2021 春季学期

计算机组成原理

MIPS 单周期 CPU

设

计

报

告

Hollow Man

目录

一、 课程设计简述

设计语言：Verilog 硬件描述语言

仿真环境：Vivado

开发板：xczu7cg-ffcv1156-2-e

频率：8.3MHz（对应周期 120ns）

实现指令数量：25 条

CPI：1

二、 设计原理

单周期 CPU

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。

CPU 在处理指令时，一般需要经过以下几个步骤：

(1) 取指令(IF)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。

(2) 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

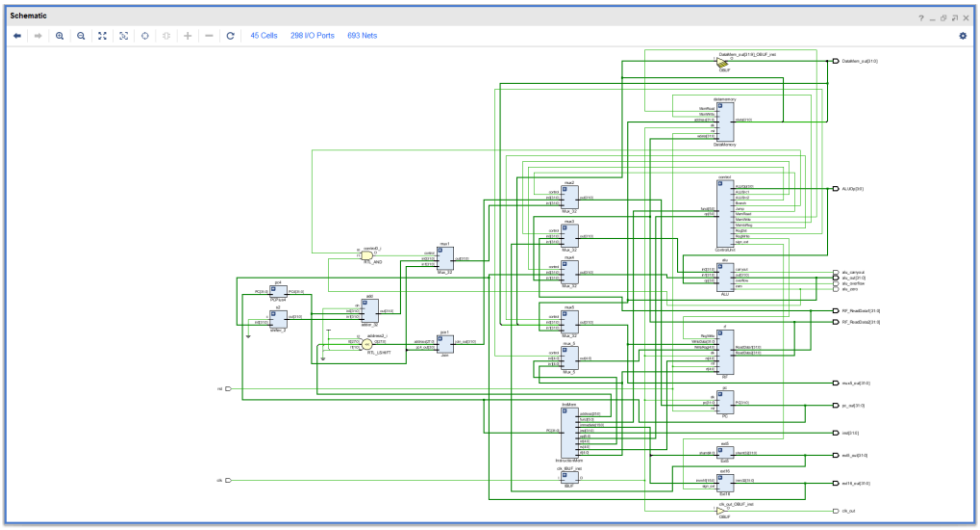
(5) 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

三、 设计内容

注：文档仅对实现原理和部分结构进行说明，具体实现请查看项目代码。

总体结构图



包含模块：PC，PCPlus4，InstructionMem，DataMemory，ControlUnit,RF,ALU(adder_32,shifter_2),Ext5,Ext16,Join,Mux_5,Mux_32

(1) PC

模块功能

给出指令在指令储存器中的地址。

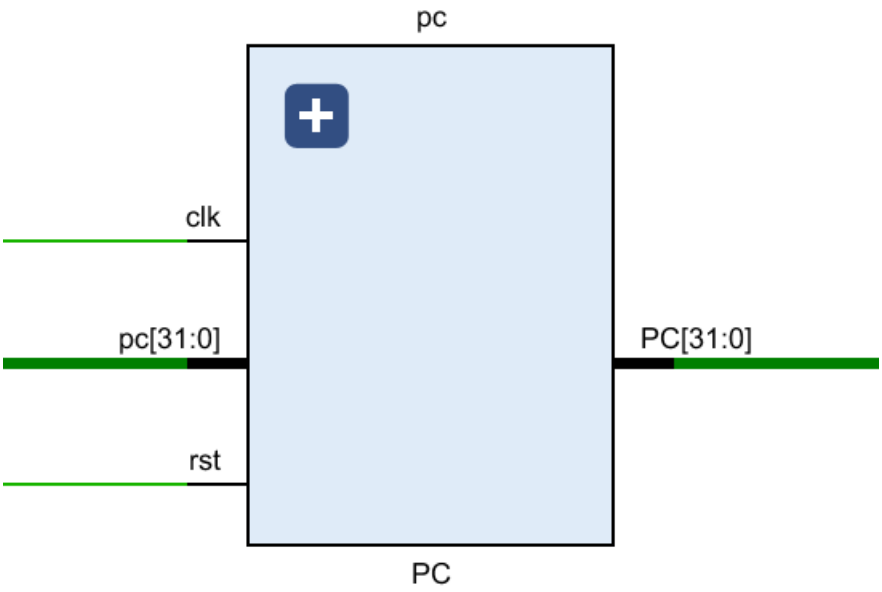
引脚及控制信号

clk：时钟信号（输入）

pc[31:0]:目标地址,跳转地址或下一条指令的地址(输入)

rst：复位信号

PC[31:0]：指令地址（输出）



(2) PCPlus4

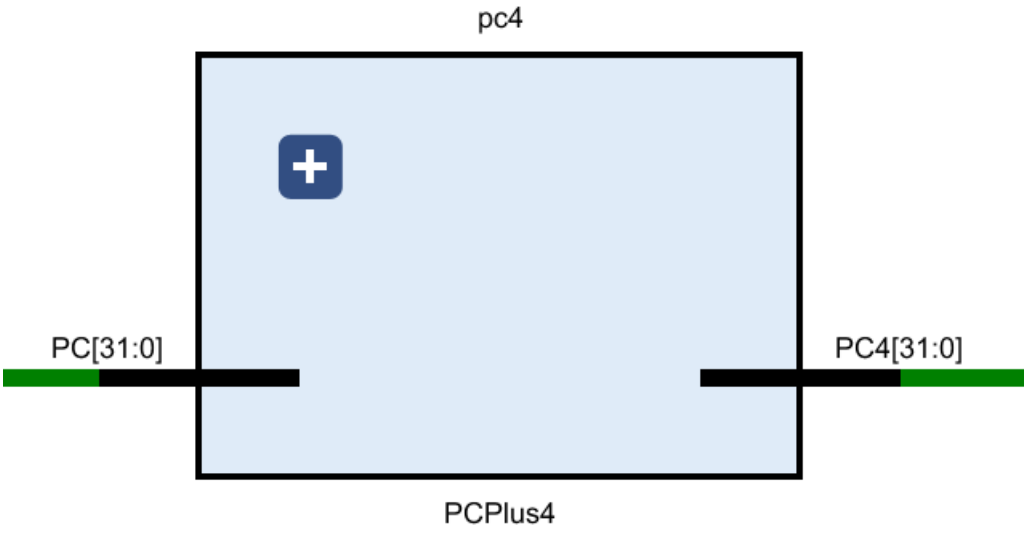
模块功能

实现 PC+4，获取下一条指令的地址

引脚及控制信号

PC[31:0]：当前指令的地址

PC4[31:0]：下一条指令的地址



(3) InstructionMem

模块功能

依据当前 PC 获取指令存储器中的指令并完成分割指令

引脚及控制信号

PC[31:0]：需获取的指令的地址（输入）

op[5:0]：指令的操作码部分（输出）

rs[4:0]：rs 寄存器的地址（输出）

rt[4:0]：rt 寄存器的地址（输出）

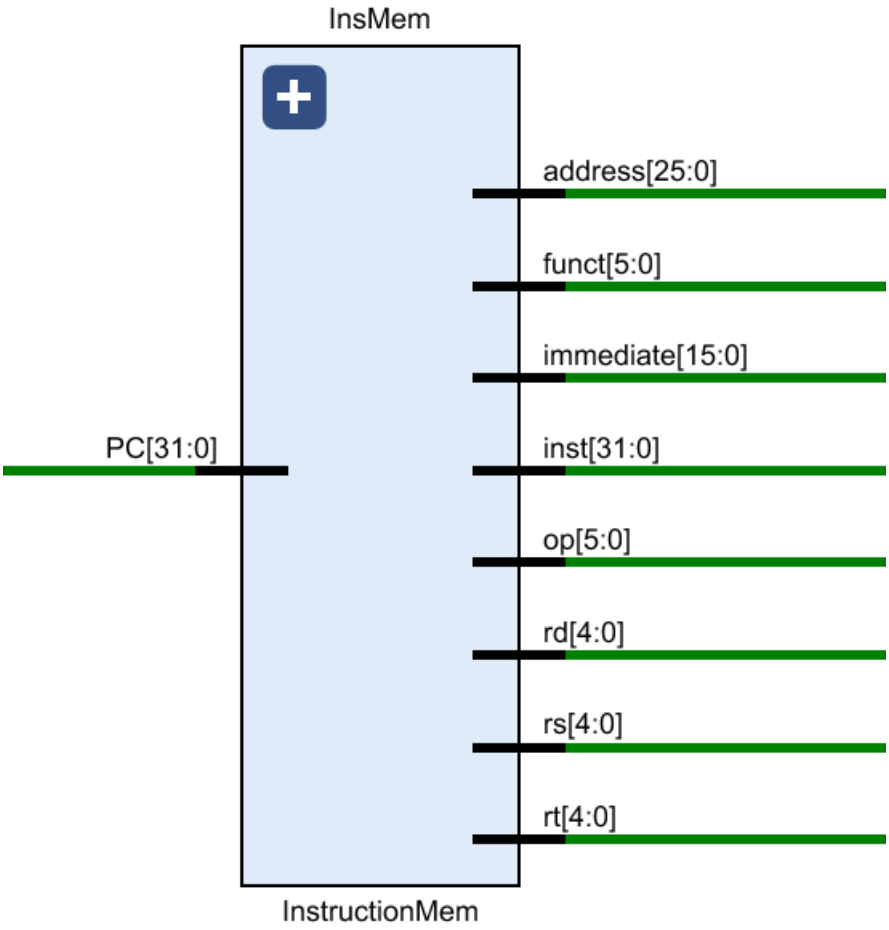
rd[4:0]：rd 寄存器的地址（输出）

funct[5:0]：指令的功能码部分（输出）

immediate [15:0]：指令（I 型）中的立即数（输出）

address [25:0]：指令（J 型）中的地址，用于生成跳转地址（输出）

inst[31:0]:用于 CPU 模块中输出当前指令以便于观察验证（输出）



寻址方式为按字节寻址，因此申请的寄存器组的宽度为 8 位，每四个单元构成一条指令。指令以文件的形式存储，使用十六进制存放和读取，指令在存储器中以大端模式存放。

```
reg[7:0] mem[0:127];
initial begin
    $readmemh("F:\\CPU\\MIPS-CPU-main\\imem.txt",mem);  //使用十六进制存放指令，故用readmemh
end
```

```
//分割指令
assign inst[31:24]=mem[PC];
assign inst[23:16]=mem[PC+1];
assign inst[15:8]=mem[PC+2];
assign inst[7:0]=mem[PC+3];
assign op=mem[PC][7:2];
assign rs[4:3]=mem[PC][1:0];
assign rs[2:0]=mem[PC+1][7:5];
assign rt=mem[PC+1][4:0];
assign rd=mem[PC+2][7:3];
assign funct=mem[PC+3][5:0];
assign immediate[15:8]=mem[PC+2];
assign immediate[7:0]=mem[PC+3];
assign address[25:24]=mem[PC][1:0];
assign address[23:16]=mem[PC+1];
assign address[15:8]=mem[PC+2];
assign address[7:0]=mem[PC+3];
```

因为指令是以十六进制的方式存放和读取的，因此指令每两位之间需要加空格或换行（若不加空格或不换行，寄存器读取指令的最后两位后直接跳到下一条指令）

```
00 43 08 20
00 41 20 21
00 64 28 22
00 c7 40 23
00 c7 48 24
00 c7 50 25
00 c7 58 26
00 c7 60 27
01 6c 68 2a
01 6c 70 2b
```

(4) DataMemory

模块功能

此模块为数据存储器，通过输入的控制信号，对数据寄存器进行读或者写操作。

引脚及控制信号

clk：时钟信号（输入）

rst：复位信号

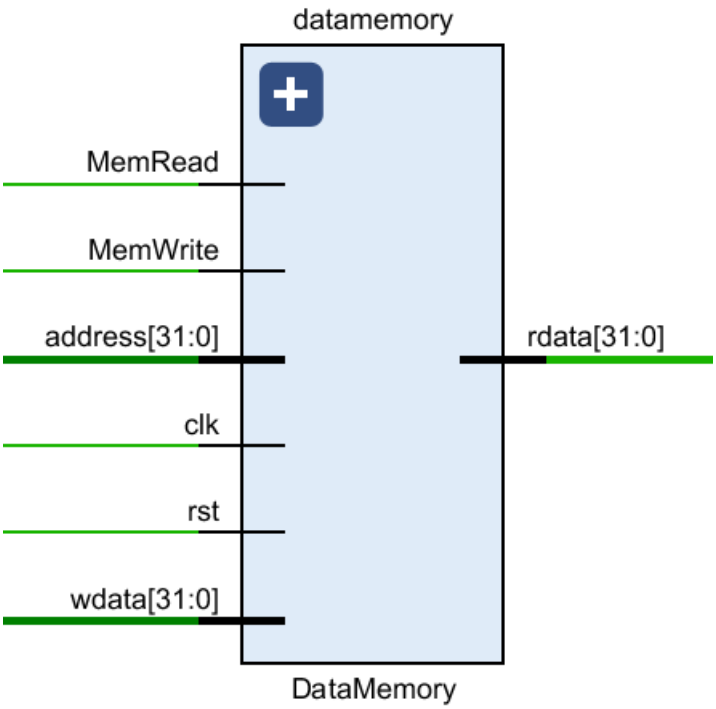
MemRead：读使能信号

MemWrite：写使能信号

Address[31:0]：读取/写入地址

wdata[31:0]：写入的数据

rdata [31:0] 读出的数据



(5) ControlUnit

模块功能

控制单元，用来输出各模块的控制信号。

引脚及控制信号

op[5:0]：输入指令的 op 部分

funct[5:0]：输入指令的 funct 部分

MemtoReg：输出选择数据来源信号

MemWrite：输出存储器写使能信号

MemRead：输出数据存储器读使能信号

ALUOp[3:0]：输出 ALU 运算功能选择

ALUSrc1：输出 ALU 操作数 1 来源选择信号

ALUSrc2：输出 ALU 操作数 2 来源选择信号

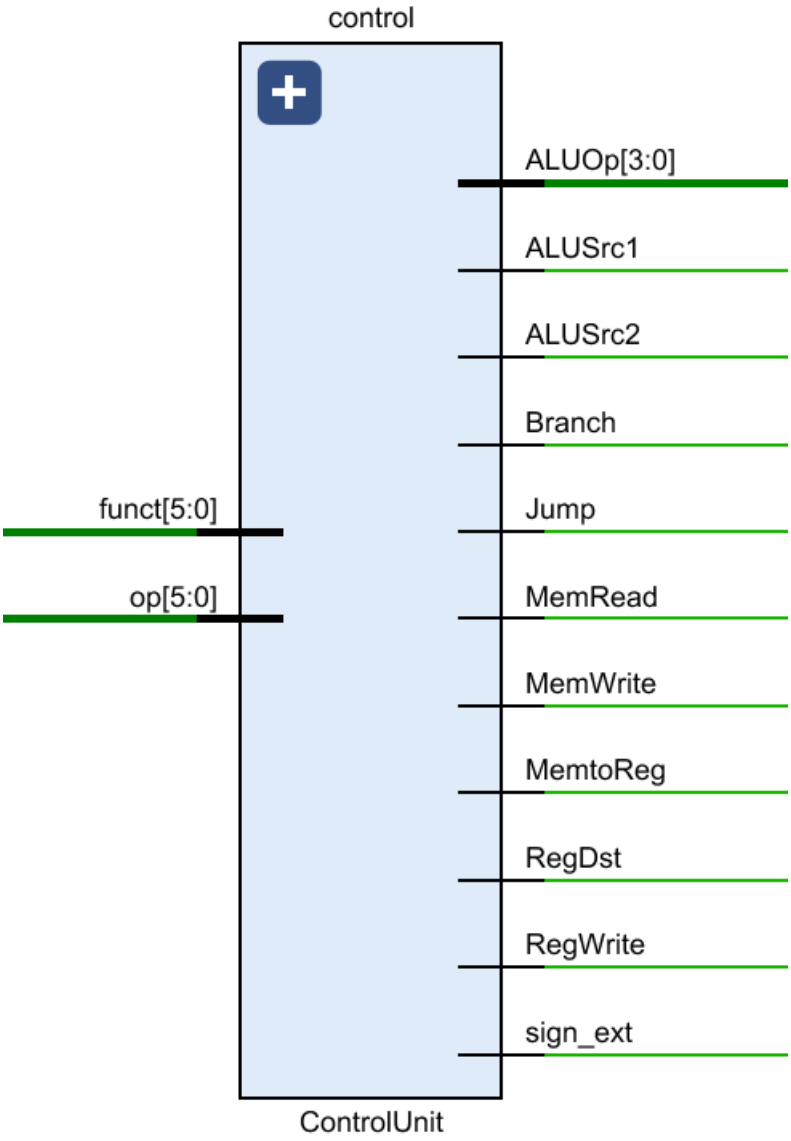
RegDst：输出写回操作的数据来源信号

RegWrite：输出寄存器堆写使能信号

Branch：配合 ALU 零标志输出生成 PCSrc

Jump：输出跳转标志信号

sign_ext：输出是否为符号扩展信号



先根据操作码和功能码确定某一条指令，再根据不同指令的数据通路生成控制信号。

指令的判断：

```
assign addu=R_type&funct[5]&~funct[4]&~funct[3]&~funct[2]&~funct[1]&funct[0];   //100001
assign sub=R_type&funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&~funct[0];    //100010
assign subu=R_type&funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&funct[0];    //100011
assign And=R_type&funct[5]&~funct[4]&~funct[3]&funct[2]&~funct[1]&~funct[0];    //100100
assign Or=R_type&funct[5]&~funct[4]&~funct[3]&funct[2]&~funct[1]&funct[0];      //100101
assign Xor=R_type&funct[5]&~funct[4]&~funct[3]&funct[2]&funct[1]&~funct[0];     //100110
assign Nor=R_type&funct[5]&~funct[4]&~funct[3]&funct[2]&funct[1]&funct[0];      //100111
assign slt=R_type&funct[5]&~funct[4]&funct[3]&~funct[2]&funct[1]&~funct[0];     //101010
assign sltu=R_type&funct[5]&~funct[4]&funct[3]&~funct[2]&funct[1]&funct[0];     //101011
assign sll=R_type&~funct[5]&~funct[4]&~funct[3]&~funct[2]&~funct[1]&~funct[0];  //000000
assign srl=R_type&~funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&~funct[0];   //000010
assign sra=R_type&~funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&funct[0];    //000011
assign sllv=R_type&~funct[5]&~funct[4]&~funct[3]&funct[2]&~funct[1]&~funct[0];  //000100
assign srlv=R_type&~funct[5]&~funct[4]&~funct[3]&funct[2]&funct[1]&~funct[0];   //000110
assign srav=R_type&~funct[5]&~funct[4]&~funct[3]&funct[2]&funct[1]&funct[0];    //000111

//I type
assign addi=~op[5]&~op[4]&op[3]&~op[2]&~op[1]&~op[0];   //001000
assign addiu=~op[5]&~op[4]&op[3]&~op[2]&~op[1]&op[0];   //001001
assign andi=~op[5]&~op[4]&op[3]&op[2]&~op[1]&~op[0];   //001100
assign ori=~op[5]&~op[4]&op[3]&op[2]&~op[1]&op[0];      //001101
assign xori=~op[5]&~op[4]&op[3]&op[2]&op[1]&~op[0];     //001110
assign lw=op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];       //100011
assign sw=op[5]&~op[4]&op[3]&~op[2]&op[1]&op[0];        //101011
assign beq=~op[5]&~op[4]&~op[3]&op[2]&~op[1]&~op[0];    //000100
assign bne=~op[5]&~op[4]&~op[3]&op[2]&~op[1]&op[0];     //000101

//J type
assign j=~op[5]&~op[4]&~op[3]&~op[2]&op[1]&~op[0];     //000010
```

生成控制信号

```
assign MemtoReg=lw;
assign MemWrite=sw;
assign MemRead=lw;
assign Branch=beq|bne;
assign ALUSrc1=add|addu|sub|subu|And|Or|Xor|Nor|slt|sltu|addi|addiu|andi|ori|xori|lw|sw|beq|bne;
assign ALUSrc2=addi|addiu|andi|ori|xori|lw|sw;
assign RegDst=add|addu|sub|subu|And|Or|Xor|Nor|slt|sltu|sll|srl|sra|sllv|srlv|srav;
assign RegWrite=add|addu|sub|subu|And|Or|Xor|Nor|slt|sltu|sll|srl|sra|sllv|srlv|srav|addi|addiu|andi|ori|xori
assign Jump=j;
assign sign_ext=lw|sw|beq;
```

(6) RF

模块功能

储存寄存器堆模块,并且可以根据地址对寄存器组进行读写。

引脚及控制信号

clk：时钟信号（输入）

rst：复位信号

RegWrite：寄存器堆写使能信号（输入）

rs [4:0]：rs 寄存器地址（输入）

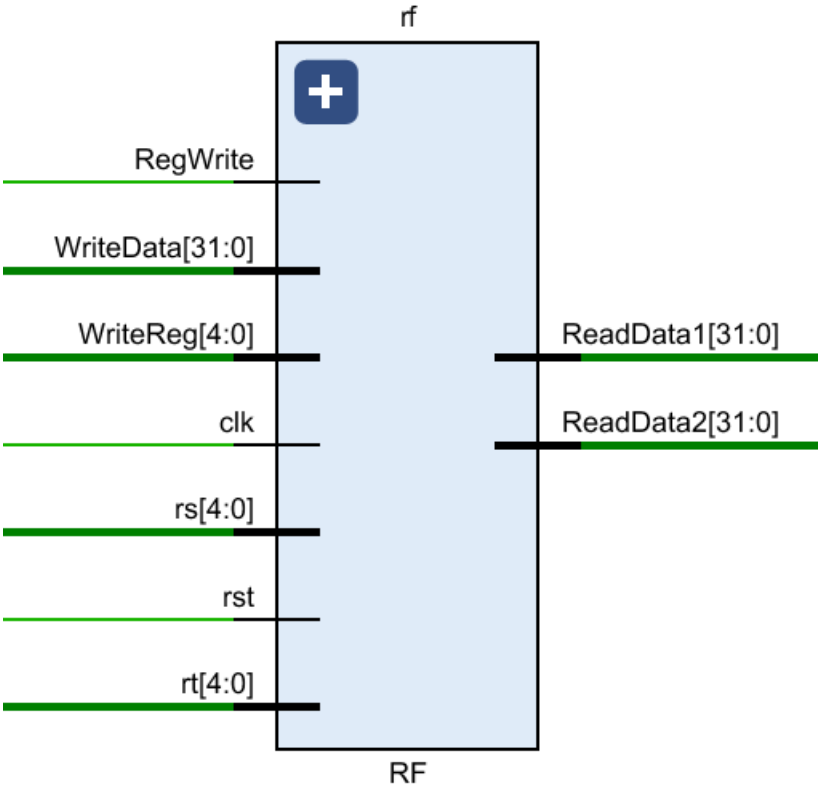
rt [4:0]：rt 寄存器地址（输入）

WriteReg [4:0]：将数据写入的寄存器，其地址来源 rt 或 rd 字段（输入）

WriteData [31:0]：写入寄存器的数据（输入）

ReadData1 [31:0]：rs 寄存器数据（输出）

ReadData2 [31:0]：rt 寄存器数据（输出）



(7) adder_32

模块功能

超前进位加法器模块,实现 32 位操作数的加减法，有四个 8 位超前进位加法器模块(adder_8)组合而成

```
module adder_32(cin,in0,in1,carryout,out);
    input [31:0] in0;
    input [31:0] in1;
    input cin;
    output [31:0] out;
    output carryout;
    //wire [31:0] in0_com,in1_com,out_com;
    wire carryout1;
    wire carryout2;
    wire carryout3;

    adder_8 adder_8_0(cin,in0[7:0],in1[7:0],carryout1,out[7:0]);
    adder_8 adder_8_1(carryout1,in0[15:8],in1[15:8],carryout2,out[15:8]);
    adder_8 adder_8_2(carryout2,in0[23:16],in1[23:16],carryout3,out[23:16]);
    adder_8 adder_8_3(carryout3,in0[31:24],in1[31:24],carryout,out[31:24]);

endmodule
```

(8) shifter_2

模块功能

实现数据的移位操作（左/右移两位），由两个基本的移位器组合而成，移位器采用门级电路实现。

```
module shifter_2(in0,c,out);
    input [31:0] in0;
    input c;
    output [31:0] out;
    wire [31:0] temp;
    shifter s1(in0,c,temp);
    shifter s2(temp,c,out);
endmodule
```

```
module shifter(in0,c,out);
    input [31:0] in0;
    input c;
    output[31:0] out;
    //reg [31:0] out;
    wire [31:0]out;
            assign out[0]=in0[1]&c;
            assign out[1]=(in0[0]&~c)|(in0[2]&c);
            assign out[2]=(in0[1]&~c)|(in0[3]&c);
            assign out[3]=(in0[2]&~c)|(in0[4]&c);
            assign out[4]=(in0[3]&~c)|(in0[5]&c);
            assign out[5]=(in0[4]&~c)|(in0[6]&c);
            assign out[6]=(in0[5]&~c)|(in0[7]&c);
            assign out[7]=(in0[6]&~c)|(in0[8]&c);
            assign out[8]=(in0[7]&~c)|(in0[9]&c);
            assign out[9]=(in0[8]&~c)|(in0[10]&c);
            assign out[10]=(in0[9]&~c)|(in0[11]&c);
            assign out[11]=(in0[10]&~c)|(in0[12]&c);
            assign out[12]=(in0[11]&~c)|(in0[13]&c);
            assign out[13]=(in0[12]&~c)|(in0[14]&c);
            assign out[14]=(in0[13]&~c)|(in0[15]&c);
            assign out[15]=(in0[14]&~c)|(in0[16]&c);
            assign out[16]=(in0[15]&~c)|(in0[17]&c);
            assign out[17]=(in0[16]&~c)|(in0[18]&c);
            assign out[18]=(in0[17]&~c)|(in0[19]&c);
            assign out[19]=(in0[18]&~c)|(in0[20]&c);
            assign out[20]=(in0[19]&~c)|(in0[21]&c);
            assign out[21]=(in0[20]&~c)|(in0[22]&c);
            assign out[22]=(in0[21]&~c)|(in0[23]&c);
```

(9) ALU

模块功能

算数逻辑运算单元，沿用上次设计的成果。

引脚及控制信号

op[3:0]：ALU 计算类型码

in0[31:0]：第一个数据输入

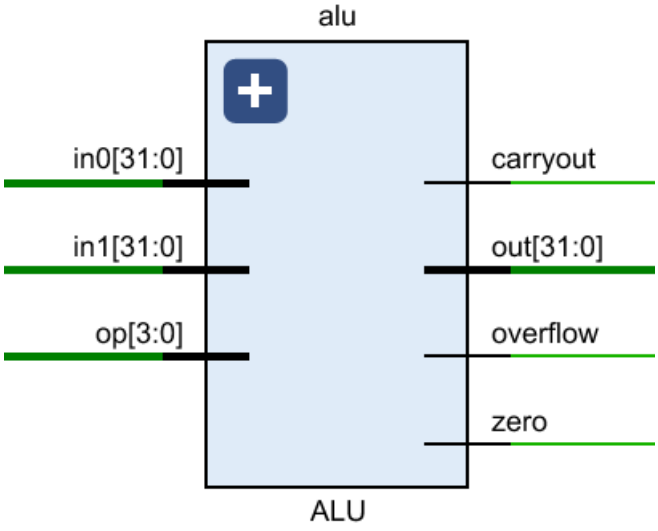
in1[31:0]：第二个数据输入

out[31:0]：输出运算结果

overflow：溢出标志

zero：零标志

carryout：进位标志



CPU 中的 ALU 与单独实现的 ALU 略有不同，因为不同的指令可以借助 ALU 中相同的操作间接实现。例如 add，addi，lw，sw 都可以借助 ALU 中 add 操作实现；and 和 andi 可以借助 ALU 中 and 操作实现；sub 和 beq 可以借助 ALU 中 sub 操作实现。

因此 CPU 中的 ALU 简化了一部分操作且自定义了 ALU 的操作码（单独实现的 ALU 中使用指令的 op 作为操作码）。

	op	op[3]	op[2]	op[1]	op[0]
add/addi/lw/sw	0000	0	0	0	0
addu/addiu	0001	0	0	0	1
sub/beq	0010	0	0	1	0
subu	0011	0	0	1	1
and/andi	0100	0	1	0	0
or/ori	0101	0	1	0	1
xor/xori	0110	0	1	1	0
nor	0111	0	1	1	1
slt	1000	1	0	0	0
sltu	1001	1	0	0	1
sllv/sll	1010	1	0	1	0
srlv/srl	1011	1	0	1	1
srav/sra	1100	1	1	0	0

```
assign ALUOp[3]=slt|sltu|sllv|sll|srlv|srl|srav|sra;
assign ALUOp[2]=And|andi|Or|ori|Xor|xori|Nor|srav|srav;
assign ALUOp[1]=sub|beq|subu|Xor|xori|Nor|sllv|sll|srlv|srl;
assign ALUOp[0]=addu|addiu|subu|Or|ori|Nor|sltu|srlv|srl;
```

(10) Ext5

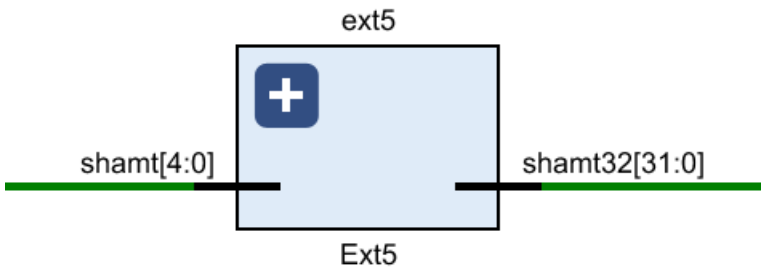
模块功能

将指令中的 5 位的移位数进行扩展至 32 位

引脚及控制信号

shamt [4:0]：移位数（5 位）shamt（输入）

shamt32 [31:0]：移位数扩展至 32 位后的结果（输出）



(11) Ext16

模块功能

用于 ori 指令的立即数无符号扩展，beq 指令的立即数符号

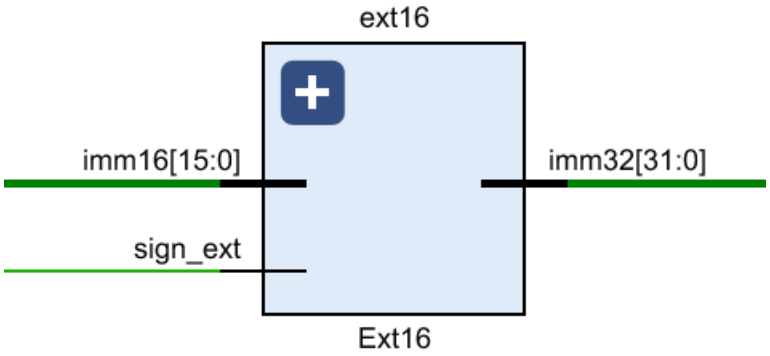
扩展以及 lw/sw 中 offset 的符号扩展。

引脚及控制信号

imm16 [15:0]：需要进行扩展的 16 位立即数（输入）

sign_ext：是否为符号扩展

imm32 [31:0]：扩展后的 32 位立即数



(12) Join

模块功能

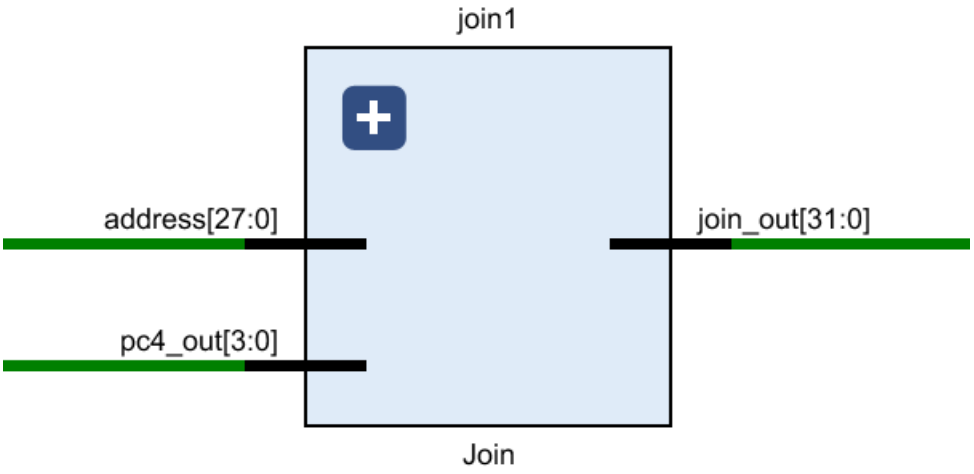
将 adress[25:0](InstructionMem 的输出)左移两位后的结果与 pc+4 高四位连接,形成 32 位的用于 J 型指令跳转的地址。

引脚及控制信号

address [27:0]：地址（输入）

pc4_out [3:0]：pc4 的输出（输入）

join_out [31:0]：组合后的值（输出）



(13) Mux_5

模块功能

多路复用器（5 位）

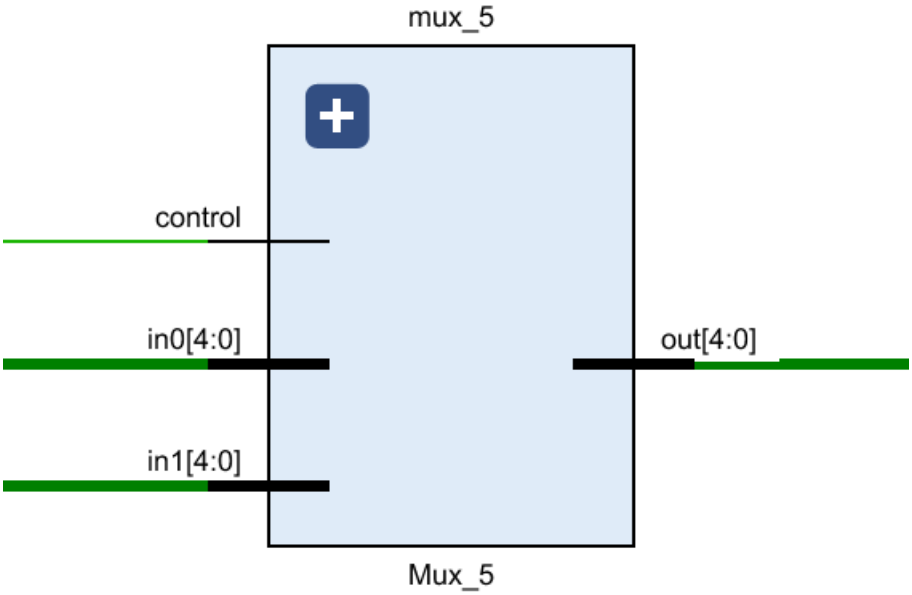
引脚及控制信号

in0 [4:0]：一路 5 位的输入

in1 [4:0]：另一路 5 位的输入

control：选择信号

out [4:0]：选择后的 5 位输出信号



(14) Mux_32

模块功能

多路复用器（32 位）

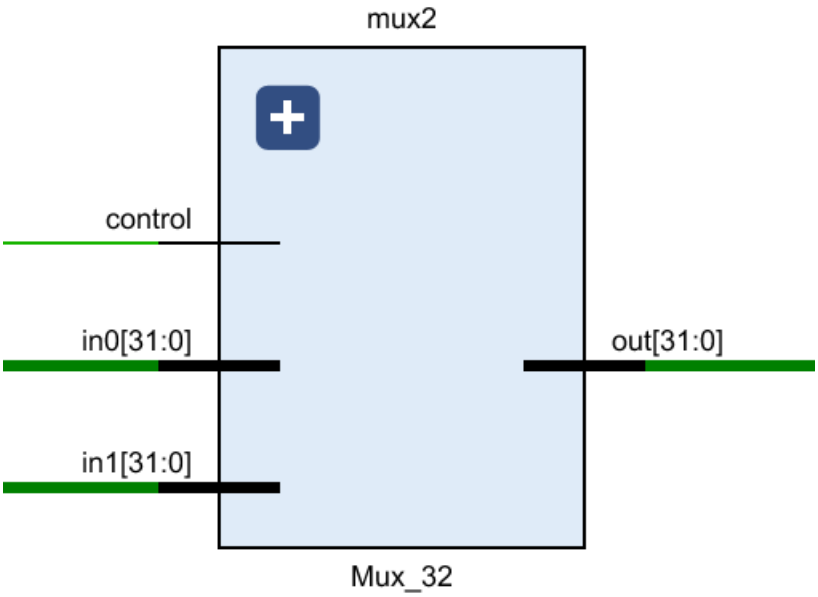
引脚及控制信号

in0 [31:0]：一路 32 位的输入

in1 [31:0]：另一路 32 位的输入

control：选择信号

out [31:0]：选择后的 32 位输出信号



(15) CPU

模块功能

整合 CPU 的各部件，从而实现 CPU 的封装。

引脚及控制信号

clk：时钟信号（输入）

rst：复位信号（输出）

clk_out：输出时钟信号（输出）

pc_out：当前指令的地址（输出）

mux5_out：32 位多路复用器输出（输出）

RF_ReadData1：寄存器组中读取的数据 1（输出）

RF_ReadData2：寄存器组中读取的数据 2（输出）

ext5_out：5 位数据扩展为 32 位的结果（输出）

ext16_out：16 位数据扩展为 32 位的结果（输出）

alu_out：ALU 运算后的结果（输出）

alu_zero：ALU 输出结果的零标志判断（输出）

alu_carryout：ALU 进位标志（输出）

ALUOp：ALU 操作码（输出）

DataMem_out：从数据存储器中读取的数据（输出）

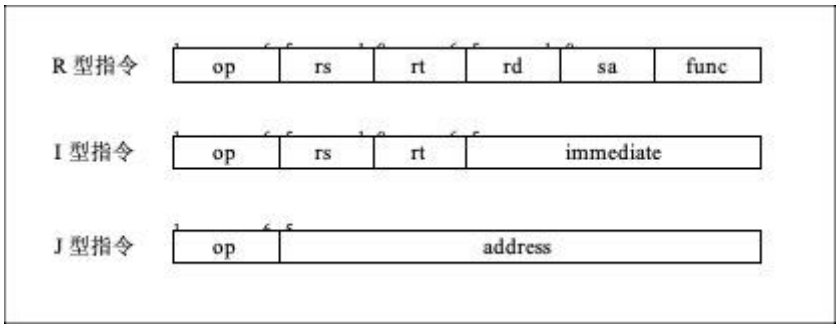
inst：当前指令

在本次 CPU 设计中将指令存储器和数据存储器作为 CPU 内部部件进行封装，因此只需要对 CPU 模块进行实例化。

```
cpu mips_cpu(clk,rst,clk_out,pc_out,mux5_out,RF_ReadData1,RF_ReadData2,Ext5_out,Ext16_out,alu_out,alu_overflow
initial begin
    clk=0;
    rst=1;
    //#200
    //rst=1;
    #60
    rst=0;
    forever begin
        #60
        clk=~clk;
    end
end
```

四、 实现内容与测试结果

本 CPU 实现了 MIPS 指令集。MIPS 指令集有 MIPS-32 和 MIPS-64 两种。本设计的 MIPS 指令选用课堂上所讲的 MIPS-32，即为 32 位。本设计覆盖了 R 型、I 型计算类、I 型取数类、I 型存数类、I 型条件判断类、J 型。



R 型指令：

add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	$1=$2+$3	rd <- rs + rt ； 其中rs = $2，rt=$3, rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+$3	rd <- rs + rt ； 其中rs = $2，rt=$3, rd=$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-$3	rd <- rs - rt ； 其中rs = $2，rt=$3, rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-$3	rd <- rs - rt ； 其中rs = $2，rt=$3, rd=$1,无符号数
and	000000	rs	rt	rd	00000	100100	and $1,$2,$3	$1=$2 & $3	rd <- rs & rt ； 其中rs = $2，rt=$3, rd=$1
or	000000	rs	rt	rd	00000	100101	or $1,$2,$3	$1=$2 | $3	rd <- rs | rt ； 其中rs = $2，rt=$3, rd=$1
xor	000000	rs	rt	rd	00000	100110	xor $1,$2,$3	$1=$2 ^ $3	rd <- rs xor rt ； 其中rs = $2，rt=$3, rd=$1(异或)
nor	000000	rs	rt	rd	00000	100111	nor $1,$2,$3	$1=~($2 | $3)	rd <- not(rs | rt) ； 其中rs = $2，rt=$3, rd=$1(或非)
slt	000000	rs	rt	rd	00000	101010	slt $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs < rt) rd=1 else rd=0 ； 其中rs = $2，rt=$3, rd=$1
sltu	000000	rs	rt	rd	00000	101011	sltu $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs < rt) rd=1 else rd=0 ； 其中rs = $2，rt=$3, rd=$1 (无符号数)
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	rd <- rt << shamt ； shamt存放移位的位数， 也就是指令中的立即数, 其中rt=$2, rd=$1
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	rd <- rt >> shamt ； (logical)，其中rt=$2, rd=$1
sra	000000	00000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	rd <- rt >> shamt ； (arithmetic) 注意符号位保留 其中rt=$2, rd=$1
sllv	000000	rs	rt	rd	00000	000100	sllv $1,$2,$3	$1=$2<<$3	rd <- rt << rs ； 其中rs = $3，rt=$2, rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1,$2,$3	$1=$2>>$3	rd <- rt >> rs ； (logical)其中rs = $3，rt=$2, rd=$1
srav	000000	rs	rt	rd	00000	000111	srav $1,$2,$3	$1=$2>>$3	rd <- rt >> rs ； (arithmetic) 注意符号位保留 其中rs = $3，rt=$2, rd=$1

I 型指令：

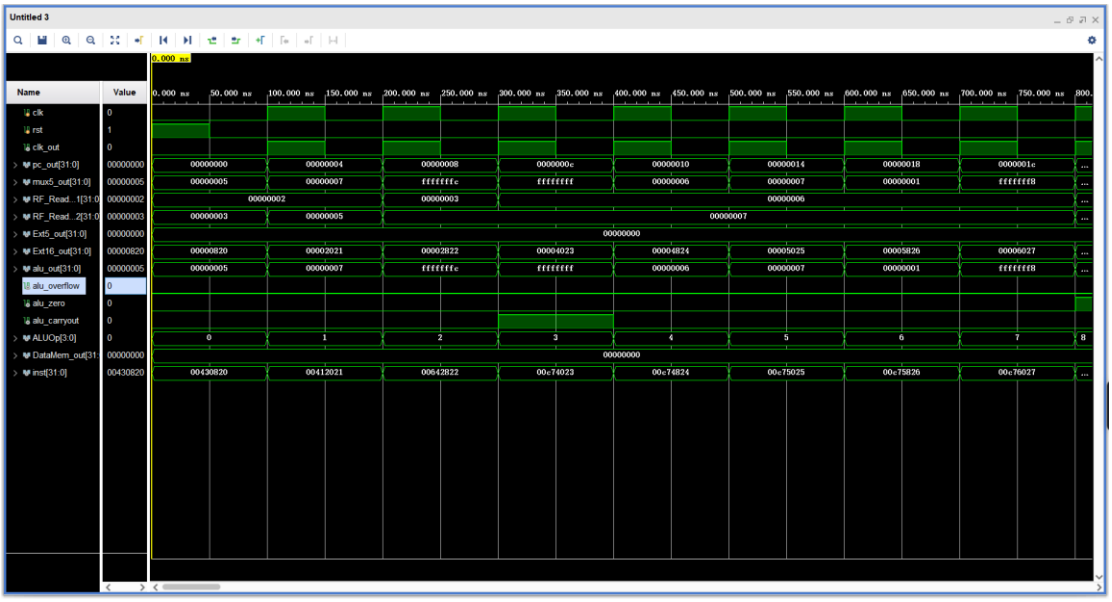
addi	001000	rs	rt	immediate	addi $1,$2,100	$1=$2+100	rt <- rs + (sign-extend)immediate ; 其中rt=$1,rs=$2
addiu	001001	rs	rt	immediate	addiu $1,$2,100	$1=$2+100	rt <- rs + (zero-extend)immediate ; 其中rt=$1,rs=$2
andi	001100	rs	rt	immediate	andi $1,$2,10	$1=$2 & 10	rt <- rs & (zero-extend)immediate ; 其中rt=$1,rs=$2
ori	001101	rs	rt	immediate	andi $1,$2,10	$1=$2 | 10	rt <- rs | (zero-extend)immediate ; 其中rt=$1,rs=$2
xori	001110	rs	rt	immediate	andi $1,$2,10	$1=$2 ^ 10	rt <- rs xor (zero-extend)immediate ; 其中rt=$1,rs=$2
lw	100011	rs	rt	immediate	lw $1,10($2)	$1=memory[$2+10]	rt <- memory[rs + (sign-extend)immediate] ; rt=$1,rs=$2
sw	101011	rs	rt	immediate	sw $1,10($2)	memory[$2+10]=$1	memory[rs + (sign-extend)immediate] <- rt ; rt=$1,rs=$2
beq	000100	rs	rt	immediate	beq $1,$2,10	if($1==$2) goto PC+4+40	if (rs == rt) PC <- PC+4 + (sign-extend)immediate<<2

J 型指令：

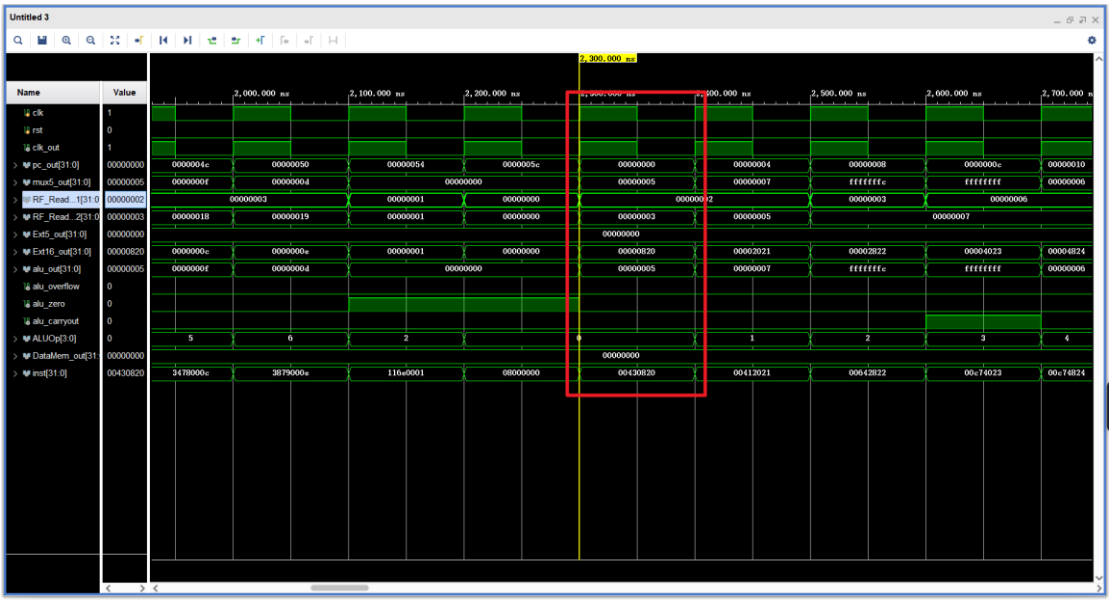
j	000010	address	j 10000	goto 10000	PC <- (PC+4)[31..28],address,0,0 ; address=10000/4

运行结果

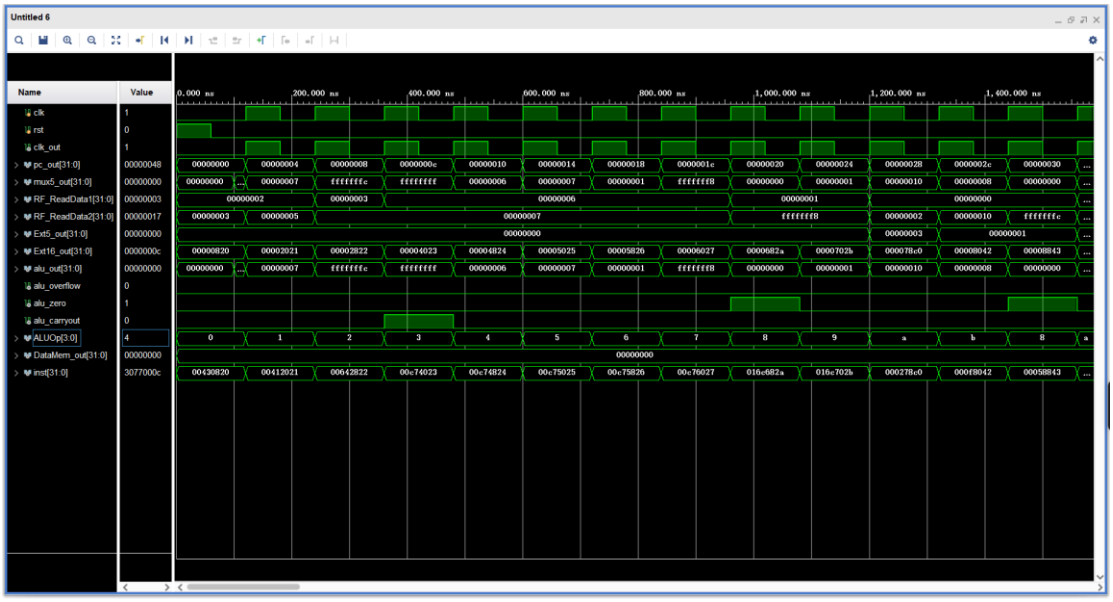
行为仿真

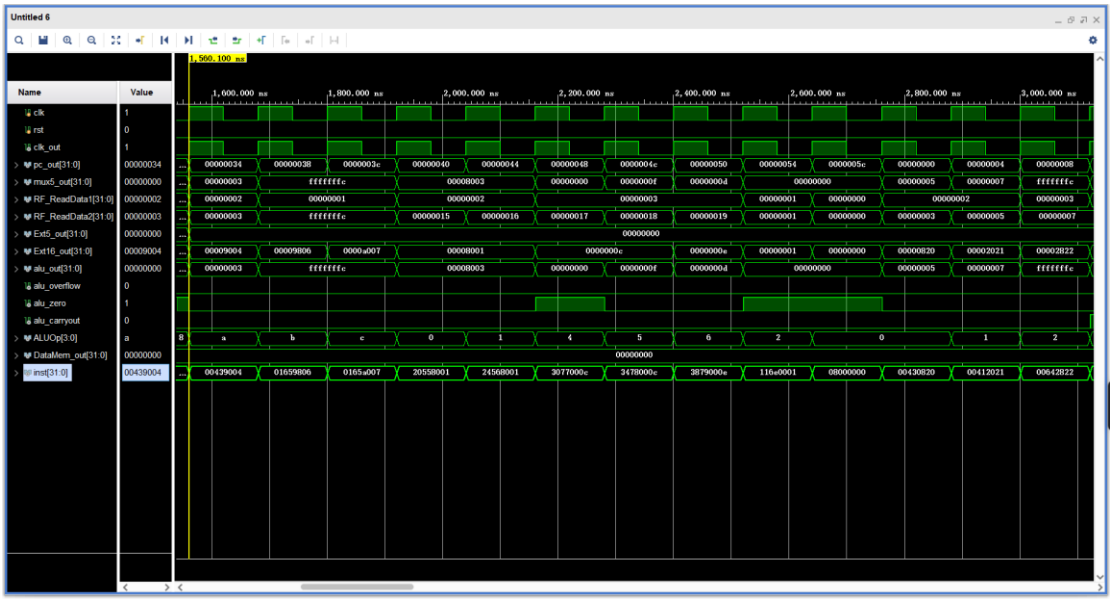


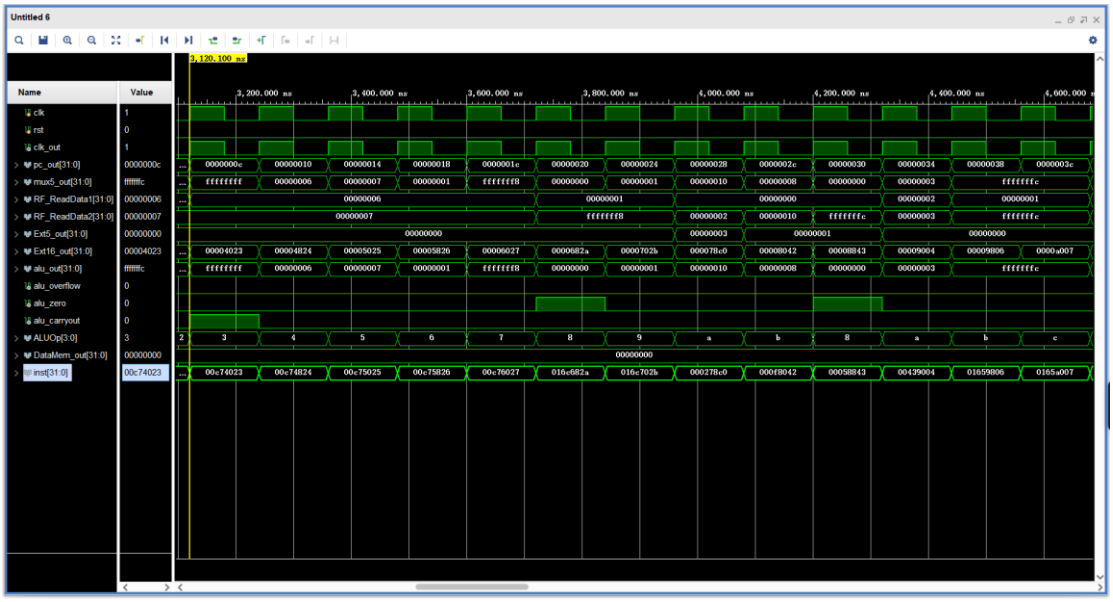
如图所示，CPU 在执行跳转指令后返回到第一条指令（pc=0）开始执行。

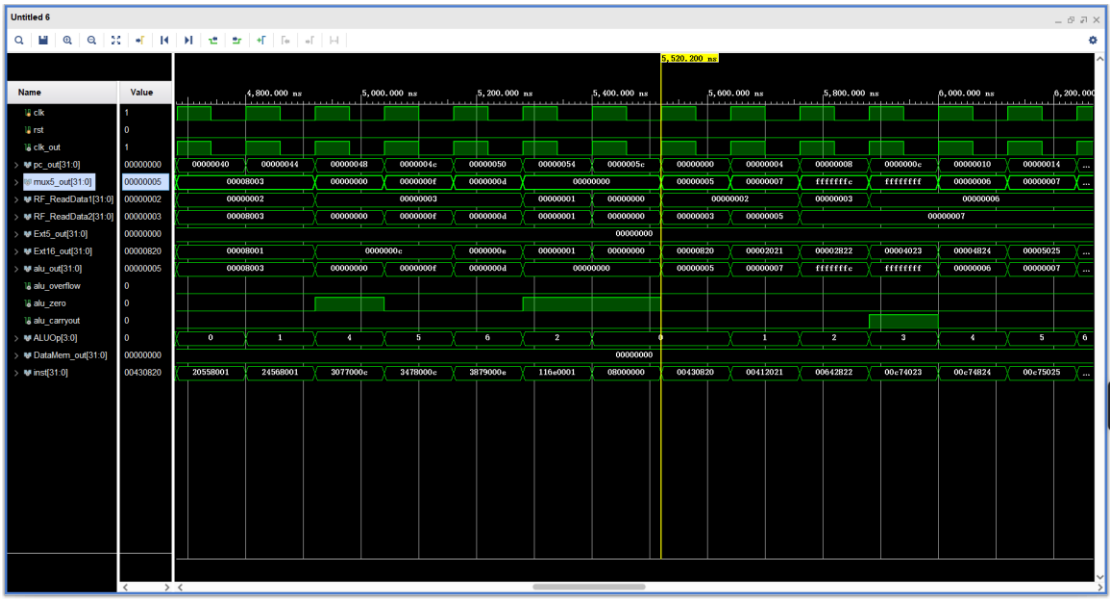


完成综合后进行功能仿真，并调整时钟信号周期，直到找到最大频率。最终将时钟周期缩短至 120ns（每 60ns 翻转一次），对应频率 8.3MHz

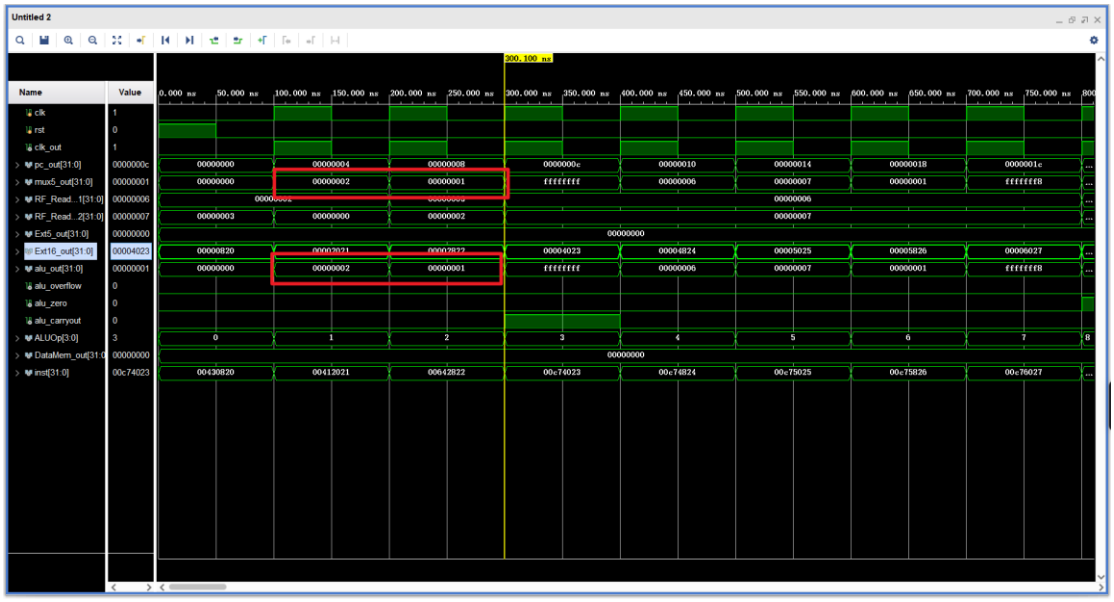

Untitled 6
Name
Value
clk
rst
clk_out
pc_out[31:0]
mux5_out[31:0]
RF_ReadData1[31:0]
RF_ReadData2[31:0]
Ext5_out[31:0]
Ext16_out[31:0]
alu_out[31:0]
alu_overflow
alu_zero
alu_carryout
ALUOp[3:0]
DataMem_out[31:0]
inst[31:0]


Untitled 6
Name
Value
clk
rst
clk_out
pc_out[31:0]
mux5_out[31:0]
RF_ReadData1[31:0]
RF_ReadData2[31:0]
Ext5_out[31:0]
Ext16_out[31:0]
alu_out[31:0]
alu_overflow
alu_zero
alu_carryout
ALUOp[3:0]
DataMem_out[31:0]
inst[31:0]

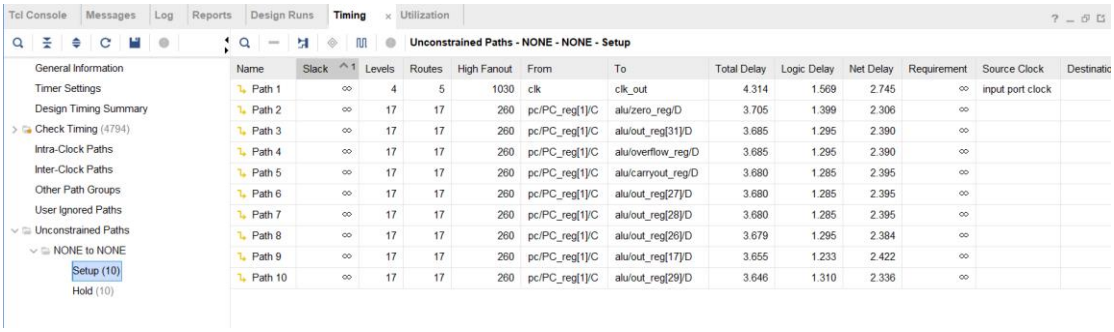




可以看到，当时钟周期降为 100ns 时结果出错



路径延迟：



Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destinatic
Path 1	∞	4	5	1030	clk	clk_out	4.314	1.569	2.745	∞	input port clock	
Path 2	∞	17	17	260	pc/PC_reg[1]/C	alu/zero_reg/D	3.705	1.399	2.306	∞		
Path 3	∞	17	17	260	pc/PC_reg[1]/C	alu/out_reg[31]/D	3.685	1.295	2.390	∞		
Path 4	∞	17	17	260	pc/PC_reg[1]/C	alu/overflow_reg/D	3.685	1.295	2.390	∞		
Path 5	∞	17	17	260	pc/PC_reg[1]/C	alu/carryout_reg/D	3.680	1.285	2.395	∞		
Path 6	∞	17	17	260	pc/PC_reg[1]/C	alu/out_reg[27]/D	3.680	1.285	2.395	∞		
Path 7	∞	17	17	260	pc/PC_reg[1]/C	alu/out_reg[28]/D	3.680	1.285	2.395	∞		
Path 8	∞	17	17	260	pc/PC_reg[1]/C	alu/out_reg[26]/D	3.679	1.295	2.384	∞		
Path 9	∞	17	17	260	pc/PC_reg[1]/C	alu/out_reg[17]/D	3.655	1.233	2.422	∞		
Path 10	∞	17	17	260	pc/PC_reg[1]/C	alu/out_reg[29]/D	3.646	1.310	2.336	∞		

五、 总结

在本次的 CPU 设计中，我们从理论到实践，实现了一个单周期的基于 MIPS 指令集的 CPU。在设计过程中，我们克服了种种困难和挫折，终于将复杂的 CPU 核拆解为一个个模块，再组装在一起，形成了本次设计。通过本次设计，我们极大地增强了对 CPU 结构的了解。

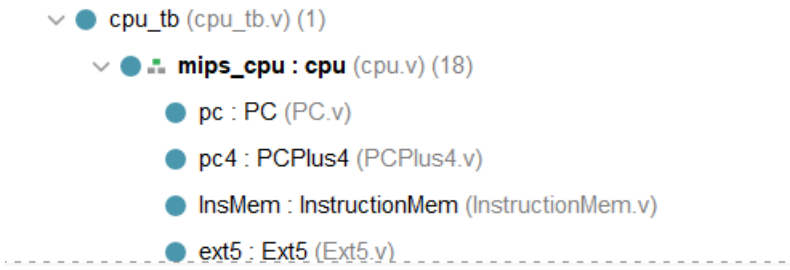
遇到的问题：

1. 指令寄存器以十六进制方式读取指令时，只能读取指令的最后两位。

   解决方案：存放指令时每两位之间添加空格或换行。

2. 进行综合时，会出现 design is empty 的错误，导致无法进行功能仿真和时序仿真。

解决方案：查找资料发现可能的原因其中一条是“顶层模块的接口只有输入信号：比如时钟和复位，没有任何输出。在这种情况下，开发工具在进行综合实现时便会将你内部逻辑全部优化掉，所以便会出现错误：design is empty”。于是将 cpu 模块置为顶层模块即可解决问题。



参考资料

1.戴维 A.帕特森(David A.Patterson) 约翰 L.亨尼斯(John L.Hennessy)《计算机组成与设计——硬件/软件接口》

2.张冬冬 王力生 郭玉臣 《数字逻辑与组成原理实践教程》

3.雷思磊《自己动手写 CPU》

4.https://blog.csdn.net/MyCodecOdecoDecodE/article/details/72670113