流水线MIPS CPU设计

实验目的

设计多周期CPU，实现如下功能：

* 实现流水线
* 实现转发和插入气泡
* 实现延迟分支
* 实现中断
* 扩充指令

实验要求：

基础指令如下：

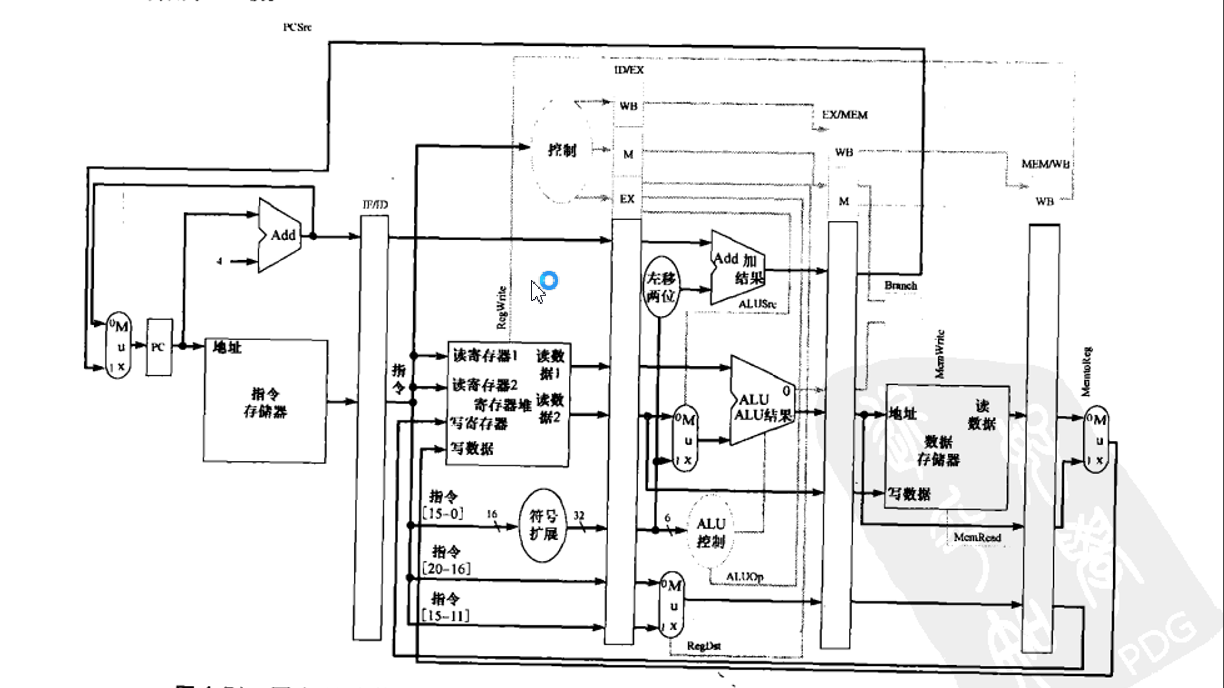
* 基本16条指令：
  + add addi addu sub subu
  + and andi or nor xor
  + bgtz bne j jr
  + lw sw

对于自行添加的指令，自己编写汇编代码，保证仿真正确

实验设计：

1. 将CPU指令运行的五个阶段作为五个独立的模块，因为每个时钟周期每个模块独立的处理不同的指令；
2. 将四个流水线寄存器设置成独立的模块
3. 在ID阶段实现译码，在EX阶段实现终端信号的检测
4. 在适当的模块如ID或EX实现前推和气泡插入检测

实验的基本电路图如图所示：



下面分别论述各个模块的功能：

IF模块：

* 实现PC的更新和PCplus4的输出，在ID模块下例化IM模块，取指令，输入IF\_ID流水线寄存器。
* PC的来源有三个：正常PC自增得到的值；因为跳转指令如beq或j指令变更的值；因为中断PC的跳转

ID模块：

* 接受来自control模块的控制信号
* 译码，输出寄存器号，立即数等
* 例化寄存器组模块，根据寄存器号取出对应的数据

control模块：

* 输入指令的操作码和功能码，根据不同的指令确定控制信号，未知指令实现终端，将控制信号输出到ID模块和ID\_EX寄存器

EX模块：

* 前推生效的地方，根据前推信号选择alu的源操作数来源
* 例化了alu模块，计算并生成溢出、除零异常等信号，以及为方便跳转指令判断的ZF和SF信号
* J指令跳转地址生成

MEM模块：

* 前推信号ForwardC生效的地方，选择写入DM的数据
  + 如果sw前一条指令是ALU指令，转发alu\_out
  + 如果前一条指令是lw指令，转发r1\_dout
  + 否则按照当前写入数据
* 负责写入和读出DM的数据

WB模块：

* 将寄存器写地址写数据等发送给ID模块

Transmit模块：

根据来自各个流水线寄存器的使能值和寄存器号，判断转发的必要性并生成前推信号

在以下情况进行转发：

* + 前一条指令的写寄存器和本指令的读寄存器相同，需要转发这周期结束生成的alu\_out作为源操作数
  + 前两条指令的写寄存器和本指令的读寄存器相同，且前一条指令的写寄存器和本指令的读寄存器不同，需要转发上周期结束生成的alu\_out作为源操作数
  + 前一条指令是lw或ALU指令，本条指令是sw指令，前一条指令的写寄存器与sw的baseR或rt相同。

Bubble模块：

根据来自流水线就餐器的使能值和寄存器号，判断插入气泡的必要性并生成流水线寄存器的清空信号和阻塞信号

在下列情况下需要阻塞流水线：

* + 有跳转指令需要清空流水线
  + 有lw+ALU指令
  + 有中断产生

IF\_ID ID\_EX EX\_MEM都有控制信号控制清空寄存器

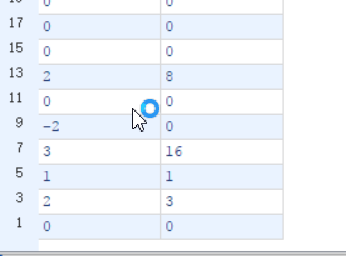
Fault模块：

* 根据来自不同模块的异常发生信号和断点地址，将cause寄存器设置成异常对应值
* 将断点保存，向IF寄存器发送异常跳转地址
* 阻塞流水线

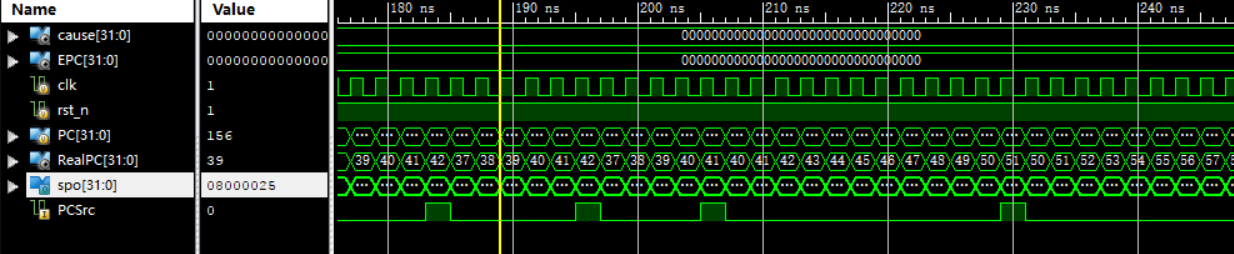
流水线寄存器一般都是转发信号，以及对流水线寄存器的清空信号作出响应，除了EX\_MEM模块的作用是选择rt的来源（这个之后会分析）

实验结果：

DM中最终的数据

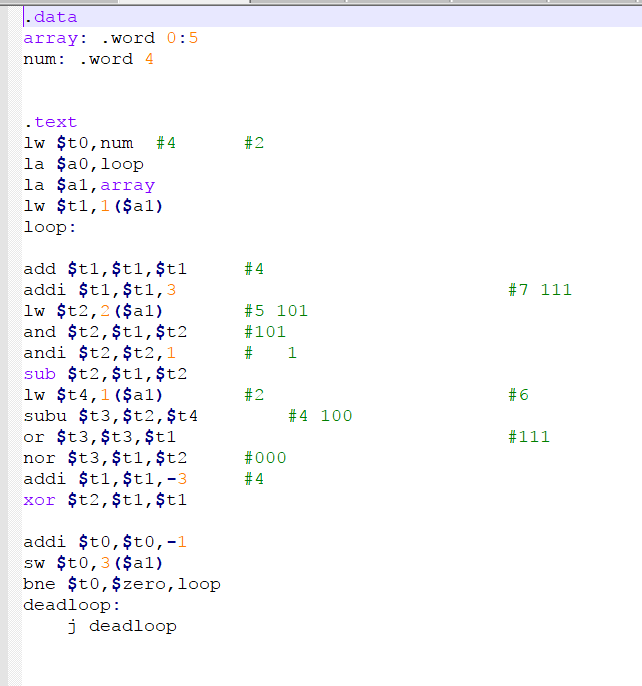


仿真图像，可知中途跳转了三次，最后跳转了一次



**另一个测试样例**

为保证前推和bubble功能的完善性，我另外写了一个测试样例，目的就是测试转发和气泡插入的完备，汇编代码如下：



经单步调试，最终每一步alu的操作数都正确（说明转发和阻塞成功），以及alu的输出结果都正确。

**对于实验过程中出现问题的一些说明**

* 在实验过程中本来转发机制是遵循书上的代码写的，但是在实际实现的过程中，发现对于R型指令，被写入的值是rd部分，对于I型指令如lw，被写入的值是rt部分，这就给转发带来问题：一方面是应该被转发的因为I型指令没有rd而得不到转发，另一方面是rt作为目的寄存器，应该是被本指令写入值，结果被当成源操作数寄存器被写入了上一条指令的结果值。

于是解决办法是增加了一条控制信号isR，区分开R型和I型，然后对于转发指令单独处理

但是对于sw指令，这条指令又很特殊，它是I型指令但是rt是存的数，属于源操作数，所以对sw指令又要根据MemRead单独处理。

在转发模块的编写过程中，为了保证代码的一致性（比如上一条指令的目的等于下一条指令的源），都将目的统一写作rt，但是在实际传参的过程中，对R型指令传rd，对I型指令传rt。

* 周期的说明

在上述模块中，control模块，transmit模块，bubble模块都是组合逻辑，在同一周期内发出信号；

Register\_group采用的是异步读。这是客观情形所致。因为每个时钟周期要完成一个模块的任务，但是每个模块都要写流水线寄存器，这样就必须要异步读，否则要到下一周期才能写入流水线寄存器。

* 为了多种条件判断指令的正确运行，增加了pattern控制信号。Pattern约定为不同的值时，对应不同的跳转条件，因为对应不同的SF和ZF组合。比如bgtz就是SF&（ZF）等等。
* 增加了sign控制信号。在译码的时候，根据指令是sub还是subu确定是有符号运算还是无符号运算，之后确定要不要判断溢出。

**源代码**

**IF.v**

module IF(

input clk,

input rst\_n,

input [31:0] PCBranch,

input PCWrite,

input PCSrc,

input brk,

input [31:0] brk\_vec,

output reg [31:0] PCplus4,

output [31:0] instr

);

reg [31:0] PC;

always@(posedge clk or negedge rst\_n)

begin

if(~rst\_n)

begin

PC=0;

PCplus4=4;

end

else if(brk==1)

PC=brk\_vec;

else if(PCWrite==1)

begin

if(PCSrc==0)

PC=PCplus4;

else PC=PCBranch;

PCplus4=PC+4; //（错,必须给PC一个初值）因为阻塞性赋值，竟然先赋值PC后PCplus4会有效。。。。,

end

end

wire [31:0] RealPC;

assign RealPC=PC>>2;

IM u\_IM(

.a (RealPC ),

.spo (instr )

);

Endmodule

**IF\_ID.v**

module IF\_ID(

input clk,

input [31:0] i\_instr,

input [31:0] i\_PCplus4,

input IF\_ID\_FLUSH,

input IF\_ID\_Write,

output reg [31:0] o\_instr,

output reg [31:0] o\_PCplus4,

output reg [4:0] rs,

output reg [4:0] rt

);

always@(posedge clk)

begin

if(IF\_ID\_FLUSH==1)

begin

o\_PCplus4=0;

o\_instr=0;

rs=0;

rt=0;

end

else if(IF\_ID\_Write==1)

begin

o\_PCplus4=i\_PCplus4;

o\_instr=i\_instr;

rs=i\_instr[25:21];

rt=i\_instr[20:16];

end

end

endmodule

**ID.v**

module ID(

input clk,

input rst\_n,

input [4:0] r3\_addr,

input r3\_wr,

input [31:0] r3\_din,

input [31:0] i\_PCplus4,

input [31:0] instr,

input Extop,

output [31:0] r1\_dout,

output [31:0] r2\_dout,

output reg [31:0] SignImm,

output [31:0] o\_PCplus4,

output [4:0] rs,

output [4:0] rt,

output [4:0] rd,

output [31:0] jBranch

);

wire [4:0] r1\_addr;

wire [4:0] r2\_addr;

assign r1\_addr=instr[25:21];

assign r2\_addr=instr[20:16];

assign o\_PCplus4=i\_PCplus4;

assign rs=instr[25:21];

assign rt=instr[20:16];

assign rd=instr[15:11];

always@(\*)

begin

// if(jump==1) //对j指令需要单独译码，否则输出的SignImm就错了

// SignImm=(instr<<6)>>6;

// else

if(Extop==0)

SignImm=((instr<<16)>>14);

else

SignImm=((instr<<16)>>16)+32'hffff0000\*instr[15];

end

assign jBranch=(instr<<6)>>4;

register\_group u\_register\_group(

.clk (clk ),

.rst\_n (rst\_n ),

.r1\_addr (r1\_addr ),

.r2\_addr (r2\_addr ),

.r3\_addr (r3\_addr ),

.r3\_din (r3\_din ),

.r3\_wr (r3\_wr ), //可以用实参的某一位例化！！

.r1\_dout (r1\_dout ),

.r2\_dout (r2\_dout )

);

Endmodule

**Register\_group.v**

module register\_group(

input clk,

input rst\_n,

input [4:0] r1\_addr,

input [4:0] r2\_addr,

input [4:0] r3\_addr,

input [31:0] r3\_din,

input r3\_wr,

output signed [31:0] r1\_dout,

output signed [31:0] r2\_dout

);

reg [31:0] reg\_r1\_dout;

reg [31:0] reg\_r2\_dout;

assign r1\_dout=reg\_r1\_dout;

assign r2\_dout=reg\_r2\_dout;

//注意逻辑结构的输出时需要加一层寄存器保存线变量的值的，输入可以直接使用

//对reg可以assign

reg [31:0] reg\_group [31:0];

integer i=0;

always@(\*)

begin

if(~rst\_n)

begin

reg\_r1\_dout=0;

reg\_r2\_dout=0;

reg\_group[0]=0;

reg\_group[1]=0;

for(i=2;i<6'd32;i=i+1)

reg\_group[i]=0;

end

else

begin

if(r3\_wr==1)

reg\_group[r3\_addr]=r3\_din;

reg\_r1\_dout=reg\_group[r1\_addr];

reg\_r2\_dout=reg\_group[r2\_addr];

end

end

/\*

always@(posedge clk or negedge rst\_n) //必须改成时序了！

begin

if(rst\_n==0)

begin

reg\_group[0]=0;

reg\_group[1]=0;

for(i=2;i<6'd32;i=i+1)

reg\_group[i]=0;

end

else

begin

if(r3\_wr==1)

reg\_group[r3\_addr]=r3\_din;

end

end

\*/

Endmodule

**ID\_EX.v**

module ID\_EX(

input clk,

input ID\_EX\_sel,

input [2:0] pattern,

input MemtoReg,

input Branch,

input MemRead,

input MemWrite,

input RegDst,

input [4:0] ALUOp,

input ALUSrc,

input RegWrite,

input [2:0] jump,

input sign,

input [31:0] ID\_PCplus4,

input [31:0] r1\_dout,

input [31:0] r2\_dout,

input [31:0] SignImm,

input [4:0] rs,

input [4:0] rt,

input [4:0] rd,

input isR,

input [31:0] jBranch,

output reg [2:0] o\_pattern,

output reg o\_MemtoReg,

output reg o\_Branch,

output reg o\_MemRead,

output reg o\_MemWrite,

output reg o\_RegDst,

output reg [4:0] o\_ALUOp,

output reg o\_ALUSrc,

output reg o\_RegWrite,

output reg [2:0] o\_jump,

output reg [31:0] o\_PCplus4,

output reg [31:0] o\_r1\_dout,

output reg [31:0] o\_r2\_dout,

output reg [31:0] o\_SignImm,

output reg [4:0] o\_rs,

output reg [4:0] o\_rt,

output reg [4:0] o\_rd,

output reg o\_isR,

output reg [31:0] o\_jBranch,

output reg o\_sign

);

always@(posedge clk)

begin

if(ID\_EX\_sel==1)

begin

o\_pattern=0;

o\_MemtoReg=0;

o\_Branch=0;

o\_MemRead=0;

o\_MemWrite=0;

o\_RegDst=0;

o\_ALUOp=0;

o\_ALUSrc=0;

o\_RegWrite=0;

o\_rs=0;

o\_rt=0;

o\_rd=0;

o\_isR=0;

o\_jump=0;

o\_jBranch=0;

o\_sign=0;

end

else

begin

o\_pattern=pattern;

o\_MemtoReg=MemtoReg;

o\_Branch=Branch;

o\_MemRead=MemRead;

o\_MemWrite=MemWrite;

o\_RegDst=RegDst;

o\_ALUOp=ALUOp;

o\_ALUSrc=ALUSrc;

o\_RegWrite=RegWrite;

o\_rs=rs;

o\_rt=rt;

o\_rd=rd;

o\_isR=isR;

o\_jump=jump;

o\_jBranch=jBranch;

o\_sign=sign;

end

o\_PCplus4=ID\_PCplus4;

o\_r1\_dout=r1\_dout;

o\_r2\_dout=r2\_dout;

o\_SignImm=SignImm;

end

endmodule

**EX.v**

module EX(

input rst\_n,

input [31:0] PCplus4,

input [31:0] r1\_dout,

input [31:0] r2\_dout,

input [31:0] SignImm,

input [4:0] rt,

input [4:0] rd,

input [4:0] rs,

input [31:0] i\_douta\_alu\_out,

input [2:0] pattern,

input ALUSrc,

input RegDst,

input [31:0] i\_alu\_out,

input [1:0] ForwardA,

input [1:0] ForwardB,

input [4:0] ALUOp,

input [2:0] jump,

input [31:0] jBranch,

input sign,

output reg bj,

output reg [31:0] alu\_b,

output [31:0] alu\_out,

output [4:0] RegMux,

output reg [31:0] PCBranch,

output reg [31:0] except\_addr,

output reg overflow

);

wire SF;

wire ZF;

reg [31:0] alu\_a;

always@(\*)

begin

case(ForwardA)

2'b00:alu\_a=r1\_dout;

2'b01:alu\_a=i\_douta\_alu\_out;

2'b10:alu\_a=i\_alu\_out;

default:

;

endcase

end

always@(\*)

begin

case(ForwardB)

2'b00:alu\_b=alu\_bpre;

2'b01:alu\_b=i\_douta\_alu\_out;

2'b10:alu\_b=i\_alu\_out;

default:

;

endcase

end

always@(\*)

begin

if(~rst\_n)

bj=0;

else if(pattern==3'd2)

bj=~ZF;

else if(pattern==1)

bj=SF&(~ZF);

end

always@(\*)

begin

if(jump==1)

PCBranch=jBranch;

else if(jump==3'd2) //此处要考虑转发

PCBranch=alu\_a;

else

PCBranch=SignImm\*4+PCplus4;

end

wire [31:0] alu\_bpre;

assign alu\_bpre=(ALUSrc==0)?r2\_dout:SignImm;

assign RegMux=(RegDst==0)?rt:rd;

wire OF;

always@(\*)

begin

if((ALUOp==1 || ALUOp==5'd2)&&sign==1 && OF==1)

begin

overflow=1;

except\_addr=PCplus4-4;

end

else

begin

overflow=0;

except\_addr=0;

end

end

alu u\_alu(

.rst\_n (rst\_n ),

.alu\_a (alu\_a ),

.alu\_b (alu\_b ),

.alu\_op (ALUOp ),

.alu\_out (alu\_out ),

.SF (SF ),

.ZF (ZF ),

.OF (OF )

);

endmodule

**alu.v**

module alu(

input rst\_n,

input signed [31:0] alu\_a,

input signed [31:0] alu\_b,

input [4:0] alu\_op,

input sign,

output signed [31:0] alu\_out,

output SF,

output ZF,

output reg OF,

output reg zerodivisor

);

reg [31:0] out\_temp;

assign alu\_out=out\_temp;

assign ZF=(alu\_out==0)?1'b1:1'b0;

assign SF=(alu\_out[31]==0)?1'b1:1'b0;

parameter A\_NOP =5'h00;

parameter A\_ADD =5'h01;

parameter A\_SUB = 5'h02;

parameter A\_AND = 5'h03;

parameter A\_OR = 5'h04;

parameter A\_XOR = 5'h05;

parameter A\_NOR = 5'h06;

always@(\*)

begin

if(~rst\_n)

zerodivisor=0;

case(alu\_op)

5'h00:out\_temp= 0;

5'h01:

begin

out\_temp = alu\_a+alu\_b;

if(alu\_a[31]==alu\_b[31] && alu\_a[31]!=out\_temp[31])

OF=1;

else

OF=0;

end

5'h02:

begin

out\_temp = alu\_a-alu\_b;

if(alu\_a[31]!=alu\_b[31] && alu\_b[31]==out\_temp[31])

OF=1;

else

OF=0;

end

5'h03:out\_temp = alu\_a & alu\_b;

5'h04:out\_temp = alu\_a | alu\_b;

5'h05:out\_temp = alu\_a ^ alu\_b;

5'h06:out\_temp = ~(alu\_a | alu\_b);

5'h07:out\_temp = alu\_a \* alu\_b;

5'd08:

begin

if(alu\_b==0)

zerodivisor=1;

else

out\_temp = alu\_a / alu\_b;

end

// 5'h00:out\_temp = 0;

default:out\_temp = 0;

endcase

end

endmodule

**EX\_MEM.v**

module EX\_MEM(

input clk,

input rst\_n,

input i\_MemtoReg,

input i\_Branch,

input i\_MemRead,

input i\_MemWrite,

input i\_RegWrite,

input [31:0] i\_alu\_out,

input i\_bj,

input [4:0] i\_RegMux,

input [31:0] i\_PCBranch,

input [31:0] i\_alu\_b,

input [4:0] i\_rd,

input [4:0] i\_rt,

input isR,

input EX\_MEM\_sel,

input [2:0] jump,

output reg o\_MemtoReg,

output reg o\_Branch,

output reg o\_MemRead,

output reg o\_MemWrite,

output reg [31:0] o\_PCBranch,

output reg o\_RegWrite,

output reg [31:0] o\_alu\_out,

output reg o\_bj,

output reg [4:0] o\_RegMux,

output reg [31:0] o\_alu\_b,

output reg [4:0] o\_rt,

output reg [4:0] o\_rd,

output reg [4:0] o\_dest,

output reg PCSrc

);

//reg行变量不能用assign赋值

reg block;

always@(posedge clk or negedge rst\_n)

begin

if(EX\_MEM\_sel==1 || rst\_n==0)

begin

o\_MemtoReg=0;

o\_Branch=0;

o\_MemRead=0;

o\_MemWrite=0;

o\_PCBranch=0;

o\_RegWrite=0;

o\_alu\_out=0;

o\_bj=0;

o\_RegMux=0;

o\_rt=0;

o\_rd=0;

o\_alu\_b=0;

o\_dest=0;

PCSrc=0;

block=0;

end

else

begin

o\_MemtoReg=i\_MemtoReg;

o\_Branch=i\_Branch;

o\_MemRead=i\_MemRead;

o\_MemWrite=i\_MemWrite;

o\_PCBranch=i\_PCBranch;

o\_RegWrite=i\_RegWrite;

o\_alu\_out=i\_alu\_out;

o\_bj=i\_bj;

o\_RegMux=i\_RegMux;

o\_rt=i\_rt;

o\_rd=i\_rd;

o\_alu\_b=i\_alu\_b;

if(isR==1)

o\_dest=i\_rd;

else

o\_dest=i\_rt;

PCSrc=(i\_Branch && i\_bj && (~block) || jump);

if(i\_Branch && i\_bj && (~block)==1)

block=1;

else

block=0;

end

end

endmodule

**MEM.v**

module MEM(

input clk,

input MemWrite,

input MemRead,

input Branch,

input bj,

input [31:0] alu\_out,

input [31:0] alu\_b,

input [4:0] RegMux,

input [1:0] ForwardC,

input [31:0] i\_douta,

input [31:0] MEM\_WB\_alu\_out,

output [4:0] o\_RegMux,

output [31:0] o\_alu\_out,

output [31:0] douta

);

reg [31:0] dina;

wire [31:0] douta\_pre;

assign douta=(MemRead==1)?douta\_pre:32'b0;

always@(\*)

begin

if(ForwardC==1)

dina=i\_douta;

else if(ForwardC==2'd2)

dina=MEM\_WB\_alu\_out;

else dina=alu\_b;

end

assign o\_RegMux=RegMux;

assign o\_alu\_out=alu\_out;

wire [31:0] Realaddr;

assign Realaddr=alu\_out>>2;

DM u\_DM(

.clk (clk ),

.a (Realaddr ),

.d (dina ),

.we (MemWrite ),

.spo (douta\_pre )

);

Endmodule

MEM\_WB.v

module MEM\_WB(

input clk,

input RegWrite,

input MemtoReg,

input MemRead,

input [31:0] douta,

input [31:0] alu\_out,

input [4:0] RegMux,

input [4:0] rt,

input [4:0] dest,

output reg o\_RegWrite,

output reg o\_MemtoReg,

output reg o\_MemRead,

output reg [31:0] o\_douta,

output reg [31:0] o\_alu\_out,

output reg [4:0] o\_RegMux,

output reg [4:0] o\_rt,

output reg [4:0] o\_dest

);

always@(posedge clk)

begin

o\_RegWrite=RegWrite;

o\_MemtoReg=MemtoReg;

o\_MemRead=MemRead;

o\_douta=douta;

o\_alu\_out=alu\_out;

o\_RegMux=RegMux;

o\_rt=rt;

o\_dest=dest;

end

endmodule

**WB.v**

module WB(

input clk,

input rst\_n,

input [31:0] douta,

input [31:0] alu\_out,

input MemtoReg,

input RegWrite,

input [4:0] RegMux,

output [31:0] r3\_din,

output [4:0] r3\_addr,

output o\_RegWrite

);

assign r3\_din=(MemtoReg==0)?alu\_out:douta;

assign r3\_addr=RegMux;

assign o\_RegWrite=RegWrite;

endmodule

**control.v**

module control(

input clk,

input rst\_n,

input [31:0] instr,

input [31:0] PCplus4,

output reg Extop,

output reg RegDst,

output reg Branch,

output reg MemRead,

output reg MemtoReg,

output reg MemWrite,

output reg [4:0] ALUOp,

output reg RegWrite,

output reg ALUSrc,

// output reg PCSrc,

output reg [2:0] pattern,

output reg [2:0] jump,

output reg sign,

output reg isR,

output reg udfist,

output reg [31:0] except\_addr

);

wire [5:0] Op;

assign Op=instr[31:26];

wire [5:0] Funct;

assign Funct=instr[5:0];

reg PCSrc;

always@(\*)

begin

if(~rst\_n)

begin

PCSrc=0;

RegDst=0;

Branch=0;

MemtoReg=0;

MemWrite=0;

ALUOp=0;

ALUSrc=0;

PCSrc=0;

Extop=0;

jump=0;

pattern=0;

MemRead=0;

RegWrite=0;

isR=0;

sign =0;

end

else

case(Op)

6'b000000:begin

if(Funct==6'b100000) //add

begin

MemRead=0;

RegDst=1;

Branch=0;

MemtoReg=0;

MemWrite=0;

ALUOp=1;

RegWrite=1;

ALUSrc=0;

PCSrc=0;

jump=0;

isR=1;

sign=1;

end

else if(Funct==6'b100001) //addu

begin

MemRead=0;

RegDst=1;

Branch=0;

MemtoReg=0;

MemWrite=0;

ALUOp=1;

RegWrite=1;

ALUSrc=0;

PCSrc=0;

jump=0;

isR=1;

sign=0;

end

else if(Funct==6'b100011) //subu

begin

MemRead=0;

RegDst=1;

Branch=0;

MemtoReg=0;

MemWrite=0;

ALUOp=5'd2;

RegWrite=1;

ALUSrc=0;

PCSrc=0;

jump=0;

isR=1;

sign=0;

end

else if(Funct==6'b100010) //sub

begin

MemWrite=0;

MemRead=0;

RegDst=1;

Branch=0;

MemtoReg=0;

RegWrite=1;

ALUOp=5'd2;

ALUSrc=0;

PCSrc=0;

jump=0;

isR=1;

sign=1;

end

else if(Funct==6'b100100) //and

begin

MemRead=0;

RegDst=1;

Branch=0;

MemtoReg=0;

MemWrite=0;

ALUOp=5'd3;

RegWrite=1;

ALUSrc=0;

PCSrc=0;

jump=0;

isR=1;

sign=0;

end

else if(Funct==6'b100101) //or

begin

MemRead=0;

RegDst=1;

Branch=0;

MemtoReg=0;

MemWrite=0;

ALUOp=5'd4;

RegWrite=1;

ALUSrc=0;

PCSrc=0;

jump=0;

isR=1;

sign=0;

end

else if(Funct==6'b100111) //nor

begin

MemRead=0;

RegDst=1;

Branch=0;

MemtoReg=0;

MemWrite=0;

ALUOp=5'd6;

RegWrite=1;

ALUSrc=0;

PCSrc=0;

jump=0;

isR=1;

sign=0;

end

else if(Funct==6'b100110) //xor

begin

MemRead=0;

RegDst=1;

Branch=0;

MemtoReg=0;

MemWrite=0;

ALUOp=5'd5;

RegWrite=1;

ALUSrc=0;

PCSrc=0;

RegWrite=1;

jump=0;

isR=1;

sign=0;

end

else if(Funct==6'b011000)

begin

MemWrite=0;

MemRead=0;

RegDst=1;

Branch=0;

MemtoReg=0;

RegWrite=1;

ALUOp=5'd7;

ALUSrc=0;

PCSrc=0;

jump=0;

isR=1;

sign=1;

end

else if(Funct==6'b001000) //jr

begin

MemRead=0;

RegDst=0;

Branch=1;

MemtoReg=0;

MemWrite=0;

ALUOp=5'd1;

RegWrite=0;

ALUSrc=0;

PCSrc=1;

Extop=0;

isR=0;

jump=2;

sign=0;

end

else

jump=0;

end

6'b001000: //addi

begin

MemRead=0;

RegDst=0;

Branch=0;

MemtoReg=0;

RegWrite=1;

ALUOp=5'd1;

ALUSrc=1;

PCSrc=0;

Extop=1;

jump=0;

isR=0;

MemWrite=0;

sign=1;

end

6'b001001: //addiu

begin

MemRead=0;

RegDst=0;

Branch=0;

MemtoReg=0;

MemWrite=0;

ALUOp=5'd1;

RegWrite=1;

ALUSrc=1;

PCSrc=0;

Extop=1;

jump=0;

isR=0;

sign=0;

end

6'b001100: //andi

begin

MemRead=0;

RegDst=0;

Branch=0;

MemtoReg=0;

MemWrite=0;

ALUOp=5'd3;

RegWrite=1;

ALUSrc=1;

PCSrc=0;

Extop=1;

jump=0;

isR=0;

sign=0;

end

6'b000111: //bgtz

begin

MemRead=0;

Branch=1;

MemWrite=0;

ALUOp=5'd2;

RegWrite=0;

ALUSrc=0;

Extop=1;

jump=0;

pattern=1; //为之后运算bj指定方案

isR=0;

sign=0;

end

6'b000101: //bne

begin

MemRead=0;

Branch=1;

MemWrite=0;

ALUOp=5'd2;

RegWrite=0;

ALUSrc=0;

Extop=1;

jump=0;

pattern=3'd2;

isR=1;

sign=0;

end

6'b000001: //bgez

begin

MemRead=0;

Branch=1;

MemWrite=0;

ALUOp=5'd2;

RegWrite=0;

ALUSrc=0;

Extop=1;

jump=0;

isR=0;

if(instr[20:16]==5'b00001)

pattern=3'd2;

else if(instr[20:16]==5'b00000)

pattern=3'd4; //小于0，bltz

end

6'b000110: //blez,<=0

begin

MemWrite=0;

MemRead=0;

Branch=1;

MemWrite=0;

ALUOp=5'd2;

RegWrite=0;

ALUSrc=0;

Extop=1;

jump=0;

pattern=3'd3;

isR=0;

end

6'b000010: //j

begin

MemRead=0;

jump=1;

Branch=1;

RegWrite=0;

MemWrite=0;

PCSrc=1;

Extop=0;

isR=0;

end

6'b000000: //jr

begin

MemWrite=0;

jump=1;

ALUSrc=0;

Branch=1;

isR=0;

end

6'b100011: //lw

begin

MemtoReg=1;

RegDst=0;

ALUOp=1;

ALUSrc=1;

MemWrite=0;

MemRead=1;

PCSrc=0;

Extop=1;

jump=0;

RegWrite=1;

isR=0;

end

6'b101011: //sw

begin

RegWrite=0;

Branch=0;

ALUOp=1;

ALUSrc=1;

PCSrc=0;

MemWrite=1;

MemRead=0;

jump=0;

Extop=1;

isR=1;

end

default:

begin

udfist=1;

except\_addr=PCplus4-4;

end

endcase

end

endmodule

**transmit.v**

module transmit(

input clk,

input rst\_n,

input [4:0] EX\_MEM\_RD, //

input [4:0] MEM\_WB\_RD, //这条及以上是假RD

input [4:0] ID\_EX\_RS,

input [4:0] ID\_EX\_RT,

input [4:0] MEM\_WB\_RT,

input MEM\_WB\_RegWrite,

input EX\_MEM\_RegWrite,

input ID\_EX\_MemWrite,

input EX\_MEM\_MemWrite,

input ID\_EX\_MemRead,

input ID\_EX\_isR,

input [4:0] EX\_MEM\_RT,

input MEM\_WB\_MemRead,

output reg [1:0] ForwardA,

output reg [1:0] ForwardB,

output reg [1:0] ForwardC

);

//转发机制要求就在于，在ID\_EX写入之后EX执行之前，就发出控制信号干涉EX

always@(\*)

begin

if(~rst\_n)

begin

ForwardA=0;

ForwardB=0;

ForwardC=0;

end

else

begin

if((MEM\_WB\_RegWrite==1) &&

(MEM\_WB\_RD!=0) &&

!((EX\_MEM\_RegWrite==1)&&(EX\_MEM\_RD!=0) &&(EX\_MEM\_RD==ID\_EX\_RS))&&

(MEM\_WB\_RD==ID\_EX\_RS))

ForwardA=2'b01;

else if((EX\_MEM\_RegWrite==1)&&

(EX\_MEM\_RD!=0)&&

(EX\_MEM\_RD==ID\_EX\_RS))

ForwardA=2'b10;

else ForwardA=2'b00;

if((MEM\_WB\_RegWrite==1) &&

(MEM\_WB\_RD!=0) &&

!((EX\_MEM\_RegWrite==1)&&(EX\_MEM\_RD!=0) &&(EX\_MEM\_RD==ID\_EX\_RT))&&

(MEM\_WB\_RD==ID\_EX\_RT))

ForwardB=2'b01;

else if((EX\_MEM\_RegWrite==1)&&

(ID\_EX\_isR==1)&& //确保后被执行的指令是R型指令

(EX\_MEM\_RD!=0)&&

(ID\_EX\_MemWrite!=1)&&

(EX\_MEM\_RD==ID\_EX\_RT))

ForwardB=2'b10;

else ForwardB=2'b00;

if((MEM\_WB\_RegWrite==1)&& //上一条指令是lw

(MEM\_WB\_MemRead==1)&&

(EX\_MEM\_MemWrite==1)&&

(EX\_MEM\_RT==MEM\_WB\_RT))

ForwardC=1;

else if((MEM\_WB\_RegWrite==1)&& //上一条指令是ALU指令

(MEM\_WB\_MemRead==0)&&

(EX\_MEM\_MemWrite==1)&&

(EX\_MEM\_RT==MEM\_WB\_RD)) //会造成前一个转发也被判定出来

ForwardC=2'd2;

else

ForwardC=0;

end

end

//这里有个曹丹的问题，rt在R型指令中是源操作数，需要转发，在I型指令中是目的操作数，不需要转发

/\*

always@(\*)

begin

if((MEM\_WB\_RegWrite==1)&&

(MEM\_WB\_RD!=0)&&

!((EX\_MEM\_RegWrite==1)&&(EX\_MEM\_RD!=0))&&

(MEM\_WB\_RD==ID\_EX\_RT))

ForwardB=2'b01;

else if((EX\_MEM\_RegWrite==1)&&

// (ID\_EX\_isR==1)&& //确保后被执行的指令是R型指令

(EX\_MEM\_RD!=0)&&

(EX\_MEM\_RD==ID\_EX\_RT))

ForwardB=2'b10;

else ForwardB=2'b00;

end

always@(\*) //对于先lw再sw的情形

begin

if((MEM\_WB\_RegWrite==1)&&

(EX\_MEM\_MemWrite==1)&&

(EX\_MEM\_RT==MEM\_WB\_RT))

ForwardC=1;

else

ForwardC=0;

end

\*/

/\*

always@(posedge clk)

begin

if((ID\_EX\_MemRead==1)&&

((ID\_EX\_RT==IF\_ID\_RS) || (ID\_EX\_RT==IF\_ID\_RT)))

stall=1;

else

stall=0;

end

\*/

Endmodule

**bubble.v**

module bubble(

input clk,

input rst\_n,

input [4:0] IF\_ID\_RS,

input [4:0] IF\_ID\_RT,

input [4:0] ID\_EX\_RT,

input ID\_EX\_MemRead,

input PCSrc,

input brk,

output reg PCWrite,

output reg IF\_ID\_Write,

output reg ID\_EX\_sel,

output reg IF\_ID\_FLUSH,

output reg EX\_MEM\_sel

);

//产生bubble的几种情况：

//lw + 算数运算型/beq型（RS/rt部分）

//lw + sw/beq型 型（作baseR）

//延迟分支，需要添加1~2个bubble

//sel是将信号清空

//write是阻止信号被覆盖，原信号向后传播

//flush是清空信号

always@(\*)

begin //lw指令

if(~rst\_n)

begin

PCWrite=1;

ID\_EX\_sel=0;

EX\_MEM\_sel=0;

IF\_ID\_Write=1;

IF\_ID\_FLUSH=1; //用这个来阻塞rst\_n，防止第一条指令充满流水线

end

else

begin

if((ID\_EX\_MemRead==1)&&

(ID\_EX\_RT==IF\_ID\_RS ||ID\_EX\_RT==IF\_ID\_RT))

begin

PCWrite=0;

ID\_EX\_sel=1;

IF\_ID\_Write=0;

IF\_ID\_FLUSH=0;

end

else if(PCSrc==1 || brk==1)

begin

IF\_ID\_FLUSH=1;

EX\_MEM\_sel=1;

ID\_EX\_sel=1;

IF\_ID\_Write=0;

end

else

begin

PCWrite=1;

ID\_EX\_sel=0;

EX\_MEM\_sel=0;

IF\_ID\_Write=1;

IF\_ID\_FLUSH=0;

end

end

end

endmodule

**fault.v**

module fault(

input overflow,

input udfist,

input [31:0] of\_except\_addr,

input [31:0] udfist\_except\_addr,

output reg [31:0] cause,

output reg [31:0] EPC,

output reg brk,

output reg [31:0] brk\_vec

);

always@(\*)

begin

if(udfist==1)

begin

cause=32'd10;

EPC=udfist\_except\_addr;

brk=1;

brk\_vec=32'h80000000;

end

else if(overflow==1)

begin

cause=32'd12;

EPC=of\_except\_addr;

brk=1;

brk\_vec=32'h80000018;

end

else

begin

cause=0;

EPC=0;

brk=0;

brk\_vec=0;

end

end

endmodule

**top.v**

module top(

input clk,

input rst\_n,

output [31:0] cause,

output [31:0] EPC

);

//发现一件事，端口的线网变量有默认声明，所以wire的声明不能放在模块的使用之后

//assign PCjump=(instr<<6)>>4+(PCplus4>>28)<<28;

//IF

wire [31:0] IF\_PCplus4;

wire [31:0] IF\_instr;

//IF\_ID

wire [31:0] IF\_ID\_instr;

wire [31:0] IF\_ID\_PCplus4;

//ID

wire [31:0] ID\_r1\_dout;

wire [31:0] ID\_r2\_dout;

wire [31:0] ID\_SignImm;

wire [31:0] ID\_PCplus4;

wire [4:0] ID\_rs;

wire [4:0] ID\_rt;

wire [4:0] ID\_rd;

wire [31:0] jBranch;

//ID\_EX

wire [2:0] ID\_EX\_pattern;

wire ID\_EX\_MemtoReg;

wire ID\_EX\_Branch;

wire ID\_EX\_MemRead;

wire ID\_EX\_MemWrite;

wire ID\_EX\_RegDst;

wire [4:0] ID\_EX\_ALUOp;

wire ID\_EX\_ALUSrc;

wire ID\_EX\_RegWrite;

wire ID\_EX\_isR;

wire [31:0] ID\_EX\_PCplus4;

wire [31:0] ID\_EX\_r1\_dout;

wire [31:0] ID\_EX\_r2\_dout;

wire [31:0] ID\_EX\_SignImm;

wire [4:0] ID\_EX\_rs;

wire [4:0] ID\_EX\_rt;

wire [4:0] ID\_EX\_rd;

wire [31:0] ID\_EX\_jBranch;

wire [2:0] ID\_EX\_jump;

wire ID\_EX\_sign;

//EX

wire EX\_bj;

wire [31:0] EX\_alu\_b;

wire [31:0] EX\_alu\_out;

wire [4:0] EX\_RegMux;

wire [31:0] EX\_PCBranch;

wire [4:0] EX\_rt;

wire [4:0] EX\_rd;

wire [2:0] EX\_jump;

//EX\_MEM

wire EX\_MEM\_MemtoReg;

wire EX\_MEM\_Branch;

wire EX\_MEM\_MemRead;

wire EX\_MEM\_MemWrite;

wire [31:0] EX\_MEM\_PCBranch;

wire EX\_MEM\_RegWrite;

wire [31:0] EX\_MEM\_alu\_out;

wire EX\_MEM\_bj;

wire [4:0] EX\_MEM\_RegMux;

wire [31:0] EX\_MEM\_alu\_b;

wire [4:0] EX\_MEM\_rt;

wire [4:0] EX\_MEM\_rd;

wire [4:0] EX\_MEM\_dest;

wire EX\_MEM\_PCSrc;

wire EX\_MEM\_isR;

//MEM

wire [31:0] MEM\_douta;

wire [31:0] MEM\_alu\_out;

wire [4:0] MEM\_RegMux;

wire [31:0] MEM\_PCBranch;

//MEM\_WB

wire MEM\_WB\_RegWrite;

wire MEM\_WB\_MemtoReg;

wire MEM\_WB\_MemRead;

wire [31:0] MEM\_WB\_douta;

wire [31:0] MEM\_WB\_alu\_out;

wire [4:0] MEM\_WB\_RegMux;

wire [4:0] MEM\_WB\_rd;

wire [4:0] MEM\_WB\_rt;

wire [4:0] MEM\_WB\_dest;

//WB

wire [31:0] WB\_r3\_din;

wire [4:0] WB\_r3\_addr;

wire WB\_RegWrite;

//CONTROL

wire Extop;

wire RegDst;

wire Branch;

wire MemRead;

wire MemtoReg;

wire MemWrite;

wire [4:0] ALUOp;

wire RegWrite;

wire ALUSrc;

wire PCSrc;

wire [2:0] pattern;

wire [2:0] jump;

wire isR;

wire sign;

//TRANSMIT

wire [1:0] ForwardA;

wire [1:0] ForwardB;

wire [1:0] ForwardC;

//BUBBLE

wire PCWrite;

wire IF\_ID\_Write;

wire ID\_EX\_sel;

wire IF\_ID\_FLUSH;

wire [4:0] IF\_ID\_rs;

wire [4:0] IF\_ID\_rt;

//fault

wire [31:0] of\_except\_addr;

wire [31:0] udfist\_except\_addr;

wire overflow;

wire udfist;

wire brk;

wire [31:0] brk\_vec;

IF u\_IF(

.PCBranch (EX\_MEM\_PCBranch ),

.PCSrc (EX\_MEM\_PCSrc ),

.clk (clk ),

.rst\_n (rst\_n ),

.PCWrite (PCWrite),

.PCplus4 (IF\_PCplus4 ),

.instr (IF\_instr ),

.brk (brk ),

.brk\_vec (brk\_vec )

);

IF\_ID u\_IF\_ID(

.clk (clk ),

.i\_instr (IF\_instr ),

.i\_PCplus4 (IF\_PCplus4),

.IF\_ID\_Write (IF\_ID\_Write ),

.IF\_ID\_FLUSH (IF\_ID\_FLUSH ),

.o\_instr (IF\_ID\_instr),

.o\_PCplus4 (IF\_ID\_PCplus4 ),

.rs (IF\_ID\_rs ),

.rt (IF\_ID\_rt )

);

ID u\_ID(

.clk (clk ),

.rst\_n (rst\_n ),

.r3\_addr (WB\_r3\_addr ),

.r3\_wr (WB\_RegWrite),

.r3\_din (WB\_r3\_din ),

.i\_PCplus4 (IF\_ID\_PCplus4 ),

.instr (IF\_ID\_instr ),

.Extop (Extop ),

.r1\_dout (ID\_r1\_dout ),

.r2\_dout (ID\_r2\_dout ),

.SignImm (ID\_SignImm ),

.o\_PCplus4 (ID\_PCplus4 ),

.rs (ID\_rs ),

.rt (ID\_rt ),

.rd (ID\_rd ),

.jBranch (jBranch )

);

ID\_EX u\_ID\_EX(

.clk (clk ),

.ID\_EX\_sel (ID\_EX\_sel ),

.pattern (pattern ),

.MemtoReg (MemtoReg ),

.Branch (Branch ),

.MemRead (MemRead ),

.MemWrite (MemWrite ),

.RegDst (RegDst ),

.ALUOp (ALUOp ),

.ALUSrc (ALUSrc ),

.RegWrite (RegWrite ),

.sign (sign ),

.ID\_PCplus4 (ID\_PCplus4 ),

.r1\_dout (ID\_r1\_dout ),

.r2\_dout (ID\_r2\_dout ),

.SignImm (ID\_SignImm ),

.rs (ID\_rs ),

.rt (ID\_rt ),

.rd (ID\_rd ),

.isR (isR ),

.jBranch (jBranch ),

.jump (jump ),

.o\_jump (ID\_EX\_jump ),

.o\_pattern (ID\_EX\_pattern ),

.o\_MemtoReg (ID\_EX\_MemtoReg),

.o\_Branch (ID\_EX\_Branch),

.o\_MemRead (ID\_EX\_MemRead),

.o\_MemWrite (ID\_EX\_MemWrite),

.o\_RegDst (ID\_EX\_RegDst ),

.o\_ALUOp (ID\_EX\_ALUOp ),

.o\_ALUSrc (ID\_EX\_ALUSrc ),

.o\_RegWrite (ID\_EX\_RegWrite),

.o\_PCplus4 (ID\_EX\_PCplus4 ),

.o\_sign (ID\_EX\_sign ),

.o\_r1\_dout (ID\_EX\_r1\_dout ),

.o\_r2\_dout (ID\_EX\_r2\_dout ),

.o\_SignImm (ID\_EX\_SignImm ),

.o\_rs (ID\_EX\_rs ),

.o\_rt (ID\_EX\_rt ),

.o\_rd (ID\_EX\_rd ),

.o\_isR (ID\_EX\_isR ),

.o\_jBranch (ID\_EX\_jBranch )

);

EX u\_EX(

.rst\_n (rst\_n ),

.PCplus4 (ID\_EX\_PCplus4 ),

.r1\_dout (ID\_EX\_r1\_dout ),

.r2\_dout (ID\_EX\_r2\_dout ),

.SignImm (ID\_EX\_SignImm ),

.rt (ID\_EX\_rt ),

.rd (ID\_EX\_rd ),

.rs (ID\_EX\_rs ),

.ALUOp (ID\_EX\_ALUOp ),

.i\_douta\_alu\_out (WB\_r3\_din ),

.pattern (ID\_EX\_pattern ),

.i\_alu\_out (EX\_MEM\_alu\_out),

.ALUSrc (ID\_EX\_ALUSrc ),

.RegDst (ID\_EX\_RegDst ),

.ForwardA (ForwardA ),

.ForwardB (ForwardB ),

.jBranch (ID\_EX\_jBranch ),

.jump (ID\_EX\_jump ),

.sign (ID\_EX\_sign ),

.bj (EX\_bj ),

.alu\_b (EX\_alu\_b ),

.alu\_out (EX\_alu\_out ),

.RegMux (EX\_RegMux ),

.PCBranch (EX\_PCBranch ),

.overflow (overflow ),

.except\_addr (of\_except\_addr )

);

EX\_MEM u\_EX\_MEM(

.clk (clk ),

.rst\_n (rst\_n ),

.i\_MemtoReg (ID\_EX\_MemtoReg ),

.i\_Branch (ID\_EX\_Branch ),

.i\_MemRead (ID\_EX\_MemRead ),

.i\_MemWrite (ID\_EX\_MemWrite ),

.i\_RegWrite (ID\_EX\_RegWrite ),

.i\_alu\_out (EX\_alu\_out ),

.i\_bj (EX\_bj ),

.i\_RegMux (EX\_RegMux ),

.i\_PCBranch (EX\_PCBranch ),

.i\_alu\_b (EX\_alu\_b ),

.i\_rt (ID\_EX\_rt ),

.i\_rd (ID\_EX\_rd ),

.isR (ID\_EX\_isR ),

.EX\_MEM\_sel (EX\_MEM\_sel ),

.jump (ID\_EX\_jump ),

.o\_MemtoReg (EX\_MEM\_MemtoReg ),

.o\_Branch (EX\_MEM\_Branch ),

.o\_MemRead (EX\_MEM\_MemRead ),

.o\_MemWrite (EX\_MEM\_MemWrite ),

.o\_PCBranch (EX\_MEM\_PCBranch ),

.o\_RegWrite (EX\_MEM\_RegWrite ),

.o\_alu\_out (EX\_MEM\_alu\_out ),

.o\_bj (EX\_MEM\_bj ),

.o\_RegMux (EX\_MEM\_RegMux),

.o\_alu\_b (EX\_MEM\_alu\_b ),

.o\_rt (EX\_MEM\_rt ),

.o\_rd (EX\_MEM\_rd ),

.o\_dest (EX\_MEM\_dest ),

.PCSrc (EX\_MEM\_PCSrc )

);

MEM u\_MEM(

.clk (clk ),

.MemWrite (EX\_MEM\_MemWrite ),

.MemRead (EX\_MEM\_MemRead ),

.Branch (EX\_MEM\_Branch ),

.bj (EX\_MEM\_bj ),

.ForwardC (ForwardC ),

.alu\_out (EX\_MEM\_alu\_out),

.alu\_b (EX\_MEM\_alu\_b),

.RegMux (EX\_MEM\_RegMux ),

.i\_douta (MEM\_WB\_douta),

.MEM\_WB\_alu\_out (MEM\_WB\_alu\_out),

.o\_alu\_out (MEM\_alu\_out ),

.o\_RegMux (MEM\_RegMux ),

.douta (MEM\_douta )

);

MEM\_WB u\_MEM\_WB(

.clk (clk ),

.RegWrite (EX\_MEM\_RegWrite ),

.MemtoReg (EX\_MEM\_MemtoReg ),

.MemRead (EX\_MEM\_MemRead ),

.douta (MEM\_douta ),

.alu\_out (MEM\_alu\_out ),

.RegMux (MEM\_RegMux ),

.rt (EX\_MEM\_rt ),

.dest (EX\_MEM\_dest ),

.o\_RegWrite (MEM\_WB\_RegWrite ),

.o\_MemtoReg (MEM\_WB\_MemtoReg ),

.o\_MemRead (MEM\_WB\_MemRead ),

.o\_douta (MEM\_WB\_douta ),

.o\_alu\_out (MEM\_WB\_alu\_out ),

.o\_RegMux (MEM\_WB\_RegMux ),

.o\_rt (MEM\_WB\_rt ),

.o\_dest (MEM\_WB\_dest )

);

WB u\_WB(

.clk (clk ),

.rst\_n (rst\_n ),

.douta (MEM\_WB\_douta ),

.alu\_out (MEM\_WB\_alu\_out),

.MemtoReg (MEM\_WB\_MemtoReg),

.RegWrite (MEM\_WB\_RegWrite),

.RegMux (MEM\_WB\_RegMux ),

.r3\_din (WB\_r3\_din ),

.r3\_addr (WB\_r3\_addr ),

.o\_RegWrite (WB\_RegWrite)

);

control u\_control(

.clk (clk ),

.rst\_n (rst\_n ),

.instr (IF\_ID\_instr),

.PCplus4 (IF\_ID\_PCplus4 ),

.Extop (Extop ),

.RegDst (RegDst ),

.Branch (Branch ),

.MemRead (MemRead ),

.MemtoReg (MemtoReg ),

.MemWrite (MemWrite ),

.ALUOp (ALUOp ),

.RegWrite (RegWrite ),

.ALUSrc (ALUSrc ),

.pattern (pattern ),

.jump (jump ),

.sign (sign ),

.isR (isR ),

.udfist (udfist ),

.except\_addr (udfist\_except\_addr)

);

transmit u\_transmit(

.clk (clk ),

.rst\_n (rst\_n ),

.ID\_EX\_RS (ID\_EX\_rs ),

.ID\_EX\_RT (ID\_EX\_rt ),

.ID\_EX\_MemRead (ID\_EX\_MemRead ),

.EX\_MEM\_RD (EX\_MEM\_dest ),

.EX\_MEM\_RT (EX\_MEM\_rt ),

.EX\_MEM\_RegWrite (EX\_MEM\_RegWrite),

.ID\_EX\_MemWrite (ID\_EX\_MemWrite ),

.EX\_MEM\_MemWrite (EX\_MEM\_MemWrite),

.MEM\_WB\_RD (MEM\_WB\_dest ),

.MEM\_WB\_RT (MEM\_WB\_rt ),

.MEM\_WB\_RegWrite (MEM\_WB\_RegWrite ),

.ID\_EX\_isR (ID\_EX\_isR ),

.MEM\_WB\_MemRead (MEM\_WB\_MemRead ),

.ForwardA (ForwardA ),

.ForwardB (ForwardB ),

.ForwardC (ForwardC )

);

bubble u\_bubble(

.clk (clk ),

.rst\_n (rst\_n ),

.IF\_ID\_RS (IF\_ID\_rs ),

.IF\_ID\_RT (IF\_ID\_rt ),

.ID\_EX\_RT (ID\_EX\_rt ),

.ID\_EX\_MemRead (ID\_EX\_MemRead ),

.PCSrc (EX\_MEM\_PCSrc ),

.brk (brk ),

.PCWrite (PCWrite ),

.IF\_ID\_Write (IF\_ID\_Write ),

.ID\_EX\_sel (ID\_EX\_sel ),

.EX\_MEM\_sel (EX\_MEM\_sel ),

.IF\_ID\_FLUSH (IF\_ID\_FLUSH )

);

fault u\_fault(

.overflow (overflow ),

.udfist (udfist ),

.of\_except\_addr (of\_except\_addr ),

.udfist\_except\_addr (udfist\_except\_addr ),

.cause (cause ),

.EPC (EPC ),

.brk (brk ),

.brk\_vec (brk\_vec )

);

Endmodule

测试文件：test.v

module test;

// Inputs

reg clk;

reg rst\_n;

// Outputs

wire [31:0] cause;

wire [31:0] EPC;

// Instantiate the Unit Under Test (UUT)

top uut (

.clk(clk),

.rst\_n(rst\_n),

.cause(cause),

.EPC(EPC)

);

initial begin

// Initialize Inputs

rst\_n = 0;

#100;

rst\_n=1;

// Wait 100 ns for global reset to finish

// Add stimulus here

end

initial begin

clk=0;

forever #1 clk=~clk;

end

endmodule