计算机组成原理 实验报告

姓名：龚小航　学号：PB18151866　实验日期：2020-5-6

一、实验题目：

Lab03 单周期 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：

设计单周期 CPU,能够执行六条指令：add，addi，lw，sw，beq，j. 结构化描述单周期 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 12 个 LED 灯上，以便实物验证。

三、实验平台：

Vivado

四、实验过程：

1. 设计单周期 CPU（不带调试单元）：

先列出该部分的输入输出关系，单周期 CPU 逻辑图如下所示：

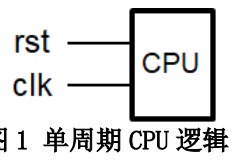
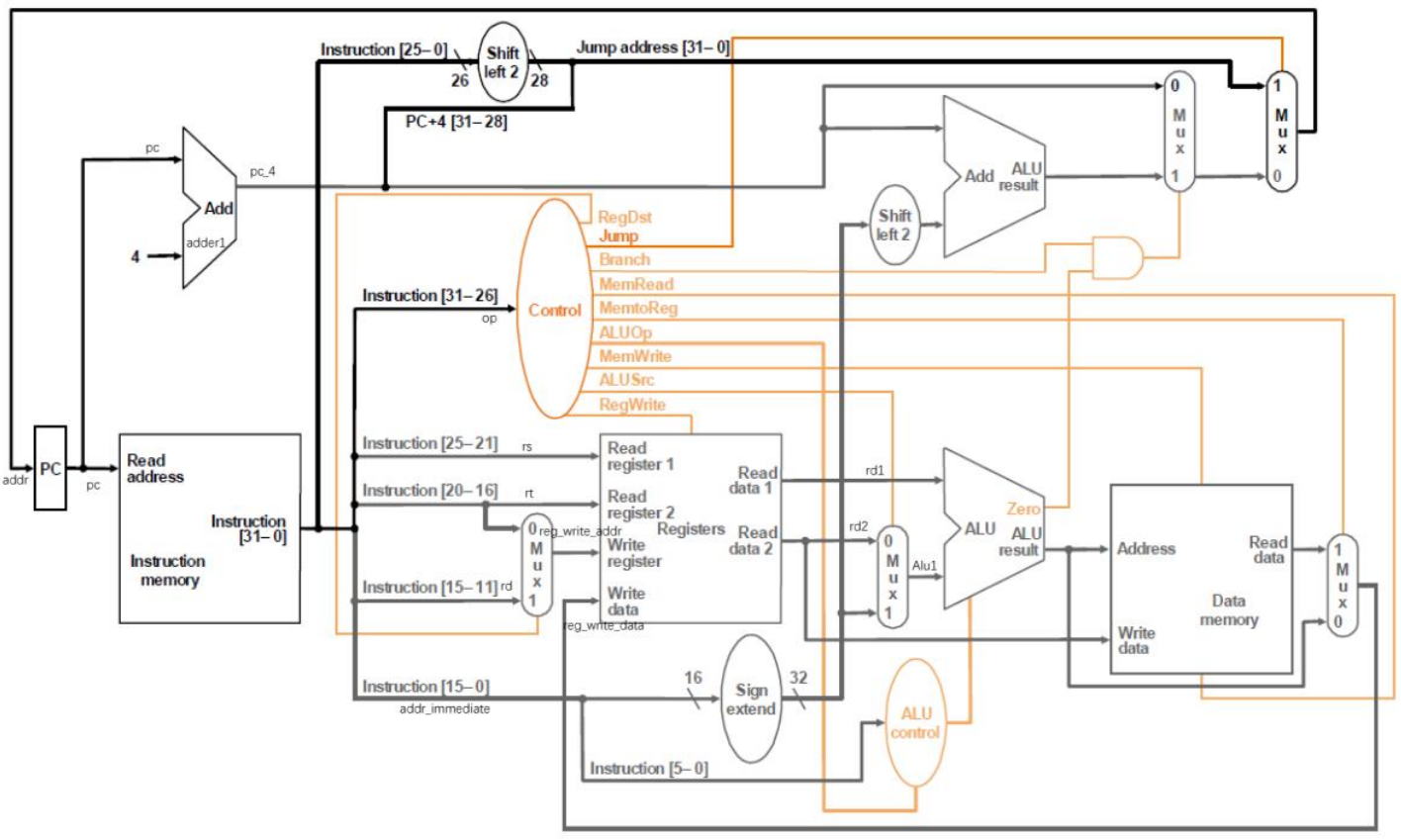


图 1 单周期 CPU 逻辑图

引脚说明：clk 为时钟信号，一个时钟周期内完成指令存储器内的一条指令，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。其中，指令存储器和数据存储器采用例化 IP 核的的方式作为数据支持，指令存储器为深度 256，位宽 32bit 的单端口 ROM；数据存储器为深度 256，位宽 32bit 的单端口 RAM.

以下先简要说明单周期 CPU 工作过程：

- 启动之前：利用.coe 文件对指令存储器和数据存储器进行初始化赋值，即将我们需要运行的测试程序代码存入指令存储器 ROM，这段代码将会产生及更改的数据初始化给数据存储器 RAM 以便测试环境与预想的一致；同时将 pc 置 0 确定开始运行的位置。
- 初始化完成，时钟接入，CPU 开始运行：（以下发生的均在一个时钟内）pc 的初始输出值已被初始化为 0，这个地址被送入指令存储器（不需要时钟，pc 输出值始终存在），ROM 根据输入会立刻在输出端输出 0 号地址的存储数据，即第一条指令。同时，pc+4 信号在这时就已经产生（实际上始终存在）
- 指令产生，执行第一条指令开始：Control Unit 是组合逻辑，一旦 pc 有值传给指令存储器，指令存储器就会输出一串 32 位的数据，控制信号单元就根据这些指令，立刻产生控制信号。

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 指令比较成功，将要跳转；

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。

之后整个系统会根据六条指令之一或是空指令来进行操作。由于 CPU 内部是纯组合逻辑实现（除了写入操作是时钟同步的），因此在一个时钟周期的时间内足够完成任意一条指令（线路延时、门延时等与时钟周期不是同数量级的）。

- 在下一个时钟上升沿来临时：将上一条指令写回寄存器堆或存储器（如果需要），同时将 pc 的输出值更新为外部输入的新 pc 值（pc+4 或跳转），然后根据 pc 的输出值再重新读取一条指令。

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；需要时钟同步写的模块仅有 pc，寄存器堆，数据存储器，其余均以组合逻辑输出。

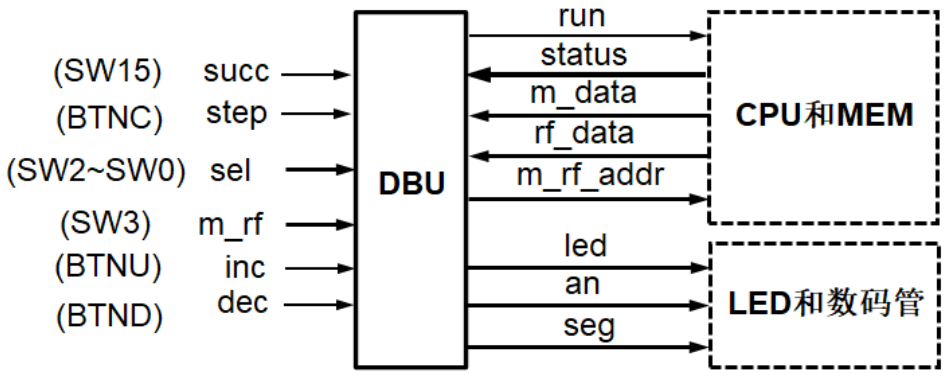
具体的代码附于源文件中，并标有注释。此处不需要具体展开。

仅贴出顶层模块的连接，这是整个程序的总体与框架。

```
module TOP(
    input clk,
    input rst
    );
    wire [31:0] pc,pc_4,ppcc;//ppcc指分支信号和跳转信号两个MUX中间的pc值
    wire [5:0] op; wire [4:0] rs,rt,rd; wire [15:0] addr_immediate; wire[27:0]jumpaddr_28bit;   //取指模块
    wire RegDst,Jump,Branch,MemtoReg,MemWrite,ALUSrc,RegWrite;  wire [2:0] ALUop;  //控制模块
    wire [4:0] reg_write_addr; wire [31:0] reg_write_data; wire [31:0] rd1,rd2;  //寄存器堆模块
    wire [31:0] sign_ext_imm;   //符号扩展模块
    wire [31:0] alu1; /*ALU下方输入*/  wire zero; wire [31:0] result;
    wire [31:0] mem_read;

    wire [31:0] addr;//要传入的pc地址

    PC programcounter (.clk(clk), .rst(rst), .addr(addr), .pc_next(pc));
    Instruction_Memory INSMEM (.pc(pc), .op(op), .rs(rs), .rt(rt), .rd(rd), .addr_immediate(addr_immediate), .jumpaddr_28bit(jumpaddr_28bit));
    adder adder1 (pc,32'h00000004,pc_4);
    Control_Unit control (op,RegDst,Jump,Branch,MemtoReg,ALUop,MemWrite,ALUSrc,RegWrite);
    MUX_2 #(5) MUX_RegDst (.in_0(rt), .in_1(rd), .s(RegDst),  .out(reg_write_addr));
    Register_File REGFILE (.clk(clk), .ra1(rs), .ra2(rt), .rd1(rd1), .rd2(rd2), .RegWrite(RegWrite), .wa(reg_write_addr), .wd(reg_write_data));
    Sign_Extend SE (.immediate(addr_immediate), .out(sign_ext_imm));
    MUX_2 #(32) MUX_ALUSrc (.in_0(rd2), .in_1(sign_ext_imm), .s(ALUSrc),  .out(alu1));
    ALU #(32) ALU (.in0(rd1), .in1(alu1), .ALUop(ALUop), .zero(zero), .result(result));
    Data_Memory DATAMEM (.clk(clk), .addr(result), .mem_write(MemWrite), .readdata(mem_read));
    MUX_2 #(32) MUX_MemtoReg (.in_0(result), .in_1(mem_read), .s(MemtoReg), .out(reg_write_data));
    MUX_2 #(32) MUX_BRANCH (.in_0(pc_4), .in_1(pc_4+(sign_ext_imm<<2)), .s(Branch & zero), .out(ppcc));
    MUX_2 #(32) MUX_JUMP (.in_0(ppcc), .in_1({pc_4[31:28],(jumpaddr_28bit<<2)}), .s(Jump), .out(addr));

endmodule
```

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）

12 个 LED 指示灯 (SW11~SW0) 依次显示控制器的控制信号 (Jump, Branch,Reg_Dst, RegWrite, MemRead, MemtoReg, MemWrite, ALUOp, ALUSrc) 和 ALUZero，其中 ALUOp 为 3 位

8 个数码管显示由 sel 选择的一个 32 位数据

sel = 1：pc_in, PC 的输入数据

sel = 2：pc_out, PC 的输出数据

sel = 3：instr, 指令存储器的输出数据

sel = 4：rf_rd1, 寄存器堆读口 1 的输出数据

sel = 5：rf_rd2, 寄存器堆读口 2 的输出数据

sel = 6：alu_y, ALU 的运算结果

sel = 7：m_rd, 数据存储器的输出数据

可以看出，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单周期 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：

```
75    wire [31:0] pc_in,pc_out;//传入和传出pc的内容
76    wire [31:0] instr;//指令存储器输出的数据
77    wire [31:0] rf_rd1;//寄存器堆读口1的数据
78    wire [31:0] rf_rd2;//寄存器堆读口2的数据
79    wire [31:0] alu_y;//ALU的运算结果
80    wire [31:0] m_rd;
81    wire Jump,Branch,RegDst,RegWrite,MemtoReg,MemWrite;
82    wire [2:0] ALUop;
83    wire ALUSrc,ALUZero;
84
85    TOP test (clk_DBU, rst, m_rf_addr, m_data,rf_data, pc_in, pc_out, instr, rf_rd1, rf_rd2, alu_y, m_rd, Jump,Branch,RegDst,RegWrite,MemtoReg,MemWrite,ALUop,ALUSrc,ALUZero);
86
87    reg [11:0]led_reg;
88    assign led=led_reg;
89    initial led_reg=0;
90
91    always@(*)begin
92      case(sel)
93       3'b000: begin//查看CPU运行结果
94          led_reg = {2'b0,m_rf_addr<<2};
95          if(m_rf) begin   //查看存储器
96              {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=m_data;
97          end
98          else begin     //查看寄存器堆
99              {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=rf_data;
100         end
101      end
102     3'b001: begin
103        led_reg={Jump,Branch,RegDst,RegWrite,1'b0,MemtoReg,MemWrite,ALUop[2],ALUop[1],ALUop[0],ALUSrc,ALUZero};
104        {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=pc_in;
105     end
106     3'b010: begin
107        led_reg={Jump,Branch,RegDst,RegWrite,1'b0,MemtoReg,MemWrite,ALUop[2],ALUop[1],ALUop[0],ALUSrc,ALUZero};
108        {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=pc_out;
109     end
110     3'b011: begin
111        led_reg={Jump,Branch,RegDst,RegWrite,1'b0,MemtoReg,MemWrite,ALUop[2],ALUop[1],ALUop[0],ALUSrc,ALUZero};
112        {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=instr;
113     end
114     3'b100: begin
115        led_reg={Jump,Branch,RegDst,RegWrite,1'b0,MemtoReg,MemWrite,ALUop[2],ALUop[1],ALUop[0],ALUSrc,ALUZero};
116        {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=rf_rd1;
117     end
118     3'b101: begin
119        led_reg={Jump,Branch,RegDst,RegWrite,1'b0,MemtoReg,MemWrite,ALUop[2],ALUop[1],ALUop[0],ALUSrc,ALUZero};
120        {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=rf_rd2;
121     end
122     3'b110: begin
123        led_reg={Jump,Branch,RegDst,RegWrite,1'b0,MemtoReg,MemWrite,ALUop[2],ALUop[1],ALUop[0],ALUSrc,ALUZero};
124        {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=alu_y;
125     end
126     3'b111: begin
127        led_reg={Jump,Branch,RegDst,RegWrite,1'b0,MemtoReg,MemWrite,ALUop[2],ALUop[1],ALUop[0],ALUSrc,ALUZero};
128        {hex7,hex6,hex5,hex4,hex3,hex2,hex1,hex0}=m_rd;
129     end
130     default: ;
131    endcase
132   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。

```
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：

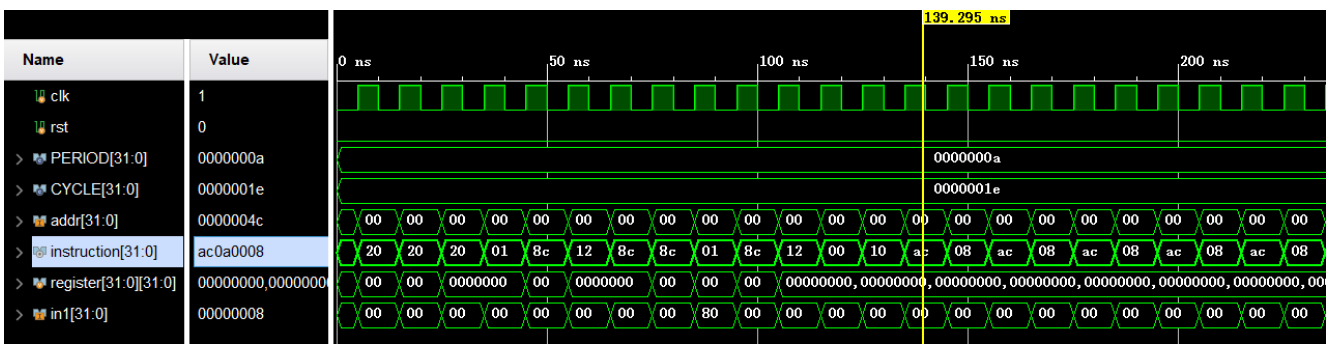
对于单周期 CPU（不带 DBU），初始化其指令存储器和数据存储器，利用一个简单的 MIPS 程序对其进行完整性和正确性的验证即可，所用的程序写成汇编代码如下：

```
# 本文档存储器以字节编址
# 初始PC = 0x00000000

.data
    .word 0,6,0,8,0x80000000,0x80000100,0x100,5,0
                #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                                    0
     addi $t1,$0,5              #t1=5                                    4
     addi $t2,$0,1              #t2=1                                    8
```

```
      addi $t3,$0,0             #t3=0                                    12

      add   $s0,$t1,$t0         #s0=t1+t0=8   测试add指令                16
      lw    $s1,12($0)          #                                        20
      beq   $s1,$s0,_next1      #正确跳到_next                           24

       j _fail

_next1:
      lw $t0, 16($0)            #t0 = 0x80000000  32
      lw $t1, 20($0)            #t1 = 0x80000100  36

      add   $s0,$t1,$t0         #s0 = 0x00000100 = 256                   40
      lw $s1, 24($0)            #                                        44
      beq   $s1,$s0,_next2      #正确跳到_success                        48

       j _fail

_next2:
      add $0, $0, $t2                 #$0应该一直为0                     56
      beq $0,$t3,_success             #                                  60


_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
         j     _fail

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

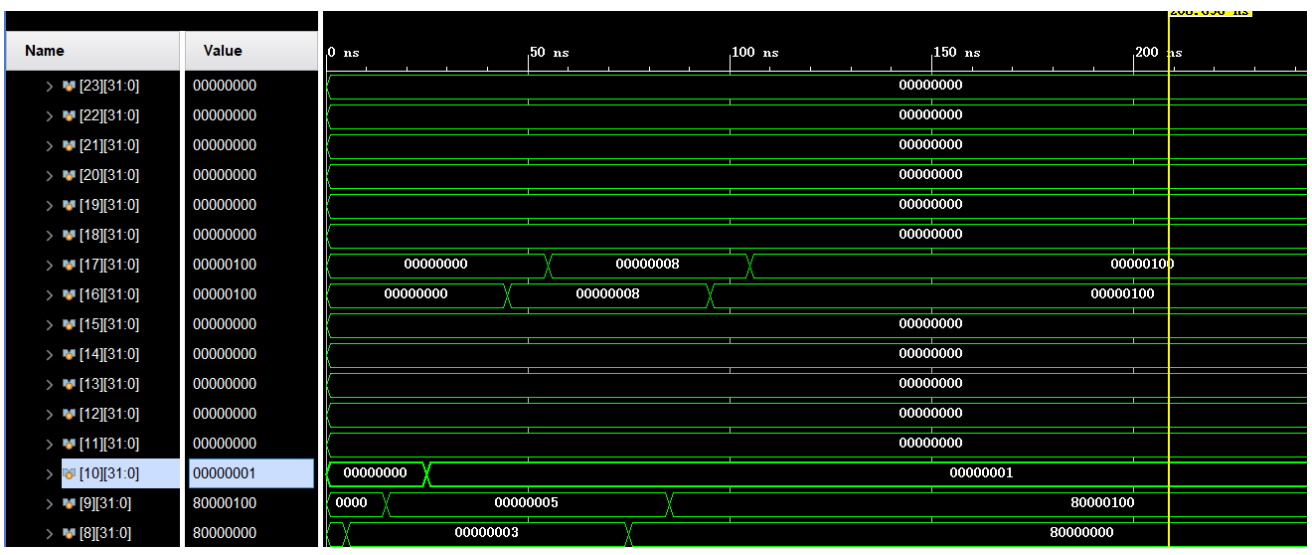
仿真结果如下： 仿真文件只是提供了 clk 信号。



可知最后指令一直在 ac0a0008，08000012，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 =
20080003
20090005
200a0001
200b0000
01288020
8c11000c
12300001
08000010
8c080010
8c090014
01288020
8c110018
12300001
08000010
000a0020
100b0002
ac0b0008
08000010
ac0a0008
08000012
```

再查看最后 10 号寄存器内的值是否为 1：



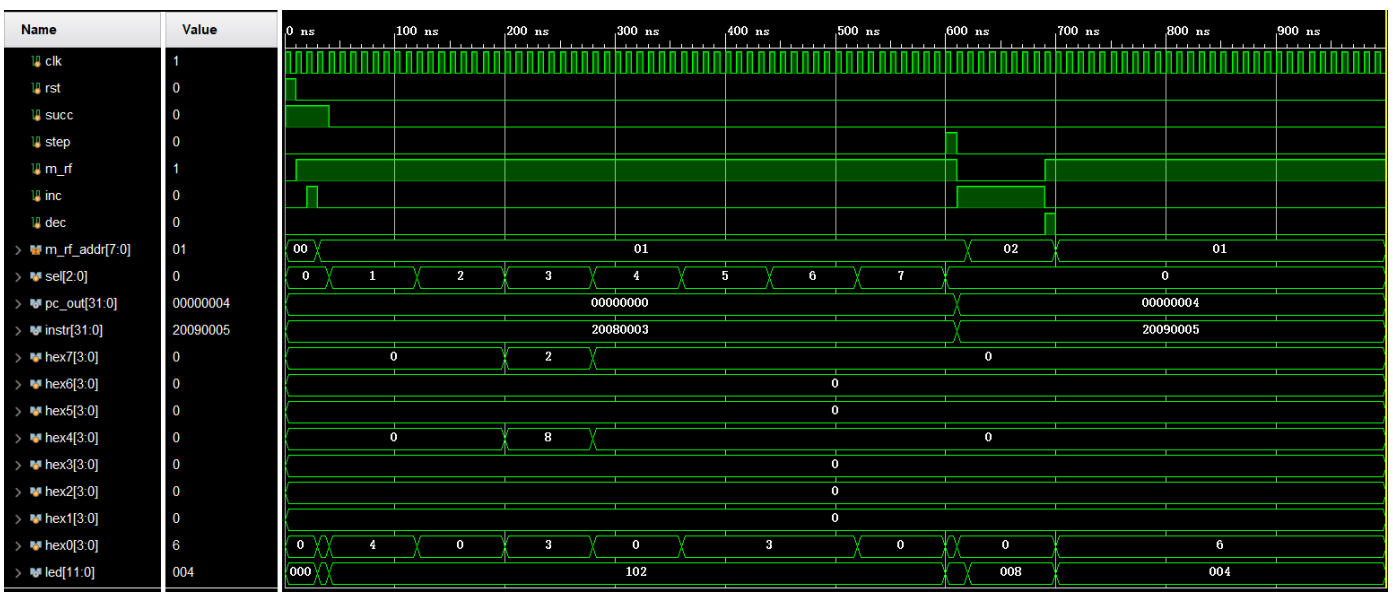
综上，CPU 的功能完整，实现了六条指令的执行。

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 [11:0] led;
    wire [7:0] SSEG_AN,SSEG_CA;
    parameter PERIOD = 10,
              CYCLE = 100;
    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
        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*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，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单周期的 CPU 及相应的调试单元 DBU。

单周期 CPU 的重点在于数据通路的设计。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，都比较简单。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。

单周期 CPU 是 CPU 中结构最简单的一种，设计出完成正确的单周期 CPU，能为多周期、流水线的 CPU 打下基础。

七、思考题

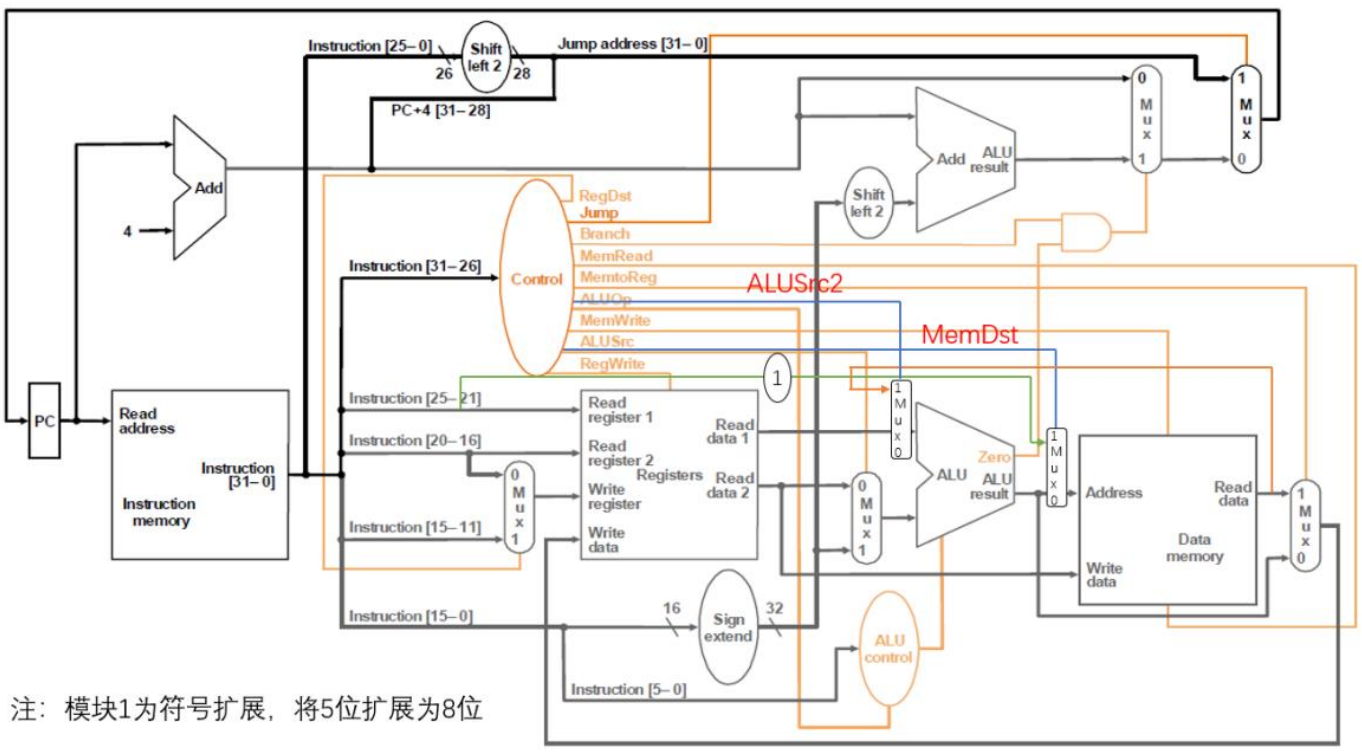
目标：新增一条指令：

accm: rd ← Mem(rs) + rt; op = 000000, funct = 101000

op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)

这条指令的功能是：将来自数据存储器的一个数和另一个来自寄存器堆的数相加，并将结果存入寄存器堆中。

需要修改数据通路，修改后的数据通路如下图所示：



注：模块1为符号扩展，将5位扩展为8位

从图中可以看出，新增了两个多选器，对应的使能信号分别是 ALUSrc2 和 MemDst，ALUSrc2 是用来选择 ALU 的第一个操作数是来自寄存器堆的读口 rd0 或是来自数据存储器读出的数据；MemDst 用来选择数据存储器的读地址来自 ALU 的结果 result 或是来自指令的 rs 部分经符号扩展后的结果。由于只增加了一条指令，数据通路变动不大。

而对应的，控制器也需要加入这两个信号的赋值，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，加入的两个新控制信号为 1；而对于新指令来说，本来存在的那些控制信号值与执行 add 指令时相同。