计算机组成原理课程设计——36条流水线CPU

一. 个人信息

学号：072110112

姓名：冉中益

班号：1621102

专业：计算机科学与技术

二. 模块定义及代码实现

1. 数据通路

datapath_PPL

基本描述

统合所有流水段，流水段寄存器以及各个冒险处理模块，从而形成完整的流水线数据通路

模块接口

接口名	方向	描述
clk	I	时钟信号
rst	I	复位信号

功能

1. 取下一条执行指令
2. 根据控制信号对不同流水段的指令进行执行和操作
3. 包含冒险处理模块 对各种冒险进行处理

代码实现

```
`include"PC_PPL.v"
`include"IUnit.v"
`include"IF_ID.v"
`include"ID.v"
`include"ID_EX.v"
`include"EX.v"
`include"EX_Mem.v"
`include"Mem.v"
`include"Mem_Wr.v"
`include"Wr.v"
`include"Pre_Data_Hazard.v"
`include"branch_Data_Hazard.v"
`include"MULT_Data_Hazard.v"
`include"CPR_Data_Hazard.v"
`include"ERET_Data_Hazard.v"
`include"Load_use.v"

module datapath_PPL(clk,rst);
```

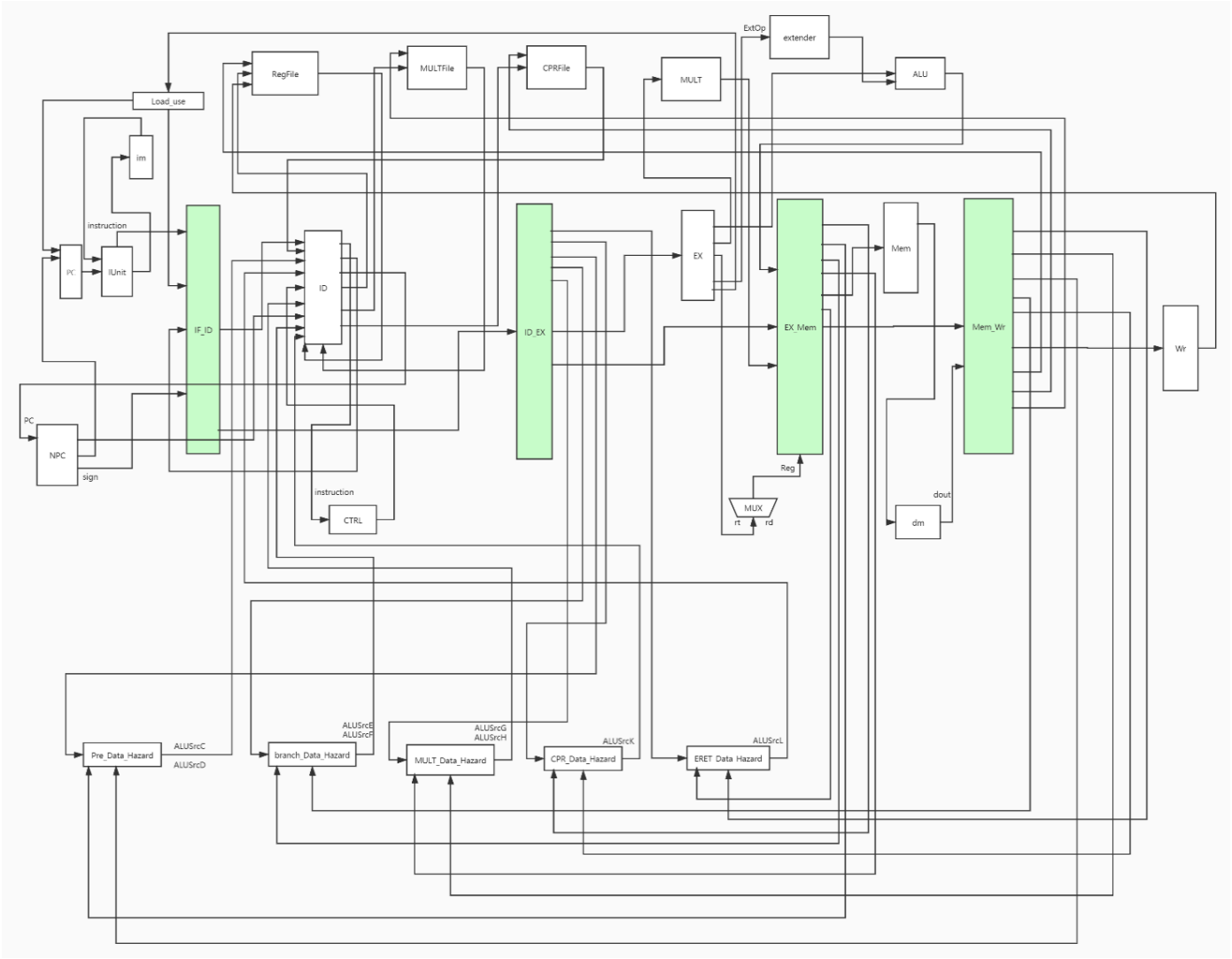
```
19
20      input clk,rst;
21
22      wire[31:2]  PC,NPC;
23      wire[31:2]  IF_PC,ID_PC,EX_PC,Mem_PC,Wr_PC;
24      wire[31:0]  instruction,ID_instruction,pre_instruction;
25      wire    sign,signal;
26      wire[31:0]  E,F;
27      wire  ID_Jump,EX_Jump;
28      wire  ID_Branch,EX_Branch;
29      wire[5:0] ID_op,EX_op,Mem_op,Wr_op;
30      wire[4:0] ID_rs,EX_rs,Mem_rs,Wr_rs;
31      wire[4:0] ID_rt,EX_rt,Mem_rt,Wr_rt;
32      wire[4:0] ID_rd,EX_rd,Mem_rd,Wr_rd;
33      wire[4:0] ID_shamt,EX_shamt,Mem_shamt,Wr_shamt;
34      wire[5:0] ID_func,EX_func,Mem_func,Wr_func;
35      wire[15:0]  ID_imm16,EX_imm16;
36      wire[25:0]  ID_target,EX_target;
37      wire[31:0]  ID_busA,EX_busA,Mem_busA,Wr_busA;
38      wire[31:0]  ID_busB,EX_busB,Mem_busB,Wr_busB;
39      wire    ID_RegDst,EX_RegDst,Mem_RegDst,Wr_RegDst;
40      wire  ID_RegWr,EX_RegWr,Mem_RegWr,Wr_RegWr;
41      wire  ID_ALUSrc,EX_ALUSrc;
42      wire  ID_MemWr,EX_MemWr,Mem_MemWr;
43      wire  ID_MemtoReg,EX_MemtoReg,Mem_MemtoReg,Wr_MemtoReg;
44      wire[1:0] ID_ExtOp,EX_ExtOp;
45      wire[4:0] ID_ALUctr,EX_ALUctr;
46      wire[31:0]  alure,Mem_alure,Wr_alure;
47      wire[4:0] EX_Reg,Mem_Reg,Wr_Reg;
48      wire[31:0]  Mem_dout,Wr_dout;
49      wire[31:0]  Wr_busW;
50      //wire[1:0] ALUSrcA,ALUSrcB;
51      wire[31:0]  A,B;
52      wire[1:0] ALUSrcC,ALUSrcD;
53      wire[1:0] ALUSrcE,ALUSrcF;
54      wire    Load_use;
55      wire    EX_MemRead;
56      //wire[31:0]  ID_hi_num;
57      wire[31:0]  EX_hi_num;
58      //wire[31:0]  ID_lo_num;
59      wire[31:0]  EX_lo_num;
60      wire[63:0]  EX_MULT_result,Mem_MULT_result,Wr_MULT_result;
```

```
61    wire[2:0] ALUSrcG,ALUSrcH;
62    wire[31:0]  G,H;
63    wire[1:0] ALUSrcK,ALUSrcL;
64    wire[31:0]  K;
65    wire[31:0]  EX_cs_num;
66
67    PC_PPL PCP(NPC,rst,clk,PC,
68  Load_use);
69    IUnit IPP(PC,IF_PC,instruction);
70    IF_ID
    IFIDPP(IF_PC,instruction,ID_PC,ID_instruction,ID_Jump,ID_Branch,ID_op,I
    D_func,clk,
71  pre_instruction,sign,signal,
72  Load_use);
73
74    ID
    IDPP(ID_PC,ID_op,ID_rs,ID_rt,ID_rd,ID_shamt,ID_func,ID_imm16,ID_target,
    ID_busA,ID_busB,ID_RegDst,ID_RegWr,ID_ALUSrc,ID_MemWr,ID_MemtoReg,
75  ID_Branch,ID_Jump,ID_ExtOp,ID_ALUctr,ID_instruction,NPC,
76  Wr_op,Wr_PC,Wr_alure,Wr_rs,Wr_rt,Wr_rd,Wr_shamt,Wr_func,Wr_busW,Wr_RegW
    r,Wr_RegDst,
77  alure,Mem_alure,Mem_dout,Mem_MemtoReg,
78  ALUSrcC,ALUSrcD,
79  PC,
80  sign,
81  pre_instruction,E,F,
82  ALUSrcE,ALUSrcF,
83  Wr_MULT_result,Wr_busA,Wr_busB,
84  G,H,EX_MULT_result,EX_busA,Mem_MULT_result,Mem_busA,
85  ALUSrcG,ALUSrcH,ALUSrcK,ALUSrcL,
86  A,B,K,
87  EX_busB,Mem_busB,
88  clk,rst);
89
90    ID_EX
    IDEXPP(ID_PC,ID_op,ID_rs,ID_rt,ID_rd,ID_shamt,ID_func,ID_imm16,ID_targe
    t,ID_busA,ID_busB,ID_RegDst,ID_RegWr,ID_ALUSrc,ID_MemWr,ID_MemtoReg,ID_
    Branch,ID_Jump,ID_ExtOp,ID_ALUctr,
91  G,H,K,
92  clk,
```

```
93  EX_PC,EX_op,EX_rs,EX_rt,EX_rd,EX_shamt,EX_func,EX_imm16,EX_target,EX_bu
    sA,EX_busB,EX_RegDst,EX_RegWr,EX_ALUSrc,EX_MemWr,EX_MemtoReg,EX_Branch,
    EX_Jump,EX_ExtOp,EX_ALUctr,
94  EX_hi_num,EX_lo_num,EX_cs_num,
95  Load_use,
96  signal,pre_instruction,E,F,
97  A,B);
98
99
100   EX
    EXPP(EX_PC,EX_op,EX_rs,EX_rt,EX_rd,EX_shamt,EX_func,EX_imm16,EX_target,
    EX_busA,EX_busB,
101 EX_RegDst,EX_RegWr,EX_ALUSrc,EX_MemWr,EX_MemtoReg,EX_Branch,EX_Jump,EX_
    ExtOp,EX_ALUctr,
102 alure,EX_Reg,clk,
103 Mem_alure,Wr_busW,
104 EX_MemRead,
105 EX_MULT_result,
106 EX_hi_num,EX_lo_num,
107 EX_cs_num);
108
109   EX_Mem
    EXMEMPP(EX_op,EX_Reg,EX_RegWr,EX_MemWr,EX_MemtoReg,EX_PC,EX_rs,EX_rt,EX
    _rd,EX_shamt,EX_func,EX_RegDst,EX_busA,EX_busB,alure,clk,
110 Mem_op,Mem_Reg,Mem_RegWr,Mem_MemWr,Mem_MemtoReg,Mem_alure,Mem_PC,Mem_rs
    ,Mem_rt,Mem_rd,Mem_shamt,Mem_func,Mem_RegDst,Mem_busA,Mem_busB,
111 EX_MULT_result,Mem_MULT_result);
112
113   Mem
    MEMPP(Mem_op,Mem_Reg,Mem_busB,Mem_RegWr,Mem_MemWr,Mem_MemtoReg,Mem_alur
    e,clk,Mem_dout);
114
115   Mem_Wr
    MEMWRPP(Mem_op,Mem_Reg,Mem_RegWr,Mem_MemtoReg,Mem_alure,Mem_dout,Mem_PC
    ,Mem_rs,Mem_rt,Mem_rd,Mem_shamt,Mem_func,Mem_RegDst,Mem_busA,Mem_busB,c
    lk,
116 Wr_op,Wr_Reg,Wr_RegWr,Wr_MemtoReg,Wr_alure,Wr_dout,Wr_PC,Wr_rs,Wr_rt,Wr
    _rd,Wr_shamt,Wr_func,Wr_RegDst,Wr_busA,Wr_busB,
117 Mem_MULT_result,Wr_MULT_result);
118
```

```
119     Wr
    WRPP(Wr_op,Wr_Reg,Wr_RegWr,Wr_MemtoReg,Wr_alure,Wr_dout,Wr_PC,Wr_rs,Wr_
    rt,Wr_rd,Wr_shamt,Wr_func,Wr_RegDst,Wr_busA,Wr_busB,clk,Wr_busW);
120
121     Pre_Data_Hazard
    PDHPP(Mem_RegWr,Mem_Reg,ID_rs,ID_rt,EX_RegWr,EX_Reg,Wr_RegWr,Wr_Reg,ALU
    SrcC,ALUSrcD);
122
123     branch_Data_Hazard
    BDHPP(Wr_RegWr,Wr_Reg,pre_instruction[25:21],pre_instruction[20:16],Mem
    _RegWr,Mem_Reg,ALUSrcE,ALUSrcF);
124
125     MULT_Data_Hazard
    MDHPP(ID_op,ID_func,EX_op,EX_func,Mem_op,Mem_func,Wr_op,Wr_func,ALUSrcG
    ,ALUSrcH);
126
127     CPR_Data_Hazard
    CDHPP(ID_op,EX_op,Mem_op,Wr_op,ID_rs,EX_rs,Mem_rs,Wr_rs,ID_rd,EX_rd,Mem
    _rd,Wr_rd,ALUSrcK);
128
129     ERET_Data_Hazard
    EDHPP(ID_op,ID_func,EX_op,Mem_op,Wr_op,EX_rs,Mem_rs,Wr_rs,EX_rd,Mem_rd,
    Wr_rd,ALUSrcL);
130
131     Load_use LUPP(EX_MemRead,EX_rt,ID_rs,ID_rt,Load_use);
132
133   endmodule
```

（附）36条流水线数据通路图



> 这张图是一开始设计45条流水线时进行绘制的 包括在后续的代码编写中也加入了45条的一些处理模块 但由于最终时间不够 没有完成额外9条指令的调试 所以在下面的介绍中不会介绍45条相对于36条的额外模块 由于将额外模块删去较为复杂 所以在提交的代码文件中仍有额外的模块！ 这些模块包括 `CPR_Data_Hazard, CPRFile, ERET_Data_Hazard, MULT_Data_Hazard, MULT, MULTFile` 另外 在各个模块中都涉及到了额外9条指令的部分 这些部分均不作解释

数据通路分析：

指令相关

1. **PC模块**： 计算取指令的地址。复位信号为1时，PC的值赋为13(PC末尾两位省略)。在没有出现Load_use冒险的情况下，时钟下降沿到来时PC赋值为NPC，否则保持不变。
2. **NPC模块**： 计算下一个PC，同时计算对分支指令是否跳转标记为sign。主要根据OP信号进行NPC计算方法的区分。
3. **im模块**： 用于取对应 `code.txt` 文件中的指令。

寄存器相关

1. **RegFile模块**： 寄存器的取数与存数。寄存器的大体框架与36条指令单周期CPU没有区别，多加了两个取数的地址参数。因为一条指令在寄存器中取数与存数的操作处于不同阶段，虽然RegFile置于ID模块，但实际使用时，Wr模块的操作也会在此进行，即写入操作。
2. **dm模块**：在数据内存中读数以及取数。同时需要考虑sb指令，其写入方法较为特殊。

冒险处理

1. **Pre_Data_Hazard模块**：处理从寄存器中读数存在的延迟问题。在ID段进行检测，处理在EX段，Mem段，Wr段可能出现的寄存器数据修改问题。取到的寄存器数应为最近一次修改的值。最后输出ALUSrcC和ALUSrcD进行冒险记录并输出。
2. **Branch_Data_Hazard模块**：处理pre_instruction从寄存器中读数存在的延迟问题。在ID段进行检测可能出现的寄存器数据修改问题。取到的寄存器数应为最近一次修改的值。最后输出ALUSrcE和ALUSrcF进行冒险记录并输出。
3. **Load_use模块**：首先检测是否遇到Load_use冒险，若遇到则置Load_use信号为1。

段寄存器模块

1. **IF_ID模块**：把IF段的数据传输到ID段。在出现syscall指令和ERET指令的情况下，将ID_instruction赋为0，若没有Load_use冒险并且不为分支跳转指令，ID_PC赋值为IF_PC，ID_instruction赋值为instruction；若没有Load_use冒险但为分支或跳转指令，则ID_PC保持不变；若出现Load_use冒险，则不进行任何操作。
2. **ID_EX模块**：把ID段的数据传输到EX段。若出现Load_use冒险，则EX段信号全部赋值为0；若出现分支指令不跳转，则赋值为保存的分支指令的下一条指令(即pre_instruction)的相关数值；其他情况下皆把EX段数值赋值为ID段数值。
3. **EX_Mem模块**：把EX段的数据传输到Mem段。
4. **Mem_Wr模块**：把Mem段的数据传输到Wr段。

运算模块

alu模块：输入端口为：ALUctr, busA, busB与exout选择后的结果data2，exout，instruction[10:6]，输出端口为alure,zero；

以上模块较为重要，在此详细说明。综合以上模块，根据对应控制信号进行操作。因R,I,J类型的指令都存在这样的情况：同种类型的指令之间存在路径不同的现象，所以按类型指令给出总体路径不具备很好的代表性，若要综述同一类型的指令之间的路径，要所有同一类型的指令都满足，则R,I,J三种类型的综述路径几乎一致，因而此处不给出类型指令数据通路演示。

alu

基本描述

根据不同的译码结果，对传入的数据信号进行不同操作的基本运算，并输出运算结果

模块接口

接口名	方向	描述
ALUctr	I	操作信号
A	I	输入A
B	I	输入B
immedia	I	扩展后立即数
shamt	I	移位数
result	O	ALU运算结果
zero	O	0标志位

功能

根据不同的操作码进行不同的基本运算

代码实现

各操作码对应的运算类型已经在代码中通过注释方式明确给出 此处不再附上操作码与运算类型的对应关系表

```
module alu(ALUctr,A,B,immedia,shamt,result,zero,G,H,K);
  input[4:0]  ALUctr;
  input[31:0] A,B,G,H,K;
  input[31:0] immedia;
  input[4:0]  shamt;
  output[31:0]  result;
  output    zero;

  reg[31:0] result;
  assign    zero = (result == 32'b0);
  integer   i;

  always @(*)
    begin
      case(ALUctr)
        5'b00000:result <= A + B;//addu
```

```
          5'b00001:result <= A - B;//subu
          5'b00010://slt
            begin
              if(A[31] ^ B[31]) result <= (A[31]==1)?1:0;
              else  result <= (A<B)?1:0;
            end
          5'b00011:result <= A & B;//and
          5'b00100:result <= ~(A | B);//nor
          5'b00101:result <= A | B;//or
          5'b00110:result <= A ^ B;//xor
          5'b00111:result <= B << shamt;//sll
          5'b01000:result <= B >> shamt;//srl
          5'b01001:result <= (A<B)?1:0;//sltu
          //5'b01010:jalr
          //5'b01011:jr
          5'b01100:result <= B << A;//sllv
          5'b01101:result <= $signed(B) >>> shamt;//sra
          5'b01110:result <= $signed(B) >>> A;//srav
          5'b01111:result <= B >> A;//srlv
          5'b10000:result <= A + immedia;//addiu sign extension
          5'b10001://slti sign extension
            begin
              if(A[31] ^ immedia[31]) result <= (A[31]==1)?1:0;
              else  result <= (A<immedia)?1:0;
            end
          5'b10010:result <= (A<immedia)?1:0;//sltiu,sign extension
          5'b10011:result <= A & immedia;//andi
          5'b10100:result <= A | immedia;//ori
          5'b10101:result <= A ^ immedia;//xori
          5'b10110:result <= immedia;//lui
          5'b10111:result <= G;//mflo
          5'b11000:result <= H;//mfhi
          5'b11001:result <= K;//mfc0
        endcase
      end
endmodule
```

dm_4k

基本描述

数据存储器 对数据进行写入存储和读出

模块接口

接口名	方向	描述
op	I	指令操作码 用于sb指令的判断
addr	I	地址码
din	I	数据输入
dout	O	数据输出
clk	I	时钟信号
we	I	写使能信号

功能

1. `we` 信号为1时将 `din` 中的数据写入 `dm` 中
2. 将 `addr` 地址码指定的数据赋给 `dout` 连接输出到其他模块中

代码实现

```
module dm_4k(op, addr, din, dout, clk, we);

input[5:0]  op;
input[31:0] addr;
input[31:0] din;
input   clk;
input   we;
output[31:0]  dout;

reg[7:0] dm[4095:0]; //采取以字节位单位存储数据才是最合理的

integer i;

always @(negedge clk)
begin
if(we)
begin
    if(op == 6'b101000)   //sb
        dm[addr] <= din[7:0];
```

```
        else {dm[addr + 3], dm[addr + 2], dm[addr + 1], dm[addr]} <= din;
    end
    end

    assign dout = {dm[addr + 3], dm[addr + 2], dm[addr + 1], dm[addr]};

    endmodule
```

extender

基本描述

位扩展器 根据传入的控制信号进行不同类型的扩展操作

模块接口

接口名	方向	描述
immedia	I	输入立即数
exop	I	扩展指令操作码
exout	O	扩展输出

功能

根据指令码 `exop` 对立即数进行0扩展、符号扩展或低位0扩展（用于lui指令）

`exop` 与操作的关系可见代码

代码实现

```
module extender(immedia,exop,exout);
  input[15:0] immedia;
  input[1:0]  exop;
  output[31:0]  exout;
  reg[31:0] exout;

  always@(*)
    if(exop == 2'b00)
      exout <= {16'b0,immedia};//zero extender
    else if(exop == 2'b01)
      exout <= {{16{immedia[15]}},immedia};//sign extender
    else if(exop == 2'b10)
      exout <= {immedia,16'b0};//lui extender
```

```
14  endmodule
```

im_4k

基本描述

从 `code.txt` 中读取并存储指令，每一周期根据地址返回响应指令

模块接口

接口名	方向	描述
addr	I	指令地址
dout	O	指令输出

功能

1. 初始化时从 `code.txt` 中读取指令 并将其按序存储在 `im` 指令存储器中
2. 每一周期根据传入的addr读取指令，通过 `dout` 接口连接到其他模块输入中

代码实现

```
1  module im_4k(addr,dout);
2    input[11:2] addr;
3    output[31:0]  dout;
4
5    reg[31:0] im[1023:0];
6    integer   i;
7    initial
8      begin
9        $readmemh("code.txt",im);
10     end
11 //read data in code.txt
12   assign  dout = im[addr];
13 endmodule
```

mux2_5

基本描述

5位宽二路选择器 根据选择信号选通不同数据

模块接口

接口名	方向	描述
a	I	数据输入a
b	I	数据输入b
s	I	选择信号
y	O	数据输出

功能

根据选通信号 `s` 选通数据 `s == 0` 时选通 `a` `s == 1` 时选通 `b`

代码实现

```
module mux2_5(a,b,s,y);
//choose data by 's'
  input[4:0]  a,b;
  input   s;
  output[4:0] y;
//5-bit binary number

  reg[4:0]  y;
//reg-type output
  always@(*)
    if(s == 0)
      y <= a;
    else if(s == 1)
      y <= b;
endmodule
```

RegFile_PPL

基本描述

32 * 32bit寄存器组 既可向指定寄存器写入数据 也可从指定寄存器中读取数据

模块接口

模块名	方向	描述
op	I	指令操作码 辅助判断一些对寄存器组较为特殊的指令
PC	I	当前指令地址
alure	I	alu运算结果
rs	I	Wr段rs寄存器编号
rt	I	Wr段rt寄存器编号
rd	I	Wr段rd寄存器编号
shamt	I	移位指令移位数
func	I	R型指令辅助操作码
data	I	输入数据
RegWr	I	寄存器组写使能信号
RegDst	I	目标寄存器选择信号
clk	I	时钟信号
reset	I	复位信号
rs2	I	ID段rs寄存器编号
rt2	I	ID段rt寄存器编号
rs3	I	pre_instruction中rs寄存器编号
rt3	I	pre_instruction中rt寄存器编号
ra	O	ID段busA输出
rb	O	ID段busB输出
ra2	O	pre_instruction的busA输出
rb2	O	pre_instrcution的busB输出

功能

1. `ID` 段实现寄存器读 将 `rs` 和 `rt` 寄存器的值分别输出到 `busA` 和 `busB`
2. `Wr` 段实现寄存器写 将 `data` 写入指定寄存器中
3. 对 `branch` 指令的情况 在 `ID` 段同时读出 `pre_instruction` 中 `rs` 和 `rt` 寄存器中的值并保存供后续使用

代码实现

```
1  module
   RegFile_PPL(op,PC,alure,rs,rt,rd,shamt,func,data,RegWr,RegDst,rs2,rt2,ra
   ,rb,rs3,rt3,ra2,rb2,clk,reset);
2    input[5:0]  op;
3    input[31:2] PC;
```

```
4     input[31:0] alure;
5     input[4:0]  rs,rt,rd,shamt;
6     input[5:0]  func;
7     input[31:0] data;
8     input   RegWr;
9     input   RegDst;
10    input     clk;
11    input   reset;
12    input[4:0]  rs2,rt2;
13    input[4:0]  rs3,rt3;
14    output[31:0]  ra,rb;
15    output[31:0]  ra2,rb2;
16
17    reg[31:0] rgs[31:0];
18    integer   i;
19    always@(negedge clk or posedge reset)
20      begin
21        if(reset == 1)
22        begin
23          for(i = 0;i < 32;i = i + 1)
24            rgs[i] <= 32'b0;
25  //initial assignment
26        end
27        if(RegWr)
28          case(op)
29            6'b100000:
30              begin
31                rgs[rt] <= data;
32                /*if(alure[1:0] == 2'b00)   rgs[rt]≤
    {{24{data[7]}},data[7:0]};
33                if(alure[1:0] == 2'b01)   rgs[rt]≤
    {{24{data[15]}},data[15:8]};
34                if(alure[1:0] == 2'b10)   rgs[rt]≤
    {{24{data[23]}},data[23:16]};
35                if(alure[1:0] == 2'b11)   rgs[rt]≤
    {{24{data[31]}},data[31:24]};*/
36              end
37            6'b100100:
38              begin
39                rgs[rt] <= data;
40                /*if(alure[1:0] == 2'b00)   rgs[rt]≤{{24'b0},data[7:0]};
41                if(alure[1:0] == 2'b01)   rgs[rt]≤{{24'b0},data[15:8]};
```

```
42              if(alure[1:0] == 2'b10)   rgs[rt]<={{24'b0},data[23:16]};
43              if(alure[1:0] == 2'b11)   rgs[rt]<=
    {{24'b0},data[31:24]};*/
44            end
45          6'b000011: rgs[31]<={(PC+30'd2),2'b00};//jal
46          6'b001111: rgs[rt]<={rd,shamt,func,16'b0};//lui
47          default:
48            begin
49              if(rt == 0 && rd == 5'b11111 && shamt == 0 && func ==
    6'b001001)
50                rgs[31] <= {(PC+30'd2),2'b00};//jalr
51              else if(RegDst == 1)  rgs[rd] <= data;
52
53              else if(RegDst == 0)  rgs[rt] <= data;
54            end
55        endcase
56     end
57
58    assign  ra = (rs2!=0)? rgs[rs2] : 0;
59    assign  rb = (rt2!=0)? rgs[rt2] : 0;
60    assign  ra2 = (rs3!=0)?rgs[rs3] : 0;
61    assign  rb2 = (rt3!=0)?rgs[rt3] : 0;
62  // get data from register
63  endmodule
```

PC_PPL

基本描述

对当前指令地址（PC）进行操作的模块

模块接口

接口名	方向	描述
NPC	I	下指令地址
rst	I	复位信号
clk	I	时钟信号
PC	O	当前指令地址
Load_use	I	Load_use冒险信号

功能

1. 对 `PC` 进行复位
2. 无 `Load_use` 冒险时对 `PC` 正常赋值
3. 有 `Load_use` 冒险时阻塞 `PC` 赋值

代码实现

```
module PC_PPL(NPC,rst,clk,PC,
Load_use);
  input[31:2] NPC;
  input   rst;
  input     clk;
  input   Load_use;

  output[31:2]  PC;

  reg[31:2] PC;
  initial
  begin
    PC = 0;
  end
  always @(negedge clk or posedge rst)
    begin
      if(rst == 1)
        PC<=0;
//initial assignment pc ⩽ 13
      else if(!Load_use)
        PC<=NPC;
    end
endmodule
```

NPC

基本描述

用来计算下地址的逻辑模块

模块接口

接口名	方向	描述
PC	I	当前指令地址
NPC	O	下指令地址
jump	I	jump类指令标志信号
branch	I	branch类指令标志信号
zero	I	0标志信号
op	I	指令操作码
target	I	J型指令target域
imm16	I	I型指令立即数
busA	I	rs寄存器的值
rs	I	rs寄存器编号
rt	I	rt寄存器编号
rd	I	rd寄存器编号
shamt	I	移位指令移位数
func	I	R型指令辅助操作码
sign	O	实际跳转标志位 0表示实际发生了跳转 1表示实际未发生

功能

1. 根据当前指令操作码计算正确的下地址 并将其赋值给 `NPC`
2. 判断跳转指令实际上是否发生了调转 将判断结果赋值给 `sign` 以供后续使用

代码实现

```
module
NPC(PC,NPC,jump,branch,zero,op,target,imm16,busA,rs,rt,rd,shamt,func,
sign,CPR14);
  input[31:2] PC;
  output[31:2]  NPC;
  input   jump,branch,zero;
  input[5:0]  op;
  input[25:0] target;
  input[15:0] imm16;
  input[31:0] busA;
  input[4:0]  rs,rt,rd,shamt;
  input[5:0]  func;
  input[31:0] CPR14;

```

```
14    output reg  sign;
15
16    reg[31:2] NPC;
17    wire[31:2]  B_NPC = {{14{imm16[15]}},imm16[15:0]} + PC - 1; //要减1是
      因为译码完成传入时已经到了跳转指令的ID段 PC已经 + 1了
18    //NPC with branch instructions
19
20    wire[31:2]  J_NPC = {PC[31:28],target};
21    //NPC with jump instructions
22
23    wire[31:2]  N_NPC = PC + 1;
24    //normal NPC
25
26    reg[31:2] EXC_ENTER_ADDR;
27    //jump address of ERET isntruction
28
29    initial
30    begin
31        NPC = 0;
32        sign = 0;
33        EXC_ENTER_ADDR = 0;
34    end
35    //initial assignment
36
37    always @(*)
38      begin
39        case(op)
40          6'b000100: //beq
41            begin
42              if(zero==1 && branch==1)//jump when two inputs equal
43              begin
44                NPC <= B_NPC;
45                sign <= 0;
46              end
47              else
48              begin
49                NPC <= N_NPC;
50                sign <= 1;
51              end
52            end
53          6'b000101: //bne
54            begin
```

```
55              if(zero==0 && branch==1)//jump when two inputs not equal
56              begin
57                NPC <= B_NPC;
58                sign <= 0;
59              end
60              else
61              begin
62                NPC <= N_NPC;
63                sign <= 1;
64              end
65            end
66          6'b000001:
67            begin
68              if(rt == 1 && branch == 1&& (busA == 0||busA[31] ==
    0))//bgez - jump when busA ⩾ 0
69              begin
70                NPC <= B_NPC;
71                sign <= 0;
72              end
73                else if(rt == 0 && branch == 1 && busA[31] == 1 && busA
    ≠ 0)//bltz - jump when busA < 0
74              begin
75                NPC <= B_NPC;
76                sign <= 0;
77              end
78              else
79              begin
80                NPC <= N_NPC;
81                sign <= 1;
82              end
83            end
84          6'b000111: //bgtz - jump when busA > 0
85            begin
86              if(branch == 1 && busA[31] == 0 && busA ≠ 0)
87              begin
88                NPC <= B_NPC;
89                sign <= 0;
90              end
91              else
92              begin
93                NPC <= N_NPC;
94                sign <= 1;
```

```
95              end
96            end
97          6'b000110: //blez - jump when busA ⩽ 0
98            begin
99              if(branch == 0 && (busA[31] == 1||busA == 0))
100             begin
101               NPC <= B_NPC;
102               sign <= 0;
103             end
104             else
105             begin
106               NPC <= N_NPC;
107               sign <= 1;
108             end
109           end
110         6'b000010: //j
111           begin
112             if(jump == 1)
113             begin
114               NPC <= J_NPC;
115               sign <= 0;
116             end
117             else
118             begin
119               NPC <= N_NPC;
120               sign <= 1;
121             end
122           end
123         6'b000011: //jal
124           begin
125             if(jump == 1)
126             begin
127               NPC <= J_NPC;
128               sign <= 0;
129             end
130             else
131             begin
132               NPC <= N_NPC;
133               sign <= 1;
134             end
135           end
136         6'b000000:
```

```
137             begin
138                 if(rt == 0 && imm16 == 16'b1111100000001001)
139               begin
140                 NPC <= busA[31:2];//jalr
141                 sign <= 0;
142               end
143                 else if(rt == 0 && imm16 == 16'b0000000000001000)
144               begin
145                 NPC <= busA[31:2];//jr
146                 sign <= 0;
147               end
148               else if(func == 6'b001100)
149               begin
150                 NPC <= EXC_ENTER_ADDR;//ERET
151                 sign <= 0;
152               end
153               else
154               begin
155                 NPC <= N_NPC;
156                 sign <= 0;
157               end
158             end
159           6'b010000:
160             begin
161               if(func == 6'b011000)//syscall
162               begin
163                 NPC <= CPR14[31:2];
164                 sign <= 0;
165               end
166               else
167               begin
168                 NPC <= N_NPC;
169                 sign <= 0;
170               end
171             end
172           default: //normal instructions
173             begin
174               NPC <= N_NPC;
175               sign <= 0;
176             end
177         endcase
178       end
```

```
179    endmodule
```

IUnit

基本描述

取指令部件

模块接口

接口名	方向	描述
PC	I	当前指令地址
IF_PC	O	IF段PC
instruction	O	指令

功能

1. 根据 `PC` 取出当前指令
2. 对 `IF_PC` 赋为 `PC` 的值

代码实现

```
1   `include"im.v"
2
3   module IUnit(PC,IF_PC,instruction);
4     input[31:2] PC;
5     output[31:2]  IF_PC;
6     output[31:0]  instruction;
7
8     im_4k imD(PC[11:2],instruction);
9     assign  IF_PC = PC;
10  endmodule
```

IF_ID

基本描述

`IF_ID` 流水段寄存器

模块接口

信号名	方向	描述
IF_PC	I	IF段PC的值
Instruction	I	IF段instruction的值
ID_PC	O	ID段PC的值
ID_instruction	O	ID段instruction的值
Jump	I	ID段jump的值
Branch	I	ID段branch的值
op	I	ID段op的值
func	I	ID段func的值
clk	I	时钟信号
Load_use	I	Load_use冒险信号
sign	I	跳转指令实际是否发生跳转的标志
ID_instruction	O	ID段指令
pre_instruction	O	保存的跳转指令的下一条指令
signal	O	跳转指令实际是否发生跳转的标志

功能

1. 作为 `IF` 和 `ID` 段的信息中转站
2. 生跳转指令时 将 `ID` 段指令清0（中止指令）并将下一条指令保存到 `pre_instruction` 中
3. 发生 `load_use` 冒险时 阻塞对 `ID` 段的指令赋值 从而实现插入气泡

代码实现

```
module
IF_ID(IF_PC,instruction,ID_PC,ID_instruction,Jump,Branch,op,func,clk,
pre_instruction,sign,signal,
Load_use);
  input[31:2] IF_PC;
  input[31:0] instruction;
  input   clk;
  input   Jump;
  input   Branch;
  input   Load_use;
  input   sign;
  input[5:0]  op,func;

  output reg[31:2]  ID_PC;
```

```
14    output reg[31:0]  ID_instruction;
15    output reg[31:0]  pre_instruction;
16    output reg    signal;
17  //assignment
18
19    initial
20    begin
21      ID_PC = 30'b0;
22      ID_instruction = 32'b0;
23      pre_instruction = 32'b0;
24      signal = 0;
25    end
26  //initial assignment
27
28    always@(negedge clk)
29    begin
30      if((op == 6'b000000 && func == 6'b001100) || (op == 6'b010000 &&
    func == 6'b011000))
31  //syscall and ERET
32      begin
33        ID_instruction <= 32'b0;
34        signal <= sign;
35      end
36      else if(!Load_use && !Jump && !Branch)
37  //normal instructions
38      begin
39        ID_PC <= IF_PC;
40        ID_instruction <= instruction;
41        signal <= sign;
42      end
43      else if(!Load_use && (Jump || Branch))
44  //jump and branch instructions with a delay slot
45      begin
46        // ID_instruction ⩽ instruction;
47        ID_instruction <= 32'b0;//this is for code without a delay slot
48        pre_instruction <= instruction;
49        signal <= sign;
50      end
51    end
52  endmodule
```

ID

基本描述

`ID` 流水段

模块接口

接口名	方向	描述
ID_PC	I	ID段PC的值
ID_op	O	ID段op的值
ID_rs	O	ID段rs的值
ID_rt	O	ID段rt的值
ID_rd	O	ID段rd的值
ID_shamt	O	ID段shamt的值
ID_func	O	ID段func的值
ID_imm16	O	ID段imm16的值
ID_target	O	ID段target的值
ID_busA	O	ID段busA的值
ID_busB	O	ID段busB的值
ID_RegDst	O	ID段RegDst的值
ID_RegWr	O	ID段RegWr的值
ID_ALUSrc	O	ID段ALUSrc的值
ID_MemWr	O	ID段MemWr的值
ID_MemtoReg	O	ID段MemtoReg的值
ID_Branch	O	ID段Branch的值
ID_Jump	O	ID段Jump的值
ID_ExtOp	O	ID段ExtOp的值
ID_ALUctr	O	ID段ALUctr的值
ID_instruction	I	ID段instruction的值
NPC	O	当前NPC的值
Wr_op	I	Wr段op的值
Wr_PC	I	Wr段PC的值
Wr_alure	I	Wr段alu结果的值
Wr_rs	I	Wr段rs的值
Wr_rt	I	Wr段rt的值
Wr_rd	I	Wr段rd的值
Wr_shamt	I	Wr段shamt的值

Wr_func	I	Wr段func的值
Wr_busW	I	Wr段busW的值
Wr_RegWr	I	Wr段RegWr的值
Wr_RegDst	I	Wr段RegDst的值
alure	I	EX段alu的结果
Mem_alure	I	Mem段的alu的结果
Mem_dout	I	Mem段dout的结果
Mem_MemtoReg	I	Mem段MemtoReg的结果
ALUSrcC	I	数据冒险检测ALUSrcC的值
ALUSrcD	I	数据冒险检测ALUSrcD的值
PC	I	当前PC的值
sign	O	判断branch指令是否跳转的标志
pre_instruction	I	跳转指令的下一条指令
E	O	pre_instruction的busA数据冒险检测结果
F	O	pre_instruction的busB数据冒险检测结果
ALUSrcE	I	数据冒险检测ALUSrcE的值
ALUSrcF	I	数据冒险检测ALUSrcF的值
Wr_busA	I	Wr段busA的值
Wr_busB	I	Wr段busB的值
EX_busA	I	EX段busA的值
Mem_busA	I	Mem段busA的值
A	O	ID段busA数据冒险检测结果
B	O	ID段busB数据冒险检测结果
EX_busB	I	EX段busB的值
Mem_busB	I	Mem段的busB的值
clk	I	时钟信号
rst	I	复位信号

功能

1. 利用 `ctrl` 模块对当前指令进行译码
2. 接收 `Wr` 段的各信号进行寄存器写
3. 直接在 `ID` 段求出下地址 `NPC`
4. 根据数据冒险的情况选择出 `EX` 段 `ALU` 两个输入端口的数值

代码实现

```
`include"ctrl.v"
`include"RegFile_PPL.v"
`include"MULTFile.v"
`include"CPRFile.v"
`include"NPC.v"

module
ID(ID_PC,ID_op,ID_rs,ID_rt,ID_rd,ID_shamt,ID_func,ID_imm16,ID_target,ID_busA,ID_busB,ID_RegDst,ID_RegWr,ID_ALUSrc,ID_MemWr,ID_MemtoReg,
ID_Branch,ID_Jump,ID_ExtOp,ID_ALUctr,ID_instruction,NPC,
Wr_op,Wr_PC,Wr_alure,Wr_rs,Wr_rt,Wr_rd,Wr_shamt,Wr_func,Wr_busW,Wr_RegWr,Wr_RegDst,
alure,Mem_alure,Mem_dout,Mem_MemtoReg,
ALUSrcC,ALUSrcD,
PC,
sign,
pre_instruction,E,F,
ALUSrcE,ALUSrcF,
Wr_MULT_result,Wr_busA,Wr_busB,
G,H,EX_MULT_result,EX_busA,Mem_MULT_result,Mem_busA,
ALUSrcG,ALUSrcH,ALUSrcK,ALUSrcL,
A,B,K,
EX_busB,Mem_busB,
clk,reset);

  input clk,reset;
  input[31:2] ID_PC;
  input[31:0] ID_instruction;
  input[5:0]  Wr_op;
  input[31:2] Wr_PC;
  input[31:0] Wr_alure;
  input[4:0]  Wr_rs,Wr_rt,Wr_rd,Wr_shamt;
  input[5:0]  Wr_func;
  input[31:0] Wr_busW;
  input Wr_RegWr,Wr_RegDst;

  input[1:0]  ALUSrcC,ALUSrcD;
  input[1:0]  ALUSrcE,ALUSrcF;
  input[31:0] alure,Mem_alure,Mem_dout;
  input Mem_MemtoReg;
```

```
38
39      input[31:2] PC;
40      input[31:0] pre_instruction;
41      input[63:0] Wr_MULT_result,EX_MULT_result,Mem_MULT_result;
42      input[31:0] Wr_busA,EX_busA,Mem_busA;
43      input[2:0]  ALUSrcG,ALUSrcH;
44      input[1:0]  ALUSrcK,ALUSrcL;
45      input[31:0] EX_busB,Mem_busB,Wr_busB;
46
47      output[5:0] ID_op;
48      output[4:0] ID_rs,ID_rt,ID_rd,ID_shamt;
49      output[5:0] ID_func;
50      output[15:0]  ID_imm16;
51      output[25:0]  ID_target;
52      output[31:0]  ID_busA,ID_busB;
53      output
    ID_RegDst,ID_RegWr,ID_ALUSrc,ID_MemWr,ID_MemtoReg,ID_Branch,ID_Jump;
54      output[1:0] ID_ExtOp;
55      output[4:0] ID_ALUctr;
56      output[31:2]  NPC;
57      output    sign;
58      output[31:0]  K;
59
60      output[31:0]  E,F;
61      output[31:0]  G,H;
62      wire[31:0]  ID_hi_num,ID_lo_num;
63
64      wire[31:0]  pre_busA,pre_busB;
65      wire[31:2]  FormerPC;
66      wire    ID_zero;
67      wire[31:0]  cs_num;
68      assign  FormerPC = ID_PC - 1;//Is this correct? check it later
69
70      assign  ID_op = ID_instruction[31:26];
71      assign  ID_rs = ID_instruction[25:21];
72      assign  ID_rt = ID_instruction[20:16];
73      assign  ID_rd = ID_instruction[15:11];
74      assign  ID_shamt = ID_instruction[10:6];
75      assign  ID_func = ID_instruction[5:0];
76      assign  ID_imm16 = ID_instruction[15:0];
77      assign  ID_target = ID_instruction[25:0];
78
```

```
79    wire[31:2]  Wr_FormerPC;
80    assign    Wr_FormerPC = Wr_PC - 1;
81
82    wire[31:0]  CPR14;
83
84    ctrl
ctrlPPL(ID_op,ID_rs,ID_rt,ID_rd,ID_shamt,ID_func,ID_RegDst,ID_RegWr,ID_
ALUSrc,ID_MemWr,ID_MemtoReg,ID_ExtOp,ID_ALUctr,ID_Branch,ID_Jump);
85    RegFile_PPL
REGPPL(Wr_op,Wr_FormerPC,Wr_alure,Wr_rs,Wr_rt,Wr_rd,Wr_shamt,Wr_func,Wr
_busW,Wr_RegWr,Wr_RegDst,ID_rs,ID_rt,ID_busA,ID_busB,pre_instruction[25
:21],pre_instruction[20:16],pre_busA,pre_busB,clk,reset);
86    MULTFile
MFPPL(Wr_op,Wr_rt,Wr_rd,Wr_shamt,Wr_func,Wr_MULT_result,ID_hi_num,ID_lo
_num,Wr_busA,clk);
87    CPRFile
CPRPPL(Wr_op,Wr_rs,Wr_rt,Wr_rd,Wr_shamt,Wr_func,ID_rd,cs_num,Wr_busB,Wr
_FormerPC,CPR14,clk);
88  //some operations with data in other segments, mainly the write
    register operation
89
90    output[31:0]  A,B;
91
92    assign  A = (ALUSrcC == 2'b00)? ID_busA :
93      (ALUSrcC == 2'b01)? alure :
94      (ALUSrcC == 2'b10 && Mem_MemtoReg == 0)? Mem_alure :
95      (ALUSrcC == 2'b10 && Mem_MemtoReg == 1)? Mem_dout :
96      (ALUSrcC == 2'b11)? Wr_busW : 0;
97    assign  B = (ALUSrcD == 2'b00)? ID_busB :
98      (ALUSrcD == 2'b01)? alure :
99      (ALUSrcD == 2'b10 && Mem_MemtoReg == 0)? Mem_alure :
100     (ALUSrcD == 2'b10 && Mem_MemtoReg == 1)? Mem_dout :
101     (ALUSrcD == 2'b11)? Wr_busW : 0;
102 //Pre_data_hazard
103
104   assign  E = (ALUSrcE == 2'b00)? pre_busA :
105     (ALUSrcE == 2'b10)? Wr_busW : 0;
106
107   assign  F = (ALUSrcF == 2'b00)? pre_busB :
108     (ALUSrcF == 2'b10)? Wr_busW : 0;
109 //branch_data_hazard
110
```

```
111     assign  G = (ALUSrcG == 3'b000)? ID_lo_num :
112       (ALUSrcG == 3'b001)? EX_busA :
113       (ALUSrcG == 3'b010)? EX_MULT_result[31:0] :
114       (ALUSrcG == 3'b011)? Mem_busA :
115       (ALUSrcG == 3'b100)? Mem_MULT_result[31:0] :
116       (ALUSrcG == 3'b101)? Wr_busA :
117       (ALUSrcG == 3'b110)? Wr_MULT_result[31:0] :0;
118
119     assign  H = (ALUSrcH == 3'b000)? ID_hi_num :
120       (ALUSrcH == 3'b001)? EX_busA :
121       (ALUSrcH == 3'b010)? EX_MULT_result[63:32] :
122       (ALUSrcH == 3'b011)? Mem_busA :
123       (ALUSrcH == 3'b100)? Mem_MULT_result[63:32] :
124       (ALUSrcH == 3'b101)? Wr_busA :
125       (ALUSrcH == 3'b110)? Wr_MULT_result[63:32] :0;
126   //MULT_data_hazard
127
128     assign  K = (ALUSrcK == 2'b00)? cs_num :
129       (ALUSrcK == 2'b01)? EX_busB :
130       (ALUSrcK == 2'b10)? Mem_busB :
131       (ALUSrcK == 2'b11)? Wr_busB :0;
132   //CPR_data_hazard
133     assign  ID_zero = (A == B)?1:0;
134
135     wire[31:0]  L;
136     assign  L = (ALUSrcL == 2'b00)? CPR14 :
137       (ALUSrcL == 2'b01)? EX_busB :
138       (ALUSrcL == 2'b10)? Mem_busB :
139       (ALUSrcL == 2'b11)? Wr_busB :0;
140   //ERET_data_hazard
141
142     NPC
      NPCPPL(PC,NPC,ID_Jump,ID_Branch,ID_zero,ID_op,ID_target,ID_imm16,A,ID_r
      s,ID_rt,ID_rd,ID_shamt,ID_func,sign,L);
143   //NPC calculation
144
145   endmodule
```

ID_EX

基本描述

`ID_EX` 流水段寄存器

模块接口

接口名	方向	描述
ID_PC	I	ID段PC的值
ID_op	I	ID段op的值
ID_rs	I	ID段rs的值
ID_rt	I	ID段rt的值
ID_rd	I	ID段rd的值
ID_shamt	I	ID段shamt的值
ID_func	I	ID段func的值
ID_imm16	I	ID的imm16的值
ID_target	I	ID段target的值
ID_busA	I	ID段busA的值
ID_busB	I	ID段busB的值
ID_RegDst	I	ID段RegDst的值
ID_RegWr	I	ID段RegWr的值
ID_ALUSrc	I	ID段ALUSrc的值
ID_MemWr	I	ID段MemWr的值
ID_MemtoReg	I	ID段MemtoReg的值
ID_Branch	I	ID段Branch的值
ID_Jump	I	ID段Jump的值
ID_ExtOp	I	ID段ExtOp的值
ID_ALUctr	I	ID段ALUctr的值
clk	I	时钟信号
EX_PC	O	EX段PC的值
EX_op	O	EX段op的值
EX_rs	O	EX段rs的值
EX_rt	O	EX段rt的值
EX_rd	O	EX段rd的值
EX_shamt	O	EX段shamt的值
EX_func	O	EX段func的值
EX_imm16	O	EX段imm16的值

EX_target	O	EX段target的值
EX_busA	O	EX段busA的值
EX_busB	O	EX段busB的值
EX_RegDst	O	EX段RegDst的值
EX_RegWr	O	EX段RegWr的值
EX_ALUSrc	O	EX段ALUSrc的值
EX_MemWr	O	EX段MemWr的值
EX_MemtoReg	O	EX段MemtoReg的值
EX_Branch	O	EX段Branch的值
EX_Jump	O	EX段Jump的值
EX_ExtOp	O	EX段ExtOp的值
EX_ALUctr	O	EX段ALUctr的值
Load_use	I	Load_use冒险判断
signal	I	跳转指令跳转判断
pre_instruction	I	跳转指令的下一条指令
E	I	Pre_instruction的busA数据冒险检测结果
F	I	Pre_instruction的busB数据冒险检测结果
A	I	ID段busA数据冒险检测结果
B	I	ID段 busB数据冒险检测结果

功能

1. 存储所有需从 `ID` 段传递到 `EX` 段的控制信号与数据
2. 发生 `Load_use` 冒险时 将传递给 `EX` 段的所有控制信号和数据清零 从而实现插入气泡
3. 发生跳转指令预测错误时 将预先保存的 `pre_instruction` 的各控制信号与数据赋值给 `EX` 段 从而实现纠错

代码实现

```
1  module
   ID_EX(ID_PC,ID_op,ID_rs,ID_rt,ID_rd,ID_shamt,ID_func,ID_imm16,ID_target
   ,ID_busA,ID_busB,ID_RegDst,ID_RegWr,ID_ALUSrc,ID_MemWr,ID_MemtoReg,ID_B
   ranch,ID_Jump,ID_ExtOp,ID_ALUctr,
2  G,H,K,
3  clk,
4  EX_PC,EX_op,EX_rs,EX_rt,EX_rd,EX_shamt,EX_func,EX_imm16,EX_target,EX_bu
   sA,EX_busB,EX_RegDst,EX_RegWr,EX_ALUSrc,EX_MemWr,EX_MemtoReg,EX_Branch,
   EX_Jump,EX_ExtOp,EX_ALUctr,
5  EX_hi_num,EX_lo_num,EX_cs_num,
```

```
Load_use,
signal,pre_instruction,E,F,
A,B);
//ID_EX segment register

  input[31:2] ID_PC;
  input[5:0]  ID_op;
  input[4:0]  ID_rs,ID_rt,ID_rd,ID_shamt;
  input[5:0]  ID_func;
  input[15:0] ID_imm16;
  input[25:0] ID_target;
  input[31:0] ID_busA,ID_busB;
  input
ID_RegDst,ID_RegWr,ID_ALUSrc,ID_MemWr,ID_MemtoReg,ID_Branch,ID_Jump;
  input[1:0]  ID_ExtOp;
  input[4:0]  ID_ALUctr;
  input   clk;
  input   Load_use;
  input   signal;
  input[31:0] pre_instruction;
  input[31:0] E,F;
  input[31:0] G,H;
  input[31:0] A,B;
  input[31:0] K;

  output reg[31:2]  EX_PC;
  output reg[5:0]   EX_op;
  output reg[4:0]   EX_rs,EX_rt,EX_rd,EX_shamt;
  output reg[5:0]   EX_func;
  output reg[15:0]  EX_imm16;
  output reg[25:0]  EX_target;
  output reg[31:0]  EX_busA,EX_busB;
  output reg
EX_RegDst,EX_RegWr,EX_ALUSrc,EX_MemWr,EX_MemtoReg,EX_Branch,EX_Jump;
  output reg[1:0]   EX_ExtOp;
  output reg[4:0]   EX_ALUctr;
  output reg[31:0]  EX_hi_num,EX_lo_num;
  output reg[31:0]  EX_cs_num;

  wire[5:0] pre_op;
  wire[4:0] pre_rs,pre_rt,pre_rd,pre_shamt;
  wire[5:0] pre_func;
```

```
46    wire[15:0]  pre_imm16;
47    wire[25:0]  pre_target;
48    wire
pre_RegDst,pre_RegWr,pre_ALUSrc,pre_MemWr,pre_MemtoReg,pre_Branch,pre_J
ump;
49    wire[1:0] pre_ExtOp;
50    wire[4:0] pre_ALUctr;
51  //pre_instruction is the instruction which is deleted when branch
instruction appears.
52  //but this only works in test code without delay slot
53    assign  pre_op = pre_instruction[31:26];
54    assign  pre_rs = pre_instruction[25:21];
55    assign  pre_rt = pre_instruction[20:16];
56    assign  pre_rd = pre_instruction[15:11];
57    assign  pre_shamt = pre_instruction[10:6];
58    assign  pre_func = pre_instruction[5:0];
59    assign  pre_imm16 = pre_instruction[15:0];
60    assign  pre_target = pre_instruction[25:0];
61
62      ctrl
CPPL(pre_op,pre_rs,pre_rt,pre_rd,pre_shamt,pre_func,pre_RegDst,pre_RegW
r,pre_ALUSrc,pre_MemWr,pre_MemtoReg,pre_ExtOp,pre_ALUctr,pre_Branch,pre
_Jump);
63  //get relevant information of pre_instruction by ctrl
64
65    initial
66    begin
67      EX_PC = 0;
68      EX_op = 0;
69      EX_rs = 0;
70      EX_rt = 0;
71      EX_rd = 0;
72      EX_shamt = 0;
73      EX_func = 0;
74      EX_imm16 = 0;
75      EX_target = 0;
76      EX_busA = 0;
77      EX_busB = 0;
78      EX_RegDst = 0;
79      EX_RegWr = 0;
80      EX_ALUSrc = 0;
81      EX_MemWr = 0;
```

```
82        EX_MemtoReg = 0;
83        EX_Branch = 0;
84        EX_Jump = 0;
85        EX_ExtOp = 0;
86        EX_ALUctr = 0;
87        EX_hi_num = 0;
88        EX_lo_num = 0;
89        EX_cs_num = 0;
90
91      end
92    //initial assignment
93
94      always@(negedge clk)
95      begin
96        if(Load_use == 1)
97        begin
98        EX_PC <= 0;
99        EX_op <= 0;
100       EX_rs <= 0;
101       EX_rt <= 0;
102       EX_rd <= 0;
103       EX_shamt <= 0;
104       EX_func <= 0;
105       EX_imm16 <= 0;
106       EX_target <= 0;
107       EX_busA <= 0;
108       EX_busB <= 0;
109       EX_RegDst <= 0;
110       EX_RegWr <= 0;
111       EX_ALUSrc <= 0;
112       EX_MemWr <= 0;
113       EX_MemtoReg <= 0;
114       EX_Branch <= 0;
115       EX_Jump <= 0;
116       EX_ExtOp <= 0;
117       EX_ALUctr <= 0;
118       EX_hi_num <= 0;
119       EX_lo_num <= 0;
120       EX_cs_num <= 0;
121       end
122
123       else if(signal == 1) //跳转指令没有发生有效跳转
```

```
124         begin
125         EX_PC <= ID_PC + 1;//预测错误 此时EX段应修正执行之前保存的
    pre_instruction 其PC应为之前没变的ID_PC + 1
126         EX_op <= pre_op;
127         EX_rs <= pre_rs;
128         EX_rt <= pre_rt;
129         EX_rd <= pre_rd;
130         EX_shamt <= pre_shamt;
131         EX_func <= pre_func;
132         EX_imm16 <= pre_imm16;
133         EX_target <= pre_target;
134         EX_busA <= E;
135         EX_busB <= F;
136         EX_RegDst <= pre_RegDst;
137         EX_RegWr <= pre_RegWr;
138         EX_ALUSrc <= pre_ALUSrc;
139         EX_MemWr <= pre_MemWr;
140         EX_MemtoReg <= pre_MemtoReg;
141         EX_Branch <= pre_Branch;
142         EX_Jump <= pre_Jump;
143         EX_ExtOp <= pre_ExtOp;
144         EX_ALUctr <= pre_ALUctr;
145         EX_hi_num <= H;
146         EX_lo_num <= G;
147         EX_cs_num <= K;
148         end
149     //pre_instruction assignment
150 
151         else
152         begin
153         EX_PC <= ID_PC;
154         EX_op <= ID_op;
155         EX_rs <= ID_rs;
156         EX_rt <= ID_rt;
157         EX_rd <= ID_rd;
158         EX_shamt <= ID_shamt;
159         EX_func <= ID_func;
160         EX_imm16 <= ID_imm16;
161         EX_target <= ID_target;
162         EX_busA <= A;
163         EX_busB <= B;
164         EX_RegDst <= ID_RegDst;
```

```
165         EX_RegWr <= ID_RegWr;
166         EX_ALUSrc <= ID_ALUSrc;
167         EX_MemWr <= ID_MemWr;
168         EX_MemtoReg <= ID_MemtoReg;
169         EX_Branch <= ID_Branch;
170         EX_Jump <= ID_Jump;
171         EX_ExtOp <= ID_ExtOp;
172         EX_ALUctr <= ID_ALUctr;
173         EX_hi_num <= H;
174         EX_lo_num <= G;
175         EX_cs_num <= K;
176         end
177     // normal assignment with no Load_use and no branch instruction without
        really jump
178       end
179
180     endmodule
```

EX

基本描述

`EX` 流水段

模块接口

接口名	方向	描述
EX_PC	I	EX段PC的值
EX_op	I	EX段op的值
EX_rs	I	EX段rs的值
EX_rt	I	EX段rt的值
EX_rd	I	EX段rd的值
EX_shamt	I	EX段shamt的值
EX_func	I	EX段func的值
EX_imm16	I	EX段imme16的值
EX_target	I	EX段target的值
EX_busA	I	EX段busA的值
EX_busB	I	EX段busB的值
EX_RegDst	I	EX段RegDst的值
EX_RegWr	I	EX段RegWr的值
EX_ALUSrc	I	EX段ALUSrc的值
EX_MemWr	I	EX段MemW的值
EX_MemtoReg	I	EX段MemtoReg的值
EX_Branch	I	EX段Branch的值
EX_Jump	I	EX段Jump的值
EX_ExtOp	I	EX段ExtOp的值
EX_ALUctr	I	EX段ALUctr的值
alure	O	EX段alu的结果
EX_Reg	O	EX段根据EX_RegDst信号选择寄存器的编号
clk	I	时钟信号
Mem_alure	I	Mem段alu的结果
Wr_busW	I	Wr段busW的值
EX_MemRead	O	EX段MemRead的值，判断是否为lw指令

功能

1. 进行 `ALU` 运算
2. 用 `RegDst` 进行目标写寄存器的选择

代码实现

```
`include"extender.v"
`include"alu.v"
`include"MULT.v"
```

```
4   `include"mux2_5.v"
5
6   module
    EX(EX_PC,EX_op,EX_rs,EX_rt,EX_rd,EX_shamt,EX_func,EX_imm16,EX_target,EX_
    busA,EX_busB,
7   EX_RegDst,EX_RegWr,EX_ALUSrc,EX_MemWr,EX_MemtoReg,EX_Branch,EX_Jump,EX_E
    xtOp,EX_ALUctr,
8   alure,EX_Reg,clk,
9   Mem_alure,Wr_busW,
10  EX_MemRead,
11  EX_MULT_result,
12  EX_hi_num,EX_lo_num,
13  EX_cs_num);
14    input[31:2] EX_PC;
15    input[5:0]  EX_op;
16    input[4:0]  EX_rs,EX_rt,EX_rd,EX_shamt;
17    input[5:0]  EX_func;
18    input[15:0] EX_imm16;
19    input[25:0] EX_target;
20    input[31:0] EX_busA,EX_busB;
21    input
    EX_RegDst,EX_RegWr,EX_ALUSrc,EX_MemWr,EX_MemtoReg,EX_Branch,EX_Jump;
22    input[1:0]  EX_ExtOp;
23    input[4:0]  EX_ALUctr;
24
25    input   clk;
26
27    //input[1:0]  ALUSrcA,ALUSrcB;
28    input[31:0] Mem_alure,Wr_busW;
29    input[31:0] EX_hi_num,EX_lo_num;
30    input[31:0] EX_cs_num;
31
32    output[31:0]  alure;//alu result
33    output[4:0] EX_Reg;//rt or rd
34
35    output  EX_MemRead;
36    output[63:0]  EX_MULT_result;
37
38    assign  EX_MemRead = (EX_op == 6'b100011)? 1:0;
39
40    wire[31:0]  EX_imm32;
41    wire[31:0]  B;
```

```
42    wire[31:0]  G,H;
43    wire    zero;
44
45    extender extM(EX_imm16,EX_ExtOp,EX_imm32);//need
46    /*assign A = (ALUSrcA == 2'b00)? EX_busA :
47      (ALUSrcA == 2'b01)? Mem_alure :
48      (ALUSrcA == 2'b10)? Wr_busW : 0;
49    assign B = (ALUSrcB == 2'b00)? EX_busB :
50      (ALUSrcB == 2'b01)? Mem_alure :
51      (ALUSrcB == 2'b10)? Wr_busW :
52      (ALUSrcB == 2'b11)? EX_imm32 : 0;
53    assign C = (ALUSrcB == 2'b00)? EX_busB :
54      (ALUSrcB == 2'b01)? Mem_alure :
55      (ALUSrcB == 2'b10)? Wr_busW :
56      (ALUSrcB == 2'b11)? EX_busB : 0;*/
57
58    assign  B = (EX_ALUSrc == 1)? EX_imm32 : EX_busB;
59    alu
    AM(EX_ALUctr,EX_busA,B,EX_imm32,EX_shamt,alure,zero,EX_lo_num,EX_hi_num,
    EX_cs_num);
60  //alu calculation
61    MULT MU(EX_op,EX_busA,EX_busB,EX_rd,EX_shamt,EX_func,EX_MULT_result);
62  //MULT calculation
63    mux2_5 mux(EX_rt,EX_rd,EX_RegDst,EX_Reg);
64  //choose data
65
66  endmodule
```

EX_Mem

基本描述

`EX_Mem` 流水段寄存器

模块接口

接口名	方向	描述
EX_op	I	EX段op的值
EX_Reg	I	EX段Reg的值
EX_MemWr	I	EX段MemWr的值
EX_MemtoReg	I	EX段MemtoReg的值
EX_PC	I	EX段PC的值
EX_rs	I	EX段rs的值
EX_rt	I	EX段rt的值
EX_rd	I	EX段rd的值
EX_shamt	I	EX段shamt的值
EX_func	I	EX段func的值
EX_RegDst	I	EX段RegDst的值
EX_busA	I	EX段busA的值
EX_busB	I	EX段busB的值
alure	I	EX段alu的结果
clk	I	时钟信号
Mem_op	O	Mem段op的值
Mem_Reg	O	Mem段Reg的值
Mem_RegWr	O	Mem段RegWr的值
Mem_MemWr	O	Mem段MemWr的值
Mem_MemtoReg	O	Mem段MemtoReg的值
Mem_alure	O	Mem段ALU的结果
Mem_PC	O	Mem段PC的值
Mem_rs	O	Mem段rs的值
Mem_rt	O	Mem段rt的值
Mem_rd	O	Mem段rd的值
Mem_shamt	O	Mem段shamt的值
Mem_func	O	Mem段func的值
Mem_RegDst	O	Mem段RegDst的值
Mem_busA	O	Mem段busA的值
Mem_busB	O	Mem段busB的值

功能

将`EX`段的各控制信号与数据传递给`Mem`段

代码实现

```
module
EX_Mem(EX_op,EX_Reg,EX_RegWr,EX_MemWr,EX_MemtoReg,EX_PC,EX_rs,EX_rt,EX_r
d,EX_shamt,EX_func,EX_RegDst,EX_busA,EX_busB,alure,clk,
Mem_op,Mem_Reg,Mem_RegWr,Mem_MemWr,Mem_MemtoReg,Mem_alure,Mem_PC,Mem_rs,
Mem_rt,Mem_rd,Mem_shamt,Mem_func,Mem_RegDst,Mem_busA,Mem_busB,
EX_MULT_result,Mem_MULT_result);
//EX_Mem segment register
  input[5:0]  EX_op;
  input[4:0]  EX_Reg;
  input   EX_RegWr,EX_MemWr,EX_MemtoReg;
  input[31:0] alure;
  input   clk;

  input[31:2] EX_PC;
  input[4:0]  EX_rs,EX_rt,EX_rd,EX_shamt;
  input[5:0]  EX_func;
  input   EX_RegDst;
  input[31:0] EX_busA,EX_busB;

  input[63:0] EX_MULT_result;

  output reg[5:0] Mem_op;
  output reg[4:0] Mem_Reg;
  output reg    Mem_RegWr,Mem_MemWr,Mem_MemtoReg;
  output reg[31:0]  Mem_alure;
  output reg[31:2]  Mem_PC;
  output reg[4:0] Mem_rs,Mem_rt,Mem_rd,Mem_shamt;
  output reg[5:0] Mem_func;
  output reg  Mem_RegDst;
  output reg[31:0]  Mem_busA,Mem_busB;
  output reg[63:0]  Mem_MULT_result;
//assignment

  initial
  begin
    Mem_op = 0;
```

```
      Mem_Reg = 0;
      Mem_RegWr = 0;
      Mem_MemWr = 0;
      Mem_MemtoReg = 0;
      Mem_alure = 0;
      Mem_PC = 0;
      Mem_rs = 0;
      Mem_rt = 0;
      Mem_rd = 0;
      Mem_shamt = 0;
      Mem_func = 0;
      Mem_RegDst = 0;
      Mem_busA = 0;
      Mem_busB = 0;
      Mem_MULT_result = 0;
    end
  //initial assignment

    always@(negedge clk)
    begin
      Mem_op <= EX_op;
      Mem_Reg <= EX_Reg;
      Mem_RegWr <= EX_RegWr;
      Mem_MemWr <= EX_MemWr;
      Mem_MemtoReg <= EX_MemtoReg;
      Mem_alure <= alure;
      Mem_PC <= EX_PC;
      Mem_rs <= EX_rs;
      Mem_rt <= EX_rt;
      Mem_rd <= EX_rd;
      Mem_shamt <= EX_shamt;
      Mem_func <= EX_func;
      Mem_RegDst <= EX_RegDst;
      Mem_busA <= EX_busA;
      Mem_busB <= EX_busB;
      Mem_MULT_result <= EX_MULT_result;
    end
  //normal assignment
  endmodule
```

Mem

基本描述

`Mem` 流水段

模块接口

接口名	方向	描述
Mem_op	I	Mem段op的值
Mem_Reg	I	Mem段Reg的值
Mem_busB	I	Mem段busB的值
Mem_RegWr	I	Mem段RegWr的值
Mem_MemWr	I	Mem段MemWr的值
Mem_MemtoReg	I	Mem段MemtoReg的值
Mem_alure	I	Mem段alu的值
clk	I	时钟信号
Mem_dout	O	Mem段读取的主存结果

功能

1. 对 `store` 型指令 在该阶段将数据存入数据存储器中
2. 对 `load` 型指令 在该阶段从数据存储器中读取数据

代码实现

```
1  `include"dm.v"
2
3  module
   Mem(Mem_op,Mem_Reg,Mem_busB,Mem_RegWr,Mem_MemWr,Mem_MemtoReg,Mem_alure,c
   lk,Mem_dout);
4    input[5:0]  Mem_op;
5    input[4:0]  Mem_Reg;
6    input[31:0] Mem_busB;
7    input     Mem_RegWr,Mem_MemWr,Mem_MemtoReg;
8    input[31:0] Mem_alure;
9    input   clk;
10
11   output[31:0]  Mem_dout;
12
13   dm_4k dmT(Mem_op,Mem_alure,Mem_busB,Mem_dout,clk,Mem_MemWr);//data
   memory invoking
```

```
14    endmodule
```

Mem_Wr

基本描述

`Mem_Wr` 流水段寄存器

模块接口

接口名	方向	描述
Mem_op	I	Mem段op的值
Mem_Reg	I	Mem段Reg的值
Mem_RegWr	I	Mem段RegWr的值
Mem_MemtoReg	I	Mem段MemtoReg的值
Mem_alure	I	Mem段alu的结果
Mem_dout	I	Mem段主存的输出结果
Mem_PC	I	Mem段PC的值
Mem_rs	I	Mem段rs的值
Mem_rt	I	Mem段rt的值
Mem_rd	I	Mem段rd的值
Mem_shamt	I	Mem段shamt的值
Mem_func	I	Mem段func的值
Mem_RegDst	I	Mem段RegDst的值
Mem_busA	I	Mem段busA的值
Mem_busB	I	Mem段busB的值
clk	I	时钟信号
Wr_op	O	Wr段op的值
Wr_Reg	O	Wr段Reg的值
Wr_RegWr	O	Wr段RegWr的值
Wr_MemtoReg	O	Wr段MemtoReg的值
Wr_alure	O	Wr段alu的结果
Wr_dout	O	Wr段主存的输出结果
Wr_PC	O	Wr段PC的值
Wr_rs	O	Wr段rs的值
Wr_rt	O	Wr段rt的值
Wr_rd	O	Wr段rd的值
Wr_shamt	O	Wr段shamt的值
Wr_func	O	Wr段func的值
Wr_RegDst	O	Wr段RegDst的值
Wr_busA	O	Wr段busA的值
Wr_busB	O	Wr段busB的值

功能

将`Mem`段的各控制信号和数据传给`Wr`段

代码实现

```
module
Mem_Wr(Mem_op,Mem_Reg,Mem_RegWr,Mem_MemtoReg,Mem_alure,Mem_dout,Mem_PC,M
em_rs,Mem_rt,Mem_rd,Mem_shamt,Mem_func,Mem_RegDst,Mem_busA,Mem_busB,clk,
Wr_op,Wr_Reg,Wr_RegWr,Wr_MemtoReg,Wr_alure,Wr_dout,Wr_PC,Wr_rs,Wr_rt,Wr_
rd,Wr_shamt,Wr_func,Wr_RegDst,Wr_busA,Wr_busB,
Mem_MULT_result,Wr_MULT_result);
//Mem_Wr segement register
  input[5:0]  Mem_op;
  input[4:0]  Mem_Reg;
  input     Mem_RegWr,Mem_MemtoReg;
  input[31:0] Mem_alure;
  input[31:0] Mem_dout;
  input   clk;

  input[31:2] Mem_PC;
  input[4:0]  Mem_rs,Mem_rt,Mem_rd,Mem_shamt;
  input[5:0]  Mem_func;
  input Mem_RegDst;
  input[31:0] Mem_busA,Mem_busB;
  input[63:0] Mem_MULT_result;

  output reg[5:0] Wr_op;
  output reg[4:0] Wr_Reg;
  output reg  Wr_RegWr,Wr_MemtoReg;
  output reg[31:0]  Wr_alure;
  output reg[31:0]  Wr_dout;

  output reg[31:2]  Wr_PC;
  output reg[4:0] Wr_rs,Wr_rt,Wr_rd,Wr_shamt;
  output reg[5:0] Wr_func;
  output reg  Wr_RegDst;
  output reg[31:0]  Wr_busA,Wr_busB;
  output reg[63:0]  Wr_MULT_result;

  initial
  begin
```

```
      Wr_op = 0;
      Wr_Reg = 0;
      Wr_RegWr = 0;
      Wr_MemtoReg = 0;
      Wr_alure = 0;
      Wr_dout = 0;
      Wr_PC = 0;
      Wr_rs = 0;
      Wr_rt = 0;
      Wr_rd = 0;
      Wr_shamt = 0;
      Wr_func = 0;
      Wr_RegDst = 0;
      Wr_busA = 0;
      Wr_busB = 0;
      Wr_MULT_result = 0;
    end
  //initial assignment

    always@(negedge clk)
    begin
      Wr_op <= Mem_op;
      Wr_Reg <= Mem_Reg;
      Wr_RegWr <= Mem_RegWr;
      Wr_MemtoReg <= Mem_MemtoReg;
      Wr_alure <= Mem_alure;
      Wr_dout <= Mem_dout;
      Wr_PC <= Mem_PC;
      Wr_rs <= Mem_rs;
      Wr_rt <= Mem_rt;
      Wr_rd <= Mem_rd;
      Wr_shamt <= Mem_shamt;
      Wr_func <= Mem_func;
      Wr_RegDst <= Mem_RegDst;
      Wr_busA <= Mem_busA;
      Wr_busB <= Mem_busB;
      Wr_MULT_result <= Mem_MULT_result;
    end
  //normal assignment
  endmodule
```

Wr

基本描述

`Wr` 流水段

模块接口

接口名	方向	描述
Wr_op	I	Wr段op的值
Wr_Reg	I	Wr段Reg的值
Wr_MemtoReg	I	Wr段MemtoReg的值
Wr_alure	I	Wr段alu的结果
Wr_dout	I	Wr段主存的输出结果
Wr_PC	I	Wr段PC的值
Wr_rs	I	Wr段rs的值
Wr_rt	I	Wr段rt的值
Wr_rd	I	Wr段rd的值
Wr_shamt	I	Wr段shamt的值
Wr_func	I	Wr段func的值
Wr_RegDst	I	Wr段RegDst的值
Wr_busA	I	Wr段busA的值
Wr_busB	I	Wr段busB的值
clk	I	时钟信号
Wr_busW	O	Wr段存入寄存器的值

功能

在该阶段根据 `op` 和 `MemtoReg` 信号选择出正确的写入数据 并同控制信号一起传入设置在 `ID` 段的寄存器组 实现写回操作

代码实现

```
1  module
   Wr(Wr_op,Wr_Reg,Wr_RegWr,Wr_MemtoReg,Wr_alure,Wr_dout,Wr_PC,Wr_rs,Wr_rt,
   Wr_rd,Wr_shamt,Wr_func,Wr_RegDst,Wr_busA,Wr_busB,clk,Wr_busW);//change
2    input[5:0]  Wr_op;
3    input[4:0]  Wr_Reg;
4    input   Wr_RegWr,Wr_MemtoReg;
5    input[31:0] Wr_alure;
```

```
    input[31:0] Wr_dout;
    input[31:2] Wr_PC;
    input[4:0]  Wr_rs,Wr_rd,Wr_rt,Wr_shamt;
    input[5:0]  Wr_func;
    input   Wr_RegDst;
    input[31:0] Wr_busA,Wr_busB;

    input clk;

    output reg[31:0]  Wr_busW;
  //statement

    always@(*)
    begin
      if(Wr_op == 6'b100000)
      begin
        Wr_busW<={{24{Wr_dout[7]}},Wr_dout[7:0]};
      end
  //this is for instruction 'lb'-load byte

      else if(Wr_op == 6'b100100)
      begin
        Wr_busW<={{24'b0},Wr_dout[7:0]};
      end
  //this is for instruction 'lbu'-load byte unsigned

      else
      begin
        if(Wr_MemtoReg == 0)  Wr_busW <= Wr_alure;
        else  Wr_busW <= Wr_dout;
        //mux2_32 muxW(Wr_alure,Wr_dout,Wr_MemtoReg,Wr_busW);
      end
  //choose data to assign Wr_busW by Wr_MemtoReg
    end
  endmodule
```

2. 控制器

conrtol

基本描述

控制器模块 接收指令的 `op` 和 `func` 域信号 对指令译码 产生该指令对应的控制信号

模块接口

接口名	方向	描述
op	I	指令操作码
rs	I	rs寄存器地址
rt	I	rt寄存器地址
rd	I	rd寄存器地址
shamt	I	偏移量
func	I	指令5-0位
RegDst	O	rt,rd寄存器选择控制信号
RegWr	O	寄存器写控制信号
ALUSrc	O	ALU数据来源控制信号
MemWr	O	数据内存写控制信号
MemtoReg	O	数据内存数据写入寄存器控制信号
ExtOp	O	立即数扩展控制信号
ALUctr	O	ALU控制信号
Branch	O	条件跳转指令控制信号
Jump	O	J型指令控制信号

功能

1. 当 `op` 不为0 即指令不为R型指令时 根据 `op` 对指令进行译码 将各信号输送给对应的控制信号
2. 当 `op` 的值为0 即指令为R型指令时 根据 `func` 对指令进行译码 将各信号书送给对应的控制信号

代码实现

```
1  module
   ctrl(op,rs,rt,rd,shamt,func,RegDst,RegWr,ALUSrc,MemWr,MemtoReg,ExtOp,ALU
   ctr,Branch,Jump);
2    input[5:0]  op;
3    input[4:0]  rs,rt,rd,shamt;
4    input[5:0]  func;
5    output    RegDst;
6    output    RegWr;
7    output    ALUSrc;
```

```
8     output    MemWr;
9     output    MemtoReg;
10    output[1:0] ExtOp;
11    output[4:0] ALUctr;
12    output    Branch;
13    output    Jump;
14
15    reg[13:0] controls;
16
17    assign{RegDst,RegWr,ALUSrc,MemWr,MemtoReg,ExtOp,ALUctr,Branch,Jump} =
    controls;
18
19    always@(*)
20      case(op)
21        6'b000000:
22          begin
23            case(func)
24              6'b100001: controls <= 14'b11000000000000;//addu
25              6'b100011: controls <= 14'b11000000000100;//subu
26              6'b101010: controls <= 14'b11000000001000;//slt
27              6'b100100: controls <= 14'b11000000001100;//and
28              6'b100111: controls <= 14'b11000000010000;//nor
29              6'b100101: controls <= 14'b11000000010100;//or
30              6'b100110: controls <= 14'b11000000011000;//xor
31              6'b000000: controls <= 14'b11000000011100;//sll
32              6'b000010: controls <= 14'b11000000100000;//srl
33              6'b101011: controls <= 14'b11000000100100;//sltu
34              6'b001001: controls <= 14'b11000000000001;//jalr
35              6'b001000: controls <= 14'b10000000000001;//jr
36              6'b000100: controls <= 14'b11000000110000;//sllv
37              6'b000011: controls <= 14'b11000000110100;//sra
38              6'b000111: controls <= 14'b11000000111000;//srav
39              6'b000110: controls <= 14'b11000000111100;//srlv
40              6'b011000: controls <= 14'b00000000000000;//mult
41              6'b010010: controls <= 14'b11000001011100;//mflo
42              6'b010000: controls <= 14'b11000001100000;//mfhi
43              6'b010011: controls <= 14'b00000000000000;//mtlo
44              6'b010001: controls <= 14'b00000000000000;//mthi
45              6'b001100: controls <= 14'b00000000000000;//syscall
46            endcase
47          end
48        6'b001001: controls <= 14'b01100011000000;//addiu
```

```
        6'b000100: controls <= 14'b00000000000110;//beq
        6'b000101: controls <= 14'b00000000000110;//bne
        6'b100011: controls <= 14'b01101011000000;//lw
        6'b101011: controls <= 14'b00110011000000;//sw
        6'b001111: controls <= 14'b01100101011000;//lui
        6'b001010: controls <= 14'b01100011000100;//slti
        6'b001011: controls <= 14'b01100011001000;//sltiu
        6'b000001: controls <= 14'b00000000000010;//bgez,bltz

        6'b000111: controls <= 14'b00000000000010;//bgtz
        6'b000110: controls <= 14'b00000000000010;//blez
        6'b100000: controls <= 14'b01101011000000;//lb
        6'b100100: controls <= 14'b01101011000000;//lbu
        6'b101000: controls <= 14'b00110011000000;//sb
        6'b001100: controls <= 14'b01100001001100;//andi
        6'b001101: controls <= 14'b01100001010000;//ori
        6'b001110: controls <= 14'b01100001010100;//xori


        6'b000010: controls <= 14'b00000000000001;//j
        6'b000011: controls <= 14'b01000000000001;//jal

        6'b010000:
        begin
          case(rs)
          5'b00000: controls <= 14'b01000001100100;//mfc0
          5'b00100: controls <= 14'b00000000000000;//mtc0
          5'b10000: controls <= 14'b00000000000000;//eret
          endcase
        end

        default : controls <= 14'b00000000000000;
      endcase
  //assign 'controls' according to instructions
  endmodule
```

（附）控制信号真值表

	RegDst	RegWr	ALUSrc	MemWr	MemtoReg	ExtOp	ALUctr	Branch	Jump	R/I/J	31-26(op)	25-21	20-16	15_11	10_6	5-0(func)
addu	1	1	0	0	0	xx	0 0000	0	0	R	0 00000	rs	rt	rd	0 0000	100001
subu	1	1	0	0	0	xx	0 0001	0	0	R	0 00000	rs	rt	rd	0 0000	100011
slt	1	1	0	0	0	xx	0 0010	0	0	R	0 00000	rs	rt	rd	0 0000	101010
and	1	1	0	0	0	xx	0 0011	0	0	R	0 00000	rs	rt	rd	0 0000	100100
nor	1	1	0	0	0	xx	0 0100	0	0	R	0 00000	rs	rt	rd	0 0000	100111
or	1	1	0	0	0	xx	0 0101	0	0	R	0 00000	rs	rt	rd	0 0000	100101
xor	1	1	0	0	0	xx	0 0110	0	0	R	0 00000	rs	rt	rd	0 0000	100110
sll	1	1	0	0	0	xx	0 0111	0	0	R	0 00000	0 0000	rt	rd	shf	0 00000
srl	1	1	0	0	0	xx	0 1000	0	0	R	0 00000	0 0000	rt	rd	shf	0 00010
addiu	0	1	1	0	0	0 1	1 0000	0	0	I	0 01001	rs	rt	imm	imm	imm
beq	x	0	0	0	x	xx	0 0001	1	0	I	0 00100	rs	rt	offset	offset	offset
bne	x	0	0	0	x	xx	0 0001	1	0	I	0 00101	rs	rt	offset	offset	offset
lw	0	1	1	0	1	0 1	1 0000	0	0	I	1 00011	base	rt	offset	offset	offset
sw	x	0	1	1	x	0 1	1 0000	0	0	I	1 01011	base	rt	offset	offset	offset
lui	0	1	1	0	0	1 0	x	0	0	I	0 01111	0 0000	rt	imm	imm	imm
j	x	0	x	0	x	x x	x	0	1	J	0 00010	target	target	target	target	target
sltu	1	1	0	0	0	xx	0 1001	0	0	R	0 00000	rs	rt	rd	0 0000	101011
jalr	1	1	0	0	0	xx	x	0	0	R	0 00000	rs	0 0000	1 1111	0 0000	0 01001
jr	1	0	0	0	0	xx	x	0	0	R	0 00000	rs	0 0000	0 0000	0 0000	0 01000
sllv	1	1	0	0	0	xx	0 1100	0	0	R	0 00000	rs	rt	rd	0 0000	0 00100
sra	1	1	0	0	0	xx	0 1101	0	0	R	0 00000	0 0000	rt	rd	shf	0 00011
srav	1	1	0	0	0	xx	0 1110	0	0	R	0 00000	rs	rt	rd	0 0000	0 00111
srlv	1	1	0	0	0	xx	0 1111	0	0	R	0 00000	rs	rt	rd	0 0000	0 00110
slti	0	1	1	0	0	0 1	1 0001	0	0	I	0 01010	rs	rt	imm	imm	imm
sltiu	0	1	1	0	0	0 1	1 0010	0	0	I	0 01011	rs	rt	imm	imm	imm
bgez	x	0	0	0	x	xx	x	1	0	I	0 00001	rs	0 0001	offset	offset	offset
bgtz	x	0	0	0	x	xx	x	1	0	I	0 00111	rs	0 0000	offset	offset	offset
blez	x	0	0	0	x	xx	x	1	0	I	0 00110	rs	0 0000	offset	offset	offset
bltz	x	0	0	0	x	xx	x	1	0	I	0 00001	rs	0 0000	offset	offset	offset
lb	0	1	1	0	1	0 1	1 0000	0	0	I	1 00000	base	rt	offset	offset	offset
lbu	0	1	1	0	1	0 1	1 0000	0	0	I	1 00100	base	rt	offset	offset	offset
sb	0	0	1	1	0	0 1	1 0000	0	0	I	1 01000	base	rt	offset	offset	offset
andi	0	1	1	0	0	0 0	1 0011	0	0	I	0 01100	rs	rt	imm	imm	imm
ori	0	1	1	0	0	0 0	1 0100	0	0	I	0 01101	rs	rt	imm	imm	imm
xori	0	1	1	0	0	0 0	1 0101	0	0	I	0 01110	rs	rt	imm	imm	imm
jal	x	1	x	0	0	xx	x	0	1	J	0 00011	target	target	target	target	target

3. 冒险处理

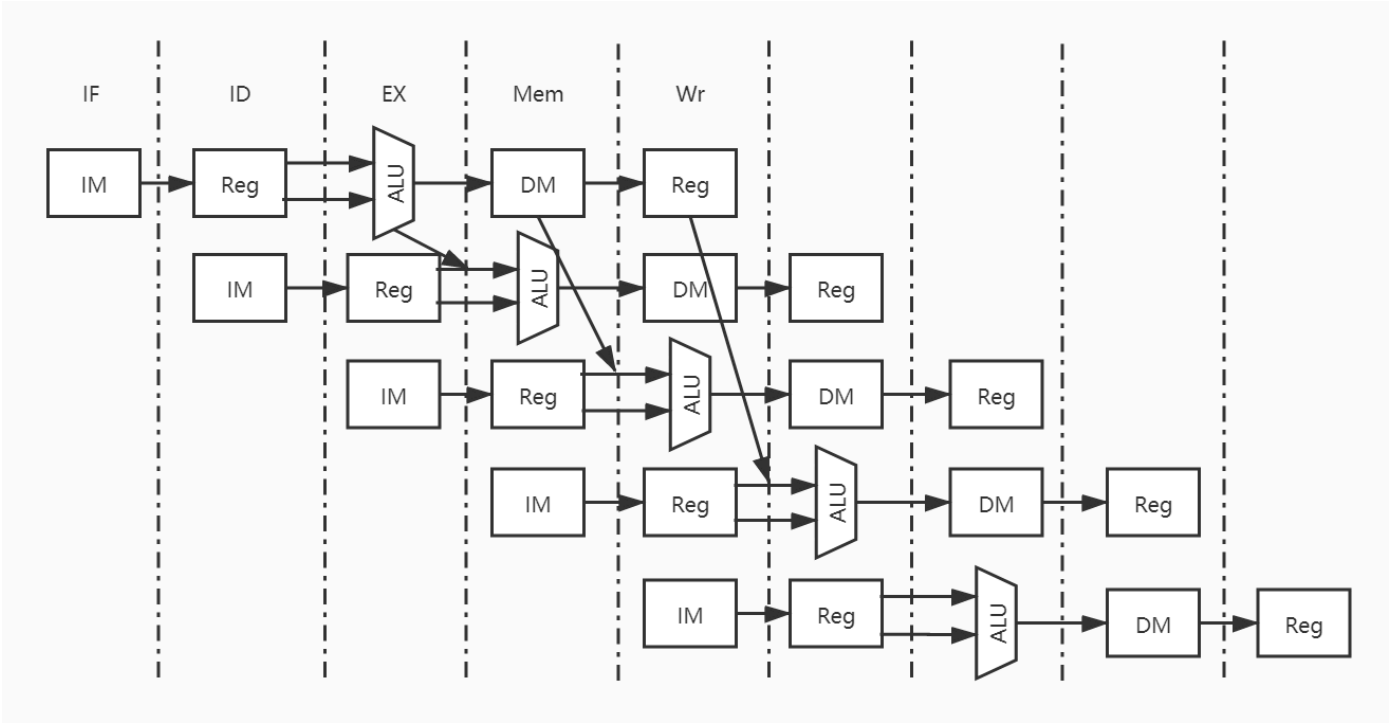
冒险分析：

数据冒险

分为 `Pre_Data_Hazard, Branch_Data_Hazard` 这两个模块处理，其基本思想一致：处理从寄存器中读数存在的延迟问题。在ID段进行检测，处理在EX段，Mem段，Wr段可能出现的寄存器数据修改问题。取到的寄存器数应为最近一次修改的值。最后将对应冒险标记变量进行冒险记录并输出

结构冒险

1. 将指令存储器与数据存储器分成两个独立的模块
2. 同一时间段寄存器的读与写相冲突，依然使用转发技术，将写操作中ALU已经产生的结果送到读操作ALU的输入端。



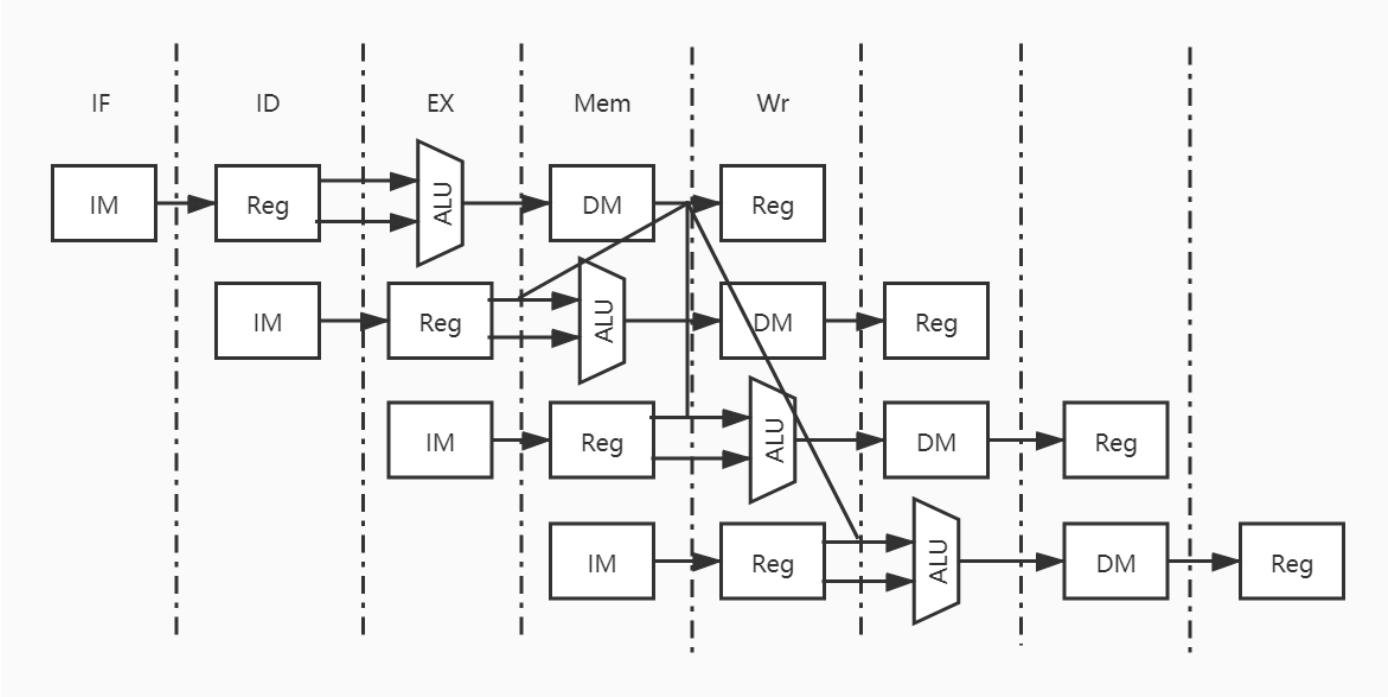
> 数据冒险与结构冒险处理

Load_use冒险

主要靠 `Load_use` 信号进行判断，在 `Load_use` 模块中进行，若出现 `Load_use` 冒险，则 `Load_use` 信号为1，进行如下操作：

1. 将 `ID_EX` 流水段的控制信号清零
2. 保持 `IF_ID` 流水段的指令不变
3. 保持 `PC` 的值不变

通过以上三个操作实现一个周期的阻塞 从而可以通过前面的数据冒险处理方式对 `Load_use` 冒险进行处理



> `Load_use` 数据冒险

控制冒险

采用静态预测，始终预测指令满足跳转。在IF_ID段寄存器中进行判断和处理，若为分支指令或跳转指令，则将IF段的指令记录为pre_instruction，并将ID段指令赋值为0以实现对下一条指令的清零，PC不变，若跳转（预测成功），则正常执行，若不跳转（预测失败），则ID_EX段寄存器中，将之前保存的pre_instruction对应的控制信号与数据赋给EX段的控制信号与数据，实现纠错。

Pre_Data_Hazard

基本描述

处理数据冒险的模块

模块接口

接口名	方向	描述
Mem_RegWr	I	Mem段RegWr的值
Mem_Reg	I	Mem段Reg的值
ID_rs	I	ID段rs的值
ID_rt	I	ID段rt的值
EX_RegWr	I	EX段RegWr的值
EX_Reg	I	EX段Reg的值
Wr_RegWr	I	Wr段RegWr的值
Wr_Reg	I	Wr段Reg的值
ALUSrcC	O	数据冒险选择信号
ALUSrcD	O	数据冒险选择信号

功能

判断相隔1条 2条或3条的数据冒险 并根据不同情况对选择信号 `ALUSrcC` 和 `ALUSrcD` 进行赋值 具体的对应关系可通过代码及注释查阅

代码实现

```
module Pre_Data_Hazard(Mem_RegWr,Mem_Reg,ID_rs,ID_rt,EX_RegWr,EX_Reg,Wr_RegWr,Wr_Reg,ALUSrcC,ALUSrcD);
  input Mem_RegWr,EX_RegWr,Wr_RegWr;
  input[4:0]  Mem_Reg,EX_Reg,Wr_Reg,ID_rs,ID_rt;
//parameters which is needed in this module

  wire  C1C,C1D,C2C,C2D,C3C,C3D;
//intermediate variables

  assign  C1C = EX_RegWr && (EX_Reg != 0) && (EX_Reg == ID_rs);
  assign  C1D = EX_RegWr && (EX_Reg != 0) && (EX_Reg == ID_rt);
  assign  C2C = Mem_RegWr && (Mem_Reg != 0) && ((EX_Reg != ID_rs)||(EX_Reg == ID_rs && EX_RegWr == 0)) && (Mem_Reg == ID_rs);
  assign  C2D = Mem_RegWr && (Mem_Reg != 0) && ((EX_Reg != ID_rt)||(EX_Reg == ID_rt && EX_RegWr == 0)) && (Mem_Reg == ID_rt);
  assign  C3C = Wr_RegWr && (Wr_Reg != 0) && ((EX_Reg != ID_rs)||(EX_Reg == ID_rs && EX_RegWr == 0)) && ((Mem_Reg != ID_rs)||(Mem_Reg == ID_rs && Mem_RegWr == 0)) && (Wr_Reg == ID_rs);
```

```
14    assign  C3D = Wr_RegWr && (Wr_Reg != 0) && ((EX_Reg != ID_rt)||(EX_Reg
      == ID_rt && EX_RegWr == 0)) && ((Mem_Reg != ID_rt)||(Mem_Reg == ID_rt &&
      Mem_RegWr == 0)) && (Wr_Reg == ID_rt);
15    //definitions
16
17    output reg[1:0] ALUSrcC,ALUSrcD;
18    //reg-type output
19
20    initial
21    begin
22      ALUSrcC = 0;
23      ALUSrcD = 0;
24    end
25    //initial assignment
26
27    always@(C1C or C2C or C3C)
28    begin
29      if(C1C == 1)  ALUSrcC <= 2'b01;//RS EX-segment data hazard
30      else if(C2C == 1) ALUSrcC <= 2'b10;//RS Mem-segment data hazard
31      else if(C3C == 1) ALUSrcC <= 2'b11;//RS Wr-segment data hazard
32      else  ALUSrcC <= 2'b00;//no data hazard
33    end
34
35    always@(C1D or C2D or C3D)
36    begin
37      if(C1D == 1)  ALUSrcD <= 2'b01;//RT EX-segment data hazard
38      else if(C2D == 1) ALUSrcD <= 2'b10;//RT Mem-segment data hazard
39      else if(C3D == 1) ALUSrcD <= 2'b11;//RT Wr-segment data hazard
40      else  ALUSrcD <= 2'b00;//no data hazard
41    end
42
43  endmodule
```

branch_Data_Hazard

基本描述

出现分支指令时 对预先保存的 `pre_instruction` 进行数据冒险判断的模块

模块接口

接口名	方向	描述
Wr_RegWr	I	Wr段RegWr的值
Wr_Reg	I	Wr段Reg的值
Pre_rs	I	Preinstruction的rs的值
Pre_rt	I	Preinstruction的rt的值
Mem_RegWr	I	Mem段RegWr的值
Mem_Reg	I	Mem段Reg的值
ALUSrcE	O	数据冒险检测信号
ALUSrcF	O	数据冒险检测信号

功能

判断 `pre_instruction` 的数据冒险 并根据不同情况对选择信号 `ALUSrcC` 和 `ALUSrcD` 进行赋值
具体的对应关系可通过代码及注释查阅

代码实现

```
1  module
   branch_Data_Hazard(Wr_RegWr,Wr_Reg,pre_rs,pre_rt,Mem_RegWr,Mem_Reg,ALUSr
   cE,ALUSrcF);
2    input Wr_RegWr,Mem_RegWr;
3    input[4:0]  Wr_Reg,Mem_Reg,pre_rs,pre_rt;
4
5    wire  C2E,C2F;
6
7    assign  C2E = Wr_RegWr && (Wr_Reg≠0) && ((Mem_Reg ≠ pre_rs) ||
   (Mem_Reg == pre_rs && Mem_RegWr == 0)) && (Wr_Reg == pre_rs);
8    assign  C2F = Wr_RegWr && (Wr_Reg≠0) && ((Mem_Reg ≠ pre_rt) ||
   (Mem_Reg == pre_rt && Mem_RegWr == 0)) && (Wr_Reg == pre_rt);
9
10   output reg[1:0] ALUSrcE,ALUSrcF;
11
12   initial
13   begin
14     ALUSrcE = 0;
15     ALUSrcF = 0;
16   end
17
18   always@(C2E)
```

```
    begin
      if(C2E == 1)  ALUSrcE <= 2'b10;
      else    ALUSrcE <= 2'b00;
    end

    always@(C2F)
    begin
      if(C2F == 1)  ALUSrcF <= 2'b10;
      else    ALUSrcF <= 2'b00;
    end

  endmodule
```

Load_use

基本描述

判断 `Load_use` 数据冒险的模块

模块接口

接口名	方向	描述
EX_MemRead	I	EX段检测是否为lw型指令
EX_rt	I	EX段rt的值
ID_rs	I	ID段rs的值
ID_rt	I	ID段rt的值
Load_use	O	Load_use冒险检测信号

功能

检测 `Load_use` 数据冒险 产生判断信号以供后续使用

代码实现

```
module Load_use(EX_MemRead,EX_rt,ID_rs,ID_rt,Load_use);
//Load_use hazard judge
  input EX_MemRead;
  input[4:0]  EX_rt,ID_rs,ID_rt;
  output reg  Load_use;
  wire  C;

```

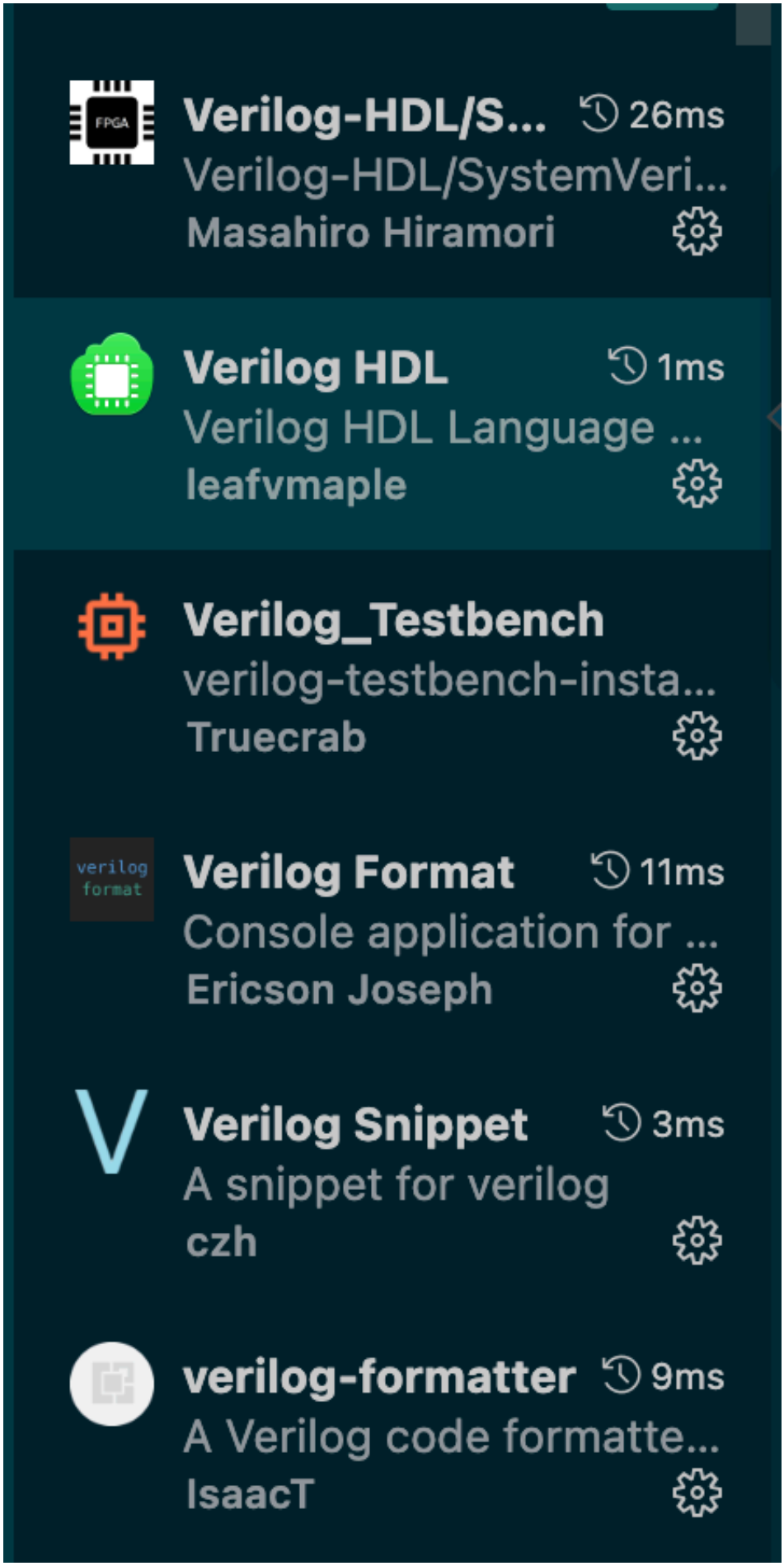
```
8     assign  C = EX_MemRead && ((EX_rt == ID_rs) || (EX_rt == ID_rt));
9
10    initial
11    begin
12      Load_use = 0;
13    end
14
15    always@(C)
16    begin
17      Load_use = C;
18    end
19
20  endmodule
```

三. 测试代码及方法

1. 环境配置

由于我的电脑是MacOS系统 无法安装modelsim进行仿真 故根据网上共享的配置方案 用vscode + 插件 + gtkwave搭配的方法对代码进行仿真与运行 具体来说 执行如下：

- 在vscode中安装如下插件


Verilog-HDL/S... 26ms
Verilog-HDL/SystemVeri...
Masahiro Hiramori
Verilog HDL 1ms
Verilog HDL Language ...
leafvmaple
Verilog_Testbench
verilog-testbench-insta...
Truecrab
verilog format
Verilog Format 11ms
Console application for ...
Ericson Joseph
Verilog Snippet 3ms
A snippet for verilog
czh
verilog-formatter 9ms
A Verilog code formatte...
IsaacT

> 安装的插件列表

- 通过Homebrew 在终端中安装如下软件

  1. 安装Iverilog

  ```
  % brew install icarus-verilog
  ```

  2. 安装verilator

  ```
  % brew install verilator
  ```

  3. 安装gtkwave（波形图展示工具）

  ```
  % brew cask install xquartz
  % brew cask insatll gtkwave
  ```

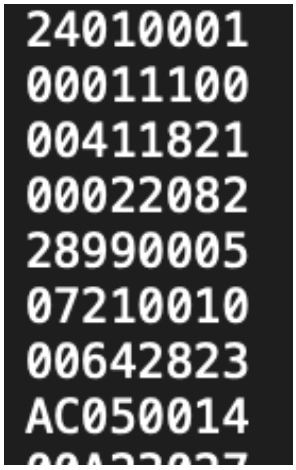
> 通过Homebrew安装的软件列表

上述配置完成后 可以在vscode中对 `Verilog` 代码进行编译运行 并使用gtkwave或vscode对波形进行仿真

2. 测试代码

来自老师发送的测试代码文件 `《36条指令-对应的汇编代码+指令字等-红色字新加说明》`

将测试代码的以十六进制存储在 `code.txt` 中效果如下：




00A23027
00C33825
00E64026
AC08001C
11030002
00C7482A
24010008
8C2A0014
15450004
00415824
AC2B001C
AC240010
0C000019
3C0C000C
004CD007
003AD804
0360F809
A07A0005
0063682B
1DA00003
00867004
000E7883
002F8006
1A000008
002F8007
240B008C
06000006
8D5C0003
179D0007
A0AF0008
80B20008
90B30008
2DF8FFFF
0185E825
01600008
31F4FFFF
35E5FFFF

```
39F6FFFF
08000000
```

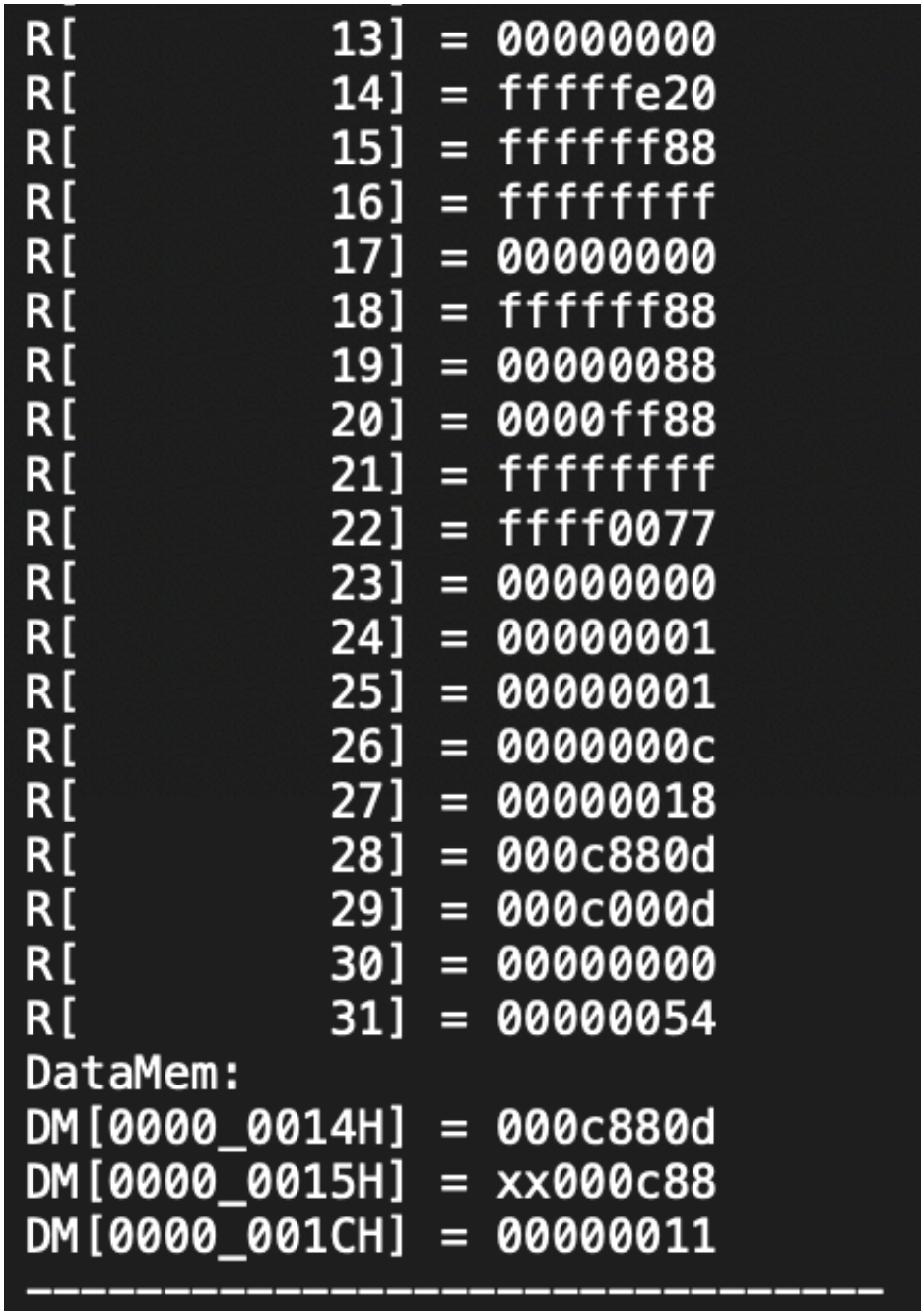
> 测试代码（十六进制）

3. 测试结果

由于 `gtkwave` 虽能仿真波形 但无法监测到模块内部定义的变量信号（如寄存器组中每个寄存器的值 数据存储器中每块内存的值）所以采用每一周期将寄存器值与数据存储器中的数据输出到 `out.txt` 文件的形式对数据进行侦测 从而判断指令执行的正确性

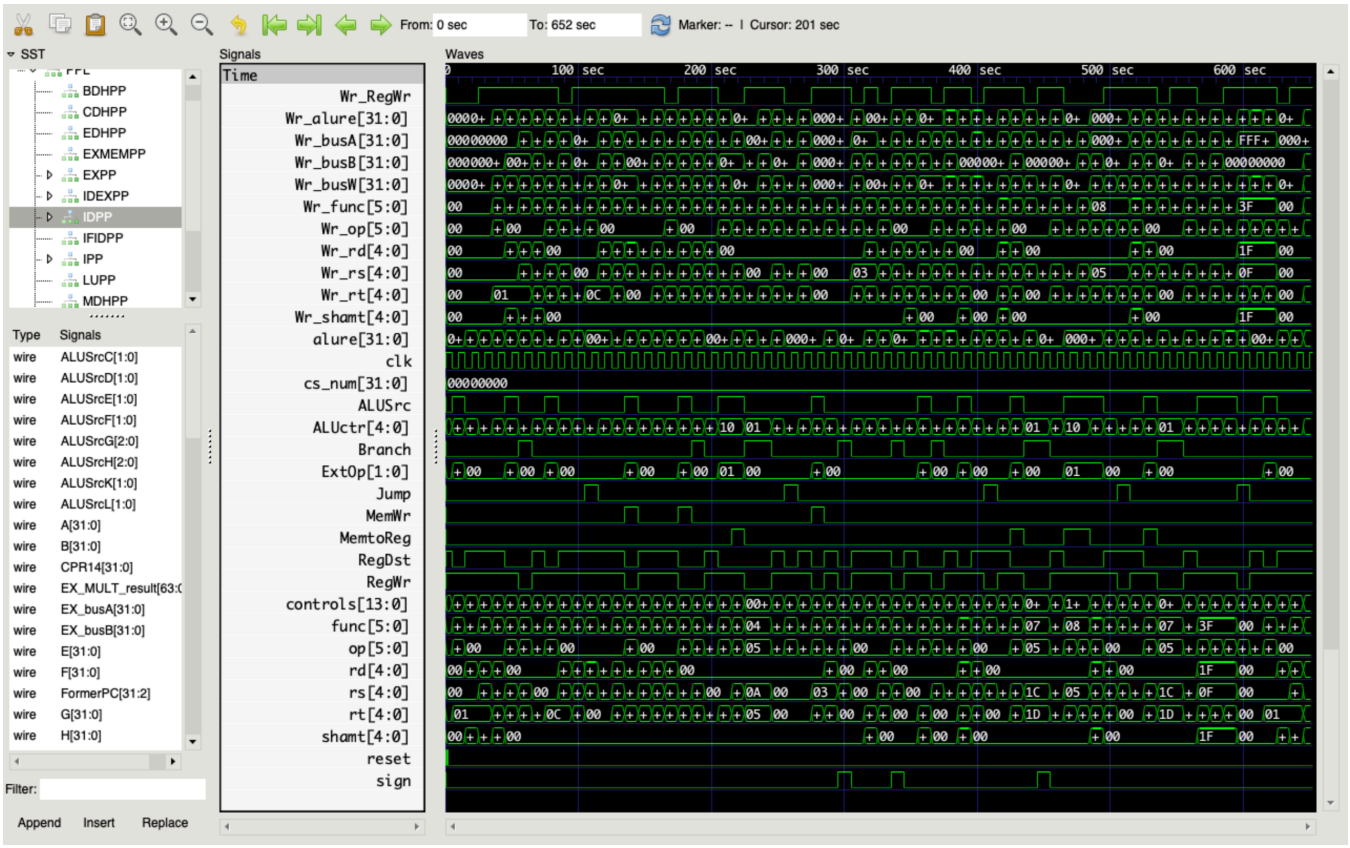
- 数据结果（存储在 `out.txt` 中）

```
--------------------------------------
clock [          66 pos]
instruction = 28990005
ID_instruction = 00022082
PC = 00000010
IF_PC = 00000010
ID_PC = 0000000c
EX_PC = 00000008
Mem_PC = 00000004
Wr_PC = 00000000
Registers:
R[           0] = 00000000
R[           1] = 00000008
R[           2] = 00000010
R[           3] = 00000011
R[           4] = 00000004
R[           5] = 0000000d
R[           6] = fffffffe2
R[           7] = fffffff3
R[           8] = 00000011
R[           9] = 00000000
R[          10] = 00000011
R[          11] = 0000008c
R[          12] = 000c0000
```



> 此处展示的是最后一周期所有代码执行完毕之后的结果

- 波形仿真结果（存储在`pipeline_cpu_wave.vcd`中）



> 由于数据结果另做处理 此处只展示控制信号的仿真结果

请老师检查时注意 由于数据存储器 `dm` 中的各数据没有初始化为0 所以在查看与数据存储器相关的波形时可能会出现红线 但并不是错误!

四. 总结与展望

在完成单周期CPU的编写后，我曾一度以为流水线CPU设计非常的简单，可是现实却击垮了我。相对于流水线CPU而言，单周期CPU就是简简单单的几个模块组合在一起，而流水线所要考虑的并行性、冒险性才是更为折磨的考验。仅仅掌握书本上流水线思想、流水线数据通路、流水线冒险判断与处理的知识，对于编写流水线CPU是完全不够的，需要经过很多次的实践才能深入理解流水线的执行。于卷考而言，通过流水线CPU的编写，使得我对卷考知识的掌握更加透彻，这也节省了我后期不少时间。

在编写相关代码时，我是先进行36条的编写并调试。而36条的调试过程是最痛苦的时间段。当时对流水线相关知识其实还是半知不解，就是按照书上和PPT上的思路搬过来，没有什么自己的思考和理解，而在调试的过程中，我深深地体会到了这样的不可行性。主要体现在我对错误发生源的寻找困难以及修改错误的编写困难。三天时间，一直都蹲在电脑前，波形不对——寻找错误——修改错误，一直这样循环往复，但在后期对冒险处理的理解更加深刻之后，难度减少了很多。我觉得最困难的地方应该是控制冒险的处理了，但这也是我看到自己思考的闪光点的一部分。我使用的方法是

插入空指令，始终预测其会跳转，但这就会影响不跳转的情况，所以我选择将IF段的instruction存储在pre_instruction中，并且添加signal判断分支指令是否跳转，若不跳转，则选择将pre_instruction中的数据存储到EX阶段。这个的调试用了相当长的时间，但最终成功了。

在扩展45条指令时，遇到的困难要比之前少很多，虽然多了HI,LO寄存器和协处理器，但我们接触到的形式其实和之前写的RegFile寄存器十分相似，以近似的方法进行编写即可，处理数据冒险的方式也基本相同。有了36条作为基础，45条的编写和调试都较快，但最终由于复习时间不够与ddl的问题，没能在截止时间前完成剩余9条的调试。

在这次的课程设计中，我通过对流水线CPU编写，从CPU的角度理解了计算机的组成，对计算机的体系结构有了一个初步的了解。感谢那个迎难而上、勇于实践的自己，更感谢身边优秀、贴心的朋友、老师。