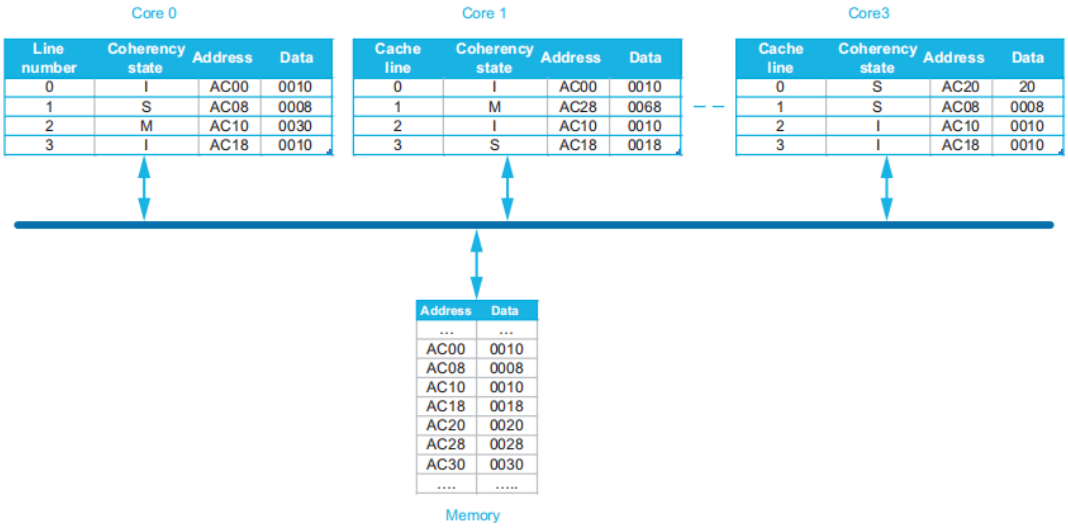
5.1

5.1. [10/10/10/10/10/10/10] <5.2> For each part of this exercise, the initial cache and memory state are assumed to initially have the contents shown in Figure 5.37. Each part of this exercise specifies a sequence of one or more CPU operations of the form



`Ccore#: R, <address>` for reads

and

`Ccore#: W, <address> <-- <value written>` for writes.

For example,

`C3: R, AC10 & C0: W, AC18 <-- 0018`

Read and write operations are for 1 byte at a time. Show the resulting state (i.e., coherence state, tags, and data) of the caches and memory after the actions given below. Show only the cache lines that experience some state change; for example:

`C0.L0: (I, AC20, 0001)` indicates that line 0 in core 0 assumes an "invalid" coherence state (`I`), stores `AC20` from the memory, and has data contents `0001`. Furthermore, represent any changes to the memory state as `M: <address> <- value`.

Different parts (a) through (g) do not depend on one another: assume the actions in all parts are applied to the initial cache and memory states.

a. [10] <5.2> `C0: R, AC20`

b. [10] <5.2> `C0: W, AC20 <-- 80`

c. [10] <5.2> `C3: W, AC20 <-- 80`

d. [10] <5.2> `C1: R, AC10`

e. [10] <5.2> `C0: W, AC08 <-- 48`

f. [10] <5.2> `C0: W, AC30 <-- 78`

g. [10] <5.2> `C3: W, AC30 <-- 78`

初始

```
C0.L0: (I, AC00, 0000)
C1.L0: (I, AC00, 0000)
C3.L0: (I, AC00, 0000)
内存：AC00 <- 0000
```

a) C0: R, AC20 (核心 0 的读操作)

```
C0.L0: (S, AC20, 0000)  # 行变为共享状态 (S)，缓存 AC20，数据不变
```

b) C0: W, AC20 <-- 80 (核心 0 的写操作)

```
C0.L0: (M, AC20, 0080)  # 行变为修改状态 (M)，缓存 AC20，数据 0080
内存：AC20 <- 0080
```

c) C3: W, AC20 <-- 80 (核心 3 的写操作)

```
C3.L0: (S, AC20, 0080)  # 行变为共享状态 (S)，缓存 AC20，数据 0080
```

d) C1: R, AC10 (核心 1 的读操作)

```
C1.L0: (S, AC10, 0000)  # 行变为共享状态 (S)，缓存 AC10，数据不变
```

e) C0: W, AC08 <-- 48 (核心 0 的写操作)

```
C0.L0: (M, AC08, 0048)  # 行变为修改状态 (M)，缓存 AC08，数据 0048
内存：AC08 <- 0048
```

f) C0: W, AC30 <-- 78 (核心 0 的写操作)

```
C0.L0: (M, AC30, 0078)  # 行变为修改状态 (M)，缓存 AC30，数据 0078
内存：AC30 <- 0078
```

g) C3: W, AC30 <-- 78 (核心 3 的写操作)

```
C3.L0: (S, AC30, 0078)  # 行变为共享状态 (S)，缓存 AC30，数据 0078
```

5.2

5.2. [20/20/20/20] <5.3> The performance of a snooping cache-coherent multiprocessor depends on many detailed implementation issues that determine how quickly a cache responds with data in an exclusive or M state block. In some implementations, a processor read miss to a cache block that is exclusive in another processor's cache is faster than a miss to a block in memory. This is because caches are smaller, and thus faster, than main memory. Conversely, in some implementations, misses satisfied by memory are faster than those satisfied by caches. This is because caches are generally optimized for "front side" or CPU references, rather than "back side" or snooping accesses. For the multiprocessor illustrated in Figure 5.37, consider the execution of a sequence of operations on a single processor core where

- read and write hits generate no stall cycles;
- read and write misses generate N_{memory} and N_{cache} stall cycles if satisfied by memory and cache, respectively;
- write hits that generate an invalidate incur $N_{invalidate}$ stall cycles; and
- a write-back of a block, either due to a conflict or another processor's request to an exclusive block, incurs an additional $N_{writeback}$ stall cycles.

Consider two implementations with different performance characteristics summarized in Figure 5.38.

Parameter	Implementation 1 Cycles	Implementation 2 Cycles
N_{memory}	100	100
N_{cache}	40	130
$N_{invalidate}$	15	15
$N_{writeback}$	10	10

Figure 5.38 Snooping coherence latencies.

To observe how these cycle values are used, we illustrate how the following sequence of operations, assuming the initial caches' states in Figure 5.37, behave under implementation 1.

```
C1: R, AC10
C3: R, AC10
```

For simplicity, assume that the second operation begins after the first completes, even though they are on different processor cores.

For Implementation 1,

- the first read generates 50 stall cycles because the read is satisfied by C0's cache: C1 stalls for 40 cycles while it waits for the block, and C0 stalls for 10 cycles while it writes the block back to memory in response to C1's request; and
- the second read by C3 generates 100 stall cycles because its miss is satisfied by memory.

Therefore this sequence generates a total of 150 stall cycles.

For the following sequences of operations, how many stall cycles are generated by each implementation?

a. [20] <5.3>
```
C0: R, AC20
C0: R, AC28
C0: R, AC30
```

b. [20] <5.3>
```
C0: R, AC00
C0: W, AC08 <-- 48
C0: W, AC30 <-- 78
```

c. [20] <5.3>
```
C1: R, AC20
C1: R, AC28
C1: R, AC30
```

d. [20] <5.3>
```
C1: R, AC00
C1: W, AC08 <-- 48
C1: W, AC30 <-- 78
```

a. 对于操作序列 C0: R AC20，C0: R AC28，C0: R AC20：

- C0: R AC20 导致读缺失，但由内存hit，因此产生 100 个停顿周期（假设 Nmemory = 100）。
- C0: R AC28 导致读缺失，但由 C1 cache hit，因此需要等待 C1 的缓存响应，产生 40 个停顿周期。
- C0: R AC20 再次导致读缺失，但由于先前的缓存块被修改并写回内存（这需要 10 个周期），因此再次从内存中获取数据，产生额外的 100 个停顿周期。

因此，在 Implementation 1 中，总共产生 100 + 40 + 10 + 100 + 10 = 260 个停顿周期。在 Implementation 2 中，由于 C0: R AC28 操作需要等待 C1 的缓存响应，因此额外产生 130 个停顿周期，总共为 350 个停顿周期。

b. 对于操作序列 C0: R AC00，C0: W AC08 <-- 48，C0: W AC20 <-- 78：

- C0: R AC00 导致读缺失，但由内存hit，因此产生 100 个停顿周期（假设 Nmemory = 100）。
- C0: W AC08 <-- 48 是写命中，不产生停顿周期。
- C0: W AC20 <-- 78 导致写缺失，但由内存hit，并且需要将旧的缓存块写回内存，因此产生额外的 10 个停顿周期。

因此，在 Implementation 1 和 Implementation 2 中，总共都产生 100 + 10 = 110 个停顿周期。

c. 对于操作序列 C1: R AC20，C1: R AC28，C1: R AC20：

- C1: R AC20 导致读缺失，但由内存hit，因此产生 100 个停顿周期（假设 Nmemory = 100）。
- C1: R AC28 是读命中，不产生停顿周期。
- C1: R AC20 再次导致读缺失，但由于先前的缓存块被修改并写回内存（这需要 10 个周期），因此再次从内存中获取数据，产生额外的 100 个停顿周期。

因此，在 Implementation 1 和 Implementation 2 中，总共都产生 100 + 0 + 100 = 200 个停顿周期。

d. 对于操作序列 C1: R AC00，C1: W AC08 <-- 48，C1: W AC20 <-- 78：

- C1: R AC00 导致读缺失，但由内存hit，因此产生 100 个停顿周期（假设 Nmemory = 100）。
- C1: W AC08 <-- 48 导致写缺失，但由内存hit，并且需要将旧的缓存块写回内存（这需要 10 个周期）。
- C1: W AC20 <-- 78 导致写缺失，但由内存hit，并且需要将旧的缓存块写回内存，因此产生额外的 10 个停顿周期。

因此，在 Implementation 1 和 Implementation 2 中，总共都产生 100 + 100 + 10 = 210 个停顿周期。

5.9

5.9. [10/10/10] <5.4> For each part of this exercise, assume that initially all caches lines are invalid, and the data in memory Mi is the byte i (0X00 <= i <= 0x07) repeated as many times as the block size. Assume that successive requests are completely serialized. That is, no core will issue a coherency request until the previous request (by same or different core) is completed.

For each of the following parts,

- show the final state (i.e., coherence state, sharers/owners, tags, and data) of the caches and directory controller (including data values) after the given transaction sequence has completed; and
- show the messages transferred (choose a suitable format for message types).

a. [10] <5.4>
```
C3: R, M4
C3: R, M2
C7: W, M4 <--0xaaaa
C1: W, M4 <--0xbbbb
```

b. [10] <5.4>
```
C3: R, M0
C3: R, M2
C6: W, M4 <--0xaaaa
C3: W, M4 <--0xbbbb
```

c. [10] <5.4> C0: R, M7
C3: R, M4
C6: W, M2 <--0xaaaa
C2: W, M2 <--0xbbbb

a. 事务序列：

1. C3 发出读（R）请求，请求内存 M4 中的数据。
2. C3 发出另一个读（R）请求，请求内存 M2 中的数据。
3. C7 发出写（W）请求，将值为 0xaaaa 的数据写入内存 M4。
4. C1 发出写（W）请求，将值为 0xbbbb 的数据写入内存 M4。

最终状态：

- 缓存和目录控制器：
  - C0：缓存行无效。
  - C1：缓存行无效。
  - C2：缓存行无效。
  - C3：缓存行无效。
  - C7：缓存行无效。
  - Dir4：M4 的目录条目标记为无效（I）。
  - Dir2：M2 的目录条目标记为无效（I）。
- 传输的消息：
  1. C3（R）-> Dir4：针对 M4 的读不命中请求（011! 010! 000! 100）
  2. Dir4 -> M4：M4 的读响应（数据）（100! 101! 111! 100）
  3. M4 -> Dir4：从 M4 读取的数据（100! 101! 111! 100）[数据：0x0004]
  4. Dir4 -> C3（R）：针对 M4 的读响应（数据）（011! 010! 000! 100）[数据：0x0004]
  5. C3（R）-> Dir2：针对 M2 的读不命中请求（011! 010）
  6. Dir2 -> M2：M2 的读响应（数据）（010! 011）
  7. M2 -> Dir2：从 M2 读取的数据（010! 011）[数据：0x0002]
  8. Dir2 -> C3（R）：针对 M2 的读响应（数据）（011! 010）[数据：0x0002]
  9. C7（W）-> Dir4：针对 M4 的写不命中请求（111! 110! 100）[数据：0xaaaa]
  10. Dir4 -> M4：M4 的数据写入请求（100! 101! 111! 100）[数据：0xaaaa]
  11. M4 -> Dir4：写入的确认消息（111! 110! 100）
  12. Dir4 -> C7（W）：写入的确认消息（100! 101! 111）
  13. C1（W）-> Dir4：针对 M4 的写不命中请求（001! 000! 100）[数据：0xbbbb]
  14. Dir4 -> M4：M4 的数据写入请求（100! 101! 111! 100）[数据：0xbbbb]
  15. M4 -> Dir4：写入的确认消息（111! 110! 100）
  16. Dir4 -> C1（W）：写入的确认消息（100! 101! 111）

b. 事务序列：

1. C3 发出读（R）请求，请求内存 M0 中的数据。
2. C3 发出另一个读（R）请求，请求内存 M2 中的数据。
3. C6 发出写（W）请求，将值为 0xaaaa 的数据写入内存 M4。
4. C3 发出写（W）请求，将值为 0xbbbb 的数据写入内存 M4。

最终状态：

- 缓存和目录控制器：
  - C0：缓存行无效。

    - C3：缓存行无效。
    - C6：缓存行无效。
    - Dir0：M0 的目录条目标记为无效（I）。
    - Dir2：M2 的目录条目标记为无效（I）。
    - Dir4：M4 的目录条目标记为无效（I）。
- 传输的消息：

    1. C3（R）-> Dir0：针对 M0 的读不命中请求（011! 010! 000! 000）
    2. Dir0 -> M0：M0 的读响应（数据）（000! 001! 111! 000）
    3. M0 -> Dir0：从 M0 读取的数据（000! 001! 111! 000）[数据：0x0000]
    4. Dir0 -> C3（R）：针对 M0 的读响应（数据）（011! 010! 000! 000）[数据：0x0000]
    5. C3（R）-> Dir2：针对 M2 的读不命中请求（011! 010）
    6. Dir2 -> M2：M2 的读响应（数据）（010! 011）
    7. M2 -> Dir2：从 M2 读取的数据（010! 011）[数据：0x0002]
    8. Dir2 -> C3（R）：针对 M2 的读响应（数据）（011! 010）[数据：0x0002]
    9. C6（W）-> Dir4：针对 M4 的写不命中请求（111! 110! 100）[数据：0xaaaa]
    10. Dir4 -> M4：M4 的数据写入请求（100! 101! 111! 100）[数据：0xaaaa]
    11. M4 -> Dir4：写入的确认消息（111! 110! 100）
    12. Dir4 -> C6（W）：写入的确认消息（100! 101! 111）
    13. C3（W）-> Dir4：针对 M4 的写不命中请求（011! 010! 100）[数据：0xbbbb]
    14. Dir4 -> M4：M4 的数据写入请求（100! 101! 111! 100）[数据：0xbbbb]
    15. M4 -> Dir4：写入的确认消息（111! 110! 100）
    16. Dir4 -> C3（W）：写入的确认消息（011! 010! 100）

c. 事务序列：

1. C0 发出读（R）请求，请求内存 M7 中的数据。
2. C3 发出读（R）请求，请求内存 M4 中的数据。
3. C6 发出写（W）请求，将值为 0xaaaa 的数据写入内存 M2。
4. C2 发出写（W）请求，将值为 0xbbbb 的数据写入内存 M2。

最终状态：

- 缓存和目录控制器：
    - C1：缓存行无效。
    - C3：缓存行无效。
    - C6：缓存行无效。
    - C2：缓存行无效。
    - C0：缓存行无效。
    - Dir7：M7 的目录条目标记为无效（I）。
    - Dir4：M4 的目录条目标记为无效（I）。
    - Dir2：M2 的目录条目标记为无效（I）。
- 传输的消息：（同样的分析如（a））
    1. C0（R）-> Dir7：针对 M7 的读不命中请求（011! 010! 000! 111）
    2. Dir7 -> M7：M7 的读响应（数据）（111! 000! 111! 111）
    3. M7 -> Dir7：从 M7 读取的数据（111! 000! 111! 111）[数据：0x0007]
    4. Dir7 -> C0（R）：针对 M7 的读响应（数据）（011! 010! 000! 111）[数据：0x0007]
    5. C3（R）-> Dir4：针对 M4 的读不命中请求（011! 010! 000! 100）
    6. Dir4 -> M4：M4 的读响应（数据）（100! 101! 111! 100）
    7. M4 -> Dir4：从 M4 读取的数据（100! 101! 111! 100）[数据：0x0004]
    8. Dir4 -> C3（R）：针对 M4 的读响应（数据）（011! 010! 000! 100）[数据：0x0004]
    9. C6（W）-> Dir2：针对 M2 的写不命中请求（111! 110! 010）[数据：0xaaaa]

10. Dir2 -> M2：M2 的数据写入请求（010! 011! 010）[数据：0xaaaa]
11. M2 -> Dir2：写入的确认消息（010! 011! 010）
12. Dir2 -> C6（W）：写入的确认消息（111! 110! 010）
13. C2（W）-> Dir2：针对 M2 的写不命中请求（001! 000! 010）[数据：0xbbbb]
14. Dir2 -> M2：M2 的数据写入请求（010! 011! 010）[数据：0xbbbb]
15. M2 -> Dir2：写入的确认消息（010! 011! 010）
16. Dir2 -> C2（W）：写入的确认消息（001! 000! 010）