计算机组成原理 实验报告

姓名：龚小航　学号：PB18151866　实验日期：2020-5-27

一、实验题目：

Lab05 流水线 CPU

二、实验目的

- 理解流水线 CPU 的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：

设计流水线 CPU,能够执行六条指令：add，addi，lw，sw，beq，j. 结构化描述流水线 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. 设计流水线 CPU（不包含调试单元）：

先列出该部分的输入输出关系，流水线 CPU 逻辑图如下所示：

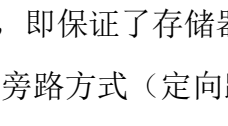
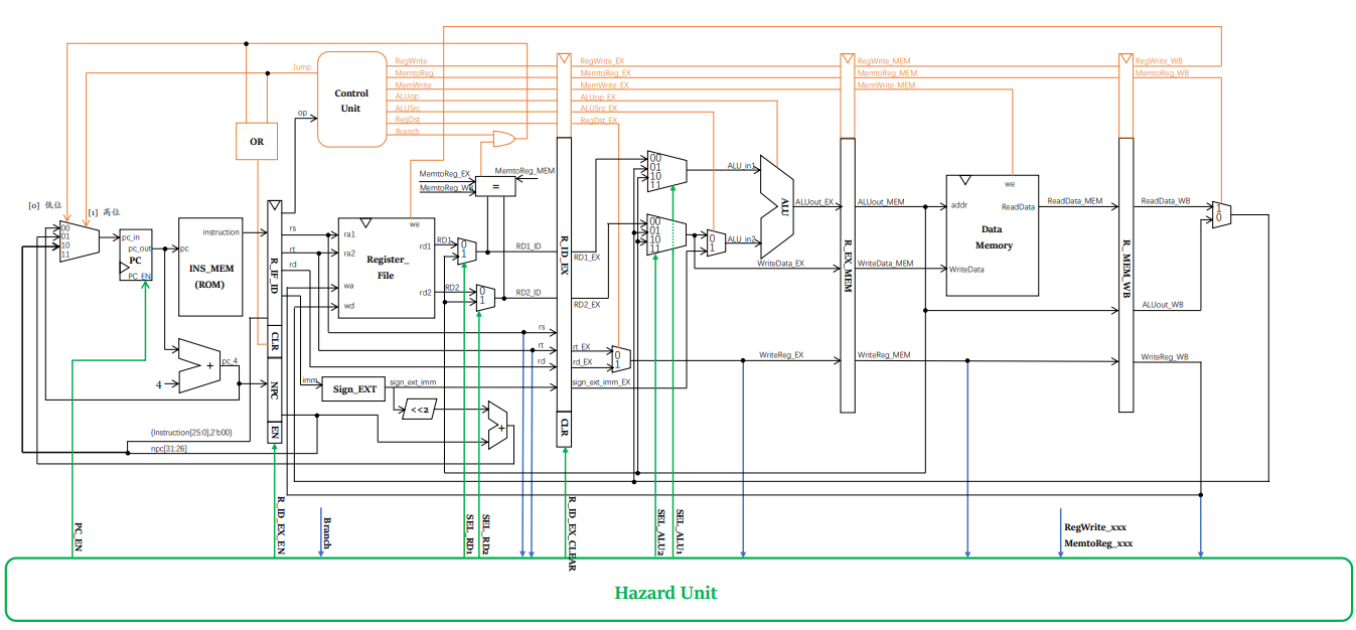


图 1 流水线 CPU 逻辑图

引脚说明：clk 为时钟信号，一个时钟脉冲流水级转化一次，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：(源文件见附件)



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，指令和数据均通过例化深度为 512，位宽 32bit 的分布式存储器实现，指令存储器为单端口 ROM，数据存储器为单端口 RAM。

以下先简要说明流水线 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分别存入指令存储器和数据存储器；同时将 pc 置 0(由具体的程序入口确定)。
- 初始化完成，时钟接入，CPU 开始运行： 流水线 CPU 的运行与单周期和多周期不同。对于流水线 CPU，决定 CPU 状态以及将要做的事的关键部分是四个段间寄存器内的内容，一个段间寄存器存储了一条指令在当前流水段及以后流水段需要的指令和数据。和多周期类似的是，五级流水线 CPU 也将一条指令分为多个周期完成，但除了分支成功和跳转指令，每一条指令都需要五个周期执行。
- 从时刻 0 开始，指令逐渐填充流水线，在同一时刻最多有五条指令在同时执行，即每个流水级都在工作，但是对 CPU 数据（寄存器堆和存储器）的更改仅在 WB 段实现。
- 由于同一时间会执行多条指令，就必然会涉及相关性问题。一般来说，流水线型的 CPU 存在三种相关：结构相关，数据相关，控制相关。而采用上述的五级流水线实现 CPU 从设计上避免了结构相关，即保证了存储器不会在同一时刻读写冲突；为了解决数据相关，本次实验通过采用旁路方式（定向路径）来解决 RAW 相关，前推数据来自 MEM 段或是 WB 段，当遇到取数（lw）后使用型相关时必须将流水线阻塞一个周期。为了解决控制相关，IF/ID 寄存器以及 ID/EX 寄存器都引入了 CLR 段，当输入高电平时就将段内所有内容清零，即让这条指令消失。指令消失以后等同于插入了一条空指令。特别的，由于 beq 指令提前到了 ID 段执行，lw+beq 型指令需要阻塞两个周期。

• Control Unit 是组合逻辑，产生信号和单周期 CPU 完全相同。

译码段产生的控制信号说明：

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

Jump：转移指令标志，Jump==1 时执行指令跳转操作；

Branch：条件分支指令信号 Branch==1 时代表 beq 指令，只要比较成功就可以跳转；

MemRead：数据寄存器读使能信号，在本例中没有作用（异步读取），实际未使用；

MemtoReg：选择写回寄存器的数据来自 ALU 的运算结果（0）或是数据存储器（1）；

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；

MemWrite：数据存储器写使能信号，为高电平时允许在下个时钟上升沿写入；

ALUSrc：选择 ALU 的第二个操作数，来自寄存器堆(0)或是符号扩展模块(1)；

RegWrite：寄存器堆写使能，高电平时允许在下个时钟上升沿写入。

和单周期 CPU 的区别仅在于这些信号在不同的流水级用到。

• 指令系列快要执行完成时，排空流水线。

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、所有段间寄存器、寄存器堆、数据存储器，其余均以组合逻辑输出，包括控制模块和相关处理模块。

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相关处理模块的实现，这是整个程序的总体框架与重点。

```
module TOP(
    input clk,
    input rst
    );
  /*以下为控制信号声明*/
    wire RegWrite,MemtoReg,MemWrite,ALUSrc,RegDst,Branch,Jump;
    wire RegWrite_EX,MemtoReg_EX,MemWrite_EX,ALUSrc_EX,RegDst_EX;
    wire [2:0] ALUop,ALUop_EX;
    wire RegWrite_MEM,MemtoReg_MEM,MemWrite_MEM;
    wire RegWrite_WB,MemtoReg_WB;
  /*以下声明了相关模块的输出信号*/
    wire PC_EN, R_IF_ID_EN, SEL_RD1, SEL_RD2, R_ID_EX_CLEAR,  R_IF_ID_CLEAR;//两个清除寄存器信号也在此处声明
    wire [1:0] SEL_ALU1, SEL_ALU2;
  /*以下为取指段需要用到的信号以及5位长的信号的声明*/
    wire [1:0] pc_sel;
    wire [31:0] pc_in, pc_out, pc_4, pc_branch, pc_jump, instruction, npc;
    wire [5:0] op;
    wire [4:0] rs,rt,rd, rs_EX, rt_EX, rd_EX, WriteReg_EX, WriteReg_MEM, WriteReg_WB;
    wire [15:0] addr_immediate;
    wire [27:0] jumpaddr_28bit;
  /*以下为译码段产生的信号声明*/
    wire [31:0] RD1,RD2,sign_ext_imm_ID,RD1_ID,RD2_ID;
    wire beq_equal;
  /*以下为EX阶段信号声明*/
    wire [31:0] RD1_EX,RD2_EX,sign_ext_imm_EX;
    wire [31:0] ALU_in1,ALU_in2, ALUout_EX;
    wire [31:0] WriteData_EX;
  /*以下为MEM段信号声明*/
    wire [31:0] ALUout_MEM, WriteData_MEM, ReadData_MEM;
  /*以下为WB段信号声明*/
    wire [31:0] ReadData_WB, ALUout_WB, WB_DATA;


    Instruction_Memory INS_MEM (pc_out, instruction);
    PC pc (clk, rst, pc_in, pc_out, PC_EN);
    R_IF_ID  R_IF_ID(clk, rst, R_IF_ID_CLEAR, R_IF_ID_EN, instruction, pc_4, op, rs, rt, rd, addr_immediate, jumpaddr_28bit, npc);
    Register_File #(32) RF (.clk(clk), .ra1(rs), .ra2(rt), .wa(WriteReg_WB), .RegWrite(RegWrite_WB), .wd(WB_DATA), .rd1(RD1), .rd2(RD2));
    Sign_Extend Sign_Extend (addr_immediate, sign_ext_imm_ID);
    R_ID_EX R_ID_EX (clk, rst, R_ID_EX_CLEAR, RegDst, MemtoReg, ALUop, MemWrite, ALUSrc, RegWrite, RegDst_EX, MemtoReg_EX, ALUop_EX,
          MemWrite_EX, ALUSrc_EX, RegWrite_EX, RD1_ID, RD2_ID, RD1_EX, RD2_EX, rs, rt, rd, rs_EX, rt_EX, rd_EX, sign_ext_imm_ID,sign_ext_imm_EX);
    ALU #(32) ALU(ALU_in1, ALU_in2, ALUop_EX, ALUout_EX);
    R_EX_MEM R_EX_MEM (clk, rst, RegWrite_EX, MemtoReg_EX, MemWrite_EX, ALUout_EX, WriteData_EX, WriteReg_EX, RegWrite_MEM, MemtoReg_MEM,
          MemWrite_MEM, ALUout_MEM, WriteData_MEM, WriteReg_MEM);
    Data_Memory DATA_MEM (clk, ALUout_MEM, WriteData_MEM, MemWrite_MEM, ReadData_MEM);
    R_MEM_WB  R_MEM_WB (clk, rst, RegWrite_MEM, MemtoReg_MEM, ReadData_MEM, ALUout_MEM, WriteReg_MEM, RegWrite_WB, MemtoReg_WB, ReadData_WB, ALUout_WB, WriteReg_WB);
    Control_Unit Control_Unit (op, RegDst, Jump, Branch, MemtoReg, ALUop, MemWrite, ALUSrc, RegWrite);
    Hazard_unit Hazard (Branch, MemtoReg_EX, RegWrite_EX, MemtoReg_MEM, RegWrite_MEM, RegWrite_WB, rs, rt, rs_EX, rt_EX, WriteReg_EX, WriteReg_MEM,
          WriteReg_WB, PC_EN, R_IF_ID_EN, SEL_RD1, SEL_RD2, R_ID_EX_CLEAR, SEL_ALU1, SEL_ALU2);
    MUX_4 #(32) MUX_PC (pc_4, {sign_ext_imm_ID[29:0],2'b00}+npc, {npc[31:28],jumpaddr_28bit}, 32'h0, pc_sel, pc_in);
    MUX_2 #(32) MUX_RD1 (RD1, ALUout_MEM, SEL_RD1, RD1_ID);
    MUX_2 #(32) MUX_RD2 (RD2, ALUout_MEM, SEL_RD2, RD2_ID);
    MUX_2 #(32) MUX_RegDst_EX (rt_EX, rd_EX, RegDst_EX, WriteReg_EX);
    MUX_4 #(32) MUX_ALU1 (RD1_EX, WB_DATA, ALUout_MEM, 32'h0, SEL_ALU1, ALU_in1);
    MUX_4 #(32) MUX_ALU2 (RD2_EX, WB_DATA, ALUout_MEM, 32'h0, SEL_ALU2, WriteData_EX);
    MUX_2 #(32) MUX_ALUSrc_EX (WriteData_EX, sign_ext_imm_EX, ALUSrc_EX, ALU_in2);
    MUX_2 #(32) MUX_MemtoReg_WB (ALUout_WB, ReadData_WB, MemtoReg_WB, WB_DATA);


    reg beq_equal_reg;
     initial beq_equal_reg=0;
    always@(*)begin
      if((MemtoReg_MEM==1)&&(MemtoReg_EX==1)) beq_equal_reg = 0;
      else if((MemtoReg_MEM==1)&&(MemtoReg_WB==1)) beq_equal_reg = 0;//(ReadData_MEM==WB_DATA)? 1:0;
      else begin
        if(RD1_ID==RD2_ID) beq_equal_reg=1;
        else beq_equal_reg=0;
      end
    end

    assign beq_equal = beq_equal_reg;
    assign pc_4 = pc_out+32'h4;
    assign pc_sel = {Jump,Branch&beq_equal};
    assign R_IF_ID_CLEAR = Jump|(Branch&beq_equal);

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。

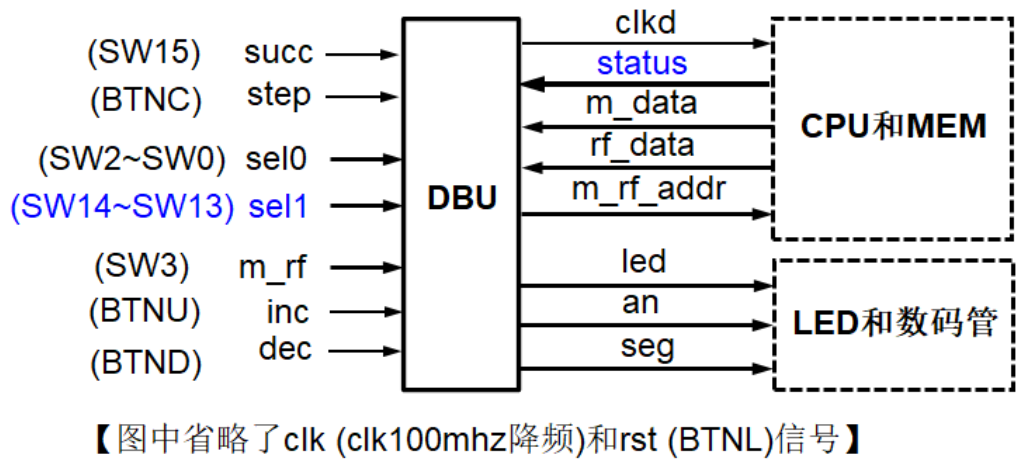
以下为相关处理模块，该模块通过监测五级流水线中的特征值，判断当前是否发生相关，若发生则输出相应的相关处理信号。

```
module Hazard_unit(         //相关处理模块，采用纯组合逻辑输出控制信号
    input Branch,
    input MemtoReg_EX,
    input RegWrite_EX,
    input MemtoReg_MEM,
    input RegWrite_MEM,
    input RegWrite_WB,
    input [4:0] rs,rt,rs_EX,rt_EX,WriteReg_EX,WriteReg_MEM,WriteReg_WB,

    output PC_EN,           //pc使能控制，若为0则阻塞pc的变化
    output R_IF_ID_EN,      //IF_ID寄存器使能信号
    output SEL_RD1,SEL_RD2,
    output R_ID_EX_CLEAR,   //ID_EX寄存器清除信号
    output [1:0] SEL_ALU1,SEL_ALU2
    );

    wire lw_stall,branch_stall;

    assign SEL_RD1 = (rs!=0) & (rs==WriteReg_MEM) & RegWrite_MEM;
    assign SEL_RD2 = (rt!=0) & (rt==WriteReg_MEM) & RegWrite_MEM;

    reg [1:0] SEL_ALU1_reg,SEL_ALU2_reg;
  always @(*) begin
        SEL_ALU1_reg = 2'b00;
        SEL_ALU2_reg = 2'b00;
     if((rs_EX != 0) && (rs_EX == WriteReg_MEM) && RegWrite_MEM)
          SEL_ALU1_reg = 2'b10;
     else if((rs_EX != 0) && (rs_EX == WriteReg_WB) && RegWrite_WB)
          SEL_ALU1_reg = 2'b01;
     if((rt_EX != 0) && (rt_EX == WriteReg_MEM) && RegWrite_MEM)
          SEL_ALU2_reg = 2'b10;
     else if((rt_EX != 0) && (rt_EX == WriteReg_WB) && RegWrite_WB)
          SEL_ALU2_reg =2'b01;
  end

  assign SEL_ALU1=SEL_ALU1_reg;
  assign SEL_ALU2=SEL_ALU2_reg;

assign lw_stall = ((rs == rt_EX) | (rt == rt_EX)) & MemtoReg_EX;
assign branch_stall = (Branch & RegWrite_EX & ((rs == WriteReg_EX) |(rt == WriteReg_EX))) | (Branch & MemtoReg_MEM & ((rs == WriteReg_MEM) |(rt == WriteReg_MEM)));
assign PC_EN = ~(lw_stall | branch_stall);
assign R_IF_ID_EN = ~(lw_stall | branch_stall);
assign R_ID_EX_CLEAR = lw_stall | branch_stall;


endmodule
```

以上就是整个相关处理单元的内容，只要严格按照设计需求，把它转化成相应的 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。
- sel0：3 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

sel0 = 1 ~ 7：查看 CPU 运行状态（status）

16 个 LED 指示灯(SW15~SW0)依次显示段间寄存器的控制信号。

七段数码管显示的数据如下所示：

sel0= 1：PC，程序计数器

sel0= 2：IR/ID 寄存器，其中 sel1=0, 显示 NPC; sel1=1,显示 IR（指令内容）

sel0= 3：ID//EX 寄存器

sel0= 4：EX/MEM 寄存器

sel0 = 5： MEM/WB 寄存器

sel0 = 6：无操作

sel0 = 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(    //已经修改流水线DBU,TOP已匹配
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] sel0,
29    input sel1,
30    input m_rf,
31    input inc,
32    input dec,
33
34    output[15:0] led,
35    output reg [7:0] SSEG_CA,//数码管输出部分
36    output reg  [7:0] SSEG_AN
37      );
```

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：

```
    wire clk_DBU; reg clkdbu_reg;

    always@(posedge clk)begin   //描述succ
      if(succ) clkdbu_reg<=clk;
      else clkdbu_reg<=stepEdge;
    end
     assign clk_DBU=clkdbu_reg;

    reg [7:0] m_rf_addr_reg; wire [7:0] m_rf_addr;
    initial m_rf_addr_reg=0;
    always@(posedge clk)begin   //描述 m_rf_addr
      if(rst) m_rf_addr_reg<=0;
      else begin
         if(incEdge) m_rf_addr_reg<=m_rf_addr_reg+1;
         else if(decEdge) m_rf_addr_reg<=m_rf_addr_reg-1;
         else m_rf_addr_reg<=m_rf_addr_reg;
      end
    end
     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：

```
    wire [31:0] PC_OUT_DBU;
    wire [31:0] IR_ID_NPC_DBU;
    wire [31:0] IR_ID_INS_DBU;
    wire [31:0] ID_EX_DBU;
    wire [31:0] EX_MEM_DBU;
    wire [31:0] MEM_WB_DBU;

    wire RegWrite,MemtoReg,MemWrite,ALUSrc,RegDst,Branch,Jump;
    wire RegWrite_EX,MemtoReg_EX,MemWrite_EX,ALUSrc_EX,RegDst_EX;
    wire [2:0] ALUop,ALUop_EX;
    wire RegWrite_MEM,MemtoReg_MEM,MemWrite_MEM;
    wire RegWrite_WB,MemtoReg_WB;

    TOP test (clk_DBU, rst, m_rf_addr, m_data, rf_data, PC_OUT_DBU, IR_ID_NPC_DBU, IR_ID_INS_DBU, ID_EX_DBU, EX_MEM_DBU,
        MEM_WB_DBU, RegWrite,MemtoReg,MemWrite,ALUSrc,RegDst,Branch,Jump, RegWrite_EX,MemtoReg_EX,MemWrite_EX,ALUSrc_EX,RegDst_EX,
        ALUop,ALUop_EX, RegWrite_MEM,MemtoReg_MEM,MemWrite_MEM, RegWrite_WB,MemtoReg_WB );

    reg [15:0]led_reg;
    assign led=led_reg;
    initial led_reg=0;

    always@(*)begin
      case(sel0)
       3'b000: begin//查看CPU运行结果
          led_reg = {2'b0,m_rf_addr<<2};
          if(m_rf) begin   //查看存储器
              {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=m_data;
          end
          else begin     //查看寄存器堆
              {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=rf_data;
          end
        end
       3'b001: begin
          led_reg=0;
          {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=PC_OUT_DBU;
        end
       3'b010: begin
          led_reg={RegWrite,MemtoReg,MemWrite,ALUop,ALUSrc,RegDst,Branch,Jump,6'b0};
         if(sel1==0)
           {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=IR_ID_NPC_DBU;
         else
           {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=IR_ID_INS_DBU;
       end
      3'b011: begin
         led_reg={RegWrite_EX,MemtoReg_EX,ALUop_EX,MemWrite_EX,ALUSrc_EX,RegDst_EX, 8'b0};
         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=ID_EX_DBU;
       end
      3'b100: begin
         led_reg={RegWrite_MEM,MemtoReg_MEM,MemWrite_MEM, 13'b0};
         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=EX_MEM_DBU;
       end
      3'b101: begin
         led_reg={RegWrite_WB,MemtoReg_WB,14'b0};
         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=MEM_WB_DBU;
       end
      3'b110: begin
         led_reg=0;
         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=0;
       end
      3'b111: begin
         led_reg=0;
         {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=0;
       end
      default: ;
     endcase
    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[15:0]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。

```
    //////////////////////////////////////////////////////////////////////////////////
     localparam N = 18; //使用低位对100Mhz的时钟进行分频
        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
        reg [3:0] hex_in; //段选控制信号
        reg [3:0] hex0,hex1,hex2,hex3,hex4,hex5,hex6,hex7;
        initial hex0=0; initial hex1=0;initial hex2=0; initial hex3=0;
        initial hex4=0; initial hex5=0;initial hex6=0; initial hex7=0;
        initial regN=0;
    //////////////////////////////////////////////////////////////////////////////////
```

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。

```
142    //数码管输出
143    always@(*)
144    begin
145     case(regN[N-1:N-3])
146           3'b000:begin
147               SSEG_AN = 8'b11111110; //选中第1个数码管
148               hex_in = hex0; //数码管显示的数字由hex_in控制，显示hex0输入的数字;
149            end
150           3'b001:begin
151               SSEG_AN = 8'b11111101; //选中第2个数码管
152               hex_in = hex1;
153            end
154           3'b010:begin
155             SSEG_AN = 8'b11111011; //选中第3个数码管
156             hex_in = hex2;
157            end
158           3'b011:begin
159             SSEG_AN = 8'b11111011; //选中第4个数码管
160             hex_in = hex3;
161            end
162           3'b100:begin
163               SSEG_AN = 8'b11101111; //选中第5个数码管
164               hex_in = hex4;
165            end
166           3'b101:begin
167               SSEG_AN = 8'b11011111; //选中第6个数码管
168               hex_in = hex5;
169            end
170           3'b110:begin
171             SSEG_AN = 8'b10111111; //选中第7个数码管
172             hex_in = hex6;
173          end
174         3'b111:begin
175             SSEG_AN = 8'b01111111; //选中第8个数码管
176             hex_in = hex7;
177          end
178         default: SSEG_AN=8'b11111111;
179      endcase
180   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。

```
182    //数码管输出部分
183    always@(*)begin
184        case(hex_in)
185                        4'h0: SSEG_CA[7:0] = 8'b00000011; //共阳极数码管
186                        4'h1: SSEG_CA[7:0] = 8'b10011111;
187                        4'h2: SSEG_CA[7:0] = 8'b00100101;
188                        4'h3: SSEG_CA[7:0] = 8'b00001101;
189                        4'h4: SSEG_CA[7:0] = 8'b10011001;
190                        4'h5: SSEG_CA[7:0] = 8'b01001001;
191                        4'h6: SSEG_CA[7:0] = 8'b01000001;
192                        4'h7: SSEG_CA[7:0] = 8'b00011111;
193                        4'h8: SSEG_CA[7:0] = 8'b00000001;
194                        4'h9: SSEG_CA[7:0] = 8'b00001001;
195                        4'ha: SSEG_CA[7:0] = 8'b00010000;
196                        4'hb: SSEG_CA[7:0] = 8'b00000000;
197                        4'hc: SSEG_CA[7:0] = 8'b01100010;
198                        4'hd: SSEG_CA[7:0] = 8'b00000010;
199                        4'he: SSEG_CA[7:0] = 8'b01100000;
200                    default: SSEG_CA[7:0] = 8'b01110000;
201          endcase
202    end
```

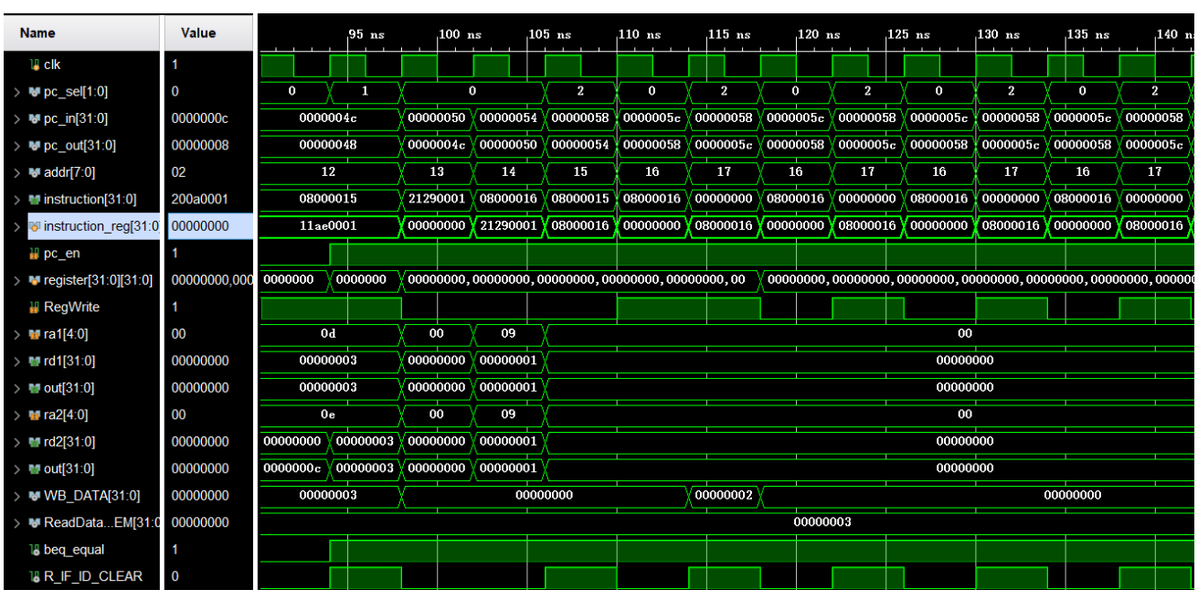
五、实验结果：

对于流水线 CPU（不带 DBU），初始化其指令存储器 ROM 以及数据存储器 RAM，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：

```
# Test cases for MIPS 5-Stage pipeline

.data
    .word 0,1,2,3,0x80000000,0x80000100,0x100,5,0

_start:
    add $t1, $0, $0      # $t1 = 0
    j _test0

_test0:
    addi $t2, $0, 1          # $t2 = 1
    addi $t2, $t2, 1         # $t2 = 2
    add $t2, $t2, $t2        # $t2 = 4
    addi $t2, $t2, -4        # $t2 = 0
    beq $t2, $0, _next0      # if $t2 == $0: $t1++, go next testcase, else: go fail
    j _fail
_next0:
    addi $t1, $t1, 1         # $t1++
    j _test1

_test1:
    addi $0, $0, 4           # $0 += 4
    lw $t2, 4($0)            # $t2 = MEM[1]
    lw $t3, 8($0)            # $t3 = MEM[2]
    add $t4, $t2, $t3
    sw $t4, 0($0)            # MEM[0] = $t4
    lw $t5, 0($0)            # $t5 = MEM[0]
    lw $t6, 12($0)           # $t6 = MEM[3]
    beq $t5, $t6, _next1
    j _fail

_next1:
    addi $t1, $t1, 1
    j _success

_fail:
    j _fail

_success:
    j _success               # if success: $t1 == 2
```

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**$t1**（9 号寄存器）内的值是否为 2，来确定该部分是否成功。

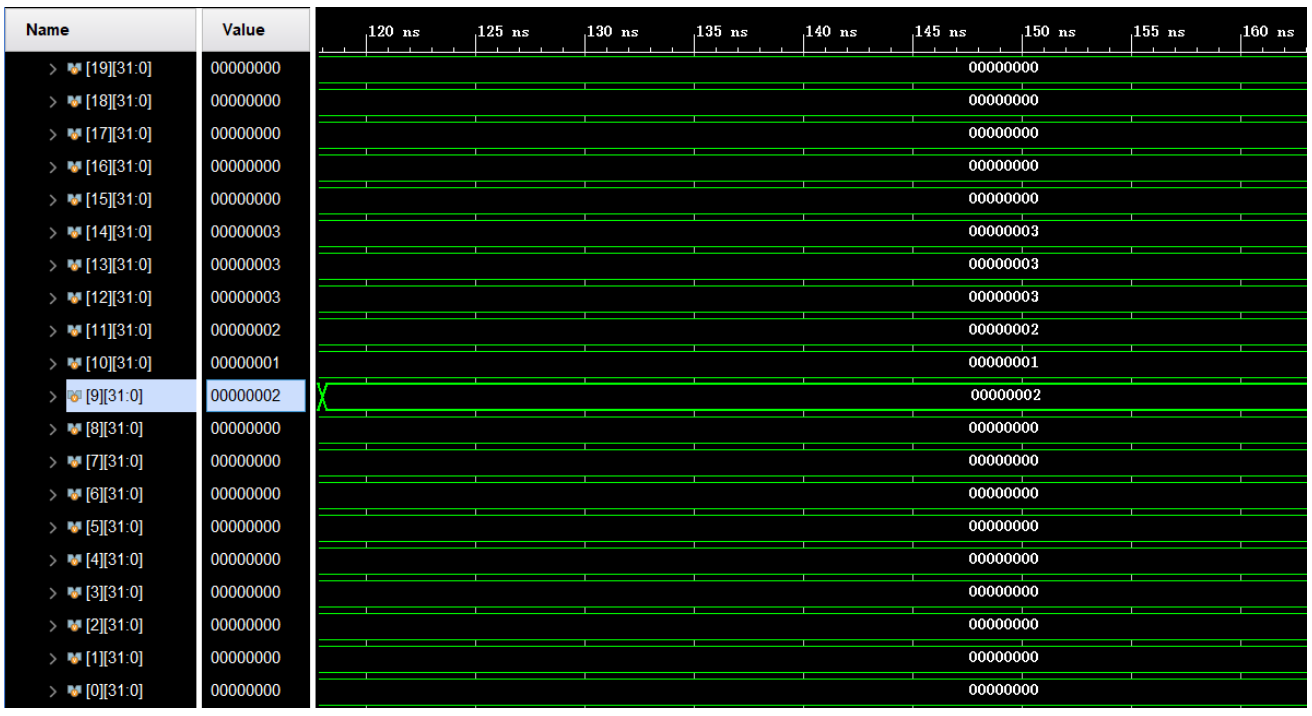
仿真结果如下： 仿真文件只是提供了 clk 信号。



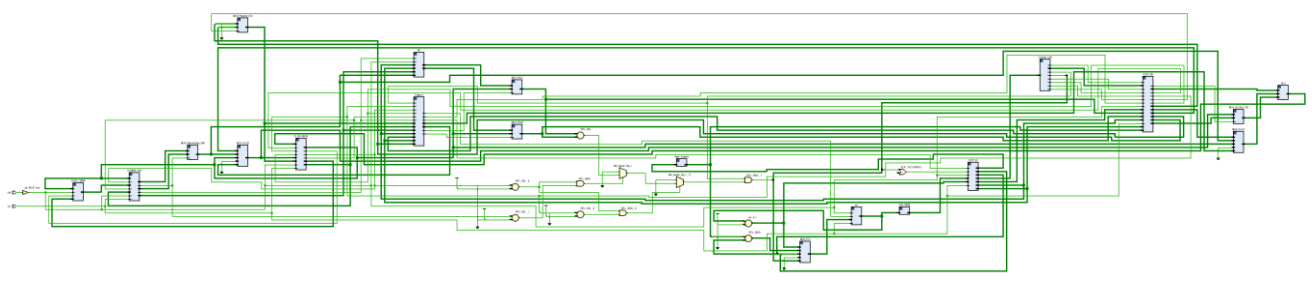
图中 instrction_reg 信号来源于 IF/ID 寄存器，是真正进入流水线处理的指令信号。可知最后指令一直在 08000016 指令上反复跳转。对比指令存储器的.coe 文件（或是将他们写成 32 位指令的形式分析），可知最后 CPU 在 success 内循环。

```
memory_initialization_radix  = 16;
memory_initialization_vector =
00004820
08000002
200a0001
214a0001
014a5020
214afffc
11400001
08000015
21290001
0800000a
20000004
8c0a0004
8c0b0008
014b6020
ac0c0000
8c0d0000
8c0e000c
11ae0001
08000015
21290001
08000016
08000015
08000016
```

再查看最后 9 号寄存器内的值是否为 2：

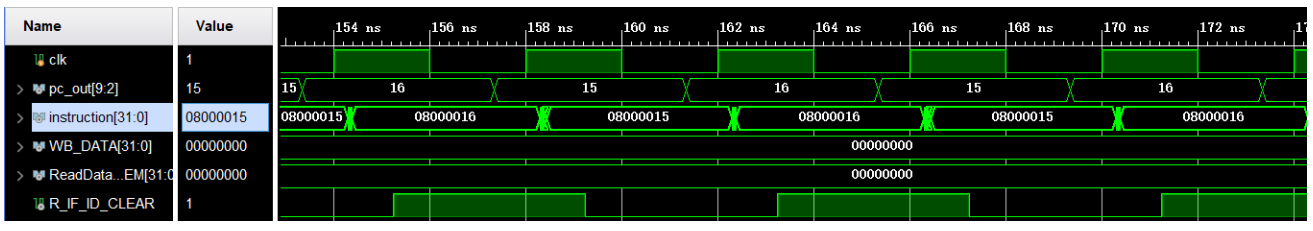


再对其进行综合，进行 RTL 分析：



将各个模块重新排列，可以看出 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，仅找到了指令存储器 ROM 输出的指令值 instruction 和一些其他的信号。可以从图中看出，和上图行为仿真得到的结果是一样的。

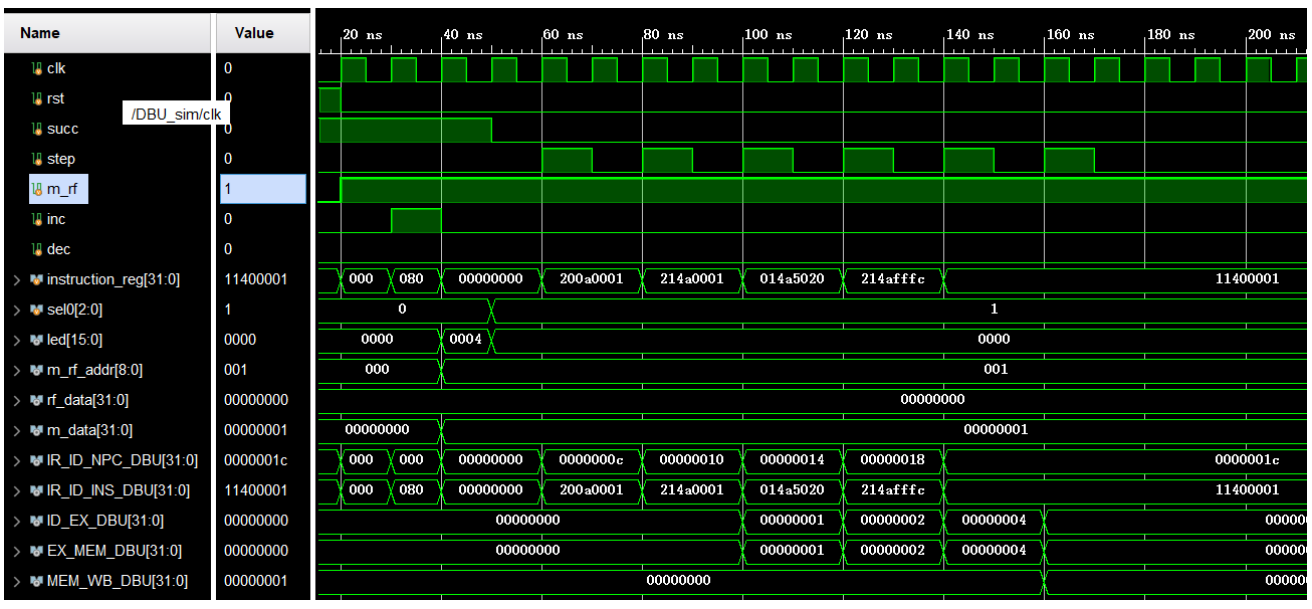
综上，CPU 的功能完整，实现了六条指令的执行。

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] sel0;
    reg sel1;
    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), .sel0(sel0), .sel1(sel1), .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;sel0 = 3'b000; sel1=0; 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;sel0 = 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)
         step = 0;
         #(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)
        sel0 = 3'b010;          //查看pc的数据
        #(PERIOD*8)
        sel0 = 3'b011;          //查看instr的数据
        #(PERIOD*8)
        sel0 = 3'b100;          //查看alu_a的数据
        #(PERIOD*8)
        sel0 = 3'b101;          //查看writedata的数据
        #(PERIOD*8)
        sel0 = 3'b110;          //查看alu_result的数据
        #(PERIOD*8)
        sel0 = 3'b111;          //查看read_data的数据
        #(PERIOD*8)
        sel0 = 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
```

做了六条指令的仿真，并停留在 beq 指令的取指段。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（也可以是对应信号的值），就可以确定 CPU 是否功能完整，正确。

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流水线 CPU 及相应的调试单元 DBU。

流水线 CPU 的重点在于数据通路的设计以及相关处理单元的实现。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。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。调试模块和单周期多周期 CPU 的没有本质上的区别，不需要重新设计可以直接复用。

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

七、思考题

分支预测指在执行 beq 指令时预测分支发生或是不发生。本次实验中采用了最简单的分支预测方法，即将 beq 指令提前到 ID 段执行，始终假定分支不发生。即 beq 指令在完成 IF 之后进行译码，此时始终从指令存储器中取出 beq 指令的下一条指令：若分支不成功则继续运行流水线即可，相比于没有做预测省出一个时钟周期；若分支成功则清除 IF/ID 段间寄存器以及 ID/EX 段间寄存器的内容，并从成功分支的地址处重新填充流水线。