实验六 Tomasulo 和 cache 一致性

一、实验目的

1. 熟悉 Tomasulo 模拟器和 cache 一致性模拟器（监听法和目录法）的使用
2. 加深对 Tomasulo 算法的理解，从而理解指令级并行的一种方式——动态指令调度
3. **掌握 Tomasulo 算法在指令流出、执行、写结果各阶段对浮点操作指令以及 load 和 store 指令进行什么处理；给定被执行代码片段，对于具体某个时钟周期，能够写出保留站、指令状态表以及浮点寄存器状态表内容的变化情况**
4. 理解监听法和目录法的基本思想，加深对多 cache 一致性的理解
5. **做到给出指定的读写序列，可以模拟出读写过程中发生的替换、换出等操作，同时模拟出 cache 块的无效、共享和独占态的相互切换**

二、Tomasulo 算法模拟器

2.1 题目

使用模拟器进行以下指令流的执行并对模拟器截图、回答问题

```
L.D     F6, 21(R2)
L.D     F2, 0(R3)
MUL.D   F0, F2, F4
SUB.D   F8, F6, F2
DIV.D   F10, F0, F6
ADD.D   F6, F8, F2
```

假设浮点功能部件的延迟时间：加减法2个周期，乘法10个周期，load/store2个周期，除法40个周期。

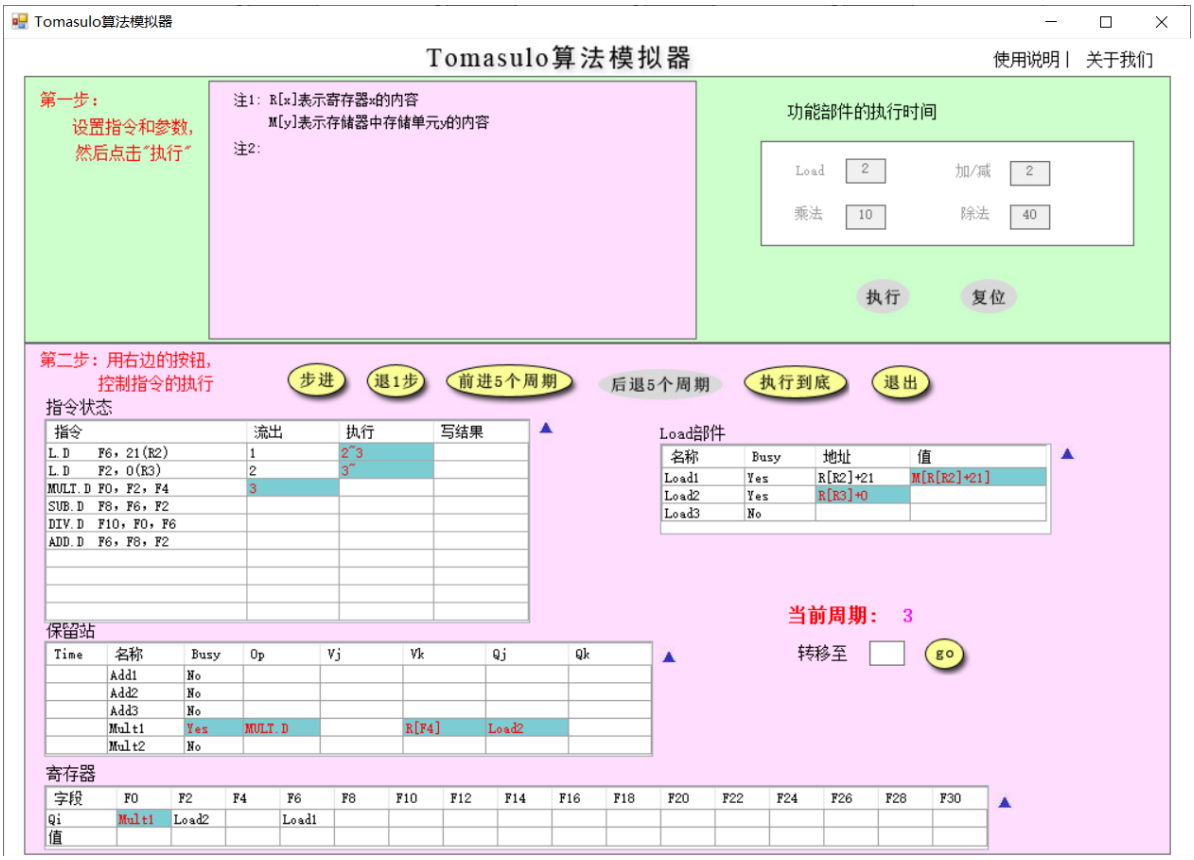
2.2 问题回答

1、分别截图（当前周期 2 和当前周期 3），请简要说明 load 部件做了什么改动

当前周期 2:

load1 和 load2 均为 busy，load1 得到存储器地址，load2 还没开始执行。

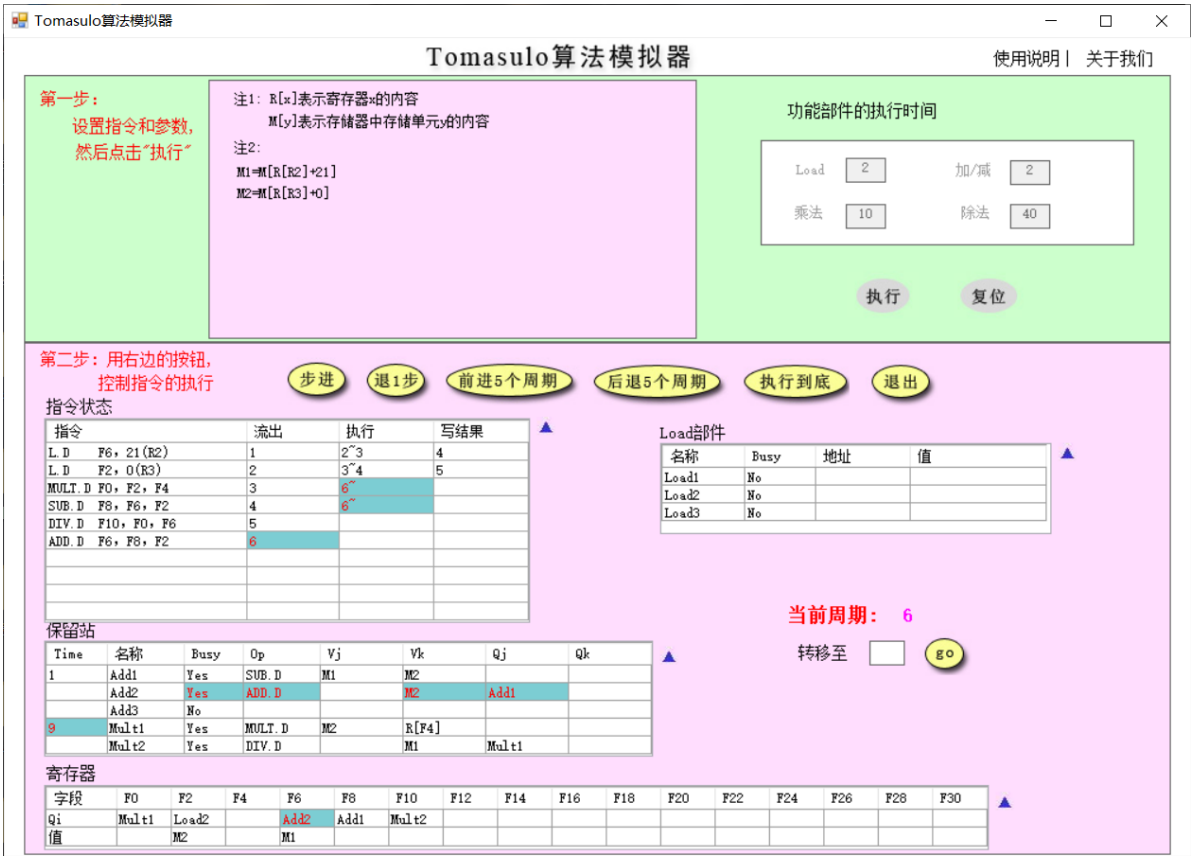
当前周期 3：



load1 部件和 load2 部件均为 busy，load1 获得了结果，尚未放到 BUS 上，load2 部件开始执行，计算出了存储地址。

2、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）

MUL.D 刚开始执行时，为第 6 个周期：



相比于上一个周期：

指令状态：MUL.D 和 SUB.D 开始执行，ADD.D 流出

保留站：Add2 部件变为 Busy，Op 设置为 ADD.D，Vj 当前不可获取，等待 Add1，Vk = M2

寄存器：F6 的值变成 Add2 计算结果

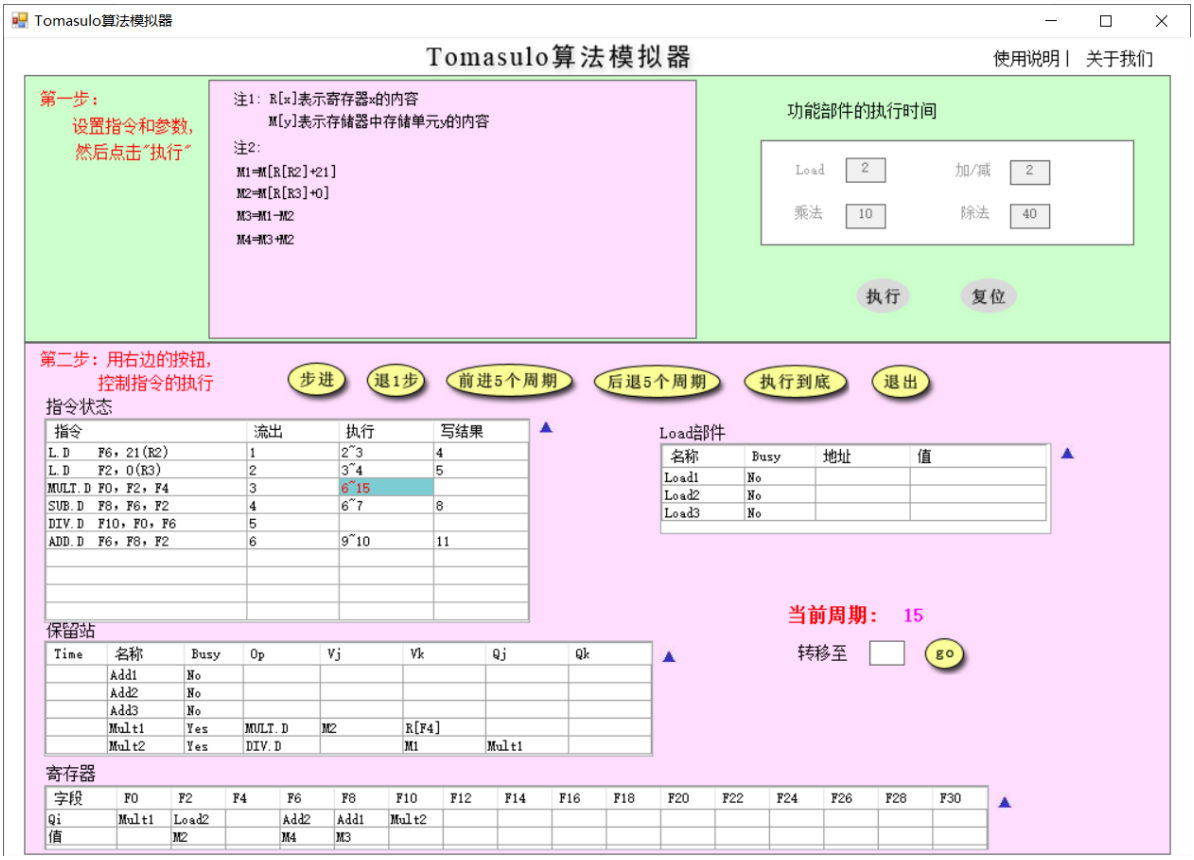
Load 组件：没有变化

3、简要说明什么相关导致 MUL.D 流出后没有立即执行

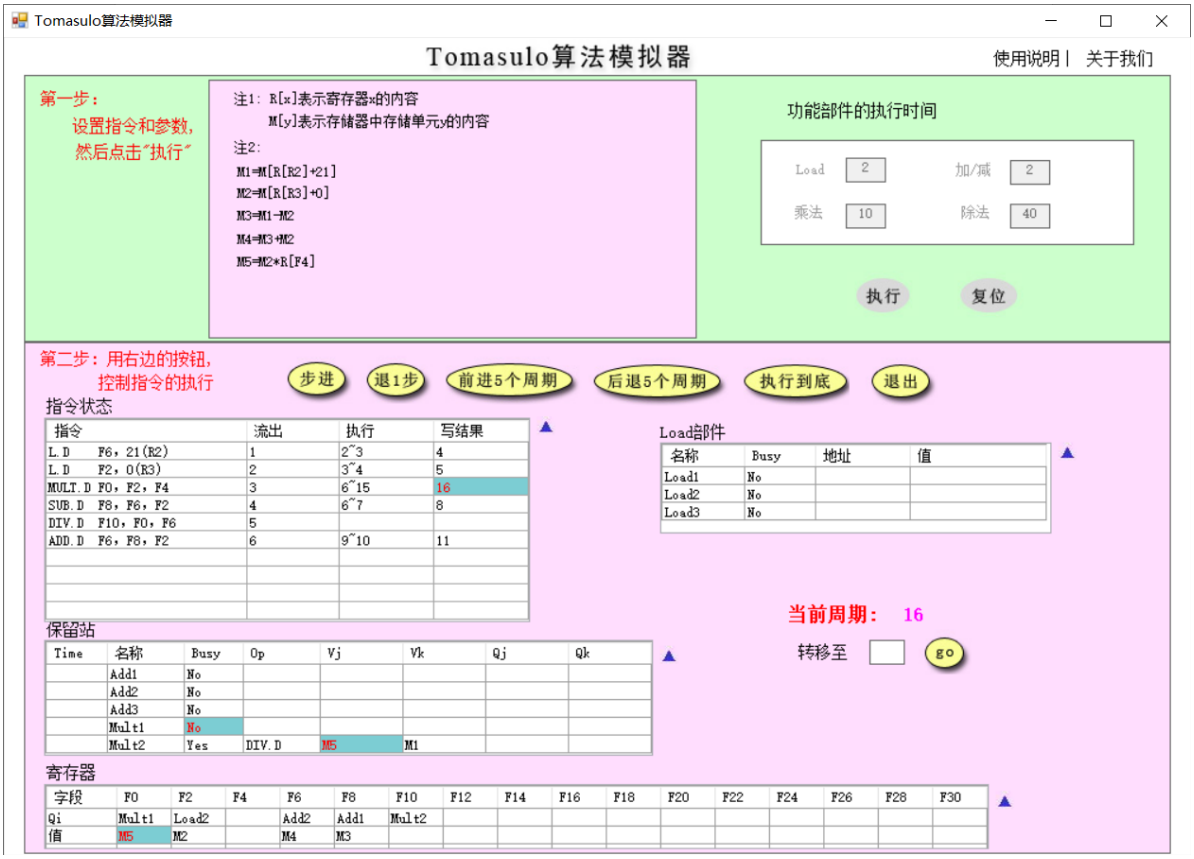
与 `L.D F2, 0(R3)` 指令的 RAW 数据相关。

4、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

15 周期：



16 周期：



指令状态：MUL.D 执行完成，将结果放到 BUS 上

保留站：Mult1 的 Busy 由 Yes 变为 No，Mult2 组件的 Vj 变为已知，为 M5

寄存器：F0 的值变为 M5

Load 组件：没有变化

5、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）

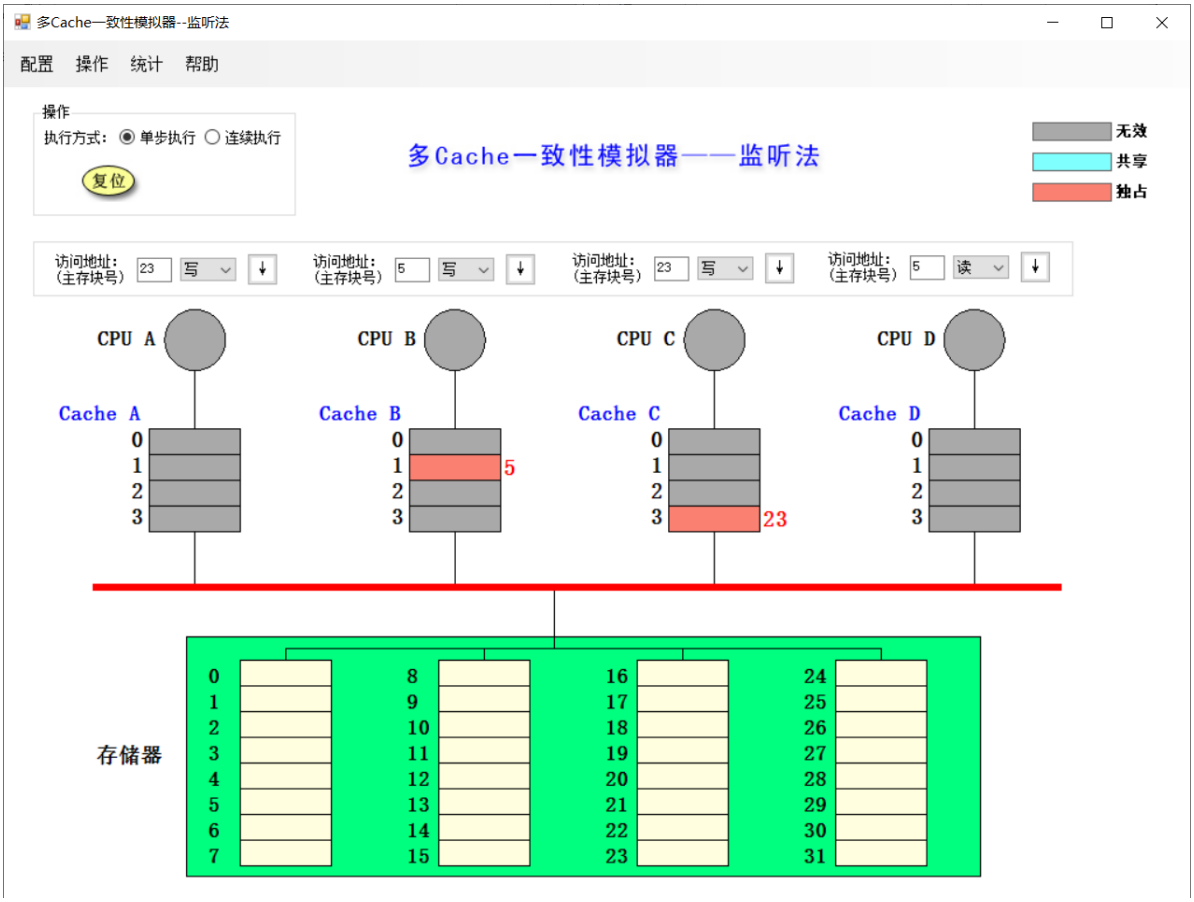
是第 57 周期

三、多 cache 一致性算法——监听法

3.1 利用模拟器进行下述操作，并填写下表

所进行的访问	是否发生了替换?	是否发生了写回?	监听协议进行的操作与块状态改变
CPU A 读第 5 块	否	否	向 BUS 传读不命中信号，从存储器中换入，更改 A 的块 5 为共享状态
CPU B 读第 5 块	否	否	向 BUS 传读不命中信号，从存储器中换入，更改 B 的块 5 为共享状态
CPU C 读第 5 块	否	否	向 BUS 传读不命中信号，从存储器中换入，更改 C 的块 5 为共享状态
CPU B 写第 5 块	否	否	向 BUS 传作废信号，更改 A 和 C 的块 5 为无效，B 的块 5 为独占
CPU D 读第 5 块	否	是	向 BUS 传读不命中信号，B 将块 5 写回存储器，再将块 5 从存储器换入到 D 的 Cache，B 和 D 的块 5 的状态均变为共享
CPU B 写第 21 块	是	否	向 BUS 传写不命中信号，将块 21 换入 B，块 5 换出，B 的块 21 变为独占
CPU A 写第 23 块	否	否	向 BUS 传写不命中信号，将块 23 换入 A，A 的块 23 变为独占
CPU C 写第 23 块	否	是	向 BUS 传写不命中信号，A 将块 23 写回到内存，再将块 23 换入到 C，C 的块 23 变为独占
CPU B 读第 29 块	是	是	向 BUS 传读不命中信号，B 将块 21 写回存储器，将块 29 换入 B，B 的的块 29 变为共享
CPU B 写第 5 块	是	否	向 BUS 传写不命中信号，B 将块 29 换出，将块 5 换入 B，B 的块 5 变为独占，D 的块 5 变为无效

3.2 截图，展示执行完以上操作后整个cache系统的状态

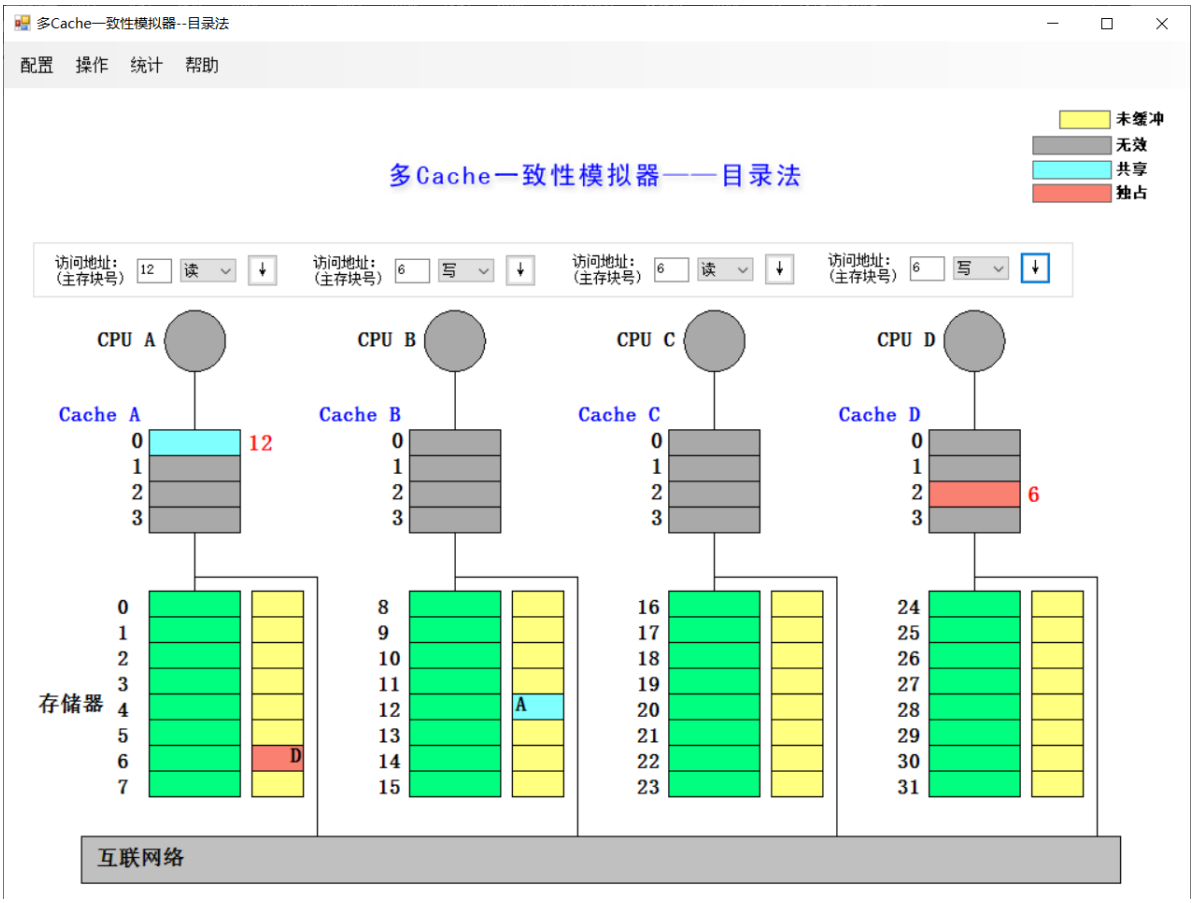


四、多 cache 一致性算法——目录法

4.1 利用模拟器进行下述操作，并填写下表

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	向存储器发送读不命中 (A, 6) 消息，将块 6 发送到 A，A 的块 6 为共享，存储器中的块 6 共享集合为 {A}
CPU B 读第 6 块	向存储器发送读不命中 (B, 6) 消息，将块 6 发送到 B，B 的块 6 为共享，存储器中的块 6 共享集合为 {AB}
CPU D 读第 6 块	向存储器发送读不命中 (D, 6) 消息，将块 6 发送到 D，D 的块 6 为共享，存储器中的块 6 共享集合为 {ABD}
CPU B 写第 6 块	向存储器发送写命中 (B, 6) 消息，向 A 和 D 发送作废 6 消息，存储器目录中，块 6 被 B 独占，B 的块 6 为独占
CPU C 读第 6 块	向存储器发送读不命中 (C, 6) 消息，向 B 中读块 6，写回到存储器，再将块 6 从存储器发送到 C，B 和 B 中的块 6 为共享，存储器中的块 6 的共享集合为 {BC}
CPU D 写第 20 块	向存储器发送写不命中 (D, 20) 消息，将块 20 发送到 D，存储器目录中，块 20 被 D 独占，D 的块 20 为独占
CPU A 写第 20 块	向存储器发送写不命中 (A, 20) 消息，向 D 发送取并作废 20 的消息，D 将块 20 写回到存储器，并标记自己的 cache 为无效，存储器将块 20 发送到 A，存储器目录中，块 20 被 D 独占，D 的块 20 为独占
CPU D 写第 6 块	向存储器发送写不命中 (D, 6) 消息，向 B 和 C 发送作废 6 消息，它们 cache 中的块 6 改为无效，存储器将块 6 发送给 D，存储器目录中，块 6 被 D 独占，D 的块 6 为独占
CPU A 读第 12 块	向存储器发送写回并修改共享集 (A, 20) 消息，A 将块 20 写回存储器，存储器块 20 的共享目录变为空，向存储器发送读不命中 (A, 12) 消息，将块 12 发送到 A，A 的块 12 为共享，存储器中的块 12 共享集合为 {A}

4.2 截图，展示执行完以上操作后整个 cache 系统的状态



五、综合问答

5.1 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？

监听法：

优点：易于实现，在小规模的多处理器系统中，监听法相对简单，因为每个处理器都可以直接监听总线，以查看是否有其他处理器在访问它们正在使用的缓存行。低延迟：由于处理器直接监听总线，因此可以立即检测到一致性问题并立即解决，这可以减少延迟。

缺点：总线的可扩放行受到一定限制，总线上能够连接的处理器数目有限，共享总线存在竞争使用问题，在由大量处理器构成的多处理器系统中，监听带宽是瓶颈。

目录法：

优点：可扩展，目录法适合于大规模的多处理器系统，因为它不依赖于广播或者共享总线，而是通过维护一个集中的目录或者分布式目录来跟踪每个数据块的状态和位置。

缺点：实现复杂。额外的存储开销，目录法需要维护目录信息，这会带来额外的存储开销。通信延迟，目录法需要通过网络发送和接收消息，这会增加通信延迟。

5.2 Tomasulo 算法相比 Score Board 算法有什么异同?

1、分别解决了什么相关

Tomasulo 算法主要用于解决数据相关和控制相关，特别是在浮点运算中。它利用寄存器重命名和动态调度来解决。

Scoreboard 也是为了解决数据相关和控制相关，使用 Scoreboard 追踪每个指令的状态，在数据没有准备好时 stall，准备好后才允许执行，并在所有依赖都解决后才写回指令结果。

2、分别是分布式还是集中式

Tomasulo 算法是一种分布式调度算法。各个功能单元的调度是分开的，各个功能单元有自己的保留站（reservation stations）进行调度。

Scoreboarding 算法是一种集中式调度算法。所有的功能单元的调度都是通过一个中心化的 Scoreboard 进行的，此表负责追踪所有在执行的和等待执行的指令状态。

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

结构相关：记录每个功能部件的状态，由结构相关时不执行指令。

RAW：操作数不可用时不执行指令；使用 CDB 广播，当一个指令完成时，它将结果广播给所有正在等待这个结果的指令。

WAR：使用寄存器重命名，每个指令都将其结果写入一个新的、唯一的寄存器位置，这样旧的寄存器值仍然可以被后续的指令访问，直到不再需要为止。

WAW：使用寄存器重命名，每个指令都将其结果写入一个新的、唯一的寄存器位置，因此不存在两个指令写入相同寄存器的问题，避免了WAW冒险。