计算机组成原理 P6 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 实现的流水线 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、mult、multu、div、divu、mfhi、mflo、mthi、mtlo }共 50 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、NPC、EXT、CMP、GRF、ALU、MDU、BE、DE，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，ID 包含 NPC、EXT、CMP、GRF，EX 包含 ALU、MDU，MEM 包含 BE、DE。

（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. MDU（MultiplyDivideUnit）

模块定义：

```
module MDU (
    input clk,
    input reset,
    input [31:0] srcA,
    input [31:0] srcB,
    input start,
    input [3:0] ctrl,
    output busy,
```

```
    output reg [31:0] HI,
    output reg [31:0] LO
);
```

表 3 MDU 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	同步复位信号
3	srcA	I	32	操作数 A
4	srcB	I	32	操作数 B
5	start	I	1	开始信号
6	ctrl	I	4	控制信号
7	busy	O	1	忙碌信号
8	HI	O	32	HI 寄存器的值
9	LO	O	32	LO 寄存器的值

内部逻辑说明：

可进行乘除法运算和 mf、mt 指令，内部采用状态机来实现乘除槽。

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. BE（ByteEnable）

模块定义：

```
module BE(
    input [31:0] WD_orig,
    input [1:0] byteSel,
    input [2:0] ctrl,
    output [3:0] byteEn,
    output [31:0] WD
);
```

表 6 BE 端口定义

序号	信号名	方向	位数	描述
1	WD_orig	I	32	原始写入数据
2	byteSel	I	4	字节选择信号
3	ctrl	I	3	控制信号
4	byteEn	I	4	字节使能信号
5	WD	I	32	写入数据

内部逻辑说明：

根据控制信号和字节选择信号，生成字节使能信号，并对写入数据进行处理。

7. DE（DataExtender）

模块定义：

```
module DE(
    input [31:0] RD_orig,
    input [1:0] byteSel,
    input [2:0] ctrl,
    output [31:0] RD
);
```

表 7 DE 端口定义

序号	信号名	方向	位数	描述
1	RD_orig	I	32	原始读取数据
2	byteSel	I	2	字节选择信号
3	ctrl	I	3	控制信号
4	RD	O	32	读取数据

内部逻辑说明：

根据字节选择信号和控制信号，对原始读取数据进行位扩展等处理，然后输出正确的读取数据。

8. EXT（Extender）

模块定义：

```
module EXT(
    input [15:0] imm16,
    input [2:0] ctrl,
    output [31:0] imm32
    );
```

表 8 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。

9. CMP（Comparer）

模块定义：

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

表 9 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。

10. 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 10 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。

11. 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 11 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。

12. 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 12 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。

13. 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 13 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。

14. 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 14 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。

15. 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 15 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。

16. 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 16 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。

17. 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 17 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。

18. 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 18 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。

19. 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 19 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	jalr
op	000100	000111	000110	000101	000001	000001		000010	000011	000000	000000
funct(rt)	–	–	–	–	00001	00000		–	–	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	hfwd	ushw	byte	usbt		word	hfwd	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

20. 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 20 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。

21. 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 21 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、CMP 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 BECtrl 和 DECtrl 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   1          2          2          2          2
assign                          { memWrite, regAddrSel, EXBackSel    , MEMBackSel       , WBBackSel       ,
       (op == `ori  ) ?         { 1'b0    ,`regAddr_rt, 2'b0         ,`MEMBack_ALUResult,`WBBack_ALUResult,
       (op == `lw   ) ?         { 1'b0    ,`regAddr_rt, 2'b0         , 2'b0             ,`WBBack_memRD    ,
       (op == `sw   ) ?         { 1'b1    ,`regAddr_0 , 2'b0         , 2'b0             , 2'b0            ,
       (op == `beq  ) ?         { 1'b0    ,`regAddr_0 , 2'b0         , 2'b0             , 2'b0            ,
       (op == `lui  ) ?         { 1'b0    ,`regAddr_rt,`EXBack_imm32 ,`MEMBack_ALUResult,`WBBack_ALUResult,
       (op == `j    ) ?         { 1'b0    ,`regAddr_0 , 2'b0         , 2'b0             , 2'b0            ,
       (op == `jal  ) ?         { 1'b0    ,`regAddr_31,`EXBack_PCAdd8,`MEMBack_PCAdd8   ,`WBBack_PCAdd8   ,
       (op == `addi ) ?         { 1'b0    ,`regAddr_rt, 2'b0         ,`MEMBack_ALUResult,`WBBack_ALUResult,
       (op == `addiu) ?         { 1'b0    ,`regAddr_rt, 2'b0         ,`MEMBack_ALUResult,`WBBack_ALUResult,
       (op == `slti ) ?         { 1'b0    ,`regAddr_rt, 2'b0         ,`MEMBack_ALUResult,`WBBack_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             ,`WBBack_memRD    ,
       (op == `lhu  ) ?         { 1'b0    ,`regAddr_rt, 2'b0         , 2'b0             ,`WBBack_memRD    ,
       (op == `lb   ) ?         { 1'b0    ,`regAddr_rt, 2'b0         , 2'b0             ,`WBBack_memRD    ,
       (op == `lbu  ) ?         { 1'b0    ,`regAddr_rt, 2'b0         , 2'b0             ,`WBBack_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,`WBBack_ALUResult,
       (op == `sltiu) ?         { 1'b0    ,`regAddr_rt, 2'b0         ,`MEMBack_ALUResult,`WBBack_ALUResult,
       (op == `xori ) ?         { 1'b0    ,`regAddr_rt, 2'b0         ,`MEMBack_ALUResult,`WBBack_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，在有多个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_WBBack  :
                         `forward_E_orig;
assign regRD2Forward_E = (regA3_M && regA3_M == regA2_E) ? `forward_E_MEMBack :
                         (regA3_W && regA3_W == regA2_E) ? `forward_E_WBBack  :
                         `forward_E_orig;
assign regRD2Forward_M = (regA3_W && regA3_W == regA2_M) ? `forward_M_WBBack  :
                         `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

6. 乘除槽

在 MDU 内部构建有限状态机，当 start 信号有效时，从初始状态变为等待状态，并保存操作数和控制信号。过 5 或 10 个周期后，写入 HI 和 LO 寄存器并返回初始状态。

二、测试方案

（一）典型测试样例

见自动测试工具和思考题第四题。

（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 进行序列合法性检查。对于 DoubleDelay 和 JalSame 这样的问题，在生成时就可避免。对于 DivZero、Ov、AdEl 和 AdEs 的问题，通过模拟 mips 程序运行，更换指令或操作数来解决。

④ 对于 jal、jr、jalr 相关的冲突，单独生成一组数据，通过特殊指令序列生成样例。

```
526  void gnrtBlock(int index, int& lastDst, int& JBFlag)
527  {
528      int type, ins, a0, a1, a2;
529      int dstReg[10] = {lastDst}, cnt = 1;
530      for (int i = 0; i < 4; i ++)
531      {
532          type = rand() % TYPENUM;
533          if (type == br_r2 || type == br_r1) // DoubleDelay
534          {
535              if (JBFlag)
536                  while (type == br_r2 || type == br_r1)
537                      type = rand() % TYPENUM, JBFlag = 0;
538              else JBFlag = 1;
539          }
540          else JBFlag = 0;
541          ins = rand() % insNum[type];
542          if (type == cal_ri) a0 = ranReg, a1 = dstReg[rand()%cnt], a2 = ranImm(5, 0), dstReg[cnt ++] = a0;
543          else if (type == cal_rr) a0 = ranReg, a1 = dstReg[rand()%cnt], a2 = dstReg[rand()%cnt], dstReg[cnt ++] = a0;
544          else if (type == br_r2) a0 = dstReg[rand()%cnt], a1 = dstReg[rand()%cnt], a2 = index;
545          else if (type == br_r1) a0 = dstReg[rand()%cnt], a1 = index;
546          else if (type == store) a0 = dstReg[rand()%cnt], a1 = 0, a2 = dstReg[rand()%cnt];
547          else if (type == load) a0 = ranReg, a1 = 0, a2 = dstReg[rand()%cnt], dstReg[cnt ++] = a0;
548          else if (type == mf) a0 = ranReg, dstReg[cnt ++] = a0;
549          else if (type == mt) a0 = dstReg[rand()%cnt];
550          else if (type == md) a0 = dstReg[rand()%cnt], a1 = dstReg[rand()%cnt];
551          else if (type == _lui) a0 = ranReg, a1 = ranImm(5, 0), dstReg[cnt ++] = a0;
552          instrs.push_back(new Instr(type, ins, a0, a1, a2));
553      }
554      instrs.push_back(new Instr(index));
555      lastDst = dstReg[cnt-1];
```

图 5 测试样例生成器部分代码

```
2563    xori $2, $2, 1
2564    bltz $2, TAG506
2565    ori $3, $2, 0
2566    TAG506:
2567    lbu $1, 0($3)
2568    mfhi $4
2569    beq $3, $1, TAG507
2570    lw $1, 0($1)
2571    TAG507:
2572    lh $3, 0($1)
2573    xor $1, $3, $3
2574    and $4, $3, $1
```

图 6 生成的部分测试样例

2. 自动执行脚本

在 P5 基础上修改了测试样例的目录结构，能进行大量样例（>100 组）的测试。

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 updated memorydata/////////////////////////////////////////////////////////////////////////////////////
  SystemIO.printString(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 updated register//////////////////////////////////////////////////////////////
           SystemIO.printString(String.format("@%08x: $%2d <= %08x\n",programCounter.getValue() - 4, i, val));
           ////////////////////////////////////////////////////////////////////////////////////////////////////
           old = (Globals.getSettings().getBackSteppingEnabled())
                 ? Globals.program.getBackStepper().addRegisterFileRestore(num,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

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

```
# generate prj and tcl file
def initISE(prj):
    verilogPath = prj + "my_files\\cpu\\"
    prjFilePath = prj + "mips.prj"
    tclFilePath = prj + "mips.tcl"

    with open(prjFilePath, "w") as prjFile, open(tclFilePath, "w") as tclFile:
        for root, dirs, files in os.walk(verilogPath):
            for fileName in files:
                if re.match(r"[\w]*\.v", fileName):
                    prjFile.write("Verilog work " + root + "\\" + fileName + "\n")
        tclFile.write("run 20000us" + "\n" + "exit")

# compile and run verilog
def runISE(prj, code, out):
    prjFilePath = prj + "mips.prj"
    tclFilePath = prj + "mips.tcl"
    exeFilePath = prj + "mips.exe"
    logFilePath = prj + "log.txt"
    codeFilePath = prj + "code.txt"

    with open(code, "r") as codeSrc, open(codeFilePath, "w") as codeDst:
        codeDst.write(codeSrc.read())

    os.environ['XILINX'] = xilinxPath
    os.system(xilinxPath + "bin\\nt64\\fuse -nodebug -prj " + prjFilePath + " -o " + exeFilePath + " mips_tb > " + logFilePath)
    os.system(exeFilePath + " -nolog -tclbatch " + tclFilePath + " > " + out)
```

图 10 ISE 运行脚本

```
ISim P.20131013 (signature 0x7708f090)
WARNING: A WEBPACK license was found.
WARNING: Please use Xilinx License Configuration Manager to check out a full ISim license.
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
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
                145@00003000: $ 1 <= 00000d16
                155@00003004: $ 2 <= 00003368
                165@00003008: $ 3 <= 00006ade
                175@0000300c: $ 4 <= 00001728
                185@00003010: $ 5 <= 00006474
                195@00003014: $ 6 <= 00006784
                205@00003018: $ 7 <= 0000392c
                215@0000301c: $ 8 <= 000020a0
                225@00003020: $ 9 <= 00004e2e
                235@00003024: $10 <= 00006e33
                245@00003028: $11 <= 000056c2
                255@0000302c: $12 <= 00003b2d
                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

```python
# compare my and std
def cmp(my, std, res):
    with open(my, "r") as myFile, open(std, "r") as stdFile, open(res, "w") as out:
        myFileText = myFile.read()
        myLogs = re.findall("@[^\n]*", myFileText)
        stdLogs = re.findall("@[^\n]*", stdFile.read())
        asmLogs = re.findall("asm: [^\n]*", myFileText)

        isAC = True

        for i in range(len(stdLogs)):
            if i < len(myLogs) and myLogs[i] != stdLogs[i]:
                out.write("On Line " + str(i+1) + "\n")
                out.write("\tGet\t\t: " + myLogs[i] + "\n")
                out.write("\tExpect\t: " + stdLogs[i] + "\n")
                print("On Line " + str(i+1))
                print("\tGet\t: " + myLogs[i])
                print("\tExpect\t: " + stdLogs[i])
                if (i < len(asmLogs)):
                    out.write("\tAsm\t\t: " + asmLogs[i] + "\n")
                    print("\tAsm\t: " + asmLogs[i])
                isAC = False
                break
            elif i >= len(myLogs):
                out.write("myLogs is too short \n")
                print("myLogs is too short")
                isAC = False
                break
        if isAC :
            out.write("All Accepted")
            print("All Accepted")
    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。

```
bitwise = [1, 2, 2, 2, 2, 1, 1, 4, 4, 3, 3, 3, 3, 3, 3]
pre = [" 1'b", "`regAddr_", "`EXBack_", "`MEMBack_", "`WBBack_", "`ALUSrcA_", "`ALUSrcB_", "`ALU_",
       "`CMP_", "`EXT_", "`NPC_", "`DM_", " 3'd", " 3'd", " 3'd"]
lens = [9, 11, 14, 18, 17, 15, 15, 8, 8, 14, 28, 8, 5, 8, 9]

with open("D:\\study\\CO\\p5\\PipelineCPU\\my_files\\tools\\gnrt_ctrlsignal\\ctrltable.txt", "r") as
    ctrlsignals = ctrltable.read().splitlines()
    for i in range(len(ctrlsignals)):
        cnt = 0
        if i != 0:
            out.write(",")
        if ctrlsignals[i] == "-":
            out.write(" " + str(bitwise[i]) + "'b0")
            cnt = 4 + len(str(bitwise[i]))
        else:
            out.write(pre[i] + ctrlsignals[i])
            cnt = len(pre[i]) + len(ctrlsignals[i])
        while cnt < lens[i]:
            out.write(" ")
            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，即可得到一列实例化时用到的.pinName(pinName)格式的代码，复制到要实例化模块的地方即可。

```
from os import truncate
import re

with open("D:\\study\\CO\\p5\\tools\\gnrt_io\\iotable.txt", "r") as iotable, open("D:\\st
    io = iotable.read().splitlines()
    for i in range(len(io)):
        s = re.search(r"(?:input|output)(?: reg)?(?: \[[0-9]+\:0\])? ([\w]+)", io[i])
        if s:
            out.write("        ." + s.group(1) + "(")
            if (False):
                out.write(s.group(1))
            out.write("),\n")
        else:
            out.write("\n")
```

图 14 生成模块端口程序

3. 模拟 mips 运行：

用 c++模拟 mips 程序的运行，主要用于编写样例生成程序时 debug

```cpp
void trace()
{
    int reg[32] = {}, HI = 0, LO = 0, mem[1024] = {};
    int type, ins, a0, a1, a2, tag = 0;
    for (int i = 0; i < (int)instrs.size(); i ++)
    {
        type = instrs[i]->type;
        if (type == -1) continue;
        ins = instrs[i]->ins;
        a0 = instrs[i]->a0;
        a1 = instrs[i]->a1;
        a2 = instrs[i]->a2;
        if (type == cal_ri)
        {
            if (ins == addi && (INT_MAX-reg[a1] < a2 || INT_MIN-reg[a1] > a2)) //overflow
                instrs[i]->ins = ins = addiu;
            if (ins == addi) reg[a0] = reg[a1] + a2;
            else if (ins == addiu) reg[a0] = reg[a1] + a2;
            else if (ins == slti) reg[a0] = reg[a1] < a2;
            else if (ins == sltiu) reg[a0] = (unsigned int)reg[a1] < (unsigned int)a2;
            else if (ins == andi) reg[a0] = reg[a1] & a2;
            else if (ins == ori) reg[a0] = reg[a1] | a2;
            else if (ins == xori) reg[a0] = reg[a1] ^ a2;
            else if (ins == sll) reg[a0] = reg[a1] << a2;
            else if (ins == srl) reg[a0] = (unsigned int)reg[a1] >> a2;
            else if (ins == sra) reg[a0] = reg[a1] >> a2;
            if (a0) printf("$%2d <= %08x\n", a0, reg[a0]);
        }
```

图 15 模拟 mips 运行程序

4. 自动分析数据：

通过 python 自带的 zipfile 库对机器码进行压缩，再放入官方提供的分析程序中进行分析。

```python
# gnrt zipfiles
with zipfile.ZipFile(zfilePath + "P6_code.zip", "w") as zfile:
    for i in range(0, tot):
        tmpzfileName = "code" + str(i) + ".zip"
        with zipfile.ZipFile(tmpzfileName, "w") as tmpzfile:
            tmpzfile.write(codePath + "code" + str(i) + ".txt", "code.txt")
        zfile.write(tmpzfileName)
        os.remove(tmpzfileName)

# run analyzer
os.chdir(analysisPath)
os.system("python analyzer.py")
```

图 16 自动分析数据程序

三、思考题

（一）为什么需要有单独的乘除法部件而不是整合进 ALU？为何需要有独立的 HI、LO 寄存器？

乘除法时间较长，如果整合进 ALU 会使在 E 级的时间大大增长，从而增大整个 CPU 的周期，影响流水线效率。乘法非常容易溢出，所以可用两个寄存器 HI 和 LO 分别保存结果，使高位不丢失。对于除法来说，商和余数是同时在进行计算的，所以可以在算出结果时用 HI 和 LO 级寄存器同时保存两个值，而不需要进行两次独立的运算。并且由于乘除法的运算是相对独立的，使用独立的 HI 和 LO 不会影响到其他指令的执行和 GRF 的效率。

（二）参照你对延迟槽的理解，试解释 “乘除槽”。

延迟槽的作用是无论是否跳转都先执行下一条指令，避免了等待跳转而造成的周期的浪费。同理，由于乘除法时间较长，需要多个周期来进行运算。为了不影响与 HI 和 LO 无关的指令的执行，设置乘除槽用来装载这些指令，提高整个流水线的效率。

（三）举例说明并分析何时按字节访问内存相对于按字访问内存性能上更有优势。（Hint： 考虑 C 语言中字符串的情况）？

以 C 语言中字符串的处理为例，在 C 语言中，每个字符所占空间都是一个字节。在读取字符串时，以字节为单位比以整字为单位更具灵活性，如果一次读取整个字，则还需要对高 24 位进行清空或扩展，需要更多的指令来完成这一操作。在存储字符串时，如果以字为单位进存储，那么高 24 位的空间就会被浪费，并且在处理字节时也需要更多操作。

（四）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

自动生成样例程序采用的算法是：

对于 jal、jr、jalr 指令，使用特殊指令序列生成数据，以 cal_rr <~~ jal 为例：

```
for (int i = 0; i < insNum[cal_rr]; i ++)
{
    printf("jal TAG%d\n", ++ tagNum);
    printf("%s $1, $ra, $0\n", insName[cal_rr][i]);
    printf("TAG%d:\n", tagNum);
    printf("%s $2, $ra, $0\n", insName[cal_rr][i]);
    printf("%s $3, $ra, $0\n", insName[cal_rr][i]);
    insCnt += 4;
    puts("");
}
```

生成的样例为：（部分样例）

```
jal TAG1
add $1, $ra, $0
TAG1:
add $2, $ra, $0
```

```
add $3, $ra, $0

jal TAG2
addu $1, $ra, $0
TAG2:
addu $2, $ra, $0
addu $3, $ra, $0

jal TAG3
sub $1, $ra, $0
TAG3:
sub $2, $ra, $0
sub $3, $ra, $0
```

对于除了这三条 jump 指令之外的指令，采用随机生成+合法性检查的方法生成测试数据。

1. 先利用 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. 进行序列合法性检查。对于 DoubleDelay 和 JalSame 这样的问题，在生成时就可避免。对于 DivZero、Ov、AdEl 和 AdEs 的问题，通过模拟 mips 程序运行，更换指令或操作数来解决。

生成的样例为：

```
lui $4, 13
mfhi $1
bne $2, $2, TAG1
mfhi $4
TAG1:
bltz $4, TAG2
```

```
lh $4, 0($4)
multu $4, $4
div $4, $4
TAG2:
sb $4, 0($4)
lui $1, 10
srav $3, $4, $4
addiu $4, $3, 3
TAG3:
```

1 组特殊数据+100 组随机数据的分析结果如下，基本上达到了百分百覆盖：

图 17 样例分析结果

程序的不足:

目前程序在合法性检查上存在一点小 bug，平均每生成 20 组数据会有一组出问题，但不影响正常使用。而且有效转发率不够高，如果能进行优化那么所需要的数据量将大大减少。

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

由于采用了集中式译码和指令驱动型译码，为了避免代码膨胀，使用 assign 语句进行控制信号的赋值，并且指令的识别和控制信号的数值全部使用宏定义。在译码时能对每个控制信号的作用更加清晰，而不用对着一堆 0101 找错，而且在新增指令时更加简单，不易出错。

```
1   // ALUCtrl
2   `define ALU_and     4'b0000
3   `define ALU_or      4'b0001
4   `define ALU_add     4'b0010
5   `define ALU_sub     4'b0011
6   `define ALU_xor     4'b0100
7   `define ALU_sll     4'b0101
8   `define ALU_srl     4'b0110
9   `define ALU_lt      4'b0111
10  `define ALU_ltu     4'b1000
11  `define ALU_sra     4'b1001
12  `define ALU_nor     4'b1010
13  `define ALU_slc     4'b1011
14  `define ALU_src     4'b1100
15
16  // MDUCtrl
17  `define MDU_none    4'b0000
18  `define MDU_mult    4'b0001
19  `define MDU_multu   4'b0010
20  `define MDU_div     4'b0011
21  `define MDU_divu    4'b0100
22  `define MDU_mthi    4'b0101
23  `define MDU_mtlo    4'b0110
24
25  // MDUState
26  `define MDUState_begin      2'b00
27  `define MDUState_mulDelay   2'b01
28  `define MDUState_divDelay   2'b10
```

图 18 宏定义文件

笔者并没有对指令先分类然后再生成控制信号，主要是因为担心课上“缝合怪”指令的出现，如果对每条指令都进行一遍控制信号的赋值，则能减少新增指令的难度。而在课下由于使用了自动生成控制信号程序，并没有在敲代码上花太

多时间。