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~

一、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：

优先级	异常	描述
1	Reset	硬件复位
2	Soft Reset	在发生致命错误后对系统的复位，是软复位
3	DSS	Debug Single Step 单步调试
4	DINT	Debug Interrupt 调试中断
5	NMI	不可屏蔽的中断
6	Machine Check	发生在 TLB 入口多重匹配时
7	Interrupt	发生在 8 个中断之一被检测到时，包括 6 个外部硬件中断、2 个软件中断
8	Deferred Watch	与观测点有关的异常
9	DIB	Debug Hardware Instruction Break Match，指令硬件断点和正在执行的指令相符合
10	WATCH	取指地址与观测寄存器中的地址相同时发生
11	AdEL	Fetch Address Align Error 取指地址对齐异常
12	TLB Refill（TLBL）	指令 TLB 失靶
13	TLB Invalid（TLBL）	指令 TLB 无效
14	IBE	Instruction Fetch Bus Error 取指令总线错误
15	DBp	断点，执行了 SDBBP 指令
16	Sys	执行了系统调用指令 syscall
	Bp	执行了 break 指令
	CpU	在协处理器不存在或不可用的情况下，执行了协处理器指令
	RI	无效指令
	Ov	算术操作指令 add、addi、sub 运算溢出
	Tr	执行了自陷指令
17	DDBL/DDBS	Data Address Break or Data Value Break on Store 存储过程中，数据地址断点或数据值断点
18	WATCH	数据地址与观测寄存器中的地址相同时发生
19	AdEL	加载数据的地址未对齐
	AdES	存储数据的地址未对齐

20	TLB Refill	TLBL TLBS	数据 TLB 失靶
21	TLB Invalid		数据 TLB 无效
22	TLB Mod		对不可写的 TLB 进行了写操作
23	DBE		加载存储总线错误
24	DDBL		Data Hardware Breakpoint matched in load data compare 加载的数据与硬件断点设置的数据相等

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

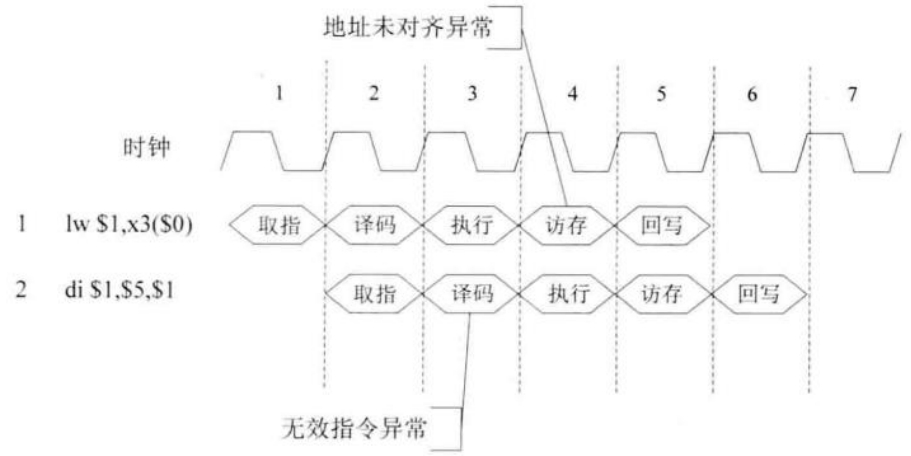
其中，硬件复位不需要考虑保护现场，只要将寄存器全部清零，从地址 0x0 处取指执行即可。

二、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。

三、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。**

异常类型	处理例程地址	引起异常的条件
中断（Interrupt）	0x20	硬件或软件中断
系统调用 Sys	0x40	执行了系统调用指令 syscall
无效指令 RI	0x40	当前指令是 OpenMIPS 不支持的指令
溢出 Ov	0x40	算术操作指令 add、addi、sub 运算溢出
自陷 Tr	0x40	执行了自陷指令

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。

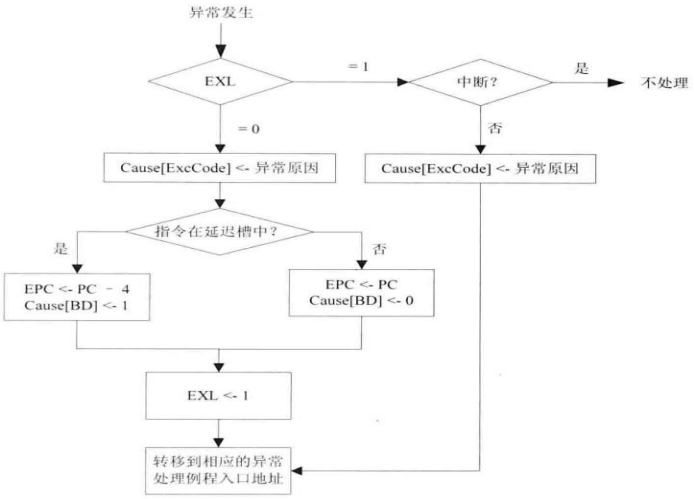
引入延迟槽之前，处理器执行转移指令的顺序：

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。

四、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：

31 26	25 21	20 16	15 6	5 0	
SPECIAL 000000	rs	rt	code	TEQ 110100	teq指令
SPECIAL 000000	rs	rt	code	TGE 110000	tge指令
SPECIAL 000000	rs	rt	code	TGEU 110001	tgeu指令
SPECIAL 000000	rs	rt	code	TLT 110010	tlt指令
SPECIAL 000000	rs	rt	code	TLTU 110011	tltu指令
SPECIAL 000000	rs	rt	code	TNE 110110	tne指令

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。

- 当功能码为 6'b110100 时，是 teq 指令。

指令用法为：teq rs, rt。

指令作用为：if GPR[rs] = GPR[rt] then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值进行比较，如果两者相等，那么引发自陷异常。

- 当功能码为 6'b110000 时，是 tge 指令。

指令用法为：tge rs, rt。

指令作用为：if GPR[rs] ≥ GPR[rt] then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值作为有符号数进行比较，如果前者大于等于后者，那么引发自陷异常。

- 当功能码为 6'b110001 时，是 tgeu 指令。

指令用法为：tgeu rs, rt。

指令作用为：if GPR[rs] ≥ GPR[rt] then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值作为无符号数进行比较，如果前者大于等于后者，那么引发自陷异常。

- 当功能码为 6'b110010 时，是 tlt 指令。

指令用法为：tlt rs, rt。

指令作用为：if GPR[rs] < GPR[rt] then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值作为有符号数进行比较，如果前者小于后者，那么引发自陷异常。

- 当功能码为 6'b110011 时，是 tltu 指令。

指令用法为：tltu rs, rt。

指令作用为：if GPR[rs] < GPR[rt] then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值作为无符号数进行比较，如果前者小于后者，那么引发自陷异常。

- 当功能码为 6'b110110 时，是 tne 指令。

指令用法为：tne rs, rt。

指令作用为：if GPR[rs] ≠ GPR[rt] then trap，将地址为 rs 的通用寄存器的值，与地址为 rt 的通用寄存器的值进行比较，如果两者不相等，那么引发自陷异常。

4.1.2 包含立即数的自陷指令

6 条如下：

31 26	25 21	20 16	15 0	
REGIMM 000001	rs	TEQI 01100	immediate	teqi指令
REGIMM 000001	rs	TGEI 01000	immediate	tgei指令
REGIMM 000001	rs	TGEIU 01001	immediate	tgeiu指令
REGIMM 000001	rs	TLTI 01010	immediate	tlti指令
REGIMM 000001	rs	TLTIU 01011	immediate	tltiu指令
REGIMM 000001	rs	TNEI 01110	immediate	tnei指令

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。

- 当第 16～20bit 的值为 5'b01100 时，表示是 teqi 指令。

指令用法为：teqi rs, immediate。

指令作用为：if GPR[rs] = sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值进行比较，如果两者相等，那么引发自陷异常。

- 当第 16～20bit 的值为 5'b01000 时，表示是 tgei 指令。

指令用法为：tgei rs, immediate。

指令作用为：if GPR[rs] ≥ sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值作为有符号数进行比较，如果前者大于等于后者，那么引发自陷异常。

- 当第 16～20bit 的值为 5'b01001 时，表示是 tgeiu 指令。

指令用法为：tgeiu rs, immediate。

指令作用为：if GPR[rs] ≥ sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值作为无符号数进行比较，如果前者大于等于后者，那么引发自陷异常。

- 当 16-20bit 的值为 5'b01010 时，表示是 tlti 指令。

指令用法为：tlti rs, immediate。

指令作用为：if GPR[rs] < sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值作为有符号数进行比较，如果前者小于后者，那么引发自陷异常。

- 当 16-20bit 的值为 5'b01011 时，表示是 tltiu 指令。

指令用法为：tltiu rs, immediate。

指令作用为：if GPR[rs] < sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值作为无符号数进行比较，如果前者小于后者，那么引发自陷异常。

- 当 16-20bit 的值为 5'b01110 时，表示是 tnei 指令。

指令用法为：tnei rs, immediate。

指令作用为：if GPR[rs] ≠ sign_extended(immediate) then trap，将地址为 rs 的通用寄存器的值，与指令中 16 位立即数符号扩展至 32 位后的值进行比较，如果两者不相等，那么引发自陷异常。

4.2 系统调用指令 syscall

格式如下图：

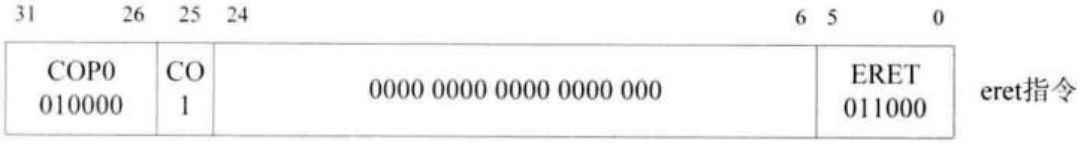
指令用法：syscall（无操作数）

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。

五、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。

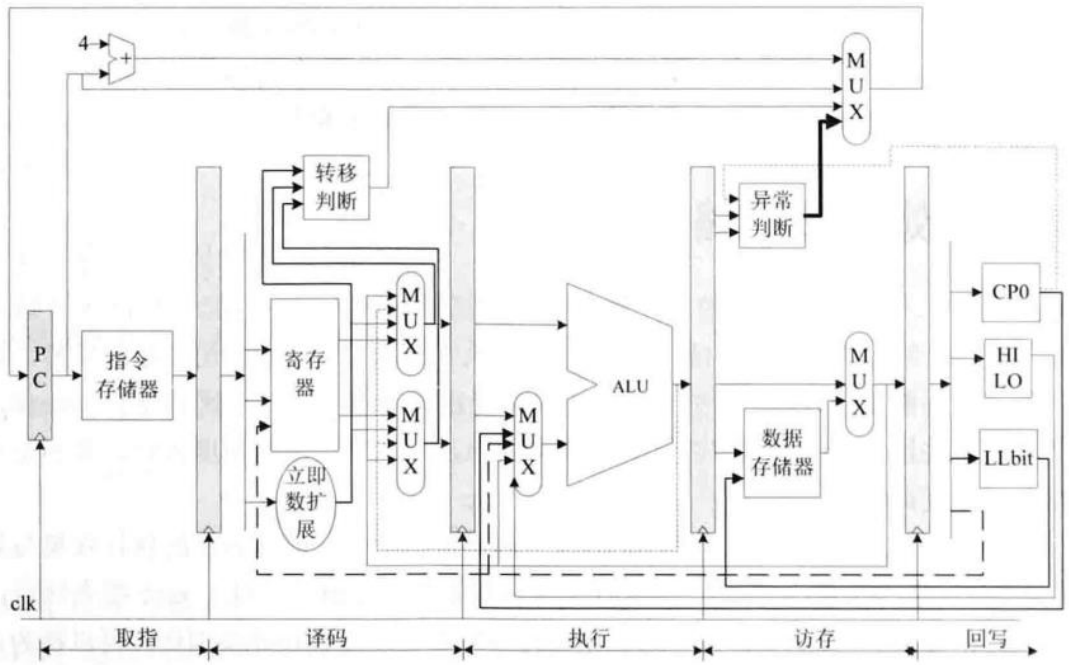


图 11-8 添加异常处理后的数据流图

5.3 修改系统结构

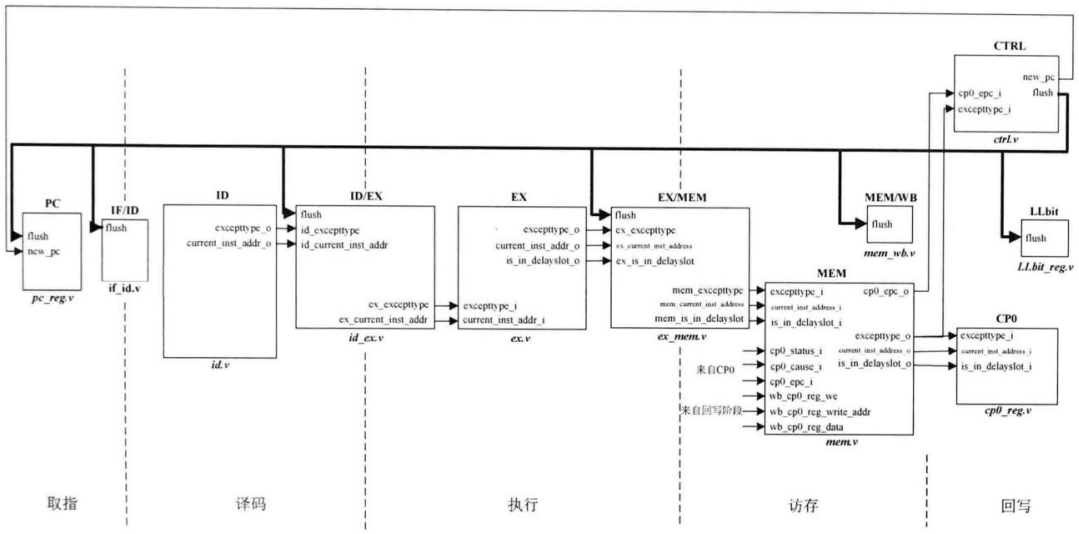


图 11-9 为实现异常处理而对 OpenMIPS 系统结构所做的修改

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：

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	flush	1	输入	流水线清除信号
2	new_pc	32	输入	异常处理例程入口地址

6.1.2 修改 IF/ID 模块

增加的接口如下表：

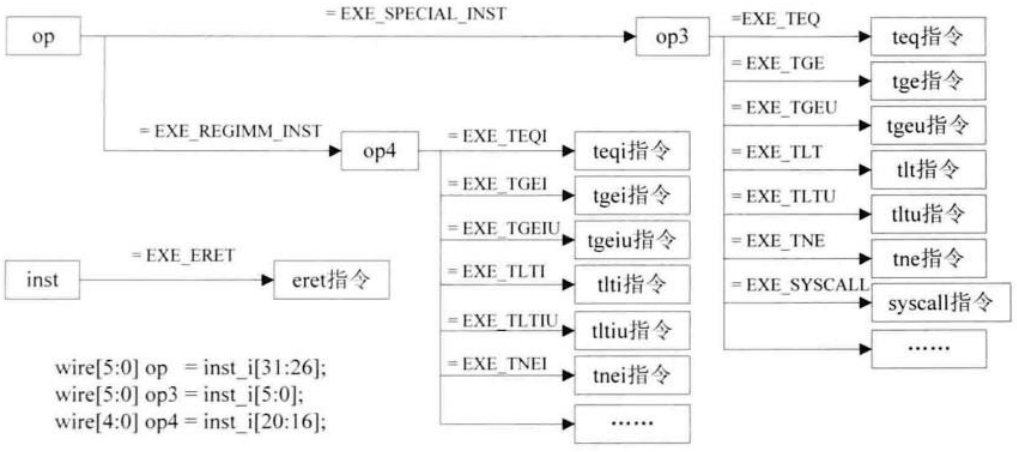
序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	flush	1	输入	流水线清除信号

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：

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	excepttype_o	32	输出	收集的异常信息
2	current_inst_addr_o	32	输出	译码阶段指令的地址



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

（2） 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：

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	flush	1	输入	流水线清除信号
2	id_excepttype	32	输入	译码阶段收集到的异常信息
3	id_current_inst_addr	32	输入	译码阶段指令的地址
4	ex_excepttype	32	输出	译码阶段收集到的异常信息
5	ex_current_inst_addr	32	输出	执行阶段指令的地址

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：

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	excepttype_i	32	输入	译码阶段收集到的异常信息
2	current_inst_addr_i	32	输入	执行阶段指令的地址
3	excepttype_o	32	输出	译码阶段、执行阶段收集到的异常信息
4	current_inst_addr_o	32	输出	执行阶段指令的地址
5	is_in_delayslot_o	1	输出	执行阶段的指令是否是延迟槽指令

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：

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	flush	1	输入	流水线清除信号
2	ex_excepttype	32	输入	译码、执行阶段收集到的异常信息
3	ex_current_inst_address	32	输入	执行阶段指令的地址
4	ex_is_in_delayslot	1	输入	执行阶段的指令是否是延迟槽指令
5	mem_excepttype	32	输出	译码、执行阶段收集到的异常信息
6	mem_current_inst_address	32	输出	访存阶段指令的地址
7	mem_is_in_delayslot	1	输出	访存阶段的指令是否是延迟槽指令

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：

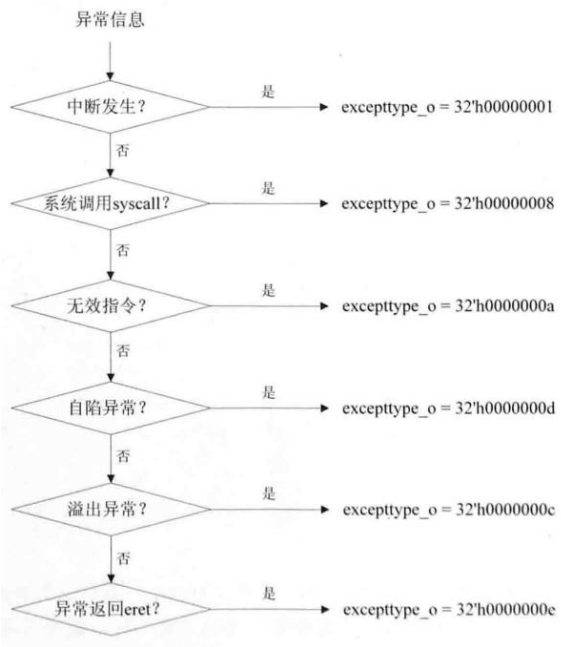
序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	excepttype_i	32	输入	译码、执行阶段收集到的异常信息
2	current_inst_address_i	32	输入	访存阶段指令的地址
3	is_in_delayslot_i	1	输入	访存阶段的指令是否是延迟槽指令
4	cp0_status_i	32	输入	CP0 中 Status 寄存器的值
5	cp0_cause_i	32	输入	CP0 中 Cause 寄存器的值
6	cp0_epc_i	32	输入	CP0 中 EPC 寄存器的值
7	wb_cp0_reg_we	1	输入	回写阶段的指令是否要写 CP0 中的寄存器
8	wb_cp0_reg_write_address	5	输入	回写阶段的指令要写的 CP0 中寄存器的地址
9	wb_cp0_reg_data	32	输入	回写阶段的指令要写入 CP0 中寄存器的值
10	excepttype_o	32	输出	最终的异常类型
11	current_inst_address_o	32	输出	访存阶段指令的地址
12	is_in_delayslot_o	1	输出	访存阶段的指令是否是延迟槽指令
13	cp0_epc_o	32	输出	CP0 中 EPC 寄存器的最新值

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：

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	flush	1	输入	流水线清除信号

就是清除流水线。

6.5 修改协处理器 CP0

新增接口如下表：

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	excepttype_o	32	输入	最终的异常类型
2	current_inst_address_o	32	输入	发生异常的指令地址
3	is_in_delayslot_o	1	输入	发生异常的指令是否是延迟槽指令

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：

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	cp0_epc_i	32	输入	EPC 寄存器的最新值
2	excepttype_i	32	输入	最终的异常类型
3	new_pc	32	输出	异常处理入口地址
4	flush	1	输出	是否清除流水线

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

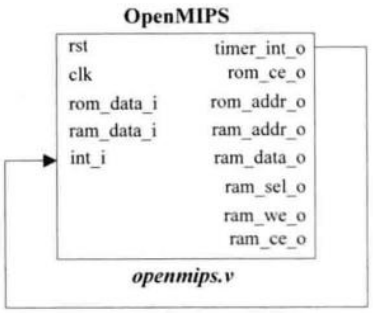
略。

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。



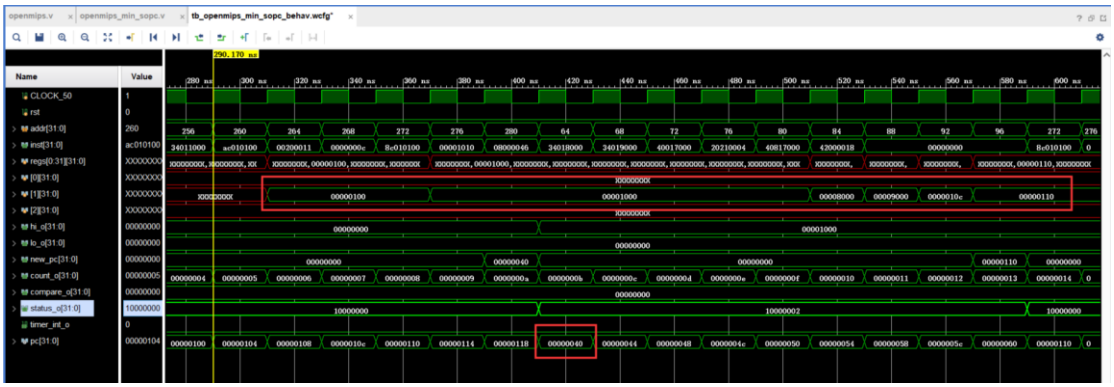
八、测试及分析

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:

```
   .org 0x0
   .set noat
   .set noreorder
   .set nomacro
   .global _start
_start:
   ori $1,$0,0x100      # $1 = 0x100
   jr $1
   nop

   .org 0x40
   ori $1,$0,0x8000     # $1 = 0x00008000
   ori $1,$0,0x9000     # $1 = 0x00009000
   mfc0 $1,$14,0x0      # $1 = 0x0000010c
   addi $1,$1,0x4       # $1 = 0x00000110
   mtc0 $1,$14,0x0
   eret
   nop

   .org 0x100
   ori $1,$0,0x1000     # $1 = 0x1000
   sw  $1, 0x0100($0)   # [0x100] = 0x00001000
   mthi $1              # HI = 0x00001000
   syscall
   lw  $1, 0x0100($0)   # $1 = 0x00001000
   mfhi $2              # $2 = 0x00001000
_loop:
   j _loop
   nop
```



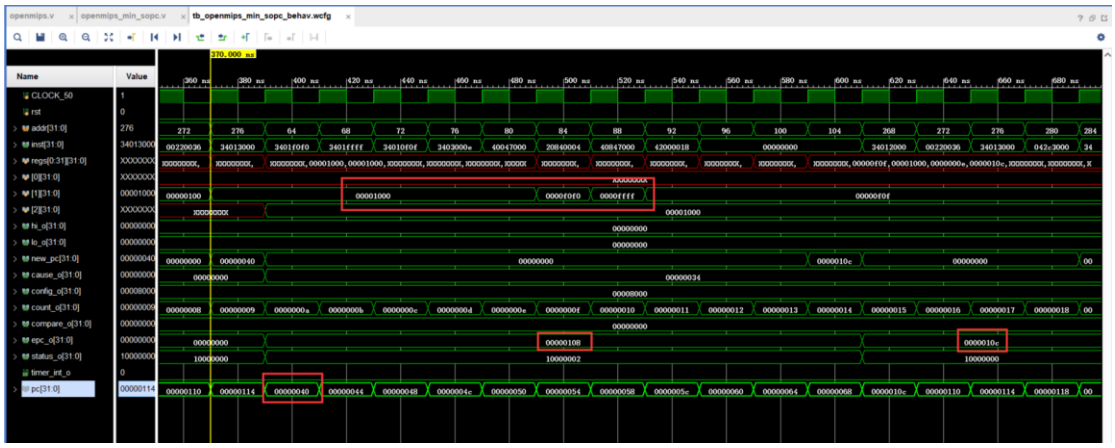
因为低地址有异常处理例程，所以启动时地址转移到 0x100。

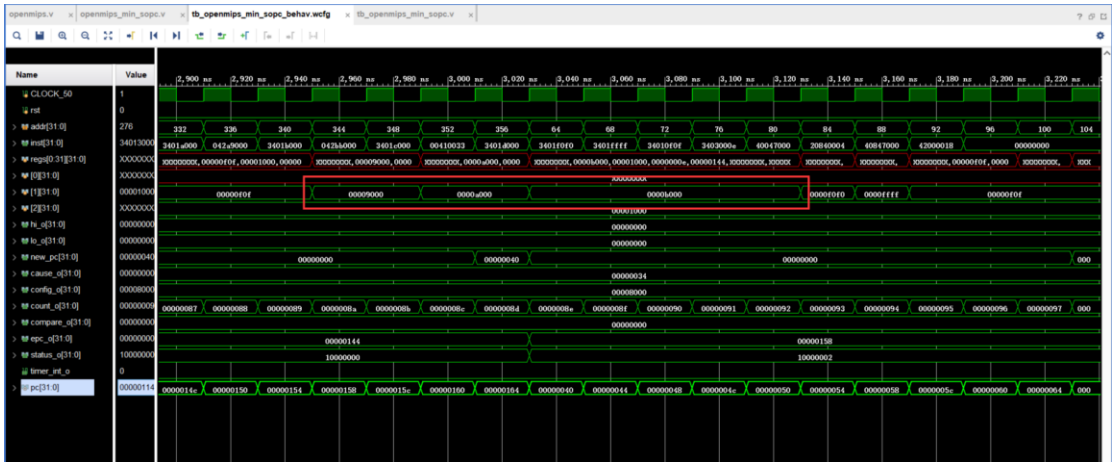
系统调用异常，取指为 0x40，会修改$1 的值。

8.2 测试自陷异常

使用下面的汇编测试程序：

```
    .set nomacro
    .global _start
_start:
    ori $1,$0,0x100      # $1 = 0x100
    jr $1
    nop

    .org 0x40
    ori $1,$0,0xf0f0     # $1 = 0x0000f0f0
    ori $1,$0,0xffff     # $1 = 0x0000ffff
    ori $1,$0,0x0f0f     # $1 = 0x00000f0f
    mfc0 $4,$14,0x0
    addi $4,$4,0x4
    mtc0 $4,$14,0x0
    eret
    nop

    .org 0x100
    ori $1,$0,0x1000     # $1 = 0x00001000
    ori $2,$0,0x1000     # $2 = 0x00001000
    teq $1,$2            # trap happen
    ori $1,$0,0x2000     # $1 = 0x00002000
    tne $1,$2            # trap happen
    ori $1,$0,0x3000     # $1 = 0x00003000
    teqi $1,0x3000       # trap happen
    ori $1,$0,0x4000     # $1 = 0x00004000
    tnei $1,0x2000       # trap happen
    ori $1,$0,0x5000     # $1 = 0x00005000
    tge $1,$2            # trap happen
    ori $1,$0,0x6000     # $1 = 0x00006000
    tgei $1,0x4000       # trap happen
    ori $1,$0,0x7000     # $1 = 0x00007000
    tgeiu $1,0x7000      # trap happen
    ori $1,$0,0x8000     # $1 = 0x00008000
    tgeu $1,$2           # trap happen
    ori $1,$0,0x9000     # $1 = 0x00009000
    tlt $1,$2            # not trap
    ori $1,$0,0xa000     # $1 = 0x0000a000
    tlti $1,0x9000       # not trap
    ori $1,$0,0xb000     # $1 = 0x0000b000
    tltiu $1,0xb000      # trap happen ecause $1=0xb000 < 0xffffb000
    ori $1,$0,0xc000     # $1 = 0x0000c000
    tltu $2,$1           # trap happen
    ori $1,$0,0xd000     # $1 = 0x0000d000
_loop:
    j _loop
    nop
```

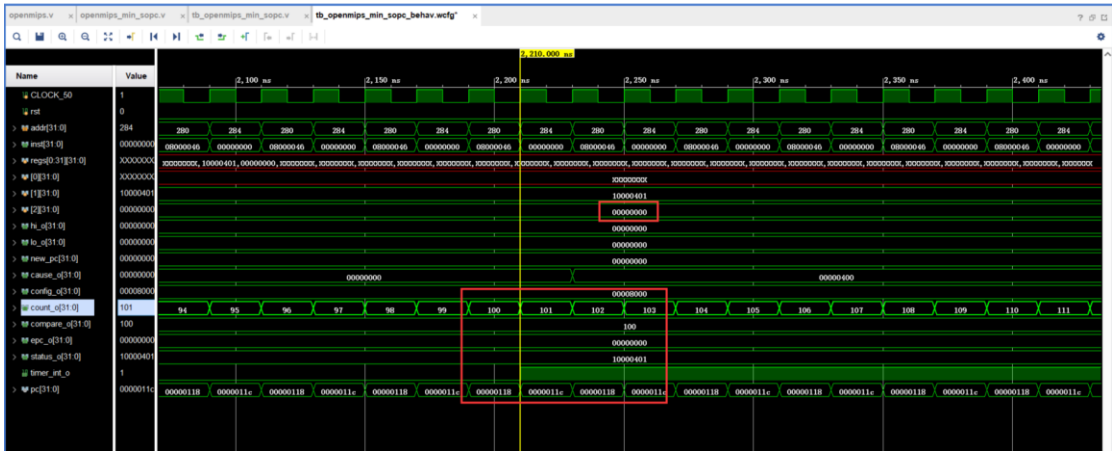




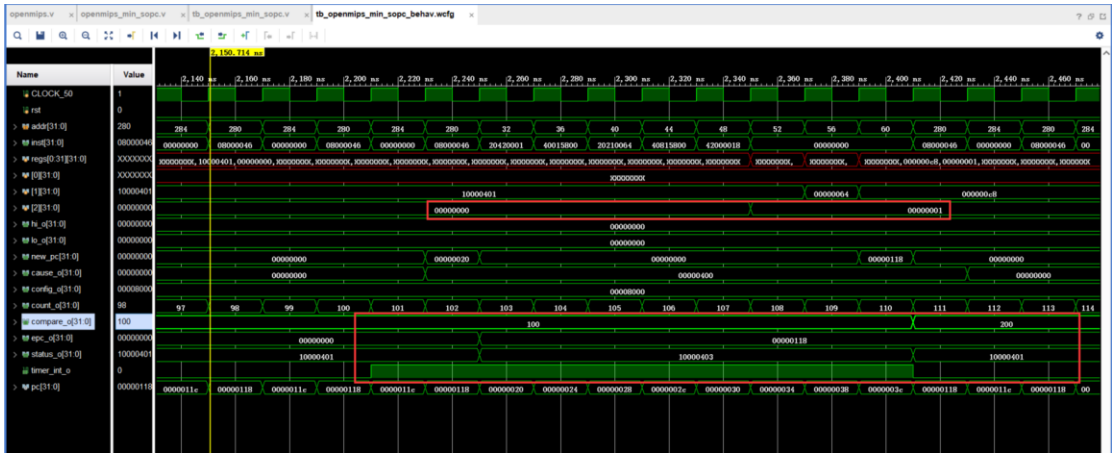
8.3 测试时钟中断

使用下面的汇编测试程序：

```
    .org 0x0
    .set noat
    .set noreorder
    .set nomacro
    .global _start
_start:
    ori $1,$0,0x100         # $1 = 0x100
    jr $1
    nop

    .org 0x20
    addi $2,$2,0x1          # $2寄存器的值加1
    mfc0 $1,$11,0x0         # 读取Compare寄存器的值
    addi $1,$1,100          # 增加100
    mtc0 $1,$11,0x0         # 再存回Compare寄存器
    eret
    nop

    .org 0x100
    ori $2,$0,0x0
    ori $1,$0,100
    mtc0 $1,$11,0x0
    lui $1,0x1000
    ori $1,$1,0x401
    mtc0 $1,$12,0x0         # 设置Status寄存器的值为0x10000401，表示使能时钟中断

_loop:
    j _loop
    nop
```



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。