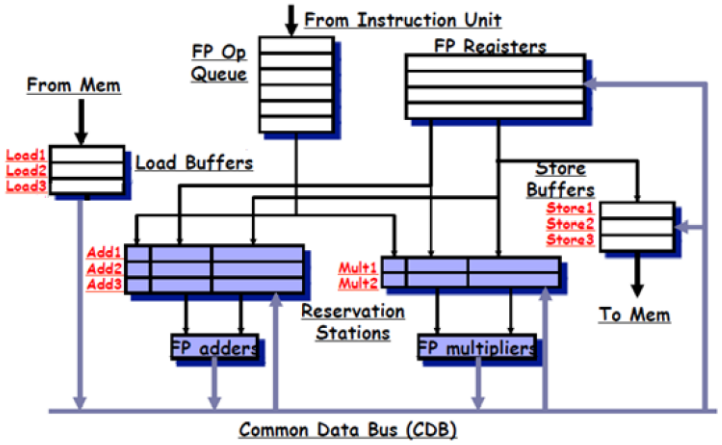
设计实现 Tomasulo 调度算法 实验报告

计 44 班 顾嘉瀚 2014011379
计 44 班 周以凡 2014011380
计 44 班 叶德铭 2014011398

实验要求

软件模拟实现 Tomasulo 调度算法，实现寄存器重命名，允许指令乱序执行。

实验要求浮点处理部件结构如下图所示：



浮点处理部件从取指单元接收指令，存入浮点操作队列。浮点操作队列每拍最多发射 1 条指令给浮点加法器或浮点乘除法器。浮点处理部件包含一个浮点加法器和一个浮点乘除法器。浮点加法器为两段流水线，输入端有三个保留站 A1、A2、A3，浮点乘除法器为六段流水线，输入端有两个保留站 M1，M2。当任意一个保留站中的两个源操作数到齐后，如果对应的操作部件空闲，可以把两个操作数立即送到浮点操作部件中执行。Load Buffer 和 Store Buffer 各缓存三条访存操作。不考虑整数寄存器。

指令细节及保留站设定如下图：

指令格式	指令说明	指令周期	保留站/缓冲队列项数
ADDD F1,F2,F3	F1, F2, F3 为浮点寄存器寄存器至少支持（F0~F10）	2 个周期	3
SUBD F1, F2, F3	同上	2 个周期	
MULD F1, F2, F3	同上	10 个周期	2
DIVD F1, F2, F3	同上	40 个周期	
LD F1, ADDR	F1 为寄存器，ADDR 为地址，0<=ADDR<4096	2 个周期	3
ST F1, ADDR	同上	2 个周期	3

算法原理

算法通过增加保留站寄存器进行换名，寄存器和保留站中同时可以记录保留站名称或是实际值。在指令进入时，从寄存器获取当前的值，值可能为某个保留站名称。运算器执行所有数据准备完毕的指令进行执行。写回阶段更新所有保留站和寄存器的值，将记录保留站名称与写回数据对应的保留站名称相同的数据更新为写回的数据，同时释放该保留站。

总体结构

算法整体以 Controller 进行统领，包含 5 类部件。

第一类部件仅有指令队列 InstrQueue，负责取指、译码和发射指令。

第二类部件仅有寄存器 Register，负责存储所有寄存器。

第三类部件为保留站/缓冲部件，包含 MemBuffer 和 CalcResSta。其中 MemBuffer 为 LD/ST 指令的缓冲区，CalcResSta 实现加减法和乘除法保留站。该部分部件负责接受 InstrQueue 给出的指令，从 Register 获取寄存器值，并向第四类部件运送数据准备完毕的指令。

第四类部件为运算器部件，包含 FlowMemory 以及两个运算器 AddExecutor 和 MulExecutor，两个运算器分别存在流水线版本 FlowAddExecutor 和 FlowMulExecutor。FlowMemory 负责处理 LD/ST 指令，并向总线发送数据。运算器负责执行运算指令，并向总线发送数据。

第五类部件为公共总线 CDB，负责接受第四类部件的数据并向第三类部件和寄存器进行写入，并且释放指令对应的保留站。

代码设计了以下接口保证了代码可扩展性。

ResSta 接口，定义了第三类部件对上层指令队列的响应函数和对下层第四类部件的指令发送函数。所有第三类部件实现此接口。

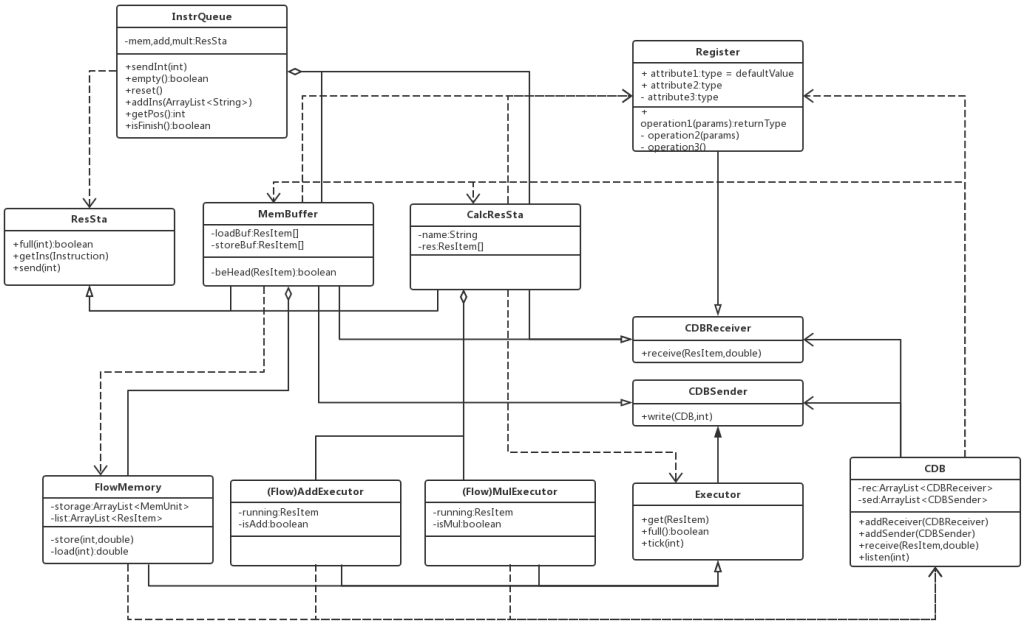
CDBReceiver 接口，定义了第三类部件和寄存器对 CDB 的响应函数。所有第三类部件和寄存器实现此接口。

CDBSender 接口，定义了部件对 CDB 发送数据的函数。所有第四类部件实现此接口的继承接口 Executor。并且，实现中将 MemBuffer 作为此接口的实现，保留了算法实现数据缓存读写的可能，但实际中未实现该函数。

Executor 接口，额外定义了第四类部件响应第三类部件发送命令的函数。所有第四类部件实现此接口。

InstrQueue、第三类部件、寄存器以及 FlowMemory 实现了 log 函数输出执行信息，供功能正确性检验。

设计 UML 图如下，未给出代码中使用的数据类型，为了图像简单，省略了接口定义的函数：



算法细节及设计特点

程序实现了 Tomasulo 算法，并在细节上进行了一定规定和简化。

简化之处包含三点：实现取消了整数寄存器，对于 LD/ST 指令的地址由立即数直接给出，方便了指令译码；实现不考虑跳转指令，不需要考虑分支预测失败的指令流修正问题；不考虑内存读写的缓存问题。

由于资料对于 Tomasulo 算法的描述并不详细，并且实验平台与硬件差异较大，因此对于算法实现行为作出了如下规定：指令发射进入保留站/缓存后，如果部件空闲，下一个周期立即开始执行该指令；保留站/缓存中的数据由 CDB 写入，指令满足执行条件后，下一个周期立即开始执行该指令；LD/ST 指令的 EX 段需要两个周期，其中 LD 指令在第二个周期进行访存，而 ST 指令在 WB 阶段进行访存，同一时刻只能有一个指令进行访存，如果产生冲突，LD 指令进行等待；CDB 一个周期只能处理一个数据请求；由于前一规定，运算器只能在指令的 WB 完成后才能继续执行下一个指令，此处假设运算器和 CDB 之间没有数据缓存。

由于流水线结构不明确，对于两运算器，设计了单指令版本和流水线版本。单指令版本假设同一时间只能有一条指令进行运算，流水线版本设计见下文**运算器流水线设计**一节。

对于以上假设和规定，最终算法会呈现如下行为：ST 后紧接的 LD 指令需要三个周期才能完成 EX 段；运算器和 FlowMemory 的数据可能由于总线抢占而等待；一条独立的 LD/ST 指令需要 4 个周期才能完整结束，一条独立的加法指令需要 12 个周期才能完整结束。

算法顶层 Controller 控制每个周期的行为。每个周期中，第一步调用运算器部件的运行，此步骤中，运算器应当已经知道自己是否对于总线具有需求。第二步 CDB 按照顺序访问运算器，调用每个运算器部件的发送函数，并且在一个运算器部件发送后停止调用，表明这个周期中，这一个运算器占用了总线。第三步由指令队列发送指令，保留站/缓冲接受指令后直接从寄存器中获取对应寄存器数值。第四部由保留站/缓冲发送指令至运算器部件。

第三步和第四步的规定使得 CDB 写回后或指令队列发送后满足数据条件的指令可以在下一个周期立即开始执行。

对于前文的假设和规定，代码进行了一定的扩展性考虑，使得规定可以较容易进行修改：修改第三步和第四步的执行顺序，可以使得保留站/缓冲需要下一个周期才能将指令发送至运算器部件；修改 CDB 中的多重总线开关，可以使得一个周期中处理多个数据发送请求；每个部件抽象为一个类，可以较容易得修改每个部件的内部逻辑。

运算器流水线设计

1. 流水加法器

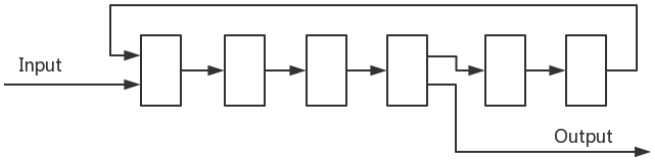
流水加法器为 2 段流水线，包含 4 个中间寄存器。首先为准备阶段寄存器，这一阶段对应硬件设计应当为保留站发送给加法器的数据暂存位置，实际硬件设计中可以不存在。然后为阶段一中间寄存器和阶段二中间寄存器，分别保存经过流水线两个阶段的数据，其中阶段二寄存器中的数据已经计算完毕，等待写回，由于计算完成位于周期末，因此需要下一个周期进行写回。最后为写回寄存器，这一寄存器用于加法器无法获得总线使用权时暂存写回数据。

流水加法器对应 FlowAddExecutor 类。程序实现中使用计算剩余时间进行记录，将指令对应至中间寄存器，实际实现没有将中间寄存器进行抽象。模拟每个周期执行时，按照执行顺序的逆向依次判断下一个阶段对应的寄存器是否被阻塞，如果阻塞则自身同样阻塞，否则执行，即剩余时间减 1。存在计算剩余时间等于指令执行时间的指令时，表示加法器无法从保留站获取新的执行指令，即加法器满。需要注意的是，由于加法保留站的大小为 3，并且加法器不存在流水循环，因此不会出现加法器满的情况。并且通过修改类中流水线段数的参

数，可以很容易得将流水线段数进行修改。

2. 流水乘除法器

流水乘除法器为 6 段流水线，包含 8 个中间寄存器。乘除法指令将在流水线中产生循环执行的情况。对于乘法指令，执行 1 个循环以及 4 个周期；对于除法指令，执行 6 个循环以及 4 个周期。流水线的大致流程如下图所示，可以看到，最后一个阶段执行的结果会返回第一个阶段进行执行，第四个阶段执行的结果可能直接进入输出环节。



流水线包含 8 个中间寄存器。首先为初始寄存器，这与流水加法器的准备阶段寄存器类似，但同样存储最后一个阶段的执行结果，如果最后一个阶段计算得到了结果，下一个周期乘除法器将不接受保留站发送的执行指令。然后为各个阶段寄存器，与流水加法器完全相同，但保存执行完毕的结果寄存器被独立出来，此部分共 6 个寄存器。最后为写回寄存器，与流水加法器完全相同。

此设计与一般乘除法器的设计并不相同，仅能满足实验书中对于乘除法器的流水阶段数以及乘除法指令执行周期的要求，但指令不执行完毕完整循环的设计不违反硬件的设计。

流水乘除法器对应 FlowMulExecutor 类。程序中使用剩余时间区分每个阶段，剩余时间 0 表示执行结束，下一个周期等待写回，-1 表示写回等待。如果存在一个执行剩余执行时间对 6 取模为 4，则表示初始寄存器被占据，无法从保留站获取执行指令，但并不表示乘除法器已经满。模拟每个周期执行时，首先判断写回是否被阻塞，将执行结束的指令送往总线写回；然后判断执行结束寄存器是否被阻塞，尝试将剩余时间为 1 的指令移动至该寄存器，如果被阻塞，则整个流水循环被阻塞，按照执行顺序逆向的方向依次判断指令能否移向下一个阶段，即剩余时间减 1，如果没有被阻塞，则在流水循环中的搜索指令剩余时间减 1。

需要注意的是，乘除法保留站大小仅有 2，因此不会出现流水线满的情况，但是仍然会因为一条指令循环执行至剩余时间对 6 取模为 4 对应的保留站，导致保留站中另一条指令无法进入流水线的情况。程序专门为此流水线设计进行实现，因此需要修改流水线参数相对复杂。

实现细节

1. 自定义类型

Instruction

该类为算法中用于存储指令的类。构造函数通过解析 String 得到指令内容。

类的所有内容均为 public 方便其他类的使用，并定义了指令类型以及对应编号的常量。

类中包含了指令原始字符串，源寄存器、目标寄存器以及指令类型，并且类型同时存储了指令名称和对应编号。对于一般指令，src0 和 src1 表示两个源寄存器，dst 表示目标寄存器。对于 LD 指令，src1 表示立即数地址，dst 表示目标寄存器。对于 ST 指令，src1 表示立即数地址，src0 表示源寄存器。

类中同时存储了指令各个阶段的完成/开始时间。阶段包含 ID、EN、EX 和 WB，分别表示译码发射完成、准备进入运算器部件、运算完成和写回完成的时间。除 EN 外，其余三者的含义与 Tomasulo 算法的定义相同。类定义了 finish(stage, cycle)函数记录阶段时间。

类同时定义了 isFinish()函数判断指令是否已经结束。该功能主要供 InstrQueue 类判断所

有指令是否全部完成。

Value

该类定义了供寄存器和保留站/缓冲使用的数值记录类型。包含一个 String 和一个 double，分别命名为 Q 和 V。当 Q 不为 null 时，Q 记录了保留站/缓冲名称，否则 V 记录了实际值。

该类提供了通过 String 和 double 的构造函数，并提供了两个类型的 setValue 进行值的修改，修改中自动维护 Q。并且通过 toString()函数可以将 Value 的值转换为字符串，统一了两个类型数据的输出。

通过 ready()函数可以判断该值是否准备完毕，即判断 Q 是否为 null。通过 wait(String)函数可以判断该 Value 等待的保留站/缓冲名称是否为给定 String，并且对于 Q 为 null 的 Value，必然返回 false。

该类的 Q 和 V 为 public，外部可以直接访问或修改，但建议通过以上函数进行操作。

ResItem

该类统一了保留站和缓冲中的数据存储格式，并且统一了 LD 和 ST 的缓冲区。

类中包含成员 name 和 busy，记录了保留站的名称和使用情况。

对于一个存储了指令的保留站/缓冲条目，ins 存储了指令的引用，value[]存储了所有源寄存器的值。

对于进入了运算器部件的指令，restTime 记录了指令 EX 段结束的剩余时间。该值在进入保留站时会被初始化为-2 表示无效；进入运算器部件后会初始化为一个非负数，表示剩余时间；EX 段结束后值变为-1 表示可以进行写回，此时如果无法获得总线，标记为 wait。同时对于这些指令，in 表示是否存在于运算器部件中，供保留站/缓冲区进行判断。这一变量的表示信息与 restTime 有所重复，属于冗余信息，但在实现中使得对应条目的含义更加明确。

所有成员变量均为 public 方便修改。

MemUnit

供 FlowMemory 存储内存数据。由二元组<addr, value>进行表示。提供了由这两个值进行构造的构造函数。二元组均为 public。

2. 接口

ResSta

函数 full(int)判断保留站/缓冲是否为空。传入参数进行一定区分，例如 MemBuffer 需要区分 load 和 store 指令的缓冲区。

函数 getIns(Instruction)表示接受一个指令并存储值保留/缓冲区。InstrQueue 进行指令发送前，会调用 full 进行判断。

函数 send(int)表示向对应的运算器部件发送一个指令，传入参数表示当前时钟周期，供 Instruction 的阶段时间标记。此函数执行后，认为 EN 阶段完成。调用 Executor 的 full()进行判断，然后 get (Instruction)进行指令的发送。

Exectuor

该接口继承自 CDBSender，表示所有运算器部件都可能需要向总线发送数据。

函数 full()判断运算器是否还能继续接受指令。

函数 get (Instruction)表示接受一个指令并运算。保留站/缓冲进行指令发送前，会调用 full 进行判断。

函数 tick(int)表示执行一个周期，传入参数表示当前时钟周期。该函数执行后，运算器部件需要完成周期的模拟，并处理不需要向总线写回的结束指令。周期模拟后，restTime 变为-1 表示指令结束，认为 EX 阶段完成。

CDBSender

函数 write(CDB,int)表示向总线写入，调用 CDB 的 receive 函数实现。传入 CDB 的引用避

免设计结构出现环状依赖。传入整数表示当前时钟周期，函数结束后认为 WB 阶段完成。该函数会由 CDB 进行调用。所有 CDBSender 的实现需要提前加入 CDB 的 Sender 队列。

函数返回值表示是否向总线发送了数据，返回值为 true 后，CDB 将停止监听，表示总线被占用。

CDBReceiver

函数 receive(ResItem,double)表示对应保留站/缓冲的指令完成，输出值为 double。所有该接口的实现需要首先加入 CDB 的 receiver 队列，由 CDB 进行调用。

3. 类

AddExecutor

接口 Executor 的实现。实现了单指令加法器，指令执行时间可以直接在类中参数进行修改。

CalcResSta

接口 ResSta 和 CDBReciever 的实现。

CDB

公共总线类。

函数 addReceiver(CDBReceiver)实现向 receiver 队列添加一个元素。

函数 addSender(CDBSender)实现向 sender 队列添加一个元素。

函数 listen(int)实现了总线监听，调用 sender 队列元素的 write 方法获得数据，并判断其返回值，判断是否需要停止监听。通过 MULTI_CDB 参数的设定可以使得总线在一个周期处理多个数据请求。

函数 receive(ResItem,val)实现了接受 sender 发送的数据，供 write 方法调用。函数会通过 receive 方法，向 receiver 队列的元素发送数据。

Controller

总控制类，控制所有部件的运行。具体实现在算法细节中给出。

FlowAddExecutor

接口 Executor 的实现。实现了流水加法器，实现参考**运算器流水线设计**一节。

FlowMemoey

访存控制类，实现了 Executor 接口，内部同时维护了内存信息。

实现了访存指令的流水操作，模拟了 LD/ST 指令需要地址计算的步骤。

FlowMulExecutor

接口 Executor 的实现。实现了流水乘除法器，实现参考**运算器流水线设计**一节。

InstrQueue

函数 sendIns(int)向保留站/缓冲发送一条指令。该函数结束后，认为 ID 阶段完成。函数调用 full(int)判断保留站/缓冲是否满，并调用其 getIns 函数进行指令发射。

函数 empty()判断所有指令是否已经发射。

函数 isFinish()判断所有指令是否已经结束。

函数 getPos()返回待发射指令编号。

函数 reset()清空指令队列以及待发射指令编号。

函数 addIns(ArrayList<String>)实现批量添加指令。

MemBuffer

访存缓冲区，将 load 和 store 的缓冲器合并实现。

实现了 ResSta 和 CDBReceiver 接口，并预留了 CDBSender 接口的实现。

由于本实验的访存地址由立即数给出，指令发射时判断了地址是否冲突，只要地址不发生冲突，访存指令可以改变顺序。考虑指令的特殊性，大部分情况不会出现这种情况。

预留 CDBSender 接口可以使得 MemBuffer 通过更改实现可以实现内存缓冲区。

MulExecutor

接口 Executor 的实现。实现了单指令乘除法器，指令执行时间可以直接在类中参数进行修改。

Register

寄存器组，实现了 CDBReceiver 接口。

函数 getValue(int)获得对应编号寄存器值的拷贝。

函数 setValue(int,String)将对应编号寄存器设置为某保留站/缓冲条目的名称。

程序及界面说明

程序界面如下图所示：

界面中各个部件如左上角名称所示。

各个保留站的 Time 表示了指令在 EX 阶段的剩余执行时间，此剩余时间并不是实际的执行剩余时间，而是在逻辑上不考虑阻塞情况，执行完执行的剩余周期数。保留站的 Busy 显示为“--”表示保留站被使用，否则显示空白。

指令队列中，ID 表示指令执行完取指 ID 段的时间；EN 为从保留站进入运算器或访存的时间，如果指令进入保留站后立即开始执行，则 EN 与 ID 相同；EX 为指令执行完 EX 段的时间；WB 为指令执行完 WB 段的时间。St 标记了当前待发射的指令，并且该指令同样会被高亮显示，指令窗口会在执行过程中自动滚动，方便查看。

加法器和乘法器展示了指令的流水情况，在乘法器中，指令循环流动，因此展示顺序与实际执行顺序有所不同，可以主要根据保留站中显示的 Time 进行查看。

在执行过程中，每个周期产生数据或指令流动将使用红点在连线上标记，红点移动方向即为流动方向。左上角按钮组右侧记录了执行周期数。如下图所示：

左上角按钮组包含了使用程序的大部分功能。Load 和 Input 按钮控制指令队列的操作，Load 提供了从文件读取指令的功能，Input 提供了以文本形式编辑指令的功能，指令中“,”（逗号字符）后不添加空格，否则认为是非法指令。

Start 按钮提供了连续执行指令的功能，Stop 按钮提供了暂停执行的功能，在执行 Start 的过程中，Stop 才可以使用，并且在当前周期执行完毕后停止。Next 按钮提供了单步执行的功能。Reset 提供了重置 PC 的功能。

内存模块未像运算器一样连入连线，这一设计是为了界面更加清晰，Load 和 Store 指令的访存未在界面上展示出来。内存模块名称右侧的按钮可以添加或修改一个内存条目，内存条目为（地址，数值）的二元组，指令产生的写入会自动进行添加。未添加的内存地址数据默认为 0。

程序测试

此处展示几个简单的测例用于说明程序的正确性以及声明部分设计行为。

第一个测试用例说明了访存的行为，结果如下图。指令的每一条都依赖于上一条指令，第 4 条指令由于访存地址冲突，需要等待第 3 条指令首先访存。在访存的流水中，第 18 个周期 ST 指令在 WB 段访存，但 LD 在 EX 的第二个周期访存，因此 LD 指令需要等待一个周期，在第 19 个周期才能成功访存，结束 EX 段的执行。

Ins	ID	EN	EX	WB
LD F0,80	1	1	3	4
MULD F4,F0,F2	2	4	14	15
ST F4,80	3	15	17	18
LD F0,80	4	16	19	20
MULD F4,F0,F2	5	20	30	31
ST F4,80	6	31	33	34

第二个测试用例说明算法执行结果的正确性，结果为下图。执行的结果与课件中给出的结果完全一致。

Ins	ID	EN	EX	WB
LD F6,34	1	1	3	4
LD F2,45	2	2	4	5
MULD F0,F2,F4	3	5	15	16
SUBD F8,F6,F2	4	5	7	8
DIVD F10,F0,F6	5	16	56	57
ADDD F6,F8,F2	6	8	10	11

第三个测试用例说明流水乘除法器的行为以及保留站满的情况，结果为下图。四条指

令分别执行了各自所需的时间，并且时间重叠，符合流水线的设计。指令的 ID 时间说明保留站满后指令停止了发射。第三条指令发射后没有立即送往运算的原因是，此时第一条指令所处的阶段阻塞了新指令的进入，因此第三条指令等待了一个周期。

Ins	ID	EN	EX	WB
DIVD F0,F8,F9	1	1	41	42
MULD F1,F8,F9	2	2	12	13
DIVD F2,F8,F9	13	14	54	55
MULD F3,F8,F9	42	42	52	53

第四个测试用例说明了 Tomasulo 算法解决 WAR 冲突的情况，结果为下图。第三条指令提前于第二条指令开始 EX 段的时间执行完毕。测试用例来源为百度百科。

Ins	ID	EN	EX	WB
MULD F3,F0,F1	1	1	11	12
MULD F4,F2,F3	2	12	22	23
ADDD F2,F0,F6	3	3	5	6

第五个测试用例说明了 Tomasulo 算法解决 WAW 冲突的情况，结果为下图。第四条指令首先进行了 F0 的写入，而后第三条指令才开始执行，但 F0 的数据是从第一条指令获取的，不会产生冲突。如果将内存区域进行设置，并添加 LD 指令将 F2,F4,F6,F6,F12,F14 的值分别赋值为 2,4,6,8,12,14，执行给出代码，中间结果见下第二张图，第三条指令获取的 F0 为 6，但此时第四条指令已经执行完毕，F0 实际值已经为 26。

Ins	ID	EN	EX	WB
ADDD F0,F2,F4	1	1	3	4
MULD F2,F6,F8	2	2	12	13
MULD F10,F0,F2	3	13	23	24
ADDD F0,F12,F14	4	4	6	7

Name	Busy	Ins	Time	Val1	Val2
Mult1	--	MULD F2,F6,F8	4	6.000000	8.000000
Mult2	--	MULD F10,F0,F2		6.000000	Mult1

实验环境及其他说明

程序使用 java 实现，界面使用 javaFX 实现。运行建议使用 javaSE-1.8。

包 assembly 下为算法逻辑实现部分，具体已经在前文说明。

包 tomasulodisplay 为界面展示部分，类基本与 assembly 中对应，主要为界面的实现，不在此过多说明。

包 type 为算法逻辑实现部分使用的自定义类，具体已经在前文说明。

文件夹 testcase 给出了程序测试用例的文件。

实验分析

1. 算法效果分析

Tomasulo 算法通过寄存器换名等操作，提高了指令并行执行的程度。对比 MIPS，这样的思路与之十分相似，一定程度实现了指令流水并允许了乱序执行。在实现中发现，EX 阶段是算法中无法控制的部分，但是从抽象意义看，此部分与算法无关，只需要提供接口，供保留站和总线判断是否能够发送运算数据以及是否能够接受结果数据，因此 Tomasulo 算法可以甚至可以作为 CPU 整体框架，只需要添加整数运算器即可。

在阶段划分上，算法将指令的取指和发射放在了 ID 阶段，并且进入保留站的指令可以立

即进入运算器运算，这一点从结构上看并不合理，将其划分为 IF,ID,EX,WB 四个阶段更加合适，否则难以处理总线写回的情况。

从实验的实现看，此算法很难处理在分支预测失败情况下的指令取消。

2. 硬件实现分析

在实现中，注意到保留站同时存储了数据的情况。对于保留站名称和数据的区分问题，课件中将其区分为 Q 和 V，实际只需要 1 比特进行区分即可，数据部分可以共用。但保留站过大会导致额外硬件资源的消耗，而过小会导致算法效果不明显，因此实际设计中的具体参数与实验必然相差较大。另一方面，每个保留站都意味着总线增加一个接受端点，使得总线扇出较大。

进入保留站的指令可以在同一个周期立即进入运算器运行，这一假设在硬件上并不合理，这会使得一个周期的时间加长，降低主频。

3. 访存问题

此实验中，对 Load 和 Store 指令进行了一定假设，即只有对同一个地址访问的指令才需要保证访存执行顺序，但实际上这一点是很难判断的。另一方面，实际 CPU 执行，访存将成为一个很大的瓶颈，因此实际算法应当默认存储器是 Cache，在 Store 后的 Load 指令可以快速获取访存的值，加速整个 CPU 的运行。

4. 可能改进

程序实现中，对于总线使用等情况直接使用了从头至尾的遍历，这可能使得指令较多时，某一部件的写回一直被阻塞，或者某一部件出现饥饿。这一点不是实验的重点，可以通过实现循环遍历的方式改进。