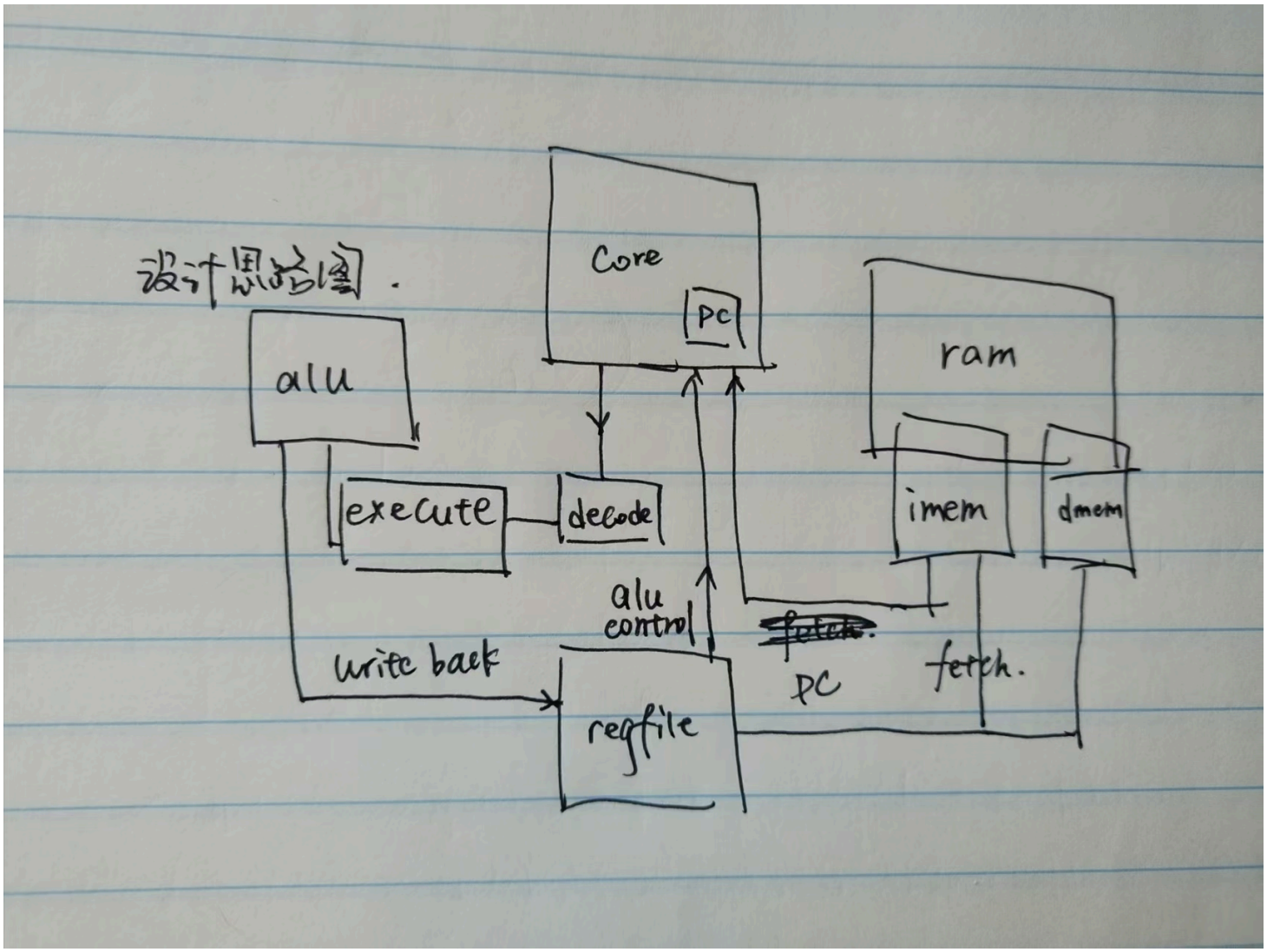
lab01 Single Cycle CPU 实验报告

1.实验概要

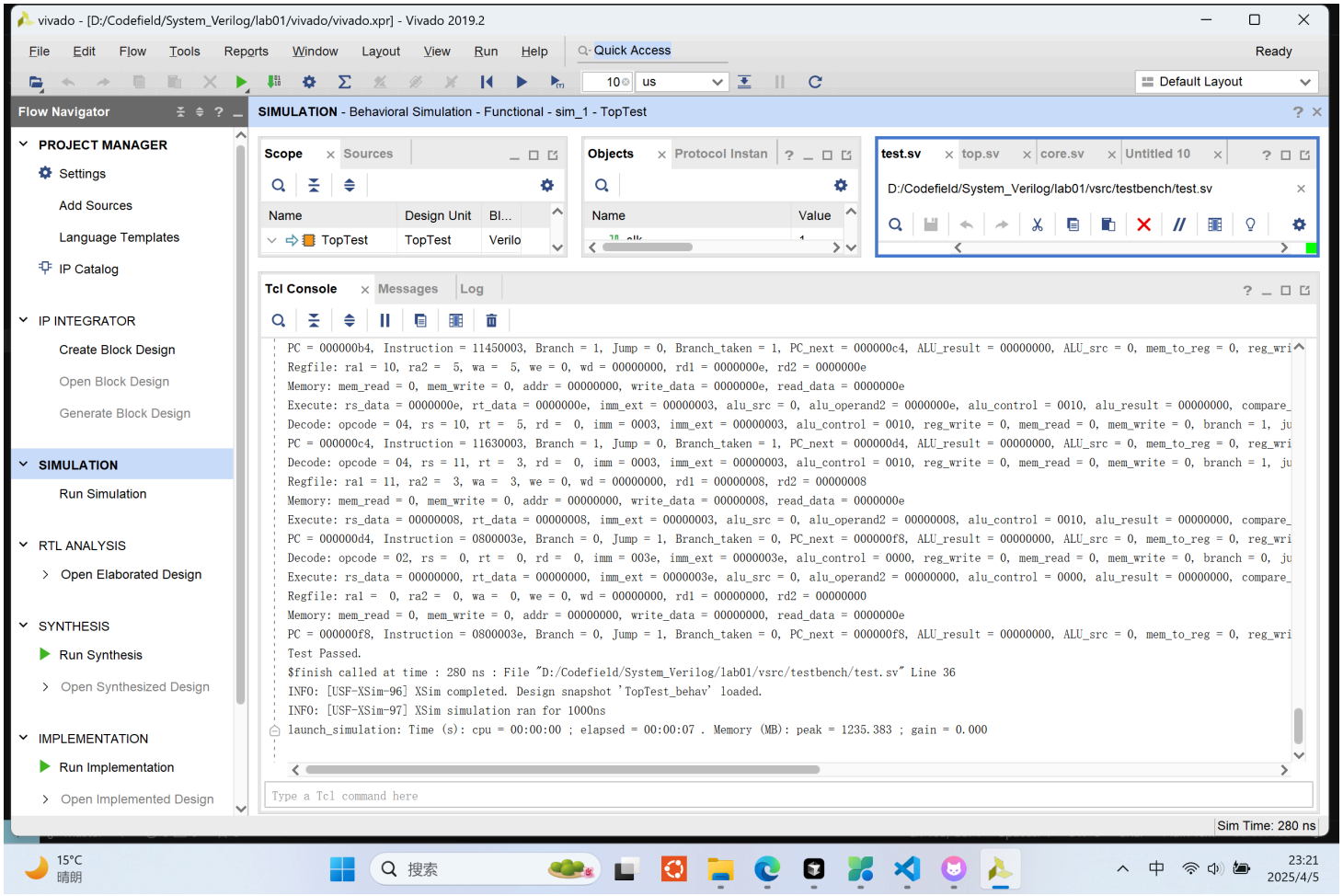
本实验实现的是一个基于MIPS指令集的单周期CPU，具有如下指令：
运算指令：add sub and or slt addi；访存指令：lw sw；分支指令：j beq；共10条。
实验的源代码仓库位于Arch-2025Spring的lab1分支。

2.电路图



3.测试通过截图

在添加了许多的调试语句后，测试通过截图如下，Test Passed信息出现在倒数第五行



4.设计思路

首先是根据上一次的lab00当中实现的alu，基本照搬就可以实现本次的alu模块，实现的唯一区别就是本次运算指令需要数量不多，所以将上次的alu中的控制信号由5位改为4位，更节省空间。

首先先看一下已有的top 和 ram模块，ram当中已经实现了指令和数据内存的模块imem, dmem

接着实现regfile的写使能，搭建好里面的寄存器，rt，rs，确定好输入输出如下，ra1,ra2是读地址，对应寄存器值rd1,rd2
写地址wa，写使能信号we，写入数据wd

```
input logic clk, reset,
        // read ports
        input creg_addr_t ra1, ra2,
        output word_t rd1, rd2,
        // write port
        input creg_addr_t wa,
        input word_t wd,
        input logic we
```

如果读地址为0，输出0，MIPS的 $zero 寄存器始终为0
同样写操作wa 为0时，应该直接忽略写操作
regfile最后是我为了调试时加入的打印信息

```
initial begin
        $monitor("Regfile: ra1 = %d, ra2 = %d, wa = %d, we = %b, wd = %h, rd1 = %h, rd2 = %h",
                 ra1, ra2, wa, we, wd, rd1, rd2);
    end
```

然后实现的是decode fetch 和 execute模块
fetch实现取指令，根据pc从imem当中读取指令，pc的更新写在core模块当中

```
always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        pc <= 32'b0;  // 复位时 PC 设为 0
    end else begin
        pc <= pc_next;  // 每个时钟上升沿更新 PC
    end
end


always_comb begin
    if (jump) begin
        pc_next = jump_addr;  // 跳转指令
    end else if (branch_taken) begin
        pc_next = pc + 4 + (imm_ext << 2);  // 分支指令
    end else begin
        pc_next = pc + 4;  // 顺序执行
    end
end
```

工作流程：
在每个时钟上升沿，pc 寄存器会更新为 pc_next 的值
pc_next 的值由组合逻辑根据当前指令类型（跳转、分支或顺序执行）计算得出

然后就是decode模块，把指令对应的位数分好，然后按照opcode的case进行分类，最后链接regfile,同时decode指令应该对输入指令进行分类，尤其是跳转指令beq和jump,这里采用了一个branch和jump信号来判断是否需要跳转

```
6'b000100: begin // BEQ
                branch = 1'b1;
                alu_control = 4'b0010; // SUB
            end
            6'b000010: begin // J
                jump = 1'b1;
                jump_addr = {instruction[31:28], instruction[25:0], 2'b00};
            end
```

然后是execute模块，execute模块其实只要把参数准备好然后传递给alu就可以，输入输出接口如下

```
input word_t rs_data,
        input word_t rt_data,
        input word_t imm_ext,
        input logic [3:0] alu_control,
        input logic alu_src,
        input logic branch,
        output word_t alu_result,
        output logic zero,
        output logic branch_taken
```

execute当中关于分支的处理如下

```
// ALU 操作数
    word_t alu_operand2;
    assign alu_operand2 = alu_src ? imm_ext : rt_data;

    // 实例化 ALU
    alu alu_inst (
        .srca(rs_data),
        .srcb(alu_operand2),
        .alufunc(alu_control),
        .result(alu_result),
        .zero(zero)
    );

    // 分支判断
    logic compare_result;
    assign compare_result = (rs_data == rt_data);
    assign branch_taken = branch && compare_result;
```

最后就是writeback模块，把计算的数据写回寄存器即可。
在实现以上模块之后，在core模块当中将这些模块实例化并且连接起来就可以。

5.实验总结

本次实验花了不少时间hhh，但是还是把以前没有搞懂的cpu现在明白很多了。代码编写方面有点复杂的就是各种模块的接口还有内部的信号要设计的比较好，不然最后链接的时候很容易冗余或者少接口，这些问题还是要在调试当中一步步发现。
本次实验学会了怎么写system verilog调试的打印测试语句，在帮助我修改jump和branch逻辑的时候有很大的用处。