MIPS-C 指令集(共计 55 条)

一、指令分类

指令类型	Instr	读寄存器	写寄存器	use(stage)	new(REG)
cal_R(13)	addu\subu\add\sub\and\nor\or\slt\sltu\sllv\srav\srlv\xor	rs, rt	rd	[rs: E] [rt: E]	[rd: M]
cal_Ist(7)	ori\addi\addiu\andi\slti\sltiu\xori	rs	rt	[rs: E]	[rt: M]
cal_It(1)	lui	---	rt	---	[rt: M]
j_addr(1)	j	---	---	---	---
j_l(1)	jal	---	31	---	[31: E(PC+8)]
j_r(1)	jr	rs	---	[rs: D]	---
j_lr(1)	jalr	rs	rd	[rs: D]	[rd: E(PC+8)]
store(3)	sw\sh\sb	rs, rt	---	[rs: E] [rt: M]	---
load(5)	lw\lh\lhu\lb\lbh	rs	rt	[rs: E]	[rt: W]
b_st(2)	beq\bne	rs, rt	---	[rs: D] [rt: D]	---
b_s(4)	blez\bgez\bltz\bgtz	rs	---	[rs: D]	---
syscall(5)	syscall\break\eret\mtc0\mfc0	---	---	---	---
shift_R(3)	sll\sra\srl	rt	rd	[rt: E]	[rd: M]
hilo_cal	mult\multu\div\divu	rs, rt	---(hilo)	[rs: E] [rt: E]	---
hilo_mf	mflo\mfhi	---(hilo)	rd	---	[rd: M]
hilo_mt	mtlo\mthi	rs	---(hilo)	[rs: E]	---

二、控制信号赋值

A. cal_R(13) \ shift_R(3)

Instr	ALUControl[3:0]	JumpOp[2:0]	DMType[2:0]	BType[2:0]	EXTOp[2:0]	ALUSrc[1:0]	MemtoReg	RegWrite	MemWrite	RegDst
addu	`ALU_add	`JumpOp_notj	`DMType_null	`BType_notb	`EXTOp_null	`ALUSrc_rt	0	1	0	rd
subu	`ALU_sub									
add	`ALU_add									
sub	`ALU_sub									
and	`ALU_and									
nor	`ALU_nor									
or	`ALU_or									
slt	`ALU_slt									
sltu	`ALU_sltu									
sllv	`ALU_sll									
srlv	`ALU_srl									
srav	`ALU_sra									
xor	`ALU_xor									
sll	`ALU_sll				`EXTOp_shamt	`ALUSrc_Imm				
sra	`ALU_sra									
srl	`ALU_srl									

*(slti/sltui 均为 SignExtended)

*(更改 EXTOp，增加输入端口 shamt，添加控制信号`EXTOp_shamt, [2:0])

*(更改 ALUSrc 控制信号，添加`ALUSrc_shamt, [2:0]) → 错误!!! 因为 shamt 扩展已在 EXT 中完成，ALU_B 端口输入仍为 EXT 的输出，因此不必增加控制信号，

*但为防止上机时新指令需要增加扩展信号，提前将信号位数增加为 2.

*(更改 ALUOp 控制信号，添加`ALU_shift_sieries, [3:0])

B. cal_Ist(7) \ cal_It(1)

Instr	ALUControl[3:0]	JumpOp[2:0]	DMType[2:0]	BType[2:0]	EXTOp[1:0]	ALUSrc[1:0]	MemtoReg	RegWrite	MemWrite	RegDst
ori	`ALU_or	`JumpOp_notj	`DMType_null	`BType_notb	`EXTOp_zero	`ALUSrc_Imm	0	1	0	rt
addi	`ALU_add				`EXTOp_sign					
addiu	`ALU_add				`EXTOp_sign					
andi	`ALU_and				`EXTOp_zero					
slti	`ALU_slt				`EXTOp_sign					
sltiu	`ALU_sltu				`EXTOp_sign					
xori	`ALU_xor				`EXTOp_zero					
lui	`ALU_add				`EXTOp_lui					

C. j(4) (略)

D. store(3) (略) \ load(5) (略)

E. b_st(2) (略) \ b_s(4) (略)

F. hilo(8)

G. syscall(5) (略)

(32)“ALU.v”: A, B, D

(25)“EXT.v”：A_shift_3, B, D, E

(50) “ALUSrc”(MUX):

三、P5\P6 新增部件

1. _Decoder(CU + Decoder)

信号名	方向	功能描述
[31:0]Instr	I	
[4:0]rs	O	Operand
[4:0]rt	O	Operand
[4:0]rd	O	Operand
[4:0]shamt	O	Operand
[15:0]Imm	O	Operand
[4:0]HILOType	O	Control_Unit
[3:0]ALUControl	O	Control_Unit
[2:0]JumpOp	O	Control_Unit
[2:0]DMType	O	Control_Unit
[2:0]BType	O	Control_Unit
[2:0]EXTOp	O	Control_Unit
[1:0]ALUSrcA	O	Control_Unit
[1:0]ALUSrcB	O	Control_Unit
[3:0]ALUControl	O	Control_Unit
MemtoReg	O	Control_Unit
RegWrite	O	Control_Unit
MemWrite	O	Control_Unit
jal	O	MUX_Sel 1: *_PC8 0: M_ALUResult\W_WD3
[5:0] InstrType	O	Instr_Type(_FSU)
[4:0] RegDst	O	mips_FSU

重大失误：

Shift_R 系列指令操作数为[rt]而不是[rs]，所以需要大改：

1. _Decoder 需要增加信号 ALUSrcA_Sel，并将原信号 ALUSrc 名称改为 ALUSrcB_Sel

2. MIPS.v 全面整改

(其实 ALU 新增端口改起来会更方便些，但是这样不够优雅)

<本次整改采用模拟考试的方式，即有条理不慌乱地按次序改正，纸笔记录全部改正过程，每一步操作务必留下证据，动手之前务必想清楚>

*

2. _FSU (将 Decoder 在_FSU 中实例化)

信号名	方向	功能描述
FSU_DInstr[31:0]	I	D_Instr
FSU_Einstr[31:0]	I	E_Instr
FSU_Minstr[31:0]	I	M_Instr
FSU_Winstr[31:0]	I	W_Instr
E_HILObusy	I	start | busy
stall	O	
FWMux_rsE_Sel	O	
FWMux_rtE_Sel	O	
FWMux_rsD_Sel	O	
FWMux_rtD_Sel	O	
FWMux_rtM_Sel	O	

附：Tuse/E\M_Tnew 赋值

信号名	方向	功能描述
Tuse_rs[3:0]	0	cal_R -> 1 cal_I -> 1 j_r -> 0 store -> 1 load -> 1 b_st -> 0 b_s -> 0 hilo_cal -> 1 hilo_mt -> 1 else -> 5(用不到)
Tuse_rt[3:0]	0	cal_R -> 1 store -> 2 b_st -> 0 hilo_cal -> 1 else -> 5(用不到)
E_Tnew[3:0]	0	cal_R -> 1 cal_I -> 1 j_I -> 0 load -> 2 cal_mf -> 1 else -> 0
M_Tnew[3:0]	0	cal_R -> 0 cal_I -> 0 j_I -> 0 load -> 1 else -> 0

单周期与流水线在周期内与上升沿工作状态不同:
单周期在第 n 个上升沿到来之际开始执行第(n+1)条指令;
流水线在第 n 个上升沿到来之际第 n 条指令进入 D 级

四、相较于 P5 的重大改动

*1. 新增 E_HILO 模块

**2. 更改 E-M 级数据通路

*a. 因为 E 级新增 HILO 部件，M_REG 输入源要发生改变;

*b. 除了增加多路选择器外，*名称*也要发生变化：E_ALUResult, E_HILOResult, E_Result,

*c. 转发源不要改;

*3. _FSU[-> mips]

*4. const

*5. _Decoder [-> mips(ok)]

*6. W_REG\M_REG[-> mips]

/***/

观察 roife_HILOUnit 暂停转发的行为。

在执行乘除法运算时是一律暂停，还是只有某些特定功能会受影响的指令暂停?

答：只有 hilo_cal\mf\mt 暂停，暂停时只是将值写入内置寄存器中，**不存在指令继续向前传递的问题**。

五、具体部件

Global	F	D	E	M	W
_Decoder		D_REG	E_REG	M_REG	W_REG
_FSU	F_IFU	D_GRF	E_ALU	M_DM	*D_GRF*
		D_EXT	E_HILO		
		D_NPC			
		D_Branchif			

六、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？

因为乘除法运算耗时较长，如果单独隔离出来，在 HILO 处于 busy 状态时，其他仅仅使用 ALU 其他运算功能的指令仍可正常运行，可以起到节省时间的作用。如果将 HI\LO 合并到 GRF 中，由于乘除法执行存在延时，在乘除法指令开始执行到产生结果这段时间内所有需要读取寄存器的指令都可能会出错。为解决这一问题，肯定要涉及出两个单独存储乘除法运算结果的寄存器；与其将两个寄存器放在 W\D 级的 GRF 中，倒不如直接归入 HILO 部件，计算出结果后直接存储。

2. 参照你对延迟槽的理解，试解释“乘除槽”。

在乘(除)法运算指令后分别加入 5(10)条无关指令，保证 CPU 正常流水，使效率最大化。

3. 举例说明并分析何时按字节访问内存相对于按字访问内存性能上更有优势。（Hint： 考虑 C 语言中字符串的情况）

因为一个字符占内存大小为 1 个字节，对字符个数不为 4 的整数倍字符串 or 单个字符进行存取操作时，按字节访问更为方便。

4. 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？

见后方测试板块。主要问题在于转发处理。

七、测试

1. store

```
li  $t0, 0x89abcdef
sw  $t0, 0($0)  # 0(0)
sb  $t0, 5($0)  # 1(4)
sh  $t0, 10($0) # 2(8)
sb  $t0, 15($0) # 3(12)

li  $t0, 0x89abcdef
sw  $t0, 16($0) # 0(16)
sb  $t0, 21($0) # 1(20)
sh  $t0, 26($0) # 2(24)
sb  $t0, 31($0) # 3(28)

li  $t1, 0x55555555
sw  $t1, 24($0) # 0(24)
sh  $t0, 26($0) # 2(24)
```

BUG_1_1:

```
`DMType_byte: begin
    DataMemory[waddr][8 * A[1 : 0] +: 8]  <=  WD[7:0];
        case(A[1:0])
        2'b00: begin
            $display("%d@%h: *%h <= %h", $time, PC, {{A[31 : 2]}, {2'b00}}, {{DataMemory[waddr][31 : 8]},{WD[7 : 0]}});
        end
        2'b01: begin
            $display("%d@%h: *%h <= %h", $time, PC, {{A[31 : 2]}, {2'b00}}, {{DataMemory[waddr][31 : 16]},{WD[7 : 0]},{DataMemory[waddr][7 : 0]}});
        end
        2'b10: begin
            $display("%d@%h: *%h <= %h", $time, PC, {{A[31 : 2]}, {2'b00}}, {{DataMemory[waddr][31 : 24]},{WD[7 : 0]},{DataMemory[waddr][15 : 0]}});
        end
        /*2'b11*/
        default: begin
            $display("%d@%h: *%h <= %h", $time, PC, {{A[31 : 2]}, {2'b00}}, {{WD[7:0]},{DataMemory[waddr][24 : 0]}});
        end
        endcase
```

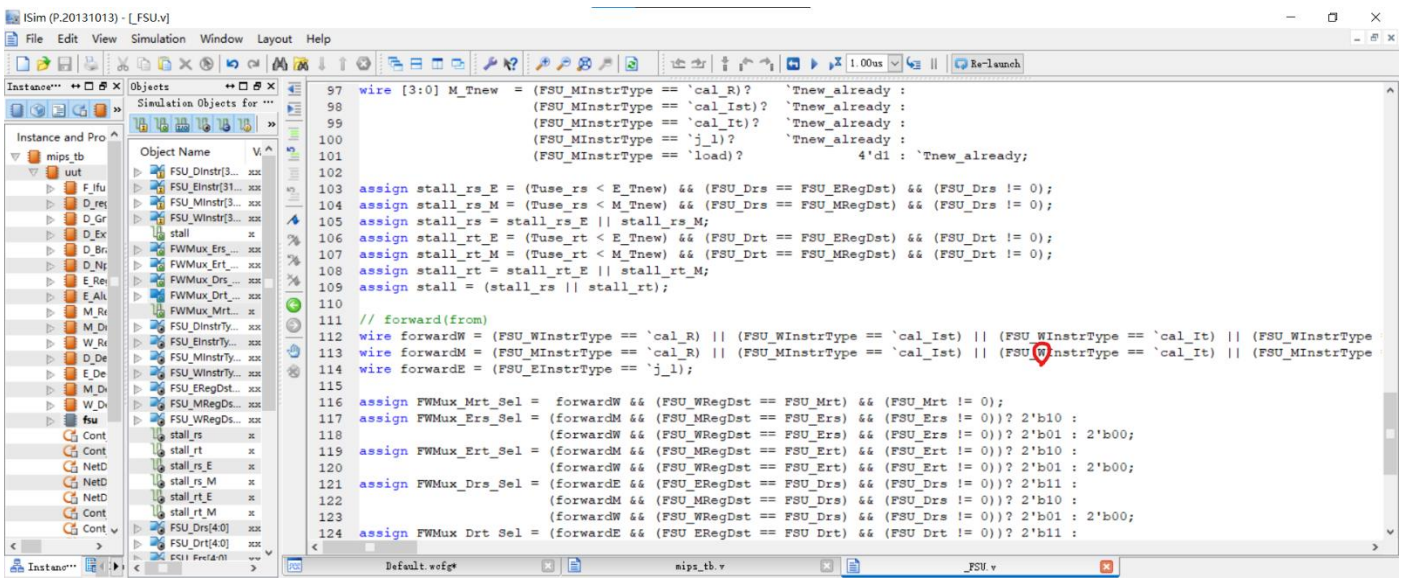
2.load

```
li  $t0, 0x89abcdef
sw  $t0, 0($0)  # 0(0)
sb  $t0, 5($0)  # 1(4)
sh  $t0, 10($0) # 2(8)
sb  $t0, 15($0) # 3(12)

li  $t0, 0x89abcdef
sw  $t0, 16($0) # 0(16)
sb  $t0, 21($0) # 1(20)
sh  $t0, 26($0) # 2(24)
sb  $t0, 31($0) # 3(28)

li  $t1, 0x55555555
sw  $t1, 24($0) # 0(24)
sh  $t0, 26($0) # 2(24)

lw  $t0, 0($0)
addu    $t0, $t0, $0    #stall (load, cal_R)
lhu $t2, 10($0) # 0x0000cdef
lh  $t3, 10($0) # 0xffffcdef
lb  $t4, 31($0) # 0xffffffef
lbu $t5, 31($0) # 0x000000ef
```

Result:

```
215@00003040: $ 8 <= 89abcdef
235@00003044: $ 8 <= 89abcdef
```

BUG_2_1:



3. bne_jal_j/4. lw

BUG_3_0:

```
155@0000302c: *0000001c <= ef000000
175@00003030: $ 1 <= 55550000
```

Q: Why stall?

A: 并没有暂停。

0x0000302c 对应指令在 M 级完成并输出（store）

0x00003030 对应指令在 W 级完成并输出（cal_lt）

BUG_4_1:

输出结果:

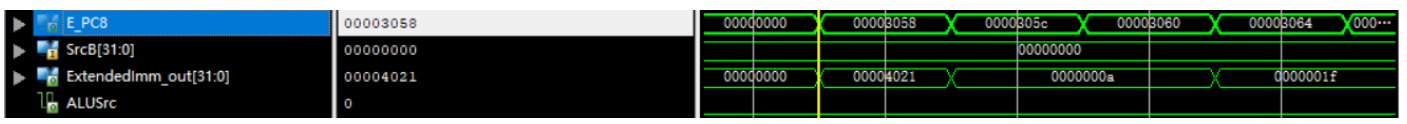
```
 55@00003000: $31 <= 00003008
 65@00003004: $ 1 <= 89ab0000
 75@0000301c: $ 1 <= 89ab0000
 85@00003020: $ 8 <= 89abcdef
 85@00003024: *00000010 <= 89abcdef
 95@00003028: *00000014 <= 0000ef00
105@0000302c: *00000018 <= cdef0000
115@00003030: *0000001c <= ef000000
145@00003038: $ 1 <= 55550000
155@00003048: $ 8 <= 00000000
175@0000304c: $ 8 <= 00000000
185@00003050: $10 <= 00000000
195@00003054: $11 <= 00000000
205@00003058: $12 <= 00000000
215@0000305c: $13 <= 00000000
ISim>
```

期待结果:

```
 55@00003000: $31 <= 00003008
 65@00003004: $ 1 <= 89ab0000
 75@0000301c: $ 1 <= 89ab0000
 85@00003020: $ 8 <= 89abcdef
 85@00003024: *00000010 <= 89abcdef
```

```
 95@00003028: *00000014 <= 0000ef00
105@0000302c: *00000018 <= cdef0000
115@00003030: *0000001c <= ef000000
155@0000304c: $ 8 <= 00000000
175@00003050: $ 8 <= 00000000
185@00003054: $10 <= 00000000
195@00003058: $11 <= 00000000
205@0000305c: $12 <= ffffffef
215@00003060: $13 <= 000000ef
ISim>
```

暂停功能正常，但最后两条 load 指令输出有误。
问题定位：E 级 Imm_or_rt 选择信号 ALUSrc 出了问题。
最后发现，在添加 lh/lb/lhu/lbu 等一系列指令时没有对 ALUSrc 赋值。。。
Jal 指令 Jump&Link 功能均无误。
Bne 指令顺序跳转无误。



跳转方向	Beq	Bne	Bgtz	Bltz	Bgez	Blez
顺序	1	1	1	1	0	0
逆序	1	1	1	1	0	0

5. bgez/blez

"bgez/blez $0, label"无法正常跳转
Assign BType 时出现小笔误。。。

```
// BType[2:0]
//D
assign BType        =     (is_beq)?               `BType_beq      :
                          (is_bne)?               `BType_bne      :
                          (is_bgtz)?              `BType_bgtz     :
                          (is_bltz)?              `BType_bltz     :
                          (is_bgez)?              `BType_bgez     :
                          (is_blez)?              `BType_blez     :       `BType_notb;
```

6. Jalr

将信号内聚时应注意信号的赋值独立性。
如采用多个单位信号耦合的赋值方式，则可以采用信号复用的方式，即同一指令可以符合 set 多个信号的特征，
e.g. jalr 可以将 j_r 与 j_l 两个信号同时置 1
然而若采用将全部信息内聚在单个多位信号的赋值方式的话，则需要将信号拆开，保证一条指令具有的全部特征仅对应其中一个信号。
e.g. `j_l `j_r `j_lr 三状态需分开设立。

```
assign InstrType    =  (is_addu || is_subu || is_add || is_sub)?          `cal_R    :
                       (is_ori)?                                          `cal_Ist  :
                       (is_j)?                                            `j_addr   :
                       (is_jal || is_jalr)?                               `j_l      :
                       (is_jr || is_jalr)?                                `j_r      :
                       (is_sw || is_sh || is_sb || is_swl || is_swr)?     `store    :
                       (is_lw || is_lhu || is_lh || is_lb || is_lbu)?     `load     :
                       (is_beq || is_bne)?                                `b_st     :
                       (is_blez || is_bgez || is_bltz || is_bgtz)?        `b_s      :
                       (is_nop)?                                          `nop      :
                       (is_lui)?                                          `cal_It   :    `InstrType_error;
```

7. HILO 系列指令

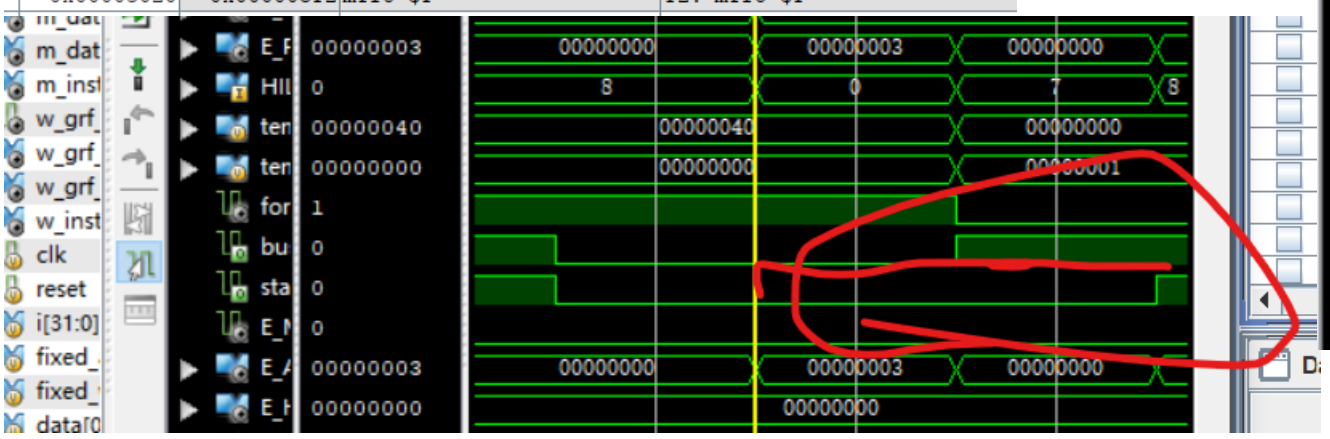
在执行 0x00003014 指令时出现问题

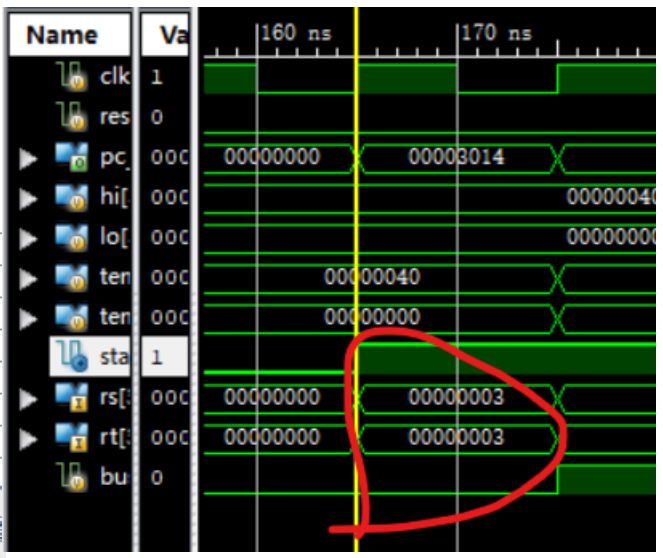
0x00003004: lui $1, 8 # $1 <= 0x00080000
0x00003008: xor # $3 <= 0
0x0000300c: mult # hi <= 0x00000040; lo <= 0x00000000
0x00003010: addiu # $3 <= 3
0x00003014: div # 行为在这里出现了错误

首先，通过对冲突的分析发现 div 指令进入 E 级寄存器时，下一条指令 mtlo 应该进入 D 级寄存器，此时的冲突无法通过转发解决，因此下一周期上升沿应暂停，这就意味着此时 stall 信号应该已经通过组合逻辑被置为 1。然而，实际上此时 stall 信号为 0，成功找到问题所在，暂停并未顺利进行。与此同时，调出 stall 信号赋值逻辑来源之一 busy 的值，发现 busy 对应值正确。这时，通过上下两个波形图的对比，我们突然发现，正确的 CPU 中 stall 较 busy 信号早产生一个周期，突然意识到在 HILO 内部信号 start 值为 1 的周期，即 busy 信号产生的前一周期，HILO 就已经投入使用，成功找到这只小臭虫!!!

```
assign HILObusy = start | busy;
```

0x00003008	0x00211826	xor $3, $1, $1	4: xor $3, $1, $1
0x0000300c	0x00210018	mult $1, $1	5: mult $1, $1
0x00003010	0x24630003	addiu $3, $3, 0x00000003	7: addiu $3, $3, 3
0x00003014	0x0063001a	div $3, $3	8: div $3, $3
0x00003018	0x00600013	mtlo $3	9: mtlo $3
0x0000301c	0x10630000	beq $3, $3, 0x00000000	10: beq $3, $3, TAG45
0x00003020	0x00000812	mflo $1	12: mflo $1





8. Forward

刚刚搭建好 CPU 模块时，进行基本转发功能测试时，

addiu　　$1, $0, 5

mult　　$1, $1

mfhi　　$1

add　　$2, $1, $0

以上测试程序导入 CPU 后测试结果有误。

原因是忘记进行乘除指令的转发调整，mfxx 系列指令有对寄存器的写入操作，必要时需要进行转发。

```
wire forwardW = (FSU_WInstrType == `load)    || (FSU_WInstrType == `shift_R) || (FSU_WInstrType == `cal_R)   || (FSU_WInstrType == `cal_Ist) || (FSU_WInstrType == `cal_It) || (FSU_WInstrType == `j_l)  ||
(FSU_WInstrType == `j_lr) || (FSU_WInstrType == `hilo_mf);
wire forwardM = (FSU_MInstrType == `shift_R) || (FSU_MInstrType == `cal_R)   || (FSU_MInstrType == `cal_Ist) || (FSU_MInstrType == `cal_It)  || (FSU_MInstrType == `j_l)    || (FSU_MInstrType == `j_lr) ||
(FSU_MInstrType == `hilo_mf);
wire forwardE = (FSU_EInstrType == `j_l)     || (FSU_EInstrType == `j_lr);
```

9. stall

```
wire [3:0] M_Tnew  = (FSU_MInstrType == `cal_R)?    `Tnew_already :
                     (FSU_MInstrType == `cal_Ist)?  `Tnew_already :
                     (FSU_MInstrType == `cal_It)?   `Tnew_already :
                     (FSU_MInstrType == `j_l)?      `Tnew_already :
                     (FSU_MInstrType == `j_lr)?     `Tnew_already :
                     (FSU_MInstrType == `shift_R)?  `Tnew_already :
                     (FSU_MInstrType == `load)?             4'd1 : `Tnew_already;
```

这是迄今为止我遇到的最难找的臭虫之一，一条笔误导致 M 级指令为`load，E 级指令恰好为 nop 时发生乱暂停的现象。

错误行为很明显，定位十分困难。