P6设计文档

22373321 张瀚文

- [x] 看往年题
- [x] 复习上周的总结
- [x] 考前自己动手添加两个指令

设计草稿

主体采用P5的代码框架，故不再赘述相同部分。

DM

P6要求DM外置。与DM相关的指令有两种，load型和save型，故新建`DMRead`和`DMWrite`模块，分别用来处理从DM中读到的要写入寄存器的数据和从寄存器中读到的要写入DM的数据。其中具体的实现方法均遵循了教程中给出的提示。

IM

IM同样在P6中要求外置，相比DM，IM只负责根据PC输出对应的机器码，故而没有复杂的设计，直接在顶层增添新的输入输出即可。

MultDiv

P6最大的特点是新增了乘除模块。具体实现如下：

- start信号，在D级controller生成，通过寄存器流水到E级以启动乘除模块。
- busy信号，在乘除模块内部生成，通过设置计数变量count来记录延迟，busy信号的确定方法为：

  `assign busy = (count != 0) ? 1 : 0;`
- 输入`ALU_op`以确定要进行的操作和对应的输出。

由于增添了乘除模块，E级现在有两个需要进行流水的变量，为了确定究竟使用哪一个结果，在`controller`模块中增添信号`useMultDiv`以确定此时处于某级的指令是否是跟乘除有关的指令。

- 在阻塞中，根据教程提供的阻塞逻辑，添加一种阻塞类型`stallMultDiv`，其确定方法为：

  `assign stallMultDiv = useMultDiv_D && (busy|start_E);`
- 关于Tuse和Tnew部分，由于乘除类指令均为R型指令，且没有用到的寄存器在机器码中均显示的是0号寄存器，因此无需特殊考虑。

测试方案

1. 测试写寄存器类指令相互冲突

```
ori $1,$zero,0x1877
ori $2,$zero,0x67ed
ori $3,$zero,0x0089
ori $4,$zero,0x0032
ori $5,$zero,0x0e78
add $1,$3,$4
add $3,$1,$3
```

```
sub $5,$3,$4
and $2,$5,$1
and $2,$2,$zero
andi $4,$2,0x0004
sh $4,0($4)
lb $5,0($4)
add $4,$5,$5
sh $4,0($5)
lh $2,0($5)
sub $3,$2,$1
sltu $5,$3,$2
slt $3,$5,$2
```

2. 测试乘除相关指令和其他指令的冲突

```
ori $1,$zero,0x0002
ori $2,$zero,0x1816
ori $3,$zero,0x0004
sw $2,0($3)
lw $4,0($3)
mult $4,$1
mflo $4
jr $4
add $4,$1,$2
add $1,$2,$4
sub $5,$2,$3
add $2,$5,$3#jump here
mfhi $5
add $2,$5,$3
mflo $4
addi $3,$4,0x0001
```

3. 测试写寄存器类指令和跳转指令的冲突

```
ori $5,$zero,0x0004
ori $4,$zero,0x0004
ori $6,$zero,0x0008
sw $6,0($6)
sw $5,0($4)
lw $6,0($5) # 6 -> 0004
lh $7,0($6) # 7 -> 0004
lb $8,4($7) # 8 -> 0008
bne $zero,$8,bne_1
nop
addi $1,$zero,0x0001
bne_1:
addi $2,$zero,0x0001
addi $2,$zero,0x0001
addi $2,$zero,0x0001
addi $2,$zero,0x0001
addi $2,$zero,0x0001
addi $2,$zero,0x0001
```

思考题

1.为什么需要有单独的乘除法部件而不是整合进 ALU？为何需要有独立的 HI、LO 寄存器?

- 乘除法获得的结果，包括64位的乘积或者分别32位的商和余数都需要两个32位寄存器才能保存。
- 乘除法需要多个周期才能计算得出结果，跟alu分离后，可以实现指令的并行，而不是遇到乘除法之后全部阻塞，可以提高cpu的工作效率。

2.真实的流水线 CPU 是如何使用实现乘除法的？请查阅相关资料进行简单说明。

- 乘法

  首先CPU会初始化三个通用寄存器用来存放被乘数，乘数，部分积的二进制数，部分积寄存器初始化为0，然后在判断乘数寄存器的低位是低电平还是高电平（0/1）：如果为0则将乘数寄存器右移一位，同时将部分积寄存器也右移一位，在位移时遵循计算机位移规则，乘数寄存器低位溢出的一位丢弃，部分积寄存器低位溢出的一位填充到乘数寄存器的高位，同时部分积寄存器高位补0，如果为1则将部分积寄存器加上被乘数寄存器，再进行移位操作。 当所有乘数位处理完成后部分积寄存器做高位乘数寄存器做低位就是最终乘法结果。

- 除法

  首先CPU会初始化三个寄存器,用来存放被除数，除数，部分商。余数(被除数与除数比较的结果)放到被除数的有效高位上。CPU做除法时和做乘法时是相反的，乘法是右移，除法是左移，乘法做的是加法，除法做的是减法。 首先CPU会把被除数bit位与除数bit位对齐，然后在让对齐的被除数与除数比较(双符号位判断)。

  双符号位判断： 比如01-10=11(前面的1是符号位) 1-2=-1 计算机通过符号位和后一位的bit位来判断大于和小于，那么01-10=11 就说明01小于10，如果得数为01就代表大于，如果得数为00代表等于。

  如果得数大于或等于则将比较的结果放到被除数的有效高位上然后在商寄存器上商：1 并向后多看一位 (上商就是将商的最低位左移1位腾出商寄存器最低位上新的商) 如果得数小于则上商：0 并向后多看一位 然后循环做以上操作当所有的被除数都处理完后，商做结果被除数里面的值就是余数。

3. 请结合自己的实现分析，你是如何处理 Busy 信号带来的周期阻塞的

- 首先需要`controller`生成`useMultDiv`信号来确定是否D级是跟乘除法有关的指令
- 然后根据busy和E级的start信号判断E级的乘除模块是否正在运行
- 最后确定阻塞信号为

  ```
  assign stallMultDiv = useMultDiv_D && (busy|start_E);
  ```

4.请问采用字节使能信号的方式处理写指令有什么好处？（提示：从清晰性、统一性等角度考虑）

就个人实现方法而言，相比较我原来的写法，使用字节使能信号的方式处理写指令不用已知该内存地址的原有数据，只需将要写的数据调整到对应位置，根据该写使能信号各位是否为1进行写入，非常清晰；同时还便于扩展其他指令。

5.我们在按字节读和按字节写时，实际从 DM 获得的数据和向 DM 写入的数据是否是一字节？在什么情况下我们按字节读和按字节写的效率会高于按字读和按字写呢?

- 不是，按字节读是会先确定要读的究竟是哪一字节，然后对该字节进行符号或者0扩展；按字节写从某种意义上来说确实只写入了一个字节。
- 当sb，lb，sh，lh此类指令较多时，按字节读和按字节写的效率会更高。

6.为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助?

- 对指令进行归类

  `add,sub,and`等指令可以归为`R_type`；`addi,ori,lui,andi`等指令可以归为`I_type`；`lw,lh,lb`等指令可以归为`load`，`sw,sh,sb`等可以归为`save`等等。这样处理以后在`controller`中输出一些控制信号时会变得非常方便，如`Tuse`和`Tnew`，`RW`等信号的确定。同时这样写也便于指令的扩展，分析新指令的行为，在相应的归类信号中加入该信号，而无需做太多改动。

- 对命名进行规范

  对于变量命名，采取以下规范：

  `变量名_运算数_级`，命名规范在`HazardUnit`中显得尤其重要，因为冒险处理单元中牵扯大量的不同阶段的变量

- 增添宏文件进行译码指令的处理，宏文件的作用包括识别指令，定义`alu_op`和`compare_op`等

- 模块低耦合

  对于DM模块，分别设置read和write模块，防止一个模块实现过多功能导致层次不清晰

7.在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的?

1. 写寄存器类指令相互冲突
2. 乘除有关指令与其他指令的冲突
3. 写寄存器类指令和跳转指令冲突

具体的测试样例可见本文档的**测试样例**部分。

8.如果你是手动构造的样例，请说明构造策略，说明你的测试程序如何保证覆盖了所有需要测试的情况；如果你是完全随机生成的测试样例，请思考完全随机的测试程序有何不足之处；如果你在生成测试样例时采用了特殊的策略，比如构造连续数据冒险序列，请你描述一下你使用的策略如何结合了随机性达到强测的效果.

- 在测试写寄存器类指令相互冲突时，首先用ori指令为1-5号寄存器赋初值，接着随机使用已经实现的写寄存器类指令进行测试，为了最大化冲突，第n条指令的运算数总是第n-1条指令要写入的寄存器，其中要注意牵扯到内存是偏移量需为4的倍数。
- 在测试乘除法相关指令和其他指令的冲突时，主要考虑两种情况，首先是乘除法指令后跟mthi/mtlo/mfhi/mflo指令的，需要stall；或者是mfhi/mflo指令涉及到的转发问题。
- 在测试写寄存器类指令和跳转指令的冲突时，主要针对已经实现的jal/jr/beq/bne指令进行测试

做题注意

1. 信号到底有没有加全？加全了今晚也不会去答疑了，再讲一遍，构造样例之前先自己好好审一遍代码，尤其是macro与controller里面的信号，有没有加全？
2. 答疑的收获：实现不同的指令的时候都重开一个，不要跟之前的代码有纠缠，尤其是那种特判特别多的
3. 关于符号：指令里面强调了unsigned，也就是加了一个u这种，就是说的无符号。ISE同样默认无符号。还是得仔细看操作指令。前面补0再比较的都是无符号比较，不用加signed
4. 关于看题：再次再次再次强调一遍：`rs`这种，说的是寄存器地址，`GPR[rs]`这种，说的是寄存器值！！
5. 下载的时候下载middle的，有备份
6. 移位指令的什么地方需要注意？就是对于sll这类不可变移位，是不需要使用rs寄存器的，因此在controller中需要额外控制一下，主要改的是和rs相关的Tuse和useA

来点上机题

1. blezalc，小于等于0时跳转并链接

   添加compare，RW，ra，compare模块修改，每一级的RW修改

2. bezal:如果GPR[rt] = 0,跳转到GPR[rs],并且写31号寄存器

   需要在controller_D里面出来一个check信号进入npc，不用在npc里添加compare信号，因为PCsrc就带了compare的结果，只需要在pcBranch里面再加一层MUX即可。

   除此之外，每一级的RW部分也需要更新成条件写的形式，condition信号注意往下流水。

   op -> branch -> branchTwo -> ra -> compare_op -> RW -> com_op -> com_write -> npc部分 -> 每一级的RW部分

3. 继续lhonez：从内存中读出半字数据，比较该数据中数字1和0的个数的多少，1多则写入rt寄存器，否则将pc+4写入31号寄存器。注意writeReg默认rt

   属于之前总结的第一种类型。

   op -> load -> ra -> compare_op -> com_op -> op_ctrl -> check_2

   -> compare部分 -> hazard部分找到跟第一种有关的，包括转发和暂停 -> 顶层修改writeReg_W和WD_W

4. lwmx 读出的数据值大于rt寄存器的值就把读到的数据写入5号寄存器，否则写入四号寄存器

   注意这一条要把rt的寄存器值传到W级，回溯一下可以发现是writeData_M，需要修改M到W的寄存器流水，把writeData_M传递过来，再连接到compare的第二个操作数上。

   属于第一种类型，修改路径如下：

   op -> load -> compare_op -> com_op -> op_ctrl -> check_2 -> 添加compare部分的比较 -> hazardunit中第一种类型的判断修改，两个可能的寄存器值修改成4和5,不转发照旧 -> 顶层修改writeReg

5. lwld 读出的数据如果满足某个条件就写入31号寄存器，否则写入rt寄存器

   属于第一种类型

   op -> load -> compare_op -> com_op -> op_ctrl -> check_2 -> compare部分增加 -> hazard部分修改第一种类型的寄存器可能情况 -> 顶层修改writeReg

6. movz 当rt寄存器值大于0时，将rs寄存器的值写入rd寄存器的值，否则不写

   条件写，condition计算好之后往下流水确定每一级的RW就行,不用额外考虑阻塞转发,注意区分一下两个op，当成计算类指令就可以

   op -> alu_op -> compare_op -> ALU_op -> com_op -> com_write -> TuseB修改 -> compare部分修改 -> ALU部分修改 -> 每一级RW修改。

7. jap //需要再看一下

   ```
   Addr <- GRF[29]

   mem[addr] <- PC + 8

   GRF[29] <- GRF[29] - 4

   PC <- PC[31:28]||imm26||00
   ```

   完全的新题

   op -> jump1 ->alu_op ->ALU_op -> RW -> useReg内部修改成29号寄存器 ,writeReg内部也修改，width_op和write或起来，Tuse为1 -> 从alu_result获得29号寄存器值 -> 减4获得真正要写入的值 -> 修改每一级的WD准备转发 -> 修改M级的writeData_M为PC+8 -> 修改E到M,M到W的寄存器把29号寄存器-4的值传下去

8. lwer 从内存中读出数据，经过一个表达式计算出寄存器的编号，然后将读出的数据写入该寄存器。

   对照上周总结的几种类型，lwer应当属于第三种类型，即只有到W级才知道要写的寄存器是什么，因此直接阻塞，同时注意不转发。修改路径：

   op -> load -> op_crtl -> check_2 -> hazardunit第三种类型，注意不转发 -> 顶层修改writeReg，WD不用修改

9. 给出rs和rt两个寄存器，较大的数除以较小的数，注意无符号。直接在multdiv里面添加即可，阻塞转发仍然无需额外考虑，在useMultDiv中添加

   op -> alu_op -> multdiv

10. bgezalr 无条件写pc+8到rd，如果rs大于等于0，则跳到rt寄存器中储存的地址。

    op -> branch -> ra -> branchTwo -> compare_op -> RW -> com_op -> check -> npc里面修改pcbranch，注意传入的是rt寄存器，在外部特判一下 -> writeReg内部修改 ,ra除去该特殊信号 ， 之所以加上ra信号是因为这样的话往回转发的数据就比较好弄

11. msub rs与rt相乘，然后用HI和LO拼接起来的结果减掉这个相乘的结果，再把结果的高位存在hi里，低位存在lo里

    坑点在于对符号的处理，需要写成下面的样子

    `{hi, lo} <= $signed({hi, lo}) - $signed($signed(64'd0) + $signed(ALUSrcA) * $signed(ALUSrcB));`

    确保保留了64位的位宽信息和确保了做有符号运算。

12. lhs 取地址的低两位，如果为0把数据的最低字节写入rt，为2就把第3个字节写入寄存器，为1则不写寄存器。

    属于第二类，但是其实是正常的判断，因为是rt寄存器，但是确实不向前转发，最后在W级writeReg处修改即可。

op -> load -> op_ctrl -> check_2 -> hazard部分挑出第二种类型的以及修改转发 -> 特判修改writeReg和WD

13. bds 无符号大数除以小数，只需要在对应的case下面添加if语句判断即可，**注意需要归到乘除法有关指令中去！！**
14. bezal rt等于0则跳转到rs 并写31号寄存器。

    条件写类型。

    op -> branch -> branchTwo -> compare_op -> ra -> RW -> com_op -> com_write -> 添加Tnew（选做，因为Tnew也就是1或者0，到E级都为0，而且只考虑EM级的，所以选做） -> npc内部修改pcbranch，controller里加一个check信号 -> 所有RW需要修改成条件写
15. 今年的lwxx。其行为属于第三种类型，直接阻塞即可，然后注意不转发和修改最后的writeReg
16. crt 把GPR[rs]循环移位得到的32个结果无符号求和（结果记为temp1），把GPR[rt]循环移位得到的32个结果无符号求和（结果记为temp2），比较这两个结果。temp1大写入1，小则写入-1，相等写0
17. btheq rs的最高位1和rt最高位1位置相同则跳转，其中任何一个为0则不跳转。跟beq什么的几乎一模一样

    op -> branch -> branchTwo -> compare_op -> com_op -> 修改compare部分即可
18. 交换hi，lo的寄存器值：

    需要注意的是因为非阻塞赋值的特性，这种交换不需要中间加一个temp，实际上加了还是错的。具体添加的位置就是乘除模块中对应的mtlo，mthi的地方.不用考虑转发和阻塞，因为只是交换两个寄存器里面的值。

    op -> alu_op - > multdiv