arm

Multicore Processors

Module 8

Module Syllabus

- Multicore
- Communication
  - Message passing
  - Shared memory
- Cache-coherence protocol
  - MESI protocol states
  - MESI protocol transitions
  - Visualizing the protocol
  - MESI state transition diagram
- Memory consistency

Motivation

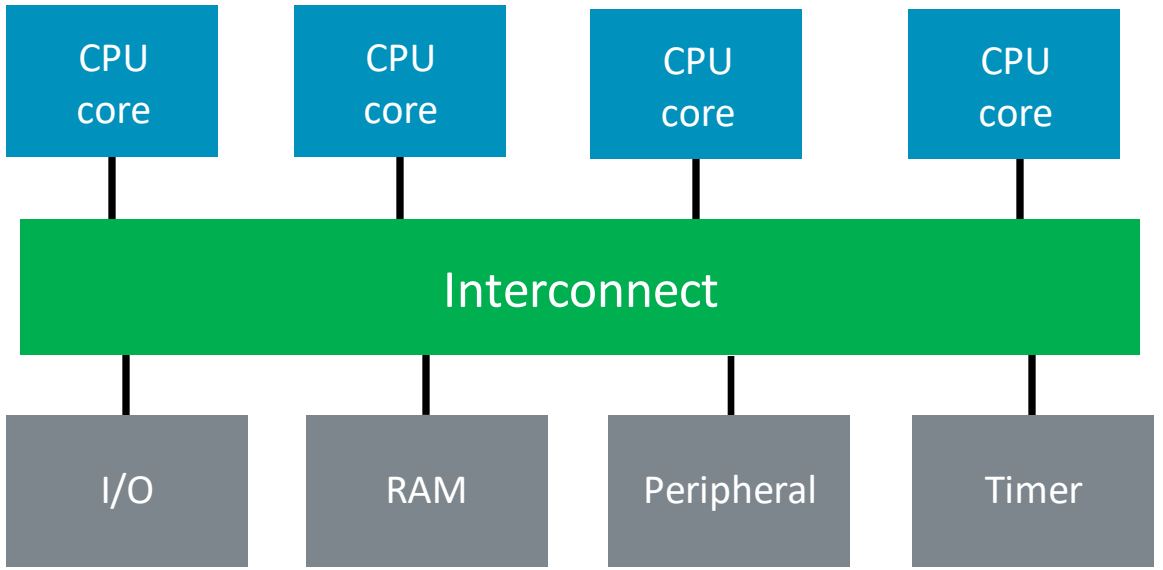
- Moore's law meant that the number of transistors available to an architect kept increasing.
  - Historically, these were used to improve the performance of a single core processor.
  - Usually through increased speculation support, but this gives sublinear performance improvement
- However, the breakdown of Dennard scaling meant these schemes were no longer viable.
  - If they consumed large amounts of power without giving commensurate performance improvements.
- Multicore architectures are an efficient way of using these transistors instead.
  - Performance comes from parallelism, specifically thread-level or process-level parallelism.
  - Note that we had multi-processor systems long before Dennard scaling failed; this just pushed them to mainstream.
- What are the challenges in providing multiple cores and how do they communicate?

Multicore

Overview

- In a multicore processor, multiple CPU cores are provided within the chip.
- Cores are connected together through some form of interconnect.
- Cores share some components on chip.
  - For example, memory interface, or a cache level

An example of multicore system



arm

Multicore

- Cores in a multicore processor are connected together and can collaborate.
- There are a number of challenges to consider when creating a system like this.
  - How do cores communicate with each other?
  - How is data synchronized?
  - How do we ensure that cores don't get stale data when it's been modified by other cores?
  - How do cores see the ordering of events coming from different cores?
- We'll explore each of these by considering the concepts of
  - Shared memory and message passing
  - Cache coherence
  - Memory consistency

Communication

Communication

Message Passing

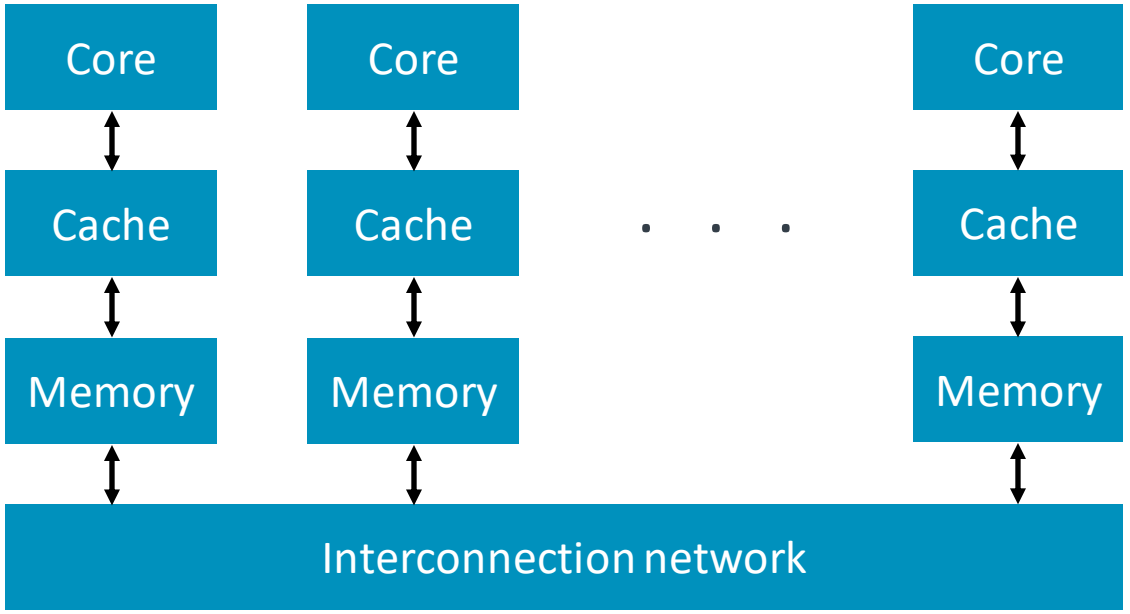
- In this paradigm, applications running on each core wrap up the data they want to communicate into messages sent to other cores.
- Explicit communication via *send* and *receive* operations
- Synchronization is implicit via blocks of messages.

Shared Memory

- In this paradigm, there is a shared memory address space accessible to all cores, where they can read and write data.
- Implicit communication via memory accesses
- Synchronization is performed using atomic memory operations.

Message Passing

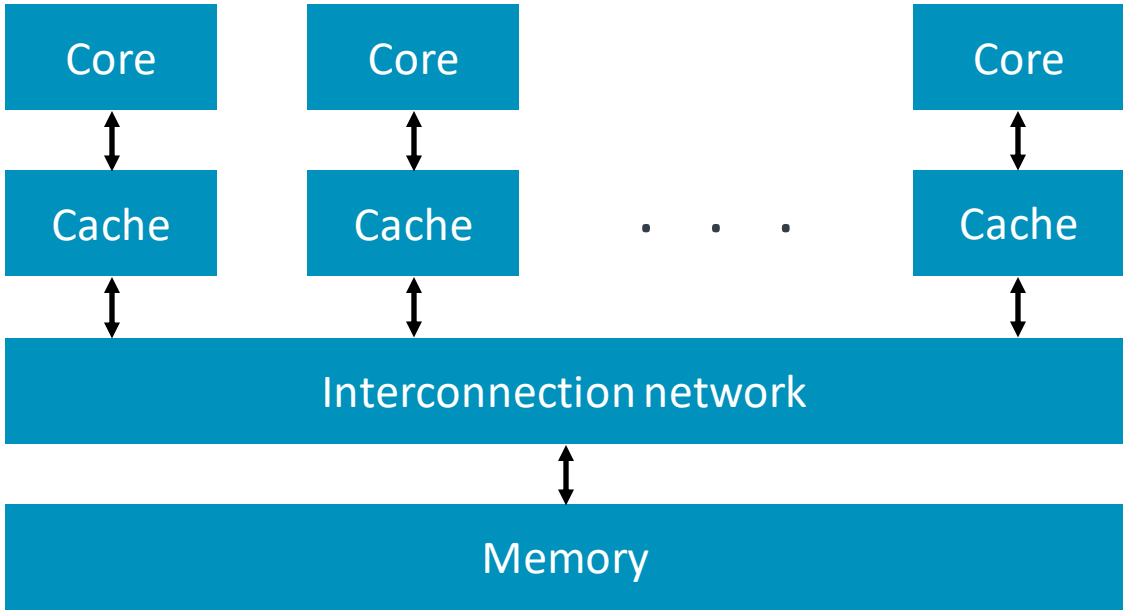
- Cores do not rely on a shared memory space.
- Each processing element (PE) has its core, data, I/O.
  - Explicit I/O to communicate with other cores
  - Synchronization via sending and receiving messages
- Advantages
  - Less hardware, easier to design
  - Focuses attention on costly non-local operations



arm

Shared Memory

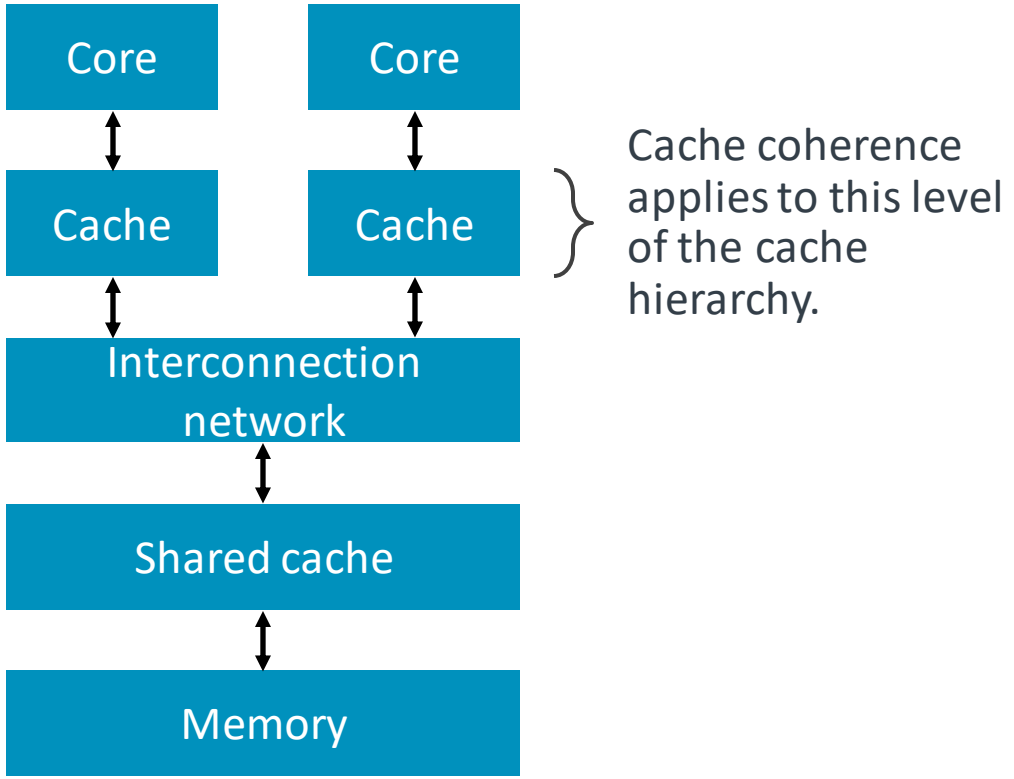
- Cores share a memory that they see as a single address space.
- Cores may have caches holding data.
  - Communication involves reading and writing to locations in memory.
  - Synchronization via atomic operations to modify memory
    - Specific instructions provided in the ISA
    - The hardware guarantees correct operation
- Advantages
  - Matches the programmer's view of the system
  - Hardware handles communication implicitly



arm

Shared Memory with Caches

- Caching shared-memory data has to be handled carefully.
  - The cache stores a copy of some of the data in memory.
- In particular, writes to the data must be dealt with correctly.
  - A core may write to data in its own cache.
  - This makes copies in other caches become stale.
  - The version in memory won't get updated immediately either, if it's a write-back cache.
  - In these situations, there is a danger of one core subsequently reading a stale (old) value.

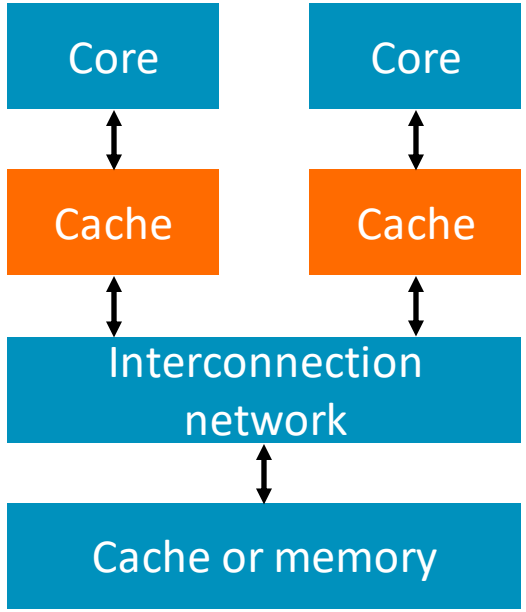


arm

Cache Coherence

Cache-coherence Protocol

- The cache-coherence protocol ensures that cores always read the most up-to-date values.
  - This means a core can always find the most recent value for some data, no matter where it is.
  - It describes the actions to take on seeing certain events from other cores.
- Cache coherence is required when caches share some memory.
- The protocols rely on caches seeing events from other cores.
  - In particular, reads and writes to the shared memory
  - Often this is realized through snooping operations on the interconnect.
- The protocol runs on each block in the cache independently.
  - This is the granularity commercial implementations track the state information.
- It runs in the private caches that have a shared ancestor.
  - The orange caches in the diagram



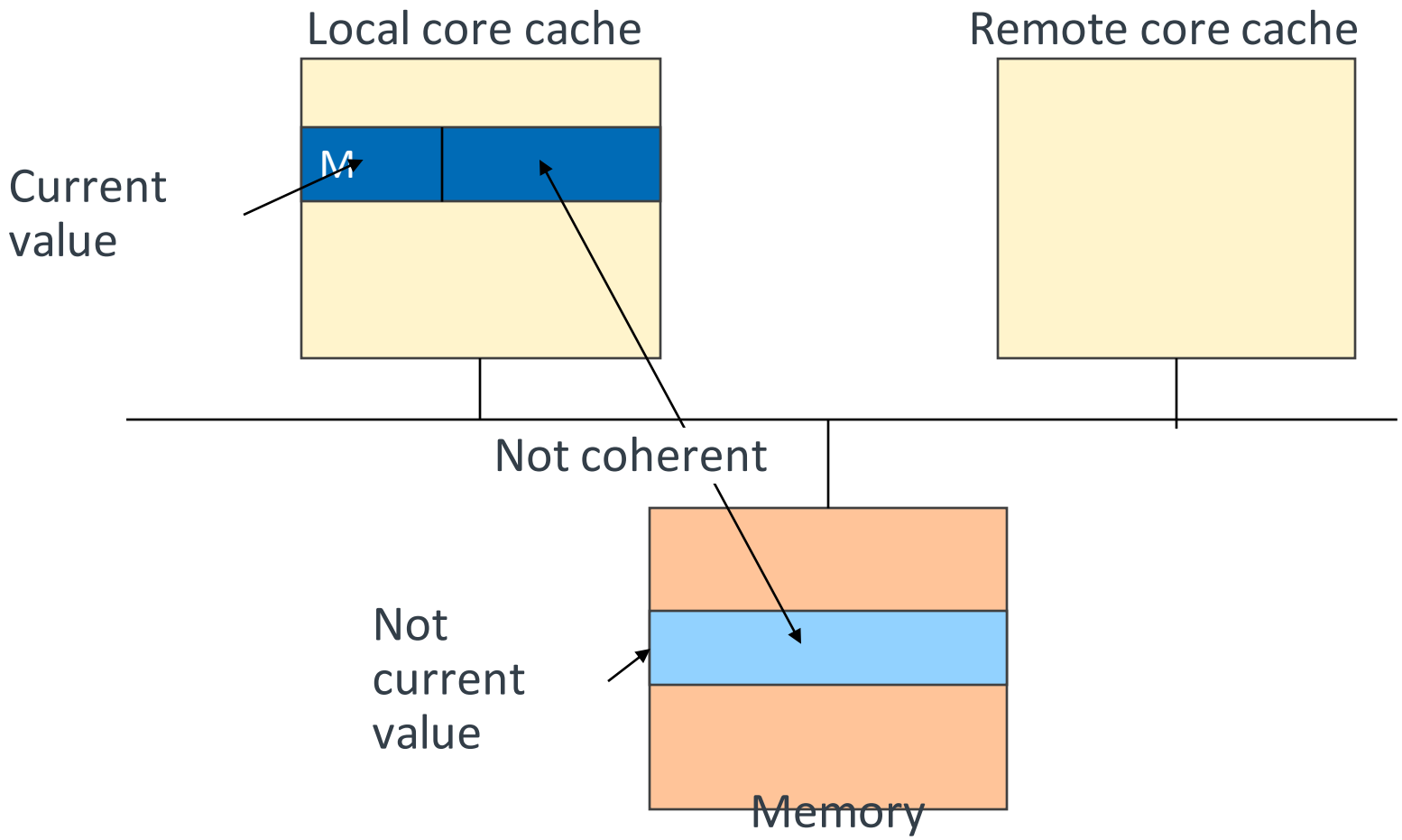
arm

MESI Cache-coherence Protocol

- The MESI protocol is a write-invalidate cache-coherence protocol.
  - When writing to a shared location, the related cache block is invalidated in the caches of all other cores.
- This protocol can manage cache coherence for a specified memory area.
- In general, uses an allocate-on-write cache policy
  - New data are loaded into the cache on both read and write misses.
- In general, uses a write-back cache
  - So caches can store data that are more recent than the value in memory.
- The MESI protocol defines states for each cache block and transitions between them.
  - Four states, corresponding to M, E, S, and I

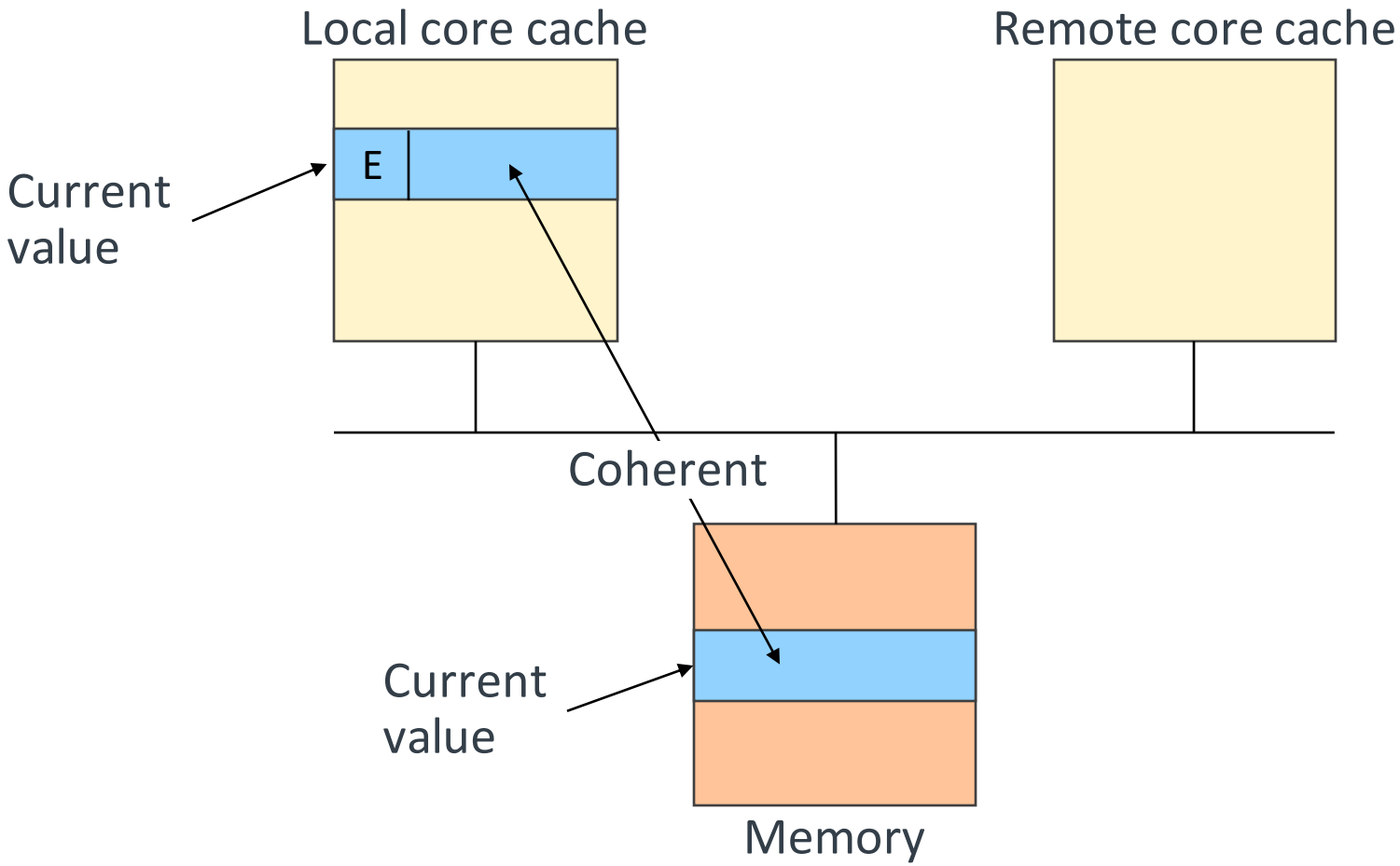
Modified State (M)

- The local cache holds the only copy of the block, which is also the most recent version of the data; memory holds old (stale) data.
- Note that memory may actually be a shared cache between the core's caches and main memory.



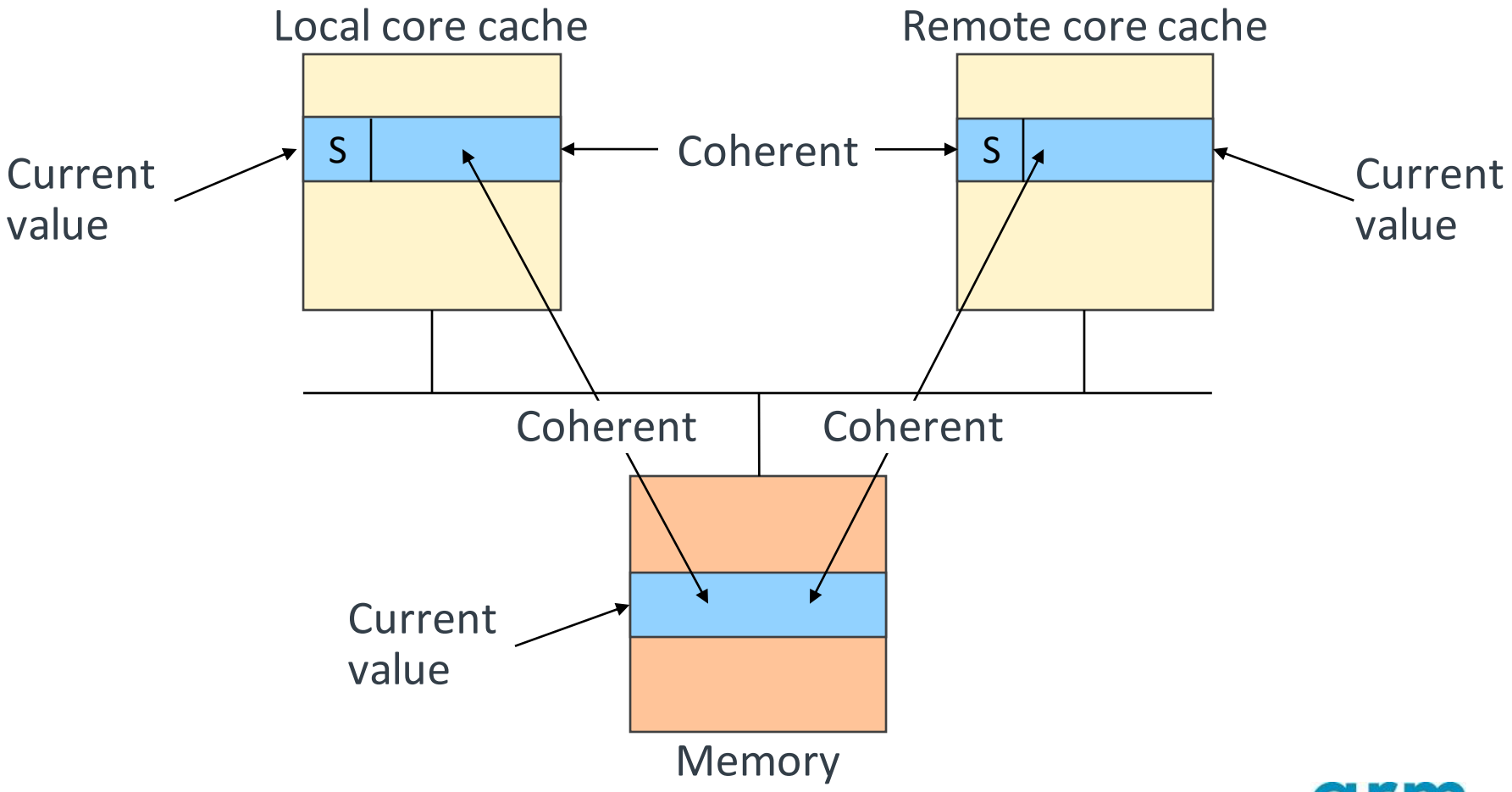
Exclusive State (E)

- The local cache holds the only copy of the block, which is identical to memory's version.



Shared State (S)

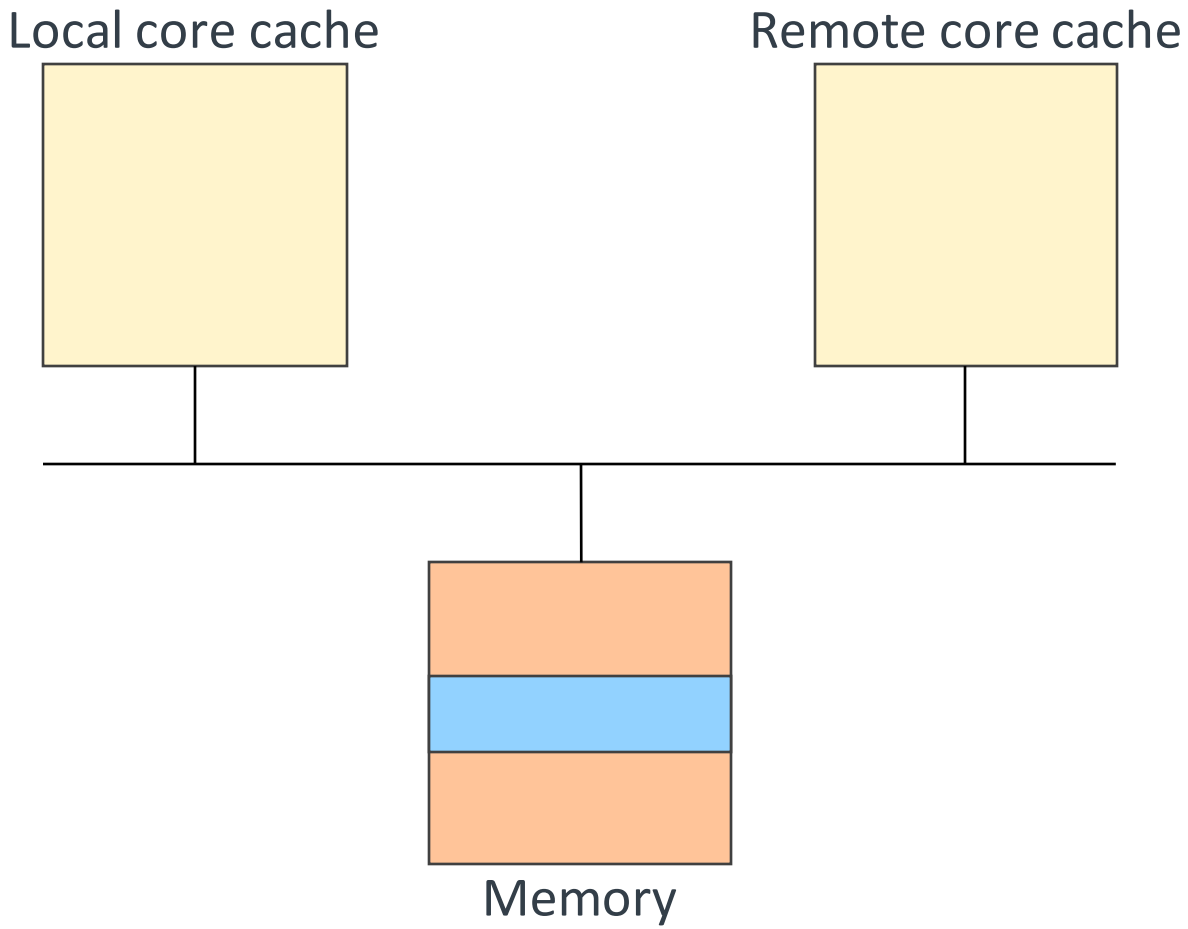
- The local cache holds a copy of the block, which is identical to memory's version; other caches may also hold the block in shared state.



arm

Invalid State (I)

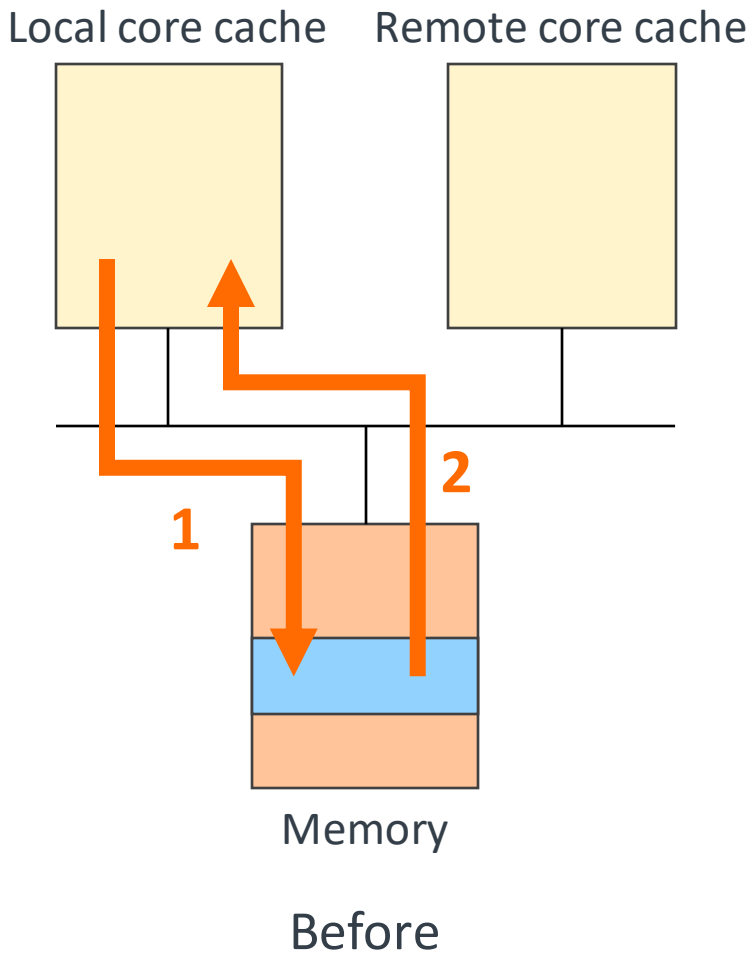
- The local cache does not hold a copy of the block.
  - This state may not be marked in the cache for each block; it could be inferred by a cache miss on that block.

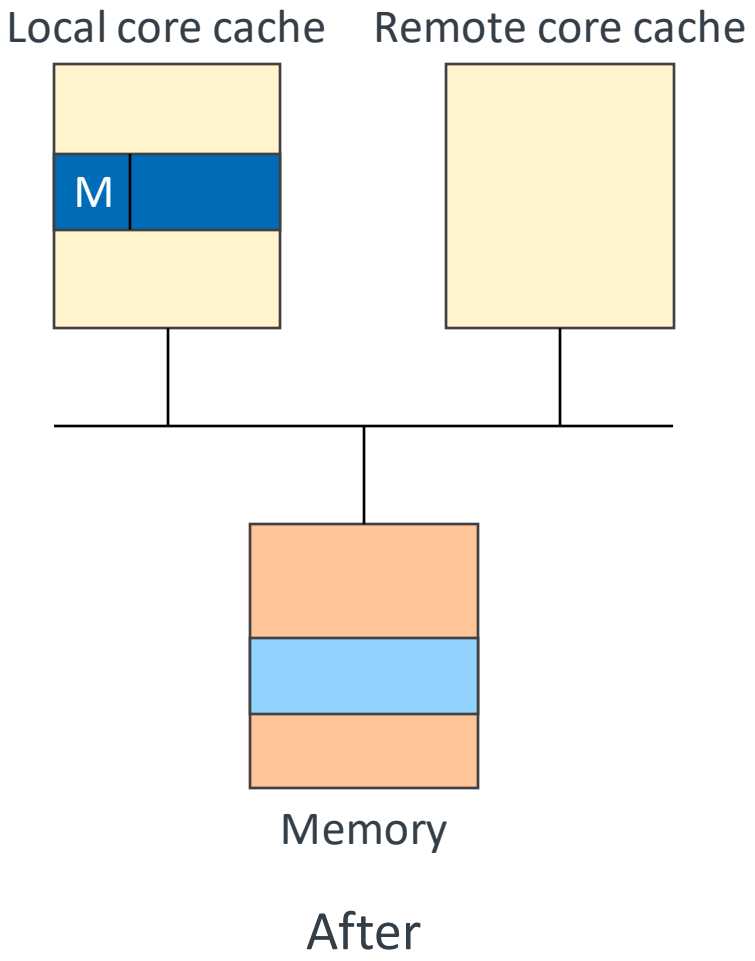


Invalid to Modified

- Occurs when the local core attempts to write some data to an address not already in the cache

1. Read-exclusive request
2. Data response

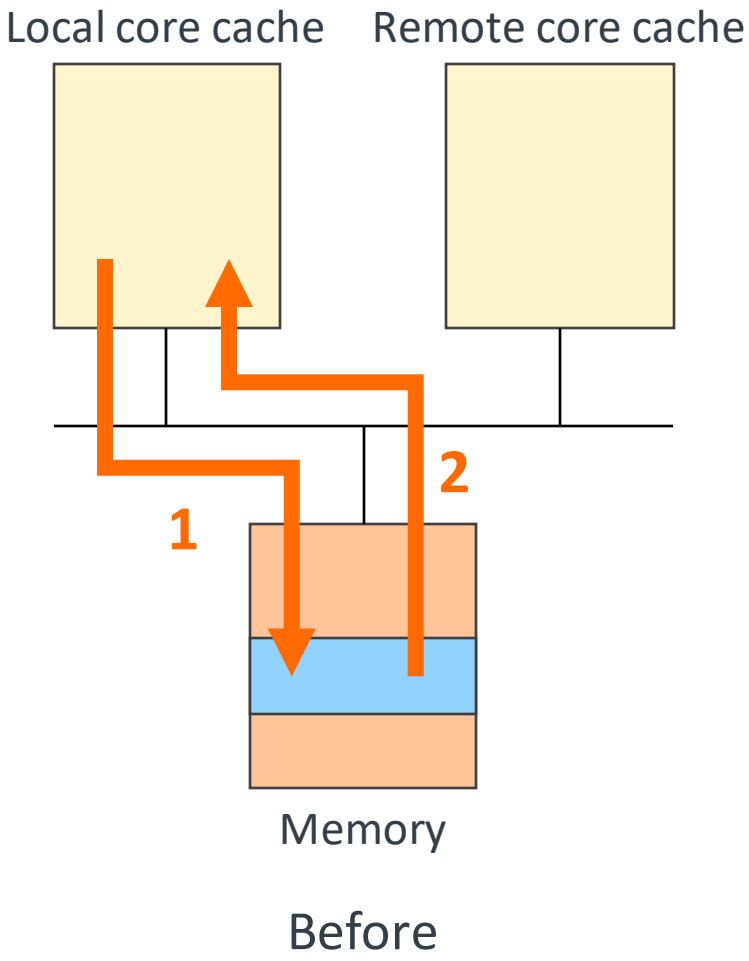


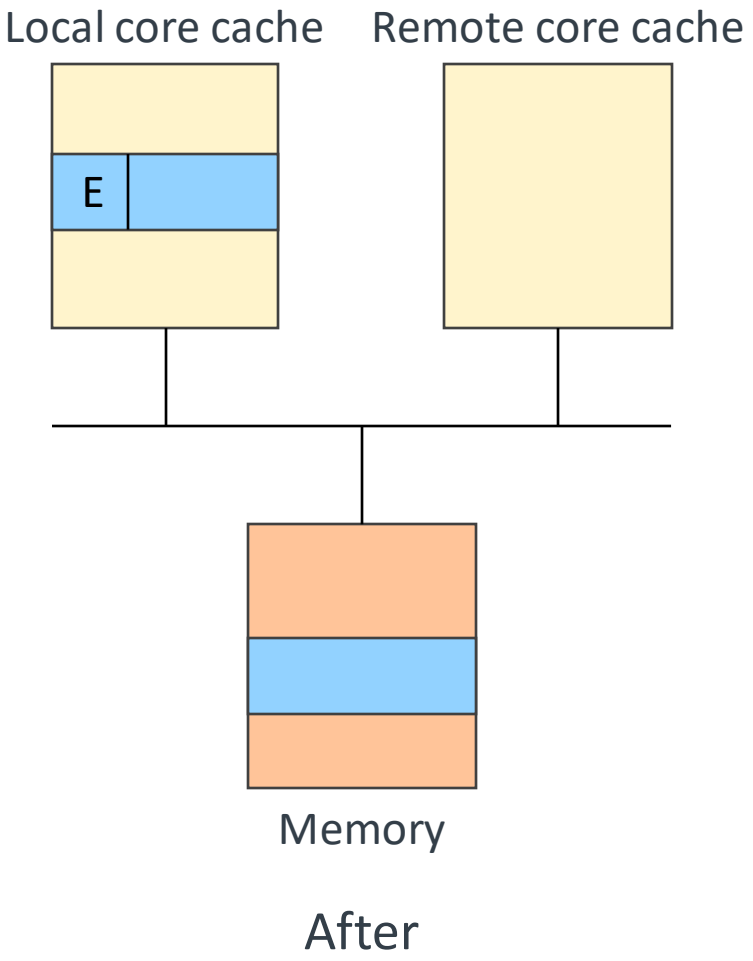


Invalid to Exclusive

- Occurs when the core attempts to read data from an address that is not already in the cache and no other cache has it

1. Read request
2. Data response

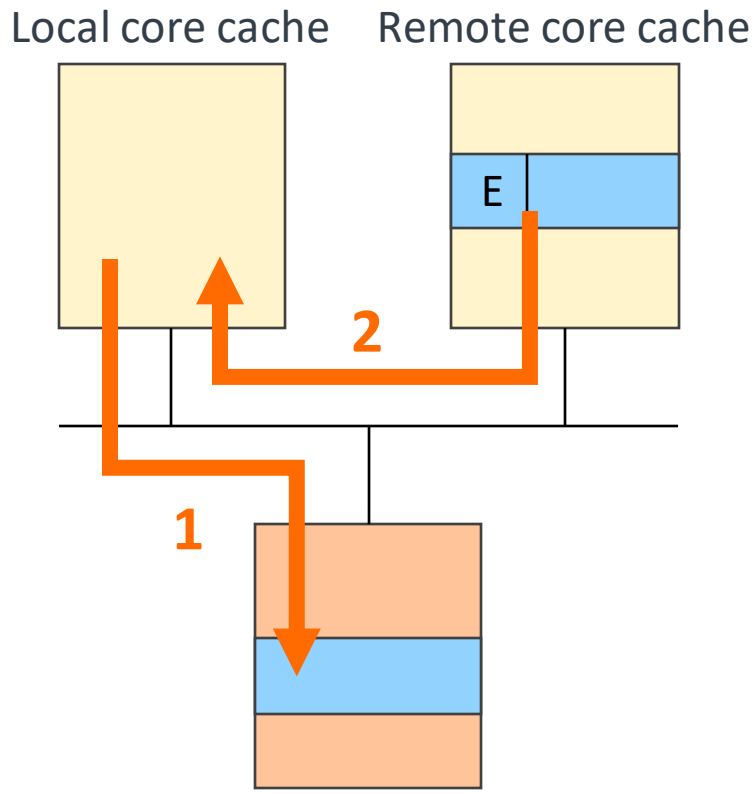




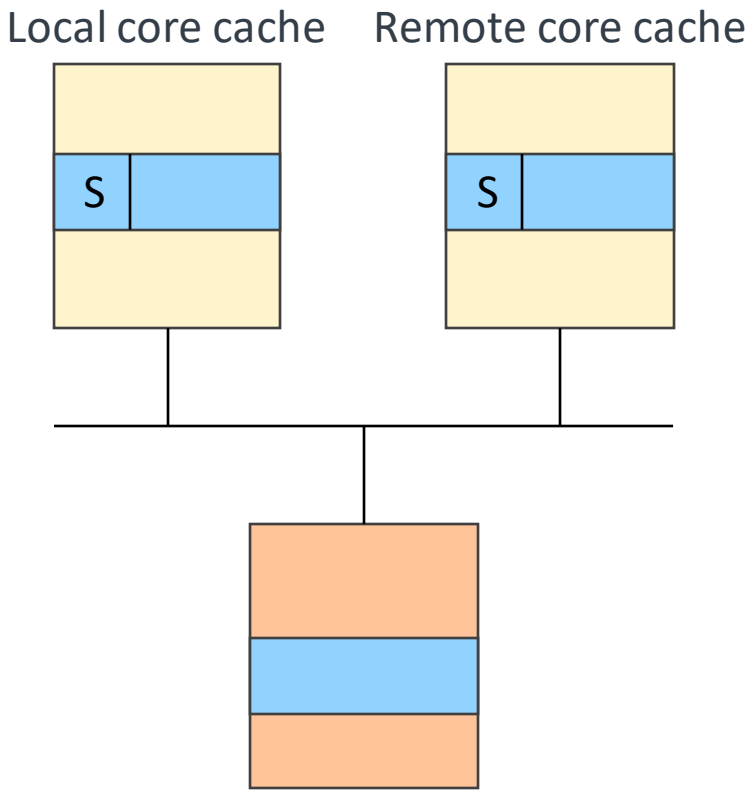
Invalid to Shared

- Occurs when the local core attempts to read data from an address that is not already in the cache, but other caches have a copy
- Data is supplied by another cache.

1. Read request
2. Data response



Before

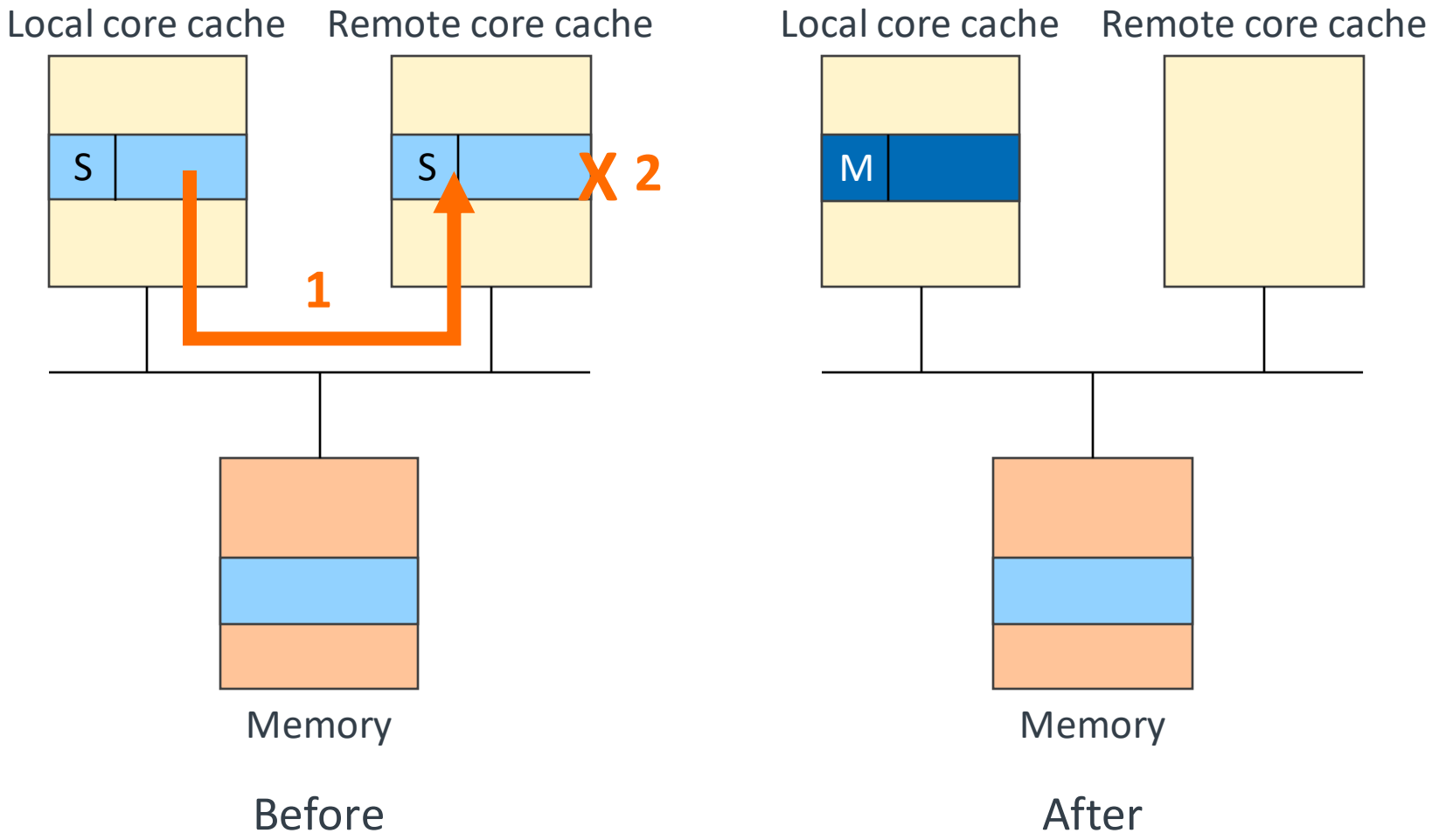


After

Shared to Modified

- Occurs when the local core attempts to write some data to an address that is already in the cache, and other caches may have a copy
- Needs to invalidate other copies

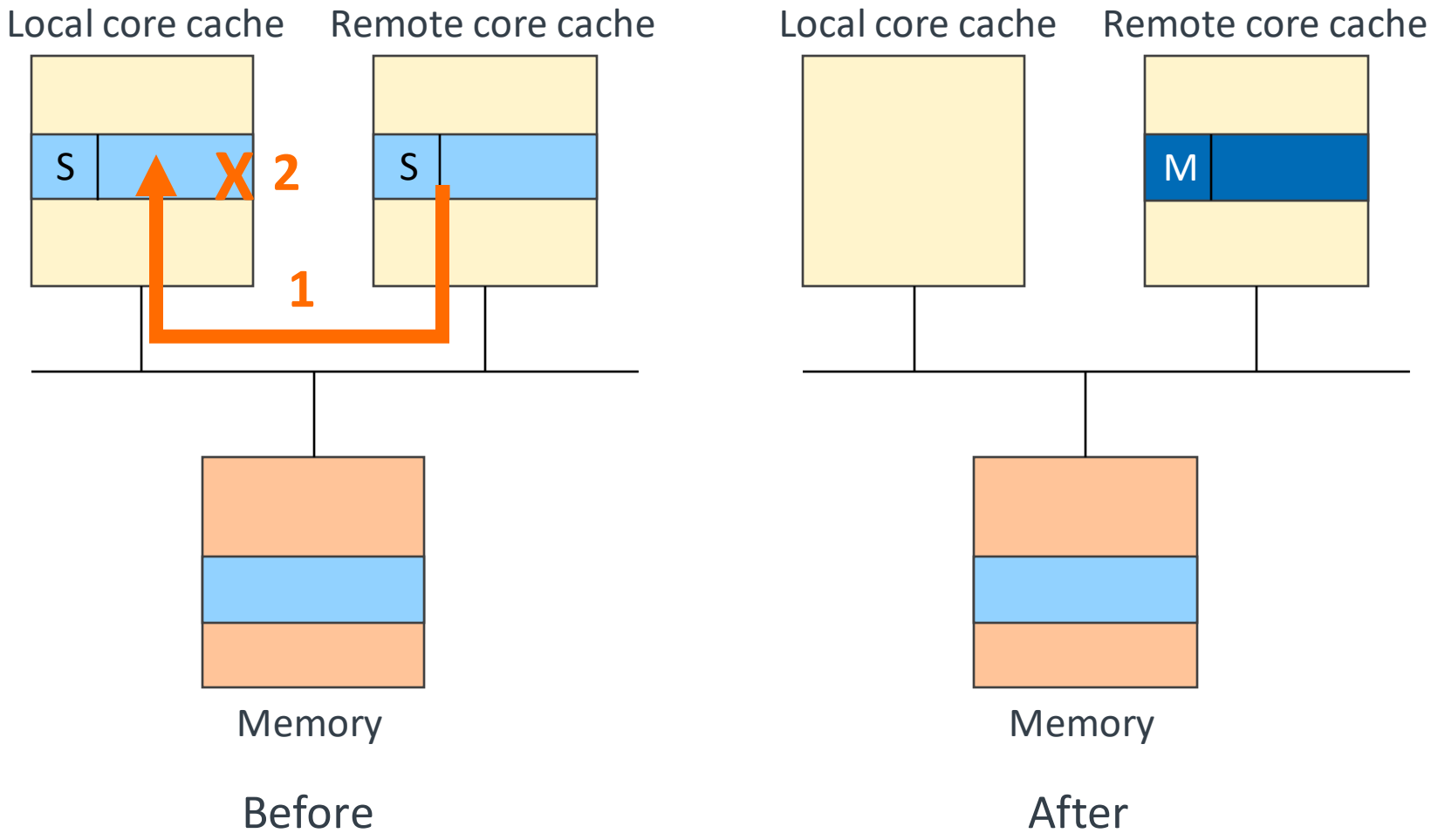
1. Read-exclusive request
2. Invalidate



Shared to Invalid

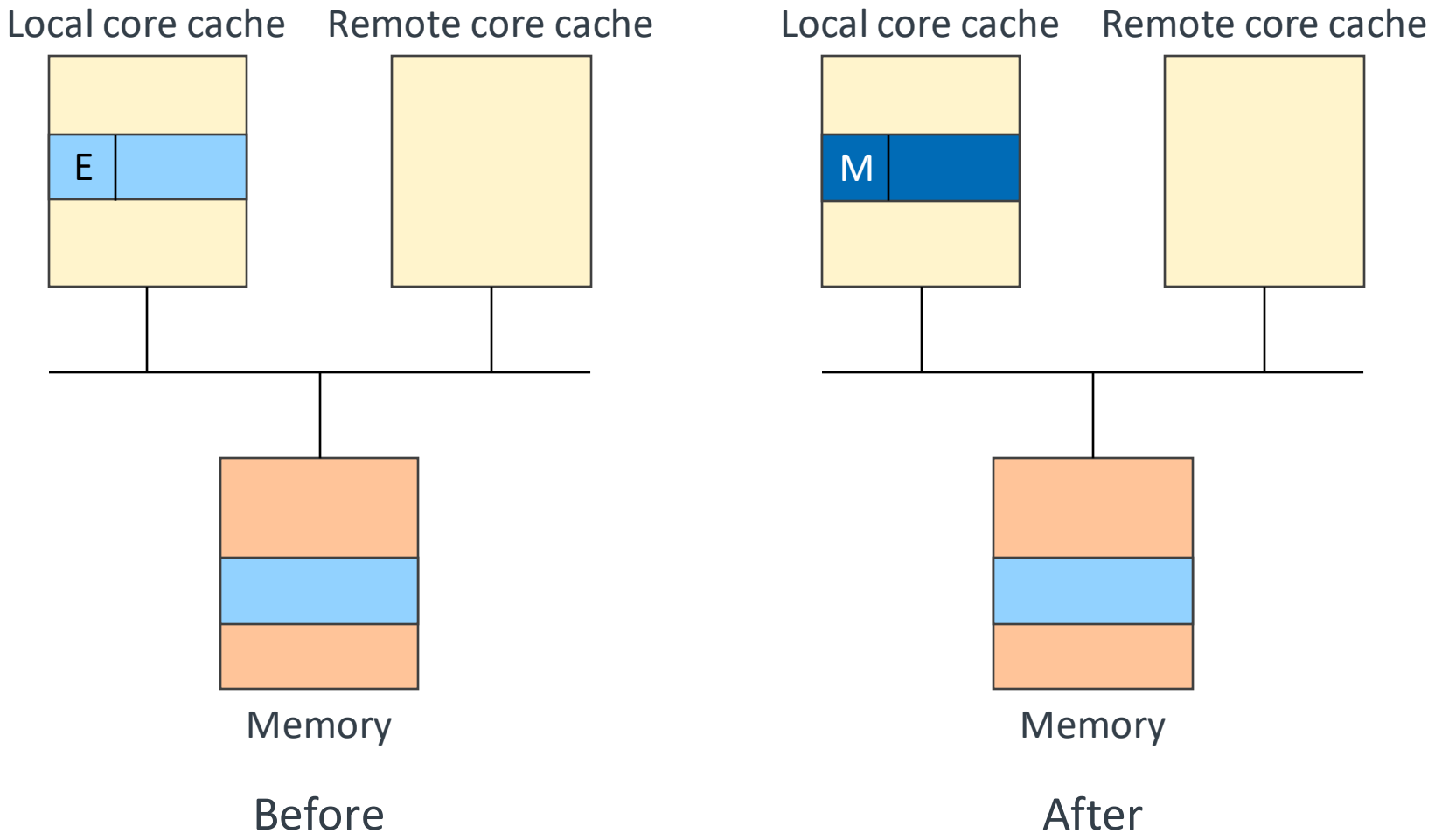
- Occurs when another core attempts to write some data to an address that is already in the cache
- The local cache snoops the exclusive read request.

1. Read-exclusive request
2. Invalidate



Exclusive to Modified

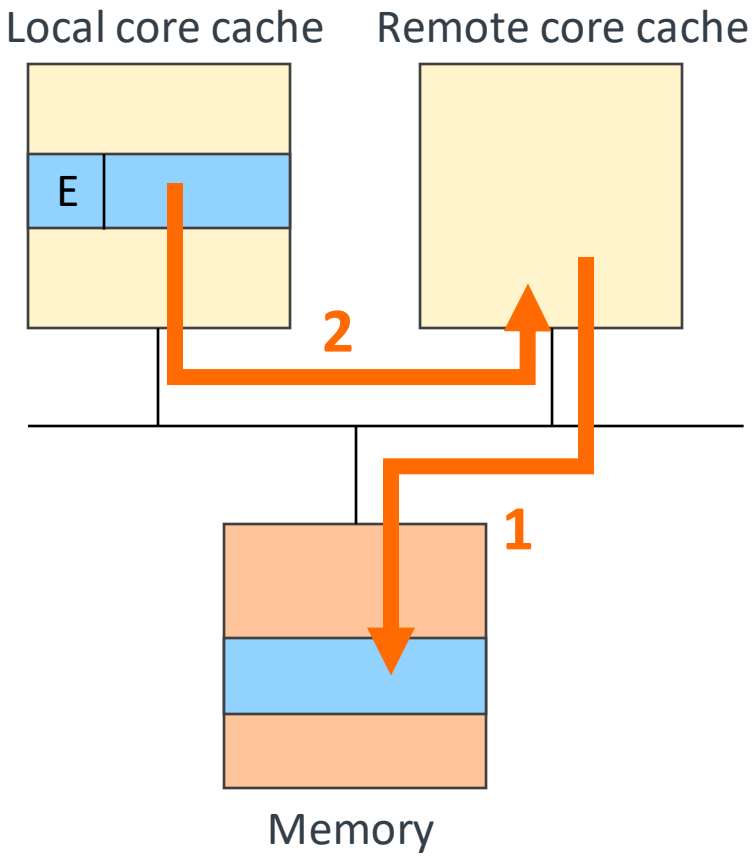
- Occurs when the local core attempts to write some data to an address that is already in the cache, and that's the only copy
- No need to invalidate other caches because we know they don't have a copy



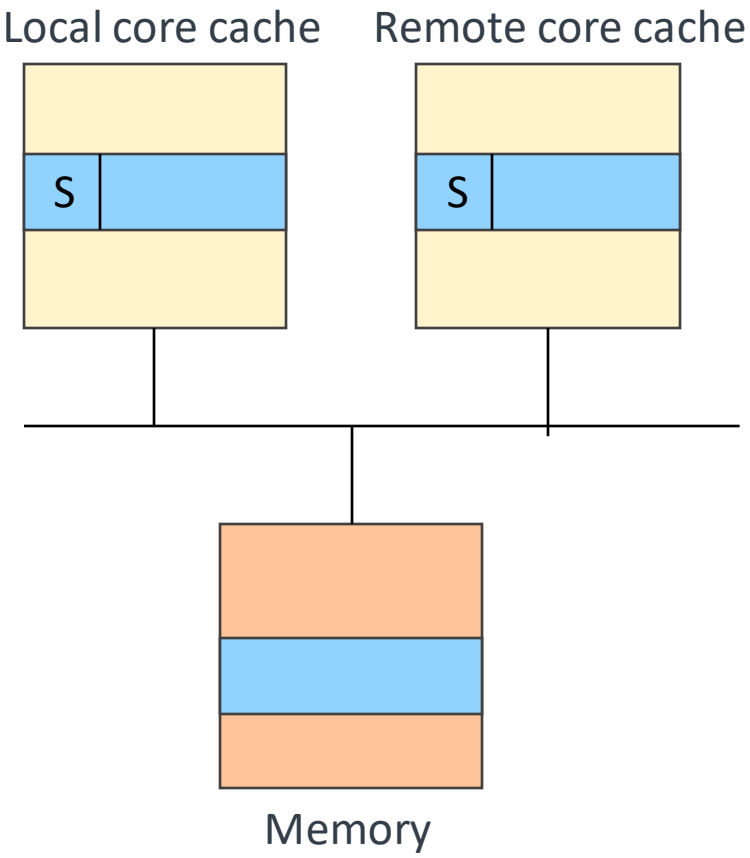
Exclusive to Shared

- Occurs when another core attempts to read data from an address that this cache has, and it's the only copy
- Data are supplied by the cache after snooping the read request.

1. Read request
2. Data response



Before

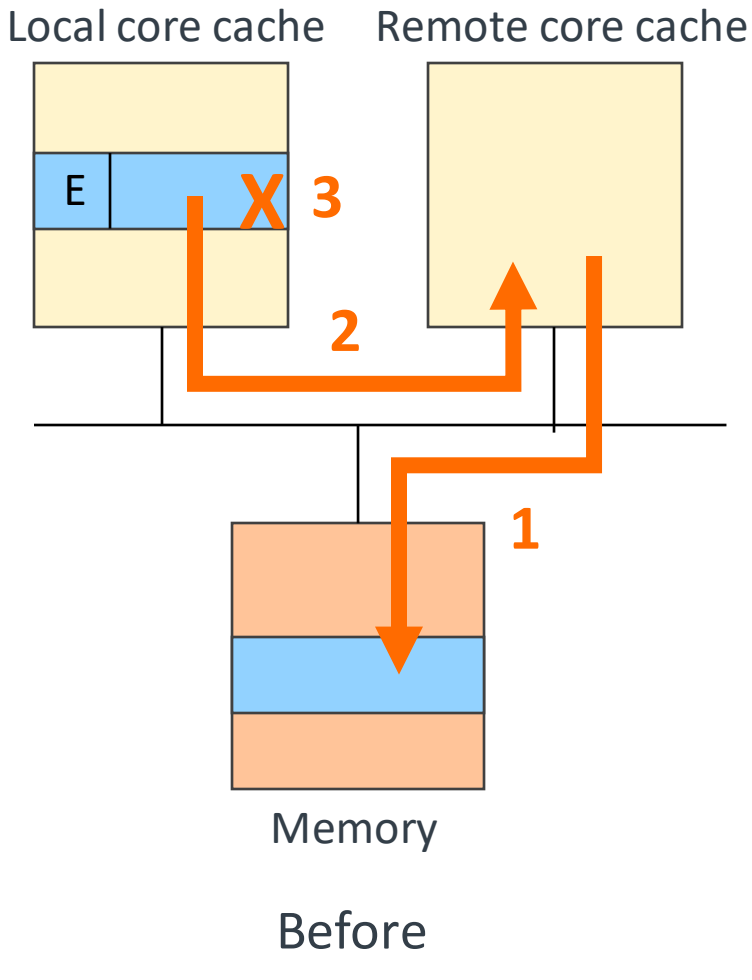


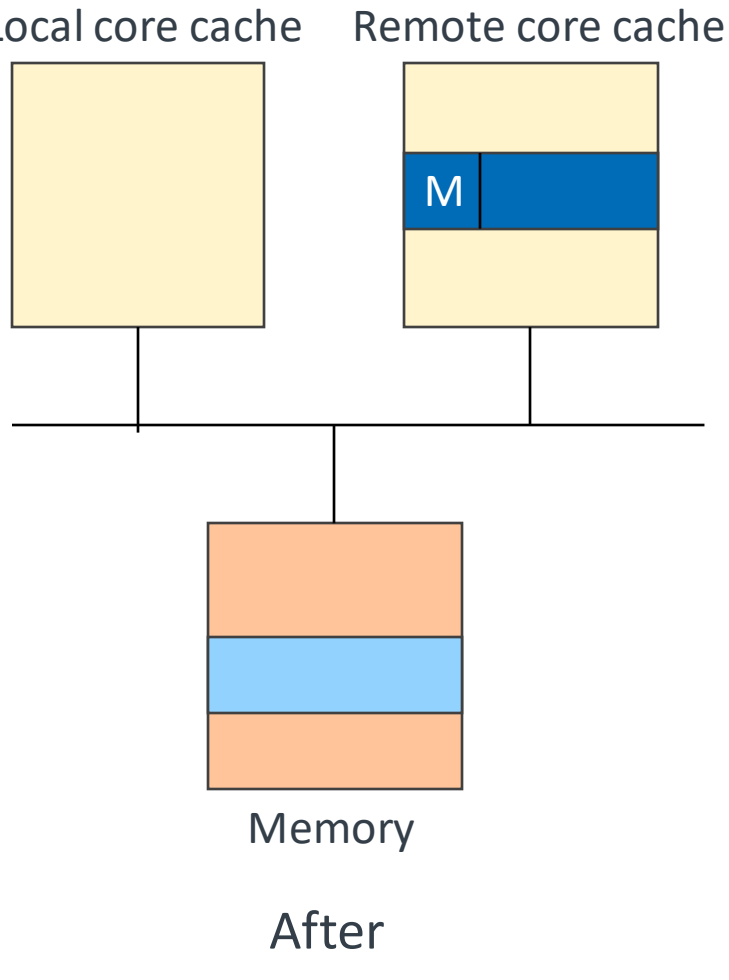
After

Exclusive to Invalid

- Occurs when another core attempts to write some data to an address that this cache has the only copy of
- The local cache snoops the exclusive read request.

1. Read-exclusive request
2. Data response
3. Invalidate

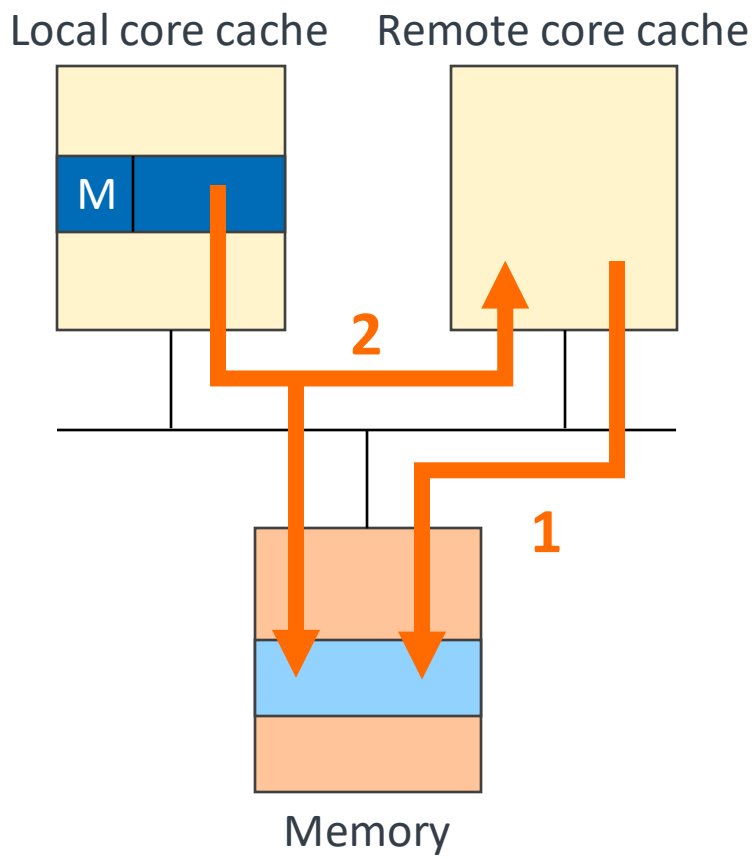


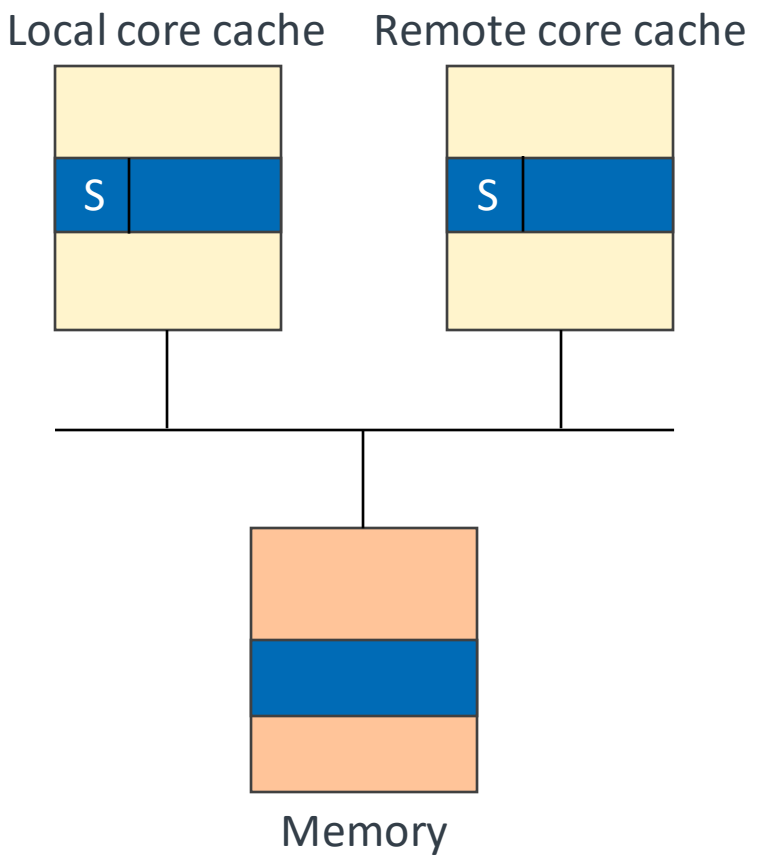


Modified to Shared

- Occurs when another core attempts to read data from an address that this cache has written to
- Must flush the data back to memory and the requesting cache

1. Read request
2. Data response

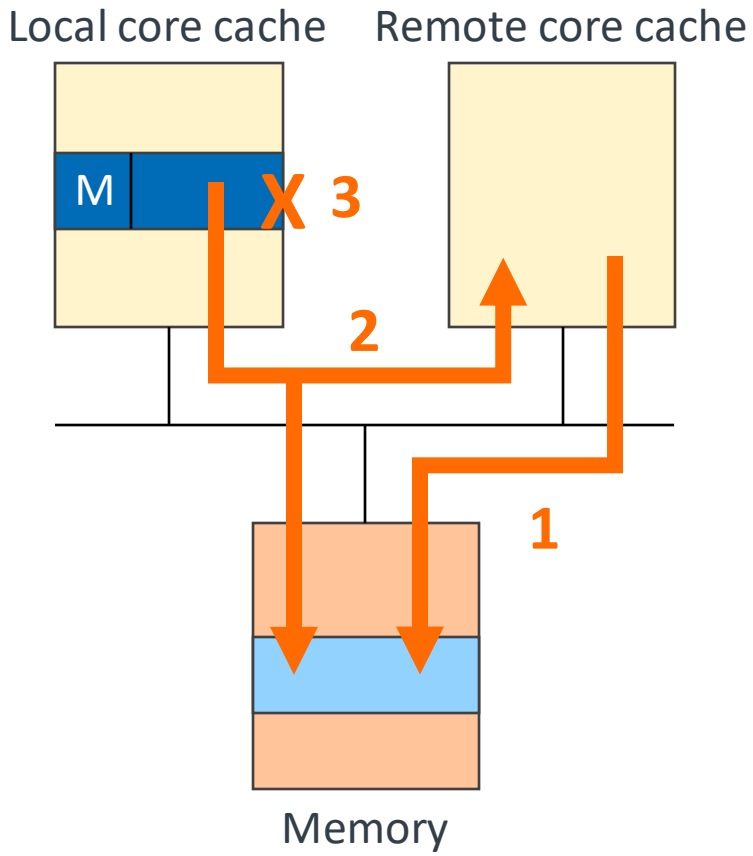


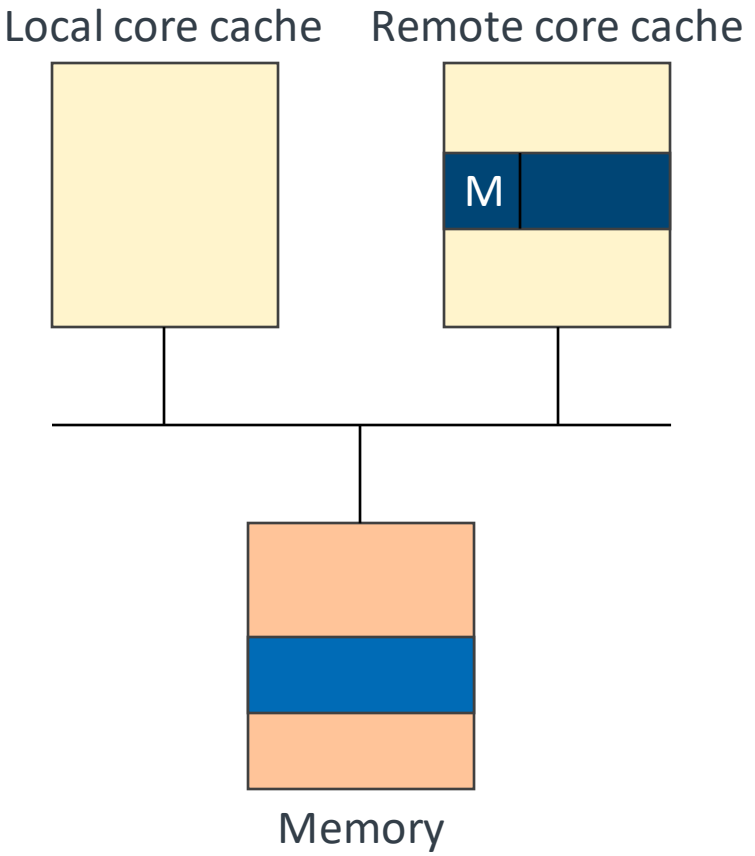


Modified to Invalid

- Occurs when another core attempts to write some data to an address that this cache has altered
- Must flush the data back to memory and the requesting cache

1. Read-exclusive request
2. Data response
3. Invalidate



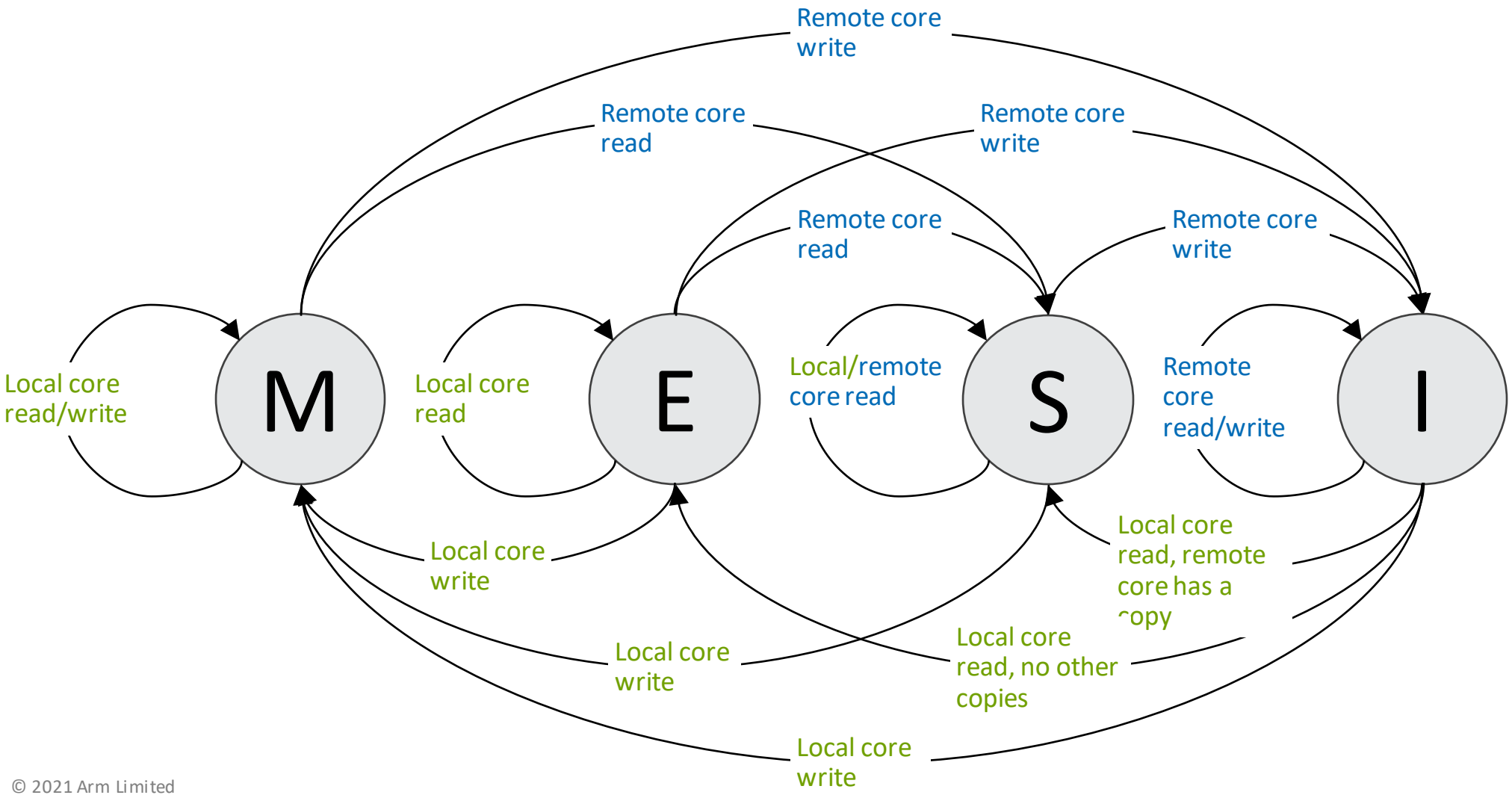


Visualizing the Protocol

- We can visualize the protocol in two ways.
- First, a table showing the valid combinations of states that two caches can have for the same block
  - For example, two caches can have it in shared.
  - But if any cache has it in exclusive or modified, then all other caches are invalid.
- Second, we can draw a state transition diagram to show all events and actions.

	M	E	S	I
M	×	×	×	✓
E	×	×	×	✓
S	×	×	✓	✓
I	✓	✓	✓	✓

MESI State-transition Diagram



arm

Memory Consistency

Memory Consistency

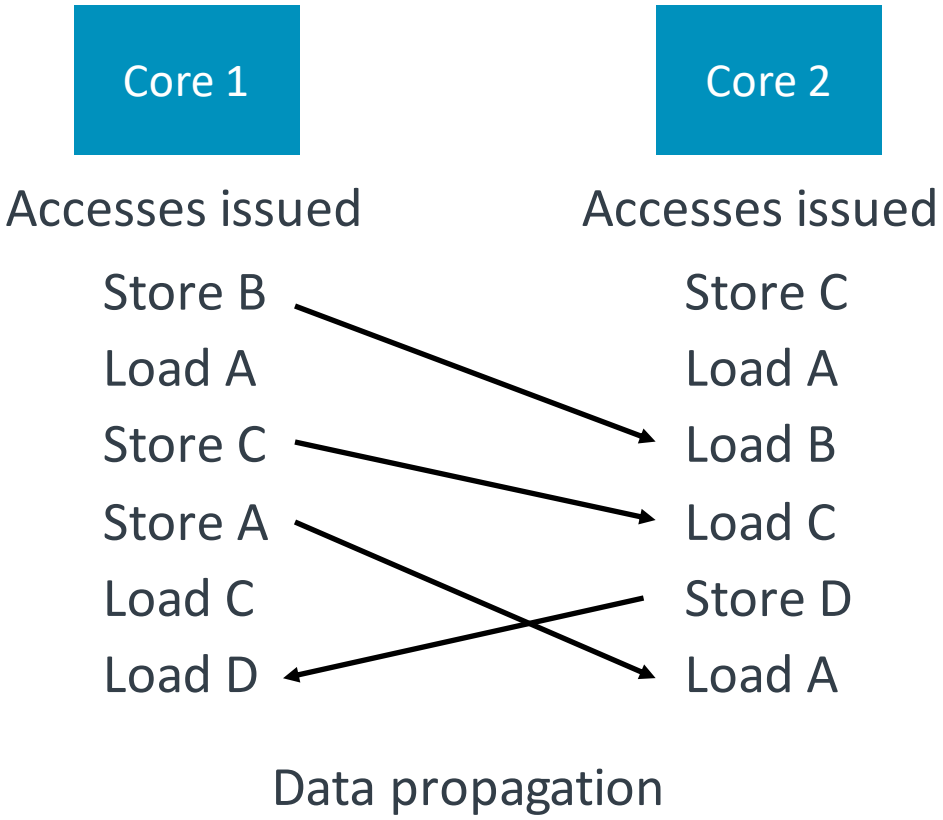
- The purpose of cache coherence is to ensure data propagation and coherence.
  - So when data are written by one core, all other cores can later read the correct value.
  - When a core attempts to write, others know that their copies are stale.
  - When a core attempts to read, others know they must provide their data, if modified.
- The cache-coherence protocol is run independently on each block of data.
  - There is no direct interaction between different blocks, as far as the protocol is concerned.
- So what about the order in which data accesses by one core are seen by others?
  - If a core performs certain reads and writes to different data, in what order do other cores see them?
- It is the purpose of the memory-consistency model to define this.
  - And the job of the memory hierarchy (and core) to implement it

Memory Consistency

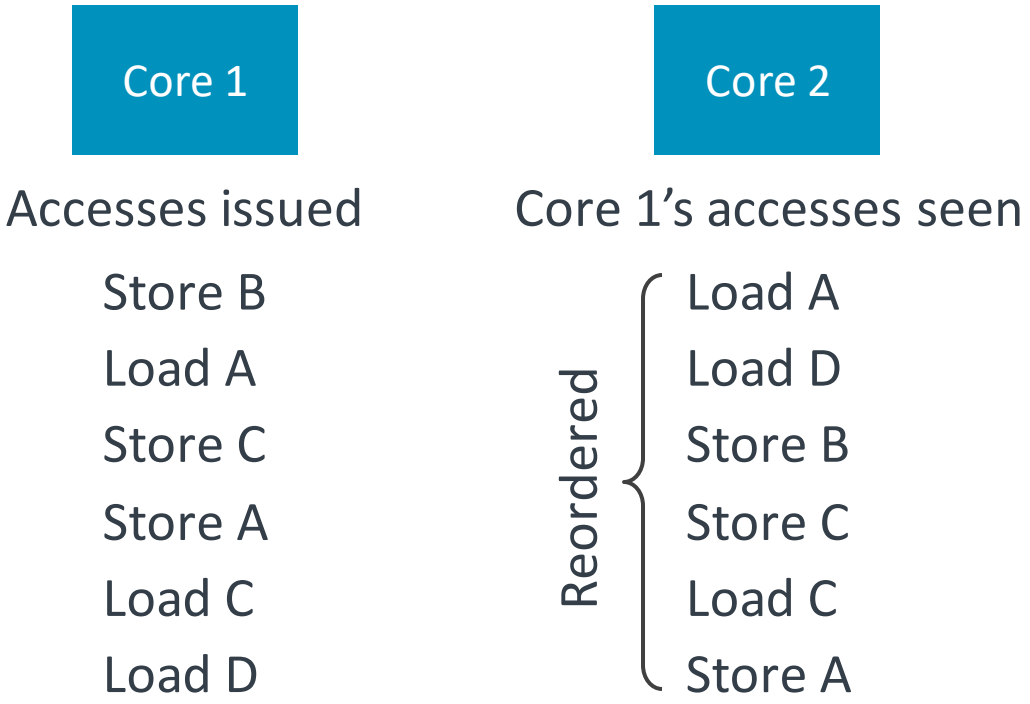
- Modern processors may reorder memory operations.
- Out-of-order processing can allow loads to access the cache ahead of older stores.
  - Either because the addresses they access don’t match
  - Or because the load has been speculatively executed and will be replayed later if a dependence is found
- This avoids stalling loads unnecessarily, even though their effects can be seen externally.
  - By other cores in the system
- Recall module 5 where this was introduced.

Memory Consistency vs Cache Coherence

Cache coherence



Memory consistency



arm

Memory Consistency

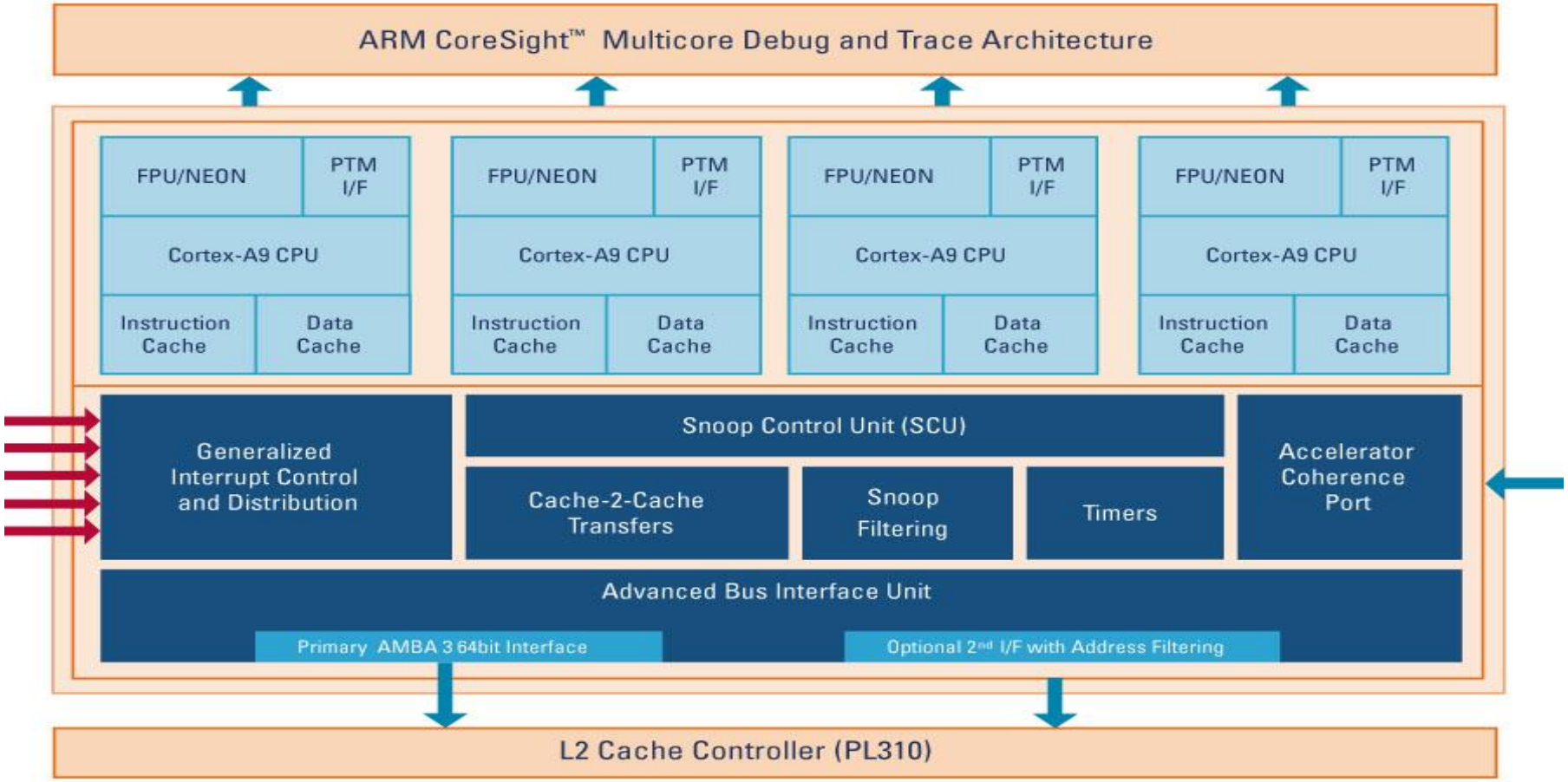
- The memory consistency model defines valid outcomes of sequences of accesses of the different cores.
- Sequential consistency (SC) is the strongest and most intuitive model.
  - The operations of each core occur in program order, and these are interleaved (at some granularity) across all cores.
  - This means that no loads or stores can bypass other loads or stores.
  - SC is overly strong because it prevents many useful optimizations without being needed by most programs.
- Total store order (TSO) is widely implemented (e.g., x86 architectures).
  - The same as sequential consistency but allows a younger load to observe a state of memory in which the effects of an older store have not yet become observable
- Forms of relaxed consistency have been adopted (e.g., Arm and PowerPC architectures).
  - In more relaxed consistency models, other constraints in SC are removed, such as a younger load observing a state of memory before an older load does.

Case Study: Cortex-A9 MPCore

Contains up to four Cortex-A9 processors

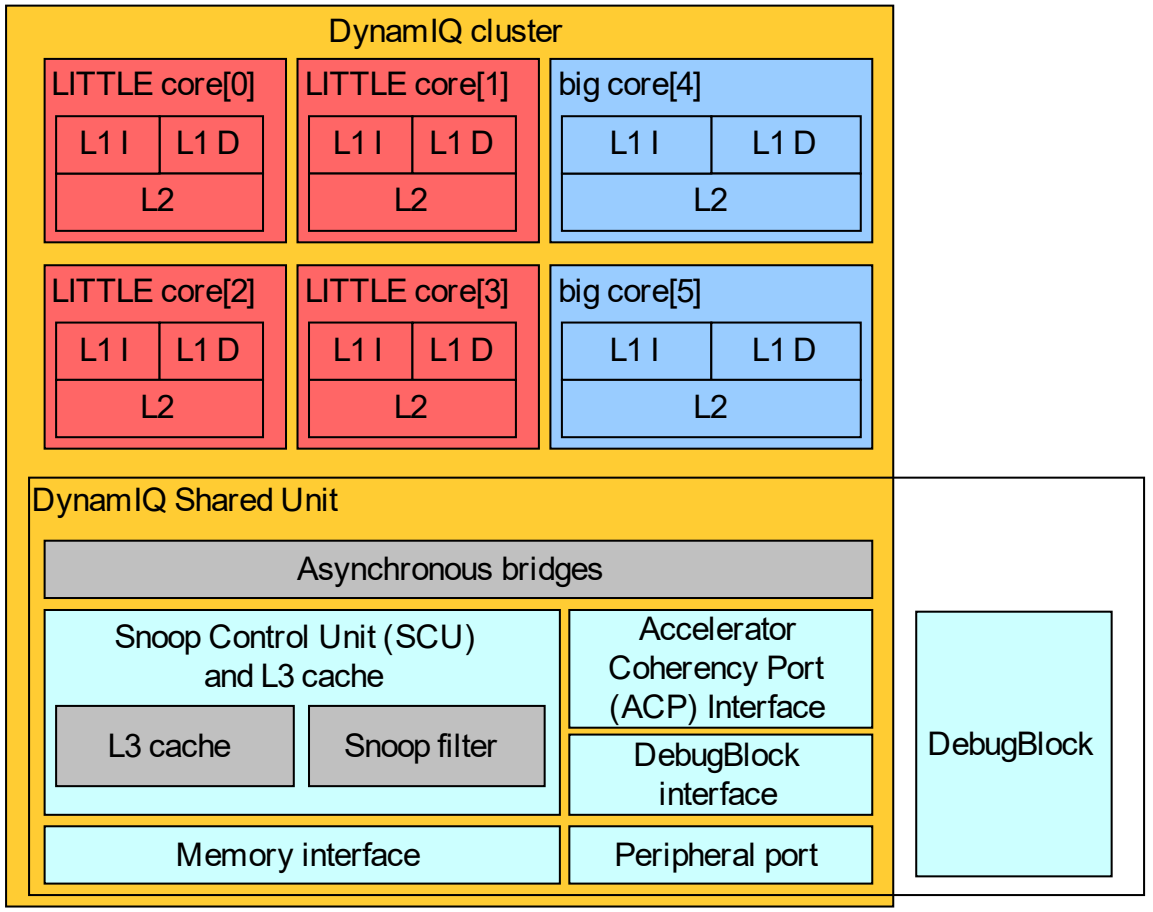
Snoop Control Unit

- Maintains L1 data cache coherency across processors
- Arbitrates accesses to the L2 memory system, through one or two external 64-bit AXI manager interfaces



Case Study: Heterogeneous Multicore

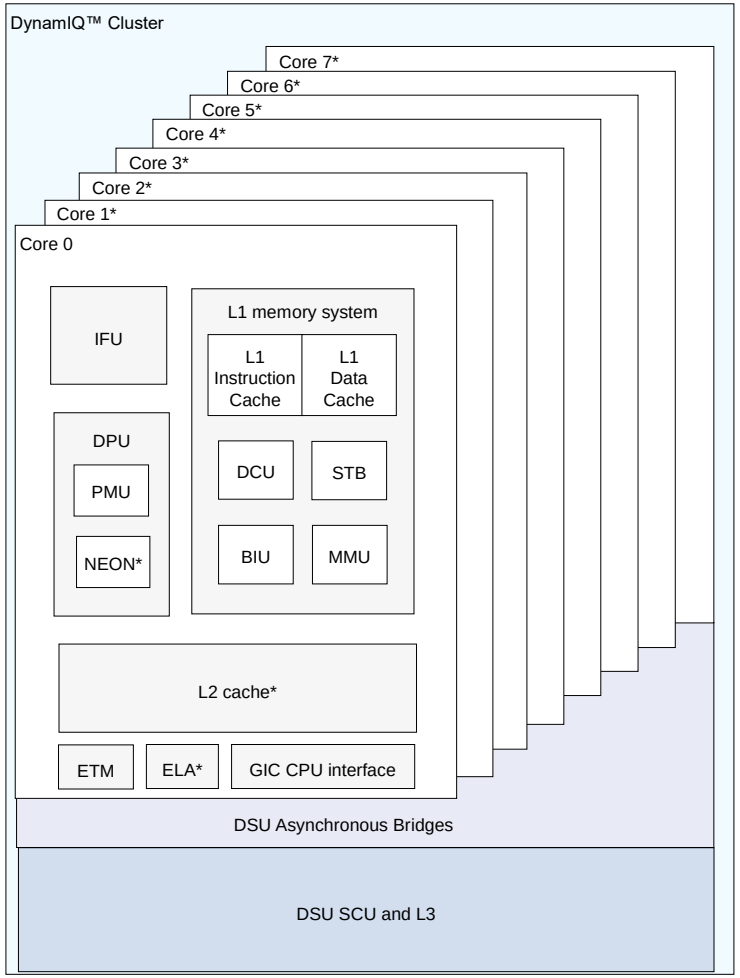
- big.LITTLE is a heterogeneous processing architecture with two types of cores.
  - “big” cores for high compute performance
  - “LITTLE” cores for power efficiency
  - L1 and L2 memory system in cores
- A DynamIQ system contains big and LITTLE cores and a shared unit containing.
  - L3 memory system
  - Control logic
  - External interfaces



arm

Case Study: Cortex-A55

- The Cortex-A55 is a LITTLE core.
- Optionally contains an L2 cache
- Uses the MESI protocol for coherence
  - M: Modified/UniqueDirty (UD) – the line is in only this cache and is dirty.
  - E: Exclusive/UniqueClean (UC) – the line is in only this cache and is clean.
  - S: Shared/SharedClean (SC) – the line is possibly in other caches and is clean.
  - I: Invalid/Invalid (I) – the line is not in this cache.
- The data-cache unit (DCU) stores the MESI state of each cache line.



* Optional

Conclusions

- Multicore processors provide performance from increasing numbers of transistors.
  - Performance comes through thread-level parallelism.
- Shared-memory systems are the most common paradigm.
  - Cores share a memory and common address space.
  - Data written by one core are read by others when accessing the same location.
- Dealing with shared memory in the presence of caches poses a challenge.
  - This is where the cache-coherence protocol comes into play.
  - We looked at the MESI protocol, but there are other more simple and more complex protocols around.
- Memory consistency defines the order that reads/writes to different addresses are seen by the different cores in the system.