大作业 I 实验报告

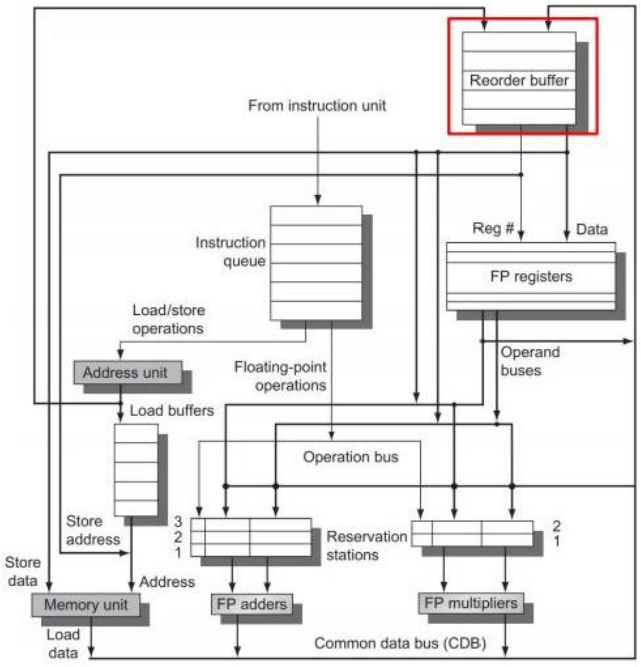
本次实验报告由：实验题目、假设部分、代码实现部分（一些废话）、实验结果及正确性分析、附加题 5 部分组成

实验题目

本次实验要求实现带有重新排序缓存的 Tomasulo 算法。重排序缓存可以让乱序执行的指令被顺序地提交，其核心思想是记录下指令在程序中的顺序，一条指令在执行完毕之后不会立马提交，而是先在 Buffer 中等待，等到前面的所有指令都提交完毕，才可以提交结果到逻辑寄存器堆。

当每个周期的指令执行完毕后，程序应逐周期输出保留站状态（reservation status）、重新排序缓存（reorder buffer）状态和寄存器结果状态（register result status）。

仿真架构如下所示：



在实验过程中，我们认为 FL load 有 1 个单元，该单元 2 个槽，可以存放 2 个 load-store 指令，FP adders 有 1 个单元，该单元有 3 个槽，可以存放 3 条 ADDD-SUBD 指令,FP multipliers 有 1 个单元，该单元有 2 个槽，可以存放 2 条 MULTD-DIVD 指令。这是基于题目要求输出 7 行寄存器状态得出的。

```
//cycle_n;
//entry1 : (busy) ,(Instruction),(State),(Destination),(Value);
//entry2 : (busy) ,(Instruction),(State),(Destination),(Value);
//entry3 : (busy) ,(Instruction),(State),(Destination),(Value);
//entry4 : (busy) ,(Instruction),(State),(Destination),(Value);
//entry5 : (busy) ,(Instruction),(State),(Destination),(Value);
//entry6 : (busy) ,(Instruction),(State),(Destination),(Value);
//Load1 : (busy) ,(op),(Vj),(Vk),(Qj),(Qk), (Dest);
//Load2 : (busy) ,(op),(Vj),(Vk),(Qj),(Qk), (Dest);
//Add1 : (busy) ,(op),(Vj),(Vk),(Qj),(Qk), (Dest);
//Add2 : (busy) ,(op),(Vj),(Vk),(Qj),(Qk), (Dest);
//Add3 : (busy) ,(op),(Vj),(Vk),(Qj),(Qk), (Dest);
//Mult1: (busy) ,(op),(Vj),(Vk),(Qj),(Qk), (Dest);
//Mult2:(busy) ,(op),(Vj),(Vk),(Qj),(Qk), (Dest);
//Reorder: f0:(status);F1:(status);.. .;F10: (status);
//Busy:f0:(status);F1:(status);.. .;F10: (status);
```

假设部分

题目给的假设如下:

1、功能单元不是流水线式的。

2、功能单元之间没有转发；结果通过公共数据总线（CDB）进行通信。

3、执行阶段（EX）既执行有效地址计算，也执行加载和存储的内存访问。因此，管道是 IF/ID/IS/EX/WB。负载需要两个时钟周期。执行有效地址计算，加载和存储的内存访问各需要一个周期，共需要两个时钟周期。

4、指令发射（IS）和结果写回（WB）阶段各需要一个时钟周期。

5、有三个加载缓冲区插槽和三个存储缓冲区插槽。

6、假设不等于零的分支（BNEZ）指令需要一个时钟周期。

7、FP 指令的部分执行延迟假设如下表所示

FP instruction	EX Cycles
fadd	2
fsub	2
fmult	10
fdvi	20

以下是我自己附加的假设和进一步说明:

1、由于功能单元**（假设 Load 也和功能单元一样）**不是流水线式的，每个单元指令需要等待前一条指令执行完（进入 write result）之后，才能进入执行阶段，前一条指令执行时，后面进来的指令只能停留在 ISSUE 阶段。比如，Input2 一开始两个 LD 指令，LD 单元只有一个，其两个槽刚好放得下两个 LD 指令，第一条 LD 指令在第 1 个周期 ISSUE，他的执行时间是第 2，3 周期。在第 4 个周期时候，第一条 LD 指令进入 write_result 阶段，第二条 LD 指令开始执行，其执行周期为第 4，5 周期。在第 6 周期进入 write_result。具体执行结果如下:

	ISSUE	EXE_COMP	WRITE_RESULT	COMMIT
LD F2 0 R2 : 1	3		4	5
LD F4 0 R3 : 2	5		6	7

其他功能单元以此类推。

2、结构冒险问题

由于 mult 单元只有 2 个槽，但在 Input2 我们发现有 3 个 MULTD-DIVID 指令需要执行，我们认为第三个 MULTD F6 F0 F2 指令，需要等第一个 DIVD F0 F4 F2 执行完 write result 指令之后，才能 ISSUE。也就是说如果 DIVD F0 F4 F2 的

执行过程是：

ISSUE EXE_COMP WRITE_RESULT COMMIT

DIVD F0 F4 F2： 3 26 27 28

那么第三个 MULTD F6 F0 F2 指令在第 28 个周期 ISSUE，也就是 DIVD 指令执行完 write_result 之后。

3、指令 commit 阶段需要一个周期

代码实现部分（一些废话）

为了提高代码的可读性设计，我们将每个表的数据封装为一个类。Instruction 类将输入的字符串分解为指令的一些有用的信息

```
#pragma once
#include<bits/stdc++.h>
using namespace std;

//定义存储指令的数据结构

class Instruction {
public:
    string op;  // 操作类型（LD, SD, DIVID, ADDD, SUB, MULTD等）
    string dest;  // 目的寄存器
    string sources1;  // 源操作数1
    string sources2;  // 源操作数2

    Instruction(std::string s);
    void print(ofstream &outputfile);
};
```

ROB_data 类存储 reorder_buffer 里面的每一条数据

```
#pragma once
#include<bits/stdc++.h>
#include"Instruction.h"
using namespace std;

// 定义Reorder buffer存储数据的结构

class ROB_data {
public:
    int entry;
    bool busy;
    string inst;  // 存储指令 eg LD F6，32(R2)
    int state;  // 状态：1表示issue,2表示execution,3表示write back,4表示commit
    string destination;  // 目的寄存器
    string value;  // 结果

    /*额外项，记录寄存器下标*/
    int source_reg1;
    int source_reg2;
    int dest_reg;

    ROB_data();
    ROB_data(Instruction instruction,int entry);
    void print(ofstream &outputfile);  // 输出该数据项
};
```

这里的构造函数将 Instruction 类改为 reorder buffer 里面存储的数据类，并提供 print 函数，封装数据的输出。可以由 reorder buffer 来调用它实现周期状态的输出。

RS_data_FP 和 RS_data_load 用来存储 reservation_station 里面的数据，设计时由于 load 需要计算地址 A 所以分开了，功能几乎一样。

```
#pragma once
#include"ROB_data.h"

// 定义存在Reservation Status add和mult单元里面的数据

class RS_data_FP {
public:
    string name;
    bool busy;
    string op;
    string vj;
    string vk;
    int Qj;
    int Qk;
    int Dest;
    string A;
    RS_data_FP();
    void print(ofstream& outputfile);
};
```

```
#pragma once
#include"ROB_data.h"

// 定义存在Reservation Status load单元里面的数据

class RS_data_load {
public:
    string name;
    bool busy;
    string op;
    string vj;
    string vk;
    int Qj;
    int Qk;
    int Dest;
    string A;
    RS_data_load();
    void print(ofstream& outputfile);
};
```

都提供了 print 函数，封装了数据的输出，可以由 reservation station 调用它实现周期状态的输出。

代码的核心控制部分时 Resevation_station 类，它负责控制每个单元的执行状态，控制每条指令从 ISSUE 到 EXECUTE，从 EXECUTE 到 WRITE RESULT 的变换，是 Tomasulo 算法的核心。

实现 Resevation_station 的伪代码如下：

Instruction state	Wait until	Action or bookkeeping
Issue FP operation	Station r empty	if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Q ← r;
Load or store	Buffer r empty	if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;
Load only		RegisterStat[rt].Qi ← r;
Store only		if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
Execute FP operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load/storestep 1	RS[r].Qj = 0 & r is head of load-store queue	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 complete	Read from Mem[RS[r].A]
Write result FP operation or load	Execution complete at r & CDB available	∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;
Store	Execution complete at r & RS[r].Qk = 0	Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;

Figure 3.13 Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the destination, rs and rt are the source register numbers, imm is the sign-extended immediate field, and r is the reservation station or buffer that the instruction is assigned to. RS is the reservation station data structure. The value returned by an FP unit or by the load unit is called result. RegisterStat is the register status data structure (not the register file, which is Regs[]). When an instruction is issued, the destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available

在 Resevation_station 类中，实现的函数有：

```
Reservation_station();
bool check_update();  // 检查是否需要更新状态
/*检查各个元件是否有状态需要更新*/
bool check_update_load();
bool check_update_add();
bool check_update_mult();
bool issue(Instruction instruction,int entry);  // 发送指令
```

```
bool cycle();  // 时钟周期，更新
// 分别实现时钟更新，返回的布尔值用于观察是否有状态更新
bool cycle_load();
bool cycle_add();
bool cycle_mult();
void write_back();  // 在周期末尾专门写一个函数写回避免冲突
void write_result(int entry, string res,bool SD);  // SD用于区分是否为SD指令

void print(ofstream& outputfile);
```

Check_update 函数检查是否有指令状态的更新，以决定是否输出当前的状态到 Output 文件中。其调用三个子函数:

```
bool Reservation_station::check_update()  // 检查是否有issue转execute，execute转write_back，A地址改变情况
{
    bool update1, update2, update3;
    update1 = check_update_load();
    update2 = check_update_add();
    update3 = check_update_mult();
    return (update1 || update2 || update3);
}
```

来实现，这三个子函数分别检查每个单元是否有状态的更新

Issue 函数处理上次的发送操作，判断是否有结构冲突并返回，如果有结构冲突，则返回 false，并不执行上层调用的 issue 操作。

Cycle 模拟时钟周期，在每个循环中调用一次 cycle。

```
bool Reservation_station::cycle() {  // 模拟一个时钟周期执行的东西
    bool update1, update2, update3;
    update1=cycle_load();
    update2=cycle_add();
    update3=cycle_mult();
    write_back();

    return (update1 || update2 || update3);
}
```

也是分为三个子函数执行，分别更新自己的指令状态。在三个单元都执行完这个 cycle 后，调用 write_back 写回。

```
void Reservation_station::write_back()  // 轮询，看是否有需要更新的
{
    if (load_write) {
        write_result(load_entry, load_res, load_SD);
        return;
    }
    if (add_write) {
        write_result(add_entry, add_res, add_SD);
        return;
    }
    if (mult_write) {
        write_result(mult_entry, mult_res, mult_SD);
        return;
    }
}
```

Write_back 根据是否有更新操作，决定 write_result 参数。最后由 write_result 函数来更新整个 Resevation_station 表。由于只有一条总线，我们每次只能广播一个数据。同时，在这个类中我们有两个指针：

```
class Reservation_station {
public:
    Reorder_buffer* rob;  // 指向reorder buffer的指针，用于协作更新状态
    Schedule* schedule;  // 用于管理schedule表
```

Write_result 函数会利用这两个指针更新 reorder buffer 和 schedule 的状态。

最后 print 函数调用下层 RS_data_load 和 RS_data_FP 的 print 函数，实现输出表的内容。

Reorder buffer 类由 Reservation station 控制

```
#pragma once
#include"ROB_data.h"
#include"Schedule.h"

class Reorder_buffer {
public:
    int* reorder;  // 指向算法的reorder数组
    bool* busy;  // 指向算法的busy数组
    Schedule* schedule;  // 用于管理schedule表

    ROB_data table[15];
    int commited;  // 指向已提交结果的下一项
    int back;  // 指向reorder buffer的最后一个元组

    Reorder_buffer();
    bool check_update();  // 检查是否需要更新表
    bool cycle();  // 每个周期检查一下是否有结果需要提交,返回的布尔值用于观察是否有状态更新
    void issue(Instruction instruction);  // 发送指令

    void print(ofstream& outputfile,int num);  // 输出表的信息
};
```

其 check_update 检查是否有表状态更新，和 Reservation station 一样的。Cycle 是每个周期检查是否有可以 commit 的指令。Issue 处理上层的 issue 操作，print 调用 ROB_data 类的 print 实现输出 reorder buffer 表的状态。这里有指向 schedule 的指针，用于控制其记录 commit 的时间。

Schedule 类记录每个指令的周期最终执行情况。其也受 Reservation station 控制。用于最后所有周期状态输出完后 输 出 一 次。输出：“(Instruction):(Issue

cycle),(Exec comp cycle),(Write result cycle),(Commit cycle)； ”

```
#pragma once
#include"Instruction.h"

/*用于记录 issue、 excute、 write result和commit的时间*/
class Schedule {
public:
    int clock;  // 记录当前时钟周期
    int num;  // 指令的数目
    int time_table[15][4];  // 存储指令状态
    string inst[15];

    Schedule();
    void issue(int entry);  // 有指令需要发送
    void write_result(int entry); // 有指令写回（execute complete在写回的前一个周期）
    void commit(int entry);  // 有指令提交
    void print(ofstream& outputfile);
};
```

Clock 变量记录当前时钟周期，方便调用时记录时间节点到 time_table 中。ISSUE 记录指令的发送时间。Write result 记录写回时间和 exec comp 的时间，commit 记录提交的时间。Print 函数输出 schedule 表的记录。

最后 main 函数负责整合这些类，时钟周期由循环模拟。每次循环，检查是否有指令可以发送，是否需要更新，调用 cycle 函数来模拟一个周期的过程。直到当所有指令都提交后，结束循环，输出 schedule。Main 函数实现较简单，就是调用下层的 Reservation station 类、reorder_buffer 类和 schedule 类就行。这三个类已经封装好了实验主要用到的三个表。提升代码可读性

实验结果及正确性分析

我们将 input1.txt 文件输出到 output11.txt 文件中，得到最终 Reservation station 表：

```
cycle_40
entry1 : No ,LD F6 34(R2),Commit,F6,Mem[34 + Reg[R2]];
entry2 : No ,LD F2 45(R3),Commit,F2,Mem[45 + Reg[R3]];
entry3 : No ,MULTD 0,F2,F4,Commit,0,Mem[45 + Reg[R3]] * Regs[F4];
entry4 : No ,SUBD F8,F6,F2,Commit,F8,(Mem[34 + Reg[R2]] - Mem[45 + Reg[R3]]) ;
entry5 : No ,DIVD F10,F0,F6,Commit,F10,Mem[45 + Reg[R3]] * Regs[F4] / Mem[34 + Reg[R2]];
entry6 : No ,ADDD F6,F8,F2,Commit,F6,((Mem[34 + Reg[R2]] - Mem[45 + Reg[R3]])  + Mem[45 + Reg[R3]]);
Load1 : No,,,,,,,;
Load2 : No,,,,,,,;
Add1 : No,,,,,,,;
Add2 : No,,,,,,,;
Add3 : No,,,,,,,;
Mult1 : No,,,,,,,;
Mult2 : No,,,,,,,;
Reorder:f0:;f1:;f2:;f3:;f4:;f5:;f6:;f7:;f8:;f9:;f10:;
Busy:f0:No;f1:No;f2:No;f3:No;f4:No;f5:No;f6:No;f7:No;f8:No;f9:No;f10:No;
```

周期：下面四个数字分别代表：(Issue cycle),(Exec comp cycle),(Write result cycle),(Commit cycle)

```
LD F6 34+ R2 : 1 3 4 5
LD F2 45+ R3 : 2 5 6 7
MULTD 0 F2 F4 : 3 16 17 18
SUBD F8 F6 F2 : 4 8 9 19
DIVD F10 F0 F6 : 5 37 38 39
ADDD F6 F8 F2 : 6 11 12 40
```

分析：运算结果均正确。对于时间表 schedule：

对于第一条指令，在第一个周期 Issue,在 2-3 周期执行（LD 指令）。第 4 个周期写回，第五个周期 commit。

第二条指令，第二个周期发送，**由于 Load 单元不是流水线式的（之前假设有提及）**，所以**他需要等待第一个 LD 指令**在第 4 个周期进入 write_result 后才开始执行，执行周期为 4-5，第 6 周期写回，第 7 周期 commit。

第三条指令：第三个周期发送，需要等待第二条指令 F2 的结果，其在第 6 周期写回，执行时间为 7-16，共 10 周期。17 周期写回，18 周期 commit。

第四条指令：第四个周期发送，需要等待第二条指令 F2 结果，其在第 6 周期写回，执行时间为 7-8（SUBD 2 周期），第 9 周期写回。Commit 需要等待上一个指令，上一条指令 18 周期 commit，此指令在 19 周期 commit。

第五条指令：第五个周期发送。需要等待上一条 MULTD 0 F2 F4 指令执行**（功能单元不是流水线式的）**。上一条 MULTD 指令在第 17 个周期进入写回阶段，DIVD 指令的执行时间为 18-37（共 20 周期），在第 38 周期写回，第 39 周期 commit。

第六条指令：第六周期发送，需要等待 SUBD 指令的 F8，其在第 9 周期写

回。执行周期为 10-11（ADDD 2 周期），第 12 周期写回。需要等待上一条指令在第 39 周期 commit，第六条指令的 commit 时间为第 40 周期。

我们将 input2.txt 文件输出到 output21.txt 文件中，得到最终 Reservation station 表：

```
cycle_52
entry1 : No ,LD F2 0(R2),Commit,F2,Mem[0 + Reg[R2]];
entry2 : No ,LD F4 0(R3),Commit,F4,Mem[0 + Reg[R3]];
entry3 : No ,DIVD F0,F4,F2,Commit,F0,Mem[0 + Reg[R3]] / Mem[0 + Reg[R2]];
entry4 : No ,MULTD F6,F0,F2,Commit,F6,Mem[0 + Reg[R3]] / Mem[0 + Reg[R2]] * Mem[0 + Reg[R2]];
entry5 : No ,ADDD F0,F4,F2,Commit,F0,(Mem[0 + Reg[R3]] + Mem[0 + Reg[R2]]);
entry6 : No ,SD F6 0(R3),Commit,F6,;
entry7 : No ,MULTD F6,F0,F2,Commit,F6,Mem[0 + Reg[R3]] / Mem[0 + Reg[R2]] * Regs[F2];
entry8 : No ,SD F6 0(R1),Commit,F6,;
Load1 : No,,,,,,,;
Load2 : No,,,,,,,;
Add1 : No,,,,,,,;
Add2 : No,,,,,,,;
Add3 : No,,,,,,,;
Mult1 : No,,,,,,,;
Mult2 : No,,,,,,,;
Reorder:f0:;f1:;f2:;f3:;f4:;f5:;f6:;f7:;f8:;f9:;f10:;
Busy:f0:No;f1:No;f2:No;f3:No;f4:No;f5:No;f6:No;f7:No;f8:No;f9:No;f10:No;
```

周期：下面四个数字分别代表：(Issue cycle),(Exec comp cycle),(Write result cycle),(Commit cycle)

```
LD F2 0 R2 : 1 3 4 5
LD F4 0 R3 : 2 5 6 7
DIVD F0 F4 F2 : 3 26 27 28
MULTD F6 F0 F2 : 4 37 38 39
ADDD F0 F4 F2 : 5 8 9 40
SD F6 0 R3 : 6 40 41 42
MULTD F6 F0 F2 : 28 47 48 49
SD F6 0 R1 : 29 50 51 52
```

分析：运算结果均正确。对于时间表 schedule：

对于第一条指令，在第一个周期 Issue,在 2-3 周期执行（LD 指令）。第 4 个周期写回，第五个周期 commit。

第二条指令，第二个周期发送，**由于 Load 单元不是流水线式的（之前假设有提及）**，所以**他需要等待第一个 LD 指令**在第 4 个周期进入 write_result 后才开

始执行，执行周期为 4-5，第 6 周期写回，第 7 周期 commit。

第三条指令：第三个周期发送，需要等待第二条指令 F4 和第一条指令 F2 的结果，F2 在第 4 周期写回，F4 在第 6 周期写回，所以在第七周期开始执行。执行时间为 7-26，共 20 周期。27 周期写回，28 周期 commit。

第四条指令：第四个周期发送，需要等待前面 DIVD 指令结束，同时**需要其产生 F0 的值**。DIVD 指令在第 27 周期进入写回阶段。MULTD 在第 28 阶段开始执行。执行时间为 28-37（MULTD 10 周期），第 38 周期写回。39 周期 commit。

第五条指令：第五个周期发送。需要等待第二条指令 F4 和第一条指令 F2 的结果，F2 在第 4 周期写回，F4 在第 6 周期写回。第五条指令的执行时间为 7-8（共 2 周期），在第 9 周期写回，前一条指令在第 39 周期 commit 因此要在第 40 周期 commit。

第六条指令：第六周期发送，需要等待第四条指令 F6，F6 在第 38 周期写回。其执行周期为 39-40（SD2 周期），第 41 周期写回。第六条指令的 commit 时间为第 42 周期。

第七条指令：MULTD 指令发送的时候，由**于 MULTD 单元只有两个槽，已经被占满，这时候发生结构冒险**。MUTLD 指令**不能发送**，需要等待第一条 DIVD F0 F4 F2 在执行完 write result 之后**（假设中有提及）**，才能 issue 指令。其 ISSUE 为第 28 周期，由于前面 MUTLD 指令没执行完，需要等待它进入写回才执行。前面 MUTLD 指令在第 38 周期写回，第七条指令执行时间为 38-47，第 48 周期写回，第 49 周期提交。

第八条指令：上一条指令在第 28 周期发送，这条指令在 29 周期 ISSUE，需要等待第七条指令的 F6，它在第 48 周期写回，第七条指令执行时间为 49-50，

51 周期写回，52 周期 commit。

综上所述，实验结果均正确，schedule 的结果符合我们根据提出的假设得到的预期结果，也是正确的。

附加题

（1） Tomasulo 算法相对于 Scoreboard 算法的优点？同时简述 Tomasulo 存在的缺点。

答：Tomasulo 成功地解决了三种冒险，实现了指令的乱序执行，且性能比记分牌更好，具体优化的地方有：

1、记分牌每条通路只能存一条指令，导致经常有指令因为结构冒险而不能发射，而 Tomasulo 引入保留站之后每条通路可以缓冲下多条指令，这样的做法平缓了指令发射的速度。

2、写后写冒险时，记分牌过度纠结寄存器名字，会把所有指令的结果都写进寄存器堆，会因为写后写冒险阻塞指令发射，而 Tomasulo 只保存最新的写入值，这样即保证了正确的结果，又减少了无谓的工作。

Tomasulo 算法是顺序发射的，即指令按照程序中的顺序一条接一条被发射到保留站。判断能否发射的唯一标准是指令对应通路的保留站是否有空余位置。记分牌为了乱序执行指令，在碰到写后写、读后写这两个冒险的时候会暂停流水线，而这其实是不必要的，因此记分牌算法还是没有最大限度地挖掘出指令的乱序潜力。重要的是“读后写”、“写后写”两个冒险是“假的冒险”，没必要为他们阻塞指令的流动。Tomasulo 通过借助重命名的思想消除了假数据冒险，从而提高了机器的乱序性能。

3、读后写冒险时，记分牌过度纠结寄存器名字，指令在执行之前一直检测

的是寄存器堆，一旦数据准备好，就会从寄存器堆中取数，这样的后果就是后序指令即使计算完结果也可能不能立刻写回寄存器堆，而 Tomasulo 则在发射时就拷贝数据，贯彻数据流的思想——“寄存器名字不重要，寄存器里的数据才重要”。

4、减小流水线停顿： Tomasulo 算法通过使用 Reservation Station 减小了数据相关的流水线停顿，允许流水线在等待数据的同时继续执行其他指令。

Tomasulo 算法存在的缺点：

1、Tomasulo 算法中每一个执行单元对应一个保留站，保留站中缓冲多条指令，所以有可能在同一周期有多条指令准备好数据，但是执行单元同时只能执行一条指令，所以就需要从中选择一条指令。电路里的 CDB 总线只有一组，这意味着每一个周期只能写回一条指令，如果同时有多条指令完成，那就只能选择一条指令进行广播，别的指令等待。

2、在当今流行的多发射处理器上实现逻辑较为复杂。需要增加寄存器堆的读写口，还要增加额外的控制逻辑使得多条指令变得有序，让它们就像单发射一样按照顺序更新保留站和寄存器结果状态表。存储指令之间也会发生数据冲突，这也需要增加控制逻辑来正确调度指令。

3、Tomasulo 算法没办法实现精确中断，精确中断是指在指令和指令之间如果出现了中断/异常，那么处理器要确保中断/异常之前的所有指令都执行完毕，而中断/异常之后的所有指令都没有执行，然后处理器把中断发生时的处理器状态给保存下来或是呈现给程序员看。要支持精确中断，就要确保指令按序提交。因此 reorder buffer 的引入就是为了解决这个问题。但就 Tomasulo 算法本身而言是无法实现精确中断的。

4、未解决的分支相关性： Tomasulo 算法并没有解决分支相关性，当发生分支预测错误时，可能需要清空流水线，导致性能下降。

5、硬件开销： Tomasulo 算法引入了大量的硬件结构，包括保留站、重命名表等，增加了硬件成本和复杂度。

6、复杂度： 由于动态调度和寄存器重命名的引入，Tomasulo 算法相对于 Scoreboard 算法更为复杂，这可能使得实现和调试变得更加困难

（2） 简要介绍引入重排序改进 Tomasulo 的原理。

答：reorder buffer 的思想：不按顺序完成指令，但在使结果对体系结构状态可见之前重新排序，ROB 的引入极大方便了 Tomasulo 算法精确的处理程序执行过程中的异常:

当指令被解码时，它会在 ROB 中保留下一个顺序条目

当指令完成时，它将结果写入 ROB 条目

当指令在 ROB 中最早并且无一例外地完成时，其结果移动到寄存器堆或存储器中

缓冲有关已解码但尚未停用（retire）/提交（commit）的所有指令的信息

正确地将指令重新排序回程序顺序

异常处理时:

使用指令的结果更新架构状态，如果指令可以毫无问题地退出

精确处理异常/中断，如果需要在停用指令之前处理异常/中断

需要有效位来跟踪结果的准备情况，并找出指令是否已完成执行

在最初的动态调度乱序执行机制中，主要分三个阶段：发射（issue）、执行

（execute）和写结果（write Result)。引入重排序的 Tomasulo 算法则增加了一个被称为“提交（Commit）”的过程。在写结果阶段，指令的结果暂时被存储在重排序缓冲区中。随后，指令执行的结果再被储存在寄存器或主存储器中。如果其他指令急切需要此结果，那么重排序缓冲区可以直接为其传输所需的数据。现在程序可能会出现，已经执行完毕，但是还没有把结果写回的情况。因为指令提交就必须循序完成，哪怕你已经执行完毕了，也得在重排序缓冲区中等待循序提交。

此时对于推测错误就很好处理了，因为哪怕已经执行了分支之后的命令，由于之前的指令还没有执行完毕，分支之后的指令就无法提交，所以很容易的将程序计数器改为正确的分支地址，而不影响其他指令结果。

重排序的具体实现方法有（参考本次实验)：引入 ROB 数组，用 Commit 指针指向顺序发送的第一条指令，该指令提交后指向顺序发射的下一条，直到顺序发射的最后一条指令提交完毕。

（3） 请分析重排序缓存的缺点。

答：重排序缓存的缺点如下：

1、资源争用： 由于处理器在乱序执行时可能会访问相同的执行单元或其他资源，因此可能导致资源争用的问题。多个指令竞争同一资源可能导致性能下降。

2、复杂性和成本： 引入重排序缓存增加了处理器的硬件复杂性。维护指令的正确顺序、确保数据相关性的正确处理以及处理异常情况等都需要额外的硬件逻辑，从而增加了成本和复杂性。

3、增加硬件资源

在基于 ROB 的 Tomasulo 算法中，一个逻辑寄存器的结果被拷贝到太多地方，数据可能存在逻辑寄存器中，也可能存在保留站中，还可能存在 ROB 中，即一个数据需要三倍于数据长度的存储空间。

另外，指令读取数据不仅通过逻辑寄存器和 CDB，还通过 ROB，这需要 ROB 配置读口，增大布线压力，且要在读取数据的线路的末尾增加选择器（把 ROB 的数据加入到选择器中），这会潜在地增加关键路径长度。在多发射的处理器中，ROB 需要支持多端口读，在一个四发射的机器里，ROB 需要支持八个读端口，压力很大。

ROB 增加了处理器的资源消耗和布线压力

4、难以预测的性能： 由于指令的执行顺序可能受到动态运行时条件的影响，重排序缓存的性能很难在编译时进行准确的预测。这增加了系统设计和调试的难度。

5、能效问题： 更复杂的硬件结构和更高的功耗可能是一个潜在的问题。虽然指令级并行性得到提高，但同时也可能引入了一些功耗和能效方面的问题。