cache模拟器报告

姓名：齐御京

学号：3122356050

专业：微电子S2251

**1.模拟器设计思想**

本cache模拟器按照作业要求，实现了双核cpu的 cache一致性四状态协议，同时还有自行发挥部分。

**1.1 模拟器结构设计**

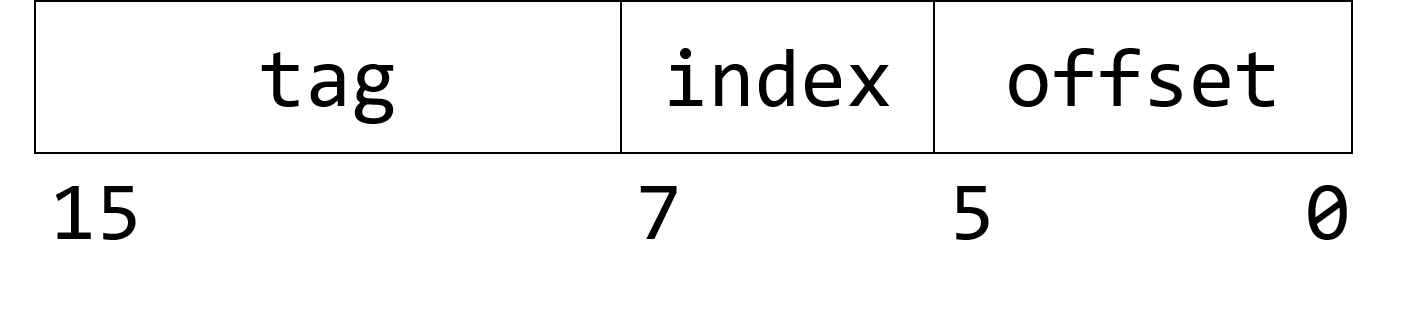
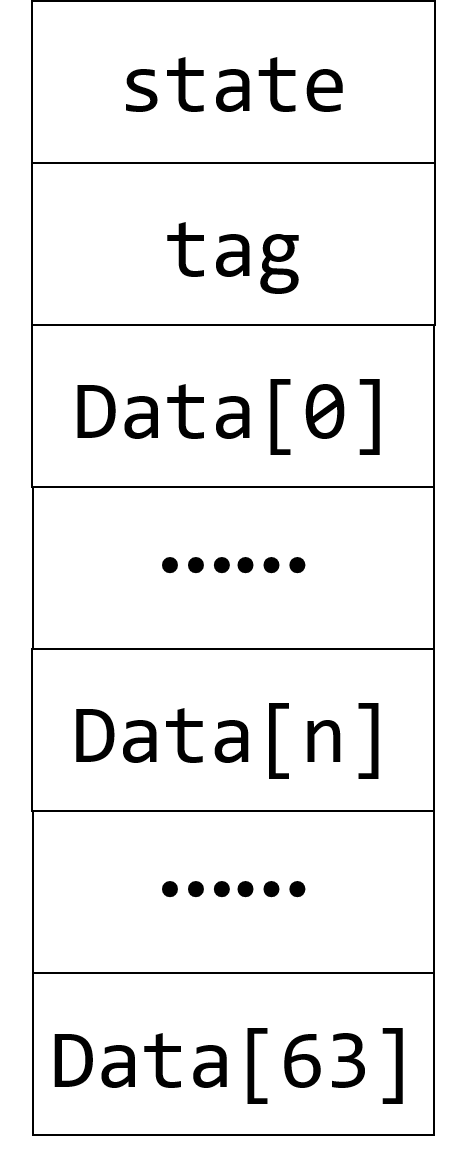
本模拟器有两个cache，单个Cache中有八个cache line，cache line示意图如图1.1.1所示，cache总容量为512Byte。

图 1.1.1

图 1.1.2

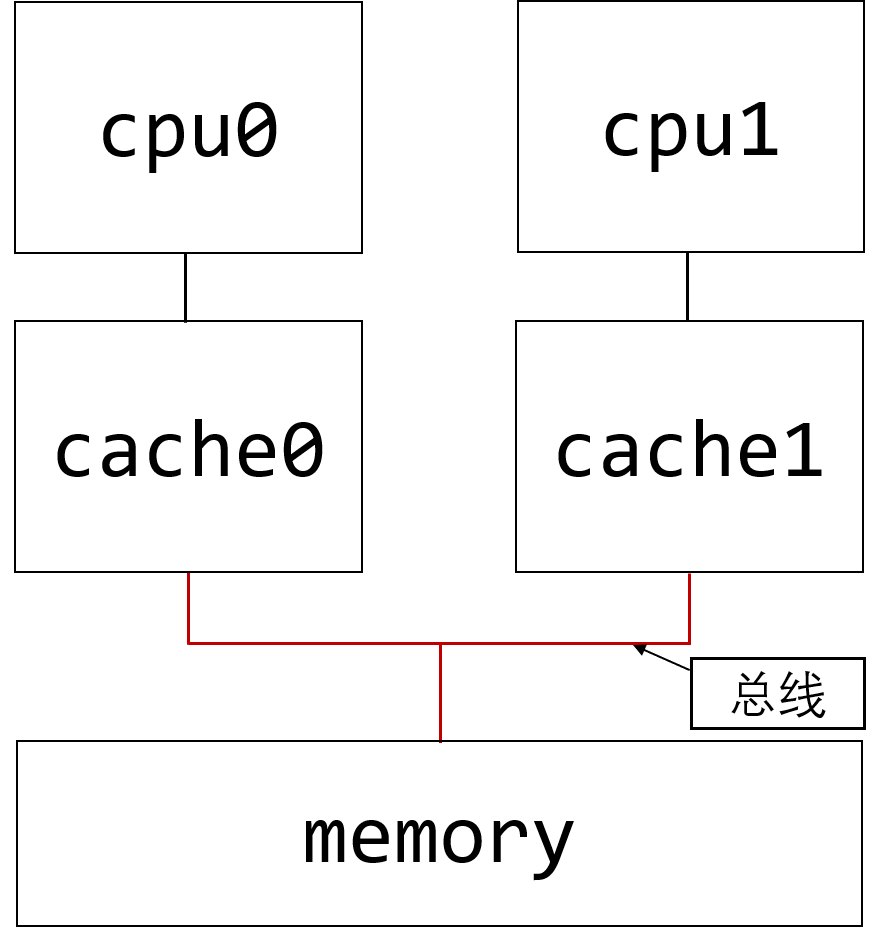


图 1.1.3

模拟器memory的地址从0x0000-0x7FFF，每个地址可以存储1Byte的数据，容量为32KB。cache与memory采用两路组相联的映射策略，cache被分为四组。主存地址共十五位，其中的tag，index，offset位如图1.1.2所示。

memory和cache使用总线连接，如图1.1.3所示。

**1.2 模拟器功能设计**

模拟器采用Write Once策略，即第一次写采用WT策略，之后的写采用WB策略。

模拟器初始化时，将memory中的所有数据设置为“10000001”，所有cache line状态置无效，数据为“00000000”。写操作时，写入的数据是当前操作次数的二进制字符。

cpu读取指令文件之后，将地址解码为tag部分，index部分和offset部分。接着cpu使用cache的目录表判断自己是否命中。若两个核心同时对同一内存地址分别进行读或写，则执行写命令的核心优先执行，以保持读到的数据是最新的。整体流程如图1.2.1所示。

**1.2.1 cache命中**

执行读操作，则直接读取数据即可，状态保持不变；执行写操作，则需要根据命中的cache line的状态执行不同的操作。若其状态为E或者M，则直接修改其中对应offset的数据，并使其状态进入或保持M。无需修改另一个cache中可能命中的cache line的状态，因为当前cache line 进入E状态时，已经广播过了，另一个cache已经执行过远程写的响应了。因为不是第一次写，所以无需修改memory中的数据；

若其状态为S，则首先修改自己的状态为E，并且更新数据，接着向总线广播写操作、写的数据、以及数据有效信号和地址，同时将总线状态信号拉高，表示已占用。如上的总线信号对另一个cache来说是远程写操作，它首先判断自己是否命中对应地址。若命中，则置对应cache line的状态为I；若不命中，则无操作。memory监听到总线上的数据有效，则主动更新对应地址的数据，与发起总线占用的cache更改后的数据保持一致。最后释放总线，所有信号都置无效。

**1.2.2 cache不命中**

第一步需要在cache中开辟一个cache line，用于保存数据。第二步从总线上加载数据。第三步才是cache的读写操作。

第一步首先判断第index组有无状态为I的cache line。若有则直接将其tag修改为本次操作地址的tag；若没有，则使用随机替换策略，随机选择其中一个cache line来占用，若要被占用的cache line 处于E或者S状态，则直接修改其tag为本次操作tag，因为memory中保存有相同的数据，所以不用担心数据丢失。若被占用的cache line处于M状态，则需要先将其数据写回到memory中，不能直接替换，当前cache发起总线写操作，总线地址、数据都是要被替换的cache line的，数据有效信号拉高，总线的目标器件地址为memory。memory监听到上述信号之后，从总线上取下数据，并将总线释放，所有信号置无效。最后将被替换的cache line的tag替换为本次操作的tag。

第二步需要从总线上读取数据，所以发起总线读请求，总线地址设为当前要操作的主存地址，总线目标器件地址设为另一个cache的器件地址。另一个cache监听到如上信号之后，首先判断自己是否命中总线上的主存地址，若不命中则置总线的response信号为-1，这意味着交由主存向总线上写数据。若命中，则它要将数据写到总线上，以供memory或发起请求的cache读取（M状态时，memory也要从主存上读数据）。发起读请求的cache从总线上拿走数据，并判断response信号是否为-1，若是，则表示该数据与内存共享，置cache line的状态为E；若不是-1，则表示该数据与其他cache以及memory共享，置cache line状态为S。

第三步，cpu对cache进行读写操作。若读，则直接从cache中读走数据即可。若写，则将自己的cache line的状态置为E，并将数据写入总线，让主存读取，另一个cache执行远程写的响应操作。

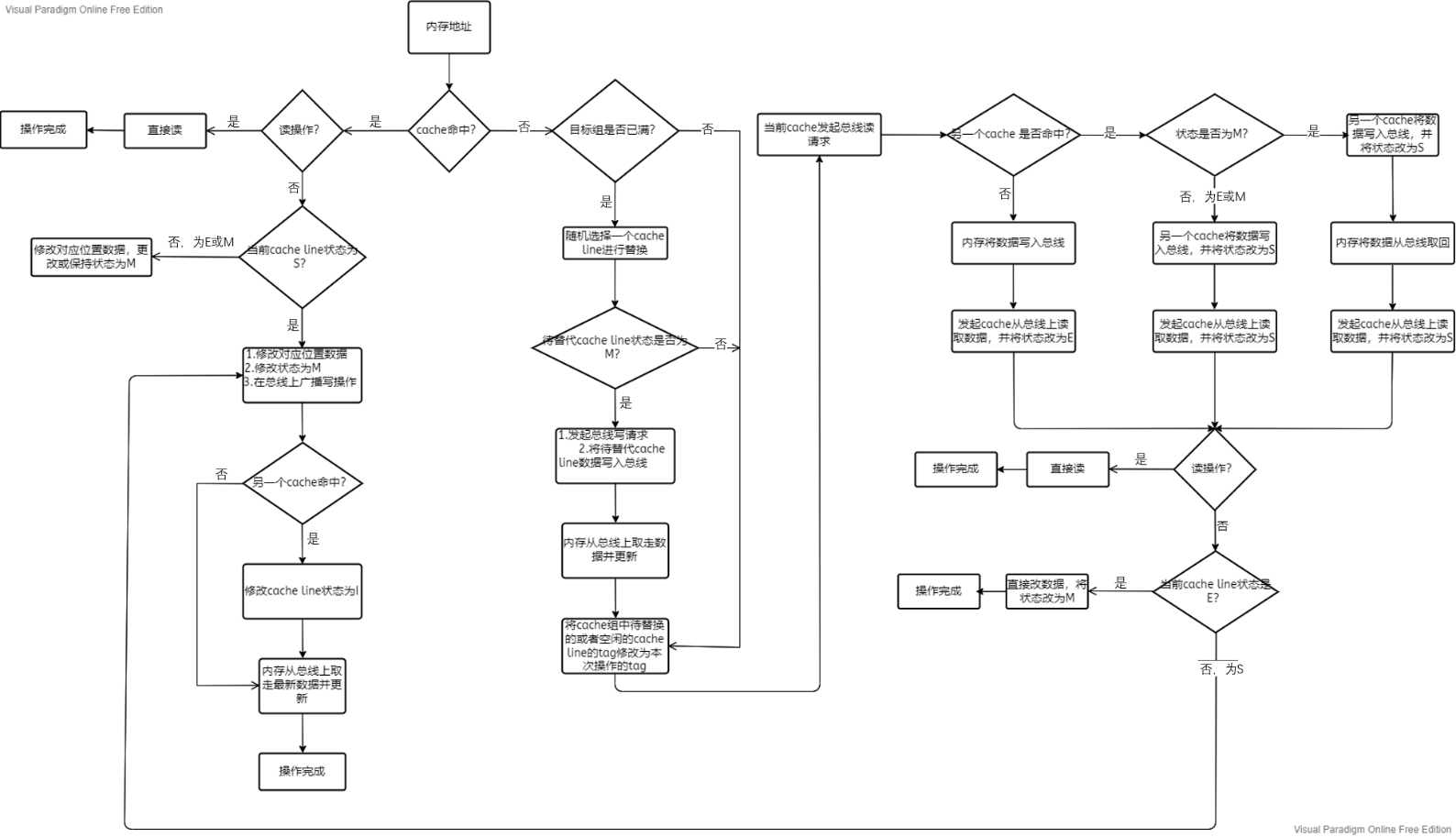
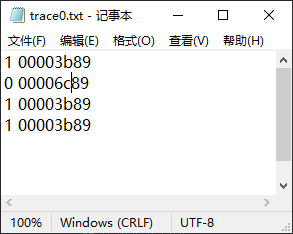
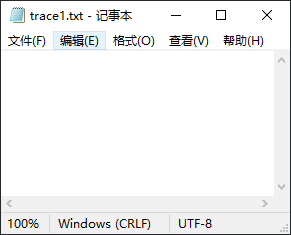


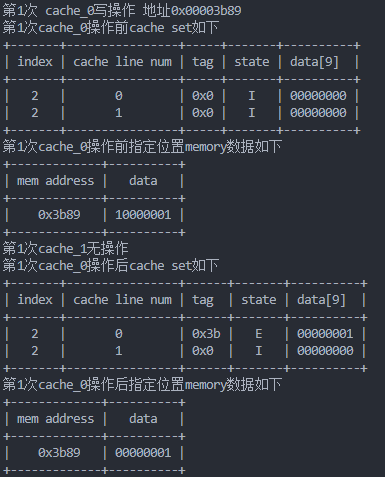
图 1.2.1

**2模拟器功能测试**

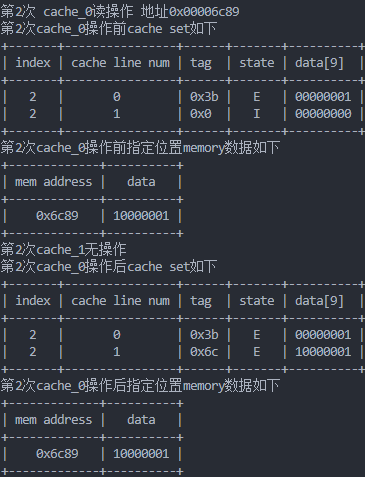
**2.1单核读写功能测试**

 Trace文件如下

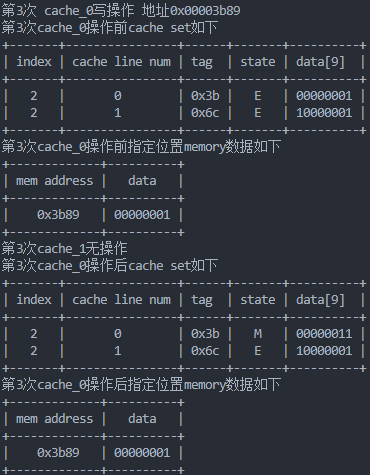
第一次操作结果如下

分析：需要写的内存地址为0x00003b89，tag为3b，index为2，offset为9，cache0在写操作后它的index等于2的组第0个cache line中的offset为9的数据已经被修改，并且cache line的状态也由I变成了E。此时mem中的数据也变成了和cache中一样的值，符合第一次写使用WT策略的设计。

第二次操作结果如下：

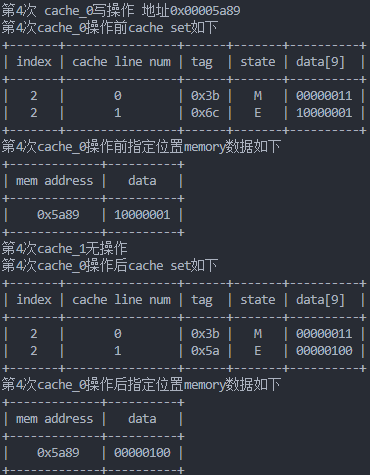
分析：需要读的地址为0x00006c89，tag是6c，index是2，offset是9。读操作完成后，可以看到这个数据被装到了与0x00003b89同样的组中，并且与内存中的数据完全一致，cache line状态也由I变成了E，符合设计预期。

第三次操作结果如下

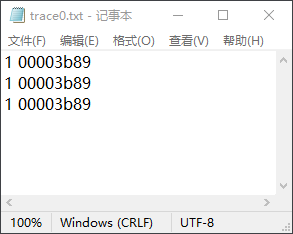
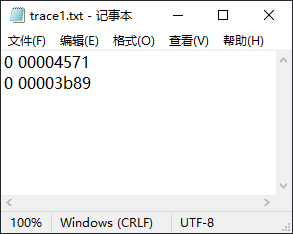


分析：此次操作的地址与第一次操作的地址一样，并且都是写操作，属于第二次写，这次的写只修改了cache中的数据，没有修改主存中的数据，并且cache line的状态由E变成了M，与设计吻合。

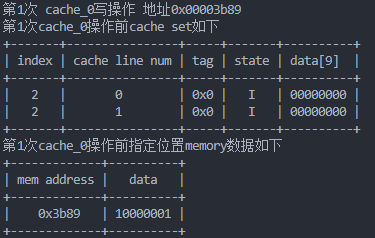
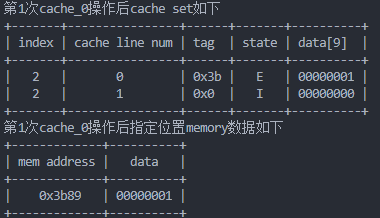
第四次操作结果如下：

分析：第四次写操作的地址也会被映射的之前操作的cache 组中，由于之前cache组已满，模拟器会从两个cache line中随机挑选一个将其替换掉，符合设计预期。

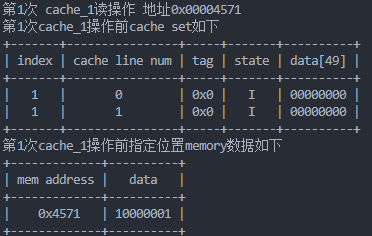
**2.1双核读写功能测试**

****

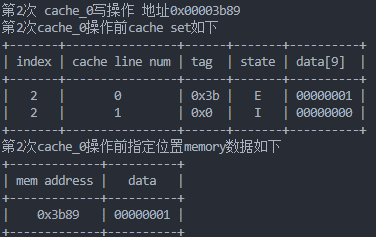
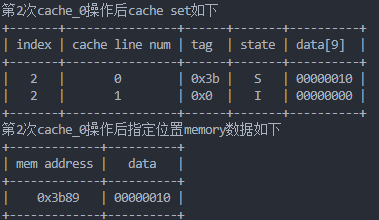
第一次操作结果：

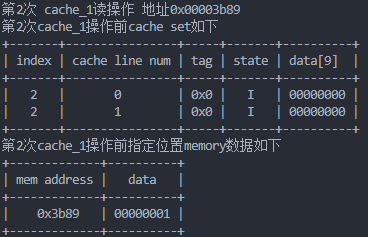
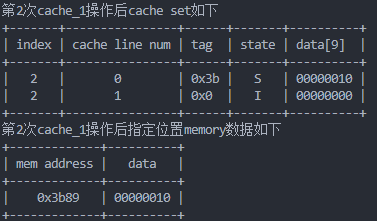
Cache\_0写操作前后：

Cache\_1读操作前后：

分析：cache\_0第一次写采用了写直达，cache\_1从主存中直接读取数据，与单核类似，不再赘述。

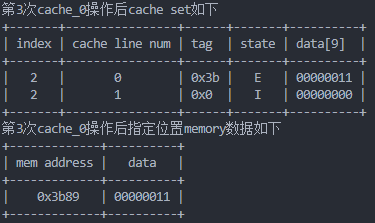
第二次操作结果：

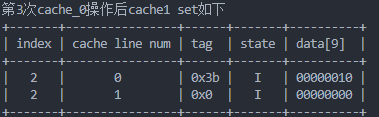
Cache\_0写操作前后：

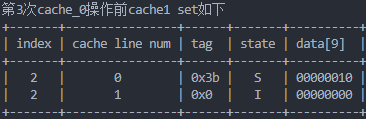
Cache\_1读操作前后：

分析：本次操作两个cache读写地址相同，为了保证读到的数据是最新的，模拟器会保证让写操作先执行。此时cache\_0并非第一次写，因此先暂时进入M状态。cache\_1读数据时发现不命中，则发起总线请求，cache\_0给出反馈，表示最新的数据只有它有，它将数据放上总线，主存和cache\_1都从总线上取走数据，并更改。两个cache line都进入S状态。测试结果也符合上述分析，mem和cache中的数据都是最新的。

第三次操作结果：

Cache\_0操作前后：

Cache\_1本次无操作，我们将它最后的状态打印出来，以便观察状态转变。

分析：cache\_0和cache\_1操作之前的cache line都是S状态，cache\_0写操作之后，进入E状态（第一次写），写完之后它会向总线广播我已修改该地址数据，cache\_0监听总线时发现自己存有该地址数据的副本，于是把自己的状态改为I。测试结果符合预期。

**3.作业感悟**

通过这次大作业，我充分理解了cache一致性问题。编程过程中也有很多感悟，写代码之前要先理清楚思路，数据结构，既要从架构上去整体把握，还要能深入细节仔细考究。

在debug的过程中，也遇到了很多问题。其中印象最深刻的就是cache状态从E或M变成或保持M状态时，只需要写回，不用写直达，代码中明明只修改了cache中的数据，但实际情况是mem中的数据也一起变了。查资料后才发现，列表在赋值的时候，等号左右两个列表是指向同一块内存空间的，改动其中一个，另一个也会跟着变，网上搜到的解决方法是用copy.deepcopy()方法，这样等号左右两个列表的元素都是不同的内存地址，但是经过我实际测试，发现只要不修改元素值，列表元素的内存地址就还是相同的。

**4.代码见bigzuoye.py**