

计算机组成原理 P5 流水线 CPU 实验报告

20373944 何天然

一、CPU 设计方案综述

（一）总体设计概述

本 CPU 为 Verilog 实现的单周期 MIPS-CPU，支持的指令集包含 { lw、sw、lb、lbu、sb、lh、lhu、sh、add、addu、sub、subu、and、or、slt、nor、xor、sll、srl、sra、sllv、srlv、srav、sltu、beq、bne、bgtz、blez、bltz、bgez、addi、addiu、andi、ori、xori、lui、slti、sltiu、j、jr、jal、jalr } 共 42 条，其中 add 和 sub 不支持溢出中断，其行为与 addu 和 subu 完全一致。为了实现这些功能，CPU 主要包含 Controller、MainController、HazardSolveUnit、Datapath、IF、IF_ID、ID、ID_EX、EX、EX_MEM、MEM、MEM_WB、PC、IM、NPC、EXT、CMP、GRF、ALU、DM，这些模块共可分成三层，顶层为 Controller 和 Datapath；Controller 可分为 MainController 和 HazardSolveUnit；Datapath 可分为 IF、ID、EX、MEM、IF_ID、ID_EX、EX_MEM、MEM_WB，其中 IF 包含 PC、IM，ID 包含 NPC、EXT、CMP、GRF，EX 包含 ALU，MEM 包含 DM。

（二）关键模块定义

1. PC (Program Counter)

模块定义：

```
module PC(  
    input clk,  
    input reset,  
    input WE,  
    input [31:0] nextPC,  
    output reg [31:0] PC  
);
```

表 1 PC 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	WE	I	1	写使能信号
4	nextPC	I	32	下一条指令的地址
5	PC	O	32	当前指令地址

内部逻辑说明：

用一个 32 位寄存器存储当前指令地址，当时钟上升沿到来时，如果同步复位信号有效，则将寄存器复位为起始地址 0x00003000，否则若写使能信号有效，将新的指令地址写入寄存器。

2. NPC（Next PC）

模块定义：

```
module NPC(
    input [31:0] PC_F,
    input [31:0] PC_D,
    input [31:0] offset,
    input [25:0] index,
    input [31:0] register,
    input [2:0] ctrl,
    output [31:0] PCAdd8,
    output [31:0] nextPC
);
```

表 2 NPC 端口定义

序号	信号名	方向	位数	描述
1	PC_F	I	32	F 级的当前指令地址
2	PC_D	I	32	D 级的当前指令地址
3	offset	I	32	branch 跳转的偏移量
4	index	I	26	j 或 jal 跳转地址的 2-27 位
5	register	I	32	jr 或 jalr 指令的跳转地址
6	ctrl	I	3	选择下条指令的地址
7	PCAdd8	O	32	当前指令地址加 8
8	nextPC	O	32	下一条指令地址

内部逻辑说明：

根据 PC_F 或 PC_D 分别计算出 PC_F+4、branch 跳转地址、j 或 jal 跳转地址、jr 或 jalr 跳转地址，然后根据 Ctrl 信号选择要输出的下条指令地址 nextPC。
PCAdd8 用于 jal 和 jalr 指令将 PC_D+8 的值存入相应寄存器。

3. IM (Instruction Memory)

模块定义：

```
module IM(
    input [31:0] A,
    output [31:0] RD
);
```

表 3 IM 端口定义

序号	信号名	方向	位数	描述
1	A	I	32	32 位指令地址
2	RD	O	32	32 位指令

内部逻辑说明：

IM 存储所有指令。根据 32 位地址的[13:2]位取出相应指令。

4. GRF（General Register File）

模块定义：

```

module GRF(
    input clk,
    input reset,
    input WE,
    input [4:0] A1,
    input [4:0] A2,
    input [4:0] A3,
    input [31:0] WD,
    input [31:0] PC,
    output [31:0] RD1,
    output [31:0] RD2
);

```

表 4 GRF 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	WE	I	1	写使能信号
4	A1	I	5	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD1
5	A2	I	5	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD2
6	A3	I	5	5 位地址输入信号，指定 32 个寄存器中的一个作为写入的目标寄存器
7	WD	I	32	32 位数据输入信号
8	PC	I	32	当前指令地址
9	RD1	O	32	输出 A1 指定的寄存器中的 32 位数据
10	RD2	O	32	输出 A2 指定的寄存器中的 32 位数据

内部逻辑说明：

通过多路分解器进行输入数据 WD 和使能信号 WE 的分配，A3 为选择信号；通过多路选择器进行输出数据 RD1 和 RD2 的选择，A1 和 A2 分别为选择信号。当时钟上升沿到来时，如果同步复位信号有效，则将所有寄存器的值清零，否则将 WD 写入 A3 对应的寄存器。PC 信号仅用于格式化输出。

注：使能信号恒为 1，是否要进行写入通过地址是否为 0 来判断。实现了内部转发，当写入地址与读取地址相同时，直接输出 WD。

5. ALU (Arithmetic Logical Unit)

模块定义：

```
module ALU(  
    input [31:0] srcA,  
    input [31:0] srcB,  
    input [3:0] ctrl,  
    output [31:0] result  
);
```

表 5 ALU 端口定义

序号	信号名	方向	位数	描述
1	srcA	I	32	操作数 A
2	srcB	I	32	操作数 B
3	ctrl	I	4	控制信号
4	result	O	32	计算结果

内部逻辑说明：

支持加、减、与、或、异或、移位、比较等运算。分开进行每种计算，最后通过多路选择器和 ctrl 信号来选择输出结果。

6. DM（Data Memory）

模块定义：

```
module DM(  
    input clk,  
    input reset,  
    input WE,  
    input [31:0] A,  
    input [1:0] byteSel,  
    input [2:0] ctrl,
```

```

input [31:0] WD,
input [31:0] PC,
output [31:0] RD
);

```

表 6 DM 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	WE	I	1	写使能信号
4	A	I	32	32 位地址输入信号
5	ctrl	I	3	控制信号
6	WD	I	32	32 位数据输入信号
7	PC	I	32	当前指令地址
8	RD	O	32	32 位数据输出信号

内部逻辑说明：

用一常量 ADDRBITS 记录地址位数，根据地址位数确定内存的大小，并从 32 位地址信号 A 中取出 [ADDRBITS-1:2] 位作为字地址，A 的[1:0]位用于字节选择。当时钟上升沿到来时，如果同步复位信号有效，则清空内存，否则若写使能信号有效，则进行数据写入。根据控制信号可对字、半字、字节进行读写。

7. EXT (Extender)

模块定义：

```

module EXT(
    input [15:0] imm16,

```

```

input [2:0] ctrl,
output [31:0] imm32
);

```

表 7 EXT 端口定义

序号	信号名	方向	位数	描述
1	imm16	I	16	16 位立即数输入信号
2	ctrl	I	3	控制信号
3	imm32	O	32	32 位立即数输出信号

内部逻辑说明：

根据控制信号，可将 16 位立即数零扩展、符号扩展、加载到高位、左移两位并进行符号扩展，最后输出 32 位立即数。

8. CMP（Comparer）

模块定义：

```

module ALU(
    input [31:0] srcA,
    input [31:0] srcB,
    input [3:0] ctrl,
    output result
);

```

表 8 CMP 端口定义

序号	信号名	方向	位数	描述
1	srcA	I	32	操作数 A
2	srcB	I	32	操作数 B
3	ctrl	I	4	控制信号
4	result	O	1	比较结果

内部逻辑说明：

支持各种比较运算，如无符号比较、有符号比较、与零比较等。分开进行每种比较，最后通过多路选择器和 ctrl 信号来选择输出结果。

9. IF (Instruction Fetch)

模块定义：

```

module IF (
    input clk,
    input reset,
    input PCWrite,
    input [31:0] nextPC,
    output [31:0] PC,
    output [31:0] instr
);

```

表 9 IF 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	PCWrite	I	1	PC 的写使能信号
4	nextPC	I	32	下一条指令地址
5	PC	O	32	当前指令地址
6	instr	O	32	当前指令

内部逻辑说明：

为流水线的 F 级。内部连接 PC、IM 等模块，用于取指令。

10. ID (Instruction Decoder)

模块定义：

```

module ID (
    input clk,
    input reset,
    input [31:0] PC_F,
    input [31:0] PC_D,
    input [31:0] PC_W,
    input [31:0] instr,
    input [4:0] regAddr,
    input [31:0] EXBack,
    input [31:0] MEMBack,
    input [31:0] WBBack,
    input regWrite,
    input [1:0] regAddrSel,
    input [3:0] CMPCtrl,

```

```
input [2:0] EXTCtrl,  
input [2:0] NPCCtrl,  
input [1:0] regRD1Forward,  
input [1:0] regRD2Forward,  
output [4:0] shamt,  
output [4:0] regA1,  
output [4:0] regA2,  
output [4:0] regA3,  
output [31:0] regRD1,  
output [31:0] regRD2,  
output [31:0] imm32,  
output CMPResult,  
output [31:0] PCAdd8,  
output [31:0] nextPC  
);
```

表 10 ID 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	PC_F	I	32	F 级的 PC
4	PC_D	I	32	D 级的 PC
5	PC_W	I	32	W 级的 PC
6	instr	I	32	指令
7	regAddr	I	5	GRF 写入地址
8	EXBack	I	32	从 E 级转发的数据
9	MEMBack	I	32	从 M 级转发的数据
10	WBBack	I	32	从 W 级转发的数据
11	regWrite	I	1	GRF 写使能信号
12	regAddrSel	I	2	GRF 写入地址选择
13	CMPCtrl	I	4	CMP 控制信号
14	EXTCtrl	I	3	EXT 控制信号
15	NPCCtrl	I	3	NPC 控制信号
16	regRD1Forward	I	2	GRF[rs]转发数据选择信号
17	regRD2Forward	I	2	GRF[rt]转发数据选择信号
18	shamt	O	5	移位位数
19	regA1	O	5	rs
20	regA2	O	5	rt

21	regA3	O	5	写入地址
22	regRD1	O	32	转发后的 GRF[rs]
23	regRD2	O	32	转发后的 GRF[rt]
24	Imm32	O	32	32 位立即数
25	CMPResult	O	1	CMP 比较结果
26	PCAdd8	O	32	PC+8
27	nextPC	O	32	下一条 PC

内部逻辑说明：

为流水线的 D 级。内部连接 CMP、EXT、GRF、NPC 等模块，进行指令的解码、W 级的写回。

11. EX (Execute)

模块定义：

```

module EX (
    input [4:0] shamt,
    input [31:0] regRD1_orig,
    input [31:0] regRD2_orig,
    input [31:0] imm32,
    input [31:0] MEMBack,
    input [31:0] WBBack,

    input ALUSrcASel,
    input ALUSrcBSel,
    input [3:0] ALUCtrl,
    input [1:0] regRD1Forward,
    input [1:0] regRD2Forward,

```

output [31:0] ALUResult,
output [31:0] regRD2
);

表 11 EX 端口定义

序号	信号名	方向	位数	描述
1	shamt	I	5	移位位数
2	regRD1_orig	I	32	上一级的 GRF[rs]
3	regRD2_orig	I	32	上一级的 GRF[rt]
4	imm32	I	32	32 位立即数
5	MEMBack	I	32	从 M 级转发的数据
6	WBBack	I	32	从 W 级转发的数据
7	ALUSrcASel	I	1	ALU 操作数 A 选择信号
8	ALUSrcBSel	I	1	ALU 操作数 B 选择信号
9	ALUCtrl	I	4	ALU 控制信号
10	regRD1Forward	I	2	GRF[rs]转发数据选择信号
11	regRD2Forward	I	2	GRF[rt]转发数据选择信号
12	ALUResult	O	32	ALU 运算结果
13	regRD2	O	32	转发后的 GRF[rt]

内部逻辑说明：

为流水线的 E 级。内部连接 ALU 和其他模块，用于选择操作数、进行计算并输出结果。

12. MEM (Memory)

模块定义：

```
module MEM (  
    input clk,  
    input reset,  
    input [31:0] ALUResult,  
    input [31:0] regRD2_orig,  
    input [31:0] PC,  
    input [31:0] WBBack,  
    input memWrite,  
    input [2:0] DMCtrl,  
    input regRD2Forward,  
    output [31:0] memRD  
);
```

表 12 MEM 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	ALUResult	I	32	ALU 计算结果
4	regRD2_orig	I	32	上一级的 GRF[rt]
5	PC	I	32	当前指令地址
6	WBBack	I	32	从 W 级转发的数据
7	memWrite	I	1	DM 写使能信号
8	DMCtrl	I	3	DM 控制信号
9	regRD2Forward	I	1	GRF[rt]转发数据选择信号
10	memRD	O	32	内存数据输出信号

内部逻辑说明：

为流水线的 MEM 级。内部连接 DM 和其他模块，用于内存的读写。

13. IF_ID (Pipeline Register IF_ID)

模块定义：

```

module IF_ID(
    input clk,
    input reset,
    input WE,
    input [31:0] instr_I,
    input [31:0] PC_I,
    output reg [31:0] instr_O,
    output reg [31:0] PC_O

```


);

表 13 IF_ID 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	WE	I	1	写使能信号
4	instr_I	I	32	指令
5	PC_I	I	32	指令地址
6	instr_O	O	32	指令
7	PC_O	O	32	指令地址

内部逻辑说明：

为流水寄存器 IF_ID，在 F 级和 D 级间流水数据。

14. ID_EX (Pipeline Register ID_EX)

模块定义：

```
module ID_EX(  
    input clk,  
    input reset,  
    input WE,  
    input [4:0] shamt_I,  
    input [4:0] regA1_I,  
    input [4:0] regA2_I,  
    input [4:0] regA3_I,  
    input [31:0] regRD1_I,  
    input [31:0] regRD2_I,
```

```

input [31:0] imm32_I,
input [31:0] PCAdd8_I,
input [31:0] PC_I,
input memWrite_I,
input [1:0] EXBackSel_I,
input [1:0] MEMBackSel_I,
input [1:0] WBBackSel_I,
input ALUSrcASel_I,
input ALUSrcBSel_I,
input [3:0] ALUCtrl_I,
input [2:0] DMCtrl_I,
input [2:0] Tnew_I,
output reg [4:0] shamt_O,
output reg [4:0] regA1_O,
output reg [4:0] regA2_O,
output reg [4:0] regA3_O,
output reg [31:0] regRD1_O,
output reg [31:0] regRD2_O,
output reg [31:0] imm32_O,
output reg [31:0] PCAdd8_O,
output reg [31:0] PC_O,
output reg memWrite_O,
output reg [1:0] EXBackSel_O,
output reg [1:0] MEMBackSel_O,
output reg [1:0] WBBackSel_O,
output reg ALUSrcASel_O,
output reg ALUSrcBSel_O,
output reg [3:0] ALUCtrl_O,
output reg [2:0] DMCtrl_O,
output reg [2:0] Tnew_O

```

);

表 14 ID_EX 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	WE	I	1	写使能信号
4	shamt_I	I	5	移位位数
5	regA1_I	I	5	rs
6	regA2_I	I	5	rt
7	regA3_I	I	5	GRF 写入地址
8	regRD1_I	I	32	GRF[rs]
9	regRD2_I	I	32	GRF[rt]
10	imm32_I	I	32	32 位立即数
11	PCAdd8_I	I	32	PC+8
12	PC_I	I	32	指令地址
13	memWrite_I	I	1	DM 写使能信号
14	EXBackSel_I	I	2	从 E 级转发的数据选择信号
15	MEMBackSel_I	I	2	从 M 级转发的数据选择信号
16	WBBackSel_I	I	2	从 W 级转发的数据选择信号
17	ALUSrcASel_I	I	1	ALU 操作数 A 选择信号
18	ALUSrcBSel_I	I	1	ALU 操作数 B 选择信号
19	ALUCtrl_I	I	4	ALU 控制信号
20	DMCtrl_I	I	3	DM 控制信号

21	Tnew_I	I	3	当前指令的 Tnew 值
22	shamt_O	O	5	移位位数
23	regA1_O	O	5	rs
24	regA2_O	O	5	rt
25	regA3_O	O	5	GRF 写入地址
26	regRD1_O	O	32	GRF[rs]
27	regRD2_O	O	32	GRF[rt]
28	imm32_O	O	32	32 位立即数
29	PCAdd8_O	O	32	PC+8
30	PC_O	O	32	指令地址
31	memWrite_O	O	1	DM 写使能信号
32	EXBackSel_O	O	2	从 E 级转发的数据选择信号
33	MEMBackSel_O	O	2	从 M 级转发的数据选择信号
34	WBBackSel_O	O	2	从 W 级转发的数据选择信号
35	ALUSrcASel_O	O	1	ALU 操作数 A 选择信号
36	ALUSrcBSel_O	O	1	ALU 操作数 B 选择信号
37	ALUCtrl_O	O	4	ALU 控制信号
38	DMCtrl_O	O	3	DM 控制信号
39	Tnew_O	O	3	当前指令的 Tnew 值

内部逻辑说明：

为流水寄存器 ID_EX，在 D 级和 E 级间流水数据。

15. EX_MEM (Pipeline Register EX_MEM)

模块定义：

```
module EX_MEM(  
    input clk,  
    input reset,  
    input WE;  
    input [4:0] regA2_I,  
    input [4:0] regA3_I,  
    input [31:0] ALUResult_I,  
    input [31:0] regRD2_I,  
    input [31:0] PCAdd8_I,  
    input [31:0] PC_I,  
    input memWrite_I,  
    input [1:0] MEMBackSel_I,  
    input [1:0] WBBackSel_I,  
    input [2:0] DMCtrl_I,  
    input [2:0] Tnew_I,  
    output reg [4:0] regA2_O,  
    output reg [4:0] regA3_O,  
    output reg [31:0] ALUResult_O,  
    output reg [31:0] regRD2_O,  
    output reg [31:0] PCAdd8_O,  
    output reg [31:0] PC_O,  
    output reg memWrite_O,  
    output reg [1:0] MEMBackSel_O,  
    output reg [1:0] WBBackSel_O,  
    output reg [2:0] DMCtrl_O,  
    output reg [2:0] Tnew_O  
);
```

表 15 EX_MEM 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	WE	I	1	写使能信号
4	regA2_I	I	5	rt
5	regA3_I	I	5	GRF 写入地址
6	ALUResult_I	I	32	ALU 运算结果
7	regRD2_I	I	32	GRF[rt]
8	PCAdd8_I	I	32	PC+8
9	PC_I	I	32	指令地址
10	memWrite_I	I	1	DM 写使能信号
11	MEMBackSel_I	I	2	从 M 级转发的数据选择信号
12	WBBackSel_I	I	2	从 W 级转发的数据选择信号
13	DMCtrl_I	I	3	DM 控制信号
14	Tnew_I	I	3	当前指令的 Tnew 值
15	regA2_O	O	5	rt
16	regA3_O	O	5	GRF 写入地址
17	ALUResult_O	O	32	ALU 运算结果
18	regRD2_O	O	32	GRF[rt]
19	PCAdd8_O	O	32	PC+8
20	PC_O	O	32	指令地址

21	memWrite_O	O	1	DM 写使能信号
22	MEMBackSel_O	O	2	从 M 级转发的数据选择信号
23	WBBackSel_O	O	2	从 W 级转发的数据选择信号
24	DMCtrl_O	O	3	DM 控制信号
25	Tnew_O	O	3	当前指令的 Tnew 值

内部逻辑说明：

为流水寄存器 EX_MEM，在 E 级和 M 级间流水数据。

16. MEM_WB (Pipeline Register MEM_WB)

模块定义：

```

module MEM_WB(
    input clk,
    input reset,
    input WE,
    input [4:0] regA3_I,
    input [31:0] ALUResult_I,
    input [31:0] memRD_I,
    input [31:0] PCAdd8_I,
    input [31:0] PC_I,
    input [1:0] WBBackSel_I,
    input [2:0] Tnew_I,
    output reg [4:0] regA3_O,
    output reg [31:0] ALUResult_O,
    output reg [31:0] memRD_O,
    output reg [31:0] PCAdd8_O,
    output reg [31:0] PC_O,
    output reg [1:0] WBBackSel_O,

```

output reg [2:0] Tnew_O

);

表 16 MEM_WB 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	WE	I	1	写使能信号
4	regA3_I	I	5	GRF 写入地址
5	ALUResult_I	I	32	ALU 计算结果
6	memRD_I	I	32	DM 数据输出
7	PCAdd8_I	I	32	PC+8
8	PC_I	I	32	指令地址
9	WBBackSel_I	I	2	从 W 级转发的数据选择信号
10	Tnew_I	I	3	当前指令的 Tnew 值
11	regA3_O	O	5	GRF 写入地址
12	ALUResult_O	O	32	ALU 计算结果
13	memRD_O	O	32	DM 数据输出
14	PCAdd8_O	O	32	PC+8
15	PC_O	O	32	指令地址
16	WBBackSel_O	O	2	从 W 级转发的数据选择信号
17	Tnew_O	O	3	当前指令的 Tnew 值

内部逻辑说明：

为流水寄存器 MEM_WB，在 M 级和 W 级间流水数据。

17. Datapath

模块定义：

```
module Datapath (  
    input clk,  
    input reset,  
    input memWrite_D,  
    input [1:0] regAddrSel_D,  
    input [1:0] EXBackSel_D,  
    input [1:0] MEMBackSel_D,  
    input [1:0] WBBackSel_D,  
    input ALUSrcASel_D,  
    input ALUSrcBSel_D,  
    input [3:0] ALUCtrl_D,  
    input [3:0] CMPCtrl_D,  
    input [2:0] EXTCtrl_D,  
    input [2:0] NPCCtrl_D,  
    input [2:0] DMCtrl_D,  
    input [2:0] Tnew_D,  
    input [1:0] regRD1Forward_D,  
    input [1:0] regRD2Forward_D,  
    input [1:0] regRD1Forward_E,  
    input [1:0] regRD2Forward_E,  
    input regRD2Forward_M,  
    input stall,  
    output [31:0] instr_D,  
    output CMPResult_D,  
    output [4:0] regA1_D,
```

```
output [4:0] regA2_D,  
output [4:0] regA1_E,  
output [4:0] regA2_E,  
output [4:0] regA2_M,  
output [2:0] Tnew_E,  
output [2:0] Tnew_M,  
output [2:0] Tnew_W,  
output [4:0] regA3_E,  
output [4:0] regA3_M,  
output [4:0] regA3_W  
);
```

表 17 Datapath 端口定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位
3	memWrite_D	I	2	DM 写使能信号
4	regAddrSel_D	I	2	GRF 输入地址选择信号
5	EXBackSel_D	I	2	E 级转发数据选择信号
6	MEMBackSel_D	I	2	M 级转发数据选择信号
7	WBBackSel_D	I	1	W 级转发数据选择信号
8	ALUSrcASel_D	I	1	ALU 操作数 A 选择信号
9	ALUSrcBSel_D	I	4	ALU 操作数 B 选择信号
10	ALUCtrl_D	I	4	ALU 控制信号
11	CMPCtrl_D	I	3	CMP 控制信号
12	EXTCtrl_D	I	3	EXT 控制信号
13	NPCCtrl_D	I	3	NPC 控制信号
14	DMCtrl_D	I	3	DM 控制信号
15	Tnew_D	I	3	D 级指令 Tnew 值
16	regRD1Forward_D	I	2	D 级 GRF[rs]转发数据选择信号
17	regRD2Forward_D	I	2	D 级 GRF[rt]转发数据选择信号
18	regRD1Forward_E	I	2	E 级 GRF[rs]转发数据选择信号
19	regRD2Forward_E	I	2	E 级 GRF[rt]转发数据选择信号
20	regRD2Forward_M	I	1	M 级 GRF[rt]转发数据选择信号

21	stall	I	1	阻塞信号
22	instr_D	O	32	当前指令
23	CMPResult_D	O	1	CMP 比较结果
24	regA1_D	O	5	D 级 rs
25	regA2_D	O	5	D 级 rt
26	regA1_E	O	5	E 级 rs
27	regA2_E	O	5	E 级 rt
28	regA2_M	O	5	M 级 rt
29	Tnew_E	O	3	E 级 Tnew
30	Tnew_M	O	3	M 级 Tnew
31	Tnew_W	O	3	W 级 Tnew
32	regA3_E	O	5	E 级 GRF 写入地址
33	regA3_M	O	5	M 级 GRF 写入地址
34	regA3_W	O	5	W 级 GRF 写入地址

内部逻辑说明：

为数据通路，将 IF、ID、EX、MEM 等流水级，IF_ID、ID_EX、EX_MEM、MEM_WB 等流水寄存器，以及数据转发的旁路等连接在一起。

18. MainController

模块定义：

```
module MainController (
    input [31:0] instr,
    input flag,
```

```
output memWrite,  
output [1:0] regAddrSel,  
output [1:0] EXBackSel,  
output [1:0] MEMBackSel,  
output [1:0] WBBackSel,  
output ALUSrcASel,  
output ALUSrcBSel,  
output [3:0] ALUCtrl,  
output [3:0] CMPCtrl,  
output [2:0] EXTCtrl,  
output [2:0] NPCCtrl,  
output [2:0] DMCtrl,  
output [2:0] Tnew,  
output [2:0] Tuse_A1,  
output [2:0] Tuse_A2  
);
```

表 18 MainController 端口定义

序号	信号名	方向	位数	描述
1	instr	I	32	当前指令
2	flag	I	1	跳转条件是否成立
3	memWrite	O	1	DM 写使能信号
4	regAddrSel	O	2	GRF 输入地址选择信号
5	EXBackSel	O	2	E 级转发数据选择信号
6	MEMBackSel	O	2	M 级转发数据选择信号
7	WBBackSel	O	2	W 级转发数据选择信号
8	ALUSrcASel	O	1	ALU 操作数 A 选择信号
9	ALUSrcBSel	O	1	ALU 操作数 B 选择信号
10	ALUCtrl	O	4	ALU 控制信号
11	CMPCtrl	O	4	CMP 控制信号
12	EXTCtrl	O	3	EXT 控制信号
13	NPCCtrl	O	3	NPC 控制信号
14	DMCtrl	O	3	DM 控制信号
15	Tnew	O	3	当前指令 Tnew 值
16	Tuse_A1	O	3	当前指令 rs 的 Tuse 值
17	Tuse_A2	O	3	当前指令 rt 的 Tuse 值

内部逻辑说明：

为主控制器，用于指令的识别和部分控制信号的生成。移码方式选择控制信号驱动型，为了防止代码膨胀，采用聚合连线的方式。

Ins	and	or	nor	xor	addu	add	subu	sub	slt	sltu	sllv	srlv	srav
op	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
funct(rt)	100100	100101	100111	100110	100001	100000	100011	100010	101010	101011	000100	000110	000111
memWrite	0	0	0	0	0	0	0	0	0	0	0	0	0
regAddrSel	rd	rd	rd	rd	rd	rd	rd	rd	rd	rd	rd	rd	rd
EXBackSel	-	-	-	-	-	-	-	-	-	-	-	-	-
MEMBackSel	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult
WBBackSel	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult
ALUSrcASel	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1
ALUSrcBSel	regRD2	regRD2	regRD2	regRD2	regRD2	regRD2	regRD2	regRD2	regRD2	regRD2	regRD2	regRD2	regRD2
ALUCtrl	and	or	nor	xor	add	add	sub	sub	lt	ltu	sll	srl	sra
CMPCtrl	-	-	-	-	-	-	-	-	-	-	-	-	-
EXTCtrl	-	-	-	-	-	-	-	-	-	-	-	-	-
NPCCtrl	add4	add4	add4	add4	add4	add4	add4	add4	add4	add4	add4	add4	add4
DMCtrl	-	-	-	-	-	-	-	-	-	-	-	-	-
Tnew	1	1	1	1	1	1	1	1	1	1	1	1	1
Tuse_A1	1	1	1	1	1	1	1	1	1	1	1	1	1
Tuse_A2	1	1	1	1	1	1	1	1	1	1	1	1	1

Ins	andi	ori	xori	addiu	addi	slti	sltiu	lui		sll	srl	sra
op	001100	001101	100110	001001	001000	001010	001011	001111		000000	000000	000000
funct(rt)	-	-	-	-	-	-	-	-		000000	000010	000011
memWrite	0	0	0	0	0	0	0	0		0	0	0
regAddrSel	rt	rt	rt	rt	rt	rt	rt	rt		rd	rd	rd
EXBackSel	-	-	-	-	-	-	-	imm32		-	-	-
MEMBackSel	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult		ALUResult	ALUResult	ALUResult
WBBackSel	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult	ALUResult		ALUResult	ALUResult	ALUResult
ALUSrcASel	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1	regRD1		shamt	shamt	shamt
ALUSrcBSel	imm32	imm32	imm32	imm32	imm32	imm32	imm32	imm32		regRD2	regRD2	regRD2
ALUCtrl	and	or	xor	add	add	lt	ltu	or		sll	srl	sra
CMPCtrl	-	-	-	-	-	-	-	-		-	-	-
EXTCtrl	zero	zero	zero	sign	sign	sign	sign	loadUpper		-	-	-
NPCCtrl	add4	add4	add4	add4	add4	add4	add4	add4		add4	add4	add4
DMCtrl	-	-	-	-	-	-	-	-		-	-	-
Tnew	1	1	1	1	1	1	1	0		1	1	1
Tuse_A1	1	1	1	1	1	1	1	5		5	5	5
Tuse_A2	5	5	5	5	5	5	5	5		1	1	1

Ins	beq	bgtz	blez	bne	bgez	bltz		j	jal	jr	jalc
op	000100	000111	000110	000101	000001	000001		000010	000011	000000	000000
funct(rt)	-	-	-	-	000001	000000		-	-	001000	001001
memWrite	0	0	0	0	0	0		0	0	0	0
regAddrSel	0	0	0	0	0	0		0	31	0	rd
EXBackSel	-	-	-	-	-	-		-	PCAdd8	-	PCAdd8
MEMBackSel	-	-	-	-	-	-		-	PCAdd8	-	PCAdd8
WBBackSel	-	-	-	-	-	-		-	PCAdd8	-	PCAdd8
ALUSrcASel	-	-	-	-	-	-		-	-	-	-
ALUSrcBSel	-	-	-	-	-	-		-	-	-	-
ALUCtrl	-	-	-	-	-	-		-	-	-	-
CMPCtrl	eq	gt	le	ne	gez	ltz		-	-	-	-
EXTCtrl	brOffset	brOffset	brOffset	brOffset	brOffset	brOffset		-	-	-	-
NPCCtrl	flag?offset: add4							index	index	reg	reg
DMCtrl	-	-	-	-	-	-		-	-	-	-
Tnew	0	0	0	0	0	0		0	0	0	0
Tuse_A1	0	0	0	0	0	0		5	5	0	0
Tuse_A2	0	0	0	0	0	0		5	5	5	5

Ins	lw	lh	lhu	lb	lbu		sw	sh	sb
op	100011	100001	100101	100000	100100		101011	101001	101000
funct(rt)	-	-	-	-	-		-	-	-
memWrite	0	0	0	0	0		1	1	1
regAddrSel	rt	rt	rt	rt	rt		0	0	0
EXBackSel	-	-	-	-	-		-	-	-
MEMBackSel	-	-	-	-	-		-	-	-
WBBackSel	memRD	memRD	memRD	memRD	memRD		-	-	-
ALUSrcASel	regRD1	regRD1	regRD1	regRD1	regRD1		regRD1	regRD1	regRD1
ALUSrcBSel	imm32	imm32	imm32	imm32	imm32		imm32	imm32	imm32
ALUCtrl	add	add	add	add	add		add	add	add
CMPCtrl	-	-	-	-	-		-	-	-
EXTCtrl	sign	sign	sign	sign	sign		sign	sign	sign
NPCCtrl	add4	add4	add4	add4	add4		add4	add4	add4
DMCtrl	word	hfw	ushw	byte	usbt		word	hfw	byte
Tnew	2	2	2	2	2		0	0	0
Tuse_A1	1	1	1	1	1		1	1	1
Tuse_A2	5	5	5	5	5		2	2	2

图 1 指令与控制信号对照表

19. HazardSolveUnit

模块定义：

```
module HazardSolveUnit (  
    input [2:0] Tuse_A1_D,  
    input [2:0] Tuse_A2_D,  
    input [4:0] regA1_D,  
    input [4:0] regA2_D,  
    input [4:0] regA1_E,  
    input [4:0] regA2_E,  
    input [4:0] regA2_M,  
    input [2:0] Tnew_E,  
    input [2:0] Tnew_M,  
    input [2:0] Tnew_W,  
    input [4:0] regA3_E,  
    input [4:0] regA3_M,  
    input [4:0] regA3_W,  
    output [1:0] regRD1Forward_D,  
    output [1:0] regRD2Forward_D,  
    output [1:0] regRD1Forward_E,  
    output [1:0] regRD2Forward_E,  
    output regRD2Forward_M,  
    output stall  
);
```

表 19 HazardSolveUnit 端口定义

序号	信号名	方向	位数	描述
1	Tuse_A1_D	I	3	D 级 rs 的 Tuse 值
2	Tuse_A2_D	I	3	D 级 rt 的 Tuse 值
3	regA1_D	I	5	D 级 rs
4	regA2_D	I	5	D 级 rt
5	regA1_E	I	5	E 级 rs
6	regA2_E	I	5	E 级 rt
7	regA2_M	I	5	M 级 rt
8	Tnew_E	I	3	E 级 Tnew
9	Tnew_M	I	3	M 级 Tnew
10	Tnew_W	I	3	W 级 Tnew
11	regA3_E	I	5	E 级 GRF 写入地址
12	regA3_M	I	5	M 级 GRF 写入地址
13	regA3_W	I	5	W 级 GRF 写入地址
14	regRD1Forward_D	O	2	D 级 GRF[rs]转发数据选择信号
15	regRD2Forward_D	O	2	D 级 GRF[rt]转发数据选择信号
16	regRD1Forward_E	O	2	E 级 GRF[rs]转发数据选择信号
17	regRD2Forward_E	O	2	E 级 GRF[rt]转发数据选择信号
18	regRD2Forward_M	O	1	M 级 GRF[rt]转发数据选择信号
19	stall	O	1	阻塞信号

内部逻辑说明：

为冲突处理单元，用于生成转发或阻塞的控制信号。当先执行指令的目的寄存器不为 0 且与后执行指令的源寄存器相匹配时，进行转发。当先执行指令的 Tnew 值小于后执行指令的 Tuse 值且寄存器相匹配且不为 0 时，进行暴力阻塞。

20. Controller

模块定义：

```
module Controller (  
    input [31:0] instr,  
    input CMPResult,  
    input [4:0] regA1_D,  
    input [4:0] regA2_D,  
    input [4:0] regA1_E,  
    input [4:0] regA2_E,  
    input [4:0] regA2_M,  
    input [2:0] Tnew_E,  
    input [2:0] Tnew_M,  
    input [2:0] Tnew_W,  
    input [4:0] regA3_E,  
    input [4:0] regA3_M,  
    input [4:0] regA3_W,  
    output memWrite,  
    output [1:0] regAddrSel,  
    output [1:0] EXBackSel,  
    output [1:0] MEMBackSel,  
    output [1:0] WBBackSel,  
    output ALUSrcASel,  
    output ALUSrcBSel,  
    output [3:0] ALUCtrl,  
    output [3:0] CMPCtrl,
```

```

output [2:0] EXTCtrl,
output [2:0] NPCCtrl,
output [2:0] DMCtrl,
output [2:0] Tnew,
output [1:0] regRD1Forward_D,
output [1:0] regRD2Forward_D,
output [1:0] regRD1Forward_E,
output [1:0] regRD2Forward_E,
output regRD2Forward_M,
output stall
);

```

表 20 Controller 端口定义

序号	信号名	方向	位数	描述
1	instr	I	32	当前指令
2	CMPResult	I	1	CMP 比较结果
3	regA1_D	I	5	D 级 rs
4	regA2_D	I	5	D 级 rt
5	regA1_E	I	5	E 级 rs
6	regA2_E	I	5	E 级 rt
7	regA2_M	I	5	M 级 rt
8	Tnew_E	I	3	E 级 Tnew
9	Tnew_M	I	3	M 级 Tnew
10	Tnew_W	I	3	W 级 Tnew
11	regA3_E	I	5	E 级 GRF 写入地址
12	regA3_M	I	5	M 级 GRF 写入地址
13	regA3_W	I	5	W 级 GRF 写入地址
14	memWrite	O	2	DM 写使能信号
15	regAddrSel	O	2	GRF 输入地址选择信号
16	EXBackSel	O	2	E 级转发数据选择信号
17	MEMBackSel	O	2	M 级转发数据选择信号
18	WBBackSel	O	1	W 级转发数据选择信号
19	ALUSrcASel	O	1	ALU 操作数 A 选择信号
20	ALUSrcBSel	O	4	ALU 操作数 B 选择信号

21	ALUCtrl	O	4	ALU 控制信号
22	CMPCtrl	O	3	CMP 控制信号
23	EXTCtrl	O	3	EXT 控制信号
24	NPCCtrl	O	3	NPC 控制信号
25	DMCtrl	O	3	DM 控制信号
26	Tnew	O	3	D 级指令 Tnew 值
27	regRD1Forward_D	O	2	D 级 GRF[rs]转发数据选择信号
28	regRD2Forward_D	O	2	D 级 GRF[rt]转发数据选择信号
29	regRD1Forward_E	O	2	E 级 GRF[rs]转发数据选择信号
30	regRD2Forward_E	O	2	E 级 GRF[rt]转发数据选择信号
31	regRD2Forward_M	O	1	M 级 GRF[rt]转发数据选择信号
32	stall	O	1	阻塞信号

内部逻辑说明：

为控制器，连接 MainController 和 HazardSolveUnit。从 Datapath 中获得数据，并向其发送控制信号。

（三）重要机制实现方法

1. 跳转

NPC 模块、EXT 模块、ALU 模块协同工作支持指令 beq 的跳转机制。

NPC 模块内置了判定单元和计算单元来独立支持指令 j、jal、jr 的跳转机制。

2. 半字、字节存取

通过控制信号 DMCtrl 来判断是对字、半字还是字节进行操作。对于写指令，直接根据地址信号，找到相应位置的字、半字或字节进行写入。对于读指令，先

根据地址取出相应位置的字、半字或字节，再根据控制信号进行零扩展或符号扩展后输出。

3. 主控制器

采用集中式译码和指令驱动型译码。为了防止代码膨胀，使用 assign 而不是 always 和阻塞赋值来生成控制信号，并利用宏区分信号类别。

```
//bitwise
assign
(
    (op == `ori) ? { memWrite, regAddrSel, EXBackSel, MEMBackSel, WBBBackSel,
    (op == `lw) ? { 1'b0, regAddr_rt, 2'b0, MEMBack_ALUResult, WBBBack_ALUResult,
    (op == `sw) ? { 1'b0, regAddr_rt, 2'b0, 2'b0, WBBBack_memRD,
    (op == `beq) ? { 1'b1, regAddr_0, 2'b0, 2'b0, 2'b0,
    (op == `lui) ? { 1'b0, regAddr_rt, EXBack_imm32, MEMBack_ALUResult, WBBBack_ALUResult,
    (op == `j) ? { 1'b0, regAddr_0, 2'b0, 2'b0, 2'b0,
    (op == `jal) ? { 1'b0, regAddr_31, EXBack_PCAdd8, MEMBack_PCAdd8, WBBBack_PCAdd8,
    (op == `addi) ? { 1'b0, regAddr_rt, 2'b0, MEMBack_ALUResult, WBBBack_ALUResult,
    (op == `addiu) ? { 1'b0, regAddr_rt, 2'b0, MEMBack_ALUResult, WBBBack_ALUResult,
    (op == `slti) ? { 1'b0, regAddr_rt, 2'b0, MEMBack_ALUResult, WBBBack_ALUResult,
    (op == `bgtz) ? { 1'b0, regAddr_0, 2'b0, 2'b0, 2'b0,
    (op == `blez) ? { 1'b0, regAddr_0, 2'b0, 2'b0, 2'b0,
    (op == `bne) ? { 1'b0, regAddr_0, 2'b0, 2'b0, 2'b0,
    (op == `lh) ? { 1'b0, regAddr_rt, 2'b0, 2'b0, WBBBack_memRD,
    (op == `lhu) ? { 1'b0, regAddr_rt, 2'b0, 2'b0, WBBBack_memRD,
    (op == `lb) ? { 1'b0, regAddr_rt, 2'b0, 2'b0, WBBBack_memRD,
    (op == `lbu) ? { 1'b0, regAddr_rt, 2'b0, 2'b0, WBBBack_memRD,
    (op == `sh) ? { 1'b1, regAddr_0, 2'b0, 2'b0, 2'b0,
    (op == `sb) ? { 1'b1, regAddr_0, 2'b0, 2'b0, 2'b0,
    (op == `andi) ? { 1'b0, regAddr_rt, 2'b0, MEMBack_ALUResult, WBBBack_ALUResult,
    (op == `sltiu) ? { 1'b0, regAddr_rt, 2'b0, MEMBack_ALUResult, WBBBack_ALUResult,
    (op == `xori) ? { 1'b0, regAddr_rt, 2'b0, MEMBack_ALUResult, WBBBack_ALUResult,

```

图 2 控制信号生成的部分代码

4. 转发机制

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，就选择该转发输入来源，在有多转发输入来源都满足条件时，最新产生的数据优先级最高。采取暴力转发的方式，即不需要判断指令间的 Tuse 和 Tnew 的关系，因为当条件不成立时会引发阻塞，而阻塞的优先级更高。

```
assign regRD1Forward_D = (regA3_E && regA3_E == regA1_D) ? `forward_D_EXBack :
    (regA3_M && regA3_M == regA1_D) ? `forward_D_MEMBack :
    `forward_D_orig;
assign regRD2Forward_D = (regA3_E && regA3_E == regA2_D) ? `forward_D_EXBack :
    (regA3_M && regA3_M == regA2_D) ? `forward_D_MEMBack :
    `forward_D_orig;
assign regRD1Forward_E = (regA3_M && regA3_M == regA1_E) ? `forward_E_MEMBack :
    (regA3_W && regA3_W == regA1_E) ? `forward_E_WBBBack :
    `forward_E_orig;
assign regRD2Forward_E = (regA3_M && regA3_M == regA2_E) ? `forward_E_MEMBack :
    (regA3_W && regA3_W == regA2_E) ? `forward_E_WBBBack :
    `forward_E_orig;
assign regRD2Forward_M = (regA3_W && regA3_W == regA2_M) ? `forward_M_WBBBack :
    `forward_M_orig;
```

图 3 转发控制信号的部分代码

5. 阻塞机制

当 D 级指令读取寄存器的地址与 E 级或 M 级的指令写入寄存器的地址相等且不为 0，且 D 级指令的 Tuse 小于对应 E 级或 M 级指令的 Tnew 时，在 D 级暂停指令。阻塞时将 PC 与 IF_ID 寄存器的写使能信号赋为 0，并且刷新 ID_EX 寄存器。

```
wire regA1Stall_E;
wire regA1Stall_M;
wire regA1Stall;
assign regA1Stall_E = (regA3_E && regA3_E == regA1_D) && (Tuse_A1_D < Tnew_E);
assign regA1Stall_M = (regA3_M && regA3_M == regA1_D) && (Tuse_A1_D < Tnew_M);
assign regA1Stall = regA1Stall_E | regA1Stall_M;

wire regA2Stall_E;
wire regA2Stall_M;
wire regA2Stall;
assign regA2Stall_E = (regA3_E && regA3_E == regA2_D) && (Tuse_A2_D < Tnew_E);
assign regA2Stall_M = (regA3_M && regA3_M == regA2_D) && (Tuse_A2_D < Tnew_M);
assign regA2Stall = regA2Stall_E | regA2Stall_M;

assign stall = regA1Stall | regA2Stall;
```

图 4 阻塞控制信号的部分代码

二、测试方案

（一）典型测试样例

对于自动生成的样例见自动测试工具和思考题第六题。

手造的样例主要是用于补充自动生成样例中不足的地方：

```
ori $t0, $0, 123
```

```
addu $a0, $t0, $0
```

```
lw $a0, -123($a0)
```

```
lui $t3, 123
```

```
lui $t3, 0
```

```
lw $t4, 0($t3)
```

```
ori $t0, $0, 0x3028
```

addu \$a0, \$t0, \$0

jr \$a0

ori \$t0, \$0, 0x3038

sw \$t0, 0(\$0)

lw \$a0, 0(\$0)

jr \$a0

nop

jal f

nop

j end

nop

f: jr \$ra

nop

end:

jal tag

nop

tag: lw \$t0, -0x3058(\$ra)

lw \$s0, -0x3038(\$t0)

lw \$s1, -0x3038(\$s0)

lw \$s2, -0x3038(\$s1)

sw \$t0, -0x3038(\$s2)

（二）自动测试工具

1. 测试样例生成器

运行环境：win10 g++ 11.1.0

程序大致流程为：

- ① 利用 `ori` 和 `sw` 初始化寄存器和部分内存。
- ② 将随机四条指令加一个标签组成一个代码块，保证代码块中后面指令的源寄存器是前面指令的目的寄存器，并且所有跳转标签为这个代码块后的标签。按以上规则随机生成若干个代码块。
- ③ 进行延迟槽检查。当延迟槽内出现跳转指令时，在前面插入一条无关指令
- ④ 进行写入地址和跳转地址检查。通过插入指令修改寄存器的值来保证地址正确。

```
void gnrtBlock(int index)
{
    char instr[50];
    int type, a0, a1, a2, isJBTag = 0;
    int dstReg[10], cnt = 0;
    for (int i = 0; i < 4; i++)
    {
        type = rand() % INSTYPENUM;
        switch (type)
        {
            case ori:
                a0 = ranReg;
                a1 = (cnt ? dstReg[rand() % cnt] : ranReg);
                a2 = ranImm(16);
                dstReg[cnt++] = a0;
                break;
            case addu:
                a0 = ranReg;
                a1 = (cnt ? dstReg[rand() % cnt] : ranReg);
                a2 = (cnt ? dstReg[rand() % cnt] : ranReg);
                dstReg[cnt++] = a0;
                break;
            case beq:
                a0 = (cnt ? dstReg[rand() % cnt] : ranReg);
                a1 = (rand() % 2 ? (cnt ? dstReg[rand() % cnt] : ranReg) : a0);
                a2 = index;
                isJBTag = 1;
                break;
            case sw:
                a0 = (cnt ? dstReg[rand() % cnt] : ranReg);
                a1 = 0;
                a2 = (cnt ? dstReg[rand() % cnt] : ranReg);
                if (a2 == 31) a2 == 30;
                break;
        }
    }
}
```

图 5 测试样例生成器部分代码

```
addu $6, $7, $10
ori $6, $0, 18444
jr $6
ori $6, $0, 1
beq $6, $6, TAG78
ori $6, $0, 16402
lw $3, -14250($6)
TAG78:
jal TAG79
ori $29, $31, 30672
lui $20, 20246
ori $20, $0, 24498
lw $16, -23442($20)
TAG79:
```

图 6 生成的部分测试样例

2. 自动执行脚本

注：将 P4 的自动执行脚本做略微改动即可使用。所做的改动有：mars 中代码段上限改为 0x00006ffc、命令行运行 mars 时加入延迟槽和执行指令数上限、修改 verilog 的工程路径。

运行环境：win10 64 位 python 3.9.6

步骤 1：爆改 mars，加入格式化输出，并把 \$gp 和 \$sp 的初始值改为 0，把代码段的上限改为 0x00006ffc。

```

public int setWord(int address, int value) throws AddressErrorException {
    if (address % WORD_LENGTH_BYTES != 0) {
        throw new AddressErrorException(
            "store address not aligned on word boundary ",
            Exceptions.ADDRESS_EXCEPTION_STORE,address);
    }
    //output upated memorydata////////////////////////////////////
    SystemIO.println(String.format("@%08x: *%08x <= %08x\n", RegisterFile.getProgramCounter() - 4, address, value));
    //////////////////////////////////////////////////
    return (Globals.getSettings().getBackSteppingEnabled())
        ? Globals.program.getBackStepper().addMemoryRestoreWord(address,set(address, value, WORD_LENGTH_BYTES))
        : set(address, value, WORD_LENGTH_BYTES);
}

```

```

public static int updateRegister(int num, int val){
    int old = 0;
    if(num == 0){
        //System.out.println("You can not change the value of the zero register.");
    }
    else {
        for (int i=0; i< regFile.length; i++){
            if(regFile[i].getNumber()== num) {
                //output the value of upated register////////////////////////////////////
                SystemIO.println(String.format("@%08x: $%2d <= %08x\n",programCounter.getValue() - 4, i, val));
                //////////////////////////////////////////////////
                old = (Globals.getSettings().getBackSteppingEnabled())
                    ? Globals.program.getBackStepper().addRegisterFileRestore(nun,regFile[i].setValue(val))
                    : regFile[i].setValue(val);
                break;
            }
        }
    }
}

```

```

private static int[] dataBasedCompactConfigurationItemValues = {
    0x00003000, // .text Base Address
    0x00000000, // Data Segment base address
    0x00001000, // .extern Base Address
    // $gp init 0 //////////////////////////////////////
    //0x00001800, // Global Pointer $gp
    0x00000000,
    //////////////////////////////////////
    0x00000000, // .data base Address
    0x00002000, // heap base address
    // $sp init 0 //////////////////////////////////////
    //0x00002ffc, // stack pointer $sp
    0x00000000,
    //////////////////////////////////////
    0x00002ffc, // stack base address
    0x00003fff, // highest address in user space
    0x00004000, // lowest address in kernel space
    0x00004000, // .ktext base address
}

```

图 7 修改的 mars 代码

步骤 2: 运行 mars, 将结果输出到文件, 并导出指令

```

def runMars(asm, code, out):
    os.system("java -jar " + marsPath + " db nc mc CompactDataAtZero a dump .text HexText " + code + " " + asm)
    os.system("java -jar " + marsPath + " " + asm + " 4096 db nc mc CompactDataAtZero > " + out)

```

图 8 运行 mars 和导出指令的脚本

```

1  @00003000: $ 1 <= 00000d16
2  @00003004: $ 2 <= 00003368
3  @00003008: $ 3 <= 00006ade
4  @0000300c: $ 4 <= 00001728
5  @00003010: $ 5 <= 00006474
6  @00003014: $ 6 <= 00006784
7  @00003018: $ 7 <= 0000392c
8  @0000301c: $ 8 <= 000020a0
9  @00003020: $ 9 <= 00004e2e
10 @00003024: $10 <= 00006e33
11 @00003028: $11 <= 000056c2
12 @0000302c: $12 <= 00003b2d
13 @00003030: $13 <= 00000bd4
14 @00003034: $14 <= 0000579a
15 @00003038: $15 <= 000054c8
16 @0000303c: $16 <= 00002ead

```

图 9 mars 的运行结果

步骤 3: 生成 prj 和 tcl 文件, 编译 verilog 文件, 进行仿真并将结果输出到文件

```

25 # generate prj and tcl file
26 def initISE(prj):
27     verilogPath = prj + "my_files\\cpu\\"
28     prjFilePath = prj + "mips.prj"
29     tclFilePath = prj + "mips.tcl"
30
31     with open(prjFilePath, "w") as prjFile, open(tclFilePath, "w") as tclFile:
32         for root, dirs, files in os.walk(verilogPath):
33             for fileName in files:
34                 if re.match(r"[w]*\v", fileName):
35                     prjFile.write("Verilog work " + root + "\\ " + fileName + "\n")
36                     tclFile.write("run 20000us" + "\n" + "exit")
37
38 # compile and run verilog
39 def runISE(prj, code, out):
40     prjFilePath = prj + "mips.prj"
41     tclFilePath = prj + "mips.tcl"
42     exeFilePath = prj + "mips.exe"
43     logFilePath = prj + "log.txt"
44     codeFilePath = prj + "code.txt"
45
46     with open(code, "r") as codeSrc, open(codeFilePath, "w") as codeDst:
47         codeDst.write(codeSrc.read())
48
49     os.environ['XILINX'] = xilinxPath
50     os.system(xilinxPath + "bin\\nt64\\fuse -nodebug -prj " + prjFilePath + " -o " + exeFilePath + " mips_tb > " + logFilePath)
51     os.system(exeFilePath + " -nolog -tclbatch " + tclFilePath + " > " + out)

```

图 10 ISE 运行脚本

```

1  ISim P.20131013 (signature 0x7708f090)
2  WARNING: A WEBPACK license was found.
3  WARNING: Please use Xilinx License Configuration Manager to check out a full ISim license.
4  WARNING: ISim will run in Lite mode. Please refer to the ISim documentation for more information on the differences between the Lite and the Full v
5  This is a lite version of ISim.
6  Time resolution is 1 ps
7  Simulator is doing circuit initialization process.
8  Finished circuit initialization process.
9
10 145@00003000: $ 1 <= 00000d16
11 155@00003004: $ 2 <= 00003368
12 165@00003008: $ 3 <= 00006ade
13 175@0000300c: $ 4 <= 00001728
14 185@00003010: $ 5 <= 00006474
15 195@00003014: $ 6 <= 00006784
16 205@00003018: $ 7 <= 0000392c
17 215@0000301c: $ 8 <= 000020a0
18 225@00003020: $ 9 <= 00004e2e
19 235@00003024: $10 <= 00006e33
20 245@00003028: $11 <= 000056c2
21 255@0000302c: $12 <= 00003b2d
22 265@00003030: $13 <= 00000bd4

```

图 11 ISE 的运行结果

步骤 4: 将 mars 和 ISE 的运行结果进行文本比对, 如果出错则给出错误信息

```
53 # compare my and std
54 def cmp(my, std, res):
55     with open(my, "r") as myFile, open(std, "r") as stdFile, open(res, "w") as out:
56         myFileText = myFile.read()
57         myLogs = re.findall("@[\n]*", myFileText)
58         stdLogs = re.findall("@[\n]*", stdFile.read())
59         asmLogs = re.findall("asm: [\n]*", myFileText)
60
61         isAC = True
62
63         for i in range(len(stdLogs)):
64             if i < len(myLogs) and myLogs[i] != stdLogs[i]:
65                 out.write("On Line " + str(i+1) + "\n")
66                 out.write("\tGet\t\t: " + myLogs[i] + "\n")
67                 out.write("\tExpect\t: " + stdLogs[i] + "\n")
68                 print("On Line " + str(i+1))
69                 print("\tGet\t\t: " + myLogs[i])
70                 print("\tExpect\t: " + stdLogs[i])
71                 if (i < len(asmLogs)):
72                     out.write("\tAsm\t\t: " + asmLogs[i] + "\n")
73                     print("\tAsm\t\t: " + asmLogs[i])
74                     isAC = False
75                     break
76                 elif i >= len(myLogs):
77                     out.write("myLogs is too short \n")
78                     print("myLogs is too short")
79                     isAC = False
80                     break
81             if isAC :
82                 out.write("All Accepted")
83                 print("All Accepted")
84         return isAC
```

图 12 文本比对程序

(三) 其他自动化工具

1. 自动生成控制信号:

将指令与控制信号对应表中的某一列复制到 ctrltable.txt 文件中, 再运行这个程序就可以得到一行格式化的控制信号, 直接复制到 MainController 即可完成新增指令的控制信号生成。

```

1 bitwise = [1, 2, 2, 2, 2, 1, 1, 4, 4, 3, 3, 3, 3, 3, 3]
2 pre = [" 1'b", "`regAddr_", "`EXBack_", "`MEMBack_", "`WBBack_", "`ALUSrcA_", "`ALUSrcB_", "`ALU_",
3        "`CMP_", "`EXT_", "`NPC_", "`DM_", " 3'd", " 3'd", " 3'd"]
4 lens = [9, 11, 14, 18, 17, 15, 15, 8, 8, 14, 28, 8, 5, 8, 9]
5
6 with open("D:\\study\\CO\\p5\\PipelineCPU\\my_files\\tools\\gnrt_ctrlsignal\\ctrltable.txt", "r") as
7     ctrlsignals = ctrltable.read().splitlines()
8     for i in range(len(ctrlsignals)):
9         cnt = 0
10        if i != 0:
11            out.write(",")
12        if ctrlsignals[i] == "-":
13            out.write(" " + str(bitwise[i]) + "'b0")
14            cnt = 4 + len(str(bitwise[i]))
15        else:
16            out.write(pre[i] + ctrlsignals[i])
17            cnt = len(pre[i]) + len(ctrlsignals[i])
18        while cnt < lens[i]:
19            out.write(" ")
20            cnt += 1

```

图 13 生成控制信号程序

2. 自动生成模块端口：

像 Controller、Datapath、流水寄存器的模块的端口高达三四十个。将模块定义时的端口声明复制到 iotable.txt 中，运行该程序，即可得到一系列实例化时用到的 .pinName(pinName) 格式的的代码，复制到要实例化模块的地方即可。

```

1 from os import truncate
2 import re
3
4 with open("D:\\study\\CO\\p5\\tools\\gnrt_io\\iotable.txt", "r") as iotable, open("D:\\st
5     io = iotable.read().splitlines()
6     for i in range(len(io)):
7         s = re.search(r"(?:input|output)(?: reg)?(?: \[[0-9]+\:0\])? ([\w]+)", io[i])
8         if s:
9             out.write("      ." + s.group(1) + "(")
10            if (False):
11                out.write(s.group(1))
12            out.write("),\n")
13        else:
14            out.write("\n")

```

图 14 生成模块端口程序

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。

对于数据通路，由于 IF_ID 流水寄存器和延迟槽的存在，应分别将 F 级的 PC

和 D 级的 PC 传入 NPC，F 级的 PC 用于计算正常的 PC+4，而 D 级的 PC 用于计算跳转地址和输出 PC+8。正常情况下，PC 每个周期加 4，但当跳转条件满足时（包括无条件跳转），NPC 会计算出相应的地址，在下个周期跳转到这个地址。

对于控制信号，在 D 级指令译码阶段，控制器根据指令会生成 NPC 控制信号，从而选择下一个 PC 值。当 beq 或 jr 发生数据冲突时，控制器还会生成转发或阻塞的控制信号。

（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 ？

因为有延迟槽的存在，所以需要链接延迟槽后一条指令，否则跳转回来时又会执行一遍延迟槽。

（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？

ALU 或 DM 等部件延时较长，如果由 ALU 或 DM 等部件提供数据，将大大增加关键路径所需时间，使流水线各阶段延迟不均衡，时钟周期增加，性能严重下降。比方说 ALU 需要来自 DM 产生的操作数，如果从 MEM_WB 流水寄存器转发回来，那么关键路径只有 ALU；但如果直接从 DM 转发回来，那么关键路径至少包含 DM+ALU，而且还可能有 DM 前的旁路，这样时钟周期可能会是原来的两倍甚至更多。

（四）“转发（旁路）机制的构造”中的 Thinking 1-4;

Thinking1：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。

如果原始数据本来是错误的，并且等到使用它时转发来源已经离开流水线，那么就会导致数据错误。

比如指令序列：

```
add $t0, $t1, $t2
```

```
nop
```

```
sw $t0, 0($s0)
```

当 sw 在 D 级、add 在 M 级时，ALU 的数据已经准备好可以转发给 D 级的 \$t0，但是 sw 在 D 级时还不需要用 \$t0，如果 sw 还继续采用原来的 \$t0，那么等到 M 级需要用时，W 级此时是 nop，转发来源已经消失，sw 存储的数据就是错误的。

Thinking2: 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？

若不采用 GRF 内部转发机制，当 A1（或 A2）与 A3 相同时，GRF 可能输出的值就有两种可能，一种是输出寄存器原本的值，另一种是输出准备写入的数据。所以采用内部转发机制可以防止输出结果发生变化或者不确定。而且不用再考虑外部转发，节省了部分资源，更加易于设计和实现。

如果不采用内部转发机制，可以通过像 E->D，M->D 那样构造旁路并增加控制信号来实现外部转发。

Thinking3: 为什么 0 号寄存器需要特殊处理？

写入 0 号寄存器相当于不写入，也就是并不会改变 0 号寄存器原有的值，如果前面的指令向 0 寄存器写入非零值，就会转发回来错误的数据。

Thinking4: 什么是“最新产生的数据”？

“最新产生的数据”就是最后一条指令写入的数据，按照“新鲜”程度排序的话，E 级>M 级>W 级，所以能从 E 级转发的就不从 M 和 W 级转发，能从 M 级转发就不从 W 级转发。

（五）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 法的核心思想是尽可能只通过 A 和 T 来生成转发和阻塞的控制信号。因为写入 0 寄存器等于不进行写入，所以可以把 GRF 的写使能信号包含在 A 中，当不写入寄存器或是写入 0 号寄存器时，A 可以都设为 0，就不需要额外传入 WE 了。

在翻译出 A 时可以进行特殊处理，如果使能信号 WE 为 0 就将 A 设成 0。但笔者并没有采用这种做法，而是“忽略”使能信号，将其始终设为 1，只通过写入地址是否为 0 来判断是否要进行写入。可以在写入地址的多路选择器上增加一个输入 0，当指令不写入寄存器时，控制器产生的控制信号可以让该多路选择器选择 0 地址，这样在 GRF 和冲突处理单元中只需要判断写入地址，在 GRF 中也不需要再在翻译 A 时做特殊处理了。

（六）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？

考虑 P5 的 MIPS-lite2 指令集，可以划分为一下几个类：寄存器立即数计算 CalRI（以 ori 为代表，后面同理），双寄存器计算 CalRR（addu），双寄存器分支 BrRR（beq），写内存 Store（sw），读内存 Load（lw），跳转并链接 jal，跳转寄存器 jr，加载高位 lui。（不考虑测试 nop、j，因为并不会引起冲突）

按转发的方向或阻塞的原因分类，可能产生的冲突类型有（同时给出样例）：
转发：

1. E->D:

```
lui $t0, 123
addu $t1, $t1, $t0
```

2. M->D:

```
addu $t1, $t0, $t0
nop
addu $t2, $t1, $t1
```

3. W->D:

```
lw $t1, 0($0)
nop
nop
```

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

4. M->E:

```
addu $t1, $t0, $t0
```

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

5. W->E:

```
addu $t1, $t0, $t0
```

```
nop
```

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

6. W->M

```
addu $t1, $t0, $t0
```

```
sw $t1, 0($0)
```

阻塞:

1. E 级导致的阻塞:

```
addu $t0, $t1, $t1
```

```
beq $t0, $t1, TAG
```

2. M 级导致的阻塞:

```
lw $t0, 0($t1)
```

```
nop
```

```
beq $t0, $t1, TAG
```

对于 W->D 的转发，可通过 GRF 的内部转发解决，其他的转发需要另开旁路，并通过多路选择器在转发数据来源端和使用端分别进行数据选择。而对于不可转发的冲突，使用 AT 法暴力阻塞即可。

笔者的样例生成程序主要方法是：先随机生成四条指令一组的指令块，在生成的时候，保证后面指令的源寄存器一定是前面指令的目的寄存器，从而构成连续数据冒险序列。并且之后会进行指令合法性检查，比如在连续的跳转指令间插入无关指令、修改访存的地址、修改跳转寄存器的值等。

程序的不足主要有两方面：一是在进行合法性检查时，由于没能想到特别好的方法，所以主要是通过插入指令的方式修改相关的值，因此会破坏本来的连续冒险序列。二是目前只从每个指令类中取一个代表性指令进行测试，并没有实现

所有相同类型的其他指令的测试。

考虑到程序有这些不足的地方，又根据样例分析程序给出的结果手动造了一组数据，用于弥补自动生成程序遗漏之处。最后经分析 `forward` 和 `stall` 的均分分别能达到 85 和 95，并且没有 `warning`。