实验五：Cache 的实现与 Cache 的设计探索

在前四个 Lab 中，我们假设指令存储与数据存储在 1 个周期内就能获取我们需要的指令与数据，但正如课上所讲，实际的硬件中，访存延迟比 CPU 的逻辑电路延迟高很多。为了缓解这个问题，CPU 的设计中往往会引入 Cache 结构，以利用程序运行时的时间、空间局部性缓解访存延迟。

本次实验中，我们将进一步拓展 Lab4 中实现的流水线处理器，为它添加一级简易的数据 cache，并将包含 Cache 的处理器连接到一个慢速的主存模型上。在此基础上，我们将进一步探索不同的 Cache 设计方案对 Cache 的性能影响。

1 实验目的

i. 掌握 Cache 基于状态机的运行机制；

ii. 掌握 Cache 不同替换策略的运行机制；

iii. 体验 Cache 的不同设计点对 Cache 性能带来的综合影响。

2 实验环境（推荐）

与实验一相同：

i. IDE: vscode

ii. verilog compiler: iverilog

iii. waveform viewer: GTKwave

3 实验任务

如图1-(a) 所示，本次实验要求把 Lab4 实现的处理器中的数据存储替换为 Cache 模块，并将 Cache 模块连接到我们给定的慢速主存模型上。大家需要完成以下三项任务：

(1) 实现基于 FIFO（先进先出）替换策略的 Cache。

(2) 为 Cache 拓展 LRU 替换策略。

(3) 在给定 Cache 存储容量和 Cache Line 大小的前提下，调节 Cache 的组数和路数，测试在不同场景下的 Cache 命中率。

注：本次实验不要求大家实现指令 *Cache*，大家仍然可以假设指令在当前周期内就能取出。

3.1 Cache 的实现思路

3.1.1 概览

本次实验中，大家首先需要完成一个使用写回 + 写分配策略的数据 Cache，并且实现FIFO 替换策略。在这种策略下，当读写命中时，Cache 直接从目标地址对应的 Cache Line（Block）读写数据。对于写命中，Cache 同时需要把对应的 Cache 行标记为脏行。而在发生读写缺失时，如果目标地址对

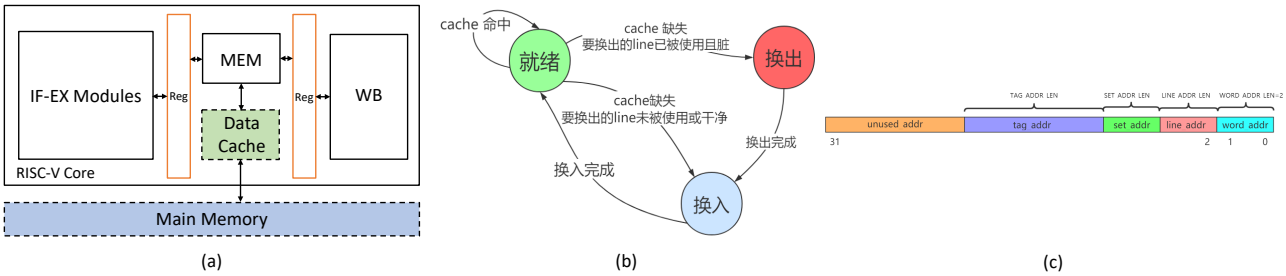

图 1: 实验概览。(a) 模块组织概览 (b) Cache 状态机 (c) Cache 编址规则

应的 Cache 组内仍然有无效的 Cache Line，目标地址所在的行会被加载至该无效行，并更新行的状态为有效。如果 Cache 组所有的行均有效，Cache 会选取当前组内最早被加载进 Cache 的行做替换。对于写缺失，Cache 同时需要把换出的行写入对应地址的主存存储中。

为了实现上述功能，Cache 中需要维护如图1-(b) 所示的状态机。当初始化/没有读写请求时，Cache 保持就绪状态。当接收到读/写请求时，Cache 检查当前请求是否命中。如果命中，Cache 仍保持在就绪状态，并在读写请求发送后的下个周期响应请求。如果缺失，则进行行换入（换入前可能需要进行换出）。在 Cache 进行换出换入时，Cache 从就绪状态跳转到相应状态，并且无法响应 CPU 当前的读写请求。换出换入执行完毕后，Cache 重新回到就绪状态，并重新响应读写请求。无论是命中还是缺失，在 Cache 返回数据的同时，Cache 需要向 CPU 发送请求完成的信号。CPU 在接收到请求完成信号之前，需要控制整个流水线进行 stall。因此，大家需要修改 Lab4 中添加的 hazard_detect_unit。

Cache 的编址规则如图1-(c) 所示。最低 2 位为 word_addr，指代 word 内的字节地址。line_addr 指代 Cache Line 内的地址偏移，以 word 为单位。set_addr 指代目标地址所在的 Cache Set 编号，它负责将读写请求路由到正确的组。tag_addr 指代地址的 Cache Tag。发生读写请求时，Cache 将 tag_addr 域的值与对应 Cache Set 内的所有 Cache Tag 作比较，如果相等则命中，不等则缺失。

3.1.2 代码分析

为了降低大家的难度，我们为大家提供了 Cache 的模块定义、Cache 的（主要）成员变量，以及状态机的时序逻辑框架。详细描述如下所示：

Cache 的模块定义如下方代码块所示。其中，地址中的有效字段位宽（line_addr 位宽、set_addr 位宽、tag_addr 位宽）以及每个 Cache Set 的路数被参数化至模块定义中，大家需要以参数化的方式实现 Cache 的功能。在整个 Lab5 中，我们规定 Cache 的总存储容量为 1KB，Cache Line 的长度为 64Byte。此外，代码中的参数是我们在评测大家实现的正确性时配置的参数，请大家在提交代码到平台时维持这些参数不变。

Cache 包含的端口可以分为三部分：(1) 时钟端口、复位端口，以及一个用于打印 debug 输出的端口。(2) 与 CPU 连接的端口。其中，输入端口包含 CPU 读写请求端口、写类型端口、地址端口、写入数据端口；输出端口包含提示 CPU 读写请求完成的端口、读数据端口。(3) 与主存连接的端口。其中，输入端口包含主存请求完成端口、主存读数据端口；输出端口包含主存读写请求端口、主存地址端口、主存写数据端口。请注意，Cache 与主存之间读写数据的交互粒度为 Cache Line 的长度。因此，大家向主存送入的地址应该是 Cache Line 在主存中对应的起始地址。此外，由于 Verilog 不支持使用数组定义接口，我们还为大家提供了将主存的读写数据接口转换成以 word 为单位的数组的解包/打包代码，大家可以参考这些代码来实现参数化的设计。

```
module DataCache #(
    parameter LINE_ADDR_LEN = 3, // Each cache line has 2^LINE_ADDR_LEN words
    parameter SET_ADDR_LEN = 3, // This cache has 2^SET_ADDR_LEN cache sets
    parameter TAG_ADDR_LEN = 10, // should in alignment with main memory's space
    parameter WAY_CNT = 4 // each cache set contains WAY_CNT cache lines
) (
    input clk, rst, debug,
    // ports between cache and CPU
    input read_request, write_request,
    input [2:0] write_type,
    input [31:0] addr, write_data,
    output reg request_finish,
    output reg [31:0] read_data,
    // ports between cache and main memory
    output mem_read_request, mem_write_request,
    output [(32*(1<<LINE_ADDR_LEN)-1):0] mem_write_data,
    output [31:0] mem_addr,
    input mem_request_finish,
    input [(32*(1<<LINE_ADDR_LEN)-1):0] mem_read_data
);

    // Used for memory read/write ports' unpack/pack
    reg [31:0] mem_write_line [0:LINE_SIZE-1];
    wire [31:0] mem_read_line [0:LINE_SIZE-1];
    genvar line;
    generate
        for(line=0; line<LINE_SIZE; line=line+1)
        begin : memory_interface
            assign mem_write_data[32*(LINE_SIZE-line)-1:32*(LINE_SIZE-line-1)] = mem_write_line[line];
            assign mem_read_line[line] = mem_read_data[32*(LINE_SIZE-line)-1:32*(LINE_SIZE-line-1)];
        end
    endgenerate

    /****************************************
     *                                      *
     *           Other Codes                *
     *                                      *
     ****************************************/

endmodule
```

Cache 中包含的成员变量和局部参数如下方代码块所示。其中，localparam 对参数中的位宽做进一步处理。三个 parameter 为 Cache 状态机中的三个状态，依次为就绪、换出、换入。Cache 中所有的存储单元使用寄存器实现，四个变量依次对应 Cache Line、Cache Tag、Cache Line 的有效位、Cache Line 的脏位。cache_state 用于记录 Cache 状态机当前所处的状态。replace_way 用于记录 Cache 替换策略接下来需要替换的 Cache Line。请注意，提供的变量只是为了实现 Cache 的所有功能至少需要声明的基本变量。大家在实现的过程中可能需要自行定义其他的状态变量。

本阶段中我们只要求大家实现FIFO 替换策略，一种简单的实现思路是：按照 Cache 数组的索引范围从小到大循环放置数据，这样每次替换数据的位置就是当前索引的下一个 Cache Line，并且我们可以通过取模的方式来实现索引的循环遍历。

注：这里用寄存器声明 Cache 存储单元只是一种简化表示。实际中的存储单元会使用 Memory Compiler 等手段生成。

```
// params to transfer bit number to count
localparam WORD_ADDR_LEN = 2; // each word contains 4 bytes
localparam MEM_ADDR_LEN = TAG_ADDR_LEN + SET_ADDR_LEN; // in cache line's granularity
localparam UNUSED_ADDR_LEN = 32 - MEM_ADDR_LEN - LINE_ADDR_LEN - WORD_ADDR_LEN;
localparam LINE_SIZE = 1 << LINE_ADDR_LEN; // each cache line has LINE_SIZE words
localparam SET_SIZE = 1 << SET_ADDR_LEN; // This cache has SET_SIZE cache sets
// cache state enumarations
parameter [1:0] READY = 2'b00;
parameter [1:0] REPLACE_OUT = 2'b01;
parameter [1:0] REPLACE_IN = 2'b10;

// cache units declaration
reg [31:0]          cache_data [0:SET_SIZE-1][0:WAY_CNT-1][0:LINE_SIZE-1];
reg [TAG_ADDR_LEN-1:0] tag [0:SET_SIZE-1][0:WAY_CNT-1];
reg                 valid [0:SET_SIZE-1][0:WAY_CNT-1];
reg                 dirty [0:SET_SIZE-1][0:WAY_CNT-1];

// current cache state
reg [1:0] cache_state;

// for replace policy, basically, we will implement FIFO policy
// For simplicity, we can assign cache lines from way 0 to way WAY_CNT-1
// In this way, FIFO is equivalant to round-robbin policy
reg [31:0] replace_way [0:SET_SIZE-1];
```

Cache 中的状态机时序逻辑框架如下方代码块所示。如果 rst 信号为高电位，Cache 把所有寄存器进行初始化。否则，Cache 在三种状态之间进行切换，并在每种状态下执行对应的 Cache 操作。

```
always @(posedge clk) begin
    if(rst) begin
        // reset all registers
    end
    else begin
        case (cache_state)
            READY: begin
                if(hit) begin
                    // 1. notify CPU whether the request can be finished
                    // 2. update cache data
                end
                else begin
                    // if current request does not hit, change cache state
                end
            end
```

```
            REPLACE_OUT: begin
                // switch to REPLACE_IN when memory write finishes
            end
            REPLACE_IN: begin
                // When memory read finishes, set cache line's data & state, then swtich to READY
            end
        endcase
    end
end
```

此外，我们还为大家提供了 debug 用的代码。大家只要将 debug 信号置为 1，就可以打印整个 Cache 的信息。这部分代码生成的输出文件也会作为评判 Cache 实现正确性的标准之一，所以请大家在提交时将这部分代码保留在提交文件中。大家也可以利用 debug 信号设计自己需要的调试代码。

3.1.3 实现说明

在提供的代码的基础上，大家需要完成的功能如下：

1. 用于地址翻译的组合逻辑。大家需要把输入的地址分解为图1-(c) 中的各个字段，并记录每个字段的值，供后续 Cache 操作使用。
2. 判断 Cache 是否命中的组合逻辑。大家需要根据 set_addr 的值检索对应 Cache Set 的 Cache Tags，并与 tag_addr 的值作比较，以判断当前请求是否命中。
3. 发送内存请求的组合逻辑。当 Cache Miss/Replacement 发生时，Cache 需要向主存发送访存请求，以将脏数据写回内存，并且获取所需数据。
4. 状态机中各状态下的 Cache 处理逻辑。(1) 当 Cache 处于就绪状态时，如果当前请求命中，Cache 处理请求，并在下个周期向 CPU 发送请求完成信号和读请求所需的数据；如果当前请求缺失，Cache 随即向主存发送请求，并根据替换行是否需要写回跳转至相应状态。(2) 当 Cache 处于换出状态时，Cache 等待主存响应写请求，主存响应后的下个周期跳转至换入状态。(3) 当 Cache 处于换入状态时，Cache 等待主存响应读请求，主存响应时，Cache 把数据写入对应的 Cache Line，设置该 Cache Line 的状态，并在下个周期内回到就绪状态。请注意，Cache 不位于就绪状态时，不应该响应任何 CPU 请求。此时 CPU 的流水线暂停，并且 CPU 会持续向 Cache 发送相同的内存请求信号。

注：对于包含参数的模块，大家可以使用 generate 和 for 循环等语句来简化代码的书写复杂度，这两种语句的语法可以参考：Verilog 初级教程 (12) Verilog 中的 generate 块和 Verilog 初级教程 (16) Verilog 中的控制块。

3.1.4 时序说明

为了便于大家理解 Cache 的处理流程，如图2所示，我们以写命中和读缺失时的时序为例，为大家展示 Cache 对 CPU 提供的时序抽象。请注意，无论是何种情况，在 Cache 向 CPU 发送 request_finish 信号之前，CPU 需要暂停整个流水线，以维持向 Cache 发送的请求信号。

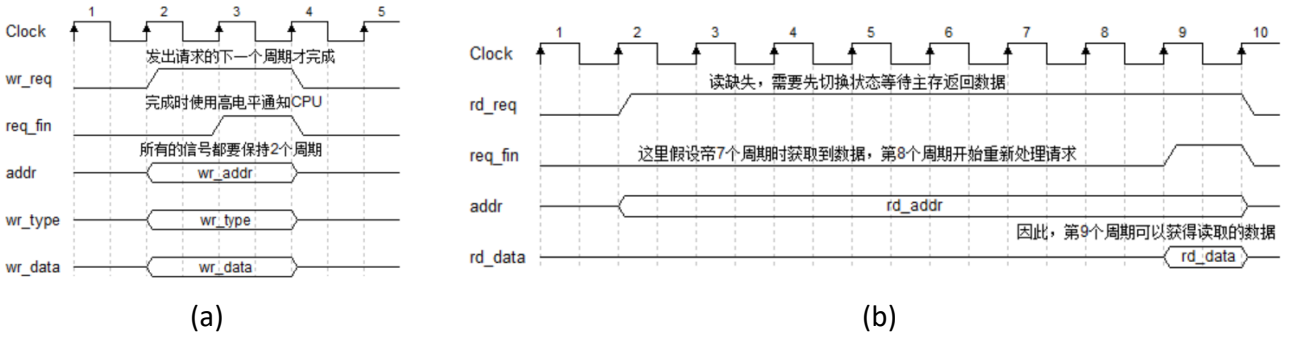

图 2: Cache 对 CPU 提供的时序示例。(a) 写命中 (b) 读缺失

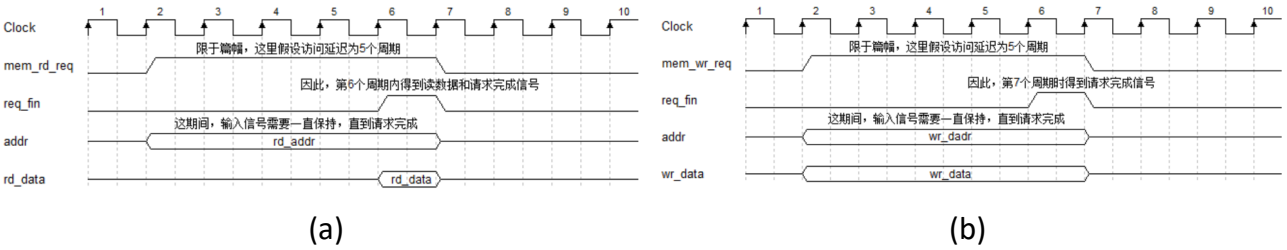

图 3: 主存模型对外提供的时序示例。(a) 读操作 (b) 写操作

当写命中时，Cache 会在发出请求后的下个周期向 CPU 响应完成信号。注意到我们实现的是写回 + 写分配策略的 Cache，所以写请求也能在 Cache 内用较短的时间完成，只是我们需要额外维护 Cache 的脏状态。

当读缺失时，Cache 需要先切换状态向主存发送请求，在从主存获取到数据后，Cache 在下一个周期重新处理请求，并在再下一个周期返回请求完成的信号与相应数据。在图里的示例中，我们假设 Cache 在第 7 个周期获取到数据，并在第 8 个周期内开始处理请求。因此，CPU 在第 9 个周期内接收到请求完成的信号。请注意，读写缺失时的等待时间会受是否需要把脏行写回主存而变化。

注：读命中和写缺失时的时序与上述两种情况分别相似，只是使用的信号有变化。

3.2 主存模型的对外抽象

为了模拟真实硬件中内存访问的高延迟，我们为大家提供了一个慢速的主存模型。这个主存模型用于存放测试程序的数据，因此大家需要把这个主存模型连接到 Cache 上。主存模型包含 main_memory_wrapper.v 和 ram.v 两个文件。其中，顶层文件为 main_memory_wrapper.v。它包裹了使用寄存器数组实现的 Data RAM，并且以 Cache Line 为粒度读写数据。它接收的地址以字节为单位，并且应该是 Cache Line 字节长度的倍数。Data RAM 的存储容量为 2^{18} 字节，因此主存接收的地址为 18 位，这一长度应与 tag_addr+set_addr+line_addr+word_addr 的位数之和相等。

主存模型对外提供的读写时序示例如图3所示。假设主存模块在第 i 个周期接收读写请求，那么它会在第 $i+20$ 个周期向上层模块（Cache）发送请求完成信号和所需数据（读请求），并在第 $i+21$ 个周期可以响应后续请求。请注意，在等待响应的 20 个周期内，上层模块需要维持传输给主存接口模块的信号。大家不需要对此模块做任何修改，只需要了解其时序行为。

主存模块对外接口如下代码块所示。主存模块包含以下几类接口：(1) 读写请求接口 (2) 读写数

据接口（3）请求完成接口（4）地址接口。可以看到，我们提供的 Cache 与主存连接的接口和下述接口定义是一一对应的。

```
module MainMemoryWrapper #(
    parameter LINE_ADDR_LEN = 3,
    parameter ADDR_LEN = 13
) (
    input clk, rst, debug,
    input read_request, write_request,
    input [(32*(1<<LINE_ADDR_LEN)-1):0] write_data,
    input [31:0] addr,
    output reg request_finish,
    output [(32*(1<<LINE_ADDR_LEN)-1):0] read_data
);
```

3.3 LRU 替换策略的实现思路

LRU（最近最少使用）替换策略是目前非常常见，也非常高效的 Cache 替换策略。当目前访问的 Cache 组的所有行均填充了有效数据时，LRU 策略会将当前组内最后一次被使用的 Cache 行替换出去。本次实验的第二个任务就是让大家把 FIFO 策略替换为 LRU 策略。为了实现 LRU 策略，大家需要在去掉 FIFO 替换逻辑的基础上，额外实现如下逻辑：

1. 添加成员变量，为每个 Cache Set 的每一路记录 Age，代表上次访问的次数间隔。
2. 添加组合逻辑，根据输入的地址所在的 Cache Set，筛选出对应 Set 内 Age 最大的那一路。
3. 添加状态机逻辑。当 Cache 命中时，更新对应组内所有路的 Age，并且在更新完毕后记录 Age 最大的路，作为下次替换时被换出的路。

为了便于大家的设计和调试，我们在 tb_cache.v 里提供了一个名为 *LRU* 的宏定义。大家可以通过使用条件编译的方式，根据这个宏定义的存在与否来选择自己使用的 Cache 替换策略，而不需要在书写代码时注释或者删除未被选择的替换策略的逻辑。条件编译的使用可以参考 Verilog 初级教程（20）Verilog 中的 'ifdef 条件编译语句。我们也在代码中给出了注释和一些条件编译语句块作为提示。

3.4 Cache 的设计探索

如课件所述，Cache 的性能会受到 Cache 的组织模式的影响。本次实验的第三项任务就是让大家在 Cache 总容量以及 Cache Line 的容量固定的情况下，调整 Cache 的组数与路数，并在不同的替换策略下运行不同的应用程序，观察 Cache 在各种场景下的 Miss 情况。具体来说，大家需要在 WAY_CNT 与 SET_SIZE（$2^{SET\_ADDR\_LEN}$）乘积（Cache 中的总路数）不变的前提下，调整 SET_ADDR_LEN 与 WAY_CNT（Cache Line 的组织形式），并在 FIFO 和 LRU 策略下分别记录运行矩阵乘法和快速排序程序时的 Cache Miss 情况。

注：在调整 Cache 组织时，实际需要的电路开销也会随之发生变化。但由于我们不做 EDA 流程，所以本次实验中我们只关注 Cache 的组织对性能（命中率）的影响。

为了统计 Cache Miss，大家需要在 Cache 内增加两个用于计数的寄存器，分别记录命中与缺失的 Cache 请求次数。此外，由于内存地址长度不变，大家在对 Cache 组织格式做调整时，可能也需要调整其他部分的位宽（如 tag_addr）。

3.5 模块组织

由于大家的 CPU 实现方式可能与我们在 Lab4 中给出的组织方式有不同，因此本次实验中我们只为大家提供主存相关代码（main_memopry_wrapper.v 和 ram.v）、指令存储相关代码（instr_mem.v）、顶层模块相关代码（riscv_top.v），以及 Cache 模块相关代码（data_cache.v）。请大家在维持顶层模块的接口定义不变的前提下进行进一步的修改与整合，修改顶层模块的接口可能导致无法取得成绩。

3.6 信号设计

本次实验中，大家只需要在 data_cache.v 内部及大家在 Lab4 中实现的模块内部进行模块设计，请大家根据自己的实现与前述 data_cache.v 的接口定义，自行添加与修改信号。

4 测试与评分

请在本地调试完毕后，再将代码提交到测试平台，并查看测试平台中显示的分数。

4.1 tb 验证思路

本次实验中，我们使用 tb 文件进行 Cache 实现的正确性测试。我们为大家提供了 4 份测试代码，其中 2 份为（伪）矩阵乘法（matmul），2 份为快速排序（quick_sort）。每类代码中，一份在执行完毕后冲刷 Cache（使用未访问过的地址对 Cache 进行访问），一份则保留了 Cache 中缓存的数据。我们对这 4 份代码分别在 FIFO 和 LRU 策略下进行测试，比对程序完成后主存和 Cache 中的数据，能全部匹配即可获得正确性测试的满分。（共八个测试，每个测试对比 mem 和 cache 的数据，两个文件数据都正确获得 12.5 分。）

此外，我们也为大家提供了一份简单的测试代码，用于测试 Cache 的功能性。该代码不纳入评分中，大家 Debug 时可以使用这份测试代码做简单的功能验证。

4.2 重要说明

如前所述，我们在测试平台上进行测试时，会固定 Cache 的参数，因此请大家将第三部分的任务在实验报告中通过图表的形式展示出来。举例来说，大家可以为两个应用各画一幅折线图，折线图横坐标为每组的路数，纵坐标为命中率（缺失率），图上使用两条折线来分别代表 FIFO 和 LRU 替换策略。此外，在第三部分中，我们使用不做冲刷的两份代码进行测试。

4.3 平台使用

1. 课程平台地址：微处理器设计与智能芯片课程实践
2. 请按规定字段进行注册，并牢记自己的密码。

3. 从测试平台上下载的代码包包括模板文件和简化的测试文件，请先仔细阅读 README，其中提到了你至少应该上传哪些文件，可以自行定义其他辅助模块一并压缩上传。
4. 测试平台会用自己的测试用例覆盖提供给同学们的用例，可以不再上传与测试相关的文件。
5. 请珍惜实验室服务器资源，切勿恶意提交代码。
6. 与平台、实验有关的任何问题可以与助教联系。
7. 欢迎大家提出有关平台、实验的各种建议，让本课程和配套实习越来越好。

4.4 评分规则

本次共有八个测试，分别对应标号为 0、1、2、3 的两份矩阵乘法和两份快速排序代码在 FIFO 策略和 LRU 策略下的运行。每个测试对比 mem 和 cache 的数据，两个文件数据都正确获得 12.5 分，共 100 分。

注：平台评分主要作为对你完成度的提示，并不等于你最终课程实习部分的分数。

4.5 实验报告

在实验的同时和实验之后撰写实验报告是非常良好的习惯。这不仅有助于提高实验效率，理清实验思路，帮助 debug，也有助于记录实验过程和结果，还可以和其他同学交流和分享。因此，我们需要大家在 lab 完成过后为我们提交一个实验报告，实验报告有以下几点注意事项：

1. 包含你的设计思路，可以包括相应的模块组织层次、一些不同于助教提示的地方等。
2. 你 debug 的过程，或崩溃或大起大落或柳暗花明的心路历程等。
3. 最终用于 Lab 评分的那次提交：包括代码包、分数截图、提交时平台为本次提交生成的 ID。
4. 实验报告不做字数要求，体现思考和实验过程即可（反卷第一名）。即使最后有 bug 没解决，没有以满分通过平台测试，讲一讲你的 debug 过程与思考也会获得一些分数的补偿：）

5 提示与帮助

会在课程网站上动态更新，大家遇到问题先去网站的 Q&A 板块看一下。