《计算机组成原理实验》
实验报告

（单周期 CPU 实现）

学院名称： 数据科学与计算机学院

专业（班级）： 18 计算机类 6 班

学生姓名： 宋渝杰

学号： 18340146

时间： 2019 年 10 月 31 日

实验一：单周期 CPU 设计与实现

一. 实验目的：

1. 掌握 vivado 单周期 CPU 的设计方法
2. 通过实践制作 CPU，加深对 CPU 内部各个模块的原理、数据通路的构造、整体的工作原理的理解

二.实验内容：

使用 verilog 硬件描述语言，设计实现一个支持精简的 MIPS 指令集的单周期 CPU，将冒泡排序的汇编程序代码转化成机器代码之后写入 CPU 的指令寄存器 ROM 中，观察执行之后的结果并验证结果的正确性。

实现的指令包括：

（1）add rd, rs, rt
（2）addi rt, rs, immediate
（3）sub rd, rs, rt
（4）ori rt, rs, immediate
（5）and rd, rs, rt
（6）or rd, rs, rt
（7）sll rd, rt, sa
（8）slt rd, rs, rt
（9）sw rt, immediate(rs)
（10）lw rt, immediate(rs)
（11）beq rs, rt, immediate
（12）bne rs, rt, immediate
（13）j addr
（14）halt

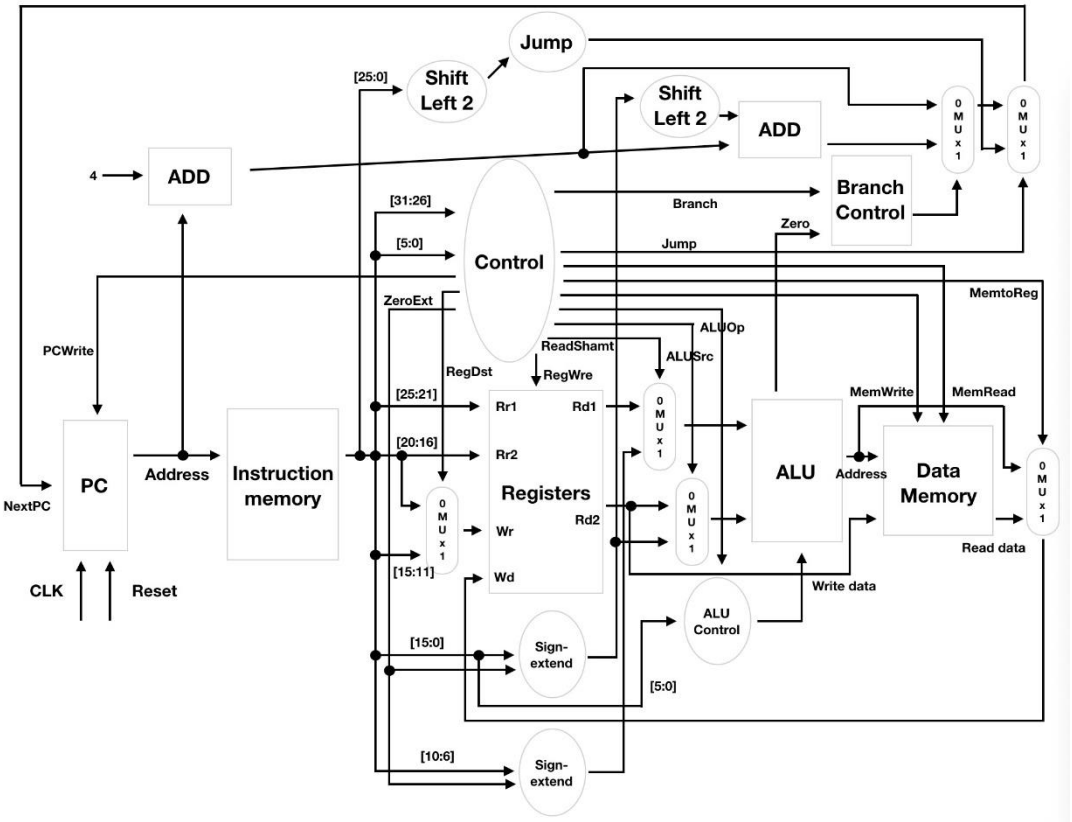
共 14 条指令

三.实验原理

CPU 的执行过程的本质是一个“取指令-操作”的循环。根据读取的指令，确定发出的控制信号，进行逻辑和算术运算，影响存储单元的内容和下一条指令的地址。

实验开始要先设计 mips 指令集、控制信号、ALU 功能表等内容，之后设计相应的 PC、指令存储器、控制单元、寄存器堆、ALU、数据存储器等主要器件，复选器、加法器、逻辑左移器等辅助器件，并通过数据通路整合为一个协同工作的系统。

本实验采用的数据通路图为本人所作，与课本的类似，我也加入了一部分创新：



四.实验器材

电脑一台，Vivado 软件一套

五.实验过程和结果

（一）设计思想

设计这个 CPU 的时候，个人采用了模块化的设计思想。先把 CPU 分为几个小模块，为几个小模块声明接口和编写代码，然后设计一个顶层模块，将各模块通过数据通路连接起来。最后将冒泡程序代码读入指令寄存器，编写测试代码，并观察测试结果。

（二）设计方法

具体步骤如下：

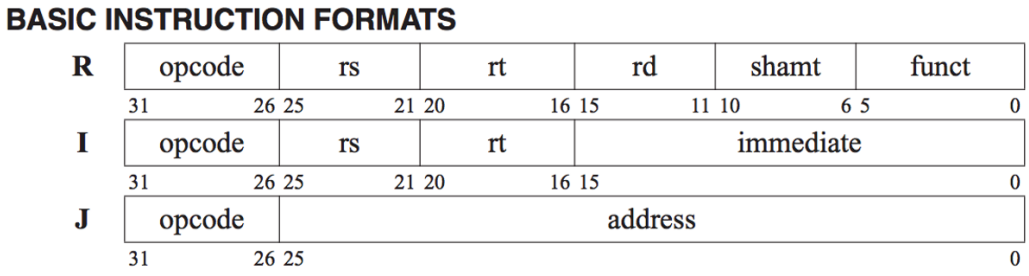
1.设计指令集、控制信号、ALU 功能表

2.将 CPU 分成几个小模块并具体实现
3.设计顶层模块，设计数据通路并连接
4.读入冒泡排序代码，编写测试代码并测试结果

（三）指令集和控制信号表

1.指令集

本 CPU 支持 R 型、I 型、J 型指令，指令格式如下：



指令集如下表：共 14 条指令，其中 6 条 R 型指令，6 条 I 型指令，2 条 J 型指令。表格描述了它们的名称、类型、OPCODE、FUNCT 和功能。除了 halt 停机指令外，其余指令完全依据真实的 MIPS 指令集编写。

指令名	类型	OPCODE	FUNCT	功能
add	R	000000	100000	rd = rs + rt
sub	R	000000	100010	rd = rs - rt
and	R	000000	100100	rd = rs & rt
or	R	000000	100101	rd = rs | rt
sll	R	000000	000000	rd = rt << (zero-extend) shamt
slt	R	000000	101010	rd = rs < rt ? 1 :0 (signed comp)
beq	I	000100		If (rs == rt) pc = pc + 4 + (sign-extend) immediate << 2 else pc = pc + 4
bne	I	000101		If (rs != rt) pc = pc + 4 + (sign-extend) immediate << 2 else pc = pc + 4
addi	I	001000		rt = rs + (sign-extend) immediate
ori	I	001101		rt = rs | (zero-extend) immediate
lw	I	100011		rt = memory [rs + (sign-extend) immediate]
sw	I	101011		memory [rs + (sign-extend) immediate] = rt
j	J	000010		Jump to address
halt	J	111111		PC no more work

2.Contrlo Unit 信号

控制信号共有 12 个，含义如下：

控制信号名	状态“0”	状态“1”
PCWre	PC 不更改	PC 更改
ALUSrc	ALU 没有立即数	ALU 读取立即数
ReadShamt	不读取	ALU 读取 R 型指令[10:6]偏移量
RegWre	不写寄存器	写寄存器
MemRead	不读数据存储器	读数据存储器
MemWrite	不写数据存储器	写数据存储器
MemtoReg	不写数据存储器	数据存储器读出的值写入存储器
ZeroSel	符号扩展	0 扩展
RegDst	写寄存器地址 rt	写寄存器地址 rd
Jump	不跳转	跳转
ALUOp [3:0]	ALU 功能	
Branch [3:0]	两种分支语句：0001：beq 0010：bne	

下表是 14 条指令的各种信号的真值表：

指令名	PCWre	ALUSrc	ReadShamt	RegWre	MemRead	MemWrite
add	1	0	0	1	0	0
sub	1	0	0	1	0	0
and	1	0	0	1	0	0
or	1	0	0	1	0	0
sll	1	0	1	1	0	0
slt	1	0	0	1	0	0
beq	1	0	0	0	0	0
bne	1	0	0	0	0	0
addi	1	1	0	1	0	0
ori	1	1	0	1	0	0
lw	1	1	0	1	1	0
sw	1	1	0	0	0	1
j	1	0	0	0	0	0
halt	0	0	0	0	0	0

指令名	MemtoReg	ZeroSel	RegDst	Jump	ALUOp [3:0]	Branch [3:0]
add	0	0	1	1	0010	0000
sub	0	0	1	1	0010	0000
and	0	0	1	1	0010	0000
or	0	0	1	1	0010	0000
sll	0	1	1	1	0010	0000
slt	0	0	1	1	0010	0000
beq	0	0	0	0	0001	0001

bne	0	0	0	0	0001	0010
addi	0	0	0	1	0100	0000
ori	0	1	0	1	0101	0000
lw	1	0	0	1	0000	0000
sw	0	0	0	0	0000	0000
j	0	0	0	0	0000	0000
halt	0	0	0	0	0000	0000

3.ALU 功能表

本实验设计的 ALU 共支持以下 8 种功能：与、或、加、减、无符号比较、有符号比较、左移、异或。

ALU Control	**功能**
0000	and
0001	or
0010	add
0110	sub
0111	slt_u
1000	slt_sign
1001	shift left
1100	nor

ALU 执行的功能是由 ALUOp 和 FUNCT 决定的

ALUOp	**FUNCT**	**ALU Control**
0000	x	0010
0001	x	0100
0010	100000	0010
0010	100010	0110
0010	100100	0000
0010	100101	0001
0010	101010	1000
0010	000000	1001
0011	x	1000
0100	x	0010
0101	x	0001

（四）设计指令执行过程

R 型指令：

CPU 读取指令寄存器内的指令，将指令中的 rs 和 rt 地址传送给寄存器组，寄存器组送出两个寄存器的值；

Control Unit 根据 FUNCT 值发出 ReadShamt 信号，决定 ALU 的输入 1 是 rs 寄存器的值还是 0 扩展后的位移量 shamt；

Control Unit 发出 ALUSrc 信号为 0，决定 ALU 输入 2 是 rt 寄存器的值；

Control Unit 发出 ALUOp 信号，ALU Control 根据 ALUOp 和 FUNCT 决定 ALU Control 信号值，决定 ALU 进行的运算方式；

ALU 将运算结果送到复用器，Control Unit 发送 MemtoReg 信号为 0，使得 ALU 运算结果送到寄存器组中，Control Unit 发送 RegDst 信号为 1，将 ALU 结果写入 rd 寄存器；

Branch Control 发出信号 0，Control Unit 发出 Jump 信号为 0，使得 PC = PC + 4。

I 型指令：

CPU 读取指令寄存器内的指令，将指令中的 rs 和 rt 地址传送给寄存器组，寄存器组送出两个寄存器的值；

Control Unit 根据 OPCODE 判断指令类型：

若是跳转类指令，Control Unit 发对应的 ALUOp 和 Branch 信号，选择相应的 ALU 功能。Control Unit 发送 ALUSrc 信号为 0，ReadShamt 信号为 0，所以 ALU 的输入来自 rs 和 rt，计算得到结果之后 ALU 发出信号 Zero，Branch Control 根据 Branch 信号和 Zero 信号决定是否跳转。若跳转，发出信号值 1，将跳转的地址 PC + 4 + (sign-extend)immediate 传给 PC。若不跳转，发出信号值 0，PC = PC + 4。

若是 addi 指令，ALUSrc 为 1，指令中的 immediate 进行符号扩展后送入 ALU 输入 2，rs 送入 ALU 输入 1，ALU 进行加法功能，最后结果写入寄存器堆，PC = PC + 4；

若是 ori 指令，Control Unit 发出 ZeroExt 信号为 1，将 immediate 进行 0 扩展，之后与 rs 分别传入给 ALU 两个输入，进行或操作后将结果写入寄存器堆。

若是 lw 指令，CPU 将指令中的 immediate 进行符号扩展，和 rs 共同作为数据存储器地址，Control Unit 发出 MemtoReg 信号为 1，使得数据存储器送出改地址的值，Control Unit 发出 RegDst 信号为 0，将值写入 rt 为地址的寄存器，PC = PC + 4。

若是 sw 指令，CPU 将指令中的 immediate 进行符号扩展，和 rs 共同作为数据存储器地址，Control Unit 发出 MemWrite 信号为 1，将 rt 的值写入指定的数据存储器。

J 型指令:

CPU 读取指令寄存器的指令，Control Unit 发出 Jump 信号为 1，PC 写入 address。如果是 halt 指令，Control Unit 发出 PCWre 指令为 0，PC 停止增加，CPU 停机。

（五）设计模块、顶层文件

1.顶层模块

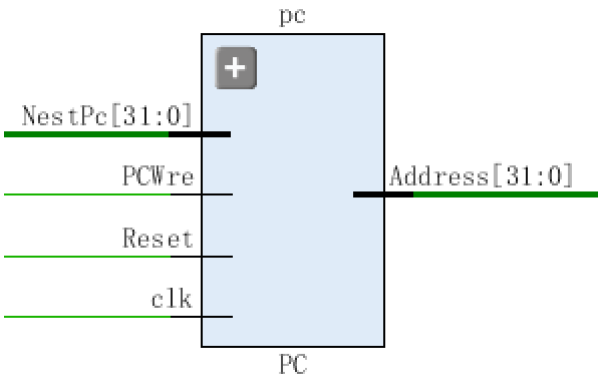
通过在顶层文件实例化各个模块，并通过 wire 类型变量对各个模块进行“连线”。之后的每个模块会在下文一一叙述：

```
wire[3:0] ALUOp;
wire[31:0] outSl2, outsze1, outsze2, outmux4, outmux5, NestPc, outAddi, outs1, outs2, JumpAddr, outAdd, Data1, Data2, DataOut;
wire[15:0] immediate;
wire[3:0] Branch, ac;
wire[4:0] rs, rt, rd, outmux1;
wire[5:0] sa, funct;
wire PCWre, ALUSrc, ReadShamt, RegWre, MemRead, MemWrite, MemtoReg, ZeroSel, RegDst, Jump, zero, outBra;
PC pc(clk, Reset, PCWre, NestPc, Address);
instructionMemory im(Address, opCode, rs, rt, rd, immediate, sa, funct, outSl2);
Mux5 mux1(rd, rt, RegDst, outmux1);
controlUnit CU(opCode, funct, PCWre, ALUSrc, ReadShamt, RegWre, MemRead, MemWrite, MemtoReg, ZeroSel, RegDst, Jump, ALUOp, Branch);
Regfile registerfile(clk, RegWre, rs, rt, outmux1, outmux4, Data1, Data2);
signZeroExtend sze(immediate, ZeroSel, outsze1);
signZeroExtend5 sze2(sa, ZeroSel, outsze2);
Mux mux2(outsze2, Data1, ReadShamt, ALUin1);
Mux mux3(outsze1, Data2, ALUSrc, ALUin2);
ALUControl au(ALUOp, funct, ac);
ALU alu(ALUin1, ALUin2, ac, zero, Result);
dataMemory dm(clk, Result, Data2, MemRead, MemWrite, DataOut);
Mux mux4(DataOut, Result, MemtoReg, outmux4);
BC bc(Branch, zero, outBra);
Addi addi(Address, outAddi);
SL2 s1(outSl2, outs1);
Jump J(outs1, outAddi[31:28], JumpAddr);// wrong?
SL2 s2(outsze1, outs2);
Add add(outAddi, outs2, outAdd);
Mux mux5(outAdd, outAddi,outBra, outmux5);
Mux mux6(JumpAddr, outmux5, Jump, NestPc);
```

2.PC

CLK 是时钟信号，Reset 是初始化置零信号，PCWre 是 PC 启动信号，当时钟信号上升沿到来且 PCWre 为 1 是时更新 Address 为 NextPc。

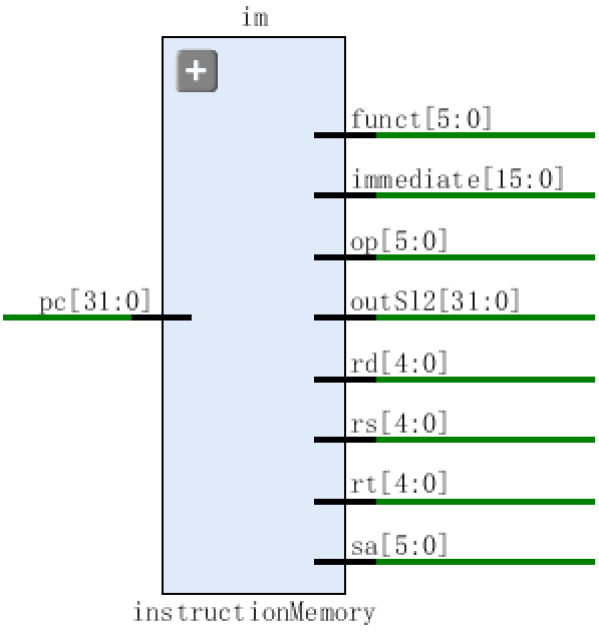
```
always @(posedge clk or negedge Reset)
    begin
        if (Reset == 0) Address = 0;
        else if (PCWre) Address = NestPc;
    end
```



3.指令寄存器 instructionMemory

根据 PC 传出的指令地址，读出指令并将其分为相应的几个部分。

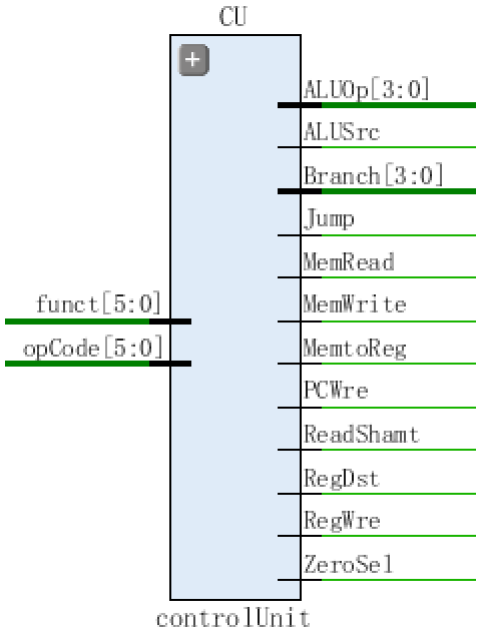
```
initial begin
    $readmemb("C:/Xilinx/test4.txt", mem); // 从文件中读取指令二进制代码赋值给mem
    instruction = 0; // 指令初始化
end

always @(pc) begin
     address = pc[25:2] << 2;
     instruction = (mem[address]<<24) + (mem[address+1]<<16) + (mem[address+2]<<8) + mem[address+3];
end

// output
assign op = instruction[31:26];
assign rs = instruction[25:21];
assign rt = instruction[20:16];
assign rd = instruction[15:11];
assign immediate = instruction[15:0];
assign sa = instruction[10:6];
assign funct = instruction[5:0];
assign outSl2 = instruction[31:0];
```



4.控制信号器 Control Unit

Control Unit 根据上文的 Control Unit 信号真值表，传出相应的信号与信号值，控制其他器件的工作状态。

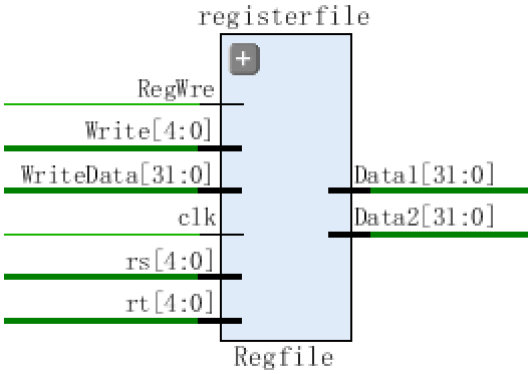
```
// 根据opcode和funct定义控制信号为1或0
assign PCWre = (opCode == 6'b111111) ? 0 : 1;
assign ALUSrc = (opCode == 6'b001000 || opCode == 6'b001101 || opCode == 6'b100011 || opCode == 6'b101011) ? 1 : 0;
assign ReadShamt = (opCode == 6'b000000 && funct == 6'b000000) ? 1 : 0;
assign RegWre = (opCode == 6'b001000 || opCode == 6'b001101 || opCode == 6'b100011 || opCode == 6'b000000) ? 1 : 0;
assign MemRead = (opCode == 6'b100011) ? 1 : 0;
assign MemWrite = (opCode == 6'b101011) ? 1 : 0;
assign MemtoReg = (opCode == 6'b100011) ? 1 : 0;
assign ZeroSel = ((opCode == 6'b000000 && funct == 6'b000000) || opCode == 6'b001101) ? 1 : 0;
assign RegDst = (opCode == 6'b000000) ? 1 : 0;
assign Jump = (opCode == 6'b000010) ? 1 : 0;
assign ALUOp[3] = 0;
assign ALUOp[2] = (opCode == 6'b001000 || opCode == 6'b001101) ? 1 : 0;
assign ALUOp[1] = (opCode == 6'b000000) ? 1 : 0;
assign ALUOp[0] = (opCode == 6'b000100 || opCode == 6'b000101 || opCode == 6'b001101) ? 1 : 0;
assign Branch[3] = 0;
assign Branch[2] = 0;
assign Branch[1] = (opCode == 6'b000101) ? 1 : 0;
assign Branch[0] = (opCode == 6'b000100) ? 1 : 0;
```



5.寄存器组 Regfile

CLK 时钟信号在下降沿时触发尝试写寄存器的操作，在 RegWre 信号为 1 且写入的寄存器号不为 0 时进行写寄存器操作，这是为了保证 0 号寄存器的值为 0 且不会被改变。

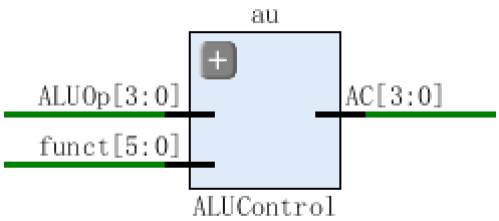
```
assign Data1 = register[rs];//传出寄存器的值
assign Data2 = register[rt];

always @(negedge clk) begin
    if (RegWre && Write) register[Write] = WriteData;
end
```



6.ALU 控制器 ALUControl

ALUControl 根据上文的 ALU 功能表，传入 Control Unit 传出的 ALUOp 信号值和 instructionMemory 传出的 funct 值，传出 ALU Control 信号，控制 ALU 执行具体的运算。

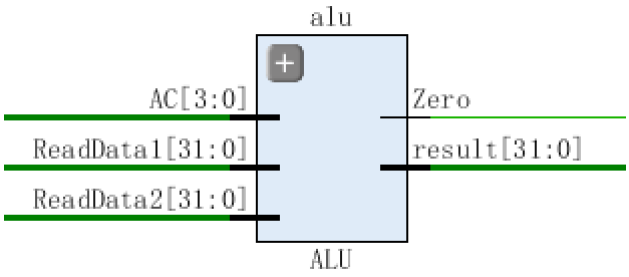
```
always @(ALUOp or funct)
    begin
        if(ALUOp == 4'b0000) AC = 4'b0010;
        if(ALUOp == 4'b0001) AC = 4'b0110;
        if(ALUOp == 4'b0010)
            begin
                if(funct == 6'b100000) AC = 4'b0010;
                if(funct == 6'b100010) AC = 4'b0110;
                if(funct == 6'b100100) AC = 4'b0000;
                if(funct == 6'b100101) AC = 4'b0001;
                if(funct == 6'b101010) AC = 4'b1000;
                if(funct == 6'b000000) AC = 4'b1001;
            end
        if(ALUOp == 4'b0011) AC = 4'b1000;
        if(ALUOp == 4'b0100) AC = 4'b0010;
        if(ALUOp == 4'b0101) AC = 4'b0001;
    end
```



7.ALU

ALU 传入两个操作数，根据 ALU Control 传出的值决定 ALU 的运算功能，传出运算结果，同时传出 Zero 给 Branch Control，用于判断是否执行跳转指令。

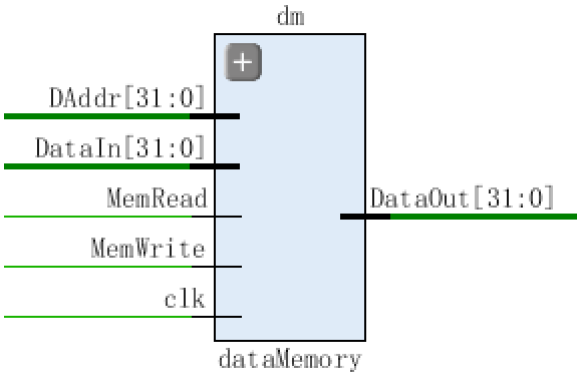
```
assign Zero = ReadData1 == ReadData2 ? 1 : 0;
always @(AC or ReadData1 or ReadData2)
    begin
        case(AC)
            4'b0000: result = ReadData1 & ReadData2;
            4'b0001: result = ReadData1 | ReadData2;
            4'b0010: result = ReadData1 + ReadData2;
            4'b0110: result = ReadData1 - ReadData2;
            4'b0111: result = (ReadData1 < ReadData2) ? 1 : 0;
            4'b1000:
                begin
                    if(ReadData1[31] == 1 && ReadData2[31] == 0)
                        result = 1;
                    else if ((ReadData1 < ReadData2) && (ReadData1[31] == ReadData2[31]))
                        result = 1;
                    else
                        result = 0;
                end
            4'b1001: result = ReadData2 << ReadData1;
            4'b1100: result = ReadData1 ^ ReadData2;
        endcase
    end
```



8.数据存储器 dataMemory

数据存储器读入 DAddr，为将要进行读取或写入操作的数据单元地址，当时钟下降沿到来时，如果 MemWrite 为 1，将读入的数据 DataIn 写入指定的地址；如果 MemRead 为 1，输出指定的地址的值。

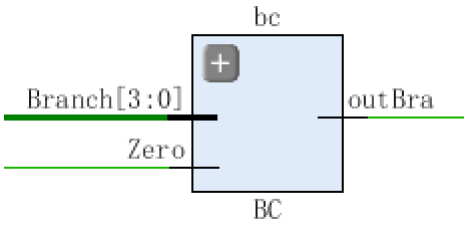
```
reg[7:0] memory[0:511];
reg[31:0] address;
    // read data
    always @(MemRead) begin
        if (MemRead == 1) begin
            // 因为一条指令由4个存储单元存储，所以要乘以4
            address = (DAddr << 2);
            // DataOut是32位的，将4个八位的内存单元合并生成32位
            // 左移24位用于设置前八位，以此类推
            DataOut = (memory[address]<<24)+(memory[address+1]<<16)+(memory[address+2]<<8)+memory[address+3];
        end
    end

    // write data
    integer i;
    initial begin
        for (i = 0; i < 512; i = i+1) memory[i] <= 0;
    end
    always @(negedge clk)
        begin
            if (MemWrite == 1) begin
                address = DAddr << 2;
                memory[address] = DataIn[31:24];
                memory[address+1]= DataIn[23:16];
                memory[address+2]=DataIn[15:8];
                memory[address+3]=DataIn[7:0];
            end
        end
```



9.跳转控制器 Branch Control

Branch Control 输入 Control Unit 传出的 Branch 信号，ALU 发出的 Zero 信号，根据这些信号的值决定是否执行分支语句。

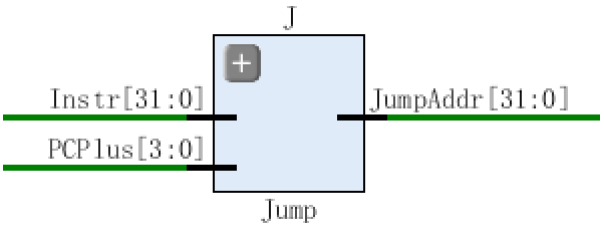
```
always@(Branch or Zero)
    begin
        case (Branch)
            4'b0000: outBra = 1'b0;
            4'b0001: outBra = Zero;
            4'b0010: outBra = ~Zero;
        endcase
    end
```



10.Jump 跳转器

生成 Jump 跳转地址

```
assign JumpAddr[27:0] = Instr[27:0];
assign JumpAddr[31:28] = 4'b0000; //消除左移带来的高位影响
```



11.其他辅助器件

其他的辅助器件包括复用器（5 位和 32 位）、加法器（加 4 和加其他立即数）、逻辑左移器、符号扩展器（0 扩展和符号扩展，5 位和 16 位扩展为 32 位）等。

```
module Mux5(
    input [4:0] a,
    input [4:0] b,
    input c,
    output reg [4:0] d
    );
    always@(a or b or c)
        d = c ? a : b;
endmodule
```

```
module Mux(
    input [31:0] a,
    input [31:0] b,
    input c,
    output reg [31:0] d
    );
    always@(a or b or c)
        d = c ? a : b;
endmodule
```

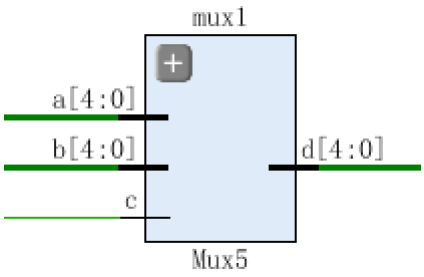
```
module Addi(
    input [31:0] A,
    output reg [31:0] out
    );

    always @(A)
        out = A + 4;
endmodule
```

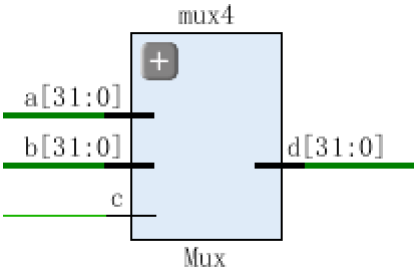
```verilog
module Add(
    input [31:0] A,
    input [31:0] B,
    output reg [31:0] out
    );
    always @(A or B)
        out = A + B;
endmodule
```

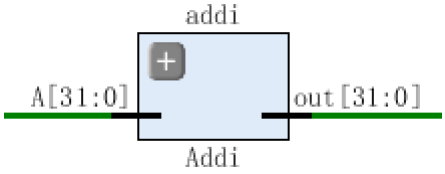
```verilog
module SL2(
    input [31:0] in,
    output reg [31:0] out
    );

    always@(in)
        out = in << 2;
endmodule
```

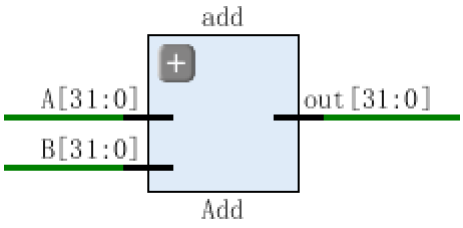
```verilog
module signZeroExtend(
    // 根据数据通路图定义输入和输出
    input [15:0] immediate,
    input ZeroSel,
    output [31:0] out
    );
    assign out[15:0] = immediate;  // 后16位存储立即数
    assign out[31:16] = ZeroSel ? 16'h0000 : (immediate[15] ? 16'hffff : 16'h0000);
```

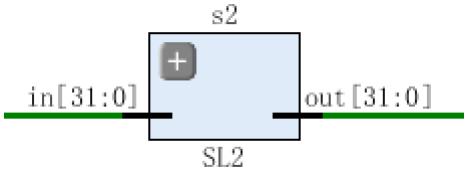
```verilog
module signZeroExtend5(
    input [4:0] immediate,
    input ZeroSel,
    output [31:0] out
    );
    assign out[4:0] = immediate;  // 后5位存储立即数
    assign out[31:5] = ZeroSel ? 27'b000000000000000000000000000 : (immediate[4] ? 27'b111111111111111111111111111 : 27'b000000000000000000000000000);
endmodule
```

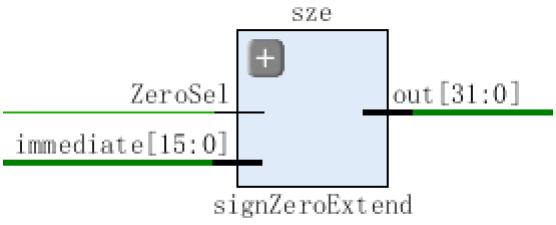




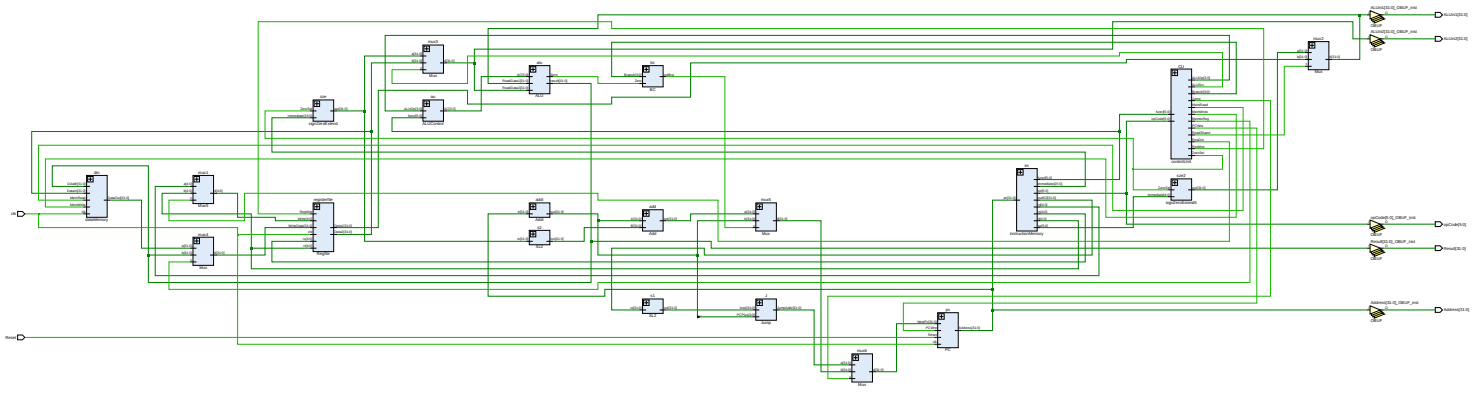








12.最终的 RTL 电路图



（六）进行仿真测试

1.编写测试用的冒泡排序代码

Mips 源码：（在代码中将 10 个数字按顺序 1、10、2、9、3、8、4、7、5、6 存入数据存储器，然后通过冒泡排序算法将十个数从大到小进行排序）

```
    add  $s0, $zero, $zero     # i=0
exloop:                        # 外循环
    add $s1, $zero, $zero      # j=0
inloop:                        # 内循环
    add $t1, $s1, $zero
    sll $t1, $t1, 2
    add $t1, $t1, $t0
    lw  $s7, 0($t1)            # 相当于 array[j]
    addi $t2, $s1, 1
    sll  $t2, $t2, 2
    add  $t2, $t2, $t0
    lw  $s6, 0($t2)            # 相当于 array[j+1]
    slt $t3, $s7, $s6          # 判断是否 array[j] < array[j+1] 若 array[j]<arr
    beq $t3, $zero exit
    sw $s7, 0($t2)
    sw $s6, 0($t1)             # 交换阶段
exit:
    addi $s1, $s1, 1           # j++
    addi $t4, $zero, 9
    sub  $t5, $t4, $s0         # 9-i
    slt  $t6, $s1, $t5         # 判断 j 是否小于 9-i，如果是的话，跳转 inloop
    bne $t6, $zero inloop
    addi $s0, $s0, 1           # i++
    slt $t6, $s0, $t4           # 判断是否 i 是否小于 9，是的话跳转 exloop:
    bne $t6, $zero exloop
```

上图为关键的冒泡排序交换代码，不包含过长的数据读入代码

Code	Basic			Sou
0x00008020	add $16,$0,$0	64:	add $s0, $zero, $zero	# i=0
0x00008820	add $17,$0,$0	66:	add $s1, $zero, $zero	# j=0
0x02204820	add $9,$17,$0	68:	add $t1, $s1, $zero	
0x00094880	sll $9,$9,0x00000002	69:	sll $t1, $t1, 2	
0x01284820	add $9,$9,$8	70:	add $t1, $t1, $t0	
0x8d370000	lw $23,0x00000000($9)	71:	lw $s7, 0($t1)	# 相当于 array[j]
0x222a0001	addi $10,$17,0x00000001	72:	addi $t2, $s1, 1	
0x000a5080	sll $10,$10,0x00000002	73:	sll $t2, $t2, 2	
0x01485020	add $10,$10,$8	74:	add $t2, $t2, $t0	
0x8d560000	lw $22,0x00000000($10)	75:	lw $s6, 0($t2)	# 相当于 array[j+1]
0x02f6582a	slt $11,$23,$22	76:	slt $t3, $s7, $s6	# 判断是否 array[j]
0x11600002	beq $11,$0,0x00000002	77:	beq $t3, $zero exit	
0xad570000	sw $23,0x00000000($10)	78:	sw $s7, 0($t2)	
0xad360000	sw $22,0x00000000($9)	79:	sw $s6, 0($t1)	# 交换阶段
0x22310001	addi $17,$17,0x00000001	81:	addi $s1, $s1, 1	# j++
0x200c0009	addi $12,$0,0x00000009	82:	addi $t4, $zero, 9	
0x01906822	sub $13,$12,$16	83:	sub $t5, $t4, $s0	# 9-i
0x022d702a	slt $14,$17,$13	84:	slt $t6, $s1, $t5	# 判断 j 是否小于 9-
0x15c0ffef	bne $14,$0,0xffffffef	85:	bne $t6, $zero inloop	
0x22100001	addi $16,$16,0x00000001	86:	addi $s0, $s0, 1	# i++
0x020c702a	slt $14,$16,$12	87:	slt $t6, $s0, $t4	# 判断是否 i 是否小
0x15c0ffeb	bne $14,$0,0xffffffeb	88:	bne $t6, $zero exloop	

16 进制代码：（将上述代码输入到 MARS 软件并进行编译后，可得到 16 进制代码；最后加上了一条 Jump 指令代码，用于测试 J 型指令，和一条自己编写的 halt 停机指令代码）

2 进制代码：（用以下 python3 代码即可实现将 16 进制代码.txt 文件转化成 2 进制代码并输出，之后将得到的 2 进制代码每 8 位分成一行即可得到最终测试的机器代码）

```
with open ( ‘ 文件绝对路径 ’ ) as src:
    for row in src.readlines():
        print(“{0:032b}".format(int(row,16)))
```

将上述得到的 2 进制代码保存为.txt 文件，在指令寄存器 instructionMemory 的 initial 语句中输入$readmemb("文件绝对路径", mem); 即可将 2 进制代码读入 CPU 的 ROM 内。

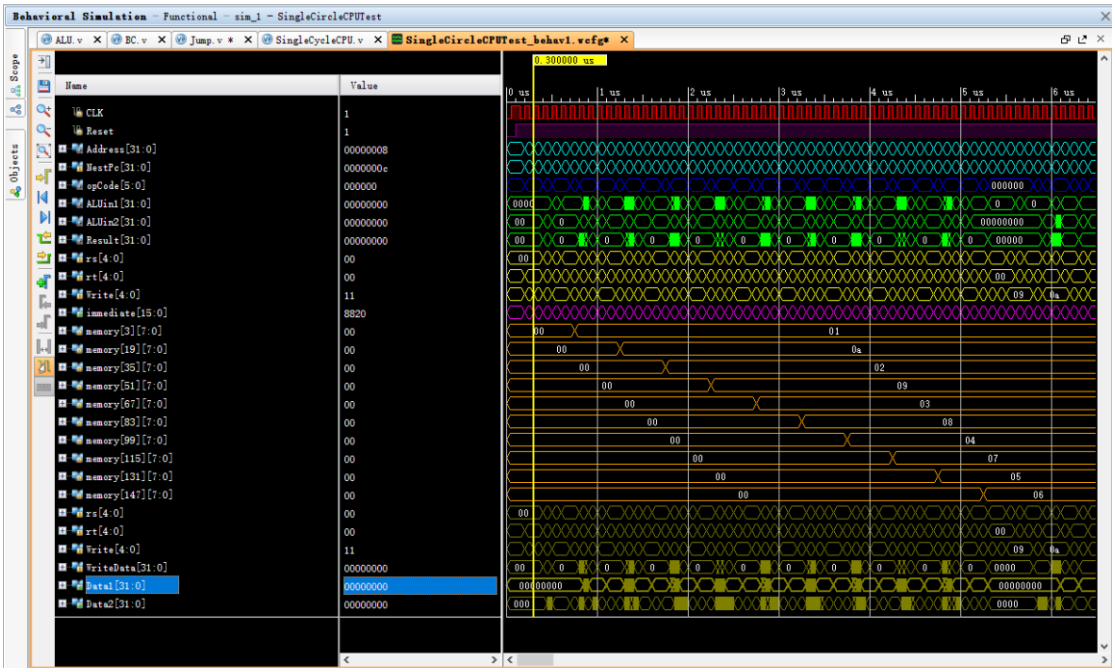
2.编写仿真代码

测试代码中实例化一个 CPU，设置好 CLK 和 Reset 信号。

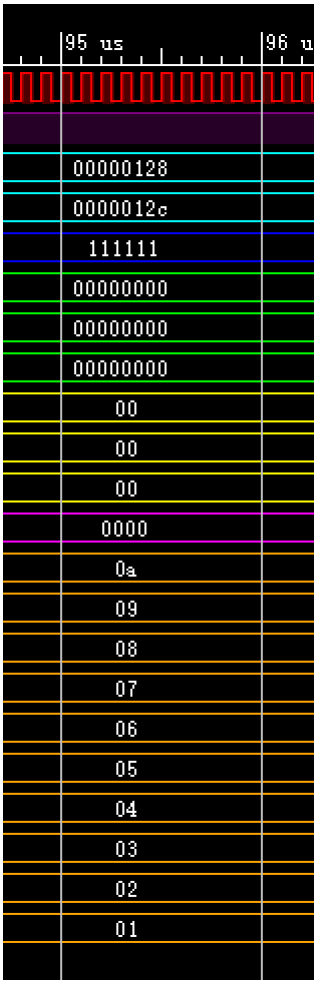
```
22 module SingleCircleCPUTest;
23     // Inputs
24     reg CLK;
25     reg Reset;
26     // Outputs
27     wire[5:0] opCode;
28     wire[31:0] ALUin1, ALUin2, Address, Result;
29     // Instantiate the Unit Under Test (UUT)
30     SingleCycleCPU uut (
31         .clk(CLK),
32         .Reset(Reset),
33         .opCode(opCode),
34         .ALUin1(ALUin1),
35         .ALUin2(ALUin2),
36         .Address(Address),
37         .Result(Result)
38     );
39     initial begin
40         // Initialize Inputs
41         CLK = 0;
42         Reset = 0;
43         #50; // 刚开始设置pc为0
44             CLK = 1;
45         #50;
46             Reset = 1;
47         forever #50 begin // 产生时钟信号
48             CLK = !CLK;
49         end
50     end
51 endmodule
```

3.仿真测试及结果

冒泡排序的测试：



上图中橙色部分为 10 个数据的数据存储器值，可以看出，程序在一开始，成功将十个数据按顺序 1、10、2、9、3、8、4、7、5、6 读入进数据存储器

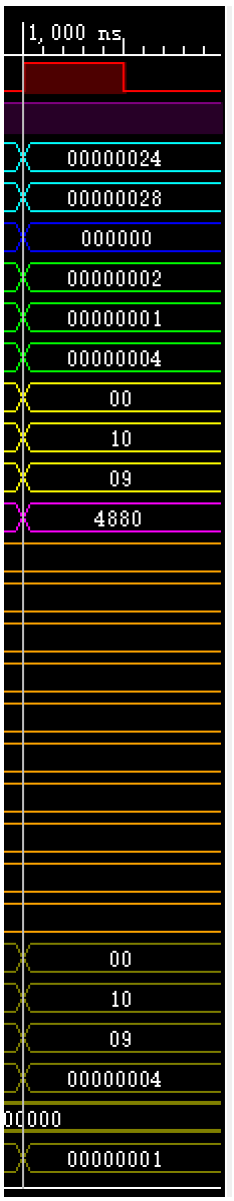


左图中程序完成，CPU 停机之后，可以看出 10 个数据已经按从小到大的顺序进行排列

三种类型指令的操作过程测试和说明：

下文中红色（第一行）为 CLK 信号，紫色（第二行）为 Reset 信号，为 1，淡蓝色（第三行、第四行）为现在的 PC 地址、下一个 PC 地址，深蓝色（第五行）为 opcode 信号，绿色（第六行、第七行、第八行）为 ALU 输入 1、2、输出，黄色（第九行、第十行、第十一行）为 rs、rt、rd（在相应的指令类型中才有实际意义），粉色（第十二行）为 immediate（在相应的指令类型中才有实际意义），橙色（十行）为十个要进行冒泡排序的数据，橄榄金色（最后六行）分别为读寄存器号 1、2，写寄存器号，写入数据，寄存器读出数据 1、2。

R 型指令：（sll $t1, $s0, 2）



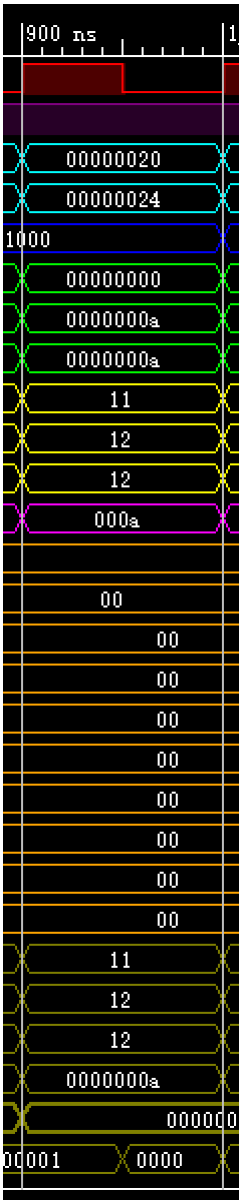
执行过程：

当 PC 地址为 00000024（十六进制）时，指令寄存器读出指令 sll $t1, $s0, 2，为 R 型指令，将指令中的 rs 和 rt 地址传送给寄存器组，寄存器组送出两个寄存器的值，但是 CU 发出信号 ReadShamt 信号为 1，ALU 输入 1 为指令中的 Shamt 部分，值为 2。
CU 发出 ALUSrc 信号为 0，ALU 输入 2 是 $s0 寄存器的值，ALU 控制器根据 opcode 和 funct，决定是左移操作，传给 ALU 执行。ALU 将运算结果送到复用器，CU 发送 MemtoReg 信号为 0，使得 ALU 运算结果送到寄存器组中，Control Unit 发送 RegDst 信号为 1，将 ALU 结果写入 $t1 寄存器；

结果分析：

绿色部分 ALU 输入为 2 和 1（$s0 的值），执行 1<<2 = 4，输出结果为 4，正确，最后将写入 $t1 寄存器。

I 型指令：（addi $s2, $s1, 10）



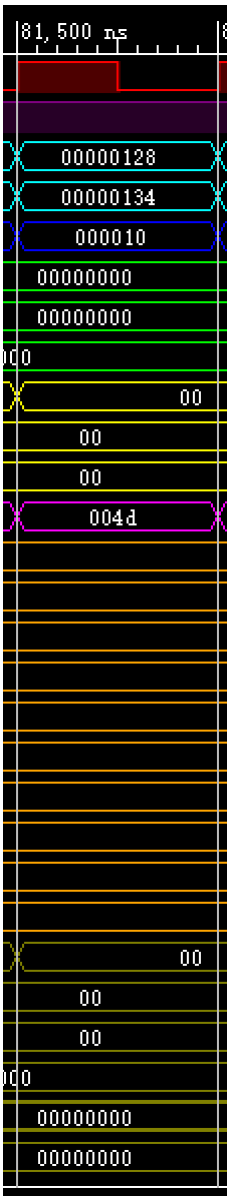
执行过程：

CPU 读取指令寄存器内的指令，将指令中的 rs 和 rt 地址传送给寄存器组，寄存器组送出两个寄存器 $s2, $s1 的值；Control Unit 根据 opcode 和 funct 判断指令类型是 addi 指令，ALUSrc 为 1，指令中的 immediate 进行符号扩展后为 a（十六进制），送入 ALU 输入 2，舍弃寄存器 $s2 送出的值。寄存器 $s1 的值 0 送入 ALU 输入 1，ALU 进行加法功能，得到结果 10，最后结果写入寄存器堆中的寄存器 $s2，PC = PC + 4。

结果分析：

绿色部分 ALU 输入为 0（$s0 的值）和 a（immediate），执行 0 + a = a，输出结果为 a，正确，最后将写入 $s2 寄存器。

J 型指令：（Jump 4dh）



执行过程：

CPU 读取指令寄存器的指令，Control Unit 根据 opcode 和 funct 判断指令类型是 Jump 指令，发出 Jump 信号为 1，将地址 4d 左移两位后变成 134 写入 PC。

结果分析：

淡蓝色当前地址为 128（十六进制），下一个地址为 134，证明 Jump 指令已经正确执行。

六.实验心得

本次单周期 CPU 设计实验中，花费了我大量的时间，几乎抽去一周里大部分的空余时间去研究 CPU 的构造和实现方式，但总的来说还是完成了这项工作。

遇到的困难：

1.对 CPU 构造图的难以理解：CPU 构造图中包含着许多的小模块，有许多的信号控制，有许多的线路连接，一段时间内个人近乎难以完全理解整个 CPU 的构造；

2.对 vivado 以及 verilog 语言的不熟悉：这次的计算机组成原理是我们第一次接触 vivado 和 verilog 语言，对这个软件的使用和 verilog 语言的理解也消耗了大部分的时间；

3.参考资料的缺乏和混乱：在实现这个 CPU 的时候，我参考了许多的各个来源的 CPU，

有往届学长的，也有网络上别人写的 mipsCPU，然而，往届的学长实验报告代码不完整，网络上的 CPU 构造图也与书本上的有不少出入，以及参考的多个 CPU 的 MIPS 指令集也有和真实的 MIPS 指令集有较大区别的情况发生。因此在吸收别人的 CPU 有用的知识并进行整合，形成自己的 CPU，也花费了不少的时间和精力。

解决方式：

1.对于 CPU 构造图的理解，我翻阅了多本教材资料，也上网站看了一些慕课，甚至是先照着别人的 CPU 设计构造打一遍代码，慢慢理解构造图，再设计自己的 CPU；

2.对于 vivado 的不断报错和 verilog 代码的写法，个人也在网上浏览了许多别人设计的 CPU 的代码设计，百度一些常见的 verilog 的语法和功能，慢慢理解和修改，一个个分析报错的原因并处理；

3.对于参考资料的混乱，个人分析了多个 CPU 的共同之处并继承到自己的 CPU 中，而对于不同之处则取长去短，比如说有的 CPU 通过$readmemb("文件绝对路径", mem);来写入测试代码，有的则直接一条条代码输入到指令寄存器中，后者比较麻烦也容易出错，所以本人的 CPU 采用了$readmemb("文件绝对路径", mem);的方式读入测试代码；对于别人的 CPU 不采用标准的 MIPS 指令集，本人也将它们的指令集和处理方式调整为标准的 MIPS 指令集后再整合进自己的 CPU 内。

实验收获：

1.完成了单周期 CPU，支持 14 条 MIPS 指令；

2.对 vivado 的仿真等操作更加熟练，了解学习了不少 verilog 代码；

3.对单周期 CPU 的构造图和功能原理有了更加深刻的理解。