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。

**四、实验代码及结果**

单周期CPU：

主控制器设计文件：

module Controller(

input [5:0] Op,Funct,

input Zero,

output MemToReg, MemWrite,

output PCSrc,ALUSrc,

output RegDst,RegWrite,

output Jump,

output [4:0] ALUControl,

output Branch);

wire[1:0] ALUOp;

MainDec MainDec\_1(Op,MemToReg,MemWrite,Branch,ALUSrc,RegDst,RegWrite,Jump,ALUOp);

ALUDec ALUDec\_1(Funct,ALUOp,ALUControl);

assign PCSrc=Branch & Zero;

endmodule

代码分析：总控制器，通过输入信号OP和Funct译码对整个CPU的各个部件进行译码控制，主要信号作用如下：  
 MemToReg, MemWrite：读内存写入寄存器控制信号，内存写控制信号

ALUSrc：ALU输入端口选择控制信号

RegDst,RegWrite：寄存器写地址，寄存器写控制信号

Jump：Jump指令信号

ALUControl：ALU运算逻辑单元控制信号

Branch：Beq指令信号

MainDec设计文件

module MainDec(

input [5:0] Op,

output MemToReg, MemWrite,

output Branch, ALUSrc,

output RegDst, RegWrite,

output Jump,

output [1:0] ALUControl);

reg[8:0] Controls;

assign{RegWrite,RegDst,ALUSrc,Branch,MemWrite,MemToReg,Jump,ALUControl}=Controls;

always@(\*)

case(Op)

6'b000000: Controls <=9'b110000010;// RTYPE

6'b100011: Controls<=9'b101001000;//LW

6'b101011: Controls <=9'b001010000;//SW

6'b000100: Controls <=9'b000100001;//BEQ

6'b001000: Controls<=9'b101000000;//ADDI

6'b000010: Controls <=9'b000000100;//J

default: Controls<=9'bxxxxxxxxx;//illegal Op

endcase

endmodule

ALUDec设计文件：

module ALUDec(//ALU

input [5:0] Funct,

input [1:0] ALUOp,

output reg[4:0] ALUControl);

always@(\*)

case(ALUOp)

2'b00: ALUControl<=5'b00000;// add(for lw/sw/addi)

2'b01: ALUControl <=5'b00001;//sub(for beq)

default: case(Funct) //R-type Instructions

6'b100000: ALUControl<=5'b00000; //add

6'b100010: ALUControl <=5'b00001;//sub

6'b100100: ALUControl <=5'b00010;//and

6'b100101: ALUControl<=5'b00011;//or

default: ALUControl<=5'bxxxxx;//???

endcase

endcase

endmodule

代码分析：两个控制子块均由之前的实验给出，及根据指令的Funct和MainDec译码得出的ALUOp信号得到ALU控制信号ALUCrntrol。

RegFile设计文件：

`define DATA\_WIDTH 32

module RegFile

#(parameter ADDR\_SIZE = 5)

(input RST,CLK, WE3,

input [ADDR\_SIZE-1:0] RA1, RA2, WA3,

input [`DATA\_WIDTH-1:0] WD3,

output [`DATA\_WIDTH-1:0] RD1, RD2);

integer i;

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

initial begin

rf[0]=0;rf[1]=1;rf[2]=2;rf[3]=3;

for(i=4;i<2\*\*ADDR\_SIZE-1;i=i+1)rf[i]=0;

end

always@(posedge CLK,posedge RST)

begin

if(RST)for( i=0;i<2\*\*ADDR\_SIZE-1;i=i+1)rf[i]=0;

else if(WE3) rf[WA3] <= WD3;

$display("adress1 is %d,address 2 is%d\n RD1 is %d,RD2 is %d",RA1,RA2,rf[RA1],rf[RA2]);

end

assign RD1 = rf[RA1];

assign RD2 = rf[RA2];

endmodule

寄存器设计文件也基本直接使用之前实验设计的模块。即完成根据地址、读写控制信号完成寄存器的读写。

ALU 设计文件：

module ALU(F, CF, A, B, OP,Zero);

parameter SIZE = 32;

output reg [SIZE-1:0] F;

output CF;

output reg Zero;

input [SIZE-1:0] A, B;

input [4:0] OP;

parameter ALU\_AND = 5'b00010;

parameter ALU\_OR = 5'b00011;

//parameter ALU\_XOR = 4'b0010;

parameter ALU\_NOR = 5'b00100;

parameter ALU\_ADD = 5'b00000;

parameter ALU\_SUB = 5'b00001;

//parameter ALU\_SLT = 4'b0110;

//parameter ALU\_SLL = 4'b0111;

wire [7:0] EN;

wire [SIZE-1:0] Fw, Fa;

assign Fa = A & B;

initial begin

F=32'bz;

end

always@(\*) begin

case(OP)

ALU\_AND: begin F <= Fa; end

ALU\_OR: begin F <= A|B; end

//ALU\_XOR: begin F <= A^B; end

ALU\_NOR: begin F <= ~(A|B); end

default: F = Fw;

endcase

if(F==0) begin Zero=1;end

else begin Zero=0; end

end

Decoder38 decoder38\_1(OP[2:0], EN);

ADD add\_1(Fw, CF, A, B, EN[0]);

SUB sub\_1(Fw, CF, A, B, EN[1]);

//SLT slt\_1(Fw, A, B, EN[6]);

//SLL sll\_1(Fw, A, B, EN[7]);

Endmodule

代码分析：

ALU设计文件稍作修改，新增Zero输出，当ALU运算结果F为0时，Zero输出0。用来做Beq指令判断是否需要进行跳转。

内存DataMemory设计文件：

module DataMemory(

input CLK,

input RST,

input W,

input [31:0] address,

input [31:0] write\_data,

output wire [31:0] read\_data

);

integer i;

reg [31:0] RAM [1023:0];

initial begin

for(i = 0; i < 1023; i = i + 1) RAM[i] <= 0;

end

always @(posedge CLK) begin

if(RST) begin

for(i = 0; i < 1023; i=i+1) RAM[i] = 0; end

else begin

if(W == 1)begin

RAM[address] = write\_data;

$display("Write success");end

end

end

assign read\_data = RAM[address];

endmodule

SimpleMIPSCPU设计文件：

module SimpleMIPSCPU(

input wire clk, // 时钟

input wire rst, // 复位

output reg [31:0] result, // 输出结果

output wire [31:0] instruction,

output wire[31:0]PC1,

output wire[31:0]RD2,

output wire[31:0]result\_from\_memory,

output wire[31:0]alu\_result,

output wire MemWrite,

output wire RegWrite,MemtoReg, ALUSrc, Branch, Jump,RegDst, Zero,CF

);

integer i;

//PC

reg [31:0]PC;

wire [31:0] ALU\_A,ALU\_B;

wire [5:0] Op,Funct;

wire [4:0] rs, rt, rd;;

wire[4:0] ALUControl;

wire [31:0] immediate,move\_immediate;

wire[4:0] WriteReg;//写入寄存器的地址

wire[31:0] WriteData,RD1;

always @(posedge clk)

$display("clk come");

initial begin

PC=0;

result=0;

end

IMem IMem1(PC1,instruction,rs,rt,rd,Op,Funct);

Controller ctrl (

.Op(Op),

.Funct(Funct),

.Zero(Zero),

.RegDst(RegDst),

.RegWrite(RegWrite),//寄存器要不要写

.MemToReg(MemtoReg),

.MemWrite(MemWrite),//内存要不要写

.ALUSrc(ALUSrc),//用不用立即数

.Jump(Jump),

.ALUControl(ALUControl),

.Branch(Branch)

);

// 寄存器文件

RegFile regfile(

.RST(rst),

.CLK(clk),

.WE3(RegWrite),//控制寄存器是否能写

.RA1(rs),

.RA2(rt),

.WA3(WriteReg),//要写的地址,5位

.WD3(WriteData),//要写的数据，32位

.RD1(RD1),

.RD2(RD2)

);

ALU alu1(

.A(ALU\_A),

.B(ALU\_B),

.OP(ALUControl),

.F(alu\_result),

.CF(CF),

.Zero(Zero)

);

// 数据存储单元,lw/sw指令用到

// 数据存储单元,lw/sw指令用到

DataMemory dmem ( // 1Kx32bit

.CLK(clk),

.W(MemWrite),

.RST(rst),

.address(alu\_result),

.write\_data(RD2),

.read\_data(result\_from\_memory)

);

assign immediate=instruction[15:0];

assign move\_immediate={14'b00000000000000,immediate[15:0],2'b00};

//初始化ALU输入

//ALU操作数选择

assign ALU\_A=RD1;

assign ALU\_B=ALUSrc ? immediate : RD2;

assign WriteData = MemtoReg ? result\_from\_memory : alu\_result;

assign WriteReg = RegDst ? rd:rt;//控制寄存器的写入地址

assign PC1=PC;

//更新result相关

always @(\*)begin

if(rst)result<=32'b0;

else begin

result= MemtoReg ? result\_from\_memory : alu\_result;end

end

//更新PC相关

initial begin

PC=0;

end

always @(posedge clk or posedge rst) begin

if (rst) begin PC=0; end

else if (Branch&&Zero) begin//执行beq

PC=PC+4+move\_immediate;

end

else if(Jump) begin

PC=PC+4;

PC={PC[31:28],instruction[25:0],2'b00};

end

else begin

PC=PC+4; end

end

endmodule

代码分析：

代码设计主要解决的问题：

不同的指令，操作数不同，需要根据指令的不同给操作数赋不同的值：

assign ALU\_A=RD1;

assign ALU\_B=ALUSrc ? immediate : RD2;

操作数A端始终为rs寄存器中的值RD1，操作数B端续根据ALUSrc信号来控制，只有ADD/SUB时B端操作数为rt寄存器中的值RD2，其他操作码下操作数B端为立即数。

在寄存器写入值时，分两种情况：

从内存中读值写入寄存器、从ALU运算输出值读入寄存器，通过控制信号MemToReg来判断写入寄存器数据线WriteData是result\_from\_memory（从存储器中读的值）还是alu\_result（运算逻辑单元的输出值）。

assign WriteData = MemtoReg ? result\_from\_memory : alu\_result;

寄存器写入时，写入地址也根据指令有所不同：

assign WriteReg = RegDst ? rd:rt;

通过RegDst信号控制写入位置是rd还是rt。

有关PC更新：

always @(posedge clk or posedge rst) begin

if (rst) begin PC=0; end

else if (Branch&&Zero) begin//执行beq

PC=PC+4+move\_immediate;

end

else if(Jump) begin

PC=PC+4;

PC={PC[31:28],instruction[25:0],2'b00};

end

else begin

PC=PC+4; end

rst信号有效时，直接复位；

Branch指令且计算得到Zero需要进行跳转时，执行=PC+4+move\_immediate;

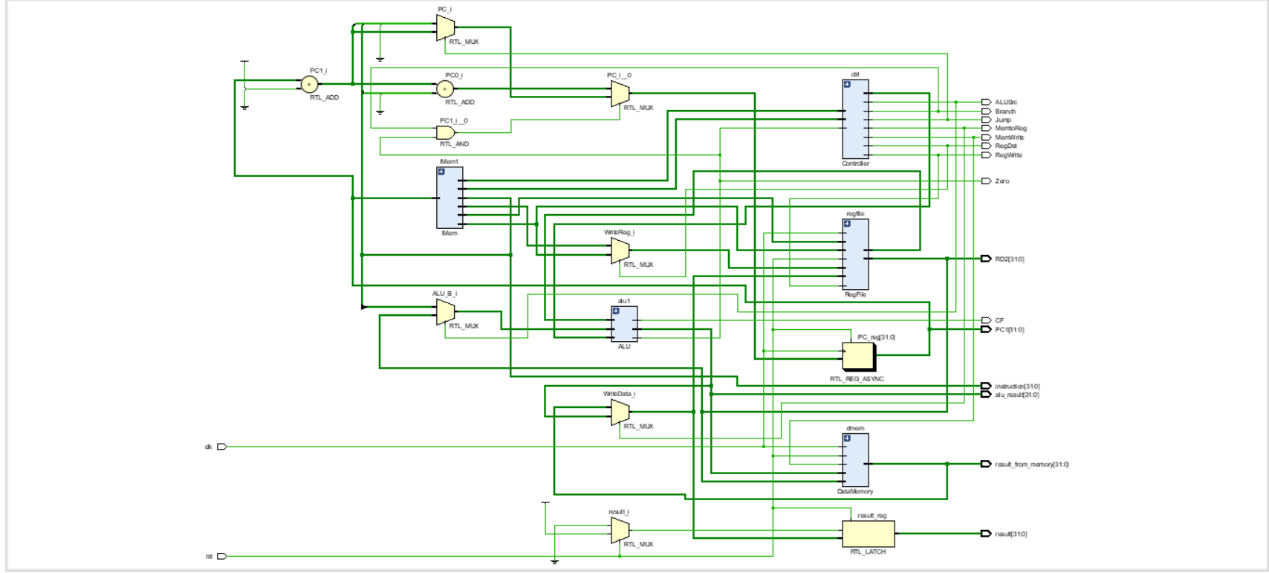
Jump指令时，执行 PC=PC+4;

PC={PC[31:28],instruction[25:0],2'b00};

跳转到指定指令

其他情况时，执行PC=PC+4.正常读取下一条指令。

RTL电路图：



仿真文件：

module sim\_SimpleMIPSCPU(

);

reg clk;

reg rst;

wire [31:0] result;

wire [31:0] instruction;

wire[31:0]PC1,RD2,result\_from\_memory,alu\_result;

wire MemWrite;

wire RegWrite,MemtoReg, ALUSrc, Branch, Jump,RegDst, Zero,CF;

initial begin

clk=0;

rst=0;

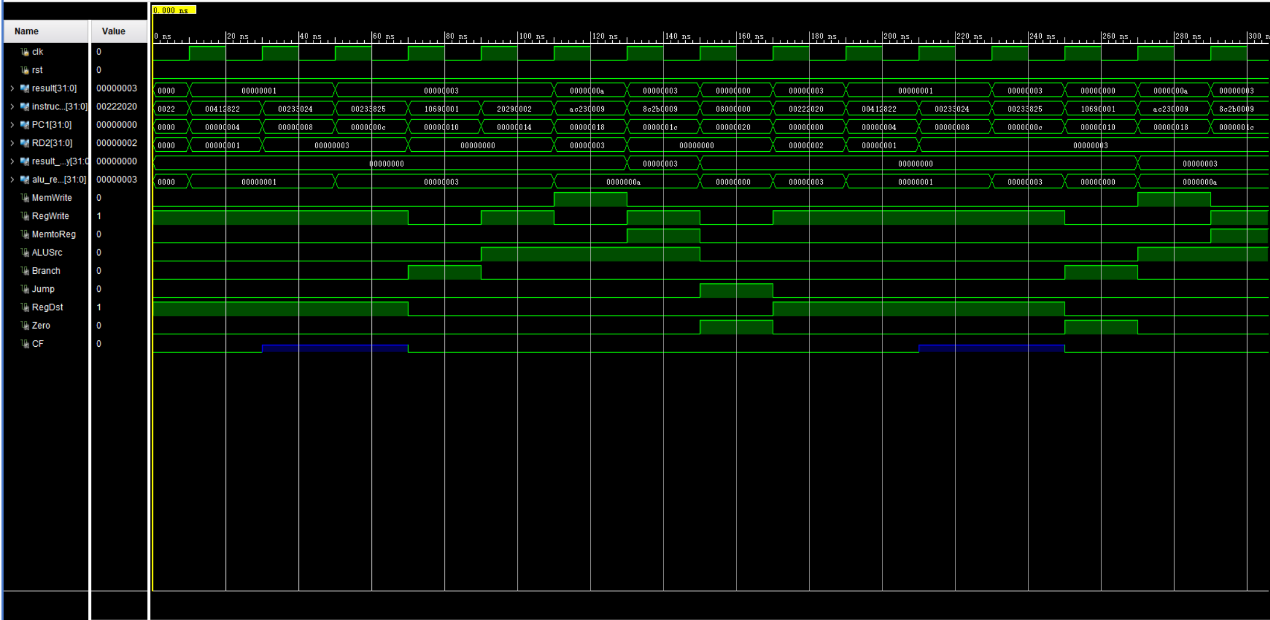
repeat(100)#10 clk=~clk;

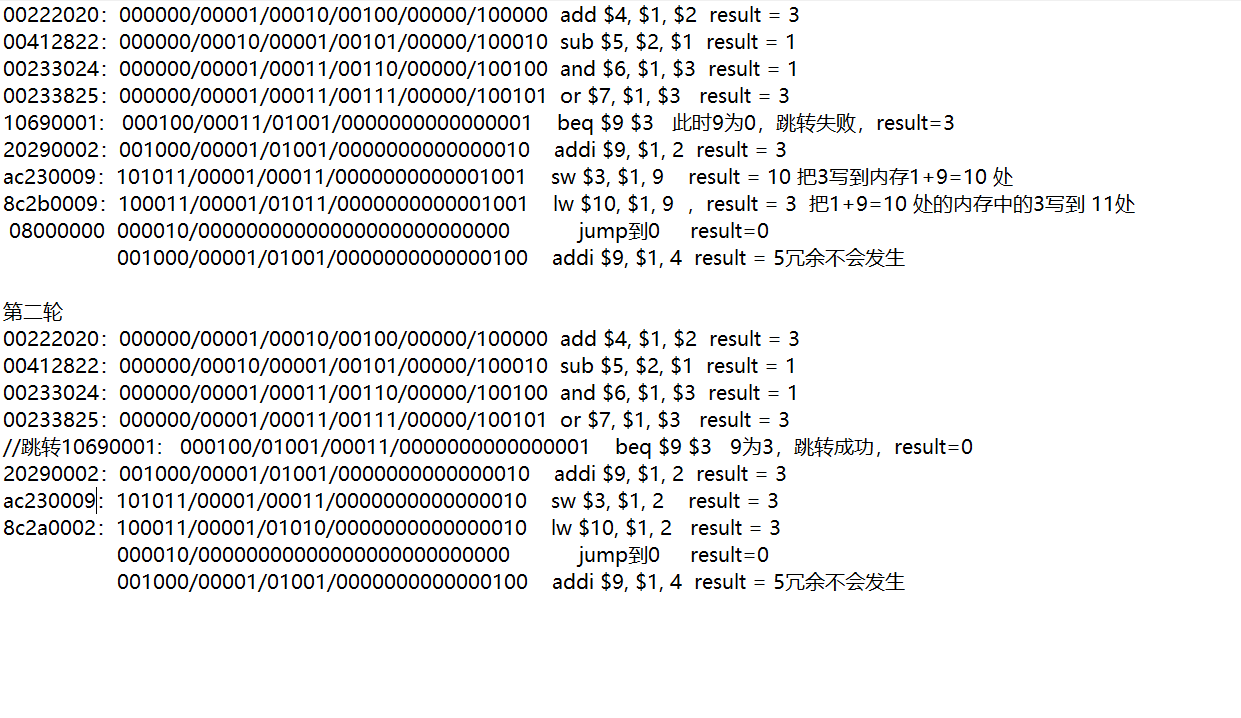
end

SimpleMIPSCPU simcpu(clk,rst,result,instruction,PC1,RD2,result\_from\_memory,alu\_result,MemWrite,RegWrite,MemtoReg, ALUSrc, Branch, Jump,RegDst, Zero,CF);

Endmodule

时序图：

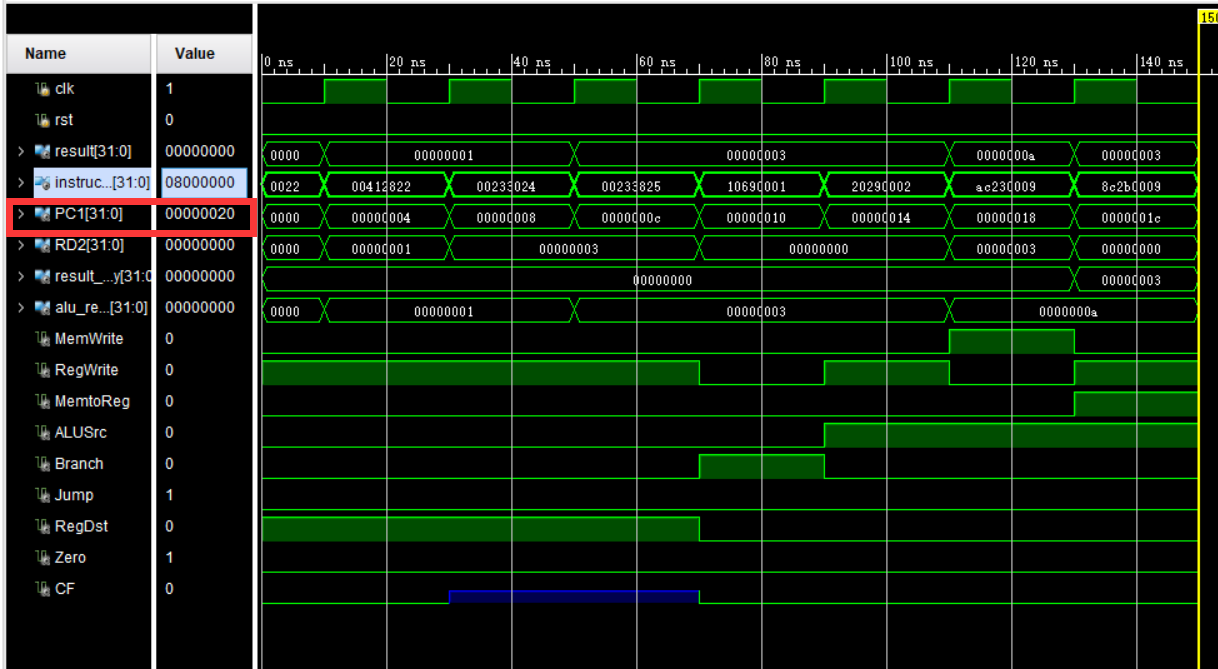


指令集：

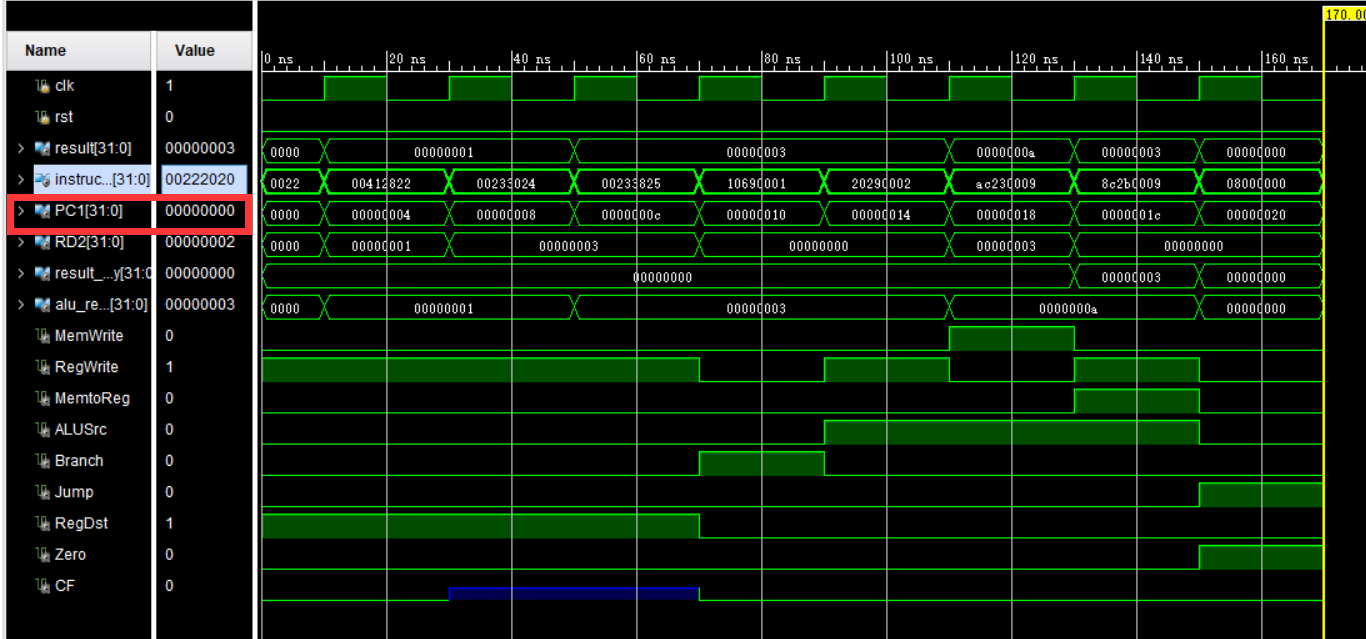
可以观察到时序图按指令集顺序进行，对应值完全相同。

主要实现的两种跳转功能：

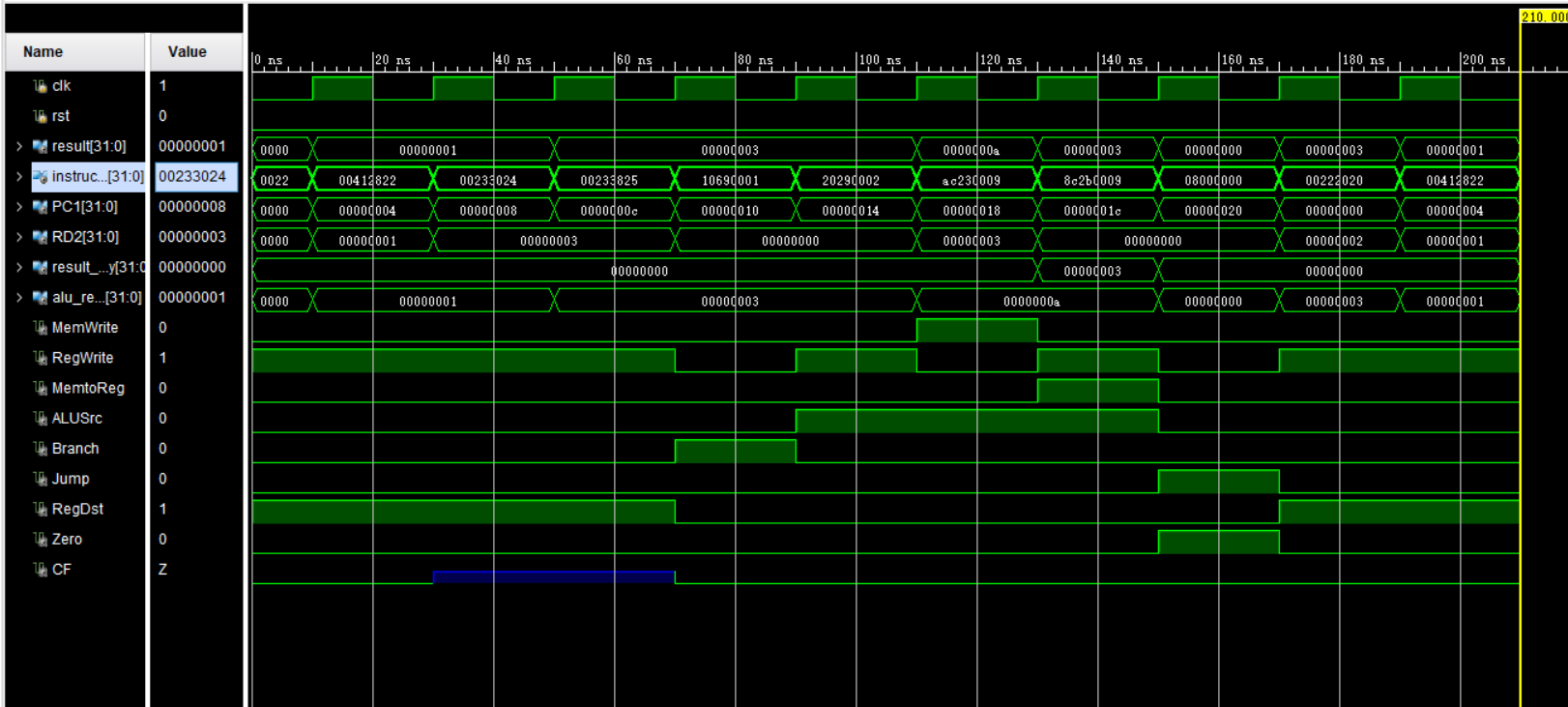
Jump实现效果

Jump之前，PC=0x20

Jump之后，按指令跳转到0

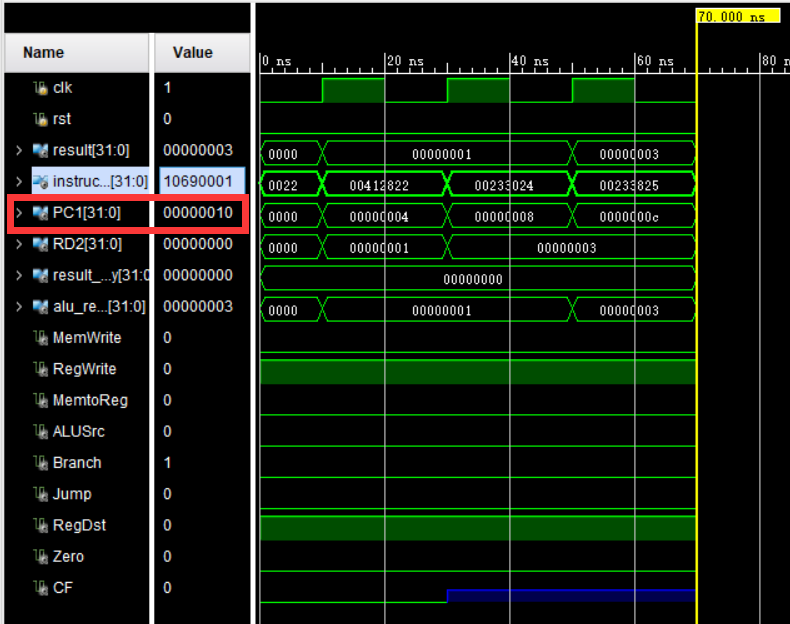


且跳转之后的指令仍然正常执行：



Branch指令执行效果

Branch指令执行之前：



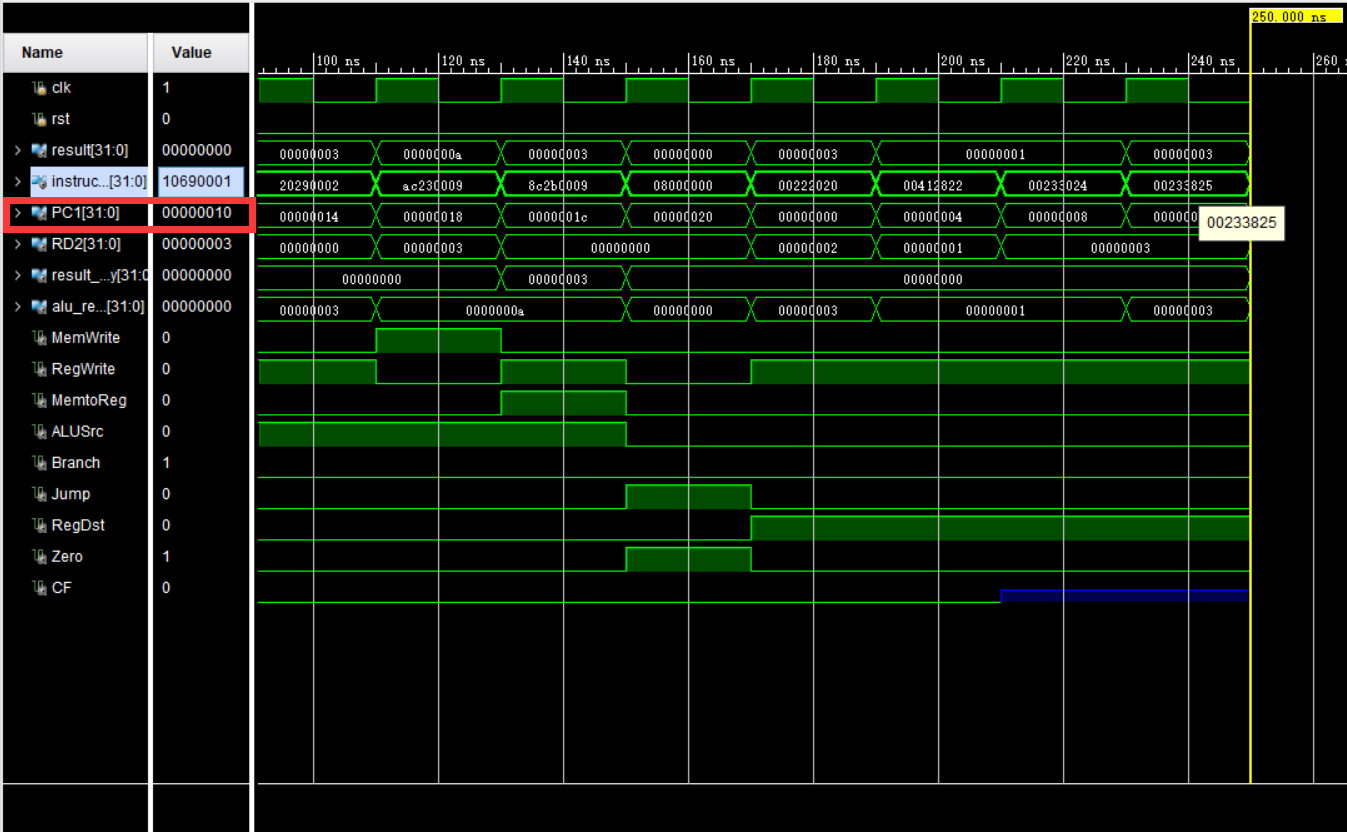
Branch指令执行之后：



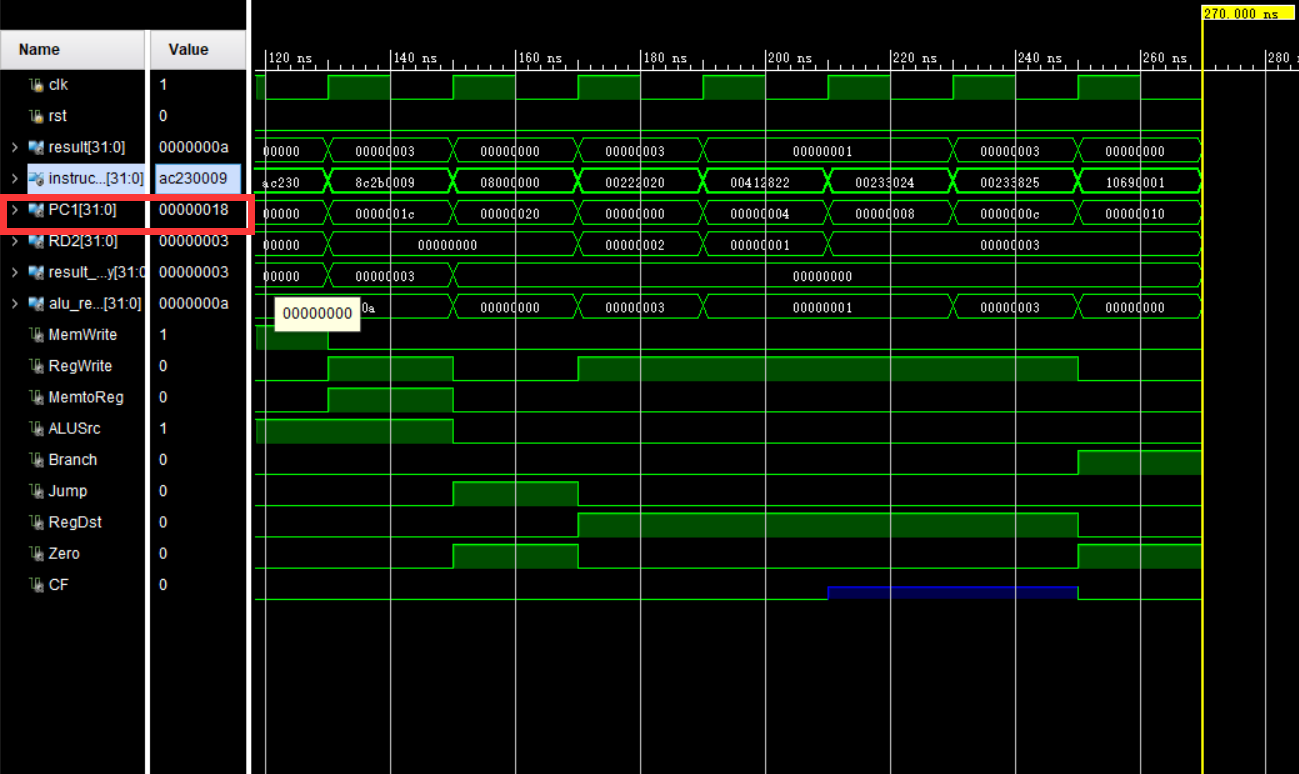
可以观察到PC只是顺序加4，并没有发生跳转，与我们设计的第一轮由于对应寄存器的值还没有修改，所以Beq不会跳转的设计相同。

第二轮：

执行Branch指令之前：



执行Branch指令之后



直接跳转到0x18，跳转成功。

多周期CPU设计：

基本部件与单周期CPU基本相同，只是对所有的组件增加一个时钟控制信号，只在时钟控制信号上升沿才能正常工作：

指令寄存器IMem：

module IMem(

input clk,

input [31:0] A,

output reg[31:0] RD,

output reg [4:0] rs,rt,rd,

output reg [5:0] Op,Funct

);

parameter IMEM\_SIZE = 64;

reg [31:0] RAM[0:1023];

initial

$readmemb("C:/Users/LeBlanc/Desktop/11.txt", RAM);

always@(posedge clk)begin

$display("instruction is %x",RAM[A/4]);

Funct=RAM[A/4][5:0];

rd=RAM[A/4][15:11];

rt=RAM[A/4][20:16];

rs=RAM[A/4][25:21];

Op=RAM[A/4][31:26];

RD = RAM[A/4];

end

endmodule

主控制逻辑单元Controller：

module Controller(

input clk,

input [5:0] Op,Funct,

input Zero,

output MemToReg, MemWrite,

output PCSrc,ALUSrc,

output RegDst,RegWrite,

output Jump,

output [4:0] ALUControl,

output Branch);

wire[1:0] ALUOp;

MainDec MainDec\_1(clk,Op,MemToReg,MemWrite,Branch,ALUSrc,RegDst,RegWrite,Jump,ALUOp);

ALUDec ALUDec\_1(clk,Funct,ALUOp,ALUControl);

assign PCSrc=Branch & Zero;

endmodule

寄存器堆RegFile：

`define DATA\_WIDTH 32

module RegFile

#(parameter ADDR\_SIZE = 5)

(input RST,CLK, WE3,

input [ADDR\_SIZE-1:0] RA1, RA2, WA3,

input [`DATA\_WIDTH-1:0] WD3,

output [`DATA\_WIDTH-1:0] RD1, RD2);

integer i;

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

initial begin

rf[0]=0;rf[1]=1;rf[2]=2;rf[3]=3;

for(i=4;i<2\*\*ADDR\_SIZE-1;i=i+1)rf[i]=0;

end

always@(posedge CLK,posedge RST)

begin

if(RST)for( i=0;i<2\*\*ADDR\_SIZE-1;i=i+1)rf[i]=0;

else if(WE3) rf[WA3] <= WD3;

$display("adress1 is %d,address 2 is%d\n RD1 is %d,RD2 is %d",RA1,RA2,rf[RA1],rf[RA2]);

end

assign RD1 = rf[RA1];

assign RD2 = rf[RA2];

endmodule

内存RAM：

module RAM(

input clk,

input [31:0] data\_in,

output reg [31:0] data\_out,

input [9:0] Addr,

input Rst,

input W\_R,

input CS

);

parameter Addr\_Width=10;

parameter Data\_Width=32;

parameter SIZE=2 \*\* Addr\_Width;

integer i;

reg[Data\_Width-1:0]RAM[SIZE-1:0];

initial begin

for(i=0;i<SIZE;i=i+1) RAM[i]=0;

end

always @(posedge clk) begin

if (Rst) begin

for(i=0;i<=SIZE-1;i=i+1) RAM[i]=0;

end else if (CS) begin

if (W\_R) begin

RAM[Addr] = data\_in;

end else begin

data\_out = RAM[Addr];

end

end

end

endmodule

运算逻辑单元ALU：

module ALU(clk,F, CF, A, B, OP,Zero);

parameter SIZE = 32;

input clk;

output reg [SIZE-1:0] F;

output reg CF;

output reg Zero;

input [SIZE-1:0] A, B;

input [4:0] OP;

parameter ALU\_AND = 5'b00010;

parameter ALU\_OR = 5'b00011;

//parameter ALU\_XOR = 4'b0010;

parameter ALU\_NOR = 5'b00100;

parameter ALU\_ADD = 5'b00000;

parameter ALU\_SUB = 5'b00001;

//parameter ALU\_SLT = 4'b0110;

//parameter ALU\_SLL = 4'b0111;

wire [7:0] EN;

wire [SIZE-1:0] Fw, Fa;

assign Fa = A & B;

initial begin

F=32'bz;

end

always@(posedge clk) begin

case(OP)

ALU\_AND: begin F <= Fa; end

ALU\_OR: begin F <= A|B; end

ALU\_NOR: begin F <= ~(A|B); end

ALU\_ADD:begin {CF,F}=A+B; end

ALU\_SUB:begin {CF,F}=A-B; end

default: F = Fw;

endcase

if(F==0) begin Zero=1;end

else begin Zero=0; end

end

endmodule

**多周期CPU主设计模块代码：**

module MultiCycleMIPSCPU(

input wire clk, // 时钟

input wire rst, // 复位

output reg [31:0] result, // 输出结果

output reg [2:0] state,

output wire [31:0] instruction,

output wire[31:0]PC1,

output wire[31:0] RD1,RD2,

output wire[31:0]result\_from\_memory,

output wire[31:0]alu\_result,

output wire MemWrite,

output wire RegWrite,MemToReg, PCSrc,ALUSrc, Branch, Jump,RegDst, Zero,CF,

output IM\_clk,Con\_clk,Reg\_clk,Alu\_clk,Ram\_clk,

output ALU\_A,ALU\_B,ALUControl

);

parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011,S4=3'b100,S5=3'b101,S6=3'b110,S7=3'b111;

initial begin

PC=0;

state=S0;

result=32'bz;

end

integer i;

//PC

reg [31:0]PC;

wire [31:0] ALU\_A,ALU\_B;

wire [5:0] Op,Funct;

wire [4:0] rs, rt, rd;;

wire[4:0] ALUControl;

wire [31:0] immediate,move\_immediate;

wire[4:0] WriteReg;//写入寄存器的地址

wire[31:0] WriteData;

reg IM\_clk,Con\_clk,Reg\_clk,Alu\_clk,Ram\_clk;

always@(posedge clk )begin

if(rst)begin PC=0;state=S0; end

else

begin

if(state==S0) IM\_clk=1;

else IM\_clk=0;

if(state==S1) Con\_clk=1;

else Con\_clk=0;

if(state==S2) Alu\_clk=1;

else Alu\_clk=0;

if(state==S3) Ram\_clk=1;

else Ram\_clk=0;

if(state==S4) Reg\_clk=1;

else Reg\_clk=0;

if(state==S5) begin

if(Zero==1)

begin PC=PC+4+move\_immediate; end

else

PC=PC+4;

end

if(state==S6)begin

PC=PC+4;

PC={PC[31:28],instruction[25:0],2'b00};

end

end

case(state)

S0:state=S1;//取出指令

S1:state=S2;//p判断为S1,已经给了译码器信号,读也读完了

S2:begin

if(MemWrite||MemToReg)state=S3;//判断为S3，已经给了ALU信号,MemWrite=1，要写存储器，跳转到状态4

else if(RegWrite)state=S4;//判断为S3，已经给了ALU信号，RegWrite=1，要写寄存器，跳转到状态5

else if(Branch)state=S5;

else if(Jump)state=S6;

else begin state=0;PC=PC+4;end

end

S3:begin state=S0;PC=PC+4; end

S4:begin state=S0;PC=PC+4; end

S5:begin state=S0;end

S6:begin state=S0;end

default: begin state=S0;PC=PC+4; end

endcase

end

IMem IM1( .clk(IM\_clk),

.A(PC1),

.RD(instruction),

.rs(rs),.rt(rt),.rd(rd),

.Op(Op),.Funct(Funct));

Controller ctrl(

.clk(Con\_clk),

.Op(Op),.Funct(Funct),

.Zero(Zero),

.MemToReg(MemToReg),.MemWrite(MemWrite),

.PCSrc(PCSrc),.ALUSrc(ALUSrc),

.RegDst(RegDst),.RegWrite(RegWrite),

.Jump(Jump),

.ALUControl(ALUControl),

.Branch(Branch));

RegFile regfile(

.RST(rst),

.CLK(Reg\_clk),

.WE3(RegWrite),//控制寄存器是否能写

.RA1(rs),

.RA2(rt),

.WA3(WriteReg),//要写的地址,5位

.WD3(WriteData),//要写的数据，32位

.RD1(RD1),

.RD2(RD2)

);

ALU alu1(

.clk(Alu\_clk),

.A(ALU\_A),

.B(ALU\_B),

.OP(ALUControl),

.F(alu\_result),

.CF(CF),

.Zero(Zero)

);

RAM ram1(.clk(Ram\_clk),

.data\_in(RD2),

.data\_out(result\_from\_memory),

.Addr(alu\_result),

.Rst(rst),

.W\_R(MemWrite),

.CS(MemWrite||MemToReg)

);

assign immediate=instruction[15:0];

assign move\_immediate={14'b00000000000000,immediate[15:0],2'b00};

//ALU操作数选择

assign ALU\_A=RD1;

assign ALU\_B=ALUSrc ? immediate : RD2;

assign WriteData = MemToReg ? result\_from\_memory : alu\_result;

assign WriteReg = RegDst ? rd:rt;//控制寄存器的写入地址

assign PC1=PC;

always @(\*)

begin

if(rst)result<=32'b0;

else begin

if(Alu\_clk)

begin

if(!MemToReg)result=alu\_result;

end

if(Ram\_clk)

begin

if(MemToReg)result=result\_from\_memory;

end

end

end

endmodule

实现多周期的方式：

使用寄存器state存储CPU当前处于哪个阶段，下一个时钟信号上升沿来临时根据当前状态的不同进行不同的工作。每个不同组件（例如ALU、寄存器）的时钟控制信号收到原始时钟上升沿和当前状态的共同控制，具体实现代码如下：

always@(posedge clk )begin

……

begin

if(state==S0) IM\_clk=1;

else IM\_clk=0;

if(state==S1) Con\_clk=1;

else Con\_clk=0;

if(state==S2) Alu\_clk=1;

else Alu\_clk=0;

if(state==S3) Ram\_clk=1;

else Ram\_clk=0;

if(state==S4) Reg\_clk=1;

else Reg\_clk=0;

if(state==S5) begin

if(Zero==1)

begin PC=PC+4+move\_immediate; end

else

PC=PC+4;

end

if(state==S6)begin

PC=PC+4;

PC={PC[31:28],instruction[25:0],2'b00};

end

end

当时公共时钟上升沿clk来临时，根据当前状态（state）的不同，对不同的部件时钟信号赋能。例如，当状态处于S0时，时钟信号来临时要进行的取指令译码操作，则对IMem的部件控制信号IM\_clk赋1，IMem部件的时钟控制信号上升沿触发操作，实现目标功能，由于不处于其他部件的周期，其他部件不工作。

状态更新：每迎来一个时钟上升沿完成多周期CPU的一个周期操作时，需更新CPU状态值，以便下一周期的操作正常有序的进行：

always@(posedge clk )begin

……

case(state)

S0:state=S1;//取出指令

S1:state=S2;//p判断为S1,已经给了译码器信号,读也读完了

S2:begin

if(MemWrite||MemToReg)state=S3;//判断为S3，已经给了ALU信号,MemWrite=1，要写存储器，跳转到状态4

else if(RegWrite)state=S4;//判断为S3，已经给了ALU信号，RegWrite=1，要写寄存器，跳转到状态5

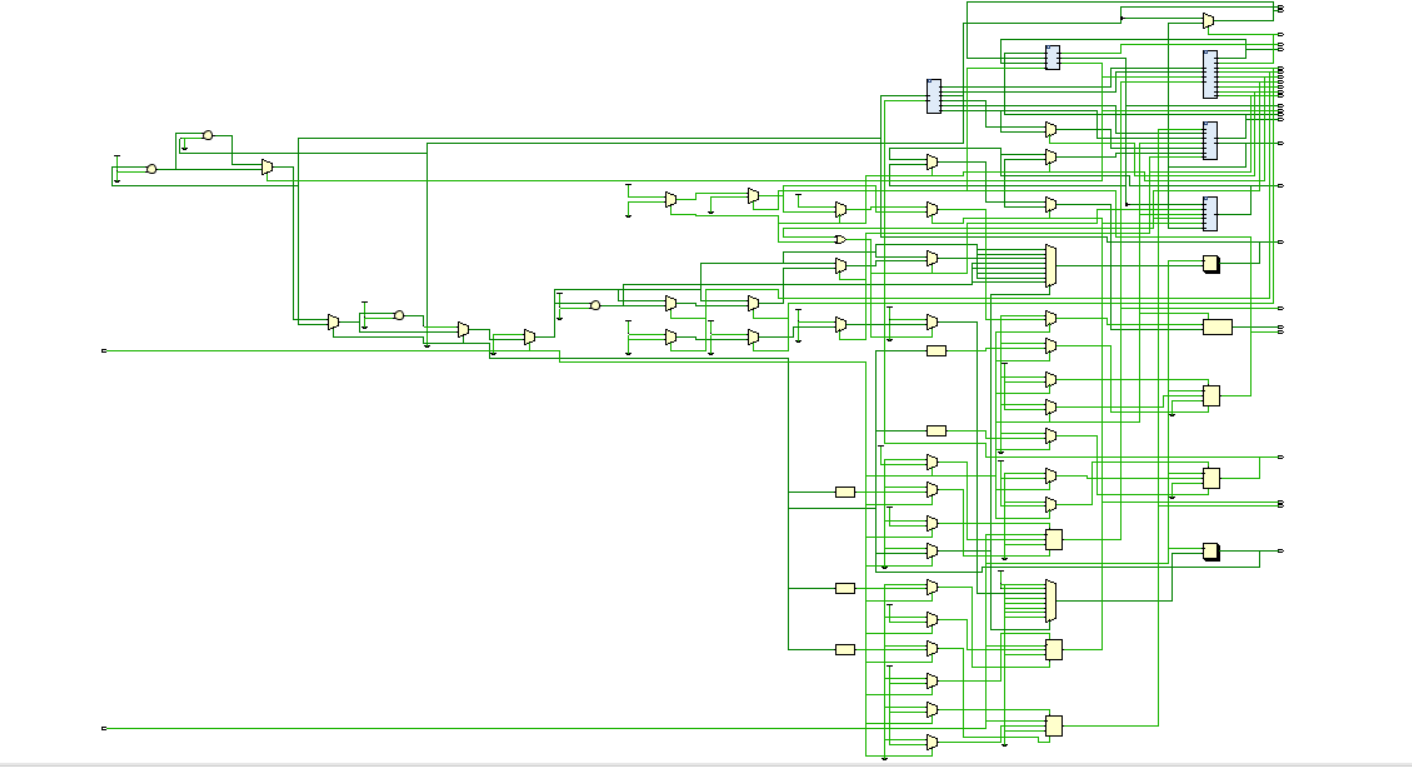
else if(Branch)state=S5;

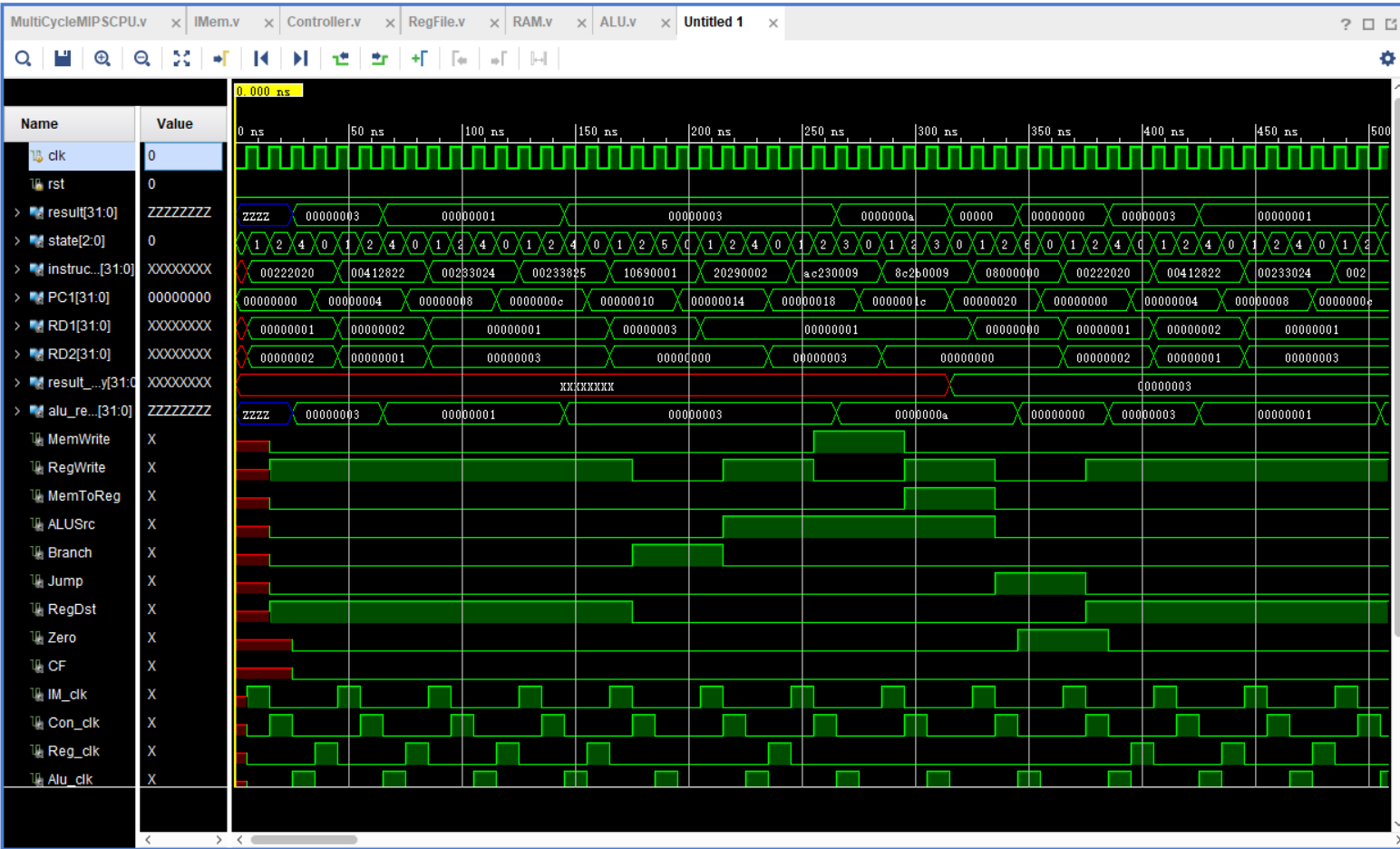
else if(Jump)state=S6;

else begin state=0;PC=PC+4;end

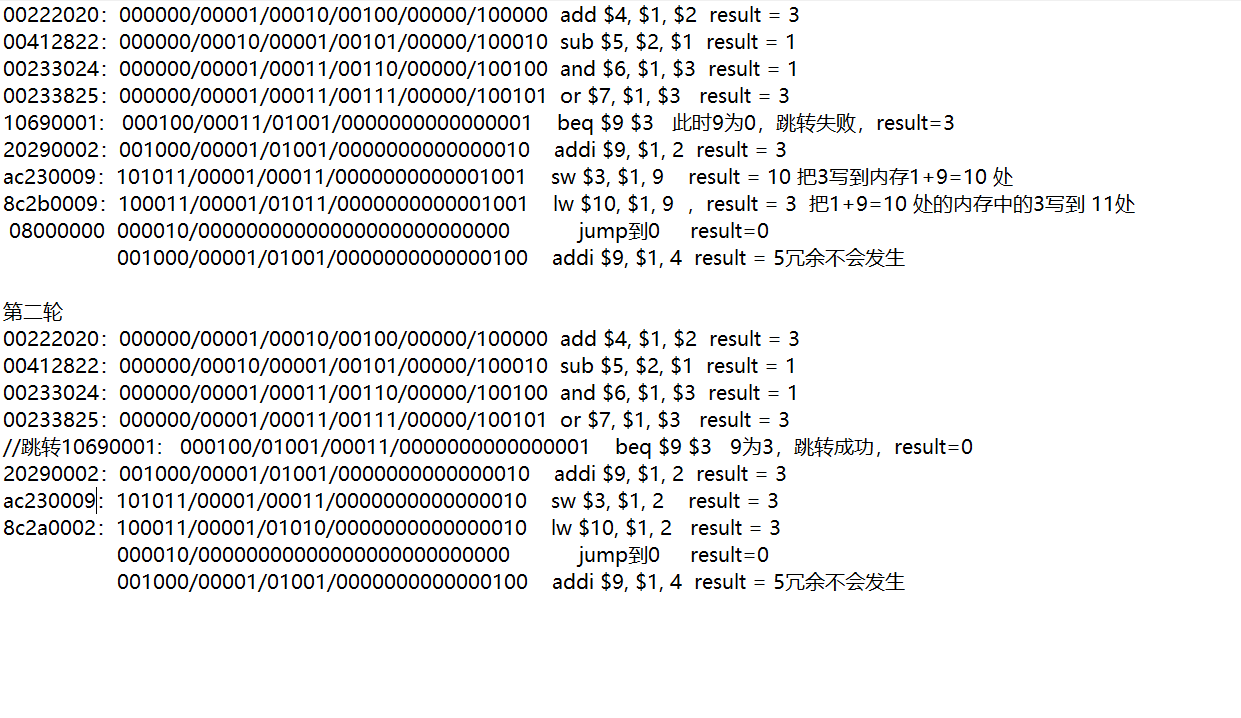
end

除开一开始公共的取指令译码周期外，从S2状态开始，根据操作的不同，跳转到不同的状态，实现不同指令的执行周期不同的功能。

RTL仿真图：

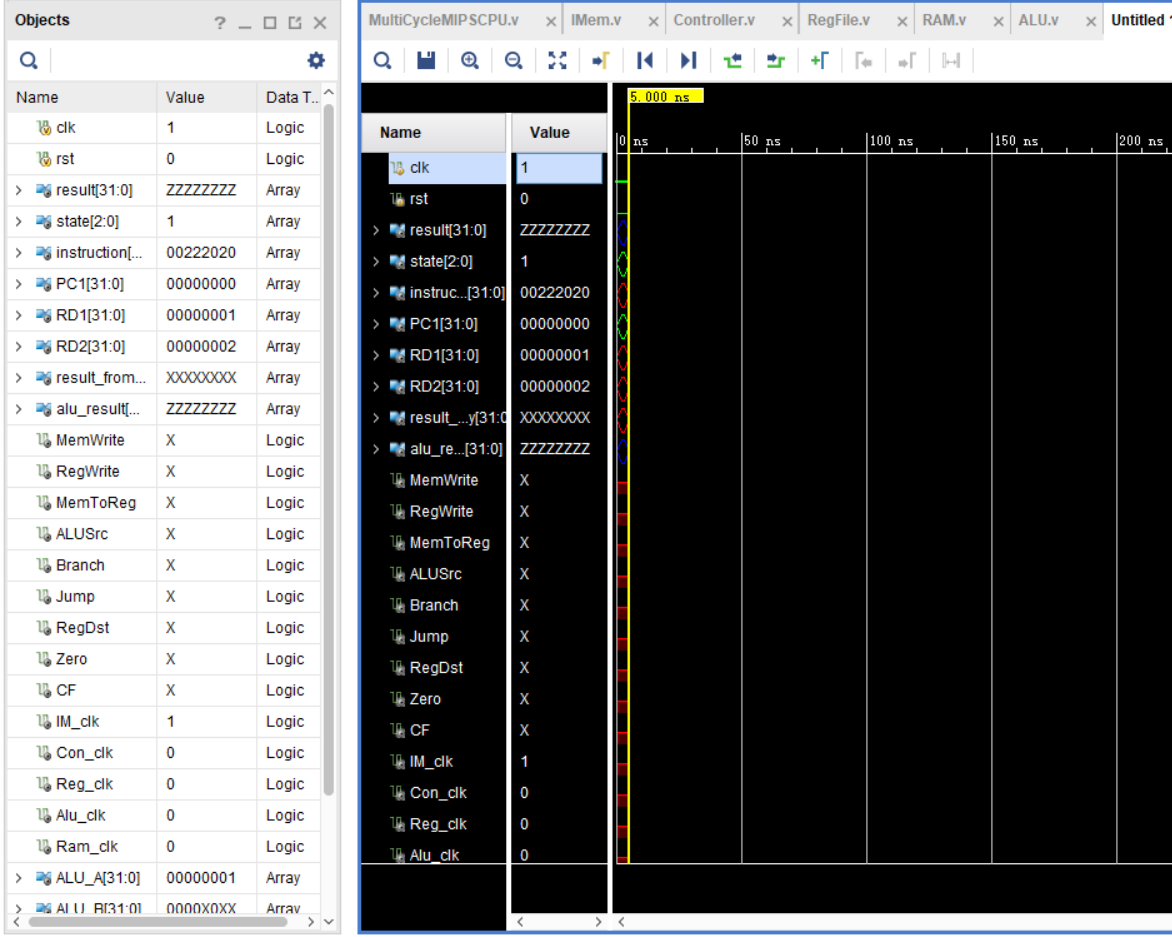
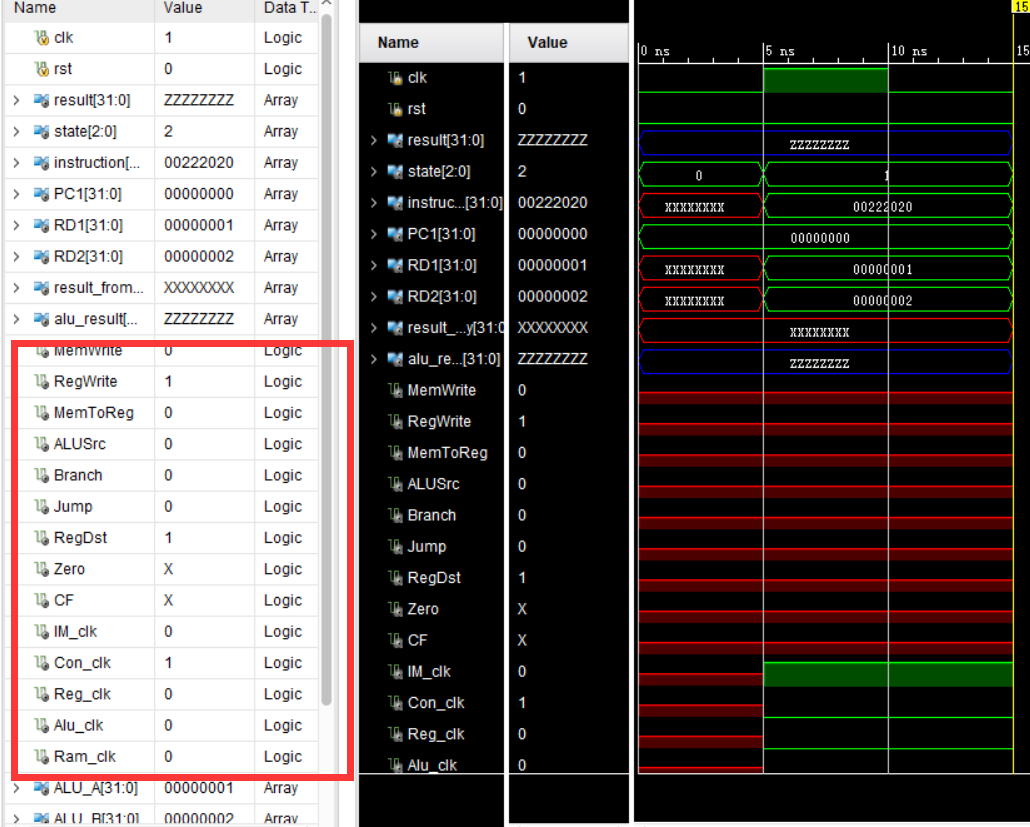
时序图：

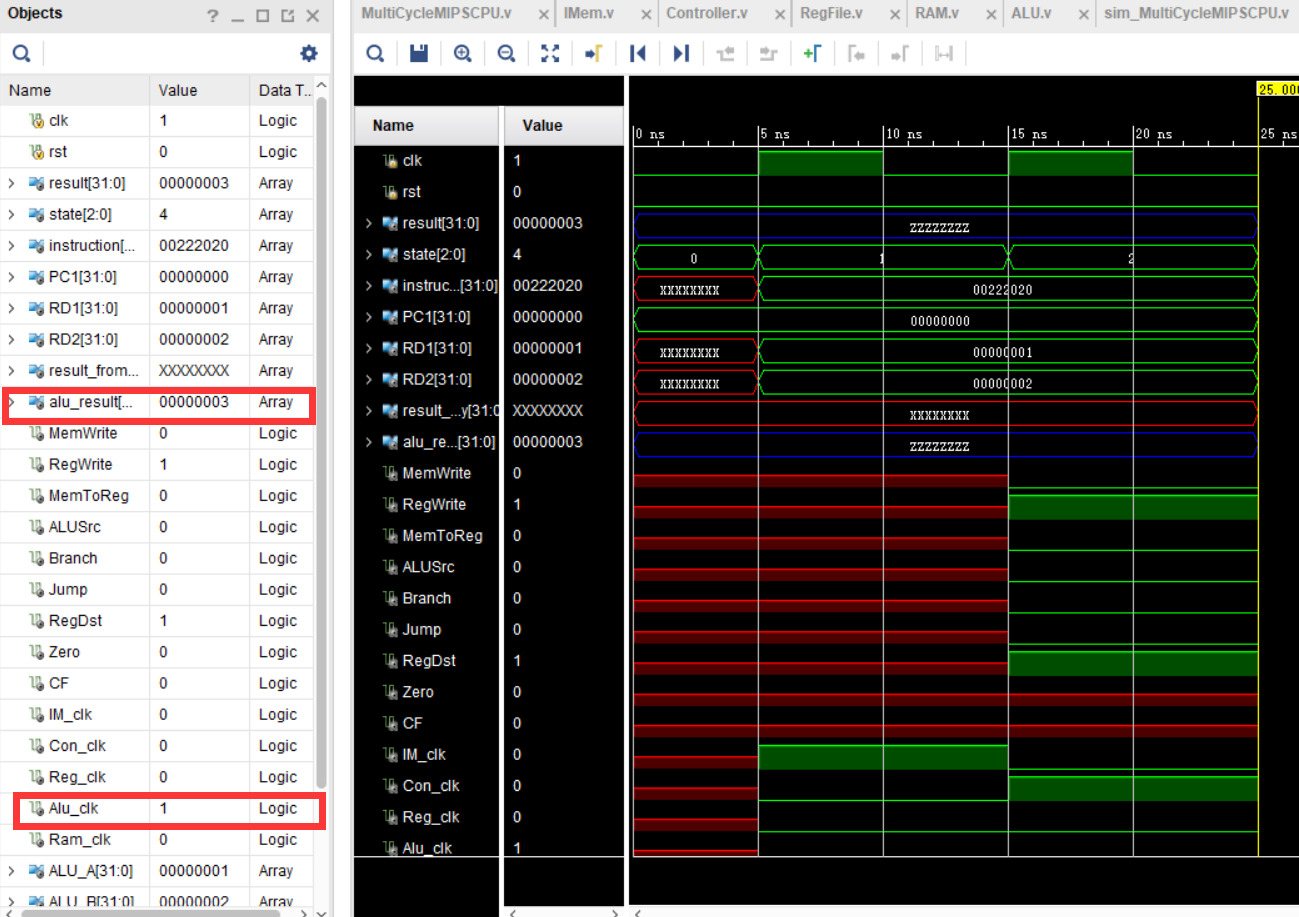
仍然使用单周期时的指令集：

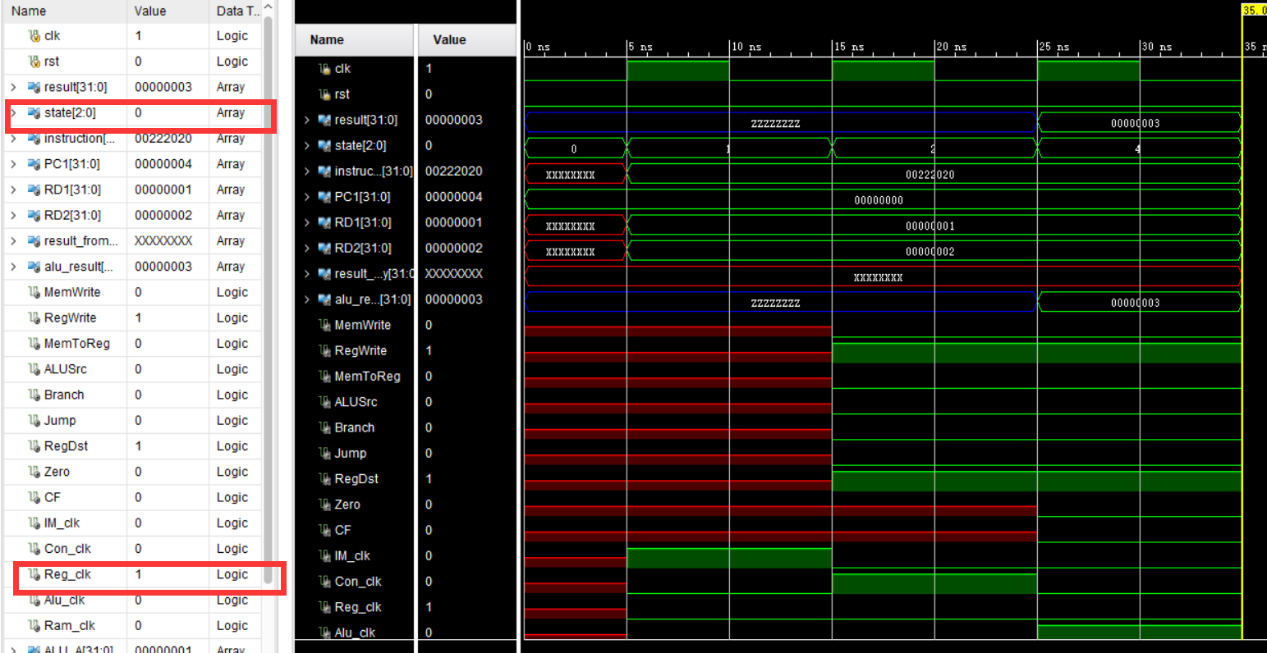


验证是否完成多周期：

5ns时刻：完成取指令周期、读寄存器

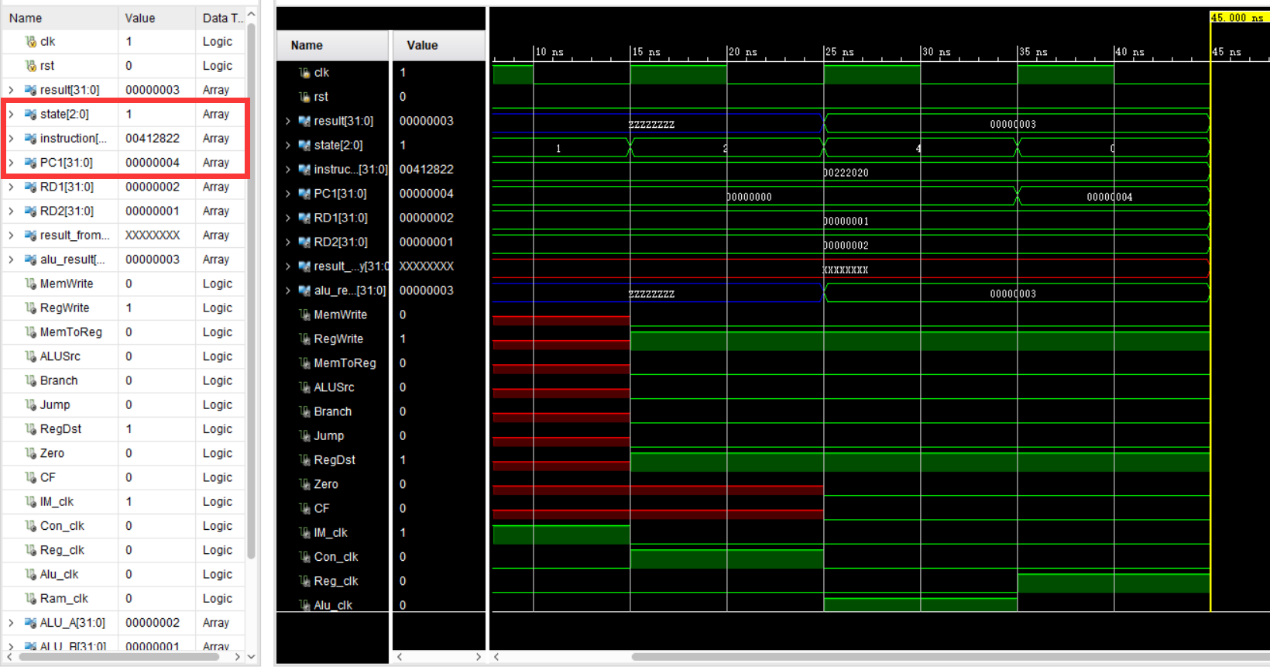
15ns：完成译码周期，控制信号开始有值

25ns：ALU运算周期，alu\_result输出值：

35ns：第一条指令为add，所以需要进行写回寄存器操作：

此时Reg\_clk为1，进行了写回操作，且写回操作结束后，此条指令执行结束，state返回S0，PC更新，以执行下一条指令。

45ns：执行下一条指令的取址周期，可以观察到state正常更新，instruction更新为新指令的值。



**五、调试和心得体会**

通过本次对单周期CPU和多周期CPU设计的实验，让我对CPU的工作原理有了更深入的了解。之前只从课本中了解到CPU的工作过程，但经过自己设计实验很多不清楚的地方也更了解了。

由于工作量较大，设计过程中遇到了很多BUG，调试过程中，学会了使用$display来打印测试各个阶段变量的值，可以发现是过程中哪一个步骤出现了问题。

且经过本次实验，让我意识到了合理代码的重要性，深刻理解了按模块设计的思想。在写主模块设计文件时，各个模块之间的声明混在一起，导致逻辑十分混乱，后续过程中，将不同模块的寄存器、wire变量分开声明，使得系统的时序、逻辑更加的清晰，哪个地方多哪个地方少就很直观的调试出来了。