Verilog 流水线 CPU 实验报告

一、CPU 设计方案综述

（一）总体设计概述

使用 Logisim 开发一个的 MIPS 流水线 PU，总体概述如下：

1. 此 CPU 为 32 位 CPU
2. 此 CPU 支持的指令集为：{addu, subu, ori, lw, sw, beq, lui,jal, jr, nop}
3. nop 机器码为 0x00000000
4. addu, subu 不支持溢出
5. 对于 b 类和 j 类指令， 流水线设计必须支持延迟槽
6. PC 的初始地址 0x0000

（二）关键模块定义

1.PC

（一）端口设置

表 1 PC 端口设置

序号	信号名	方向	描述
1	clk	I	时钟信号
2	PC[31:0]	I	当前 PC 值
3	reset	I	异步复位信号，将 PC 值置为 0x00003000 0：无效 1：复位
4	PC_en	I	PC 写入信号，判断是否冻结 PC 0：冻结 1：写入
5	NPC[31:0]	O	更新后 PC 的值，即下一条指令所在的地址

(二)功能定义

表 2 PC 功能定义

序号	功能	描述
1	存储指令的地址	保存当前执行指令在 IM 中的地址

2.IM

（一）端口设置

表 3 IM 端口设置

序号	信号名	方向	描述
1	PC[31:0]	I	当前 PC 值
2	Instr[31:0]	O	将 IM 中存储的指令输出

（二）功能定义

表 4 IM 功能定义

序号	功能	描述
1	输出指令	根据 PC 的值，取出 IM 中的指令

3. IFID

（一）端口定义

表 5-IDIF 功能定义

序号	信号名	方向	描述
1	clk	I	时钟信号
2	IFID_en	I	使能信号（反向暂停信号）
3	reset	I	同步复位信号
4	PCF[31:0]	I	PC 在 F 级的值
5	InstrF[31:0]	I	instr 在 F 级的值
6	PCD[31:0]	O	PC 在 D 级的值
7	InstrD[31:0]	O	instr 在 D 级的值

（二）功能定义

表 6-IFID 功能定义

序号	功能名称	功能描述

1	传递	当时钟信号处于上升沿且暂停信号无效时，指令和 PC 输出赋值为输入
2	清零	当重置信号有效时，将输出清零

4.NPC

（一）端口说明

表 7 NPC 端口说明

序号	信号名	方向	描述
1	jr	I	是否是 jr 指令
2	j_jal	I	是否是 j/jal 指令
3	beq	I	指令是不是 beq 0：不是 1：是
4	PCF[31:0]	I	F 级 PC 值
5	JrJump[31:0]	I	$ra 中储存的 PC 值，用于 jr 指令
6	Imm32[31:0]	I	beq 指令中 0-15 位的 32 位符号扩展
7	instr[31:0]	I	IM 中要执行的指令
8	PCD[31:0]	I	D 级 PC 值
9	NPC[31:0]	O	更新后的 PC 值，指向下一条应该执行的指令

（二）功能定义

表 8 NPC 功能定义

序号	功能	描述
1	更新 PC 的值	当 Zero 和 beq 皆为 1 时， PC = PC + 4 + imm32 * 4 当 jr 为 1 时 PC = PC_jr 当 j_jal 为 1 时 PC = {PC[31:28], Instr[25:0], 2'b00} 否则，PC = PC +4

5. GRF

（一）端口说明

表 9 GRF 端口说明

序号	信号名	方向	描述
1	A1[4:0]	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD1
2	A2[4:0]	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD2
3	A3[4:0]	I	5 位地址输入信号，指定 32 个寄存器中的一个，作为 RD 的写入地址
4	WD[31:0]	I	32 位写入到 A3 的数据
5	clk	I	时钟信号
6	clr	I	异步复位信号 0：无效 1：清零
7	GRF_WE	I	写使能信号 1：可向 GRF 中写入数据 0：不能向 GRF 中写入数据
8	RD1	O	输出 A1 指定的寄存器的 32 位数据
9	RD2	O	输出 A2 指定的寄存器的 32 位数据
10	PC[31:0]	I	题目要求，输出相应 PC 值

（二）功能定义

表 10 GRF 功能定义

序号	功能	描述
1	异步复位	reset 为 1 时，将所有寄存器清零
2	读数据	将 A1 和 A2 地址对应的寄存器的值分别通过 RD1 和 RD2 读出，读出$31 的值，方便执行 jr 指令

3	写数据	当 WE 为 1 且时钟上升沿来临时，将 WD 写入到 A3 对应的寄存器内部

6.CMP

(一)端口说明

表 11 CMP 端口说明

序号	信号名	方向	描述
1	A[31:0]	I	操作数 A
2	B[31:0]	I	操作数 B
3	Zero	O	A==B？

（二）功能描述

表 12 CMP 功能定义

序号	功能	描述
1	判断 A 和 B 是否相等	若 A 等于 B，Zero=1，否则置零

7.EXT

（一）端口说明

表 13 EXT 端口说明

序号	信号名	方向	描述
1	imm16[15:0]	I	代扩展的 16 位信号
2	EXTOp	I	无符号或符号扩展选择信号 0：无符号扩展 1：符号扩展
3	imm32[31:0]	O	扩展后的 32 位的信号

（二）功能定义

表 14 EXT 功能定义

序号	功能	描述
1	无符号扩展	当 EXTOp 为 0 时，将 imm16 无符号扩展输出
2	符号扩展	当 EXTOp 为 1 时，将 imm16 符号扩展输出

8.IDEX

(一)端口说明

表 15 IDEX 端口说明

序号	信号名	方向	描述
1	reset	I	同步复位信号
2	flush	I	清除信号，用于暂缓
3	clk	I	时钟信号
4	RD1D[4:0]	I	RD1 在 D 级的值
5	RD2D[4:0]	I	RD2 在 D 级的值
6	InstrD[31:0]	I	Instr 在 D 级的值
7	MemtoRegD[1:0]	I	MemtoReg 在 D 级的值
8	MemWriteD	I	MemWritre 在 D 级的值
9	A3D[4:0]	I	A3 在 D 级的值
10	WDD[31:0]	I	WD 在 D 级的值
11	ALUOpD[2:0]	I	ALUOp 在 D 级的值
12	ALUSrcD	I	ALUSrc 在 D 级的值
13	RD1E[4:0]	O	RD1 在 E 级的值
14	RD2E[4:0]	O	RD2 在 E 级的值
15	InstrE[31:0]	O	Instr 在 E 级的值
16	MemtoRegE[1:0]	O	MemtoReg 在 E 级的值
17	MemWriteE	O	MemWritre 在 E 级的值
18	A3E[4:0]	O	A3 在 E 级的值
19	WDE[31:0]	O	WD 在 E 级的值
20	ALUOpE[2:0]	O	ALUOp 在 E 级的值
21	ALUSrcE	O	ALUSrc 在 E 级的值

（二）功能定义

表 16-IFID 功能定义

序号	功能名称	功能描述

1	传递	当时钟信号处于上升沿且暂停信号无效时，指令和 PC 输出赋值为输入
2	清零	当重置信号有效时，将输出清零
3	冲刷	当暂缓信号有效时，将输出清零

9. ALU

（一）端口说明

表 17 ALU 端口说明

序号	信号名	方向	功能描述
1	SrA[3:0]	I	与运算的第一个数
2	SrB[31:0]	I	与运算的第二个数
3	ALUOp[2:0]	I	决定 ALU 做何种操作 000：无符号加 001：无符号减 010：或 011:LUI（加载至最高位）
4	Result[31:0]	O	运算的结果
5	shamt[4:0]	I	传递的位移数

（二）功能定义

表 18 ALU 功能定义

序号	功能	描述
1	加运算	Ans = A + B
2	减运算	Ans = A - B
3	或运算	Ans = A | B
4	LUI’运算	Ans’ = imm16||0^{16}

10.EXMEM

（一）端口说明

表 19 EXMEM 端口说明

序号	信号名	方向	描述
1	reset	I	同步复位信号
2	PCE[31:0]	I	PC 在 E 级的值
3	clk	I	时钟信号
4	InstrE31:0]	I	Instr 在 E 的值
5	MemtoRegE[1:0]	I	MemtoReg 在 E 级的值
6	MemWriteE	I	MemWritre 在 E 级的值
7	A3E[4:0]	I	A3 在 E 级的值
8	WDE[31:0]	I	WD 在 E 级的值
9	ResE[31:0]	I	Res 在 E 的值
10	imm32E[31:0]	I	imm32 在 E 值
11	InstrM31:0]	O	Instr 在 M 级的值
12	MemtoRegEM[1:0]	O	MemtoReg 在 E 级的值
13	MemWriteM	O	MemWritre 在 E 级的值
14	A3M[4:0]	O	A3 在 M 级的值
15	WDM[31:0]	O	WD 在 M 级的值
16	ResM[31:0]	O	ResM 在 M 级的值
17	PCM31:0]	O	PC 在 M 的值
18	imm32M:0]	O	imm32 在 M 的值

（二）功能定义

表 20FID 功能定义

序号	功能名称	功能描述
1	传递	当时钟信号处于上升沿且暂停信号无效时，指令和 PC 输出赋值为输入
2	清零	当重置信号有效时，将输出清零

10.DM

（一）端口说明

表 19 DM 端口说明

序号	信号名	方向	描述
1	clk	I	时钟信号
2	reset	I	异步复位信号 0：无效 1：内存值全部清零
3	MemWrite	I	写使能信号 0：禁止写入 1：允许写入
4	Aaddress[31:0]	I	读取或写入信号地址
5	WD[31:0]	I	32 位写入数据
6	Sel[1:0]	I	读取/写入模式 00：字 01：字节 10：半字
7	RD[31:0]	O	32 位读出数据

（二）功能定义

表 20 DM 功能定义

序号	功能	描述
1	异步复位	当 reset 为 1 时，DM 中所有数据清零
2	写入数据	当 MemWrite 有效时，时钟上升沿来临时，WD 中数据写入 Address 对应的 DM 地址中
3	读出数据	RD 永远读出 Address 对应的 DM 地址中的值

11.MEWB

(一)功能定义

用于存储连接 M 级和 W 级

12.Controller

（一）端口说明

表 23 Controller 端口说明

序号	信号名	方向	描述
1	Insre[5:0]	I	IM 中的指令
2	jr	O	instr 是否为 jr 信号 0：不是 1：是
3	ALUOp[2:0]	O	ALU 的控制信号 000：无符号加 001：无符号减 010：或 011:LUI（加载至最高位）
4	RegWrite	O	GRF 写使能信号 0：禁止写入 1：允许写入
5	MemWrite	O	DM 的写入信号 0：禁止写入 1：允许写入
6	beq	O	instr 是否为 beq 信号 0：不是 1：是
7	AluSrc	O	参与 ALU 运算的第二个数，来自 GRF 还是 imm 0：来自 GRF 1：imm
8	WhichtoReg[1:0]	O	将何种数据写入 GRF？ 00：ALU 计算结果 01：DM 读出信号 10：PC+4
9	RegDst[1:0]	O	GRF 写入地址选择信号 00：Rd

			01：Rt 10：$r31
10	J_jal	O	instr 是否为 j/jal 信号 0：不是 1：是
11	SignExt	O	对 imm16 进行扩展的方式 0：0 扩展 1：符号扩展

（二）功能定义

表 24　Controller 功能定义

序号	功能	描述
1	判断指令类型	根据 op 和 func，判断 instr[31:0]具体是哪条指令
2	转换控制信号	利用数据通路，分析对应指令下哪些信号的选择

（三）真值表

表 17 Controller 对应的真值表

端口	addu	subu	ori	lw	sw	lui	beq	j	jal	jr
op	000000	000000	001101	100011	101011	001111	000100	000010	000011	000000
func	100001	100011								001000
AluOp	000	001	010	000	000	011	000	000	000	000
RegWrite	1	1	1	1	0	1	0	0	1	0
MemWrite	0	0	0	0	1	0	0	0	0	0
beq	0	0	0	0	0	0	1	0	0	0
ALUSrc	0	0	1	1	1	0	0	0	0	0
WhichtoReg	00	00	00	01	00	00	00	00	10	00
RegDst	00	00	01	01	01	01	01	00	10	00
SignExt	0	0	0	1	1	0	1	0	0	0
j_jal	0	0	0	0	0	0	0	1	1	0
jr	0	0	0	0	0	0	0	0	0	1

（三）重要机制实现方法

1. 转发机制

采用标记转发法，即每一级中，对于会使 GRF 的产生变化的指令达标记

```
assign FlagE = (InstrE[`op]==`JAL);

assign FlagM =
(InstrM[`op]==`JAL)||(InstrM[`op]==`LUI)||(InstrM[`op]==`
R&&InstrM[`func]==`ADDU)||(InstrM[`op]==`R&&InstrM[`func]
==`SUBU)||(InstrM[`op]==`ORI);

assign FlagW =
(InstrW[`op]==`JAL)||(InstrW[`op]==`LUI)||(InstrW[`op]==`
R&&InstrW[`func]==`ADDU)||(InstrW[`op]==`R&&InstrW[`func]
==`SUBU)||(InstrW[`op]==`ORI)||(InstrW[`op]==`LW);
```

同时需要为他们建立地址和数据来源的编码信息，方便转发

地址编码：

```
assign addrE =(InstrE[`op]==`JAL) ? 5'd31 : 5'd0;

assign addrM = (InstrM[`op]==`JAL) ? 5'd31:
(InstrM[`op]==`LUI || InstrM[`op] == `ORI)?InstrM[`rt]:
(InstrM[`op]==`R && (InstrM[`func]==`ADDU ||
InstrM[`func]==`SUBU ))?InstrM[`rd]:5'd0;
```

数据选择编码：

```
assign DataM= (InstrM[`op]==`JAL)? 2'd1:2'd0; //1 for
PC+8, 0 for ALU Result
```

因为对 GRF 采用了内部转发的机制，所以无需对 M/W 级寄存器，而 E 级可以

转发的只有 JAL，因此特判以下就好了

内部转发机制：

```
assign RD1 = (A1 == A3 && A1 != 0)? WD:Reg[A1];
assign RD2 = (A2 == A3 && A2 != 0)? WD:Reg[A2];
```

转发数据的确定：

```
assign FdataE = PCE+8;
assign FdataM = (FdataselM == 2'd0)? ResM:PCM+8;
```

接下来分析需要转发的位点。当某一部件需要使用 GPR 中的值时，如果此时这个值存在于后续某个流水线寄存器中，而还没来得及写入 GPR，我们就需要通过转发（旁路）机制将这个值从流水线寄存器中送到该部件的输入处。

根据我们对数据通路的分析，这样的位点有：

1．D 级比较器的两个输入（含 NPC 逻辑中寄存器值的输入）；

2．E 级 ALU 的两个输入；

3．M 级 DM 的输入。

对应的就有以下数据的选择

```
//Forward for the CMP
assign RD1D = (InstrD[`rs]==0)              ?0:
            (InstrD[`rs]==FaddrE && FflagE)?FdataE:
            (InstrD[`rs]==FaddrM && FflagM)?FdataM:
            (InstrD[`rs]==A3W && FflagW)   ?WDW:
                                            RD1;
assign RD2D = (InstrD[`rt]==0)              ?0:
              (InstrD[`rt]==FaddrE && FflagE)?FdataE:
              (InstrD[`rt]==FaddrM && FflagM)?FdataM:
              (InstrD[`rt]==A3W && FflagW)    ?WDW:
```

```
                                                    RD2;
//forward before the ALU
assign RD1BALUE = (InstrE[`rs]==0)                ?0:
                   (InstrE[`rs]==FaddrM &&FflagM)?FdataM:
                   (InstrE[`rs]==A3W && FflagW)   ?WDW:
                                                   RD1E;
assign RD2BALUE = (InstrE[`rt]==0)    ?0:
                  (InstrE[`rt]==FaddrM && FflagM)?FdataM:
                  (InstrE[`rt]==A3W && FflagW)    ?WDW:
                                                    RD2E;
//forward before the DM
assign WDBDMM = (InstrM[`rt] == 0) ?           0:
                (InstrM[`rt] == A3W && FflagW) ? WDW:
                                                 WDMemoryM;
```

（2）暂停机制

按照 T_{use} 和 T_{new}，对每个进入 D 阶段的指令以及正在运行执行的指令进行判断，以指令为导向，判断是否应该暂停

总结得到下表：

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

因此可以得到以下暂停表达式的构造

```
 //b_type
assign Stall_b = beq_D &&((Cal_r_E &&
(InstrE[`rd]==InstrD[`rs]||InstrE[`rd]==InstrD[`rt]))||(C
al_i_E &&
(InstrE[`rt]==InstrD[`rs]||InstrE[`rt]==InstrD[`rt]))||(L
oad_E &&
(InstrE[`rt]==InstrD[`rs]||InstrE[`rt]==InstrD[`rt]))||(L
oad_M &&
(InstrM[`rt]==InstrD[`rs]||InstrM[`rt]==InstrD[`rt])));
   //calculate_r
    assign Stall_r = Cal_r_D &&((Load_E &&
(InstrE[`rt]==InstrD[`rs]||InstrE[`rt]==InstrD[`rt])));
   //calculate_i
    assign Stall_i = Cal_i_D &&((Load_E &&
InstrE[`rt]==InstrD[`rs]));
```

```
   //load
    assign Stall_l = Load_D &&((Load_E &&
InstrE[`rt]==InstrD[`rs]));
   //store
    assign Stall_s = Store_D &&((Load_E &&
InstrE[`rt]==InstrD[`rs]));
   //jr
    assign Stall_jr = jr_D &&((Cal_r_E &&
InstrE[`rd]==InstrD[`rs])||(Cal_i_E &&
InstrE[`rt]==InstrD[`rs])||(Load_E &&
InstrE[`rt]==InstrD[`rs])||(Load_M &&
InstrM[`rt]==InstrD[`rs]));

assign
Stall=Stall_b|Stall_r|Stall_i|Stall_l|Stall_s|Stall_jr;
```

二、测试方案

（一）典型测试样例

（1）测试代码

```
   ori $a0,$0,1999
   ori $a1,$a0,111
   lui $a2,12345
   lui $a3,0xffff
   lui $t0,0xffff
   beq $a3,$t0,eee
   addu $s7,$0,$a0
   nop
   ori $a3,$a3,0xffff
   addu $s0,$a0,$a1
```

```
addu $s1,$a3,$a3
addu $s2,$a3,$s0
beq $s2,$s3,eee
subu $s0,$a0,$s2
subu $s1,$a3,$a3
eee:
subu $s2,$a3,$a0
subu $s3,$s2,$s1
ori $t0,$0,0x0000
sw $a0,0($t0)
nop
sw $a1,4($t0)
sw $s0,8($t0)
sw $s1,12($t0)
sw $s2,16($t0)
sw $s5,20($t0)
lw $t1,20($t0)
lw $t7,0($t0)
lw $t6,20($t0)
sw $t6,24($t0)
lw $t5,12($t0)
jal end
ori $t0,$t0,1
ori $t1,$t1,1
ori $t2,$t2,2
beq $t0,$t2,eee
lui $t3,1111
jal out
end:
addu $t0,$t0,$t7
```

```
jr $ra
out:
addu $t0,$t0,$t3
ori $t2,$t0,0
beq $t0,$t2,qqq
lui $v0,10
qqq:
lui $v0,11
j www
nop
www:
lui $ra,100
```

（2）标准输出结果

```
@00003000: $ 4 <= 000007cf
@00003004: $ 5 <= 000007ef
@00003008: $ 6 <= 30390000
@0000300c: $ 7 <= ffff0000
@00003010: $ 8 <= ffff0000
@00003018: $23 <= 000007cf
@0000303c: $18 <= fffef831
@00003040: $19 <= fffef831
@00003044: $ 8 <= 00000000
@00003048: *00000000 <= 000007cf
@00003050: *00000004 <= 000007ef
@00003054: *00000008 <= 00000000
@00003058: *0000000c <= 00000000
@0000305c: *00000010 <= fffef831
@00003060: *00000014 <= 00000000
@00003064: $ 9 <= 00000000
```

@00003068: $15 <= 000007cf

@0000306c: $14 <= 00000000

@00003070: *00000018 <= 00000000

@00003074: $13 <= 00000000

@00003078: $31 <= 00003080

@0000307c: $ 8 <= 00000001

@00003094: $ 8 <= 000007d0

@0000309c: $ 8 <= 000007d0

@00003080: $ 9 <= 00000001

@00003084: $10 <= 00000002

@0000308c: $11 <= 04570000

@00003090: $31 <= 00003098

@00003094: $ 8 <= 00000f9f

@0000309c: $ 8 <= 04570f9f

@000030a0: $10 <= 04570f9f

@000030a8: $ 2 <= 000a0000

@000030ac: $ 2 <= 000b0000

@000030b8: $31 <= 00640000

（3）搭建的 CPU 运行结果

9@00003000: $ 4 <= 000007cf

11@00003004: $ 5 <= 000007ef

13@00003008: $ 6 <= 30390000

15@0000300c: $ 7 <= ffff0000

17@00003010: $ 8 <= ffff0000

23@00003018: $23 <= 000007cf

25@0000303c: $18 <= fffef831

27@00003040: $19 <= fffef831

29@00003044: $ 8 <= 00000000

29@00003048: *00000000 <= 000007cf

33@00003050: *00000004 <= 000007ef

35@00003054: *00000008 <= 00000000

37@00003058: *0000000c <= 00000000

39@0000305c: *00000010 <= fffef831

41@00003060: *00000014 <= 00000000

45@00003064: $ 9 <= 00000000

47@00003068: $15 <= 000007cf

49@0000306c: $14 <= 00000000

49@00003070: *00000018 <= 00000000

53@00003074: $13 <= 00000000

55@00003078: $31 <= 00003080

57@0000307c: $ 8 <= 00000001

59@00003094: $ 8 <= 000007d0

63@0000309c: $ 8 <= 000007d0

65@00003080: $ 9 <= 00000001

67@00003084: $10 <= 00000002

73@0000308c: $11 <= 04570000

75@00003090: $31 <= 00003098

77@00003094: $ 8 <= 00000f9f

79@0000309c: $ 8 <= 04570f9f

81@000030a0: $10 <= 04570f9f

87@000030a8: $ 2 <= 000a0000

89@000030ac: $ 2 <= 000b0000

95@000030b8: $31 <= 00640000

因为时钟周期 2s, 所以总共经过了 48 个 cycle

由计组提供的辅助测试中

```
standard pipeline-cycle: 46
    slow pipeline-cycle: 76
  accepted cycle range: [40, 67]
```

满足条件

三、思考题

（一）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。

数据通路：

NPC 为 PC+4、branch 型指令、Jr 跳转和 J 型指令。

控制信号：

controller 解码出应当选择的 NPC 值即可。

（二）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？？

Jal 有延迟槽，无论怎样 PC+4 都会执行，因此有效的 PC 值为 PC+8

（三）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

转发的目的是：若与前面冲突，算出来后，才转发，否则暂停。这是使用“暂停-转发”策略下的核心思想。如果由部件提供，一个周期内各部件运行的功能会有一定程度上的紊乱，使控制变得复杂

（四）转发旁路机制构造的 Thing1-4

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。

转发的优先级。以 jr 为例，在 D 级就要产生写入的 NPC 数据，此时若 E 级和 M 级要求写入同一个寄存器，导致数据冲突，则需要选择 E 级流水线的数

据。M 级更新的数据也会被 E 级覆盖。

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调用的 rs，rt 寄存器出现错误。

如果不采用内部转发机制，可以在转发机制中考虑 W 级转发，即如果先前给 E 级，M 级 data 类型和 addr 类型编码信号一样，对流入 W 级寄存器的信号进行转发

Thinking 3：为什么 0 号寄存器需要特殊处理？

0 号寄存器恒为 0，不会有其他值，不参与转发

Thinking 4：什么是“最新产生的数据”？

通过看转发处距所需要数据的组件的距离远近来判断，距离所需要的地方越近，则越新。

（五）在 AT，即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？

如果不需要写寄存器，直接将 A 译码为为 0，这样甚至可以直接省略 we。

（五）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？？

见测试样例与转发暂停机制

（五）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。

按照教程的说法，将指令分类，枚举两条相关的指令，因为 P5 十条指令，可以把每个指令都测试一下。

如转发机制下，以 D 级 rs 为主体，分别检测其与 E 级（jal），M 级（r 型计算，i 型计算，jal, lui），W 级（r 型 cal，i 型 cal, jal, lui, lw），同理构造 E 级 rs rt 等

根据上述的表格也可以构建出需要测试的暂停机制

D 级为 Beq_rs/rt：E 级 cal_r_rd，E 级 cal_i_rt，E 级 load_rt，M 级 load_rt

D 级为 Cal_r_rs/rt：E 级 load_rt

D 级为 Cal_i_rs E 级 load_rt

D 级为 load_rs：E 级 load_rt

D 级为 store_rs：E 级 load_rt

D 级为 jr_rs：D 级为 Beq_rs/rt：E 级 cal_r_rd，E 级 cal_i_rt，E 级 load_rt，M 级 load_rt

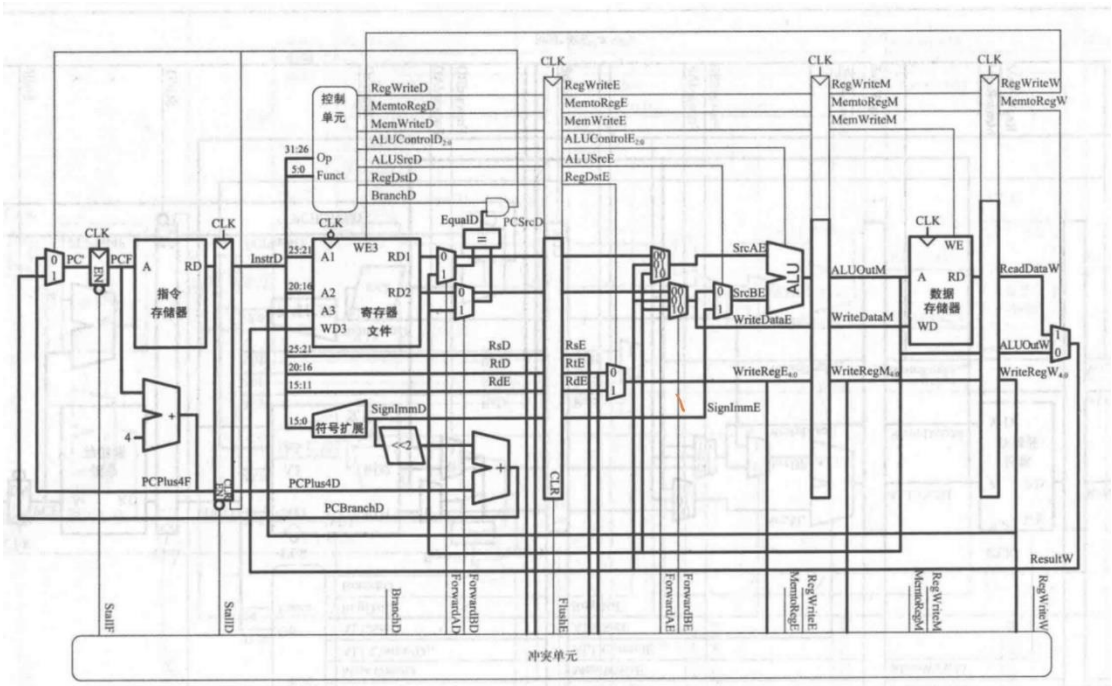

图7-58　处理所有冲突的流水线处理器