CPU 设计文档

一、 数据通路设计

1. 模块规格 datapath.v

1） 端口说明

表 1- 1 端口说明

信号	方向	描述
PauseF	I	
PauseD	I	
ClearE	I	
Forward_RS_D_src[2:0]	I	
Forward_RS_D_src[2:0]	I	
Forward_RS_D_src[1:0]	I	
Forward_RS_D_src[1:0]	I	
Forward_RS_D_src	I	
Clk	I	时钟信号
Reset	I	复位信号
IRD_out[31:0]	O	传入冲突控制的指令信号

2） 功能定义

表 1- 2 功能定义

功能	描述
grf 写入	若 grf 需要写入数据，则输出写入的位置及写入的值。
dm 写入	若 dm 需要写入数据，则输出写入的位置及写入的值。

2. 取指

1） 模块规格 F_unit.v

表 2- 1 端口说明

信号	方向	描述
NPC[31:0]	I	
PCsrc[1:0]	I	下一指令地址控制信号
Branch	I	是否跳转
Clk	I	时钟信号
Reset	I	复位信号

PauseF	I	暂停更新 PC
PauseD	I	暂停更新 D 级流水线寄存器
IRD[31:0]	O	当前指令
PC4D[31:0]	O	

2） 程序计数器 pc_F.v

表 2- 2 端口说明

信号	方向	描述
NPC[31:0]	I	下一指令地址/j 型和分支
Clk	I	时钟信号
Reset	I	复位信号
En	I	使能信号
add4	I	Pc+4 的值
j_r	I	跳转寄存器的指令地址
Branch	I	分支指令控制信号
PCsrc	I	下一指令控制
PC[31:0]	O	当前指令地址

表 2- 3 功能定义

功能	描述
存储指令地址	输出当前指令地址。时钟上升沿到来，且使能端有效时保存下一指令地址
复位	复位信号有效时，PC 同步复位于 0x0000_3000

3） 指令存储器 im.v

表 2- 4 端口说明

信号	方向	描述
addr[11:2]	I	指令地址
instr[31:0]	O	指令

表 2- 5 功能定义

功能	描述
存储指令	初始化时，加载指令码。容量为 32bit*1024
输出指令	输出地址信号对应的指令内容

3. 译码

1） 模块规格 D.v

表 3- 1 端口说明

信号	方向	描述
IRD[31:0]	I	当前指令

PC4D[31:0]	I	下一指令地址
PC4W	I	用于写寄存器时输出指令的地址
Clk	I	时钟信号
Reset	I	复位信号
ClearE	I	清空 E 流水级寄存器
Forward_RS_D_src[2:0]	I	转发 MUX 控制信号
Forward_RT_D_src[2:0]	I	转发 MUX 控制信号
PC4_forw_E[31:0]	I	
PC4_forw_M[31:0]	I	
AO[31:0]	I	
W_RF_WD_OUT[31:0]	I	
W_RF_A3_OUT[31:0]	I	
Branch	O	是否跳转
PCsrc[1:0]	O	
NPC[31:0]	O	跳转至地址
RS_D_F[31:0]	O	D 级读寄存器值-jr
IRE[31:0]	O	
PC4E[31:0]	O	
RSE[31:0]	O	
RTE[31:0]	O	
EXTE[31:0]	O	
We	I	

2） 寄存器文件 rf_D.v

表 3- 2 端口说明

信号	方向	描述
A1[4:0]	I	第一操作数地址
A2[4:0]	I	第二操作数地址
A3[4:0]	I	第三操作数地址
WD[31:0]	I	待写入数据
En	I	写使能信号
Clk	I	时钟信号
Reset	I	复位信号
RD1[31:0]	O	第一操作数
RD2[31:0]	O	第二操作数

表 3- 3 功能定义

功能	描述
复位	复位信号有效时，全部寄存器同步复位为 0

3） 扩展单元 ext_D.v

表 3- 4 端口说明

信号	方向	描述
EXTop[1:0]	I	扩展功能
In[15:0]	I	待扩展立即数
Out[31:0]	O	扩展结果

表 3- 5 功能定义

功能	ExtOp[1:0]	描述
{16'b0, Imm}	00	无符号扩展
{{16{Imm[15]}}, Imm}	01	有符号扩展
{Imm, 16'b0}	10	扩展至低位
32'b0	11	复位

4） 指令自增单元 npc_D.v

表 3- 6 端口说明

信号	方向	描述
PC4[31:0]	I	
I26[25:0]	I	指令的低 26 位
NPCsrc	I	控制信号
NPC[31:0]	O	

表 3- 7 功能定义

功能	NPCOp	描述
Branch	0	NPC = PC+4+{{14{I26[15]}}, I26[15:0], 2'b00}
Jump	1	NPC = {PC[31:28], I26[25:0], 2'b00}

5） 比较单元 cmp_D.v

表 3- 8 端口说明

信号	方向	描述
RD1[31:0]	I	第一个操作数
RD2[31:0]	I	第二个操作数
OPCode[5:0]	I	指令类型
Branch	O	是否跳转

4. 执行

1） 模块规格 E_unit.v

表 4- 1 端口说明

信号	方向	描述
IRE[31:0]	I	

PC4E[31:0]	I	
RSE[31:0]	I	
RTE[31:0]	I	
EXTE[31:0]	I	
Clk	I	时钟信号
Reset	I	复位信号
IRM[31:0]	O	
PC4M[31:0]	O	
AOM[31:0]	O	
RTM[31:0]	O	
Forward_RS_E_src[2:0]	I	转发 MUX 控制信号
Forward_RT_E_src[2:0]	I	转发 MUX 控制信号
AO[31:0]	I	
PC4_forw_M[31:0]	I	
W_RF_WD_OUT[31:0]	I	

2） 计算单元 alu_E.v

表 4- 2 端口说明

信号	方向	描述
SrcA[31:0]	I	A
SrcB[31:0]	I	B
ALUOp[2:0]	I	运算类型
Shift[4:0]	I	移位
ALUresult[31:0]	O	

表 4- 3 功能定义

功能	ALUOp[2:0]	描述
Nop	000	0
Add	001	A+B
Sub	010	A-B
And	011	A&B
Or	100	A|B
Xor	101	A^B
Le	110	A<B
B	111	B

3） 乘除模块 md_E.v

信号	方向	描述
D1[31:0]	I	A
D2[31:0]	I	B
opsrc[1:0]	I	运算类型

Clk	I	
Reset	I	
Start	I	运算驱动信号
Wsrc	I	写 LO,HI 信号
Highout	O	HI,LO 寄存器输出
lowout	O	
Busy	O	标记计算中的信号

5. 访存

1） 模块规格 M.v

表 5- 1 端口说明

信号	方向	描述
IRM[31:0]	I	
PC4M[31:0]	I	
AOM[31:0]	I	
RTM[31:0]	I	
Clk	I	时钟信号
Reset	I	复位信号
IRW[31:0]	O	
PC4W[31:0]	O	
AOW[31:0]	O	
DRW[31:0]	O	
Forward_RT_M_src[2:0]	I	转发 MUX 控制信号
W_RF_WD_OUT[31:0]	I	

2） 数据寄存器 dm.v

表 5- 2 端口说明

信号	方向	描述
Addr[11:2]	I	数据地址
WD[31:0]	I	待写入数据
WE	I	写使能信号
Clk	I	时钟信号
Reset	I	复位信号
LStype	I	存取类型
Instr	I	存取指令地址
RD[31:0]	O	数据

表 5- 3 功能定义

功能	描述
复位	复位信号有效时，dm 同步复位于 0x0000_0000

6. 写回

1） 模块规格 W_unit.v

表 6- 1 端口说明

信号	方向	描述
IRW[31:0]	I	
PC4W[31:0]	I	
AOW[31:0]	I	
DRW[31:0]	I	
W_RF_A3_OUT[4:0]	O	写回寄存器编号
W_RF_WD_OUT[31:0]	O	写回数据
RegWrite	O	

二、 控制器设计

1. 控制单元 ctrl.v

表 1- 1 端口说明

信号	方向	描述
OPCode[31:26]	I	
FunctCode[5:0]	I	
NPCsrc	O	下一指令地址来源
PCsrc[1:0]	O	指令跳转方式
ExtOp@D[1:0]	O	立即数扩展方式
ALUSrc@E	O	ALU 第二操作数来源
ALUOp@E[2:0]	O	ALU 运算方式
MemWrite@M	O	数据存储器写使能信号
RegDst@W[1:0]	O	待回写寄存器地址来源
MemtoReg@W[1:0]	O	待回写数据来源
RegWrite@W	O	寄存器写使能信号
LStype	O	主存读写方式
M_Dop	O	乘除模块的操作方式
MD_Start	O	乘除模块运算驱动信号
ALU_OUT	O	乘除和普通 alu 模块结果片选信号
MD_WE[1:0]	O	写 LO,HI 寄存器使能和信号

表 1- 2 R 型指令功能定义

Funct	000000	001000	001001	100001	100011	101010
OpCode	000000	000000	000000	000000	000000	000000

	nop	jr	jalr	addu	subu	slt
NPCsrc@F	00	11	11	00	00	00
PCsrc@D[1:0]	x	x	x	x	x	x
ExtOp@D[1:0]	xx	xx	xx	xx	xx	xx
ALUSrc@E	x	0	0	0	0	0
ALUOp@E[2:0]	xxx	100	100	001	010	110
MemWrite@M	0	0	0	0	0	0
RegDst@W[1:0]	xx	xx	10	00	00	00
MemtoReg@W[1:0]	xx	xx	10	00	00	00
RegWrite@W	0	0	1	1	1	1

表 1- 3 I 型指令功能定义

OpCode	000100	001001	001101	001111	100011	101011
	beq	addiu	ori	lui	lw	sw
NPCsrc@F	10	00	00	00	00	00
PCsrc@D[1:0]	0	x	x	x	x	x
ExtOp@D[1:0]	xx	01	00	10	01	01
ALUSrc@E	0	1	1	1	1	1
ALUOp@E[2:0]	010	001	100	111	001	001
MemWrite@M	0	0	0	0	0	1
RegDst@W[1:0]	xx	01	01	01	01	xx
MemtoReg@W[1:0]	xx	00	00	00	01	xx
RegWrite@W	0	1	1	1	1	0

表 1- 4 J 型指令功能定义

OpCode	000010	000011
	j	jal
NPCsrc@F	01	01
PCsrc@D[1:0]	1	1
ExtOp@D[1:0]	xx	xx
ALUSrc@E	x	x
ALUOp@E[2:0]	xxx	xxx
MemWrite@M	0	0
RegDst@W[1:0]	xx	10
MemtoReg@W[1:0]	xx	10
RegWrite@W	0	1

2. 冲突单元 Hazard_unit.v

表 2- 1 端口说明

信号	方向	描述
IRD[31:0]	I	D 级流水线寄存器指令

Clk	I	时钟信号
Reset	I	复位信号
PauseF	O	
PauseD	O	
ClearE	O	
Forward_RS_D_src[2:0]	O	
Forward_RT_D_src[2:0]	O	
Forward_RS_E_src[2:0]	O	
Forward_RT_E_src[2:0]	O	
Forward_RT_M_src[2:0]	O	

3. AT 解码器

表 3- 1 **端口说明**

信号	方向	描述
Instr[31:0]	I	D 级流水线寄存器指令
RS[4:0]	O	第一个读寄存器编号，如不需要则为 0
RT[4:0]	O	第二个读寄存器编号，如不需要则为 0
WR[4:0]	O	写寄存器编号，如不需要则为 0
Instype[3:0]	O	指令类型

4. 不确定写寄存器控制单元

信号	方向	描述
DJ_E	I	E 级流水线寄存器(D 信号)
DJ_M	O	M 级流水线寄存器(D 信号)
DJ_W	O	W 级流水线寄存器(D 信号)
EJ_M		
EJ_W		
MJ_W		

三、 测试程序

```
#the first part is about the cal_r stall:all about load instruction will
cause the stall.
ori $1,$0,-100
```

```
lui $2,100
add $1,$2,$1
sw $1,0($0)
lw $2,0($0)
addu $3,$2,$0#lw-addu-rs:stall
lw $3,0($0)
subu $4,$0,$3#lw-subu-rt:stall
lh $4,0($0)
add $5,$4,$0#lh-add-rs:stall
lhu $5,2($0)
sub $6,$0,$5#lhu-sub-rt:stall
lb $6,0($0)
and $7,$6,$3#lb-and-rs:stall
lb $7,1($0)
or $8,$2,$7#lb-or-rt:stall
lbu $8,2($0)
xor $9,$8,$1#lbu-xor-rs:stall
lbu $9,3($0)
nor $10,$1,$9#lbu-nor-rt:stall
lbu $11,3($0)
slt $12,$11,$9#lbu-slt-rs:stall
lhu $13,0($0)
sltu $14,$12,$13#lhu-sltu-rt:stall
#the next part is about the forward function:all about the
load-x-cal_r,cal_i-cal_r,shift-cal_r,jumpandlink-cal_r,mfhi/lo-cal_r
ori $1,$0,300
add $2,$1,$1#ori-add-rs,rt:forward
slt $3,$1,$2#add-slt-rt:forward
sltu $4,$3,$0#slt-sltu-rs:forward
ori $5,$0,-80
slt $6,$5,$0#ori-slt-rs:forward
ori $7,$0,-100
sltu $8,$7,$5#ori-sltu-rt:forward
slt $8,$7,$5
or $9,$7,$6
xor $10,$9,$9#or-xor-rs,rt:forward
nor $11,$10,$9#xor-nor-rs:forward
and $12,$11,$1#nor-and-rs:forward
sll $13,$7,5
or $14,$13,$0#sll-or-rs:forward
srl $15,$7,5
or $16,$0,$15#srl-or-rt:forward
sra $17,$5,5
or $18,$17,$0#sra-or-rs:forward
```

```
ori $19,$0,7
sllv $20,$5,$19
or $21,$20,$0#sllv-or-rs:forward
srlv $22,$5,$19
or $23,$22,$19#srlv-or-rs:forward
srav $24,$5,$19
or $25,$24,$0#srav-or-rs:forward
jal next1
nop
mult1:ori $1,$0,100
ori $2,$0,-100
mult $1,$2
mflo $3
add $3,$3,$5
jr $5
next1:addu $31,$31,$0#jal-nop-addu-rs:forward
sltu $30,$31,$0
jalr $5,$31
nop
end1:
#this part is mainly focus on the load-cal_i-rs type stall.
ori $1,$0,-100
lui $2,64
add $1,$2,$1
sw $1,0($0)
lw $2,0($0)
addi $3,$2,20#lw-addi-rs:stall
lh $3,0($0)
addiu $4,$3,-20#lh-addiu-rs:stall
lhu $4,2($0)
ori $5,$4,200#lhu-ori-rs:stall
lb $5,0($0)
xori $6,$5,74#lb-xori-rs:stall
lbu $6,0($0)
slti $7,$6,-20#lbu-slti-rs:stall
lbu $7,0($0)
sltiu $8,$7,-20#lbu-sltiu-rs:stall
lb $8,0($0)
slti $9,$8,0#lb-slti-rs:stall
lb $9,0($0)
sltiu $10,$9,0#lb-sltiu-rs:stall
#this part is mainly focus on the forward type.
lui $1,0xffff
addi $1,$1,1000#cal_i-cal_i-rs:forward
```

```
addiu $2,$1,0#addi-addiu-rs:forward
andi $3,$2,0xffff#addiu-andi-rs:forward
ori $4,$3,0#andi-ori-rs:forward
xori $5,$4,0xf0f0#ori-xori-rs-forward
ori $6,$0,-50
slti $7,$6,0#ori-slti-rs:forward
ori $6,$0,-90
sltiu $8,$6,0#ori-sltiu-rs:forward
addu $1,$1,$8
ori $1,$1,0#addu-ori:forward
lw $9,0($0)
nop
slti $9,$9,0#lw-otherinstruction-slti-ori:forward
ori $10,$0,-98
sll $11,$10,7
ori $12,$11,0#sll-ori-rs:forward
srl $13,$10,5
ori $14,$13,0#srl-ori-rs:forward
sra $15,$10,3
ori $16,$15,0#sra-ori-rs:forward
ori $17,$0,4
sllv $18,$10,$17
ori $18,$18,0#sllv-ori-rs:forward
srlv $19,$10,$17
ori $19,$19,0
srav $20,$10,$17
ori $20,$20,0#srav-ori-rs:forward
jal next2
nop
mult2:ori $21,$0,100
ori $22,$0,-23
divu $21,$22
addi $21,$21,1
mflo $23
subi $23,$23,1#mflo-subi-rs:forward.
mfhi $24
slti $25,$24,0#mfhi-slti-rs:forward
jr $5
nop
next2:addiu $31,$31,0#jal-nop-addiu-rs:forward
jalr $5,$31
nop
end2:
#the first part is mainly test the stall:load
```

```
ori $1,$0,4
sw $1,0($0)
ori $1,$0,100
sw $1,4($0)
lw $2,0($0)
lw $3,0($2)#lw-lw-rs:stall
lh $4,0($0)
lw $5,0($4)#lh-lw-rs:stall
lhu $6,0($0)
lw $7,0($6)#lhu-lw-rs:stall
lb $8,0($0)
lw $9,0($8)#lb-lw-rs:stall
lbu $10,0($0)
lw $11,0($10)#lbu-lw-rs:stall
#the second part is mainly test the forward
ori $1,$0,4
addu $2,$1,$0
lw $3,0($2)#addu-lw-rs:forward
addi $3,$0,4
lw $4,0($3)
ori $5,$0,1
sll $5,$5,2
lw $6,0($5)#shift-lw-rs:forward
mult3:jal next3
nop
ori $1,$0,4
ori $2,$0,1
multu $1,$2
mflo $3
lw $4,0($3)
jr $5
nop
next3:jalr $5,$31
nop
end3:
#the first part is focused on the stall.
ori $1,$0,4
sw $1,0($0)
ori $1,$0,0x5678
lui $2,0x1234
addu $1,$1,$2
sw $1,4($0)
lw $2,0($0)
sw $1,0($2)#lw-sw-rs:stall
```

```
lh $3,0($0)
sh $1,0($3)#lh-sh-rs:stall
lhu $4,0($0)
sh $1,2($4)#lhu-sh-rs:stall
lb $5,0($0)
sb $1,0($5)#lb-sb-rs:stall
lb $6,0($0)
sb $1,1($6)#lb-sb-rs:stall
#this part is mainly focused on the forward
li $30,0x12345678
ori $1,$0,4
addu $2,$0,$1
sw $30,0($2)#addu-sw-rs:forward
addu $30,$30,$1
sh $30,2($0)#addu-sw-rt:forward
li $30,0x12345678
sw $30,4($0)#ori-sw-rt:forward
addi $4,$0,5
sb $30,0($4)#ori-sb-rs:forward
ori $5,$0,1
sll $5,$5,2
sw $30,4($5)#sll-sw-rs:forward
sra $29,$30,2
sw $29,8($5)#sra-sw-rt:forward
mult4:jal next4
nop
sw $5,0($0)#jalr-nop-sw-rt:forward
ori $1,$0,8
ori $2,$0,2
div $1,$2
mflo $3
sw $31,0($3)#mflo-sw-rt:forward
ori $7,$0,29
ori $9,$0,30
mult $7,$9
mfhi $4
sw $4,4($3)#mfhi-sw-rs:forward
jr $5
nop
next4:
sw $31,0($0)#jal-nop-sw-rt:forward
jalr $5,$31
nop
end4:
```

```
#the first part is mainly focused on the shift instructon's
stall.->load-shift
ori $1,$0,-50
sw $1,0($0)
lw $2,0($0)
sll $3,$2,20#lw-sll-rs:forward
lh $3,0($0)
srl $4,$3,5#lw-srl-rs:stall
lb $4,0($0)
sra $5,$4,5
ori $1,$0,50
sw $1,4($0)
lw $5,4($0)
sra $5,$5,4#lw-sra-rs:stall
ori $1,$0,-50
sll $2,$1,6
addiu $1,$1,0xffff
srl $1,$1,7#addiu-srl-rs:forward
lui $1,0xffff
sra $1,$1,9#lui-sra-rs:forward
addiu $1,$1,200
sll $1,$1,5#addiu-sll-rs"forward
ori $1,$1,1
sll $1,$1,31#ori-sll-rs:forwaard
sra $1,$1,20
srl $1,$1,9#sra-srl-rs:forward
jal next5
nop
mult5:
ori $1,$0,0xffff
lui $2,0xffff
mult $1,$2
mfhi $3
sll $3,$3,4
mflo $4
sra $4,$4,7
multu $1,$2
mfhi $3
srl $3,$3,5
mflo $4
sra $4,$4,6
mthi $1
mtlo $2
mfhi $1
```

```
sll $1,$1,20
mflo $2
sra $2,$2,10
jr $5
nop
next5:sll $29,$31,16#jal-sll-rs
sra $28,$29,7
srl $27,$28,9
jalr $5,$31
nop
end5:
ori $1,$0,1
addiu $2,$0,20
for1:beq $1,$2,endfor1
sw $1,0($0)
addiu $1,$1,1
j for1
nop
endfor1:
ori $1,$0,3
addiu $2,$0,3
for2:bne $1,$2,endfor2
sb $2,4($2)
addiu $1,$1,1
mult $2,$2
mflo $2
j for2
nop
endfor2:
ori $2,$0,10
subu $3,$2,$0
for3:blez $3,endfor3
sw $3,0($0)
addiu $2,$2,-1
subu $3,$2,$0
j for3
nop
endfor3:
ori $2,$0,11
subu $3,$0,$2
for4:bgtz $3,endfor4
sw $3,8($0)
addiu $2,$2,-1
subu $3,$0,$2
```

```
j for4
nop
endfor4:
ori $2,$0,10
subu $3,$2,$0
for5:bltz $3,endfor5
sw $3,12($0)
addiu $2,$2,-1
subu $3,$2,$0
j for5
nop
endfor5:
ori $2,$0,11
subu $3,$0,$2
for6:bgez $3,endfor6
sw $3,0($0)
addiu $2,$2,1
subu $3,$2,$0
j for6
nop
endfor6:
ori $2,$0,11
subu $3,$0,$2
#for7:bgezal $3,endfor7
#sw $3,0($0)
#addiu $2,$2,1
#subu $3,$2,$0
#j for7
nop
endfor7:
ori $2,$1,-100
lui $3,123
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
```

```
addi $4,$4,1
mthi $4
mfhi $5
ori $1,$0,0x1234
sw $1,0($0)
lw $2,0($0)
mthi $2
mfhi $3
```

思考题：

第一节

1. 为什么需要独立的乘除模块，因为乘除算法需要阵列加减法，运算的速度较慢，周期较长，如果整合进 ALU，会导致写入 ALU 结果的重叠，即后续运算覆盖掉乘除运算的结果，而采用另一模块，可以实现指令并行，乘除运算的延迟不影响后续 alu 运算的进行。

2. 独立的 HI,LO 寄存器避免了整合进 GRF 的问题:会出现较为复杂的冲突，需要增加额外的冒险控制逻辑来实现对乘除法结果的保留和写入。而采用额外的两个寄存器，在 E 级处理过程中读数据或写数据，这样的好处是，我们可以不用考虑读写 HI,LO 寄存器的指令之间的关于这两个寄存器的数据冲突，而只关注这些指令与其他三十二个寄存器之间的冲突。因为引入 E 级读写，缩短了乘除法指令的流水级，对于存取 HI,LO 的指令来说，相当于只有 D 级和 W 级，这样就不需要对 HI,LO 转发了（寄存器内部转发）。之所以这样设计的理念是利用了指令功能的特别和单一性，对于操作 HI,LO 的指令来说只有 move 这一种功能，所以可以特化两个寄存器来简化这一类指令的执行。

第二节：

1. 以 C 语言字符串为例，在 C 中，每个字符类型变量占用一字节存储空间，unicode 编码占用两字节，在读取字符串中的某一个字符时，如果按照 lw 会取出这一字的存储空间的四字节数值，会增加后续片选的处理过程。当然，这个前提基于存储器按字节编址，而不是我们试验里用 32 位 reg 变量设计的存储器。

第三节：

我的流水线设计基于 ATdecoder 方法和暴力转发，即工程化方法，属于规划者模型，主要的抽象规范就是根据指令度写寄存器的功能对其进行分类，以便于计划控制逻辑和冲突解决的复杂度。

第四节：

其中，乘除类指令归为 cal_r 一类，mthi/mtlo 归为 cal_i 一类。

测试类型			cal_r	cal_i	b	load	store	jr	jalr
前序类型	冲突阶段	冲突寄存器	测试序列	测试序列	测试序列	测试序列	测试序列	测试序列	测试序列
cal_r	E	rs							
cal_r	E	rt							
cal_r	M	rs							
cal_r	M	rt							

cal_r	W	rs							
cal_r	W	rt							
cal_i	E	rs							
cal_i	E	rt							
cal_i	M	rs							
cal_i	M	rt							
cal_i	W	rs							
cal_i	W	rt							
load	E	rs							
load	E	rt							
load	M	rs							
load	M	rt							
load	W	rs							
load	W	rt							
jal	E	rs							
jal	E	rt							
jal	M	rs							
jal	M	rt							
jal	W	rs							
jal	W	rt							
jalr	E	rs							
jalr	E	rt							
jalr	M	rs							
jalr	M	rt							
jalr	W	rs							

除了部分不需要读第二个寄存器的指令以外，上表中每格基本都代表一种冲突。比如第一行第一格表示 cal_r 指令在 D 级时，若 E 级有 cal_r 指令，且后者 rd 值与前者 rs 值相等则会引发冲突。流水线多条指令冲突转发优先级的问题还没有测试全面。