Chapter 7

Pagetables and the VMSA

This chapter is based, in part, on: Chapter D5 of the Arm Architecture Reference Manual DDI 0487H.a; and, Relaxed virtual memory in Armv8-A [54] by Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell, published in the proceedings of the 31st European Symposium on Programming (ESOP, 2022).

7.1 Introduction

Modern computers heavily rely on *virtual memory* to enforce security boundaries: hypervisors and operating systems manage mappings from virtual to physical addresses in order to restrict the access individual processes and guest operating systems have to the underlying physical memory, and to memory-mapped devices. With the endemic use of memory-unsafe languages, even for critical infrastructure, understanding and verifying the programs which manage virtual memory mappings is more vital than ever, driving current interests in hypervisors. The virtual machines those hypervisors enable are the key pieces of software which have become solely responsible for implementing such critical security properties.

The following chapters focus on these aspects of the architecture, on virtual memory and virtualisation and the software they enable, with the aim of giving a precise formal semantics for the purpose of verifying real systems software which use those features.

I first give a description of the sequential behaviour of Arm's virtual memory (this chapter); then describe the *relaxed* behaviours and any open questions about Arm's virtual memory (§8); give our precise axiomatic semantics that capture these behaviours (§9); give an overview of the tooling and validation of the given model(s) (§10); and, finally, a sketch of an equivalent operational semantics (§11).

This chapter overview The remainder of this chapter will give: a brief overview of Arm's virtual memory systems architecture (§7.2); a detailed description of the Arm translation table format (§7.3); an overview of the multiple stages of translation (§7.4), and the different translation regimes (§7.5); a detailed explanation of the official Arm translation table walk pseudocode (§7.6); and finally a discussion on the existence and purpose of translation lookaside buffers (§7.7). This chapter does not present any new contributions or novel research, instead, it is a brief but necessary overview of the required architectural features.

7.2 Virtual Memory

Armv8-A8's *virtual memory system architecture* (or *VMSA*) defines the virtual memory and virtulisation features of the Arm architecture. Its structure is described, in detail, in Chapter D5 of the Arm Architecture Reference Manual [1].

Conventionally, we think of memory as being a flat array of bytes, indexed by *physical addresses*. For smaller trusted devices, such as microcontrollers, this may be the end of the story. However larger 'application' class processors rely heavily on virtual addressing: interposing one or more layers of indirection between the accesses using the *virtual* addresses of the program and the 'true' physical addresses of memory. This indirection allows systems running on those processors to:

1. partition the physical resources between different programs, giving access to only those resources that each program needs, and protecting those resources from other programs that do not need to access them;
2. indirect accesses through specific ranges of addresses with convenient numeric values; and
3. update those indirections at runtime to add, remove, or otherwise modify, the mappings to physical memory, to support techniques such as copy-on-write and paging.

To manage all this, typical operating systems splits the programs into distinct *processes* and associates each process with its own virtual to physical mapping. These mappings take the form of partial functions from the process' own (virtual) addresses to the real hardware physical addresses along with some permissions:

$$\texttt{translate} : \texttt{VirtualAddress} \rightharpoonup \texttt{PhysicalAddress} \times 2^{\{\texttt{Read},\texttt{Write},\texttt{Execute}\}}$$

Note that this is a simplification. A more accurate `translate` function is given later on **TODO: ?REF?**.

Typically an operating system would create one such mapping for every process, partitioning the physical memory into disjoint subsets of physical addresses (the *range* of the translate function), and would allocate some convenient numeric values to be the virtual addresses the process interacts with (the *domain* of the translate function). Having this separation allows the processes to be given convienently aligned contiguous chunks of virtual address space even if the underlying physical resources are highly fragmented, or, in the case of paging, potentially not present in memory at all. Additionally, operating systems can provide many processes with mappings to the same physical resource (such as memory-mapped devices) and control which processes have access to such devices at any point in time.

These mappings give rise to separate *address spaces* for each process. The diagram in Figure 7.1 illustrates an example, with two processes named `P0` and `P1` each with their own virtual address space. The left-hand side shows a representation of the 'memory' as the processes see it, with the memory split into *pages* (fixed-size blocks of contiguous addresses). The right-hand side is the equivalent representation of the actual physical memory, with each *physical* page of the available RAM. Note that this diagram shows the virtual address space as being smaller than the physical one, but in general, they may be the same size, or the virtual address space may be even larger than the physical space.

If we assume each page has size `0x1000` then page `1` contains addresses `0x1000` to `0x1FFF` inclusive, and we can interpret the diagram like so:

- ▷ For `P0`:
  - virtual addresses in pages `1`, and `3` are unmapped.
  - virtual addresses in pages `0` and `2` map to physical addresses in physical page `1`.
  - virtual addresses in page `4` map to physical addresses in physical page `5`.
- ▷ For `P1`:
  - virtual addresses in pages `0` and `4` are unmapped.
  - virtual addresses in page `1` map to physical addresses in physical page `5`.
  - virtual addresses in page `2` map to physical addresses in physical page `7`.
  - virtual addresses in page `3` map to physical addresses in physical page `8`.

For example, if process `P0` reads the address `0x2305`, it will actually read from the physical location `0x1305`, since virtual page `2` was mapped to physical page `1` in `P0`'s address space.

Each address space corresponds to a distinct `translate` function. Note that these mappings may be: non-injective (contain *aliasing* of multiple virtual addresses to the same physical address); partial (where some virtual addresses do not map to a physical address at all); or overlapping with other processes' address spaces, in either the domain (for example, the physical page `5` is mapped in both `P0` and `P1`), or range (for example, the virtual page `2` is mapped in both `P0` and `P1` but to different physical pages), or both.

Large application-class processor architectures, such as Armv8-A, often provide hardware support in the form of the *memory management unit* (the MMU), which, once configured by software, will automatically perform the translation from virtual to physical addresses. Software is then required to manage a set of translation functions, and is responsible for ensuring the correct translation function is being used by the MMU whenever a context switch occurs, and handle any *faults* that the MMU generates.

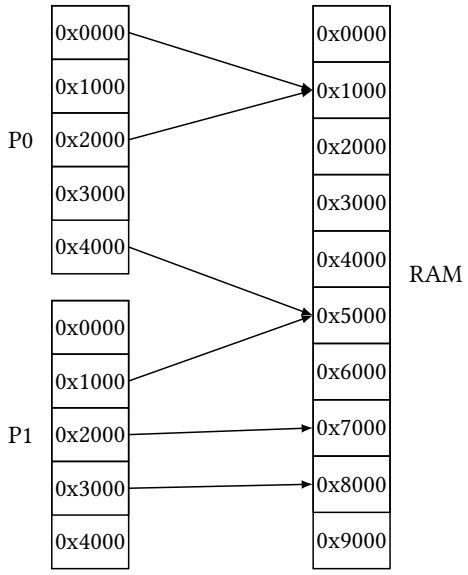


Figure 7.1: Example virtual and physical address spaces for two processes.

7.3 Arm Translation Tables

Software configures the MMU through the creation and modification of sets of *translation tables* (also referred to as *page tables*) for each of the translation functions.

The translation tables form an in-memory tree data structure which encode the (partial) `translate` function. Software creates and maintains these trees, and tells the MMU which tree (and so which translation function) to use at runtime. The hardware then reads from this tree structure to perform the translation, or from one of the various caching structures described in **TODO: ?REF?**, whenever the process reads from, or writes to, memory.

A pointer to the root of the tree is stored in the TTBR ("Translation table base register") register (or rather, one of the various base registers described in more detail in **TODO: ?REF?**), and this determines which translation function is currently in use by that processor.

Each node in the tree is a page-aligned chunk of memory which is treated as an array of 64-bit entries. Each entry is responsible for mapping some fixed part of the domain of the translation function, with the root table mapping the entire address space.

The tree is separated into different *levels*. with a root table pointed to by the base register and each subsequent child tree increases in level going deeper into the tree. Typically the root is at level 0 with a maximum depth of 4 (down to level 3), but the various configurations are discussed in the next section.

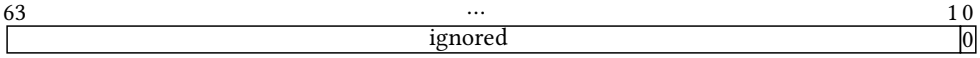
7.3.1 Translation table format

Arm's virtual memory system architecture is highly configurable. Writing to the SCTLR ("System control register") and TCR ("Translation control register") system registers allow the software developer to choose a configuration from a whole host of various options. To give a flavour of this configurability I list some of the configuration bits, some of which will be discussed in more detail in the next chapter; these include: the size of virtual addresses; the number of levels in the tree; the starting level; the size of a single page (or in Arm terminology, the size of the *translation granule*); the number of ASIDs and VMIDs; alignment requirements; memory attributes for hardware walks; enabling hardware management of access flags and dirty bits; write-execute-never permissions; and so on. To simplify things, in this dissertation, we consider just one common configuration, the one currently used by the Linux kernel: a tree of translation tables with maximum depth 4, with 4KiB pages with 48-bit addresses, unless explicitly stated otherwise.

Each node is a table of 512 64-bit entries, bound as one 4096-byte block of memory. Each table controls the mapping of a fixed range of the virtual address space. This range is split into 512 equal slices, with each entry responsible for its slice. Each of those entries can be one of:

1. An *invalid* entry, which indicates that this slice of the domain is unmapped;
2. A *table* entry, pointing to a next-level table (a child tree) which recursively maps this slice of the domain; or
3. A *page* (last-level) or *block* (non-last-level) entry which defines a single fixed-size mapping for this slice of the domain.

Invalid entries An invalid entry is defined by `bit[0]` of the entry being `0`. The top 63 bits are ignored by hardware, and software is free to use those bits to store any metadata it wishes. Invalid entries may exist at any level in the tree.



Block or page entries Block and page entries are similar to each other; both create a mapping for a contiguous slice of the domain mapped by the entry, encoded as an output address (OA) with some metadata (including access permissions, memory type, and some software-defined bits).

The OA is aligned to the size of the slice of the domain being mapped. For page entries, the OA is aligned on a page boundary. A block entry's OA at level 2 would be 2MiB aligned, and a block entry's OA at level 1 would be GiB aligned. This corresponds to the hardware reserving `bits[n:12]` of the entry to be `0` depending on how deep the entry is: at level 1 `n==30`; at level 2 `n==21`; and at level 3 `n==12`.

Block entries can exist at levels 1 and 2. Page entries can only exist at level 3.

For block entries `bit[1]` is `0`, for page entries `bit[1]` is 1.

Metadata (access permissions, shareability, memory type) are encoded into the `attrs` bits.

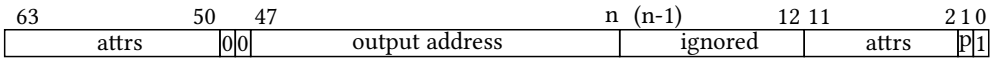
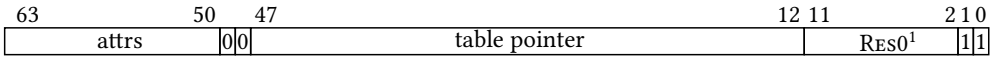


Table entries A table entry contains a page-aligned pointer to a child table, but can also contain similar metadata as the block or page entry, including access permissions (read/write/execute), which are combined with any permissions from the child table.

Table entries are allowed only at levels 0–2.



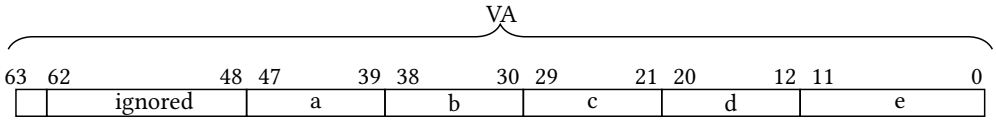
7.3.2 The Arm translation table walk

When the processor executes an instruction which takes an address, such as the Arm `LDR` or `STR` instructions, those addresses are virtual (addresses used by instructions are always virtual addresses). The hardware converts each virtual address to a physical address, and the MMU performs this conversion.

To do this, the MMU reads the `TTBR` to get the currently in-use tree of translation tables. Then the MMU itself reads memory and walks the tree (except when it can read from a previously cached translation, as described in the next chapter) effectively computing the partial `translate` function the tree encodes, producing the physical address and any permissions, or reporting a fault back to the processor if the virtual address was unmapped, or if the permissions forbid the requested operation.

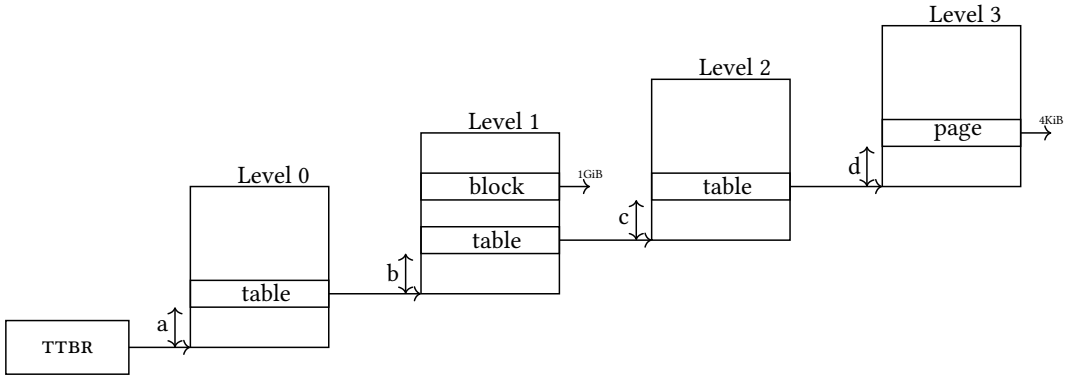
Walk overview The hardware walker first slices up the input virtual address into chunks: the most-significant bit is used to determine which base register to use (see §7.5); the next 15 bits are typically ignored by hardware; the rest of the address is split into 9–bit fields which we refer to as fields *a*–*d*, with the remaining bits as field *e*. Fields *a*–*d* are used for indexing into the tables; and field *e* is the offset in the page, which is added to the final output address.

[1]The Arm architecture requires these bits are 0 and are reserved for future use.



The walk then proceeds, with the MMU taking the following steps:

1. Read the `TTBR` register.
2. Perform a 64-bit single-copy atomic read of `Mem[TTBR+8*a]` to read the entry in the Level 0 table. Call the result `L0entry`.
   - *a* If `L0entry[0]` is 0 (that is, it's an invalid entry) then report a fault back to the processor.
   - *b* Otherwise if `L0entry[1]` is 0 then report a fault back to the processor (top-level tables cannot have block mappings).
3. Perform a 64-bit single-copy atomic read of `Mem[L0entry.table_pointer+8*b]` to read the entry in the Level 1 table, which we will call `L1entry`.
   - *a* If `L1entry[0]` is 0 then report a fault back to the processor.
   - *b* If `L1entry[1]` is 0 (it's a block entry):
     - *i* If the access is not permitted (See §7.3.2 "**Access permissions**"), report a fault to the processor.
     - *ii* Otherwise, return the output address (See §7.3.2 "**Computing the final output address**") back to the processor.
4. Perform a 64-bit single-copy atomic read of `Mem[L1entry.table_pointer+8*c]` to read the entry in the Level 1 table, which we will call `L2entry`.
   - *a* If `L2entry[0]` is 0 then report a fault back to the processor.
   - *b* If `L2entry[1]` is 0 (it's a block entry):
     - *i* If the access is not permitted, report a fault to the processor.
     - *ii* Otherwise, return the output address back to the processor.
5. Perform a 64-bit single-copy atomic read of `Mem[L2entry.table_pointer+8*d]` to read the entry in the Level 3 table, which we will call `L3entry`.
   - *a* If `L3entry[0]` is 0 then report a fault back to the processor.
   - *b* Else if `L3entry[1]` is 0, report a fault back to the processor (this encoding is reserved and is treated as invalid).
   - *c* `L3entry[1]` is 1 (it's a page entry):
     - *i* If the access is not permitted, report a fault to the processor.
     - *ii* Otherwise, return the output address back to the processor.



Computing the final output address The output address (OA) of the final descriptor is the start of the range mapped by the entry. The low order bits are all 0 in the output address, and need to be added on to compute the final output address of the translation.

To compute this final output address the MMU takes the OA from the entry, and the level in the tree the entry is at, and 'completes' the address by bitwise appending the remaining fields to create the complete 48-bit output address. Recall that the `OA` field of the block mappings gets wider the deeper in the tree you are, and so for a 1GiB entry the `OA` field is only 18 bits wide but for a 4KiB page entry its `OA` field is the full 36 bits.

- ▷ For a 1GiB (level 1) block entry; `PA = OA::c::d::e`
- ▷ For a 2MiB (level 2) block entry; `PA = OA::d::e`

- For a 4KiB (level 3) page entry; `PA = OA::e`

Note that this process means that the least-significant 12 bits of the input VA are unchanged and remain the same in the final output PA, regardless of how the translation function is configured.

Access permissions Once the walk is complete, and the final output address calculated, the MMU checks to see whether the requested access is permitted. Each level of the table can contain some access permissions and those permissions get combined at the end to calculate the final permissions.

For data accesses (reading and writing), table entries have an `APTable` field (`bits[62:61]`), and block/page entries have `AP[2:1]`[1] field (`bits[7:6]`). These fields can be decoded using the following table:

Field	When set (1)	When unset (0)
`AP[2]`	Read-only	Read&Write
`AP[1]`	Allow at EL1&0	Allow at EL1 only
`APTable[1]`	Force read-only	No effect on permissions.
`APTable[0]`	Force forbid access at EL0	No effect on EL0 permissions.

For executable permissions, which permit or forbid instruction fetching from some region of memory, there are no dedicated encodings of the access permission bits. Instead, all mappings are executable by default, unless one of the following applies: the region is mapped writeable at EL0, as writable EL0 regions are never executable at EL1; a global `WXN` ("Write-execute-never") configuration bit is set, and the entry was writeable; or, when one of the various translation table entry `XN` ("Execute-never") bits are set. For simplicity, this chapter assumes that execute-never bits are always disabled; see the full description in the Arm ARM **TODO: ?REF?**for more information.

To combine access permissions from the whole walk, the MMU takes the bitwise union of each of the `APTable` fields from each table entry, and then intersects the result with the final `AP[2:1]` field to produce a final set of permissions. Figure 7.2 contains a decoding table for a given table and leaf access permissions, for testing whether a requested access is permitted. If the requested access is not permitted, then the MMU generates a permission fault, which is reported back to the processor.

Faults The MMU may emit one of several fault types during a translation table walk (these are referred to by Arm as the *MMU fault* types):

- Translation fault.
  These are caused by the mapping being invalid, either because `bit[0]` was 0, or because the descriptor encoding was reserved-as-invalid. Translation faults also result from trying to translate an address that is outside the 48-bit input address range.
- Permission fault.
  For when the mapping was valid, but the access permissions do not permit the requested access (for example, trying to write to a read-only address).
- Access flag fault.
  These are generated when hardware management of access flags is disabled and the access flag bit is set.
- TLB Conflict aborts (see **TODO: ?REF?**).
- Alignment fault.
  Generated when an operation expects an aligned memory address, but is given a misaligned one, and alignment checking is enabled in the `SCTLR`.
- Address size fault.
  For when the OA, or `TTBR`, has a value that is out of the physical address range.
- Synchronous external abort on a translation table walk.
  These are *external aborts* (that come from the system not from the MMU) that happen due to accesses that the MMU generated. For example, if the next-level table field pointed to an address for which there was no memory or device, the system-on-chip would return a fault to the processor.

These faults lead to processor exceptions. The fault type is stored in the `ESR_ELn` ("exception syndrome register") register's `EC` ("exception class") field, and any supplementary information is stored in its `ISS` ("instruction specific syndrome") field (such as which level in the tree the fault came from, whether the originating instruction was a read or a write, and). Exception handling code can read the `ESR` register to determine the fault type and cause,

[1] Block/page entries do not store the entire `AP` field but only `AP[2:1]`. `AP[0]` is not present in AArch64.

APTable[1]	APTable[0]	AP[2]	AP[1]	EL1			EL0		
				R	W	X	R	W	X
0	0	0	0	✓	✓	✓	×	×	✓
0	0	0	1	✓	✓	×	✓	✓	✓
0	0	1	0	✓	×	×	×	×	✓
0	0	1	1	✓	×	✓	✓	×	×
0	1	0	0	✓	✓	✓	×	×	✓
0	1	0	1	✓	✓	×	×	×	✓
0	1	1	0	✓	×	×	×	×	✓
0	1	1	1	✓	×	✓	×	×	×
1	0	0	0	✓	×	✓	×	×	✓
1	0	0	1	✓	×	×	✓	×	✓
1	0	1	0	✓	×	×	×	×	✓
1	0	1	1	✓	×	✓	✓	×	×
1	1	0	0	✓	×	✓	×	×	✓
1	1	0	1	✓	×	×	×	×	✓
1	1	1	0	✓	×	×	×	×	✓
1	1	1	1	✓	×	✓	×	×	×

Figure 7.2: Merging Access Permissions (Stage 1, EL1&0).
Entries in **red** highlight differences from the `APTable=00`.

and can read the `FAR_ELn` ("fault address register") to determine the virtual address which triggered the fault, and handle the fault appropriately.

Memory Attributes The processor does not necessarily know what is located at any physical address. There may be some dynamic random-access memory (DRAM, what one would generally consider 'memory'), but there may also be other memory-mapped devices, or non-volatile memory, or other peripherals, or possibly nothing at all.

To help accommodate this, hardware allows software to mark regions of memory as one of either *device* memory, *normal cacheable* memory, or normal *non-cacheable* memory, using the translation tables.

The desired memory type is determined from the `AttrIndx` field (`bits[4:2]`) in block and page entries. Instead of being directly encoded into this field, Arm chose to have the actual attributes stored in a separate register: the `MAIR` ("Memory attribute indirection register") register. The `MAIR` stores an array of eight 8-bit fields each of which contains an encoding of a memory type. The `AttrIndx` field in the entry is an integer in the range 0–7, which is the index of the field in the `MAIR` register to use.

This indirection means that the final result of translation depends not only on the value of the final leaf entry in memory, but on the value of certain system registers, such as the `MAIR`, at that time of the translation table walk.

Below are the three most common encodings for a `MAIR` field, and the ones that will be useful later when discussing tests:

- `0b0000_0000`: device memory.
- `0b0100_0100`: normal non-cacheable memory.
- `0b1111_1111`: normal cacheable memory, inner&outer write-back non-transient, read&write-allocating.

Memory locations marked as device tell the hardware that reads or writes to those locations may have side-effects. This means hardware treats those locations differently: there will be no speculative instruction fetches, reads, or writes to those locations; writes to those locations will not *gather* into larger writes; reads and writes to those locations will not re-order with respect to others; those locations generally will not get cached; and other thread-local optimizations get disabled. Note that Arm define a wide range of device memory types, allowing

the systems programmer to selectively re-enable some of the previously described behaviours to enable better performance where they deem it safe to do so.

For normal memory the software can choose between *cacheable* or non-*cacheable* memory. Arm provide a range of different options for the cacheability:

- non-cacheable
- write-back cacheable
- write-through cacheable

As with other features, there is a wide scope for configuration: separately configuring inner (L1,L2) and outer (L3) caches, and adding cache allocation hints (allocating on reads, writes or both).

As we will see later (**TODO: ?REF?**), the ability to change cacheability, or even have multiple aliases with different cacheability attributes, give rise to interesting behaviours and security considerations.

7.4 Virtualisation and a second stage of translation

So far this chapter has focused on operating systems and processes. However, modern systems isolate not just processes within an operating system but entire operating systems from one another, within a hypervisor.

To do this, software uses the virtual memory abstraction again, adding an extra layer. This layer, like the previous one, is supported by hardware. Processes use virtual addresses which are converted to *intermediate physical* (also sometimes known as *guest physical*) addresses using the operating system's configured translation tables but then these intermediate physical addresses (IPAs) go through another round of translation to convert those IPAs into the final physical address.

Arm calls these *stages* of translation, and the MMU supports both stages and can perform the full translation from virtual to physical (via the intermediate physical) address.

This means software must manage two sets of translation tables: operating systems manage the *stage 1* tables to convert VAs to IPAs; and hypervisors manage *stage 2* tables to convert those IPAs to PAs; this gives two separate `translate` functions, which the MMU composes together at runtime:

$$\texttt{translate\_stage1} : \texttt{VirtualAddress} \rightharpoonup \texttt{IPA} \times \texttt{Permissions} \times \texttt{MemoryType}$$
$$\texttt{translate\_stage2} : \texttt{IPA} \rightharpoonup \texttt{PhysicalAddress} \times \texttt{Permissions} \times \texttt{MemoryType}$$

Hypervisors (running at EL2) can configure the stage 2 translate function by creating translation tables with a similar format as before and then storing a pointer to the root of this tree in the `VTTBR` ("Virtualization translation table base register") register. The MMU will read the `VTTBR` whenever it needs to perform a second-stage translation to convert an IPA to a PA, and will do the translation table walk over that tree in much the same way as described earlier for (what we can now call) the first-stage translation.

This results in two address spaces, a virtual address space and an intermediate-physical address space. Figure 7.3 contains an example layout of these address spaces for a machine running three processes (`P0`,`P1`,`P2`) in two operating systems (`OS0`,`OS1`). As with the earlier diagram in Figure 7.1, each column is a (set of) address spaces, with transformations between them defined by their respective translation functions. On the left-hand side are the virtual address spaces of the various processes, whose virtual addresses are translated (using the translation tables pointed to by the `TTBR` register) into intermediate-physical addresses in the central address spaces (for the respective OS). Those IPAs are then translated (using the `VTTBR`) into the final physical address.

Concretely, if `P1` reads from address `0x1001`, it will be translated into the IPA `0x3001` in `OS0`'s address space, which then gets translated again, and the processor will actually read from RAM at location `0x6001`.

Differences in the translation table format from stage 1 Stage 2 translation tables are similar to their stage 1 counterparts, but there are some minor differences:

- Stage 2 table entries do not have any additional attributes, and so do not have an `APTable` field.
- Stage 2 `AP` field (called `S2AP`) has a slightly different (and simpler) format, see Figure 7.4.
- Stage 2 block and page entries do not have a `MemAttrIndx` field but rather encode the memory type directly into the `MemAttr` field `bits[5:2]` (see the full description in the Arm ARM [1, D5-4874] for all possible encodings):

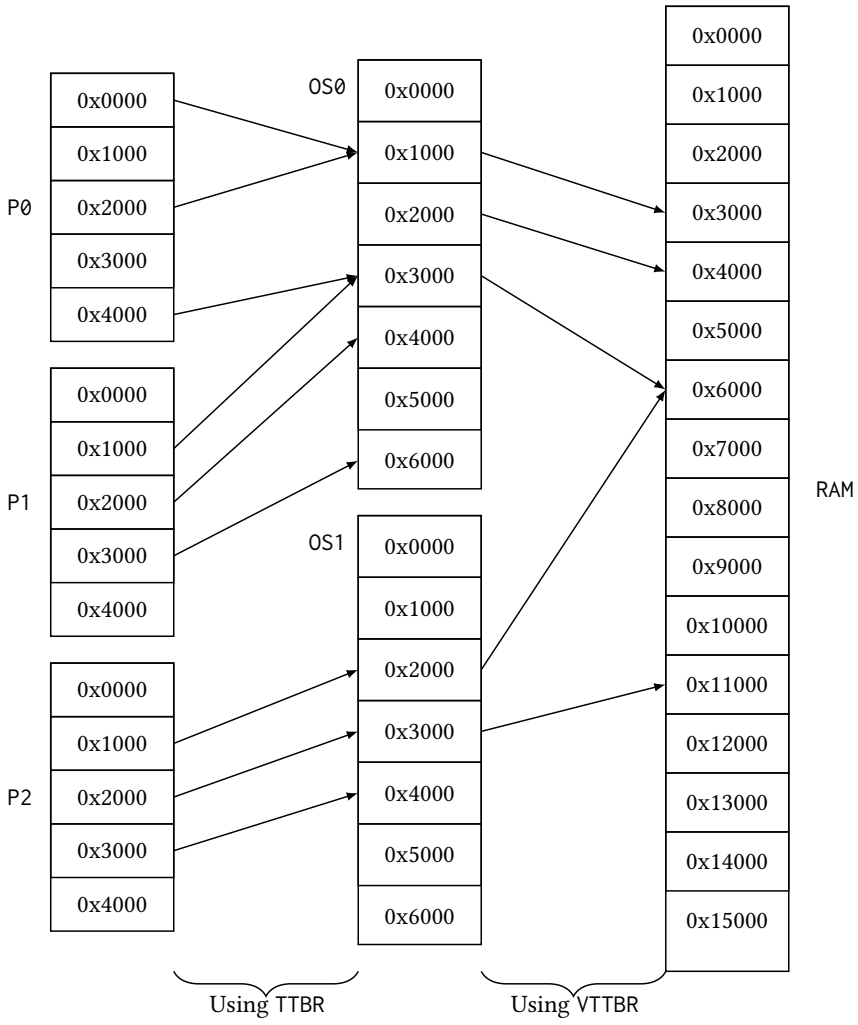


Figure 7.3: Example virtual, intermediate physical, and physical address spaces for three processes running on two operating systems.

Field	When set (1)	When unset (0)
S2AP[1]	Writeable	not Writeable
S2AP[0]	Readable	not Readable

Figure 7.4: **S2AP** field encoding.

- `0b0000`: Device memory.
- `0b0101`: Normal non-cacheable.
- `0b1111`: Normal write-back inner&outer cacheable.

These are interesting as they mean that the stage 1 and stage 2 attributes (permissions and memory types) must be *combined* in order to produce the final output. This combination is not just a case of letting stage 2 overrule the stage 1 settings but rather that both stages get a veto: if stage 1 sets the memory type to be device or non-cacheable then it overrules what stage 2 sets. Similarly, if stage 1 permissions forbid an access then the stage 2 permissions cannot overrule that.

Second-stage translations during a first-stage walk There is a complication with the story so far. The stage 1 tables are created by the operating system, which is using an intermediate physical address space, not a physical one. The writes the OS does to the tables will be translated, as they are normal data writes. But, the tables themselves contain references to other tables, and those entries will be intermediate physical addresses, and so, they must also be translated, including the value of the `TTBR` itself.

In our assumed configuration of 4KiB pages and 4 levels of translation, this leads to a maximum of 24 memory accesses to perform the translation: 4 reads of stage 1 translation tables, 16 reads of stage 2 translation tables during those stage 1 walks, and a final 4 reads of the stage 2 translation tables to translate the output IPA into the final PA.

7.5 Translation regimes

As mentioned earlier, there are multiple translation table base registers. Each of them defines a translation function, pointing to the root of the tree of translation tables which define it. These translation functions are then composed together into various translation *regimes*, each defining the set of translation functions (and therefore which translation table base registers) which will be used for translations done by the processor.

Arm define a set of these translation regimes. Figure 7.5 gives an overview of three of the most common regimes, which are:

- ▷ EL1&0 (two-stage)
  - For programs executing at EL0 or EL1 when virtualisation (at EL2) is enabled.
  - VAs with the high bit set are translated into IPAs using the EL1-configured register, `TTBR1_EL1`. VAs are typically split into 'high' and 'low' regions with different translations, primarily used for separate kernel and user address spaces.
  - VAs without the high bit set are translated into IPAs using the EL1-configured register, `TTBR0_EL1`.
  - IPAs are translated to PAs using the EL2-configured `VTTBR_EL2` register.
- ▷ EL1&0 (single-stage)
  - For programs executing at EL0 or EL1 when virtualisation (at EL2) is disabled.
  - VAs with the high bit set are translated into PAs using the EL1-configured register, `TTBR1_EL1`.
  - VAs without the high bit set are translated into PAs using the EL1-configured register, `TTBR0_EL1`.
- ▷ EL2
  - For programs executing at EL2.
  - VAs without the high bit set are translated into PAs using the EL2-configured register, `TTBR0_EL2`.
  - VAs with the high bit set are always unmapped.

Which translation regime is being used is defined by various system registers and the current system state.

- ▷ Translations at EL1 or EL0 use one of the EL1&0 regimes.
- ▷ Translations at EL2 use the EL2 regime.
- ▷ `TCR_EL2` (set at EL2) determines whether the EL1&0 is a single-stage or two-stage regime.
- ▷ `TTBR0_EL1`, `TTBR1_EL1` determine the stage 1 of the EL1&0 regimes, and can be set at EL1 or higher.
- ▷ `TTBR0_EL2` determines the stage 1 of the EL2 regime, and can only be set at EL2 or higher.

▷ VTTBR_EL2 determines the stage 2 of the EL1&0 regime, and can only be set at EL2 or higher.

Arm define a wide range of other regimes, see the Arm ARM **TODO: ?REF?**. For simplicity, we ignore secure modes, including all of EL3.

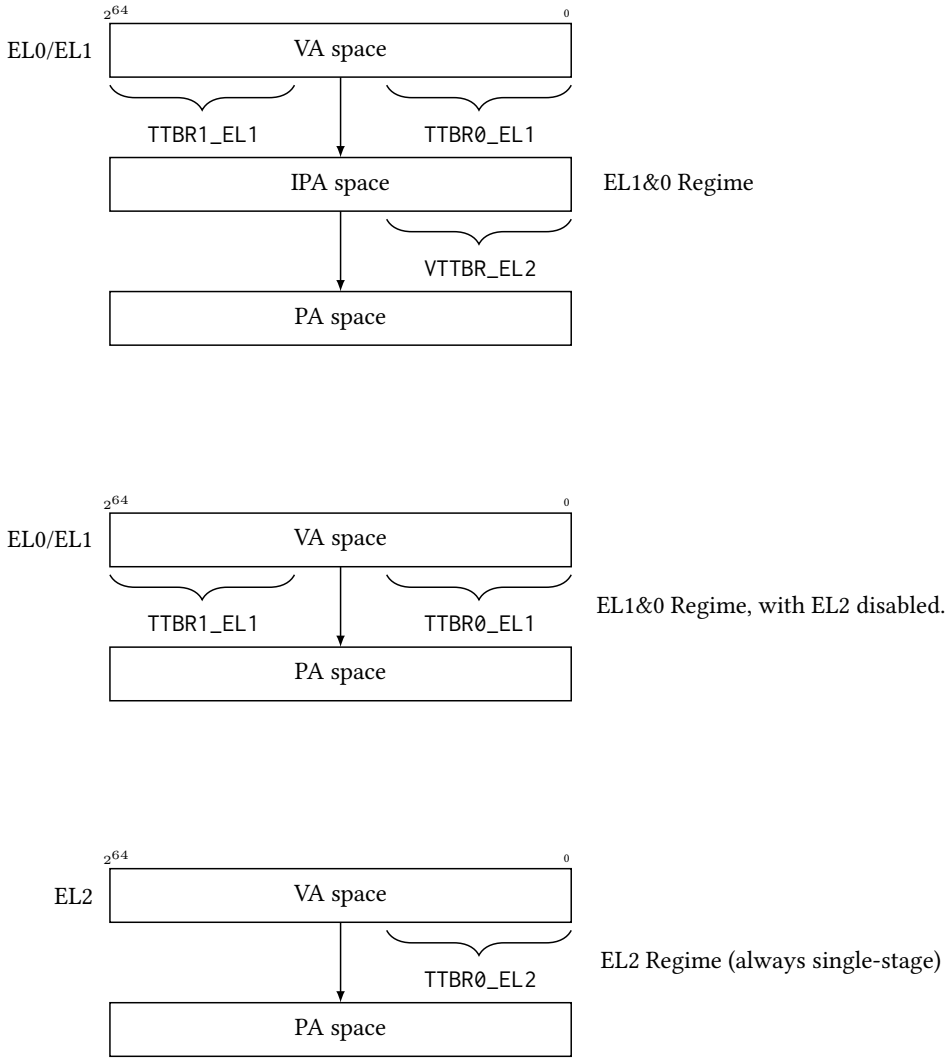


Figure 7.5: Translation regimes that apply to EL0,EL1, and EL2.

7.6 Arm pseudocode

It is now useful to examine the official Arm pseudocode, especially those parts that relate to memory events.

We will do this in three steps: first, by looking at the pseudocode that is executed for an Arm store instruction; following the memory accesses that it performs down to any translations it performs; finally looking at the Arm translation table walker in full. There is a lot of detail infused throughout the Arm psueocode, so in this section we shall focus on the most pertient parts, and give some idea of what detail is omitted.

7.6.1 The lifecycle of a store

Arm give precise executable semantics for every instruction in their domain-specific Architecture Specification Language (ASL). This ASL code defines the sequential intra-instruction behaviour of each instruction, including memory accesses, and any translation table walks they perform.

```
1         bits(64) address;
2         bits(datasize) data;
3
4         if HaveMTE2Ext() then
5             SetTagCheckedInstruction(tag_checked);
6
7         if n == 31 then
8             if memop != MemOp_PREFETCH then CheckSPAlignment();
9             address = SP[];
10        else
11            address = X[n];
12
13        if ! postindex then
14            address = address + offset;
15
16        case memop of
17            when MemOp_STORE
18                if rt_unknown then
19                    data = bits(datasize) UNKNOWN;
20                else
21                    data = X[t];
22                Mem[address, datasize DIV 8, acctype] = data;
23
24            when MemOp_LOAD
25                data = Mem[address, datasize DIV 8, acctype];
26                if signed then
27                    X[t] = SignExtend(data, regsize);
28                else
29                    X[t] = ZeroExtend(data, regsize);
30
31            when MemOp_PREFETCH
32                Prefetch(address, t<4:0>);
33
34        if wback then
35            if wb_unknown then
36                address = bits(64) UNKNOWN;
37            elsif postindex then
38                address = address + offset;
39            if n == 31 then
40                SP[] = address;
41            else
42                X[n] = address;
43
```

Figure 7.6: Arm "STR (immediate)" ASL code.

TODO: the importance of the ASL, and of sequential v concurrent behaviour will already be explained, but recap here anyway?

Figure 7.6 shows the Arm ASL for the "STR (Immediate)" instruction: `STR Xt,[Xn]`. This instruction writes the value contained in register `Xt` into the memory location stored in register `Xn`. The figure has some uninteresting (for this thesis) parts greyed out: those parts that deal with optional extensions such as memory tagging; unknown register values; register writeback; and, the load and prefetch instructions which use the same ASL code.

The ASL code first reads the virtual address either from the stack pointer (line 9) or by reading register `Xn` (line 11). It then reads the data from the register `Xt` (line 21), which will be written to memory. Finally, it performs the store itself using the `Mem[]` function (line 22).

7.6.2 Writes to memory

The `Mem[]` function is responsible for checking alignment and performing each memory access the instruction does. The ASL for `Mem[]` can be found in Figure 7.7.

It does some alignment checks, and then calls `MemSingle[]` once for each single copy atomic write the access performs.

For example, for a fully aligned store, it calls `MemSingle[]` just once (lines 37 or 57), and, for a misaligned store, it will call `MemSingle[]` once for each byte (line 51).

The `MemSingle[]` call then performs the translation, and (if successful), the actual write to memory. Its ASL can be found in Figure 7.8, with parts for extensions and store pair greyed out. On line 12, it calls `AArch64.TranslateAddress` to do the translation table walk. If the translation succeeds, then the code calls `PhysMemWrite` (on line 40), an uninterpreted function with no behaviour in ASL, which represents the actual write to memory. After perhaps handling any external aborts from the write, the function returns.

```
Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
    boolean iswrite = TRUE;
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    bits(halfsize*8) lowhalf, highhalf;
    boolean atomic;
    boolean aligned;
    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if ispair then
        atomic = CheckAllInAlignedQuantity(address, size, 16);
    elsif size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
    elsif (acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);
    else
        // 16-byte integer access
        atomic = address == Align(address, 16);

    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_aligned = TRUE;
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype, single_is_aligned, ispair] = lowhalf;
        AArch64.MemSingle[address + halfsize, halfsize, acctype, single_is_aligned, ispair] = highhalf;
    elsif atomic && ispair then
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    elsif !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype, aligned, ispair] = lowhalf;
        AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair] = highhalf;
    else
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;
```


Figure 7.7: Arm `Mem[]` write function call ASL code.

```
AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned
  , boolean ispair] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    if HaveLSE2Ext() then
        assert CheckAllInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size)
    ;
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.mematttrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    AccessDescriptor accdesc;
    if HaveTME() then
        accdesc = CreateAccessDescriptor(acctype);
        accdesc.transactional = TSTATE.depth > 0;
        if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.mematttrs)
     then
            FailTransaction(TMFailure_IMP, FALSE);
    else
        accdesc = CreateAccessDescriptor(acctype);

    if HaveMTE2Ext() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

    PhysMemRetStatus memstatus;
    (atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.mematttrs,
     size, acctype, iswrite, aligned, ispair);
    if atomic then
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    elsif splitpair then
        assert ispair;
        bits(halfsize*8) lowhalf, highhalf;
        <highhalf, lowhalf> = value;

        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
    else
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return;
```

Figure 7.8: Arm `MemSingle[]` write function call ASL code.

7.6.3 Translation table walks

It is the AArch64.TranslateAddress function which begins the process that performs the actual translation table walk, converting the input virtual address to the physical one. The full ASL code is too much to contain in a single figure, and so it can be found in §7.8 at the end of this chapter. This section will reference the relevant lines from the translation table walk ASL.

Figure 7.9 is an example trace of the execution of the STR Xt,[Xn] instruction, as it would happen if we were to execute it from EL1 in the EL1&0 two-stage regime. Each node represents an event in the trace (a memory or register access), and the arrows between them represent control flow. **TODO: Generate from an actual isla trace rather than by hand? at least to be proper... TODO: Give labels to each event?**

As described before, the instruction starts by reading the Xt and Xn registers, before beginning the call to AArch64.TranslateAddress.

The events drawn inside the dotted box come from accesses during the call to the translation table walk functions. It first calls FullTranslate (in AArch64.TranslateAddress, page 61, line 2), which calls S1Translate (in AArch64.FullTranslate, page 62, line 12), which calls S1Walk (in AArch64.S1Translate, page 63, line 29) to do the actual first-stage translation table walk. It begins by reading the relevant TTBR register to get the root table address (in AArch64.S1Walk, page 66, line 9). This is stored in a walkstate struct, which the ASL code uses to keep track of the state that changes as the walk progresses, notably, the next-level table address and any accumulated permissions. It then begins the loop to do the walk, starting from the table address read from the TTBR. On each iteration of the loop, the intermediate-physical address of the entry to be read is computed (in AArch64.S1Walk, page 66, line 38), and passed through a second stage of translation (in AArch64.S1Walk, page 66, line 47).

This second stage translation calls S2Walk, which behaves similarly to the S1Walk function, taking the following steps: it reads the VTTBR (in AArch64.S2Walk, page 70, line 11); computes the (now) physical address of the entry to read (in AArch64.S2Walk, page 70, line 41); and reads it (in AArch64.S2Walk, page 70, line 44), eventually calling PhysMemRead (in AArch64.FetchDescriptor, page 72, line 23), which appears as the first R S2 L0 node in Figure 7.9.

S2Walk continues to loop, each time updating the running walkstate with the next-level table address from the decoded descriptor (in AArch64.S2Walk, page 70, line 53), until a leaf entry is found. It is either invalid (in AArch64.S2Walk, page 71, line 65), or, a valid page or block entry (in AArch64.S2Walk, page 71, line 70). These correspond to the next three R S2 Ln events in the figure.

Assuming the walk did not fail with a fault, the S2Translate function returns with the physical address of the stage 1 level 0 table. S1Walk can continue, performing a read of the physical memory in the table (in AArch64.S1Walk, page 66, line 52). From there, S1Walk continues in much the same way as the stage 2 walk did: computing the current table intermediate-physical address, translating it to get the physical address, performing the read of memory to get the descriptor, until a leaf entry is found.

This process generates all the events up to, and including, the final stage 1 entry read (the R S1 L3 event), returning the intermediate-physical address that S1Walk computed.

Finally, FullTranslate calls S2Translate one last time (in AArch64.FullTranslate, page 62, line 22) on the intermediate-physical address, generating the last Rreg(VTTBR) and R S2 Ln events, and producing the final PA of the translation.

This output PA is what is passed to the PhysMemWrite of the MemSingle[] call we saw earlier, generating the final W [pa]=data event in the trace.

7.7 Caching in TLBs

Hardware does not simply perform the (up to) 24 additional memory accesses for every instruction-fetch, read, or write. This would have an unacceptable performance penalty. Instead, the results of previous translations of the same address are cached, in specialised structures called *Translation Lookaside Buffers*, or simply TLBs. These TLBs can store whole translation results, or the separate virtual and intermediate-physcical mappings, or individual translation table entries, or a mix of the above, which we will explore more in the next chapter.

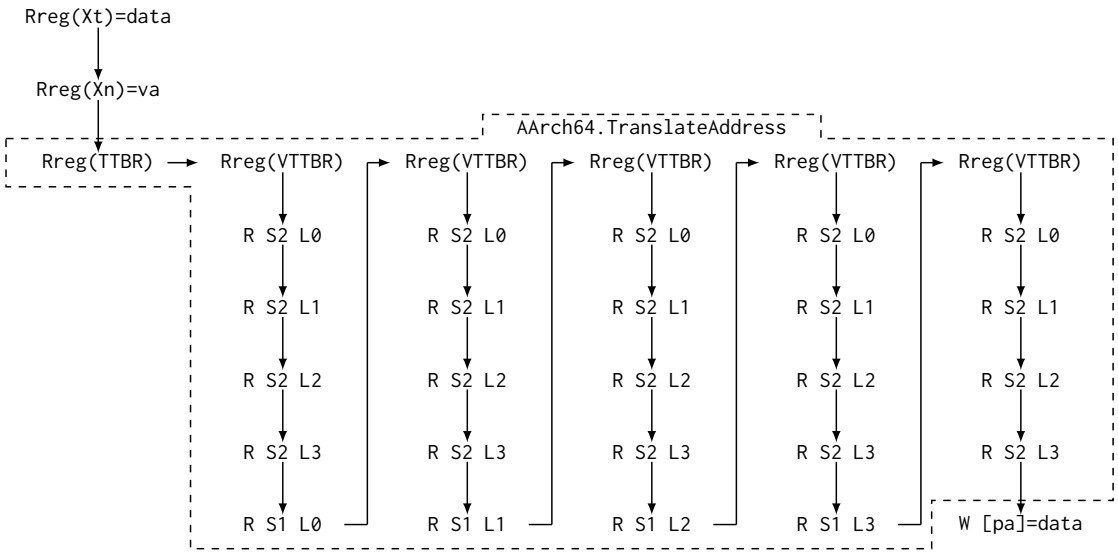


Figure 7.9: Memory and register accesses during a '`STR Xt,[Xn]`' instruction.

When the processor translates a virtual address, it first looks for it in the TLB. If there is no entry, then this is called a *TLB miss* and a translation table walk must be performed. The results of this walk are typically then cached in the TLB, so future translations of the same address can directly grab the physical address, memory attributes, and permissions, without needing to do another translation table walk. This process and the various microarchitectural structures are explored more in §8.3.1.

If there is an entry, this is referred to as a *TLB hit*. In this case, the result can be taken directly from the TLB.

Under normal circumstances, the TLB is invisible to userspace programs. However, systems code is expected to manage the TLBs explicitly, using a set of instructions which Arm provide specifically for this purpose: the family of `TLBI` TLB-maintenance instructions. When context switching, the systems software must manually manage the TLB, invalidating stale entries for old mappings out of the cache. The behaviours that arise from reading from potentially stale TLB entries are explored in detail in §8.5.

Address space identifiers TLB misses and TLB maintenance are both expensive operations, and so to reduce the burden, Arm provide a mechanism to permit multiple processes' address spaces to be loaded into the TLB at the same time, by allowing the software to mark each address space with a numeric label. Arm call these *address space identifiers* (or ASIDs).

Entries in the TLB are tagged with the current ASID, and so only that process will see entries in the TLB with that ASID.

The current ASID is encoded in the high order bits of the current `TTBR`. During a context switch, the system software needs only switch to the new translation tables for the new address space of the other process, without doing TLB maintenance, so long as it ensures the ASIDs are distinct.

There are only finitely many ASIDs available (typically it is an 8-bit field), and so eventually TLB maintenance is required to re-use a previously allocated ASID for a new address space. But this happens far less frequently than the context switches themselves. The provided TLB maintenance instructions can target specific ASIDs, avoiding the need to over-invalidate other cached address space translations, preventing a cascade of TLB misses in other processes, further improving the runtime performance for a small amount of additional effort on the software side.

VMIDs Address space identifiers are used only for stage 1 translations. Stage 2 has virtual machine identifiers (VMIDs).

As before, the current VMID is encoded in the `VTTBR_EL2` register, and the TLB entries are additionally tagged with the current VMID (as well as the ASID), and a translation will only use TLB entries that match the current ASID and VMID.

TLB maintenace instructions Arm define a whole family of instructions under the `TLBI` mnemonic.

The format for a `TLBI` instruction is a product of fields:

```
TLBI <type><level><broadcast>{,<reg>}

<type> =
  ALL | VMALL | ASID | VA{A|L} | IPAS2
<level> =
  E1 | E2
<broadcast> =
  {IS}
<reg> =
  X0 | X1 | ... | X30
```

Again, see the full description in the Arm manual for a more complete description [1, D5-4915].

The most common, and the ones that will be discussed in the following chapters, are as follows:

- `TLBI VAE1,Xn`: Invalidate this CPU's cached copies of entries used to translate the virtual address in register `Xn`, for the EL1&0 regime, for the current ASID and VMID.
- `TLBI VALE1,Xn`: Invalidate this CPU's cached copies of any last-level entries used to translate the virtual address in register `Xn`, for the EL1&0 regime, for the current ASID and VMID.
- `TLBI VAAE1,Xn`: Invalidate this CPU's cached copies of any last-level entries used to translate the virtual address in register `Xn`, for the EL1&0 regime, for the current VMID, for any ASID.
- `TLBI VAE1IS,Xn`: Invalidate all CPU's cached copies of entries used to translate the virtual address in register `Xn`, for the EL1&0 regime, for the current ASID and VMID.
  (...and equivalent `TLBI VAE2`, `TLBI VALE2`, `TLBI VAE2IS` instructions for virtual addresses in the EL2 regime)
- `TLBI IPAS2E1,Xn`: Invalidate this CPU's cached copies of entries used to translate the intermediate physical address in register `Xn`, for the EL1&0 regime, for the current VMID.
- `TLBI IPAS2LE1,Xn`: Invalidate this CPU's cached copies of any last-level entries used to translate the intermediate physical address in register `Xn`, for the EL1&0 regime, for the current VMID.
- `TLBI IPAS2E1IS,Xn`: Invalidate all CPU's cached copies of entries used to translate the intermediate physical address in register `Xn`, for the EL1&0 regime, for the current VMID.
- `TLBI VMALLE1`: Invalidate this CPU's cached copies of entries for the EL1&0 regime, for the current VMID.
- `TLBI VMALLE1IS`: Invalidate all CPU's cached copies of entries for the EL1&0 regime, for the current VMID.
- `TLBI ALLE1`: Invalidate this CPU's cached copies of entries for the EL1&0 regime, for any ASID or VMID.
- `TLBI ALLE1IS`: Invalidate all CPU's cached copies of entries for the EL1&0 regime, for any ASID or VMID.
  (...and equivalent `TLBI ALLE2`, and `TLBI ALLE2IS` instructions for the EL2 regime)
- `TLBI ASIDE1,Xn`: Invalidate this CPU's cached copies of entries for the EL1&0 regime, for the ASID specified in register `Xn`.
- `TLBI ASIDE1IS,Xn`: Invalidate this CPU's cached copies of entries for the EL1&0 regime, for the ASID specified in register `Xn`.
  (Note that the EL2 regime does not have ASIDs)

7.8 Arm ASL Reference

Here I include the actual Arm ASL for the various parts of the translation machinery. This listing contains a verbatim subset of the ASL pseudocode for the translation table walk.

The sources have per-function line numbers and are annotated to direct the reader to those parts highlighted in §7.6.3. Lines which handle out-of-scope features (access flags, dirty bits, shareability domains, debugging, realms, secure states, atomics) are greyed out. Key lines have coloured annotations.

7.8.1 AArch64.TranslateAddress

```
1  AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccType acctype, boolean
       iswrite, boolean aligned, integer size)
2    result = AArch64.FullTranslate(va, acctype, iswrite, aligned);← Do the translation
3
4    if !IsFault(result) && acctype != AccType_IFETCH then
5      result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);
6
7    if HaveRME() && !IsFault(result) && (acctype != AccType_DC ||
8      boolean IMPLEMENTATION_DEFINED "GPC Fault on DC operations") then
9      accdesc = CreateAccessDescriptor(acctype);
10     result.fault.gpcf = GranuleProtectionCheck(result, accdesc);
11
12     if result.fault.gpcf.gpf != GPCF_None then
13       result.fault.statuscode = Fault_GPCFOnOutput;
14       result.fault.paddress   = result.paddress;
15       result.fault.acctype    = acctype;
16       result.fault.write      = iswrite;
17
18   if !IsFault(result) && acctype == AccType_IFETCH then
19     result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);
20
21   // Update virtual address for abort functions
22   result.vaddress = ZeroExtend(va);
23
24   return result;
```

7.8.2 AArch64.FullTranslate

```
AddressDescriptor AArch64.FullTranslate(bits(64) va, AccType acctype, boolean
    iswrite, boolean aligned)

  fault = NoFault();
  fault.acctype = acctype;
  fault.write   = iswrite;

  ispriv = PSTATE.EL != EL0 && !(acctype IN {AccType_UNPRIV, AccType_UNPRIVSTREAM});
  regime = TranslationRegime(PSTATE.EL, acctype);
  ss = SecurityStateAtEL(PSTATE.EL);

  AddressDescriptor ipa;
  (fault, ipa) = AArch64.S1Translate(fault, regime, ss, va, acctype, aligned,
    iswrite, ispriv);

  if fault.statuscode != Fault_None then
    return CreateFaultyAddressDescriptor(va, fault);

  assert (ss == SS_Realm) IMPLIES EL2Enabled();
  if regime == Regime_EL10 && EL2Enabled() then
    s1aarch64 = TRUE;
    s2fs1walk = FALSE;
    AddressDescriptor pa;
    (fault, pa) = AArch64.S2Translate(fault, ipa, s1aarch64, ss, s2fs1walk, acctype,
     aligned, iswrite, ispriv);

    if fault.statuscode != Fault_None then
      return CreateFaultyAddressDescriptor(va, fault);
    else
      return pa;
  else
    return ipa;
```

Do the first stage of translation

Check for stage 1 translation fault

Do the second stage of translation

Check for stage 2 translation fault

7.8.3 AArch64.S1Translate

```
(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime regime, SecurityState ss, bits(64) va, AccType acctype, boolean aligned_in, boolean iswrite_in, boolean ispriv)
  FaultRecord fault = fault_in;
  boolean aligned = aligned_in;
  boolean iswrite = iswrite_in;
  // Prepare fault fields in case a fault is detected
  fault.secondstage = FALSE;
  fault.s2fs1walk   = FALSE;

  if !AArch64.S1Enabled(regime) then
    return AArch64.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);

  walkparams = AArch64.GetS1TTWParams(regime, va);

  if (AArch64.S1InvalidTxSZ(walkparams) ||
      (!ispriv && walkparams.e0pd == '1') ||
      (!ispriv && walkparams.nfd == '1' && IsDataAccess(acctype) && TSTATE.depth > 0) ||
      (!ispriv && walkparams.nfd == '1' && acctype == AccType_NONFAULT) ||
      )AArch64.VAIsOutOfRange(va, acctype, regime, walkparams)) then
    fault.statuscode = Fault_Translation;
    fault.level    = 0;
    return (fault, AddressDescriptor UNKNOWN);

  AddressDescriptor descaddress;
  TTWState walkstate;
  bits(64) descriptor;
  bits(64) new_desc;
  bits(64) mem_desc;
  repeat
    (fault, descaddress, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va, regime, ss, acctype, iswrite, ispriv);

    if fault.statuscode != Fault_None then
      return (fault, AddressDescriptor UNKNOWN);

    if acctype == AccType_IFETCH then
      // Flag the fetched instruction is from a guarded page
      SetInGuardedPage(walkstate.guardedpage == '1');

    if AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmd, walkstate.mematttrs) then
      fault.statuscode = Fault_Alignment;
    elsif IsAtomicRW(acctype) then
      if AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams, ispriv, acctype, FALSE) then
        // The Permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write    = FALSE;
      elsif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams, ispriv, acctype, TRUE) then
        // The Permission fault was caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write    = TRUE;
    elsif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams, ispriv, acctype, iswrite) then
      fault.statuscode = Fault_Permission;

    new_desc = descriptor;
    if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
      // Set descriptor AF bit
      new_desc<10> = '1';
```

Check VA is valid

Do the translation table walk

Check for S1 translation fault

Check for permission fault

```
57      // If HW update of dirty bit is enabled, the walk state permissions
58      // will already reflect a configuration permitting writes.
59      // The update of the descriptor occurs only if the descriptor bits in
60      // memory do not reflect that and the access instigates a write.
61      if (fault.statuscode == Fault_None &&
62          walkparams.ha  == '1' &&
63          walkparams.hd  == '1' &&
64          descriptor<51> == '1' && // Descriptor DBM bit
65          (IsAtomicRW(acctype) || iswrite) &&
66          !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
67        // Clear descriptor AP[2] bit permitting stage 1 writes
68        new_desc<7> = '0';
69
70      AddressDescriptor descupdateaddress;
71      FaultRecord s2fault;
72      // Either the access flag was clear or AP<2> is set
73      if new_desc != descriptor then
74        if regime == Regime_EL10 && EL2Enabled() then
75          s1aarch64 = TRUE;
76          s2fs1walk = TRUE;
77          aligned   = TRUE;
78          iswrite   = TRUE;
79          (s2fault, descupdateaddress) = AArch64.S2Translate(fault, descaddress,
      s1aarch64, ss, s2fs1walk, AccType_ATOMICRW, aligned, iswrite, ispriv);
80
81          if s2fault.statuscode != Fault_None then
82            return (s2fault, AddressDescriptor UNKNOWN);
83        else
84          descupdateaddress = descaddress;
85
86        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
      walkparams.ee, descupdateaddress);
87
88    until new_desc == descriptor || mem_desc == new_desc;
89
90    if fault.statuscode != Fault_None then
91      return (fault, AddressDescriptor UNKNOWN);
92
93    // Output Address
94    oa = StageOA(va, walkparams.tgx, walkstate);<-- Compute IPA
95    MemoryAttributes mematTrs;
96    if (acctype == AccType_IFETCH &&
97      (walkstate.mematTrs.memtype == MemType_Device || !AArch64.S1ICacheEnabled(regime
      ))) then
98      // Treat memory attributes as Normal Non-Cacheable
99      mematTrs = NormalNCMemAttr();
100     mematTrs.xs = walkstate.mematTrs.xs;
101   elsif (acctype != AccType_IFETCH && !AArch64.S1DCacheEnabled(regime) &&
102        walkstate.mematTrs.memtype == MemType_Normal) then
103     // Treat memory attributes as Normal Non-Cacheable
104     mematTrs = NormalNCMemAttr();
105     mematTrs.xs = walkstate.mematTrs.xs;
106
107     // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
108     // on the Tagged attribute
109     if HaveMTE2Ext() && walkstate.mematTrs.tagged then
110       mematTrs.tagged = ConstrainUnpredictableBool(Unpredictable_S1CTAGGED);
111   else
112     mematTrs = walkstate.mematTrs;
113
114   // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION
      DEFINED
115   // to be either effective value or descriptor value
116   if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
117       !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1"))
      then
```

```
118       memattrs.shareability = walkstate.memattrs.shareability;
119     else
120       memattrs.shareability = EffectiveShareability(memattrs);
121
122     if acctype == AccType_ATOMICLS64 && memattrs.memtype == MemType_Normal then
123       if memattrs.inner.attrs != MemAttr_NC || memattrs.outer.attrs != MemAttr_NC then
124         fault.statuscode = Fault_Exclusive;
125         return (fault, AddressDescriptor UNKNOWN);
126
127     ipa = CreateAddressDescriptor(va, oa, memattrs);
128     return (fault, ipa);
```

Return IPA and Memory Attributes

7.8.4 AArch64.S1Walk

```
(FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S1Walk(FaultRecord fault_in, S1TTWParams walkparams, bits(64) va, Regime regime, SecurityState ss, AccType acctype, boolean iswrite_in, boolean ispriv)
  FaultRecord fault = fault_in;
  boolean iswrite = iswrite_in;
  if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
    fault.statuscode = Fault_Translation;
    fault.level     = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

  walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, ss);   <-- read TTBR

  // Detect Address Size Fault by TTB
  if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
    fault.statuscode = Fault_AddressSize;
    fault.level     = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

  bits(64) descriptor;
  AddressDescriptor walkaddress;

  walkaddress.vaddress = va;
  if !AArch64.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
  else
    walkaddress.memattrs = walkstate.memattrs;

  // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
  // to be either effective value or descriptor value
  if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
      !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
  else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

  DescriptorType desctype;
  repeat   <-- For each level in {0,1,2,3}
    fault.level = walkstate.level;
    FullAddress descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx, walkparams.txsz, va, walkstate.baseaddress);
                                                          <-- Get IPA of entry to read
    walkaddress.paddress = descaddress;

    if regime == Regime_EL10 && EL2Enabled() then
      s1aarch64 = TRUE;
      s2fs1walk = TRUE;                                   <-- Do S2 translation to get the PA of the entry
      aligned   = TRUE;
      iswrite   = FALSE;
      (s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaddress, s1aarch64, ss, s2fs1walk, AccType_TTW, aligned, iswrite, ispriv);

      if s2fault.statuscode != Fault_None then   <-- Check for S2 fault
        return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

      (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault);
    else                                  <-- Read memory to get descriptor
      (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

    if fault.statuscode != Fault_None then   <-- Check for external abort
      return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
```

```
59      desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.
        tgx, walkstate.level);
60
61      case desctype of
62        when DescriptorType_Table
63          walkstate = AArch64.S1NextWalkStateTable(walkstate, regime, walkparams,
        descriptor);                                          Extract next level table address
64
65          // Detect Address Size Fault by table descriptor
66          if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
67            fault.statuscode = Fault_AddressSize;
68            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64)
        UNKNOWN);
69
70        when DescriptorType_Page, DescriptorType_Block
71          walkstate = AArch64.S1NextWalkStateLast(walkstate, regime, ss, walkparams,
        descriptor);                                          Extract page start address
72
73        when DescriptorType_Invalid
74          fault.statuscode = Fault_Translation;
75          return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN
        );
                    Return fault if invalid
76
77        otherwise
78          Unreachable();
79
80    until desctype IN {DescriptorType_Page, DescriptorType_Block};
81
82    if (walkstate.contiguous == '1' &&
83        AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level))
       then
84      fault.statuscode = Fault_Translation;
85    elsif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
86      fault.statuscode = Fault_Translation;
87    // Detect Address Size Fault by final output
88    elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
89      fault.statuscode = Fault_AddressSize;
90    // Check descriptor AF bit
91    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
92        !(acctype IN {AccType_DC, AccType_IC} &&
93        !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC
        operations")) then
94      fault.statuscode = Fault_AccessFlag;
95
96    return (fault, walkaddress, walkstate, descriptor);
```

7.8.5 AArch64.S2Translate

```
(FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault_in,
    AddressDescriptor ipa, boolean s1aarch64, SecurityState ss, boolean s2fs1walk,
    AccType acctype, boolean aligned, boolean iswrite, boolean ispriv)
  walkparams = AArch64.GetS2TTWParams(ss, ipa.paddress.paspace, s1aarch64);
  FaultRecord fault = fault_in;

  // Prepare fault fields in case a fault is detected
  fault.statuscode  = Fault_None; // Ignore any faults from stage 1
  fault.secondstage = TRUE;
  fault.s2fs1walk   = s2fs1walk;
  fault.ipaddress   = ipa.paddress;

  if walkparams.vm != '1' then                          <-- Check if in a two-stage regime
    // Stage 2 translation is disabled
    return (fault, ipa);

  if (AArch64.S2InvalidTxSZ(walkparams, s1aarch64) ||
      AArch64.S2InvalidSL(walkparams) ||
      AArch64.S2InconsistentSL(walkparams) ||
      AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams)) then
    fault.statuscode = Fault_Translation;
    fault.level    = 0;
    return (fault, AddressDescriptor UNKNOWN);

  AddressDescriptor descaddress;
  TTWState walkstate;
  bits(64) descriptor;
  bits(64) new_desc;
  bits(64) mem_desc;
  repeat
    (fault, descaddress, walkstate, descriptor) = AArch64.S2Walk(fault, ipa,
    walkparams, ss, acctype, iswrite, s1aarch64);                  <-- Do translation table walk

    if fault.statuscode != Fault_None then              <-- Check for stage 2 translation fault
      return (fault, AddressDescriptor UNKNOWN);

    if AArch64.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
      fault.statuscode = Fault_Alignment;
    elsif IsAtomicRW(acctype) then
      if AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams, ispriv,
       acctype, FALSE) then
        // The Permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write    = FALSE;
      elsif AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams,
      ispriv, acctype, TRUE) then
        // The Permission fault was caused by lack of write permissions.
        // However, HW updates, which are atomic writes for stage 1
        // descriptors, permissions fault reflect the original access.
        fault.statuscode = Fault_Permission;
        if !fault.s2fs1walk then
          fault.write  = TRUE;
    elsif AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams, ispriv
    , acctype, iswrite) then                            <-- Check for stage 2 permission fault
      fault.statuscode = Fault_Permission;

    new_desc = descriptor;
    if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
      // Set descriptor AF bit
      new_desc<10> = '1';

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
```

```
    // memory do not reflect that and the access instigates a write.
    if (fault.statuscode == Fault_None &&
        walkparams.ha  == '1' &&
        walkparams.hd  == '1' &&
        descriptor<51> == '1' && // Descriptor DBM bit
        (IsAtomicRW(acctype) || iswrite) &&
        !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
      // Set descriptor S2AP[1] bit permitting stage 2 writes
      new_desc<7> = '1';

    // Either the access flag was clear or S2AP<1> is clear
    if new_desc != descriptor then
      (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
  walkparams.ee, descaddress);

  until new_desc == descriptor || mem_desc == new_desc;

  if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

  ipa_64 = ZeroExtend(ipa.paddress.address, 64);
  // Output Address
  oa = StageOA(ipa_64, walkparams.tgx, walkstate);<-- Compute final PA
  MemoryAttributes s2_mematTrs;
  if ((s2fs1walk &&
      walkstate.mematTrs.memtype == MemType_Device && walkparams.ptw == '0') ||
    (acctype == AccType_IFETCH &&
      (walkstate.mematTrs.memtype == MemType_Device || HCR_EL2.ID == '1')) ||
    (acctype != AccType_IFETCH &&
       walkstate.mematTrs.memtype == MemType_Normal && HCR_EL2.CD == '1')) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_mematTrs = NormalNCMemAttr();
    s2_mematTrs.xs = walkstate.mematTrs.xs;
  else
    s2_mematTrs = walkstate.mematTrs;

  if !s2fs1walk && acctype == AccType_ATOMICLS64 && s2_mematTrs.memtype ==
    MemType_Normal then
    if s2_mematTrs.inner.attrs != MemAttr_NC || s2_mematTrs.outer.attrs !=
    MemAttr_NC then
      fault.statuscode = Fault_Exclusive;
      return (fault, AddressDescriptor UNKNOWN);

  MemoryAttributes mematTrs;
  if walkparams.fwb == '0' then
    mematTrs = S2CombineS1MemAttrs(ipa.mematTrs, s2_mematTrs);<-- Merge memory attributes
  else
    mematTrs = s2_mematTrs;

  pa = CreateAddressDescriptor(ipa.vaddress, oa, mematTrs);
  return (fault, pa);<-- Return PA and Memory Attributes
```

7.8.6 AArch64.S2Walk

```
(FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S2Walk(
    FaultRecord fault_in, AddressDescriptor ipa, S2TTWParams walkparams,
    SecurityState ss, AccType acctype, boolean iswrite, boolean s1aarch64)

  FaultRecord fault = fault_in;
  ipa_64 = ZeroExtend(ipa.paddress.address, 64);

  TTWState walkstate;
  if ss == SS_Secure then
    walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
  else
    walkstate = AArch64.S2InitialTTWState(ss, walkparams);          <-- read VTTBR

  // Detect Address Size Fault by TTB
  if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
    fault.statuscode = Fault_AddressSize;
    fault.level    = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

  bits(64) descriptor;
  AddressDescriptor walkaddress;

  walkaddress.vaddress = ipa.vaddress;
  if HCR_EL2.CD == '1' then
    walkaddress.mattrs = NormalNCMemAttr();
    walkaddress.mattrs.xs = walkstate.mattrs.xs;
  else
    walkaddress.mattrs = walkstate.mattrs;

  walkaddress.mattrs.shareability = EffectiveShareability(walkaddress.mattrs);

  DescriptorType desctype;
  repeat                                                            <-- For each level in {0,1,2,3}
    fault.level = walkstate.level;

    FullAddress descaddress;
    if walkstate.level == AArch64.S2StartLevel(walkparams) then
      // Initial lookup might index into concatenated tables
      descaddress = AArch64.S2SLTTEntryAddress(walkparams, ipa.paddress.address,
    walkstate.baseaddress);
    else
      ipa_64 = ZeroExtend(ipa.paddress.address, 64);
      descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx,
    walkparams.txsz, ipa_64, walkstate.baseaddress);
                                                                    <-- Get PA of entry to read
    walkaddress.paddress = descaddress;
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);
                                                                    <-- Read descriptor from memory
    if fault.statuscode != Fault_None then                          <-- Check for external abort
      return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.
    tgx, walkstate.level);

    case desctype of
      when DescriptorType_Table
        walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);
                                                                    <-- Extract next level table address
        // Detect Address Size Fault by table descriptor
        if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64)
    then
          fault.statuscode = Fault_AddressSize;
          return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64)
    UNKNOWN);

      when DescriptorType_Page, DescriptorType_Block
```

```
61          walkstate = AArch64.S2NextWalkStateLast(walkstate, ss, walkparams, ipa,
      descriptor);                                          Extract page start address
62
63        when DescriptorType_Invalid
64          fault.statuscode = Fault_Translation;
65          return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN
      );                    Return fault if invalid
66
67        otherwise
68          Unreachable();
69
70    until desctype IN {DescriptorType_Page, DescriptorType_Block};
71
72    if (walkstate.contiguous == '1' &&
73        AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level))
        then
74      fault.statuscode = Fault_Translation;
75    elsif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
76      fault.statuscode = Fault_Translation;
77    // Detect Address Size Fault by final output
78    elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
79      fault.statuscode = Fault_AddressSize;     Check output address is within bounds
80    // Check descriptor AF bit
81    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
82        !(acctype IN {AccType_DC, AccType_IC} &&
83        !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC
      operations")) then
84      fault.statuscode = Fault_AccessFlag;
85
86    return (fault, walkaddress, walkstate, descriptor);
```

7.8.7 AArch64.FetchDescriptor

```
(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress,
    FaultRecord fault_in)
  // 32-bit descriptors for AArch32 Short-descriptor format
  // 64-bit descriptors for AArch64
  //or AArch32 Long-descriptor format
  assert N == 32 || N == 64;
  bits(N) descriptor;
  FaultRecord fault = fault_in;
  AccessDescriptor walkacc;

  walkacc.acctype = AccType_TTW;
  // MPAM PARTID for translation table walk is determined by the access invoking the
     translation
  walkacc.mpam  = GenMPAMcurEL(fault.acctype);

  if HaveRME() then
    fault.gpcf = GranuleProtectionCheck(walkaddress, walkacc);
    if fault.gpcf.gpf != GPCF_None then
      fault.statuscode = Fault_GPCFOnWalk;
      fault.paddress   = walkaddress.paddress;
      fault.gpcfs2walk = fault.secondstage;
      return (fault, bits(N) UNKNOWN);

  PhysMemRetStatus memstatus;
  (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkacc);
  if IsFault(memstatus) then
    fault = HandleExternalTTWAbort(memstatus, fault.write, walkaddress, walkacc, N
    DIV 8, fault);
    if IsFault(fault.statuscode) then
      return (fault, bits(N) UNKNOWN);

  if ee == '1' then
    descriptor = BigEndianReverse(descriptor);

  return (fault, descriptor);
```

Chapter 8

Relaxed virtual memory

This chapter is based, in part, on: Relaxed virtual memory in Armv8-A [54] by Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Published in the proceedings of the 31st European Symposium on Programming (ESOP, 2022).

Now we will introduce the main concurrency architecture design questions that arise for virtual memory in Arm. As usual, the architecture defines the *envelope* of behaviours which hardware must guarantee and on which software may rely. This envelope must be tight enough to give the guarantees software needs to function, but still loose enough to admit the range of existing and conceivable microarchitectures whose optimization techniques are necessary for performance.

This chapter therefore will discuss both the relevant microarchitecture as we understand it, and also the behaviours which it is believed software relies upon. The discussion will touch on points of several kinds: some which are clear in the current Arm prose documentation; some where Arm are in the process of architecting a change; some that are not documented but where the semantics is (perhaps, after discussion with Arm) clear or constrained by current hardware or software practice; and, some where their modelling raised questions for which the architecture is not yet well-defined, and Arm must make an architectural decision.

Ideally, we would be able to specify which points belong to which kind. It is, however, not so easy. There is no clean separation between aspects there are clearly defined in the architecture reference, and those that are not; instead, the manual has a shallow covering of many of the behaviours described here. In other places, the reference may have been updated or changed over the course of the work, clarifying parts of the architecture, and while this may have happened concurrently with discussing those and other points with Arm, the reference text itself is solely the responsibility of Arm. In §8.9 we will return to this question, and more directly address the kinds of each point discussed.

Chapter overview The body of this chapter will explore a sequence of key behaviours, some of which the architecture guarantee and some that it does not. Each contains a description of the behaviour, including whether software relies on it or known hardware guarantees it; a short discussion of the architectural intent as we understand it; and any associated litmus tests.

This chapter will discuss a variety of interesting behaviours. In an attempt to make this chapter more approachable, it is broken down into a logical progression: slowly building up from the most simple and fundamental parts of the architecture, to increasingly more complex cases.

We will first discuss (in §8.2) how translation affects the prior 'data memory' **TODO: PS: user-mode?** tests covered in previous work. Then, we shall see how the caching of translation entries is limited (§??) and the fundamental behaviours of the translation table walk (§8.4). Building upon that, we will see that these translation table walks may be cached and re-used in later translations, which is explored in detail in §8.5. Then (in §8.6), we will explore how the various kinds of TLB maintenance interact with those cached translations, and other translation table walks. Finally, we touch on how all of the above fit together with system registers and other context changing and synchronising operations in §8.7.

Chapter Contents

8.1 Virtual memory litmus tests

As previously discussed, one fundamental idea to come out of the field of relaxed memory is the concept of litmus tests. Virtual memory is no different, and exploring the architectural intent is best done through the creation, discussion and evaluation of small programs which are representative examples of common patterns.

However, as we explore more of the system semantics more and more of the system state plays an integral role in the behaviours we see. For this reason we need a new language for describing the state of the system, with features not supported by the language supported by the previous litmus, rmem, herd, and diy tools [17, 28, 14, 21], in particular, the translation table state.

The litmus tests here are given in the `isla-axiomatic` test format. I describe the isla tool itself, and the extended test format syntax, in more detail in **TODO: ?REF?**.

A virtual memory litmus test To illustrate this isla test format, Figure 8.1 contains the test listing for a non-trivial virtual memory litmus test called **CoW** (or "Copy-on-Write").

This test is derived from sequence of operations the Linux kernel takes when performing copy-on-write. Thread 0 tries to write to a location (call it x) that is currently read-only (line 1 in the thread code), then when the fault is taken the Linux exception handler begins executing (line 1 in the handler), Linux performs some checks that it's okay to copy and that it hasn't already done so (not part of the test), and then copies the physical page (lines 3 and 4 in the handler, although the test here only copies one value as demonstration), before flushing the data caches (line 5) so that later reads will be guaranteed to see the copied values. Then Linux needs to swap over the pagetable entry for x from a read-only view on the original page to a writeable mapping on the freshly copied page. It does this by first 'breaking' the entry, making it invalid (line 7), then performing the necessary TLB maintenance (line 9), before writing a new mapping to the new page (line 11). Now, Linux can return from the handler (line 13) and re-try the store instruction, hopefully this time successfully writing to the new page.

The test format is split into 4 main parts:

- The initial state, comprised of:
  - the per-thread register state.
  - the global memory and pagetable state.
- The thread code and any exception-handler code.
- The interesting final state, as a predicate over the final register and memory state.
- And, optionally, whether the outcome is allowed or forbidden by the model.

AArch64 CoW

Initial State

```
0:R0=0x2
0:R1=x
0:R3=y
0:R4=z
0:R5=z
0:R6=0b0
0:R7=pte3(x)
0:R8=page(x)
0:R9=mkdesc3(oa=pa2)
0:R10=pte3(x)
0:R20=0b0
0:VBAR_EL1=0x1000
0:PSTATE.EL=0b00
```

```
virtual x y z;
physical pa1 pa2;
x ↦ pa1 with [AP = 0b11] and default;
x ↦ invalid;
x ↦ pa2 with [AP = 0b01] and default;
y ↦ pa1;
z ↦ pa2;
identity 0x1000 with code;
*pa1 = 1;
*pa2 = 0;
```

Thread 0

```
01. STR X0,[X1]
```

Thread 0 EL1 Handler

```
01. 0x1400:
02.     CBNZ X20,exit
03.     LDR X2,[X3]
04.     STR X2,[X4]
05.     DC CIVAC,X5
06.     DSB SY
07.     STR X6,[X7]
08.     DSB SY
09.     TLBI VALE1IS,X8
10.     DSB SY
11.     STR X9,[X10]
12.     MOV X20,#1
13.     ERET
14. exit:
15.     MRS X21,ELR_EL1
16.     ADD X21,X21,#4
17.     MSR ELR_EL1,X21
18.     ERET
```

Final State

```
pa1=1 & pa2=2
```

Allow

Figure 8.1: Test CoW: code listing

Initial state The initial state has three virtual addresses (x, y and z), and two physical addresses (pa1 and pa2). Initial register values are written like `0:R4=z`, meaning register R4 on Thread 0 initially contains the value z (in this case, a virtual address). Helper functions like `pte3`, `page` and `mkdesc3` are used to get the address of the leaf entry, the page offset and to create a new valid descriptor with the given OA, a more detailed description of the functions are given later.

Behind the scenes, isla creates a full instantiation of the Arm translation tables, but with some holes for symbolic values where the test may modify the tables. There is a default translation table, where the code and the tables themselves are mapped by default and everything else is invalid.

The pagetable setup is then defined in a small DSL which defines a delta to that default table, specifying that certain pages should be mapped or unmapped initially, as well as being able to specify the set of locations and

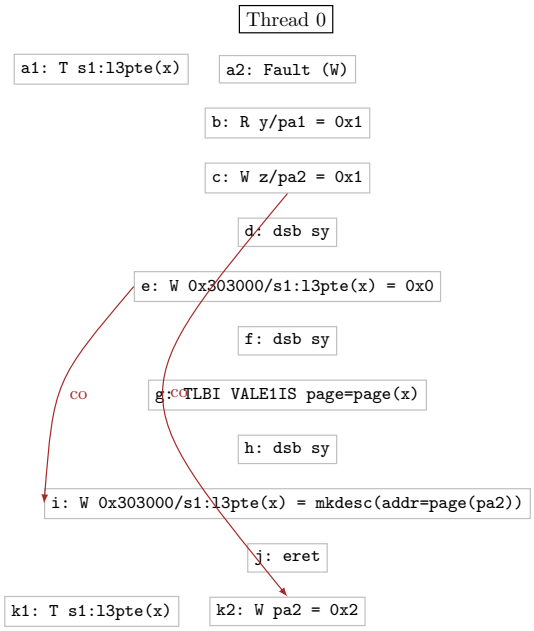


Figure 8.2: Test CoW: execution diagram

their initial memory values the test will need.

Fundamentally we categorise those locations as either virtual, intermediate, or physical. The line `virtual x y z` in the CoW test allocates 3 virtual contiguous pages, and labels their page-aligned addresses as `x`, `y`, and `z`. It then allocates two physical pages with addresses `pa1` and `pa2`. Next, the setup defines the initial value of the translation tables, as well as specifying the set of potential translation tables that may be in use by the test (for isla to create symbolic 'holes' for those). Namely, the initial state starts with `x` mapped to `pa1` with the access permissions bits set to `0b11` (read-only). The next two lines tell isla that during the test `x` may become unmapped (the descriptor may be invalid), or mapped to `pa2` with `AP=0b01`. The test also defines two other variables, `y` and `z` as aliases to the two physical pages, to help with copying the data between them, just as Linux would. Since there is an exception handler in this test, we need to ensure that the code page of the handler is mapped executable at EL1, which is what the `identity 0x1000 with code` line does (note that the handler section starts within the `0x1000` page). Finally, we say that the initial values of `pa1` and `pa2` are `1` and `0` respectively.

Register translation helpers The initial register state can reference parts of the initial state related to pagetables through the use of helper functions. Here are the helpers used by CoW, and most of the tests in this section. The full description of this format is given in **TODO: ?REF?**if more information is needed.

- `pte<N>(va)`: The (intermediate) physical address of the level N entry in the default translation tables that maps `va`.
- `desc<N>(va)`: The 64-bit descriptor from the initial state of the level N entry that maps `va` (the value of `pte<N>(va)` in the initial state).
- `page(va)`: The page number that `va` is in (equivalently: `va` $\gg$ `12`).
- `mkdesc<N>(oa=pa)`: A fresh 64-bit descriptor for a valid leaf entry at level N where the output address is given by the `oa` parameter.
- `mkdesc<N>(table=pa)`: A fresh 64-bit descriptor for a valid table entry at level N where the next-level-table address is given by the `table` parameter.

Entries listed as `f<N>` mean a family of functions `f1`, `f2`, `f3` and so on.

Execution diagrams Figure 8.2 is the isla-generated execution diagram for the CoW test. It illustrates a candidate execution which isla found (with any symbolic holes filled with concrete values) which matched the final state of the execution, and was consistent with the axioms of the model (given in Chapter 9).

The execution is rendered as a diagram, with separate traces for each thread, with multiple columns per thread, for translations and explicit events. In the diagram, there is one thread (Thread 0), and all events belong to

its trace. There are two columns; the right-hand side are the explicit events rendered in program-order, and the left-hand side contains translation events alongside any explicit events from the same instruction. Not all events from the trace are displayed in the execution diagram; many uninteresting events, of register reads and writes, and translation reads of unchanged entries, are suppressed. The execution displayed here is one where the initial store's translation table walk (event a1) reads an valid entry from the initial state but which did not have permissions to do a write, and so generates a Fault event (a2). The execution continues, copying the memory over to a new page (events b-c), before updating the translation tables to point to the new page (d-h, see §8.6.5), before returning from the exception handler (j) and re-trying the store which succeeds in writing to the new page (k2), giving a final state consistent with the expected final state from the test listing in Figure 8.1.

In general, while there could be multiple executions that correspond to the final execution, the tests are usually written in a way to ensure that there is only one consistent candidate execution which corresponds to the final state. In cases where the test is forbidden by the model, we still have isla induce a concrete candidate, and render a diagram of the interesting forbidden execution.

8.2 Aliased data memory

Much of the previous work on relaxed memory has been concerned with what we shall call 'data memory': the weak behaviour of concurrent loads and stores to memory. For Arm, we shall see that these previous models were implicitly assuming that all locations in the test were virtual addresses, with well-formed, constant, and injective, address translation mappings, which mapped all locations as readable, writable, and executable, normal cacheable memory.

Consider a non-injective mapping. Such mappings give rise to *aliasing*: the situation where two distinct virtual addresses in the same address space map to the same output physical address. This section will explore how the behaviours of those data memory tests change in the presence of aliasing.

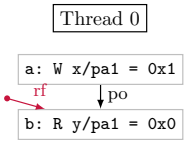
8.2.1 Virtual coherence

For data memory accesses, one of the most fundamental guarantee that architectures provide is *coherence*: in any execution, for each memory location, there is a total order of the accesses to that location, consistent with the program order of each thread, with reads reading from the most recent write in that order. Hardware implementations provide this, despite their elaborate cache hierarchies and out-of-order pipelines, by a combination of coherent cache protocols and pipeline hazard checking, identifying and restarting instructions when possible coherence violations are detected.

For Arm, coherence is with respect to physical addresses [1, B2.3.1 (p157)] [1, D5.11.1 (p4931)] . This means that if two virtual addresses alias to the same physical address, then:

- a load from one virtual address cannot ignore a program-order previous store to the other, as seen in the following CoWR.alias test [Figure 8.3]:

AArch64	CoWR.alias
Initial State	
0:R0=0x1 0:R1=x 0:R3=y	
physical pa1; x |-> pa1; y |-> pa1; *pa1 = 0;	
Thread 0	
STR X0,[X1] **LDR** X2,[X3]	
Final State	
0:X2=0	
Forbid	

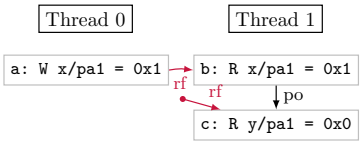


This test is a variation on the standard CoWR test, where the VA is replaced with two distinct VAs, which both alias to the same PA.
The initial state is a configuration with two virtual addresses, `x` and `y`, which are both mapped to the physical address `pa1`, whose initial value is `0`. The thread then stores `1` to `x`, then loads `y`. It is then forbidden for this load to read `0`.
While the Armv8-A architecture reference manual describes data caches as being physically-indexed [1, D5.11.1 (p4931)] and so accesses via the same PA are 'fully coherent', further discussions with Arm clarify that this implies not just this coherence test, but that all prior data memory behaviours previously examined still apply when subjected to aliasing.

Figure 8.3: CoWR.alias test

- ▷ a load from one virtual address cannot ignore the write that a program-order previous load of the other address saw (CoRR0.alias+po [Figure 8.4], CoRR2.alias+po [Figure 8.5]).
- ▷ a load from one virtual address can have its value forwarded from a store to the other, and similarly on a speculative branch (MP.alias3+rfi-data+dmb [Figure 8.6], PPOCA.alias [Figure 8.6]).

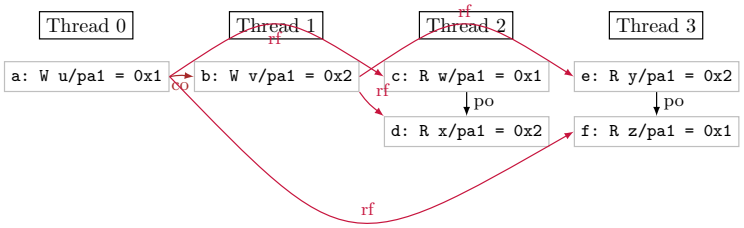
AArch64	CoRR0.alias+po
Initial State	
0:R0=0b1 0:R1=x	1:R1=x 1:R3=y 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1; x |-> pa1; y |-> pa1; *pa1 = 0;	
Thread 0	Thread 1
STR X0,[X1]	**LDR** X0,[X1] **LDR** X2,[X3]
Final State	
1:X0=1 & 1:X2=0	
Forbid	



This test is a variation of the data memory CoRR0 test, where one of the loads has been replaced with a load of a distinct virtual address which aliases to the same underlying physical address.
Note that, like the original test, it is forbidden to read from the initial state in the later load, as this would violate coherence: exactly what the earlier text from the manual explicitly forbade.

Figure 8.4: CoRR0.alias+po test

AArch64			CoRR2.alias+po
Initial State			
0:R0=0b01 0:R1=u	1:R0=0b10 1:R1=v	2:R1=w 2:R3=x 2:PSTATE.SP=0b0 2:PSTATE.EL=0b00	3:R1=y 3:R3=z 3:PSTATE.SP=0b0 3:PSTATE.EL=0b00
physical pa1; u |-> pa1; v |-> pa1; w |-> pa1; x |-> pa1; y |-> pa1; z |-> pa1; *pa1 = 0;			
Thread 0	Thread 1	Thread 2	Thread 3
STR X0,[X1]	**STR** X0,[X1]	**LDR** X0,[X1] **LDR** X2,[X3]	**LDR** X0,[X1] **LDR** X2,[X3]
Final State			
2:X0=1 & 2:X2=2 & 3:X0=2 & 3:X2=1			
Forbid			



This test is a variation of the data memory CoRR2 test. Here there are many options for adding aliasing, so we choose the maximally aliased version where each individual store and load uses a distinct virtual address but where all those virtual addresses alias to the same physical one.
This gives us a classic coherence shape, where it is forbidden for different threads to observe writes to the same physical location in different orders.

Figure 8.5: CoRR2.alias+po test

AArch64	MP.alias3+rfi-data+dmb
Initial State	
0:R0=0x1 0:R1=x 0:R3=z 0:R5=y	1:R1=y 1:R3=x

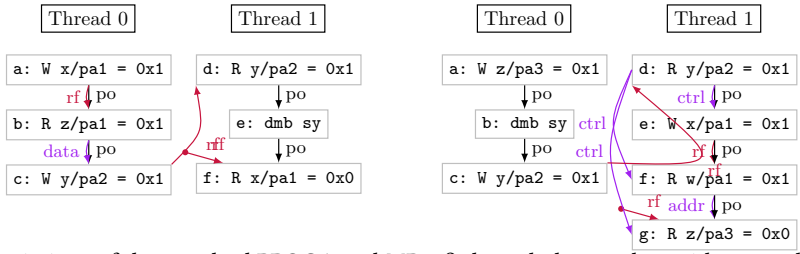
physical pa1 pa2;
x |-> pa1;
y |-> pa2;
z |-> pa1;
*pa1 = 0;
*pa2 = 0;	

Thread 0	Thread 1
STR X0,[X1] **LDR** X2,[X3] **STR** X2,[X5]	**LDR** X0,[X1] **DMB** SY **LDR** X2,[X3]
Final State	
1:X0=1 & 1:X2=0	
Allow	

AArch64	PPOCA.alias
Initial State	
0:R0=0x1 0:R1=z 0:R2=0x1 0:R3=y	1:R1=y 1:R2=0x1 1:R3=x 1:R5=w 1:R7=z

physical pa1 pa2 pa3;
w |-> pa1;
x |-> pa1;
y |-> pa2;
z |-> pa3;
*pa1 = 0;
*pa2 = 0;
*pa3 = 0;	

Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3]	**LDR** X0,[X1] **CBNZ** X0,L0 L0: **STR** X2,[X3] **LDR** X4,[X5] **EOR** X8,X4,X4 **LDR** X6,[X7,X8]
Final State	
1:X0=1 & 1:X4=1 & 1:X6=0	
Allow	



These tests are variations of the standard PPOCA and MP+rfi-data+dmb tests, but with some aliasing. Both are examples of *forwarding*: a thread-local read of a write before that write has been propagated to memory. These two tests, determined to be allowed architecturally from our discussions with Arm, show that the processor can forward from a write even if the read was for a different virtual address so long as the physical addresses match, even down a speculative path.

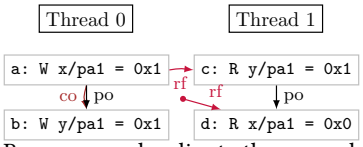
Figure 8.6: PPOCA.alias and MP.alias3+rfi-data+dmb tests.

8.2.2 Aliasing different locations

In the previous section, we explored taking tests over a single location, and rewriting the test to use many locations, which all alias to the same address. One can also take a test that has multiple locations and make some of them alias to the same address.

Multi-location data memory tests, which are architecturally allowed, may become forbidden in the presence of aliasing. For example, taking the traditional MP+pos test, when the two locations are aliased to the same physical address then we get the forbidden MP.alias+pos test [Figure 8.7]. This new test is, essentially, equivalent to the old CoRR0 test: coherence with two writes and two reads to the same location; just using different aliases.

AArch64	MP.alias+pos
Initial State	
0:R0=0x1 0:R1=x 0:R2=0x1 0:R3=y	1:R1=y 1:R3=x
physical pa1; x |-> pa1; y |-> pa1; *pa1 = 0;	
Thread 0	Thread 1
STR X0,[X1] **STR** X2,[X3]	**LDR** X0,[X1] **LDR** X2,[X3]
Final State	
1:X0=1 & 1:X2=0	
Forbid	



Because x and y alias to the same phsical address pa1, the two loads (c and d) read the same location, and so cannot read different writes out-of-order.

Figure 8.7: Test MP.alias+pos

8.2.3 Might be same (physical) address

There is a corner case that we now should consider. For load and store instructions, when the last register used in the calculation of the address is read, the address becomes known. This allows, in the flat model, for program-order later instructions to begin execution (or at least, know they will not be restarted) at that point.

With the introduction of address translation, however, this point happens much later, after the whole translation table walk is performed. Between the read of the register and the completion of the translation table walk, other instructions may perform some part of their functionality. This may include reading from a different virtual address, before the physical address of a program-order previous instruction is known, but after the virtual address is known.

One might expect that, when deciding whether to propagate a store, if the page offset of the virtual address is different to that of the in-flight program-order earlier instructions, then the write could go ahead early, knowing that the access could not be to the same physical address as any of those instructions. However, this is not the case. Although the accesses definitely will not access the same physical address, the program-order earlier access may still fault, meaning the write will not be reached. This means that writes must wait for program-order earlier translations to finish (or at least, be known to not fault) before they can be propagated to other threads.

8.3 What can be cached in TLBs

As was described in §7.7, Arm hardware can have TLBs, caching previously seen translations. But, there are some restrictions to this; both in what information a TLB must cache when it does so, but also in what kind of information it is not permitted to cache at all.

8.3.1 Microarchitectural TLBs

Here we must make a clear distinction between the actual *microarchitectural* translation caching one may encounter inspecting hardware, and the architectural model being discussed here.

While there are possibly many different ways to describe the same architectural intent, here we carefully choose one which will make building tooling, extending the model, discussions with architects, and explaining individual tests easier. We will first look at a specific example to pin down terminology and gain some intuition for hardware, before giving a model MMU and TLB that abstracts away from the details.

Microarchitectural MMU – A53 Let us explore more closely how the actual hardware fill and walk works on a modern microprocessor. The Arm Cortex A53 is an Arm-designed application class processor. Previous relaxed memory work included exercising this core design extensively during litmus testing validation of the models, finding it to be relaxed, exhibiting many relaxed behaviours, but not aggressively so. This makes the A53 a good candidate as a demonstrator of an average relaxed processor design. While other processors by Arm are more aggressive in their optimisations, the MMU and TLB layout of the A53 seems typical: other cores, such as the A57 **TODO: ?CITE?**, A72 **TODO: ?CITE?**, A76 **TODO: ?CITE?**, A78 **TODO: ?CITE?** and A715 **TODO: ?CITE?** all have comparable, or simpler, TLB configurations.

The Arm A53 Technical Reference Manual (TRM) describes, in detail, the structure of the Memory Management Unit [57, 5-2] of the A53, and its constituent parts. Figure 8.8 shows a hand-written block diagram representing the key information from the TRM.

We see that each core has its own MMU, and that each MMU contains a unit that will perform the translation table walk, in addition to a selection of translation caching structures:

- ▷ one instruction micro-TLB;
- ▷ one data micro-TLB;
- ▷ one unified TLB;
- ▷ one walk cache; and,
- ▷ one IPA cache.

The microarchitectural TLBs store whole translations: virtual to physical mappings, plus permissions and so-on, tagged with their context. The TLBs are arranged hierarchically. With small, 10-entry, ‘micro’ TLBs for instruction and data streams separately, and one large 512-entry unified TLB.

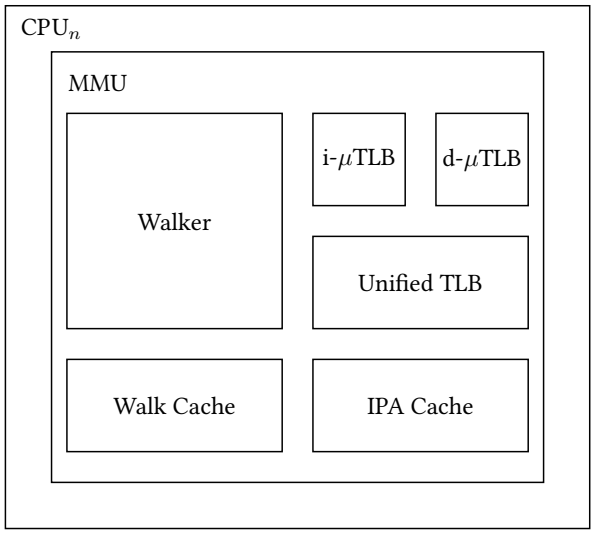


Figure 8.8: A53 Memory Management Block Diagram.

On a TLB miss, the MMU performs a translation table walk using the walker, computing the Arm translation table walk ASL code which we previously explored in §7.6.

When it begins this walk, the MMU first checks the walk cache for a matching entry. Walk cache entires are mappings from virtual address to the physical address of the last level translation table. If an entry is present the MMU can skip most of the walk entirely, performing just the very last read to read the leaf entry.

If a second stage of translation is required during the walk, the IPA cache is used (and may be, or not, used many times during the same walk). The IPA cache stores mappings from intermediate physical to physical memory — with no associated virtual address — which can be used during both the final stage 2 walk and any intermediate stage 2 walks during a stage 1 walk.

TODO: PS: walk cache s1 only? BS: that is one of thibaut's questions to RG

The MMU is free to save the result of any translation table walk into these structures, including for walks due to speculation, prefetching, or architectural execution. This, essentially, allows the MMU to perform a walk for any arbitrary VA or IPA, at any point in time.

8.3.2 Model MMU

To abstract away from any specific microarchitecture, we will model the MMU as if it were a separate asynchronous unit, one for each thread, each with an overapproximate 'TLB'.

Later, we will see tests that justify and ground this particular choice of abstraction, and we will explore this model and the mathematics which corresponds to it in more rigorous detail. But for now, we can imagine this model MMU as a set of (concurrently) executing translation table walks and a 'model TLB' cache of translation table entries.

Model TLB entries In general, the architecture permits hardware to cache whatever information from the translation process the hardware sees fit, this may include the output of whole translation table walks (complete virtual to physical mappings) or individual translation table entries, or even the result of partial walks (the address of the last-level table, for example).

It would not be feasible to even attempt to enumerate all the possible shapes of TLBs and the kinds of information they can cache. Instead, we will define a *model* TLB. This model will treat the TLB as a cache of writes of translation table entries, each tagged with some context. This allows the model to cache any combination of entries read from a translation table walk, making it weak enough to allow all known TLB implementations, but strong enough to not break any of the guarantees Arm require of those TLB implementations. These guarantees are explored, in detail, in §8.4 and §8.5.

```
TranslationTableEntry ≡ u64
Context ≡ ArchContext × Stage × option VA × option IPA × PA × Level
ArchContext ≡ VMID × ASID × Regime
CachedTranslationTableEntry ≡ PA × TranslationTableEntry × Context
TLB ≡ set CachedTranslationTableEntry
```

Figure 8.9: Model TLB type definitions.

Each entry in the model TLB contains the information about the write itself: the physical address of the entry, and the cached 64-bit entry. But it must also be tagged with some contextual information, some used during TLB lookup and some used to identify cached entries during TLB invalidation. Figure 8.9 gives a consise summary of the model TLB definition in some psuedo-type-definitions.

This contextual information includes:

- the architectural context information of the translation: the VMID, ASID (or a "global indicator"), and the translation regime;
- some *extended context* information, required for implementing TLB maintenance:
  - the virtual address, intermediate physical address, and/or physical address of the translation;
  - the translation stage and level at which the write was used;
  - the system register values used in the translation (those which can be cached); and,
  - for an entry used for a Stage 1 translation, whether it has been invalidated at both stages.

The model MMU then performs all translations by doing a full translation table walk, but being able to optionally satisfy any read during that walk from a matching entry in the model TLB which matches the architectural context and input address.

We imagine that any behaviour exhibited by a specific micro-architectural MMU and TLB configuration would also be explainable in this model.

TLB fills Hardware has a variety of mechanisms which may lead to a translation table walk: direct architectural execution of instructions, pre-fetching of data or instructions, and speculation down branches. These translation table walks may result in TLB misses, and those misses then result in reads from memory and the MMU 'filling' the TLB with a copy of the information it can use in future.

Arm do not wish to enumerate all the possible speculation machinery or prefetchers so instead opt for a model that is weaker: at any point in time, any thread's MMU can spontaneously perform a translation table walk for any virtual or intermediate-physical address for the current architectural context (VMID, ASID, etc, as in §8.3.2), and any reads that the translation table walk performs can either read from other TLB entries, or perform a non-TLB read of memory and then potentially cache a copy of the write it reads from in the TLB tagged with the extended context information from the walk. The behaviour of those non-TLB reads are explored more in §8.4.

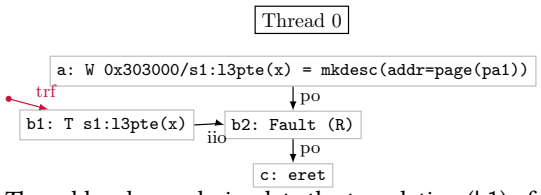
8.3.3 Invalid entries

It is architecturally forbidden to cache information from attempted translations which result in translation faults, access flag faults, or address size faults (Note that a translation table walk may give rise to other faults as well, as discussed in §7.3.2, such as permission faults and alignment faults, which do not impose restrictions on TLB caching). More specifically, a TLB entry cannot be a write of a translation table entry which is the *direct* cause of such a fault. In particular, the TLB cannot cache translation table entries whose valid bit is not set.

This is important, as it gives software a mechanism in which it can safely update a mapping without potentially having multiple entries in the TLB for the same virtual address. These problems are described in more detail during the exploration of break-before-make in §8.6.5.

TODO: PS: no forward refs to tests?

AArch64	CoWTf.inv+po
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x) 0:R3=x 0:VBAR_EL1=0x1000 0:PSTATE.SP=0b0	
physical pa1; x |-> invalid; x ↦ pa1; y |-> pa1; *pa1 = 1; **identity** 0x1000 **with code**;	
Thread 0	
STR X0,[X1] **LDR** X2,[X3]	
Thread 0 EL1 Handler	
0x1400: **MOV** X2,#0 **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**	
Final State	
0:X2=0	
Allow	



Thread local re-ordering lets the translation (b1) of the load instruction happen earlier than the write to the translation table (a). This allows the load to trigger a data abort (a translation fault, b2).

Figure 8.10: Test CoWTf.inv+po

8.4 Reads not from TLB

The requirement that invalid entries are not cached in the TLB gives us a way to directly observe non-TLB reads: translation table reads which result in a translation fault *must* have come from a non-TLB read.

We will see that these reads have some important properties that software can rely on, but that some of those properties will depend on certain architecture features being enabled (namely FEAT_ETS).

In this section will we explore the properties these reads have, and the guarantees software can rely on. We shall see that these reads are affected by thread-local re-ordering, even to a greater extent than data memory reads, and the synchronization that recovers the sequential semantics. We will see how these reads from the translation table walk relate to data memory reads, with respect to coherence, multi-copy atomicity, write forwarding and so on. Finally, we will see how the FEAT_ETS architectural feature can change the required synchronization software needs to perform.

8.4.1 Out-of-order execution

First, let us consider whether reads that do not come from the TLB preserve the original program order.

po-previous writes One of the simplest questions one might ask is whether a translation-table-walk non-TLB read can ignore a program-order previous store.

This scenario is captured by the CoWTf.inv+po test [Figure 8.10]. Starting with a VA x initially invalid at level 3, and so cannot have its level 3 entry cached in any TLB (directly or indirectly), the test then overwrites the invalid entry with a new valid entry pointing to the physical address pa1. Program-order later, the thread then attempts to read x.

We see that the thread can take a translation fault. This fault is caused by reading an invalid entry, which was read from a stale entry in memory, ignoring the program-order previous store to the translation table entry's location.

One explanation that suffices to allow this outcome is that the instructions can be locally re-ordered; the translation table walk of the later load instruction can happen much earlier than the program-order previous store, and satisfy its read from memory first.

po-previous reads Similarly, the reads of a translation table walk can be locally re-ordered with respect to program-order earlier loads of the translation table entry, as demonstrated in the CoRpteTf.inv+po test [Figure 8.11].

AArch64	CoRpteTf.inv+po
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x)	1:R1=**pte3**(x) 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
option default_tables = true; **physical** pa1; **intermediate** ipa1; x |-> invalid; x ↦ pa1; y |-> pa1; **identity** 0x1000 **with code**; *pa1 = 1;	
Thread 0	Thread 1
STR X0,[X1]	**LDR** X0,[X1] **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=**desc3**(y) & 1:X2=0	
Allow	

The translation read (event c1) can be re-ordered with respect to the program-order previous load of `l3pte(x)` (b), even though the load read the new translation table entry, for the same location the translation reads from.

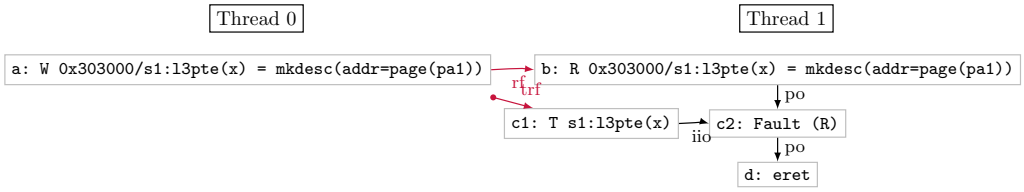


Figure 8.11: Test CoRpteTf.inv+po

AArch64	LB.TT.inv+pos
Initial State	
0:R1=x 0:R2=**mkdesc3**(oa=pa1) 0:R3=**pte3**(y) 0:VBAR_EL1=0x1000 0:PSTATE.SP=0b0 0:PSTATE.EL=0b00	1:R1=y 1:R2=**mkdesc3**(oa=pa1) 1:R3=**pte3**(x) 1:VBAR_EL1=0x2000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1; x |-> invalid; y |-> invalid; x ↦ pa1; y ↦ pa1; *pa1 = 1; **identity** 0x1000 **with code**; **identity** 0x2000 **with code**;	
Thread 0	Thread 1
MOV X0,#0 **LDR** X0,[X1] **STR** X2,[X3]	**MOV** X0,#0 **LDR** X0,[X1] **STR** X2,[X3]
Thread 0 EL1 Handler	Thread 1 EL1 Handler
0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**	0x2400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
0:X0=1 & 1:X0=1	
Forbid	

The writes to the translation tables (b and d) are forbidden from propagating to other threads before the program-order earlier translations (a1 and c1) are satisfied, forbidding them from reading from each other's writes.

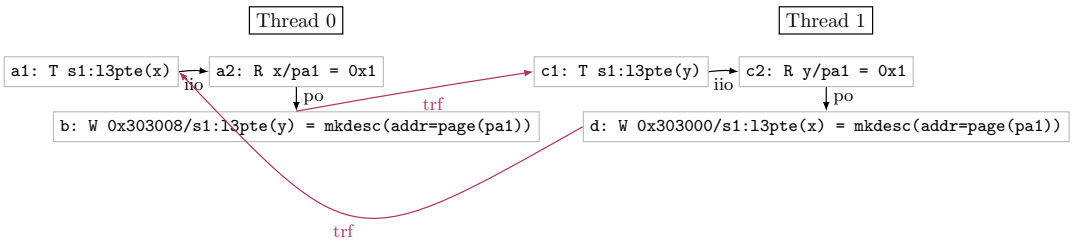


Figure 8.12: Test LB.TT.inv+pos

po-future writes A translation table walk read may not, in general, be re-ordered with program-order later stores.

This is consistent with the description in §8.2.3, as the program-order later store might not architecturally happen if the translation table walk read were to fault. So, the later writes are speculative until the translation has finished, preventing the write from propagating until then.

This forbids both the general re-ordering of the propagation of the write to other threads (LB.TT.inv+pos [Figure 8.12]) with program-order earlier translation table walks, and, translations reading from program-order later writes (CoTW1.inv [Figure 8.13]).

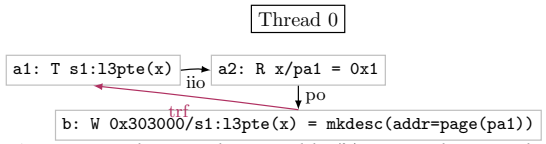
8.4.2 Enforcing thread-local ordering

Since non-TLB reads do not necessarily preserve the program order, it appears that there are no coherence guarantees one can make about them. However, by introducing some thread-local ordering constructs, we can recover some of the strong guarantees we are used to.

To force a non-TLB read to happen after some program-order earlier event we can insert the two-instruction sequence `DSB SY ; ISB` between them. The `DSB` ("Data Synchronization Barrier") waits for all loads to satisfy and for all stores to have finished and be visible to translation table walkers, before the `ISB` ("Instruction Synchronization Barrier") flushes the pipeline and restarts any program-order later instructions, including any translation table walks they perform.

Locally-ordered-previous writes If we introduce this sequence into the previous CoWTf.inv+po test we obtain the CoWTf.inv+dsb-isb test [Figure 8.14], which is forbidden by Arm. This is because the non-TLB reads, in the

AArch64	CoTW1.inv
Initial State	
0:R1=x 0:R2=**desc3**(y) 0:R3=**pte3**(x) 0:VBAR_EL1=0x1000 0:PSTATE.EL=0b00 0:PSTATE.SP=0b0	
physical pa1; x |-> invalid; x ↦ pa1; y |-> pa1; *pa1 = 1; **identity** 0x1000 **with code**;	
Thread 0	
LDR X0,[X1] **STR** X2,[X3]	
Thread 0 EL1 Handler	
0x1400: **MOV** X0,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**	
Final State	
0:X0=1	
Forbid	



The store to the translation table (b) cannot be re-ordered with the program-order earlier translation table walk (a), preventing that walk from reading from the store.

Figure 8.13: Test CoTW1.inv

absence of non-coherent TLB caching structures (discussed more in §8.6.1), will read from the coherent storage subsystem, and so will be required to see the new write, or something coherence after it.

Locally-ordered-previous reads If a program-order previous load has already seen some other-thread write, either through a translation (CoTTf.inv+dsb-isb [Figure 8.15]), or through a normal data load of the translation table (CoRpteTf.inv+dsb-isb [Figure 8.16]), then translation table non-TLB reads which are ordered after that read must also see that write, or a write coherence after it. These tests use the `DSB; ISB` sequence previously described, but any ordering to the translation table walk (described in §8.4.3) will suffice.

Microarchitecturally this is because translation table walkers are 'separate observers'. The idea is that the MMU performs reads of memory the same way any of the other observers (threads) do, meaning that those reads behave almost exactly like normal data memory reads.

This 'separate observers' principle is a reasonable model, however, we will see later on in §8.4.4 where it begins to break down.

Instruction synchronization barrier and control dependencies The `ISB` instruction naturally orders all translation table walks of program-order later instructions with the `ISB` itself. This is because the `ISB` effectively restarts all program-order later instructions, including any translations they do.

However, an `ISB` is not naturally ordered with respect to program-order *earlier* instructions. That is why in the previous tests we introduced a `DSB`. But a control-dependency would also work (CoTTf.inv+ctrl-isb [Figure 8.17]).

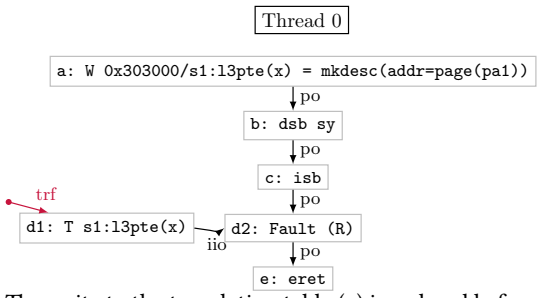
Address dependencies In previous work, address dependencies were assumed fundamental, but now we can define what an address dependency is: a register dataflow dependency into the translation table walk reads.

Address dependencies remain a strong way to order events. Arm, here and in general, avoid speculation of the values and addresses of the explicit reads and writes to memory. This means that a translation table walk will not start until after its address dataflow dependent registers are fully determined. Note, that this does not mean that pre-fetching and caching of the walk cannot happen, it's just that the architectural translation table walk must retrieve any cached values after it is known what the address will be, see §**TODO: ?REF?**.

For non-TLB translation reads this means that a non-TLB read is locally ordered after any read whose value flows into the non-TLB read, as in CoRpteTf.inv+addr [Figure 8.18].

Memory barriers Much of the earlier work in relaxed-memory concurrency was dedicated to the behaviour of *barriers*. The Arm data memory barrier (`DMB`) creates ordering between memory events program-order earlier than the barrier, and memory events program-order after the barrier.

AArch64	CoWTf.inv+dsb-isb
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x) 0:R3=x 0:VBAR_EL1=0x1000 0:PSTATE.SP=0b0	
physical pa1; x |-> invalid; x ↦ pa1; y |-> pa1; *pa1 = 1; **identity** 0x1000 **with code**;	
Thread 0	
STR X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]	
Thread 0 EL1 Handler	
0x1400: **MOV** X2,#0 **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**	
Final State	
0:X2=0	
Forbid	



The write to the translation table (a) is ordered before the non-TLB read of the entry (d1) because of the intervening DSB;ISB sequence, creating local order. This ordering ensures that the non-TLB read respects the coherence order up to the point of the write a, preventing the non-TLB read from reading from a write coherence-before a.

Figure 8.14: Test CoWTf.inv+dsb-isb

AArch64	CoTTf.inv+dsb-isb
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x)	1:R1=x 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1; x |-> invalid; x ↦ pa1; y |-> pa1; *pa1 = 1; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1]	**LDR** X2,[X1] **MOV** X0,X2 **DSB** SY **ISB** **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Forbid	

The second translation-table non-TLB read of x (e1) is locally ordered after the first translation table walk (b1) because of the intervening dsb; isb sequence, and so cannot see a write coherence-before the write the earlier (b1) translation-read read from.

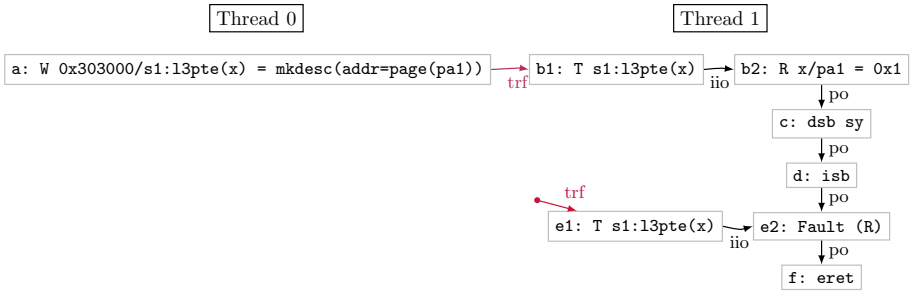


Figure 8.15: Test CoTTf.inv+dsb-isb

AArch64	CoRpteTf.inv+dsb-isb
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x)	1:R1=**pte3**(x) 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
option default_tables = true; **physical** pa1; **intermediate** ipa1; x |-> invalid; x ↦ pa1; y |-> pa1; **identity** 0x1000 **with code**; *pa1 = 1;	
Thread 0	Thread 1
STR X0,[X1]	**LDR** X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=**desc3**(y) & 1:X2=0	
Forbid	

The final translation table walk of x (e1) cannot be reordered with the program-order previous load of pte3(x) (b), because of the intervening DSB;ISB sequence. The non-TLB translation read of pte3(x) (e1) therefore must read from the same write as the earlier load, or something coherence-after it.

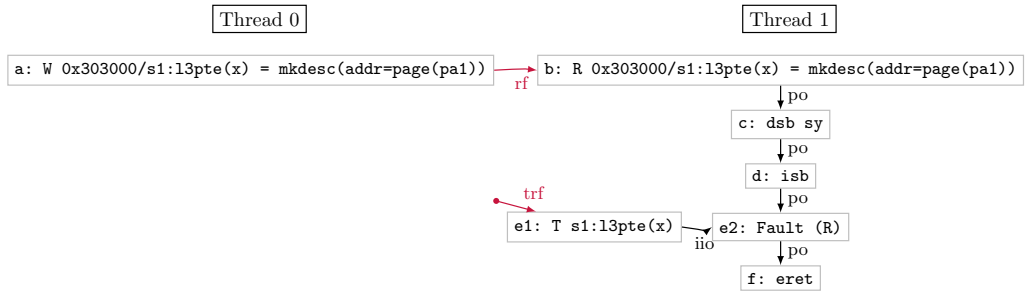


Figure 8.16: Test CoRpteTf.inv+dsb-isb

AArch64	CoTTf.inv+ctrl-isb
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x)	1:R1=x 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1; x |-> invalid; x ↦ pa1; y |-> pa1; *pa1 = 1; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1]	**MOV** X0,#0 **LDR** X0,[X1] **EOR** X4,X0,X0 **CBNZ** X4,LC00 LC00: **ISB** **MOV** X2,#0 **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Forbid	

Control-ISB locally-orders the later translation table walk (d1) after the resolution of the control flow, which happens only after the satisfaction of the read b2.

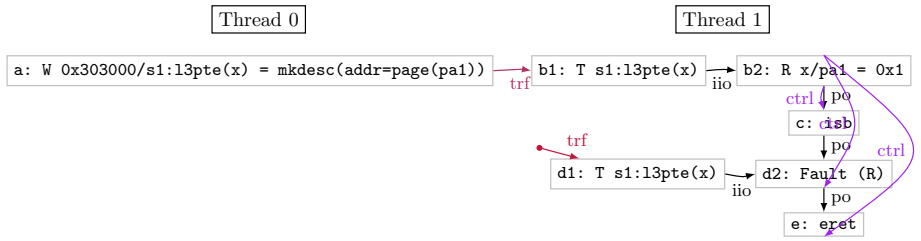


Figure 8.17: Test CoTTf.inv+ctrl-isb

AArch64	CoRpteTf.inv+addr
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x)	1:R1=**pte3**(x) 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
option default_tables = true; **physical** pa1; **intermediate** ipa1; x |-> invalid; x ↦ pa1; y |-> pa1; **identity** 0x1000 **with code**; *pa1 = 1;	
Thread 0	Thread 1
STR X0,[X1]	**LDR** X0,[X1] **EOR** X4,X0,X0 **LDR** X2,[X3,X4]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=**desc3**(y) & 1:X2=0	
Forbid	

The address dependency from the load b to the second load, orders the reads due to the translation table walk of that load (c1) after b. Since c1 is a non-TLB read, it cannot read from a write coherence-before the write b read from.

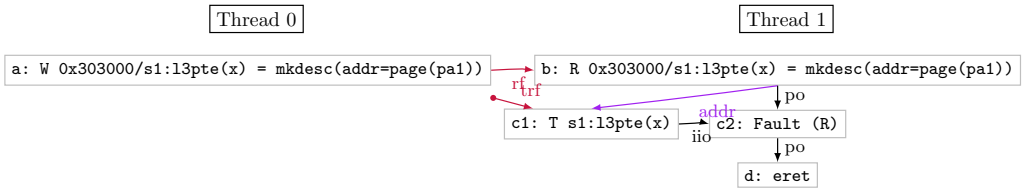
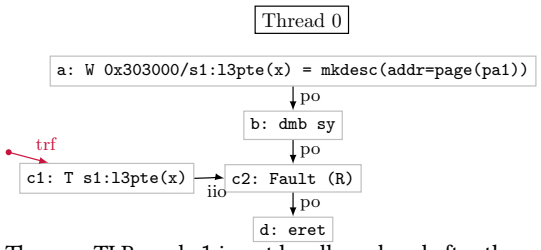


Figure 8.18: Test CoRpteTf.inv+addr

AArch64	CoWTf.inv+dmb
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x) 0:R3=x 0:VBAR_EL1=0x1000 0:PSTATE.SP=0b0	
physical pa1; x |-> invalid; x ↦ pa1; y |-> pa1; *pa1 = 1; **identity** 0x1000 **with code**;	
Thread 0	
STR X0,[X1] **DMB** SY **LDR** X2,[X3]	
Thread 0 EL1 Handler	
0x1400: **MOV** X2,#0 **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**	
Final State	
0:X2=0	
Allow (if not ETS)	



The non-TLB read c1 is not locally ordered after the write a, despite the intervening dmb sy barrier (b).

Figure 8.19: Test CoWTf.inv+dmb

We will see that this applies to *explicit* memory events only: the principal reads and writes that load and store instructions perform, not the implicit reads and writes they do during translations (or instruction fetching, **TODO: ref: ifetch chapter**).

Ordering of the explicit memory events does not, automatically, induce ordering between those explicit events and any reads due to translation table walks performed by those instructions. In the next subsection, we will see how FEAT_ETS (§8.4.3) extends the architecture to include more orderings between translations and other memory events in the same thread.

Figure 8.19 shows a simple coherence test, with a data memory barrier between a store to the translation tables and a load whose translation table walk might read from that. We can see that the barrier does not enforce that the translation table walk sees the update to the translation tables. From the previous tests, we know this means that the translation table walk happened (microarchitecturally) before the store was propagated to memory.

The arm DMB vs DSB instructions TODO: PS: discuss DMB v DSB

The architectural intent for DMB's ordering with respect to translation table walkers in the absence of FEAT_ETS is still tentative, so we shall focus on the fragment with FEAT_ETS**TODO: ... and continue.**

8.4.3 Enhanced Translation Synchronization

TODO: PS: litmus tests?

Recent versions of the Arm architecture require support for FEAT_ETS: Enhanced Translation Synchronization. This feature does not change the ISA, but instead, requires implementations to enforce extra ordering.

The Arm Architecture Reference Manual says the following [1, D5.2.5 (p4802)] :

> If FEAT_ETS is implemented, and a memory access RW1 is Ordered-before a second memory access RW2, then RW1 is also Ordered-before any translation table walk generated by RW2 that generates any of the following:
> - ▷ A Translation fault.
> - ▷ An Address size fault.
> - ▷ An Access flag fault.

This prose description is a little ambiguous, and we feel, needs some clarification: The scenario being described here is a case with two instructions, I_1 and I_2, each either a load or store. Imagine I_1 and I_2 both executing to completion, without generating any translation, address size, or access flag faults. Then, each instruction would

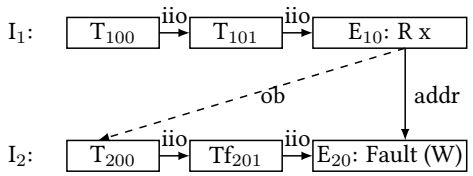


Figure 8.20: ETS Ghost events example: A load instruction (I_1) followed followed (in program order) by a store instruction (I_2), which faults. The address dependency means that the read event E_{10} is syntactically ordered-before the (ghost) write event E_{20}, and so the read event is ordered before the reads of the translation table walk for I_2 read from the TLB or memory (represented by the dashed ob line).

have generated one or more explicit memory events. For example, a store might generate up to 8 separate write events (one for each byte). Call those events E_{ij} for the jth explicit event of instruction I_i.

Each explicit event E_{ij} would have required a translation table walk, generating translation read events which we can call T_{ijk} for the kth translation-table-walk read for the jth explicit memory event for instruction I_i.

Then, if I_2 generates a translation, address size, or access flag fault, and E_{1n} would have been locally-ordered-before E_{2m} in the imagined execution without the fault (and which we can consider a kind of *ghost* event in the real execution), and FEAT_ETS is enabled, then, E_{1n} is locally ordered before any translation table read $T_{2m\_}$ in the execution with the fault. This scenario is described pictographically in Figure 8.20.

The intuition here is that, microarchitecturally, on implementations that support FEAT_ETS, when an instruction takes an exception, the access that caused it is re-tried once the prefix of instructions is non-restartable. This reduces *spurious aborts*: faults that come from an out-of-order read of a (what is now) stale value from memory.

Other effects The architecturally desired effect of FEAT_ETS seems to be that no additional context-synchronisation should be required to prevent spurious aborts, and that simple local orderings (barriers, dependencies) should be enough. To make this so, ETS must implicitly enforce more than just the aforementioned ordering constraints.

Specifically, TLBI instructions must have stronger thread-local orderings to translation-table walks (described in more detail later); translation table walks must be (other) multi-copy atomic; and, translation table walk reads must be coherent and single-copy atomic.

non-ETS fragment There is a question here as to whether we should consider the non-ETS behaviours of the architecture. On the one hand, hardware in use today is from a pre-ETS version of the architecture and so we cannot assume that the behaviours of those devices are consistent with ETS. On the other hand, ETS is a feature that is widely assumed by software, even if not present on hardware.

Linux, for example, assumes implementations are ETS compatible even when they are not. Building models that capture the full extent of the non-ETS fragment would have questionable benefits as one would have to assume an ETS model when verifying software. Additionally, as ETS is becoming a mandatory feature, the concerns over non-ETS hardware will diminish over time, perhaps even by the publication of this thesis, they will be questions of the past. Finally, the semantics of this non-ETS fragment is still unclear; there are numerous questions, especially around forwarding and multi-copy atomicity generally, which are grey areas in the non-ETS fragment which Arm have yet to explicitly decide one way or another.

For these reasons we will assume FEAT_ETS is present and enabled, unless explicitly stated otherwise.

Ordering to the translation table walk We can now define which constructs give rise to local ordering into a translation table walk. Address dependencies, and locally-ordered context-synchronisation (in particular, the DSB; ISB sequence) always give rise to ordering to the translation table walks. Control dependencies, on their own, never give rise to such ordering. If using FEAT_ETS, then a plain DSB orders translation table walks of program-order later instructions after it. **TODO: BS: even if there's no fault?** Other barriers may give ordering to the translation table walker, if using FEAT_ETS and the translation results in a translation fault, and those barriers would have ordered the event that would have happened otherwise.

AArch64		R.TR.inv+dmb+trfi
Initial State		
0:R0=0x2 0:R1=x 0:R2=0x2 0:R3=**pte3**(w)	1:R0=**mkdesc3**(oa=pa1) 1:R1=**pte3**(w) 1:R3=w 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00	2:R1=**pte3**(w)
physical pa1; w |-> invalid; w ↦ pa1; w ↦ raw(2); x |-> pa1; *pa1 = 0; **identity** 0x1000 **with code**;		
Thread 0	Thread 1	Thread 2
STR X0,[X1] **DMB** SY **STR** X2,[X3]	**STR** X0,[X1] **MOV** X2,#1 **LDR** X2,[X3]	**LDR** X0,[X1] **LDR** X2,[X1]
	Thread 1 EL1 Handler	
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**	
Final State		
1:X2=0 & 2:X0=2 & 2:X2=**mkdesc3**(oa=pa1)		
Allow		

The write of the new valid entry (d) can be forwarded locally to the translation of w (e1) allowing the read of w (e2) to satisfy early. **TODO: PS: Thread2 needs explaining**

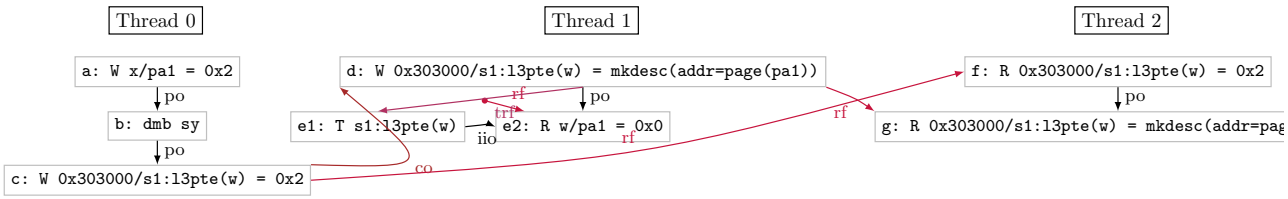


Figure 8.21: Test R.TR.inv+dmb+trfi

8.4.4 Forwarding to the translation table walker

Writes take time to propagate out to memory to other cores. One common performance optimization is *gathering*: collecting multiple writes together in a store buffer and propagating them all out together.

To maintain uniprocessor semantics, the core can read from its own store buffer, in effect, allowing it to read from writes before they've been propagated out to other cores. This behaviour is referred to as *write forwarding*.

Although the translation table walker is described as a 'separate' observer, it is also part of the core that hosts it, and is allowed to read from that core's store buffer, effectively allowing writes to be 'forwarded' to the walker, as shown in the R.TR.inv+dmb+trfi test [Figure 8.21].

The simplest model here is one where non-TLB translation reads behave as a normal data memory read, reading either from forwarding from the store buffer, or from the coherence-latest write in the storage subsystem.

8.4.5 Speculative execution

To facilitate fast out-of-order pipelines the machine has to begin fetching and executing the next instruction before the earlier instructions are finished. But, those instructions might control the flow of execution through the program. Executing later instructions before they are finished means that those later instructions are being executed *speculatively*: they may, if the predicted flow turns out to be incorrect, need to be discarded, **TODO: PS: what about restarting on coherence violations?** to avoid the need for rollback across threads.

When executing down a speculative path like this, there are additional restrictions that the core must adhere to. For example, stores should not be propagated out to memory, although they can still be read from by program-order later reads in the same thread.

Since we know reads and writes can be performed speculatively, their associated translations must also be allowed

AArch64	MP.RTf.inv+dmb+ctrl
Initial State	
0:R0=**desc3**(z) 0:R1=**pte3**(x) 0:R2=0b1 0:R3=y	1:R1=y 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1 pa2; x |-> invalid; x ↦ pa1; z |-> pa1; *pa1 = 1; y |-> pa2; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3]	**LDR** X0,[X1] **CBNZ** X0,L0 L0: **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Allow	

The non-TLB read in Thread 1 (e1) is not locally ordered after the earlier load (d), despite the control dependency. This is because the processor can speculatively perform the translation table walk, before the earlier read is satisfied.

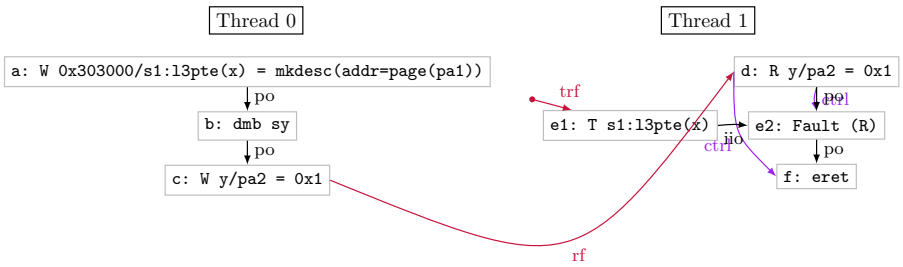


Figure 8.22: Test MP.RTf.inv+dmb+ctrl

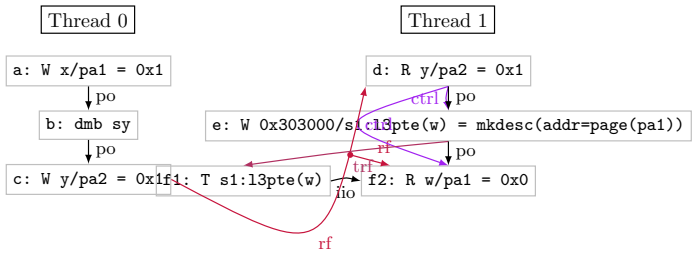
to have been performed speculatively. This is what allows the MP.RTf.inv+dmb+ctrl test [Figure 8.22] to see an old value for the translation table entry, as the translation can be performed speculatively. **TODO: PS: If this were a "user" test, I'd say that e1 was satisfied out-of-order w.r.t. d, not that e1 was "performed speculatively". Or I'd expect to see a test with control-flow speculation, or argument that the second instruction is speculative until the first is known not to fault. Are you not distinguishing between out-of-order and speculative execution any more? TODO: BS: but speculation implies OoO?**

However, forwarding from a speculative write to the translation table walker is disallowed. Since reads to read-sensitive locations (such as devices) can have side-effects, software can protect those locations by marking them as device memory in the translation tables, or leaving them unmapped altogether. A speculative write could update the translation tables arbitrarily, including allowing reads to read-sensitive locations, so it must be forbidden for a translation read to read from a still speculative write. The MP.RT.inv+dmb+ctrl-trfi test [Figure 8.23] demonstrates this, requiring that the translation table walk on the speculative path cannot read from the still-speculative store to the translation tables.

Instruction restarts A related, but separate, concept, is that of instruction restarts. In the **TODO: PS: user-mode?** base memory model a read might be satisfied early, out-of-order with respect to program-order previous instructions, even before those instructions' accesses addresses are known. If such an earlier access turned out to be to the same address, and the later access is not a read of the same write, then the later access must be restarted to avoid coherence violations.

Translation table walk reads, while they are reads, do not do this hazard checking, and so are not required to be restarted to recover coherence. See §8.2 for more discussion on this. **TODO: PS: 8.2 has a lot of stuff, point to specifics?**

AArch64	MP.RT.inv+dmb+ctrl-trfi
Initial State	
0:R0=0b1 0:R1=x 0:R2=0b1 0:R3=y	1:R1=y 1:R2=**mkdesc3**(oa=pa1) 1:R3=**pte3**(w) 1:R5=w 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1 pa2; w |-> invalid; w ↦ pa1; x |-> pa1; *pa1 = 0; y |-> pa2; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3]	**LDR** X0,[X1] **CBZ** X0,LC00 LC00: **STR** X2,[X3] **LDR** X4,[X5]
	Thread 1 EL1 Handler
	0x1400: **MOV** X4,#2 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X4=0	
Forbid	



The non-TLB read of the translation table entry (f1) cannot read from a forwarded thread-local write (event e) when on a speculative path, requiring that f1 be ordered after d. **TODO: PS: manual layout this**

Figure 8.23: Test MP.RT.inv+dmb+ctrl-trfi

8.4.6 Single-copy atomicity

In the base memory model, there are two key guarantees on the *atomicity* of reads and writes: single-copy and multi-copy atomicity.

Recall that, single-copy atomic reads always read the maximum it can from another single-copy atomic write; in particular a 64-bit atomic never partially reads from another 64-bit atomic write.

Translation table walk reads are 64-bit single-copy-atomic reads of memory. This means that each of the reads generated by a translation table walk will read the entire descriptor in one shot. This causes the CoWroW.inv+dsb-isb test [Figure 8.24] to be forbidden, disallowing reading the output address obtained from one write, and access permissions from another.

8.4.7 Multi-copy atomicity

Multi-copy atomicity is a guarantee that requires any update to memory to propagate to all other threads simultaneously. This is one of the core guarantees Armv8 and RISC-V give, but earlier versions of Arm and IBM's current Power architectures do not. This has a caveat for Armv8, which is described as *other*-multi-copy atomic: threads can observe their own writes early (through write forwarding).

Microarchitecturally, a thread can only read another thread's write by reading from a global coherent storage subsystem. This ensures that after the thread reads from that write, any other thread must also see that write, or something coherence after it. While this is a property that the base model seems to have, whether it is true for accesses during translation table walks is a separate question.

The non-TLB reads during a translation table walk, in fact, do seem to respect this property: if one other thread has observed a write through a translation table walk then future translation table walk non-TLB reads by other threads will also observe that write (or something newer). Axiomatically, if one thread translation-reads-from a write, then all translation-table-walk reads locally-ordered after another memory event, which is itself ordered after the other thread's translation-table-walk read, will be ordered after that translation-table-walk read.

There are three combinations of multi-thread reads of interest, where a weaker architecture (with separate pagetable and data memory storage) might have mixed non-multi-copy atomic behaviours. The first of these is

AArch64	CoWroW.inv+dsb-isb
Initial State	
0:R0=**mkdesc3**(oa=pa1, AP=0b11) 0:R1=**pte3**(x) 0:R2=0x1 0:R3=x 0:VBAR_EL1=0x1000 0:PSTATE.SP=0b0	
physical pa1; x |-> invalid; x ↦ pa1 **with** [AP = 0b11] and default; *pa1 = 0; **identity** 0x1000 **with code**;	
Thread 0	
STR X0,[X1] **DSB** SY **ISB** **STR** X2,[X3]	
Thread 0 EL1 Handler	
0x1400: **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**	
Final State	
pa1=1	
Forbid	

The translation table walk of the second store must read from the entire write from the earlier store, or not at all, forbidding its translation walk from reading a mix of both the initial state and the earlier write. This means there should be no way the final store can happen, as it must either be invalid or read-only.
Note that, isla does not generate candidates with non-atomic reads which are supposed to be single-copy atomic, and so the diagram is hand-drawn **TODO: Draw it**.

Figure 8.24: Test CoWroW.inv+dsb-isb

the most basic; translation-read to translation-read, that is, the pagetable accesses are multi-copy atomic, and this is what forbids reading the old translation table value in Thread 2 in the WRC.TRTf.inv+po+dsb-isb test [Figure 8.25]. The other two are combinations of read-to-translation-read and translation-read-to-read, these show us that the translation accesses and explicit data accesses are architecturally unified: information about the memory state learned through one kind of access apply to accesses of the other. This is what forbids the following WRC.RRTf.inv+dmb+dsb-isb [Figure 8.26] and WRC.TRR.inv+po+dsb [Figure 8.27] tests, from reading the old value from memory at the end of Thread 2.

TODO: PS: these all need text captions

8.4.8 Translation-table-walk intra-walk ordering

All the tests so far have been concerned with changes to at most one of the translation table entries during a single walk, however, as we saw in §7 a translation table walk may perform many reads for a single translation.

The ASL for the translation table walker performs each translation, in order, starting with the root, and ending with the leaf entry.

While reads in a thread can be re-ordered, translation-reads within a translation table walk cannot, as this would require the hardware to do value speculation on the next-level table address, and as discussed in §8.4.5 reading from speculative values in a translation table walk is generally forbidden.

Requiring the translation reads from a translation table walk to be satisfied in translation walk order has an observable effect, for example in the following ROT.inv+dsb test [Figure 8.28] the translation table walk of the read in Thread 1 must see the writes to the translation table done by Thread 0 in the order they were propagated out to memory, and so reading from the old level 3 entry is forbidden.

8.4.9 Multiple translations within a single instruction

Some instructions generate multiple explicit memory events, such as for the load pair and store pair instructions, or misaligned accesses, or potentially some read-modify-writes. When there are multiple explicit memory events, there will be a dedicated translation for each of them, with its own translation table walk.

Here the architecture as it is written today is overly sequentialised: the ASL for these cases performs each translation (and the respective access) in some order, but the architectural intent is that the separate translations should be unordered with respect to each other.

AArch64		WRC.TRTf.inv+po+dsb-isb
Initial State		
0:R0=**desc3**(z) 0:R1=**pte3**(x) 0:PSTATE.EL=0b00 0:PSTATE.SP=0b0	1:R1=x 1:R2=0b1 1:R3=y 1:VBAR_EL1=0x1000 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0	2:R1=y 2:R3=x 2:VBAR_EL1=0x2000 2:PSTATE.SP=0b0 2:PSTATE.EL=0b00
physical pa1 pa2; x |-> invalid; x ↦ pa1; z |-> pa1; *pa1 = 1; y |-> pa2; **identity** 0x1000 **with code**; **identity** 0x2000 **with code**;		
Thread 0	Thread 1	Thread 2
STR X0,[X1]	**LDR** X0,[X1] **STR** X2,[X3]	**LDR** X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]
	Thread 1 EL1 Handler	Thread 2 EL1 Handler
	0x1400: **MOV** X0,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**	0x2400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State		
1:X0=1 & 2:X0=1 & 2:X2=0		
Forbid		

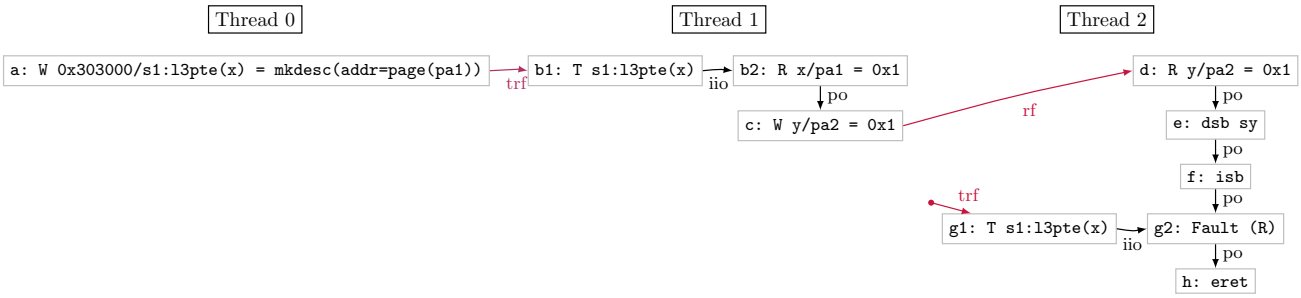


Figure 8.25: Test WRC.TRTf.inv+po+dsb-isb

AArch64		WRC.RRTf.inv+dmb+dsb-isb
Initial State		
0:R0=**desc3**(z) 0:R1=**pte3**(x) 0:PSTATE.EL=0b00 0:PSTATE.SP=0b0	1:R1=**pte3**(x) 1:R2=0b1 1:R3=y 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0	2:R1=y 2:R3=x 2:VBAR_EL1=0x2000 2:PSTATE.SP=0b0 2:PSTATE.EL=0b00
physical pa1 pa2; x |-> invalid; x ↦ pa1; z |-> pa1; *pa1 = 1; y |-> pa2; **identity** 0x1000 **with code**; **identity** 0x2000 **with code**;		
Thread 0	Thread 1	Thread 2
STR X0,[X1]	**LDR** X0,[X1] **DSB** SY **STR** X2,[X3]	**LDR** X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]
		Thread 2 EL1 Handler
		0x2400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State		
1:X0=**desc3**(z) & 2:X0=1 & 2:X2=0		
Forbid		

~~TODO: PS: why DSB not just any R/R ordering.~~

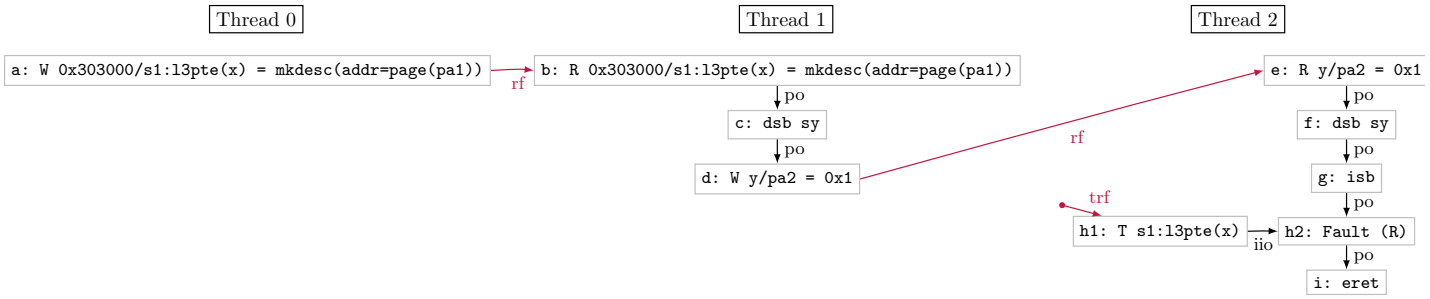


Figure 8.26: Test WRC.RRTf.inv+dmb+dsb-isb

AArch64		WRC.TRR.inv+po+dsb
Initial State		
0:R0=**mkdesc3**(oa=pa1) 0:R1=**pte3**(x) 0:PSTATE.EL=0b00 0:PSTATE.SP=0b0	1:R0=0b0 1:R1=x 1:R2=0b1 1:R3=y 1:VBAR_EL1=0x1000 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0	2:R1=y 2:R3=**pte3**(x) 2:PSTATE.SP=0b0 2:PSTATE.EL=0b00
physical pa1 pa2; x |-> invalid; x ↦ pa1; y |-> pa2; *pa1 = 1; **identity** 0x1000 **with code**;		
Thread 0	Thread 1	Thread 2
STR X0,[X1]	**LDR** X0,[X1] **STR** X2,[X3]	**LDR** X0,[X1] **DSB** SY **LDR** X2,[X3]
	Thread 1 EL1 Handler	
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**	
Final State		
1:X0=1 & 2:X0=1 & ~2:X2=0		
Allow		

TODO: PS: why DSB not just any R/R ordering.

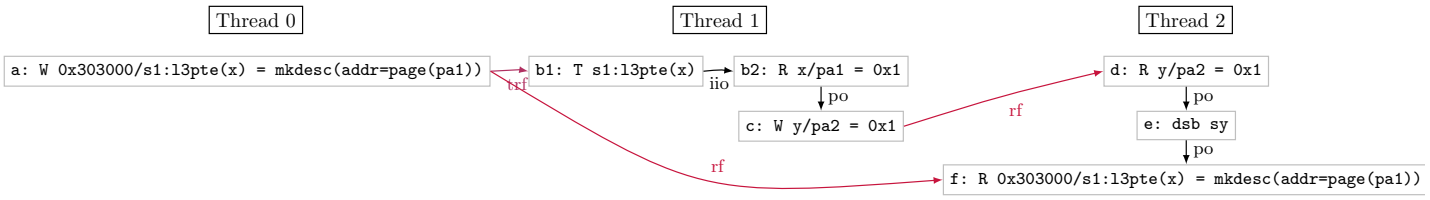


Figure 8.27: Test WRC.TRR.inv+po+dsb

AArch64	ROT.inv+dsb
Initial State	
0:R0=**mkdesc3**(oa=ipa1) 1:R1=x 0:R1=**pte3**(x, new_table) 1:VBAR_EL1=0x1000 0:R2=**mkdesc2**(table=0x283000) 0:R3=**pte2**(x) 0:PSTATE.EL=0b01	

physical pa1;
intermediate ipa1;
assert pa1 == ipa1;
ipa1 |-> pa1;
x |-> invalid at level 2;
x ↦ table(0x283000) at level 2;
sltable new_table 0x280000 {
  x |-> invalid;
  x ↦ ipa1;
};
identity 0x1000 **with code;**	

Thread 0	Thread 1
STR X0,[X1] **DSB** SY **STR** X2,[X3]	**LDR** X0,[X1]
	Thread 1 EL1 Handler
	0x1400: *// read ESR_EL1.ISS, to see* **MRS** X14,ESR_EL1 **AND** X14,X14,#0b111 **CMP** X14,#0b111 **MOV** X17,#1 **MOV** X18,#2 *// if ESR_EL1.ISS.DFSC ==* **CSEL** X0,X17,X18,eq *// advance ELR* **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 *// return* **ERET**
Final State	
1:X0=1	
Forbid	

The translation-table walk from the read of x in Thread 1 must perform its translation non-TLB reads in the order they appear in the walk, forbidding reading from the new level 2 table entry in d1, but then reading the stale initial value for that entry from memory.

The test listing contains some concrete values to make it executable in isla, namely fixing the location of the new table at `0x280000` so it's not symbolic, and the exact location of the level 3 entry within the new table will be at `0x283000` (known from the fixed isla configuration). Whether the exception comes from the level 2 or the level 3 entry can be determined by reading the ISS field of the ESR_EL1 register, which the exception handler does.

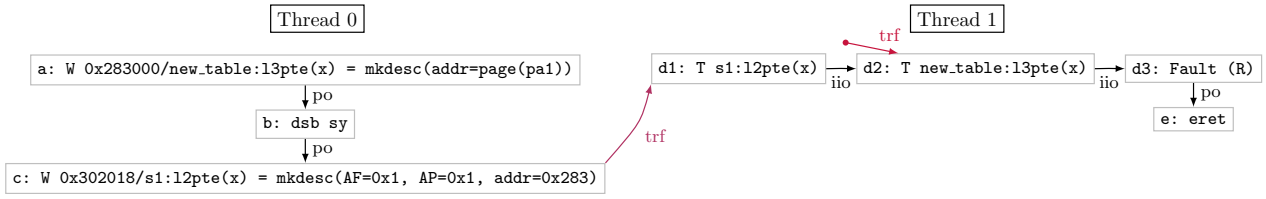
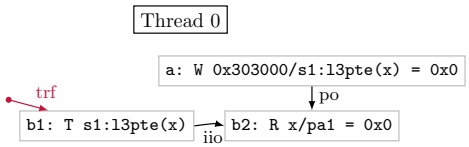


Figure 8.28: Test ROT.inv+dsb

Misaligned accesses, and the load and store pair instructions, should generate explicit memory events and associated translations which are unordered with respect to each other.

TODO: PS: litmus test with misalgined?

AArch64	CoWinvT+po
Initial State	
0:R0=0b0 0:R1=**pte3**(x) 0:R3=x 0:VBAR_EL1=0x1000 0:PSTATE.SP=0b0	
physical pa1 pa2; x |-> pa1; x ↦ invalid; **identity** 0x1000 **with code**;	
Thread 0	
STR X0,[X1] **LDR** X2,[X3]	
Thread 0 EL1 Handler	
0x1400: **MOV** X2,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**	
Final State	
0:X2=0	
Allow	



The translation read (b1) of the last-level entry for x can be re-ordered with respect to the program-order earlier store (a) to pte3(x).

Figure 8.29: Test CoWinvT+po

8.5 Caching of translations in TLBs

We have seen in §8.4 that, while non-TLB reads do not necessarily preserve the program-order without additional synchronisation due to the out-of-order execution of instructions, those translation table reads get satisfied from the coherent storage subsystem or from forwarding from earlier stores, much like the normal explicit data reads do. This section will explore what happens when translation table walk reads may instead be satisfied from the TLB.

Unfortunately for the programmer, the TLB need not be coherent with memory: it can have stale values. This section explores the behaviours that arise from this caching of stale values.

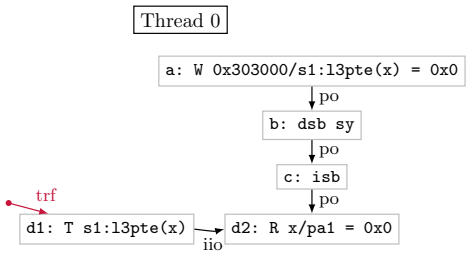
8.5.1 Cached translations

In the previous section we carefully constructed tests which began with an initially invalid translation, to avoid TLB caching issues. Here, we will generally start with entries that are valid, and so might be present in the TLB.

The following CoWinvT+po test [Figure 8.29] begins with an *initially valid* (and therefore potentially initially *cached in the TLB*) translation for the virtual address x. It then updates the last-level translation table entry for x, setting it to 0, making it invalid (and thus unmapping x). Then, program order later, the same thread tries to read x.

The read can succeed, as its translation can read from the old value from memory. We saw earlier that translation table walks can be re-ordered with respect to program order, but even inserting thread-local ordering to the translation, such as in test CoWinvT+dsb-isb [Figure 8.30], does not forbid it.

AArch64	CoWinvT+dsb-isb
Initial State	
0:R0=0b0 0:R1=**pte3**(x) 0:R3=x 0:VBAR_EL1=0x1000 0:PSTATE.SP=0b0	
physical pa1 pa2; x |-> pa1; x ↦ invalid; **identity** 0x1000 **with code;**	
Thread 0	
STR X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]	
Thread 0 EL1 Handler	
0x1400: **MOV** X2,#1 **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**	
Final State	
0:X2=0	
Allow	



The translation read (d1) of the last-level entry for x is required to be satisfied after the earlier store (a) to the entry's location because of the intervening `dsb sy; isb` sequence, but can be satisfied from a cached value in the TLB, allowing d1 to read from a stale value.

Figure 8.30: Test CoWinvT+dsb-isb

8.5.2 TLB fills

Translation table walks can be requested by the core in two different ways: (1) through the architectural execution of an instruction; or, (2) from a spontaneous translation table walk (for example, due to speculation and prefetching of data or instructions). In either case, the result of that walk can be cached in the TLB and recalled for other translation table walks.

Architecturally a TLB fill is no different to a normal translation table walk; each fill originates from a non-TLB read, with all the behaviours described in the previous sections. Later translation table walks are allowed, however, to recall an earlier value and then reuse that rather than doing a fresh read.

Spontaneous walks The hardware may, at any time, try to prefetch or speculatively read some address. Architecturally these appear as spontaneous translation table walks. Those spontaneous walks may be cached. We can see this occuring in the following MP.RT.inv+poloc-dmb+ctrl-isb test [Figure 8.31], where a spontaneous translation and the resulting TLB fill allows a future translation table walk to see a stale value.

Speculative paths Since translation table walks, and therefore TLB fills from the result of those walks, can happen at any point, there is no need to consider TLB fills of architectural translation table walks down speculative paths as any such behaviour is subsumed by a spontaneous fill.

However, as described earlier, we saw that writes cannot be forwarded to translation table walks when down speculative paths (§8.4.5), as this would lead to security violations. This naturally excludes TLB fills of still speculative writes; since a speculative write cannot be used in the result of a translation table walk, it cannot end up cached in a TLB.

8.5.3 micro-TLBs

So far we have spoken as if entries are, at any particular moment in time, either present in the TLB or not. Hardware, however, may have multiple *micro*-TLBs for the same thread, each with their own potential cached entry for the same address.

In effect, these micro-TLBs behave as if they were a larger non-deterministic TLB with potentially many values for each entry. The presence of these smaller caching structures in a superscalar machine means that different instructions may be accessing different TLBs at the same time. This allows later instructions to 'skip' over a previously seen cached entry, and then see it again later.

These effects can be seen in the CoTfT+dsb-isb test [Figure 8.32], where the presence of these micro-TLBs (or other distributed caching structures) allow later events (even locally-ordered later) to see old cached entries after

AArch64	MP.RT.inv+poloc-dmb+ctrl-isb
Initial State	
0:R0=**mkdesc3**(oa=pa1) 0:R1=**pte3**(x) 0:R2=0b0 0:R3=**pte3**(x) 0:R4=0b1 0:R5=y	1:R1=y 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00

physical pa1 pa2;
x |-> invalid;
x ↦ pa1;
y |-> pa2;
*pa1 = 0;
*pa2 = 0;
identity 0x1000 **with code**;	

Thread 0	Thread 1
STR X0,[X1] **STR** X2,[X3] **DMB** SY **STR** X4,[X5]	**LDR** X0,[X1] **CBNZ** X0,L0 L0: **ISB** **MOV** X2,#1 **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Allow	

A spontaneous walk and fill can happen on Thread 1 after the write of the valid entry to `pte3(x)` (a), but before the immediate re-invalidation of that entry (b), allowing the later translation table walk to see the old cached entry (g1), even though the architectural translation table walk could not have happened while the valid entry was visible.

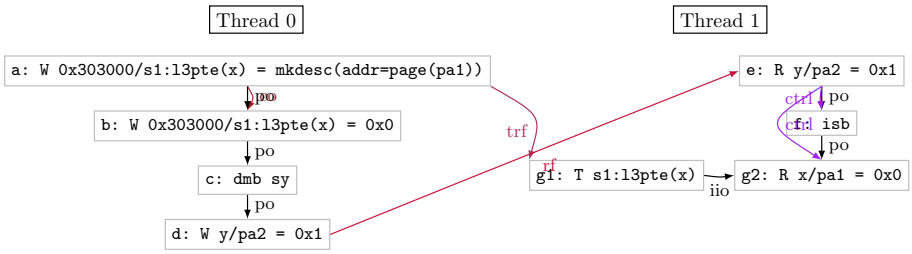
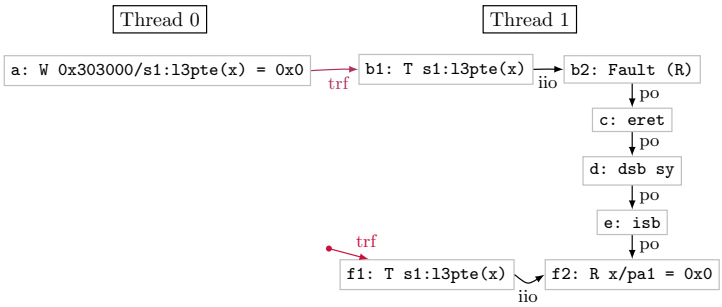


Figure 8.31: Test MP.RT.inv+poloc-dmb+ctrl-isb

AArch64	CoTfT+dsb-isb
Initial State	
0:R0=0b0 0:R1=**pte3**(x)	1:R1=x 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1; x |-> pa1; x ↦ invalid; y |-> pa1; *pa1 = 0; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1]	**LDR** X2,[X1] **MOV** X0,X2 **DSB** SY **ISB** **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Allow	



The earlier translation read (b1) reads from the new invalid entry, reading from memory (as it cannot have been in the TLB), but a later translation read (f1) of the same location can still potentially see a stale cached entry.

Figure 8.32: Test CoTfT+dsb-isb

earlier events witnessed a TLB miss.

Break-before-make and restrictions We will see later that the ability to have multiple cached entries for a single address causes problems for software managing coherence, and imposes extra restrictions on software practice. This means that, in general, the effects of the micro-TLBs are restricted to only those combinations that do not cause *break-before-make violations* (see §8.6.5).

8.5.4 Partial caching of walks

TLBs need not cache entire virtual to physical translations. Instead, they are free to cache any subset of the reads from the walk separately.

Caching up to last-level table The most common kind of caching structure we are aware of in microarchitecture is the walk cache (see §8.3.1). Traditionally a TLB would store entire virtual to physical mappings, making it fast to lookup the translation (often a single cycle), but there was limited space and induced heavy burden on a TLB miss or TLB invalidation. Walk caches store the last-level table entry, allowing TLB invalidation of leaf entries and TLB misses to re-use a prefix of the walk and perform a minimal number of accesses. This can be seen in the MP.RTT.inv3+dmb-dmb+dsb-isb test [Figure 8.33], where a walk cache could allow the table entry to be cached separately from the last-level entry, allowing the last translation read to read from a much newer value.

Caching of whole translation A common configuration for the TLB is to cache whole translation walks, from virtual to physical. This kind of caching has an important caveat: there is no requirement for the TLB to remember the intermediate physical address of any stage 2 translations that were done during the walk, including the final stage 2 walk of the access address itself. This means that TLB invalidations by IPA might not remove all the cached data associated with a cached entry for that IPA, if there is a whole cached translation which used that entry. **TODO: ?REF?**.

Independent caching of IPAs In a two-stage regime, the virtual addresses are first translated into intermediate physical address. The secondary translations based on the intermediate physical addresses, either of the final output address or of any of the intermediate table addresses, may be cached in the TLB without remembering the originating virtual address.

This means these cached translations may be recalled for translations of different virtual addresses.

AArch64	MP.RTT.inv3+dmb-dmb+dsb-isb
Initial State	
0:R0=0b0 0:R1=**pte2**(x) 0:R2=**mkdesc3**(oa=pa1) 0:R3=**pte3**(x) 0:R4=0b1 0:R5=y	1:R1=y 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00

virtual x y;
physical pa1 pa2;
assert x[48..21] = y[48..21];!
x |-> invalid;
x ↦ pa1;
x ↦ invalid at level 2;
y |-> pa2;
*pa1 = 0;
*pa2 = 0;
identity 0x1000 **with code**;	

Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3] **DMB** SY **STR** X4,[X5]	**LDR** X0,[X1] **DSB** SY **ISB** **MOV** X2,#1 **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Allow	

The translation-read of the level 2 entry for x (i1) can read from stale writes from a translation that the subsequent level 3 translation-read (i2) does not read from, as the level 2 entry could have been cached in the 'TLB' (in this case, a co-located 'walk cache' structure), while the level 3 entry gets read from memory. **TODO: PS: explain the magic numbers and tfr edge a bit.**

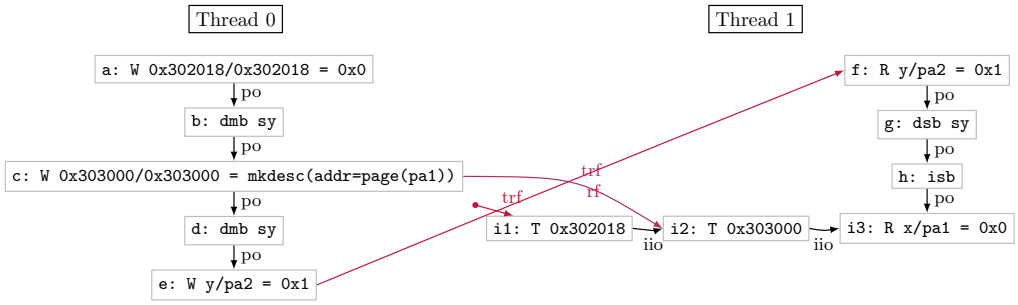


Figure 8.33: Test MP.RTT.inv3+dmb-dmb+dsb-isb

In addition, pre-fetching may perform translations of arbitrary IPAs. This means that any cached translations might not correspond to any valid whole translation table walk, but may still be used during such walks.

This is most clear in ROT.invs1+dmb2 [Figure 8.34], where, although the IPA was never reachable from the stage 1 translations, the old IPA to PA mapping was cached and used later.

Caching of individual entries Architecturally, Arm wish to allow many more implementations of TLBs and translation caching structures than currently known hardware contains.

The weakest variation on this is allowing each individual translation table entry to be cached separately and independently.

One could construct litmus tests for each of the possible combination of translation table entries, but there would be overwhelmingly many of these, or even a 'most relaxed' version where every translation table entry comes from different previous translations, but this would be too large to show here. So, for simplicity I show just one of them here, ROT.inv2+dmb [Figure 8.35]; where the last-level entry came from a newer value than the previous levels.

8.5.5 Reachability

One key property that the TLB must have is that it may only cache translation table entries which are *reachable* from a translation in the current context. That is, it can only cache an entry which is the result of a valid translation table walk, either using values from memory or other valid translation table entries from the TLB, using the current system register state.

This means that writes coherence-before the most recent write at the time a translation table entry location becomes reachable are not visible to the walker, and cannot have been cached in any TLB.

This is captured in the RUE+isb [Figure 8.36] ("Read-unreachable-entry") test, which is forbidden as the write to the translation table from before the time the location becomes reachable by translation table walkers cannot have been cached in any TLBs, or read from by any spontaneous walks.

This area is currently under discussion with Arm.

8.6 TLB maintenance

Recovering coherence for translation reads in the presence of TLB caching can be achieved through the use of TLB *maintenance* instructions: namely the `TLBI` ("TLB invalidate" instruction).

TLB maintenance generally causes two microarchitectural effects: erasing stale entries from the TLB, ensuring future TLB fills (for example, due to a translation read) will see the coherent value from memory; and, discarding any partially executed instructions, on other cores, which had already begun execution using a stale entry but had not yet finished executing. We will explore both of these effects and the subtle interaction with other parts of the virtual memory systems architecture in more detail throughout this section.

8.6.1 Recovering coherence

We saw earlier, in Section 8.5.1, that stale values cached in the TLB can cause coherence violations in the translation, for example, in the CoWinvT+dsb-isb test [Figure 8.30]. By inserting the correct `TLBI` sequence into that test, we produce a new test, CoWinvT.EL1+dsb-tlbi-dsb-isb [Figure 8.37], which is now forbidden.

There are many flavours of `TLBI` that could have been inserted into this test, the one in the figure is `TLBI VAE1`, or, TLB invalidation by virtual address, for the EL1&0 translation regime. Using a `TLBI`-by-VA means the programmer has to provide the virtual page to invalidate, and the `TLBI` only affects addresses for that specific invalidated entry, not all of them.

Using the incorrect `TLBI` leads to insufficient invalidation occuring. For example, if in the aforementioned CoWinvT.EL1+dsb-tlbi-dsb-isb the TLBI had the wrong page, then it would have no effect and the test would remain allowed.

AArch64	ROT.invs1+dmb2
Initial State	
0:R0=**mkdesc3**(oa=pa1) 0:R1=**pte3**(x, s2_table) 0:R2=0b0 0:R3=**pte3**(x, s2_table) 0:R4=**mkdesc3**(oa=ipa1) 0:R5=**pte3**(x) 0:PSTATE.EL=0b01	1:R1=x 1:VBAR_EL1=0x1000 1:VBAR_EL2=0x2000
physical pa1; **intermediate** ipa1; x |-> invalid at level 2; x ↦ ipa1; ipa1 |-> pa1; ipa1 ↦ invalid; *pa1 = 1; **identity** 0x1000 **with code**; **identity** 0x2000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3] **DMB** SY **STR** X4,[X5]	**MOV** X0,#0 **LDR** X0,[X1]
	Thread 1 EL1 Handler
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Thread 1 EL2 Handler
	0x2400: **MRS** X13,ELR_EL2 **ADD** X13,X13,#4 **MSR** ELR_EL2,X13 **ERET**
Final State	
1:X0=1	
Allow	

The translation read of the stage 2 leaf entry for x (f2) can read from an old cached version, from the write (a) even though it was not reachable by any translation table walk for any VA, as the IPA it maps was not mapped by any stage 1 tables before it was overwritten by (b).
This test relies on translation table walks being naturally ordered (by `iio`), see §8.4.8.

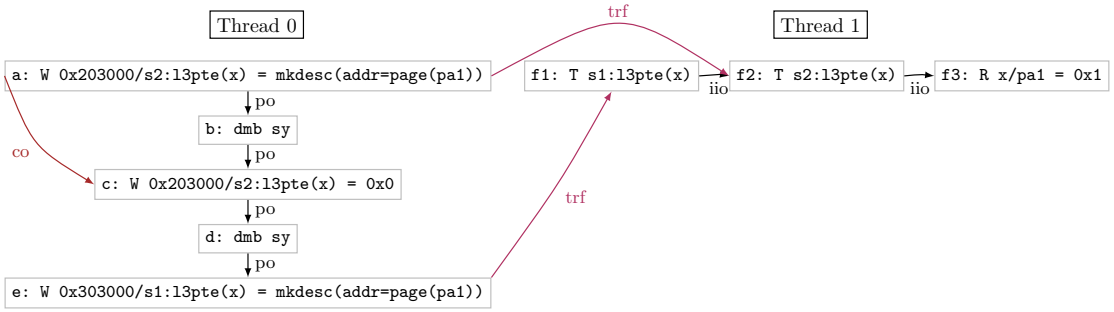


Figure 8.34: Test ROT.invs1+dmb2

AArch64	ROT.inv2+dmb
Initial State	
0:R0=0b0 0:R1=**pte3**(x, new_table) 0:R2=**mkdesc2**(table=0x283000) 0:R3=**pte2**(x) 0:PSTATE.EL=0b01	1:R1=x 1:VBAR_EL1=0x1000

physical pa1;
intermediate ipa1;
assert pa1 == ipa1;
ipa1 |-> pa1;
x |-> invalid at level 2;
x ↦ table(0x283000) at level 2;
s1table new_table 0x280000 {
  x |-> ipa1;
  x ↦ invalid;
};
identity 0x1000 **with code**;	

Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3]	**MOV** X0,#1 **LDR** X0,[X1]
	Thread 1 EL1 Handler
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=0	
Allow	

The translation-read of the level 3 entry (d2) can read from a stale cached translation, which was cached before the write to the level 2 entry (c). Note that this test assumes that the original `new_table` was reachable (and therefore could be cached) before the write c. See §8.5.5 for a discussion on this.

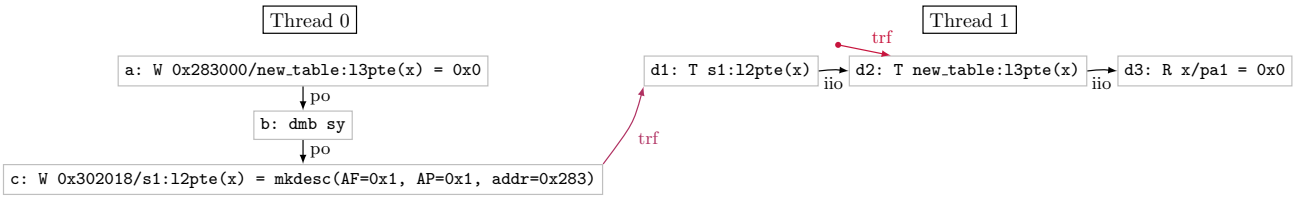
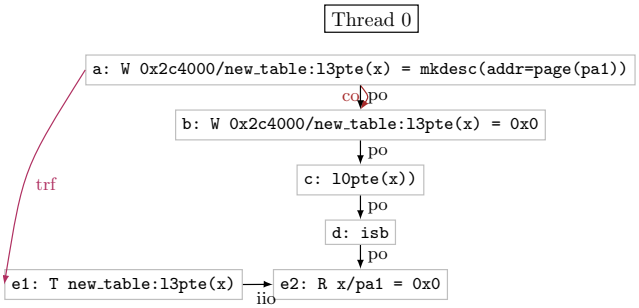


Figure 8.35: Test ROT.inv2+dmb

AArch64	RUE+isb
Initial State	
0:R0=**mkdesc3**(oa=pa1) 0:R1=0x0 0:R2=**pte3**(x, new_table) 0:R3=**ttbr**(asid=0x01, base=new_table) 0:R4=x 0:VBAR_EL1=0x1000 0:PSTATE.EL=0b01 0:PSTATE.SP=0b1	

intermediate ipa1;
physical pa1;
 	
*pa1 = 0;
 	
sltable new_table 0x2C0000 {
  identity 0x1000 **with code**;
  x |-> invalid;
  x ↦ pa1;
};
 	
identity 0x1000 **with code**;	

Thread 0	
01. **STR** X0,[X2] 02. **STR** X1,[X2] 03. **MSR** TTBR0_EL1,X3 04. **ISB** 05. **MOV** X1,#1 06. **LDR** X1,[X4]	
Thread 0 EL1 Handler	
01. 0x1200: 02. **MRS** X20,ELR_EL1 03. **ADD** X20,X20,#4 04. **MSR** ELR_EL1,X20 05. **ERET**	
Final State	
0:X1=0	

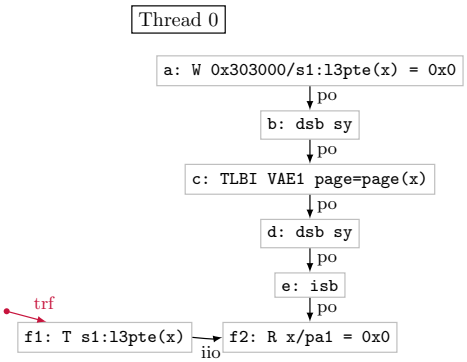


The write to the `new_table` translation table entry for `x` (a) is not visible at the point of the change of `TTBR` (c), and so the later translation table walk (e1) cannot read from it.

Note that isla currently does not do any kind of reachability analysis, and so does not forbid this test.

Figure 8.36: Test RUE+isb

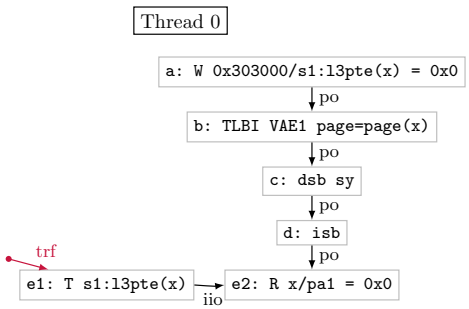
AArch64	CoWinvT.EL1+dsb-tlbi-dsb-isb
Initial State	
0:R0=0b0 0:R1=**pte3**(x) 0:R3=x 0:R5=**page**(x) 0:VBAR_EL1=0x1000 0:PSTATE.EL=0b01	
physical pa1 pa2; x |-> pa1; x ↦ invalid; **identity** 0x1000 **with code**;	
Thread 0	
STR X0,[X1] **DSB** SY **TLBI** VAE1,X5 **DSB** SY **ISB** **LDR** X2,[X3]	
Thread 0 EL1 Handler	
0x1000: **MOV** X2,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**	
Final State	
0:X2=0	
Forbid	



The read of the translation table entry for `x` (f1) is required to happen after the earlier store (a), because of the intervening `dsb sy; isb` sequence (d and e), and cannot be satisfied from the TLB, because of the `TLBI` (c), forbidding it from still seeing a stale value. Note that `TLBI` instructions can only be executed from EL1, so this test starts execution at EL1 rather than the usual default of EL0.

Figure 8.37: Test CoWinvT.EL1+dsb-tlbi-dsb-isb

AArch64	CoWinvT.EL1+tlbi-dsb-isb
Initial State	
0:R0=0b0 0:R1=**pte3**(x) 0:R3=x 0:R5=**page**(x) 0:VBAR_EL1=0x1000 0:PSTATE.EL=0b01	
physical pa1 pa2; x |-> pa1; x ↦ invalid; **identity** 0x1000 **with code**;	
Thread 0	
STR X0,[X1] **TLBI** VAE1,X5 **DSB** SY **ISB** **LDR** X2,[X3]	
Thread 0 EL1 Handler	
0x1000: **MOV** X2,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**	
Final State	
0:X2=0	



The TLBI (b) can be re-ordered with program-order earlier events, due to the lack of DSBs ordering it after them, allowing the store (a) to happen later, letting the final translation read (e1) still see the old stale translation.

Figure 8.38: Test CoWinvT.EL1+tlbi-dsb-isb

FEAT_nTLBPA

Armv8.4-A introduced a new optional Arm feature, FEAT_nTLBPA [1, A2.2.1 (p79)] .

This feature adds a field to the memory model feature register (AA64MMFR1_EL1) which can identify whether the current processor's TLB (and related microarchitectural caching structures) may contain non-coherent copies of stage 1 entries indexed by those entries intermediate physical address. Microarchitecturally, this corresponds to there being non-coherent caches associated with the TLB, which must be flushed on a TLBI.

These caches would allow TLB misses to read from a non-coherent cache, thus not seeing the most up-to-date value from the coherent storage subsystem like described in §8.4.

Note that the text in the reference manual is a little ambiguous, the entry in A2.2.1 describes it as a "mechanism to identify if [TLB caching] does not include non-coherent caches [of old translation entries] since the last completed TLBI". This change adds a field to the register, whose reserved value in Armv8.0 corresponds to the non-coherent caches existing. This implies that implementation of the feature is not only the existence of the runtime identification register's field, but additionally that its value is 0b0001 (that is, that non-coherent caches do not exist). This further implies that in processors without FEAT_nTLBPA one should assume that TLBs may contain non-coherent caching structures.

8.6.2 Thread-local ordering and TLBI

TLB maintenance instructions are not naturally locally ordered with respect to other instructions in the instruction stream, this means that they can be re-ordered with other instructions. To ensure they are synchronized with other instructions, the programmer can use the DSB barrier instruction to order instructions before and after it.

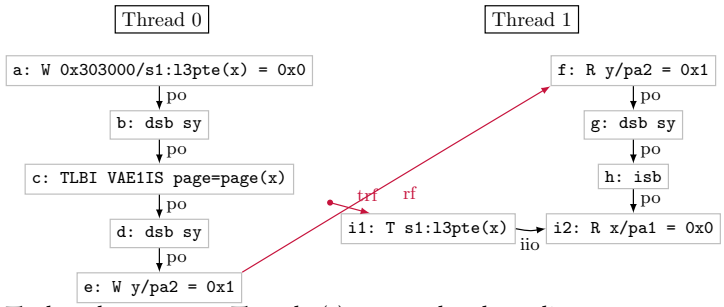
Leaving out one, or both, of the DSBs around the TLBI leads to insufficient ordering around the TLBI and allows the invalidation to occur at the wrong time. For example, the CoWinvT.EL1+tlbi-dsb-isb test [Figure 8.38] is allowed as the initial write and TLBI may be re-ordered, negating the architectural effect of the TLBI.

TODO: talk about FEAT_ETS

8.6.3 Broadcast

Arm provide broadcast variants of the TLBI instructions. These are generally suffixed with the letters IS (for "Inner-shareable").

AArch64	MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb
Initial State	
0:R0=0b0 0:R1=**pte3**(x) 0:R2=0b1 0:R3=y 0:R4=**page**(x) 0:PSTATE.EL=0b01	1:R1=y 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1 pa2; x |-> pa1; x ↦ invalid; y |-> pa2; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DSB** SY **TLBI** VAE1IS,X4 **DSB** SY **STR** X2,[X3]	**LDR** X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Forbid	



The broadcast TLBI on Thread 0 (c) ensures that the earlier unmapping (a) is seen by the ordered later translation read on Thread 1 (i1), by ensuring Thread 1's local TLB is cleaned of any stale entries for x.

Figure 8.39: Test MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb

Broadcast TLBIs, sometimes referred to as TLB *shootdowns*, allow one processor to perform maintenance on another core's TLB.

This is in contrast to other systems, such as for IBM's Power architecture, where maintenance of other cores must be achieved in software through the use of only thread-local invalidation instructions.

TLB invalidation on another core One of the simplest examples is a message passing invalidation pattern, where the old entry is removed and a message is sent to another core. This can be seen in the MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb test [Figure 8.39].

Instruction restarts Broadcast TLBIs must do more than touch the other thread's TLB. If the other processor had already performed translation, using the old stale value, but has not yet finished execution, then that instruction must be restarted.

This ensures that Arm broadcast TLBIs have the same behaviour as the traditional software IPI-based shootdown (With context synchronization); but also provides a needed security guarantee.

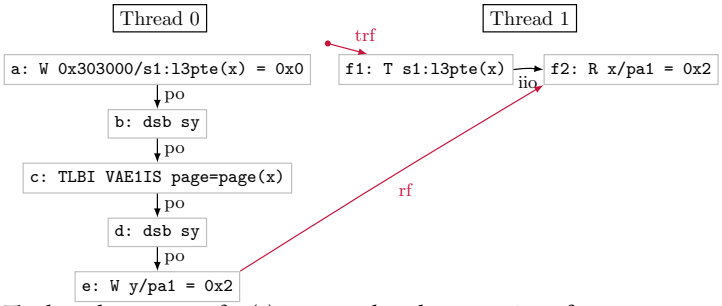
If a mapping is taken away from a process, then future writes to the physical location it used to map to, should not be visible to that process anymore.

This guarantee is captured in the RBS+dsb-tlbiis-dsb [Figure 8.40] (**Read-broken-secret**) test. Once a mapping has been *broken*, and sufficient TLB maintenance performed, any future reads or writes to the original physical location will not be visible through that mapping anymore. Note, however, that this does not mean that instructions which have already completed their execution will be restarted, even if they occur after an earlier restarted instruction. This can be seen in the RBS+dsb-tlbiis-dsb+poloc test [Figure 8.41], where the program-order later load can see the old value, even after the first faults.

While here I describe things in terms of instruction restarting, these behaviours can be (and presumably are) implemented in terms of waiting: instead of the TLBI forcibly restarting instructions that already started but haven't finished, the TLBI can simply wait for them to complete. This phrasing of waiting for completion is how this process is described in the Arm ARM [1, D5.10.2 (p4928)] .

Atomic TLBIs In the previous RBS-shaped tests, I describe the behaviour in terms of writes that occur 'before' the TLBI.

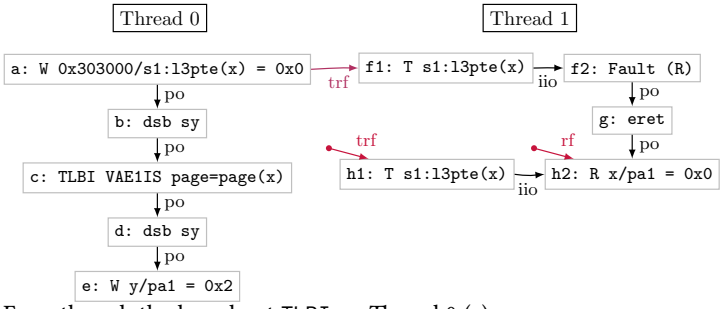
AArch64	RBS+dsb-tlbiis-dsb
Initial State	
0:R0=0b0 0:R1=**pte3**(x) 0:R5=**page**(x) 0:R2=0x2 0:R3=y 0:PSTATE.EL=0b01	1:R1=x 1:VBAR_EL1=0x1000
physical pa1; x |-> pa1; x ↦ invalid; y |-> pa1; *pa1 = 0; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DSB** SY **TLBI** VAE1IS,X5 **DSB** SY **STR** X2,[X3]	**LDR** X0,[X1]
	Thread 1 EL1 Handler
	0x1400: **MOV** X0,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=2	
Forbid	



The broadcast TLBI of x (c) ensures that the execution of the load of x in Thread 1 either entirely executes using the old translation and finishes before the TLBI does, or begins execution after the TLBI finishes.

Figure 8.40: Test RBS+dsb-tlbiis-dsb

AArch64	RBS+dsb-tlbiis-dsb+poloc
Initial State	
0:R0=0b0 0:R1=**pte3**(x) 0:R5=**page**(x) 0:R2=0x2 0:R3=y 0:PSTATE.EL=0b01	1:R1=x 1:R3=x 1:VBAR_EL1=0x1000
physical pa1; x |-> pa1; x ↦ invalid; y |-> pa1; *pa1 = 0; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DSB** SY **TLBI** VAE1IS,X5 **DSB** SY **STR** X2,[X3]	**MOV** X0,#1 **LDR** X0,[X1] **MOV** X2,#1 **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	



Even though the broadcast TLBI on Thread 0 (c) ensures that not-yet-completed instructions using the old mapping are restarted, it does not require that the second load of x in Thread 1 (h) be restarted if it has already satisfied its value, as that value must have come from a write before the TLBI.

Figure 8.41: Test RBS+dsb-tlbiis-dsb+poloc

Microarchitecturally a TLBI instruction is very non-atomic: it sends messages to all other cores, performs some action, and sends messages back to the originating core. The program-order earlier DSB ensures that program-order earlier instructions are complete before sending the messages. The program-order later DSB ensures that all program-order later instructions wait for those messages to return.

The presence of these DSBs ensure that the TLBI's effect happens entirely at that point in the instruction stream, and cannot be broken up and re-ordered amongst the other instructions in the stream. This, coupled with the fact that these messages *strengthen* and never weaken the behaviour of other cores, means that you cannot observe a partial TLBI effect. So long as the programmer takes care to maintain the required thread-local ordering.

Because of this, we can think of the TLBI as executing either before an instruction or after an instruction, but do not need to consider a TLBI executing in the middle of another instruction. This allows us to simplify things, fitting TLBIs into a (generalised) coherence order, with other writes occurring either before or after.

8.6.4 Virtualization

Throughout this section we have considered tests for stage 1 translation with virtual mappings. But many of these questions and behaviours also apply to the stage 2 intermediate physical mappings, with some key differences.

Virtual to physical and IPA caches The existence of TLBs that cache virtual to physical mappings (§8.5.4) complicates the TLB maintenance sequence required for changes to the intermediate physical mappings.

When invalidating stale second stage entries from the TLB, it is required for the programmer to do *two* sets of invalidations: first one TLB invalidation to remove any of the old entries for the old IPA to PA, then, perhaps surprisingly, a second TLB invalidation is needed to remove any stale whole translation, VA to PA mappings or any combination thereof, as these could have indirectly cached the result of a second stage translation without remembering the IPA.

This can be seen in MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb [Figure 8.42], where invalidation of *just* the IPA is not enough. Adding an invalidation of the VA (or all VAs), like in MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb [Figure 8.43], ensures that later translations cannot see the stale value anymore.

AArch64	MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb
Initial State	
0:R0=0b0 0:R1=**pte3**(ipa1, s2_table) 0:R2=0b1 0:R3=z 0:R4=**page**(ipa1) 0:PSTATE.EL=0b10	1:R1=y 1:R3=x 1:VBAR_EL2=0x2000 1:PSTATE.EL=0b00

physical pa1 pa2;
intermediate ipa1 ipa2;
x |-> ipa1;
ipa1 |-> pa1;
ipa1 ↦ invalid;
y |-> ipa2;
ipa2 |-> pa2;
z |-> pa2;
identity 0x2000 **with code**;
*pa1 = 0;
*pa2 = 0;	

Thread 0	Thread 1
STR X0,[X1] **DSB** SY **TLBI** IPAS2E1IS,X4 **DSB** SY **STR** X2,[X3]	**LDR** X0,[X1] **DSB** SY **ISB** **MOV** X2,#1 **LDR** X2,[X3]
	Thread 1 EL2 Handler
	0x2400: **MRS** X13,ELR_EL2 **ADD** X13,X13,#4 **MSR** ELR_EL2,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Allow (if not ETS)	

Despite the TLB invalidation of the stale IPA (c), a later stage 2 translation-read of that IPA (i1) can still see the old stale value.

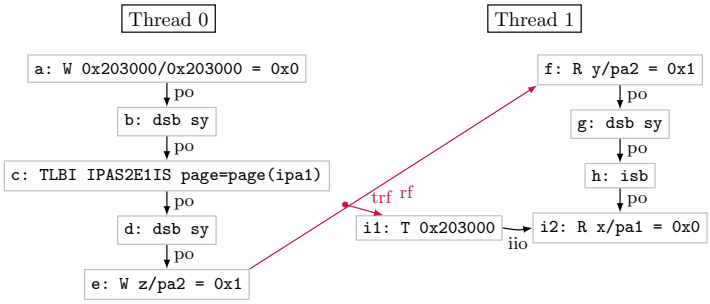


Figure 8.42: Test MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb

AArch64	MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb
Initial State	
0:R0=0b0 0:R1=**pte3**(ipa1, s2_table) 0:R2=0b1 0:R3=z 0:R4=**page**(ipa1) 0:PSTATE.EL=0b10	1:R1=y 1:R3=x 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:VBAR_EL2=0x2000

physical pa1 pa2;
intermediate ipa1 ipa2;
x |-> ipa1;
ipa1 |-> pa1;
ipa1 ↦ invalid;
y |-> ipa2;
ipa2 |-> pa2;
z |-> pa2;
identity 0x2000 **with code**;
*pa1 = 0;
*pa2 = 0;	

Thread 0	Thread 1
STR X0,[X1] **DSB** SY **TLBI** IPAS2E1IS,X4 **DSB** SY **TLBI** VMALLE1IS **DSB** SY **STR** X2,[X3]	**LDR** X0,[X1] **DSB** SY **isb** **LDR** X2,[X3]
	Thread 1 EL2 Handler
	0x2400: **MOV** X2,#1 **MRS** X13,ELR_EL2 **ADD** X13,X13,#4 **MSR** ELR_EL2,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Forbid	

By performing TLB invalidation of the stage 1 entries (e) after invalidating the stage 2 ones (c1), it is guaranteed that the later translation-read (k1) cannot see the old stale value anymore.

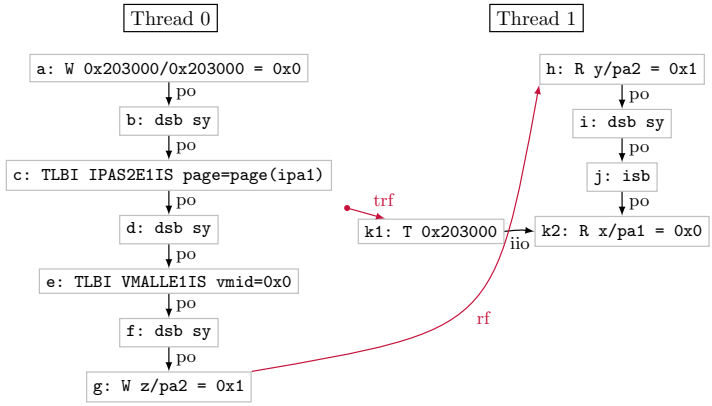


Figure 8.43: Test MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb

8.6.5 Break-before-make

TLBs are not required to store only a single cached translation for a given address. There may, in general, be multiple valid translations cached in the TLB.

To avoid this possibility, the architecture provides a *break-before-make* sequence, which will ensure that there cannot be two cached translations existing in the TLB at the same time.

The architecture requires break-before-make when writing to the translation tables to update an already valid entry with a new valid entry, and the change involves any of the following[1]:

- ▷ A change in output address, if the new or old entry is writeable.
- ▷ A change in output address, if the new and old locations have different contents.
- ▷ A change in memory type.
- ▷ A change in block size (e.g. replacing a page of 4KiB leaf with a 2MiB block mapping).

For those cases where break-before-make is required, the programmer must:

(1) write an invalid entry to overwrite the currently valid translation table entry in memory;
(2) perform a `dsb sy` (or equivalent);
(3) perform any TLB maintenance required to sufficiently invalidate the old entry from any TLB(s) required;
(4) perform a `dsb sy` (or equivalent);
(5) write the new valid translation table entry, overwriting the old invalid entry.

Litmus test For completeness, the BBM+dsb-tlbiis-dsb [Figure 8.44] gives possibly the simplest valid to valid concurrent update test,

Break-before-make violations

Architecturally, there is no hard requirement to perform break-before-make. Failure to do so simply leads to a degraded state, defined by ConstrainedUnpredictable behaviour.

The Arm reference manual does make it clear that failure to perform break-before-make when required can lead to failure of single-copy atomicity, coherence or even the full breakdown of uniprocessor semantics. While the reference manual does not give motivation for this, we can speculate that this is to allow hardware to perform multiple translations during execution of the instruction, for example, during hazard checking. As such, we do not try to give a full picture of ConstrainedUnpredictable behaviour arising from break-before-make not being followed.

Understanding ConstrainedUnpredictable in full is future work, but a quick summary might be 'any behaviour that this program could have performed if it wanted to'. That is, an instantenous change in the state to a random new state that would have been reachable by executing arbitrary code at that same exception level, security state and translation regime.

8.6.6 ASIDs and VMIDs

In an effort to reduce the expense of TLB maintenance the architecture provides a mechanism to separate out the address spaces by tagging translations with *address space identifiers* (or ASIDs). These ASIDs allow TLB entries to be tagged with only the address space they are used with, and allow TLB maintenance operations to selectively target only the address space being updated.

Crucially, this allows software to switch between address spaces without having to invalidate the TLB.

This idea is extended not just to address spaces at EL1 (used primary for the operating system and its processes), but to EL2 with *virtual machine identifiers* (or VMIDs). These VMIDs serve the same function as ASIDs, giving IDs to address spaces, except in this case IDs to second-stage IPA to PA address spaces.

[1] See the Arm ARM "TLB maintenance requirements and the TLB maintenance instructions" [1, D5.10.1 (p4913)] . for the full list of conditions.

AArch64	BBM+dsb-tlbiis-dsb
Initial State	
0:R0=0b0 0:R1=**pte3**(x) 0:R2=**mkdesc3**(oa=pa2) 0:R4=0b1 0:R6=**page**(x) 0:PSTATE.EL=0b01	1:R1=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1 pa2; x |-> pa1; x ↦ invalid; x ↦ pa2; **identity** 0x1000 **with code**; *pa2 = 2;	
Thread 0	Thread 1
STR X0,[X1] **DSB** SY **TLBI** VAE1IS,X6 **DSB** SY **STR** X2,[X1]	**LDR** X0,[X1]
	Thread 1 EL1 Handler
	0x1400: **MOV** X0,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=0	
Allow	

The update of the translation table entry for x in Thread 0 follows the break-before-make sequence, first *breaking* x (a), then performing the necessary TLBI sequence (b-c-d), before *making* x be the new mapping (e). This ensures the concurrent access in Thread 1 is guaranteed to see either the old value, the intermediate broken page (and so a page fault), or the new value. This test is the variant whose final state asserts that the old value was read.

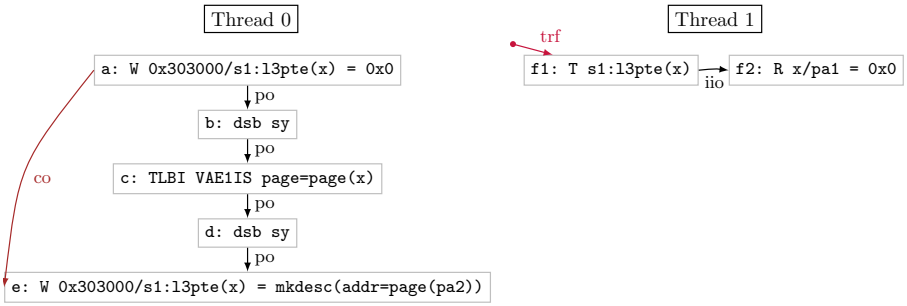


Figure 8.44: Test BBM+dsb-tlbiis-dsb

8.6.7 Access permissions

Accesses which result in permission faults can have been satisfied from the TLB, and writes which update translation table entries AP field can be cached in the TLB.

Translations can give rise to permission faults. These are unlike translation faults, in that, they are based not just upon the descriptor read, but also on the *kind* of access requested: whether a read, or a write.

Accesses which result in permission faults result in exceptions, much like translation faults do, but may have been read from the TLB. This can clearly be seen in the CoWinvTp.ro+dsb-isb test [Figure 8.45], where ordered after a write to the translation tables a permission failure is experienced, whose descriptor must have come from the TLB.

Multiple cached entries The changing of access permissions not necessarily being break-before-make violations allows us to observe multiple cached entries within the TLB. It is permitted for these entries to exist simultaneously.

When reading from the TLB, and there existing multiple entries for the same input address, it is allowed for the hardware to generate a *TLB conflict abort*. These aborts are reported as data aborts.

If the hardware does not generate a conflict abort, then translation reads of that address are ConstrainedUnpredictable, nondeterministically able to read one or the other or an "amalgamation" of the values [1, K1.2.3 (p11243)] .

Here there seems a contradiction:it is not required to perform break-before-make, but there is no requirement that only one entry be cached in the TLB. We can side step this issue by constructing a test that only changes a single bit of the descriptor, in a way that is not a break-before-make violation, and therefore avoiding any questions about how 'amalgamation' of entries happens. This can be seen with the MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb test [Figure 8.46], where the existence of multiple cached entries in the TLB allows multiple translation-reads to read from different stale writes.

Atomic TLB reads Existence of multiple cached translation table entries in the TLB, without break-before-make violations, introduces the question of whether those TLB fills and subsequence TLB reads must read from entire single-copy atomic writes of the original translation table entries (much like a read of memory would) or whether a translation read can read from a mix of different writes. RMD+dmb [Figure 8.47] ("Read-mixed-descriptor") shows that translation reads cannot read partially read from a write, it must read from the entire write or none of it.

AArch64 CoWinvTp.ro+dsb-isb

Initial State

```
0:R0=0x0
0:R1=pte3(x)
0:R2=0x1
0:R3=x
0:VBAR_EL1=0x1000
0:PSTATE.SP=0b0
```

```
physical pa1;
x |-> pa1 with [AP = 0b11] and default;
x ↦ invalid;
*pa1 = 0;
identity 0x1000 with code;
```

Thread 0

```
STR X0,[X1]
DSB SY
ISB
MOV X13,#0
STR X2,[X3]
```

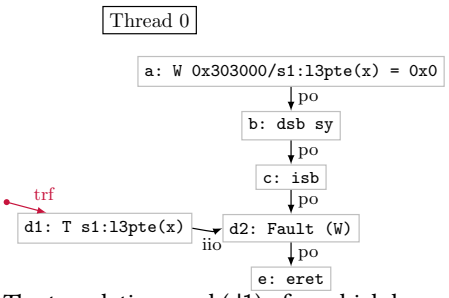
Thread 0 EL1 Handler

```
0x1400:
// read ESR_EL1.ISS to see if Permission or Tra
    MRS X14,ESR_EL1
    AND X14,X14,#0b1111
    CMP X14,#0b1111
    MOV X15,#1 // Permission
    MOV X16,#2 // Translation
    CSEL X13,X15,X16,eq
    MRS X20,ELR_EL1
    ADD X20,X20,#4
    MSR ELR_EL1,X20
    ERET
```

Final State

```
0:X13=1
```

Allow



The translation-read (d1) of x, which happens after the program-order earlier store to the translation tables (a) because of the intervening `dsb; isb` sequence (b-c), can read from a stale value and result in a permission fault, as the read-only entry from the initial state may be cached in the TLB.

Figure 8.45: Test CoWinvTp.ro+dsb-isb

AArch64	MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb
Initial State	
0:R0=**mkdesc3**(oa=pa1, AP=0x3) 0:R1=**pte3**(x) 0:R2=0b0 0:R3=**pte3**(x) 0:R4=0b1 0:R5=y	1:R1=y 1:R4=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0

physical pa1 pa2;
x |-> pa1 **with** [AP = 0b11] and default;
x ↦ pa1 **with** [AP = 0b10] and default;
x ↦ invalid;
y |-> pa2;
*pa1 = 0;
identity 0x1000 **with code**;	

Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3] **DMB** SY **STR** X4,[X5]	**LDR** X0,[X1] **DSB** SY **ISB** **LDR** X13,[X4] **MOV** X2,X13 **DSB** SY **ISB** **LDR** X13,[X4] **MOV** X3,X13
	Thread 1 EL1 Handler
	0x1400: // read ESR_EL1.ISS to see **MRS** X14,ESR_EL1 **AND** X14,X14,#0b1 **CMP** X14,#0b1111 **MOV** X15,#1 // Perm **MOV** X16,#2 // Tran **CSEL** X13,X15,X16 **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**
Final State	
1:X0=1 & 1:X2=1 & 1:X3=0	
Allow	

The first translation-read of x (i1) reads from the write that removes read permissions (a) and this write must have come from the TLB because of the intervening invalidation (c), message pass (e-f), and `dsb sy` ; `isb` sequence (g-h). The later translation-read of x (m1) can still see an even older value with read permissions, from the initial state, as it may *also* have been cached in the TLB.

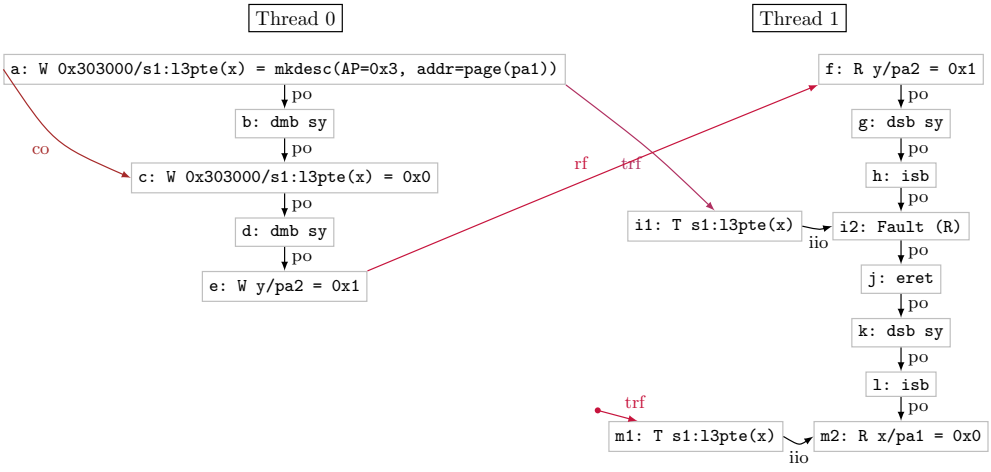


Figure 8.46: Test MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb

AArch64	RMD+dmb
Initial State	
0:R0=**mkdesc3**(oa=pa2, AP=0b10) 0:R1=**pte3**(x) 0:R2=0x1 0:R3=y	1:R1=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0

physical pa1 pa2;
x |-> pa1 **with** [AP = 0b11] and default;
x ↦ pa2 **with** [AP = 0b10] and default;
y |-> pa2;
*pa1 = 0;
*pa2 = 1;
identity 0x1000 **with code**;	

Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3]	**MOV** X0,#0 **LDR** X0,[X1]
	Thread 1 EL1 Handler
	0x1400: **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**
Final State	
1:X0=1	
Forbid	

The translation-read of x (d1) cannot read from both the 64-bit single-copy atomic write a as well as from the initial state. Note that this test does not, as far as we can see, violate the break-before-make requirements, as currently prescribed by the Arm manual, as the contents in memory of both locations pa1 and pa2 are the same at the time of the write to the translation tables.
This diagram was generated by hand, as isla does not generate a candidate execution of this shape.

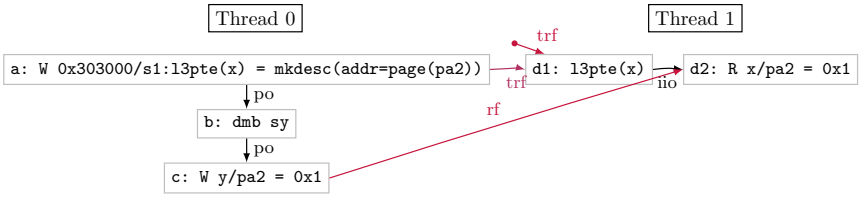


Figure 8.47: Test RMD+dmb

8.7 Context synchronisation

There are many operations which change the current context the system is in. We will focus in on two of these: taking and returning from exceptions, and writing to system registers.

These actions can change the context that the system is executing in: the current exception level, the translation regime, the translation table base, the ASID or VMID, and a variety of other system configuration state.

8.7.1 Relaxed system registers

So far, in this and previous work, register reads and writes have been completely coherent: instructions program-order after a write to a register will always read from that write (or an intervening write) when it reads that register. System registers break this guarantee.

Arm System registers may require the programmer to insert explicit synchronization, as stated clearly in the Arm reference manual [1, D13.1.2 (p5235)] :

> Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:
> - ▷ Any data dependencies between the instructions, including read-after-read dependencies, are respected.
> - ▷ The reads to the register do not occur earlier than the most recent Context synchronization event to its architectural position in the instruction stream.

This means a read of a system register might not read from the most recent write to that system register.

To ensure that writes to system registers are seen by program-order later reads, the programmer can ensure that a *Context synchronization* event occurs. These are typically things which flush the pipeline causing future instructions to restart: The `ISB` instruction and taking and returning from exceptions.

There are two important caveats: (1) this does not apply to non-System registers, such as special-purpose or general-purpose registers, and they never require synchronization; and (2), the synchronization required for System registers depends on the *kind* of accesses.

There are typically two kinds of accesses to System registers: direct, and indirect. Direct accesses are the way we think of registers: instructions which specifically read or write to those registers. Indirect accesses happen when an instruction which does not explicitly mention the register by name performs an access, a read or a write, to that register, during the execution of its behaviour.

Because of the out-of-order nature of the pipeline, these indirect register reads and writes may occur out-of-order with respect to any program-order earlier direct reads or writes of that register. This means that before any direct read, and after any direct write, the programmer must perform a context-synchronizing event to ensure that these direct accesses occur in-order with respect to other indirect accesses. The programmer does not have to insert context-synchronization *after* any direct read, as it is guaranteed that register reads or writes cannot be affected by program-order later accesses.

System register ASL In the previous chapter we explored the Arm ASL code for the translation table walk and for one of the store instructions. We saw that this ASL code reads from system registers (as indirect reads).

A naive attempt at a first interpretation of the relaxed semantics is to allow these reads to read-from the most recent indirect write and any program-order later direct writes since the last context synchronization event. However, this would not give the correct behaviour.

The Arm ASL is not written to accommodate relaxed system register behaviours. It leaves questions open about whether these registers can be redundant re-read during execution, whether the instruction reads the entire register at once or piecemeal over the course of execution, and whether repeated accesses to the same register within an instruction are able to read-from different writes. These questions, and others, are still under discussion with Arm.

We will see later in §**TODO: ?REF?** that we give a simple incomplete (and possibly unsound) interpretation in our model in the *pointed set* semantics of system registers, which allows the model to observe some of the known behaviours in this area, without yet fully exploring the architecture.

Caching of system registers in TLBs In addition to being out-of-order due to pipeline effects, some system registers may be indirectly cached within the TLB.

We have already seen one of these: the `MAIR` register. Direct writes to the `MAIR` may not be seen by program-order later translations, even after context-synchronization, as the translations may get their value from the TLB and the TLB may have stored a result which depended on the previous value of the `MAIR`, effectively causing a stale read of it at that point in the instruction stream.

To ensure that an update to the MAIR is observed by later translations therefore requires both TLB maintenance and context synchronization, in that order.

The registers which can be cached in this way, and the behaviours that arise from this caching, are still under current investigation with Arm.

8.8 Details likely to change

There have been a few places so far I've added words to the effect of 'this is currently under discussion with Arm'. In this section I will summarise those things which we know some things, but also know that it is is likely to change and the ways in which it will.

Caching of entries in the TLB The biggest change we are aware of to the model is a strengthening in caching of entries in the TLB. We have assumed that the TLB can cache any combination of translation table entries for a walk, and recall any cached combination as well. We are aware that Arm wish to strengthen this, to make the model TLB more in-line with the hardware TLB: essentially requiring the model TLB to behave as a walk cache, caching whole walks (or prefixes of walks) rather than individual entries.

TLB Invalidation We have primarily considered TLB invalidations of cached level 3 (that is, last level) entries. When invalidating entries higher in the table, they affect more of the address space (as described in §7.3.1), and so the TLB invalidation must affect addresses outside of just the page referenced. The model currently does not support this, this is a simple oversight and we believe not hard to update the model to handle this case.

More complex invalidation patterns, for example, zeroing and invalidating table entries, is still under discussion with Arm.

ETS We are aware of changes to the architecture regarding ETS. Every attempt has been made to try incorporate those changes into the model as we become aware, however, often they are changes driven by others and we only become aware as they are publically released.

It is very likely the parts of the model dedicated to ETS will gain new strength over the coming weeks and months, it is unfortunately not possible at this time to give a detailed description of what the final state of ETS will look like, partially for confidentiality reasons and partially because Arm have not yet decided.

System registers As previously described, the current state of relaxed system register reads and writes is unclear. We are in discussion with Arm on this aspect. It is not possible at this time to describe what the final model will look like, or what changes will need to be done to the model presented here.

Exceptions and context-synchronisation We are in discussion with Arm about the nature of exceptions and their context-synchronising nature and how this interacts with the memory model.

We believe the changes required to the model presented here will be minor, although they will probably be neither a relaxation nor a strengthening of the current model, but rather an incomparable change.

8.9 Contributions

We have now covered all the relaxed memory behaviours, and will, in the next chapter, move on to discuss the extant models created to capture those behaviours. But before that, it may at this point be unclear what the *contribution* of this chapter is. They come in three forms: (1) the attempt at some systematic coverage of the kinds of behaviours which systems software must account for; (2) the precise, formal description (in prose, and as litmus

tests) of those behaviours; and, (3) the clarification of the architecture where such behaviours were otherwise unclear.

Coverage of behaviours While this chapter attempts to systematically cover the behaviours we imagine software may try to rely on, starting from the basics of translation table walks and exploring the effects of out-of-order pipelines, caching, and barriers, we cannot claim it is *exhaustive*. As this is a manually compiled and curated list of behaviours, from reading the text and talking with architects, there are surely corner cases missed and software patterns overlooked. However, we believe we have covered those patterns known and required for the features we cover enough for software verification efforts of microkernels and hypervisors.

Clarification of architecture Attempts to clarify the architecture come primarily from the confidential discussions with architects. The behaviours discussed usually fell into one of three categories, whether they were clear already, needed further exploration or are, still, under invesitgation by Arm.

The first major category are those behaviours which were already clear and potentially covered in the architecture text. As alluded to right at the start of this chapter, these are not whole sections or sub-sections or even necessarily whole tests. The most obvious cases are §8.3.3 ('Invalid entries'), §8.2.1 ('Virtual coherence'), and §8.6.5 ('Break-before-make'). These are fundamental behaviours to the correctness of all modern systems software, and for which the architecture reference manual has clear words (at least, enough to cover the basic sequences software rely upon).

Most of the subsections fall into a more general category, of things that either had some associated reference materials, or was otherwise clear from discussion with architects, but for which further invesitgation was needed. This includes: forwarding (§8.4.4) and speculation (§8.4.5) for translation table walks; multi-copy atomic translation table walks (§8.4.7); intra-instruction ordering (§8.4.8,§8.4.9); micro-TLBs (§8.5.3) and partial walk caching (§8.5.4); a variety of `TLBI` questions (§8.6); and, system register accesses (§8.7.1).

Despite the work conducted here, from reading the architecture reference text, discussions with architects, and the testing of existing hardware, there are still many questions which are under current invesitgation with Arm. These include further questions about the scope of `TLBI`s, interaction with exceptions and interrupts, changes in cacheability, translations for instruction fetching, and relaxed system register accesses. Those areas will require more work before giving a concrete semantics.

Chapter 9

An axiomatic VMSA model

This chapter is based, in part, on: Relaxed virtual memory in Armv8-A [54] by Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Published in the proceedings of the 31st European Symposium on Programming (ESOP, 2022).

We now define a semantic model for Armv8-A relaxed virtual memory that, to the best of our knowledge, captures the Arm architectural intent for the questions discussed in Chapter 8, including Stage 1 and Stage 2 translation-table walks and the required TLB maintenance.

In §8 we described the design issues in microarchitectural terms, discussing the behaviour of translation table walks and TLB caching, along with the needs of system software. We now abstract from microarchitecture: constructing a model based on ordering between translation-read events and others, avoiding modelling TLBs and out-of-order pipelines directly.

This chapter will present this model, as an extention to the 'user-mode' Armv8-A axiomatic model presented in §**TODO: ?REF?**.

9.1 Extended candidate executions

The base Armv8-A axiomatic model is defined as a predicate over *candidate executions*, each of which is a graph with various events (reads, writes, barriers) and relations over them, notably the per-thread program order `po`, the per-location coherence order `co`, the reads-from relation `rf` from writes to reads, the `addr`, `data`, and `ctrl`-dependency subsets of `po`, and others.

We extend these candidates with both new events and new relations over those events, as well as modifying some of the original ones.

9.1.1 Candidate events

In addition to the events of the original model, we add the following events to the candidates:

- ▷ `T` for reads originating from architected translation-table walks.
  These roughly correspond to the actual satisfaction from memory which with TLBs may happen very early.
- ▷ `TLBI` events for each `TLBI` instruction, with a single such event per `TLBI` instruction, corresponding to the `TLBI` being completed on all relevant cores.
- ▷ `TE` and `ERET` events for taking and returning from an exception (these might not correspond to changes in exception level).
- ▷ `MSR` events for writes to relevant system registers, such as the `TTBR`.
- ▷ `DSB` events for `DSB` instructions.
- ▷ `Fault` events for translation and permission faults.

Translation-reads During execution of the ASL `TranslateAddress` function (§7.6) there will be many reads, which would usually generate `R` events. When those reads happen during the `TranslateAddress` call, they instead generate `T` events. This means that each translation table walk may generate up to 24 `T` events, before the instruction generates the (explicit) `R|W` event.

Alternative representations were explored, including leaving them as R events or collecting all reads into a single large translation event. But these options did not give the clarity and fine granularity we desired in the model, and would require more relations and axioms than presented here.

We also choose not to include TLB hits and misses in the model directly, but instead model the TLB as a relaxation of the values the walk can read from, much like normal R data memory read events and modelling load buffering, write gathering and caches.

We add a helper set, T_f, for translation reads which read-from a write whose value is even. That is, an entry whose invalid bit is 0. If a translation read results in a fault, either because it was an invalid entry and we get a translation fault, or because the access permissions of the resulting translation do not permit the kind of requested access and so result in a permission fault, the candidate will contain a Fault event (partitioned into Fault_t and Fault_p for translation and permission faults) in po order where the explicit memory event would have been. See text on obETS for more discussion of these 'ghost' fault events.

We partition the T set into two subsets: Stage1 and Stage2 for translation read events from a stage 1 or stage 2 walk respectively (stage 2 reads during a stage 1 walk are marked as stage 2, not stage 1).

Finally, we leave the M set unchanged, which contains only the explicit reads and writes performed by instructions.

TLBIs As described in §7.7 Arm have a variety of TLBI instructions, with varying arguments. All of these TLBIs generate a single TLBI event.

To aide in modelling, there are a set of subsets of TLBI for various kinds of TLBI:

- ▷ TLBI-S1 for invalidations of Stage 1 entries.
- ▷ TLBI-S2 for invalidations of Stage 2 entries.
- ▷ TLBI-IPA for invalidations by intermediate physical address.
- ▷ TLBI-VA for invalidations by virtual address.
- ▷ TLBI-ASID for invalidations by ASID.
- ▷ TLBI-VMID for invalidations by VMID.
- ▷ TLBI-ALL for the TLBI ALL instructions.
- ▷ TLBI-IS for broadcast TLBIs.
- ▷ TLBI-EL1 for invalidations of the EL1&0 regime.
- ▷ TLBI-EL2 for invalidations of the EL2 regime.

These events do not *cut* the TLBI set into partitions, but rather any TLBI event may belong to multiple. For example, a TLBI VAE1IS event would belong to TLBI-VA, TLBI-VMID, TLBI-EL1, and TLBI-IS.

We also include all TLBIs in a general C ("Cache maintenance") set.

Exceptions Despite not modelling exceptions in general in this work, we do need to include some exception machinery in the model to capture the minimal ordering requirements arising from both their context synchronisation effects and also behaviours from crossing exception level boundaries.

To support this we add two new events to capture taking and returning from exceptions: TE ("Take-exception") and ERET.

Barriers The Arm DSB ("Data synchronization barrier") instruction is required for TLB maintenace, as was seen in the previous chapter. We include DSB events, one for each kind of DSB instruction:

- ▷ DSBSY and DSBISH (here, equivalent as we do not model shareability domains)
- ▷ DSBNSH, for thread-local effects.
- ▷ DSBST, DSBLD, for DSBs affecting only loads or stores.
- ▷ DSBISHST, DSBISHLD, and so on, for all combinations of DSB instruction domain and access types.

Arm define a hierarchy of barriers where, for example: DMB.LD < DMB.SY < DSB.SY That is, any ordering imposed by a DMB.LD is also imposed by a DMB.SY, and therefore also a DSB.SY.

To help capture this, and reduce the explosion in the number of relations in the model later on, we simplify and update the barrier story in the Arm model and include the helper sets given in Figure 9.1.

```
let dsbsy = DSBISH | DSBSY | DSBNSH
let dsbst = dsbsy | DSBST | DSBISHST | DSBNSHST
let dsbld = dsbsy | DSBLD | DSBISHLD | DSBNSHLD
let dsbnsh = DSBNSH
let dmbsy = dsbsy | DMBSY
let dmbst = dmbsy | dsbst | DMBST | DSBST | DSBISHST | DSBNSHST
let dmbld = dmbsy | dsbld | DMBLD | DSBISHLD | DSBNSHLD
let dmb = dmbsy | dmbst | dmbld
let dsb = dsbsy | dsbst | dsbld
```

Figure 9.1: Barrier helper sets.

Context changing and synchronisation Finally, we add events for context-changing and context-synchronising operations. Context changes involve updates to system registers which change the current translation regime, which generate MSR events. We add a general context-synchronisation event set CSE which includes ISB, TE and ERET.

Changes to system registers may have relaxed behaviours, as described in §8.7.1, but full relaxation of the system register reads done by the Arm psueocode is unlikely to be valid, consistent or meaningful. Instead, we introduce a *pointed-set semantics*: when generating a candidate, we keep a per-system-register set of writes to that register, remembering which one is the most recent. On a write to that system register, we add it to the set. On a read of that system register, we generate one candidate for each value in the set, and then 'lock' the remainder of the execution of that instruction to that value so repeated reads will see the new value. When a context-synchronization event is generated (that is, an event that will be in the CSE set) all the sets are reduced to singleton sets containing only the most recent write.

This gives us some relaxed behaviours, enough to see relaxed behaviours around changes to the TTBR, but we note that this is unlikely to be the full story for relaxed system registers.

9.1.2 Candidate relations

In addition to those new events, we introduce new relations over those (and other) events:

- trf and tfr as analogues to rf and fr but for translation-read (T) events.
- iio ("intra-instruction order") which relates events of the same instruction in the order they occur during execution of that instruction's intra-instruction semantics as defined by the Arm ASL.
- same-va, same-ipa, same-pa relations which relate events whose virtual, intermediate physical or physical address of the associated explicit memory access are the same.
- same-va-page, same-ipa-page, same-pa-page which relate events whose associated explicit memory events are in the same page (e.g. 4KiB chunk) of the virtual, intermediate physical or physical address space.
- same-asid, same-vmid relates events for which translations for the associated memory event are using the same ASID or VMID.
- wco, a generalised coherence order which includes TLBIs.

Addresses, ASIDs and VMIDs Each translation table walk will read from registers and system registers and get a value for the (input) address, the current ASID and current VMID. We then relate each T with any other T where the translation associated with it is for the same virutal address (with same-va), the same intermediate-physical address (with same-ipa), or the same resulting physical address (same-pa). This means that all T events within a translation have the same same-* relations. We also include relations which match translation's virtual, intermediate physical and physical addresses if they are in the same page rather than exactly, with the same rules, but as a same-*-page relation.

If two translations are for the same ASID, their translation reads are related by same-asid. If two translations are for the same VMID, their translation reads are related by same-vmid.

To use these relations we also include TLBI events. A TLBI-X is related to T by same-X if the parameter to the TLBI instruction (the page, vmid, or asid) either passed by register, immediate or through the current context, if the T event's associated translation matches X. For example, a TLBI-IPA event would be same-ipa-page related to a T whose translation was for an intermediate physical address in the page provided as the parameter to the TLBI IPA instruction.

Generalised coherence order We create an extended coherence order wco, which is the usual co (a per-location total order of writes to that location) as well as their relative ordering to all TLBI events.

One might be concerned at the validity of doing this, on two fronts. Generally, extending coherence to a total order over all locations is sound [9, §10.5 p174], and so there is no issue in doing this. Secondly, for broadcast TLBIs, microarchitecture will implement these with message passing to and from each core separately, and so there is no single moment the TLBI 'happens'. However, as described in §8.6.7 we seem to be able to consider TLBI instructions as executing 'atomically' so long as there are no break-before-make violations. This is a similar justification as to including DC and IC events in a similar generalised coherence order for instruction fetching [58, §5 p29].

Dependencies A candidate execution consists not only of events, and reads-from relations but also a set of dependencies: addr, data, ctrl, po and loc. We add iio and tdata to these.

The intra-instruction ordering iio relation relates two events in the same instruction in the order the Arm pseudocode generated the events. This relation therefore captures a total order over all events within an instruction, regardless of the intra-instruction dependencies (control, data) or unordered accesses (for example, for misaligned accesses). We are currently invesitagting a relaxation of this ordering, and associated changes in the underlying Arm pseudocode definitions, to enable a more relaxed definition of the ordering within an instruction to handle these cases.

We make loc relate events with the same physical address (for T events, this is the physical location of the translation table entry).

Program order (po) is restricted to explicit events: R, W, F, C, CSE and MSR. Implicit translation reads (T) and any indirect reads or writes of registers are not included in po.

Address dependencies were once fundamental, but now we can define address dependencies in the presence of address translation as dependencies into the translation table walk. To do this, we include a new relation tdata that relates reads with the translation read events of a translation which reads from the register written by that read to compute the address. The traditional addr can be recovered as `tdata ; iio* ; [M]`.

9.2 Cat model

The base Arm axiomatic model had three axioms: internal, external, and atomic. These were acyclicity and emptyness checks of unions of set of relations: obs, dob, aob and bob. We will slightly modify three of these relations obs, bob and dob, and add 5 new ones (tob, obtlbi, ctxob, obfault, obETS) to handle the ordering between translations and TLBIs, and include them in the external acyclicity check. Then we will introduce one final new axiom translation-internal.

Figure 9.2 contains the axioms and relations for the updated Armv8-A relaxed virtual memory axiomatic model (RVM). Unchanged parts from the original are greyed out. Note that some helper relations are elided here, and will be described in more detail later.

9.3 Axioms

The RVM model axioms are, mostly, a syntactic extension to the original Armv8-A axiomatic model presented in §**TODO: ref intro**. This is deliberate. Although there may be other, perhaps even nicer or more succinct, ways of phrasing the given model, the variation presented here is designed to be syntactically as close as possible to the original. This helps with readability for those familiar with the original; it allows us to present the differences to the original in an easier form; it makes recovery of the original model easier; and, it makes it easier to prove equivalance of the axiomatic models in the presence of constant address translation, increasing the confidence we have in the model.

The model has 3 kinds of axioms: internal ones for per-location guarantees, an external axiom for the global happens-before ordering, and the atomic axiom for RMWs (untouched in this work).

Internal axioms The new model has two per-location axioms: internal and translation-internal.

```
let speculative =
    ctrl
  | addr; po
  | [T]; instruction-order

(* translation-ordered-before *)
let tob =
    [T_f]; tfre
  | [T]; iio; [R|W]; po; [W]
  | speculative; trfi

(* observed by *)
let obs = rfe | fr | wco
  | trfe

(* ordered-before TLBI and translate *)
let obtlbi_translate =
    [T & Stage1] ; tlb_barriered ; [TLBI-
    S1]
  | ([T & Stage2] ; tlb_barriered ; [TLBI
    -S2])
     & (same-translation ; [T & Stage1] ;
     trf^-1 ; wco^-1)
  | (([T & Stage2] ; tlb_barriered ; [
    TLBI-S2]) ; wco? ; [TLBI-S1])
    & (same-translation ; [T & Stage1] ;
    maybe_TLB_cached)

(* ordered-before TLBI *)
let obtlbi =
    obtlbi_translate
  | [R|W|Fault_T]; iio^-1; (
    obtlbi_translate & ext); [TLBI]

(* context-change ordered-before *)
let ctxob =
    speculative; [MSR]
  | [CSE]; instruction-order
  | [ContextChange]; po; [CSE]
  | speculative; [CSE]
  | po; [ERET]; instruction-order; [T]

(* ordered-before a translation fault *)
let obfault =
    data; [Fault_T & FaultFromW]
  | speculative; [Fault_T & FaultFromW]
  | [dmbst]; po; [Fault_T & FaultFromW]
  | [dmbld]; po; [Fault_T & (FaultFromW|
    FaultFromR)]
  | [A|Q]; po; [Fault_T & (FaultFromW |
    FaultFromR)]
  | [R|W]; po; [Fault_T &
    FaultFromReleaseW]

(* ETS-ordered-before *)
let obETS =
    (obfault; [Fault_T]); iio^-1; [T_f]
  | ([TLBI]; po; [dsb]; instruction-order
    ; [T]) & tlb-affects

(* dependency-ordered-before *)
let dob =
    addr | data
  | speculative; [W]
  | addr; po; [W]
  | (addr | data); rfi
  | (addr | data); trfi

(* atomic-ordered-before *)
let aob = rmw
  | [range(rmw)]; rfi; [A | Q]

(* barrier-ordered-before *)
let bob = [R]; po; [dmbld]
  | [W]; po; [dmbst]
  | [dmbst]; po; [W]
  | [dmbld]; po; [R|W]
  | [L]; po; [A]
  | [A | Q]; po; [R | W]
  | [R | W]; po; [L]
  | [F | C]; po; [dsbsy]
  | [dsb]; po

(* Ordered-before *)
let ob = (obs | dob | aob | bob
  | iio | tob | obtlbi | ctxob | obfault
    | obETS)^+

(* Internal visibility requirement *)
acyclic po-loc | fr | co | rf as internal
(* External visibility requirement *)
irreflexive ob as external
(* Atomic requirement *)
empty rmw & (fre; coe) as atomic
(* Writes cannot forward to po-future
    translates *)
acyclic (po-pa | trfi) as translation-
    internal
```

Figure 9.2: RVM axioms and relations

```
(* Internal visibility requirement *)
acyclic po-loc | fr | co | rf as internal
(* Writes cannot forward to po-future translates *)
acyclic (po-pa | trfi) as translation-internal
```

Unchanged from the original, the `internal` axiom captures the SC-per-location guarantee briefly discussed in §**TODO: ?REF?**. Translations, however, do not have the same per-location guarantees. To account for this, we introduce a second axiom, `translation-internal`, which captures the weaker per-location guarantee for translation table walks. Since translation reads, in the presence of TLB caching and out-of-order pipelines, do not guarantee even coherence, the only behaviour this axiom ends up preventing is translation reads reading from program-order later stores.

External axiom The `external` axiom asserts acyclicity of the global happens-before ordering for Arm. The happens-before (called ob, 'ordered-before', in Arm) relation is the union of all the ordering relations, given in §9.4.

```
(* Ordered-before *)
let ob = (obs | dob | aob | bob | iio | tob | obtlbi | ctxob | obfault |
    obETS)^+
(* External visibility requirement *)
irreflexive ob as external
```

We choose to include all the pipeline and TLB effects as ordering requirements, rather than introducing new ordering axioms just for translation and TLB invalidation. This produces a model that is more consistent with the previous Arm memory models, and ensures ordering information gained through observing translation table walks are respected by non-translation-table accesses.

Atomic axiom The `atomic` axiom remains unchanged. In this work we do not consider the interaction of translation with atomic accesses.

```
(* Atomic requirement *)
empty rmw & (fre; coe) as atomic
```

9.4 Relations

The RVM model modifies some of the original, and introduces some new, ordering relations. This section goes through each in detail, describing the mechanism and justifying the existence or non-existence of particular clauses.

9.4.1 obs

```
(* observed by *)
let obs = rfe | fr | wco | trfe
```

The 'observed-by' relation. It includes the original `rf` and `fr` (over physical locations), the *generalised coherence order* `wco` (§9.1.2), and the `trfe` (translation-reads-from-external) relation.

Generalised coherence Including `wco`, which is existentially quantified over the candidates, fixes some global order the writes and `TLBI`s happen in. Consider, informally, some microarchitectural execution. It would propagate writes to the coherent storage subsystem, and would complete `TLBI` instructions, and these events would be interleaved in some whole-machine trace. The generalised `wco` relation captures the relative ordering of these events in the axiomatic model, as they would have happened in the traces of machine executions. The model is then quantified over all such orderings, accounting for any interleaving of these events.

External translation reads Inclusion of `trfe` enforces that translation-table-walk translation reads, which could not come from forwarding, must have originally come from the coherent storage subsystem and so the write must have been globally propagated before the translation read happened (§8.4.2, §8.4.7).

However, the translation read might have happened much later, either due to extreme out-of-order (§8.4.1) or TLB caching (§8.5.1), and so we do not include `tfre` (translation-from-reads-external) in ob.

Additionally, writes may be propagated to that thread's translation table walker before they are propagated to the coherent storage subsystem (§8.4.4), in other words, they can be forwarded. Therefore we do not include `trfi` (translation-reads-from-internal) in obs.

9.4.2 dob

```
let dob =
    addr | data
  | speculative; [W]
  | addr; po; [W]
  | (addr | data); rfi
  | (addr | data); trfi
```

The dependency-ordered-before relation is mostly unchanged, we add a single `(addr | data); trfi` clause to the end to forbid thin-air creation of values (§8.4.1, §8.4.2, **TODO: need dedicated thin air paragraph/test in prev chapter**) similarly to the original model for data memory reads.

9.4.3 bob

```
let bob =
    [R]; po; [dmbld]
  | [W]; po; [dmbst]
  | [dmbst]; po; [W]
  | [dmbld]; po; [R|W]
  | [L]; po; [A]
  | [A | Q]; po; [R | W]
  | [R | W]; po; [L]
  | [F | C]; po; [dsbsy]
  | [dsb]; po
```

We rewrite the original barrier-ordered-before relation to use the barrier helpers defined in Figure 9.1. This does not change the underlying model for DMB instructions, but allows those same clauses to capture the barrier hierarchy imposing the same ordering when using stronger barriers (namely, DSBs).

The Arm DSB instruction has some extra ordering however. Firstly that a `DSB SY` orders `TLBI` instructions (§8.6.2) and so we include `[F|C];po;[dsbsy]`. Secondly, all program-order later events must wait for an earlier DSB to finish before performing its explicit memory events, so we also include `[dsb];po` in ob.

9.4.4 tob

```
let tob =
    [T_f]; tfre
  | [T]; iio; [R|W]; po; [W]
  | speculative; trfi
```

Translation table walks themselves impose ordering on the surrounding events.

Invalid writes The first of these is one of the key behaviours described in §8.3.3, that reads of invalid entries must not have come from the TLB. So we add the `[T_f];tfre` edge to capture this, that any translation-reads which read an invalid entry must happen before any writes coherence after the one it read from.

There is a major caveat here: write forwarding to the translation table walker. We cannot simply have `[T_f];tfr` as a thread-local write may be forwarded to the translation table walker before it's propagated to memory (§8.4.4).

However, it should not be the case that the write is forwarded from a write that is too old or behind a DSB if FEAT_ETS, except it may be the case that there might be other intervening writes in between. For now, we are unable to give a precise bound on the ordering for thread-local [T_f];tfr, and this area is still currently under investigation with Arm.

Speculation As we saw earlier, speculation interacts with translation in two ways: first, it is forbidden to read-from a still speculative write (§8.4.5), and, secondly, events program-order after an instruction which does a translation table walk are speculative until the translation table walk completes (§8.4.1).

To capture these we first define when one event is considered speculative until another event happens, with a new speculative relation, defined as following:

```
let speculative = ctrl | addr; po | [T]; instruction-order
```

This captures all the control-flow dependencies that we model here, the classic ctrl and addr; po, as well as a new general [T]; instruction-order which says that all events ordered (iio|po)+ after a translation read are speculative until the translation read satisfies. We can then include speculative ; trfi to succinctly forbid any forwarding of still-speculative writes to translation table walks.

Finally, we include [T]; iio; [M]; po; [W] which captures that writes cannot propagate until program-order earlier instructions have their physical address (so, do not fault). Although, this edge is subsumed by the speculative; [W] edge in dob, it is kept here for clarity.

9.4.5 ctxob

NOTE: The model for exceptions and context-synchronising events is currently under revision, and what is presented here is likely to change.

```
let ctxob =
    speculative; [MSR]
  | [CSE]; instruction-order
  | [ContextChange]; po; [CSE]
  | speculative; [CSE]
  | po; [ERET]; instruction-order; [T]
```

The ctxob relation captures the orderings required from context changing and synchronising operations, without trying to capture the full extent of the relaxed behaviours. As such, these orderings are likely to be incomparable to the real semantics.

Speculation The first guarantee we see is that context changes and synchronisation should not happen speculatively. Speculative context changes may end up creating translation table roots and therefore translation table walks using unreachable writes (§8.5.5). To prevent this we ensure that context changing operations only happen once they are non-speculative, by enforcing speculative; [MSR] in ob. Forbidding speculative execution of context synchronisation is done through the inclusion of speculative; [CSE] in ob.

Context synchronising A context synchronisation event (such as an ISB or ERET instruction) should ensure that program-order earlier context-changing events are seen by program-order later instructions. Microarchitecturally this is achieved by having context-synchronisation events flushing the pipeline, restarting all program-order later instructions. For now this effect seems fixed in the architecture (§8.7), and so we get [CSE]; instruction-order in ob subsuming the earlier ISB orderings.

To ensure that context changes are seen after the synchronisation we include [ContextChange]; po; [CSE], and the union of these two relations ensures the context change is ordered before any program-order later events.

Exceptions Taking and returning from exceptions are context synchronising (§8.7), and so those are captured by the previous clauses. However, translation reads of a lower exception level should not satisfy during execution at a higher exception level. We over approximate this with po; [ERET]; instruction-order; [T] ensuring all translation reads after an ERET wait.

9.4.6 obfault and obETS

```
(* ordered-before a translation fault *)
let obfault =
    data ; [Fault_T & FaultFromW]
  | speculative ; [Fault_T & FaultFromW]
  | [dmbst] ; po ; [Fault_T & FaultFromW]
  | [dmbld] ; po ; [Fault_T & (FaultFromW | FaultFromR)]
  | [A|Q] ; po ; [Fault_T & (FaultFromW | FaultFromR)]
  | [R|W] ; po ; [Fault_T & FaultFromW & FaultFromReleaseW]

(* ETS-ordered-before *)
let obETS =
    (obfault; [Fault_T]); iio^-1; [T_f]
  | ([TLBI]; po; [dsb]; instruction-order; [T]) & tlb-affects
```

To capture the specific guarantees described by FEAT_ETS (§8.4.3, §8.6.2), we include 'ghost' Fault events in the candidate executions. These events sit in the execution (in po order) where the explicit memory event would have been if there was no fault, and tags the fault with the kind of fault it was (translation or permission).

Ordering to a fault To fully capture the strength of FEAT_ETS we keep track of syntactic dependencies *into* the instruction which faulted, and apply those dependencies to the Fault event itself. obfault then the syntactic subset of bob and dob where the right-hand side of each clause is substituted with a Fault_T (a translation fault).

Using obfault we can then keep track of the (syntactic) subset of ob that would have ordered the explicit event after, and associate those relations with the Fault_T event instead. obETS's first clause then adds to ob this ordering, but attached to the translation read of the invalid entry itself, as architected by FEAT_ETS.

Note that dependencies and orderings *from* a faulting instruction seem not respected, and so we do not induce orderings out of a Fault_T.

FEAT_ETS and TLBI The second clause of obETS captures the second architected behaviour of FEAT_ETS (§**TODO: TLBI ordering needs ETS explained**), that faults after a thread-local TLBIs do not need context synchronisation to be ordered after the TLBI. Note that one still needs a DSB to complete the TLBI in that case.

9.4.7 obtlbi

```
(* ordered-before TLBI *)
let obtlbi =
    obtlbi_translate
  | [R|W|Fault_T]; iio^-1; (obtlbi_translate & ext); [TLBI]

(* translate ordered-before TLBI *)
let obtlbi_translate =
  [T & Stage1] ; tlb_barriered ; [TLBI-S1]
| (([T & Stage2] ; tlb_barriered ; [TLBI-S2]) ; wco? ; [TLBI-S1])
  & (same-translation ; [T & Stage1] ; maybe_TLB_cached)
| ([T & Stage2] ; tlb_barriered ; [TLBI-S2])
    & (same-translation ; [T & Stage1] ; trf^-1 ; wco^-1)
```

Finally, there is the obtlbi relation which captures the ordering from translations (and their explicit memory events) and the TLB invalidations which affect them. The relation is split in two: the obtlbi_translate clause enforces order between stale translations and the TLBIs they are invalidated by, the second clause covers broadcast TLBIs.

Capturing stale TLB entries When a translation read happens, it is allowed for it to read from a stale write (§8.5.1). That is, the translation need not be ordered before writes which come after the write it actually reads from. Consequently the tfre relation is not included in ob.

We strengthen this, by including some edges from translations to TLBIs, when there is an interposing newer write. The general shape of this ordering is illustrated in Figure 9.3.

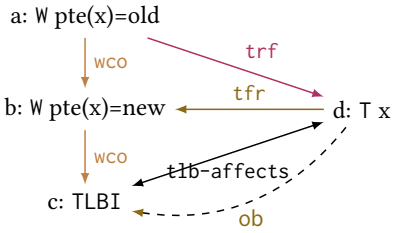


Figure 9.3: General `obtlbi_translate` shape.

This shape is succinctly captured by the `tlb_barriered` auxiliary relation, which relates any translate-read that reads from a write which is `wco` before another write which is `wco` before a `TLBI` which targets the address, ASID or VMID of the translation:

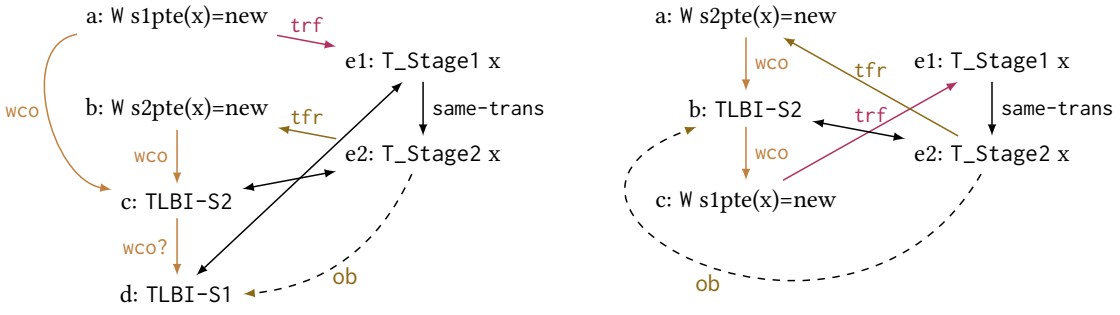
```
let tlb_barriered =
  ([T] ; tfr ; wco ; [TLBI]) & tlb-affects^-1
```

We cannot simply include `tlb_barriered` in ob, however. Instead, we must consider the orderings for stage 1 and stage 2 translation reads separately.

Stale stage 1 reads For stage 1 translation reads, either in single-stage regimes or as part of a two-stage regime, we can include a variant of `tlb_barried` specialised to stage 1 translation-reads and `TLBI`s which affect stage 1 entries.

Stale stage 2 reads Stage 2 walks are more subtle. The requirement to perform stage 1 invalidation (§8.6.4) means that, in those instances, we do not get `tlb_barriered` directly.

Instead, we have to case split on the execution: either, (1) the translation table walk does a stage 1 translation read which reads-from an older write, in which case there may have been a whole cached translation that must be invalidated; or, (2) one of the stage 1 translation reads of the translation table walk reads from a write that is newer than the stage 2 `TLBI` and so there cannot have been any cached whole translation entries in the TLB and so, logically, we only need the stage 2 invalidation. These cases are illustrated in Figure 9.4, and correspond to the two clauses of `obtlbi_translate` which match on stage 2 translation reads.



Case (1) Case (2)

Figure 9.4: `obtlbi` stage 2 scenarios.

We capture the general shape of (1), where a translation-read may have been cached in the TLB, with the following `maybe_TLB_cached` relation:

```
let maybe_TLB_cached = ([T]; trf^-1; wco; [TLBI]) & tlb-affects^-1
```

We then use this relation to add ordering from a stage 2 translation-read to the stage 1 TLBI, wco-after a stage 2 TLBI that removed any stale IPA mappings, which would remove any cached whole-translation any stage 1 translation-read might have read from, and after which any fresh translation table walk would be required to not see the stale stage 2 entry the translation-read read from.

Broadcast TLBIs Recall that broadcast TLBIs impose restrictions on other threads (§8.6.3). When a broadcast TLBI's invalidation affects a translation on another core, then it must also affect the explicit memory effect associated with it. This shape is illustrated in Figure 9.5, and corresponds to the final clause of obtlbi.

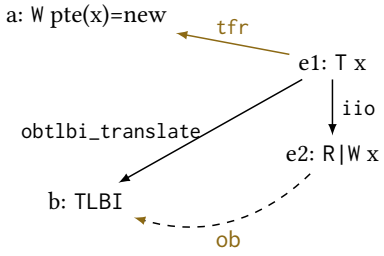


Figure 9.5: obtlbi broadcast TLBI shape.

Connecting TLB invalidations to translation reads The final part of the puzzle is how to relate TLBI events with translations which may be affected by the invalidation. Recall that the TLBIs are grouped into subsets of TLBI-S1, TLBI-VA, and so on. We define a tlb_might_affect that is the cross-product of these with the same-* relations:

```
let tlb_might_affect =
  [ TLBI-S1 & ~TLBI-S2 &  TLBI-VA  &  TLBI-ASID & TLBI-VMID] ; (same-va-
    page & same-asid & same-vmid) ; [T & Stage1]
| [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  &  TLBI-ASID & TLBI-VMID] ; (same-
    asid & same-vmid) ; [T & Stage1]
| [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  & ~TLBI-ASID & TLBI-VMID] ; same-vmid
     ; [T & Stage1]
| [~TLBI-S1 &  TLBI-S2 &  TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; (same-ipa
    -page & same-vmid) ; [T & Stage2]
| [~TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; same-vmid
     ; [T & Stage2]
| [ TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; same-vmid
     ; [T]
| ( TLBI-S1 &            ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID) * (T &
    Stage1)
| (            TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID) * (T &
    Stage2)
```

Finally, we get tlb-affects by attaching tlb_might_affect to events in the same thread, and if a TLBI-IS, to ones in other threads too:

```
let tlb-affects =
  [TLBI-IS]; tlb_might_affect
| ([~TLBI-IS]; tlb_might_affect) & int
```

9.5 Interface

To support the new Armv9 ISA and the new concurrency interface, we produce architecture-specific definitions using the new isla-cat language features in isla-axiomatic.

Barriers are instances of the sail_barrier outcome. For Arm we instantiate these with the Arm Barrier union.

Figure 9.6 contains the isla-cat definitions for the Arm barriers. The sail Arm Barrier union is reproduced here for the reader's benefit but is not required (nor present) in the source cat file. Similar unions, structs, enums and

```
(* F for all fences *)
accessor F: bool = is sail_barrier

(* from the Arm ASLl, compiled to sail *)
union Barrier = {
  Barrier_DSB : DxB,
  Barrier_DMB : DxB, // The nXS field is ignored from DMBs
  Barrier_ISB : unit,
  Barrier_SSBB : unit,
  Barrier_PSSBB : unit,
  Barrier_SB : unit,
}

(* accessors for each relevant constructor *)
accessor z_dmb: bool =
    .match {
        Barrier_DMB => true,
        _ => false
    }

accessor z_dsb: bool =
    .match {
        Barrier_DSB => true,
        _ => false
    }

accessor z_isb: bool =
    .match {
        Barrier_ISB => true,
        _ => false
    }

(* cat event sets for the different barriers *)
define DMB(ev: Event): bool =
    F(ev) & z_dmb(ev)

define DSB(ev: Event): bool =
    F(ev) & z_dsb(ev)

define ISB(ev: Event): bool =
    F(ev) & z_isb(ev)
```

Figure 9.6: `isla-cat` accessors for Arm barriers.

corresponding accessors and definitions exist for the Arm barrier domains (`NSH`, `ISH`, `OSH`) and access types (`ST`, `LD`, `SY`), elided here for brevity.

We make use of **`accessor`**s to access fields of the sail structs and unions, both here for barriers, and also for exceptions (faults), and `TLBI`s, as well as defining the `trf` and `wco` (found in Figure 9.7) relations. The full `isla-cat`-defined interface can be found in the Appendix**TODO: make appendix**.

```
declare wco(Event, Event): bool

(* wco has domain and range of W,CacheOp *)
assert forall ev1: Event, ev2: Event =>
    wco(ev1, ev2) -->
    (W(ev1) | C(ev1) | (ev1 == IW)) & (W(ev2) | C(ev2))

(* wco is transitive *)
assert forall ev1: Event, ev2: Event, ev3: Event =>
    wco(ev1, ev2) & wco(ev2, ev3) --> wco(ev1, ev3)

(* wco is total *)
assert forall ev1: Event, ev2: Event, ev3: Event =>
    wco(ev1, ev3) & wco(ev2, ev3) & ~(ev1 == ev2) -->
    wco(ev1, ev2) | wco(ev2, ev1)

(* wco is irreflexive *)
assert forall ev1: Event, ev2: Event, ev3: Event =>
    wco(ev1, ev2) --> ~(ev1 == ev2)

(* wco is antisymmetric *)
assert forall ev1: Event, ev2: Event =>
    wco(ev1, ev2) --> ~wco(ev2, ev1)

(* all write/cache-op pairs are wco related *)
assert forall ev1: Event, ev2: Event =>
    W(ev1) & C(ev2) -->
    wco(ev1, ev2) | wco(ev2, ev1)

(* wco is consistent with co *)
assert forall ev1: Event, ev2: Event =>
    co(ev1, ev2) --> wco(ev1, ev2)

(* all C are wco after IW
 * n.b. all W are wco after IW, because all W are co after IW and co =>
    wco
 *)
assert forall ev: Event =>
    C(ev) --> wco(IW, ev)
```

Figure 9.7: `wco.cat`: `isla-cat` definition of `wco`.