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{addu、subu、ori、lw、sw、beq、lui、j、jal、jr、nop、sll、slt、sh、lb、bgtz、jalr}共 17 条。为了实现这些功能，CPU 主要包含了 PC、IM、FlowReg_D、RF、NPC、CMP、EXT、FlowReg_E、ALU、FlowReg_M、DM、FlowReg_W、controller、RiskSolveUnit，共 14 个模块。其中 F 级包含 PC、IM 两个模块；D 级包含 RF、NPC、CMP、EXT 四个模块；E 级包含 ALU 一个模块；M 级包含 DM 一个模块。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. IM

模块功能：指令存储器

模块定义：

```
module IM (
    input  wire [31:0] address,
    output wire [31:0] command
);
```

表二 IM 信号定义

序号	信号名称	数据方向	位数	功能描述
1	Address	I	32	输入的地址
2	Command	O	32	输出地址对应的指令

在 IM 模块中，采用系统任务`$readmemh("code.txt",command_momery);`读取 `code.txt` 文件中的指令机器码，根据指令地址信号减 0x3000 并右移两位的值读取对应的地址。

3. 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 [31:0] command_F,
    input  wire [31:0] commandAddr_F,
```

```
    output reg [31:0] command_D,
    output reg [31:0] commandAddr_D
);
```

表三 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	command_F	I	32	F 级指令信号
5	commandAddr_F	I	32	F 级当前指令地址
6	Command_D	O	32	D 级指令信号
7	commandAddr_D	O	32	D 级当前指令地址

4. 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
);
```

表四 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];
```

5. 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
);
```

表五 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

6. 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
);
```

表六 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
3	branch_true	O	1	是否进行 B 类跳转

7. 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
);
```

表七 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

8. 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,
    input  wire [1:0] rfwdsel_D, asel_D, bsel_D, dmtype_D,
    input  wire [2:0] tnew_D,
    input  wire [4:0] aluop_D, read1addr_D, read2addr_D, writeaddr_D,
    input  wire [31:0] rfrd1_D, rfrd2_D, pc8_D, extres_D, command_D, commandAddr_D,
    output reg rfwe_E, dmwe_E,
    output reg [1:0] rfwdsel_E, asel_E, bsel_E, dmtype_E,
    output reg [2:0] tnew_E,
    output reg [4:0] aluop_E, read1addr_E, read2addr_E, writeaddr_E,
    output reg [31:0] rfrd1_E, rfrd2_E, pc8_E, extres_E, command_E, commandAddr_E
);
```

表八 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	rfwesel_D	I	2	D 级指令 RF 写入数据选择信号
7	asel_D	I	2	D 级指令 ALU A 操作数选择信号
8	bsel_D	I	2	D 级指令 ALU B 操作数选择信号
9	dmtype_D	I	2	D 级指令 DM 写入类型选择信号 0(word),1(half word),2(byte)
10	tnew_D	I	3	D 级指令 Tnew 信号
11	aluop_D	I	5	D 级指令 ALU 操作选择信号
12	read1addr_D	I	5	D 级指令 RF 读取地址 1
13	read2addr_D	I	5	D 级指令 RF 读取地址 2
14	writeadddr_D	I	5	D 级指令 RF 写入地址
15	rfrd1_D	I	32	D 级指令 RF 读取数据 1
16	rfrd2_D	I	32	D 级指令 RF 读取数据 2
17	pc8_D	I	32	D 级指令的地址+8
18	extres_D	I	32	D 级指令立即数扩展结果
19	command_D	I	32	D 级指令
20	commandAddr_D	I	32	D 级指令的地址
21	rfwe_E	O	1	E 级指令 RF 写使能信号
22	dmwe_E	O	1	E 级指令 DM 写使能信号
23	rfwesel_E	O	2	E 级指令 RF 写入数据选择信号

24	asel_E	O	2	E 级指令 ALU A 操作数选择信号
25	bsel_E	O	2	E 级指令 ALU B 操作数选择信号
26	dmtype_E	O	2	E 级指令 EM 写入类型选择信号 0(word),1(half word),2(byte)
27	tnew_E	O	3	E 级指令 Tnew 信号
28	aluop_E	O	5	E 级指令 ALU 操作选择信号
29	read1addr_E	O	5	E 级指令 RF 读取地址 1
30	read2addr_E	O	5	E 级指令 RF 读取地址 2
31	writeadddr_E	O	5	E 级指令 RF 写入地址
32	rfrd1_E	O	32	E 级指令 RF 读取数据 1
33	rfrd2_E	O	32	E 级指令 RF 读取数据 2
34	pc8_E	O	32	E 级指令的地址+8
35	extres_E	O	32	E 级指令立即数扩展结果
36	command_E	O	32	E 级指令
37	commandAddr_E	O	32	E 级指令的地址

9. 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
);
```

表九 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

序号	输入信号	宏定义	执行的操作
1.	0	`AaddB	两个操作数做加法
2.	1	`AsubB	两个操作数做减法
3.	2	`AorB	两个操作数按位或
4.	3	`luiB	第二个操作数的低 16 位作为输出的高 16 位，后面补零
5.	4		以下接口预留未定义
6.	5		

10. 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, dmtype_E,
```

```
    input  wire [2:0] tnew_E,
    input  wire [4:0] read2addr_E, writeaddr_E,
    input  wire [31:0] ALUres_E, wddm_E, pc8_E, command_E, commandAddr_E,
    output reg rfwe_M, dmwe_M,
    output reg [1:0] rfwdsel_M, dmtype_M,
    output reg [2:0] tnew_M,
    output reg [4:0] read2addr_M, writeaddr_M,
    output reg [31:0] ALUres_M, wddm_M, pc8_M, command_M, commandAddr_M
);
```

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	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	dmtype_E	I	2	E 级指令 DM 写入类型选择信号 0(word),1(half word),2(byte)
7	tnew_E	I	3	E 级指令 Tnew 信号
8	read2addr_E	I	5	E 级指令 RF 读取地址 2
9	writeaddr_E	I	5	E 级指令 RF 写入地址
10	ALUres_E	I	32	E 级指令 ALU 运算结果

11	wddm_E	I	32	E 级指令 DM 输入数据
12	pc8_E	I	32	E 级指令地址+8
13	command_E	I	32	E 级指令
14	CommandAddr_E	I	32	E 级指令的地址
15	rfwe_M	O	1	M 级指令 RF 写使能信号
16	dmwe_M	O	1	M 级指令 DM 写使能信号
17	rfwdsel_M	O	2	M 级指令 RF 写入数据选择信号
18	dmtype_M	O	2	M 级指令 DM 写入类型选择信号 0(word),1(half word),2(byte)
19	tnew_M	O	3	M 级指令 Tnew 信号
20	read2addr_M	O	5	M 级指令 RF 读取地址 2
21	writeaddr_M	O	5	M 级指令 RF 写入地址
22	ALUres_M	O	32	M 级指令 ALU 运算结果
23	wddm_M	O	32	M 级指令 DM 输入数据
24	pc8_M	O	32	M 级指令地址+8
25	command_M	O	32	M 级指令
26	CommandAddr_M	O	32	M 级指令的地址

11. DM

模块功能：数据存储器

模块定义：

```
module DM (
    input  wire clk,
    input  wire reset,
    input  wire dmwe_M,
    input  wire [31:0] dmAddr,
    input  wire [31:0] wd,
    input  wire [1:0] dmtype,
    input  wire [31:0] commandAddr_M,
    output wire [31:0] dmrd
);
```

表十一 数据存储器信号定义

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	reset	I	1	DM 同步复位信号
3	dmwe_M	I	1	内存写入使能信号，1 写入有效
4	dmAddr	I	32	32 位地址输入信号
5	wd	I	32	32 位数据输入信号
6	dmtype	I	2	内存操作类型：0(word),1(half word),2(byte)
7	commandAddr_M	I	32	当前指令地址，评测需要
7	dmrd	O	32	输出Address指定地址对应的32 位数据

12. 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,
    input  wire [4:0] writeaddr_M,
    input  wire [31:0] ALUres_M, DMRes_M, pc8_M, commandAddr_M,
    output reg rfwe_W,
    output reg [1:0] rfwdsel_W,
    output reg [4:0] writeaddr_W,
    output reg [31:0] ALUres_W, DMRes_W, pc8_W, commandAddr_W
);
```

表十二 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
	reset	I	1	W 级寄存器复位信号
	rfwe_M	I	1	M 级指令 RF 写使能信号
	rfwdsel_M	I	2	M 级指令 RF 写入数据选择信号
	writeaddr_M	I	5	M 级指令 RF 写入地址
	ALUres_M	I	32	M 级指令 ALU 运算结果
	DMRes_M	I	32	M 级指令 DM 输出数据
	pc8_M	I	32	M 级指令地址+8

	commandAddr_M	I	32	M 级指令地址
	rfwe_W	O	1	W 级指令 RF 写使能信号
	rfwdsel_W	O	2	W 级指令 RF 写入数据选择信号
	writeaddr_W	O	5	W 级指令 RF 写入地址
	ALUres_W	O	32	W 级指令 ALU 运算结果
	DWRes_W	O	32	W 级指令 DW 输出数据
	pc8_W	O	32	W 级指令地址+8
	commandAddr_W	O	32	W 级指令地址

13. Controller

模块功能：控制器

模块定义：

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

表十三 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
	rfwe_D	O	1	D 级指令 RF 写使能信号

	dmwe_D	O	1	D 级指令 DM 写使能信号
	wrsel_D	O	2	D 级指令 RF 写入地址选择信号
	wdsel_D	O	2	D 级指令 RF 写入数据选择信号
	extop	O	2	D 级指令位扩展方法
	asel_D	O	2	D 级指令 ALU A 操作数选择信号
	bsel_D	O	2	D 级指令 ALU B 操作数选择信号
	dmtype_D	O	2	D 级指令 DM 写入类型选择信号 0(word),1(half word),2(byte)
	npcop	O	3	D 级指令的下一条指令地址计算操作类型
	tuse_Drs	O	3	D 级指令使用 RF 的 rs 地址对应的数据的时间
	tuse_Drt	O	3	D 级指令使用 RF 的 rt 地址对应的数据的时间
	tnew_D	O	3	D 级指令产生新的写入 RF 的时间
	cmpop	O	4	D 级指令 cmp 操作类型选择信号
	aluop_D	O	5	D 级指令 ALU 操作类型选择信号

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。

14. 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,
    input  wire [4:0] read1addr_D, read2addr_D, read1addr_E, read2addr_E,
read2addr_M, writeaddr_E, writeaddr_M, writeaddr_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, flush_E,

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

表十四 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
	rfwe_E	I	1	E 级指令 RF 写使能信号
	rfwe_M	I	1	M 级指令 RF 写使能信号
	rfwe_W	I	1	W 级指令 RF 写使能信号
	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
	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
	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

	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
	read2addr_M	I	5	即将参与 M 级 DM 写入的数据对应的寄存器地址
	writeaddr_E	I	5	E 级结果写入的 RF 地址
	writeaddr_M	I	5	M 级结果写入的 RF 地址
	writeaddr_W	I	5	W 级结果写入的 RF 地址
	forward_CMPa_D	O	2	D 级 CMP A 操作数转发选择信号
	forward_CMPb_D	O	2	D 级 CMP B 操作数转发选择信号
	forward_ALUa_E	O	2	E 级 ALU A 操作数转发选择信号
	forward_ALUb_E	O	2	E 级 ALU B 操作数转发选择信号
	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				
	tuse_Drs	I	3	D 级指令 GRF[rs]的使用时间
	tuse_Drt	I	3	D 级指令 GRF[rt]的使用时间
	tnew_E	I	3	E 级指令得到新的向 RF 写入的数据的时间

	tnew_M	I	3	M 级指令得到新的向 RF 写入的数据的时间
	stall_F	O	1	PC 寄存器冻结信号
	stall_D	O	1	D 级流水线寄存器冻结信号
	flush_E	O	1	E 级流水线寄存器清除信号
	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，根据指令信号判断指令类型，并得出指令的控制信号。下面列出了和逻辑和或逻辑真值表，并根据真值表进行指令信号的解码和控制信号的选取和输出。

1. 指令信号解码

R 型指令（addu subu）：前六位均为 0，可以通过后五位编码

I J 型指令（ori lw sw beq lui j jal jr）：可以直接通过前六位编码

Nop 指令：全零

在 Verilog 实现的过程中，我们可以先根据 command[31:26]是否为零，将指令分为 R 型和 IJ 型两类，再根据具体的 OPCode 和 Funct 进行编码。

PART1 OPCode == 0

```
`define nop    6'b000000 // op == 0
`define addu   6'b100001 // op == 0
`define subu   6'b100011 // op == 0
`define jr     6'b001000 // op == 0
`define sll    6'b000000 // op == 0
`define slt    6'b101010 // op == 0
```

PART2 OPCode != 0

```
`define  ori     6'b001101  // op != 0
`define  lw      6'b100011  // op != 0
`define  sw      6'b101011  // op != 0
`define  beq     6'b000100  // op != 0
`define  lui     6'b001111  // op != 0
`define  jal     6'b000011  // op != 0
`define  j       6'b000010  // op != 0
`define  sh      6'b101001  // op != 0
`define  lb      6'b100000  // op != 0
`define  bgtz    6'b000111  // op != 0
```

2. 控制信号输出

这部分是各指令对应的控制信号真值表，与 P4 相比，P5 增加的控制信号为 Tuse_Drs,Tuse_Drt 以及 Tnew，分别表示 D 级指令使用 GPR[rs], GPR[rt]以及产生新的写入到寄存器中的数据的时间。对于不使用寄存器中数据的指令，将它的 Tuse_Drs,Tuse_Drt 设置成 5；对于不产生新的写入到寄存器中数据的指令，将它的 Tnew 设置为 0。

表十五 与逻辑真值表（一）

	npcop	wrsel_D	wdsel_D	rfwe_D	extop_D	asel_D	bsel_D
addu	0	1	0	1	x	0	0
subu	0	1	0	1	x	0	0
ori	0	0	0	1	0	0	1
lw	0	0	1	1	1	0	1
sw	0	x	x	0	1	0	1
beq	1	x	x	0	1	x	x

lui	0	0	0	1	0	0	1
jal	2	2	2	1	x	0	x
jr	3	x	x	0	x	0	x
j	2	x	x	0	x	x	x
nop	0	x	x	0	x	x	x
sll	0	1	0	1	0	1	0
slt	0	1	0	1	x	0	0
sh	0	x	x	0	1	0	1
lb	0	0	1	1	1	0	1
bgtz	1	x	x	0	1	x	x
jalr	3	1	2	1	0	0	0

表十五 与逻辑真值表（二）

	aluop_D	dmwe_D	dmtype_D	tnew_D	tuse_Drs	tuse_Drt	cmp_op
addu	0	0	x	2	1	1	x
subu	1	0	x	2	1	1	x
ori	2	0	x	2	1	5	x
lw	0	0	0	3	1	5	x
sw	0	1	0	0	1	2	x
beq	x	0	x	0	0	0	0
lui	3	0	x	2	1	5	x
jal	x	0	x	2	5	5	x
jr	x	0	x	0	0	5	x
j	x	0	x	0	5	5	x
nop	x	0	x	0	5	5	x
sll	4	0	x	2	5	1	x
slt	5	0	x	2	1	1	x

sh	0	1	1	0	1	2	x
lb	0	0	2	3	1	5	x
bgtz	x	0	x	0	0	0	1
jalr	0	0	0	2	0	5	0

（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
addu $a3,$v1,$a1  # 负负
addu $t0,$a2,$a3  # 正负
addu $t1,$a3,$a2  # 负正

subu $t2,$a2,$at  # 正正
subu $t3,$a1,$a3  # 负负
subu $t4,$a2,$a3  # 正负
subu $t5,$a3,$t4  # 负正

ori $t6,0
```

```
ori $t7,1
nop
sw $t2,0($t6)
sw $t3,8($t6)
sw $t4,12($t6)
sw $t5,4($t6)

ori $k0,0
ori $k1,1
nop
lw $s0,12($t6)
lw $s1,8($t6)
lw $s2,4($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

  # ALU+sw
addu $t3,$t1,$t0
subu $t4,$t0,$t3
addu $t5,$t3,$t4
```

```
sw $t5,0($0)

  # ALU+sw
ori $t6,$t6,100
sw $6,4($0)
lui $t7,100
sw $t7,8($0)

  Jal+sw
jal f1
sw $ra,12($0)

  Jal+ALU
f1:
jal f2
addu $s0,$t7,$ra

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

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

3. 暂停测试

```
  # ALU+beq
ori $t0,$t0,4
ori $t1,$t1,4
beq $t0,$t1,f1
nop
ori $s0,10

  # Jal+Beq
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)

# lw+beq（暂停两拍）
lw $t3,0($0)
lw $t4,4($0)
beq $t3,$t4,f4
```

```
    nop

      # lw+ALU
    f4:
    lw $s5,8($0)
    lw $s6,12($0)
    addu $k0,$s5,$s6

      # lw+sw
    lw $s5,0($0)
    sw $s6,0($s5)

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

4. 跳转及延迟槽测试

```
ori $t0,$t0,4
ori $t1,$t1,5
ori $t2,$t2,4

  # beq跳转
beq $t0,$t1,f1
addu $s0,$t0,$t2
addu $s1,$t1,$t3

f1:
subu $t4,$t1,$t0
```

```
  # jal跳转
jal f2
lui $t5,100
ori $t5,300
  # beq跳转
beq $0,$0,end

f2:
addu $t6,$ra,$t2
ori $k0,$k0,10
ori $k1,$k1,20

f3:
  # beq跳转
beq $k0,$k1,f4
addu $t6,$t5,$k0
addu $k0,$k0,$k0
  # 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

测试样例生成器，编写过程中采用了讨论区同学的思路，生成的指令包含 MIPS-lite2={ addu, subu, ori, lw, sw, beq, lui, j, jal, jr, nop }的所有指令，含小 bug（sw 指令向 DM 存储的 index 可能会超出 DM 的地址范围，jr 指令跳转会出问题）有可能不能在 Mars 上正常运行，产生的代码可用于和其它同学的 CPU 对拍。

```
import random


data1 = []  # 记录顺序执行的代码
data2 = []  # 记录跳转代码
interval = []  # 记录中间插入部分代码的行数
helper = []  # 记录 sw 和 lw 之前 ori 指令的相关内容
acnt = 0  # index of helper



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(101):
```

```
        print("sw ${},{}(${})".format(random.randint(0, 27), i * 4, 0))



def print_instr(x):
    if data1[x].type == 1:
        print("addu ${},${},${}".format(data1[x].rs, data1[x].rt, data1[x].rd))
    elif data1[x].type == 2:
        print("subu ${},${},${}".format(data1[x].rs, data1[x].rt, data1[x].rd))
    elif data1[x].type == 3:
        print("ori ${},${},{}".format(data1[x].rs, data1[x].rt, data1[x].rd))
    elif data1[x].type == 4:
        print("lui ${},{}".format(data1[x].rs, data1[x].rt))
    elif data1[x].type == 5:
        print("nop")
    elif data1[x].type == 6:
        y = data1[x].oriindex
        print("ori ${},$0,{}".format(helper[y].rs, helper[y].rt))
        print("lw ${},{}(${})".format(data1[x].rs, data1[x].rd, data1[x].rt))
    elif data1[x].type == 7:
        y = data1[x].oriindex
        print("ori ${},$0,{}".format(helper[y].rs, helper[y].rt))
        print("sw ${},{}(${})".format(data1[x].rs, data1[x].rd, data1[x].rt))



if __name__ == "__main__":
    pre_fill()
    total_blocks = random.randint(1, 15)  # 代码块数
    total_lines = 0  # 代码行数

    # 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, 2), random.randint(1, 20)))
    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, 20))
    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, 7)))  # 5->nop
    if data1[i].type == 1 or data1[i].type == 2:  # addu subu
        data1[i].rs = random.randint(2, 27)
        data1[i].rt = random.randint(2, 27)
        data1[i].rd = random.randint(2, 27)
    elif data1[i].type == 3:  # ori
        data1[i].rs = random.randint(2, 27)
        data1[i].rt = random.randint(2, 27)
        data1[i].rd = random.randint(0, 65535)
    elif data1[i].type == 4:  # lui
        data1[i].rs = random.randint(2, 27)
        data1[i].rt = random.randint(0, 65535)
    elif data1[i].type == 6 or data1[i].type == 7:  # 6->lw  7->sw
        data1[i].oriindex = acnt
        acnt += 1
        helper.append(Instruction(3))  # 构建 ori 指令，保证位置是 4 的倍数
        helper[acnt - 1].rs = random.randint(2, 27)
        helper[acnt - 1].rt = 4 * random.randint(0, 100)
        data1[i].rs = random.randint(2, 27)
        data1[i].rt = helper[acnt - 1].rs
        if helper[acnt - 1].rt >= 50:
            data1[i].rd = -4 * random.randint(0, 12)  # offset 是 4 的倍数
        else:
```

```
                data1[i].rd = 4 * random.randint(0, 12)


        for i in range(total_blocks):
            if data2[i].type == 0:  # data2[i]==0-->beq  data2[i]==1-->jal  data2[i]==2-->j
                coin = random.randint(0, 2)
                if coin != 0:
                    data2[i].rs = data2[i].rt = random.randint(0, 27)  # coin !=0 保证 rs=rt
跳转
                else:
                    data2[i].rs = random.randint(0, 27)
                    data2[i].rt = random.randint(0, 27)


        print("ori $29,$0,0x00002F00")


        # 输出过程
        k = 0
        for i in range(total_blocks):
            for j in range(interval[i]):  # 输出 interval
                print_instr(k)
                k += 1
            if data2[i].type == 0:  # 输出 data2[i]
                # print("beq ${},${},branch{}".format(data2[i].rs, data2[i].rt, i))
                print("ori $3,$3,100")  # 延迟槽
                print("ori $4,$4,200")
                for j in range(data2[i].lines):
                    print_instr(k)
                    k += 1
                print("branch{}:".format(i))
            elif data2[i].type == 1:
                print("ori $2,$0,4")
```

```
            print("subu $sp,$sp,$2")
            print("sw $ra,4($sp)")
            print("jal func{}".format(i))
            print("ori $3,$3,100")  # 延迟槽
            print("ori $4,$4,200")
            print("ori $2,$0,4")
            print("lw $ra,4($sp)")
            print("addu $sp,$sp,$2")
        elif data2[i].type == 2:
            print("j func{}".format(i))
            print("ori $3,$3,100")  # 延迟槽
            print("ori $4,$4,200")
    # print("jr $ra")
    for i in range(total_blocks):
        if data2[i].type == 1 or data2[i].type == 2:
            print("func{}:".format(i))
            for j in range(data2[i].lines):
                k += 1
                print_instr(k)
        if data2[i].type == 1:
            print("ori $2,$0,4")
            print("addu $sp,$sp,$2")
            print("lw $ra,4($sp)")
            print("jr $ra")
            print("ori $3,$3,100")  # 延迟槽
            print("ori $4,$4,200")


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

2. 自动执行脚本

```
import os
import re


def gen(num):
    os.system(r"python create.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）

    # 仿真需要的两个文件
    tclFile = open("test.tcl", "w")
    tclFile.write("run 1us;\nexit")

    prjFile1 = open("testmine.prj", "w")
    for root, dirs, files in os.walk(r"E:\1-Project\P5\test\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\P5\test\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_tb >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_tb >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(1):
            gen(str(i))
```

3. 正确性判定脚本

```
    import re
    import sys



    def cmp(num):
        # 读取三个文本文件
        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()

        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 = []

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

        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])

    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] and operation2[i] == operation3[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]))

    if flag:
        print("完全正确")


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

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。

从数据通路方面，PC 值的更新有四种不同的情况：第一种情况为顺序执行的情况，PC_F 加四即可；第二种为分支跳转，PC 更新为 commandAddr_D + 4 + {{14{command_D[15]}},{command_D[15:0]},2'd0}；第三种情况为立即数跳转指令，PC 更新为{commandAddr_D[31:28],command_D[25:0],2'd0}；第四种情况为跳转寄存器指令，PC 更新为对应寄存器存储的 32 位数据。

从控制信号的方面，PC 更新所需要的控制信号即为对上述四条数据通路的选择信号。在我的 CPU 实现过程中，controller 模块分析指令并生成 npcop 信号输入到 PC 模块，对其中的四条数据通路进行选择。具体来说，如果 npcop 为 0，PC 选择第一条通路的数据；npcop 为 1，PC 选择第二条通路的数据，以此类推……值得注意的是，PC 还需要接入 CMP 输出的跳转指令判断结果作为控制信号。当 D 级指令为 B 类分支跳转指令时，CMP 模块会根据指令的判断条件和操作数得出是否要执行跳转，当判断结果成立时，PC 更新为上述第二条数据通路中的数据，结果不成立时，PC 仍然按照第一种情况直接输入 F 级指令地址加四的结果。

（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？

需要将地址写入到寄存器中的指令包括 J、Jal 以及 B 类的一些跳转并链接指令，这些指令在流水线 CPU 的 D 级会根据当前地址计算向 RF 输入的数据。由于我们设计的流水线 CPU 含有延迟槽，也就是说无论跳转是否发生，跳转下一条指令都必将被执行，所以地址为 PC+4 的指令必将被执行，PC+8 是不跳转相对于跳转产生差异的第一条指令的地址，所以我们要向寄存器回写 PC+8，这样才能保证 Jr 指令返回同一寄存器保存的地址时，PC+4 地址对应的指令不会被重复执行。

（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？

流水线 CPU 的优势就在于它执行指令的速度较快，转发数据的供给者采用上一级流水线寄存器中的数据也是为了提高流水线 CPU 处理指令的速度。

ALU 和 DM 等功能部件，其自身结构较为复杂，执行得到正确结果需要一定的时间。如果我们采用 ALU 或者 DM 等部件提供数据，等待这些部件得到正确结果的时间就会被白白浪费掉。更重要的是，这些时间还会被附加在等待转发的模块所在的流水级，相当于这两个流水级叠加到了一起，会使得该流水级所需时间增长，可能会导致流水线 CPU 的最短时钟周期变长，从而丧失流水线 CPU 的优势。相反，如果我们采用存储了上一级传来的各种数据的流水级寄存器作为供给者，就可以直接得到数据，流水线 CPU 的最小时钟周期会小于采用 ALU 或者 DM 等部件提供数据的流水线 CPU。

从本质上讲，流水线 CPU 提高运行速度的方式就是将单周期 CPU 的各个步骤割裂开来，在每条指令在单个时钟周期只执行一个步骤，这样，我们就可以达到将 CPU 的一个运行周期分摊给 5 条指令的效果，这就要求 CPU 的每一级是独立的，数据只能在流水级外部传递。而如果我们在转发时采用的是由 ALU 或者 DM 等流水级内部功能部件提供的数据，相当于人为的将这两个流水级合并到了一起，没有达到我们想要的 5 条指令分摊一个 CPU 运行周期的效果，这与流水线 CPU 的设计初衷是相悖的。

（四）

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。

直接采用上一级的原始数据相当于完全没有考虑数据冲突问题，很多指令序列都会引发这个问题，以下列举几种可能会引发这一问题的指令序列：

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


  # ALU+sw
addu $t3,$t1,$t0
subu $t4,$t0,$t3
addu $t5,$t3,$t4
sw $t5,0($0)


  # ALU+beq
```

```
ori $s3,100
ori $s4,100
beq $s3,$s4,f4
ori $s5,$s5,235
ori $s6,$s6,794
f4:

# lw+beq（暂停两拍）
lw $t3,0($0)
lw $t4,4($0)
beq $t3,$t4,f4
nop

  # lw+ALU
lw $s5,8($0)
lw $s6,12($0)
addu $k0,$s5,$s6

  # lw+sw(暂停)
lw $s5,0($0)
sw $s6,0($s5)

  # lw+sw（转发不暂停）
lw $t0,4($0)
lw $t1,8($0)
sw $t1,0($t0)
```

此外 Jal 指令和以$31 为任一操作数的计算指令，条件跳转指令构成的指令序列也会出现这种问题

（五）

我们为什么要对 GRF 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？

GRF 是一个特殊的部件，它既可以视为 D 级的一个部件，也可以视为 W 级之后的流水线寄存器。因此我们可以对 GRF 进行内部转发。

GRF 内部转发的好处是无需在 RiskSolveUnit 模块对 W 级回写寄存器进行转发判断，相当于 W 级的回写寄存器数据直接对 D 级可见，减少了很多判断信号和选择信号。RF 内部转发的实现代码为：

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

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

（六）

为什么 0 号寄存器需要特殊处理？

因为 0 号寄存器中的数值始终为 0，不受任何写入数据的影响。假如我们不对 0 号寄存器做特殊处理，若流水线 CPU 中未写入 RF 的前一条指令已经得到向 0 号寄存器的写入数据（非 0），当前指令恰好需要读取 0 号寄存器中的数据，这样便会触发转发，前一条指令的非零数据会被转发到当前指令的数据读取端，这显然是不对的，因此我们要对 0 号寄存器做特殊处理。

（七）

什么是“最新产生的数据”？

最新产生的数据即最后产生的数据，是将流水线 CPU 从 D 级断开，RF 寄存器中的最终数据。在流水线 CPU 中，最新产生的数据是流水线 CPU 中最低流水级保存的 RF 回写数据。在向 E 级 ALU 中回写数据时，我们需要选取最新产生的回写数据作为转发数据。当 M 级和 W 级均有回写数据产生时，我们优先选择 M 级的数据作为转发数据，因为 M 级相对于 W 级是最新产生的数据。

（八）

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？

我个人在转发条件时加入了对 rfwe 的判断，如果想要达到“供给者需求者的 A 相同，且不为 0”即转发需要在 D 级进行指令解码后即根据 rfwe_D 对寄存器写入地址进行一次选择。如果 rfwe_D 为 0，就将写入地址赋成 0；rfwe_D 为 1，就输出原本的写入地址。这样就不用把 rfwe 信号随流水线一级级传下去了，RF 的写入也不需要 rfwe 信号了，能够降低设计的复杂度。

（九）

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
```

b. ALU+sw

```
addu $t5,$t3,$t4
sw $t5,0($0)

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

```
        sw $t5,4($t1)
```

c. ALU+lw

```
        addu $t1,$t2,$t3
        lw $t5,4($t1)
```

d. jal+ALU

```
        jal f1
        addu $t0,$ra,$t1
```

e. jal+sw

```
        jal f1
        sw $ra,0($0)

        jal f2
        sw $0,0($ra)
```

f. jal+lw

```
        jal f1
        lw $t0,0($ra)
```

2. 靠暂停解决的指令序列

a. ALU+beq

```
        ori $t0,$t0,4
        ori $t1,$t1,4
        beq $t0,$t1,f1
        nop
        ori $s0,10
```

b. lw+beq（暂停两拍）

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

c. lw+ALU

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

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.