计算机组成原理实验报告

20231164 张岳霖

一、CPU 设计方案综述

（一）总体设计概述

本 CPU 为 Verilog 实现的流水线 MIPS - CPU，共包含 F、D、E、M 和 W 五级，支持的指令集包含{add、addu、sub、subu、slt、sltu、and、or、nor、xor、sllv、srlv、srav、sll、srl、sra、jr、jalr、mfhi、mflo、mthi、mtlo、mult、multu、div、divu、addi、addiu、slti、sltiu、andi、ori、xori、lui、sw、sh、sb、lw、lh、lhu、lb、lbu、beq、bne、bgtz、blez、bgez、bltz、j、jal、mfc0、mtc0、exet}共 53 条。为了实现这些功能，CPU 主要包含了 PC、FlowReg_D、RF、NPC、CMP、EXT、controller、FlowReg_E、ALU、multdiv、CP0、errorjudger、FlowReg_M、FlowReg_W、RiskSolveUnit，共 15 个模块，外设包括 IM、DM、timecounter0、timecounter1、systembridge，共 5 个模块，在顶层模块文件 mips.v 中连接 CPU 和外设。

（二）关键模块定义

1. PC

模块功能：程序计数器

模块定义：

```
module PC (
    input  wire clk,
    input  wire reset,
    input  wire pcenable,
    input  wire [2:0] npcop,
    input  wire [31:0] pc_D_B,
    input  wire [31:0] pc_D_JJal,
    input  wire [31:0] pc_D_Jr,
```

```
    input  wire branch,
    input  wire jumpto4180,
    output reg  [31:0] currentpc_F
);
```

表一 PC 信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号，时钟上升沿更新当前指令地址为下一条指令地址
2	reset	I	1	同步复位信号，信号为 1 时将指令初始化为 0x0000_3000
3	pcenable	I	1	使能信号，为 0 时 PC 寄存器保持原有值不变，用于阻塞。
4	npcop	I	3	下一条指令计算方法选择信号
5	PC_D_B	I	32	B 类指令的地址
6	PC_D_JJal	I	32	J 和 Jal 指令的跳转地址
7	PC_D_Jr	I	32	Jr 指令的跳转地址
8	branch	I	1	B 类指令满足跳转条件信号，满足条件时信号为 1
9	jumpto4180	I	1	地址跳转到 0x4180，高电平时 F 级指令地址跳转为 0x4180，优先级最高

10	currentpc_F	O	32	输入到指令寄存器和 D 级流水线寄存器的地址信号

npcop 选择信号对应表

npcop	对应的 npc 计算方法
3’d0	PC+4
3’d1	PC_B
3’d2	PC_JJal
3’d3	PC_Jr
3’d4	PC_eret

2. FlowReg_D

模块功能：D 级流水线寄存器

模块定义：

```
module FlowReg_D (
    input  wire clk,
    input  wire En_D,
    input  wire reset,
    input  wire clear_D,
    input  wire jumpto4180,
    input  wire wrongpc_F,
    input  wire [31:0] command_F,
    input  wire [31:0] commandAddr_F, pc_eret,

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

表二 D 级流水线寄存器信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号
2	En_D	I	1	D 级寄存器使能
3	reset	I	1	D 级寄存器复位信号
4	clear_D	I	1	D 级寄存器复位信号，用于清空延迟槽
5	jumpto4180	I	1	跳转至 0x4180 信号
6	wrongpc_F	I	1	F 级 PC 异常信号
7	command_F	I	32	F 级指令信号
8	commandAddr_F	I	32	F 级当前指令地址
9	pc_eret	I	32	CP0 EPC 寄存器中保存的指令地址
10	wrongpc_D	I	1	D 级 PC 异常信号
11	Command_D	O	32	D 级指令信号
12	commandAddr_D	O	32	D 级当前指令地址

3. RF

模块功能：寄存器堆

模块定义：

```
module RF (
    input  wire clk,
    input  wire reset,
    input  wire [4:0] a1,
    input  wire [4:0] a2,
```

```
    input  wire [4:0] a3,
    input  wire rfwe,
    input  wire [31:0] rfwd,
    input  wire [31:0] commandAddr_W,
    output wire [31:0] rfrd1,
    output wire [31:0] rfrd2
);
```

表三 RF 信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位信号,信号为 1 将 32 个寄存器全部清零
3	a1	I	5	地址输入信号，指定 32 个寄存器中的某一个，将其中存储的数据读入到 rfrd1
4	a2	I	5	地址输入信号，指定 32 个寄存器中的某一个，将其中存储的数据读入到 rfrd2
5	a3	I	5	地址输入信号，指定 32 个寄存器中的某一个作为写入的目标寄存器
6	rfwe	I	1	GRF 写入使能信号，1 写入有效
7	rfwd	I	32	32 位数据输入信号
8	commandAddr_W	I	32	W 级当前指令地址，评测需要
9	rfrd1	O	32	输出 A1 指定的寄存器中的 32

				位数据
10	rfrd2	O	32	输出 A2 指定的寄存器中的 32 位数据

注 RF 采用内部转发，W 级写入 RF 的数据对 D 级可见。

实现方法为：

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

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

4. NPC

模块功能：下一指令地址计算模块

模块定义：

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

表四 NPC 信号定义

序号	信号名称	数据方向	位数	功能描述
1	commandAddr_D	I	32	D 级当前指令地址
2	command_D	I	32	D 级当前指令
3	rfrd1_D	I	32	D 级 RF 的 RD1 信号通路中数据，Jr 指令的跳转地址
4	pc_D_B	O	32	计算出的 B 类指令 NPC

5	pc_D_JJal	O	32	计算出的 Jal/J 指令 NPC
6	pc_D_Jr	O	32	计算出的 Jr 指令 NPC
7	pc8_D	O	32	当前指令+8 后的地址，Jal 指令向$ra 寄存器存储该地址

5. CMP

模块功能：比较器

模块定义：

```
module CMP (
    input  wire [31:0] cmpa,
    input  wire [31:0] cmpb,
    input  wire [3:0]  cmpop,
    output reg branch_true
);
```

表五 CMP 信号定义

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

6. EXT

模块功能：位扩展器

模块定义：

```
module EXT (
    input  wire [15:0] immoffset,
    input  wire [1:0] extop,
```

```
    output wire [31:0] extres
);
```

表六 位扩展器信号定义

序号	信号名称	数据方向	位数	功能描述
1	immoffset	I	16	待扩展数据
2	EXTOp	I	2	扩展方式选择信号：零扩展(0)，符号扩展(1)，立即数 10-6 位零扩展
3	EXTRes	O	32	扩展后的数据

7. FlowReg_E

模块功能：E 级流水线寄存器

模块定义：

```
module FlowReg_E (
    input  wire clk, CLR_E, rfwe_D, dmwe_D, reset, startmd_D, unknowinstr_D, wrongpc_D,
jumpto4180,
    input  wire [1:0] rfwdsel_D, rfwrsel_D, asel_D, bsel_D,
    input  wire [2:0] tnew_D, dmtype_D,
    input  wire [3:0] multdivop_D,
    input  wire [4:0] aluop_D, commandtype_D, read1addr_D, read2addr_D,
    input  wire [31:0] rfrd1_D, rfrd2_D, pc8_D, extres_D, command_D, commandAddr_D, pc_eret,

    output reg rfwe_E, dmwe_E, startmd_E, unknowinstr_E, wrongpc_E,
    output reg [1:0] rfwdsel_E, rfwrsel_E, asel_E, bsel_E,
    output reg [2:0] tnew_E, dmtype_E,
    output reg [3:0] multdivop_E,
    output reg [4:0] aluop_E, commandtype_E, read1addr_E, read2addr_E,
    output reg [31:0] rfrd1_E, rfrd2_E, pc8_E, extres_E, command_E, commandAddr_E
```

);

表七 E 级流水线寄存器信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号
2	CLR_E	I	1	E 级寄存器清零信号
3	reset	I	1	E 级寄存器复位信号
4	rfwe_D	I	1	D 级指令 RF 写使能信号
5	dmwe_D	I	1	D 级指令 DM 写使能信号
6	startmd_D	I	1	D 级指令开始进行乘除运算信号
7	unknowinstr_D			D 级指令未知信号
8	wrongpc_D			D 级指令地址异常信号
9	jumpto4180			跳转至 0x4180 信号
10	rfwesel_D	I	2	D 级指令 RF 写入数据选择信号
11	rfwrsel_D	I	2	D 级指令 RF 写入地址选择信号
12	asel_D	I	2	D 级指令 ALU A 操作数选择信号
13	bsel_D	I	2	D 级指令 ALU B 操作数选择信号
14	dmtype_D	I	3	D 级指令 DM 写入类型选择信号
15	tnew_D			D 级指令新数据产生时间
16	multdivop_D	I	4	D 级指令乘除运算单元操作选择信号
17	aluop_D	I	5	D 级指令 ALU 操作选择信号

18	commandtype_D	I	5	D 级指令类型
19	read1addr_D	I	5	D 级指令 RF 读取地址 1
20	read2addr_D	I	5	D 级指令 RF 读取地址 2
21	rfrd1_D	I	32	D 级指令 RF 读取数据 1
22	rfrd2_D	I	32	D 级指令 RF 读取数据 2
23	pc8_D	I	32	D 级指令的地址+8
24	extres_D	I	32	D 级指令立即数扩展结果
25	command_D	I	32	D 级指令
26	commandAddr_D	I	32	D 级指令的地址
27	pc_eret			CP0 EPC 寄存器保存的异常指令地址
28	rfwe_E	O	1	E 级指令 RF 写使能信号
29	dmwe_E	O	1	E 级指令 DM 写使能信号
30	startmd_E			E 级指令开始进行乘除运算信号
31	unknowinstr_E			E 级指令未知信号
32	wrongpc_E			E 级指令地址异常信号
33	rfwesel_E	O	2	E 级指令 RF 写入数据选择信号
34	rfwrsel_E	O	2	E 级指令 RF 写入地址选择信号
35	asel_E	O	2	E 级指令 ALU A 操作数选择信号
36	bsel_E	O	2	E 级指令 ALU B 操作数选择信号

37	dmtype_E	O	3	E 级指令 DM 写入类型选择信号
38	tnew_E	O	3	E 级指令 Tnew 信号
39	multdivop_E			E 级指令乘除运算单元操作选择信号
40	aluop_E	O	5	E 级指令 ALU 操作选择信号
41	commandtype_E			E 级指令乘除运算单元操作选择信号
42	read1addr_E	O	5	E 级指令 RF 读取地址 1
43	read2addr_E	O	5	E 级指令 RF 读取地址 2
44	rfrd1_E	O	32	E 级指令 RF 读取数据 1
45	rfrd2_E	O	32	E 级指令 RF 读取数据 2
46	pc8_E	O	32	E 级指令的地址+8
47	extres_E	O	32	E 级指令立即数扩展结果
48	command_E	O	32	E 级指令
49	commandAddr_E	O	32	E 级指令的地址

8. ALU

模块功能：算术运算单元

模块定义：

```
module ALU (
    input  wire [31:0] a,
    input  wire [31:0] b,
    input  wire [4:0] aluop,
    output  wire overflow,
    output wire [31:0] res
```

```
);
```

表八 ALU 信号定义

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

ALU 运算定义表

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

9. multdiv

模块功能：乘除法运算单元

模块定义：

```
module multdiv (
    input  wire clk, reset, start, jumpto4180,
    input  wire [3:0] multdivop,
    input  wire [31:0] a, b,
    output wire busy,
    output reg [31:0] registerhi,
    output reg [31:0] registerlo
);
```

表九 multdiv 信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位信号
3	start	I	1	开始进行乘除运算信号
4	jumpto4180	I	1	跳转至 0x4180 信号
5	multdivop	I	4	乘除运算方法选择信号
6	a	I	32	a 操作数
7	b	I	32	b 操作数
8	busy	O	1	乘除运算单元繁忙信号
9	registerhi	O	32	hi 寄存器中数值
10	registerlo	O	32	lo 寄存器中数值

10. CP0

模块功能：协处理器

模块定义：

```
module CP0 (
    input  wire clk, reset, isDelay,
    input  wire [4:0] errorcode,
    input  wire [5:0] interruptrequest,
    input  wire [31:0] command_E, write_data, pc_E,
    output wire jumpto4180, interruptout,
    output wire [31:0] pc_eret,
    output wire [31:0] readfromcp
);
```

表十 CP0 信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位信号
3	isDelay	I	1	当前指令是否为延迟槽指令
4	errorcode	I	5	当前指令的异常码
5	interruptrequest	I	6	六位中断信号
6	command_E	I	32	E 级指令信号
7	write_data	I	32	向 CP0 中寄存器写入的数据
8	pc_E	I	32	E 级指令地址
9	jumpto4180	O	1	跳转到 0x4180
10	interruptout	O	1	响应外部中断信号时置 1
11	pc_eret	O	32	EPC 中保存的异常指令地址
12	readfromcp	O	32	从 CP0 模块中读到的数据

11. errorjudger

模块功能：指令异常判断器

模块定义：

```
module errorjudger (
    input  wire unknowinstr, overflow, wrongpc_E,
    input  wire [4:0] commandtype_E,
    input  wire [31:0] command_E, commandAddr_E, alures,
    output reg [4:0] errorcode
);
```

表十一 errorjudger 信号定义

序号	信号名称	数据方向	位数	功能描述
	unknowinstr	I	1	未知指令信号
	overflow	I	1	算术溢出信号
	wrongpc_E	I	1	地址异常信号
	commandtype_E	I	5	E 级指令类型
	command_E	I	32	E 级指令
	commandAddr_E	I	32	E 级指令地址
	alures	I	32	ALU 计算指令
	errorcode	O	5	异常码

12. FlowReg_M

模块功能：M 级流水线寄存器

模块定义：

```
module FlowReg_M (
    input  wire clk, rfwe_E, dmwe_E, reset,
    input  wire [1:0] rfwdsel_E, rfwrsel_E,
```

```
    input  wire [2:0] tnew_E, dmtype_E,
    input  wire [4:0] read2addr_E, commandtype_E,
    input  wire [31:0] ALUres_E, registerhi_E, registerlo_E, wddm_E, pc8_E,
command_E, commandAddr_E,

    output reg rfwe_M, dmwe_M,
    output reg [1:0] rfwdsel_M, rfwrsel_M,
    output reg [2:0] tnew_M, dmtype_M,
    output reg [4:0] read2addr_M, commandtype_M,
    output reg [31:0] ALUres_M, registerhi_M, registerlo_M, wddm_M, pc8_M,
command_M, commandAddr_M
);
```

表十二 M 级流水线寄存器信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	M 级寄存器时钟信号
2	reset	I	1	M 级寄存器复位信号
3	rfwe_E	I	1	E 级指令 RF 写使能信号
4	dmwe_E	I	1	E 级指令 DM 写使能信号
5	rfwdsel_E	I	2	E 级指令 RF 写入数据选择信号
6	rfwrsel_E	I	2	E 级指令 RF 写入地址选择信号
7	dmtype_E	I	3	E 级指令 DM 写入类型选择信号
8	tnew_E	I	3	E 级指令 Tnew 信号
9	read2addr_E	I	5	E 级指令 RF 读取地址 2

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

29	wddm_M	O	32	M 级指令 DM 输入数据
30	pc8_M	O	32	M 级指令地址+8
31	command_M	O	32	M 级指令
32	CommandAddr_M	O	32	M 级指令的地址

13. FlowReg_W

模块功能：W 级流水线寄存器

模块定义：

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

表十三 E 级流水线寄存器信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	W 级寄存器时钟信号
2	reset	I	1	W 级寄存器复位信号
3	rfwe_M	I	1	M 级指令 RF 写使能信号

4	rfwdsel_M	I	2	M 级指令 RF 写入数据选择信号
5	rfwrsel_M	I	2	M 级指令 RF 写入地址选择信号
6	commandtype_M	I	5	M 级指令类型
7	ALUres_M	I	32	M 级指令 ALU 运算结果
8	registerhi_M	I	32	M 级指令 hi 寄存器数值
9	registerlo_M	I	32	M 级指令 lo 寄存器数值
10	DMRes_M	I	32	M 级指令 DM 输出数据
11	pc8_M	I	32	M 级指令地址+8
12	commandAddr_M	I	32	M 级指令地址
13	command_M	I	32	M 级指令
14	rfwe_W	O	1	W 级指令 RF 写使能信号
15	rfwdsel_W	O	2	W 级指令 RF 写入数据选择信号
16	rfwrsel_W	O	2	W 级指令 RF 写入地址选择信号
17	commandtype_W	O	5	W 级指令类型
18	ALUres_W	O	32	W 级指令 ALU 运算结果
19	registerhi_W	O	32	W 级指令 hi 寄存器数值
20	registerlo_W	O	32	W 级指令 lo 寄存器数值
21	DWRes_W	O	32	W 级指令 DW 输出数据
22	pc8_W	O	32	W 级指令地址+8

23	commandAddr_W	O	32	W 级指令地址
24	command_W	O	32	W 级指令

14. Controller

模块功能：控制器

模块定义：

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

表十四 控制器信号定义

序号	信号名称	数据方向	位数	功能描述
1	command	I	32	指令的机器码
2	startmd_D	O	1	D 级指令开始进行乘除运算信号
3	rfwe_D	O	1	D 级指令 RF 写使能信号
4	dmwe_D	O	1	D 级指令 DM 写使能信号
5	wrsel_D	O	2	D 级指令 RF 写入地址选择信号
6	wdsel_D	O	2	D 级指令 RF 写入数据选择信号
7	extop	O	2	D 级指令位扩展方法
8	asel_D	O	2	D 级指令 ALU A 操作数选择信号

9	bsel_D	O	2	D 级指令 ALU B 操作数选择信号
10	dmtype_D	O	3	D 级指令 DM 写入类型选择信号
11	unknowinstr	O	1	D 级指令为未知指令时置 1
12	npcop	O	3	D 级指令的下一条指令地址计算操作类型
13	tuse_Drs	O	3	D 级指令使用 RF 的 rs 地址对应的数据的时间
14	tuse_Drt	O	3	D 级指令使用 RF 的 rt 地址对应的数据的时间
15	tnew_D	O	3	D 级指令产生新的写入 RF 的时间
16	cmpop	O	4	D 级指令 cmp 操作类型选择信号
17	multdivop_D	O	4	D 级指令乘除运算单元操作选择信号
18	aluop_D	O	5	D 级指令 ALU 操作类型选择信号
19	commandtype_D	O	5	D 级指令类型

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

15. RiskSolveUnit

模块功能：冲突化解单元

模块定义：

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

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

    input  wire [31:0] command_D, command_E, command_M, command_W
);
```

表十五 冲突化解单元信号定义

序号	信号名称	数据方向	位数	功能描述
1	rfwe_E	I	1	E 级指令 RF 写使能信号
2	rfwe_M	I	1	M 级指令 RF 写使能信号
3	rfwe_W	I	1	W 级指令 RF 写使能信号
4	busy_E	I	1	E 级乘除单元繁忙指令
5	startmd_E	I	1	E 级乘除单元开始运算指令
6	read1addr_D	I	5	即将参与 D 级 CMP 比较的 A 操作数对应的寄存器地址
7	read2addr_D	I	5	即将参与 D 级 CMP 比较的 B 操作数对应的寄存器地址

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

23	tuse_Drs	I	3	D 级指令 GRF[rs]的使用时间
24	tuse_Drt	I	3	D 级指令 GRF[rt]的使用时间
25	tnew_E	I	3	E 级指令得到新的向 RF 写入的数据的时间
26	tnew_M	I	3	M 级指令得到新的向 RF 写入的数据的时间
27	stall_F	O	1	PC 寄存器冻结信号
28	stall_D	O	1	D 级流水线寄存器冻结信号
29	flush_E	O	1	E 级流水线寄存器清除信号
30	command_D	I	32	D 级指令信号
31	command_E	I	32	E 级指令信号
32	command_M	I	32	M 级指令信号
33	command_W	I	32	W 级指令信号

16. systembridge

模块功能：系统桥

模块定义：

```
module systembridge (
    input  wire we_cpu,
    input  wire [31:0] addr_cpu,
    input  wire [31:0] data_dm, data_timer0, data_timer1,
    output wire we_dm, we_timer0, we_timer1,
    output wire [31:0] addr_dm, addr_timer0, addr_timer1,
```

```
    output wire [31:0] data_to_cpu
);
```

表十六 冲突化解单元信号定义

序号	信号名称	数据方向	位数	功能描述
1	we_cpu	I	1	CPU 传来的写使能信号
2	addr_cpu	I	32	CPU 传来的写地址信号
3	data_dm	I	32	DM 传来的数据
4	data_time0	I	32	timer0 传来的数据
5	data_time1	I	32	timer1 传来的数据
6	we_dm	O	1	DM 写使能信号
7	we_time0	O	1	timer0 写使能信号
8	we_time1	O	1	timer1 写使能信号
9	addr_dm	O	32	DM 写入地址
10	addr_time0	O	32	timer0 写入地址
11	addr_time1	O	32	timer 写入地址
12	data_to_cpu	O	32	传给 CPU 的数据

17. timecounter

模块功能：计时器

模块定义：

```
module timecounter(
    input clk,
    input reset,
    input [31:2] addr,
```

```
input we,
input [31:0] Din,
output [31:0] Dout,
output IRQ
);
```

表十七 计时器信号定义

序号	信号名称	数据方向	位数	功能描述
1	clk	I	1	时钟信号
2	reset	I	1	同步复位信号
3	addr	I	30	写入地址信号
4	we	I	1	写使能信号
5	Din	I	32	输入的数据
6	Dout	O	32	读出的数据
7	IRQ	O	1	计时器中断信号

（三）控制模块设计

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

P7 搭建的 CPU 能够处理 53 条指令，与之前搭建的 CPU 相比，支持的指令数量有显著增加。为了简化控制单元设计，我采用了指令分类的方法，根本目的是将控制信号大部分相同的指令的控制信号在一个 if 块中输出。

1. 指令信号解码

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

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

Nop 指令：全零

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

mtc0 11'b01000000100

mfc0 11'b01000000000

eret 32'b01000010000000000000000000011000

PART1 OPCode == 0

```
`define  nop    6'b000000  // op == 0

// double register

`define  add    6'b100000  // op == 0

`define  addu   6'b100001  // op == 0

`define  sub    6'b100010  // op == 0

`define  subu   6'b100011  // op == 0

`define  slt    6'b101010  // op == 0

`define  sltu   6'b101011  // op == 0

`define  and    6'b100100  // op == 0

`define  or     6'b100101  // op == 0

`define  nor    6'b100111  // op == 0

`define  xor    6'b100110  // op == 0

`define  sllv   6'b000100  // op == 0

`define  srlv   6'b000110  // op == 0

`define  srav   6'b000111  // op == 0

// shift

`define  sll    6'b000000  // op == 0
```

```
    `define  srl     6'b000010  // op == 0
    `define  sra     6'b000011  // op == 0
    // jump register
    `define  jr      6'b001000  // op == 0
    // jump register and link
    `define  jalr    6'b001001  // op == 0
    // mult and div
    `define  mfhi    6'b010000  // op == 0
    `define  mflo    6'b010010  // op == 0
    `define  mthi    6'b010001  // op == 0
    `define  mtlo    6'b010011  // op == 0
    `define  mult    6'b011000  // op == 0
    `define  multu   6'b011001  // op == 0
    `define  div     6'b011010  // op == 0
    `define  divu    6'b011011  // op == 0
PART2  OPCode != 0
    // register and imm
    `define  addi    6'b001000  // op != 0
    `define  addiu   6'b001001  // op != 0
    `define  slti    6'b001010  // op != 0
    `define  sltiu   6'b001011  // op != 0
    `define  andi    6'b001100  // op != 0
    `define  ori     6'b001101  // op != 0
    `define  xori    6'b001110  // op != 0
```

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

    `define  mtc0   11'b01000000100
    `define  mfc0   11'b01000000000
    `define  eret   32'b01000010000000000000000000011000
```

2. 指令分类

根据指令的操作类型，我将指令分为一下几类：

a. jumpandlink(跳转并链接类)

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

包含：jal、jalr

b. calmudv（乘除法计算类）

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

包含：mult、multu、div、divu

c. readmudv(读取乘除运算结果类)

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

包含：mflo、mfhi

d. setmudv(对 hi 或 lo 寄存器赋值类)

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

包含：mtlo、mthi

e. caldoubleregister(双寄存器运算类)

此类指令的特点是两个寄存器中的数值进行运算，并将结果写入到第三个寄存器中。

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

f. shift(移位(立即数位）类)

包含指令：sll、srl、sra

g. calregisterimmsign(寄存器和符号扩展立即数计算类)

包含指令：addi、addiu、slti、sltiu

h. calregisterimmzero(寄存器和零扩展立即数计算类)

包含指令：andi、ori、xori、lui

i. save(存储类)

包含指令：sw、sh、sb

j. load(载入类)

包含指令：lw、lh、lhu、lb、lbu

k. aboutc0 (存取 c0 类)

包含指令：mfc0、mtc0

l. 未分类指令

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

3. 控制信号输出

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

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

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

下面列出各个指令的控制信号真值表：

表十八 控制信号逻辑真值表（一）

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

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

lw	0	0	1	1	1	0	1
lh	0	0	1	1	1	0	1
lhu	0	0	1	1	1	0	1
lb	0	0	1	1	1	0	1
lbu	0	0	1	1	1	0	1
beq	1	x	x	0	1	0	0
bne	1	x	x	0	1	0	0
bgtz	1	x	x	0	1	0	0
blez	1	x	x	0	1	0	0
j	2	x	x	0	x	0	0
jal	2	2	2	1	x	0	0
bgtz	1	x	x	0	1	0	0
bltz	1	x	x	0	1	0	0
nop	0	x	x	0	x	x	x
madd	0	x	x	0	x	x	x
maddu	0	x	x	0	x	x	x
msub	0	x	x	0	x	x	x
msubu	0	x	x	0	x	x	x
mfc0	0	x	x	1	x	x	x
mtc0	0	x	x	0	x	x	x
eret	4	x	x	0	x	x	x

表十九 控制信号逻辑真值表（二）

	aluop_D	dmwe_D	dmtype_D	tnew_D	tuse_Drs	tuse_Drt	cmp_op
add	1	0	x	2	1	1	x
addu	1	0	x	2	1	1	x
sub	2	0	x	2	1	1	x
subu	2	0	x	2	1	1	x
slt	3	0	x	2	1	1	x

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

sw	1	1	1	0	1	2	x
sh	1	1	2	0	1	2	x
sb	1	1	4	0	1	2	x
lw	1	0	1	3	1	5	x
lh	1	0	2	3	1	5	x
lhu	1	0	3	3	1	5	x
lb	1	0	4	3	1	5	x
lbu	1	0	5	3	1	5	x
beq	0	0	x	0	0	0	1
bne	0	0	x	0	0	0	2
bgtz	0	0	x	0	0	5	3
blez	0	0	x	0	0	5	6
j	0	0	x	0	5	5	x
jal	0	0	x	2	5	5	x
bgez	0	0	x	0	0	5	4
bltz	0	0	x	0	0	5	5
nop	0	x	0	0	5	5	x
madd	0	0	x	0	1	1	x
maddu	0	0	x	0	1	1	x
msub	0	0	x	0	1	1	x
msubu	0	0	x	0	1	1	x
mfc0	0	0	x	2	5	5	x
mtc0	0	0	x	0	5	1	x
eret	0	0	x	0	5	5	x

（四）CP0 模块设计

CP0 模块在 CPU 中的功能是记录 CPU 当前运行的状态并根据 CPU 当前状态和中断信号以及 errorjudge 模块传来的 error 信号，判断 CPU 在下一个时钟周

期是否要进入中断/异常处理状态。

在 CP0 中，我定义了 state_register, cause_register, epc_register, prid_register 这四个寄存器，并通过组合逻辑根据当前 E 级指令信号（我的 CP0 模块位于 CPU 的 E 级）以及 CPU 当前的 state 来判断是否要响应异常中断或是 eret 指令从异常处理返回用户态。同时在下一个时钟上升沿改变 CP0 中对应寄存器中的数值，以记录当前状态。

在实现过程中，CP0 还要接收当前指令是否为延迟槽指令（存储 EPC 时需要存 PC-4）并传出是否响应了外部中断信号（在下一个下降沿要确保向 7f20 写入数据从而终止外部中断）。我根据 M 级指令类型，判断 E 级指令是否为延迟槽指令：E 级指令为延迟槽指令当且仅当 M 级指令为跳转类指令；是否响应了外部中断信号通过 HWInt[4]、IM[4]和 IE 通过组合逻辑给出。在 CP0 得到已响应外部中断信号后的时钟上升沿，将一个 reg 类型的中间变量置 1，目的是保存当前已经响应外部中断，确保在接下来的时钟下降沿时中间变量仍然可以处于高电平。当中间变量处于高电平时，我们向 0x7f20 写入值，这样我们就可以确保在响应外部中断后的时钟下降沿终止外部中断信号，外部中断信号只能存在一个时钟周期。

（五）Bridge 和 IO 设计

Bridge 在我的 CPU 中的功能仅具有多路分配器的功能，根据当前 M 级的指令地址判断 CPU 当前的操作对象是 DM，Timer0，Timer1 中的哪一个，以及 CPU 将从三者中哪一个部件读入数据。

IO 端口是在 mips_tb 文件中的，可以在其中设定中断次数以及外部中断对应的宏观 PC。

```
reg [31:0] need_interrupt = 100;

always @(negedge clk) begin

    if (~reset) begin

        if (need_interrupt == 32'd100 && fixed_macroscopic_pc == 32'h3010) begin

# 实现在 0x3010 处中断

                interrupt <= 1;

                need_interrupt <= need_interrupt - 1;
```

```
        end
    # 此处可添加中断信号
    end
end
```

（六）重要机制实现方法

1. 跳转

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

2. 流水线延迟槽

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

3. 转发

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

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

数据的流水级（M 级和 W 级）不难得知转发的位点有且只有三处：

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

转发数据来源：

a. M 级 PC+8（pc8_M）

b. M 级 ALU 的结果（ALUres_M）

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

（2） E 级 ALU 的两个输入端

转发数据来源：

a. M 级 PC+8（pc8_M）

b. M 级 ALU 的结果（ALUres_M）

c. W 级回写寄存器的数据（res_W）

d 沿流水线寄存器传递到 E 级 ALU 输入端的数据（rfrd_E）

（3） M 级 DM 的数据输入端

转发数据来源：

a. W 级回写寄存器的数据（res_W）

b 沿流水线寄存器传递到 M 级 DM 数据输入端的数据（wddm_M）

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

```
if(rfwe_M == 1 && read1addr_E == writeaddr_M && read1addr_E == 5'd31 &&
                                                command_M[31:26] == `jal)begin
   forward_ALUa_E = 2'd3;
end
```

```
else if(rfwe_M == 1 && read1addr_E == writeaddr_M && read1addr_E != 0) begin
    forward_ALUa_E = 2'd2;
end
else if(rfwe_W == 1 && read1addr_E == writeaddr_W && read1addr_E != 0) begin
    forward_ALUa_E = 2'd1;
end
else begin
    forward_ALUa_E = 2'd0;
end
```

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

4. 暂停

阻塞是要解决当一条指令的执行需要某个寄存器中的数值，但是该指令之前的指令即将改变该寄存器的值且得到改变数值的时间要大于需求该寄存器数值的时间，也就是说之前的指令无法在该指令需要该寄存器值之前得到更新该寄存器的值，这时候我们就不得不阻塞流水线 CPU 的运行，直到之前的指令能够在该指令需求前得到更新后的寄存器数值。

阻塞的实现采用的是教程中的供给时间模型。

对于某一个指令的某一个数据需求，我们定义需求时间 Tuse 为：这条指令位于 D 级的时候，再经过多少个时钟周期就必须要使用相应的数据。

对于某个指令的数据产出，我们定义供给时间 Tnew 为：位于某个流水级的某个指令，它经过多少个时钟周期可以算出结果并且存储到流水级寄存器里。

在设计过程中，我在 controller 模块进行指令译码的同时，赋予指令 Tnew_D、Tuse_Drs 以及 Tuse_Drt 的具体数值，其中 Tnew 为动态数值，在不同流水级中的数值不同，需要随流水线向下传递，其计算方法为 Tnew_n = max(Tnew_n-1,0) n 为流水线级数。通过分析得知，我们只需要在指令刚刚进入流水线（处于 D 级）时做出判断：通过比对 D 级指令的 Tuse_Drs、Tuse_Drt 和 E、M 级指令的 Tnew，当 Tnew>Tuse 需要暂停流水线，其余情况均可通过转发避免数据冲突。（注： 对于不产生新数据的指令，将 Tnew 设置成 0；对于不需要使用数据的指令，将 Tuse

设置成 5，这样能够确保不产生新数据或不需要新数据的指令不会引发暂停）。当我们得知需要对流水线做暂停操作时，即在 RiskSolveUnit 输出 PC 寄存器的冻结信号，D 级流水线寄存器的冻结信号以及 E 级流水线的清除信号，这样就相当于我们在 D 级指令之前插入了一个 nop 指令。如此，就完成了暂停机制的构建。

5. 异常

由于所有异常指令都可以在 E 级判断完毕，我在 E 级中加入了 errorjudge 模块，用于判断指令的异常行为，并生成异常指令对应的异常码传递至 CP0 模块。在 CP0 模块中，如果接收到异常码且当前未处于中断或异常处理状态，则会进入异常处理状态，产生异常信号（命名为 jumpto4180），该信号具有的功能包括清除 DEM 级流水线寄存器以及更改 F 级当前 PC 至 0x4080。同时在 CP0 模块中也会将 EXL 设为 1，用于标记当前已进入异常处理状态，并记录中断产生的指令地址以及是否为延迟槽指令等信息。

6. 中断

中断处理则是将外部中断信号以及两个计时器的中断信号传递至 CP0 模块中，CP0 模块会根据 CPU 当前转态以及是否中断使能来确定是否响应此中断，若响应此中断，则会向 cause 寄存器中写入当前六个中断位的中断状态，其余操作与异常处理类似。

在处理异常和中断时值得注意的是，由于我的宏观 PC 选取是 E 级 PC，当中断或者异常发生，清除 DE 流水线寄存器时需要将指令地址更改为中断或异常跳转的目标地址。即如果响应中断或异常，则在清除 DE 流水线寄存器时将指令地址更改为 0x4180；如果 eret 返回用户态，则在清除 DE 流水线寄存器时将指令地址更改为 pc_eret。这样我们才能确保 E 级的 macro_pc 始终为 CPU 处理完毕的下一条指令地址，不至于 E 级 macro_pc 出现为 0 的状态。

二、测试方案

（一）典型测试样例

1. 顺序执行测试

```
ori $at,123
ori $v0,456  # 构造正数

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

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

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

mult $t1,$t2
mfhi $t0
mflo $t1
mthi $s0
```

```
divu $t1,$t3
mflo $t5

ori $t6,0
ori $t7,1
nop
sw $t2,0($t6)
sh $t3,6($t6)
sb $t4,7($t6)
sw $t5,4($t6)

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

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

2. 转发测试

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

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

  # ALU（包括乘除类）+save 类
ori $t6,$t6,100
sw $6,4($0)
lui $t7,100
sh $t7,8($0)

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

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

  Jal or jalr+save 类
f2:
jal f3
sw $ra,24($0)
addu $k0,$ra,5
f3:
sw $s2,16($0)
```

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

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

3. 暂停测试

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

  # Jal+B 类
f1:
ori $s0,100
jal f2
ori $t2,$s0,320
f2:
beq $ra,$t2,f3
nop
f3:
```

```
sw $t0,0($0)
sw $t1,4($0)
sw $s0,8($0)
sw $t2,12($0)

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

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

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

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

  # 乘除法+乘除法
 mult $t0,$t1
```

```
mflo $t2

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

# 针对 P7 新增指令
mtc0 $t0,$14
eret
```

4. 跳转及延迟槽测试

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

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

f1:
subu $t4,$t1,$t0

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

```
f3:

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

  # B类跳转
beq $k0,$k1,f4
addu $t6,$t5,$k0
addu $k0,$k0,$k0
bgtz $k0,f5
nop

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

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

4. 中断和异常测试

构造教程中提到的异常类型

AdEL

```
j 0x3002
j 0x7f20
ori $t0,$t0,1
lw $t1,0($t0)
lbu $t2,0($t0)
lui $t1,0x7fff
lh $t2,0x7fff($t1)
```

AdES

```
sw $t0,2($0)
sh $t0,1($0)
sb $t0,0x7f03($0)
lui $t0,0x7fff
sh $t2,0x7fff($t0)
sw $t2,0x7f08($0)
```

Ov

```
lui $t0,0x7fff
lui $t1,0x7fff
lui $t2,0x8000
add $t3,$t0,$t1
addi $t4,$t0,0x7fff
sub $t5,$t2,$t0
```

中断信号可以通过外部中断和 Timer 计时器产生，在测试过程中可以调节 handler 中对两个timer设置的初始值和更改mips_tb中的外部中断产生对应的指令来改变中断信号产生的时间。在这里我构造了中断异常同时产生，eret 返回用户态紧跟异常或中断，连续中断以及连续异常等边界条件，进行中断和异常的测试。

例如连续外部中断：

```
if (need_interrupt == 32'd3 && fixed_macroscopic_pc == 32'h3010) begin
      interrupt <= 1;
      need_interrupt <= need_interrupt - 1;
end
if (need_interrupt == 32'd2 && fixed_macroscopic_pc == 32'h3014) begin
      interrupt <= 1;
      need_interrupt <= need_interrupt - 1;
end
if (need_interrupt == 32'd1 && fixed_macroscopic_pc == 32'h3018) begin
      interrupt <= 1;
      need_interrupt <= need_interrupt - 1;
end
```

在进行 P7 测试时，我发现的值得注意的点包括：

1．对于指令序列 mtc0 eret，eret 需要阻塞一个周期（理论上转发也可）

2．由于我的宏观 PC 选择的是 E 级 PC，所以在响应中断和异常以及从异常返回用户态清除流水线寄存器时，都要注意将流水线寄存器中的地址更改为恰当的数值

3．要注意当 mtc0 指令和乘除运算指令产生异常或中断时，不应改变 E 级 CP0 中寄存器以及 register_hi 和 register_lo 寄存器中保存的数值。

（二）自动测试工具（python 环境）

1. 测试样例生成器

```
# 测试样例生成器，包含 P7 的全部指令
import random

data1 = []  # 记录顺序执行的代码
data2 = []  # 记录跳转代码  # beq bne blez bgtz bltz bgez j    jal jalr jr
interval = []  # 记录中间插入部分代码的行数
helper = []  # 记录 sw 和 lw 之前 ori 指令的相关内容
```

```
    acnt = 0  # index of helper


    # 九个大指令类
    calmudv = ['mult', 'multu', 'div', 'divu']
    readmudv = ['mflo', 'mfhi']
    setmudv = ['mtlo', 'mthi']
    caldoubleregister = ['add', 'addu', 'sub', 'subu', 'slt', 'sltu', 'and', 'or', 'nor',
'xor', 'sllv', 'srlv', 'srav']
    shift = ['sll', 'srl', 'sra']
    calregisterimm = ['addi', 'addiu', 'andi', 'ori', 'xori']
    saveload = ['sw', 'sh', 'sb', 'lw', 'lh', 'lhu', 'lb', 'lbu']
    shifti = ['slti', 'sltiu']
    lui = ['lui']
    aboutc0 = ['mfc0', 'mtc0']


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



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

```
    def pre_fill():
        for i in range(2, 28):
            print("lui ${},{}".format(i, random.randint(0, 65535)))
            print("ori ${},${},{}".format(i, i, random.randint(0, 65535)))
        for i in range(30):
            print("sw ${},{}(${})".format(random.randint(0, 27), i * 4, 0))



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

```
random.randrange(0, 8192, 4)))
        elif data1[x].type == 8:
            print("{} ${},${},{}".format(random.choice(shifti), data1[i].rs, data1[i].rt,
random.randint(0, 31)))
        elif data1[x].type == 9:
            print("{} ${},{}".format(random.choice(lui), data1[i].rt, random.randint(0,
65535)))
        elif data1[x].type == 10:
            print("{} ${}, ${}".format(random.choice(aboutc0), data1[i].rs,
random.randint(12, 14)))
        else:
            print("nop")



    if __name__ == "__main__":
        pre_fill()
        total_blocks = random.randint(1, 25)  # 代码块数
        total_lines = 0  # 代码行数
        cnt = 0
        # 构建过程
        for i in range(total_blocks):  # 代码块头尾记录在 data2 中
            data2.append(Instruction(random.randint(0, 3), random.randint(1, 25)))
            total_lines += data2[i].lines
        for i in range(total_blocks + 1):  # 两个 interval 之间夹一个代码块
            interval.append(random.randint(1, 30))
            total_lines += interval[i]
        for i in range(0, total_lines):  # 随机生成顺序执行代码，保存在 data1 中
            data1.append(Instruction(random.randint(1, 11)))
            data1[i].rs = random.randint(2, 31)
            data1[i].rt = random.randint(2, 27)
```

```
            data1[i].rd = random.randint(2, 27)


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


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


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

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

```
            #     print("jalr ${}, ${}")
            print("ori $3,$3,100")  # 测试延迟槽
            print("ori $4,$4,200")
            print("interval{}:".format(cnt))
            cnt += 1
            for j in range(data2[i].lines):
                print_instr(k)
                k += 1
            print("beq $0,$0,interval{}".format(cnt))
            print("ori $3,$3,100")  # 测试延迟槽
            print("ori $4,$4,200")
            print("funct{}:".format(i))
            print("jr $ra")
            print("ori $3,$3,100")  # 测试延迟槽
            print("ori $4,$4,200")

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

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

2. 自动执行脚本

```
import os
import re
```

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

    prjFile1 = open("testmine.prj", "w")
    for root, dirs, files in os.walk(r"E:\1-Project\P7\testprogram\modulemine"):
        for fileName in files:
            if re.match(r"[\w]*\.v", fileName):
                prjFile1.write("Verilog work " + root + "\\" + fileName + "\n")

    prjFile2 = open("testzqy.prj", "w")
    for root, dirs, files in os.walk(r"E:\1-Project\P7\testprogram\modulezqy"):
        for fileName in files:
            if re.match(r"[\w]*\.v", fileName):
                prjFile2.write("Verilog work " + root + "\\" + fileName + "\n")

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

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

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

    os.system(
        r"D:\ISESetup\14.7\ISE_DS\ISE\bin\nt64\fuse --nodebug --prj testzqy.prj -o
```

```
testmipszqy.exe mips_txt >CompileLogzqy.txt")


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


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



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

3. 正确性判定脚本

```
    import os
    import re
    import sys



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


        text1 = file1.read()
        text2 = file2.read()


        file1.close()
        file2.close()


        # 由于不同程序的结果输出数据的时间可能有差异，且Mars输出的结果不带有时间，所以采用正则表达式匹
配，并构造(指令地址，写入对象，写入数据三元组)
```

```
obj = re.compile(r".*?@(?P<pc>.*?): (?P<place>.*?) <= (?P<number>.*?)\n", re.S)
result1 = obj.finditer(text1)
result2 = obj.finditer(text2)

operation1 = []
operation2 = []

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

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

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

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

res.close()

if flag:
```

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


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

三、思考题

（一）

我们计组课程一本参考书目标题中有“硬件/软件接口”接口字样，那么到底什么是“硬件/软件接口”？（Tips：什么是接口？和我们到现在为止所学的有什么联系？）

硬件接口指的是两个硬件设备之间的连接方式，硬件之间可以通过接口传递信息。硬件接口既包括物理上的接口，还包括逻辑上的数据传送协议。软件接口指程序中具体负责在不同模块之间传输或接受数据的并做处理的类或者函数，能够赋予开发人员无需理解软件程序源码就可以了解和访问软件内部功能和工作机制的能力，如各种软件 API 和人与软件的交互界面等等。

在我们目前设计的 CPU 中，不同计算机硬件之间通过硬件接口相互连接。包括 CPU 与 IM 的接口、CPU 与系统桥的接口等等。硬件接口允许两个硬件之间进行数据的传输和沟通，能够将所有硬件的功能协调和整合起来。同时，两个相互连接的接口应该是高度匹配的，需要我们在构建时加以注意。

（二）

BE 部件对所有的外设都是必要的吗？

Bridge 的功能类似于多路分配器的功能，是通过分析 CPU 传入的地址信号，将得到的数据传输到恰当的 CPU 外设，我们设计 BE 部件的根本原因是 DM、Timer1 和 Timer2 共用一个地址信号，需要用 BE 模块区分操作对象。但是对于 CPU 产生的仅涉及单一外设的沟通信号时，BE 信号不是必要的，我们可以直接将 CPU 和单一外设相连，就如我们设计的 CPU 可以直接与 IM 相连。因此 BE 部件不是必要的。

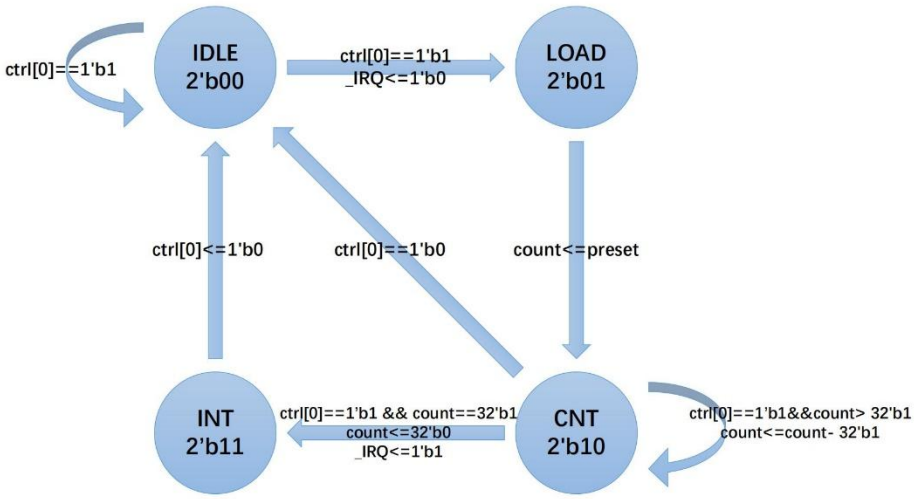
（三）

请阅读官方提供的定时器源代码，阐述两种中断模式的异同，并分别针对每一种模式绘制状态移图。

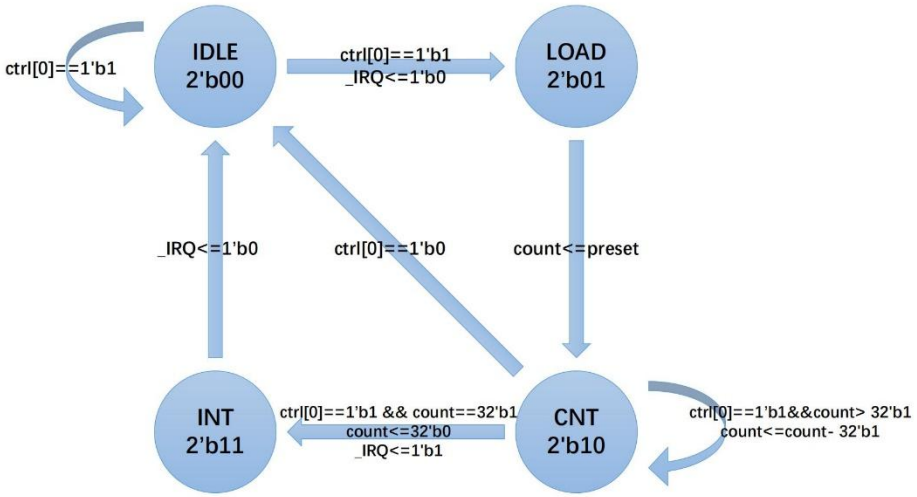
两种中断模式的相同点是都能够以初值寄存器的 32 位数值为初始值开始计数，每个时钟周期减一，当计数器为 0 时产生中断信号。

两种中断模式的不同点是计数器倒计数为 0 时，两个处理模式的处理方法不同。在模式 0 的条件下，计数器倒计数为 0 后，控制寄存器的使能端将自动变为 0，当使能端变为 1 后，初值寄存器值才再次被加载至计数器，计数器重新启动倒计数，模式 0 通常用于产生定时中断；在模式 1 的条件下，当计数器倒计数为 0 后，初值寄存器值被自动加载至计数器，计数器继续倒计数。模式 1 通常用于产生周期性脉冲。不同于模式 0，模式 1 下计数器每次计数循环中只产生一周期的中断信号

状态转移图：



模式0　ctrl[2:1]==2'b00



模式1 ctrl[2:1]==2'b01

（四）

请开发一个主程序以及定时器的 exception handler。整个系统完成如下功能：

（1）定时器在主程序中被初始化为模式 0；

（2）定时器倒计数至 0 产生中断；

（3）handler 设置使能 Enable 为 1 从而再次启动定时器的计数器。(2) 及 (3) 被无限重复。

（4）主程序在初始化时将定时器初始化为模式 0，设定初值寄存器的初值为某个值，如 100 或 1000。（注意，主程序可能需要涉及对 CP0.SR 的编程，推荐阅读过后文后再进行。）

```
.ktext 0x4180

   ori  $t2, $0, 9

   sw  $t2, 0x7f00($0)

   sw  $t2, 0x7f10($0)  # 重新设定 ctrl

   mfc0  $t0, $14

   addi  $t0, $t0, 4

   mtc0  $t0, $14  # 对 EPC 加 4
```

```
    eret


.text
    ori $t0, $0, 1000
    sw  $t0, 0x7f04($0)  # 初始化 preset in timer0 为 1000


    ori $t1, $0, 100
    sw  $t1, 0x7f14($0)  # 初始化 preset in timer1 为 100


    ori $t2, $0, 9
    sw  $t2, 0x7f00($0)
    sw  $t2, 0x7f10($0)   # 设定 ctrl 为 32'b1001  允许产生中断_
```

模式 0_计数器使能

（五）

请查阅相关资料，说明鼠标和键盘的输入信号是如何被 CPU 知晓的？

键盘、鼠标这类的低速设备是通过中断请求的方式进行 IO 操作的。即当键盘上按下一个按键的时候，键盘会发出一个中断信号，中断信号经过中断控制器传到 CPU，然后 CPU 根据不同的中断号执行不同的中断响应程序，然后进行相应的 IO 操作，把按下的按键编码读到寄存器（或者鼠标的操作），最后放入内存中。