中山大学数据科学与计算机学院本科生实验报告

（2020 学年第 1 学期）

课程名称：**计算机组成**　　任课教师：周杰英

年级	19	专业（方向）	网络安全
学号	19335295	姓名	周明豪
电话	18138405319	Email	1059759941@qq.com
开始日期		完成日期	

1. 实验目的

在 vivado 上实现单周期 CPU 仿真

2. 实验要求

要求 CPU 能完成冒泡排序及一些常用的指令内容

3. 实验内容

首先是先附上冒泡排序的程序图

```
1  sort:    addi $sp,$sp,-12     #001000
2           sw $s0,0($sp)
3           sw $s1,4($sp)
4           sw $ra,8($sp)
5           addi $s0,$0,0
6  loop1:   slt $t0,$s0,$a0   #s0 = i,s1 = j a0 =n a1 = arr
7           beq $t0,$0,exit1 #if(i<n)
8           addi $s1,$s0,-1
9  loop2:   slt $t0,$s1,$zero
10          bne $t0,$0,exit2
11          sll $t1,$s1,2
12          add $t0,$a1,$t1
13          lw $t1,0($t0)
14          lw $t2,4($t0)
15          slt $t3,$t2,$t1
16          beq $t3,$0,exit2
17          addi $sp,$sp,-4
18          sw $a0,0($sp)
19          addi $a0,$s1,0
20          jal swap
21          lw $a0,0($sp)
22          addi $sp,$sp,4
23 loopCon:addi $s1,$s1,-1
24          j loop2
25 exit2:   addi $s0,$s0,1
26          j loop1
27 exit1:   lw $s0,0($sp)
28          lw $s1,4($sp)
29          lw $ra,8($sp)
30          addi $sp,$sp,12
31          jr $ra
32 swap:    sll $t0,$a0,2
33          add $t0,$a1,$t0
34          lw $t1,0($t0)
35          lw $t2,4($t0)
36          sw $t2,0($t0)
37          sw $t1,4($t0)
38          jr $ra
39
```

0x00400000	0x23bdfff4	addi $29,$29,0xfffffff4	1:	sort:	addi $sp,$sp,-12 #001000
0x00400004	0xafb00000	sw $16,0x00000000($29)	2:		sw $s0,0($sp)
0x00400008	0xafb10004	sw $17,0x00000004($29)	3:		sw $s1,4($sp)
0x0040000c	0xafbf0008	sw $31,0x00000008($29)	4:		sw $ra,8($sp)
0x00400010	0x20100000	addi $16,$0,0x00000000	5:		addi $s0,$0,0
0x00400014	0x0204402a	slt $8,$16,$4	6:	loop1:	slt $t0,$s0,$a0 #s0 = i,s1 = j a0 =n a1 = arr
0x00400018	0x11000013	beq $8,$0,0x00000013	7:		beq $t0,$0,exit1 #if(i<n)
0x0040001c	0x2211ffff	addi $17,$16,0xffffffff	8:		addi $s1,$s0,-1
0x00400020	0x0220402a	slt $8,$17,$0	9:	loop2:	slt $t0,$s1,$zero
0x00400024	0x1500000e	bne $8,$0,0x0000000e	10:		bne $t0,$0,exit2
0x00400028	0x00114880	sll $9,$17,0x00000002	11:		sll $t1,$s1,2
0x0040002c	0x00a94020	add $8,$5,$9	12:		add $t0,$a1,$t1
0x00400030	0x8d090000	lw $9,0x00000000($8)	13:		lw $t1,0($t0)
0x00400034	0x8d0a0004	lw $10,0x00000004($8)	14:		lw $t2,4($t0)
0x00400038	0x0149582a	slt $11,$10,$9	15:		slt $t3,$t2,$t1
0x0040003c	0x11600008	beq $11,$0,0x00000008	16:		beq $t3,$0,exit2
0x00400040	0x23bdfffc	addi $29,$29,0xfffffffc	17:		addi $sp,$sp,-4
0x00400044	0xafa40000	sw $4,0x00000000($29)	18:		sw $a0,0($sp)
0x00400048	0x22240000	addi $4,$17,0x00000000	19:		addi $a0,$s1,0
0x0040004c	0x0c10001f	jal 0x0040007c	20:		jal swap
0x00400050	0x8fa40000	lw $4,0x00000000($29)	21:		lw $a0,0($sp)
0x00400054	0x23bd0004	addi $29,$29,0x00000004	22:		addi $sp,$sp,4
0x00400058	0x2231ffff	addi $17,$17,0xffffffff	23:	loopCon:	addi $s1,$s1,-1
0x0040005c	0x08100008	j 0x00400020	24:		j loop2
0x00400060	0x22100001	addi $16,$16,0x00000001	25:	exit2:	addi $s0,$s0,1
0x00400064	0x08100005	j 0x00400014	26:		j loop1
0x00400068	0x8fb00000	lw $16,0x00000000($29)	27:	exit1:	lw $s0,0($sp)
0x0040006c	0x8fb10004	lw $17,0x00000004($29)	28:		lw $s1,4($sp)
0x00400070	0x8fbf0008	lw $31,0x00000008($29)	29:		lw $ra,8($sp)
0x00400074	0x23bd000c	addi $29,$29,0x0000000c	30:		addi $sp,$sp,12
0x00400078	0x03e00008	jr $31	31:		jr $ra
0x0040007c	0x00044080	sll $8,$4,0x00000002	32:	swap:	sll $t0,$a0,2
0x00400080	0x00a84020	add $8,$5,$8	33:		add $t0,$a1,$t0
0x00400084	0x8d090000	lw $9,0x00000000($8)	34:		lw $t1,0($t0)
0x00400088	0x8d0a0004	lw $10,0x00000004($8)	35:		lw $t2,4($t0)
0x0040008c	0xad0a0000	sw $10,0x00000000($8)	36:		sw $t2,0($t0)
0x00400090	0xad090004	sw $9,0x00000004($8)	37:		sw $t1,4($t0)
0x00400094	0x03e00008	jr $31	38:		jr $ra

Code	Type	Opcode	Function	RegDst	RegWre	ALUSrcA	ALUSrcB	PCSrc	ALUOp	jump
ADD	R	000000	100000	1	1	0	0	000	000	
JR	R	000000	001000	1	0	0	0	100	000	10
SLL	R	000000	000000	1	1	1	0	000	010	
SLT	R	000000	101010	1	1	0	0	000	110	
ADDI	I	001000		0	1	0	1	000	000	
BNE	I	000101		0	0	0	0	000/001	001	
SLTI	I	001010		0	1	0	1	000	101	
BEQ	I	000100		0	0	0	0	001/000	001	
SW	I	101011		0	0	0	1	000	000	
LW	I	100011		0	1	0	1	000	000	
J	J	000010		0	0	0	0	010	000	
JAL	J	000011		0	1	0	0	010	000	1
HALT		111111		0	0	0	0	000	000	

一开始没多想，后来发现完全没必要实现 jal 和 jr 指令，平添些许麻烦，不过好些是实现了，问题不大。

模块介绍部分

PC：程序运转的开始部分，也是第一个写完的模块。如果 PCWre 的信号为 1，就让 CurPC=NextPC 就好了，每次时钟信号来时触发

```
always@(posedge CLK or negedge Reset)
begin
    if(!Reset) // Reset == 0, PC = 0
        begin
            curPC <=0;
        end
    else
        begin
            if(PCWre) // PCWre == 1
                begin
                    curPC <= nextPC;
                end
            else    // PCWre == 0, halt
                begin
                    curPC <= curPC;
                end
        end
end
```

PCADD 部分

PCadd 部分，为了节省模块，我将后面的两个 mux 都集成在了这个模块里面，也是后头加功能的时候吃了苦头，下次还是老老实实写在外面的好。PCSrc 的功能分别为普通+4；I 型指令加立即数左移两位的值；J 型指令，取 PC 高四位，扩展为的 32 位指令，最后再左移两位；停机，也就是 pc 不再往后了；JAL 指令的部分，将$ra 的地址加四给回 pc。如果 Reset 信号低电平了，就停止。

```
always@(negedge CLK or negedge Reset)
begin
    if(!Reset) begin
        nextPC <= 32'h00400000;
        PC_4 <= 32'h00400000;
    end
    else begin
        pc <= curPC + 4;
        PC_4 <= curPC + 4;
        case(PCSrc)
            3'b000: nextPC <= curPC + 4;
            3'b001: nextPC <= curPC + 4 + immediate * 4;
            3'b010: nextPC <= {pc[31:28], addr, 2'b00};
            3'b011: nextPC <= nextPC;
            3'b100: nextPC <= jal_pos + 4;
        endcase
    end
end
```

InsMem

指令存储器部分，存储了冒泡排序需要的所有尺寸。

在仿真阶段的时候可以用指令从 txt 中读取出来，但似乎写板的时候不太行，所以得把指令都写在相应的位置。由于有一百多条，一条条打容易出错又费事，写了个简单的 C++几秒钟就搞定了，科技改变生活.jpg :-)

```
    reg [7:0] rom[256:0];  // 存储器定义必须用reg类型，存储器存储单元8位长度，共128个存储单元，可以存32条指令

    // 加载数据到存储器rom。注意：必须使用绝对路径
    initial
    begin
        //$readmemb("D:\\Desktop\\bubble_test.txt", rom);
        rom[0]=8'b00100011;
        rom[1]=8'b10111101;
        rom[2]=8'b11111111;
        rom[3]=8'b11110100;
        rom[4]=8'b10101111;
        rom[5]=8'b10110000;
```

(这里的指令并不是全部的,全部有百四十行,放出来太占篇幅。以及无视上面的注释，一开始写的时候没想到指令这么长,放不下,于是就又开大了一些.

Control Unit:

这个部分将读入的指令转化为各个信号输出出去，控制着整个过程的开始，结束，复位等等。

```
initial begin
    PCWre = 1;
    InsMemRW = 1;
    RegWre = 1;
    mRD = 0;
    mWR = 0;
    DBDataSrc = 0;
    jump = 0;
end

always@(op or zero or func)
begin
    PCWre = (op == 6'b111111) ? 0 : 1;   //halt
    InsMemRW = (op == 6'b111111) ? 0 : 1;
    mWR = (op == 6'b101011) ? 1 : 0;     //sw
    mRD = (op == 6'b100011) ? 1 : 0;     //lw
    DBDataSrc = (op == 6'b100011) ? 1 : 0;      //lw,为1时输出取出的数，不然输出ALU结果
    jump = 0;   //0是数据，01是写跳转地址(jal),10是读跳转地址(jr)
    case(op)
        6'b000000:
            case(func)
                //add
                6'b100000:
                    begin
                        RegDst = 1;
```

halt 是复位信号下面的 case 依据表来填就好了

RegisterFile

寄存器组

```
    reg [31:0] regFile[0:31]; //  寄存器定义必须用reg类型    32个32位的寄存器，reg是指示这
    integer i;
    initial begin
        for (i = 0; i < 32; i = i+ 1)
            regFile[i] <= 0;          //把32个寄存器清零
        regFile[4] <= 10;
        regFile[5] <= 1;
        regFile[29] <= 127;
        //regFile[2] <=1;
    end

    always@(ReadReg1 or ReadReg2 or jump)
    begin
        ReadData1 = (jump == 2'b10)? regFile[31] : regFile[ReadReg1];     //busA=寄存器组[r
        ReadData2 = (jump == 2'b10)? 0 : regFile[ReadReg2];      //busB=寄存器组[rt(寄存器
        //$display("regfile %d %d\n", ReadReg1, ReadReg2);
    end


    always@(negedge CLK)
    begin
        if( jump == 2'b01)
            i = 31;
        else
            i = WriteReg;

        //$0恒为0，所以写入寄存器的地址不能为0
        if(RegWre && i)  //如果写信号是1且写寄存器不为NULL
            begin               //1，写入当前地址
                  regFile[i] <= WriteData;
            end
    end
```

同样是把 mux 综合在了里面，那个 jump 是控制跳转的信号，如果是 jr，就读取 31 号寄存器的值，同样，写的那边也对应了 jal

ALU

没什么特别的地方，就普通的运算逻辑单元。

```
    case(ALUOp)
        3'b000: result = A + B;
        3'b001: result = A - B;
        3'b010: result = B << A;
        3'b011: result = A | B;
        3'b100: result = A & B;
        3'b101: result = (A < B) ? 1 : 0;
        3'b110: result = (((ReadData1 < ReadData2) && (ReadData1[31] == ReadData2[31] )) ||( ( ReadData1[31] ==1 && ReadData2[31] == 0))) ? 1:0;
        3'b111: result = A ^ B;
    endcase
    zero = (result == 0) ? 1 : 0;
```

六和五一个是有符号比较，一个是无符号比较，其他就没什么太值得注意的地方了

DataMem

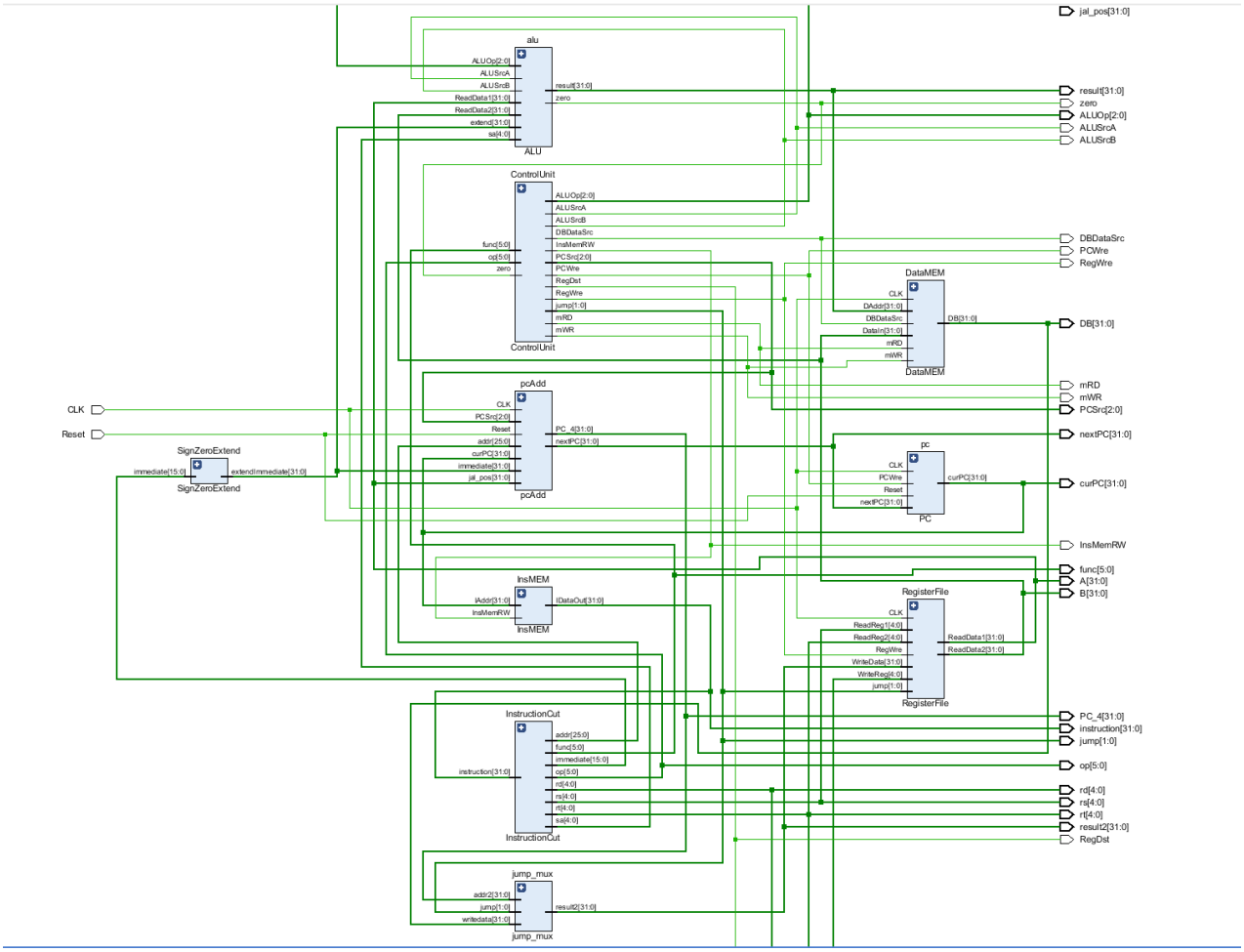
```
    always@(mRD or DAddr or DBDataSrc)
    begin
        //读
        DataOut[7:0] = mRD ? ram[DAddr + 3] : 8'bz; // z 为高阻态
        DataOut[15:8] = mRD ? ram[DAddr + 2] : 8'bz;
        DataOut[23:16] = mRD ? ram[DAddr + 1] : 8'bz;
        DataOut[31:24] = mRD ? ram[DAddr] : 8'bz;

        DB = (DBDataSrc == 0) ? DAddr : DataOut;   //Daddr是ALU算出来的结果，可能是数据，也可能是地址,DataOut是从存
    end

    always@(negedge CLK)
    begin
        //写
        if(mWR)
            begin
                ram[DAddr] = DataIn[31:24];
                ram[DAddr + 1] = DataIn[23:16];
                ram[DAddr + 2] = DataIn[15:8];
                ram[DAddr + 3] = DataIn[7:0];
            end
        //$display("mwr: %d $12 %d %d %d %d", mWR, ram[12], ram[13], ram[14], ram[15]);
    end
```

大端存放所以读入的时候顺序要注意一下

RTL 图

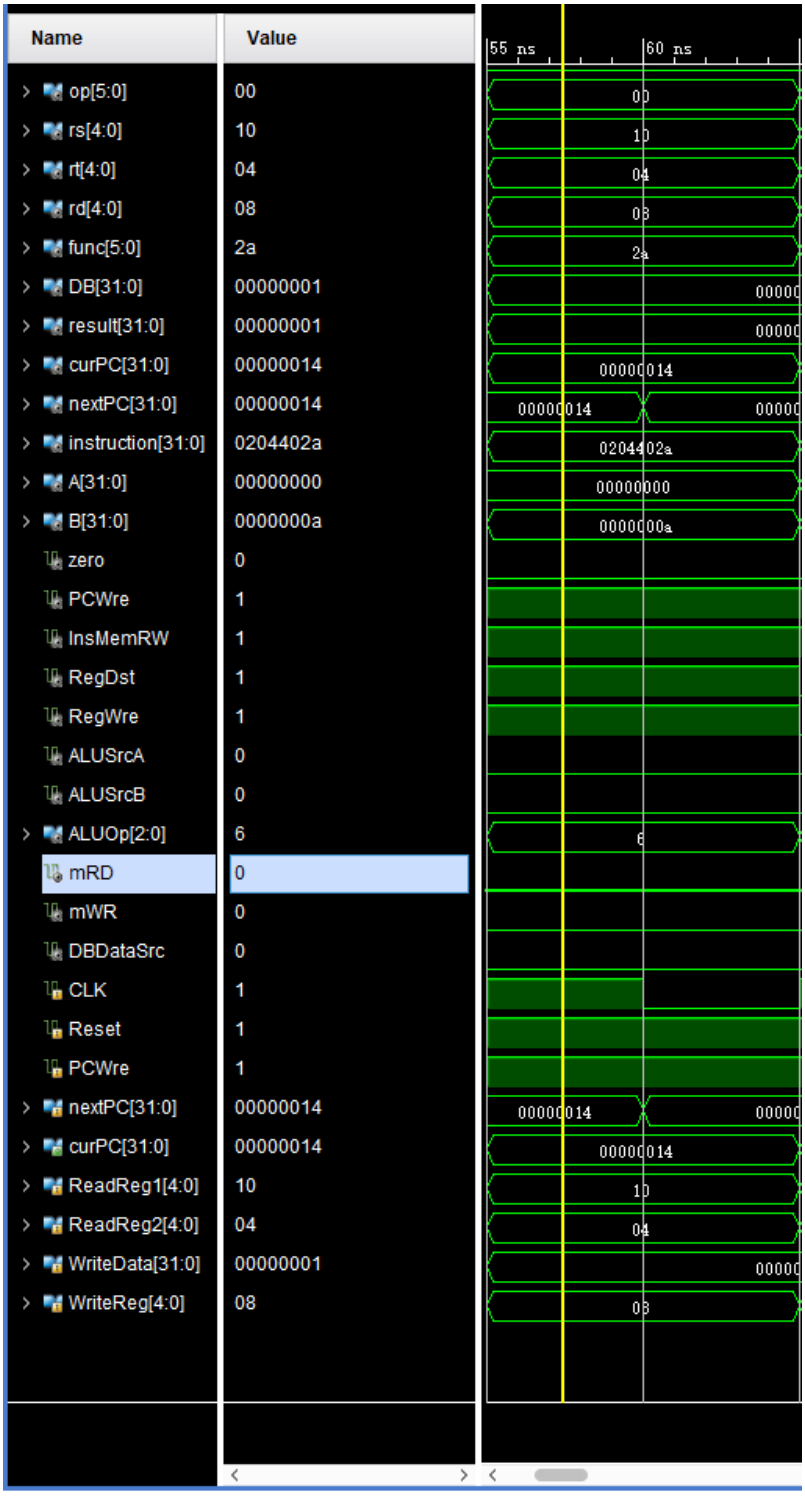


对仿真结果的解释

检查三条指令，观察结果是否正确。

R SLT 0204402a

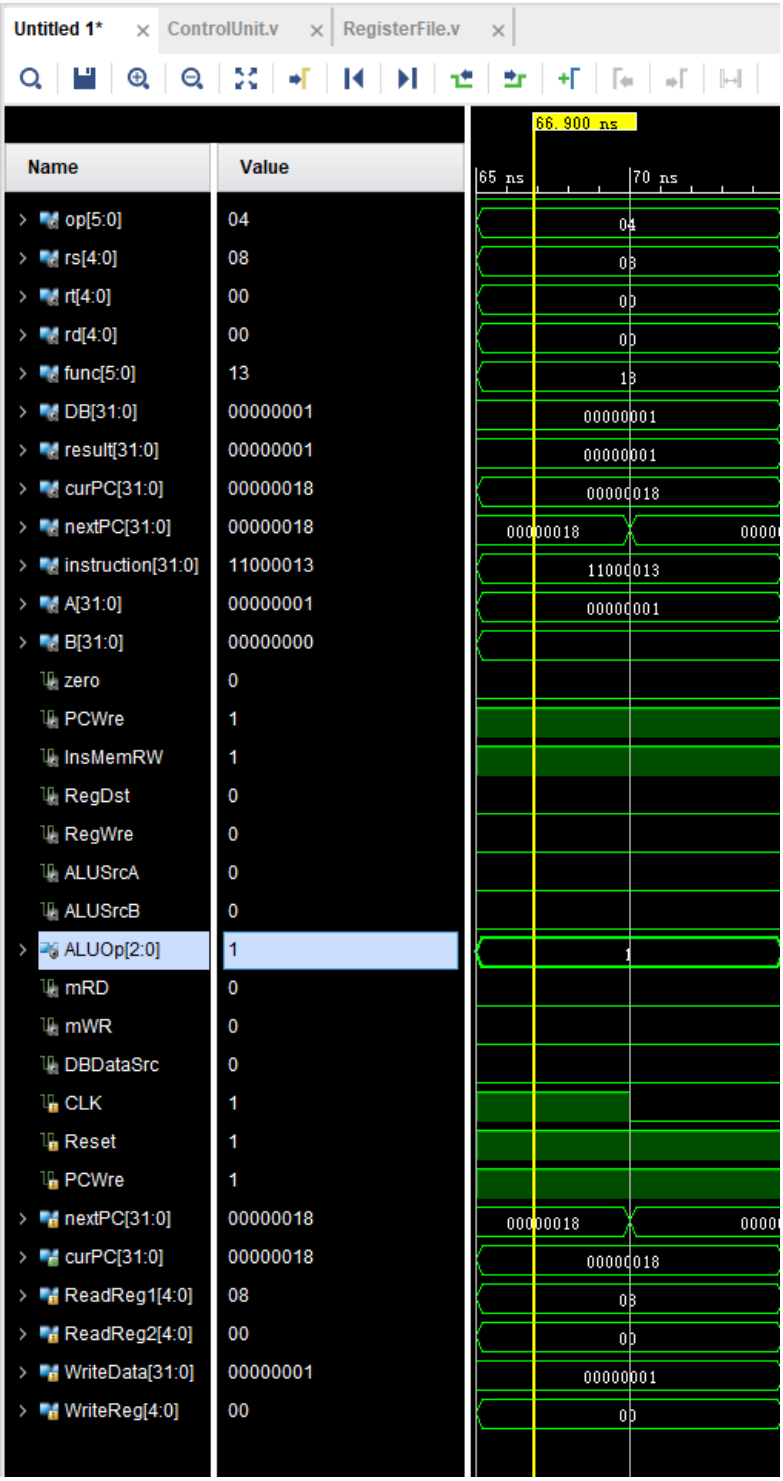
slt $t0,$s0,$a0 (其中 s0 是 i，此时为 0，a0 是 n，此时为 10)



可以看到 ALU 输入 A 是 0，B 是 a（16 进制），然后 0<a，ALU 输出了 1，写入 8 号寄存器。

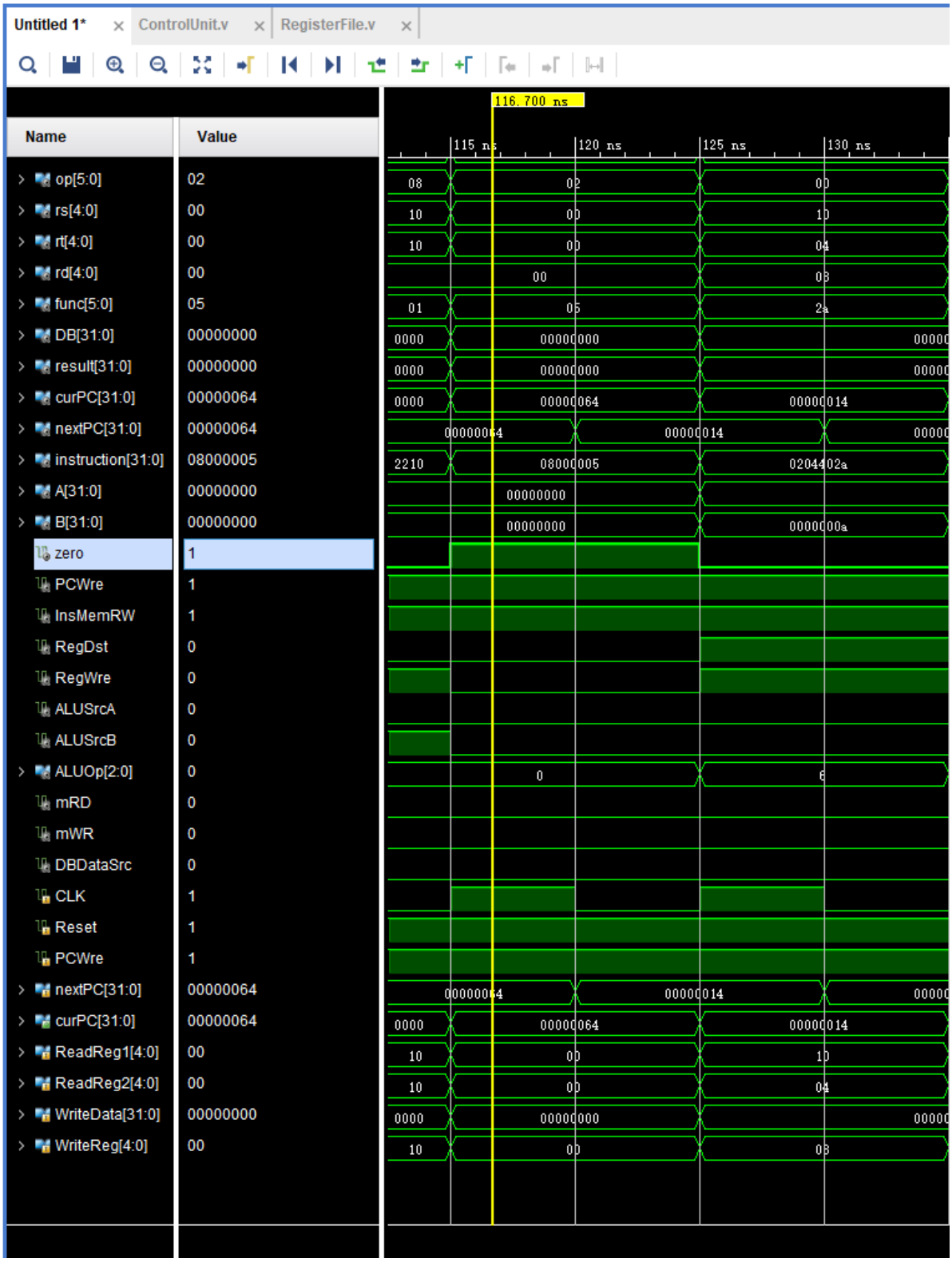
I BEQ 1100013

beq $t0 , $0, exit1



可以看到，读出了 8 号寄存器与 0 号寄存器进行了比较，发现不相等，zero 位为 0，所以不跳转，继续进行下一步操作

J Jump 08000005



通过这张图片，我们可以看到它的下一个指令，0204402a 正是它的目标 loop1，即上面那条，同时 pc 也从 64 变成了 14，跳转成功。

4. 实验总结与心得

这次实验还是有一定难度的，十分考察对单周期 CPU 的理解程度，各个指令的运行都运用到了很多个模块，每一条指令跑起来都要经过许多步骤。当最后看到指令运行成功的时候真的是非常有成就感，感觉也有很多收获。这次实验主要是在 jal 和 jr 指令的实现部分卡了一段时间，书上没有这部分的实现解释，只能自己摸索下，还是有一定难度的，但是相应的做完以后，也很有成就感。总的来说，是一次不错的实验。