作业A-Tomasulo算法实现

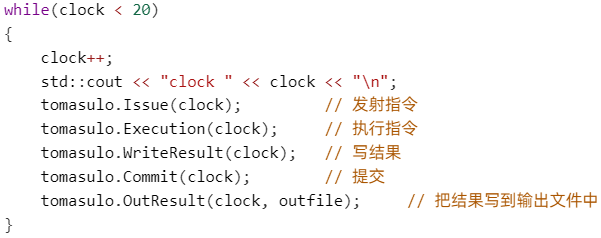
## 1 支持推测执行的双发射模拟器代码分析

文件结构：

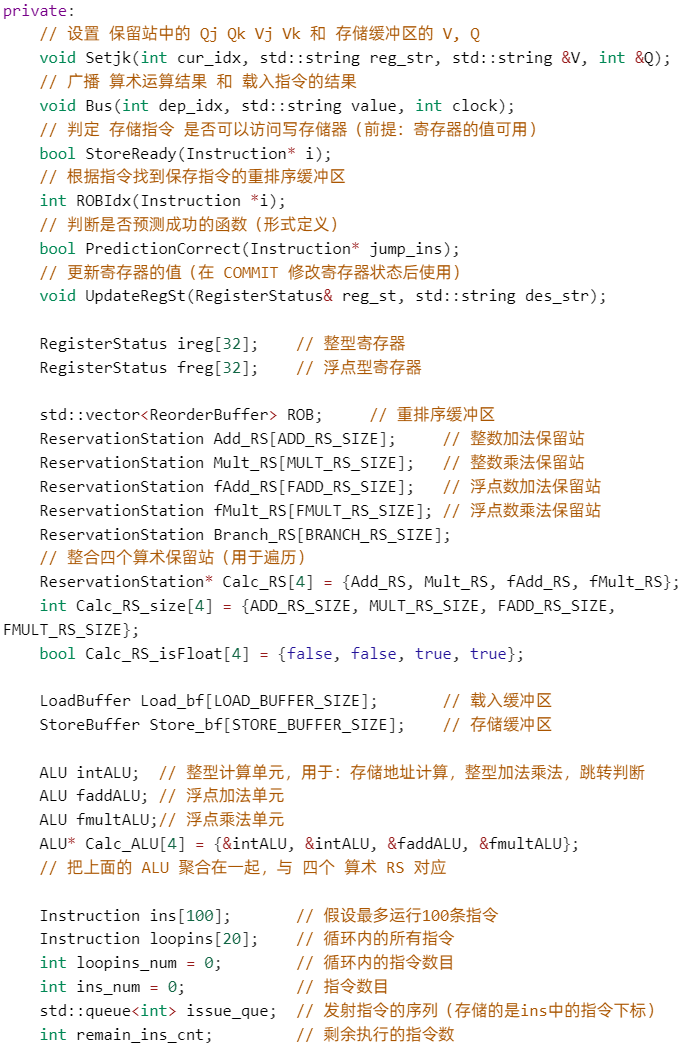
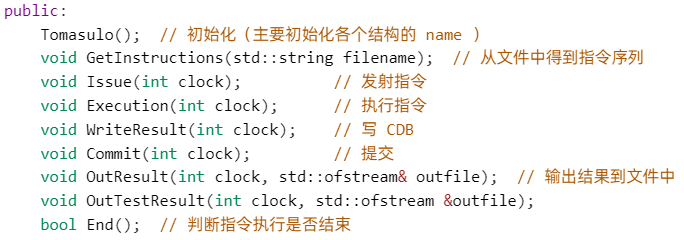


### 1.1 各个部件的数据结构

主程序的执行逻辑：

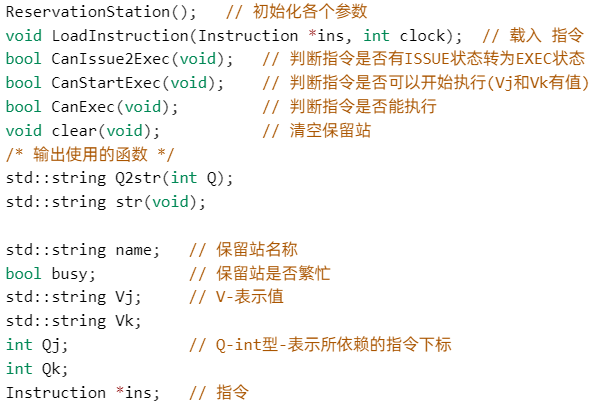


Tomasulo 算法封装的类：

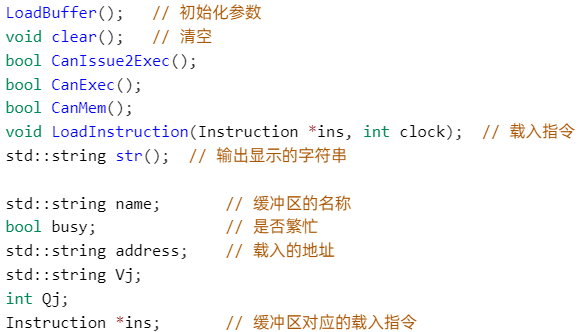


关键组件的数据结构：

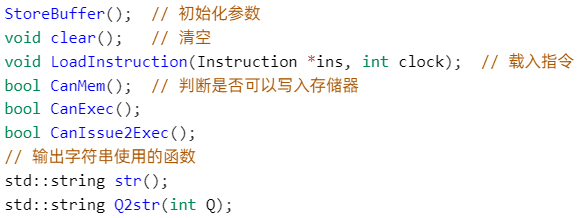
* ReservationStation：保留站结构

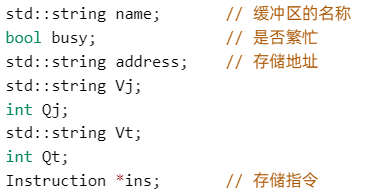


* LoadBuffer：载入缓冲区结构



* StoreBuffer ：存储缓冲区结构





### 1.2 指令处理关键逻辑的设计

**Issue函数-指令发射处理逻辑**

* 检查资源：根据指令类型，检查相应的保留站、加载缓冲区或存储缓冲区是否有空闲位置。若资源不足，暂停发射该指令，等待资源释放。
* 指令分配：将指令分配到对应的保留站或缓冲区中。同时将指令写入重排序缓冲区（ROB），以便后续追踪和提交。若寄存器需要依赖值，设置依赖关系（Q）以追踪数据状态。
* 更新状态：更新相关寄存器的状态为 “忙碌”。
* 每个时钟周期最多可以发射两条指令。

给出算术指令的发射代码，其他指令在源码中有分类实现。



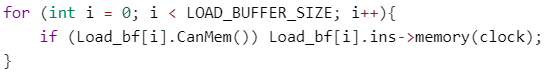
**Execution函数-指令执行代码逻辑**

**注：**根据实验要求，程序设定：一个整型计算单元负责地址计算、分支判定、整型计算。一个浮点计算单元负责浮点加法，一个浮点计算单元负责浮点乘法。（后面的实验结果显示这种硬件分配很大程度上限制了推测执行实现的吞吐量提升。）

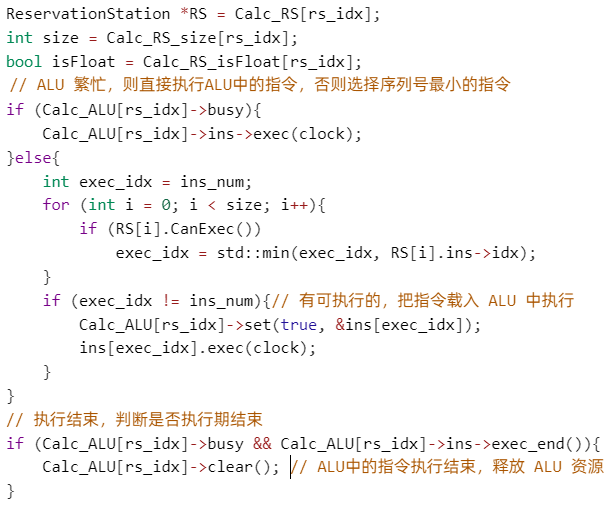
* 资源分配与执行：如果功能单元忙碌，则直接执行功能单元内的指令。如果功能单元空闲，由于每个功能单元每个周期只能处理一条指令，优先选择保留站中指令序列号最小的。
* 指令执行：运行被分配到功能单元的指令，并更新其状态。检查指令是否完成，如果完成，则释放功能单元资源。
* 载入与存储：上一个时钟周期执行结束的Load 指令可在执行阶段进行访存操作。
* 指令状态更新：对已经发射但是还未执行的指令，先判断是否具备执行条件，再将指令从 ISSUE 状态设置为EXEC状态。

核心代码：

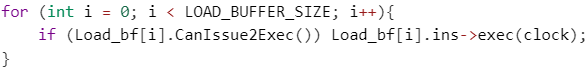
Load访存：



算术指令的执行，以浮点运算为例（浮点加法有浮点加法器，浮点乘法有浮点乘法器，所以不需要考虑其他指令）。



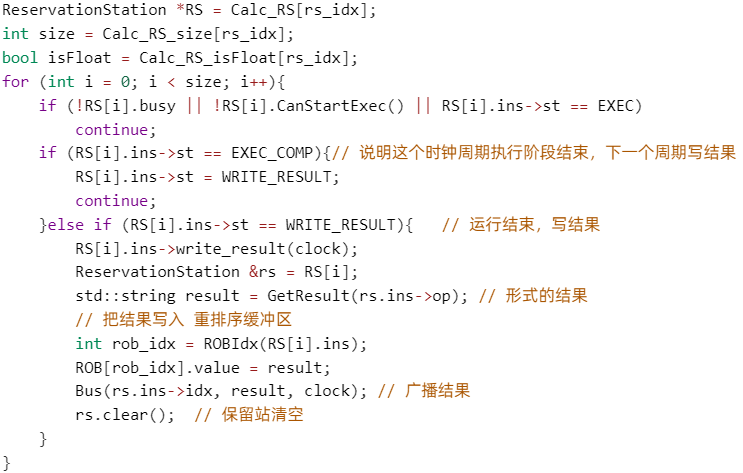
Load指令状态更新（其他指令代码逻辑相同）：



**WriteResult函数 – 写CDB代码逻辑分析**

* 遍历 Load 缓冲区和保留站，找到已完成执行的指令（状态为 EXEC\_COMP）。将指令状态更新为 WRITE\_RESULT，准备下一个时钟周期写回。
* 写回：对于状态为 WRITE\_RESULT 的指令，执行结果写回操作，包括将结果广播到 CDB ，并更新 ROB。
* 清理资源：写回后释放相关资源（如保留站或缓冲区），以便后续指令使用。

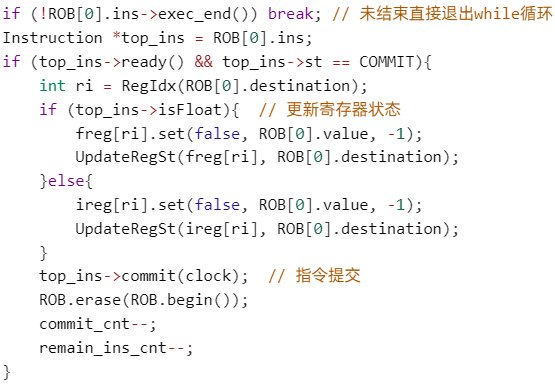
算术指令的写CDB代码：



**Commit函数 – 指令提交逻辑分析**

* 提交指令：从 ROB 顶部依次检查指令，如果已完成执行且进入提交状态，则提交指令。提交时根据指令类型执行特定操作（如写寄存器、存储器或分支判断），更新状态为 ENDING，同时从 ROB 中移除。
* 状态更新：遍历 ROB 中其他指令，检查是否满足进入提交状态的条件：算术和 Load 指令需完成写结果。Store 指令需目标值已准备好，地址计算完成。分支指令需计算完成并检查预测是否正确，若预测失败则清空 ROB 后续指令。
* 资源清理：对于提交或进入提交状态的指令，清空对应缓冲区或保留站资源，释放硬件资源供后续指令使用。

使用while循环提交指令，通过计数变量限制每个时钟周期最多提交两条指令。While循环内算术指令的提交逻辑：



对于可进入提交状态的指令的状态更新逻辑比较简单，判断结束之后直接更新状态。

## 2 指令的执行情况

### 任务1的指令执行结果

* 整型加法执行 1 个时钟周期，访存存储器需要1个时钟周期
* 浮点加法执行 2 个时钟周期
* 浮点乘法执行 6 个时钟周期

不支持推测执行：



支持推测执行：



### 2.2 任务2的指令执行结果

* 整型加法执行 1 个时钟周期（地址计算、整型运算、分支判断均在一个ALU上处理）
* 访存存储器需要1个时钟周期





支持推测执行：



## 3 推测执行改进Tomasulo算法的原理和实际效果

推测执行的原理：推测执行的核心思想是基于分支预测的结果，提前执行分支后的指令，以最大化处理器的资源利用率。其基本流程是：

* 分支预测：假定分支预测永远正确，按预测的分支结果提前发射指令。
* 动态调度：预测结果正确时，指令动态调度并执行。
* 错误处理。

推测执行引入了几个重要的概念：

* 重排序缓存（ROB）：用来缓存已发射但尚未提交的指令，通过ROB保证指令按正确顺序提交，避免异常和中断错误。
* 按序发射，乱序执行，按序提交：推测执行需要确保指令按序发射，但可以乱序执行，最后按序提交。

推测执行改进Tomasulo算法的原理

* 指令发射：指令发射时会为每个操作数分配保留站（RS）。指令的执行顺序不再严格依赖程序的顺序，而是通过分支预测结果提前执行后续指令。将指令发射后的信息保存到重排序缓存（ROB）中，以保证最终按顺序提交。
* 指令执行：执行阶段（EXEC）依赖于操作数的可用性。对于推测执行，操作数可能提前通过分支预测获取，从而启动执行。执行结果不直接写入寄存器，而是通过CDB广播到保留站，这样可以处理依赖关系，避免因预测错误带来的问题。
* 写回与提交：在推测执行中，指令结果会写入ROB，而不会立即提交，直到确认分支预测是正确的。当分支预测正确时，指令可以按顺序提交，写入寄存器或内存。如果预测错误，保留站后续的指令会被丢弃，避免影响程序的正确性。

改进后的Tomasulo算法效果

结合推测执行后，Tomasulo算法的执行过程更加高效，理论上可以实现：

* 提升吞吐量：利用分支预测结果提前执行指令，减少了程序因分支指令导致的延迟，显著提升了处理器的指令吞吐量。
* 减少延迟：通过引入重排序缓存（ROB），即使分支预测错误，系统也能通过删除错误指令后续的ROB项目来避免错误的指令影响执行结果。在指令写回时，如果分支预测错误，错误指令会被丢弃，系统能够保持正确的指令顺序。
* 提高指令级并行性（ILP）：推测执行通过跨基本块的调度，使得处理器能够在等待条件成立的同时执行其他指令，进一步提高了指令级并行性。

实际执行结果：

* 由于实验提供的指令均为整型指令，执行时间只有1个时钟周期，又由于整型的ALU只有一个，因此所有指令的执行过程轮流在一个ALU上执行，导致指令执行很紧密。即便是使用推测执行，指令执行差异不大。
* 但是，我最初给地址计算、分支判断、整型算术执行这三个功能设定分配不同的ALU部件执行的时候，在使用推测执行的时候可以很明显的看出第二次迭代的指令能够比不推测执行的指令提前几个时钟周期执行，因此，在硬件资源较多的情况下，指令吞吐量可以提升很多。

所以，硬件很大程度上限制了推测执行改进Tomasulo算法实现的吞吐量的提升程度。

## 4 计算 IPC

分别计算4组实验的IPC(Instructions Per Clock/Cycle，CPU每个时钟周期内执行的指令数)，并结合你的实验结果说明 多发射超标量 的优势和挑战。

以循环指令（实验文件中的第二个指令序列）为例执行来计算IPC。

1. 任务1：

不支持推测执行：IPC = 15/18

支持推测执行：IPC = 15/22

1. 任务2：

不支持推测执行：IPC = 15/16

支持推测执行：IPC = 15/20(以提交结束作为指令结束的标志)

由于实验提供的循环指令片段都是整型加法，导致所有指令都在一个整型单元上执行，进而导致部分本来可以提前执行的指令因为等待ALU不得不推迟执行，进而即便是在支持推测执行的情况下，IPC似乎没有增长，由于指令结束的判定标准改变并且指令执行增加了一个提交阶段，导致最后计算得到的IPC反而不如没有推测执行的。

但是就**多发射超标量**而言，双发射处理器指令执行的IPC 比单发射处理器指令执行的IPC都要高。说明多发射超标量可以提高指令执行的吞吐量。对多发射超标量的优势和挑战做出总结：

优势：

* 提高指令吞吐量：多发射超标量架构通过同时发射多条指令，可以显著提高每个时钟周期的指令执行数，从而提高了指令的吞吐量。
* 更高的指令级并行性：多发射超标量架构可以同时处理多条指令，从而有效利用多个执行单元，提升了指令级并行性（ILP）。当有多个可执行的指令时，多发射超标量能够将它们分配到不同的功能单元中进行并行执行，从而加速整体程序的执行。
* 资源利用率的提升：多发射超标量能够更好地利用处理器中的多个功能单元，避免了单个功能单元的资源瓶颈，尤其是在指令之间相互独立时，能够提高系统的并行处理能力。

挑战：

* 资源瓶颈：当指令依赖于同一功能单元（如 ALU）时，即使在多发射架构下，也可能由于资源竞争而导致性能无法提升。在我的实验中，所有指令都集中在一个 ALU 单元上执行，导致部分指令等待执行，进而影响了性能。如果资源（如 ALU、寄存器、缓存等）不足以支持多条指令的并行执行，可能会发生瓶颈，导致性能无法提升，甚至下降。
* 增加的硬件开销：发射架构通常需要更多的硬件资源来提升性能，例如更多的功能单元、寄存器组和调度单元。这不仅增加了设计和实现的复杂度，还可能导致硬件功耗的上升。

**补充**

使用任务2的不支持推测执行的代码执行arch-lec04-ILP-part3.pdf的相同代码逻辑的代码，并设定和课件相同的时钟周期数（整型1，浮点加法3）

Loop: fld f0,0(x1)

fadd f4,f0,f2

fsd f4,0(x1)

addi x1,x1,-8

bne x1,x2,Loop

可以得到和课件相同的结果：

