浙江大学

题目(中)　计算机系统 I Project 实验报告

姓名与学号　张远帆 3230104567

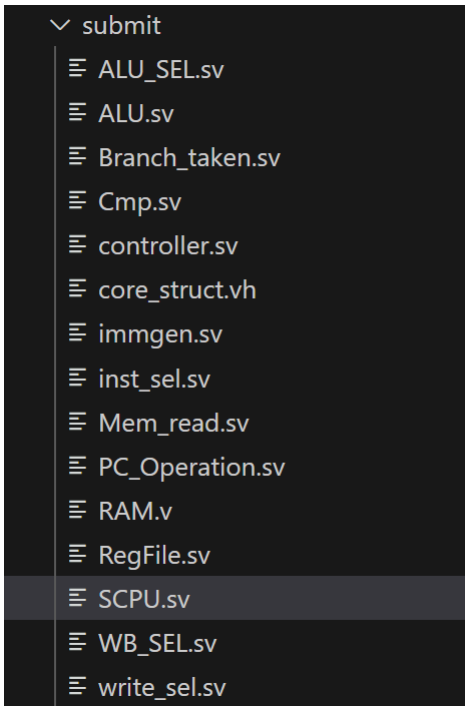
指导教师　卢立

年级与专业　大一 信息安全

所在学院　求是学院

System-Project Report

数据通路设计



文件夹结构见上图

1. 指令获取阶段：由 PC_OP 模块完成。（关于 next_pc 的操作在后面的 branchtaken 模块会提及）对于读入的指令，由于其为 64 位（16 位 16 进制数），故需根据指针模 8 的值判断取高 8 位还是低 8 位。

2. 指令译码阶段：由 controller 模块完成。Controller 模块读入一条指令 inst，后将其分解成各个信号，区别需要执行的操作，如是否需要读、写内存，寄存器位置，指令类型等等，包括 Opcode, Rs1, Rs2, Rd, Imm 等。

   （此处代码过于复杂就不放了……）

3. 指令执行阶段：分为 ALU,CMP,IMM_GEN,ALU_SEL 等模块

ALU：负责进行算数运算，如 add，sub 等等。

```
always @* begin
    case (alu_op)

      ALU_ADD: result = a + b;
      ALU_SUB: result = a - b;
      ALU_AND: result = a & b;
      ALU_OR: result = a | b;
      ALU_XOR: result = a ^ b;
      ALU_SLT: begin
        if(a == 0 && b ==-1)result = 0;
        else if (a == -1 && b == 0)result = 1;
        else result = (a < b) ? 1 : 0;
      end
      ALU_SLTU: result = (a < b) ? 1 : 0; // 无符号比较
      ALU_SLL: result = a << b[5:0];
      ALU_SRL: result = a >> b[5:0];
      ALU_SRA: begin
        result = $signed(a) >>> b[5:0];
      end
      ALU_ADDW: result = $signed(a) + $signed(b);
      ALU_SUBW: result = $signed(a) - $signed(b);
```

图表 1ALU 部分代码

ALU_SEL:负责对 ALU 的运算数和符号进行选择，根据指令的不同选择 PC 或 reg1、reg2 的值作为 alu 的操作数。

```
reg [63:0] alu_tp1, alu_tp2;
always @* begin//ld指令将alu_op1设置为reg_data1，将alu_op2设置为imm_va
    alu_tp1 = (alu_asel-1 == 0) ? reg_data1 : pc;  // ALU A 选择
    alu_tp2 = (alu_bsel-1 == 0) ? reg_data2 : imm_val;  // ALU B 选
end
assign alu_op1 = alu_tp1;
assign alu_op2 = alu_tp2;
```

CMP：负责对 b 型指令的判断，如果条件成立则 branch_taken=1 使 nextpc 的值变更为 alu 的计算结果。

```
    always @* begin
        case (cmp_op)
            CMP_EQ: cmp_result = (a == b);
            CMP_NE: cmp_result = (a != b);
            CMP_LT: begin
            if(a == 0 && $signed(b)>0)cmp_result = 1;
            else
            cmp_result = ($signed(a) < $signed(b));
            end
            CMP_GE: cmp_result = ($signed(a) >= $signed(b));
            CMP_LTU: cmp_result = ($unsigned(a) < $unsigned(b)); /
            CMP_GEU: cmp_result = ($unsigned(a) >= $unsigned(b));
            default: cmp_result = 0; // 默认情况下，输出0
        endcase
    end
```

IMMGEN：对于不同种类指令，从 inst 当中选取正确的部分作为立即数(itype,rtype,……)

```
always @(*) begin
        case (immgen_op)
            3'b001: reg_val = {{52{inst[31]}}, inst[31:20]};  // I-Ty
            3'b010: reg_val = {{52{inst[31]}}, inst[31:25], inst[11:7
            3'b011: reg_val = {{52{inst[31]}}, inst[7], inst[30:25],
            3'b100: reg_val = {inst[31:12], 44'b0};  // U-Type
            3'b101: reg_val = {{44{inst[31]}}, inst[19:12], inst[20],
            default: reg_val = 64'b0;  // Default case
        endcase
    end
assign imm_val = reg_val;
endmodule
```

Branch_taken:此处单独新建了一个模块来判断是否需要跳转，如果需要跳转（branchtaken or jumptaken）则修改 nextpc 的值为所需的跳转值，如果无需跳转则将 nextpc 置为 pc+4。

```
    always @(posedge clk or negedge rstn) begin
        if(~rstn) begin
            next_pc = 64'b0;
        end
        else begin
        if (branch_taken) begin
            next_pc = alu_res;
        end
        else if(jump_taken) begin
            next_pc = pc + imm_val;
        end
        else begin
            next_pc = pc + 4;  // 下一条指令
        end
        end
        end
endmodule
```

4. 访存阶段：此处运用了较为简单的内存（即可以在一个周期内完成读取和写入，与 reg 类似）。内存部分分为 imem 和 dmem，分别对应指令内存和数据内存（虽然这两部分数据在本次实验中放在了一起）。由于种种原因，未能在所给的 mem_ift 接口下完成（但本实验为单周期 CPU 设计，使用该接口无法在一个周期内得到返回数据，因此必须进行多周期改造，我之前的想法就是这样，但考虑到难度还是放弃了，后来才发现这其实是 bonus 内容，但我已经懒得做了 qwq）

   Imem 通过一个通道完成，输入 pc 的值（即 addr），可得到返回的 inst。注意需要根据 pc%8 的值来确定 inst 应该取高八位还是低八位。

   Dmem 有两个通道（但每次仅会使用一个），分别用于读和写，使用 we_mem 信号区别。注意所有的地址都是 byte-adderessing,所以需要将地址除以 8。

5. 写回阶段：此处用 wb_val 模块判断写回的值，是 alu 的值还是内存中读取的值还是 pc，最后得出一个结果交给 mem 或 reg 写回。

VERILATE 仿真输出 PASS!!!

```
core   0: >>>>  pass
core   0: 0x00000000000001b0 (0x00000093) li      ra, 0
core   0: 3 0x00000000000001b0 (0x00000093) x1  0x0000000000000000
RegFile: write_addr=01, write_data=0000000000000000

wb_sel = 0
wb_tmp=0000000000000000
core   0: 0x00000000000001b4 (0xc0001073) unimp
core   0: exception trap_illegal_instruction, epc 0x00000000000001b4
core   0:           tval 0x00000000c0001073
core   0: >>>>
core   0: 0x0000000000000000 (0x00100193) li      gp, 1
```

上图是对于每一类指令的测试，Rtype,itype,jtype,stype,utype 和 remain 都能通过，均显示以上内容（请忽略部分用于调试的输出 qwq）

```
rdata=000000000000006f
re_mem=0
read_data=000000000000006f
rdata=000000000000006f,wb_sel=3
core   0: 3 0x00000000000009a8 (0x0000006f)
branch_taken=0
next_pc=00000000000009a8
a=0000000000000000, b=0000000000000000, cmp_op=0
ALU: a=00000000000009a8, b=0000000000000000, alu_op=01010,alu_res=000000000000^Cmake: *** [Makefile:29: verilate] Interr
upt

root@LAPTOP-78LSF82F:/mnt/d/zju/system/sys1-sp23/src/lab5#
```

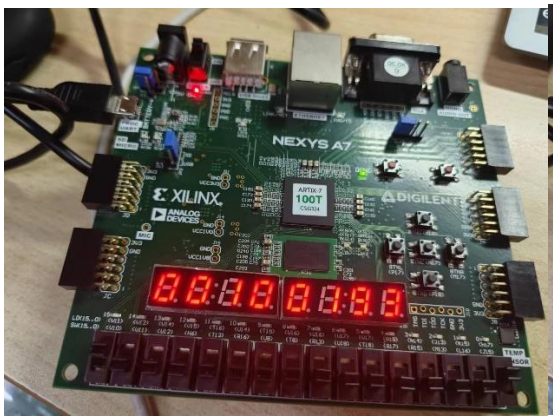
对于 full.hex,程序则会死循环在 9a8 地址，与实验文档所说的一致

上板验证

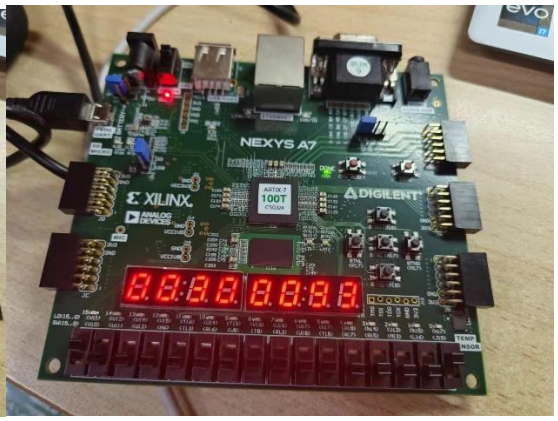
本次上板的是 full.hex 文件，full.hex 前几条命令如下图所示

```
core   0: 0x0000000000000000 (0x00100193) li      gp, 1
core   0: 3 0x0000000000000000 (0x00100193) x3  0x0000000000000001
core   0: 0x0000000000000004 (0x00300093) li      ra, 3
core   0: 3 0x0000000000000004 (0x00300093) x1  0x0000000000000003
core   0: 0x0000000000000008 (0x00700113) li      sp, 7
core   0: 3 0x0000000000000008 (0x00700113) x2  0x0000000000000007
core   0: 0x000000000000000c (0x00208733) add     a4, ra, sp
core   0: 3 0x000000000000000c (0x00208733) x14 0x000000000000000a
core   0: 0x0000000000000010 (0x00a00393) li      t2, 10
core   0: 3 0x0000000000000010 (0x00a00393) x7  0x000000000000000a
core   0: 0x0000000000000014 (0x187718e3) bne     a4, t2, pc + 2448
core   0: 3 0x0000000000000014 (0x187718e3)
core   0: >>>>  test_2
core   0: 0x0000000000000018 (0x00200193) li      gp, 2
core   0: 3 0x0000000000000018 (0x00200193) x3  0x0000000000000002
core   0: 0x000000000000001c (0x00d00093) li      ra, 13
core   0: 3 0x000000000000001c (0x00d00093) x1  0x000000000000000d
core   0: 0x0000000000000020 (0x00b00113) li      sp, 11
core   0: 3 0x0000000000000020 (0x00b00113) x2  0x000000000000000b
```

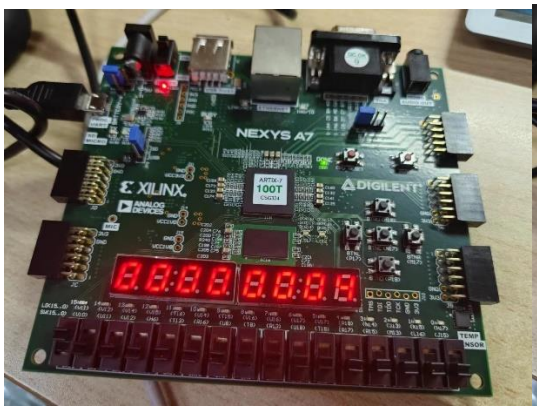
上板结果:



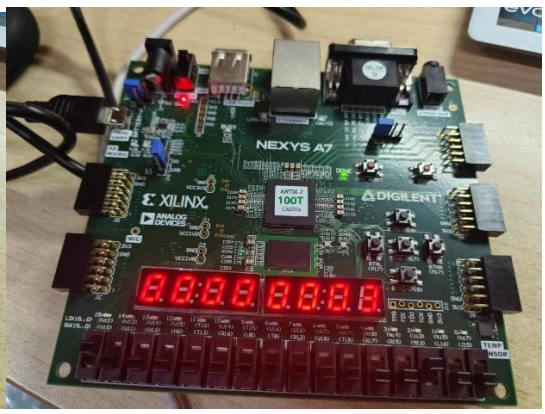
下板验证 1inst1



下板验证 2inst2

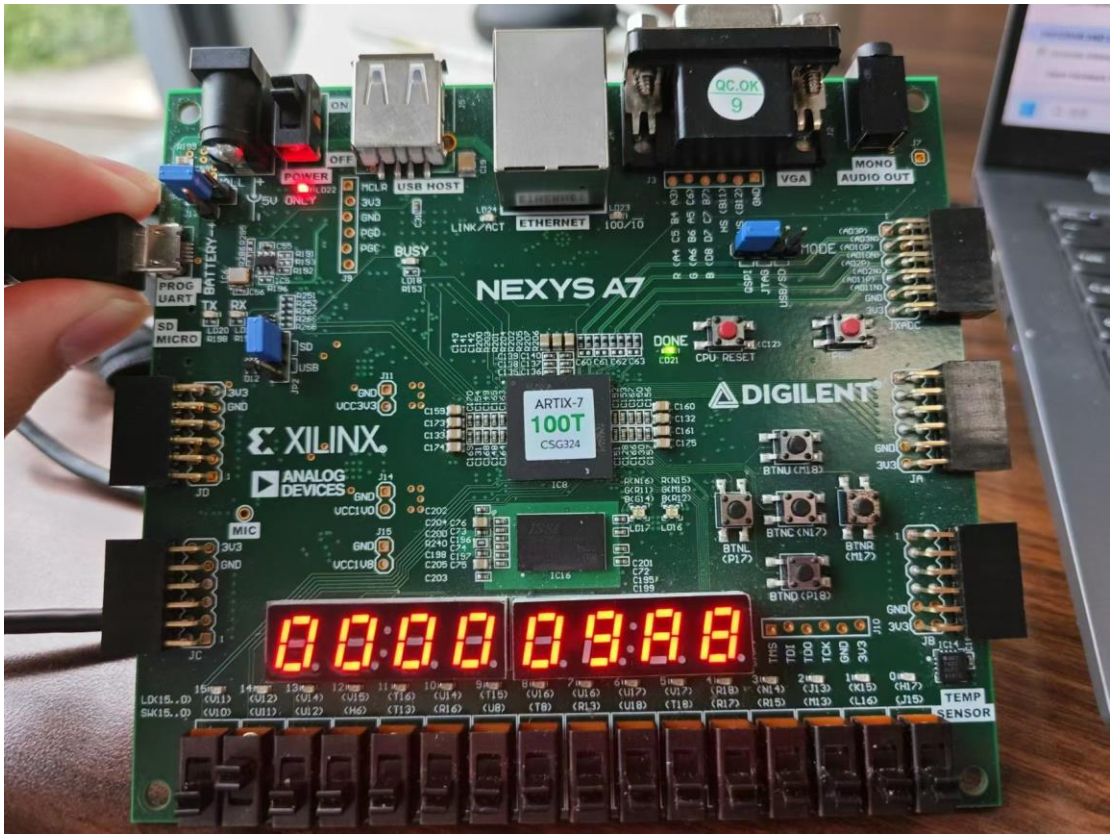


下板验证 3 寄存器的值



下板验证 4rs

可以看到，下板结果均和所给出的指令相符



不拨上 SW15 时，程序顺利卡死在 pc=9A8 处!

（SW14 为时钟分频控制开关，拨上时可以将数码管输出的数字变化至人肉眼可见的变化频率）

对于 System I 的一些建议与个人感受

首先得感谢助教们编写的实验文档！事实上，我觉得看这部分文档，自己一步一步跟着文档操作还是效果比较好的，毕竟计算机也是一门实践性学科。而实验文档编写的也非常通俗易懂，而且知识点和实验步骤兼具，令人可以在做实验的过程中也增长自己的知识。不过，个人认为对于大一的学生来说这部分内容有谢太难了。

我个人最喜欢的实验是 lab5-2,(只可惜这个实验我还是查看了 c 的源代码，理论上应该只看汇编）我觉得这个实验才符合我对信息安全的印象。这个实验应该也是 CMU 的 lab 的一部分，这种从国外先进课程引进的做法也值得点赞!

对于最后的 project 部分，确实是要求比较高，隔壁计科好歹还可以组队，我们每个人都要写，不过我觉得这更是对个人的一种锻炼（相比于 FDS 所谓的"project"，我觉得这才是真正的大作业）当然，在写 project 的过程中，我自然是遇到了无数困难,但通过种种方法(比如问 chatgpt 以及 github 提供的 copilot,真的感谢它们，要不然我觉得我也不可能这么快完成）最终还是写成了完整的数据通路，基本实现了 RICSCV 指令集的内容。多少次被 project 折磨得想要放弃，但最终还是挺过来了。尽管在以后的工作中,我也不会用到 verilog(虽然系统二、三中还要用。。)，但我觉得通过对它的学习，我对计算机系统的理解更进一步加深了。现在，我可以说，我已经知道一台计算机是怎么工作起来的了。不过，对于我个人而言，我觉得我其实还是没有将 verilog 学的很扎实，感觉还是像实验文档中写的"像写 c 语言一样写 verilog"，问题包括而不限于寄存器阻塞与非阻塞赋值乱用，指针更新时序有问题，各个模块之间执行顺序有问题等等……这当然

非常不好，但这便是 verilog 和 c 语言最大的不同之处吧!

总之，对于整个的计算机系统一课程，我个人感觉体验还是比较好的，确实能够学到很多东西，我也非常期待之后的系统二、三课程!

真的感谢助教 gg 们的付出！我们后会有期!