Tomasulo 算法仿真器

项目背景

本项目是计算机系统结构课程选题之一，目的是通过手写仿真器来实现Tomasulo算法，加深对Tomasulo算法的理解。

项目功能

本项目实现的仿真器可以模拟 Tomasulo 算法的功能，输出 t 条指令在各周期的状态和保留站、寄存器状态表的信息。

项目部署及使用说明

运行系统：Windows 系统
运行环境：Dev-C++（Version 6.3）
Dev-C++安装链接：https://github.com/Embarcadero/Dev-Cpp/releases
运行代码的方法：
1.下载并安装 Dev-C++
2.下载项目源代码“Tomasulo 算法模拟器.cpp”文件
3.在 Dev-C++中打开源代码文件，点击“编译运行”
4.复制粘贴测试用例或者输入指令条数 t，然后回车换行并输入 t 条指令
5.再点击“Enter”键若干次，即可依次看到每个周期指令状态表、保留站内容和寄存器状态表的所有信息

项目原理

Tomasulo 算法中指令执行步骤分为三个阶段：
一、流出
从指令队列的头部取一条指令。若该指令的操作所要求的保留站有空闲（避免结构冲突），就把该指令送到该保留站(设为 r)。若其操作数在寄存器中已经就绪，就将这些操作数送入保留站 r；若其操作数未就绪，就把将产生该操作数的保留站的标识送入保留站 r，一旦被记录的保留站完成计算，直接把数据送给保留站 r。通过寄存器换名和对操作数进行缓冲，消除 WAR 冲突。另外，还要完成对目的寄存器的预约工作。
由于指令是按序流出的，如果有多条指令写同一个结果寄存器时，最后留下的预约结果一定是最后一条指令的，即消除了 WAW 冲突。
二、执行
当两个操作数都就绪后，本保留站就用相应的功能部件开始执行指令规定的操作，把发生 RAW 冲突的可能性减小到最少。load 和 store 指令的执行需要两个步骤：计算有效地址并把有效地址放入 load 或 store 缓冲器，load 指令还要从存储器中读取数据。
三、写结果

功能部件计算完毕后，如果没有其他指令正在写结果，CDB 是就绪的，就将计算结果放到 CDB 上，所有等待该计算结果的寄存器和保留站都同时从 CDB 上获得所需要的数据。store 指令完成对存储器的写入。

技术路线

* 数据结构

1.与指令有关的数据结构

```
int t; //指令条数
struct instr  //结构体 instr 用于存放指令的各个细节
{
    std::string op;//指令的操作码
    std::string name;//整条指令的信息
    int fi,fj,fk,time,device,vj,vk,qj,qk,a;
    int issue,execution,write;
    int begin,result1,result2,result3,result4,result5,result6,flag1,flag2;
/*fi、fj、fk 分别为指令的目的寄存器号和两个源操作数所在的寄存器号；
time 是指令执行周期数-在后面的 read 函数中设置；
device 为指令所在的保留站的编号；
vj、vk 为指令的两个源操作数的值；
qj、qk 为指令的源操作数的就绪状态，如果已就绪，设置 qj、qk 为 0，否则将即将产生该操作数的保留站的编号赋值给 qj、qk；
a 为 load 指令的立即数的值；
issue、execution、write 分别代表指令在第几个周期流出、完成执行和写结果；
begin 代表正在执行，begin 初值为 0，当操作数都就绪时，指令进入执行阶段，begin++，并在指令状态表 execution 对应的位置输出“正在执行”，当指令执行结束时，begin 变为 0；
result1、result2、result3 存储写结果阶段写入保留站 Vj 字段的结果；
result4、result5、result6 存储写结果阶段写入保留站 Vk 字段的结果；
flag1、flag2 为接收其他保留站结果的标记，为 1 时分别代表有其他保留站要向该保留站 Vj、Vk 字段写结果；
*/
} instruction[100]; //声明结构体变量 instruction
std::queue<int> q;//指令队列，指令从指令队列中顺序流出
struct st          //声明结构体 st，其成员为 n（指令号）和 step（指令处于第几阶段）
{
    int n,step;
};
std::queue<st> now;//声明结构体变量 now，储存当前周期的队列变换
std::set<int> step[6];//存放各个阶段的指令队列
std::set<int>::iterator it,ite;//迭代器，用来枚举
std::map<std::string,int> map;//用来做映射，将数字与指代的指令的操作码联系，例如字符串“LD”映射为数字 1，“SUBD”、“ADDD”映射为数字 3，“DIVD”、“MULTD”映射为数字 6
```

std::map<int,std::string> mapp;//用来做映射，将数字与指代的保留站联系，例如数字 1 映射为字符串“load1”，表示为指令分配对应类型的保留站，对于 LD 指令分配 load1 或 load2 保留站，对于 SUBD 和 ADDD 指令分配 add1、add2 或 add3 保留站，对于 DIVD 和 MULTD 指令分配 mult1 或 mult2 保留站

2.与寄存器有关的数据结构

int reg[32];//每个寄存器都有对应的一项，代表寄存器状态，如果 reg 数组为 0 代表该寄存器中数据就绪，否则存储即将产生该操作数的保留站的编号，该寄存器将接收对应保留站的结果
int result[3][32]; //存放寄存器的值，即寄存器在指令写结果阶段接收的保留站产生的结果，对于寄存器 i，result[0][i]存放保留站中指令的第一操作数，result[1][i]存放保留站中指令的第二操作数或是 LD 指令的立即数，result[3][i]代表保留站中指令的操作码（如 result[2][i]为 1 代表指令操作码是 LD）

3.与保留站有关的数据结构

int device[8];//每个保留站都有对应的一项，存放占用该保留站的指令号

* 项目框架

1.init 函数指明数字与指代的指令的操作码的映射关系和数字与指代的保留站之间的映射关系。map 用来将数字与指代的指令的操作码联系，例如字符串“LD”映射为数字 1，“SUBD”、“ADDD”映射为数字 3，“DIVD”、“MULTD”映射为数字 6。mapp 用来将数字与指代的保留站联系，例如数字 1 映射为字符串“load1”，表示为指令分配对应类型的保留站，对于 LD 指令分配 load1 或 load2 保留站，对于 SUBD 和 ADDD 指令分配 add1、add2 或 add3 保留站，对于 DIVD 和 MULTD 指令分配 mult1 或 mult2 保留站。

```
inline void init()
{
    map["LD"] = 1;map["SUBD"] = 3;map["ADDD"] = 3;map["DIVD"] = 6;map["MULTD"] = 6;
    mapp[1] = "load1";mapp[2] = "load2";mapp[3] = "add1";mapp[4] = "add2";mapp[5] = "add3"; mapp[6] = "mult1"; mapp[7] = "mult2";
}
```

2.read 函数用于读取指令的信息，包括指令的操作码 op、目的寄存器号 fi、源操作数所在的寄存器号 fj 和 fk，以及设置指令执行所需的周期数 time,load 指令:2 个时钟周期；加、减法指令:2 个时钟周期；乘法指令:10 个时钟周期；除法指令:20 个时钟周期。

```
inline void read()
{
    scanf("%d",&t);  //输入要执行的指令条数
    for(int i = 1;i <= t;i++)
    {
        std::string s;
        getline(std::cin,s);
        instruction[i].name = s;
        //以处理 LD 指令为例执行如下代码
        if (s[0] == 'L')
        {
            instruction[i].op = "LD"; //load 指令的操作码为"LD"
```

```
instruction[i].time = 2; //设置 load 指令的执行阶段需要 2 个时钟周期
int cnt = 0;
for(int j = 2;j < s.length();j++)
{
    if (s[j] != ' ')
    {
        cnt = j;
        break;
    }
}
instruction[i].fi = s[cnt + 1] -'0'; //读取指令的目的寄存器号 fi
if (s[cnt + 2] >= '0' && s[cnt + 2] <= '9')
{
    instruction[i].fi *= 10;
    instruction[i].fi += s[cnt + 2] - '0';
}
for(int j = cnt + 1;j < s.length();j++)
{
    if (s[j] == ',')
    {
        cnt = j;
        break;
    }
}
instruction[i].a = s[cnt + 1] - '0'; //读取 load 指令的立即数
if (s[cnt + 2] >= '0' && s[cnt + 2] <= '9')
{
    instruction[i].a *= 10;
    instruction[i].a += s[cnt + 2] - '0'; //处理立即数是两位数的情况
}
for(int j = cnt + 1;j < s.length();j++)
{
    if (s[j] == 'R')
    {
        cnt = j;
        break;
    }
}
instruction[i].fj = s[cnt + 1] - '0'; //读取指令的源操作数所在的寄存器号 fj
if (s[cnt + 2] >= '0' && s[cnt + 2] <= '9')
{
    instruction[i].fj *= 10;
    instruction[i].fj += s[cnt + 2] - '0';
}
```

```
        }
    }
}
```

3.print 函数用于输出每个周期的指令状态表、保留站内容和寄存器状态表的信息。

```
inline void print(int x)
{
    printf("                                                周期%d:\n",x);
    //输出指令状态表
    printf("指令名        |Issue|Execution completed|Write Result|\n");
    //遍历输入的 t 条指令并分别输出在第几个周期流出、执行完成和写结果

    //输出保留站内容
    printf("保留站名称|Busy |Op   |Vj          |Vk          |Qj   |Qk   |A         |\n");
    //遍历 7 个保留站并依次输出保留站各字段的值
```

//以输出 Vj 字段为例：

//对于 load 指令：

```
    //根据 reg[instruction[device[i]].fj 的值是否为零判断操作数是否就绪，进行寄存器换名
    if(reg[instruction[device[i]].fj] == 0)
    printf("R[R%d]        |             |0    |     ",instruction[device[i]].fj);//若操作数就绪，输出操作数，即寄存器中的值
    else{
    printf("             |             |");
    std::cout << mapp[reg[instruction[device[i]].fj]]; //若未就绪，输出要写 fj 寄存器的保留站的编号
```

//对于浮点操作指令：

```
    if(instruction[device[i]].flag1 == 1 && instruction[device[i]].qj == 0){
    //如果 flag1 为 1 并且 instruction[device[i]].qj 为 0，代表有其他指令向该保留站的 Vj 字段写结果，该结果已产生，则输出该结果（在写结果阶段已存入保留站的 result1、result2 和 result3 中）
    if(instruction[device[i]].result3==1)printf("Mem[R[R%d]+%d]|",instruction[device[i]].result1,instruction[device[i]].result2);  //例如如果 instruction[device[i]].result3==1，代表该结果是由 load 缓冲器写入的，则输出 load 指令从存储器中取出的数据
    else             if(instruction[device[i]].result3==2)             printf("R[F%d]+R[F%d]|",instruction[device[i]].result1,instruction[device[i]].result2);
    else             if(instruction[device[i]].result3==3)             printf("R[F%d]-R[F%d]|",instruction[device[i]].result1,instruction[device[i]].result2);
    else             if(instruction[device[i]].result3==4)             printf("R[F%d]*R[F%d]|",instruction[device[i]].result1,instruction[device[i]].result2);
    else             if(instruction[device[i]].result3==5)             printf("R[F%d]/R[F%d]|",instruction[device[i]].result1,instruction[device[i]].result2);
}
    else if(instruction[device[i]].flag1 != 1 && instruction[device[i]].qj == 0)
//如果 flag1 不为 1 并且 instruction[device[i]].qj 为 0，代表该操作数已在寄存器中就绪，则输出寄存器中的操作数
    else printf("             |");//如果操作数未就绪，则在 Vj 字段对应的位置输出空格
```

//以输出 Qj 字段为例：如果操作数就绪，输出 Qj 的值 0；如果未就绪，输出即将产生该操作数的保留站的名称

```
    if(instruction[device[i]].qj == 0) printf("0    |");
    else std::cout << mapp[instruction[device[i]].qj];

    //输出寄存器状态表
    //遍历所有浮点寄存器，分别输出寄存器号、Qi 的内容和寄存器中的值
    //输出 Qi,Qi 为寄存器的状态，如果寄存器中的数据就绪，Qi 为 0，如果未就绪，Qi 为即将写该寄存器的保留站的名称
if (reg[i] != 0) std::cout << mapp[reg[i]];
else printf("             ");
    //输出寄存器中的值，如果 reg[i]==0，寄存器中的数据就绪，输出寄存器中存的保留站产生的结果，否则输出空格
if(reg[i] == 0 && result[2][i]==1) printf("Mem[R[R%d]+%d]",result[0][i],result[1][i]);
else if(reg[i] == 0 && result[2][i]==2) printf("R[F%d]+R[F%d]  ",result[0][i],result[1][i]);
else if(reg[i] == 0 && result[2][i]==3) printf("R[F%d]-R[F%d]  ",result[0][i],result[1][i]);
else if(reg[i] == 0 && result[2][i]==4) printf("R[F%d]*R[F%d]  ",result[0][i],result[1][i]);
else if(reg[i] == 0 && result[2][i]==5) printf("R[F%d]/R[F%d]  ",result[0][i],result[1][i]);
else printf("             ");
}
```

4.procedure 函数用于实现算法具体流程，在每个周期按顺序判断每条指令的状态。

如果有空闲保留站，指令按序流出到保留站，根据寄存器状态表的值是否为 0 判断源操作数是否就绪，进行寄存器换名，修改保留站对应字段，预约目的寄存器。

如果指令的操作数都就绪就进入执行阶段，进行运算，否则指令停顿。

当执行达到规定的执行周期数，并且在这个周期没有其他指令正在写结果，CDB 是就绪状态时，指令进入写结果阶段，遍历保留站的 Qj、Qk 表以及寄存器状态表，将运算结果传送到需要该结果的地方，并释放保留站资源。

```
inline void procedure()
{
    for(int i = 1;i <= t;i++) q.push(i); //所有指令按顺序进入队列
    int time = 0;
    while(1)
    {
        std::queue<st> now;//储存当前周期的队列变换，st 是结构体，储存 n（指令号）和 step（指令处于第几阶段）
        time++;
        //流出阶段
        if (!q.empty())
        {
            int x = q.front(); //从指令队列头部取一条指令
            //寻找是否存在空闲保留站，以 load 指令为例，如果 device[1]为 0，代表 1 号保留站空闲，则分配 load1 保留站，否则判断 2 号保留站是否空闲
            if (map[instruction[x].op] == 1)
            {
```

```
            if (device[1] == 0) instruction[x].device = 1;
            else if (device[2] == 0) instruction[x].device = 2;
        }
        //分配保留站 把代表该指令的编号放到指令 x 的结构体的 device 中
        if (instruction[x].device != 0)
        {
                instruction[x].issue = time; //可以流出
                reg[instruction[x].fi] = instruction[x].device; //即将要写目的寄存器的指令是 x
                if (reg[instruction[x].fj] == 0) instruction[x].qj = 0; //进行寄存器换名，如果第一操作数就绪，将 qj 置为 0，否则 qj 为即将写该寄存器的保留站的编号
                else
                {
                    instruction[x].qj = reg[instruction[x].fj];
                }
                if (reg[instruction[x].fk] == 0) instruction[x].qk = 0; //判断第二操作数是否就绪并设置 qk 的值
                else
                {
                    instruction[x].qk = reg[instruction[x].fk];
                }
                device[instruction[x].device] = x;
                //将流出阶段完成的 x 放入执行队列 包括待执行以及正在执行的指令
                now.push((st){x,2});
                q.pop();
            }
        }
        //执行阶段
        for(it = step[2].begin();it != step[2].end();++it)
        {
            int x = *it;
            //判断两个源操作数是否就绪
            if (instruction[x].op[0] == 'L' ||(instruction[x].qj==0 && instruction[x].qk==0))
            {
                instruction[x].begin++;
                instruction[x].time--;
                if (instruction[x].time == 0)
            {
                instruction[x].execution = time; //达到所需的执行周期，执行完毕之后，将执行完的时钟周期数放到指令的结构体的 execution 里保存
                instruction[x].begin=0;
                //将执行阶段完成的指令 x 放入写结果队列，并从执行队列删除
                now.push((st){x,3});
            }
```

```
            }
        }
         //写结果阶段
        for(it = step[3].begin();it != step[3].end();++it)
        {
            int x = *it;  //printf("%d",x); //x 为指令号，代表正在写结果的指令
                instruction[x].write = time;
//遍历寄存器状态表 Qi（即每个寄存器对应的 reg[i]的值），将运算结果存储到正在等待该结果的寄存器对应的 result 二维数组中，以便 print 函数将结果输出
                for(int i = 0;i <= 12;i+=2){
                    if(reg[i] == instruction[x].device){
                    result[0][i]=instruction[x].fj;
                    if(instruction[x].op=="LD") result[1][i]=instruction[x].a;
                    else result[1][i]=instruction[x].fk;
                    if (instruction[x].op == "LD") result[2][i]=1;
                    else if (instruction[x].op == "ADDD") result[2][i]=2;
                    else if (instruction[x].op == "SUBD") result[2][i]=3;
                    else if (instruction[x].op == "MULTD") result[2][i]=4;
                    else if (instruction[x].op == "DIVD") result[2][i]=5;
                }
                }
//同时将 Qi 的值设为 0，代表该寄存器中的数据已经就绪
                for(int i = 0;i <= 12;i+=2){
                    if(reg[i] == instruction[x].device) reg[i] = 0;
                }
                device[instruction[x].device] = 0;//释放保留站资源
//遍历保留站的 Qj、Qk 表，将运算结果传送到需要该结果的保留站的 result1 等变量中存储，以便 print 函数输出结果
                for (int i = 1; i <= 7; i++) {
                    if(device[i]!=0){
//以向保留站的 Qj 和 Vj 字段传结果为例：
                    if (instruction[device[i]].qj == instruction[x].device) {
                            instruction[device[i]].result1 = instruction[x].fj;
                            if(instruction[x].op=="LD")
instruction[device[i]].result2=instruction[x].a;
                            else instruction[device[i]].result2 = instruction[x].fk;
                    if (instruction[x].op == "LD") {instruction[device[i]].result3=1; }
                    else if (instruction[x].op == "ADDD") {instruction[device[i]].result3=2;}
                    else if (instruction[x].op == "SUBD") {instruction[device[i]].result3=3;}
                    else if (instruction[x].op == "MULTD") {instruction[device[i]].result3=4;}
                    else if (instruction[x].op == "DIVD") {instruction[device[i]].result3=5;}
                    instruction[device[i]].flag1 = 1;
                    instruction[device[i]].qj = 0;
                    }
```

```
                }
                }
                //将写结果阶段完成的 x 放入结束队列，并从写结果队列中删除
                now.push((st){x,4});
                break;  //控制一个周期最多只有一条指令写结果
        }
                //执行该指令周期的队列更新
        while(!now.empty())
        {
            st x = now.front();
            now.pop();
            step[x.step - 1].erase(x.n);
            step[x.step].insert(x.n);
        }
        //输出结果
        print(time);
        //判断是否所有指令均已完成
        if (step[4].size() == t) break;
    }
}
```

* 关键技术

* 状态机

每条指令有三个状态：流出、执行和写结果。当指令满足进入条件时，进入下一状态。

如果有空闲保留站，指令进入流出状态，按序流出到保留站，根据寄存器状态表的值是否为 0 判断源操作数是否就绪，修改保留站对应字段，预约目的寄存器。如果指令的操作数都就绪就进入执行状态，进行运算。当执行达到规定的执行周期数，并且在这个周期没有其他指令正在写结果，CDB 是就绪状态时，指令进入写结果状态，遍历保留站的 Qj、Qk 表以及寄存器状态表，将运算结果传送到需要该结果的地方，并释放保留站资源。

关键代码：

```
for(int i = 1;i <= t;i++) q.push(i); //队列是先进先出的数据结构，所有指令按顺序进入流出队列
std::queue<st> now;//储存当前周期的队列变换
std::set<int> step[6];//存放各个阶段的指令队列
//流出阶段
int x = q.front(); //从指令队列头部取一条指令，寻找是否存在空闲保留站，如果有空闲的保留站，分配保留站并设置指令的 issue 值为当前周期数，指令可以流出
now.push((st){x,2});//将流出阶段完成的 x 放入执行队列，包括待执行以及正在执行的指令
q.pop();//指令按顺序从 q 队列中弹出
//执行阶段
for(it = step[2].begin();it != step[2].end();++it)
{
```

```
    int x = *it; //x 为指令号，代表正在执行队列的的指令
    //判断两个源操作数是否就绪，如果都已就绪，指令状态为正在执行，当达到所需的执行周期，执行完毕之后，设置指令的 execution 值为当前时钟周期数：instruction[x].execution = time;
    now.push((st){x,3});//将执行阶段完成的指令 x 放入写结果队列，并从执行队列删除
}
//写结果阶段
for(it = step[3].begin();it != step[3].end();++it)
{
    int x = *it;  //x 为指令号，代表正在写结果的指令
    instruction[x].write = time;//设置指令的 write 值为当前时钟周期数
    now.push((st){x,4});//将写结果阶段完成的 x 放入结束队列，并从写结果队列中删除
    break;  //控制一个周期最多只有一条指令写结果
}
//执行该指令周期的队列更新
        while(!now.empty())
        {
            st x = now.front();
            now.pop();
            step[x.step - 1].erase(x.n);
            step[x.step].insert(x.n);
        }
//输出结果
        print(time);
//判断是否所有指令均已完成
        if (step[4].size() == t) break;
```

* 指令循环

通过 while(1)大循环判断每个周期指令的状态，read 函数中使用 for 循环读取每条指令的操作码、操作数等信息，对于每个阶段对应的指令队列中，使用 for 循环和迭代器 it 遍历指令并修改相应的值。

* 整数与字符串之间的映射

```
std::map<std::string,int> map;//用来做映射，将数字与指代的指令的操作码联系
std::map<int,std::string> mapp;//用来做映射，将数字与指代的保留站联系
map["LD"] = 1;map["SUBD"] = 3;map["ADDD"] = 3;map["DIVD"] = 6;map["MULTD"] = 6;
mapp[1] = "load1";mapp[2] = "load2";mapp[3] = "add1";mapp[4] = "add2";mapp[5] = "add3"; mapp[6] = "mult1"; mapp[7] = "mult2";
```

如当 map[instruction[x].op] == 1 时，代表该指令的操作码是 LD，所对应的是 load 缓冲器。

再比如在 print 函数中 std::cout << mapp[reg[instruction[device[i]].fj]]; //用 mapp 做映射，从要写 fj 寄存器的保留站的编号映射出保留站名称（字符串）。

* 写结果时 CDB 就绪态的判断

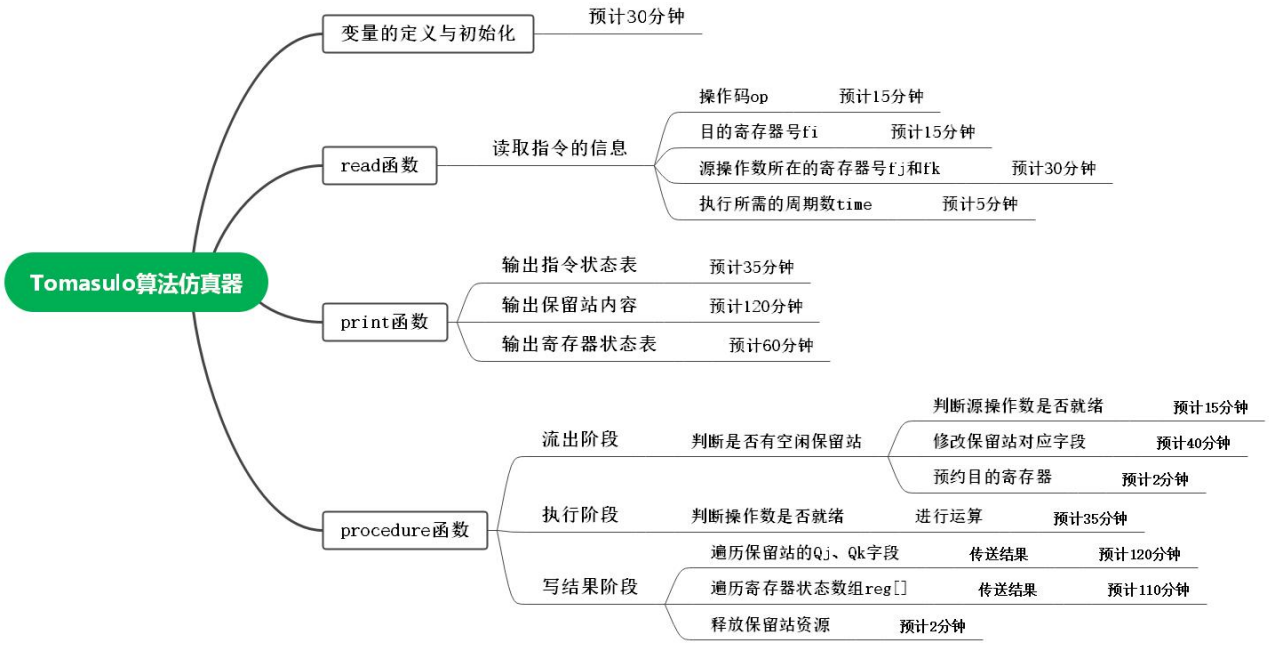
在写结果阶段在 for 循环中使用 break，只从写结果队列取一条指令，将保留站产生的结果传给寄存器和保留站的对应字段中，然后 break 跳出循环来保证每个周期最多只有一条指令写结果。

项目实现

本项目使用 Dev-C++编写代码，可以在该环境中编译运行。编程语言为 C 语言和 C++。

工作量

* 代码行数：524 行
* 预计用时：共计 10 小时 34 分钟
* WBS 工作任务分解图



* 实际用时：共计 34 小时 56 分钟
* 时间记录日志

日期	开始时间	结束时间	净时间	活动
6.2	11:00	12:00	1 小时	配置环境 Dev-C++，解决中文字符不显示的问题
6.3	14:20	17:40	3 小时 20 分钟	理清算法大致思路与框架
6.5	17:24	23:30	6 小时 6 分钟	写读指令和输出结果两个函数
6.8	14:30	18:00	3 小时 30 分钟	编写流出阶段代码，将寄存器号换名为保留站编号或者操作数

6.10	17:00	23:00	6 小时	使用 result 二维数组保存保留站的结果，在写结果阶段传入等待该结果的的寄存器中存储
6.11	8:30	10:00	1 小时 30 分钟	写结果阶段使用 instruction[x].result1 等六个变量存储保留站的结果并传入保留站 Qj、Qk 字段需要的位置；与老师沟通交流
6.12	20:30	22:00	1 小时 30 分钟	在写结果阶段使用 break 来保证每个周期最多只有一条指令写结果
6.13	11:00	15:00	4 小时	解决写结果阶段传值的问题；进行测试；修复 bug
6.13	19:00	23:00	4 小时	完成技术文档项目背景、原理、框架部分
6.14	16:30	19:30	3 小时	完成技术文档中关键技术和工作量部分
6.15	9:00	10:00	1 小时	上传项目至 github

项目测试

本项目给出了典型的测试用例，例如具有写后写冲突、读后写冲突和写后读冲突的指令，可以体现 Tomasulo 算法的核心思想，即把发生写后读冲突的可能性减小到最少，以及通过保留站实现寄存器换名，消除写后写冲突和读后写冲突。

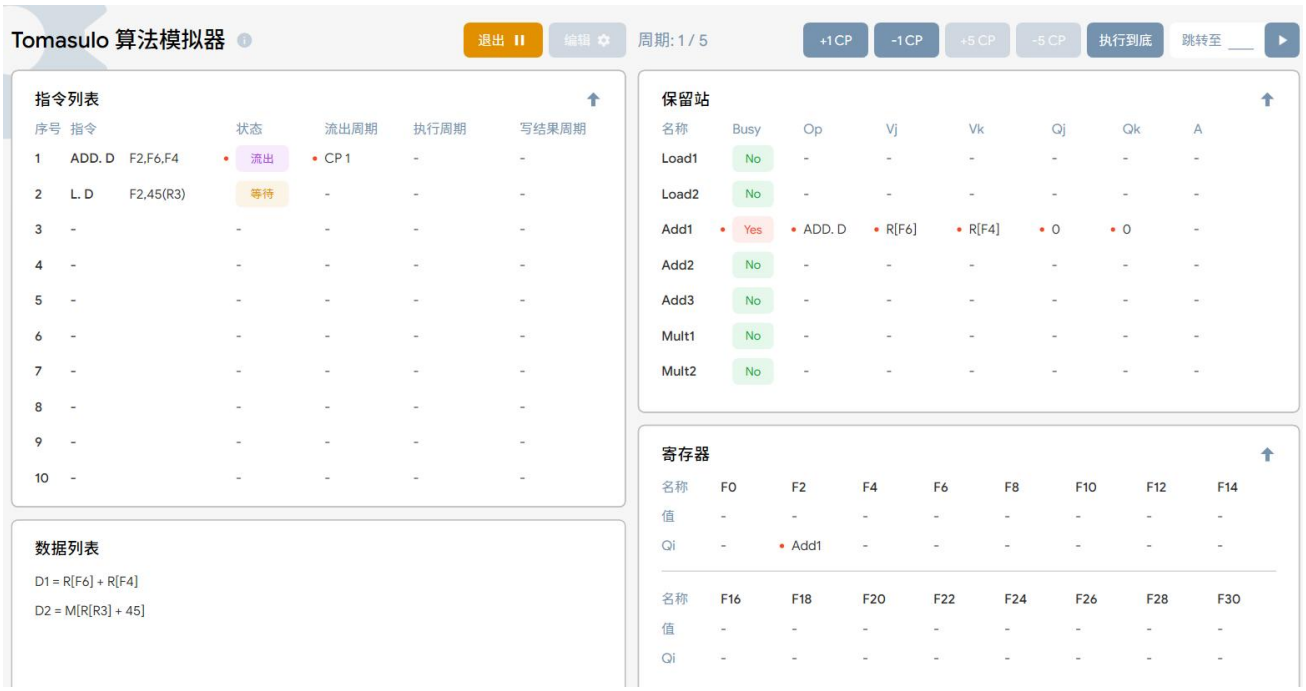
```
//用例 1：写后写冲突
2
ADDD F2,F6,F4
LD F2,45(R3)
```

对比各周期预期结果（由教材配套的何永杰和张晨曦老师的模拟器给出）和实际代码运行结果：

测试用例 1：

周期 1：

预期结果：



实际结果：

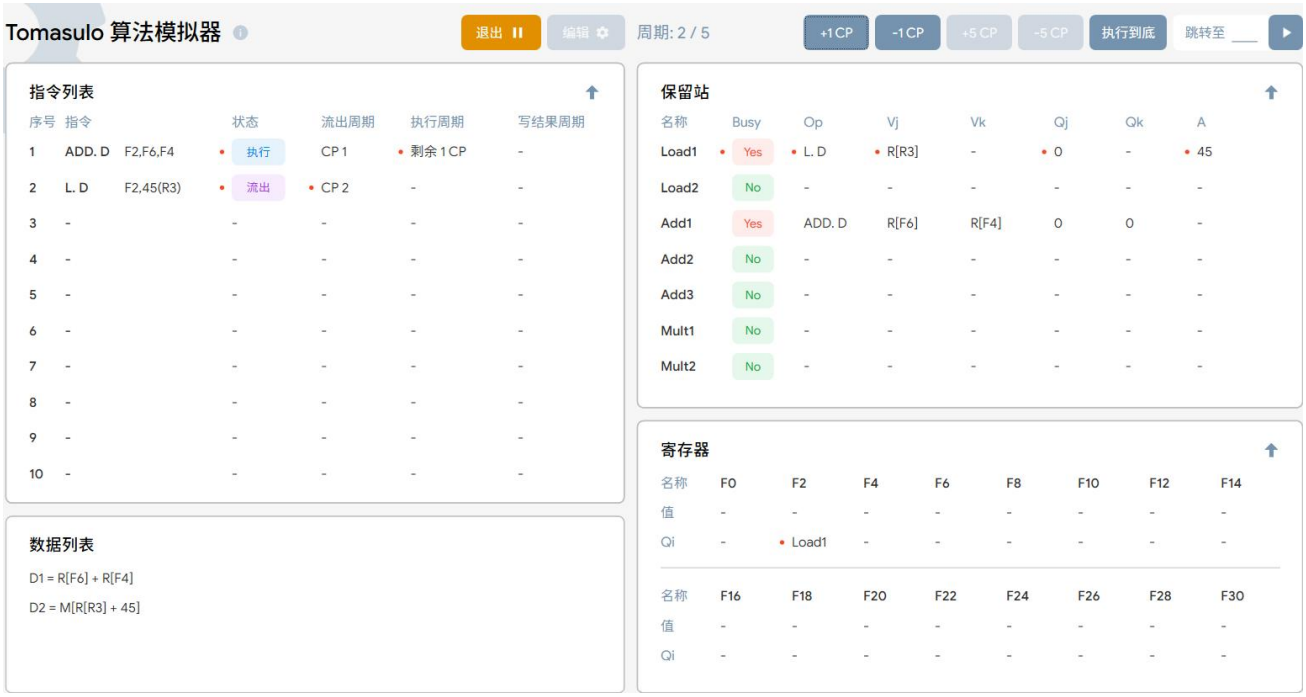
```
                                        周期1:
指令状态:
指令名          |Issue|Execution completed|Write Result|
ADDD F2,F6,F4   |    1|                   |            |
LD F2,45(R3)    |     |                   |            |

保留站内容:
保留站名称|Busy |Op   |Vj           |Vk           |Qj   |Qk   |A          |
load1     |no   |     |             |             |     |     |           |
load2     |no   |     |             |             |     |     |           |
add1      |yes  |ADDD |R[F6]        |R[F4]        |0    |0    |           |
add2      |no   |     |             |             |     |     |           |
add3      |no   |     |             |             |     |     |           |
mult1     |no   |     |             |             |     |     |           |
mult2     |no   |     |             |             |     |     |           |

寄存器状态表:
寄存器号|F0           |F2           |F4           |F6           |F8           |F10          |F12          |...|F30  |
Qi      |             |add1         |             |             |             |             |             |...|F30  |
值      |             |             |             |             |             |             |             |...|F30  |
Press any key to continue . . .
```

周期 2：

预期结果：



实际结果：

```
                                        周期2:
指令状态:
指令名          |Issue|Execution completed|Write Result|
ADDD F2,F6,F4 |    1|           正在执行|            |
LD F2,45(R3)  |    2|                    |            |

保留站内容:
保留站名称|Busy |Op   |Vj          |Vk          |Qj   |Qk   |A          |
load1     |yes  |LD   |R[R3]       |            |0    |     |R[R3]+45   |
load2     |no   |     |            |            |     |     |           |
add1      |yes  |ADDD |R[F6]       |R[F4]       |0    |0    |           |
add2      |no   |     |            |            |     |     |           |
add3      |no   |     |            |            |     |     |           |
mult1     |no   |     |            |            |     |     |           |
mult2     |no   |     |            |            |     |     |           |

寄存器状态表:
寄存器号|F0          |F2          |F4          |F6          |F8          |F10         |F12         |...|F30  |
Qi      |            |load1       |            |            |            |            |            |...|F30  |
值      |            |            |            |            |            |            |            |...|F30  |
Press any key to continue . . .
```

周期 3：

预期结果：

实际结果：

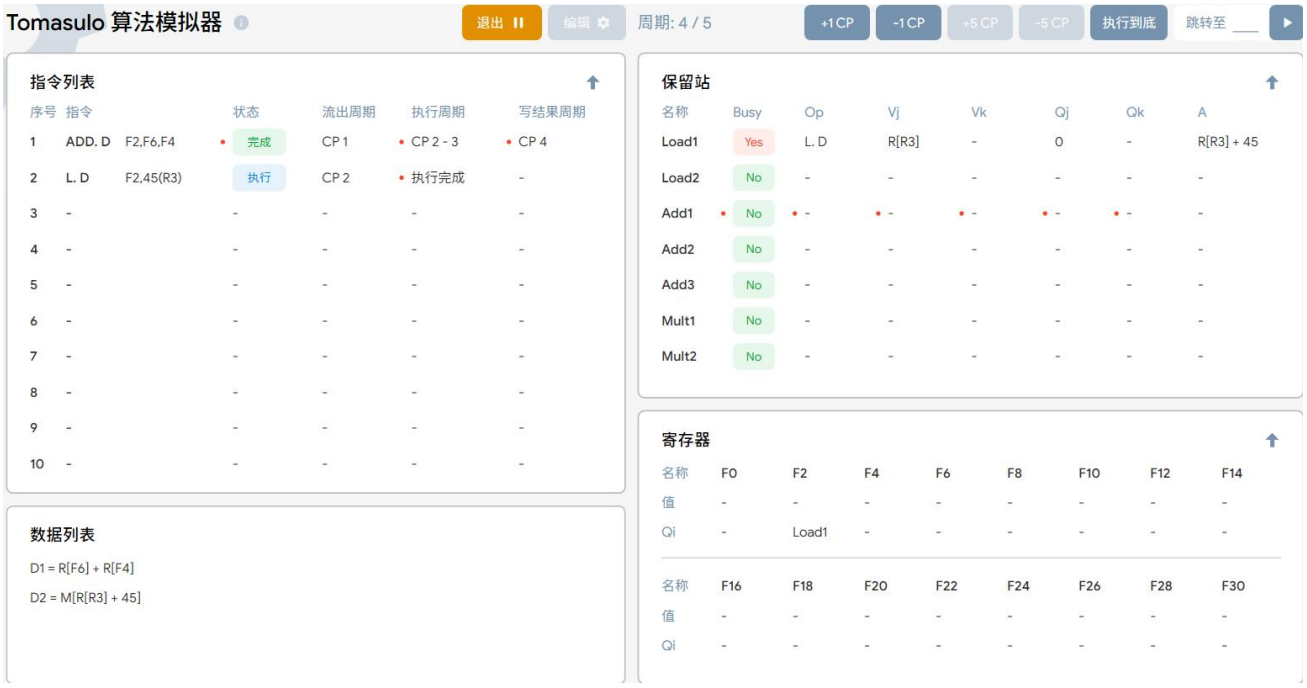
```
                                        周期3:
指令状态:
指令名          |Issue|Execution completed|Write Result|
ADDD F2,F6,F4 |    1|                  3|            |
LD F2,45(R3)  |    2|           正在执行|            |

保留站内容:
保留站名称|Busy |Op   |Vj          |Vk          |Qj   |Qk   |A          |
load1     |yes  |LD   |R[R3]       |            |0    |     |R[R3]+45   |
load2     |no   |     |            |            |     |     |           |
add1      |yes  |ADDD |R[F6]       |R[F4]       |0    |0    |           |
add2      |no   |     |            |            |     |     |           |
add3      |no   |     |            |            |     |     |           |
mult1     |no   |     |            |            |     |     |           |
mult2     |no   |     |            |            |     |     |           |

寄存器状态表:
寄存器号|F0          |F2          |F4          |F6          |F8          |F10         |F12         |...|F30  |
Qi      |            |load1       |            |            |            |            |            |...|F30  |
值      |            |            |            |            |            |            |            |...|F30  |
Press any key to continue . . .
```

周期 4：

预期结果：



实际结果：

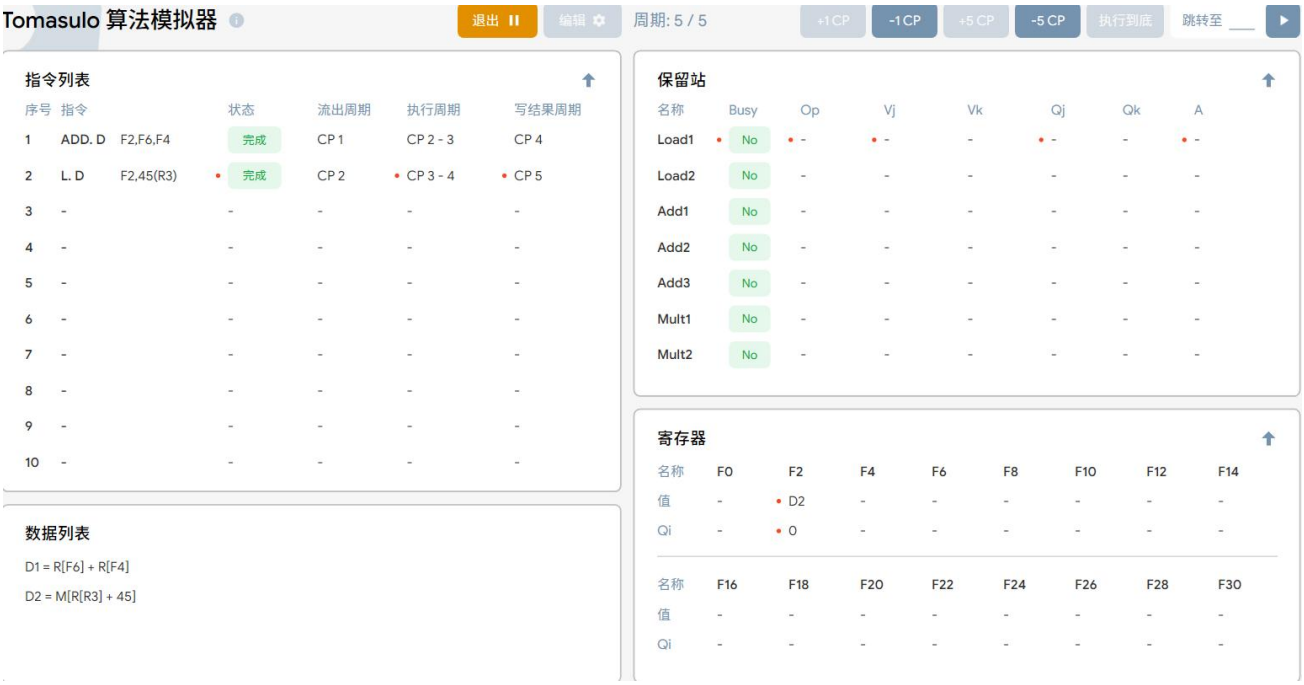
```
                                        周期4:
指令状态:
指令名          |Issue|Execution completed|Write Result|
ADDD F2,F6,F4 |    1|                  3|           4|
LD F2,45(R3)  |    2|                  4|            |

保留站内容:
保留站名称|Busy |Op   |Vj           |Vk           |Qj   |Qk   |A          |
load1     |yes  |LD   |R[R3]        |             |0    |     |R[R3]+45   |
load2     |no   |     |             |             |     |     |           |
add1      |no   |     |             |             |     |     |           |
add2      |no   |     |             |             |     |     |           |
add3      |no   |     |             |             |     |     |           |
mult1     |no   |     |             |             |     |     |           |
mult2     |no   |     |             |             |     |     |           |

寄存器状态表:
寄存器号|F0           |F2           |F4           |F6           |F8           |F10          |F12          |...|F30  |
Qi      |             |load1        |             |             |             |             |             |...|F30  |
值      |             |             |             |             |             |             |             |...|F30  |
Press any key to continue . . .
```

周期 5：

预期结果：



实际结果：

```
                                        周期5:
指令状态:
指令名          |Issue|Execution completed|Write Result|
ADDD F2,F6,F4   |    1|                  3|           4|
LD F2,45(R3)    |    2|                  4|           5|

保留站内容:
保留站名称|Busy |Op   |Vj           |Vk           |Qj   |Qk   |A          |
load1     |no   |     |             |             |     |     |           |
load2     |no   |     |             |             |     |     |           |
add1      |no   |     |             |             |     |     |           |
add2      |no   |     |             |             |     |     |           |
add3      |no   |     |             |             |     |     |           |
mult1     |no   |     |             |             |     |     |           |
mult2     |no   |     |             |             |     |     |           |

寄存器状态表:
寄存器号|F0           |F2           |F4           |F6           |F8           |F10          |F12          |...|F30  |
Qi      |             |             |             |             |             |             |             |...|F30  |
值      |             |Mem[R[R3]+45]|             |             |             |             |             |...|F30  |
```

通过对比发现两者结果一致，测试通过。

//用例 2：读后写冲突

3

MULTD F0,F2,F4

MULTD F6,F8,F0

ADDD F0,F8,F2

//用例 3：写后读冲突

3

LD F6,34(R2)

LD F6,45(R3)

ADDD F0,F2,F6

参考资料

《计算机系统结构教程（第二版）》张晨曦、王志英等编著以及教材配套的何永杰和张晨曦老师的 Tomasulo 算法模拟器