P5 设计文档

22373321 张瀚文

怠惰必将导致惨剧。

相较于P4代码的优化

- 使用注释作为代码段落的分隔符，更为清晰
- 每个阶段都使用一个controller，避免信号在不同的阶段来回传播，同时考虑到每个阶段所需要的信号不一样，对controller模块可以选择性输出。

整体设计思路

- 本次P5需要实现的指令有{add,sub,ori,lw,sw,beq,lui,jal,jr,nop}
- 整个流水线被分为五个阶段：Fetch|Decoder|Execute|Memory|Writeback,分别被称为：F级、D级、E级、M级、W级
- 要尽最大可能支持转发以解决数据冒险
- 每个阶段分别译码，输出Tues等控制信号
- 流水线寄存器的作用：保存在前一周期中的上一阶段所传来的数据，同时在后一周期中为下一阶段提供数据。
- 需要流水的数据只有一个衡量标准：就是在其后的流水阶段中需不需要这个数据，每个流水线寄存器都保存着一条指令完成后续操作所需要的全部信息。注意**流水数据在CPU中可能存在多个值，且这些值并不相同，在编程的时候需要使用流水阶段名前缀**将其区分开。
- 对Tnew的处理：每个controller只负责翻译D级的Tnew，后续会根据所在阶段的不同减去相对应的数，得到真正的Tnew
- **在通过转发更新寄存器的值时，存入流水线寄存器的值也应当是转发后的值**

  下附表格代表不同信号的含义：

信号名	信号种类	信号含义/作用
ForwardA_D	选择信号	WD_E(2'b11) | WD(2'b10) | WD_M(2'b01) | RD1_D(2'b00)
ForwardB_D	选择信号	WD_E(2'b11) | WD(2'b10) | WD_M(2'b01) | RD2_D(2'b00)
ForwardA_E	选择信号	WD_M(2'b01) | WD(2'b10) | RD1_E
ForwardB_E	选择信号	WD_M(2'b01) | WD(2'b10) | RD2_E
ForwardB_M	选择信号	WD（1） | writeData_M_ori
stall	阻塞信号	控制阻塞（冻结PC，冻结D级寄存器，E级寄存器清零）

信号名	信号种类	信号含义/作用
useA_D	输入	D级的A操作数是否要被用到
useB_D	输入	D级的B操作数是否要被用到
useReg_A_D	输入	D级的A寄存器
useReg_B_D	输入	D级的B寄存器
Tnew_E	输入	E阶段指令更新的周期数
Tnew_M	输入	M阶段更新的周期数
RW_M	使能	M级指令是否写寄存器
RW_W	使能	W级指令是否写寄存器
RW_E	使能	E级指令是否写寄存器
useReg_A_M	输入	M级指令使用的A寄存器
useReg_B_M	输入	M级指令使用的B寄存器
writeReg_M	输入	M级指令写的寄存器
writeReg_E	输入	E级指令要写的寄存器
Tuse_A_D	输入	D级指令的A操作数需要用的周期数
Tuse_B_D	输入	D级指令的B操作数需要用的周期数
useReg_A_E	输入	E级指令用到的A操作数所在的寄存器
useReg_B_E	输入	E级指令用到的B操作数所在的寄存器
ForwardB_M	选择信号	WD(1)|writeData_M_ori（0）,也即转发过来的第二个寄存器的值
PC_D/E/M/W	PC值	各级的PC值，对应各级指令的地址
com_srcA	32位寄存器值	相当于转发后的第一个操作数的值（npc在返回寄存器中的地址时就用的这个）
com_srcB	32位寄存器值	转发后的第二个操作数的值
writeReg_W	寄存器地址	W级指令要写的寄存器的地址，其已经根据W级controller进行过判断和选择
Tnew_ori_E	选择信号	E级指令翻译出的原始Tnew值

信号名	信号种类	信号含义/作用
Tnew_E	选择信号	根据Tnew_ori_E减一的结果决定最终E级信号的Tnew_E值
ALU_srcA(/B)_E	选择信号	为1，选择shamt/立即数，否则选转发后的RD1,RD2
ALU_result_M	32位数据信号	E级中计算出的ALU通过寄存器到达M级
srcA_fore	32位数据信号	根据ForwardA_E信号的不同选取不同的转发值
srcB_fore	32位数据信号	根据ForwardB_E信号的不同选取不同的转发值
writeData_M_ori	32位数据信号	从E级接受而来的第二个寄存器通过**转发后的**信号，也即srcB_fore
compare_condition_M	比较信号	根据寄存器流下来的比较信号
D2R_E	选择信号	E级信号选择要写入的寄存器控制信号
WD_E	数据信号	E级要往回转发的值
WD_M	数据信号	M级需要往回转发的值（PC_M+8/ALU_result_M)
writeData_M	数据信号	最后真正要往DM中写入的数据，是从W级往回转发的和前面经过转发的传过来的寄存器的值再选择一次的结果（Forward_B_M)

冲突的解决

阻塞

在本CPU中，冲突指令会被阻塞在D级上。

在D级阻塞的时候，向下一流水级传递的指令不应当是D级指令，应当插入nop空指令来避免这种情况。

阻塞具体的实现

当指令到达D级后,将其Tuse值和后面每一级的Tnew进行比较，当Tuse<Tnew时（且A值检验通过时）需要阻塞流水线。

阻塞具体表现在：

- 冻结PC的值
- 冻结D级流水线的值
- 将E级流水线的寄存器清0（等价于插入一个nop指令）

预测

在D级提前对branch类指令的条件进行判断，并据此更新PC值。

增加延迟槽

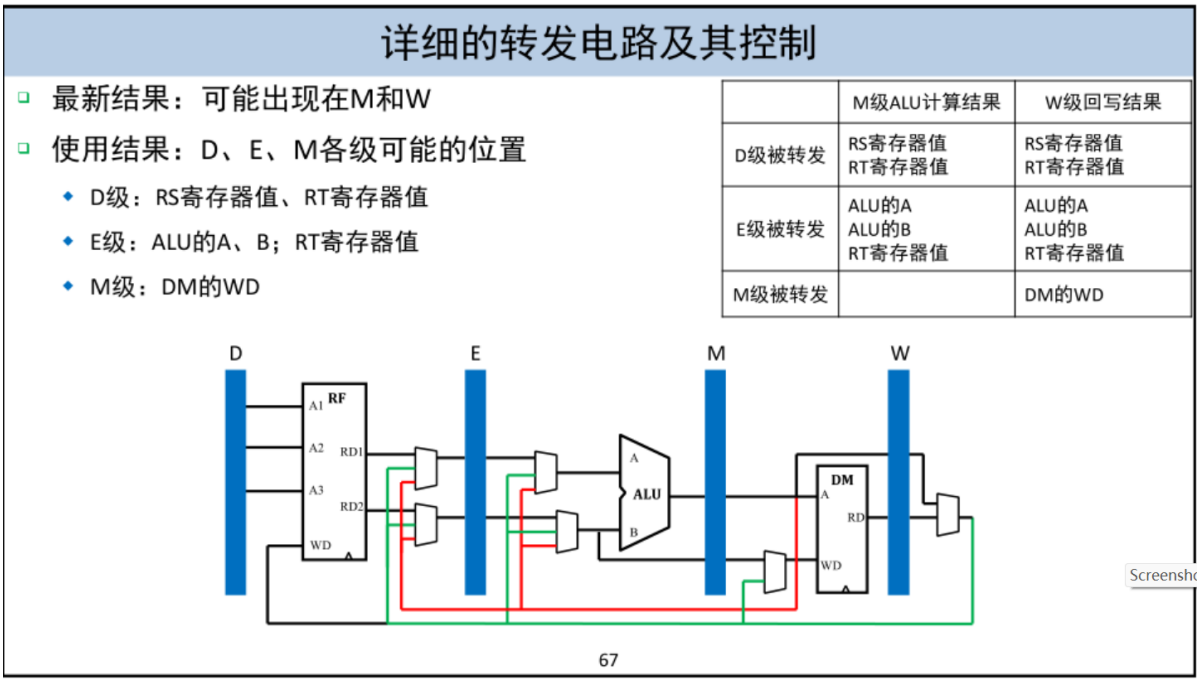
在跳转类指令后添加延迟槽，无论是否跳转，都需要执行跳转指令的下一条指令。进而也不需要废除指令

这解释了为什么对于jal指令（或者其他需要存储下一条指令地址的指令），要把D_PC+8存入31号寄存器。

重定向

重定向和阻塞的区别之一：重定向是需要用的数据已经算出来或者已经读出来了，只不过是没有写到目标寄存器内。因此添加控制信号和复用器即可。但是在其他情况下，比如lw指令，在下一条指令需要用到的时候，这个数据根本就没有读出来，因此只能阻塞而不能重定向。

下附重定向数据通路：



内部转发

当前GPR被写入的值会及时反馈到读取端上。

也即，当读寄存器的地址和同周期写寄存器的地址相同时，我们将读取的内容改为写寄存器的内容，而不是其寄存器中存的值。

A模型

也即课本中给出的重定向的触发条件：即：可以提供的（要写的寄存器）和需要的寄存器（另一条指令要用的寄存器）是同一个寄存器，且均不为0号寄存器。

T模型

Tnew ，经过多少时钟周期可以算出结果并且储存到**流水级寄存器**内。

Tues，指令位于D级时，在经过多少个时钟周期就必须要使用对应的数据，如add指令在E级才使用数据，因此Tuse=1，sw指令，rsTues=1，rtTuse = 2

结论：

- 当Tuse >= Tnew，即使用周期大于其计算周期，可以通过重定向解决
- 当Tnew > Tuse，即计算周期大于使用周期，只能通过阻塞解决。

单独的冲突处理单元

考虑设计单独的冲突处理单元，不和控制器混淆。

其需要产生的信号包括但不限于：冻结信号，刷新信号，供给者选择器信号，需求者选择器信号。

测试数据

1. 首先使用了平台提供的代码进行了初步正确性的测试
2. 对于ALU计算冲突的测试指令：

```
label0: ori $a1, $a3, 50774
label1: ori $0, $v0, 32154
label2: add $a3, $v0, $v1
label9: add $a0, $a1, $0
label10: add $v1, $a2, $a3
label11: sub $a0, $a0, $a1
label15: add $v0, $0, $a2
label16: add $0, $v1, $v1
label18: add $0, $a1, $a3
label21: ori $v0, $v1, 58765
label23: sub $a3, $a2, $v0
label28: ori $v0, $v1, 26022
label31: sub $a3, $a0, $a0
label32: sub $0, $a0, $v1
label34: sub $a2, $v0, $v1
label35: ori $v0, $a0, 5861
label38: sub $v1, $0, $a3
label40: ori $a1, $a3, 59856
label45: add $0, $a3, $v0
label46: sub $v1, $a3, $0
label49: ori $0, $v1, 14314
label54: add $v1, $v0, $a3
label55: sub $0, $a0, $a2
label58: sub $v1, $0, $a3
label68: ori $v1, $a3, 49892
label69: ori $v1, $a3, 24858
label71: ori $v1, $v0, 40501
```

```
label72: ori $0, $0, 3817
label74: ori $a0, $a3, 50379
label75: sub $a3, $0, $v0
label85: add $a3, $0, $a1
label87: sub $a0, $a0, $v1
label89: add $v0, $a3, $v1
label90: sub $v1, $v0, $0
label91: add $v1, $a0, $v1
label93: sub $a0, $v0, $a1
label96: ori $a0, $a3, 2018
label98: add $a2, $a1, $a3
label99: sub $v0, $a3, $v1
label100: add $0, $a2, $a1
label102: add $0, $a1, $a2
label104: ori $a3, $0, 43189
```

3. 对于跳转类指令和ALU计算冲突的测试指令：

```
ori $t2,$0,0x3010
add $t3, $t0, $t2
jr $t3
ori $s0, $zero, 1
ori $s1, $zero, 2
ori $s2, $zero, 3
```

其中通过此测试点解决了jr的Tuse输出错误而导致没有成功阻塞的bug

4. 对于branch类指令与冲突的测试：

```
ori $2,$0,4
ori $3,$3,1
sw $2,0($0)
lw $1,0($0)
sub $1,$1,$3
add $1,$1,$3
add $3,$1,$0
beq $1,$2,labe1
add $1,$1,$2
add $1,$1,$2
labe2: beq $0,$0,labe2
nop
labe1: beq $0,$0,labe1
nop
```

思考题

1. 当beq指令所比较的寄存器值需要更新而未被更新时，仍需要考虑使用重定向或者阻塞的方法解决该问题。如

```
lw $1,0($0)
beq $1,$2,label
```

2. 因为延迟槽的作用即在于：无论是否跳转，跳转指令的下一条指令，也就是延迟槽，都会被执行。因此真正要跳转执行的实际上是延迟槽的下一条指令，其地址为PC+8。
3. 如果来自于功能部件，相当于还是在一个周期内只执行一个指令，与流水线的初衷相悖。
4. GPR内部转发，即W级向D级的转发，确保了从GRF中读出的寄存器的值是正确的值。然而似乎并不需要这样，因为仍然可以通过Forward信号在外部进行转发。
5. - 需求者：D级、E级、M级
   - 供给者：E级、M级、W级

     其中E级可能提供PC+8，M级可能提供ALU_result和PC+8，W级可能提供WD。

     每一个供给者都可以向顺序在它之前的阶段转发数据，因此有E->D,M->D,M->E,W->D,W->E,W->M
6. 就笨人设计的CPU架构来看：首先无论是什么种类的指令，都需要在macro.v宏文件中补充对应的指令判断，然后进行下一步操作。
   - 计算类指令：在controller中扩展ALU_op，RW, imm_ext等信号；接着在ALU_op中实现对应的操作
   - branch类信号：仿照beq信号操作，扩展对应的compare_op等，需要注意的是类似于bgezal类的条件写指令，RW不但与指令有关，还和条件是否成立有关
   - jump类信号：仿照jal，jr来写
   - 存取类信号：仿照sw和lw来写
   - 其他复杂类信号：仔细阅读指令说明，增添Tnew和Tues值，还有可能改变阻塞的逻辑，是否需要在流水线中流水新的信号等
7. 译码器架构：
   - 在单周期cpu的基础上增添了四个寄存器，并把整个cpu分为了F,D,E,M,W五个阶段
   - 设置冲突处理单元输出转发和阻塞信号
   - 每个阶段对应一个controller，译码每个阶段的信号。

优势：不同阶段使用不同的controller似乎避免了信号的一级一级传送

不足：过多的controller因为指令的不同增大了写出bug的可能性。

注意事项

- 为了解决数据冒险而设计的转发数据来源必须是某级流水线寄存器，不允许对功能部件的输出直接进行转发。否则跟单周期CPU没有区别，在一个周期内进行了比较和更新，而不是并行。
- Tnew_E跟Tuse比较就看E阶段指令要写的寄存器和D阶段指令要用的寄存器一不一样
- Tnew_M只需要Tnew_E减一在跟0取一个最大值即可。比较的话也是同理
- 然后输出结果，分别对应EM段是阻塞还是转发
- 注意Tnew的意思，是存入流水级寄存器，而不是存入寄存器堆里的寄存器！！！
- 注意assign里面的冒号不能写成中文冒号

bug点

- GRF的RW看的不是D阶段控制器的RW，因为真正写的时候那条指令是在W阶段！！打印的PC也是，他的PC应该是被传到W阶段的PC，而不是现在读出的PC，以及究竟写哪个寄存器也是根据W级的指令决定的，跟D级指令无关。
- DM同理，DM的PC也是传过去的PC！！
- 由于向寄存器中写的数据都在最后一个阶段统一处理，所以但凡是要往寄存器里写的数据都得通过寄存器往后传
- jal的话因为已经把PC值一直传过去了，所以直接在最后加8就行。
- 注意译码器译码的和真正执行的是不是一个指令！！（GRF写操作的教训）
- 关于jr：很多跳转出错的，多半是jr的Tuse由于也属于R类型被搞成了2,导致在应该阻塞的时候没有阻塞。
- compare_condition信号需要一级一级往下传（条件写的情况中，以便最后在条件写寄存器的时候判断条件是否成立）
- D级的npc在计算时需要传入两个值：PC_F和PC_D。PC_F用来计算常规情况，PC_D用来计算跳转等的情况，在jal的存入中，PC+8中的PC用的是流到W级的PC
- 凡是通过寄存器往下流的值，一定是转发之后的值！
- 对于某一个指令的某一个数据需求，我们定义需求时间 T_{use} 为：这条指令位于 D 级的时候，再经过多少个时钟周期就必须要使用相应的数据。例如，对于 beq 指令，立刻就要使用数据，所以 $T_{use}=0$。对于 add 指令，等待下一个时钟周期它进入 E 级才要使用数据，所以 $T_{use}=1$。而对于 sw 指令，在 E 级它需要 GPR[rs] 的数据来计算地址，在 M 级需要 GPR[rt] 来存入值，所以 $rs\_T_{use}=1$，$rt\_T_{use}=2$。

  我们可以归纳出 T_{use} 的两条性质：

  - T_{use} 是一个定值，每个指令的 T_{use} 是一定的；
  - 一个指令可以有两个 T_{use} 值。

  对于某个指令的数据产出，我们定义供给时间 T_{new} 为：位于**某个流水级**的**某个指令**，它经过多少个时钟周期可以算出结果并且存储到流水级寄存器里。例如，对于 add 指令，当它处于 E 级，此时结果还没有存储到流水级寄存器里，所以此时它的 $T_{new}=1$，而当它处于 M 或者 W 级，此时结果已经写入了流水级寄存器，所以此时 $T_{new}=0$。

  Screenshot(Alt + A)

  我们可以归纳出 T_{new} 的两条性质：

  - T_{new} 是一个动态值，每个指令处于流水线不同阶段有不同的 T_{new} 值；
  - 一个指令在一个时刻至多有一个 T_{new} 值（一个指令至多写一个寄存器）。

周一上午任务：

1. 看往年博客，搞明白一些复杂指令该怎么填上去
2. 再把转发阻塞什么的搞一搞，举几个例子自己运行一下来理解里面的机理

周一update

1. 各类指令在不同级时可能出现的数据冒险。怎么转发，转发什么！！（列一个表出来）
2. 向量部分选择：src[x-:2]，选择src中x和x-1两位输出，src[x+:2]，选择src中x和x+1两位输出

   `A[base + : width];`或者`A[base - : width]`

   其中base可以是变量，但是width必须是常数

3. 通过寄存器往下流水的数据必须是通过转发之后的值

4. 画一个类似于这样的表格

time	D	E	M	W
1	lw $4, 0($0)	nop	nop	nop
2	sw $4, 0x2000($0)	lw $4, 0($0)	nop	nop
3	sw $4, 0x2004($0)	sw $4, 0x2000($0)	lw $4, 0($0)	nop
4	sw $4, 0x2008($0)	sw $4, 0x2004($0)	sw $4, 0x2000($0)	lw $4, 0($0)
5	sw $4, 0x200c($0)	sw $4, 0x2008($0)	sw $4, 0x2004($0)	sw $4, 0x2000($0)

5. 对于不需要读寄存器的指令（Tuse不存在）缺省值应当设置为最大值3

6. jal存的一直是pc+8，因为可以跳回，这跟branch类没关系，branch是不用管的

7. 跳转类指令分类：

   - BXXZAL：无条件link，即只要是这个指令就需要把pc+8存入。但是只有符合条件才跳转？~~maybe,课上还是得好好研究操作解释~~
   - BXXZALC：条件写，用老方法就行，必须记得把对应的信号都流水下去。（其实也就一个compare_condition需要流，因为RW可以过去了再翻译，PC是一直往后带的已经在最后的地方加了。这么说来compare模块其实也可以跟controller模块一样每个地方都有一块，分布式比较。
   - BXXZALR：条件link到指定的寄存器，需要流的数据：compare_condition，写入的寄存器编号可以从W级的controller获得，写入数据也是W级获得，只传compare_condition即可。

     扩展步骤：在macro里添加（注意ra等的信号），在compare里添加对应的比较方式，在controller里的条件写里面补充，同时补充compare_op。

     无论过不过P5都要把P5的代码优化一下，compare也写成分布式的，就不用传递了。
   - BXXZALL：清空延迟槽，条件跳转，否则清空延迟槽。还是compare中补充，都在D中完成即可，类似PCsrc，branch &(!con_reset |compare) 传给F_D寄存器的reset，但是如果是stall状态下的话也不清空。内部的enable信号优先级最高
   - lwso：lw的data大于0写入，否则不写入：仍然是条件写寄存器

     只用把那个compare_condition在改一改，不用在内部再加了，外部判断一下作为一个新的信号传入W级，W级的compare_condition用控制信号选择是从D级传过来的信号还是M级传过来的信号。这个控制信号需要从controller里面输出出来。

8. 今天彻底是把函数稍微弄懂一点了，上次写错有以下原因：

9.
```
89      function [7:0] gray(
90         input [7:0] src
91      );
92      wire [1:0] g1;
93      wire [1:0] g2;
94      wire [1:0] g3;
95      wire [1:0] g4;
96      begin
97         assign g1 = src[1:0];
98         assign g2 = src[3:2];
99         assign g3 = src[5:4];
100        assign g4 = src[7:6];
101        assign gray = {g4,g3,g2,g1};
102     end
103     endfunction
```

- 首先是begin_end是顺序块，不能使用连续赋值语句，也不能在这其中定义变量。也即assign在这里面是不能使用的，同时g1等变量的声明也必须放在外面
- 之所以想到把wire型改成reg型，是忽然想到之前用到的函数都是integer类型，自动就是一个寄存器，于是抱着试一试的心态，没想到真的成了。

```
function [7:0] gray;
   input [7:0] src;
   reg [1:0] g1;
   reg [1:0] g2;
   reg [1:0] g3;
   reg [1:0] g4;
   begin
   g1 = src[1:0];
   g2 = src[3:2];
   g3 = src[5:4];
   g4 = src[7:6];
   gray = {g1,g2,g3,g4};
  end
endfunction
```

附上一张正确的代码，这次用函数可别再用错辣。

10. branch_link：写回当前PC+8
11. 清空延迟槽：清空F_D寄存器，不冻结PC，也不清空D_E寄存器，branch_likely，

12. bonall：值互为相反数则执行延迟槽后跳转并连接PC+8，否则不执行延迟槽，还连接PC+8。
    - 因为不管怎么样都连接PC+8，所以不是条件写，跟jal的信号同步
    - 不执行延迟槽：也就是清空延迟槽，考虑在controller信号里面输出一个com_flush,F_D寄存器的reset信号除了本身的reset信号，还有这个com_flush和compare_condition并在一起的信号，还是不能处于阻塞状态，即enable的优先级最高，
    - 由于是branch类，所以Tuse为0，无关信号的Tnew都设成最大值。
13. lhogez：从DM中取半字，半字中1多就写入半字到rt，0多就往寄存器里面写PC+4
    - 看来compare的实例化是不得不搞了，否则这个根本没办法写外面，太麻烦了，并且这里实例化后要注意和原来的流下来的compare_condition区分。
    - 实例化后可以直接在controller里面新增一个对应的com_op，以及判断是否是这个指令的sel_write信号。
    - 然后在W级的compare模块中把从M级传过来的数据传进去，得到compare_condition
    - 对于mux的选择，推荐是不要改动原来的，而是把原来的当作复位器的一个选择进行输入。

      对于数据选择，就把com_condition，sel_write拼在一起形成一个两位的数据！，如果是2'b11，就选择原来的writedata（因为lhogez和lh的信号一致，或者直接搞读出来的信号，更直接），如果是2'b10，选择原来的writedata（不是这个信号），如果是2'b01，选择pc+4（是这个信号但是不满足）

      对于选择目标寄存器的信号：仍然是com_condition和sel_write拼成两位信号：2'b11，原来的RD_W；2'b10，原来的RD_W；2'b01，31号寄存器，2'b00，原来的RD_W

      碎碎念一下：拼成两位信号真的好好用，所有信号一目了然

      然后强调一下：传入hazard的writeReg_W一定是经过判断之后的writeReg_W，因此要把经过选择后的writeReg_W再传入Hazard中。

      类比到其他指令，如果因为条件之类的使得use或者write的寄存器跟本来controller输出的不同了，一定要把最终使用的传给hazard。

- 考虑这个指令的Tnew和Tues，Tnew就还是正常的3，Tues的话跟load类型指令一样，不用纠结。因此阻塞的话还是根据自己写的阻塞逻辑来。
- bltzal：rs的值小于0时，跳转到***的位置，无条件连接pc+8
    - 由于是无条件连接PC+8，因此jal有的信号bltzal都有
    - 扩展新的com_op和compare模块
    - 注意跟正常的branch的nextpc不同了，因此需要扩展npc模块，同时增加一个信号对PCbranch进行选择，1则选择bltzal，0则选择原来的pcbranch。
    - Tues还是0，写的寄存器是31，这个包在ra里面了，不用管
- addei，不支持溢出的有符号加法

  判断溢出则向rt中写入扩展后的立即数，不溢出则向rt中写入加法的结果。

  直接将以上所有的计算过程，包括判断是否溢出什么的全封装到alu里面。

  注意添加immext的信号，增加成两位，处理高位1扩展。

  因为是计算指令，所以其他地方需要改的不多
- lbget，读一个字节，字节读出来小于0，添加到base里面，读出来大于等于0，写到rt中。
    - 跟之前那个lh类的很像，首先是前期指令信号都和lb相同，有lb的都或一个lbget

    - 然后还是需要多实例化一个compare模块，扩展compare信号和compare模块内部操作，得到compare_condition，此处的compare_condition跟流水线流下来的compare_condition不能混淆，因为branch类写寄存器的控制信号可能就是这么来的，可以设置一个总的comdition，根据单独的那个信号选一下是用本来的还是用这个新生成的。
    - 在controller中用一个单独的任意信号指明是这个lbget信号
    - 以上两个信号拼在一起成为一个两位的信号。并对RD_W进行选择。

- 新lwso（严重怀疑指令的名字可以随便起）

  31号寄存器和对应内存地址中储存的数相加，溢出的话就不写31寄存器，不溢出的话就把相加的结果写入31号寄存器（目测没有rt，或者rt的值固定为31）

    - 仍然需要在W级实例化一个compare模块，如果rt的值在机器码中固定是31还比较好办，但如果不是固定为31，那就需要在grf模块中添加一个端口输出31号寄存器的值，然后流水到最后，同时注意内部转发。
    - 将用到的寄存器值向后流水
    - 在W级中进行判断得到是否溢出的信号。
    - controller内部D2R改一下，给这个信号用，同时com_write也需要用上了，WD需要添加，但是目标寄存器什么的都不用动。主要控制的是RW_W
    - RW_W还是用老方法进行判断。

- blezalc

  学长说课上给出的RTL表述有问题，表示很害怕（

  如果跳转则链接寄存器

    - branch类条件写寄存器。condition是rs<=0，把compare信号往下传下去就行
    - 写寄存器那些都按jal给加上，仍然是老方法控制那个RW，写不写入

- lrm

  根据得到的内存中的数据的0-4位决定要写入的寄存器，并把rt写入这个寄存器

    - 首先是load类指令，load类该有的他也有
    - 接着是在W级中，controller新增信号表明是该指令，然后在已经确定的writeReg和0-4位之间进行再次选择，注意把已经确定的writereg改名，令最终的writereg跟传进冒险处理单元的一致。

感觉忽然对那个E级向前转发有点明白了。

转发的来源其实就是无非就是以下几种：PC值，ALU值，内存值

而E阶段目前只能往前转发PC值，因为ALU还没到寄存器中，寄存器写回值它更是没有。因此往前转发E阶段指令地址的PC+8.比如E阶段现在是jal指令，D阶段是个跳转指令，需要用到$ra去比较，那么此时E级即往前转发，还是**E级要写的就是D级要用的**，用E级的D2R_E信号对其进行选择。2'b10就传PC+8,其他传高阻值。

M阶段，可以往前转发ALU_result，和PC值，比如M阶段现在是jal/jalr指令，而E阶段是一个需要用到$31/目标寄存器去运算的指令，于是向前转发PC+8。如何进行具体的选择（即转发pc+8还是ALU_result），不要看前面，看的是后面，即M阶段是要写什么，根据冲突的定义，**M阶段要写的就是E阶段需要用的**，进而根据M阶段的controller输出的D2R_M，确定往前传输的数据，合并称为WD_M

W阶段，可以往前转发内存值、PC值和ALU_result值，按照上述方法合并成WD，也即最终写回寄存器的值。

如果是条件写的话，每一级的RW也都得是处理过之后的RW

条件访存注意阻塞，一般的处理方式是只要是这个信号就阻塞