第 9 次实验报告
实验 11 多周期处理器实验
22336313 郑鸿鑫

一． 实验准备

前面我们已经完成的了单周期实验，搭建了完整的单周期处理器 datapath。这节课我们开始在单周期实验的基础上，将单周期处理器扩展为多周期处理器。在理论课上，我们讲解如何将单周期改造成多周期处理器，主要需要改动 2 处:(1)在 datapath 中每个 stage（即取指、译码、执行、访存、回写 5 个阶段）中加入寄存器

(2）需要将 controller 由原来的组合逻辑变成状态机；通过前面的实验，单周期 CPU 的设计大家已经完成，CPU datapath 各个通路基本模块均已具备。

准备工作如下:

1.熟悉 Vivado 的仿真功能 (行为仿真)。

2.熟悉课堂上讲的状态机的设计流程（用于实现控制器）。

3.理解多周期处理器的处理流程， 以及控制器的设计流程。

二． 实验目的

1. 掌握多周期 CPU 数据通路及其设计方法；

2. 掌握状态机的设计方法并实现多周期 CPU 控制器；

3. 掌握多周期 CPU 的实现方法， 代码实现方法；

4. 掌握测试多周期 CPU 的方法。

三． 实验设备

PC 机一台， Basys3 开发板， Xilinx Vivado 开发套件。

四． 实验原理

多周期 CPU 指的是将整个 CPU 的执行过程分成几个阶段， 每个阶段用一个时钟去完成， 然后开始下一条指令的执行， 而每种指令执行时所用的时钟数不尽相同， 这就是所谓的多周期 CPU。

CPU 在处理指令时， 一般需要经过以下几个阶段：

(1) 取指令(IF)： 根据程序计数器 pc 中的指令地址， 从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， 则控制器把“转移地址”送入 pc， 当然得到的“地址”需要做些变换才送入 pc。

(2) 指令译码(ID)： 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，从而产生相应的操作控制信号， 用于驱动执行状态中的各种操作。

(3) 指令执行(EXE)： 根据指令译码得到的操作控制信号， 具体地执行指令动作， 然后转移到结果写回状态。

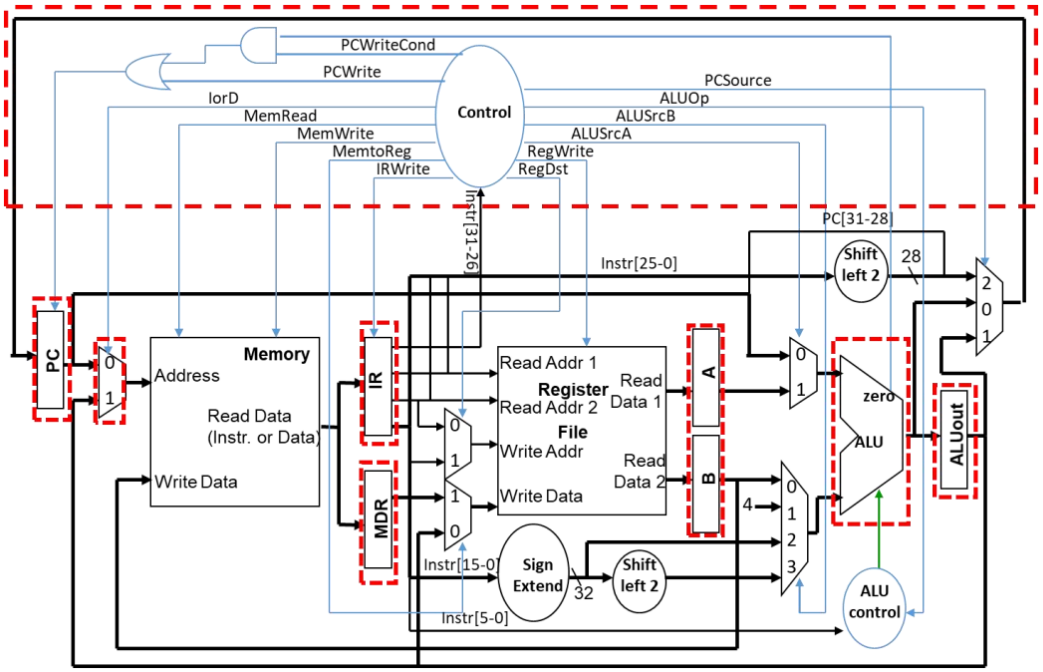
(4) 存储器访问(MEM)： 所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回(WB)： 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。

实验中就按照这五个阶段进行设计，这样一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况要根据该条指令的情况而定，有些指令不需要五个时钟周期的，这就是多周期的 CPU。

五． 实验内容

1. 基于单周期处理器 CPU 的数据通路代码进行改造， 在各个需要插入寄存器组 IR, MDR, A,B, ALUOut（系统框图红色部分）,注意其中 IR 和 PC 寄存器都需要进行使能控制。用于控制在当前指令没有结束的时候，禁止下一条指令导入。



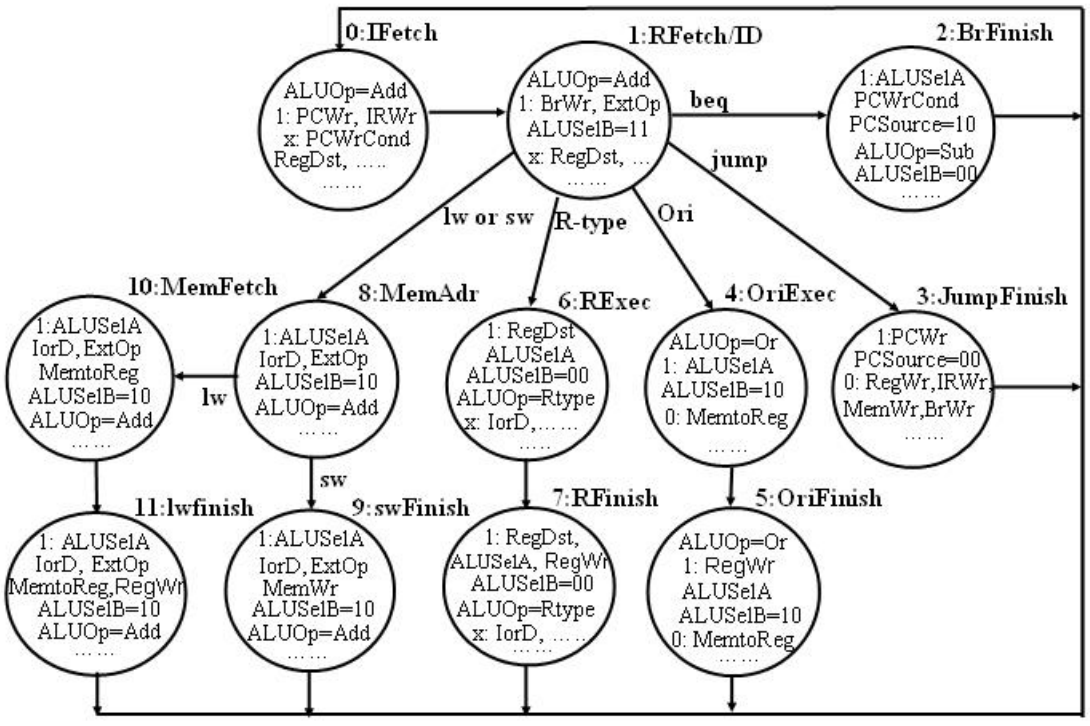
2. 在多周期处理器里面，由于指令存储器 inst_mem(Single Port Rom)和数据存储器。data_mem(Single Port Ram)是在不同周期里面使用的，因此可以分时使用将两个存储器合并；同上次实验一样，使用 BlockMemory Generator IP 构造指令，注意考虑 PC 地址位数统一。

3.修改控制单元，将模块修改为控制器状态机，在不同指令的不同周期数输出不同的控制信号（上图红色框图部分）。

六． 实验步骤

1.根据控制器在每个周期的输出控制信号，根据状态机设计控制器：

不同的指令有不同的周期数， 不同指令在不同的周期（stage）也有不同的输出控制信号。多周期的控制器需要依赖于状态机设计实现。 根据理论课上的讲解内容， 我们可以设计如下的状态机：



注意： 蓝色部分代表的立即数的计算， 不仅仅代表 Ori 指令。图中的圆圈代表不同指令在不同周期的控制信号输出的集合。

根据有限状态机编写 maindec.v 模块代码如下：

各种控制信号的功能已写在注释中

```
module maindec(
input wire clk, rst,
input wire [5:0] op,//指令高6位的 OP Code
input wire zero,
output wire PCWrite,//PC 的写使能信号,为0 时不更改,为1 时更改
output reg [1:0] PC_src,//选择下一条指令的来源
output reg IRWrite,
//IRWrite =0： IR(指令寄存器)不更改；IR 寄存器写使能。向指令存储器发出读
//指令代码后，这个信号也接着发出，在时钟上升沿，IR 接收从指令存储器送来
//的指令代码。每条指令都相关。
RegDst,//寄存器写入地址是rd 还是rt
RegWritem,//Registerfile 写入控制信号,0 为读,1 为写入
output reg ALUSrcA,
//ALUSrcA=0,ALU 的输入来自 PC;ALUSrcA=1,ALU 的输入来自寄存器堆
output reg [1:0] ALUSrcB,
//ALUSrcB=00,ALU 的输入来自寄存器堆
//ALUSrcB=01,ALU 的输入等于4,用于做PC+4
//ALUSrcB=10,ALU 的输入来自与立即数
//ALUSrcB=11,ALU 的输入来自与立即数迁移,主要用于beq 指令
output reg IorD,//选择选指令存储器还是数据存储器
MemtoReg,
//MemtoReg=0, 来自 ALU 的输出写入 registerfile， 例如 R-type 指令
//MemtoReg=1, 来自 Memory 出写入 registerfile, 例如 lw 指令
MemWrite,//读写指令和存储存储器
output reg [1:0] alu_op//ALU8 种功能运算
);
reg [3:0]state,nxt;//记录状态机当前状态和下一个状态
always @(negedge clk, posedge rst) begin
    if(rst)
        state <= 4'b0;
    else
        state <= nxt;
end
always @(*) begin
    case(state)
        0: nxt <= 1;
        1: begin
            case(op)
                6'b000100: nxt <= 2;    // beq
                6'b000010: nxt <= 3;    // jump
                6'b001000: nxt <= 4;    // addi
                6'b000000: nxt <= 6;    // r-type
                6'b100011: nxt <= 8;    // lw
                6'b101011: nxt <= 8;    // sw
                default:   nxt <= 0;    // skip
            endcase
        end
        2: nxt <= 0;
        3: nxt <= 0;
        4: nxt <= 5;
        5: nxt <= 0;
        6: nxt <= 7;
        7: nxt <= 0;
        8: nxt <= op == 6'b100011 ? 10 : 9;
```

```
            9: nxt <= 0;
            10: nxt <= 11;
            11: nxt <= 0;
            default: nxt <= 0;
        endcase
    end
    reg PC_write, PC_wirtecond;//Beq 信号的PC 更新旁路信号
    assign PCWrite = PC_write | (PC_wirtecond & zero);
    //将ALU 的zero 输出和PC_Writecond 相与后再和PC_Write
    相或得到PC 的写使能信号
    always @(*) begin
        if(state == 0 | state == 3)
            PC_write <= 1;
        else
            PC_write <= 0;
        if(state == 0 | state == 1)
            ALUSrcA <= 0;
        else
            ALUSrcA <= 1;
        if(state == 6 | state == 2 |state == 7)
            ALUSrcB <= 0;
        else if(state == 0)
            ALUSrcB <= 1;
        else if(state == 1)
            ALUSrcB <= 3;
        else
            ALUSrcB <= 2;
        if(state == 11)
            MemtoReg <= 1;
        else
            MemtoReg <= 0;
        if(state == 5 | state == 7 | state == 11)
            RegWritem <= 1;
        else
            RegWritem <= 0;
        if(state == 7)
            RegDst <= 1;
        else
            RegDst <= 0;
        if(state == 9)
            MemWrite <= 1;
        else
            MemWrite <= 0;
        if(state == 0)
            IRWrite <= 1;
        else
            IRWrite <= 0;
        // extop <= 1;
        if(state == 3)
            PC_src <= 2;
        else if(state == 2)
            PC_src <= 1;
        else
            PC_src <= 0;
        if(state == 6 | state == 7)
            alu_op <= 2'b10;
```

```
        else if(state == 3)
            alu_op <= 2'b11;
        else if(state == 2)
            alu_op <= 2'b01;
        else
            alu_op <= 2'b00;
        if(state == 2)
            PC_wirtecond <= 1;
        else
            PC_wirtecond <= 0;
        if(state == 9 | state == 10)
            IorD <= 1;
        else
            IorD <= 0;
    end
endmodule
```

2.从上一次实验中，导入各个模块，并按照五.1 中的图进行改写 datapath：（详细说明已写在注释中）

改写 datapath 模块的代码如下：

```
module datapath(
    input wire clk, rst,
    input wire [4:0] debug_addr,
    input wire [31:0] mem_rd,
    input wire PC_we,
    input wire [1:0] PC_src,
    input wire IR_we, reg_dst, reg_we,
    input wire alu_srcA,
    input wire [1:0] alu_srcB,
    input wire [2:0] alu_ctr,
    input wire IorD, memtoreg,
    output wire [31:0] mem_addr,
    output wire [31:0] mem_wd,
    output wire [31:0] instr,
    output wire [31:0] debug_data,
    output wire overflow, zero
    );
    // IF
    wire [31:0] PC_nxt,//下一条PC
    PC_addr;//从PC 寄存器中读出的值，作为mem_addr 的第0 种选择
    REG PC(clk, rst, PC_we, PC_nxt, PC_addr);//PC 寄存器
    REG IR(~clk, rst, IR_we, mem_rd, instr);//IR 寄存器
    // ID
    wire [4:0] reg_wa;//写入的寄存器的编号
    wire [31:0] reg_wd,//写入寄存器的数据
    reg_rd1,//寄存器1 中读到的值
    reg_rd2,//寄存器2 中读到的值
    rd1R_out,//从A 寄存器中得到的值
    rd2R_out;//从B 寄存器中得到的值，作为ALU 第二个操作数的第0 种选择
    //reg_dst 用于选择是写回rt 还是rd 字段的寄存器
    mux2 #(5) reg_wamux(instr[20:16], instr[15:11],reg_dst,reg_wa);
```

```
    regfile Reg(~clk, reg_we, instr[25:21],
instr[20:16],reg_wa,reg_wd,reg_rd1,reg_rd2,debug_addr,debug_data);
    REG rd1R(clk, rst, 1'b1, reg_rd1, rd1R_out);
    REG rd2R(clk, rst, 1'b1, reg_rd2, rd2R_out);
    //由于不必控制写使能，所以直接置为1
    // EXE
    wire [31:0] alu_A, //ALU 的第一个操作数
    alu_B,//ALU 的第二个操作数
    alu_out,
    //ALU 的输出，作为PC_nxt 的第0 种选择,也作为mem_addr 的第一种选择
    aluR_out;
    //ALU 寄存器读出的值，作为PC_nxt 的第1 种选择,也是reg_wd 的第0 种选择
    mux2 alu_muxA(PC_addr,rd1R_out,alu_srcA,alu_A);
    //选择ALU 的第一个操作数
    wire [31:0] imm,//符号扩展后得到的立即数，为ALU 第二个操作数的第2 种选择
    boffset;//imm 左移两位得到，为ALU 第二个操作数的第3 种选择
    signext alu_sxt(instr[15:0], imm);//对指令低16 位进行有符号符号扩展
    sl2 alu_sl2(imm, boffset);//将立即数左移两位，将偏移数转化位字节数
    Mux4 alu_muxB(alu_srcB, rd2R_out, 4, imm, boffset, alu_B);
    //选择ALU 的第二个操作数
    alu alu(alu_A, alu_B, alu_ctr, alu_out, overflow, zero);//ALU
    REG aluR(clk, rst, 1'b1, alu_out, aluR_out);//ALU 寄存器

    // PC addr
    // {{PC_addr[31:28]}, {instr[25:0]}, 2'b00}是将指令低25 位左移后和PC
的高4 位拼接得到，作为PC_nxt 的第2 种选择
    //wire [31:0] PC_jaddr;
    Mux4 PC_mux(PC_src, alu_out, aluR_out, {{PC_addr[31:28]},
{instr[25:0]}, 2'b00}, 0, PC_nxt);//选择下一条PC
    // MEM
    mux2 mem_mux(PC_addr,aluR_out,IorD,mem_addr);
    //选择哪个作为Memory 的地址输入
    assign mem_wd = rd2R_out;
    wire [31:0] MDR_out;
    //MDR 寄存器中读到的值，作为寄存器堆写回数据的第1 种选择
    REG MDR(~clk, rst, 1'b1, mem_rd, MDR_out);//MDR 寄存器
    // WB
    mux2 reg_wdmux(aluR_out, MDR_out,memtoreg, reg_wd);
    //选择写回寄存器堆的数据
endmodule
```

3.根据前面的存储器实验使用 Block Memory，生成统一的 memory，在 top 模块中调用 memory 如下：

```
module top(
    input wire clk,rst,
    output wire [31:0]writedata,mem_addr,
    output wire memwrite,
    input wire [3:0] ra_debug_button
    // input [15:0] switch_value,
    // output [3:0] select,
    // output [6:0] seg
    );
```

```
    wire [31:0] rd_debug;
    wire [4:0] ra_debug;
    // assign ra_debug = {1'b0,ra_debug_button};
    wire[31:0] readdata,instr,readdata_mem,outdata;
    //clock_div CC(clk,CLK);//时钟分频器
    mips
mips(clk,rst,mem_addr,memwrite,writedata,readdata,ra_debug,rd_debug);
    // Inst_Rom imem(CLK,mem_addr[9:2],instr);
    // Data_Ram dmem(~CLK,memwrite,dataadr,writedata,readdata_mem);
    Memory mem(
    .clka(clk),    // input wire clka
    .wea(memwrite),      // input wire [0 : 0] wea
    .addra(mem_addr[9:2]),  // input wire [7 : 0] addra
    .dina(writedata),    // input wire [31 : 0] dina
    .douta(readdata)  // output wire [31 : 0] douta
    );
    // assign readdata = (mem_addr[14] == 0) ? readdata_mem :
{16'b0,switch_value};
    //用于检查当前访问的内存区域是存储器空间还是外设空间并选通对应的数据
    //display U2(clk,rst,rd_debug[15:0],seg,select);
    //将寄存器中的值的低 16 位在 7 段数码管上显示
endmodule
```

4. 参考实验原理，连接各模块；

5. 导入顶层文件及仿真文件，运行仿真。

仿真文件代码如下:

```
module testbench();
    reg clk;
    reg rst;
    reg [3:0] ra_debug_buttton;
    reg [15:0] switch_value;
    // wire [3:0] select;
    // wire [6:0] seg;
    wire[31:0] writedata,dataadr;
    wire memwrite;
    top dut(clk,rst,writedata,dataadr,memwrite,ra_debug_buttton);
    //top dut(clk,rst,ra_debug_buttton,switch_value,select,seg);
    // initial begin
    //  ra_debug_buttton = 4'b0001;
    //  switch_value = 16'b0000_0000_0000_0000;
    // end
    initial begin
        rst <= 1;
        #100;
        rst <= 0;
    end
    always begin
        clk <= 1;
        #5;
        clk <= 0;
        #5;
    end
```

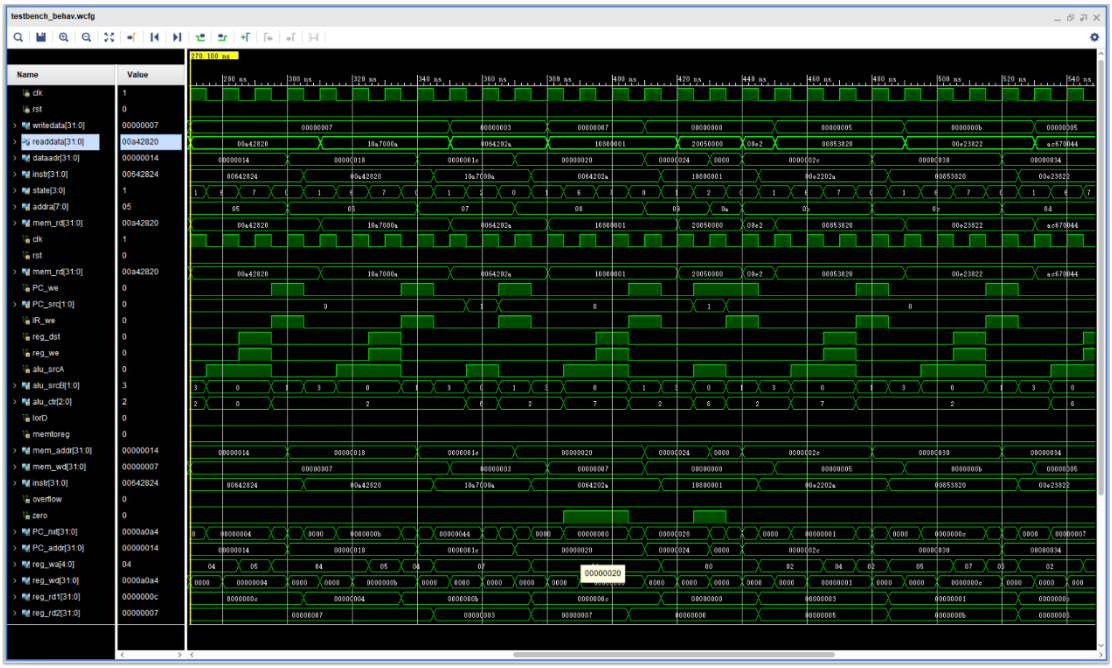
```
    always @(negedge clk) begin
        if(memwrite) begin
            if(dataadr === 84 && writedata === 7) begin
                /* code */
                $display("Simulation succeeded");
                $stop;
            end
        end
    end
endmodule
```

七． 实验结果

在存储器中存入以下指令

#	Assembly	Description	Address	Machine	
main:	addi $2, $0, 5	# initialize $2 = 5	0	20020005	20020005
	addi $3, $0, 12	# initialize $3 = 12	4	2003000c	2003000c
	addi $7, $3, -9	# initialize $7 = 3	8	2067fff7	2067fff7
	or $4, $7, $2	# $4 <= 3 or 5 = 7	c	00e22025	00e22025
	and $5, $3, $4	# $5 <= 12 and 7 = 4	10	00642824	00642824
	add $5, $5, $4	# $5 = 4 + 7 = 11	14	00a42820	00a42820
	beq $5, $7, end	# shouldn't be taken	18	10a7000a	10a7000a
	slt $4, $3, $4	# $4 = 12 < 7 = 0	1c	0064202a	0064202a
	beq $4, $0, around	# should be taken	20	10800001	10800001
	addi $5, $0, 0	# shouldn't happen	24	20050000	20050000
around:	slt $4, $7, $2	# $4 = 3 < 5 = 1	28	00e2202a	00e2202a
	add $7, $4, $5	# $7 = 1 + 11 = 12	2c	00853820	00853820
	sub $7, $7, $2	# $7 = 12 - 5 = 7	30	00e23822	00e23822
	sw $7, 68($3)	# [80] = 7	34	ac670044	ac670044
	lw $2, 80($0)	# $2 = [80] = 7	38	8c020050	8c020050
	j end	# should be taken	3c	08000011	08000011
	addi $2, $0, 1	# shouldn't happen	40	20020001	20020001
end:	sw $2, 84($0)	# write adr 84 = 7	44	ac020054	ac020054

经过综合确认无误后，进行仿真，结果如下所示：



下面是对仿真波形图的检查说明:

1.第一条指令 20020005 对应的汇编指令是一条 I 型指令，代表将 0 号寄存器的值 0 和立即数 5 相加赋给 2 号寄存器，可以看到 a 为 0， b 为 5 是 ALU 的两个源操作数， aluout 和 writedata 为 5，和预期相符合;

2. 接下来两条指令也是 I 型指令，同理可知符合预期;

3.接下来，两个读寄存器分别为 7 号和 2 号，写回寄存器选择的是 4 号， aluout 为 7（3 和 5 按位或的结果），和预期符合；

4.接下来 add 指令和 and 指令也同理，检查知符合实验预期；

5.然后到第一条 beq 指令，由仿真图中可以看出 10a7000a 的下一条指令是 0064202a，并未发生跳转，而是采用 PC+4 为下一条指令的地址，这是因为 5 号寄存器中存的是 11， 7 号寄存器中存的是 3，所以不相等不产生跳转， 符合预期；

6.可以看到第二条 beq 指令是在 PC 等于 20 的时候，此时检查 4 号寄存器中的值是否为 0，发现恰好为 0，所以应该跳转，由仿真图可以看出，该指令 10800001 的下一条是 00e2202a，跳过了它原本的下一条指令 20050000，跳到 around 地址对应的指令，也符合预期；

7.接下来可以看到 sw 指令，此时 writedata 变为 7， memwrite 变为 1，符合预期；

8.紧接着 lw 指令 readdata 变为 7 为相应访存的位置的值，其余时间 readdata 都为 0，也符合预期；

9.最后一个检查的点是 j 指令，当 PC 为 3c，指令为 08000011 时，需要跳过它的下一条指令 20020001，跳转到 end 的位置，对应 ac020054，观察仿真波形图可知，成功跳过，符合预期。