Lab4 简易单周期 CPU I/O 扩展和新指令扩展实验报告

黄奔皓*

上海交通大学

目录

1 **实验目的** 2

2 **实验设备** 2

3 **实验要求** 2

4 **实验任务** 3

4.1 为数据存储器添加 I/O 支持 3

4.2 修改 lab3 中的顶层文件 sc_computer_main 4

4.3 顶层文件连接 5

4.3.1 时钟产生 5

4.3.2 inport 拓展 6

4.3.3 IO 连线 6

4.4 仿真模拟 7

4.5 板级验证 8

4.6 添加汉明距离指令 8

4.6.1 仿真模拟 10

4.6.2 板级验证 11

5 **总结思考** 11

*感谢陈老师的悉心指导

1 实验目的

1. 掌握 CPU 的外设 I/O 模块的设计方法。理解 I/O 地址空间的译码设计方法。

2. 掌握 Vivado 仿真、实现、板级验证方式。

3. 通过扩展新指令的实现，深入理解 CPU 对指令的译码、执行原理和实现方式。

2 实验设备

1. 计算机 1 台 (尽可能达到 8G 及以上内存)；

2. Xilinx 的 Vivado 开发套件 (2020.2)

3 实验要求

1. 添加 I/O 接口模块，实现 CPU 对 I/O 空间的访问。

2. 依据提供的标准测试程序代码，对单周期 CPU 的 I/O 模块进行板级验证。要求 CPU 能够采用查询方式接收输入按键或开关的状态，并产生相应的输出状态，驱动数码管显示结果，从而验证 CPU 可正确执行 I/O 读写指令。

3. 自定义一条 R-Type 新指令，求两个 32bit 操作数的汉明距离。修改 CPU 控制部件和执行部件模块代码，支持新指令的操作。

4. 修改提供的测试程序代码，对新增加指令进行板级验证。

4 实验任务

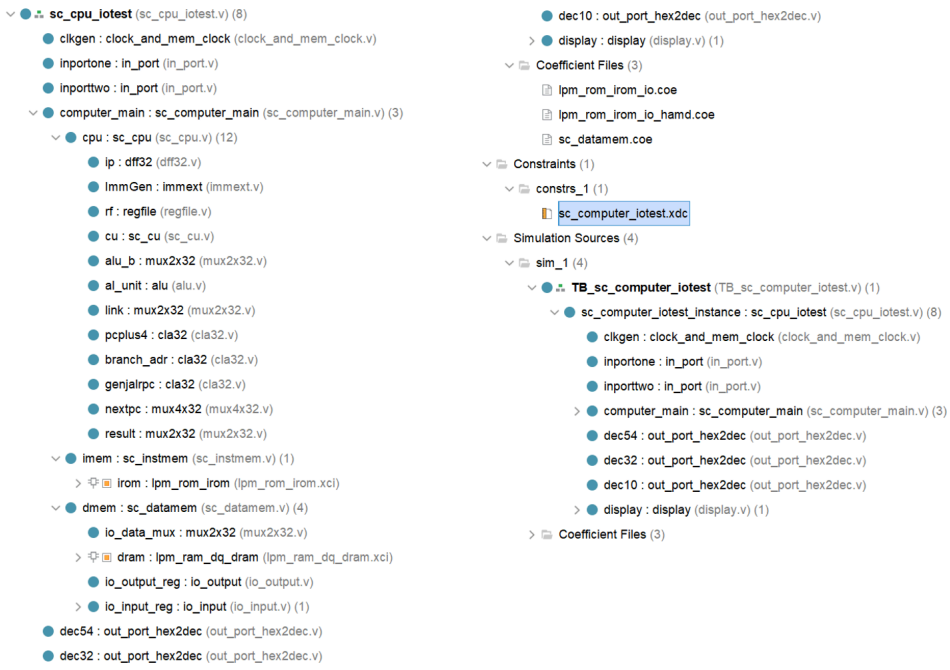

Figure 1: 实验文件树

4.1 为数据存储器添加 I/O 支持

首先，我们要对 sc_datamem.v 进行修改，为其添加 I/O 支持。添加代码如下：

```
42        //IO output , io_output ,add here
43        io_output io_output_reg (
44            .resetn(resetn),
45            .addr(addr),
46            .datain(datain),
47            .io_clk(dmem_clock),
48            .write_io_enable(write_io_enable),
49            .out_port0(out_port0),
50            .out_port1(out_port1),
51            .out_port2(out_port2)
52        );
53
54
55        //IOinput , io_input ,add here
56        io_input io_input_reg (
57            .addr(addr),
58            .io_clk(dmem_clock),
59            .io_read_data(io_read_data),
60            .in_port0(in_port0),
61            .in_port1(in_port1)
62        );
```

Figure 2: 添加 I/O 支持

注意 根据图 3，其中 io_clock 参数应该用 dmem_clock。以及要注意 write_io_enable 的使用，我们通过 addr[7] 来区分当前是 IO 写使能，还是数据存储器写使能。**这等于是起到了一个二路选择器的作用**。

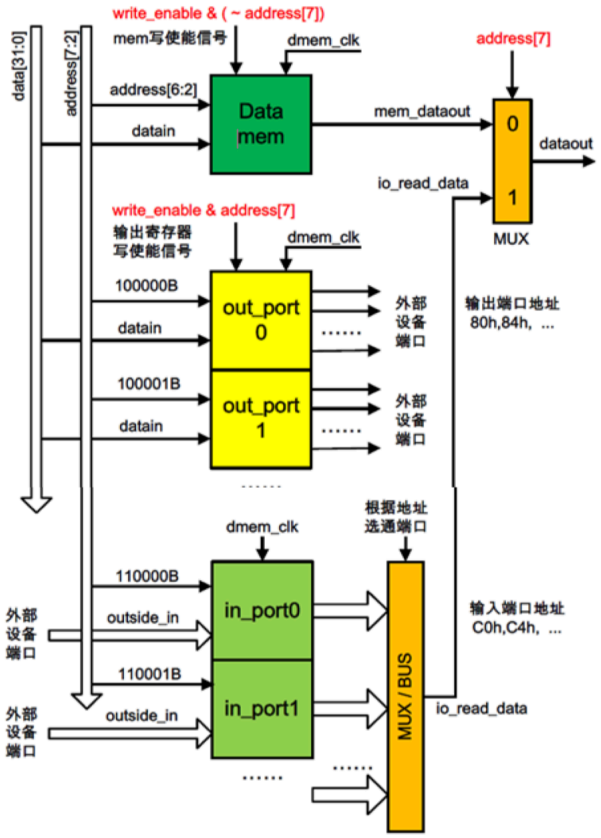

Figure 3: 统一编址方式 I/O 端口扩展原理图解

4.2 修改 lab3 中的顶层文件 sc_computer_main

由于 IO 口的加入和 imem_clock 与 mem_clock 的分划，我们需要对 lab3 中的顶层文件进行相应的修改。修改后的部分代码如下：

```
//sc_cpu ,CPU module.
sc_cpu cpu(
.clock(clock),
.resetn(resetn),
.inst(inst),
.mem(memout),
.pc(pc),
.wmem(wmem),
.aluout(aluout),
.data(data));

//  sc_instmem, instruction memory.
  sc_instmem  imem (
  .addr(pc),
  .inst(inst),
  .clock(clock),
  .imem_clock(imem_clock));
```

```
//sc_datamem, data memory.
  sc_datamem  dmem (
  .resetn(resetn),
  .addr(aluout),
  .datain(data),
  .dataout(memout),
  .we(wmem),
  .clock(clock),
  .dmem_clock(dmem_clock),
  .out_port0(out_port0),
  .out_port1(out_port1),
  .out_port2(out_port2),
  .in_port0(in_port0),
  .in_port1(in_port1)
  );
```

Figure 4: 修改 lab3 顶层文件

注意 修改的重点在于区分三个时钟：clock,imem_clock,dmem_clock。我们可以根据模块的相应参数名的提示进行填入。在增加 hamd 指令之前，lab3 文件中我们只需对顶层文件进行修改，其他部分可以保持不动。

4.3 顶层文件连接

4.3.1 时钟产生

本次实验通过 clock_and_mem_clock 模块产生三个时钟：clock,imem_clock,dmem_clock，分别用于 sc_cpu，Instruction ROM，DataRAM、io_input、io_output。

```
//clock_and_mem_clock unit, generate clock, imem_clock, dmem_clock  , add here
wire imem_clock, dmem_clock;
wire clock;


clock_and_mem_clock clkgen(
     .sys_clk_in(sys_clk_in),
     .clock_out(clock),
     .imem_clock(imem_clock),
     .dmem_clock(dmem_clock)
);
```

Figure 5: 时钟产生模块

注意 在顶层文件的提供代码中没有声明三个时钟变量，为此我们需要自行声明。我们不能用 reg 来声明，而需要用 wire，因为对于 clock_and_mem_clock 模块，三个时钟为输出端口，而从模块外部来看，**输出必须连接到线网（wire）类型的变量，而不能连接到** reg **类型的变量**。

4.3.2 inport 拓展

在顶层文件中，我们需要用 in_port 模块将 5 位的 sw_pin 和 dp_pin 拓展为 32 位。in_port 模块代码如下:

```
module in_port (sw,out);
    input [4:0] sw;
    output [31:0] out;
    assign out = {27'h0,sw};
endmodule
```

Figure 6: in_port 模块

根据模块的声明我们知道，在顶层文件中我们需要如下例化两个 in_port 模块:

```
//extend in_port0 to 32bit,  in_port0 = {27'b0,sw_pin}; add here
 in_port inportone(
    .sw(sw_pin),
    .out(in_port0)
 );


//extend in_port1 to 32bit,  in_port1 = {27'b0,dip_pin}; add here
  in_port inporttwo(
    .sw(dip_pin),
    .out(in_port1)
 );
```

Figure 7: 例化 in_port

4.3.3 IO 连线

在顶层文件中并没有声明输出端口和输入端口，如果不进行主动声明，那么在模拟中就会遇到输出端口 1=ZZZZZ 的情况，这种情况通常是因为线没有成功连上。如果没有输入，端口就会是高阻。因此，需要在顶端文件中声明变量:

```
module sc_cpu_iotest(
    input sys_rst_n,  //active low global reset
    input sys_clk_in, //100MHz clock
    input [4:0] sw_pin,//left 8 sw_pin
    input [4:0] dip_pin,//right 8 dip_pin
    output [7:0] seg_data_0_pin, //output DP1,G1,F1,E1,D1,C1,B1,A1,  left
    output [7:0] seg_data_1_pin,  //output DP0,G0,F0,E0,D0,C0,B0,A0,  right
    output [7:0] seg_cs_pin, //DN1_K4,DN1_K3,DN1_K2,DN1_K0,DN0_K4,DN0_K3,DN0_K2,DN0_K1 ,left to right
    output [0:15] led_pin //left to right
     );

    wire [3:0] HEX4b5,HEX4b4,HEX4b3,HEX4b2,HEX4b1,HEX4b0;
    wire [31:0] out_port0, out_port1, out_port2;
    wire [31:0] in_port0, in_port1;
```

Figure 8: 声明 IO 变量

4.4 仿真模拟

完成以上的代码书写后，我们就可以正常进行仿真了。

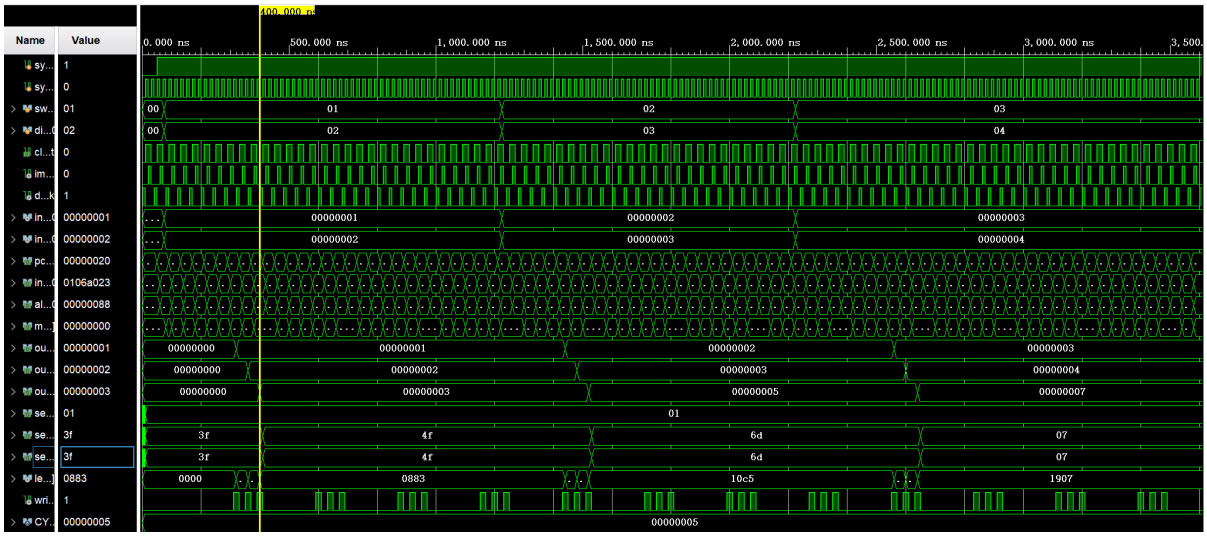

Figure 9: 仿真模拟图

图 10 为仿真结果的部分截图。当程序执行到 400ns，完成第一次计算求和的输出。根据代码，outport2 应该输出 inport0 和 inport1 的和，这与仿真图吻合。而 outport0 输出值与 inport0 一致，outport1 输出值也与 inport1 一致，说明数据通路正确连接，代码运行正常。

4.5 板级验证

模拟仿真成功后，我们产生比特流文件，并烧入电路板进行板级验证。如下是板级验证图：

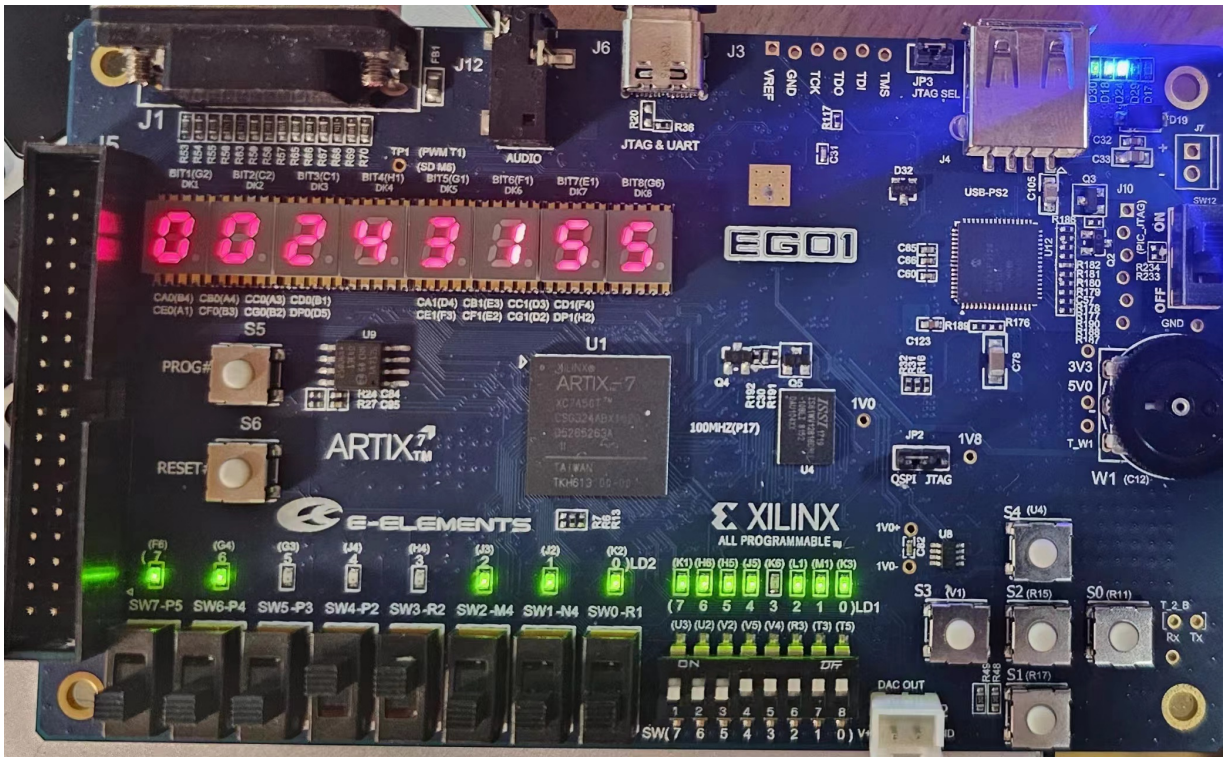

Figure 10: 板级验证图

此时当 sw_pin 输入为 11000 (24)，dip_pin 输入 (11111) 31。求和输出的结果 55。

注 sw_pin 的数值由板子左下角的开关控制，而 dip_pin 由其右边的白色小开关控制。

4.6 添加汉明距离指令

汉明距离是是两数的二进制表示中对应位不相同的数量之和。我们添加 R-type 的 hamd 指令，并设置二进制码如下：

Table 1: hamd 指令格式各位段二进制码和机器码

func3	func7	op	aluc
111	0100000	0110011	1111

在不与其他指令的 aluc 重复的情况下，我们选取 hamd 指令的 aluc 值 =1111。我们如下修改 sc_cu.v 文件:

```
// hamd instruction
wire i_hamd = r_type & (func3 == 3'b111) & (func7 == 7'b0100000);

//  please complete the deleted code.
wire i_and = r_type & (func3 == 3'b111 );          //111
wire i_or  = r_type & (func3 == 3'b110 );          //110
wire i_xor = r_type & (func3 == 3'b100 );          //100
wire i_sll = r_type & (func3 == 3'b001 );//001
wire i_srl = r_type &( func3 == 3'b101 ) & ~inst[30];          //101锛?0
wire i_sra = r_type & (func3 == 3'b101 ) & inst[30];          //101,1

assign aluc[3] = i_sub | i_srai | i_beq | i_bne | i_hamd;
assign aluc[2] = i_and | i_or | i_xor | i_srl | i_sra | i_andi | i_ori| i_xori | i_srli | i_srai | i_hamd;
assign aluc[1] = i_and | i_or | i_andi | i_ori |i_lui | i_hamd;
assign aluc[0] = i_and | i_sll | i_srl | i_sra | i_andi| i_slli | i_srli | i_srai | i_hamd;
```

Figure 11: sc_cu 代码修改

而为了添加 hamd 指令的计算过程，我们需要修改加法器文件 alu.v。a 和 b 汉明距离的一种计算方式为：将 a 和 b 取异或后得到的值存入 x，其中 a，b，x 均为 32 位二进制数。当 x 不为 0 时，查看 x 的 lsb 位是否为 1，若是则将汉明距离加 1。此后将 x 右移 1 位，左边用 0 补充。如此反复操作 32 次。

具体代码如下:

```
4'b1111:
    begin
    s = 0;
    x = a ^ b;
    for (i = 1;i <= 32; i = i + 1) begin
        if (x & 8'b1)
            begin
                s = s + 1;
            end
        x = x >> 1;
        end
    end
```

Figure 12: alu 单元增加汉明距离计算功能

注意 注意，这里的循环体不能用 while+ 含变量的条件判断来实现。见错误代码:

```
4'b1111:  // hamming distance
    begin
    s = 0;
    x = a ^ b;
    while(x!=32'b0) begin
        if (x & 8'b1) begin
            s = s + 1;
        end
        x = x >> 1;
    end
    end
```

Figure 13: 汉明距离计算的错误实现

如果在这段代码的基础上运行仿真，会发现结果是正确的，但是无法进行板级验证，信息将显示循环次数超过 2000 次。这是因为 while 内的条件含有 x 这个变量，而程序在进行综合的时候，实际上是将循环展开的，如果含有变量，它将无法确定将循环展开多少次，从而出错。

4.6.1 仿真模拟

修改完控制单元和加法单元后，我们进行仿真模拟。

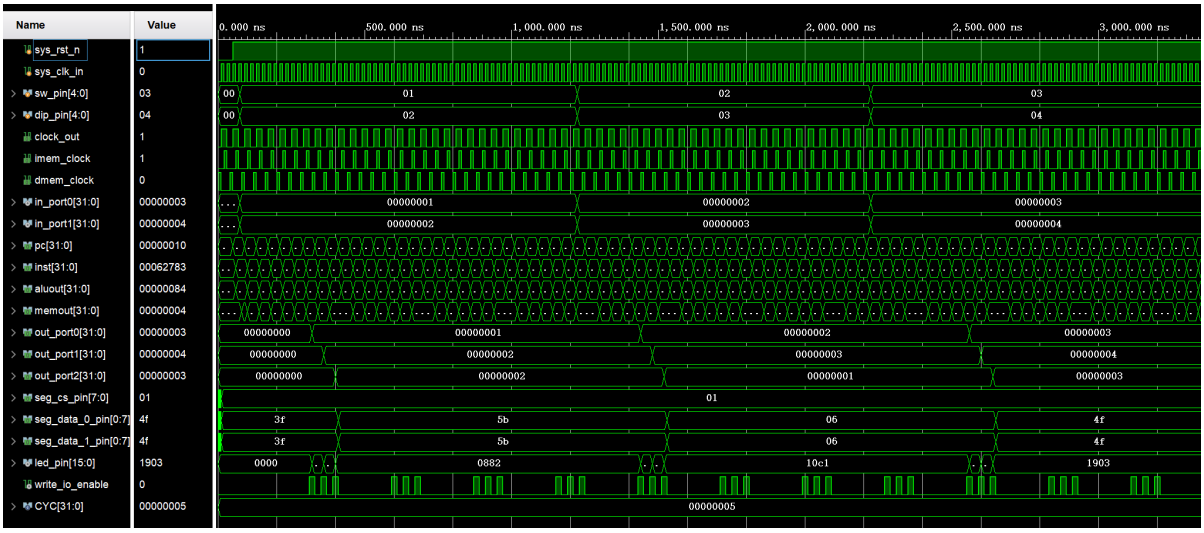

Figure 14: 加入 hamd 指令后的仿真模拟图

图 14 为仿真结果的部分截图。当程序执行到 400ns，完成第一次汉明距离计算的输出。根据代码，outport2 应该输出 inport0=1 和 inport1=2 的汉明距离 2，这与仿真图吻合。而 outport0 输出值与 inport0 一致，outport1 输出值也与 inport1 一致，说明数据通路正确连接，hamd 指令正常实现。

4.6.2 板级验证

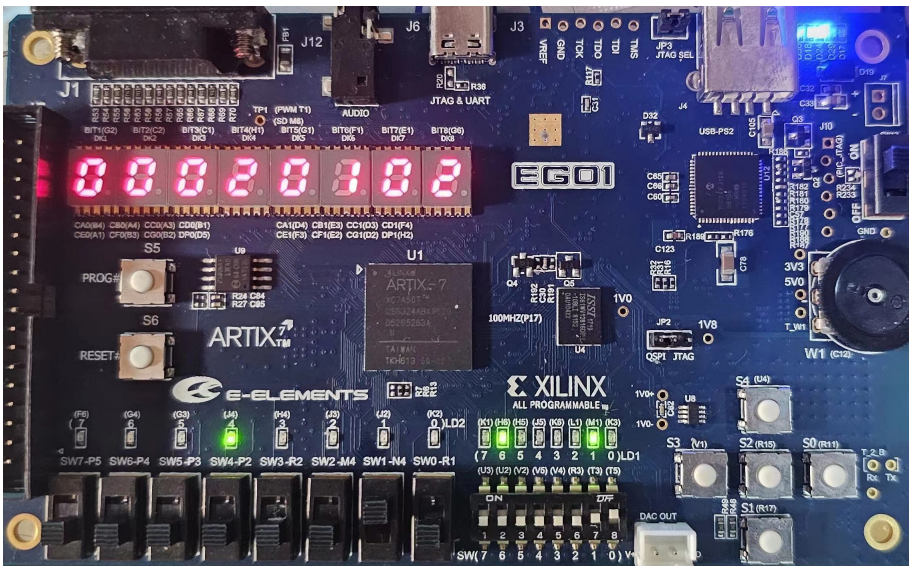

Figure 15: 加入 hamd 指令后的板级验证

图 15 中是加入加入 hamd 指令后的板级验证的结果。此时 sw_pin=2 (10)，而 dip_pin=1，而 1 和 2 的汉明距离为 2，这与板上 outport2 显示为 02 相吻合，说明 hamd 指令得到了正确实现。

5 总结思考

经过本次实验，我有以下的思考与经验总结：

- IP 例化时要注意 width 和 depth，注意例化 ROM 时要调整 single port RAM 为 single port ROM，初始化时要取消勾选 Primitives Output Register，勾选 Fill Remaining Memory Locations
- 对于 verilog 语言，reg 和 wire 类型在例化模块时要着重区分：

  (1) 输入端口（input）：从模块内部看，输入端口必须为线网（wire）数据类型；从模块外部看，输入端口可以连接到线网或 reg 数据类型的变量。

  (2) 输出端口 (output)：从模块内部来讲，输出端口可以是线网或 reg 数据类型；从模块外部来看，输出必须连接到线网（wire）类型的变量，而不能连接到 reg 类型的变量。

- 如果出现模拟正常，但是产生比特流时报错，则需要在约束文件中添加

- 输出端口的例化时要保证名字一致。比如 sc_data_mem 模块中的 dmem_clock 是否在例化的时候写成了 dmem_clk。此外，输入端口名的时候不要过快，一旦出现错误，比如一个字母打错了，多了一个下划线，之后 debug 的时候会不容易发现，导致浪费时间。

- 在写代码的时候也要结合已经学习的计算机组成课本知识。拿汉明距离的计算中关于循环的问题来说，老师在讲如何将 c 语言写的程序用 RISC-V 语言转写时，讲过循环展开这个概念。我们要善于找到知识之间的联系。

- Verilog 模块化的思想很重要。经过 lab1 到 lab4 的实验过程，我发现我们的文件树越来越庞大，但层次也越来越清晰，分工也越来越明确。我认为这是设计一个复杂系统需要掌握的思想。