Lab 4 报告

学号 2020K8009929013
姓名史文轩
箱子号 69

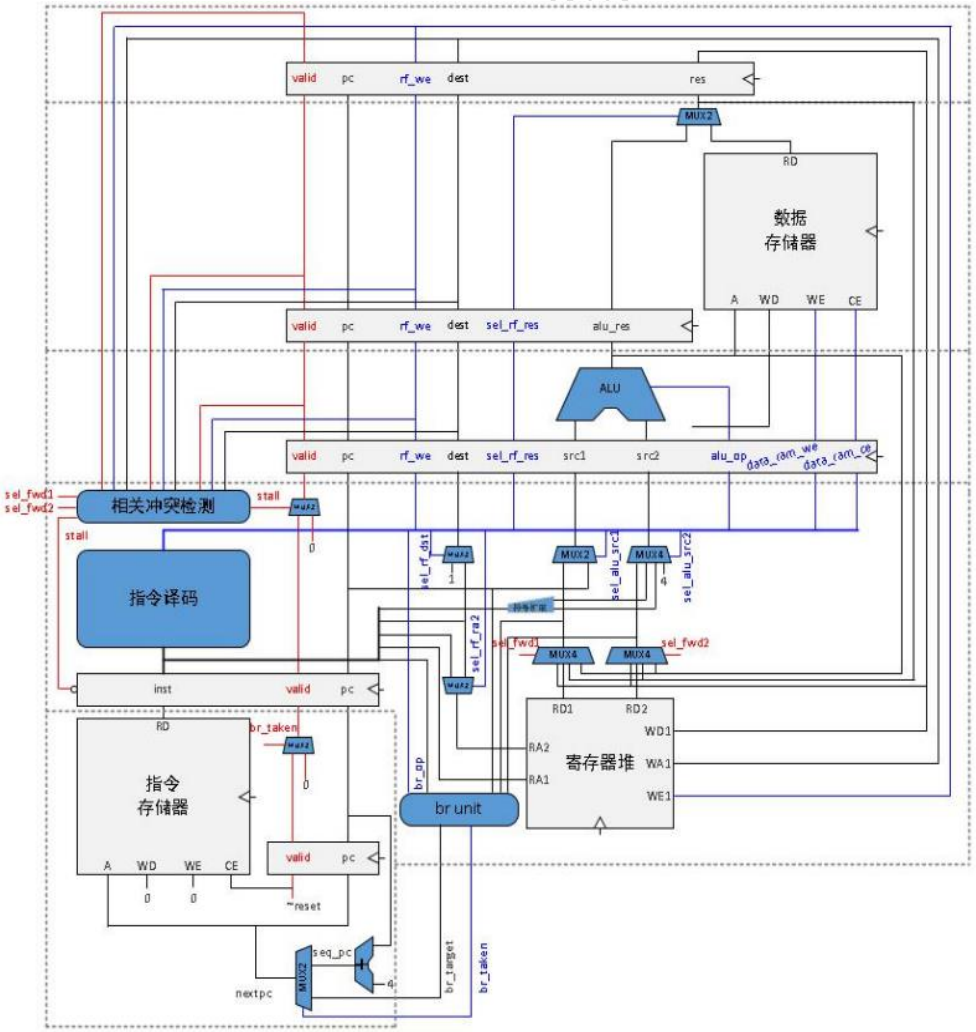
一、实验任务（10%）

本次实验分成了三个阶段进行，第一阶段将原来的 20 条指令单周期 CPU 更改为不考虑相关冲突处理的流水线 CPU（实际上是要考虑指令冲突的）；第二阶段和第三阶段则是分别用阻塞和前递这两种方法解决了由写后读这种数据相关引发的冲突。通过老师提供的有针对性的测试文件进行仿真比对，可以检验各阶段的 CPU 是否得以实现。

二、实验设计（40%）

（一）总体设计思路

流水线的 CPU 是在单周期 CPU 基础上进行实现的，最终实现的结构设计图大概如下图所示：



首先，要实现指令流水，必不可少的就是给每一级流水线加上缓存，流水线的每一级在当前时钟周期内把本级中的指令相关内容处理完后把下级需要的内容存入缓存，等待下个时钟周期到来时被下个流水级取出。当然流水线的设计还要有一些控制信号，以便后续的阻塞和前递技术使用。（在顶层的设计图中并未完全显示这些控制信号，将在下述具体模块设计中展示）

其次，在实现了简单的指令流水之后，还要考虑处理相关，在实现阻塞和前递机制的时候都必不可少冲突检测模块。通过比对译码级要读出的寄存器号和后面流水级将要写回的寄存器号进行是否阻塞或者是如何前递的判断。

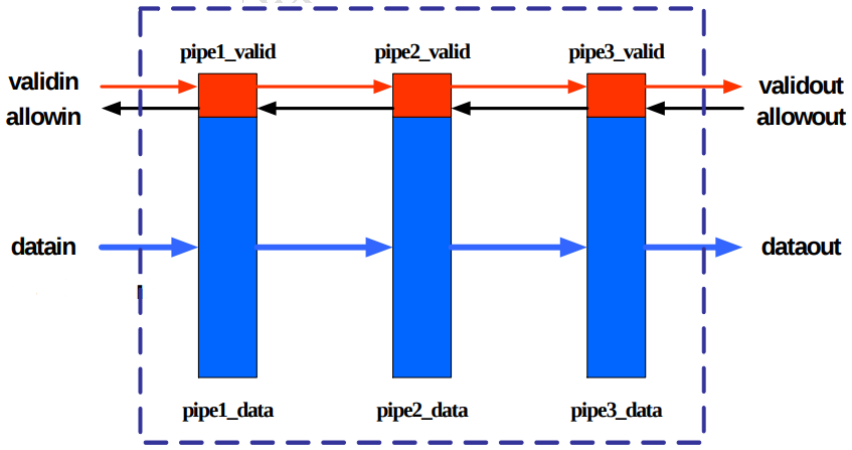
此外，为了引入同步 RAM，还要给流水线加上一个 pre-IF 的虚拟流水级，它没有自己的缓存，但是通过这个阶段，我们可以实现 PC 和指令同步取入到 IF 流水级，如果没有这个阶段的话，取值阶段就至少占用了两拍，大大降低了效率。

最后是前递机制及其选择，为实现前递机制，要把 EXE、MEM、WB 阶段中要进入缓存中的写回结果送到此时的译码阶段，并且要在译码阶段的寄存器读结果处加上两个四选一寄存器，以将正确的寄存器读的结果放入缓存，进入下一级流水。

（二）重要模块 1 设计：流水线缓存

1、工作原理

各级缓存是通过控制信号进行互相连接的，通过各级的 valid 信号，ready_go 信号和 allow_in 信号可以对各级缓存中的存储的数据进行有效的控制。比如解决控制相关时可以用 valid 信号的传递性取消一条错误指令，解决数据相关时可以用 ready_go 信号进行流水线阻塞等。大概的流水线缓存结构如下：



2、接口定义

名称	方向	位宽	功能描述
Validin	IN	1	与复位信号绑定，控制整个流水线进入
Validout	OUT	1	控制流水线流出
Allowin	OUT	1	与 validin 握手，共同决定流水线启动
Allowout	IN	1	与 validout 握手，同上
Datain	IN	X	存入缓存的数据
Dataout	OUT	X	流出缓存的数据

3、功能描述

每一级的流水线缓存都基本遵从如下规则设计：当本流水线的要求阻塞时，将 ready_go 信号赋值为 0，正常流水时则赋值为 1；本级流水要求取消时，可将本级的 valid 信号赋值为 0，valid 信号具有传递性，流水到最后一级仍然为 0，则不会进行写入等操作；每一级都有 allow_in 和 to_valid 信号，二者同时受上下级流水线的影响，只有当下一级的 allow_in 和上一级的 to_valid 信号进行了握手，上一级缓存中存入的数据才会进入下一级。

（三）重要模块 2 设计：冲突检测模块

1、工作原理

通过比对译码级要读出的寄存器号和后面流水级将要写回的寄存器号，来判断是否进行阻塞或是如何前递。但是由于不同指令进行寄存器读的位置在指令中不同，比如有的指令读的是 rj 和 rk 寄存器，有的就读 rd 寄存器等。因此，要把这些情况分开考虑。在写阻塞方法时，我按照不同的流水线阶段进行了冲突的判断。而在写前递功能时，由于前递信号存在优先级，因而要按照一、二两个读端口分别进行冲突的判断。

2、接口定义

此模块输入 ID 阶段的读寄存器号和后续流水级的目标写回寄存器号，输出阻塞信号或者前递选择信号。

3、功能描述

当有阻塞需要时，该模块会输出阻塞信号阻塞流水线；当有数据相关时，该模块会进行判断是哪一个寄存器读端口并输出正确的选择信号。

（三）重要模块 3 设计：前递机制设计

1、工作原理

为实现前递机制，要把 EXE、MEM、WB 阶段中要进入缓存中的写回结果送到此时的译码阶段，并且要在译码阶段的寄存器读结果处加上两个四选一寄存器，以将正确的寄存器读的结果放入缓存，进入下一级流水。

2、接口定义

前递机制输入了后三个流水级的写回结果，通过四选一选择后在译码阶段输出了正确的读出结果。

3、功能描述

前递机制通过把后续的流水级中的写回结果直接前递到译码级，确保了在译码级一个时钟周期内必定能拿到寄存器读应该拿到的正确结果（ld.w 指令除外），大大提升了流水线的效率。

三、实验过程（50%）

（一）实验流水账

9.7 号，仔细阅读了讲义的第五章中改造单周期 CPU 为流水线 CPU 的部分。

9.8 号，在原有单周期 CPU 的基础上加上了缓存和缓存相关的控制信号。

9.9 号，进行仿真调试和 debug 工作，跑通。

9.10 号，同组组员把实验箱给我，上板通过。

9.14 号，仔细阅读了讲义的第五章中用阻塞解决数据相关的部分。

9.15 号，增加了判断阻塞是否进行的冲突判断机制。

9.16 号，debug 过程中解决了阻塞过程停止取值和取消指令不阻塞的细节。

9.17 号，debug 完成，上板通过。

9.19 号，增加了前递机制。

9.20 号，debug 过程中解决了一个很隐蔽的 bug。

9.21 号，上板通过。

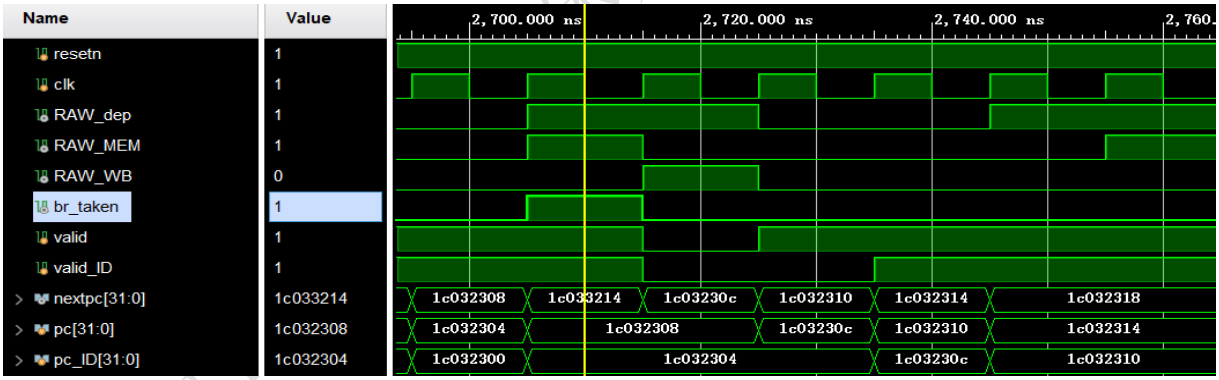
9.24 号，写实验报告。

（二）错误记录

1、错误 1： br_taken 信号的无效拉高导致的错误跳转

（1）错误现象

在某个时刻发现一条指令的 PC 值与 ref 提供的参考值不符，在仔细查看波形后发现从出错的 PC 从取值级就开始错误延后了一拍。波形如下：（1c03230c 这条指令整体延后了一拍）



（2）分析定位过程

首先我查看了相关 PC 对应的汇编代码如下：

```
36550   1c032304:   5c0f1184    bne $r12,$r4,3856(0xf10) # 1c033214 <inst_err
36551   1c032308:   14000004    lu12i.w $r4,0
36552   1c03230c:   1400004c    lu12i.w $r12,2(0x2)
```

可以看到从 1c032304 这条条件跳转指令开始，nextpc 接连出现 1c032308、1c033214（跳转目标地址）、1c03230c，这显然是不合理的。如果条件成立发生跳转，那么 1c033214 取出后不应该再跳回到 1c03230c；如果条件不成立不发生跳转，那么 nextpc 不应该取出跳转目标地址。

在仔细分析寄存器中存储的值之后，我发现这条指令是不应该跳转的，也就是说 nextpc 不应该取出跳转目标地址 1c033214. 这次多余的取出也就导致后续指令都延后了一拍，与仿真波形写回级的错误正好符合。

那么 nextpc 为什么会出现跳转目标地址呢？从波形中可以看到此时的 br_taken 信号是拉高的，这就是错误的根源。bne 指令判断是否进行跳转是要比较寄存器的值是否相等，牵扯到读寄存器，进而可能牵扯到数据相关。在观察阻塞信号后，我发现正是当跳转指令处于 ID 阶段时，有效数据还没有写回，就进行了是否跳转的判断，导致了 br_taken 信号的误判。

（3）错误原因

跳转指令处于 ID 阶段时，在阻塞期间没有拿到正确的寄存器读的值就进行了跳转信号的判断，导致 next_pc 取出无效的跳转目标地址，导致出错。

（4）修正效果

修改了 br_taken 信号的判断逻辑，使得阻塞期间拿不到真值的时候 br_taken 信号不拉高。

```
assign br_taken = (   inst_beq  &&  rj_eq_rd
                   || inst_bne  && !rj_eq_rd
                   || inst_jirl
                   || inst_bl
                   || inst_b
                  ) && valid_ID;
```

改为：

```
assign br_taken = (   inst_beq  &&  rj_eq_rd
                   || inst_bne  && !rj_eq_rd
                   || inst_jirl
                   || inst_bl
                   || inst_b
                  ) && valid_ID && IDreg_ready_go;
```

仿真发现波形正确。

（5）归纳总结（可选）

此错误就是典型的最开始设计的时候考虑不够周全导致仿真的时候必然出现的逻辑性 bug，设计的时候尽量想得周到一点，这类错误就可以少一点。

2、错误 2：被取消的指令依然进行了阻塞

（1）错误现象

在某个时刻发现一条指令的寄存器写回值与 ref 提供的参考值不符，在仔细查看波形后发现从出错的 PC 在与取出的指令不匹配。波形如下：

（2）分析定位过程

我们从波形上就可以看出相关指令的关系大概是从 1c0279f4 跳转到 1c027a0c，中间的 1c0279f8 对应的指令是错误取出的指令，按照处理控制相关的方法，这条指令应该被取消掉了。但是这条指令处于 ID 阶段时与处于 EXE 阶段的另一条指令发生了数据相关，导致产生了阻塞（写后读处理机制）。由于在设计阻塞时，有阻塞是不允许更新 PC 和指令的要求，因而此时的指令读使能信号 inst_sram_en 信号为 0。但是这就导致了此时的处于 pre_IF 阶段的 1c027a10 没有取出指令，由于被取消的指令 valid 信号都为 0，所以是不会跟正常指令一样等数据写回后还有一拍给处于 pre_IF 级的 PC 可以用于正常取指。因而 1c027a10 的指令就会慢一拍取出，从而导致指令和 PC 值错位。错误由此产生。

（3）错误原因

被取消的指令依然进行了阻塞，导致处于 pre_IF 级的指令在取指级没能正常拿到指令。

（4）修正效果

修改了 ID_ready_go 信号的赋值逻辑，使得被取消的指令不会被阻塞。

仿真发现波形正确。

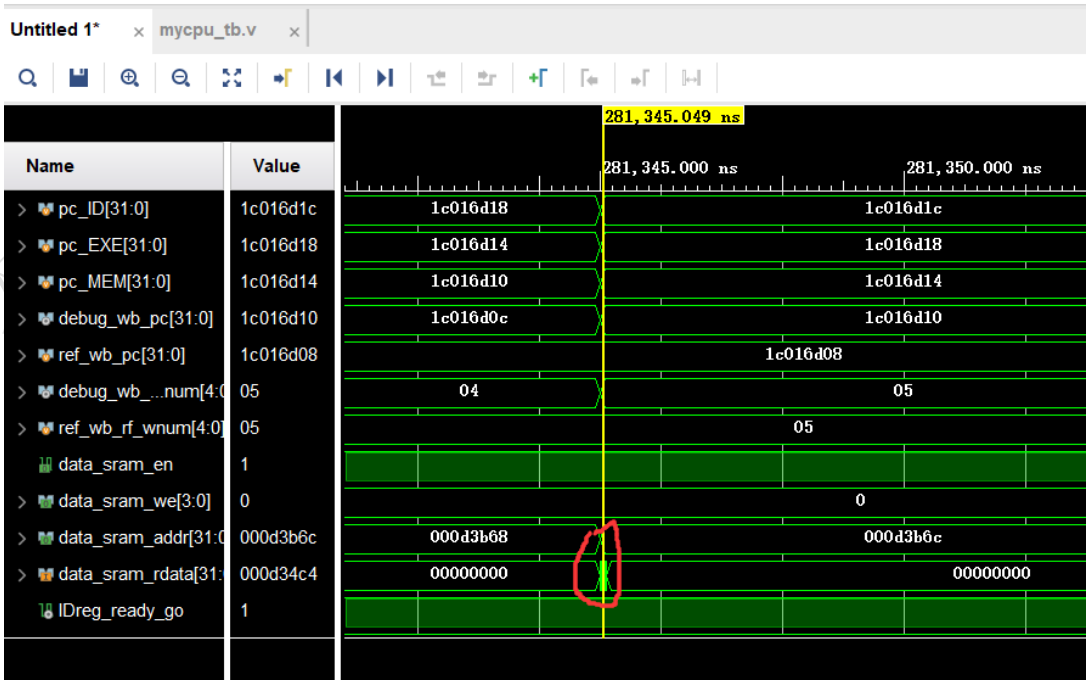
（5）归纳总结（可选）

此错误就是典型的最开始设计的时候考虑不够周全导致仿真的时候必然出现的逻辑性 bug，设计的时候尽量想得周到一点，这类错误就可以少一点。

3、错误 3：错判 st.w 的前递指令类型导致的连锁 bug

（1）错误现象

在某个时刻发现一条指令的寄存器写回值与 ref 提供的参考值不符，在仔细查看波形后发现是出错的 PC（对应一条 ld 指令）在 MEM 阶段取出的内存数据是错误的。波形如下：



本来应该读出的数据（这个数据本身也是错误的）在闪了极短时间后回复为 0。

（2）分析定位过程

Ld 指令读出的数据出错，很自然地想到会不会是之前的 st 指令写入错误呢？抱着这样的信念，我查找了对应的汇编指令，相关的指令如下：

```
1c016cf4:   0293018c    addi.w  $r12,$r12,1216(0x4c0)
1c016cf8:   1590458b    lu12i.w $r11,-228820(0xc822c)
1c016cfc:   029fa16b    addi.w  $r11,$r11,2024(0x7e8)
1c016d00:   299aa18d    st.w    $r13,$r12,1704(0x6a8)
1c016d04:   02801184    addi.w  $r4,$r12,4(0x4)
1c016d08:   02bfe185    addi.w  $r5,$r12,-8(0xff8)
1c016d0c:   299aa084    st.w    $r4,$r4,1704(0x6a8)
1c016d10:   299aa0a5    st.w    $r5,$r5,1704(0x6a8)
1c016d14:   289aa18a    ld.w    $r10,$r12,1704(0x6a8)
```

我们看到读出错误数据的 1c016d14 对应的 ld.w 指令寻址地址为($r12+0x6a8),往上翻找，我们发现在没有其他指令更改过$r12 值的情况下，1c016d00 对应的 st.w 指令在相同的地址存入了数据。理所当然地，ld.w 读出的值应该就是 st.w 载入的值。但是，当我查看了波形后发现两者的值是不同的，错误显然就是因此而生。我们继续往上看寻找与这条 st.w 指令相关的其他指令的时候发现了 1c016cf4 对应的 addi.w 指令与 st.w 指令存在数据相关，即$r12 的值。我发现 st.w 指令读取的$r12 的值与 addi.w 写回的$r12 的值不同，在检查了此处 st.w 与 addi.w 低龄数据相关的处理后，我发现 st.w 没有判断为前递请求，原因是我在写前递请求逻辑的时候错判了 st.w 读寄存器号的类型（把 rj 搞错成了 rd）。

（3）错误原因

在写前递请求逻辑的时候错判了 st.w 读寄存器号的类型，把 rj 搞错成了 rd，导致 st.w 指令的前递没能正确实现，进而影响到后面的 ld.w 指令。

（4）修正效果

修正了判断前递时 st.w 读寄存器号的类型，使得 st.w 能够正确接受前递。

仿真发现波形正确。

（5）归纳总结（可选）

此错误是典型的手下笔误造成的重大“事故”，本来并不是什么原理和逻辑上的严重错误，但是导致的后果更有甚之，debug 的时候很难看得出来，所以以后一定要更加仔细，少出这种低级失误。

四、实验总结（可选）

本次实验实现了简单的流水线 CPU，大大提升了 CPU 的速度。在这个过程中，整体并不算太难，但是遇到了大大小小各种各样的问题。在错误记录中记录了典型的几个错误，尤其是第三个错误，给我带来了极大的麻烦，刚

开始怎么也想不通怎么会从内存中读出来这么奇怪的数据，死磕了有半天才在汇编文件中找出问题所在，最后发现竟然是这么低级的写代码时犯的这么低级的失误，属于是“粗枝大叶一时爽，debug 时火葬场”的典型代表了。总的来说，还是有不小收获的，对流水线有了更深入的理解，也提高了自己的 debug 能力。