计算机组成原理 实验报告

姓名：龚小航　学号：PB18151866　实验日期：2020-5-13

一、实验题目：

Lab04 多周期 CPU

二、实验目的

- 理解计算机硬件的基本组成、结构和工作原理；
- 掌握数字系统的设计和调试方法；
- 熟练掌握数据通路和控制器的设计和描述方法；
- 具体目标：

设计多周期 CPU,能够执行六条指令：add，addi，lw，sw，beq，j. 结构化描述多周期 CPU 的数据通路和控制器，并进行功能仿真；再连接加入调试单元 DBU，通过调试单元进行功能仿真。调试单元需要将输出量显示在数码管和 16 个 LED 灯上，以便实物验证。

三、实验平台：

Vivado

四、实验过程：

1.设计多周期 CPU（不包含调试单元）：

先列出该部分的输入输出关系，多周期 CPU 逻辑图如下所示：

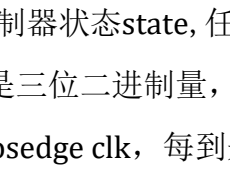
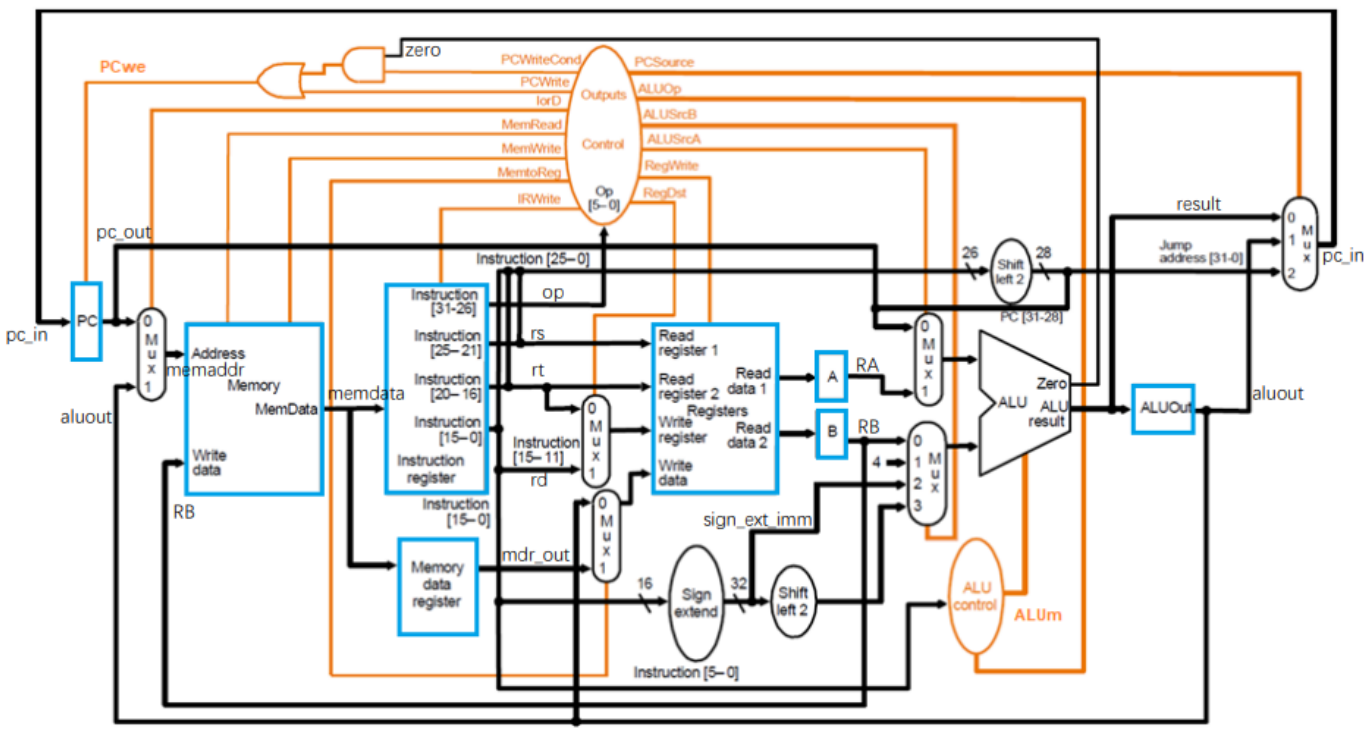


图 1 多周期 CPU 逻辑图

引脚说明：clk 为时钟信号，一个时钟脉冲做一个多周期阶段，上升沿有效；rst 为异步复位信号，上升沿有效。

CPU 内部的逻辑图如下所示：



图中所有模块全部采用例化实现。顶层模块 TOP 仅声明信号且例化模块，而 TOP 中的线网型信号的名称也标于图中。其中，指令和数据共用一个单端口 RAM，该存储器采用例化 IP 核的的方式实现，深度为 512，位宽 32bit.

以下先简要说明多周期 CPU 工作过程：

- 启动之前：利用.coe 文件对存储器进行初始化赋值，即为将需要运行的测试程序代码与数据存入 RAM 存储器；同时将 pc 置 0(由具体的程序入口确定)。由于指令和数据共用存储器，更好的方法是将所有指令代码放在低 256 位，数据放在高 256 位，pc 为 7 位，传入 RAM 的地址为{IorD,pc},这样可以极大的改进程序的可移植性，因为跳转、分支指令都只需要考虑指令部分占用地址。这和单周期 CPU 把指令、数据存储器分开是同样的效果。由于本次测试用例已经给出，就将指令和数据一起存放在低地址即可。
- 初始化完成，时钟接入，CPU 开始运行： 多周期 CPU 的运行与单周期不同。对于多周期 CPU，决定 CPU 状态的是控制器状态state,任一时刻控制器必处于五个状态之一：IF, ID, EXE, MEM, WB. 而state是三位二进制量，多余的状态全部归并入IF。唯一能控制状态转换的是时钟上升沿posedge clk，每到来一个时钟脉冲控制器就更改一次状态，而下一个状态是由当前状态以及当前指令决定的。状态的控制能实现一条指令做完就能将下个周期设为取指，开始新的指令。相比于单周期，多周期 CPU 每一个周期能非常短（需要的操作少），而指令需要多少周期就用多少，只要合适的选取时钟周期，CPU 空余时间就可以明显减少。

• Control Unit 是有限状态机，在不同的状态做不同的事。

控制信号说明：

PCWriteCond:分支指令标志，若指令为 BEQ，在 EX 阶段将其设为 1;

PCWrite:跳转标志，若指令为 J，在 EX 阶段将其设为 1;

IorD:选择传入存储器 RAM 的地址来源，来自 pc(0)或是 ALUOut(1);

MemWrite: 存储器写使能信号，为高电平时允许在下个时钟上升沿写入;

MemtoReg: 选择写回寄存器的数据来自 ALUOut（0）或是存储器数据寄存器(1);

IRWrite: 当此信号为 1 时指令寄存器数据才可被更新。仅当 IF 段此信号为 1.

PCSourse(2 位): 选择送入 PC 的值的来源（BEQ/pc+4/J）

ALUop(3 位): ALU 的操作码.由于本例只有六条指令，令其始终为 000（+）即可;

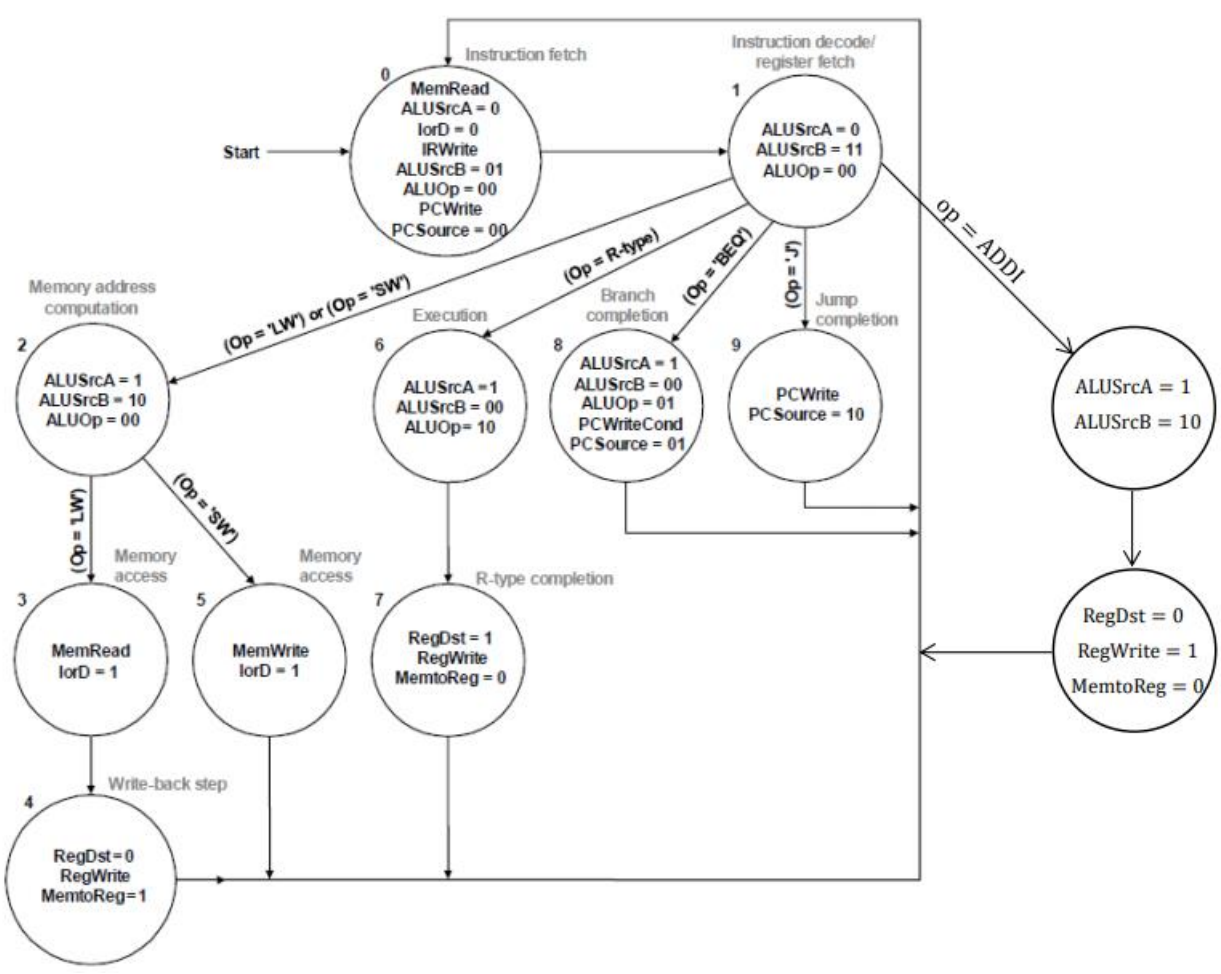
ALUSrcB(2 位): 选择 ALU 的第二个操作数(RB/4/符号扩展/符号扩展后左移 2 位);

ALUSrcA: 选择 ALU 的第一个操作数(pc/RA);

RegWrite: 寄存器堆写使能，高电平时允许在下个时钟上升沿写入;

RegDst: 选择写目标寄存器来自 rt (RegDst = 0)还是 rd (RegDst = 1);

以下是控制器的状态转换图：



系统会根据指令和状态来进行操作。每个状态根据当前的指令来决定需要做哪些操作，并根据现在的状态和指令确定次态。

• 在下一个时钟上升沿来临时：控制器进入新的状态。

在具体实现每个模块的时候，需要注意：寄存器堆 0 号寄存器禁用赋值功能；需要时钟同步的模块有 pc，寄存器堆，RAM 存储器，存储器数据寄存器，指令寄存器 RA,RB，ALUOut，其余均以组合逻辑输出。

具体的代码附于源文件中，并标有注释。此处不需要具体展开。

仅贴出顶层模块的连接以及控制器模块的实现，这是整个程序的总体框架。

```
module TOP(
    input clk,
    input rst
    );

    wire PCwe;
    wire [31:0] pc_in, pc_out, aluout, memdata, RA, RB, mdr_out, writedata, sign_ext_imm, rdA, rdB, result, alu_in0,alu_in1;
    wire [7:0] memaddr;//未加最高位，最高位=1为数据，=0为指令

    wire PCWriteCond, PCWrite, IorD, MemWrite,  MemtoReg, IRWrite, ALUSrcA, RegWrite, RegDst,  zero;
    wire [1:0] PCSourse,ALUSrcB;
    wire [2:0] ALUop;

    wire [5:0] op;
    wire [4:0] rs, rt, rd, writeregister;
    wire [15:0] addr_immediate;
    wire [27:0] jumpaddr_28bit;

    PC programcounter(clk,rst,PCwe,pc_in,pc_out);
    MUX_2 #(8) MUX_IorD(.in_0(pc_out[9:2]), .in_1(aluout[9:2]), .sel(IorD), .out(memaddr));
    Memory MEM (clk, {1'b0,memaddr}, MemWrite, RB, memdata);
    INS_Register INS_R (clk,rst,memdata,IRWrite, op,rs,rt,rd,addr_immediate,jumpaddr_28bit);
    MEM_DATA_REGISTER MDR (clk,memdata,mdr_out);
    MUX_2 #(5) MUX_RegDst(.in_0(rt), .in_1(rd), .sel(RegDst), .out(writeregister));
    MUX_2 #(32) MUX_MemtoReg(.in_0(aluout), .in_1(mdr_out), .sel(MemtoReg), .out(writedata));
    Sign_Extend SE (addr_immediate,sign_ext_imm);
    Register_File #(32) RF (clk,rs,rdA,rt,rdB,writeregister,RegWrite,writedata);
    REG_32Bit A (clk,rst,rdA,RA);
    REG_32Bit B (clk,rst,rdB,RB);

    MUX_2 #(32) MUX_ALUSrcA (.in_0(pc_out), .in_1(RA), .sel(ALUSrcA), .out(alu_in0));
    MUX_4 #(32) MUX_ALUSrcB (.in_0(RB), .in_1(32'h00000004), .in_2(sign_ext_imm), .in_3(sign_ext_imm<<2), .sel(ALUSrcB), .out(alu_in1));
    ALU #(32) alu (.in0(alu_in0), .in1(alu_in1), .ALUop(ALUop), .zero(zero), .result(result));
    REG_32Bit ALUOut (clk,rst,result,aluout);
    MUX_4 #(32) MUX_PCSourse (.in_0(result), .in_1(aluout), .in_2({pc_out[31:28],jumpaddr_28bit}), .in_3(32'h0), .sel(PCSourse), .out(pc_in));
     wire ppcc;
    //and A1 (ppcc,zero,PCWriteCond);
    //or A2 (PCwe,ppcc,PCWrite);

    assign ppcc = zero && PCWriteCond;
    assign PCwe = ppcc || PCWrite;

    Control_Unit control (clk,rst,op,PCWriteCond, PCWrite, IorD, MemWrite,  MemtoReg, IRWrite, PCSourse, ALUop, ALUSrcB, ALUSrcA, RegWrite, RegDst);


endmodule

```

可见 TOP 模块仅仅是对所有用到模块的调用例化，完全按照逻辑图连线设计。TOP 模块是程序的整体视角，本身不关心每个功能块具体怎么实现。

以下为控制器模块，先展示第一部分：输入输出信号与内部信号定义及初始化。

```
module Control_Unit(
    input clk,
    input rst,
    input [5:0] op,

    output PCWriteCond,
    output PCWrite,
    output IorD,
    //output MemRead
    output MemWrite,
    output MemtoReg,
    output IRWrite,
    output [1:0] PCSourse,
    output [2:0] ALUop,
    output [1:0] ALUSrcB,
    output ALUSrcA,
    output RegWrite,
    output RegDst

    );

    localparam [5:0] ADD=6'b000000, ADDI=6'b001000, LW=6'b100011, SW=6'b101011, BEQ=6'b000100, J=6'b000010;
    localparam [2:0] IF=3'b000,ID=3'b001,EX=3'b010,MEM=3'b011,WB=3'b100;

    reg [2:0] state,next_state;
    reg  PCWriteCond_reg, PCWrite_reg, IorD_reg, MemWrite_reg,  MemtoReg_reg, IRWrite_reg, ALUSrcA_reg, RegWrite_reg, RegDst_reg;
    reg [1:0] PCSourse_reg;
    reg [2:0] ALUop_reg;
    reg [1:0] ALUSrcB_reg;

    initial begin
       PCWriteCond_reg=0; PCWrite_reg=0; IorD_reg=0; MemWrite_reg=0;  MemtoReg_reg=0; IRWrite_reg=0; ALUSrcA_reg=0; RegWrite_reg=0; RegDst_reg=0;
       PCSourse_reg=0; ALUop_reg=0; ALUSrcB_reg=0;
       state=IF;
    end
```

这里定义了两组局部常量，利用 localparam 可以让程序更直观并且符合人的认识。之后是定义了输出的控制信号的寄存器形式，并对它们赋初值 0（仅在仿真时生效）。

之后是状态转换条件和声明线网型输出信号。在时钟边沿将次态赋给现态，并且若有复位信号，则立即将控制器状态置为取指：

```
    //时钟边沿将次态赋给现态
    always@(posedge clk or posedge rst)begin
      if(rst) state<=IF;
      else state<=next_state;
    end



    //将寄存器型数据按线网型输出.
    assign PCWriteCond=PCWriteCond_reg;
    assign PCWrite=PCWrite_reg;
    assign IorD=IorD_reg;
    assign MemWrite=MemWrite_reg;
    assign MemtoReg=MemtoReg_reg;
    assign IRWrite=IRWrite_reg;
    assign ALUSrcA=ALUSrcA_reg;
    assign RegWrite=RegWrite_reg;
    assign RegDst=RegDst_reg;

    assign PCSourse=PCSourse_reg;
    assign ALUop=ALUop_reg;
    assign ALUSrcB=ALUSrcB_reg;
```

至此，还需要一个 always 块，用于描述状态机的输出。这是一个米利型状态机，输出由输入和状态共同决定。输入量为当前的指令类型，现态为当前状态 state，输出量为次态 next_state 以及 12 个寄存器型控制信号 xxx_reg。取指和译码阶段所有指令执行相同的操作，输出控制信号与次态都相同：

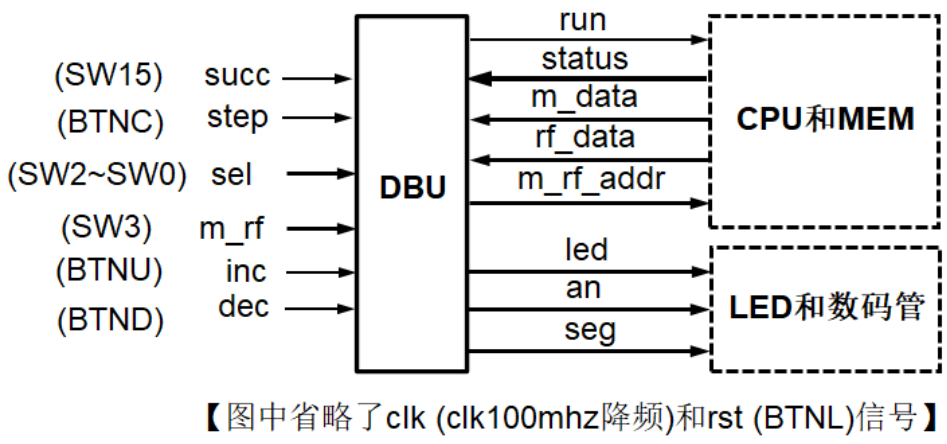
```
  always@(*)begin //描述输出
    case(state)
       IF: begin   //执行pc<=pc+4,  IR=Mem[pc] 关键信号用缩进表示。
          /*PCWriteCond_reg <= 0;*/
             PCWrite_reg <= 1;
             IorD_reg <= 0;
          MemWrite_reg <= 0;
        /*  MemtoReg_reg <= 0; */
             IRWrite_reg <= 1;
             PCSourse_reg <= 2'b00;
             ALUop_reg <= 3'b000;
             ALUSrcB_reg <= 2'b01;
             ALUSrcA_reg <= 0;
          RegWrite_reg <= 0;
          /*RegDst_reg <= 0;*/

          next_state <= ID;
        end

       ID: begin  //译码阶段，完成A、B寄存器和ALUout寄存器的赋值
          PCWriteCond_reg <= 0;
          PCWrite_reg <= 0;
          IorD_reg <= 0;
          MemWrite_reg <= 0;
          MemtoReg_reg <= 0;
          IRWrite_reg <= 0;
          PCSourse_reg <= 2'b00;
             ALUop_reg <= 3'b000;
             ALUSrcB_reg <= 2'b11;
             ALUSrcA_reg <= 0;
          RegWrite_reg <= 0;
          RegDst_reg <= 0;

          next_state <= EX;
        end
```

接下来的 EXE,MEM,WB 三个阶段输出均与当前指令有关，根据状态转换图，分别列出输出信号的值即可。这部分比较冗长，具体代码附于源码之中。

以上就是整个控制器的内容，只要严格按照状态转换图，把它转化成 verilog 代码即可。

2. 设计调试单元 DBU：

先列出该部分的输入输出关系，调试单元 DBU 的逻辑图如下所示：



【图中省略了clk (clk100mhz降频)和rst (BTNL)信号】

图 2 调试单元端口及其与 CPU 连接逻辑图

引脚说明：

- succ：1 位，控制 CPU 的运行方式。Succ==1 则连续执行指令，否则每按动 step 一次，DBU 输出维持一个周期的高电平。
- sel：3 位， 根据不同的值可以选择输出的 CPU 内部的不同内容。

sel = 0：查看 CPU 运行结果 (存储器或者寄存器堆内容)

m_rf： 1，查看存储器(MEM)；0，查看寄存器堆(RF)

m_rf_addr： MEM/RF 的调试读口地址(字地址)，复位时为零

inc/dec：m_rf_addr 加 1 或减 1

rf_data/m_data：从 RF/MEM 读取的数据字

16 个 LED 指示灯显示 m_rf_addr

8 个数码管显示 rf_data/m_data

sel = 1 ~ 7：查看 CPU 运行状态（status）

16 个 LED 指示灯 (SW15~SW0) 依次显示控制器的控制信号 PCSource(2)、PCwe、IorD、MemWrite、IRWrite、RegDst、MemtoReg、RegWrite、ALUm(3)、ALUSrcA、ALUSrcB(2) 和 ALUZero 8 个数码管显示由 sel 选择的一个 32 位数据

sel = 1：PC，程序计数器

sel = 2：IR，指令寄存器

sel = 3：MD, 存储器读出数据寄存器

sel = 4：RA, 寄存器堆读出寄存器 RA

sel = 5：RB, 寄存器堆读出寄存器 RB

sel = 6：ALUOut, ALU 运算结果寄存器

sel = 7：无操作

CPU 的运行完全受到 DBU 的控制，在实体的开发板中，各个信号的输入依靠按钮来实现。而内部信号能依靠数码管和 LED 灯观察到，只有这样才可以说明多周期 CPU 功能的完整性和正确性。

由于增加了很多输出，原本的 CPU 模块必须增加必要的输出信号。为使寄存器堆和存储器内的数据可以被 DBU 随时调用，必须新增一个异步读取端口。寄存器堆增加一个读口比较容易，直接在代码中增加一组读取的输入输出即可；而为了异步读取数据存储器中的任一内容，就必须将例化的单端口 RAM 更改为双端口 RAM，并且第二组端口 dpra,dpo 只允许读取操作而不允许写入。再将各需要的信号量都作为模块输出，传给 DBU 模块即可。

在 DBU 模块中，将 CPU 模块传入的数据作为输入，在根据设计的功能把这些信号显示在数码管或是 LED 灯上即可。

以下简略的说明 DBU 模块的实现：

首先是 DBU 模块的输入输出部分，如下图：

```
23  module DBU(
24    input clk,
25    input rst,
26    input succ,   //这个信号只要为1，对CPU来说立刻就能执行完程序
27    input step,
28    input [2:0] sel,
29    input m_rf,
30    input inc,
31    input dec,
32
33    output[11:0] led,
34    output reg [7:0] SSEG_CA,//数码管输出部分
35    output reg  [7:0] SSEG_AN
36      );
```

succ 信号通过开关 sw15 输入，因此对 CPU 时钟来说，只要某时刻测试者把 sw15 拨动到 1，立刻就有成千上万的时钟周期经过，立刻就能执行完程序。因此这个信号去抖动与否都没有关系，而按照功能要求，也不可以取边沿。而另外的按钮、开关输入均需要去抖动和取边沿处理。在本例中，通过调用模块直接生成处理后的信号，这个信号才可以用于下面的逻辑中去。

对于 succ 和 m_rf_addr 的描述可以用以下的两个 always 块：

```
43    wire clk_DBU; reg clkdbu_reg;
44
45    always@(posedge clk)begin   //描述succ
46      if(succ) clkdbu_reg<=clk;
47      else clkdbu_reg<=stepEdge;
48    end
49     assign clk_DBU=clkdbu_reg;
50
51    reg [7:0] m_rf_addr_reg; wire [7:0] m_rf_addr;
52    initial m_rf_addr_reg=0;
53    always@(posedge clk)begin   //描述 m_rf_addr
54      if(rst) m_rf_addr_reg<=0;
55      else begin
56         if(incEdge) m_rf_addr_reg<=m_rf_addr_reg+1;
57         else if(decEdge) m_rf_addr_reg<=m_rf_addr_reg-1;
58         else m_rf_addr_reg<=m_rf_addr_reg;
59      end
60    end
61     assign m_rf_addr = m_rf_addr_reg;
```

其中 clk-DBU 是传入 CPU 模块的 clk 信号。

接下来就是 sel 选择时，其他输出信号的处理。用一个 always 组合逻辑描述：

```
76    wire [31:0] rf_data,m_data;
77    wire [31:0] pc_out_DBU;//传入和传出pc的内容
78    wire [31:0] IR_DBU;//指令存储器输出的数据
79    wire [31:0] MD_DBU;//寄存器堆读口1的数据
80    wire [31:0] rf_A_DBU;//寄存器堆读口2的数据
81    wire [31:0] rf_B_DBU;//ALU的运算结果
82    wire [31:0] aluout_DBU;
83
84    wire [1:0] PCSourse,ALUSrcB;
85    wire PCwe,IorD,MemWrite,IRWrite,RegDst,MemtoReg,RegWrite;
86    wire [2:0] ALUop;
87    wire ALUSrcA,ALUZero;
88
89    TOP test (clk_DBU, rst, m_rf_addr, m_data,rf_data,pc_out_DBU, IR_DBU, MD_DBU, rf_A_DBU, rf_B_DBU, aluout_DBU, PCSourse,PCwe,IorD,MemWrite,IRWrite,RegDst,MemtoReg,RegWrite,ALUop,ALUSrcA,ALUSrcB,ALUZero);
90
91    reg [15:0]led_reg;
92    assign led=led_reg;
93    initial led_reg=0;
94
95    always@(*)begin
96      case(sel)
97       3'b000: begin//查看CPU运行结果
98          led_reg = {2'b0,m_rf_addr<<2};
99          if(m_rf) begin   //查看存储器
100            {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=m_data;
101         end
102         else begin     //查看寄存器堆
103            {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=rf_data;
104         end
105      end
106      3'b001: begin
107         led_reg={PCSourse,PCwe,IorD,MemWrite,IRWrite,RegDst,MemtoReg,RegWrite,ALUop,ALUSrcA,ALUSrcB,ALUZero};
108         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=pc_out_DBU;
109      end
110      3'b010: begin
111         led_reg={PCSourse,PCwe,IorD,MemWrite,IRWrite,RegDst,MemtoReg,RegWrite,ALUop,ALUSrcA,ALUSrcB,ALUZero};
112         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=IR_DBU;
113      end
114      3'b011: begin
115         led_reg={PCSourse,PCwe,IorD,MemWrite,IRWrite,RegDst,MemtoReg,RegWrite,ALUop,ALUSrcA,ALUSrcB,ALUZero};
116         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=MD_DBU;
117      end
118      3'b100: begin
119         led_reg={PCSourse,PCwe,IorD,MemWrite,IRWrite,RegDst,MemtoReg,RegWrite,ALUop,ALUSrcA,ALUSrcB,ALUZero};
120         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=rf_A_DBU;
121      end
122      3'b101: begin
123         led_reg={PCSourse,PCwe,IorD,MemWrite,IRWrite,RegDst,MemtoReg,RegWrite,ALUop,ALUSrcA,ALUSrcB,ALUZero};
124         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=rf_B_DBU;
125      end
126      3'b110: begin
127         led_reg={PCSourse,PCwe,IorD,MemWrite,IRWrite,RegDst,MemtoReg,RegWrite,ALUop,ALUSrcA,ALUSrcB,ALUZero};
128         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=aluout_DBU;
129      end
130      3'b111: begin
131         led_reg={PCSourse,PCwe,IorD,MemWrite,IRWrite,RegDst,MemtoReg,RegWrite,ALUop,ALUSrcA,ALUSrcB,ALUZero};
132         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=0;
133      end
134      default: ;
135    endcase
136   end
```

这部分逻辑比较简单，直接在对应情况给对应输出即可。

再就是输出模块，根据 led[11:0]的值以及数码管要显示的内容，将这些输出展现在 LED 灯以及数码管上。为使八位数码管显示不同的内容，必须时分复用。引入一个分频变量[18:0]Reg_N，兼以位选功能。每个时钟周期（DBU 外接板载时钟，100MHz）将 Reg_N 的值+1，该变量的最高三位作为位选信号，对应 8 个数码管，且八根数码管占用的显示时长相等。

```
64    ////////////////////////////////////////////////////////////////////////
65     localparam N = 18; //使用低位对100Mhz的时钟进行分频
66        reg [N-1:0] regN; //高位作为控制信号，低位为计数器，对时钟进行分频
67        reg [3:0] hex_in; //段选控制信号
68        reg [3:0] hex0,hex1,hex2,hex3,hex4,hex5,hex6,hex7;
69        initial hex0=0; initial hex1=0;initial hex2=0; initial hex3=0;
70        initial hex4=0; initial hex5=0;initial hex6=0; initial hex7=0;
71        initial regN=0;
72    ////////////////////////////////////////////////////////////////////////
```

最后在下方声明位选以及要显示的内容，数码管部分就完成了。

```
//数码管输出
 always@(*)
 begin
  case(regN[N-1:N-3])
         3'b000:begin
             SSEG_AN = 8'b11111110; //选中第1个数码管
             hex_in = hex0; //数码管显示的数字由hex_in控制，显示hex0输入的数字;
          end
         3'b001:begin
             SSEG_AN = 8'b11111101; //选中第2个数码管
             hex_in = hex1;
          end
         3'b010:begin
           SSEG_AN = 8'b11111011; //选中第3个数码管
           hex_in = hex2;
          end
         3'b011:begin
           SSEG_AN = 8'b11111011; //选中第4个数码管
           hex_in = hex3;
          end
         3'b100:begin
             SSEG_AN = 8'b11101111; //选中第5个数码管
             hex_in = hex4;
          end
         3'b101:begin
             SSEG_AN = 8'b11011111; //选中第6个数码管
             hex_in = hex5;
          end
         3'b110:begin
           SSEG_AN = 8'b10111111; //选中第7个数码管
           hex_in = hex6;
        end
         3'b111:begin
             SSEG_AN = 8'b01111111; //选中第8个数码管
             hex_in = hex7;
          end
        default: SSEG_AN=8'b11111111;
    endcase
end
```

Hex_in 决定了该时刻数码管显示的内容；SSEG_AN 决定哪个数码管发光。

```
//数码管输出部分
always@(*)begin
    case(hex_in)
                    4'h0: SSEG_CA[7:0] = 8'b00000011; //共阳极数码管
                    4'h1: SSEG_CA[7:0] = 8'b10011111;
                    4'h2: SSEG_CA[7:0] = 8'b00100101;
                    4'h3: SSEG_CA[7:0] = 8'b00001101;
                    4'h4: SSEG_CA[7:0] = 8'b10011001;
                    4'h5: SSEG_CA[7:0] = 8'b01001001;
                    4'h6: SSEG_CA[7:0] = 8'b01000001;
                    4'h7: SSEG_CA[7:0] = 8'b00011111;
                    4'h8: SSEG_CA[7:0] = 8'b00000001;
                    4'h9: SSEG_CA[7:0] = 8'b00001001;
                    4'ha: SSEG_CA[7:0] = 8'b00010000;
                    4'hb: SSEG_CA[7:0] = 8'b00000000;
                    4'hc: SSEG_CA[7:0] = 8'b01100010;
                    4'hd: SSEG_CA[7:0] = 8'b00000010;
                    4'he: SSEG_CA[7:0] = 8'b01100000;
                  default: SSEG_CA[7:0] = 8'b01110000;
        endcase
end
```

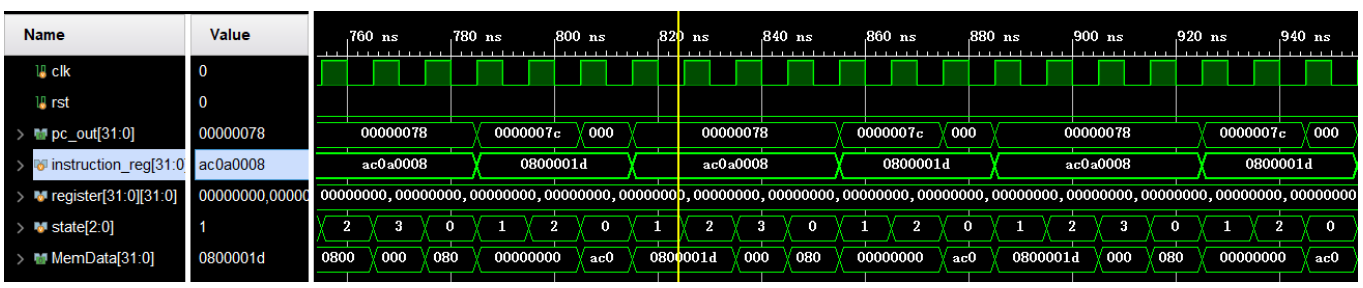
五、实验结果：

对于多周期 CPU（不带 DBU），初始化其 RAM 存储器，利用一个简单的 MIPS 程序对其进行完整性和正确性的验证即可，所用的程序写成汇编代码如下：

```
# 本文档存储器以字编址
# 初始PC = 0x00000000

      j _start     #                                                    0

.data
    .word 0,6,0,8,0x80000000,0x80000100,0x100,5,0,0,0
                          #编译成机器码时，编译器会在前面多加个0，所以后面lw指令地址会多加4

_start:
       addi $t0,$0,3          #t0=3                                      44
       addi $t1,$0,5          #t1=5                                      48
       addi $t2,$0,1          #t2=1                                      52
       addi $t3,$0,0                #t3=0                                56

       add  $s0,$t1,$t0             #s0=t1+t0=8   测试add指令            60
       lw   $s1,12($0)              #                                    64
       beq  $s1,$s0,_next1          #正确跳到_next                       68

       j _fail

_next1:
   lw $t0, 16($0)             #t0 = 0x80000000                           76
       lw $t1, 20($0)         #t1 = 0x80000100                           80
       add  $s0,$t1,$t0       #s0 = 0x00000100 = 256                     84
       lw $s1, 24($0)               #                                    88
       beq  $s1,$s0,_next2          #正确跳到_success                    92

       j _fail

_next2:
       add $0, $0, $t2              #$0应该一直为0                      100
       beq $0,$t3, success          #                                   104

_fail:
       sw   $t3,8($0)
                 #失败通过看存储器地址0x08里值，若为0则测试不通过，最初地址0x08里值为0  108
       j    _fail

_success:
       sw   $t2,8($0)    #全部测试通过，存储器地址0x08里值为1                116
       j   _success

                          #判断测试通过的条件是最后存储器地址 0x08 里值为 1，说明全部通过测试
```

由于只有 CPU 模块，最后存储器里的值不得而知，需要加了 DBU 以后才能看出。不过可以通过仿真得知 CPU 的功能是否完整；查看最后的指令循环是在 fail 内还是在 success 内以及寄存器$t2（10 号寄存器）内的值是否为 1，来确定该部分是否成功。

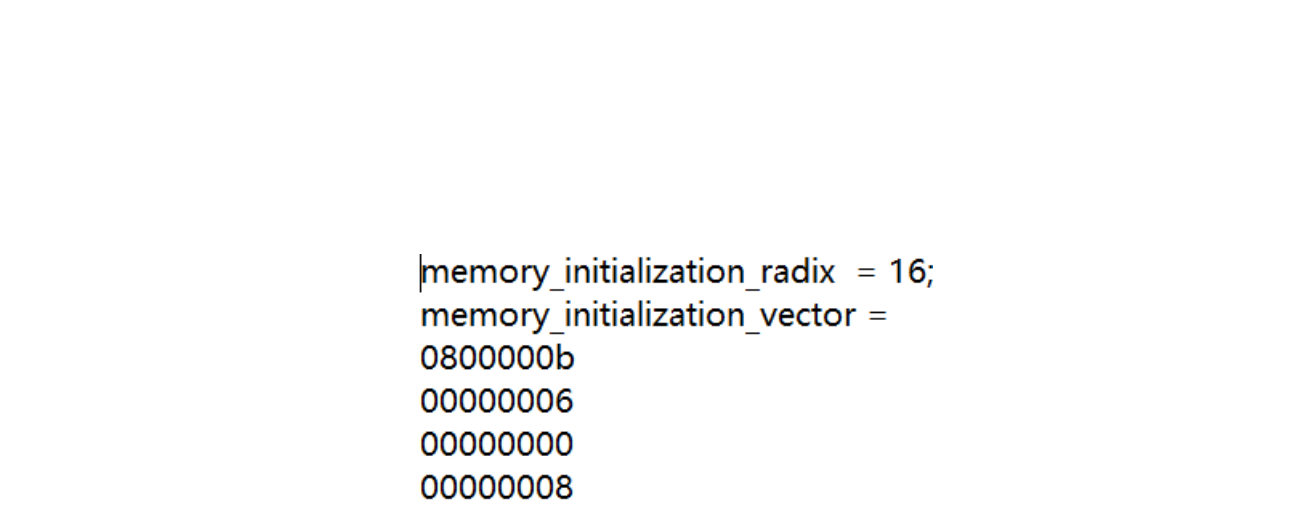
仿真结果如下： 仿真文件只是提供了 clk 信号。



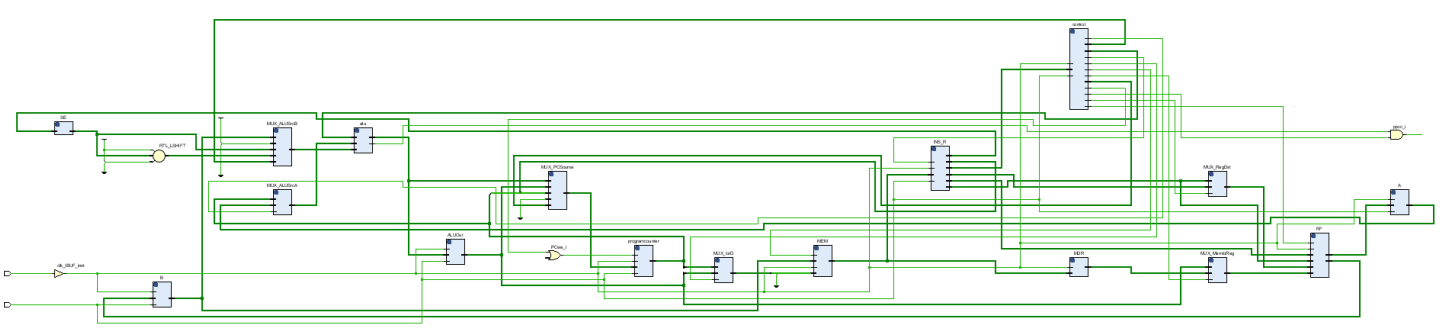
可知最后指令一直在 ac0a0008，0800001d 两条指令之间跳转。对比指令存储器的 coe 文件（或是将他们写成 32 位指令的形式分析），可知最后 CPU 在 success 内循环。

```
memory_initialization_radix  = 16;
memory_initialization_vector =
0800000b
00000006
00000000
00000008
80000000
80000100
00000100
00000005
00000000
00000000
00000000
20080003
20090005
200a0001
200b0000
01288020
8c11000c
12300001
0800001b
8c080010
8c090014
01288020
8c110018
12300001
0800001b
000a0020
100b0002
ac0b0008
0800001b
ac0a0008
0800001d
```

再查看最后 10 号寄存器内的值是否为 1：

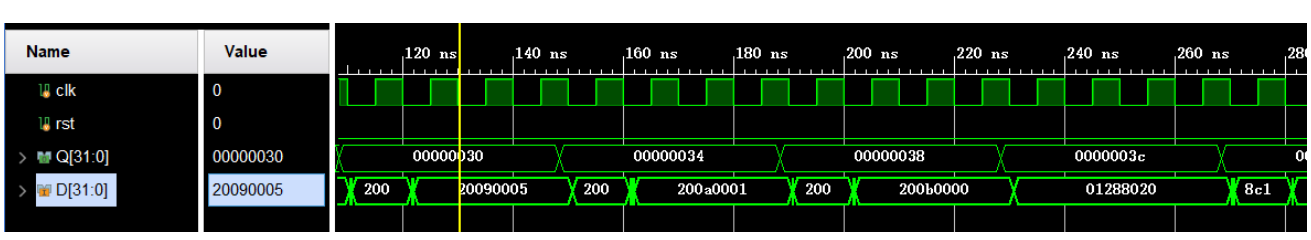


再对其进行综合，进行 RTL 分析和时序仿真：



不难看出，RTL 级电路和设计的逻辑图是等价的。

再运行综合后时序仿真：



由于接口信号名称变动比较大，仅找到了 pc 值和指令值。

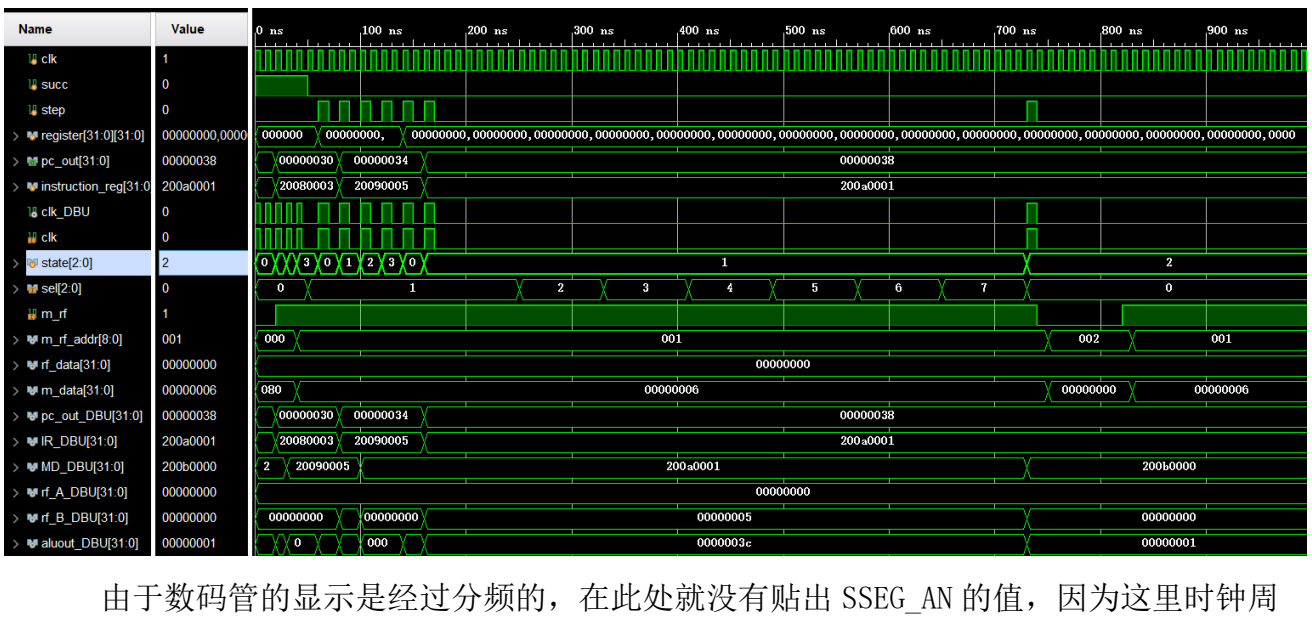
综上，CPU 的功能完整，实现了六条指令的执行。

再是对测试单元 DBU 的调试：

以下为仿真代码：

```
`timescale 1ns / 1ps
module DBU_sim( );
    reg clk,rst;
    reg succ,step,m_rf,inc,dec;
    reg [2:0] sel;
    wire [15:0] led;
    wire [7:0] SSEG_AN,SSEG_CA;
    parameter PERIOD = 10,
              CYCLE = 200;
    DBU TEST (.clk(clk), .rst(rst), .succ(succ), .step(step), .m_rf(m_rf), .inc(inc), .dec(dec), .sel(sel), .led(led), .SSEG_AN(SSEG_AN), .SSEG_CA(SSEG_CA));

    initial
    begin
        clk = 1;
        repeat (2 * CYCLE)
            #(PERIOD/2) clk = ~clk;
        $finish;
    end

    initial
    begin
        rst = 1;succ = 1;step = 0;sel = 3'b000;inc = 0;dec = 0;m_rf = 0;    //连续运行，查看寄存器堆地址0的内容
        #(PERIOD*2)
        rst = 0;m_rf = 1;       //查看存储器地址0的内容
        #(PERIOD)
        inc = 1;                //查看存储器地址1的内容
        #(PERIOD)
        inc = 0;
        #(PERIOD)
        succ = 0;sel = 3'b001;      //改为按步运行，并选择查看pc1的数据
         #(PERIOD)
          step = 1;
         #(PERIOD)
          step = 0;
         #(PERIOD)
          step = 1;
         #(PERIOD)

          #(PERIOD)
          step = 1;
         #(PERIOD)
          step = 0;
          #(PERIOD)
          step = 1;
         #(PERIOD)
          step = 0;
          #(PERIOD)
          step = 1;
         #(PERIOD)
          step = 0;
          #(PERIOD)
          step = 1;
         #(PERIOD)
          step = 0;
        #(PERIOD*8)
        sel = 3'b010;           //查看pc的数据
        #(PERIOD*8)
        sel = 3'b011;           //查看instr的数据
        #(PERIOD*8)
        sel = 3'b100;           //查看alu_a的数据
        #(PERIOD*8)
        sel = 3'b101;           //查看writedata的数据
        #(PERIOD*8)
        sel = 3'b110;           //查看alu_result的数据
        #(PERIOD*8)
        sel = 3'b111;           //查看read_data的数据
        #(PERIOD*8)
        sel = 3'b000;step = 1;  //回到查看运行结果
        #(PERIOD)
        inc = 1;m_rf = 0;step = 0;      //查看寄存器堆地址2的内容
        #(PERIOD*8)
        inc = 0;dec = 1;m_rf = 1;        //查看寄存器堆地址2的内容
        #(PERIOD)
        inc = 0;dec = 0;
    end
endmodule
```

做了两条指令的仿真，并停留在第三条指令的取值和译码两段。波形如下所示：



由于数码管的显示是经过分频的，在此处就没有贴出 SSEG_AN 的值，因为这里时钟周期太少，19 位的分频信号不会变化。但是可以通过查看 hex0-7 的值以及 led 的值，就可以确定 CPU 是否功能完整，正确。

对各个信号进行查看。前两条指令为：

addi $t0,$0,3

addi $t1,$0,5

对应的二进制 32 位代码分别为（十六进制表示）：0x20080003，0x20090005

根据功能叙述，对应仿真结果，显然仿真的结果符合预期，pc，led，hex 的值都是所期望得到的。

六、心得体会：

本次实验需要设计一个多周期的 CPU 及相应的调试单元 DBU。

多周期 CPU 的重点在于数据通路的设计以及控制器的实现。当数据通路设计完成后，相应的写出每个模块的 verilog 实现代码即可，而控制器则需要画出状态转移图，并明确每个状态的输入输出。调试单元 DBU 的重点在于需要将设计好的 CPU 模块连接出所有需要的输出信号，并在 DBU 模块中实现组合逻辑的设计，并确定数码管和 led 灯的输出。

多周期 CPU 是对单周期 CPU 的一个改进，能够大大提升 CPU 的利用率。

七、思考题

新增一条指令：

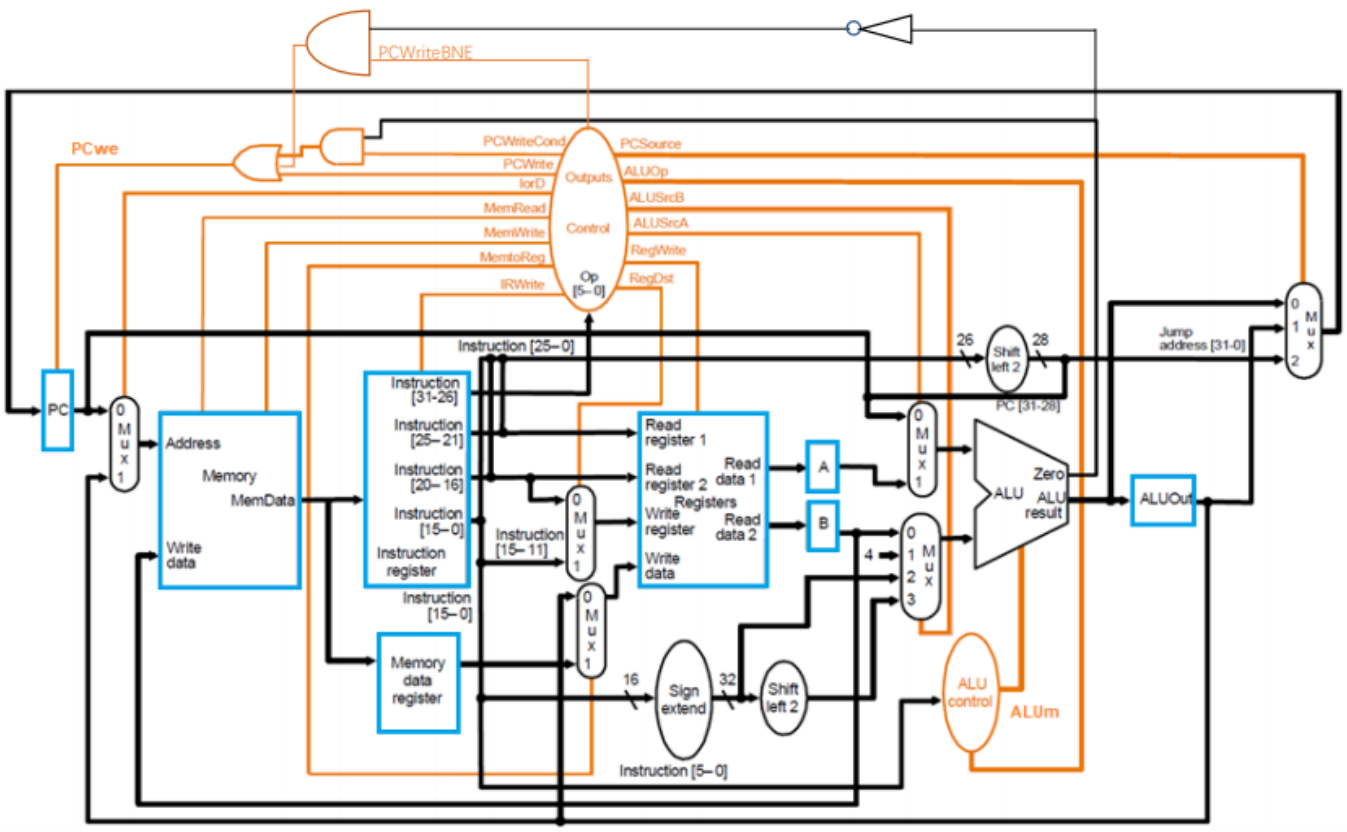
bne: if (rs != rt) then pc <- pc + 4 + addr << 2, else pc <- pc + 4;

op = 000101;

op(6 bits)	rs(5 bits)	rt(5 bits)	addr/immediate(16 bits)

这条指令的功能是：BEQ 指令的反面，若两个目标寄存器中的值不相等，则跳转到指定地址。

需要修改数据通路，修改后的数据通路如下图所示：



从图中可以看出，新增了一个与门，当两个寄存器内的内容不一致时，这一支就会开启或门，而使 PC 可写，达到指令目标。由于只增加了一条指令，数据通路变动不大。

而对应的，控制器也需要加入一个信号 PCWriteBNE，以及之前已经存在的信号对新指令的赋值。仅有执行新指令时，加入的新控制信号为 1；而对于新指令来说，本来存在的那些控制信号值与执行 BEQ 指令时相同。