浙江大学

本科实验报告

课程名称: 计算机组成

姓　　名: 李向

学　　院: 计算机科学与技术学院

专　　业: 计算机科学与技术

学　　号: 3180106148

指导教师: 洪奇军

2020 年　8 月　11 日

实验七-- CPU 设计-指令集扩展

实验报告

姓名：李向　　学号：3180106148　　专业：计算机科学与技术

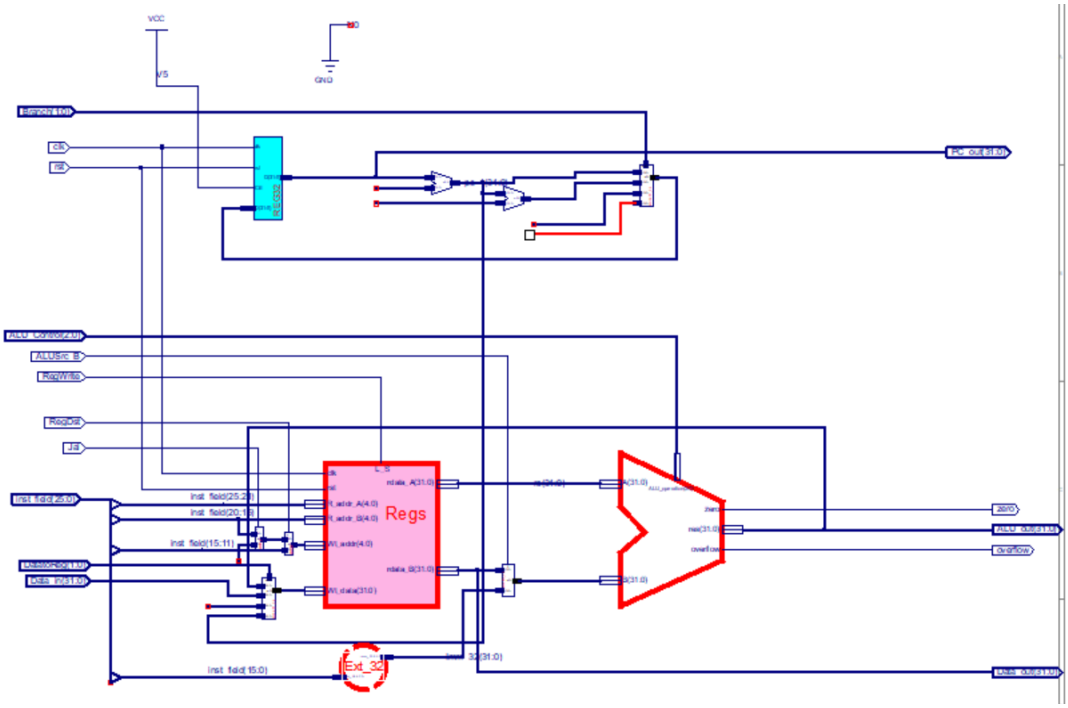
课程名称：计算机组成　　同组学生姓名：无

实验时间：2020-08-11　　实验地点：东四 509　　指导老师：洪奇军

一、操作方法与实验步骤

1. 设计指令扩展后的 Data_path_more 模块

通过原理图设计 Data_path_more 模块，原理图如下：



2. 根据新 DataPath 结构设计控制器 SCPU_ctrl_more

通过结构描述设计 SCPU_ctrl_more 模块，实现代码如下所示：

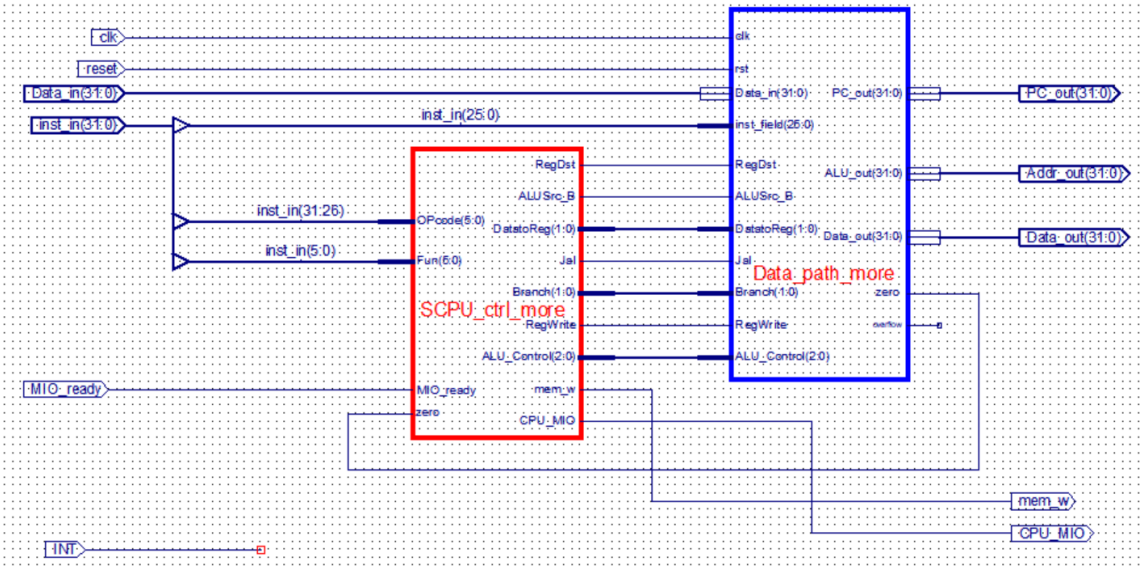
```
module SCPU_ctrl_more( input[5:0]OPcode,                    //OPcode
                       input[5:0]Fun,
    //Function
```

```
                              input MIO_ready,                      //CPU
Wait
                              input zero,

                             output reg RegDst,
                              output reg ALUSrc_B,
                              output reg [1:0]DatatoReg,
                              output reg Jal,
                              output reg [1:0]Branch,
                              output reg RegWrite,
                              output reg mem_w,
                              output reg [2:0]ALU_Control,
                              output reg CPU_MIO
                            );

    `define    CPU_ctrl_signals    {RegDst,    ALU_Control,    ALUSrc_B,
DatatoReg[1:0], Jal, Branch[1:0], RegWrite, mem_w, CPU_MIO}
     always@* begin
        case(OPcode)
            6'b000000: begin
                    case(Fun)
                        6'b100000:
`CPU_ctrl_signals = 13'b1_010_0_00_0_00_100;  //add
                        6'b100010:
`CPU_ctrl_signals = 13'b1_110_0_00_0_00_100;  //sub
                        6'b100100:
`CPU_ctrl_signals = 13'b1_000_0_00_0_00_100;   //and
                        6'b100101:
`CPU_ctrl_signals = 13'b1_001_0_00_0_00_100;   //or
                        6'b100110:
`CPU_ctrl_signals = 13'b1_011_0_00_0_00_100;   //xor
                        6'b100111:
`CPU_ctrl_signals = 13'b1_100_0_00_0_00_100;   //nor
                        6'b101010:
`CPU_ctrl_signals = 13'b1_111_0_00_0_00_100;   //slt
                        6'b000010:
`CPU_ctrl_signals = 13'b1_101_0_00_0_00_100;   //srl
                        6'b001000:
`CPU_ctrl_signals = 13'b1_000_0_00_1_11_000;  //jr
                        6'b001001:
`CPU_ctrl_signals = 13'b1_010_0_11_1_11_100;   //jalr
                    endcase
                end

            6'b001000:
`CPU_ctrl_signals = 13'b0_010_1_00_0_00_100;   //addi
            6'b001100:
`CPU_ctrl_signals = 13'b0_000_1_00_0_00_100;   //andi
            6'b001101:
`CPU_ctrl_signals = 13'b0_001_1_00_0_00_100;   //ori
            6'b101110:
`CPU_ctrl_signals = 13'b0_011_1_00_0_00_100;   //xori
            6'b101111:
`CPU_ctrl_signals = 13'b0_010_0_10_0_00_100;   //lui
            6'b100011:
`CPU_ctrl_signals = 13'b0_010_1_01_0_00_100;   //lw
            6'b101011:
`CPU_ctrl_signals = 13'b1_010_1_00_0_00_010;   //sw
            6'b000100: begin
                                if(zero==1'b1)
`CPU_ctrl_signals = 13'b0_110_0_00_0_01_000;   //beq(equal)
                                else
`CPU_ctrl_signals = 13'b0_110_0_00_0_00_000;   //beq(n_equal)
                            end
            6'b000101: begin
                                if(zero==1'b0)
`CPU_ctrl_signals = 13'b0_110_0_00_0_01_000;   //bne(n_equal)
                                else
```

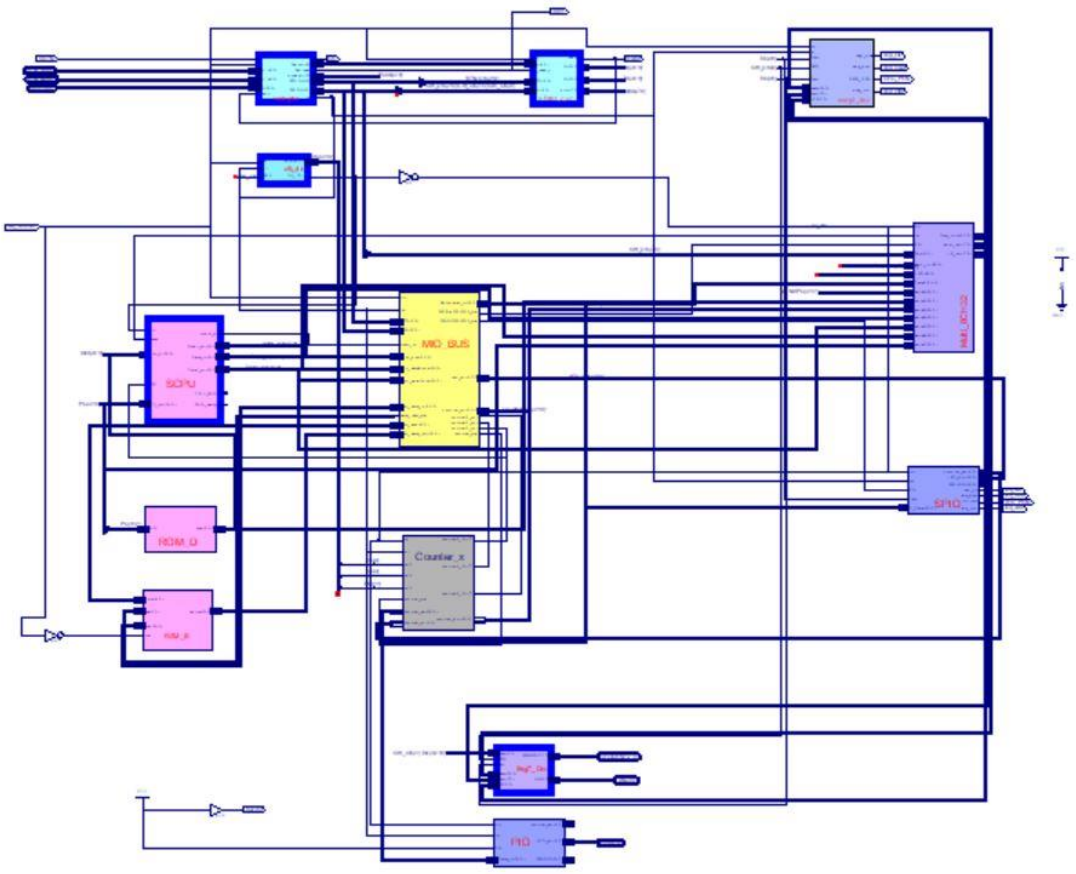
```
`CPU_ctrl_signals = 13'b0_110_0_00_0_00_000;   //bne(equal)
                            end
            6'b001010:
`CPU_ctrl_signals = 13'b0_111_1_00_0_00_100;   //slti

            6'b000010:
`CPU_ctrl_signals = 13'b0_000_0_00_0_10_000;   //j
            6'b000011:
`CPU_ctrl_signals = 13'b0_010_0_11_1_10_100;   //jal
            default:
`CPU_ctrl_signals = 13'b0_000_0_00_0_00_000;
        endcase
    end

endmodule
```

3. 根据新的控制器和数据通路接口信号设计 SCPU 模块

通过原理图设计 SCPU 模块，原理图如下：

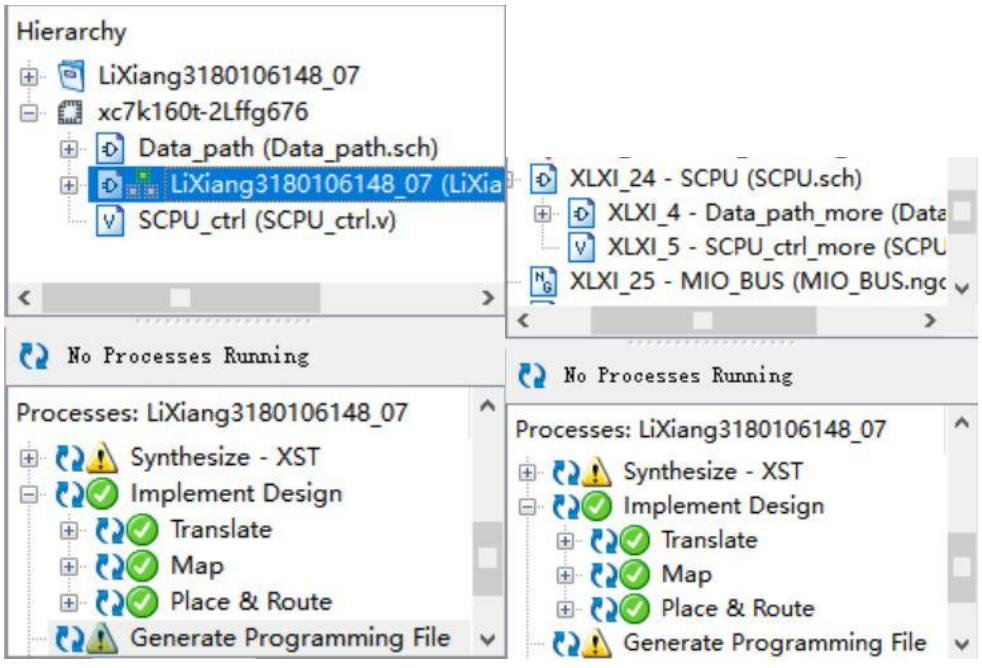


4. 搭建物理验证输入平台



顶层电路图（沿用 Exp06）

直接利用 Exp06 资源重建工程即可。



工程名及 Design 窗口运行成功详细说明/SCPU 核替换后的结构

引脚约束说明:

```
NET "clk_100mhz"                  LOC = AC18     | IOSTANDARD = LVCMOS18 ;
NET "RSTN"                        LOC = W13      | IOSTANDARD = LVCMOS18 ;
NET "clk_100mhz"                   TNM_NET = TM_CLK ;
TIMESPEC TS_CLK_100M       = PERIOD "TM_CLK"      10 ns HIGH 50%;

NET "led_clk"              LOC = N26      | IOSTANDARD = LVCMOS33 ;
NET "led_clrn"             LOC = N24      | IOSTANDARD = LVCMOS33 ;
NET "led_sout"             LOC = M26      | IOSTANDARD = LVCMOS33 ;
NET "LED_PEN"              LOC = P18      | IOSTANDARD = LVCMOS33 ;

NET "seg_clk"              LOC = M24      | IOSTANDARD = LVCMOS33 ;
NET "seg_clrn"             LOC = M20      | IOSTANDARD = LVCMOS33 ;
NET "seg_sout"             LOC = L24      | IOSTANDARD = LVCMOS33 ;
NET "SEG_PEN"              LOC = R18      | IOSTANDARD = LVCMOS33 ;

NET "RDY"          LOC = U21      | IOSTANDARD = LVCMOS33 ;
NET "readn"        LOC = U22      | IOSTANDARD = LVCMOS33 ;
NET "CR"               LOC = V22      | IOSTANDARD = LVCMOS33 ;
#NET "tri_led1_r_n"        LOC = U24      | IOSTANDARD = LVCMOS18 ;
#NET "tri_led1_g_n"        LOC = U25      | IOSTANDARD = LVCMOS18 ;
#NET "tri_led1_b_n"        LOC = V23      | IOSTANDARD = LVCMOS18 ;

NET "BTN_x[0]"         LOC = V17       | IOSTANDARD = LVCMOS18 ;
NET "BTN_x[1]"             LOC = W18      | IOSTANDARD = LVCMOS18 ;
NET "BTN_x[2]"             LOC = W19      | IOSTANDARD = LVCMOS18 ;
NET "BTN_x[3]"             LOC = W15      | IOSTANDARD = LVCMOS18 ;
NET "BTN_x[4]"             LOC = W16      | IOSTANDARD = LVCMOS18 ;



NET "BTN_y[0]"             LOC = V18      | IOSTANDARD = LVCMOS18 ;
NET "BTN_y[1]"             LOC = V19      | IOSTANDARD = LVCMOS18 ;
NET "BTN_y[2]"             LOC = V14      | IOSTANDARD = LVCMOS18 ;
NET "BTN_y[3]"             LOC = W14      | IOSTANDARD = LVCMOS18 ;

NET "SW[0]"            LOC = AA10     | IOSTANDARD = LVCMOS15 ;
NET "SW[1]"            LOC = AB10     | IOSTANDARD = LVCMOS15 ;
NET "SW[2]"            LOC = AA13     | IOSTANDARD = LVCMOS15 ;
NET "SW[3]"            LOC = AA12     | IOSTANDARD = LVCMOS15 ;
NET "SW[4]"            LOC = Y13      | IOSTANDARD = LVCMOS15 ;
NET "SW[5]"            LOC = Y12      | IOSTANDARD = LVCMOS15 ;
NET "SW[6]"            LOC = AD11     | IOSTANDARD = LVCMOS15 ;
NET "SW[7]"            LOC = AD10     | IOSTANDARD = LVCMOS15 ;
NET "SW[8]"            LOC = AE10     | IOSTANDARD = LVCMOS15 ;
NET "SW[9]"            LOC = AE12     | IOSTANDARD = LVCMOS15 ;
NET "SW[10]"           LOC = AF12     | IOSTANDARD = LVCMOS15 ;
NET "SW[11]"           LOC = AE8      | IOSTANDARD = LVCMOS15 ;
NET "SW[12]"           LOC = AF8      | IOSTANDARD = LVCMOS15 ;
NET "SW[13]"           LOC = AE13     | IOSTANDARD = LVCMOS15 ;
NET "SW[14]"           LOC = AF13     | IOSTANDARD = LVCMOS15 ;
NET "SW[15]"           LOC = AF10     | IOSTANDARD = LVCMOS15 ;


#ArDUNIO-IO
NET "Buzzer"           LOC = AF24     | IOSTANDARD = LVCMOS33 ;
NET "SEGMENT[0]"       LOC = AB22     | IOSTANDARD = LVCMOS33 ;#a
NET "SEGMENT[1]"       LOC = AD24      | IOSTANDARD = LVCMOS33 ;#b
NET "SEGMENT[2]"       LOC = AD23      | IOSTANDARD = LVCMOS33 ;
NET "SEGMENT[3]"       LOC = Y21       | IOSTANDARD = LVCMOS33 ;
NET "SEGMENT[4]"       LOC = W20       | IOSTANDARD = LVCMOS33 ;
NET "SEGMENT[5]"       LOC = AC24      | IOSTANDARD = LVCMOS33 ;
NET "SEGMENT[6]"       LOC = AC23      | IOSTANDARD = LVCMOS33 ;#g
NET "SEGMENT[7]"       LOC = AA22      | IOSTANDARD = LVCMOS33 ;#point

NET "AN[0]"            LOC = AD21     | IOSTANDARD = LVCMOS33 ;
```

```
NET "AN[1]"             LOC = AC21      | IOSTANDARD = LVCMOS33 ;
NET "AN[2]"             LOC = AB21      | IOSTANDARD = LVCMOS33 ;
NET "AN[3]"             LOC = AC22      | IOSTANDARD = LVCMOS33 ;

NET "LED[0]"            LOC = AB26      | IOSTANDARD = LVCMOS33 ;
NET "LED[1]"            LOC = W24       | IOSTANDARD = LVCMOS33 ;
NET "LED[2]"            LOC = W23       | IOSTANDARD = LVCMOS33 ;
NET "LED[3]"            LOC = AB25      | IOSTANDARD = LVCMOS33 ;
NET "LED[4]"            LOC = AA25      | IOSTANDARD = LVCMOS33 ;
NET "LED[5]"            LOC = W21       | IOSTANDARD = LVCMOS33 ;
NET "LED[6]"            LOC = V21       | IOSTANDARD = LVCMOS33 ;
NET "LED[7]"            LOC = W26       | IOSTANDARD = LVCMOS33 ;

#NET "PS2_clk"          LOC = N18       | IOSTANDARD = LVCMOS33 ;
#NET "PS2_data"         LOC = M19       | IOSTANDARD = LVCMOS33 ;

#NET "PS2_clk"          LOC = N18       | IOSTANDARD = LVCMOS33 | SLEW =
FAST |  PULLUP ;
#NET "PS2_data"     LOC = M19      | IOSTANDARD = LVCMOS33 | SLEW = FAST |
PULLUP ;

#NET "Blue[0]"          LOC = T20       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Blue[1]"          LOC = R20       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Blue[2]"          LOC = T22       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Blue[3]"          LOC = T23       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Green[0]"         LOC = R22       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Green[1]"         LOC = R23       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Green[2]"         LOC = T24       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Green[3]"         LOC = T25       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Red[0]"           LOC = N21       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Red[1]"           LOC = N22       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Red[2]"           LOC = R21       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "Red[3]"           LOC = P21       | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "HSYNC"                LOC = M22     | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
#NET "VSYNC"                LOC = M21     | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
```

二、实验结果与分析

1. SCPU_ctrl_more 模块仿真结果与分析

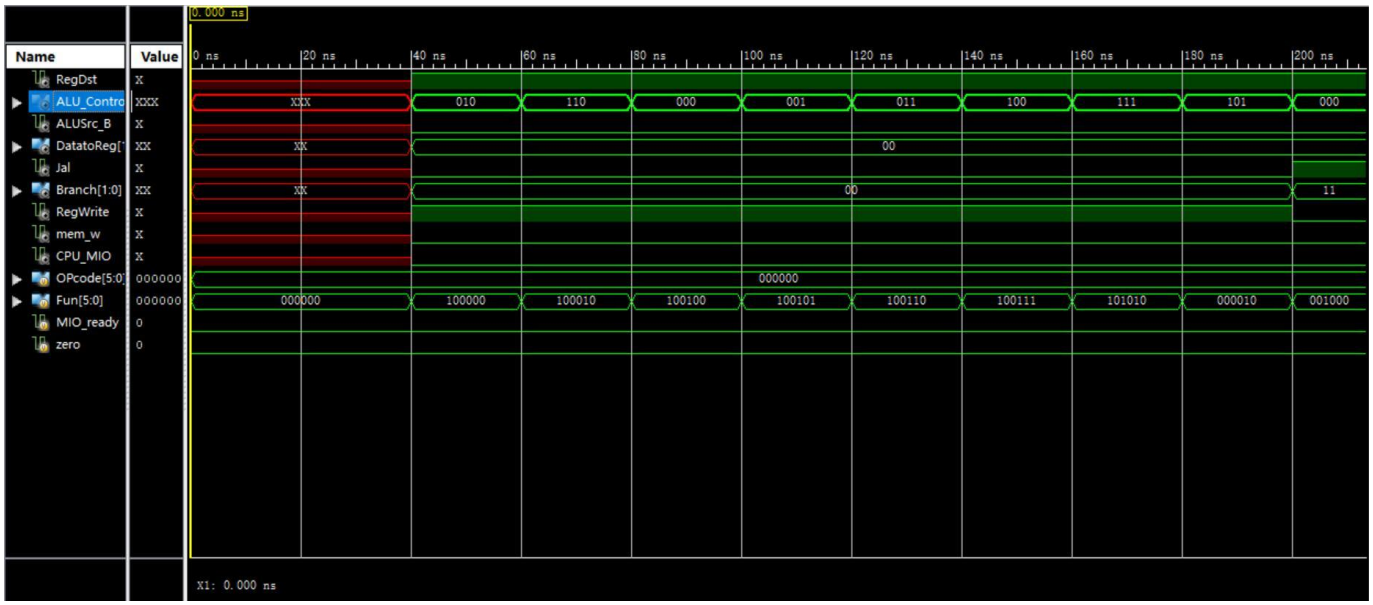
仿真代码:

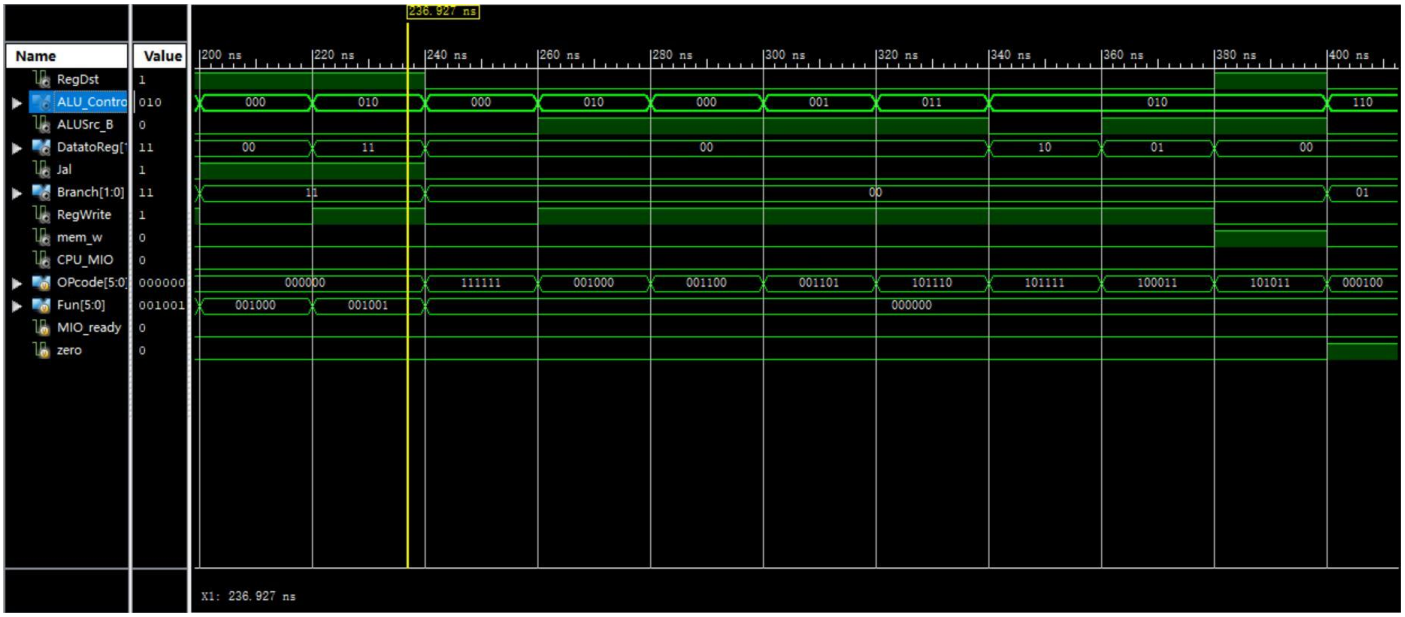
```
initial begin
        // Initialize Inputs
        OPcode = 0;
        Fun = 0;
        MIO_ready = 0;
        zero = 0;
```

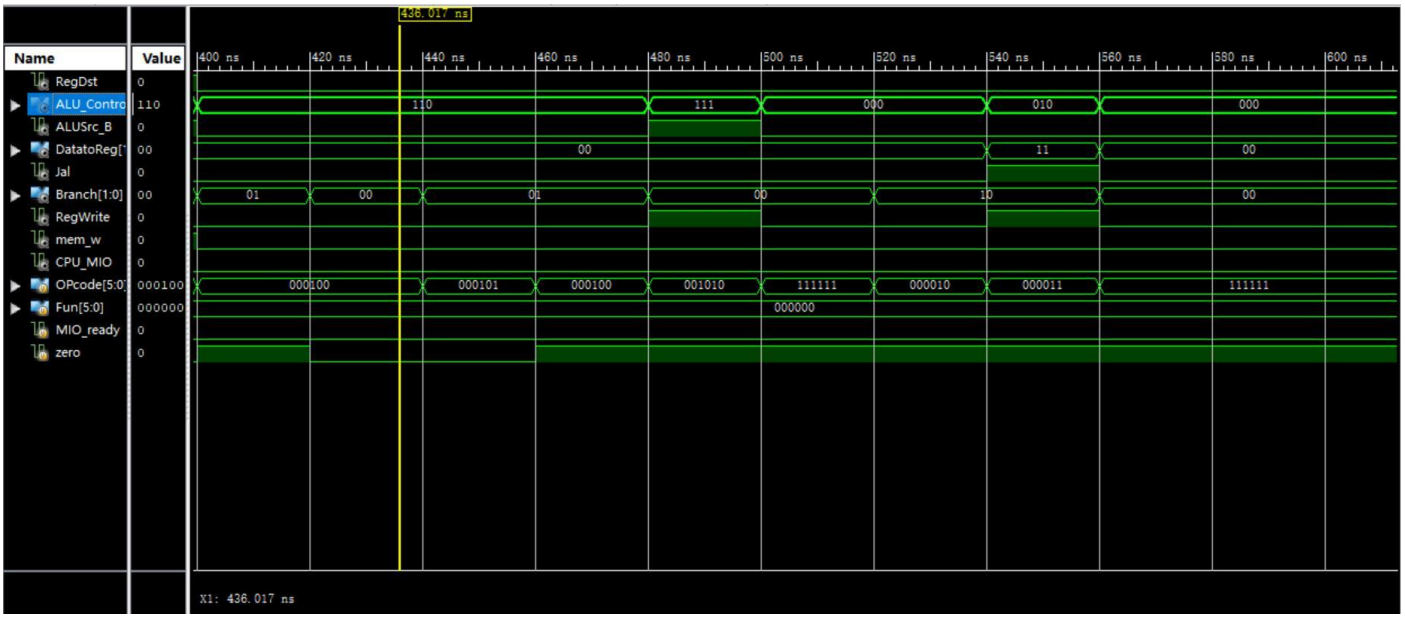
```
        // Wait 40 ns for global reset to finish
        #40;

        //检查输出信号和关键信号输出是否满足真值表

        /***********R 型指令***********/
        /*`define CPU_ctrl_signals {RegDst, ALU_Control, ALUSrc_B,
        DatatoReg[1:0], Jal, Branch[1:0], RegWrite, mem_w, CPU_MIO}*/

        OPcode = 0;
//add,检查`CPU_ctrl_signals = 13'b1_010_0_00_0_00_100
        Fun = 6'b100000;
        #20;
//sub,检查`CPU_ctrl_signals = 13'b1_110_0_00_0_00_100
        Fun = 6'b100010;
        #20;
//and,检查`CPU_ctrl_signals = 13'b1_000_0_00_0_00_100
        Fun = 6'b100100;
        #20;
//or,检查`CPU_ctrl_signals = 13'b1_001_0_00_0_00_100
        Fun = 6'b100101;
        #20;
//xor,检查`CPU_ctrl_signals = 13'b1_011_0_00_0_00_100
        Fun = 6'b100110;
        #20;
//nor,检查`CPU_ctrl_signals = 13'b1_100_0_00_0_00_100
        Fun = 6'b100111;
        #20;
//slt,检查`CPU_ctrl_signals = 13'b1_111_0_00_0_00_100
        Fun = 6'b101010;
        #20;
//srl,检查`CPU_ctrl_signals = 13'b1_101_0_00_0_00_100
        Fun = 6'b000010;
        #20;
//jr,检查`CPU_ctrl_signals = 13'b1_000_0_00_1_11_000
        Fun = 6'b001000;
        #20;
//jalr,检查`CPU_ctrl_signals = 13'b1_010_0_11_1_11_100
        Fun = 6'b001001;
        #20;

        OPcode = 6'b111111; //间隔
        Fun = 6'b000000;
        #20;

        /***********I 型指令***********/

//addi,检查`CPU_ctrl_signals = 13'b0_010_1_00_0_00_100
        OPcode = 6'b001000;
        #20;
//andi,检查`CPU_ctrl_signals = 13'b0_000_1_00_0_00_100
        OPcode = 6'b001100;
        #20;
//ori,检查`CPU_ctrl_signals = 13'b0_001_1_00_0_00_100
        OPcode = 6'b001101;
        #20;
//xori,检查`CPU_ctrl_signals = 13'b0_011_1_00_0_00_100
        OPcode = 6'b101110;
        #20;
//lui,检查`CPU_ctrl_signals = 13'b0_010_0_10_0_00_100
        OPcode = 6'b101111;
        #20;
//lw,检查`CPU_ctrl_signals = 13'b0_010_1_01_0_00_100
        OPcode = 6'b100011;
        #20;
```

```
//sw,检查`CPU_ctrl_signals = 13'b1_010_1_00_0_00_010
        OPcode = 6'b101011;
        #20;
//beq(equal),检查`CPU_ctrl_signals = 13'b0_110_0_00_0_01_000
        OPcode = 6'b000100;
        zero = 1;
        #20;
//beq(n_equal),检查`CPU_ctrl_signals = 13'b0_110_0_00_0_00_000
        OPcode = 6'b000100;
        zero = 0;
        #20;
//bne(n_equal),检查`CPU_ctrl_signals = 13'b0_110_0_00_0_01_000
        OPcode = 6'b000101;
        zero = 0;
        #20;
//bne(equal),检查`CPU_ctrl_signals = 13'b0_110_0_00_0_00_000
        OPcode = 6'b000100;
        zero = 1;
        #20;
//slti,检查`CPU_ctrl_signals = 13'b0_111_1_00_0_00_100
        OPcode = 6'b001010;
        #20;

        OPcode = 6'b111111; //间隔
        #20;

        /***********J型指令***********/

//j,检查`CPU_ctrl_signals = 13'b0_000_0_00_0_10_000
        OPcode = 6'b000010;
        #20;
//jal,检查`CPU_ctrl_signals = 13'b0_010_0_11_1_10_100
        OPcode = 6'b000011;
        #20;

        OPcode = 6'b111111; //间隔
        #20;
    end
```

仿真结果图:




236.927 ns
Name
Value
RegDst
ALU_Contro
ALUSrc_B
DatatoReg[
Jal
Branch[1:0]
RegWrite
mem_w
CPU_MIO
OPcode[5:0]
Fun[5:0]
MIO_ready
zero
X1: 236.927 ns


436.017 ns
Name
Value
RegDst
ALU_Contro
ALUSrc_B
DatatoReg[
Jal
Branch[1:0]
RegWrite
mem_w
CPU_MIO
OPcode[5:0]
Fun[5:0]
MIO_ready
zero
X1: 436.017 ns

	RegDst	ALU_Control	ALUSrc_B	DatatoReg	Jal	Branch	RegWrite	Mem_w	CPU_MIO
ADD	1	010	0	00	0	00	1	0	0
SUB	1	110	0	00	0	00	1	0	0
AND	1	000	0	00	0	00	1	0	0
OR	1	001	0	00	0	00	1	0	0
XOR	1	011	0	00	0	00	1	0	0
NOR	1	100	0	00	0	00	1	0	0
SLT	1	111	0	00	0	00	1	0	0
SRL	1	101	0	00	0	00	1	0	0
ERET	1	010	0	00	0	11	1	0	0
JALR	1	010	0	11	1	11	1	0	0
ADDI	0	010	1	00	0	00	1	0	0
ANDI	0	000	1	00	0	00	1	0	0
ORI	0	001	1	00	0	00	1	0	0
XORI	0	011	1	00	0	00	1	0	0
SLTI	0	111	1	00	0	00	1	0	0
LW	0	010	1	01	0	00	1	0	0
SW	1	010	1	00	0	00	0	1	0
BEQ(EQ)	0	110	0	00	0	01	0	0	0
~EQ	0	110	0	00	0	00	0	0	0
BNE(~EQ)	0	110	0	00	0	01	0	0	0
EQ	0	110	0	00	0	00	0	0	0
JR	1	000	0	00	1	11	0	0	0
J	0	000	0	00	0	10	0	0	0
JAL	0	010	0	11	1	10	1	0	0
LUI	0	010	0	10	0	00	1	0	0

如图所示，按照仿真代码的注释所写，依次对各个指令进行检查，可以看出仿真结果与 PPT 提供的各指令应该输出的控制信号值的表格完全相同，说明成功实现了 SCPU_ctrl_more 模块的设计。

2. Data_path_more 模块仿真结果与分析

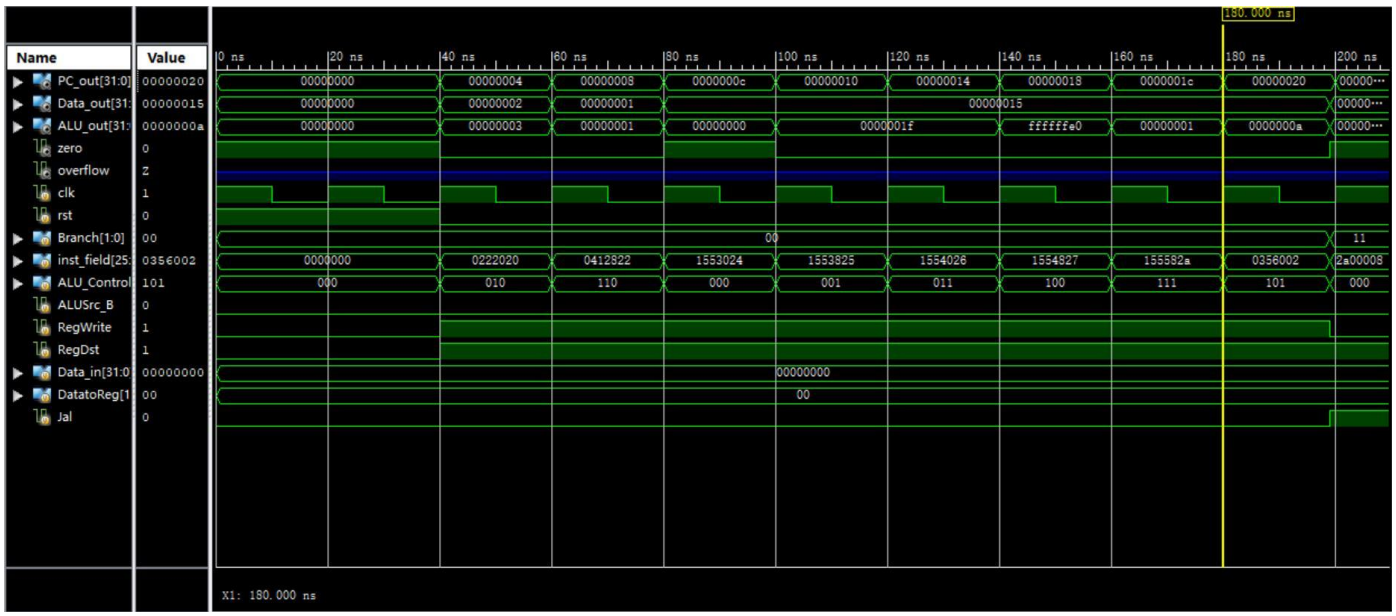
仿真代码:

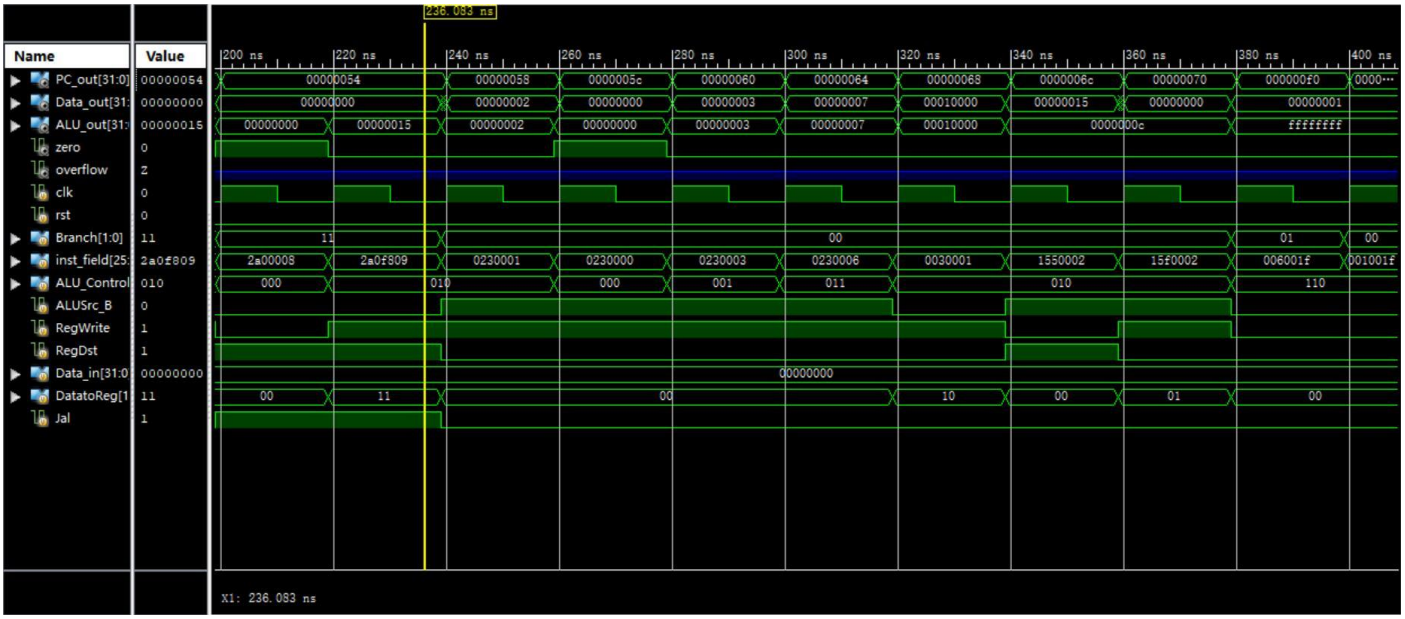
```
initial begin
        rst = 1;
        Branch = 0;
        inst_field = 0;
        ALU_Control = 0;
        ALUSrc_B = 0;
        RegWrite = 0;
        RegDst = 0;
        Data_in = 0;
        DatatoReg = 0;
        Jal = 0;
        #40;

        /**********R 型指令**********/

        `define   CPU_ctrl_signals   {RegDst,   ALU_Control,   ALUSrc_B,
DatatoReg[1:0], Jal, Branch[1:0], RegWrite}

        rst = 0;
        //add:寄存器 1 和 2 的值相加，存储到寄存器 4 中
        `CPU_ctrl_signals = 13'b1_010_0_00_0_00_1;
        inst_field = 26'b00001_00010_00100_00000_100000;
        #20;

        //sub:寄存器 2 和 1 的值相减，存储到寄存器 5 中
        `CPU_ctrl_signals = 13'b1_110_0_00_0_00_1;
        inst_field = 26'b00010_00001_00101_00000_100010;
        #20;
```

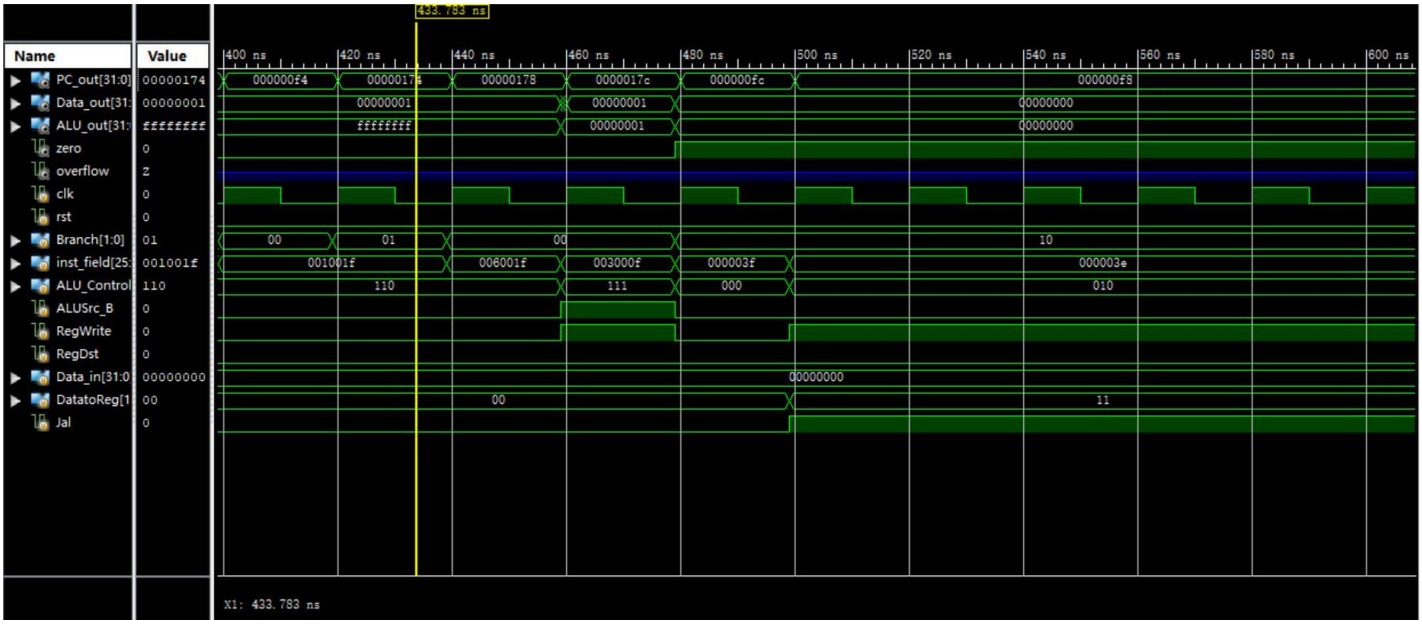
```
        //and:寄存器 10 和 21 的值按位与，存储到寄存器 6 中
        `CPU_ctrl_signals = 13'b1_000_0_00_0_00_1;
        inst_field = 26'b01010_10101_00110_00000_100100;
        #20;

        //or:寄存器 10 和 21 的值按位或，存储到寄存器 7 中
        `CPU_ctrl_signals = 13'b1_001_0_00_0_00_1;
        inst_field = 26'b01010_10101_00111_00000_100101;
        #20;

        //xor:寄存器 10 和 21 的值按位异或，存储到寄存器 8 中
        `CPU_ctrl_signals = 13'b1_011_0_00_0_00_1;
        inst_field = 26'b01010_10101_01000_00000_100110;
        #20;

        //nor:寄存器 10 和 21 的值按位或非，存储到寄存器 9 中
        `CPU_ctrl_signals = 13'b1_100_0_00_0_00_1;
        inst_field = 26'b01010_10101_01001_00000_100111;
        #20;

        //slt:判断寄存器 10 的值是否小于寄存器 21 的值，结果存储到寄存器 11 中
        `CPU_ctrl_signals = 13'b1_111_0_00_0_00_1;
        inst_field = 26'b01010_10101_01011_00000_101010;
        #20;

        //srl:寄存器 21 的值右移寄存器 1 的值（1 位），结果存储到寄存器 12 中
        `CPU_ctrl_signals = 13'b1_101_0_00_0_00_1;
        inst_field = 26'b00001_10101_01100_00000_000010;
        #19;

        //jr:无条件跳转到由寄存器 21 指定的指令
        `CPU_ctrl_signals = 13'b1_000_0_00_1_11_0;
        inst_field = 26'b10101_00000_00000_00000_001000;
        #20;

        //jalr:无条件跳转到由寄存器 21 指定的指令，并将下一条指令的地址保存到寄存器 31
中
        `CPU_ctrl_signals = 13'b1_010_0_11_1_11_1;
        inst_field = 26'b10101_00000_11111_00000_001001;
        #20;

        /***********I 型指令***********/

        //addi:寄存器 1 的值加上 1 存到寄存器 3 中去
        `CPU_ctrl_signals = 13'b0_010_1_00_0_00_1;
        inst_field = 26'b00001_00011_0000000000000001;
        #20;

        //andi:寄存器 1 的值与上 0 存到寄存器 3 中去
        `CPU_ctrl_signals = 13'b0_000_1_00_0_00_1;
        inst_field = 26'b00001_00011_0000000000000000;
        #20;

        //ori:寄存器 1 的值或上 3 存到寄存器 3 中去
        `CPU_ctrl_signals = 13'b0_001_1_00_0_00_1;
        inst_field = 26'b00001_00011_0000000000000011;
        #20;

        //xori:寄存器 1 的值异或上 6 存到寄存器 3 中去
        `CPU_ctrl_signals = 13'b0_011_1_00_0_00_1;
        inst_field = 26'b00001_00011_0000000000000110;
        #20;

        //lui:立即数 1 的低位值存入寄存器 3 的高位地址，将寄存器的低位值置为 0
        `CPU_ctrl_signals = 13'b0_010_0_10_0_00_1;
```

```
        inst_field = 26'b00000_00011_0000000000000001;
        #20;

        //sw:把寄存器 21 的值存储到内存地址为寄存器 10 的值加上 2 的内存中去
        `CPU_ctrl_signals = 13'b1_010_1_00_0_00_0;
        inst_field = 26'b01010_10101_0000000000000010;
        #20;

        //lw:把内存地址为寄存器 10 的值加上 2 的内存中存储的值加载到寄存器 31 中去
        `CPU_ctrl_signals = 13'b0_010_1_01_0_00_1;
        inst_field = 26'b01010_11111_0000000000000010;
        #20;

        //beq(equal):如果寄存器 0 的值和寄存器 6 的值相等，则跳转到地址
pc_4+000000000001111100
        `CPU_ctrl_signals = 13'b0_110_0_00_0_01_0;
        inst_field = 26'b00000_00110_0000000000011111;
        #20;

        //beq(n_equal):如果寄存器 0 的值和寄存器 1 的值不相等，则接着执行下一条指令
        `CPU_ctrl_signals = 13'b0_110_0_00_0_00_0;
        inst_field = 26'b00000_00001_0000000000011111;
        #20;

        //bne(n_equal):如果寄存器 0 的值和寄存器 1 的值不相等，则跳转到地址
pc_4+000000000001111100
        `CPU_ctrl_signals = 13'b0_110_0_00_0_01_0;
        inst_field = 26'b00000_00001_0000000000011111;
        #20;

        //bne(equal):如果寄存器 0 的值和寄存器 6 的值相等，则接着执行下一条指令
        `CPU_ctrl_signals = 13'b0_110_0_00_0_00_0;
        inst_field = 26'b00000_00110_0000000000011111;
        #20;

        //slti:判断寄存器 0 的值是否小于立即数 15，将结果存放到寄存器 3 中
        `CPU_ctrl_signals = 13'b0_111_1_00_0_00_1;
        inst_field = 26'b00000_00011_0000000000001111;
        #20;

        /***********J 型指令***********/

        //jump:跳转到地址 0000_0000_0000_0000_0000_0000_1111_1100
        `CPU_ctrl_signals = 13'b0_000_0_00_0_10_0;
        inst_field = 26'b00000000000000000000000111111;
        #20;

        //jal:跳转到地址 0000_0000_0000_0000_0000_0000_1111_1000，并将下一条指
令的地址保存到寄存器 ra 中
        `CPU_ctrl_signals = 13'b0_010_0_11_1_10_1;
        inst_field = 26'b00000000000000000000000111110;
        #20;

        end

        always begin
        clk=1;#10;
        clk=0;#10;
end
```

仿真结果图:


Name
Value
PC_out[31:0]
Data_out[31:
ALU_out[31:
zero
overflow
clk
rst
Branch[1:0]
inst_field[25:
ALU_Control
ALUSrc_B
RegWrite
RegDst
Data_in[31:0
DatatoReg[1
Jal
180.000 ns
X1: 180.000 ns


Name
Value
PC_out[31:0]
Data_out[31:
ALU_out[31:
zero
overflow
clk
rst
Branch[1:0]
inst_field[25:
ALU_Control
ALUSrc_B
RegWrite
RegDst
Data_in[31:0
DatatoReg[1
Jal
236.083 ns
X1: 236.083 ns


Name
Value
PC_out[31:0]
Data_out[31:
ALU_out[31:
zero
overflow
clk
rst
Branch[1:0]
inst_field[25:
ALU_Control
ALUSrc_B
RegWrite
RegDst
Data_in[31:0
DatatoReg[1
Jal
433.783 ns
X1: 433.783 ns

如图所示，仿真测试了 add， sub， and， or， xor， nor， slt， srl， jr， jalr; addi， andi， ori， xori， lui， lw， sw， beq， bne， slti ; J， Jal 这 22 条指令的效果，可参见仿真代码。仿真的思路在仿真代码的注释中有详细的描述。前 40ns rst 置为 1，进行模块的初始化（此处 Regs 的初始化相对于 Exp04 有调整，Exp04 中是将所有 32 个寄存器的值都初始化为 0，而此处将 0 号寄存器初始化为 0，1 号寄存器初始化为 1，以此类推）。

1. 初始化完成后，首先进行 add 指令的测试，inst_field = 26'b00001_00010_00100_00000_100000，含义是寄存器 1 和 2 的值相加，存储到寄存器 4 中，可以看到相应的 PC_out 的输出为 0x4，Data_out 的输出为 0x2（Data_out 输出 rt 寄存器存储的值），ALU_out 的输出为 0x3（1+2=3），结果与预期相符。
2. sub 指令的测试，inst_field = 26'b00010_00001_00101_00000_100010，含义是寄存器 2 和 1 的值相减，存储到寄存器 5 中，可以看到相应的 PC_out 的输出为 0x8，Data_out 的输出为 0x1（Data_out 输出 rt 寄存器存储的值），ALU_out 的输出为 0x1（2-1=1），结果与预期相符。
3. and 指令的测试，inst_field = 26'b01010_10101_00110_00000_100100，含义是寄存器 10 和 21 的值按位与，存储到寄存器 6 中，可以看到相应的 PC_out 的输出为 0xc，Data_out 的输出为 0x15（Data_out 输出 rt 寄存器存储的值），ALU_out 的输出为 0x0（01010&10101=0），结果与预期相符。
4. or 指令的测试，inst_field = 26'b01010_10101_00111_00000_100101，含义是寄存器 10 和 21 的值按位或，存储到寄存器 7 中，可以看到相应

的 PC_out 的输出为 0x10, Data_out 的输出为 0x15 (Data_out 输出 rt 寄存器存储的值), ALU_out 的输出为 0x1f (01010|10101=11111), 结果与预期相符。

5. xor 指令的测试, inst_field = 26'b01010_10101_01000_00000_100110, 含义是寄存器 10 和 21 的值按位异或, 存储到寄存器 8 中, 可以看到相应的 PC_out 的输出为 0x14, Data_out 的输出为 0x15 (Data_out 输出 rt 寄存器存储的值), ALU_out 的输出为 0x1f (01010 xor 10101=11111), 结果与预期相符。
6. nor 指令的测试, inst_field = 26'b01010_10101_01001_00000_100111, 含义是寄存器 10 和 21 的值按位或非, 存储到寄存器 9 中, 可以看到相应的 PC_out 的输出为 0x18, Data_out 的输出为 0x15 (Data_out 输出 rt 寄存器存储的值), ALU_out 的输出为 0xfffffffe0 (0x0000000a nor 0x00000015=0xfffffffe0), 结果与预期相符。
7. slt 指令的测试, inst_field = 26'b01010_10101_01011_00000_101010, 含义是判断寄存器 10 的值是否小于寄存器 21 的值, 结果存储到寄存器 11 中, 可以看到相应的 PC_out 的输出为 0x1c, Data_out 的输出为 0x15 (Data_out 输出 rt 寄存器存储的值), ALU_out 的输出为 0x1 (10<21 成立), 结果与预期相符。
8. srl 指令的测试, inst_field = 26'b00001_10101_01100_00000_000010, 含义是寄存器 21 的值右移寄存器 1 的值 (1 位), 结果存储到寄存器 12 中, 可以看到相应的 PC_out 的输出为 0x20, Data_out 的输出为 0x15 (Data_out 输出 rt 寄存器存储的值), ALU_out 的输出为 0xa

(10101>>1=1010=0xa)，结果与预期相符。

9. jr 指令的测试，inst_field = 26'b10101_00000_00000_00000_001000，含义是无条件跳转到由寄存器 21 指定的指令，可以看到相应的 PC_out 的输出为 0x54(10101<<2=1010100=0x54)，Data_out 的输出为 0x0，ALU_out 的输出为 0x0，结果与预期相符。
10. jalr 指令的测试，inst_field = 26'b10101_00000_11111_00000_001001，含义是无条件跳转到由寄存器 21 指定的指令，并将下一条指令的地址保存到寄存器 31 中，可以看到相应的 PC_out 的输出为 0x54(10101<<2=1010100=0x54)，Data_out 的输出为 0x0，ALU_out 的输出为 0x15，结果与预期相符。
11. addi 指令的测试，inst_field = 26'b00001_00011_0000000000000001，含义是寄存器 1 的值加上 1 存到寄存器 3 中去，可以看到相应的 PC_out 的输出为 0x58，Data_out 的输出为 0x2，ALU_out 的输出为 0x2(1+1=2)，结果与预期相符。
12. andi 指令的测试，inst_field = 26'b00001_00011_0000000000000000，含义是寄存器 1 的值与上 0 存到寄存器 3 中去，可以看到相应的 PC_out 的输出为 0x5c，Data_out 的输出为 0x0，ALU_out 的输出为 0x0(1&0=0)，结果与预期相符。
13. ori 指令的测试，inst_field = 26'b00001_00011_0000000000000011，含义是寄存器 1 的值或上 3 存到寄存器 3 中去，可以看到相应的 PC_out 的输出为 0x60，Data_out 的输出为 0x3，ALU_out 的输出为 0x3(01&11=11=3)，结果与预期相符。

14. xori 指令的测试，inst_field = 26'b00001_00011_0000000000000110，含义是寄存器 1 的值异或上 6 存到寄存器 3 中去，可以看到相应的 PC_out 的输出为 0x64，Data_out 的输出为 0x7，ALU_out 的输出为 0x7(001xor110=111=7)，结果与预期相符。
15. lui 指令的测试，inst_field = 26'b00000_00011_0000000000000001，含义是立即数 1 的低位值存入寄存器 3 的高位地址，将寄存器的低位值置为 0，可以看到相应的 PC_out 的输出为 0x68，Data_out 的输出为 0x10000，ALU_out 的输出为 0x10000(1 放到高位，低位置 0 也就是 0x00010000)，结果与预期相符。
16. sw 指令的测试，inst_field = 26'b01010_10101_0000000000000010，含义是把寄存器 21 的值存储到内存地址为寄存器 10 的值加上 2 的内存中去，可以看到相应的 PC_out 的输出为 0x6c，Data_out 的输出为 0x15，ALU_out 的输出为 0xc(0xa+0x2=0xc)，结果与预期相符。
17. lw 指令的测试，inst_field = 26'b01010_11111_0000000000000010，含义是把内存地址为寄存器 10 的值加上 2 的内存中存储的值加载到寄存器 31 中去，可以看到相应的 PC_out 的输出为 0x70，Data_out 的输出为 0x0(Data_in 的值为 0)，ALU_out 的输出为 0xc(0xa+0x2=0xc)，结果与预期相符。
18. beq 指令的测试，inst_field = 26'b00000_00110_0000000000011111，含义是如果寄存器 0 的值和寄存器 6 的值相等，则跳转到地址 pc_4+000000000001111100，可以看到相应的 PC_out 的输出为 0xf0 (0x70+0x4+0x7c=0xf0)，Data_out 的输出为 0x1，ALU_out 的输出为

0xffffffff，结果与预期相符。

19. bne 指令的测试，inst_field = 26'b00000_00001_0000000000011111，含义是如果寄存器 0 的值和寄存器 1 的值不相等，则跳转到地址 pc_4+000000000001111100，可以看到相应的 PC_out 的输出为 0x174 (0xf4+0x4+0x7c=0x174)，Data_out 的输出为 0x1，ALU_out 的输出为 0xffffffff，结果与预期相符。

20. slti 指令的测试，inst_field = 26'b00000_00011_0000000000001111，含义是判断寄存器 0 的值是否小于立即数 15，将结果存放到寄存器 3 中，可以看到相应的 PC_out 的输出为 0x17c，Data_out 的输出为 0x1，ALU_out 的输出为 0x1(0<15 成立)，结果与预期相符。

21. jump 指令的测试，inst_field = 26'b00000000000000000000111111，含义是跳转到地址 0xfc，可以看到相应的 PC_out 的输出为 0xfc，Data_out 的输出为 0x0，ALU_out 的输出为 0x0，结果与预期相符。

22. jal 指令的测试，inst_field = 26'b00000000000000000000111110，含义是跳转到地址 0xf8，并将下一条指令的地址保存到寄存器 ra 中，可以看到相应的 PC_out 的输出为 0xf8，Data_out 的输出为 0x0，ALU_out 的输出为 0x0，结果与预期相符。

综上所述，成功实现了 Data_Path_more 模块的设计。

3. 指令拓展的单周期 CPU 下板验证结果与分析

RAM_B 的初始化文件为 D_mem.coe，内容如下：

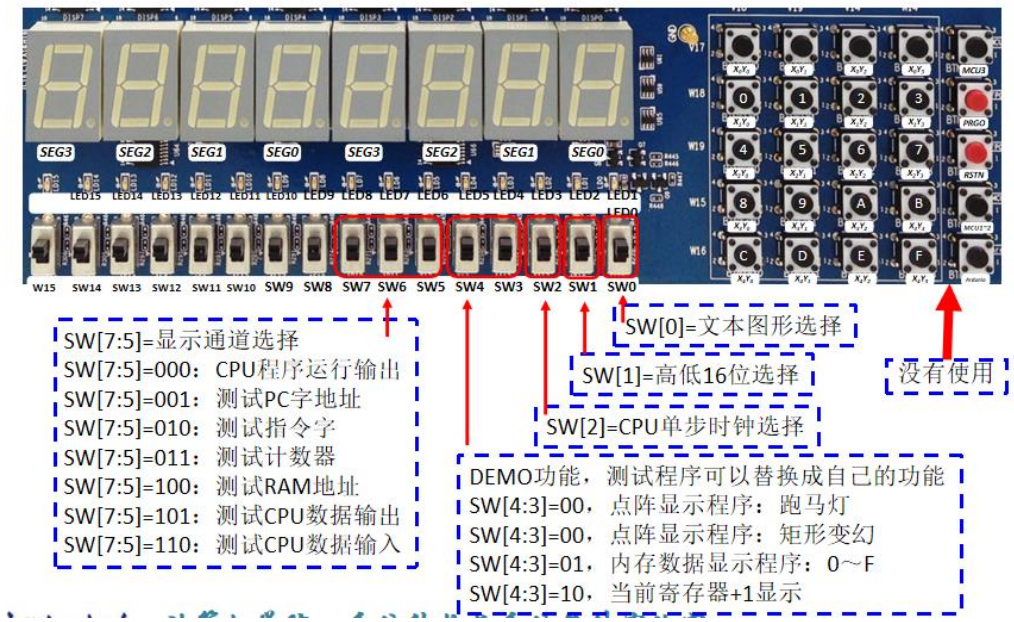
```
memory_initialization_radix=16;
memory_initialization_vector=
f0000000, 000002AB, 80000000, 0000003F, 00000001, FFF70000, 0000FFFF,
80000000, 00000000, 11111111,
22222222, 33333333, 44444444, 55555555, 66666666, 77777777, 88888888,
```

```
99999999, aaaaaaaa, bbbbbbbb,
cccccccc, dddddddd, eeeeeeee, ffffffff, 557EF7E0, D7BDFBD9, D7DBFDB9,
DFCFFCFB, DFCFBFFF, F7F3DFFF,
FFFFDF3D, FFFF9DB9, FFFFBCFB, DFCFFCFB, DFCFBFFF, D7DB9FFF, D7DBFDB9,
D7BDFBD9, FFFF07E0, 007E0FFF,
03bdf020, 03def820, 08002300;
```

ROM_D 的初始化文件为 I9_mem.coe，内容如下：

```
;cpu_IO_test1.asm
memory_initialization_radix=16;
memory_initialization_vector=
08000008, 00000020, 00000020, 00000020, 00000020, 00000020, 00000020,
00000020, 00000827, 00211820,
00631820, 00631820, 00631820, 00631820, 00631820, 0060a027, 00631820,
00631820, 00631820, 00631820,
00631820, 00631820, 00631820, 00631820, 00631820, 00631820, 00631820,
00631820, 00631820, 00631820,
00631820, 00631820, 00631820, 00631820, 00631820, 00631820, 00633020,
00c61820, 00632020, 00846820,
01ad4020, 0001102a, 00427020, 01ce7020, 00005027, 014a5020, ac660004,
8c650000, 00a52820, 00a52820,
ac650000, 01224820, ac890000, 8c0d0014, 8c650000, 00a52820, 00a52820,
ac650000, 8c650000, 00a85824,
01a26820, 11a00017, 8c650000, 01ce9020, 0252b020, 02569020, 00b25824,
11600005, 1172000a, 01ce9020,
1172000b, ac890000, 08000036, 11410001, 0800004d, 00005027, 014a5020,
ac8a0000, 08000036, 8e290060,
ac890000, 08000036, 8e290020, ac890000, 08000036, 8c0d0014, 014a5020,
01425025, 022e8820, 02348824,
01224820, 11210001, 0800005f, 000e4820, 01224820, 8c650000, 00a55820,
016b5820, ac6b0000, ac660004,
8c650000, 00a85824, 0800003e;
```

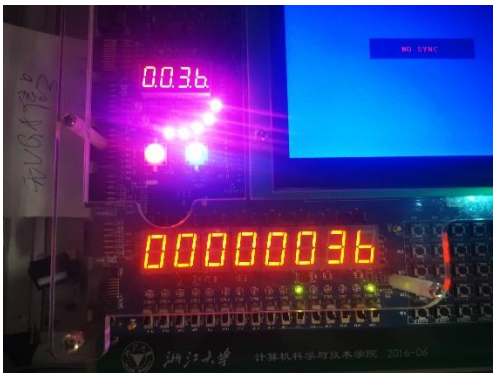
验证思路如下图所示：



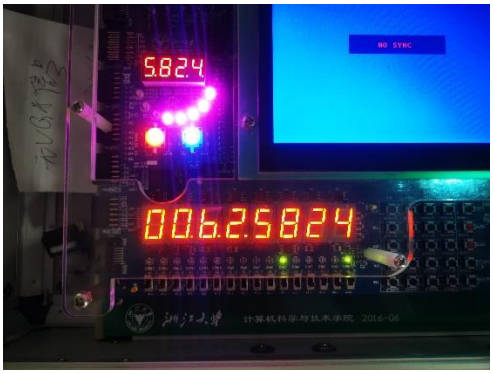
生成的 bit 文件下板验证的结果和说明如下:

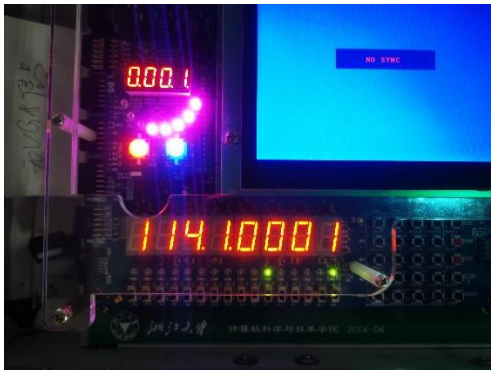
1. 拨动开关，使得 SW[0]=1，进入文本显示模式。SW[7:5]=000，进入显示通道 0，八位数码管将显示 CPU 运行程序后的输出。再让 SW[4:3]=01，这时对应的就是内存数据显示程序，实验中可以观察到八位数码管的显示从 00000000 到 FFFFFFFF 循环变换，如下所示为部分实验结果照片。

2. 继续保持 SW[0]=1，为文本显示模式。拨动开关调整 SW[7:5]=001，选择显示通道 1，八位数码管将显示 PC 地址的数据。此时若保持 SW2 为 0 则为 CPU 时钟，速度较快，不便于观察显示结果。可以拨动开关调整 SW2 位 1，切换到单步时钟，这样就可以较好地观察到实验结果。实验中可以观察到八位数码管的显示从 00000040 到 00000043 依次递增 1 显示，跳到 00000049，再从 00000049 到 0000004A 依次递增 1 显示，跳到 0000004D，接着显示 0000004E，之后跳到 00000036，再从 00000036 到 0000003F 依次递增 1 显示。整个过程为 1 个循环，数码管不断循环显示上述结果。如下所示为部分实验结果照片。

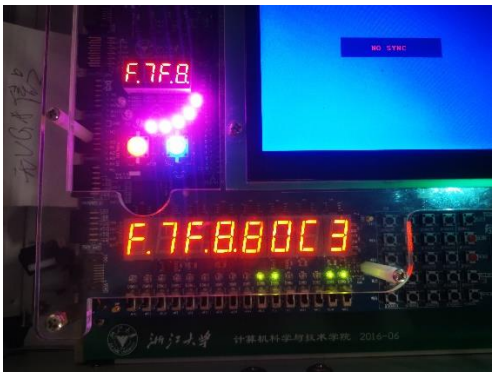


3. 继续保持 SW[0]=1，为文本显示模式。拨动开关调整 SW[7:5]=010，选择显示通道 2，八位数码管将显示 Counter 的输出。此时若保持 SW2 为 0 则为 CPU 时钟，速度较快，不便于观察显示结果。可以拨动开关调整 SW2 位 1，切换到单步时钟，这样就可以较好地观察到实验结果。如下所示为部分实验结果照片。

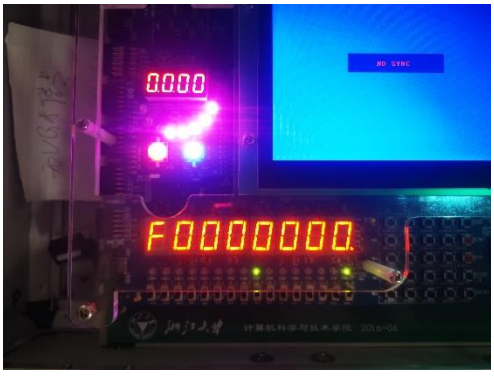


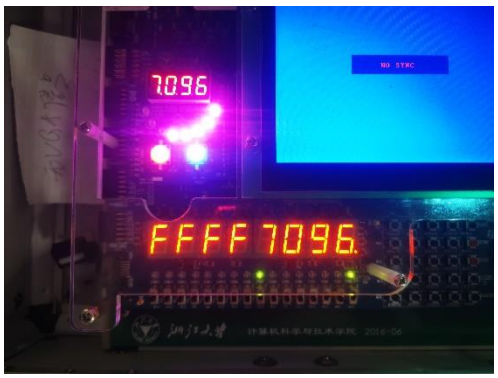


4. 继续保持 SW[0]=1，为文本显示模式。拨动开关调整 SW[7:5]=011，选择显示通道 3，八位数码管将显示 Inst_in 的值。若是拨动开关调整 SW1 为 1，则 Arduino 板上的四位七段数码管将显示高 16 的内容，如下所示为部分实验结果照片。

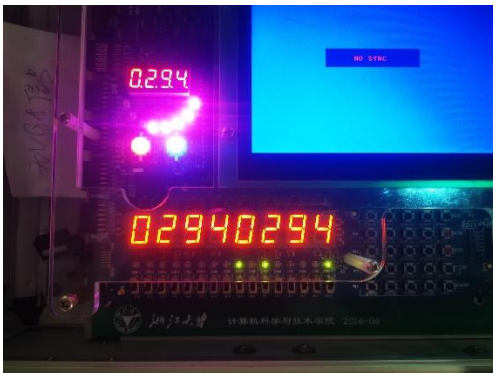


5. 继续保持 SW[0]=1，为文本显示模式。拨动开关调整 SW[7:5]=100，选择显示通道 4，八位数码管将显示 Addr_out 的值。此时若保持 SW2 为 0 则为 CPU 时钟，速度较快，不便于观察显示结果。可以拨动开关调整 SW2 位 1，切换到单步时钟，这样就可以较好地观察到实验结果。如下所示为部分实验结果照片。

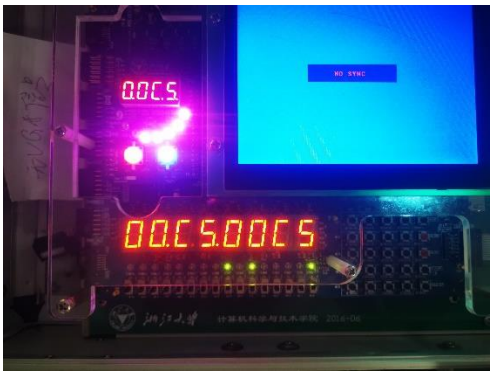




6. 继续保持 SW[0]=1，为文本显示模式。拨动开关调整 SW[7:5]=101，选择显示通道 5，八位数码管将显示 Cpu_data2bus 的值。此时若保持 SW2 为 0 则为 CPU 时钟，速度较快，不便于观察显示结果。可以拨动开关调整 SW2 位 1，切换到单步时钟，这样就可以较好地观察到实验结果。如下所示为部分实验结果照片。

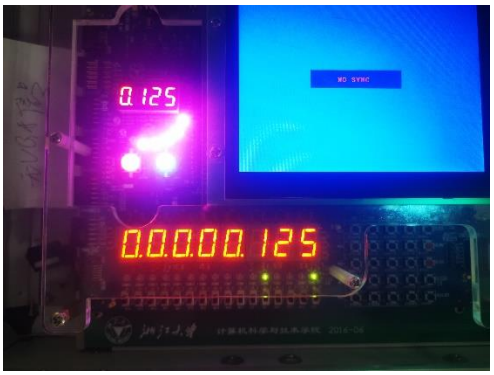


7. 继续保持 SW[0]=1，为文本显示模式。拨动开关调整 SW[7:5]=110，选择显示通道 6，八位数码管将显示 Cpu_data4bus 的值。此时若保持 SW2 为 0 则为 CPU 时钟，速度较快，不便于观察显示结果。可以拨动开关调整 SW2 位 1，切换到单步时钟，这样就可以较好地观察到实验结果。如下所示为部分实验结果照片。

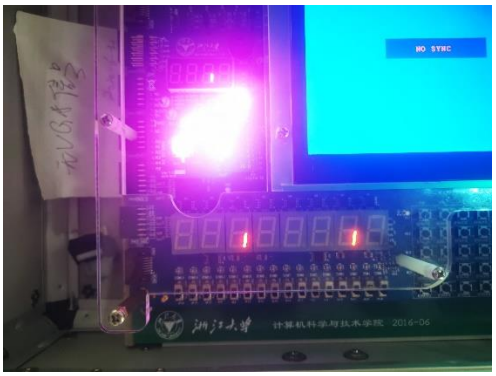


8. 继续保持 SW[0]=1，为文本显示模式。拨动开关调整 SW[7:5]=111，选择显示通道 7，八位数码管将显示 PC_out 的值。此时若保持 SW2 为 0 则为 CPU 时钟，速度较快，不便于观察显示结果。可以拨动开关调整 SW2 位 1，切换到单步时钟，这样就可以较好地观察到实验结果。如下所示为部分实验结果照片。

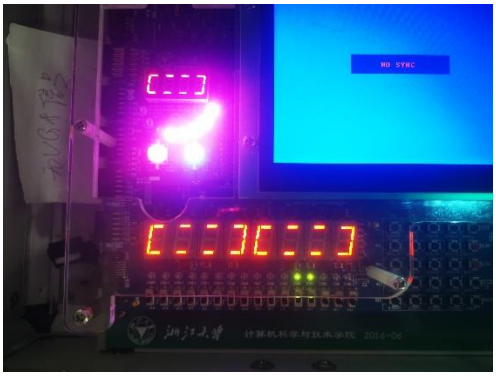
9. 继续保持 SW[0]=1，为文本显示模式。拨动开关调整 SW[7:5]=000，选择显示通道 0，再调整 SW[4:3]=10，八位数码管将显示当前寄存器不断递增 1 的值。此时若保持 SW2 为 0 则为 CPU 时钟，速度较快，不便于观察显示结果。可以拨动开关调整 SW2 位 1，切换到单步时钟，这样就可以较好地观察到实验结果。如下所示为部分实验结果照片。



10. 拨动开关调整 SW[0]=0，为图像显示模式。调整 SW[7:5]=000，选择显示通道 0，即 CPU 运行程序的输出。再调整 SW[4:3]=00，八位数码管将呈现跑马灯的效果。七段数码管上的图案从最顶端开始依次从右向左移动，之后转到下一行，不断循环显示。此时若拨动开关使得 SW2 变为 1，则转为单步时钟，图案的变化将变得非常缓慢，如下所示为部分实验结果照片。

11. 保持 SW[0]=0，为图像显示模式保持 SW[7:5]=000，选择显示通道 0，即 CPU 运行程序的输出。再调整 SW[4:3]=11，八位数码管将呈现矩阵变换的效果。七段数码管上的显示的矩阵不断由小变大，再由大变小，不断循环显示。此时若拨动开关使得 SW2 变为 1，则转为单步时钟，图案的变化将变得非常缓慢，如下所示为部分实验结果照片。

通过下板验证的结果可以看出，本次指令集拓展后的单周期 CPU 设计成功实现，可以正常执行相应的功能。

三、讨论、心得

仿真测试阶段心得：本次实验是计算机组成课程的第七次实验，是单周期 CPU

设计的第三个部分，指令集扩展。本次实验需要对指令进行拓展，因此需要重新设计数据通路和控制器模块。由于实验六中对于控制器的设计我选择了采用Verilog代码结构描述设计的方法实现，这样在实验七中我进行指令拓展也方便了许多。本次实验的重难点就在于对数据通路和控制器重新设计的过程中，首先需要根据需要拓展的指令提前考虑好控制信号是否需要进行拓展，就比如Branch需要从1位拓展到2位，需要提前考虑清楚之后才能进行设计。另外就是对每条指令的含义，每个字段的含义都必须非常清楚，另外对每条指令执行时各个控制信号的值也应该很清楚。对此我采用了画表格的方法辅助我进行设计，将每条指令对应的机器码，以及各个控制信号的值放在同一张表格中，这样能够更直观的进行设计。尽管做了这么多准备，我在设计过程中还是出现了差错，误将jalr指令的branch控制信号设置为了10，导致对数据通路进行仿真时PC的值出现了问题，将jalr指令的branch控制信号修改为11后解决了这个问题，这说明细心和耐心都是很重要的。本次实验是单周期CPU设计中的最后一个实验，而单周期CPU由于周期的时间是根据执行时间最长的指令设计的，所以会有一定的时间浪费，而接下来将要进行的多周期CPU的设计便会针对这个问题进行优化，期待接下来的实验。

下板验证阶段心得：由于在线下实验开始之前，老师就提到了在下板验证的过程中遇到问题是十分正常的，所以我也算是做好了心里准备。毕竟仿真的结果是软件范畴，而下板验证还有更多硬件上的问题要考虑清楚才行。果然第一次下板就遇到了问题：我的数码管一直显示为AA5555AA（如下图所示），让我有些摸不着头脑。

在求助老师后得知，数码管显示为 AA5555AA 算是最常见的一类错误了，但是造成这个错误的原因很多，直接原因是 SCPU 没有正常工作而是只显示了初始值。于是我首先针对顶层原理图中的 SCPU 模块进行了检查，发现可能是 SCPU 的时钟信号设置有问题，更改之后便能正常显示跑马灯和矩阵变换了。但之后又发现 Arduino 小板上的显示有问题，只有最后一位有显示。推测是之前设计的 Seg7_Dev 模块有问题，将其替换为 ngc 核实现后发现能够正常显示，这样就确定了是 Seg7_Dev 模块的问题。经过仔细排查之后，发现问题出在 ScanSync 模块上，我在扫描时的 AN 信号全部都赋了 4'b1110，这样的话肯定就只有最后一位显示了。修改之后，终于能够正常显示了。

下板验证的过程中，最重要的就是耐心和细心了，因为任何一点微小的错误都会导致结果的不正确。另外我体会到利用 Verilog 代码进行设计相对于原理图设计来说其实更加清晰，不容易出错，就算是出错了也比原理图更好排查一些。此外，通过用依次用 ngc 核来替换自己设计的模块也是进行错误排查时非常有用的方式。

思考题:

1. 指令扩展时控制器用二级译码设计存在什么问题?

指令拓展时相对于未拓展之前的设计来说，如果使用二级译码设计将会

变得十分繁琐，且再拓展性比较差。而使用 verilog 代码来进行设计的话就会方便后期的调试和修改，想要再拓展一些其他指令也方便的多。

2. 设计 bne 指令需要增加控制信号吗?

不需要，bne 指令与 beq 指令的设计逻辑其实基本是完全相同的，只需要在 beq 对 zero 信号的处理基础上取反即可。

3. 设计 andi 指令需要增加新的数据通道吗?

不需要。andi 指令只需要立即数也可以进入 ALU 参与运算，而这条数据通道在未拓展指令前的 Datapath 中已经存在，沿用之前的即可。

四、个人照片