实验2 单周期CPU IO模块设计

518030910408 周韧

一. 实验环境

本次实验在Quartus II 13.0(64-bit)开发软件中进行。

二. 实验目的

1. 在理解计算机 5 大组成部分的协调工作原理，理解存储程序自动执行的原理和掌握运算器、存储器、控制器的设计和实现原理基础上，掌握 I/O 端口的设计方法，理解 I/O 地址空间的设计方法。
2. 通过设计 I/O 端口与外部设备进行信息交互。
3. 通过设计并实现新的自定义指令拓展 CPU 功能，深入理解 CPU 对指令的译码、执行原理和实现方式。（选做）

三. 设计思路分析

1.分析顶层结构

本次实验的实验指导中给出了整个实验项目的顶层结构设计图，但是顶层文件需要我们自己来实现，我们需要在顶层文件的结构框架下，通过结合上次单周期CPU项目的部分代码，实现IO端口模块的设计。

首先我们建立 sc_io_cpu.v 的顶层文件，在其中需要例化 clock_and_mem_clock 模块、两个 in_port 模块、sc_computer_main 模块、三个 out_port 模块。sc_io_computer.v 顶层文件的输入信号中，有 SW0 - SW9 十个一位二进制数，我们将其分成两组并在in_port模块中进行组合扩展，将拼接后的信号作为 sc_computer_main模块的输入信号，得到三个out_port端口输出信号，将这三个信号输入out_port_seg，经过 sevenreg 七段译码器处理之后，输出我们最终六个的HEX信号。

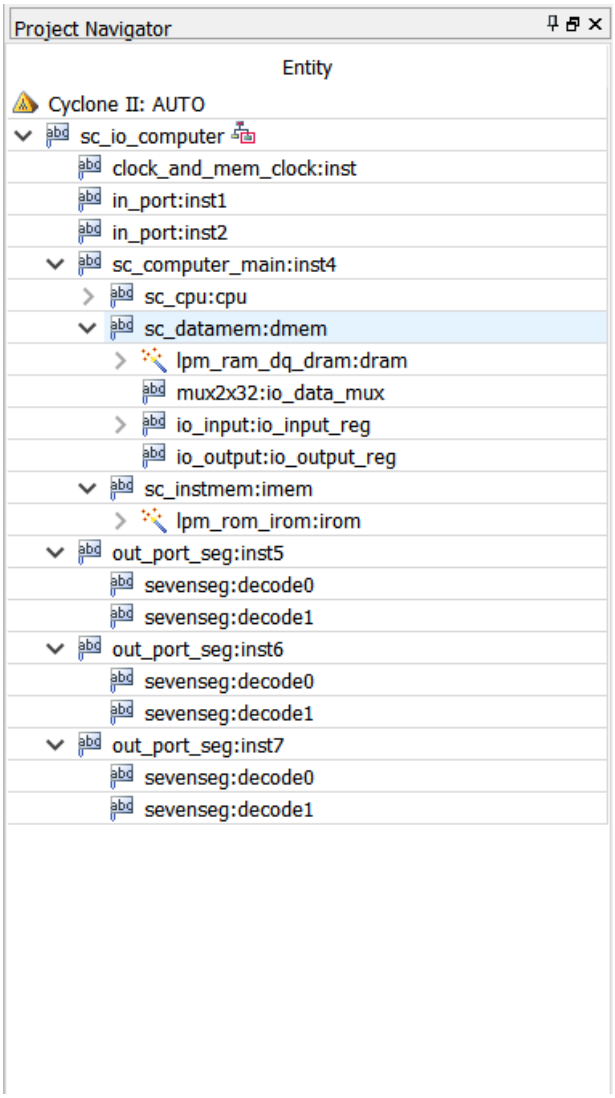
同时在顶层模块sc_io_cpu中，我们需要对输入的时钟信号进行分频处理，得到频率为之前的二分之一的新的mem_clk时钟信号, 该信号和原先的clk一起作为控制sc_instmem模块和sc_datamem模块的时钟信号。

至此我们分析完了整个顶层文件需要实现的内容，展示顶层文件下的代码：

```
module sc_io_computer(SW0, SW1, SW2, SW3, SW4, SW5, SW6, SW7, SW8, SW9,
                              HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, clk, reset);

    input SW0, SW1, SW2, SW3, SW4, SW5, SW6, SW7, SW8, SW9;
    input clk, reset;
    output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    wire [31:0] in_port0, in_port1;
    wire [31:0] pc, aluout, memout, ins3;
```

```
    wire imem_clk, dmem_clk;
    wire [31:0] out_port0, out_port1, out_port2;
    wire [31:0] mem_dataout;
    wire [31:0] io_read_data;
    wire clock_out;

    clock_and_mem_clock inst(reset, clk, clock_out);

    in_port inst1(SW0, SW1, SW2, SW3, SW4, in_port1);
    in_port inst2(SW5, SW6, SW7, SW8, SW9, in_port0);

    sc_computer_main inst4(reset, clock_out, clk, pc, inst3, aluout, memout,
imem_clk, dmem_clk, out_port0, out_port1, out_port2, in_port0, in_port1,
mem_dataout, io_read_data);

    out_port_seg inst5(out_port0, HEX0, HEX1);
    out_port_seg inst6(out_port1, HEX2, HEX3);
    out_port_seg inst7(out_port2, HEX4, HEX5);

endmodule
```

顶层模块设计图如下所示



2. 核心模块设计

2.1 clock_and_mem_clock.v 分频模块

将复位信号 reset 和时钟信号 clk 作为输入，分频后的信号clock_out作为输出，在 clk 信号上升沿对clock_out 取反，从而得到频率为 clk 二分之一的 clock_out 信号。

```
module clock_and_mem_clock(reset, clk, clock_out);
    input reset, clk;
    output clock_out;
    reg clock_out;

    initial
    begin
        clock_out = 0;
    end

    always@(posedge clk)
    begin
        if(~reset)
            clock_out <= 0;
        clock_out <= ~clock_out;
    end
endmodule
```

2.2 in_port.v 输入端口模块

这里输入五个二进制信号，输出拼接后的信号，输出信号前27位扩展为0，一共为32位。输出的32位信号将作为sc_computer_main.v的输出信号。

```
module in_port(sw0, sw1, sw2, sw3, sw4, res);
    input sw0, sw1, sw2, sw3, sw4;
    output wire [31:0] res;
    assign res = {{27{1'b0}}, sw4, sw3, sw2, sw1, sw0};
endmodule
```

2.3 sc_computer_main.v 主模块

sc_computer_main.v模块是整个项目的核心模块，在结构上与实验一中的顶层模块sc_computer.v基本相同，不同的地方是多出了一些输入输出端口信号等。主要需要修改的地方是例化的sc_datamem模块对应的文件。

```
// sc_computer_main.v
module sc_computer_main(resetn, clock, mem_clk, pc, inst, aluout, memout,
imem_clk, dmem_clk, out_port0, out_port1, out_port2, in_port0, in_port1,
mem_dataout, io_read_data);

   input resetn, clock, mem_clk;
    input [31:0] in_port0, in_port1;
   output [31:0] pc, inst, aluout, memout;
   output imem_clk, dmem_clk;
    output [31:0] out_port0, out_port1, out_port2;
    output [31:0] mem_dataout, io_read_data;
   wire [31:0] data;
   wire wmem; // all these "wire"s are used to connect or interface the
cpu,dmem,imem and so on.
```

```
    sc_cpu cpu (clock, resetn, inst, memout, pc, wmem, aluout, data);          //
CPU module.
    sc_instmem imem (pc, inst, clock, mem_clk, imem_clk);                      //
instruction memory.
     sc_datamem dmem (aluout, data, memout, wmem, clock, mem_clk, dmem_clk,
resetn, out_port0, out_port1, out_port2, in_port0, in_port1, mem_dataout,
io_read_data); // data memory.

endmodule
```

sc_datamem.v模块与实验一中有一定区别，这部分代码已经在实验指导详细地给出，clock 和降频后的 mem_clock共同决定了data memory的时钟周期dmem_clock，dmem_clock 与 mem_clock 周期相等。addr[7]决定了是向data memory中写入数据还是向IO中写入数据。

```
// sc_datamem.v
module sc_datamem(addr, datain, dataout, we, clock, mem_clk, dmem_clk, resetn,
out_port0, out_port1, out_port2, in_port0, in_port1, mem_dataout, io_read_data);

    input [31:0] addr;
    input [31:0] datain;
    input [31:0] in_port0, in_port1;
    input we, clock, mem_clk, resetn;

    output [31:0] dataout;
    output dmem_clk;
    output [31:0] out_port0, out_port1, out_port2;
     output [31:0] mem_dataout, io_read_data;

    wire dmem_clk;
    wire write_enable;
     wire [31:0] dataout;
     wire [31:0] mem_dataout;
     wire write_data_enable;
     wire write_io_enable;

    assign write_enable = we & (~clock);
     assign dmem_clk = mem_clk & (~clock);
     assign write_data_enable = write_enable & (~addr[7]);
    assign write_io_enable = write_enable & addr[7];
     mux2x32 io_data_mux(mem_dataout, io_read_data, addr[7], dataout);
    lpm_ram_dq_dram dram(addr[6:2], dmem_clk, datain, write_data_enable,
mem_dataout);
     io_output io_output_reg(addr, datain, write_io_enable, dmem_clk, out_port0,
out_port1, out_port2, resetn);
     io_input io_input_reg(addr, dmem_clk, io_read_data, in_port0, in_port1);
endmodule
```

在sc_datamem.v中，我们还需要例化io_input.v模块和io_output.v模块。这两部分代码的作用是建立IO和CPU之间的数据通路，io_put.v模块中建立将数据从IO输入到CPU中的数据通路，io_output.v模块将建立数据从CPU输出到IO中数据通路。这部分主要的代码已经在实验指导中给出，这里不再过多解释，我在这里修改了io_output.v模块，向其中增加了resetn复位信号。

```
// io_output.v
module io_output(addr, datain, write_io_enable, io_clk, out_port0, out_port1,
out_port2, resetn);
```

```
    input [31:0] addr, datain;
    input write_io_enable, io_clk;
    input resetn;
    output [31:0] out_port0, out_port1, out_port2;

    reg [31:0] out_port0;
    reg [31:0] out_port1;
    reg [31:0] out_port2;

    always@(posedge io_clk or negedge resetn)
    begin
        if(resetn == 0)
        begin
            out_port0 <= 0;
            out_port1 <= 0;
            out_port2 <= 0;
        end
        else
        begin
            if(write_io_enable == 1)
            case(addr[7:2]) // 根据alu的值选择不同输出的端口
                6'b100000: out_port0 <= datain;//80h
                6'b100001: out_port1 <= datain;//84h
                6'b100010: out_port2 <= datain;//88h
            endcase
        end
    end
endmodule
```

```
// io_input.v
module io_input(addr, io_clk, io_read_data, in_port0, in_port1);
    input[31:0]addr;
    input io_clk;
    input[31:0]in_port0,in_port1;
    output[31:0]io_read_data;

    reg[31:0]in_reg0,in_reg1;

    io_input_mux io_input_mux2x32(in_reg0,in_reg1,addr[7:2],io_read_data);

    always@(posedge io_clk)
        begin
            in_reg0<=in_port0;//输入端口在io_clk上升沿时进行数据锁存
            in_reg1<=in_port1;
        end
endmodule

module io_input_mux(a0,a1,sel_addr,y);
    input [31:0] a0,a1;
    input [5:0] sel_addr;
    output [31:0] y;
    reg [31:0] y;
    always@*
        case(sel_addr)
            6'b100000:y=a0;
            6'b100001:y=a1;
            default:y=32'h0;
```

```
        endcase
endmodule
```

2.4 out_port_seg.v 输出端口模块

最后一部分是out_port_seg.v模块，将sc_computer模块中得到的三个输出信号拆分成六个四位信号并将其转为10进制，并在七段译码器中译码得到六个七位最终输出信号。

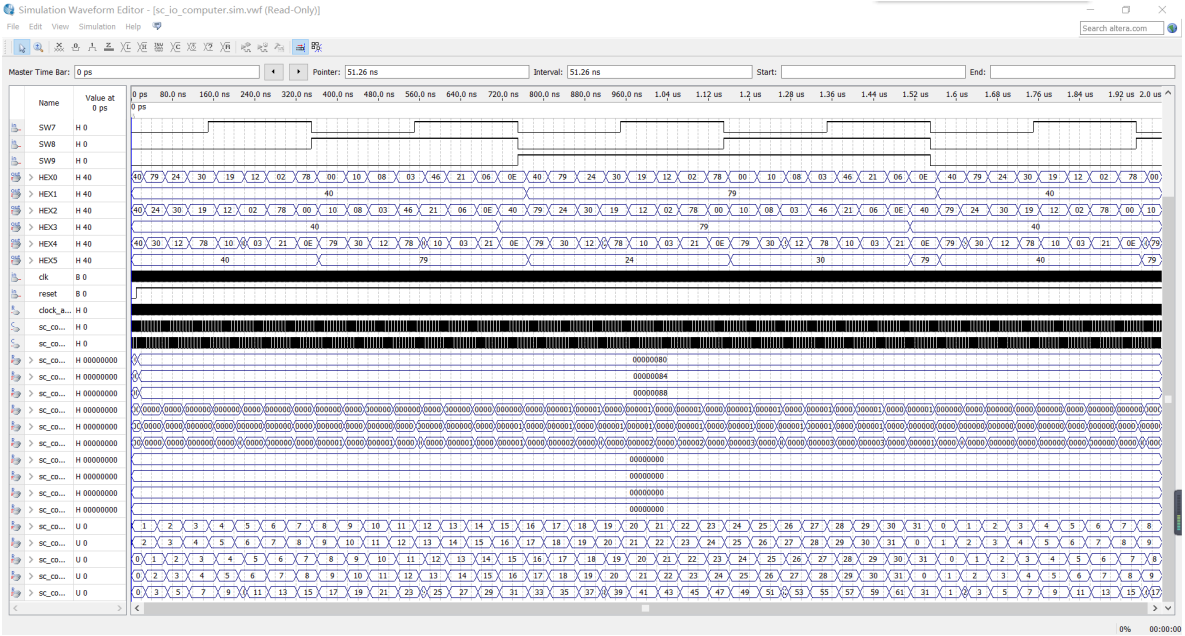
```
// out_port_seg.v
module out_port_seg(outport, hex_data0, hex_data1);
    input wire [31:0] outport;
    output wire [6:0] hex_data0, hex_data1;
    wire [3:0] out0, out1;
    assign out0 = outport % 10;
    assign out1 = outport / 10;
    sevenseg decode0(out0, hex_data0);
    sevenseg decode1(out1, hex_data1);
endmodule
```

```
// sevenreg.v
module sevenseg(in, sevenseg);
    input [3:0] in;
    output [6:0] sevenseg;
    reg [6:0] sevenseg;
    initial
    begin
        sevenseg = 0;
    end
    always @(*)
    begin
        case(in)
            4'h0: sevenseg[6:0] = 7'b1000000;
            4'h1: sevenseg[6:0] = 7'b1111001;
            4'h2: sevenseg[6:0] = 7'b0100100;
            4'h3: sevenseg[6:0] = 7'b0110000;
            4'h4: sevenseg[6:0] = 7'b0011001;
            4'h5: sevenseg[6:0] = 7'b0010010;
            4'h6: sevenseg[6:0] = 7'b0000010;
            4'h7: sevenseg[6:0] = 7'b1111000;
            4'h8: sevenseg[6:0] = 7'b0000000;
            4'h9: sevenseg[6:0] = 7'b0010000;
            4'hA: sevenseg[6:0] = 7'b0001000;
            4'hB: sevenseg[6:0] = 7'b0000011;
            4'hC: sevenseg[6:0] = 7'b1000110;
            4'hD: sevenseg[6:0] = 7'b0100001;
            4'hE: sevenseg[6:0] = 7'b0000110;
            4'hF: sevenseg[6:0] = 7'b0001110;
            default: sevenseg[6:0] = 7'b1111111;
        endcase
    end
endmodule
```

四. 仿真结果展示与分析

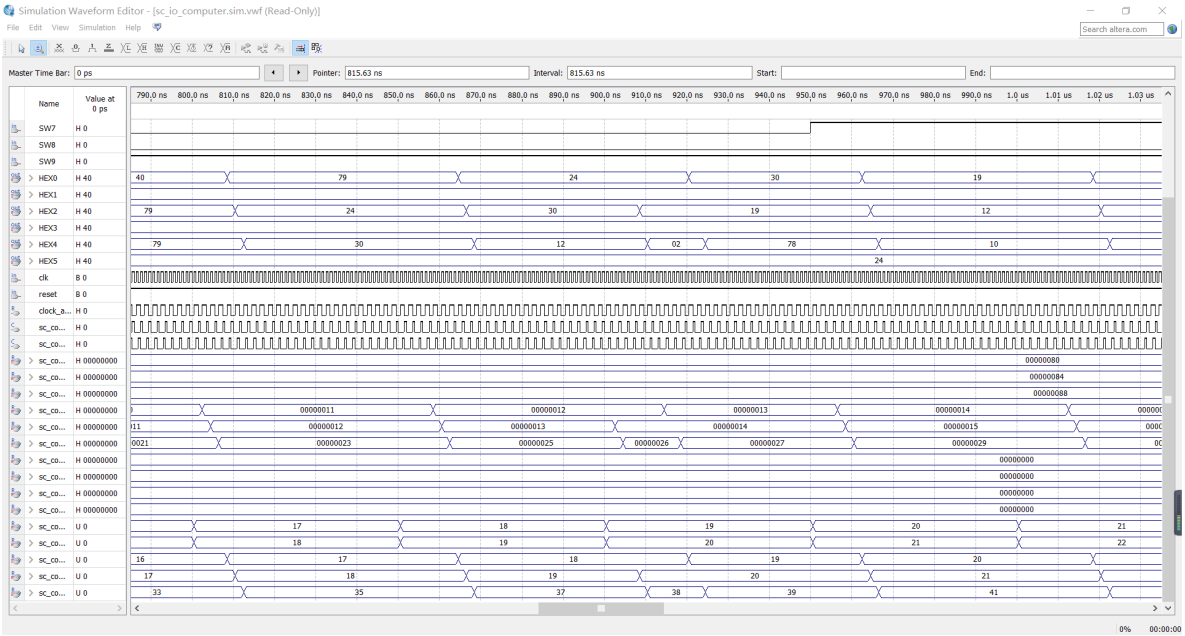
将上述核心部分修改完成并且编译无误后，使用 sc_computer_test_wave_02.vwf 波形文件对结果进行仿真，仿真图形如下图所示。

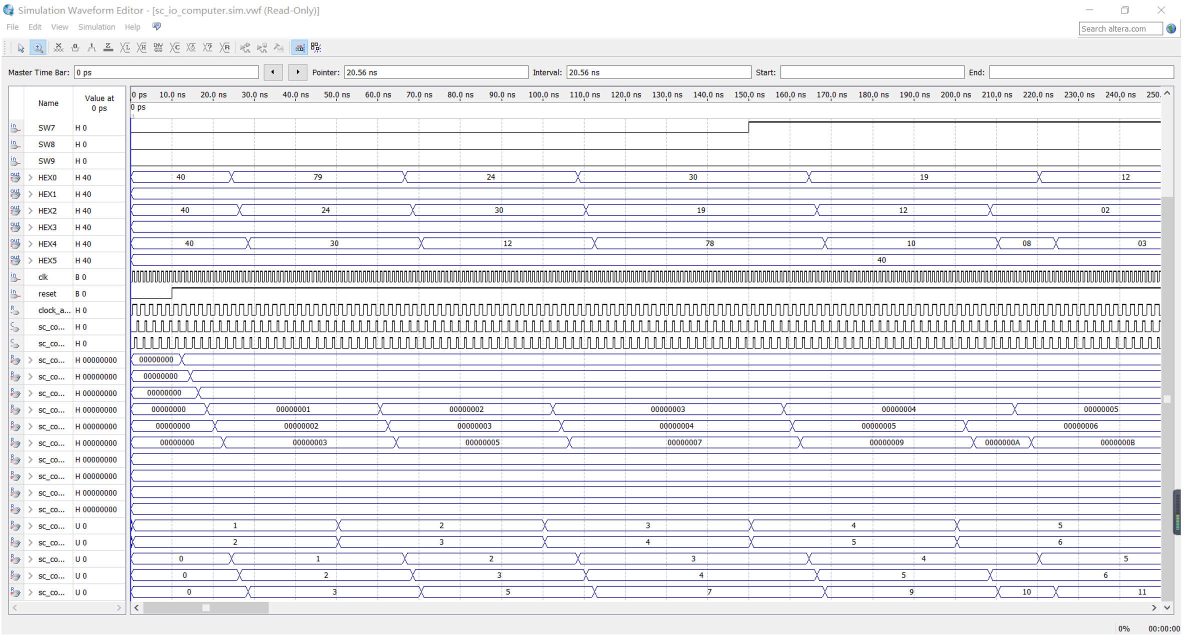
先展示全局波形：



由图中可以看到，无论是clock_out、in_port、out_port还是HEX都有都有相应的波形或者数字显示出来，我们对波形进行进一步的放大，结合mif文件中的汇编语句分析部分结果。

可以看到当reset信号置为高电平以后，图中in_port0和in_port1的值分别对应了SW9-SW5，SW4-SW0的拼接后的值，这个值被取出后分别输出给out_port0, out_port1，这两个值相加的结果输出给out_port2，再分别对三个out_port中的值进行七段译码，输出六个HEX。实验结果与预期一致。





五.扩展实验

扩展实验部分要求实现汉明距离的计算，这部分修改的内容主要有alu.v，sc_cu.v，sc_instmem.v三个文件。

首先给假设求前四位hamm距离的操作是先对输入的a, b求异或，再将最低四位的值加起来，即求出最低四位不同的值的位数。

```
module alu (a,b,aluc,s,z);
   input [31:0] a,b;
   input [3:0] aluc;
   output [31:0] s;
   output        z;
   reg [31:0] s;
   reg        z;
    reg [31:0] axorb;
   always @ (a or b or aluc)
      begin                                // event
         casex (aluc)
                4'b1011:
                begin
                   axorb = a ^ b;          // 1011 先求a异或b;
                    s = axorb[0] + axorb[1] + axorb[2] + axorb[3] + axorb[4];
// 计算5位汉明距离
                end
             4'bx000: s = a + b;           //x000 ADD
             4'bx100: s = a - b;           //x100 SUB
             4'bx001: s = a & b;           //x001 AND
             4'bx101: s = a | b;           //x101 OR
             4'bx010: s = a ^ b;           //x010 XOR
             4'bx110: s = a << 16;         //x110 LUI: imm << 16bit

             4'b0011: s = $unsigned(b) << a;  //0011 SLL: rd <- (rt << sa)
             4'b0111: s = $unsigned(b) >> a;  //0111 SRL: rd <- (rt >> sa)
(logical)
             4'b1111: s = $signed(b) >>> a;   //1111 SRA: rd <- (rt >> sa)
(arithmetic)
```

```
            default: s = 0;
         endcase
         if (s == 0 )  z = 1;
            else z = 0;
      end
endmodule
```

我们将求hamm距离设为R型指令，指令的opcode各位均为0，这里假设funct为110000，由此在aluc.v中修改相应的信号。

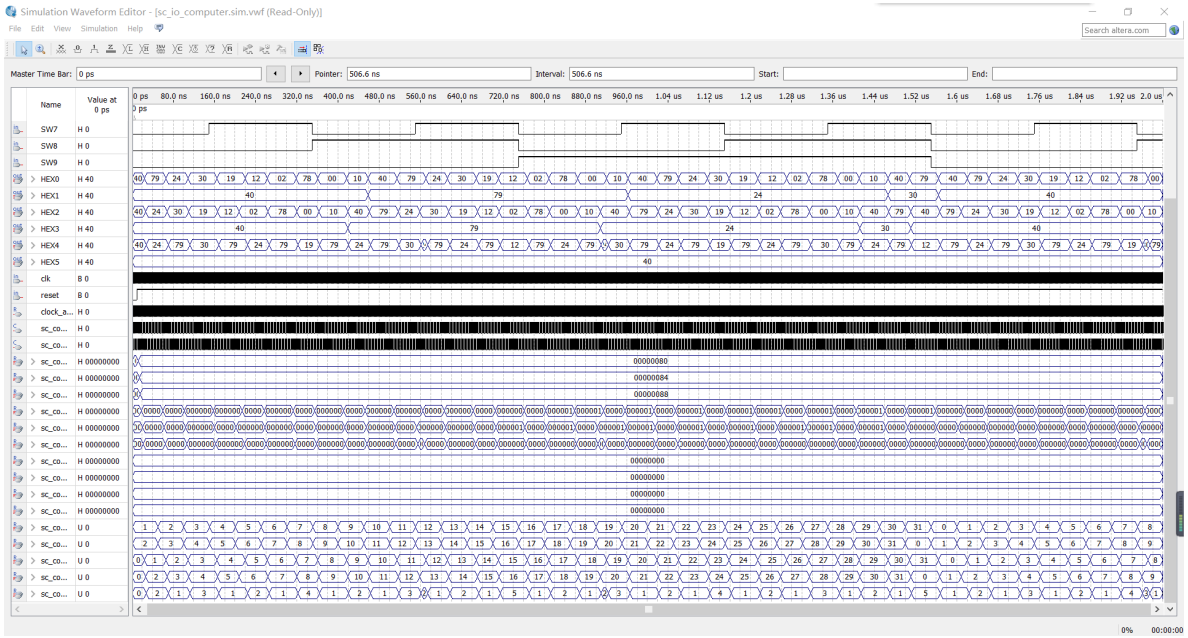
```
module sc_cu (op, func, z, wmem, wreg, regrt, m2reg, aluc, shift,
              aluimm, pcsource, jal, sext);
   input  [5:0] op,func;
   input        z;
   output       wreg,regrt,jal,m2reg,shift,aluimm,sext,wmem;
   output [3:0] aluc;
   output [1:0] pcsource;
   wire r_type = ~|op;
   wire i_add = r_type & func[5] & ~func[4] & ~func[3] &
                ~func[2] & ~func[1] & ~func[0];          //100000
   wire i_sub = r_type & func[5] & ~func[4] & ~func[3] &
                ~func[2] &  func[1] & ~func[0];          //100010

   wire i_and = r_type & func[5] & ~func[4] & ~func[3] &
                     func[2] & ~func[1] & ~func[0];             //100100
   wire i_or  = r_type & func[5] & ~func[4] & ~func[3] &
                     func[2] & ~func[1] & func[0];              //100101
   wire i_xor = r_type & func[5] & ~func[4] & ~func[3] &
                     func[2] & func[1] & ~func[0];              //100110
   wire i_sll = r_type & ~func[5] & ~func[4] & ~func[3] &
                ~func[2] & ~func[1] & ~func[0];          //000000
   wire i_srl = r_type & ~func[5] & ~func[4] & ~func[3] &
                ~func[2] & func[1] & ~func[0];           //000010
   wire i_sra = r_type & ~func[5] & ~func[4] & ~func[3] &
                ~func[2] & func[1] & func[0];          //000011
   wire i_jr  = r_type & ~func[5] & ~func[4] & func[3] &
                ~func[2] & ~func[1] & ~func[0];          //001000
    wire i_hamm = r_type & func[5] & func[4] &
                     ~func[3] & ~func[2] & ~func[1] & ~func[0]; //110000 求汉明距离的操作funct设置为110000

   wire i_addi = ~op[5] & ~op[4] &  op[3] & ~op[2] & ~op[1] & ~op[0]; //001000
   wire i_andi = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] & ~op[0]; //001100

   wire i_ori  = ~op[5] & ~op[4] &  op[3] &  op[2] & ~op[1] &  op[0]; //001101
   wire i_xori = ~op[5] & ~op[4] &  op[3] &  op[2] &  op[1] & ~op[0]; //001110
   wire i_lw   =  op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] &  op[0]; //100011
   wire i_sw   =  op[5] & ~op[4] &  op[3] & ~op[2] &  op[1] &  op[0]; //101011
   wire i_beq  = ~op[5] & ~op[4] & ~op[3] &  op[2] & ~op[1] & ~op[0]; //000100
   wire i_bne  = ~op[5] & ~op[4] & ~op[3] &  op[2] & ~op[1] &  op[0]; //000101
   wire i_lui  = ~op[5] & ~op[4] &  op[3] &  op[2] &  op[1] &  op[0]; //001111
   wire i_j    = ~op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] & ~op[0]; //000010
   wire i_jal  = ~op[5] & ~op[4] & ~op[3] & ~op[2] &  op[1] &  op[0]; //000011


   assign pcsource[1] = i_jr | i_j | i_jal;
   assign pcsource[0] = ( i_beq & z ) | (i_bne & ~z) | i_j | i_jal ;
```

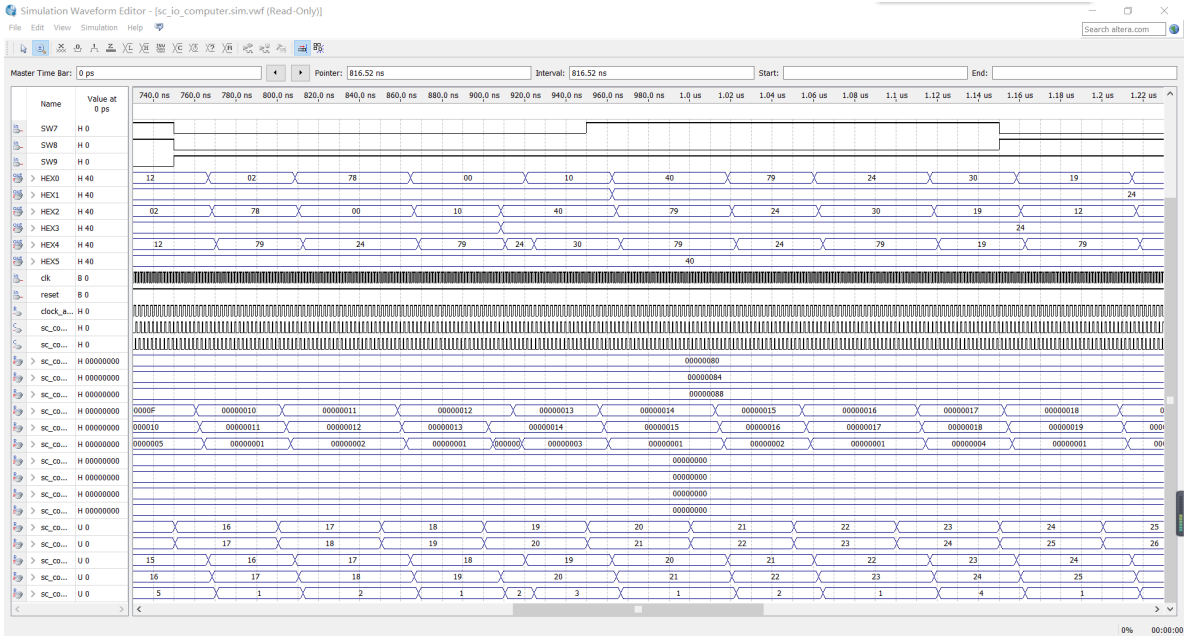
```
   assign wreg = i_add | i_sub | i_and | i_or   | i_xor  |
                 i_sll | i_srl | i_sra | i_addi | i_andi |
                 i_ori | i_xori | i_lw | i_lui  | i_jal | i_hamm; //加上hamm信号

    // hamm运算的aluc值设为1011
   assign aluc[3] = i_sra | i_hamm;
   assign aluc[2] = i_sub | i_or | i_srl | i_sra | i_ori | i_lui | i_beq |
i_bne;
   assign aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori | i_lui | i_hamm;
   assign aluc[0] = i_and | i_or | i_sll | i_srl | i_sra | i_andi | i_ori |
i_hamm;
   assign shift   = i_sll | i_srl | i_sra ;

   assign aluimm  = i_addi | i_andi | i_ori | i_xori | i_lw | i_sw | i_lui;
   assign sext    = i_addi | i_lw | i_sw | i_beq | i_bne | i_lui;
   assign wmem    = i_sw;
   assign m2reg   = i_lw;
   assign regrt   = i_addi | i_andi | i_ori | i_xori | i_lw | i_sw | i_lui;
   assign jal     = i_jal;

endmodule
```

最后修改mif文件中的一条机器码，使其计算a hamm b;

```
DEPTH = 16;           % Memory depth and width are required %
WIDTH = 32;           % Enter a decimal number %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless %
                      % otherwise specified, radixes = HEX %
CONTENT
BEGIN

0 : 20010080;        % (00) main: addi $1, $0, 128 # outport0, inport0
   %
1 : 20020084;        % (04)       addi $2, $0, 132 # outport1, inport1
   %
2 : 20030088;        % (08)       addi $3, $0, 136 # outport2
    %
3 : 8c240000;        % (0c) loop: lw   $4, 0($1)   # input inport0 to $4
   %
4 : 8c450000;        % (10)       lw   $5, 0($2)   # input inport1 to $5
   %
5 : 00853030;        % (14)       hamm  $6, $4, $5  # add inport0 with inport1
to $6 %
6 : ac240000;        % (18)       sw   $4, 0($1)   # output inport0 to outport0
    %
7 : ac450000;        % (1c)       sw   $5, 0($2)   # output inport1 to outport1
    %
8 : ac660000;        % (20)       sw   $6, 0($3)   # output result to outport2
   %
9 : 08000003;        % (24)       j loop           #
   %
END ;
```

之后运行仿真波形，展示结果如下





简单验证一下：17的二进制码为10001，18的二进制码为10010，五位hamm距离的确为2，证明我们的结果正确。

六. 实验总结和感想

本次实验中，我们了解了如何在原有单周期CPU的基础上，加上IO端口的扩展，实现了从IO模块中取值、计算最终输出回IO中的功能，并且直观地感受了CPU从data memory和 IO模块中取值的区别，同时我们通过自己编写Verilog代码，使我们对Verilog语法的使用有了进一步的认识。这次实验设计让我们在实践中加强了对CPU、存储单元和IO扩展单元互相之间联系和工作的了解，非常感谢各位老师和助教对我的帮助和指导！