***2021***

****

**计算机组成原理 ·实验报告·**

|  |  |
| --- | --- |
| 学 院： | 计算机与信息学院 |
| 班 级： | 物联网19-2班 |
| 学 号： | 2019217819 |
| 姓 名： | 付炎平 |
| 电 话： | 13225636828 |
| 邮 件： | [2402636523@qq.com](mailto:2402636523@qq.com) |
| 完成日期： | 2021-07-11 |
| 指导教师： | 阙 夏 |

目 录

[1 实验二、CPU 部件实现之 ALU 和寄存器堆 2](#_Toc76893060)

[1.1 设计要求 2](#_Toc76893061)

[1.2 方案设计 2](#_Toc76893062)

[1.3 实验步骤 3](#_Toc76893063)

[1.4 故障与调试 7](#_Toc76893064)

[1.5 仿真及分析 8](#_Toc76893065)

[2 实验四、单周期 CPU 设计与实现——单指令 CPU 12](#_Toc76893066)

[2.1 设计要求 12](#_Toc76893067)

[2.2 方案设计 12](#_Toc76893068)

[2.3 实验步骤 14](#_Toc76893069)

[2.4 故障与调试 22](#_Toc76893070)

[2.5 仿真与分析 22](#_Toc76893071)

[3 总结与心得 24](#_Toc76893072)

[3.1 实验总结 24](#_Toc76893073)

[3.2 实验心得 24](#_Toc76893074)

[参考文献 25](#_Toc76893075)

# 实验二、CPU 部件实现之 ALU 和寄存器堆

## 设计要求

1. 使用 Verilog 完成 ALU 的设计,并编写测试仿真文件验证其正确性。要求： λ ALU 支持 16 位的加、减、与、或以及移位运算。

2. 使用 Verilog 完成通用寄存器堆的设计，并编写测试仿真文件验证其正确性。要求 λ 寄存器堆包含 8 个 16 位的寄存器； λ 寄存器堆有两个读端口和一个写端口。

## 方案设计

### ALU设计

1. 我们要ALU能够分别进行非、与、或、左移、右移、加、减这七种运算，可以利用一个3位选择信号表示8选1选择器，表示选择这7种情况，同时确定输入操作数位数和输出结果位数，其中输入操作数的位数应该和输出结果的位数相同。
2. 确定输入和输出端，输入端需要一个使能端，3位选择端，两组16位输入操作数，一组16位的输出结果端。
3. 我们在时钟周期里首先判断使能端是否为1，当它为1时，我们判断选择端信号，3位选择端信号的不同我们来把他分别表示我们不同的运算，最后得到结果。

### 寄存器堆设计

1. 寄存器分别表示读和写，写是把数据写到寄存器，需要输入数据，把输入数据写到哪个寄存器中需要选择；读是读出寄存器里的数据，需要输出数据，输出哪个寄存器里的数据也要进行选择。
2. 输入需要2个读端口和1个写端口，16位的输入数据端口用来写到寄存器中，16位的输出数据端口表示寄存器里的数据写到该输出端口，2位的选择端口进行选择输入输出寄存器，其中用AX，BX，CX，DX，SP，BP，SI，DI表示8个通用寄存器.
3. 在时钟周期里，当使能引脚为1时，判断读和写，当写信号为1时，利用2位的选择端口进行选择输入数据存到哪个寄存器，其中当一个读信号为1时，用2位选择端口选择前4个寄存器，当另一个读信号为1时，用2为的选择端口选择后4个寄存器。

## 实验步骤

1. Verilog 关键代码描述

ALU:

`timescale 1ns / 1ps

module ALU(

input wire EN,//定义使能引脚

input wire [2:0] select,//定义选择端口

input wire [15:0] src1,src2,//定义输入数据端

output reg [15:0] out//定义输出数据端

);

always@(\*)

begin

if(EN == 1'b1)

begin

case(select)

3'b000: out <= ~src1; // 16位非运算

3'b001: out <= src1 & src2; // 16位与运算

3'b010: out <= src1 | src2; // 16位或运算

3'b011: out <= src1 << src2; // 16位左移

3'b100: out <= src1 >> src2; // 16位右移

3'b101: out <= src1 + src2; // 16位加法器

3'b110: out <= src1 - src2; // 16位减法器

default: $display("[ERROR]select=%b",select);

endcase

end

end

endmodule

寄存器堆：

`timescale 1ns / 1ps

module RegisterFile(

input wire EN,//定义使能端

input wire WR,//定义可写端口

input wire [15:0] data,//定义输入数据端

input wire RD1, RD2,//定义可读端口

input wire [1:0] select,//定义选择端口

output reg [15:0] AX,BX,CX,DX,SP,BP,DI,SI,//定义输出寄存器端

output reg [15:0] out//定义输出数据端

);

always@(\*)

begin

if(EN == 1'b1)//使能有效

begin

if(WR == 1'b1)//写有效

begin

case(select)//选择信号

2'b00: AX <= data;

2'b01: BX <= data;//数据写到寄存器

2'b10: CX <= data;

2'b11: DX <= data;

endcase

end

if(RD1 == 1'b1)//读有效

begin

case(select)//选择信号

2'b00: out <= AX;

2'b01: out <= BX;//前四个寄存器的数据到输出端

2'b10: out <= CX;

2'b11: out <= DX;

endcase

end

if(RD2 == 1'b1)//读有效

begin

case(select)//选择信号

2'b00: out <= SP;

2'b01: out <= BP;//后四个寄存器的数据到输出端

2'b10: out <= DI;

2'b11: out <= SI;

endcase

end

end

end

endmodule

1. 测试文件描述

ALU测试：

module ALU\_tb(

reg [15:0] inData\_1, inData\_2;//定义ALU输入输出

wire [15:0] result;

wire [2:0] select;

wire EN;

initial begin//初始化输入数据

inData\_1 = 16'b0101010101010101;

inData\_2 = 16'b1010101010101010;

EN=1'b1;

select=3'b011;

end

ALU test1(//调用ALU模块

.src1(inData\_1), .src(inData\_2),

.EN(EN),

.out(result),

.select(select)

);

endmodule

寄存器堆测试：

module Register\_tb();

wire EN; //定义输入输出数据

wire WR;

wire [15:0] data;

wire RD1,RD2;

wire [1:0] select;

reg [15:0] AX,BX,CX,DX,SP,BP,SI,DI;

reg [15:0] out;

initial begin //数据初始化

EN=1'b1;

WR=1'b1;

RD1=1'b0;

RD2=1'b0;

select=2'b01;

data=16'b0101110010101100

end

RegisterFile test1(//调用寄存器堆模块

.data(data), .EN(EN),

.WR(WR),

.RD1(RD1),

.RD2(RD2),

.data(data),

.select(select)

.out(out)

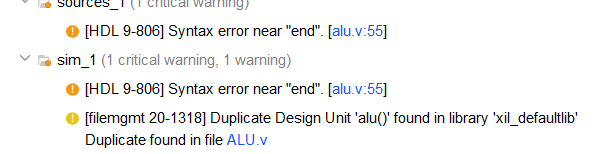
);

endmodule

## 故障与调试

### 故障1

**故障现象：**

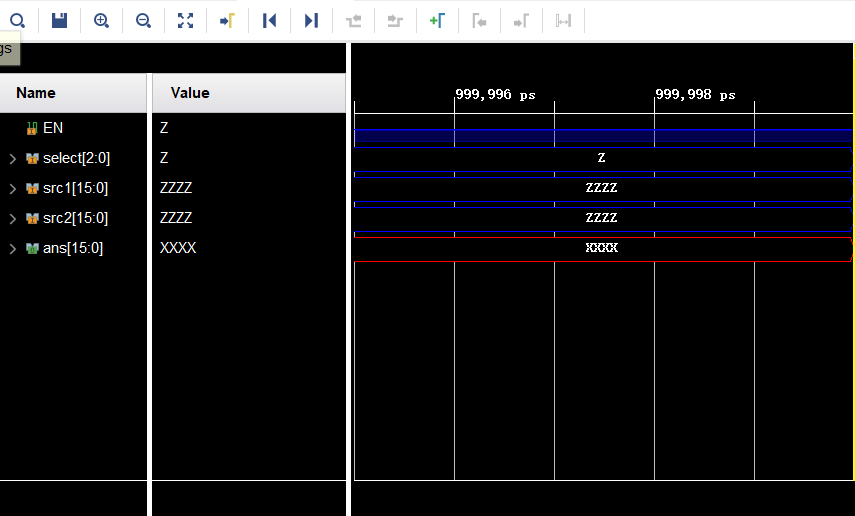


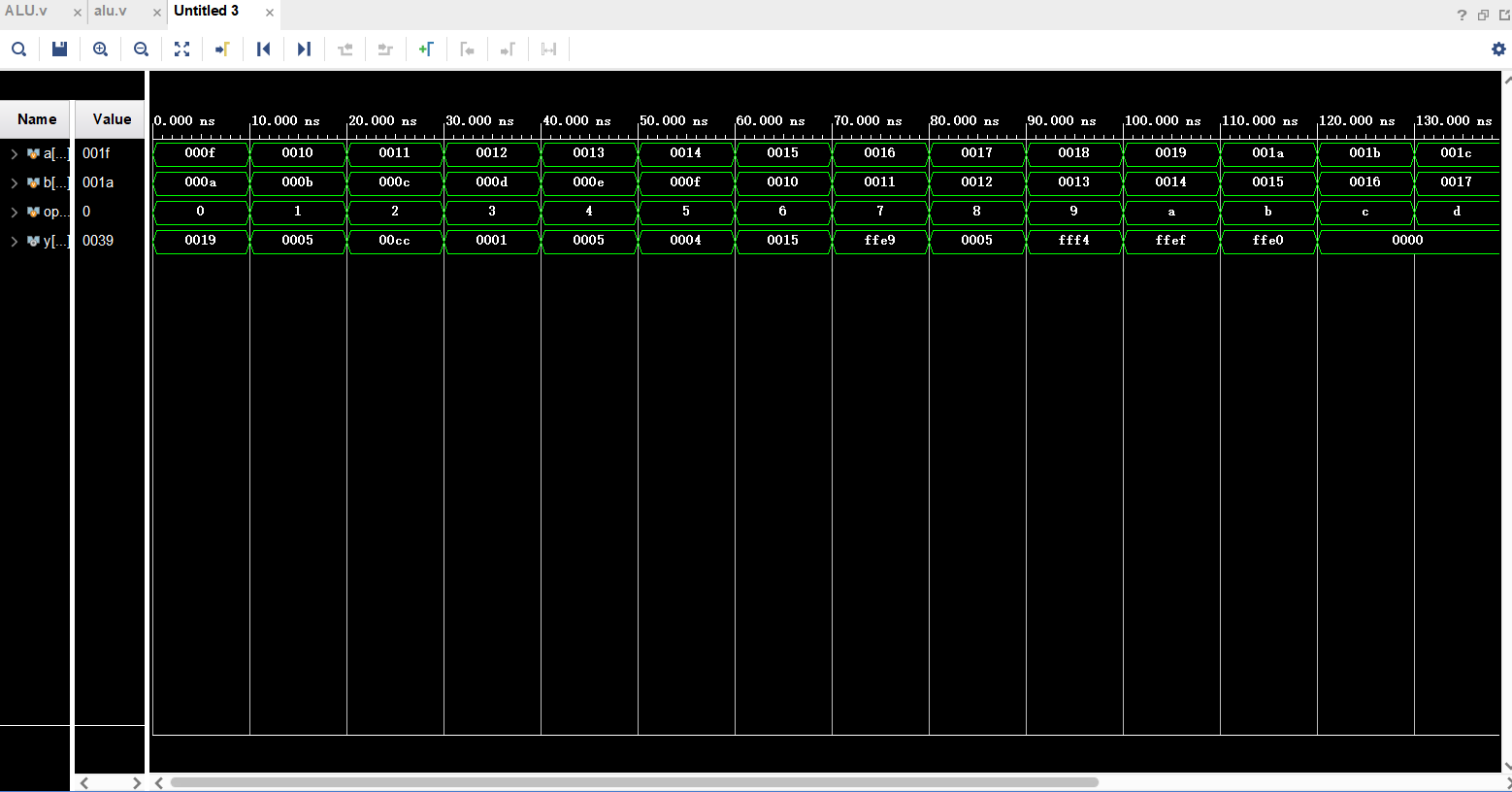
**原因分析：** begin后面没有加上end

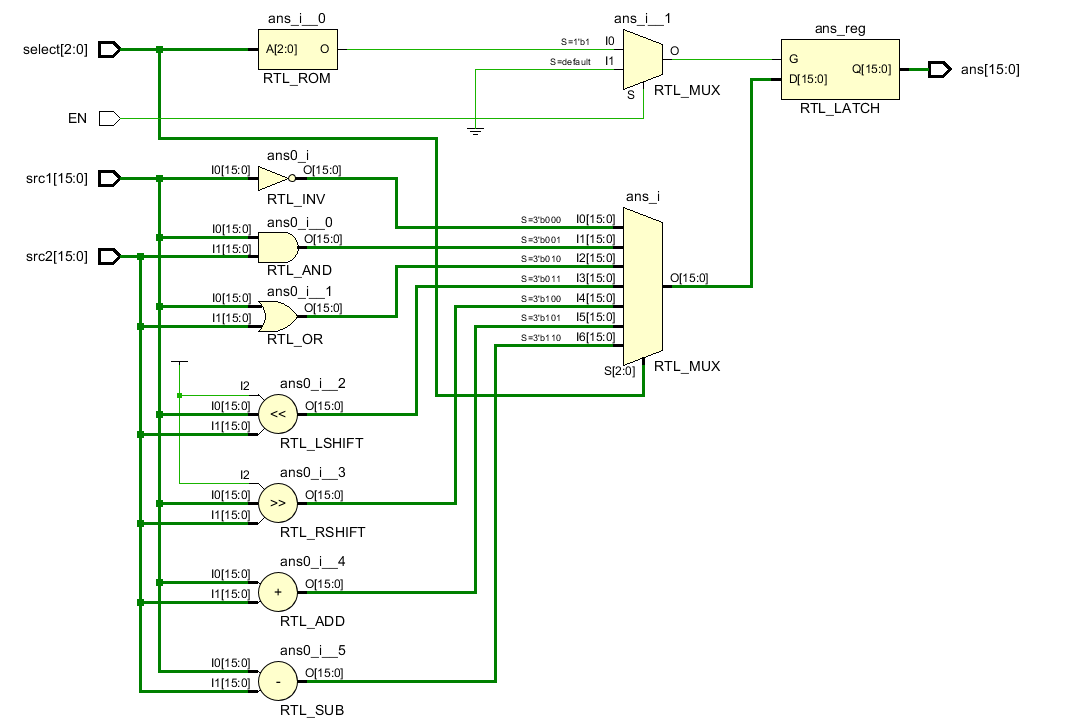
**解决方案：** 在相应的begin后面加上end

## 仿真及分析

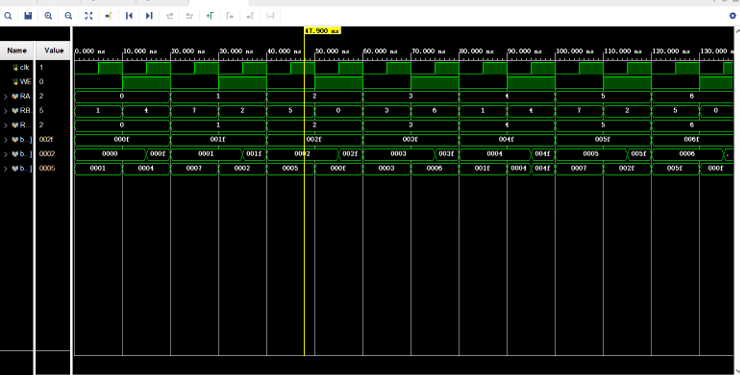
### ALU



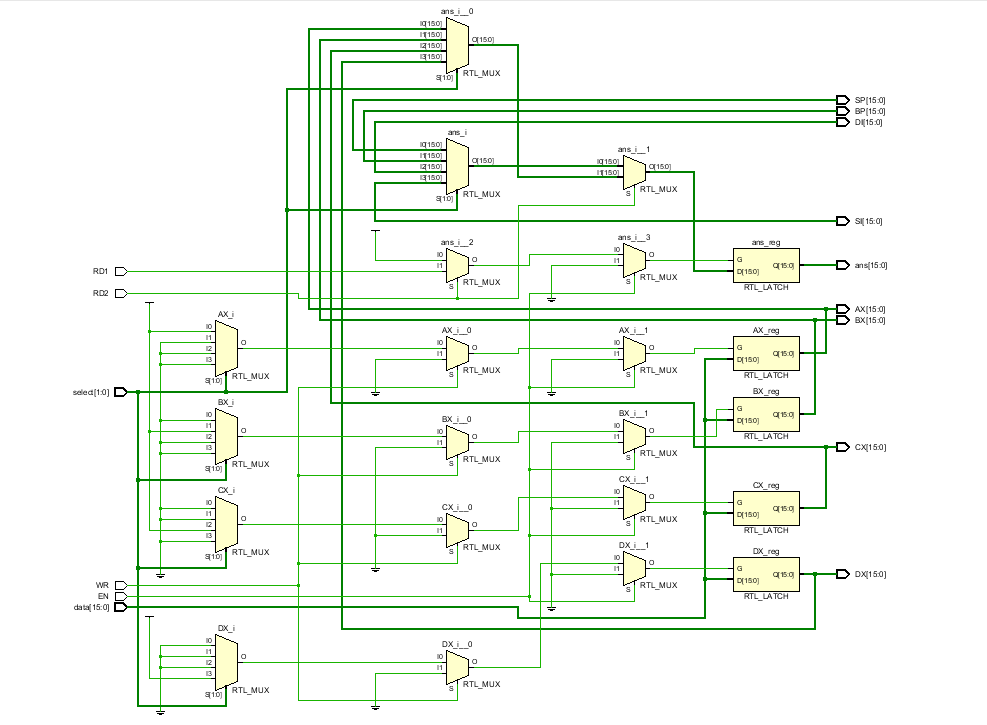
src1,src2分别是输入的两个参数，select为要进行的动作的信号，当select=3b101时，进行加法运算，当select=3b001进行与运算。每10ns会换一个操作以及输入的参数是src1,src2。依次将所有操作都进行一遍。以第一个为例，输入的开两个值分别是000f,000a, ,进行的是加运算。结果就是ox19，我们可以根据最后的模拟仿真结果发现最后的输出y是正确的。其他周期的也是如此。



### 寄存器堆



通过模拟仿真的结果，我们可以发现读数据是读正确的，写数据也是找到对应的寄存器中写入的。而且为了方便观察写操作的正确，我特意让读的单元和写的单元是同一个。在模拟仿真的测试文件中就是让read\_reg1和write\_reg相等，在途中显示的就是RA和RW相同，我们看最后的BusA结果也可以知道刚开始是读取的原来内存的值，但是在新的值被写进去之后，就读取出来新的值。比如说第一个小周期，BusA刚开始是0，但是后来我们将写入000f，之后又改变成000f。之后的同理。而BusB读取的就是RB对应的地址内容，可以发现是和存储器的值一样的（存储器刚开始的内容就是对应的标号）。 所以，我们就能观察到上述的图像。这个图像是符合我们的逻辑的。



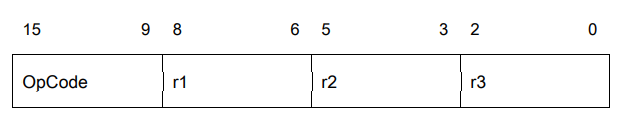
# 实验四、单周期 CPU 设计与实现——单指令 CPU

## 设计要求

设计和实现一个支持加法指令的单周期 CPU。要求该加法指令（表示为 add r1，r2， r3）格式约定如下：

采用寄存器寻址，r1，r2，r3 为寄存器编号，r1 和 r2 存放两个源操作数，r3 为目标寄存器，其功能为[r1] + [r2] -> r3；

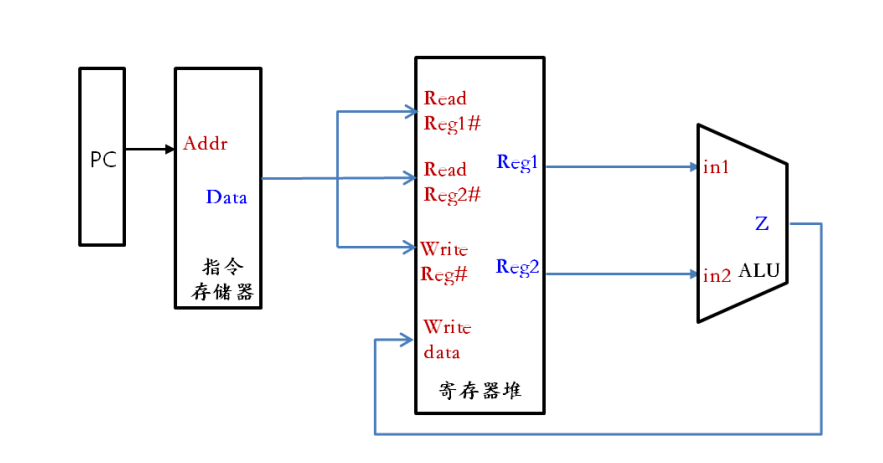
指令字长 16 位，操作码和地址码字段分配如下所示：



## 方案设计

### 根据功能和格式完成 CPU 的数据通路设计

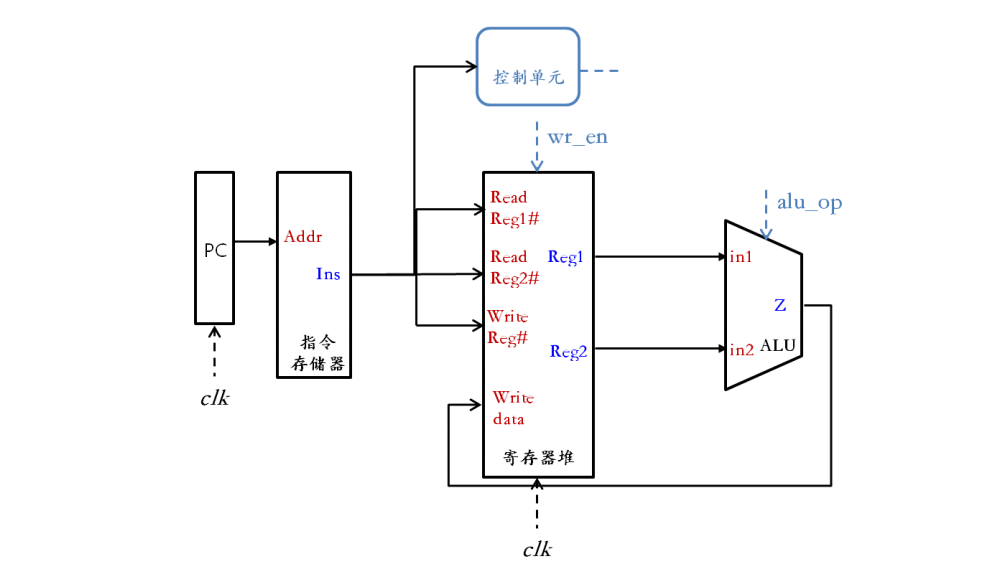
本实验需要设计的 CPU 只需要支持一条加法指令，而该指令的功能是在一个时钟周期内从寄存器组中 r1 和 r2 中取出两个操作数，然后送到 ALU 进行加法运算，最后把计算结果保存到 r1 寄存器中。下图给出了改加法指令的数据通路图。



此外，还需要确定各个部件的位数，为了简单起见，我们假设目标 CPU 的机器字长、存储字长和指令字长相等均为 16 位，存储单元个数假设为 256，按字寻址，并取 PC 位数为8。

### 根据指令功能、数据通路完成控制单元的设计

控制单元的功能是为当前要执行的指令产生微操作命令从而完成该指令的执行。为了能够完成加法指令的执行，结合图 1，控制单元需要在取出指令后根据指令操作码（本例中是加法指令），控制 ALU（参考实验二）做加法（通过给 alu\_op 信号线相应赋值），并把结果写回寄存器组（参考实验三）中（通过给 wr\_en 赋值为 true）。图 2 给出了整合控制单元后目标 CPU 的原理图，系统时钟信号也已标注。



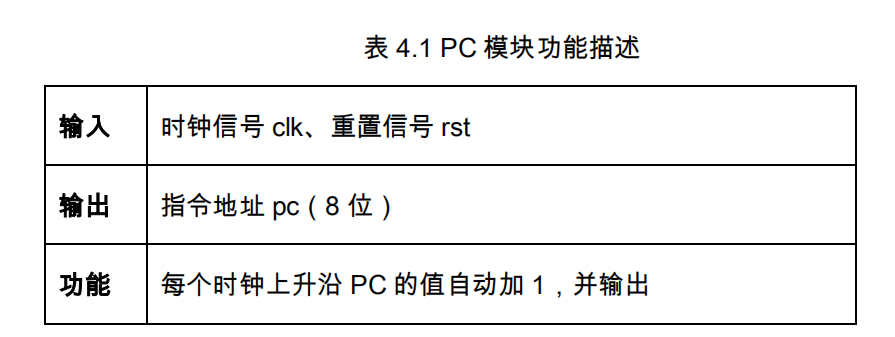
## 实验步骤

在第三部分通过对该 CPU 实现细节的分析、设计，并得到该 CPU 的原理图后，就可以依次实现各个模块，并进行仿真验证了。

1. CPU 各模块 Verilog 实现

在前面实验中，已经分别设计和实现了 PC、指令存储器、寄存器组和 ALU，接下来会进行一个总结。

1）PC 模块



Verilog 关键代码：

module pc(clk,rst,pc);

input clk,rst;

output pc;

reg[7:0] pc;

initial pc = 0;

always@(posedge clk or negedge rst)begin

if(!rst) //表示是否置0

pc = 0;

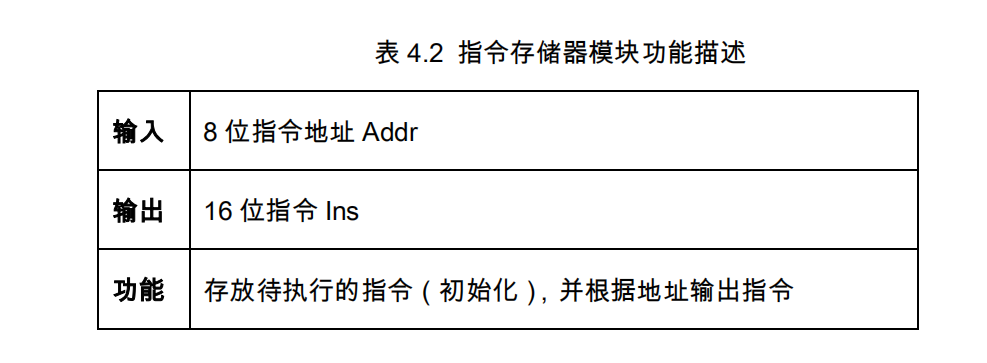
else

pc = pc + 1;

end

endmodule

2）指令存储器模块



Verilog 关键代码：

module ram(data\_write,address,data\_read);

input WE;

input[15:0] data\_write;

input[7:0] address;

output data\_read;

reg[15:0] data\_read;

reg[15:0] r[255:0];

integer i;

initial begin

for(i=0;i<256;i=i+1)

r[i] = i;

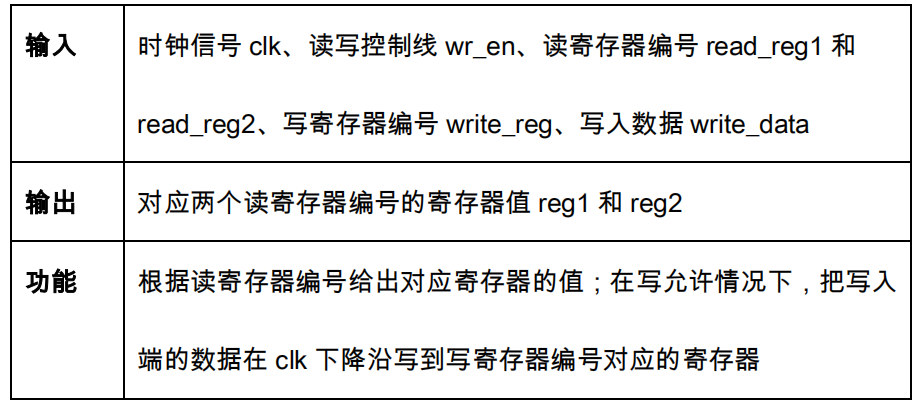
end

always@(\*)begin

data\_read = r[address];

end

Endmodule

1. 寄存器堆 

Verilog 关键代码：

module regFile(clk,wr\_en,read\_reg1,read\_reg2,write\_reg,busA,busB,write\_data);

input clk,wr\_en;

input[2:0] read\_reg1,read\_reg2,write\_reg;

input[15:0] write\_data;//要写入寄存器堆的内容

output[15:0] busA,busB;//从寄存器堆读出的内容

reg[15:0] regfile[7:0];//寄存器堆

integer i;

initial begin // 初始化寄存器堆

for(i=0;i<8;i=i+1)

regfile[i] = i;

end

always@(posedge clk)

begin

if(wr\_en == 1) //写操作

regfile[write\_reg] = write\_data;

end

assign busA = regfile[read\_reg1];

assign busB = regfile[read\_reg2];

endmodule

1. ALU

ALU运算单元描述

|  |  |
| --- | --- |
| 输入 | 操作数a,b，操作选择信号op |
| 输出 | ALU运算结果y |
| 功能 | 根据操作选择信号计算a,b的运算结果 |

Verilog 关键代码：

module alu(a,b,op,y);

input[15:0]a,b;

input[2:0]op;

output y;

reg[15:0] y;

always@(\*)begin

case(op)

//算术运算

4'd0: y = a + b;

4'd1: y = a - b;

4'd2: y = a \* b;

4'd3: y = a / b;

4'd4: y = a % b; //求余

//位运算

4'd5: y = a & b; //与

4'd6: y = a | b; //或

4'd7: y = ~a; //非

4'd8: y = a ^ b; //异或

4'd9: y = a ~^ b; //同或

4'd10: y = ~(a & b);//与非

4'd11: y = ~(a | b);//或非

//移位运算

4'd12: y = a << b;//逻辑左移和算术左移结果相同

4'd13: y = a >> b;//逻辑右移

4'd14: y = a >>> b;//算术右移

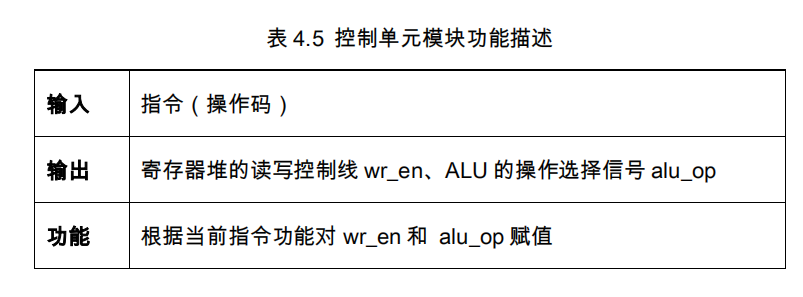
4'd15: y = 0;

endcase

end

endmodule

5）控制单元



Verilog 关键代码：

module cu(

input wire [6:0] Ins\_op,

output reg wr\_en,

output reg [2:0] alu\_op

);

always@\* begin

if(Ins\_op == 0)

wr\_en = 1;

alu\_op = 3'b000;

end

endmodule

1. CPU 顶层文件封装实现

通过根据图 2 将以上定义的模块进行连接、封装就得到了目标 CPU，该 CPU 的输入为系统时钟信号 clk 和重置信号 reset。

Verilog 关键代码：

module cpu(

clk,rst

);

input wire clk,rst;

wire wr;

wire[2:0] op;

wire[7:0] addr;

wire[15:0] y,ins,r1,r2;

pc pc(.clk(clk),.rst(rst),.pc(addr));

ram ram(.address(addr),.data\_read(ins));

cu cu(.ins\_op(ins[15:9]),.wr\_en(en),.alu\_op(op));

regFile regFile(.clk(clk),.wr\_en(wr),.read\_reg1(ins[8:6]),

.read\_reg2(ins[5:3]),.write\_reg(ins[2:0]),.write\_data(y),.busA(r1),.busB(r2));

alu alu(.a(r1),.b(r2),.op(op),.y(y));

endmodule

1. CPU 模拟仿真

为了仿真验证所实现的 CPU，需要定义测试文件并在测试文件中对指令存储器和寄存器堆中的相应寄存器的值进行初始化，并通过仿真波形图查看是否指令得到了确执行。

1. TestBench 关键代码：

module cpu\_test;

reg clk, rst;

wire[15:0] z;

always #1 clk = ~clk;

initial begin

clk = 1;

rst = 1;

#1 rst = 0;

#10 $stop;

end

cpu uut(

.clk(clk), .rst(rst),.y(z)

);

endmodule

## 故障与调试

### 故障1

**故障现象：**



**原因分析：** 可能是因为存储器的内容不合适或者代码出现错误造成的。

**解决方案：** 通过又查看命令的存储器的内容发现，确实是因为存储的内容造成的，由于之前实验的时候没有考虑很多，所以造成了部分内容没办法适应现在的需求。而且发现由于使用的是之前的接口，所以会出现有一些模块的参数没有使用，比如说就是命令存储器中的we信号，这个控制着读写，但是后来在写的时候发现这个引脚没有使用。根据题意，这个实验不需要写命令，我就把这个写命令的相关代码和引脚删除了。

## 仿真与分析



在这里，为了方便查看结果，我们将运算的结果和输入的命令调出来看一下。根据ins给出的结果，我们很容易就知道参与运算的两个数据以及需要放入的寄存器编号。我们先看第一个结果，根据ins我们很容易知道是2号和1号编号的寄存器放入到0号寄存器中。而参与运算的两个寄存器内容分别是2和1，然后我们又发现op是0，则参与的是加运算。看到最后的结果z，我们发现是3，结果是对的。后边的结果也是如此。从这个图中我们就可以知道整个系统已经实现了。

# 总结与心得

## 实验总结

1. 完成方案总结（完成了单指令的CPU设计）（完成了实验二CPU部件实现之ALU和寄存器堆）（pc运算）（命令存储器）（cu控制初始化参数变量）（ALU的几种运算）（寄存器的读写操作）。
2. 功能总结（CPU加法计算）（ALU中包括的常见运算）（pc实现了不断+1，指向地址）（寄存器堆实现了对寄存器的读写）（命令存储器实现了对命令的存储读出）。
3. 其他需要总结的内容，（ALU和寄存器堆的测试文件测试了ALU和寄存器的运行情况，cpu的测试文件体现了cpu的所有功能）。

## 实验心得

在设计ALU时，需要考虑寄存器的几种基本运算，例如与、或、非、加法、减法、移位运算，并且考虑用选择器选择这几种情况，同时掌握运算前后输入输出操作数的位数。在设计寄存器时，要考虑寄存器的读和写操作，并用选择器选择这几种操作，同时考虑吧使能信号来控制数据有效。在设计单周期CPU时，需要把CPU分几个模块，分别实现ALU计算，CU控制，PC加一，存储器存储等情况。

这次实验让我堆Verilog语言有了更加深入的理解，同时掌握了怎样使用Vivado进行仿真和分析仿真结果，这次实验让我们动手编写代码进行仿真，和上课学习理论的感受有大的不同，让我对计算机内部的组成有了更深的理解，同时对计算机内部的执行也有了更深刻的感受，掌握了Verilog的编程，在选择输入输出端时锻炼了我逻辑思考的能力，这次实验让我收获很多，提高了我的能力，我希望《计算机组成原理》能开展的越来越好。

# 参考文献

1. 谭志虎,秦磊华,胡迪青.计算机组成原理实践教程.北京:清华大学出版社，2018年.
2. 秦磊华，吴非，莫正坤.计算机组成原理. 北京:清华大学出版社，2011年.
3. 张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008年.

|  |
| --- |
| 一、原创性声明 |
| 本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。  特此声明！  **作者签字: 嵌入签名图片** |
| 二、对课程实验的学术评语（教师填写） |
|  |
| 三、对课程实验的评分（教师填写） |
| |  |  |  |  | | --- | --- | --- | --- | | 评分项目  （分值） | 报告撰写  （30分） | 课设过程  （70分） | 最终评定  （100分） | | 得分 |  |  |  | |
| **指导教师签字:** |