实验六 CPU综合设计

1. **实验目的**

1 掌握复杂系统设计方法。

2 深刻理解计算机系统硬件原理。

**二、实验内容**

1）设计一个基于MIPS指令集的CPU，支持以下指令：{add, sub, addi, lw, sw, beq, j, nop}；

2）CPU需要包含寄存器组、RAM模块、ALU模块、指令译码模块；

3）该CPU能运行基本的汇编指令；（D~C+）

以下为可选内容：

4）实现多周期CPU（B-~B+）；

5）实现以下高级功能之一（A-~A+）：

(1)实现5级流水线CPU；

(2)实现超标量；

(3)实现4路组相联缓存；

可基于RISC V 、ARM指令集实现。

**如发现代码为抄袭代码，成绩一律按不及格处理。**

**三、实验要求**

编写相应测试程序，完成所有指令测试。

1. **实验代码及结果**

顶层模块：五级流水线

module Pipelined\_Processer(

input clk

);

//Fetch Instruction

reg[31:0] PC\_in;

wire[31:0] PC\_out;

wire StallF;

reg StallFReg;

wire StallD;

reg StallDReg;

wire[31:0] Instr\_in;

wire[31:0] Instr\_out;

reg[31:0] PCPlusF;

wire[31:0] PCPlusD;

reg PCSrc;

reg PCSrcD;

reg FlushEReg;

initial begin

PC\_in=0;

StallFReg=0;

StallDReg=0;

PCSrcD=0;

FlushEReg=0;

end

ProgramCounter PC (

.En(~StallFReg),

.clk(clk),

.PC\_in(PC\_in),

.PC\_out(PC\_out)

);

always @(\*)begin

PCPlusF=PC\_out+1;

end

InstructionMemory IM(

.Address(PC\_out),

.Instruction(Instr\_in)

);

FI\_ID PipeLinedReg1(

.clk(clk),

.En(~StallDReg),

.CLR(PCSrcD),

.Instr\_in(Instr\_in),

.PCPlus\_in(PCPlusF),

.Instr\_out(Instr\_out),

.PCPlus\_out(PCPlusD)

);

//Instruction Decode

reg PCBranchD;

wire RegWriteD;

wire MemtoRegD;

wire MemWriteD;

wire MemReadD;

wire[1:0] ALUOp;

wire[3:0] ALUcontrolD;

wire ALUSrcD;

wire RegDstD;

wire BranchD;

wire[31:0]ImmD;

wire[31:0] ReadData1D;

wire[31:0] ReadData2D;

/\* wire[4:0] RsD;

wire[4:0] RtD;

wire[4:0] RdD; \*/

wire ForwardAD;

wire ForwardBD;

wire[4:0] WriteRegW;

reg[31:0] ResultW;

CU\_completed CU(

.Opcode(Instr\_out[31:26]),

.RegDst(RegDstD),

.ALUSrc(ALUSrcD),

.MemtoReg(MemtoRegD),

.RegWrite(RegWriteD),

.MemWrite(MemWriteD),

.MemRead(MemReadD),

.Branch(BranchD),

.ALUOp(ALUOp)

);

ALUControlUnit ALUCU(

.Funct(Instr\_out[5:0]),

.ALUOp(ALUOp),

.ALUControl(ALUcontrolD)

);

RegFile rf(

.clk(clk),

.RegWrite(RegWriteW),

.ReadReg1(Instr\_out[25:21]),

.ReadReg2(Instr\_out[20:16]),

.WriteReg(WriteRegW),

.WriteData(ResultW),

.ReadData1(ReadData1D),

.ReadData2(ReadData2D)

);

SignExt exten(

.IR(Instr\_out[15:0]),

.SignExtImm(ImmD)

);

always @(\*) begin

PCBranchD=PCPlusD+ImmD;

end

wire FlushE;

wire RegWriteE;

wire MemToRegE;

wire MemWriteE;

wire MemReadE;

wire[3:0] ALUControlE;

wire ALUSrcE;

wire RegDstE;

wire[31:0] ReadData1E;

wire[31:0] ReadData2E;

wire[4:0] RsE;

wire[4:0] RtE;

wire[4:0] RdE;

wire[31:0] ImmE;

ID\_EX PipeLinedReg2(

.clk(clk),

.CLR(FlushEReg),

.RegWrite\_in(RegWriteD),

.MemToReg\_in(MemtoRegD),

.MemWrite\_in(MemWriteD),

.MemRead\_in(MemReadD),

.ALUControl\_in(ALUcontrolD),

.ALUSrc\_in(ALUSrcD),

.RegDst\_in(RegDstD),

.ReadData1\_in(ReadData1D),

.ReadData2\_in(ReadData2D),

.Rs\_in(Instr\_out[25:21]),

.Rt\_in(Instr\_out[20:16]),

.Rd\_in(Instr\_out[15:11]),

.Imm\_in(ImmD),

.RegWrite\_out(RegWriteE),

.MemToReg\_out(MemToRegE),

.MemWrite\_out(MemWriteE),

.MemRead\_out(MemReadE),

.ALUControl\_out(ALUControlE),

.ALUSrc\_out(ALUSrcE),

.RegDst\_out(RegDstE),

.ReadData1\_out(ReadData1E),

.ReadData2\_out(ReadData2E),

.Rs\_out(RsE),

.Rt\_out(RtE),

.Rd\_out(RdE),

.Imm\_out(ImmE)

);

//Excute stage

wire[1:0] ForwardAE;

wire[1:0] ForwardBE;

reg[31:0] SrcAE;

reg[31:0] SrcBE;

reg[31:0] WriteDataE;

wire[31:0] ALUResultE;

reg[4:0] WriteRegE;

ALU\_noclk alu(

.A(SrcAE),

.B(SrcBE),

.ALUControl(ALUControlE),

.Result(ALUResultE)

);

wire RegWriteM;

wire MemToRegM;

wire MemWriteM;

wire MemReadM;

wire[31:0] ALUOutM;

wire[31:0] WriteDataM;

wire[4:0] WriteRegM;

EX\_MA PipeLinedReg3(

.clk(clk),

.RegWrite\_in(RegWriteE),

.MemToReg\_in(MemToRegE),

.MemWrite\_in(MemWriteE),

.MemRead\_in(MemReadE),

.ALUResult\_in(ALUResultE),

.WriteData\_in(WriteDataE),

.WriteReg\_in(WriteRegE),

.RegWrite\_out(RegWriteM),

.MemToReg\_out(MemToRegM),

.MemWrite\_out(MemWriteM),

.MemRead\_out(MemReadM),

.ALUResult\_out(ALUOutM),

.WriteData\_out(WriteDataM),

.WriteReg\_out(WriteRegM)

);

wire[31:0] ReadDataM;

DataMemory DM(

.ReadData(ReadDataM),

.Address(ALUOutM),

.MemRead(MemReadM),

.MemWrite(MemWriteM),

.WriteData(WriteDataM)

);

//Write Back

wire MemToRegW;

wire[31:0] ReadDataW;

wire[31:0] ALUOutW;

MA\_WB PipeLinedReg4(

.clk(clk),

.RegWrite\_in(RegWriteM),

.MemToReg\_in(MemToRegM),

.ReadData\_in(ReadDataM),

.ALUResult\_in(ALUOutM),

.WriteReg\_in(WriteRegM),

.RegWrite\_out(RegWriteW),

.MemToReg\_out(MemToRegW),

.ReadData\_out(ReadDataW),

.ALUResult\_out(ALUOutW),

.WriteReg\_out(WriteRegW)

);

HazardControlUnit HazardHandle(

.RegWriteM(RegWriteM),

.RegWriteW(RegWriteW),

.RegWriteE(RegWriteE),

.WriteRegM(WriteRegM),

.WriteRegW(WriteRegW),

.WriteRegE(WriteRegE),

.BranchD(BranchD),

.RsE(RsE),

.RtE(RtE),

.ForwardAE(ForwardAE),

.ForwardBE(ForwardBE),

.MemToRegE(MemToRegE),

.MemToRegM(MemToRegM),

.RsD(Instr\_out[25:21]),

.RtD(Instr\_out[20:16]),

.ForwardAD(ForwardAD),

.ForwardBD(ForwardBD),

.StallF(StallF),

.StallD(StallD),

.FlushE(FlushE)

);

always @(\*) begin

StallFReg=StallF;

StallDReg=StallD;

FlushEReg=FlushE;

PCSrcD=PCSrc;

end

always @(\*) begin

if (PCSrc) PC\_in=PCBranchD;

else PC\_in=PCPlusF;

end

reg[31:0] cmpA;

reg[31:0] cmpB;

always @(\*) begin

if (ForwardAD) cmpA=ALUOutM;

else cmpA=ReadData1D;

end

always @(\*) begin

if (ForwardBD) cmpB=ALUOutM;

else cmpB=ReadData2D;

end

always @(\*) begin

PCSrc=BranchD&(cmpA==cmpB);

end

always @(\*) begin

if (RegDstE) WriteRegE=RdE;

else WriteRegE=RtE;

end

always @(\*) begin

if (MemToRegW) ResultW=ReadDataW;

else ResultW=ALUOutW;

end

always @(\*) begin

case (ForwardAE)

2'b00:SrcAE=ReadData1E;

2'b01:SrcAE=ResultW;

2'b10:SrcAE=ALUOutM;

default:SrcAE=ReadData1E;

endcase

end

always @(\*) begin

case (ForwardBE)

2'b00:WriteDataE=ReadData2E;

2'b01:WriteDataE=ResultW;

2'b10:WriteDataE=ALUOutM;

default:WriteDataE=ReadData2E;

endcase

end

always @(\*) begin

if (ALUSrcE) SrcBE=ImmE;

else SrcBE=WriteDataE;

end

endmodule

程序计数器PC：

module ProgramCounter(

input En,

input clk,

input [31:0] PC\_in,

output reg[31:0] PC\_out

);

always @(posedge clk) begin

if (En) PC\_out<=PC\_in;

end

endmodule

指令存储器：

module InstructionMemory(

Address,Instruction

);

parameter Instruction\_Width=32;

parameter IMEM\_Size=64;

input [31:0]Address;

output [Instruction\_Width-1:0]Instruction;

reg [31:0] ROM[IMEM\_Size-1:0];

integer i=0;

initial begin

for (i=10;i<IMEM\_Size;i=i+1) ROM[i]=i;

end

initial begin

ROM[0] = 32'h34080005; // ori $0, $8, 5

ROM[1] = 32'h3409000A; // ori $0, $9, 10

ROM[2] = 32'h01095021; // addu $t8, $t9, $18

ROM[3] = 32'h01085823; // subu $8, $8, $11

ROM[4] = 32'hAC080000; // sw $0, 0($8)

ROM[5] = 32'h8C180000; // lw $0, 0($24)

ROM[6] = 32'h10000007; // beq $0, $t0, label=7

ROM[7] = 32'h3C0A1234; // lui $10, 0x1234

ROM[8] = 32'h00000000; // nop

ROM[9] = 32'h00000000; // nop (label 后续指令)

end

assign Instruction=ROM[Address];

endmodule

这里用于测试的8条指令依次进行的操作为：

指令0：将0寄存器值与5或运算，存入8寄存器；

指令1：将0寄存器值与10或运算，存入9寄存器；

指令2：将8寄存器和9寄存器值相加，存入18寄存器；

指令3：将8寄存器与8寄存器相减，存入11寄存器；

指令4：将8寄存器值存入0寄存器值加上0偏移量指向的存储器单元中；

指令5：将0寄存器值加上0偏移量指向的存储单元值取入24寄存器中；

指令6：比较0寄存器与0寄存器内容，如果相等跳转执行指令7；

指令7：将立即数0x1234存入10寄存器中。

控制模块，包括控制信号单元CU和ALU控制信号单元ALUCU

module CU\_completed(

//input clk,

input [5:0] Opcode,

output reg RegDst,

output reg ALUSrc,

output reg MemtoReg,

output reg RegWrite,

output reg MemRead,

output reg MemWrite,

output reg Branch,

output reg [1:0] ALUOp

);

always @(\*) begin

case (Opcode)

6'b000000: begin // R型指令

RegDst = 1;

ALUSrc = 0;

MemtoReg = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

Branch = 0;

//Jump=0;

ALUOp = 2'b10;

end

6'b100011: begin // LW

RegDst = 0;

ALUSrc = 1;

MemtoReg = 1;

RegWrite = 1;

MemRead = 1;

MemWrite = 0;

Branch = 0;

//Jump=0;

ALUOp = 2'b00;

end

6'b101011: begin // SW

RegDst = 0; // 无意义

ALUSrc = 1;

MemtoReg = 0; // 无意义

RegWrite = 0;

MemRead = 0;

MemWrite = 1;

Branch = 0;

//Jump=0;

ALUOp = 2'b00;

end

6'b000100: begin // BEQ

RegDst = 0; // 无意义

ALUSrc = 0;

MemtoReg = 0; // 无意义

RegWrite = 0;

MemRead = 0;

MemWrite = 0;

Branch = 1;

//Jump=0;

ALUOp = 2'b01;

end

6'b001101: begin // ORI

RegDst = 0;

ALUSrc = 1;

MemtoReg = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

Branch = 0;

//Jump=0;

ALUOp = 2'b11;

end

6'b001111: begin // LUI

RegDst = 0;

ALUSrc = 1;

MemtoReg = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

Branch = 0;

//Jump=0;

ALUOp = 2'b00; // ALU操作不重要

end

6'b000010: begin // jump

RegDst = 0;

ALUSrc = 0;

MemtoReg = 0;

RegWrite = 0;

MemRead = 0;

MemWrite = 0;

Branch = 0;

//Jump=1;

ALUOp = 2'b00; // ALU操作不重要

end

default: begin // NOP或未知指令

RegDst = 0;

ALUSrc = 0;

MemtoReg = 0;

RegWrite = 0;

MemRead = 0;

MemWrite = 0;

Branch = 0;

//Jump=0;

ALUOp = 2'b00;

end

endcase

end

endmodule

module ALUControlUnit(

Funct,ALUOp,ALUControl

);

input [5:0] Funct;

input [1:0] ALUOp;

output reg [3:0] ALUControl;

always @(\*)

case(ALUOp)

2'b00:ALUControl<=4'b0100;//add

2'b01:ALUControl<=4'b0101;//sub

2'b11:ALUControl<=4'b0001;//or

default:case(Funct)

6'b100001:ALUControl<=4'b0100;//加法

6'b100011:ALUControl<=4'b0101;//减法

6'b000010:ALUControl<=4'b0000;//与

6'b000011:ALUControl<=4'b0001;//or

6'b000100:ALUControl<=4'b0010;//xor

default: ALUControl<=4'bxxxx;

endcase

endcase

endmodule

寄存器堆rf：

module RegFile(

clk,RegWrite,ReadReg1,ReadReg2,WriteReg,WriteData,ReadData1,ReadData2

);

parameter ADDR\_SIZE=5;

input clk,RegWrite;

input [ADDR\_SIZE-1:0] ReadReg1,ReadReg2,WriteReg;

input [`DATA\_WIDTH-1:0] WriteData;

output wire[`DATA\_WIDTH-1:0]ReadData1,ReadData2;

reg [`DATA\_WIDTH-1:0] rf[2\*\*ADDR\_SIZE-1:0];

integer i;

integer first=1;

initial begin

rf[0]=0;

for (i=1;i<2\*\*ADDR\_SIZE;i=i+1) rf[i]=first<<i;

end

always @(negedge clk) begin

if (RegWrite) rf[WriteReg]=WriteData;

end

assign ReadData1=(ReadReg1!=0)? rf[ReadReg1]:0;

assign ReadData2=(ReadReg2!=0)? rf[ReadReg2]:0;

/\*always @(posedge clk) begin

if (ReadReg1!=0) ReadData1=rf[ReadReg1];

else ReadData1<=0;

if (ReadReg2!=0) ReadData2=rf[ReadReg2];

else ReadData2<=0;

end\*/

endmodule

立即数符号扩展：

module SignExt(

input[15:0] IR,

output reg[31:0] SignExtImm

);

always @(\*) begin

SignExtImm={{16{IR[15]}}, IR[15:0]};

end

endmodule

ALU

module ALU\_noclk(

input [31:0]A,

input [31:0]B,

input [3:0]ALUControl,

output reg[31:0] Result

//output Zero

);

always @(\*) begin

case(ALUControl)

4'b0100:Result<=A+B;//add

4'b0101:Result<=A-B;//sub

4'b0001:Result<=A|B;//or

4'b0000:Result<=A&B;//and

4'b0010:Result<=~(A|B);//xor

default:Result<=32'bx;

endcase

end

//assign Zero=(Result==0)? 1:0;

Endmodule

数据存储器：

module DataMemory(

ReadData,Address,MemRead,MemWrite,WriteData

);

parameter Addr\_Width=12;

parameter Data\_Width=32;

output reg [Data\_Width-1:0] ReadData;

input [Addr\_Width-1:0] Address;

input MemRead;

input MemWrite;

input [Data\_Width-1:0] WriteData;

integer i;

reg [Data\_Width-1:0] RAM[2\*\*Addr\_Width-1:0];

initial begin

for (i=0;i<2\*\*Addr\_Width;i=i+1) RAM[i]=i;

end

always @(\*) begin

if (MemRead) ReadData=RAM[Address];

if (MemWrite) RAM[Address]=WriteData;

end

endmodule

这里初始化每个存储单元的值与地址相同。

险象控制单元：

module HazardControlUnit(

input RegWriteM,

input RegWriteW,

input RegWriteE,

input[4:0] WriteRegM,

input[4:0] WriteRegW,

input[4:0] WriteRegE,

input BranchD,

input[4:0] RsE,

input[4:0] RtE,

output reg[1:0] ForwardAE,

output reg[1:0] ForwardBE,

input MemToRegE,

input MemToRegM,

input[4:0] RsD,

input[4:0] RtD,

output reg ForwardAD,ForwardBD,

output reg StallF,StallD,FlushE

);

reg lwstall;

reg branhstall;

always @(\*) begin

if ((RsE!=0)&(RsE==WriteRegM)& RegWriteM) ForwardAE=10;//源操作数是上条指令计算结果

else if ((RsE!=0)&(RsE==WriteRegW)& RegWriteW) ForwardAE=01;//源操作数是上上条指令计算结果

else ForwardAE=0;

end

always @(\*) begin

if ((RtE!=0)&(RtE==WriteRegM)& RegWriteM) ForwardBE=10;//源操作数是上条指令计算结果

else if ((RtE!=0)&(RtE==WriteRegW)& RegWriteW) ForwardBE=01;//源操作数是上上条指令计算结果

else ForwardBE=00;

end

always @(\*) begin

ForwardAD=(RsD!=0) & (RsD==WriteRegM) & RegWriteM;

ForwardBD=(RtD!=0) & (RtD==WriteRegM) & RegWriteM;

end

always @(\*) begin

lwstall=((RsD==RsE) | (RtD==RtE)) & MemToRegE;

branhstall=(BranchD & RegWriteE & (WriteRegE==RsD | WriteRegE==RtD)) |

(BranchD & MemToRegM & (WriteRegM==RsD | WriteRegM==RtD));

StallD=lwstall | branhstall;

StallF=lwstall | branhstall;

FlushE=lwstall | branhstall;

end

endmodule

主要实现避免数据冒险和控制冒险。对于数据冒险，如果ALU输入或者从寄存器读出的值在之前的指令中被修改但流水线还没有及时写回，则设置旁路并利用多路选择器进行选择，当检测到此冒险时，寄存器读出的值或者ALU输入的值将会是还没写回的计算结果；否则正常读取寄存器或是ALU输入即可；对于控制冒险，在译码阶段即进行跳转的检测，一旦检测需要跳转则会更改下一个译码阶段要译码的指令。

各流水线阶段间的流水线寄存器，共4个，分别位于取值/译码之间、译码/执行之间、执行/访存之间、访存/写回之间。在时钟上升沿将存储的状态输出给下一级流水阶段，下降沿存入新的指令状态：

module FI\_ID(

input clk,

input En,

input CLR,

input[31:0] Instr\_in,

input[31:0] PCPlus\_in,

output reg[31:0] Instr\_out,

output reg[31:0] PCPlus\_out

);

reg[31:0] Instr;

reg[31:0] PCPlus;

always @(posedge clk) begin

Instr\_out=Instr;

PCPlus\_out=PCPlus;

end

always @(negedge clk) begin

if (CLR)

begin

Instr=0;

PCPlus=0;

end

if (En)

begin

Instr=Instr\_in;

PCPlus=PCPlus\_in;

end

end

endmodule

module ID\_EX(

input clk,

input CLR,

input RegWrite\_in,

input MemToReg\_in,

input MemWrite\_in,

input MemRead\_in,

input[3:0] ALUControl\_in,

input ALUSrc\_in,

input RegDst\_in,

input[31:0] ReadData1\_in,

input[31:0] ReadData2\_in,

input[4:0] Rs\_in,

input[4:0] Rt\_in,

input[4:0] Rd\_in,

input[31:0] Imm\_in,

output reg RegWrite\_out,

output reg MemToReg\_out,

output reg MemRead\_out,

output reg MemWrite\_out,

output reg[3:0] ALUControl\_out,

output reg ALUSrc\_out,

output reg RegDst\_out,

output reg[31:0] ReadData1\_out,

output reg[31:0] ReadData2\_out,

output reg[4:0] Rs\_out,

output reg[4:0] Rt\_out,

output reg[4:0] Rd\_out,

output reg[31:0] Imm\_out

);

reg RegWrite;

reg MemToReg;

reg MemWrite;

reg MemRead;

reg[3:0] ALUControl;

reg ALUSrc;

reg RegDst;

reg[31:0] ReadData1;

reg[31:0] ReadData2;

reg[4:0] Rs;

reg[4:0] Rt;

reg[4:0] Rd;

reg[31:0] Imm;

always @(posedge clk) begin

RegWrite\_out=RegWrite;

MemToReg\_out=MemToReg;

MemWrite\_out=MemWrite;

MemRead\_out=MemRead;

ALUControl\_out=ALUControl;

ALUSrc\_out=ALUSrc;

RegDst\_out=RegDst;

ReadData1\_out=ReadData1;

ReadData2\_out=ReadData2;

Rs\_out=Rs;

Rt\_out=Rt;

Rd\_out=Rd;

Imm\_out=Imm;

end

always @(negedge clk) begin

if (CLR) begin

RegWrite=0;

MemToReg=0;

MemWrite=0;

MemRead=0;

ALUControl=0;

ALUSrc=0;

RegDst=0;

ReadData1=0;

ReadData2=0;

Rs=0;

Rt=0;

Rd=0;

Imm=0;

end

else begin

#25

RegWrite=RegWrite\_in;

MemToReg=MemToReg\_in;

MemWrite=MemWrite\_in;

MemRead=MemRead\_in;

ALUControl=ALUControl\_in;

ALUSrc=ALUSrc\_in;

RegDst=RegDst\_in;

ReadData1=ReadData1\_in;

ReadData2=ReadData2\_in;

Rs=Rs\_in;

Rt=Rt\_in;

Rd=Rd\_in;

Imm=Imm\_in;

end

end

endmodule

module EX\_MA(

input clk,

input RegWrite\_in,

input MemToReg\_in,

input MemWrite\_in,

input MemRead\_in,

input[31:0] ALUResult\_in,

input[31:0] WriteData\_in,

input[4:0] WriteReg\_in,

output reg RegWrite\_out,

output reg MemToReg\_out,

output reg MemWrite\_out,

output reg MemRead\_out,

output reg[31:0] ALUResult\_out,

output reg[31:0] WriteData\_out,

output reg[4:0] WriteReg\_out

);

reg RegWrite;

reg MemToReg;

reg MemWrite;

reg MemRead;

reg[31:0] ALUResult;

reg[31:0] WriteData;

reg[4:0] WriteReg;

always @(posedge clk) begin

RegWrite\_out=RegWrite;

MemToReg\_out=MemToReg;

MemWrite\_out=MemWrite;

MemRead\_out=MemRead;

ALUResult\_out=ALUResult;

WriteData\_out=WriteData;

WriteReg\_out=WriteReg;

end

always @(negedge clk) begin

RegWrite=RegWrite\_in;

MemToReg=MemToReg\_in;

MemWrite=MemWrite\_in;

MemRead=MemRead\_in;

ALUResult=ALUResult\_in;

WriteData=WriteData\_in;

WriteReg=WriteReg\_in;

end

endmodule

module MA\_WB(

input clk,

input RegWrite\_in,

input MemToReg\_in,

input[31:0] ReadData\_in,

input[31:0] ALUResult\_in,

input[4:0] WriteReg\_in,

output reg RegWrite\_out,

output reg MemToReg\_out,

output reg[31:0] ReadData\_out,

output reg[31:0] ALUResult\_out,

output reg[4:0] WriteReg\_out

);

reg RegWrite;

reg MemToReg;

reg[31:0] ReadData;

reg[31:0] ALUResult;

reg[4:0] WriteReg;

always @(posedge clk) begin

RegWrite\_out=RegWrite;

MemToReg\_out=MemToReg;

ReadData\_out=ReadData;

ALUResult\_out=ALUResult;

WriteReg\_out=WriteReg;

end

always @(negedge clk) begin

RegWrite=RegWrite\_in;

MemToReg=MemToReg\_in;

ReadData=ReadData\_in;

ALUResult=ALUResult\_in;

WriteReg=WriteReg\_in;

end

endmodule

仿真模块：给一个时钟周期信号即可，测试时直接观测各内部信号：

module sim\_PipeLinedProcesser(

);

reg clk;

initial begin

clk=0;

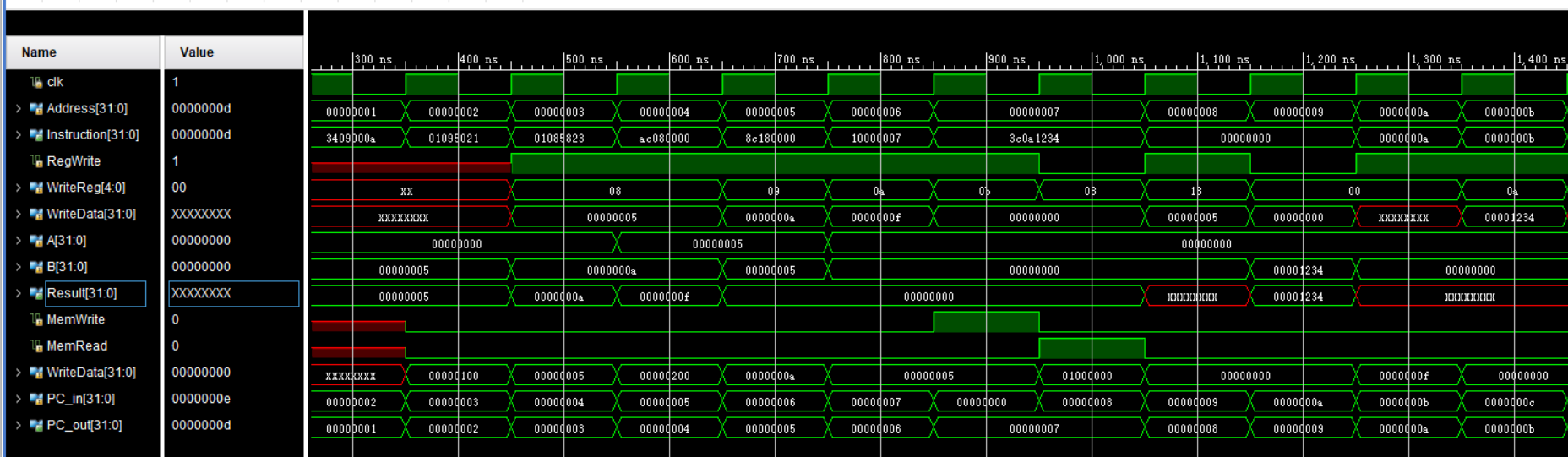
forever #50 clk=~clk;

end

Pipelined\_Processer test(clk);

Endmodule

仿真波形：



可以大致看出，在取第3条指令时，流水线各级都已经有指令在执行。第0条指令已经要将结果5写入寄存器8中，而第1条指令或运算已经算出结果32’h0000000a；对于那些要计算并写入寄存器的指令（第1、2、3条指令），在450ns、550ns、650ns分别计算出了结果32h’0000000a、32’h0000000f、32’h00000000这些结果在650ns、750ns、850ns的时钟上升沿写入对应寄存器；同时对于访存读写的指令（第4、5条指令），控制信号分别在850ns和950ns传递到存储器，其中存储器的值在1050ns写入寄存器。这里我们的时钟周期为100ns。而最后一条指令也在1350ns将立即数写入寄存器。可见在流水线CPU中各条指令同时在CPU中执行不同的阶段，连续指令的相同阶段会在连续的时钟周期执行。

**五、调试和心得体会**

1 对于复杂的CPU设计，可以从简单做起，从单周期、多周期做到流水线，熟悉各模块单元数据通路的连接关系，将复杂系统基于简单的系统上进行改进。

2 自行设计测试指令，要能测试CPU的多种功能，能覆盖到不同的微操作，同时要对连续指令的流水线执行根据时钟周期预测每个周期的状态，逐个周期地执行来排查，以准确发现存在问题的单元。