同济大学计算机系

计算机组成原理实验报告

学　　号 2151769

姓　　名 吕博文

专　　业 计算机科学与技术

授课老师 陈永生

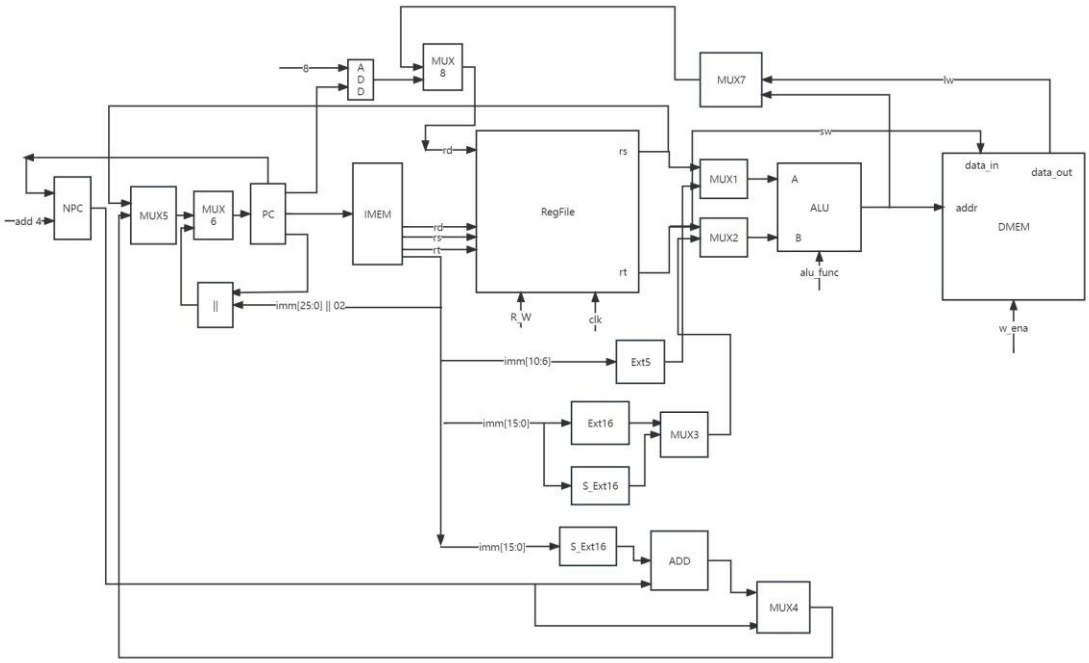
一、实验内容

在本次实验中，我们将使用 Verilog HDL 语言实现 31 条 MIPS 指令 1 的单周期 CPU 的设计和仿真，要求我们自己分析 31 条指令，画出数据通路图，设计控制器，运算器，寄存器单元，内存单元等等并将其通过总线相连，构成并实现 CPU 的功能，通过各个指令的汇编语言测试并和 MARS 文件中的正确结果相比较无误，之后继续通过 Modelsim 完成后仿真，利用七段数码管完成下板实验，完成整个实验。

二、硬件逻辑图

（实验步骤中要求用 logisim 画图的实验，在该部分给出 logisim 原理图，否则该部分在实验报告中不用写）

31 条 CPU 的数据通路图设计如下：



三、模块建模

（该部分要求对实验中建模的所有模块进行功能描述，并列出各模块建模的 verilog 代码）

顶层模块：sccomp_dataflow.v:(通过网站测试的版本)：

```
1.`timescale 1ns / 1ps
2.
3.module sccomp_dataflow(
4.    input clk_in,
5.    input reset,
6.    output[31:0]inst,
7.    output[31:0]pc
```

```
    );
    wire[31:0] _pc;//符合 MARS 格式的 pc
     assign _pc = pc-32'h00400000;
     wire [31:0] _addr,addr;//符合 MARS 格式的地址
     assign _addr = (addr -32'h10010000)/4;//这里除以 4 是因为我们在 DMEM 中定义的 reg 是 32 位的,不同于常规的 8 位，所以要除以 4

     wire[31:0]ram_wdata,ram_rdata;
     wire ram_ena;

     //CPU
     cpu sccpu(
         .clk(clk_in),
         .rst(reset),
         .inst(inst),
         .pc(pc),
         .ram_ena(ram_ena),
         .ram_addr(addr),
         .ram_rdata(ram_rdata),
         .ram_wdata(ram_wdata)
     );

     //ROM
     IMEM rom(
         .addr(_pc[12:2]),
         .inst(inst)
     );

     //RAM
     DMEM ram(
         .clk(clk_in),
         .w_ena(ram_ena),
         .addr(_addr[10:0]),
         .wdata(ram_wdata),
         .rdata(ram_rdata)
     );
endmodule
```

顶层模块：sccomp_dataflow(实验下板的版本)：

```
`timescale 1ns / 1ps

module sccomp_dataflow(
    input clk_in,
    input reset,
    output[7:0] o_seg,
```

```
    output[7:0] o_sel
    );
    wire [31:0]inst;
     wire [31:0]pc;
     wire[31:0] _pc;//符合 MARS 格式的 pc
     assign _pc = pc-32'h00400000;
     wire [31:0] _addr,addr;//符合 MARS 格式的地址
     assign _addr = (addr -32'h10010000)/4;//这里除以 4 是因为我们在 DMEM 中定义的 reg 是 32 位的,不同于常规的 8 位，所以要除以 4

     wire[31:0]ram_wdata,ram_rdata;
     wire ram_ena;

     wire clk_cpu;//下板仿真所用的时钟

     Divider u_divider(
         .clk(clk_in),
         .rst_n(~reset),
         .clk_out(clk_cpu)
     );

     //CPU
     cpu sccpu(
         .clk(clk_cpu),
         .rst(reset),
         .inst(inst),
         .pc(pc),
         .ram_ena(ram_ena),
         .ram_addr(addr),
         .ram_rdata(ram_rdata),
         .ram_wdata(ram_wdata)
     );

     //ROM
     IMEM rom(
         .addr(_pc[12:2]),
         .inst(inst)
     );

     //RAM
     DMEM ram(
         .clk(clk_cpu),
         .w_ena(ram_ena),
         .addr(_addr[10:0]),
```

```
        .wdata(ram_wdata),
        .rdata(ram_rdata)
    );

    seg7x16 seg7x16_inst(
    .clk(clk_in),
    .reset(reset),
    .cs(1'b1),
    .i_data(inst),
    .o_seg(o_seg),
    .o_sel(o_sel)
    );
endmodule
```

CPU 模块：cpu.v:

```
`timescale 1ns / 1ps

module cpu(
    input clk,
    input rst,
    input[31:0]inst,
    output [31:0]pc,
    output ram_ena,
    output[31:0]ram_addr,
     input [31:0]ram_rdata,
     output[31:0]ram_wdata
     );
     //定义需要在模块中调用的变量
     wire[31:0]pc_next;
     wire[4:0]rs,rt;
     wire[31:0]rdata1,rdata2,wdata;
     wire[31:0]alu_a,alu_b,alu_res;
     wire w_ena;
     wire[5:0]alu_func;
     wire[4:0]waddr;
     wire zero,carry,negative,overflow;

     //PC port，完成 PC_next-->PC 的工作
     PCReg pc_reg(
         .clk(clk),
         .rst(rst),
         .ena(1),
         .pc_in(pc_next),
         .pc_out(pc)
     );
```

```
31.
32.     //Control_Unit 模块，完成指令的译码并产生各总线信号的模块
33.     Control_Unit my_control(
34.         .inst(inst),
35.         .pc(pc),
36.         .pc_out(pc_next),
37.         .rs(rs),
38.         .rt(rt),
39.         .rdata1(rdata1),
40.         .rdata2(rdata2),
41.         .alu_func(alu_func),
42.         .alu_a(alu_a),
43.         .alu_b(alu_b),
44.         .ram_rdata(ram_rdata),
45.         .alu_res(alu_res),
46.         .w_ena(w_ena),
47.         .waddr(waddr),
48.         .wdata(wdata),
49.         .ram_ena(ram_ena),
50.         .ram_addr(ram_addr),
51.         .ram_wdata(ram_wdata)
52.     );
53.
54.     //Register
55.     RegFile cpu_ref(
56.         .clk(clk),
57.         .rst(rst),
58.         .w_ena(w_ena),
59.         .waddr(waddr),
60.         .wdata(wdata),
61.         .raddr1(rs),
62.         .rdata1(rdata1),
63.         .raddr2(rt),
64.         .rdata2(rdata2)
65.     );
66.
67.     //ALU
68.     ALU alu_part(
69.         .a(alu_a),
70.         .b(alu_b),
71.         .res(alu_res),
72.         .alu_func(alu_func),
73.         .zero(zero),
74.         .carry(carry),
```

```
        .negative(negative),
        .overflow(overflow)
    );
endmodule
```

控制器 Control_Unit.v:

```
`timescale 1ns / 1ps

//本模块作用是分析指令确定所有控制信号
module Control_Unit(
    input [31:0] inst,//得到的 cpu 指令
    input [31:0] pc,//上一条 pc
    output reg [31:0] pc_out,//因为涉及到转移指令，所以需要分情况讨论下一条 pc 的值

    output [4:0] rs,
     output [4:0] rt,//解析指令得到寄存器编码并输出方便 RegFile 模块调用

     input[31:0] rdata1,
     input [31:0] rdata2,//从 RegFile 里读取到的数据

     output reg[5:0] alu_func,//分析指令得到的 alu 执行操作类型信号
     output reg[31:0] alu_a,
     output reg[31:0] alu_b,

     input [31:0] ram_rdata,//从 DMEM 中读取到的数据
     input [31:0] alu_res,//从 ALU 模块中计算得到的数据

     output reg w_ena,//RegFile 写使能信号
     output [4:0] waddr,//写寄存器信号,指出是哪个寄存器
     output reg [31:0] wdata,//具体要写入寄存器中的值

     output ram_ena,//RAM 使能信号，控制读和写
     output [31:0] ram_addr,//RAM 写入的地址
     output reg [31:0] ram_wdata //RAM 写入的值
     );

     //接下来首先定义几个有关指令具体类型的常量，方便接下来继续编写代码
     //alu_func 的类型
     parameter ADD = 0 ;
     parameter ADDU =1;
     parameter SUB = 2;
     parameter SUBU = 3;
     parameter AND = 4;
     parameter OR = 5;
```

```
39.     parameter XOR = 6;
40.     parameter NOR = 7;
41.     parameter SLT = 8;
42.     parameter SLTU = 9;
43.     parameter SLL = 10;
44.     parameter SRL = 11;
45.     parameter SRA = 12;
46.     parameter SLLV = 13;
47.     parameter SRLV = 14;
48.     parameter SRAV = 15 ;
49.     parameter LUI = 16;
50.     //三种指令类型及其具体分类
51.     parameter No_op = 6'b111111;
52.     parameter No_func = 6'b111111;
53.     parameter R_type_op = 6'b000000;
54.     parameter add_func = 6'b100000;
55.     parameter addu_func = 6'b100001;
56.     parameter sub_func = 6'b100010;
57.     parameter subu_func = 6'b100011;
58.     parameter and_func = 6'b100100;
59.     parameter or_func = 6'b100101;
60.     parameter xor_func = 6'b100110;
61.     parameter nor_func = 6'b100111;
62.     parameter slt_func = 6'b101010;
63.     parameter sltu_func = 6'b101011;
64.     parameter sll_func = 6'b000000;
65.     parameter srl_func = 6'b000010;
66.     parameter sra_func = 6'b000011;
67.     parameter sllv_func = 6'b000100;
68.     parameter srlv_func = 6'b000110;
69.     parameter srav_func = 6'b000111;
70.     parameter jr_func = 6'b001000;
71.
72.     parameter addi_op = 6'b001000;
73.     parameter addiu_op = 6'b001001;
74.     parameter andi_op = 6'b001100;
75.     parameter ori_op = 6'b001101;
76.     parameter xori_op = 6'b001110;
77.     parameter lw_op = 6'b100011;
78.     parameter sw_op = 6'b101011;
79.     parameter beq_op = 6'b000100;
80.     parameter bne_op = 6'b000101;
81.     parameter slti_op = 6'b001010;
82.     parameter sltiu_op = 6'b001011;
```

```
    parameter lui_op = 6'b001111;

    parameter j_op = 6'b000010;
    parameter jal_op = 6'b000011;

    //首先，我们按照所有可能的指令格式将指令分解为可能有意义的几段
    wire[5:0]op;
    wire [4:0]shamt;
    wire [5:0]func;
    wire [15:0] imm;
    wire [25:0] addr;
    wire [5:0] rd;
    assign op = inst[31:26];
    assign shamt = inst[10:6];
    assign func = inst[5:0];
    assign imm = inst[15:0];
    assign addr = inst[25:0];
       assign rs = inst[25:21];
       assign rt = inst[20:16];
       assign rd = inst[15:11];
       wire [31:0] shamt_ex;//shamt 的 32 位扩充
       wire [31:0] imm_ex;//imm 的 32 位扩充
       assign shamt_ex = {27'b0,shamt};
       assign imm_ex = (op == andi_op||op == ori_op||op == xori_op)?{
   16'b0,imm}:{{16{imm[15]}},imm};

       //确定指令需要写入哪个寄存器
       assign waddr = (op == R_type_op)?rd:((op == jal_op)?5'b11111:r
   t);

       //确定所有可能的下一个 pc 的值
       wire[31:0]npc;
       wire [31:0]pc_jump;
       wire [31:0] pc_branch;
       assign npc = pc+4;
       assign pc_jump = {npc[31:28],addr,2'b00};
       assign pc_branch = npc +{{14{imm[15]}},imm,2'b00};

       //根据指令确定 DMEM 的信号
       assign ram_ena = (op == sw_op)?1:0;//1 是写信号，0 是读信号
       assign ram_addr = rdata1 + imm_ex;//lw 或 sw 指令需要的 DMEM 地址
       reg [31:0]load_data;//向 DMEM 中写入的值

       //ALU 部分信号
```

```
125.        always@(*)
126.        begin
127.            //首先给 ALU_data 赋值
128.            case(op)
129.            R_type_op:
130.            begin
131.                case(func)
132.                add_func,sub_func,
133.                addu_func,subu_func,
134.                and_func,or_func,xor_func,
135.                nor_func,slt_func,sltu_func,
136.                sllv_func,srlv_func,srav_func:
137.                begin
138.                    alu_a<=rdata1;
139.                    alu_b<=rdata2;
140.                end
141.                sll_func,srl_func,sra_func:
142.                begin
143.                    alu_a<=shamt_ex;
144.                    alu_b<=rdata2;
145.                end
146.                default:
147.                begin
148.                    alu_a<=rdata1;
149.                    alu_b<=rdata2;
150.                end
151.                endcase
152.            end
153.            addi_op,addiu_op,andi_op,
154.            ori_op,xori_op,slti_op,
155.            sltiu_op,lui_op:
156.            begin
157.                alu_a<=rdata1;
158.                alu_b<=imm_ex;
159.            end
160.            default:
161.            begin
162.                alu_a<=rdata1;
163.                alu_b<=rdata2;
164.            end
165.            endcase
166.
167.            //接下来给 ALU_func 赋值
168.            case(op)
```

```
169.                R_type_op:
170.                begin
171.                    case(func)
172.                        add_func : alu_func<=ADD;
173.                        addu_func : alu_func<=ADDU;
174.                        sub_func : alu_func<=SUB;
175.                        subu_func : alu_func<=SUBU;
176.                        and_func : alu_func<=AND;
177.                        or_func : alu_func<=OR;
178.                        xor_func : alu_func<=XOR;
179.                        nor_func : alu_func<=NOR;
180.                        slt_func : alu_func<=SLT;
181.                        sltu_func : alu_func<=SLTU;
182.                        sll_func : alu_func<=SLL;
183.                        sllv_func : alu_func<=SLLV;
184.                        srl_func : alu_func<=SRL;
185.                        srlv_func : alu_func<=SRLV;
186.                        sra_func : alu_func<=SRA;
187.                        srav_func : alu_func<=SRAV;
188.                        default :alu_func<=ADDU;
189.                    endcase
190.                end
191.                addi_op : alu_func<=ADD;
192.                addiu_op : alu_func<=ADDU;
193.                andi_op : alu_func<=AND;
194.                ori_op : alu_func<=OR;
195.                xori_op : alu_func<=XOR;
196.                slti_op : alu_func<=SLT;
197.                sltiu_op : alu_func<=SLTU;
198.                lui_op : alu_func<=LUI;
199.                default : alu_func<=ADDU;
200.            endcase
201.        end
202.
203.        //RAM 部分信号
204.        always@(*)
205.        begin
206.            //load 信号
207.            case(op)
208.                lw_op : load_data<=ram_rdata;
209.                default : load_data<=ram_rdata;
210.            endcase
211.            //sw 信号
212.            case(op)
```

```
            sw_op : ram_wdata<=rdata2;
            default : ram_wdata<=rdata2;
        endcase
    end

    //RegFile 部分信号
    always@(*)
    begin
        case(op)
            sw_op,beq_op,bne_op,
            j_op : w_ena<=0;
            default : w_ena<=1;
        endcase

        case(op)
            R_type_op : wdata<=alu_res;
            jal_op : wdata<=npc;
            lw_op : wdata<=load_data;
            default : wdata<=alu_res;
        endcase
    end

    //PCRegFile
    always@(*)
    begin
        case(op)
            R_type_op :
            begin
                case(func)
                    jr_func : pc_out<=rdata1;
                    default : pc_out<=npc;
                endcase
            end
            j_op,jal_op : pc_out<=pc_jump;
            beq_op : pc_out<=(rdata1 == rdata2)?pc_branch:npc;
            bne_op : pc_out<=(rdata1!=rdata2)?pc_branch:npc;
            default : pc_out<=npc;
        endcase
    end
endmodule
```

运算器 ALU.v:

```
`timescale 1ns / 1ps

module ALU(
```

```
    input[31:0] a,
    input[31:0] b,
    output[31:0] res,
    input[5:0] alu_func,
    output zero,
    output carry,
     output negative,
     output overflow
     );
     //参数中的 alu_func 代表执行运算的类型，由指令的 opt 和 func 决定，在传入 ALU 模块之前就计算好了
     //下面定义几个运算类型的表示,按照文档顺序定义
     parameter ADD = 0 ;
     parameter ADDU =1;
     parameter SUB = 2;
     parameter SUBU = 3;
     parameter AND = 4;
     parameter OR = 5;
     parameter XOR = 6;
     parameter NOR = 7;
     parameter SLT = 8;
     parameter SLTU = 9;
     parameter SLL = 10;
     parameter SRL = 11;
     parameter SRA = 12;
     parameter SLLV = 13;
     parameter SRLV = 14;
     parameter SRAV = 15 ;
     parameter LUI = 16;
     wire signed [31:0]sign_a,sign_b;//将给定数据转化为有符号数方便接下来进行计算
     assign sign_a = a;
     assign sign_b = b;
     reg[32:0]tmp_res;//暂时存储结果
     always@(*)
     begin
         case(alu_func)
         ADD : tmp_res<=sign_a+sign_b;
         ADDU : tmp_res<=a+b;
         SUB : tmp_res<=sign_a-sign_b;
         SUBU : tmp_res<=a-b;
         AND : tmp_res<=a & b;
         OR : tmp_res<=a | b;
         XOR : tmp_res<= a ^ b;
```

```
        NOR : tmp_res<= ~(a|b);
        SLT : tmp_res<=(sign_a<sign_b);
        SLTU : tmp_res<=(a<b);
        SLL : tmp_res<=(b<<a);
        SRL : tmp_res<=(b>>a);
        SRA : tmp_res<=(sign_b>>>sign_a);
        SLLV : tmp_res<= (b<<a[4:0]);
        SRLV : tmp_res<=(b>>a[4:0]);
        SRAV : tmp_res<=(sign_b>>>sign_a[4:0]);
        LUI : tmp_res<={b[15:0],16'b0};
        endcase
    end
    assign res = tmp_res[31:0];
    assign zero=(tmp_res==32'b0)?1:0;
    assign carry = tmp_res[32];
    assign overflow = tmp_res[32];
    assign negative = tmp_res[31];
endmodule
```

寄存器单元 RegFile.v:

```
`timescale 1ns / 1ps


module RegFile(
    input clk,
    input rst,
    input w_ena,
    input [4:0]waddr,
    input [31:0] wdata,
     input [4:0]raddr1,
     output[31:0]rdata1,
     input [4:0]raddr2,
     output[31:0]rdata2
     );
     reg[31:0] array_reg[0:31];//定义 RegFile 中的寄存器
     integer i;
     //write 信号
     always@(negedge clk or posedge rst)
     begin
         if(rst)begin
             for(i=0;i<32;i=i+1)array_reg[i]<=32'b0;//rst 信号发挥作用，全部寄存器内容清空
         end
         else if(rst==0)begin
             if(w_ena&&waddr!=5'b0)begin
```

```
                //写信号且不能改变寄存器 0 中的内容
                array_reg[waddr]<=wdata;
            end
        end
    end

    //read 信号
    assign rdata1 = array_reg[raddr1];
    assign rdata2 = array_reg[raddr2];
endmodule
```

PCReg.v:

```
`timescale 1ns / 1ps

module PCReg(
    input clk,
    input rst,
    input ena,
    input [31:0]pc_in,
    output  reg[31:0] pc_out
    );
     always@(negedge clk or posedge rst)
     begin
         if(rst)pc_out<=32'h00400000;//这是 MARS 中取指令的开始地址
         else if(rst==0&&ena)pc_out<=pc_in;
     end
endmodule
```

IMEM.v:

```
`timescale 1ns / 1ps


module IMEM(
    input [10:0] addr,
    output [31:0] inst
    );
    //IMEM 模块就是调用了我们事先实现好的 IP 核，给定一个地址，返回对应的指令，在 cpu 指令流程图中与 pc 相连接，起到读取指令的作用
    dist_mem_gen_0 inst_get(.a(addr),.spo(inst));
endmodule
```

DMEM.v:

```
`timescale 1ns / 1ps

module DMEM(
    input clk,
    input w_ena,//控制在存储器中的读写信号
```

```
    input [10:0]addr,
    input [31:0] wdata,
    output [31:0] rdata
    );
     reg[31:0] DM_data[0:31];//32 个 32 位的通用寄存器
     //这里我们向寄存器写采用同步逻辑，而读寄存器内容采用异步逻辑
     always@(negedge clk)
     begin
         if(w_ena)DM_data[addr]<=wdata;
     end
     assign rdata=(w_ena==0?DM_data[addr]:32'bz);
endmodule
```

分频器 Divider.v:

```
`timescale 1ns / 1ps

module Divider(
    input clk,
    input rst_n,
    output reg clk_out
    );
    reg [31:0] count3=32'd0; //50,000,000 分频
    //50,000,000 分频
     always @(posedge clk)
     begin
         if(!rst_n)
         begin
         count3 <= 1'b0;
         clk_out <= 0;
         end
         else if(count3 == 32'd50000000)
         begin
             count3 <= 32'd0;
             clk_out <= ~clk_out;
         end
         else
             count3 <= count3+1'b1;
     end
endmodule
```

1. 七段数码管 seg7x16.v:

```
`timescale 1ns / 1ps

module seg7x16(
    input clk,
    input reset,
```

```
6.    input cs,
7.    input [31:0] i_data,     //需要数码管输出的内容
8.    output [7:0] o_seg,      //输出内容
9.    output [7:0] o_sel       //片选信号
10.    );
11.
12.    reg [14:0] cnt;
13.     always @ (posedge clk, posedge reset)
14.      if (reset)
15.        cnt <= 0;
16.      else
17.        cnt <= cnt + 1'b1;
18.
19.    wire seg7_clk = cnt[14];
20.
21.     reg [2:0] seg7_addr;
22.
23.     always @ (posedge seg7_clk, posedge reset)
24.       if(reset)
25.          seg7_addr <= 0;
26.        else
27.          seg7_addr <= seg7_addr + 1'b1;
28.
29.     reg [7:0] o_sel_r;
30.
31.     always @ (*)
32.       case(seg7_addr)
33.          7 : o_sel_r = 8'b01111111;
34.          6 : o_sel_r = 8'b10111111;
35.          5 : o_sel_r = 8'b11011111;
36.          4 : o_sel_r = 8'b11101111;
37.          3 : o_sel_r = 8'b11110111;
38.          2 : o_sel_r = 8'b11111011;
39.          1 : o_sel_r = 8'b11111101;
40.          0 : o_sel_r = 8'b11111110;
41.        endcase
42.
43.     reg [31:0] i_data_store;
44.     always @ (posedge clk, posedge reset)
45.       if(reset)
46.          i_data_store <= 0;
47.        else if(cs)
48.          i_data_store <= i_data;
49.
```

```
50.     reg [7:0] seg_data_r;
51.     always @ (*)
52.       case(seg7_addr)
53.          0 : seg_data_r = i_data_store[3:0];
54.          1 : seg_data_r = i_data_store[7:4];
55.          2 : seg_data_r = i_data_store[11:8];
56.          3 : seg_data_r = i_data_store[15:12];
57.          4 : seg_data_r = i_data_store[19:16];
58.          5 : seg_data_r = i_data_store[23:20];
59.          6 : seg_data_r = i_data_store[27:24];
60.          7 : seg_data_r = i_data_store[31:28];
61.        endcase
62.
63.     reg [7:0] o_seg_r;
64.     always @ (posedge clk, posedge reset)
65.       if(reset)
66.          o_seg_r <= 8'hff;
67.        else
68.          case(seg_data_r)
69.                4'h0 : o_seg_r <= 8'hC0;
70.                4'h1 : o_seg_r <= 8'hF9;
71.                4'h2 : o_seg_r <= 8'hA4;
72.                4'h3 : o_seg_r <= 8'hB0;
73.                4'h4 : o_seg_r <= 8'h99;
74.                4'h5 : o_seg_r <= 8'h92;
75.                4'h6 : o_seg_r <= 8'h82;
76.                4'h7 : o_seg_r <= 8'hF8;
77.                4'h8 : o_seg_r <= 8'h80;
78.                4'h9 : o_seg_r <= 8'h90;
79.                4'hA : o_seg_r <= 8'h88;
80.                4'hB : o_seg_r <= 8'h83;
81.                4'hC : o_seg_r <= 8'hC6;
82.             4'hD : o_seg_r <= 8'hA1;
83.                4'hE : o_seg_r <= 8'h86;
84.                4'hF : o_seg_r <= 8'h8E;
85.          endcase
86.
87.     assign o_sel = o_sel_r;
88.     assign o_seg = o_seg_r;
89.
90. endmodule
```

四、测试模块建模

（要求列写各建模模块的 test bench 模块代码）

Test_cpu(通过汇编指令测试的前仿真版本):

```
`timescale 1ns / 1ps

module test_tb(

    );
reg clk;            //时钟信号
reg rst;            //复位信号
wire [31:0] inst;   //要执行的指令
wire [31:0] pc;     //下一条指令的地址
reg  [31:0] cnt;    //计数器，已经执行了几条指令
integer file_open;

initial
begin
    clk = 1'b0;
    rst = 1'b1;
    #50 rst = 1'b0;
    cnt = 0;
end

always  #1 clk = ~clk;

always @ (posedge clk) begin
    cnt = cnt + 1'b1;
    if(1)begin
        file_open = $fopen("C:\\Users\\14864\\Desktop\\output_my.txt", "a+");
        $fdisplay(file_open, "OP: %d", cnt);
        $fdisplay(file_open, "Instr_addr = %h", sc_inst.inst);
        $fdisplay(file_open, "$zero = %h", sc_inst.sccpu.cpu_ref.array_reg[0]);
        $fdisplay(file_open, "$at   = %h", sc_inst.sccpu.cpu_ref.array_reg[1]);
        $fdisplay(file_open, "$v0   = %h", sc_inst.sccpu.cpu_ref.array_reg[2]);
        $fdisplay(file_open, "$v1   = %h", sc_inst.sccpu.cpu_ref.array_reg[3]);
        $fdisplay(file_open, "$a0   = %h", sc_inst.sccpu.cpu_ref.array_reg[4]);
```

```
34.         $fdisplay(file_open, "$a1   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[5]);
35.         $fdisplay(file_open, "$a2   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[6]);
36.         $fdisplay(file_open, "$a3   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[7]);
37.         $fdisplay(file_open, "$t0   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[8]);
38.         $fdisplay(file_open, "$t1   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[9]);
39.         $fdisplay(file_open, "$t2   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[10]);
40.         $fdisplay(file_open, "$t3   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[11]);
41.         $fdisplay(file_open, "$t4   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[12]);
42.         $fdisplay(file_open, "$t5   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[13]);
43.         $fdisplay(file_open, "$t6   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[14]);
44.         $fdisplay(file_open, "$t7   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[15]);
45.         $fdisplay(file_open, "$s0   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[16]);
46.         $fdisplay(file_open, "$s1   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[17]);
47.         $fdisplay(file_open, "$s2   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[18]);
48.         $fdisplay(file_open, "$s3   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[19]);
49.         $fdisplay(file_open, "$s4   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[20]);
50.         $fdisplay(file_open, "$s5   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[21]);
51.         $fdisplay(file_open, "$s6   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[22]);
52.         $fdisplay(file_open, "$s7   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[23]);
53.         $fdisplay(file_open, "$t8   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[24]);
54.         $fdisplay(file_open, "$t9   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[25]);
55.         $fdisplay(file_open, "$k0   = %h", sc_inst.sccpu.cpu_ref.arra
    y_reg[26]);
```

```
        $fdisplay(file_open, "$k1   = %h", sc_inst.sccpu.cpu_ref.array_reg[27]);
        $fdisplay(file_open, "$gp   = %h", sc_inst.sccpu.cpu_ref.array_reg[28]);
        $fdisplay(file_open, "$sp   = %h", sc_inst.sccpu.cpu_ref.array_reg[29]);
        $fdisplay(file_open, "$fp   = %h", sc_inst.sccpu.cpu_ref.array_reg[30]);
        $fdisplay(file_open, "$ra   = %h", sc_inst.sccpu.cpu_ref.array_reg[31]);
        $fdisplay(file_open, "$pc   = %h\n", sc_inst.pc);
        $fclose(file_open);
    end
end

sccomp_dataflow sc_inst(
    .clk_in(clk),
    .reset(rst),
    .inst(inst),
    .pc(pc)
);
endmodule
```

Test_cpu(通过汇编指令测试的后仿真版本):

```
`timescale 1ns / 1ps

module test_tb(

    );
reg clk;            //时钟信号
reg rst;            //复位信号
wire [31:0] inst;   //要执行的指令
wire [31:0] pc;     //下一条指令的地址
//integer file_open;

initial
begin
    clk = 1'b0;
    rst = 1'b1;
    #50 rst = 1'b0;
end

always  #50 clk = ~clk;

```

```
sccomp_dataflow sc_inst(
    .clk_in(clk),
    .reset(rst),
    .inst(inst),
    .pc(pc)
);

endmodule
```

五、实验结果

（该部分可截图说明，要求 logisim 逻辑验证图、modelsim 仿真波形图、以及下板后的实验结果贴图（实验步骤中没有下板要求的实验，不需要下板贴图））

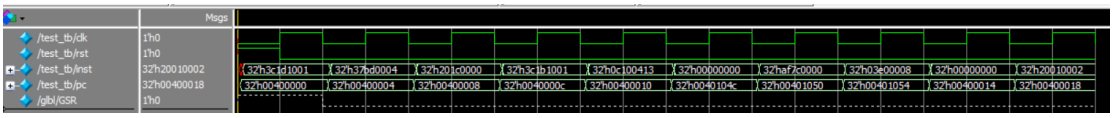
（1） 指令测试：

在指令正确性测试方面，我们将 31 条指令的汇编测试文件在 MARS 中运行得到一份正确结果，同时借助 MARS 将汇编命令变为机器码后转为.coe 文件后导入 vivado 中的 IP 核中，运行产生自己的运行结果，将两份结果进行比较，示例如下：

```
C:\Users\14864\Desktop>fc output_demo.txt output_my.txt
正在比较文件 output_demo.txt 和 OUTPUT_MY.TXT
FC: 找不到差异
```

（2） 后仿真：

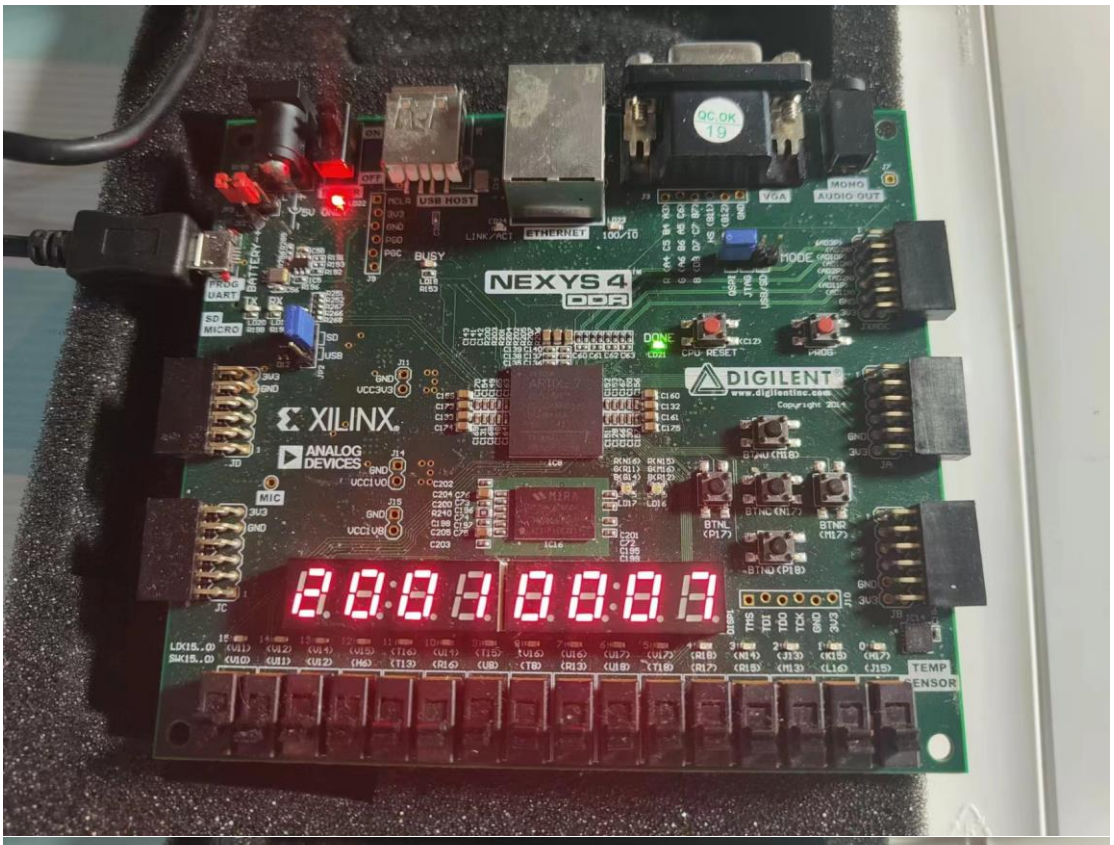
我们通过 Modelsim 中自带的后仿真功能进行仿真结果如下：

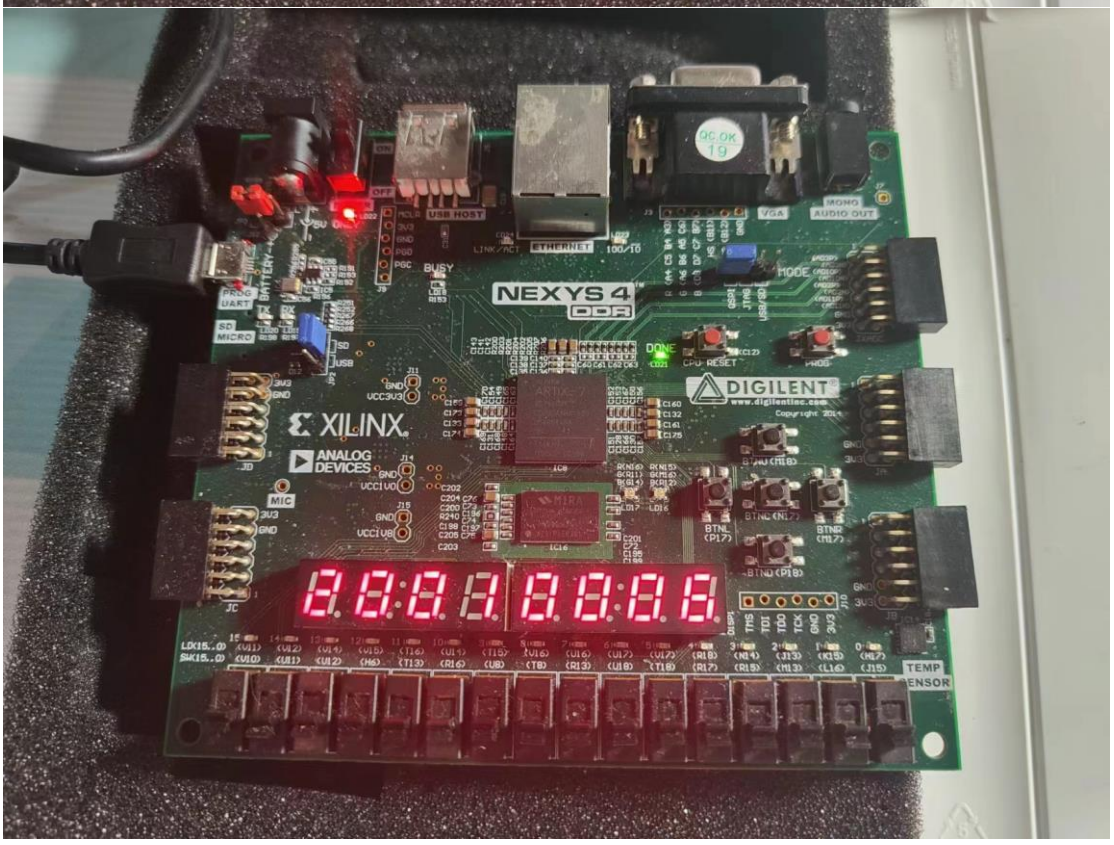


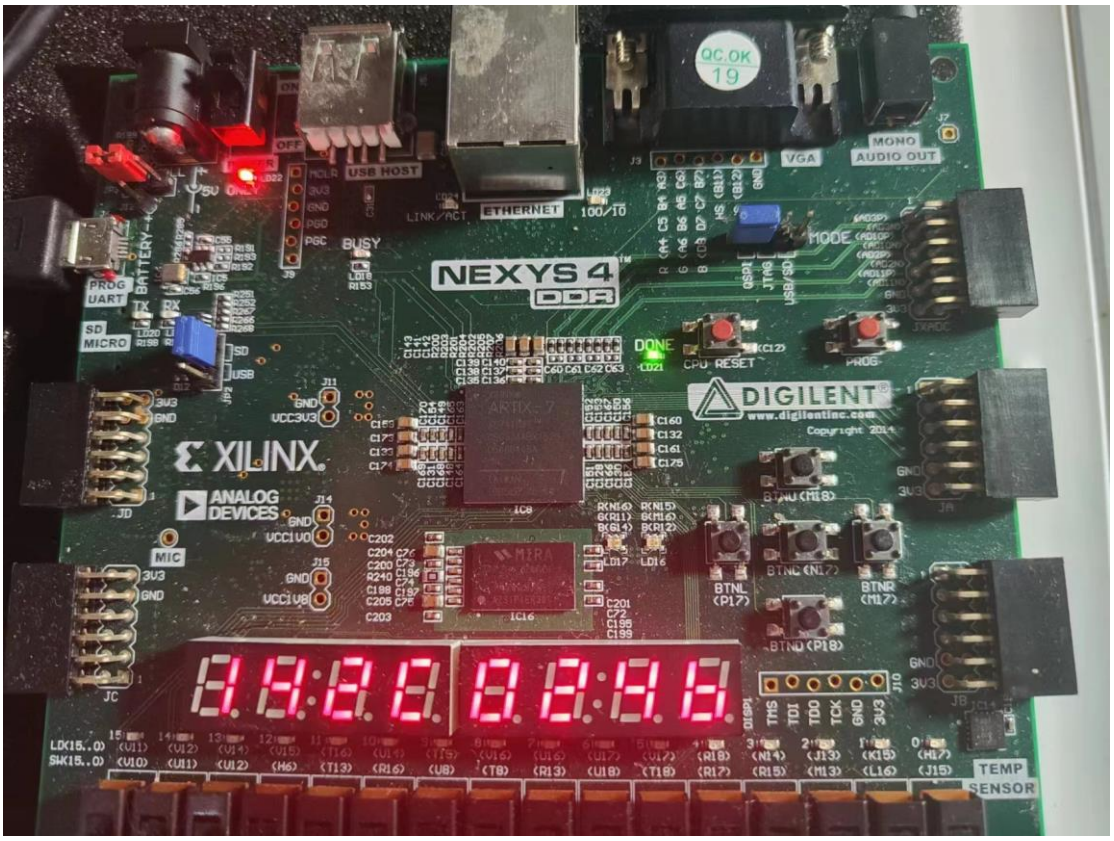
明显可以看出后仿真下门电路出现了一定的延迟效果，更好的仿真了实际电路的结果。

（3） 下板：

我们利用七段数码管模块，通过控制时钟分频，大概控制每 1 秒在七段数码管上显示一条指令，下面只列出几条示例结果：


NEXYS 4
DDR
XILINX
ANALOG DEVICES
DIGILENT
ETHERNET
USB HOST
VGA
MONO AUDIO OUT
PROG UART
SD MICRO
CPU RESET
TEMP SENSOR


NEXYS 4
DDR
XILINX
ANALOG DEVICES
DIGILENT
ETHERNET
USB HOST
VGA
MONO AUDIO OUT
PROG UART
SD MICRO
CPU RESET
TEMP SENSOR


NEXYS 4
DDR
DIGILENT
www.digilentinc.com
XILINX
ANALOG DEVICES
ETHERNET
USB HOST
VGA
MONO AUDIO OUT
PROG UART
SD MICRO
MIC
JC
JD
JA
JB
CPU RESET
PROG
DONE
BTNU (M18)
BTNL (P17)
BTNC (N17)
BTNR (M17)
BTND (P18)
TEMP SENSOR
QC.OK 19