实验三 单周期CPU

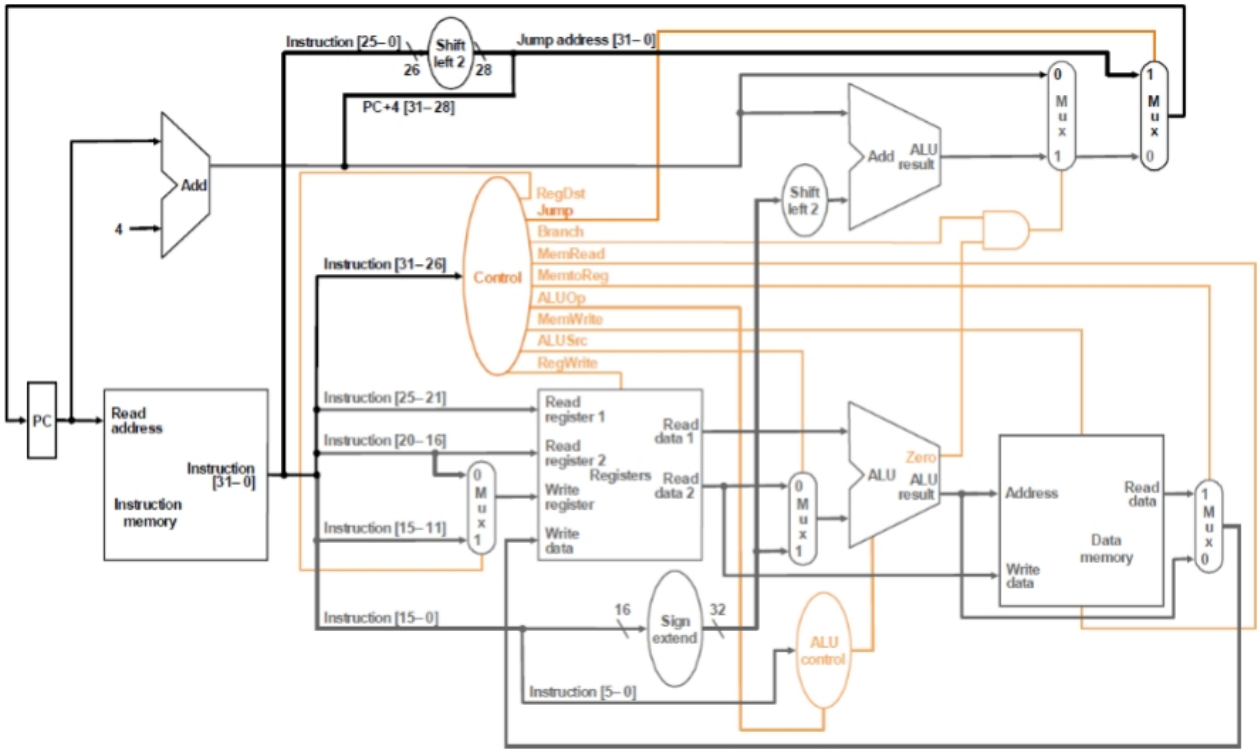
一. 实验目标

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

二. 实验内容

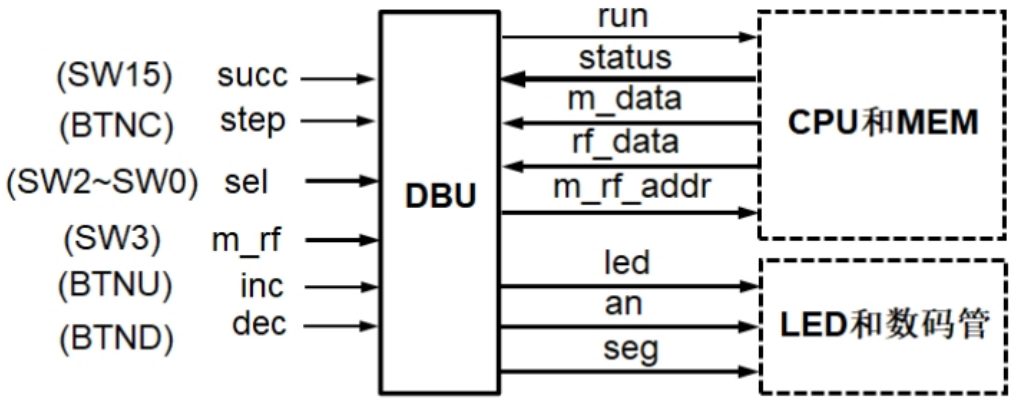
1. 单周期CPU

实现6条指令：add、addi、lw、sw、beq、j



2. 调试单元（Debug Unit，DBU）

为了方便下载调试，设计一个调试单元DBU，该单元可以用于控制CPU的运行方式，显示运行过程的中间状态和最终运行结果。DBU的端口与CPU以及FPGA开发板外设（拨动/按钮开关、LED指示灯、7-段数码管）的连接如图-3所示。为了DBU在不影响CPU运行的情况下，随时监视CPU运行过程中寄存器堆和数据存储器的内容，可以为寄存器堆和数据存储器增加1个用于调试的读端口。



【图中省略了clk (clk100mhz降频)和rst (BTNL)信号】

- 控制CPU运行方式

- succ = 1：控制CPU连续执行指令，run = 1（一直维持）

- succ = 0：控制CPU执行一条指令，每按动step一次，run输出维持一个时钟周期的脉冲

- 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

三. 实验步骤

1. 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；
2. 设计实现调试单元（DBU），并进行功能仿真；

3. 将CPU和DBU下载至FPGA中测试：端口与FPGA开发板N4-DDR的外设的连接关参见图-3所示。

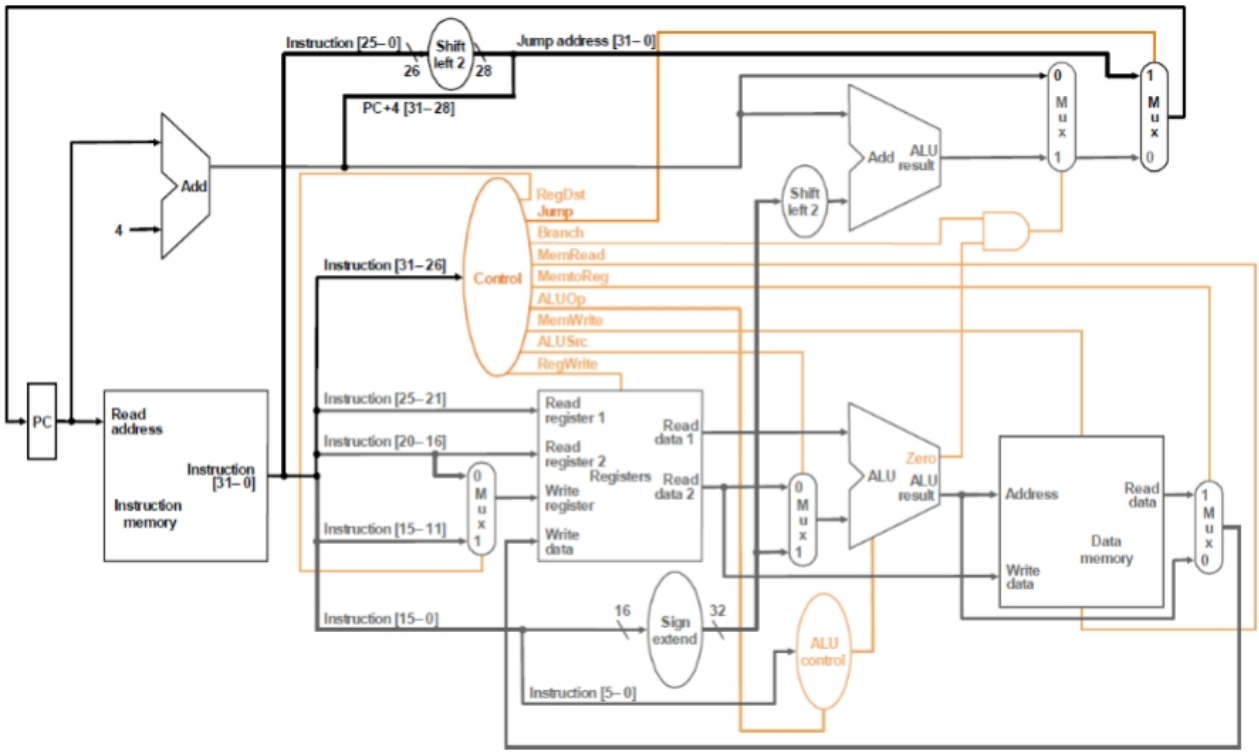
四. 实验检查

1. 检查单周期CPU的功能仿真；
2. 检查调试单元DBU的功能仿真；
3. 检查CPU和DBU下载至FPGA后的运行功能。

五. 实验过程

1. 单周期CPU

设计思路：根据单周期CPU的图分别设计各个模块，并且通过信号拼接实现顶层模块。



ALU使用了lab1设计的：

```
module ALU
#(parameter WIDTH = 32)    //数据宽度
(output reg [WIDTH-1:0] y,     //运算结果
output reg zf,             //零标志
output reg cf,             //进位/借位标志
output reg of,             //溢出标志
input [WIDTH-1:0] a, b,    //两操作数
input [2:0] m          //操作类型
);
always@(*)
    begin
        case(m)
            3'b000: // +
            begin
                {cf, y} = a + b;
```

```
                of = (~a[WIDTH-1] & ~b[WIDTH-1] & y[WIDTH-1])
                    | (a[WIDTH-1] & b[WIDTH-1] & ~y[WIDTH-1]);
                zf = ~|y;
            end
            3'b001: // -
            begin
                {cf, y} = a - b;
                of = (~a[WIDTH-1] & b[WIDTH-1] & y[WIDTH-1])
                    | (a[WIDTH-1] & ~b[WIDTH-1] & ~y[WIDTH-1]);
                zf = ~|y;
            end
            3'b010: // &
            begin
                y = a & b;
                zf = ~|y;
                cf = 0;
                of = 0;
            end
            3'b011: // |
            begin
                y = a | b;
                zf = ~|y;
                cf = 0;
                of = 0;
            end
            3'b100: // ^
            begin
                y = a ^ b;
                zf = ~|y;
                cf = 0;
                of = 0;
            end
            default:
            begin
                y = 0;
                zf = 0;
                cf = 0;
                of = 0;
            end
        endcase
end
endmodule
```

寄存器堆用了lab2设计的：

```
module Registers        //32 x WIDTH寄存器堆
#(parameter WIDTH = 32)           //数据宽度
(
    input clk,            //时钟（上升沿有效）
```

```
    input [4:0] ra0,         //读端口0地址
    output reg [WIDTH-1:0] rd0,          //读端口0数据
    input [4:0] ra1,         //读端口1地址
    output reg [WIDTH-1:0] rd1,        //读端口1数据
    input [4:0] wa,          //写端口地址
    input we,              //写使能，高电平有效
    input [WIDTH-1:0] wd         //写端口数据
);
reg [WIDTH-1:0] mem [255:0];
// 初始化 RAM 的内容
initial
begin
    //$readmemh("C:/Users/mi/Desktop/text.txt", mem, 0, 255);
    $readmemh("C:/Users/mi/Desktop/lab3/initReg.vec", mem, 0, 255);
end
// 异步读
always@(*)
begin
    rd0 = mem[ra0];
    rd1 = mem[ra1];
end
// 同步写
always@(posedge clk)
begin
    if(we)
        mem[wa] <= wd;
end
endmodule
```

用到两种不同数据宽度的多路选择器：

```
module Mux5(
    input control,
    input [4:0] in1, in0,
    output [4:0] out
);
assign out = control? in1:in0;
endmodule

module Mux32(
    input control,
    input [31:0] in1, in0,
    output [31:0] out
);
assign out = control? in1:in0;
endmodule
```

符号扩展：

```
module Sign_extend(
    input [15:0] imm,
    output [31:0] extendImm
);
assign extendImm[15:0] = imm;
assign extendImm[31:16] = imm[15] ? 16'hffff : 16'h0000;
endmodule
```

控制信号：其中MemRead 和 ALUOp 后来没有用上

```
module Control(
    input [5:0] instruction,
    output reg RegDst,
    output reg ALUSrc,
    output reg MemtoReg,
    output reg RegWrite,
    output reg MemRead,
    output reg MemWrite,
    output reg Branch,
    output reg ALUOp1, ALUOp0,
    output reg Jump
);
// add addi lw sw beq j
// x 都归为 0
always @(instruction)
begin
    case(instruction)
        6'b000000: // add
            {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
            MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'b1001000100;
        6'b100011: // lw
            {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
            MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'b0111100000;
        6'b101011: // sw
            {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
            MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'bx1x0010000;
        6'b000100: // beq
            {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
            MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'bx0x0001010;
        6'b001000: // addi
            {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
            MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'b0101000000;
        6'b000010: // j
            {RegDst, ALUSrc, MemtoReg, RegWrite, MemRead,
            MemWrite, Branch, ALUOp1, ALUOp0, Jump} <= 10'bxxx000x011;
    endcase
end
endmodule
```

PC更新模块：

```
module PC(
    input clk,
    input rst,
    input [31:0] new_addr,
    output reg [31:0] cur_addr
);
initial
    cur_addr <= 0;
always@(posedge clk or posedge rst)
begin
    if(rst)
        cur_addr <= 0;
    else
        cur_addr <= new_addr;
end
endmodule
```

顶层模块：按图接线

```
module cpu_one_cycle( // 单周期 CPU
    input clk,         // 时钟（上升沿有效）
    input rst         // 异步复位，高电平有效
);
// PC
wire [31 : 0] new_addr, cur_addr, PCadd;
wire [31 : 0] insturction;        // insturction
wire [31 : 0] ALUresult0;
wire [27 : 0] Shift0;
wire [31 : 0] JumpAddr, Mux4out;
wire [33 : 0] Shift1;
// 信号
wire [5:0] op;
wire RegDst, Jump, Branch, MemtoReg, MemWrite, ALUSrc, RegWrite;
// Registers
wire [4 : 0] ReadReg1, ReadReg2, ReadReg3, WriteReg;
wire [31 : 0] ReadData1, ReadData2, WriteData;
// signExtend
wire [15 : 0] imm;
wire [31 : 0] extendImm;
// ALU
wire [31 : 0] ALUin2, ALUresult1;
wire zero;
wire [2 : 0] m;
// Mem
wire [31 : 0] MemReadData;
```

```
// assign
assign op = insturction[31 : 26];
assign ReadReg1 = insturction[25 : 21];
assign ReadReg2 = insturction[20 : 16];
assign ReadReg3 = insturction[15 : 11];
assign imm = insturction[15 : 0];
assign m = 3'b000;
assign PCadd = cur_addr + 4;
assign Shift0 = {insturction[25 : 0], 2'b00};
assign JumpAddr = {PCadd[31 : 28], Shift0};
assign Shift1 = {extendImm, 2'b00};

Mux32 mux4(
    .control(Branch&zero),
    .in0(PCadd),
    .in1(ALUresult0),
    .out(Mux4out)
);

Mux32 mux5(
    .control(Jump),
    .in0(Mux4out),
    .in1(JumpAddr),
    .out(new_addr)
);

ALU alu0(
    .y(ALUresult0),
    .zf(),
    .cf(),
    .of(),
    .a(PCadd),
    .b(Shift1[31 : 0]),
    .m(m)
);

/*
module PC(
    input clk,
    input rst,
    input [31:0] new_addr,
    output reg [31:0] cur_addr
);*/
PC pc(
    .clk(clk),
    .rst(rst),
    .new_addr(new_addr),
    .cur_addr(cur_addr)
);
```

```
// InstructionMemory 256*32
dist_mem_gen_1 rom(
    .a(cur_addr[9:2]),          // 读地址
    .spo(insturction)       // 读数据
);

/*
module Control(
    input [5:0] insturction,
    output reg RegDst,
    output reg ALUSrc,
    output reg MemtoReg,
    output reg RegWrite,
    output reg MemRead,
    output reg MemWrite,
    output reg Branch,
    output reg ALUOp1, ALUOp0
    output reg Jump
);
*/
Control control(
    .instruction(op),
    .RegDst(RegDst),
    .ALUSrc(ALUSrc),
    .MemtoReg(MemtoReg),
    .RegWrite(RegWrite),
    .MemRead(),
    .MemWrite(MemWrite),
    .Branch(Branch),
    .ALUOp1(),
    .ALUOp0(),
    .Jump(Jump)
);

/*
module Registers        //32 x WIDTH寄存器堆
#(parameter WIDTH = 32)           //数据宽度
(
    input clk,             //时钟（上升沿有效）
    input [4:0] ra0,         //读端口0地址
    output reg [WIDTH-1:0] rd0,          //读端口0数据
    input [4:0] ra1,         //读端口1地址
    output reg [WIDTH-1:0] rd1,        //读端口1数据
    input [4:0] wa,          //写端口地址
    input we,              //写使能，高电平有效
    input [WIDTH-1:0] wd         //写端口数据
);
*/
```

```
Registers registers(
    .clk(clk),
    .ra0(ReadReg1),
    .rd0(ReadData1),
    .ra1(ReadReg2),
    .rd1(ReadData2),
    .wa(WriteReg),
    .we(RegWrite),
    .wd(WriteData)
);

/*
module Mux5(
    input control,
    input [4:0] in1, in0,
    output [4:0] out
);
*/
Mux5 mux0(
    .control(RegDst),
    .in0(ReadReg2),
    .in1(ReadReg3),
    .out(WriteReg)
);

Mux32 mux1(
    .control(ALUSrc),
    .in0(ReadData2),
    .in1(extendImm),
    .out(ALUin2)
);

/*
module Sign_extend(
    input [15:0] imm,
    output [31:0] extendImm
);
*/
Sign_extend signExtend(
    .imm(imm),
    .extendImm(extendImm)
);

/*
module ALU
#(parameter WIDTH = 32)   //数据宽度
(output reg [WIDTH-1:0] y,    //运算结果
output reg zf,          //零标志
output reg cf,          //进位/借位标志
```

```
output reg of,           //溢出标志
input [WIDTH-1:0] a, b,   //两操作数
input [2:0] m          //操作类型
);
*/
ALU alu1(
    .y(ALUresult1),
    .zf(zero),
    .cf(),
    .of(),
    .a(ReadData1),
    .b(ALUin2),
    .m(m)
);

// DataMemory 256*32
dist_mem_gen_0 dist0 (
  .a(ALUresult1[9 : 2]),  // input wire [7 : 0]  地址
  .d(ReadData2),      // input wire [31 : 0] d 写数据
  .clk(clk),          // input wire clk
  .we(MemWrite),      // input wire we 写使能
  .spo(MemReadData)   // output wire [31 : 0] spo 读数据
);

Mux32 mux2(
    .control(MemtoReg),
    .in0(ALUresult1),
    .in1(MemReadData),
    .out(WriteData)
);

endmodule
```

仿真代码：先初始化，时钟持续60周期。

```
module tb();
reg clk;
reg rst;

cpu_one_cycle cpu(
    .clk(clk),
    .rst(rst)
);
parameter PERIOD = 10;
initial
begin
    rst = 1;
    # PERIOD;
    rst = 0;
```

```
end

initial
begin
    clk = 0;
    repeat (60) // 待定
        #(PERIOD/2) clk = ~clk;
    $finish;
end
endmodule
```

ROM初始化文件：

命令的含义是

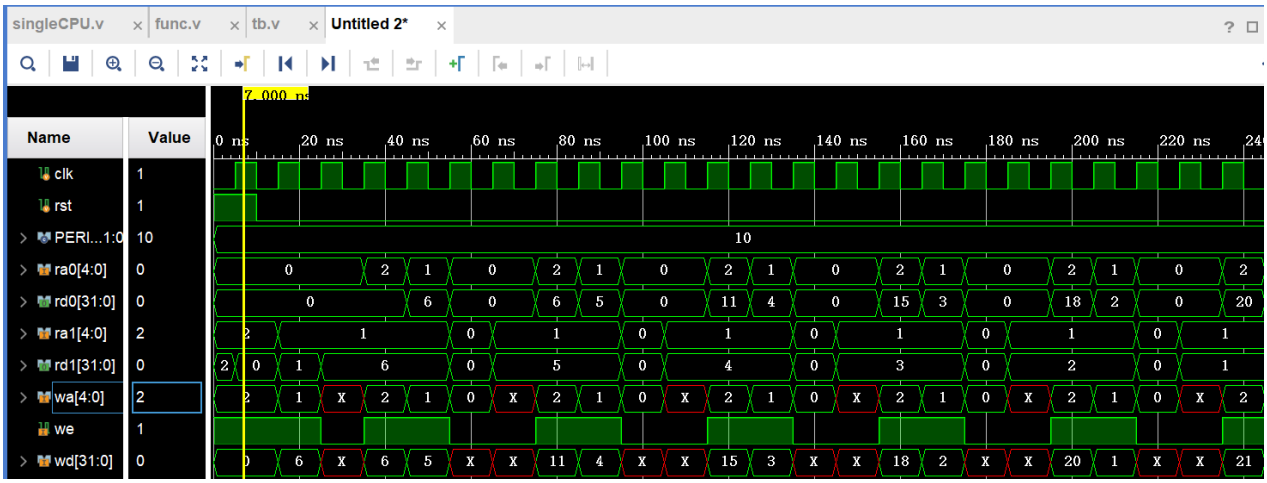
```
addi R2, R0, 0
lw R1, MEM[0]
COMPARE beq R0, R1,END
add R2, R2, R1
addi R1, R1, -1
j COMPARE
END sw R2, MEM[1]
```

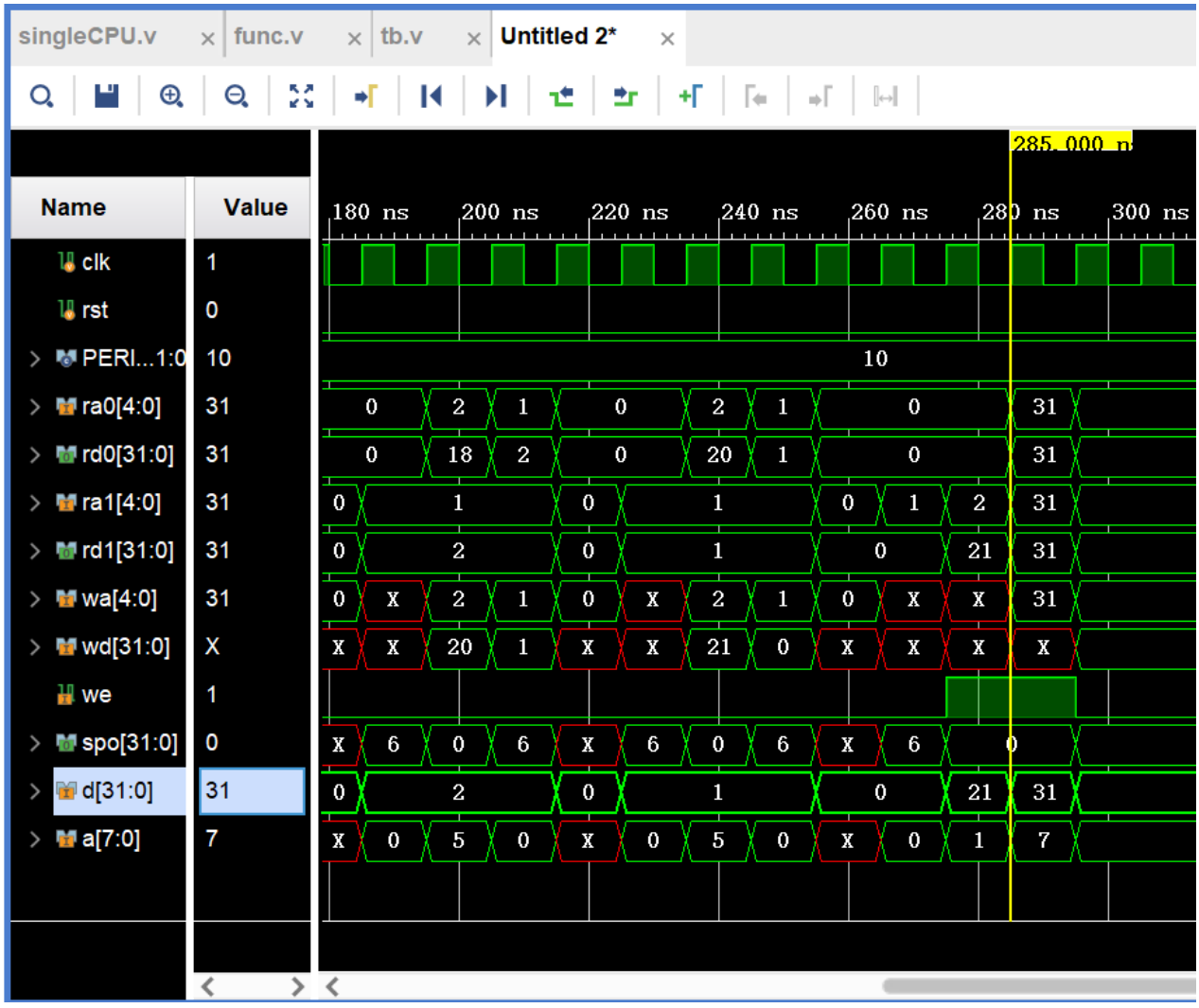
```
memory_initialization_radix = 2;
memory_initialization_vector =
00100000000000100000000000000000,
10001100000000010000000000000000,
00010000000000010000000000000011,
00000000010000010001000000100000,
00100000010000111111111111111111,
00001000000000000000000000000010,
10101100000000100000000000000100,
11111111111111111111111111111111;
```

RAM初始化文件

```
memory_initialization_radix = 2;
memory_initialization_vector =
00000000000000000000000000000110,
00000000000000000000000000000000,
00000000000000000000000000000000;
```

仿真结果如图：





符合预期。

2. 调试单元

DBU：

```
module DBU(
    input clk,
    input rst,
    input succ,
    input step,
```

```
    input [2 : 0] sel,
    input m_rf,
    input inc,
    input dec,
    output [15 : 0] led,
    output reg [7:0] SSEG_CA, // seg
    output reg [7:0] SSEG_AN // an
);
wire run;
wire step_edge, rst_edge;
reg [7 : 0] m_rf_addr;
wire [31 : 0] m_data, rf_data, data0, data;
assign data0 = m_rf? m_data : rf_data;
assign run = succ? clk : step_edge;

// 计数
always@(posedge inc or posedge rst)
begin
    if(rst)
        m_rf_addr <= 0;
    else
        m_rf_addr <= m_rf_addr+1;
end

always@(posedge dec or posedge rst)
begin
    if(rst)
        m_rf_addr <= 0;
    else
        m_rf_addr <= m_rf_addr-1;
end

// 七段数码管
reg [31 : 0] a;
wire [63 : 0] spo;
dist_mem_gen_0 dist_mem_gen_0(
    .a (a[3:0]),
    .spo (spo[7:0])
);
dist_mem_gen_0 dist_mem_gen_1(
    .a (a[7:4]),
    .spo (spo[15:8])
);
dist_mem_gen_0 dist_mem_gen_2(
    .a (a[11:8]),
    .spo (spo[23:16])
);
dist_mem_gen_0 dist_mem_gen_3(
    .a (a[15:12]),
```

```
    .spo (spo[31:24])
);
dist_mem_gen_0 dist_mem_gen_4(
    .a (a[19:16]),
    .spo (spo[39:32])
);
dist_mem_gen_0 dist_mem_gen_5(
    .a (a[23:20]),
    .spo (spo[47:40])
);
dist_mem_gen_0 dist_mem_gen_6(
    .a (a[27:24]),
    .spo (spo[55:48])
);
dist_mem_gen_0 dist_mem_gen_7(
    .a (a[31:28]),
    .spo (spo[63:56])
);

// 分时复用
reg [18:0] timer;
//always@(posedge clk or posedge rst)
always@(posedge clk)
begin
    if(rst)
        timer <= 0;
    else
        timer <= timer+1;
    case (timer[18:16])
        3'b000: begin
            SSEG_CA <= spo[7:0];
            SSEG_AN <= 8'b11111110;
        end
        3'b001: begin
            SSEG_CA <= spo[15:8];
            SSEG_AN <= 8'b11111101;
        end
        3'b010: begin
            SSEG_CA <= spo[23:16];
            SSEG_AN <= 8'b11111011;
        end
        3'b011: begin
            SSEG_CA <= spo[31:24];
            SSEG_AN <= 8'b11110111;
        end
        3'b100: begin
            SSEG_CA <= spo[39:32];
            SSEG_AN <= 8'b11101111;
        end
```

```
        3'b101: begin
            SSEG_CA <= spo[47:40];
            SSEG_AN <= 8'b11011111;
        end
        3'b110: begin
            SSEG_CA <= spo[55:48];
            SSEG_AN <= 8'b10111111;
        end
        3'b111: begin
            SSEG_CA <= spo[63:56];
            SSEG_AN <= 8'b01111111;
        end
        default: SSEG_AN <= 8'b11111111;
    endcase
end

always@(*)
begin
    if(sel == 0)
        a <= data0;
    else
        a <= data;
end

/*
module signal_edge(
    input clk,
    input button,
    output button_edge);*/
signal_edge step0(
    .clk(clk),
    .button(step),
    .button_edge(step_edge)
);

signal_edge rst0(
    .clk(clk),
    .button(rst),
    .button_edge(rst_edge)
);


cpu_one_cycle cpu(
    .clk(run),
    .rst(rst_edge),
    .m_data(m_data),
    .rf_data(rf_data),
    .data(data),
    .sel(sel),
```

```
    .led(led),
    .m_rf_addr(m_rf_addr)
);

endmodule
```

修改过的CPU顶层模块：

```
module cpu_one_cycle( // 单周期 CPU
    input clk,          // 时钟（上升沿有效）
    input rst,          // 异步复位，高电平有效
    output [31 : 0] m_data,
    output [31 : 0] rf_data,
    input [7 : 0] m_rf_addr,
    input [2 : 0] sel,
    output reg [31 : 0] data,
    output reg [15 : 0] led
);
// PC
wire [31 : 0] new_addr, cur_addr, PCadd;
wire [31 : 0] insturction;        // insturction
wire [31 : 0] ALUresult0;
wire [27 : 0] Shift0;
wire [31 : 0] JumpAddr, Mux4out;
wire [33 : 0] Shift1;
// 信号
wire [5 : 0] op;
wire RegDst, Jump, Branch, MemtoReg, MemWrite, ALUSrc, RegWrite, MemRead;
wire [2 : 0] ALUOp;
wire ALUOp1, ALUOp0;
// Registers
wire [4 : 0] ReadReg1, ReadReg2, ReadReg3, WriteReg;
wire [31 : 0] ReadData1, ReadData2, WriteData;
// signExtend
wire [15 : 0] imm;
wire [31 : 0] extendImm;
wire [5 : 0] funct;
// ALU
wire [31 : 0] ALUin2, ALUresult1;
wire zero;
wire [2 : 0] m;
// Mem
wire [31 : 0] MemReadData;

// assign
assign op = insturction[31 : 26];
assign ReadReg1 = insturction[25 : 21];
assign ReadReg2 = insturction[20 : 16];
assign ReadReg3 = insturction[15 : 11];
```

```
assign imm = insturction[15 : 0];
assign m = 3'b000;
assign PCadd = cur_addr + 4;
assign Shift0 = {insturction[25 : 0], 2'b00};
assign JumpAddr = {PCadd[31 : 28], Shift0};
assign Shift1 = {extendImm, 2'b00};
assign funct = insturction[5 : 0];

wire [31 : 0] status;
assign status = {4'b0, Jump, Branch, RegDst, RegWrite, MemRead, MemWrite,
ALUOp, ALUSrc, zero};
always@(*)
begin
    case(sel)
        3'b000:begin
            led <= {8'h0, m_rf_addr};
        end
        3'b001:begin
            led <= status;
            data <= new_addr;
        end
        3'b010:begin
            led <= status;
            data <= cur_addr;
        end
        3'b100:begin
            led <= status;
            data <= insturction;
        end
        3'b101:begin
            led <= status;
            data <= ReadData1;
        end
        3'b110:begin
            led <= status;
            data <= ReadData2;
        end
        3'b111:begin
            led <= status;
            data <= ALUresult1;
        end
    endcase
end

Mux32 mux4(
    .control(Branch&zero),
    .in0(PCadd),
    .in1(ALUresult0),
    .out(Mux4out)
```

```
);

Mux32 mux5(
    .control(Jump),
    .in0(Mux4out),
    .in1(JumpAddr),
    .out(new_addr)
);

ALU alu0(
    .y(ALUresult0),
    .zf(),
    .cf(),
    .of(),
    .a(PCadd),
    .b(Shift1[31 : 0]),
    .m(m)
);

/*
module PC(
    input clk,
    input rst,
    input [31:0] new_addr,
    output reg [31:0] cur_addr
);*/
PC pc(
    .clk(clk),
    .rst(rst),
    .new_addr(new_addr),
    .cur_addr(cur_addr)
);

// InstructionMemory 256*32
dist_mem_gen_1 rom(
    .a(cur_addr[9:2]),          // 读地址
    .spo(insturction)       // 读数据
);

/*
module Control(
    input [5:0] insturction,
    output reg RegDst,
    output reg ALUSrc,
    output reg MemtoReg,
    output reg RegWrite,
    output reg MemRead,
    output reg MemWrite,
    output reg Branch,
```

```
    output reg ALUOp1, ALUOp0
    output reg Jump
);
*/
Control control(
    .instruction(op),
    .RegDst(RegDst),
    .ALUSrc(ALUSrc),
    .MemtoReg(MemtoReg),
    .RegWrite(RegWrite),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .Branch(Branch),
    .ALUOp1(ALUOp1),
    .ALUOp0(ALUOp0),
    .Jump(Jump)
);

/*
module Registers         //32 x WIDTH寄存器堆
#(parameter WIDTH = 32)            //数据宽度
(
    input clk,             //时钟（上升沿有效）
    input [4:0] ra0,         //读端口0地址
    output reg [WIDTH-1:0] rd0,          //读端口0数据
    input [4:0] ra1,         //读端口1地址
    output reg [WIDTH-1:0] rd1,        //读端口1数据
    input [4:0] wa,          //写端口地址
    input we,              //写使能，高电平有效
    input [WIDTH-1:0] wd         //写端口数据
    input [4:0] ra2,         //读端口2地址
    output reg [WIDTH-1:0] rd2,          //读端口2数据
);
*/
Registers registers(
    .clk(clk),
    .ra0(ReadReg1),
    .rd0(ReadData1),
    .ra1(ReadReg2),
    .rd1(ReadData2),
    .wa(WriteReg),
    .we(RegWrite),
    .wd(WriteData),
    .ra2(m_rf_addr[4:0]),
    .rd2(rf_data)
);

/*
module Mux5(
```

```
    input control,
    input [4:0] in1, in0,
    output [4:0] out
);
*/
Mux5 mux0(
    .control(RegDst),
    .in0(ReadReg2),
    .in1(ReadReg3),
    .out(WriteReg)
);

Mux32 mux1(
    .control(ALUSrc),
    .in0(ReadData2),
    .in1(extendImm),
    .out(ALUin2)
);

/*
module Sign_extend(
    input [15:0] imm,
    output [31:0] extendImm
);
*/
Sign_extend signExtend(
    .imm(imm),
    .extendImm(extendImm)
);

/*
module ALUControl(
    input Op1, Op0,
    input [5 : 0] funct,
    output reg [2 : 0] ALUOp
);*/
ALUControl alu_control(
    .Op1(ALUOp1),
    .Op0(ALUOp0),
    .funct(funct),
    .ALUOp(ALUOp)
);

/*
module ALU
#(parameter WIDTH = 32)   //数据宽度
(output reg [WIDTH-1:0] y,    //运算结果
output reg zf,          //零标志
output reg cf,          //进位/借位标志
```

```
output reg of,          //溢出标志
input [WIDTH-1:0] a, b,   //两操作数
input [2:0] m          //操作类型
);
*/
ALU alu1(
    .y(ALUresult1),
    .zf(zero),
    .cf(),
    .of(),
    .a(ReadData1),
    .b(ALUin2),
    .m(ALUOp)
);

/*
// DataMemory 256*32
dist_mem_gen_0 dist0 (
  .a(ALUresult1[9 : 2]),  // input wire [7 : 0]  地址
  .d(ReadData2),      // input wire [31 : 0] d 写数据
  .clk(clk),          // input wire clk
  .we(MemWrite),      // input wire we 写使能
  .spo(MemReadData)  // output wire [31 : 0] spo 读数据
);*/

// DataMemory 256*32
dist_mem_gen_2 dist0 (
  .a(ALUresult1[9 : 2]),  // input wire [7 : 0]  地址
  .d(ReadData2),      // input wire [31 : 0] d 写数据
  .clk(clk),          // input wire clk
  .we(MemWrite),      // input wire we 写使能
  .spo(MemReadData),  // output wire [31 : 0] spo 读数据
  .dpra(m_rf_addr),
  .dpo(m_data)
);


Mux32 mux2(
    .control(MemtoReg),
    .in0(ALUresult1),
    .in1(MemReadData),
    .out(WriteData)
);

endmodule
```

增加的功能模块 ALU Control：

```
module ALUControl(
```

```
    input Op1, Op0,
    input [5 : 0] funct,
    output reg [2 : 0] ALUOp
);
always@(*)
begin
    case({Op1, Op0})
        2'b00: ALUOp <= 3'b000;
        2'b01: ALUOp <= 3'b001;
        2'b10: begin
            case(funct)
                6'b100000: ALUOp <= 3'b000;
                6'b100010: ALUOp <= 3'b001;
                6'b100100: ALUOp <= 3'b010;
                6'b100101: ALUOp <= 3'b011;
                default: ALUOp <= 3'b111;
            endcase
        end
    endcase
end
endmodule
```

仿真文件：

```
module tb();
reg clk, rst, succ, step;
reg [2:0] sel;
reg m_rf;
reg inc, dec;
wire [15 : 0] led;
wire [7 : 0] SSEG_CA, SSEG_AN;

DBU dbu(
    .clk(clk),
    .rst(rst),
    .succ(succ),
    .step(step),
    .sel(sel),
    .m_rf(m_rf),
    .inc(inc),
    .dec(dec),
    .led(led),
    .SSEG_CA(SSEG_CA),
    .SSEG_AN(SSEG_AN)
);

parameter PERIOD = 10;
/*
initial
```

```
begin
    rst = 0;
    succ = 1;
    inc = 0;
    dec = 0;
    # PERIOD;
    rst = 1;
    sel = 0;
    m_rf = 1;
    # PERIOD;
    rst = 0;
end

initial
begin
    inc = 0;
    # (PERIOD*2);
    inc = 1;
    # PERIOD;
    inc = 0;
end
*/
initial
begin
    rst = 0;
    succ = 1;
    # PERIOD;
    rst = 1;
    sel = 7;
    # PERIOD;
    rst = 0;
end

initial
begin
    clk = 0;
    repeat (65) // 待定
        #(PERIOD/2) clk = ~clk;
    $finish;
end
endmodule
```

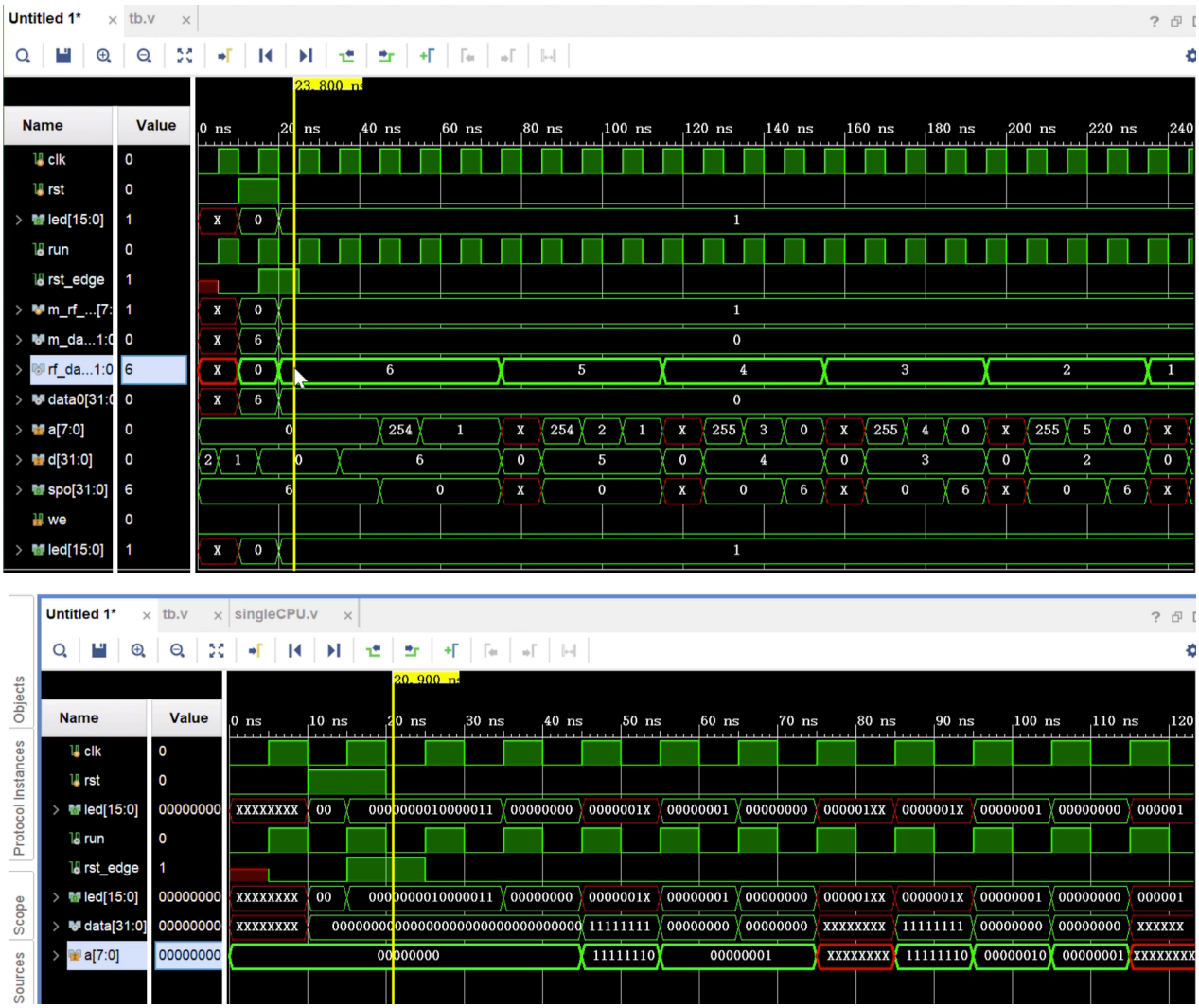
RAM初始化：

```
memory_initialization_radix = 2;
memory_initialization_vector =
00000000000000000000000000000110,
00000000000000000000000000000000,
00000000000000000000000000000000;
```

ROM初始化：

```
memory_initialization_radix = 2;
memory_initialization_vector =
00100000000001000000000000000000,
10001100000000010000000000000000,
00010000000000010000000000000011,
00000000010000010001000000100000,
00100000010000111111111111111111,
00001000000000000000000000000010,
10101100000000010000000000000100,
11111111111111111111111111111111;
```

仿真结果：



全部符合预期。

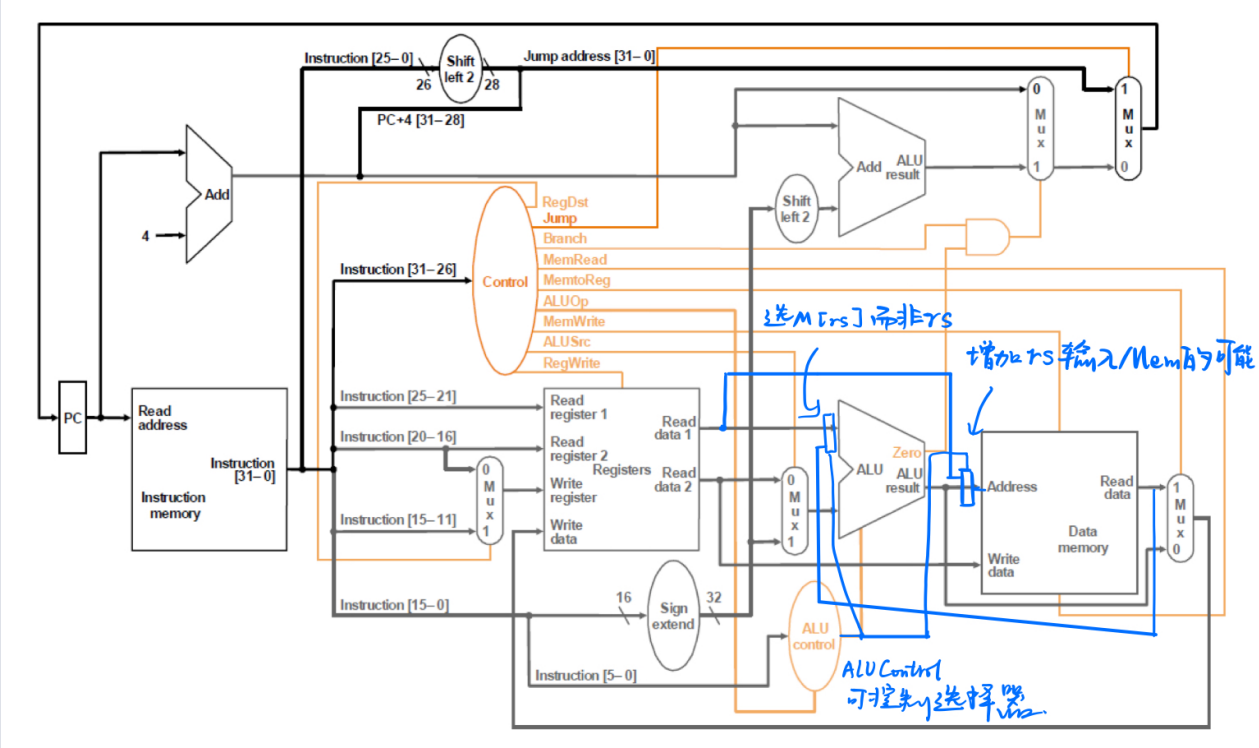
六. 思考题

1. 修改数据通路和控制器，增加支持如下指令：

accm: rd <- M(rs) + rt; op = 000000, funct = 101000

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

示意图：



七. 实验总结

一开始想到设计单周期CPU是非常懵的。但是想通了wire的用处、完全理解了电路与Verilog的对应关系之后，一切就迎刃而解。

在这个过程中，也加深了对调试方法的理解，以及了解到了更多仿真和调试的用途。