信息，修改协处理器CP0中相关寄存器的值。

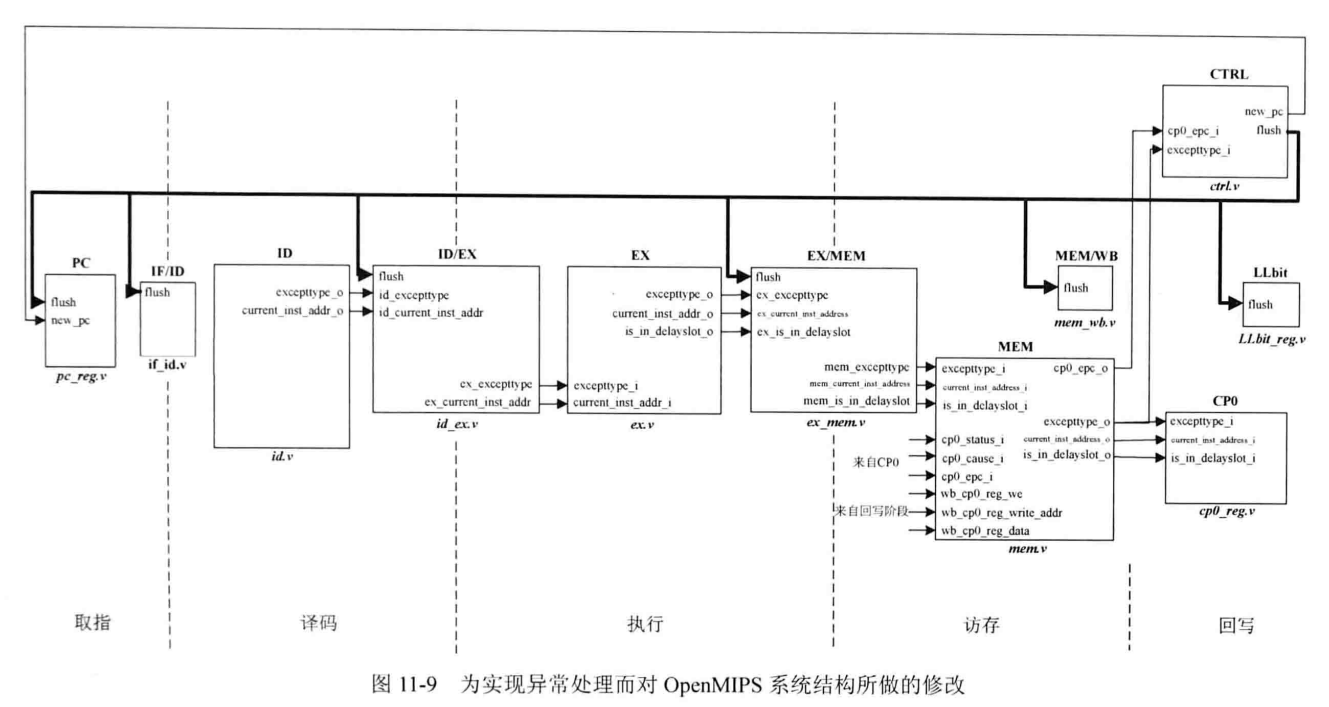
对于为什么要清除除了回写阶段外的所有信息，是因为“精确异常”，回写阶段可能是上一条指令正在正确执行，所以不能清除掉。

## 5.2 修改数据流图

要增加异常判断模块，主要作用是根据从译码、执行阶段传递过来以及CP0中的值判断是否要处理异常，如果要处理异常，那么按照异常类型给出新的指令地址送入PC。



## 5.3 修改系统结构



具体过程说明如下：

（1） 流水线译码阶段ID模块会判断是否是系统调用指令syscall、异常返回指令eret、无效指令，这些信息通过excepttype\_o接口传递到执行阶段，同时将指令地址通过current\_inst\_addr\_o接口传递到执行阶段。

（2）流水线执行阶段EX模块会进一步判断是否有自陷异常，或者溢出异常。这些信息会融合到译码阶段给出的异常信息（通过EX模块的excepttype\_i接口传入），然后通过excepttype\_o接口传递到访存阶段。同时，通过current\_inst\_addr\_o接口将指令地址传递到访存阶段，通过is\_in\_delayslot\_o接口指出指令是否位于延迟槽中，该信息也被传递到访存阶段。

（3）流水线访存阶段MEM模块会依据传递过来的异常类型excepttype\_i、Cause寄存器的值（通过cp0\_cause\_i接口输入）、Status寄存器的值（通过cp0\_status\_i接口输入），综合判断是否需要处理异常，如果需要处理，那么最终的异常类型会通过excepttype\_o接口送入CTRL模块，CTRL模块据此给出异常处理入口的入口地址（通过new\_pc接口送至PC）。

（4）如果要处理异常，那么还需要修改协寄存器中的EPC、Status、Cause等寄存器的值，因此访存阶段给出的最终的异常还要通过excepttype\_o接口送入CP0模块，同时送入的还有发生异常的指令是否在延迟槽中（通过is\_in\_delayslot\_i接口送入）、发生异常的指令的地址（通过current\_inst\_address\_o接口送入），修改CP0寄存器的值。

（5）如果要处理异常，那么还需要清楚流水线上除了回写阶段外所有寄存器的值，CTRL模块通过送出flush信号实现此目的。

（6）之前第九章实现ll、sc指令的时候引入了LLbit寄存器，当ll指令执行的时候会设置LLbit为1，当sc执行的时候会检查该寄存器的值是否为1，如果为1就正常执行，如果为0则认为出现了干扰，不进行存储操作。出现干扰的原因之一就是在ll、sc指令之间产生了异常，所以在异常处理过程中会多进行一步操作，就是将LLbit寄存器置为1，因此LLbit模块也有flush信号输入。

# 六、修改代码以实现异常处理

## 6.1 修改取指阶段

### 6.1.1 修改PC模块

增加的接口如下表：



### 6.1.2 修改IF/ID模块

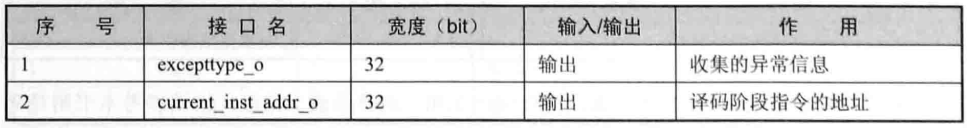
增加的接口如下表：

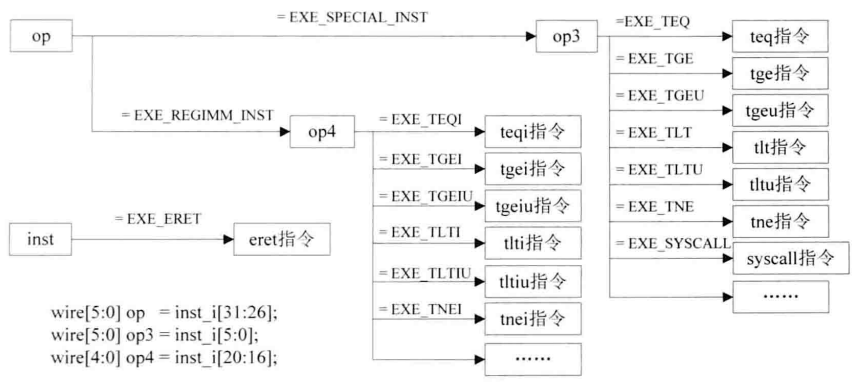


## 6.2 修改译码阶段

### 6.2.1 修改ID模块

新增加的接口如下表：





这段代码比较杂乱，其实跟之前的差不多，指令较多。

**变量excepttype\_o收集译码阶段得到的异常信息，其第8bit表示是都是syscall指令引起的系统调用异常，第9bit表示是否是无效指令引起的异常，第12bit表示是否是返回指令eret。**

1. teq指令

* 不需要写通用寄存器， 因此wreg\_o为WriteDisable
* 要读取两个寄存器rs、rt，因此reg1\_read\_o和reg2\_read\_o、为1.默认通过Regfile模块读端口1读取的寄存器地址reg1\_addr\_o是指令的21-25bit，就是teq指令中的rs，默认通过Regfile模块读端口2读取的寄存器地址reg2\_addr\_o是指令的16-20bit，正是rt。
* teq指令不需要写通用寄存器，设置alusel\_o为EXE\_RES\_NOP，另外设置aluop\_o为EXE\_TEQ\_OP

1. 其余类推，syscall不需要写目的寄存器，不需要读取通用寄存器，不执行运算，设置excepttype\_is\_syscall为True。

### 6.2.2 修改ID/EX模块

增加接口如下表：



代码主要功能：

1. 在flush为高电平时，清除ID/EX模块中所有的值（设置为初始值）
2. 在没有流水线清除事件（flush为0）时，并且译码阶段没有暂停的情况下，将译码阶段得到的异常信息id\_excepttype、指令地址id\_current\_inst\_address传递到执行阶段。

## 6.3 修改执行阶段

### 6.3.1 修改EX模块

接收译码阶段传递过来的信息，进一步判断是否有自陷异常或者溢出异常。

新增接口如下表：



对于大于或者小于等于之类指令的跳转或者陷阱指令，其比较方法还是做减法。尽管这里在算数操作指令那儿已经介绍过，这里还是再复习一下。

如果是减法运算、有符号比较运算、有符号自陷指令，我们将reg2\_i\_mux设为第二个操作数reg2\_i的补码，意指将减法变成加法运算。其方法就是逐位取反最后加一。这时result\_sum = reg1\_i + reg2\_i\_mux.这应该很容易理解了。

计算操作数1是否小于操作数2，分两种情况：

首先就是当前指令为有符号比较指令或者有符号自陷异常指令的时候，此时又分三种情况：

1. reg1\_i为负数、reg2\_i为正数，显然前者小
2. reg1\_i为正数、reg2\_i也是正数，并且前者减去后者的值小于0，就是result\_sum为负数，此时也是前者小
3. reg1\_i为负数、reg2\_i为负数，并且前者减后者小于0，此时也是前者小

然后就是当前指令为无符号比较指令或者无符号自陷异常指令的时候，直接使用比较运算符比较两者大小。Reg1\_lt\_reg2及表示操作数1是否小于操作数2.

下面就是判断是否满足自陷异常和溢出异常。

**在执行阶段收集的异常信息与译码阶段收集的异常信息一起通过接口excepttype\_o传递到访存阶段，其中第10bit表示是否有自陷异常，第11bit表示是否有溢出异常。**

同时传递到访存阶段的还有指令地址、是否是延迟槽指令等信息，当异常发生时，这两个信息用来确认保存到EPC寄存器的值。

### 6.3.2 修改EX/MEM模块

新增接口如下图所示：

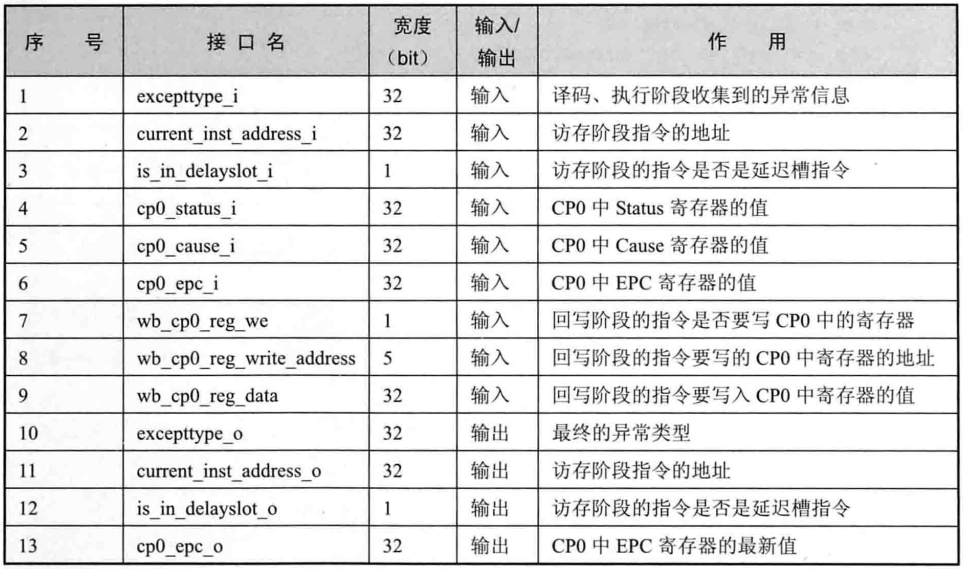


主要目的是清除流水线。

## 6.4 修改访存阶段

### 6.4.1 修改MEM模块

OpenMIPS处理器会在访存阶段的MEM模块综合所有的异常信息、CP0寄存器的值，最终判断是否有要处理的异常。新增接口如下表：

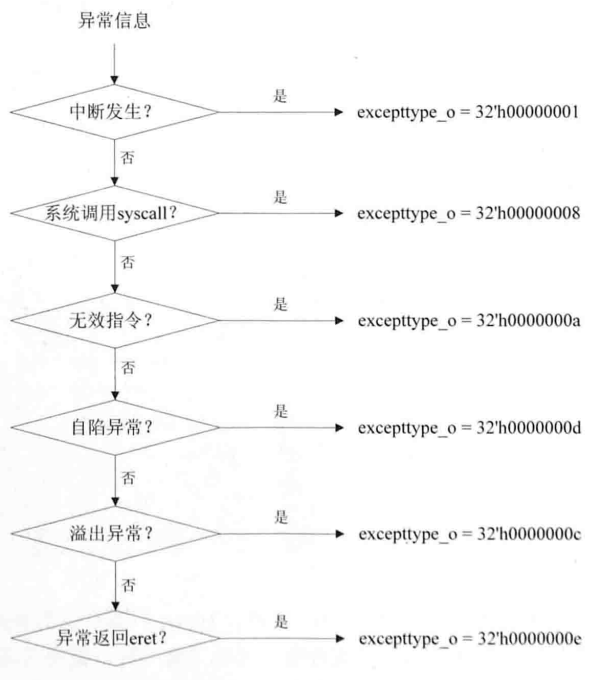


上述代码分为三段理解：

（1）第一段：从CP0模块传入的Status、EPC、Cause等寄存器的值不一定是最新值，因为处于回写阶段的指令可能要写这些值，这也就是解决**数据相关**的问题。wb\_cp0\_reg\_data就是回写阶段要写入的寄存器的新值，如果没有数据相关，那么CP0模块传入的cp0\_status\_i（举例）就是Status寄存器的值。

（2）根据CP0中寄存器的值，以及译码、执行阶段收集到的异常类型，得到最终的异常类型。首先判断当前处于访存指令的地址是否为0，如果为0表示处于复位状态，或者处于flush状态，亦或处于暂停状态，这三种状态都不处理异常。

如果当前处于访存阶段指令的地址不为0，那么可以进一步判断有没有异常，是何种异常，从而给excepttype\_o赋值。举例如下：

发生中断：Cause寄存器IP不为0，且Status寄存器中相应的的中断掩码字段也不为0，另外Status寄存器的EXL字段为0，表示不处于异常处理过程中，Status寄存器的IE字段为1，表示中断使能。

系统调用异常：excepttype\_i第8bit为1.

无效指令异常：excepttype\_i第9bit为1.

自陷异常：excepttype\_i第10bit为1.

溢出异常：excepttype\_i第11bit为1.

异常返回eret：excepttype\_i第12bit为1.

（3）OpenMIPS处理器要实现精确异常，也就是发生异常时，引起异常的指令及其后面已经进入流水线的指令都会失效。如果引起异常的是存储指令，那么就要使其失效，就要停止修改数据存储器，因此修改mem\_we\_o的值，如果发生异常，将其值修改为0，就不会修改数据存储器了。

### 6.4.2 修改MEM/WB模块

增加接口如图：



就是清除流水线。

## 6.5 修改协处理器CP0

新增接口如下表：



这里主要讲一下中断（Interrupt）和系统调用异常（Syscall）

* 中断

依据发生异常的指令是否位于延迟槽中，设置EPC的值，以及Cause寄存器的BD字段，如果位于延迟槽中，那么设置EPC寄存器为上一条指令的地址，Cause寄存器的BD字段为1.反之设置EPC寄存器为发生异常指令的地址，BD为0.另外，设置Status寄存器的EXL字段为1，表示处于异常级，中断禁止。最后设置Cause寄存器的ExcCode字段为5’b00000，表示异常原因是中断。

* 系统调用异常

分两种情况：

1. 如果Status的EXL字段为0，按照中断的操作进行；
2. 如果Status的EXL字段为1，表明处于异常级，又发生了新的异常，只需要将异常原因保存到Cause的ExcCode字段。

其他异常类似系统调用异常。

## 6.6 修改控制模块CTRL

CTRL模块根据异常类型，给出新的取指地址（就是异常处理例程入口地址），同时决定是否要清除流水线。新增接口如下表：



其中excepttype\_i、cp0\_epc\_i都来自MEM模块。

当发生异常（excepttype\_i不为0），根据异常类型设置new\_pc为异常处理例程入口地址，同时清除流水线（flush=1）.

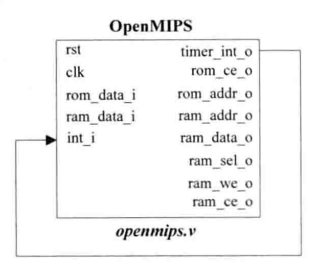
对于eret指令，要返回异常发生前的状态继续执行，因此赋值为EPC寄存器中的值。

## 6.7 修改OpenMIPS

略。

# 七、修改SOPC

在SOPC模块，将时钟中断输出作为一个中断信号输出，就可以处理时钟中断了，如图：

 这里，int\_i的宽度为6，时钟中断输出接口timer\_int\_o连接到int\_i的最低位。

代码修改略。

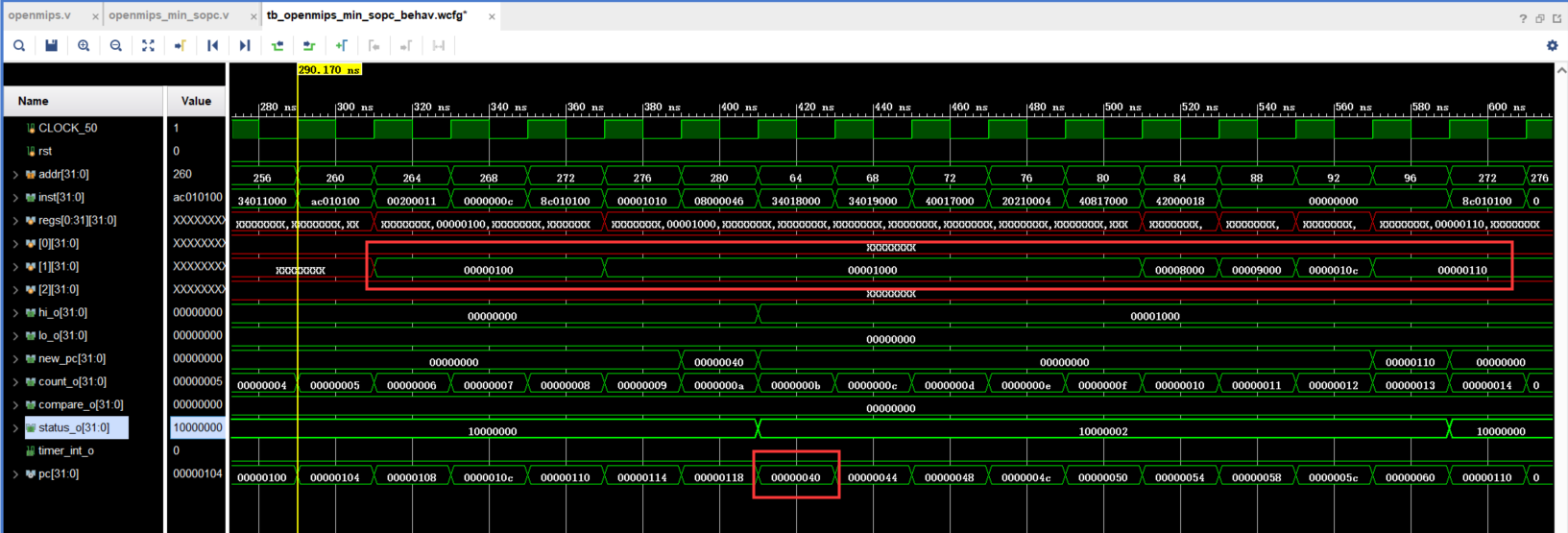
# 八、测试及分析

Nice！终于到测试了，这一章真的又臭又长~

## 8.1 测试系统调用异常

使用下面的汇编测试程序：



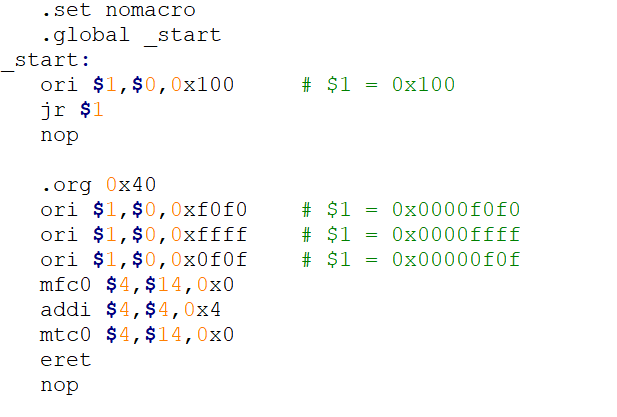


因为低地址有异常处理例程，所以启动时地址转移到0x100。

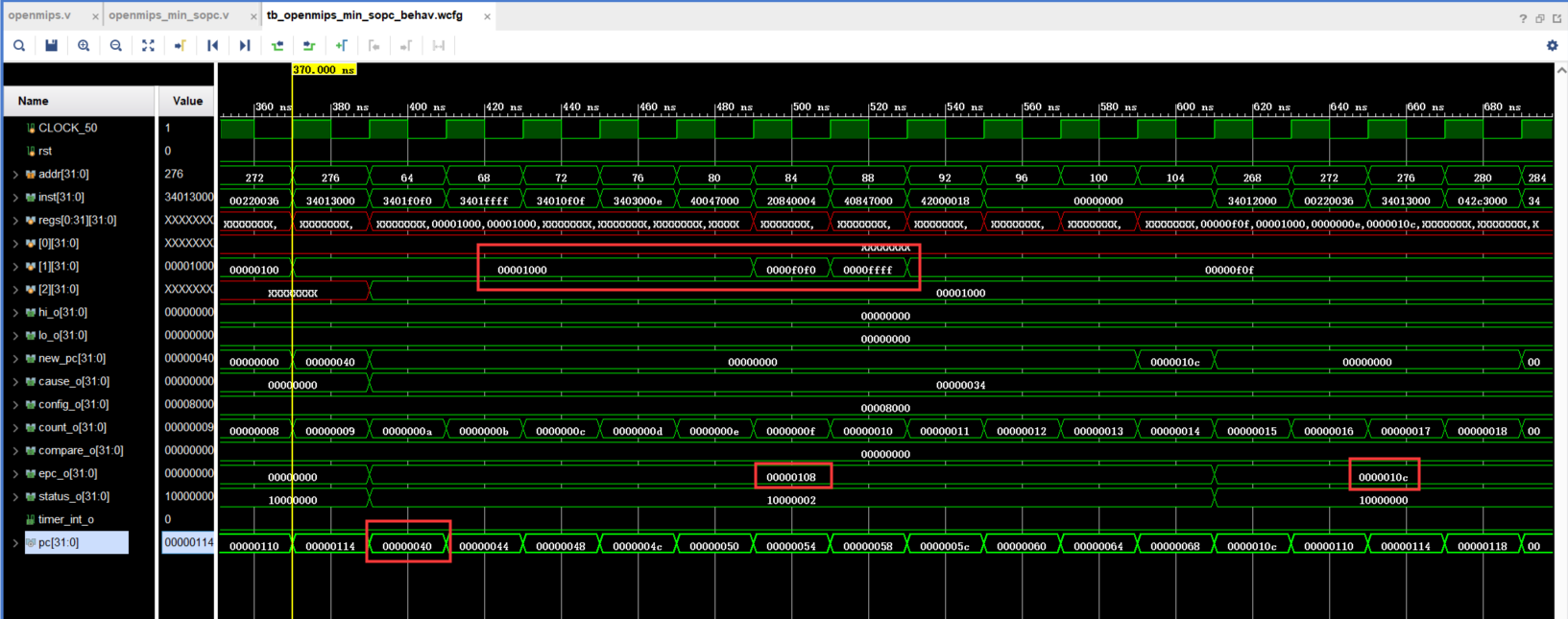
系统调用异常，取指为0x40，会修改$1的值。

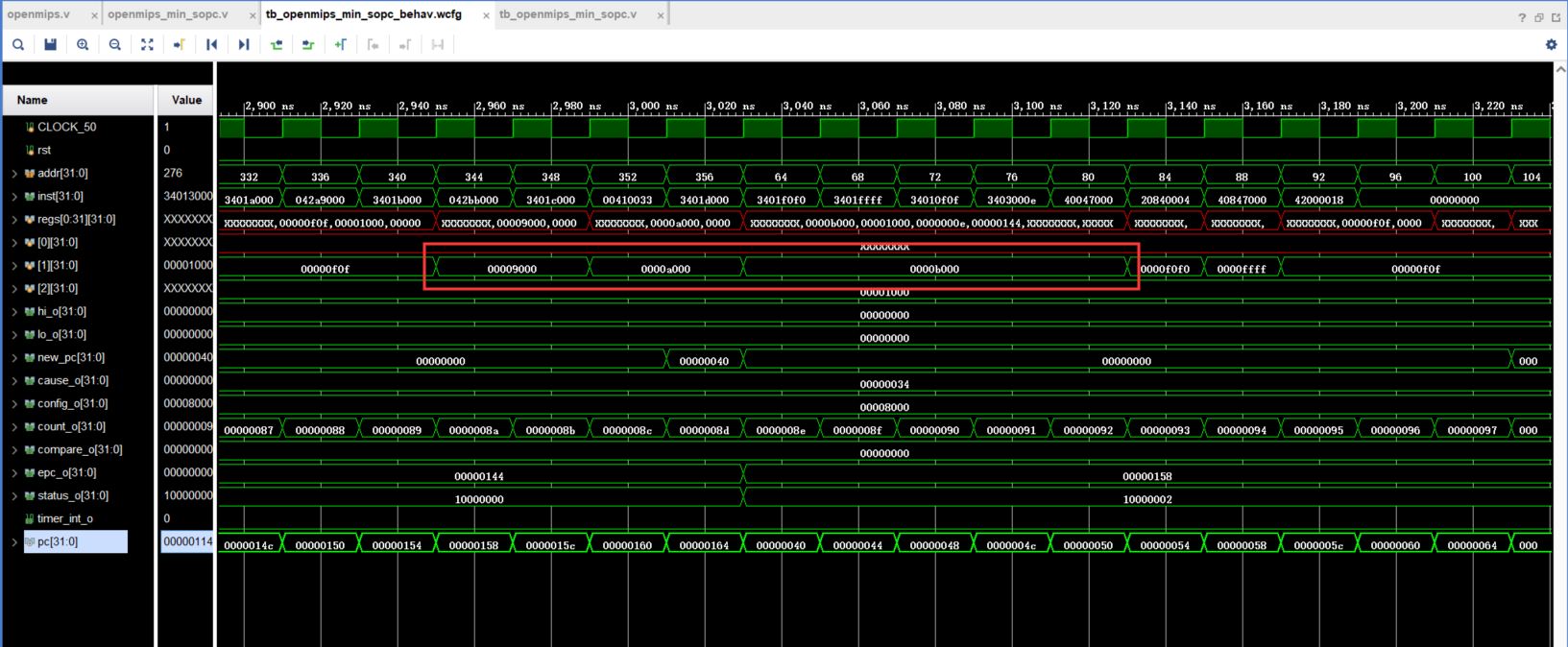
## 8.2 测试自陷异常

使用下面的汇编测试程序：



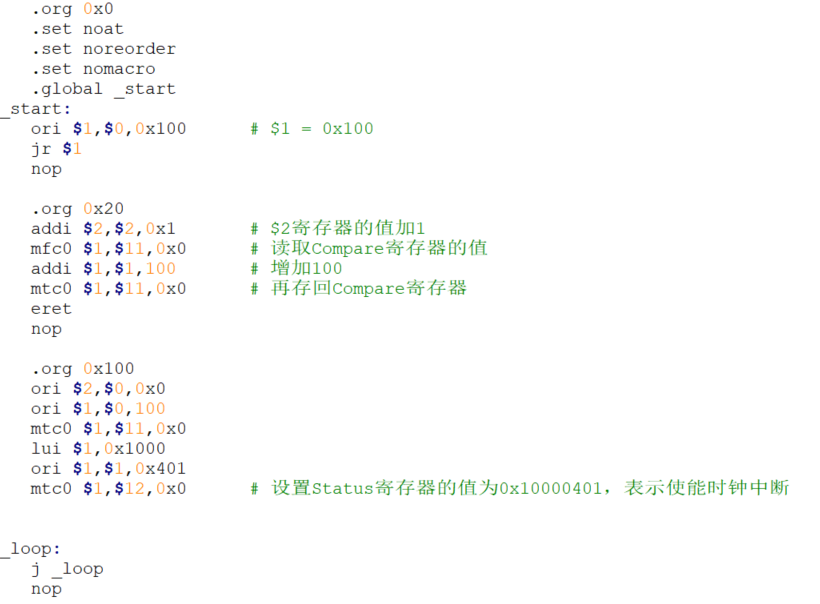






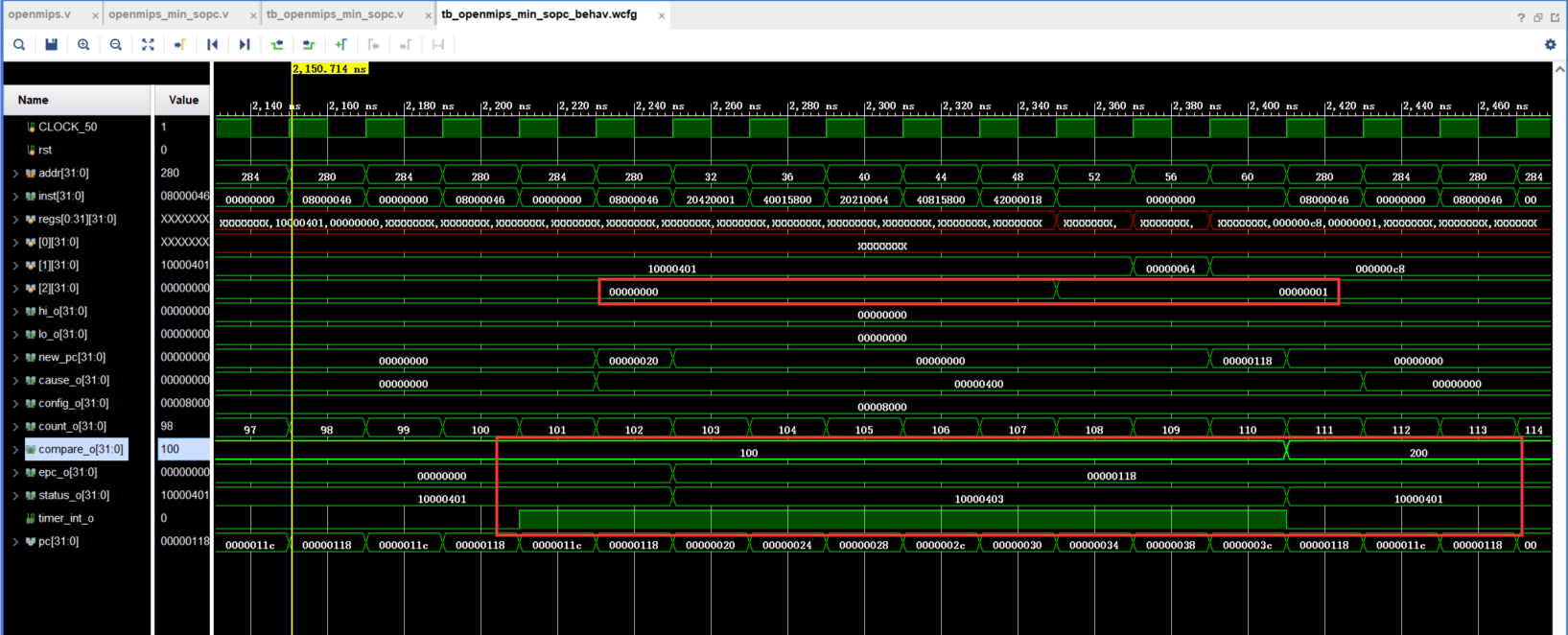
## 8.3 测试时钟中断

使用下面的汇编测试程序：





哦吼，发生了问题。在Compare和Status寄存器值相等的时候，Compare寄存器没有按照汇编程序中那样加100，$2寄存器也没有加1，但是CP0寄存器中的值全部写进去了呀，而写time\_int\_o使能信号也打开了。仔细一看，发现pc根本就没有到中断例程处去，根本没有发生中断。结果发现是我模块例化的时候，模块接口值没有传…太粗心了，改过之后就正确了，正确的如下图，寄存器$2会不断加1，目的就是为了便于观察时钟中断的次数。



# 九、总结

至此，教学版的OpenMIPS就全部完成了，接下来的工作就是将该处理器与SDRAM控制器、GPIO模块，Flash控制器、UART控制器、Wishbone总线互联矩阵等模块组成一个小型SOPC，然后下载到FPGA芯片进行上板验证，最后还会移植一个嵌入式实时操作系统μC/OS-II.

过程很辛苦，一切都是值得了，牺牲了复习计组的时间来完成这个，得抓紧接着复习计组了。