北京科技大学实验报告

学院：计算机与通信工程学院　　专业：　计算机科学与技术　　班级：　计 184

姓名：　王丹琳　　学号：　41824179　　实验日期：　2020 年　6 月 2 日

实验名称：

单周期 CPU 指令扩展与仿真

实验要求：

用 VerilogHDL 或 VHDL 语言在原处理器基础上扩展两条指令，给出设计思路及扩展后的控制信号表格，仿真波形图，和对仿真波形的具体分析。最后提交该工程文件全部代码。代码应有适当的注释，并在实验报告中体现；报告中需要有指令的分析设计过程（一定包括对数据通路的分析），仿真验证过程需要有仿真波形图及波形分析。

实验仪器：

OS：Win7 64 位

Software：Vivado2018.1 开发工具

实验原理：

（一）srl 指令：它的功能是逻辑右移：

指令格式						示例	示例含义	操作及解释
BIT#	31..26	25..21	20..16	15..11	10..6			
000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	(rd)<- (rt)>>shamt,rt=$2,rd=$1,shamt=10

从 srl 的指令格式中可以知道，这是一个 R 类型的指令，function 是 6'b000010，所以当 control 模块发现高 6bit 是 6'b001101 且 func 字段为时 6'b000010，就知道当前正在处理的是 srl 指令。srl 指令实现的是将 rt 的内容逻辑右移 shamt 位，存入$rd。显然该指令需要在 ALU 模块进行处理，并将 ALU 的计算结果存入寄存器堆。

1）数据通路分析：

IFU 模块取指以后，Regfile 模块取出 inst[20:16]所代表寄存器中的数据，送到 rf_data_r2。CPU 模块控制 ct_alu_src 信号使之为 0，从而 alu_src2 通过 MUX 选择器选择 rf_data_r2 中的数据而不是立即数。然后 alu_src2（即 inst[20:16]代表的寄存器中存的数据）被送到 ALU，

Control 模块控制 ct_alu 信号使让 ALU 进行 srl 运算。右移的位数 shamt（inst[10:6]）直接接入到 ALU 模块中。ALU 模块进行了 srl 运算，得出结果赋值给 alu_res。Control 模块控制 ct_data_rf 信号使得 rf_data_w 通过 MUX 选择器选择来自 ALU 的运算结果 alu_res。然后，Control 模块控制 ct_rf_dst 信号使得 rf_addr_w 通过 Mux 选择器选择写入位置为 rd。Control 模块控制 rf_wen 信号使寄存器组写使能，运算结果 alu_res 在时钟下降沿被写入 rd 中。

2）实现的思路是:

1.对于 ALU 模块进行的操作： ALU 的控制信号 ct_alu 增加一个取值用来表示左移位（ct_alu 是 4bit 可以有 16 种不同指令）

2.对于 shamt 值的传入：把 shamt 直接接入 ALU 即可。当指示 ALU 计算左移位时，由指令解释可以看出（忽略 rs ）直接用 rt （alu_src2）和 shamt 计算。因此 ALU 模块具有 3 个输入：alu_src1、alu_src2 和 shamt 在（计算移位指令时用到）。

3.对于结果的写入：控制 control 模块输出正确信号使让 Mux 选择 rf_addr_w 的写入位置为 rd。

（二）ori 指令：它的功能是|运算：

op	rs	rt	immediate	示例	示例含义	操作及解释
001101	rs	rt	immediate	ori $1,$2,10	$1=$2|10	(rt)<-(rs)|(zero-extend)immediate,rt=$1,rs=$2

从 ori 的指令格式中可以知道，这是一个 I 类型的指令，ori 指令的指令码是 6'b001101，所以当处理器发现正在处理的指令的高 6bit 是 6'b001101 时，就知道当前正在处理的是 ori 指令。指令主要实现的是将指令中的 16 位立即数 immediate 进行无符号扩展至 32 位的 ext_data，然后与索引为 rs 的通用寄存器的值进行逻辑“或”运算，运算结果保存到索引为 rt 的通用寄存器中。很明显，该指令需要在 ALU 模块进行处理，并将 ALU 的计算结果存入通用寄存器中。

1）数据通路分析：

IFU 模块取址以后，Regfile 模块取出 inst[21:25]所代表寄存器中的数据，inst[15:0]所表示的 immediate 进行符号扩展赋值给 ext_data。Control 模块控制 ct_alu_src 信号使得 alu_src2 通过 Mux 选择器选择被符号拓展的立即数。另一个操作数（rs 中存的数据）被取出输进 ALU 模块。Control 模块控制 ct_alu 信号使得 ALU 模块进行 ori 运算，得出结果赋值给 alu_res。运算结果 alu_res 信号，alu_res 信号到达 Regfile 附近的 Mux。Control 模块控制 ct_data_rf 信号使得 rf_data_w 通过 MUX 选择器选择来自 ALU 的运算结果 alu_res 来写入寄存器中。Control 模块控制 ct_rf_dst 信号使得 rf_addr_w 通过 Mux 选择器选择写入位置为 rt。Control 模块控制 rf_wen 信号使寄存器组写使能，运算结果 alu_res 在时钟下降沿被写入 rt 中。

2）实现的思路是:

1. 对于立即数的无符号扩展：原先代码已实现

2. 对于 ALU 模块进行的操作： ALU 的控制信号 ct_alu 增加一个取值用来表示|运算（ct_alu 是 4bit 可以有 16 种不同指令）

3．对于立即数的传入：control 模块产生正确的 ct_alu_src 信号使得立即数被选择进入 ALU

4．对于结果的写入：控制 control 模块产生正确的 ct_data_rf 信号让 alu_res 送来的结果被选择写入 regfile 中。同时 control 模块产生正确的 ct_rf_dst 信号使结果写入 rt 中，产生正确的 rf_wen 信号让写使能打开，使数据被写入 rt。

实验内容与步骤：

列出扩展指令后的控制信号表。

控制	信号名	R型	lw	sw	beq	J	addiu	ori
输入	ct_inst(inst[31:26])	0	1	1	0	0	0	0
		0	0	0	0	0	0	0
		0	0	1	0	0	1	1
		0	0	0	1	0	0	1
		0	1	1	0	1	0	0
		0	1	1	0	0	1	1
输出	ct_rf_dst	1	0	x	x	x	0	0
	ct_rf_wen	1	1	0	0	0	1	1
	ct_alu_src	0	1	1	0	x	1	1
	ct_alu	srl:0001 addu:0010	0000	0000	0110	xxxx	0010	1000
	ct_branch	0	0	0	1	0	0	0
	ct_men_ren	0	1	0	0	0	0	0
	ct_men_wen	0	0	1	0	0	0	0
	ct_data_rf	0	1	x	x	x	0	0
	ct_jump	0	0	0	0	1	0	0

模块	控制信号	作用
IFU	ct_jump	执行beq指令时变为有效信号
	ct_branch	执行分支指令时变为有效信号
DataMem	ct_men_wen	往DataMem写入数据时变为有效信号
	ct_mem_ren	DataMem读出数据时变为有效信号
ALU	ct_alu	选择 ALU 要执行的运算，例如选择执行加法 或其他运算
Mux	ct_alu_src	二选一多路选择器的控制信号
	ct_rf_dst	
	ct_data_rf	
RegFile	ct_rf_wen	往RegFile写入数据数据时变为有效信号

给出关键代码并加注释

（一）srl 指令：

ALUct 模块中，给 alu_ct 增添一种取值

```
module ALUCt(
    input rst,
    input[5:0] funct,
    input[2:0] alu_ct_op,
    output reg[3:0] alu_ct//控制 ALU 进行何种操作（加/减/逻辑右移）//其由 alu_ct_op 和 function 决定
    );
    always@(*)
      if(!rst) alu_ct = 0;
      else   case(alu_ct_op)
            3'b000:alu_ct= 4'b0010;//+
            3'b010:alu_ct= 4'b0110;//beq 减法
            3'b001:alu_ct= 4'b1000;//|
            3'b100:begin case(funct)//R 型指令
//在此补充代码:当指令中 funct 段为 100001 时，alu_ ct 输出 4'b0010 (执行加法操作)。
                  6'b100001:alu_ct=4'b0010;//加法
                  6'b000010:alu_ct=4'b0001;//逻辑右移（srl 为 R 型指令）
                  default: alu_ct = 0;
                  endcase end
            default: alu_ct = 0;
            endcase
```

```
endmodule
```

ALU 模块根据 ALUct 模块增添的一种取值（4'b0001）来增加 srl 操作

另外，因为 srl 操作需要 inst[10:6]（即 shamt），故在 ALU 模块增加一个 input 来直接传入 inst[10:6]部分。

```
module ALU(
input rst,
input[3:0] alu_ct,//何种操作
input[31:0] alu_src1,alu_src2,//2 个运算数
input[4:0] shamt,
output alu_zero,//是否为 0
output reg [31:0] alu_res//运算结果
    );
    assign alu_zero= (alu_res==0)?1:0;
    always@(*)
        if(!rst) alu_res = 32'b0;
        else begin
            case(alu_ct)//在此补充代码:当 alu_ ct 为 4'b0010,执行加法运算:为 4'b0110 时， 执行减法运算。
                4'b0010:alu_res=alu_src1+alu_src2;
                4'b0110:alu_res=alu_src1-alu_src2;
                4'b0001:alu_res=alu_src2>>shamt;// 逻辑右移
                4'b1000:alu_res=alu_src1|alu_src2;// |ori 操作
                default: alu_res = 32'b0;
            endcase
        end

endmodule
```

（二）ori 指令：

在 Control 模块中，增添 wire inst_ori 来对指令高 6 位进行逻辑判断是否为 ori 指令

因为 ori 指令需要写入通用寄存器 rt 中，故需对 ct_rf_wen 进行修改

ori 指令完成后，需要传入 alu_res，所以是 ori 指令时，ct_alu_res 需要为 1

另外，因为 ori 与 addu 的 alu_ct_op 相同，无法进行区分，所以将 ct_alu_op 变为 3 位

```
module Control(
    input rst,
    input[5:0] ct_inst,
    input[5:0] aluct_inst,
    output ct_rf_dst,
    output ct_rf_wen,
    output ct_alu_src,
    output[3:0] ct_alu,
    output ct_mem_wen,
    output ct_mem_ren,
    output ct_data_rf,
    output ct_branch,
    output ct_jump
    );
    wire inst_r,inst_lw,inst_sw,inst_beq,inst_j, inst_addiu,inst_addi,inst_ori;
    wire[2:0] ct_alu_op;
    ALUCt aluct0(rst,aluct_inst,ct_alu_op,ct_alu);
    //二级逻辑阵列
    //与阵
    assign inst_r = (!ct_inst[5])&&(!ct_inst[4])&&(!ct_inst[3])&&(!ct_inst[2])&&(!ct_inst[1])&&(!ct_inst[0]);
    assign inst_lw = (ct_inst[5])&&(!ct_inst[4])&&(!ct_inst[3])&&(!ct_inst[2])&&(ct_inst[1])&&(ct_inst[0]);
    assign inst_sw = (ct_inst[5])&&(!ct_inst[4])&&(ct_inst[3])&&(!ct_inst[2])&&(ct_inst[1])&&(ct_inst[0]);
    assign inst_beq =
(!ct_inst[5])&&(!ct_inst[4])&&(!ct_inst[3])&&(ct_inst[2])&&(!ct_inst[1])&&(!ct_inst[0]);
    assign inst_j = (!ct_inst[5])&&(!ct_inst[4])&&(!ct_inst[3])&&(!ct_inst[2])&&(ct_inst[1])&&(!ct_inst[0]);
    assign inst_addiu =
(!ct_inst[5])&&(!ct_inst[4])&&(ct_inst[3])&&(!ct_inst[2])&&(!ct_inst[1])&&(ct_inst[0]);
    //补充的拓展指令
    assign inst_ori = (!ct_inst[5])&&(!ct_inst[4])&&(ct_inst[3])&&(ct_inst[2])&&(!ct_inst[1])&&(ct_inst[0]);
    //在此补充完整其余 5 条指令 inst_1w, inst_Sw, inst_beq,inst_j, inst_addiu 的表达式。
    //或阵
    assign ct_rf_dst = rst?inst_r:0;//逻辑右移必须为 1->rst==1//ori 必须为 0
    assign ct_rf_wen = rst?inst_r|| inst_lw|| inst_addiu||inst_ori:0;
    assign ct_alu_src = inst_lw|| inst_sw||inst_addiu||inst_ori;//ori 时 ct_alu_src 为 1 ，使得立即数传
入
    assign ct_alu_op[2:0] = {inst_r,inst_beq,inst_ori}; //ori 时，为 001
    assign ct_branch = inst_beq;
    assign ct_mem_ren = inst_lw;
    assign ct_mem_wen = inst_sw;
    assign ct_data_rf = inst_lw;
    assign ct_jump = inst_j;
    //在此补充完整其余控制信号的表达式:ct_branch, ct_mem_ren, ct_mem_wen, ct_data_rf,ct_jump

endmodule
```

ALUCt 模块中，将 alu_ct_op 改为[2:0]

根据 Control 模块 alu_ct_op 的取值，给 alu_ct 增添一种取值

```
module ALUCt(
    input rst,
    input[5:0] funct,
    input[2:0] alu_ct_op,
    output reg[3:0] alu_ct//控制 ALU 进行何种操作（加/减/逻辑右移）//其由 alu_ct_op 和 function 决
定
    );
    always@(*)
      if(!rst) alu_ct = 0;
      else   case(alu_ct_op)
              3'b000:alu_ct= 4'b0010;//+
              3'b010:alu_ct= 4'b0110;//beq 减法
              3'b001:alu_ct= 4'b1000;//|操作
              3'b100:begin case(funct)//R 型指令
//在此补充代码:当指令中 funct 段为 100001 时，alu_ ct 输出 4'b0010 (执行加法操作)。
                      6'b100001:alu_ct=4'b0010;//加法
                      6'b000010:alu_ct=4'b0001;//逻辑右移
                      default: alu_ct = 0;
                      endcase end
              default: alu_ct = 0;
              endcase

endmodule
```

ALU 模块根据 ALUct 模块增添的一种取值（4'b0001）来增加 srl 操作

```
module ALU(
input rst,
input[3:0] alu_ct,//何种操作
input[31:0] alu_src1,alu_src2,//2 个运算数
input[4:0] shamt,
output alu_zero,//是否为 0
output reg [31:0] alu_res//运算结果
    );
    assign alu_zero= (alu_res==0)?1:0;
    always@(*)
        if(!rst) alu_res = 32'b0;
        else begin
            case(alu_ct)//在此补充代码:当 alu_ ct 为 4'b0010,执行加法运算:为 4'b0110 时， 执行减法
运算。
                4'b0010:alu_res=alu_src1+alu_src2;
                4'b0110:alu_res=alu_src1-alu_src2;
                4'b0001:alu_res=alu_src2>>shamt;
```

```
                4'b1000:alu_res=alu_src1|alu_src2;// |ori 操作
                default: alu_res = 32'b0;
            endcase
        end

endmodule
```

给出关键仿真代码

对整个 CPU 模块仿真：

```
module CPU_tb;
   reg clk = 0;
   reg rst = 0;
   wire[31:0] addr;
   CPU cpu(clk, rst);
   initial begin
    forever #10 clk = ~clk;
    end

     initial begin
       #25 rst = 1;
       #10 rst = 0;
       #30 rst = 1;
       end
endmodule
```

实验数据：

用于测试扩展指令的汇编代码、和相应的机器码，以及转换为 16 进制的结果：

汇编代码

```
nop
addiu $8,$0,8
srl $12,$8,2
addiu $9,$0,2
ori $9,$8, 65535
```

机器代码

```
0000 0000 0000 0000 0000 0000 0000 0000 //nop
0010 0100 0000 1000 0000 0000 0000 1000// addiu $8,$0,8
0000 0000 0000 1000 0110 0000 1000 0010// srl $12,$8,2
0010 0100 0000 1001 0000 0000 0000 0010// addiu $9,$0,2
0011 0101 0000 1001 1111 1111 1111 1111// ori $9,$8, 65535
```

16 进制结果

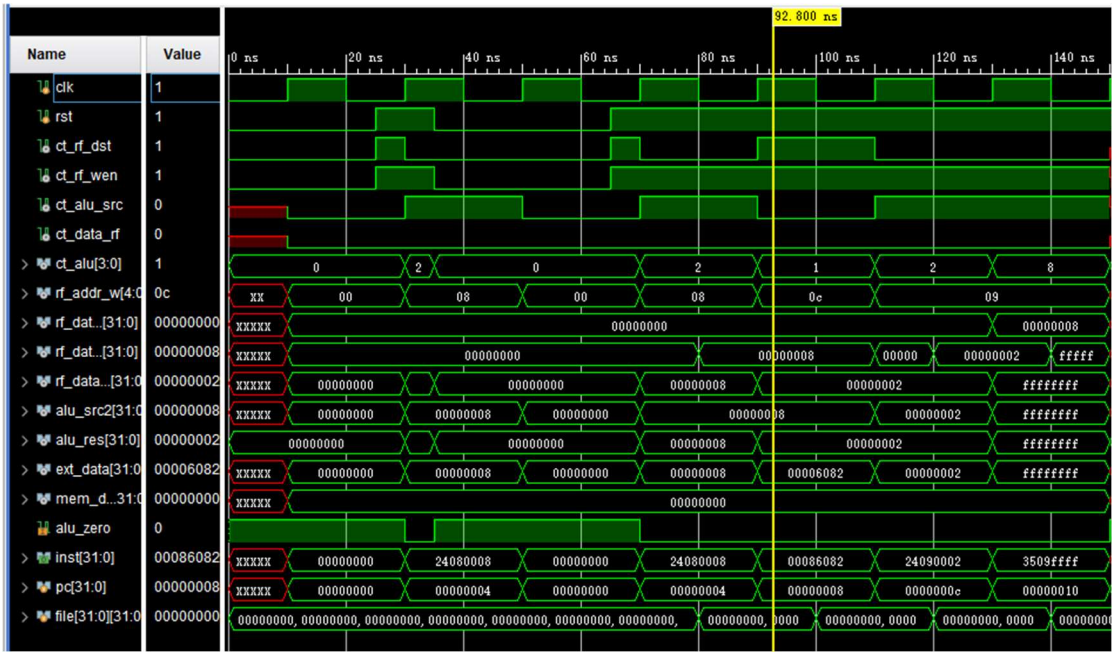
```
00000000//nop
24080008// addiu $8,$0,8
```

00086082// srl $12,$8,2

24090002// addiu $9,$0,2

3509ffff// ori $9,$8, 65535

仿真波形图



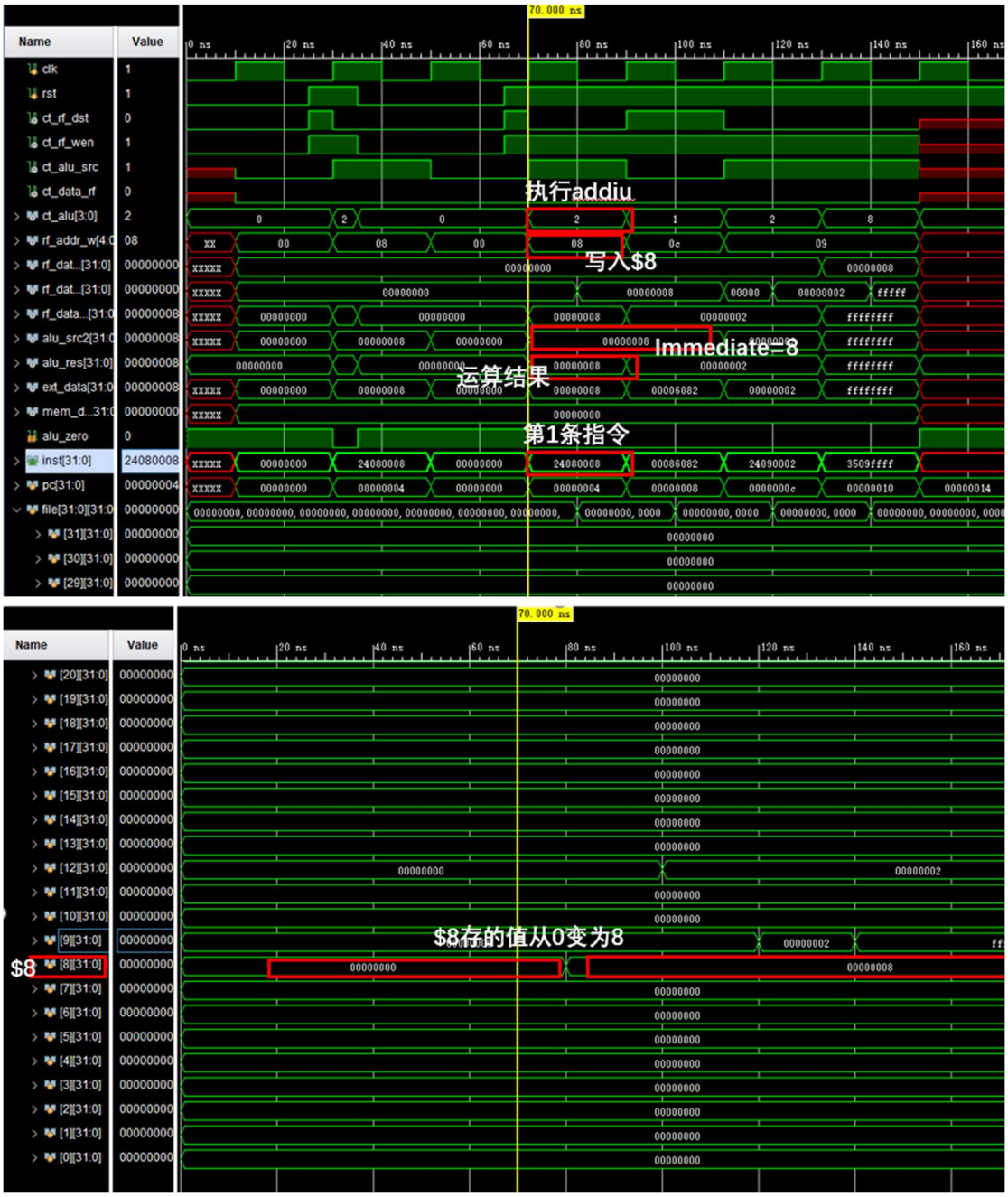
实验结果与分析：

仿真波形中关键信号值的分析，及实验结果分析。

IFU 取到的第 0 条指令是 0000 0000，此时 pc 的值加 4，取下一条指令

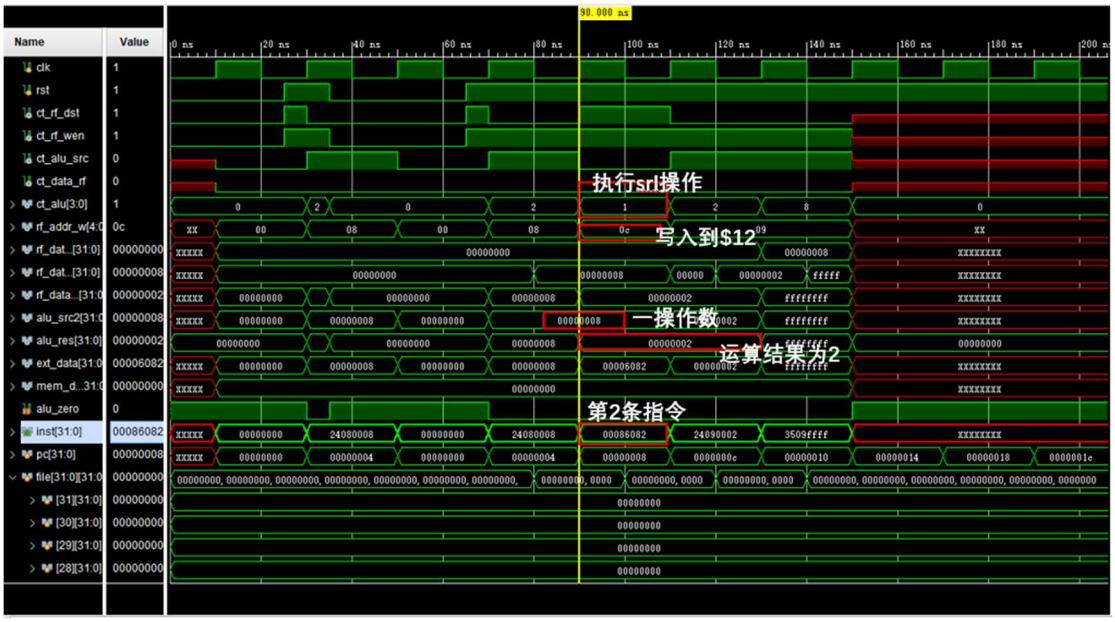
IFU 取到的第一条指令是 24080008（ addiu $8,$0,8），二进制为 0010 0100 0000 1000 0000 0000 0000 1000，执行的操作是$8=$0+8（rt=$8，rs=$0，immediate=8）

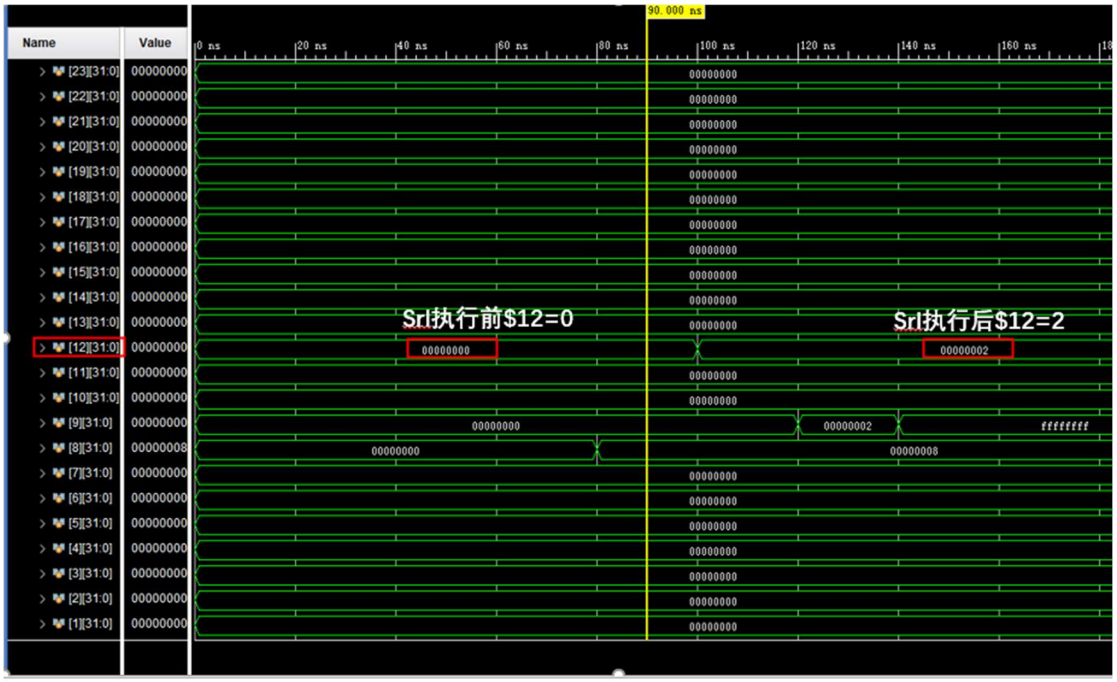
执行之前$8 的值为 0，执行之后$8 的值为 8，执行结果正确，如下图：



第一条指令完成后，pc 的值自动+4，变为 8，

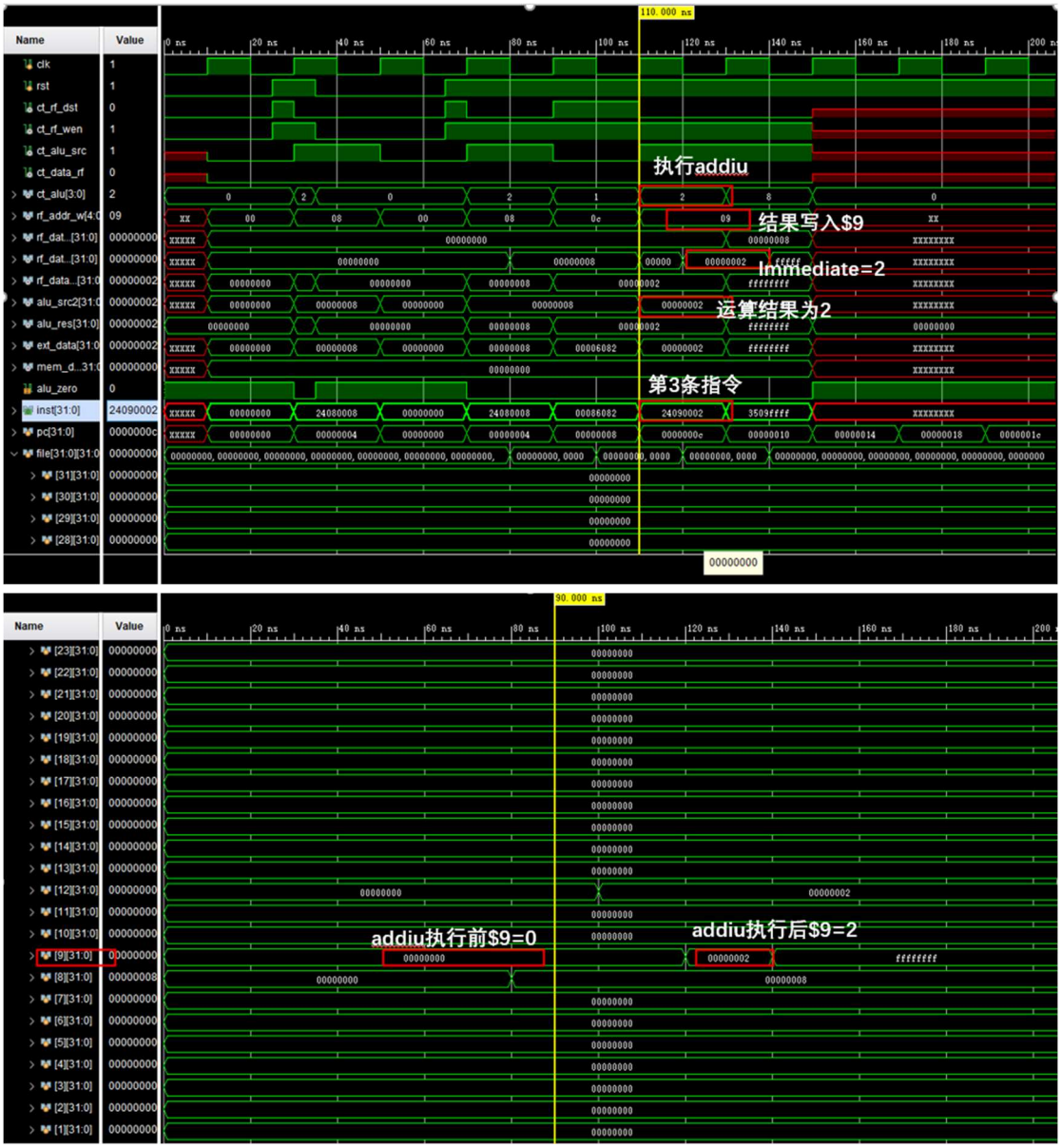
IFU 取到的第二条指令是 00086082（ srl $12,$8,2），二进制为 0000 0000 0000 1000 0110 0000 1000 0010，执行的操作是$12=$8>>2（rt=$8，rd=$12，shamt=2）执行之前$12 的值为 0，执行之后$12 的值为 2，执行结果正确，如下图：





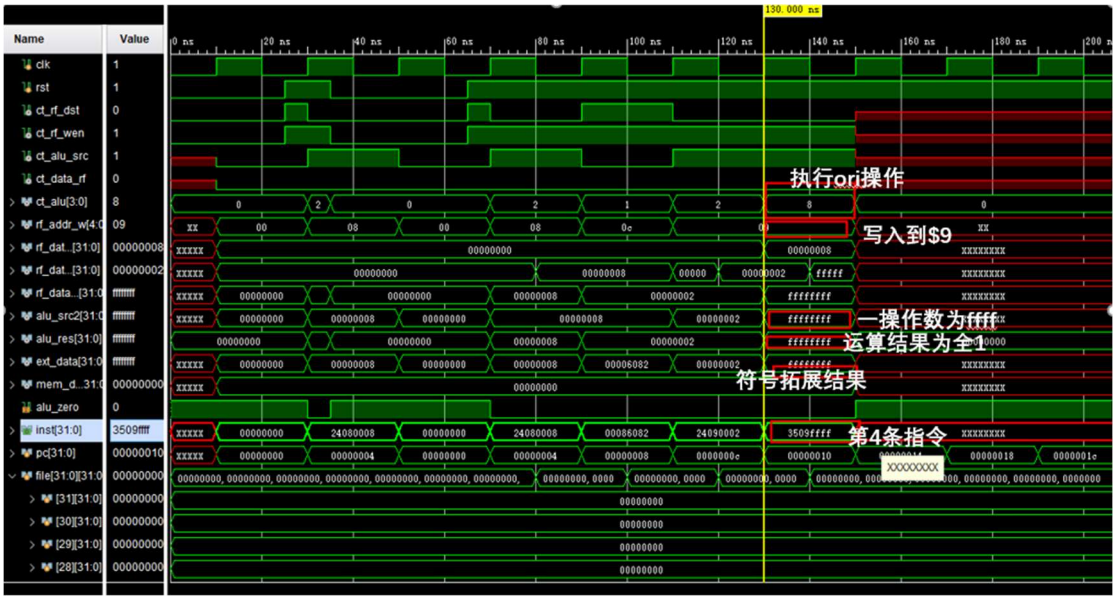
第二条指令完成后，pc 的值自动+4，变为 12，

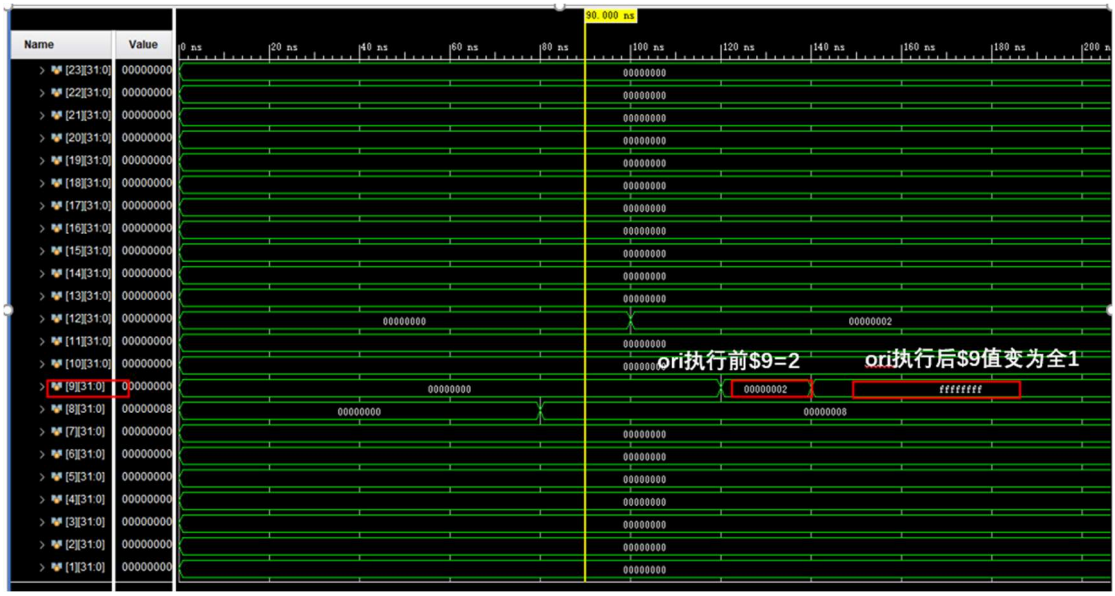
IFU 取到的第三条指令是 24090002（addiu $9,$0,2），二进制为 0010 0100 0000 1001 0000 0000 0000 0010，执行的操作是$9=$0+2（rt=$9，rs=$0，immediate=2）执行之前$9 的值为 0，执行之后$9 的值为 2，执行结果正确，如下图：



第三条指令完成后，pc 的值自动+4，变为 16，

IFU 取到的第四条指令是 3509ffff（ori $9,$8, 65535），二进制为 0011 0101 0000 1001 1111 1111 1111 1111，执行的操作是$9=$8|65535（rt=$9，rs=$0，immediate=65535）执行之前$9 的值为 2，执行之后$9 的值为变为全 1，执行结果正确，如下图：





实验总结：

通过这次实验，我实现了一个简单的单周期 MIPS 处理器，并用 Verilog 实现了该处理器的 8 条指令（add、addiu、lw、sw、beq、j、ori、srl）

模块评测阶段：

在指导书的帮助下，十分顺利的完成各个模块的代码实现。在阅读指导书的过程中，我学习到了许多知识。在最后一个模块评测（CPU）中，我通过评测成功找出代码中的一个错误，这寻找错误的过程让我受益颇多。

指令拓展阶段：

通过本次实验，我对单周期 CPU 处理指令的五个步骤（取指令、指令译码、指令执行、存储器访问、写回）有了全新的认识。同时也对各个部件的输入输出和功能有了深

刻的认识。在尝试拓展指令的过程中，我逐步厘清了 Control 模块如何对各个部件进行控制，数据如何进行处理和传送。

除此之外，我对三种指令格式也有了新的认识。通过指令拓展，我了解了 CPU 对这三种指令的处理，如 ALUCt 模块中的一个 case 解决了 R 和 I 的处理。在这个简单的 CPU 上，我对 MIPS 的指令认识更加深刻，对计算机执行指令的过程有了透彻的了解。