***2021***

****

**计算机组成原理 ·实验报告·**

|  |  |
| --- | --- |
| 学 院： | 计算机与信息学院 |
| 班 级： | 计算机19-1班 |
| 学 号： | 2019210000 |
| 姓 名： | 范XX |
| 电 话： | 138XXXXXXXX |
| 邮 件： | [134567890@qq.com](mailto:134567890@qq.com) |
| 完成日期： | 2021-06-05 |
| 指导教师： | 阙 夏 |

目 录

[1 Verilog与vivado基础 3](#_Toc69573810)

[1.1 设计要求 3](#_Toc69573811)

[1.2 方案设计 3](#_Toc69573812)

[1.3 实验步骤 5](#_Toc69573813)

[1.4 故障与调试 6](#_Toc69573814)

[1.5 仿真及分析 6](#_Toc69573815)

[2 XXX实验 8](#_Toc69573816)

[2.1 设计要求 8](#_Toc69573817)

[2.2 方案设计 8](#_Toc69573818)

[2.3 实验步骤 8](#_Toc69573819)

[2.4 故障与调试 8](#_Toc69573820)

[2.5 仿真及分析 9](#_Toc69573821)

[3 XXX实验 10](#_Toc69573822)

[3.1 设计要求 10](#_Toc69573823)

[3.2 方案设计 10](#_Toc69573824)

[3.3 实验步骤 10](#_Toc69573825)

[3.4 故障与调试 10](#_Toc69573826)

[3.5 仿真与分析 11](#_Toc69573827)

[4 XXX实验 12](#_Toc69573828)

[4.1 设计要求 12](#_Toc69573829)

[4.2 方案设计 12](#_Toc69573830)

[4.3 实验步骤 12](#_Toc69573831)

[4.4 故障与调试 12](#_Toc69573832)

[4.5 仿真与分析 13](#_Toc69573833)

[5 总结与心得 14](#_Toc69573834)

[5.1 实验总结 14](#_Toc69573835)

[5.2 实验心得 14](#_Toc69573836)

[参考文献 15](#_Toc69573837)

# CPU部件实现之ALU和寄存器堆实验

## 设计要求

理解和掌握 CPU 中的算术逻辑运算部件（ALU）和寄存器堆（Register File）的工作原理，并使用 Verilog 和 ModelSim 进行设计和仿真。

## 方案设计

ALU方案:

ALU 支持 16 位的加、减、与、或以及移位运算。根据ALU输入的参数，我们选择不同的功能。输入分别是参与运算的两个数，指令，以及输出。我们根据指令的不同选择不同的功能。

寄存器堆：

寄存器堆实际上就是根据输入的参数得到或者改变相应寄存器的内容。输入就是两个读寄存器的编号以及一个使能信号en，如果是能信号en =1，那么久进行写操作。根据输入的需要写的寄存器的编号，找到对应的寄存器，然后再根据输入的需要写进去的参数，将数据写入。但是读信号是始终有的。这时候，两个输出就是得到本次运行得到的两个相应寄存器的内容。

## 实验步骤

1. Verilog 关键代码描述

|  |  |
| --- | --- |
| 输入 | 指令编号op，参数a,b |
| 输出 | y |
| 功能 | 根据op实现不同的运算 |

Alu:

always@(\*)begin

case(op)

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

寄存器堆：

initial begin //对寄存器内容进行初始化

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

r[i][15:9] = 0;

r[i][2:0] = i%5;

r[i][5:3] = i%5 + 1;

r[i][8:6] = i%5 + 2;

end

always@(\*)begin

if(WE == 1) // 低电平为写信号，根据不同的信号选择不同的操作

data\_read = r[address];

else

r[address] = data\_write;

end

1. 测试文件描述

Alu:

module ALU;

reg [15:0] a,b;

reg [3:0] op;

wire [15:0] y;

initial begin //初始化参与的参数

a = 15;b = 10;op = 0;

#170 $stop;

end

alu f(.a(a),.b(b),.op(op),.y(y)); //调用模块进行运算

always begin //改变参数继续运算，得到不同结果

#10 a = a + 1;

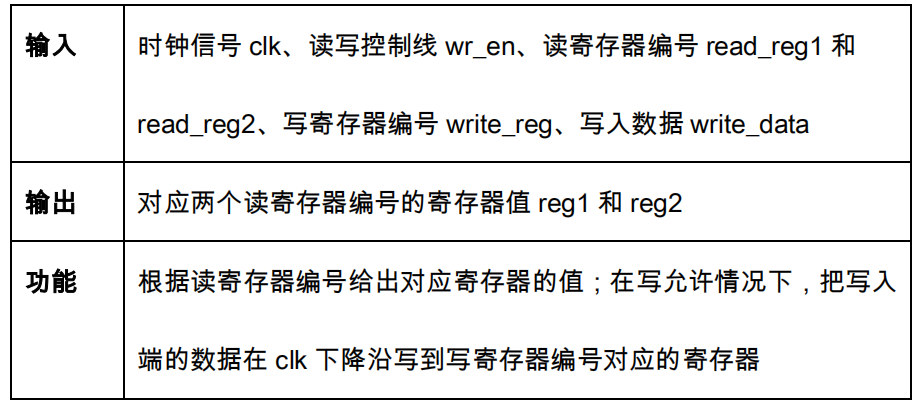
b = b+1;

op = op+1;

end

Endmodule

寄存器堆：



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];

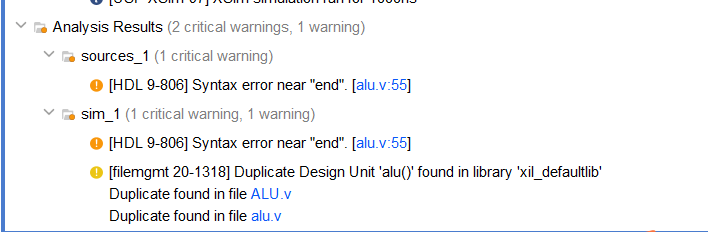
assign busB = regfile[read\_reg2];

endmodule

1. 进行模拟仿真测试。

## 故障与调试

### 故障1

**故障现象：**

**编译报错，不能仿真。**

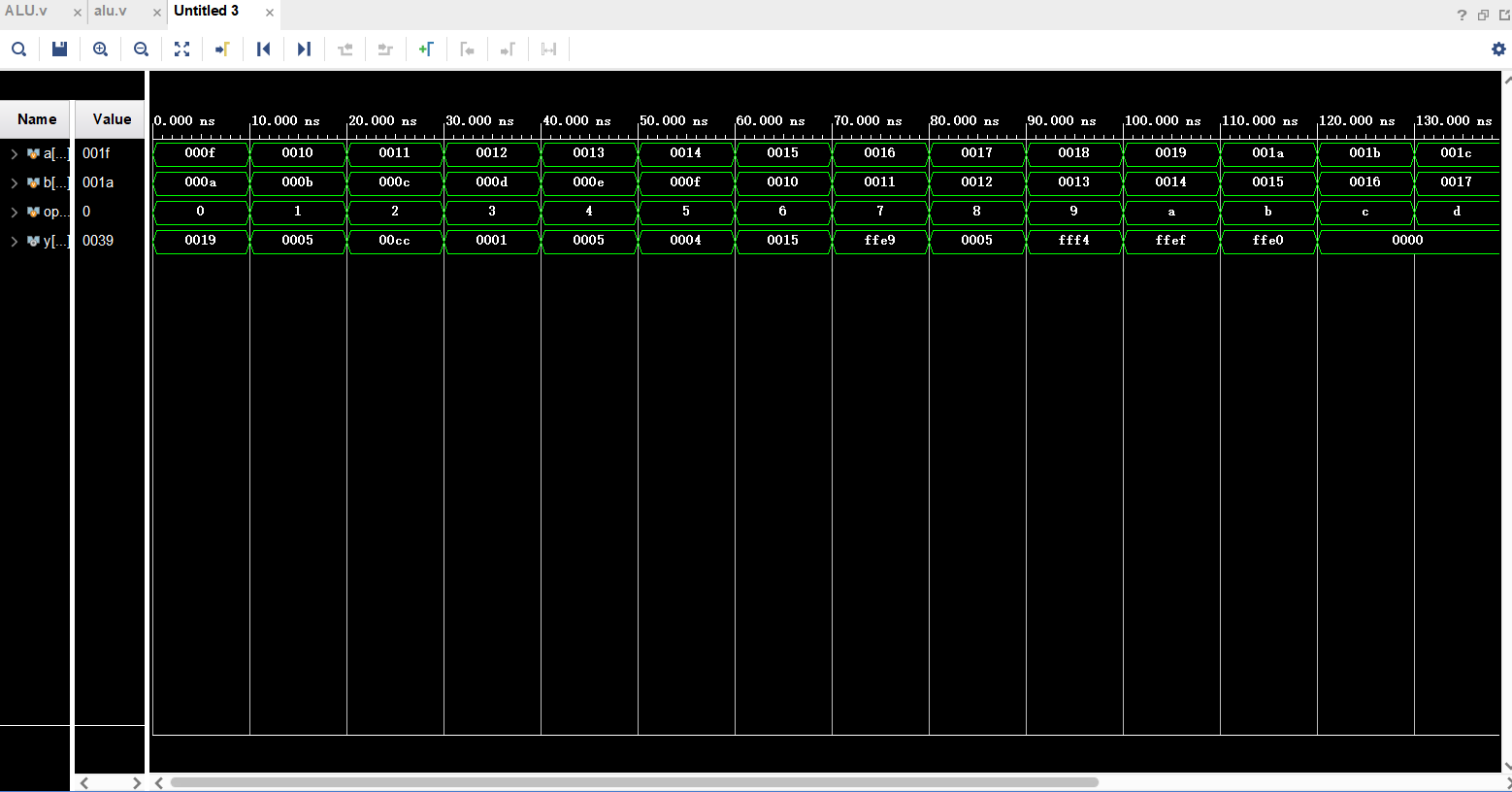
**原因分析：** 根据输入信息我们知道是ALU的代码有问题，应该是end那边的出现了错误

**解决方案：**

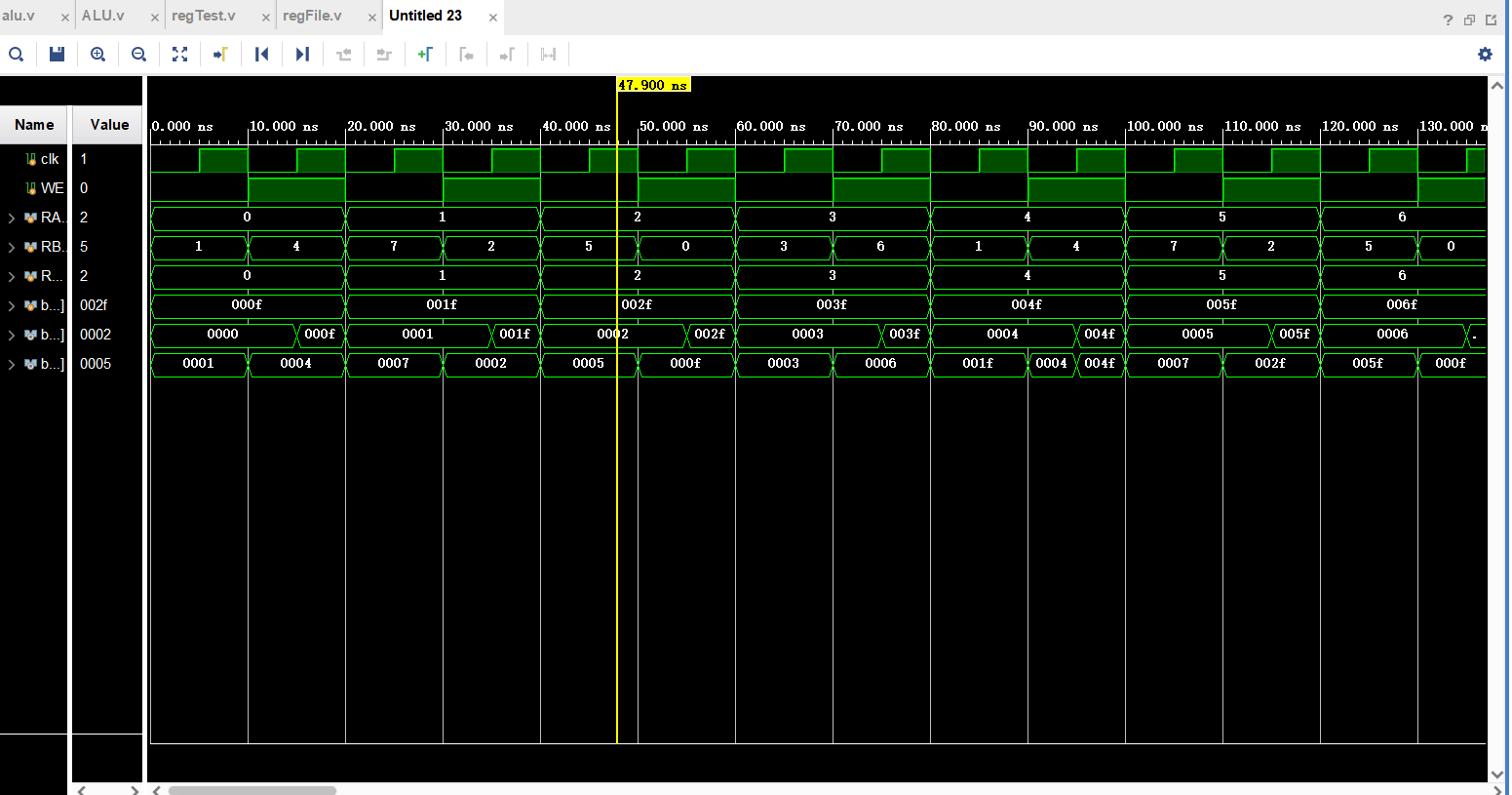
找到相应的代码位置，发现case对应的endcase没有写上去，只要找到相应的位置，写上即可。

## 仿真及分析

Alu:

a,b分别是输入的两个参数，op为要进行的动作的信号，当op=0时，进行a，b的加法运算，当op=1进行减法运算。每10ns会换一个操作以及输入的参数a,b。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,op=1,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。

# **3 单周期 CPU 设计与实现**实验

## 设计要求

其进行的代码如下通过设计并实现支持一条指令的 CPU，理解和掌握 CPU 设计的基本原理和过程。

设计和实现

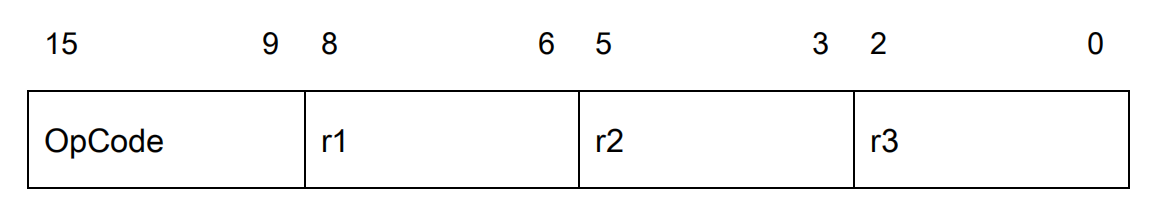
一个支持加法指令的单周期 CPU。要求该加法指令（表示为 add r1，r2，

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；

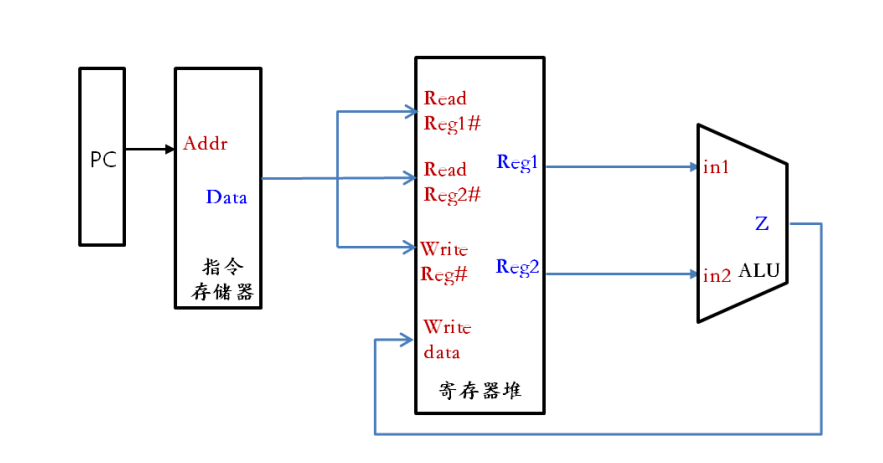
 指令字长 16 位，操作码和地址码字段分配如下所示：



## 方案设计

### 根据功能和格式完成 CPU 的数据通路设计

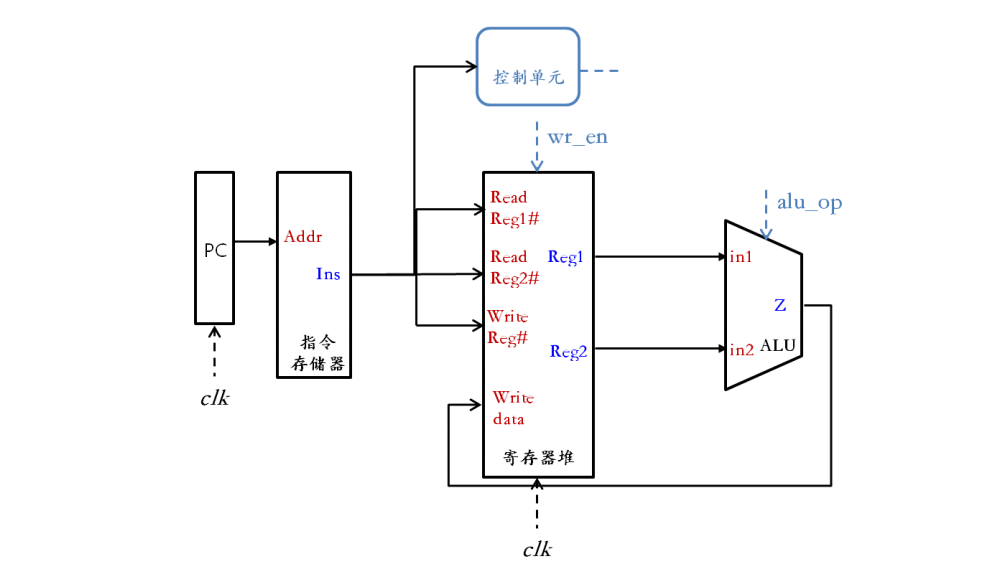
本实验需要设计的 CPU 只需要支持一条加法指令，而该指令的功能是在一个时钟周期内从寄存器组中 r1 和 r2 中取出两个操作数，然后送到 ALU 进行加法运算，最后把计算结果保存到 r1 寄存器中。下图给出了改加法指令的数据通路图。



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。



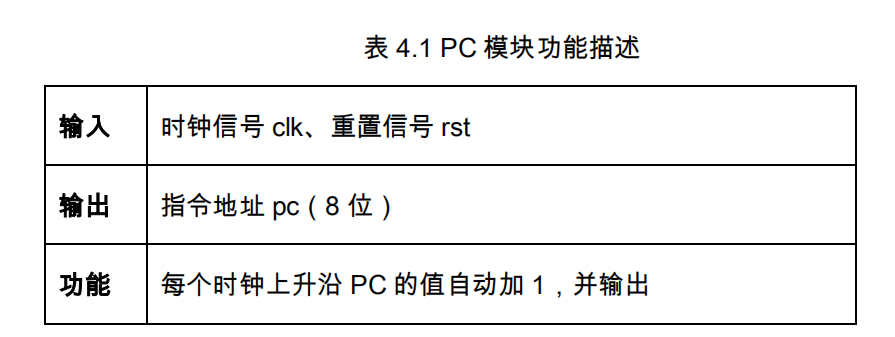
## 实验步骤

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;

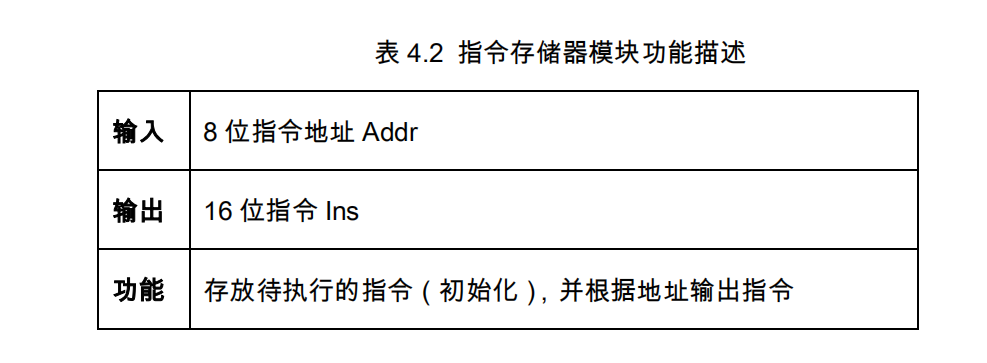
else

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;

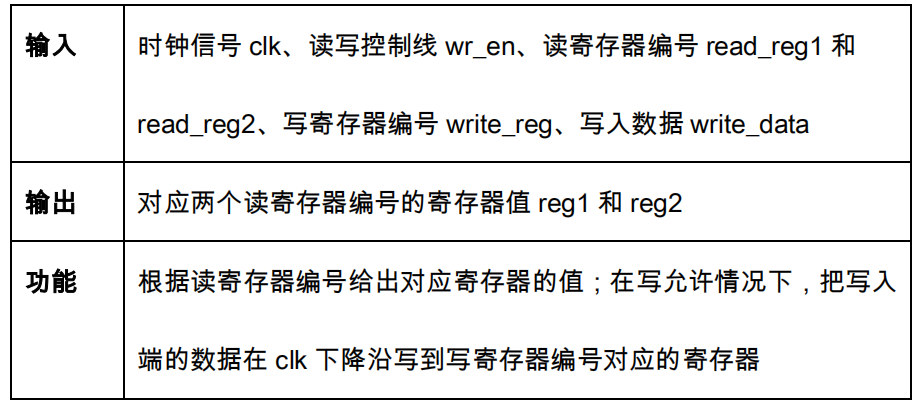
end

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

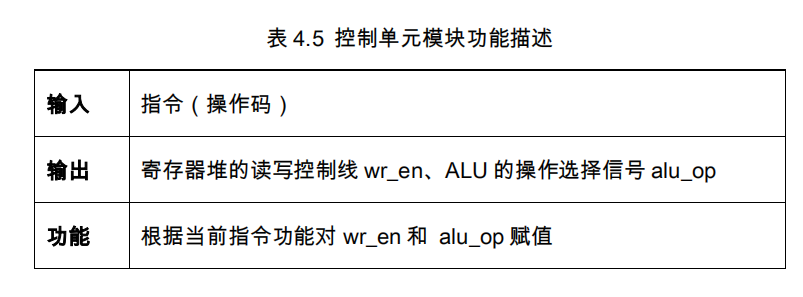
4'd15: y = 0;

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)

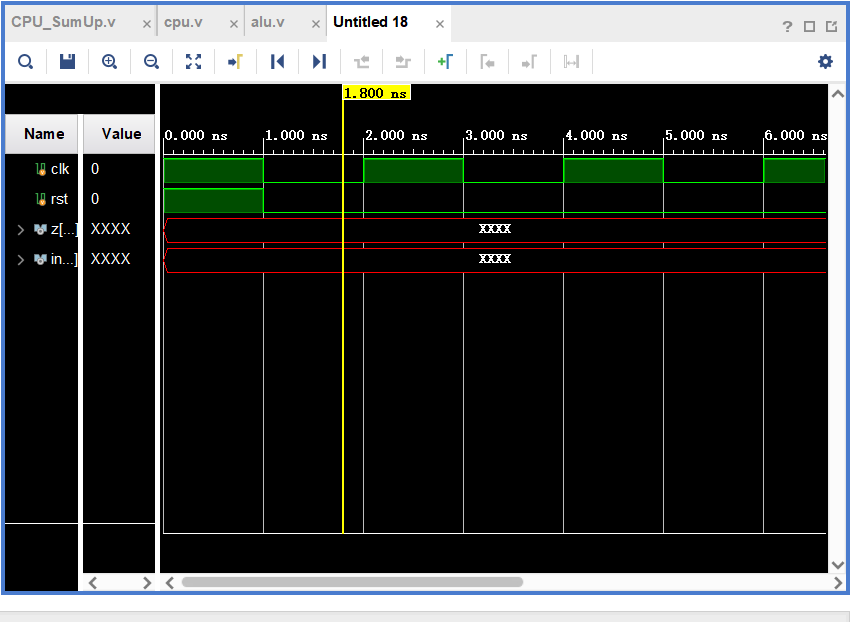
);

endmodule

## 故障与调试

### 故障1

**故障现象：**

图 4.2 1图

**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。

# 5总结与心得

## 实验总结

本次实验主要完成了如下几点工作：

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）。
2. 功能总结（CPU加法计算）（alu中包括了16种计算）（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，（cpu的测试文件实现了cpu的全部功能 ）。

## 实验心得

通过这次实验，我学会了对cpu的各个模块的设计，也对整个过程有了更加深刻的理解。在整个设计的过程中，最让我印象深刻的就是最后的cpu顶层设计，cu设计以及命令存储器的内容。命令存储器的内容关系到最后结果，命令执行的内容。因为本实验是设计的单指令的，所以我把命令存储器关于操作码的一部分全部设置为0，都进行加法。当我们想设置多指令的时候，只需要将存储器的内容改一下即可。但是这个多指令也是在现有的框架下的多指令，没有办法达到一指令，零指令。但是这个简单的实验，让我对cpu内部各个部件的功能和他们之间的联系有了很深刻的理解。本来只是按照书上的内容学习的，现在结合理论知识再参加实验，就有了跟深层次的理解和记忆。还有就是在实验的时候，我忽然理解了verilog为啥要并行运算。在理论学习的时候，我一直以为的是整个过程是顺序进行。但其实在实际中，因为摸一个引脚变化而引起其他变化确实是顺序进行的，但是在没有改变之前，各个部分由于不断通电是一直在工作的，可以将并行运算理解为开机的时候，各个部件马上初始化参数进行工作。

总之，通过这次实验我受益良多，对这门课，对整个过程有了更加深刻的理解，这些理解也反哺了我的理论学习。

# 参考文献

1. DAVID A.PATTERSON(美).计算机组成与设计硬件/软件接口(原书第5版).北京:机械工业出版社.
2. David Money Harris(美).数字设计和计算机体系结构（第二版）. 机械工业出版社
3. 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.
4. 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.
5. 袁春风编著. 计算机组成与系统结构. 北京:清华大学出版社，2011年.
6. 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:** |