Mythen und Legenden

Zum shared_ptr als formaler Parameter

Agenda

- Überblick Smart Pointer
- Atomkraft
- MESI Protokoll
- Barrieren
- Fazit

Smart pointer

- Was ist ein smart pointer
  - Shared pointer → std::shared_ptr
  - Unique pointer → std::unique_ptr
  - Auto pointer → std::auto_ptr
  - Scoped Pointer → boost::scoped_ptr
  - RAII
  - Automatisches Löschen des referenzierten Objectes beim verlassen des Scopes
  - Sinnvoll im Falle von Exceptions
  - ...
- Alle verschiedenen smart pointer sind Zeiger, intelligente Zeiger, aber dennoch Zeiger

Der Star des Abends

std::shared_ptr

Übergabe eines shared_ptr an eine Funktion/Methode

- Ein Zeiger wird im Allgemeine wie folgt übergeben
  - void my_function(Obj* obj)

- Ein shard_ptr verhält sich wie ein Zeiger, daher wird er per value übergeben, wie es guter C++11 Stil ist
  - void my_function(std::shared_ptr<Obj> obj)

Was ist ein shared_ptr (1)

- Der C++11 Standard sagt
  - The shared_ptr class template stores a pointer, usually obtained via new. shared_ptr implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. A shared_ptr object is empty if it does not own a pointer.
- Beispiel:

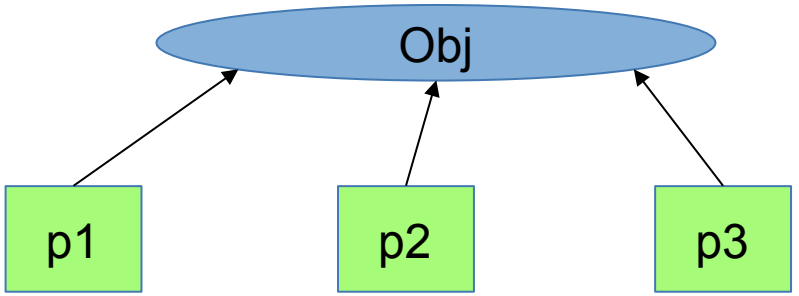
```
std::shared_ptr<Obj> p1(new Obj);

std::shared_ptr<Obj> p2 = p1;

std::shared_ptr<Obj> p3(p2);
```

- Damit zeigen p1, p2 und p3 auf das selbe Object

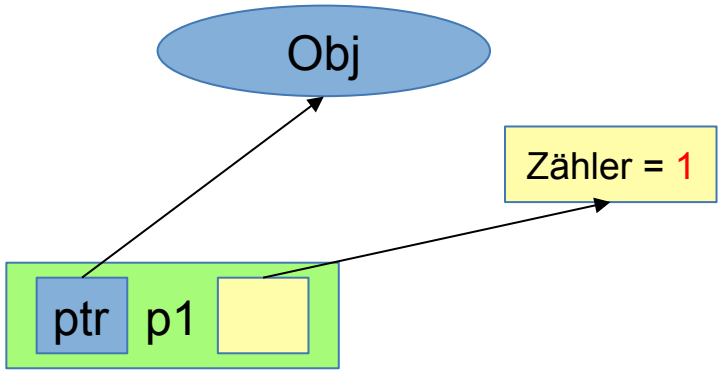
Was ist ein shared_ptr (2)



- Das Object (Obj) lebt so lange bis der letzte shared_ptr der auf das Object (Obj) zeigt zerstört wird
- Also
  - delete p1 → Obj exisitiert noch
  - delete p3 → Obj exisitiert weiterhin noch
  - delete p2 → Nun wird auch Obj gelöscht

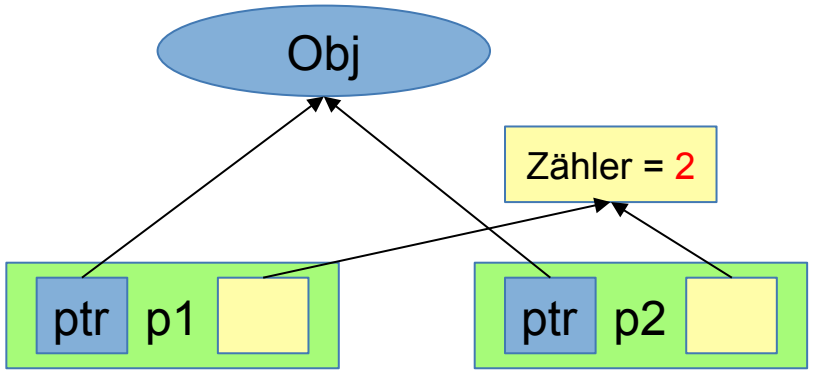
Innenansicht eines std::shared_ptr

- Besteht prinzipiell aus zwei Elementen
  1) Zeiger auf das eigentliche Objekt
  2) Einem (gemeinsamen) Zähler



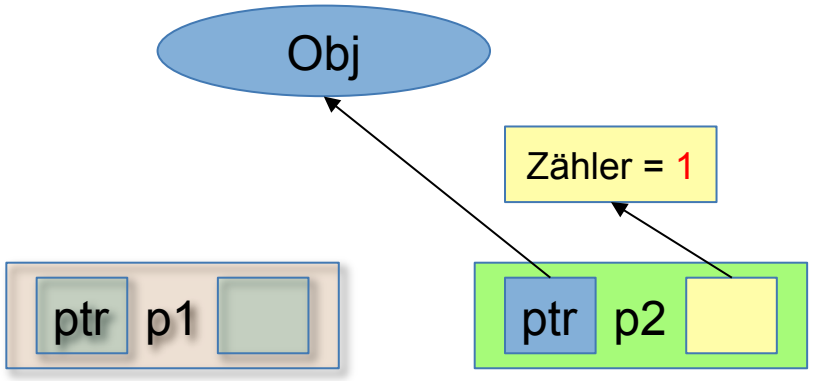
Innenansicht eines std::shared_ptr

- Besteht prinzipiell aus zwei Elementen
  1) Zeiger auf das eigentliche Objekt
  2) Einem (gemeinsamen) Zähler



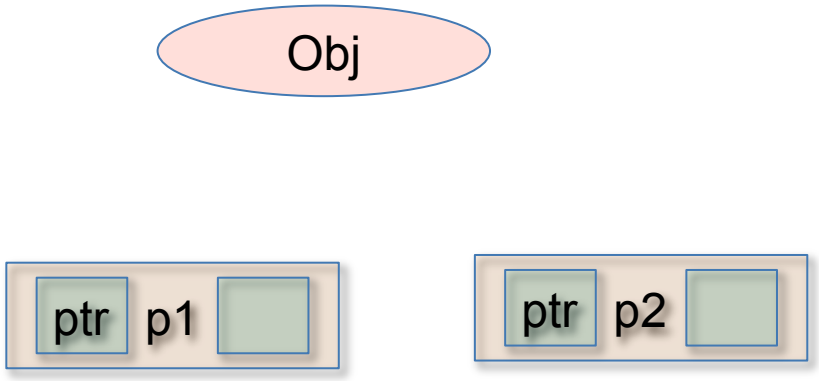
Innenansicht eines std::shared_ptr

- Besteht prinzipiell aus zwei Elementen
  1) Zeiger auf das eigentliche Objekt
  2) Einem (gemeinsamen) Zähler



Innenansicht eines std::shared_ptr

- Besteht prinzipiell aus zwei Elementen
  1) Zeiger auf das eigentliche Objekt
  2) Einem (gemeinsamen) Zähler



Übergabe per Value

- Bei der Übergabe eines std::shared_ptr in eine Funktion oder Methode wird eine temporäre Kopie erzeugt. Innerhalb der Funktion bzw. Methode wird mit dem std::shared_ptr gearbeitet, evtl. weitergereicht und am Ende der Funktion/Methode wieder zerstört.
- Beispiel

```
void my_function(std::shared_ptr<Obj> ptr)
{
    …
    ptr->do_something();
    …
}
```

Temporäres std::shared_ptr Object wird erzeugt

Temporäres std::shared_ptr Object wird zerstört

Innenansicht eines std::shared_ptr (2)

- Bei einer Kopie wird
  - Der Zeiger kopiert und der (gemeinsame) Zähler erhöht
- Wird diese Kopie wieder zerstört
  - Wird der (gemeinsame) Zähler um eins vermindert und falls der Zähler nun Null ist, das Objekt auf das der interne Zeiger zeigt, ebenfalls zerstört
- Problem: Der Zähler muss geschützt werden, weil in einer Multithreaded Umgebung konkurrierend auf den std::shared_ptr manipulierend zugegriffen (kopiert, verschoben, getauscht, gelöscht) werden kann.

Schutz des Zähler eines std::shared_ptr

- Die Veränderung des Zählers kann durch ein Mutex geschützt werden
- Beispiel:

```
void inc()
{
      std::lock_guard<std::mutex> lock(p_mutex);
      ++p_cnt;
}
```

- Besser ist ein std::atomic
  - Sehr vereinfacht: Ein atomic führt eine Operation so durch, dass sie von allen Thread „gesehen“ wird. Es kann also nicht sein, dass ein Thread einen veralteten Wert läd oder zwei Threads gleichzeitig versuchen den Wert zu aktualisieren und nicht „mitbekommen“, dass der andere Thread ihm zuvorgekommen ist → *race condition*

std::atomic

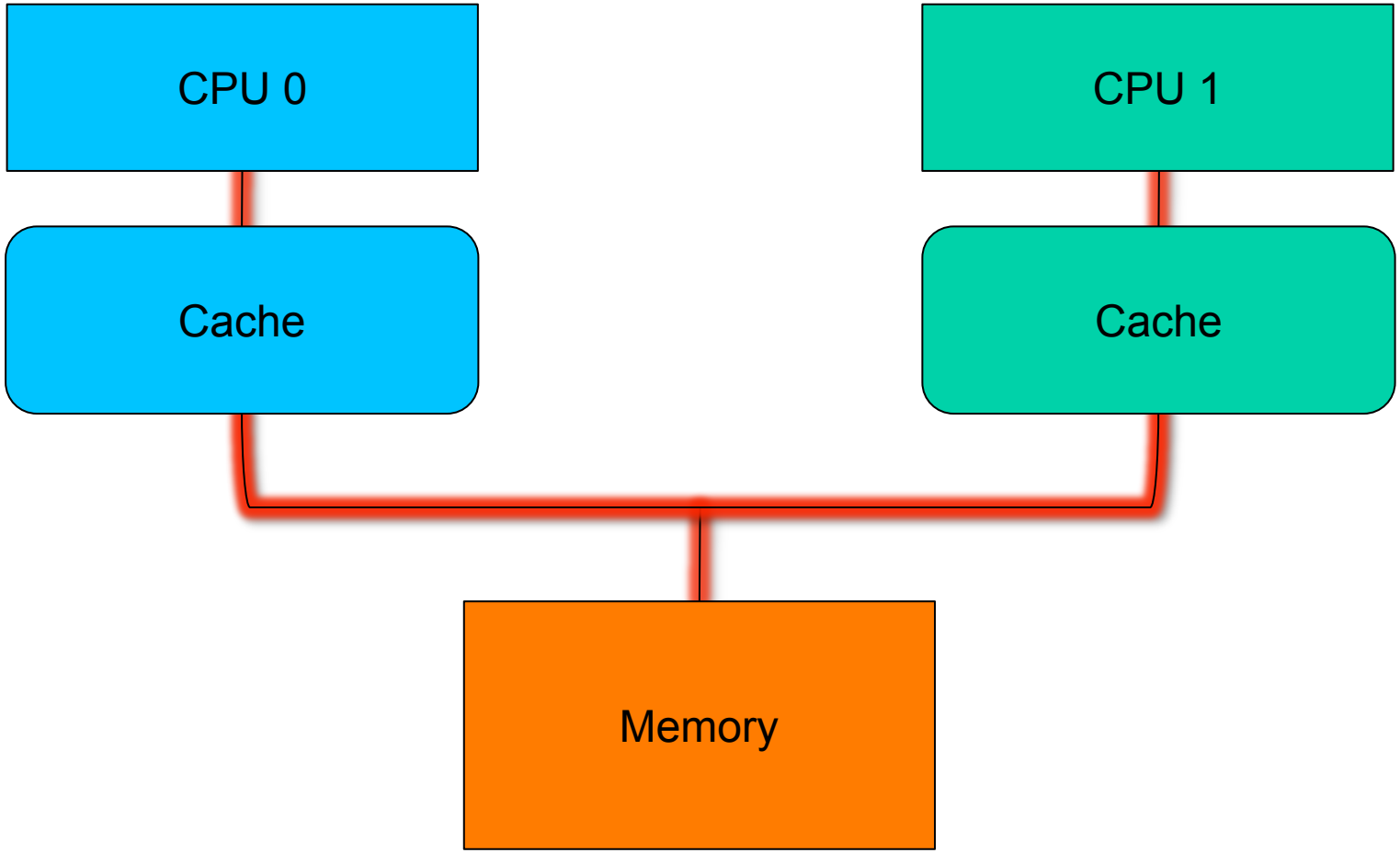
- Auf einem realen Processor wird ein std::atomic mit einem sog. compare-and-swap (CAS) abgebildet (bei Intel IA32-64 LOCK CMPXCHG).
- Bei einer CAS Operation wird ein vorgegebener Wert, mit einem Neuen, nicht unterbrechbar, also atomar, getauscht oder nicht falls sich der Wert in der zwischenzeit geändert hat, somit ihm ein anderer Thread zuvorgekommen ist.
- Code Beispiel zum atomic

```
bool compare_and_swap(int *dest, int& oldval, int newval)
{ // Pseudo-Code!
  if (oldval == *dest) {
      *dest = newval;
      return true;
  } else {
      oldval = *dest;
      return false;
  }
}
```

Atomic! Alles klar?

- Wird ein std::shared_ptr per value übergeben, so wird der atomic zweimal verändert!
- Soweit alles klar?
- Was ist nun aber so schlimm am atomic ?

Symmetric Multiprocessor System



Warum Caches?

- CPU aus dem Jahr 2006 zehn Instructions pro Nanosekunde
- Speicherzugriffe mehr als 2 Größenordnungen langsamer
- Ungleichgewicht in Geschwindigkeit führte zu:
  - Multimegabyte Caches
  - Zugriff auf den Cache in wenigen “Cycles”

Cache-Misses

Cache-Misses

- Cache besteht aus “Cache Lines”
  - Blöcke fester Länge
- Ist benötigtes “Data-Item” nicht im Cache spricht man von “cache-miss”
- Cache-misses führen zu CPU “stalls”
  - CPU kann mehrere hundert Cycles nicht arbeiten

Cache-Coherence Protokolle

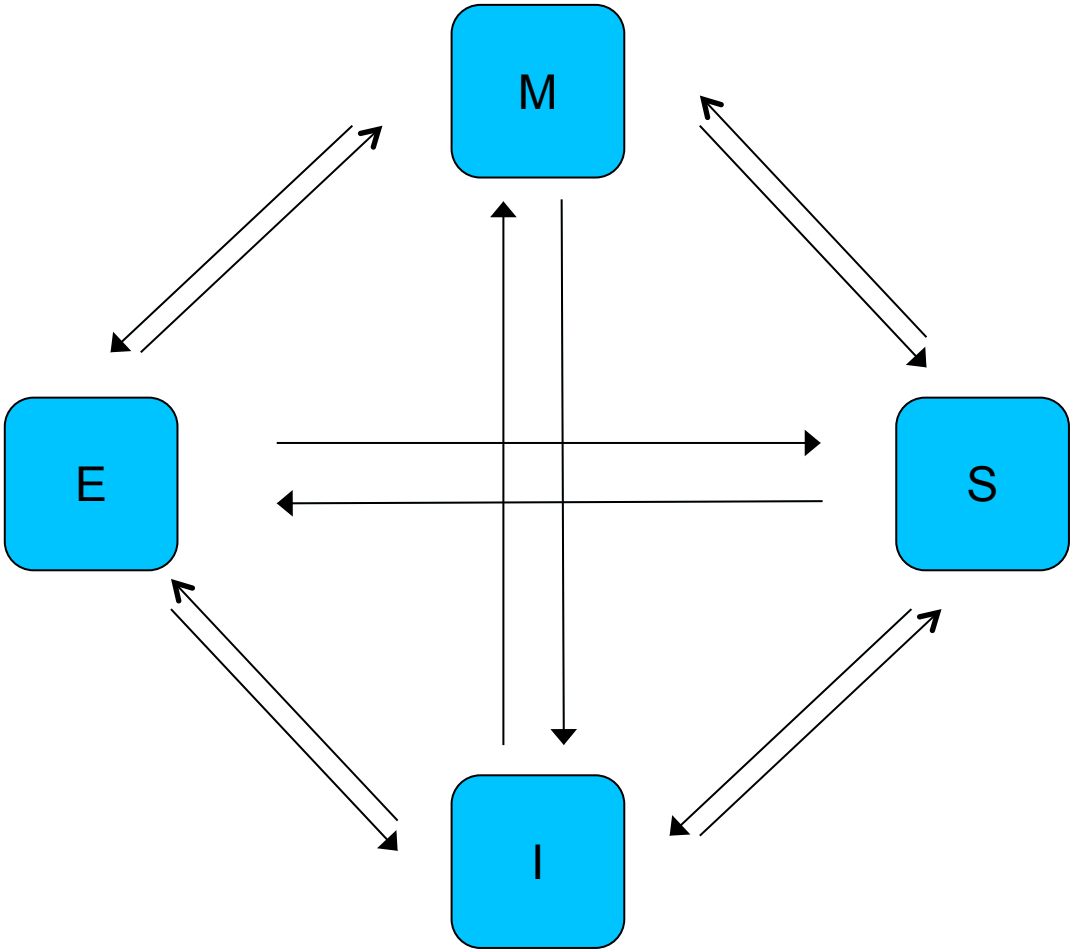
- CCPs managen den Status von Cache-Lines
  - Koordinieren die Bewegung von Cache-Lines durch das System
- Vielzahl von komplexen Protokollen
- Für unsere Betrachtungen hier ist das MESI Protokoll ausreichend

MESI Protokoll

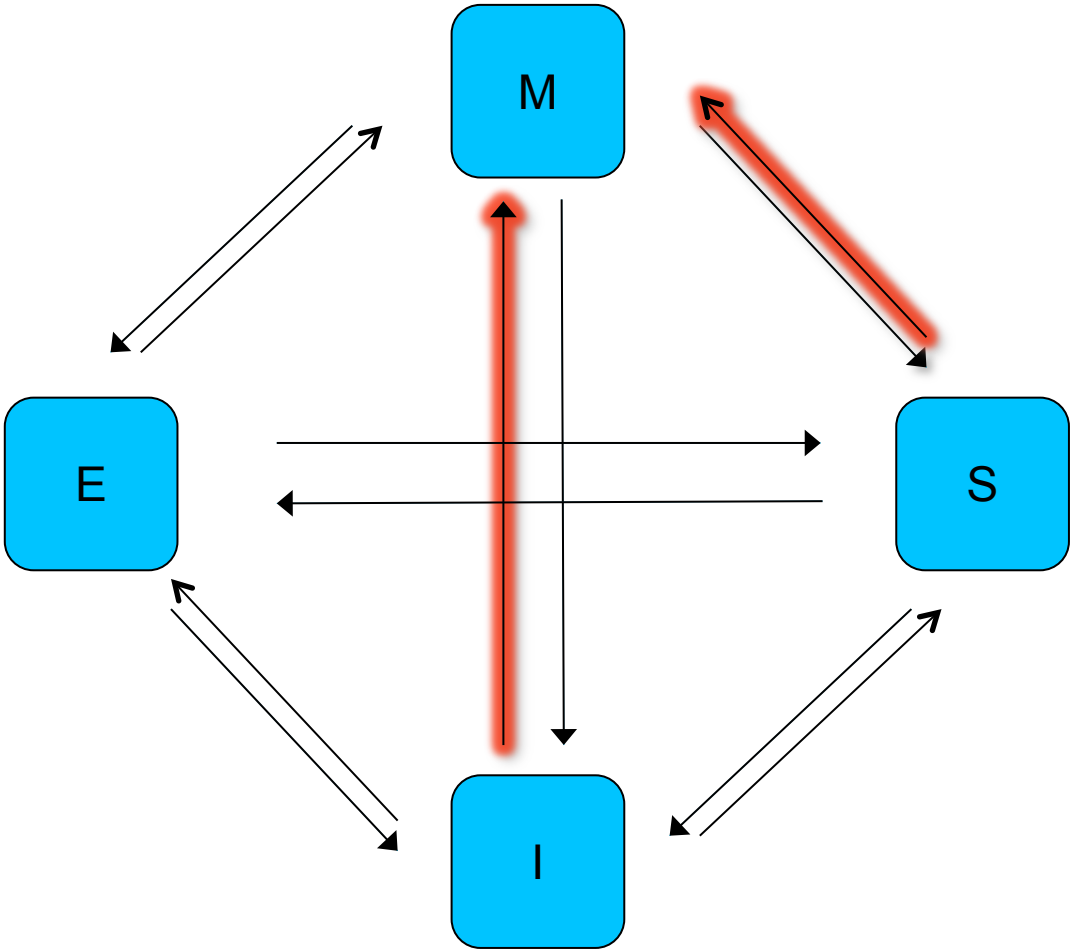
MESI Protokoll besteht aus 4 Zuständen:

1. Modified
2. Exclusive
3. Shared
4. Invalid

MESI State Diagram

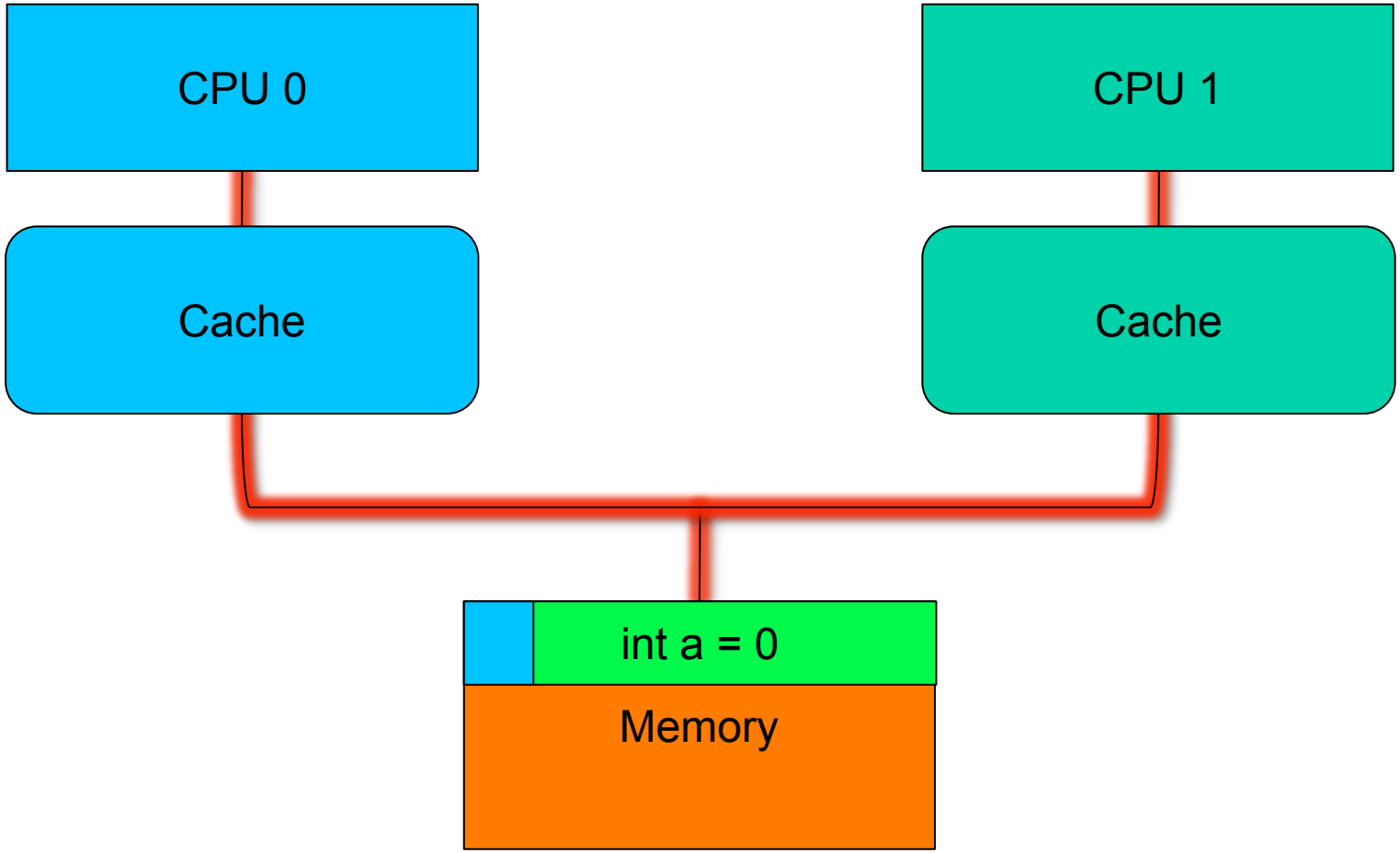


MESI State Diagram

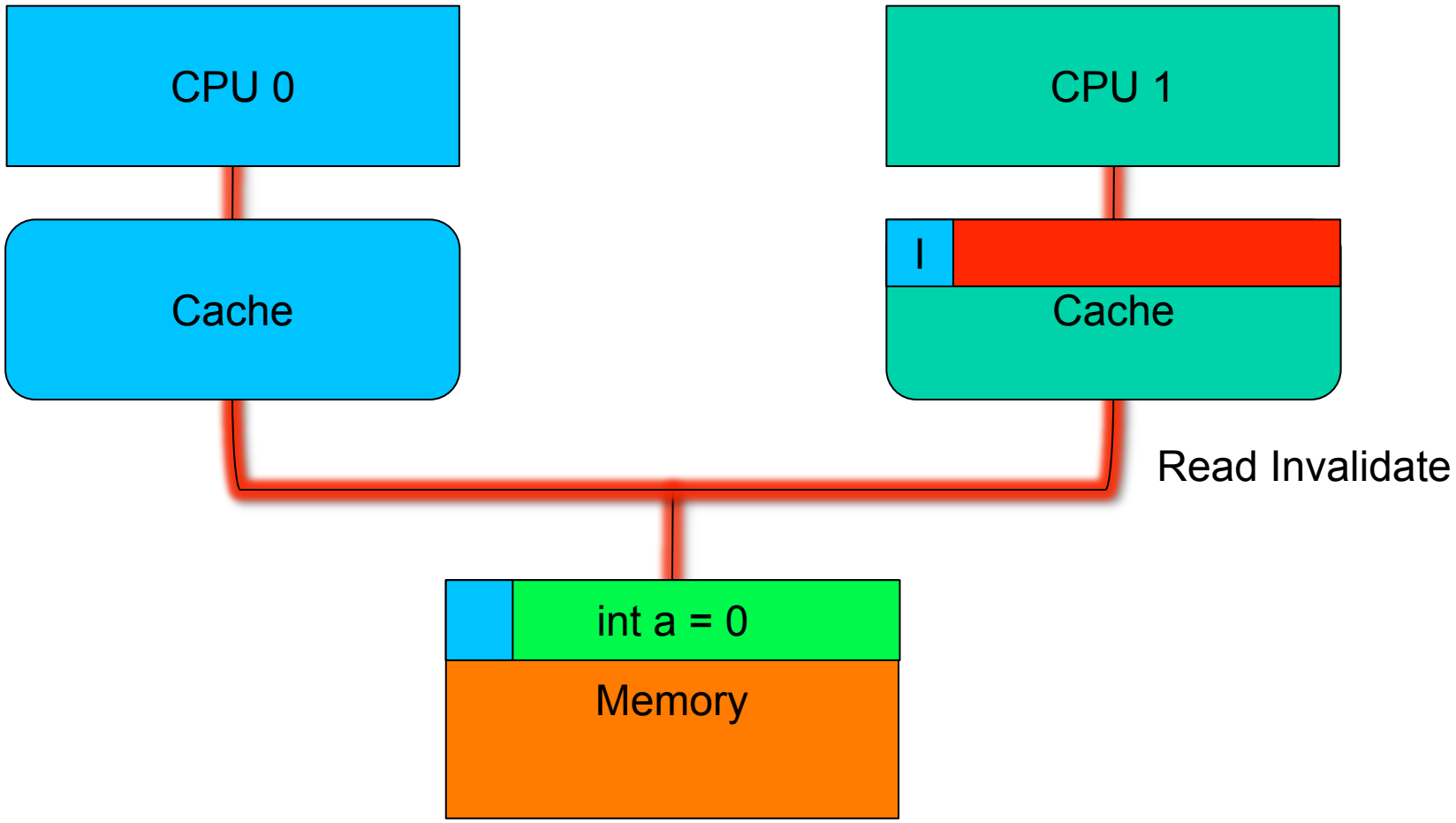


Thorsten Wendt & Michael Wielpütz

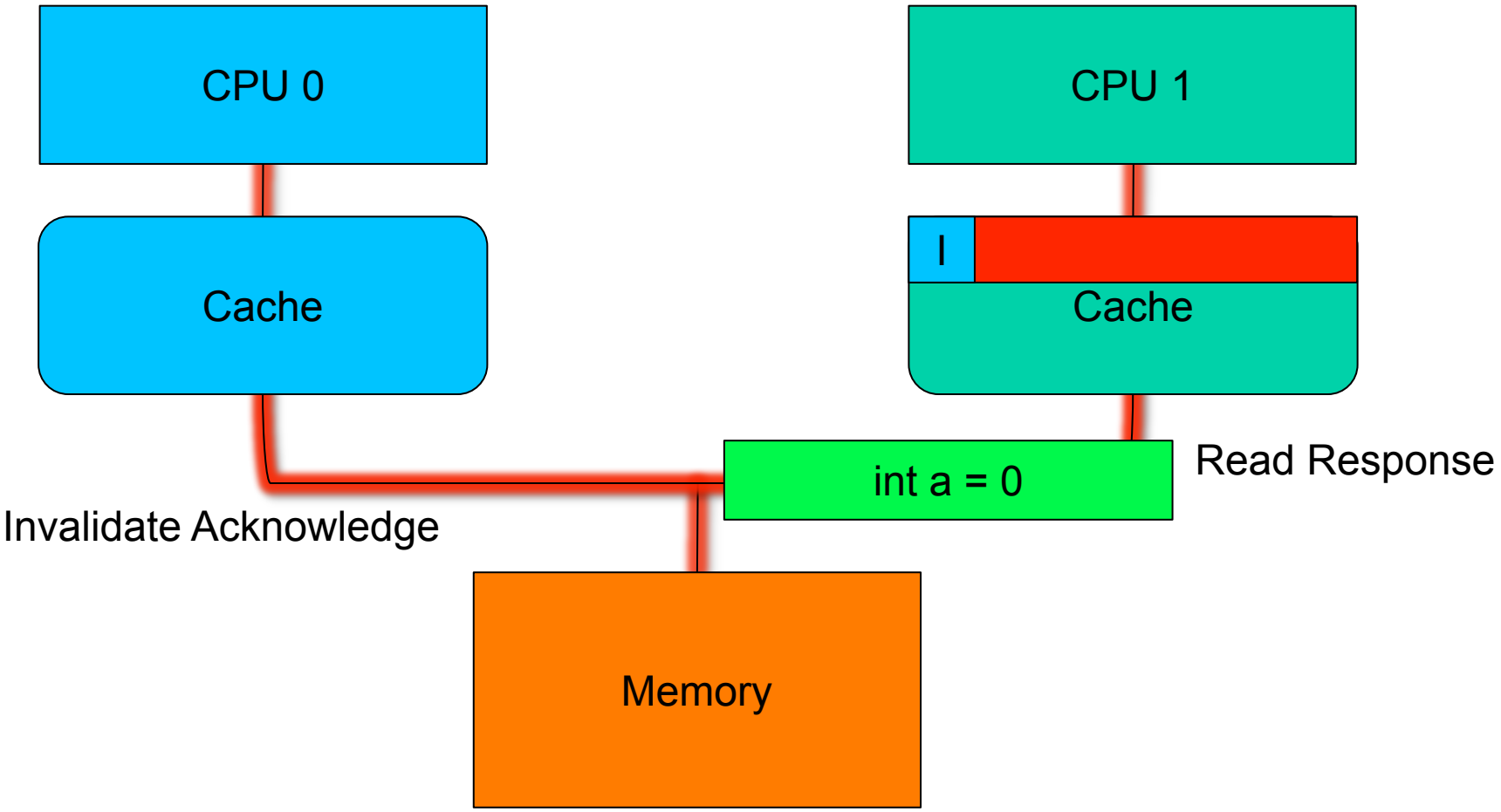
MESI Protokoll



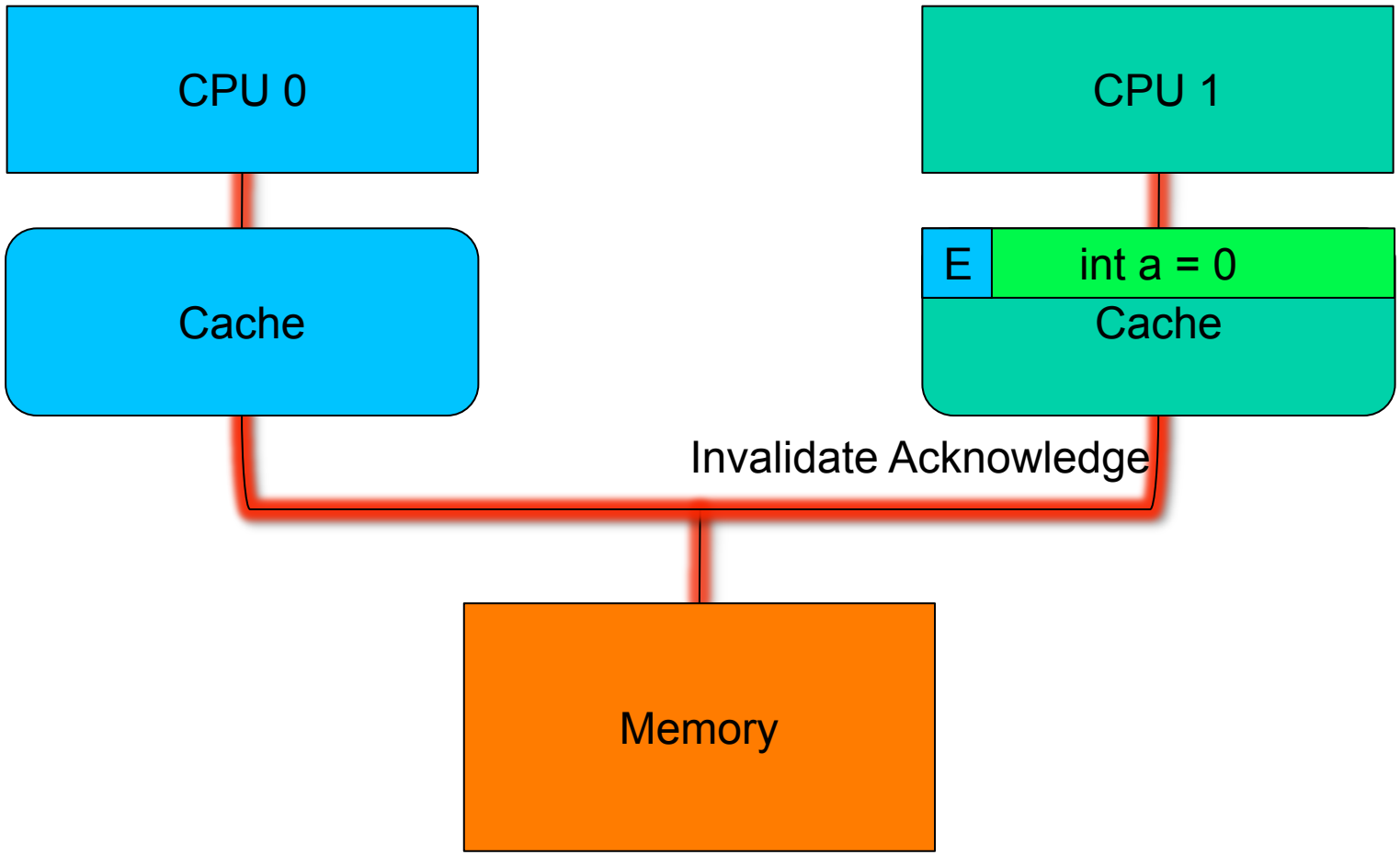
MESI Protokoll



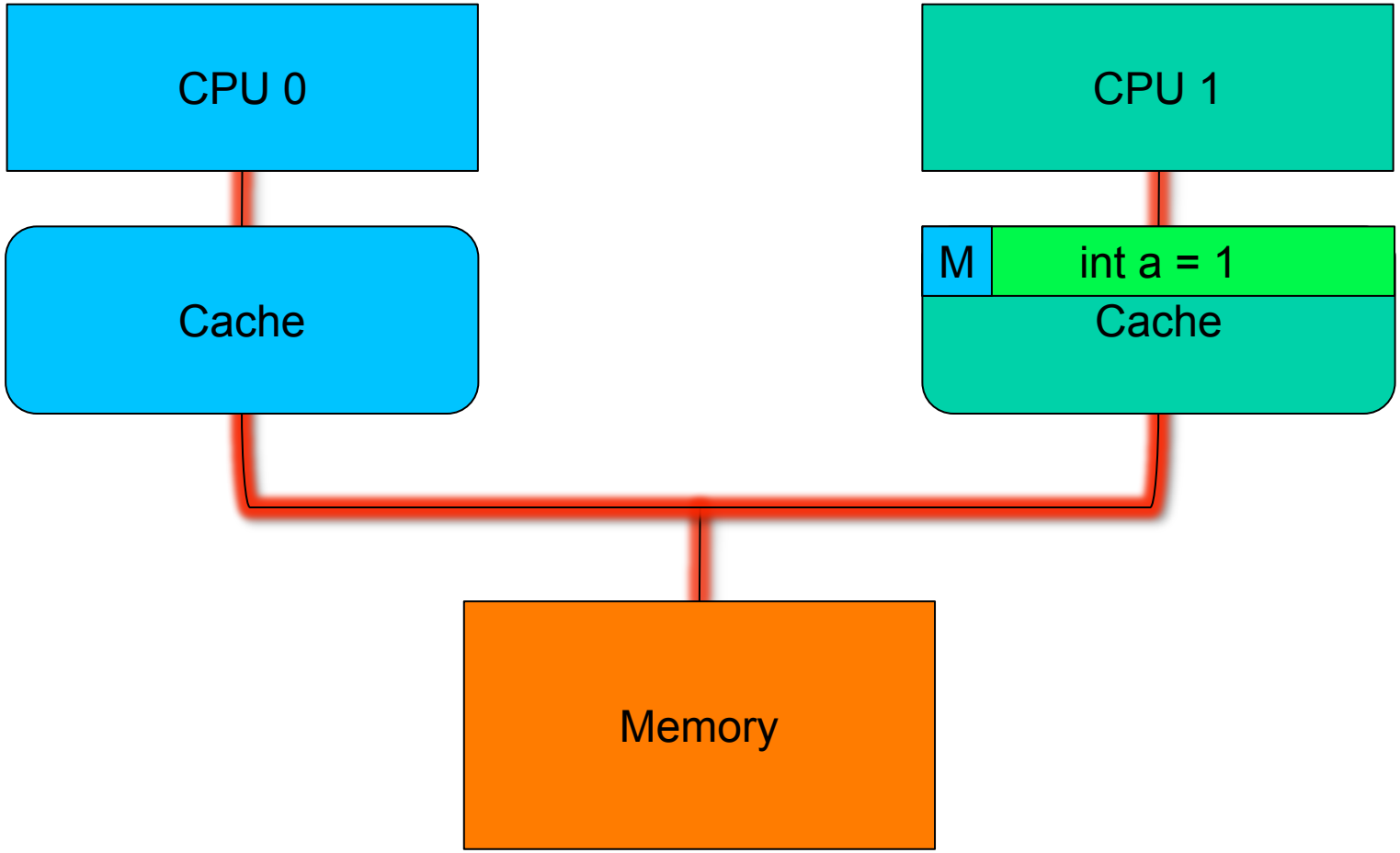
MESI Protokoll



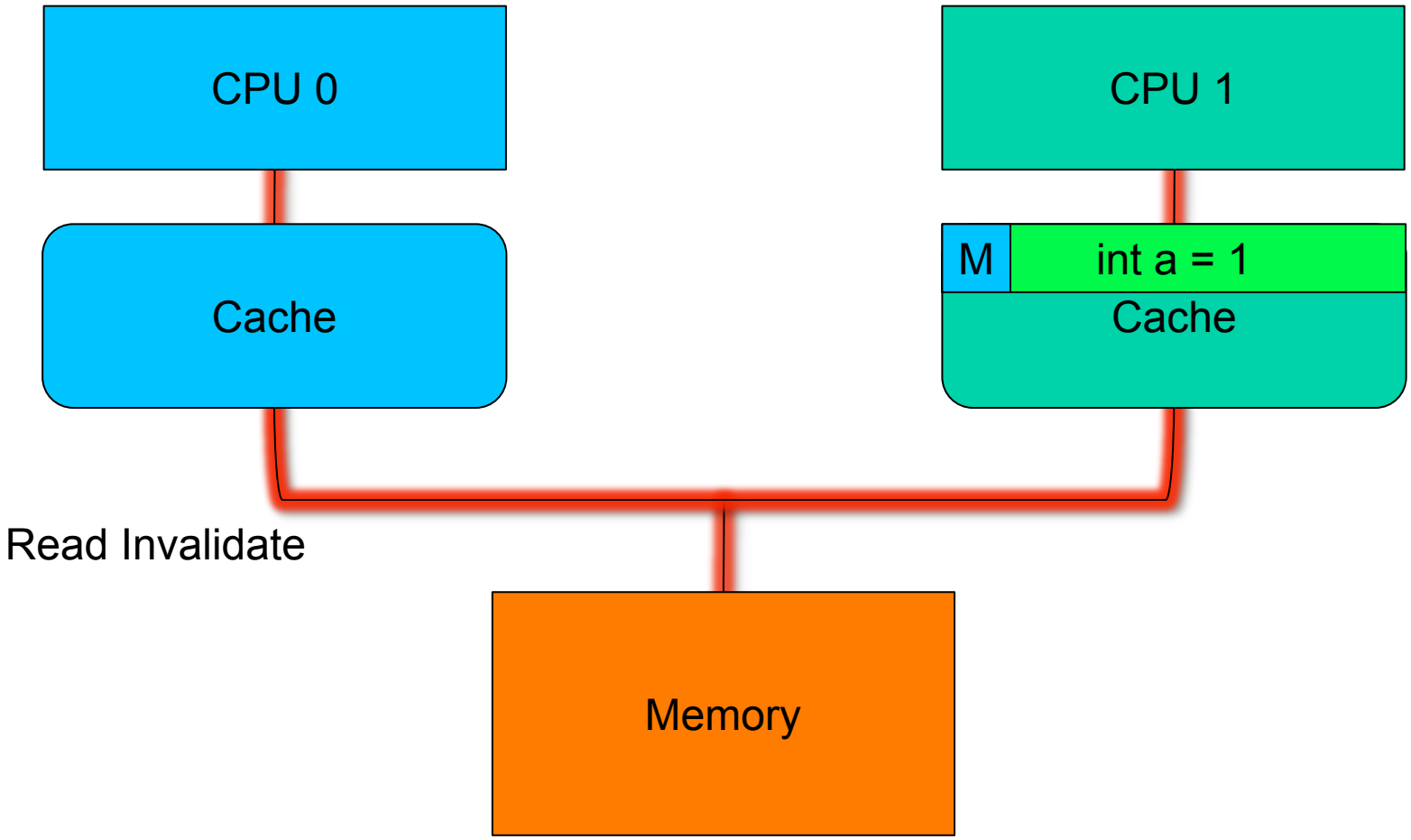
MESI Protokoll



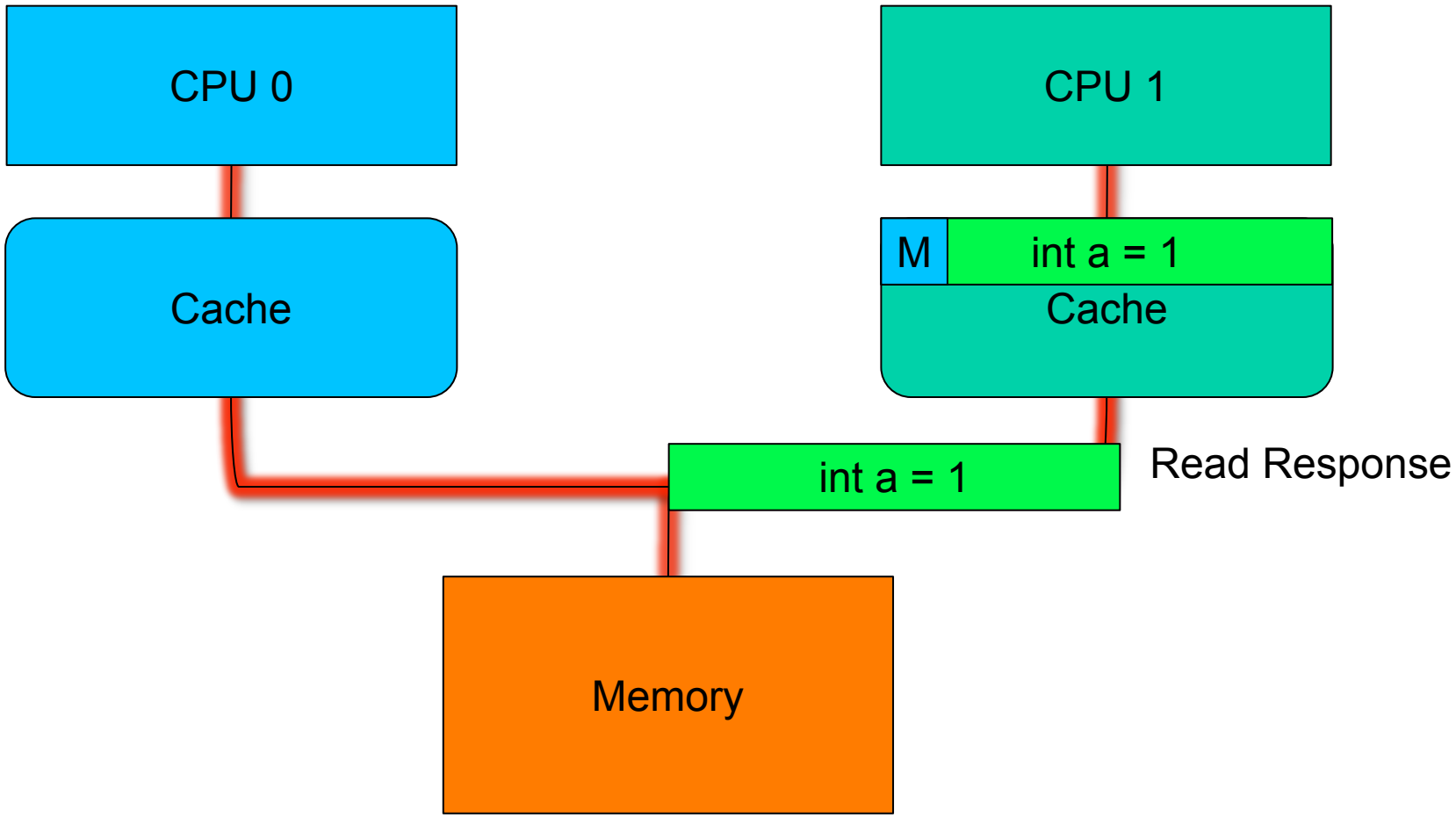
MESI Protokoll



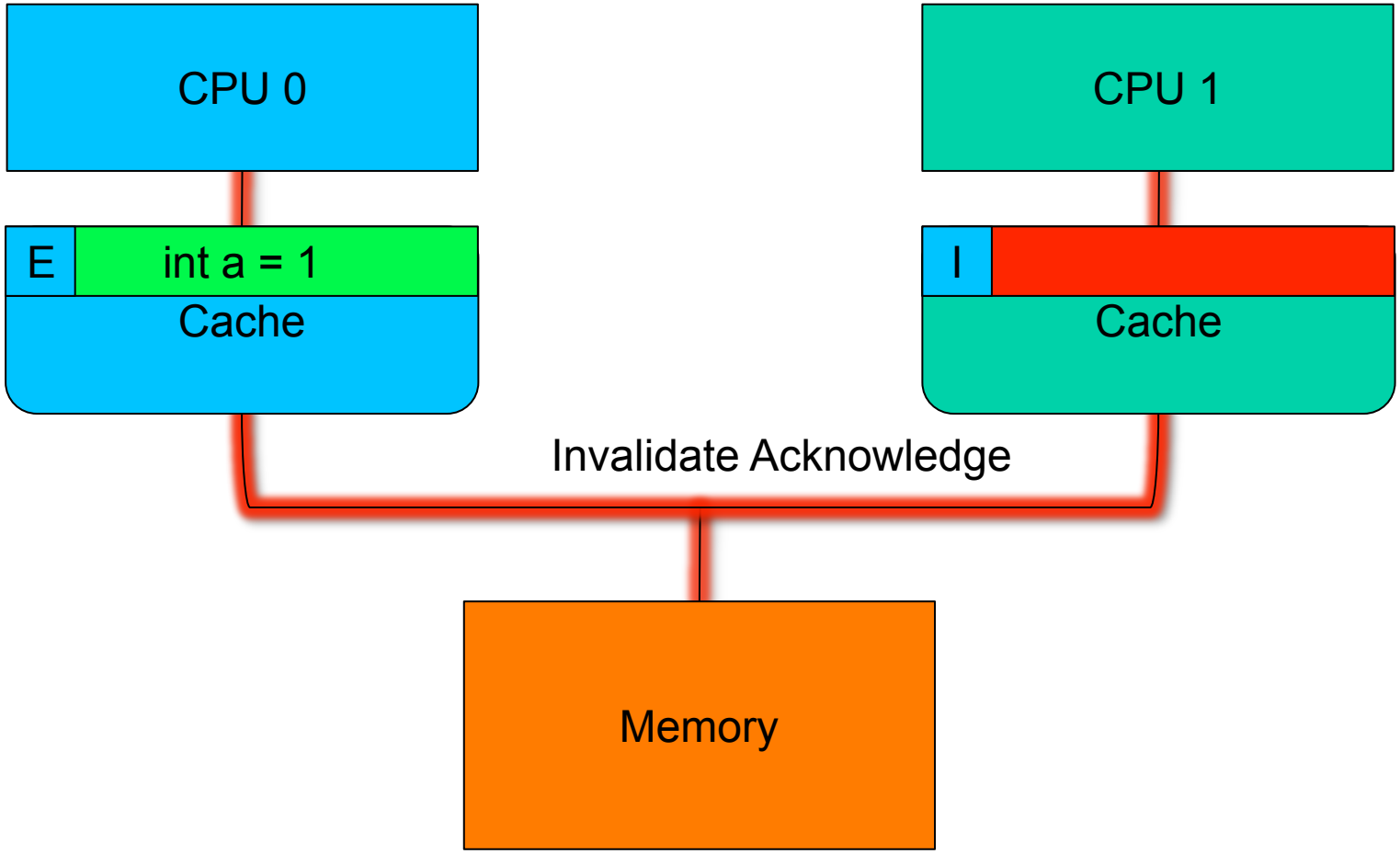
MESI Protokoll



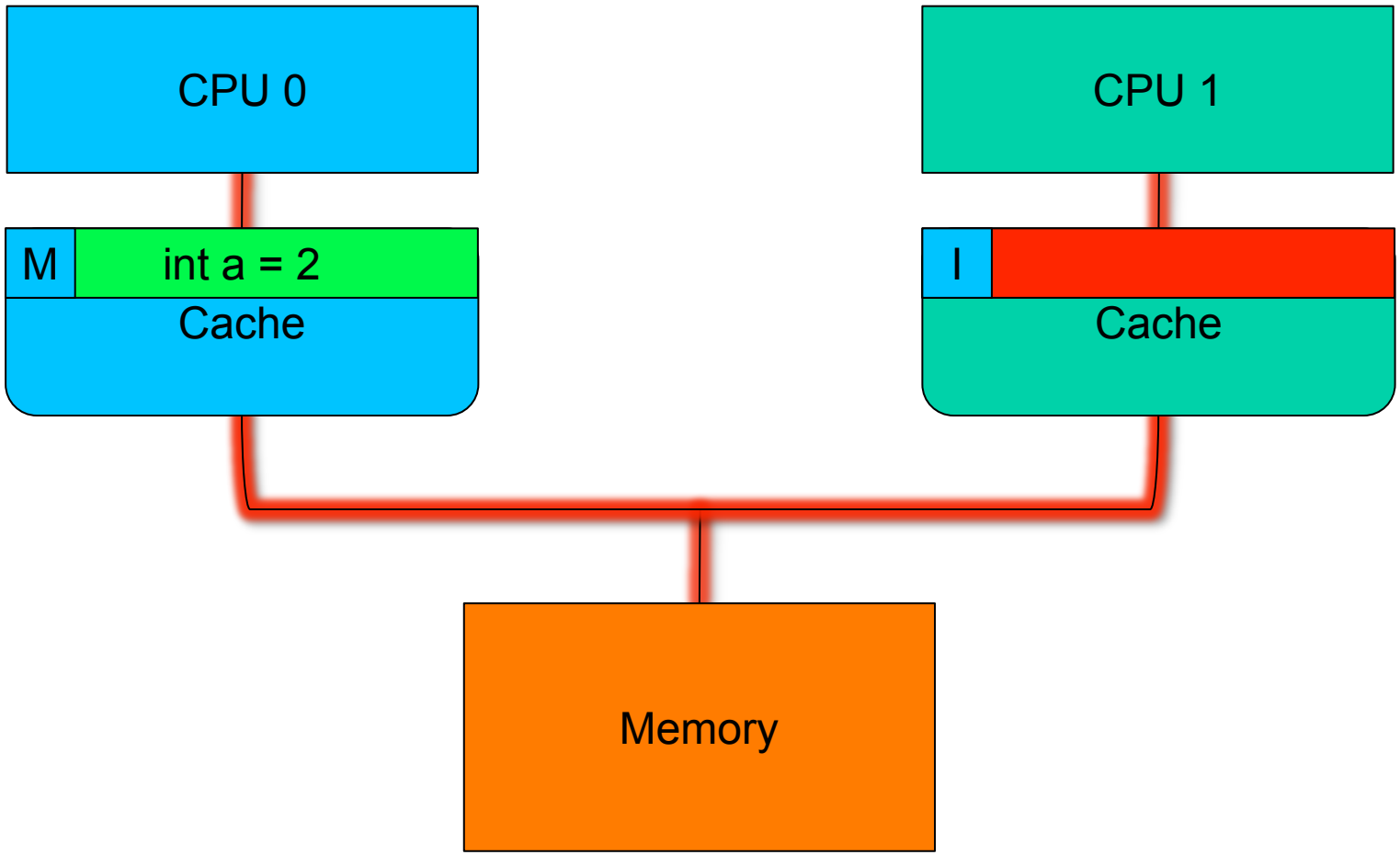
MESI Protokoll



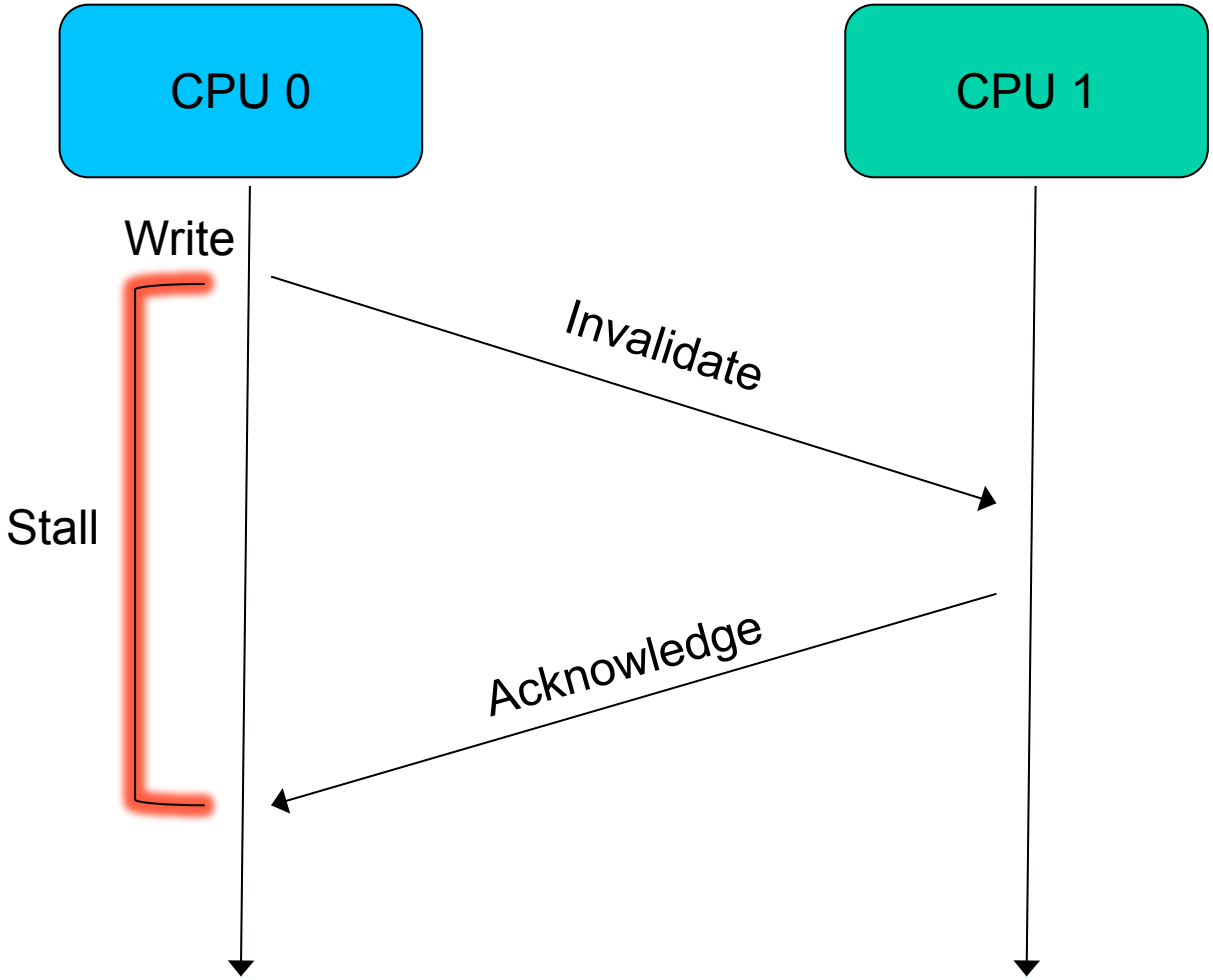
MESI Protokoll



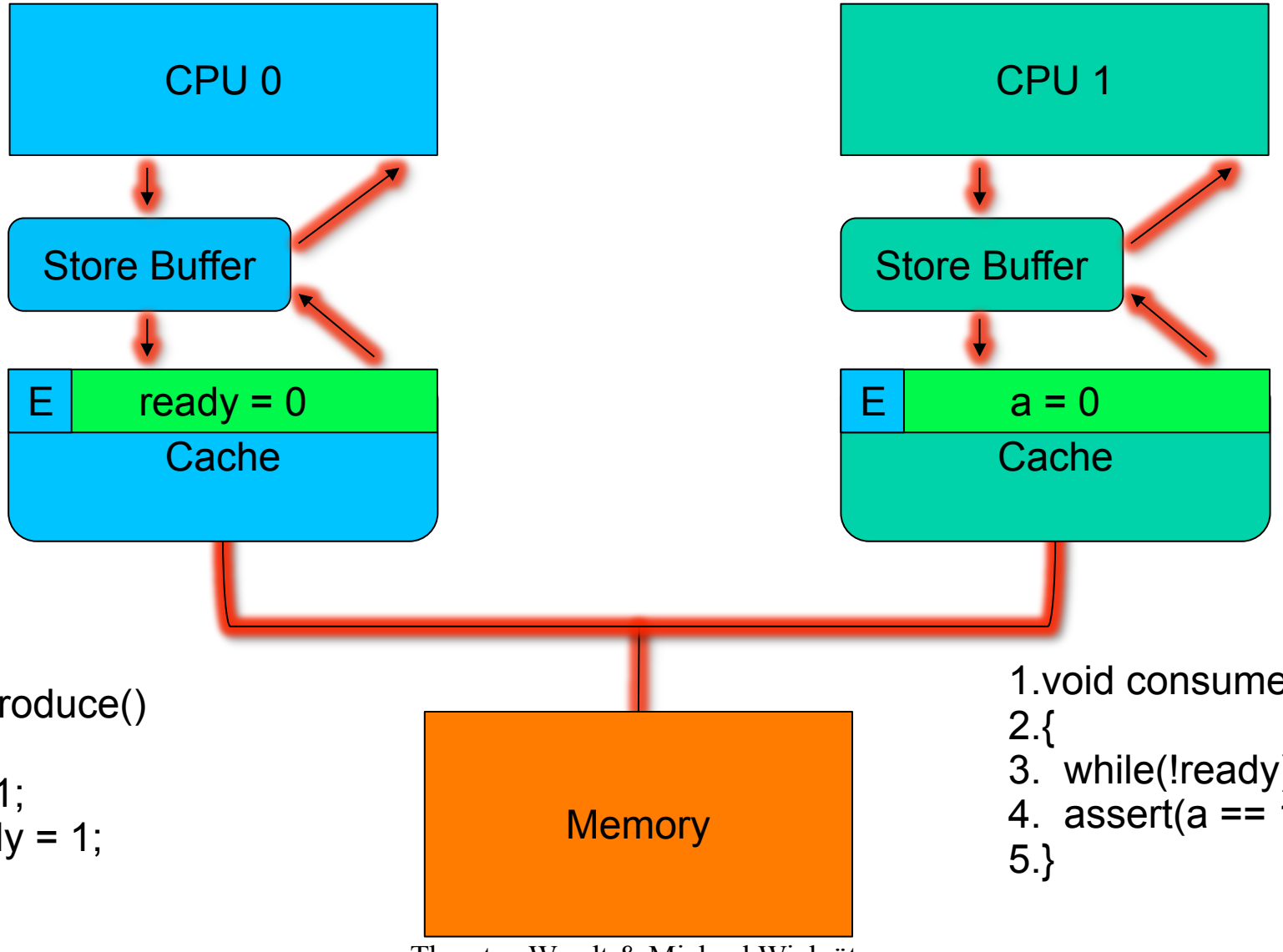
MESI Protokoll



Write Stall



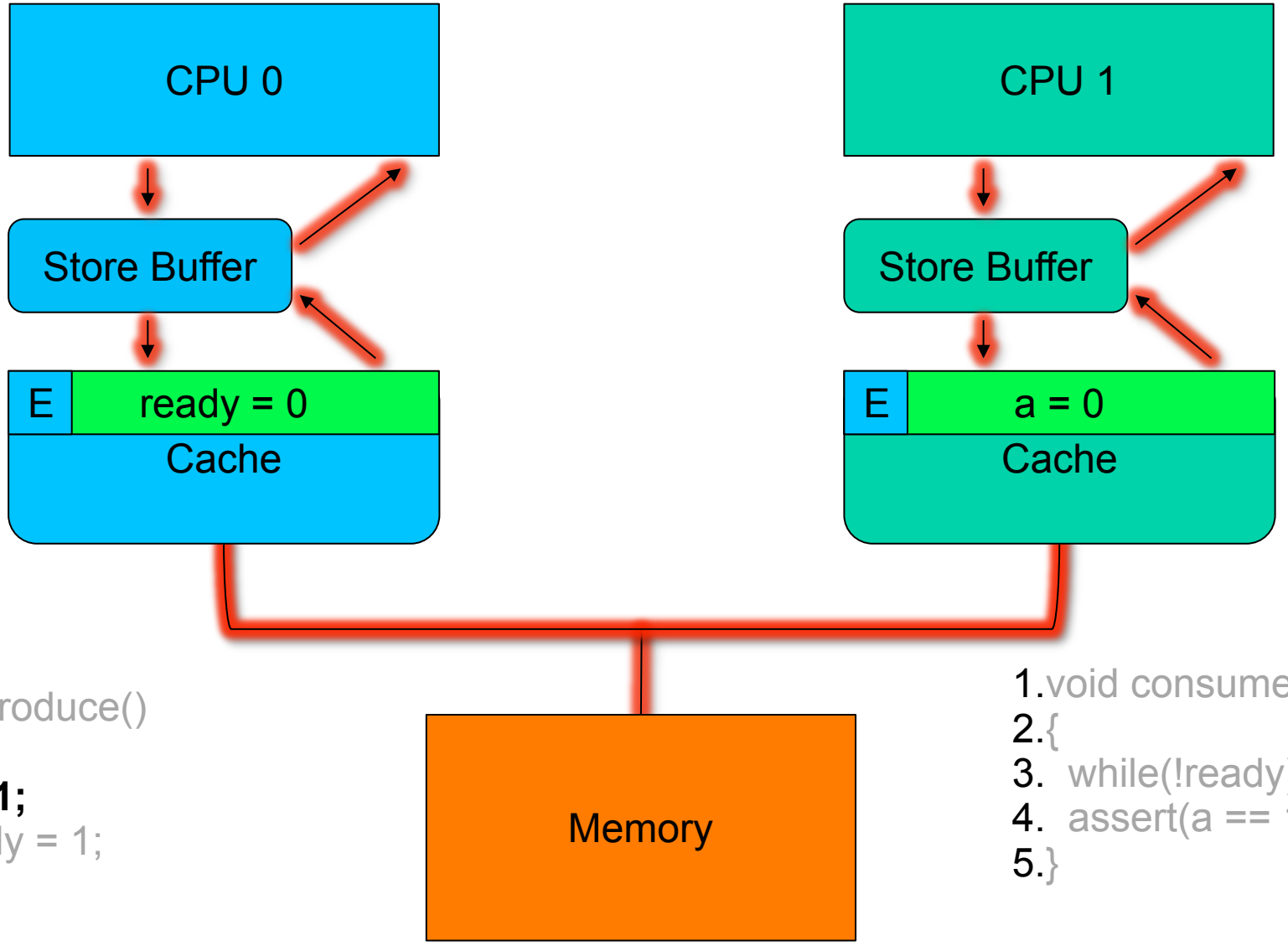
Store Buffer – Die Lösung?




```
1.  void produce()
2.  {
3.     a = 1;
4.     ready = 1;
5.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

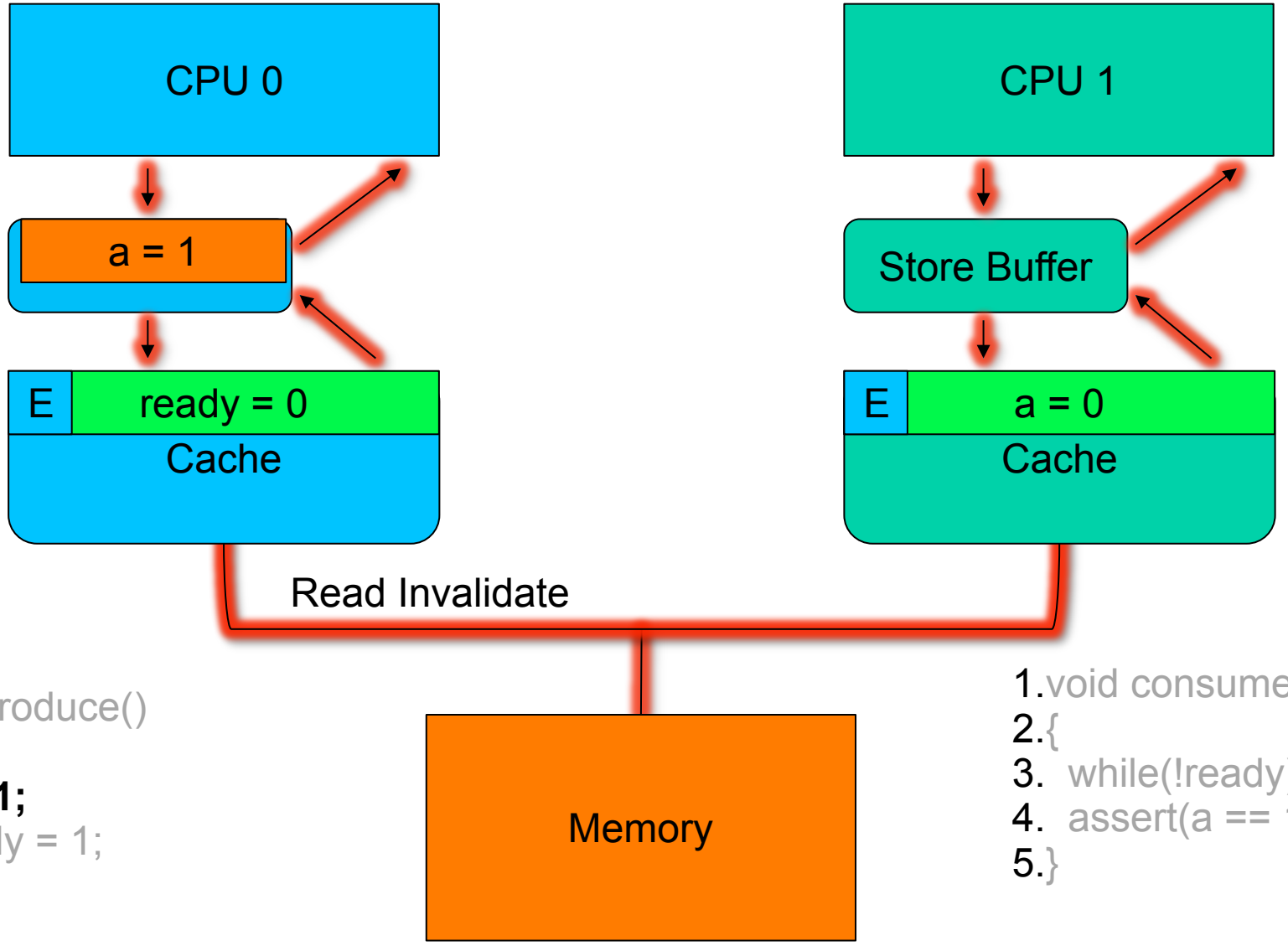
Store Buffer – Die Lösung?




```
1.  void produce()
2.  {
3.     a = 1;
4.     ready = 1;
5.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

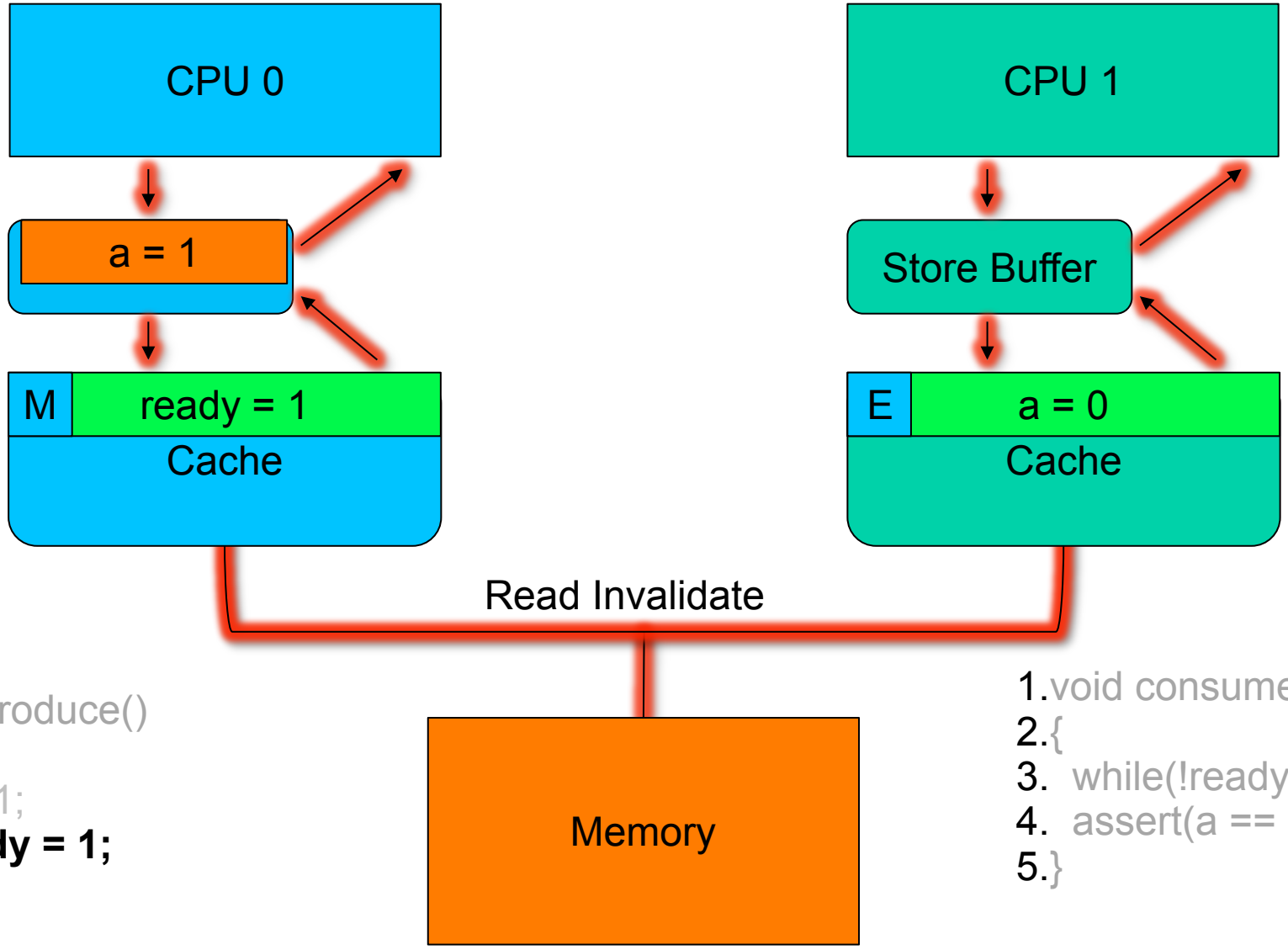
Store Buffer – Die Lösung?




```
1.  void produce()
2.  {
3.     a = 1;
4.     ready = 1;
5.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

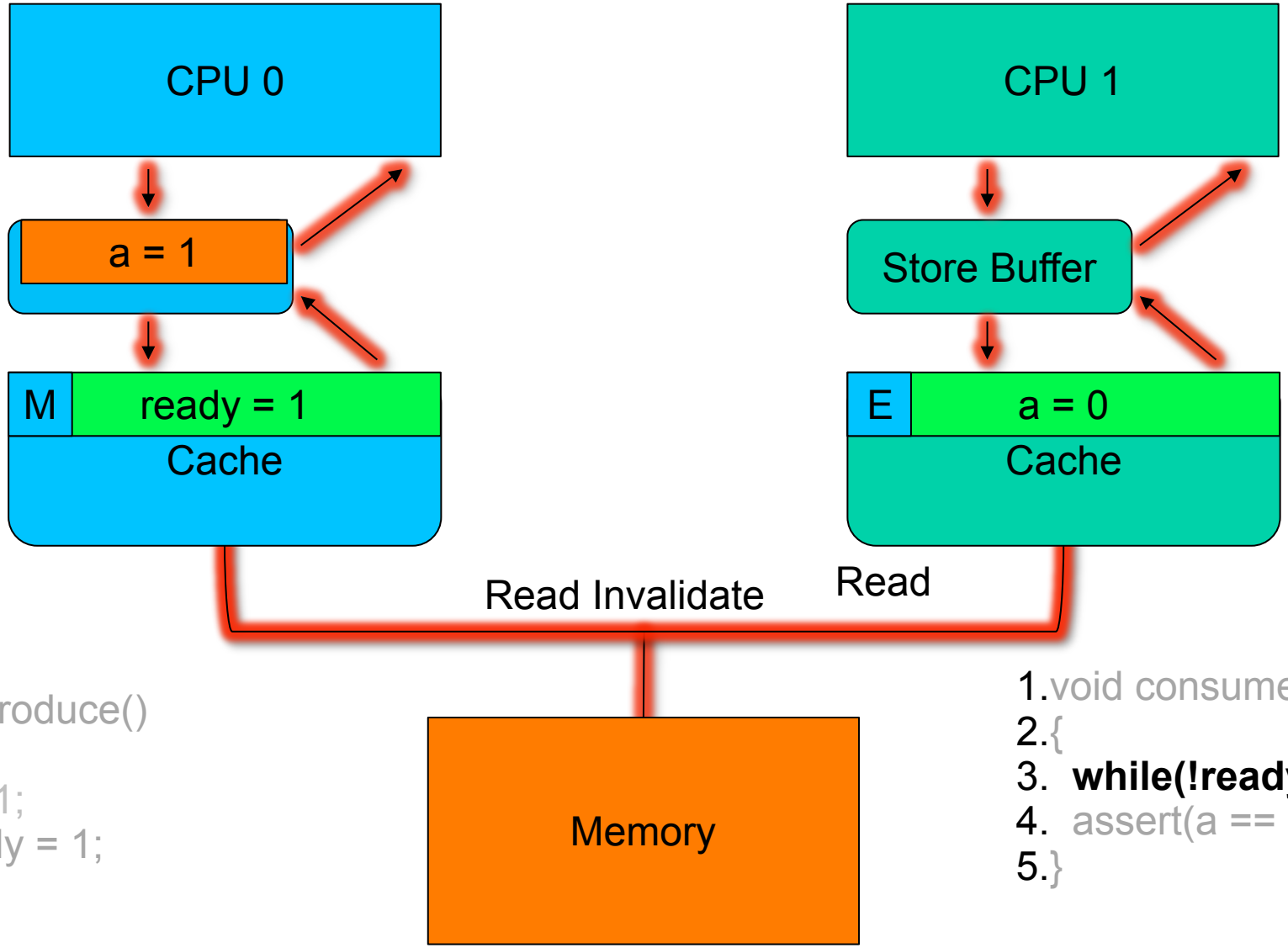
Store Buffer – Die Lösung?




```
1.  void produce()
2.  {
3.     a = 1;
4.     ready = 1;
5.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

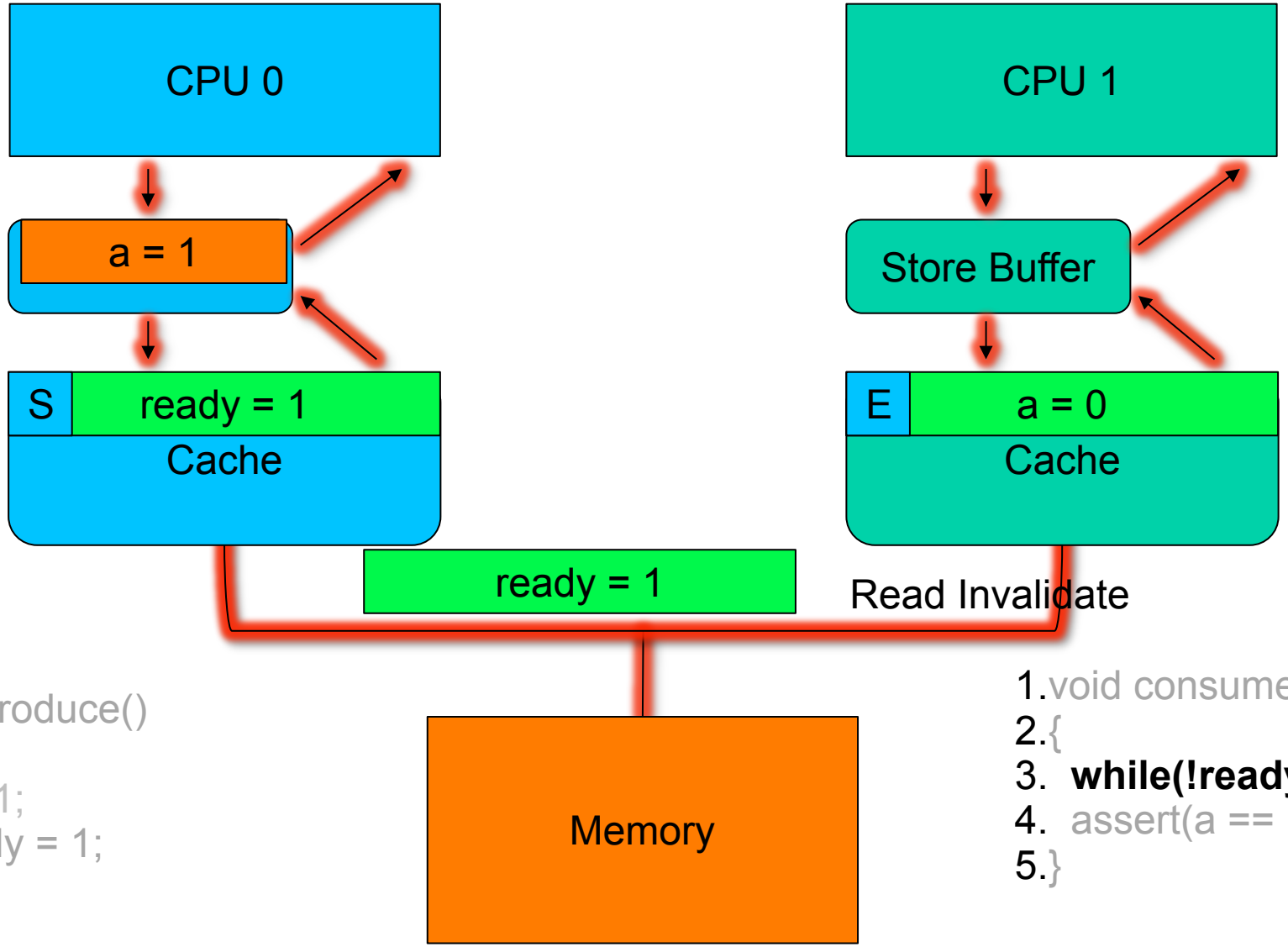
Store Buffer – Die Lösung?




```
1.  void produce()
2.  {
3.     a = 1;
4.     ready = 1;
5.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

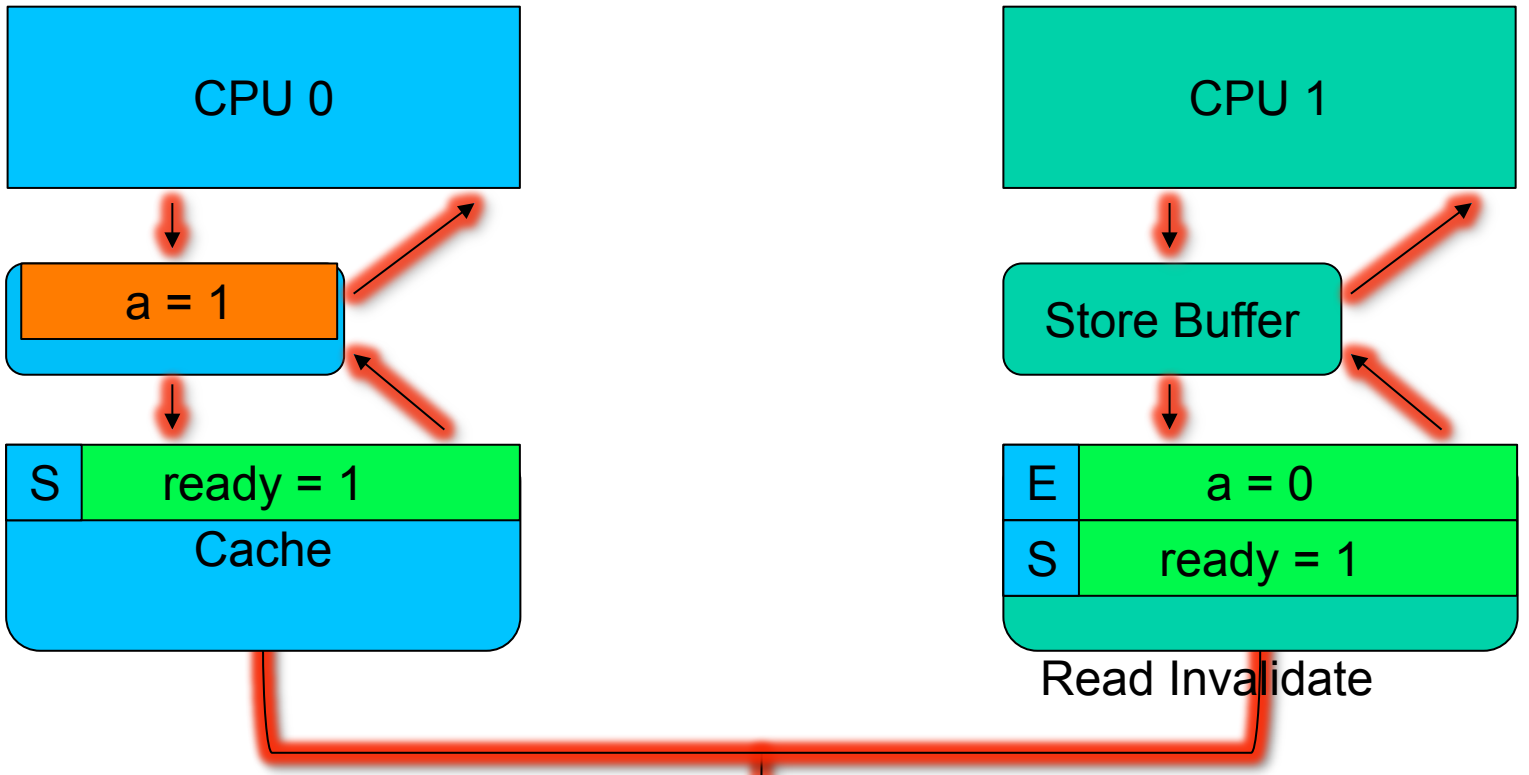
Store Buffer – Die Lösung?



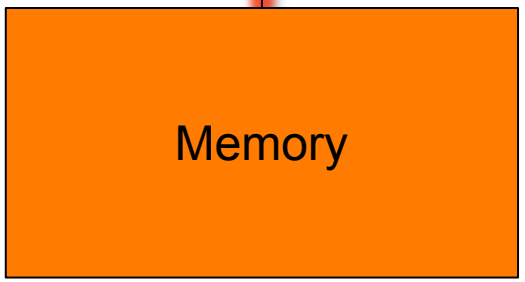

```
1.  void produce()
2.  {
3.    a = 1;
4.    ready = 1;
5.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

Store Buffer – Die Lösung?




```
1.  void produce()
2.  {
3.     a = 1;
4.     ready = 1;
5.  }
```



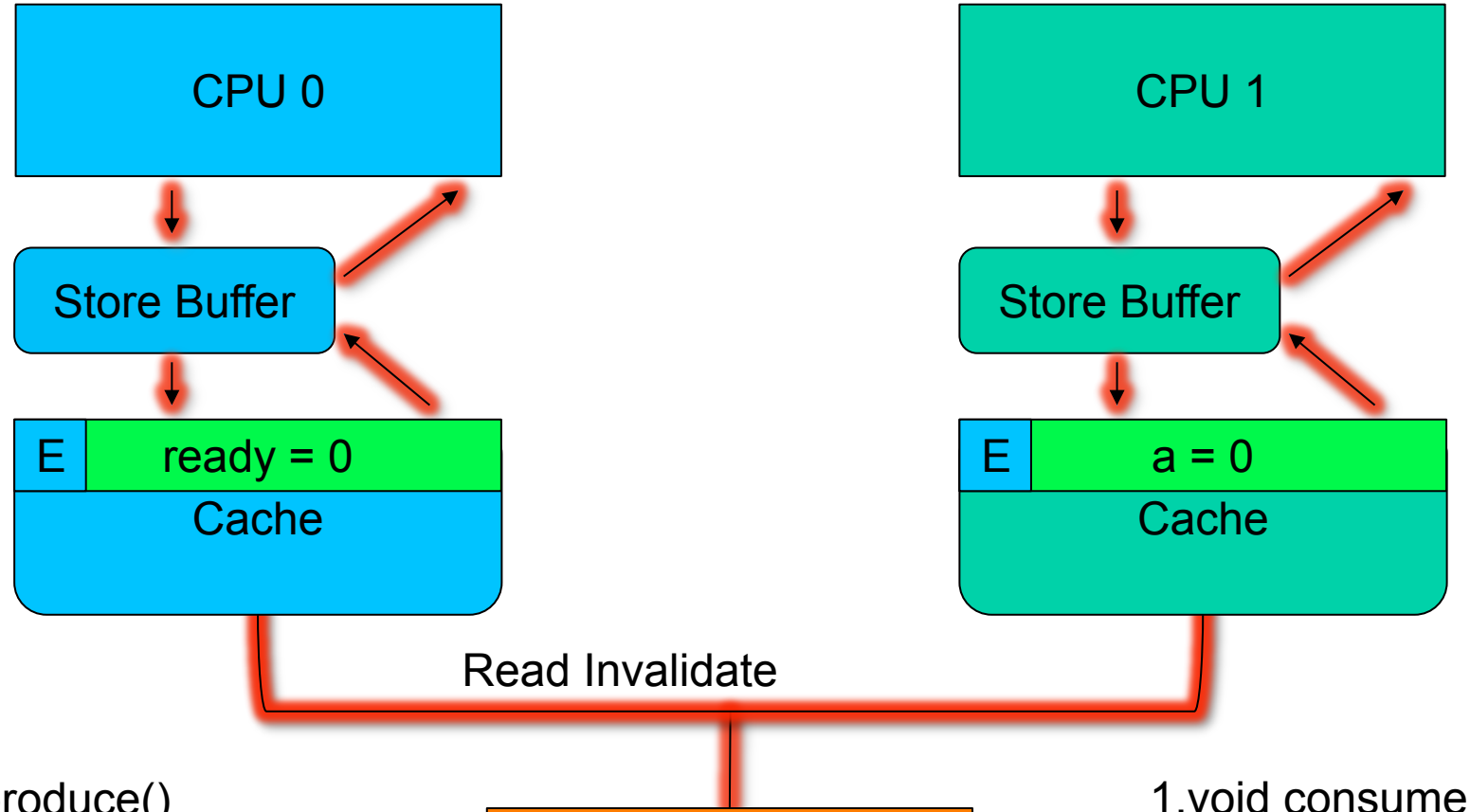

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

Everyone wants to go to heaven, but nobody wants to die. ~Peter Tosh

Store Buffer – Memory Barrier

- Durch Store Buffer kann es zu einem reordering von Änderungen kommen
- Hardware Designer wissen nicht wie Variablen zusammen hängen
- Daher gibt es “Memory Barriers”
  - Erlauben der Software die Sichtbarkeit von Änderungen in allen Caches zu erzwingen

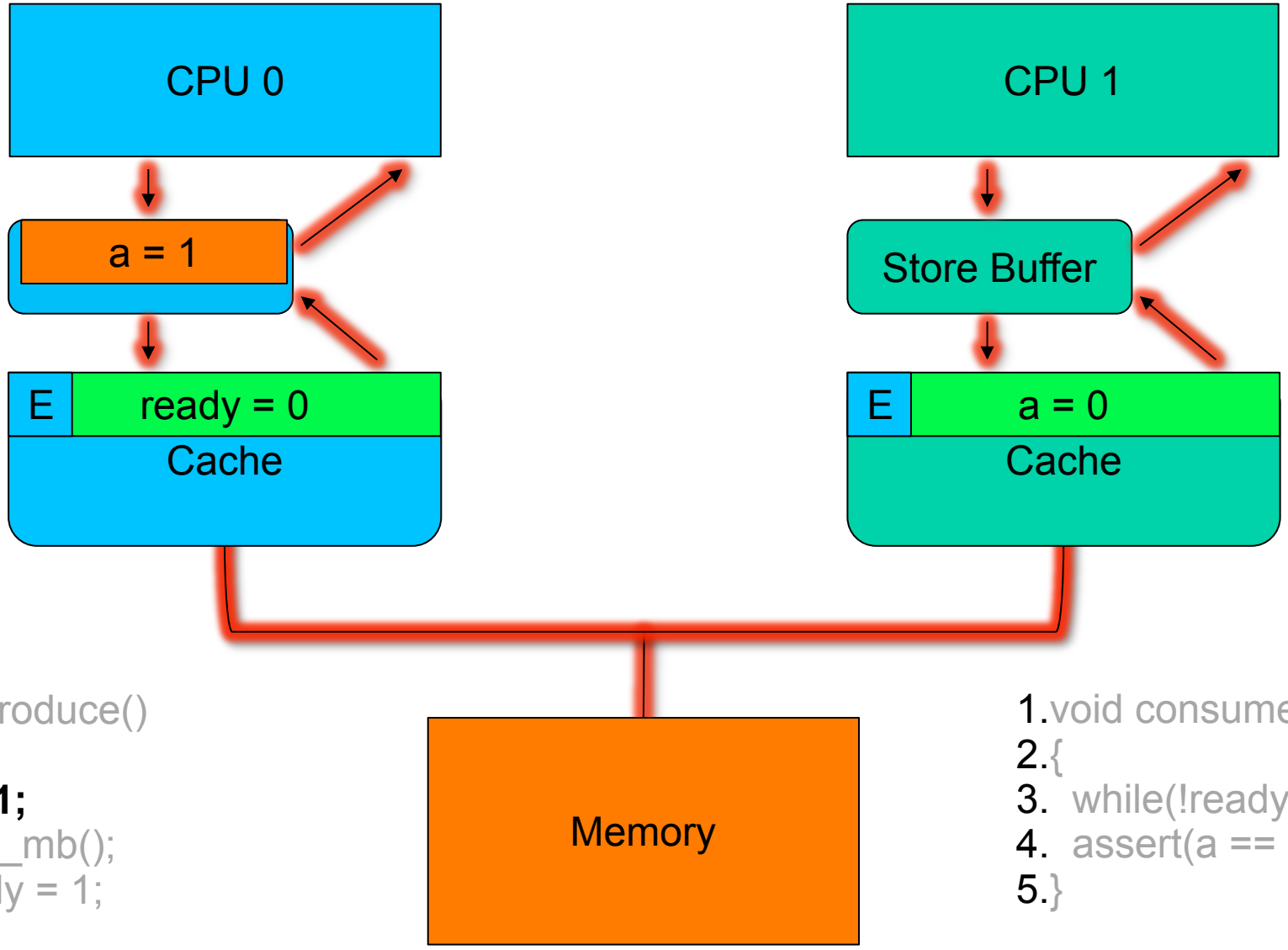
Store Buffer – Memory Barrier




```
1.  void produce()
2.  {
3.      a = 1;
4.      smp_mb();
5.      ready = 1;
6.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

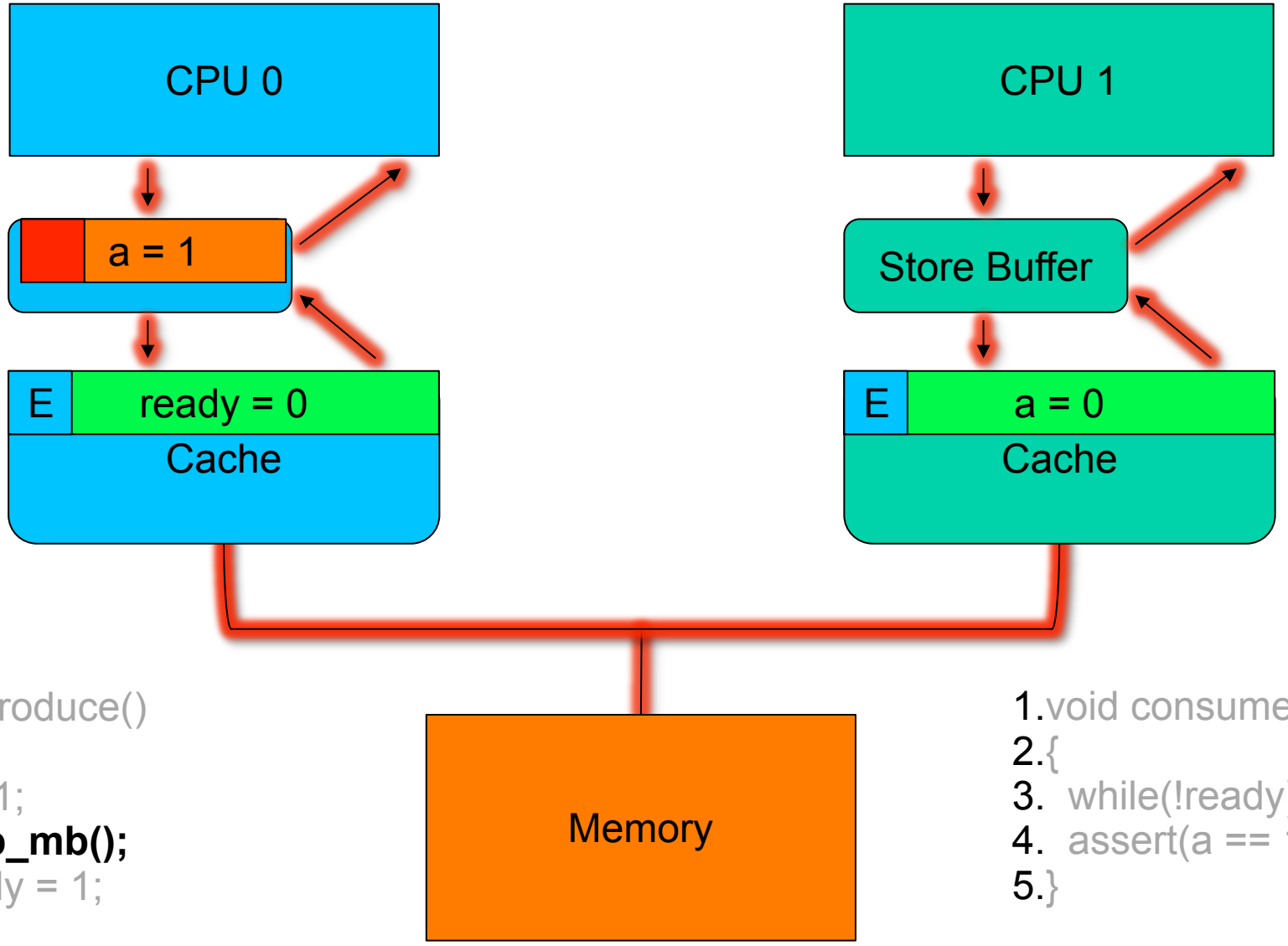
Store Buffer – Memory Barrier




```
1.  void produce()
2.  {
3.     a = 1;
4.     smp_mb();
5.     ready = 1;
6.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

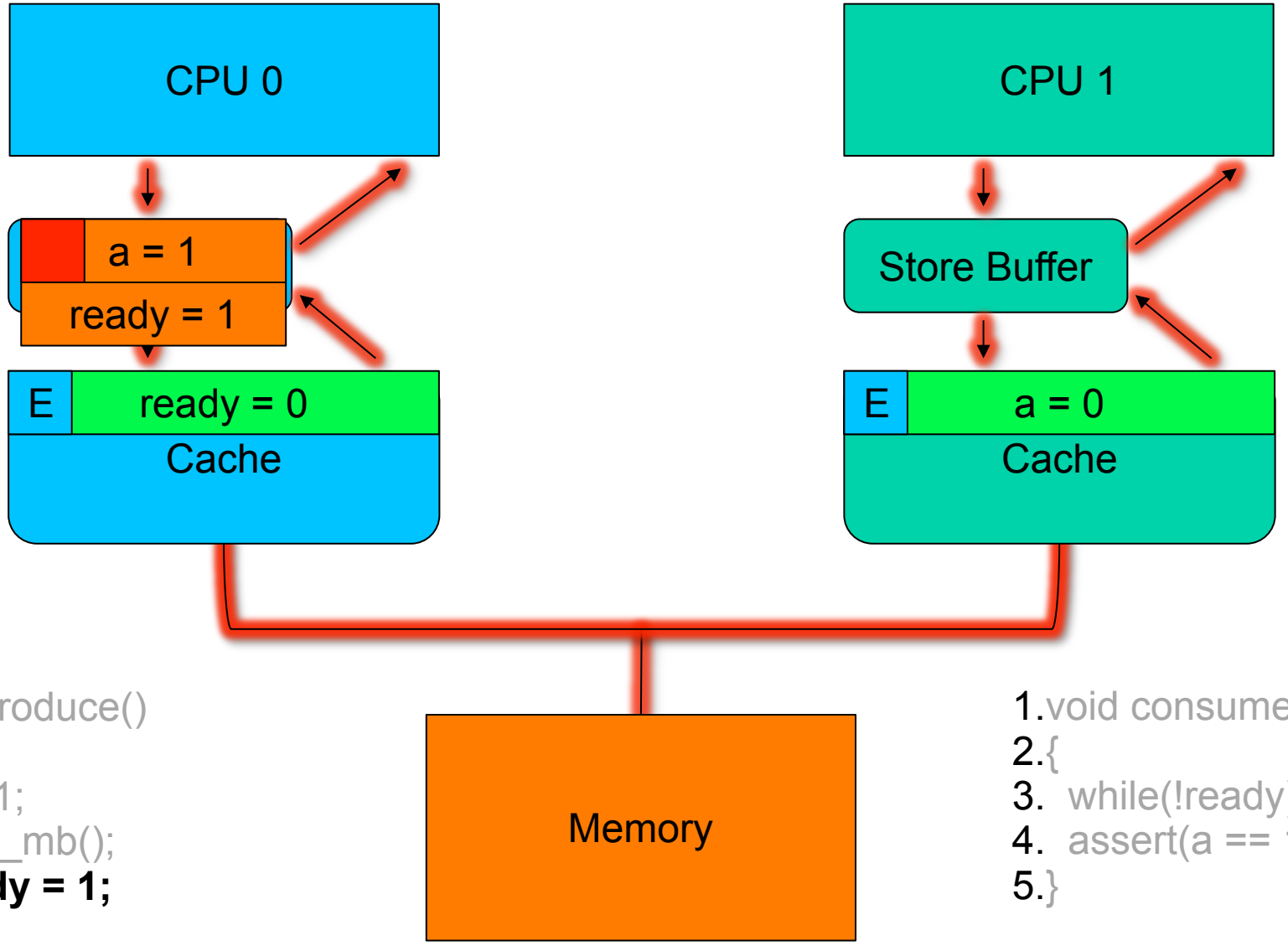
Store Buffer – Memory Barrier




```
1.  void produce()
2.  {
3.      a = 1;
4.      smp_mb();
5.      ready = 1;
6.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

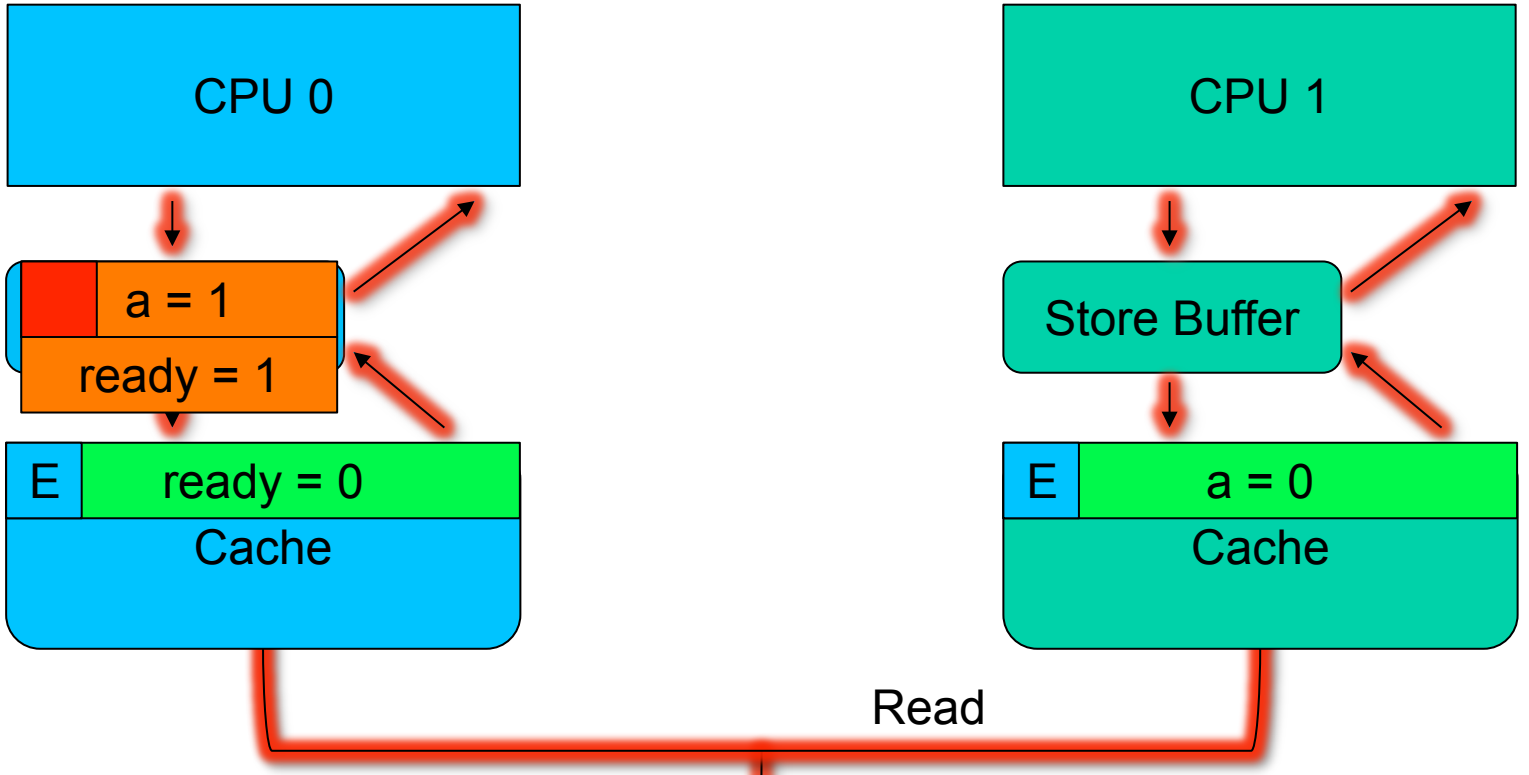
Store Buffer – Memory Barrier




```
1.  void produce()
2.  {
3.      a = 1;
4.      smp_mb();
5.      ready = 1;
6.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

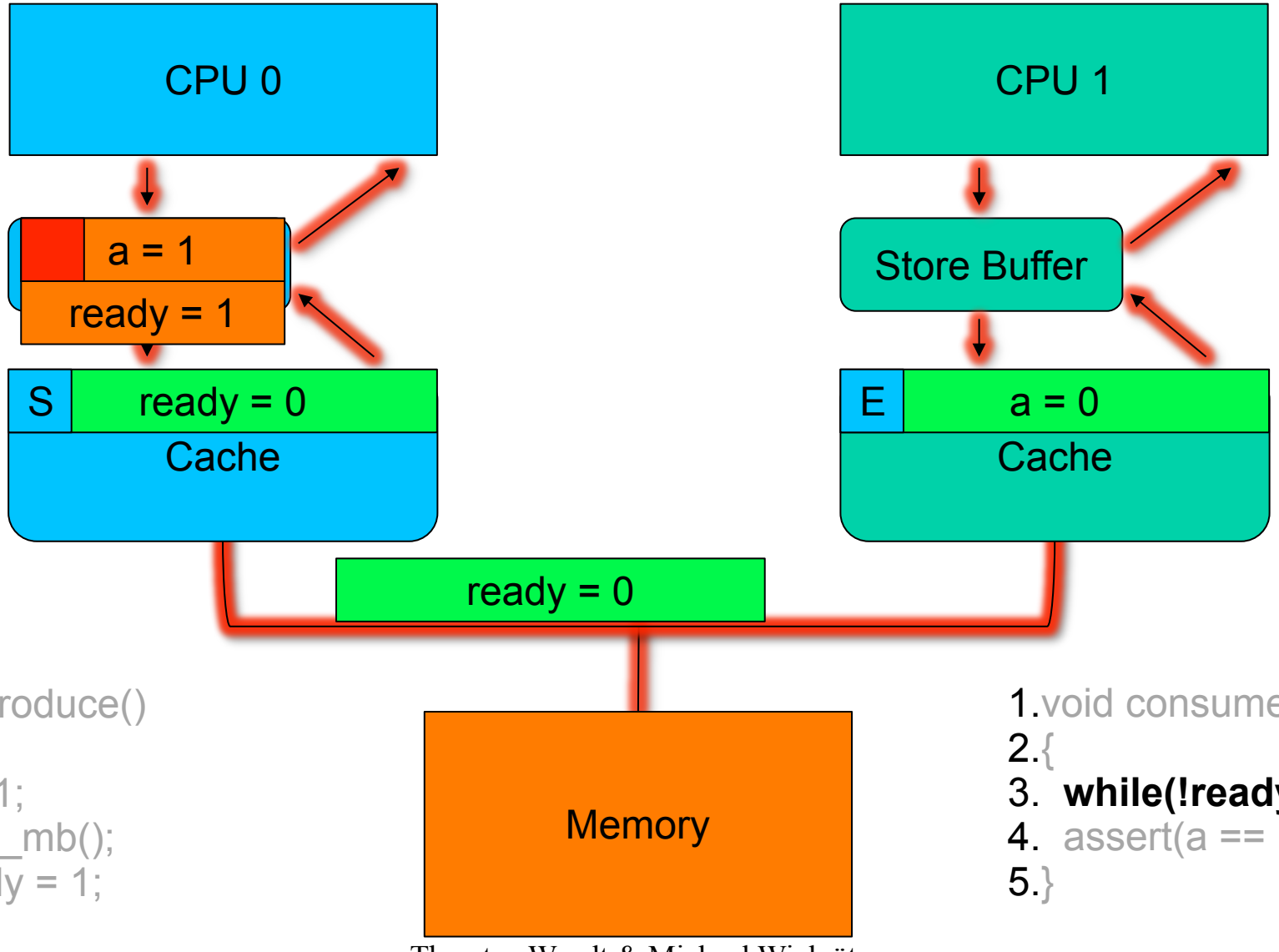
Store Buffer – Memory Barrier




```
1. void produce()
2. {
3.    a = 1;
4.    smp_mb();
5.    ready = 1;
6. }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

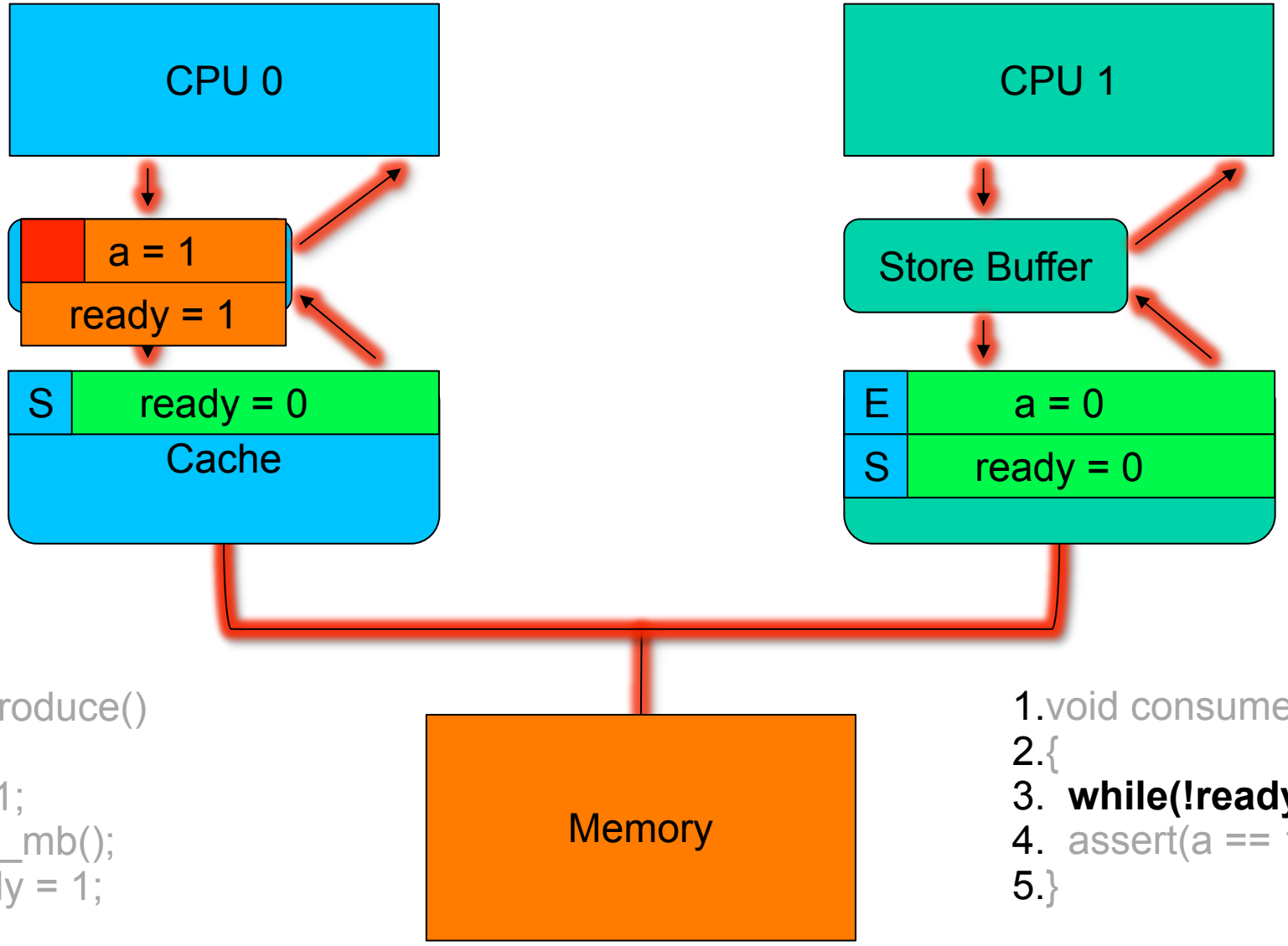
Store Buffer – Memory Barrier




```
1. void produce()
2. {
3.    a = 1;
4.    smp_mb();
5.    ready = 1;
6. }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

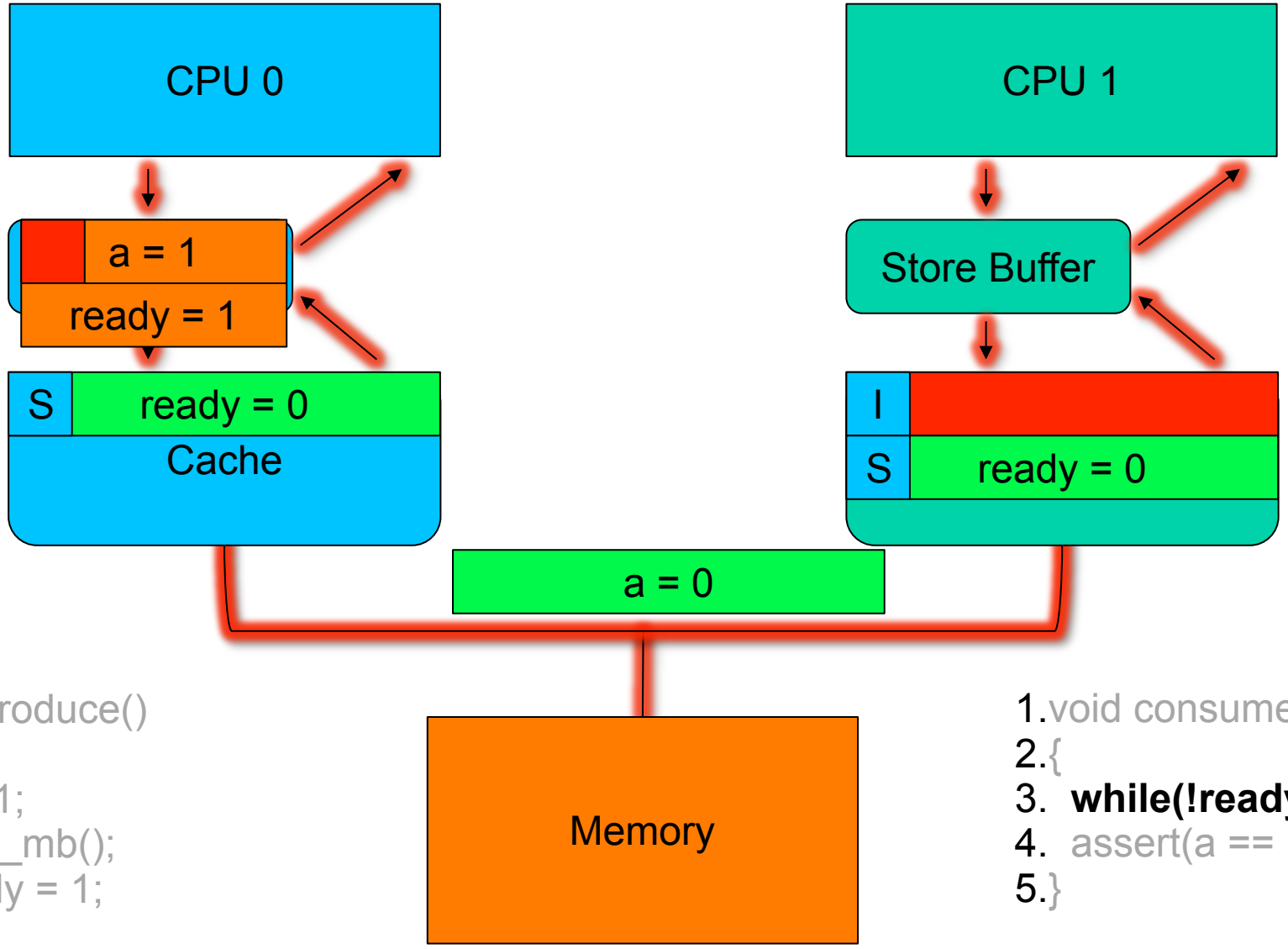
Store Buffer – Memory Barrier




```
1. void produce()
2. {
3.    a = 1;
4.    smp_mb();
5.    ready = 1;
6. }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

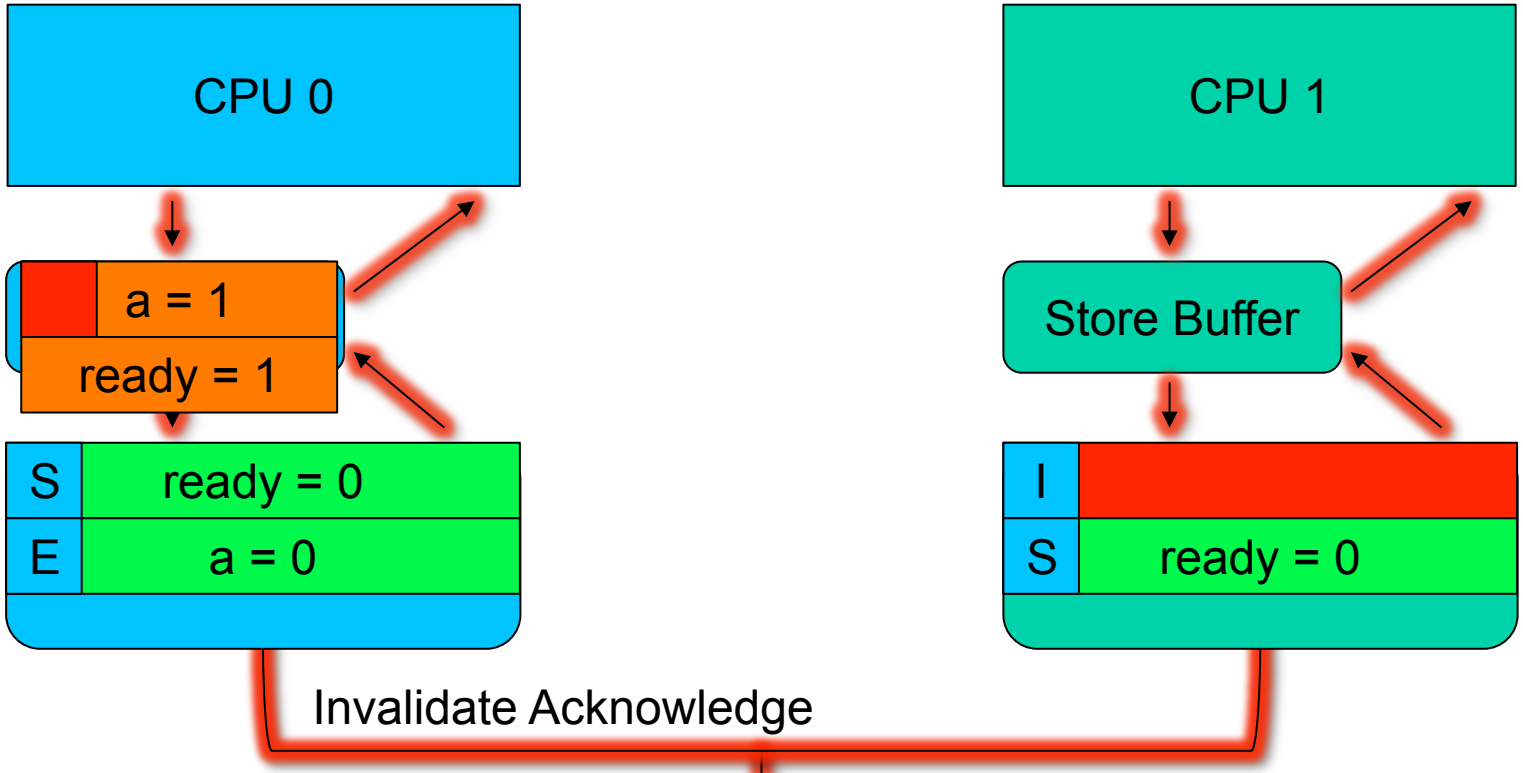
Store Buffer – Memory Barrier




```
1.  void produce()
2.  {
3.      a = 1;
4.      smp_mb();
5.      ready = 1;
6.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

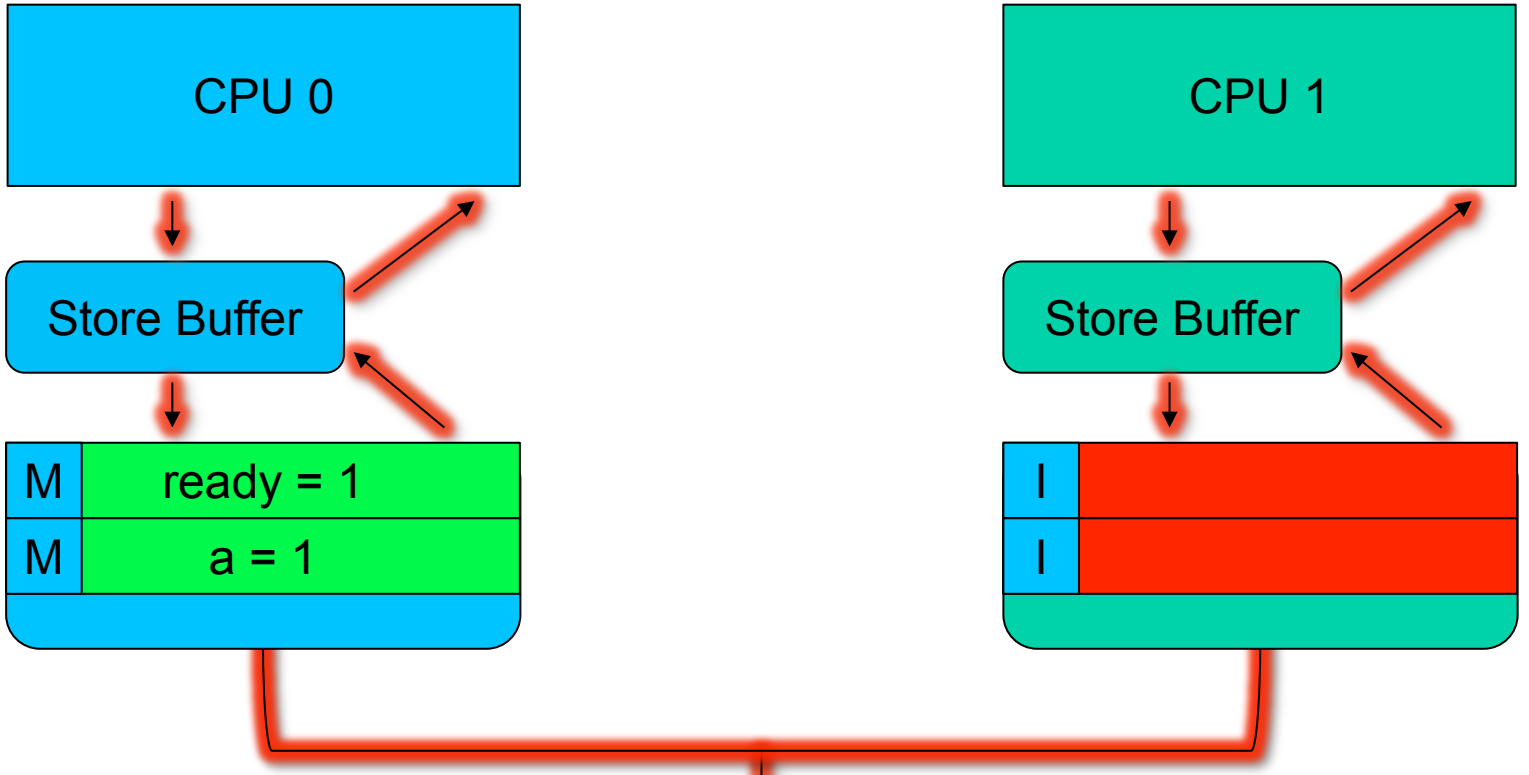
Store Buffer – Memory Barrier




```
1.  void produce()
2.  {
3.      a = 1;
4.      smp_mb();
5.      ready = 1;
6.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

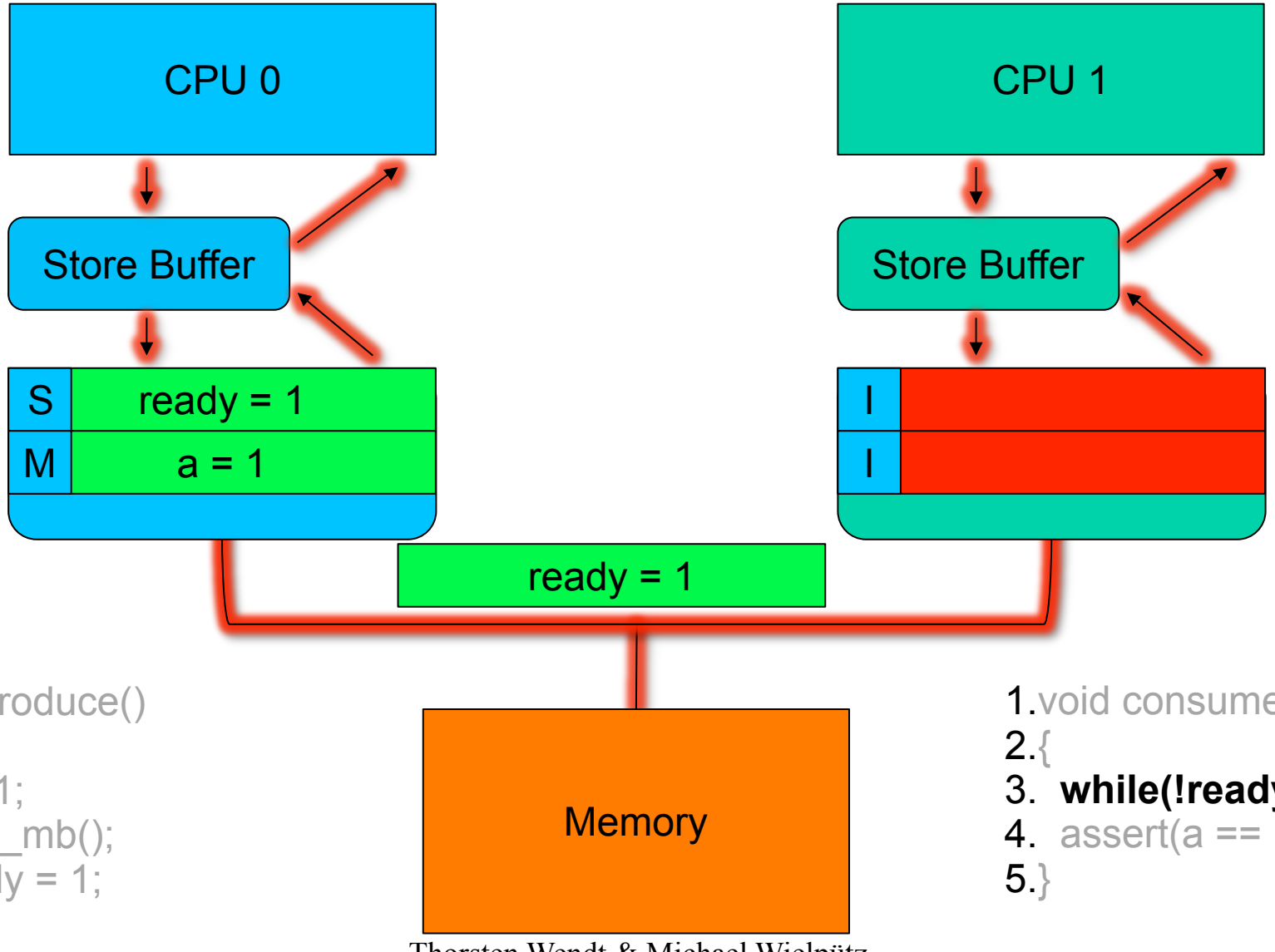
Store Buffer – Memory Barrier




```
1.  void produce()
2.  {
3.      a = 1;
4.      smp_mb();
5.      ready = 1;
6.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

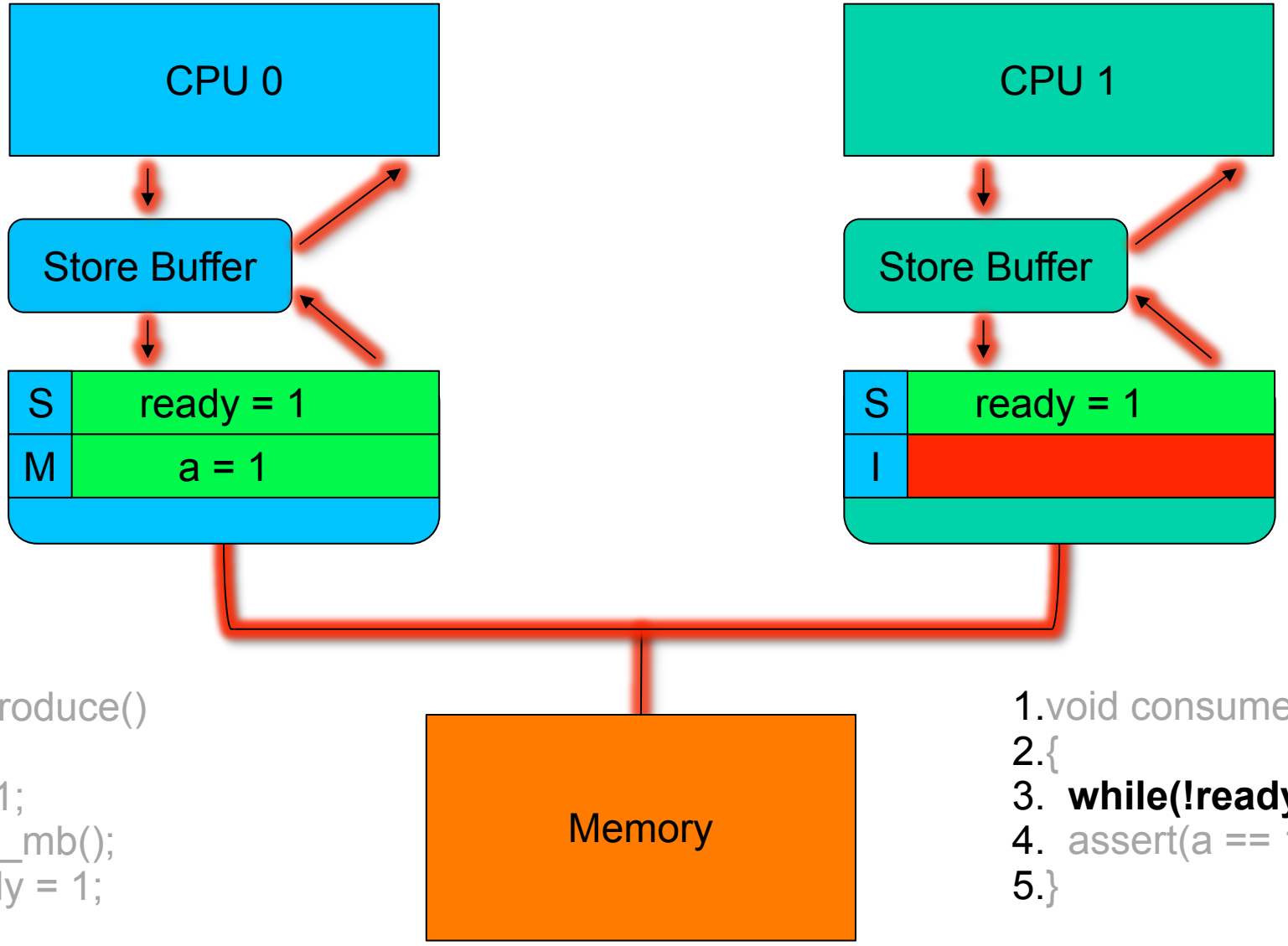
Store Buffer – Memory Barrier




```
1.  void produce()
2.  {
3.      a = 1;
4.      smp_mb();
5.      ready = 1;
6.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

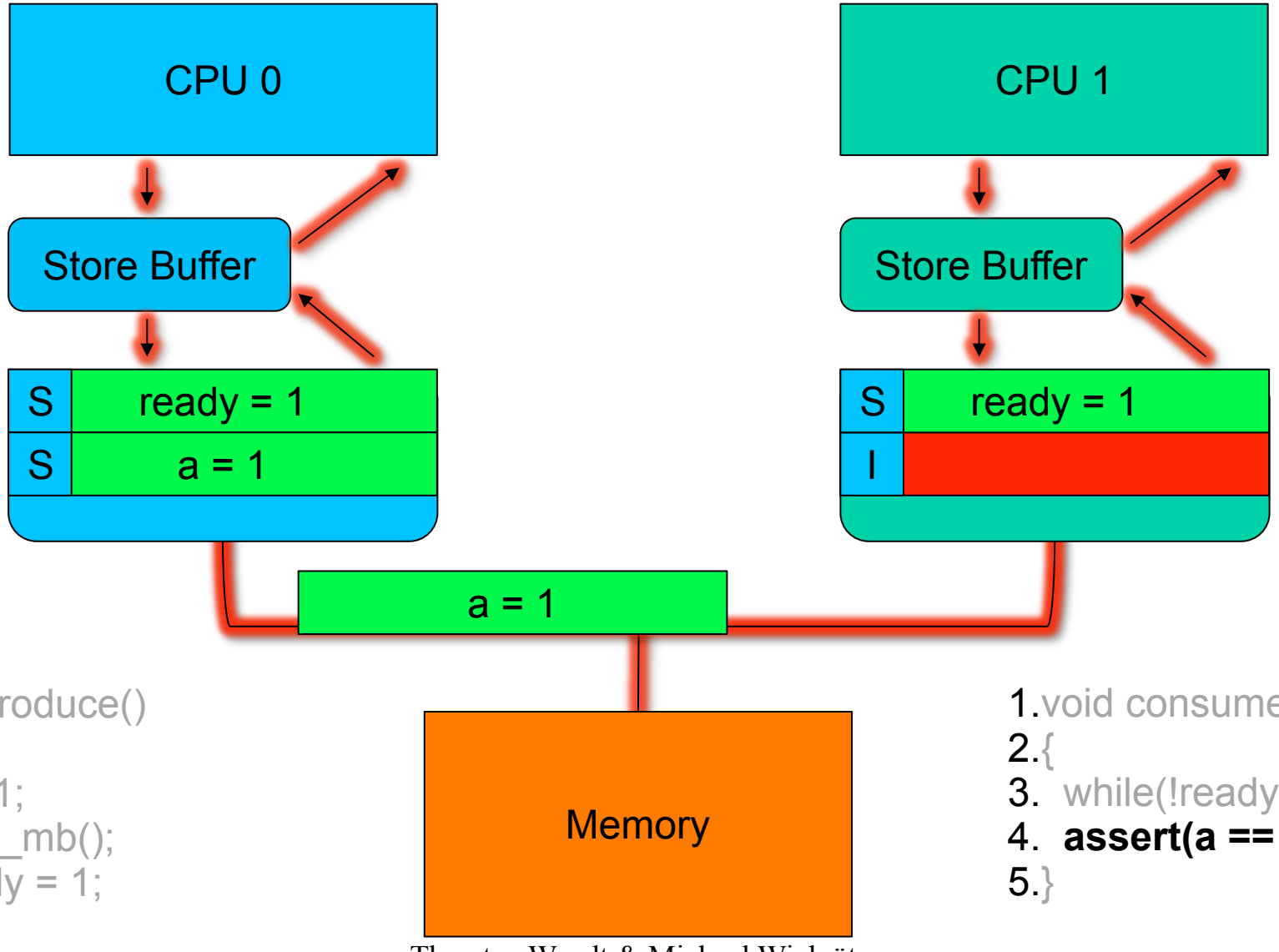
Store Buffer – Memory Barrier




```
1. void produce()
2. {
3.    a = 1;
4.    smp_mb();
5.    ready = 1;
6. }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

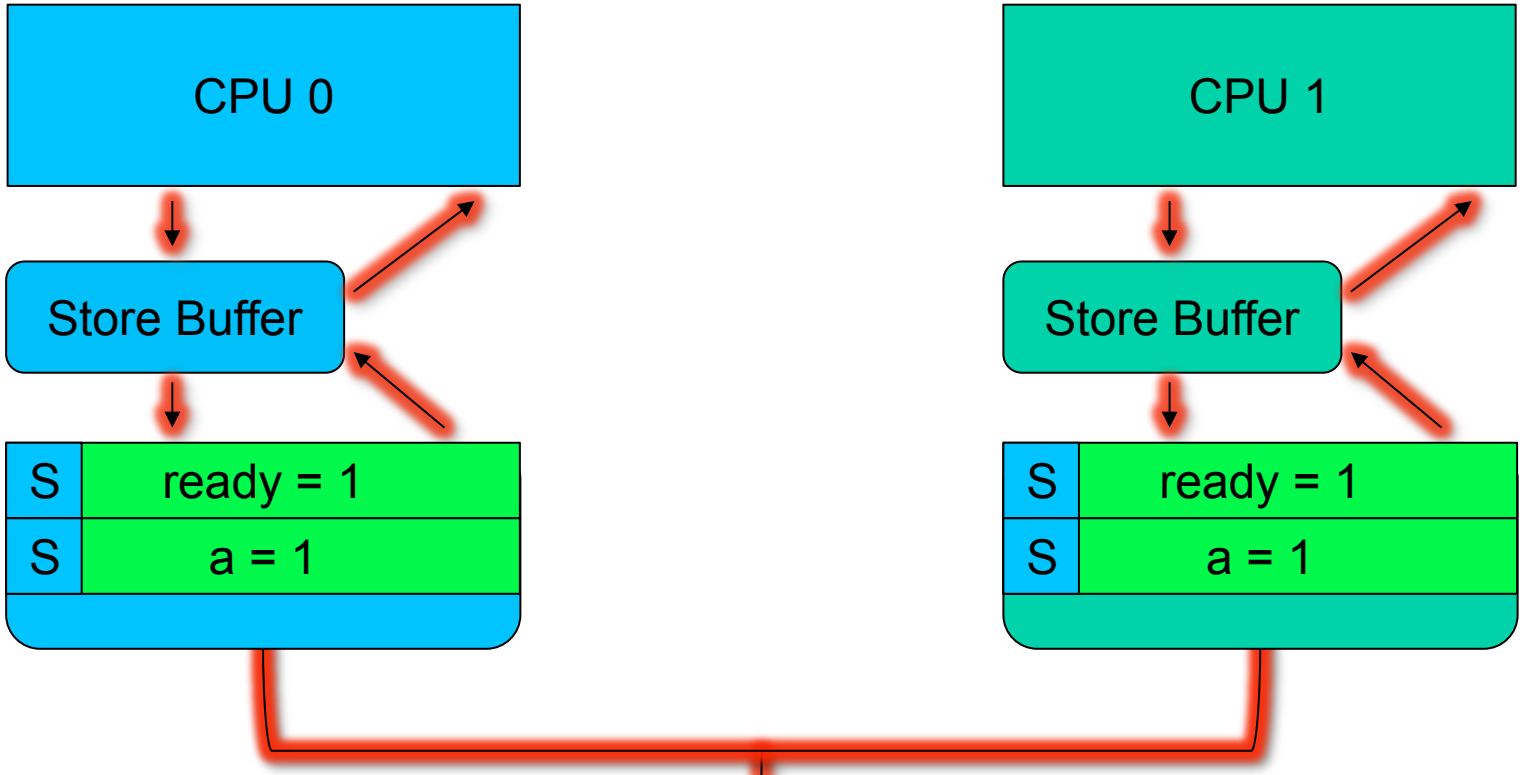
Store Buffer – Memory Barrier




```
1.  void produce()
2.  {
3.      a = 1;
4.      smp_mb();
5.      ready = 1;
6.  }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

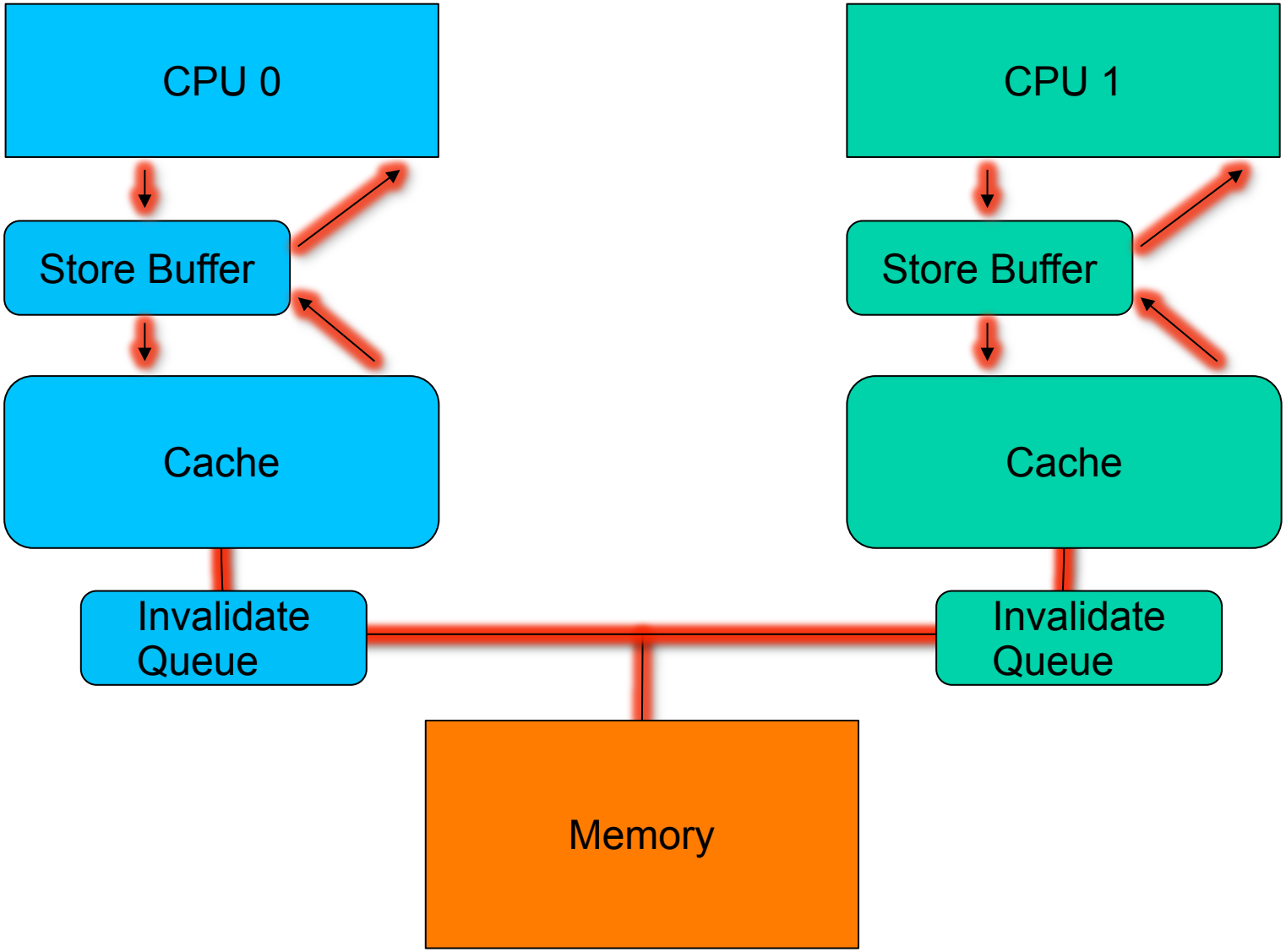
Store Buffer – Memory Barrier




```
1. void produce()
2. {
3.    a = 1;
4.    smp_mb();
5.    ready = 1;
6. }
```

```
1.void consume()
2.{
3.  while(!ready) {}
4.  assert(a == 1)
5.}
```

Invalidate Queues



Invalidate Queues – Memory Barrier

```
1.  void producer()
2.  {
3.     a = 1;
4.     smp_mb();
5.     ready = 1;
6.  }

7.  void consumer()
8.  {
9.    while(! ready) {}
10.   smp_mb();
11.   assert(a == 1);
12. }
```

Atomic! Alles klar?

- Wird ein std::shared_ptr per value übergeben, so wird der atomic zweimal verändert!
- Soweit alles klar?
- Was ist nun aber so schlimm am atomic ?

Gar nichts! Ein atomic ist nur teuer !

std::atomic

```
bool compare_and_swap(int *dest,
                      int& oldval,
                      int newval)
{ // Beware Pseudo-Code...
  smp_mb(); // Read Barrier
  if (oldval == *dest) {
      *dest = newval;
      smp_mb(); // Write Barrier
      return true;
  } else {
      oldval = *dest;
      return false;
  }
}
```

Messung

shared_ptr Übergabe “by value”:

```
360977.690921 task-clock                #    7.660 CPUs utilized
772,320,139,727 cycles                  #    2.140 GHz                      [83.39%]
728,438,855,520 stalled-cycles-frontend #   94.32% frontend cycles idle     [83.27%]
633,864,803,337 stalled-cycles-backend  #   82.07% backend  cycles idle     [66.55%]
117,720,264,553 instructions            #    0.15  insns per cycle
                                        #    6.19  stalled cycles per insn [83.35%]
18,447,531,734 branches                 #   51.104 M/sec                    [83.43%]
5,290,863 branch-misses                 #    0.03% of all branches          [83.37%]

47.127517406 seconds time elapsed
```

shared_ptr Übergabe “by reference”:

```
9115.128729 task-clock                  #    6.424 CPUs utilized
22,671,956,627 cycles                   #    2.487 GHz                      [83.06%]
15,418,085,375 stalled-cycles-frontend  #   68.01% frontend cycles idle     [83.16%]
3,715,144,620 stalled-cycles-backend    #   16.39% backend  cycles idle     [66.63%]
23,992,564,292 instructions             #    1.06  insns per cycle
                                        #    0.64  stalled cycles per insn [83.17%]
3,208,273,085 branches                  #  351.972 M/sec                    [83.52%]
338,330 branch-misses                   #    0.01% of all branches          [83.78%]

1.418994024 seconds time elapsed
```

- Messungen mit perf. Testprogramm wird auf die Webseite hochgeladen.
- Testprogramm wurde mit 8 Threads ausgeführt.

Fazit

- shared_ptr haben einen Preis
- Man kann Performance-Kosten minimieren durch Übergabe “by reference”
- Übergabe “by reference” hat auch Nachteile:
  - mangelnde Sicherheit
  - Ownership muss klar sein

Fazit:

Fragen?