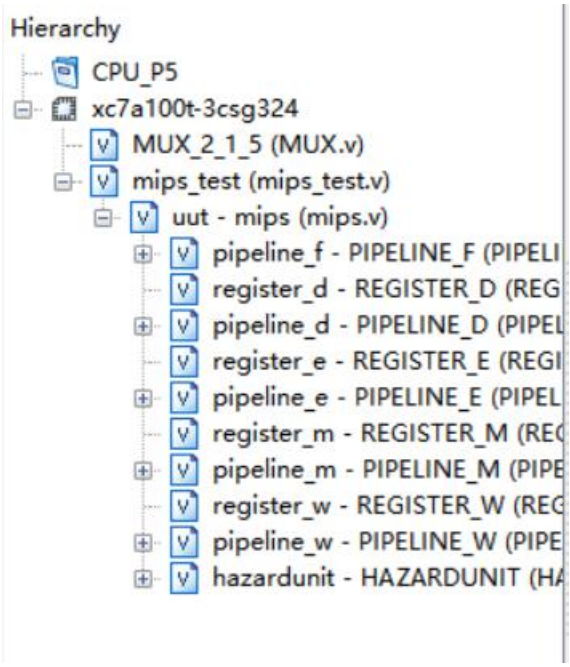
Project5 Verilog 完成流水线 CPU 开发

一、顶层设计




```
49  module mips(
50     input clk,
51     input reset
52      );
53      wire Stall, PCSel, RegWrite_W, RegWrite_D, RegWrite_E, RegWrite_M, ForwardRTM;
54      wire [1:0] ForwardRSD, ForwardRTD, ForwardRSE, ForwardRTE;
55      wire [4:0] RD_W,RS_E, RT_E, RD_E, WriteRd_E,RD_M;
56      wire [31:0] NPCOut, Instr_F, PCPlus4_F, Instr_D, PCPlus4_D, MUXRFWDOut, ALUOutput_M, EXTOut, RSV_D, RTV_D,
57                  Instr_E, PCPlus4_E, RSV_E, RTV_E, EXTOut_E, ALUOutput, WriteData_E, Instr_M, RTV_M,
58                  PCPlus4_M, MemOut, Instr_W, ReadData_W, PCPlus4_W, ALUOutput_W;
59  ////////////////////////////////////////////////////////////////////////////////
60  //F级部件
61  //IM 指令存储单元
62  //IFU 取指令单元
63  //MUXPC 选择下一条PC值的多选器
64
65  //B  cal_r  cal_i  J  load  store
66
67     PIPELINE_F pipeline_f(
68     .CLK(clk),                      //输入来自MIPS模块的时钟信号，控制IFU的时钟周期                    /**/
69     .Reset(reset),                  //输入来自MIPS模块的复位信号，控制IFU是否清空                      /**/
70     .Stall(Stall),                  //输入来自冲突单元的暂停信号，控制IFU是否暂停
71     .PCSel(PCSel),                  //输入来自D级部件（D级控制器）的下一条PC的选择信号，选择IFU下一个值
72     .NPCOut(NPCOut),                //输入来自D级部件（D级NPC）下一条PC可能的值，输入来自D级的跳转指令的IFU的下一个值
73
74     .Instr_F(Instr_F),              //输出F级的指令         To: D级寄存器
75     .PCPlus4_F(PCPlus4_F)           //输出F级的PC+4         To: D级寄存器
76     );
77
78  //F级部件产生指令和PC+4
79
80  ////////////////////////////////////////////////////////////////////////////////
```

```
//D级寄存器
   REGISTER_D register_d(
   .Instr_F(Instr_F),                  //输入来自F级部件（IM）的指令
   .PCPlus4_F(PCPlus4_F),              //输入来自F级部件（IFU）的PC+4
   .CLK(clk),                          //输入来自MIPS模块的时钟信号                                    /**/
   .Stall(Stall),                      //输入来自冲突单元的暂停信号
   .Reset(reset),                      //输入来自MIPS模块的复位信号                                    /**/

   .Instr_D(Instr_D),                  //输出D级的指令      To:  D级部件 E级寄存器 冲突单元
   .PCPlus4_D(PCPlus4_D)               //输出D级的PC+4      To:  D级部件 E级寄存器
   );
//D级寄存器产生D级的指令和PC+4

////////////////////////////////////////////////////////////////////////////////
//D级部件
//GRF 寄存器堆
//EXT 符号扩展器
//NPC PC计算选择器
//CMP 比较器
//CONTROLLER_D D级控制器
//MFRSD  转发指令在D级要读取的GRFRD1的值
//MFRTD  转发指令在D级要读取的GRFRD2的值

   PIPELINE_D pipeline_d(
   .CLK(clk),                          //输入来自MIPS模块的时钟信号                                     /**/
   .Reset(reset),                      //输入来自MIPS模块的复位信号                                     /**/
   .RegWrite_W(RegWrite_W),            //输入来自W级寄存器的写入信号，控制是否写入寄存器堆
   .RD_W(RD_W),                        //输入来自W级寄存器的写入地址，决定写入的地址
   .Instr_D(Instr_D),                  //输入来自D级寄存器的指令
   .MUXRFWDOut(MUXRFWDOut),            //输入来自W级部件（MUXRFWD）的写入数据，暨转发所需
   .PCPlus4_D(PCPlus4_D),              //输入来自D级寄存器的PC+4
   .ALUOutput_M(ALUOutput_M),          //输入来自M级寄存器的ALU输出，转发所需

   .ForwardRSD(ForwardRSD),            //输入来自冲突单元的RS转发控制信号
   .ForwardRTD(ForwardRTD),            //输入来自冲突单元的RT转发控制信号
   .PCPlus4_W(PCPlus4_W),              //输入来自W级寄存器的PC+4

   .EXTOut(EXTOut),                    //输出立即数扩展结果            To:  E级寄存器
   .NPCOut(NPCOut),                    //输出下一条可能的PC值          To:  F级部件
   .PCSel(PCSel),                      //输出PC选择信号                To:  F级部件
   .RegWrite_D(RegWrite_D),            //输出D级指令的写信号，在D级就提前产生写信号，之后在流水线中传输        To:  E级寄存器
   .RSV_D(RSV_D),                      //输出GRFRD1     To:  E级寄存器
   .RTV_D(RTV_D)                       //输出GRFRD2     To:  E级寄存器
   );
//D级部件产生 立即数扩展结果 跳转指令后PC的值 PC选择信号 D级指令写信号 寄存器两个读取的值

////////////////////////////////////////////////////////////////////////////////
//E级寄存器
   REGISTER_E register_e(
   .CLK(clk),                          //输入来自MIPS模块的时钟信号                                      /**/
   .Reset(reset),                      //输入来自MIPS模块的复位信号                                      /**/
   .FlushE(Stall),                     //输入来自冲突单元的暂停信号，实质上是清空信号，相当于插入NOP
   .Instr_D(Instr_D),                  //输入来自D级寄存器的指令
   .PCPlus4_D(PCPlus4_D),              //输入来自D级寄存器的PC+4
   .RSV_D(RSV_D),                      //输入来自D级部件（MFRSD）的GRFRD1
   .RTV_D(RTV_D),                      //输入来自D级部件（MFRTD）的GRFRD2
   .EXTOut_D(EXTOut),                  //输入来自D级部件（EXT）的扩展数
   .RegWrite_D(RegWrite_D),            //输入来自D级部件（CONTROLLER_D）的写信号
   .RS_D(Instr_D[25:21]),              //输入来自D级寄存器的指令的读寄存器地址1
   .RT_D(Instr_D[20:16]),              //输入来自D级寄存器的指令的读寄存器地址2
   .RD_D(Instr_D[15:11]),              //输入来自D级寄存器的指令的写寄存器地址

   .Instr_E(Instr_E),                  //输出E级的指令      To:  E级部件 M级寄存器 冲突单元
   .PCPlus4_E(PCPlus4_E),              //输出E级的PC+4      To:  E级部件 M级寄存器
   .RSV_E(RSV_E),                      //输出E级的GRFRD1    To:  E级部件

   .RTV_E(RTV_E),                      //输出E级的GRFRD2    To:  E级部件     /*store类型要用这个值存储，但是得经过E级部件选择*/
   .EXTOut_E(EXTOut_E),                //输出E级的扩展数    To:  E级部件
   .RegWrite_E(RegWrite_E),            //输出E级的写入信号  To:  M级寄存器 冲突单元
   .RS_E(RS_E),                        //输出E级指令的读寄存器地址1    To:  没啥用，不管了，实际上它的作用在E级部件中被Instr_E代替了
   .RT_E(RT_E),                        //输出E级指令的读寄存器地址2    To:  没啥用，不管了，实际上它的作用在E级部件中被Instr_E代替了
   .RD_E(RD_E)                         //输出E级指令的写寄存器地址     To:  没啥用，不管了，实际上它的作用在E级部件中被Instr_E代替了
   );
////////////////////////////////////////////////////////////////////////////////
//E级部件
//ALU 算术逻辑运算单元
//CONTROLLER_E E级控制器
//MUXALUOutput 选择ALU的输出，用来区分一般的对寄存器进行算术操作的指令和对31号寄存器进行操作的指令的输出
//MUXRDE 选择真正的写入寄存器的地址，应该将其输出连接至冲突单元，不然E级的写地址会错误
//MUXALUB 选择ALU的第二个操作数是寄存器数还是立即数
//MFRSE 选择ALU的第一个操作数的转发
//MFRTE 选择ALU的第二个操作数的转发

   PIPELINE_E pipeline_e(
   .Instr_E(Instr_E),                  //输入来自E级寄存器的指令
   .RTV_E(RTV_E),                      //输入来自E级寄存器的GRFRD2
   .RSV_E(RSV_E),                      //输入来自E级寄存器的GRFRD1
   .EXTOut_E(EXTOut_E),                //输入来自E级寄存器的扩展数
   .MUXRFWDOut(MUXRFWDOut),            //输入来自W级部件（MUXRFWD）的写入数据，转发所需
   .ALUOutput_M(ALUOutput_M),          //输入来自M级寄存器的ALU输出，转发所需
   .ForwardRSE(ForwardRSE),            //输入来自冲突单元的RSE控制信号
   .ForwardRTE(ForwardRTE),            //输入来自冲突单元的RTE控制信号
   .PCPlus4_E(PCPlus4_E),              //输入来自E级寄存器的PC+4，为对31号寄存器进行jal类操作的数准备

   .ALUOutput(ALUOutput),              //输出ALU的运算结果          To:  M级寄存器
   .WriteRd_E(WriteRd_E),              //输出真正的E级的写入地址    To:  M级寄存器              /*真正的经过的选择后的写入地址在E级确定*/
   .WriteData_E(WriteData_E)           //输出要写入内存的数据，应该来自经过转发MFRTE选择器选择的数据    To:  M级寄存器
   );
```

```
//E级部件产生 ALU的运算结果 E级的真正的写入地址 E级的真正的写入数据

////////////////////////////////////////////////////////////////////////////////
//M级寄存器
   REGISTER_M register_m(
   .CLK(clk),                       //输入来自MIPS模块的时钟信号                                  /**/
   .Reset(reset),                   //输入来自MIPS模块的复位信号                                  /**/
   .Instr_E(Instr_E),               //输入来自E级寄存器的指令
   .RTV_E(WriteData_E),             //输入来自E级部件（MFRTE）的内存写入数据          /**/     /*此处应为来自E级部件的输出，而不是来自E级寄存器的值*/
   .PCPlus4_E(PCPlus4_E),           //输入来自E级寄存器的PC+4
   .ALUOutput_E(ALUOutput),         //输入来自E级部件（MUXALUOutput）的ALU运算结果
   .RegWrite_E(RegWrite_E),         //输入来自E级寄存器的写信号
   .WriteRd_E(WriteRd_E),           //输入来自E级部件（MUXRDE）的真正的写寄存器地址

   .RegWrite_M(RegWrite_M),         //输出M级的写信号    To:  W级寄存器 冲突单元
   .Instr_M(Instr_M),               //输出M级的指令      To:  M级部件 W级寄存器 冲突单元
   .RTV_M(RTV_M),                   //输出M级的写入数据  To:  M级部件
   .PCPlus4_M(PCPlus4_M),           //输出M级的PC+4      To:  M级部件 W级寄存器
   .ALUOutput_M(ALUOutput_M),       //输出M级的ALU运算结果  To:  D级部件 E级部件 M级部件 W级寄存器
   .RD_M(RD_M)                      //输出M级的写寄存器地址  To:  W级寄存器 冲突单元
   );
////////////////////////////////////////////////////////////////////////////////
//M级部件
//DM 内存单元
//CONTROLLER_M M级控制器
//MFRTM 选择写入寄存器的值的转发
//如果添加LH LB SH SB之类的指令，在此加两个MUX即可

   PIPELINE_M pipeline_m(
   .CLK(clk),                       //输入来自MIPS模块的时钟信号              /**/
   .Reset(reset),                   //输入来自MIPS模块的复位信号              /**/
   .ALUOutput_M(ALUOutput_M),       //输入来自M级寄存器的的ALU输出

   .RTV_M(RTV_M),                   //输入来自M级寄存器的写入数据
   .Instr_M(Instr_M),               //输出来自M级寄存器的指令
   .ForwardRTM(ForwardRTM),         //输入来自冲突单元的RT转发控制信号
   .PCPlus4_M(PCPlus4_M),           //输入来自M级寄存器的PC+4
   .MUXRFWDOut(MUXRFWDOut),         //输入来自W级部件（MUXRFWD）写数据，转发所需

   .MemOut(MemOut)                  //输出内存读数据  To:  W级寄存器
   );
//M级部件产生 内存读数据

////////////////////////////////////////////////////////////////////////////////
//W级寄存器
   REGISTER_W register_w(
   .CLK(clk),                       //输入来自MIPS模块的时钟信号          /**/
   .Reset(reset),                   //输入来自MIPS模块的复位信号          /**/
   .Instr_M(Instr_M),               //输入来自M级寄存器的指令
   .PCPlus4_M(PCPlus4_M),           //输入来自M级寄存器的PC+4
   .ALUOutput_M(ALUOutput_M),       //输入来自M级寄存器的ALU运算结果
   .ReadData_M(MemOut),             //输入来自M级部件（DM）的内存读数据                  /**/        /*此处输入MemOut*/
   .RD_M(RD_M),                     //输入来自M级寄存器的写寄存器地址
   .RegWrite_M(RegWrite_M),         //输入来自M级寄存器的写寄存器数据

   .Instr_W(Instr_W),               //输出W级的指令         To:  W级部件 冲突单元
   .PCPlus4_W(PCPlus4_W),           //输出W级的PC+4         To:  W级部件
   .ALUOutput_W(ALUOutput_W),       //输出W级的ALU运算结果  To:  W级部件
   .ReadData_W(ReadData_W),         //输出W级的内存读数据   To:  W级部件
   .RD_W(RD_W),                     //输出W级的寄存器写地址  To:  D级部件 冲突单元
   .RegWrite_W(RegWrite_W)          //输出W级的寄存器写信号  To:  D级部件 冲突单元
   );
////////////////////////////////////////////////////////////////////////////////
//W级部件
//CONTROLLER W W级的控制器
```

```
//MUXRFWD 选择真正的写入数据
//GRF 理论上存在的写入寄存器堆

   PIPELINE_W pipeline_w(
   .Instr_W(Instr_W),               //输入来自W级寄存器的指令
   .ReadData_W(ReadData_W),         //输入来自W级寄存器的内存读数据
   .ALUOutput_W(ALUOutput_W),       //输入来自W级寄存器的ALU运算结果
   .PCPlus4_W(PCPlus4_W),           //输入来自W级寄存器的PC+4

   .MUXRFWDOut(MUXRFWDOut)          //输出寄存器要写入的数据       To:  D级部件 E级部件 M级部件      /*决定真正要写入的数据是在W级*/
   );
//W级部件产生 写入寄存器的值

////////////////////////////////////////////////////////////////////////////////
//冲突单元
   HAZARDUNIT hazardunit(
   .Instr_D(Instr_D),               //输入来自D级寄存器的指令
   .Instr_E(Instr_E),               //输入来自E级寄存器的指令
   .Instr_M(Instr_M),               //输入来自M级寄存器的指令
   .Instr_W(Instr_W),               //输入来自W级寄存器的指令
   .RegWrite_E(RegWrite_E),         //输入来自E级寄存器的写信号
   .RegWrite_M(RegWrite_M),         //输入来自M级寄存器的写信号                          /*之前连线出错，连到了E级的写信号*/
   .RegWrite_W(RegWrite_W),         //输入来自W级寄存器的写信号
   .A3_E(WriteRd_E),                //输入来自E级寄存器的写地址        /**/        /*由于我的设计像7.5.8那样，是在E级处理RegDst的，所以得传入MUX后的RD_E */
   .A3_M(RD_M),                     //输入来自M级寄存器的写地址        /**/
   .A3_W(RD_W),                     //输入来自W级寄存器的写地址        /**/

   .Stall(Stall),                   //输出暂停信号          To:  F级部件 D级寄存器 E级寄存器
   .ForwardRSD(ForwardRSD),         //输出RSD转发信号       To:  D级部件
   .ForwardRTD(ForwardRTD),         //输出RTD转发信号       To:  D级部件
   .ForwardRSE(ForwardRSE),         //输出RSE转发信号       To:  E级部件
   .ForwardRTE(ForwardRTE),         //输出RTE转发信号       To:  E级部件
   .ForwardRTM(ForwardRTM)          //输出RTM转发信号       To:  M级部件
   );
endmodule
```

二、模块设计

1、PIPELINE_F

（1）端口定义

信号	方向	描述
CLK	I	时钟信号
Reset	I	复位信号
Stall	I	输入来自冲突单元的暂停信号
PCSel	I	来自 D 级部件的 PC 选择信号
NPCOut	I	来自 D 级部件的 NPC 输出
Instr_F	O	输出 F 级的指令
PCPlus4_F	O	输出 F 级的 PC+4

（2）功能定义

序号	内部部件	功能描述
1	IM	存储和输出当前指令
2	IFU	存储和输出当前指令地址 PC
3	MUXPC	选择下一条指令地址

（3）内部部件

1）IM

```
module IM(
    input [31:0] InstrAddr,              //输入指令地址信号
    output [31:0] Instr_F                //输出指令
    );
   reg [31:0] IM[0:1023];

   initial begin
      $readmemh("code.txt",IM);
   end
   assign Instr_F = IM[InstrAddr[11:2]];
endmodule
//////////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////////////////
```

①端口定义

信号	方向	描述
InstrAddr	I	输入指令地址
Instr_F	O	输出当前指令

②功能定义

序号	功能名称	功能描述
1	取指令	根据 InstrAddr 指定的地址从 IM 中取出指令

2）IFU

```
//////////////////////////////////////////////////////////////////////////////////
module IFU(
    input CLK,                          //输入时钟信号
    input Reset,                        //输入复位信号
    input Stall,                        //输入暂停信号
    input [31:0] PCIn,                  //输入下一条PC的值
    output [31:0] PCPlus4_F,            //输出当前的PC+4
    output reg [31:0] PCOut = 32'h00003000 //输出当前的PC
    );

    assign PCPlus4_F = PCOut + 4;       //PC+4

    always @ (posedge CLK)begin
        if(Reset)                       //如果复位则置为0x3000
            PCOut <= 32'h00003000;
        else if(~Stall)                 //否则不复位的话，如果不暂停，就更新，暂停就不变
            PCOut <= PCIn;
    end
endmodule
//////////////////////////////////////////////////////////////////////////////////
```

①端口定义

信号	方向	描述
CLK	I	时钟信号
Reset	I	复位信号
Stall	I	来自冲突单元的暂停信号
PCIn	I	来自 MUXPC 的下一个 PC 值
PCOut	O	输出 PC 值
PCPlus4_F	O	输出当前 PC+4

②功能定义

序号	功能名称	功能描述
1	复位	当 Reset 信号有效时，PC 被置为 0x00003000

3）MUX_PC（MUX_2_1_32）

```
49    MUX_2_1_32 muxpc(
50    .A(PCPlus4_F),                    //输入PC+4
51    .B(NPCOut),                       //输入来自NPC的可能的值
52    .Sel(PCSel),                      //PC的选择信号
53
54    .C(PCIn)                          //PC的输入
55    );
```

```
21  module MUX_2_1_32(
22     input [31:0] A,
23     input [31:0] B,
24     input Sel,
25     output [31:0] C
26      );
27      assign C = Sel ? B : A ;
28  endmodule
```

①端口定义

信号	方向	描述
PCPlus4_F	I	输入来自 IFU 的 PC+4
NPCOut	I	输入来自 D 级部件的 B 或 J 类指令对应的 PC
PCSel	I	输入来自 D 级部件的 PC 选择信号
PCIn	O	输出下一条要更新的 PC 值

②功能定义

序号	功能名称	功能描述
1	选择下一条指令地址	当 PCSel 有效时选择 NPCOut,即跳转指令的地址

2、Regsiter_D

```
21  module REGISTER_D(
22     input [31:0] Instr_F,                    //来自F级的指令
23     input [31:0] PCPlus4_F,                  //来自F级的PC+4
24     input CLK,                               //时钟信号
25     input Stall,                             //暂停信号
26     input Reset,                             //复位信号
27     output reg [31:0]  Instr_D = 0,          //输出下一级D级的指令
28     output reg [31:0] PCPlus4_D = 0          //输出下一级D级的PC+4
29      );
30     always @(posedge CLK)begin
31        if(Reset)begin                        //如果是复位信号
32           Instr_D <= 0;                      //指令清零
33           PCPlus4_D <= 0;                    //PC+4清零
34        end
35        else if(~Stall)begin                  //如果不是复位信号
36           Instr_D <= Instr_F;                //更新指令
37           PCPlus4_D <= PCPlus4_F;            //更新PC+4
38        end
39     end
40
41  endmodule
42  //////////////////////////////////////////////////////////////////////////////////
```

（1）模块接口

信号	方向	描述

Instr_F	I	来自 F 级部件的指令
PCPlus4_F	I	来自 F 级部件的 PC+4
CLK	I	时钟信号
Stall	I	来自冲突单元的暂停信号
Reset	I	复位信号
Instr_D	O	输出 D 级寄存器的指令
PCPlus4_D	O	输出 D 级寄存器的 PC+4

3、PIPELINE_D

（1）端口定义

信号	方向	描述
CLK	I	时钟信号
Reset	I	复位信号
RegWrite_W	I	输入来自 W 级寄存器的写信号
RD_W	I	输入来自 W 级寄存器的写地

		址
Instr_D	I	输入来自 D 级寄存器的指令
MUXRFWDOut	I	输入来自 W 级部件的写数据，也是转发所需要的数据
PCPlus4_D	I	输入来自D级寄存器的PC+4
ALUOutput_M	I	输入来自M级寄存器的ALU运算结果,也是转发所需要的数据
ForwardRSD	I	输入来自冲突单元的 RSD 转发控制信号
ForwardRTD	I	输入来自冲突单元的 RTD 转发控制信号
PCPlus4_W	I	输入来自 W 级寄存器的 PC+4
EXTOut	O	输出 D 级部件产生的立即数扩展结果
NPCOut	O	输出 D 级部件的跳转指令产生的 PC 值
PCSel	O	输出D级部件产生的PC选择信号
RegWrite_D	O	输出 D 级部件的指令的写信号
RSV_D	O	输出寄存器堆读数据 1
RTV_D	O	输出寄存器堆读数据 2

（2）功能定义

序号	功能名称	功能描述

1	读数据	输出指令要读的 rs、rt 的寄存器对应的值
2	扩展立即数	扩展指令中的立即数作为 E 级的 ALU 操作数
3	计算跳转地址	计算跳转指令对应的跳转地址
4	控制跳转	判断当前指令是否为跳转指令
5	写数据	将指令要写的数据写入 rd 寄存器中

（3）内部部件

1）GRF

```
117 module GRF(
118     input CLK,
119     input Reset,
120     input RegWrite,
121     input [4:0] GRF_Rs,
122     input [4:0] GRF_Rt,
123     input [4:0] GRF_Rd,
124     input [31:0] GRF_WD,
125     input [31:0] PCPlus4_W,
126     output [31:0] GRF_RD1,
127     output [31:0] GRF_RD2
128      );
129     reg [31:0] GRF[0:31];
130 
131     integer i;
132     initial begin
133        for(i = 0;i < 32 ; i = i + 1)
134           GRF[i] <= 0;
135     end
136 
137     assign GRF_RD1 = GRF[GRF_Rs];
138     assign GRF_RD2 = GRF[GRF_Rt];
139 
140     always @ (posedge CLK)begin
141        if(Reset)begin
142           for(i = 0;i < 32 ; i = i + 1)
143              GRF[i] <= 0;
144        end
145        else if (RegWrite)begin
146        $display("%d@%h: $%d <= %h", $time, PCPlus4_W - 4, GRF_Rd,GRF_Rd == 0 ? 0 : GRF_WD);
147           GRF[GRF_Rd] <= GRF_Rd == 0 ? 0 : GRF_WD;
148        end
149     end

151 endmodule
152 //////////////////////////////////////////////////////////////////////////////////
```

①模块接口

信号	方向	描述
GRF_Rs	I	读寄存器地址 1
GRF_Rt	I	读寄存器地址 2
GRF_Rd	I	写寄存器地址
GRF_WD	I	写寄存器数据

CLK	I	时钟信号
Reset	I	复位信号
RegWrite	I	寄存器写信号
PCPlus4_W	I	写指令的地址
GRF_RD1	O	读寄存器数据 1
GRF_RD2	O	读寄存器数据 2

②功能定义

序号	功能名称	功能描述
1	读数据	读出 GRF_Rs 和 GRF_Rt 地址对应寄存器中的数据到 GRF_RD1、GRF_RD2
2	写数据	当 RegWrite 信号有效且时钟上升沿到来时，将 GRF_WD 写入 GRF_Rd 对应的寄存器中

2）EXT

```
153 module EXT(
154     input [15:0] Imm16,
155     input [3:0] EXTSel,
156     output [31:0] EXTOut
157     );
158 
159     wire [31:0] zero_ext, sign_ext, lui_ext;
160     assign zero_ext = {{16{1'b0}},{Imm16[15:0]}};
161     assign sign_ext = {{16{Imm16[15]}},{Imm16[15:0]}};
162     assign lui_ext = {{Imm16[15:0]},{16{1'b0}}};
163 
164     assign EXTOut = (EXTSel == 4'b0000)? zero_ext :
165                     (EXTSel == 4'b0001)? sign_ext :
166                     (EXTSel == 4'b0010)? lui_ext  :
167                                          0        ;
168 
169 endmodule
170 //////////////////////////////////////////////////////////////////////////////////
```

①模块接口

信号	方向	描述
Imm16	I	进行扩展的立即数

EXTSel	I	扩展选择信号
EXTOut	O	扩展结果

②功能定义

序号	功能名称	功能描述
1	零扩展	对立即数进行高位补 0 扩展
2	符号扩展	对立即数进行符号扩展至 32 位
3	加载到高位	将立即数加载到高位,低位补 0

3）CMP

```
198 module CMP(
199     input [31:0] CMPD1,
200     input [31:0] CMPD2,
201     output [1:0] CMPOut
202     );
203     assign CMPOut = (CMPD1 == CMPD2)? 2'b00 :
204                     ($signed(CMPD1) > $signed(CMPD2))?  2'b01 ://注意是有符号
205                     ($signed(CMPD1) < $signed(CMPD2))?  2'b10 :
206                                         2'b00 ;//!!!!!!!
207 endmodule
208 //////////////////////////////////////////////////////////////////////////////////
```

①模块接口

信号	方向	描述
CMPD1	I	比较的第一个数
CMPD2	I	比较的第二个数
CMPOut	O	比较的结果

②功能定义

序号	功能名称	功能描述
1	比较两个操作数	00：CMPD1 == CMPD2 01：CMPD1>CMPD2 10:CMPD1<CMPD2

4）NPC

```
171 module NPC(                                                    //JAL注意存的是PC+8？？？但是跳的话还是PC+4
172     input [31:0] Instr_D,
173     input [31:0] GRF_RD1,
174     input [31:0] PCPlus4_D,
175     input [1:0] NPCSel,
176     output [31:0] NPCOut
177     );
178
179     wire [31:0] B_index, J_index, B_offset_ext;
180     wire [25:0] J_offset;
181     wire [15:0] B_offset;
182
183     assign J_offset = Instr_D[25:0];
184     assign B_offset = Instr_D[15:0];
185
186     assign B_offset_ext = {{14{B_offset[15]}},{B_offset[15:0]},{2'b00}};
187
188     assign B_index = PCPlus4_D + B_offset_ext;
189     assign J_index = {{PCPlus4_D[31:28]},{J_offset[25:0]},{2'b00}};
190
191     assign NPCOut = (NPCSel == 2'b00)? B_index:
192                     (NPCSel == 2'b01)? J_index:
193                     (NPCSel == 2'b10)? GRF_RD1:
194                                        32'h00003000;//!!!!!!!!
195
196 endmodule
197 //////////////////////////////////////////////////////////////////////////////////
```

①模块接口

信号	方向	描述
GRF_RD1(MFRSDOut)	I	输入 Ra 中存储的地址
PCPlus4_D	I	输入来自 D 级寄存器的 PC+4
Instr_D	I	输入 D 级寄存器的指令，实质上是传输指令中的偏移量
NPCSel	I	输入 NPC 选择信号
NPCOut	O	输出 B J JR 指令对应的 PC

②功能定义

序号	功能名称	功能描述
1	输出跳转地址	NPCSel00：B 型指令地址 NPCSel01：J 型指令地址 NPCSel10：JR 指令地址

5）MFRSD（MUX_4_1_32）

```
95    MUX_4_1_32 MFRSD(
96    .A(GRF_RD1),
97    .B(MUXRFWDOut),
98    .C(ALUOutput_M),
99    .D(32'h00000000),
100   .Sel(ForwardRSD),
101
102   .E(MFRSDOut)
103   );
```

```
30  module MUX_4_1_32(
31     input [31:0] A,
32     input [31:0] B,
33     input [31:0] C,
34     input [31:0] D,
35     input [1:0] Sel,
36     output [31:0] E
37     );
38     assign E = (Sel == 2'b00)? A :
39                (Sel == 2'b01)? B :
40                (Sel == 2'b10)? C :
41                                D ;
42  endmodule
```

①模块接口

信号	方向	描述
GRF_RD1	I	输入 GRF 的读数据 1
MUXRFWDOut	I	输入来自 W 级部件的 GRF 要写入的数据
ALUOutput_M	I	输入来自 M 级寄存器的 ALU 运算结果
ForwardRSD	I	输入来自冲突单元的 RSD 转发信号
MFRSDOut	O	输出选择后的指令要用的 GRF 操作数 1

②功能定义

序号	功能名称	功能描述
1	输出转发选择后的 GRF 读数据 1	ForwardRSD： 00　GRF_RD1 01　MUXRFWDOut 10　ALUOutput_M

6）MFRTD（MUX_4_1_32）

```
105    MUX_4_1_32 MFRTD(
106    .A(GRF_RD2),
107    .B(MUXRFWDOut),
108    .C(ALUOutput_M),
109    .D(32'h00000000),
110    .Sel(ForwardRTD),
111
112    .E(MFRTDOut)
113    );
```

```
30  module MUX_4_1_32(
31     input [31:0] A,
32     input [31:0] B,
33     input [31:0] C,
34     input [31:0] D,
35     input [1:0] Sel,
36     output [31:0] E
37     );
38     assign E = (Sel == 2'b00)? A :
39                (Sel == 2'b01)? B :
40                (Sel == 2'b10)? C :
41                                D ;
42  endmodule
```

①模块接口

信号	方向	描述
GRF_RD2	I	输入 GRF 的读数据 2
MUXRFWDOut	I	输入来自 W 级部件的 GRF 要写入的数据
ALUOutput_M	I	输入来自 M 级寄存器的 ALU 运算结果
ForwardRTD	I	输入来自冲突单元的 RTD 转发信号
MFRTDOut	O	输出选择后的指令要用的 GRF 操作数 2

②功能定义

序号	功能名称	功能描述
1	输出转发选择后的 GRF 读数据 2	ForwardRTD： 00　GRF_RD2 01　MUXRFWDOut 10　ALUOutput_M

7）Controller_D

```
36  module CONTROLLER_D(
37     input [31:0] Instr_D,                              //输入来自D级寄存器的指令
38     input [1:0] CMPOut,                                //输入来自D级部件（CMP）的比较结果
39     output [1:0] NPCSel,                               //输出给D级部件NPC的NPC选择信号
40     output [3:0] EXTSel,                               //输出给D级部件EXT的EXT选择信号
41     output RegWrite_D,                                 //输出给E级寄存器的当前D级指令的写寄存器信号
42     output PCSel                                       //输出给F级部件的PC选择信号
43      );
44     wire [5:0] opcode = Instr_D[`OPCODE], func = Instr_D[`FUNC];
45
46     assign NPCSel[0] = (opcode == `J) || (opcode == `JAL);   //Bclass Jclass JR
47     assign NPCSel[1] = (opcode == `RCLASS && func == `JR);
48
49     assign EXTSel[0] = (opcode == `LW) || (opcode == `SW);   //应该是SW LW要SIGN_EXT而不是BEQ store load cal_i
50     assign EXTSel[1] = opcode == `LUI;
51     assign EXTSel[2] = 0;//还没用到
52     assign EXTSel[3] = 0;//还没用到
53
54     assign RegWrite_D = (opcode == `LW) || (opcode == `RCLASS && func == `ADDU) || (opcode == `RCLASS && func == `SUBU) || (opcode == `ORI) |
55                         (opcode == `LUI) || (opcode == `JAL);//cal_r cal_i load jal
56
57     assign PCSel = ((CMPOut == 2'b00) && (opcode == `BEQ)) || (opcode == `J) || (opcode == `JAL) || (opcode == `RCLASS && func == `JR);
58                    //Bclass Jclass JR jal
59  endmodule
```

①模块接口

信号	方向	描述
Instr_D	I	输入来自 D 级寄存器的指令
CMPOut	I	输入来自 D 级部件 CMP 的比较结果
NPCSel	O	输出给 NPC 的选择信号
EXTSel	O	输出给 EXT 的选择信号
RegWrite_D	O	输出给 E 级寄存器的 GRF 写信号
PCSel	O	输出给 F 级部件 MUXPC 的选择信号

②功能定义

序号	功能名称	功能描述
1	选择指令跳转类型	通过 NPCSel 选择指令跳转地址
2	选择是否跳转	通过 PCSel 选择是否跳转
3	选择立即数扩展类型	通过 EXTSel 选择扩展类型
4	判断指令是否进行写寄存器操作	通过 RegWrite_D 进行判断

4、Register_E

（1）模块接口

信号	方向	描述
CLK	I	输入时钟信号
Reset	I	输入复位信号
FlushE	I	来自冲突单元的清空信号，实质上是 Stall
Instr_D	I	输入来自 D 级寄存器的指令
PCPlus4_D	I	输入来自 D 级寄存器的 PC+4
RSV_D	I	输入来自 D 级部件的 RS 的数据
RTV_D	I	输入来自 D 级部件的 RT 的数据
EXTOut_D	I	输入来自 D 级部件的 EXT 结果
RegWrite_D	I	输入来自 D 级部件的 GRF 写信号
RS_D	I	输入来自 D 级寄存器的 RS 地址
RT_D	I	输入来自 D 级寄存器的 RT
RD_D	I	输入来自 D 级寄存器的 RD
Instr_E	O	输出 E 级的指令
PCPlus4E	O	输出 E 级的 PC+4
RSV_E	O	输出 E 级的 RS 对应的 GRF 读数据 1
RTV_E	O	输出 E 级的 RD 对应的 GRF 读数据 2
EXTOut_E	O	输出 E 级的 EXT 结果

RegWrite_E	O	输出 E 的 GRF 写信号
RS_E	O	输出 E 级的 RS 地址
RT_E	O	输出 E 的 RT 地址
RD_E	O	输出 E 的 RD 地址

5、PIPELINE_E

（1）模块接口

信号	方向	描述
Instr_E	I	输入来自 E 级寄存器的指令
RTV_E	I	输入来自 E 级寄存器的 GRF 读数据 2
RSV_E	I	输入来自 E 级寄存器的 GRF 读数据 1
EXTOut_E	I	输入来自 E 级寄存器的扩展数

MUXRFWDOut	I	输入来自W级部件的GRF写数据，是转发所需数据
ALUOutput_M	I	输入来自 M 级寄存器的 ALU 运算结果，是转发所需数据
ForwardRSE	I	输入来自冲突单元的 RSE 转发选择信号
ForwardRTE	I	输入来自冲突单元的 RTE 转发选择信号
PCPlus4_E	I	输入来自E级寄存器的PC+4，也是 jal 类指令要写入 31 寄存器的数
ALUOutput	O	输出 ALU 运算结果
WriteRd_E	O	输出寄存器写地址
WriteData_E	O	输出内存写数据

（2）功能定义

序号	功能名称	功能描述
1	ALU 运算	通过 ALU 进行指令要求的运算
2	选择 ALU 输出，ALU 输出可作为 load store 类指令的地址，可作为 cal 运算的结果，可作为 jal 指令的写入值	如果是 JAL 类对 31 号寄存器进行地址写入的指令，置为 PCPlus4_E+4，否则如果是正常的 ALU 运算，就置为 ALU 运算结果
3	选择写入寄存器地址	如果是三寄存器操作指令，置为 rd，如果是二寄存器操作指令，置为 rt，如果是 jal 类指令，置为 31

4	选择写入内存数据，仅仅针对 store 类指令	store 类指令要写入内存的数据可在 E 级部件通过转发获取

（3）内部部件

1）ALU

```
108 module ALU(
109     input [31:0] ALUOprand_A,
110     input [31:0] ALUOprand_B,
111     input [3:0] ALUSel,
112     output [31:0] ALUOutput
113      );
114     wire [31:0] addu_r, subu_r, ori_r;//不能填小写指令名
115 
116     assign addu_r = ALUOprand_A + ALUOprand_B;
117     assign subu_r = ALUOprand_A - ALUOprand_B;
118     assign ori_r =  ALUOprand_A | ALUOprand_B;
119 
120     assign ALUOutput = (ALUSel == 4'b0000)? addu_r :
121                        (ALUSel == 4'b0001)? subu_r :
122                        (ALUSel == 4'b0010)? ori_r  :
123                                             0  ;
124 endmodule
125 
```

①模块接口

信号	方向	描述
ALUOprand_A	I	ALU 操作数 A
ALUOprand_B	I	ALU 操作数 B
ALUSel	I	ALU 选择信号
ALUOutput	O	ALU 运算结果

②功能定义

序号	功能名称	功能描述
1	加运算	ALUOutput=ALUOprand_A+ ALUOprand_B
2	减运算	ALUOutput=ALUOprand_A- ALUOprand_B
3	或运算	ALUOutput=ALUOprand_A| ALUOprand_B

2）Controller_E

```
//////////////////////////////////////////////////////////////////////////////////
module CONTROLLER_E(
    input [31:0] Instr_E,
    output [3:0] ALUSel,
    output MUXALUBSel,
    output [1:0] RegDst,                    //处理JAL
    output ALUOutputSel                     //处理普通的运算指令和对RA进行操作的指令的ALU输出的问题
     );
    wire [5:0] opcode = Instr_E[`OPCODE], func = Instr_E[`FUNC];

    assign ALUSel[0] = (opcode == `RCLASS && func == `SUBU);    //cal_r cal_i load store
    assign ALUSel[1] = opcode == `ORI;
    assign ALUSel[2] = 0;//还没用到
    assign ALUSel[3] = 0;//还没用到

    assign MUXALUBSel = (opcode == `ORI) || (opcode == `LUI) || (opcode == `LW) || (opcode == `SW);// 1 的话就是立即数运算
    //cal_r cal_i load store
    assign RegDst = (opcode == `JAL)?                                                   2:
                    ((opcode == `RCLASS&& func == `ADDU) || (opcode == `RCLASS && func == `SUBU))? 1 :
                                                                                        0 ;
    //cal_r jal
    assign ALUOutputSel = (opcode == `JAL);//jal

endmodule
//////////////////////////////////////////////////////////////////////////////////
```

①模块接口

信号	方向	描述
Instr_E	I	输入来自 E 级寄存器的指令
ALUSel	O	输出 ALU 操作选择信号
MUXALUBSel	O	输出 ALUB 功能选择信号，选择 ALU 操作数 B 是寄存器数还是立即数
RegDst	O	输出当前指令的写寄存器选择信号
ALUOutputSel	O	输出 ALU 的结果选择信号，是 jal 型对应的 PC+8 还是正常的 ALU 运算结果

②功能定义

序号	功能名称	功能描述
1	选择 ALU 运算类型	通过 ALUSel 选择当前 E 级指令对应的 ALU 运算
2	选择 ALU 运算数	通过 MUXALUBSel 选择当前 E 级指令对应的 ALU 运算数是寄存器数还是立即数
3	选择写寄存器地址	通过 RegDst 选择当前 E 级指令写入寄存器的地址是 rt 还是 rd 还是 31

4	选择 ALU 输出	通过 ALUOutputSel 选择当前 E 级指令要用的 ALU 输出是 PC+8（JAL）还是正常的运算结果（正常运算指令）

3）MUXALUB

```
79    MUX_2_1_32 MUXALUB(
80    .A(MFRTEOut),
81    .B(EXTOut_E),
82    .Sel(MUXALUBSel),
83
84    .C(MUXALUBOut)
85    );
86
```

```
21  module MUX_2_1_32(
22     input [31:0] A,
23     input [31:0] B,
24     input Sel,
25     output [31:0] C
26      );
27      assign C = Sel ? B : A ;
28  endmodule
```

①模块接口

信号	方向	描述
MFRTEOut	I	输入 ALUB 转发多选器的选择结果，为寄存器操作数
EXTOut_E	I	输入来自 E 级寄存器的扩展数
MUXALUBSel	I	输入来自 E 级控制器的 MUXALUB 的选择信号
MUXALUBOut	O	输出 ALUB

②功能定义

序号	功能名称	功能描述
1	选择 ALU 的第二个操作数	当 MUXALUBSel 为 1 时选择立即数，为 0 时选择寄存器操作数

4）MFRSE

```
87    MUX_4_1_32 MFRSE(              //选择ALU的第一个操作数
88    .A(RSV_E),
89    .B(MUXRFWDOut),
90    .C(ALUOutput_M),
91    .D(32'h00000000),
92    .Sel(ForwardRSE),
93
94    .E(MFRSEOut)
95    );
```

```
30  module MUX_4_1_32(
31     input [31:0] A,
32     input [31:0] B,
33     input [31:0] C,
34     input [31:0] D,
35     input [1:0] Sel,
36     output [31:0] E
37     );
38     assign E = (Sel == 2'b00)? A :
39                (Sel == 2'b01)? B :
40                (Sel == 2'b10)? C :
41                                D ;
42  endmodule
```

①模块接口

信号	方向	描述
RSV_E	I	来自 E 级寄存器的的 GEF 读数据 1
MUXRFWDOut	I	来自 W 级部件的 GRF 写回数据
ALUOutput_M	I	来自 M 级寄存器的 ALU 的输出
ForwardRSE	I	来自 E 级控制器的 MFRSE 的选择信号
MFRSEOut	O	输出 MFRSE 选择后的数据

②功能定义

序号	功能名称	功能描述
1	选择 ALU 的第一个操作数	ForwardRSE: 00 RSV_E 01 MUXRFWDOut 10 ALUOutput_M

5）MFRTE

```
97    MUX_4_1_32 MFRTE(              //选择ALU的第二个操作数的寄存器数据
98    .A(RTV_E),
99    .B(MUXRFWDOut),
100   .C(ALUOutput_M),
101   .D(32'h00000000),
102   .Sel(ForwardRTE),
103
104   .E(MFRTEOut)
105   );
```

```
30  module MUX_4_1_32(
31     input [31:0] A,
32     input [31:0] B,
33     input [31:0] C,
34     input [31:0] D,
35     input [1:0] Sel,
36     output [31:0] E
37     );
38     assign E = (Sel == 2'b00)? A :
39                (Sel == 2'b01)? B :
40                (Sel == 2'b10)? C :
41                                D ;
42  endmodule
```

①模块接口

信号	方向	描述
RTV_E	I	来自 E 级寄存器的 GRF 的读数据 2
MUXRFWDOut	I	来自 W 级寄存器的 GRF 写回数据
ALUOutput_M	I	来自 M 级寄存器的 ALU 的输出
ForwardRTE	I	来自 E 级控制器的 MFRTE 的选择信号
MFRTEOut	O	输出 MFRTE 选择后的数据

②功能定义

序号	功能名称	功能描述
1	选择 ALU 的第二个操作数中的寄存器数	ForwardRTE: 00 RTV_E 01 MUXRFWDOut 10 ALUOutput_M

6）MUXRDE

```
69    MUX_4_1_5 MUXRDE(              //选择写入寄存器，处理JR JAL
70    .A(Instr_E[20:16]),
71    .B(Instr_E[15:11]),
72    .C(5'b11111),
73    .D(5'b00000),
74    .Sel(RegDst),
75
76    .E(WriteRd_E)
77    );
```

```
53  module MUX_4_1_5(
54     input [4:0] A,B,C,D,
55     input [1:0] Sel,
56     output [4:0] E
57     );
58     assign E = (Sel == 2'b00)? A :
59                (Sel == 2'b01)? B :
60                (Sel == 2'b10)? C :
61                                D ;
62  endmodule
```

①模块接口

信号	方向	描述
RT_E	I	来自 E 级寄存器的 GRF 写地址 1
RD_E	I	来自 E 级寄存器的 GRF 写地址 2
5’b11111	I	31 号寄存器
RegDst	I	来自 E 级控制器的写寄存器选择信号
WriteRd_E	O	输出 E 级当前指令要写的寄存器地址

②功能定义

序号	功能名称	功能描述
1	选择当前指令要写入的寄存器地址	RegDst: 00 RT_E 01 RD_E 10 31

7）MUXALUOutput

```
61    MUX_2_1_32 MUXALUOutput(      //选择E级部件的ALU输出结果，如果是jal等对31号寄存器进行写入PC+8的指令，将ALU输出置为PC+8
62    .A(ALUOutput_ALU),
63    .B(PCPlus4_E + 4),
64    .Sel(ALUOutputSel),
65
66    .C(ALUOutput)
67    );
```

```
21  module MUX_2_1_32(
22     input [31:0] A,
23     input [31:0] B,
24     input Sel,
25     output [31:0] C
26      );
27      assign C = Sel ? B : A ;
28  endmodule
```

①模块接口

信号	方向	描述
ALUOutput_ALU	I	输入来自 ALU 的运算结果
PCPlus4_E+4	I	输入来自 E 级寄存器的 PCPlus4_E+4，作为 JAL 指令对应要输入的值
ALUOutputSel	I	输入来自 E 级寄存器 E 级的 ALU 输出选择信号
ALUOutput	O	输出 E 级的 ALU 运算结果，其可能是正常指令的运算结果，也可能是对应 JAL 指令的 PC+8

②功能定义

序号	功能名称	功能描述
1	选择 E 级部件的 ALU 运算输出是正常 ALU 运算指令的运算结果还是 JAL 指令对应的 PC+8	ALUOutputSel 0 ALUOutput_ALU 1 PCPlus4_E+4

6、Register_M

（1）模块接口

信号	方向	描述
CLK	I	输入时钟信号
Reset	I	输入复位信号
Instr_E	I	输入来自 E 级寄存器的指令
RTV_E	I	输入来自 E 级部件的要写入内存的数据
PCPlus4_E	I	输入来自 E 级寄存器的的 PC+4
ALUOutput_E	I	输入来自 E 级部件的 ALU 运算结果
RegWrite_E	I	输入来自 E 级寄存器的 GRF 写信号
WriteRd_E	I	输入来自 E 级部件的当前指令对应的写寄存器地址
RegWrite_M	O	输出 M 级寄存器的 GRF 写信号
Instr_M	O	输出 M 级寄存器的指令
RTV_M	O	输出 M 级寄存器的要写入内存的数据
PCPlus4_M	O	输出 M 级寄存器的 PC+4
ALUOutput_M	O	输出 M 级寄存器的 ALU 运算结果
RD_M	O	输出 M 级寄存器的写寄存器地址

7、PIPELINE_M

（1）模块接口

信号	方向	描述
CLK	I	输入时钟信号
Reset	I	输入复位信号
ALUOutput_M	I	输入 M 级寄存器的 ALU 运算结果，作为存取内存的地址
RTV_M	I	输入来自 M 级寄存器的写入内存数据
Instr_M	I	输入来自 M 级寄存器的指令
ForwardRTM	I	输入来自冲突单元的 RTM 转发控制信号
PCPlus4_M	I	输入来自 M 级寄存器的 PC+4，用来 display
MUXRFWDOut	I	输入来自 W 级部件的 GRF 写入数据，是转发所需数据
MemOut	O	输出 M 级部件读出的内存数据

（2）功能定义

序号	功能名称	功能描述
1	读内存	根据输入的 ALUOutput_M 作为内存地址读出内存中的数据
2	写内存	当写内存信号有效且在时钟上升沿将数据写入对应的内存地址

3	复位	当 Reset 信号有效且时钟上升沿到来时将所有内存数据置零

（3）内部部件

1）DM

```
module DM(
   input CLK,
   input Reset,
   input MemWrite,
   input [31:0] MemAddr,
   input [31:0] MemWD,
   input [31:0] PCPlus4_M,
   output [31:0] MemOut
    );
   reg [31:0] DM[0:1023];

   assign MemOut = DM[MemAddr[11:2]];                    //连接输出

   integer i;
   initial                                          //初始化数据存储器
   begin
      for(i = 0;i < 1024 ;i = i + 1)
         DM[i] <= 0;
   end

   always @ (posedge CLK)begin                      //在时钟上升沿
      if(Reset)begin                                //复位信号全部置0
         for(i = 0;i < 1024 ;i = i + 1)
            DM[i] <= 0;
      end
      else if(MemWrite)begin                              //否则如果存数信号有效就存数
      $display("%d@%h: *%h <= %h", $time,PCPlus4_M - 4, MemAddr,MemWD);
         DM[MemAddr[11:2]] <= MemWD;
      end
   end

endmodule

```

①模块接口

信号	方向	描述
CLK	I	输入时钟信号
Reset	I	输入复位信号
MemWrite	I	输入来自 M 级控制器的内存写使能信号
MemAddr	I	输入来自 M 级寄存器额内存写地址，即 ALU 运算结果
MemWD	I	输入来自 MFRTM 的内存写数据
PCPlus4_M	I	输入来自 M 级寄存器的 PC+4，用来 display
MemOut	O	输出内存读数据

②功能定义

（4）

序号	功能名称	功能描述
1	读内存	根据输入的 ALUOutput_M 作为内存地址读出内存中的数据
2	写内存	当写内存信号有效且在时钟上升沿将数据写入对应的内存地址
3	复位	当 Reset 信号有效且时钟上升沿到来时将所有内存数据置零

2）Controller_M

```
84  //////////////////////////////////////////////////////////////////////////////////
85  module CONTROLLER_M(
86     input [31:0] Instr_M,
87     output MemWrite
88      );
89     wire [5:0] opcode = Instr_M[`OPCODE], func = Instr_M[`FUNC];
90
91     assign MemWrite = opcode == `SW; //store
92
93  endmodule
94  //////////////////////////////////////////////////////////////////////////////////
```

①模块接口

信号	方向	描述
Instr_M	I	来自 M 级的指令
MemWrite	O	输出内存写使能信号

②功能定义

序号	功能名称	功能描述

1	判断是否写内存	根据当前的 M 级指令输出 MemWrite 信号控制是否写内存

3）MFRTM

```
51    MUX_2_1_32 MFRTM(
52    .A(RTV_M),
53    .B(MUXRFWDOut),
54    .Sel(ForwardRTM),
55
56    .C(MFRTMOut)
57    );
```

```
21  module MUX_2_1_32(
22     input [31:0] A,
23     input [31:0] B,
24     input Sel,
25     output [31:0] C
26      );
27      assign C = Sel ? B : A ;
28  endmodule
```

①模块接口

信号	方向	描述
RTV_M	I	输入来自 M 级寄存器的 GRF 读数据 2
MUXRFWDOut	I	输入来自 W 级部件的 GRF 写回数据，是转发所需数据
ForwardRTM	I	输入来自冲突单元的 MFRTM 选择信号
MFRTMOut	O	输出 MFRTM 的转发选择结果，作为写入内存的数据

②功能定义

序号	功能名称	功能描述
1	选择写入内存的数据	ForwardRTM: 0 RTV_M 1 MUXRFWDOut

8、Register_W

信号	方向	描述
CLK	I	输入时钟信号
Reset	I	输入复位信号
Instr_M	I	输入来自 M 级寄存器的指令
PCPlus4_M	I	输入来自 M 级寄存器的 PC+4
ALUOutput_M	I	来自M级寄存器的ALU输出
ReadData_M	I	输入来自 M 级部件的内存读数据
RD_M	I	输入来自 M 级寄存器的 GRF 写地址
RegWrite_M	I	输入来自 M 级寄存器的 GRF 写信号
Instr_W	O	输出 W 级寄存器的指令
PCPlus4_W	O	输出 W 级寄存器的 PC+4
ALUOutput_W	O	输出W级寄存器的ALU输出
ReadData_W	O	输出 W 级寄存器的内存读数据
RD_W	O	输出W级寄存器的GRF写地址地址

RegWrite_W	O	输出W级寄存器的GRF写信号

9、PIPELINE_W

（1）模块接口

信号	方向	描述
Instr_W	I	输入来自 W 级寄存器的指令
ReadData_W	I	输入来自 W 级寄存器的内存读数据
ALUOutput_W	I	输入来自 W 级寄存器的 ALU 运算结果
PCPlus4_W	I	输入来自 W 级寄存器的 PC+4
MFRDWDOut	O	输出经过选择后的 GRF 要写入的数据

（2）功能定义

序号	功能名称	功能描述
1	选择写入 GRF 的数据	MemtoReg_W: 0 ALUOutput_W 1 ReadData_W

（3）内部部件

1）Controller_W

```
95  module CONTROLLER_W(
96     input [31:0] Instr_W,
97     output [1:0] MemtoReg_W
98      );
99     wire [5:0] opcode = Instr_W[`OPCODE], func = Instr_W[`FUNC];
100
101    assign MemtoReg_W = (opcode == `JAL)? 2:  //load jal
102                        (opcode == `LW)?  1:
103                                          0;
104
105 endmodule
106
```

①模块接口

信号	方向	描述
Instr_W	I	来自 W 级的指令
MemtoReg_W	O	MUXRFWD 的选择信号

②功能定义

序号	功能名称	功能描述
1	输出选择写入 GRF 的数据的控制信号	load 指令 1，其他指令 0

2）MUXRFWD

```
34    MUX_4_1_32 MUXRFWD(
35    .A(ALUOutput_W),
36    .B(ReadData_W),
37    .C(PCPlus4_W + 4),//JAL是写PC+8
38    .D(32'h00000000),
39    .Sel(MemtoReg_W),
40
41    .E(MUXRFWDOut)
42    );
```

```
30  module MUX_4_1_32(
31     input [31:0] A,
32     input [31:0] B,
33     input [31:0] C,
34     input [31:0] D,
35     input [1:0] Sel,
36     output [31:0] E
37     );
38     assign E = (Sel == 2'b00)? A :
39                (Sel == 2'b01)? B :
40                (Sel == 2'b10)? C :
41                                D ;
42  endmodule
```

①模块接口

信号	方向	描述
ReadData_W	I	来自 W 级的内存数据
ALUOutput_W	I	来自 W 级的 ALU 输出
MemtoReg_W	I	MUXRFWD 选择信号
MUXRFWDOut	O	输出寄存器写入数据

②功能定义

序号	功能名称	功能描述
1	选择写入 GRF 的数据	MemtoReg_W: 0 ALUOutput_W 1 ReadData_W

10、CONTROLLER

```
`define OPCODE 31:26
`define FUNC 5:0
`define RCLASS 6'b000000

`define LW 6'b100011
`define SW 6'b101011
`define ADDU 6'b100001
`define SUBU 6'b100011
`define ORI 6'b001101
`define LUI 6'b001111
`define BEQ 6'b000100
`define J 6'b000010
`define JAL 6'b000011
`define JR 6'b001000

module CONTROLLER_D(
    input [31:0] Instr_D,                                   //输入来自D级寄存器的指令
    input [1:0] CMPOut,                                     //输入来自D级部件（CMP）的比较结果
    output [1:0] NPCSel,                                    //输出给D级部件NPC的NPC选择信号
    output [3:0] EXTSel,                                    //输出给D级部件EXT的EXT选择信号
    output RegWrite_D,                                      //输出给E级寄存器的当前D级指令的写寄存器信号
    output PCSel                                            //输出给F级部件的PC选择信号
     );
    wire [5:0] opcode = Instr_D[`OPCODE], func = Instr_D[`FUNC];

    assign NPCSel[0] = (opcode == `J) || (opcode == `JAL);   //Bclass Jclass JR
    assign NPCSel[1] = (opcode == `RCLASS && func == `JR);

    assign EXTSel[0] = (opcode == `LW) || (opcode == `SW);   //应该是SW LW要SIGN_EXT而不是BEQ store load cal_i
    assign EXTSel[1] = opcode == `LUI;
    assign EXTSel[2] = 0;//还没用到
    assign EXTSel[3] = 0;//还没用到

    assign RegWrite_D = (opcode == `LW) || (opcode == `RCLASS && func == `ADDU) || (opcode == `RCLASS && func == `SUBU) || (opcode == `ORI)
                        (opcode == `LUI) || (opcode == `JAL);//cal_r cal_i load jal

    assign PCSel = ((CMPOut == 2'b00) && (opcode == `BEQ)) || (opcode == `J) || (opcode == `JAL) || (opcode == `RCLASS && func == `JR);
                   //Bclass Jclass JR jal
endmodule
//////////////////////////////////////////////////////////////////////////////////
module CONTROLLER_E(
    input [31:0] Instr_E,
    output [3:0] ALUSel,
    output MUXALUBSel,
    output [1:0] RegDst,                        //处理JAL
    output ALUOutputSel                         //处理普通的运算指令和对RA进行操作的指令的ALU输出的问题
     );
    wire [5:0] opcode = Instr_E[`OPCODE], func = Instr_E[`FUNC];

    assign ALUSel[0] = (opcode == `RCLASS && func == `SUBU);    //cal_r cal_i load store
    assign ALUSel[1] = opcode == `ORI;
    assign ALUSel[2] = 0;//还没用到
    assign ALUSel[3] = 0;//还没用到

    assign MUXALUBSel = (opcode == `ORI) || (opcode == `LUI) || (opcode == `LW) || (opcode == `SW);// 1 的话就是立即数运算
    //cal_r cal_i load store
    assign RegDst = (opcode == `JAL)?                                                   2:
                    ((opcode == `RCLASS&& func == `ADDU) || (opcode == `RCLASS && func == `SUBU))? 1 :
                                                                                        0 ;
    //cal_r jal
    assign ALUOutputSel = (opcode == `JAL);//jal

endmodule
//////////////////////////////////////////////////////////////////////////////////
module CONTROLLER_M(
    input [31:0] Instr_M,

    output MemWrite
     );
    wire [5:0] opcode = Instr_M[`OPCODE], func = Instr_M[`FUNC];

    assign MemWrite = opcode == `SW; //store

endmodule
//////////////////////////////////////////////////////////////////////////////////
module CONTROLLER_W(
    input [31:0] Instr_W,
    output [1:0] MemtoReg_W
     );
    wire [5:0] opcode = Instr_W[`OPCODE], func = Instr_W[`FUNC];

    assign MemtoReg_W = (opcode == `JAL)? 2:  //load jal
                        (opcode == `LW)?  1:
                                          0;

endmodule
```

	op code	func	NPC Sel	EXT Sel	PC Sel	Reg Write	Reg Dst	ALU Output Sel	ALU Sel	MUX ALU B Sel	Mem Write	Mem toReg
addu	000000	100001	X	X	0	1	1	0	0	0	0	0
subu	000000	100011	X	X	0	1	1	0	1	0	0	0

ori	**001101**	**N/A**	**X**	**0**	**0**	**1**	**0**	**0**	**2**	**1**	**0**	**0**
lw	**100011**	**N/A**	**X**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**
sw	**101011**	**N/A**	**X**	**1**	**0**	**0**	**X**	**X**	**0**	**1**	**1**	**0**
beq	**000100**	**N/A**	**0**	**X**	**1**	**0**	**X**	**X**	**0**	**0**	**0**	**0**
j	**000010**	**N/A**	**1**	**X**	**1**	**0**	**X**	**X**	**0**	**X**	**0**	**0**
jr	**000000**	**001000**	**2**	**X**	**1**	**0**	**X**	**X**	**0**	**X**	**0**	**0**
jal	**000011**	**N/A**	**1**	**X**	**1**	**1**	**2**	**1**	**0**	**X**	**0**	**0**
lui	**001111**	**N/A**	**X**	**2**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**

（1）Controller_D

①模块接口

信号	方向	描述
Instr_D	I	输入来自 D 级寄存器的指令
CMPOut	I	输入来自 D 级部件 CMP 的比较结果
NPCSel	O	输出给 NPC 的选择信号
EXTSel	O	输出给 EXT 的选择信号
RegWrite_D	O	输出给 E 级寄存器的 GRF 写信号
PCSel	O	输出给 F 级部件 MUXPC 的选择信号

②功能定义

序号	功能名称	功能描述
1	选择指令跳转类型	通过 NPCSel 选择指令跳转地址
2	选择是否跳转	通过 PCSel 选择是否跳转
3	选择立即数扩展类型	通过 EXTSel 选择扩展类型
4	判断指令是否进行写寄存器操作	通过 RegWrite_D 进行判断

（2）Controller_E

①模块接口

信号	方向	描述
Instr_E	I	输入来自 E 级寄存器的指令
ALUSel	O	输出 ALU 操作选择信号
MUXALUBSel	O	输出 ALUB 功能选择信号，选择 ALU 操作数 B 是寄存器数还是立即数
RegDst	O	输出当前指令的写寄存器选择信号
ALUOutputSel	O	输出 ALU 的结果选择信号，是 jal 型对应的 PC+8 还是正常的 ALU 运算结果

②功能定义

序号	功能名称	功能描述
1	选择 ALU 运算类型	通过 ALUSel 选择当前 E 级指令对应的 ALU 运算
2	选择 ALU 运算数	通过 MUXALUBSel 选择当前 E 级指令对应的 ALU 运算

		数是寄存器数还是立即数
3	选择写寄存器地址	通过 RegDst 选择当前 E 级指令写入寄存器的地址是 rt 还是 rd 还是 31
4	选择 ALU 输出	通过 ALUOutputSel 选择当前 E 级指令要用的 ALU 输出是 PC+8（JAL）还是正常的运算结果（正常运算指令）

（3）Controller_M

①模块接口

信号	方向	描述
Instr_M	I	来自 M 级的指令
MemWrite	O	输出内存写使能信号

②功能定义

序号	功能名称	功能描述
1	判断是否写内存	根据当前的 M 级指令输出 MemWrite 信号控制是否写内存

（4）Controller_W

①模块接口

信号	方向	描述
Instr_W	I	来自 W 级的指令
MemtoReg_W	O	MUXRFWD 的选择信号

②功能定义

序号	功能名称	功能描述
1	输出选择写入 GRF 的数据的控制信号	load 指令 1，其他指令 0

11、HAZARDUNIT

```
`define OPCODE 31:26
`define FUNC 5:0
`define RS 25:21
`define RT 20:16
`define RD 15:11
`define RCLASS 6'b000000
`define LW 6'b100011
`define SW 6'b101011
`define ADDU 6'b100001
`define SUBU 6'b100011
`define ORI 6'b001101
`define LUI 6'b001111
`define BEQ 6'b000100
`define J 6'b000010
`define JAL 6'b000011
`define JR 6'b001000
module HAZARDUNIT(
   input [31:0] Instr_D, Instr_E, Instr_M, Instr_W,//传入指令
   input RegWrite_E, RegWrite_M, RegWrite_W,//传入写信号
   input [4:0] A3_E, A3_M, A3_W,//传入A3_M方便，不然在这里判断很麻烦
   output Stall,
   output [1:0] ForwardRSD, ForwardRTD, ForwardRSE, ForwardRTE,
   output ForwardRTM
    );

    wire Tuse_RS0, Tuse_RS1, Tuse_RT0, Tuse_RT1, Tuse_RT2;//这个用来表示指令，这样容易判断寄存器使用情况，如果仅仅是单纯的译码出时间，不行
    wire Stall_RS0_E1, Stall_RS0_E2, Stall_RS0_M1, Stall_RS1_E2, Stall_RS,
         Stall_RT0_E1, Stall_RT0_E2, Stall_RT0_M1, Stall_RT1_E2, Stall_RT;
    wire [1:0] Tnew_E, Tnew_M;
   //产生Tnew
    DECODER_Tnew_E decoder_tnew_e(
    .Instr(Instr_E),
    .Tnew(Tnew_E)
    );

    DECODER_Tnew_M decoder_tnew_m(
    .Instr(Instr_M),
    .Tnew(Tnew_M)
    );
   //产生Tuse
   //Tuse和Tnew更多的是用来判断暂停的

   //在D级还有0个周期需要使用RS  beq jr
   assign Tuse_RS0 = (Instr_D[`OPCODE] == `BEQ) || ((Instr_D[`OPCODE] == `RCLASS) && (Instr_D[`FUNC] == `JR));

   //在D级还有1个周期需要使用RS  cal_r cal_i load store
   assign Tuse_RS1 = ((Instr_D[`OPCODE] == `RCLASS) && (Instr_D[`FUNC] == `ADDU)) ||
                     ((Instr_D[`OPCODE] == `RCLASS) && (Instr_D[`FUNC] == `SUBU)) ||
                     (Instr_D[`OPCODE] == `LW) || (Instr_D[`OPCODE] == `SW) || (Instr_D[`OPCODE] == `ORI) || (Instr_D[`OPCODE] == `LUI);

   //在D级还有0个周期需要使用RT   beq
   assign Tuse_RT0 = (Instr_D[`OPCODE] == `BEQ);

   //在D级还有1个周期需要使用RT   cal_r
   assign Tuse_RT1 = ((Instr_D[`OPCODE] == `RCLASS) && (Instr_D[`FUNC] == `ADDU)) ||
                     ((Instr_D[`OPCODE] == `RCLASS) && (Instr_D[`FUNC] == `SUBU));

   //在D级还有2个周期需要使用RT   store
   assign Tuse_RT2 = (Instr_D[`OPCODE] == `SW);

   //产生Stall

   //如果在D级还有0个周期要使用RS，而E级指令仍需1个周期才能产生所要写入的数据
   assign Stall_RS0_E1 = Tuse_RS0 && (Tnew_E == 2'b01) && (Instr_D[`RS] == A3_E) && RegWrite_E;

   //如果在D级还有0个周期要使用RS，而E级指令仍需2个周期才能产生所要写入的数据
```

```
    assign Stall_RS0_E2 = Tuse_RS0 && (Tnew_E == 2'b10) && (Instr_D[`RS] == A3_E) && RegWrite_E;

    //如果在D级还有0个周期要使用RS，而M级指令仍需1个周期才能产生所要写入的数据
    assign Stall_RS0_M1 = Tuse_RS0 && (Tnew_M == 2'b01) && (Instr_D[`RS] == A3_M) && RegWrite_M;

    //如果在D级还有1个周期要使用RS，而E级指令仍需2个周期才能产生所要写入的数据
    assign Stall_RS1_E2 = Tuse_RS1 && (Tnew_E == 2'b10) && (Instr_D[`RS] == A3_E) && RegWrite_E;

    assign Stall_RS = Stall_RS0_E1 || Stall_RS0_E2 || Stall_RS0_M1 || Stall_RS1_E2;


    //如果在D级还有0个周期要使用RT，而E级指令仍需1个周期才能产生所要写入的数据
    assign Stall_RT0_E1 = Tuse_RT0 && (Tnew_E == 2'b01) && (Instr_D[`RT] == A3_E) && RegWrite_E;

    //如果在D级还有0个周期要使用RT，而E级指令仍需2个周期才能产生所要写入的数据
    assign Stall_RT0_E2 = Tuse_RT0 && (Tnew_E == 2'b10) && (Instr_D[`RT] == A3_E) && RegWrite_E;

    //如果在D级还有0个周期要使用RT，而M级指令仍需1个周期才能产生所要写入的数据
    assign Stall_RT0_M1 = Tuse_RT0 && (Tnew_M == 2'b01) && (Instr_D[`RT] == A3_M) && RegWrite_M;

    //如果在D级还有1个周期要使用RT，而E级指令仍需2个周期才能产生所要写入的数据
    assign Stall_RT1_E2 = Tuse_RT1 && (Tnew_E == 2'b10) && (Instr_D[`RT] == A3_E) && RegWrite_E;

    assign Stall_RT = Stall_RT0_E1 || Stall_RT0_E2 || Stall_RT0_M1 || Stall_RT1_E2;

    assign Stall = Stall_RS || Stall_RT;

    //产生Forward
    //跳转指令之后规定不能再接跳转指令
    //由于对一条指令如SW的RT可能存在多次转发，所以的确得保证优先级问题，最新更新最优先
    //指令在越后面要用到寄存器的值，所用的值经过的转发更新就越多

    //在D级的指令（beq）需要用到RS，但是如果别的指令在这里需要的话，会提前转发
```

```
    assign ForwardRSD = ((Instr_D[`RS] == A3_M) && (Tnew_M == 0) && (A3_M != 0) && RegWrite_M)? 2 :
                                    ((Instr_D[`RS] == A3_W) && (A3_W != 0) && RegWrite_W)? 1 :
                                                                                          0 ;

    //在D级的指令（beq）需要用到RT
    assign ForwardRTD = ((Instr_D[`RT] == A3_M) && (Tnew_M == 0) && (A3_M != 0) && RegWrite_M)? 2 :
                                    ((Instr_D[`RT] == A3_W) && (A3_W != 0) && RegWrite_W)? 1 :
                                                                                          0 ;

    //在E级的指令（cal_r cal_i load store）需要用到RS
    assign ForwardRSE = ((Instr_E[`RS] == A3_M) && (Tnew_M == 0) && (A3_M != 0) && RegWrite_M)? 2 :
                                    ((Instr_E[`RS] == A3_W) && (A3_W != 0) && RegWrite_W)? 1 :
                                                                                          0 ;

    //在E级的指令（cal_r）需要用到RT
    assign ForwardRTE = ((Instr_E[`RT] == A3_M) && (Tnew_M == 0) && (A3_M != 0) && RegWrite_M)? 2 :
                                    ((Instr_E[`RT] == A3_W) && (A3_W != 0) && RegWrite_W)? 1 :
                                                                                          0 ;

    //在M级的指令（sw）需要用到RT
    assign ForwardRTM = ((Instr_M[`RT] == A3_M) && (Tnew_M == 0) && (A3_M != 0) && RegWrite_M)? 2 :
                                    ((Instr_M[`RT] == A3_W) && (A3_W != 0) && RegWrite_W)? 1 :
                                                                                          0 ;

endmodule
```

（1）模块接口

信号	方向	描述
Instr_D	I	输入来自 D 级寄存器的指令
Instr_E	I	输入来自 E 级寄存器的指令
Instr_M	I	输入来自 M 级寄存器的指令
Instr_W	I	输入来自 W 级寄存器的指令
RegWrite_E	I	输入来自 E 级的写使能信号
RegWrite_M	I	输入来自 M 级的写使能信号
RegWrite_W	I	输入来自 W 级的写使能信号
A3_E	I	输入来自 E 级的写地址
A3_M	I	输入来自 M 级的写地址
A3_W	I	输入来自 W 级的写地址
Stall	O	输出暂停信号
ForwardRSD	O	输出 RSD 转发控制信号
ForwardRTD	O	输出 RTD 转发控制信号

ForwardRSE	O	输出 RSE 转发控制信号
ForwardRTE	O	输出 RTE 转发控制信号
ForwardRTM	O	输出 RTM 转发控制信号

（2）功能定义

序号	功能名称	功能描述
1	判断暂停	当前 D 级指令所需使用的寄存器的时间周期大于 E、M、W 级相同的寄存器写入指令的产生写入数据时间周期时，需要暂停。有两类寄存器和八种情况 RS0_E1　RS0_E2 RS0_M1　RS1_E2 RT0_E1　RT0_E2 RT0_M1　RT1_E2
2	判断转发	当前 D 级指令所需使用的寄存器的时间周期小于或等于 E、M、W 级相同的寄存器写入指令的产生写入数据时间周期时，可以转发。有两类寄存器和五种情况 FRSD　FRTD FRSE　FRTE

		FRTM

	cal_r		cal_i		store	load					
	addu	subu	ori	lui	sw	lw	beq	jr	jal	nop	j
Tuse_RS0	0	0	0	0	0	0	1	1	X	X	X
Tuse_RS1	1	1	1	1	1	1	0	0	X	X	X
Tuse_RT0	0	0	X	X	0	X	1	X	X	X	X
Tuse_RT1	1	1	X	X	0	X	0	X	X	X	X
Tuse_RT2	0	0	X	X	1	X	0	X	X	X	X
Tnew	1	1	1	1	3	2	3	3	3	3	3

注：Tnew=3 代表不产生写入寄存器数据

IF/ID当前指令			ID/EX(Tnew)/E					EX/MEM(Tnew)/M					MEM/WB(Tnew)/W				
指令类型	源寄存器	Tuse	cal_r 1/rd	cal_i 1/rt	load 2/rt	jal 0/ra	jalr 0/rd	cal_r 0/rd	cal_i 0/rt	load 1/rt	jal 0/ra	jalr 0/rd	cal_r 0/rd	cal_i 0/rt	load 0/rt	jal 0/ra	jalr 0/rd
jr	rs	0	stall	stall	stall					stall							
beq	rs/rt	0	stall	stall	stall					stall							
cal_r	rs/rt	1			stall												
cal_i	rs	1			stall												
load	rs	1			stall												
store	rs	1			stall												
store	rt	2															

（3）内部部件 DECODER

```
`define OPCODE 31:26
`define FUNC 5:0
`define RCLASS 6'b000000

`define LW 6'b100011
`define SW 6'b101011
`define ADDU 6'b100001
`define SUBU 6'b100011
`define ORI 6'b001101
`define LUI 6'b001111
`define BEQ 6'b000100
`define J 6'b000010
`define JAL 6'b000011
`define JR 6'b001000
module DECODER_Tnew_E(//这里产生的Tnew是针对E而言的
   input [31:0] Instr,
   output reg [1:0] Tnew = 2'b11
   );
    always @(*)begin
      case(Instr[`OPCODE])
         `RCLASS:
            case(Instr[`FUNC])
               `ADDU:begin
                  Tnew = 2'b01;
               end
               `SUBU:begin
                  Tnew = 2'b01;
               end
               default:begin//////////////其他指令默认不产生吧
                  Tnew = 2'b11;
               end
            endcase
         `ORI:begin

            Tnew = 2'b01;
         end
         `LUI:begin
            Tnew = 2'b01;
         end
         `LW:begin
            Tnew = 2'b10;
         end
         `JAL:begin
                  Tnew = 2'b00;
         end
      default:begin
         Tnew = 2'b11;
      end
      endcase
   end
endmodule
//////////////////////////////////////////////////////////////
module DECODER_Tnew_M(//这里产生的Tnew是针对M而言的
   input [31:0] Instr,
   output reg [1:0] Tnew = 2'b11
   );
    always @(*)begin
      case(Instr[`OPCODE])
         `RCLASS:
            case(Instr[`FUNC])
               `ADDU:begin
                  Tnew = 2'b00;
               end
               `SUBU:begin

                  Tnew = 2'b00;
               end
               default:begin//////////////
                  Tnew = 2'b00;
               end
            endcase
         `ORI:begin
            Tnew = 2'b00;
         end
         `LUI:begin
            Tnew = 2'b00;
         end
         `LW:begin
            Tnew = 2'b01;
         end
         `JAL:begin
                  Tnew = 2'b00;
         end
      default:begin
         Tnew = 2'b11;
      end
      endcase
   end
endmodule

```

12、MUX

```
21  module MUX_2_1_32(
22     input [31:0] A,
23     input [31:0] B,
24     input Sel,
25     output [31:0] C
26      );
27      assign C = Sel ? B : A ;
28  endmodule
29
30  module MUX_4_1_32(
31     input [31:0] A,
32     input [31:0] B,
33     input [31:0] C,
34     input [31:0] D,
35     input [1:0] Sel,
36     output [31:0] E
37     );
38     assign E = (Sel == 2'b00)? A :
39                (Sel == 2'b01)? B :
40                (Sel == 2'b10)? C :
41                                D ;
42  endmodule
43
44  module MUX_2_1_5(
45     input [4:0] A,
46     input [4:0] B,
47     input Sel,
48     output [4:0] C
49     );
50     assign C = Sel ? B : A ;
51  endmodule
52
```

```
53  module MUX_4_1_5(
54     input [4:0] A,B,C,D,
55     input [1:0] Sel,
56     output [4:0] E
57     );
58     assign E = (Sel == 2'b00)? A :
59                (Sel == 2'b01)? B :
60                (Sel == 2'b10)? C :
61                                D ;
62  endmodule
```

三、测试程序

```
ori $0,$0,0          #测试 ori
ori $1,$0,1
ori $2,$0,2
ori $3,$0,3
ori $4,$0,4
ori $5,$0,5
ori $6,$0,6
ori $7,$0,7
ori $8,$0,8
ori $9,$0,9
lui $10,10           #测试 lui
ori $11,$0,11
ori $12,$0,12
ori $13,$0,13
ori $14,$0,14
ori $15,$0,15
ori $16,$0,16
ori $17,$0,17
ori $18,$0,18
ori $19,$0,19
ori $20,$0,20
ori $21,$0,21
ori $22,$0,22
```

```
ori $23,$0,23
ori $24,$0,24
ori $25,$0,25
ori $26,$0,26
ori $27,$0,27
lui $28,28
ori $29,$0,29
ori $30,$0,30
ori $31,$0,31
addu $t1,$t1,$t2     #测试冲突
subu $t1,$t1,$t2
ori $t1,$0,8
sw $t1,0($0)         #测试冲突
lw $t1,0($0)
ori $t1,$0,12        #测试冲突
sw $t1,-4($t1)       #测试冲突
sw $t1,4($t1)
ori $t1,$0,4
loop1:
lw $t1,4($t1)        #测试冲突
lw $t1,0($t1)
beq $t1,$0,loop1
addu $t1,$t1,$8
beq $t1,$s0,loop2
nop
a3:
j a1
nop
nop
nop
nop
nop
a1:
jal a2
addu $t1,$ra,$t1
sw $ra,0($8)
addu $8,$8,$4
a2:
subu $t1,$ra,$t1
jr $ra
nop
loop2:
addu $t1,$t1,$t2
subu $t3,$t3,$t4
```

```
ori $t1,$t1,0x00001234
jal a3
addu $t1,$ra,$t1
```

期望输出：

```
 90@00003000: $ 0 <= 00000000
110@00003004: $ 1 <= 00000001
130@00003008: $ 2 <= 00000002
150@0000300c: $ 3 <= 00000003
170@00003010: $ 4 <= 00000004
190@00003014: $ 5 <= 00000005
210@00003018: $ 6 <= 00000006
230@0000301c: $ 7 <= 00000007
250@00003020: $ 8 <= 00000008
270@00003024: $ 9 <= 00000009
290@00003028: $10 <= 000a0000
310@0000302c: $11 <= 0000000b
330@00003030: $12 <= 0000000c
350@00003034: $13 <= 0000000d
370@00003038: $14 <= 0000000e
390@0000303c: $15 <= 0000000f
410@00003040: $16 <= 00000010
430@00003044: $17 <= 00000011
450@00003048: $18 <= 00000012
470@0000304c: $19 <= 00000013
490@00003050: $20 <= 00000014
510@00003054: $21 <= 00000015
530@00003058: $22 <= 00000016
550@0000305c: $23 <= 00000017
570@00003060: $24 <= 00000018
590@00003064: $25 <= 00000019
610@00003068: $26 <= 0000001a
630@0000306c: $27 <= 0000001b
650@00003070: $28 <= 001c0000
670@00003074: $29 <= 0000001d
690@00003078: $30 <= 0000001e
710@0000307c: $31 <= 0000001f
730@00003080: $ 9 <= 000a0009
750@00003084: $ 9 <= 00000009
770@00003088: $ 9 <= 00000008
770@0000308c: *00000000 <= 00000008
810@00003090: $ 9 <= 00000008
830@00003094: $ 9 <= 0000000c
830@00003098: *00000008 <= 0000000c
```

```
                810@00003090: $ 9 <= 00000008
                830@00003094: $ 9 <= 0000000c
                830@00003098: *00000008 <= 0000000c
                850@0000309c: *00000010 <= 0000000c
                890@000030a0: $ 9 <= 00000004
                910@000030a4: $ 9 <= 0000000c
                950@000030a8: $ 9 <= 00000000
ISim>
# run 1.00us
                1030@000030b0: $ 9 <= 00000008
                1050@000030a4: $ 9 <= 00000000
                1090@000030a8: $ 9 <= 00000008
                1170@000030b0: $ 9 <= 00000010
                1250@000030f0: $ 9 <= 000a0010
                1270@000030f4: $11 <= ffffffff
                1290@000030f8: $ 9 <= 000a1234
                1310@000030fc: $31 <= 00003104
                1330@00003100: $ 9 <= 000a4338
                1390@000030d4: $31 <= 000030dc
                1410@000030d8: $ 9 <= 000a7414
                1430@000030e4: $ 9 <= fff5bcc8
                1470@000030dc: *00000008 <= 000030dc
                1510@000030e0: $ 8 <= 0000000c
                1530@000030e4: $ 9 <= 000a7414
                1570@000030dc: *0000000c <= 000030dc
                1610@000030e0: $ 8 <= 00000010
                1630@000030e4: $ 9 <= fff5bcc8
                1670@000030dc: *00000010 <= 000030dc
                1710@000030e0: $ 8 <= 00000014
                1730@000030e4: $ 9 <= 000a7414
                1770@000030dc: *00000014 <= 000030dc
                1810@000030e0: $ 8 <= 00000018
                1830@000030e4: $ 9 <= fff5bcc8
                1870@000030dc: *00000018 <= 000030dc
                1910@000030e0: $ 8 <= 0000001c
                1930@000030e4: $ 9 <= 000a7414
                1970@000030dc: *0000001c <= 000030dc
ISim>
```

该程序最后是将所有 DM 中的值都赋为 0x000030dc 的死循环

四、思考题

1. 在本实验中你遇到了哪些不同指令组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？请有条理的罗列出来。(非常重要)

序号	类型	Tuse_RS	Tuse_RT	Tnew_E	冲突寄存器	解决方案	测试程序
1	R-J	0		1	RS	暂停 1 个时钟周期后从 M 到 D 转发	ori $ra $0,0x0000300c jr $ra
2	L-J	0		2	RS	暂停 2 个时钟周期后从 W 到 D 转发	lw $ra,0($0) jr $ra
3	R-B	0	0	1	RS 或 RT	暂停 1 个时钟周期后从 M 到 D 转发	addu $t1,$t1,$t2 beq $t1,$t2,loop
4	L-B	0	0	2	RS 或 RT	暂停两个	lw $t1,0($0)

						时钟周期后从 W 到 D 转发	beq $t1,$t2,loop
5	R-R	1	1	1	RS 或 RT	从 M 到 E 转发	addu $t1,$t1,$t2 subu $t1,$t1,$t2
6	L-R	1	1	2	RS 或 RT	暂停一个时钟周期后从 W 到 E 转发	lw $t1,0($0) addu $t1,$t1,$t2
7	R-R	1		1	RS	从 M 到 E 级转发	ori $t1,$0,1 lui $t1,1
8	L-R	1		2	RS	暂停一个时钟周期后从 W 到 E 转发	lw $t1,0($0) lui $t1,1
9	R-S	1	2	1	RS	从 M 到 E 级转发	lui $t1,1 sw $t1,0($0)
10	L-S	1	2	2	RS	暂停 1 个周期后从 W 到 E 级转发	lw $t1,0($0) sw $t2,0($t1)
11	L-S	1	2	2	RT	从 W 到 M 级转发	lw $t1,0($0) sw $t1,0($0)

序号	类型	Tuse_RS	Tuse_RT	Tnew_M	冲突寄存器	解决方案	测试程序
1	L-R-J	0		1	RS	暂停一个周期后从 W 到 D 级转发	lw $ra,0($0) lui $t2,1 jr $ra
2	L-R-B	0	0	1	RT 或 RT	暂停一个周期后从 W 到 D 级转发	lw $t1,0($0) lui $t2,1 beq $t1,$t2,loop

3	L-R-R	1	1	1	RS 或 RT	从 W 到 E 级转发	lw $t1,0($0)
							lui $t2,1
							addu $t1,$t3,$t1

4	L-R-R	1		1	RS	从 W 到 E 级转发	lw $t1,0($0) lui $t2,1 ori $t1,$t1,1

5	L-R-S	1	2	1	RS	从 W 到 E 级转发	lw $t1,0($0)
							lui $t2,1
							sw $t1,0($0)

6	R-R-J	0		0	RS	从 M 级到	addu $ra,$t1,$t2

						D 级转发	lui $t2,1 jr $ra
7	R-R-B	0	0	0	RS 或 RT	从 M 级到 D 级转发	addu $t1,$t1,$t2 lui $t3,1 beq $t1,$t2,loop

序号	类型	Tuse_RS	Tuse_RT	Tnew_W	冲突寄存器	解决方案	测试程序
1	R-N-N-J	0		0	RS	从 W 到 D 级转发	addu $ra,$t1,$t2 nop nop jr $ra
2	R-N-N-B	0	0	0	RS 或 RT	从 W 到 D 级转发	addu $t1,$t1,$t2 nop nop beq $t1,$t2,loop