From weak to weedy

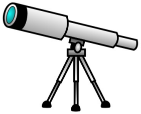
Effective use of memory barriers in the ARM Linux Kernel

Will Deacon
will.deacon@arm.com

Embedded Linux Conference Europe
Edinburgh, UK

October 24, 2013

Scope



Memory ordering is a complex topic!

- Different rules across different versions/implementations of different architectures
- Not well understood by most software engineers
- Great potential for subtle, non-repeatable software bugs
- Key contributor to overall system performance

We will focus on the ARMv7 Linux kernel from a SW perspective (the ARM ARM remains authoritative!).

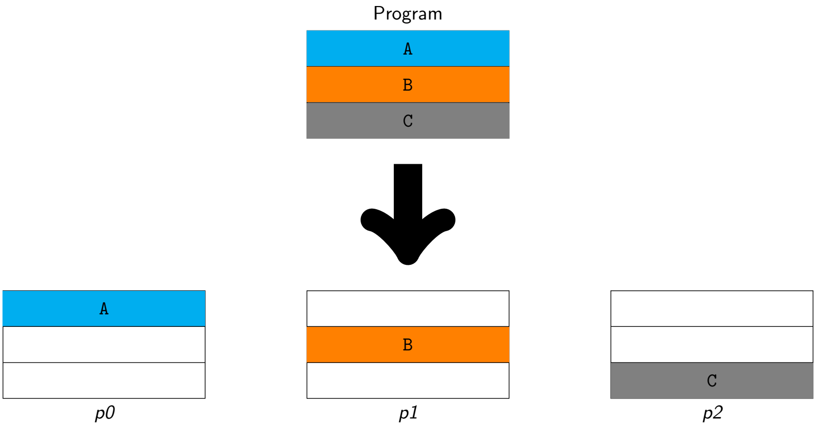
Sequential Consistency

A talk about memory ordering wouldn't be complete without a brief description of *sequential consistency*.

Sequential Consistency (SC):

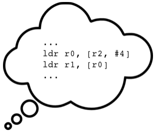
'A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.' – Leslie Lamport (1979)

Sequential Consistency (2)



Sequential Consistency (3)

SC makes SMP systems nice and easy to reason about...



```
...
ldr r0, [r2, #4]
ldr r1, [r0]
...
```

...but the hardware guys hate it!

- Out-of-order and speculative execution
- Caches (and coherency in SMP)
- Write atomicity
- Store buffers (read bypass and write merging)
- Multi-ported bus topologies
- Memory-mapped I/O

Back to square one with memory latency!

Memory Ordering

To facilitate these hardware optimisations, ordering of memory operations is often relaxed from *program order*, potentially leading to SC violations.

Initially: $A = B = 0$

p0	*p1*	*Results*	*SC*
`a: A = 2;`	`c: C = B;`	`(C, D) == (0, 0)`	?
`b: B = 1;`	`d: D = A;`	`(C, D) == (0, 2)`	?
		`(C, D) == (1, 2)`	?
		`(C, D) == (1, 0)`	?

This is defined by the *memory (consistency) model* for the architecture.

Memory Ordering

To facilitate these hardware optimisations, ordering of memory operations is often relaxed from *program order*, potentially leading to SC violations.

Initially: $A = B = 0$

p0	*p1*	*Results*	*SC*
`a: A = 2;`	`c: C = B;`	`(C, D) == (0, 0)`	`Y (c, d, a, b)`
`b: B = 1;`	`d: D = A;`	`(C, D) == (0, 2)`	`Y (c, a, d, b)`
		`(C, D) == (1, 2)`	`Y (a, b, c, d)`
		`(C, D) == (1, 0)`	`N (d, a, b, c)`

This is defined by the *memory (consistency) model* for the architecture.

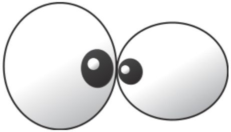
Safety Nets

Weakly ordered memory models offer *safety nets* to the programmer for explicit control over access ordering. These are commonly referred to as *barriers* or *fences*.

The ARMv7 memory model includes:

- A range of barrier instructions
- Defined dependencies between accesses
- Memory types with different ordering constraints

Observers



An *observer* is an agent in the system that can access memory:

- Not necessarily a CPU (which contains multiple observers!)
- Master within a given *shareability domain* (more later)
- Slave interfaces cannot observe any accesses

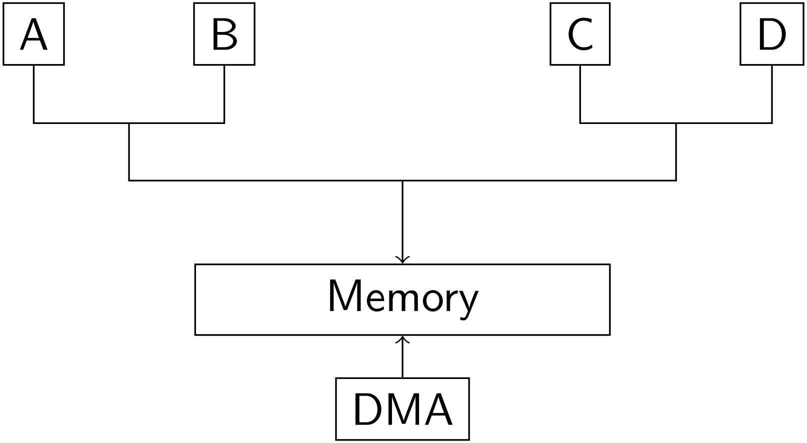
Shareability Domains

Shareability domains define sets of observers within a system.

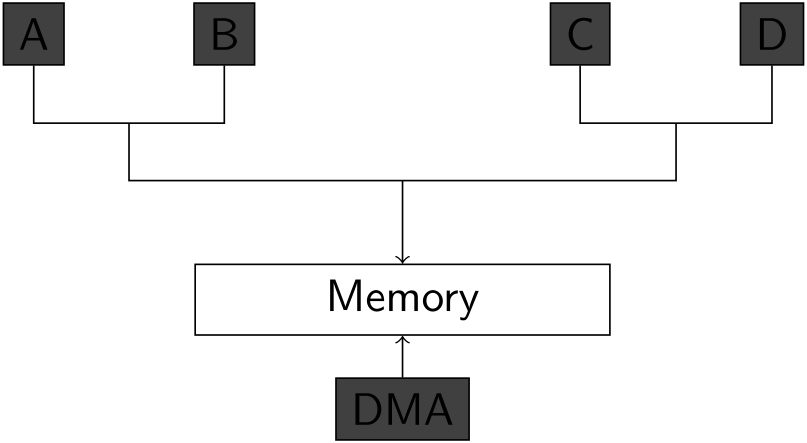
- {Non, Inner, Outer}-shareable and Full System
- Impact on cache coherency and shared memory
- Multiple domain instances (no strictly nested)
- System-specific, but architectural (and Linux) expectations

'This architecture (ARMv7) is written with an expectation that all processors using the same operating system or hypervisor are in the same Inner Shareable shareability domain.'

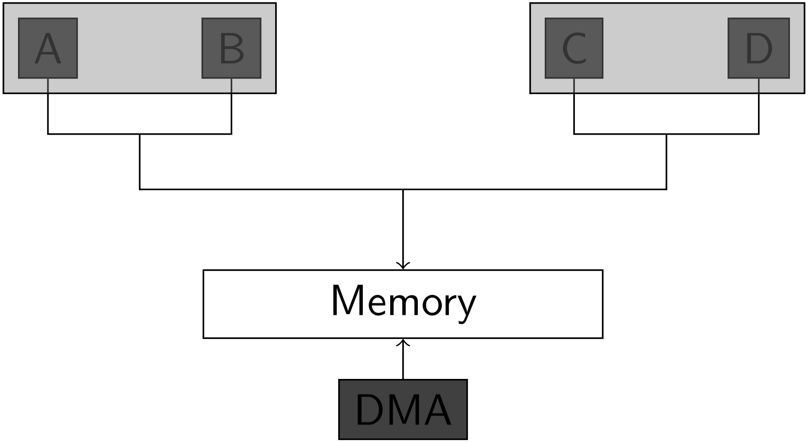
Example Domains



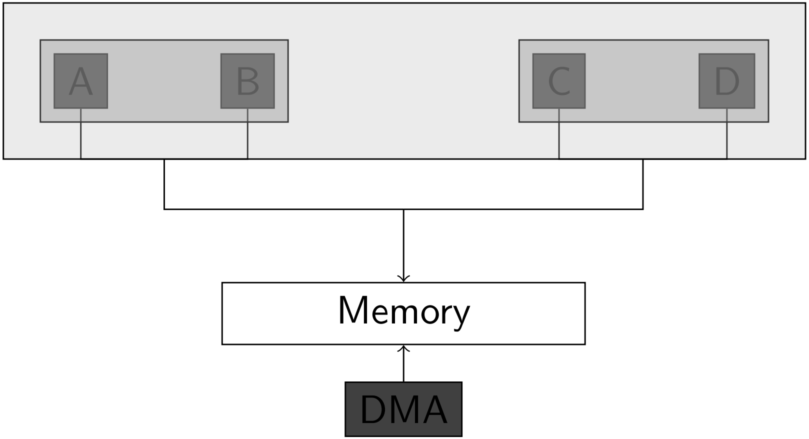
Example Domains (NSH)



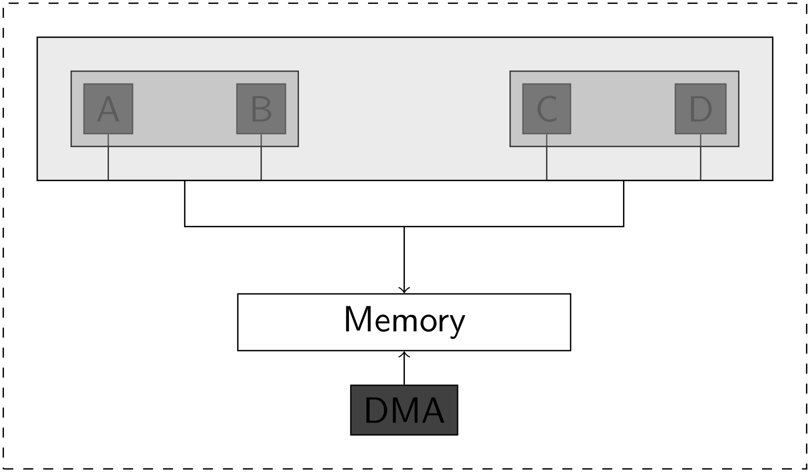
Example Domains (ISH)



Example Domains (OSH)



Example Domains (SY)



Observability

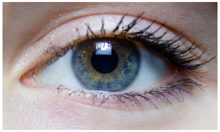
Ordering is defined in terms of *observability* by memory masters.

Writes

'A write to a location in memory is said to be observed by an observer when: (1) A subsequent read of the location by the same observer will return the value written by the observed write, or written by a write to that location by any observer that is sequenced in the coherence order of the location after the observed write and (2) A subsequent write of the location by the same observer will be sequenced in the coherence order of the location after the observed write'

This is actually pretty intuitive. . .

Observability (2)



. . . but reads are observable too!

Reads

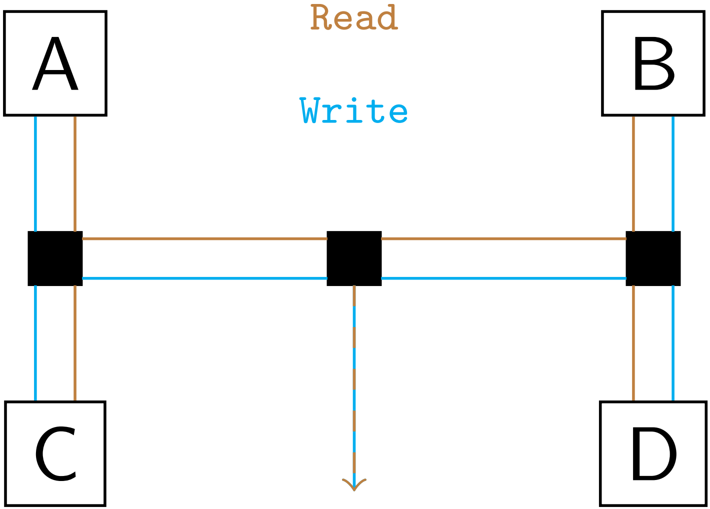
'A read of a location in memory is said to be observed by an observer when a subsequent write to the location by the same observer will have no effect on the value returned by the read.'

Global Observability and Completion

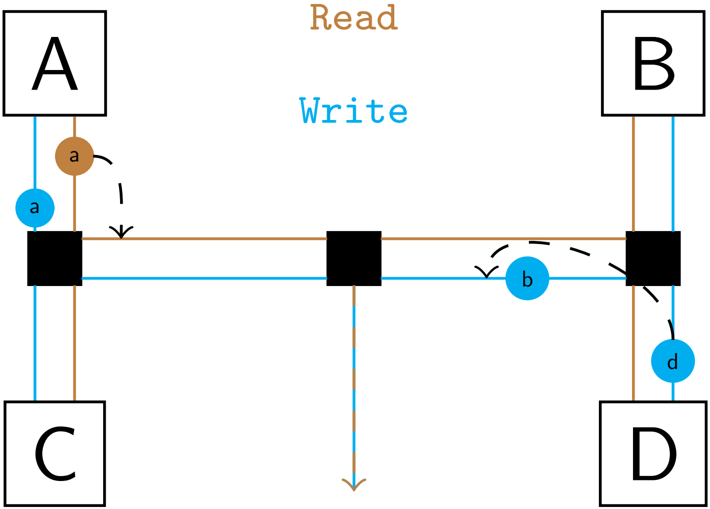
- A normal memory access is *globally observed* for a shareability domain when it is observed by all observers in that domain.
- A table walk is *complete* for a shareability domain when its accesses are globally observed in that domain and the TLB is updated.
- An access is complete for a shareability domain when it is globally observed in that domain and any table walks associated with it have completed in the same domain.

Maintenance operations also have the notion of completion.

Ordering Diagrams



Ordering Diagrams



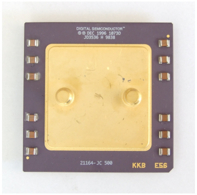
Dependencies

In the absence of explicit barriers, *dependencies* define observation order of normal memory accesses.

Address:	value returned by a read is used to compute the address of a subsequent access.
Control:	value returned by a read is used to determine the condition flags and the flags are used in the condition code checking that determines the address of a subsequent access.
Data:	value returned by a read is used as data written by a subsequent write.

There are also a few other rules (RaR, store speculation).

Dependency Examples



```
ldr   r1, [r0, #4]
and   r1, #0xfff
ldr   r3, [r2, r1]
```

(address)

```
ldr     r1, [r0, #4]
cmp     r1, #1
addeq   r2, #4
ldr     r3, [r2]
```

(control)

```
ldr   r1, [r0, #4]
add   r1, #5
str   r1, [r2]
```

(data)

Question: Which dependencies enforce ordering of observability?

Memory Barriers

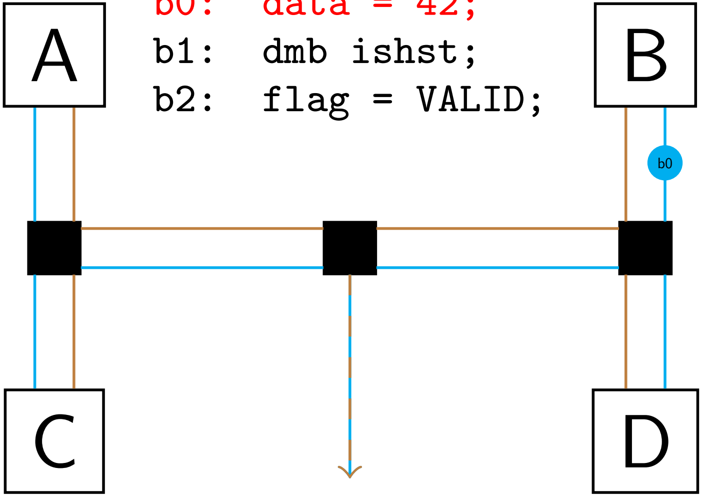
The ARMv7 architecture defines three barrier instructions:

`isb`	Pipeline flush and context synchronisation
`dmb <option>`	Ensure ordering of memory accesses
`dsb <option>`	Ensure completion of memory accesses

The `<option>` argument specifies the required shareability domain (`NSH, ISH, OSH, SY`) and access type (`ST`). Defaults to 'full system', all access types if omitted.

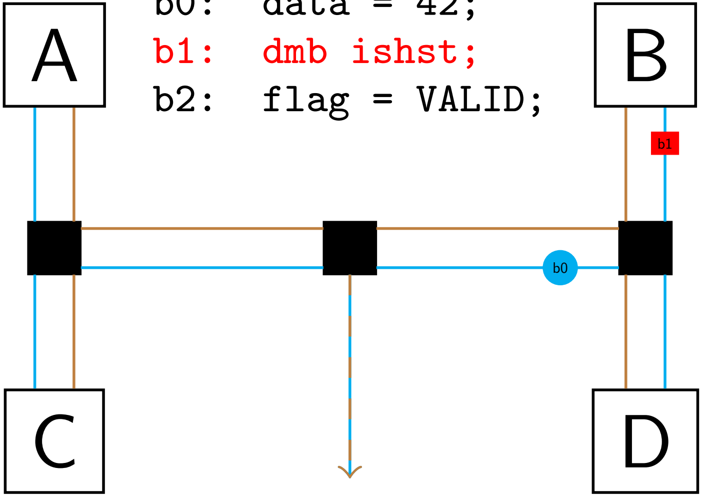
Ordering Diagrams (DMB)

```
b0:   data = 42;
b1:   dmb ishst;
b2:   flag = VALID;
```



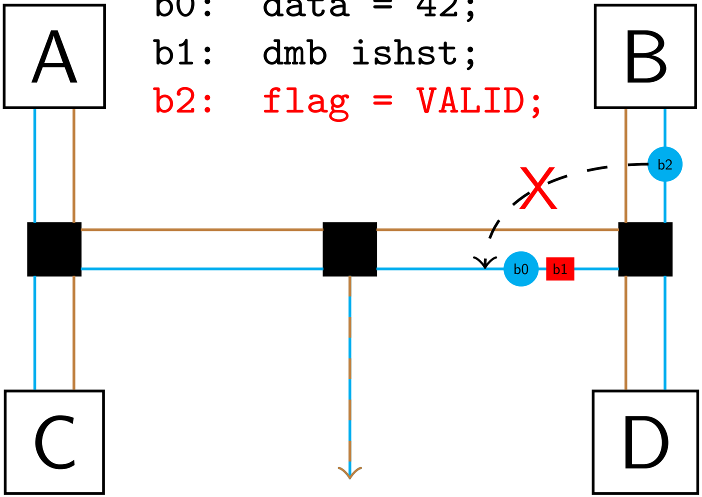
Ordering Diagrams (DMB)

```
b0:   data = 42;
b1:   dmb ishst;
b2:   flag = VALID;
```



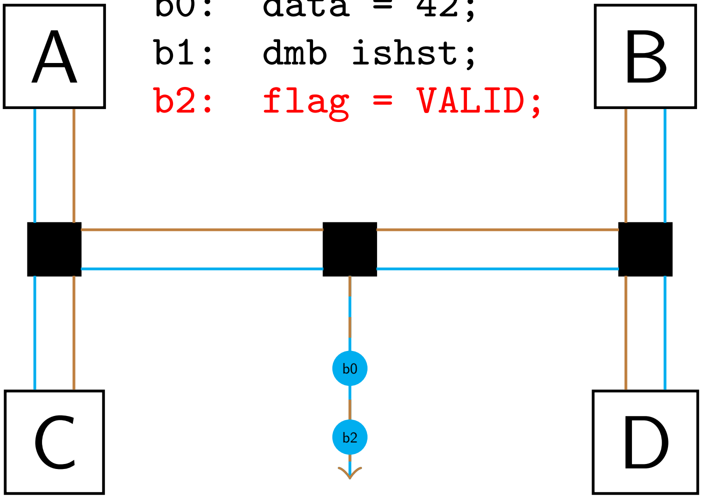
Ordering Diagrams (DMB)

```
b0:   data = 42;
b1:   dmb ishst;
b2:   flag = VALID;
```



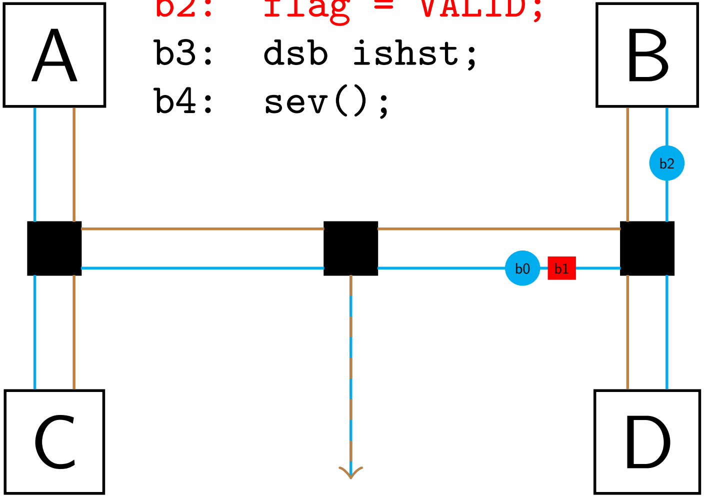
Ordering Diagrams (DMB)

```
b0:   data = 42;
b1:   dmb ishst;
b2:   flag = VALID;
```



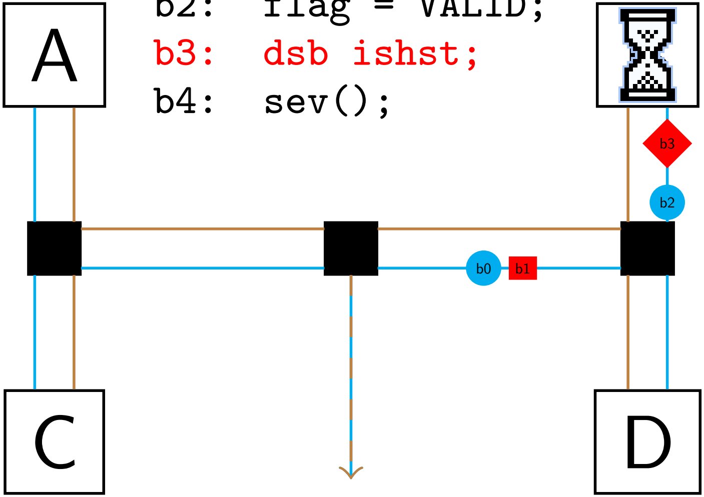
Ordering Diagrams (DSB)

```
b2:   flag = VALID;
b3:   dsb ishst;
b4:   sev();
```



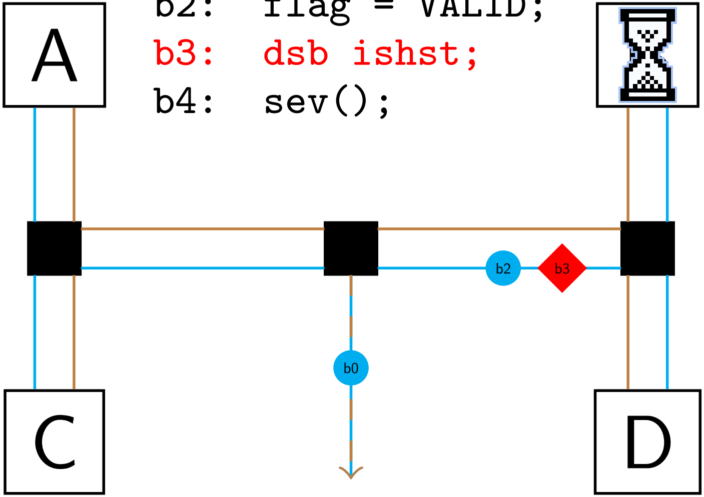
Ordering Diagrams (DSB)

```
b2:   flag = VALID;
b3:   dsb ishst;
b4:   sev();
```



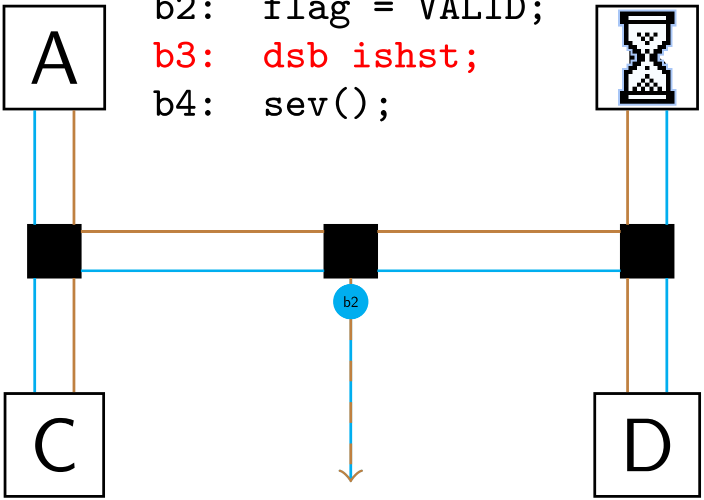
Ordering Diagrams (DSB)

```
b2:   flag = VALID;
b3:   dsb ishst;
b4:   sev();
```



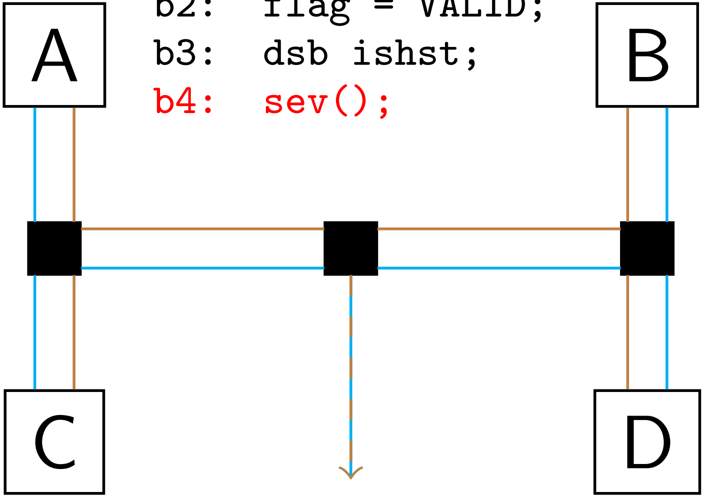
Ordering Diagrams (DSB)

```
b2:   flag = VALID;
b3:   dsb ishst;
b4:   sev();
```



Ordering Diagrams (DSB)

```
b2:   flag = VALID;
b3:   dsb ishst;
b4:   sev();
```



Overloading of barrier instructions

The barrier instructions are also overloaded to affect other parts of the system:

Cache	maintenance ordered by `dmb [st]` and completed using `dsb [st]` *on the same CPU*
Branch predictor	maintenance is completed at a context synchronisation operation (e.g. `isb`)
TLB	maintenance completed using `dsb`
PTE	updates 'published' to walker with `dsb [st]` (MP extensions)

`isb` required for explicit synchronisation with instruction stream.

Barriers in Linux

Linux defines more barrier types than you can shake a stick at!

Compiler:	`barrier()`
Mandatory:	`mb()`, `wmb()`, `rmb()`, (`read_barrier_depends()`)
SMP conditional:	`smp_*` – domain?
MMIO write:	(`mmiowb()`)

Also implicit barriers in locks, atomics, bitops, I/O accessors. . . (see `Documentation/memory-barriers.txt`).

Low-level barriers

The ARM architecture port maps the Linux barriers onto the v7 instruction set:

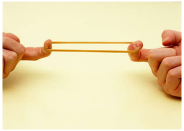
- `smp_*` ⇒ `dmb [sy];` (SMP)
- `rmb` ⇒ `dsb [sy];`
- `[w]mb` ⇒ `dsb [sy]; [outer_sync();]` (DMA)

There are also low-level barrier macros for ARM-specific code:

- `dmb` ⇒ `dmb [sy];`
- `dsb` ⇒ `dsb [sy];`

Spot the problem? (we've been getting away with it so far. . .)

Extended API



From Linux `3.12`, we can specify the domain and access type for low-level barriers. This gives us a measurable performance boost, but increases the scope for horrible bugs!

```
/* Write local pte */
    dsb(nshst);
/* TLB invalidation */
    dsb(nsh);
```

All implemented write barriers take the `-st` option and the `smp_*` barriers become inner-shareable. Be sure to grab a 'recent' binutils.

Example: spin_unlock

```
/*
 * Ensure accesses don't leak out
 * from critical section
 */
smp_mb();
/* Release the lock */
lock->tickets.owner++;
/* Wake up waiting CPUs */
dsb_sev();
```

```
@ 3.11
dmb     sy
ldrh    r3, [r0]
add     r3, r3, #1
strh    r3, [r0]
dsb     sy
sev
```

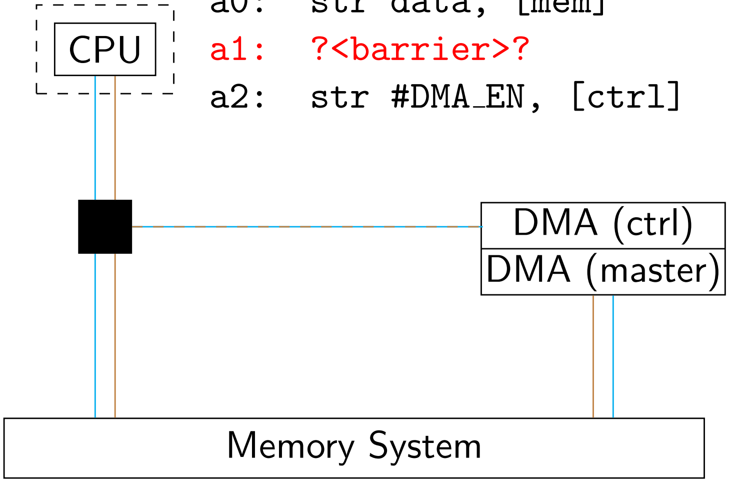
Example: spin_unlock

```
/*
 * Ensure accesses don't leak out
 * from critical section
 */
smp_mb();
/* Release the lock */
lock->tickets.owner++;
/* Wake up waiting CPUs */
dsb_sev();
```

```
@ 3.12
dmb     ish
ldrh    r3, [r0]
add     r3, r3, #1
strh    r3, [r0]
dsb     ishst
sev
@ ~5% hackbench
@ improvement on TC2!
```

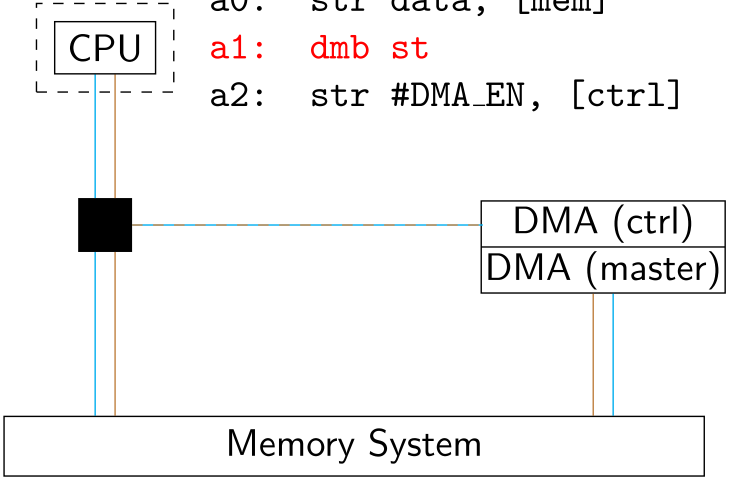
Example: DMA To Device

```
a0:  str data, [mem]
a1:  ?<barrier>?
a2:  str #DMA_EN, [ctrl]
```



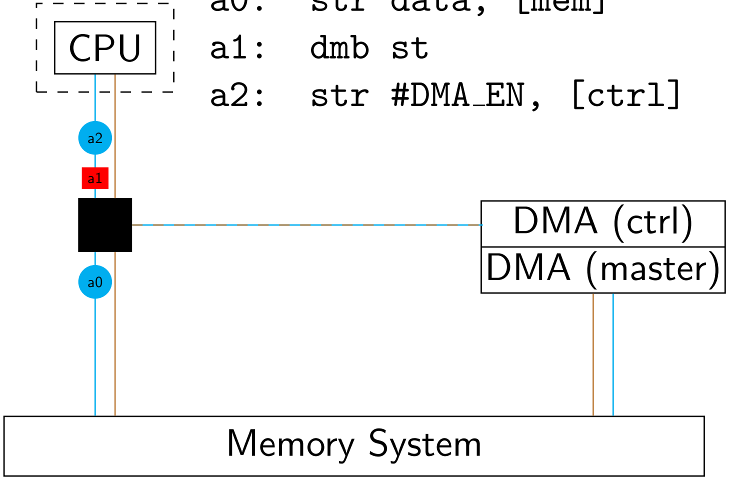
Example: DMA To Device

```
a0:   str data, [mem]
a1:   dmb st
a2:   str #DMA_EN, [ctrl]
```



Example: DMA To Device

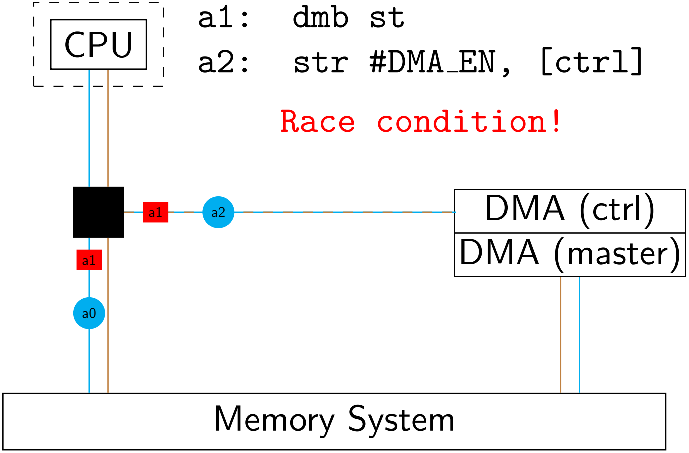
```
a0:   str data,  [mem]
a1:   dmb st
a2:   str #DMA_EN,  [ctrl]
```



Ordering of Observability Satisfied!

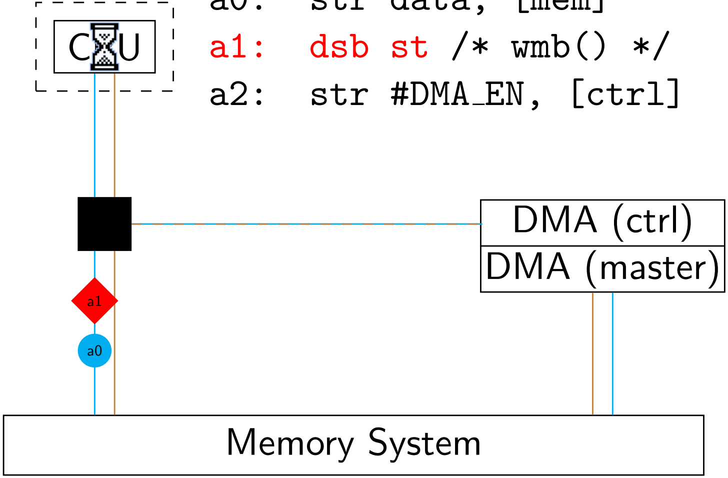
```
a0:  str data, [mem]
a1:  dmb st
a2:  str #DMA_EN, [ctrl]
```

Race condition!



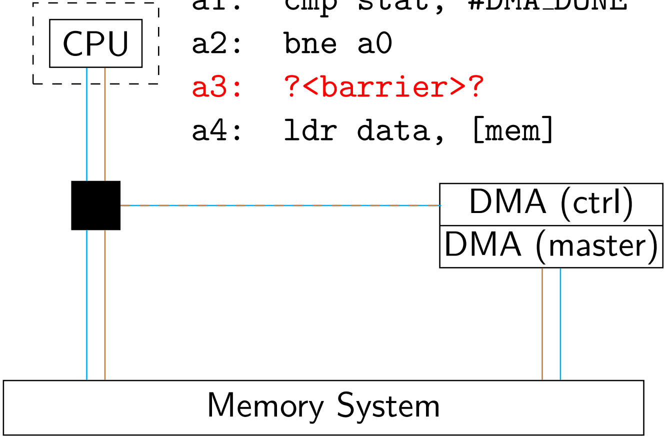
Example: DMA To Device

```
a0:  str data, [mem]
a1:  dsb st /* wmb() */
a2:  str #DMA_EN, [ctrl]
```



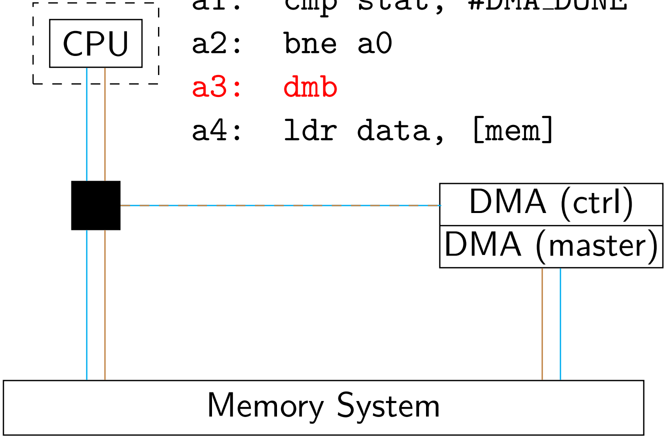
Example: DMA From Device

```
a0:  ldr stat, [ctrl]
a1:  cmp stat, #DMA_DONE
a2:  bne a0
a3:  ?<barrier>?
a4:  ldr data, [mem]
```



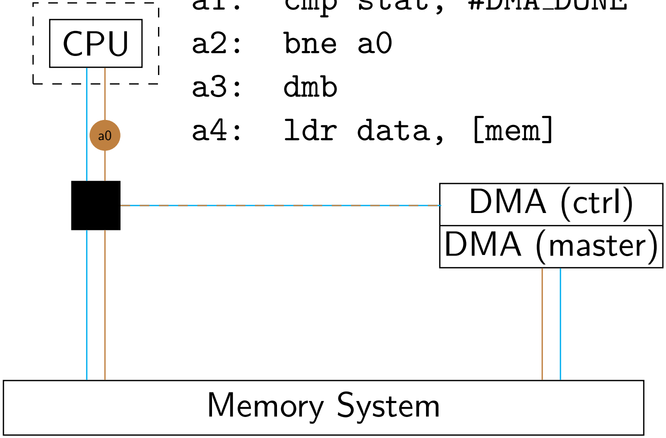
Example: DMA From Device

```
a0:  ldr stat, [ctrl]
a1:  cmp stat, #DMA_DONE
a2:  bne a0
a3:  dmb
a4:  ldr data, [mem]
```



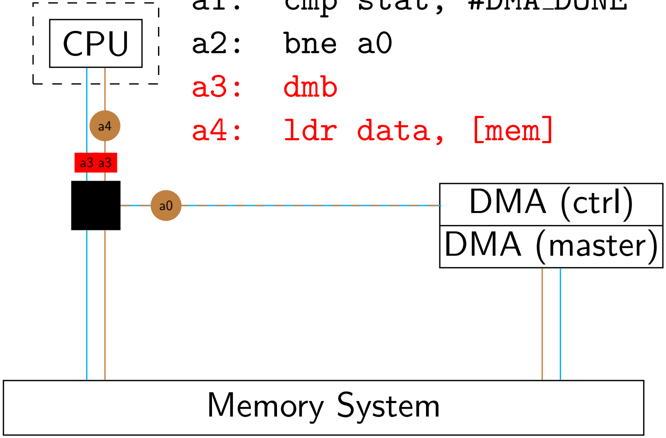
Example: DMA From Device

```
a0:  ldr stat, [ctrl]
a1:  cmp stat, #DMA_DONE
a2:  bne a0
a3:  dmb
a4:  ldr data, [mem]
```



Speculation Through Control Dependency!

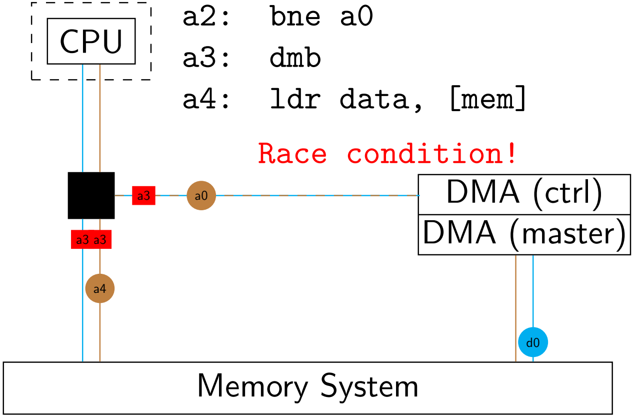
```
a0:  ldr stat, [ctrl]
a1:  cmp stat, #DMA_DONE
a2:  bne a0
a3:  dmb
a4:  ldr data, [mem]
```



Speculation Through Control Dependency!

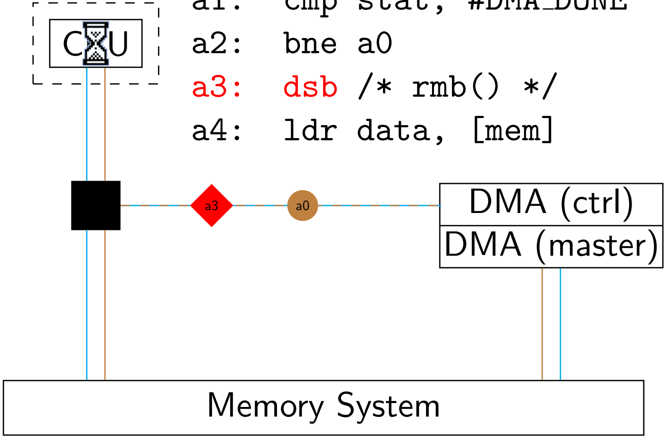
```
a0:  ldr stat, [ctrl]
a1:  cmp stat, #DMA_DONE
a2:  bne a0
a3:  dmb
a4:  ldr data, [mem]
```

Race condition!



Example: DMA From Device

```
a0:  ldr stat, [ctrl]
a1:  cmp stat, #DMA_DONE
a2:  bne a0
a3:  dsb /* rmb() */
a4:  ldr data, [mem]
```



Which Barrier Should I Use?

Ignoring maintenance operations, memory barriers are typically required when publishing to and consuming from other observers (*data* vs *control*).

1. Do you even need a barrier? (dependencies)
2. Do you only care about ordering between CPUs? (`smp_*`)
3. Only care about reads or writes? (`*[rw]mb`)
4. Low-level barriers rarely needed (`nsh`, `osh` and maintenance)
5. I/O accessors and relaxed variants (`readl`, `writel`)

Questions?

ARMv8

ARMv8 introduces some exciting new features to the memory model!

`-ld`	barrier option to order reads against reads/writes
Half barriers	in the form of *acquire/release* operations
Device memory attributes	`nGnRnE`

There's also the problem of defining `*_relaxed` across architectures. . .