



On ~~Formal~~ Specifications at Arm

Towards computer-aided Architecture development and consumption

Jade Alglave
2026-01-29



We're hiring!
3 engineer roles
3 PhD studentships

arm

Arm Architecture

Arm Architecture

- Arm publishes Architecture specifications
- Specifications allow their readers to find answers to their questions about the behaviour of Arm systems (for example telephones, tablets, laptops...)

An example of question about the behaviour of Arm systems

AArch64 MP

```
{  
0:X1=x; 0:X3=y;  
1:X0=y; 1:X2=x;  
}
```

P0		P1	;
MOV W0,#1		LDR W1,[X0]	;
STR W0,[X1]		LDR W3,[X2]	;
MOV W2,#1			;
STR W2,[X3]			;
exists (1:X1=1 $\wedge$ 1:X3=0)			

What happens if two threads P0 and P1 communicate via shared memory locations x and y, in the following way:

P0 does a store of value 1 to x, and sets the flag y with another store.

P1 does a load of the flag y and a load of x.

If P1 sees the flag set, can it still see the old value of x?

Another example of question

AArch64 MP+dmb.st+dmb.ld

```
{  
0:X1=x; 0:X3=y;  
1:X0=y; 1:X2=x;  
}
```

P0		P1	;
MOV W0,#1		LDR W1,[X0]	;
STR W0,[X1]		DMB LD	
DMB ST		LDR W3,[X2]	;
MOV W2,#1			;
STR W2,[X3]			;
exists (1:X1=1 /\ 1:X3=0)			

What happens if two threads P0 and P1 communicate via shared memory locations x and y, in the following way:

P0 does a store of value 1 to x, and after using a DMB ST barrier, sets the flag y with another store.

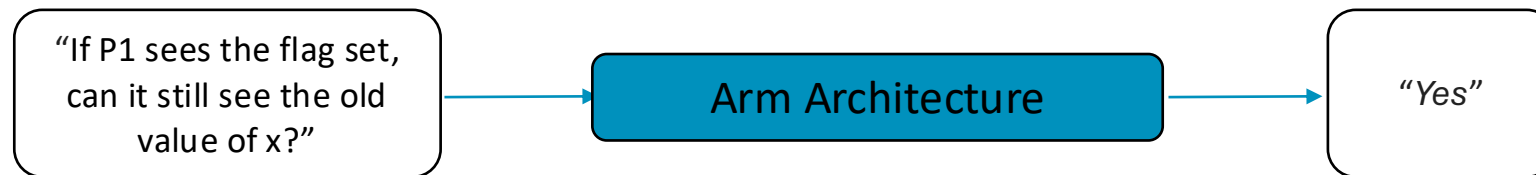
P1 does a load of the flag y and after using a DMB LD barrier, a load of x.

If P1 sees the flag set, can it still see the old value of x?

Demo on an Arm system

Running MP and MP+dmb.st+dmb.ld on hardware with the litmus7 tool

Specification



Arm Architecture

- Arm publishes Architecture specifications
- Architecture specifications (the Architecture, for short) prescribe, for a given program written in Arm assembly, which behaviours are Allowed or Forbidden. This should be true on **all** Arm systems.
- In other words, the Architecture describes the envelope of all legitimate hardware implementations.

An example of Architecturally Allowed behaviour

AArch64 MP

```
{  
0:X1=x; 0:X3=y;  
1:X0=y; 1:X2=x;  
}
```

P0		P1	;
MOV W0,#1		LDR W1,[X0]	;
STR W0,[X1]		LDR W3,[X2]	;
MOV W2,#1			;
STR W2,[X3]			;
exists (1:X1=1 $\wedge$ 1:X3=0)			

What happens if two threads P0 and P1 communicate via shared memory locations x and y, in the following way:

P0 does a store of value 1 to x, and sets the flag y with another store.

P1 does a load of the flag y and a load of x.

If P1 sees the flag set, can it still see the old value of x?

An example of Architecturally Forbidden behaviour

AArch64 MP+dmb.st+dmb.ld

```
{  
0:X1=x; 0:X3=y;  
1:X0=y; 1:X2=x;  
}
```

P0		P1	
MOV W0,#1		LDR W1,[X0]	;
STR W0,[X1]		DMB LD	
DMB ST		LDR W3,[X2]	;
MOV W2,#1			;
STR W2,[X3]			;
exists (1:X1=1 /\ 1:X3=0)			

What happens if two threads P0 and P1 communicate via shared memory locations x and y, in the following way:

P0 does a store of value 1 to x, and after using a DMB ST barrier, sets the flag y with another store.

P1 does a load of the flag y and after using a DMB LD barrier, a load of x.

If P1 sees the flag set, can it still see the old value of x?

Demo

Running MP and MP+dmb.st+dmb.ld with the herd7 tool

Arm Architecture

- The Architecture describes the envelope of all legitimate hardware implementations
- In other words, the Architecture is an interface, or a contract, between hardware and software
- This entails that:
 - if a hardware vendor builds their processor in a way that is compliant with the Arm ARM,
 - and if a software developer builds their stack in a way that is compliant with the Arm ARM,
 - then that software should run on that hardware,
 - and that software should run on other Arm processors too
- To help confirm compliance, Arm distributes a set of tests (called the ACK) to architecture licensees

arm

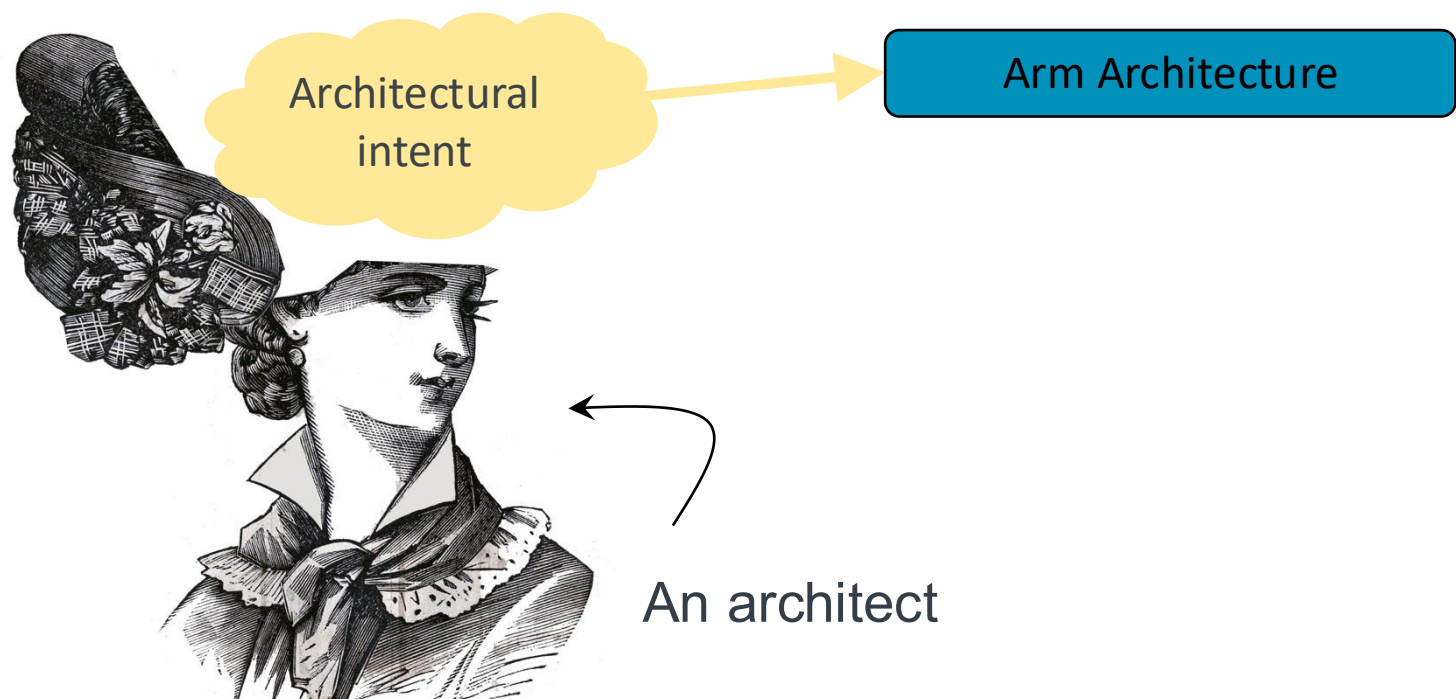
What is Architecture?

From a consumer point of view

Arm Architecture

- Arm publishes Architecture specifications
- In particular, the Arm Architecture Reference Manual (Arm ARM)

Architecture is a record of the architect's intent



The AArch64 Virtual Memory System Architecture
D8.1 Address translation

D8.1 Address translation

R _{DWFWP}	If an implementation is executing in AArch64 state, then that implementation uses either or both of the VMSAv8-64 and the VMSAv9-128 translation systems.
R _{RKHUV}	All of the following determine whether a translation stage uses the VMSAv9-128 translation system: • For stage 1 translations in the EL1&0 translation regime, the Effective value of TCR2_EL1.D128 is 1. • For stage 1 translations in the EL2&0 translation regime, the Effective value of TCR2_EL2.D128 is 1. • For stage 1 translations in the EL3 translation regime, the Effective value of TCR_EL3.D128 is 1. • For stage 2 translations, the Effective value of VTCR_EL2.D128 is 1. Otherwise, the translation stage uses the VMSAv8-64 translation system. This applies even when the translation stage is disabled.
I _{CCTQS}	Address translation converts the addresses used by instructions to the addresses used by the physical memory system.
R _{CRPHV}	When a data address or instruction address is used in an instruction, it is a <i>virtual address</i> (VA). This includes any address stored in one of the following registers: • <i>Program counter</i> (PC). • <i>Stack pointers</i> (SP). • <i>Link register</i> (LR). • <i>Exception link register</i> (ELR).
R _{CRDGS}	When an access is made to the physical memory system, a <i>physical address</i> (PA) is used.
R _{KCNIX}	An address translation maps a VA to a PA.
R _{CCCQQ}	An address translation requires one of the following: • A single translation stage, stage 1. • Two sequential translation stages, stage 1 and stage 2.
R _{ZKJWW}	An address translation stage maps an <i>input address</i> (IA) to an <i>output address</i> (OA).

Contractually, the Architecture is what is given in the Arm ARM

<https://developer.arm.com/documentation/ddi0487/latest/>

The AArch64 Virtual Memory System Architecture
D8.1 Address translation

D8.1 Address translation

R _{DWFWP}	If an implementation is executing in AArch64 state, then that implementation uses either or both of the VMSAv8-64 and the VMSAv9-128 translation systems.
R _{RKHHV}	All of the following determine whether a translation stage uses the VMSAv9-128 translation system: • For stage 1 translations in the EL1&0 translation regime, the Effective value of TCR2_EL1.D128 is 1. • For stage 1 translations in the EL2&0 translation regime, the Effective value of TCR2_EL2.D128 is 1. • For stage 1 translations in the EL3 translation regime, the Effective value of TCR_EL3.D128 is 1. • For stage 2 translations, the Effective value of VTCR_EL2.D128 is 1. Otherwise, the translation stage uses the VMSAv8-64 translation system. This applies even when the translation stage is disabled.
I _{CTQS}	Address translation converts the addresses used by instructions to the addresses used by the physical memory system.
R _{CRPHV}	When a data address or instruction address is used in an instruction, it is a <i>virtual address</i> (VA). This includes any address stored in one of the following registers: • <i>Program counter</i> (PC). • <i>Stack pointers</i> (SP). • <i>Link register</i> (LR). • <i>Exception link register</i> (ELR).
R _{CRDGS}	When an access is made to the physical memory system, a <i>physical address</i> (PA) is used.
R _{KCNRX}	An address translation maps a VA to a PA.
R _{CCCQQ}	An address translation requires one of the following: • A single translation stage, stage 1. • Two sequential translation stages, stage 1 and stage 2.
R _{ZKJWW}	An address translation stage maps an <i>input address</i> (IA) to an <i>output address</i> (OA).

The Arm ARM

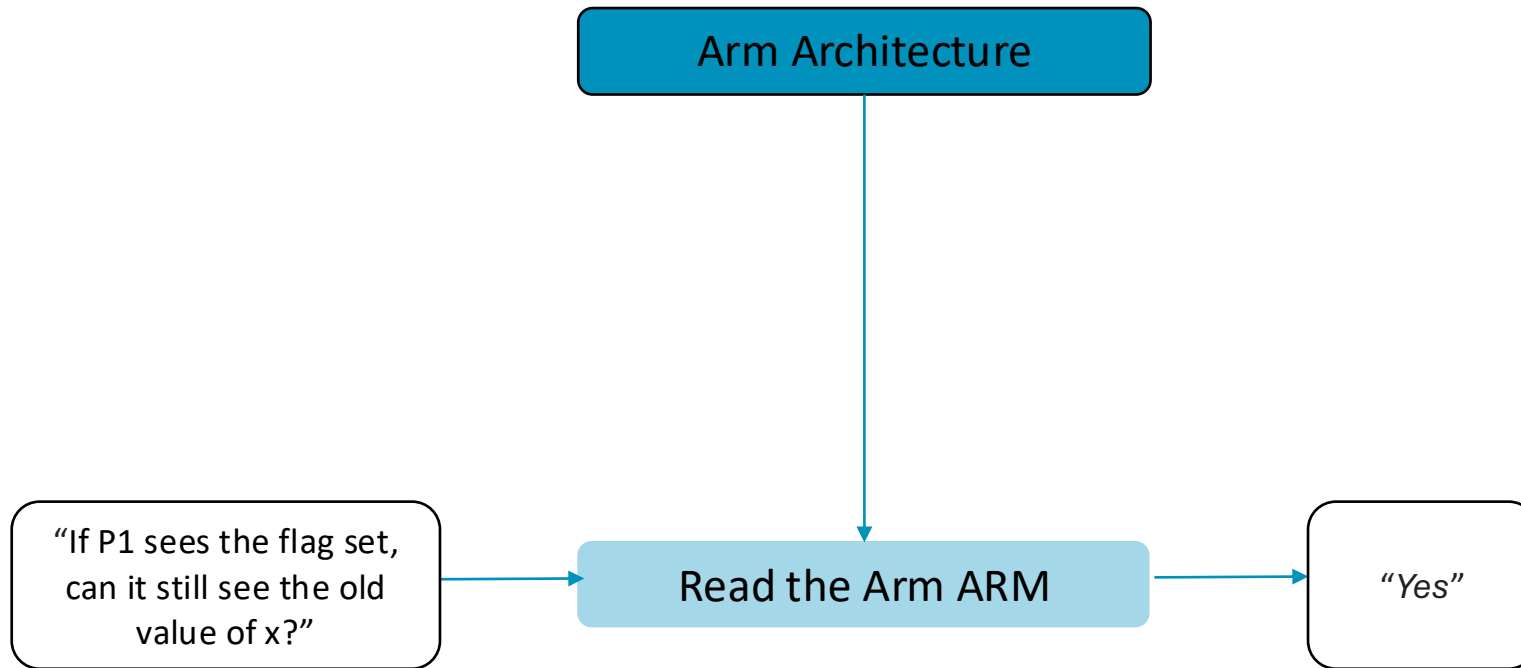
- This document is rather long: 16 825 pages
- For the most part, written in rather technical, but informal, English prose
- Corrections may be sent to architectural licensees, as diffs to that prose

arm

Answering questions about the
behaviour of Arm systems

Reading the Arm ARM

Read the Arm ARM



How to determine whether this behaviour is Architecturally Forbidden

AArch64 MP+dmb.st+dmb.ld

```
{  
0:X1=x; 0:X3=y;  
1:X0=y; 1:X2=x;  
}
```

P0		P1	;
MOV W0,#1		LDR W1,[X0]	;
STR W0,[X1]		DMB LD	
DMB ST		LDR W3,[X2]	;
MOV W2,#1			;
STR W2,[X3]			;
exists (1:X1=1 /\ 1:X3=0)			

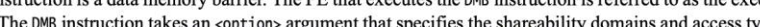
What happens if two threads P0 and P1 communicate via shared memory locations x and y, in the following way:

P0 does a store of value 1 to x, and after using a DMB ST barrier, sets the flag y with another store.

P1 does a load of the flag y and after using a DMB LD barrier, a load of x.

If P1 sees the flag set, can it still see the old value of x?

Arm ARM A.k (30 September 2016)

DDI0487A_k_armv8_arm.pdf	
Page 88 of 5,732	
Data Memory Barrier (DMB) The DMB instruction is a data memory barrier. The PE that executes the DMB instruction is referred to as the executing PE, PEE. The DMB instruction takes an <option> argument that specifies the shareability domains and access types to which the instruction applies, see Shareability and access limitations on the data barrier operations on page B2-90. If the required shareability is <i>Full system</i> then the operation applies to all observers within the system. A DMB creates two groups of memory accesses, Group A and Group B: Group A Contains: • All explicit memory accesses of the required access types from observers in the same required shareability domain as PEE that are observed by PEE before the DMB instruction. These accesses include any accesses of the required access types performed by PEE. • All loads of required access types from an observer PEX in the same required shareability domain as PEE that have been observed by any given different observer, PEY, in the same required shareability domain as PEE before PEY has performed a memory access that is a member of Group A. Group B Contains: • All explicit memory accesses of the required access types by PEE that occur in program order after the DMB instruction. • All explicit memory accesses of the required access types by any given observer PEX in the same required shareability domain as PEE that can only occur after a load by PEX has returned the result of a store that is a member of Group B. Any observer with the same required shareability domain as PEE observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined by the shareability and cacheability of the memory addresses accessed by the group members. If members of Group A and members of Group B access the same memory-mapped peripheral of arbitrary system-defined size, then members of Group A that are accessing Device or Normal Non-cacheable memory arrive at that peripheral before members of Group B that are accessing Device or Normal Non-cacheable memory. Where the members of Group A and Group B that must be ordered are from the same PE, a DMB NSH is sufficient for this guarantee.	Data Memory Barrier (DMB) The DMB instruction is a data memory barrier. The PE that executes the DMB instruction is referred to as the executing PE, PEE. The DMB instruction takes an <option> argument that specifies the shareability domains and access types to which the instruction applies, see Shareability and access limitations on the data barrier operations on page B2-90. If the required shareability is <i>Full system</i> then the operation applies to all observers within the system. A DMB creates two groups of memory accesses, Group A and Group B: Group A Contains: • All explicit memory accesses of the required access types from observers in the same required shareability domain as PEE that are observed by PEE before the DMB instruction. These accesses include any accesses of the required access types performed by PEE. • All loads of required access types from an observer PEX in the same required shareability domain as PEE that have been observed by any given different observer, PEY, in the same required shareability domain as PEE before PEY has performed a memory access that is a member of Group A. Group B Contains: • All explicit memory accesses of the required access types by PEE that occur in program order after the DMB instruction. • All explicit memory accesses of the required access types by any given observer PEX in the same required shareability domain as PEE that can only occur after a load by PEX has returned the result of a store that is a member of Group B. Any observer with the same required shareability domain as PEE observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined by the shareability and cacheability of the memory addresses accessed by the group members. If members of Group A and members of Group B access the same memory-mapped peripheral of arbitrary system-defined size, then members of Group A that are accessing Device or Normal Non-cacheable memory arrive at that peripheral before members of Group B that are accessing Device or Normal Non-cacheable memory. Where the members of Group A and Group B that must be ordered are from the same PE, a DMB NSH is sufficient for this guarantee.

arm

Concurrency

Meanwhile

- Luc and I had developed a DSL called `cat` to describe concurrent behaviours, implemented in the `herd` tool (2014)
- We had also developed a tool called `diy` that would generate systematic families of litmus tests (2010-11)
- We would test deployed hardware to deduce experimental models, with the `litmus` tool (2009-10)
- We found a few bugs on deployed hardware (2009, 2014, 2015, ...)
- And proposed models for IBM Power and Armv7 in `cat`, that seemed sound w.r.t. our hardware experiments (2014)

MP+dmb.st+dmb.ld

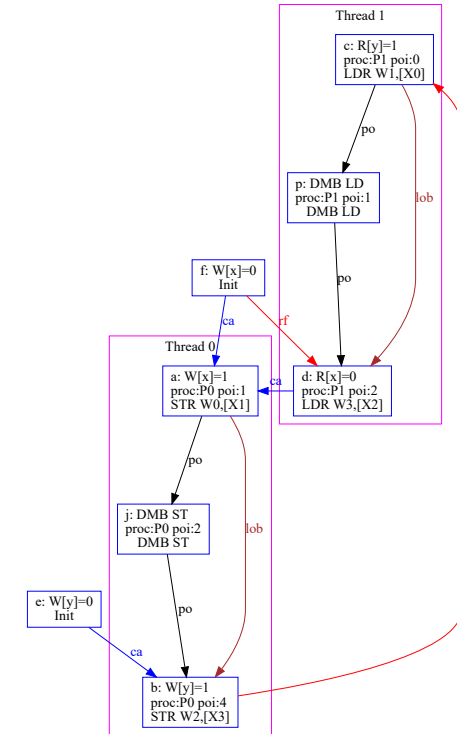
Architecturally Forbidden

Test

AArch64 MP+dmb.st+dmb.ld

```
{  
0:X1=x; 0:X3=y;  
1:X0=y; 1:X2=x;  
}  
P0      | P1      ;  
MOV W0,#1 | LDR W1,[X0] ;  
STR W0,[X1] | DMB LD      ;  
DMB ST      | LDR W3,[X2] ;  
MOV W2,#1   |          ;  
STR W2,[X3] |          ;  
exists (1:X1=1 /\ 1:X3=0)
```

Execution



Arm concurrency at a high level

- The Arm concurrency gives ordering requirements over memory accesses, intuitively stating which local orderings must be respected by hardware.
- Those constraints forbid certain shapes in program executions.
- The External visibility requirement forbids cycles in the Ordered-before relation, which consists of observations between threads (Observed-by relation) and local orderings (Hardware-requirements relation).

Arm ARM B.a (31 March 2017)

<https://developer.arm.com/documentation/ddi0487/latest/>

DDI0487B_a_armv8_arm.pdf
Page 88 of 6,354

Ordering and observability at a Location

Memory effects on a Location are related by the following relations:

Reads-from

A *Reads-from* relation that couples reads and writes to the same Location such that each read is paired with a single write in the program. A read R_2 of a Location Reads-from a write W_1 to the same Location if and only if R_2 takes its data from W_1 .

— **Note** —

The Reads-from relation represents a read being satisfied by a write and then returning the written data.

Coherence order

A *Coherence order* relation for each Location in the program that provides a total order on all writes from all coherent Observers to that Location, starting with a notional write of the initial value.

— **Note** —

The Coherence order of a Location represents the order in which writes to the Location arrive at memory.

DDI0487B_a_armv8_arm.pdf
Page 90 of 6,354

B2 The AArch64 Application Level Memory Model
B2.3 Definition of the ARMv8 memory model

Ordered-before

An arbitrary pair of Memory effects is ordered if it can be linked by a chain of ordered accesses consistent with external observation. A read or a write RW_1 is *Ordered-before* a read or a write RW_2 if and only if any of the following cases apply:

- RW_1 is Observed-by RW_2 .
- RW_1 is Dependency-ordered-before RW_2 .
- RW_1 is Atomic-ordered-before RW_2 .
- RW_1 is Barrier-ordered-before RW_2 .
- RW_1 is Ordered-before a read or a write that is Ordered-before RW_2 .

DDI0487B_a_armv8_arm.pdf
Page 89 of 6,354

Barrier-ordered-before

Barrier instructions order prior Memory effects before subsequent Memory effects generated by the same Observer. A read or a write RW_1 is *Barrier-ordered-before* a read or a write RW_2 from the same Observer if and only if RW_1 appears in program order before RW_2 and any of the following cases apply:

- RW_1 appears in program order before a DMB FULL that appears in program order before RW_2 .
- RW_1 is a write W_1 generated by a Store-Release instruction and RW_2 is a read R_2 generated by a Load-Acquire instruction.
- RW_1 is a read R_1 and either:
 - R_1 appears in program order before a DMB LD that appears in program order before RW_2 .
 - R_1 is generated by a Load-Acquire instruction.
- RW_2 is a write W_2 and either:
 - RW_1 is a write W_1 appearing in program order before a DMB ST that appears in program order before W_2 .
 - W_2 is generated by a Store-Release instruction.
 - RW_1 appears in program order before a write W_3 generated by a Store-Release instruction and W_2 is Coherence-after W_3 .

External visibility requirement

For a read or a write RW_1 that is Ordered-before a read or a write RW_2 , to the same Location and from a different Observer, the external visibility requirement requires that exactly one of the following statements is true:

- RW_2 is a write W_2 and RW_1 is Observed-by W_2 .
- RW_1 is a write W_1 and RW_2 is a read R_2 such that either:
 - W_1 is Observed-by R_2 .

B2-90 Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved. ARM DDI 0487B.a ID033117
Non-Confidential

- There is a write W_3 such that W_3 is Observed-by R_2 and W_1 is Observed-by W_3 .
- RW_1 and RW_2 are both reads R_1 and R_2 and there is a write W_3 such that either:
 - W_3 is Observed-by both R_1 and R_2 .
 - R_1 is Observed-by W_3 and W_3 is Observed-by R_2 .

Arm ARM B.b (25 September 2017)

*B2 The AArch64 Application Level Memory Model
B2.3 Definition of the ARMv8 memory model*

External visibility requirement

For a read or a write RW_1 from an **Observer** that is **Ordered-before** a read or a write RW_2 from a different **Observer**, the external visibility constraint requires that RW_2 is not **Observed-by** RW_1 . This means that an Architecturally well-formed execution must not exhibit a cycle in the **Ordered-before** relation.

Convention: correspondence between cat and manual English transliteration

For example: External visibility

In cat

```
let rec ob = obs
    | hw-reqs
    | ob; ob
```

irreflexive ob as external

In English

Ordered-before

An Effect E1 is **Ordered-before** an Effect E2 if and only if one of the following applies:

- E1 is a Read or Write Memory Effect RW1, E2 is a Read or Write Memory Effect RW2 and RW1 is **Observed-by** RW2
- E1 is **hardware-required-ordered-before** E2
- E1 is **Ordered-before** an Effect E3 and E3 is **Ordered-before** E2.

External visibility

An architecturally well-formed execution **must not exhibit a cycle** in the **ordered-before** relation.

Developing these definitions is a slow and partially social process

- We read the prose in the Arm ARM
- We test machines
- We interview our colleagues (architects, implementors, software developers): we show them hundreds to thousands of tests and ask them if they think the behaviours should be Allowed or Forbidden
- We extrapolate a set of definitions, which we regress against deployed hardware
- We ratify the definitions with Arm partners
- Then their transliteration lands in the Arm ARM

Between 2016 and now

- We added quite a few features (e.g. address translation) to the concurrency definitions
- Demo on the interface

Arm ARM J.a (21 April 2023)

Hardware-required-ordered-before

An effect E_1 is in Hardware-required-ordered-before an effect E_2 if one of the following applies:

- E_1 is in Translation-intrinsically-before E_2 .
- E_1 is in Tag-check-intrinsically-before E_2 .
- E_1 is an Explicit memory effect or an Implicit Tag Read Memory effect and E_2 is an Explicit memory effect or an Implicit Tag Read Memory effect or a MMU Fault effect or a Tag-Check Fault effect and E_1 is in Locally-ordered-before or Pick-locally-ordered-before E_2 .
- FEAT_ETTS2 is implemented, E_1 is an Explicit Memory effect, E_2 is an Implicit TTD Memory Read effect and all of the following apply:
 - E_1 is in program-order-before a TLBUncacheable Fault effect E_3 .
 - E_2 is in Translation-intrinsically-before E_3 .
- E_1 is in DSB-ordered-before E_2 .
- E_1 is in CSE-ordered-before E_2 .
- E_1 is a CSE effect and E_1 appears in program-order-before E_2 .
- E_1 is an Implicit TTD Read Memory effect R_1 , E_2 is a Write Memory effect W_2 , R_1 and W_2 are to the same Location and R_1 is in program-order-before W_2 .
- E_1 is an Implicit TTD Read Memory effect R_1 , E_2 is a Hardware Update effect W_2 , and R_1 and W_2 form a Read-Modify-Write.
- E_1 is an Explicit Memory effect and E_2 is a TLBUncacheable Fault effect, E_1 and E_2 are to the same Location and E_1 appears in program-order-before E_2 .
- E_1 is an Explicit Read Memory effect, E_2 is a TLBI effect and there is an Address Dependency from E_1 to E_2 .
- E_1 is an Explicit Read Memory effect, E_2 is a Hardware Update Write effect and there is a Control Dependency from E_1 to E_2 .
- There exists an effect E_3 such that E_1 is Hardware-required-ordered-before E_3 and E_3 is Hardware-required-ordered-before E_2 .

External visibility requirement

Ordered-before

An arbitrary pair of [Effects](#) is ordered if it can be linked by a chain of ordered effects consistent with external observation. An effect E_1 is *Ordered-before* an effect E_2 if one of the following cases apply:

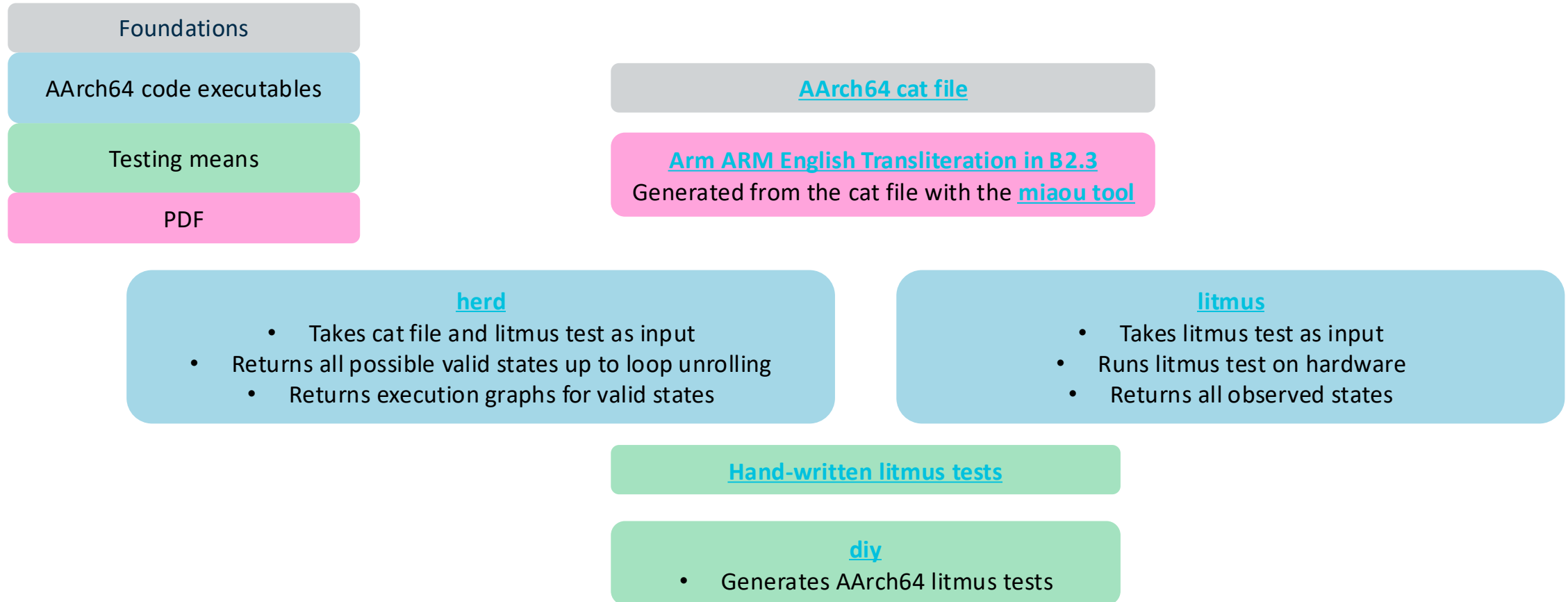
- E_1 is Hardware-required-ordered-before E_2 .
- E_1 is Hazard-ordered-before E_2 .
- E_1 is Observed-by either E_2 or an effect that belongs to the same single-copy-atomicity class as E_2 .
- E_1 is an Implicit TTD effect, and E_2 Reads-from E_1 .
- E_2 is an Implicit TTD effect, and E_2 Reads-from E_1 .
- E_2 is a TLBUncacheable-Successor of E_1 .
- E_2 is a Hardware-Update-Successor of E_1 .
- E_1 is Coherence-before E_2 , and either E_1 or E_2 is a Hardware Update effect.
- E_2 is TLBI-Coherence-after E_1 .
- E_1 is TLBI-ordered-before E_2 .
- E_1 is Ordered-before an effect E_3 and E_3 is Ordered-before E_2 .

The more the cat file grows, the more the prose grows

- This is very hard to transliterate accurately by hand
- So Luc and I invented a tool called miaou, which takes the cat and produces the very constrained English prose that appears in the Arm ARM
- We use miaou to generate (some of) the English prose since K.a (20 March 2024)

Herdtools: a toolset for AArch64 concurrency

Source material distributed at <https://github.com/herd/herdtools7>



For any query or comment please contact memory-model@arm.com

arm

Instruction semantics

MP+dmb.st+dmb.ld

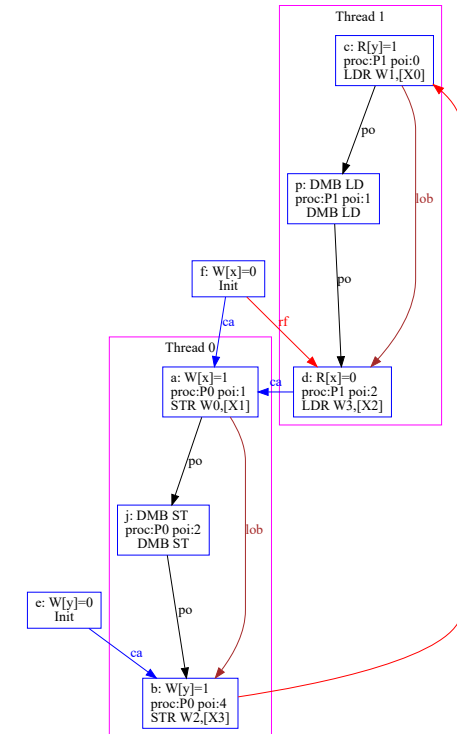
Architecturally Forbidden

Test

AArch64 MP+dmb.st+dmb.ld

```
{  
0:X1=x; 0:X3=y;  
1:X0=y; 1:X2=x;  
}  
P0      | P1      ;  
MOV W0,#1 | LDR W1,[X0] ;  
STR W0,[X1] | DMB LD      ;  
DMB ST      | LDR W3,[X2] ;  
MOV W2,#1   |          ;  
STR W2,[X3] |          ;  
exists (1:X1=1 /\ 1:X3=0)
```

Execution



Virtual and physical memory

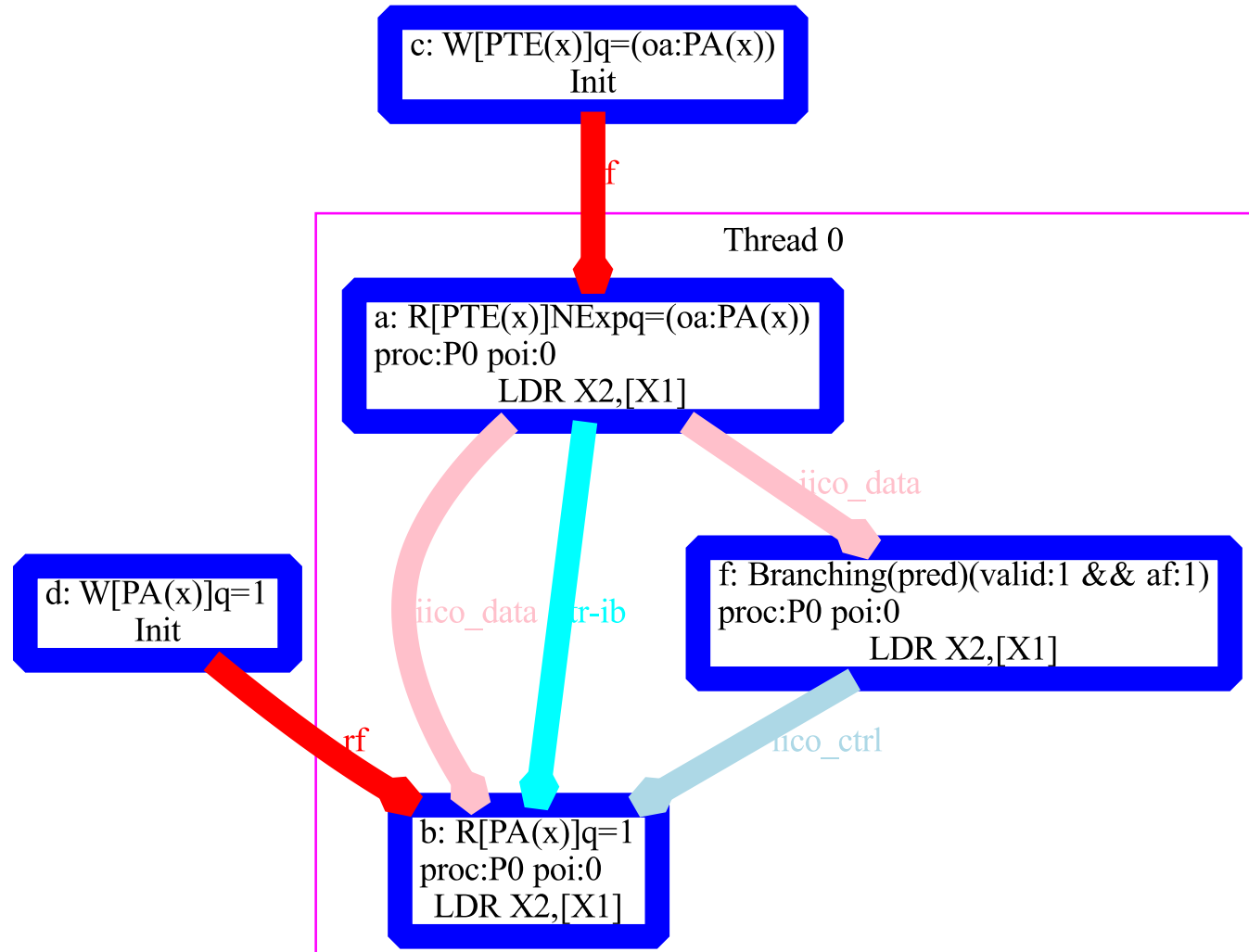
- In the previous examples I showed, we assumed that 'x' and 'y' were *physical memory locations*.
- But usually one interacts with *virtual memory addresses* rather than physical memory directly.
- Virtual memory gives an abstract view of the physical memory resources, which amongst other benefits:
 - frees software developers from having to manage a shared memory space, and
 - affords them more memory than might be physically available.
- The mapping from virtual addresses to physical locations resides in a collection of *page tables*. In other words, page tables are used to translate the virtual addresses seen by software into physical locations used by the hardware

Semantics of LDR

With page table considerations

AArch64 LDR

```
{  
  0:X2=x;  
  0:X1=0;  
  x=1;  
}  
P0      ;  
LDR X1,[X2];  
exists(0:X1=1)
```



Page tables

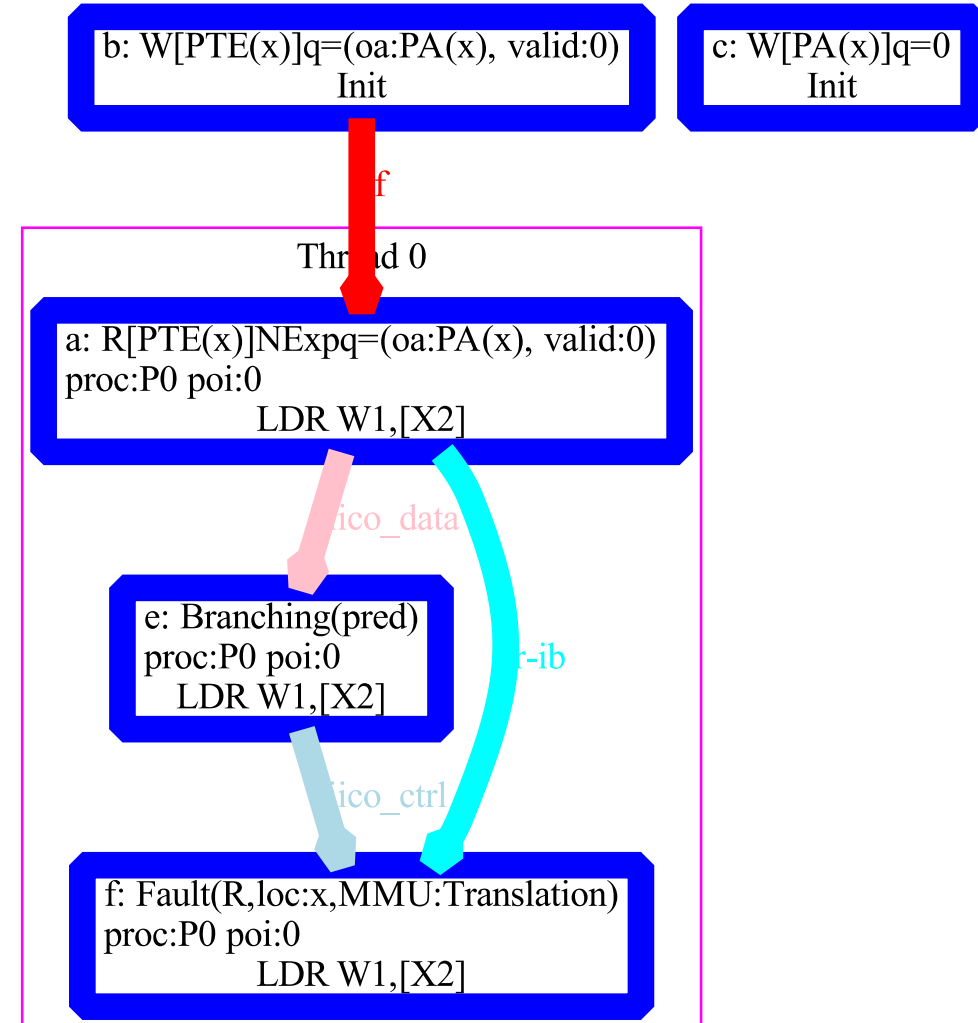
- Page tables are used to translate the virtual addresses seen by software into physical locations used by the hardware
- Each *page table entry* (PTE) holds a flag indicating whether the entry is *valid or not*:
 - If it is valid, reading the PTE gives the physical location corresponding to a virtual address;
 - If it is not valid, accessing a virtual address via that PTE raises a *fault*.

Semantics of LDR

When the valid bit is 0

AArch64 LDRv0

```
{  
  PTE(x)=(valid:0);  
  0:X2=x;  
}  
P0      ;  
L0:LDR W1,[X2] ;  
forall(fault(P0:L0,x))
```

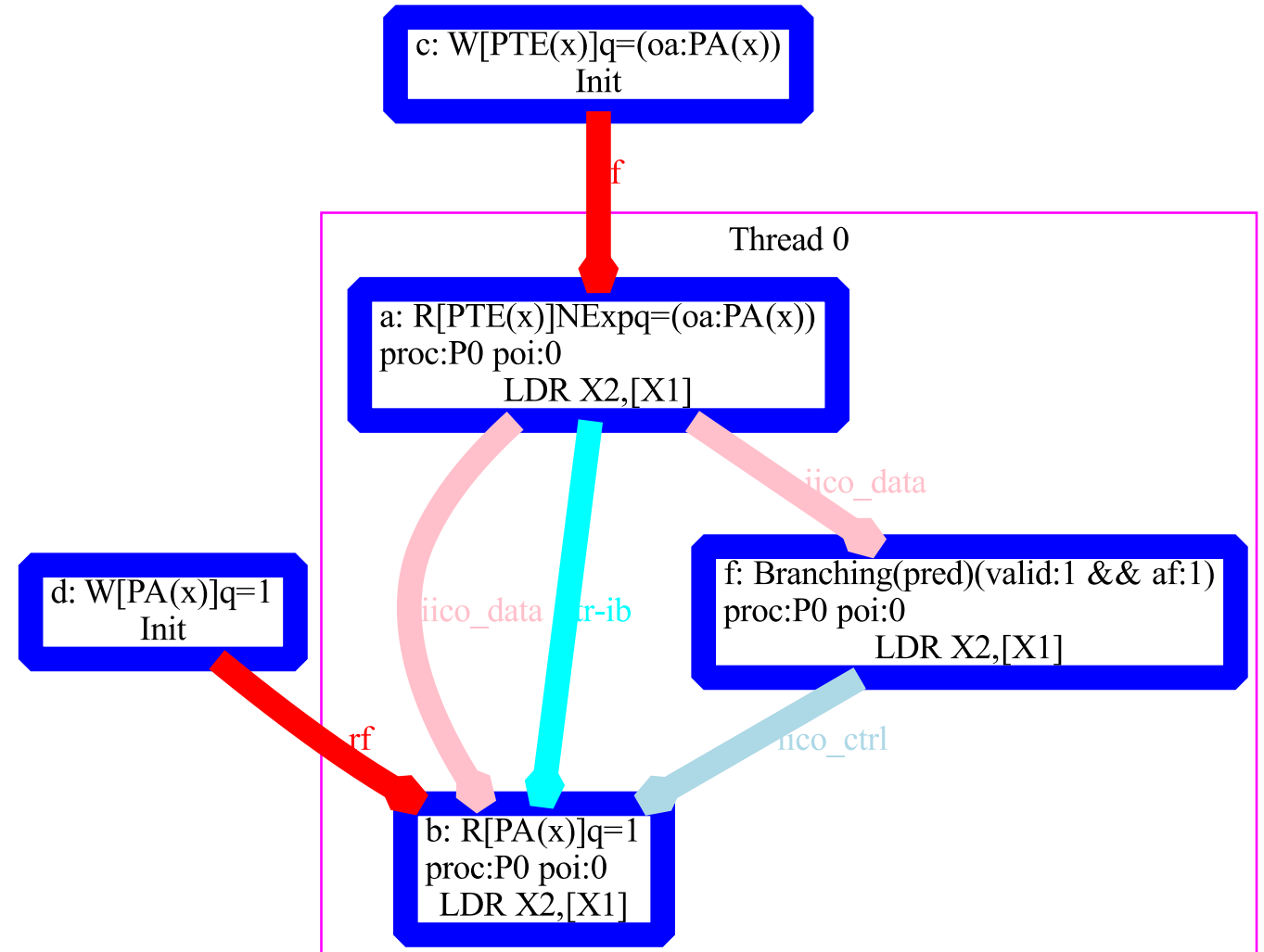


Semantics of LDR

When the valid bit is 1

AArch64 LDRv1

```
{  
  PTE(x)=(valid:1);  
  x=1;  
  0:X1=x;  
}  
P0      ;  
L0:     ;  
LDR X2,[X1] ;  
forall(0:X2=1 /\ ~fault(P0:L0,x))
```



PTE values

A page table entry value such as $\text{PTE}(x)$ is a tuple $(oa, \text{valid}, af, db, dbm)$, where:

- oa is the “output address”
- valid is the valid bit
- af is the Access Flag
- db is the Dirty Bit
- dbm is the Dirty Bit Management bit

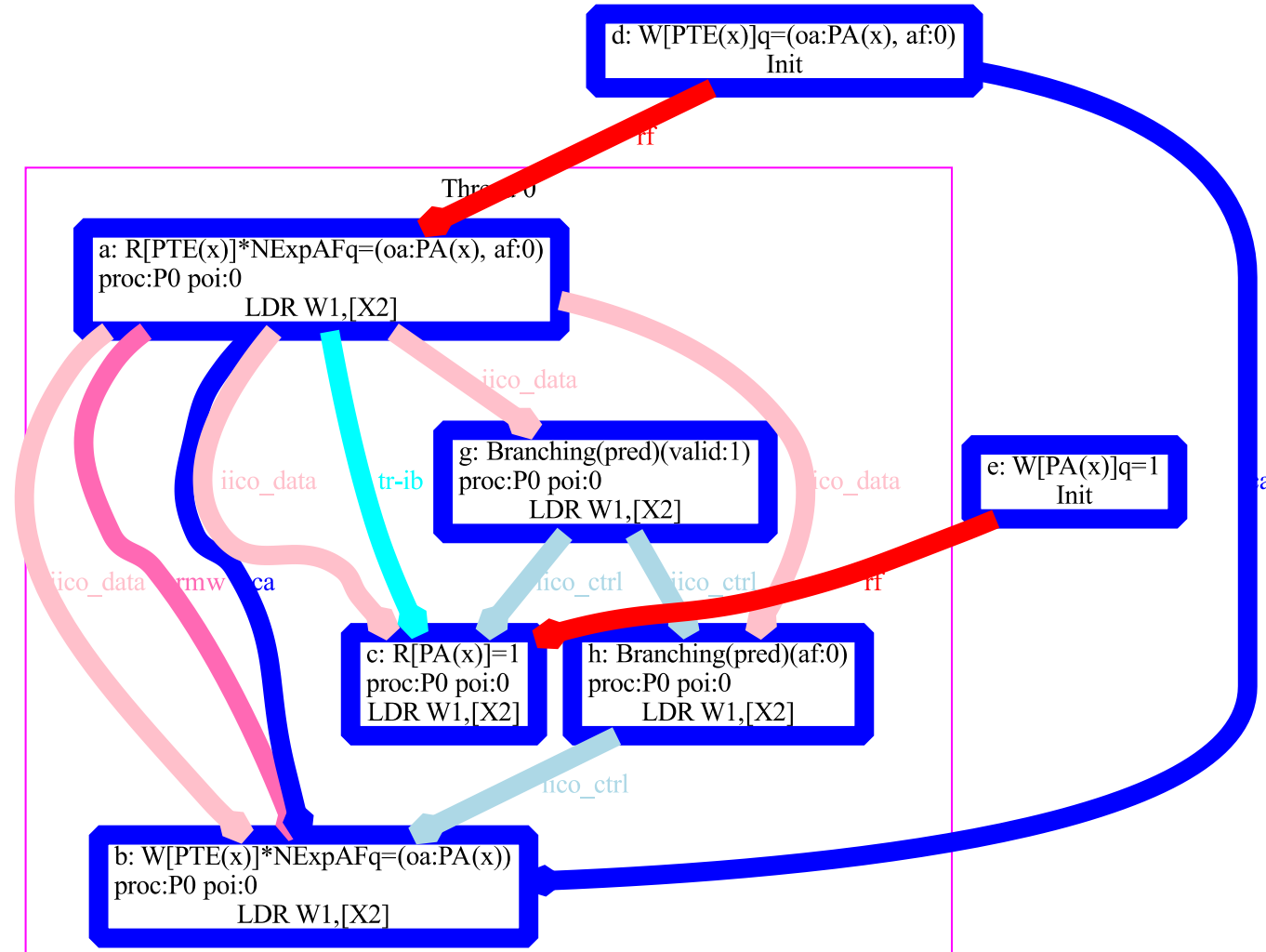
Semantics of LDR in herd

When af:0 and TTHM HA is on

AArch64 LDRaf0-HA

TTHM=P0:HA

```
{  
  [PTE(x)]=(oa:PA(x),af:0);  
  x=1;  
  0:X2=x;  
}  
P0      ;  
L0:     ;  
LDR W1,[X2] ;  
forall(0:X1=1 /\ ~fault(P0:L0,x))
```

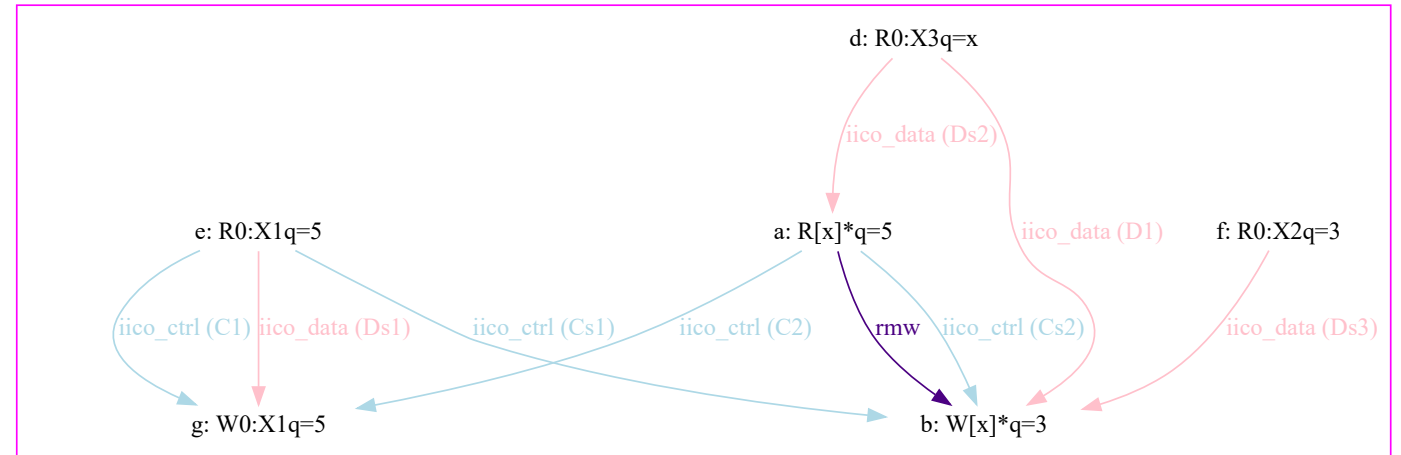


Semantics of CAS in herd

When the CAS is successful

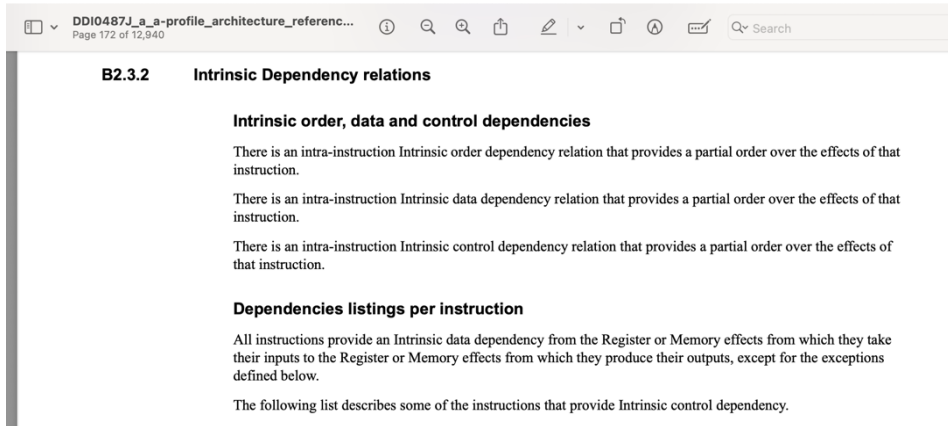
AArch64 cas-ok

```
{ x = 5;  
  0: X1 = 5;  
  0: X2 = 3;  
  0: X3 = x; }  
P0      ;  
CAS X1, X2, [X3] ;  
exists (x = 3  $\wedge$  0: X1 = 5)
```



We record certain Intrinsic dependencies in the Arm ARM

Example: Intrinsic dependencies for CAS in the Arm ARM J.a



Compare-and-Swap instructions

In all cases all of the following apply to the effects generated by the CAS $Xs, Xt, [Xn]$ instruction:

- D_1 : There is an Intrinsic data dependency from the Read Register effect of Xn to the Read memory effect of the Memory Location addressed by Xn .
- C_1 : There is an Intrinsic control dependency from the Read Register effect of Xs to the Write Register effect of Xs .
- C_2 : There is an Intrinsic control dependency from the Read memory effect of the Memory Location addressed by Xn to the Write Register effect of Xs .

If the CAS instruction fails all of the following apply:

- Df_1 : There is an Intrinsic data dependency from the Read memory effect of the Memory Location addressed by Xn to the Write Register effect of Xs .

If the CAS instruction succeeds all of the following apply:

- Ds_1 : There is an Intrinsic data dependency from the Read Register effect of Xs to the Write Register effect of Xs .
- Ds_2 : There is an Intrinsic data dependency from the Read Register effect of Xn to the Write Memory effect of the Memory Location addressed by Xn .
- Ds_3 : There is an Intrinsic data dependency from the Read Register effect of Xt to the Write Memory effect of the Memory Location addressed by Xn .
- Cs_1 : There is an Intrinsic control dependency from the Read Register effect of Xs to the Write Memory effect of the Memory Location addressed by Xn .
- Cs_2 : There is an Intrinsic control dependency from the Read memory effect of the Memory Location addressed by Xn to the Write Memory effect of the Memory Location addressed by Xn .

The ARM logo, consisting of the lowercase letters 'arm' in a white, sans-serif font, is positioned on the left side of the slide. The background is a solid orange color with a subtle grid of small white plus signs.

Architecture Specification Language

ASL

- Each instruction in the ISA has a description of its operation (what it is meant to do) written in ASL
- Demo of the code for LDR, CAS

Compute the herd graphs from the ASL

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == 'l1' then 64 else 32;
constant integer datasize = 8 << scale;
```

Operation

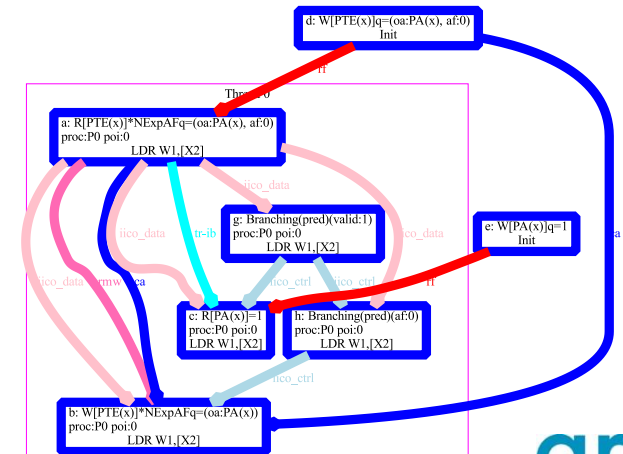
```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;
```

```
boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, PSTATE.privileged, memOp);
```

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
address = GenerateAddress(address, offset,
data = Mem[address, datasize DIV 8, accdes
X[t, regsize] = ZeroExtend(data, regsize);

// AArch64.SettingAccessFlagPermitted()
// =====
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.SettingAccessFlagPermitted(FaultRecord fault)
    if fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode IN {Fault_Alignment, Fault_Permission} then
        return ConstrainUnpredictableBool(Unpredictable_AFUPDATE);
    else
        return FALSE;
```



arm

Detour:
definition of ASL

A (brief, simplified) history of ASL



ASLRef: `if b then e1 else e2`

reference
implementation

15.5 Conditional Expressions

15.5.1 Syntax

$\text{expr} \rightarrow \text{"if" expr "then" expr "else" expr}$

15.5.2 Abstract Syntax

$\text{expr} \rightarrow E_Cond(\overset{\text{condition}}{\text{expr}}, \overset{\text{then}}{\text{expr}}, \overset{\text{else}}{\text{expr}})$

15.5.3 Typing

TypingRule.ECond

Example: [Lowest Common Ancestor](#) shows examples of typing conditional expressions.

Prose

All of the following apply:

- `e` denotes a conditional expression with condition `e_cond` with two options `e_true` and `e_false`:
- annotating `e_cond` in `tenv` results in `(t_cond, e_cond', ses_cond)` */#TE*;
- annotating `e_true` in `tenv` results in `(t_true, e_true', ses_true)` */#TE*;
- annotating `e_false` in `tenv` results in `(t_false, e_false', ses_false)`;
- obtaining the lowest common ancestor of `t_true` and `t_false` results in `t` */#TE*;
- `new_e` is the condition `e_cond'` with two options `e_true'` and `e_false'`, that is, `E_Cond(e_cond', e_true', e_false')`;
- define `ses` as the union of `ses_cond`, `ses_true`, and `ses_false`.

Formally

```
annotate_expr(tenv, e_cond)  $\xrightarrow{\text{type}}$  (t_cond, e_cond', ses_cond) // #TE
annotate_expr(tenv, e_true)  $\xrightarrow{\text{type}}$  (t_true, e_true', ses_true) // #TE
annotate_expr(tenv, e_false)  $\xrightarrow{\text{type}}$  (t_false, e_false', ses_false) // #TE
lowest_common_ancestor(t_true, t_false)  $\xrightarrow{\text{type}}$  t // #TE
ses := ses_cond  $\cup$  ses_true  $\cup$  ses_false
-----
annotate_expr(E_Cond(e_cond, e_true, e_false))  $\xrightarrow{\text{type}}$ 
(t, E_Cond(e_cond', e_true', e_false'), ses)
```

```
let expr :=
  (* ... *)
  | IF; ~=expr; THEN; ~=expr; ELSE; ~=expr; < E_Cond >
```

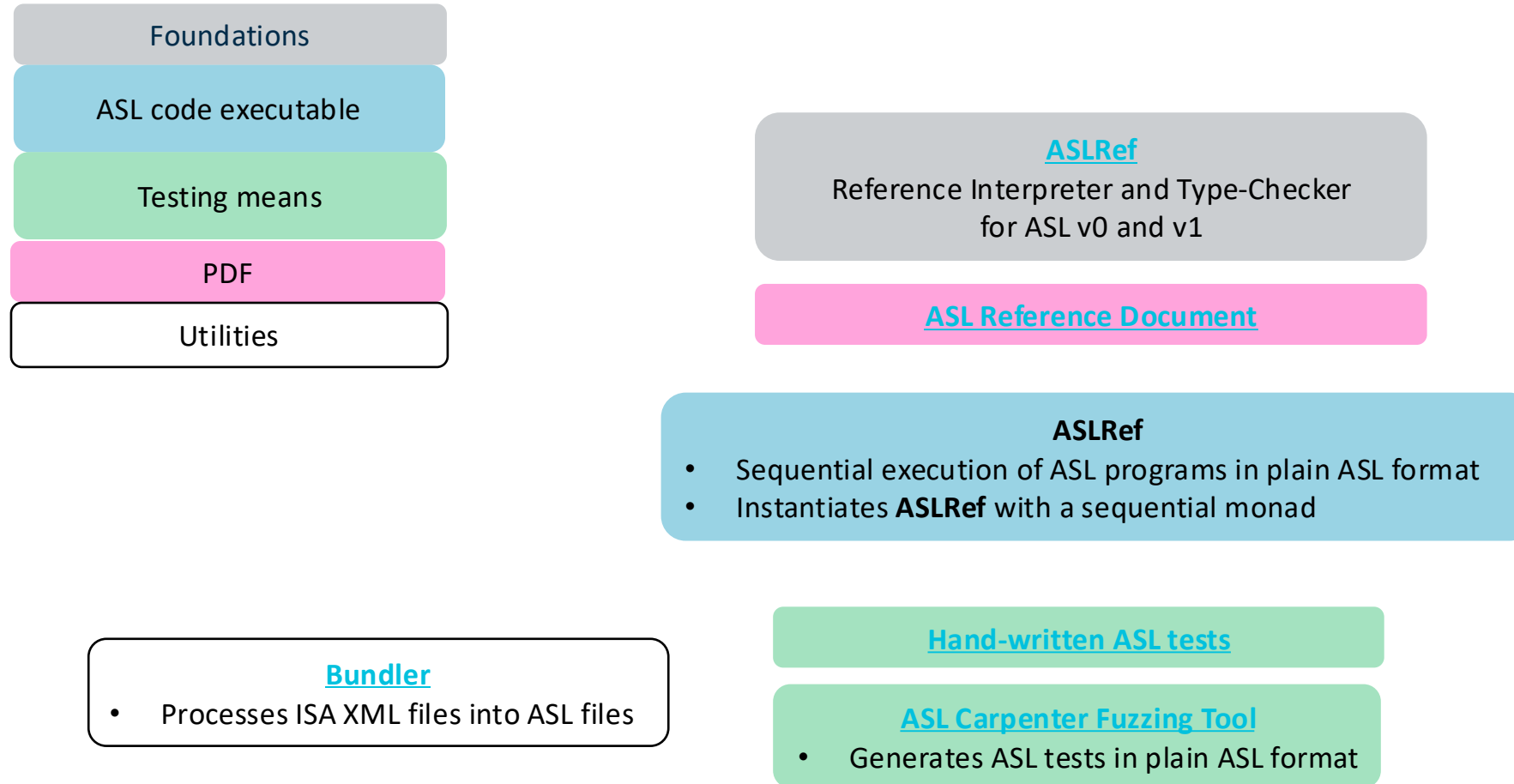
```
type expr_desc =
  (* ... *)
  | E_Cond of expr * expr * expr
```

```
| E_Cond (e_cond, e_true, e_false) ->
  let t_cond, e_cond', ses_cond = annotate_expr env e_cond in
  let+ () = check_structure_boolean ~loc env t_cond in
  let t_true, e_true', ses_true = annotate_expr env e_true in
  let t_false, e_false', ses_false = annotate_expr env e_false in
  let t =
    best_effort t_true (fun _ ->
      match Types.lowest_common_ancestor ~loc:e env t_true t_false with
      | None ->
        fatal_from ~loc (Error.UnreconcilableTypes (t_true, t_false))
      | Some t -> t)
  in
  let ses = SES.union3 ses_cond ses_true ses_false in
  (t, E_Cond (e_cond', e_true', e_false') |> here, ses)
```

formal
definitions
prose + maths

ASLRef: a toolset for ASL

Source material distributed at <https://github.com/herd/herdtools7/tree/master/asllib>



For any query or comment please contact atg-formal@arm.com

Stdlib proofs: Sint

Conjecture to be proved by ACL2

```
def-asl-subprogram sint-correct
  :function "SInt"
  :params (n)
  :args (x)
  :hyps (< 0 n.val)
  :return-values ((v_int (sign-ext n.val
x.val)))
```

Narration

This form instructs ACL2 to do a correctness proof for function "SInt", calling its one input parameter `n` and its one argument `x`, assuming that `n's value is greater than 0`, and showing that it produces a single integer return value that is the sign-extension of `x` at width `n`.

Stdlib proofs: Sint

Theorem proved by ACL2

```
1. (DEFTHM SINT-CORRECT
2.   (IMPLIES
3.     (AND (SUBPROGRAMS-MATCH '("Sint" "UInt")
4.           (GLOBAL-ENV->STATIC (ENV->GLOBAL ENV))
5.           (STDLIB-STATIC-ENV))
6.     (< 0 (V_INT->VAL N))
7.     (EQUAL (VAL-KIND N) :V_INT)
8.     (EQUAL (VAL-KIND X) :V_BITVECTOR)
9.     (EQUAL (V_BITVECTOR->LEN X) (V_INT->VAL N))
10.    (<= 1 (IFIX CLK)))
11.   (LET* ((RES (MV-NTH 0 (EVAL_SUBPROGRAM ENV "Sint" (LIST N)
12.                                     (LIST X))))
13.     (SPEC
14.       (EV_NORMAL
15.         (FUNC_RESULT (LIST (V_INT (SIGN-EXT (V_INT->VAL N)
16.                                               (V_BITVECTOR->VAL X))))
17.         (ENV->GLOBAL ENV))))))
18.   (EQUAL RES SPEC))))
```

Narration

Assuming:

- Env has the correct definitions for functions Sint and UInt (line 3)
- N is an ASL value of integer type, greater than 0 (lines 6,7)
- X is an ASL value of bitvector type, with width equal to N (lines 8,9)
- The interpreter recursion limit is at least 1 (line 10)

Then:

the result of evaluating the ASL subprogram "Sint" on input parameter N and input argument X (lines 11,12)

Equals

- a normal function result (no errors or exceptions) (line 14)
- producing a single return value that is the sign-extend of X at width N (lines 15,16)
- and an unmodified global environment (line 17).

ISA Proofs: ADC

Theorem proved by ACL2

```
1. (DEFTHM INSTR_ADC-CORRECT
2.   (IMPLIES (AND (type-consistent storage env)
3.                 (unsigned-byte-p 5 n)
4.                 (b* ((dst-idx (bitops::part-select instr :low 0 :high 4)))
5.                     (and (not (equal n dst-idx)) (not (equal dst-idx 31))))
6.                 (stringp s) (not (equal s "_R"))))
7.   (b* ((global-env (make-global-env :static (static-env) :storage storage))
8.         (env (make-env :global global-env :local (empty-local-env)))
9.         (_r (extract "_R" storage))
10.        (pspace (v_record->rec (extract "PSTATE" storage)))
11.        ((v_bitvector c) (extract "C" pspace))
12.        (dsz (if (logbitp 31 instr) 64 32))
13.        ((list datasize op1-idx op2-idx) (decode-instr instr))
14.        ((v_bitvector op1) (reg-read op1-idx dsz _r))
15.        ((v_bitvector op2) (reg-read op2-idx dsz _r))
16.        ((v_bitvector spec) (v_bitvector dsz (adc-res-spec dsz op1.val
17.                                                         op2.val c.val)))
18.        ((mv res orac) (eval_subprogram env "instr_adc" nil (list
19.                                                         (v_bitvector 32 instr) :clk 100000
20.                                                         (storage-new (extract-storage res))
21.                                                         (_r-new (extract "_R" storage-new))
22.                                                         ((v_bitvector dst) (reg-read dst-idx dsz _r-new))
23.                                                         ((v_bitvector regn-new) (reg-read n dsz _r-new))
24.                                                         ((v_bitvector regn) (reg-read n dsz _r))
25.                                                         (other (extract s storage))
26.                                                         (other-new (extract s storage-new)))
27.                                                         (AND (normalp res) (not (errorp res))
28.                                                         (equal dst.val spec.val)
29.                                                         (equal regn-new.val regn.val)
30.                                                         (equal other-new other))))))
```

Narration

Assuming:

- 'storage' variable is of a type consistent with the AST loaded from the prototype (line 2)
- and 'n' is a 5-bit index into register file different from the index of the destination register and 'n' is not 31 (lines 3-5)
- and 's' is any name of a storage element different from the register file _R (line 6)

Then:

- Let 'env' be an initial environment with the global environment defined by the AST and empty local environment (lines 7-8)
- Let c be the value of PSPACE.C (lines 9-11)
- Let dsz, op1 and op2 be values decoded from the 'instr' and loaded from register file, resp. (lines 12-15)
- The evaluation of the ASL function stored in the 'storage' of 'env' under name instr_adc, starting from the initial environment 'env', and argument 'instr'
- terminates normally and without an error (line 25)

And for the resulting environment the following is true:

- The destination register changed consistently with the specification(adc-res-spec dsz op1 op2 C) (lines 16,26)
- All other general registers remain unchanged (line 27)
- All other storage elements (e.g. PSTATE) remain unchanged (line 28)

arm

Instruction semantics:
ASL and herd graphs

Where we wanted to get to

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
constant integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;
```

```
boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGP(MemOnLoad, PSTATE.privileged, mdtbl);
```

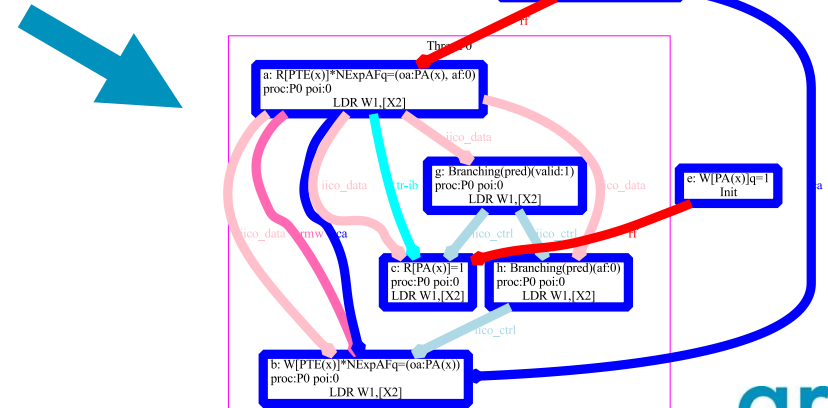
```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
```

```
address = GenerateAddress(address, offset,
```

```
data = Mem[address, datasize DIV 8, accdes  
X[t, regsize] = ZeroExtend(data, regsize);
```

```
// AArch64.SettingAccessFlagPermitted()
// =====
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.SettingAccessFlagPermitted(FaultRecord fault)
    if fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode IN {Fault_Alignment, Fault_Permission} then
        return ConstrainUnpredictableBool(Unpredictable_AFUPDATE);
    else
        return FALSE;
```



CAS (success case) at a high level

Litmus test

AArch64 cas-ok

```
{ x = 5;  
  0: X1 = 5;  
  0: X2 = 3;  
  0: X3 = x; }  
P0      ;  
CAS X1, X2, [X3] ;  
exists (x = 3 /\ 0: X1 = 5)
```

In words

CAS X1, X2, [X3] (success case):

- reads register X3 and gets a location x from X3
- reads the new value (3) given in X2
- reads the value to compare to (5) given in X1
- reads the value stored in memory location x (5)
- writes the value stored in memory (5) into register X1
- since the value in memory (5) is equal to the value to compare to (5), the CAS is successful
- therefore we write the new value (3) into memory location x

CAS (success case) at a high level

Litmus test

AArch64 cas-ok

```
{ x = 5;  
  0: X1 = 5;  
  0: X2 = 3;  
  0: X3 = x; }  
P0      ;  
CAS X1, X2, [X3] ;  
exists (x = 3 /\ 0: X1 = 5)
```

In words – zooming in

CAS X1, X2, [X3] (success case):

[...]

- reads the value to compare to (5) given in X1 – we will call this “the CAS register read”
- reads the value stored in memory from location x (5) – we will call this “the CAS memory read”
- writes the value stored in memory (5) into register X1 – we will call this “the CAS register write”

[...]

The question we are going to examine is: which value 5 do we write into register X1?

The one coming from the CAS register read, or the one coming from the CAS memory read?

CAS Intrinsic dependencies

As listed in B2.3 pp. B2-173 in J.a

Compare-and-Swap instructions

In all cases all of the following apply to the effects generated by the CAS $Xs, Xt, [Xn]$ instruction:

- D_1 : There is an Intrinsic data dependency from the Read Register effect of Xn to the Read memory effect of the Memory Location addressed by Xn .
- C_1 : There is an Intrinsic control dependency from the Read Register effect of Xs to the Write Register effect of Xs .
- C_2 : There is an Intrinsic control dependency from the Read memory effect of the Memory Location addressed by Xn to the Write Register effect of Xs .

If the CAS instruction fails all of the following apply:

- Df_1 : There is an Intrinsic data dependency from the Read memory effect of the Memory Location addressed by Xn to the Write Register effect of Xs .

If the CAS instruction succeeds all of the following apply:

- DS_1 : There is an Intrinsic data dependency from the Read Register effect of Xs to the Write Register effect of Xs .
- DS_2 : There is an Intrinsic data dependency from the Read Register effect of Xn to the Write Memory effect of the Memory Location addressed by Xn .
- DS_3 : There is an Intrinsic data dependency from the Read Register effect of Xt to the Write Memory effect of the Memory Location addressed by Xn .
- CS_1 : There is an Intrinsic control dependency from the Read Register effect of Xs to the Write Memory effect of the Memory Location addressed by Xn .
- CS_2 : There is an Intrinsic control dependency from the Read memory effect of the Memory Location addressed by Xn to the Write Memory effect of the Memory Location addressed by Xn .

CAS Intrinsic dependencies

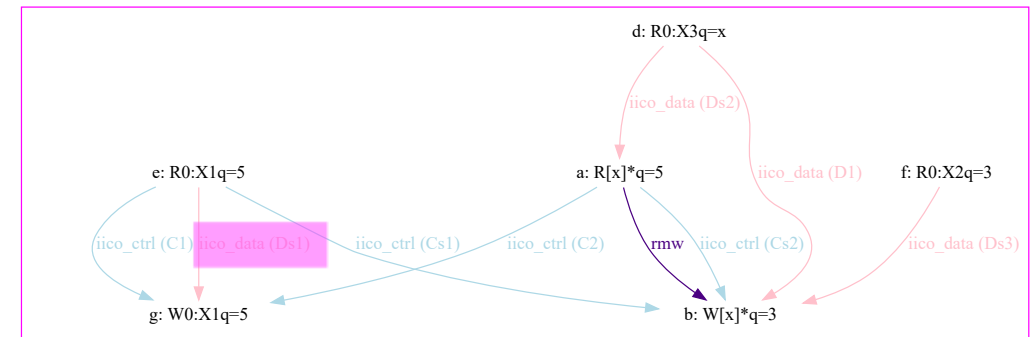
handwritten in herd as per B2.3

CAS AArch64 litmus test – success case

AArch64 cas-ok

```
{ x = 5;  
  0: X1 = 5;  
  0: X2 = 3;  
  0: X3 = x; }  
P0      ;  
CAS X1, X2, [X3] ;  
exists (x = 3 /\ 0: X1 = 5)
```

CAS herd graph as per B2.3 – success case



CAS Intrinsic dependencies

As conveyed by the Arm ARM ASL code for CAS

In the execute part of CAS, we can find the following code:

```
AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release, tagchecked);
comparevalue = X[s, datasize];
newvalue = X[t, datasize];
[...]
data = MemAtomic(address, comparevalue, newvalue, accdesc);
X[s, regsize] = ZeroExtend(data, regsize);
```

We can see that the value written in the register (the variable 'data') is the value returned from MemAtomic.

This is a simplified excerpt from the definition of MemAtomic:

```
bits(size) MemAtomic(bits(64) address, bits(size) cmpoperand,
    bits(size) operand, AccessDescriptor accdesc)
// All observers in the shareability domain observe the following load and store atomically.
oldvalue = PhysMemRead(memaddrdesc, bytes, accdesc);
if comparevalue == oldvalue then
    PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);
return oldvalue;
```

The code for MemAtomic returns the value read from memory (stored in the variable 'oldvalue').

CAS Intrinsic dependencies

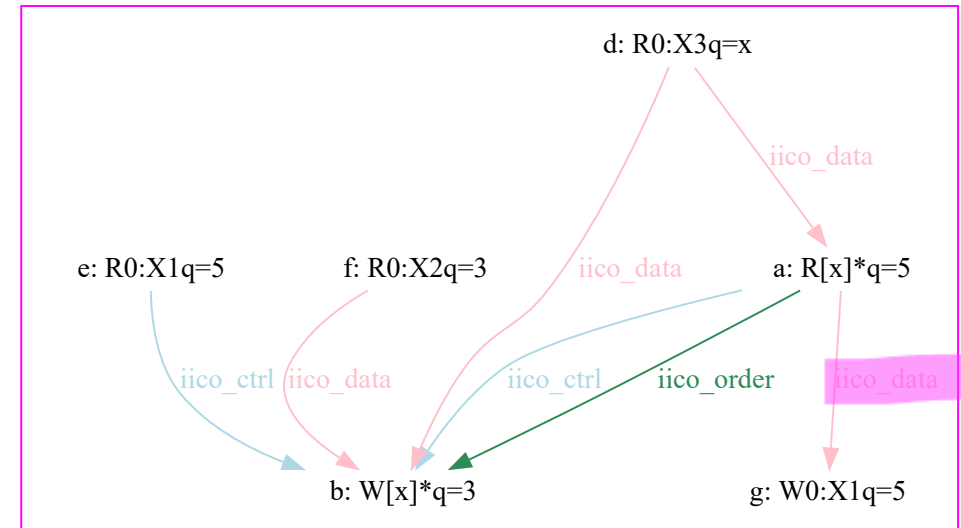
computed in herd from the Arm ARM ASL code: herd7 -variant ASL

CAS AArch64 litmus test – success case

AArch64 cas-ok

```
{ x = 5;  
  0: X1 = 5;  
  0: X2 = 3;  
  0: X3 = x; }  
P0      ;  
CAS X1, X2, [X3] ;  
exists (x = 3 /\ 0: X1 = 5)
```

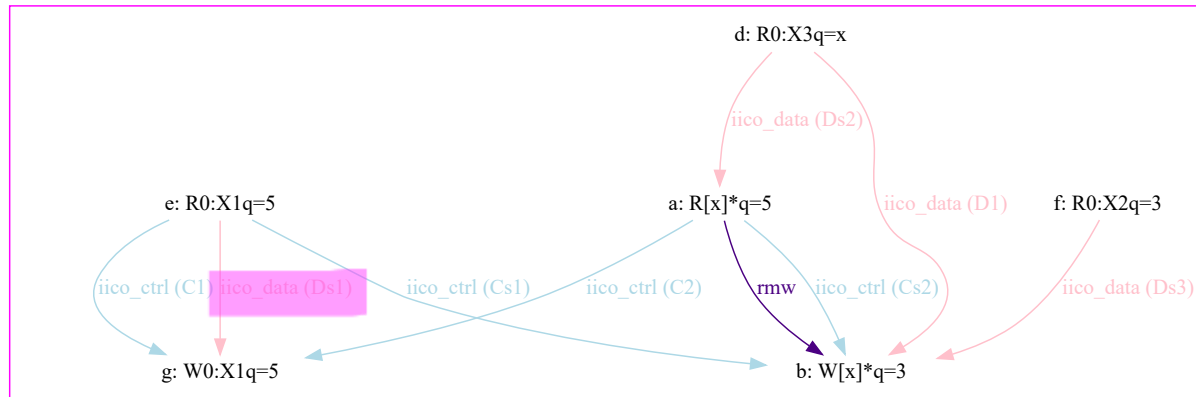
CAS herd graph computed from ASL – success case



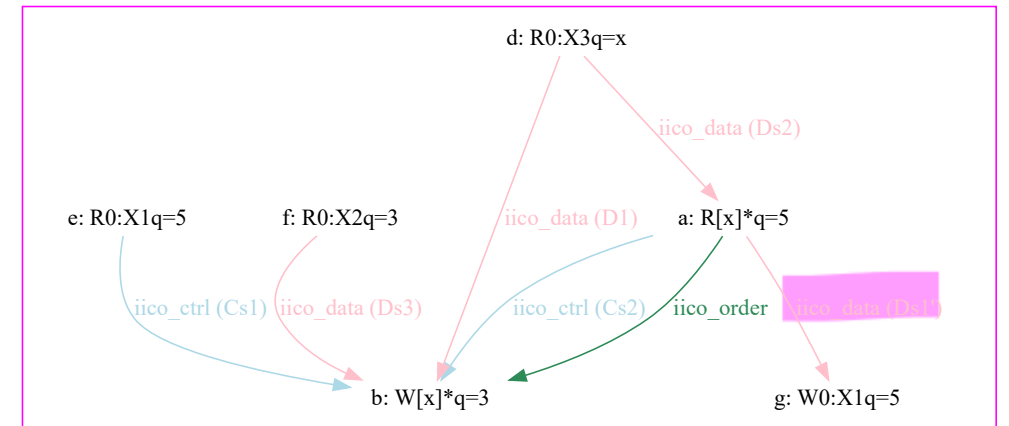
Comparing CAS Intrinsic dependencies

success case

herd graph with handwritten semantics as per B2.3



herd graph computed from Arm ARM ASL code



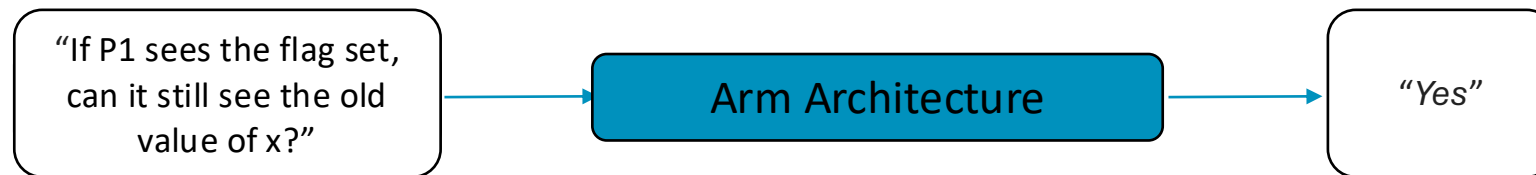
arm

From the Arm ARM
to

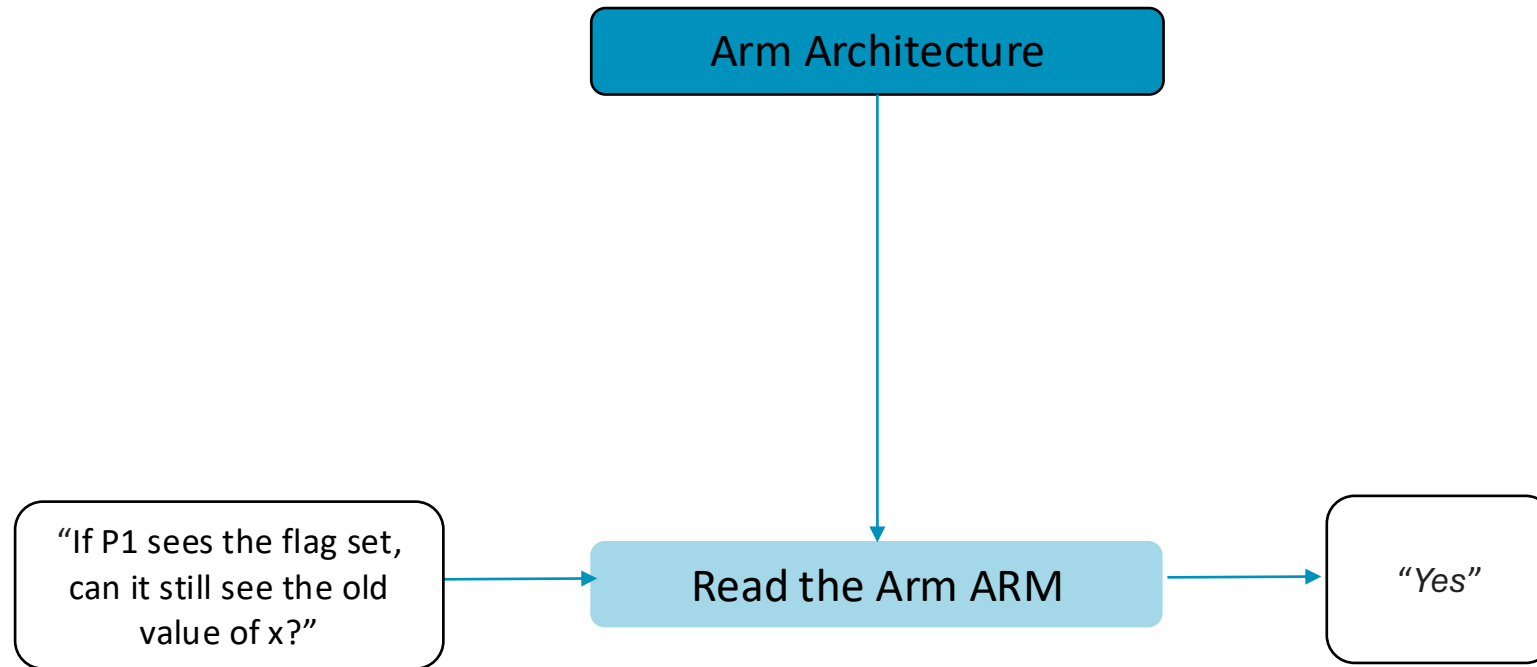
formal and executable specifications

Viewing the Architecture as an executable artefact

Specifications



Read the Arm ARM



The architecture can be seen as an executable artefact

What does that mean?

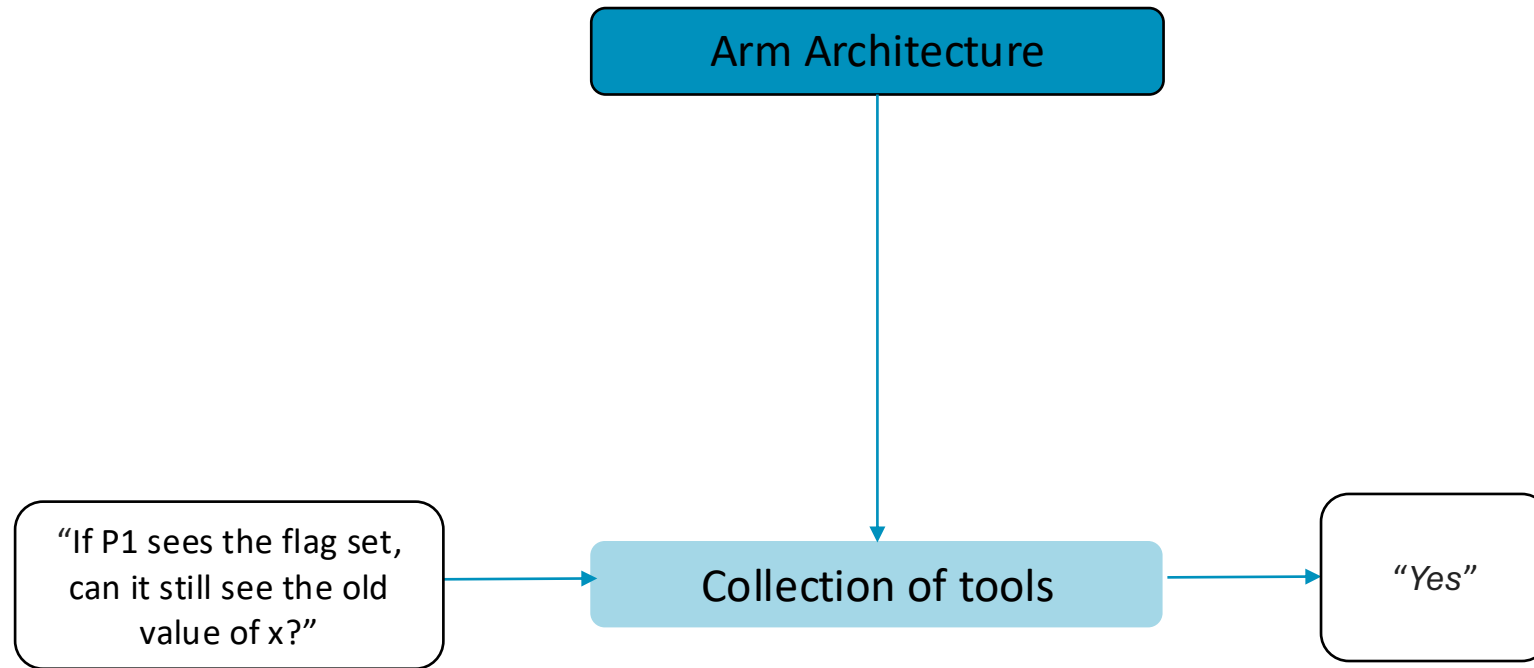
An approximate view is that the architecture is the product of:

- the “basic blocks” (e.g. state components and instruction semantics) on the one hand
- how these blocks combine (e.g. concurrency) on the other hand

This indicates that this artefact should perhaps rather be:

- several artifacts, e.g.:
 - Something to interpret the instruction semantics
 - Something to orchestrate the concurrency model
- operating in tandem

Formal and Executable specifications



arm

Architectural Data

Different types of data

Contractually, the Architecture is what is given in the Arm ARM

<https://developer.arm.com/documentation/ddi0487/latest/>

The AArch64 Virtual Memory System Architecture
D8.1 Address translation

D8.1 Address translation

- R_DWFWP** If an implementation is executing in AArch64 state, then that implementation uses either or both of the VMSAv8-64 and the VMSAv9-128 translation systems.
- R_RKHHV** All of the following determine whether a translation stage uses the VMSAv9-128 translation system:
- For stage 1 translations in the EL1&0 translation regime, the **Effective value** of **TCR2_EL1.D128** is 1.
 - For stage 1 translations in the EL2&0 translation regime, the **Effective value** of **TCR2_EL2.D128** is 1.
 - For stage 1 translations in the EL3 translation regime, the **Effective value** of **TCR_EL3.D128** is 1.
 - For stage 2 translations, the **Effective value** of **VTCR_EL2.D128** is 1.
- Otherwise, the translation stage uses the VMSAv8-64 translation system.
This applies even when the translation stage is disabled.
- I_CCTQS** Address translation converts the addresses used by instructions to the addresses used by the physical memory system.
- R_RCRPHV** When a data address or instruction address is used in an instruction, it is a *virtual address* (VA). This includes any address stored in one of the following registers:
- Program counter* (PC).
 - Stack pointers* (SP).
 - Link register* (LR).
 - Exception link register* (ELR).
- R_CCRDGS** When an access is made to the physical memory system, a *physical address* (PA) is used.
- R_KCNRX** An address translation maps a VA to a PA.
- R_CCCQO** An address translation requires one of the following:
- A single translation stage, stage 1.
 - Two sequential translation stages, stage 1 and stage 2.
- R_ZKJWW** An address translation stage maps an *input address* (IA) to an *output address* (OA).

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
constant integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

DDI0487L_a_a-profile_ar...
Page 256 of 14,568

ARM DDI 0487
La

Copyright © 2013-2024 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B2-255

The AArch64 Application Level Memory Model
B2.3 Ordering requirements defined by the formal concurrency model

- All of the following apply:
 - One of the following applies:
 - E₁ is an Explicit Memory Effect generated by an instruction with Acquire semantics.
 - E₁ is an Explicit Memory Effect generated by an instruction with AcquirePC semantics.
 - There is an Intrinsic Order Dependency from E₁ to E₂.
 - One of the following applies:
 - E₂ is an Explicit Memory Effect.
 - E₂ is an Implicit Tag Memory Read Effect.
 - E₂ is an MMU Fault Effect.
- All of the following apply:
 - One of the following applies:
 - E₁ is an Explicit Memory Effect.
 - E₁ is an Implicit Tag Memory Read Effect.
 - E₁ appears in program order before E₂.
 - E₂ is an Explicit Memory Effect or a Fault Effect and generated by an instruction with Release semantics.
- All of the following apply:
 - One of the following applies:
 - E₁ is an Explicit Memory Effect.
 - E₁ is an Implicit Tag Memory Read Effect.
 - There is an Intrinsic Order Dependency from E₁ to E₂.
 - E₂ is an Explicit Memory Effect or a Fault Effect and generated by an instruction with Release semantics.

Locally-ordered-before

An Effect E₁ is *Locally-ordered-before* an Effect E₂ if one of the following applies:

- E₁ is Tag-check-intrinsically-before E₂.
- E₁ is Translation-intrinsically-before E₂.

72

© 2026 Arm Limited

arm

Three examples of Architectural Data types

- English prose written by hand
- ASL for instruction semantics
- English prose automatically produced by miaou from the cat file

Overall we care a lot about Architectural Data

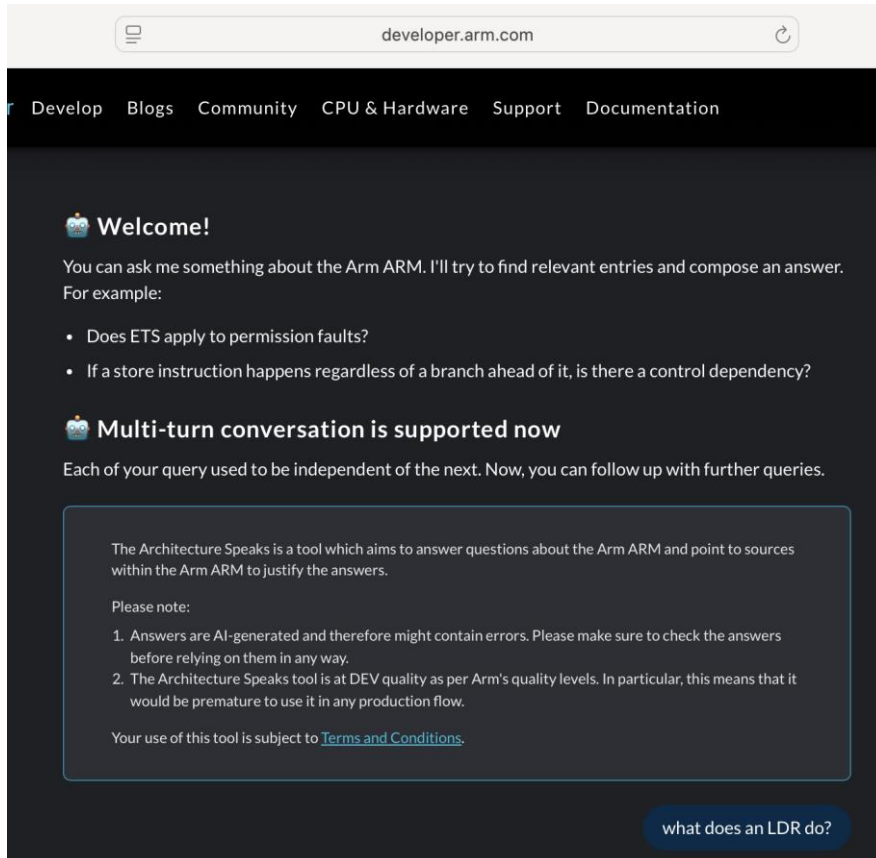
We build tools to examine Architectural Data, to answer questions about the Architecture.

Often this requires manipulating the Architectural Data in a number of ways, for example:

- Preparing Architectural Data to make it suitable for analysis
- Producing Architectural Data when it does not exist
- Giving meaning to Architectural Data
- Interpreting Architectural Data to answer questions
- Confronting two sources of Architectural Data

Up until recently, we only exploited the “prose written by hand” data type through reading and conceptualising

The Architecture Speaks



developer.arm.com

Develop Blogs Community CPU & Hardware Support Documentation

 **Welcome!**

You can ask me something about the Arm ARM. I'll try to find relevant entries and compose an answer. For example:

- Does ETS apply to permission faults?
- If a store instruction happens regardless of a branch ahead of it, is there a control dependency?

 **Multi-turn conversation is supported now**

Each of your query used to be independent of the next. Now, you can follow up with further queries.

The Architecture Speaks is a tool which aims to answer questions about the Arm ARM and point to sources within the Arm ARM to justify the answers.

Please note:

1. Answers are AI-generated and therefore might contain errors. Please make sure to check the answers before relying on them in any way.
2. The Architecture Speaks tool is at DEV quality as per Arm's quality levels. In particular, this means that it would be premature to use it in any production flow.

Your use of this tool is subject to [Terms and Conditions](#).

what does an LDR do?

Summary

- **What LDR does:** LDR performs a memory load — it reads data from memory (address formed according to the instruction form) and writes that data into a register. This is as per [C.3.2.1. Load/store register \[L105\]](#).
- **Note:** Exact size (byte/halfword/word/doubleword), addressing mode, and destination register type depend on the specific LDR encoding (register, immediate, literal, SIMD/SVE variants). This is as per [C.6.2.191. LDR \(immediate\) \[L40\]](#).

Top 5 retrieved sources

- [D.14.3.2. Common microarchitectural events](#)
- [C.7.2.209. LDR \(literal, SIMD&FP\)](#)
- [Glossary](#)
- [C.5.2.4. DIT, Data Independent Timing](#)
- [C.7.2.210. LDR \(register, SIMD&FP\)](#)

Definition

Load register (register)

This instruction calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about addressing modes, see Load/Store addressing modes.

This is verbatim as per [C.6.2.193. LDR \(register\) \[L40\]](#).

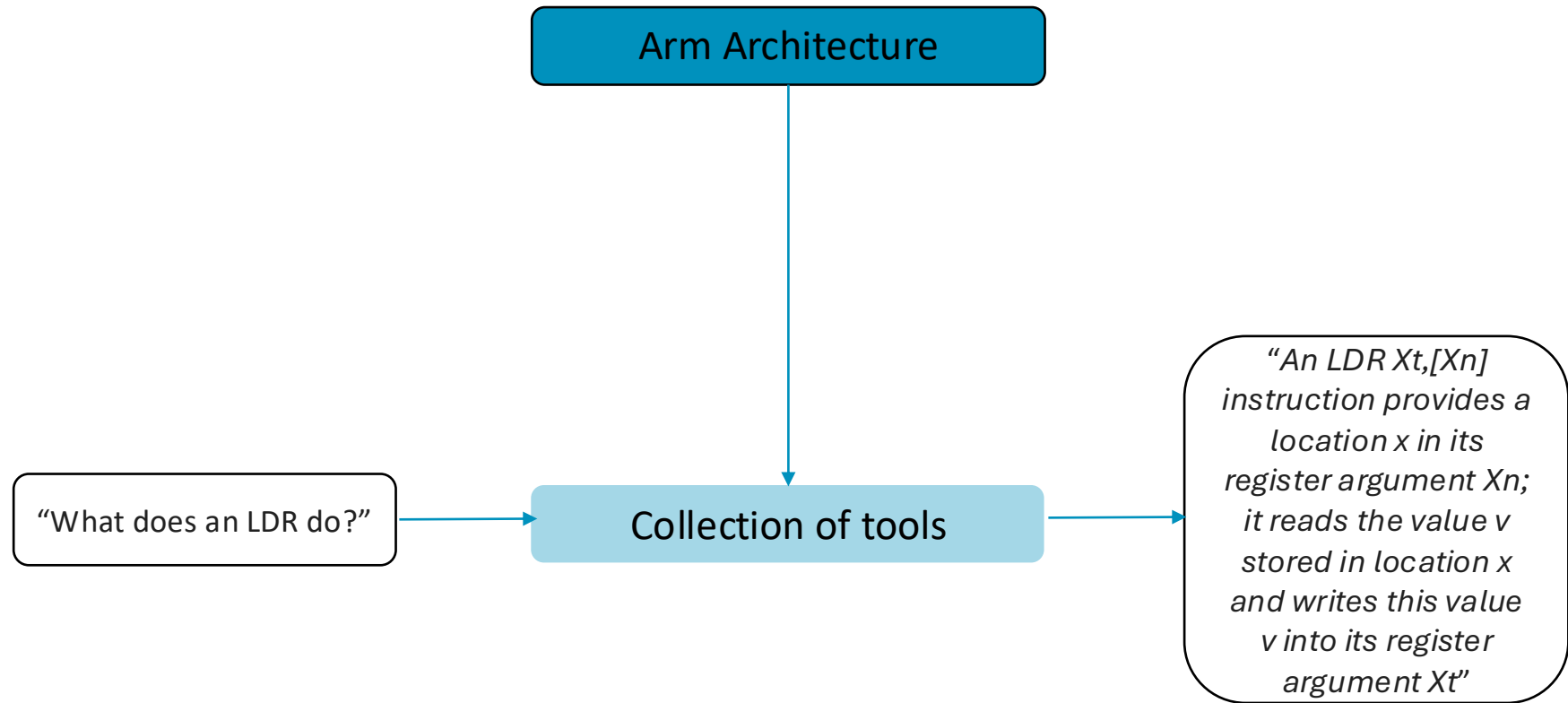
Variants

- **Register-offset:** The instruction calculates an address from a base register and an offset register, loads a word from memory, and writes it to a register. This is as per [C.6.2.193. LDR \(register\) \[L40\]](#).
- **Immediate-offset:** LDR can use a base plus an immediate offset (post-/pre-index or unsigned-offset); it loads a word or doubleword (scaled by the accessed size) and writes it to a register. This is as per [C.6.2.191. LDR \(immediate\) \[L40\]](#).
- **Literal (PC-relative):** LDR can form the address from the PC plus an immediate (a literal) and load from that address into a register. This is as per [C.6.2.192. LDR \(literal\) \[L41\]](#).
- **SIMD/Floating-point and SVE variants:** There are LDR encodings that load SIMD/FP registers or SVE predicate/vector registers (these use similar addressing forms but transfer to SIMD/FPU or SVE registers). This is as per [C.7.2.208. LDR \(immediate, SIMD&FP\) \[L41\]](#).

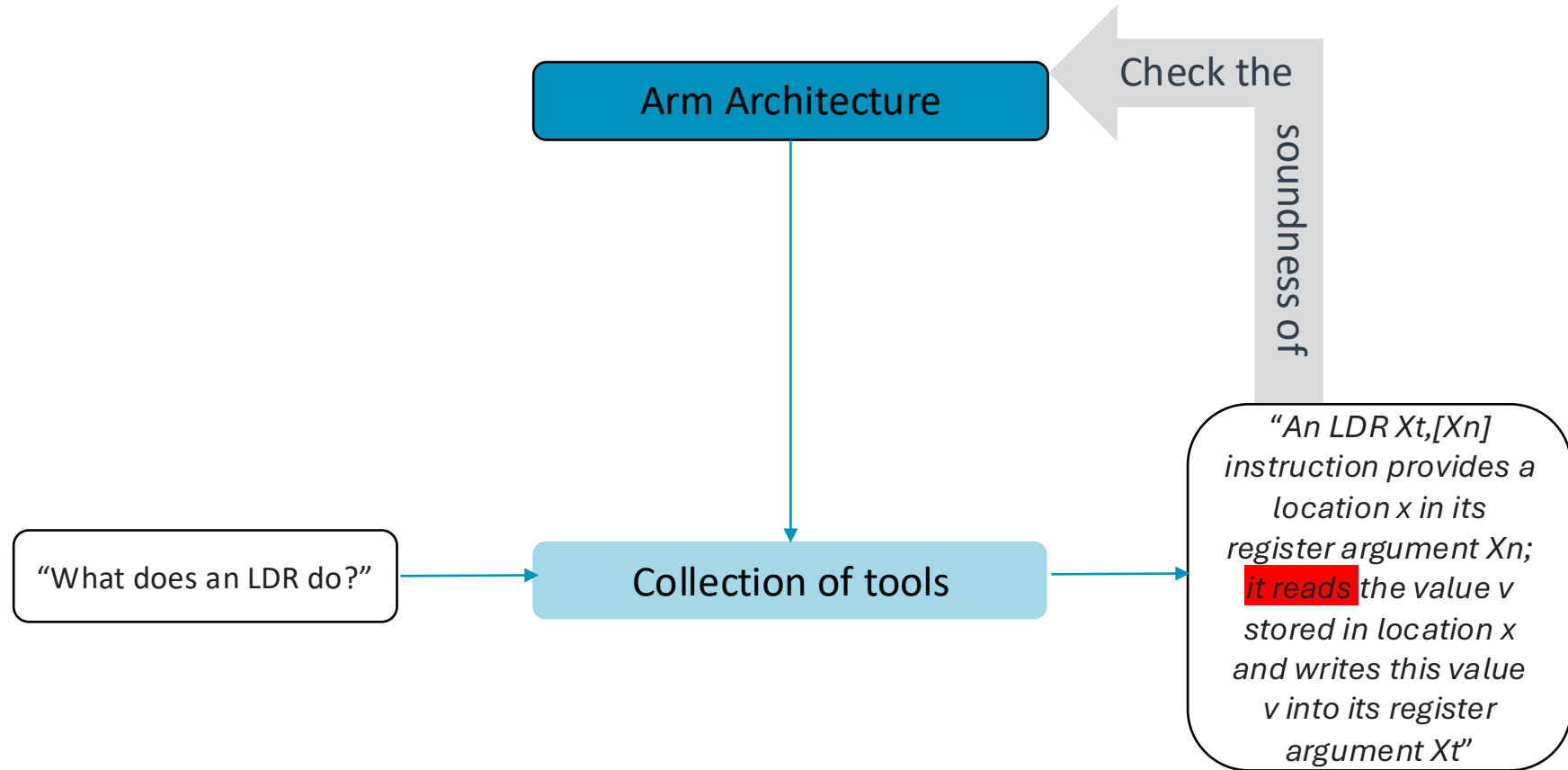
arm

Our team's goal

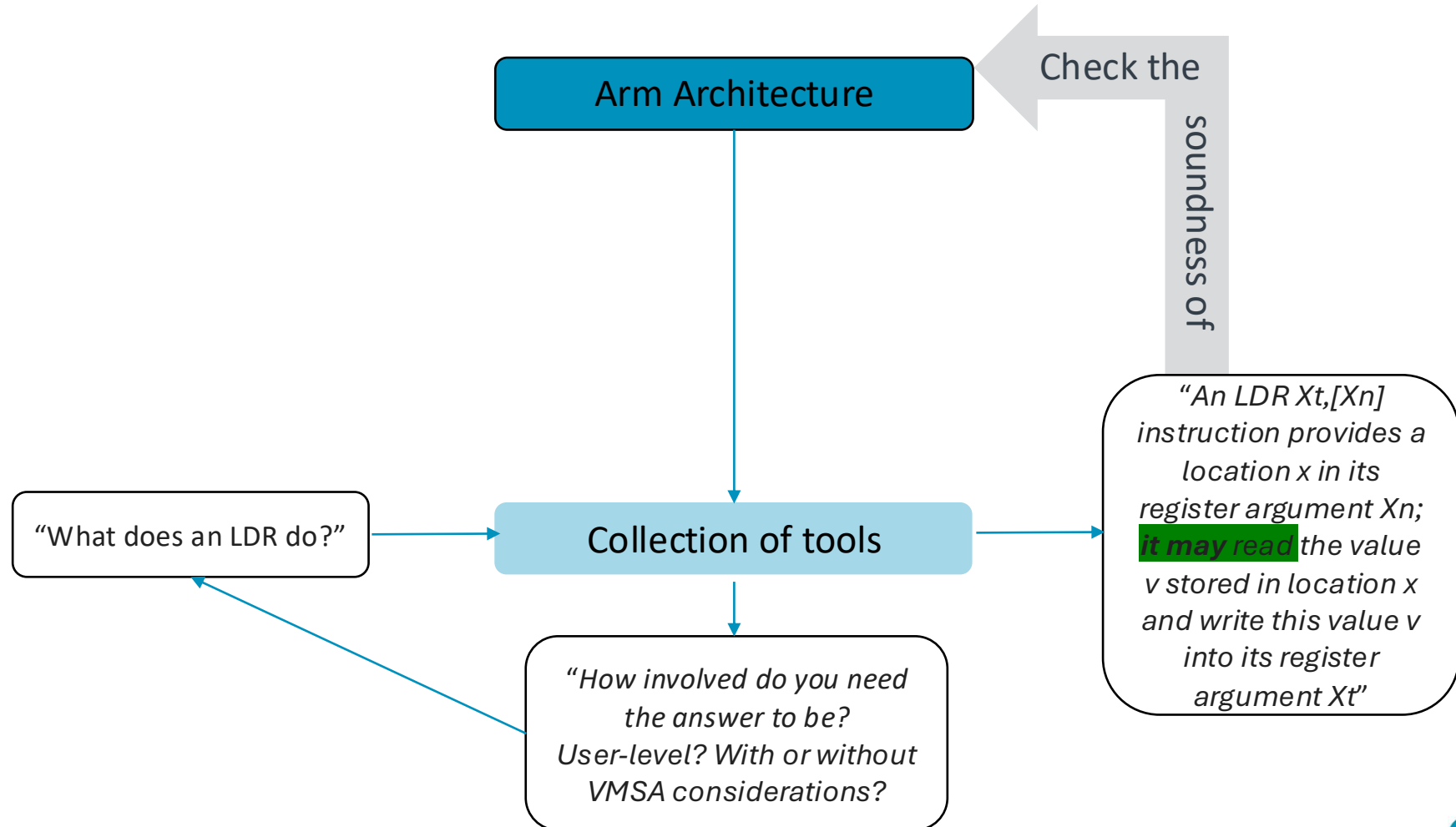
Concise specifications



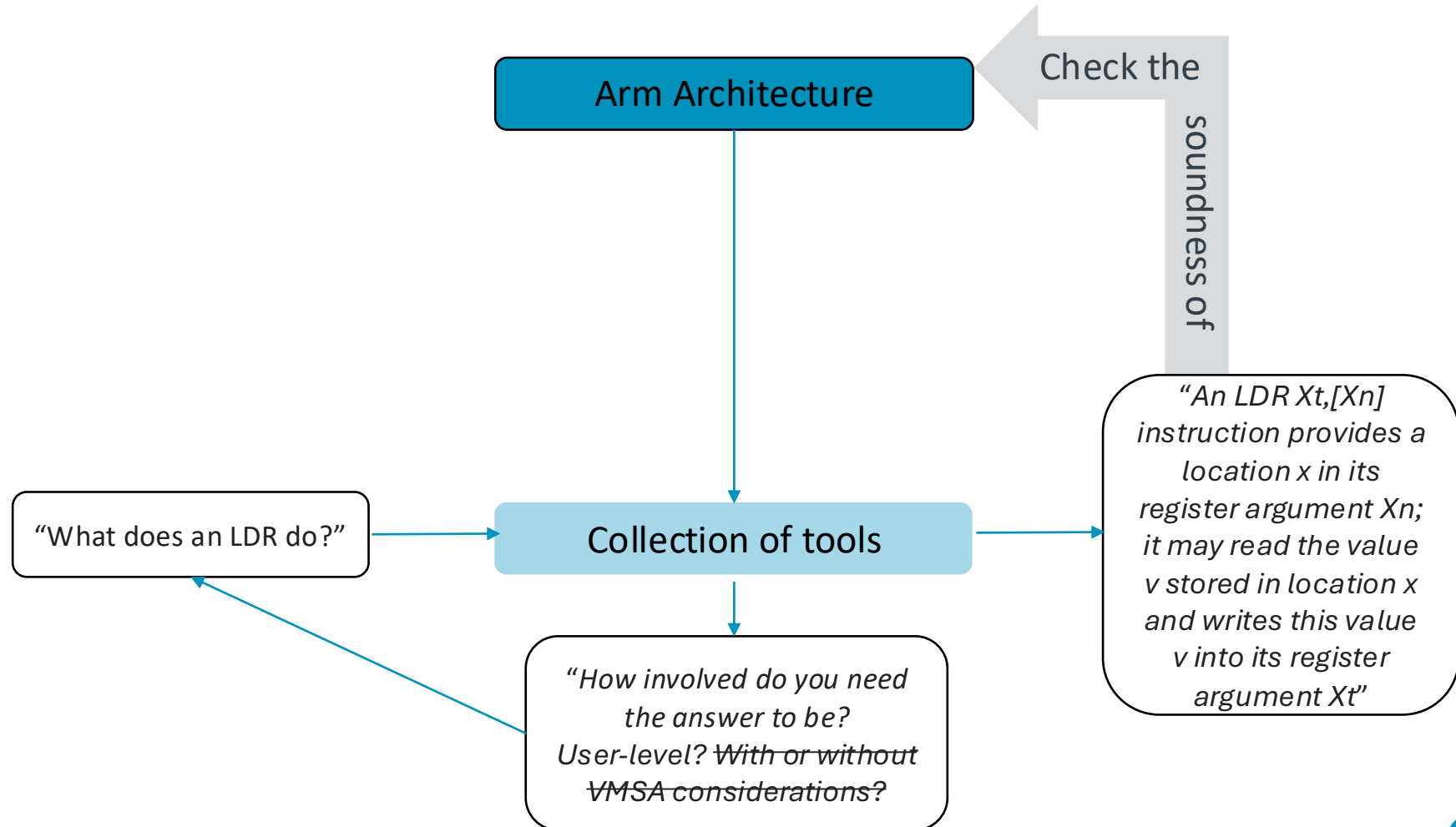
Sound specifications



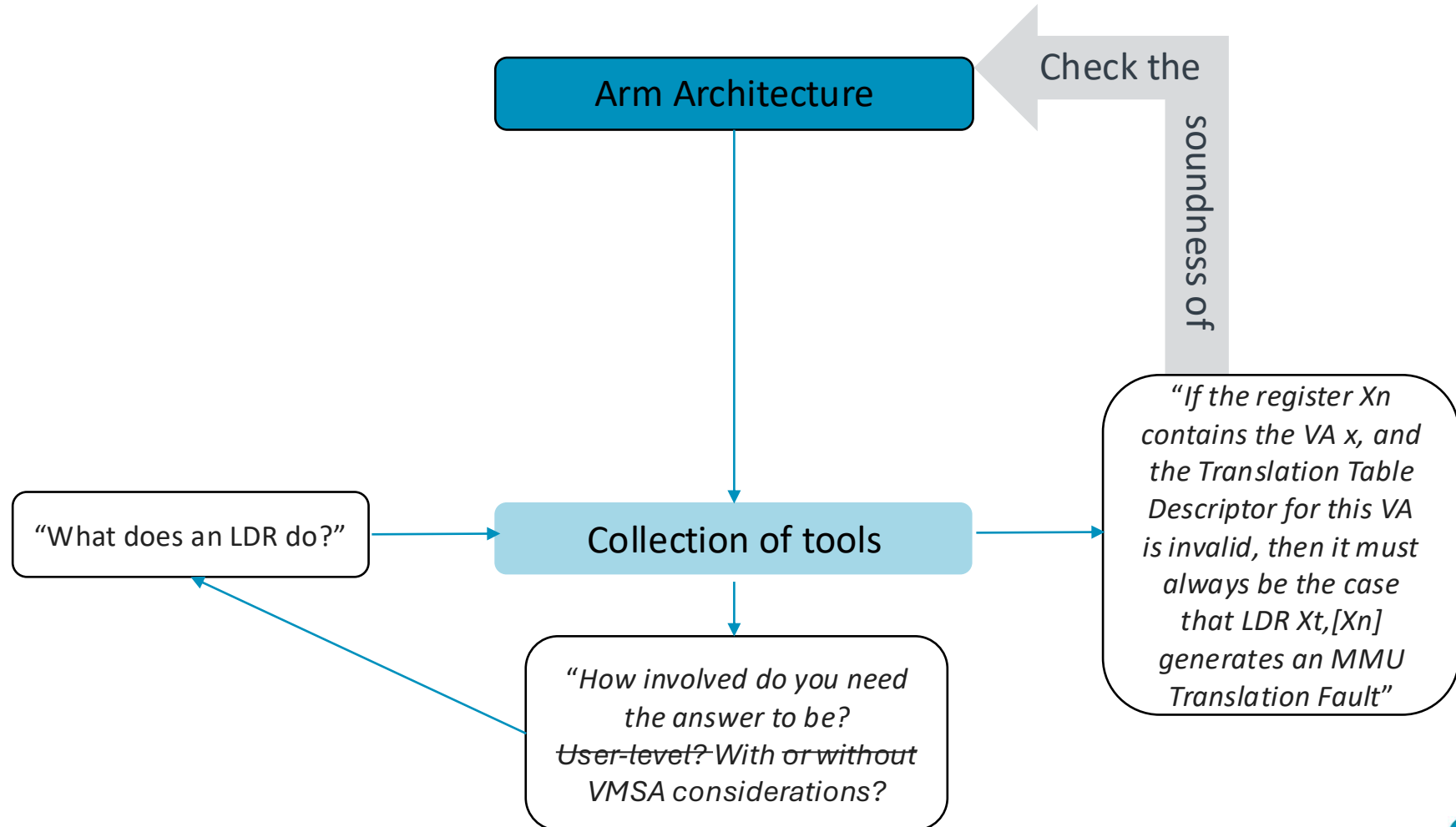
Demonstrably sound specifications



Layerable specifications



Layerable specifications

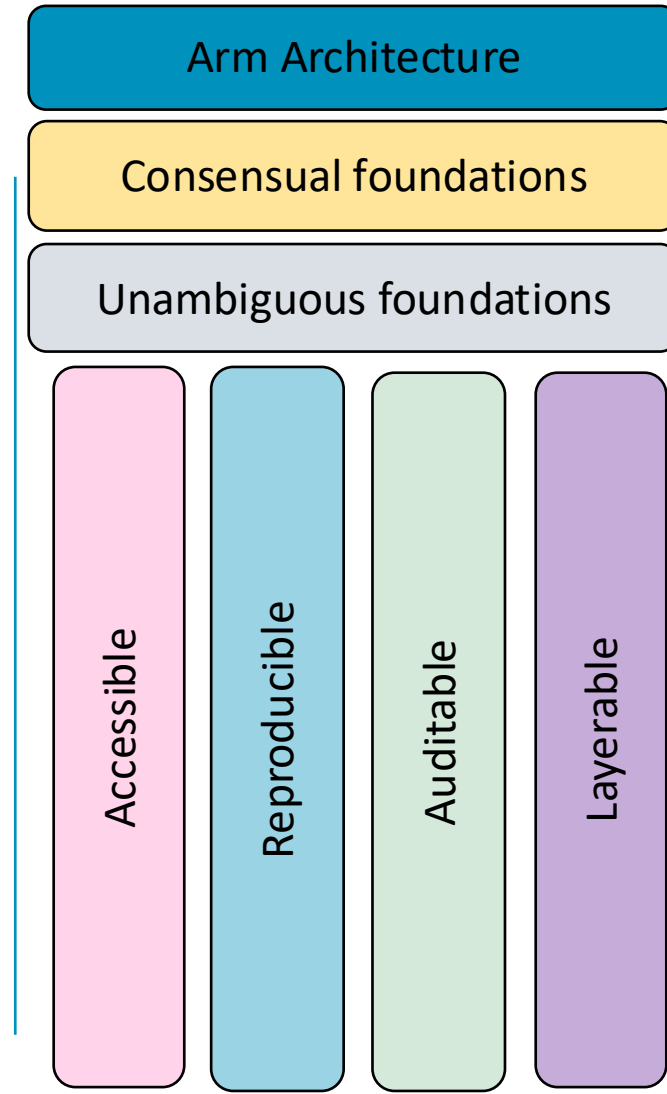


arm

Principles for Architecture Quality

In an ideal world, the Architecture would be

- *Built on consensual foundations*: the same basic concepts need to be understood and agreed by all Architects (e.g. Effect vs access vs instruction vs read/write vs load/store)
- *Built on unambiguous foundations*: all basic concepts need to be defined and used throughout all features
- *Accessible*: the sheer length of the Arm ARM is at odds with that - plus all the supplements and extensions
- *Reproducible*: the same conclusion needs to be reachable by different people, or the same person several months apart
- *Extensible*: we should simply be able to copy-paste/reuse concepts (e.g. “completion in finite time” from DSB to GSB)
- *Auditable*: reviewing the Architecture at design time needs to become easier
- *Maintainable*: reviewing the Architecture a posteriori needs to become easier
- *Layerable*: we need to be able to abstract certain parts away, or select certain parts or levels of abstraction (e.g. “today I only need the user-level, or VMSA+MTE, but not SVE”)



Viewing the Architecture as an artefact

The only way to provide unambiguous foundations is through formal definitions.

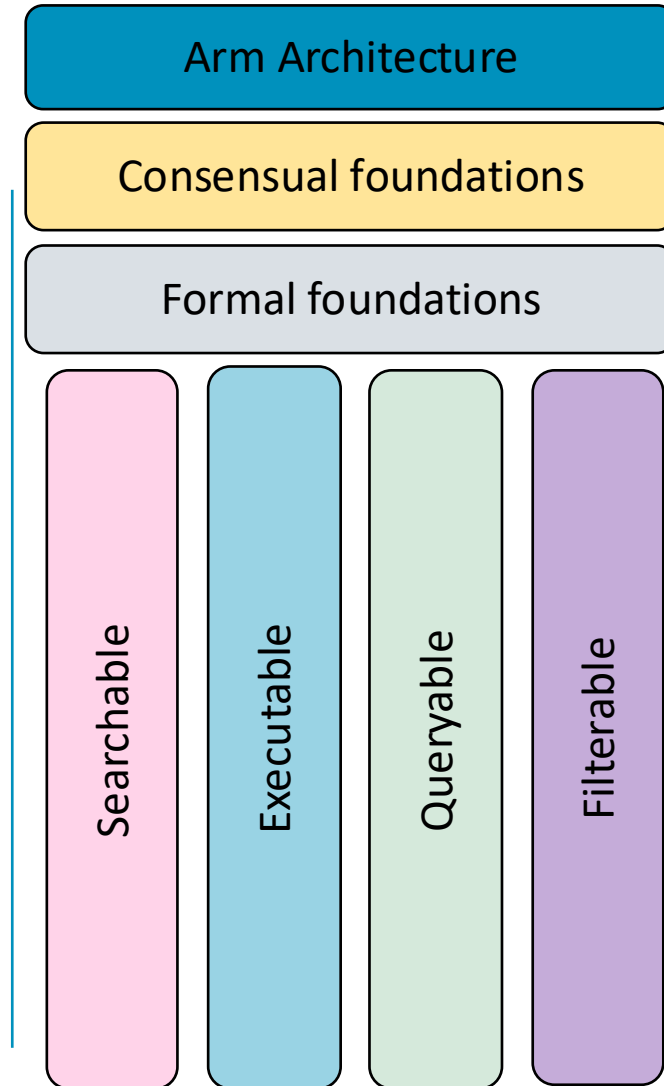
The rest can partly be achieved by viewing the Architecture as an artefact that is:

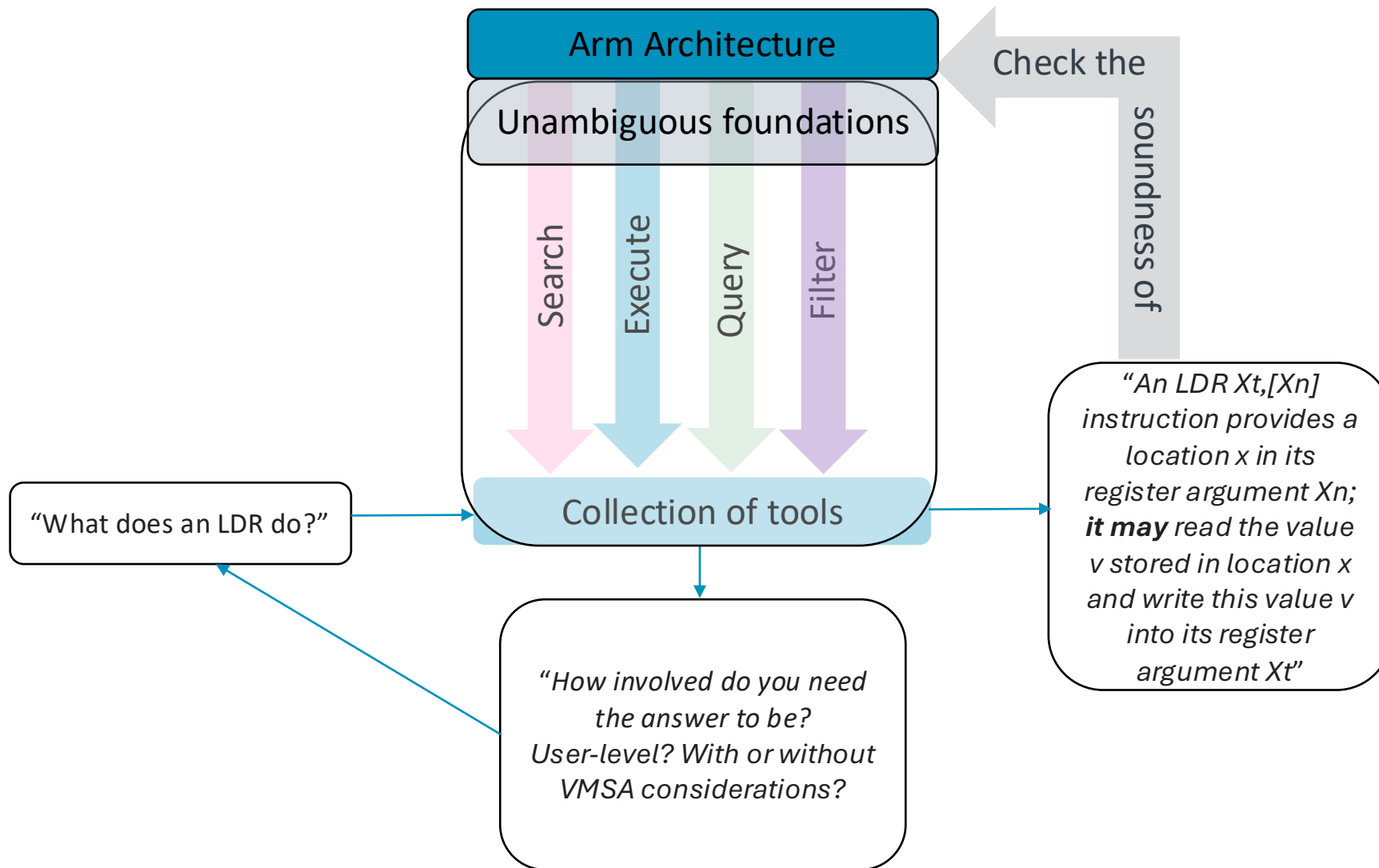
- Searchable (by humans): for accessibility
- Executable: for reproducibility and extensibility
- Queryable (by machines): for auditability and maintainability
- Filterable: for layerability

And an artefact with these traits can help us ensure consensual foundations.

Simon PJ says:

“Computers are remorseless but non-judgmental critics of woolly thinking and faulty logic”







We're hiring!
3 engineer roles
3 PhD studentships

The ARM logo is displayed in a white, lowercase, sans-serif font. The letters are bold and modern, with the 'a' and 'm' having a slightly rounded, friendly appearance. The logo is centered horizontally on the left side of the slide.

arm

The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks

There are even more types of Architectural data

This instruction is a data-independent-time instruction as described in [About PSTATE.DIT](#).

```
// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags
```

A2.2.2 The Armv8.1 architecture extension

- It is Armv8.0 compliant.
- It includes all of the Armv8.1 architectural features that are mandatory.

- Armv8.1 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.2, subject only to those constraints that require that certain features be implemented together.

FEAT_Debugv8p1 is OPTIONAL from Armv8.0.

- ID_AA64DFR0_EL1.DebugVer.
- ID_DFR0_EL1.CopDbg.
- ID_DFR0.CopDbg.
- EDDEVARCH.ARCHVER.

When FEAT_VHE is implemented, FEAT_E2H0 indicates that HCR_EL2.E2H can be programmed to the value 0. The absence of FEAT_E2H0 results in a relaxed behavior where HCR_EL2.E2H is RES1, with EL2 host mode always enabled.

FEAT_E2H0 is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_E2H0:

- ID_AA64MMFR4_EL1.E2H0.

Figure D5-1 Alignment Synchronization Packet

When the Effective value of `HCR_EL2.E2H` is 1, without explicit synchronization, accesses from EL3 using the accessor name `SCTLR_EL1` or `SCTLR_EL12` are not guaranteed to be ordered with respect to accesses using the other accessor name.

Accesses to this register use the following encodings in the System register encoding space:

```
if !IsFeatureImplemented(FEAT_AA64) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR.EL2.TRMV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_HGTR) then
        HGRTR.EL2.SCTLR.EL1 == '1' then
```

In Armv8.0, the event number space is 10 bits. Armv8.1 extends the event number space, and therefore the `PMEVTYPER<n> EL0.evtCount` field, to 16 bits, and is allocated as Table D14-2 shows. For more information about the entries in the Allocation column see the text that follows this table:

See `PMEVTYPER<n>.evtCount` for details of the PE behavior when an event number for a reserved or unimplemented PMU event is written to `evtCount`.

Table D14-3 lists the number, mnemonic, and description of PMU events.

Table D14-3 Event index

Number	Mnemonic	Description
0x0000	SW_INCR	Instruction architecturally executed, Condition code check pass, software increment.
0x0001	L1I_CACHE_REFILL	Level 1 instruction cache refill.
0x0002	L1I_TLB_REFILL	Level 1 instruction TLB refill.

The SCTLR_EL1 characteristics are

Purpose

Provides top-level control of the system, including its memory system, at EL1 and EL0.

AArch64 System register SCTL_R_EL1 bits [31:0] are architecturally mapped to AArch32 SCTL_R[31:0].

This register is present only when FEAT_AA64 is implemented. Otherwise, direct access is UNDEFINED.

SCTLR_EL1 is a 64-bit register.

Figure 1: Schematic representation of the human genome. The top part shows a chromosome with bands numbered 63 to 36. A grey box labeled 'RESO' is located on chromosome 10. Various genes are mapped to specific bands: TIDCP, SPINMASK, ENTF2, TCSO, ENASR, ENASO, ENAS, EPAN, TWEDEN, DSBS, ATA, TCF, TCFO, ITFSB, and EN. The bottom part shows a chromosome with bands numbered 31 to 4. A grey box labeled 'RESO' is located on chromosome 10. Various genes are mapped to specific bands: ENIA, ENIB, LSMAOR, nLMD, ENDA, SPAN, IESB, TSCXT, nTWE, RESO, nTWE, I, ROS, nMEDI, ITFA, and CP15.

When FEAT_TIDCP1 is implemented:

Trap IMPLEMENTATION DEFINED functionality. When the Effective value of `HCR_EL2.{E2H, TGE}` is not {1, 1}, traps EL0 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL1.

TIDCP	Meaning
01-0	Ministrations concerning the Sacraments in Sacramental instruction were not transmitted. This