

# 北航2021计组P5设计文档

黄雨石20376156

## 一、CPU设计方案

### (一) 总体设计概述

本CPU为verilog实现的流水线MIPS - CPU，支持的MIPS-lite2指令集包含 { addu, subu, ori, lw, sw, beq, lui, j, jal, jr, nop } 共11条指令 处理器为五级流水线设计。为了实现这些功能，CPU主要包含了PC、IM、NPC、CMP、GRF、ALU、DM、EXT、CTRL、以及每级模块之间的流水寄存器模块、确定转发和暂停机制的处理冒险的模块单元等

### (二) 关键模块定义

- F级

- 1.PC

端口名	方向	描述
npc[31:0]	I	D级npc当前值
pc[31:0]	O	F级pc当前值
pwe	I	使能信号
reset	I	复位信号
clk	I	时钟信号

- 2.IM

端口名	方向	描述
PC[31:0]	I	F级pc当前值
instr[31:0]	O	F级指令

- D级

- 1.D\_REG

端口名	方向	描述
reset	I	复位信号
clk	I	时钟信号
we	I	使能信号
instr_in[31:0]	I	F级指令
instr_out[31:0]	O	F级流水过来的指令
pc_in[31:0]	I	F级pc值
pc_out[31:0]	O	F级流水过来的pc值

## 2.D\_CMP

端口名	方向	描述
rs_d[31:0]	I	D级rs寄存器数据
rt_d[31:0]	I	D级rt寄存器数据
type[2:0]	I	选择信号
b_j	O	是否跳转

- type:
  - 3'd0: 跳转指令为 beq
  - 3'd1: 跳转指令为 bne
  - 3'd2: 跳转指令为 blez
  - 3'd3: 跳转指令为 bgez
  - 3'd4: 跳转指令为 bltz
  - 3'd5: 跳转指令为 bgtz

## 3.D\_EXT

端口名	方向	描述
imm16[15:0]	I	D级16位立即数
ext_sel	O	选择信号
ext_out[31:0]	I	扩展后结果

- ext\_sel:
  - 1'b0: 0拓展
  - 1'b1: 符号拓展

## 4.D\_GRF

端口名	方向	描述
rs[4:0]	I	rs寄存器地址
rt[4:0]	I	rt寄存器地址
gwa[4:0]	I	写入寄存器的地址
gwd[31:0]	I	写入寄存器的值
rs_d	O	rs寄存器的值
rt_d	O	rt寄存器的值
gwe	I	寄存器写使能
reset	I	复位信号
clk	I	时钟信号
W_pc	I	W级pc值

### 5.D\_NPC

端口名	方向	描述
D_pc[31:0]	I	D级pc值
F_pc[31:0]	I	F级pc值
imm26[25:0]	I	26位立即数
imm16[15:0]	I	16位立即数
rs_d[31:0]	I	rs寄存器的值
npc_sel[2:0]	I	选择信号
b_j	I	分支指令是否跳转
npc[31:0]	O	F级pc下一周期的值

- npc\_sel:
  - 3'd0:  $F\_pc + 4$
  - 3'd1: j\_i 类指令 {D\_pc[31:28], imm26, 2'b0}
  - 3'd2 j\_r 类指令: rs\_d
  - 3'd3: branch 类指令  $D\_pc + 4 + \{14\{imm16[15]\}\}, imm16, 2'b0\}$

- E级

#### 1.E\_REG

端口名	方向	描述
reset	I	复位信号
clk	I	时钟信号
we	I	使能信号
instr_in[31:0]	I	D级指令
instr_out[31:0]	O	D级流水过来的指令
pc_in[31:0]	I	D级pc值
pc_out[31:0]	O	D级流水过来的pc值
ext_in[31:0]	I	D级立即数拓展结果
ext_out[31:0]	O	D级流水过来的立即数拓展结果
rs_d_in[31:0]	I	D级rs寄存器值
rs_d_out[31:0]	O	D级流水过来的rs寄存器值
rt_d_in[31:0]	I	D级rt寄存器值
rt_d_out[31:0]	O	D级流水过来的rt寄存器值

## 2.ALU

端口名	方向	描述
a[31:0]	I	数据a
b[31:0]	I	数据b
alu_out[31:0]	O	alu计算结果
alu_sel[4:0]	I	选择信号

### o alu\_sel:

- 5'd0: add:a+b
- 5'd1: sub:a-b
- 5'd2: or:a|b
- 5'd3: and:a&b
- 5'd4: xor:a^b
- 5'd5: sll:a<<b
- 5'd6: srl:a>>b
- 5'd7: sra:\$signed(\$signed(a)>>>b)
- 5'd8: slt:(\$signed(a)<\$signed(b))?32'b1:32'b0
- 5'd9: sltu:a<b
- 5'd10: lui:b<<16

## • M级

### 1.M\_REG

端口名	方向	描述
reset	I	复位信号
clk	I	时钟信号
we	I	使能信号
instr_in[31:0]	I	E级指令
instr_out[31:0]	O	E级流水过来的指令
pc_in[31:0]	I	E级pc值
pc_out[31:0]	O	E级流水过来的pc值
alu_in[31:0]	I	E级alu的计算结果
alu_out[31:0]	O	E级流水过来的alu的计算结果
rt_d_in[31:0]	I	流水到E级的rt寄存器的值
rt_d_out[31:0]	O	流水到M级的rt寄存器的值

## 2.M\_DM

端口名	方向	描述
clk	I	时钟信号
reset	I	复位信号
dwe	I	写使能
dm_out[31:0]	O	dm的输出结果
dm_sel[2:0]	I	dm的选择信号
dwd[31:0]	I	dm的写入值
dwa[31:0]	I	dm的写入地址
M_pc[31:0]	I	M级pc值

### ◦ dm\_sel:

- 3'd0:按字读或写
- 3'd0:按半字读或写
- 3'd0:按字节读或写
- 3'd0:按无符号半字读
- 3'd0:按无符号字节读

## • W级

### 1.W\_REG

端口名	方向	描述
reset	I	复位信号
clk	I	时钟信号
we	I	使能信号
alu_in[31:0]	I	流水到M级的alu计算结果
alu_out[31:0]	O	流水到W级的alu计算结果
pc_in[31:0]	I	M级pc值
pc_out[31:0]	O	W级pc值
dm_in[31:0]	I	M级dm输出结果
dm_out[31:0]	O	流水到W级dm输出结果
instr_in[31:0]	I	M级指令
instr_out[31:0]	O	W级指令

- 控制

- 1.CTRL

端口名	方向	描述
instr[31:0]	I	指令
rs[4:0]	O	rs寄存器地址
rt[4:0]	O	rt寄存器地址
rd[4:0]	O	rd寄存器地址
imm16[15:0]	O	16位立即数
imm26[25:0]	O	26位立即数
shamt[4:0]	O	移位数
load	O	读取dm类指令
store	O	储存到dm类指令
count_i	O	有立即数的计算类指令
count_r	O	无立即数的计算类指令
branch	O	分支类指令
shifts	O	用shamt的移位指令
shiftr	O	用寄存器值进行移位的指令
j_r	O	用寄存器值进行跳转类指令
j_i	O	用立即数进行跳转类指令
j_l	O	跳转并链接类指令
type[2:0]	O	CMP选择信号
ext_sel	O	EXT选择信号
gwe	O	GRF写使能信号
gwa_res[4:0]	O	GRF写入地址选择信号
gwd_sel[2:0]	O	GRF写入地址值选择信号
npc_sel[2:0]	O	NPC选择信号
alu_sel[4:0]	O	ALU选择信号
alu_a_sel[1:0]	O	ALU数据a选择信号
alu_b_sel[2:0]	O	ALU数据b选择信号
dwe	O	DM写使能信号
dm_sel[2:0]	O	DM选择信号

◦ gwa\_res:

- count\_r:rd
- count\_i | load:rt

- jal: 5'd31
- o gwd\_sel:
  - 3'd0: alu
  - 3'd1: dm
  - 3'd2: pc+8
- o alu\_a\_sel:
  - 2'b0: rs\_d
  - 2'b1: rt\_d
- o alu\_b\_sel:
  - 3'd0: rt\_d
  - 3'd1: ext\_out
  - 3'd2: rs\_d[4:0]
  - 3'd3: shamt
- o 分类

```
assign load = lw | 0;
assign store = sw | 0;
assign count_r = addu | subu | 0;
assign count_i = ori | lui | 0;
assign branch = beq | 0;
assign shifts = 0;//shift shamt
assign shiftv = 0;//shift $x
assign j_r = jr | 0;//jump $x
assign j_i = jal | j | 0;//jump imm26
assign j_l = jal | 0;//link
```

## • 冒险

### 1.SU

端口名	方向	描述
D_instr[31:0]	I	D级指令
E_instr[31:0]	I	E级指令
M_instr[31:0]	I	M级指令
stall	O	阻塞信号

## (三)转发暂停控制

### • 转发

此处我采用的是教程中的转发机制，即只要下一级的写入地址一样且不为0就转发（W到D级我们使用内部转发）对于写入地址我做了如下处理

```
assign gwa_res = (count_r)? rd : (count_i | load)? rt : (jal)? 5'd31 : 5'b0;
```

由于已经对指令进行了分类,我们对于要写入寄存器的指令，我的grf的写入地址就直接是它们要写入的实际地址，而对于不需要写入寄存器我们直接认为它写入的地址为0



```
assign fw_D_rs_d = (D_rs == 0)? 0:
                  (D_rs == E_gwa_res)? fw_E_wd:
                  (D_rs == M_gwa_res)? fw_M_wd:
                  D_rs_d;
```

因此我们首先判断该级读出地址是否为0，如果为0那么读出的就是0，如果不为0那么对于不写寄存器的指令就会因为其写入地址为0而不转发，因此在此转发机制下不考虑写使能的判断，同时原级也视为一种优先级最低的转发，然后按最近的级数优先级递减，对于还没有产生结果转发了会怎么样呢？当然在之后它产生了结果它还是会被再转发一次，即二次转发（在此不考虑直到数据必须被使用时还没产生正确结果的情况，因为这个时候就需要暂停阻塞了，即插入nop直到判断在数据必须被使用前产生了结果）

那么我们要转发的接收位置一共有以下几种

- D级
  1. CMP:rs\_d, rt\_d
  2. NPC:rs\_d(j\_r类指令)
- E级
  1. ALU:a ,b
- M级
  2. DM:dwd
- W级
  1. GRF:rs\_d, rt\_d(内部转发解决)

那么对于具体要转发过来的东西，我做了如下处理：

```
assign fw_E_wd = (E_gwd_sel == `gwd_pc8)? E_pc + 8:
                 32'd0;

assign fw_M_wd = (M_gwd_sel == `gwd_alu)? M_alu_out:
                 (M_gwd_sel == `gwd_pc8)? M_pc + 8:
                 32'd0;

assign fw_W_wd = (W_gwd_sel == `gwd_alu)? W_alu_out:
                 (W_gwd_sel == `gwd_dm)? W_dm_out:
                 (W_gwd_sel == `gwd_pc8)? W_pc + 8:
                 32'd0;
```

即有可能被转发的结果接着流水到下一级，再由多路选择器判断该级需要转发回去的是该指令从上一级流水过来的数据还是该级产生的数据（由于没有遵守必须从寄存器转发的要求，因此需满足从非寄存器输出端口的转发一定转发到该级之前），该判断的目的是对要转发的数据进行一下简单的筛选，保证一级只有一个数据被转发，但仍然可能转发错误的数据(在上面已经说明转发了错误的数据对于结果没有影响)

在该方案中有个细节十分重要，就是我们参与流水的寄存器的值必须是转发结果，如果直接流水当级寄存器的值，即使正确的寄存器的值在之后仍会被转发过来替代该值，但当正确寄存器的值已经写入寄存器而该寄存器的值却还没到要使用的阶段，那么对于该条指令就不可能得到正确结果，因为已经写入寄存器的值除内部转发外不会再转发比如(假设所有非0号寄存器中的值均不为0)

```
addu $1, $2, $3
nop
sw $1, 0($0)
```

综上，对于该方法转发单元的处理不必单独列出转发表判断何时转发，每次转发的数据，接收到的数据是否正确都不需要考虑了，同时我将转发单元集成到mip.v的顶层模块中更加直观便于修改

- 暂停

对于暂停机制我采取了教程中的AT法

首先，定义一下 Tuse 和 Tnew：

- Tuse：指令进入 **D 级**后，其后的某个功能部件再经过多少时钟周期就**必须要**使用寄存器值。对于有两个操作数的指令，其**每个操作数的 Tuse 值可能不等**（如 store 型指令 rs、rt 的 Tuse 分别为 1 和 2）。
- Tnew：位于 **E 级及其后各级**的指令，再经过多少周期就能够产生要写入寄存器的结果。在我们目前的 CPU 中，W 级的指令 Tnew 恒为 0；对于同一条指令， $Tnew@M = \max(Tnew@E - 1, 0)$

具体机制如下：

当两条指令发生数据冲突（**前面指令的写入寄存器，等于后面指令的读取寄存器**），我们就可以根据 Tnew 和 Tuse 值来判断策略。

1. Tnew=0，说明结果已经算出，如果指令处于 WB 级，则可以通过寄存器的内部转发设计解决，不需要任何操作。如果指令不处于 WB 级，则可以通过转发结果来解决。
2. Tnew<=Tuse，说明需要的数据可以及时算出，可以通过转发结果来解决。
3. Tnew>Tuse，说明需要的数据不能及时算出，必须暂停流水线解决。

对于Tuse,Tnew,将Tuse分为Tuse\_rt,Tuse\_rs这样便于通过先前的指令分类分别判断，因为对于同一条指令的两个操作数的Tuse可能不同，如上述的store,Tnew则分为Tnew\_E,Tnew\_M,(**Tnew\_W 不需要，因为可以通过内部转发解决**)

对于stall的暂停信号同样分为stall\_rs,stall\_rt,并且继续做如下细分，stall\_rs\_E,stall\_rs\_M.....

那么当Tuse,Tnew全部被计算出来后可以做如下处理

```
assign stall_rs_E = (Tuse_rs < Tnew_E)&&(D_rs != 5'd0 && D_rs == E_gwa_res);
assign stall_rs_M = (Tuse_rs < Tnew_M)&&(D_rs != 5'd0 && D_rs == M_gwa_res);
assign stall_rs = stall_rs_E | stall_rs_M | 0;

assign stall_rt_E = (Tuse_rt < Tnew_E)&&(D_rt != 5'd0 && D_rt == E_gwa_res);
assign stall_rt_M = (Tuse_rt < Tnew_M)&&(D_rt != 5'd0 && D_rt == M_gwa_res);
assign stall_rt = stall_rt_E | stall_rt_M | 0;

assign stall = stall_rs | stall_rt;
```

对于Tuse:

- Tuse\_rt:

```
assign Tuse_rs = (D_branch | D_j_r)? 3'd0 :
((D_count_r & !D_shifts) | D_count_i | D_load | D_store)? 3'd1:
3'd3;
```

- Tuse\_rs:

```
assign Tuse_rt = (D_branch)? 3'd0:
                (D_count_r)? 3'd1:
                (D_store)? 3'd2:
                3'd3;
```

对于Tnew:

- Tnew\_E

```
assign Tnew_E = (E_count_i | E_count_r)? 3'd1:
                (E_load)? 3'd2:
                3'd0;
```

- Tnew\_M

```
assign Tnew_M = (M_load)? 3'd1:
                3'd0;
```

一但要暂停，我们直接将指令冻结在D级，即在下一个时钟上升沿到下一个时钟上升沿来临之间该指令仍然在D级，E级被清空，即插入nop指令，即如下：

- 冻结 PC 的值
- 冻结 F/D 级流水线寄存器的值
- 将 D/E 级流水线寄存器清零（这等价于插入了一个 nop 指令）

如此，就完成了暂停机制的构建。

我将阻塞单元放在单独的SU模块中，最后附上阻塞表（只列出课下要求指令）：

IF/ID当前指令			ID/EX当前指令			EX/MEM
指令类型	源寄存器	Tuse	Tnew			
			cal_r/rd	cal_i/rt	load/rt	load/rt
			1	1	2	1
beq	rs/rt	0	stall	stall	stall	stall
cal_r	rs/rt	1			stall	
cal_i	rs	1			stall	
load	rs	1			stall	
store	rs	1			stall	
store	rt	2				
jr	rs	0	stall	stall	stall	stall

## 二、测试方案

自己构造小范围测试测每个冲突与转发，使用github学长写的对拍测试，以下为学长数据强度

```
"forward_valid_ratio": 0.7892376681614349,
"forward_count": 57,
"stall_count": 9,
"forward_coverage": 0.5327102803738317,
"stall_coverage": 0.6,
"grade": {
  "forward": {
    "average": 54.82758620689654,
    "warning": [
```

```

        "cal_ri <~~ jal",
        "br_r2 <~~ load",
        "br_r2 <~~ jal",
        "load <~~ cal_rr",
        "load <~~ load",
        "load <~~ lui",
        "load <~~ jal",
        "store <~~ jal",
        "jr <~~ cal_ri",
        "jr <~~ load",
        "jr <~~ jal"
    ],
    "details": {
        "cal_rr <~~ cal_rr": 96.66666666666666,
        "cal_rr <~~ cal_ri": 100.0,
        "cal_rr <~~ load": 90.0,
        "cal_rr <~~ lui": 100.0,
        "cal_rr <~~ jal": 66.66666666666667,
        "cal_ri <~~ cal_rr": 86.66666666666666,
        "cal_ri <~~ cal_ri": 100.0,
        "cal_ri <~~ load": 80.0,
        "cal_ri <~~ lui": 100.0,
        "cal_ri <~~ jal": 0,
        "br_r2 <~~ cal_rr": 100.0,
        "br_r2 <~~ cal_ri": 100.0,
        "br_r2 <~~ load": 0,
        "br_r2 <~~ lui": 100.0,
        "br_r2 <~~ jal": 0,
        "load <~~ cal_rr": 0,
        "load <~~ cal_ri": 73.33333333333333,
        "load <~~ load": 0,
        "load <~~ lui": 0,
        "load <~~ jal": 0,
        "store <~~ cal_rr": 80.0,
        "store <~~ cal_ri": 100.0,
        "store <~~ load": 73.33333333333333,
        "store <~~ lui": 73.33333333333333,
        "store <~~ jal": 0,
        "jr <~~ cal_rr": 70.0,
        "jr <~~ cal_ri": 0,
        "jr <~~ load": 0,
        "jr <~~ jal": 0
    }
},
    "stall": {
        "average": 58.0,
        "warning": [
            "load <~~ load",
            "store <~~ load",
            "jr <~~ cal_ri",
            "jr <~~ load"
        ]
    },
    "details": {
        "cal_rr <~~ load": 100.0,
        "cal_ri <~~ load": 100.0,
        "br_r2 <~~ cal_rr": 100.0,
        "br_r2 <~~ cal_ri": 100.0,
        "br_r2 <~~ load": 100.0,

```

```

        "load <~~ load": 0,
        "store <~~ load": 0,
        "jr <~~ cal_rr": 80.0,
        "jr <~~ cal_ri": 0,
        "jr <~~ load": 0
    }
}
},

```

期望输出（结果将与魔改版mars的结果进行对比）：

```
test1:Accepted!
```

## 三、思考题

### 流水线冒险

**1. 在采用本节所述的控制冒险处理方式下，PC的值应当如何被更新？请从数据通路和控制信号两方面进行说明。**

PC值更新有四种情况，一是指令为beq的时候，根据npc\_sel与b\_j信号控制更新， $PC = PC + 4 + \text{sign\_ext}(\text{imm16})$ ；二是j或jal指令时，根据pc\_sel信号控制更新， $PC = \text{sign\_ext}(\text{imm26})$ ；三是jr指令时，根据npc\_sel信号一起控制更新 $PC = \text{grf}[\text{rs}]$ ；四，其余指令，则 $PC = PC + 4$ 。

**2. 对于jal等需要将指令地址写入寄存器的指令，为什么需要回写PC+8？**

因为采用延迟槽的编译优化方法，在D级决定是否进行跳转，这个时候下一条延迟槽中的指令已经进入了流水线，所以在跳转指令的下一条指令必须被执行。从而达到充分利用流水线的性能，故应该回写延迟槽之后那条指令的地址。

### 数据冒险的分析

**1. 为什么所有的供给者都是存储了上一级传来的各种数据的流水级寄存器，而不是由ALU或者DM等部件来提供数据？**

此次CPU采用的是流水线方法构造，每一级流水线寄存器存储对应流水级的指令以及相关数据值，在时钟上升沿到来时，将相关数据送入对应流水级供该级使用，流水线寄存器起到隔断作用，让每个流水级中的数据不会受到上一个流水级的数据通路的影响。

### AT法处理流水线数据冒险

**1. “转发（旁路）机制的构造”中的Thinking 1-4；**

**Thinking 1: 如果不采用已经转发过的数据，而采用上一级中的原始数据，会出现怎样的问题？试列举指令序列说明这个问题。**

将会出现已经修改过的寄存器数值，在之后的指令中使用了未被更改过的原始数值，导致运行结果错误的情况。

代码如下：

```
ori $1, $0, 1
nop
nop
nop
nop
lw $1, 0($0)
sw $1, 4($0)
```

**Thinking 2: 我们为什么要对GPR采用内部转发机制? 如果不采用内部转发机制, 我们要怎样才能解决这种情况下的转发需求呢?**

为了防止W级还未写入GRF的数据在之后的指令中使用导致运行结果出现错误。如果不采用内部转发机制, 可以采用将即将从W级写入寄存器文件的数据端口连到cmp输入端口的转发端来解决, 同时在转发机制中考虑W级转发

**Thinking 3: 为什么0号寄存器需要特殊处理?**

0号寄存器不能更改值, 必须一直维持为0x00000000, 如果之后是向0写值, 那么就不会被转发

**Thinking 4: 什么是“最新产生的数据”?**

数据的“新旧”是通过看转发处距D级/E级/M级(所需要的地方)的距离远近来判断, 距离所需要的地方越近, 则越新。

**2.在AT方法讨论转发条件的时候, 只提到了“供给者需求者的A相同, 且不为0”, 但在CPU写入GRF的时候, 是有一个we信号来控制是否要写入的。为何在AT方法中不需要特判we呢? 为了用且仅用A和T完成转发, 在翻译出A的时候, 要结合we做什么操作呢?**

AT法使建立了指令和对应流水级到相应的A和T信号的映射, 之后比较需要的GRF地址和待转发的数据值是否相等且需要的地址不为0, we是控制是否向GRF写值, AT法中, 如果采用当we为0, 将A(供给者)设为0, 那么转发条件将不会满足, 在此情况下则不需要特判we对其做任何操作。

### 在线测试相关说明

**1.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? 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。此思考题请同学们结合自己测试CPU使用的具体手段, 按照自己的实际情况进行回答。**

解决方案与产生冲突: (具体解决方案见上文(三)转发暂停控制部分)

```
阻塞解决: "cal_rr <~~ load"
           "cal_ri <~~ load"
           "br_r2 <~~ cal_rr"
           "br_r2 <~~ cal_ri"
           "br_r2 <~~ load"
           "load <~~ load"
           "store <~~ load"
           "jr <~~ cal_rr"
           "jr <~~ cal_ri"
           "jr <~~ load"
转发解决: "cal_rr <~~ cal_rr"
           "cal_rr <~~ cal_ri"
           "cal_rr <~~ load"
```

```

"cal_rr <~~ lui"
"cal_rr <~~ jal"
"cal_ri <~~ cal_rr"
"cal_ri <~~ cal_ri"
"cal_ri <~~ load"
"cal_ri <~~ lui"
"cal_ri <~~ jal"
"br_r2 <~~ cal_rr"
"br_r2 <~~ cal_ri"
"br_r2 <~~ load"
"br_r2 <~~ lui"
"br_r2 <~~ jal"
"load <~~ cal_rr"
"load <~~ cal_ri"
"load <~~ load"
"load <~~ lui"
"load <~~ jal"
"store <~~ cal_rr"
"store <~~ cal_ri"
"store <~~ load"
"store <~~ lui"
"store <~~ jal"
"jr <~~ cal_rr"
"jr <~~ cal_ri"
"jr <~~ load"
"jr <~~ jal"

```

相应测试代码由自动生成器生成个别无法测到就手动生成，手动生成：就由上述冲突构造相应案例，由于上述冲突是由课程组给出所以必然全面，自动生成器：观察了一下学长自动生成器的源码，确实有刻意生成冲突，但是由于是有随机数生成故而要避免有的越界情况，因此有的冲突无法生成，必须手动解决。

**对于这款 cpu 加指令注意事项**(以下内容转自<https://roife.github.io/2020/11/23/buaa-co-lab-p5/>)

1. 计算绝对是最简单的，重点在于你的写法。如果你用 `always@(*)` 那么问题不大。但是如果你用的是 `assign` 那么最好使用 `function` 来实现计算指令。
2. 跳转一般也不难，一般是条件跳转 + 条件写。跳转指令一个好处在于它是在 D 级决定是否跳转的，也就是说在 D 级你可以获得全部的正确信息（相反如果是类似于 `lws0` 这种，你必须读到 DM 的值才能决定怎么做）。所以我们的方案是 D 级生成一个 `D_check` 信号然后流水它。然后每一级根据这个信号判断写入地址/写入值之类的。

```

// 检测信号
D_check = D_bgezal & D_b_jump;
//
CU
assign RFDst = // ...
                bgezal ? (check ? 5'd31 : 5'd0) :
                5'd0;

```

3. 条件存储一般是最难的。但是掌握了套路之后也还好。条件存储的特点是必须要到 M 级才知道要写啥，这就给转发之类的造成了困难，所以我们的策略是如果 D 级要读寄存器，而且新指令

可能

要写这个寄存器，那么就 stall。具体来说是这样的：

```
// lws0
wire stall_rs_e = (TuseRS < TnewE) && D_rs_addr && (D_lws0 ? D_rs_addr ==
5'd31 : D_rs_addr == E_RFDst);
// lrm
wire stall_rs_e = (TuseRS < TnewE) && D_rs_addr && (D_lrm ? D_rs_addr :
D_rs_addr == E_RFDst);
```

在 CU 中的写法则与条件跳转类似。

```
// lws0
M_check = D_lws0 && condition;
// CU
assign RFDst = // ...
               lws0 ? (check===1'd1 ? 5'd31 : 5'd0) : // 注意不是直接一个 check
               5'd0;
```

这里要注意的一点是，如果你用的是统一 CU 的写法，那么会出现一个问题：check 信号只有在 M 级传入。这个时候在 E 级的 CU 里面这个信号是不定态 x，这样会导致 RFDst 信号出锅。所以我们这里的写法是 check===1'd1，这样可以排除 x 的情况。