实验 5 报告

第 32 小组
袁峥

一、实验任务（10%）

为 myCPU 增加例外与中断支持， 完成功能测试， 并支持运行一定的应用程序。

本次实验分两周完成， 需要完成：

(1) CPU 增加 MTC0、 MFC0、 ERET、SYSCALL(syscall 例外支持)、BREAK(break 例外支持)指令。

(2) CPU 增加 CP0 寄存器 STATUS、CAUSE、EPC、COUNT、COMPARE 和 BADVADDR。

(3) CPU 增加地址错、 整数溢出、 保留指令例外支持。

(4) CPU 增加时钟中断支持，时钟中断要求固定绑定在硬件中断 5 号上，也就是 CAUSE 对应的 IP7 上。

(5) CPU 增加 6 个硬件中断支持，编号为 0~5， 对应 CAUSE 的 IP7~IP2。

(6) CPU 增加 2 个软件中断支持， 对应 CAUSE 的 IP1~IP0。

(7) 完成 lab5_fun_1、lab5_fun_2 功能测试。

(8) 在 myCPU 上运行 lab4 的电子表程序， 要求能实现相同功能。

(9) 在 myCPU 上运行记忆游戏程序，要求能正确运行。

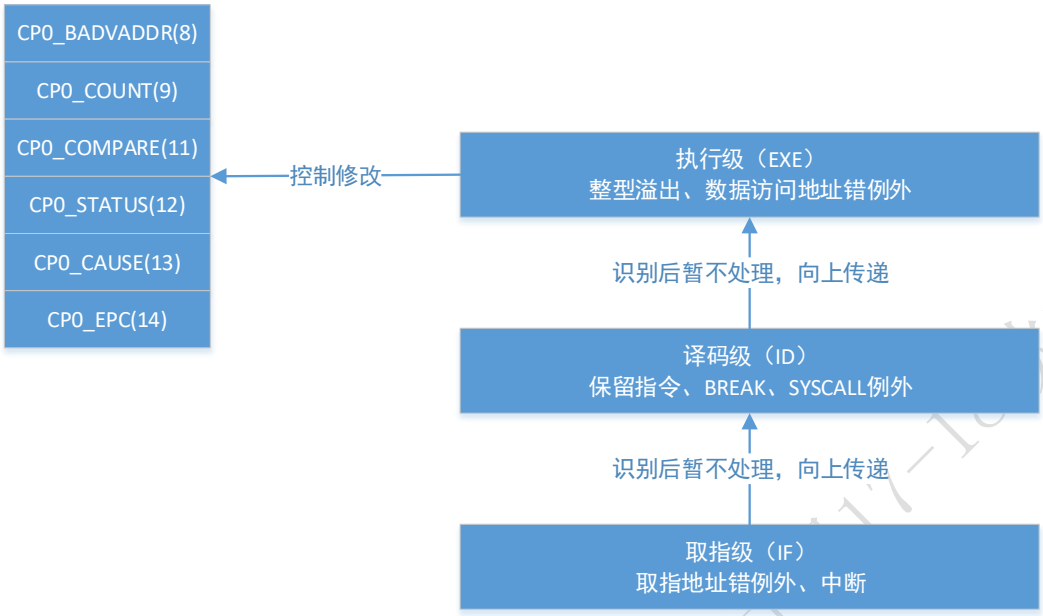
二、实验设计（30%）

这次的实验主要是在实验 3 中完成的 5 级流水线中加入对于中断和例外的支持。

在总体设计上，将增加的 6 个 CP0 寄存器全部都放在 cpu_top 中，由于可能同时需要更改多个 CP0 寄存器的值，因此每个 CP0 寄存器的读写都单独控制。在中断和例外方面，统一在执行级（EXE）进行处理，在取指级（IF）可以判断出取指地址错例外，在译码级（ID）可以判断出保留指令例外、BREAK 例外和 SYSCALL 例外，在执行级（EXE）可以判断出整数溢出例外、数据访问地址错例外，在 IF 和 ID 级判断出的例外将信号传递至执行级后统一处理，为了保持一致性，因此将中断也统一放在执行级处理，同时，所有的 CP0 寄存器的修改操作也同时在执行级完成。

在中断处理上，首先通过 cpu_top 读入的 hard_interrupt 信号，在时钟上升沿修改 CP0_CAUSE 的[15:10]，对应 6 个硬件检测。然后再将 CP0_CAUSE 与 CP0_STATUS 对应做与操作并结合 CP0_STATUS 的 EXL 和 IE 位后触发中断。

中断例外的整体处理流程如下图：



本次实验的整体设计比较清晰，但是存在许多细节上需要注意的地方。

1、 所有的 CP0 寄存器统一在执行级进行修改，如果在不同流水级分别进行修改，可能会发生同时读写同一个寄存器的情况。

2、 CP0_STATUS 和 CP0_CAUSE 寄存器在软件修改的时候并非所有位都可以修改，CP0_STATUS 只有[15:8]、[1:0]可以写，CP0_CAUSE 只有[9:8]可以写。

3、 通过查看 MIPS 手册得知，ERET 不存在延迟槽，ERET 指令的下一条指令自动不执行，因此在译码级译码识别出 ERET 指令后，进行指令跳转的同时，也要清空取指级当前的指令。

4、 需要注意指令的执行顺序，如果跳转指令和延迟槽指令同时触发例外，由于保证在跳转之前延迟槽指令必须完成，因此需要先处理延迟槽指令的例外。由于跳转指令触发的只可能是取指地址错例外，因此将取指错例外放到取指级处理而不是在译码级或者执行级处理，这样可以保证指令执行的顺序满足要求。

5、 触发中断时，将中断绑定到当前的取指级指令但暂不处理，而是继续往下级流水传递，直到执行级统一处理。

6、 在执行级开始处理中断和例外时，需要同时修改 CP0_CAUSE 的 BD 和 ExcCode 位，CP0_EPC（如果在延迟槽需要减 4）、CP0_STATUS 的 EXL 位。

7、 CP0_COUNT 每两个时钟周期需要自动加 1。硬件需要一直检测 CP0_COMPARE 和 CP0_COUNT 是否相等，一旦相等，CP0_CAUSE 的 TI 位需要置 1。在使用 MTC0 写 CP0_COMPARE 时，需要同时自动清除 CP0_CAUSE 的 TI 位。

8、 当 CP0_STATUS 的 EXL 位为 1 时，所有硬件和软件中断被屏蔽，CP0_EPC、CP0_CAUSE 的 BD 位在发

生新的例外时不做更新。

9、 在发生地址错例外时，需要将出错的虚地址存入 CP0_BADVADDR。

三、实验过程（60%）

（一）实验流水账

11 月 16 日晚 8 点至次日 2 点修改代码并调试。

11 月 17 日 10 点至 18 点调试程序，同时发现了测试程序的多处错误。最终通过功能测试，上板情况功能测试和记忆游戏正常，电子表出现会异常重置的情况。

11 月 20 日下午 4 点至 6 点通过强制激励在仿真时发现错误并修改，完成电子表的上板测试。

（二）错误记录

1、错误 1

（1）错误现象

仿真时运行一段时间后突然时间不再增加，而且波形也不改变。

（2）分析定位过程

通过查看文档“LEC04_仿真调试说明”，得知可能原因是组合环路。由于这是在上次实验的基础上修改的，因此将代码进行版本回退后，一点一点修改并进行仿真，最终发现导致组合环路的语句。

（3）错误原因

在 cpu_top 中汇总中断例外信号时使用了执行级传递的 EXE_syscall，同时在执行级产生 EXE_syscall 信号又使用了 cpu_top 中的中断例外汇总信号。

（4）修正效果

在执行级增加了一个寄存器，将 cpu_top 传递的输入信号存入寄存器后再使用。

（5）归纳总结

在设计时使用不同流水级之间传递的信号时需要格外主要，不要相互使用以避免组合环路，必要时可以使用寄存器进行切分后再使用。

2、错误 2

（1）错误现象

MFC0 指令读取 CP0_STATUS 时与 golden trace 不一致。

（2）分析定位过程

报错的位置是在中断处理程序中，从波形继续向前跟踪发现了 CP0_STATUS 的修改与自己的设想不一样。

（3）错误原因

由于 CP0_STATUS 只有一些位可以用软件修改，因此在写的时候使用了掩码，但是在掩码的操作上出现了错误。

（4）修正效果

重新手算了一遍修改特定位的规则，修改了代码，重新仿真后功能正常。

（5）归纳总结

对二进制运算还不够熟悉和细心，对于只能修改特定位的情况，假设 A 为原寄存器的值，可以写的位置 1 的掩码为 mask，新写入的数据为 W，那么修改后的数据应该为(A &~mask) | (W & mask)。

3、错误 3

（1）错误现象

CP0 寄存器的值错误，与 golden trace 的值不相同。

（2）分析定位过程

通过跟踪仿真的波形，发现在修改时传入的值不正确。

（3）错误原因

仔细分析了前后的指令，发现 MTC0 的前一条指令是 ADD 指令，且目的寄存器与 MTC0 使用的寄存器相同，发生了数据相关，而当时的设计是在译码级就更新 CP0 寄存器，此时 ADD 指令还没有完成计算。

（4）修正效果

在这里有两种修改方式，一种是从执行级的计算结果前递至取指级，继续使用在取指级修改 CP0 寄存器的策略。另一种是将修改 CP0 寄存器的操作延后至执行级，此时 ALU 也完成运算。最终经过全局考虑，采用了后一种修改方式。

（5）归纳总结

每次在对以前的代码进行修改和更新时，仍需要考虑一些以前的问题，比如数据相关和前递的情况，必要时需要增加前递方案或者修改一些流水级的操作，如将写 CP0 寄存器延后至执行级。

4、错误 4

（1）错误现象

MTC0 指令读出的 CP0_CAUSE 的结果不正确，与 golden trace 不符。

（2）分析定位过程

对比了两者 CP0_CAUSE 的结果，发现触发的例外类型不对。查看汇编代码后发现，产生例外时执行级为 J 指令，但跳转的指令地址错误，没有对齐，此时触发取指地址错例外，但是延迟槽中的指令为 ADD，同时该指令的结果会触发整型溢出例外，按照指令规范应该先指令延迟槽指令，因此实际应该触发整型溢出例外。

（3）错误原因

当时的设计考虑到取指地址错误可以在执行级发现，因此在执行级直接触发该例外。但是遇到如上的情况会导致指令执行的先后顺序不对，因此需要重新思考触发例外的位置。最后修改为在执行级不考虑取指地址是否正确，而且将地址依旧送到取指级，在取指级再判断指令地址是否正确，如果不正确，在取指级触发指令地址错例

外，并且不立刻相应，而是将例外向上传递到执行级再处理。这样，由于此时延迟槽指令已经到达译码级，因此延迟槽指令会优先于触发指令地址错例外的指令到达执行级，也就先相应延迟槽所触发的例外。

（4）修正效果

对程序进行了较大幅度的修改，最终仿真测试通过了之前错误的地方。

（5）归纳总结

在设计的时候没有把情况考虑全面，导致完成设计和代码编写后发现流水级的分配上出现了问题，导致代码的修改幅度较大，也比较费时。以后在编码代码前要尽量把各种情况都考虑全面，保证大的思路上不出现错误，减少后序代码的修改量。

5、错误 5

（1）错误现象

MFHI 指令的结果与 golden trace 的结果不同。

（2）分析定位过程

考虑到 MFHI 指令在前面的实验中已经反复验证过，因此该指令的实现应该没有问题。因此向前查找修改 CP0_HI 寄存器的地方，发现前面的一条保留指令意外触发了除法，从而导致除法完成后修改了 CP0_HI 寄存器。

（3）错误原因

将保留指令的指令码翻译成二进制后发现，其中的操作码与程序中除法的操作码相同，因此被误认为是除法指令。因此需要在进行 ALUOP 判断时同时考察该条指令是否已经被标记例外。

（4）修正效果

修改了 ALUop 操作数的相关判断逻辑后，仿真通过该测试点。

6、错误 6

（1）错误现象

在电子表上板测试时，意外发现会偶尔导致数码管置零后重新开始计时。

（2）分析定位过程

这个错误的定位花费了很多时间，中间大概整整思考了一天，由于是上板发现的错误，无法一下定位到错误点，而且错误的现象十分匪夷所思。另外，电子表的软件代码也是自己编写的，不排除在软件代码上有错误，因此无法准确判断是硬件还是软件的问题。

最后决定先反复上板观察并复现错误，看看具体是在什么情况下触发错误。进行多次测试后发现，如果在设置模式下，先长按一个按钮使某一位连续增加，松开后再按一下按钮，此时会触发数码管突然置零的情况。成功复现错误后，考虑使用在仿真中强制加激励的方法来观察波形和寄存器的值。按照上板时发生错误的方法，编写了激励代码后运行仿真程序，发现程序运行过程中存放当前时间的寄存器果然被意外清零。

查看了此时的其他信号的值，此时的 PC 值跳转到程序初始化的地方进行寄存器值初始化，因此寄存器被清零，同时发现 CP0_EPC 寄存器的数据异常，里面存放的地址是一条 ERET 指令的地址。继续往前跟踪 CP0_EPC 寄存

器的变化，最终找到了问题的根源。

（3）错误原因

最初导致错误发生的情况如下：在长按按钮时，一次中断处理退出时，当 ERET 指令在译码级被识别后，将 CP0_EPC 的值送至 next_pc，同时打开中断，下一拍时，ERET 指令来到执行级，同时由于打开中断且按键依旧被按住，因此重新触发硬件中断，由于此时的设计是将硬件中断绑定在执行级，因此将此时执行级的 PC 送到 CP0_EPC，而此时执行级的指令为 ERET，因此导致了如上的错误。

（4）修正效果

反复思考后，将硬件中断的标记移至取指级，同时标记后不立即处理，而是随着流水线传递到执行级后，进行中断例外的统一处理，这样可以规避上面碰到的问题。

（5）归纳总结

感觉这个问题在设计的时候也不会考虑到这么细，因此如果设计不当的话也无法规避。这次调错学会了如何软硬件协同分析，尤其是在上板时发现错无法立刻定位错误时。先通过反复尝试复现错误并发现触发错误的规律，然后通过编写仿真程序强制激励来在仿真时模拟错误发生，在通过跟踪波形分析最后发现并解决问题。

四、实验总结

（一）组员：袁峥

由于一些感情问题和之前的队友闹崩了，因此决定两人分组各自做实验。由于之前的代码其实基本都是我一个人编写的，内容也十分熟悉，因此问题不大。

这次实验的任务是在之前的五级流水基础上增加中断和例外的支持。整体思路比较清晰，但是细节情况太多而且十分繁琐，因此难免有考虑步骤，也导致此次实验在调试时花费了挺长的时间。

由于实验调试开始的比较早，因此连续发现了测试程序的三处错误，这也是导致调试时间较长的原因之一。更惨的是，在踩完所有雷之后，在第 93 个测试点发现取指错例外的报出安排在执行级是有问题的，思考后决定应该放在取指级再报出例外，因此需要对修改的程序进行大规模修改，几乎相当于重写了一遍。这也告诫自己下次做实验时，在编写代码以前尽可能将要考虑的情况考虑全面，避免出现这种大规模返工。

另外一处卡顿时间较长的地方是电子表在第一次上板时意外发生了重置零的情况，具体情况已经在上述报告中描述。一开始觉得无从下手，一方面电子表的程序在上次实验编写时我认为考虑的情况应该已经比较全面了，不太会出现程序上的问题，另一方面，CPU 硬件方面单纯从上板结果也无法直接看出错误的原因。好再后来反复实验后复现了错误，并通过将类似的操作写成仿真的激励，从而在仿真时也复现了错误，最终得以解决问题。