

# 1 On the role of prose in specifications

2 Jade Alglave 

3 Arm, Cambridge, United Kingdom

4 University College London, United Kingdom

## 5 — Abstract —

---

6 Specifications should let users find answers to their questions. Those answers should be accessible,  
7 unambiguous, consensual, reproducible, auditable, and they should be traceable to the artefacts  
8 users actually read.

9 This paper uses work done by the Arm Architecture Formal Team as a case study in the tensions  
10 between those requirements.

11 **2012 ACM Subject Classification** Theory of computation → Concurrency; Software and its engin-  
12 eering → Formal software verification; Computer systems organization → Architectures

13 **Keywords and phrases** Arm Architecture, (prose, formal, executable, queryable, accessible) specific-  
14 ations, concurrency, herdttools, litmus tests, instruction set, Architecture Specification Language  
15 (ASL), The Architecture Speaks, query interface

16 **Digital Object Identifier** 10.4230/LIPIcs.CONCUR.2026.XX

17 **Acknowledgements** I would like to thank Nikos Nikoleris, Simon Peyton Jones and Daryl Stewart  
18 for taking the time to review various drafts and provide helpful feedback. I would like to give a big  
19 shout-out to the Arm Architecture Formal Team who works tirelessly and with good cheer towards  
20 our ambition. I would like to thank Richard Grisenthwaite for giving the team the time and space  
21 needed to pursue our work and shape our vision.

## 22 **1** Introduction

23 Specifications should let users find answers to their questions. Often, specifications are  
24 written in prose, even technical ones. This may have drawbacks; for example, prose may  
25 be ambiguous, or less accessible to non-native speakers. There might be a temptation to  
26 resolve this by formalising specifications, which at Arm is the original goal of the Architecture  
27 Formal Team. Throughout the paper, I use work done by the team as illustration.

28 For the longest time I saw the prose aspect of our work through a negative lens: often as  
29 a burden, something we needed to formalise away and simply would not see the end of before  
30 we were all very old indeed; sometimes as a “necessary evil” to communicate formalisations,  
31 which invariably would need to be discussed in depth and at length to reach relatively  
32 consensual phrasings with stakeholders and be ratified before entering the Arm Architecture  
33 Reference Manual.

34 Somewhat recently, my stance on prose has changed and I try to expose why, and how the  
35 team’s work has evolved under that renewed steer. For the Arm Architecture Formal Team,  
36 prose cannot be left outside specification engineering: prose is the main communication  
37 medium for our stakeholders.

38 Hence for us prose cannot be replaced by formal models, even executable ones. Instead  
39 prose needs to be engineered: connected to formal artefacts, generated where appropriate,  
40 queried by tools, tested through behavioural questions, and treated as a first-class specification  
41 artefact. Relatedly certain specification aspects cannot necessarily all be derived from a  
42 single (formal) source of truth, and therefore we need to envisage and engineer specifications  
43 as ecosystems of connected artefacts.

44 Throughout the paper I discuss a number of observations which have helped us shape  
45 the team’s philosophy for engineering specifications:



© Jade Alglave;

licensed under Creative Commons License CC-BY 4.0

37th International Conference on Concurrency Theory (CONCUR 2026).

Editors: Ana Sokolova and Patrick Totzke; Article No. XX; pp. XX:1–XX:15

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## XX:2 On the role of prose in specifications

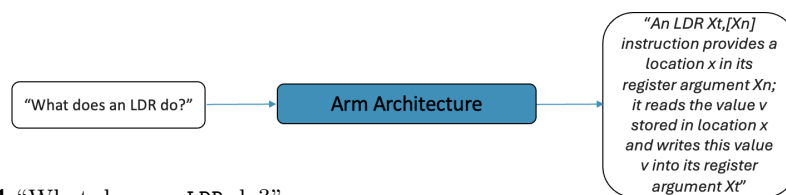
- 46 1. Behavioural questions are a unit of specification work.
- 47 2. Specifications need to provide accessible, unambiguous, consensual, reproducible and
- 48 auditable answers.
- 49 3. Prose is a necessary but arduous specification medium.
- 50 4. Formal and executable specifications can help, but cannot replace prose.
- 51 5. The necessity for consensual, accessible prose influences the formalisation choices.
- 52 6. Creating programmatic bridges from formalisation to prose seems feasible in practice.
- 53 7. Disagreement between specification artefacts is a useful review signal.
- 54 8. Certain specifications may need to be engineered as ecosystems of connected artefacts,
- 55 not necessarily all generated from one single source of truth.
- 56 9. Interfaces to query the prose as it is, not as we wish it to be, make prose specifications
- 57 more directly inspectable.

### 2 Specifications let us answer behavioural questions

59 Arm publishes Architecture specifications (the Architecture, for short), which prescribe,  
60 for a program in Arm assembly, which behaviours are Allowed or Forbidden. This should  
61 be true on all Arm systems: the Architecture describes the envelope of all valid hardware  
62 implementations. Thus the Architecture is a contract between hardware and software.

63 More prosaically, the Architecture allows readers to find answers to questions about the  
64 behaviour of Arm systems (e.g. telephones, tablets, laptops, servers. . .).

65 An example of a question about the behaviour of an Arm system is: “What does an LDR  
66 instruction do?” The Architecture lets a user answer that question, as illustrated in Figure 1.



■ **Figure 1** “What does an LDR do?”

67 Another example of a question about the behaviour of an Arm system is a *litmus test*: a  
68 small concurrent program written in Arm assembly, which asks whether a certain behaviour  
69 is Allowed or Forbidden on Arm hardware. Figure 2 gives two litmus tests, called MP and  
70 MP+DMB.ST+DMB.LD.

```
AArch64 MP                                AArch64 MP+DMB.ST+DMB.LD
{                                           {
0:X1=x; 0:X3=y;                            0:X1=x; 0:X3=y;
1:X3=y; 1:X1=x;                            1:X3=y; 1:X1=x;
}                                           }
P0      | P1      ;                          P0      | P1      ;
MOV W0,#1 | LDR W4,[X3] ;                    MOV W0,#1 | LDR W4,[X3] ;
STR W0,[X1] | LDR W5,[X1] ;                  STR W0,[X1] | DMB LD ;
MOV W2,#1 | ;                                DMB ST | LDR W5,[X1] ;
STR W2,[X3] | ;                              MOV W2,#1 | ;
exists (1:X4=1 /\ 1:X5=0)                  STR W2,[X3] | ;
exists (1:X4=1 /\ 1:X5=0)                  exists (1:X4=1 /\ 1:X5=0)
```

■ **Figure 2** Two examples of questions about the behaviour of an Arm system.

71 The MP test asks what happens if two threads P0 and P1 communicate via shared  
72 memory locations x and y, as follows:

- 73 ■ P0 stores value 1 to  $x$ , and sets the flag  $y$  with another store.  
 74 ■ P1 loads the flag  $y$  and loads  $x$ .

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

76 The MP+DMB.ST+DMB.LD litmus test asks the same question of a program where P0  
 77 and P1 have special instructions, called barriers: does this change the outcome?

78 Figure 3 again illustrates the fact that the Architecture lets the user answer such questions.



■ **Figure 3** “If P1 sees the flag set, can it still see the old value of  $x$ ?”

79 Thus, as trivial as it may sound, a specification lets users find out answers to their  
 80 questions. This leads to the following observation:

81 *Behavioural questions are a unit of specification work.*

82 Ideally, a specification would let users find answers to their questions relatively easily.  
 83 What easily means may depend on the audience: some will prefer to get answers at the  
 84 click of a button, others will prefer an answer in high-level prose that they feel to be more  
 85 accessible, others still will prefer an answer in very precise formal language. At the same  
 86 time, the ease of finding an answer should not be at the cost of that answer being ambiguous,  
 87 up for debate, not reproducible, unjustified or wrong. This leads to the following observation:

88 *Specifications need to provide*  
 89 *accessible, unambiguous, consensual, reproducible and auditable answers.*

90 Of course, there are natural tensions between all these traits.

### 91 **3 Accessibility pulls us towards prose**

92 For accessibility reasons, specifications may be given in prose, as prose may feel a more  
 93 accessible medium than others. But a specification also needs to provide the users with all  
 94 the necessary details exhaustively.

95 As an illustration, in the context of Arm Architecture, the specification is determined by  
 96 the Arm Architecture Reference Manual (Arm ARM for short) [8], a technical document  
 97 made of different types of data, predominantly English prose. The Arm ARM contains the  
 98 necessary information to answer questions about the behaviour of an Arm system, such as  
 99 the LDR semantics question in Figure 1 and the final state of the MP litmus test in Figure 3.

100 Whilst the Arm ARM gives a complete specification, which is of course necessary for  
 101 its suitability as a specification, this affects the length of the document, and therefore its  
 102 accessibility. The latest issue of the Arm ARM is M.b, which counts 17153 pages: it may be  
 103 difficult to find the exact elements of response to a given question.

104 Another example of tensions between desirable traits for specifications is as follows. A  
 105 prose specification, perhaps therefore perceived as accessible, may however be ambiguous,  
 106 and therefore invite debates as to how to interpret the specification. Further to this, a prose  
 107 specification may invite divergences in interpretation, either due to the ambiguity of the  
 108 text, or the difficulty in grasping its content. In turn this means that two different readers,

## XX:4 On the role of prose in specifications

109 or perhaps the same reader several months apart, may reach different answers to a given  
110 question, therefore contradicting the notion that answers should be reproducible.

111 As an illustration, the Arm ARM describes the envelope of legitimate behaviours of  
112 relatively complex systems, viz, Arm multiprocessor chips. This means that the specification  
113 itself may be complex in places, due to the complexity of the systems it specifies. For  
114 example the snapshot of the Arm ARM issue A.k dated 30<sup>th</sup> of September 2016 given in  
115 Figure 4 shows the definition of the barrier instruction DMB at that time. From that, the  
116 reader needed to be able to determine whether the behaviour MP+DMB.ST+DMB.LD was  
117 Forbidden.

### 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 *Full system* 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.

■ **Figure 4** A snapshot of the Arm ARM issue A.k dated 30<sup>th</sup> of September 2016

118 This leads to the following observation:

119 *Prose is a necessary but arduous specification medium.*

120 Prose gives users a way into the material, but for large behavioural questions it can ask  
121 too much of the reader.

122 For the longest time, I used to think of the unconstrained English prose that constitutes  
123 the vast majority of the information in the Arm ARM as a burden, something that we needed  
124 to formalise away to remove ambiguity and simply would not see the end of before we were  
125 all very old indeed. I also used to think, rather naively, that once we had written formal  
126 models of the Architecture our job was done.

127 But the contact with my colleagues, engineers and partners, made me rethink that very  
128 deeply. Evidently (although it took me years to realise this), we cannot just write a formal  
129 model and throw it over the fence, hoping for customers to “just use it”, even when we  
130 provide tools to execute it.

131 This leads to the following observation:

132 *Formal and executable specifications can help, but cannot replace prose.*

## 133 **4 Unambiguity and reproducibility pull us towards formalisation and** 134 **executability**

135 A natural way to provide unambiguous foundations is through formal definitions. For the  
136 rest, I once heard Simon Peyton Jones say:

137 *Computers are remorseless but non-judgmental critics*  
138 *of woolly thinking and faulty logic.*

139 Taken in tandem, these insights point to building formal and executable specifications:  
140 ■ formal, so that each concept in the specification is precise, well defined;  
141 ■ executable, so that users can interact with the specification to check their understanding,  
142 which in turn can help support consensual foundations and reproducibility of answers.

143 As an illustration, the Arm Architecture Formal Team develops a number of formal and  
144 executable artefacts, which together aim to give a formal and executable specification of  
145 the Architecture. At a high level, the Architecture combines instruction semantics for the  
146 entirety of the Instruction Set, meaning the Effects that instructions may have on registers  
147 and memory, and the Concurrency Model, meaning how those Effects are ordered with  
148 respect to one another.

### 149 **4.1 Concurrency Model**

150 Historically, the concurrency aspects of the Architecture were described in informal prose, as  
151 shown in Figure 4. Nowadays, Arm uses a domain-specific language called cat to describe  
152 concurrent behaviours. The cat language is implemented in the herd tool [5, 6], which lets  
153 users ask behavioural questions phrased as litmus tests, and obtain reproducible answers.

154 The Arm Architecture Formal Team develops, maintains and enhances the cat file and  
155 the associated herd, diy [3] and litmus [4] tools and regularly upstreams their contributions  
156 to GitHub [6]. The website [developer.arm.com/herd7](https://developer.arm.com/herd7) hosts an interface to the herd tool.

157 At a high-level, the Arm Concurrency Model gives ordering requirements over memory  
158 accesses, stating which local orderings must be respected by hardware. Those constraints  
159 forbid certain shapes in program executions. The External visibility requirement forbids cycles  
160 in the Ordered-before relation, which consists of observations between threads (Observed-by  
161 relation) and local orderings (Hardware-Required-Ordered-Before relation) [8, Chapter B2.3].

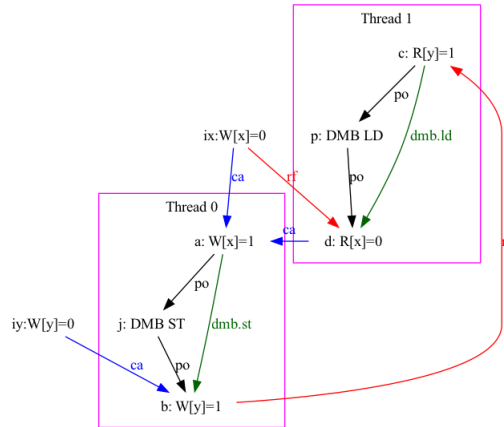
162 For example, let's examine the Forbidden execution of MP+DMB.ST+DMB.LD given in  
163 Figure 5. Each pink box gives the semantics of one of the two threads P0 and P1.

164 On each thread, the semantics of each instruction is given in terms of Effects; for example:

- 165 ■ The store instruction STR W0,[X1] on P0 corresponds to an Explicit Memory Write  
166 Effect a:W[x]=1.
- 167 ■ The STR W2,[X3] corresponds to b:W[y]=1.
- 168 ■ The load instruction LDR W4,[X3] on P1 corresponds to an Explicit Memory Read Effect  
169 c:R[y]=1.
- 170 ■ The LDR W5,[X1] corresponds to d:R[x]=0.
- 171 ■ The DMB ST barrier instruction on P0 corresponds to a Barrier Effect j:DMB.ST.

172 Effects are connected by relations of various kinds, as shown in Figure 5.

173 The main point here is not the particular names of the relations. The Concurrency Model  
174 computes relations over the candidate execution and rejects it because these relations create



■ **Figure 5** A Forbidden execution of MP+DMB.ST+DMB.LD

175 an Ordered-before cycle. Hence a behavioural question, phrased as a litmus test, is answered  
 176 unambiguously and in a reproducible manner.

177 **4.2 Instruction Semantics**

178 When we looked at the Forbidden execution of MP+DMB.ST+DMB.LD, we mapped a store  
 179 instruction STR to a Memory Write Effect, a load instruction LDR to a Memory Read Effect,  
 180 and a barrier instruction to a Barrier Effect. We implicitly or unconsciously assumed that  
 181 x and y were physical memory locations. But usually one interacts with virtual memory  
 182 addresses rather than physical memory directly.

183 The mapping from virtual addresses to physical locations resides in a collection of *page*  
 184 *tables*, or *translation tables* in Arm jargon. In other words, translation tables are used to  
 185 translate the virtual addresses seen by software into physical locations used by hardware.

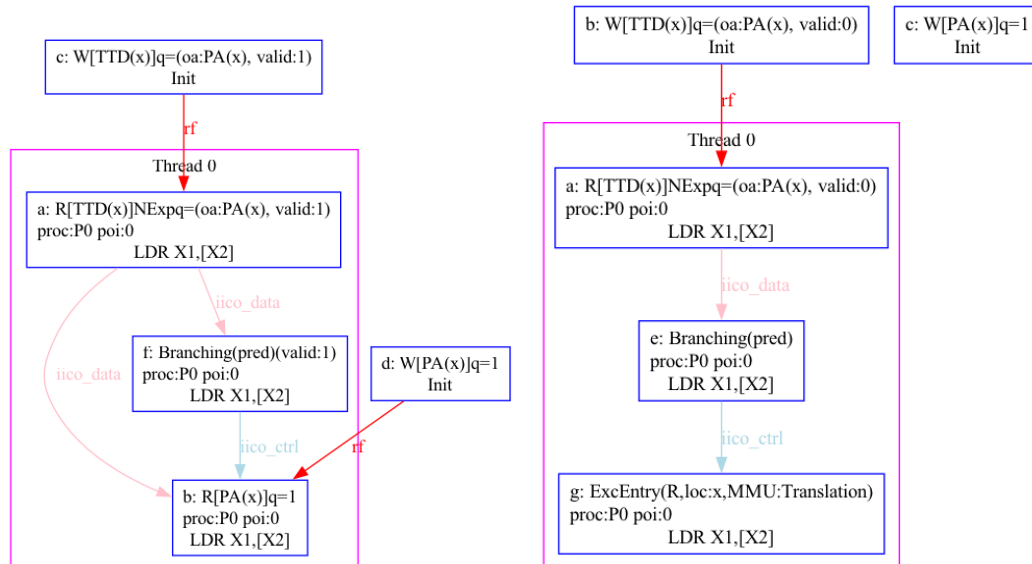
186 Thus the semantics of instructions need to adjust to this level of abstraction: for example,  
 187 a load instruction LDR does not simply map to an Explicit Memory Read Effect like it  
 188 did at user-level abstraction. Instead, the semantics of LDR at the virtual memory level of  
 189 abstraction needs to account for accesses made to translation tables, which determine which  
 190 physical memory location the LDR can read from, if any.

191 Consider e.g. the left-hand-side of Figure 6: the semantics of the LDR instruction is  
 192 contained within the pink box. There the load starts off with an Implicit Memory Effect  
 193  $a:R[TTD(x)]=(oa:PA(x))$ , which reads the translation table entry for the virtual address x,  
 194 and gets a physical address PA(x). This lets us know where in memory the load should be  
 195 taking its value from, which is represented by the Explicit Memory Effect  $b:R[PA(x)]=1$ .

- 196 Each translation table descriptor (TTD) indicates whether the entry is valid or not:
- 197 ■ If it is valid, reading the TTD gives the physical location corresponding to a virtual  
 198 address as is the case on the left-hand-side of Figure 6.
  - 199 ■ If it is not valid, accessing a virtual address via that TTD raises an Exception: the  
 200 right-hand-side of Figure 6 shows the case where the TTD of the virtual address x was  
 201 invalid, leading to Exception Entry Effect g.

202 Historically, we write these instruction semantics by hand into the source of herd. We  
 203 invent these semantics through educated guesses after reading the Arm ARM, reviewing our  
 204 candidate semantics with colleagues and partners, and testing them against hardware.

205 But the Arm ARM also describes each instruction in the Arm Instruction Set in terms of  
 206 a piece of code in a language called Architecture Specification Language (ASL for short).  
 207 Figure 7 shows an excerpt of ASL code for the LDR instruction [11].



■ **Figure 6** Aspects of the virtual memory semantics of LDR

```

Operation
var address : bits(64);
let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
tagchecked, t);
if n == 31 then
  CheckSPAlignment();
  address = SP(64)();
else
  address = X(64)(n);
end;
if !postindex then
  address = AddressAdd(address, offset, accdesc);
end;
let data : bits(datasize) = Mem(datasize)(address, accdesc);
X(resize)(t) = ZeroExtend(resize)(data);
if wback then
  if wb_unknown then
    address = ARBITRARY : bits(64);
  elseif postindex then
    address = AddressAdd(address, offset, accdesc);
  end;
  if n == 31 then
    SP(64)() = address;
  else
    X(64)(n) = address;
  end;
end;
    
```

■ **Figure 7** Excerpt of ASL code for the LDR instruction

208 Historically, ASL did not have any defined syntax nor semantics, although implementations  
 209 existed, e.g. [15, 16]. Arm decided to invest in a formal definition of that language, and make  
 210 some adjustments to the syntax along the way. The resulting language is called ASL1, whilst  
 211 the old pseudocode language is referred to as ASL0. The Arm ARM has now entirely moved  
 212 to be in ASL1, since issue M.b. The ASL Reference documentation [9], tooling (type-checker  
 213 and interpreter) [12] and standard library proofs [10] are developed, maintained and enhanced

## XX:8 On the role of prose in specifications

214 by the Architecture Formal Team. Arm also provides, as for herd, a web interface to the  
215 ASLRef tooling [14].

216 Work is well under way to be able to link, from a formal and executable point of view, the  
217 instruction semantics given by the ASL code of instructions, and the handwritten semantics  
218 present in herd [17]. This should allow much more scalability in handling instructions in  
219 herd, and expand the scope of what the formal Arm Concurrency Model covers.

220 Naturally our work on ASL is adjacent to the work on Sail [18]. Arm chose not to use  
221 Sail for a number of reasons:

- 222 ■ the syntax of Sail was considered to be unintuitive by Architects;
- 223 ■ we could not seem to find a formal semantics of Sail.

### 224 **5 Consensual prose shapes formalisation**

225 The readership of the Arm ARM is very large—potentially the entire Arm ecosystem—and  
226 not necessarily equipped with formal knowledge.

227 Thus a rather large amount of the Architecture Formal Team’s time is spent translating  
228 our formal definitions and findings into hopefully relatively accessible insights. We do this  
229 socially with our stakeholders, by composing emails, hosting meetings and writing slides.

230 A relatively simple illustration of the consequences of our social situation—the necessity  
231 to communicate our artefacts in ways that are intelligible and accessible to our audience—is  
232 as follows. The Arm Concurrency Model defines a relation called Ordered-before, `ob` in `cat`,  
233 which is the transitive closure of the union of two other relations as discussed above, viz,  
234 Hardware-Required-Ordered-Before (`hw-reqs` in `cat`) and Observed-by (`obs` in `cat`). Hence  
235 the `cat` definition of Ordered-before could (but is not) phrased very succinctly thus:

```
236 let ob = (hw-reqs | obs)+
```

237 The transliteration of this `cat` clause, in other words the English prose that would appear  
238 in the Arm ARM and that most readers would refer to instead of the `cat`, would necessarily  
239 have to refer to the notion of the transitive closure. This is because we aim to have the  
240 `cat` and the transliterated English be as close to each other as possible, in particular for  
241 debuggability reasons.

242 But the notion of transitive closure may not be entirely natural or known by our audience,  
243 and therefore we needed to find a different `cat` phrasing for Ordered-before, a phrasing whose  
244 transliteration would be more readily accessible to our audience, whilst remaining semantically  
245 equivalent to the transitive closure. As a consequence, the definition of Ordered-before in `cat`  
246 is thus:

```
247 let rec ob =  
248     hw-reqs  
249     | obs  
250     | ob; ob
```

251 This is then transliterated to the following [8, Section B2.3.9], where the transitive closure is  
252 phrased explicitly:

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

- 254 ■  *$E1$  is Hardware-required-ordered-before  $E2$ .*
- 255 ■  *$E1$  is Observed-by  $E2$ .*
- 256 ■ *All of the following apply:*
  - 257 ■  *$E1$  is Ordered-before  $E3$ .*

258       ■ *E3 is Ordered-before E2.*

259       This leads to the following observation:

260       *The necessity for consensual, accessible prose influences the formalisation choices.*

261       **6   Engineering specifications as ecosystems of connected artefacts**

262       Developing these formal definitions is a slow and partially social process. For example, for  
263       the Concurrency Model:

- 264       ■ We read the prose in the Arm ARM.
- 265       ■ We test machines.
- 266       ■ We interview our colleagues (Architects, implementors, software developers): we show  
267       them hundreds to thousands of tests and ask them if they think the behaviours should  
268       be Allowed or Forbidden.
- 269       ■ We extrapolate a set of definitions, which we regress against deployed hardware;
- 270       ■ We ratify the definitions with Arm partners.
- 271       ■ Then the transliteration of these definitions lands in the Arm ARM.

272       Indeed, the Architecture is determined by what is given in the Arm ARM. In other words,  
273       if the cat file does not appear somehow in the Arm ARM, it is not an authoritative reference  
274       point. Thus there is a need to transliterate the cat file into the Arm ARM.

275       By convention, we aim for 1:1 correspondence between cat and English transliteration, to  
276       help us “debug” the prose. Indeed we can execute and test the cat file, and the immediacy of  
277       the correspondence between the cat file and its English transliteration helps convey some of  
278       that debuggability to the prose. Figure 8 shows a fragment of the cat file and its corresponding  
279       English transliteration in the latest issue M.b of the Arm ARM.

**Ordered-before**

An Effect  $E_1$  is *Ordered-before* an Effect  $E_2$  if one of the following applies:

- $E_1$  is Hardware-required-ordered-before  $E_2$ .
- $E_1$  is Observed-by  $E_2$ .
- All of the following apply:
  - $E_1$  is Ordered-before  $E_3$ .
  - $E_3$  is Ordered-before  $E_2$ .

```

(** Ordered-before **)
let rec ob =
  hw-reqs
  | obs
  | ob; ob

(** External visibility requirement **)
irreflexive ob as external
    
```

**External visibility requirement**

The external visibility requirement is defined as follows: an Architecturally Allowed Execution must not exhibit a cycle in the Ordered-before relation.

■ **Figure 8** Ordered-before fragment of the latest issue M.b of the Arm ARM and the corresponding fragment of the cat file

280       Since this formalisation effort started, we added quite a few features, such as virtual  
281       memory and address translation [2].

282       With all the added features and extensions, the cat file and the corresponding definitions  
283       have grown tremendously. Figure 9 shows the definition of Hardware-Required-Ordered-  
284       Before in the latest issue M.b of the Arm ARM.

285       Evidently, the more the cat file grows, the more the prose grows, and the transliteration  
286       becomes a rather arduous exercise. Since the Arm ARM is the reference definition for the

## XX:10 On the role of prose in specifications

287 whole Arm ecosystem, what seems like a modest typo at our end can have consequences on  
288 hardware designs, or software developments.

---

### Locally-ordered-before

An Effect  $E_1$  is *Locally-ordered-before* an Effect  $E_2$  if one of the following applies:

- $E_1$  is Tag-check-intrinsically-before  $E_2$ .
- $E_1$  is Translation-intrinsically-before  $E_2$ .
- $E_1$  is Fetch-intrinsically-before  $E_2$ .
- $E_1$  is ETS-ordered-before  $E_2$ .
- $E_1$  is Fetch-ordered-before  $E_2$ .
- $E_1$  is Same-Cache-Line-ordered-before  $E_2$ .
- $E_1$  is DSB-ordered-before  $E_2$ .
- $E_1$  is Instruction-fetch-barrier-ordered-before  $E_2$ .
- $E_2$  is a Local memory write successor of  $E_1$ .
- All of the following apply:
  - $E_3$  is a Local memory write successor of  $E_1$ .
  - $E_3$  belongs to the same single-copy-atomic grouping as  $E_2$ .
- $E_1$  is Dependency-ordered-before  $E_2$ .
- $E_1$  is Pick-ordered-before  $E_2$ .
- $E_1$  is Atomic-ordered-before  $E_2$ .
- $E_1$  is Barrier-ordered-before  $E_2$ .
- All of the following apply:
  - $E_1$  is Locally-ordered-before  $E_3$ .
  - $E_3$  is Locally-ordered-before  $E_2$ .

### Pick-locally-ordered-before

An Effect  $E_1$  is *Pick-locally-ordered-before* an Effect  $E_2$  if all of the following apply:

- There is a Pick dependency from  $E_1$  to  $E_3$ .
- $E_3$  is Locally-ordered-before  $E_2$ .
- $E_2$  is an Explicit Memory Write Effect.

### Locally-hardware-required-ordered-before

An Effect  $E_1$  is *Locally-hardware-required-ordered-before* an Effect  $E_2$  if one of the following applies:

- $E_1$  is Locally-ordered-before  $E_2$ .
- $E_1$  is Pick-locally-ordered-before  $E_2$ .
- All of the following apply:
  - $E_1$  is Locally-hardware-required-ordered-before  $E_3$ .
  - $E_3$  is Locally-hardware-required-ordered-before  $E_2$ .

### Hardware-required-ordered-before

An Effect  $E_1$  is *Hardware-required-ordered-before* an Effect  $E_2$  if one of the following applies:

- $E_1$  is Locally-hardware-required-ordered-before  $E_2$ .
- $E_1$  is Hazard-ordered-before  $E_2$ .

■ **Figure 9** Definition of Hardware-Required-Ordered-Before in issue M.b of the Arm ARM

289 So we invented a tool called miaou, which takes the cat file and produces the constrained  
290 English prose of the Arm ARM programmatically. This tool is distributed on GitHub [6],  
291 and we have used it since the Arm ARM issue K.a (dated 20 March 2024).

292 This leads to the following observation:

293 *Creating programmatic bridges from formalisation to prose seems feasible in practice.*

294 The insight here is not to build a prose generator, but rather this idea of engineering a  
295 bridge, to keep a formal and executable artefact and authoritative prose close enough that

296 the prose can be debugged by proxy.

297 This idea of generating prose from the formal and executable artefact is reminiscent of  
 298 the approach taken by the WebAssembly ecosystem with SpecTec [19]: having one single  
 299 source of truth out of which other representations are derived.

300 Whilst this is a tempting approach, in the context of our work we need to depart from  
 301 this ambition a little. Indeed we are not, and cannot be, the only authors of the specification:  
 302 different parts of the Arm ARM are authored by separate teams with different methodologies,  
 303 tools or workflows. The vast majority of the material that appears in the Arm ARM is  
 304 relatively unconstrained prose written by Arm Architects to describe the rules of their  
 305 respective areas of the Arm Architecture. Figure 10 gives an illustration of prose written by  
 306 Arm Architects, an excerpt of the chapter on Address Translation.

307 The lesson is not that single-source approaches are undesirable. They are attractive when  
 308 one authoritative authoring process can produce all the representations users need. In our  
 309 setting, however, different communities author different parts of the Architecture, different  
 310 users need different representations, and the authoritative prose cannot simply be replaced  
 311 by a formal artefact.

#### D8.1 Address translation

R <sub>DWFWP</sub>	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 <sub>RRHIV</sub>	All of the following determine whether a translation stage uses the VMSAv9-128 translation system: <ul style="list-style-type: none"> <li>• For stage 1 translations in the EL1&amp;0 translation regime, the <i>Effective value</i> of TCR2_EL1.D128 is 1.</li> <li>• For stage 1 translations in the EL2&amp;0 translation regime, the <i>Effective value</i> of TCR2_EL2.D128 is 1.</li> <li>• For stage 1 translations in the EL3 translation regime, the <i>Effective value</i> of TCR_EL3.D128 is 1.</li> <li>• For stage 2 translations, the <i>Effective value</i> of VTCR_EL2.D128 is 1.</li> </ul> Otherwise, the translation stage uses the VMSAv8-64 translation system. This applies even when the translation stage is disabled.
I <sub>CTQS</sub>	Address translation converts the addresses used by instructions to the addresses used by the physical memory system.
R <sub>CRPHV</sub>	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: <ul style="list-style-type: none"> <li>• <i>Program counter</i> (PC).</li> <li>• <i>Stack pointers</i> (SP).</li> <li>• <i>Link register</i> (LR).</li> <li>• <i>Exception link register</i> (ELR).</li> </ul>
R <sub>CRDGS</sub>	When an access is made to the physical memory system, a <i>physical address</i> (PA) is used.
R <sub>KCNRX</sub>	An address translation maps a VA to a PA.
R <sub>CCCQQ</sub>	An address translation requires one of the following: <ul style="list-style-type: none"> <li>• A single translation stage, stage 1.</li> <li>• Two sequential translation stages, stage 1 and stage 2.</li> </ul>
R <sub>ZKJWW</sub>	An address translation stage maps an <i>input address</i> (IA) to an <i>output address</i> (OA).
R <sub>BCKLH</sub>	If one address translation stage is used, then a VA is mapped to a PA using all of the following steps: <ol style="list-style-type: none"> <li>1. The VA is input as the IA to the translation stage.</li> <li>2. The PA is output as the OA from the translation stage.</li> </ol>

■ **Figure 10** English prose written by Arm Architects from Arm ARM M.b

312 Thus the Arm ARM features a number of different types of Architectural Data, including:

- 313 1. The ASL code for each instruction—see e.g. Figure 7.
- 314 2. The constrained English prose programmatically generated from the cat file using the miaou tool—see e.g. Figure 9.
- 315 3. The less constrained English prose written by Arm Architects to describe the rules of  
 316 their respective areas of the Arm Architecture—see e.g. Figure 10.  
 317

318 Specification engineering often requires manipulating the Architectural Data in a number  
 319 of ways, for example:

- 320 ■ Producing Architectural Data when it does not exist (e.g. the Concurrency Model).
- 321 ■ Giving meaning to Architectural Data (e.g. the ASL1 semantics).

## XX:12 On the role of prose in specifications

- 322 ■ Interpreting Architectural Data to answer questions.
- 323 ■ Preparing Architectural Data to make it suitable for analysis.
- 324 ■ Confronting two sources of Architectural Data.

325 This is also why multiple artefacts are useful. Hardware experiments can confront a  
326 model with observed behaviour. A model can confront prose with its formal consequences.  
327 Generated prose can confront a formal definition with the demands of readability. A query  
328 interface can confront the manual with the questions users actually ask. Expert review can  
329 confront all of these with architectural intent.

330 Disagreement is one of the ways the socio-technical system that develops, maintains and  
331 uses the Architecture learns and improves. Of course, unresolved disagreement in a published  
332 specification is a problem. But during development, tension between artefacts is often the  
333 signal that something needs to be understood more clearly.

334 This leads to the following observation:

335 *Disagreement between specification artefacts is a useful review signal.*

336 Instruction semantics is a concrete example of this tension. Aspects of an instruction such  
337 as LDR appear as Effects consumed by the Concurrency Model (see Figure 6), as ASL code  
338 (see Figure 7) and as prose (e.g. Figure 10). These representations serve different audiences  
339 and different tools.

340 Hence specifications can quickly become ecosystems of connected representations. For us,  
341 this points to trying to engineer specifications as a collection of connected artefacts, rather  
342 than one single artefact trying to meet all these somewhat contradicting traits.

343 This leads to the following observation:

344 *Certain specifications may need to be engineered as ecosystems of connected artefacts,  
345 not necessarily all generated from one single source of truth.*

### 346 **7 Working to make prose specifications more directly inspectable**

347 Up until recently, we only exploited the less constrained English prose written by Architects  
348 through reading and conceptualising.

349 More recently, LLMs took the world by storm, and allowed us to interrogate large bodies  
350 of prose, so we started experimenting with them:

- 351 ■ could they help us examine the unconstrained prose more immediately, not by proxy as  
352 we do for the Concurrency Model?
- 353 ■ could they help us make our formal artefacts more accessible, acting as informal interfaces  
354 to the formal tools?

355 An initial result of our experiments is a grounded query interface called The Architecture  
356 Speaks, that is hosted on the Arm website [13, 1]. This interface is experimental, and a work  
357 in progress, but can already answer behavioural questions such as the ones we discussed in  
358 the introduction, as shown in Figure 11, and accessible at [7].

359 We have taken great care in making sure that the query interface points the user back  
360 to the Arm ARM location of the material used in the answer, thereby making its answers  
361 grounded in the source material. As one can see, we seem to be miles away from formal  
362 specifications all of a sudden. But we are already finding that people find this tool to be  
363 useful, simply to navigate the Arm ARM.

364 The main point is not so much the relative novelty of a query interface for the Arm  
365 ARM. Instead, the point is auditability of the current artefact, as it is written. A grounded

**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?

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:

- Answers are AI-generated and therefore might contain errors. Please make sure to check the answers before relying on them in any way.
- 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?**

**Definition of LDR (Load Register)**

**C6.2.215 LDR (immediate)**  
Load register (immediate)

This instruction loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

This is verbatim as per [C6.2.215 LDR \(immediate\) \[L27\]](#).

**Examples**

- Simple word load to a general-purpose register:**
  - `LDR W5, [X1]` loads a 32-bit word from the address in X1 into W5. The address is just the base register (offset 0), which is one of the standard A64 load/store addressing modes. [C1.3.3 Load/store addressing modes \[L55\]](#), [K1.2.1 Message passing \[L57\]](#).
- Immediate offset example:**
  - `LDR X0, [X1, #16]` (immediate form) loads a 64-bit doubleword from the address X1+16 into X0, using the LDR (immediate) behavior described above. [C6.2.215 LDR \(immediate\) \[L27\]](#), [C1.3.3 Load/store addressing modes \[L62\]](#).

**What LDR does in general**

- Basic effect:** LDR is a load instruction. It reads a value from memory and writes that value into a destination register. The value can be a 32-bit word or a 64-bit doubleword for the general-purpose LDR (immediate) form. This is as per [C6.2.215 LDR \(immediate\) \[L27\]](#).
- Address calculation:** For the general-purpose LDR (immediate) form, the memory address is formed from a base register (Xn or SP) plus an immediate offset, using one of the standard A64 load/store addressing modes (offset, pre-indexed, post-indexed, or unsigned scaled offset). This is as per [C6.2.215 LDR \(immediate\) \[L31\]](#) and [C1.3.3 Load/store addressing modes \[L27\]](#).
- Variants and sizes:** There are many LDR variants that differ only in what they load and how the address is formed:
  - General-purpose register loads: LDR (register offset), LDR (immediate), and LDR (literal) for words and doublewords. [C3.2.1 Load/store register \[L83\]](#).
  - Byte/halfword/signed variants (LDRB, LDRH, LDRSB, LDRSH, LDRSW) for different element sizes and sign-extension behavior. [C3.2.1 Load/store register \[L100\]](#).
  - SIMD & floating-point scalar loads: LDR used to load a single SIMD/FP register, with register, immediate, or PC-relative literal addressing. [C3.2.14 Load/store scalar: SIMD and floating-point \[L63\]](#).
  - SVE vector and predicate loads: LDR (vector) loads an SVE Z register, and LDR (predicate) loads an SVE predicate register, using a scalar base plus an immediate index. [C3.3.5 Unpredicated vector register load/store \[L27\]](#), [C3.3.6 Unpredicated predicate register load/store \[L27\]](#).
  - SME array/table loads: LDR (array vector) loads a ZA array vector, and LDR (table) loads the ZTO table register, from a scalar base (optionally with an immediate index). [C9.2.15 LDR \(array vector\) \[L27\]](#), [C9.2.158 LDR \(table\) \[L27\]](#), [C3.4.1 Array vector/table load and store \[L29\]](#).
- Unscaled vs scaled offsets:** Some LDR forms use an unscaled 9-bit signed immediate (byte offset used directly), while others use a scaled 12-bit unsigned immediate (offset multiplied by the access size). The unscaled forms exist to disambiguate cases where both encodings could represent the same byte offset. This is described for the general load/store register class in [C3.2.2 Load/store register \(unscaled offset\) \[L27\]](#).

**Summary**

- LDR is a family of load instructions that transfer data from memory into registers (general-purpose, SIMD/FP, SVE vector/predicate, or SME ZA/ZTO), using various addressing modes based on a base register plus optional offset or PC-relative literal. [C6.2.215 LDR \(immediate\) \[L27\]](#), [C3.2.1 Load/store register \[L83\]](#), [C3.2.14 Load/store scalar: SIMD and floating-point \[L63\]](#), [C3.3.5 Unpredicated vector register load/store \[L27\]](#), [C3.3.6 Unpredicated predicate register load/store \[L27\]](#), [C3.4.1 Array vector/table load and store \[L29\]](#).

Figure 11 Snapshot from The Architecture Speaks on the question “What does an LDR do?” [7]

366 query interface lets us ask whether the manual, as it is written, supports the questions users  
 367 actually ask. This leads to the following observation:

368 *Interfaces to query the prose as it is, not as we wish it to be,*  
 369 *make prose specifications more directly inspectable.*

370 The interface is not meant to be an oracle, nor to replace the Arm ARM. Its purpose  
 371 is to make the current prose more inspectable. This is why answers must be grounded in  
 372 source locations, why users must be able to audit the material used in the answer, and why  
 373 failures to answer are themselves useful signals: they may indicate that the prose does not  
 374 support a user question clearly enough.

375 **8 Towards computer-aided architecture development and consumption**

376 The Arm Architecture Formal Team’s experience suggests elements of a philosophy of  
 377 specification engineering that may be reusable in other contexts:

- 378 1. Treat behavioural questions as units of specification work.
- 379 2. Decide what kind of answer each behavioural question needs: accessible, unambiguous,  
 380 consensual, reproducible, auditable, maintainable, layerable.
- 381 3. Accept that perhaps a single artefact may not satisfy all those properties.
- 382 4. Engineer explicit bridges between artefacts.

## XX:14 On the role of prose in specifications

383 5. Use disagreement between artefacts as a review signal.

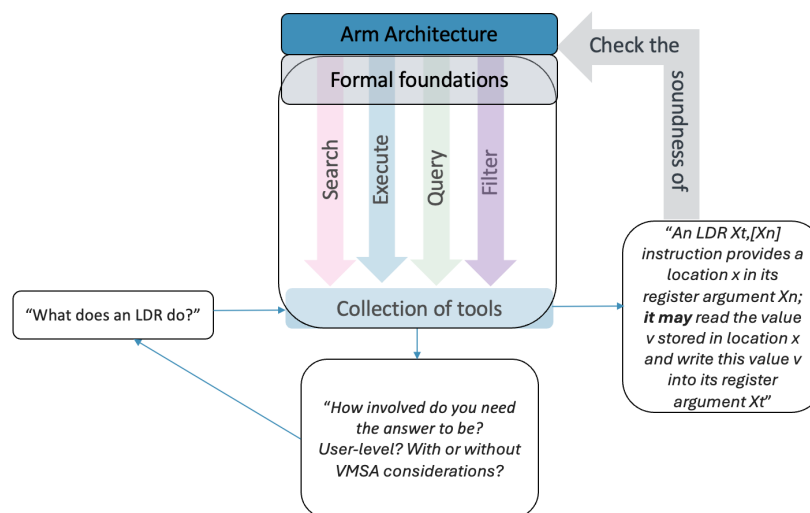
384 6. Inspect the prose users actually read.

385 Figure 12 shows how these elements arise from the examples in this paper.

Question	Artefacts	Properties
What does an LDR do?	Arm ARM prose, ASL, query interface	Accessibility, auditability
Is MP Allowed?	Litmus test, cat model, herd	Unambiguity, reproducibility
Why is the barrier MP test Forbidden?	Execution graph, Ordered-before relation, cat model	Unambiguity, auditability
How to write Ordered-before?	cat definition, Arm ARM transliteration, stakeholder review	Consensuality, accessibility, auditability

■ **Figure 12** Behavioural questions as units of specification work.

386 We believe that, as shown in Figure 13, certain specifications may need to be maintained  
 387 as connected artefact ecosystems, with prose, formal models, generated bridges, executable  
 388 tools, expert review and grounded query interfaces each contributing different kinds of answer.



■ **Figure 13** Vision for the Arm Architecture Formal Team

389 — **References** —

- 390 1 Jade Alglave. Introducing “The Architecture Speaks”. 2026. [https://](https://developer.arm.com/community/arm-community-blogs/b/ai-blog/posts/introducing-the-architecture-speaks)  
 391 [developer.arm.com/community/arm-community-blogs/b/ai-blog/posts/](https://developer.arm.com/community/arm-community-blogs/b/ai-blog/posts/introducing-the-architecture-speaks)  
 392 [introducing-the-architecture-speaks](https://developer.arm.com/community/arm-community-blogs/b/ai-blog/posts/introducing-the-architecture-speaks).
- 393 2 Jade Alglave, Richard Grisenthwaite, Artem Khyzha, Luc Maranget, and Nikos Nikoleris. Puss  
 394 In Boots: On Formalizing Arm’s Virtual Memory System Architecture. *IEEE MICRO*, 2024.
- 395 3 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in Weak Memory  
 396 Models. *CAV*, 2010.
- 397 4 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against  
 398 hardware. *TACAS*, 2011.
- 399 5 Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats. *TOPLAS*, 2014.
- 400 6 Jade Alglave and Luc et al Maranget. herdtools7. 2026. [https://github.com/herd/](https://github.com/herd/herdtools7)  
 401 [herdtools7](https://github.com/herd/herdtools7).
- 402 7 The Architecture Speaks chatbot. “what does an LDR do?”. 2026. [https://](https://developer.arm.com/architecture/the-architecture-speaks/?threadId=228d9997-7d58-4b2d-a1ce-2bfd6339aec3)  
 403 [developer.arm.com/architecture/the-architecture-speaks/?threadId=](https://developer.arm.com/architecture/the-architecture-speaks/?threadId=228d9997-7d58-4b2d-a1ce-2bfd6339aec3)  
 404 [228d9997-7d58-4b2d-a1ce-2bfd6339aec3](https://developer.arm.com/architecture/the-architecture-speaks/?threadId=228d9997-7d58-4b2d-a1ce-2bfd6339aec3).
- 405 8 Arm. Arm Architecture Reference Manual for A-Profile Architecture. [https://developer.](https://developer.arm.com/documentation/ddi0487/mb/?lang=en)  
 406 [arm.com/documentation/ddi0487/mb/?lang=en](https://developer.arm.com/documentation/ddi0487/mb/?lang=en).
- 407 9 Arm. ASL Reference. 2025. [https://developer.arm.com/architectures/](https://developer.arm.com/architectures/architecture%20specification%20language)  
 408 [architecture%](https://developer.arm.com/architectures/architecture%20specification%20language)  
 409 [20specification%20language](https://developer.arm.com/architectures/architecture%20specification%20language).
- 410 10 Arm. ASL stdlib proofs. 2025. [https://github.com/herd/herdtools7/tree/master/acl2/](https://github.com/herd/herdtools7/tree/master/acl2/asl/proofs/stdlib)  
 411 [asl/proofs/stdlib](https://github.com/herd/herdtools7/tree/master/acl2/asl/proofs/stdlib).
- 412 11 Arm. ASL1 code for LDR. 2025. [https://developer.arm.com/documentation/ddi0602/](https://developer.arm.com/documentation/ddi0602/2025-12/Base-Instructions/LDR--immediate---Load-register--immediate--)  
 413 [2025-12/Base-Instructions/LDR--immediate---](https://developer.arm.com/documentation/ddi0602/2025-12/Base-Instructions/LDR--immediate---Load-register--immediate--)  
 414 [Load-register--immediate--](https://developer.arm.com/documentation/ddi0602/2025-12/Base-Instructions/LDR--immediate---Load-register--immediate--).
- 415 12 Arm. ASLRef tooling. 2025. <https://github.com/herd/herdtools7/tree/master/asllib>.
- 416 13 Arm. The Architecture Speaks. 2026. [https://developer.arm.com/architecture/](https://developer.arm.com/architecture/the-architecture-speaks/)  
 417 [the-architecture-speaks/](https://developer.arm.com/architecture/the-architecture-speaks/).
- 418 14 Arm. Web interface to ASLRef tooling. 2026. [https://developer.arm.com/architectures/](https://developer.arm.com/architectures/tools/aslref)  
 419 [tools/aslref](https://developer.arm.com/architectures/tools/aslref).
- 420 15 Alastair Reid. Trustworthy specifications of Arm v8-A and v8-M system level architecture.  
 421 *FMCAD*, 2016.
- 422 16 Alastair Reid. ASL0 interpreter. 2020. [https://github.com/alastairreid/](https://github.com/alastairreid/asl-interpreter)  
 423 [asl-interpreter](https://github.com/alastairreid/asl-interpreter).
- 424 17 Hadrien Renaud. Executable semantics of Arm’s Architecture Specification Language. *JFLA*,  
 425 2024. <https://hal.science/hal-04406399/document>.
- 426 18 The Sail instruction-set semantics specification language. Kathryn E. Gray, Peter Sewell,  
 427 Christopher Pulte, Shaked Flur, robert Norton-Wright. [https://www.cl.cam.ac.uk/~pes20/](https://www.cl.cam.ac.uk/~pes20/sail/manual.pdf)  
 428 [sail/manual.pdf](https://www.cl.cam.ac.uk/~pes20/sail/manual.pdf).
- 429 19 Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner,  
 430 Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. Bringing  
 the WebAssembly Standard up to Speed with SpecTec. [https://dl.acm.org/doi/10.1145/](https://dl.acm.org/doi/10.1145/3656440)  
 3656440.