计算机体系结构实验六实验报告

PB21111681 朱炜荣

一、Tomasulo 算法模拟器

初始配置图如下所示：

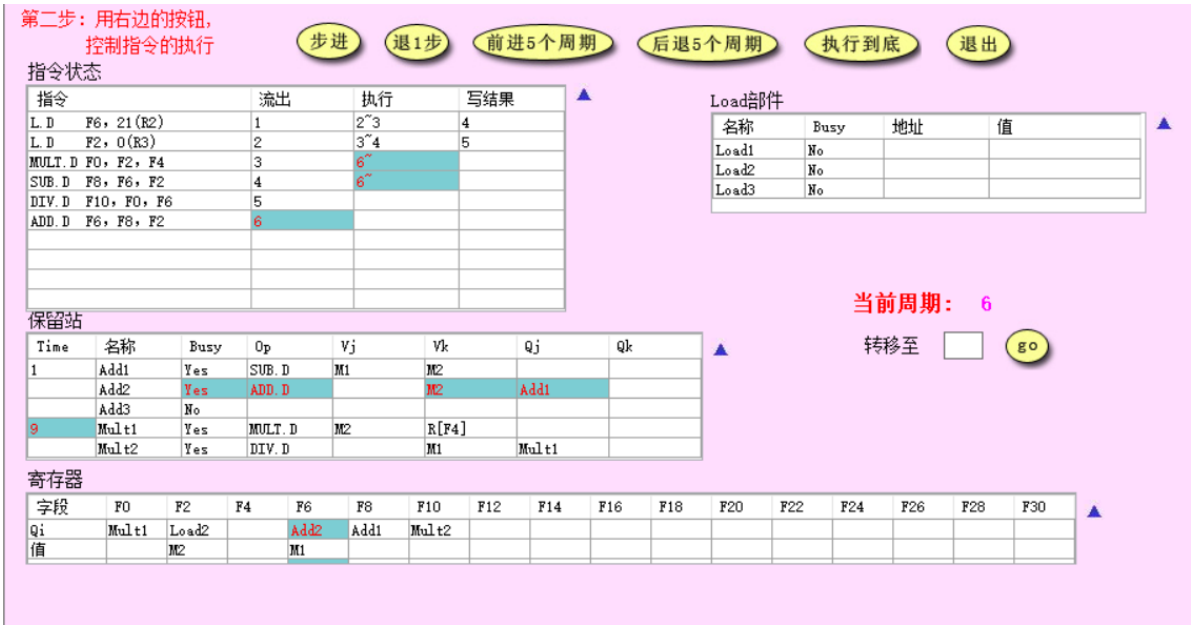
1.1

当前周期2和当前周期3截图如下：

- 第二个周期：load1部件的地址从21变为R[R2]+21，load2部件从空闲变为busy，并将偏移量0移入地址中。

- 第三个周期：load1部件取得了对应地址的数据，而load2部件得地址从0变为R[R3]+0。

1.2

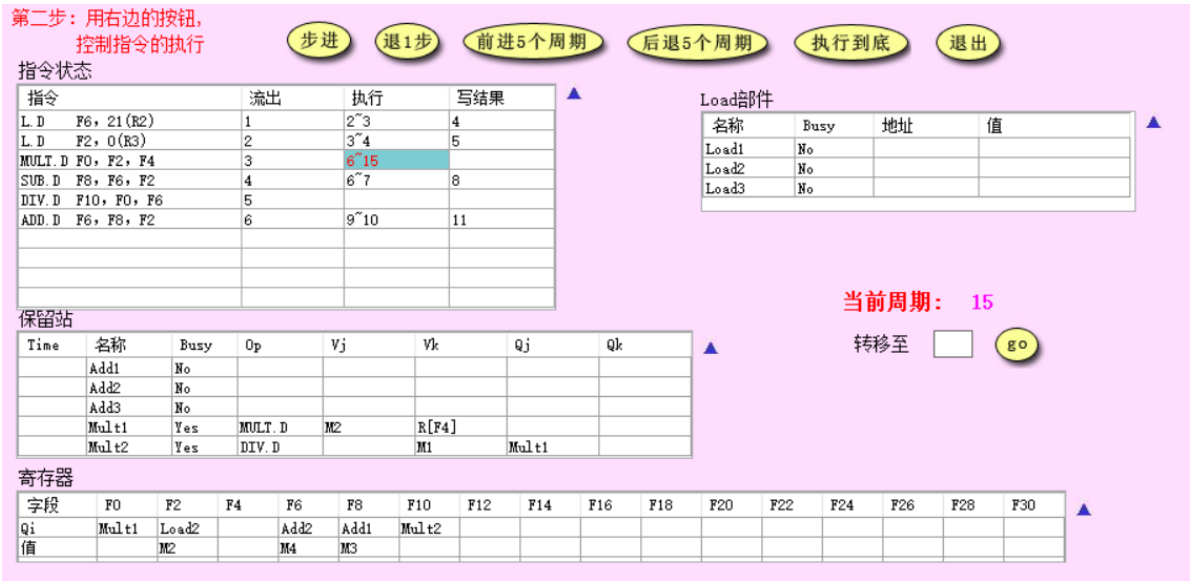


- 在第六个周期，MULT.D指令进入到执行阶段
- 在这个周期中ADD.D指令流出，同时MULT.D和SUB.D同时进入到执行阶段
- load部件没有发生变化，此时两条load指令已经执行结束
- 保留站中Add2进入Busy状态，这是因为Add1还在执行SUB.D相关操作。由于F8的值需要等待SUB.D的值，所以Qj更新为Add1。由于F2的值已经在第二条L.D中算出，所以放在了Vk中；Mult1由于执行了一个周期，因此Time变为9
- 寄存器F6从空闲变为busy，值被定义为从ADD.D指令获得结果

1.3

- 因为MULT.D中的F2的数值在第六个周期之前还没有被加载到寄存器中

1.4



- 第十五个周期：MULT.D的指令执行完毕，执行周期为6-15。同时保留站中Mult1的剩余时间清零。

- 第十六个周期：MULT.D的指令将得到的结果写回。保留站中将mult1的状态设置为空闲，同时将mult2的Vj设置为M5，也就是MULT.D刚刚算出得到的结果。在寄存器中，F0的值被设置为M5。

1.5

所有指令刚刚执行完毕时是第57个周期。

二、多 cache 一致性算法监听法

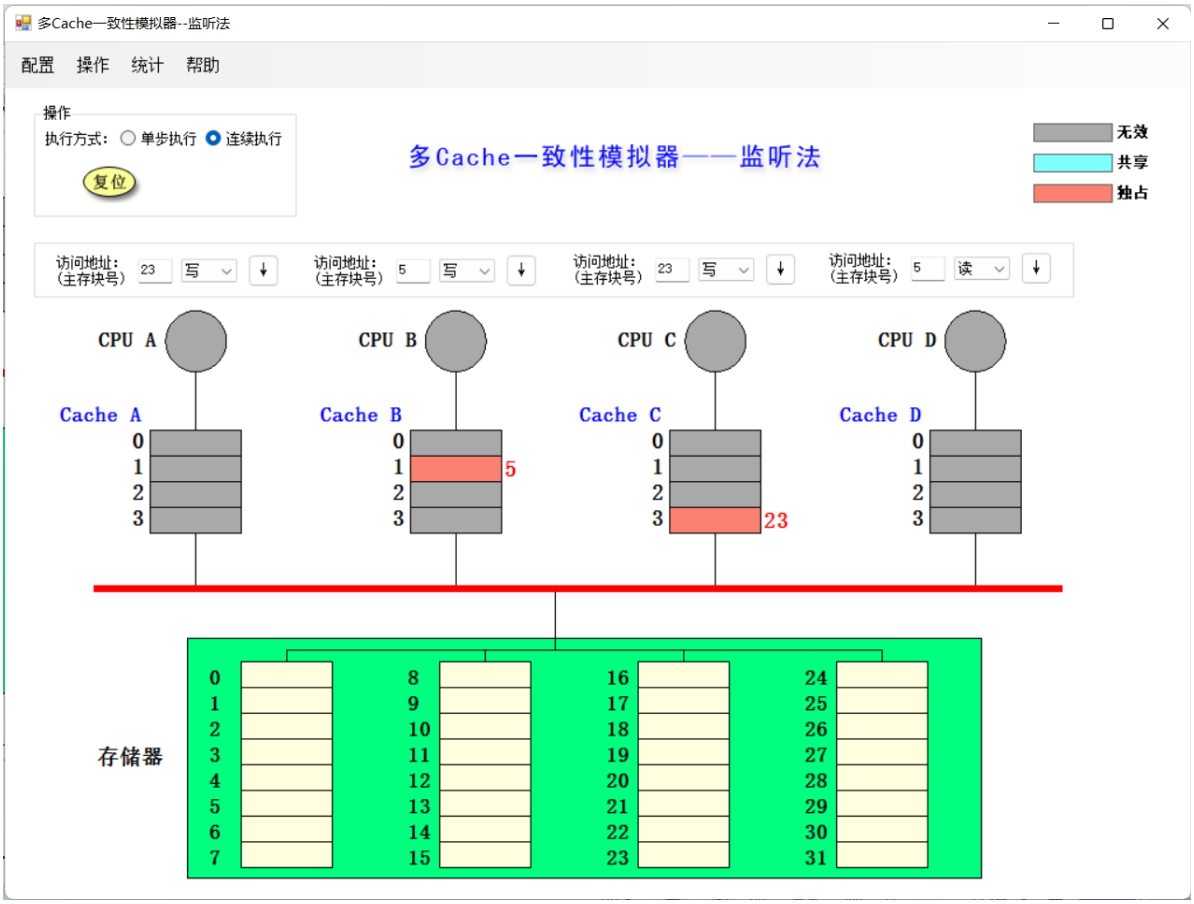
2.1多 cache 一致性算法监听法

利用模拟器进行下述操作，具体填写情况如下：

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	是	否	CPU A读Cache A，读没有命中，将存储器第5块的数据换入到Cache A第1行，并将其设置为共享
CPU B 读第 5 块	是	否	CPU B读Cache B，读没有命中，将存储器第5块的数据换入到Cache B第1行，并将其设置为共享
CPU C 读第 5 块	是	否	CPU C读Cache C，读没有命中，将存储器第5块的数据换入到Cache C第1行，并将其设置为共享
CPU B 写第 5 块	否	否	CPU B写Cache B，写命中，写入新数据并向总线发出“作废第5块”的信号， Cache A和C的第1行数据被无效，Cache B第1行被覆写并将其设置为独占
CPU D 读第 5 块	是	是	CPU D读Cache D，读没有命中，Cache B将其独占的第1行写回主存第5块，并将其设置为共享，Cache D将存储器第5块数据换入到第1行，并将其设置为共享
CPU B 写第 21 块	是	否	CPU B写Cache B，写没有命中，将存储器第21块换入到Cache B第1行，CPU B写入新数据并将其设置为独占
CPU A 写第 23 块	是	否	CPU A写Cache A，写没有命中，将存储器第5块的数据换入到Cache A第3行，并将其设置为独占
CPU C 写第 23 块	是	是	CPU C写Cache C，写没有命中，Cache A将其独占的第3行写回主存第23块，并将其设置为无效， Cache C将存储器第23块数据换入到第3行，CPU C写入新数据并将其设置为独占
CPU B 读第 29 块	是	是	CPU B读Cache B，读没有命中，Cache B将其独占的第1行写回主存第21块，将存储器第29块换入到Cache B第1行，并将其设置为共享

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 B 写第 5 块	是	否	CPU B读Cache B，读没有命中，Cache B将存储器第5块换入到Cache B第1行中，并将其设置为共享

执行过后的情况如下所示：

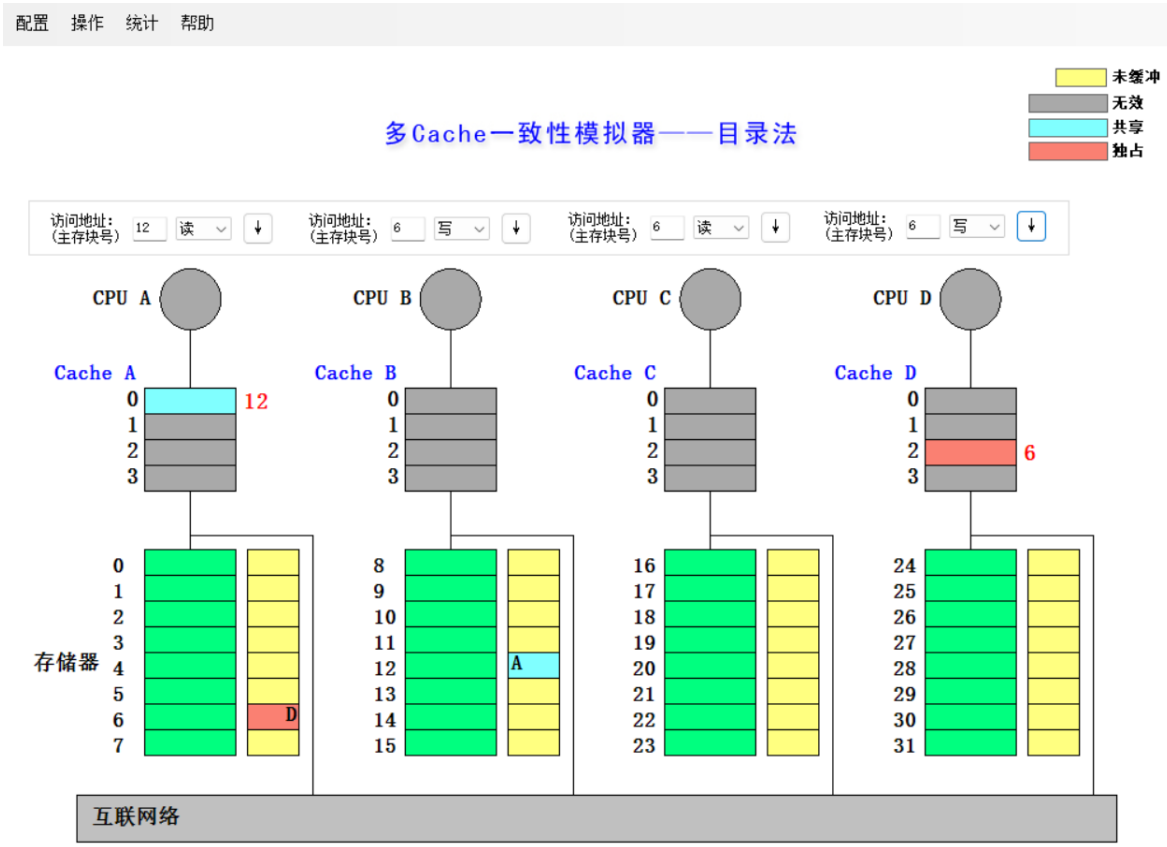


2.2多 cache 一致性算法-目录法

利用模拟器进行下述操作：

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读Cache A，读没有命中，本地向宿主节点发读不命中(A, 6)消息，宿主把数据块送给本地节点并写入第2行，设置第6块共享集合为{A}，Cache A将第2行设置为共享
CPU B 读第 6 块	CPU B读Cache B，读没有命中，本地向宿主节点发读不命中(B, 6)消息，宿主把数据块送给本地节点并写入第2行，设置第六块共享集合为{A, B}，Cache B将第2行设置为共享
CPU D 读第 6 块	CPU D读Cache D，读没有命中，本地向宿主节点发读不命中(D, 6)消息，宿主把数据块送给本地节点并写入第2行，设置第六块共享集合为{A, B, D}，Cache D将第2行设置为共享
CPU B 写第 6 块	CPU B写Cache B，写命中，本地向宿主节点发写命中(B, 6)消息，宿主向远程节点A和D发作废(6)消息，设置第6块共享集合为{B}且为独占，Cache B将第2行设置为独占
CPU C 读第 6 块	CPU C读Cache C，读没有命中，本地向宿主节点发读不命中（B,6)的消息，存储器读取CacheB第2行并写回，宿主将数据块送给CacheC 2，设置为共享状态，共享集合为{B,C}
CPU D 写第 20 块	CPU D写Cache D，写没有命中，本地向宿主节点发写不命中（D,20)的消息，存储器向D的Cache发送第20块，Cache D将第0行设置为独占状态，共享集合为{D}
CPU A 写第 20 块	CPU A写Cache A，写没有命中，本地向宿主节点发写不命中（A,20)的消息，宿主向远程节点Cache D发送取并作废(20)的消息，Cache D将数据块送给宿主节点并将其第0行作废，宿主把数据块送给Cache A，设置第20块共享集合为{A}且为独占，CPU A将数据写入第0行，并将其设置为独占
CPU D 写第 6 块	CPU D写Cache D，写没有命中，本地向宿主节点发送写不命中(D, 6)消息，宿主向远程节点Cache B和C发送作废(6)消息，Cache B和C无效掉读的第2行，宿主把数据块送给本地节点Cache D，设置第6块共享集合为{D}且为独占，CPU D将数据写入第2行，并将其设置为独占
CPU A 读第 12 块	CPU A读Cache A，读不命中，本地向被替换的宿主节点发写回并修改共享集(A, 20)消息，并将第0行写回主存，宿主也将第20行设置为未缓冲。本地再向宿主节点发送读不命中(A, 12)消息，宿主将数据块送给本地节点，设置第6块共享集合{A}，Cache A将第0行设置为共享

展示执行完以上操作后整个 cache 系统的状态如下所示：



三、问题回答

1. **目录法（Directory-based）**：
   - **优势**：
     - 数据一致性较高：目录法维护了一个中心化的目录或者缓存，记录了数据的位置和状态，因此可以提供较高的数据一致性。
     - 性能较好：在大规模的系统中，目录法通常比监听法有更好的性能表现，因为监听法可能面临过多的通信负担。
     - 对于大规模系统较为适用：在大规模系统中，目录法可以更好地管理和维护数据的一致性。
   - **劣势**：
     - 单点故障：由于目录是中心化的，因此目录成为了系统的单点故障，一旦目录发生故障，可能会导致整个系统的故障。
     - 数据一致性开销：维护数据一致性需要额外的通信和处理开销，这可能会影响系统的整体性能。
     - 扩展性限制：目录法在扩展性方面存在一定的限制，当系统规模过大时，可能会面临管理和维护目录的挑战。

   监听法（Snoop-based）：
   - **优势**：
     - 分布式特性：监听法通常是分布式的，每个节点都可以独立地处理数据请求，因此不存在单点故障。
     - 低延迟：由于数据通常存储在本地缓存中，并且节点可以直接与缓存进行交互，因此监听法通常具有较低的访问延迟。
     - 较好的扩展性：由于监听法是分布式的，因此具有较好的扩展性，可以方便地增加节点以支持更大规模的系统。
   - **劣势**：

        - 数据一致性较低：由于监听法中的节点是分布式的，节点之间的数据一致性可能受到影响，需要额外的机制来保证数据的一致性。
        - 高通信开销：监听法中的节点需要频繁地监听和响应其他节点的请求，这可能会导致较高的通信开销，尤其在大规模系统中。
        - 配置复杂性：由于监听法涉及多个节点之间的协作，系统的配置和管理可能会更加复杂，需要更多的管理和维护工作。
2. Tomasulo算法主要解决了数据相关性和指令级并行性问题。通过动态调度和乱序执行，它能够在保持数据依赖关系的同时，充分利用硬件资源，提高指令级并行度。

   Scoreboard算法也解决了数据相关性问题，但其主要目标是控制指令的执行顺序，以保证数据相关性的正确性。它通过中心化的控制器来管理指令的执行状态和资源的分配情况。

   Tomasulo算法是一种分布式的算法。它将资源分配和指令调度的控制分布在各个功能单元中，通过数据流标记的方式实现动态调度和乱序执行。而Scoreboard算法是一种集中式的算法。它采用一个中心化的控制器来管理指令执行的状态和资源的分配，通过一个全局的状态表（scoreboard）来维护指令的状态信息。

3.
    - 结构相关：Tomasulo算法通过对每个功能单元的状态进行跟踪，来避免多个指令同时访问同一功能单元的问题。
    - RAW相关：Tomasulo算法使用操作数寄存器和保留站来解决RAW相关问题。
    - WAR相关：Tomasulo算法使用寄存器重命名来解决WAR相关问题。
    - WAW相关：Tomasulo算法也使用寄存器重命名来解决WAW相关问题。