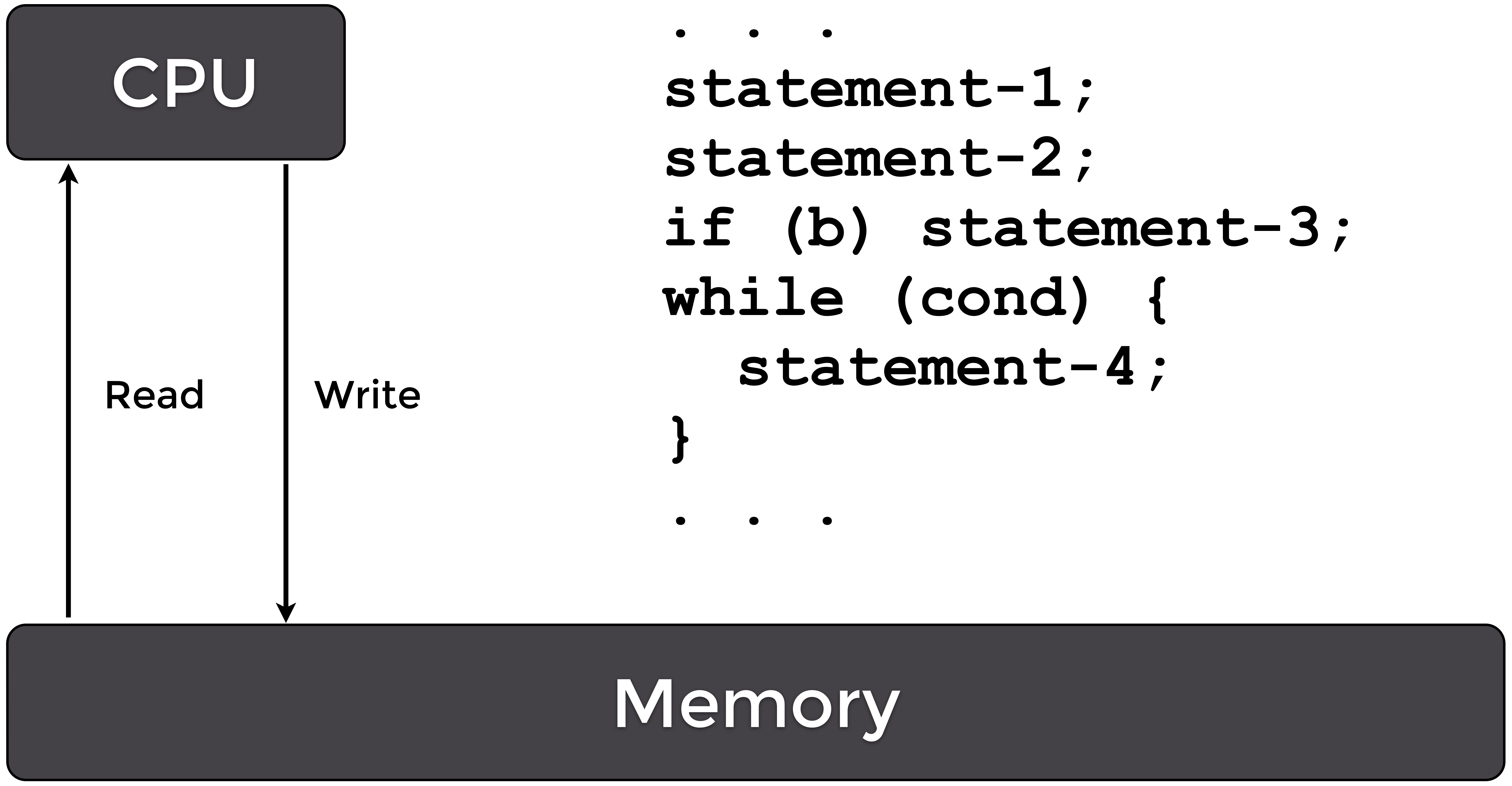
Stalking the Lost Write: Memory Visibility in Concurrent Java

Jeff Berkowitz, New Relic

QCon San Francisco, November 2014

The Computer We Imagine



```
. . .
statement-1;
statement-2;
if (b) statement-3;
while (cond) {
  statement-4;
}
. . .
```

The Compiler We Imagine

Typical assembly language - no particular CPU

<u>Java</u>

```
x++;
```

```
y++;
```

<u>Assembly Language</u>

```
mov mem.x, reg1
incr reg1
mov reg1, mem.x
```

```
mov mem.y, reg1
incr reg1
mov reg1, mem.y
```

The Compiler We Imagine

Typical assembly language - no particular CPU

Java	Assembly Language*

`x++;`	`mov mem.x, reg1`
	`incr reg1`
	`mov reg1, mem.x`

`y++;`	`mov mem.y, reg1`
	`incr reg1`
	`mov reg1, mem.y`

The Compiler We Get

Java

```
x++;
```

```
y++;
```

Assembly Language

```
mov mem.x, reg1
mov mem.y, reg2

incr reg1
mov reg1, mem.x

incr reg2
mov reg2, mem.y
```

The Compiler We Get

Java

```
x++;

y++;
```

Assembly Language

```
mov mem.x, reg1
mov mem.y, reg2

incr reg1
mov reg1, mem.x

incr reg2
mov reg2, mem.y
```

The End Result

Typical micro operations - no particular CPU

Java

```
x++;

y++;
```

Assembly Language

```
mov mem.x, reg1
mov mem.y, reg2

incr reg1
mov reg1, mem.x

incr reg2
mov reg2, mem.y
```

Hardware Level

```
rd.issue(x)
rd.issue(y)

resp.mov(r1)
resp.mov(r2)
incr r1
wr.async(r1, x)

incr r2
wr.async(r2, y)
```

The End Result

Typical micro operations - no particular CPU

Java

```
x++;

y++;
```

Assembly Language

```
mov mem.x, reg1
mov mem.y, reg2

incr reg1
mov reg1, mem.x

incr reg2
mov reg2, mem.y
```

Hardware Level

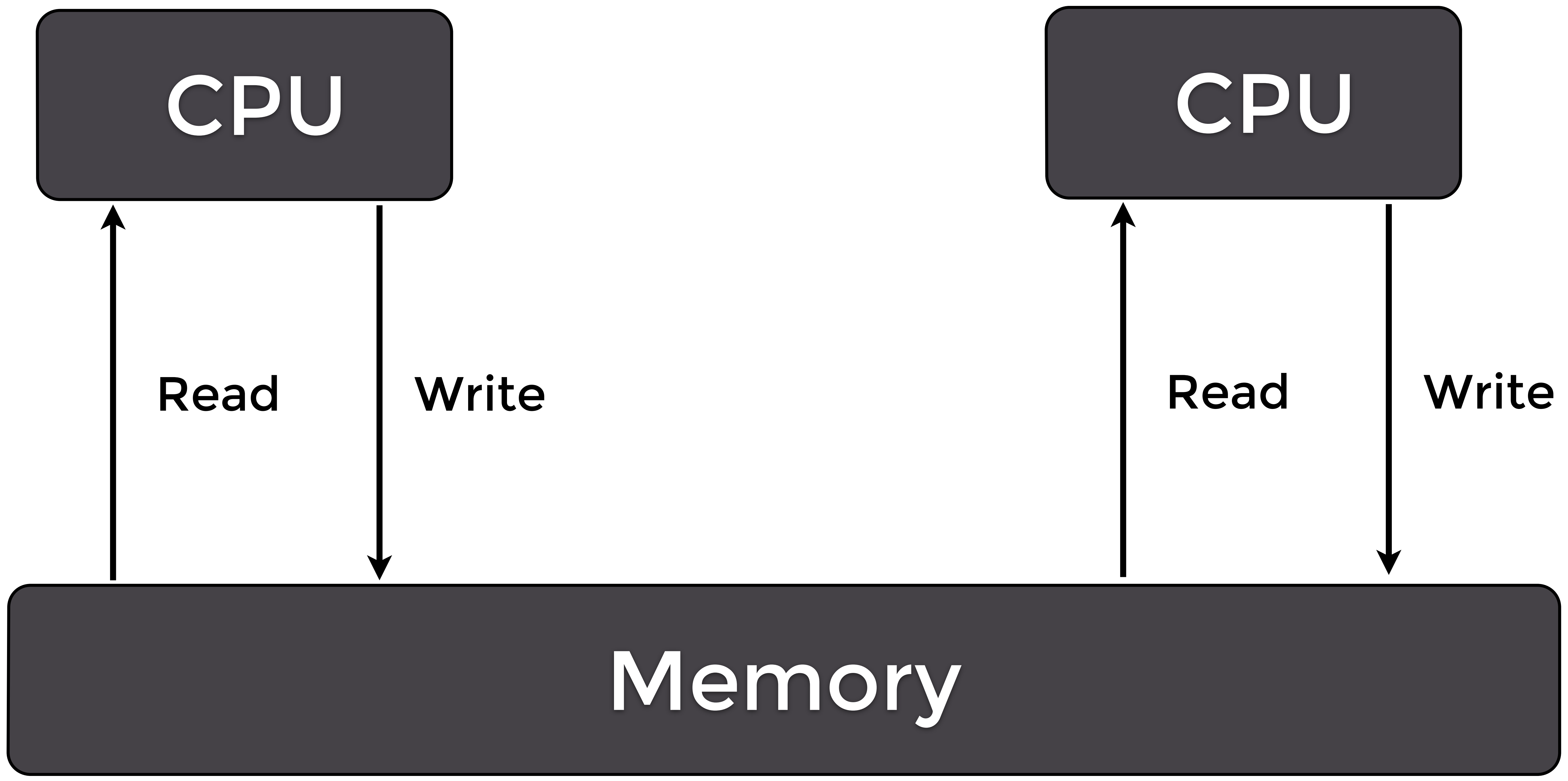
```
rd.issue(x)
rd.issue(y)

resp.mov(r1)
resp.mov(r2)
incr r1
wr.async(r1, x)

incr r2
wr.async(r2, y)
```

The Multiprocessor We Imagine

There are no caches or memory buffering here



Code Example 1

```
int x, y, a, b; // all zero
```

<u>CPU 1</u>

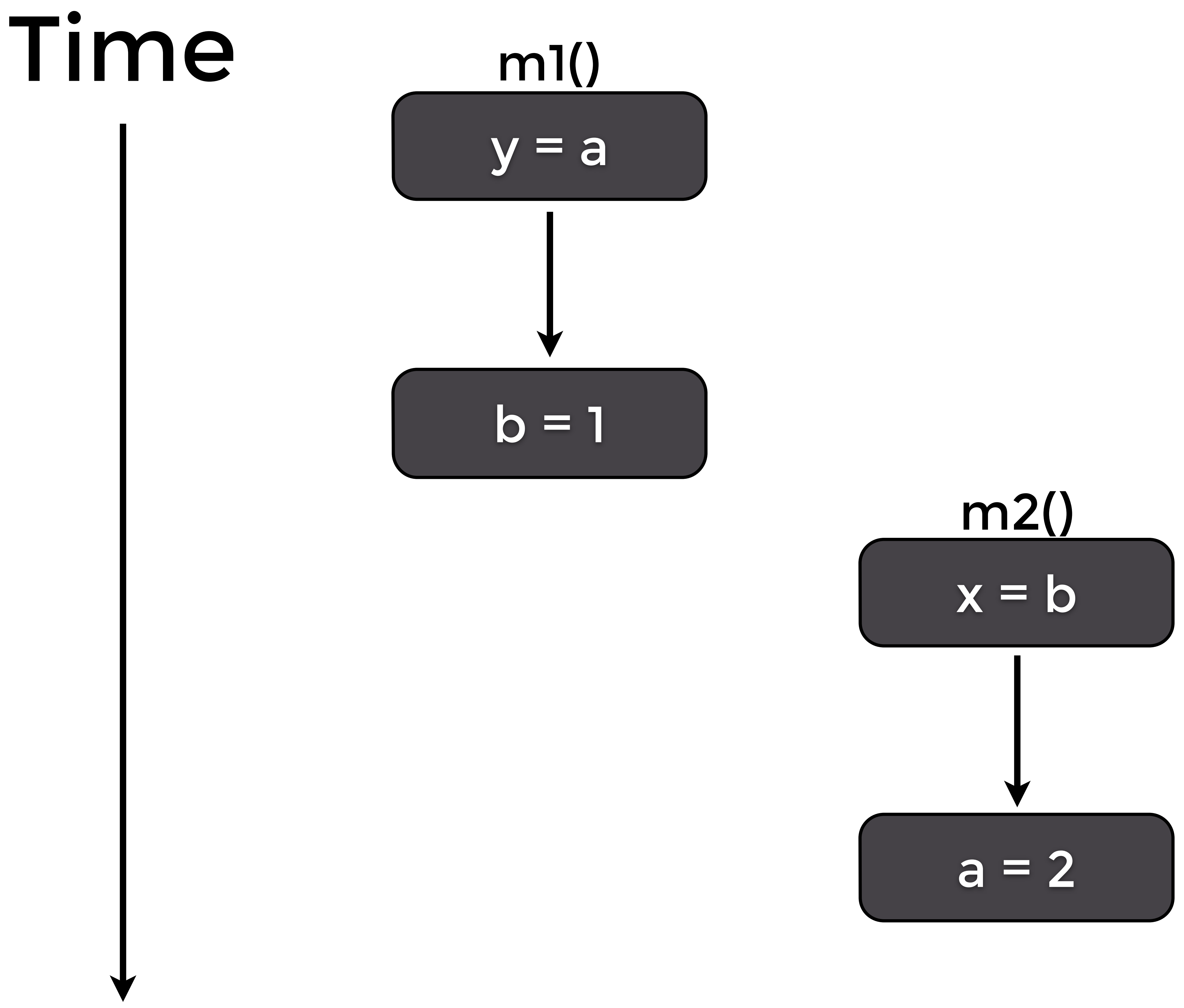
```
void m1() {
  y = a;
  b = 1;
}
```

<u>CPU 2</u>

```
void m2() {
  x = b;
  a = 2;
}
```

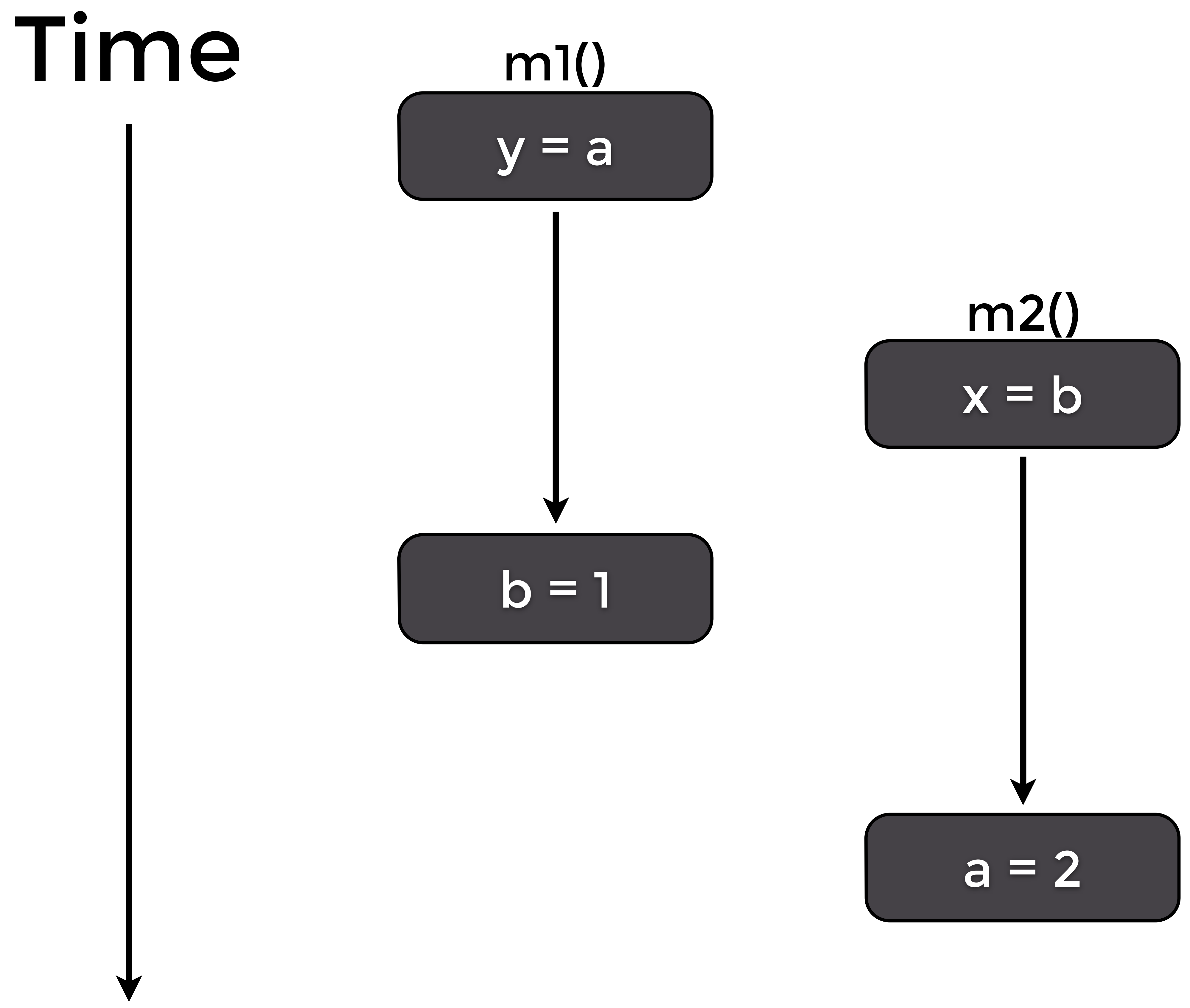
Possible outcomes for x and y?

Possible Trace 1



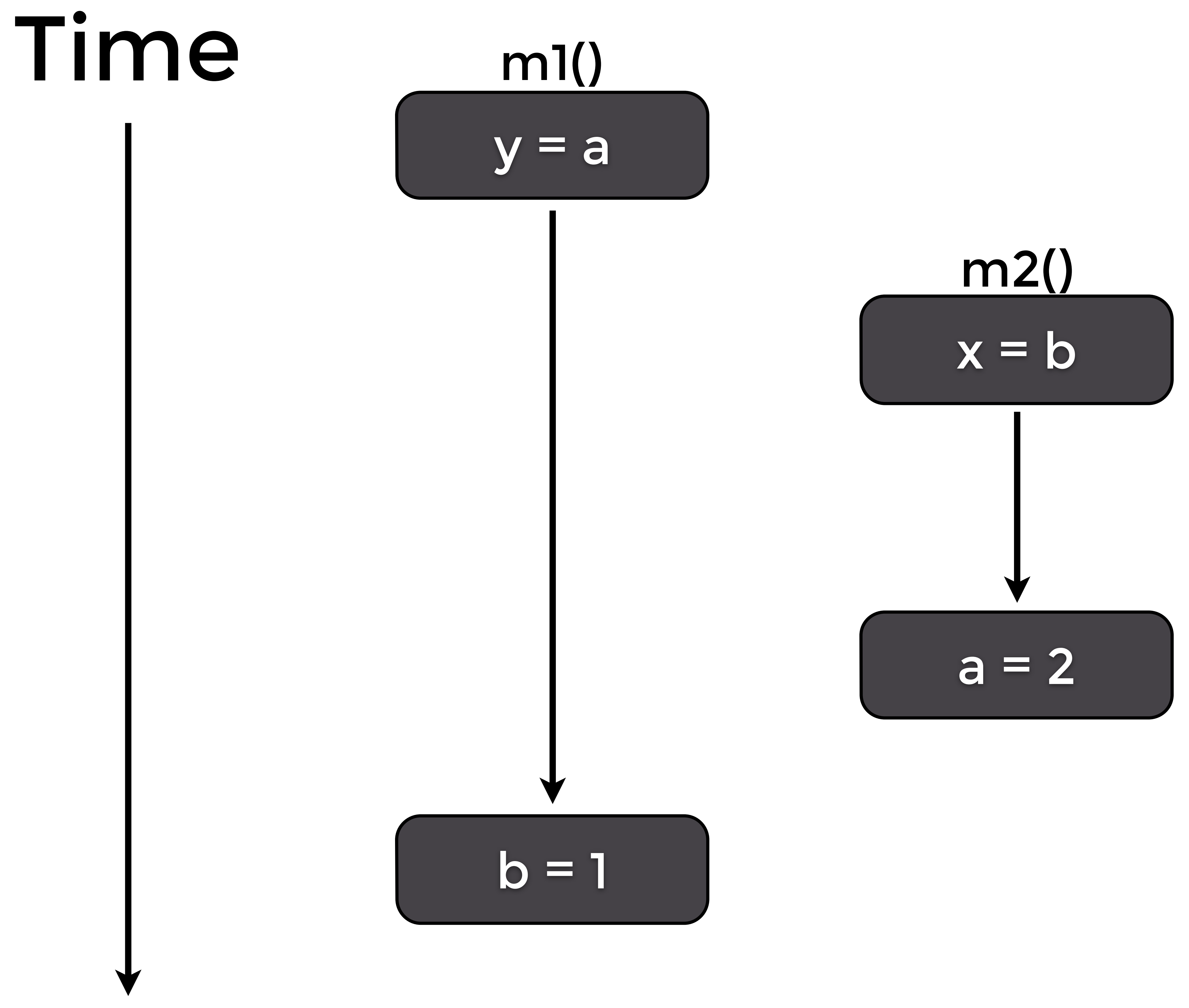
Outcome: x == 1, y == 0

Possible Trace 2



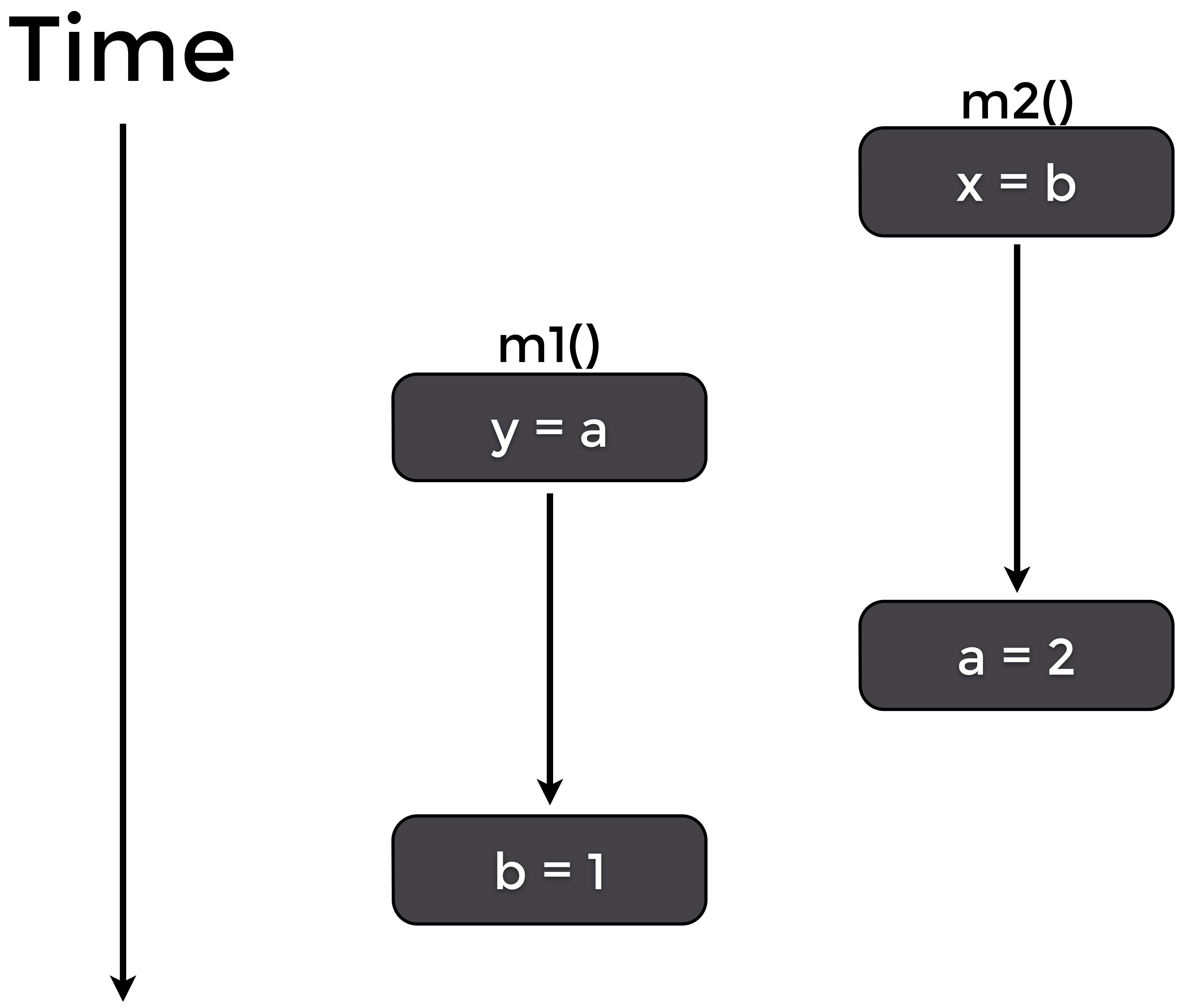
Outcome: x == 0, y == 0

Possible Trace 3



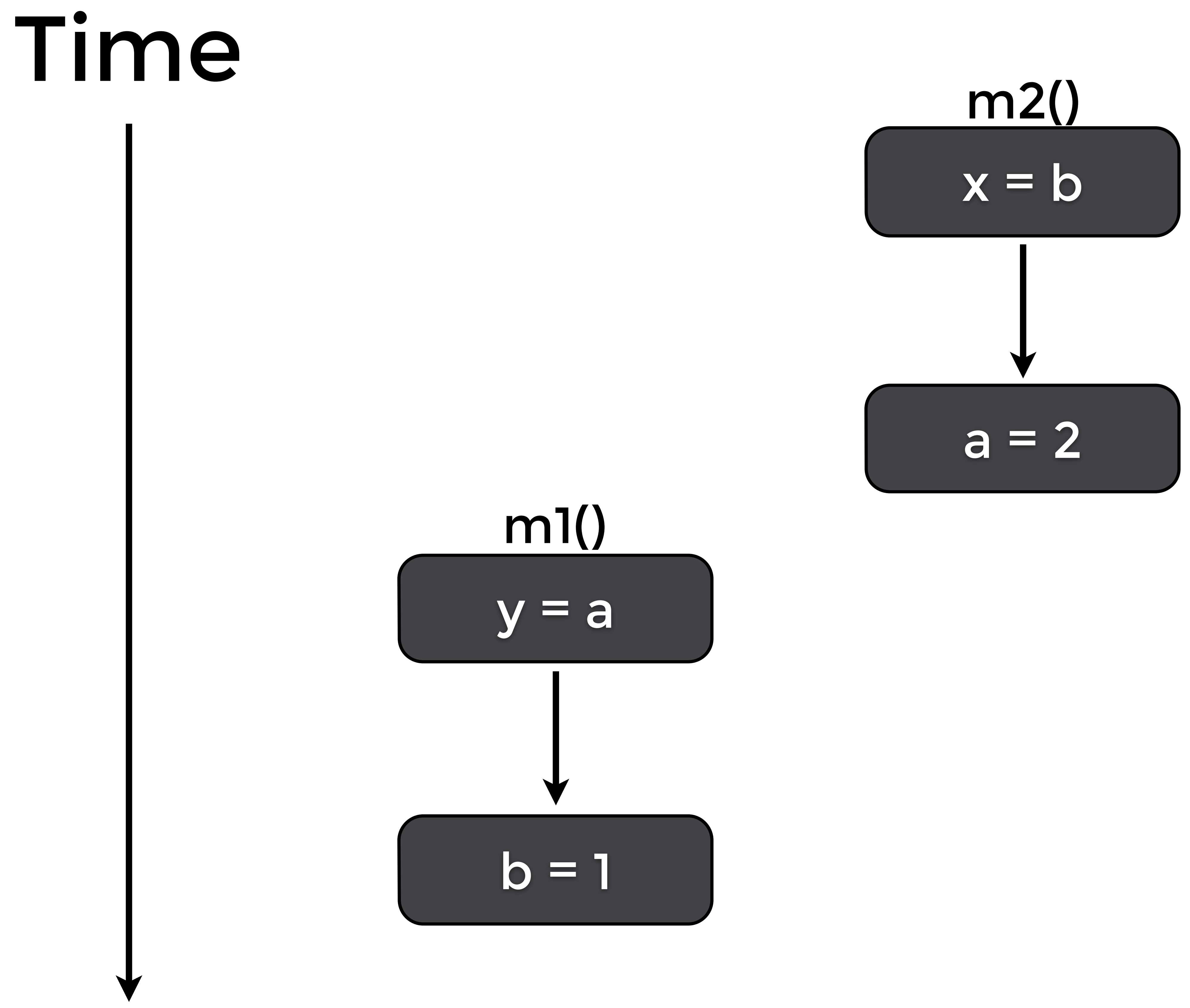
Outcome: x == 0, y == 0

Possible Trace 4



Outcome: x == 0, y == 0

Possible Trace 5



Outcome: x == 0, y == 2

Is That It?

- It looks like x or y must be 0 in the result
  - Makes sense: the first statement of m1() grabs a 0, and so does the first statement of m2()
- Is our reasoning correct?

```
int x, y, a, b; // all zero
void m1() {         void m2() {
  y = a;              x = b;
  b = 1;              a = 2;
}                   }
```

Surprisingly, No

Counterintuitively, the compiler can reverse the order

```
void m1() {
```

$$\begin{tikzcd}[column sep=huge]
\texttt{y = a;} \arrow[dr] & \texttt{mov \#1, mem.b} \\
\texttt{b = 1;} \arrow[ur] & \texttt{mov mem.a, mem.y}
\end{tikzcd}$$

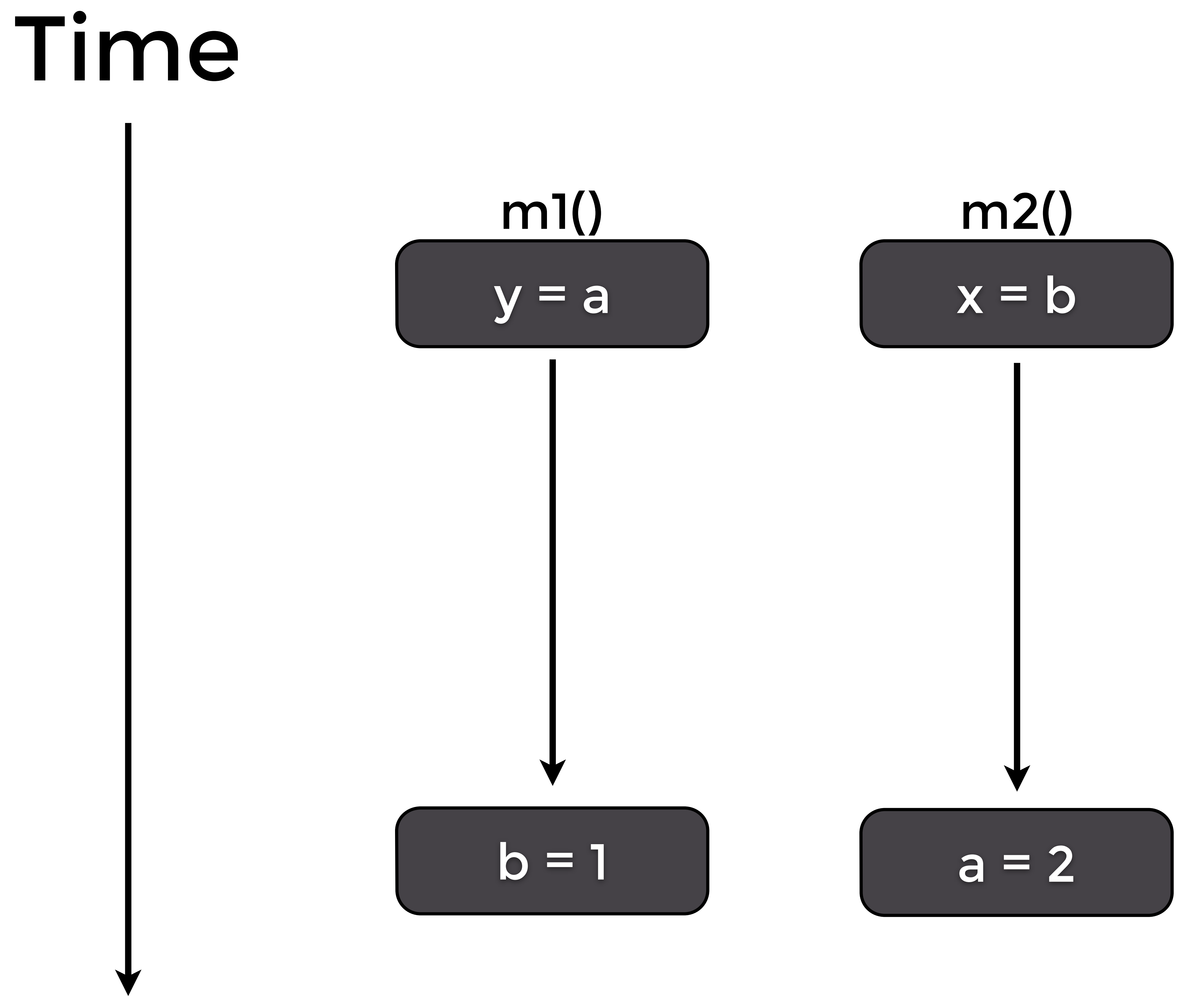
```
}
```

```
void m2() {
```

$$\begin{tikzcd}[column sep=huge]
\texttt{x = b;} \arrow[dr] & \texttt{mov \#2, mem.a} \\
\texttt{a = 2;} \arrow[ur] & \texttt{mov mem.b, mem.x}
\end{tikzcd}$$

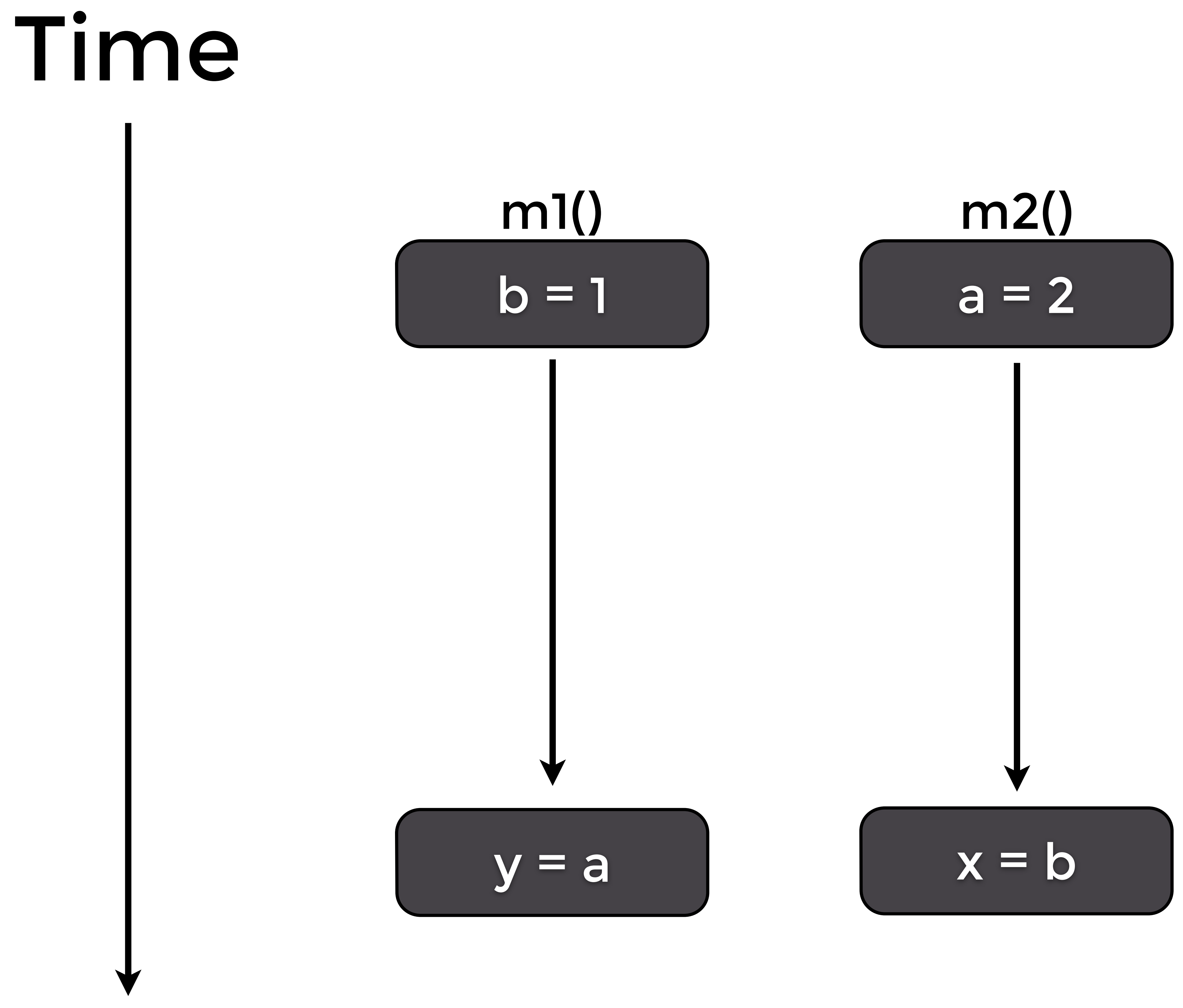
```
}
```

Intuitive Trace



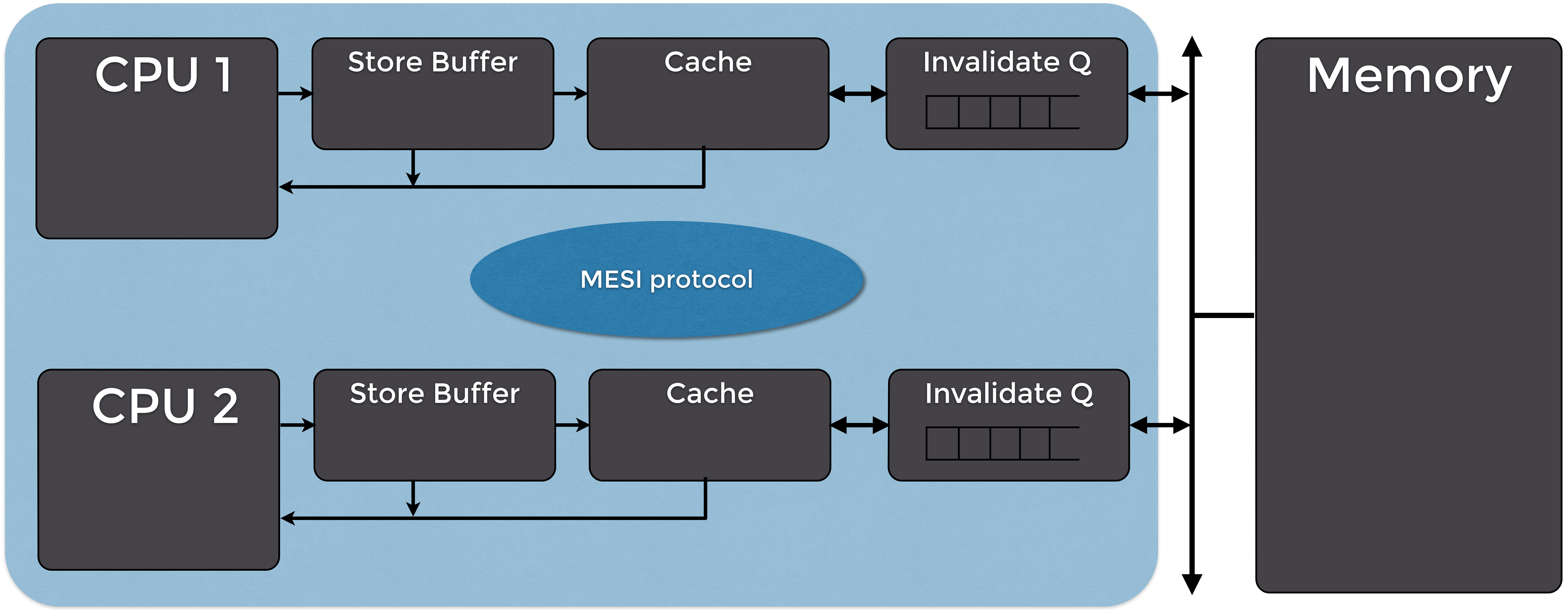
Outcome: x == 0, y == 0

Surprising Trace



Outcome: x == 1, y == 2

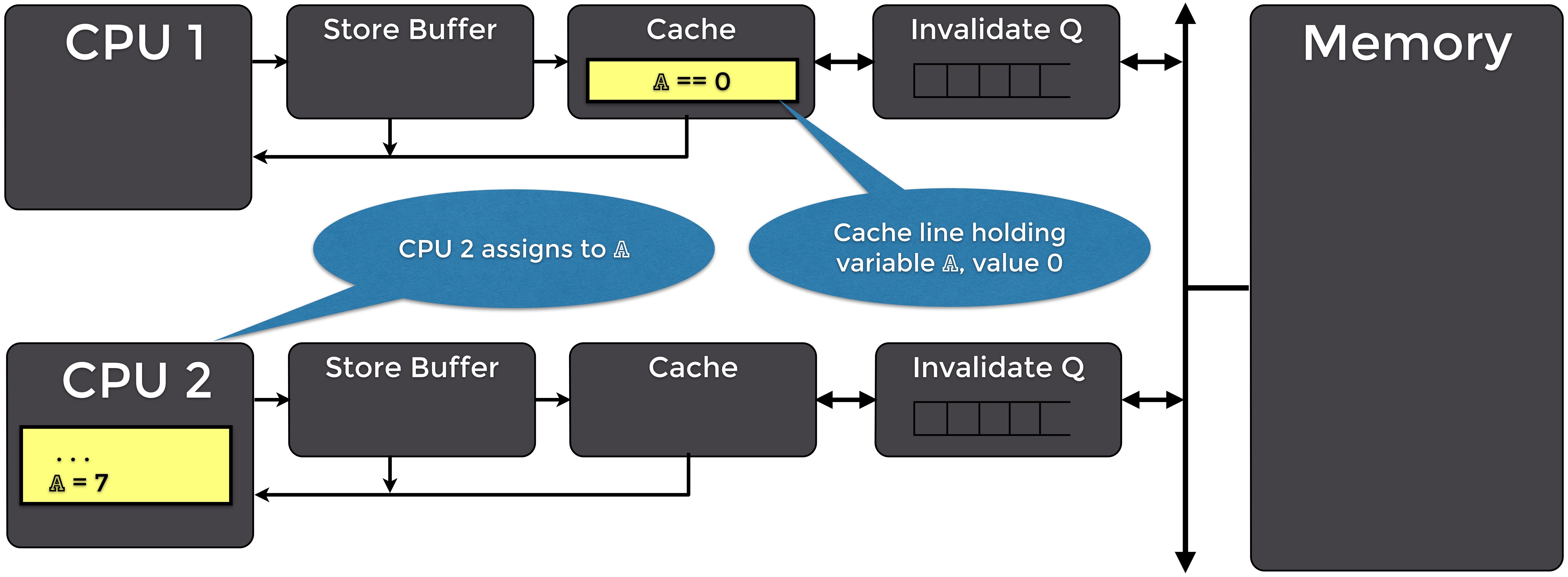
And It Gets Worse …



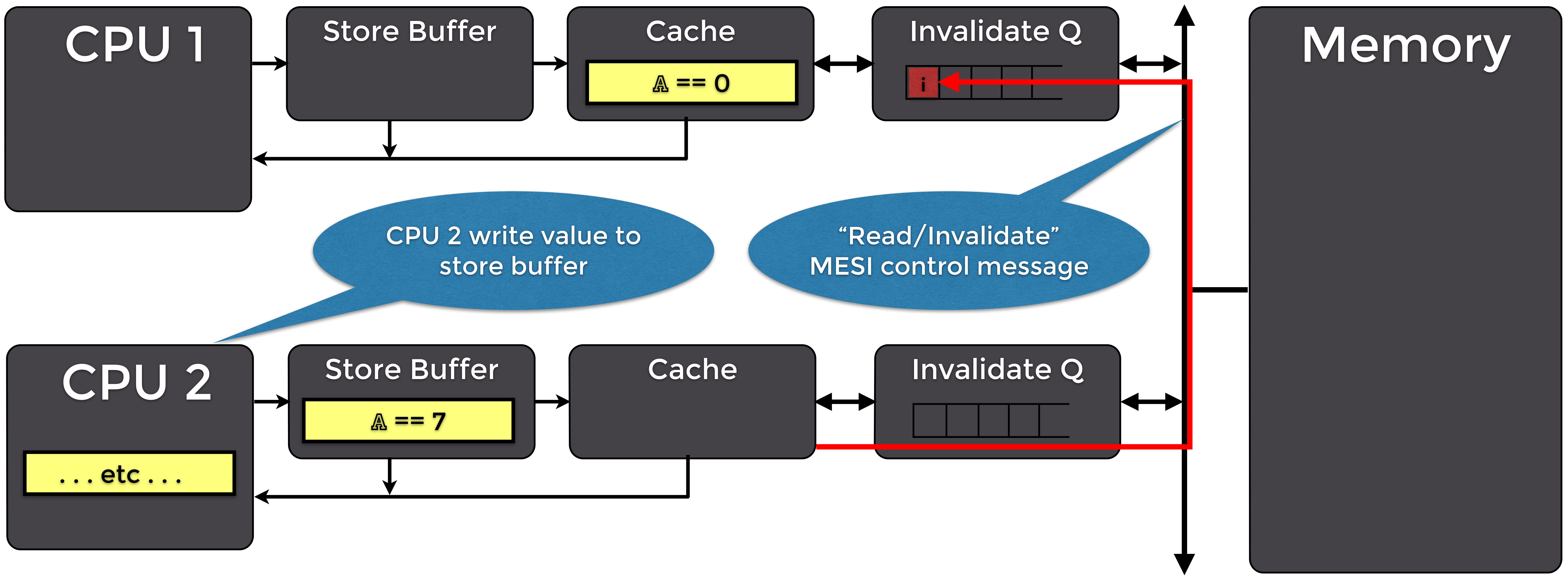
MESI Protocol

- Widely-known cache coordination protocol
- Acronym for cache line states:
  - Modified Exclusive Shared Invalid
- Transfers cache-line “messages” between processor caches
- Typically coordinated by parallel signaling “bus” within chip or single board

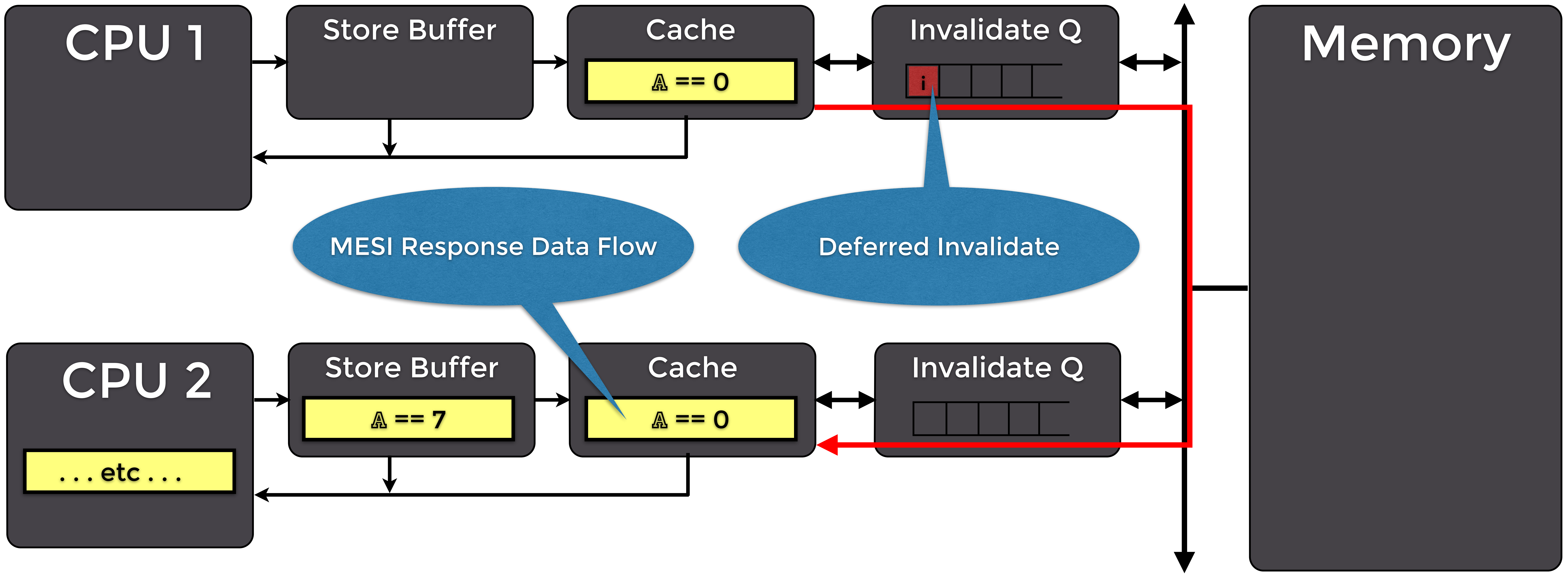
MESI Example 1-1



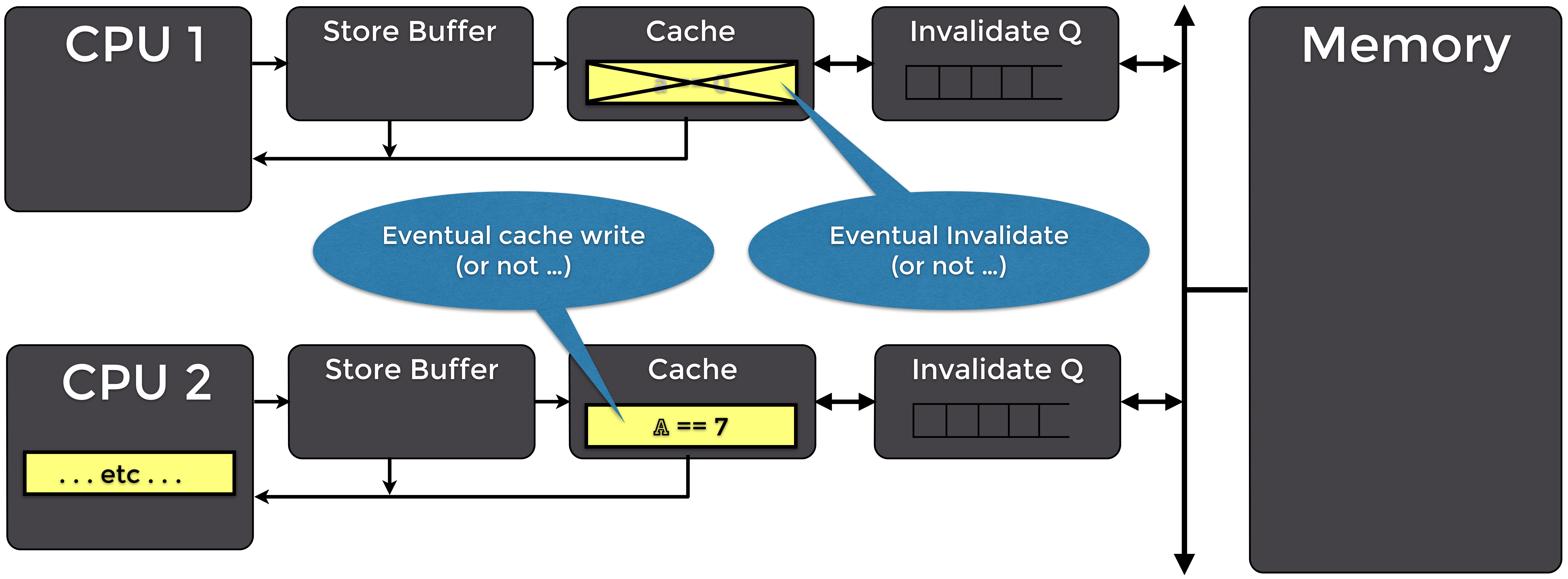
MESI Example 1-2



MESI Example 1-3



MESI Example 1-4



MESI Example 2-1

Credit: http://bit.ly/pjug2013-mckenney-parallel

```
int A, B; // both zero
```

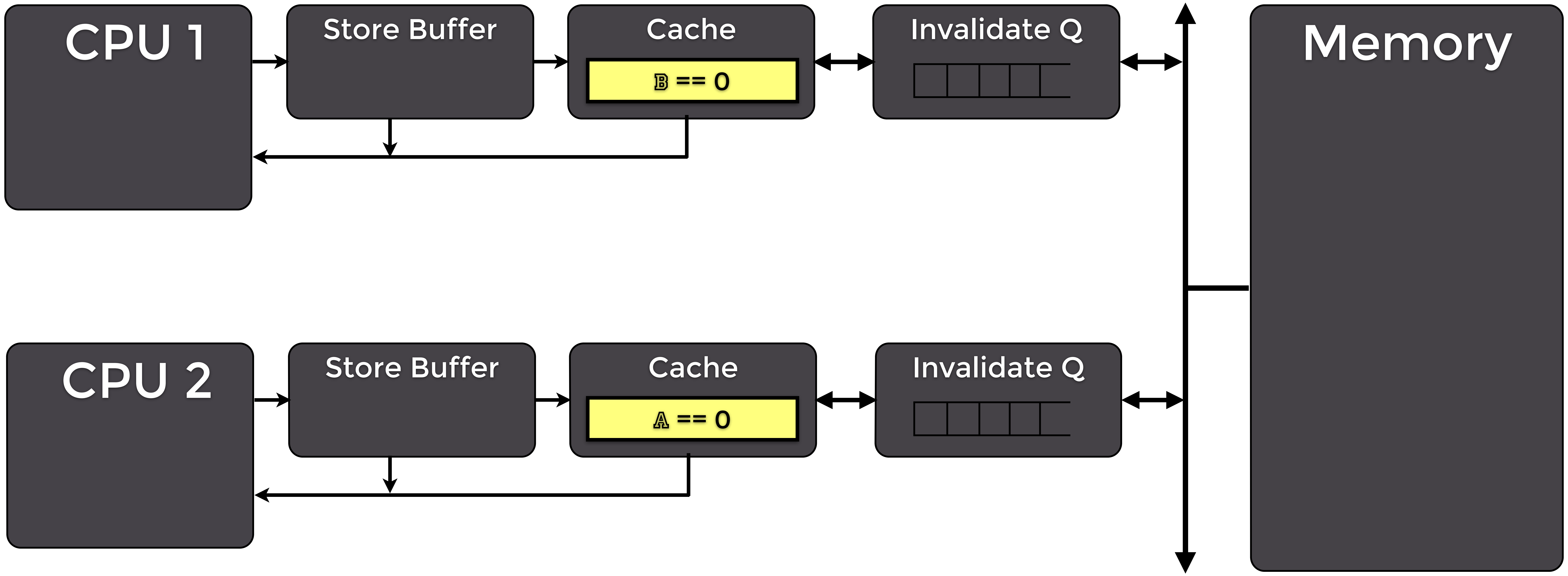
CPU 1

```
void m1() {
  A = 1;
  B = 1;
}
```

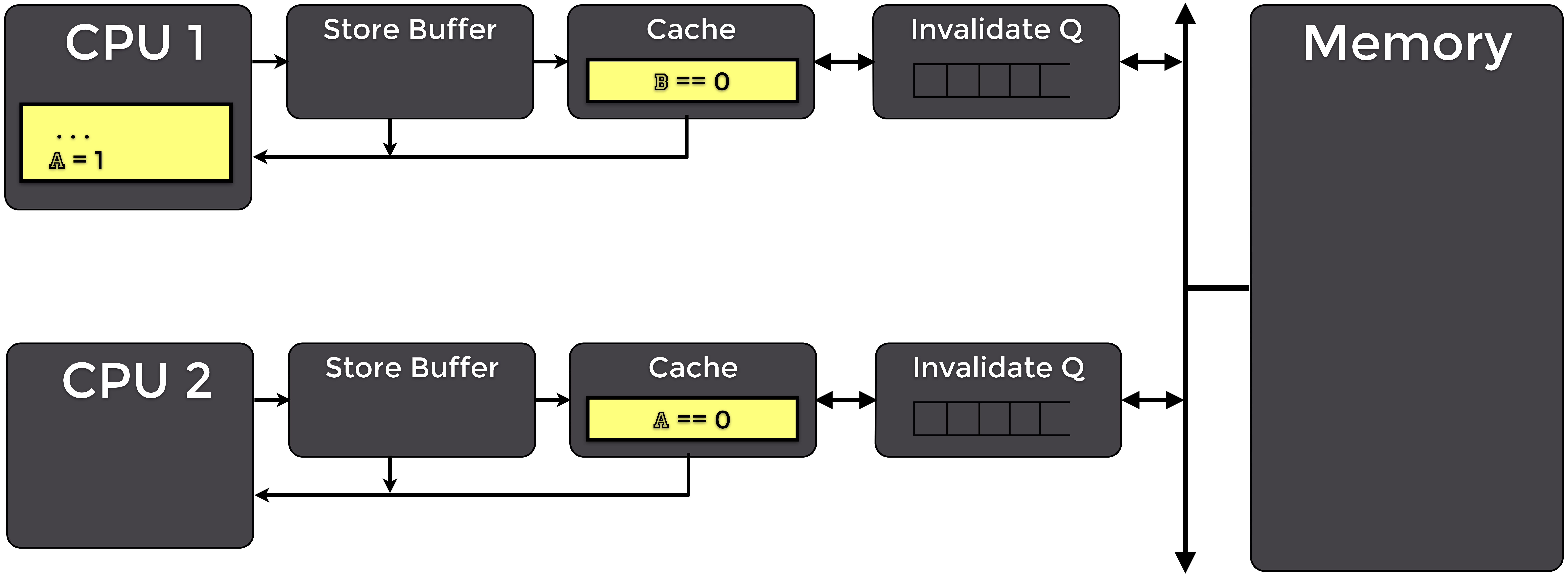
CPU 2

```
void m2() {
  while (B == 0)
        ;
  assert(A == 1);
}
```

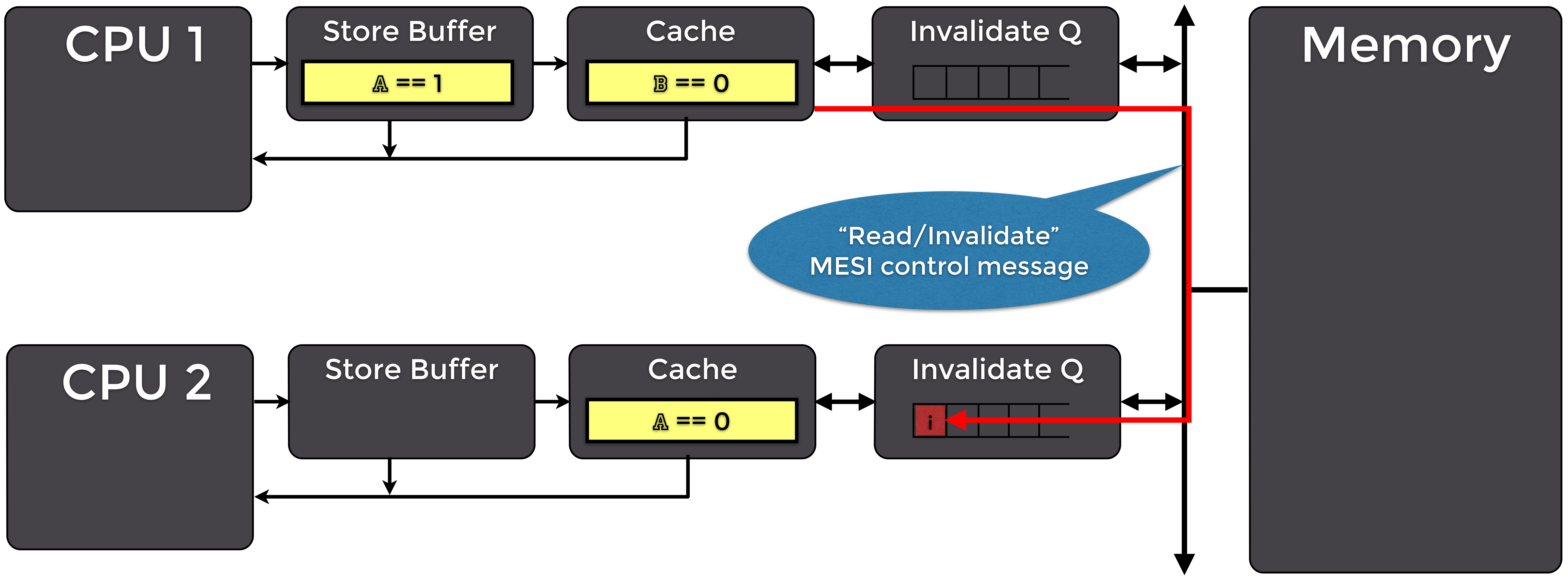
MESI Example 2-2



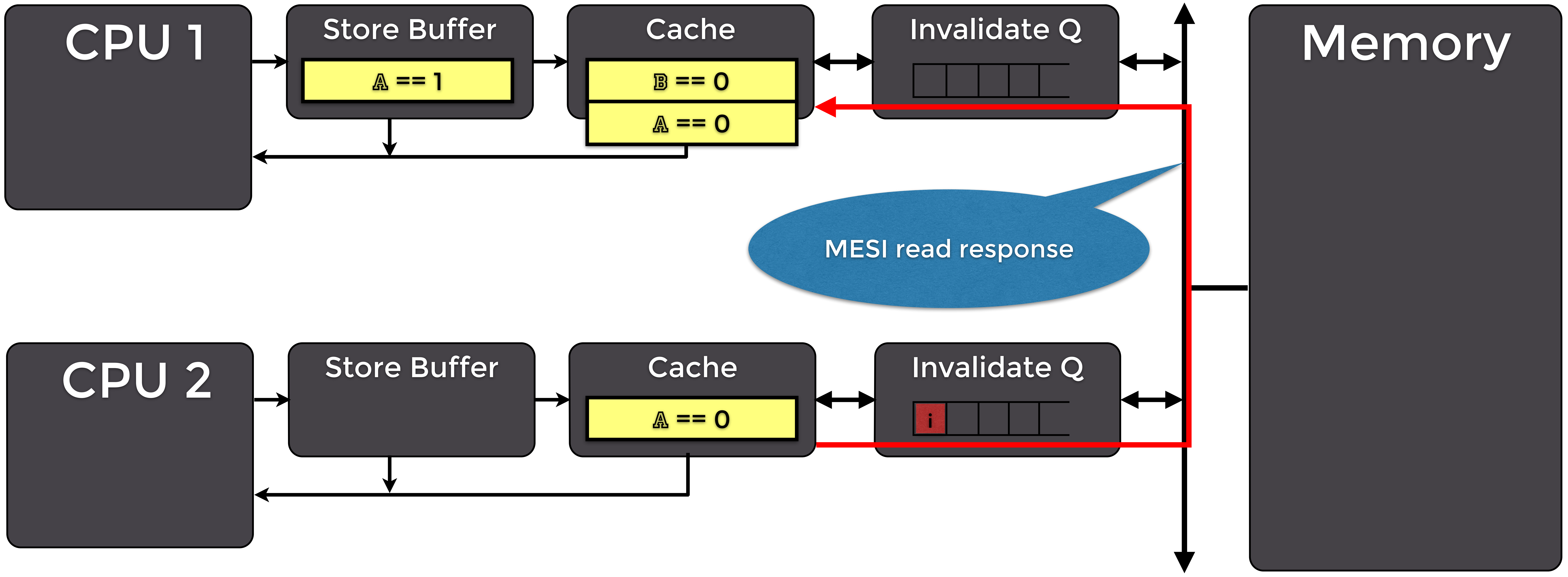
MESI Example 2-3



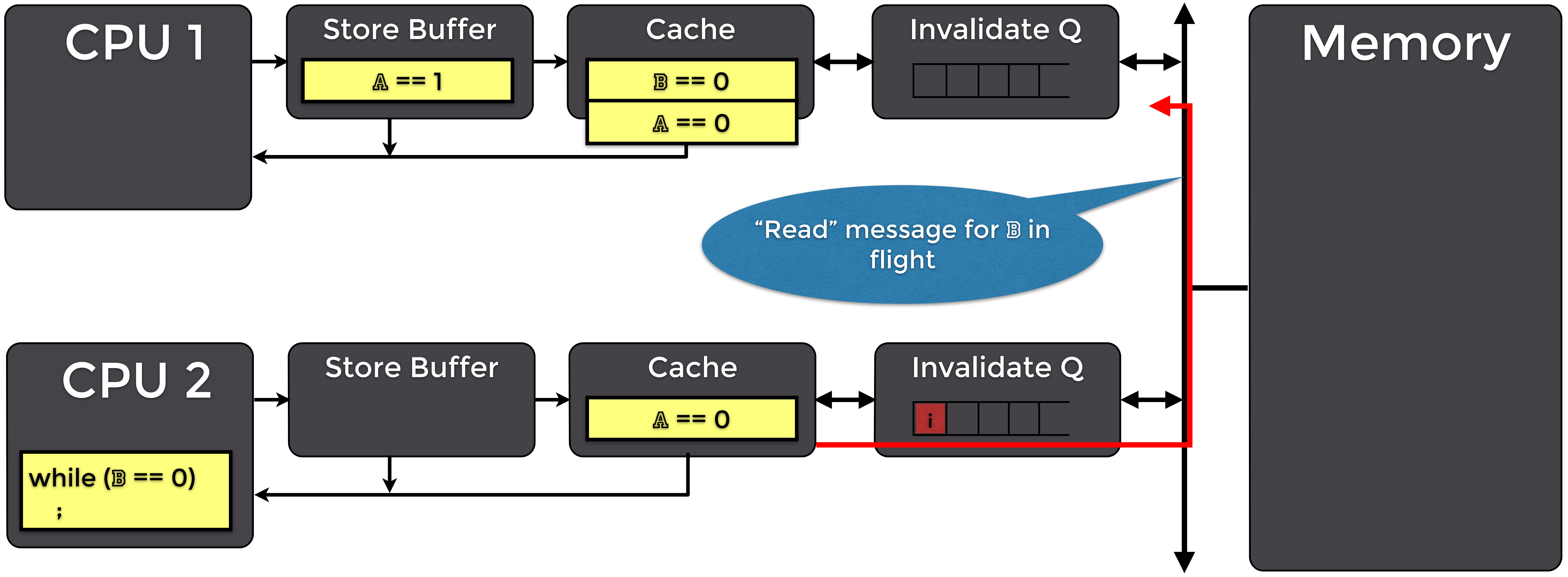
MESI Example 2-4



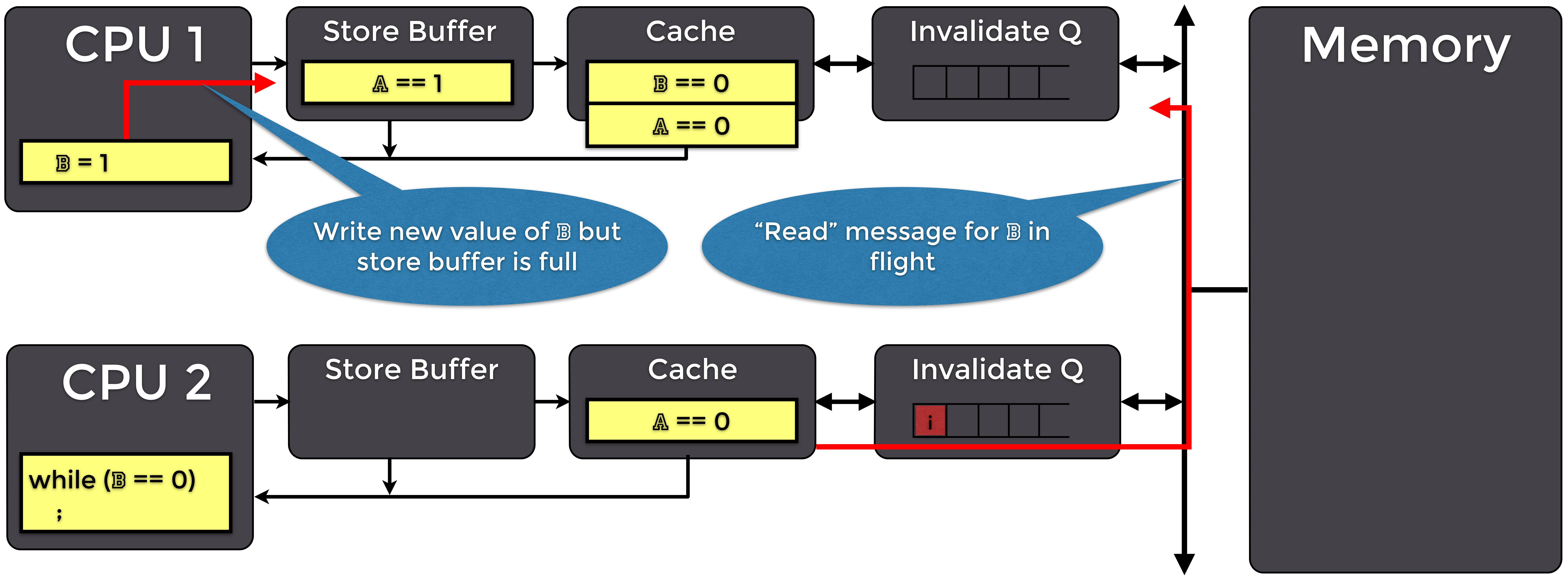
MESI Example 2-4



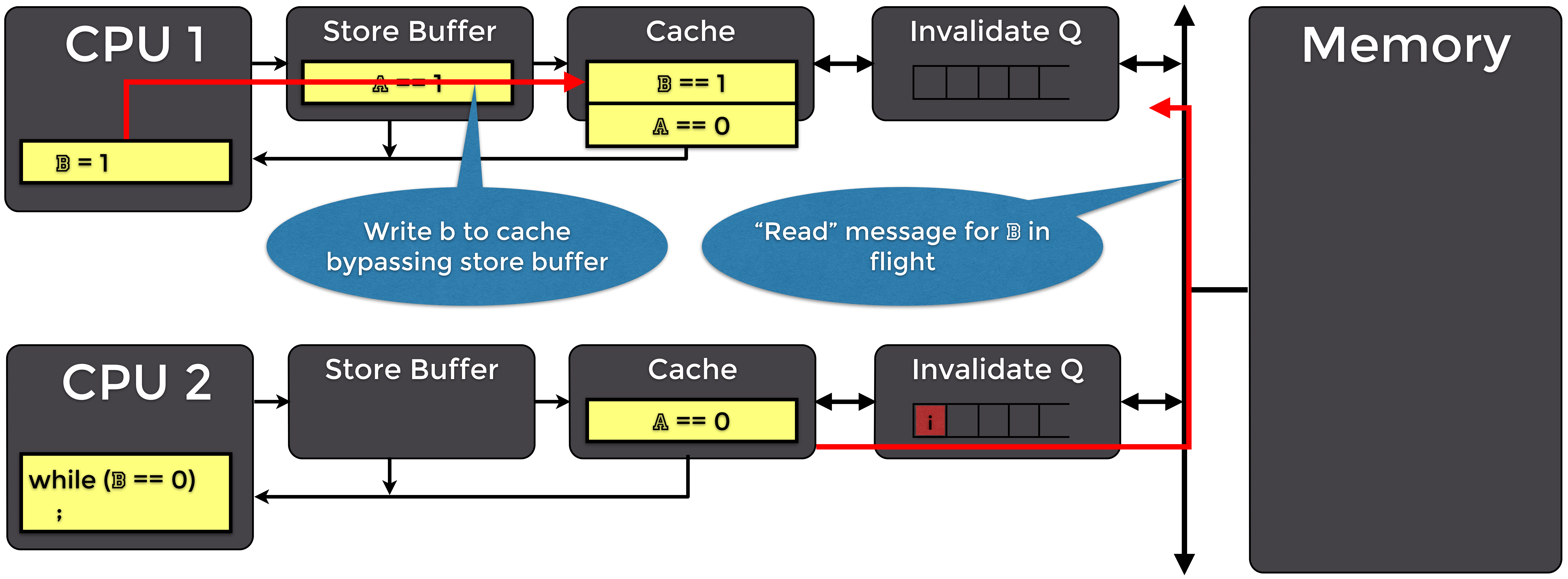
MESI Example 2-5



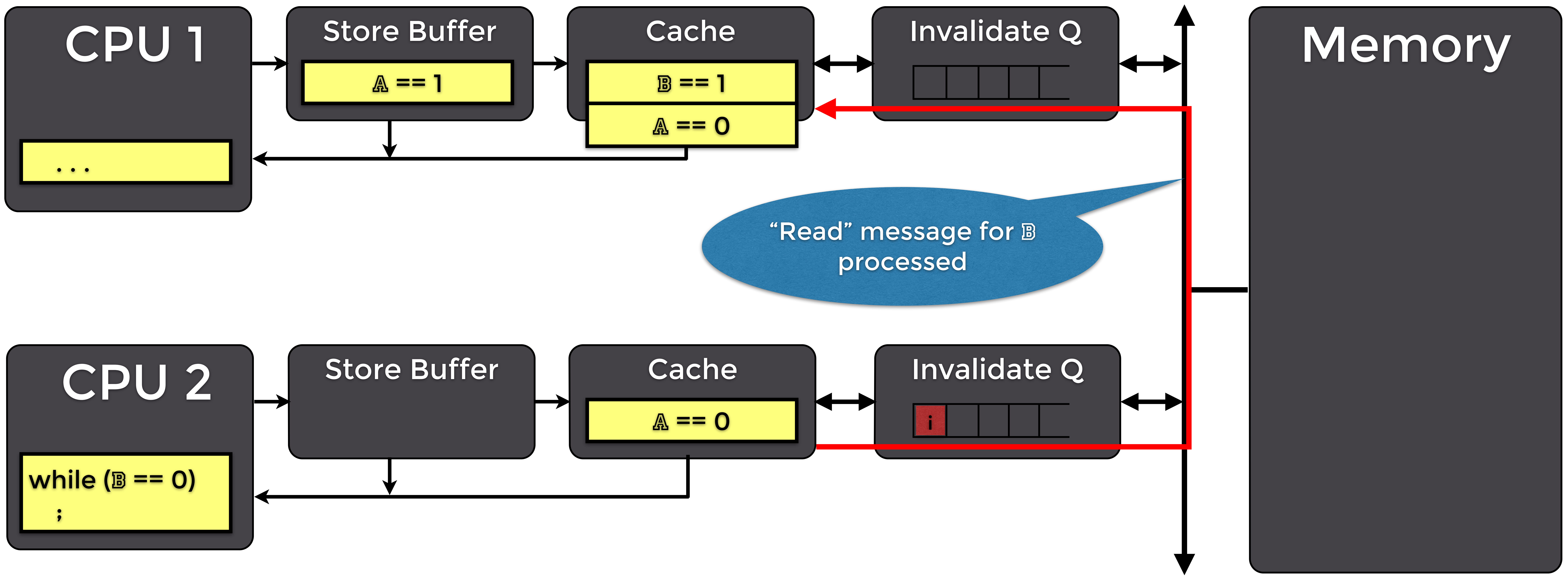
MESI Example 2-6



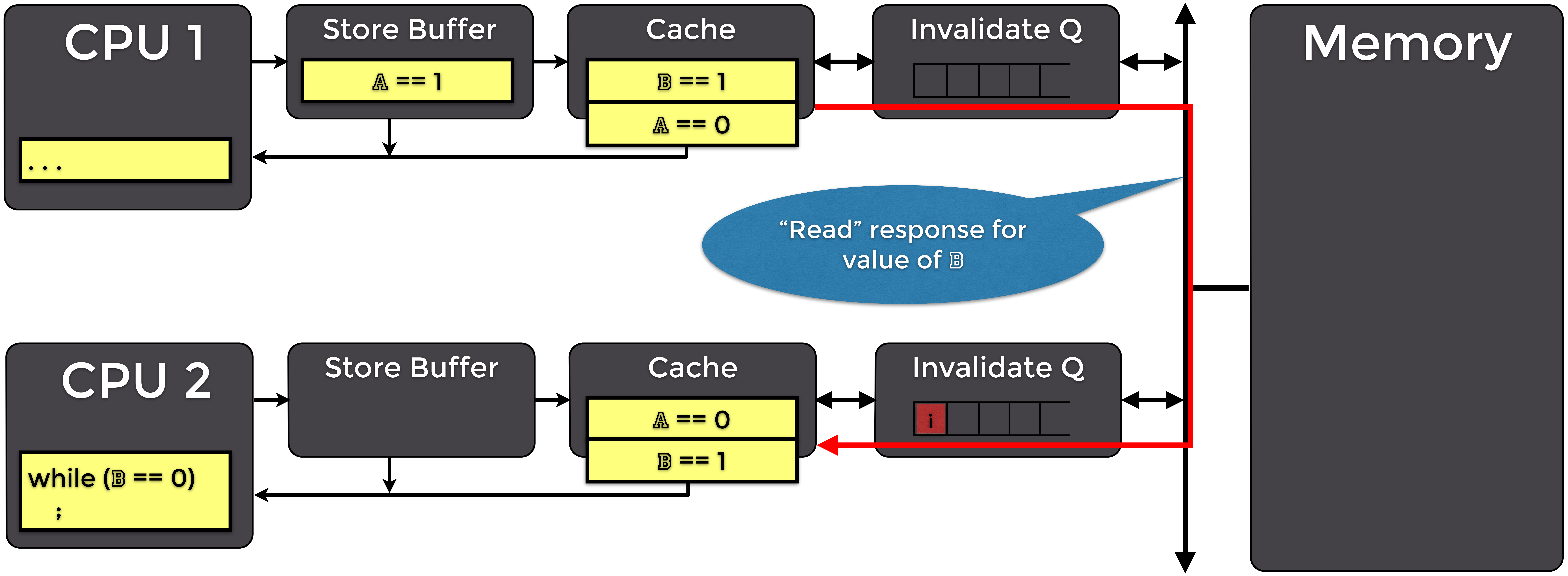
MESI Example 2-6



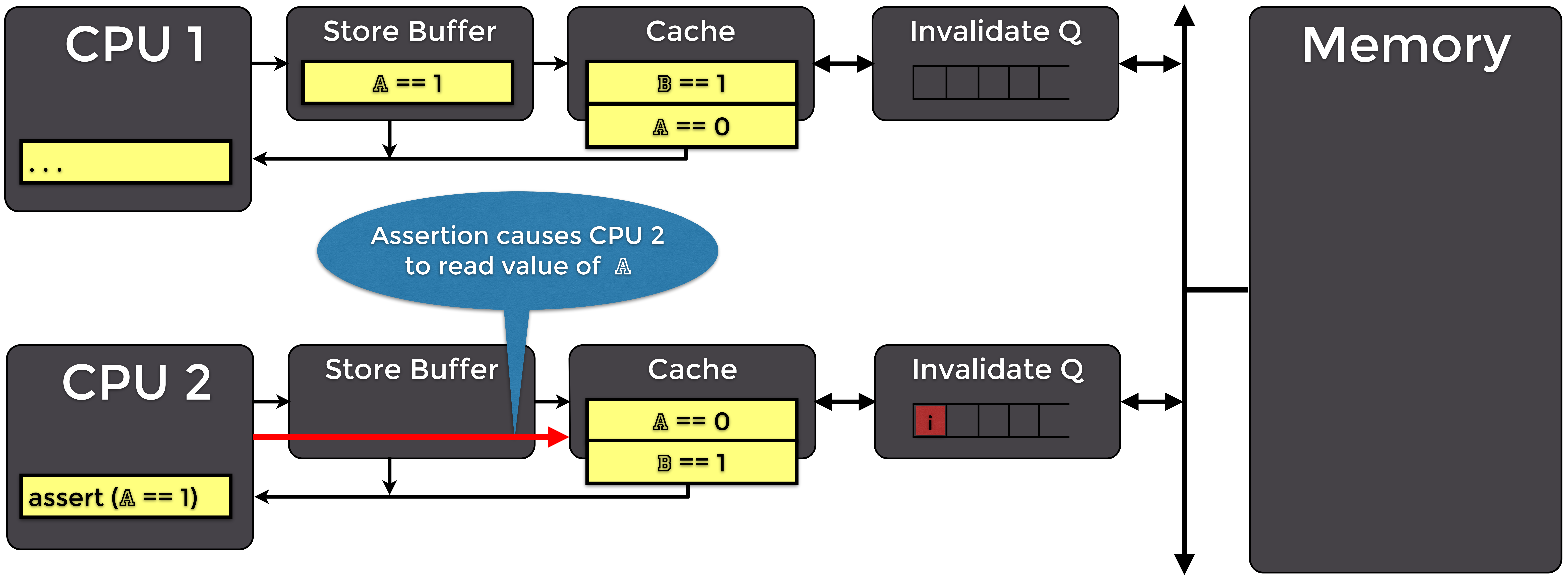
MESI Example 2-7



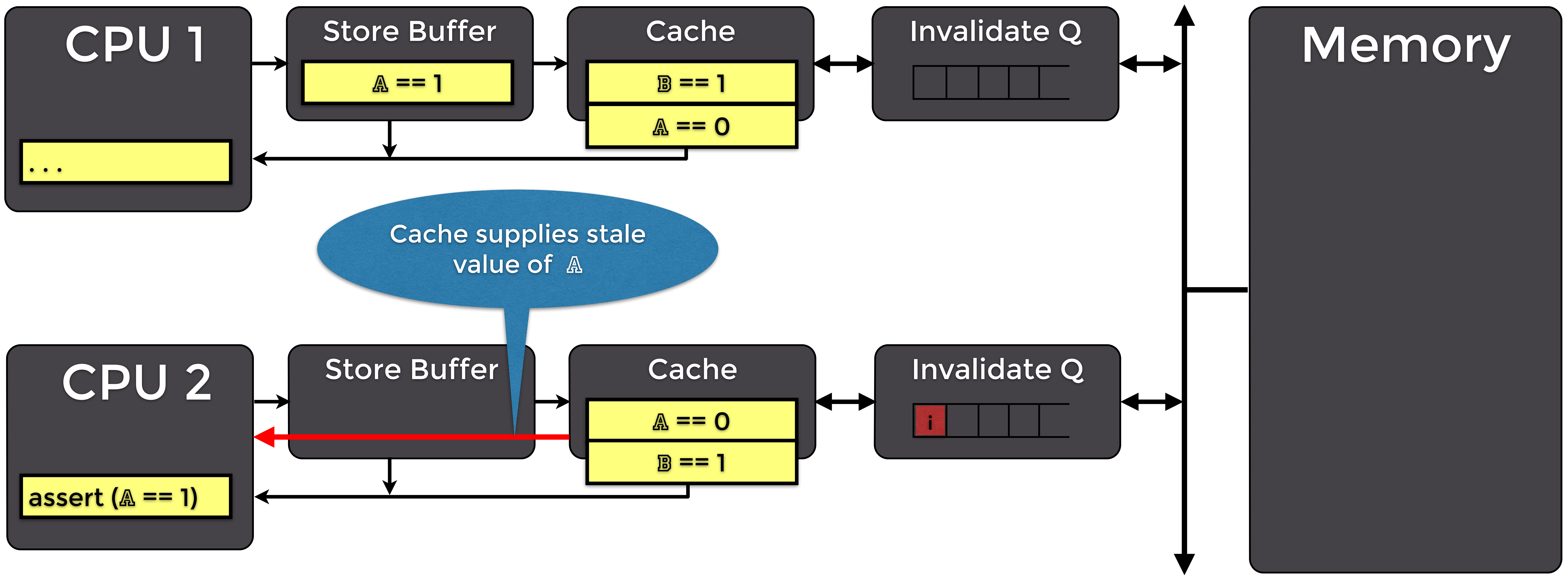
MESI Example 2-8



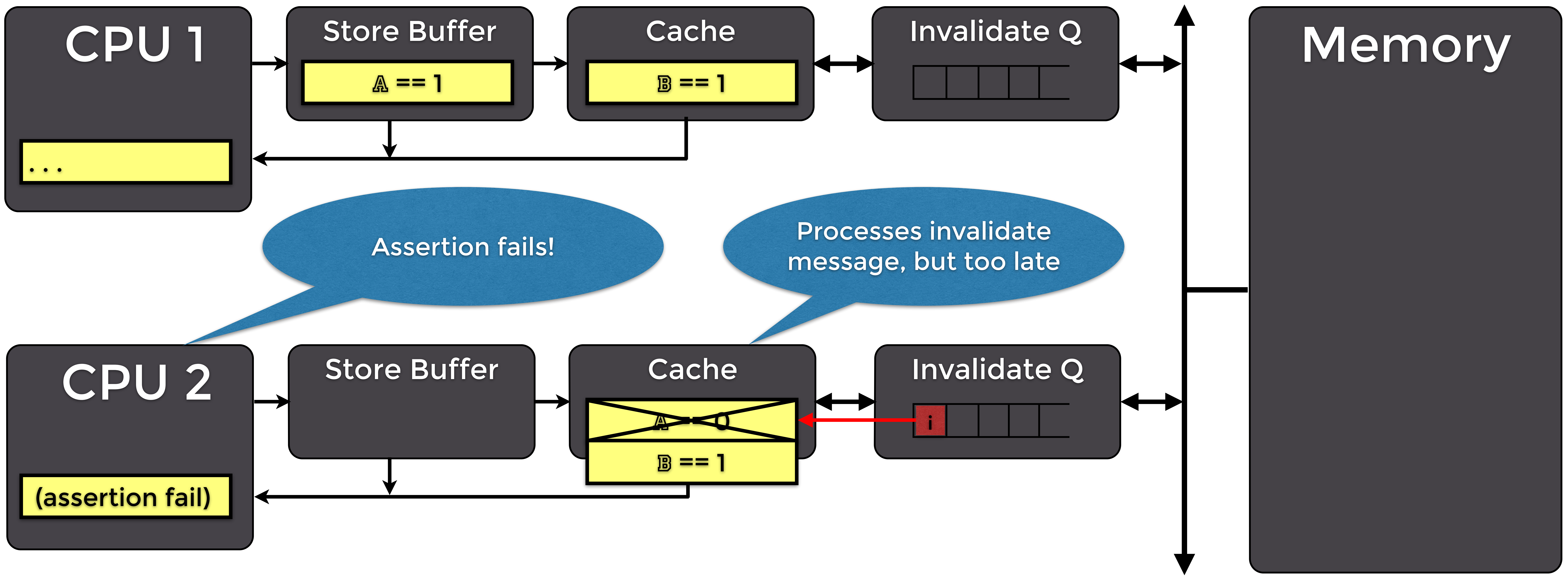
MESI Example 2-9



MESI Example 2-9



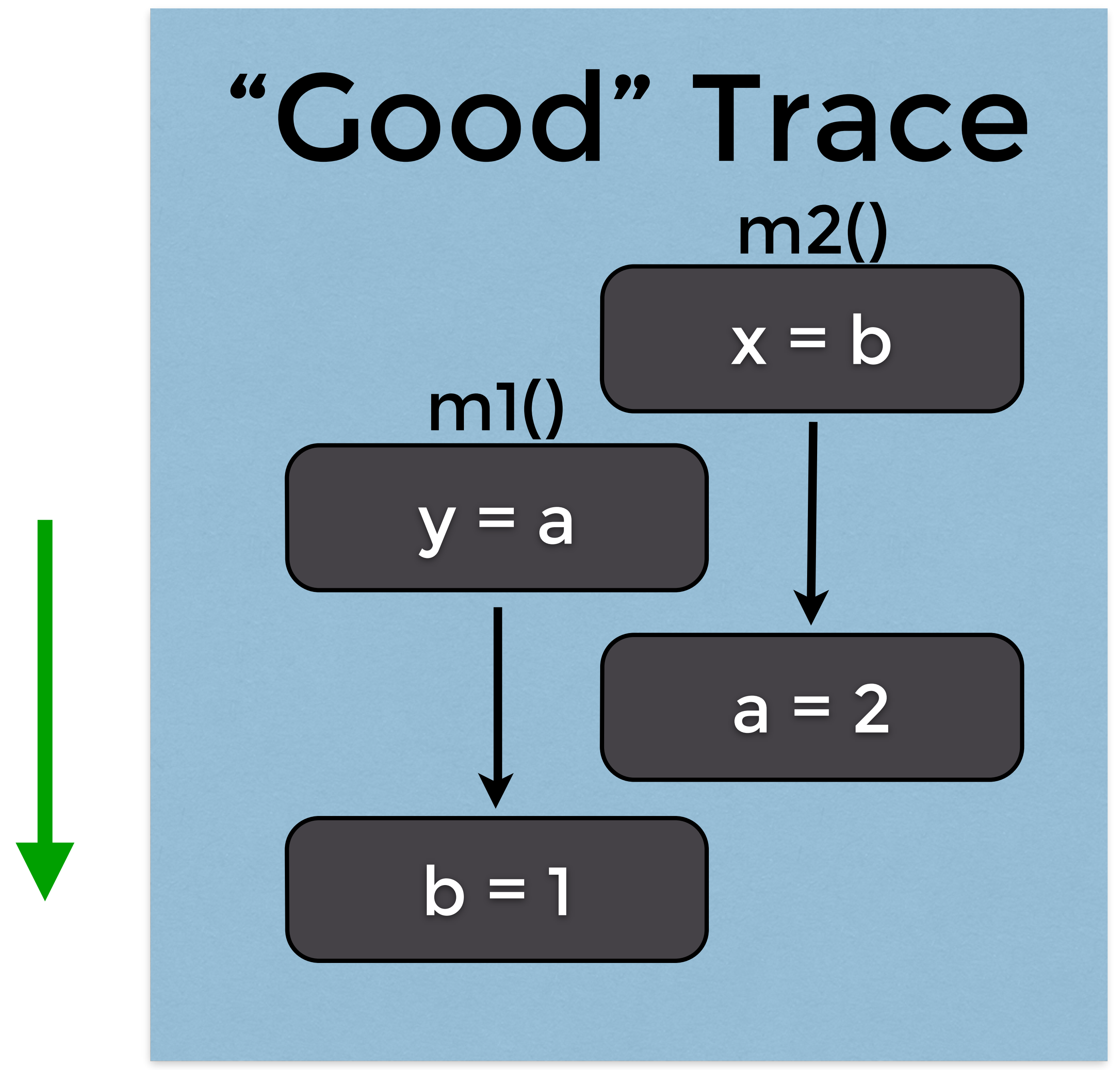
MESI Example 2-10



Where Are We?

- Some concurrent traces (“Good” traces) seem much more intuitive than others

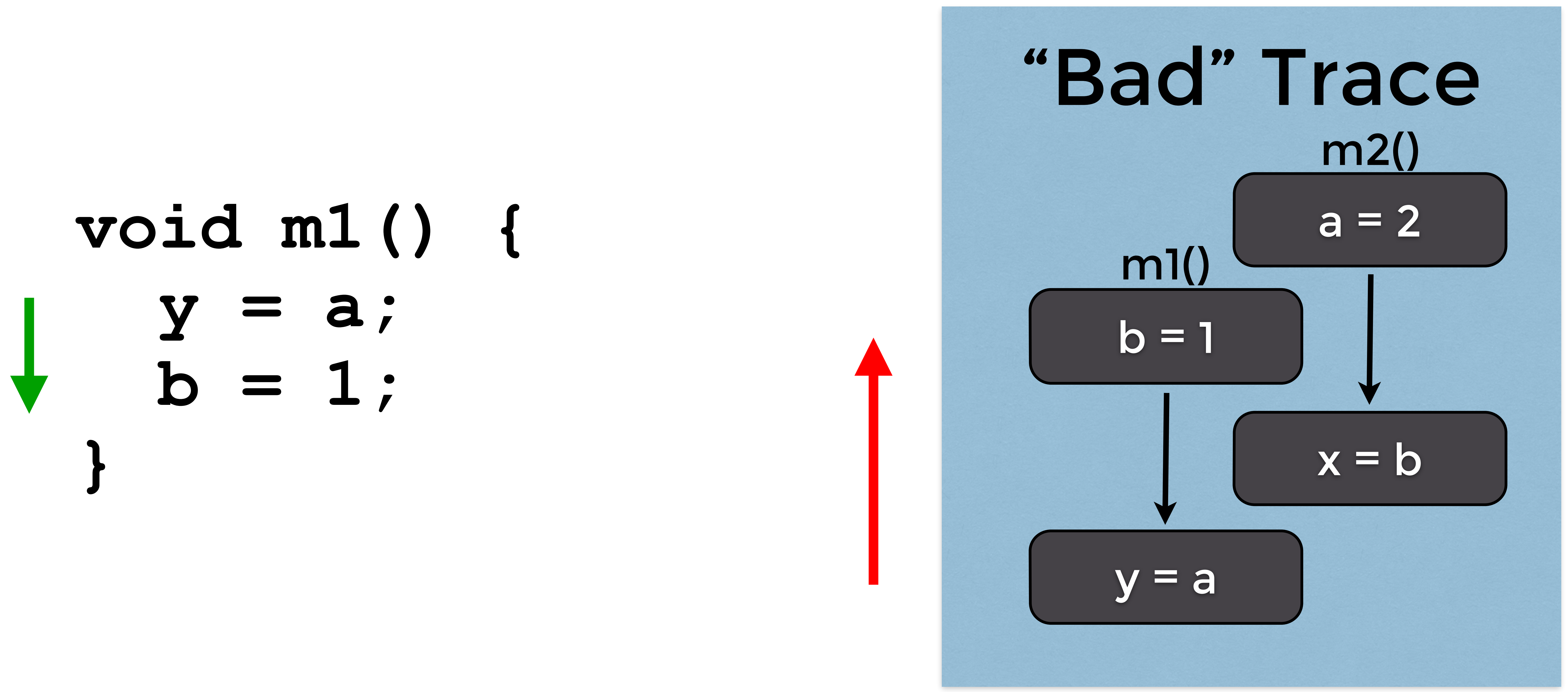
```
void m1() {
  y = a;
  b = 1;
}
```



Others Not So Much

- “Bad” traces don’t correspond to any possible sequential execution of the original statements

```
void m1() {
  y = a;
  b = 1;
}
```



Sequential Consistency

- “Good” traces correspond to **some sequential execution** of the original language statements
- The concept of **some sequential execution** can be formalized as **sequential consistency** or **SC**.
- “Bad” traces can be prevented by specifying rules allowing programmers to ensure their code is SC.

Java Memory Model (JMM)

- Early Java was broken
- JMM introduced in Java 1.5 (2004)
- Now section 17.4 and 17.5 of JLS
- Based on the concept of a **partial order**
  - Most memory operations are unordered
- Abstract **Happens-before** operator defines ordering of specific memory operations

Typical Rules from JMM

“Every memory operation on a given thread **happens-before** the next memory operation by the same thread in program order.”

“All memory operations prior to writing a volatile variable on one thread **happen-before** a read of the same volatile from another thread.”

Modified Example 1

```
volatile int a, b, x, y;
```

CPU 1

```
void m1() {
   y = a;
   b = 1;
}
```

y == a must be visible to any thread that can observe b == 1

CPU 2

```
void m2() {
   x = b;
   a = 2;
}
```

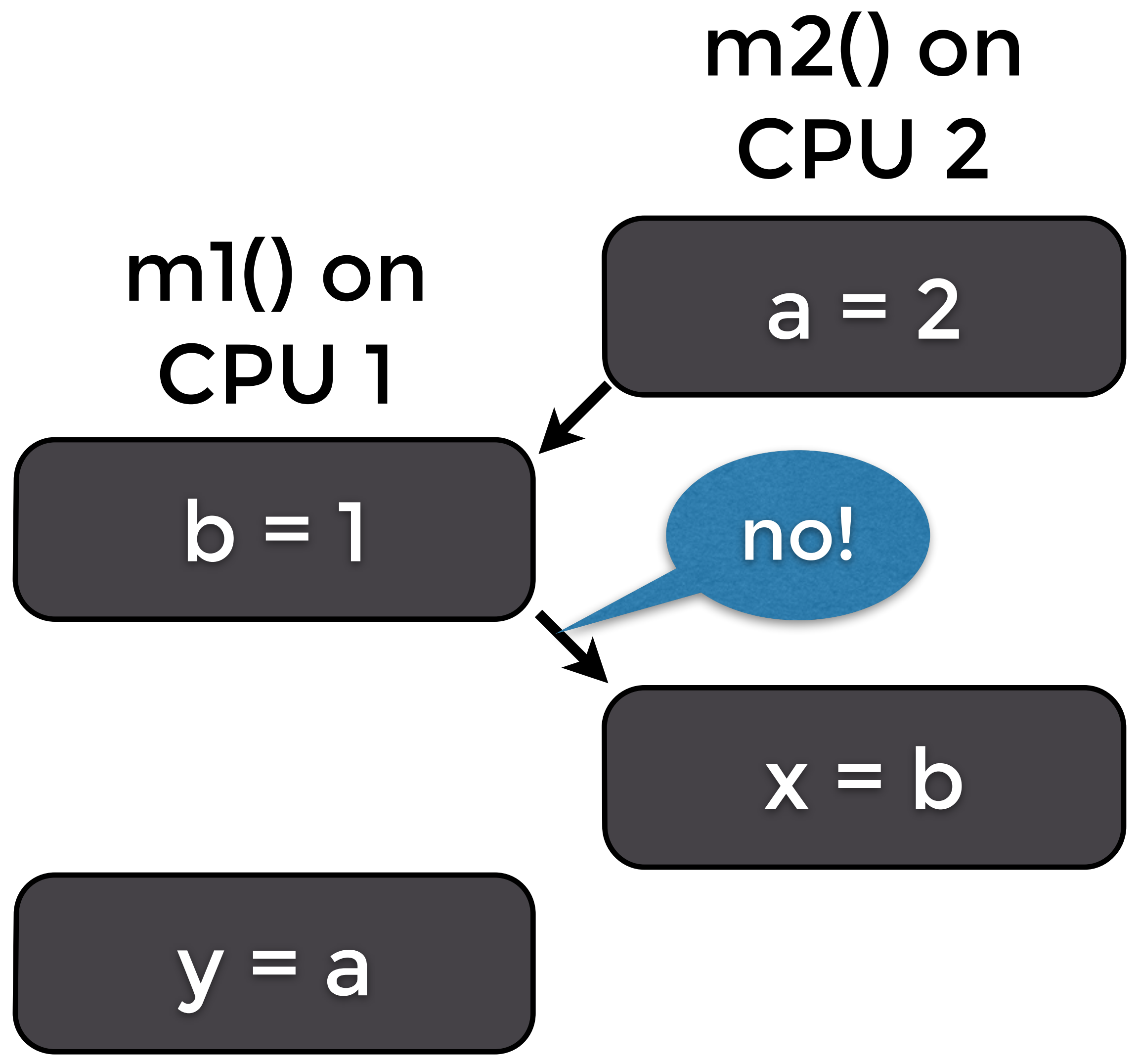
x == b must be visible to any thread that can observe a == 2.

Result

The two **happens-before** operations mean that if CPU 2 can observe **b = 1**, it must also observe **y = a**.

The compiler and runtime cooperate to prevent the non-SC trace from occurring.

Surprising Trace Prevented



Rights and Responsibilities

- Programmer is responsible for ensuring the presence of a **happens-before** between every pair of references to a given datum.
- In exchange, JMM guarantees that program behavior will be **SC**
- Terminology: a missing happens-before is called a **data race**.

Ensuring Happens-Before

- Single-threaded code naturally has H-Bs
- To ensure H-Bs in concurrent code, use:
  - immutability (final) with safe publication
  - primitives (volatile, mutex, atomics)
  - concurrency-safe library classes
  - concurrency-safe frameworks and programming models, e.g. Akka

MESI Example 3-1

MESI Example 2 but modified with volatile

```
volatile int a, b; // both zero
```

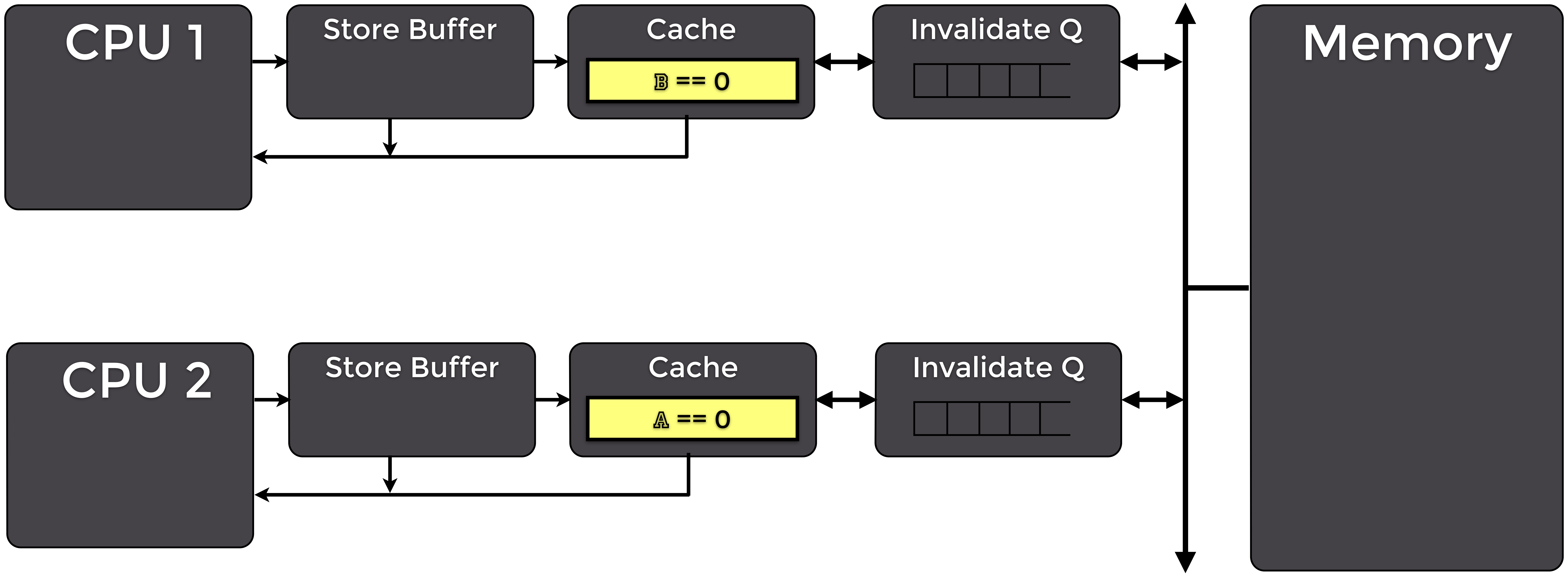
CPU 1

```
void m1() {
   A = 1;
   B = 1;
}
```

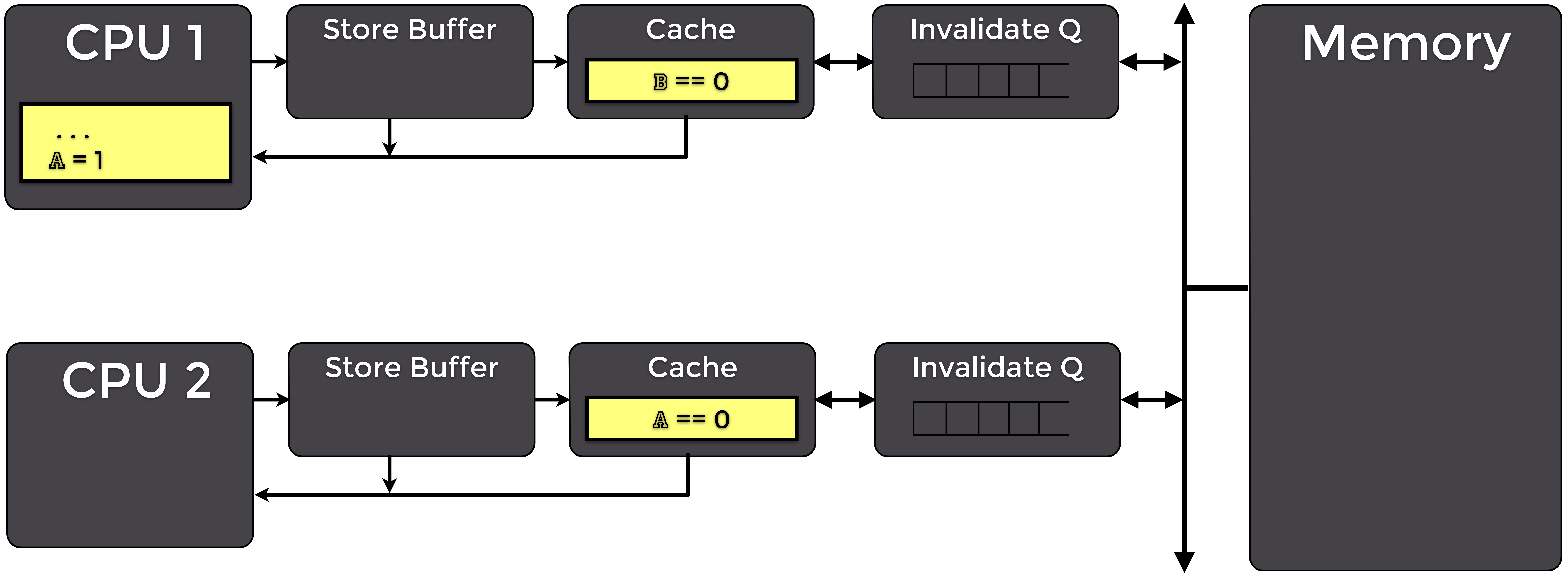
CPU 2

```
void m2() {
   while (B == 0)
          ;
   assert(A == 1);
}
```

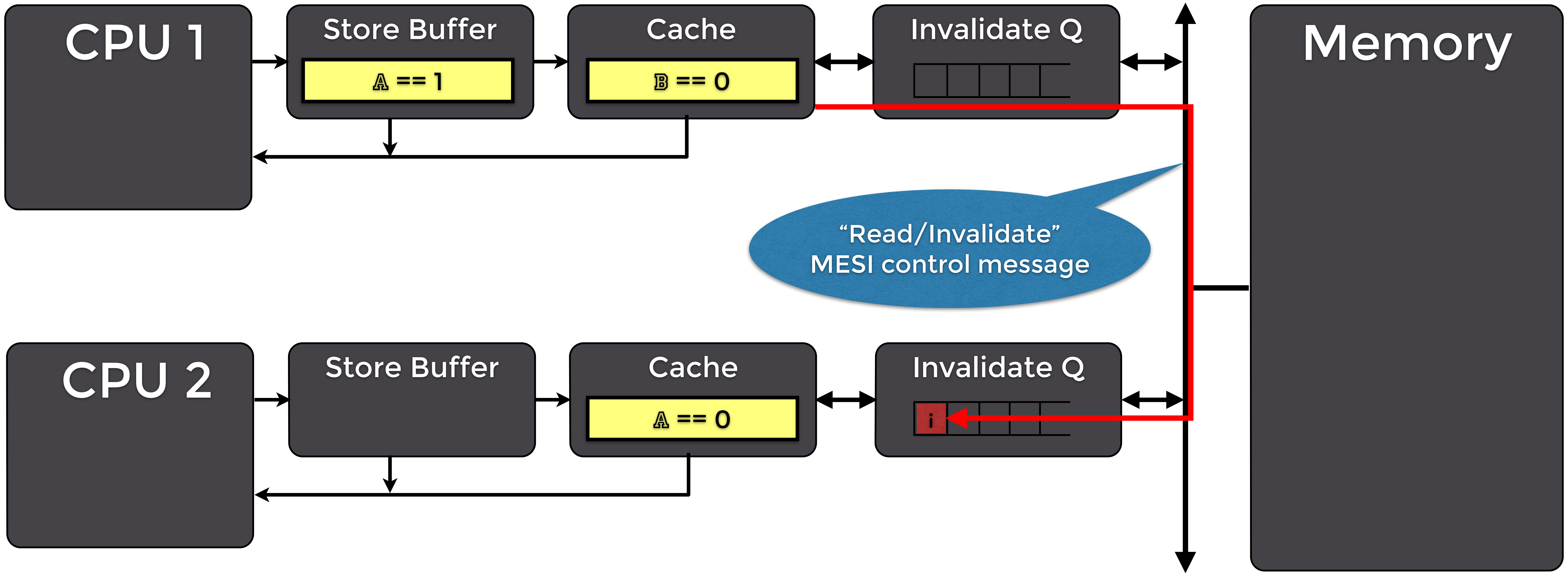
MESI Example 3-2



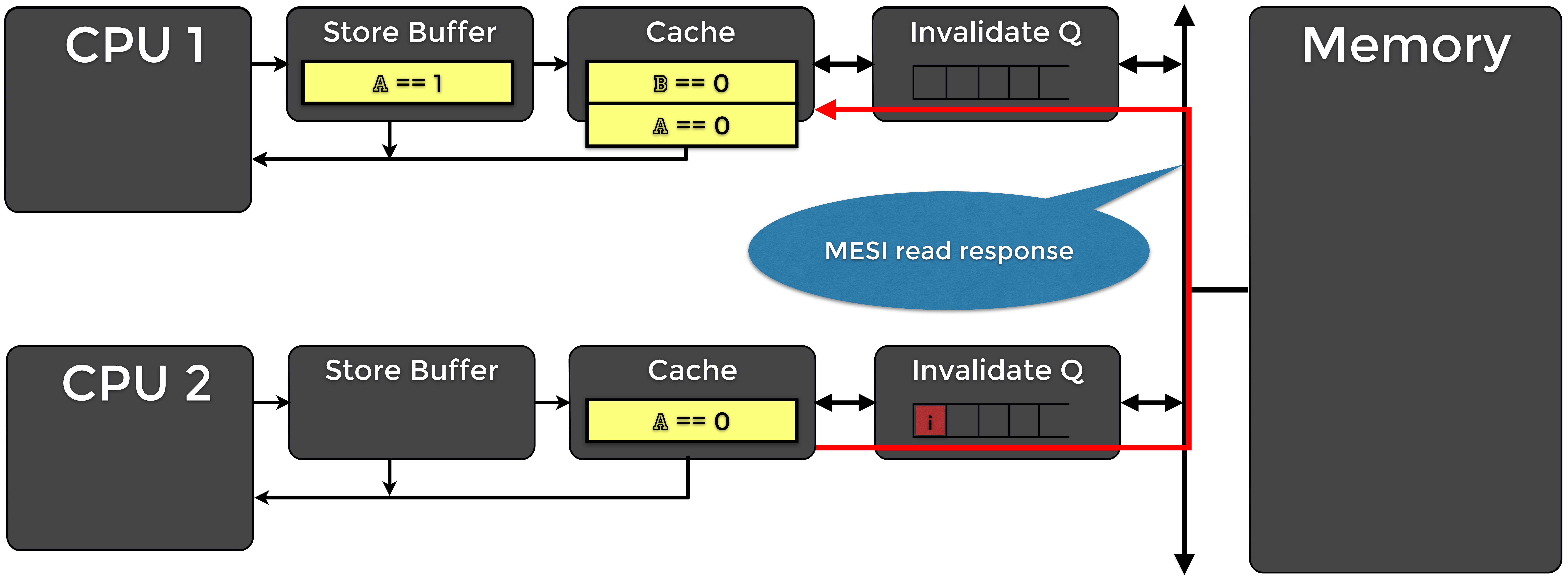
MESI Example 3-3



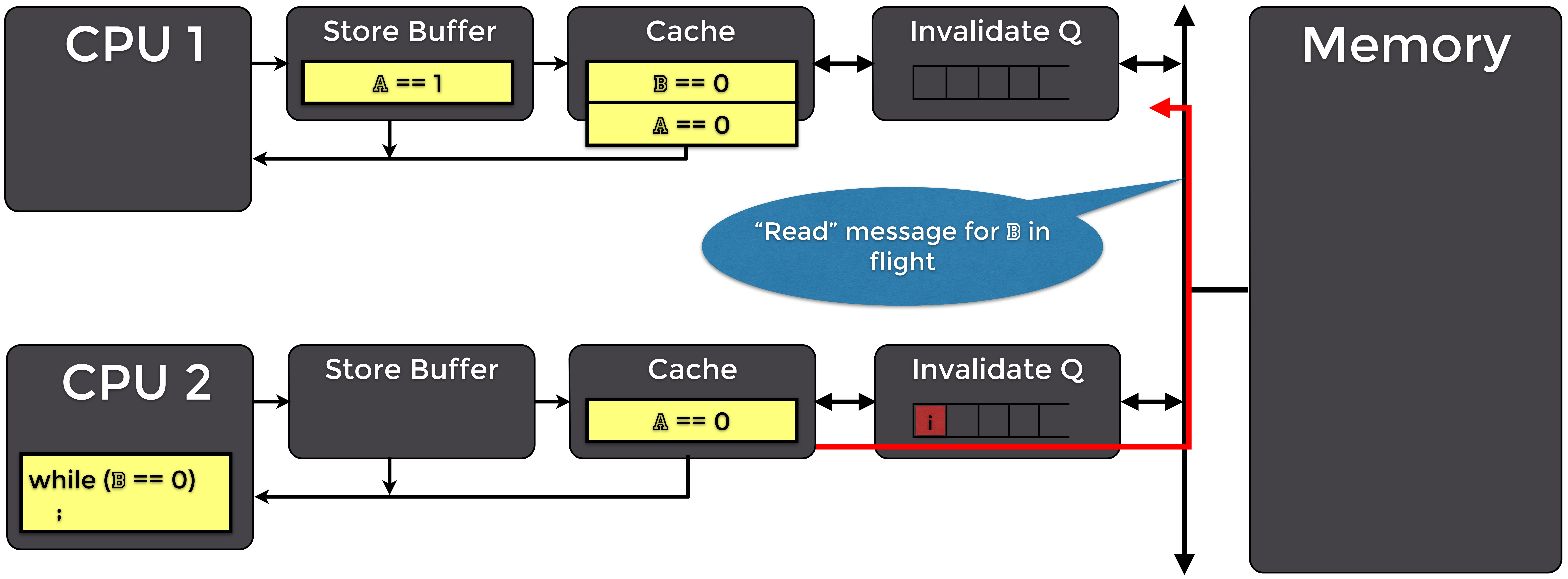
MESI Example 3-4



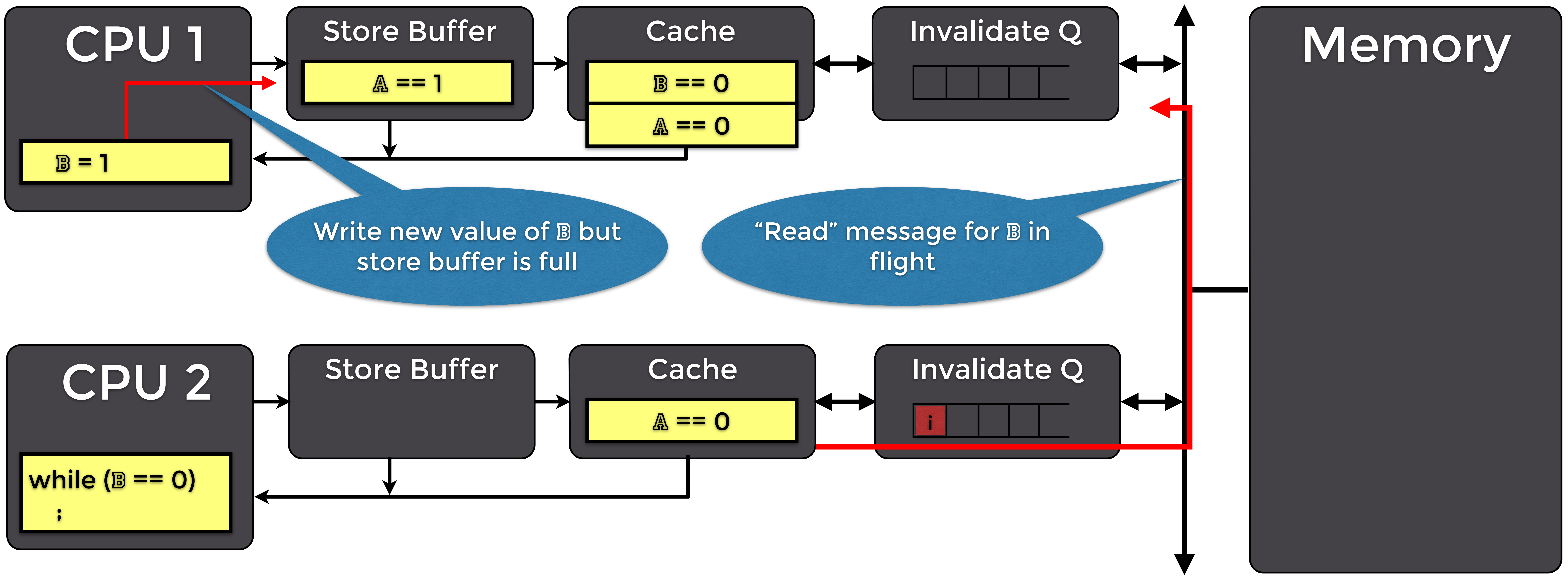
MESI Example 3-5



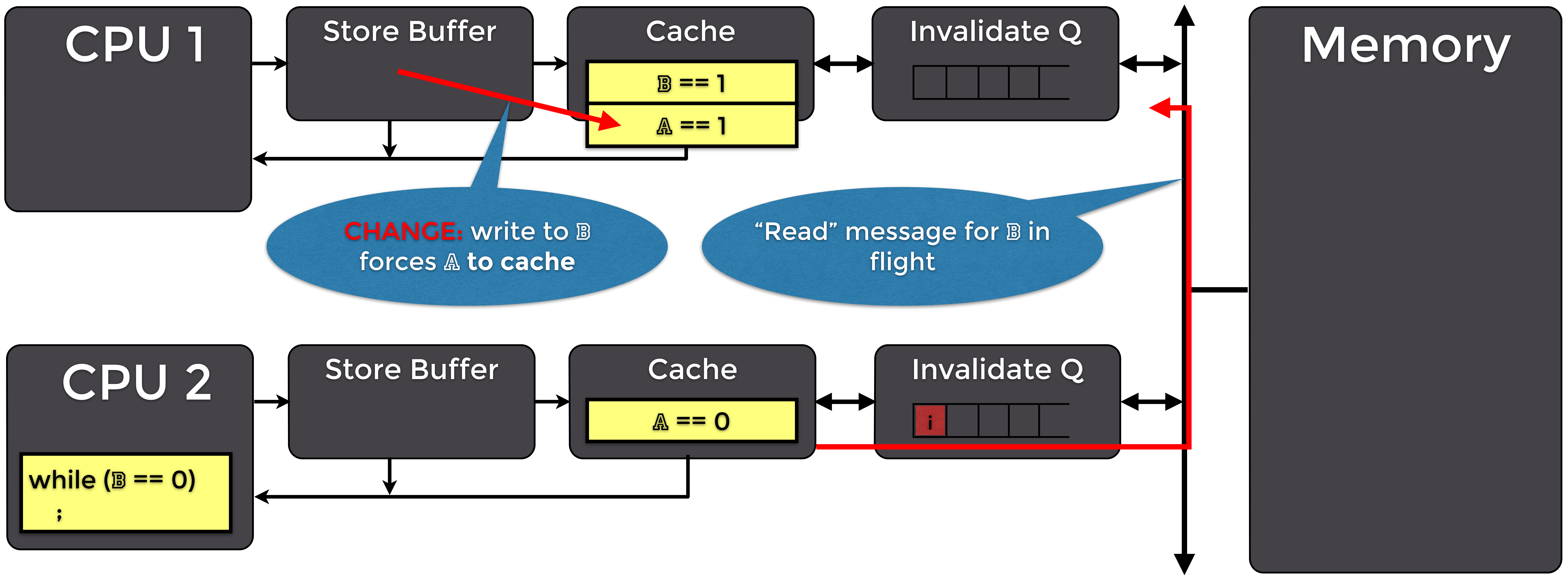
MESI Example 3-6



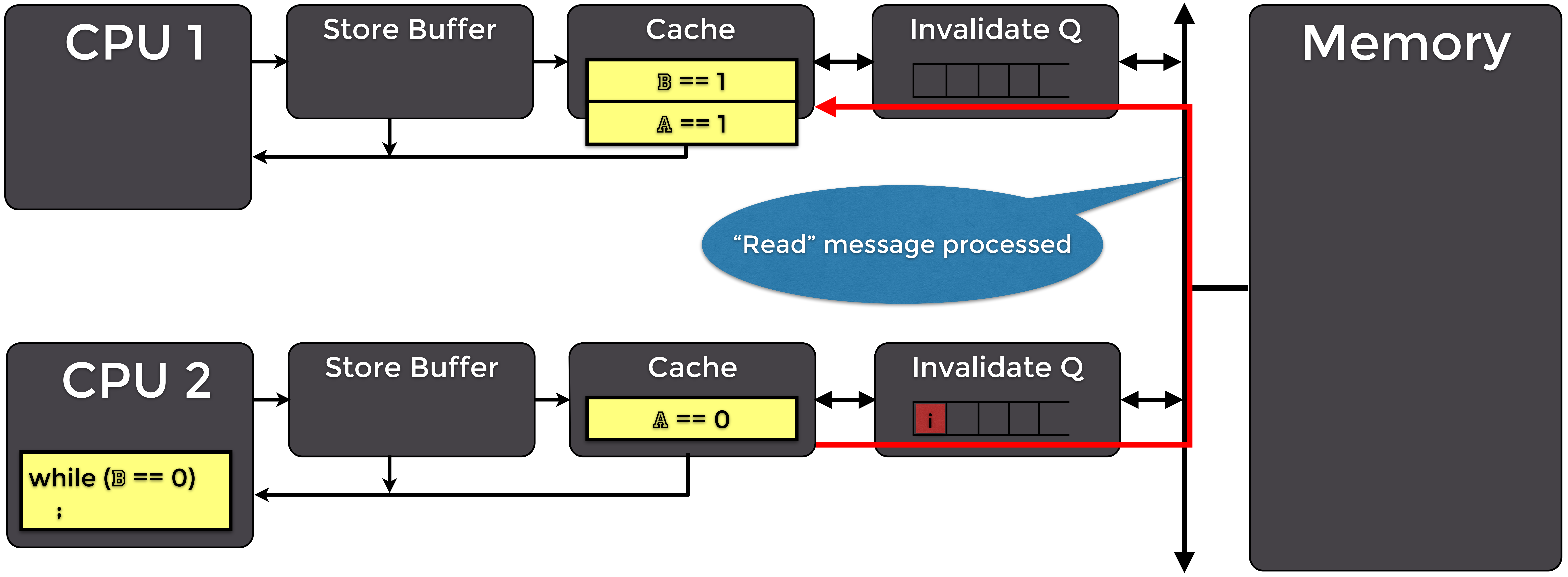
MESI Example 3-7



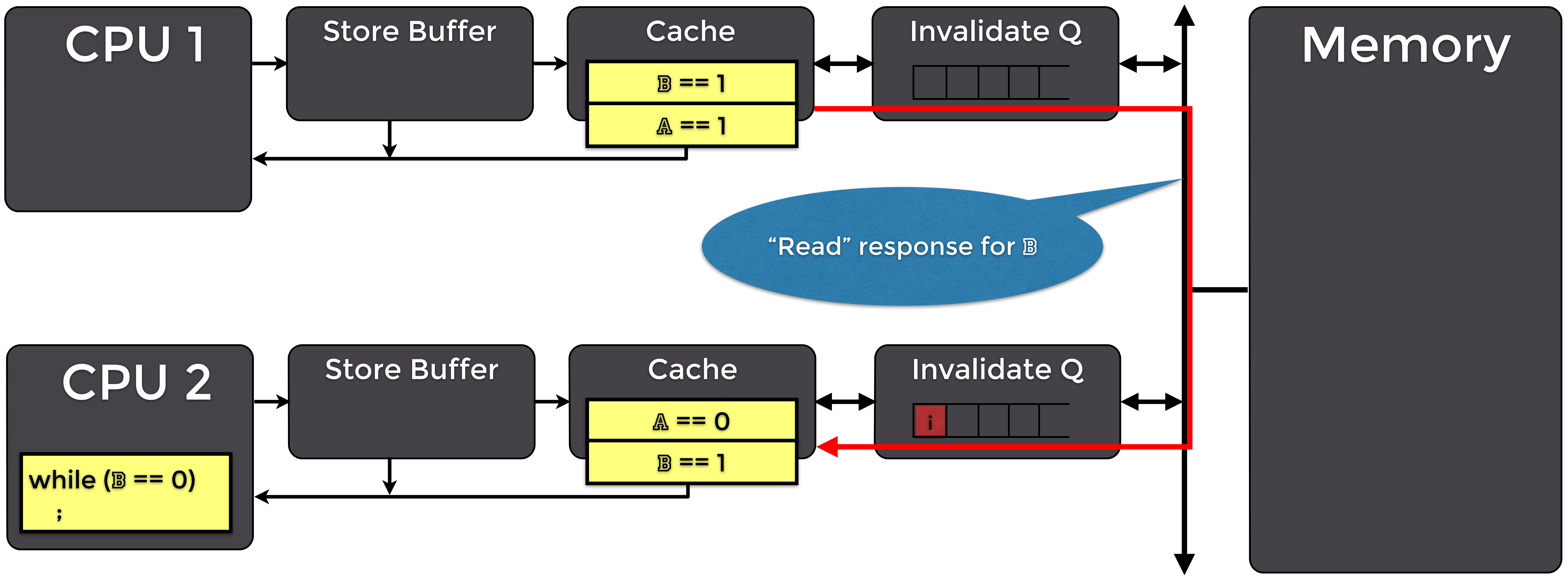
MESI Example 3-8



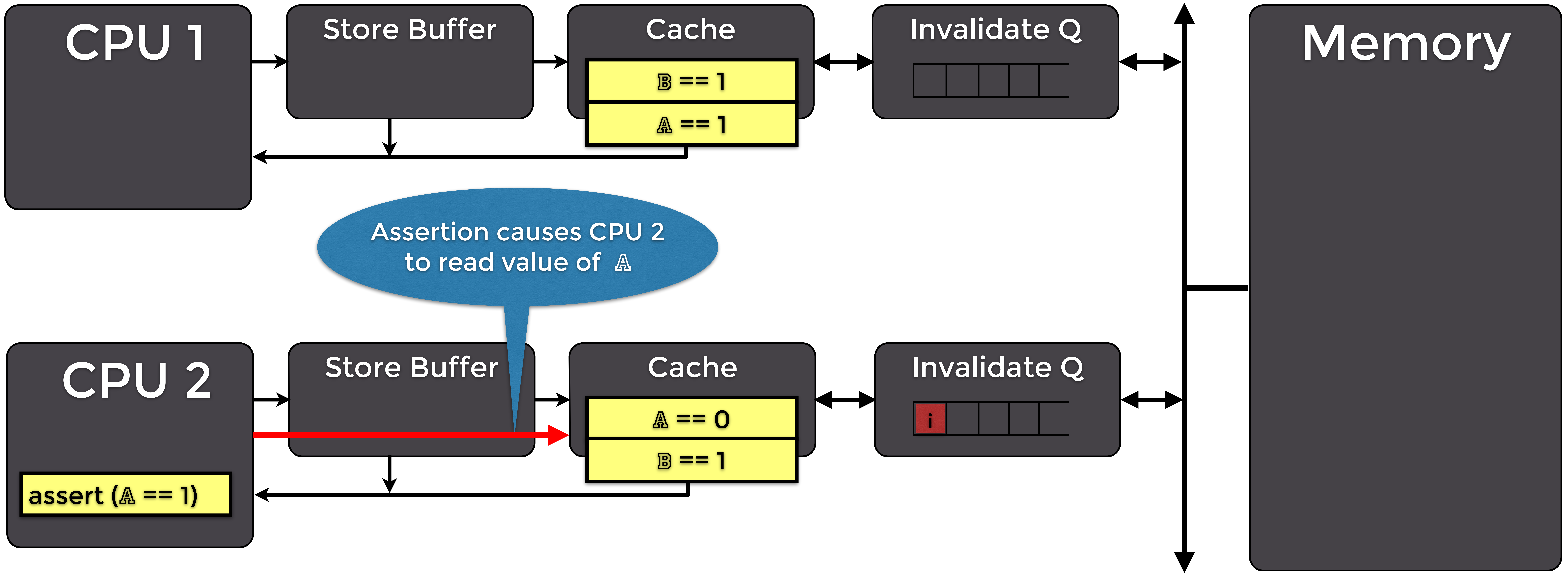
MESI Example 3-9



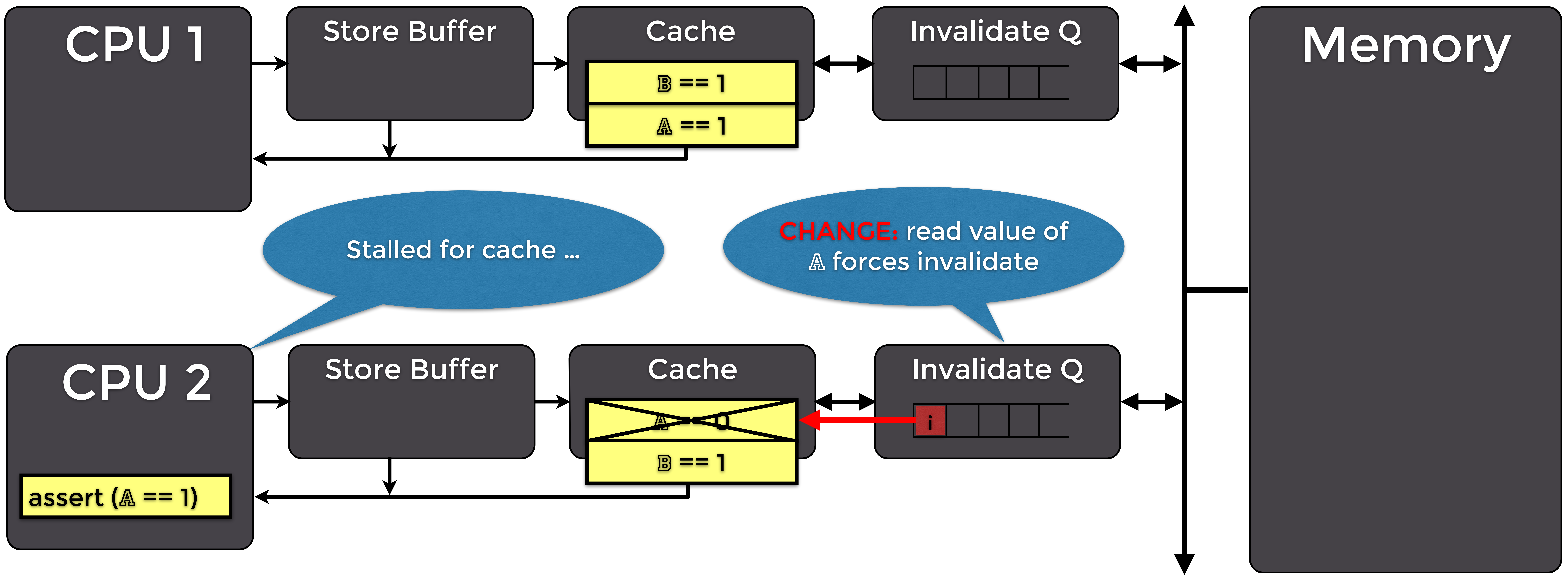
MESI Example 3-10



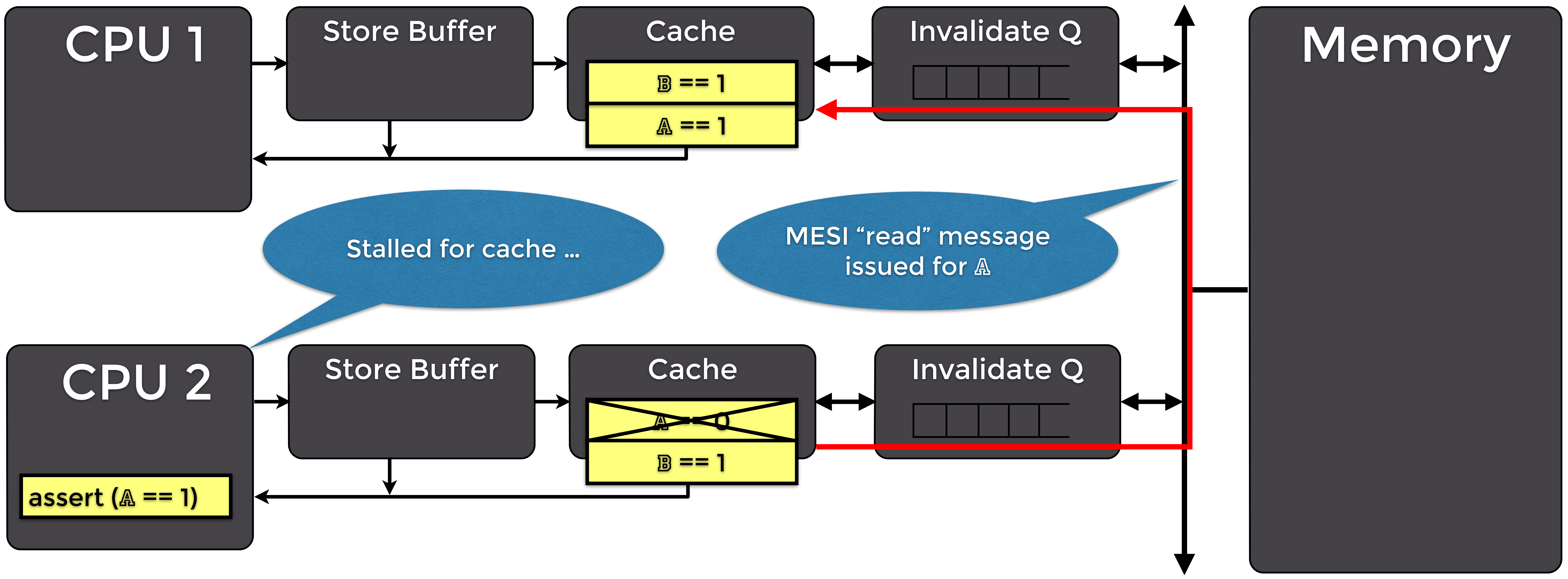
MESI Example 3-11



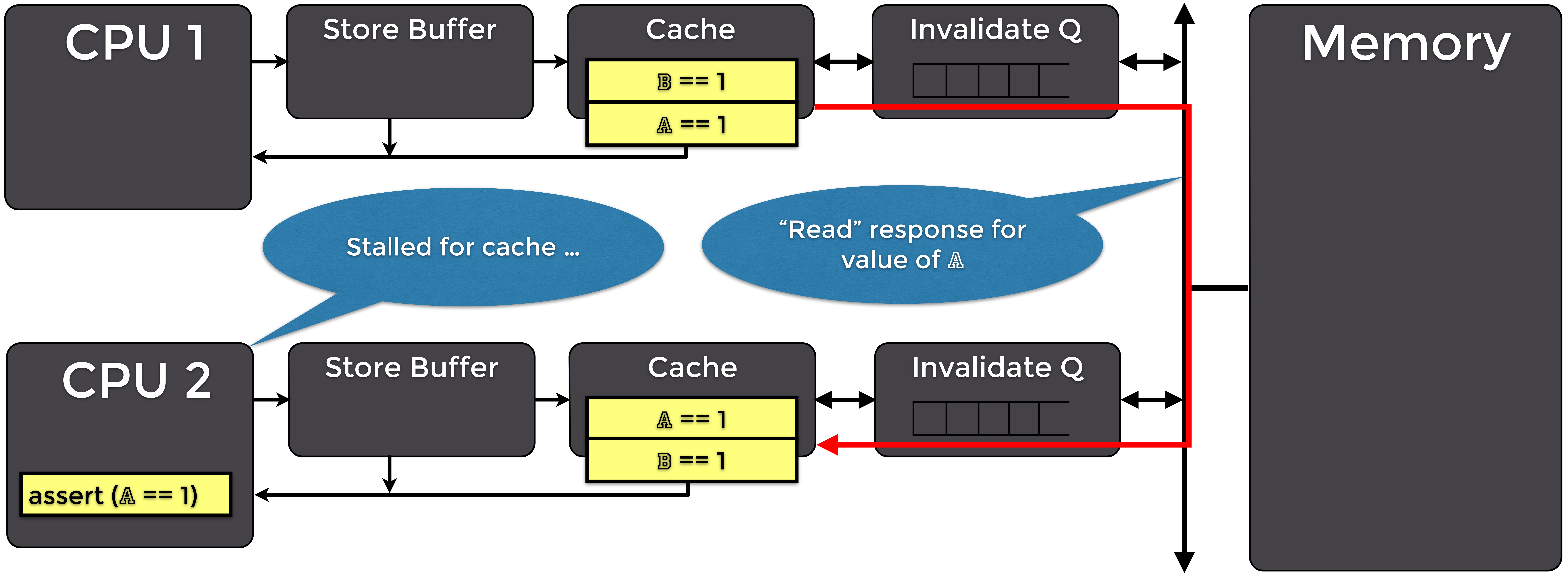
MESI Example 3-12



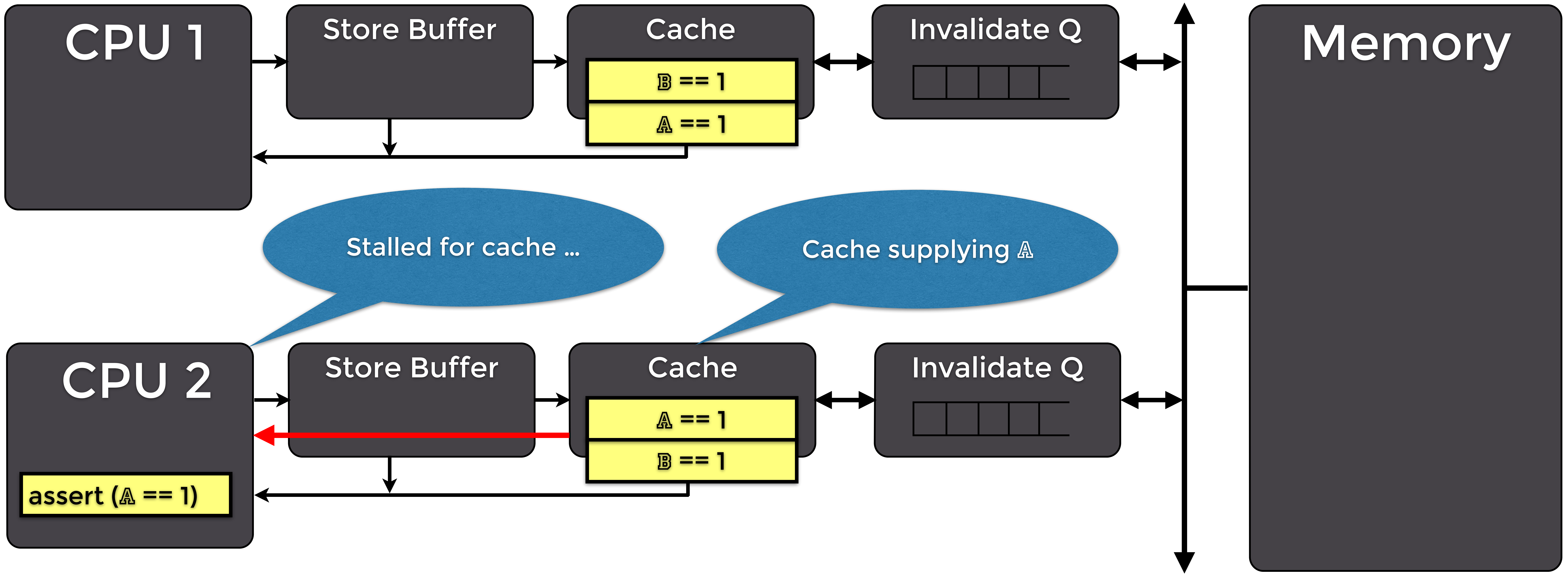
MESI Example 3-13



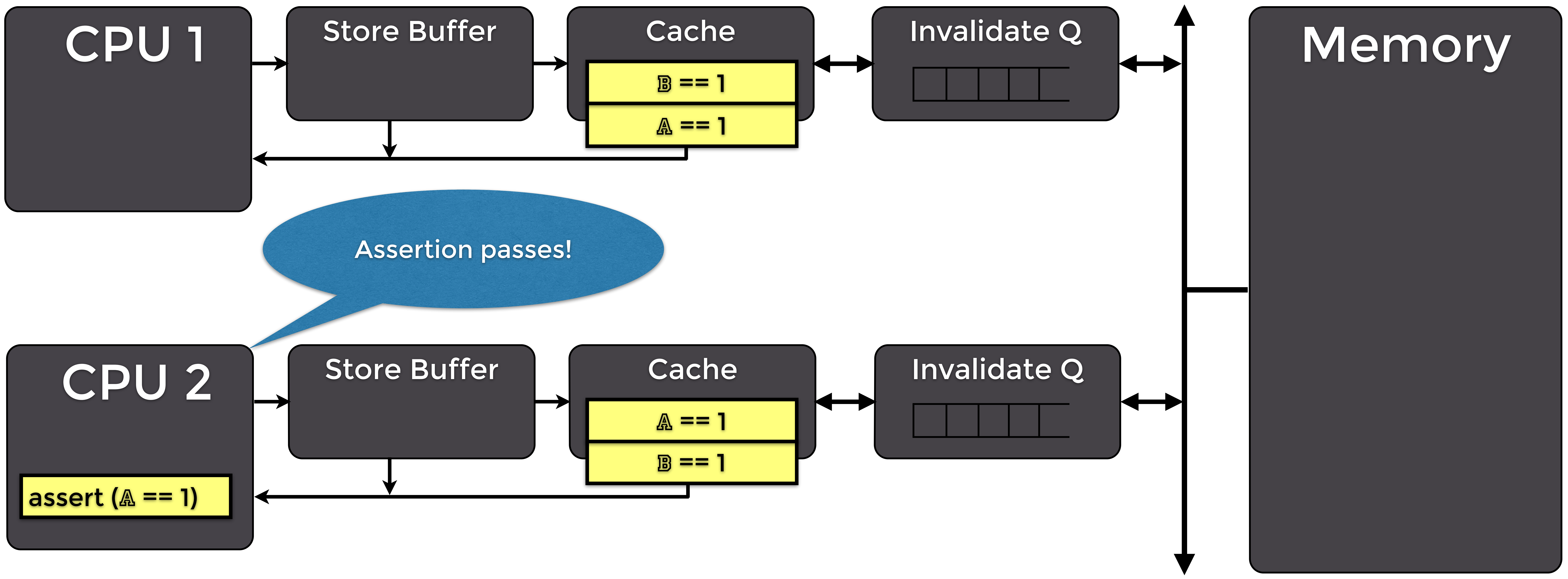
MESI Example 3-14



MESI Example 3-15



MESI Example 3-16



Where Are We?

- Volatile is one way to express happens-before relationships
- Prevents reordering in the compilers
- At runtime, JIT generates architecture-specific opcodes to
  - prevent memory op reordering in hardware
  - prevent deferred processing in hardware

Generalizing on the JMM…

- Go
  - New language from Google
  - Memory model expressed in terms of “happens-before” as in JMM.
- Akka
  - Async framework for Java, Scala, …
  - Spec makes reference to JMM

Other Languages

- C
  - Explicit (compiler directives, asms)
- C++
  - Interesting memory model in C++ 2011
- Objective-C
  - Also low level, language-specific features

And More Languages

- C#
  - Similar to Java
- Rust
  - Concurrent task abstraction (a lá Occam?); No shared memory in “safe” code
- Dalvik (Android virtual machine)
  - Historically broken (Stackoverflow post)

Explicit Control in C

- Compiler directives/annotations/asms to prevent aggressive compiler reordering
- Linux kernel: macros expand to explicit memory barrier instructions

```
void m1(void) {
  stmt-1;
  smp_mb();
  stmt-2;
}
```

Summary

- These issues affect all languages that support programming with threads
- Java community was ahead of the curve in addressing them
- Awareness wins - you may not program against the JMM, but understanding it is powerful.
- Keep learning - avoid “DIY” and use the highest level tools you can.

References

http://bitly.com/bundles/pdxjjb/2

Contains all the “bit.ly” links
from this presentation

THANK YOU

- Java Agent team and so many others at New Relic for attending my practice talks and providing feedback…
- And everyone who has attended one version or another of this talk.

Q&A

Followed By

Lunch