

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 计算机体系结构 |
| 姓 名： |  |
| 学 院： | 计算机科学与技术学院 |
| 系： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 学 号： |  |
| 指导教师： |  |

2023年 10 月 17 日

**浙江大学实验报告**

课程名称： 计算机体系结构 实验类型： 综合

实验项目名称： 流水线CPU的中断与异常处理

学生姓名： 专业： 计算机科学与技术 学号：

同组学生姓名： 指导老师：

实验地点： 曹西301 实验日期： 2023 年 10 月 17 日

1. 实验目的和要求

实验任务梗概：

Design of Pipelined CPU supporting exception and interrupt.Verify the Pipelined CPU with program and observe the execution of program.

实验目标：

1. Understand the principle of CPU exception and interrupt and its processing procedure.

2. Master the design methods of pipelined CPU supporting exception and interrupt.

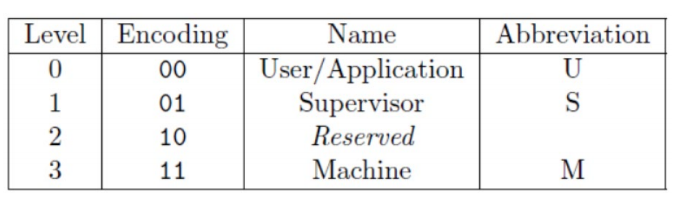
3. master methods of program verification of Pipelined CPU supporting exception and interrupt.

1. 实验内容和原理

在本次实验中，我们需要处理流水线CPU运行时可能出现的部分异常与中断信号，因而我们需要一些前置的知识，来说明流水线CPU的实际运行原理，以及其与硬件之间的交互过程。

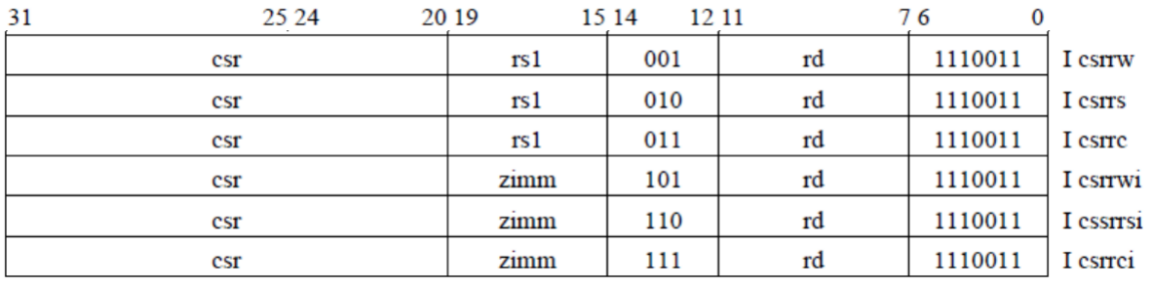
2.1 计算机系统的模式

在riscv架构中，我们主要有三种已经定义的模式以及一种保留定义的模式，其中定义的三种模式为机器模式、系统内核模式以及用户模式。模式的具体encode结果如下：



2.2 在本次实验中，我们需要使用到一些特殊的寄存器，这些寄存器不同于我们先前设计的数据存储寄存器；这些csr寄存器虽然也用于存储数据，但这些数据与处理器甚至机器的运行极度相关，因而这些寄存器需要额外的保存于设定手段。

对于riscv而言，架构已为我们额外提供了一些指令，帮助我们对这些寄存器进行操作运算，更好的调度系统运行的状态。

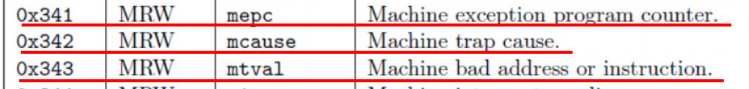


一些操作指令已在上成列。其中csr表示我们需要读或写的csr特殊类寄存器，rs1表示我们需要将csr寄存器中的数据写入的目标寄存器，rd寄存器记录着我们需要写入csr特殊寄存器中的数值；其中的imm表示立即数，需要扩展至32bit才能放入csr寄存器中。

2.3 部分重要的csr寄存器

在这里，我们给出一些本实验需要修改以介导处理器（流水线CPU）正常运行的csr寄存器，并阐述其功能。

37f8bd83242fa0eefc513ed75563865315251fb508e1e59bb0d382f90bf2d9



其中第一列为csr寄存器对映的存放地址，第三列为csr寄存器的一般名称，最后一列则简要介绍了该csr寄存器的基本功能。

2.4异常、中断类型及处理

在本实验中，我们主要考虑以下四种可能发生的异常与中断。

1. 外部中断，通过按键触发的异步外部中断。
2. 访问错误异常，当物理内存的地址不支持访问类型时发生（例如尝试写入ROM时）。
3. 环境调用异常，在执行ecall指令时发生。
4. 非法指令异常，在译码阶段可能发现无效操作码时发生。

在处理异常与中断时，我们需要实现一下硬件如下的状态转换过程：

1. 异常指令的PC被保存在mepc中，PC被设置为mtvec。mepc指向导致异常的指令；对于中断，它指向中断处理后应该恢复执行的位置。
2. 根据异常来源设置mcause，并将mtval设置为出错的地址或者其它适用于特定异常的信息字。
3. 把控制状态寄存器mstatus中的MIE位置零以禁用中断，并把先前的MIE值保留到MPIE中。

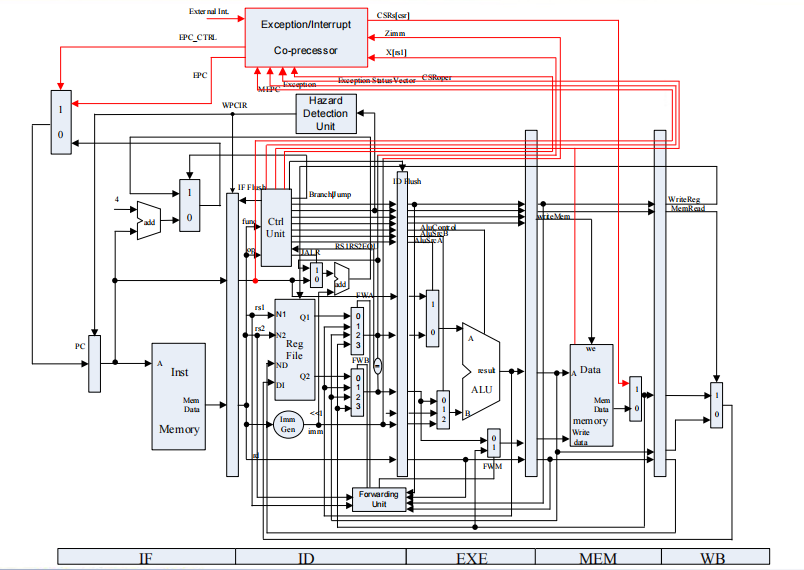
4.发生异常之前的权限模式保留在mstatus的MPP域中，再把权限模式更改为M。

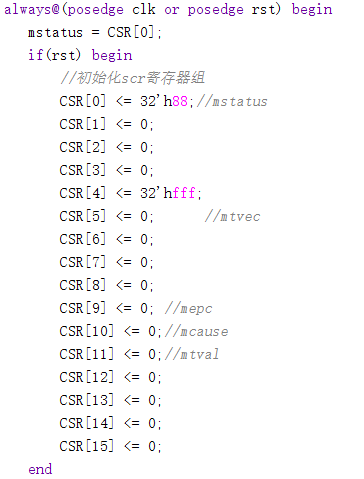
1. 实验过程

在本实验的实现任务中，我们主要是基于已有的流水线CPU框架，进行中断及异常的识别与处理模块的设计。主要是ExceptionUnit模块与其中CSRRegs寄存器存储模块的实现。

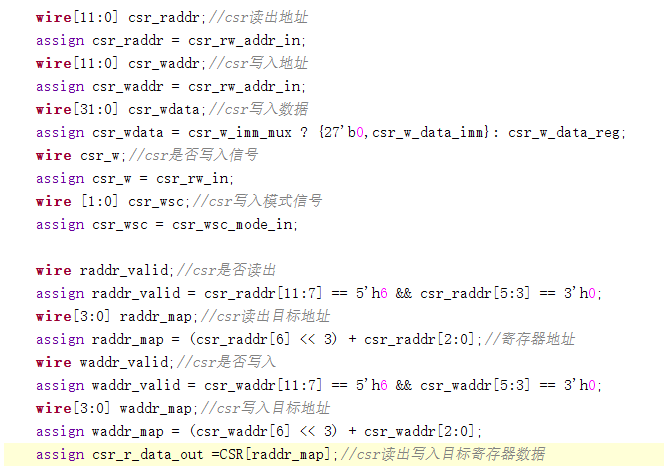
但在具体代码实现的过程中，我们发现CSRRegs模块每次只能读出及写入一个同样的寄存器，这对于处理中断或异常而言，是极其拖沓的。因而我考虑使用直接释放CSRRegs模块的方法，让CSRRegs寄存器在ExceptionUnit模块中实现，这也方便我们对CSR寄存器的直接修改与使用。

具体的逻辑电路示意图如下：

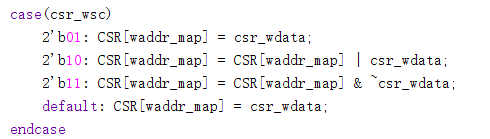


 在具体的代码设计中，我们先实现csrr等对csr寄存器进行读写操作的伪指令的功能。首先，我们需要定义一个CSR寄存器组，其中需要记录多个CSR寄存器的信息，并进行初始值的设定。

接着，我们需要定义一些变量，用于记录需要读取的CSR寄存器的需要，以及写入CSR寄存器的数据、CSR寄存器的数据取出等。

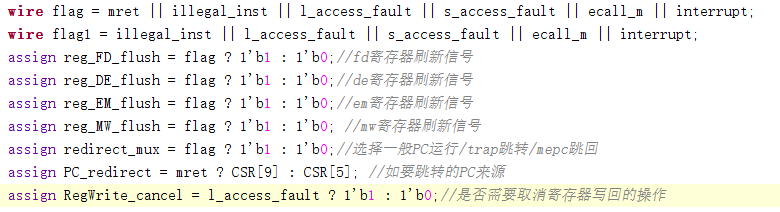


另外，在写入CSR寄存器时，我们需要根据不同的mode进行不同的写入模式选择。

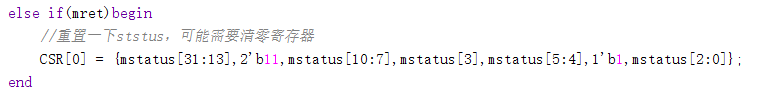


接下来，我们需要对异常及中断信号进行捕捉与处理。在本实验中，我们主要需要捕捉mret（中断返回）、illegal\_inst（错误指令） l\_access\_fault（L型指令存入）、s\_access\_fault（S型指令存入越界）、ecall\_m（自发的内部中断）、interrupt（外部中断）六类中断及异常。

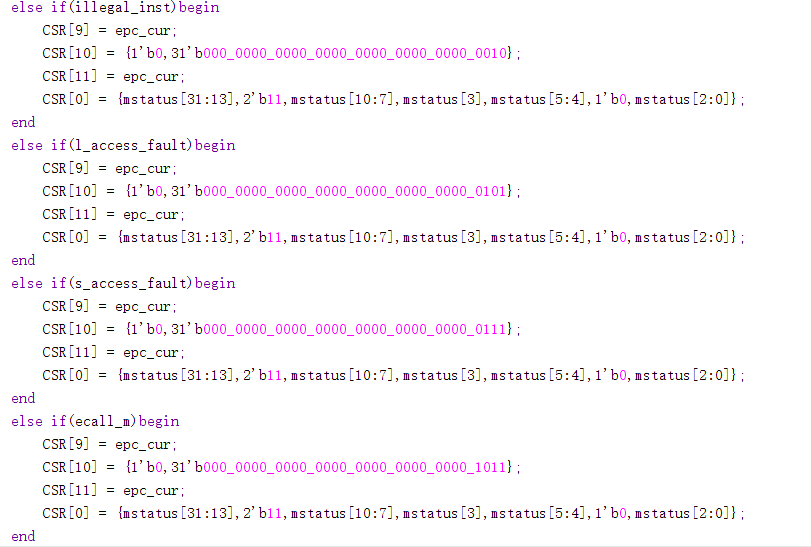
在本实验的实现中，我们认为WB阶段指令为被打断的指令，因而我们需要在中断或异常触发之时，清空所有IF/ID/EXE/MEM的寄存器信息，并记录WB阶段的PC信息。因为我们认为WB阶段的PC指令已经被完全执行了，但是其后的指令并没有进行，因而我们需要将这些指令操作带来的效果归零，并在结束中断处理后，重启这些指令的运行。



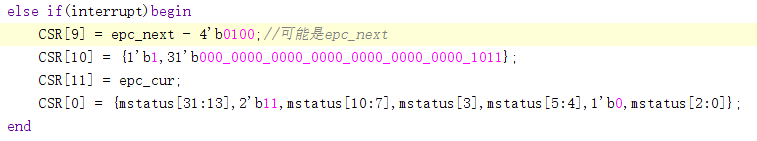
在这之后，我们就需要使用多个判断，根据不同的异常或中断类型，来选择不同的处理方式。其中，除mret外的异常与中断都会将mstatus寄存器中的中断可行位置置0，并将原先中断可行位置的数值放置于MPIE位置上；而mret则是需要建立中断可行性，因而需要将mstatus寄存器中的MIE位置置1。



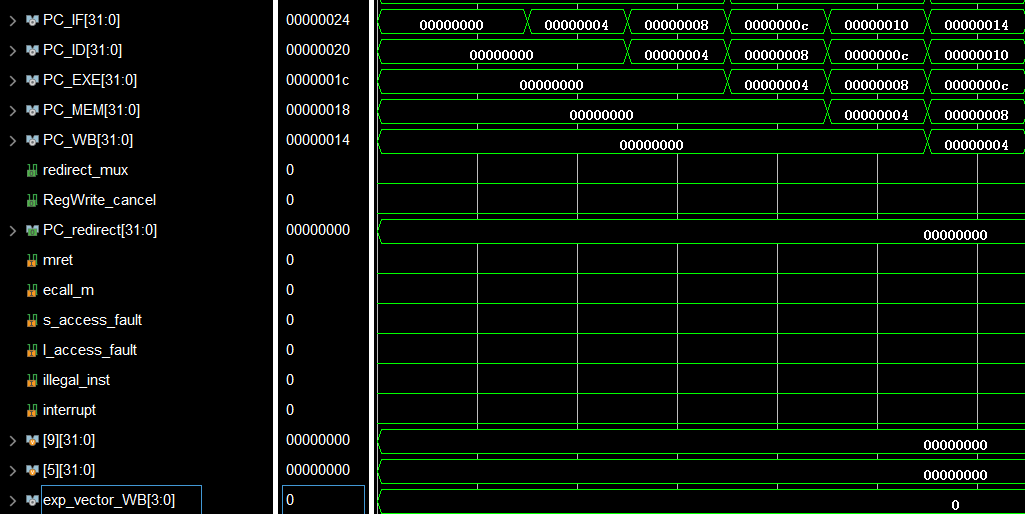
另外，对于异常处理而言，我们需要将当前WB阶段的PC指令放入mepc寄存器中存储，在中断程序中，trap会自动实现PC + 4的功能；因而我们可以顺利获得需要继续执行的下一条的指令信息。另外，对于不同的异常触发而言，我们需要在mcause寄存器中设置不同的取值，以区分这些不同的引发因素。



但是对于中断而言，我们需要设置mepc为当前WB阶段前最近未被flush的阶段指令，这可以很好的帮助我们在中断结束后回到需要继续执行的指令位置，而不至于随意前侧跳转，这可能会触发flush指令的运行故障。



1. 实验结果分析
   1. 仿真结果

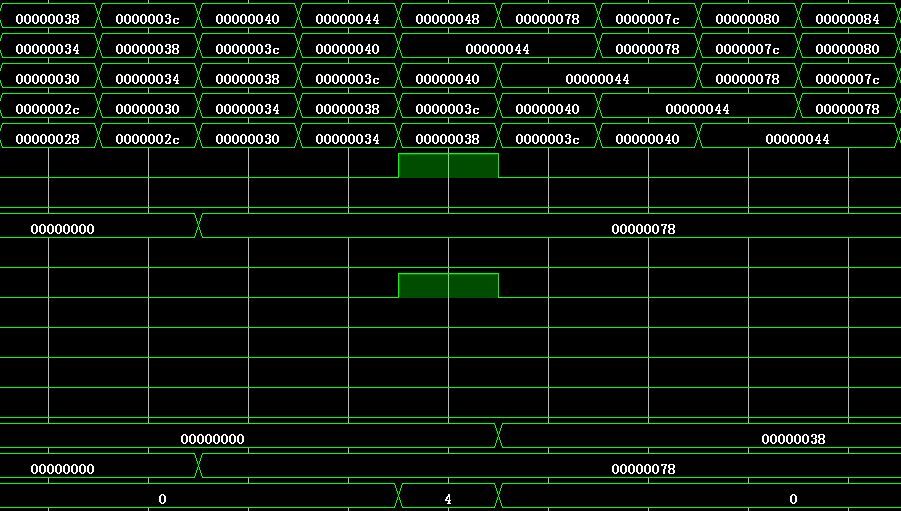
以下是部分仿真结果的测试截图。

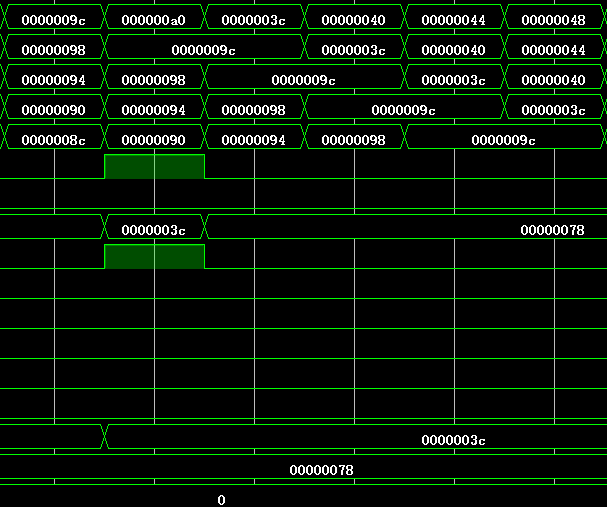
我们先进行这些选取验证仿真结果的信号解释。对于前五个PC\_XX而言，记录着流水线CPU五个阶段中不同阶段正在执行指令的PC地址。可以看到，随着仿真的开始，处于IF阶段的PC执行不断随着时钟周期向后移动。另外，redirect\_mux信号表示当前PC的选择信号，是简单的选择PC + 4来进行操作还是选择由CSR寄存器中取得的PC\_direct信号。

RegWrite\_cancel这一信号介导着是否允许写入寄存器，这对于中断操作的处理指令nop插入很重要；另外，mret、ecall\_m、s\_access\_fault、l\_access\_fault等相连的六个信号，记录着当前WB阶段的PC指令发生着什么类型的中断或异常。

最后，我也是选用了CSR中的5号与9号寄存器，来详细追踪中断跳跃的具体PC值以及对映异响操作exp\_vector\_WB信号。

第一次PC跳跃结果如下：

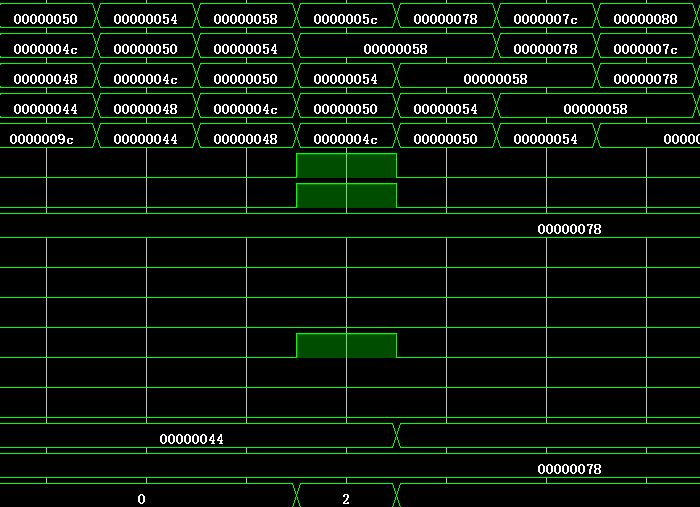


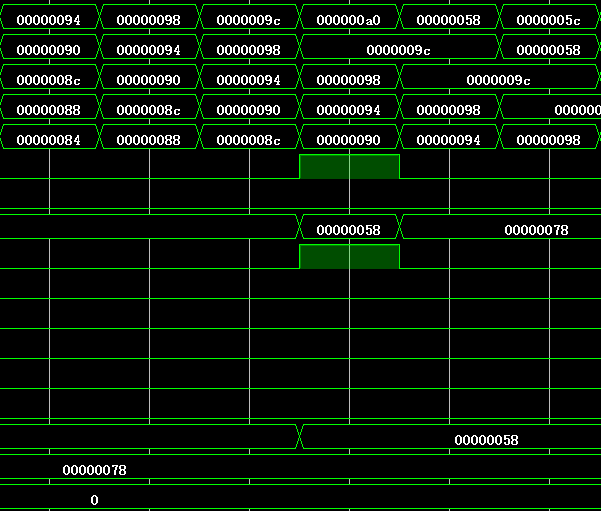
我们可以从结果截图中看到，当IF阶段的PC运行到0x00000048时，因为处于WB阶段的PC指令是ecall，因而触发了中断信号。此时redirect\_mux被置位为1需要进行PC的CSR寄存器中内容跳转，对映更多ecall信号也被置1。而我们的9号寄存器，则保存了ecall触发地址的代码，便于后续中断处理结束的返回，而5号寄存器则保存了中断处理程序trap指令开始的PC地址。

接着，在trap程序处理结束后，我们的程序顺利从a0地址跳转到了3c地址，也就是我们保存的PC地址后的PC地址位置，从而我们可以继续运行我们设定的代码。

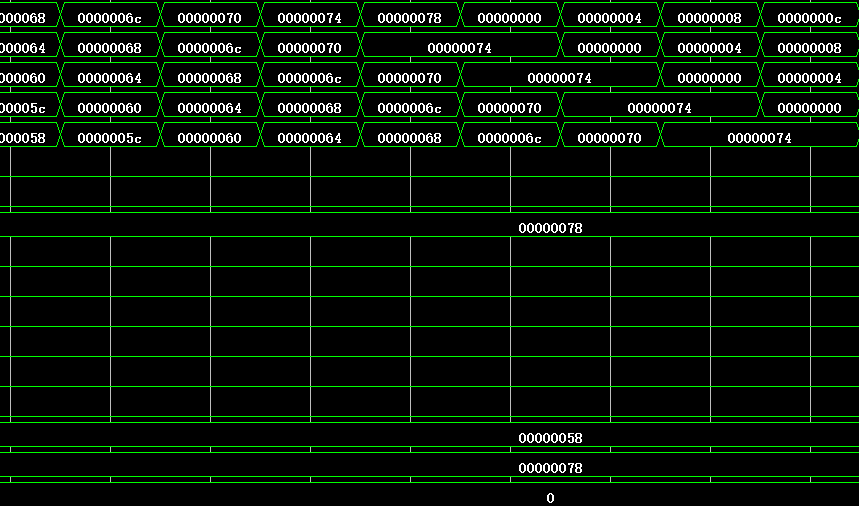
因为这个跳转的动作由mret语句介导，所以mret的中断信号被置1，且redirect\_mux的PC选择信号也被置1。

我们在此继续陈列一些仿真结果截图：

第三次跳转

最后一次中断处理跳回

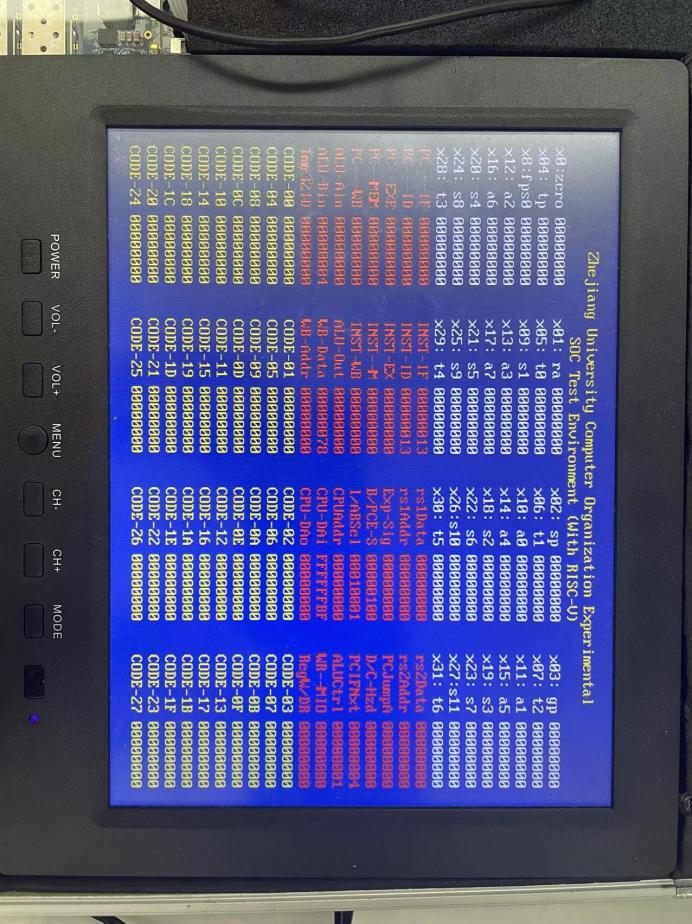
最后，我们聚焦最后一次PC的跳回。我们可以看到，在最后一次跳回时，我们执行的是jal x指令，我们的程序也正确的跳转到了0x00000000的位置。另外，因为这条语句不是中断或异常触发的，因而没有特殊信号被置位。如果我们继续往下查看其他信号，可以看到Branch\_ctrl信号被置位为1，意味着PC指令会进行跳转运行。

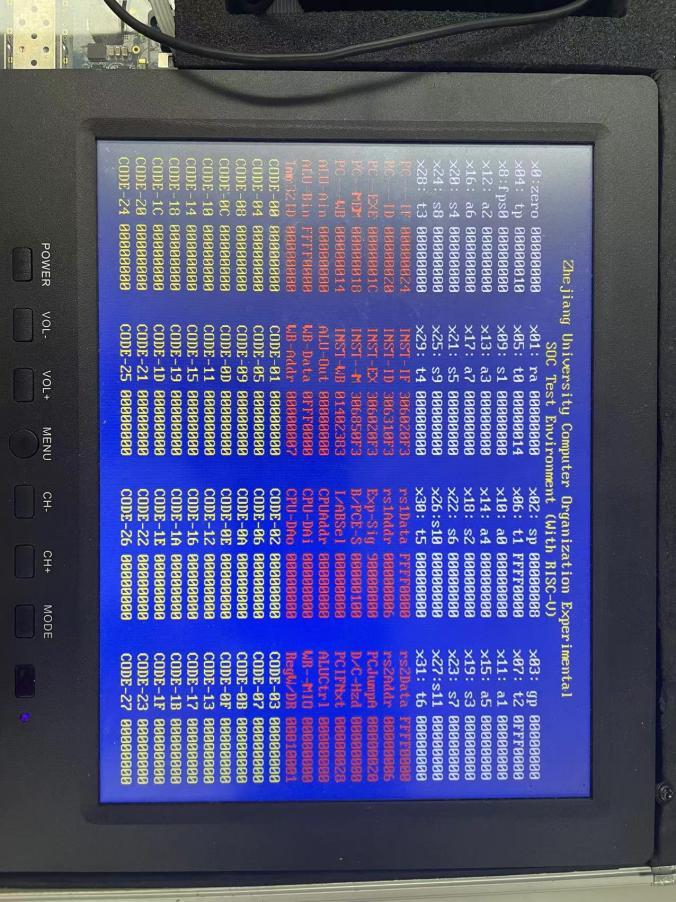


4.2 上板结果

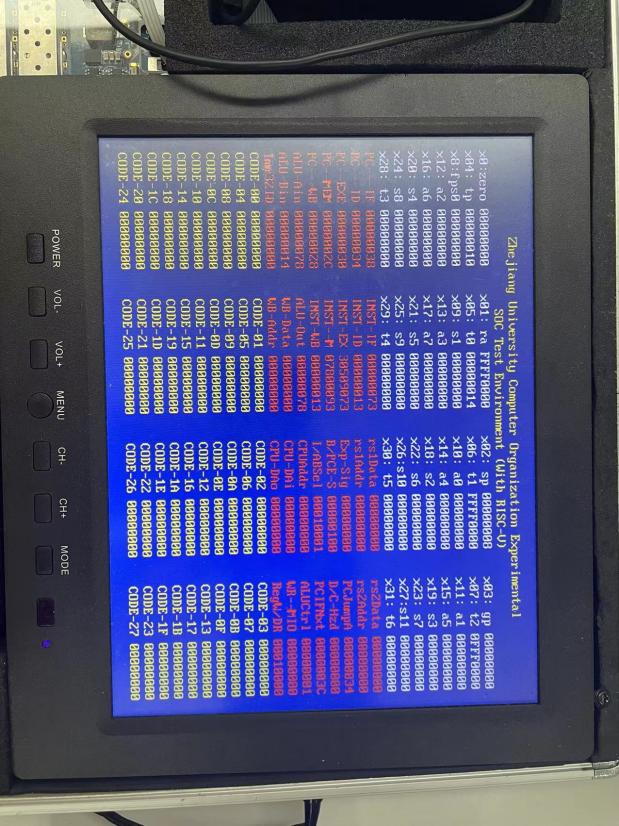
这里陈列部分实际上板的关键信息截图。

代码的载入与正常运行：

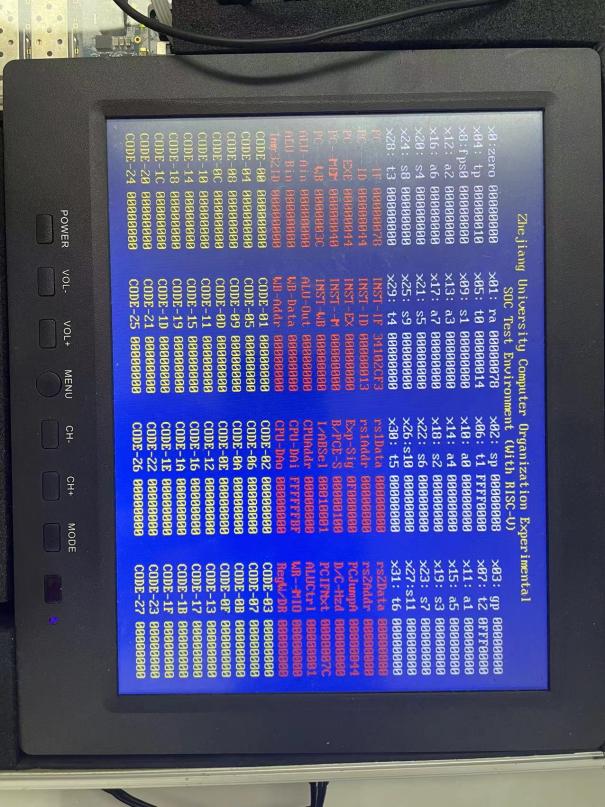




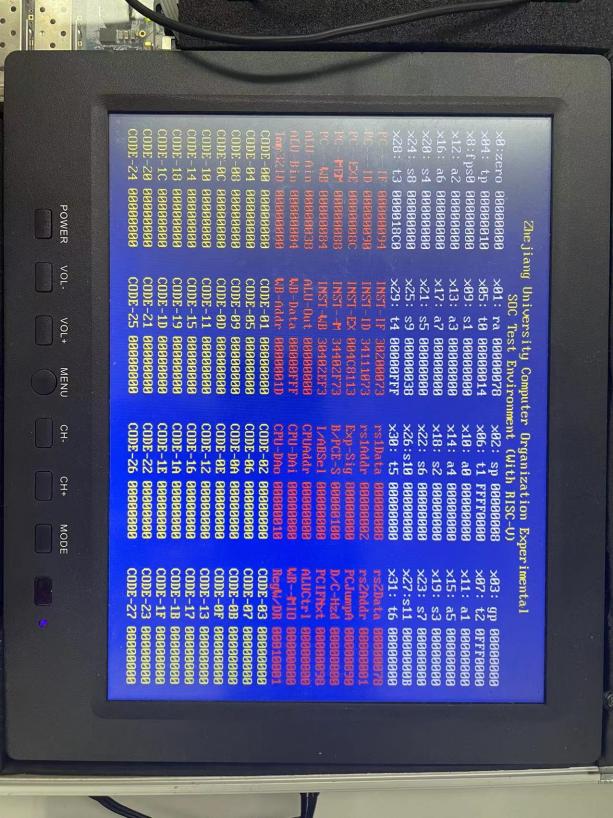
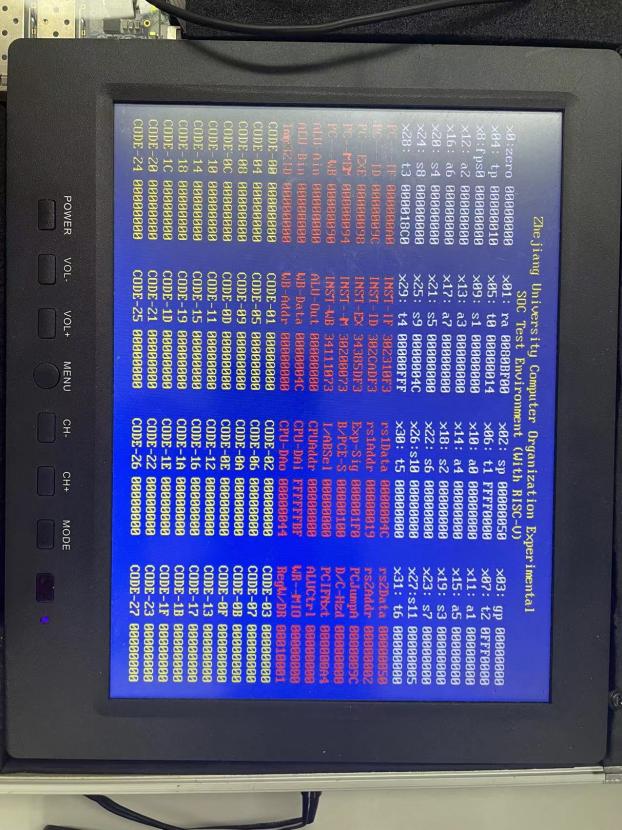
以下是ecall指令进入IF阶段时流水线上下文的写照。可以预见，流水线CPU并没有进行指令跳转，这与我们在WB阶段统一介导中断、异常的函数处理设计一致。



我们继续单步运行流水线CPU，当ecall指令进入WB阶段后，我们发现IF阶段的指令已经被中断/异常处理程序对映的首个指令所占用。中断/异常处理程序PC指令跳转成功，具体结果见下：

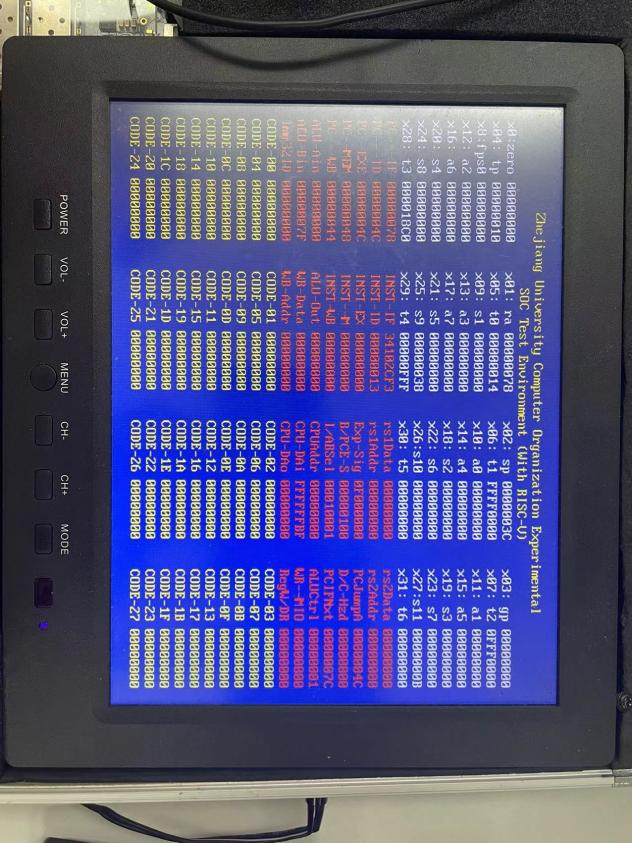


接下来，当trap段的中断/异常处理程序执行到mret语句时，流水线CPU也不会进行及时的反映；直到mret进行到WB阶段时，流水线CPU才触发了跳转操作。

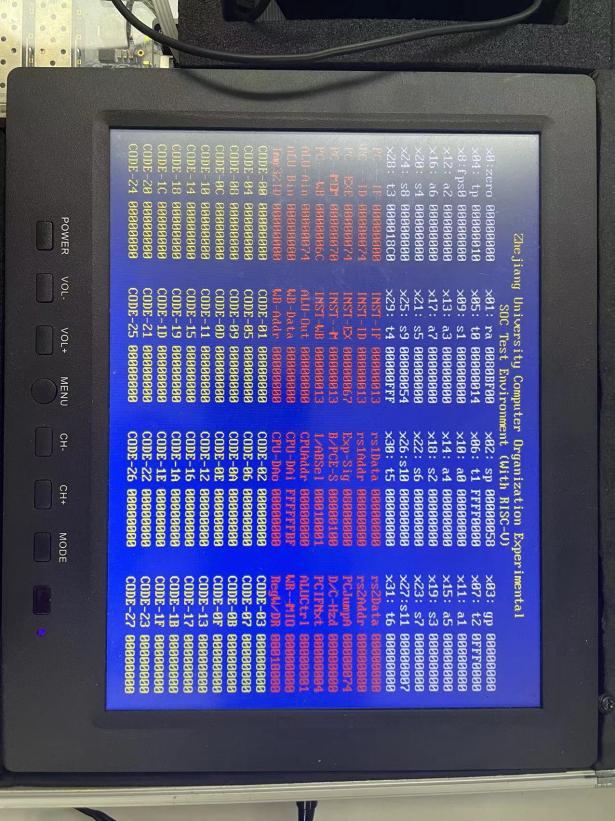


Trap中断/异常处理程序运行结束后，我们IF阶段的PC值成功跳回了异常产生语句的发生点。

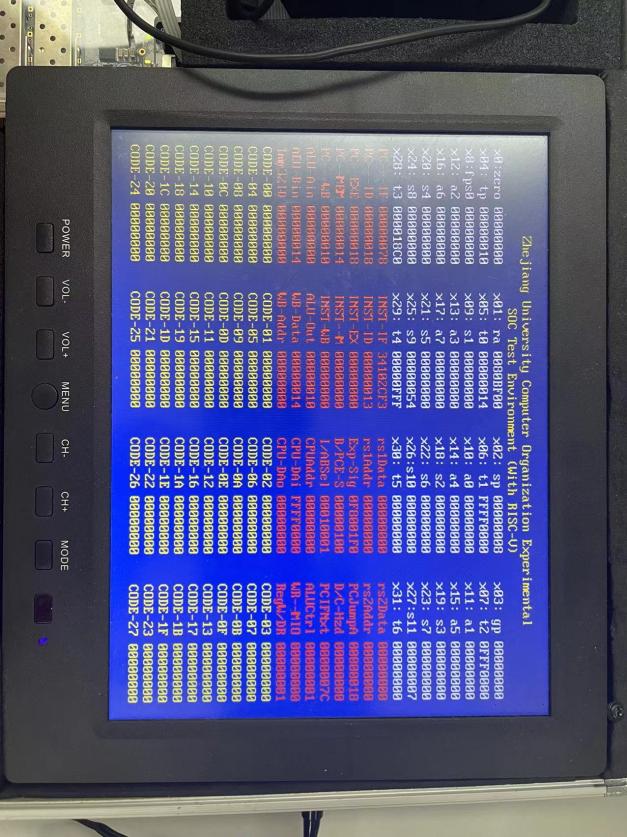
在此之后触发的异常跳转、异常处理以及处理后成功跳回过程如下：



在执行完所有代码之后，流水线CPU进行jr伪指令的处理，成功介导了指令的返回。

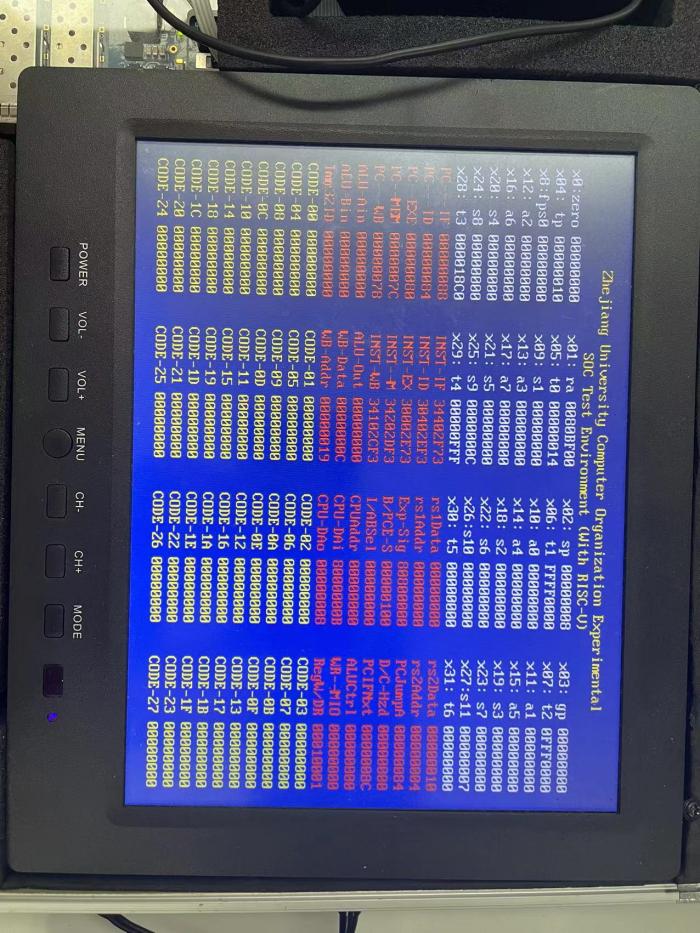


最后，我们尝试interrupt外部中断的处理结果。在初始化各个系统csr寄存器后，我们使用外部按钮触发interrupt外部中断指令；此时，继续进行单步运行指令，我们可以看到程序进入中断：



在一般的流水线CPU运行中，当中断、异常触发并进入相对映的处理程序时，我们需要关闭中断触发的可能性。因为在中断处理过程中触发中断是复杂及危险的事情。

我们在中断处理的过程中，进行了再次中断的尝试，发现中断不能再被二次触发，符合我们的设计需求。



1. 讨论与心得

对于这次实验，总体而言算是顺利，没有想象中的那么困难，也可能是我修改了其中的CSRRegs模块，使其直接拼接于ExceptionUnit模块内，方便了很多CSR寄存器之间的操作。

但是对于这次实验，我还有一些残留的问题，或者说是此次框架下一些残存的bug。首先，ROM中指令存放的缺失问题。这次框架下给出的ROM测试代码，在中断处理完成后，使用mret语句进行返回，但是mret语句后没有添加无效语句，因而在我使用的方法时会导致系统进入无指令读取的状态，可能导致跳转时错误的发生。

另外，Interrupt指令PC的保存，也成为一个界定的难题。在本次试验中，我们使用统一的trap程序对异常与中断信号进行处理，在处理函数中，都是简单的对PC进行 +4 操作。这个做法对于异常而言是正确的，因为我们只需要记录当前的PC指令，就可以在异常处理结束后顺利的跳转至我们需要执行的后一条指令。但在Interrupt中，可能PC本身已经被flush掉，因而再给其 +4 也不能跳转至正常的执行位置。这时，我们就需要PC\_next来记录需要跳回的PC地址，但是，这一地址需要手动 -4，这很关键，因为我们的trap程序会统一的为我们进行PC + 4的操作。

其他则是S型指令的问题，当一条语句在WB阶段被中断打扰时，我们认为WB阶段的指令已经结束；而IF/ID/EXE/MEM阶段的指令并未执行完全，因而需要撤回，以等中断处理结束后再重新启动执行。这是，如果处于MEM阶段的指令为S型指令，在没有任何干扰的前提下，S型指令会直接写入memory中，但按理而言我们的S型指令不能产生任何效果，这里就会存在一个S型指令越期产生效果的问题，可能会对后续的trap处理带来一定的干扰。因而，在框架中，可能还是需要一个额外的memory写入拒绝信号，来阻止此类行为的发生。

当然除却这些问题之外，在本次的实验中，我也收获满满。在本次实验过程中，我也体会到了代码不规范带来的极大问题，最直接的，就是implementation的失败。在此也将失败的诱因做一个总结：

1. 时钟边沿触发寄存器的读取，如果轻易将output信号改为寄存器，可能会使控制信号的产生延后一个时期（上升沿记录左侧的运算值），可能会导致响应不及时的情况；因而尽可能使用wire作为output输出。
2. 寄存器多次赋值问题，尽量不要为了代码的简便，先给寄存器附上初值，然后根据不同的条件，再为寄存器附上正确的结果；这样的做法极易导致数据的赋值混乱，使得最终implementation失败。因此，尽可能在同一个if-else语句中完成此类操作。
3. 对于不同always的使用，使用不同的always是区分当前模块不同功能的良好举措，可以使代码的功能分块更加明显。但是，若在不同always中修改同一寄存器，也会引起寄存器数据赋值的混乱现象，导致最终implementation的失败。因而，对同一寄存器的数据存储，还是放置于同一always触发状态下的同一个if-else语句中为妙。