P6 - 更完善的流水线CPU - 设计文档

加入对乘除的支持，并添加部分指令。

这一部分工作，主要难点在于`set`类指令，和乘除相关指令的添加。

总结先前编写设计文档的经验，我们只对相较P5做出了什么更改进行说明，如有本文未提及的，请参阅P5设计文档。

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？

ALU内的算数/逻辑操作延迟显著低于乘除法操作的延迟；如整合进ALU，则为了照顾乘除法操作，必须降低CPU整体的时钟频率，代价过大。

有独立的LO，HI寄存器，主要是为了**将乘除法部件与关键路径隔开，保证乘除计算仍在进行时，其它与乘除操作无关的指令能够不暂停，继续进行。** 同时，这也是为何乘除法结果需要存在HI、LO中的原因。

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。

对于乘法，我们可以发现，其是通过与运算得到的。具体实现上，我们需要了解两个东西：**高斯树与保存进位加法器（Carry-Save Adder, CSA）。** 前者进行竖式乘法的累加操作，而保存进位加法器则是高斯树中用到的加法单元，用于逐级减少加数。

对于除法，除法器的实现其实有很多种。以阵列除法器为例，我们通过**恢复余数法**，持续将被除数减去除数，直到不能再减，来进行除法运算；而阵列除法器，则是在模仿计算过程中的竖式除法。

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？

先回顾一下乘除法操作的执行流程：在乘除操作进入E级时，控制器向乘除模块传入一`Start`信号；随后，在乘除法正在进行时，乘除模块数除一`Busy`信号。

对于要读取/操作计算结果，即HI、LO寄存器的指令，我们需要在乘除法操作仍在进行的时候，阻止该指令的读取，即产生暂停。分析不难得知，即`Start`或`Busy`有效时，且当前指令为`lfhi/lo`类指令时，产生暂停。

```
assign _stall_MD_HI = (_CTRL_E_MD_Start | _MD_Busy & ) & (_CTRL_D_read_hi);
```

```
  assign _stall_MD_LO = (_CTRL_E_MD_Start | _MD_Busy) & (_CTRL_D_read_lo);

  assign _stall_MD = _stall_MD_HI | _stall_MD_LO;

// 在原有暂停条件基础上，新引入乘除法有关的暂停条件
  assign _stall = _stall_rd1 | _stall_rd2 | _stall_MD;
```

但是！教程里面提到，为了追求效率，是存在E级传入新的乘除指令，取消当前乘除操作的设计的。这个思路能不能进一步发展下去呢？是可以的！

注意到，我们存在`lthi, ltlo`这么两条**直接写入HI,LO**的指令。从编程者的角度，若其在`mult/div`后执行，会替换掉乘除法的计算结果。

那么，我们就有了个激进的想法：

- 不阻塞`lthi/ltlo`，让其直接对当前HI，LO进行复写，**无论乘除法是否正在进行**
- 对于乘除法，在乘除法结束，将要写入时，**若先前发生了复写操作，则算出的乘除法结果不覆盖之前复写的数据。没发生复写，则正常写入。**
- 对于`lfhi/lo`，如果前一条指令发生了对应HI/LO的复写，则可以**不阻塞，直接读出。**

最后控制信号实现如下，具体请见设计草稿。

```
assign _stall_MD_HI = (_CTRL_E_MD_Start | (_MD_Busy & !_MD_ovrd_hi)) &
(_CTRL_D_read_hi);

assign _stall_MD_LO = (_CTRL_E_MD_Start | (_MD_Busy & !_MD_ovrd_lo)) &
(_CTRL_D_read_lo);

  assign _stall_MD = _stall_MD_HI | _stall_MD_LO;


  assign _stall = _stall_rd1 | _stall_rd2 | _stall_MD;
```

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）

我们都知道，DM里的数据是整字存储的。向内写入的时候，我们必须传入32位数据；对于`sh,sb`，我们**不得不使用这种办法**，来进行对特定地址字节/半字的写入。

既然都这样了，不如`sw`也用这套使能信号吧，这就是统一性的考量。

然而依我愚见，这种做法在清晰性上并不见得有多优秀。在我看来，这应该是**DM内部实际的写使能信号**，将其直接由外部控制器信号指定并不清晰。本人采用以下方案：向DM输入写使能信号，并指定当前DM的操作模式（w,h,b），由这两个信号译码产生字节使能。

5. 请思考，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？

上面第四题已经提到，实际写入不是一字节。实际获得的也不是，获得的是对应那个字的数据，需要我们按地址进行截取。

对于我们当前的实现，这种情况我认为不存在，因为按地址进行的操作，本身就会因为条件判断产生延迟。

这个题干本身就写得有问题，我猜其意思是，DM内部存数据的单元是字节，还是字。

如果是这样的话，那么对于单个ASCII字符的读写会更快，因为其只占一个字节。

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？

高内聚低耦合方面，对于一个模块，我们只对数据的计算操作进行功能划分。以ALU而言，我们按`add,sub,and,sll`等进行划分。至于每个指令，参与计算的值不同的问题，则由ALU外部的MUX进行指定，不在模块内部进行调整。

且对于DM，我们选择将`lb,lh`读出的数据在DM内部直接处理完毕。

在控制器方面，主要就是对指令进行分类，按类进行控制信号的生成。

在控制信号方面，我早在P4就采取了宏定义进行模块操作信号命名的操作。

冲突与冒险方面，直接沿用AT法即可。我们要干的，只是将新加入的指令归入对应的Tuse, Tnew。

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？

   两种。一种是之前P5中，Tuse与Tnew关系产生的，一种是乘除法产生的。

   解决方法这TM用说？AT法的直接沿用之前的转发与暂停逻辑，乘除法的请参见上面思考题的解法。

   AT法的测试样例沿用P5的即可（不过需要加入新加的`and`等指令），乘除法的测试如下：

```
ori $2,$0,0xffffffff
ori $3,$0,0x7fffffff
```

```
        ori $20,$0,0x0000eeee
        ori $30,$0,0x0000dddd

        slt $8,$2,$3
        sltu $9,$2,$3


        mult $2,$3
        mfhi $4
        mflo $5
        multu $2,$3
        mfhi $4
        mflo $5
        div $2,$3
        mfhi $4
        mflo $5
        divu $2,$3
        mfhi $4
        mflo $5

        mult $2,$3
        nop
        nop
        nop
        mthi $20
        mfhi $4
        mflo $5
        multu $2,$3
        mtlo $30
        mfhi $4
        mflo $5
        div $2,$3
        mthi $30
        mfhi $4
        mflo $5
        divu $2,$3
        mtlo $20
        mfhi $4
        mflo $5
```

```
        ori $2,$0,0xffffffff
        ori $3,$0,0x7fffffff
        div $2,$3
        mthi $2
        mtlo $3
        mfhi $2
        mflo $3

        ori $2,$0,0xffffffff
        ori $3,$0,0x7fffffff
        mthi $2
        mtlo $3
        div $2,$3
        mfhi $2
        mflo $3

        ori $2,$0,0xffffffff
        ori $3,$0,0x7fffffff
        div $2,$3
        mfhi $2
        mflo $3

        ori $2,$0,0xffffffff
        ori $3,$0,0x7fffffff
        divu $2,$3
        mfhi $2
        mflo $3

        ori $2,$0,0xffffffff
        ori $3,$0,0x7fffffff
        multu $2,$3
        mfhi $2
        mflo $3
```

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。

测试主要分两类：功能测试与冒险。

在功能测试部分，我们利用`ori`指令，按照待测功能/待测情况给待测指令的输入赋特定值。随后，接上要测试的指令，检查指令行为是否正常。

这里的测试实际上仍然存在冒险，出于构造方便的考量，我们不插入nop进行规避。

这部分测试正常后，我们进入冒险的测试。对这一部分，我们分由AT法引起的冒险，与由乘除引起的冒险两种情况进行构造。

- AT法：由Tuse和Tnew的不同，对指令进行排列组合。注意这里需要排除掉乘除法有关操作。
- 乘除法操作：我们可以将乘除相关操作分为算、读寄存器、写寄存器三类。对三类指令进行排列组合，进行测试。

随后，我们对AT法，乘除两种综合引起的情况进行构造。这里可由前AT，后乘除，或先乘除，后AT两种情况进行构造。

完全随机的测试程序会产生死循环，无法高效测试边界情况，覆盖率不稳定的问题。

结合随机性的话，可以从读写的目标寄存器，和每类指令中所取的指令，以及指令的执行顺序进行入手。

设计草稿

相较P5的更改

- 添加了MD模块，支持乘除相关操作。
- 完善ALU，增加了按位与等操作，以支持新加入的`add`，`addi`等指令。
- 在D级添加了`slt`，`sltu`；不过**这一设计是否合适有待商榷，因为大部分运算指令，都会导致该设计下的set类指令暂停，加速仅对于条件跳转等Tuse = 0的指令有效。**
- 补齐了P5中未设计的E级至D级转发。

转发要注意的问题

对于转发，我们需要明确：**只有当前产生了新值**，且没写入GRF时，我们才需要进行转发！

也就是说，对于每一级的转发，我们的使能信号应该**只对当前与之前层级中产生新值的指令有效**，不能单纯靠指令要对GRF写入进行转发条件的判断！

最后本人的转发使能写法如下：

```
assign forward_e = _set | _link;

assign forward_m = (_calc & !_memory) | _link | _set | _md_read;
```

```
assign forward_w = (_calc & !_memory_write) | _link | _set | _md_read;
```

乘除模块 （MD）

接口名称	方向	说明
start	I	指示模块开始运算
busy	O	指示模块当前正在运算
MDOp	I	指示乘除操作
MDSigned	I	指示是否有符号运算
MDOvrd	I	mthi/lo用，指示是否为lthi/lo
MDOvrdDest	I	指示load to 的对象为HI还是LO
D1	I	乘数/被除数
D2	I	第二乘数/除数
LO	O	LO当前的值
HI	O	HI当前的值
ovrd_hi	O	指示HI先前是否发生了load to类的复写
ovrd_lo	O	指示LO先前是否发生了load to类的复写

乘除槽

这部分还是值得提出来说一说的。

为了更高的性能，我们不仅采用了教程中提到的顶替未完成乘除操作的设计，**还将类似思路运用到了`lthi`，`ltlo`两条指令中。**

在我们开始之前，我们不妨先讨论这么做的合理性。我们都知道，乘除法的结果，在算完后，都是存在HI，LO两个寄存器中的。**除非执行`lfhi/lo`，HI/LO实际上都是没有用到的！也就是说，我们这个时候可以放心大胆地顶替未完成的计算，最后结果仍然是一致的！**

那对于`lthi/lo`，我们怎么运用这个思路呢？在`lthi/lo`到E级执行时，我们直接将其指定的值写入HI/LO中，**且不进行暂停**；假设此时乘除单元仍在进行运算，我们在写入结果的时候，只对lthi/lo没写入的寄存器进行写入。

为了实现这一功能，我们需要一个Ovrd的输入信号，表明此时正在进行lthi/lo的复写操作；同时需要在乘除模块内加一个状态寄存器，指示当前的HI/LO是否先前被lthi/lo写入过。执行lthi/lo后，状态寄存器对应为1；在OVRD信号为1，或状态寄存器为1时，当前计算完的乘除结果不写入对应寄存器。

在进行新一轮运算的时候，状态寄存器复位回0。

同时，为了使复写后一条的lfhi/lo可以不暂停进行，我们引出两个控制信号`ovrd_hi`与`ovrd_lo`，指示先前是否发生了复写。若发生了，则对应的lfhi/lo不发生暂停，直接读取。

SLT/SLTU

由于我们为了`beq`等跳转指令，将比较模块前移到了D级，孩子看CMP在D级，就把set类指令放到D类来执行了。

顺便也借此机会，补齐了P5里没写的，E级寄存器到D级的转发。

再次，自暂停的角度，这么设计是否合理有待商榷。因为对于大部分计算指令，Tnew = 1 ，如此设计会导致暂停，加速效果仅对Tuse = 0 的指令有效。

测试方案

沿用P5的测试数据，并针对P6新加入的指令进行了测试，尤其是set类与乘除相关指令。

P5:

```
.text
# 操控$5为0xffff_ffff,$6为0x8000_0000
# 测试$0
ori $0,$0,65535
# 若通过，测试ori；zero_ext
ori $3,$0,4096 # 高第三位为1，测试$0是否不影响转发
ori $3,$3,65534 #最后一位0，高第三位为1，测试是否转发
# 若通过，测试lui
lui $3,65535
lui $0,65535
# 若通过，测试减
ori $4,$0,5 #init
ori $2,$0,1
sub $3,$4,$2 #正常减法
ori $2,$0,6
sub $3,$3,$2 #跨到-1
ori $2,$0,1
sub $3,$6,$2 # -2137483648 - 1，溢出到2137483647
sub $0,$3,$1 # $0
# 若通过，测试加
ori $4,$0,5 #init
ori $2,$0,1
add $3,$4,$2 #正常+法
ori $2,$0,1
```

```
sub $3,$0,$2 #$3跨到-1
add $3,$3,$2 #-1跨到0
ori $2,$0,1
add $3,$5,$2 # 2137483647 + 1, 溢出到-2137483648
add $0,$3,$2 # $0
# sll
ori $3,$0,65535
sll $3,$3,16
sll $0,$3,1
sll $0,$0,0 #nop, 无事发生
# 通过, 测试sw
ori $3,$0,1
ori $2,$0,4
sw $3,0($2)
sw $3,4($2)
sw $3,-4($2)
# lw
# 完成上述测试后, 0, 4, 8应存有1
ori $4,$0,1
ori $2,$0,4
lw $3,4($2)
lw $3,0($2)
lw $3,-4($3) # 暂停
lw $0,0($2) # $0
# beq
ori $2,1
ori $3,2
ori $4,1
that:
beq $2,$3,that
nop

ori $2,$0,16
ori $3,$0,16
ori $7,$0,0
ori $8,$0,4
sw $2,0($7)
sw $3,0($8)
ori $2,$0,65535
ori $3,$0,65534
lw $2,0($7)
lw $3,0($8)
beq $2,$3,next # 暂停两个周期
ori $2,$0,16
ori $3,$0,65535
next:
```

```
ori $3,$0,16
ori $2,$0,255
ori $3,$0,255
ori $4,$0,126
ori $5,$0,127
ori $4,$0,127
beq $4,$5,next1 # 暂停一个周期
ori $4,$0,0
ori $4,$0,511
ori $5,$0,511
next1:
ori $2,$0,511
ori $3,$0,511

jal test # 测试跳转，以及下方延迟槽，以及jal有关转发是否正确
ori $31,$31,0
ori $3,$0,65535
test:
ori $3,$31,0 # 继续测转发
ori $10,$0,65535
ori $21,$10,65535
```

P6:

```
sltu $9,$2,$3
slt $8,$2,$3


ori $2,$0,1
sltu $9,$2,$3
slt $8,$2,$3


ori $2,$0,-1
sltu $9,$2,$3
slt $8,$2,$3



ori $2,$0,0xffffffff
ori $3,$0,0x7fffffff
ori $20,$0,0x0000eeee
ori $30,$0,0x0000dddd

slt $8,$2,$3
```

```
sltu $9,$2,$3


mult $2,$3
mfhi $4
mflo $5
multu $2,$3
mfhi $4
mflo $5
div $2,$3
mfhi $4
mflo $5
divu $2,$3
mfhi $4
mflo $5

mult $2,$3
nop
nop
nop
mthi $20
mfhi $4
mflo $5
multu $2,$3
mtlo $30
mfhi $4
mflo $5
div $2,$3
mthi $30
mfhi $4
mflo $5
divu $2,$3
mtlo $20
mfhi $4
mflo $5

ori $2,$0,0xffffffff
ori $3,$0,0x7fffffff
div $2,$3
mthi $2
mtlo $3
mfhi $2
mflo $3
slt $4,$2,$3
sltu $4,$2,$3
```

```
ori $2,$0,0xffffffff
ori $3,$0,0x7fffffff
mthi $2
mtlo $3
div $2,$3
mfhi $2
mflo $3
slt $4,$2,$3
sltu $4,$2,$3

ori $2,$0,0xffffffff
ori $3,$0,0x7fffffff
div $2,$3
mfhi $2
mflo $3
slt $4,$2,$3
sltu $4,$2,$3

ori $2,$0,0xffffffff
ori $3,$0,0x7fffffff
divu $2,$3
mfhi $2
mflo $3
slt $4,$2,$3
sltu $4,$2,$3

ori $2,$0,0xffffffff
ori $3,$0,0x7fffffff
multu $2,$3
mfhi $2
mflo $3
slt $4,$2,$3
sltu $4,$2,$3

ori $3,$0,0x12345678
ori $2,$0,0x12345678
sw $2,0($0)
lw $3,0($0)
lh $3,0($0)
lb $3,0($0)
lh $3,2($0)
lb $3,1($0)
```