

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

Log Creative

2021 年 6 月 27 日

目录

1 实验目的	2
2 原理实现	2
2.1 指令存储器	2
2.2 顶层模块设计（一）	3
2.3 Ctr 扩展	4
2.4 ALUCtr 扩展	5
2.5 跳转与PC	6
2.6 JAL 和寄存器堆	7
2.7 顶层模块设计（二）	7
3 仿真结果	10
4 实验心得	13

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

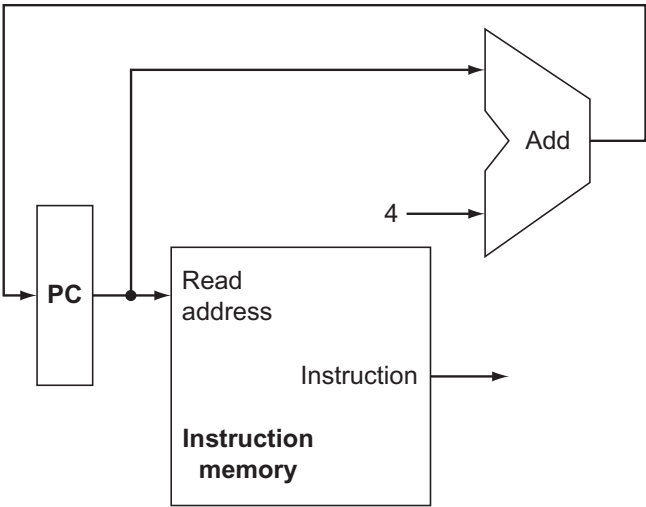


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

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

```
1 module InstMemory(
2     input [31:0] readAddress,
3     output [31:0] inst
4 );
```

```

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

```

2.2 顶层模块设计（一）

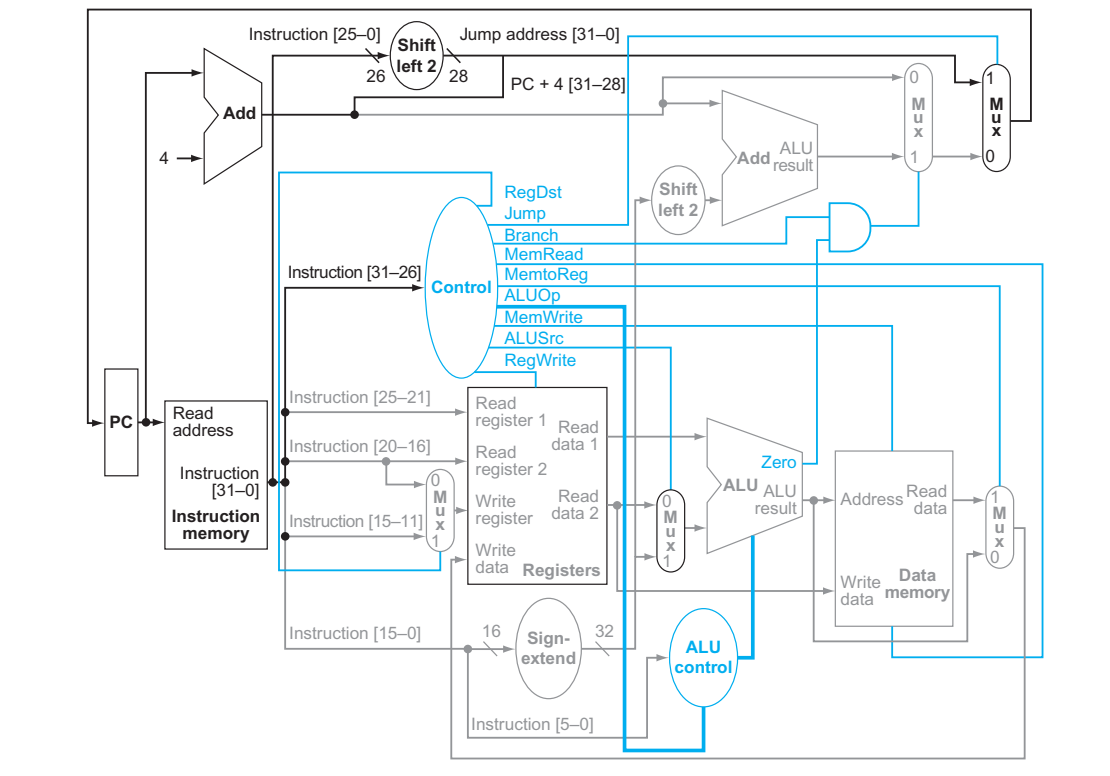


图 2: 顶层设计

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

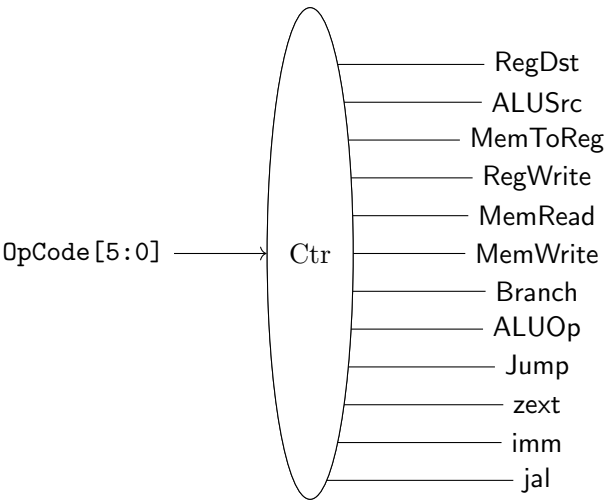


图 3: 扩展端口后的 Ctr

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

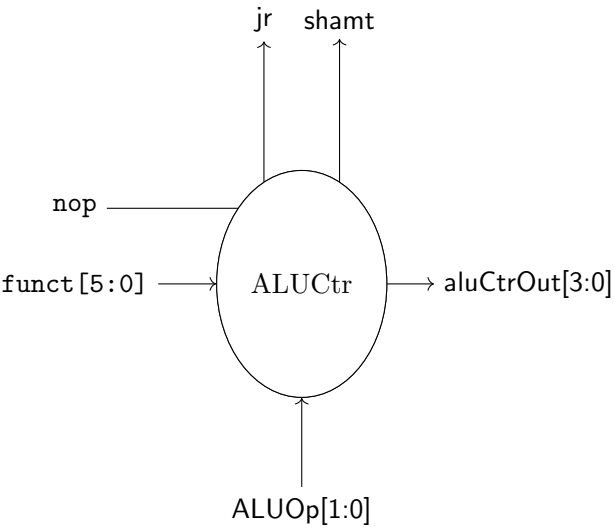


图 4: 扩展端口后的 ALUCtr

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。

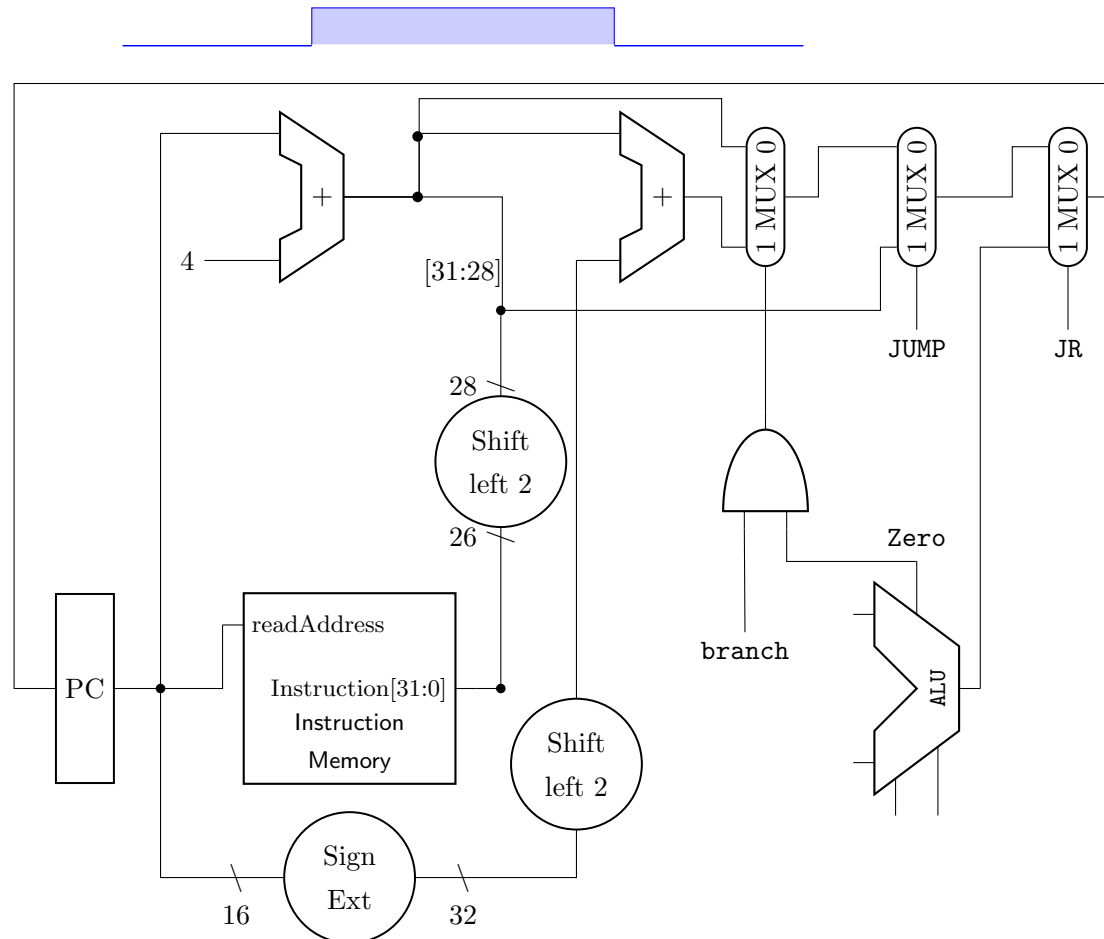


图 5: 跳转与 PC

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

```

1 always @(negedge clk) begin
2     PC <= JR ? ALU_RES : (JUMP ? (PC[31:28] + INST[25:0] << 2) :
3         ((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）。

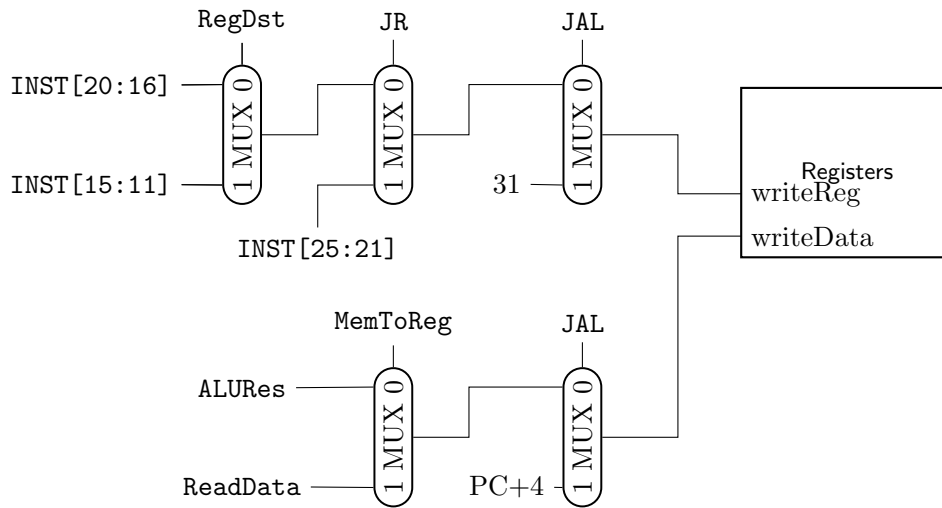


图 6: JAL 和寄存器堆

```

1  Registers registers(
2      .clk(clk),
3      .reset(reset),
4      .readReg1(INST[25:21]),
5      .readReg2(INST[20:16]),
6      .writeReg(JAL ? 5'b11111 : (JR ? INST[25:21] : (REG_DST ? INST
7          [15:11] : INST[20:16]))),
8      .writeData(JAL ? PC + 4 : (MEM_TO_REG ? READ_DATA : ALU_RES)), //
9          Jal will jump to PC + 4
10     .regWrite(REG_WRITE),
11     .readData1(READ_DATA1),
12     .readData2(READ_DATA2)
13 );

```

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

```

1  `timescale 1ns / 1ps
2  //////////////////////////////////////
3  // Company:
4  // Engineer: Zilong Li
5  //
6  // Create Date: 2021/05/26
7  // Design Name:
8  // Module Name: Top
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////
21
22
23 module Top(
24     input clk,
25     input reset
26 );
27
28     reg [31:0] PC;
29
30     always @(posedge clk) begin
31         if(reset)
32             PC <= 0;
33         else begin
34             PC <= PC + 4;
35
36         end
37
38     wire [31:0] INST;
39
40     InstMemory instMemory(
41         .readAddress(PC),
42         .inst(INST)
43     );
44
45     wire REG_DST,
46         JUMP,
47         BRANCH,
48         MEM_READ,
49         MEM_TO_REG,
50         MEM_WRITE;
51
52     wire [1:0] ALU_OP;
53
54     wire ALU_SRC,
55         REG_WRITE,
56         ZEXT,
57         IMM,
58         JAL;
59
60     Ctr mainCtr(
61         .opCode(INST[31:26]),
62         .regDst(REG_DST),
63         .jump(JUMP),
64         .branch(BRANCH),
65         .memRead(MEM_READ),
66         .memToReg(MEM_TO_REG),
67         .aluOp(ALU_OP),
68         .memWrite(MEM_WRITE),
69         .aluSrc(ALU_SRC),
70         .regWrite(REG_WRITE),
71         .zext(ZEXT),
72         .imm(IMM),

```



```

71     .jal(JAL)
72 );
73
74 wire [31:0] READ_DATA1;
75 wire [31:0] READ_DATA2;
76 wire [31:0] OPRAND;
77 wire [3:0] ALU_CTR;
78 wire ZERO;
79 wire [31:0] ALU_RES;
80 wire [31:0] READ_DATA;
81 wire JR;
82 wire SHAMT;
83
84 Registers registers(
85     .clk(clk),
86     .reset(reset),
87     .readReg1(INST[25:21]),
88     .readReg2(INST[20:16]),
89     .writeReg(JAL ? 5'b11111 :
90         (JR ? INST[25:21] : (
91             REG_DST ? INST[15:11] :
92             INST[20:16]))),
93     .writeData(JAL ? PC + 4 : (
94         MEM_TO_REG ? READ_DATA :
95         ALU_RES)), // Jal will
96         jump to PC + 4
97     .regWrite(REG_WRITE),
98     .readData1(READ_DATA1),
99     .readData2(READ_DATA2)
100 );
101
102 signext signExt(
103     .inst(INST[15:0]),
104     .zext(ZEXT),
105     .data(OPRAND)
106 );
107
108 ALUctr aluctr(
109     .nop(INST == 0 ? 1'b1 : 1'
110         b0), // Avoid nop
111         conflict
112     .funct(IMM ? INST[31:26] :
113         INST[5:0]),
114     .aluOp(ALU_OP),
115     .aluCtrOut(ALU_CTR),
116     .jr(JR),
117     .shamt(SHAMT)
118 );
119
120 ALU alu(
121     .input1(SHAMT ? INST[10:6]
122         : READ_DATA1),
123     .input2(ALU_SRC ? OPRAND :
124         READ_DATA2),
125     .aluCtr(ALU_CTR),
126     .zero(ZERO),
127     .aluRes(ALU_RES)
128 );
129
130 dataMemory DataMemory(
131     .Clk(clk),
132     .address(ALU_RES),
133     .writeData(READ_DATA2),
134     .memWrite(MEM_WRITE),
135     .memRead(MEM_READ),
136     .readData(READ_DATA)
137 );
138
139 always @(negedge clk) begin
140     PC <= JR ? ALU_RES : (JUMP ?
141         (PC[31:28] + INST
142         [25:0] << 2) :

```

```

131 // 26 -> 28
      ((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

```

```
3 00000004
4 00000000
5 00000000
6 00000000
7 00000000
8 00000000
9 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

```
1 module Basic_tb(
2
3     );
4
5     reg clk;
6     reg reset;
7
8     Top Proc(.clk(clk),.reset(reset));
9
10    initial begin
11        $readmemb("mem_inst.mem",Proc.instMemory.instructions);
12        $readmemh("mem_data.mem",Proc.DataMemory.MemFile,10'h0);
13        reset = 1;
14        clk = 0;
15    end
16
17    always #10 clk = ~clk;
18
19    initial begin
20        #80 reset = 0;
21    end
22
23 endmodule
```

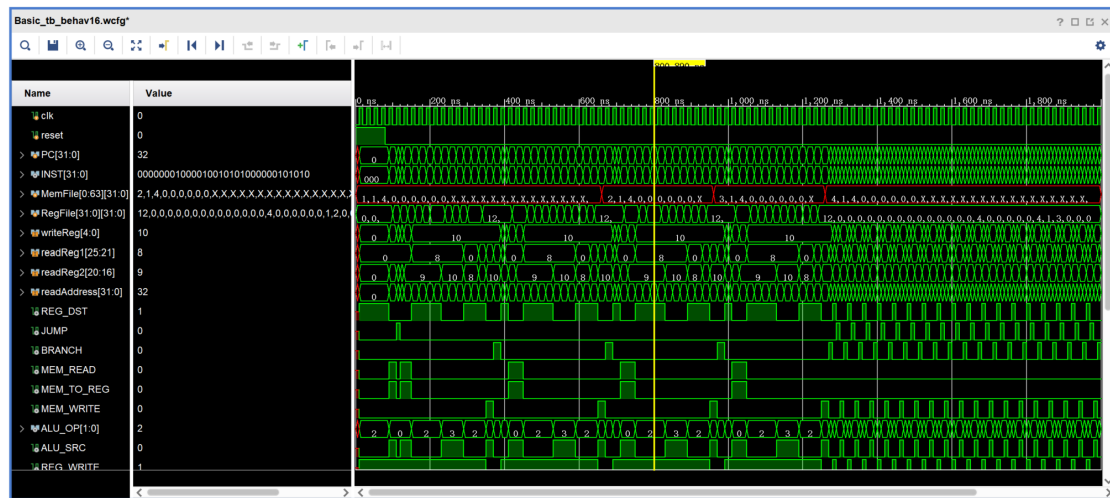


图 7: 16条指令的仿真结果

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。

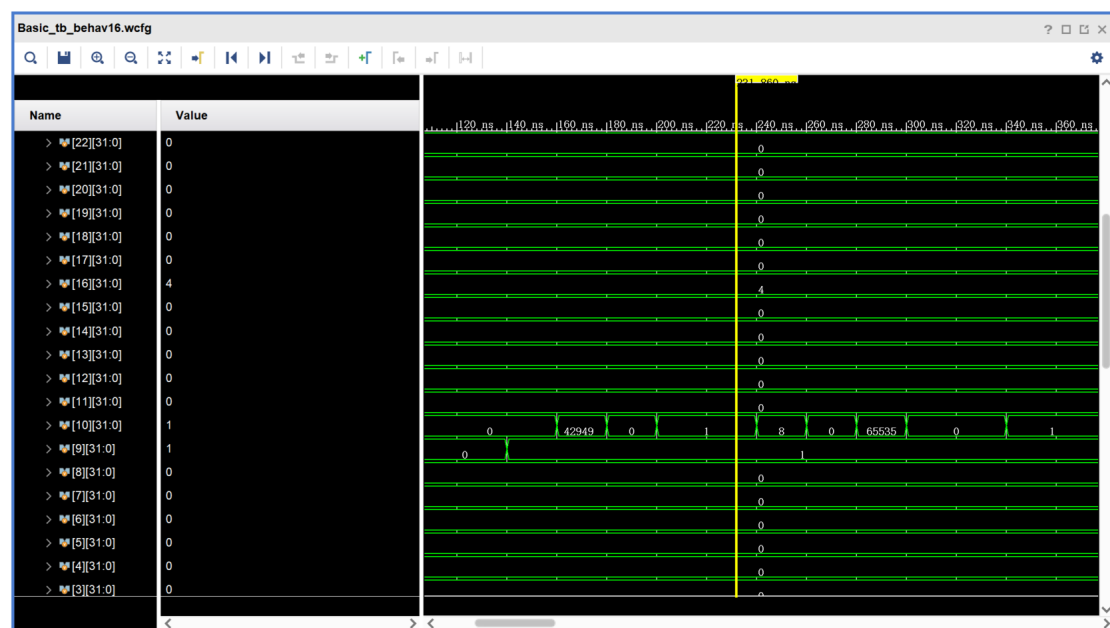


图 8: 运算细节

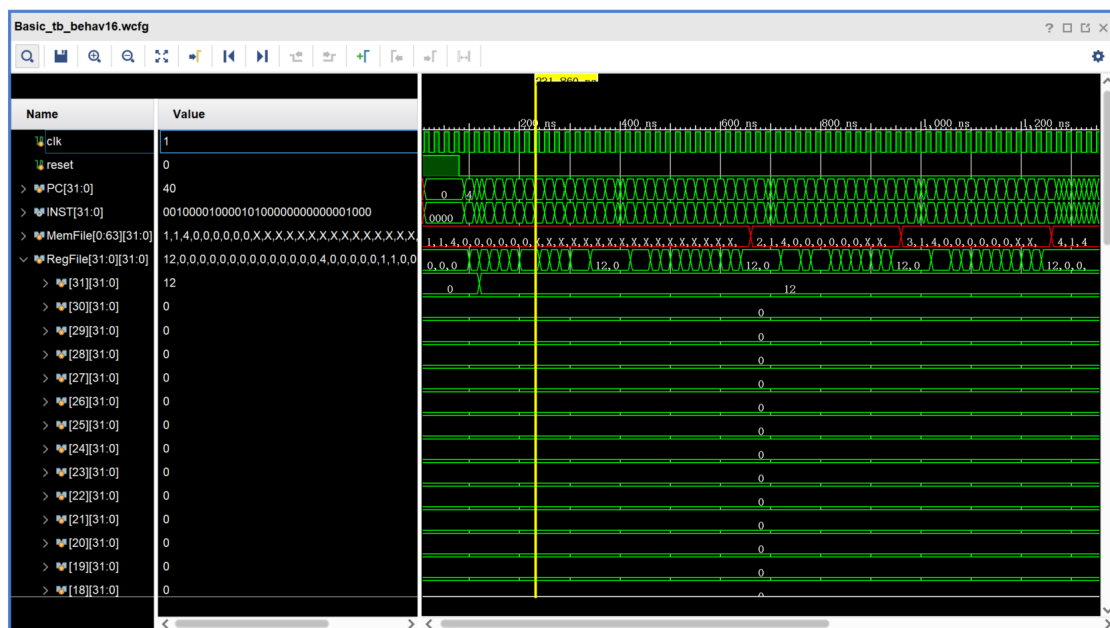


图 9: jal让31号寄存器有了地址

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。