计算机系统结构实验

# 实验 5 报告

类MIPS单周期处理器的设计与实现

李子龙
518070910095

2021 年 6 月 9 日

## 目录

# 1 实验目的

1. 完成单周期的类MIPS处理器
2. 设计支持16条MIPS指令（add, sub, and, or, addi, andi, ori, slt, lw, sw, beq, j, jal, jr, sll, srl）的单周期CPU

# 2 原理实现

## 2.1 指令存储器

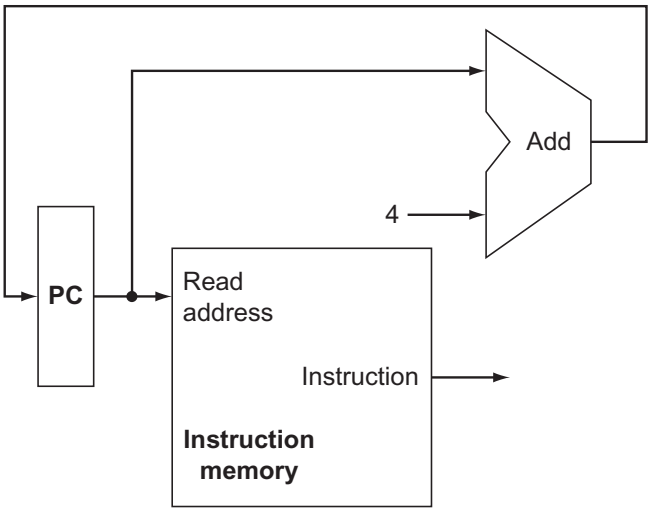


**FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter.** The fetched instruction is used by other parts of the datapath.

图 1: 指令存储器

指令存储器通过读取 PC 的值获取需要读取的指令地址。如果读取的内存地址越界，就读取第一个指令。

指令存储器被设定为 64 行。指令存储器的实现如下：

**Listing 1:** InstMemory.v

```
module InstMemory(
    input [31:0] readAddress,
    output [31:0] inst
    );
```

```
    reg [31:0] instructions [0:63];
    assign inst = instructions[readAddress / 4 < 64 ? readAddress / 4 : 0];
endmodule
```

## 2.2 顶层模块设计（一）

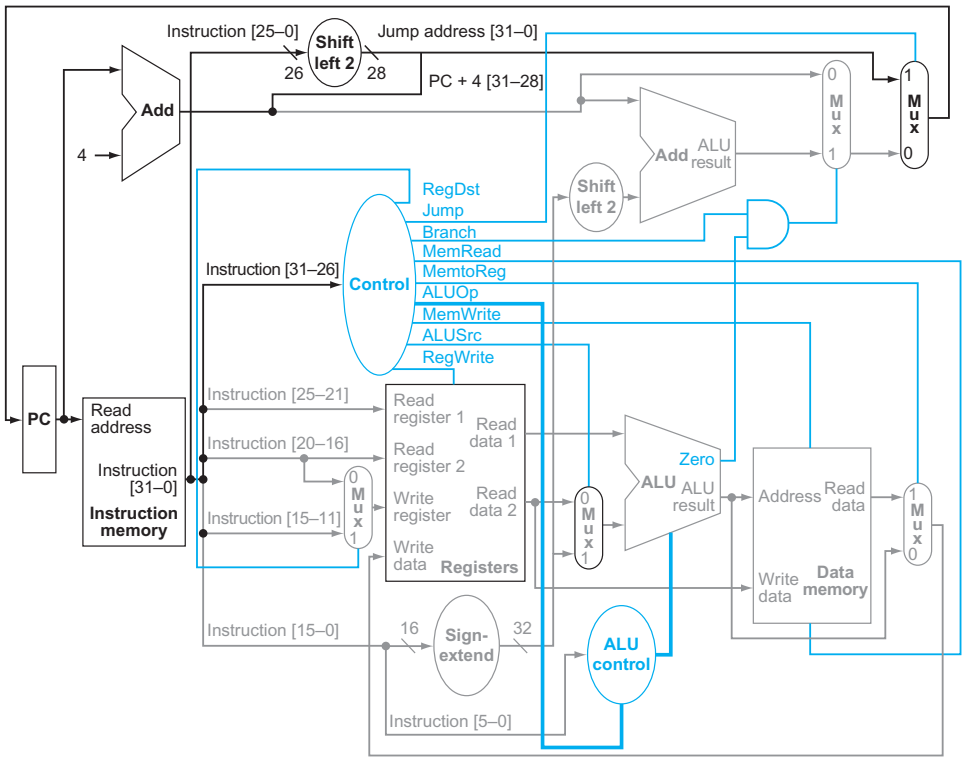


图 2: 顶层设计

单周期 MIPS 处理器共分为五个阶段：指令获取（IF）、指令译码（ID）、执行（EX）、存储（MEM）、写回（WB）。图 2 展示了针对 9 条指令的顶层模块设计。而针对 16 条指令，需要对一些原有的输入输出进行扩展。

表 1: 16条指令

| | | | |
|---|---|---|---|
| add | sub | and | or |
| **addi** | **andi** | **ori** | slt |
| lw | sw | **sll** | **srl** |
| beq | j | **jal** | **jr** |

### 2.3 Ctr 扩展

Ctr 模块会被扩展三个输出信号，相关信息如表 2 和图 3 所示。

表 2: Ctr 的扩展信号

| 信号 | 描述 | 输出 | 输入 |
|---|---|---|---|
| zext | 是否为零扩展 | Ctr | sigext |
| imm | 是否是立即数命令 | Ctr | ALUCtr |
| jal | 是否是跳转链接命令 | Ctr | Registers |

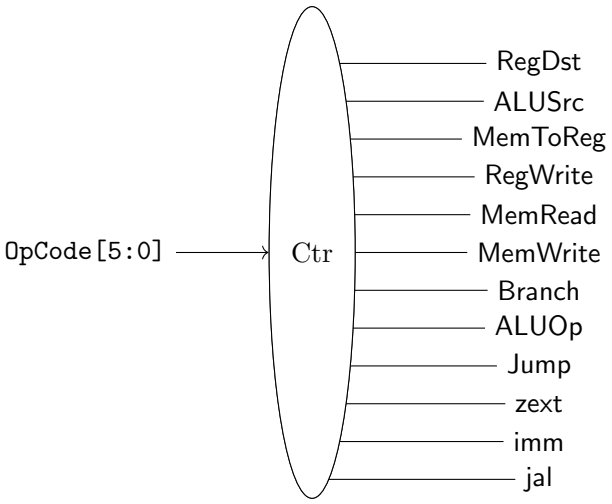


图 3: 扩展端口后的 Ctr

扩展信号输出后就可以支持 addi, andi, ori, jal ， 在实验 3 表 2 的基础上扩增：

表 3: 增扩指令

| 信号 | addi | andi | ori | jal |
|---|---|---|---|---|
| zext | 0 | 1 | 1 | 0 |
| imm | 1 | 1 | 1 | 0 |
| jal | 0 | 0 | 0 | 1 |

## 2.4 ALUCtr 扩展

针对 ALUCtr 也会进行信号扩增，如表 4 和图 4 所示。

表 4: ALUCtr 的扩展信号

| 信号 | 描述 | 输出 | 输入 |
|---|---|---|---|
| nop | 是否为空指令 | InstMemory | ALUCtr |
| jr | 是否为跳转寄存器指令 | ALUCtr | PC |
| shamt | ALU第一个信号是否是 shamt 位 | ALUCtr | ALU |

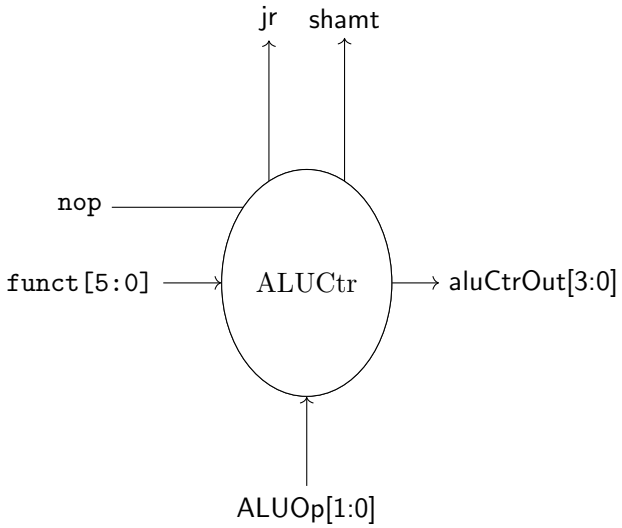


图 4: 扩展端口后的 ALUCtr

添加 nop 信号是因为其指令与 sll 冲突，如表 5 所示。

表 5: 增广指令

| 指令 | op | rs | rt | rd | shamt | funct | ALUCtrOut |
|---|---|---|---|---|---|---|---|
| nop | 000000 | 00000 | 00000 | 00000 | 00000 | 000000 | 1111 |
| sll | 000000 | 00000 | rt | rd | shamt | 000000 | 1000 |
| srl | 000000 | 00000 | rt | rd | shamt | 000010 | 1001 |
| jr | 000000 | rs | 00000 | 00000 | 00000 | 001000 | 1111 |

## 2.5 跳转与PC

本处理器为统一起见，将时钟上跳沿设定为取指和递增，下跳沿将会处理跳转与分支相关的指令。示意图如图 5 所示。

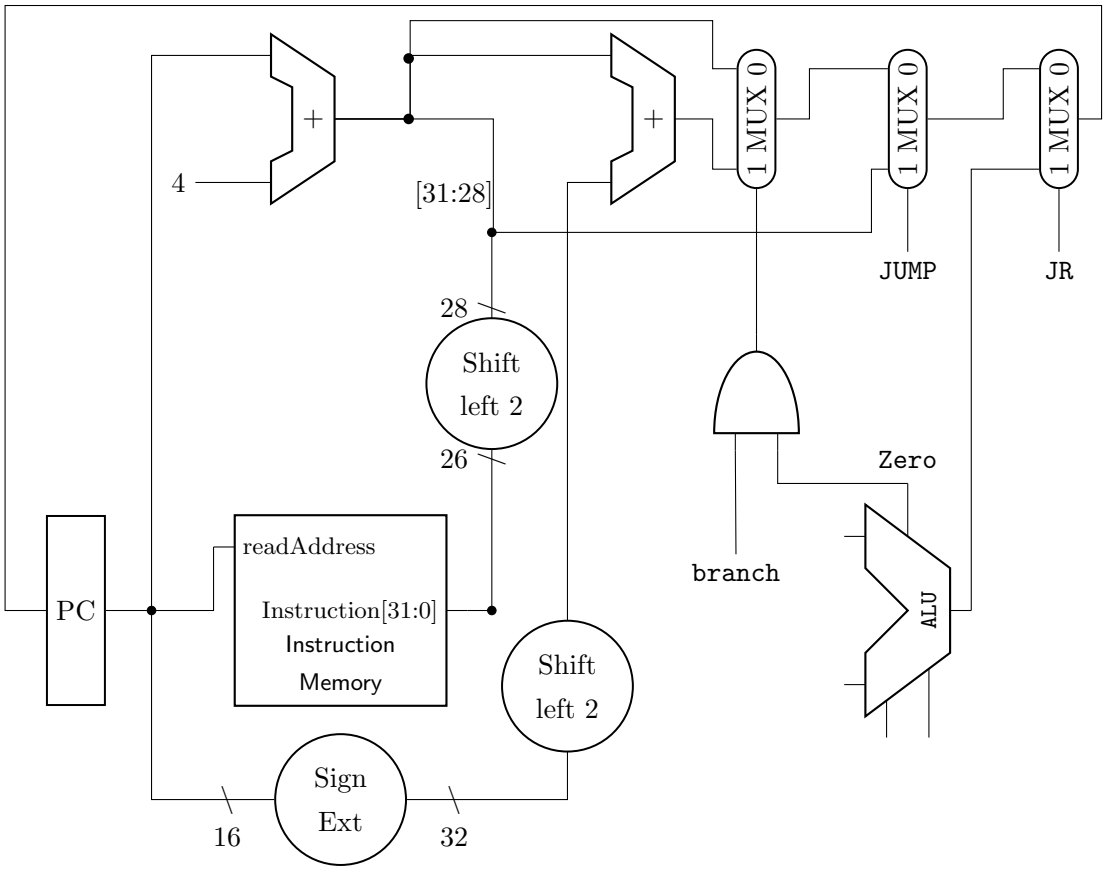


图 5: 跳转与 PC

```
always @(posedge clk) begin
    if(reset) PC <= 0; else PC <= PC + 4; end
InstMemory instMemory(.readAddress(PC), .inst(INST));
```

```
always @(negedge clk) begin
    PC <= JR ? ALU_RES : (JUMP ? (PC[31:28] + INST[25:0] << 2) :
                ((BRANCH & ZERO) ? (PC + (OPRAND << 2)) : PC)); end
```

## 2.6 JAL 和寄存器堆

Jump And Link (JAL) 指令会影响寄存器堆的输入。如图 6 所示，在有 JAL 指令时会直接写入第 31 号寄存器，并写入 PC + 4 的值（此时 PC 还没有变化，保留上一次末尾值）。

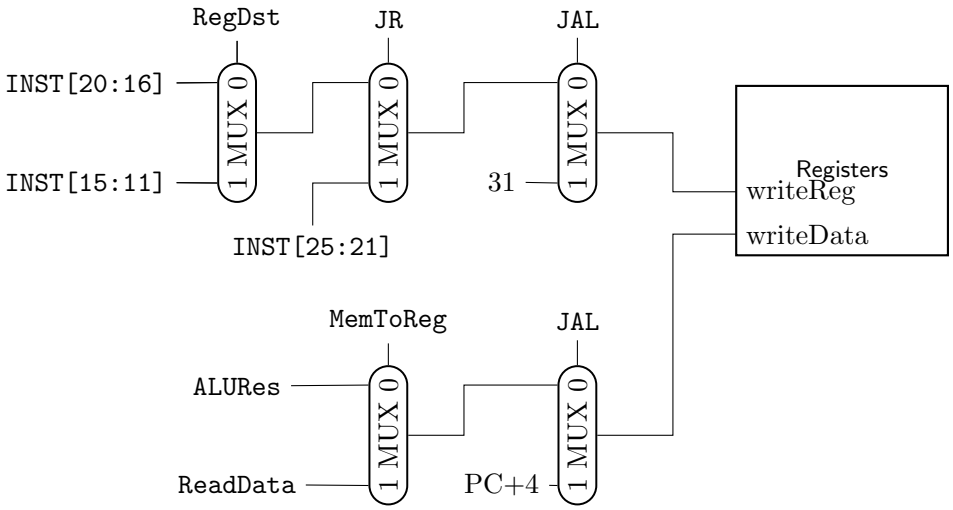


图 6: JAL 和寄存器堆

```
Registers registers(
    .clk(clk),
    .reset(reset),
    .readReg1(INST[25:21]),
    .readReg2(INST[20:16]),
    .writeReg(JAL ? 5'b11111 : (JR ? INST[25:21] : (REG_DST ? INST
        [15:11] : INST[20:16]))),
    .writeData(JAL ? PC + 4 : (MEM_TO_REG ? READ_DATA : ALU_RES)), //
        Jal will jump to PC + 4
    .regWrite(REG_WRITE),
    .readData1(READ_DATA1),
    .readData2(READ_DATA2)
);
```

## 2.7 顶层模块设计（二）

添加这些数据线后，就可以支持 16 条指令了。

**Listing 2:** Top.v

```
`timescale 1ns / 1ps
//////////////////////////////////
// Company:
// Engineer: Zilong Li
//
// Create Date: 2021/05/26
    08:25:11
// Design Name:
// Module Name: Top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////


module Top(
        input clk,
        input reset
    );

    reg [31:0] PC;

    always @(posedge clk) begin
        if(reset)
            PC <= 0;
        else begin
            PC <= PC + 4;
        end
    end

    wire [31:0] INST;

    InstMemory instMemory(
        .readAddress(PC),
        .inst(INST)
    );

    wire REG_DST,
        JUMP,
        BRANCH,
        MEM_READ,
        MEM_TO_REG,
        MEM_WRITE;
    wire [1:0] ALU_OP;
    wire ALU_SRC,
        REG_WRITE,
        ZEXT,
        IMM,
        JAL;

    Ctr mainCtr(
        .opCode(INST[31:26]),
        .regDst(REG_DST),
        .jump(JUMP),
        .branch(BRANCH),
        .memRead(MEM_READ),
        .memToReg(MEM_TO_REG),
        .aluOp(ALU_OP),
        .memWrite(MEM_WRITE),
        .aluSrc(ALU_SRC),
        .regWrite(REG_WRITE),
        .zext(ZEXT),
        .imm(IMM),
```

```
        .jal(JAL)
    );

    wire [31:0] READ_DATA1;
    wire [31:0] READ_DATA2;
    wire [31:0] OPRAND;
    wire [3:0] ALU_CTR;
    wire ZERO;
    wire [31:0] ALU_RES;
    wire [31:0] READ_DATA;
    wire JR;
    wire SHAMT;

    Registers registers(
        .clk(clk),
        .reset(reset),
        .readReg1(INST[25:21]),
        .readReg2(INST[20:16]),
        .writeReg(JAL ? 5'b11111 :
            (JR ? INST[25:21] : (
            REG_DST ? INST[15:11] :
            INST[20:16]))),
        .writeData(JAL ? PC + 4 : (
            MEM_TO_REG ? READ_DATA :
             ALU_RES)), // Jal will
            jump to PC + 4
        .regWrite(REG_WRITE),
        .readData1(READ_DATA1),
        .readData2(READ_DATA2)
    );

    signext signExt(
        .inst(INST[15:0]),
        .zext(ZEXT),
        .data(OPRAND)
    );


    ALUCtr aluctr(
        .nop(INST == 0 ? 1'b1 : 1'
            b0), // Avoid nop
            conflict
        .funct(IMM ? INST[31:26] :
            INST[5:0]),
        .aluOp(ALU_OP),
        .aluCtrOut(ALU_CTR),
        .jr(JR),
        .shamt(SHAMT)
    );

    ALU alu(
        .input1(SHAMT ? INST[10:6]
            : READ_DATA1),
        .input2(ALU_SRC ? OPRAND :
            READ_DATA2),
        .aluCtr(ALU_CTR),
        .zero(ZERO),
        .aluRes(ALU_RES)
    );

    dataMemory DataMemory(
        .Clk(clk),
        .address(ALU_RES),
        .writeData(READ_DATA2),
        .memWrite(MEM_WRITE),
        .memRead(MEM_READ),
        .readData(READ_DATA)
    );

    always @(negedge clk) begin
    PC <= JR ? ALU_RES : (JUMP ?
                (PC[31:28] + INST
                    [25:0] << 2) :
```

```
                // 26 -> 28
131             ((BRANCH & ZERO) ? (
                    PC + (OPRAND <<
                    2)) : PC));
```

```
132     end
133
134 endmodule
```

# 3 仿真结果

使用了下面的指令文件进行仿真。该指令文件主要的作用是测试所有的运算功能，并在每一个循环对 10 号寄存器 + 1，并存储到 0 号存储单元中，直到其超过刚开始的限制寄存器的存储数字（这里是 4），之后就会进入短循环，不会再对寄存器和存储器进行修改。

**Listing 3:** simple.asm

```
1  nop
2  lw  $16, 8($0)          # $0 zero register
3  jal 4
4  nop
5  lw  $8, 0($0)
6  lw  $9, 4($0)
7  sub $10, $8, $9
8  and $10, $8, $9
9  slt $10, $8, $9
10 or  $10, $8, $9
11 addi $10, $8, 8
12 andi $10, $8, -1
13 ori  $10, $8, -1
14 sll $10, $8, 1
15 srl $10, $8, 1
16 add $10, $8, $9        # final save: += 1
17 sw  $10, 0($0)
18 beq $10, $16, 1
19 jr $31
20 j 16
```

**Listing 4:** mem_data.mem

```
1 00000001
2 00000001
```

```
00000004
00000000
00000000
00000000
00000000
00000000
00000000
```

激励文件加载指令集的时候，要注意 `$readmemb` 是二进制读取，`$readmemh` 是十六进制读取。

**Listing 5:** Basic_tb.v

```
module Basic_tb(

    );

    reg clk;
    reg reset;

    Top Proc(.clk(clk),.reset(reset));

    initial begin
        $readmemb("mem_inst.mem",Proc.instMemory.instructions);
        $readmemh("mem_data.mem",Proc.DataMemory.MemFile,10'h0);
        reset = 1;
        clk = 0;
    end

    always #10 clk = ~clk;

    initial begin
        #80 reset = 0;
    end

endmodule
```

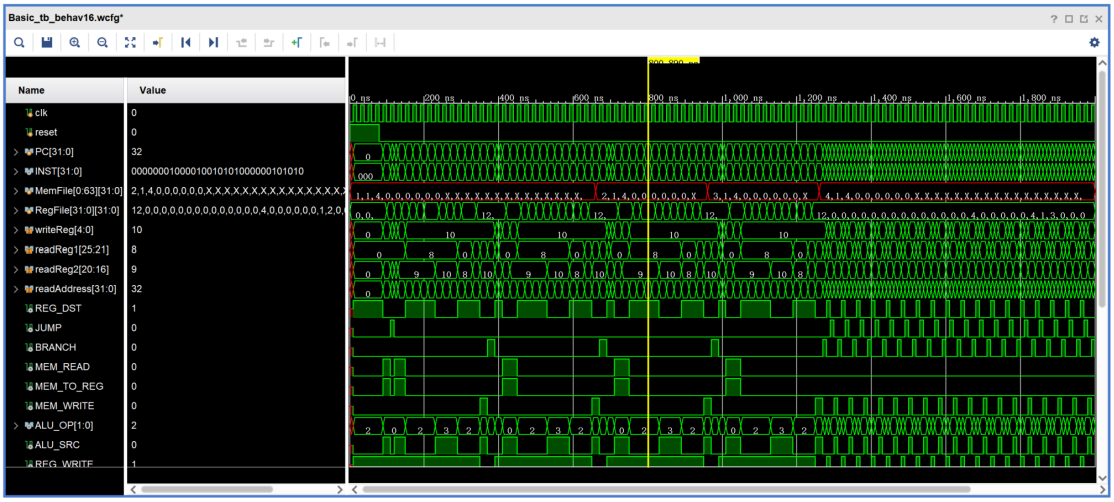


图 7: 16条指令的仿真结果

运算全过程见图 7，正常运行，MemFile 在最后为 4 之后就不再改变。每一个循环的运算细节见图 8。最后跳转的情况见图 9。所有的运算都得到了正确的结果。

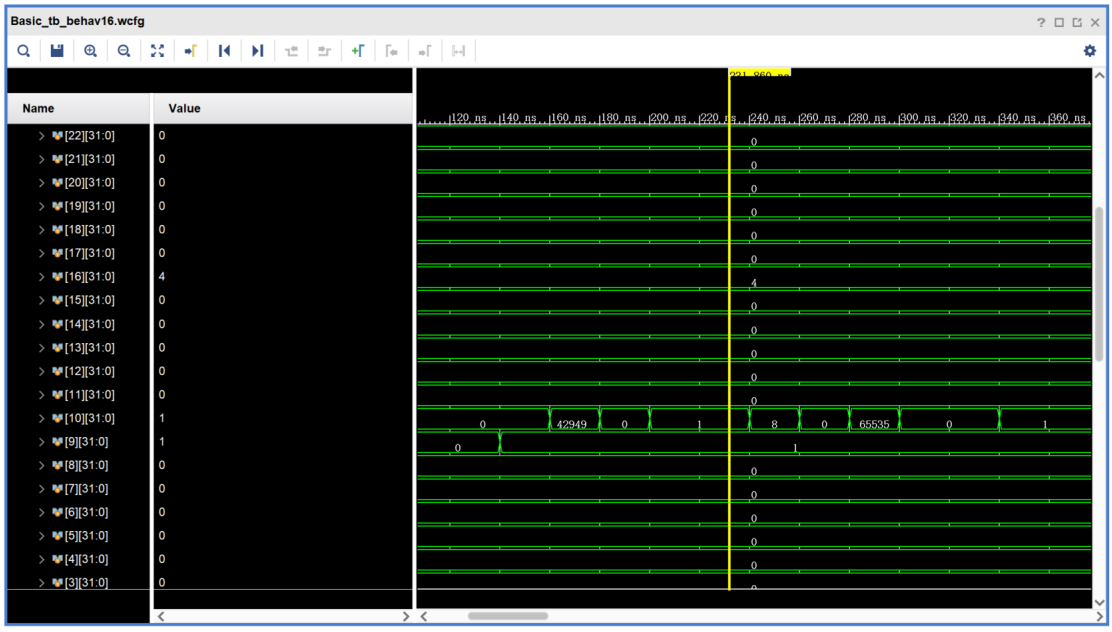


图 8: 运算细节

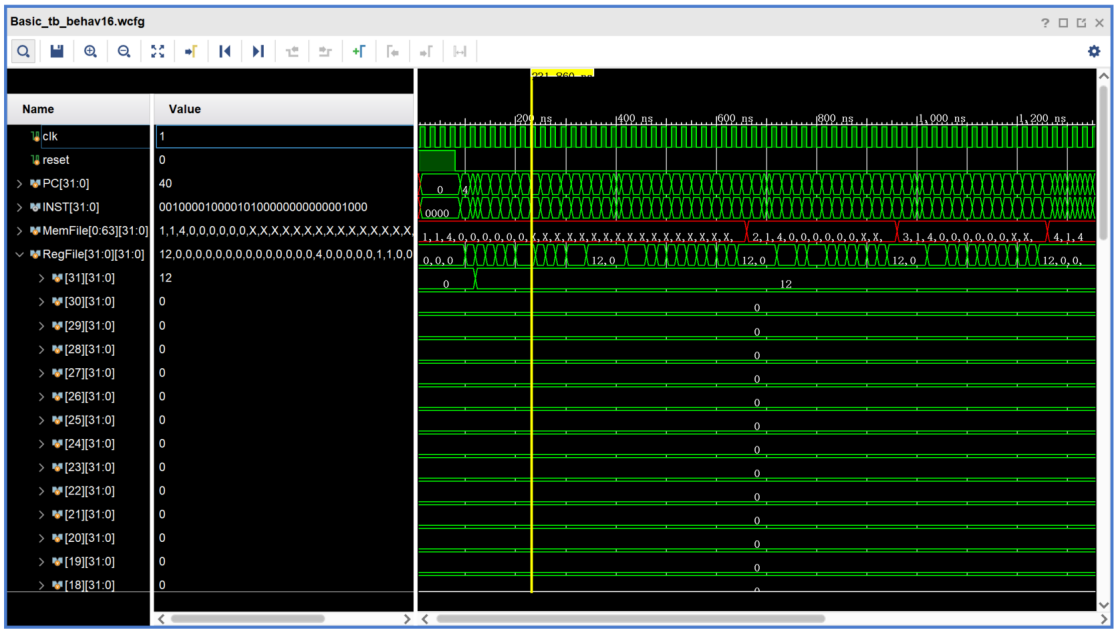


图 9: jal让31号寄存器有了地址

# 4 实验心得

本次实验调试时间较长，花费近 8 个小时。

主要是顶层设计的连线方案比较复杂，对于增扩的指令，需要重新设计端口加以实现。查看波形进行逐个波形的调试，有助于排查故障。这里面对于 ALUCtr 使用了 nop 扩增非常关键，否则会导致空指令执行了某些移位操作导致 0 寄存器非零，从而影响下面数据的读取。对于 PC 的写入顺序进行了比较多的调试，最后确定了上跳沿递增、下跳沿跳转的方案，可以互不干扰。并且选路器没有采用模块化设计，而是使用三目运算符，让代码更加简洁易懂。

本次实验增加了我对单周期处理器设计的理解，对于理解一些原理有很大的帮助，对于下面的流水线处理器设计也会起到关键的作用。