计算机组成原理 P8 MIPS 微系统实验报告

20373944 何天然

一、设计方案综述

（一）总体设计概述

本微系统包含了 Verilog 实现的流水线 MIPS-CPU、系统桥、计时器、外设控制器等，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、mfc0、mtc0、eret }共 53 条，支持 AdEL、AdES、RI、Ov 等异常、外设中断和读写。为了实现这些功能，CPU 主要包含 CPU、Clock、Bridge、IM、DM、TC、DT、IO、UART、CP0、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，这些模块共可分成四层，顶层为 CPU、Clock、Bridge、TC、DT、IO、UART，其中 CPU 包含 Controller 和 Datapath；Controller 包含 MainController 和 HazardSolveUnit；Datapath 包含 IF、ID、EX、MEM、IF_ID、ID_EX、EX_MEM、MEM_WB、CP0，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。

22. CPU：见 P8 部分

23. CP0: 见（四）

24. Bridge: 见（四）

25. TC: 见计时器说明文档

（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。

（四）异常、中断与外设读写

1. CP0（Coprocessor0）设计

模块定义：

```
module CP0 (
    input clk,
    input reset,
    input [4:0] A1,
    input [4:0] A2,
    input [31:0] WD,
    input [31:0] PC,
    input [4:0] excCode,
    input [5:0] HWInt,
    input WE,
    input EXLReset,
    input BD,
    output intExcReq,
    output reg [31:0] EPC,
    output [31:0] RD,
    output reg respTbInt
);
```

表 23 CP0 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	A1	I	5	CP0 寄存器读地址
4	A2	I	5	CP0 寄存器写地址
5	WD	I	32	CP0 寄存器写入数据
6	PC	I	32	宏观 PC
7	excCode	I	5	异常码
8	HWInt	I	6	外设中断
9	WE	I	1	CP0 寄存器写使能
10	EXLReset	I	1	EXL 复位信号
11	BD	I	1	延迟槽判断
12	intExcReq	O	1	中断异常响应信号
13	EPC	O	32	EPC 寄存器
14	RD	O	32	CP0 寄存器读出数据
15	respTbInt	O	1	响应 testbench 中断信号

内部逻辑说明：

内部含有 SR、Cause、EPC、PrID 四个寄存器，功能与 SMRL 一致。当写使能信号有效时，进行寄存器数据的写入。当异常码不为 0 且不在内核态（EXL!=1）时，响应异常；当全局中断使能有效、不在内核态、外部中断信号 HTInt 有效且相应位没有被屏蔽（(IM & HTInt) != 0）时，响应中断；当异常和中断需要同时响应时，中断优先级更高。进入 handler 时，将 EXL 设为 1，宏观 PC 写

入 EPC，宏观 BD 写入 Cause[31]，excCode 写入 Cause[6:2]，同时每周期更新 IP。

2. Bridge 与 IO 设计

模块定义：

```
module Bridge (
    input [31:0] addr,
    input [3:0] byteEn,
    input [31:0] DMRD,
    input [31:0] TC1RD,
    input [31:0] TC2RD,
    output [3:0] DMByteEn,
    output TC1Write,
    output TC2Write,
    output [31:0] RD
);
```

表 24 Bridge 端口定义

序号	信号名	方向	位数	描述
1	addr	I	32	读取或写入地址
2	byteEn	I	4	字节使能
3	DMRD	I	32	DM 读取数据
4	TC1RD	I	32	计时器 1 读取数据
5	TC2RD	I	32	计时器 2 读取数据
6	DMByteEn	O	4	DM 字节使能
7	TC1Write	O	1	TC1 写使能
8	TC2Write	O	1	TC2 写使能
9	RD	O	32	读取数据

内部逻辑说明：

采用和高老板 ppt 相同的方式，通过地址范围来判断是命中 DM、命中 TC1 还是命中 TC2。如果命中 DM，那么 DM 字节使能与原字节使能一致，否则为 0，同时读出数据选择 DM 的数据；如果命中 TC1，那么当原字节使能为 4'b1111 时，TC1 的写使能信号有效，否则无效，同时读出数据选择 TC1 的数据；对于 TC2 同理。

二、测试方案

（一）典型测试样例

1. testpoint0:测试除 AdEL_Instr 之外的异常

```
.ktext 0x4180
    mfc0 $k0, $12
    mfc0 $k0, $13
```

```
    mfc0 $k0, $14
    addiu $k0, $k0, 4
    mtc0 $k0, $14
    eret
    mfc0 $k0, $14

.text
    ori $s0, $0, 0x1001
    ori $s1, $0, 0x1
    mtc0 $s0, $12

    # AdEL_DM
    lw $2, 1($0)
    lh $2, 1($0)
    lhu $2, 1($0)
    li $3, 2147483647
    lw $2, 1($3)
    lw $2, -1($0)
    lw $2, 0x3000($0)
    lw $2, 0x7f1c($0)

    # AdEL_Timer
    lh $2, 0x7f00($0)
    lhu $2, 0x7f00($0)
    lb $2, 0x7f00($0)
    mtc0 $s1, $12
    lbu $2, 0x7f00($0)
    lh $2, 0x7f10($0)
    mtc0 $s0, $12
    lhu $2, 0x7f10($0)
```

```
lb $2, 0x7f10($0)
lbu $2, 0x7f10($0)

# AdES_DM
sw $2, 1($0)
sh $2, 1($0)
li $3, 2147483647
sw $2, 1($3)
sw $2, -1($0)
sw $2, 0x3000($0)
sw $2, 0x7f1c($0)

# AdES_Timer
sw $2, 0x7f08($0)
sh $2, 0x7f00($0)
mtc0 $s1, $12
sb $2, 0x7f00($0)
sw $2, 0x7f18($0)
mtc0 $s0, $12
sh $2, 0x7f10($0)
sb $2, 0x7f10($0)

# RI
movz $0, $0, $0

# Ov
lui $1, 0x7fff
lui $2, 0x7fff
lui $3, 0xffff
add $4, $1, $2
```

```
addi $4, $1, 0x7fff
sub $4, $1, $3

end:
beq $0, $0, end
nop
```

2. testpoint1：测试 AdEL_Instr 异常（跳转到错误地址引起）

```
.ktext 0x4180
    mfc0 $k0, $12
    mfc0 $k0, $13
    mfc0 $k0, $14
    addiu $k1, $k1, 8
    mtc0 $k1, $14
    eret
    mfc0 $k0, $14

.text
    ori $s0, $0, 0x1001
    ori $s1, $0, 0x1
    mtc0 $s0, $12

    # AdEL_Instr
    ori $k1, $0, 0x301c
    ori $1, $0, 0x300e
    ori $2, $0, 0x2fff
    ori $3, $0, 0x6ffd
    jr $1
    nop
    mtc0 $s1, $12
```

```
jr $2
nop
mtc0 $s0, $12
jr $3
nop
jalr $31, $1
nop
jalr $31, $2
nop
jalr $31, $3
nop

end:
beq $0, $0, end
nop
```

3. Testpoint2:测试延迟槽相关异常

```
.ktext 0x4180
    mfc0 $k0, $12
    mfc0 $k0, $13
    mfc0 $k0, $14
    addiu $k0, $k0, 8
    mtc0 $k0, $14
    eret
    mfc0 $k0, $14

.text
    ori $s0, $0, 0x1001
    ori $s1, $0, 0x1
    mtc0 $s0, $12
```

```
# BranchDelay
ori $1, $0, 2
sw $1, 0($0)
lw $2, 0($0)
beq $2, $2, TAG1
lw $0, 1($0)
TAG1:

add $2, $1, $0
mtc0 $s1, $12
beqz $2, TAG2
sw $0, 1($0)
TAG2:
mtc0 $s0, $12

mult $1, $1
j TAG3
mflo $1
TAG3:

jal TAG4
movz $0, $0, $0
TAG4:

lui $1, 0x7fff
lui $2, 0x7fff
lui $3, 0xffff
bne $1, $2, TAG5
add $4, $1, $2
```

```
TAG5:

mtc0 $s1, $12
bgtz $0, TAG6
mtc0 $s0, $12
TAG6:

end:
beq $0, $0, end
nop
```

4. Testpoint3: 测试计时器模式 0 与 IO

```
.text
    ori $t0, $0, 0x1c01 # IE -> 1, IM -> 6'b000111
    mtc0 $t0, $12
    ori $t1, $0, 7              # preset -> 7
    sw $t1, 0x7f04($0)
    ori $t1, $0, 11
    sw $t1, 0x7f14($0)
    ori $t2, $0, 9              # ctrl -> 4'b1001
    sw $t2, 0x7f00($0)
    sw $t2, 0x7f10($0)
    ori $t3, $0, 1              # ctrl -> 4'b0001
    ori $t4, $0, 8              # ctrl -> 4'b1000

    ori $s0, 100
    for_begin:
    beq $s1, $s0, for_end
        sw $t3, 0x7f00($0)
        sw $t3, 0x7f10($0)
```

```
        lw $a1, 0x7f08($0)
        mfc0 $a2, $13
        sw $t2, 0x7f00($0)
        sw $t2, 0x7f10($0)
    addiu $s1, $s1, 1
    j for_begin
    for_end:

    nop
    nop
    sw $t3, 0x7f00($0)
    sw $t3, 0x7f10($0)
    end:
    beq $0, $0, end
    nop

.ktext 0x4180
    mfc0 $a2, $12
    mfc0 $a2, $13
    mfc0 $a2, $14
    sw $t2, 0x7f00($0)
    sw $t2, 0x7f10($0)
    eret
```

5. Testpoint4: 测试计时器模式 1 与 IO

```
.text
    ori $t0, $0, 0x1c01 # IE -> 1, IM -> 6'b000111
    mtc0 $t0, $12
    ori $t1, $0, 7                  # preset -> 7
```

```
sw $t1, 0x7f04($0)
ori $t1, $0, 11
sw $t1, 0x7f14($0)
ori $t2, $0, 11          # ctrl -> 4'b1011
sw $t2, 0x7f00($0)
sw $t2, 0x7f10($0)
ori $t3, $0, 1           # ctrl -> 4'b0011
ori $t4, $0, 8           # ctrl -> 4'b1010

ori $s0, 100
for_begin:
beq $s1, $s0, for_end
    sw $t3, 0x7f00($0)
    sw $t3, 0x7f10($0)
    lw $a1, 0x7f08($0)
    mfc0 $a2, $13
    sw $t2, 0x7f00($0)
    sw $t2, 0x7f10($0)
addiu $s1, $s1, 1
j for_begin
for_end:

nop
nop
sw $t3, 0x7f00($0)
sw $t3, 0x7f10($0)
end:
beq $0, $0, end
nop
```

```
.ktext 0x4180
    mfc0 $a2, $12
    mfc0 $a2, $13
    mfc0 $a2, $14
    sw $t2, 0x7f00($0)
    sw $t2, 0x7f10($0)
    eret
```

6. Testpoint5~n：自动生成的数据，见测试样例生成器

（二）自动测试工具

1. 测试样例生成器

运行环境：win10 g++ 11.1.0

程序大致思路为：

① 利用 ori 初始化$28-$31 寄存器。

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 在生成代码块时只检查 DoubleDelay、JalrSame，对于其他可能产生的异常，作为异常的测试点

```
109 void gnrtBlock(int index, int& lastDst, int& JBFlag)
110 {
111     int type, ins, a0, a1, a2;
112     int dstReg[10] = {lastDst}, cnt = 1;
113     for (int i = 0; i < 4; i ++)
114     {
115         type = rand() % TYPENUM;
116         if (type == br_r2 || type == br_r1 || type == _jal || type == _jalr || type == _jr) // DoubleDelay
117         {
118             if (JBFlag)
119                 while (type == br_r2 || type == br_r1 || type == _jal || type == _jalr || type == _jr)
120                     type = rand() % TYPENUM, JBFlag = 0;
121             else JBFlag = 1;
122         }
123         else JBFlag = 0;
124         ins = rand() % insNum[type];
125         if (type == cal_ri) a0 = ranReg, a1 = dstReg[rand()%cnt], a2 = ranImm(15, 1), dstReg[cnt++] = a0;
126         else if (type == cal_rr) a0 = ranReg, a1 = dstReg[rand()%cnt], a2 = dstReg[rand()%cnt], dstReg[cnt++] = a0;
127         else if (type == br_r2) a0 = dstReg[rand()%cnt], a1 = dstReg[rand()%cnt], a2 = index;
128         else if (type == br_r1) a0 = dstReg[rand()%cnt], a1 = index;
129         else if (type == store) a0 = dstReg[rand()%cnt], a1 = 0, a2 = dstReg[rand()%cnt];
130         else if (type == load) a0 = ranReg, a1 = 0, a2 = dstReg[rand()%cnt], dstReg[cnt++] = a0;
131         else if (type == mf) a0 = ranReg, dstReg[cnt++] = a0;
132         else if (type == mt) a0 = dstReg[rand()%cnt];
133         else if (type == md) a0 = dstReg[rand()%cnt], a1 = dstReg[rand()%cnt];
134         else if (type == _lui) a0 = ranReg, a1 = ranImm(16, 1), dstReg[cnt++] = a0;
135         else if (type == _jal) a0 = index;
136         else if (type == _jr) a0 = dstReg[rand()%cnt];
137         else if (type == _jalr) { a1 = dstReg[rand()%cnt]; while ((a0 = ranReg) == a1); dstReg[cnt++] = a0; } // JalrSame
138         else if (type == _mfc0) a0 = ranReg, a1 = ranCP0Reg, dstReg[cnt++] = a0;
139         else if (type == _movz) a0 = a1 = a2 = 0;
140         instrs.push_back(new Instr(type, ins, a0, a1, a2));
141     }
142     instrs.push_back(new Instr(index));
143     lastDst = dstReg[cnt-1];
144 }
```

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 P6 基础上新增了对拍功能，以及全面测试外部中断

运行环境：win10 64 位 python 3.9.6

步骤 1：导出 code 与 handler

```
# dump text and handler (and run in Mars)
def runMars(asm, codeFilePath, out):
    path = os.path.dirname(codeFilePath) + "\\"
    code = path + "code.tmp"
    handler = path + "handler.tmp"
    os.system("java -jar " + marsPath + " db nc mc CompactDataAtZero a dump .text HexText " + code + " " + asm)
    os.system("java -jar " + marsPath + " db nc mc CompactDataAtZero a dump 0x00004180-0x00005180 HexText " + handler + " " + asm)
    # os.system("java -jar " + marsPath + " " + asm + " 4096 db nc mc CompactDataAtZero > " + out)
    with open(code, "r") as codeSrc, open(handler, "r") as handlerSrc, open(codeFilePath, "w") as codeDst:
        codeText = codeSrc.read()
        codeDst.write(codeText)
        for i in range(len(codeText.splitlines()), 1120):
            codeDst.write("00000000\n")
        codeDst.write(handlerSrc.read())
    os.remove(code)
    os.remove(handler)
```

步骤 2：生成 prj 和 tcl 文件

```
# gnrt 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 200us" + "\n" + "exit")

```

步骤 3：分别编译运行两个 CPU，进行仿真

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

    os.chdir(prj)
    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)

```

步骤 4：将两个运行结果进行文本比对，如果出错则给出错误信息

```python
# cmp myAns and stdAns
def cmp(interrupt, my, std, cmpRes):
    with open(my, "r") as myFile, open(std, "r") as stdFile, open(cmpRes, "a") as out:
        myLogs = re.findall("\@[^\n]*", myFile.read())
        stdLogs = re.findall("\@[^\n]*", stdFile.read())

        if interrupt != 0:
            out.write("interrupt at " + str(hex(interrupt)) + " : \n")
            print("interrupt at " + str(hex(interrupt)) + " : ")
        else:
            out.write("no interrupt : \n")
            print("no interrupt : ")

        for i in range(len(stdLogs)):
            if i < len(myLogs) and myLogs[i] != stdLogs[i]:
                out.write("\tOn Line " + str(i+1) + "\n")
                out.write("\tGet\t\t: " + myLogs[i] + "\n")
                out.write("\tExpect\t: " + stdLogs[i] + "\n")
                print("\tOn Line " + str(i+1))
                print("\tGet\t: " + myLogs[i])
                print("\tExpect\t: " + stdLogs[i])
                return False
            elif i >= len(myLogs):
                out.write("\tmyLogs is too short\n")
                print("\tmyLogs is too short")
                return False
        if len(myLogs) > len(stdLogs):
            out.write("\tmyLogs is too long\n")
            print("\tmyLogs is too long")
            return False
        out.write("\tAccepted\n")
        print("\tAccepted")
        return True
```

测试模式：

有三种测试模式选择：无外部（testbench）中断，对某一个 pc 产生外部中断，对所有 pc 分别产生一次外部中断信号。

图为自动修改 mips_tb 的部分

```python
# change interrupt position in testbench
def changeIntPos(tbPath, intPos):
    text = ""
    with open(tbPath, "r") as testbench:
        text = testbench.read()
        if intPos == 0:
            text = text.replace("`define INT", "`define noneINT")
        else:
            text = text.replace("`define noneINT", "`define INT")
            text = re.sub(r"fixed_macroscopic_pc == 32'h[0-9a-f]+",
                "fixed_macroscopic_pc == 32'h" + str(hex(intPos)).removeprefix("0x"), text)
    with open(tbPath, "w") as testbench:
        testbench.write(text)
```

（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
```

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")
```

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

```
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]);
        }
```

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。

```
# 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")
```

三、外设控制与读写功能的实现（P8 相关）

（一）顶层模块定义

1. Clock

模块定义：

```
module Clock(
   input          clk_fpga,
   output         clk_cpu,
   output         clk_IMDM
);
```

表 1 Clock 端口定义

序号	信号名	方向	位数	描述
1	Clk_fpga	I	1	FPGA 的时钟信号
2	Clk_cpu	O	1	CPU 的时钟信号
3	Clk_IMDM	O	1	IM、DM 的时钟信号

内部逻辑说明：

使用 IP 核生成，可将 FPGA 的 25MHz 时钟频率转换为 CPU 的 60MHz 时钟频率以及 IM、DM 的 120MHz 时钟频率。

2. CPU（Central Processing Unit）

模块定义：

```
module CPU (
    input clk,
    input reset,
```

```
    input [5:0] HWInt,
    input [31:0] instr_F,
    input [31:0] memRD,
    output [31:0] PC_F,
    output [31:0] memAddr,
    output [31:0] memWD,
    output [3:0] byteEn,
    output [31:0] PC_M,
    output regWrite,
    output [4:0] regAddr,
    output [31:0] regWD,
    output [31:0] PC_W,
    output respTbInt
);
```

表 2 CPU 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	HWInt	I	6	硬件中断
4	Instr_F	I	32	F 级指令
5	memRD	I	32	内存读取数据
6	PC_F	O	32	F 级 PC
7	memAddr	O	32	内存写入地址
8	memWD	O	32	内存写入数据
9	bytenEn	O	4	字节使能
10	PC_M	O	32	M 级 PC
11	regWrite	O	1	GRF 写使能
12	regAddr	O	5	GRF 写入地址
13	regWD	O	32	GRF 写入数据
14	PC_W	O	32	W 级 PC
15	respTbInt	O	1	响应 tb 中断信号

内部逻辑说明：

内部包含了控制器和数据通路，实现了控制器和数据通路间、CPU 和外部的数据与信号的交互

3. IM（Instruction Memory）

模块定义：

```
module IM (
    input clka,
    input [11:0] addra,
    output [31:0] douta
);
```

表 3 IM 端口定义

序号	信号名	方向	位数	描述
1	clka	I	1	时钟信号
2	addra	I	12	12 位地址
3	douta	O	32	读取数据

内部逻辑说明：

使用 IP 核生成，时钟频率经过调制，为 120MHz，内存大小为 4096 Words，需要等待一周期才能获得有效数据。

4. Bridge

模块定义：

```
module Bridge (
    input [31:0] addr,
    input [3:0] byteEn,
    input [31:0] DMRD,
    input [31:0] TCRD,
    input [31:0] DTRD,
    input [31:0] UARTRD,
```

```
    input [31:0] IORD,
    output [3:0] DMByteEn,
    output TCWrite,
    output [3:0] DTByteEn,
    output UARTWrite,
    output [3:0] IOByteEn,
    output [31:0] RD
);
```

表 4 Bridge 端口定义

序号	信号名	方向	位数	描述
1	addr	I	32	写入地址
2	byteEn	I	4	原字节使能
3	DMRD	I	32	DM 读取数据
4	TCRD	I	32	计时器读取数据
5	DTRD	I	32	数码管读取数据
6	UARTRD	I	32	串口读取数据
7	IORD	I	32	通用 IO 读取数据
8	DMByteEn	O	4	DM 字节使能
9	TCWrite	O	1	计时器写使能
10	DTByteEn	O	4	数码管字节使能
11	UARTWrite	O	1	串口写使能
12	IOByteEn	O	4	通用 IO 字节使能
13	RD	O	32	读取数据

内部逻辑说明：

通过地址范围来判断是命中 DM、命中 TC 还是别的外设。以 DM 为例，如果命中 DM，那么 DM 字节使能与原字节使能一致，否则为 0，同时读出数据选择 DM 的数据。

5. DM（Data Memory）

模块定义：

```
module Clock (
    input clka,
    Input [3:0] wea,
    input [11:0] addra,
    Input [31:0] dina,
    output [31:0] douta
);
```

表 5 DM 端口定义

序号	信号名	方向	位数	描述
1	clka	I	1	时钟信号
2	wea	I	4	字节使能
3	addra	I	12	12 位地址
4	dina	I	32	写入数据
5	douta	O	32	读取数据

内部逻辑说明：

使用 IP 核生成，时钟频率经过调制，为 120MHz，内存大小为 3072 Words，需要等待一周期才能获得有效数据。

6. TC（Timer Counter）

模块定义：

```
module TC(
    input clk,
    input reset,
    input [3:2] Addr,
    input WE,
    input [31:0] Din,
    output [31:0] Dout,
    output IRQ
);
```

表 6 TC 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	Addr	I	2	地址信号
4	WE	I	1	写使能信号
5	Din	I	32	写入数据
6	Dout	O	32	读取数据
7	IRQ	O	1	中断信号

内部逻辑说明：

与 p7 计时器无差别，见计时器说明文档

7. DT（Digital Tube）

模块定义：

```
module DT (
    input clk,
    input reset,
    input [3:0] byteEn,
    input Addr,
    input [31:0] WD_orig,
    output [31:0] RD,
    output [7:0] code0,
    output [7:0] code1,
    output [7:0] code2,
    output reg [3:0] sel0,
    output reg [3:0] sel1,
    output reg sel2
);
```

表 7 DT 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	byteEn	I	4	字节使能
4	Addr	I	1	地址信号
5	WD_orig	I	32	原始写入数据
6	RD	O	32	读取数据
7	Code0	O	8	0-3 位数码管编码
8	Code1	O	8	4-7 位数码管编码
9	Code2	O	8	第 8 位数码管编码
10	Sel0	O	4	0-3 位数码管选择信号
11	Sel1	O	4	4-7 位数码管选择信号
12	Sel2	O	1	第 8 位数码管选择信号

内部逻辑说明：

0-3 号和 4-7 号数码管分别为两个四位共阳极八段数码管，采用不断切换选择信号的方式进行动态显示。内部寄存器定义与官方给的相同，支持字节读写。

8. MiniUART（Mini Universal Asynchronous Receiver/Transmitter）

模块定义：

```
module MiniUART(
    Input CLK_I,
    Input [31:0] DAT_I,
    Output [31:0] DAT_O,
```

```
    Input RST_I,
    Input [4:2] ADD_I,
    Input STB_I,
    Input WE_I,
    Output ACK_O,
    Input RxD,
    Output TxD,
    Output reg interrupt,
    Input respInt
);
```

表 8 UART 端口定义

序号	信号名	方向	位数	描述
1	CLK_I	I	1	时钟信号
2	DAT_I	I	32	写入数据
3	DAT_O	O	32	读取数据
4	RST_I	I	1	同步复位信号
5	ADD_I	I	3	地址信号
6	STB_I	I	1	选通信号
7	WE_I	I	1	写使能信号
8	ACK_O	O	1	操作结束方式信号
9	RxD	I	1	接收数据
10	TxD	O	1	发送数据
11	interrupt	O	1	中断信号
12	respInt	I	1	中断响应信号

内部逻辑说明：

在官方的基础上进行了如下修改：把时钟频率调整到 60MHz（与自己的 cpu 一致），并且加入中断功能，当读取到完整有效数据时，中断信号持续输出 1，直到接收到来自 CPU 的响应信号后停止。

9. IO（General-Purpose Input Output）

模块定义：

```
module IO (
    input clk,
    input reset,
```

```
    input [3:0] byteEn,
    input [4:2] Addr,
    input [31:0] WD_orig,
    input [31:0] dips0_3,
    input [31:0] dips4_7,
    input [7:0] key,
    output [31:0] RD,
    output [31:0] LED
);
```

表 9 IO 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	byteEn	I	4	字节使能
4	Addr	I	3	地址信号
5	WD_orig	I	32	原始写入数据
6	dips0_3	I	32	0-3 组拨码开关
7	dips4_7	I	32	4-7 组拨码开关
8	key	I	8	按键开关
9	RD	O	32	读取数据
10	LED	O	32	LED 控制信号

内部逻辑说明：

控制 0-8 组共 128 位拨码开关、8 个按键开关以及 32 个 LED 灯。内部寄存器定义与官方一致，支持字节存取。

10.其他 IP 核（以有符号乘法器 Mult 为例）

模块定义：

```
module Mult (
    Input clk,
    Input [31:0] a,
    Input [31:0] b,
    Output [63:0] p
);
```

表 10 Mult 定义

序号	信号名	方向	位数	描述
1	clk	I	1	时钟信号
2	a	I	32	操作数 a
3	b	I	32	操作数 b
4	p	O	64	乘法结果

内部逻辑说明：

内部采用流水线结构，需要经过五个周期得到结果。分别用有符号乘法器 Mult，无符号乘法器 Multu，有符号除法器 Div，无符号除法器 Divu 代替原来 MDU 中乘除法运算，便可支持 MDU 相关指令。

（二）自动化工具

自动导出指令代码并生成 coe 文件：

```python
import os

marsPath = "G:\\mars\\Mars_test.jar"
prjPath = "D:\\study\\CO\\p8\\MIPSMicroSystem\\"
asmPath = "D:\\study\\CO\\p8\\upload\\calculator\\calculator.asm"
# asmPath = "D:\\study\\CO\\p8\\upload\\timer\\timer.asm"
# asmPath = "D:\\study\\CO\\p8\\upload\\uart_echo\\uart_echo.asm"

code = prjPath + "code.tmp"
handler = prjPath + "handler.tmp"
codeFile = prjPath + "code.coe"
os.system("java -jar " + marsPath + " db nc mc CompactDataAtZero a dump .text HexText " + code + " " + asmPath)
os.system("java -jar " + marsPath + " db nc mc CompactDataAtZero a dump 0x00004180-0x00005180 HexText " + handler + " " + asmPath)
with open(code, "r") as codeSrc, open(codeFile, "w") as codeDst:
    codeDst.write("memory_initialization_radix=16;\n")
    codeDst.write("memory_initialization_vector=\n")
    codeText = codeSrc.read().splitlines()
    for i in range(len(codeText)):
        codeDst.write(codeText[i] + ",\n")


    if os.path.exists(handler):
        with open(handler, "r") as handlerSrc:
            for i in range(len(codeText), 1120):
                codeDst.write("00000000,\n")
            handlerText = handlerSrc.read().splitlines()
            for i in range(len(handlerText)):
                codeDst.write(handlerText[i] + ",\n")
        os.remove(handler)

    codeDst.write("00000000;")
os.remove(code)
```

（三）汇编程序

1. 简易计算器

功能：支持 2 个 32 位操作数的 8 中运算，0-7 号按键开关分别对应与、或、异或、加、减、左移、逻辑右移、算术右移，当没有按键按下时，保持上次结果，当多个按键按下时，靠右的按键优先级最高。

代码思路：无限循环并检测操作数是否发生改变，当发生改变且操作有效时，分别进行不同的运算并输出结果到数码管。选择不同的运算采用了跳转表的结构（类似 switch-case 语句），根据最右侧按键的位置计算出跳转地址，然后过去进行计算，再跳转到同一的结束地址。

代码：

```
# 0->7 : & | ^ + - << >> >>>
# priority 0 > ... > 7


# li $t0, 1
# li $t1, 2
# li $t2, 0x8
# sw $t0, 0x7f54($0)
```

```
# sw $t1, 0x7f50($0)
# sb $t2, 0x7f58($0)

endless_loop:

    # get operands and operator
    lw $a0, 0x7f54($0)
    lw $a1, 0x7f50($0)
    lb $a2, 0x7f58($0)

    # jump table
    li $t0, 0
    while_1_begin:
    beqz $a2, while_1_end
    nop
        and $t1, $a2, 1
        beqz $t1, if_1_end
        nop
            la $ra, op_begin
            addu $ra, $ra, $t0
            jr $ra
            nop
        if_1_end:
        srl $a2, $a2, 1
        addiu $t0, $t0, 8
    j while_1_begin
    nop
    while_1_end:
    j endless_loop
    nop
```

```
op_begin:
    j op_end
    and $s0, $a0, $a1
    j op_end
    or $s0, $a0, $a1
    j op_end
    xor $s0, $a0, $a1
    j op_end
    addu $s0, $a0, $a1
    j op_end
    subu $s0, $a0, $a1
    j op_end
    sllv $s0, $a0, $a1
    j op_end
    srlv $s0, $a0, $a1
    j op_end
    srav $s0, $a0, $a1

op_end:
    sw $s0, 0x7f40($0)
    j endless_loop
    nop
```

2. 计时器

功能：从 0-3 组拨码开关读取初始值，随后开始倒计时，倒计时到 0 时，32 个 LED 灯亮一秒，随后重新开始计时。当复位或初始值改变时，也重新开始计时

代码思路：无限循环检测初始值是否发生改变，当改变时重新载入初始值进

行计时并重置内部计时器。微系统内部的计时器作用是计算一秒的时间，将 TC 的 preset 设置为 60000000（CPU 的主频），然后使用模式 0 开始计时。每当 TC 倒计时结束并产生中断信号时，表示过去了 1 秒，进入中断处理程序以后，将显示在数码管上的数值减 1，然后将 TC 的使能信号 ctrl[0]置 1，重新开始计时。

代码：

```
.text
    # li $t0, 10
    # sw $t0, 0x7f50($0)

    # set default values and init
    li $t0, 0x1401         # 1_0100_0000_0001
    mtc0 $t0, $12
    li $t0, 60000000      # frequency -> preset
    sw $t0, 0x7f04($0)
    li $a0, 9   # 1001
    li $a1, -1 # 111...111

    li $s0, 0 # cnt
    li $s1, 0 # pre-value

    loop:
        lw $t0, 0x7f50($0)
        beq $s1, $t0, if_1_end
        nop
            sw $0, 0x7f00($0) # stop count
            addu $s1, $0, $t0
            addu $s0, $0, $t0 # reload
            sw $0, 0x7f60($0) # LED off
            sw $a0, 0x7f00($0) # restart count
            if_2_end:
```

```
        if_1_end:
        sw $s0, 0x7f40($0) # digitalTube
    j loop
    nop

.ktext 0x4180
    sw $a0, 0x7f00($0)
    sw $0, 0x7f60($0) # LED off
    bnez $s0, if_3_end
    nop
        addu $s0, $0, $s1 # reload
        eret
    if_3_end:
    addiu $s0, $s0, -1
    bnez $s0, if_4_end
    nop
        sw $a1, 0x7f60($0) # LED on
        eret
    if_4_end:
    eret
```

3. UART 回显

功能：将串口输入直接原样输出，并且在数码管上显示接收的字符数，支持 9600,19200,38400，57600，115200 五种波特率模式，通过 0-3 号用户按键来控制，当 0 个或>1 个按键按下时，使用 9600 模特率，最右侧按键按下时，采用 19200 波特率，以此类推。

代码思路：当 MiniUART 模块成功读取一个完整有效数据时会产生中断信号，随后进入中断处理程序，先检测 LSR 寄存器，如果接受数据有效，就将该数据读出并存入缓冲区，如果发送保持器空，就从缓冲区取一个字符并发送。

由于接收数据的采样时间远远大于发送数据的时间，所以每当数据读取好时，发送保持器总是为空，因此暂时可以不设置缓冲区。当接收数据的时间与发送数据时间区别不大时，存在缓冲区无法及时输出的情况，因此本程序也可加入轮询功能（目前为不影响性能已注释掉），方法是将内部计时器设置为模式 1，每隔一定周期进行一次中断，检测缓冲区是否有未发送的数据。

代码：

```
.text
li $sp, 0x2fff   # buffer
li $t0, 0x1001
mtc0 $t0, $12
li $t0, 0xb
sb $t0, 0x7f44($0)
li $t0, 0x96
sb $t0, 0x7f41($0)
# li $t0, 1000           # check per 1000 cycles
# sw $t0, 0x7f04($0)
# li $t0, 11             # 1011
# sw $t0, 0x7f00($0)   # start timer in mode 1
li $s0, 0

loop:

    lb $a0, 0x7f58($0)
    andi $a0, $a0, 0xf
    beq $s0, $a0, loop
    nop

    addiu $s0, $a0, 0

    bne $a0, 1, if_3_else_1
```

```
nop
    li $t0, 389
    li $t1, 3124
    li $t2, 0x92
    li $t3, 0x01
    j if_3_end
    nop
if_3_else_1:

bne $a0, 2, if_3_else_2
nop
    li $t0, 194
    li $t1, 1561
    li $t2, 0x84
    li $t3, 0x03
    j if_3_end
    nop
if_3_else_2:

bne $a0, 4, if_3_else_3
nop
    li $t0, 129
    li $t1, 1040
    li $t2, 0x76
    li $t3, 0x05
    j if_3_end
    nop
if_3_else_3:

bne $a0, 8, if_3_else_4
```

```
    nop
        li $t0, 64
        li $t1, 519
        li $t2, 0x52
        li $t3, 0x11
        j if_3_end
        nop
    if_3_else_4:

    li $t0, 780
    li $t1, 6249
    li $t2, 0x96
    li $t3, 0x00

    if_3_end:
    sw $t0, 0x7f34($0)
    sw $t1, 0x7f38($0)
    sb $t2, 0x7f41($0)
    sb $t3, 0x7f42($0)

beq $0, $0, loop
nop

.ktext 0x4180
lw $t0, 0x7f30($0)
andi $t1, $t0, 1
andi $t2, $t0, 32
beqz $t1, if_1_end
nop
    lw $t0, 0x7f20($0)
```

```
        sb $t0, 0($sp)
        addiu $sp, $sp, -1
    if_1_end:
    beqz $t2, if_2_end
    nop
    bge $sp, 0x2fff, if_2_end
    nop
        addiu $sp, $sp, 1
        lb $t0, 0($sp)
        sw $t0, 0x7f20($0) # echo

        lb $t0, 0x7f43($0)
        addiu $t0, $t0, 1
        sb $t0, 0x7f43($0) # update digital tube
    if_2_end:

    eret
```