Arch Lab 5

MIPS 单周期处理器的设计与实现

Alex Chi

日期：2020 年 5 月 1 日

目录

1 简介

在本次实验中，我完成了下面的任务。

- 结合之前实验手册，和本实验的需求，重新设计了大部分模块。（包括 DataMemory, RegisterFile, ALU, Ctr 等）

- 完成了一个支持几乎所有 MIPS 指令 (36 个) 的单周期 MIPS CPU。
- 通过汇编器自己编写了 5 个程序，通过 Vivado 进行测试，和汇编器结果比较。

2 实验目的

- 完成单周期的类 MIPS 处理器。
- 设计支持 36 条 MIPS 指令（具体指令见下文）的单周期 CPU。

3 MIPS CPU 设计

本次实验中，我编写的单周期 MIPS CPU 支持下面这些指令。

R-type	add	addu	and	noop	or	sllv	slt	sltu		srlv	subu	sub	xor
Immediate	addi	addiu	andi	lui	ori	sll	slti	sltiu	sra	srl			xori
Branch	beq	bgez	bgtz	blez	bltz	bne							
Jump and Link	j	jal	jr										
Memory	lb	lw	sb	sw									

这些指令可以大体分为以下几类。

- R-type 和 I-type 数值运算指令,比如 add, addi。这些指令读取寄存器的值,仅修改 Register File。
- 分支和跳转指令，如 beq, j, jr。这些指令读取寄存器的值，并且有可能修改 PC。
- 跳转连接指令，如 jal。这些指令不仅修改 PC，还会写入寄存器。
- 内存指令，如 sb, lb。这些指令会读写内存、修改寄存器的值。

由于指令数繁多，在单周期 CPU 中，需要有行之有效的方法，在减少代码量的同时高效处理这些指令。下面列表总结这些指令对元器件的依赖性。

	寄存器读	寄存器写	内存	修改 PC
数值运算	Y	Y	N	N
内存读	N	Y	Y	N
内存写	Y	N	Y	N
分支指令	Y	N	N	Y
jal	N	Y	N	Y
jr	Y	N	N	Y
j	N	N	N	Y

下面将具体介绍每一类指令在不同阶段的执行过程。

3.1 读寄存器

从指令内存读取到指令的第一步，就是分析指令的操作，并从寄存器中读取所需要的操作数。在这里，要考虑下面几种情况。

```
0000 00ss ssst tttt dddd d000 00ff ffff (R-type)
oooo ooss ssst tttt iiii iiii iiii iiii (I-type, load, store, lui)
oooo ooss ssst tttt iiii iiii iiii iiii (beq, bne)
oooo ooss sss0 000x iiii iiii iiii iiii (bgez, bgtz, blez, bltz)
0000 10ii iiii iiii iiii iiii iiii iiii (j, jal)
0000 00ss sss0 0000 0000 0000 0000 1000 (jr)
0000 00ss ssst tttt dddd dhhh hh00 0000 (sll, sra, srl)
```

可以发现，对于不同的 opcode，指令中各比特位代表的含义不同。因此，在读取寄存器前，要通过逻辑电路确定所需要读取的寄存器。

- 对于 R-type，需要的寄存器就是 rs 和 rt。写入 rd。
- 对于内存指令，需要的寄存器是 rs，另一个可以设置为 0。写入 rt。
- 对于不是 R-type 的分支指令，仅需要 rs。rt 的值本身是分支指令需要比较的值。
- 对于跳转指令，jal 需要写入 31 号寄存器。jr 需要读取 rs。
- 对于位移指令，还需要把指令中的 shamt (shift amount) 读取出来。

通过 IsShift 模块确认是否需要读取 shamt 的值。

```
module IsShift(
    input [5:0] funct,
    output reg shift);

    always @ (funct) begin
        case (funct)
            6'h02: shift = 1;
            6'h03: shift = 1;
            6'h00: shift = 1;
            default: shift = 0;
        endcase
    end
endmodule
```

而对于读到的立即数，一些指令需要做符号位扩展，另一些要做 0 扩展。在这一阶段，把两个值都计算出来。通过这些特殊判断，我们可以编写出这一阶段的 Verilog 程序。

```
wire [5:0] opcode = inst[31:26];
wire [4:0] rs = inst[25:21];
wire [4:0] rt = inst[20:16];
wire [4:0] rd = inst[15:11];
wire [4:0] shamt = inst[10:6];
wire [5:0] funct = inst[5:0];
```

```
wire [15:0] imm = inst[15:0];
wire [31:0] imm_sign_ext;
wire [31:0] imm_zero_ext;
wire [31:0] shamt_zero_ext = {{27'b0}, shamt};
SignExt signExt(.unextended (imm), .extended (imm_sign_ext));
ZeroExt zeroExt(.unextended (imm), .extended (imm_zero_ext));
wire is_shift;
IsShift isShift(.funct (funct), .shift (is_shift));
wire is_type_R = (opcode == 0);
wire use_shamt = is_shift && is_type_R;
wire [31:0] jump_target = {4'b00, inst[25:0], 2'b00} | (pc & 32'hf0000000);
wire is_branch;
wire is_memory;

wire [4:0] rf_src1 = rs;
wire [4:0] rf_src2 = is_type_R || is_branch || is_memory ? rt : 0;
wire [4:0] rf_dest =  is_type_R ? rd : (
                        opcode == 3 ? 31 : rt);
wire [31:0] rf_out1;
wire [31:0] rf_out2;
wire [31:0] rf_data;
wire rf_write;
```

3.2 做运算

在做运算之前，我们要确定 ALU 的输入。分两部分讨论：ALU 的两个数值输入，和 ALU 进行的操作。

- 对于 R-type 的数值运算指令，直接把寄存器的两个输出作为 ALU 的输入即可。ALU 的操作和 funct 一致。
- 对于 I-type 的数值运算指令，根据 opcode 对立即数做符号扩展（通过 ExtMode 模块），然后作为 ALU 的输入。ALU 的操作和 opcode 一致。
- 对于内存指令，我们需要通过 ALU 计算读写内存的地址。这个过程和 I-type 数值指令一致。ALU 的操作为加法。
- 对于分支指令，我们需要通过 ALU 判断两个数的大小关系。对于 bne, beq，直接将两个寄存器的值作为 ALU 的输入即可。而 bgez, bgtz, blez, bltz 指令的实质，就是将寄存器的值和 -1, 0, 1 做比较。因此这些分支指令的 ALU 输入是寄存器和 -1, 0, 1。在本文的实现中，通过 BranchOp 模块生成这些数。
- 对于跳转指令，用不到 ALU。在这一阶段不做特殊处理。

```
module BranchOp(
  input [5:0] opcode,
  output reg branch_op,
  output reg override_rt,
  output reg [31:0] rt_val);
```

```
  always @ (*) begin
      case (opcode)
          // beq
          6'h04: branch_op = 1;
          // bne
          6'h05: branch_op = 1;
          // bgez, bltz
          6'h01: branch_op = 1;
          // bgtz
          6'h07: branch_op = 1;
          // blez
          6'h06: branch_op = 1;
          default: branch_op = 0;
      endcase
      case (opcode)
          // blez
          6'h06: begin override_rt = 1; rt_val = 1; end
          // bgtz
          6'h07: begin override_rt = 1; rt_val = 1; end
          // bgez, bltz
          6'h01: begin override_rt = 1; rt_val = 0; end
          default: begin override_rt = 0; rt_val = 0; end
      endcase
  end
endmodule
```

```
module ExtMode(
  input [5:0] opcode,
  output reg signExt);

  always @ (opcode) begin
      case (opcode)
          6'h0c: signExt = 0;
          6'h0d: signExt = 0;
          6'h0e: signExt = 0;
          6'h24: signExt = 0;
          6'h25: signExt = 0;
          default: signExt = 1;
      endcase
  end
endmodule
```

通过 ALUOp 模块将 opcode 和 funct 映射到 ALU 操作所对应的编号。

```
module ALUOp(
  input [5:0] opcode,
  output reg [5:0] ALUopcode);
```

```
  always @ (opcode) begin
      case (opcode)
          // branch instructions
          // beq, bne = sub
          6'h04: ALUopcode = 6'h22;
          6'h05: ALUopcode = 6'h22;
          // bgez, bltz = slt
          6'h01: ALUopcode = 6'h2A;
          // bgtz, blez = slt
          6'h06: ALUopcode = 6'h2A;
          6'h07: ALUopcode = 6'h2A;
          // lb, lw, sb, sw = add
          6'h20: ALUopcode = 6'h20;
          6'h23: ALUopcode = 6'h20;
          6'h28: ALUopcode = 6'h20;
          6'h2B: ALUopcode = 6'h20;
          // other instructions stay the same
          default: ALUopcode = opcode;
      endcase
  end
endmodule
```

最后通过逻辑电路得到送给 ALU 的数据。

```
wire ext_mode;
ExtMode extMode (.opcode (opcode), .signExt (ext_mode));
wire [5:0] alu_op = is_type_R ? funct : mapped_op;
wire [31:0] alu_imm = ext_mode ? imm_sign_ext : imm_zero_ext;
wire [31:0] alu_src1 = use_shamt ? shamt_zero_ext : rf_out1;
wire [31:0] alu_src2 = is_type_R ? rf_out2 : (
                        is_branch ?
                            (override_rt ? branch_rt_val : rf_out2)
                        : alu_imm);
wire [31:0] alu_out;
wire alu_zero;
```

3.3 ALU 运算

ALU 模块通过之前阶段得到的两个操作数和操作编号进行运算。

```
module ALU(
    input [5:0] ALUopcode,
    input [31:0] op1,
    input [31:0] op2,
    output reg [31:0] out,
    output reg zero);
```

```
always @ (ALUopcode or op1 or op2) begin
    case (ALUopcode)
        // add
        6'h20: out = op1 + op2;
        // addu
        6'h21: out = op1 + op2;
        // addi
        6'h08: out = op1 + op2;
        // addiu
        6'h09: out = op1 + op2;
        // sub
        6'h22: out = op1 - op2;
        // subu
        6'h23: out = op1 - op2;
        // and
        6'h24: out = op1 & op2;
        // andi
        6'h0C: out = op1 & op2;
        // nor
        6'h27: out = ~(op1 | op2);
        // or
        6'h25: out = op1 | op2;
        // ori
        6'h0D: out = op1 | op2;
        // xor
        6'h26: out = op1 ^ op2;
        // xori
        6'h0E: out = op1 ^ op2;
        // lui
        6'h0F: out = {op2[15:0], op1[15:0]};
        // sll
        6'h00: out = op2 <<< op1;
        // sllv
        6'h04: out = op2 <<< op1;
        // sra
        6'h03: out = $signed(op2) >>> op1;
        // srav
        6'h07: out = $signed(op2) >>> op1;
        // srl
        6'h02: out = op2 >>> op1;
        // srlv
        6'h06: out = op2 >>> op1;
        // slt
        6'h2A: if ($signed(op1) < $signed(op2)) out = 1; else out = 0;
        // slti
        6'h0A: if ($signed(op1) < $signed(op2)) out = 1; else out = 0;
```

```
            // sltu
            6'h2B: if (op1 < op2) out = 1; else out = 0;
            // sltiu
            6'h0B: if (op1 < op2) out = 1; else out = 0;
            default: out = 0;
        endcase

        if (out == 0) zero = 1; else zero = 0;
    end
endmodule
```

3.4 分支判断

在得到 ALU 的输出后，我们可以根据计算的值确定分支是否需要跳转。下面列表说明分支跳转和 ALU 输出之间的关系。

指令	ALU 操作数 1	ALU 操作数 2	ALU 指令	跳转对应情况	对应 alu_zero
beq	rs	rt	sub	rs = rt	1
bne	rs	rt	sub	rs != rt	0
bgez	rs	0	slt	rs >= 0	1
bgtz	rs	1	slt	rs >= 1 (rs > 0)	1
blez	rs	1	slt	rs < 1 (rs <= 0)	0
bltz	rs	0	slt	rs < 0	0

通过 TakeBranch 模块进行这个判断。

```
module TakeBranch(
    input [5:0] opcode,
    input [4:0] rt,
    input alu_zero,
    output reg take_branch);

    always @ (*) begin
        casez ({opcode, rt[3:0]})
            10'h4?: take_branch = alu_zero;
            10'h5?: take_branch = !alu_zero;
            10'h11: take_branch = alu_zero;
            10'h10: take_branch = !alu_zero;
            10'h70: take_branch = alu_zero;
            10'h60: take_branch = !alu_zero;
            default: take_branch = 0;
        endcase
    end
endmodule
```

而后根据 TakeBranch 模块的返回计算下一轮的 PC。在这里要考虑：

- 正常指令，不跳转。PC = PC + 4。
- 分支指令，在需要跳转时跳转。
- 跳转指令，必然跳转。j 和 jal 跳转到立即数所对应的位置。jr 跳转到寄存器存储的位置。

```
wire [31:0] new_pc = take_branch ? branch_pc : (
                            (opcode == 2 || opcode == 3) ? jump_target : (
                                (opcode == 0 && funct == 8) ? rf_out1 : next_pc));
```

3.5 内存操作

由于本人编写的 MIPS 模拟器支持按字节操作地址，所以内存模块需要能够按字节寻址。

- sb, lb 指令只读取、修改一个字节。lb 指令对读到的字节做符号扩展。
- sw 指令修改四个字节。lw 指令读取四个字节。本人设计的 CPU 采用 little endian。

由此可以设计 DataMemory 模块。在读写 4 个字节时，要将数据拆成四份读写。

```
module DataMemory(
  input clk,
  input [31:0] address,
  input [31:0] writeData,
  input [2:0] mode,
  input memWrite,
  input memRead,
  input reset,
  output [31:0] readData);

  parameter mem_size = 65536;
  reg [7:0] memFile [0:mem_size];

  always @ (negedge clk) begin
      if (memWrite)
          case (mode)
              1: memFile[address] <= writeData[7:0];
              2: begin
                  // assume little endian
                  memFile[address] <= writeData[7:0];
                  memFile[address + 1] <= writeData[15:8];
                  memFile[address + 2] <= writeData[23:16];
                  memFile[address + 3] <= writeData[31:24];
              end
          endcase
  end

  assign readData = (reset || !memRead) ? 0 : (
      mode == 1 ? {{24{memFile[address][7]}}, memFile[address]} : (
          mode == 2 ? {memFile[address + 3], memFile[address + 2], memFile[address
              + 1], memFile[address]} : 0
```

```
      ));
endmodule
```

mode 信号由 MemoryOp 模块产生。

```
module MemoryOp(
    input [5:0] opcode,
    output reg store,
    output reg load,
    output reg memory_op,
    output reg [2:0] memory_mode);

    always @ (*) begin
        case (opcode)
            6'h20: load = 1;
            6'h23: load = 1;
            default: load = 0;
        endcase
        case (opcode)
            6'h28: store = 1;
            6'h2b: store = 1;
            default: store = 0;
        endcase
        case (opcode)
            // lb, sb
            6'h20: memory_mode = 1;
            6'h28: memory_mode = 1;
            // lw, sw
            6'h23: memory_mode = 2;
            6'h2B: memory_mode = 2;
            default: memory_mode = 0;
        endcase
        memory_op = load | store;
    end
endmodule
```

CPU 中对于 Memory 阶段的处理如下。

```
assign dmem_addr = alu_out;
assign dmem_in = rf_out2;
assign dmem_write = is_memory_store;
assign dmem_read = is_memory_load;
```

3.6 写寄存器

只有数值运算、读内存和部分跳转指令需要写入寄存器。在时钟下降沿更新 PC。

```
assign rf_write = !is_branch && !dmem_write && opcode != 2;
```

```
assign rf_data = is_memory_load ? dmem_out : (
    opcode == 3 ? pc + 4 : alu_out);

always @ (negedge clk) begin
    pc <= reset ? 0 : new_pc;
end
always @ (negedge reset) begin
    pc <= 0;
end
```

3.7 指令内存

指令内存中要提前装载文件中的数据。由于 PC 是 4 的倍数，而指令内存是 4 字节寻址，这里需要对 PC 进行换算再读取。

```
module InstMemory(
    input wire clk,
    input [31:0] address,
    output [31:0] readData);

    parameter mem_size = 65536;
    parameter mem_file = "mips_hex/7-jump.mem";

    reg [31:0] memFile [0:mem_size];

    integer i;
    initial begin
        for(i = 0; i < mem_size; i = i + 1) begin
      memFile[i] = 0;
    end
        $readmemh(mem_file, memFile);
    end

    assign readData = memFile[address >>> 2];
endmodule
```

4 仿真与测试

在实验手册提供的测试之外，我还自己通过汇编器编写了测试。

4.1 数值运算测试

```
li $zero, 100
add $zero, $zero, -1000
```

```
li $at, 0xffff
lui $at, 0xffff
move $a0, $at
move $a1, $at
move $a2, $at
move $a3, $at
add $a0, $a0, $a0
addu $a1, $a1, $a1
addi $a2, $a2, 233
addiu $a2, $a2, 233
li $t0, 100
lui $t0, 233
li $t1, 233
lui $t1, 0xffff
or $t2, $t0, $t1
ori $t3, $t0, 0xfffe
and $t4, $t0, $t1
andi $t5, $t0, 0xfffe
sub $t6, $zero, $t0
subu $t7, $zero, $t0
subi $s1, $zero, 0xffff
subiu $s2, $zero, 0xffff
xor $s3, $t0, $t1
xori $s4, $t0, 0xffff
nor $s5, $t0, $t1
```

4.2 比较指令测试

```
li $a0, 1000000
li $a1, 2000000
li $a2, 20
li $a3, -100000
sll $t0, $a0, 10
sllv $t1, $a0, $a2
sra $t2, $a0, 10
srav $t3, $a0, $a2
srl $t4, $a0, 10
srlv $t5, $a0, $a2
slt $t6, $a3, $a0
sltu $t7, $a3, $a0
slti $s0, $a3, -10000
sltiu $s1, $a3, 10000
```

4.3 分支测试

```
li $v0, 100
li $v1, 50
li $a1, -50
beq $v0, $v0, beq_test_success
li $t0, 233
beq_test_success:
bne $v0, $v1, bne_test_success
li $t1, 233
bne_test_success:
bgez $zero, bgez_test_success_1
li $t2, 233
bgez_test_success_1:
bgez $v0, bgez_test_success_2
li $t2, 233
bgez_test_success_2:
bgtz $v0, bgtz_test_success
li $t3, 233
bgtz_test_success:
ble $v1, $v0, ble_test_success_1
li $t4, 233
ble_test_success_1:
ble $v1, $v1, ble_test_success
li $t4, 233
ble_test_success:
bltz $a1, bltz_test_success
li $t6, 233
bltz_test_success:
blez $zero, blez_test_success_1
li $t7, 233
blez_test_success_1:
blez $a1, blez_test_success
li $t7, 233
blez_test_success:
```

4.4 循环 + 内存 + 数值运算测试

```
li $s1, 0x2000
li $s2, 0x0
li $s5, 0x200
loop:
add $s3, $s1, $s2
sb $s2, 5($s3)
add $s2, $s2, 1
ble $s2, $s5, loop
```

```
li $s1, 0x2000
li $s2, 0x0
li $s5, 0x200
li $s4, 0x0
li $s7, 0x0
loop2:
add $s3, $s1, $s2
lb $s4, 5($s3)
add $s2, $s2, 1
add $s7, $s4, $s7
ble $s2, $s5, loop2
```

4.5 跳转指令测试

```
jal test
li $a1, 2333
j final
li $s1, 233
test:
li $a2, 233
jr $ra
li $a3, 23333
final:
li $s0, 23333
```

4.6 预期输出

	zero	at	v0	v1	a0	a1	a2	a3
数值运算测试	0	FFFF	0	0	FFFE0000	FFFE0000	FFFF01D2	FFFF0000
比较指令测试	0	FFFE0000	0	0	F4240	1E8480	14	FFFE7960
分支测试	0	0	64	32	0	FFFFFFCE	0	0
循环 + 内存 + 数值运算测试	0	1	0	0	0	0	0	0
跳转指令测试	0	0	0	0	0	91D	E9	0
	t0	**t1**	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**
数值运算测试	E90000	FFFF0000	FFFF0000	E9FFFE	E90000	0	FF170000	FF170000
比较指令测试	3D090000	24000000	3D0	0	3D0	0	1	0
分支测试	0	0	0	0	0	0	0	0
循环 + 内存 + 数值运算测试	0	0	0	0	0	0	0	0
跳转指令测试	0	0	0	0	0	0	0	0
	s0	**s1**	**s2**	**s3**	**s4**	**s5**	**s6**	**s7**
数值运算测试	0	FFFF0001	FFFF0001	FF160000	E9FFFF	FFFF	0	0
比较指令测试	1	0	0	0	0	0	0	0
分支测试	0	0	0	0	0	0	0	0
循环 + 内存 + 数值运算测试	0	2000	201	2200	0	200	0	FFFFFF00
跳转指令测试	5B25	0	0	0	0	0	0	0
	t8	**t9**	**k0**	**k1**	**gp**	**sp**	**fp**	**ra**
数值运算测试	0	0	0	0	0	0	0	0
比较指令测试	0	0	0	0	0	0	0	0
分支测试	0	0	0	0	0	0	0	0
循环 + 内存 + 数值运算测试	0	0	0	0	0	0	0	0
跳转指令测试	0	0	0	0	0	0	0	4

4.7 仿真与测试结果

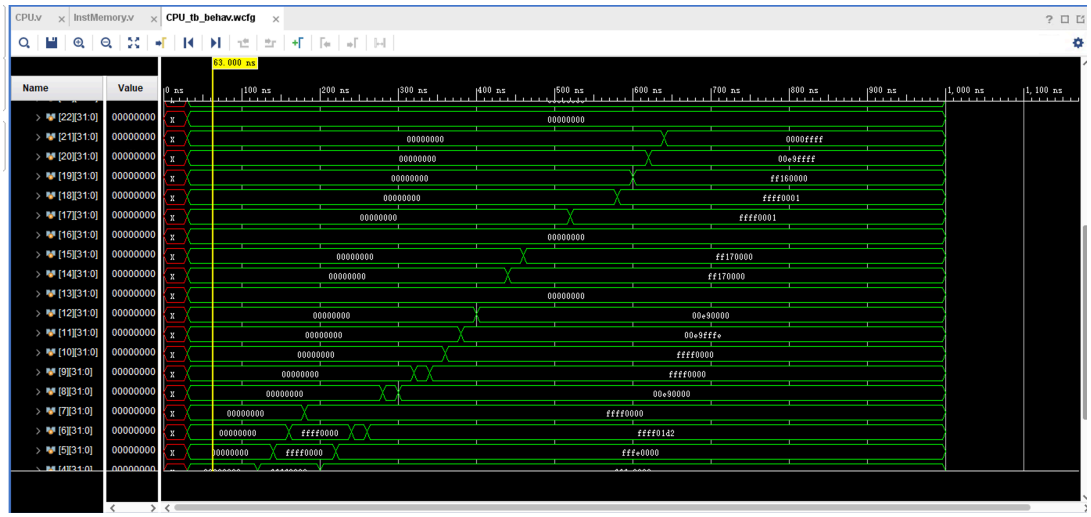

图 1: 数值运算测试结果

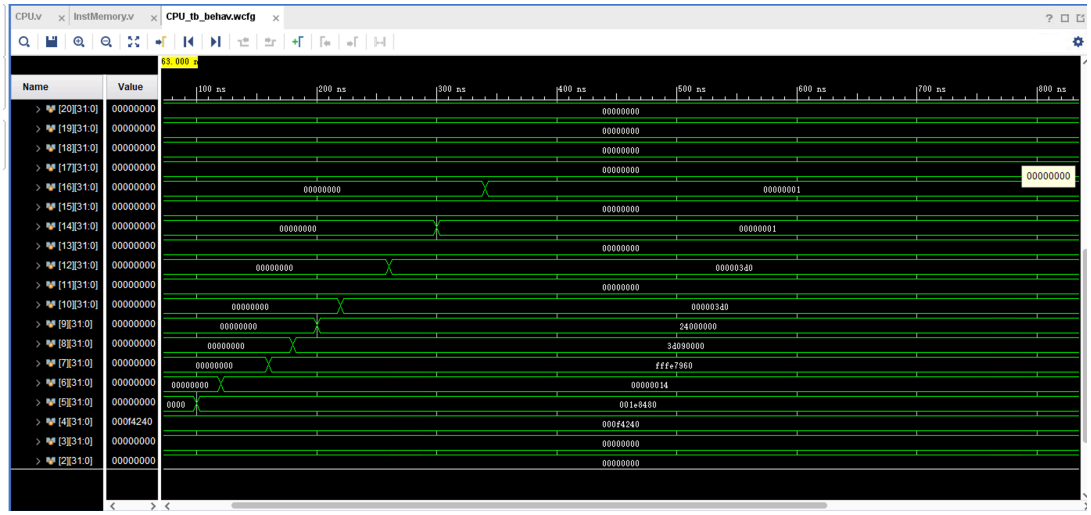

图 2: 比较指令测试结果

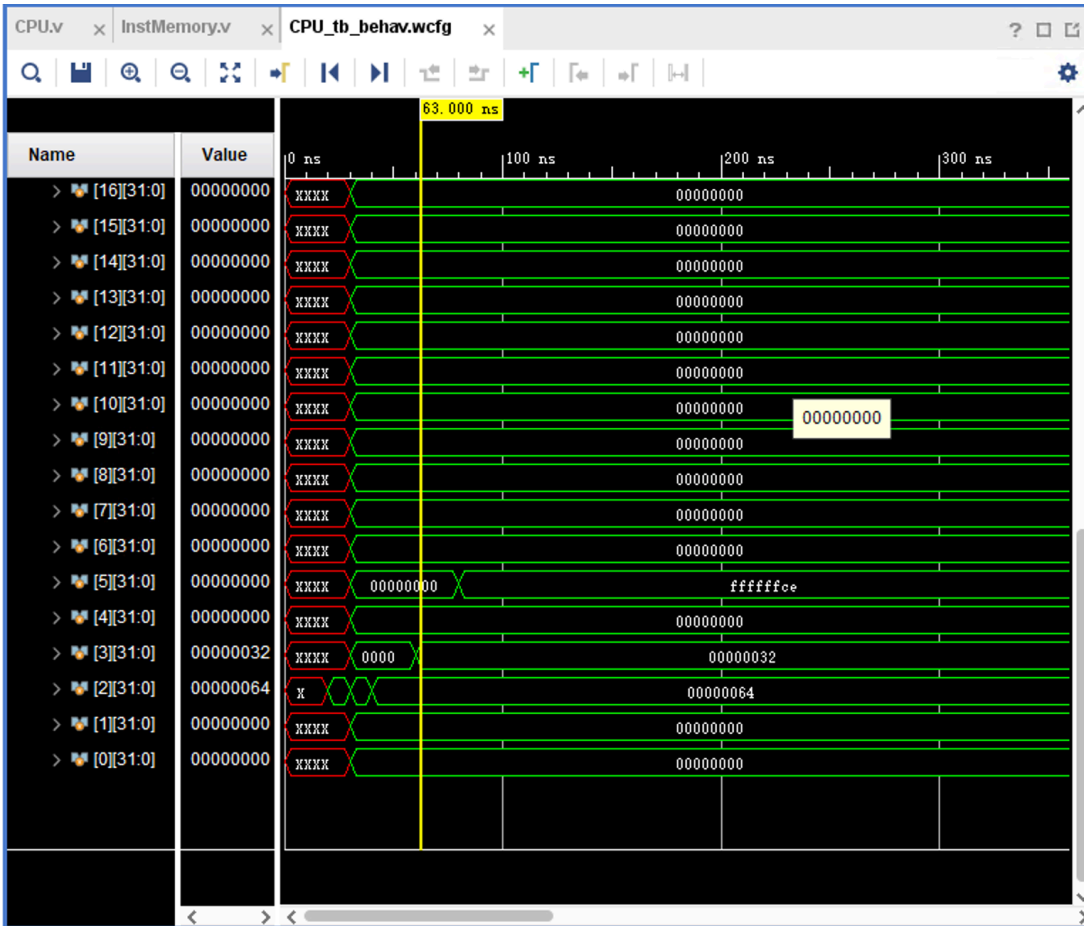

图 3: 分支测试结果

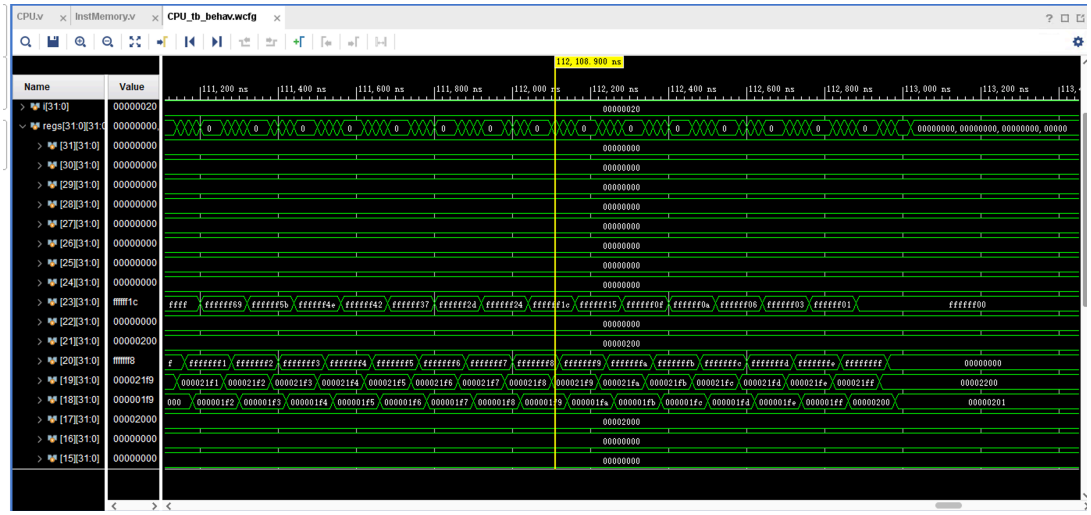

图 4: 循环 + 内存 + 数值运算测试结果

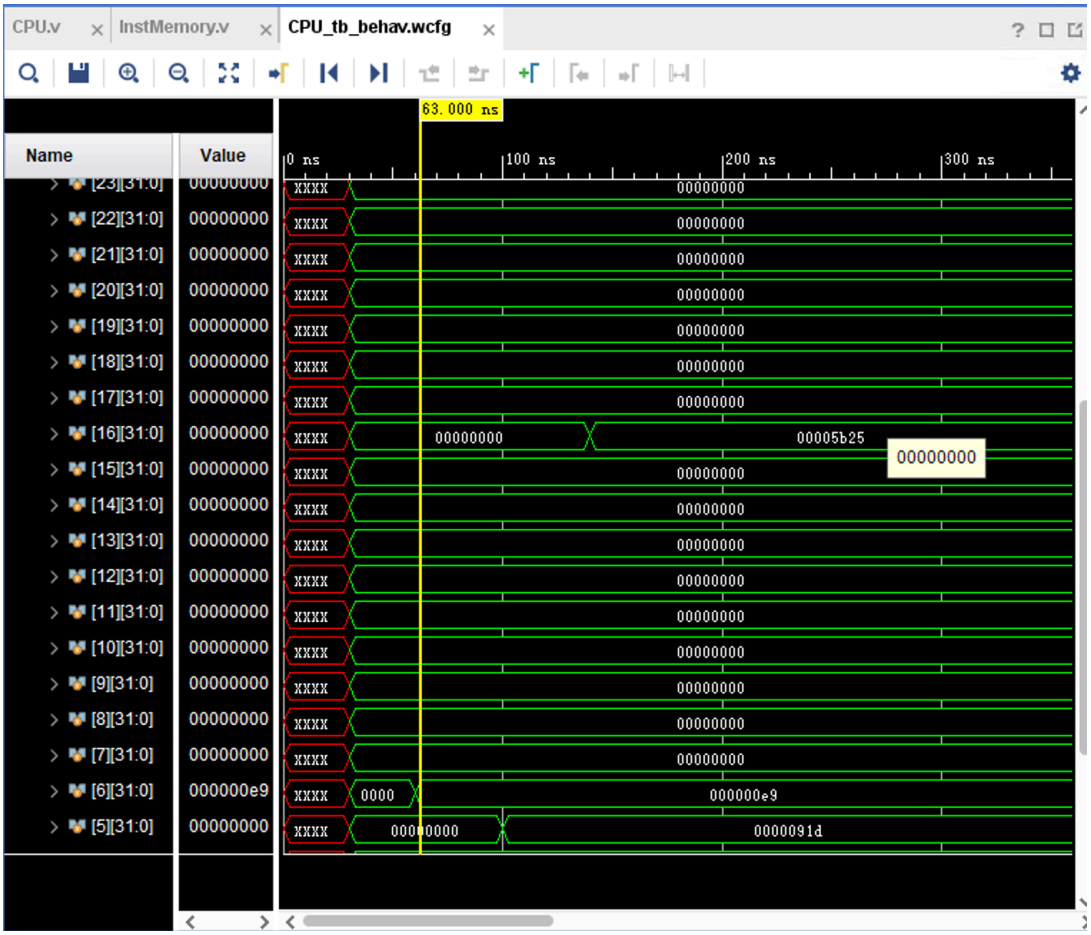

图 5: 跳转指令测试结果

5 感悟

This part is not made public.