记分牌算法仿真器项目

1 项目概述

本项目基于计算机系统结构课程的记分牌算法部分，编写的记分牌算法仿真器。旨在通过所学的课程知识，模拟实现记分牌算法在每一个时钟周期结束时的结果，进一步了解记分牌算法的完整执行过程，并借此机会提升个人的代码编写能力。

2 项目背景

2.1 记分牌算法的四个执行阶段

1.流出（issue）

Algorithm 1 Issue

Input: Operation to perform in the unit Op, Destination register number D, Source-register numbers $S1, S2$, Functional Unit FU

```
1:  function issue(Op, D, S1, S2)
2:      wait until (!Busy[FU] AND !Result[D]);
3:      Busy[FU] ← yes;
4:      Op[FU] ← op;
5:      Fi[FU] ← D;
6:      Fj[FU] ← S1;
7:      Fk[FU] ← S2;
8:      Qj[FU] ← Result[S1];
9:      Qk[FU] ← Result[S2];
10:     Rj[FU] ← Qj[FU] == 0;
11:     Rk[FU] ← Qk[FU] == 0;
12:     Result[D] ← FU;
13: end function
```

2.读操作数（read operand）

Algorithm 2 Read operands

Input: Functional Unit FU

```
1:  function read_operands(FU)
2:      wait until (Rj[FU] AND Rk[FU]);
3:      Rj[FU] ← no;
4:      Rk[FU] ← no;
5:      Qj[FU] ← 0;
6:      Qk[FU] ← 0;
7:  end function
```

3. 执行（execution）

Algorithm 3 Execute

Input: Functional Unit *FU*

```
1: function execute(FU)
2:     Execute whatever FU must do
3: end function
```

4.写结果（write result)

Algorithm 4 Write Result

Input: Functional Unit *FU*

```
1: function write_back(FU)
2:     wait until (∀f((Fj[f]≠Fi[FU] OR Rj[f]=no) & (Fk[f]≠Fi[FU] OR Rk[f]=no)))
3:     ∀f(if Qj[F]=FU then Rj[f] ← yes);
4:     ∀f(if Qk[F]=FU then Rk[f] ← yes);
5:     Result[D] ← 0;
6:     Busy[FU] ← no;
7: end function
```

2.2 记分牌算法的数据结构

1.指令状态表（Instruction Status)

指示正在执行的每条指令处于哪一个阶段。

指令	指令状态表			
	流出	读操作数	执行	写结果

2.功能部件状态表（Functional Unit Status)

记录各个功能部件的状态。

- Busy:忙标志，指出该功能部件正在使用。初值为no；
- Op：该功能部件正在执行或将要执行的操作；
- Fi：目的寄存器编号；Fj，Fk：源寄存器编号；
- Qj，Qk：指出向源寄存器Fj、Fk写数据的功能部件（即Fj、Fk中的数据由它们产生）；
- Rj、Rk：标志位，为yes表示Fj、Fk中的操作数就绪且还未被取走，否则就被置为no。

部件名称	功能部件状态表								
	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk

3.结果寄存器状态表（Register Status）

每个寄存器在该表中有一项，用于指出哪个功能部件（编号）将把结果写入该寄存器。

	结果寄存器状态表									
	F0	F2	F4	F6	F8	F10	F12	F14	...	F30
部件名称										

2.3 现有记分牌算法仿真器存在的问题

在本组（东北师大我们喜欢你组）进行讲座准备的过程中，就查询过大量现有的记分牌仿真器，发现基本全部现有的记分牌仿真器包括讲解视频都存在一个问题：Rj、Rk值修改的时间问题。

在记分牌实际的伪代码中，通常是在读操作数阶段，就将功能部件状态表中的Rj、Rk的由yes修改为no。但是基本全部现有的记分牌仿真器包括讲解视频，均在执行阶段的第一个时钟周期结束时才将Rj、Rk的由yes修改为no。当然，通过询问老师，我们得到了一个合理的解释，即两个修改的情况都可以，因为无论是何时修改，只要在这个过程中对最终的结果没有影响就都是可以的。

然而，在之后的进一步准备中，本组又提出了一种特殊的情况，即产生读后写（WAR）冲突的情况（如下图，Fu1部件执行的指令进入写结果阶段，Fu2部件执行的指令进入读操作数阶段）。此时，仅根据每个时钟周期结束时的结果进行判断，若不在读操作数阶段（设置为X）将Rj、Rk的由yes修改为no，那么就会导致在下一个时钟周期（执行阶段的第一个时钟周期，即X+1）才将Rj、Rk的由yes修改为no，让另一条指令对应的写结果阶段阻塞至再往后一个时钟周期（即X+2）才开始进行，这就与实际（依照算法执行）的情况（在X+1阶段就进行写结果阶段）产生了不符。但是，本组并未造出具体的样例用于证明大部分现有记分牌算法仿真器的可能存在问题，所出的题目中也并不存在相应的冲突。

部件名称	功能部件状态表								
	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Fu1	yes	Op1	F1	F2				no	
Fu2	yes	Op2	F3	F1	F4			yes	yes
Fu3	no								
Fu4	no								

在本项目撰写之初，为了了解现有项目的具体情况，又参看了陈胤儒学长的记分牌仿真器代码，通过阅读具体代码，我了解到了Rj、Rk值修改问题的真正原因。因为大部分仿真器的设计思路是，通过while循环来实现时钟周期的轮转，每一个时钟周期都要依次按照流出、读操作数、执行、写结果的顺序来分别判断并执行对应的指令，给以了各阶段的指令执行一个具体的先后顺序。如果在读操作数阶段将Rj、Rk

的由yes修改为no，那么当进行到同一时钟周期的写结果阶段的判断是，此时对应的Rj、Rk值已经被修改为no，让对应的指令在这一时钟周期就进入了写结果阶段，反而会于实际的情况相冲突。只有将Rj、Rk值的修改挪到下一个时钟周期进行，即在执行阶段的第一个时钟周期进行，就能避免这种问题，得到与实际情况相符的答案。

但是，由于项目运行过程中，输出值又是每个时钟周期结束时，记分牌中指令状态表、功能部件状态表和结果寄存器状态表的状态。因此，从输出答案上来看，Rj、Rk的值修改就出现在了执行阶段的第一个时钟周期结束时。这样，当遇到存在WAR冲突的指令时，让使用者产生疑惑：明明上个时钟周期结束时，存在一个功能部件的Fj/Fk值与当前功能部件的Fi值相同，并且对应的Rj/Rk值又为yes，为什么这个时钟周期该功能部件就能进入写结果阶段?

因此，本项目在设计的过程中，希望通过一种合理的方法来修正这个问题，让每一时钟周期的状态都遵照既定的算法来进行。

3 技术路线

3.1 数据结构

全局变量

```
enum OP { Op_Null, L_D = 1, ADD_D, SUB_D, MULT_D, DIV_D };
// 枚举操作（使用enum是为了后续代码具备更好的可读性） Op_Null:未定义操作
string ReOP[] = {"","L.D","ADD.D","SUB.D","MULT.D","DIV.D"};
// 操作转回字符，用于输出时快速获取对应操作

enum FU { Fu_Null, Integer = 1, Mult1, Mult2, Add, Divide };
// 枚举功能部件 Fu_Null:未定义功能部件
string ReFU[] = {"", "Integer", "Mult1", "Mult2", "Add", "Divide"};
// 功能部件转回字符，用于输出时快速获取对应操作

enum Status {UnStart=0,IS,RO,EXB,EXE,WR};
// 枚举指令状态阶段 UnStart:未开始; IS:issue; RO:read operand; EXB:execution begin;
// EXE:execution end; WR:write result;

int exeT[] = {0,1,2,2,10,40};
// 浮点执行所需时钟周期（顺序与OP中的操作对应）
// 即加法（减法）2个时钟周期，乘法10个时钟周期，除法40个时钟周期

struct InstructionStatus{
    OP Op = OP::Op_Null;
    string D = "", S1 = "", S2 = "";
    int Imm; // Imm:立即数
    int StatusTable[6] = {0}; // StatusTable:记录每个阶段的时钟周期
    Status status=Status::UnStart; // status:当前阶段
    bool finished = false,run=false; // finished:是否执行完成 run:该时钟周期是否执行
    FU Fu=Fu_Null; // Fu:记录指令对应的操作部件
};
vector<InstructionStatus> instructionStatus; // 指令状态表

struct FunctionUnit{
    bool busy = false;
    OP Op = OP::Op_Null;
    string Fi = "", Fj = "", Fk = "";
    FU Qj = FU::Fu_Null, Qk = FU::Fu_Null;
    bool Rj = false, Rk = false;
```

```
} FuStatus[Fu_size + 1]; // 功能部件状态表

map<string, FU> RegStatus; // 寄存器状态表
```

其中，为了代码具备更好的可读性，本项目采用了enum枚举对操作、功能部件和指令执行状态进行了设定，防止在代码阅读的过程中存在一些难以读懂的数组。

同时，本项目在记分牌算法的基本数据结构：指令状态表、功能部件状态表、寄存器状态表的基础上，在指令状态表上做了修改，将执行阶段时钟周期的记录拆分成执行阶段开始的时钟周期和执行阶段结束的时钟周期，方便通过数组进行基础。并且，增加status变量记录指令的当前阶段，以及finished、run和Fu变量分别用于记录当前指令是否执行完成，当前指令在当前时钟周期是否执行和指令对应的操作部件，具体执行部分代码的撰写。

3.2 代码函数部分

为了解决项目背景中存在的问题，即Rj、Rk值修改的时间问题。本项目设计之初思考的两种解决办法：

1. 修改四个阶段的判断和执行的顺序，将读操作数阶段放置在写结果阶段之后。
   虽然这一方法解决了先进行读操作数阶段带来的Rk、Rj值提前修改，导致存在WAR冲突的指令提前进行写结果的问题，但是也同样带来的新的问题。若两个代码之间存在写后读（RAW）冲突，此时正好一个指令（指令A）处在将要进入写结果的状态，另一个指令（指令B）处在将要进入读操作数的状态。先进行指令A的写结果阶段就会修改指令比对应的Rj/Rk的值，将其置为yes。则在后续指令B的读操作数的判断过程中，由于Rj/Rk的值已经修改为yes，记分牌会认为当前时钟周期指令B的两个源操作数均已准备好，在当前时钟周期指令B就进入了读操作阶段，这就与实际情况又产生了不符。
2. 依照做题的思路，将每个阶段的代码拆分成两个部分（判断部分和执行部分）。
   由于无论如何调整读操作数和写结果阶段的先后顺序，如果完全依照记分牌算法撰写的内容修改Rj、Rk的值，都会产生与实际情况不符的问题。那么，依照我们做题的思路，先对每个指令的下一个阶段进行判断，记录当前时钟周期，指令是否被阻塞（指令在当前时钟周期是否进行下一阶段），再让需要进行下一阶段的指令进行该阶段所需的操作，将阶段是否进行的判断与阶段的执行内容拆分成两个部分，就能实现完全依照记分牌算法撰写的内容修改Rj、Rk的值的同时，保证得到与实际情况相符的答案。

基于以上的思考，本项目将每个阶段的代码拆分成了判断部分和执行部分，构建以下函数：

```
void issue(InstructionStatus &Ins){...} // 判断流出
void runIssue(InstructionStatus &Ins){...} // 执行流出
void readOperand(InstructionStatus &Ins){...} // 判断读操作数
void runReadOperand(InstructionStatus &Ins){...} // 执行读操作数
void Execution(InstructionStatus &Ins){...} // 执行
void writeResult(InstructionStatus &Ins){...} // 判断写结果
void runWriteResult(InstructionStatus &Ins){...} // 执行写结果
```

判断流出

该函数主要是基于操作匹配对应的功能部件，进行功能部件是否空闲和有没有存在写后写（WAW）冲突的判断，对应了记分牌算法中流出阶段的判断部分。若功能部件是否空闲且不存在WAW冲突，则将对应指令的run变量修改为true，status变量修改为Status中对应IS的值，表示当前时钟周期该指令进行流出阶段，并记录当前指令执行对应的功能部件Fu。

```
void issue(InstructionStatus &Ins){ // 判断流出
    switch (Ins.Op){
        case OP::L_D:
```

```
            if (FuStatus[FU::Integer].busy == false &&
                (RegStatus.count(Ins.D) == 0 || RegStatus[Ins.D] == Fu_Null)){
                // 判断功能部件空闲且没有写后写（WAW）冲突
                Ins.run = true;
                Ins.status = Status::IS;
                Ins.Fu = FU::Integer;
            }
            break;
        case OP::ADD_D:
        case OP::SUB_D:
            if (FuStatus[FU::Add].busy == false &&
                (RegStatus.count(Ins.D) == 0 || RegStatus[Ins.D] == Fu_Null)){
                // 判断功能部件空闲且没有写后写（WAW）冲突
                Ins.run = true;
                Ins.status = Status::IS;
                Ins.Fu = FU::Add;
            }
            break;
        case OP::MULT_D:
            if (FuStatus[FU::Mult1].busy == false &&
                (RegStatus.count(Ins.D) == 0 || RegStatus[Ins.D] == Fu_Null)){
                // 判断功能部件空闲且没有写后写（WAW）冲突
                Ins.run = true;
                Ins.status = Status::IS;
                Ins.Fu = FU::Mult1;
            }
            else if (FuStatus[FU::Mult2].busy == false &&
                     (RegStatus.count(Ins.D) == 0 || RegStatus[Ins.D] ==
Fu_Null)){
                // 判断功能部件空闲且没有写后写（WAW）冲突
                Ins.run = true;
                Ins.status = Status::IS;
                Ins.Fu = FU::Mult2;
            }
            break;
        case OP::DIV_D:
            if (FuStatus[FU::Divide].busy == false &&
                (RegStatus.count(Ins.D) == 0 || RegStatus[Ins.D] == Fu_Null)){
                // 判断功能部件空闲且没有写后写（WAW）冲突
                Ins.run = true;
                Ins.status = Status::IS;
                Ins.Fu = FU::Divide;
            }
            break;
    }
}
```

执行流出

该函数主要是流出阶段具体的执行内容，基本与伪代码内容相对应。
其中，仅增加了对StatusTable的修改用于记录指令执行该阶段时的时钟周期。

```
void runIssue(InstructionStatus &Ins){ // 执行流出
    Ins.StatusTable[Status::IS] = clockCycle;
    FuStatus[Ins.Fu].busy = true;
```

```
    FuStatus[Ins.Fu].Op = OP::L_D;
    FuStatus[Ins.Fu].Fi = Ins.D;
    FuStatus[Ins.Fu].Fj = Ins.S1;
    FuStatus[Ins.Fu].Fk = Ins.S2;
    FuStatus[Ins.Fu].Qj =
        RegStatus.count(FuStatus[Ins.Fu].Fj) == 0 ? FU::Fu_Null :
RegStatus[FuStatus[Ins.Fu].Fj];
    FuStatus[Ins.Fu].Qk =
        RegStatus.count(FuStatus[Ins.Fu].Fk) == 0 ? FU::Fu_Null :
RegStatus[FuStatus[Ins.Fu].Fk];
    FuStatus[Ins.Fu].Rj = FuStatus[Ins.Fu].Qj == FU::Fu_Null ? true : false;
    FuStatus[Ins.Fu].Rk = FuStatus[Ins.Fu].Qk == FU::Fu_Null ? true : false;
    RegStatus.find(Ins.D)->second = Ins.Fu;
}
```

判断读操作数

该函数主要判断功能部件表上对应的的Rj、Rk的值是否为true (yes)，判断当前时钟周期该指令是否进行读操作数阶段的操作。

```
void readOperand(InstructionStatus &Ins){ // 判断读操作数
    if (FuStatus[Ins.Fu].Rj && FuStatus[Ins.Fu].Rk)
    {
        Ins.run = true;
        Ins.status = Status::RO;
    }
}
```

执行读操作数

该函数主要是读操作数阶段具体的执行内容，基本与伪代码内容相对应。

```
void runReadOperand(InstructionStatus &Ins){ // 执行读操作数
    Ins.StatusTable[Status::RO] = clockCycle;
    FuStatus[Ins.Fu].Rj = false;
    FuStatus[Ins.Fu].Rk = false;
    FuStatus[Ins.Fu].Qj = FU::Fu_Null;
    FuStatus[Ins.Fu].Qk = FU::Fu_Null;
}
```

执行

该函数代码主要基于执行阶段开始和执行阶段结束的两个判断展开。

- 执行阶段开始：执行阶段本身并不存在任何判断内容，但是由于无论是否是刚开始执行都将进入该函数，因此需要判断判断当前指令是否刚进入执行阶段，并记录执行阶段开始的时钟周期。
- 执行阶段结束：执行阶段结束的判断主要通过是否达到对应功能部件执行所需的时钟周期，并记录执行阶段结束的时钟周期。

```
void Execution(InstructionStatus &Ins){ // 执行
    if(Ins.status==Status::RO){ // 判断执行阶段开始
        Ins.status = Status::EXB;
        Ins.StatusTable[Status::EXB] = clockCycle;
    }
    if (clockCycle - Ins.StatusTable[Status::EXB]+1==exeT[Ins.Op]){ // 判断执行阶段
结束
        Ins.status = Status::EXE;
        Ins.StatusTable[Status::EXE] = clockCycle;
    }
}
```

判断写结果

该函数主要是判断是否存在WAR冲突，判断当前时钟周期该指令是否进行写结果阶段的操作。

```
void writeResult(InstructionStatus &Ins){ // 判断写结果
    bool flag=true;
    for(int i=FU::Integer;i<=FU::Divide;i++){
        if(i!=Ins.Fu){
            if ((FuStatus[i].Fj == FuStatus[Ins.Fu].Fi && FuStatus[i].Rj == true)
||
                (FuStatus[i].Fk == FuStatus[Ins.Fu].Fi && FuStatus[i].Rk ==
true)){
                flag=false;
            }
        }
    }
    if(flag){
        Ins.run = true;
        Ins.status = Status::WR;
    }
}
```

执行写结果

该函数主要是写结果阶段具体的执行内容，基本与伪代码内容相对应。
其中，增加了对于对应功能部件的变量恢复成初值，这些代码实际也可以不用，因为在下一个指令流出的过程中同样也会对全部的变量值进行修改。这里修改的原因是为了在输出结果时，能够让用户更加了解明确当前指令已经完成了全部的阶段（也是为了对应本组ppt所演示的过程）。

```
void runWriteResult(InstructionStatus &Ins){ // 执行写结果
    Ins.StatusTable[Status::WR] = clockCycle;
    for (int i = FU::Integer; i <= FU::Divide; i++)
    {
        if (i != Ins.Fu)
        {
            if (FuStatus[i].Qj == Ins.Fu)
                FuStatus[i].Rj = true;
            if (FuStatus[i].Qk == Ins.Fu)
                FuStatus[i].Rk = true;
        }
    }
    RegStatus.find(Ins.D)->second = FU::Fu_Null;
```

```
    FuStatus[Ins.Fu].busy = false;
    FuStatus[Ins.Fu].Op = OP::Op_Null;
    FuStatus[Ins.Fu].Fi = "";
    FuStatus[Ins.Fu].Fj = "";
    FuStatus[Ins.Fu].Fk = "";
    FuStatus[Ins.Fu].Qj = FU::Fu_Null;
    FuStatus[Ins.Fu].Qk = FU::Fu_Null;
    Ins.finished=true;
    finishNum++;
}
```

3.3 主函数部分

本项目主要采用了文件输入输出的方法，方便用户进行输入值的修改和输出值的查看。

在读入指令后，本项目同样采用while循环来模拟比时钟周期的循环，在每个时钟周期执行的过程中，我们将各阶段执行情况的判断与各阶段具体的执行拆分开。先遍历一遍全部指令，确定好哪些指令在本时钟周期进行某一阶段的操作，通过再一次的遍历执行这些需要进行的操作，并在每个时钟周期结束时，输出该时钟周期指令状态表、功能部件状态表、寄存器状态表的状态。

```
int main()
{
    freopen("input.txt", "r", stdin);     // 输入文件
    freopen("output.txt", "w", stdout); // 输出文件
    cin >> opt;
    string str = "\n";
    getline(cin, str);
    getInstruction();
    PrintStatus();
    while(finishNum<opt){
        clockCycle++;
        bool issueFlag=false;
        for(int i=0;i<instructionStatus.size();i++){
            instructionStatus[i].run=false; // 设定运行情况初值
            if(!instructionStatus[i].finished){
                switch(instructionStatus[i].status){
                    case Status::UnStart:
                        issue(instructionStatus[i]);
                        issueFlag=true;
                        break;
                    case Status::IS:
                        readOperand(instructionStatus[i]);
                        break;
                    case Status::RO:
                    case Status::EXB:
                        Execution(instructionStatus[i]);
                        break;
                    case Status::EXE:
                        writeResult(instructionStatus[i]);
                        break;
                }
            }
            if(issueFlag) break;
        }
        for(int i=0;i<instructionStatus.size();i++){
```

```
            if(instructionStatus[i].run){
                switch (instructionStatus[i].status)
                {
                case Status::IS:
                    runIssue(instructionStatus[i]);
                    break;
                case Status::RO:
                    runReadOperand(instructionStatus[i]);
                    break;
                case Status::WR:
                    runWriteResult(instructionStatus[i]);
                    break;
                }
            }
        }
        PrintStatus();
    }
    return 0;
}
```

3.4 项目测试部分

测试部分通过fc命令来比较程序输出文件的内容与答案文件的内容是否相同，来对程序进行测试，判断程序的正确性。

同时，测试部分还可用于用户作业答案的检验。

```
int main()
{
    system("scoreboard.exe"); // 运行待测程序
    int result = system("fc answer.txt output.txt"); // 比对输出和答案是否一样
    if (result == 0){
        cout << "The files are identical." << endl;
    }
    else{
        cout << "The files are different." << endl;
    }
    system("pause");
    return 0;
}
```

> **项目测试部分的问题**:
>
> 因为测试使用的是`fc`命令。`fc`会将空格、制表符和换行符视为文件内容的一部分。如果输出文件中的空白字符与预期结果文件中的空白字符存在微小差异（例如，由于不同的文本编辑器或操作系统产生的换行符差异），即使实际输出结果是正确的，`fc`也会认为文件不同。这就导致实际测试过程中，要求答案文件的格式完全符合项目的输出格式，使得答案文件的撰写非常麻烦，不能有一点格式上的问题。
>
> 后续进一步改进项目时，也需要改进这一方面的问题，让用户能够更加方便的准备测试所需的答案文件，方便进行测试。

3.5 项目创新点

为了使输出的结果更加符合记分牌算法的每一时钟周期结果时的状态，本项目在现有的记分牌算法仿真器的基础上做了进一步的修改和创新，通过将每个阶段的代码拆分成了判断部分和执行部分，进行分开执行，解决了现有的记分牌算法仿真器存在的Rj、Rk值修改的时间问题，使输出结果遵照记分牌算法的伪代码，在读操作数阶段完成将Rj、Rk的由yes修改为no。

3.6 项目未来规划（这部分应该不算在分数里）

由于时间的原因，本项目仅完成了基本的记分牌算法模拟器和测试，即完成v1.0版本的项目构建。在考试完成后，本项目也希望能够在后续做进一步的调整和改进工作，进一步拓展构建v2.0版本。

对于v2.0版本，本项目期望能够在v2.0版本通过Qt框架，构建GUI程序，实现用户可交互的图形界面，并实现包括执行所需时钟周期等多个变量的可修改，让非代码用户也能实现个性化的记分牌算法模拟。还希望能够实现每一时钟周期的按钮跳转和输入跳转，方便用户进行更好的查看。同时，前文也提到了关于项目测试部分存在的问题，也希望v2.0版本能够找到一个合适的办法来解决这一问题，方便用户对程序进行测试和答案检查。

4 工作量

4.1 代码行数

scoreboard.cpp：347行

test.cpp：18行

4.2 工作分解结构（WBS）

由于项目开始时间较晚，距离考试仅一周时间。因此，项目的第一目的是能够保证项目的基本实现，即完成v1.0版本的项目构建。后续如果还有时间，期望能够再进行拓展，开启v2.0版本甚至v3.0版本。

整体任务	细分任务	时间	完成情况
v1.0版本 基本记分牌算法及测试	项目初步构建	6月17日下午	确定项目使用的语言，搭建项目编写的vscode环境，整理项目相关资料，梳理项目整体撰写逻辑。
	项目部分代码撰写	6月17日下午至6月17日晚	完成项目大致数据结构构建，实现指令的输入。
	项目文档撰写	6月17日晚	将现有项目内容部署到github上，撰写项目文档的大致框架。之后就能通过github的版本控制，进行项目撰写内容的相应记录。
	部分代码编写	6月18日至6月21日	由于每天花在这部分的上的时间较为碎片化，无法很好的做整体的时间区分，因此合并为一个细分任务。 完成读操作数、执行、写结果部分的函数编写，并在编写过程中，进一步修改整体数据结构。
	算法模拟器整体编写	6月22日晚	完成记分牌算法整体基本代码编写，能够实现在输入指令队列的情况下，输出每一个时钟周期结束时，记分牌算法内数据的值
	算法测试编写		撰写测试代码，实现用户不读懂代码也能够进行测试，判断当前输入输出的正确性。

4.3 测试

目前，使用已有的作业题和部分自编指令进行相应测试，配合使用`陈胤儒_记分牌.cpp`进行相关校验。
测试相关内容已保存在test文件夹中。
也可按照readme中撰写的测试方法进行相关代码的测试，进一步验证代码的正确性。

4.4 版本控制和发布

项目自撰写之初就发布于github仓库中，其中大部分的项目修改和提交都有相关记录，可供参考和审阅。
项目目前已完成v1.0版本的撰写，保存在scoreboard v1.0文件夹中，后续进一步的拓展将通过建立v2.0、v3.0等文件夹进行区分。

各版本间的区别

- v1.0版本 - 基本实现记分牌算法的模拟，能够读入指令后正确输出每一个时钟周期结束的时候记分牌中数据的值。构建基本的项目测试方法，实现项目结果的正确性测试。
- v2.0版本（待开发）- 实现项目的GUI功能，减少对大部分数据的初值限制，希望能够使使用者能够更加自由的进行记分牌算法的仿真模拟。