《计算机组成原理实验》

实验报告

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	计算机类教务一班
学 生 姓 名 ：	侯少森
学　　号 ：	18340055
时　　间 ：	2019 年 12 月 10 日

成 绩 ：

单周期CPU设计与实现

一. 实验目的

1. 理解 MIPS 常用的指令系统并掌握单周期 CPU 的工作原理与逻辑功能实现。
2. 通过对单周期 CPU 的运行状况进行观察和分析，进一步加深理解。

二. 实验内容

利用 HDL 语言，基于 Xilinx FPGA basys3 实验平台，用 Verilog HDL 语言以及 VHDL 语言来编写，实现单周期 CPU 的设计，这个单周期 CPU 能够完成 34 条MIPS 指令，包含以下指令:

- 支持基本的内存操作如 lw，sw 指令
- 支持基本的算术逻辑运算如 add，addu，addi，addiu，sub，subu，and，andi，or，ori，xor，xori，nor，slt，sltu，slti，sltiu，sll，sllv，srl，srlv，sra，srav，lui指令
- 支持基本的程序控制如 beq，bne，bgtz，bltz，j，jr，jal，halt指令

其中 ALU 的运算结果能够在开发板数码管上显示出来。

实验具体要求如下:

1. PC和寄存器组写状态使用时钟边缘触发。
2. 指令存储器数据宽度选 32 位，数据存储器存储单元宽度使用 8 位，（便于添加字节存取）即一个字节的存储单位。深度的选择满足测试代码的长度。
3. 控制器部分用控制信号真值表方法分析问题并写出逻辑表达式，或者用 case 语句方法逐个产生各指令控制信号。
4. 测试用的汇编程序段进行测试所设计的 CPU ，包含所有设计的指令。
5. 实验报告中，对每条指令有指令执行的波形（截图），且图上必须包含关键信号，同时还要对关键信号进行文字说明，以说明该指令的正确性。

三. 实验原理

1. 单周期 CPU

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行,即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。单周期 CPU 在每个 CLK 上升沿时更新 PC ，并读取新的指令。此指令无论执行时间长短，都必须在下一个上升沿到来之前完成。其时序示意如图1。

图1 单时钟周期 CPU 时序示意图

单周期 CPU 在处理指令时，一般需要经过以下几个步骤:

1. 取指令(Instruction Fetch)：根据程序计数器 PC 中的指令地址，从指令存储器中取出一条指令。同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到地址转移指令时，则由控制器把经过一些变换得到的转移地址送入 PC 。
2. 指令译码(Instruction Decode)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的控制信号，用于驱动执行状态中的各种操作。
3. 指令执行(Execute)：根据指令译码得到的控制信号，具体地执行指令动作，然后转移到结果写回状态。
4. 存储器访问(Memory Access)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。
5. 结果写回寄存器(Register Write)：将指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

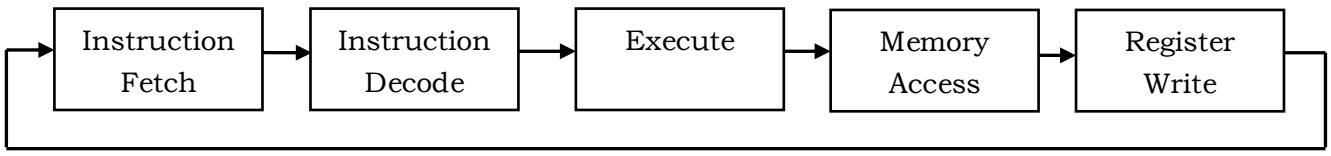


图2 单时钟周期 CPU 指令处理示意图

2. MIPS的指令的三种格式

R 型指令：

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	sa	funct

I 型指令：

31 26	25 21	20 16	15 0
op	rs	rt	immediate

J 型指令：

31 26	25 0
op	address

其中 rs 和 rt 为两个源操作数寄存器，rd 为目的操作数寄存器。shamt 为移位操作时的移位运算值，是一个立即数。func 为R型指令的功能码。immediate 为I型指令的立即数。Address为地址。

3. 数据通路图

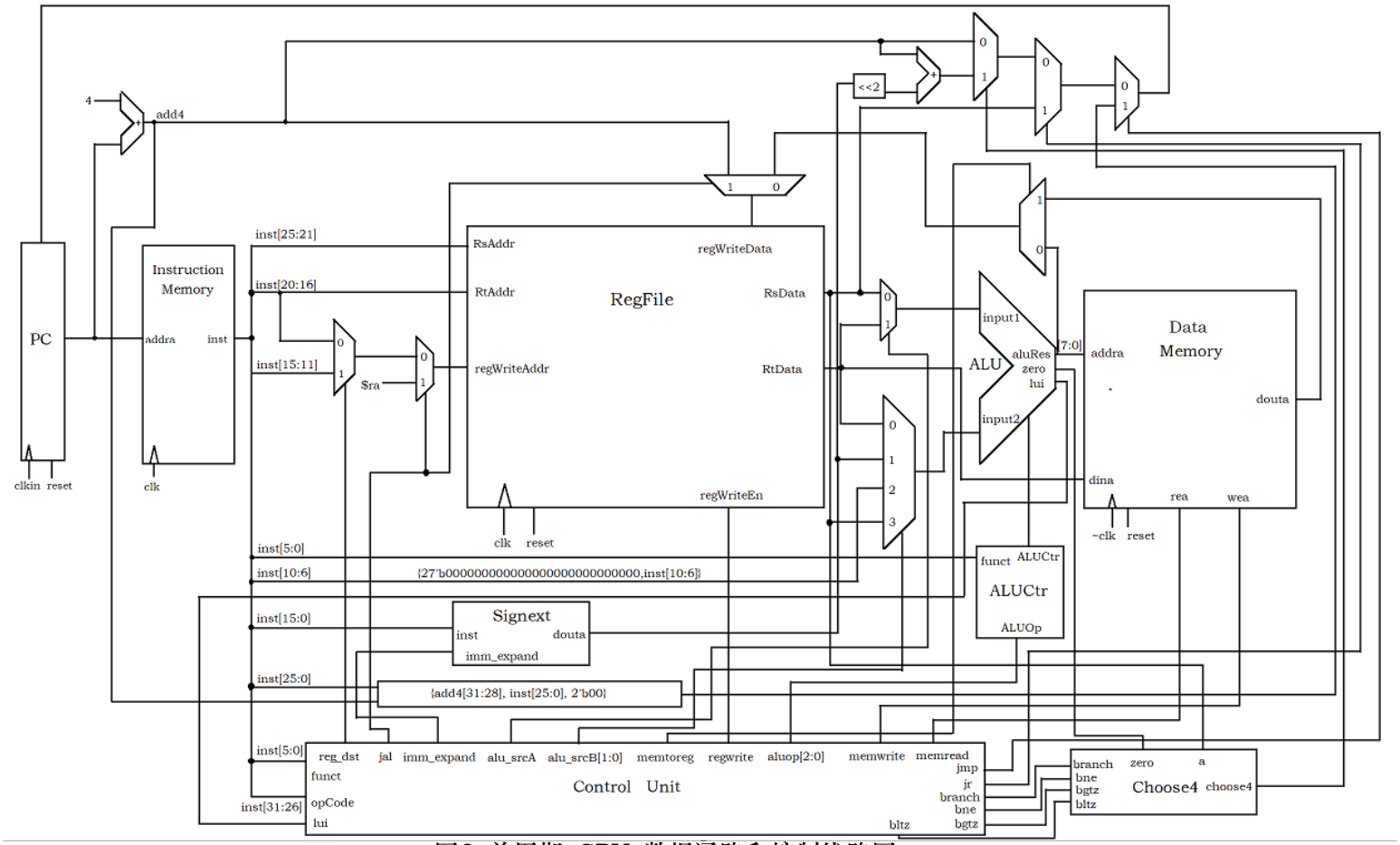


图3 单周期 CPU 数据通路和控制线路图

注：上图为本人所作，如需高清大图，请前往如下网址查看或下载

https://github.com/CrossingX/LPCO/blob/master/SingleCycleCPUDataPath.png

图3是单周期 CPU 上的数据通路和控制线路图。其中指令和数据存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时,先给出内存地址,然后由读或写信号控制操作。对于寄存器组,先给出寄存器地址,读操作时,输出端直接输出相应数据;而在写操作时,在 WE 使能信号为1时,在时钟边沿触发将数据写入寄存器。指令执行的结果总是在时钟下降沿保存到寄存器和存储器中,PC 的改变是在时钟下降沿进行的,这样稳定性较好。图中控制信号作用如表1所示,表2是 ALU 运算功能表。

表1 控制信号的作用

控制信号名	0	1
reg_dst	写寄存器组寄存器的地址，来自rt字段	写寄存器组寄存器的地址，来自rd字段
imm_expand	(sign-extend)immediate（符号扩展）	(zero-extend)immediate（0扩展）
alu_srcA	来自寄存器堆 RsData 输出	来自寄存器堆 RtData 输出，相关指令
memtoreg	来自ALU运算结果(aluRes)的输出	来自数据存储器（Data Memory）的输出
memwrite	无操作	写数据存储器（Data Memory）
memread	输出高阻态（z）	读数据存储器（Data Memory）
regwrite	无操作	写指令存储器（Instruction Memory）
jmp	无操作	跳转至{add4[31:28], inst[25:0], 2'b00}
jal	无操作	将PC+4的地址写入寄存器$31（$ra）
jr	无操作	跳转至寄存器中所存储的地址
branch	无操作	跳转至pc+4+(sign-extend)immediate<<2（zero=0）
bne	无操作	跳转至pc+4+(sign-extend)immediate<<2（zero≠0）
bgtz	无操作	若RsData>0则跳转至pc+4+(sign-extend)immediate<<2
bltz	无操作	若RsData<0则跳转至pc+4+(sign-extend)immediate<<2
lui	无操作	将16位立即数放到目的寄存器高16位，寄存器的低16位填0
alu_srcB[1:0]	00：RtData 01：(extend)immediate 10：{27'b000000000000000000000000000,inst[10:6]}	

	11：RsData
aluop[2:0]	具体见表2 ALU 运算功能表

表2 ALU运算功能表

aluop[2:0]	funct[5:0]	aluCtr[3:0]	功能描述
010/111	xxxxxx/100100	0000	与
011/111	xxxxxx/100101	0001	或
000/111	xxxxxx/10000x	0010	加
100/111	xxxxxx/100110	0100	异或
111	100111	0101	或非
001/111	xxxxxx/10001x	0110	减
101/111	xxxxxx/101010	0111	小于设置(有符号数)
111	000x00	1000	逻辑左移
111	000x10	1001	逻辑右移
111	000x11	1010	算数右移
110/111	xxxxxx/101011	1101	小于设置(无符号数)

相关部件以及引脚说明：

Instruction Memory 指令存储器：

clk：时钟输入端口

addra：指令存储器地址输入端口

inst：指令存储器指令输出端口

Register File 寄存器组：

clk：时钟输入端口

reset：重置输入端口

RsAddr：rs 寄存器地址输入端口

RtAddr：rt 寄存器地址输入端口

regWriteAddr：写入寄存器地址输入端口

regWriteData：写入寄存器数据输入端口

regWriteEn：写使能信号输入端口，为1时在 clk 上升沿出发写入

RsData：rs 寄存器数据输出端口

RtData：rt 寄存器数据输出端口

ALU 算术逻辑单元：

input1：数据1输入端口

input2：数据2输入端口

ALUCtr：ALU 控制信号输入端口，控制 ALU 的运算

aluRes：ALU 运算结果输出端口

zero：运算结果是否为 0 标志输出端口，结果为 0 则 zero 为 1，反之为 0

lui：lui控制信号输入端口

Data Memory 数据存储器：

clk：时钟信号输入端口

reset：重置信号输入端口

addra：数据存储器地址输入端口

dina：数据存储器数据输入端口

rea：读控制信号输入端口

wea：写控制信号输入端口

douta：数据存储器数据输出端口

需要说明的是数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能(当然，指令没有完全用到提供的 ALU 所有功能，但至少必须能实现所需功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出表1，这样，从表1可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表，再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

四．实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3 板一块。

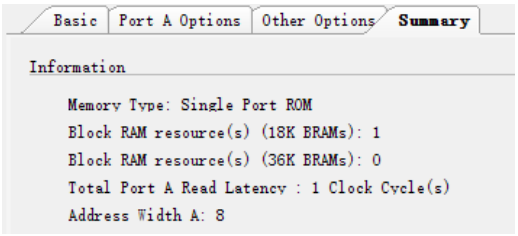
五．实验过程与结果

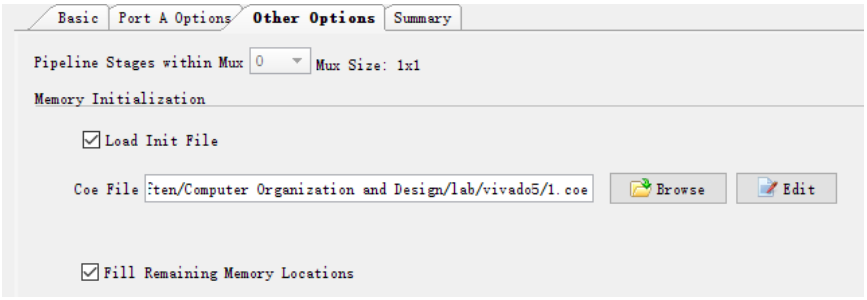
单周期 CPU 设计时，主要参考的流程图是数据通路和控制线路图。从图左边的 Instruction Memory 模块开始，从左向右进行各个模块的设计。

一、模块设计

1、Instruction Memory（指令存储器）

本实验的指令存储器模块是使用 Xilinx Vivado 提供的IP核来实现，Data width 选8位，因为指令是 32 位二进制数，要连续取四个字节。该模块由地址线、数据线和时钟信号线组成。



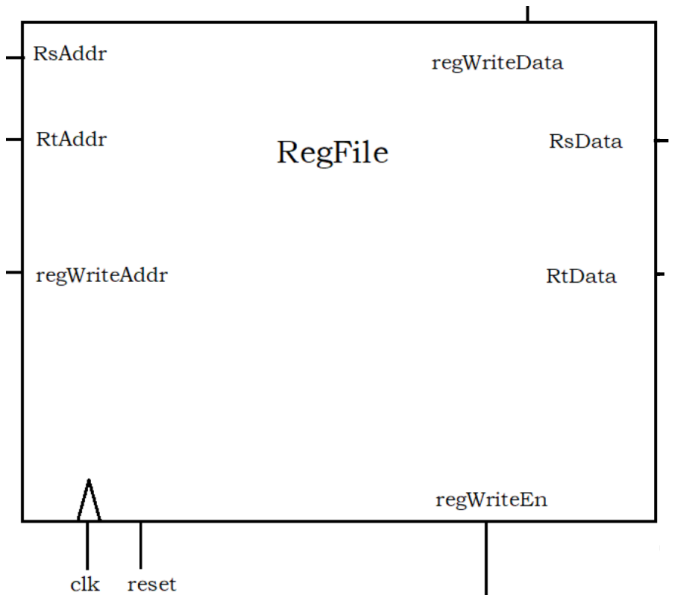


本实验用来测试单周期 CPU 运行的指令即存放在上图中的 1.coe 文件中。

2、Register File（寄存器组）

寄存器组是指令操作的主要对象，MIPS处理器里一共有32个32位的寄存器，故可以声明一个包含32个32位的寄存器数组。读寄存器时需要Rs，Rd 的地址，得到其数据。写寄存器 Rd 时需要所写地址，所写数据，同时需要写使能。以上所有操作需要在时钟和复位信号控制下操作。故寄存器组设计如下：

input	clk
input	reset
input	RsAddr[4:0]
input	RtAddr[4:0]
input	regWriteData[31:0]
input	regWriteAddr[4:0]
input	regWriteEn



output	RsData[31:0]
output	RtData[31:0]

```
module regFile(
    input clk,
    input reset,
    input [31:0] regWriteData, //写寄存器的值
    input [4:0] regWriteAddr, //写寄存器时寄存器的地址（即写哪个寄存器）
    input regWriteEn, //写寄存器使能
    input [4:0] RsAddr,   //哪个寄存器作为 RS 寄存器
    input [4:0] RtAddr,       //哪个寄存器作为 RT 寄存器
    output [31:0] RsData,         //RS 段寄存器的值
    output [31:0] RtData //RT 寄存器的值
    );
    //寄存器地址都是 5 位二进制数，因为寄存器只有 32 个，5 位就能表示所有寄
    存器
    Reg [31:0] regs [0:31]; // 寄存器组
    //  根据地址读出 Rs、Rt 寄存器数据
    assign RsData = (RsAddr == 5'b0 ) ? 32'b0 : regs[RsAddr];
    assign RtData = (RtAddr == 5'b0 ) ? 32'b0 : regs[RtAddr];
    integer i;
    always @( posedge clk ) // 时钟上升沿操作
    begin
        if(!reset) begin
            if(regWriteEn == 1) begin
            //  写使能信号为 1 时写操作
                regs[regWriteAddr] = regWriteData;
            end
        end
        else begin
```

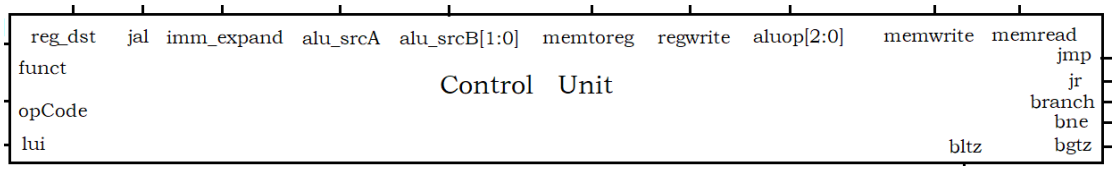
```
        //  重置时所有寄存器赋值为 0，复位
                for(i = 0; i < 32; i = i + 1)   regs[i] = 0;
            end
        end
    endmodule
```

3、Control Unit（控制单元）

控制器模块输入为指令的操作码 opcode 段，输出各个复用器、存储器读写等的信号，控制数据通路的正常进行。根据指令的不同，输出不同的信号。

控制单元的设计如下：

input	opcode[5:0]
input	funct[5:0]
output	reg_dst
output	imm_expand
output	alu_srcA
output	alu_srcB[1:0]
output	memtoreg
output	regwrite
output	memread
output	memwrite
output	aluop[2:0]
output	jmp
output	jal
output	jr
output	branch
output	bne
output	bgtz
output	bltz
output	lui



控制信号与指令的真值表如下：

第一部分：

opCode	指令	reg_dst	imm_expand	jmp	jal	jr	branch	bne	regwrite	bgtz	bltz
000000	add	1	0	0	0	0	0	0	1	0	0
000000	addu	1	0	0	0	0	0	0	1	0	0
000000	sub	1	0	0	0	0	0	0	1	0	0
000000	subu	1	0	0	0	0	0	0	1	0	0
000000	and	1	0	0	0	0	0	0	1	0	0
000000	or	1	0	0	0	0	0	0	1	0	0
000000	xor	1	0	0	0	0	0	0	1	0	0
000000	nor	1	0	0	0	0	0	0	1	0	0
000000	slt	1	0	0	0	0	0	0	1	0	0
000000	sltu	1	0	0	0	0	0	0	1	0	0
000000	sll	1	0	0	0	0	0	0	1	0	0
000000	srl	1	0	0	0	0	0	0	1	0	0
000000	sra	1	0	0	0	0	0	0	1	0	0
000000	sllv	1	0	0	0	0	0	0	1	0	0

000000	srlv	1	0	0	0	0	0	0	1	0	0
000000	srav	1	0	0	0	0	0	0	1	0	0
000000	jr	1	0	0	0	1	0	0	0	0	0
001000	addi	0	0	0	0	0	0	0	1	0	0
001001	addiu	0	0	0	0	0	0	0	1	0	0
001100	andi	0	1	0	0	0	0	0	1	0	0
001101	ori	0	1	0	0	0	0	0	1	0	0
001110	xori	0	1	0	0	0	0	0	1	0	0
001111	lui	0	0	0	0	0	0	0	1	0	0
100011	lw	0	0	0	0	0	0	0	1	0	0
101011	sw	0	0	0	0	0	0	0	0	0	0
000100	beq	0	0	0	0	0	1	0	0	0	0
000101	bne	0	0	0	0	0	1	1	0	0	0
000001	bltz	0	0	0	0	0	0	0	0	0	1
000111	bgtz	0	0	0	0	0	0	0	0	1	0
001010	slti	0	0	0	0	0	0	0	1	0	0
001011	sltiu	0	0	0	0	0	0	0	1	0	0
000010	j	0	0	1	0	0	0	0	0	0	0

000011	jal	0	0	0	1	0	0	0	1	0	0
111111	halt	0	0	0	0	0	0	0	0	0	0

第二部分:

opCode	指令	alu_srcA	alu_srcB	memtoreg	memread	memwrite	aluop	lui
000000	add	0	00	0	0	0	111	0
000000	addu	0	00	0	0	0	111	0
000000	sub	0	00	0	0	0	111	0
000000	subu	0	00	0	0	0	111	0
000000	and	0	00	0	0	0	111	0
000000	or	0	00	0	0	0	111	0
000000	xor	0	00	0	0	0	111	0
000000	nor	0	00	0	0	0	111	0
000000	slt	0	00	0	0	0	111	0
000000	sltu	0	00	0	0	0	111	0
000000	sll	1	10	0	0	0	111	0
000000	srl	1	10	0	0	0	111	0
000000	sra	1	10	0	0	0	111	0
000000	sllv	1	11	0	0	0	111	0
000000	srlv	1	11	0	0	0	111	0

000000	srav	1	11	0	0	0	111	0
000000	jr	0	00	0	0	0	111	0
001000	addi	0	01	0	0	0	000	0
001001	addiu	0	01	0	0	0	000	0
001100	andi	0	01	0	0	0	010	0
001101	ori	0	01	0	0	0	011	0
001110	xori	0	01	0	0	0	100	0
001111	lui	0	01	0	0	0	000	1
100011	lw	0	01	1	1	0	000	0
101011	sw	0	01	0	0	1	000	0
000100	beq	0	00	0	0	0	001	0
000101	bne	0	00	0	0	0	001	0
000001	bltz	0	01	0	0	0	001	0
000111	bgtz	0	01	0	0	0	001	0
001010	slti	0	01	0	0	0	101	0
001011	sltiu	0	01	0	0	0	110	0
000010	j	0	00	0	0	0	000	0
000011	jal	0	00	0	0	0	000	0

111111	halt	0	00	0	0	0	000	0

```
module ctr(
    input [5:0] opCode,   //操作符
    input [5:0] funct,    //功能码
    output reg regDst,    //写寄存器组地址信号
    output reg aluSrcA,   //ALU左操作数选择信号
    output reg [1:0] aluSrcB,    //ALU右操作数选择信号
    output reg memToReg,  //写入寄存器数据选择信号
    output reg regWrite,  //寄存器组写使能信号
    output reg memRead,   //数据寄存器读使能信号
    output reg memWrite,  //数据寄存器写使能信号
    output reg branch,    //分支指令信号
    output reg [2:0] aluop,      //ALU功能选择信号
    output reg jmp,   //跳转指令信号
    output reg jal,   //跳转并链接指令信号
    output reg jr,    //跳转寄存器指令信号
    output reg lui,   //
    output reg bne,   //分支指令信号
    output reg bltz,  //分支指令信号
    output reg bgtz,  //分支指令信号
    output reg imm_expand, //0 is signext, 1 is zeroext
);
always @(opCode,funct)
begin
    case(opCode)
        6'b001111:    //lui
        begin
            regDst = 0;
            aluSrcA = 0;
```

```
        aluSrcB = 2'b01;
        memToReg = 0;
        regWrite = 1;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b000;
        jmp = 0;
        jal=0;
        jr=0;
        lui=1;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
    end
    6'b000010: // j
    begin
        regDst = 0;
        aluSrcA = 0;
        aluSrcB = 2'b00;
        memToReg = 0;
        regWrite = 0;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b000;
        jmp = 1;
        jal=0;
        jr=0;
```

```
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
    end
    6'b000011: // Jal型指令
    begin
        regDst = 0;
        aluSrcA = 0;
        aluSrcB = 2'b00;
        memToReg = 0;
        regWrite = 1;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b000;
        jmp = 1;
        jal=1;
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
    end
    6'b000000: // R型指令
    begin
        regDst = 1;
        memToReg = 0;
```

```
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b111;
        jmp = 0;
        jal=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
        if(funct==6'b001000)
            begin regWrite = 0; jr=1;  end   //jr
        else
            begin regWrite = 1; jr=0;  end
        //sll,srl,sra
        if(funct==6'b000000||funct==6'b000010||funct==6'b000011)
        begin aluSrcA=1'b1; aluSrcB = 2'b10;  end  //rt shamt
        //sllv srlv srav
        else
        if(funct==6'b000100||funct==6'b000110||funct==6'b000111)
        begin aluSrcA=1'b1; aluSrcB = 2'b11;  end  //rt rs
        else
        begin aluSrcA=1'b0; aluSrcB = 2'b00;  end  //rs rt
    end
    6'b100011: // lw
    begin
        regDst = 0;
        aluSrcA = 1'b0;
        aluSrcB = 2'b01;
```

```
        memToReg = 1;
        regWrite = 1;
        memRead = 1;
        memWrite = 0;
        branch = 0;
        aluop = 3'b000;
        jmp = 0;
        jal=0;
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
    end
    6'b101011: // sw
    begin
        regDst = 0;
        aluSrcA = 1'b0;
        aluSrcB = 2'b01;
        memToReg = 0;
        regWrite = 0;
        memRead = 0;
        memWrite = 1;
        branch = 0;
        aluop = 3'b000;
        jmp = 0;
        jal=0;
        jr=0;
        lui=0;
```

```
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
        flag=2'b11;
    end
    6'b000100: //beq指令
    begin
        regDst = 0;
        aluSrcA = 1'b0;
        aluSrcB = 2'b00;
        memToReg = 0;
        regWrite = 0;
        memRead = 0;
        memWrite = 0;
        branch = 1;
        aluop = 3'b001;
        jmp = 0;
        jal=0;
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
    end
    6'b000101: //bne指令
    begin
        regDst = 0;
        aluSrcA = 1'b0;
```

```
        aluSrcB = 2'b00;
        memToReg = 0;
        regWrite = 0;
        memRead = 0;
        memWrite = 0;
        branch = 1;
        aluop = 3'b001;
        jmp = 0;
        jal=0;
        jr=0;
        lui=0;
        bne=1;
        bltz=0;
        bgtz=0;
        imm_expand=0;
    end
    6'b000001: //bltz指令
    begin
        regDst = 0;
        aluSrcA = 1'b0;
        aluSrcB = 2'b01;
        memToReg = 0;
        regWrite = 0;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b001;
        jmp = 0;
        jal=0;
        jr=0;
```

```
        lui=0;
        bne=0;
        bltz=1;
        bgtz=0;
        imm_expand=0;
    end
    6'b000111: //bgtz指令
    begin
        regDst = 0;
        aluSrcA = 1'b0;
        aluSrcB = 2'b01;
        memToReg = 0;
        regWrite = 0;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b001;
        jmp = 0;
        jal=0;
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=1;
        imm_expand=0;
    end
    6'b001000: // addi指令
    begin
        regDst = 0;
        aluSrcA = 1'b0;//
```

```
        aluSrcB = 2'b01;
        memToReg = 0;
        regWrite = 1;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b000;
        jmp = 0;
        jal=0;
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
        flag=2'b11;
    end
    6'b001001: // addiu指令
    begin
        regDst = 0;
        aluSrcA = 1'b0;
        aluSrcB = 2'b01;
        memToReg = 0;
        regWrite = 1;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b000;
        jmp = 0;
        jal=0;
```

```
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
    end
    6'b001100: // andi指令
    begin
        regDst = 0;
        aluSrcA = 1'b0;
        aluSrcB = 2'b01;
        memToReg = 0;
        regWrite = 1;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b010;
        jmp = 0;
        jal=0;
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=1;
    end
    6'b001101: // ori指令
    begin
        regDst = 0;
```

```
        aluSrcA = 1'b0;
        aluSrcB = 2'b01;
        memToReg = 0;
        regWrite = 1;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b011;
        jmp = 0;
        jal=0;
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=1;
    end
    6'b001110: // xori指令
    begin
        regDst = 0;
        aluSrcA = 1'b0;
        aluSrcB = 2'b01;
        memToReg = 0;
        regWrite = 1;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b100;
        jmp = 0;
        jal=0;
```

```
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=1;
    end
    6'b001010: // slti指令
    begin
        regDst = 0;     //rt
        aluSrcA = 1'b0; //rs
        aluSrcB = 2'b01;//SignImm
        memToReg = 0;
        regWrite = 1;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b101;
        jmp = 0;
        jal=0;
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
    end
    6'b001011: // sltiu指令
    begin
        regDst = 0;     //rt
```

```
        aluSrcA = 1'b0; //rs
        aluSrcB = 2'b01;//SignImm
        memToReg = 0;
        regWrite = 1;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b110;
        jmp = 0;
        jal=0;
        jr=0;
        lui=0;
        bne=0;
        bltz=0;
        bgtz=0;
        imm_expand=0;
    end
    default: // 缺省值
    begin
        regDst = 0;
        aluSrcA = 1'b1;
        aluSrcB = 2'b00;
        memToReg = 0;
        regWrite = 0;
        memRead = 0;
        memWrite = 0;
        branch = 0;
        aluop = 3'b000;
        jmp = 0;
        jal=0;
```

```
            jr=0;
            lui=0;
            bne=0;
            bltz=0;
            bgtz=0;
            imm_expand=0;
        end
    endcase
end
endmodule
```

4、ALUCtr（ALU 控制译码）

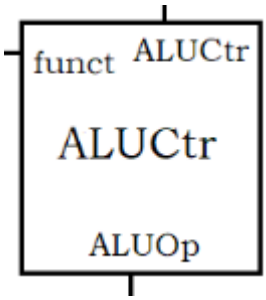
ALU主要执行 5 种操作：与，或，加，减，小于设置。这五种操作可以使用四位的编码表示：0000，0001，0010，0110，0111。指令不同，则对应的 ALU 运算不同，所以该模块需要根据指令来控制 ALU 进行正确的运算。

该模块实现 3 位操作码以及6位功能码输出 4 位 ALU 控制信号码。ALU 控制信号表如下：

aluop[2:0]	funct[5:0]	aluCtr[3:0]	功能描述
010/111	xxxxxx/100100	0000	与
011/111	xxxxxx/100101	0001	或
000/111	xxxxxx/10000x	0010	加
100/111	xxxxxx/100110	0100	异或
111	100111	0101	或非
001/111	xxxxxx/10001x	0110	减
101/111	xxxxxx/101010	0111	小于设置(有符号数)
111	000x00	1000	逻辑左移
111	000x10	1001	逻辑右移
111	000x11	1010	算数右移
110/111	xxxxxx/101011	1101	小于设置(无符号数)

ALUCtr的设计如下：

input	ALUOp[2:0]
input	funct[5:0]
output	ALUCtr[3:0]



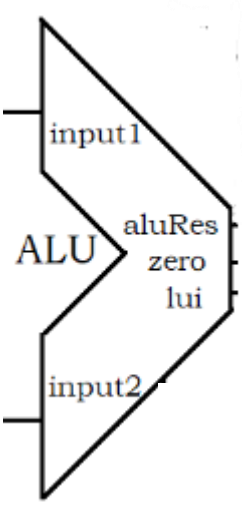

```
module aluctr(
    input [2:0] ALUOp,
    input [5:0] funct,
    output reg [3:0] ALUCtr
);
    always @(ALUOp or funct)
        casex({ALUOp, funct})
        //非R型
        9'b000xxxxxx: ALUCtr = 4'b0010; // add
        9'b001xxxxxx: ALUCtr = 4'b0110; // sub
        9'b010xxxxxx: ALUCtr = 4'b0000; // and
        9'b011xxxxxx: ALUCtr = 4'b0001; // or
        9'b100xxxxxx: ALUCtr = 4'b0100; // xor
        9'b101xxxxxx: ALUCtr = 4'b0111; // slt
        9'b110xxxxxx: ALUCtr = 4'b1101; // sltu
        //R型
        9'b111_10000x: ALUCtr = 4'b0010; // add/addu
        9'b111_10001x: ALUCtr = 4'b0110; // sub/subu
        9'b111_100100: ALUCtr = 4'b0000; // and
        9'b111_100111: ALUCtr = 4'b0101; // nor
```

```
            9'b111_100101: ALUCtr = 4'b0001; // or
            9'b111_100110: ALUCtr = 4'b0100; // xor
            9'b111_101010: ALUCtr = 4'b0111; // slt
            9'b111_101011: ALUCtr = 4'b1101; // sltu
            9'b111_000000: ALUCtr = 4'b1000; // sll
            9'b111_000010: ALUCtr = 4'b1001; // srl
            9'b111_000011: ALUCtr = 4'b1010; // sra
            9'b111_000100: ALUCtr = 4'b1000; // sllv
            9'b111_000110: ALUCtr = 4'b1001; // srlv
            9'b111_000111: ALUCtr = 4'b1010; // srav
            default:ALUCtr = 4'b0010;
            endcase
    endmodule
```

5、ALU（算术逻辑运算单元）

算术逻辑运算单元，根据控制信号的不同，选择对输入操作数进行不同的算术或逻辑运算，得到结果。算术功能包括加，减和移位，输出包含结果与符号位；逻辑功能包括或，与，小于设置和异或，输出包含结果与标志位。ALU 模块设计如下：

input	input1[31:0]
input	input2[31:0]
input	lui
output	alures[31:0]
output	zero




```
module alu(
    input [31:0] input1,
    input [31:0] input2,
    input [3:0] aluCtr,
    input lui,
    output reg[31:0] aluRes,
    output reg zero
);
    always @(input1 or input2 or aluCtr) // 运算数或控制码变化时操作
    begin
        case(aluCtr)
        4'b0110: // 减
        begin
            aluRes = input1 - input2;
            if(aluRes == 0)
            zero = 1;
            else
            zero = 0;
        end
        4'b0010: // 加
        begin
```

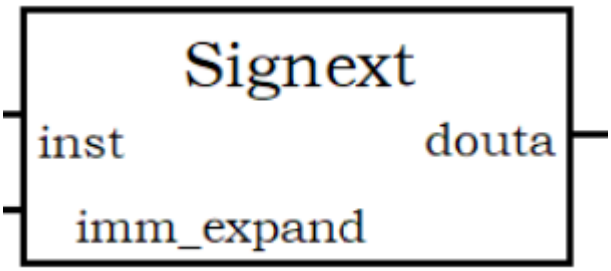
```
    if(lui==1) aluRes = input1 + (input2<<16);
    else aluRes = input1 + input2;
end
4'b0000: // 与
aluRes = input1 & input2;
4'b0001: // 或
aluRes = input1 | input2;
4'b0101: // 或非
aluRes = ~(input1 | input2);
4'b0100: // 异或
aluRes = input1 ^ input2;
4'b1000: // 左移
aluRes = input1<<input2;
4'b1001: // 右移
aluRes = input1>>input2;
4'b1010: // 右移
aluRes = $signed(input1)>>>input2;
4'b1101://sltu
begin
    if(input1<input2)
    aluRes = 1;
    else aluRes = 0;
end
4'b0111: // 小于设置
begin
    if((input1  <  input2  &&  ~(input1[31]^input2[31]))  ||
(input1[31] & ~input2[31]))
    aluRes = 1;
    else aluRes = 0;
end
```

```
            default:
            aluRes = 0;
            endcase
        end
    endmodule
```

6、Signext（符号扩展）

根据拓展选择信号，选择拓展方式：符号位拓展或零拓展，对 16 位立即数进行拓展，输出 32 位数。其中，拓展选择信号为 0 时，进行零拓展，为 1 时，进行符号位拓展。符号扩展模块设计如下：

input	inst[15:0]
input	imm_expand
output	data[31:0]

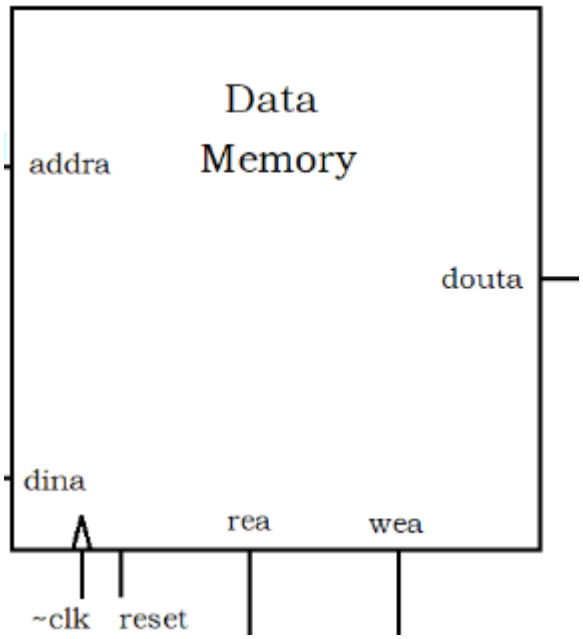



```
module signext(
    input [15:0] inst, // 输入16位
    input imm_expand,
    output [31:0] data // 输出32位
);
    // 根据符号补充符号位
    // 如果符号位为1，则补充16个1，即16'h ffff
    // 如果符号位为0，则补充16个0
    assign data = (imm_expand == 1'b1)?{16'h0000,inst}
            :(inst[15:15]?{16'hffff,inst}:{16'h0000,inst});
endmodule
```

7、Data Memory（数据存储器）

RAM模块作数据存储器，主要在指令sw、lw中使用，将数值存进相应地址的内存中或根据取值地址将内存数据取出。采用 8 位一字节模拟内存，当读使能信号为 1 时，读取数据；当写使能信号为 1 且时钟到达下降沿时，写入数据。数据存储器的设计如下：

input	clk
input	reset
input	addra
input	dina
input	rea
input	wea
output	douta




```
module dram (
    input clk,     //时钟输入
    input wea,     //写数据使能输入
    input rea,     //读数据使能输入
    input reset,   //置位输入
    input [7:0] addra,    //数据内存地址输入
    input [31:0] dina,    //数据输入
    output [31:0] douta   //数据输出
```

```
);
    reg [7:0] RAM[255:0];
    //读数据
    //assign douta={{RAM[addra+3]},{RAM[addra+2]},
            {RAM[addra+1]},{RAM[addra+0]}};
    assign douta[7:0] = (rea)?RAM[addra]:8'bz;
    assign douta[15:8] = (rea)?RAM[addra+1]:8'bz;
    assign douta[23:16] = (rea)?RAM[addra+2]:8'bz;
    assign douta[31:24] = (rea)?RAM[addra+3]:8'bz;
    integer i;
    always @ (posedge clk,posedge reset)
    begin
        //初始化内存
        if(reset)begin
            for(i = 0; i < 256; i = i + 1)
                RAM[i]=0;
        end
        //写数据
        else if (we) begin
        {{RAM[addra+3]},{RAM[addra+2]},
        {RAM[addra+1]},{RAM[addra+0]}}=dina;
        end
    end
endmodule
```

8、top（顶层模块）

本模块的作用是CPU封装及与各个模块连接。

- 到程序ROM中取指令
- 对PC值进行＋4处理

- 完成各种跳转指令的PC修改功能
- 在有中断的情况下处理中断到来时的PC修改

顶层模块需要将前面的多个模块实例化后，通过导线以及多路复用器将各个部件链接起来，并且在时钟的控制下修改PC的值，PC是一个32位的寄存器，每个时钟沿自动增加4。多路复用器MUX直接通过三目运算符实现：Assign $OUT=SE$?$INPUT1$：$INPUT2$;其中，OUT，SEL，INPUT1，INPUT2都是预先定义的信号。顶层模块需要输入时钟和复位信号，然后首先读取rom的机器指令，然后通过各个模块执行。

```
module top(
    input clkin,
    input reset,
    input sel,
    output reg [31:0] pc,
    output [31:0] inst,
    output [2:0] aluop,
    output [3:0] aluCtr,
    output [31:0] aluRes,
    output reg_dst,
    output jmp,
    output branch,
    output memread,
    output memwrite,
    output memtoreg,
    output jal,
    output jr,
    output lui,
    output bne,
    output bltz,
    output bgtz,
    output imm_expand,
    output alu_srcA,
```

```
    output regwrite,
    output [1:0] flag,
    output [1:0] alu_srcB,
    output zero,
    output [31:0] memreaddata,
    output [31:0] RsData,
    output [31:0] RtData,
    output [31:0] expand,
    output [31:0] expand2,
    output [31:0] address,
    output [31:0] jmpaddr,
    output [4:0] mux1,
    output [31:0] mux2,
    output [31:0] mux3,
    output [31:0] mux4,
    output [31:0] mux5,
    output [31:0] mux6,
    output [4:0] mux7,
    output [31:0] mux8,
    output [6:0] seg,
    output [3:0] sm_wei
);

    // 指令寄存器pc
    reg [31:0] add4;
    wire choose4;
    always @(negedge clkin) // 时钟下降沿操作
    begin
        if(!reset)
            begin
```

```
            pc = mux5; // 计算下一条pc，修改pc
            #10
            add4 = pc + 32'b100;
        end
    else
        begin
            pc = 32'b0; // 复位时pc写0
            #10
            add4 = 32'h4;
        end
end

integer clk_cnt;
reg clk;                //1.33秒一个时钟周期，以此为cpu的时钟周期
10^8/0.75*10^8=4/3=1.33
always @(posedge clkin)
if(clk_cnt==32'd75_000_000)
begin
clk_cnt <= 1'b0;
clk <= ~clk;
end
else
clk_cnt <= clk_cnt + 1'b1;
// 实例化控制器模块
ctr mainctr(
.opCode(inst[31:26]),
.funct(inst[5:0]),
.regDst(reg_dst),
.aluSrcA(alu_srcA),
.aluSrcB(alu_srcB),
```

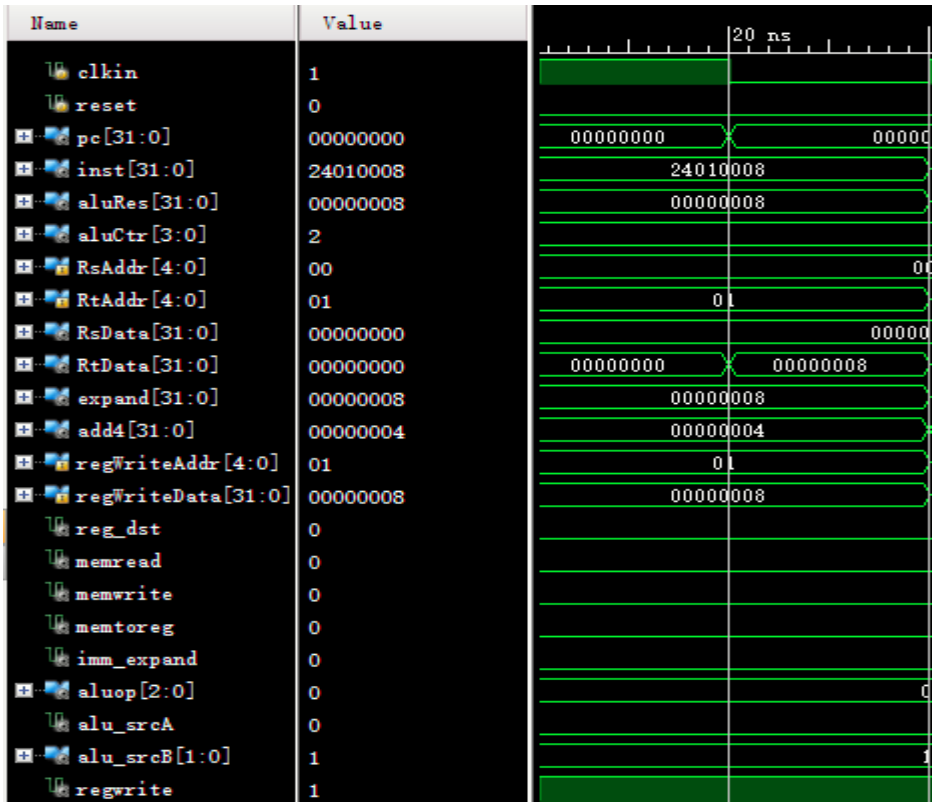
```
.memToReg(memtoreg),
.regWrite(regwrite),
.memRead(memread),
.memWrite(memwrite),
.branch(branch),
.aluop(aluop),
.jmp(jmp),
.jal(jal),
.jr(jr),
.lui(lui),
.bne(bne),
.bltz(bltz),
.bgtz(bgtz),
.imm_expand(imm_expand)
);
// 实例化ALU模块
alu alu(.input1(mux6),
.input2(mux2),
.aluCtr(aluCtr),
.zero(zero),
.lui(lui),
.aluRes(aluRes));
// 实例化ALU控制模块
aluctr aluctr1(
.ALUOp(aluop),
.funct(inst[5:0]),
.ALUCtr(aluCtr));
// 实例化ram模块
dram dmem(
.clk(~clkin),
```

```
.wea(memwrite),
.rea(memread),
.reset(reset),
.addra(aluRes[7:0]),
.dina(RtData),
.douta(memreaddata)
);
// 实例化rom模块
irom imem (
.clka(clkin),    // input wire clka
.addra(pc[9:2]),  // input wire [7 : 0] addra
.douta(inst)  // output wire [31 : 0] douta
);
// 实例化寄存器模块
regFile regfile(
.RsAddr(inst[25:21]),
.RtAddr(inst[20:16]),
.clk(!clkin),
.reset(reset),
.regWriteAddr(mux7),
.regWriteData(mux8),
.regWriteEn(regwrite),
.RsData(RsData),
.RtData(RtData)
);
mux mux(
.q(RtData),
.w(expand),
.e({27'b000000000000000000000000000,inst[10:6]}),
.r(RsData),
```

```
    .a(alu_srcB),
    .c(mux2)
    );
    choose choose(
    .branch(branch),
    .bne(bne),
    .bltz(bltz),
    .bgtz(bgtz),
    .zero(zero),
    .a(RsData),
    .choose4(choose4)
    );
    // 实例化符号扩展模块
    signext
    signext(.inst(inst[15:0]),.imm_expand(imm_expand), .data(expa
    nd));
    // 各个控制信号线，地址，符号扩展
    assign mux1 = reg_dst ? inst[15:11] : inst[20:16];
    assign mux3 = memtoreg ? memreaddata : aluRes;
    assign mux4 = choose4 ? address : add4;
    assign mux5 = jmp ? jmpaddr : (jr ? RsData:mux4);
    assign mux6 = alu_srcA ? RtData:RsData;
    assign mux7 = mux1 | {5{jal}};
    assign mux8 = (jal==1'b1)? add4: mux3;
    assign expand2 = expand << 2;
    assign jmpaddr = {add4[31:28], inst[25:0], 2'b00};
    assign address = pc + expand2 + 4;
endmodule
```

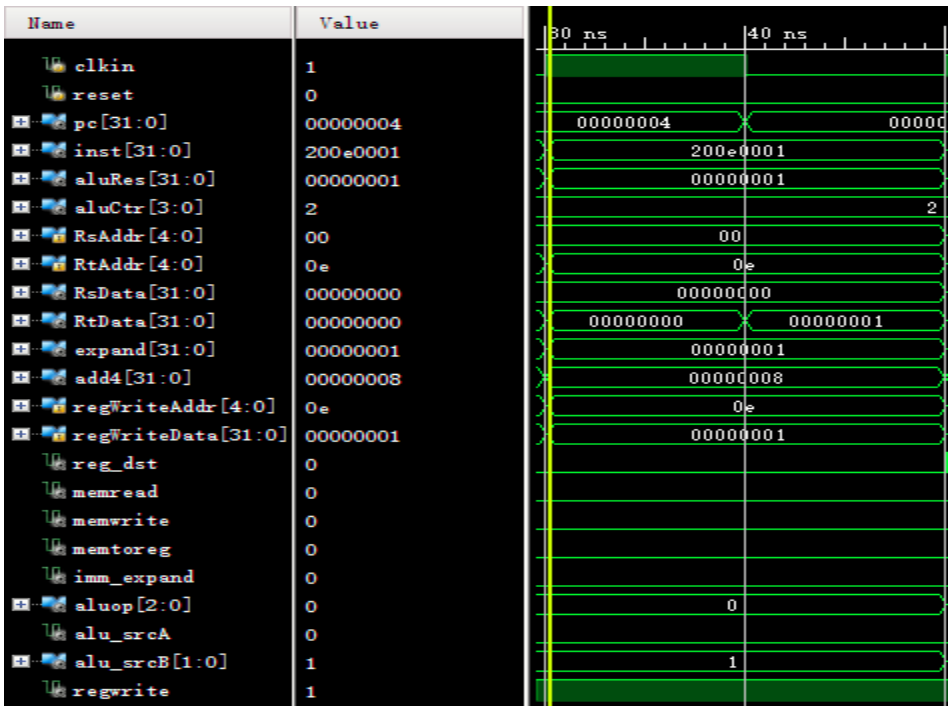
二、仿真验证

1、addiu $1, $0, 8



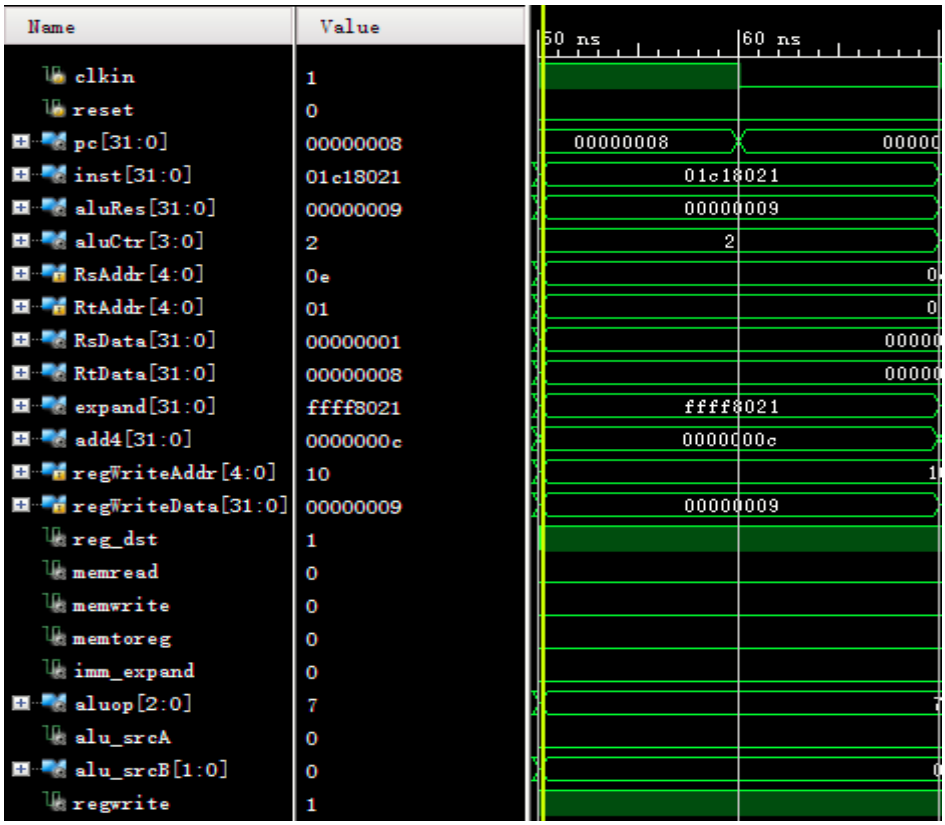
当前指令地址 pc 为 0，下一条 add4 为 4。立即数 expand 为 8。RsAddr 为 0 号寄存器，对应值 RsData 为 0；RtAddr 为 1 号寄存器，对应值 RtData 在时钟下降沿时由 0 变为 8。ALU 运算结果 aluRes 为 8。写入寄存器 regWriteAddr 为 1 号寄存器，写入数据 regWriteData 为 8。指令执行正确($1 = $0 + 8 = 0 + 8 = 8)。

2、addi $14, $0, 1



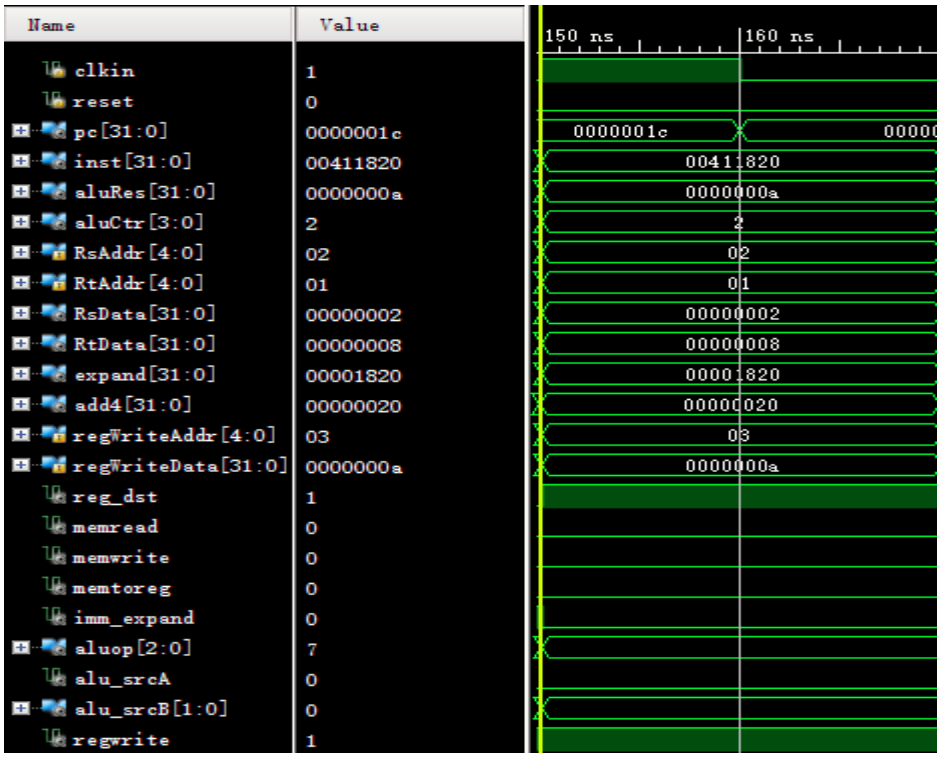
当前指令地址 pc 为 4，下一条 add4 为 8。立即数 expand 为 1。RsAddr 为 0 号寄存器，对应值 RsData 为 0；RtAddr 为 14 号寄存器，对应值 RtData 在时钟下降沿时由 0 变为 1。ALU 运算结果 aluRes 为 1。写入寄存器 regWriteAddr 为 14 号寄存器，写入数据 regWriteData 为 1。指令执行正确($14 = $0 + 1 = 0 + 1 = 1)。

3、addu $16, $14, $1



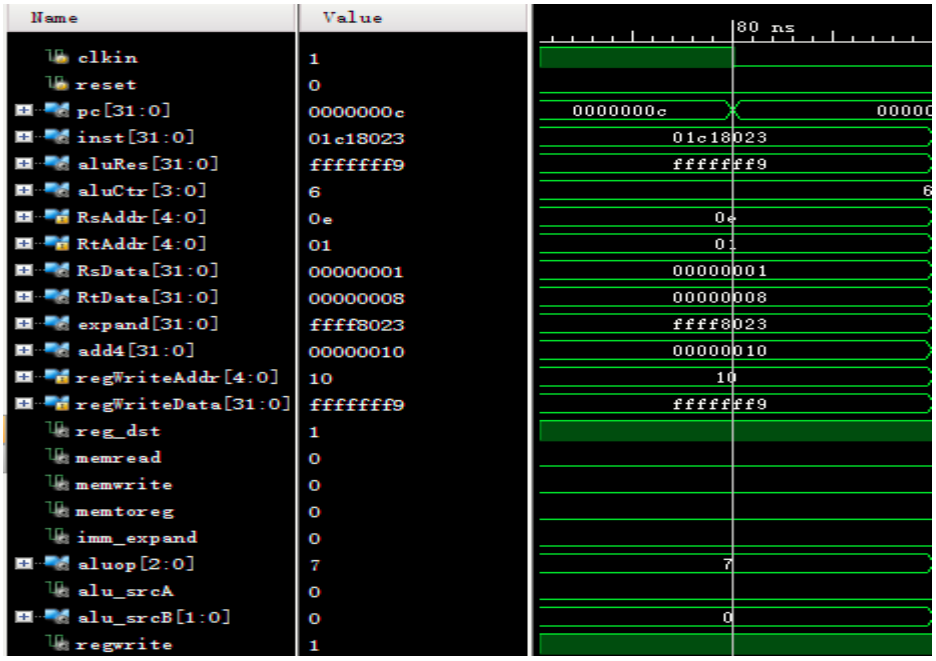
当前指令地址 pc 为 8，下一条 add4 为 12。RsAddr 为 14 号寄存器，对应值 RsData 为 1；RtAddr 为 1 号寄存器，对应值 RtData 为 8。ALU 运算结果 aluRes 为 9。写入寄存器 regWriteAddr 为 16 号寄存器，写入数据 regWriteData 为 9。指令执行正确($16 = $14 + $1 = 1 + 8 = 9)。

4、add $3, $2, $1



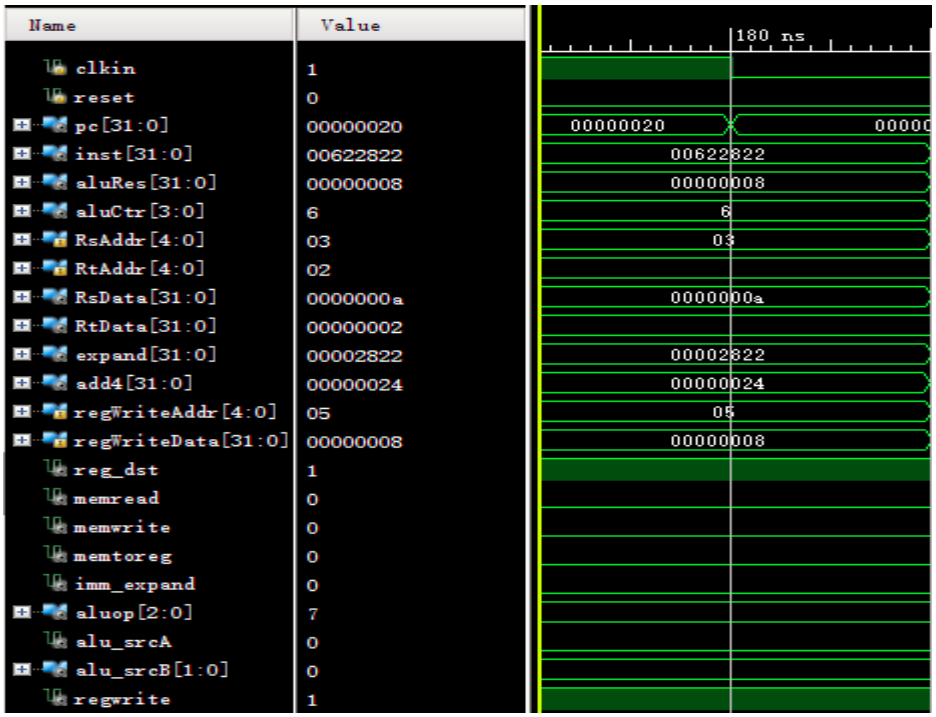
当前指令地址 pc 为 28，下一条 add4 为 32。RsAddr 为 2 号寄存器，对应值 RsData 为 2；RtAddr 为 1 号寄存器，对应值 RtData 为 8。ALU 运算结果 aluRes 为 10。写入寄存器 regWriteAddr 为 3 号寄存器，写入数据 regWriteData 为 10。指令执行正确($3 = $2 + $1 = 2 + 8 = 10)。

5、subu $16, $14, $1



当前指令地址 pc 为 12，下一条 add4 为 16。RsAddr 为 14 号寄存器，对应值 RsData 为 1；RtAddr 为 1 号寄存器，对应值 RtData 为 8。ALU 运算结果 aluRes 为 -7。写入寄存器 regWriteAddr 为 16 号寄存器，写入数据 regWriteData 为 -7。指令执行正确($16 = $14 - $1 = 1 - 8 = -7)。

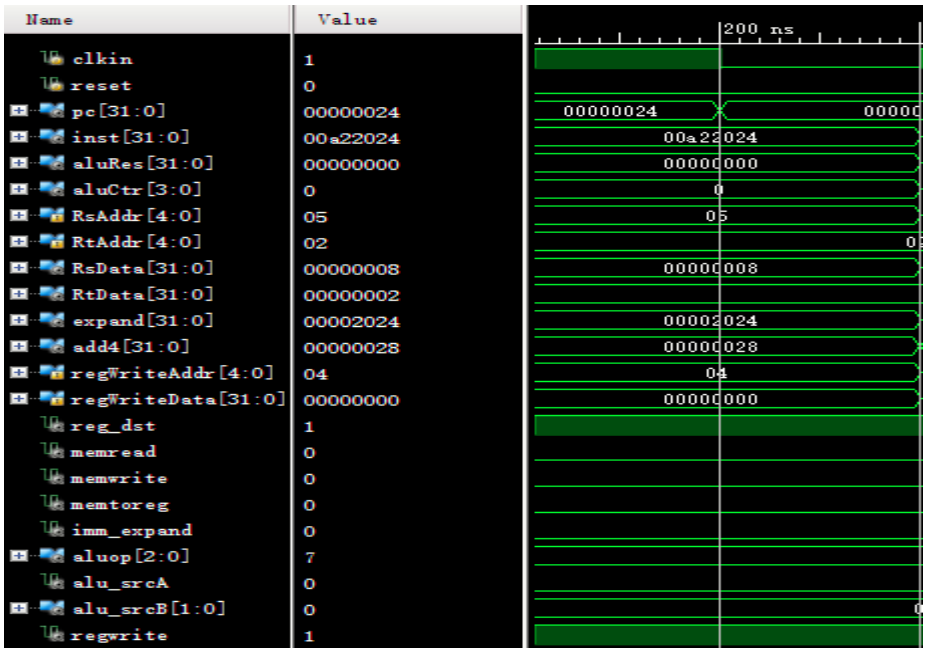
6、sub $5, $3, $2



当前指令地址 pc 为 32，下一条 add4 为 36。RsAddr 为 3 号寄存器，对应值 RsData 为 10；RtAddr 为 2 号寄存器，对应值 RtData 为 2。ALU 运算结果 aluRes

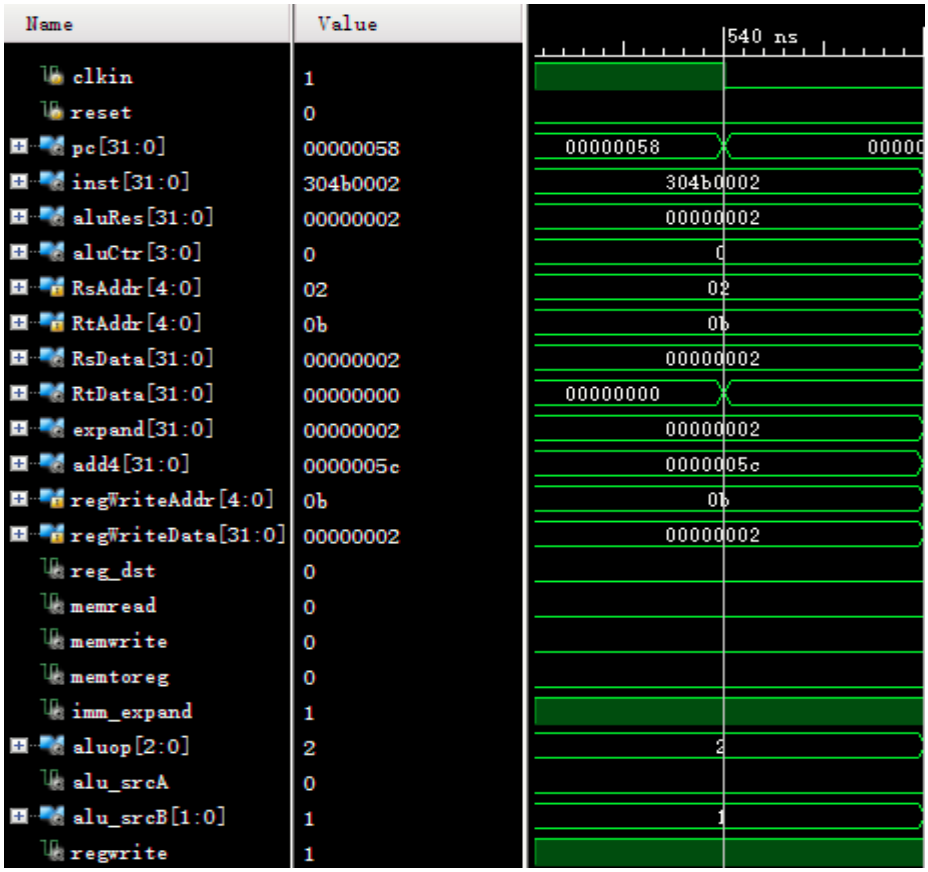
为 8。写入寄存器 regWriteAddr 为 5 号寄存器，写入数据 regWriteData 为 8。指令执行正确($5 = $3 - $2 = 10 - 2 = 8)。

7、and $4, $5, $2



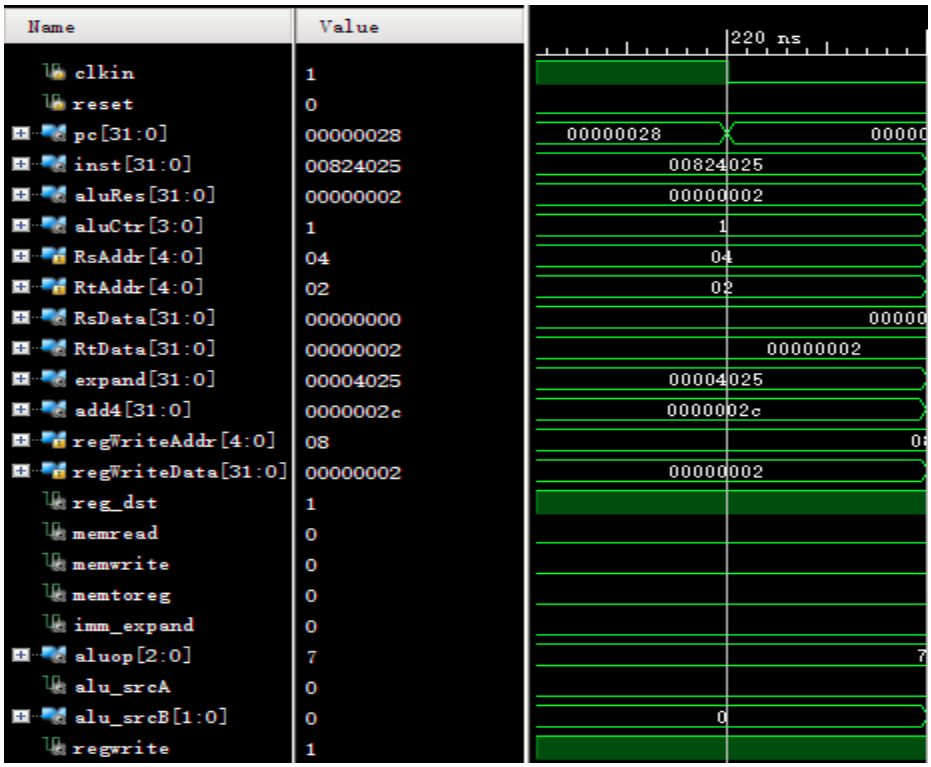
当前指令地址 pc 为 36，下一条 add4 为 40。RsAddr 为 5 号寄存器，对应值 RsData 为 8；RtAddr 为 2 号寄存器，对应值 RtData 为 2。ALU 运算结果 aluRes 为 0。写入寄存器 regWriteAddr 为 4 号寄存器，写入数据 regWriteData 为 0。指令执行正确($4 = $5 and $2 = 8 and 2 = 0)。

8、andi $11, $2, 2



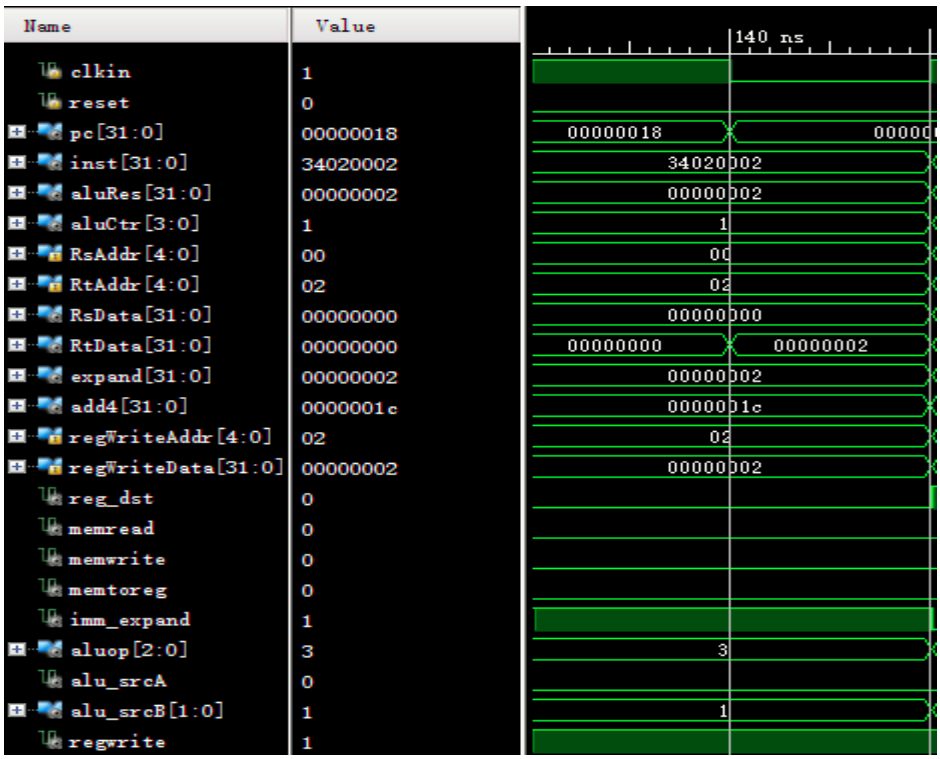
当前指令地址 pc 为 88，下一条 add4 为 92。立即数 expand 为 2。RsAddr 为 2 号寄存器，对应值 RsData 为 2；RtAddr 为 11 号寄存器，对应值 RtData 为 0。ALU 运算结果 aluRes 为 2。写入寄存器 regWriteAddr 为 11 号寄存器，写入数据 regWriteData 为 2。指令执行正确($11 = $2 and 2 = 2 and 2 = 2)。

9、or $8, $4, $2



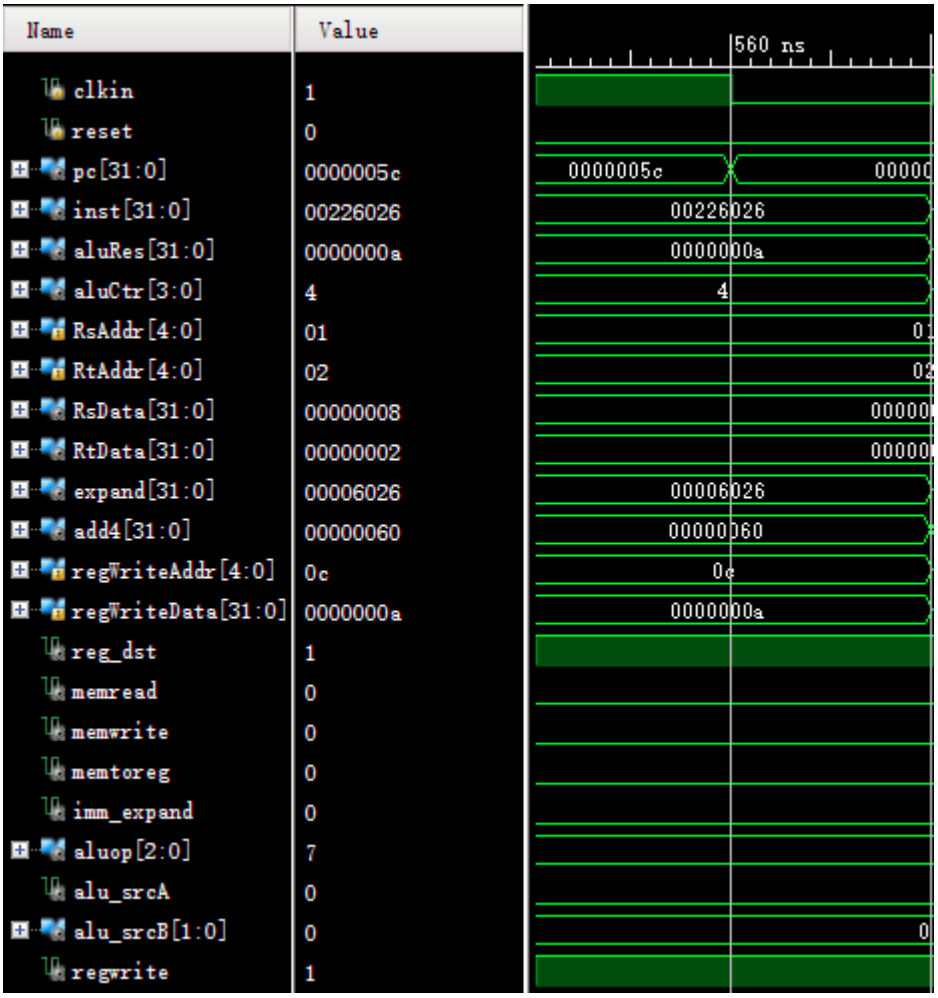
当前指令地址 pc 为 40，下一条 add4 为 44。RsAddr 为 4 号寄存器，对应值 RsData 为 0；RtAddr 为 2 号寄存器，对应值 RtData 为 2。ALU 运算结果 aluRes 为 2。写入寄存器 regWriteAddr 为 8 号寄存器，写入数据 regWriteData 为 2。指令执行正确($8 = $4 or $2 = 0 or 2 = 2)。

10、ori $2, $0, 2



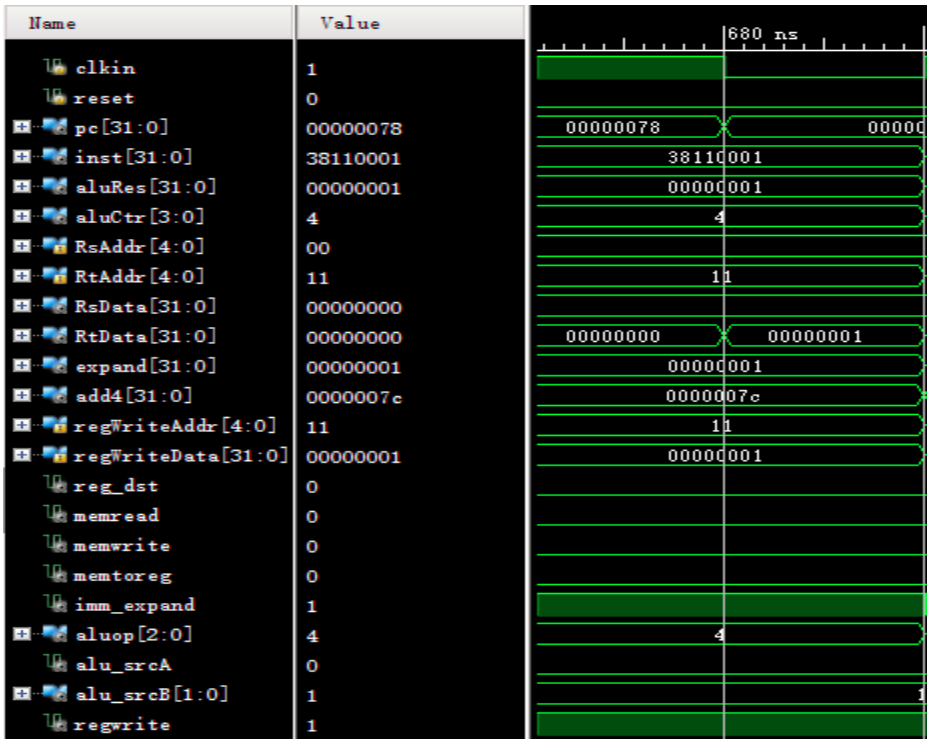
当前指令地址 pc 为 24，下一条 add4 为 28。立即数 expand 为 2。RsAddr 为 0 号寄存器，对应值 RsData 为 0；RtAddr 为 2 号寄存器，对应值 RtData 为 0。ALU 运算结果 aluRes 为 2。写入寄存器 regWriteAddr 为 2 号寄存器，写入数据 regWriteData 为 2。指令执行正确($2 = $0 or 2 = 0 or 2 = 2)。

11、xor $12, $1, $2



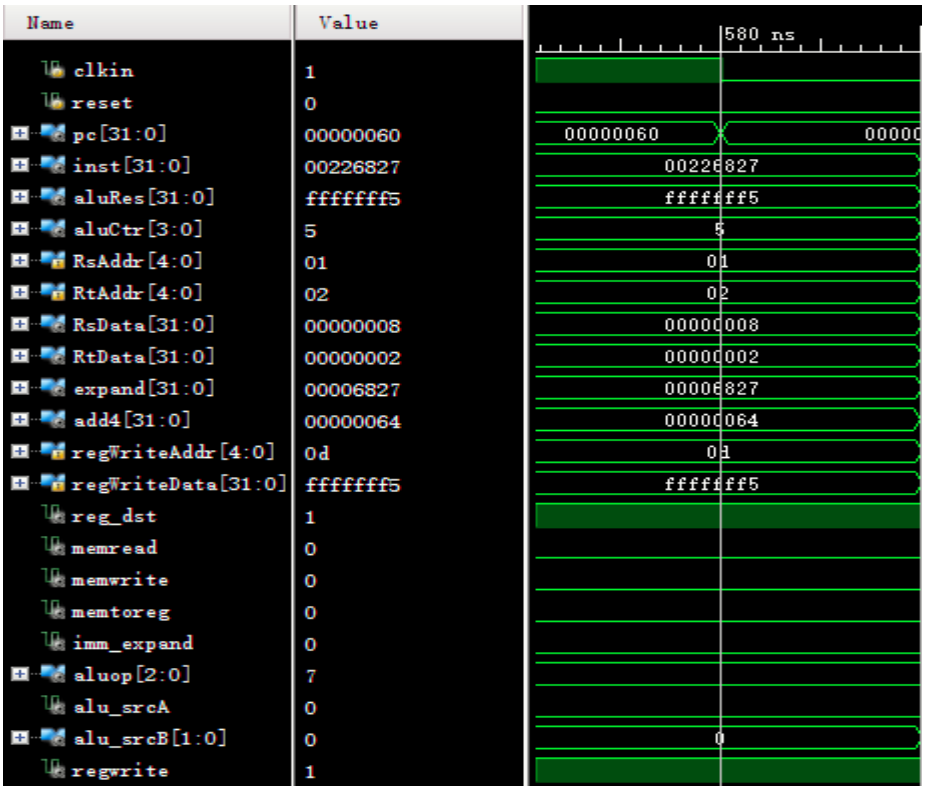
当前指令地址 pc 为 92，下一条 add4 为 96。RsAddr 为 1 号寄存器，对应值 RsData 为 8；RtAddr 为 2 号寄存器，对应值 RtData 为 2。ALU 运算结果 aluRes 为 10。写入寄存器 regWriteAddr 为 12 号寄存器，写入数据 regWriteData 为 10。指令执行正确($12 = $1 xor $2 = 8 xor 2 = 10)。

12、xori $17, $0, 1



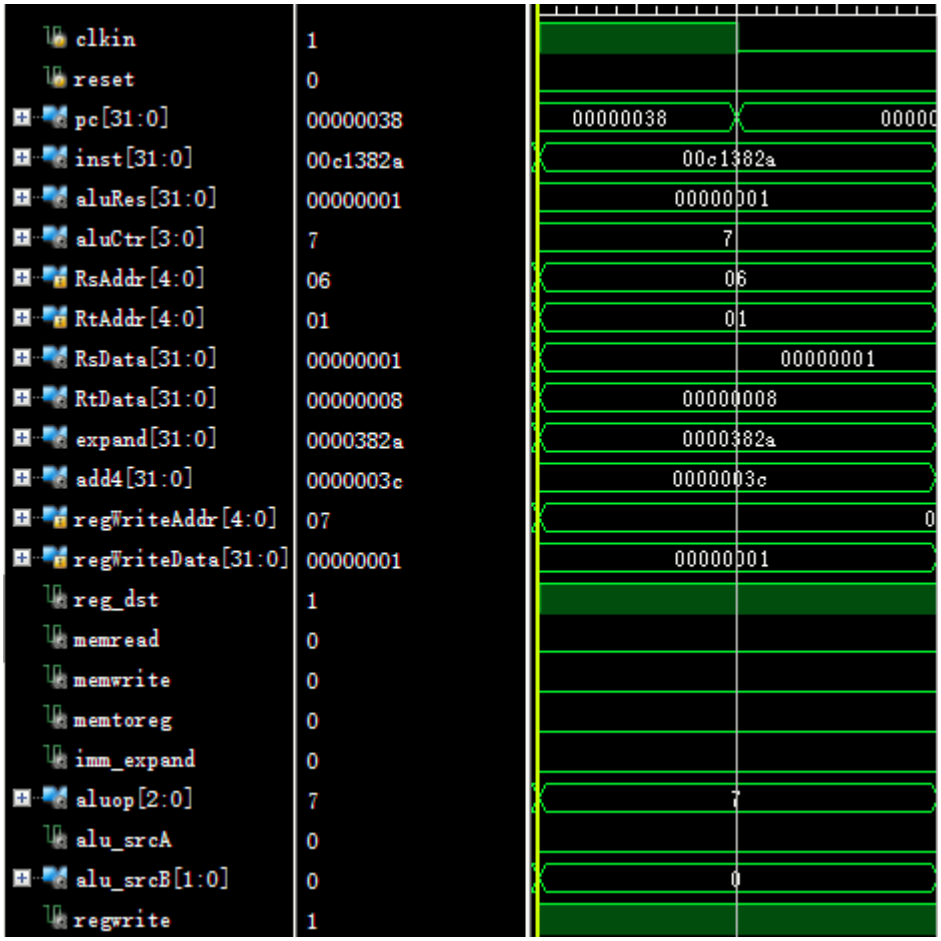
当前指令地址 pc 为 120，下一条 add4 为 124。立即数 expand 为 1。RsAddr 为 0 号寄存器，对应值 RsData 为 0；RtAddr 为 17 号寄存器，对应值 RtData 为 0。ALU 运算结果 aluRes 为 1。写入寄存器 regWriteAddr 为 17 号寄存器，写入数据 regWriteData 为 1。指令执行正确($17 = $0 xor 1 = 0 xor 1 = 1)。

13、nor $13, $1, $2



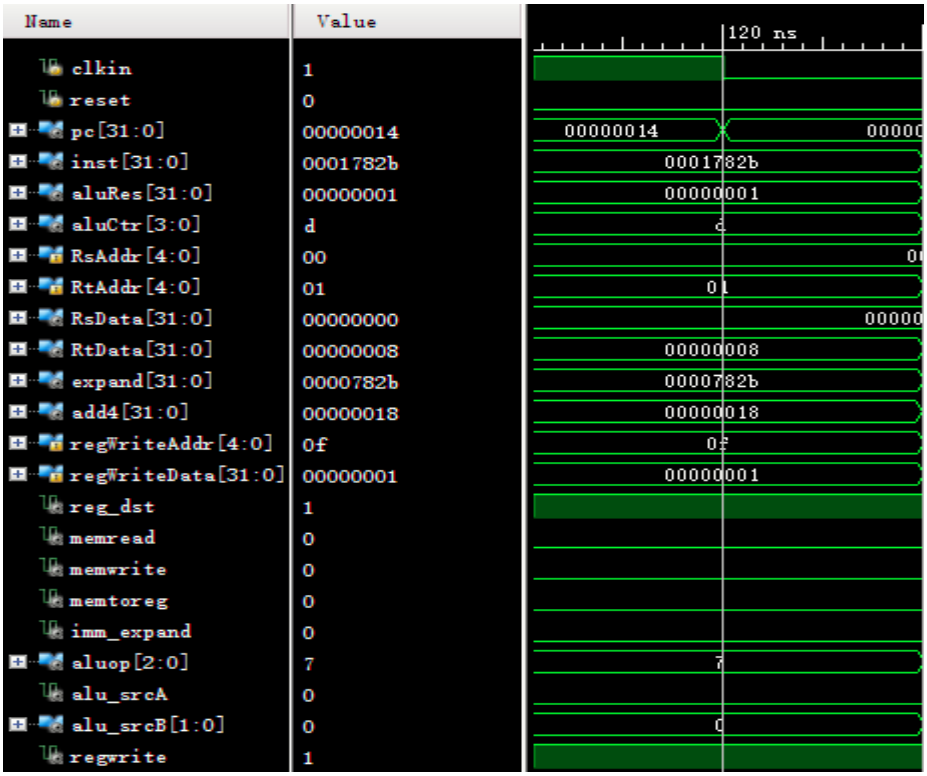
当前指令地址 pc 为 96，下一条 add4 为 100。RsAddr 为 1 号寄存器，对应值 RsData 为 8；RtAddr 为 2 号寄存器，对应值 RtData 为 2。ALU 运算结果 aluRes 为 fffffff5。写入寄存器 regWriteAddr 为 13 号寄存器，写入数据 regWriteData 为 fffffff5。指令执行正确($13 = ~ ($1 or $2) = ~ (8 or 2) = fffffff5)。

14、slt $7, $6, $1



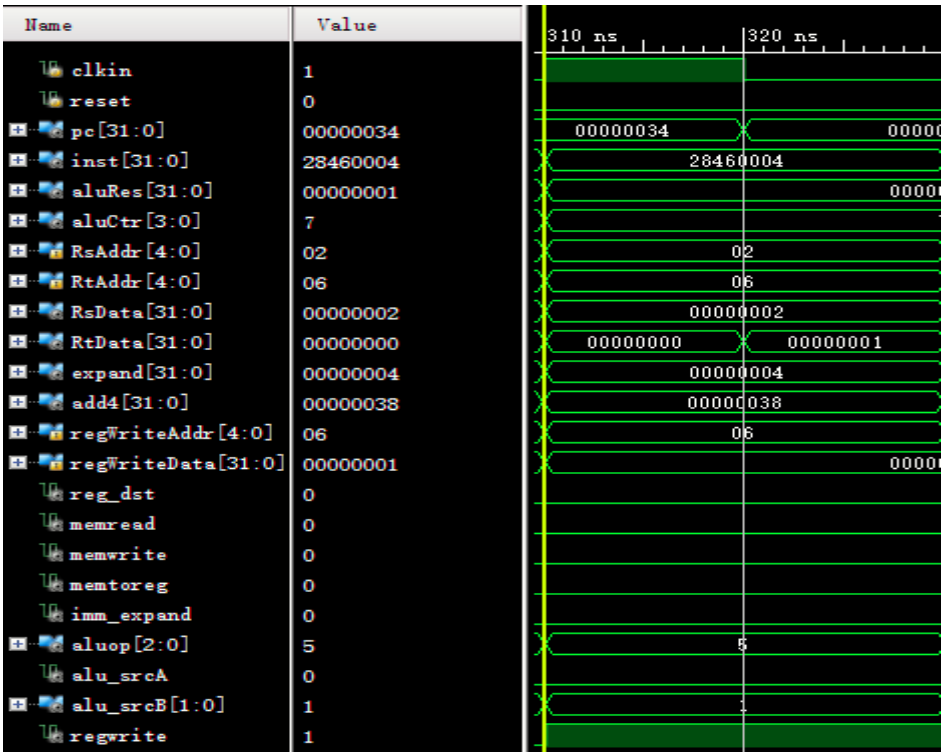
当前指令地址 pc 为 56，下一条 add4 为 60。RsAddr 为 6 号寄存器，对应值 RsData 为 1；RtAddr 为 1 号寄存器，对应值 RtData 为 8。ALU 运算结果 aluRes 为 1。写入寄存器 regWriteAddr 为 7 号寄存器，写入数据 regWriteData 为 1。指令执行正确($7 = $6 < $1 = 1 < 8 = 1)。

15、sltu $15, $0, $1



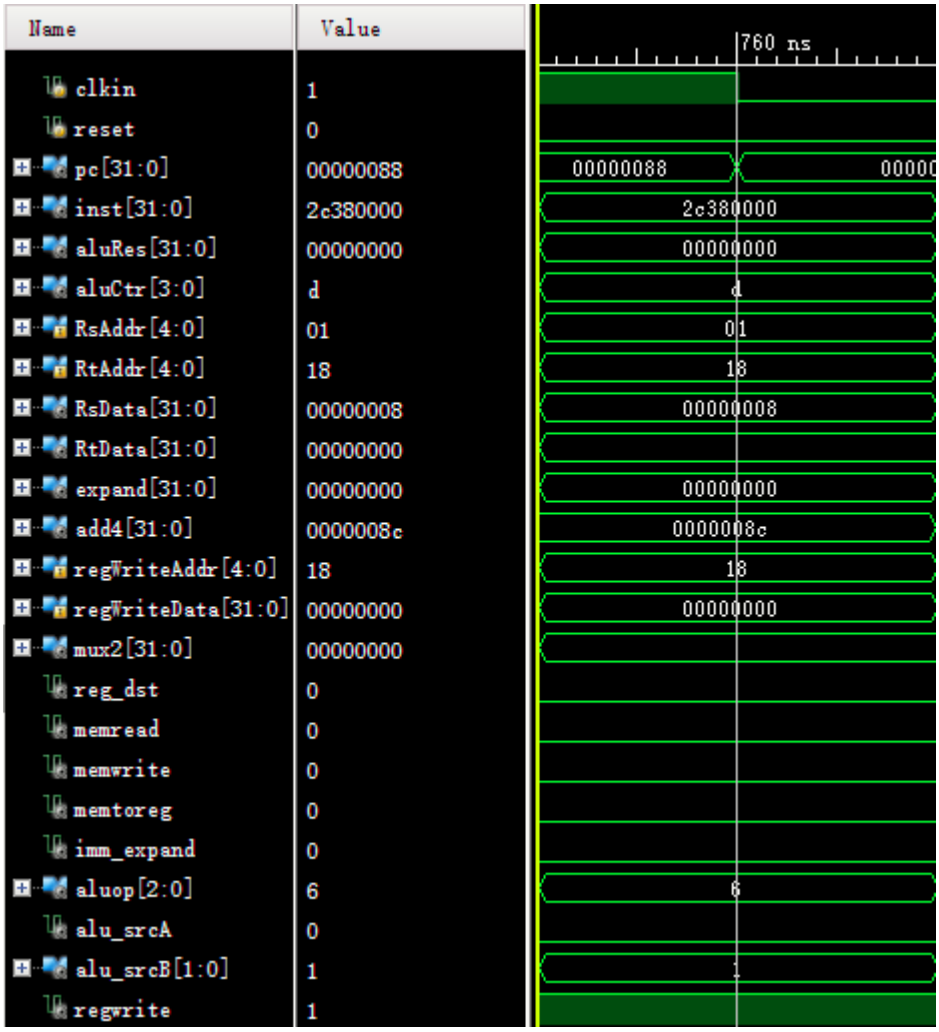
当前指令地址 pc 为 20，下一条 add4 为 24。RsAddr 为 0 号寄存器，对应值 RsData 为 0；RtAddr 为 1 号寄存器，对应值 RtData 为 8。ALU 运算结果 aluRes 为 1。写入寄存器 regWriteAddr 为 15 号寄存器，写入数据 regWriteData 为 1。指令执行正确($15 = $0 < $1 = 0 < 8 = 1)。

16、slti $6, $2, 4



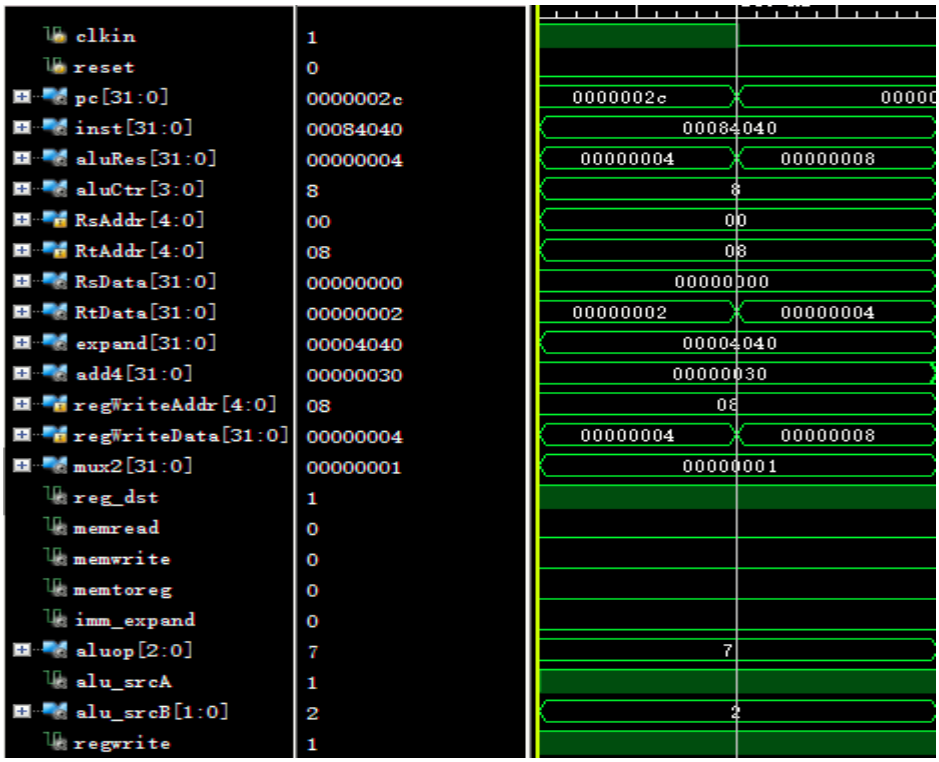
当前指令地址 pc 为 52，下一条 add4 为 56。立即数 expand 为 4。RsAddr 为 2 号寄存器，对应值 RsData 为 2；RtAddr 为 6 号寄存器，对应值 RtData 为 0。ALU 运算结果 aluRes 为 1。写入寄存器 regWriteAddr 为 6 号寄存器，写入数据 regWriteData 为 1。指令执行正确($6 = $2 < 4 = 2 < 4 = 1)。

17、sltiu $24, $1, 0



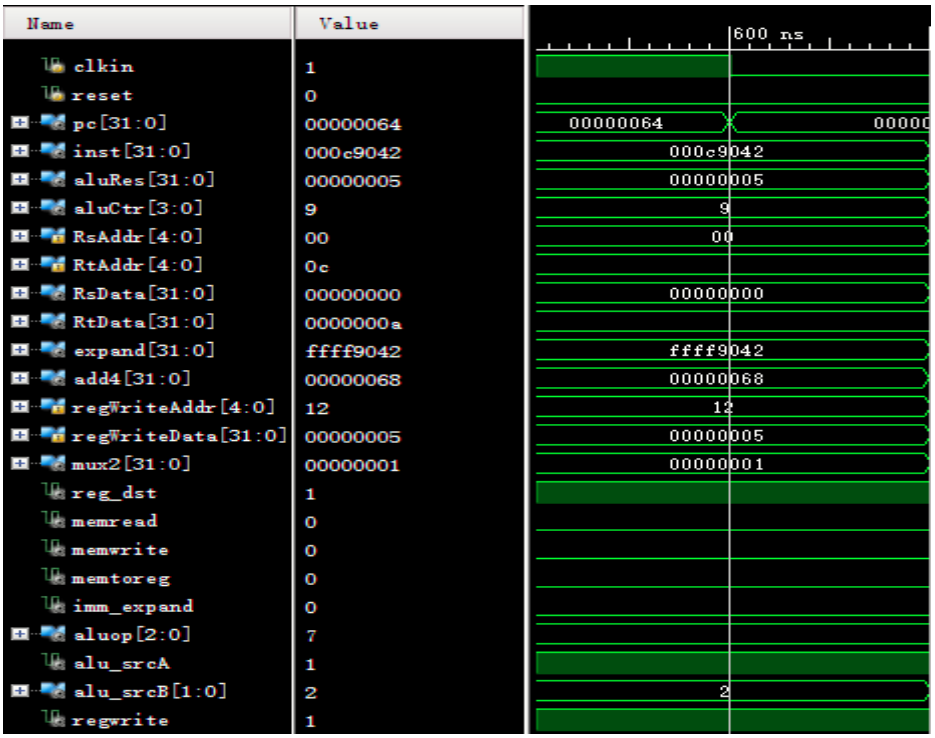
当前指令地址 pc 为 136，下一条 add4 为 140。立即数 expand 为 0。RsAddr 为 1 号寄存器，对应值 RsData 为 8；RtAddr 为 24 号寄存器，对应值 RtData 为 0。ALU 运算结果 aluRes 为 0。写入寄存器 regWriteAddr 为 24 号寄存器，写入数据 regWriteData 为 0。指令执行正确($24 = $1 < 0 = 8 < 0 = 0)。

18、sll $8, $8, 1



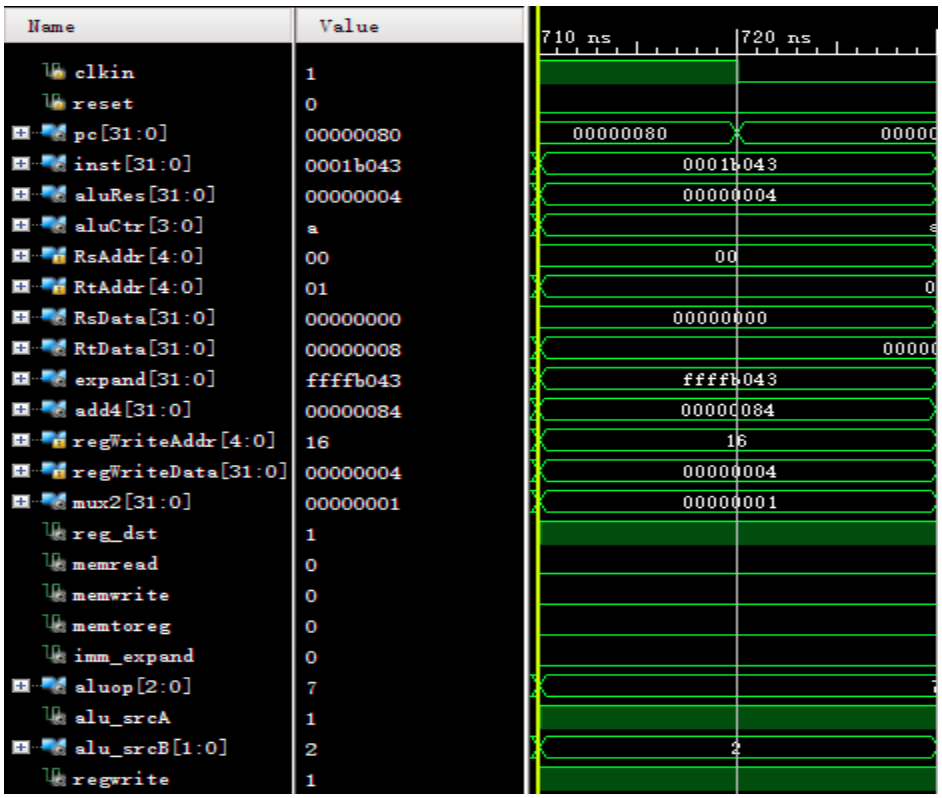
当前指令地址 pc 为 44，下一条 add4 为 48。偏移量 mux2 为 1。RsAddr 为 0 号寄存器，对应值 RsData 为 0；RtAddr 为 8 号寄存器，对应值 RtData 为 2。ALU 运算结果 aluRes 为 4。写入寄存器 regWriteAddr 为 8 号寄存器，写入数据 regWriteData 为 4。指令执行正确($8 = $8 << 1 = 2 << 1 = 4)。

19、srl $18, $12, 1



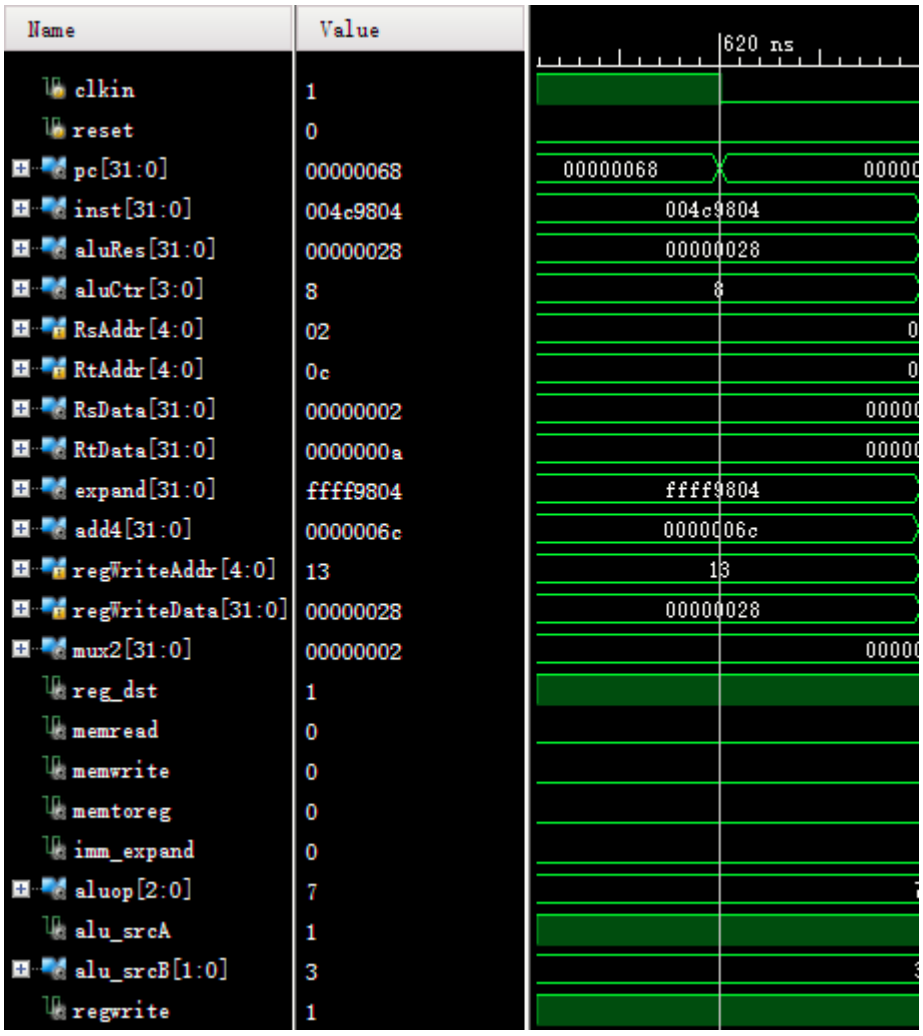
当前指令地址 pc 为 100，下一条 add4 为 104。偏移量 mux2 为 1。RsAddr 为 0 号寄存器，对应值 RsData 为 0；RtAddr 为 12 号寄存器，对应值 RtData 为 10。ALU 运算结果 aluRes 为 5。写入寄存器 regWriteAddr 为 18 号寄存器，写入数据 regWriteData 为 5。指令执行正确($18 = $12 >> 1 = 10 >> 1 = 5)。

20、sra $22, $1, 1



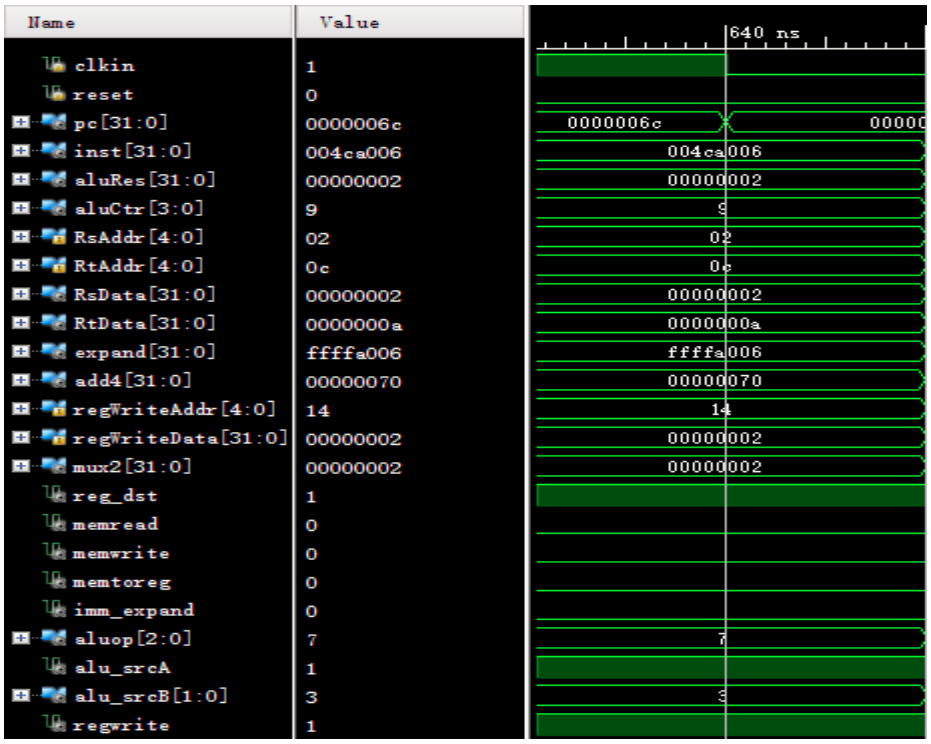
当前指令地址 pc 为 128，下一条 add4 为 132。偏移量 mux2 为 1。RsAddr 为 0 号寄存器，对应值 RsData 为 0；RtAddr 为 1 号寄存器，对应值 RtData 为 8。ALU 运算结果 aluRes 为 4。写入寄存器 regWriteAddr 为 22 号寄存器，写入数据 regWriteData 为 4。指令执行正确($22 = $1 >> 1 = 8 >> 1 = 4)。

21、sllv $19 ,$12, $2



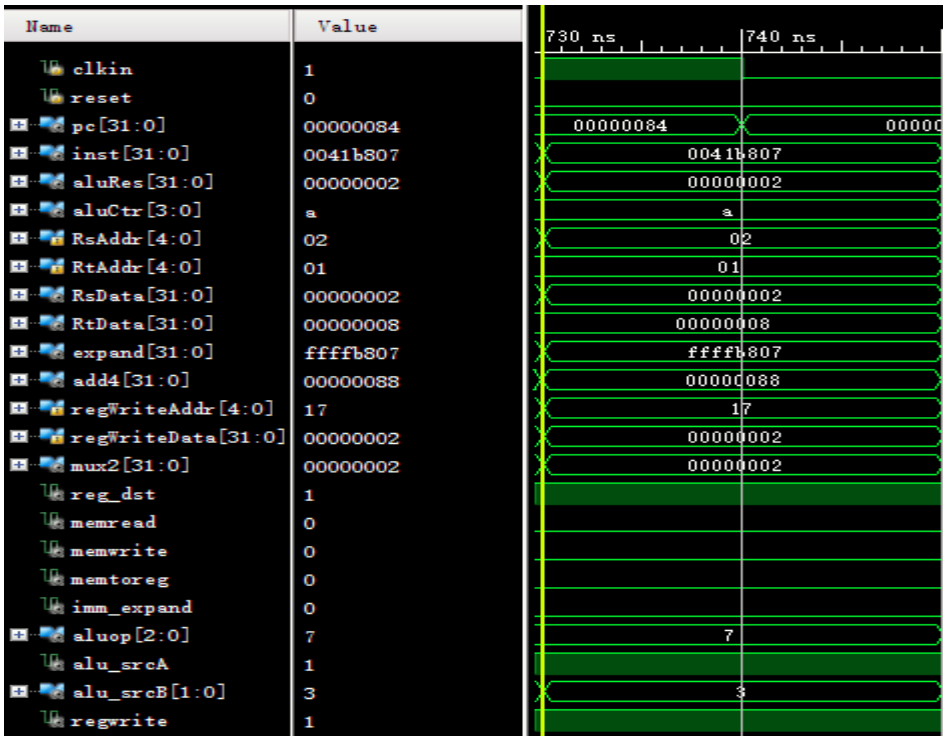
当前指令地址 pc 为 104，下一条 add4 为 108。RsAddr 为 2 号寄存器，对应值 RsData 为 2;RtAddr 为 12 号寄存器,对应值 RtData 为 10。ALU 运算结果 aluRes 为 40。写入寄存器 regWriteAddr 为 19 号寄存器，写入数据 regWriteData 为 40。指令执行正确($19 = $12 << $2 = 10 << 2 = 40)。

22、srlv $20 ,$12, $2



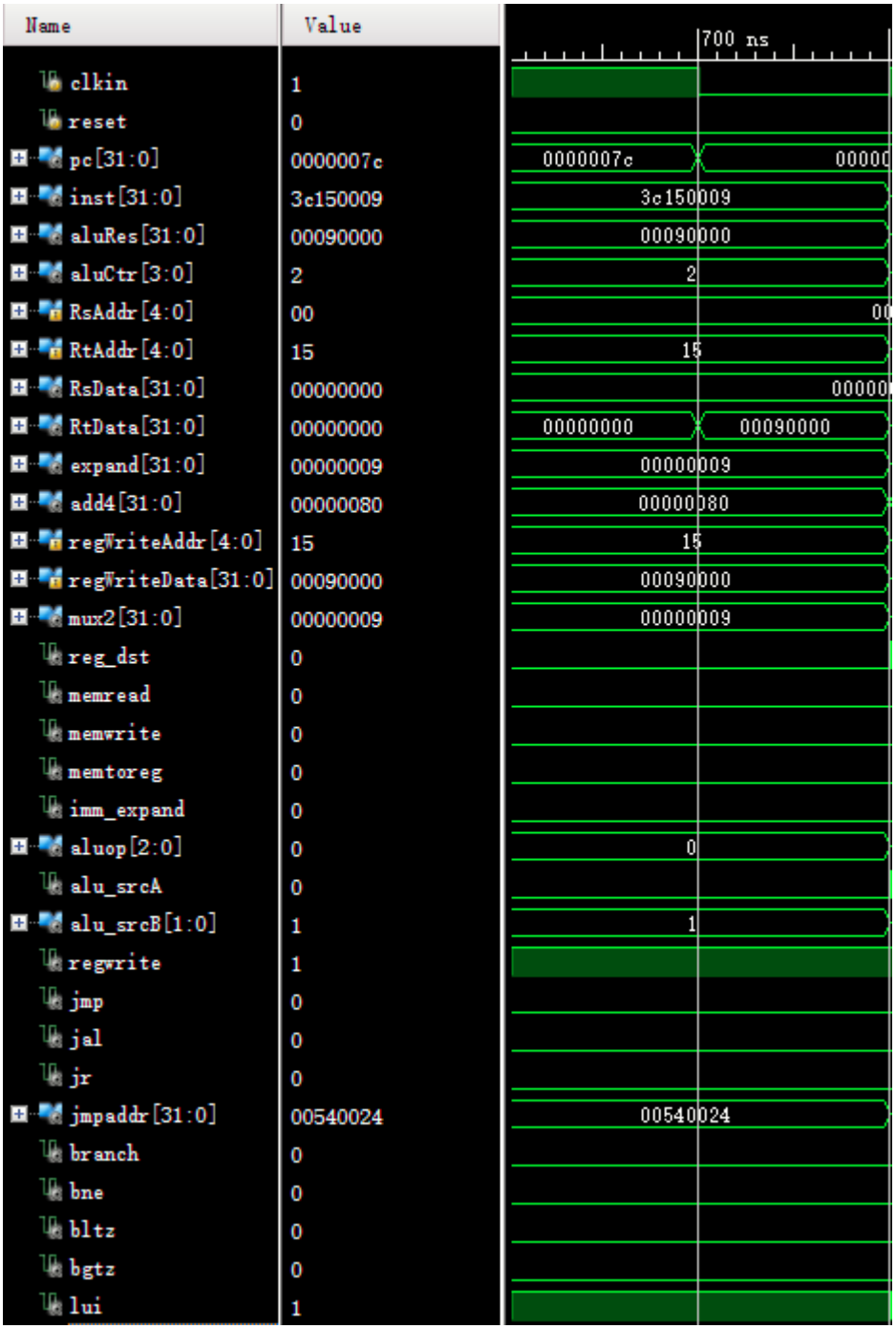
当前指令地址 pc 为 108，下一条 add4 为 112。RsAddr 为 2 号寄存器，对应值 RsData 为 2;RtAddr 为 12 号寄存器,对应值 RtData 为 10。ALU 运算结果 aluRes 为 2。写入寄存器 regWriteAddr 为 20 号寄存器，写入数据 regWriteData 为 2。指令执行正确($20 = $12 >> $2 = 10 >> 2 = 2)。

23、srav $23, $1, $2



当前指令地址 pc 为 132，下一条 add4 为 136。RsAddr 为 2 号寄存器，对应值 RsData 为 2；RtAddr 为 1 号寄存器，对应值 RtData 为 8。ALU 运算结果 aluRes 为 2。写入寄存器 regWriteAddr 为 23 号寄存器，写入数据 regWriteData 为 2。指令执行正确($23 = $1 >> $2 = 8 >> 2 = 2)。

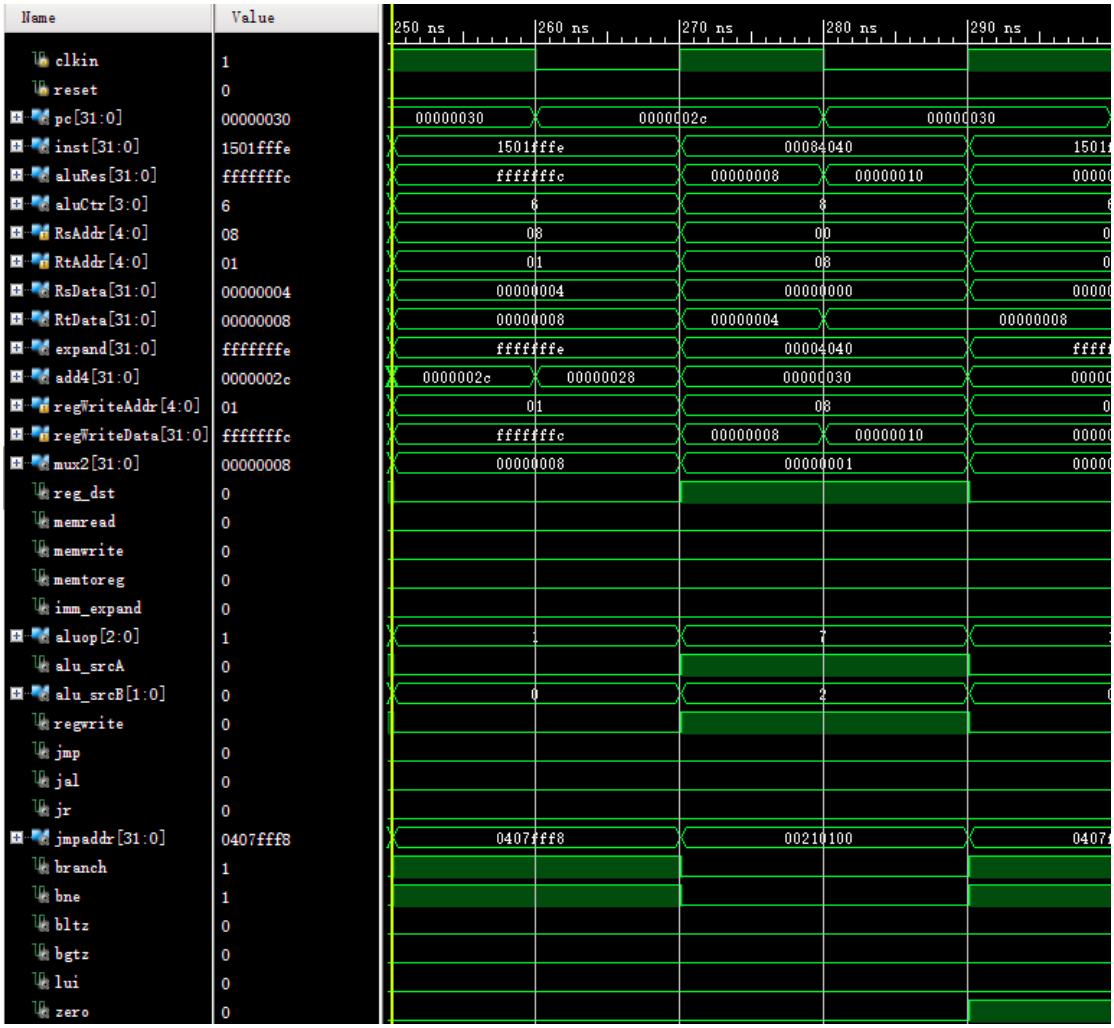
24、lui $21, 9



当前指令地址 pc 为 124，下一条 add4 为 128。控制信号 lui 为1。立即数 expand 为9。RsAddr 为 0 号寄存器，对应值 RsData 为 0；RtAddr 为 21 号寄存

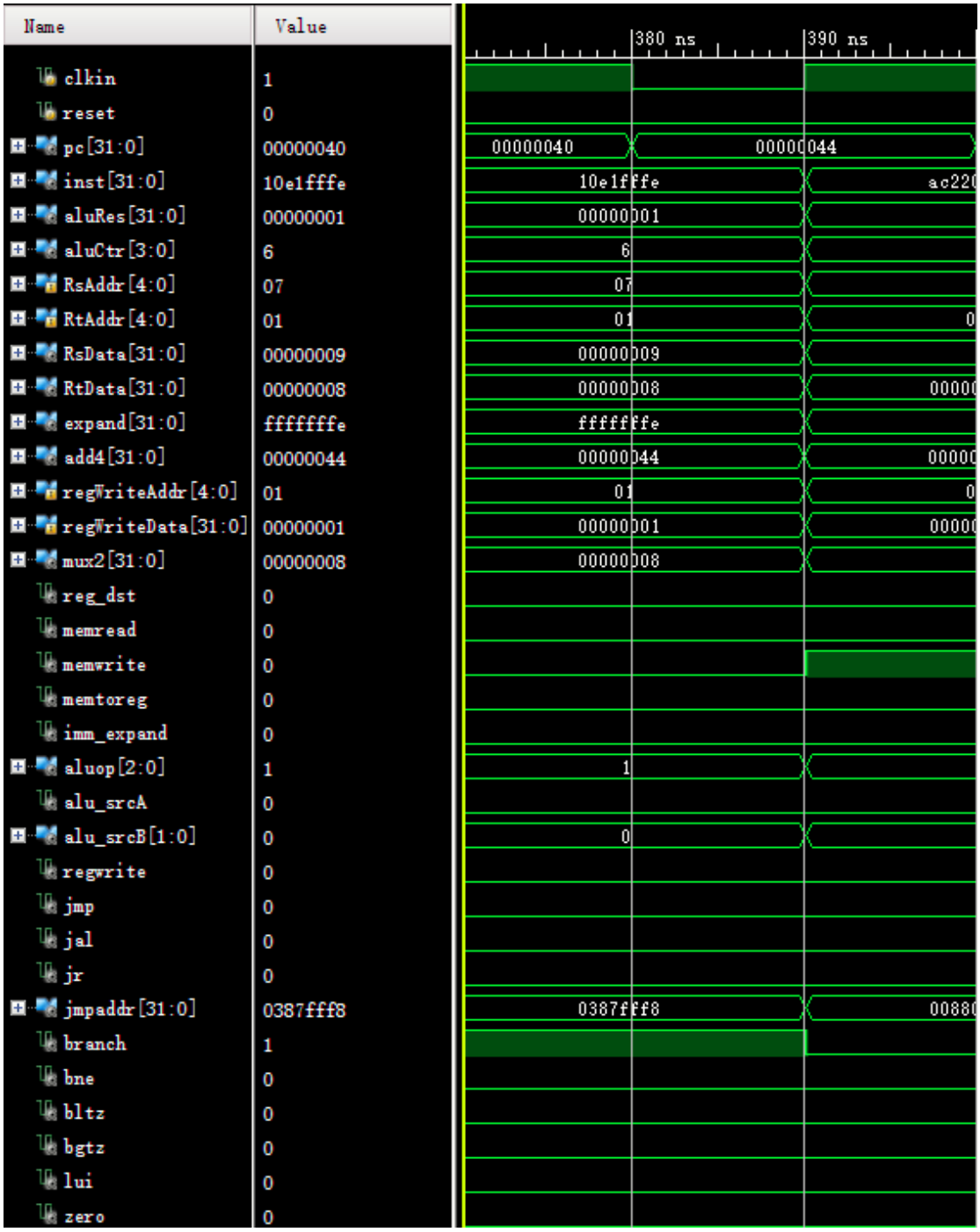
器，对应值 RtData 为 0。ALU 运算结果 aluRes 为 00090000。写入寄存器 regWriteAddr 为 21 号寄存器，写入数据 regWriteData 为 00090000。指令执行正确($21 = 9 << 16 = 00090000)。

25、bne $8, $1, -2



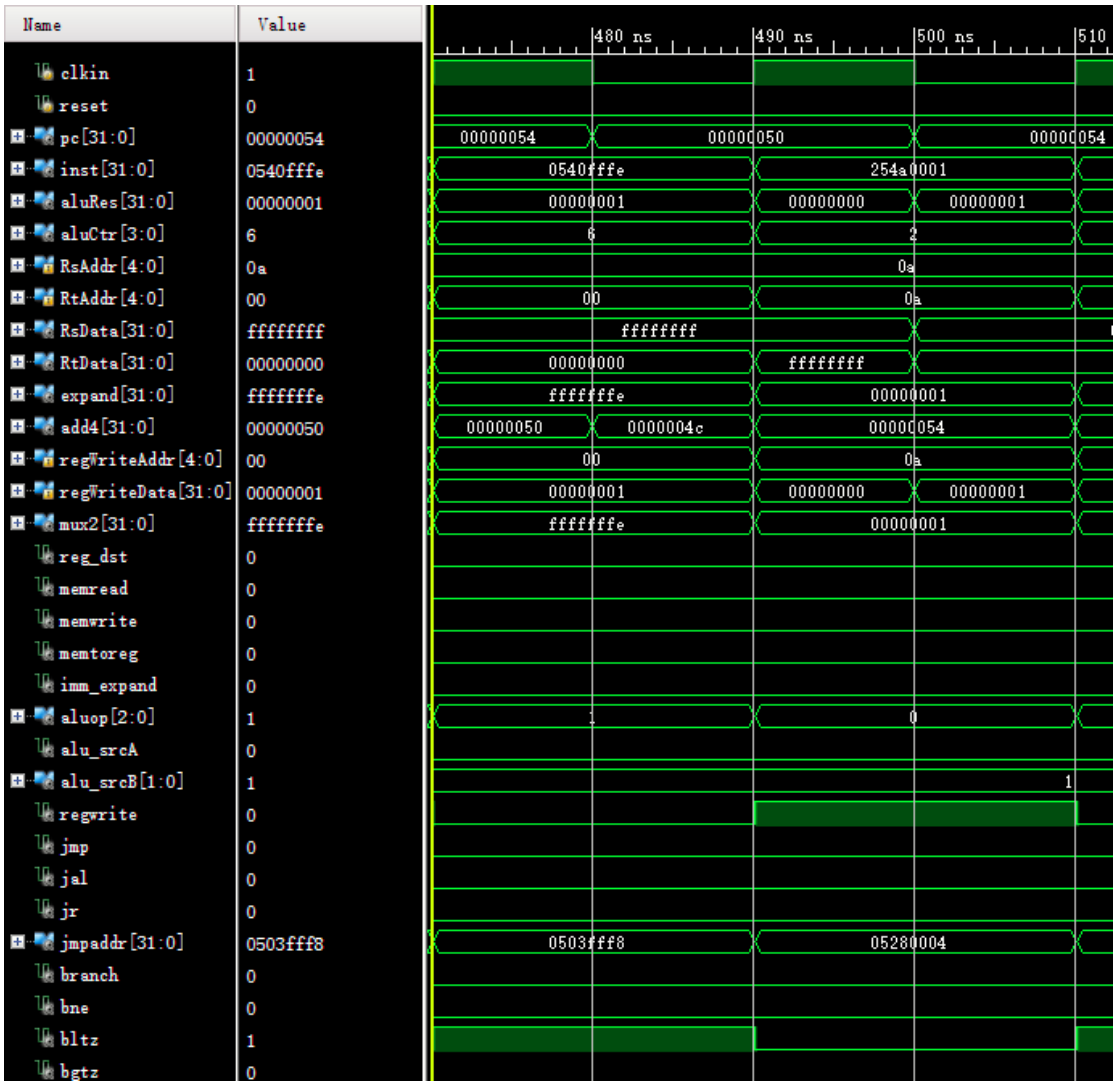
当前指令地址 pc 为 48，下一条 add4 为 52。控制信号 branch 和 bne 均为1。立即数 expand 为 -2。RsAddr 为 8 号寄存器，对应值 RsData 为 4；RtAddr 为 1 号寄存器，对应值 RtData 为 8。ALU 运算结果 aluRes 为 fffffffc，zero 为0，可知此时满足跳转条件，故地址跳回48 + 4 − 2 × 4 = 44。然后再次回到 bne 指令时，可知 zero 为 1 ，不满足跳转条件，故 pc直接加 4 。所以可以知道指令执行正确。

26、beq $7, $1, -2



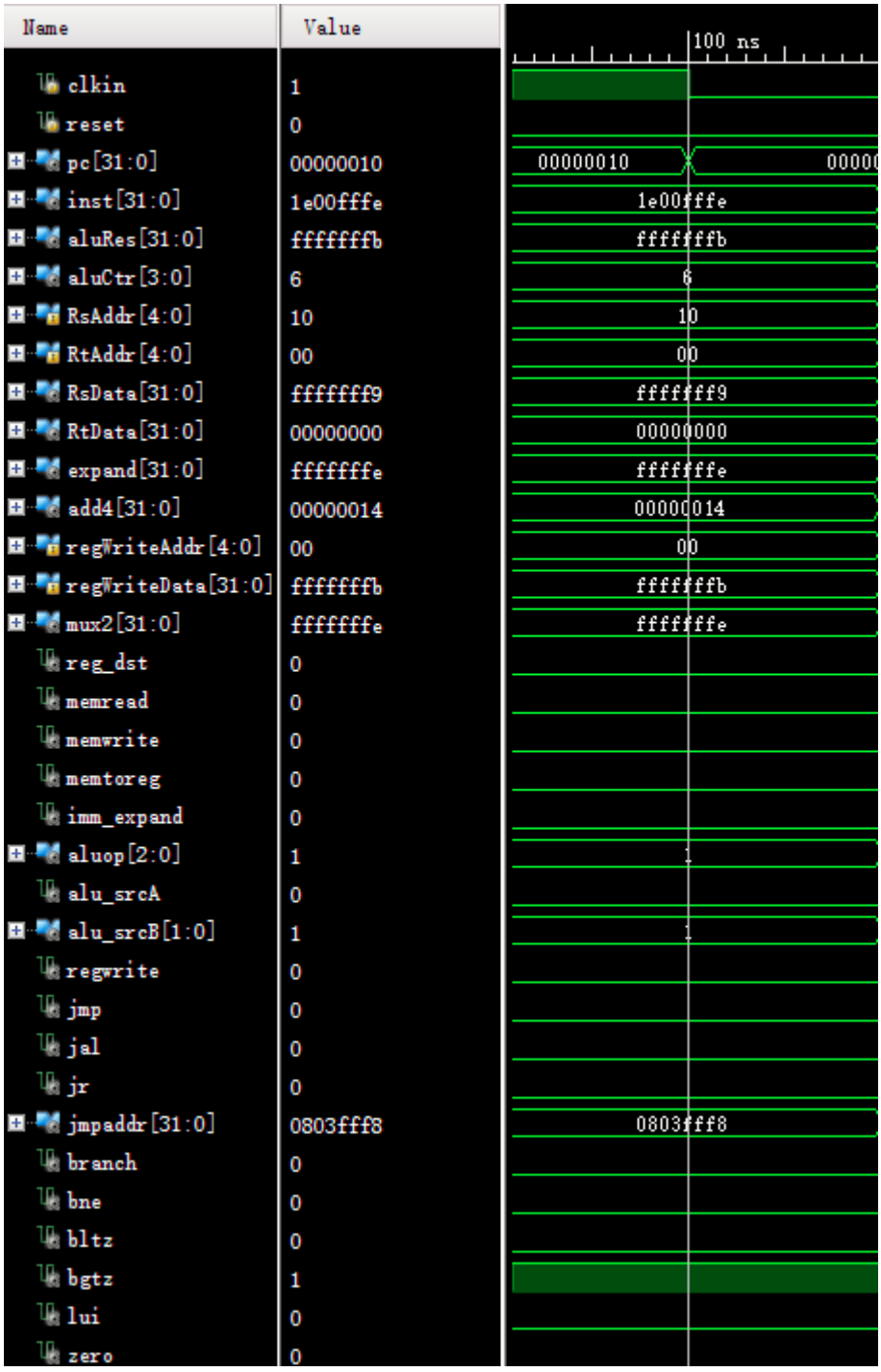
当前指令地址 pc 为 64，下一条 add4 为 68。控制信号 branch 为1。立即数 expand 为 -2。RsAddr 为 7 号寄存器，对应值 RsData 为 9；RtAddr 为 1 号寄存器，对应值 RtData 为 8。ALU 运算结果 aluRes 为 1，zero 为0，可知此时不满足跳转条件，故 pc 直接加 4 。指令执行正确。

27、bltz $10, -2



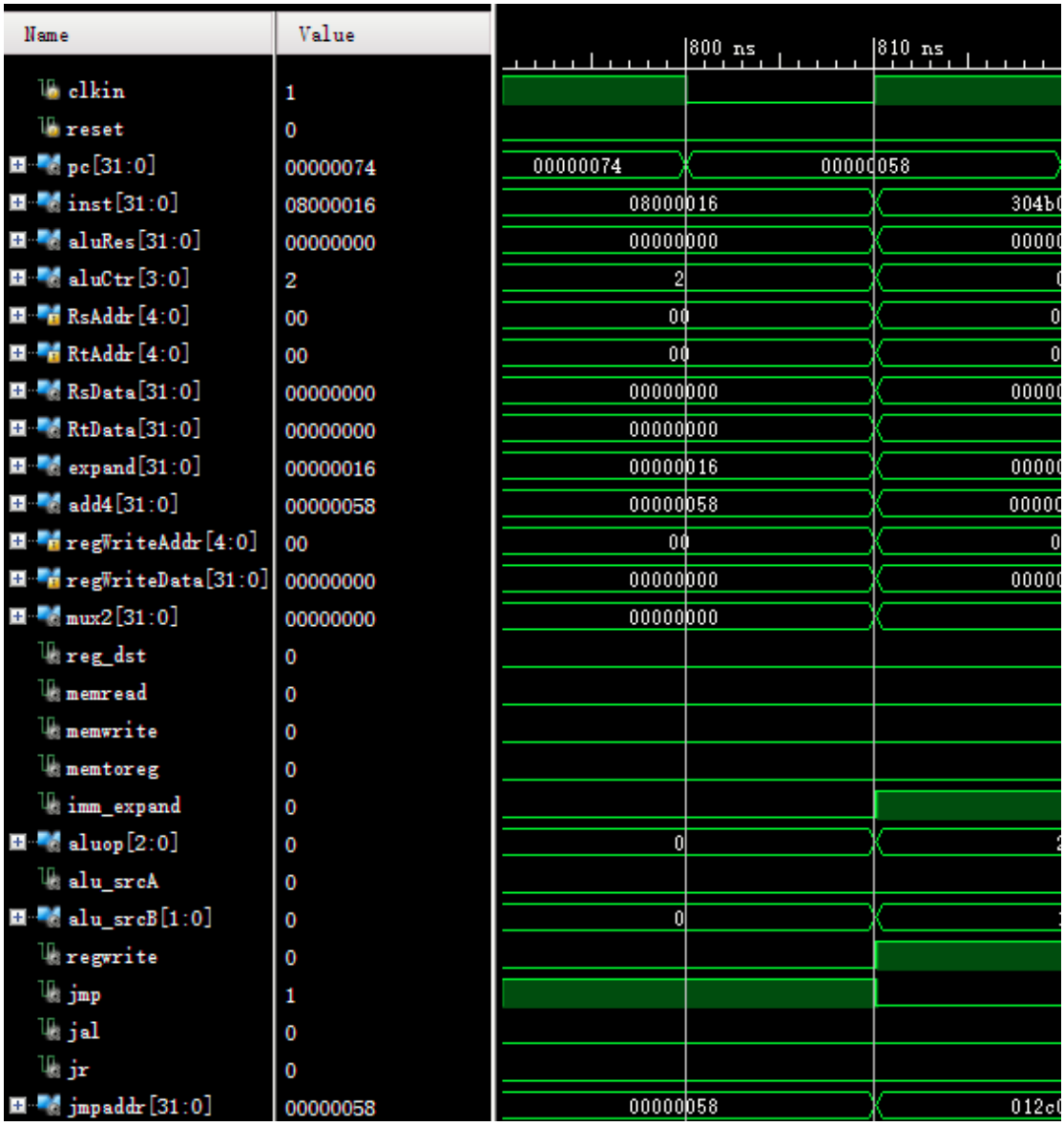
当前指令地址 pc 为 84，下一条 add4 为 88。控制信号 bltz 为1。立即数 expand 为 -2。RsAddr 为 10 号寄存器，对应值 RsData 为 ffffffff；RtAddr 为 0 号寄存器，对应值 RtData 为 0。ALU 运算结果 aluRes 为 1 > 0，可知此时满足跳转条件，故跳转到84 + 4 − 2 × 4 = 80，再次回到本指令时，可知不满足跳转条件，故 pc 直接加 4 。指令执行正确。

28、bgtz $16, -2



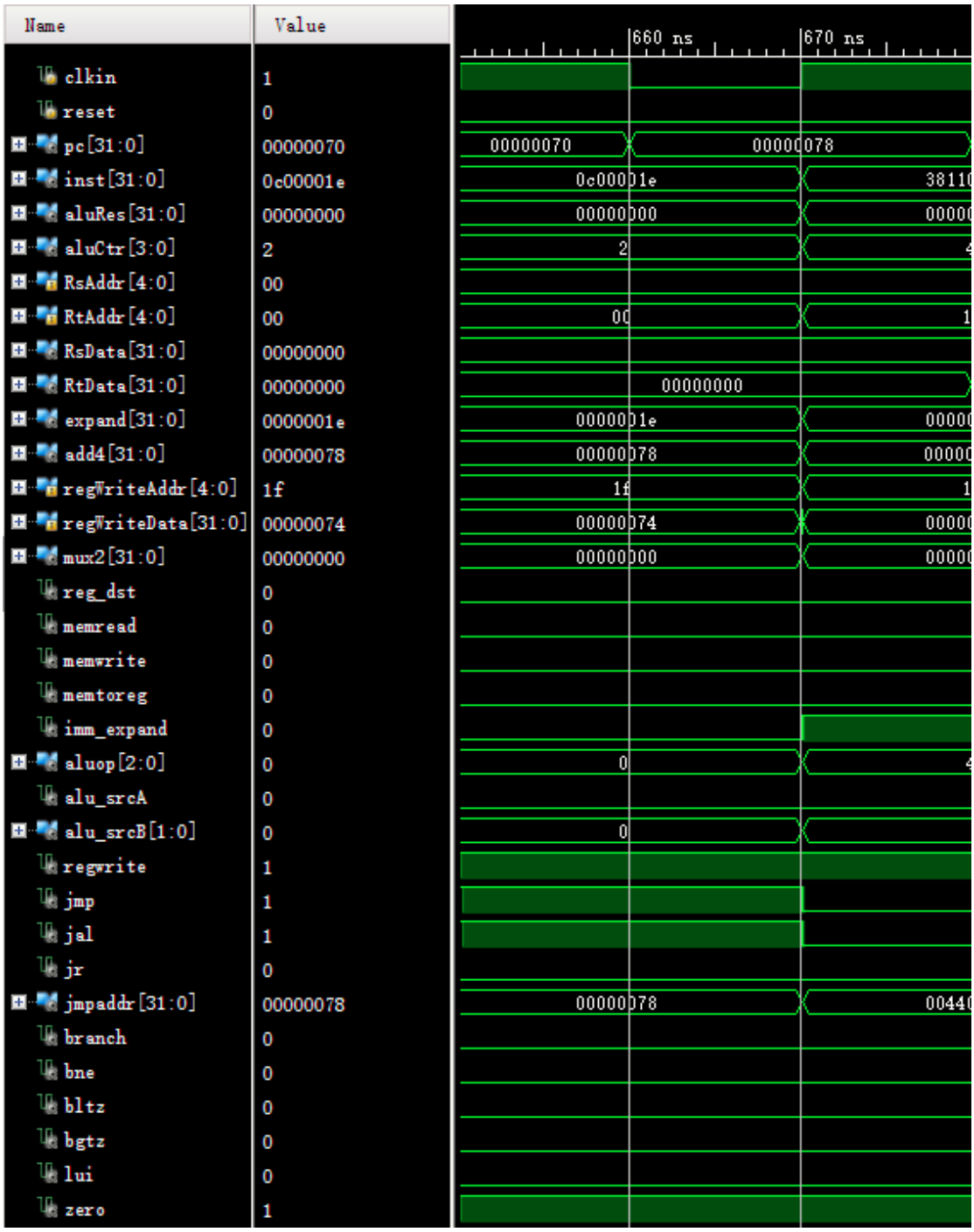
当前指令地址 pc 为 16，下一条 add4 为 20。控制信号 bgtz 为1。立即数 expand 为 -2。RsAddr 为 16 号寄存器，对应值 RsData 为 fffffff9；RtAddr 为 0 号寄存器，对应值 RtData 为 0。ALU 运算结果 aluRes 为 fffffffb < 0，可知此时不满足跳转条件，故 pc 直接加 4 。指令执行正确。

29、j 0x00000058



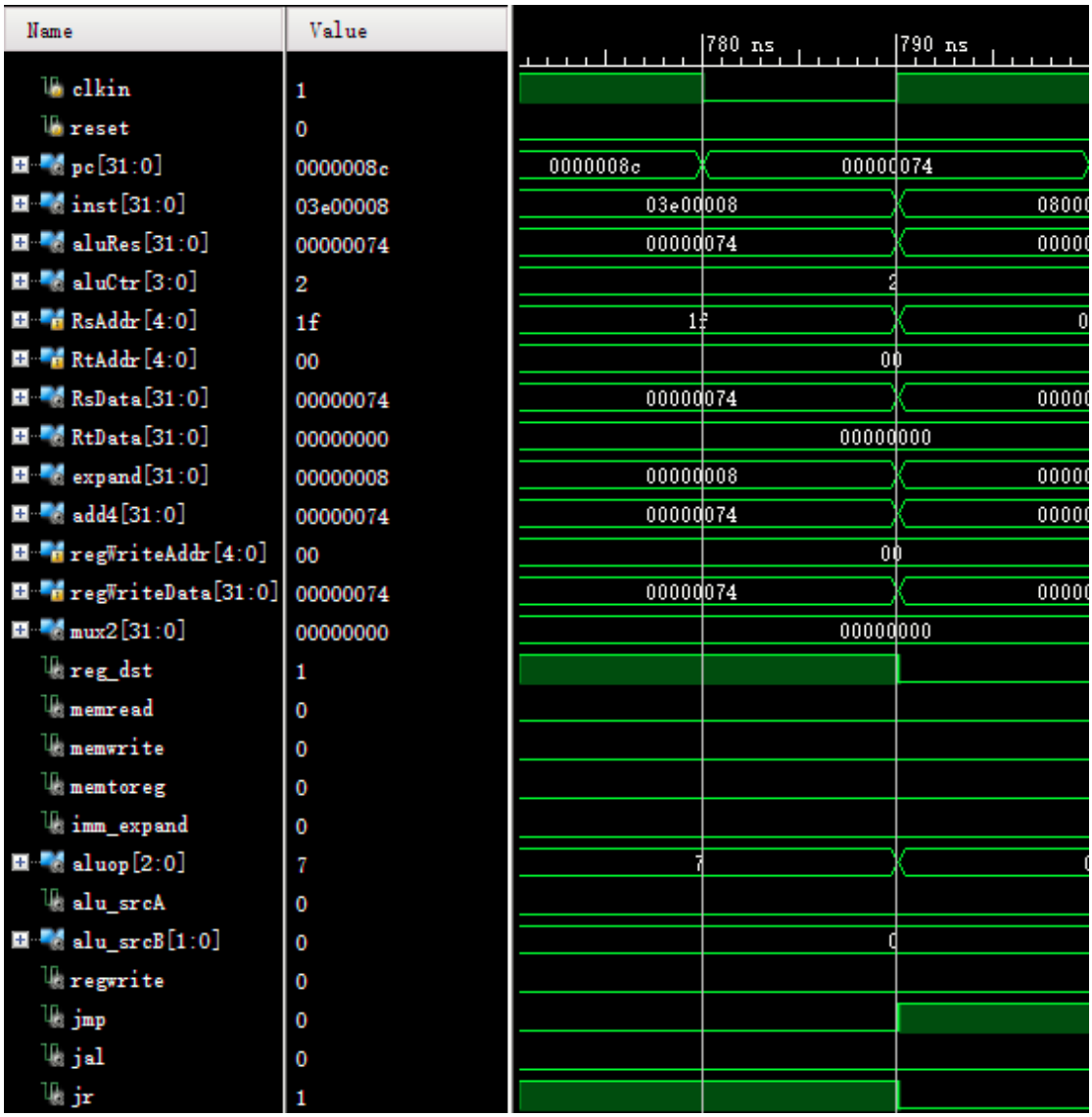
当前指令地址 pc 为 116，下一条指令地址应该为 120。控制信号 jmp 为1。跳转地址 jmpaddr 为 88（0x00000058）。可以看到 add4 为 0x00000058，pc 直接跳转至 88 （0x00000058）。指令执行正确。

30、jal 0x00000078



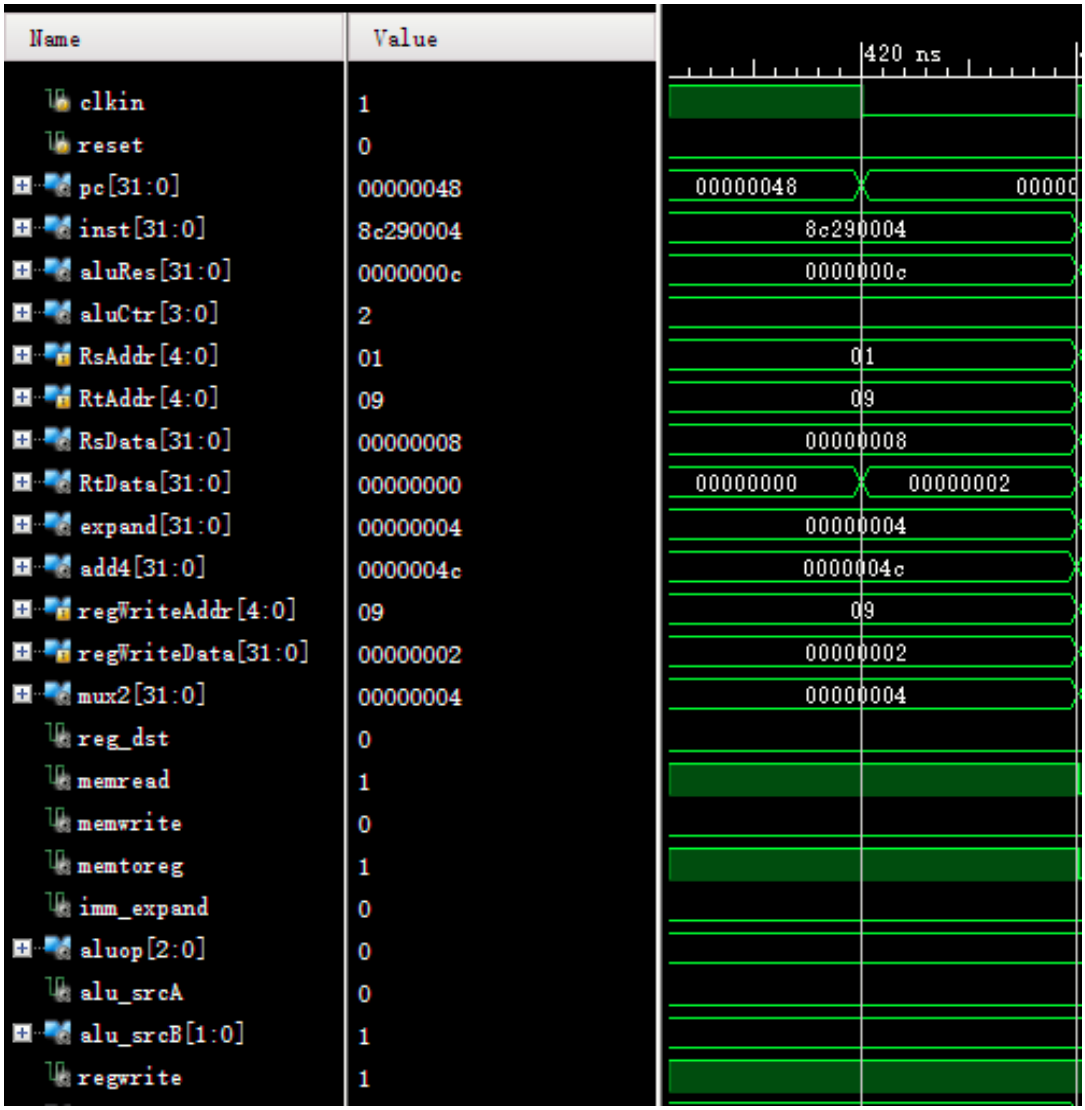
当前指令地址 pc 为 112，下一条指令地址应该为 116。控制信号 jmp 和 jal 均为 1。跳转地址 jmpaddr 为 120（0x00000078）。可以看到 add4 为 0x00000078，pc 直接跳转至 120 （0x00000078）。同时，regwrite 信号为 1 ，写入的寄存器 regWriteAddr 为 1f，即31，即 $ra，写入的内容为0x00000074，可见指令执行正确。

31、jr $31



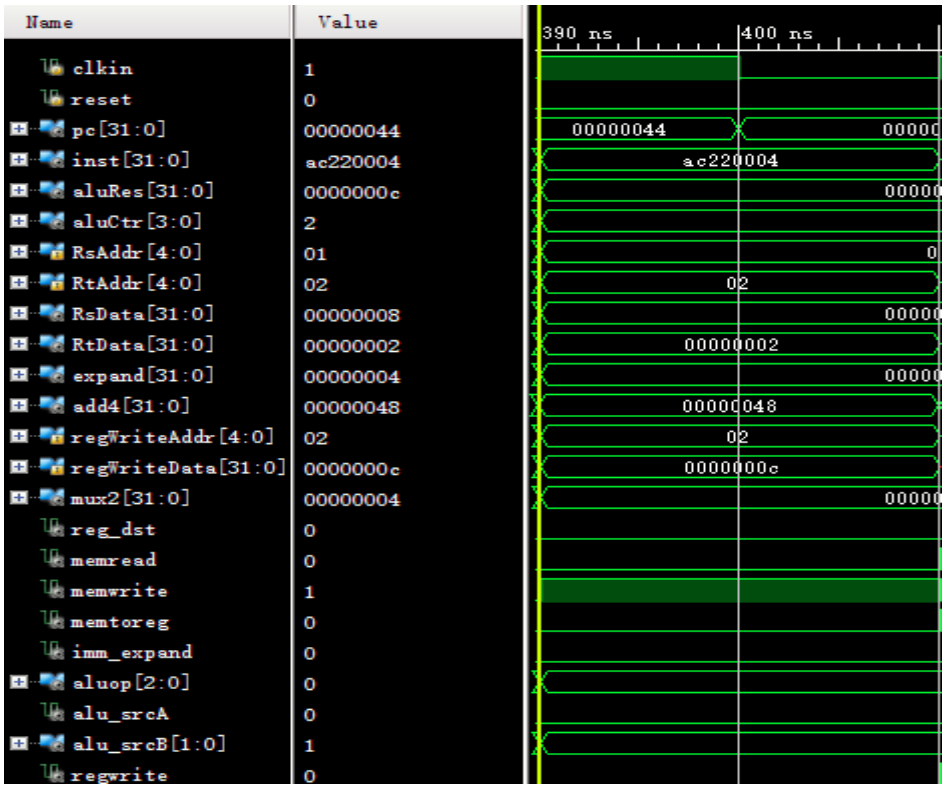
当前指令地址 pc 为 140，下一条指令地址应该为 144。控制信号 jr 为1。RsAddr 为 31 号寄存器，对应值 RsData 为 0x00000074。故直接跳转至0x0000074。指令执行正确。

32、lw $9, 4($1)



当前指令地址 pc 为 72，下一条 add4 为 76。立即数 expand 为 4。RsAddr 为 1 号寄存器，对应值 RsData 为 8；RtAddr 为 9 号寄存器，对应值 RtData 为 0。ALU 运算结果 aluRes 为 12 ，即$2的地址 。写入寄存器 regWriteAddr 为 9 号寄存器，写入数据 regWriteData 为 2。指令执行正确。

33、sw $2, 4($1)



当前指令地址 pc 为 68，下一条 add4 为 72。立即数 expand 为 4。RsAddr 为 1 号寄存器，对应值 RsData 为 8；RtAddr 为 2 号寄存器，对应值 RtData 为 2。ALU 运算结果 aluRes 为 12 。从寄存器组中可以看到 $2 的值为2，故指令执行正确。

34、halt

停机指令， pc 的值不变。

三、在 Basys3 板上运行

可以采用 Basys3 板上的七段数码管来显示ALU的运算结果。

七段译码显示的内容是 16 位的，而ALU的运算结果是 32 位的，将运算结果分为高十六位和低十六位，分别传进七段译码模块；可以用一个开关，来选择是显示高 16 位还是低16位。将输入的十六位 data 显示在四个数码管上，sm_wei 选择哪一个数码管亮，sm_duan 选择数码管的哪一段亮，sm_wei 变换的速度是 1 秒 1000 次，使人眼看起来数码管是同时显示数值的。

```
module smg_ip_model(clk,data,sel,sm_wei,sm_duan);
    input clk;
    input [31:0] data;     //数据输入
    input sel;             //选择高低16位信号
    output [3:0] sm_wei;
```

```
    output [7:0] sm_duan;
//-------------------------------------------------------------//位控制
    reg [3:0]wei_ctrl=4'b1110; //0的那个是亮的那个
    always @(posedge clk)
    wei_ctrl <= {wei_ctrl[2:0],wei_ctrl[3]}; //段控制 ,就是把最左边的一位弄到最右边
    reg [3:0]duan_ctrl;
    always @(wei_ctrl)
    case(wei_ctrl)
    4'b1110:duan_ctrl=sel?data[19:16]:data[3:0];
    4'b1101:duan_ctrl=sel?data[23:20]:data[7:4];
    4'b1011:duan_ctrl=sel?data[27:24]:data[11:8];
    4'b0111:duan_ctrl=sel?data[31:28]:data[15:12];
    default:duan_ctrl=4'hf;
    endcase
//-------------------------------------------------------------
    //解码模块
    reg [7:0]duan;
    always @(duan_ctrl)
    case(duan_ctrl) //让数码管相应的显示单元亮
    4'h0:duan=8'b1100_0000;//0
    4'h1:duan=8'b1111_1001;//1
    4'h2:duan=8'b1010_0100;//2
    4'h3:duan=8'b1011_0000;//3
    4'h4:duan=8'b1001_1001;//4
    4'h5:duan=8'b1001_0010;//5
    4'h6:duan=8'b1000_0010;//6
    4'h7:duan=8'b1111_1000;//7
    4'h8:duan=8'b1000_0000;//8
    4'h9:duan=8'b1001_0000;//9
```

```
        4'ha:duan=8'b1000_1000;//a
        4'hb:duan=8'b1000_0011;//b
        4'hc:duan=8'b1100_0110;//c
        4'hd:duan=8'b1010_0001;//d
        4'he:duan=8'b1000_0110;//e
        4'hf:duan=8'b1000_1110;//f
        // 4'hf:duan=8'b1111_1111;//不显示
        default : duan = 8'b1100_0000;//0
        endcase
        //------------------------------------------------------------
        assign sm_wei = wei_ctrl;
        assign sm_duan = duan;
    endmodule
```

然后通过仿真得到的 allures 与 Basys3 板上的数码管显示结果相对比，若两者相同，则可以认为在 Basys3 板上实现成功。

可以取连续五条以上的算数指令来验证，左边表示前 16 位，右边表示后 16 位。

1、addiu $1, $0, 8　　ALU结果应该为0x00000008

2、addi $14, $0, 1　　ALU结果应该为0x00000001

3、addu $16, $14, $1　ALU结果应该为0x00000009

4、subu $16, $14, $1　ALU结果应该为0xfffffff9


5、sltu $15, $0, $1　　ALU结果应该为0x00000001


6、ori $2, $0, 2　　ALU结果应该为0x00000002


7、add $3, $2, $1　　ALU结果应该为0x0000000a


8、sub $5, $3, $2　　ALU结果应该为0x00000008


9、and $4, $5, $2　　ALU结果应该为0x00000000


10、or $8, $4, $2　　ALU结果应该为0x00000002


然后可以看看一些分支跳转指令和内存操作指令的ALU运算结果

1、bne $8, $1, -2　　ALU结果应该为0xfffffffc < 0 故可以跳转


2、lw $9, 4($1)　　ALU结果应该为0x0000000c


3、sw　$2, 4($1)　　ALU结果应该为0x0000000c


4、jr $31　　ALU结果应该为0x00000074


5、lui $21, 9　　ALU结果应该为0x00090000


通过以上结果可以验证单周期 CPU 在 Basys3 板上成功实现。

六. 实验心得

一开始实现单周期 CPU 的时候，感觉一头雾水，看着课件上庞大的数据通路，不知道如何入手，通过实验课上老师的逐个模块的讲解，慢慢了解了CPU的设计方法，即先实现各个模块，将这些模块封装起来，有对外的输入输出端口，在完成所有基本的模块后，通过一个top（顶层文件）来将这些模块衔接起来，从而完成单周期 CPU 的设计。

单周期 CPU 的数据通路和控制信号图是我自己在电脑上慢慢画出来的，耗时一整个下午，虽然过程非常煎熬，但是当我完成了整个的数据通路时，我对 CPU 的各个模块的衔接有了更加深刻的理解。

在仿真单周期 CPU 时，有一些指令没有实现预期的结果，我就会找到那一条指令，

从控制信号开始看起，逐个模块地排查问题，最后进行修改，在这个过程中，我对每一条指令的执行过程渐渐变得非常熟练，完成一些相似的指令时也更得心应手了。

在 Basys3 板上烧写时，板子上的数码管不能一次显示16位数，后来了解到是分频的问题，调整了数码管的显示频率后就解决了问题。

后来在第一次找助教检查实验结果时，发现指令的Rs寄存器和Rt寄存器相同时（如addi $2, $2, 1），ALU在一个周期内会进行两次运算，但是只将第一次的运算结果存入寄存器或内存中，可是这样在数码管上显示的时候就会显示两个结果，听取了助教的建议后，我将测试指令中的这两个寄存器修改为不同的寄存器，或者也可以在ALU中添加代码，使其在一个周期内只进行一次运算，最后解决了问题。