计算机体系结构 Lab6 实验报告

PB20111686 黄瑞轩

1 Tomasulo 算法模拟

Q1：分别截图（当前周期 2 和 3），请简要说明 load 部件做了什么改动

A1：当前周期 2，load 部件的改动是：

（1）第一条 LD 指令计算目标地址；

（2）第二条 LD 指令 issue，占领 Load2 部件。

当前周期 3，load 部件的改动是：

（1）第一条 LD 指令获得了值，并将值填入 Load1 部件；

（2）第二条 LD 指令计算目标地址。

图 1　当前周期 2 的系统状态

图 2　当前周期 3 的系统状态

Q2：请截图（MUL.D 刚开始执行时系统状态），并说明该周期相比上一周期整个系统发生了哪些改动（指令状态、保留站、寄存器和 Load 部件）

A2：该周期相比上一周期整个系统发生的改动是：

（1）MUL.D 和 SUB.D 指令同时进入执行阶段，ADD.D 指令 issue；

（2）保留站中，ADD.D 指令 issue 占领 Add2 部件，相关值被填入 Add2 相应的位置中；MUL.D 开始执行，相应 Mult1 部件左侧开始计时；

（3）寄存器中，因为 ADD.D 指令 issue，其目标寄存器被占领标记；

（4）load 部件无变化。

图 4　MUL.D 刚开始执行时的系统状态

Q3：简要说明是什么相关导致 MUL.D 流出后没有立即执行

A3：是**数据相关（RAW）**。在当前周期 4 开始时，F2 的值还没有读出，F2 作为 MUL.D 指令的源操作数，导致了 MUL.D issue 后没有立即执行。

Q4：请分别截图（15 周期和 16 周期的系统状态），并分析系统发生了哪些变化

A4：当前周期 15，系统发生的变化是：

（1）MUL.D 指令完成了执行过程，准备写回；

当前周期 16，系统发生的变化是：

（1）MUL.D 指令完成了写回，解除对 Mult1 部件的占领；

（2）写回通知了总线，F0 的值广播并被 Mult2 部件源操作数接收填入。

图 5　当前周期 15 的系统状态

图 6　当前周期 16 的系统状态

Q5：回答所有指令刚刚执行完毕时是第多少周期，同时请截图（最后一条指令写 CBD 时认为指令流执行结束）

A5：所有指令执行完毕是第 57 周期，截图如图 7 所示。

图 7　所有指令执行完毕时的系统状态

2 监听法模拟

Q1：利用模拟器进行下述操作，并填写下表

A1：

所进行的访问	替换？	写回？	监听协议进行的操作与块状态改变
CPU A 读第 5 块	N	N	CPU A Read Miss，从存储器中读第 5 块，状态变为共享
CPU B 读第 5 块	N	N	CPU B Read Miss，从存储器中读第 5 块，状态变为共享
CPU C 读第 5 块	N	N	CPU C Read Miss，从存储器中读第 5 块，状态变为共享
CPU B 写第 5 块	N	N	CPU B Write Hit，写第 5 块（Cache B 中第 1 块），向总线发送作废其他 Cache 中该块，Cache B 该块状态变为独占
CPU D 读第 5 块	N	Y	CPU D Read Miss，Cache B 中第 5 块写回，状态变为共享；Cache D 从存储器中读第 5 块，状态变为共享
CPU B 写第 21 块	Y	N	CPU B Write Miss，从存储器中读第 21 块，用之替换 Cache B 第 1 块的位置，然后写，然后状态变为独占
CPU A 写第 23 块	N	N	CPU A Write Miss，从存储器中读第 23 块，然后写，然后状态变为独占
CPU C 写第 23 块	N	Y	CPU C Write Miss，Cache A 中第 3 块写回，状态变为无效；Cache C 从存储器中读第 23 块，状态变为独占
CPU B 读第 29 块	Y	Y	CPU B Read Miss，Cache B 将第 1 块内容写回存储器，然后从存储器中读第 29 块，用之替换 Cache B 第 1 块的位置，状态变为共享
CPU B 写第 5 块	Y	N	CPU B Write Miss，Cache B 从存储器中读第 5 块，用之替换 Cache B 中第 1 块的位置，然后写，然后状态变为独占，然后向总线发送作废其他 Cache 中该块，Cache D 该块状态变为无效

Q2：请截图，展示执行完以上操作后整个 cache 系统的状态

A2：

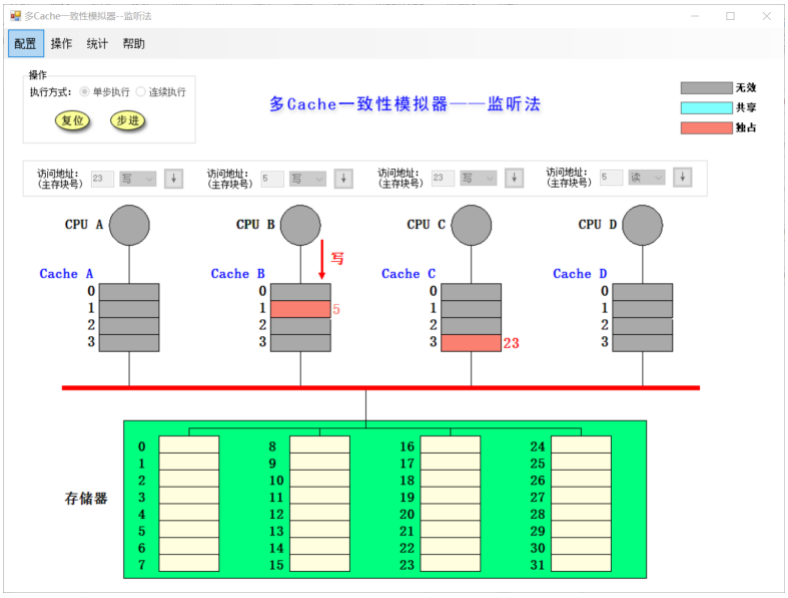

图 8　执行完操作后整个 cache 系统的状态

3 目录法模拟

Q1：利用模拟器进行下述操作，并填写下表

A1：

所进行的访问	监听协议进行的操作与块状态改变
CPU A 读第 6 块	CPU A Read Miss，存储器向 CPU A 发送第 6 块，Cache A 第 2 块状态变为共享；存储器目录第 6 块状态变为共享，共享集合为{*A*}
CPU B 读第 6 块	CPU B Read Miss，存储器向 CPU B 发送第 6 块，Cache B 第 2 块状态变为共享；存储器目录第 6 块状态不改变，共享集合为{*A*, *B*}
CPU D 读第 6 块	CPU D Read Miss，存储器向 CPU D 发送第 6 块，Cache D 第 2 块状态变为共享；存储器目录第 6 块状态不改变，共享集合为{*A*, *B*, *D*}
CPU B 写第 6 块	CPU B Write Hit，向存储器发送写命中消息，存储器作废 Cache A、Cache D 相应块（状态变为无效），Cache B 写第 6 块，状态变为独占，存储器目录第 6 块状态变为独占（*B*）
CPU C 读第 6 块	CPU C Read Miss，存储器先从 Cache B 中取回第 6 块数据，Cache B 和存储器中相应块状态都变为共享，然后向 CPU C 发送第 6 块，Cache C 中相应块状态变为共享；存储器目录第 6 块共享集合为{*B*, *C*}
CPU D 写第 20 块	CPU D Write Miss，向存储器发送写不命中消息，存储器向 Cache D 发送第 20 块，Cache D 写第 0 块，状态变为独占，存储器目录第 20 块状态变为独占（*D*）
CPU A 写第 20 块	CPU A Write Miss，向存储器发送写不命中消息，存储器从 Cache D 取第 0 块内容并作废（状态变为无效），然后向 Cache A 发送第 20 块，Cache A 写第 0 块，状态变为独占，存储器目录第 20 块状态变为独占（*A*）
CPU D 写第 6 块	CPU D Write Miss，向存储器发送写不命中消息，存储器向 Cache D 发送第 6 块，并作废 Cache B、Cache C 中相应块（状态变为无效）；Cache D 写第 2 块，相应状态变为独占；存储器目录第 6 块状态变为独占（*D*）
CPU A 读第 12 块	CPU A Read Miss，向存储器发送读不命中消息，向存储器写回 Cache A 第 0 块的内容并通知共享集合修改，存储器目录第 20 块状态共享集合变为未缓冲；存储器向 Cache A 发送第 12 块，Cache A 第 0 块状态变为共享，存储器目录第 12 块状态变为共享，共享集合为{*A*}

Q2：请截图，展示执行完以上操作后整个 cache 系统的状态

A2：

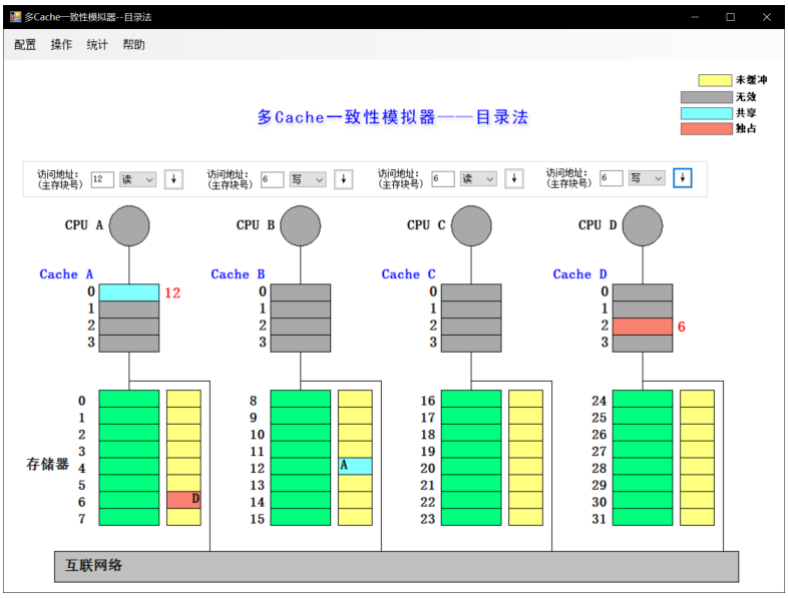

图 9　执行完操作后整个 cache 系统的状态

4 综合问答

Q1：目录法和监听法分别是集中式和基于总线，两者优劣是什么？

A1：

	目录法	监听法
优点	目录省去了广播，在多核处理器中优势明显；可以更好地控制并发访问，目录可以知道每个缓存块的状态并且可以维护一个访问序列	CPU 核心较少时，总线压力小，保持一致性的成本低，效率高
缺点	中心目录可能成为性能瓶颈，并且需要更多的存储硬件支持	在高负载情况下可能会出现总线竞争问题，因为多个处理器可能会同时访问同一个缓存块

Q2：Tomasulo 算法相比 Score Board 算法有什么异同？

A2：Tomasulo 算法和 Score Board 算法都是用于实现指令级并行的动态调度技术。它们的主要区别在于它们如何管理处理器中的功能单元和寄存器，以及如何进行指令间的冲突检测和解决。

相同点：

（1） 都支持指令级并行，可在多个指令间共享处理器中的功能单元和寄存器；

（2） 都使用了一个中央调度单元来调度指令（Tomasulo 算法引入了公共数据总线用于在多个功能单元之间传递数据，以实现更高效的指令调度）。

不同点：

（1） 在 Score Board 算法中，每个功能单元和寄存器都有一个状态位，用于跟踪它们是否正在被占用。这些状态位由中央调度单元维护，并根据指令的执行情况进行更新。在 Tomasulo 算法中，每个功能单元都有一个缓冲区，其中包含指令的操作码和操作数。这些缓冲区由功能单元本身维护，并通过公共数据总线进行交换。缓冲区中的指令可以执行，只要它们所需要的操作数已经就绪；

（2） 在 Score Board 算法中，每个指令都需要等待它所需要的操作数可用后才能执行。这可能会导致一些指令之间的 Bubble，从而降低处理器的效

率。在 Tomasulo 算法中，指令可以立即开始执行，只要它们所需的操作数已经就绪。这可以减少空闲周期，并提高处理器的效率。此外，Tomasulo 算法还支持指令重排序，可以进一步提高处理器的效率。

（3） Tomasulo 算法相比 Scoreboard 算法有更高的效率和更好的可扩展性。但它需要更多的硬件支持和更复杂的实现。

Q3：Tomasulo 算法是如何解决结构、RAW、WAR 和 WAW 相关的？

A3：Tomasulo 算法通过使用寄存器重命名和动态调度技术来解决这些相关。

（1） 结构相关

Tomasulo 算法通过对每个指令的操作数进行寄存器重命名来解决结构相关。每个指令都被分配一个单独的标识符（保留站名字），并且每个标识符都与一个重命名寄存器相关联。当指令需要一个操作数时，它将使用重命名寄存器而不是实际的物理寄存器。这样，每个指令都可以在不受其他指令影响的情况下独立地执行。

（2） RAW 相关

Tomasulo 算法通过使用寄存器重命名和动态调度技术来解决 RAW 相关。当一个指令需要一个操作数时，它将从重命名寄存器中读取该操作数。如果该操作数还没有就绪，那么指令将等待直到该操作数变为可用。这样，每个指令都可以在不受 RAW 相关影响的情况下独立地执行。

（3） WAR 相关

Tomasulo 算法通过使用寄存器重命名来解决 WAR 相关。Score Board 算法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。

（4） WAW 相关

Tomasulo 算法通过使用寄存器重命名来解决 WAW 相关。Score Board 算法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。