Cache Coherency and Multi-Core Programming

Christian Gyrling

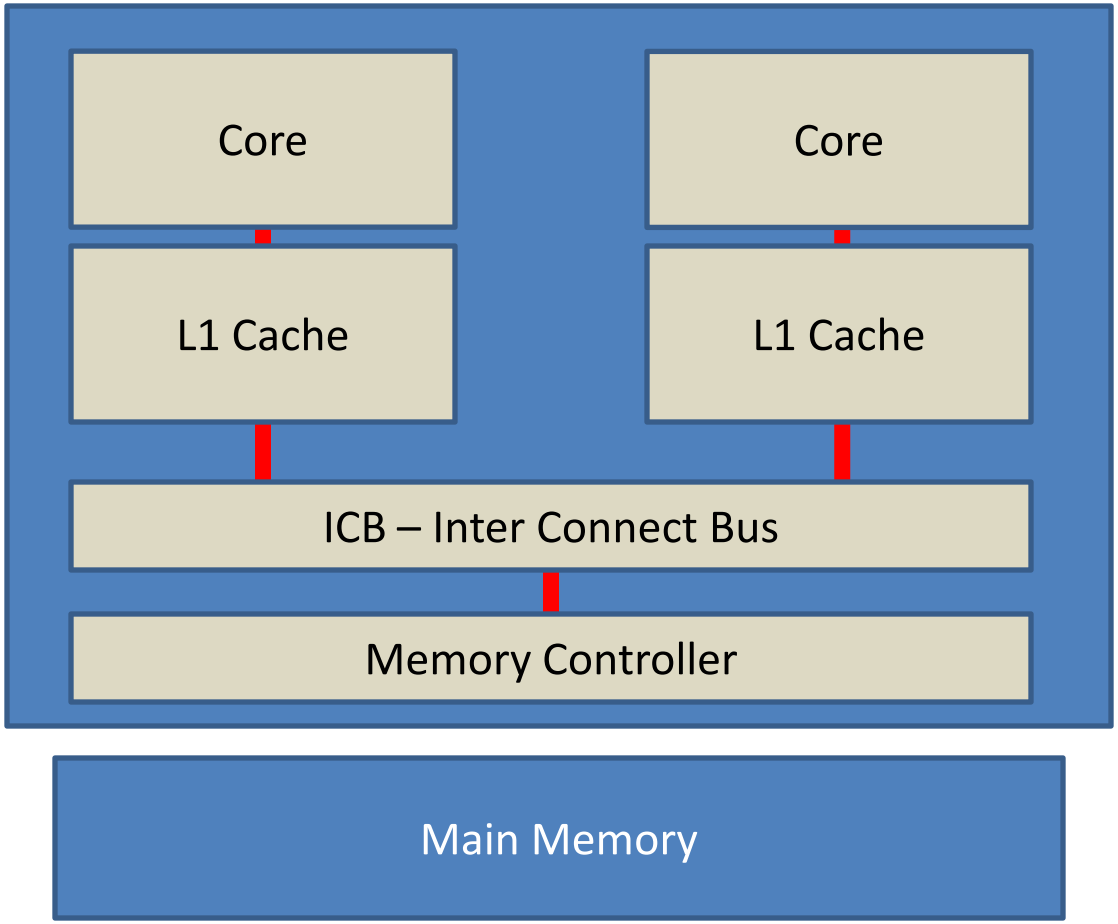
Naughty Dog

This is a ***<u>very</u>*** technical talk so this is the only fun slide. Enjoy it! ☺

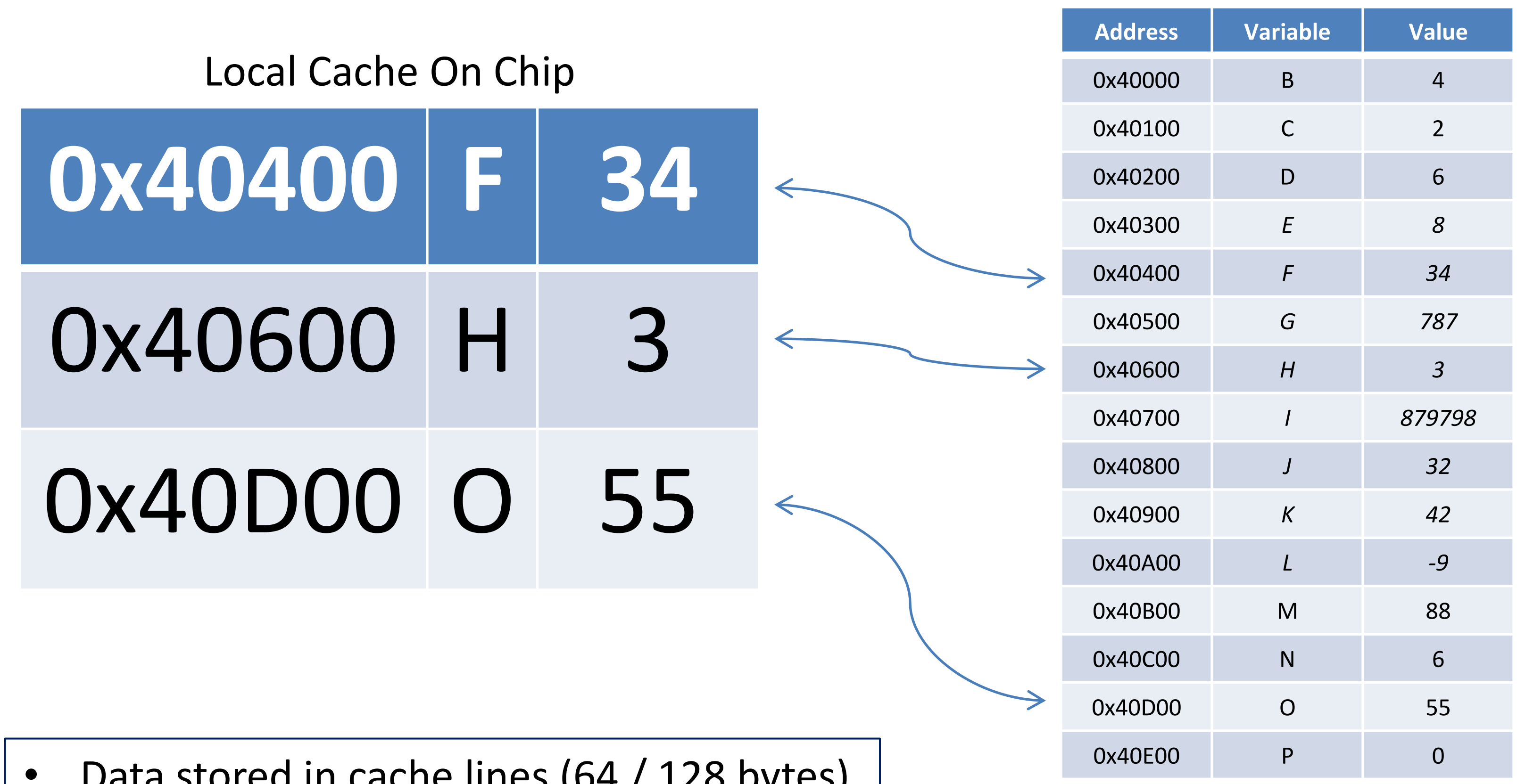
Questions

- How does the cache share data between cores?
- How does the data stay consistent when multiple cores are updating memory at the same time?

Simple 2-core CPU



Caching



Local Cache On Chip

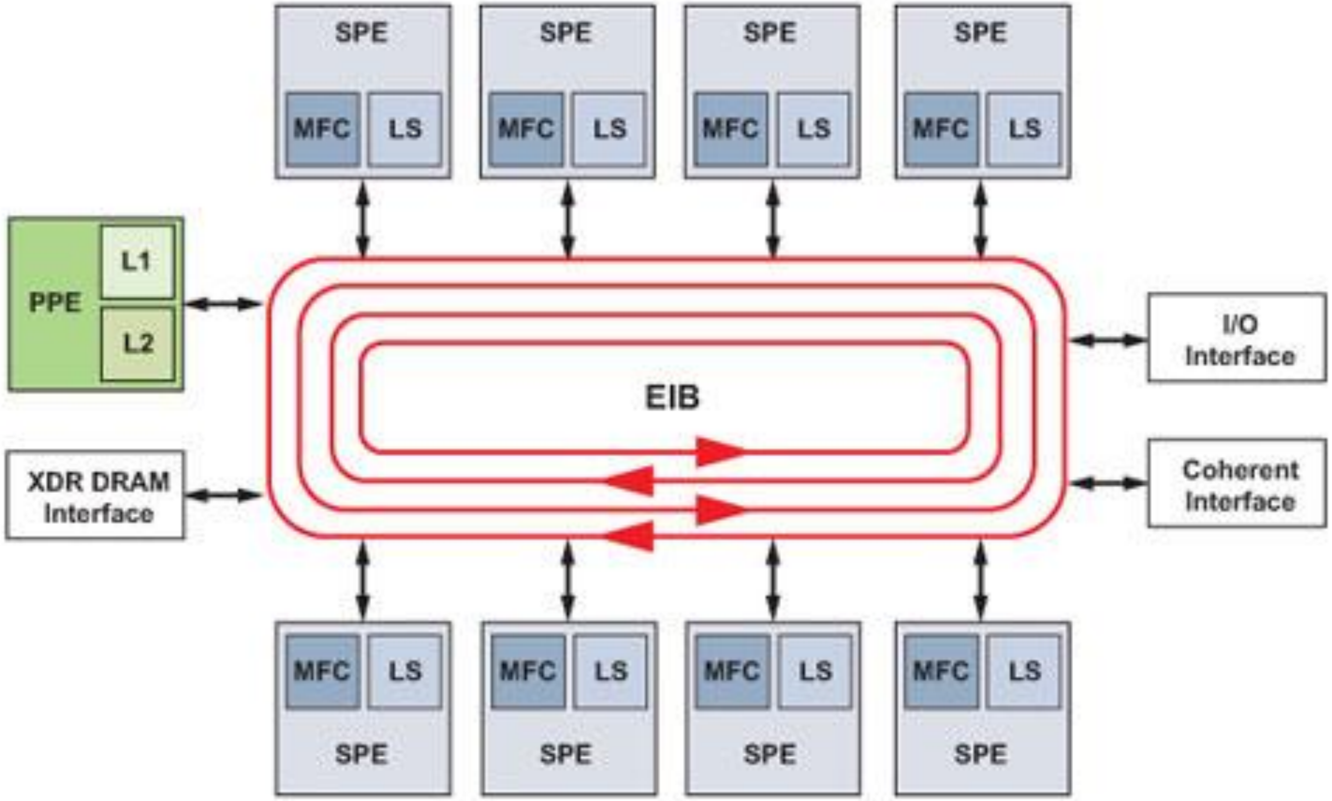
0x40400	F	34
0x40600	H	3
0x40D00	O	55

Main Memory

Address	Variable	Value
0x40000	B	4
0x40100	C	2
0x40200	D	6
0x40300	*E*	*8*
0x40400	*F*	*34*
0x40500	*G*	*787*
0x40600	*H*	*3*
0x40700	*I*	*879798*
0x40800	*J*	*32*
0x40900	*K*	*42*
0x40A00	*L*	*-9*
0x40B00	M	88
0x40C00	N	6
0x40D00	O	55
0x40E00	P	0

- Data stored in cache lines (64 / 128 bytes)
- Fast access to recently used cache lines

Memory Is Far Away



ICB – Inter Connect Bus

- Connects cores
- Not just data
- Cache coherence protocol
- “Cache coherence domain”
  - Usually all processors and all cores

The MESI Protocol

- Cache Coherence
- Any given cache line can only be modified by one core at a time.
- A cache line can be in 4 states
  - (M)odified
    - Exclusively modified copy of main memory among all cores
  - (E)xclusive
    - Exclusive copy of main memory among all cores
  - (S)hared
    - An exact copy of what is in main memory AND other cores may also have an unmodified copy
  - (I)nvalid
    - The cache line is stale and is no longer valid

MESI Protocol Messages

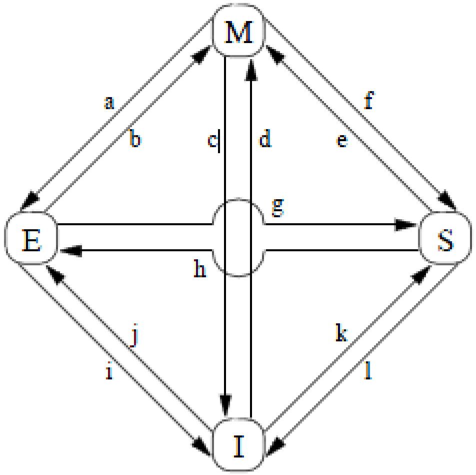
- Messages are sent on the ICB to maintain coherency between the caches
- Anyone on the ICB can reply to the ‘Read’ messages
  - Not just the memory controller but also other cores.

MESI Message Types

- Message Types (refers to a cache line)
  - Read / Read Acknowledge
  - RWITW – Read With Intent To Write
    - Read + Invalidate
  - Invalidate / Invalidate Acknowledge
    - Ask other cores to invalidate this cache line
  - Writeback
    - Write back cache line to main memory

Cache line transitions

- Read cache line
  - Invalid -> Exclusive
    - only core with a copy
  - Invalid -> Shared
    - other cores also have a copy
- Write to cache line
  - Exclusive -> Modified
  - Shared -> Modified
    - all other cores invalidate their version of this cache line
- Told to invalidate
  - Exclusive / Shared -> Invalid
  - Modified -> Invalid
    - triggers a 'writeback' to main memory
- Another core want to read our modified cache line
  - Modified -> Shared
    - triggers a 'writeback' to main memory



The Players…

- Example
  - Core 0 - Producer

```
void foo()
{
   data = 1;
   flag = 1;
}
```

  - Core 1 - Consumer

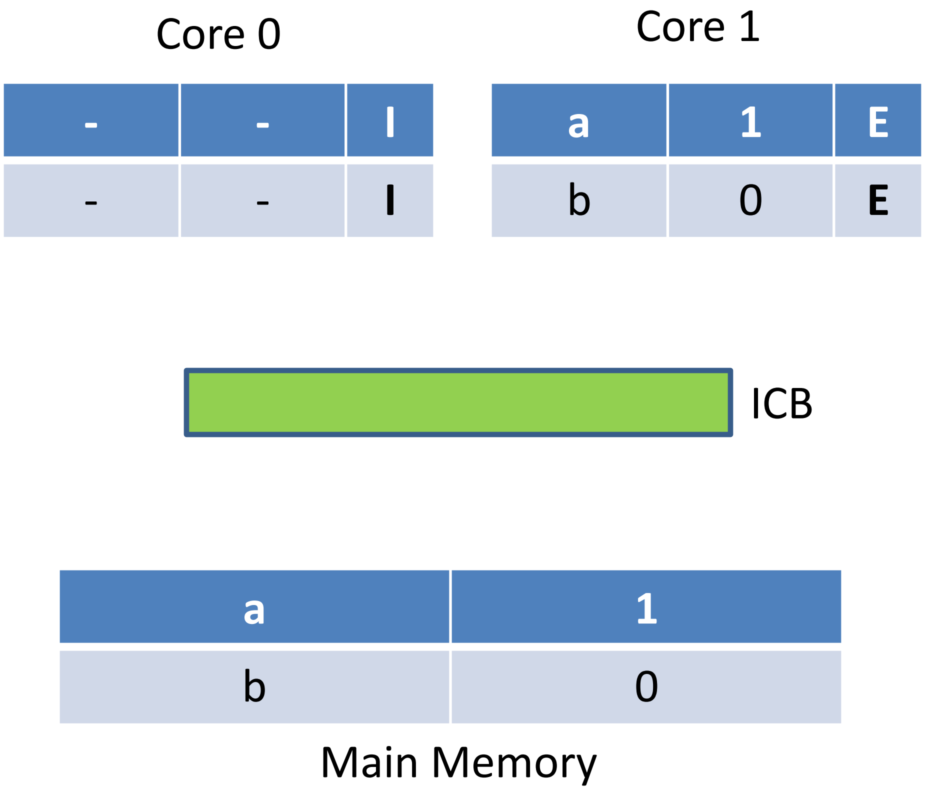
```
void bar()
{
   while (flag == 0);
   assert(data);
}
```

Cache Ownership Example

Initially Core 0's cache is empty and Core 1's contain the 'a' and 'b' cache lines

```
if (a)
{
    b = 4;
}
```

'a' and 'b' are on separate cache lines



Core 0

-	-	I
-	-	I

Core 1

a	1	E
b	0	E

a	1
b	0

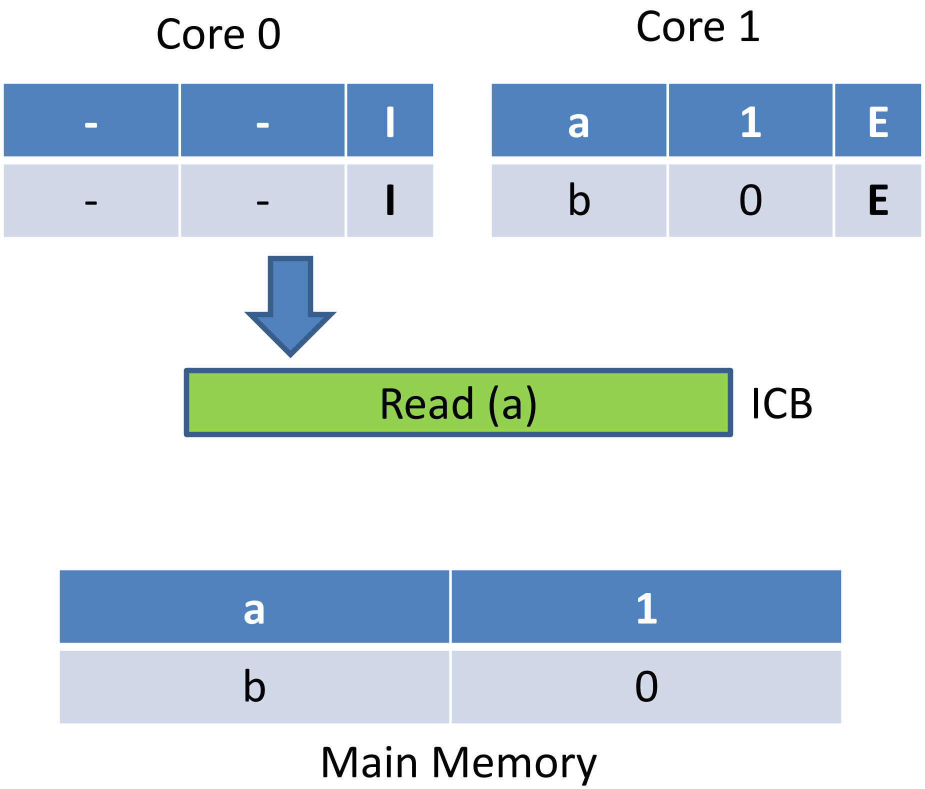
Main Memory

Cache Ownership Example

Core 0 does not have ‘a’ in its cache and therefore requests it

```
if (a)
{
    b = 4;
}
```

‘a’ and ‘b’ are on separate cache lines



Core 0

-	-	I
-	-	I

Core 1

a	1	E
b	0	E

Read (a)

ICB

a	1
b	0

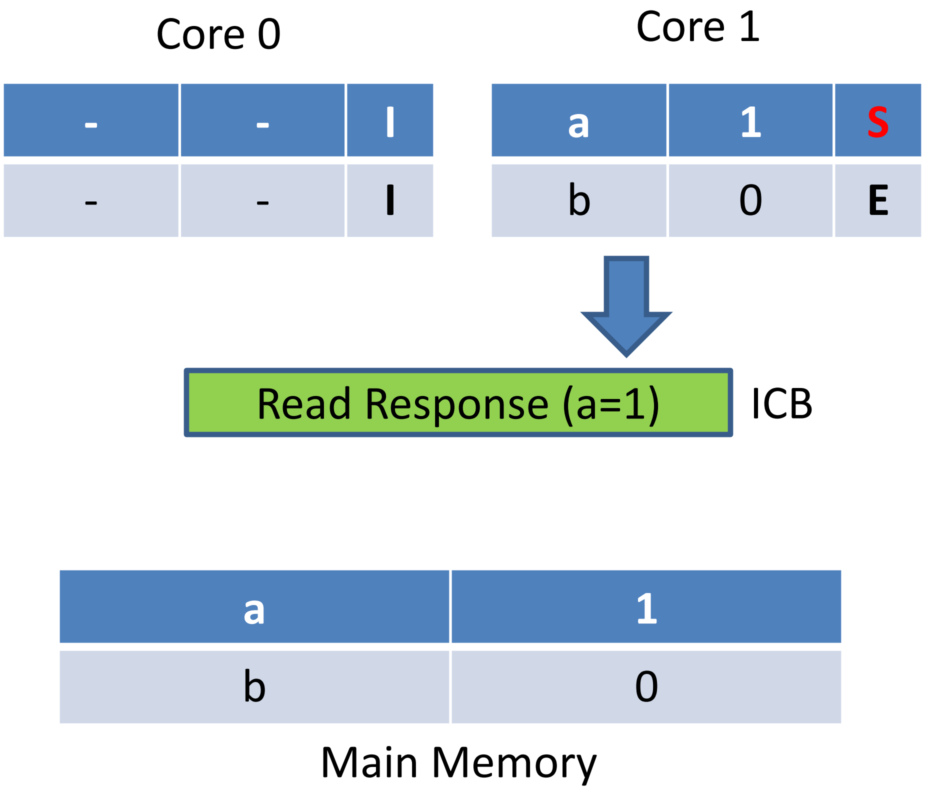
Main Memory

Cache Ownership Example

Core 1 sees the request and has the cache line ‘a’. It responds with the cache line and marks its own version as ‘Shared’

```
if (a)
{
    b = 4;
}
```

‘a’ and ‘b’ are on separate cache lines



Core 0

-	-	I
-	-	I

Core 1

a	1	S
b	0	E

Read Response (a=1)

ICB

a	1
b	0

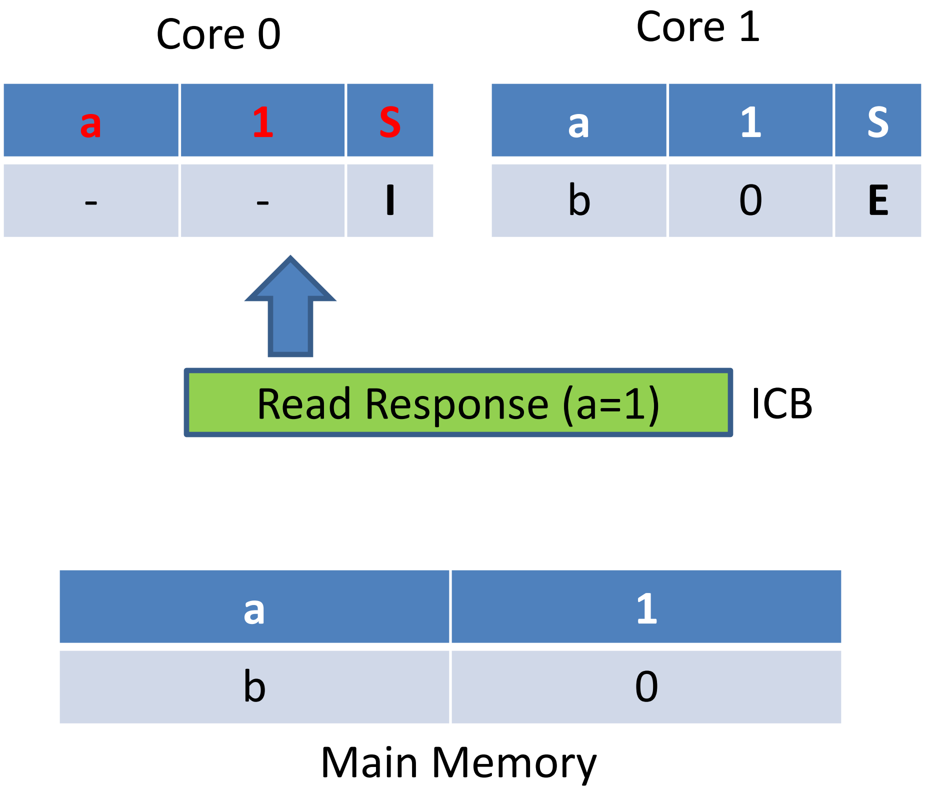
Main Memory

Cache Ownership Example

Core 0 receives the cache line and installs it in its cache. The branch can now be evaluated

```
if (a)
{
    b = 4;
}
```

‘a’ and ‘b’ are on separate cache lines



Core 0

a	1	S
-	-	I

Core 1

a	1	S
b	0	E

Read Response (a=1)

ICB

a	1
b	0

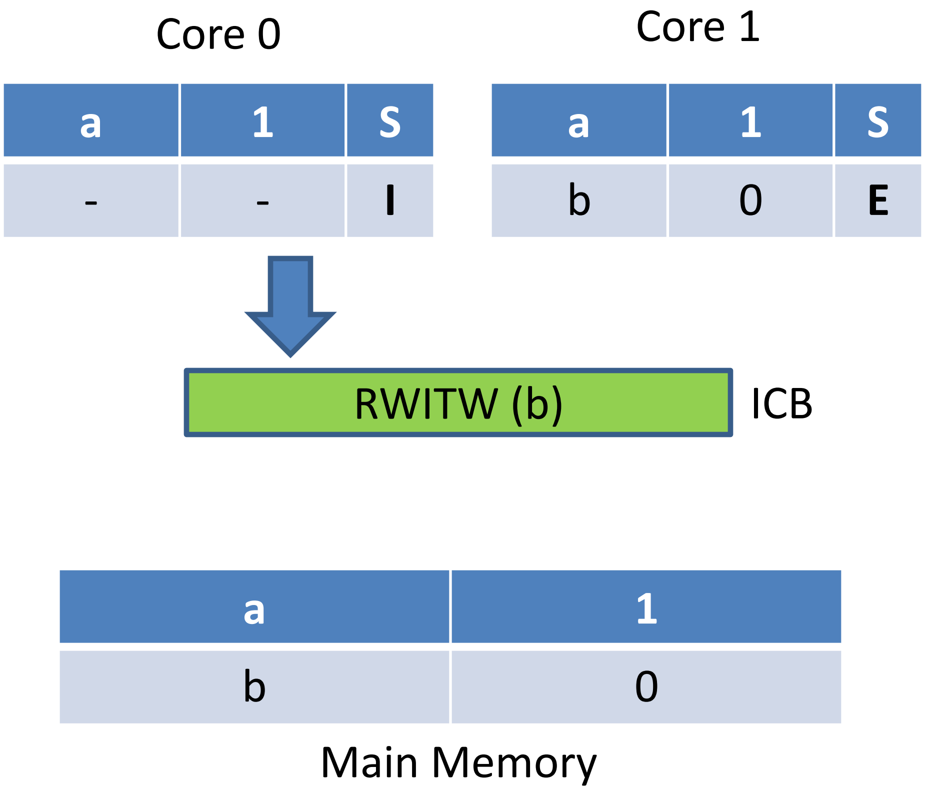
Main Memory

Cache Ownership Example

Core 0 does not have ‘b’ in its cache and therefore requests it. This time the request has a hint to indicate the intent to write to ‘b’.

```
if (a)
{
    b = 4;
}
```

‘a’ and ‘b’ are on separate cache lines



Core 0

a	1	S
-	-	I

Core 1

a	1	S
b	0	E

RWITW (b)

ICB

a	1
b	0

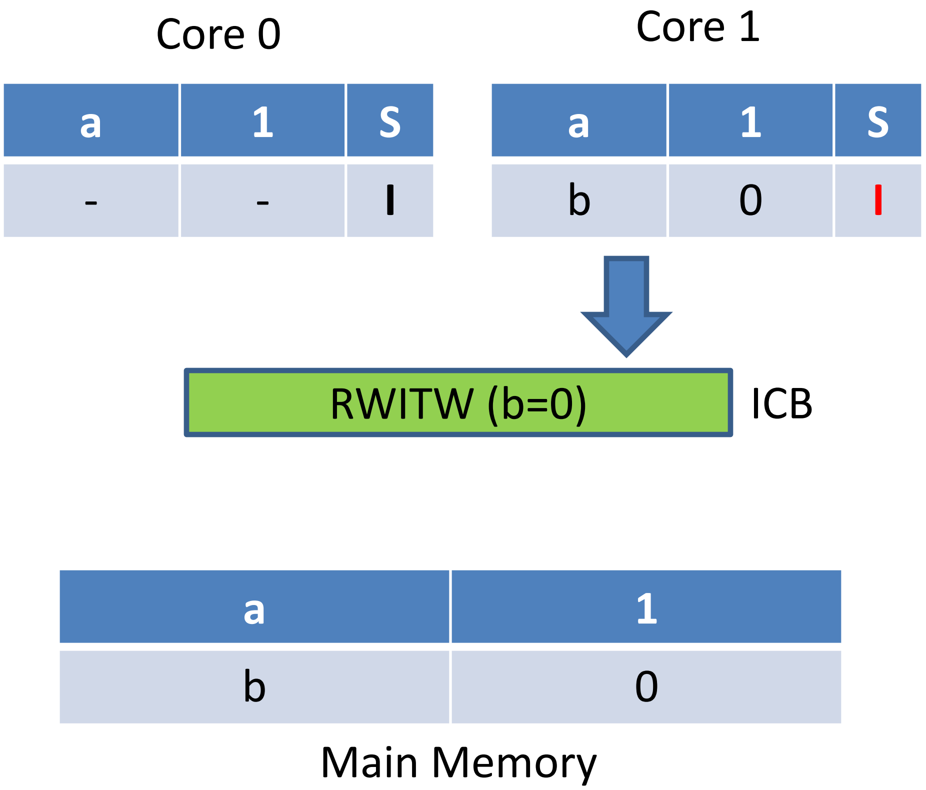
Main Memory

Cache Ownership Example

Core 1 sees the request on the ICB and returns the cache line. Because the ‘RWITW’ implies an invalidate request Core 1 now also invalidates ‘b’

```
if (a)
{
    b = 4;
}
```

‘a’ and ‘b’ are on separate cache lines



Core 0

a	1	S
-	-	I

Core 1

a	1	S
b	0	I

a	1
b	0

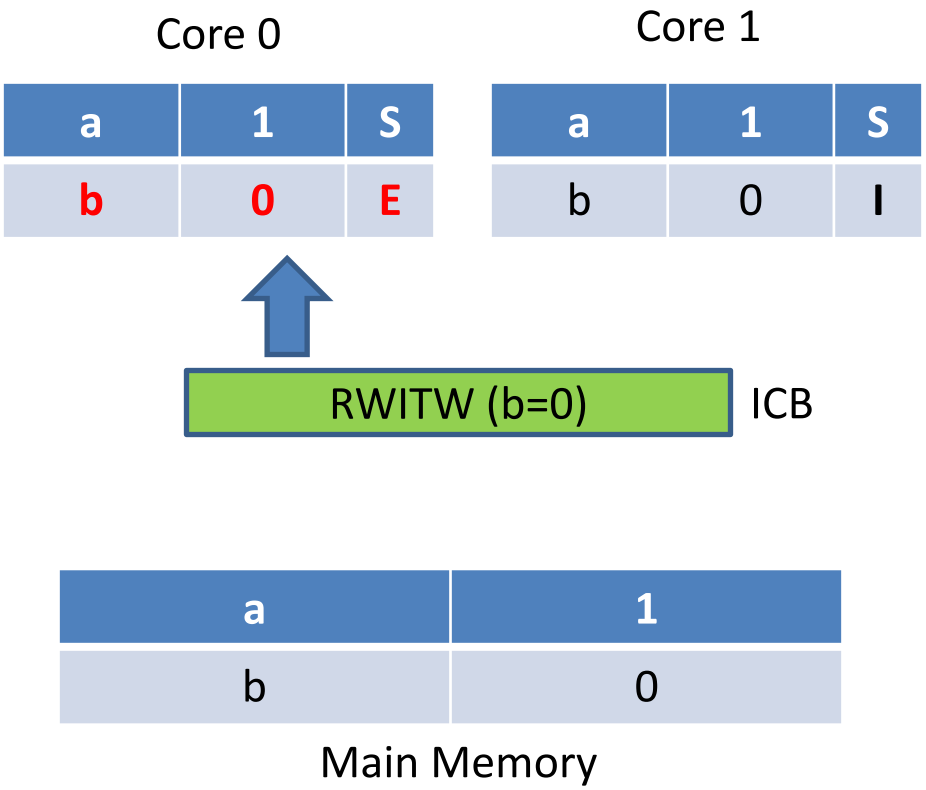
Main Memory

Cache Ownership Example

Core 0 receives the ‘b’ cache line and installs it in its cache as ‘Exclusive’

```
if (a)
{
    b = 4;
}
```

‘a’ and ‘b’ are on separate cache lines



Core 0

a	1	S
b	0	E

Core 1

a	1	S
b	0	I

a	1
b	0

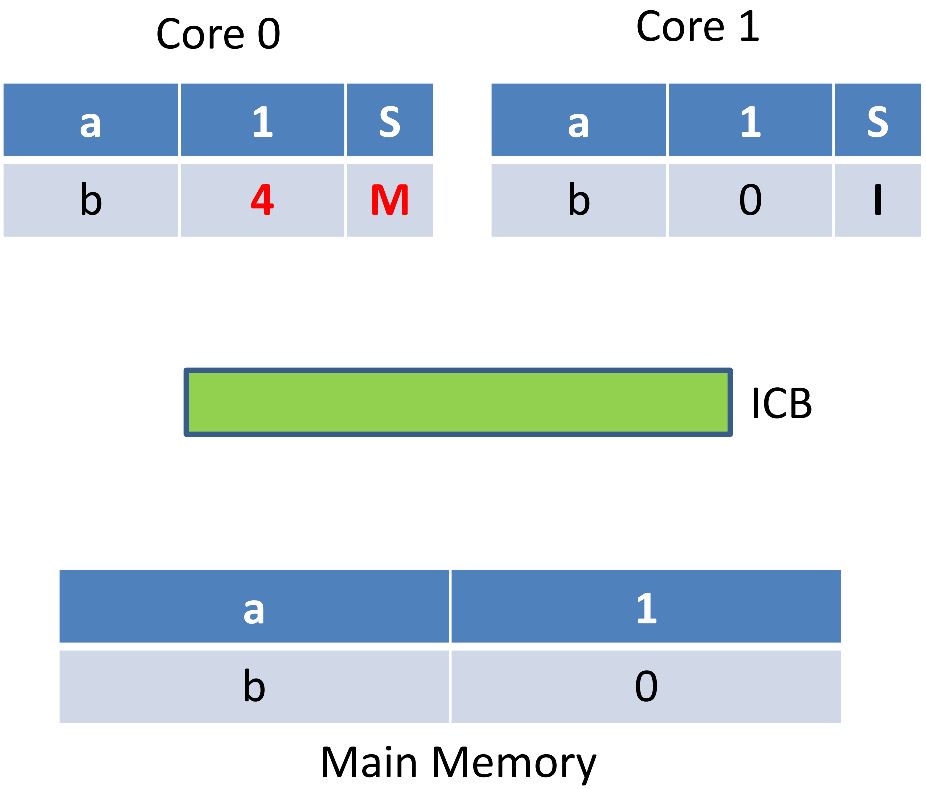
Main Memory

Cache Ownership Example

Core 0 now has the cache line and can commit the store to ‘b’. This marks the cache line as ‘Modified’ but stays in the cache and is not saved to main memory.

```
if (a)
{
    b = 4;
}
```

‘a’ and ‘b’ are on separate cache lines



Core 0

a	1	S
b	**4**	**M**

Core 1

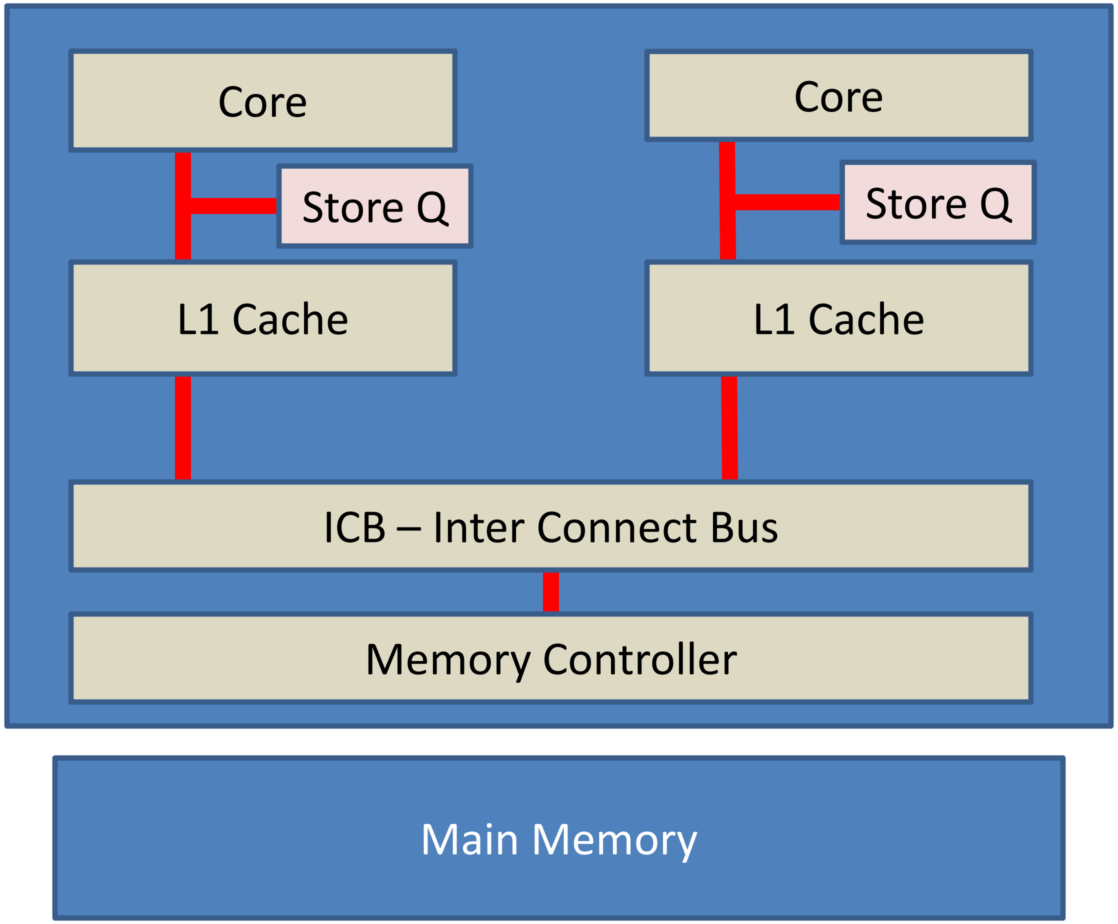
a	1	S
b	0	I

ICB

a	1
b	0

Main Memory

2-core CPU + Store Qs



Reasons for Store Q

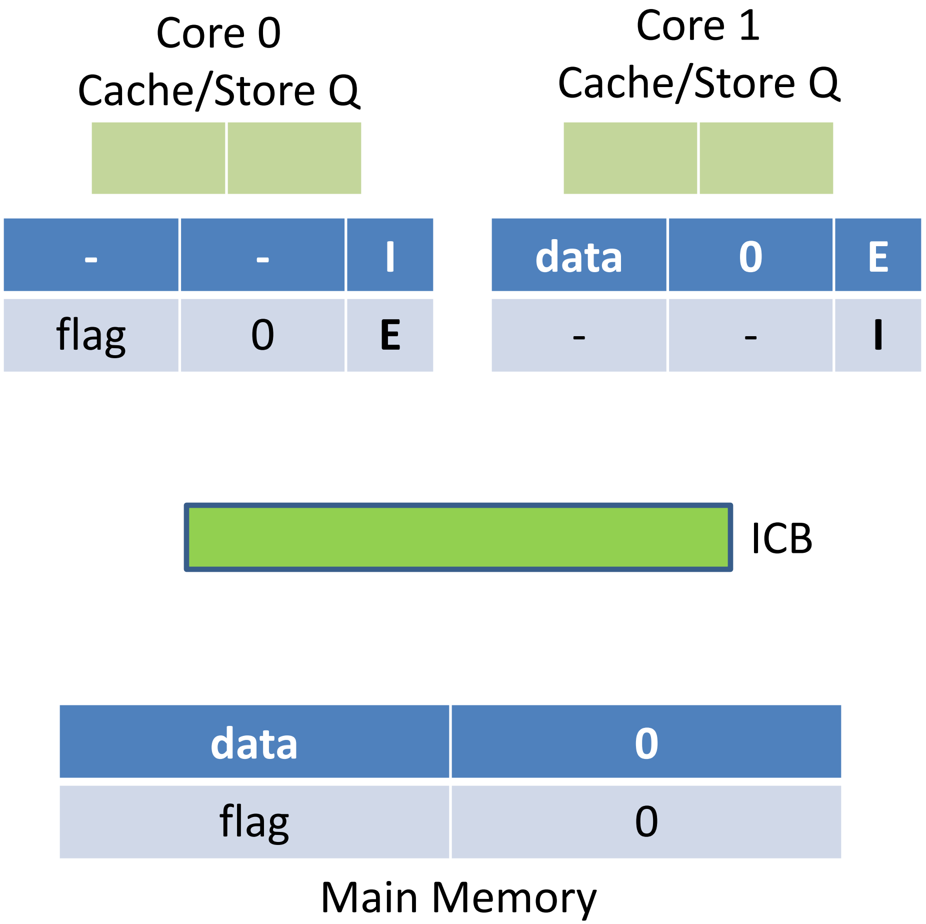
- Prevent CPU execution stall while waiting for a missing/invalid cache line
- Loads can now “pass” stores if the cache line is more readily available
  - It might be available in the local cache already or by a neighboring core.
- Requires snooping the Store Q for loads to ensure that memory looks the same for the locally running core.
  - Even if the store hasn’t made it into the cache a subsequent load should load the value that was stored

Store Q Issue Example

Core 0 executes ‘foo’
Core 1 executes ‘bar’
‘flag’ cache line is owned by ‘0’
‘data’ cache line is owned by ‘1’

```
void foo()
{
    data = 1;
    flag = 1;
}
```

```
void bar()
{
    while (flag == 0);
    assert(data);
}
```



Core 0
Cache/Store Q

-	-	I
flag	0	E

Core 1
Cache/Store Q

data	0	E
-	-	I

data	0
flag	0

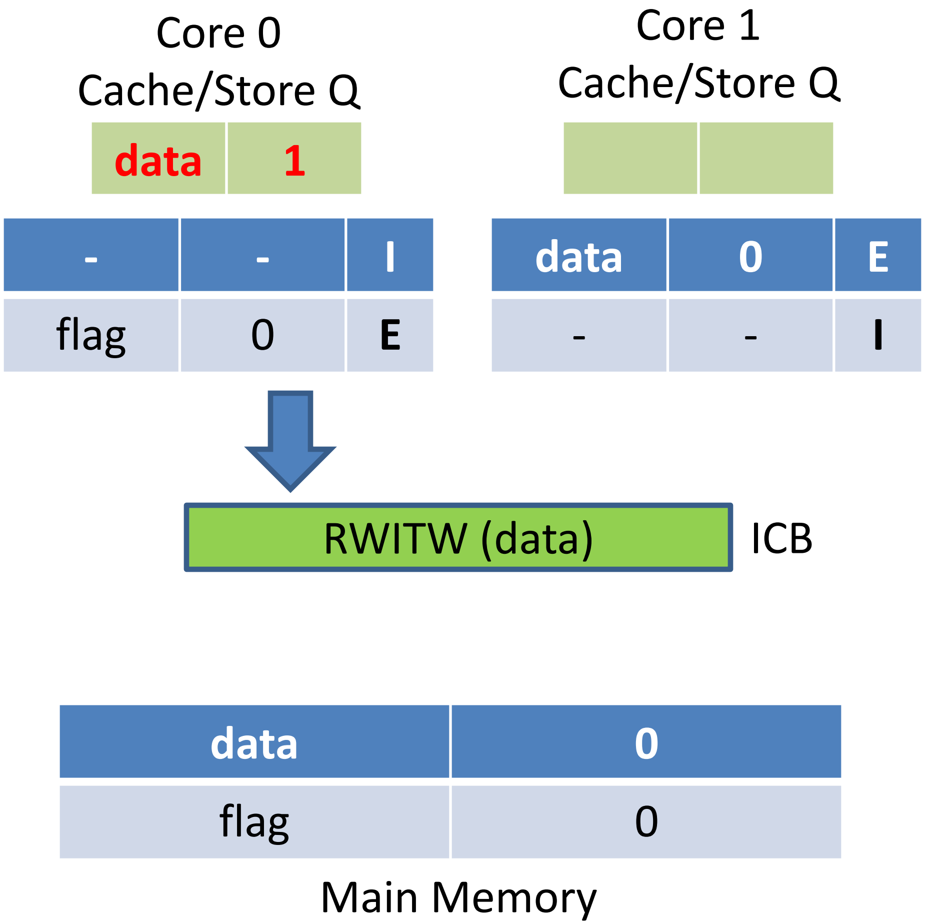
Main Memory

Store Q Issue Example

Core 0 saves the store in the Store Q and issues a RWITW message for ‘data’ due to it not being in the cache

```
void foo()
{
   data= 1;
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

data	**1**

-	**-**	**I**
flag	0	**E**

Core 1 Cache/Store Q

data	**0**	**E**
-	-	**I**

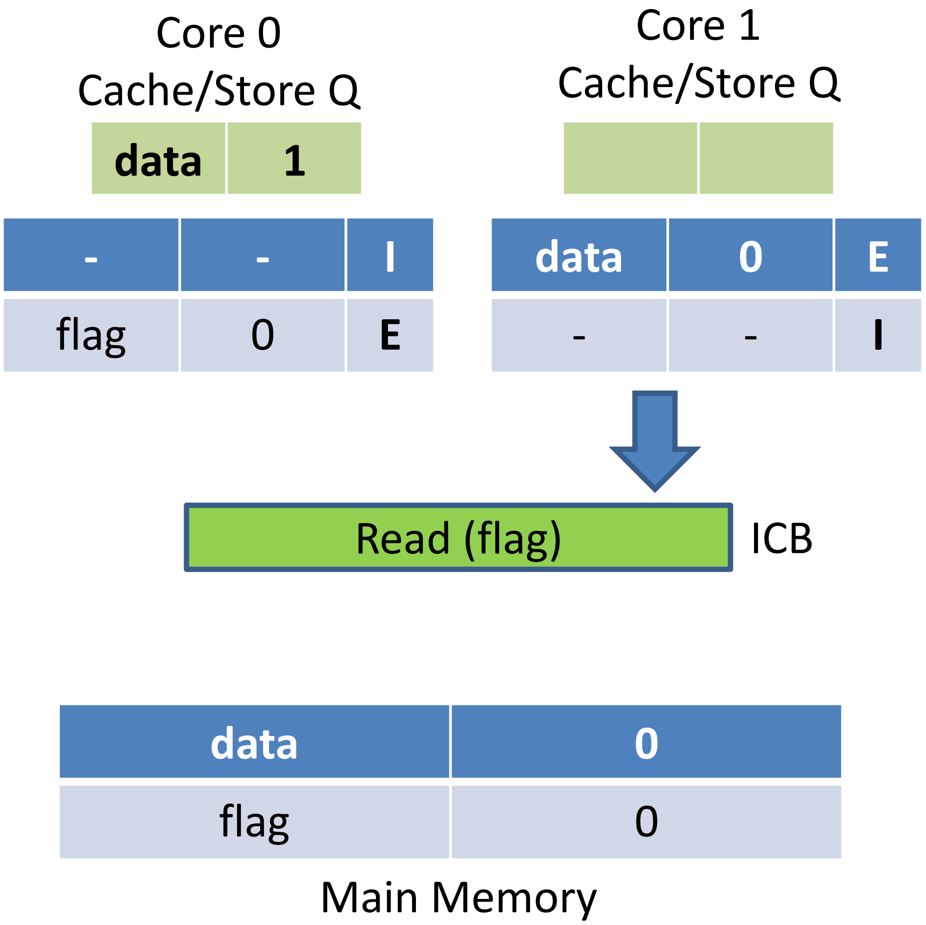
RWITW (data) ICB

data	**0**
flag	0

Main Memory

Store Q Issue Example

Core 1 issues a read message for 'flag' due to it not being in the cache



Core 0
Cache/Store Q

data	1

-	-	I
flag	0	E

Core 1
Cache/Store Q

data	0	E
-	-	I

Read (flag)

ICB

data	0
flag	0

Main Memory

```
void foo()
{
   data= 1;
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```

Store Q Issue Example

Core 0 owns ‘flag’ and hence updates the cache with ‘1’ and marks as modified

Core 0 Cache/Store Q

data	1

-	-	I
flag	1	M

Core 1 Cache/Store Q

data	0	E
-	-	I

```
void foo()
{
   data= 1;
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```

ICB

data	0
flag	0

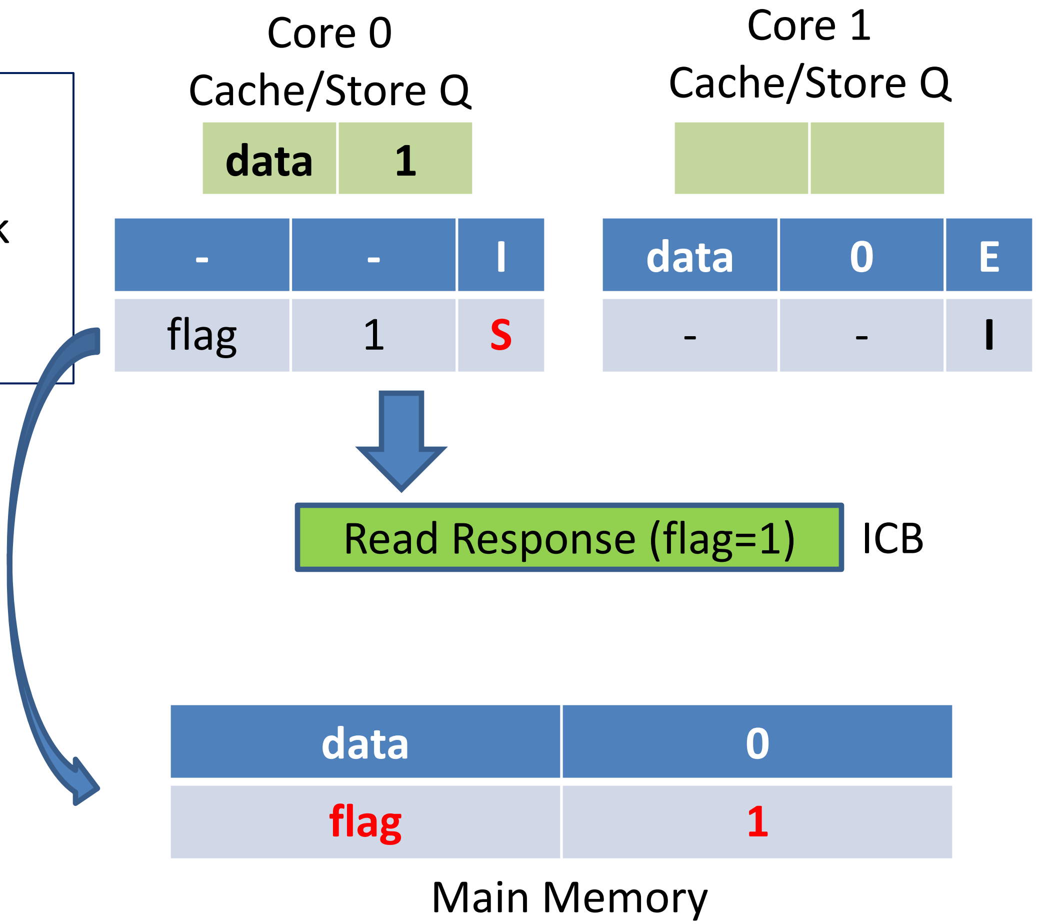
Main Memory

Store Q Issue Example

Core 0 respond to the read request of ‘flag’
The cache line is written back to main memory and also marked as Shared.

```
void foo()
{
   data= 1;
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

data	**1**

-	-	I
flag	1	**S**

Core 1 Cache/Store Q

data	**0**	**E**
-	-	I

Read Response (flag=1)

ICB

data	**0**
flag	**1**

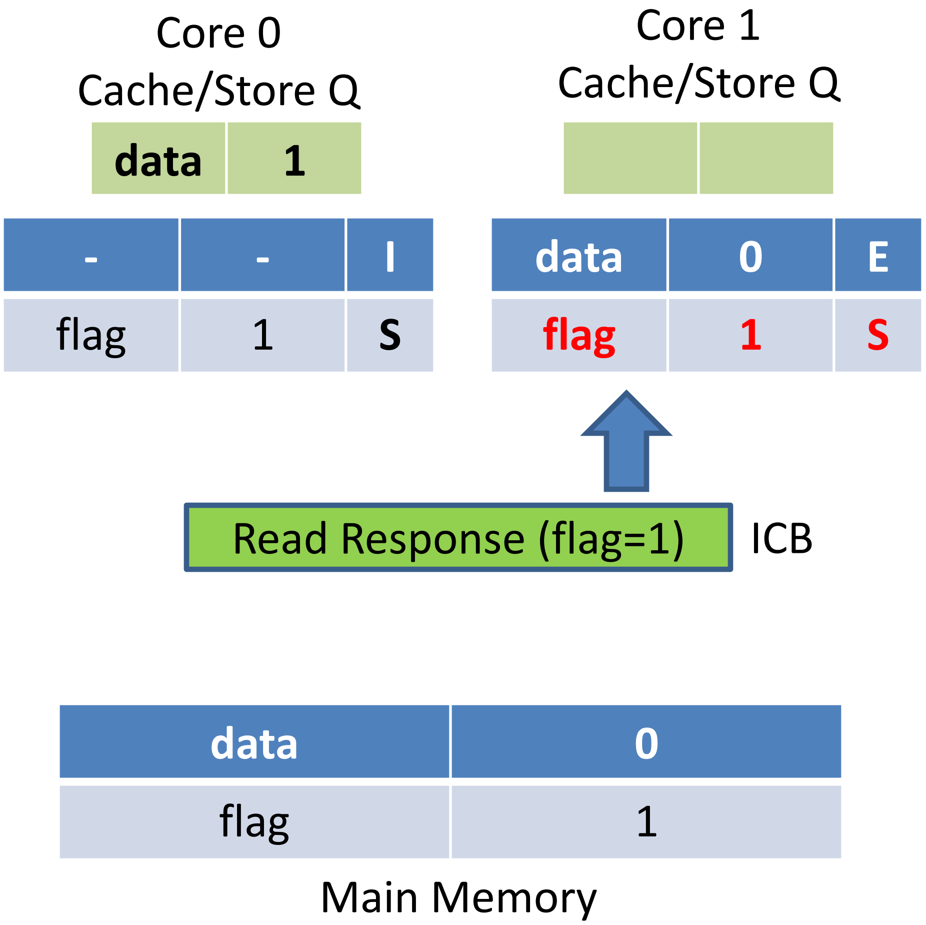
Main Memory

Store Q Issue Example

Core 1 receives the read response and marks the cache line as Shared

```
void foo()
{
   data= 1;
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0
Cache/Store Q

data	1

-	-	I
flag	1	S

Core 1
Cache/Store Q

data	0	E
flag	**1**	**S**

Read Response (flag=1)

ICB

data	0
flag	1

Main Memory

Store Q Issue Example

Core 1 now moves on to the next instruction. ‘data’ is in the cache and is therefore read. ASSERT!!

Core 0
Cache/Store Q

data	1

-	-	I
flag	1	S

Core 1
Cache/Store Q

data	0	E
flag	1	S

ICB

data	0
flag	1

Main Memory

```
void foo()
{
   data= 1;
   flag = 1;
}
```

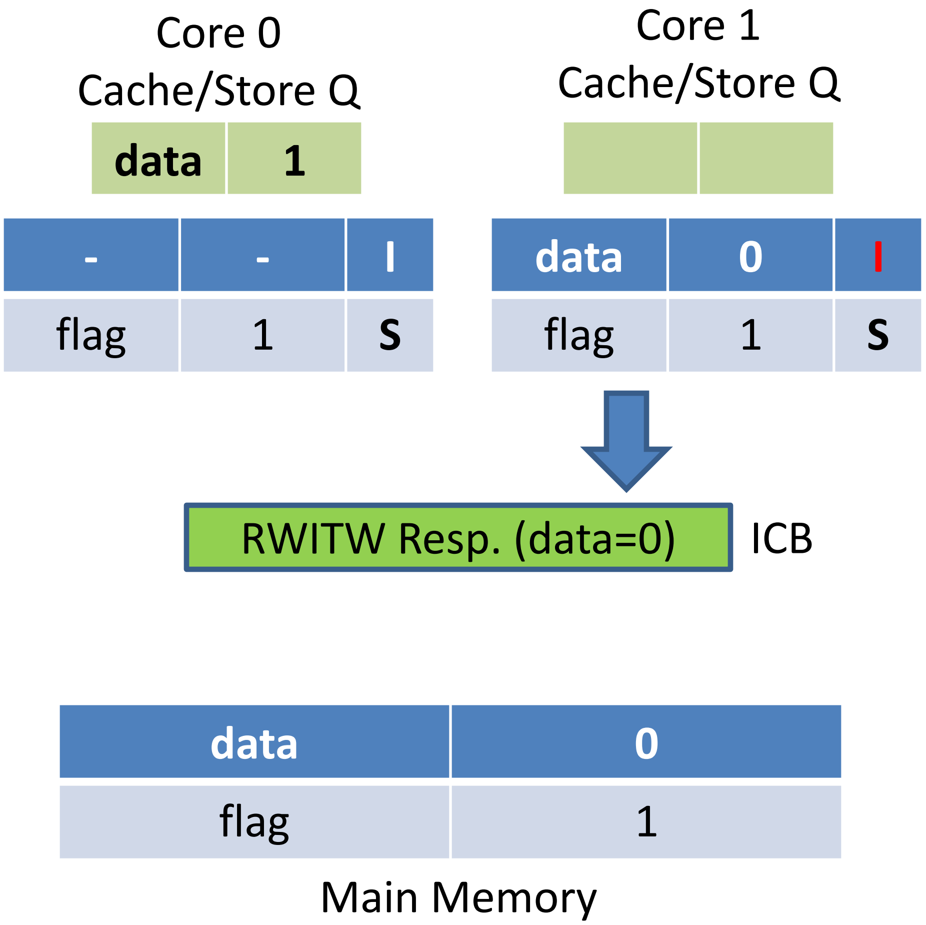
```
void bar()
{
   while (flag == 0);
   assert(data);
}
```

Store Q Issue Example

Core 1 now receive the delayed “Read Invalidate” message. It replies and marks its cache line as invalid

```
void foo()
{
   data= 1;
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

data	1

-	-	I
flag	1	S

Core 1 Cache/Store Q

data	0	I
flag	1	S

RWITW Resp. (data=0)

ICB

data	0
flag	1

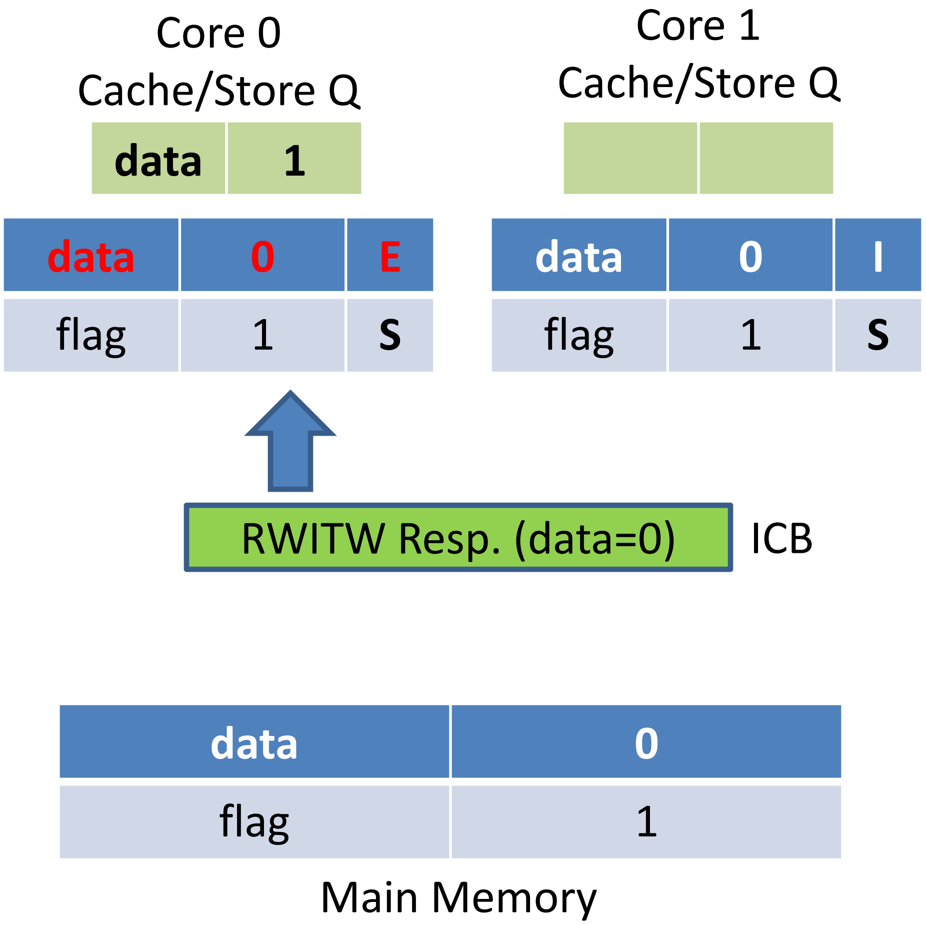
Main Memory

Store Q Issue Example

Core 0 receives the cache line and installs it in its cache.

```
void foo()
{
   data= 1;
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

data	1

data	0	E
flag	1	S

Core 1 Cache/Store Q

data	0	I
flag	1	S

RWITW Resp. (data=0)

ICB

data	0
flag	1

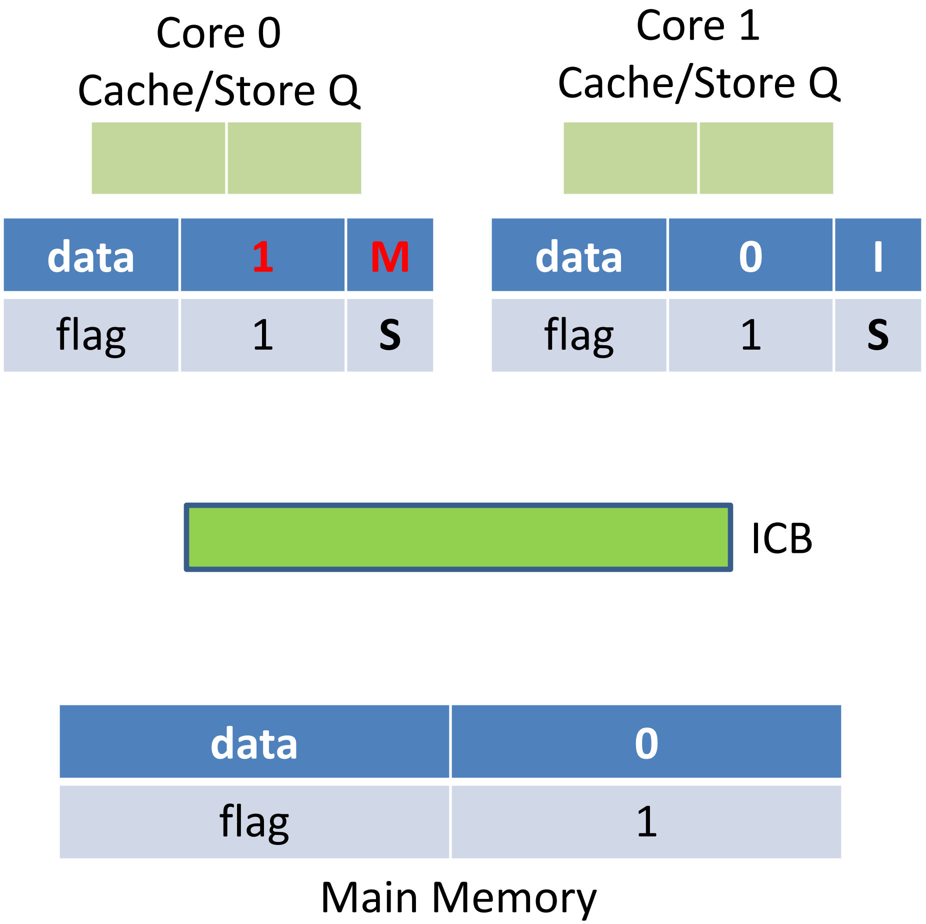
Main Memory

Store Q Issue Example

Core 0's Store Q can finally commit the write to the 'flag' cache line but it is too late. Core 1 is halted and execution stops.

```
void foo()
{
   data= 1;
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0
Cache/Store Q

data	1	M
flag	1	S

Core 1
Cache/Store Q

data	0	I
flag	1	S

data	0
flag	1

Main Memory

How do we solve this issue?

- All caches have a coherent view of main memory BUT local writes are not part of that
- We need a way to ensure that our stored data is part of the ‘cache coherent domain’
  - – I.e Visible by other cores
    - I.e Can be fetched by other caches
- Can we flush the store Q to the cache?
  - – Memory Store Barriers (__mb_release)

Memory Store Barriers

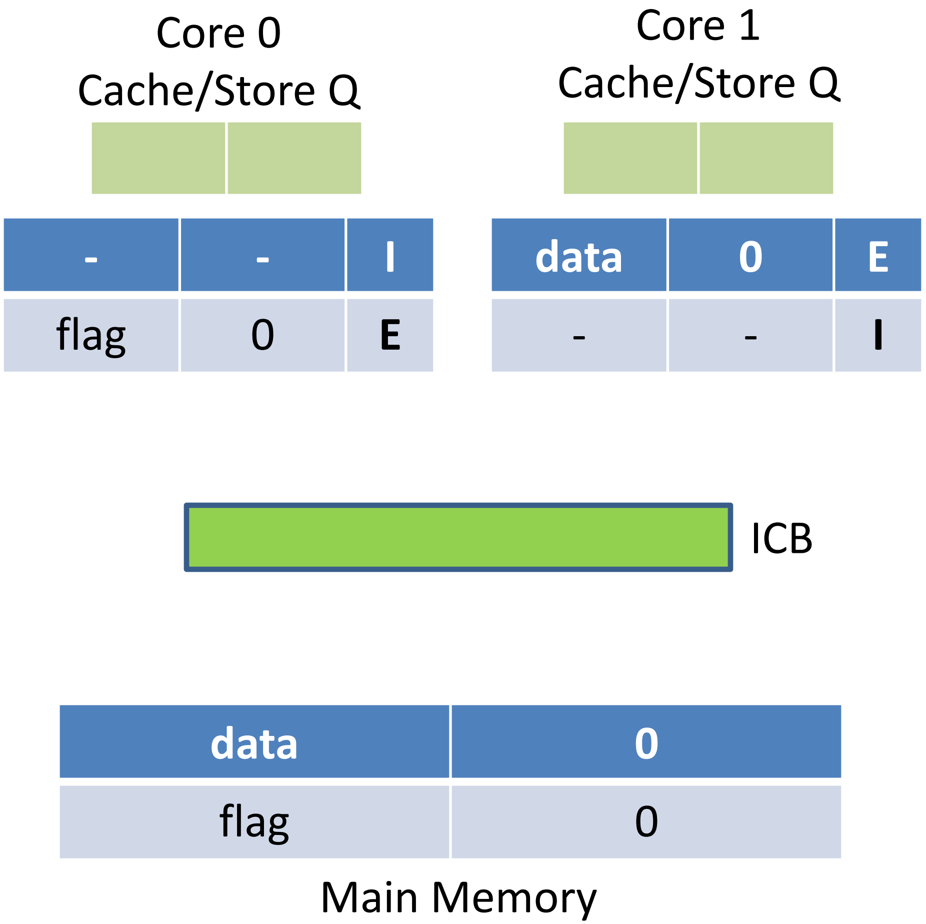
- CPU instruction that won’t return until all data in the Store Q preceding the memory barrier is in the cache
  - CPUs are evil!
- Prevents compilers from optimize memory stores across this barrier.
  - Compilers are evil!
- Once the data is in the cache it can be seen by all other caches due to the cache line being invalidated in all other caches.
  - RWITW (Read With Intent To Write)
    - Read + Invalidate

Store Q Issue Example (Fixed)

Core 0 executes ‘foo’
Core 1 executes ‘bar’
‘data’ cache line is owned by ‘1’
‘flag’ cache line is owned by ‘0’

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0
Cache/Store Q

-	-	I
flag	0	E

Core 1
Cache/Store Q

data	0	E
-	-	I

data	0
flag	0

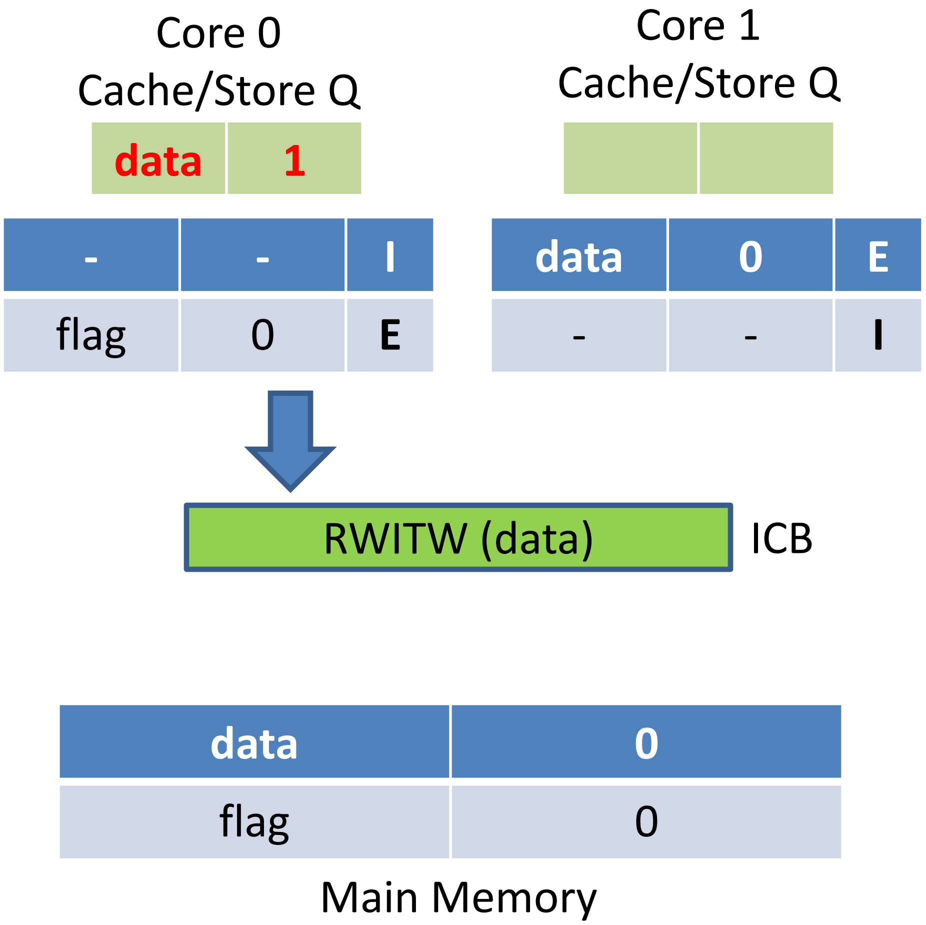
Main Memory

Store Q Issue Example (Fixed)

Core 0 saves the write in the Store Q and issues a RWITW message for 'data' due to it not being in the cache

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

data	1

-	-	I
flag	0	E

Core 1 Cache/Store Q

data	0	E
-	-	I

data	0
flag	0

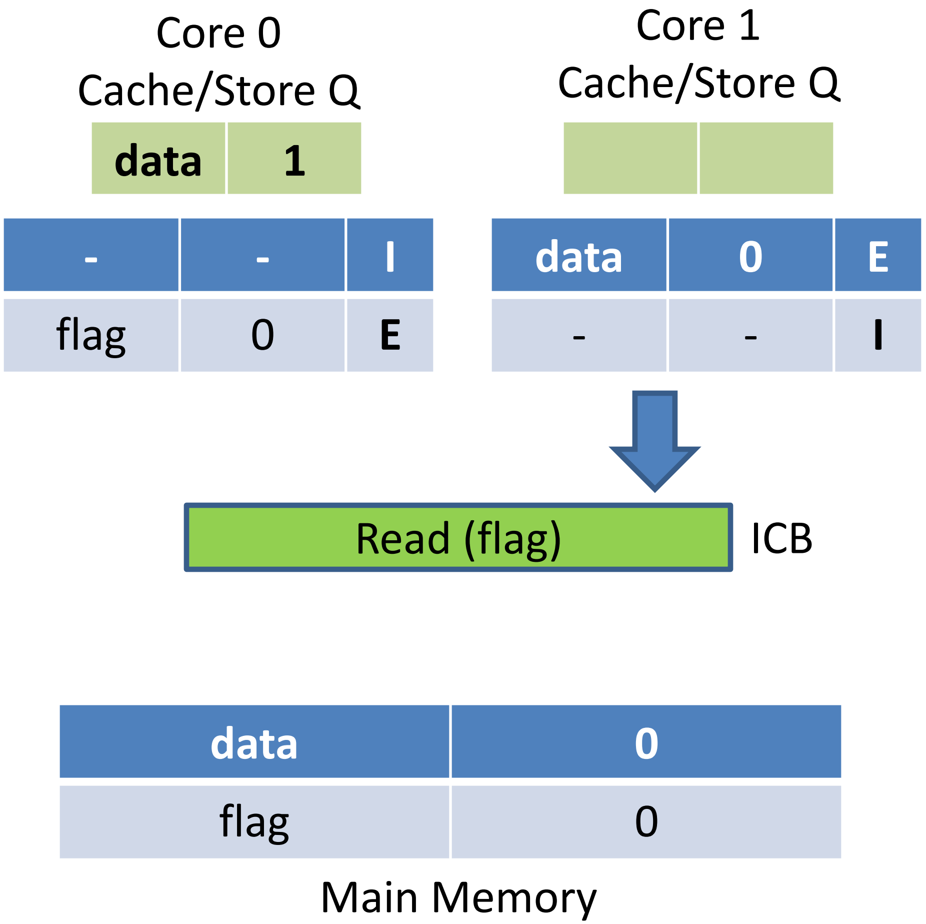
Main Memory

Store Q Issue Example (Fixed)

Core 1 issues a read message for ‘flag’ due to it not being in the cache

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0
Cache/Store Q

data	**1**

-	**-**	**I**
flag	0	**E**

Core 1
Cache/Store Q

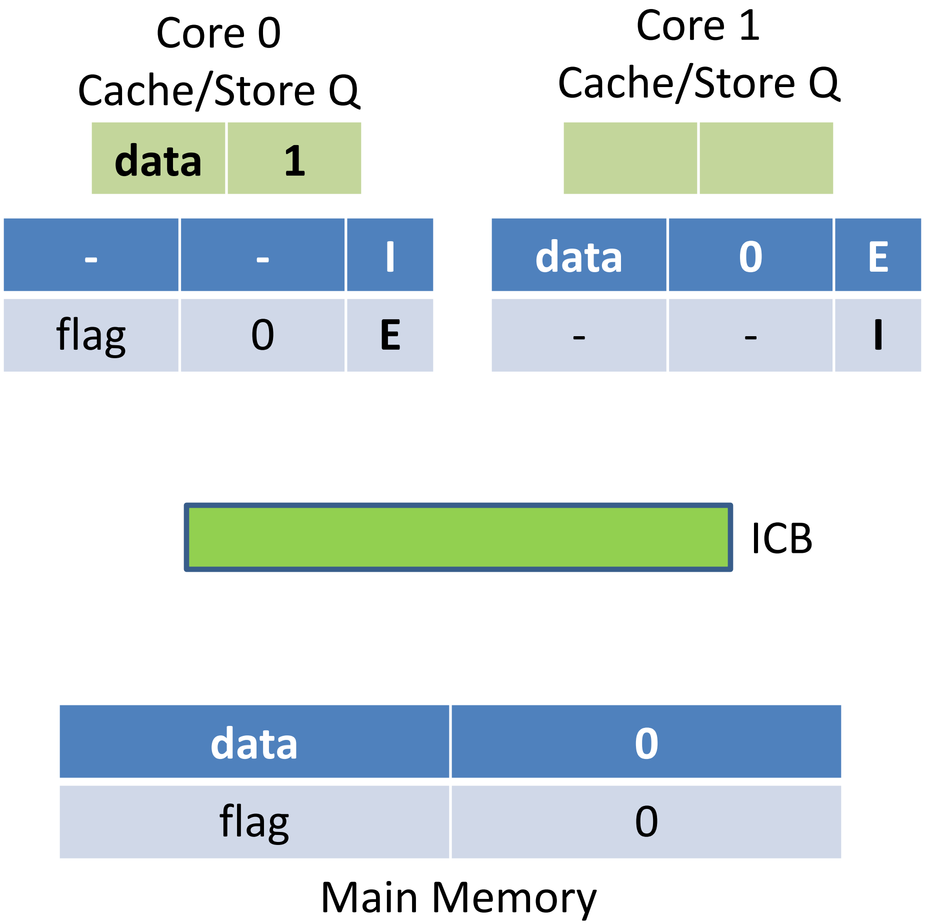
data	**0**	**E**
-	-	**I**

data	**0**
flag	0

Main Memory

Store Q Issue Example (Fixed)

Core 0 blocks on the memory barrier for the store Q to be flushed to the cache



Core 0 Cache/Store Q

data	1

-	-	I
flag	0	E

Core 1 Cache/Store Q

data	0	E
-	-	I

ICB

data	0
flag	0

Main Memory

```
void foo()
{
    data = 1;
    __mb_release();
    flag = 1;
}
```

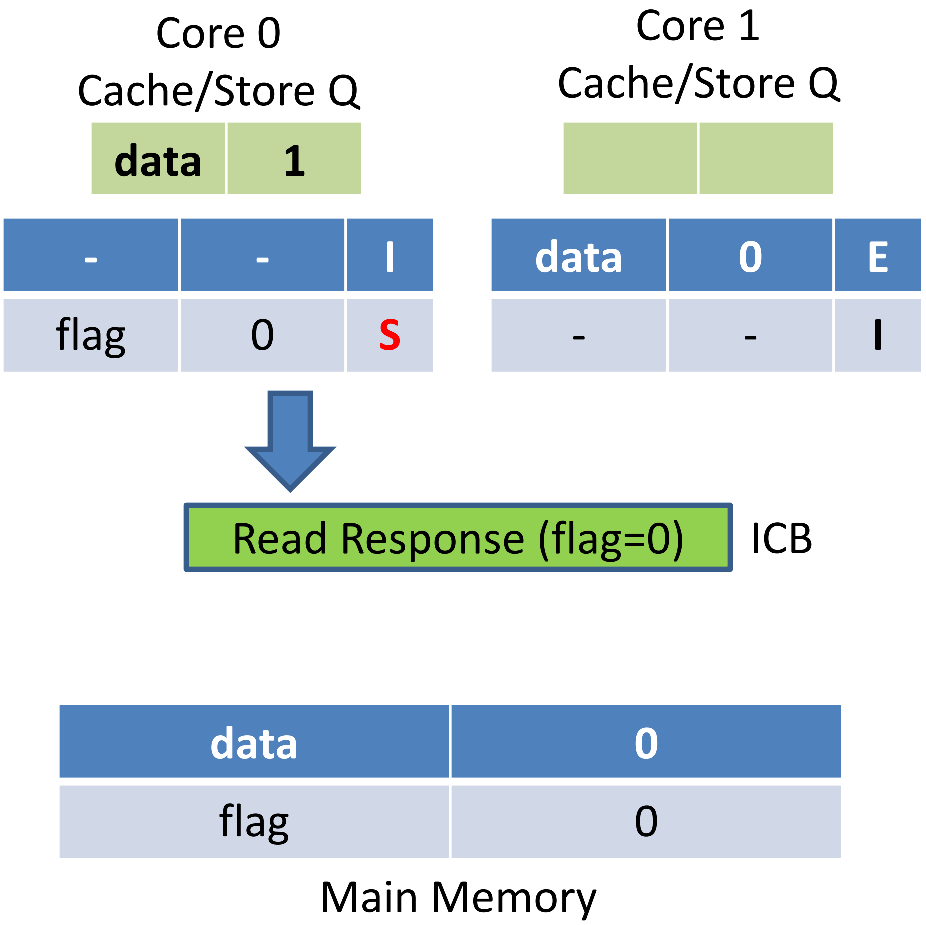
```
void bar()
{
    while (flag == 0);
    assert(data);
}
```

Store Q Issue Example (Fixed)

Core 0 respond to the read request of ‘flag’
The cache line sent to Core 1 and also marked as Shared on Core 0.

```
void foo()
{
    data = 1;
    __mb_release();
    flag = 1;
}
```

```
void bar()
{
    while (flag == 0);
    assert(data);
}
```



Core 0
Cache/Store Q

data	**1**

-	**-**	**I**
flag	0	**S**

Core 1
Cache/Store Q

data	**0**	**E**
-	-	**I**

data	**0**
flag	0

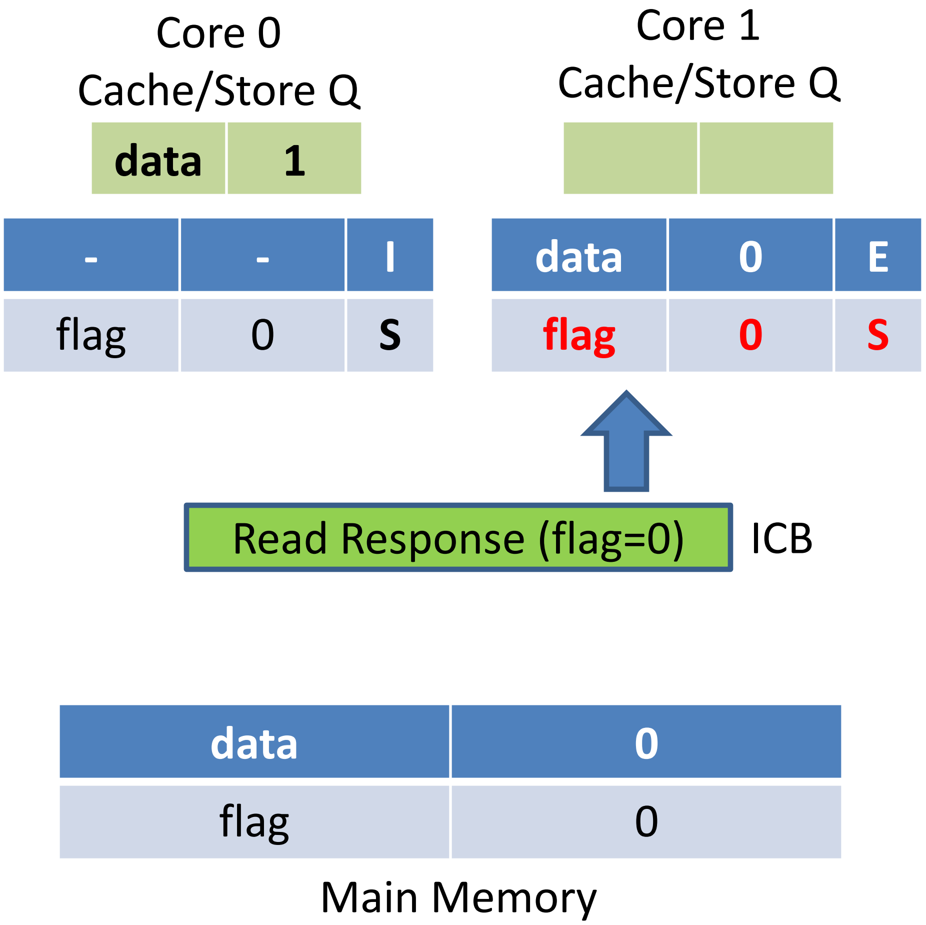
Main Memory

Store Q Issue Example (Fixed)

Core 1 receives the read response and marks the cache line as Shared

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0
Cache/Store Q

data	1

-	-	I
flag	0	S

Core 1
Cache/Store Q

data	0	E
flag	0	S

Read Response (flag=0)

ICB

data	0
flag	0

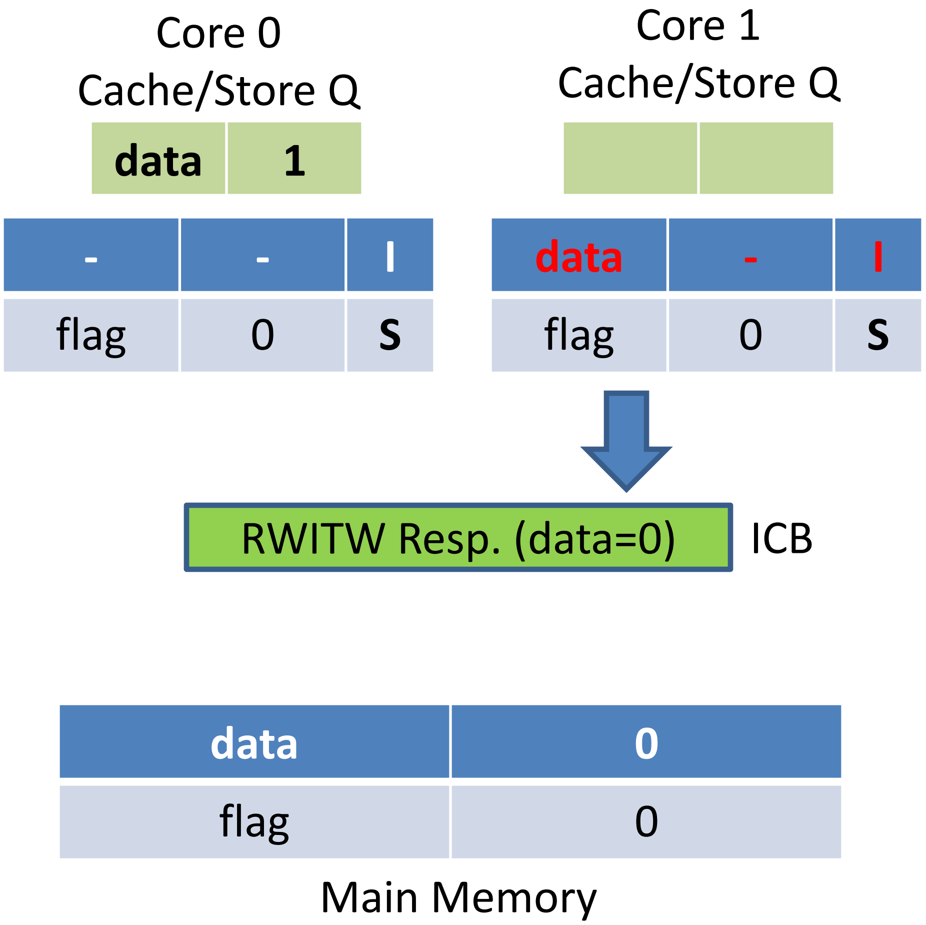
Main Memory

Store Q Issue Example (Fixed)

Core 1 now receive the delayed Read Invalidate message. It replies and marks its cache line as ‘Invalid’

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

data	1

-	-	I
flag	0	S

Core 1 Cache/Store Q

data	-	I
flag	0	S

RWITW Resp. (data=0)

ICB

data	0
flag	0

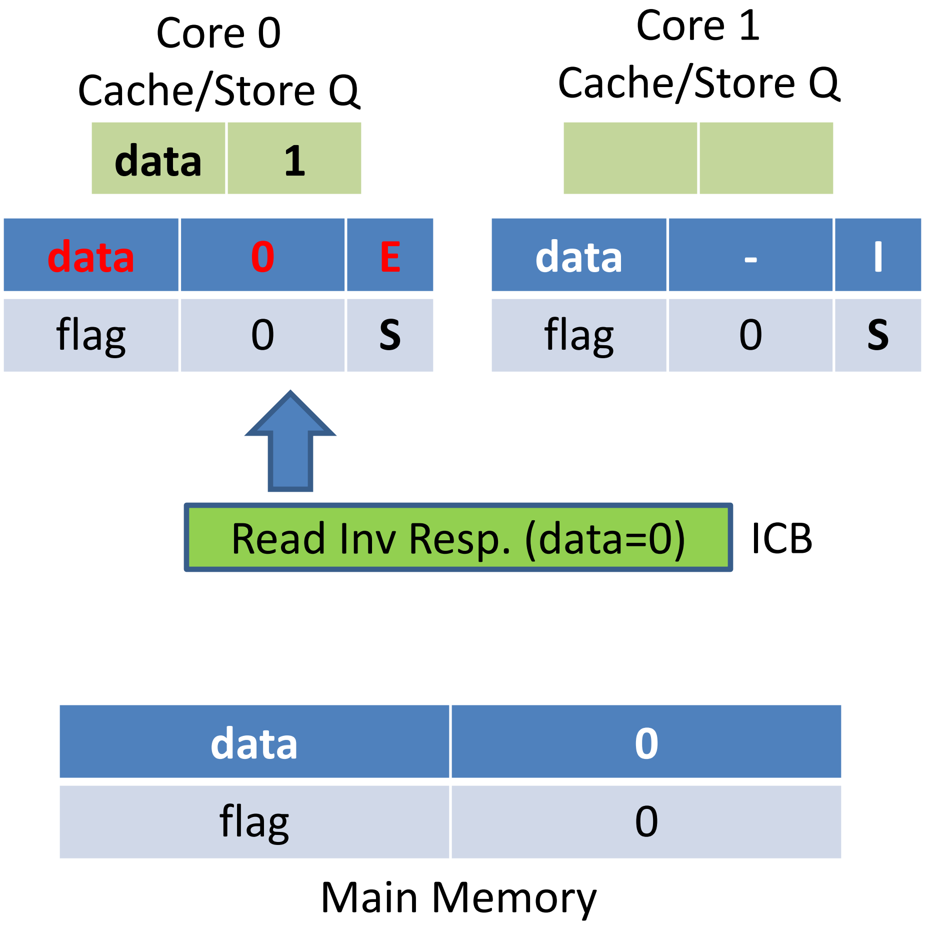
Main Memory

Store Q Issue Example (Fixed)

Core 0 receives 'data' as the "Read Invalidate" response from Core 1

```
void foo()
{
    data = 1;
    __mb_release();
    flag = 1;
}
```

```
void bar()
{
    while (flag == 0);
    assert(data);
}
```



Core 0
Cache/Store Q

data	1

data	0	E
flag	0	S

Core 1
Cache/Store Q

data	-	I
flag	0	S

Read Inv Resp. (data=0)

ICB

data	0
flag	0

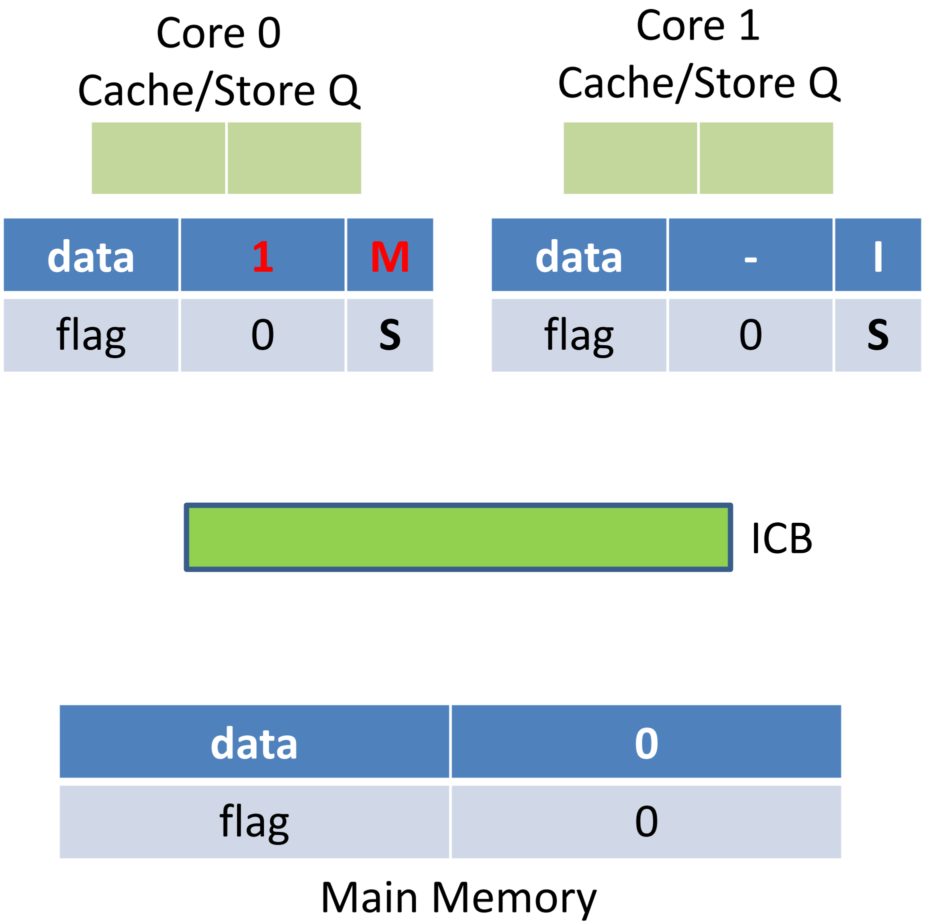
Main Memory

Store Q Issue Example (Fixed)

Core 0 now commits the write in the Store Q into the cache and marks it as ‘Modified’

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0
Cache/Store Q

data	1	M
flag	0	S

Core 1
Cache/Store Q

data	-	I
flag	0	S

data	0
flag	0

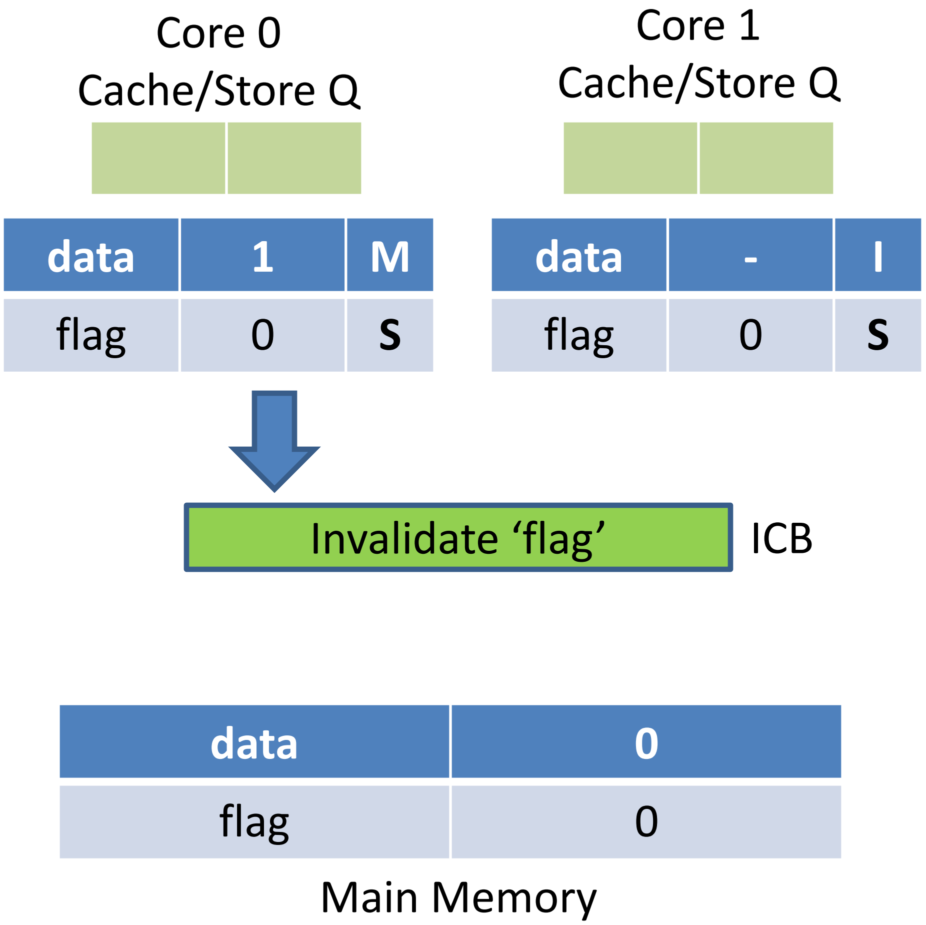
Main Memory

Store Q Issue Example (Fixed)

Core 0 want to set ‘flag’ to 1 but because ‘flag’ is shared between cores an ‘Invalidate’ message needs to be sent out first

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0
Cache/Store Q

data	1	M
flag	0	S

Core 1
Cache/Store Q

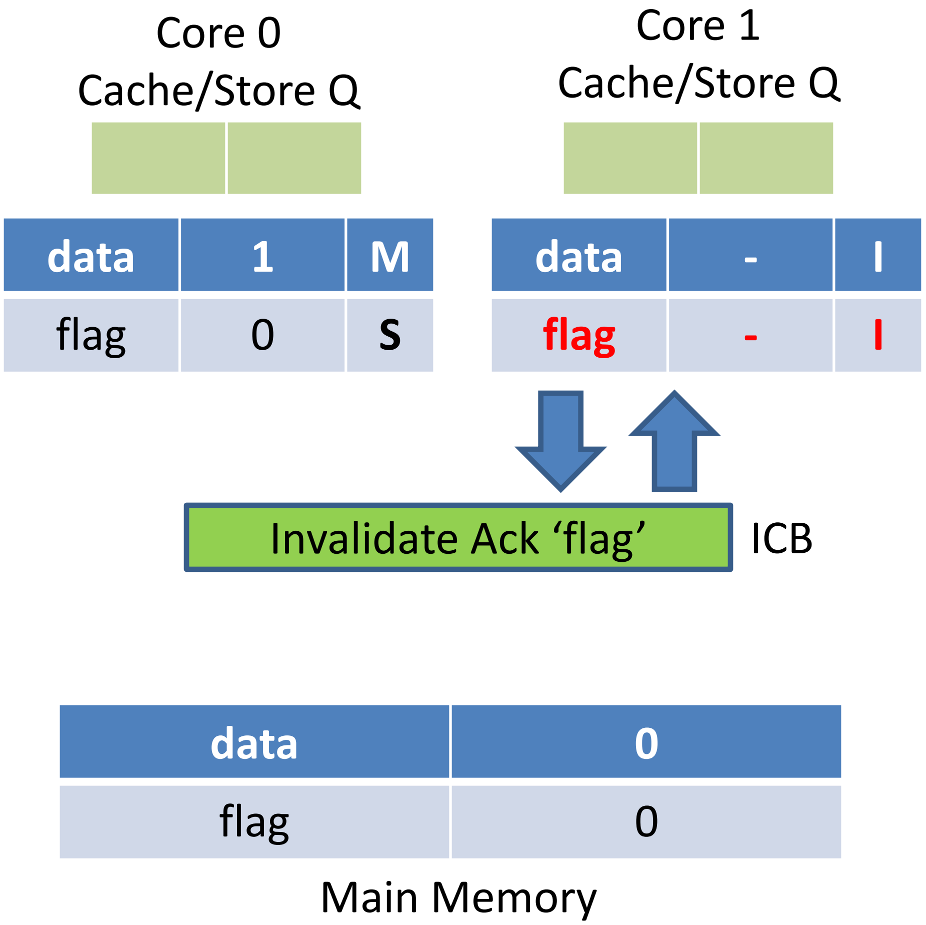
data	-	I
flag	0	S

data	0
flag	0

Main Memory

Store Q Issue Example (Fixed)

Core 1 receives the 'Invalidate' and sends an 'Invalidate Acknowledge' response



Core 0
Cache/Store Q

data	1	M
flag	0	S

Core 1
Cache/Store Q

data	-	I
flag	-	I

data	0
flag	0

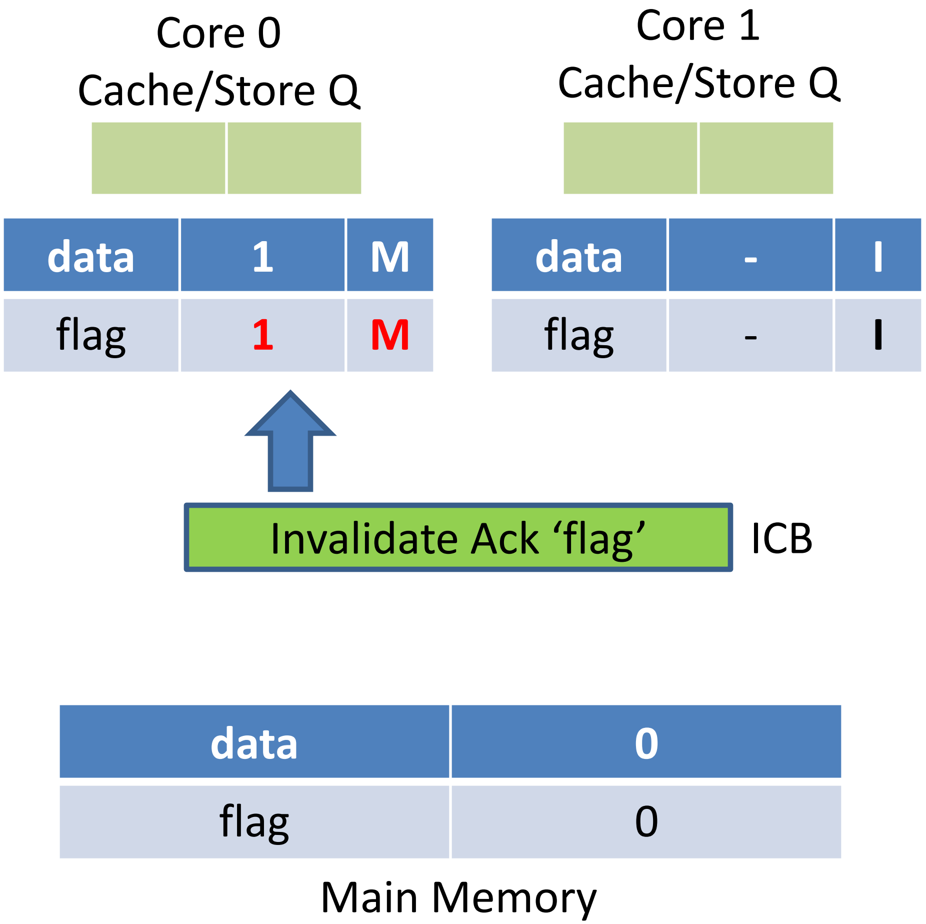
Main Memory

```
void foo()
{
    data = 1;
    __mb_release();
    flag = 1;
}
```

```
void bar()
{
    while (flag == 0);
    assert(data);
}
```

Store Q Issue Example (Fixed)

Core 0 receives the ‘Invalidate Ack’ and can now modify ‘flag’



Core 0
Cache/Store Q

data	1	M
flag	1	M

Core 1
Cache/Store Q

data	-	I
flag	-	I

Invalidate Ack ‘flag’

ICB

data	0
flag	0

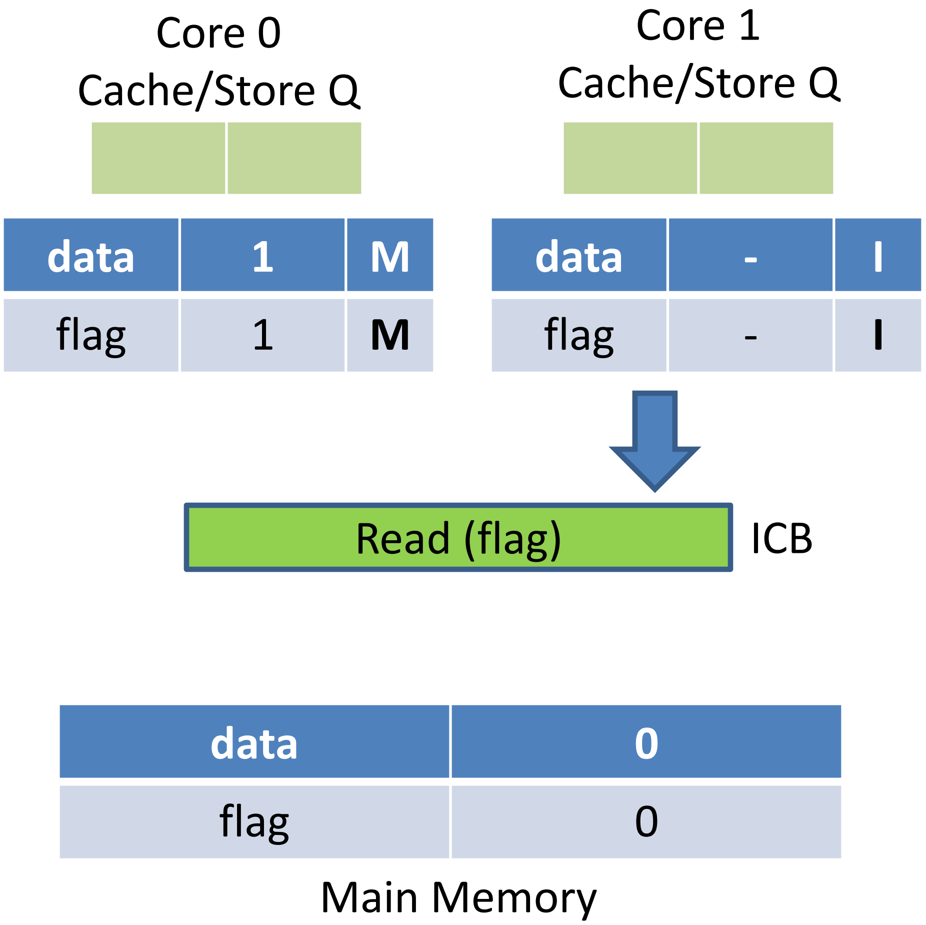
Main Memory

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```

Store Q Issue Example (Fixed)

Core 1 issues a read message for ‘flag’ due to it being marked ‘Invalid’ in the cache



Core 0 Cache/Store Q

data	1	M
flag	1	M

Core 1 Cache/Store Q

data	-	I
flag	-	I

data	0
flag	0

Main Memory

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

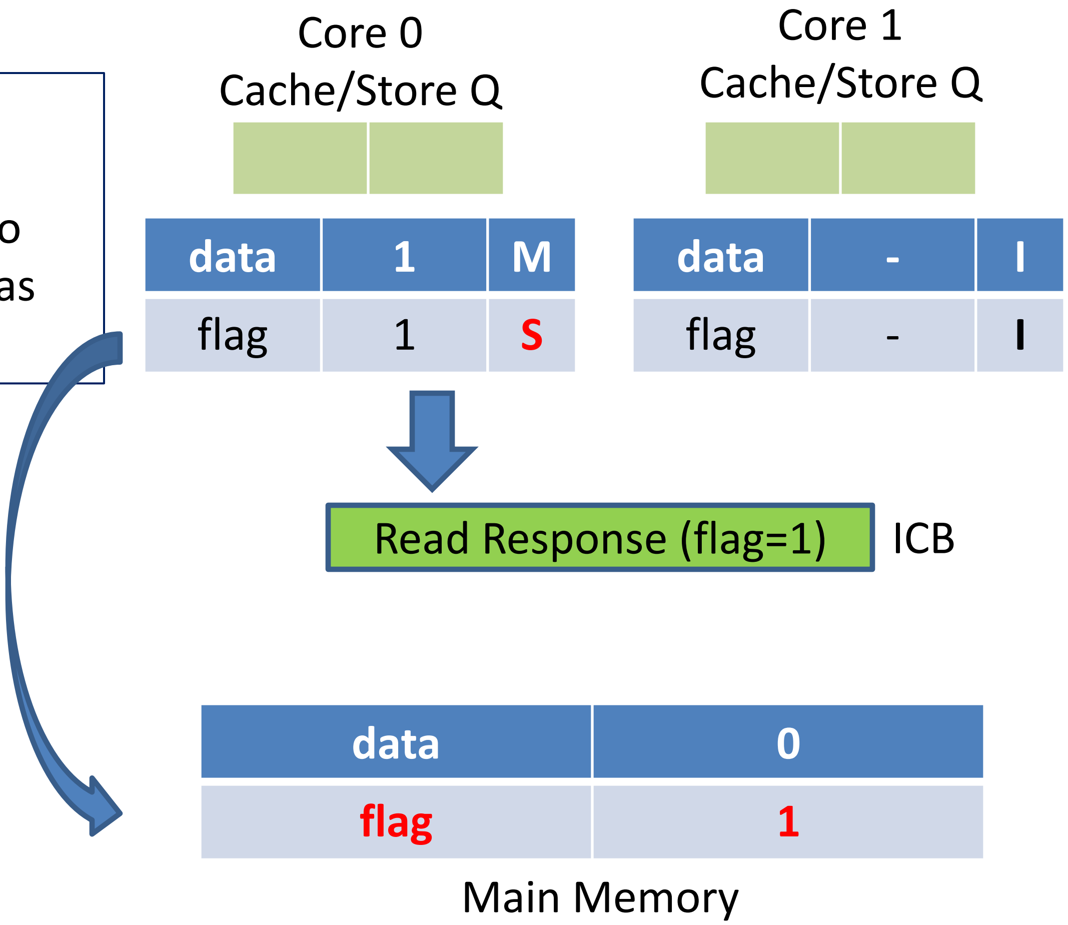
```
void bar()
{
   while (flag == 0);
   assert(data);
}
```

Store Q Issue Example (Fixed)

Core 0 respond to the read request of ‘flag’
The cache line is also sent to main memory and marked as ‘Shared’.

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0
Cache/Store Q

data	1	M
flag	1	S

Core 1
Cache/Store Q

data	-	I
flag	-	I

Read Response (flag=1)

ICB

data	0
flag	1

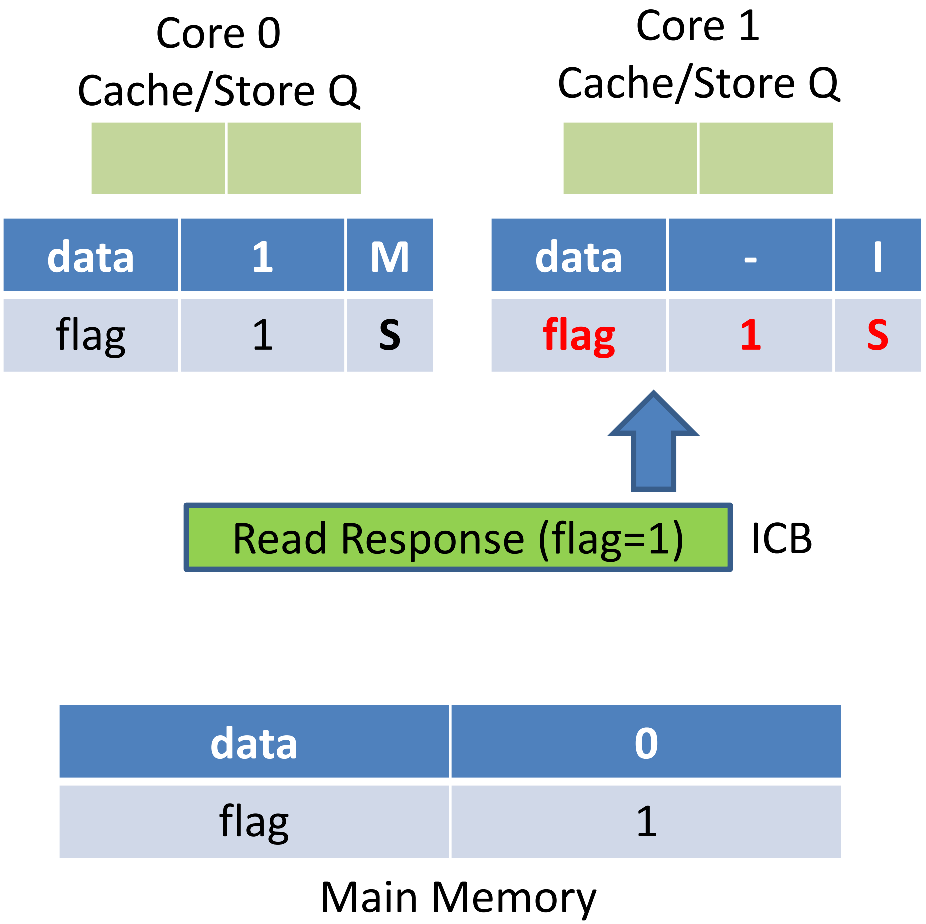
Main Memory

Store Q Issue Example (Fixed)

Core 1 receives the read response and marks the cache line as Shared.
Execution can now continue.
The same sequence plays out to fetch ‘data’… and all is well.

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0
Cache/Store Q

data	1	M
flag	1	S

Core 1
Cache/Store Q

data	-	I
flag	1	S

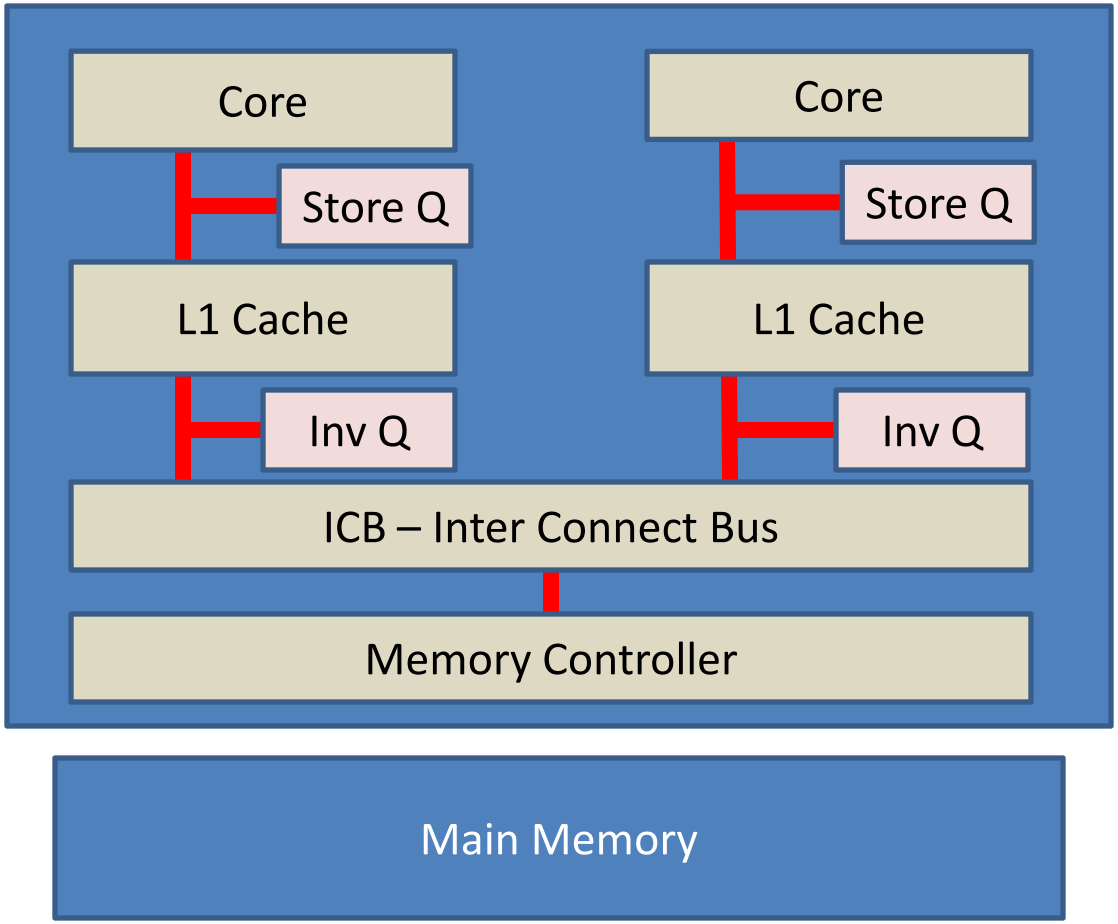
Read Response (flag=1)

ICB

data	0
flag	1

Main Memory

2-core CPU + Store Qs + Inv Q



Reasons for Invalidate Q

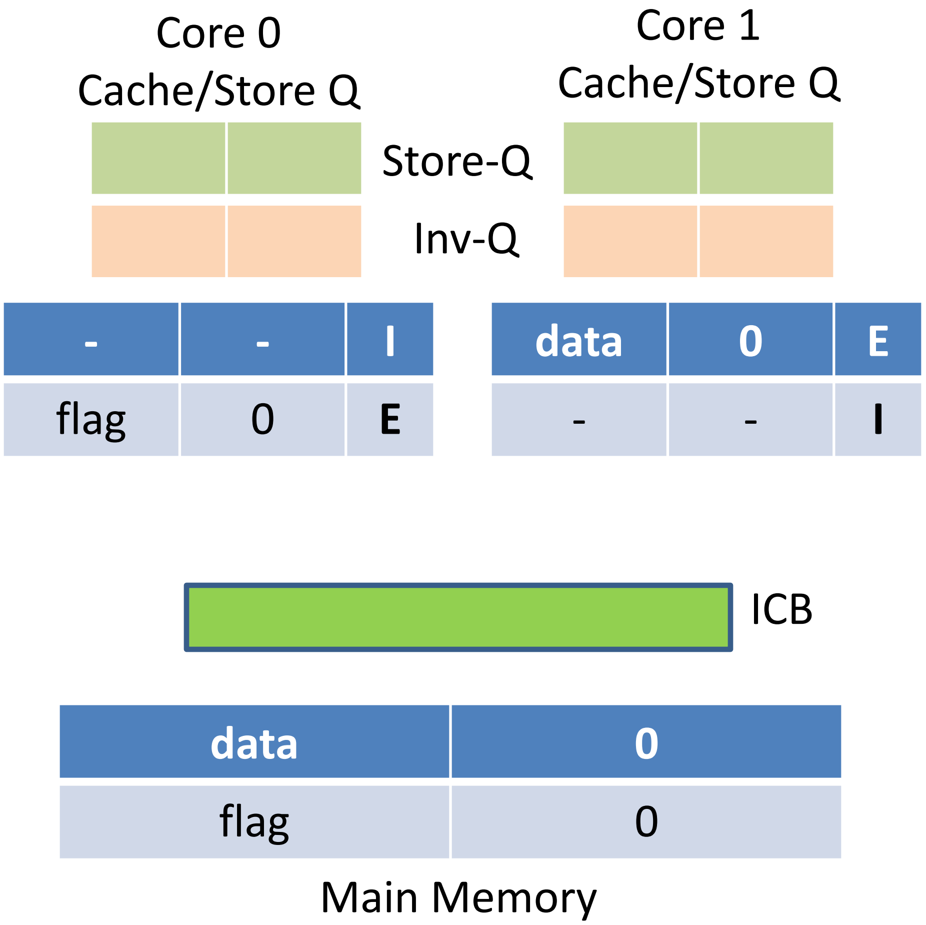
- Faster invalidate response from other cores
  - A busy core could take a while to reply
  - The ‘Invalid Acknowledge’ response cannot be sent until the cache has actually invalidated the cache line
- Contract: No MESI messages regarding that cache line will be sent by this core until all queued messages for that cache line have been processed

Invalidate Q Issue Example

Core 0 executes ‘foo’
Core 1 executes ‘bar’
‘flag’ cache line is owned by ‘0’
‘data’ cache line is owned by ‘1’

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

Core 1 Cache/Store Q

Store-Q

Inv-Q

Core 0 cache:

-	-	I
flag	0	E

Core 1 cache:

data	0	E
-	-	I

ICB

Main Memory

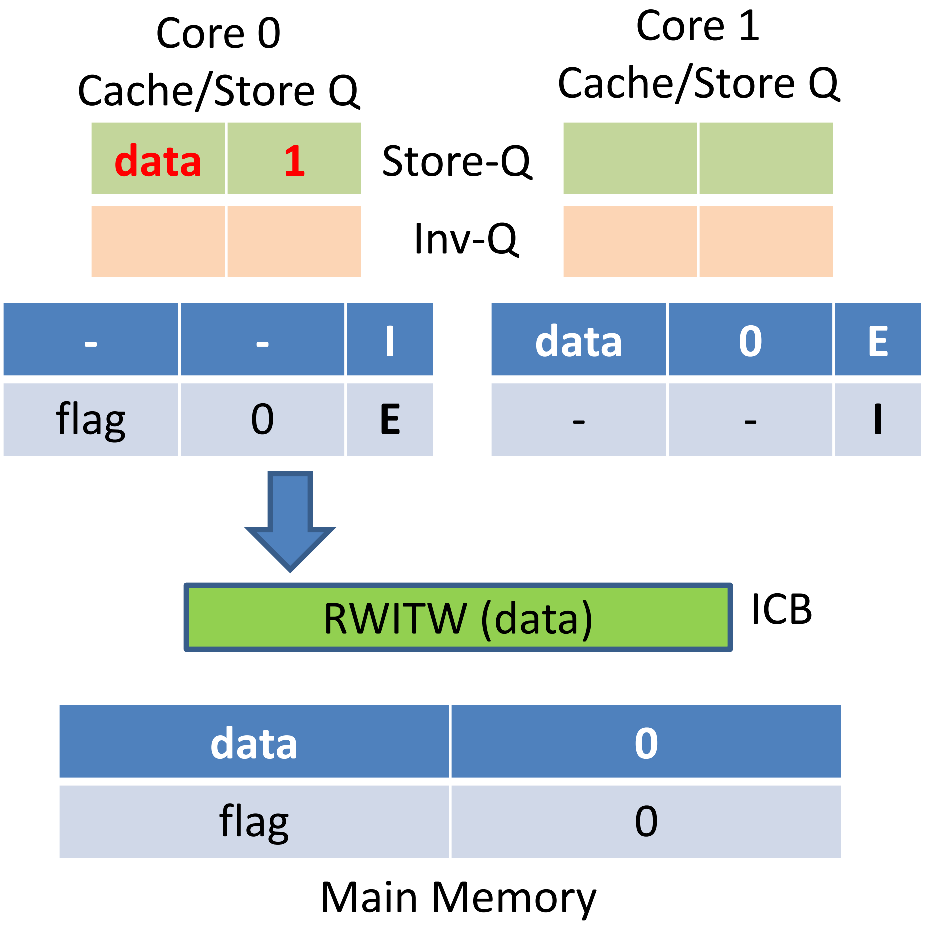
data	0
flag	0

Invalidate Q Issue Example

Core 0 does not have ‘data’ in the cache and requests the cache line. It saves the write in the Store Q pending the cache line

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

Core 1 Cache/Store Q

Store-Q

data	1		

Inv-Q

Core 0 cache:

-	-	I
flag	0	E

Core 1 cache:

data	0	E
-	-	I

RWITW (data)

ICB

data	0
flag	0

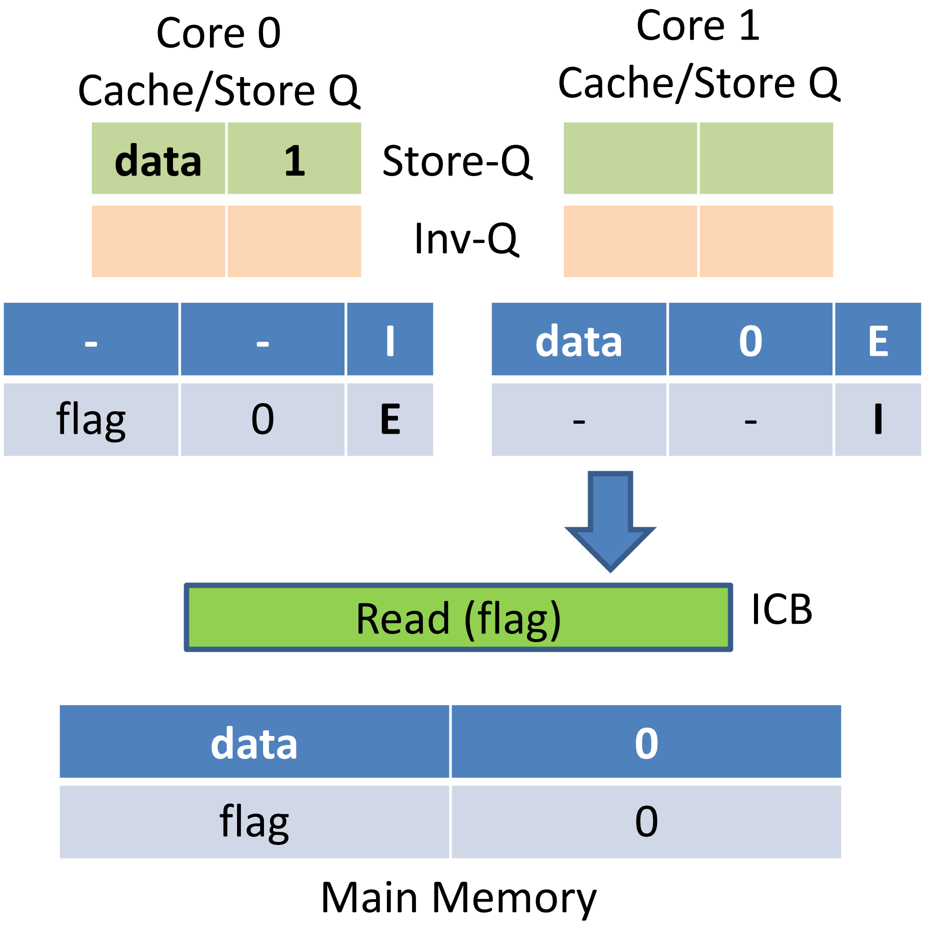
Main Memory

Invalidate Q Issue Example

Core 1 does not have 'flag' in the cache and issues a read request

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

Core 1 Cache/Store Q

Core 0			Core 1	
data	**1**	Store-Q		
		Inv-Q		

Core 0			Core 1		
-	**-**	**I**	**data**	**0**	**E**
flag	0	**E**	-	-	**I**

data	**0**
flag	0

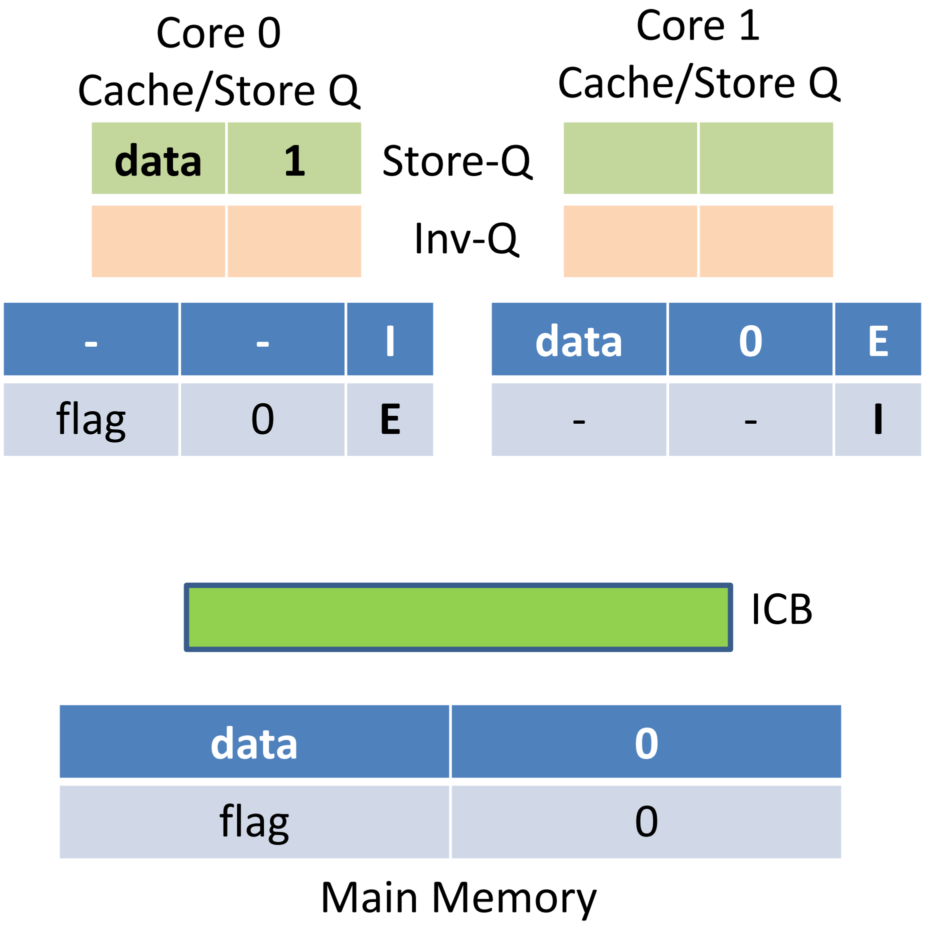
Main Memory

Invalidate Q Issue Example

Core 0 continues execution but stops on the memory barrier where it waits for all stores to complete

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

Store-Q

data	**1**

Inv-Q

-	**-**	**I**
flag	0	**E**

Core 1 Cache/Store Q

Store-Q

Inv-Q

data	**0**	**E**
-	-	**I**

ICB

data	**0**
flag	0

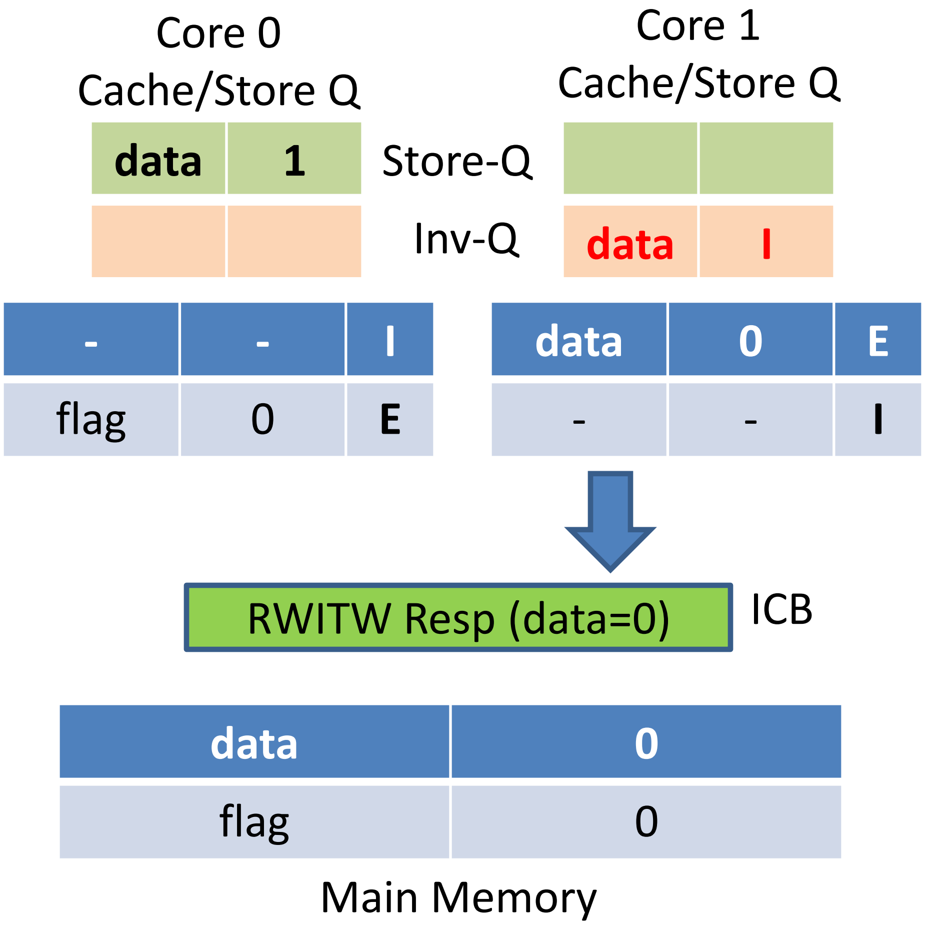
Main Memory

Invalidate Q Issue Example

Core 1 receives the 'Read + Invalidate' request. It replies with the cache line and records the Invalidate of 'data' in the Inv Q but doesn't invalidate it just yet.

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

data	1

Store-Q

Inv-Q

Core 1 Cache/Store Q

data	I

-	-	I
flag	0	E

data	0	E
-	-	I

RWITW Resp (data=0)

ICB

data	0
flag	0

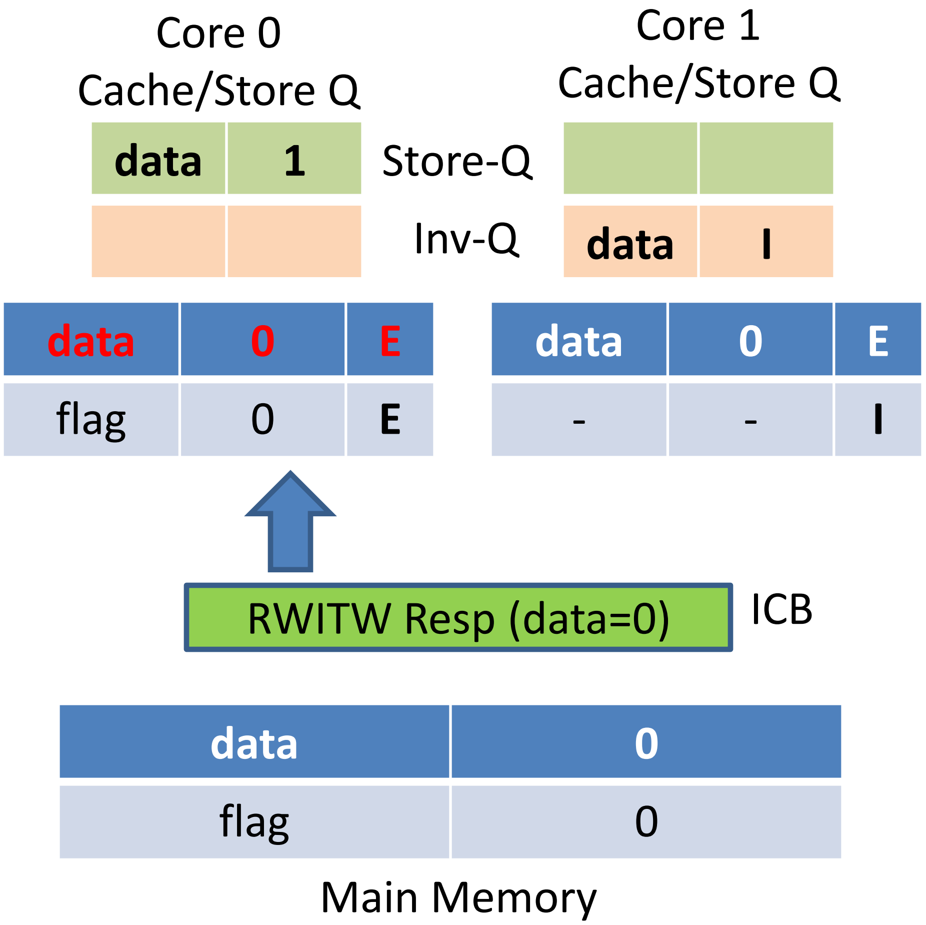
Main Memory

Invalidate Q Issue Example

Core 0 receives the 'data' cache line and installs it in the cache. As far as Core 0 is concerned all other cores have replied "Yes, I have invalidated 'data'" so it marks 'data' as 'Exclusive'

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

Core 1 Cache/Store Q

Core 0			Core 1	
data	1	Store-Q		
		Inv-Q	data	I

Core 0			Core 1		
data	0	E	data	0	E
flag	0	E	-	-	I

RWITW Resp (data=0)

ICB

data	0
flag	0

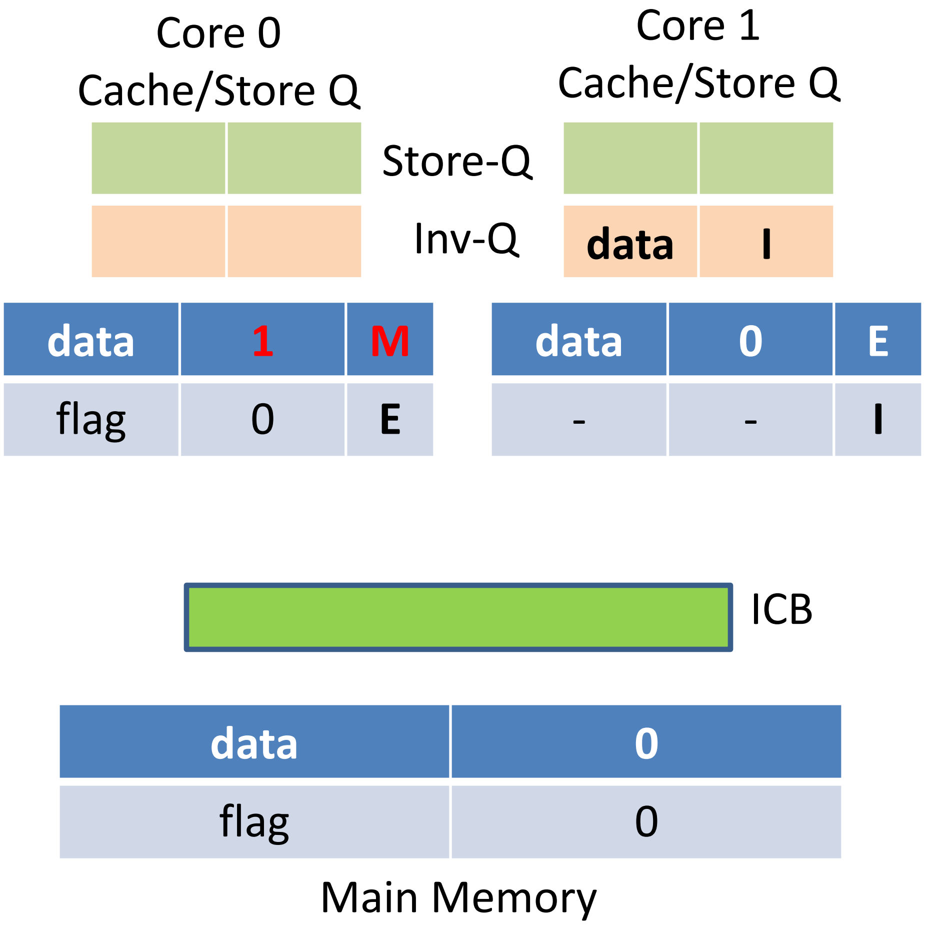
Main Memory

Invalidate Q Issue Example

Core 0 commits the Store Q to the cache and marks ‘data’ as Modified. It is now free to continue executing

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

Store-Q

Inv-Q

data	1	M
flag	0	E

Core 1 Cache/Store Q

Store-Q

Inv-Q

data	I

data	0	E
-	-	I

ICB

data	0
flag	0

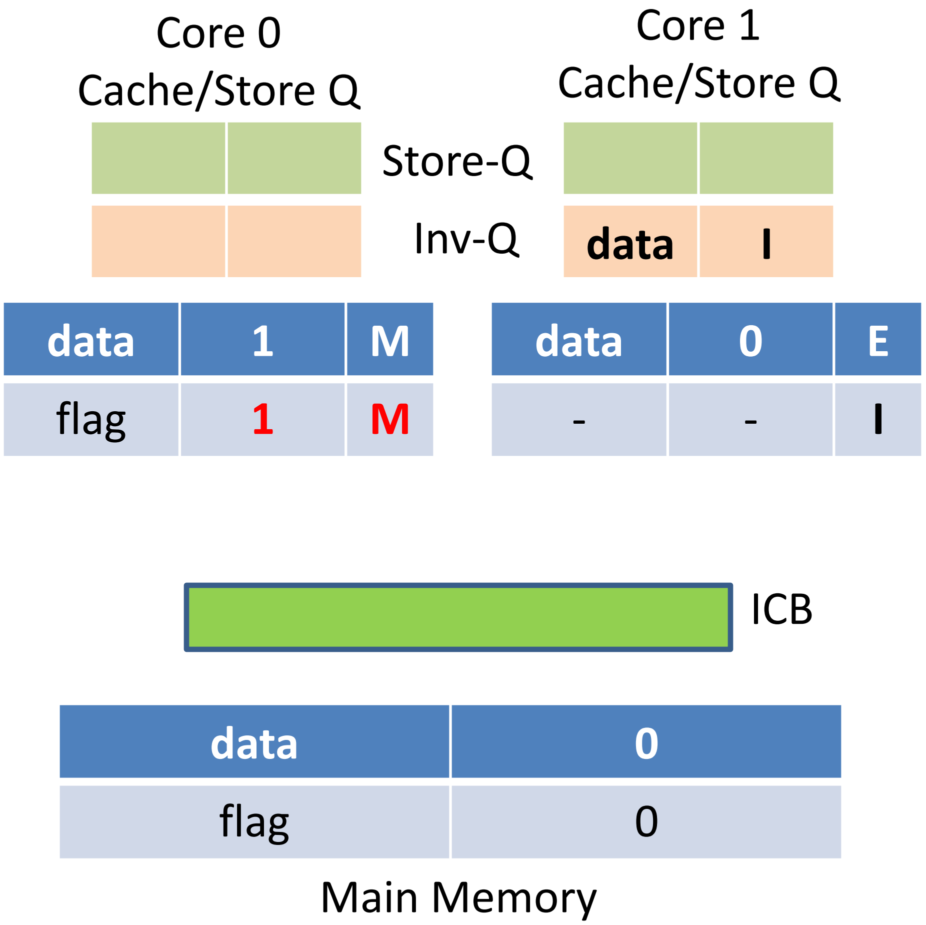
Main Memory

Invalidate Q Issue Example

Core 0 now updates ‘flag’ directly as it is marked as ‘Exclusive’.

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

Core 1 Cache/Store Q

	Core 0		Core 1	
Store-Q				
Inv-Q			data	I

data	1	M
flag	1	M

data	0	E
-	-	I

ICB

data	0
flag	0

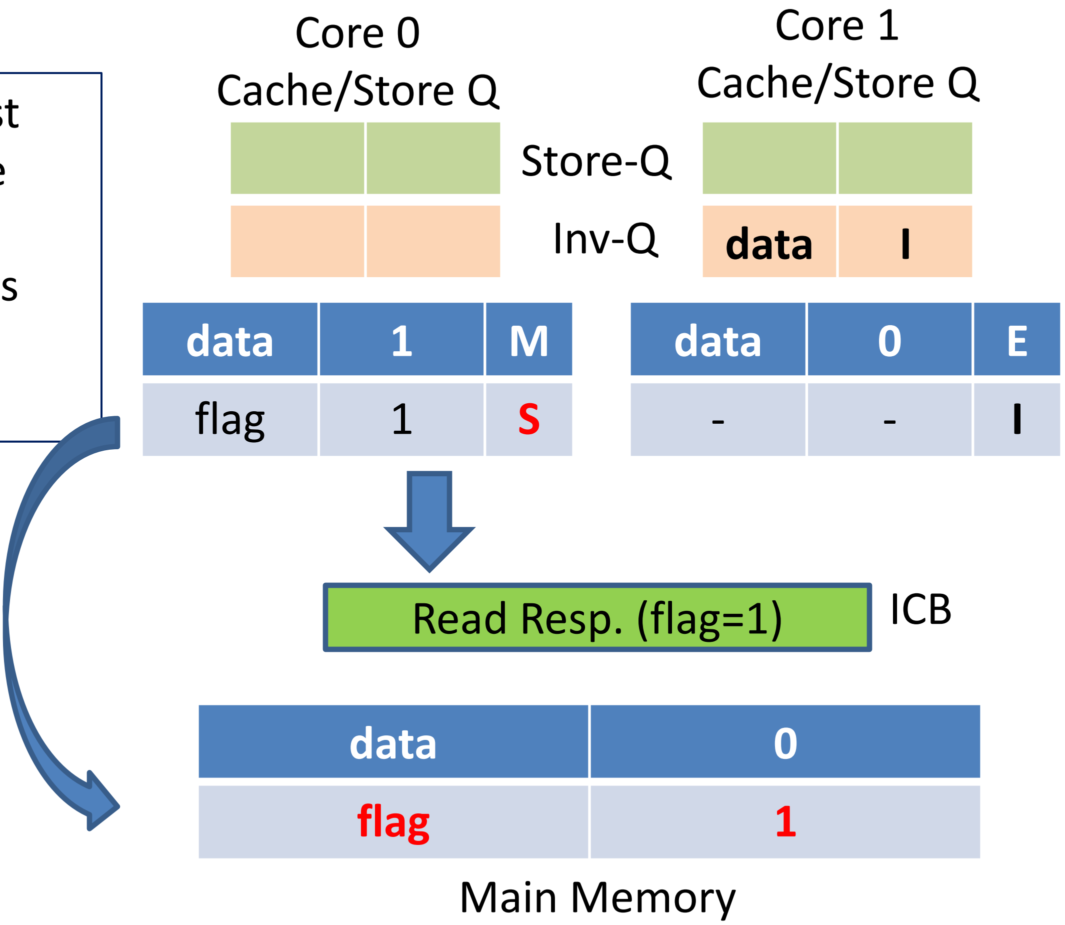
Main Memory

Invalidate Q Issue Example

Core 0 receives read request for ‘flag’. Because the cache line is modified it triggers a ‘writeback’ and then returns the now updated value and marks ‘flag’ as ‘Shared’

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

Store-Q		
Inv-Q		

data	1	M
flag	1	S

Core 1 Cache/Store Q

Store-Q		
Inv-Q	data	I

data	0	E
-	-	I

Read Resp. (flag=1)

ICB

Main Memory

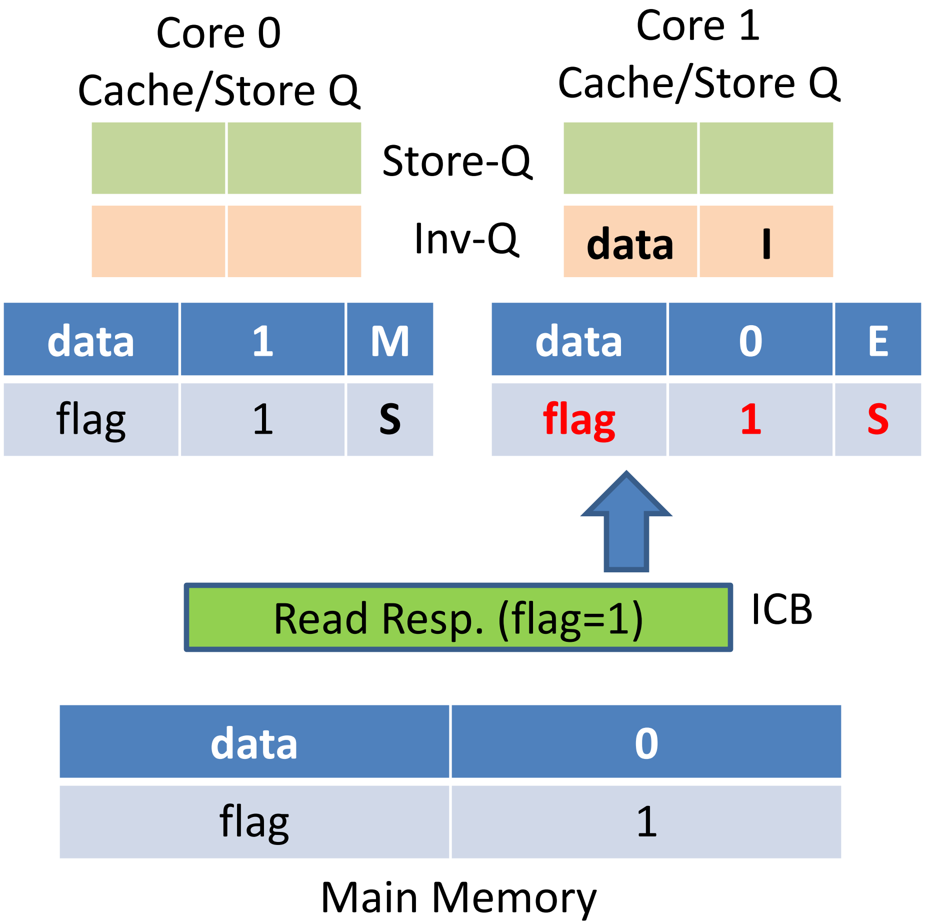
data	0
flag	1

Invalidate Q Issue Example

Core 1 receives the ‘flag’ cache line and installs it in the cache as ‘Shared’

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0 Cache/Store Q

Core 1 Cache/Store Q

Store-Q

Core 0 Store-Q		Core 1 Store-Q	

Inv-Q

Core 0 Inv-Q		Core 1 Inv-Q	
		data	I

Core 0		
data	1	M
flag	1	S

Core 1		
data	0	E
flag	1	S

Read Resp. (flag=1)

ICB

data	0
flag	1

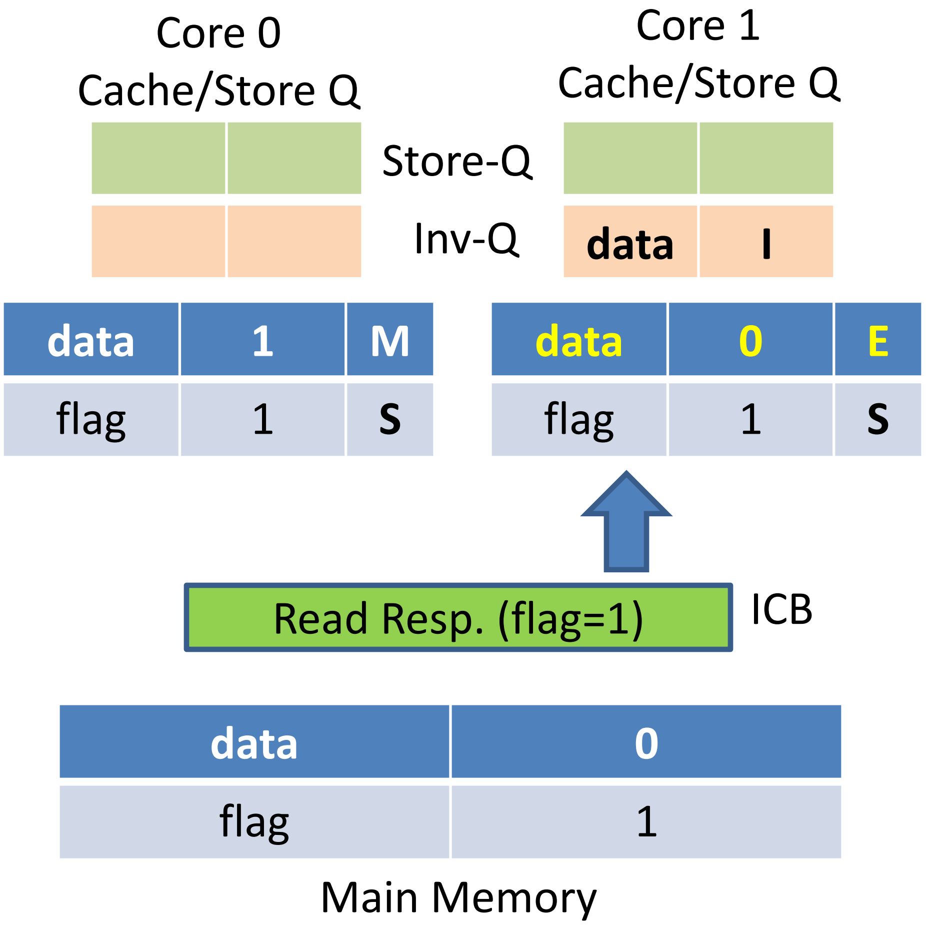
Main Memory

Invalidate Q Issue Example

Core 1 can now continue execution. The ‘data’ cache line is in the cache and valid and the stale value is read.
Note: No MESI msg was sent so the contract is upheld.

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



	Core 0 Cache/Store Q		Core 1 Cache/Store Q	
Store-Q				
Inv-Q			data	I

Core 0:

data	1	M
flag	1	S

Core 1:

data	0	E
flag	1	S

data	0
flag	1

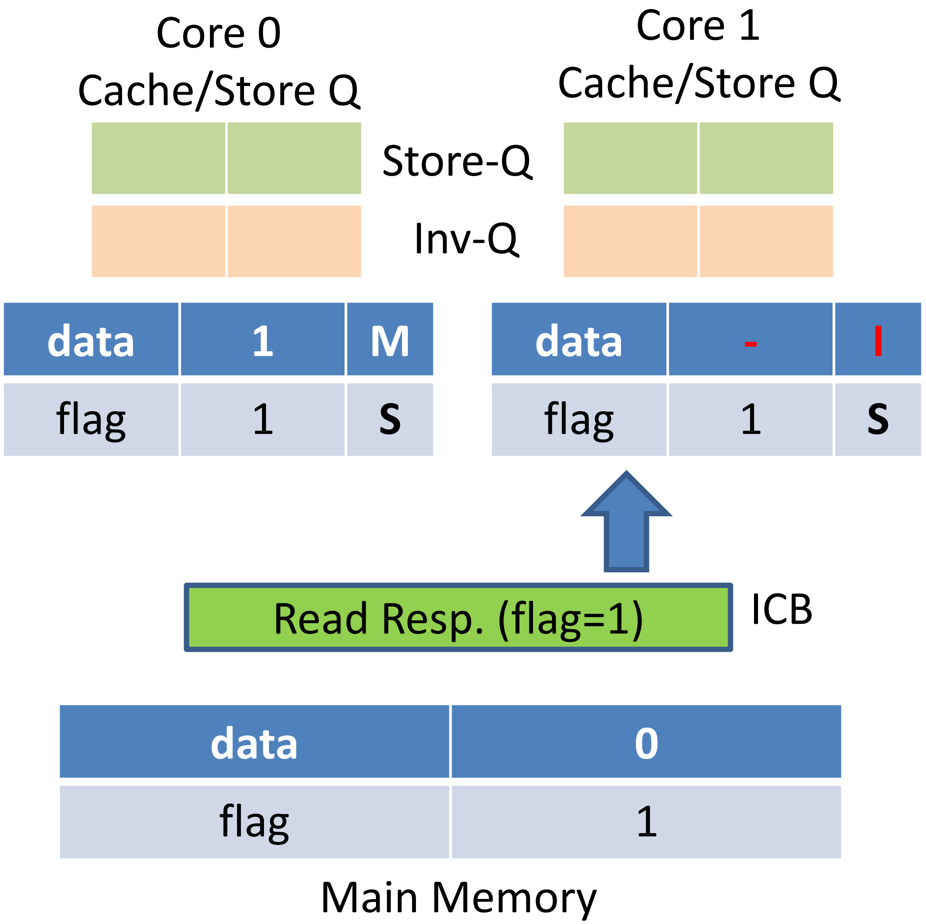
Main Memory

Invalidate Q Issue Example

Core 1 finally applies the ‘Invalidate’ from the Inv Q but it is too late.
CRASH!

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   assert(data);
}
```



Core 0
Cache/Store Q

Store-Q

Inv-Q

Core 1
Cache/Store Q

data	1	M
flag	1	S

data	-	I
flag	1	S

Read Resp. (flag=1)

ICB

data	0
flag	1

Main Memory

How do we solve this issue?

- This time the local core isn’t using all information it has when servicing the read
  - Why?
    - Speed, speed, speed
- Is there a way for us to force the core to use all information?
  - Yes, we can flush the ‘Invalidate Q’
  - Memory Load Barriers (__mb_acquire)

Memory Load Barriers

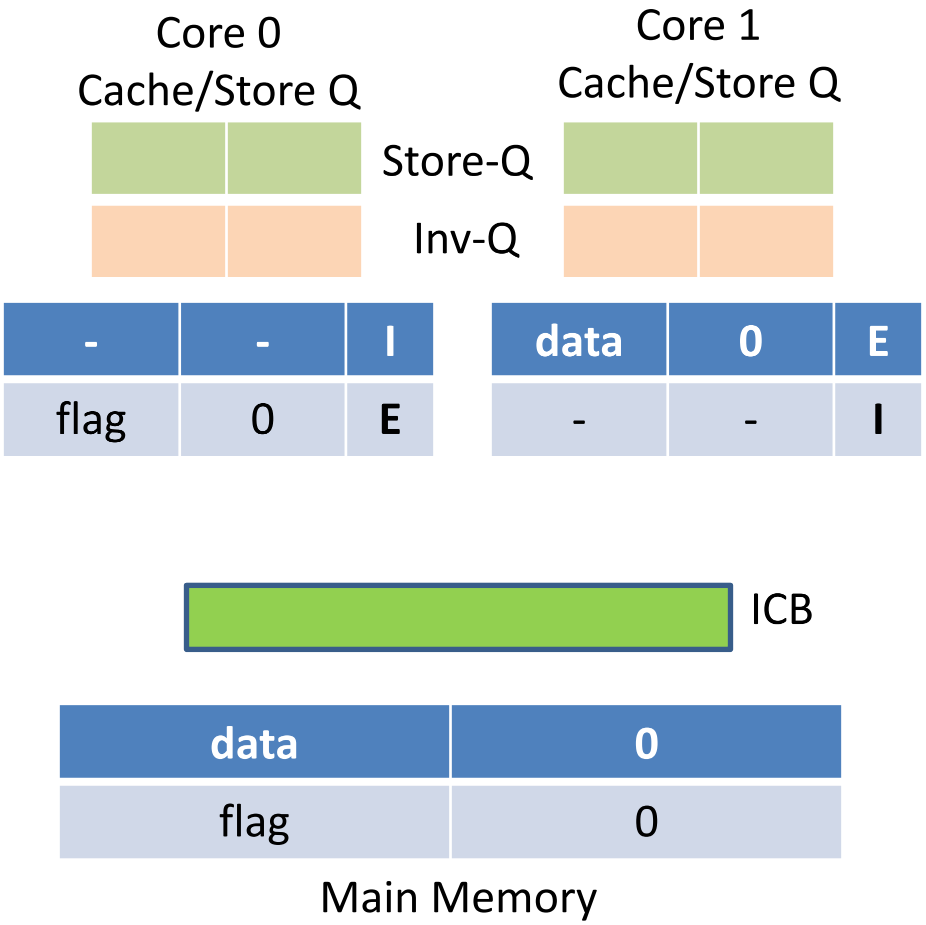
- CPU instruction
- All messages in the Invalidate Q is processed
- All loads preceding the barrier will complete
  - Did I mention that CPUs are evil!
- Prevents compilers from optimize memory loads across this barrier.
  - Compilers are evil!
- Guarantees that data read after the barrier will be freshly pulled from other caches/main memory
  - Stale cache lines are effectively evicted

Invalidate Q Issue Example (Fixed)

Core 0 executes ‘foo’
Core 1 executes ‘bar’
‘flag’ cache line is owned by ‘0’
‘data’ cache line is owned by ‘1’

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```



Core 0 cache:

-	-	I
flag	0	E

Core 1 cache:

data	0	E
-	-	I

data	0
flag	0

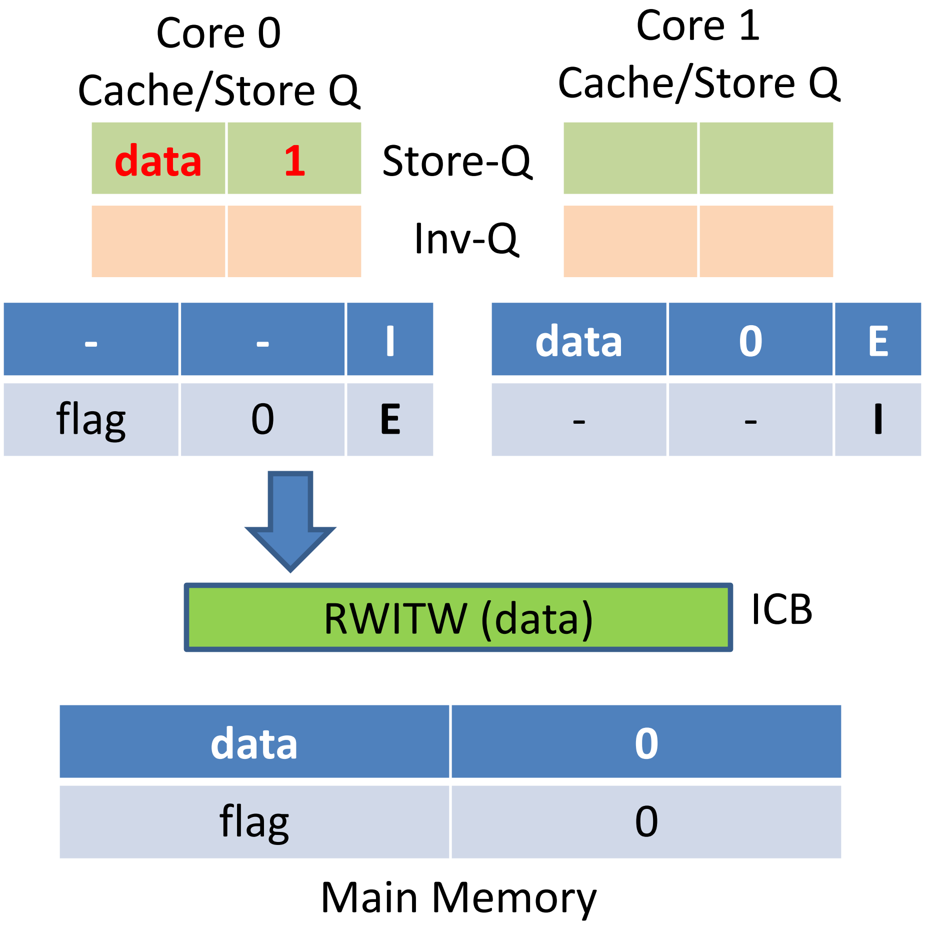
Main Memory

Invalidate Q Issue Example (Fixed)

Core 0 does not have ‘data’ in the cache and requests the cache line. It saves the write in the Store Q pending the cache line

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```



Core 0 Cache/Store Q

data	1

Store-Q

Inv-Q

Core 1 Cache/Store Q

Core 0 cache:

-	-	I
flag	0	E

Core 1 cache:

data	0	E
-	-	I

RWITW (data)

ICB

data	0
flag	0

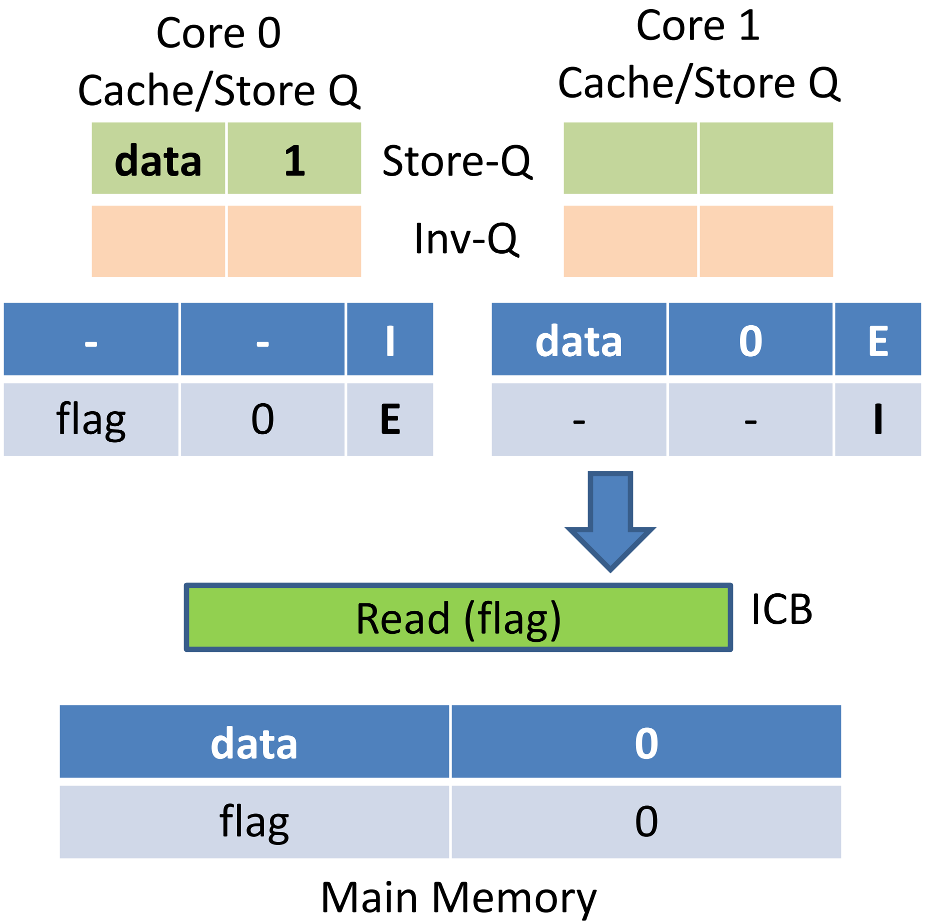
Main Memory

Invalidate Q Issue Example (Fixed)

Core 1 does not have ‘flag’ in the cache and issues a read request

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```



Core 0 Cache/Store Q			Core 1 Cache/Store Q	
data	1	Store-Q		
		Inv-Q		

Core 0			Core 1		
-	-	I	data	0	E
flag	0	E	-	-	I

ICB: Read (flag)

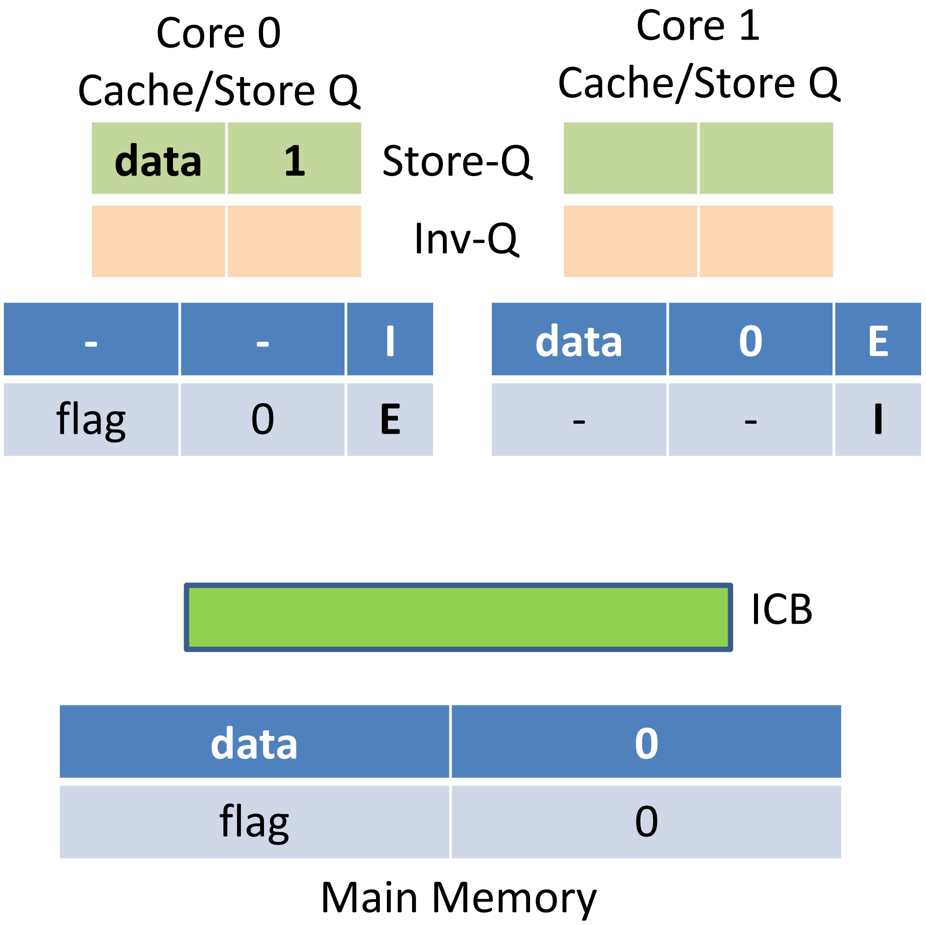
Main Memory	
data	0
flag	0

Invalidate Q Issue Example (Fixed)

Core 0 continues execution but stops on the memory barrier where it waits for all stores to complete

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```



Core 0 Cache/Store Q

Store-Q	data	1
Inv-Q		

-	-	I
flag	0	E

Core 1 Cache/Store Q

Store-Q		
Inv-Q		

data	0	E
-	-	I

ICB

Main Memory

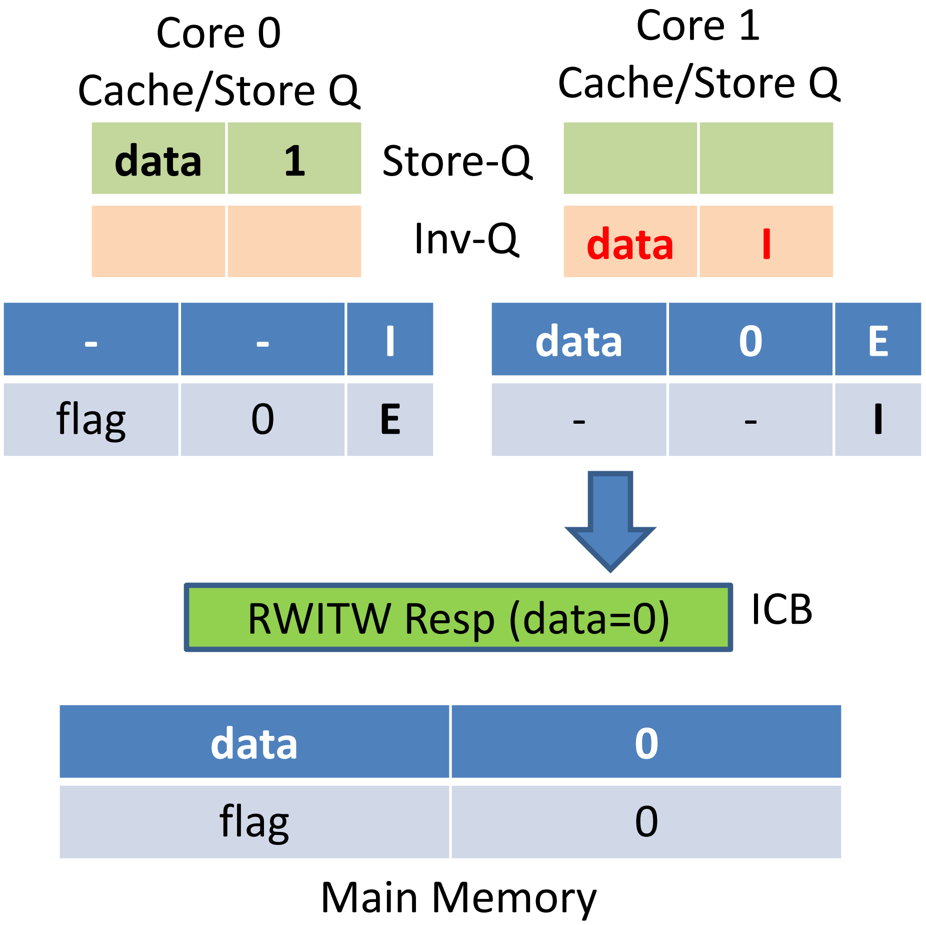
data	0
flag	0

Invalidate Q Issue Example (Fixed)

Core 1 receives the Read + Invalidate request. It replies with the cache line and records the Invalidate of ‘data’ in the Inv Q but doesn’t invalidate it just yet.

```
void foo()
{
    data = 1;
    __mb_release();
    flag = 1;
}
```

```
void bar()
{
    while (flag == 0);
    __mb_acquire();
    assert(data);
}
```



Core 0 Cache/Store Q			Core 1 Cache/Store Q	
data	1	Store-Q		
		Inv-Q	data	I

-	-	I	data	0	E
flag	0	E	-	-	I

RWITW Resp (data=0) ICB

data	0
flag	0

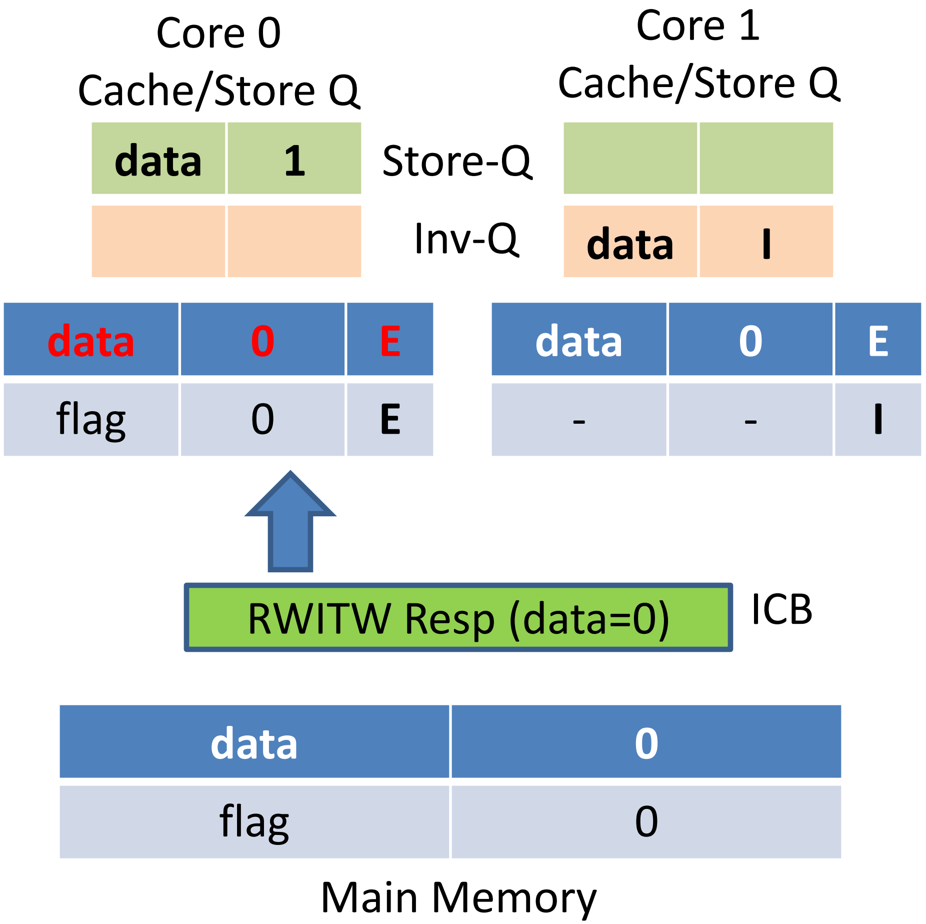
Main Memory

Invalidate Q Issue Example (Fixed)

Core 0 receives the 'data' cache line and installs it in the cache. As far as Core 0 is concerned all other cores have replied "Yes, I have invalidated 'data'" so it marks 'data' as 'Exclusive'

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```



Core 0 Cache/Store Q			Core 1 Cache/Store Q	
data	1	Store-Q		
		Inv-Q	data	I

data	0	E	data	0	E
flag	0	E	-	-	I

RWITW Resp (data=0)

ICB

data	0
flag	0

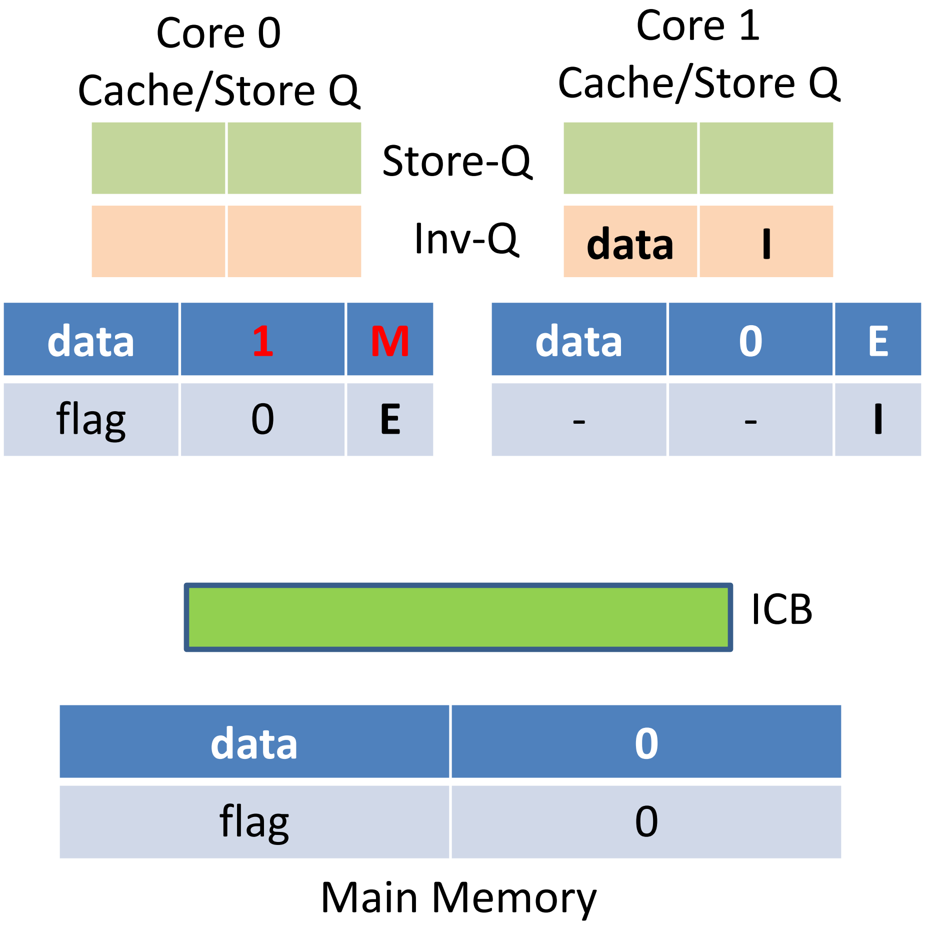
Main Memory

Invalidate Q Issue Example (Fixed)

Core 0 commits the Store Q to the cache and marks ‘data’ as ‘Modified’. It is now free to continue executing

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```



Core 0 Cache/Store Q

Store-Q		
Inv-Q		

data	1	M
flag	0	E

Core 1 Cache/Store Q

Store-Q		
Inv-Q	data	I

data	0	E
-	-	I

data	0
flag	0

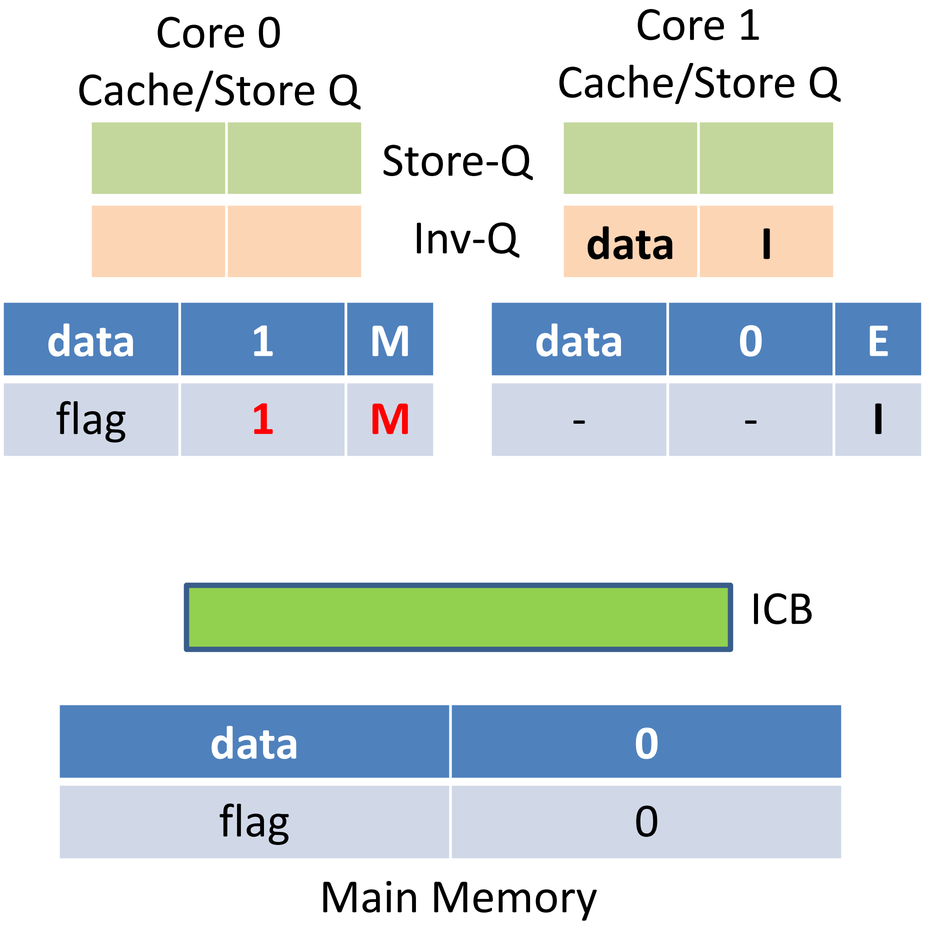
Main Memory

Invalidate Q Issue Example (Fixed)

Core 0 now updates ‘flag’ directly as it is marked as ‘Exclusive’.

```
void foo()
{
    data = 1;
    __mb_release();
    flag = 1;
}
```

```
void bar()
{
    while (flag == 0);
    __mb_acquire();
    assert(data);
}
```



Core 0 Cache/Store Q

Store-Q		
Inv-Q		

data	1	M
flag	1	M

Core 1 Cache/Store Q

Store-Q		
Inv-Q	data	I

data	0	E
-	-	I

ICB

Main Memory

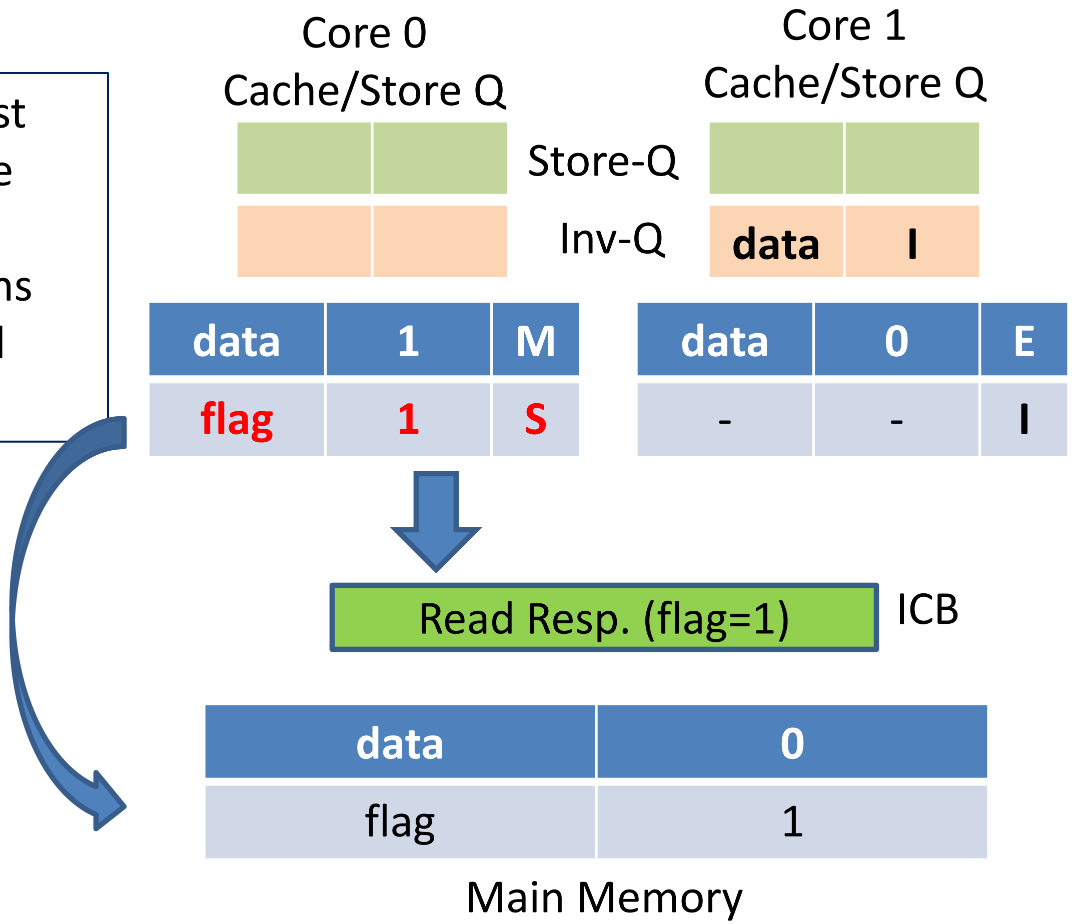
data	0
flag	0

Invalidate Q Issue Example (Fixed)

Core 0 receives read request for ‘flag’. Because the cache line is modified it triggers a ‘Writeback’ and then returns the now updated value and marks ‘flag’ as ‘Shared’

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```



Core 0 Cache/Store Q

Store-Q	
Inv-Q	

data	1	M
flag	1	S

Core 1 Cache/Store Q

Store-Q	
Inv-Q	data I

data	0	E
-	-	I

Read Resp. (flag=1)

ICB

data	0
flag	1

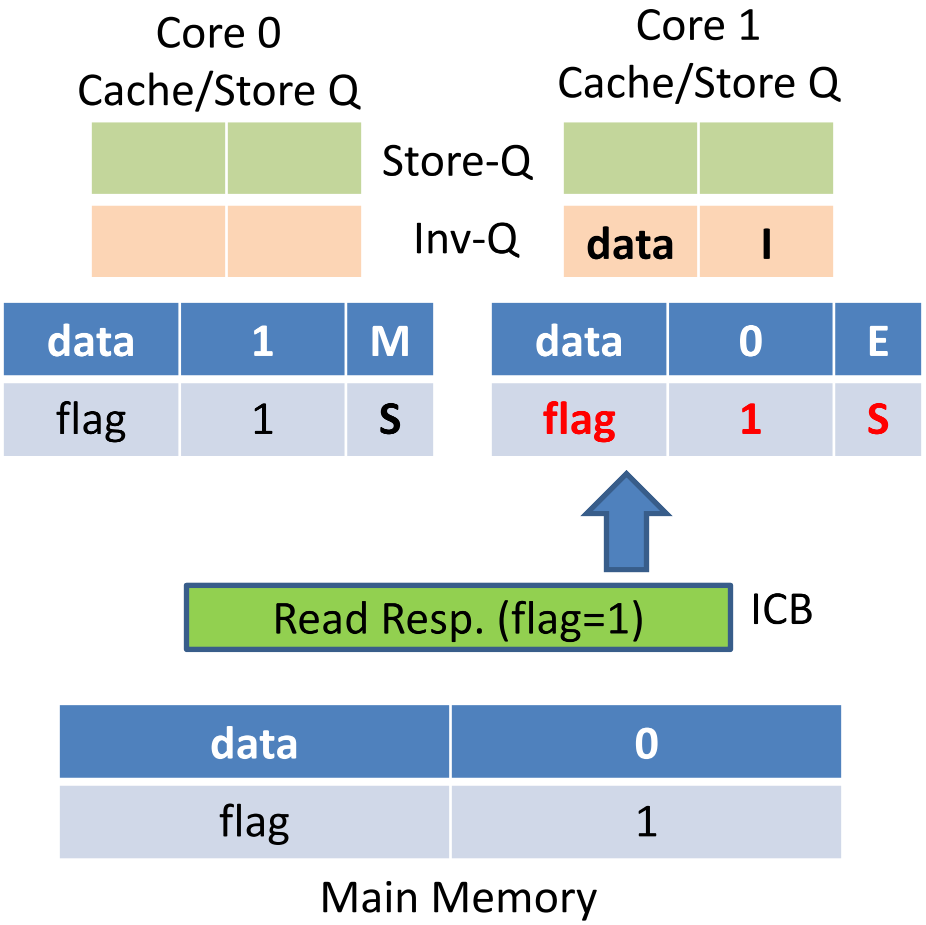
Main Memory

Invalidate Q Issue Example (Fixed)

Core 1 receives the ‘flag’ cache line and installs it in the cache as ‘Shared’

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```



	Core 0 Cache/Store Q		Core 1 Cache/Store Q	
Store-Q				
Inv-Q			data	I

Core 0

data	1	M
flag	1	S

Core 1

data	0	E
flag	1	S

Main Memory

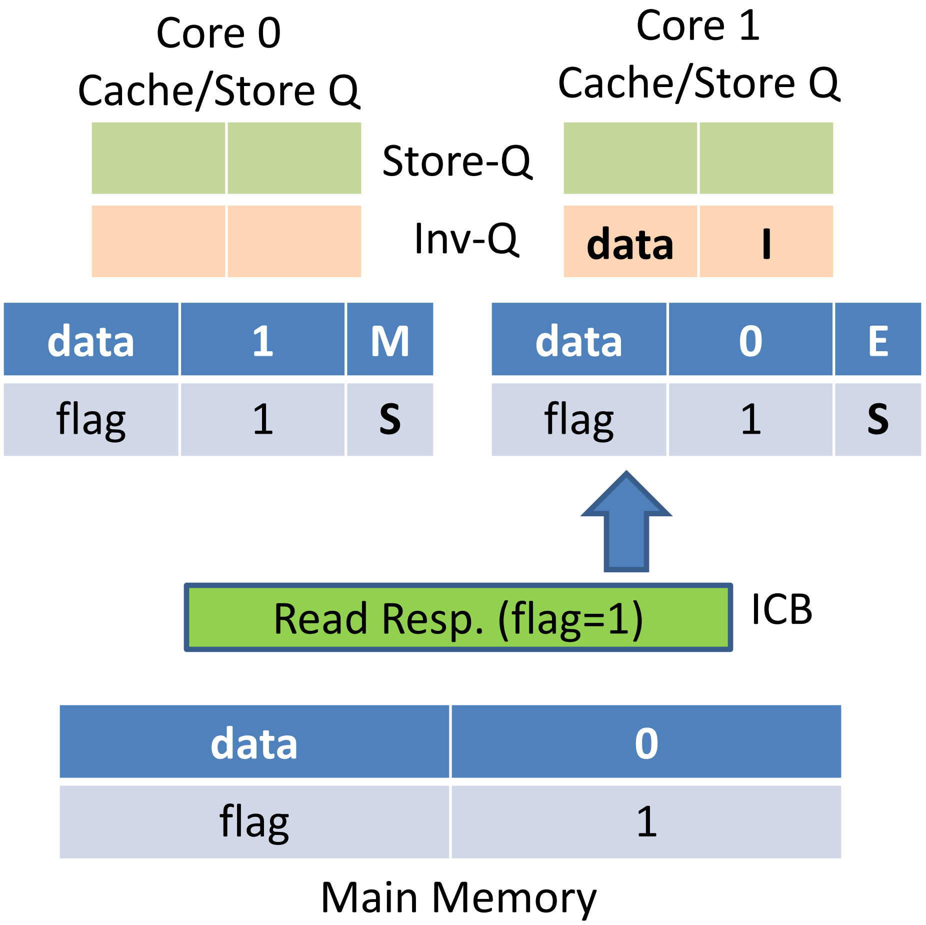
data	0
flag	1

Invalidate Q Issue Example (Fixed)

Core 1 can now continue execution. It arrives at the memory barrier and now waits for the 'Invalidate Q' to clear.

```
void foo()
{
    data = 1;
    __mb_release();
    flag = 1;
}
```

```
void bar()
{
    while (flag == 0);
    __mb_acquire();
    assert(data);
}
```



Core 0 Cache/Store Q

Store-Q	
Inv-Q	

data	1	M
flag	1	S

Core 1 Cache/Store Q

Store-Q	
Inv-Q	data I

data	0	E
flag	1	S

Read Resp. (flag=1)

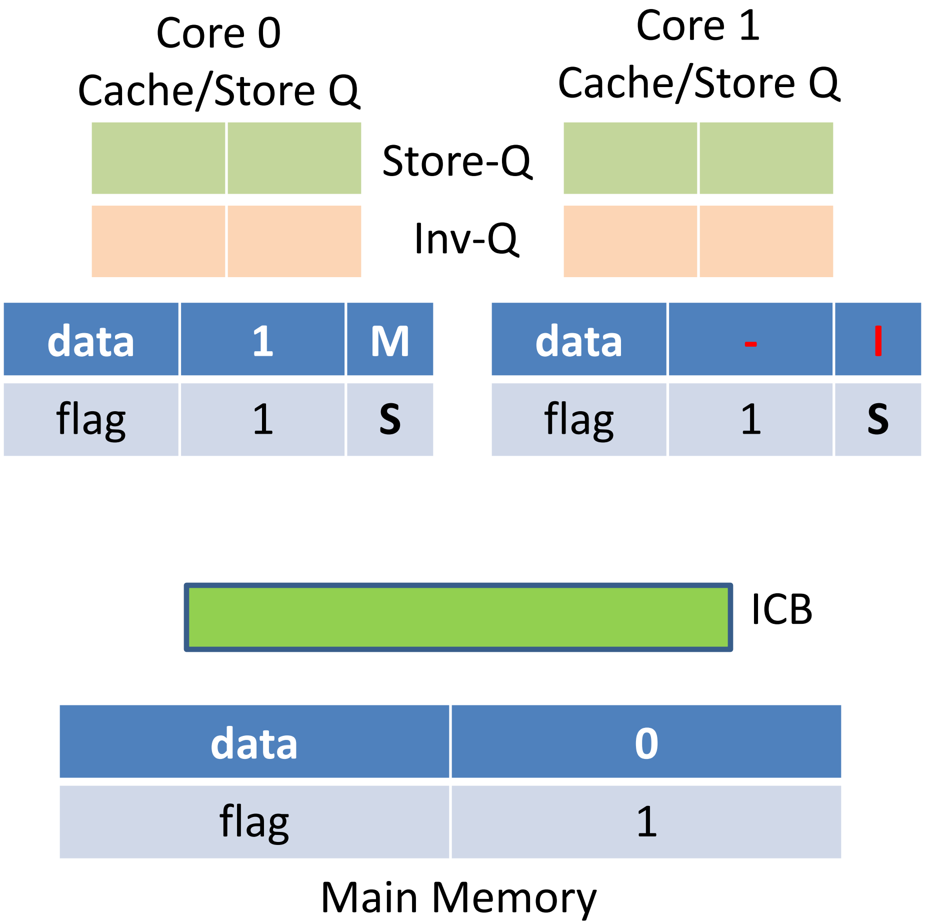
ICB

data	0
flag	1

Main Memory

Invalidate Q Issue Example (Fixed)

Core 1 process the invalidate Q and marks the 'data' cache line as invalid.



Core 0 Cache/Store Q

data	1	M
flag	1	S

Core 1 Cache/Store Q

data	-	I
flag	1	S

Main Memory

data	0
flag	1

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

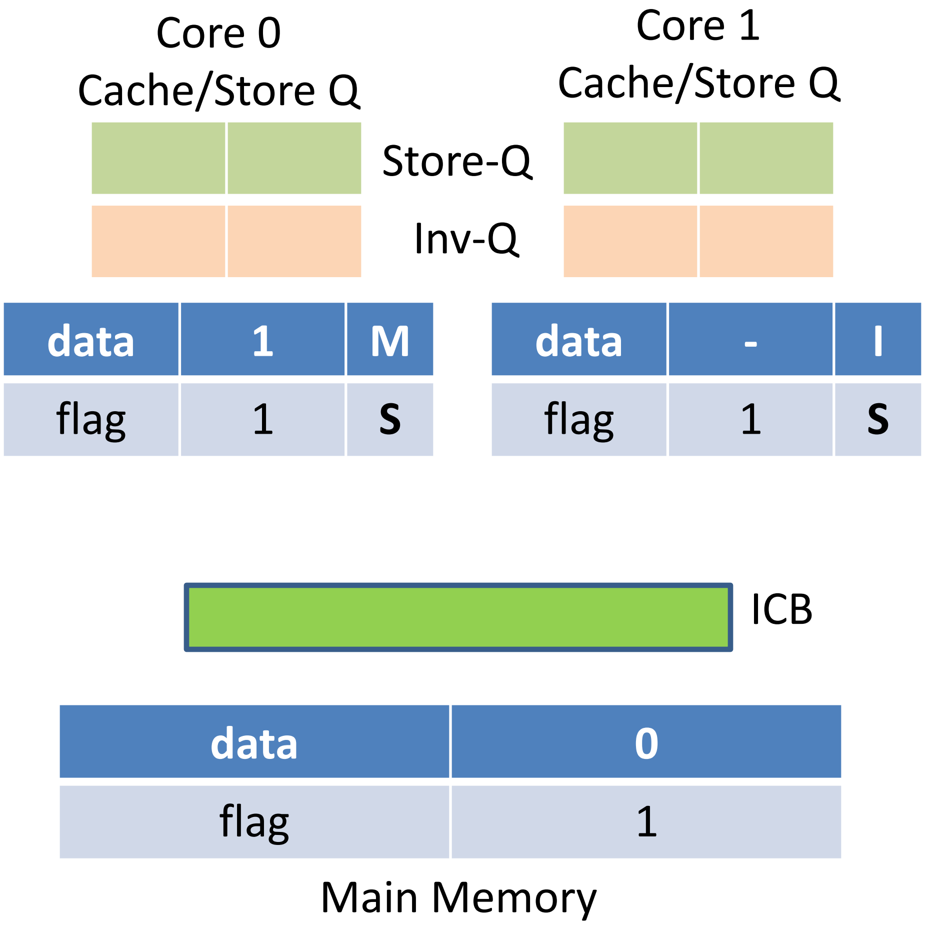
```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```

Invalidate Q Issue Example (Fixed)

Core 1 can now continue executing as the memory barrier has waited for the Inv Q to clear and loads to complete.

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```



Core 0 Cache/Store Q

Core 1 Cache/Store Q

Store-Q

Inv-Q

Core 0:

data	1	M
flag	1	S

Core 1:

data	-	I
flag	1	S

ICB

data	0
flag	1

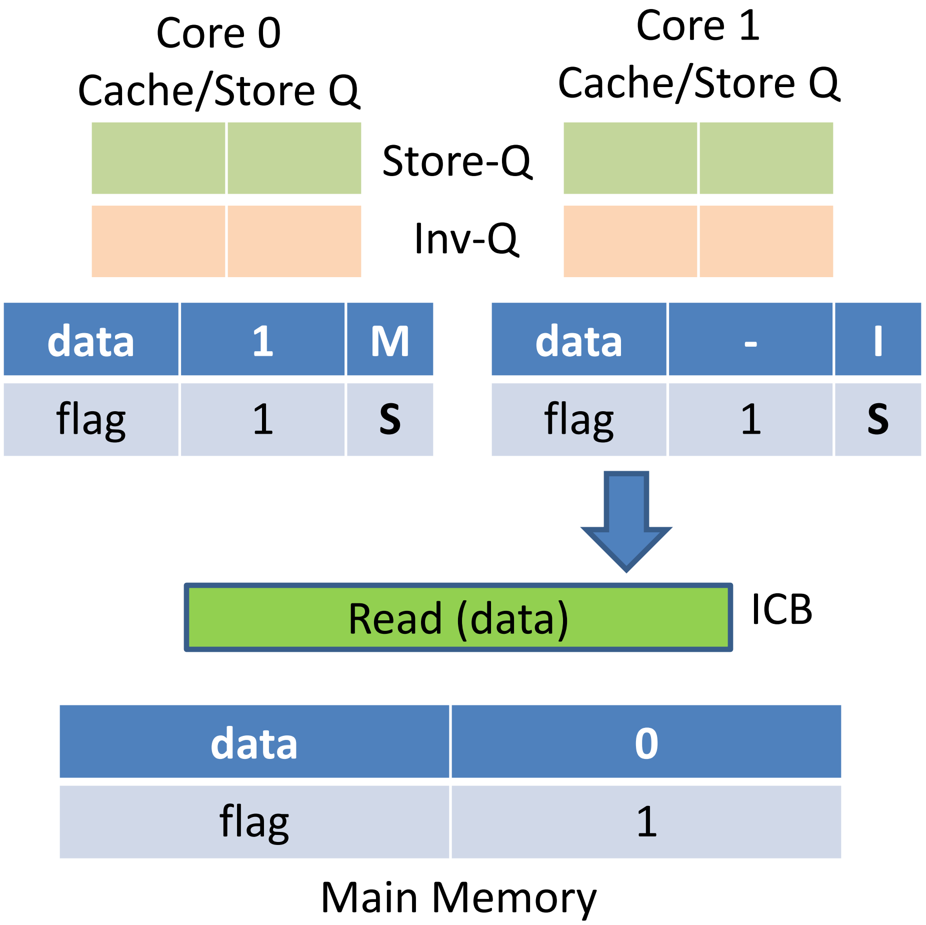
Main Memory

Invalidate Q Issue Example (Fixed)

Core 1 now have to request the cache line for 'data' as it is marked as 'Invalid'.

```
void foo()
{
    data = 1;
    __mb_release();
    flag = 1;
}
```

```
void bar()
{
    while (flag == 0);
    __mb_acquire();
    assert(data);
}
```



Core 0
Cache/Store Q

Core 1
Cache/Store Q

Store-Q

Inv-Q

data	1	M
flag	1	S

data	-	I
flag	1	S

Read (data)

ICB

data	0
flag	1

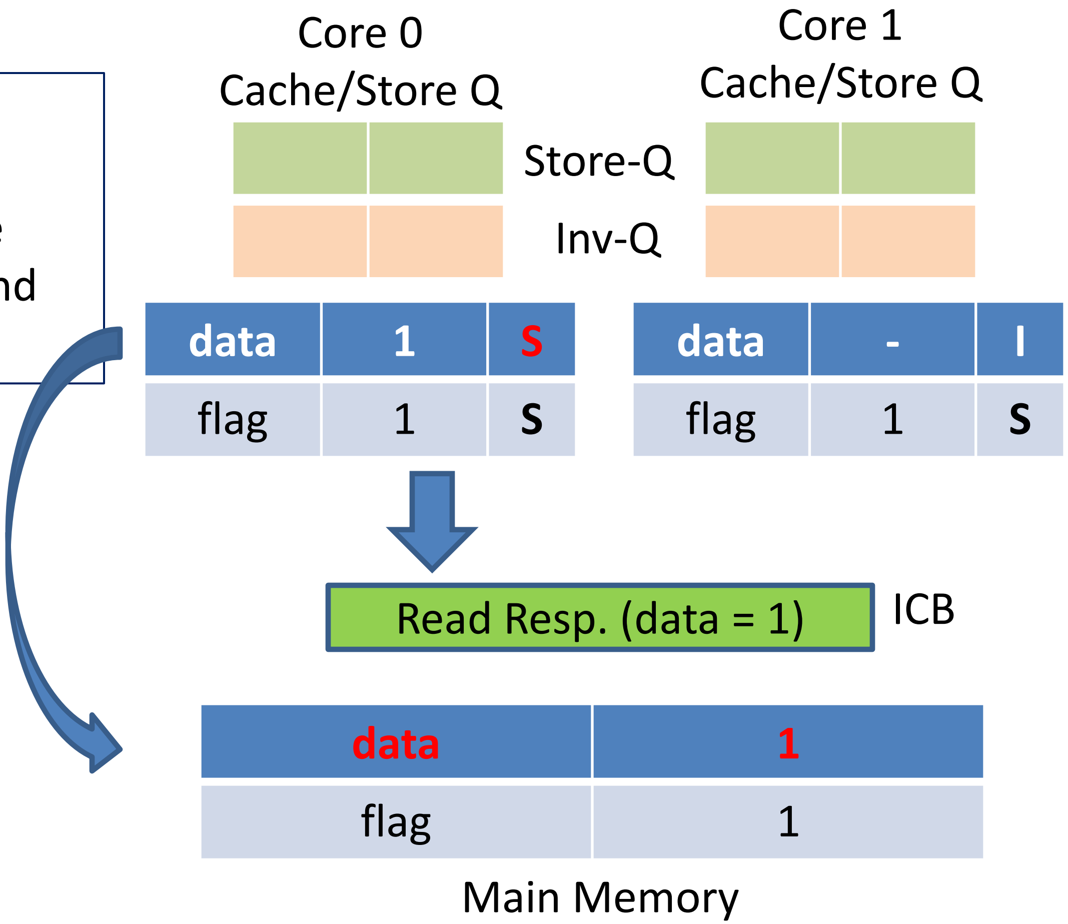
Main Memory

Invalidate Q Issue Example (Fixed)

Core 0 receives the read request. This triggers a 'Writeback' followed by the cache line being sent out and marked as 'Shared'

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```



Core 0 Cache/Store Q

data	1	S
flag	1	S

Core 1 Cache/Store Q

data	-	I
flag	1	S

Main Memory

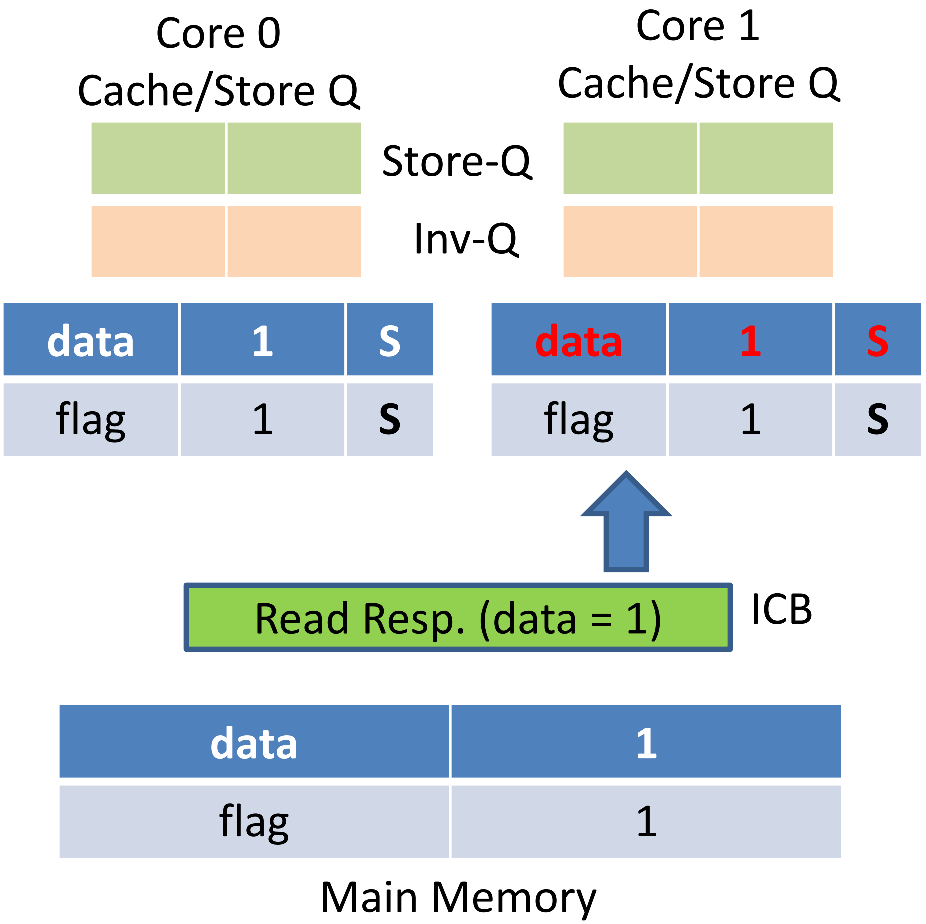
data	1
flag	1

Invalidate Q Issue Example (Fixed)

Core 1 finally receives the ‘data’ cache line and execution continues past the assert and life is good

```
void foo()
{
    data = 1;
    __mb_release();
    flag = 1;
}
```

```
void bar()
{
    while (flag == 0);
    __mb_acquire();
    assert(data);
}
```



Core 0
Cache/Store Q

Core 1
Cache/Store Q

Store-Q

Inv-Q

data	1	S
flag	1	S

data	1	S
flag	1	S

Read Resp. (data = 1)

ICB

data	1
flag	1

Main Memory

The Solutions (Memory Barriers)

- Two types
  - Release Semantic
    - Flush Store Q
    - Producer behavior
    - Prevent compiler from moving stores across barrier

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

  - Acquire Semantic
    - Flush Cache Invalidate Q
    - Prior Loads Complete
    - Consumer behavior
    - Prevent compiler from moving loads across barrier

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```

Memory Barriers – Release Semantics

- Prevents reordering of writes by compiler *or* CPU
  - Used when handing out access to data
- x86/x64: _ReadWriteBarrier(); _mm_sfence();
  - Compiler intrinsic, prevents compiler reordering
- PowerPC: __lwsync();
  - Hardware barrier, prevents CPU write reordering
- **Positioning is crucial!**
  - **Write the data**
  - **__mb_release()**
  - **Write the control value**

```
void foo()
{
   data = 1;
   __mb_release();
   flag = 1;
}
```

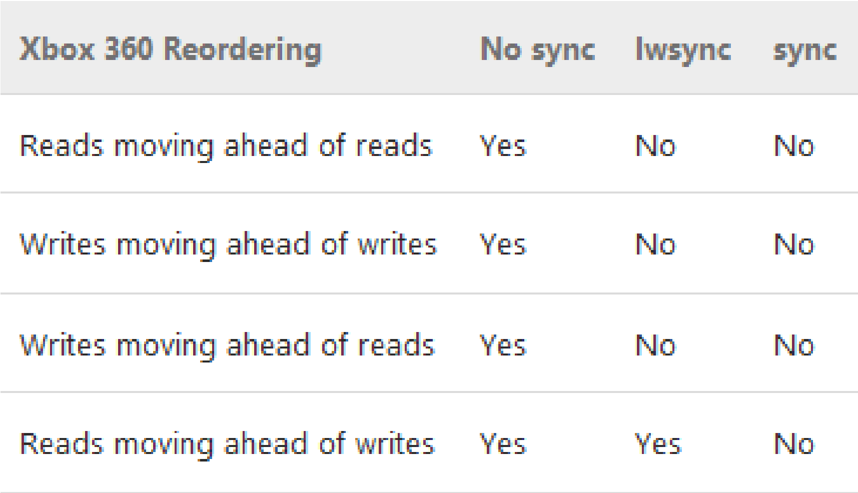
Memory Barriers – Acquire Semantics

- Prevents reordering of reads by compiler *or* CPU
  - Used when gaining access to data
- x86/x64: _ReadWriteBarrier(); _mm_lfence();
  - Compiler intrinsic, prevents compiler reordering
- PowerPC: __lwsync(); or isync();
  - Hardware barrier, prevents CPU read reordering
  - __lwsync()
    - This syncs the Store Q, Inv Q and waits for Loads
  - __isync()
    - This empties the Inv Q and clears the I-cache. This in turn prevent instruction prefetching and therefore speculative loads. This guarantees that any data read after is up to date.
- **Positioning is crucial!**
  - **Read the control value**
  - **__mb_acquire()**
  - **Read the data**

```
void bar()
{
   while (flag == 0);
   __mb_acquire();
   assert(data);
}
```

A quick note on __lwsync()

- It does NOT enforce ordering of LOADS following STORES
- Be careful on ‘acquire’ memory barriers



Xbox 360 Reordering	No sync	lwsync	sync
Reads moving ahead of reads	Yes	No	No
Writes moving ahead of writes	Yes	No	No
Writes moving ahead of reads	Yes	No	No
Reads moving ahead of writes	Yes	Yes	No

False Sharing

- Avoid having data members operated on by different cores on the same cache line
  - Not doing this results in cache line ping-ponging and degrades performance
- Keep the access control flag for shared data on its own cache line to prevent this

Three Independent Layers

- Execution
  - The CPU can execute independently from the cache as long as the cache has the data
- Cache
  - The cache **doesn’t** have to write any data to main memory in order to ensure cache coherency
  - Other optimizations exist to further prevent writes to main memory unless really needed. (Example: MOESI)
- Main Memory
  - I/O devices(read GPU) sees this and not caches. You need to really force the data all the way down to main memory if you want another device to be able to read it.
  - Main memory is REALLY FAR AWAY!!!

Compare And Swap (CAS)

- Atomic update of an aligned native sized memory location
  - 32-bit, 64-bit and sometimes other sizes
- Operates on the ‘Cache Layer’, not main memory
- PowerPC
  - Load With Reservation (addr)
  - Conditional Store (addr, newValue)
    - Returns success(0)/failure(!0)
- Intel/AMD
  - CompareAndExchange (addr, expected, newValue)
    - Returns the previous value in ‘addr’.
    - prevValue == expected ? Success : Failure

Example: Compare And Swap (CAS)

- PowerPC/PS3:
  - lwarx / stwcx
    - The PowerPC processor can hold only one reservation at a time.
  - Does NOT guarantee that all other prior loads/stores are visible by the other cores
    - Solution: Add __lwsync or __isync after the successful CAS.
- Intel/AMD
  - lock cmpxchg [ecx], edx
    - ecx – pointer to the variable
    - eax – expected value of variable
    - edx – value to write to IF variable == eax
  - DOES guarantee that all prior loads/stores are visible by the other cores

Acquire Example (Spinlock)

Apple OS code

```
spinlock_64_try_mp:
        mr r5, r3               // ‘r5’ is our lock address
        li r3, 1                // load default return value ‘success’
1:
        lwarx r4,0,r5           // load with reservation
        li r6,-1                // locked == -1
        cmpwi r4,0
        bne-- 2f                // early out if locked

        stwcx. r6,0,r5          // conditional store
        isync                   // cancel speculative execution
        beqlr++                 // if successful CAS return success (r3=1)
        b 1b                    // … else, try again
2:
        li r6,-4
        stwcx. r5,r6,r1         // clear the pending reservation (dummy write)
        li r3,0                 // we did not get the lock, return fail (r3=0)
        blr
```

Release Example (Spinlock)

Apple OS code

```
spinlock_64_unlock_mp:
        lwsync    // complete prior stores before unlock
        li r4,0
        stw r4,0(r3)
        blr
```

Multi-Core Programming Is Hard

- It is wise to use existing OS synchronization primitives (CriticalSection, Mutex, Events)
- …unless you really want performance
  - Mutex.Lock = thousands of cycles
  - Barriers = ~hundred cycles

Timings in cycles

	XBox PowerPC	Windows Intel
lwsync	33-48	20-90
InterlockedIncrements CAS (OS func)	225-260	36-90
CriticalSection (Acq+Rel)	~345	40-100
Mutex (Acq+Rel)	~2350	750-2500

Thread-Safe != Concurrent

- Thread-Safe (my own definition)
  - Many threads can safely call this function
  - Does not guarantee progress by more than one thread
- Concurrent (my own definition)
  - Many threads can safely call this function
  - Most of the threads are having forward progress at all times

Best Practices

- Prefer using OS provided CriticalSections
- Use the Acquire/Release MemoryBarriers
- Align shared data/flags to 128 byte boundaries
  - Even on Intel/AMD due to hardware prefetching
- PS3: When in doubt and you are loading data after a lock, use ‘isync’ over ‘lwsync’
- Let your peers review any code written at this level to ensure that your code is functional BEFORE checking in

Take Away

- Use specific instructions to force data to other layers
  - Cache Layer – (Other Cores) - lwsync() and isync()
  - Main Memory (IO devices) – eioio and sync()
- It’s a bit like SPU programming and DMA’s
  - Explicitly ‘transfer’ data between layers
- This is still a very simplified view of how processors work!
  - Example: Memory Access Modes
    - Write-Through
    - Cache-Inhibited
    - Cache-Coherent
    - Guarded
- Be scared!!
  - If you’re not scared you didn’t understand this presentation

References

- Dr. Dobbs Articles [Herb Sutter]
  - The Pillars of concurrency
    - *www.drdobbs.com/dept/architect/200001985*
  - Lock-Free Code: A False Sense of Security
    - *www.**drdobbs**.com/cpp/210600279*
  - Writing Lock-Free Code: A Corrected Queue
    - *www.**drdobbs**.com/parallel/210604448*
  - A Generalized Concurrent Queue
    - *www.drdobbs.com/parallel/211601363*
- Memory Barriers: A Hardware view for software hackers
  - *http://irl.cs.ucla.edu/~yingdi/paperreading/whymb.2010.06.07c.pdf*
- Wikipedia: MESI Protocol
  - *http://en.wikipedia.org/wiki/MESI_protocol*
- Lock-Less Programming [Bruce Dawson]
  - *http://www.gdcvault.com/play/1751/Lockless-Programming-in*
- MSDN References
  - Lockless Programming Considerations
    - *http://msdn.microsoft.com/en-us/library/windows/desktop/ee418650%28v=vs.85%29.aspx*
  - Memory Barrier and _ReadWriteBarrier
    - *http://msdn.microsoft.com/en-us/library/windows/desktop/ms684208%28v=vs.85%29.aspx*
    - *http://msdn.microsoft.com/en-us/library/f20w0x5e%28v=vs.80%29.aspx*
  - *InterlockedCompareExchange (CAS – Compare and Swap)*
    - *http://msdn.microsoft.com/en-US/library/ttk2z1ws%28v=vs.80%29.aspx*
- PowerPC Documentation
  - *http://www.ibm.com/developerworks/systems/articles/powerpc.html*
  - *http://www.opensource.apple.com/source/xnu/xnu-792.2.4/osfmk/ppc/commpage/atomic.s*
  - *http://www.opensource.apple.com/source/xnu/xnu-1504.9.37/osfmk/ppc/commpage/spinlocks.s*
- Intel – Chapters 8.1, 11.2 and 11.4
  - *ftp://download.intel.com/design/processor/manuals/253668.pdf*