## 一、设计目的与要求

### 1.实验目的

1) 了解流水线CPU基本功能部件的设计与实现方法，

2) 了解提高CPU性能的方法。

3) 掌握流水线MIPS微处理器的工作原理。

4) 理解数据冒险、控制冒险的概念以及流水线冲突的解决方法。

5) 掌握流水线MIPS微处理器的测试方法。

### 2 实验要求

至少支持add、sub、and、or、addi、andi、ori、lw、sw、beq、bne十一条指令。

采用5级流水线技术（可选:具有数据前推机制）,寄存器堆的写操作提前半个时钟周期，下降沿读取，上升沿写入数据。

## 二、课程设计器材

1 硬件平台 基于 Xilinx FPGA basys3 实验平台 开发系统中实现MIPS微处理器。

2 软件平台

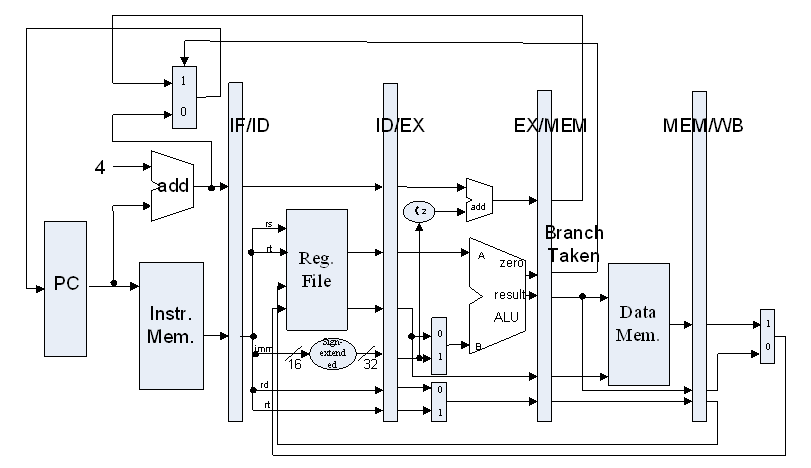
操作系统：Win 10。

开发平台：Vivado 2015.2。

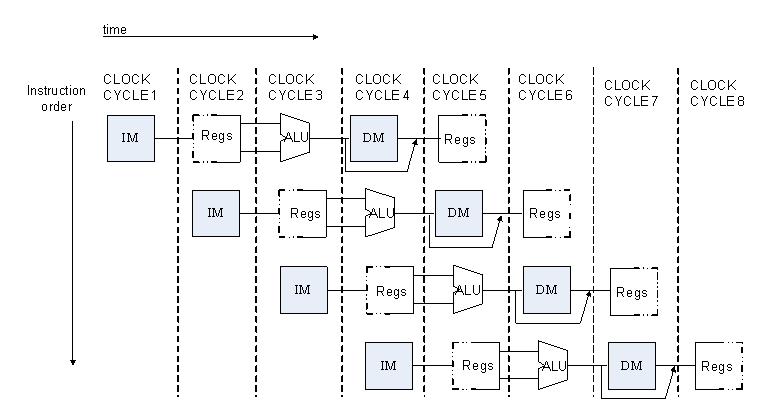
编程语言：VerilogHDL 硬件描述语言。

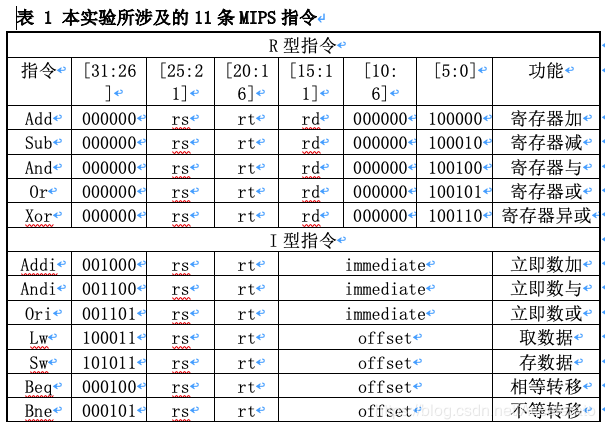
## 三、CPU逻辑设计总体方案

流水线是数字系统中一种提高系统稳定性和工作速度的方法，广泛应用于现代CPU的架构中。根据MIPS处理器的特点，将整体的处理过程分为取指令（IF）、指令译码（ID）、执行（EX）、存储器访问（MEM）和寄存器会写（WB）五级，对应多周期的五个处理阶段。一个指令的执行需要5个时钟周期，每个时钟周期的上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级处理。  
  
流水线设计参考如下结构图（没有冒险处理）：

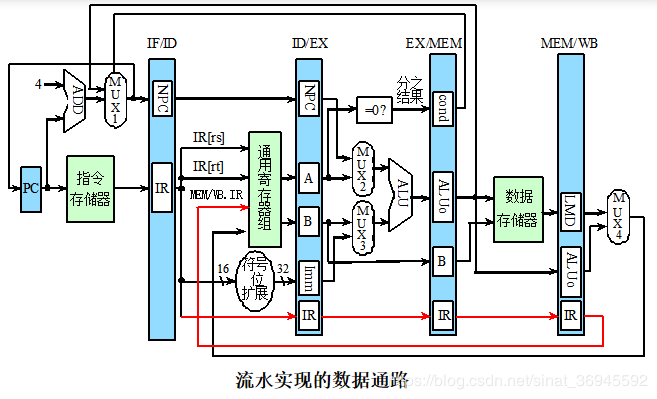


指令执行过程参考图





## 四、实验过程和数据记录



//级间寄存器

module flipflop(clk,reset,in,out);

parameter WIDTH=1;//根据需要改宽度

input clk;//

input reset;

output [WIDTH-1:0] out;//

reg[WIDTH-10] out;

always@(posedeg clk)

if(reset)

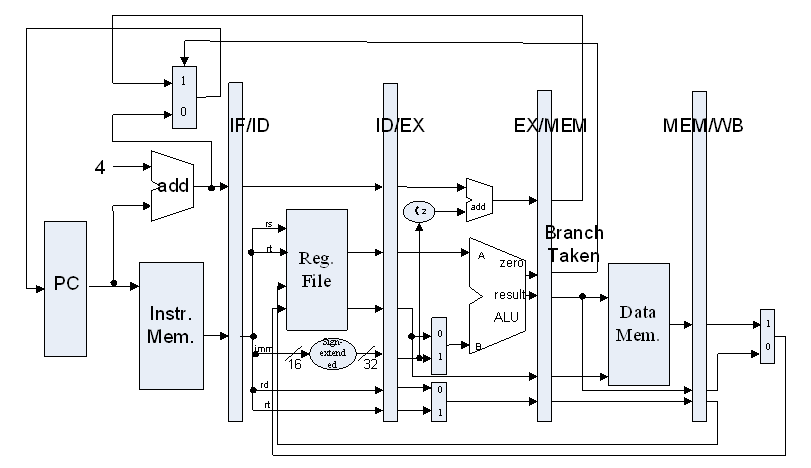
out<={WIDTH{1’b0}};

else

out<=in;

endmodule

### 1.取指令部分（IF）



IF/ID.IR←Mem[PC]（将PC的指令取出放入指令寄存器表示当前运行的指令）

NPC←PC+4（PC指向下一条指令（32位加4））

PC模块功能

实现思路  
由于PC是32位，所以增加一个32位加法器，固定与32位的立即数4进行相加，PC+4结果在时钟信号的上升沿更新写进PC寄存器。

主要实现代码  
1.PCAdd4

module adder\_if(a,b, c);

input [31:0] a,b;//偏移量

output [31:0] PCadd4;//新指令地址

assign c=a+b;

endmodule

2.指令存储器

INSTMEM 参照单周期CPU

MUX

REG\_ifid

参考代码

module IF(clk,reset,branch\_or\_pc,

branch\_addr,next\_pc\_if,inst\_if,

pc

);

input clk;

input reset;

input branch\_or\_pc;//Branch&ALU\_zero

input[31:0] branch\_addr;//Branch跳转地址

output[31:0] next\_pc\_if;//pc+4

output[31:0] inst\_if;//从ROM中读的指令

output[31:0] pc;

//PC的多选器

reg[31:0] pc\_in;//pc选择

always@(\*)

begin

case(branch\_or\_pc)

1'b0:pc\_in<=next\_pc\_if;//没有分支也没有jump

1'b1:pc\_in<=branch\_addr;//有Branch

endcase

end

//PC寄存器

reg[31:0] pc;

always@(posedge clk)

begin

if(reset) pc<=32'b0;//复位

else pc<=pc\_in;

end

//计算下一个PC的加法器

adder\_if adder32\_bits\_if(

.a(pc),

.b(32'b00000000000000000000000000000100),

.c(next\_pc\_if)

);

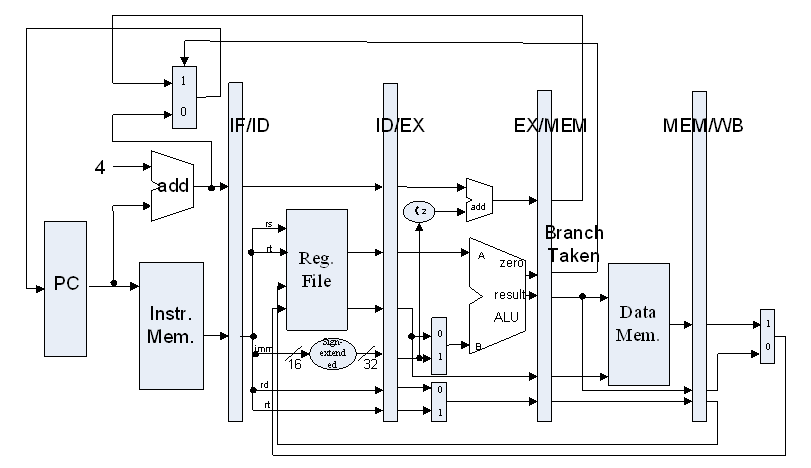
//指令ROM

InstructionROM InstructionROM(.a(pc[11:2]),.spo(inst\_if));

endmodule

其中next\_pc\_if(NPC) 和 inst\_if(IR) 需要送到下个阶段

 2) Design ID



    ID means "instruction fetch". This module includes the CPU controller, which is responsible for decoding, and register files. The codes of MEM are as follow:

module ID(clk,reset,inst\_id,

RegWrite\_wb,RegWriteAddr\_wb,RegWriteData\_wb,

RegDst\_id,MemtoReg\_id,RegWrite\_id,

MemWrite\_id,MemRead\_id,ALUCode\_id,

ALUSrcB\_id,Branch\_id,

Imm\_id,RsData\_id,RtData\_id,

RtAddr\_id,RdAddr\_id

);

input clk;

input reset;

input[31:0] inst\_id;//IF给的指令

//WB级的输入

input RegWrite\_wb;

input[4:0] RegWriteAddr\_wb;

input[31:0] RegWriteData\_wb;

//八个信号输出

output RegWrite\_id;

output RegDst\_id;

output MemRead\_id;

output MemWrite\_id;

output ALUSrcB\_id;

output Branch\_id;

output MemtoReg\_id;

output[2:0] ALUCode\_id;

//其他输出

output[31:0] Imm\_id;//符号拓展

output[31:0] RsData\_id;//寄存器堆输出1

output[31:0] RtData\_id;//寄存器堆输出2

output[4:0] RtAddr\_id;//rt

output[4:0] RdAddr\_id;//rd

assign RtAddr\_id=inst\_id[20:16];//rt

assign RdAddr\_id=inst\_id[15:11];//rd

assign Imm\_id={{16{inst\_id[15]}},inst\_id[15:0]};//符号扩展成32位立即数

/\*控制模块\*/

CtrlUnit CtrlUnit(

//输入

.inst(inst\_id),

//输出

.RegWrite(RegWrite\_id),.RegDst(RegDst\_id),

.Branch(Branch\_id),.MemRead(MemRead\_id),.MemWrite(MemWrite\_id),

.ALUCode(ALUCode\_id),.ALUSrc\_B(ALUSrcB\_id),

.MemtoReg(MemtoReg\_id)

);

/\*寄存器堆模块\*/

RegisterFiles RegisterFiles(

//输入，由WB级来提供

.clk(clk),.rst(reset),.L\_S(RegWrite\_wb),

.R\_addr\_A(inst\_id[25:21]),.R\_addr\_B(inst\_id[20:16]),

.Wt\_addr(RegWriteAddr\_wb),.wt\_data(RegWriteData\_wb),

//输出

.rdata\_A(RsData\_id),.rdata\_B(RtData\_id)

);

endmodule

 The following shows the design details for control unit:

    module CtrlUnit(inst,RegWrite,RegDst,

Branch,MemRead,

MemWrite,ALUCode,

ALUSrc\_B,

MemtoReg

);

input[31:0] inst;

output RegWrite;

output RegDst;

output Branch;

output MemRead;

output MemWrite;

output[2:0] ALUCode;

output ALUSrc\_B;

output MemtoReg;//1:来自mem

wire[5:0] op;

wire[5:0] func;

wire[4:0] rt;

assign op=inst[31:26];//op字段

assign func=inst[5:0];//func字段

//R指令

parameter R\_type\_op=6'b000000;

parameter ADD\_func=6'b100000;

parameter AND\_func=6'b100100;

parameter XOR\_func=6'b100110;

parameter OR\_func=6'b100101;

parameter NOR\_func=6'b100111;

parameter SUB\_func=6'b100010;

//R\_type

wire ADD,AND,NOR,OR,SUB,XOR,R\_type;

assign ADD=(op==R\_type\_op)&&(func==ADD\_func);

assign AND=(op==R\_type\_op)&&(func==AND\_func);

assign NOR=(op==R\_type\_op)&&(func==NOR\_func);

assign OR=(op==R\_type\_op)&&(func==OR\_func);

assign SUB=(op==R\_type\_op)&&(func==SUB\_func);

assign XOR=(op==R\_type\_op)&&(func==XOR\_func);

assign R\_type=ADD||AND||NOR||OR||SUB||XOR;

//Branch

parameter BEQ\_op=6'b000100;

parameter BNE\_op=6'b000101;

wire BEQ,BNE,Branch;

assign BEQ=(op==BEQ\_op);

assign BNE=(op==BNE\_op);

assign Branch=BEQ||BNE;

// I\_type instruction decode

parameter ADDI\_op=6'b001000;

parameter ANDI\_op=6'b001100;

parameter XORI\_op=6'b001110;

parameter ORI\_op=6'b001101;

wire ADDI,ANDI,XORI,ORI,I\_type;

assign ADDI=(op== ADDI\_op);

assign ANDI=(op==ANDI\_op);

assign XORI=(op==XORI\_op);

assign ORI=(op==ORI\_op);

assign I\_type=ADDI||ANDI||XORI||ORI;

// SW ,LW instruction decode

parameter SW\_op=6'b101011;

parameter LW\_op=6'b100011;

wire SW,LW;

assign SW=(op==SW\_op);

assign LW=(op==LW\_op);

// Control Singal

assign RegWrite=LW||R\_type||I\_type;//要写寄存器

assign RegDst=R\_type;//RegDst=1，选择rd，只有R指令这样

assign MemWrite=SW;

assign MemRead=LW;

assign MemtoReg=LW;

assign ALUSrc\_B=LW||SW||I\_type;

// ALUCode

//自己定义的，只要能在ALU里对应的上就行

parameter alu\_add=3'b010;

parameter alu\_sub=3'b110;

parameter alu\_and=3'b000;

parameter alu\_or=3'b001;

parameter alu\_xor=3'b011;

parameter alu\_nor=3'b100;

reg[2:0] ALUCode;

always@(\*)begin

if(op==R\_type\_op)begin

case(func)

ADD\_func: ALUCode<=alu\_add;

AND\_func: ALUCode<=alu\_and;

XOR\_func: ALUCode<=alu\_xor;

OR\_func: ALUCode<=alu\_or;

NOR\_func: ALUCode<=alu\_nor;

SUB\_func: ALUCode<=alu\_sub;

default: ALUCode<=alu\_add;

endcase

end

elsebegin

case(op)

BEQ\_op: ALUCode<=alu\_sub;

BNE\_op: ALUCode<=alu\_sub;

ADDI\_op: ALUCode<=alu\_add;

ANDI\_op: ALUCode<=alu\_and;

XORI\_op: ALUCode<=alu\_xor;

ORI\_op: ALUCode<=alu\_or;

SW\_op: ALUCode<=alu\_add;

LW\_op: ALUCode<=alu\_add;

default: ALUCode<=alu\_add;

endcase

end

end

endmodule

    The details for register files:

    module RegisterFiles(

input clk, rst, L\_S,

input[4:0] R\_addr\_A, R\_addr\_B, Wt\_addr,

input[31:0] wt\_data,

output[31:0] rdata\_A, rdata\_B

);

reg[31:0] register [1:31];

integer i;

assign rdata\_A=(R\_addr\_A==0)?0: register[R\_addr\_A];

assign rdata\_B=(R\_addr\_B==0)?0: register[R\_addr\_B];

always@(posedge clk orposedge rst)begin

if(rst==1)

for(i=1; i<32; i= i+1)

register[i]<=0;

elseif((Wt\_addr!=0)&&(L\_S==1))

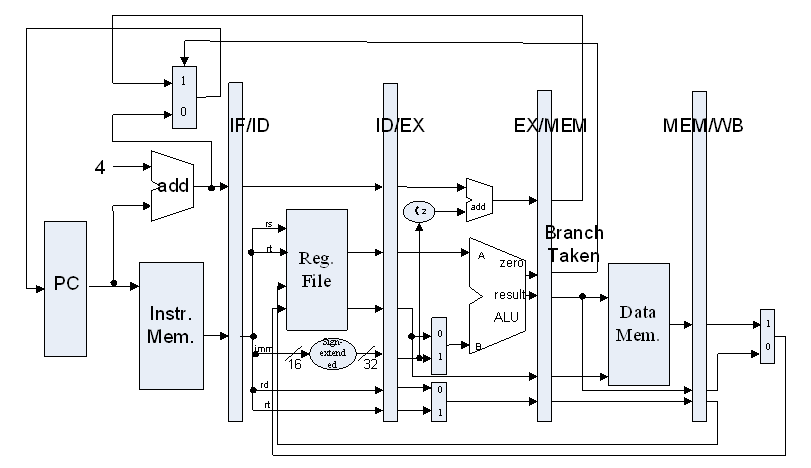
register[Wt\_addr]<= wt\_data;

end

endmodule

    The register files are the same as the previous (used inComputer Organization course)

3) Design EX



    EX means "execution". It contains ALU and an adder.

    The codes for EX module are shown below:

    module EX(clk,next\_pc\_ex,

ALUCode\_ex,ALUSrcB\_ex,

RegDst\_ex,

Imm\_ex,RsData\_ex,RtData\_ex,

RtAddr\_ex,RdAddr\_ex,

//输出

Branch\_addr\_ex,

alu\_zero\_ex,alu\_res\_ex,RegWriteAddr\_ex

);

input clk;

input[31:0] next\_pc\_ex;

input[2:0] ALUCode\_ex;

input ALUSrcB\_ex;

input RegDst\_ex;

input[31:0] Imm\_ex;

input[31:0] RsData\_ex;

input[31:0] RtData\_ex;

input[4:0] RtAddr\_ex;

input[4:0] RdAddr\_ex;

//

output[31:0] Branch\_addr\_ex;

output alu\_zero\_ex;

output[31:0] alu\_res\_ex;

outputreg[4:0] RegWriteAddr\_ex;

//分支地址

adder\_32bits adder\_32bits\_ex(.a(next\_pc\_ex),.b(Imm\_ex<<2),.c(Branch\_addr\_ex));

//ALUSrcB的多选器

reg[31:0] alu\_in;

always@(\*)begin

case(ALUSrcB\_ex)

1'b0:alu\_in<=RtData\_ex;//来自寄存器堆第二个输出

1'b1:alu\_in<=Imm\_ex;//来自符号扩展

endcase

end

//ALU

ALU ALU(.ALU\_operation(ALUCode\_ex),.A(RsData\_ex),.B(alu\_in),

.res(alu\_res\_ex),.zero(alu\_zero\_ex),.overflow()//overflow什么也不连

);

//写寄存器堆地址的多选器

always@(\*)begin

case(RegDst\_ex)

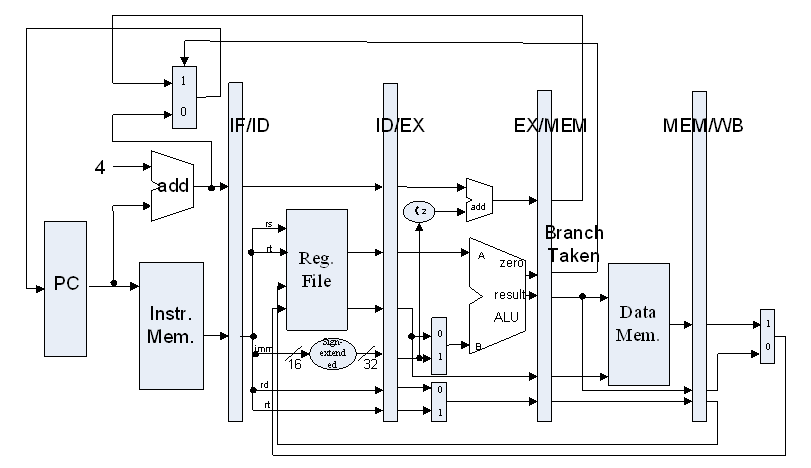
1'b0:RegWriteAddr\_ex<=RtAddr\_ex;//rt

1'b1:RegWriteAddr\_ex<=RdAddr\_ex;//rd

endcase

end

endmodule



  4) Design MEM

    module MEM(clk,MemRead\_mem,

MemWrite\_mem,Branch\_mem,alu\_zero\_mem,

alu\_res\_mem,RtData\_mem,

branch\_or\_pc\_mem,Dout\_mem

);

input clk;

//MemRead信号暂时不要了

input MemRead\_mem;

input MemWrite\_mem;

input Branch\_mem;

input alu\_zero\_mem;

input[31:0]alu\_res\_mem;

input[31:0] RtData\_mem;

output branch\_or\_pc\_mem;

output[31:0] Dout\_mem;

DataRAM DataRAM(

.clka(clk),//input clka

.wea(~MemRead\_mem&MemWrite\_mem),//input [0:0] wea

.addra(alu\_res\_mem[11:2]),//input [9 : 0] addra

.dina(RtData\_mem),//input [31:0] dina

.douta(Dout\_mem)//output [31:0] douta

);

//and模块，确定跳转信号

and\_1bit and\_1bit(.a(Branch\_mem),.b(alu\_zero\_mem),.c(branch\_or\_pc\_mem));

endmodule

    Also an adder should be included, which is serving for calculating the proper address of writing back.

    module adder\_32bits(

input[31:0] a,

input[31:0] b,

output[31:0] c

);

assign c= a+ b;

endmodule

    5) Design WB

    The stage is simple and can be done in top:

    /\*WB级\*/

//只有一个多选器，直接在顶层实现

//选择写回的内容

reg[31:0] reg\_data\_wb;

always@(\*)begin

case(MemtoReg\_wb)

1'b0:reg\_data\_wb<=alu\_res\_wb;//来自ALU

1'b1:reg\_data\_wb<=Dout\_wb;//来自RAM

endcase

end

    6) Design top module

    Finally we come to the top module, this module is actually for connecting lines between registers and different stages, which is shown as below:

    module MipsPipelineCPU(clk,reset,inst\_if,

alu\_res\_ex,Dout\_mem,

RtData\_id,PC\_out

);

//CPU模块输入：clk、reset

//CPU模块输出：PC地址、指令、ALU运算结果、寄存器堆的数据输出B、Memory结果

//这些数据都是一开始产生就传递给输出

input clk;//100Mhz

input reset;

output[31:0] inst\_if;//指令,送给顶层的data2

output[31:0] alu\_res\_ex;//ALU结果送给data4

output[31:0] Dout\_mem;//memory输出送给data6,就是图里的Data\_in

output[31:0] RtData\_id;//寄存器堆的输出B，送给data5，就是图里的Data\_out

output[31:0] PC\_out;//pc，送给data7

/\*IF级\*/

wire branch\_or\_pc\_mem;//本来是MEM级的！

wire[31:0] Branch\_addr\_mem;//本来是MEM级的！

wire[31:0] next\_pc\_if;

wire[31:0] inst\_if;

IF IF(

//输入

.clk(clk),

.reset(reset),

.branch\_or\_pc(branch\_or\_pc\_mem),//需要MEM的输入，branch\_or\_pc\_mem

.branch\_addr(Branch\_addr\_mem),//需要EX/MEM的输入

//输出

.next\_pc\_if(next\_pc\_if),

.inst\_if(inst\_if),

.pc(PC\_out)//当前pc

);

/\*IF-ID寄存器\*/

wire[31:0] next\_pc\_id;

wire[31:0] inst\_id;

flipflop#(.WIDTH(32))IF\_ID1(

.clk(clk),

.reset(reset),

.in(inst\_if),//送指令

.out(inst\_id)

);

flipflop#(.WIDTH(32))IF\_ID2(

.clk(clk),

.in(next\_pc\_if),//送pc+4

.reset(reset),

.out(next\_pc\_id)

);

//注意这里申明了WB级的东西：RegWrite和RegWriteAddr，有点混乱，写WB级注意不要重复！

wire[4:0] RtAddr\_id,RdAddr\_id;

wire RegWrite\_wb,MemtoReg\_id,RegWrite\_id,MemWrite\_id;

wire MemRead\_id,ALUSrcB\_id,RegDst\_id,Branch\_id;

wire[4:0] RegWriteAddr\_wb;

wire[2:0] ALUCode\_id;

wire[31:0] Imm\_id,RsData\_id,RtData\_id;

/\*ID级\*/

wire[31:0] RegWriteData\_wb;//WB级的东西，注意！

assign RegWriteData\_wb=reg\_data\_wb;

ID ID(.clk(clk),.reset(reset),.inst\_id(inst\_id),

.RegWrite\_wb(RegWrite\_wb),.RegWriteAddr\_wb(RegWriteAddr\_wb),

.RegWriteData\_wb(RegWriteData\_wb),//送进来的数据要经过选择，在WB命名为reg\_data\_wb！

.RegWrite\_id(RegWrite\_id),.RegDst\_id(RegDst\_id),.MemtoReg\_id(MemtoReg\_id),

.MemWrite\_id(MemWrite\_id),.MemRead\_id(MemRead\_id),

.ALUCode\_id(ALUCode\_id),.ALUSrcB\_id(ALUSrcB\_id),

.Branch\_id(Branch\_id),.Imm\_id(Imm\_id),.RsData\_id(RsData\_id),.RtData\_id(RtData\_id),

.RtAddr\_id(RtAddr\_id),.RdAddr\_id(RdAddr\_id));

/\*ID-EX级间寄存器\*/

//总共14根线

wire[4:0] RtAddr\_ex,RdAddr\_ex;

wire MemtoReg\_ex,RegWrite\_ex,MemWrite\_ex;

wire MemRead\_ex,ALUSrcB\_ex,RegDst\_ex,Branch\_ex;

wire[2:0] ALUCode\_ex;

wire[31:0] Imm\_ex,RsData\_ex,RtData\_ex,next\_pc\_ex;

flipflop#(.WIDTH(1))ID\_EX1(

.clk(clk),

.reset(reset),

.in(RegWrite\_id),//RegWrite

.out(RegWrite\_ex)

);

flipflop#(.WIDTH(1))ID\_EX2(

.clk(clk),

.reset(reset),

.in(RegDst\_id),//RegDst

.out(RegDst\_ex)

);

flipflop#(.WIDTH(1))ID\_EX3(

.clk(clk),

.reset(reset),

.in(MemRead\_id),//MemRead

.out(MemRead\_ex)

);

flipflop#(.WIDTH(1))ID\_EX4(

.clk(clk),

.reset(reset),

.in(MemWrite\_id),//MemWrite

.out(MemWrite\_ex)

);

flipflop#(.WIDTH(1))ID\_EX5(

.clk(clk),

.reset(reset),

.in(ALUSrcB\_id),//ALUSrcB\_id

.out(ALUSrcB\_ex)

);

flipflop#(.WIDTH(1))ID\_EX6(

.clk(clk),

.reset(reset),

.in(MemtoReg\_id),//MemtoReg

.out(MemtoReg\_ex)

);

flipflop#(.WIDTH(1))ID\_EX7(

.clk(clk),

.reset(reset),

.in(Branch\_id),//Branch

.out(Branch\_ex)

);

flipflop#(.WIDTH(3))ID\_EX8(//注意这里的宽度是3！

.clk(clk),

.reset(reset),

.in(ALUCode\_id),//ALUCode

.out(ALUCode\_ex)

);

flipflop#(.WIDTH(32))ID\_EX9(//注意是32位！

.clk(clk),

.reset(reset),

.in(next\_pc\_id),//pc+4

.out(next\_pc\_ex)

);

flipflop#(.WIDTH(32))ID\_EX10(

.clk(clk),

.reset(reset),

.in(RsData\_id),//寄存器堆A

.out(RsData\_ex)

);

flipflop#(.WIDTH(32))ID\_EX11(

.clk(clk),

.reset(reset),

.in(RtData\_id),//寄存器堆B

.out(RtData\_ex)

);

flipflop#(.WIDTH(32))ID\_EX12(

.clk(clk),

.reset(reset),

.in(Imm\_id),//Imm,符号拓展

.out(Imm\_ex)

);

flipflop#(.WIDTH(5))ID\_EX13(//注意宽度是5！

.clk(clk),

.reset(reset),

.in(RtAddr\_id),//rt

.out(RtAddr\_ex)

);

flipflop#(.WIDTH(5))ID\_EX14(

.clk(clk),

.reset(reset),

.in(RdAddr\_id),//rd

.out(RdAddr\_ex)

);

/\*EX级\*/

wire[31:0] Branch\_addr\_ex;

wire[31:0] alu\_res\_ex;

wire alu\_zero\_ex;

wire[4:0] RegWriteAddr\_ex;

EX EX(.clk(clk),.next\_pc\_ex(next\_pc\_ex),

.ALUCode\_ex(ALUCode\_ex),.ALUSrcB\_ex(ALUSrcB\_ex),

.RegDst\_ex(RegDst\_ex),

.Imm\_ex(Imm\_ex),.RsData\_ex(RsData\_ex),.RtData\_ex(RtData\_ex),

.RtAddr\_ex(RtAddr\_ex),.RdAddr\_ex(RdAddr\_ex),

//输出

.Branch\_addr\_ex(Branch\_addr\_ex),

.alu\_zero\_ex(alu\_zero\_ex),.alu\_res\_ex(alu\_res\_ex),

.RegWriteAddr\_ex(RegWriteAddr\_ex)

);

/\*EX-MEM级间寄存器\*/

wire RegWrite\_mem;

wire MemRead\_mem;

wire MemWrite\_mem;

wire MemtoReg\_mem;

wire[31:0] alu\_res\_mem;

wire alu\_zero\_mem;

wire[31:0] RtData\_mem;

wire[4:0] RegWriteAddr\_mem;

flipflop#(.WIDTH(1))EX\_MEM1(

.clk(clk),

.reset(reset),

.in(RegWrite\_ex),//RegWrite

.out(RegWrite\_mem)

);

flipflop#(.WIDTH(1))EX\_MEM2(

.clk(clk),

.reset(reset),

.in(MemRead\_ex),//MemRead

.out(MemRead\_mem)

);

flipflop#(.WIDTH(1))EX\_MEM3(

.clk(clk),

.reset(reset),

.in(MemWrite\_ex),//MemWrite

.out(MemWrite\_mem)

);

flipflop#(.WIDTH(1))EX\_MEM4(

.clk(clk),

.reset(reset),

.in(MemtoReg\_ex),//MemtoReg

.out(MemtoReg\_mem)

);

flipflop#(.WIDTH(1))EX\_MEM5(

.clk(clk),

.reset(reset),

.in(Branch\_ex),//Branch

.out(Branch\_mem)

);

flipflop#(.WIDTH(32))EX\_MEM6(//注意是32位！

.clk(clk),

.reset(reset),

.in(Branch\_addr\_ex),//Branch地址

.out(Branch\_addr\_mem)//注意这里送回IF级！

);

flipflop#(.WIDTH(32))EX\_MEM7(

.clk(clk),

.reset(reset),

.in(alu\_res\_ex),//alu结果

.out(alu\_res\_mem)

);

flipflop#(.WIDTH(1))EX\_MEM8(

.clk(clk),

.reset(reset),

.in(alu\_zero\_ex),//alu的零信号

.out(alu\_zero\_mem)

);

flipflop#(.WIDTH(32))EX\_MEM9(

.clk(clk),

.reset(reset),

.in(RtData\_ex),//RtData

.out(RtData\_mem)

);

flipflop#(.WIDTH(5))EX\_MEM10(

.clk(clk),

.reset(reset),

.in(RegWriteAddr\_ex),//写回地址

.out(RegWriteAddr\_mem)

);

/\*MEM级\*/

wire[31:0] Dout\_mem;

MEM MEM(

.clk(clk),.MemRead\_mem(MemRead\_mem),.MemWrite\_mem(MemWrite\_mem),

.Branch\_mem(Branch\_mem),

.alu\_zero\_mem(alu\_zero\_mem),

.alu\_res\_mem(alu\_res\_mem),.RtData\_mem(RtData\_mem),

.branch\_or\_pc\_mem(branch\_or\_pc\_mem),.Dout\_mem(Dout\_mem)//注意信号要往回送，给IF

);

/\*MEM-WB级间寄存器\*/

wire[31:0] Dout\_wb;

wire[31:0] alu\_res\_wb;

wire MemtoReg\_wb;

flipflop#(.WIDTH(1))MEM\_WB1(

.clk(clk),

.reset(reset),

.in(RegWrite\_mem),//RegWrite

.out(RegWrite\_wb)

);

flipflop#(.WIDTH(1))MEM\_WB2(

.clk(clk),

.reset(reset),

.in(MemtoReg\_mem),//MemtoReg

.out(MemtoReg\_wb)

);

flipflop#(.WIDTH(32))MEM\_WB3(//注意这里是32位

.clk(clk),

.reset(reset),

.in(Dout\_mem),//Dout，RAM的输出

.out(Dout\_wb)

);

flipflop#(.WIDTH(32))MEM\_WB4(

.clk(clk),

.reset(reset),

.in(alu\_res\_mem),//alu的结果

.out(alu\_res\_wb)

);

flipflop#(.WIDTH(5))MEM\_WB5(//注意是5位

.clk(clk),

.reset(reset),

.in(RegWriteAddr\_mem),//RegWriteAddr

.out(RegWriteAddr\_wb)

);

/\*WB级\*/

reg[31:0] reg\_data\_wb;

always@(\*)begin

case(MemtoReg\_wb)

1'b0:reg\_data\_wb<=alu\_res\_wb;//来自ALU

1'b1:reg\_data\_wb<=Dout\_wb;//来自RAM

endcase

end

endmodule

    8) Debug

    The step is a tradition after finishing the writing the codes. Simply run the program again and again and eliminate the bugs.

    9) Test

        a. The content of RAM

        memory\_initialization\_radix=16;

        memory\_initialization\_vector=

00000000, 00000000, 00000000, 00000000, 00000000, 00000002,

00000002, 00000000, 00000000, 00000000, 00000000, 00000000,

00000000, 00000000, 00000000, 00000000, 00000000, 00000000,

00000000, 00000000, 00000001, 00000004, 00000000, 00000000;

        b. Instructions in ROM

        memory\_initialization\_radix=16;

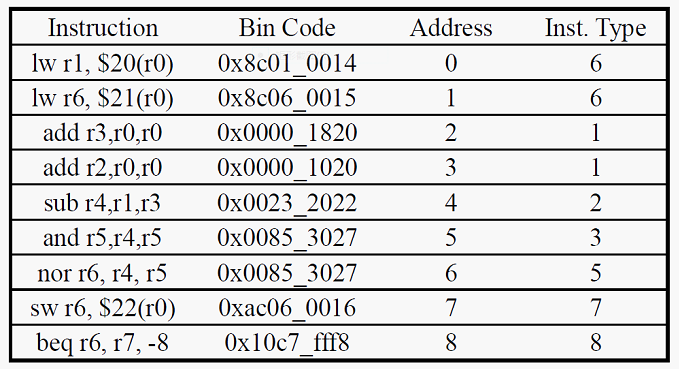
        memory\_initialization\_vector=

8c010014,8c020015,00221820,00001020,

00232022,00642824,00853027,ac060016,

10c7fff8;

            The corresponding instructions are:



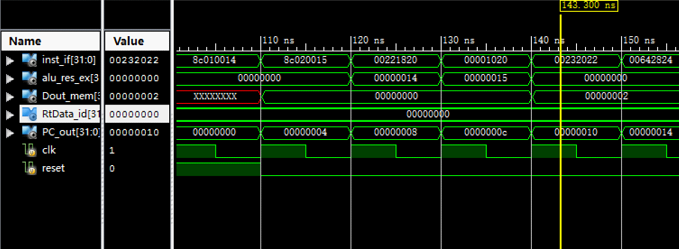
        c.

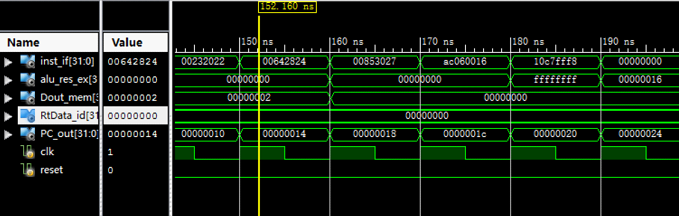
    10) Verify on the experiment box.

2. Data records

    1) Simulation

        The results for simulation are:





    From the picture we can see that:

    a. lw r1,$20(r0)

    The content of address 14H is 0, this instruction makes r1 get 2. And from the simulation result we can see that the output of ALU is 14H.

    b. lw r2,$21(r0)

    Notice that in address 15H, I put a 00000002 here. Therefore this instruction is supposed to make r2 2, but this does NOT happen until WB finishes. As a result, if any instruction trying to load the value of r2 before WB, the data hazard occurs!

    c. add r3,r1,r2

    Here comes the data hazard. Because r2 can't be written back until WB, the current value of r2 will still be 0, which leads the result of r1+r2 to 0. From the simulation result we can see that the result of ALU is 0 rather than 2, which proofs this point.

    d. add r2,r0,r0

    The output of ALU is 0.

    e. sub r4,r1,r3

    The output of ALU is still 0.

    f. and r5,r3,r4

    The output of ALU is still 0.

    g. nor r6,r4,r5

    Here because r4=0, r5=0, r6 should be ffffffff, as shown in the simulation diagram above.

    h. sw r6,$22(r0)

    Nothing special.

    i. beq r6,r7,-8

    The last instruction, notice that jump will NOT really happen.

    2)Pictures

    I took some pictures here, merely as a memorabilia.

四、实验结果分析

    Let's consider the test codes and test data:

        a. The content of RAM

        memory\_initialization\_radix=16;

        memory\_initialization\_vector=

00000000, 00000000, 00000000, 00000000, 00000000, 00000002,

00000002, 00000000, 00000000, 00000000, 00000000, 00000000,

00000000, 00000000, 00000000, 00000000, 00000000, 00000000,

00000000, 00000000, 00000001, 00000004, 00000000, 00000000;

        b. Instructions in ROM

        memory\_initialization\_radix=16;

        memory\_initialization\_vector=

8c010014,8c020015,00221820,00001020,

00232022,00642824,00853027,ac060016,

10c7fff8;

    We can proof that this is a pipeline CPU by observing the output of ALU. The idea is quite simple: we observe the output of ALU each time and compare it with our expectation. If any pipeline hazard occurs, the output will be different.

    Now let's analyses the experiment result instruction by instruction:

    a. lw r1,$20(r0)

    The content of address 14H is 0, this instruction makes r1 get 2. And from the simulation result we can see that the output of ALU is 14H.

    b. lw r2,$21(r0)

    Notice that in address 15H, I put a 00000002 here. Therefore this instruction is supposed to make r2 2, but this does NOT happen until WB finishes. As a result, if any instruction trying to load the value of r2 before WB, the data hazard occurs!

    c. add r3,r1,r2

    Here comes the data hazard. Because r2 can't be written back until WB, the current value of r2 will still be 0, which leads the result of r1+r2 to 0. From the simulation result we can see that the result of ALU is 0 rather than 2, which proofs this point.

    d. add r2,r0,r0

    The output of ALU is 0.

    e. sub r4,r1,r3

    The output of ALU is still 0.

    f. and r5,r3,r4

    The output of ALU is still 0.

    g. nor r6,r4,r5

    Here because r4=0, r5=0, r6 should be ffffffff, as shown in the simulation diagram above.

    h. sw r6,$22(r0)

    Nothing special.

    i. beq r6,r7,-8

    The last instruction, the output of ALU will be 0 (since r6=r7=0, 0-0=0).

五、讨论与心得

    The experiment is quite hard, which took me almost 3 weeks to complete. First, one should pay special attention to a questions: what need to be sent from the current stage to the next one? What's more, how to store them?

    For the first question, I got the answer from the schematic provided by the teacher. The answer to it is mentioned in the design principles. For the next question, I chose to use a flip-flop:

