《计算机组成原理实验》
实验报告

（多周期 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 12 月 18 日

实验一：多周期 CPU 设计与实现

一. 实验目的：

1. 掌握 vivado 多周期 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）mul rd, rs, rt
（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）syscall（停机）

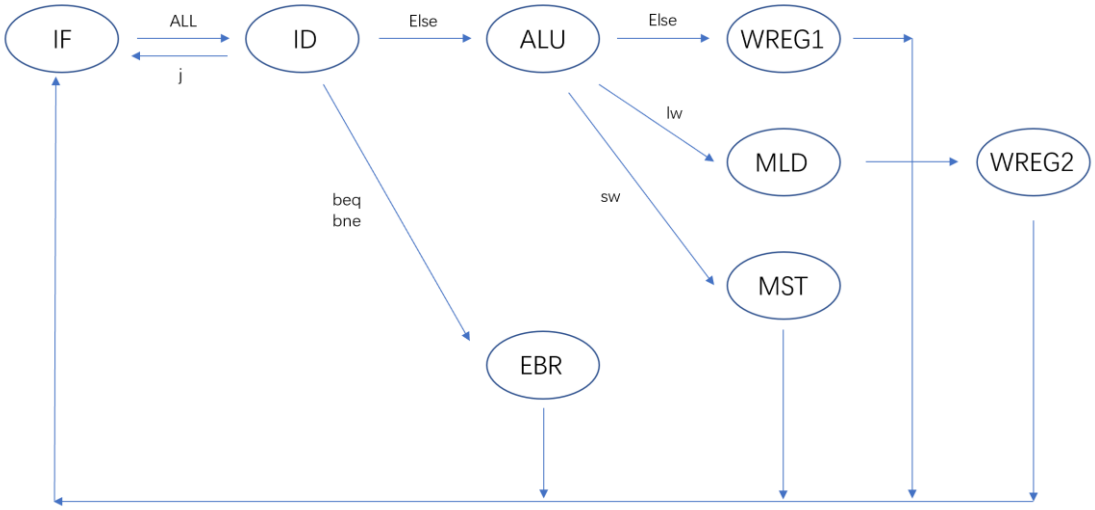
共 14 条指令

三.实验原理

多周期 CPU 本质执行的是一个“取指令（IF）、译码（ID）、执行（EXE）、存储器访问（MEM）、数据写回（WB）”的循环，根据读取的指令和当前状态，发出控制信号，进行逻辑和算术运算，影响储存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、状态转换图等内容，之后设计相应的 PC、指令/数据存储器、控制单元、寄存器堆、ALU 等主要器件，并通过数据通路整合为一个协同工作的系统。

本 CPU 的状态转化图如下图所示：共分为 5 个阶段，IF 为阶段 1，ID 为阶段二，ALU、EBR 为阶段 3，WREG1、MLD、MST 为阶段 4，WREG2 为阶段 5，同一个阶段的不同部分执行的大致内容相同，只有具体细节有所差异：



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、状态转换图

2.将 CPU 分成几个小模块并具体实现

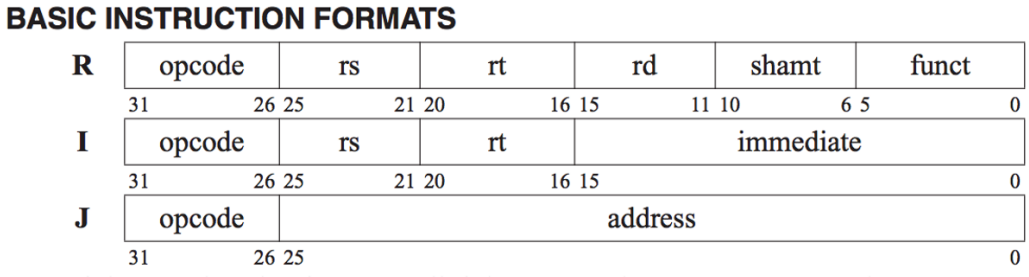
3.设计顶层模块，设计数据通路并连接

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 7 条 R 型指令，5 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 syscall 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
mul	R	011100	000010	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
lw	I	100011		rt = memory [rs + (sign-extend) immediate]
sw	I	101011		memory [rs + (sign-extend) immediate] = rt
j	J	000010		Jump to address
syscall	J	000000	001100	PC no more work

2.Contrlo Unit 信号

控制信号共有 11 个，含义如下：

控制信号名	状态“0”	状态“1”
PCWre	PC 不更改	PC 更改
IorD	访问地址为来自 PC 的访问指令的地址	存储器的访问地址是来自 ALUout 的访问数据的地址
IRWrite	不允许写入 IR	允许在时钟下降沿时写入 IR 寄存器
MemWrite	不允许写入存储器	允许在时钟下降沿时写入存储器
RegWrite	不允许写入寄存器堆	允许在时钟下降沿时写入寄存器堆
Jal	写入寄存器堆的数据由 MemToReg 控制	选择写入寄存器堆的数据，此时为 PC 的值
MemToReg	写入寄存器堆的数据为来自 ALU 的 ALUout	选择写入寄存器堆的数据，此时为存储器 dataout 的值

续表：

控制信号名	功能	状态“11”	状态“10”	状态“01”	状态“00”
PCsrc	选择 PC 的变化方式	j 的跳转	无	beq、bne 的跳转	非跳转状态，pc=pc+4
RegDst	选择写入的寄存器	rd	31 号寄存器	rt	rd
ALUsrc1	选择 ALU 的操作数一	无	扩展至 32 位的 shamt	PC 的值	reg[rs]
ALUsrc2	选择 ALU 的操作数二	imme*4	imme	4	reg[rt]

3.状态行为表

该 CPU 的 5 个阶段（共 8 种状态）的具体行为表如下：

状态名称	状态译码	操作	涉及指令
IF	000	IR <= Memory[PC] PC <= PC + 4	all
ID	001	pc <= {pc[31:28],instruct[25:0],2b'00}	j
		A <= reg[rs] B <= reg[rt] ALUresult = PC + (sign-extend)imme ALUout <= ALUresult	else

ALU	010	ALUresult = B << shamt zero = (ALUresult==0 ?)1 : 0 ALUout <= ALUresult	sll
		ALUresult = A ALUop B zero = (ALUresult==0 ?)1 : 0 ALUout <= ALUresult	其余 R 型指令
		ALUresult = A ALUop (sign-extend)imme zero = (ALUresult==0 ?)1 : 0 ALUout <= ALUresult	I 型指令
WREG1	011	reg[rt] <= ALUout	I 型指令
		reg[rd] <= ALUout	R 型指令
WREG2	100	reg[rt] <= dataout	lw
MLD	101	dataout = mem[address]	lw
MST	110	mem[address] <= datain	sw
EBR	111	if(A==B) PC <= ALUout	beq
		if(A!=B) PC <= ALUout	bne

各种状态和 Control Unit 信号的关系表如下：

状态	指令	PCWrite	IorD	IRwrite	MemWrite	PCsrc	ALUop	ALUsrc1	ALUsrc2	RegWrite	RegDst	MemToReg	Jal
IF	all	1			1		111	1	1				
ID	j	1				11							
	else	1					111	1	1				
ALU	sll						下文	10	0				
	addi、lw、sw							0	10				
	else							0	0				
WREG1	addi									1	1		
	else									1	0		
WREG2	lw									1	1	1	
MLD			1										
MST			1		1								
EBR	beq	1				1							
	bne	1				1	1						

4.ALU 功能信号表

本实验设计的 ALU 共支持以下 8 种功能：

ALUop	运算	相关指令
0000	ALUresult = B << A	sll
0001	ALUresult = A - B	sub、bne、beq
0010	ALUresult = A & B	and
0011	ALUresult = A | B	or
0100	ALUresult = ~A	not
0110	ALUresult = (A < B)? 1 : 0	slt
0111	ALUresult = A + B	add、addi、sw、lw
1001	ALUresult = A * B	mul

（四）设计指令执行过程

第一阶段：

IF：PC = PC + 4，将地址传给指令/数据寄存器。

第二阶段：

ID：指令/数据寄存器收到地址后，读取相应位置的指令，分析指令的类型：如果是 J 型指令，将指令的地址<<2 后传给 PC，回到 IF 阶段；如果是 beq、bne 类型的指令，将 rs、rt 位置的数据传给 ALU，ALU 将执行减法、判断符号操作；如果是其他类型的指令，将 rs、rt 或 immediate 位置的数据传给 ALU，Control Unit 也根据 opcode 和 funct 判断 ALUOP 的值，并传给 ALU 做相应算术/逻辑操作。

第三阶段：

ALU：首先由 ALU 前的复用器根据信号判断传入 rs、rt 还是 immediate 的值给 ALU，然后 ALU 根据上一阶段的 ALUOP 值做相应算术/逻辑操作，并输出结果。

EBR：如果是 beq、bne 指令，ALU 也执行减法、判断符号操作，之后判断结果是否等于 0，然后判断是否进行跳转，跳转的话修改 PC 值，否则 PC 不变，最后回到 IF。

第四阶段：

MST：如果是 sw 指令，ALU 运算结果成为地址，将 rt 位置的值存入数据存储器相应地址处，之后回到 IF。

MLD：如果是 lw 指令，ALU 运算结果成为地址，数据存储器输出相应位置的数据。

WREG1：其他指令时，将 ALU 运算结果存入寄存器堆相应的寄存器，回到 IF。

第五阶段：

WREG2：lw 指令，将数据存储器输出的数据存入寄存器堆相应的寄存器，回到 IF。

（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”，同时在此处通过一些代码实现复用器的功能，节省复用器模块的设计。

之后的每个模块会在下文一一叙述：

“连线”关键代码：

```
PC pc(Address_next,PCWrite,clk,reset,Address);
ALU alu(A,B,ALUop,zero,ALUresult);
SizeExtend se(shfamt,imme15_0,imme25_0,imme10_6e,imme15_0e,imme15_0e2,imme25_0e);
RegHeap rh(clk,RegWrite,rs,rt,writeReg,writeData,Data1,Data2);
ControlUnit cu(clk,reset,zero,opcode,funct,halt,PCWrite,IorD,IRWrite,MemWrite,PCsrc,ALUop,ALUsrc1,ALUsrc2,RegWrite,RegDst,MemToReg,Jal);
Memory memory(MemWrite,add_in,clk,DataB,dataout);
```

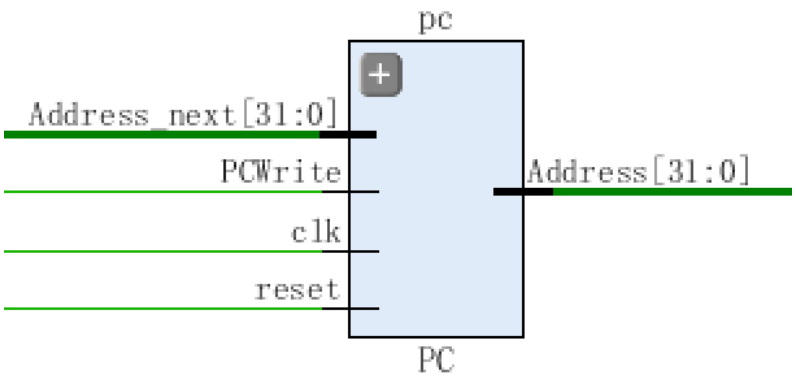
复用器实现关键代码：

```
wire [31:0]A;
wire [31:0]B;
assign A = (ALUsrc1==2'b10) ? imme10_6e : ((ALUsrc1==2'b01) ? Address : DataA);
assign B = (ALUsrc2==2'b11) ? imme15_0e2 : ((ALUsrc2==2'b10) ? imme15_0e : ((ALUsrc1==2'b01) ? 4 : DataB));
```

2.PC

CLK 是时钟信号，Reset 是初始化置零信号，PCWrite 是 PC 启动信号，当时钟信号下降沿到来且 PCWitre 为 1 时，更新 Address 为 Address_next。

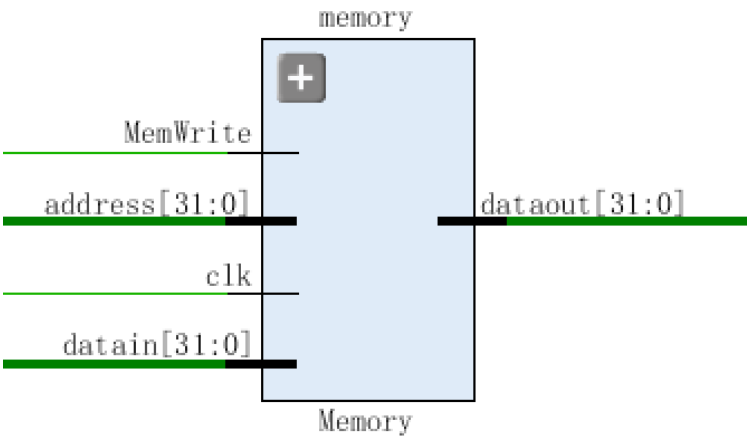
```
initial begin
    Address<=932;
end
 always @(negedge clk or negedge reset)
     begin
          if (reset == 0)    Address <= 932;
          else if(PCWrite)Address <= Address_next;
      end
```



3.指令/数据寄存器 Memory

同时存储指令和数据，932-1040 号内存用于存储指令，事先读取冒泡排序代码存入相应内存位置。0-931 号内存用于存储数据，事先将冒泡排序的 10 个数字存入相应位置。

```
initial begin
      dataout <= 0;
      $readmemh("C:/Users/Song/Desktop/CPU_MultiCycle_V1/instructions", mem,932);
      for (i = 0; i < 932; i = i+1) begin
          mem[i] <= 0;
      end
      mem[3]<=1;
      mem[7]<=10;
      mem[11]<=2;
      mem[15]<=9;
      mem[19]<=3;
      mem[23]<=8;
      mem[27]<=4;
      mem[31]<=7;
      mem[35]<=5;
      mem[39]<=6;
end
always@(negedge clk)begin
    if(MemWrite)begin
        mem[address] <= datain[31:24];
        mem[address+1] <= datain[23:16];
        mem[address+2] <= datain[15:8];
        mem[address+3] <= datain[7:0];
    end
end
always@(*)begin
    dataout = {mem[address],mem[address+1],mem[address+2],mem[address+3]};
end
```



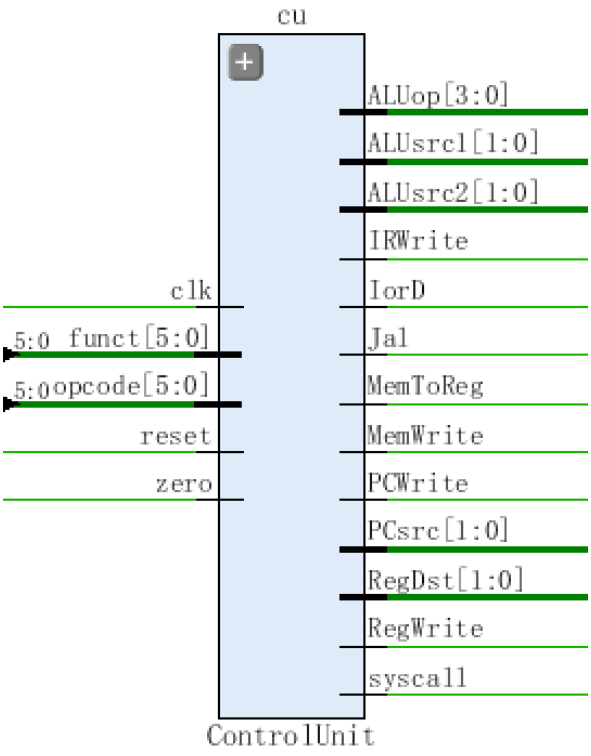
4.控制信号器 Control Unit

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。

```
always@(*)begin
    case(state)
        IF:begin
            next_state = ID;
        end
        ID:begin
            next_state = (j | jal | jr) ? IF : (beq | bne) ? EBR : ALU;
        end
        ALU:begin
            if(lw) next_state = MLD;
            else if(sw) next_state = MST;
            else next_state = WREG1;
        end
        WREG1:begin
            next_state = IF;
        end
        WREG2:begin
            next_state = IF;
        end
        MLD:begin
            next_state = WREG2;
        end
        MST:begin
            next_state = IF;
        end
        EBR:begin
            next_state = IF;
        end
    endcase
end
```

```
case(state)
    IF:begin
        PCWrite = 1;
        IRWrite = 1;
        ALUsrc1 = 2'b01;
        ALUsrc2 = 2'b01;
        ALUop = 4'b0111;
    end
    ID:begin
        if(j)begin
            PCWrite = 1;
            PCsrc = 2'b11;
        end
        else if(jal)begin
             PCWrite = 1;
             PCsrc = 2'b11;
             RegWrite = 1;
             RegDst = 2'b10;
             Jal = 1;
        end
        else if(jr)begin
            PCWrite = 1;
            PCsrc = 2'b10;
        end
        else begin
            ALUsrc1 = 2'b01;
            ALUsrc2 = 2'b11;
            ALUop = 4'b0111;
        end
    end
```

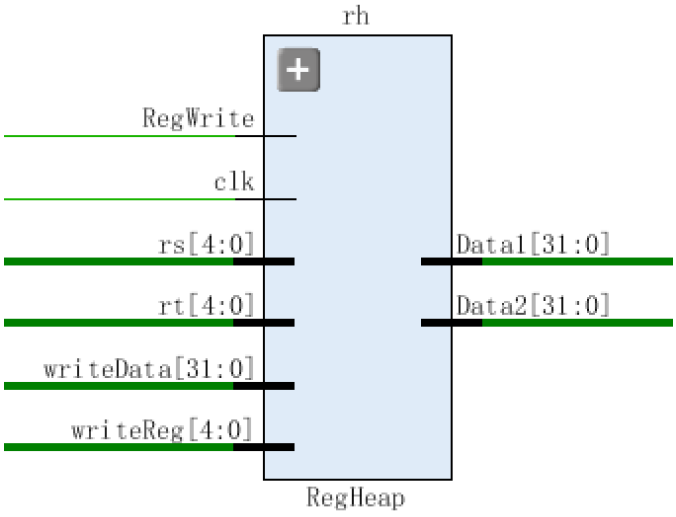
```
    ALU:begin
        ALUsrc1 = (srl | sll)? 2'b10 : 2'b00;
        ALUsrc2[1] = (addi | lw | sw) ? 1 : 0;
        ALUop[0] = (add | sub | mul | Or | Xor | addi | lw | sw | bne | beq) ? 1 : 0;
        ALUop[1] = (add | slt | And | Or | addi | lw | sw) ? 1 : 0;
        ALUop[2] = (add | slt | Not | Xor | addi | lw | sw) ? 1 : 0;
        ALUop[3] = (srl | mul) ? 1 : 0;
    end
    WREG1:begin
        RegWrite = 1;
        RegDst = (addi | lw) ? 2'b01 : 2'b00;
    end
    WREG2:begin
        RegWrite = 1;
        RegDst = 2'b01;
        MemToReg = 1;
    end
    MLD:begin
        IorD = 1;
    end
    MST:begin
        IorD = 1;
        MemWrite = 1;
    end
    EBR:begin
        ALUop = 4'b0001;
        if((beq & zero) | (bne & !zero)) begin
            PCWrite = 1;
            PCsrc = 2'b01;
        end
    end
```



5.寄存器组 RegHeap

CLK 时钟信号在下降沿时触发尝试写寄存器的操作，在 RegWrite 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。

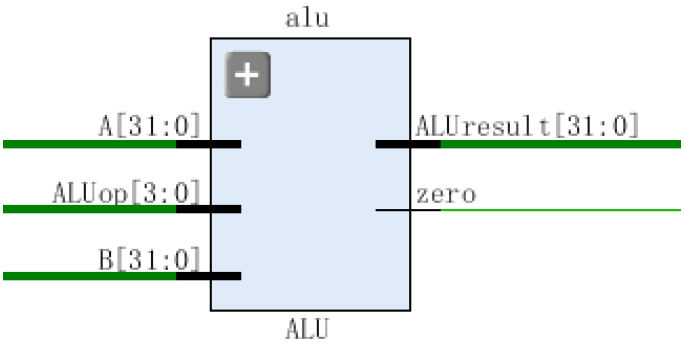
```
always @(posedge clk) begin
    Data1 <= register[rs];
    Data2 <= register[rt];
end
always @(negedge clk)begin
    if (RegWrite && writeReg)
         register[writeReg] <= writeData;
end
```



6.ALU

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

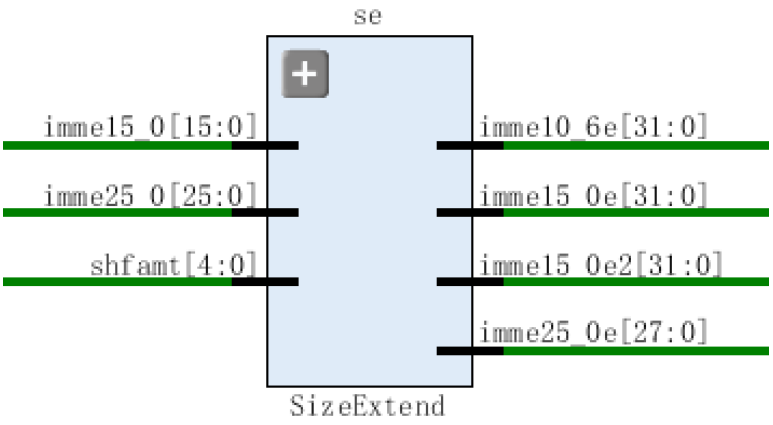
```
always @(*)
    begin
        case(ALUop)
            4'b0000:
                ALUresult = B << A;
            4'b0001:
                ALUresult = A - B;
            4'b0010:
                ALUresult = A & B;
            4'b0011:
                ALUresult = A | B;
            4'b0100:
                ALUresult = ~A;
            4'b0101:
                ALUresult = A ^ B;
            4'b0110:
                ALUresult = (A < B)? 1 : 0;
            4'b0111:
                ALUresult = A + B;
            4'b1000:
                ALUresult = B >> A;
            4'b1001:
                ALUresult = A * B;
     endcase
end
```



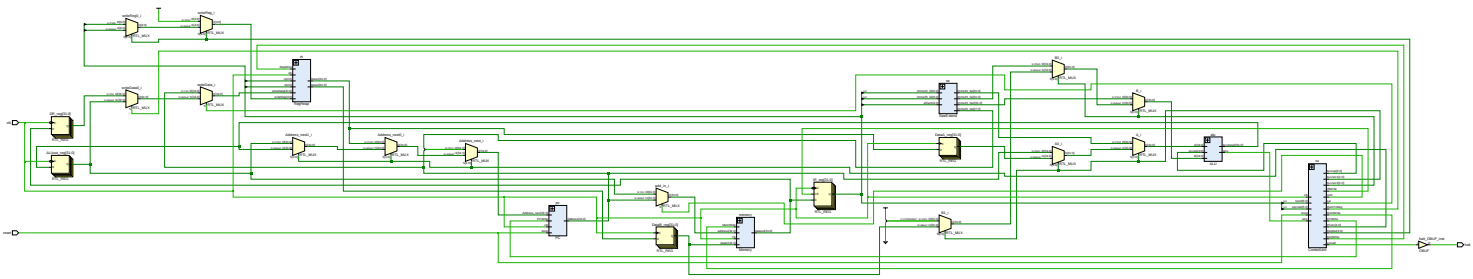
7.符号扩展/0 扩展器

对数据进行符号扩展/0 扩展.

```
assign imme10_6e[4:0] = shfamt;
assign imme10_6e[31:5] = 26'b00000000000000000000000000;
assign imme15_0e[15:0] = imme15_0;
assign imme15_0e[31:16] = imme15_0[15]? 16'b1111111111111111 : 16'b0000000000000000;
assign imme15_0e2 = imme15_0e << 2;
assign imme25_0e = imme25_0<<2;
```



8.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小到大进行排序

```
sort1:
sll $t0,$a1,2
add $t0,$t0,$s0
lw $t1,0($t0)
lw $t2,4($t0)
slt $t3,$t2,$t1
beq $t3,$zero,sort2
sw $t2,0($t0)
sw $t1,4($t0)

sort2:
addi $a1,$a1,1
addi $t4,$0,9
sub $t4,$t4,$a0
slt $t5,$a1,$t4
bne $t5,$zero,sort1
addi $a0,$a0,1
addi $t4,$0,10
slt $t5,$a0,$t4
addi $a1,$0,0
bne $t5,$zero,sort1
```

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	
0x00054080	sll $8,$5,0x00000002	2: sll $t0,$a1,2
0x01104020	add $8,$8,$16	3: add $t0,$t0,$s0
0x8d090000	lw $9,0x00000000($8)	4: lw $t1,0($t0)
0x8d0a0004	lw $10,0x00000004($8)	5: lw $t2,4($t0)
0x0149582a	slt $11,$10,$9	6: slt $t3,$t2,$t1
0x11600002	beq $11,$0,0x00000002	7: beq $t3,$zero,sort2
0xad0a0000	sw $10,0x00000000($8)	8: sw $t2,0($t0)
0xad090004	sw $9,0x00000004($8)	9: sw $t1,4($t0)
0x20a50001	addi $5,$5,0x00000001	12: addi $a1,$a1,1
0x200c0009	addi $12,$0,0x00000009	13: addi $t4,$0,9
0x01846022	sub $12,$12,$4	14: sub $t4,$t4,$a0
0x00ac682a	slt $13,$5,$12	15: slt $t5,$a1,$t4
0x15a0fff3	bne $13,$0,0xfffffff3	16: bne $t5,$zero,sort1
0x20840001	addi $4,$4,0x00000001	17: addi $a0,$a0,1
0x200c000a	addi $12,$0,0x0000000a	18: addi $t4,$0,10
0x008c682a	slt $13,$4,$12	19: slt $t5,$a0,$t4
0x20050000	addi $5,$0,0x00000000	20: addi $a1,$0,0
0x15a0ffee	bne $13,$0,0xffffffee	21: bne $t5,$zero,sort1

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 syscall 停机指令代码）

将上述得到的 16 进制代码保存为.txt 文件，在指令/数据寄存器 Memory 的 initial 语句中输入$readmemh("文件绝对路径", mem, 932); 即可将 2 进制代码读入 CPU 的 ROM 内。

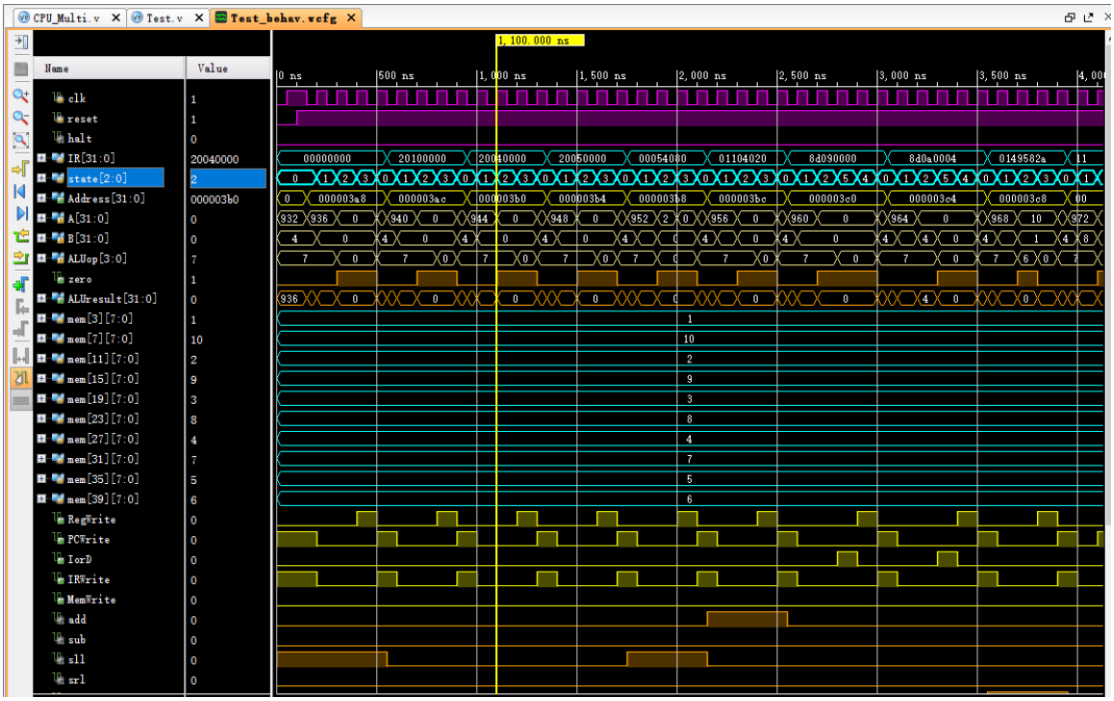
2.编写仿真代码

测试代码中实例化一个 CPU，设置好 CLK 和 Reset 信号。

```
module Test();
    reg clk;
    reg reset;
    wire halt;
    CPU_Multi cpu_multi (
        .halt(halt),
        .clk(clk),
        .reset(reset)
    );
    initial begin
            clk = 0;
            reset = 0;
            #50;
                clk = 1;
            #50;
                reset = 1;
            while(!halt)begin
                #50;
                clk=!clk;
            end
    end
endmodule
```

3.仿真测试及结果

冒泡排序的测试：



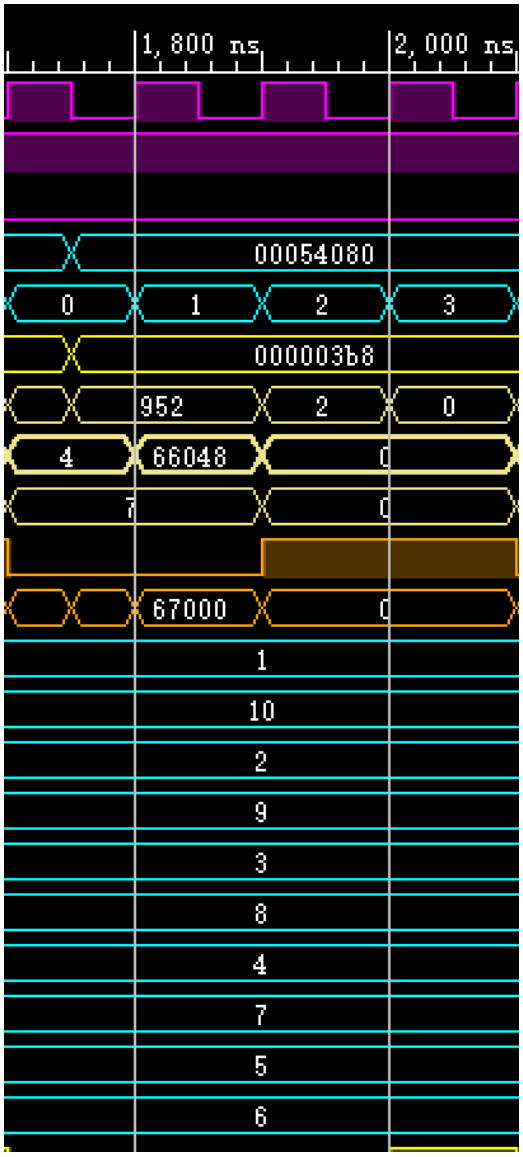
上图中间天蓝色部分为 10 个数据的数据存储器值，可以看出，程序在一开始，成功将十个数据按顺序 1、10、2、9、3、8、4、7、5、6 读入进指令/数据存储器

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：

下文中浅紫色（1-3 行）为 CLK、reset、halt 信号，天蓝色（4-5 行）为指令 16 进制代码、目前处于的阶段（0 为 IF，1 为 ID，2 为 ALU，3 为 WREG1，4 为 WREG2，5 为 MLD，6 为 MST，7 为 EBM），淡黄色（6-9 行）为现在的 PC 地址、ALU 输入 A 和 B、ALUOP 信号，橙色（10-11 行）为 zero 信号、ALU 运算结果，天蓝色（12-21 行）为 10 个冒泡排序数字。

R 型指令：（sll $t0, $a1, 2）



执行过程：

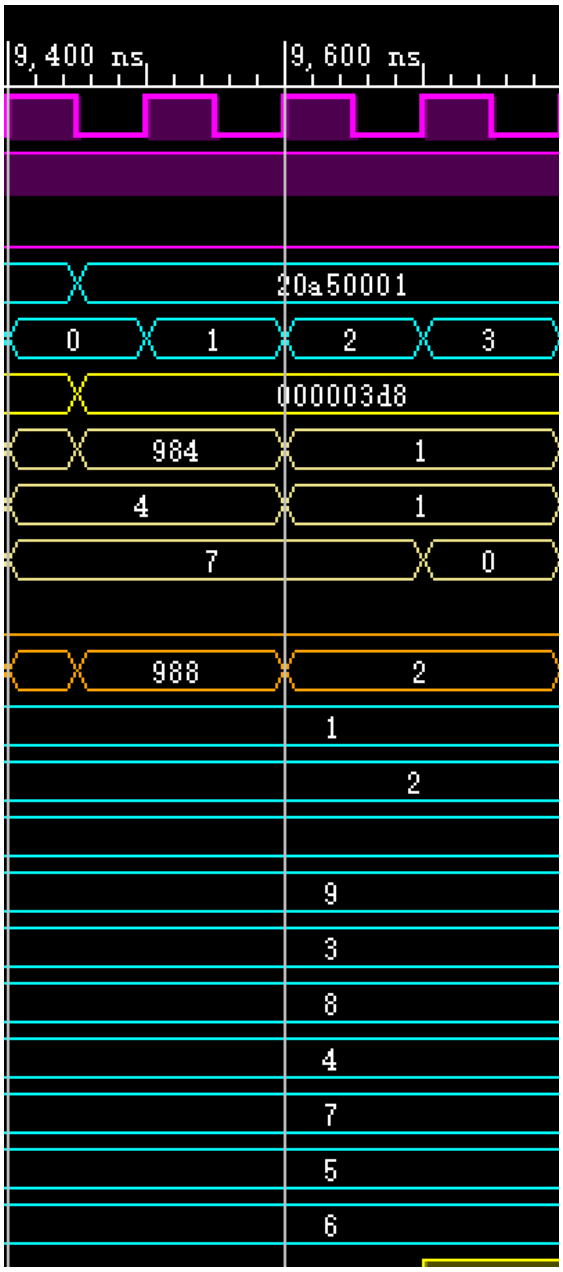
IF 阶段：时钟下降沿时传入地址给指令/数据寄存器，指令寄存器得到地址后显示指令 16 进制代码为 00054080。

ID 阶段：分析指令，为 sll 指令，将 shamt 数值传入 ALU 输入 1，将 rt 数值传入 ALU 输入 2，Control Unit 输出 ALUOP，控制 ALU 执行左移操作。

ALU 阶段：ALU 得到输入 A 的值为 2，输入 B 的值为 0，执行 0<<2 = 0，结果为 0。

WREG1 阶段：ALU 的结果 0 写入 t0 寄存器，指令执行完成。

I 型指令：（addi $a1,$a1,1）



执行过程：

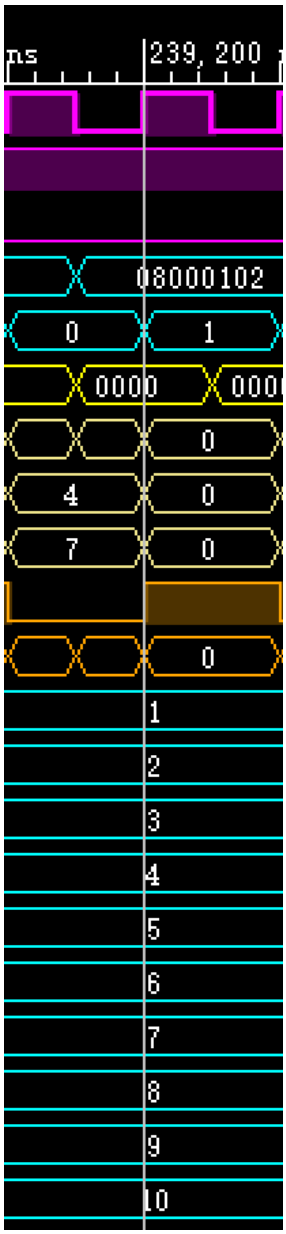
IF 阶段：时钟下降沿时传入地址给指令/数据寄存器，指令寄存器得到地址后显示指令 16 进制代码为 20a50001。

ID 阶段：分析指令，为 addi 指令，将 rt 数值传入 ALU 输入 1，将 immediate 数值传入 ALU 输入 2，Control Unit 输出 ALUOP，控制 ALU 执行加法操作。

ALU 阶段：ALU 得到输入 A 的值为 1，输入 B 的值为 1，执行 1 + 1 = 2，结果为 2。

WREG1 阶段：ALU 的结果 2 写入 a1 寄存器，指令执行完成。

J 型指令：（Jump 102h）



执行过程：

IF 阶段：时钟下降沿时传入地址给指令/数据寄存器，指令寄存器得到地址后显示指令 16 进制代码为 08000102。

ID 阶段：分析指令，为 j 指令，将 address 102h 左移两位后变成 408h，传给 PC，指令执行结束。

六.实验心得

本次多周期 CPU 设计实验中，也花费了我大量的时间，由于多周期和单周期的区别还是挺大的，我多次在原有的单周期中做了不少修改，希望能设计成多周期模式，但是都失败了。于是决定直接按多周期模式重新设计 CPU，虽然工作量大了不少，但总的来说还是完成了这项工作。

遇到的困难：

1.多周期 CPU 的流程的设计具有一定难度：在判断周期的转换、各种指令的附属周期，以及同一个阶段如何根据不同指令类型进行不同的操作，花费了不少精力去研究。

2.参考资料的缺乏和混乱：在实现多周期 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.对于多周期的同一个阶段根据不同指令类型进行不同的操作，我选择将一个阶段分成几个不同的部分，例如第三阶段分为 ALU 和 EBR，它们同属于第三阶段，但 EBR 只为 beq、bne 指令服务，进行 ALU 减法、判断符号操作，再根据符号决定地址是否跳转。而其他指令在这个阶段基本都是只需要进行 ALU 运算，则将这些指令归属于 ALU 部分，只执行 ALU 运算即可。

2.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$readmemh("文件绝对路径", mem);来写入测试代码，有的则直接一条条代码输入到指令寄存器中，后者比较麻烦也容易出错，所以本人的 CPU 采用了$readmemh("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.完成了多周期 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.对多周期 CPU 的构造图和功能原理有了更加深刻的理解。