Verilog 期末设计实验报告 —七路裁判打分电路

邢清画 2211999 网络空间安全学院

一、 实验名称：七路裁判打分电路

实验描述：7 个裁判各自在不同时刻打分，满分 15 分，输出平均整数得分，从第三个裁判给出分数开始，计算平均分时要去掉一个最高分和一个最低分。

二、 实验目的

本实验旨在通过设计和实现一个数字电路模块，深化对 Verilog 编程语言的理解，特别是其数据类型、模块结构和控制流语句。此外，实验强调时序逻辑设计的重要性，目的是理解时钟信号在同步数字电路中的作用，并应用这些知识来设计有效的时序电路。

实验还包括对数据处理和算法实现的实践，设计电路以处理和计算 7 个裁判的打分数据，这涉及数据存储、排序、去除极值和计算平均值等功能，从而增强对硬件中复杂算法实现的理解。

熟悉数字电路设计工具 Vivado 的使用，包括从编写代码到仿真验证再到电路综合的整个流程，以提高对 FPGA 编程和设计流程的理解。

最后，实验旨在培养解决电路设计中遇到的问题的能力，例如处理复位逻辑、解决时序问题和优化资源使用等，并将理论知识与实际操作相结合，以加深对数字电路设计理论的理解和应用。

三、 设计思路

实现一个 Verilog 模块 ScoreProcessor，用于计算七位裁判逐个打分的实时平均分。该模块能够在每次收到新的分数时更新平均分，当收到三个及以上分数时，去除一个最高分和一个最低分后计算平均分。该模块还通过一个清零信号(rst)重置，以便开始处理新一轮的分数。

3.1 输入输出定义

输入:
clk: 时钟信号，用于同步分数处理。rst: 异步清零信号，用于重置模块状态。score: 当前裁判的分数。score_valid: 标识当前分数是否有效。
输出:
averageScore: 计算出的平均分数。

3.2 分数处理逻辑

分数存储：使用一个寄存器数组 scores[6:0] 存储每位裁判的分数。
有效分数计数：使用 num_scores 记录已经打分的裁判数量。
分数更新：在收到有效的分数（score_valid 为高）时，更新 scores 数组，并增加

num_scores。

平均分计算：

当 num_scores 少于 3 时，直接计算所有分数的平均值。

当 num_scores 大于等于 3 时，首先找出最高分和最低分，然后计算去除这两个分数后剩余分数的平均值。

3.3 清零和重置逻辑

在 rst 信号为高时，重置 num_scores、averageScore 以及 scores 数组，以便模块能够开始处理新一轮的分数。

3.4 Testbench 设计

时钟信号生成：生成一个测试时钟信号，模拟实际硬件中的时钟。

多轮打分模拟：模拟多轮裁判打分过程，每轮包括 7 个分数的输入，每输入一个分数后观察平均分的变化。

清零信号模拟：在每轮打分结束后发出清零信号，重置 ScoreProcessor，然后开始下一轮打分。

结果监视：监视并打印每次分数输入和平均分的变化。

四、 实验代码

ScoreProcessor 模块代码

```
module ScoreProcessor(
    input clk,  // 时钟信号
    input rst,  // 清零信号
    input [3:0] score,  // 当前裁判的打分
    input score_valid,  // 当前分数有效信号
    output reg [3:0] averageScore  // 平均分数输出
);
    reg [3:0] scores[6:0]; // 存储裁判的分数
    reg [2:0] num_scores = 0; // 记录已经打分的裁判数量
    integer i, sum, max, min;

    // 重置逻辑
    always @(posedge rst) begin
        if (rst) begin
            num_scores <= 0;
            averageScore <= 0;
            for (i = 0; i < 7; i = i + 1) begin
                scores[i] <= 0;
            end
        end
    end

    // 时钟控制的主要逻辑
    always @(posedge clk) begin
        if (score_valid) begin
            scores[num_scores] <= score;
```

```
            num_scores <= num_scores + 1;
        end

        // 重置计算用的变量
        sum = 0; max = 0; min = 15;

        // 计算总和以及最大和最小值
        for (i = 0; i < num_scores; i = i + 1) begin
            sum = sum + scores[i];
            if (scores[i] > max) max = scores[i];
            if (scores[i] < min) min = scores[i];
        end

        // 计算平均分
        if (num_scores >= 3) begin
            // 当有三个或更多的分数时，去掉一个最高分和一个最低分计算平均值
            averageScore <= (sum - max - min) / (num_scores - 2);
        end else if (num_scores > 0) begin
            // 当少于三个分数时，直接计算所有分数的平均值
            averageScore <= sum / num_scores;
        end
    end
endmodule
```

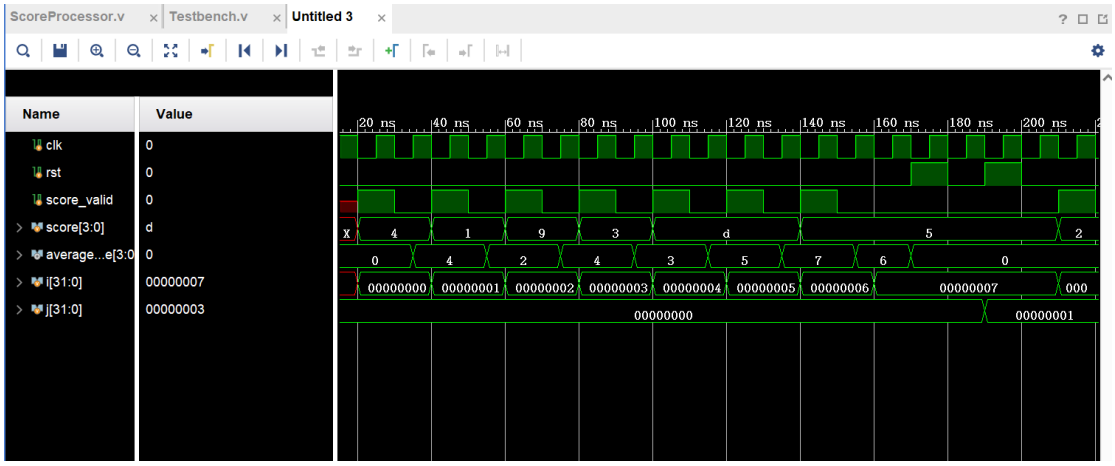
Testbench 代码

```
timescale 1ns / 1ps

module Testbench;
    reg clk, rst, score_valid;
    reg [3:0] score;
    wire [3:0] averageScore;
    integer i, j;

    // ScoreProcessor 模块实例化
    ScoreProcessor sp(clk, rst, score, score_valid, averageScore);

    // 生成时钟信号
    initial clk = 0;
    always #5 clk = ~clk;

    // 测试逻辑
    initial begin
        // 模拟多组打分
        for (j = 0; j < 5; j = j + 1) begin
            rst = 1; #10;  // 发出清零信号
            rst = 0; #10;  // 结束清零
            score_valid = 0;  // 确保开始时分数无效

            // 逐个输入裁判的分数
            for (i = 0; i < 7; i = i + 1) begin
                score = $random % 16;
                score_valid = 1; #10;
                score_valid = 0; #10;
```

```
            end

            // 等待一个额外的时钟周期，确保最后一个分数的平均值计算完成
            #10;

            rst = 1; #10;  // 重置
            rst = 0; #10;  // 准备下一组打分
        end
        $finish;
    end

    // 监控和打印
    initial begin
        $monitor("Time = %0t: Score = %d, Average = %d", $time, score, averageScore);
    end
endmodule
```

五、 实验结果

波形图展示：



如图所示，第一至第七位裁判打分依次为：4,1,9,3,13,13,5

第一位裁判打完分后只有一个分数，所以平均分为 4；

第二位裁判打完分之后，平均分为（4+1）/2=2.5，向下取整为 2；

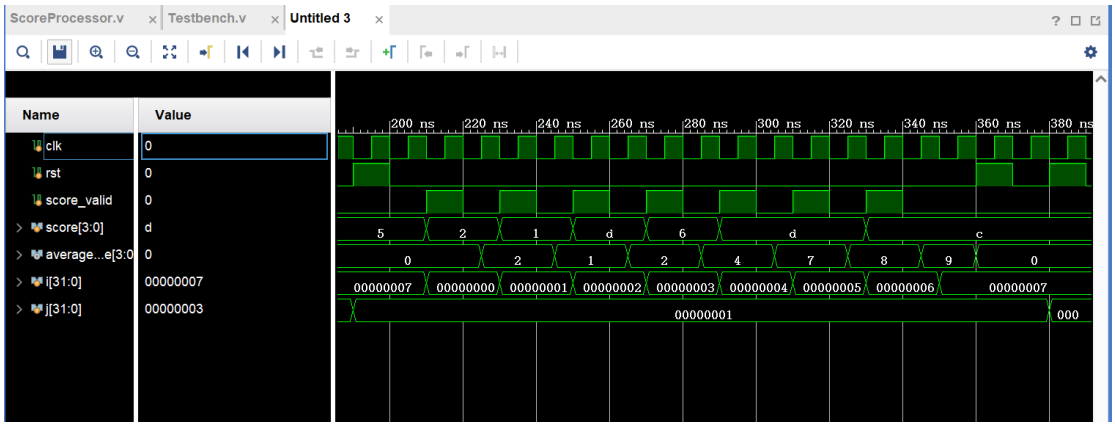
第三位裁判打分后，去掉目前最高分 9 和最低分 1，平均分为 4；

第四位裁判打分后，去掉目前最高分 9 和最低分 1，平均分为（4+3）/2=3.5 向下取整为 3；

第五位裁判打分后，去掉目前最高分 13 和最低分 1，平均分为（4+3+9）/3=（向下取整）5；

第六位裁判打分后，去掉目前最高分 13 和最低分 1，平均分为（4+3+9+13）/4=（向下取整）7；

第七位裁判打分后，去掉目前最高分 13 和最低分 1，平均分为（4+3+9+13+5）/4=（向下取整）6 ；经检验，答案正确。



如图所示，第一至第七位裁判打分依次为：2,1,13,6,13,13,12

第一位裁判打完分后只有一个分数，所以平均分为 2；

第二位裁判打完分之后，平均分为（2+1）/2=1.5，向下取整为 1；

第三位裁判打分后，去掉目前最高分 13 和最低分 1，平均分为 2；

第四位裁判打分后，去掉目前最高分 13 和最低分 1，平均分为（2+6）/2= 4；

第五位裁判打分后，去掉目前最高分 13 和最低分 1，平均分为（2+6+13）/3=（向下取整）7；

第六位裁判打分后，去掉目前最高分 13 和最低分 1，平均分为（2+6+13+13）/4=（向下取整）8；

第七位裁判打分后，去掉目前最高分 13 和最低分 1，平均分为（2+6+13+13+12）/4=（向下取整）9 ；经检验，答案正确。

六、 思考与总结

6.1 其他因素考虑

6.1.2 有裁判弃票

在模拟实际投票过程中，可能存在裁判弃票的情况，这时需要对原有的 ScoreProcessor 模块和测试逻辑进行修改。这里的关键是要区分有效的打分和弃票（例如，可以用一个特定的分数表示弃票）。假设使用 score 值为 16 来表示弃票（因为正常分数的范围是 0 到 15），修改代码来忽略这些弃票。

1. **跳过弃票的处理**：在收到新分数时，检查分数是否为弃票标志，如果是，则不将其计入 scores 数组。
2. **更新计算逻辑**：在计算平均分时，只考虑有效的分数。

修改后的模块的关键部分如下：

```
// 时钟控制的主要逻辑
always @(posedge clk) begin
    if (score_valid && score != 16) begin  // 检查分数是否为弃票标志
        scores[num_scores] <= score;
        num_scores <= num_scores + 1;
    end

    // 重置计算用的变量
    sum = 0; max = 0; min = 15;
    integer valid_scores = 0;  // 有效分数的数量

    // 计算总和以及最大和最小值
    for (i = 0; i < num_scores; i = i + 1) begin
        if (scores[i] != 16) begin  // 忽略弃票
            sum = sum + scores[i];
            if (scores[i] > max) max = scores[i];
            if (scores[i] < min) min = scores[i];
            valid_scores = valid_scores + 1;
        end
    end

    // 计算平均分
    if (valid_scores >= 3) begin
        averageScore <= (sum - max - min) / (valid_scores - 2);
    end else if (valid_scores > 0) begin
        averageScore <= sum / valid_scores;
    end
end
```

3. 修改测试逻辑：在测试逻辑中需要确保在某些情况下生成弃票的分数。例如，可以随机决定是否为弃票：

```
// 逐个输入裁判的分数
for (i = 0; i < 7; i = i + 1) begin
    if ($random % 5 == 0) begin  // 随机决定是否弃票
        score = 15;  // 弃票标志
    end else begin
        score = $random % 15;  // 正常分数
    end
    score_valid = 1; #10;
    score_valid = 0; #10;
end
```

在实际测试中，需要根据实际情况调整弃票的概率和逻辑。

6.1.2 重复或无效的分数输入

确保模块能够正确处理重复输入的分数或无效的分数信号。例如，如果同一裁判不小心输入了两次分数，或者分数输入信号不稳定，模块应该能够识别并处理这种情况。能够识别并处理异常或不合理的分数，例如超出预定范围的分数。

1. 异常分数处理

增加了一个条件检查，只有当 score 小于 16 时，才会将其计入 scores 数组，并增加 num_scores 的值。这样可以确保只有有效的分数才会被处理。

```
45      // 时钟控制的主要逻辑
46      always @(posedge clk) begin
47          if (score_valid) begin
48              // 只处理 0 到 15 范围内的分数
49              if (score <16) begin
50                  scores[num_scores] <= score;
51                  num_scores <= num_scores + 1;
52              end
53          end
```

由于 Verilog 使用无符号数，因此不需要专门处理负数，任何负数值在硬件中都会被解释为一个大于或等于 0 的数。

2. 裁判输入分数数量不唯一

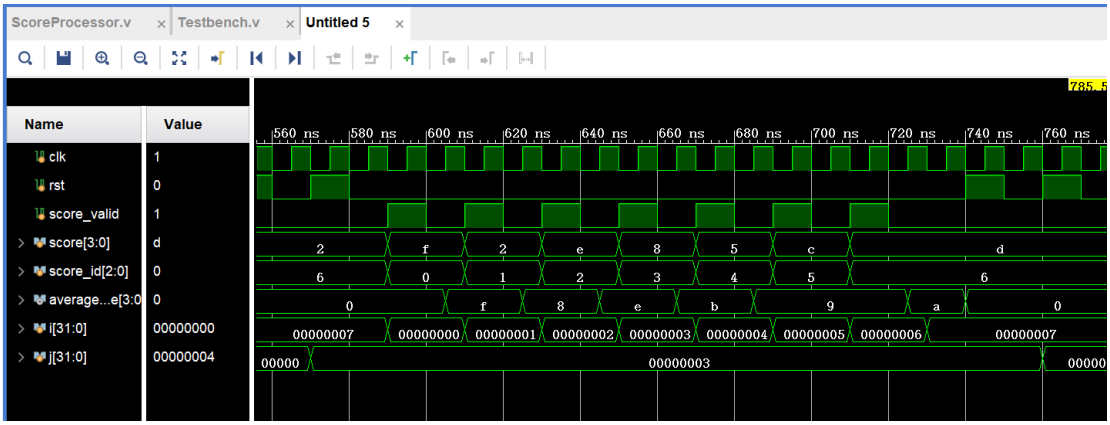
当一个裁判输入多个分数时，可以采用取第一次有效值作为最终分数。由于要判断每个裁判输入的分数是否有效，以及每个裁判输入的分数是否唯一，**这要求需要知道哪个分数对应于哪个裁判，需要额外的输入来表示当前输入的分数是哪个裁判的（score_id）**，下面是解决这类问题的原始代码：

ScoreProcessor

```
module ScoreProcessor(
    input clk,   // 时钟信号
    input rst,   // 清零信号
    input [3:0] score,   // 当前裁判的打分
    input [2:0] score_id, // 当前裁判的ID
    input score_valid,   // 当前分数有效信号
    output reg [3:0] averageScore  // 平均分数输出
);
    reg [3:0] scores[6:0]; // 存储裁判的分数
    reg [6:0] score_entered; // 跟踪每个裁判是否已经输入分数
    reg [2:0] num_scores = 0; // 记录已经打分的裁判数量
    integer i, sum, max, min;

    // 重置逻辑
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            num_scores <= 0;
            averageScore <= 0;
            score_entered <= 0;
            for (i = 0; i < 7; i = i + 1) begin
                scores[i] <= 0;
            end
        end else if (score_valid && !score_entered[score_id]) begin
            scores[num_scores] <= score;
            num_scores <= num_scores + 1;
            score_entered[score_id] <= 1;

            // 重置计算用的变量
            sum = 0; max = 0; min = 15;

            // 计算总和以及最大和最小值
            for (i = 0; i < num_scores; i = i + 1) begin
                sum = sum + scores[i];
                if (scores[i] > max) max = scores[i];
                if (scores[i] < min) min = scores[i];
            end

            // 计算平均分
            if (num_scores >= 3) begin
                averageScore <= (sum - max - min) / (num_scores - 2);
            end else if (num_scores > 0) begin
                averageScore <= sum / num_scores;
            end
        end
    end
endmodule
```

Testbench

```
module Testbench;
    reg clk, rst, score_valid;
    reg [3:0] score;
    reg [2:0] score_id;
    wire [3:0] averageScore;
    integer i, j;

    // ScoreProcessor 模块实例化
    ScoreProcessor sp(clk, rst, score, score_id, score_valid, averageScore);

    // 生成时钟信号
    initial clk = 0;
    always #5 clk = ~clk;
//测试逻辑
initial begin
    // 模拟多组打分
    for (j = 0; j < 5; j = j + 1) begin
        rst = 1; #10;  // 发出清零信号
        rst = 0; #10;  // 结束清零
        score_valid = 0;  // 确保开始时分数无效

        // 逐个输入裁判的分数
        for (i = 0; i < 7; i = i + 1) begin
            score_id = i;  // 设置裁判ID
            score = $random % 16;  // 生成随机分数
            score_valid = 1; #10;
            score_valid = 0; #10;
```



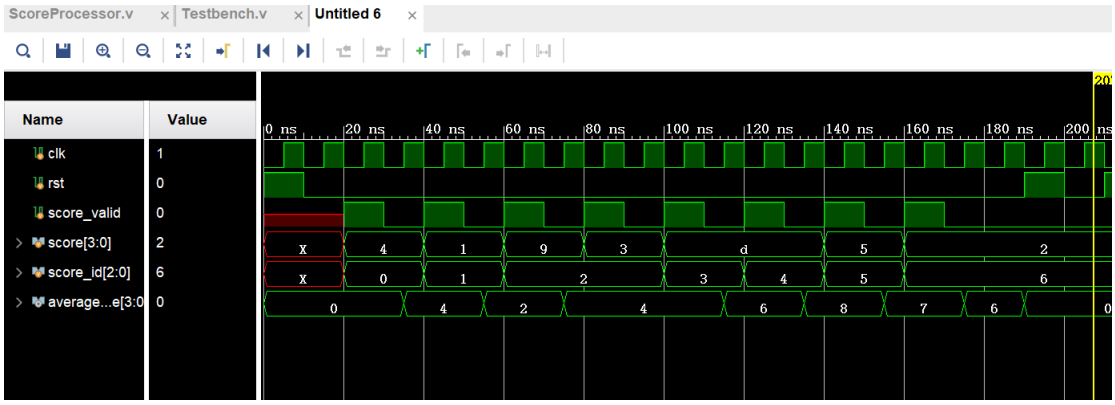
处理办法 1：同一裁判的多次打分只取第一次的分数

在这个版本中使用了两个 always 块：一个用于处理分数的输入和重置逻辑，另一个用于计算平均分。当一个新的分数输入时，如果是首次输入，它将被记录并用于后续的平均

分计算。这个设计确保了即使一个裁判尝试多次输入分数，也只会记录他们的第一次输入。

```
31    reg [3:0] scores[6:0]; // 存储裁判的分数
32    reg [6:0] score_entered; // 跟踪每个裁判是否已经输入分数
33    reg [2:0] num_scores = 0; // 记录有效打分的裁判数量
34    integer i, sum, max, min;
35
36    // 重置逻辑
37    always @(posedge rst or posedge clk) begin
38        if (rst) begin
39            num_scores <= 0;
40            averageScore <= 0;
41            score_entered <= 0;
42            for (i = 0; i < 7; i = i + 1) begin
43                scores[i] <= 0;
44            end
45        end else if (score_valid && !score_entered[score_id]) begin
46            scores[num_scores] <= score;
47            num_scores <= num_scores + 1;
48            score_entered[score_id] <= 1;
49        end
50    end
```

```
37    // 测试逻辑
38    initial begin
39        // 模拟多组打分
40        for (integer j = 0; j < 5; j = j + 1) begin
41            rst = 1; #10;  // 发出清零信号
42            rst = 0; #10;  // 结束清零
43            score_valid = 0;  // 确保开始时分数无效
44
45            // 逐个输入裁判的分数
46            for (integer i = 0; i < 7; i = i + 1) begin
47                score_id = i;
48                score = $random % 16;
49                score_valid = 1; #10;
50                score_valid = 0; #10;
51
52                // 模拟裁判ID为2的裁判重复打分
53                if (i == 2) begin
54                    score = $random % 16;
55                    score_valid = 1; #10;
56                    score_valid = 0; #10;
57                end
58            end
59
```



可以看到第三位裁判打分两次，记录了第一次的 9 分作为后续计算的值。

当然也可以由裁判自行决定取哪一次的分数，只需模拟裁判进行选择即可

6.2 总结

这个实验通过设计和实现一个数字电路模块，深化了对 Verilog 编程语言的理解，尤其在数据类型、模块结构和控制流语句方面。突出了时序逻辑设计的重要性，特别是在理解时钟信号在同步数字电路中的作用，并运用这些知识来设计有效的时序电路。实验还包括了对数据处理和算法实现的实践，设计电路以处理和计算 7 个裁判的打分数据，这涉及数据存储、排序、去除极值和计算平均值等功能，加深了对硬件中复杂算法实现的理解。通过熟悉数字电路设计工具 Vivado 的使用，从编写代码到仿真验证再到电路综合的整个流程，实验有助于提高对 FPGA 编程和设计流程的理解。培养了解决电路设计中遇到的问题的能力，例如处理复位逻辑、解决时序问题和优化资源使用等，并将理论知识与实际操作相结合，加深了对数字电路设计理论的理解和应用。

在统计多路裁判打分取平均分的影响因素时，可能有更多的情况需要考虑，或者有更多的解决方案。