计算机组成原理实验报告

20231164 张岳霖

一、CPU 设计方案综述

（一）总体设计概述

本 CPU 为 Verilog 实现的流水线 MIPS - CPU，共包含 F、D、E、M 和 W 五级，支持的指令集包含{add、addu、sub、subu、slt、sltu、and、or、nor、xor、sllv、srlv、srav、sll、srl、sra、jr、jalr、mfhi、mflo、mthi、mtlo、mult、multu、div、divu、addi、addiu、slti、sltiu、andi、ori、xori、lui、sw、sh、sb、lw、lh、lhu、lb、lbu、beq、bne、bgtz、blez、bgez、bltz、j、jal、madd、maddu、msub、msubu}共 54 条，其中 add、addi 和 sub 不考虑因溢出而产生的异常。为了实现这些功能，CPU 主要包含了 PC、FlowReg_D、RF、NPC、CMP、EXT、FlowReg_E、ALU、multdiv、FlowReg_M、FlowReg_W、controller、RiskSolveUnit，共 13 个模块，指令存储器和数据存储器外置。其中 F 级包含 PC 一个模块；D 级包含 RF、NPC、CMP、EXT 四个模块；E 级包含 ALU、multdiv 两个模块；M 级不包含小模块。在实现过程中，将 D、M、E、W 各级分别作为一个独立的模块，数据只能在相邻模块之间传递，并在顶层模块文件 mips.v 中连接各流水线寄存器和模块。

（二）关键模块定义

1. PC

模块功能：程序计数器

模块定义：

```
module PC (
    input  wire clk,
    input  wire reset,
    input  wire pcenable,
    input  wire [2:0] npcop,
```

```
    input  wire [31:0] pc_D_B,
    input  wire [31:0] pc_D_JJal,
    input  wire [31:0] pc_D_Jr,
    input  wire branch,
    output reg  [31:0] currentpc_F
);
```

表一 PC 信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号，时钟上升沿更新当前指令地址为下一条指令地址
2	reset	I	1	同步复位信号，信号为 1 时将指令初始化为 0x0000_3000
3	pcenable	I	1	使能信号，为 0 时 PC 寄存器保持原有值不变，用于阻塞。
4	npcop	I	3	下一条指令计算方法选择信号
5	PC_D_B	I	32	B 类指令的地址
6	PC_D_JJal	I	32	J 和 Jal 指令的跳转地址
7	PC_D_Jr	I	32	Jr 指令的跳转地址
8	branch	I	1	B 类指令满足跳转条件信号，满足条件时信号为 1
9	currentpc_F	O	32	输入到指令寄存器和 D 级流水线寄存器的地址信号

npcop 选择信号对应表

npcop	对应的 npc 计算方法
3’d0	PC+4
3’d1	PC_B
3’d2	PC_JJal
3’d3	PC_Jr

2. FlowReg_D

模块功能：D 级流水线寄存器

模块定义：

```
module FlowReg_D (
    input  wire clk,
    input  wire En_D,
    input  wire reset,
    input  wire clear_D,  // 清空延迟槽信号
    input  wire [31:0] command_F,
    input  wire [31:0] commandAddr_F,
    output reg [31:0] command_D,
    output reg [31:0] commandAddr_D
);
```

表二 D 级流水线寄存器信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号
2	En_D	I	1	D 级寄存器使能
3	reset	I	1	D 级寄存器复位信号
4	clear_D	I	1	D 级寄存器复位信号，

				用于清空延迟槽
5	command_F	I	32	F 级指令信号
6	commandAddr_F	I	32	F 级当前指令地址
7	Command_D	O	32	D 级指令信号
8	commandAddr_D	O	32	D 级当前指令地址

3. RF

模块功能：寄存器堆

模块定义：

```
module RF (
    input  wire clk,
    input  wire reset,
    input  wire [4:0] a1,
    input  wire [4:0] a2,
    input  wire [4:0] a3,
    input  wire rfwe,
    input  wire [31:0] rfwd,
    input  wire [31:0] commandAddr_W,
    output wire [31:0] rfrd1,
    output wire [31:0] rfrd2
);
```

表三 RF 信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位信号,信号为 1 将 32 个寄存器全部清零
3	a1	I	5	地址输入信号，指定 32 个寄

				存器中的某一个，将其中存储的数据读入到 rfrd1
4	a2	I	5	地址输入信号，指定 32 个寄存器中的某一个，将其中存储的数据读入到 rfrd2
5	a3	I	5	地址输入信号，指定 32 个寄存器中的某一个作为写入的目标寄存器
6	rfwe	I	1	GRF 写入使能信号，1 写入有效
7	rfwd	I	32	32 位数据输入信号
8	commandAddr_W	I	32	W 级当前指令地址，评测需要
9	rfrd1	O	32	输出 A1 指定的寄存器中的 32 位数据
10	rfrd2	O	32	输出 A2 指定的寄存器中的 32 位数据

注 RF 采用内部转发，W 级写入 RF 的数据对 D 级可见。

实现方法为：

```
assign rfrd1 = (a1 == a3 && a3 != 5'd0 && rfwe == 1'd1) ? rfwd : register[a1];

assign rfrd2 = (a2 == a3 && a3 != 5'd0 && rfwe == 1'd1) ? rfwd : register[a2];
```

4. NPC

模块功能：下一指令地址计算模块

模块定义：

```
module NPC (
    input  wire [31:0] commandAddr_D,
```

```
    input  wire [31:0] command_D,
    input  wire [31:0] rfrd1_D,
    output wire [31:0] pc_D_B,
    output wire [31:0] pc_D_JJal,
    output wire [31:0] pc_D_Jr,
    output wire [31:0] pc8_D
);
```

表四 NPC 信号定义

序号	信号名称	数据方向	位数	功能描述
1	commandAddr_D	I	32	D 级当前指令地址
2	command_D	I	32	D 级当前指令
3	rfrd1_D	I	32	D 级 RF 的 RD1 信号通路中数据，Jr 指令的跳转地址
4	pc_D_B	O	32	计算出的 B 类指令 NPC
5	pc_D_JJal	O	32	计算出的 Jal/J 指令 NPC
6	pc_D_Jr	O	32	计算出的 Jr 指令 NPC
7	pc8_D	O	32	当前指令+8 后的地址，Jal 指令向$ra 寄存器存储该地址

5. CMP

模块功能：比较器

模块定义：

```
module CMP (
    input  wire [31:0] cmpa,
    input  wire [31:0] cmpb,
    input  wire [3:0]  cmpop,
```

```
    output reg branch_true
);
```

表五 CMP 信号定义

序号	信号名称	数据方向	位数	功能描述
1	cmpa	I	32	进行比较的第一个操作数
2	cmpb	I	32	进行比较的第二个操作数
3	cmpop	I	4	比较操作选择信号
4	branch_true	O	1	是否进行 B 类跳转

6. EXT

模块功能：位扩展器

模块定义：

```
module EXT (
    input  wire [15:0] immoffset,
    input  wire [1:0] extop,
    output wire [31:0] extres
);
```

表六 位扩展器信号定义

序号	信号名称	数据方向	位数	功能描述
1	immoffset	I	16	待扩展数据
2	EXTOp	I	2	扩展方式选择信号：零扩展(0)，符号扩展(1)
3	EXTRes	O	32	扩展后的数据

7. FlowReg_E

模块功能：E 级流水线寄存器

模块定义：

```
module FlowReg_E (
    input  wire clk, CLR_E, rfwe_D, dmwe_D, reset, startmd_D,
    input  wire [1:0] rfwdsel_D, rfwrsel_D, asel_D, bsel_D,
    input  wire [2:0] tnew_D, dmtype_D,
    input  wire [3:0] multdivop_D,
    input  wire [4:0] aluop_D, commandtype_D, read1addr_D, read2addr_D,
    input  wire [31:0] rfrd1_D, rfrd2_D, pc8_D, extres_D, command_D, commandAddr_D,
    output reg rfwe_E, dmwe_E, startmd_E,
    output reg [1:0] rfwdsel_E, rfwrsel_E, asel_E, bsel_E,
    output reg [2:0] tnew_E, dmtype_E,
    output reg [3:0] multdivop_E,
    output reg [4:0] aluop_E, commandtype_E, read1addr_E, read2addr_E,
    output reg [31:0] rfrd1_E, rfrd2_E, pc8_E, extres_E, command_E, commandAddr_E
);
```

表七 E 级流水线寄存器信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号
2	CLR_E	I	1	E 级寄存器清零信号
3	reset	I	1	E 级寄存器复位信号
4	rfwe_D	I	1	D 级指令 RF 写使能信号
5	dmwe_D	I	1	D 级指令 DM 写使能信号
6	startmd_D	I	1	D 级指令开始进行乘除运算信号
7	rfwesel_D	I	2	D 级指令 RF 写入数据选择信号
8	rfwrsel_D	I	2	D 级指令 RF 写入地址选择信号
9	asel_D	I	2	D 级指令 ALU A 操作数选择信

				号
10	bsel_D	I	2	D 级指令 ALU B 操作数选择信号
11	dmtype_D	I	3	D 级指令 DM 写入类型选择信号
12	tnew_D	I	3	D 级指令 Tnew 信号
13	multdivop_D	I	4	D 级指令乘除运算单元操作选择信号
14	aluop_D	I	5	D 级指令 ALU 操作选择信号
15	commandtype_D	I	5	D 级指令类型
16	read1addr_D	I	5	D 级指令 RF 读取地址 1
17	read2addr_D	I	5	D 级指令 RF 读取地址 2
18	rfrd1_D	I	32	D 级指令 RF 读取数据 1
19	rfrd2_D	I	32	D 级指令 RF 读取数据 2
20	pc8_D	I	32	D 级指令的地址+8
21	extres_D	I	32	D 级指令立即数扩展结果
22	command_D	I	32	D 级指令
23	commandAddr_D	I	32	D 级指令的地址
24	rfwe_E	O	1	E 级指令 RF 写使能信号
25	dmwe_E	O	1	E 级指令 DM 写使能信号
26	startmd_E			E 级指令开始进行乘除运算信号
27	rfwesel_E	O	2	E 级指令 RF 写入数据选择信号
28	rfwrsel_E	O	2	E 级指令 RF 写入地址选择信号

29	asel_E	O	2	E 级指令 ALU A 操作数选择信号
29	bsel_E	O	2	E 级指令 ALU B 操作数选择信号
30	dmtype_E	O	3	E 级指令 DM 写入类型选择信号
31	tnew_E	O	3	E 级指令 Tnew 信号
32	aluop_E	O	5	E 级指令 ALU 操作选择信号
33	commandtype_E			E 级指令乘除运算单元操作选择信号
34	read1addr_E	O	5	E 级指令 RF 读取地址 1
35	read2addr_E	O	5	E 级指令 RF 读取地址 2
36	rfrd1_E	O	32	E 级指令 RF 读取数据 1
37	rfrd2_E	O	32	E 级指令 RF 读取数据 2
38	pc8_E	O	32	E 级指令的地址+8
39	extres_E	O	32	E 级指令立即数扩展结果
40	command_E	O	32	E 级指令
41	commandAddr_E	O	32	E 级指令的地址

8. ALU

模块功能：算术运算单元

模块定义：

```
module ALU (
    input  wire [31:0] a,
    input  wire [31:0] b,
```

```
    input  wire [4:0] aluop,
    output wire [31:0] res
);
```

表八 ALU 信号定义

序号	信号名称	数据方向	位数	功能描述
1	a	I	32	操作数 1
2	b	I	32	操作数 2
3	aluop	I	5	ALU 操作类型选择信号
4	res	O	32	运算结果

ALU 运算定义表

输入信号	宏定义	执行的操作
1	`AaddB	两个操作数做加法
2	`AsubB	两个操作数做减法
3	`AltB	第一个操作数小于第二个操作数（有符号）
4	`uAltB	第一个操作数小于第二个操作数（无符号）
5	`AandB	两个操作数做与运算
6	`AorB	两个操作数做或运算
7	`AnorB	两个操作数做或非运算
8	`AxorB	两个操作数做异或运算
9	`BsllA	B 逻辑左移 A[4:0]位
10	`BsrlA	B 逻辑右移 A[4:0]位
11	`BsraA	B 算术右移 A[4:0]位
12	`luiB	将 B 的低 16 位加载到高 16 位，后面补零

9. FlowReg_M

模块功能：M 级流水线寄存器

模块定义：

```
module FlowReg_M (
    input  wire clk, rfwe_E, dmwe_E, reset,
    input  wire [1:0] rfwdsel_E, rfwrsel_E,
    input  wire [2:0] tnew_E, dmtype_E,
    input  wire [4:0] read2addr_E, commandtype_E,
    input  wire [31:0] ALUres_E, registerhi_E, registerlo_E, wddm_E, pc8_E,
command_E, commandAddr_E,
    output reg rfwe_M, dmwe_M,
    output reg [1:0] rfwdsel_M, rfwrsel_M,
    output reg [2:0] tnew_M, dmtype_M,
    output reg [4:0] read2addr_M, commandtype_M,
    output reg [31:0] ALUres_M, registerhi_M, registerlo_M, wddm_M, pc8_M,
command_M, commandAddr_M
);
```

表九 M 级流水线寄存器信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	M 级寄存器时钟信号
2	reset	I	1	M 级寄存器复位信号
3	rfwe_E	I	1	E 级指令 RF 写使能信号
4	dmwe_E	I	1	E 级指令 DM 写使能信号
5	rfwdsel_E	I	2	E 级指令 RF 写入数据选择信号
6	rfwrsel_E	I	2	E 级指令 RF 写入地址选择信号
7	dmtype_E	I	3	E 级指令 DM 写入类型选择信

				号
8	tnew_E	I	3	E 级指令 Tnew 信号
9	read2addr_E	I	5	E 级指令 RF 读取地址 2
10	commandtype_E	I	5	E 级指令类型
11	ALUres_E	I	32	E 级指令 ALU 运算结果
12	registerhi_E	I	32	E 级指令 hi 寄存器数值
13	registerlo_E	I	32	E 级指令 lo 寄存器数值
14	wddm_E	I	32	E 级指令 DM 输入数据
15	pc8_E	I	32	E 级指令地址+8
16	command_E	I	32	E 级指令
17	CommandAddr_E	I	32	E 级指令的地址
18	rfwe_M	O	1	M 级指令 RF 写使能信号
19	dmwe_M	O	1	M 级指令 DM 写使能信号
20	rfwdsel_M	O	2	M 级指令 RF 写入数据选择信号
21	rfwrsel_M	O	2	M 级指令 RF 写入地址选择信号
22	dmtype_M	O	3	M 级指令 DM 写入类型选择信号
23	tnew_M	O	3	M 级指令 Tnew 信号
24	read2addr_M	O	5	M 级指令 RF 读取地址 2
25	commandtype_M	O	5	M 级指令类型
26	ALUres_M	O	32	M 级指令 ALU 运算结果

27	registerhi_M	O	32	M 级指令 hi 寄存器数值
28	registerlo_M	O	32	M 级指令 lo 寄存器数值
29	wddm_M	O	32	M 级指令 DM 输入数据
30	pc8_M	O	32	M 级指令地址+8
31	command_M	O	32	M 级指令
32	CommandAddr_M	O	32	M 级指令的地址

10. FlowReg_W

模块功能：W 级流水线寄存器

模块定义：

```
module FlowReg_W (
    input  wire clk, rfwe_M, reset,
    input  wire [1:0] rfwdsel_M, rfwrsel_M,
    input  wire [4:0] commandtype_M,
    input  wire [31:0] ALUres_M, registerhi_M, registerlo_M,
DMRes_M, pc8_M, commandAddr_M, command_M,
    output reg rfwe_W,
    output reg [1:0] rfwdsel_W, rfwrsel_W,
    output reg [4:0] commandtype_W,
    output reg [31:0] ALUres_W, registerhi_W, registerlo_W, DMRes_W,
pc8_W, commandAddr_W, command_W
);
```

表十 E 级流水线寄存器信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	W 级寄存器时钟信号

2	reset	I	1	W 级寄存器复位信号
3	rfwe_M	I	1	M 级指令 RF 写使能信号
4	rfwdsel_M	I	2	M 级指令 RF 写入数据选择信号
5	rfwrsel_M	I	2	M 级指令 RF 写入地址选择信号
6	commandtype_M	I	5	M 级指令类型
7	ALUres_M	I	32	M 级指令 ALU 运算结果
8	registerhi_M	O	32	M 级指令 hi 寄存器数值
9	registerlo_M	O	32	M 级指令 lo 寄存器数值
10	DMRes_M	I	32	M 级指令 DM 输出数据
11	pc8_M	I	32	M 级指令地址+8
12	commandAddr_M	I	32	M 级指令地址
13	command_M	I	32	M 级指令
14	rfwe_W	O	1	W 级指令 RF 写使能信号
15	rfwdsel_W	O	2	W 级指令 RF 写入数据选择信号
16	rfwrsel_W	O	2	W 级指令 RF 写入地址选择信号
17	commandtype_W	O	5	W 级指令类型
18	ALUres_W	O	32	W 级指令 ALU 运算结果
19	registerhi_W	O	32	W 级指令 hi 寄存器数值
20	registerlo_W	O	32	W 级指令 lo 寄存器数值

21	DWRes_W	O	32	W 级指令 DW 输出数据
22	pc8_W	O	32	W 级指令地址+8
23	commandAddr_W	O	32	W 级指令地址
24	command_W	O	32	W 级指令

11. Controller

模块功能：控制器

模块定义：

```
module controller (
    input  wire [31:0] command,
    output wire startmd_D,
     output wire [4:0] commandtype_D,
    output reg rfwe_D, dmwe_D,
    output reg [1:0] wrsel_D, wdsel_D, extop, asel_D, bsel_D,
    output reg [2:0] npcop, tuse_Drs, tuse_Drt, tnew_D, dmtype_D,
    output reg [3:0] cmpop, multdivop_D,
    output reg [4:0] aluop_D
);
```

表十一 控制器信号定义

序号	信号名称	数据方向	位数	功能描述
1	command	I	32	指令的机器码
2	startmd_D	O	1	D 级指令开始进行乘除运算信号
3	rfwe_D	O	1	D 级指令 RF 写使能信号
4	dmwe_D	O	1	D 级指令 DM 写使能信号
5	wrsel_D	O	2	D 级指令 RF 写入地址选择信号
6	wdsel_D	O	2	D 级指令 RF 写入数据选择信号

7	extop	O	2	D 级指令位扩展方法
8	asel_D	O	2	D 级指令 ALU A 操作数选择信号
9	bsel_D	O	2	D 级指令 ALU B 操作数选择信号
10	dmtype_D	O	3	D 级指令 DM 写入类型选择信号
11	npcop	O	3	D 级指令的下一条指令地址计算操作类型
12	tuse_Drs	O	3	D 级指令使用 RF 的 rs 地址对应的数据的时间
13	tuse_Drt	O	3	D 级指令使用 RF 的 rt 地址对应的数据的时间
14	tnew_D	O	3	D 级指令产生新的写入 RF 的时间
15	cmpop	O	4	D 级指令 cmp 操作类型选择信号
16	multdivop_D	O	4	D 级指令乘除运算单元操作选择信号
17	aluop_D	O	5	D 级指令 ALU 操作类型选择信号
18	commandtype_D	O	5	D 级指令类型

控制器 controller 的设计过程本质是一个译码的过程，其功能是将每一条机器指令包含的信息转化为恰当的 CPU 控制信号，并将各个控制信号输出传递到 CPU 的各个功能单元，通过对功能单元的控制达到控制 CPU 运行进程的效果。在 Controller 实现过程中，我采用了宏定义的方法定义了 ALU 的控制信号和指令对应的机器码信号，并在 always@(*)块中进行指令的判断和控制信号的赋值。

12. RiskSolveUnit

模块功能：冲突化解单元

模块定义：

```
module RiskSolveUnit (
    input  wire rfwe_E, rfwe_M, rfwe_W, busy_E, startmd_E,
    input  wire [4:0] read1addr_D, read2addr_D, read1addr_E, read2addr_E,
read2addr_M, writeaddr_E, writeaddr_M, writeaddr_W,
    input  wire [4:0] commandtype_D, commandtype_E, commandtype_M,
commandtype_W,
    output reg [1:0] forward_CMPa_D, forward_CMPb_D, forward_ALUa_E,
forward_ALUb_E, forward_DM_M,

    input  wire [2:0] tuse_Drs, tuse_Drt, tnew_E , tnew_M,
    output reg stall_F, stall_D, clear_D, flush_E,

    input  wire [31:0] command_M
);
```

表十二 冲突化解单元信号定义

序号	信号名称	数据方向	位数	功能描述
1	rfwe_E	I	1	E 级指令 RF 写使能信号
2	rfwe_M	I	1	M 级指令 RF 写使能信号
3	rfwe_W	I	1	W 级指令 RF 写使能信号
4	busy_E	I	1	E 级乘除单元繁忙指令
5	startmd_E	I	1	E 级乘除单元开始运算指令
6	read1addr_D	I	5	即将参与 D 级 CMP 比较的 A 操作数对应的寄存器地址

7	read2addr_D	I	5	即将参与 D 级 CMP 比较的 B 操作数对应的寄存器地址
8	read1addr_E	I	5	即将参与 E 级 ALU 运算的 A 操作数对应的寄存器地址
9	read2addr_E	I	5	即将参与 E 级 ALU 运算的 B 操作数对应的寄存器地址
10	read2addr_M	I	5	即将参与 M 级 DM 写入的数据对应的寄存器地址
11	writeaddr_E	I	5	E 级结果写入的 RF 地址
12	writeaddr_M	I	5	M 级结果写入的 RF 地址
13	writeaddr_W	I	5	W 级结果写入的 RF 地址
14	commandtype_D	I	5	D 级指令类型
15	commandtype_E	I	5	E 级指令类型
16	commandtype_M	I	5	M 级指令类型
17	commandtype_W	I	5	W 级指令类型
18	forward_CMPa_D	O	2	D 级 CMP A 操作数转发选择信号
19	forward_CMPb_D	O	2	D 级 CMP B 操作数转发选择信号
20	forward_ALUa_E	O	2	E 级 ALU A 操作数转发选择信号
21	forward_ALUb_E	O	2	E 级 ALU B 操作数转发选择信号
22	forward_DM_M	O	2	M 级 DM 写入数据转发选择信

				号
上为判断转发的信号，下为判断阻塞的信号				
23	tuse_Drs	I	3	D 级指令 GRF[rs]的使用时间
24	tuse_Drt	I	3	D 级指令 GRF[rt]的使用时间
25	tnew_E	I	3	E 级指令得到新的向 RF 写入的数据的时间
26	tnew_M	I	3	M 级指令得到新的向 RF 写入的数据的时间
27	stall_F	O	1	PC 寄存器冻结信号
28	stall_D	O	1	D 级流水线寄存器冻结信号
29	flush_E	O	1	E 级流水线寄存器清除信号
30	command_M	I	32	M 级指令信号

（三）控制模块设计

控制器的设计本质是译码的过程，根据指令信号定位到具体的指令，并根据指令的操作得到对应的控制信号。

P6 搭建的 CPU 能够处理 50 条指令，与之前搭建的 CPU 相比，支持的指令数量有显著增加。为了简化控制单元设计，我采用了指令分类的方法，根本目的是将控制信号大部分相同的指令的控制信号在一个 if 块中输出。

1. 指令信号解码

R 型指令（add addu sub subu slt sltu and or nor xor sllv srav sll srl sra jr jalr mfhi mflo mthi mtlo mthi mtlo mult multu div divu）：前六位均为 0，可以通过后五位编码

I J型指令（addi addiu slti sltiu andi ori xori lui sw sh sb lw lh lhu lb lbu beq bne bgtz blez j jal ）：可以直接通过前六位编码

Nop指令：全零

特殊指令：bgez、bltz op = 6'b000001 根据指令的20-16位编码

madd、maddu、msub、msubu op = 6'b011100 根据指令的后六位编码

PART1 OPCode == 0

```
`define nop    6'b000000  // op == 0
// double register
`define add    6'b100000  // op == 0
`define addu   6'b100001  // op == 0
`define sub    6'b100010  // op == 0
`define subu   6'b100011  // op == 0
`define slt    6'b101010  // op == 0
`define sltu   6'b101011  // op == 0
`define and    6'b100100  // op == 0
`define or     6'b100101  // op == 0
`define nor    6'b100111  // op == 0
`define xor    6'b100110  // op == 0
`define sllv   6'b000100  // op == 0
`define srlv   6'b000110  // op == 0
`define srav   6'b000111  // op == 0
// shift
```

```
    `define  sll    6'b000000  // op == 0
    `define  srl    6'b000010  // op == 0
    `define  sra    6'b000011  // op == 0
    // jump register
    `define  jr     6'b001000  // op == 0
    // jump register and link
    `define  jalr   6'b001001  // op == 0
    // mult and div
    `define  mfhi   6'b010000  // op == 0
    `define  mflo   6'b010010  // op == 0
    `define  mthi   6'b010001  // op == 0
    `define  mtlo   6'b010011  // op == 0
    `define  mult   6'b011000  // op == 0
    `define  multu  6'b011001  // op == 0
    `define  div    6'b011010  // op == 0
    `define  divu   6'b011011  // op == 0
PART2  OPCode != 0
    // register and imm
    `define  addi   6'b001000  // op != 0
    `define  addiu  6'b001001  // op != 0
    `define  slti   6'b001010  // op != 0
    `define  sltiu  6'b001011  // op != 0
    `define  andi   6'b001100  // op != 0
    `define  ori    6'b001101  // op != 0
```

```
    `define  xori    6'b001110  // op != 0
    `define  lui     6'b001111  // op != 0
    // save and load
    `define  sw      6'b101011  // op != 0
    `define  sh      6'b101001  // op != 0
    `define  sb      6'b101000  // op != 0
    `define  lw      6'b100011  // op != 0
    `define  lh      6'b100001  // op != 0
    `define  lhu     6'b100101  // op != 0
    `define  lb      6'b100000  // op != 0
    `define  lbu     6'b100100  // op != 0
    // double register jump
    `define  beq     6'b000100  // op != 0
    `define  bne     6'b000101  // op != 0
    // single register jump
    `define  bgtz    6'b000111  // op != 0
    `define  blez    6'b000110  // op != 0
    // jump
    `define  j       6'b000010  // op != 0
    // jump and link
    `define  jal     6'b000011  // op != 0
PART3 special command
   OPCode = 6'b000001
       `define  bgez    5'b00001
       `define  bltz    5'b00000
   OPCode = 6'b011100
       `define  madd    6'b000000
       `define  maddu   6'b000001
       `define  msub    6'b000100
      `define  msubu   6'b000101
```

2. 指令分类

根据指令的操作类型，我将指令分为一下几类：

a. jumpandlink(跳转并链接类)

此类指令特点是向寄存器写入 PC+8，在 RiskSolveUnit 判断转发回写数据时会用到。

包含：jal、jalr

b. calmudv（乘除法计算类）

此类指令的特点是需要在乘除运算单元做运算，且会使得乘除运算单元 busy 一定的时钟周期

包含：mult、multu、div、divu

c. readmudv(读取乘除运算结果类)

此类指令的特点是会读取 hi 或 lo 寄存器中的数值，但不会使得乘除运算单元 busy

包含：mflo、mfhi

d. setmudv(对 hi 或 lo 寄存器赋值类)

此类指令的特点是会对 hi 或 lo 寄存器赋特定的值，需要在乘除运算单元进行运算，但不会使乘除运算单元 busy

包含：mtlo、mthi

e. caldoubleregister(双寄存器运算类)

此类指令的特点是两个寄存器中的数值进行运算，并将结果写入到第三个寄存器中。

包含指令：add、addu、sub、subu、slt、sltu、and、or、nor、xor、sllv、srlv、srav

f. shift(移位(立即数位）类)

包含指令：sll、srl、sra

g. calregisterimmsign(寄存器和符号扩展立即数计算类)

包含指令：addi、addiu、slti、sltiu

h. calregisterimmzero(寄存器和零扩展立即数计算类)

包含指令：andi、ori、xori、lui

i. save(存储类)

包含指令：sw、sh、sb

j. load(载入类)

包含指令：lw、lh、lhu、lb、lbu

k. mudvALU(乘除加减运算类)

包含指令：madd、maddu、msub、msubu

l. 未分类指令

nop、j、jr、(beq、bne)、(bgez、bltz)、(bgtz、blez)

3. 控制信号输出

这部分是各指令对应的控制信号真值表，与前几个 Project 不同的是，P6 采用的是根据指令类输出控制信号，代码实现举例：

```
if(commandtype_D == `caldoubleregister)begin
    npcop = 0;          wrsel_D = 1;
    wdsel_D = 0;        rfwe_D = 1;
    extop = 0;          cmpop = `None;
    asel_D = 0;         bsel_D = 0;
    dmwe_D = 0;         dmtype_D = `None;
    tnew_D = 2;         tuse_Drs = 1;
```

```
        tuse_Drt = 1;
        case(command[5:0])
            `add : begin aluop_D = `AaddB; end
            `addu : begin aluop_D = `AaddB; end
            `sub : begin aluop_D = `AsubB; end
            `subu : begin aluop_D = `AsubB; end
            `slt : begin aluop_D = `AltB;  end
            `sltu : begin aluop_D = `uAltB; end
            `and : begin aluop_D = `AandB; end
            `or   : begin aluop_D = `AorB;  end
            `nor : begin aluop_D = `AnorB; end
            `xor  : begin aluop_D = `AxorB; end
            `sllv: begin aluop_D = `BsllA; end
            `srlv :  begin aluop_D = `BsrlA; end
            `srav: begin aluop_D = `BsraA; end
            default :  begin aluop_D = 5'd0; end
        endcase
    end
```

下面列出各个指令的控制信号真值表：

表十三 控制信号逻辑真值表（一）

	npcop	wrsel_D	wdsel_D	rfwe_D	extop_D	asel_D	bsel_D
add	0	1	0	1	x	0	0
addu	0	1	0	1	x	0	0
sub	0	1	0	1	x	0	0
subu	0	1	0	1	x	0	0
slt	0	1	0	1	x	0	0
sltu	0	1	0	1	x	0	0
and	0	1	0	1	x	0	0
or	0	1	0	1	x	0	0

nor	0	1	0	1	x	0	0
xor	0	1	0	1	x	0	0
sllv	0	1	0	1	x	0	0
srlv	0	1	0	1	x	0	0
srav	0	1	0	1	x	0	0
sll	0	1	0	1	2	1	0
slr	0	1	0	1	2	1	0
sra	0	1	0	1	2	1	0
jr	3	x	x	0	0	0	0
jalr	3	1	2	1	0	0	0
mfhi	0	1	0	1	0	0	0
mflo	0	1	0	1	0	0	0
mthi	0	x	x	0	x	0	0
mtlo	0	x	x	0	x	0	0
mult	0	x	x	0	x	0	0
multu	0	x	x	0	x	0	0
div	0	x	x	0	x	0	0
divu	0	x	x	0	x	0	0
addi	0	0	0	1	1	0	1
addiu	0	0	0	1	1	0	1
slti	0	0	0	1	1	0	1
sltiu	0	0	0	1	1	0	1
andi	0	0	0	1	0	0	1
ori	0	0	0	1	0	0	1
xori	0	0	0	1	0	0	1
lui	0	0	0	1	0	0	1
sw	0	x	x	0	1	0	1
sh	0	x	x	0	1	0	1
sb	0	x	x	0	1	0	1

lw	0	0	1	1	1	0	1
lh	0	0	1	1	1	0	1
lhu	0	0	1	1	1	0	1
lb	0	0	1	1	1	0	1
lbu	0	0	1	1	1	0	1
beq	1	x	x	0	1	0	0
bne	1	x	x	0	1	0	0
bgtz	1	x	x	0	1	0	0
blez	1	x	x	0	1	0	0
j	2	x	x	0	x	0	0
jal	2	2	2	1	x	0	0
bgtz	1	x	x	0	1	0	0
bltz	1	x	x	0	1	0	0
nop	0	x	x	0	x	x	x
madd	0	x	x	0	x	x	x
maddu	0	x	x	0	x	x	x
msub	0	x	x	0	x	x	x
msubu	0	x	x	0	x	x	x

表十四 控制信号逻辑真值表（二）

	aluop_D	dmwe_D	dmtype_D	tnew_D	tuse_Drs	tuse_Drt	cmp_op
add	1	0	x	2	1	1	x
addu	1	0	x	2	1	1	x
sub	2	0	x	2	1	1	x
subu	2	0	x	2	1	1	x
slt	3	0	x	2	1	1	x
sltu	4	0	x	2	1	1	x
and	5	0	x	2	1	1	x
or	6	0	x	2	1	1	x

nor	7	0	x	2	1	1	x
xor	8	0	x	2	1	1	x
sllv	9	0	x	2	1	1	x
srlv	10	0	x	2	1	1	x
srav	11	0	x	2	1	1	x
sll	9	0	x	2	5	1	x
slr	10	0	x	2	5	1	x
sra	11	0	x	2	5	1	x
jr	0	0	x	0	0	5	x
jalr	0	0	x	2	0	5	x
mfhi	0	0	x	2	5	5	x
mflo	0	0	x	2	5	5	x
mthi	0	0	x	0	1	5	x
mtlo	0	0	x	0	1	5	x
mult	0	0	x	0	1	1	x
multu	0	0	x	0	1	1	x
div	0	0	x	0	1	1	x
divu	0	0	x	0	1	1	x
addi	1	0	x	2	1	5	x
addiu	1	0	x	2	1	5	x
slti	3	0	x	2	1	5	x
sltiu	4	0	x	2	1	5	x
andi	5	0	x	2	1	5	x
ori	6	0	x	2	1	5	x
xori	8	0	x	2	1	5	x
lui	12	0	x	2	1	5	x
sw	1	1	1	0	1	2	x
sh	1	1	2	0	1	2	x
sb	1	1	4	0	1	2	x

lw	1	0	1	3	1	5	x
lh	1	0	2	3	1	5	x
lhu	1	0	3	3	1	5	x
lb	1	0	4	3	1	5	x
lbu	1	0	5	3	1	5	x
beq	0	0	x	0	0	0	1
bne	0	0	x	0	0	0	2
bgtz	0	0	x	0	0	5	3
blez	0	0	x	0	0	5	6
j	0	0	x	0	5	5	x
jal	0	0	x	2	5	5	x
bgez	0	0	x	0	0	5	4
bltz	0	0	x	0	0	5	5
nop	0	x	0	0	5	5	x
madd	0	0	x	0	1	1	x
maddu	0	0	x	0	1	1	x
msub	0	0	x	0	1	1	x
msubu	0	0	x	0	1	1	x

（四）重要机制实现方法

1. 跳转

NPC 模块利用 D 级指令地址，计算 D 级指令的下一条指令所有可能的跳转地址（不区分 D 级指令的类型），得到 pc_D_B, pc_D_JJal, pc_D_Jr, 并将所有计算出的、可能的地址传输到 PC 中。PC 模块计算 F 级指令地址+4 的结果，并接收由 controller 传递的下一条指令运算方法信号（npcop），利用 mux 元件对得到的所有可能的地址进行选择，得到正确的地址，储存到 PC 模块的寄存器中。

2. 流水线延迟槽

流水线延迟槽即当前指令为分支类指令时，不论是否发生跳转，CPU 都执行下一条指令，这样我们就能够通过编译调度，将适当的指令移到延迟槽中，从而充分发挥流水线 CPU 的性能。具体到 CPU 设计实现的过程，我们只需要在 D 级进行完跳转指令的判断后的下一个时钟周期，无论结果如何、是否跳转，都不清除 D 级流水线寄存器中的指令，这样分支指令的下一条指令无论如何都将被执行，也就是达到了延迟槽的功能。

3. 转发

转发是要解决当一条指令的执行需要某个寄存器中的数值，但是该指令之前的指令已经改变了该寄存器的值且改变后的值还没有写入到寄存器中的情况。为了得到当前寄存器中的正确的值，我们需要将已经得到的、改变的寄存器编号与本条指令当前需要的寄存器编号相同的、存储于后续流水级的数据转发到之前的流水级中。

转发的本质是 RF 涉及到流水线寄存器的两个级（D 级和 W 级），造成的 RF 读和写阶段的分离使得我们需要通过转发来解决数据冲突的问题。通过分析需要用到 RF 数据的流水级（D 级、E 级和 M 级），以及能够保存新的 RF 寄存器中数据的流水级（M 级和 W 级）不难得知转发的位点有且只有三处：

（1） D 级比较器的两个输入端（此处转发的结果也作为 D 级 RF 寄存器中读取的数据传递到 E 级）

转发数据来源：

a. M 级 PC+8（pc8_M）

b. M 级 ALU 的结果（ALUres_M）

c. RF 寄存器读出的结果（注：CPU 设计时 RF 采用了内部转发，此处无需转发 W 级回写数据）

（2） E 级 ALU 的两个输入端

转发数据来源：

a. M 级 PC+8（pc8_M）

b. M 级 ALU 的结果（ALUres_M）

c. W 级回写寄存器的数据（res_W）

d 沿流水线寄存器传递到 E 级 ALU 输入端的数据（rfrd_E）

（3） M 级 DM 的数据输入端

转发数据来源：

a. W 级回写寄存器的数据（res_W）

b 沿流水线寄存器传递到 M 级 DM 数据输入端的数据（wddm_M）

CPU 设计时，在 RiskSolveUnit 模块产生上述转发的多路选择器选择信号，大致的思想是“有转发数据则用转发数据，多个转发数据采用新的转发数据”，这里的新的指的就是出现时间较晚的指令，也就是靠前的流水级。在 RiskSolveUnit 模块，转发选择信号的生成规则是：只要当前位点的读取寄存器地址和某转发输入来源的写入寄存器地址相等且不为 0，而转发信号的优先级可以通过对 if-else 指令排列的顺序来确定，对优先级高的信号优先判断即可。例如 E 级 ALU 的 A 输入端的转发数据选择信号的生成方法为：

```
if(rfwe_M == 1 && read1addr_E == writeaddr_M && read1addr_E == 5'd31 &&
                                                command_M[31:26] == `jal)begin
    forward_ALUa_E = 2'd3;
end
else if(rfwe_M == 1 && read1addr_E == writeaddr_M && read1addr_E != 0) begin
    forward_ALUa_E = 2'd2;
end
else if(rfwe_W == 1 && read1addr_E == writeaddr_W && read1addr_E != 0) begin
    forward_ALUa_E = 2'd1;
end
else begin
    forward_ALUa_E = 2'd0;
end
```

这样我们就能够在 RiskSolveUnit 模块中得到正确的选择信号，将其输入到各个转发部件的选择信号端口，即可转发恰当的数据。

4. 暂停

阻塞是要解决当一条指令的执行需要某个寄存器中的数值，但是该指令之前

的指令即将改变该寄存器的值且得到改变数值的时间要大于需求该寄存器数值的时间，也就是说之前的指令无法在该指令需要该寄存器值之前得到更新该寄存器的值，这时候我们就不得不阻塞流水线 CPU 的运行，直到之前的指令能够在该指令需求前得到更新后的寄存器数值。

阻塞的实现采用的是教程中的供给时间模型。

对于某一个指令的某一个数据需求，我们定义需求时间 Tuse 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。

对于某个指令的数据产出，我们定义供给时间 Tnew 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。

在设计过程中，我在 controller 模块进行指令译码的同时，赋予指令 Tnew_D、Tuse_Drs 以及 Tuse_Drt 的具体数值，其中 Tnew 为动态数值，在不同流水级中的数值不同，需要随流水线向下传递，其计算方法为 Tnew_n = max(Tnew_n-1,0) n 为流水线级数。通过分析得知，我们只需要在指令刚刚进入流水线（处于 D 级）时做出判断：通过比对 D 级指令的 Tuse_Drs、Tuse_Drt 和 E、M 级指令的 Tnew，当 Tnew>Tuse 需要暂停流水线，其余情况均可通过转发避免数据冲突。（注： 对于不产生新数据的指令，将 Tnew 设置成 0；对于不需要使用数据的指令，将 Tuse 设置成 5，这样能够确保不产生新数据或不需要新数据的指令不会引发暂停）。当我们得知需要对流水线做暂停操作时，即在 RiskSolveUnit 输出 PC 寄存器的冻结信号，D 级流水线寄存器的冻结信号以及 E 级流水线的清除信号，这样就相当于我们在 D 级指令之前插入了一个 nop 指令。如此，就完成了暂停机制的构建。

二、测试方案

（一）典型测试样例

1. 顺序执行测试

```
ori $at,123
ori $v0,456  # 构造正数


lui $v0,0xffff
```

```
ori $v1,$v0,456
lui $a0,0xffff
ori $a1,$a0,795  # 构造负数

addu $a2,$at,$v0  # 正正
subu $a3,$v1,$a1  # 负负
slt  $t0,$a2,$a3  # 正负
sltu $t1,$a3,$a2  # 负正

and  $t2,$a2,$at  # 正正
or   $t3,$a1,$a3  # 负负
nor  $t4,$a2,$a3  # 正负
xor  $t5,$a3,$t4  # 负正
sllv $t6,$a1,$a2
srlv $t7,$a2,$a3
srav $s0,$t1,$s2

mult $t1,$t2
mfhi $t0
mflo $t1
mthi $s0
divu $t1,$t3
mflo $t5

ori $t6,0
ori $t7,1
nop
sw $t2,0($t6)
sh $t3,6($t6)
sb $t4,7($t6)
```

```
sw $t5,4($t6)

ori $k0,0
ori $k1,1
nop
lw $s0,12($t6)
lh $s1,10($t6)
lb $s2,4($t6)
lw $s3,0($t6)

ori $v1,4
ori $k0,4
subu $k1,$k1,$k1
```

2. 转发测试

```
  # ALU+ALU
lui $t0,0xffff
ori $t0,$t0,0xffff
lui $t1,10
ori $t1,$t1,100
mthi $t1

  # ALU（包括乘除类）+save 类
addu $t3,$t1,$t0
sllv $t4,$t0,$t3
mult $t3,$t4
mflo $t5
sw $t5,0($0)

  # ALU（包括乘除类）+save 类
```

```
ori $t6,$t6,100
sw $6,4($0)
lui $t7,100
sh $t7,8($0)

  Jal or jalr+save类
jal f1
sb $ra,12($0)

  Jal or jalr+ALU（包括乘除类）
f1:
jal f2
srav $s0,$t7,$ra

  Jal or jalr+save类
f2:
jal f3
sw $ra,24($0)
addu $k0,$ra,5
f3:
sw $s2,16($0)
ori $s3,100
ori $s4,100
bne $s3,$s4,f4
ori $s5,$s5,235
ori $s6,$s6,794

f4:
sw $s6,20($0)
```

3. 暂停测试

```
 # ALU（包括乘除类）+B 类
ori $t0,$t0,4
ori $t1,$t1,4
bne $t0,$t1,f1
nop
ori $s0,10
mflo $t0
bgtz $t0,f2
nop
f2:

  # Jal+B 类
f1:
ori $s0,100
jal f2
ori $t2,$s0,320
f2:
beq $ra,$t2,f3
nop
f3:
sw $t0,0($0)
sw $t1,4($0)
sw $s0,8($0)
sw $t2,12($0)

# load 类+B 类（暂停两拍）
lw $t3,0($0)
lw $t4,4($0)
```

```
beq $t3,$t4,f4
nop

  # load类+ ALU（包括乘除类）
f4:
lw $s5,8($0)
lw $s6,12($0)
addu $k0,$s5,$s6
lbu $t0,3($0)
mthi $t0

  # load类+save类
lw $s5,0($0)
sw $s6,0($s5)

  # load+save（不暂停）
lw $t0,4($0)
lw $t1,8($0)
sw $t1,0($t0)

  # 乘除法+乘除法
mult $t0,$t1
mflo $t2

mult $s0,$s1
divu $t1,$t2
```

4. 跳转及延迟槽测试

```
ori $t0,$t0,4
ori $t1,$t1,5
```

```
ori $t2,$t2,4

  # B 类跳转
bne  $t0,$t1,f1
subu $s0,$t0,$t2
sllv $s1,$t1,$t3

f1:
subu $t4,$t1,$t0

  # jal or jalr 跳转
jal f2
lui $t5,100
ori $t5,300
la $s1,f3
jalr $31,$s1
nop
nop
f3:

  # B 类跳转
beqz $t0,end
f2:
addu $t6,$ra,$t2
ori $k0,$k0,10
ori $k1,$k1,20

  # B 类跳转
beq $k0,$k1,f4
addu $t6,$t5,$k0
```

```
addu $k0,$k0,$k0
bgtz $k0,f5
nop

  # j 跳转
j f3
subu $t7,$t6,$k1
ori $a0,$a0,10
f4:

  # jr 跳转
jr $ra
sw $ra,0($0)
lw $ra,4($0)
end:
sw $t0,0($0)
```

（二）自动测试工具（python 环境）

1. 测试样例生成器

```
# 测试样例生成程序，包含 P6 要求的除 add,addi,sub(因为可能产生溢出，Mars 报错)外的所有指令

import random

data1 = []  # 记录顺序执行的代码
data2 = []  # 记录跳转代码  # beq bne blez bgtz bltz bgez j    jal jr
interval = []  # 记录中间插入部分代码的行数

# 九个大指令类
calmudv = ['mult', 'multu', 'div', 'divu']
```

```
    readmudv = ['mflo', 'mfhi']
    setmudv = ['mtlo', 'mthi']
    caldoubleregister = ['addu', 'subu', 'slt', 'sltu', 'and', 'or', 'nor', 'xor', 'sllv',
'srlv', 'srav']
    shift = ['sll', 'srl', 'sra']
    calregisterimm = ['addiu', 'andi', 'ori', 'xori']
    saveload = ['sw', 'sh', 'sb', 'lw', 'lh', 'lhu', 'lb', 'lbu']
    shifti = ['slti', 'sltiu']
    lui = ['lui']

    # 跳转类
    btypetwo = ['beq', 'bne']
    btypeone = ['blez', 'bgtz', 'bltz', 'bgez']
    jforward = ['jal']
    jback = ['jr']


    class Instruction:
        def __init__(self, type, lines=0, rs=None, rt=None, rd=None, oriindex=None):
            self.type = type
            self.lines = lines
            self.rs = rs
            self.rt = rt
            self.rd = rd
            self.oriindex = oriindex


    def pre_fill():
        for i in range(2, 28):
            print("lui ${},{}".format(i, random.randint(0, 65535)))
```

```
            print("ori ${},${},{}".format(i, i, random.randint(0, 65535)))
        for i in range(10):
            print("sw ${},{}(${})".format(random.randint(0, 27), i * 4, 0))



    def print_instr(x):
        if data1[x].type == 1:
            print("{} ${},${}".format(random.choice(calmudv), data1[x].rs, data1[x].rt,
data1[x].rd))
        elif data1[x].type == 2:
            print("{} ${}".format(random.choice(readmudv), data1[i].rs))
        elif data1[x].type == 3:
            print("{} ${}".format(random.choice(setmudv), data1[i].rs))
        elif data1[x].type == 4:
            print("{} ${},${},${}".format(random.choice(caldoubleregister), data1[i].rs,
data1[i].rt, data1[i].rd))
        elif data1[x].type == 5:
            print("{} ${},${},{}".format(random.choice(shift), data1[i].rs, data1[i].rt,
random.randint(0, 31)))
        elif data1[x].type == 6:
            print(
                "{} ${},${},{}".format(random.choice(calregisterimm), data1[i].rs,
data1[i].rt, random.randint(0, 65535)))
        elif data1[x].type == 7:
            print("{} ${},{}($0)".format(random.choice(saveload), data1[i].rs,
random.randrange(0, 8192, 4)))
        elif data1[x].type == 8:
            print("{} ${},${},{}".format(random.choice(shifti), data1[i].rs, data1[i].rt,
random.randint(0, 31)))
        elif data1[x].type == 9:
```

```
            print("{} ${},{}".format(random.choice(lui), data1[i].rt, random.randint(0,
65535)))
        else:
            print("nop")



    if __name__ == "__main__":
        pre_fill()
        total_blocks = random.randint(1, 25)  # 代码块数
        total_lines = 0  # 代码行数
        cnt = 0
        # 构建过程
        for i in range(total_blocks):  # 代码块头尾记录在 data2 中
            data2.append(Instruction(random.randint(0, 3), random.randint(1, 25)))
            total_lines += data2[i].lines
        for i in range(total_blocks + 1):  # 两个 interval 之间夹一个代码块
            interval.append(random.randint(1, 30))
            total_lines += interval[i]
        for i in range(0, total_lines):  # 随机生成顺序执行代码，保存在 data1 中
            data1.append(Instruction(random.randint(1, 10)))
            data1[i].rs = random.randint(2, 31)
            data1[i].rt = random.randint(2, 27)
            data1[i].rd = random.randint(2, 27)


        for i in range(total_blocks):
            coin = random.randint(0, 3)
            if coin == 0:
                data2[i].rs = data2[i].rt = random.randint(0, 31)
            else:
                data2[i].rs = random.randint(0, 31)
```

```
            data2[i].rt = random.randint(0, 31)


        print("ori $29,$0,0x00002F00")
        print("jal funct")
        print("ori $3,$3,100")  # 测试延迟槽
        print("ori $4,$4,200")
        print("funct:")  # 初始化 31 号寄存器


        # 输出过程
        k = 0
        for i in range(total_blocks):
            print("interval{}:".format(cnt))
            cnt += 1
            for j in range(interval[i]):  # 输出 interval
                print_instr(k)
                k += 1
            if data2[i].type == 0:  # 输出 data2[i]
                print("{} ${},${},branch{}".format(random.choice(btypetwo), data2[i].rs,
data2[i].rt, i))
                print("ori $3,$3,100")  # 测试延迟槽
                print("ori $4,$4,200")
                print("interval{}:".format(cnt))
                cnt += 1
                for j in range(data2[i].lines):
                    print_instr(k)
                    k += 1
                print("branch{}:".format(i))
            elif data2[i].type == 1:
                print("{} ${} branch{}".format(random.choice(btypeone), data2[i].rs, i))
                print("ori $3,$3,100")  # 测试延迟槽
```

```
            print("ori $4,$4,200")
            print("interval{}:".format(cnt))
            cnt += 1
            for j in range(data2[i].lines):
                print_instr(k)
                k += 1
            print("branch{}:".format(i))
        elif data2[i].type == 2:
            print("j branch{}".format(i))
            print("ori $3,$3,100")  # 测试延迟槽
            print("ori $4,$4,200")
            print("interval{}:".format(cnt))
            cnt += 1
            for j in range(data2[i].lines):
                print_instr(k)
                k += 1
            print("branch{}:".format(i))
        elif data2[i].type == 3:
            coin = random.randint(0, 1)
            if coin == 0:  # jal
                print("jal funct{}".format(i))
            else:
                print("la $t0,funct{}".format(i))
                print("jalr $31, $t0")
            print("ori $3,$3,100")  # 测试延迟槽
            print("ori $4,$4,200")
            print("interval{}:".format(cnt))
            cnt += 1
            for j in range(data2[i].lines):
                print_instr(k)
```

```
                k += 1
            print("beq $0,$0,interval{}".format(cnt))
            print("ori $3,$3,100")  # 测试延迟槽
            print("ori $4,$4,200")
            print("funct{}:".format(i))
            print("jr $ra")
            print("ori $3,$3,100")  # 测试延迟槽
            print("ori $4,$4,200")

    print("interval{}:".format(cnt))
    for j in range(interval[-1]):  # 输出interval
        print_instr(k)
        k += 1

    # print("end:")
    # print("beq $0, $0, end")
    # print("nop")
```

2. 自动执行脚本

```
import os
import re


def gen(num):
    os.system(r"python createP6.py >" + num + "-mips.asm")  # 生成随机代码
    os.system(
        "java -jar mars2.jar " + num + "-mips.asm nc mc CompactDataAtZero a dump .text
HexText code.txt")  # 导出Mars机器码
    os.system(
        "java -jar mars2.jar " + num + "-mips.asm db nc mc CompactDataAtZero> " + num
```

```
+ "-ans.txt")  # 导出 Mars 输出结果（初始数据地址为 0，开启延迟槽）


        # 生成仿真需要的两个文件
        tclFile = open("test.tcl", "w")
        tclFile.write("run 10us;\nexit")


        prjFile1 = open("testmine.prj", "w")
        for root, dirs, files in os.walk(r"E:\1-Project\P6\testprogram\modulemine"):
            for fileName in files:
                if re.match(r"[\w]*\.v", fileName):
                    prjFile1.write("Verilog work " + root + "\\" + fileName + "\n")


        prjFile2 = open("testzqy.prj", "w")
        for root, dirs, files in os.walk(r"E:\1-Project\P6\testprogram\modulezqy"):
            for fileName in files:
                if re.match(r"[\w]*\.v", fileName):
                    prjFile2.write("Verilog work " + root + "\\" + fileName + "\n")


        tclFile.close()
        prjFile1.close()
        prjFile2.close()


        os.environ['XILINX'] = r"D:\ISESetup\14.7\ISE_DS\ISE"
        # 命令行运行 ISim
        os.system(
            r"D:\ISESetup\14.7\ISE_DS\ISE\bin\nt64\fuse --nodebug --prj testmine.prj -o
testmipsmine.exe mips_txt >CompileLogmine.txt")


        os.system("testmipsmine.exe -nolog -tclbatch test.tcl >" + num + "-myoutput.txt")
```

```
        os.system(
            r"D:\ISESetup\14.7\ISE_DS\ISE\bin\nt64\fuse --nodebug --prj testzqy.prj -o
testmipszqy.exe mips_txt >CompileLogzqy.txt")

        os.system("testmipszqy.exe -nolog -tclbatch test.tcl >" + num + "-zqyoutput.txt")

        os.system(r"python compare.py " + num)  # 调用 compare.py 比对输出


    if __name__ == "__main__":
        for i in range(20):  # 可设定执行次数
            gen(str(i))
```

3. 正确性判定脚本

```
    import os
    import re
    import sys


    def cmp(num):
        # 读取三个 txt 文件
        file1 = open(num + '-myoutput.txt', "r")
        file2 = open(num + '-zqyoutput.txt', "r")
        file3 = open(num + '-ans.txt', "r")

        text1 = file1.read()
        text2 = file2.read()
        text3 = file3.read()
```

```
        flag = True
        if text1 == text2:
            print("时间完全一致")
        else:
            print("时间不一致")
            flag = False

        file1.close()
        file2.close()
        file3.close()

        # 由于不同程序的结果输出数据的时间可能有差异，且Mars输出的结果不带有时间，所以采用正则表达式匹配，并构造(指令地址，写入对象，写入数据三元组)
        obj = re.compile(r".*?@(?P<pc>.*?): (?P<place>.*?) <= (?P<number>.*?)\n", re.S)
        result1 = obj.finditer(text1)
        result2 = obj.finditer(text2)
        result3 = obj.finditer(text3)

        operation1 = []
        operation2 = []
        operation3 = []

        for it in result1:
            operation1.append((it.group('pc'), it.group('place'), it.group('number')))
        for it in result2:
            operation2.append((it.group('pc'), it.group('place'), it.group('number')))
        for it in result3:
            operation3.append((it.group('pc'), it.group('place'), it.group('number')))

        # 将三元组按照指令地址排序
```

```
    operation1.sort(key=lambda x: x[0])
    operation2.sort(key=lambda x: x[0])
    operation3.sort(key=lambda x: x[0])

    res = open(num + '-diff.txt', 'w')

    if not (len(operation1) == len(operation2) and len(operation2) == len(operation3)):
        flag = False
        res.write("diff in length\n")
        res.write("len(res1) = {}\n".format(len(operation1)))
        res.write("len(res2) = {}\n".format(len(operation2)))
        res.write("len(res3) = {}\n\n".format(len(operation3)))

    for i in range(min(len(operation1), len(operation2), len(operation3))):
        if not (operation1[i] == operation2[i]):
            flag = False
            res.write("diff in len({})\n".format(i + 1))
            res.write("res1 = {}\n".format(operation1[i]))
            res.write("res2 = {}\n".format(operation2[i]))
            res.write("res3 = {}\n\n".format(operation3[i]))

    res.close()

    if flag:
        print("完全正确")
        os.remove(num + '-diff.txt')


if __name__ == "__main__":
    for i in range(1, len(sys.argv)):
```

```
        strs = sys.argv[i]

        cmp(strs)
```

三、思考题

（一）

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？

首先，相比于其它 ALU 支持的运算，乘除法的运算时间较长(这也就是教程要求我们模拟乘除法计算延迟、产生 busy 信号的原因)，将其整合到 ALU 中会使得 E 级流水级的最短时钟周期变长，从而拖慢整个流水线的时钟周期，降低流水线的工作效率。

其次，乘法涉及到溢出，除法涉及到不整除产生商和余数的情况，这就意味着在一次乘除法运算中，我们能够得到两个结果。如果我们采用单独的乘除法部件，其内部包含独立的 HI 和 LO 寄存器，就意味着我们在一次乘除运算中即可产生我们可能需要的两个结果，并将其保存在 HI 和 LO 寄存器中。相反，如果我们的乘除运算单元没有寄存器或只有一个独立的寄存器，就意味着一次乘除运算仅能产生高 32 位或低 32 位或商或余数，而如果我们既需要高 32 位有需要低 32 位，或者既需要商又需要余数，我们就只能进行两次乘除法运算，这与直接用 HI 和 LO 寄存器保存两个结果相比显然是更差的实现方法。

（二）

参照你对延迟槽的理解，试解释“乘除槽”。

延迟槽是指无论前一条跳转指令结果如何，我们都会执行下一条指令。从而我们可以利用编译优化，在延迟槽中插入不会对之后的程序产生影响的指令，从而提高程序运行效率。

同理，对于乘除法运算，需要大量的时间，我们也可以通过编译调度，提取出与乘除操作无关的指令，将其放在乘除运算的后面。这样，CPU 就可以利用等待乘除运算产生结果的时间来运行之后的指令，从而提高 CPU 的执行效率。

（三）

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）

数据存储是以字节为单位，考虑 C 语言中字符串的情况，一个字符需要一

个字节大小的存储空间，为了提高存储空间的利用率，我们显然不能用一个字大小的存储空间来储存一个字符，而应在一个字大小的存储空间中存储 4 个字符，在这种情况下，按字节访问内存就体现出了它的优势。如果我们具备按字节访存的能力，就可以直接在内存中提取或者存入出我们需要的字符，因为一个字符和一个字节大小的存储单元是相对应的。相反，如果我们只具备按字存储的能力，在存入字符时，我们必须先将内存中一字大小的数据提取出来，将待存储的数据拼接其中，再将其存入内存中；在取出字符时，我们必须从内存中取出一字大小的数据，从中分离出我们想要的 8 位数据，再进行位扩展至 32 位。显然，在处理一个字节大小的数据时，按字节存储要比按字存储有极大的优势。

（四）

为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？

1. 对于功能相对较多的存储或者运算部件，如 ALU、CMP、DM、multdiv，将它们的运算或存储方法选择信号用宏定义，使代码有了较高的可读性，且易于维护和修改；

2. 采用了对指令进行分类的方法。由于 P6 的指令可以大致分为几类，各类中的指令控制信号大体相同，如双寄存器计算类指令的控制信号除 ALU 操作选择信号不同外，其余均相同。所以在控制单元中，我将同一指令类中指令的控制信号在一个 if 条件块中统一输出，这样极大降低了控制单元的代码量，同时也有利于后续的冲突解决单元以及乘除运算单元根据不同的指令类型产生不同的操作。

3. 控制单元分析指令产生控制信号后，将所有的控制信号按照固定的顺序拼接后输出，这样可以极大降低流水线寄存器的端口数量。因为需要改动之处过多，这种手段在我的 CPU 中并没有实际实现，但我认为采用这种方法能够降低信号在流水级之间信号传递的复杂度，类似于将分布式译码和集中式译码的优势综合起来，即无需在每一流水级进行指令的译码，也达到了降低了在流水级之间传递信息的复杂度。

（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？

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

流水线 CPU 可能出现的冒险类型为结构冒险，控制冒险和数据冒险，在本实验中，RF 的读和写是分离端口的，指令存储和数据存储元件是相互分离的，所以我们不必考虑控制冒险；对于控制冒险，我们通过将比较提前至 E 级，并应用了延迟槽，解决了可能出现的问题。我们需要关注的指令冲突类型为特定指令序列造成的数据冲突。

本实验的数据冲突类型可分为靠转发解决和靠暂停解决两大类，下面分别列出这两类包含的指令序列类型：

1. 靠转发解决的指令序列

a. ALU+ALU

```
lui $t0,0xffff
ori $t0,$t0,0xffff
mult $t0,$t1
```

b. ALU+save

```
addu $t5,$t3,$t4
mult $t5,$t3
sw $t5,0($0)
addu $t1,$t2,$t3
divu $t1,$t2
sw $t5,4($t1)
```

c. ALU+load

```
addu $t1,$t2,$t3
```

```
        lw $t5,4($t1)
        mflo $t5
        lw $t6,0($t5)
```

d. jal or jalr+ ALU（包括乘除类）

```
        jal f1
        nop
        addu $t0,$ra,$t1
        jalr $31,$t0
        nop
        multu $31,$t0
```

e. jal or jalr+save

```
        jal f1
        sw $ra,0($0)
        jalr $20,$t1
        sw $0,0($20)
```

f. jal or jalr+load

```
        jal f1
        lw $t0,0($ra)
        jalr $21,$t1
        lw $t2,0($21)
```

2. 靠暂停解决的指令序列

a. ALU（包括乘除类）+B 类跳转

```
        ori $t0,$t0,4
        subu $t1,$t1,4
        beq $t0,$t1,f1
        nop
        mflo $t0
        bgtz $t0,f1
        bop
```

```
f1:
```

b. load+B 类跳转（暂停两拍）

```
lw $t3,0($0)
lw $t4,4($0)
beq $t3,$t4,f4
nop
```

c. load + ALU（包括乘除类）

```
lw $s5,8($0)
mult $s5,$s1
lbu $s6,12($0)
addu $k0,$s5,$s6
```

d. 乘除运算+乘除运算

```
mult $t0,$t1
mflo $s0
divu $t1,$t0
multu $t0,$s0
```

我采用的测试方法是手动构造典型样例+自动生成大量样例随机测试的方法。手动按照上述可能引发冲突的所有指令类型构造代码序列，对几个关键点进行测试，接着用自动生成的大量代码对 CPU 进行测试。

P6 上机注意事项

注意添加指令时要加到特定的指令类里面：

1．写 PC+8 的指令要加到 jump and link 类里，RiskSolveUnit 才会有操作

2．乘除运算类指令要加到对应的指令类里面（read set cal ALU），且乘在 controller 最后的部分有乘除运算单元操作选择信号(set cal ALU)，要注意添加。

增加指令类的时候更需要注意

RiskSolveUnit 里特殊处理的指令类：

1. jump and link 写 PC+8

2. calmudv setmudv readmudv mudvALU 判断暂停需要使用，一定要注意!!!!

E 级特殊处理的指令类：

1．readmudv 在 ALU 进行多路选择的时候会用到

如果新增指令涉及到这几种操作，一定要将它加入到指令类里面

注意乘除类运算指令会不会产生 start 和 busy 信号，在 controller 产生这两种信号的指令类为 calmudv、mudvALU

EMW 三个写入地址是相互独立的，若要修改其中一级，该级之后的也要同时修改

Mars 配置：

1. settings -> delaybranching √

2. settings -> memory configuration -> Compact,Data at Address0(中)

暂停功能是否成功，需要看 D 级指令

Tuse

!!! 注意仔细看指令操作，区分 Tuse_rs 和 Tuse_rt

1. CMP 需要，Tuse=0

2. ALU 需要，Tuse=1

3. DM 需要（sw 的存储数据），Tuse=2

4. 不需要新数据，Tuse=5

Tnew

1. 运算类指令，在 ALU 中产生结果的，Tnew=2

2. 跳转链接存地址类指令，与 ALU 结果同时转发，Tnew=2

3. 取 DM 类指令，在 DM 中产生结果的，Tnew=3

4. 不产生新结果的，Tnew=0

不考虑延迟槽需要的改动

1. D 级寄存器加清除信号，清除信号优先级低于阻塞信号

2. link register 时可能写入 PC+4

关于位拼接

```
Res = {16'b0,immoffset}  # 零扩展
```

```
Res = {{16{immoffset[15]}},immoffset}  # 符号扩展
```

注意事项：

宏定义的 opcode 和 funct 一定要仔细检查

复制完代码记得改宏

注意有符号数和无符号数

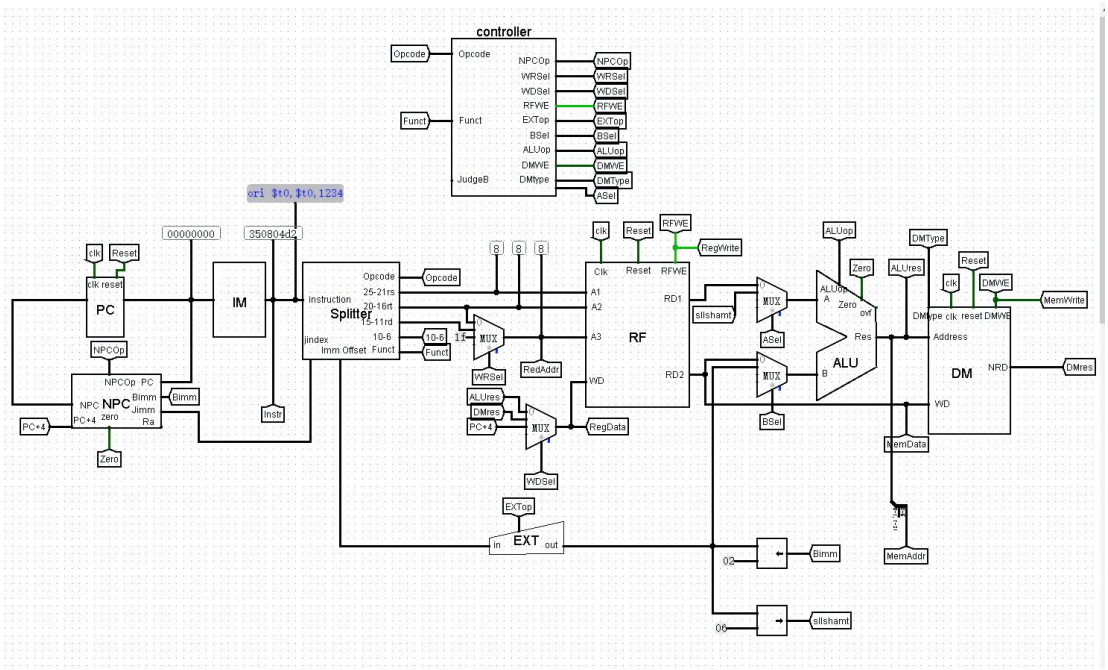
有符号数移位，注意三目运算符中的算数右移$signed($signed(A) >>> B)

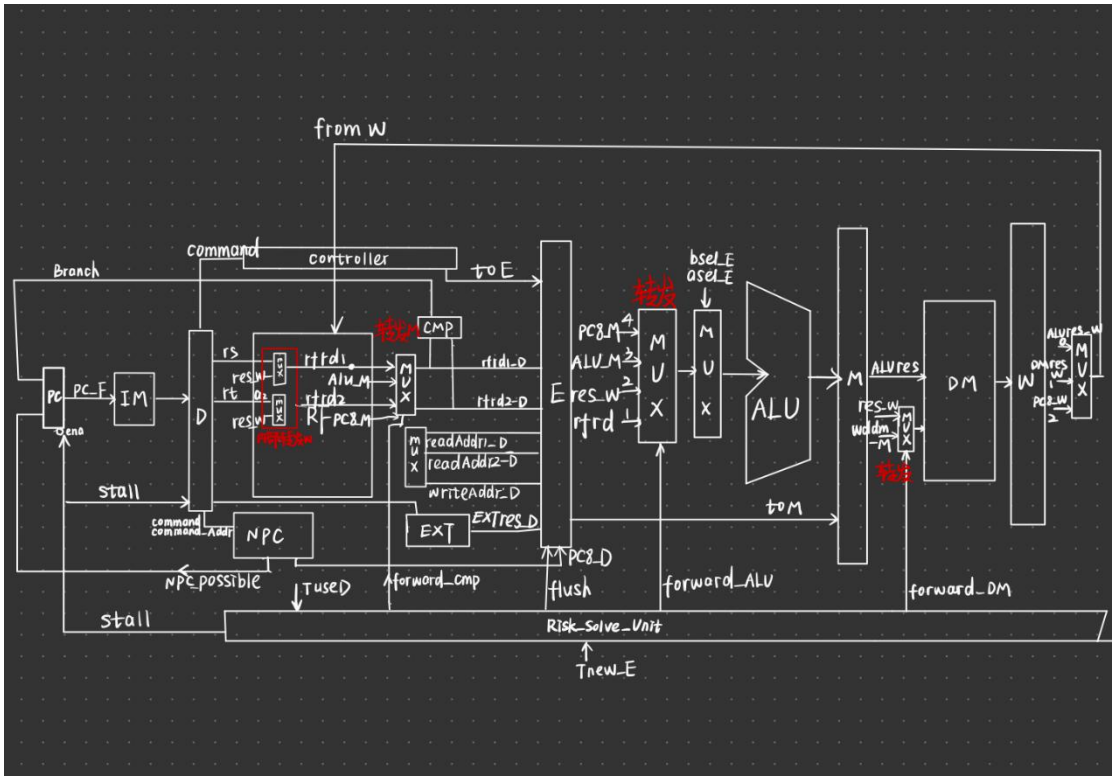
有符号数比较大小，$signed(A) > $signed(B)

有符号数{temphi, templo} <= {registerhi, registerlo} + $signed($signed(64'd0) + $signed(a) * $signed(b));

信号：

```
npcop wrsel wdsel rfwe extop cmpop aluop asel bsel dmwe dmtype tnew tuse_rs tuse_rt
```





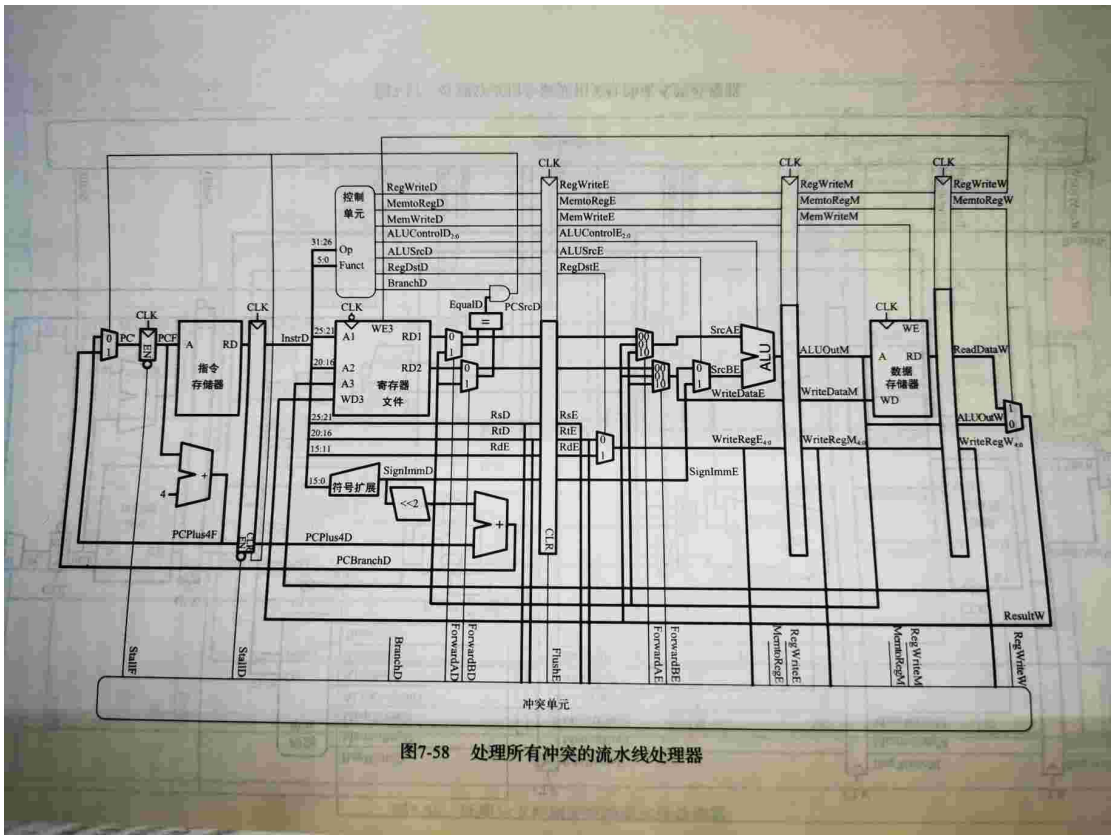


图7-58 处理所有冲突的流水线处理器