计算机组成原理实验报告

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，支持的指令集包含{addu、subu、addi、ori、lw、sw、beq、lui、sll、srl、sra、slt、j、jal、jr、jalr、nop}。为了实现这些功能，CPU 在顶层模块 mips 下并列包含了 DATAPATH、FORWARD_CONTROL、STALL_CONTROL 三个模块。

DATAPATH 模块下分五个流水级 F、D、E、M、W。F 级包含了 PC、IM 部件，D 级包含了 GRF、EXT、NPC、CMP 部件；E 级包含了 ALU 部件；M 级包含了 DM 部件；W 级连接到 D 级寄存器。相邻两流水级之间还各设置了一个流水线寄存器 REG_D、REG_E、REG_M、REG_W，用于存储流水的信息。此外，为满足数据冒险的转发需求，还设置了六个转发多路选择器：D 级 MF_GRFRD1_D、MF_GRFRD2_D 分别用于选择需进入 CMP 部件和 E 级寄存器的两个寄存器值；E 级 MF_ALUA_E 用于选择参与 ALU 运算的第一个数据，MF_ALUB_E 用于选择参与 ALU 运算的第二个来自寄存器的数据，MF_MemData_E 用于选择进入 M 级寄存器的数据；MF_DMWD_M 用于选择写入 DM 的数据。

FORWARD_CONTROL 模块用于生成转发 MUX 的选择控制信号。

STALL_CONTROL 模块用于生成暂停信号。

此外，工程文件中包含了名为 MACRO 的.v 文件用于宏定义。本流水线 CPU 主要采用分布式译码，设置一个专门的 DECODER 模块，在 D、E、M、W 级流水线寄存器分别实例化，传入各级指令，同时传出该级所需要的控制信号。

（二）关键模块定义

1. PC（F 级）

当前 PC 值的存储由一个 32 位 reg 变量保存。PC 模块端口定义如下：

信号名	方向	位宽	描述
clk	I	1	时钟信号
Pc_en	I	1	Pc 写使能信号
reset	I	1	同步复位信号
NPC	I	32	由 NPC 模块计算的下一个 PC 值
PC_F	O	32	当前的 PC
PC_4F	O	32	当前的 PC + 4
PC_8F	O	32	当前的 PC + 8

端口 Verilog 声明如下：

```
module PC(
    input clk,
    input pc_en,
    input reset,
    input [31:0] NPC,
    output [31:0] PC_F,
    output [31:0] PC4_F,
    output [31:0] PC8_F
    );
```

2. IM（F 级）

IM 端口声明如下：

信号名	方向	位宽	描述
PC	I	32	当前 PC 值
IR	O	32	当前指令序列

端口 Verilog 声明如下：

```
module IM(
    input [31:0] PC,
    output [31:0] IR
    );
```

IM 内部 32 bit * 4096 字的存储器的具体实现如下：

```
reg [31:0] im[0:4095];
```

3. REG_D（D 级）

REG_D 模块端口定义如下：

信号名	方向	位宽	描述
IR_F	I	32	F 级 32 位指令序列
pc4_F	I	32	F 级的 pc + 4
Pc8_F	I	32	F 级的 pc + 8
D_en	I	1	D 级寄存器写使能信号
Clk	I	1	时钟信号
Reset	I	1	同步复位信号
IR_D	O	32	D 级 32 位指令序列
Pc4_D	O	32	D 级的 pc + 4
Pc8_D	O	32	D 级的 pc + 8

端口 Verilog 声明如下：

```
module REG_D(
    input [31:0] IR_F,
    input [31:0] pc4_F,
    input [31:0] pc8_F,
    input D_en,
    input clk,
    input reset,
    output reg [31:0] IR_D,
    output reg [31:0] pc4_D,
    output reg [31:0] pc8_D
    );
```

4. D_decoder（D 级）

该模块为 DECODER.v 译码器在 D 级的实例化，专门产生 D 级的控制信号。

D_decoder 模块端口定义如下：

信号名	方向	位宽	描述
IR_D	I	32	D 级 32 位指令序列
A1_D	O	5	D 级指令对应传入 GRF A1 端口的寄存器编号
A2_D	O	32	D 级指令对应传入 GRF A2 端口的寄存器编号
EXTOp	O	1	EXT 控制信号
NPCOp	O	3	NPC 控制信号

端口 Verilog 实例化如下：（DECODER 模块声明见后）

```
DECODER d_decoder (
    .IR(IR_D),
    .A1(A1_D),
    .A2(A2_D),
    .EXTOp(EXTOp),
    .NPCOp(NPCOp)
);
```

5. NPC（D 级）

NPC 模块端口定义如下：

信号名	方向	位宽	描述
IR_D	I	32	D 级 32 位指令序列
NPCOp	I	3	NPC 控制信号
Pc_F	I	32	F 级的 pc
Pc4_D	I	32	D 级的 pc + 4
Pc8_D	I	32	D 级的 pc + 8
GPR_rs	I	32	GRF RD1 端口读出的值
Zero	I	1	CMP 的判零信号
npc	O	32	下一个 pc 值，传入 pc

NPCOp 控制信号真值表如下：

NPCOp[2:0]	功能	NPCOp[2:0]	功能
000	NPC = pc_F + 4	100	\
001	BEQ 指令 NPC 计算	101	\
010	跳转到 26 位立即数	110	\
011	跳转到 GPR_rs 寄存器	111	\

端口 Verilog 声明如下：

```
module NPC(
    input [31:0] IR_D,
    input [2:0] NPCOp,
    input [31:0] pc_F,
    input [31:0] pc4_D,
    input [31:0] pc8_D,
    input [31:0] GPR_rs,
    input zero,
    output reg [31:0] npc
    );
```

6. EXT

EXT 模块端口定义如下：

信号名	方向	位宽	描述
EXTOp	I	1	扩展信号选择
IR_D	I	32	D 级 32 位指令序列
EXTResult	O	32	扩展后的 32 位信号

端口 Verilog 声明如下：

```
module EXT(
    input [31:0] IR_D,
    input EXTOp,
    output [31:0] EXTResult
    );
```

7. GRF

GRF 中包含 32 个 32 位寄存器，分别对应 0~31 号寄存器，其中 0 号寄存器读取的结果恒为 0。具体模块端口定义如下：

信号名	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
RegWrite_W	I	1	写使能信号
Pc4_W	I	32	W 级的 pc + 4
Pc4_D	I	32	D 级的 pc + 4
Pc8_D	I	32	D 级的 pc + 8
A1_D	I	5	读寄存器的地址
A2_D	I	5	读寄存器的地址
A3_W	I	5	写寄存器的地址
RegData	I	32	32 位写入数据
RD1	O	32	32 位读出数据
RD2	O	32	32 位读出数据

端口 Verilog 声明如下：

```
module GRF(  // support "internal forwarding"
    input clk,
    input reset,
    input RegWrite_W,
    input [31:0] pc4_W,
    input [31:0] pc4_D,
    input [31:0] pc8_D,
    input [4:0] A1_D,
    input [4:0] A2_D,
    input [4:0] A3_W,
    input [31:0] RegData,
    output reg [31:0] RD1,
    output reg [31:0] RD2
    );
```

GRF 内部 32 bit * 32 的寄存器的具体实现如下：

```
reg [31:0] register[0:31];
```

8. CMP

CMP 模块端口定义如下：

信号名	方向	位宽	描述
V1	I	32	来自转发 mux MF_GRFRD1_D 的输出信号
V2	I	32	来自转发 mux MF_GRFRD2_D 的输出信号
zero	O	32	判零信号

端口 Verilog 声明如下：

```
module CMP(
    input [31:0] V1,
    input [31:0] V2,
    output zero
    );
```

9. REG_E（E 级）

REG_E 模块端口定义如下：

信号名	方向	位宽	描述
IR_D	I	32	D 级 32 位指令序列
V1	I	32	来自转发 mux MF_GRFRD1_D 的输出信号
V2	I	32	来自转发 mux MF_GRFRD2_D 的输出信号
EXTResult	I	32	扩展后的 32 位信号
pc4_D	I	32	D 级的 pc + 4
Pc8_D	I	32	D 级的 pc + 8
E_clr	I	1	E 级寄存器清除信号
Clk	I	1	时钟信号

Reset	I	1	同步复位信号
IR_E	O	32	E 级 32 位指令序列
V1_E	O	32	来自转发 mux MF_GRFRD1_D 的输出信号流水到 E 级
V2_E	O	32	来自转发 mux MF_GRFRD2_D 的输出信号流水到 E 级
EXTResult_E	O	32	扩展后的 32 位信号流水到 E 级
Pc4_E	O	32	E 级的 pc + 4
Pc8_E	O	32	E 级的 pc + 8

端口 Verilog 声明如下：

```
module REG_E(
    input [31:0] IR_D,
    input [31:0] V1,
    input [31:0] V2,
    input [31:0] EXTResult,
    input [31:0] pc4_D,
    input [31:0] pc8_D,
    input clk,
    input reset,
    input E_clr,
    output reg [31:0] IR_E,
    output reg [31:0] V1_E,
    output reg [31:0] V2_E,
    output reg [31:0] EXTResult_E,
    output reg [31:0] pc4_E,
    output reg [31:0] pc8_E
    );
```

10. E_decoder（E 级）

该模块为 DECODER.v 译码器在 E 级的实例化，专门产生 E 级的控制信号。

E_decoder 模块端口定义如下：

信号名	方向	位宽	描述
IR_E	I	32	E 级 32 位指令序列
ALU_Aop	O	1	ALU 中 ALUSrcA 端口选择信号

ALU_Bop	O	2	ALU 中 ALUSrcB 端口选择信号
ALUOp	O	4	ALU 控制信号
Shamt_E	O	5	E 级指令的 shamt 字段

端口 Verilog 实例化如下：（DECODER 模块声明见后）

```
DECODER e_decoder (
    .IR(IR_E),
    .ALU_Aop(ALU_Aop),
    .ALU_Bop(ALU_Bop),
    .ALUOp(ALUOp),
    .shamt(shamt_E)
);
```

11. ALU（E 级）

ALU 支持 AND、OR、ADD、SUB、SLL、SRL、SRA、SLT（有符号）等运算，其中 ADD、SUB 运算不支持溢出检测。

ALU 端口声明如下：

信号名	方向	位宽	描述
ALUSrcA	I	32	ALU 操作数 1
ALUSrcB	I	32	ALU 操作数 2
ALUOp	I	4	ALU 功能选择信号
Shamt_E	I	5	来自 E 级译码器的 shamt_E
ALUResult_E	O	32	ALU 运算结果

端口 Verilog 声明如下：

```
module ALU(
    input [31:0] ALUSrcA,
    input [31:0] ALUSrcB,
    input [3:0] ALUOp,
    input [4:0] shamt_E,
    output reg [31:0] ALUResult_E
    );
```

ALU 控制信号真值表如下：

ALUOp[3:0]	功能	ALUOp[3:0]	功能
0000	A AND B	1000	\
0001	A OR B	1001	\
0010	A + B	1010	\
0011	A - B	1011	\
0100	A 逻辑左移 B	1100	\
0101	A 逻辑右移 B	1101	\
0110	A 算数右移 B	1110	\
0111	SLT	1111	\

12. REG_M（M 级）

REG_M 模块端口定义如下：

信号名	方向	位宽	描述
IR_E	I	32	E 级 32 位指令序列
V2_E	I	32	来自转发 mux MF_GRFRD2_D 的输出信号流水到 E 级的信号
ALUResult_E	I	32	ALU 计算结果
pc4_E	I	32	E 级的 pc + 4
Pc8_E	I	32	E 级的 pc + 8
Clk	I	1	时钟信号

Reset	I	1	同步复位信号
IR_M	O	32	M 级 32 位指令序列
V2_M	O	32	来自转发 mux MF_GRFRD2_D 的输出信号流水到 M 级
ALUResult_M	O	32	扩展后的 32 位信号流水到 M 级
Pc4_M	O	32	M 级的 pc + 4
Pc8_M	O	32	M 级的 pc + 8

端口 Verilog 声明如下：

```
module REG_M(
    input [31:0] IR_E,
    input [31:0] V2_E,
    input [31:0] ALUResult_E,
    input [31:0] pc4_E,
    input [31:0] pc8_E,
    input clk,
    input reset,
    output reg [31:0] IR_M,
    output reg [31:0] ALUResult_M,
    output reg [31:0] V2_M,
    output reg [31:0] pc4_M,
    output reg [31:0] pc8_M
    );
```

13. M_decoder（M 级）

该模块为 DECODER.v 译码器在 M 级的实例化，专门产生 M 级的控制信号。

M_decoder 模块端口定义如下：

信号名	方向	位宽	描述
IR_M	I	32	M 级 32 位指令序列
MemWrite_M	O	1	DM 写使能信号

端口 Verilog 实例化如下：（DECODER 模块声明见后）

```
DECODER m_decoder (
    .IR(IR_M),
    .MemWrite(MemWrite_M)
);
```

14. DM（M 级）

DM 端口声明如下：

信号名	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
Pc4_M	I	32	M 级的 pc + 4
MemData	I	32	待写入 DM 的 32 位数据信号
MemWrite	I	1	写数据使能信号
MemAddr	I	32	读地址
DM_RD	O	32	DM 中 MemAddr 地址读出的数据

端口 Verilog 声明如下：

```
module DM(
    input clk,
    input reset,
    input [31:0] pc4_M,
    input [31:0] MemAddr, // from ALUResult_E
    input [31:0] MemData, // from V2_M
    input MemWrite_M,
    output [31:0] DM_RD
    );
```

DM 内部 32 bit * 3072 字的存储器的具体实现如下：

```
reg [31:0] dm[0:3071];
```

15. REG_W（W 级）

REG_W 模块端口定义如下：

信号名	方向	位宽	描述
IR_M	I	32	M 级 32 位指令序列
ALUResult_M	I	32	ALU 计算结果流水到 M 级
pc4_M	I	32	M 级的 pc + 4
Pc8_M	I	32	M 级的 pc + 8
Clk	I	1	时钟信号
Reset	I	1	同步复位信号
DM_RD	I	32	DM 读出的 32 位数据
IR_W	O	32	W 级 32 位指令序列
ALUResult_W	O	32	扩展后的 32 位信号流水到 W 级
Pc4_W	O	32	W 级的 pc + 4
Pc8_W	O	32	W 级的 pc + 8
DM_RD_W	O	32	DM 读出的 32 位数据流水到 W 级

端口 Verilog 声明如下：

```
module REG_W(
    input [31:0] IR_M,
    input [31:0] ALUResult_M,
    input clk,
    input reset,
    input [31:0] pc4_M,
    input [31:0] pc8_M,
    input [31:0] DM_RD,
    output reg [31:0] IR_W,
    output reg [31:0] ALUResult_W,
    output reg [31:0] pc4_W,
    output reg [31:0] pc8_W,
    output reg [31:0] DM_RD_W
    );
```

16. W_decoder（W 级）

该模块为 DECODER.v 译码器在 W 级的实例化，专门产生 W 级的控制信号。

W_decoder 模块端口定义如下：

信号名	方向	位宽	描述
IR_W	I	32	W 级 32 位指令序列
RegWrite_W	O	1	GRF 写使能信号
A3_W	O	5	GRF 写入地址
REGorMEM_W	O	1	回写数据信号的选择 0: ALUResult_W; 1: DM_RD_W;
REGop_W	O	2	GRF 写入数据端口选择信号

端口 Verilog 实例化如下：（DECODER 模块声明见后）

```
DECODER m_decoder (
    .IR(IR_M),
    .MemWrite(MemWrite_M)
);
```

17. DECODER

译码器 DECODER 端口声明如下：

端口声明	方向	位宽	描述
IR	I	32	指令 Instr
REGorMEM	O	1	grf 回写控制信号之一： 0：ALUResult 回写； 1：DM 读出的值回写
MemWrite	O	1	DM 写使能信号： 0：不能写入； 1：可以写入
NPCOp	O	3	NPC 计算方式选择信号： 000：PC = PC + 4; 001：PC = PC + 4 + ALUZero && Sign_ext(16imm || 00)

			(beq) 010：PC = PC[31:28] || 26imm || 00 (j、jal) 011：PC = GPR[rs] (jr $ra，注：$ra 不一定得是$31) 100：暂未定义 101：暂未定义 110：暂未定义 111：暂未定义
EXTOp	O	1	EXT 对 16 位立即数的扩展方式： 0：符号扩展 1：零扩展
A1op	O	2	GRF 中 A1 端口选择信号： 00：读取 rs（Instr[25:21]） 01：读取 rt（Instr[20:16]） 10：暂未定义 11：暂未定义
A2op	O	1	GRF 中 A2 端口选择信号： 0：读取 rt（Instr[20:16]） 1：暂未定义
A3op	O	2	GRF 中 A3 端口选择信号： 00：读取 rd（Instr[15:11]） 01：读取 rt（Instr[20:16]） 10：读取常数 31（即$r31 址） 11：暂未定义
REGop	O	2	GRF 写入数据端口选择信号： 00：回写 REGorMEM 选择后的信号 01：回写 16 位立即数加载至高位后的 32 位数据 10：回写当前 PC 加 4 的值 11：暂未定义
RegWrite	O	1	GRF 写使能信号：

			0：不可写 1：可写
ALU_Aop	O	1	ALU SrcA 端口选择信号： 0：读取 RD1 1：暂未定义
ALU_Bop	O	2	ALU SrcB 端口选择信号： 00：读取 RD2 01：读取 16 位立即数（根据 ALUZero 选择）扩展后的 32 位数据 10：读取指令 shamt 片段 0 扩展后的结果 11：暂未定义
ALUOp	O	4	ALU 功能选择信号： 0000：and 0001：or 0010：add（不考虑溢出） 0011：sub（不考虑溢出） 0100：sll 0101：srl 0110：sra 0111：slt（注：有符号!） 1000—1111：暂未定义
A1	O	5	GRF 的 A1 端口输入地址
A2	O	5	GRF 的 A2 端口输入地址
A3	O	5	GRF 的 A3 端口输入地址
shamt	O	5	IR 中 shamt 字段
Tuse_A1_0	O	1	A1 端口数据需要在 D 级使用
Tuse_A1_1	O	1	A1 端口数据需要在 E 级使用
Tuse_A2_0	O	1	A2 端口数据需要在 D 级使用
Tuse_A2_1	O	1	A2 端口数据需要在 E 级使用

Tuse_A2_2	O	1	A2 端口数据需要在 M 级使用
Tnew	O	3	该指令产生结果所需要的时钟周期

端口 verilog 声明如下：

```
module DECODER(
    input [31:0] IR,
    output reg REGorMEM,
    output reg MemWrite,
    output reg [2:0] NPCOp,
    output reg EXTOp,
    output reg [1:0] A1op,
    output reg A2op,
    output reg [1:0] A3op,
    output reg [1:0] REGop,
    output reg RegWrite,
    output reg ALU_Aop,
    output reg [1:0] ALU_Bop,
    output reg [3:0] ALUOp,
    output [4:0] A1,
    output [4:0] A2,
    output [4:0] A3,
    output [4:0] shamt,
//------------------------------
    output reg Tuse_A1_0,
    output reg Tuse_A1_1,
    output reg Tuse_A2_0,
    output reg Tuse_A2_1,
    output reg Tuse_A2_2,
    output reg [2:0] Tnew
    );
```

译码器真值表如下：（本电路实现中 X 全置 0）

指令	opcode	funct	REGorMEM	MemWrite	NPCop	EXTOp	A1op	A2op	A3op	REGop	RegWrite	ALU_Aop	ALU_Bop	ALUOp
addu	000000	100001	0	0	000	0	00	0	00	00	1	0	00	0010
subu	000000	100011	0	0	000	0	00	0	00	00	1	0	00	0011
addi	001000		0	0	000	0	00	0	01	00	1	0	01	0010
ori	001101		0	0	000	1	00	0	01	00	1	0	01	0001
lw	100011		1	0	000	0	00	0	01	00	1	0	01	0010
sw	101011		0	1	000	0	00	0	00	00	0	0	01	0010
beq	000100		0	0	001	0	00	0	00	00	0	0	00	0000
lui	001111		0	0	000	0	00	0	01	00	1	0	01	0000
sll	000000	000000	0	0	000	0	01	0	00	00	1	0	00	0100
srl	000000	000010	0	0	000	0	01	0	00	00	1	0	00	0101
sra	000000	000011	0	0	000	0	01	0	00	00	1	0	00	0110
slt	000000	101010	0	0	000	0	00	0	00	00	1	0	00	0111
j	000010		0	0	010	0	00	0	00	00	0	0	00	0000
jal	000011		0	0	010	0	00	0	10	10	1	0	00	0000
jalr	000000	001001	0	0	011	0	00	0	00	10	1	0	00	0000
jr	000000	001000	0	0	011	0	00	0	00	00	0	0	00	0000

（三）重要机制实现方法

1. 跳转

该 CPU 涉及的跳转指令有：beq、j、jal、jr、jalr。

BEQ：相等时转移

编码	31 26	25 21	20 16	15 0
	beq 000100	rs	rt	offset
	6	5	5	16
格式	beq rs, rt, offset			
描述	if (GPR[rs] == GPR[rt]) then 转移			

操作	if (GPR[rs] == GPR[rt])
	PC ← PC + 4 + sign_extend(offset‖0^2)
	else
	PC ← PC + 4			

示例	beq $s1, $s2, -2			
其他				

对于 beq 指令，控制器解析 beq 指令，输出的 NPCOp = 001，同时 ALU 判断两个操作数 GPR[rs]、GPR[rt]是否相等，若相等将 ALUZero 置一。在 NPC 内，当 ALUZero = 1 并且 NPCOp = 001 时，才执行跳转，即 PC = PC + 4 + ALUZero

&& Sign_ext(16imm || 00)，若不满足该条件，则只执行 PC = PC + 4。

J：跳转

编码	31 26 25 0	
	j 000010	instr_index
	6	26
格式	j target	
描述	j 指令是 PC 相关的转移指令。当把 4GB 划分为 16 个 256MB 区域，j 指令可以在当前 PC 所在的 256MB 区域内任意跳转。	
操作	PC ← PC31..28||instr_index||0^2	
示例	j Loop_End	
其他	如果需要跳转范围超出了当前 PC 所在的 256MB 区域内时，可以使用 JR 指令。	

对于 j 指令，控制器解析 j 指令，将 NPCOp 置为 010，同时 NPC 读入 26 位立即数，PC = {PC[31:28]，26imm，2{0}}（verilog 位拼接运算符），从而实现跳转。

JAL：跳转并链接

编码	31 26 25 0	
	jal 000011	instr_index
	6	26
格式	jal target	
描述	jal 指令是函数指令，PC 转向被调用函数，同时将当前 PC+4 保存在 GPR[31]中。当把 4GB 划分为 16 个 256MB 区域，jal 指令可以在当前 PC 所在的 256MB 区域内任意跳转。	
操作	PC ← PC31..28||instr_index||0^2 GPR[31] ← PC + 4	
示例	jal my_function_name	
其他	jal 与 jr 配套使用。jal 用于调用函数，jr 用于函数返回。当所调用的函数地址超出了当前 PC 所在的 256MB 区域内时，可以使用 jalr 指令。	

对于 jal 指令，其 NPC 相关控制信号与上述 j 指令相同。不同的是，对于 jal 指令，还需要通过 REGop 信号选择 10，将 PC + 4 的值存入$31 寄存器，为此，在 grf 写数据端口 WD3 放置了一个 4 选 1 多路选择器，其第三个选择端口为当前 PC 加 4 的值。

JR：跳转至寄存器

编码	31 26	25 21	20 11	10 6	5 0
	special 000000	rs	0 00 0000 0000	0 00000	jr 001000
	6	5	10	5	6
格式	jr rs				
描述	PC ← GPR[rs]				
操作	PC ← GPR[rs]				
示例	jr $31				
其他	jr 与 jal/jalr 配套使用。jal/jalr 用于调用函数，jr 用于函数返回。				

对于 jr 指令，其将 GPR[rs]寄存器存的 32 位值存入 PC，在数据通路上只需要在 NPC 的输入端口增加一个读入寄存器（32 位数据）的端口，同时设置 NPCOp = 011 时，将 pc 设置为输入的 32 位数据的值。

JALR：跳转并链接

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	0 00000	rd	0 00000	jalr 001001
	6	5	5	5	5	6
格式	jalr rd, rs					
描述	jalr 指令是函数指令，PC 转向被调用函数(函数入口地址保存在 GPR[rs]中)，同时将当前 PC+4 保存在 GPR[rd]中。					
操作	PC ← GPR[rs] GPR[rd] ← PC + 4					
示例	jalr $s1, $31					

对于 jalr 指令，其涉及的 PC 操作与 jr 相同，都是将 GPR[rs]存入 PC。同时，它与 jal 指令相似，都需要将当前 pc 加 4 的值存入寄存器。不同的是，jal 将 pc + 4 存入$31 寄存器，故 A3op 选择 10；而 jalr 将 pc + 4 存入$rd 寄存器，故 A3op 选择 00。

2. 转发

转发 mux：

	D		E			M
Forward MUX	MF_GRFRD1_D	MF_GRFRD2_D	MF_ALUA_E	MF_ALUB_E	MF_MemData_E	MF_DMWD_M
sequence_input	RD1	RD2	V1_E_left	V2_E_left	V2_E_left	V2_M
forward_input	ALUResult_M	ALUResult_M	ALUResult_M	ALUResult_M	ALUResult_M	ALUResult_W
	pc4_M	pc4_M	pc4_M	pc4_M	pc4_M	DM_RD_W
	ALUResult_W	ALUResult_W	ALUResult_W	ALUResult_W	ALUResult_W	pc4_W
	DM_RD_W	DM_RD_W	DM_RD_W	DM_RD_W	DM_RD_W	
	pc4_W	pc4_W	pc4_W	pc4_W	pc4_W	
select	MF_GRFRD1_D_sel	MF_GRFRD2_D_sel	MF_ALUA_E_sel	MF_ALUB_E_sel	MF_MemData_E_sel	MF_DMWD_M_sel
output	V1	V2	V1_E_right	V2_E_right_ALUB	V2_E_right2_RegM	MemData

3. 暂停

Tuse-Tnew 表格：

		Tuse			Tnew		
		A1	A2		功能部件	E	M	W
	addu	1	1		ALU	1	0	0
	subu	1	1		ALU	1	0	0
	ori	1			ALU	1	0	0
	lw	1			DM	2	1	0
	sw	1	2					
	beq	0	0					
	lui				ALU	1	0	0
	sll	1			ALU	1	0	0
	srl	1			ALU	1	0	0
	sra	1			ALU	1	0	0
	slt	1	1		ALU	1	0	0
	j							
	jal				PC	0	0	0
	jalr	0			PC	0	0	0
	jr	0						
	nop							

注：sll、srl、sra的A1为rt字段

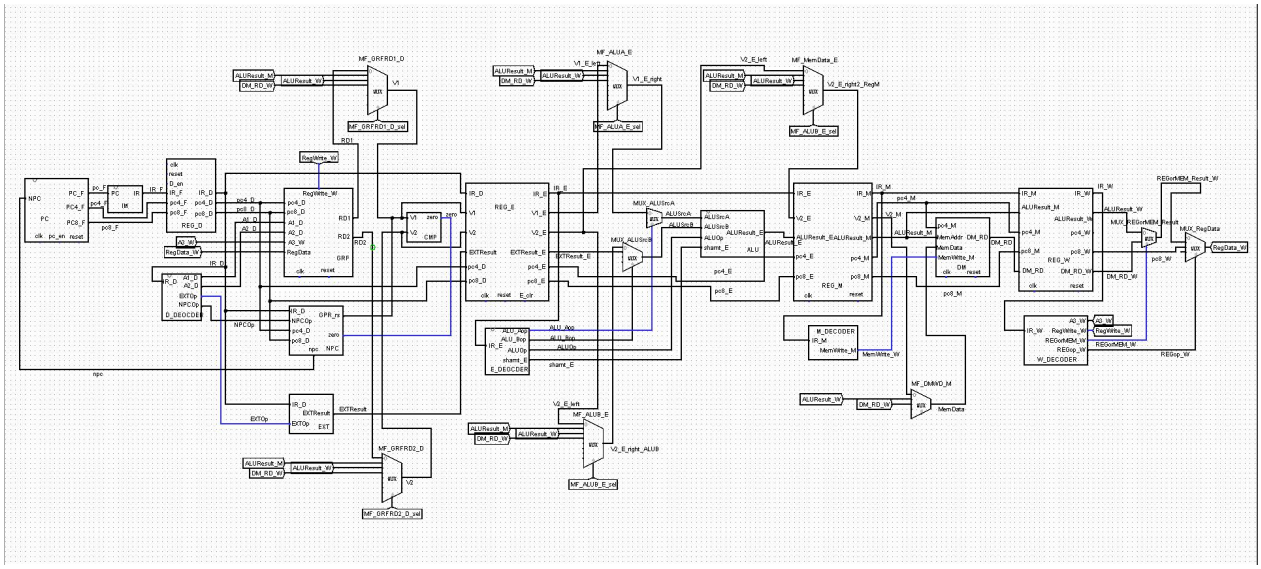
		E			M			W		
	Tnew	ALU	DM	PC	ALU	DM	PC	ALU	DM	PC
A1	Tuse	1	2	0	0	1	0	0	0	0
	0	S	S	F	F	S	F	F	F	F
	1	F	S	F	F	F	F	F	F	F

		E			M			W		
	Tnew	ALU	DM	PC	ALU	DM	PC	ALU	DM	PC
A2	Tuse	1	2	0	0	1	0	0	0	0
	0	S	S	F	F	S	F	F	F	F
	1	F	S	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F

4. 附：无转发数据通路表格

I	J	K	L	M	N	O	P
REG_E	E		REG_M	M		REG_W	
	实例化DECODER	ALU		实例化DECODER	DM		实例化DECODER
IR_D	IR_E	ALUSrcA	IR_E	IR_M	clk	IR_M	IR_W
V1	ALU_Aop	ALUSrcB	V2_E	MemWrite_M	reset	ALUResult_M	A3_W
V2	ALU_Bop	ALUOp	ALUResult_E		MemAddr=ALUResult_E	clk	RegWrite_W
EXTResult	ALUOp	shamt	pc4_E		MemData=V2_M	reset	
pc4_D	shamt_E	ALUResult_E	pc8_E		MemWrite_M	pc4_M	
pc8_D			clk		pc4_M	pc8_M	
clk			reset		DMRD	DMRD	
reset			IR_M			IR_W	
E_clr			ALUResult_M			ALUResult_W	
IR_E			V2_M			pc4_W	
V1_E			pc4_M			pc8_W	
V2_E			pc8_M			DMRD_W	
EXTResult_E							
pc4_E							
pc8_E							
	Tnew_D			Tnew_E			Tnew_M
	Tnew_E			Tnew_M			Tnew_W

5. 附：LOGISIM 数据通路



二、测试方案

（一）典型测试样例

1. 顺序执行指令（无流水线冒险）

```
# no forwarding, no stall, no jump or b_instr
# sequence
lui $0, 0x1234
lui $1, 0xf234
lui $2, 0x2345
lui $3, 0xf456

ori $4, 0x4567
ori $5, 0x5678
ori $6, 0x6789
ori $7, 0x789a

addu $8, $0, $0
addu $9, $1, $1
addu $10, $2, $2
addu $11, $3, $3

subu $12, $2, $1
subu $13, $3, $2
subu $14, $4, $3
subu $15, $5, $4

sll $16, $1, 2
sll $17, $2, 3
sll $18, $3, 4
sll $19, $4, 5

srl $20, $1, 2
srl $21, $2, 3
srl $22, $3, 4
srl $23, $4, 5

sra $24, $1, 2
sra $25, $2, 3
sra $26, $3, 4
sra $27, $4, 5

slt $28, $1, $2
slt $29, $2, $3
```

```
40  slt $30, $3, $4
41  slt $31, $4, $5
42
43  sw $1, 0($0)
44  sw $2, 4($0)
45  sw $3, 8($0)
46  sw $4, 12($0)
47  sw $5, 16($0)
48  sw $6, 20($0)
49
50  lw $11, 0($0)
51  lw $12, 4($0)
52  lw $13, 8($0)
53  lw $14, 12($0)
54  lw $15, 16($0)
55  lw $16, 20($0)
56
57  nop
58  nop
59  nop
60  nop
```

2. 纯 ALU 计算类指令（有流水线冒险）

```
lui $1, 0xffff
ori $2, $1, 0xffff
ori $3, $1, 0x1234
ori $4, $3, 0x5678
ori $5, $4, 0x3333
ori $6, $4, 0x1111
ori $7, $4, 0x2222

lui $8, 0x000f
addu $9, $8, $7
subu $10, $9, $8
addu $11, $10, $9
subu $12, $11, $0
subu $13, $9, $10

sll $14, $13, 2
sll $15, $2, 4
ori $15, $15, 0x1111
sll $16, $15, 2
sll $17, $16, 2
srl $18, $17, 2
srl $19, $18, 1
srl $20, $19, 1
srl $21, $19, 2
sra $22, $21, 2
sra $23, $21, 2
sra $24, $22, 1
sra $25, $22, 2

addu $1, $0, $0
lui $2, 0x000f

slt $26, $24, $25
slt $27, $26, $25
slt $27, $27, $26
slt $28, $25, $27
lui $1, 0x00ff
slt $29, $1, $2
slt $30, $0, $29
```

3. load 指令（有流水线冒险）

```
# initial
lui $1, 0xffff
ori $1, $1, 0xffff
lui $2, 0x1234
ori $2, $2, 0x1234
lui $3, 0x2345
ori $3, $3, 0x2345
lui $4, 0x3456
ori $4, $4, 0x3456
lui $5, 0x4567
ori $5, $5, 0x4567

sw $1, 0($0)
sw $2, 4($0)
sw $3, 8($0)
sw $4, 12($0)
sw $5, 16($0)

# start
lw $6, 0($0)
addu $7, $6, $1
addu $7, $7, $0
addu $7, $7, $0
addu $7, $7, $0
addu $7, $7, $0
subu $8, $6, $2
subu $9, $6, $3

lw $10, 4($0)
ori $11, $10, 0x0001
ori $12, $10, 0x0010
ori $13, $10, 0x0100

lw $14, 8($0)
sll $15, $14, 2
srl $16, $14, 1
sra $17, $14, 1

lw $18, 12($0)
lw $19, 16($0)
slt $20, $18, $19
nop
```

4. store 指令（有流水线冒险，无跳转指令相关数据冒险）

```
lui $1, 0x1234
ori $1, $1, 0x1234
sw $1, 0($0)

lui $2, 0x234
ori $2, $2, 0x234
sw $2, 4($0)

lw $3, 0($0)
sw $3, 8($0)

lw $4, 0($0)
lui $6, 0x1234
subu $5, $4, $6
addu $4, $4, $4
sw $4, 0($5)
#-------test continue-----
#initial
ori $30, $0, 0x0234
ori $29, $0, 0x0324
ori $28, $0, 0x0844
ori $27, $0, 0x0538
ori $26, $0, 0x011c

sw $30, 0($0)
sw $29, 4($0)
sw $28, 8($0)
sw $27, 12($0)
sw $26, 16($0)
#start
lw $10, 0($0)
lw $11, 4($0)
sw $10, 0($11)

lw $11, 8($0)
lw $12, 12($0)
sw $12, 0($11)
lw $12, 16($0)
sw $11, 0($12)
```

5. store 指令（有流水线冒险，有跳转指令相关数据冒险（如：jal））

```
1  ori $1, $0, 0x0020
2  lui $2, 0x3562
3  sll $3, $1, 4
4  sw $2, 4($1)
5
6  ori $3, $1, 0x0028
7  lw $4, 4($1)
8  srl $5, $2, 4
9  sw $4, 8($3)
10 slt $10, $3, $4
11 sw $3, 12($3)
12 lw $3, 8($3)
13 ori $1, $1, 0x0030
14
15 jal here
16 lui $12, 0x4566
17 sw $ra, 16($1)
18 here:
19 jal there
20 addu $11, $ra, $1
21 sw $ra, -400($ra)
22 there:
23 nop
```

6.beq 指令（有流水线冒险）

```
# initial
lui $1, 0xffff
ori $1, $1, 0xffff
lui $2, 0x1234
ori $2, $2, 0x1234
lui $3, 0x2345
ori $3, $3, 0x2345
lui $4, 0x3456
ori $4, $4, 0x3456
lui $5, 0x4567
ori $5, $5, 0x4567

sw $1, 0($0)  # $1 = 0xffff_ffff
sw $2, 4($0)  # $2 = 0x1234_1234
sw $3, 8($0)  # $3 = 0x2345_2345
sw $4, 12($0) # $4 = 0x3456_3456
sw $5, 16($0) # $5 = 0x4567_4567

# start
lw $1, 8($0)  # $1 = $3
lw $2, 8($0)  # $2 = $3

beq $1, $2, yes1
ori $6, $1, 0x1234  # delayed branching
yes2:
ori $7, $1, 0x1234
sll $8, $7, 2
yes1:
beq $7, $8, yes2
sll $7, $7, 2
yes3:
lw $9, 0($0)  # $9 = $1
beq $9, $1, yes3  # delayed branching
sll $1, $1, 2

beq $0, $1, no4
addu $1, $0, $0   # delayed branching

beq $1, $0, yes5
subu $1, $3, $2   # delayed branching
no4:
nop

yes5:
addu $9, $1, $2
addu $10, $3, $4
beq $9, $10, no6
slt $11, $9, $10  # delayed branching

beq $11, $0, no7
srl $12, $11, 31  # delayed branching

beq $12, $0, yes8
srl $13, $12, 31  # delayed branching

yes8:
beq $13, $0, no9
srl $13, $13, 31  # delayed branching

no6:
no7:
nop

no9:
nop
```

7.j 指令（有流水线冒险）

```
ori $1, $0, 1  # $1 = 1
ori $2, $0, 32  # $2 = 32
ori $30, $0, 0x0040  # $30 = 0x0040 (base address)


loop:
    beq $1, $2, end_loop

    sll $3, $1, 4
    addu $4, $3, $30
    addu $20, $1, $1
    sw $20, 4($4)
    sw $1, 0($4)
    j loop
    lw $1, 4($4)  # delayed branch
end_loop:
    ori $6, $1, 0
    j j1
    addu $5, $1, $1

j1:
    beq $5, $6, j1
    addu $6, $6, $6
    j end
    nop
    nop
end:
```

8.jal 指令（有流水线冒险）

```
ori $3, $0, 8
ori $1, $0, 0x0034
ori $2, $0, 0x2345
sw $2, 0($1)
lw $ra, 0($1)
jal label1  # 0x0000_3014
addu $ra, $ra, $3
subu $ra, $ra, $3
label3:
subu $ra, $ra, $3
jal label2
sra $ra, $ra, 4

label1:
    j label3
    sll $ra, $ra, 4
label2:
    nop
    addu $ra, $ra, $ra
```

9.jr 指令（有流水线冒险）

```
1  addu $1, $0, 0x3000  # $1 = base_pc
2  ori $2, $1, 0x0020
3  sw $2, 0($0)
4  lw $1, 0($0)
5  jr $1
6  addu $2, $2, $1
7  ori $3, $1, 0x0020
8  sll $4, $1, 2
9  ori $2, $1, 0x002c  # here
10 addu $2, $2, $1
11 sw $2, 0($0)
12 lw $3, 0($0)
13 beq $2, $3, end
14 nop
15 nop
16 end:
17 nop
18 subu $3, $2, $3
```

三、思考题

（一）题目描述：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。

数据通路方面：由于寄存器值比较提前至位于 D 级，考虑将 NPC 部件同样置于 D 级，这样便于保证比较与跳转操作的接续性。此处注意 beq 判断为 0 时 npc 应输出 pc4_D+4（或 pc4_F），jal 等写入寄存器的指令地址也应该是 pc4_D+4（或 pc4_F）。由于需要暂停控制，还需要从暂停控制器接到 PC 的通路，接入使能信号用于控制暂停。

控制信号方面：将 D 级指令信号译码得到 NPC 操作的控制信号 NPCOp，在 NPC 模块中进行控制。PC 中的使能信号为暂停信号取反，当使能信号为 1 时 PC 正常工作，为 0 时 PC 值冻结。

（二）题目描述：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？

由于使用了延迟槽，CPU 会自动执行跳转指令后一个指令，该指令是在编

译过程中编译器优化加入的，因此返回至跳转位置时应执行的是跳转指令后第二条指令，即 pc（pc_D）+8，所以回写入寄存器的指令应是 pc（pc_D）+8。

（三）题目描述：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？

用例子说明：对于 M – ALU 的转发电路，如果由 ALU 提供供给数据，则会出现 ALU 输出的数据直接连接到 ALU 的输入，引发电路震荡。

（四）题目描述：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。

会产生数据冒险。例子如下：

addu $1, $2, $3

subu $4, $1, $2

mips 指令序列如上所示，第一条 写寄存器$1，第二条指令读$1，若不采用转发电路，则第一条指令计算正确，但第二条指令取出的$1 为 addu 指令执行前的数值，结果写入$4 的数据不是我们想要的。

（五）题目描述：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？

当某一时钟周期读出寄存器编号与写入寄存器编号相同时，需要采用内部转发，将写入数据 RegData 直接读出，才能保证读出的寄存器值正确。

若不采用 GPR 内部转发，可以设置从 W 级流水线寄存器到 GRF 输出数据的转发电路。

（六）题目描述：为什么 0 号寄存器需要特殊处理？

因为 0 号寄存器时钟为 0,。如果前序指令是写一个非 0 数到 0 号寄存器，后序指令读 0 号寄存器，如果不特殊处理，则可能从 0 号寄存器读出非零值。因此需特殊处理使 0 号寄存器始终为 0

（七）题目描述：什么是“最新产生的数据”？

即最新的结果。例如前两个指令都是写$3 寄存器的 ALU 计算类指令。当第

一个指令进入 W 级，第二个指令进入 M 级时，M 级寄存器保存的结果最新。

（八）题目描述：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？

AT 方法中不需要特判 we：当 we = 1 时，转发电路生效，电路行为符合预期。当 we = 0 时，由于 GRF 不写入数据，因此转发与否不影响结果。故不需要特判 we。

翻译时若 we 为 1 则照常翻译，若 we 为 0，则可直接将 A（即写入寄存器地址）设为 0（0 号寄存器写入无效）。

（九）题目描述：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。

我是手动构造测试样例。

首先将指令按照 Tnew 与 Tuse 分类：

		Tuse			Tnew		
		A1	A2	功能部件	E	M	W
addu		1	1	ALU	1	0	0
subu		1	1	ALU	1	0	0
ori		1		ALU	1	0	0
lw		1		DM	2	1	0
sw		1	2				
beq		0	0				
lui				ALU	1	0	0
sll		1		ALU	1	0	0
srl		1		ALU	1	0	0
sra		1		ALU	1	0	0
slt		1	1	ALU	1	0	0
j							
jal				PC	0	0	0
jalr		0		PC	0	0	0
jr		0					
nop							

注：sll、srl、sra的A1为rt字段

A1		E			M			W		
	Tnew	ALU	DM	PC	ALU	DM	PC	ALU	DM	PC
	Tuse	1	2	0	0	1	0	0	0	0
	0	S	S	F	F	S	F	F	F	F
	1	F	S	F	F	F	F	F	F	F

A2		E			M			W		
	Tnew	ALU	DM	PC	ALU	DM	PC	ALU	DM	PC
	Tuse	1	2	0	0	1	0	0	0	0
	0	S	S	F	F	S	F	F	F	F
	1	F	S	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F

再根据每个指令构造需要转发的指令序列。

如构造两个 ALU 类指令的测试样例，需包含以下几种情况：

(1) 两者无数据冒险

(2) 两者有数据冒险，且需要从 M 级转发到 ALU 的 A 端口或 B 端口

(3) 两者有数据冒险，且需要从 W 级转发到 ALU 的 A 端口或 B 端口

若为构造 load 相关指令，则还会涉及暂停等条件。