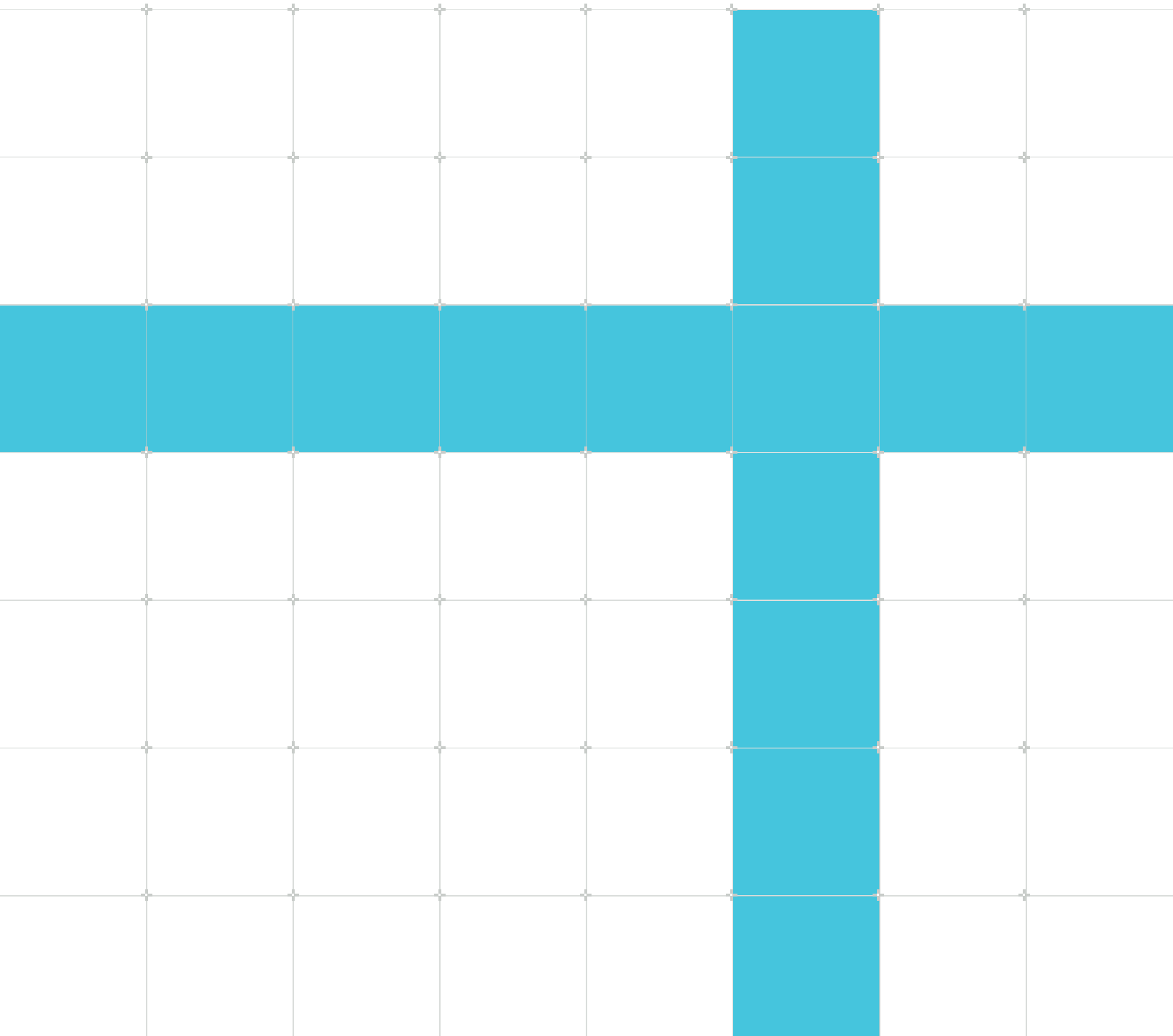
Learn the architecture - AArch64 Memory Management

Version 1.6

Guide

Non-Confidential
Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates).
All rights reserved.

Issue 00
101811_0106_00_en



Learn the architecture - AArch64 Memory Management Guide

This document is Non-Confidential.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.

This document is protected by copyright and other intellectual property rights.
Arm only permits use of this document if you have reviewed and accepted Arm's Proprietary Notice found at the end of this document.

This document (101811_0106_00_en) was issued on 2026-10-06. There might be a later issue at https://developer.arm.com/documentation/101811

The product version is 1.6.

See also: Proprietary notice | Product and document information | Useful resources

Start reading

If you prefer, you can skip to the start of the content.

Intended audience

Software Developers

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Contents

1. Overview **5**

2. What is memory management? **6**
2.1 Why is memory management needed? 6

3. Virtual and physical addresses **7**

4. The Memory Management Unit (MMU) **9**
4.1 Table entry 10
4.2 Table lookup 11
4.3 Multilevel translation 12

5. Address spaces **14**
5.1 Physical Addresses 16
5.2 Address sizes 18
5.3 Size of virtual addresses 18
5.4 Size of physical addresses 20
5.5 Size of intermediate physical addresses 20
5.6 Address Space Identifiers - Tagging translations with the owning process 20
5.7 Virtual Machine Identifiers - Tagging translations with the owning VM 22
5.8 Common not Private 22

6. Controlling address translation Translation table format **23**

7. Translation granule **24**
7.1 The starting level of address translation 24
7.2 Registers that control address translation 25
7.3 MMU disabled 26

8. TLBs and Changing translations **27**
8.1 Updating translations 27
8.2 Format of a TLB operation 29
8.3 TLBI Domains 30

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

9. Contiguous mappings..... 32
9.1 Contiguous bit..... 32
9.2 FEAT_CCONT..... 35

10. Address translation instructions..... 40

11. Check your knowledge..... 41

12. Related information..... 42

13. Next steps..... 43

Proprietary notice..... 44

Product and document information..... 46
Product status..... 46
Revision history..... 46
Conventions..... 47

Useful resources..... 49

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

1. Overview

This guide introduces memory translation in AArch64, which is key to memory management. It explains how virtual addresses are translated to physical addresses, the translation table format, and how software manages the Translation Lookaside Buffers (TLBs).

This information is useful for anyone who is developing low-level code, such as boot code or drivers. It is particularly relevant to anyone who is writing code to set up or manage the Memory Management Unit (MMU).

At the end of this guide, you can check your knowledge. You will have learned how a virtual address is translated to a physical address. You will be able to name the different address spaces, and describe how the address spaces map onto the stages of translation. You will also have learned when software must perform TLB maintenance, and the syntax of TLB maintenance commands.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2. What is memory management?

Memory management describes how access to memory in a system is controlled. The hardware performs memory management every time that memory is accessed by either the OS or applications. Memory management is a way of dynamically allocating regions of memory to applications.

2.1 Why is memory management needed?

Application processors are designed to run a rich OS, such as Linux, and to support virtual memory systems. Software that executes on the processor only sees virtual addresses, which the processor translates into physical addresses. These physical addresses are presented to the memory system and point to the actual physical locations in memory.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3. Virtual and physical addresses

The benefit of using virtual addresses is that it allows management software, such as an Operating System (OS), to control the view of memory that is presented to application software. The OS can control what memory is visible, the virtual address at which that memory is visible, and what accesses are permitted to that memory. This allows the OS to sandbox applications (hiding the resources of one application from another application) and to provide abstraction from the underlying hardware.

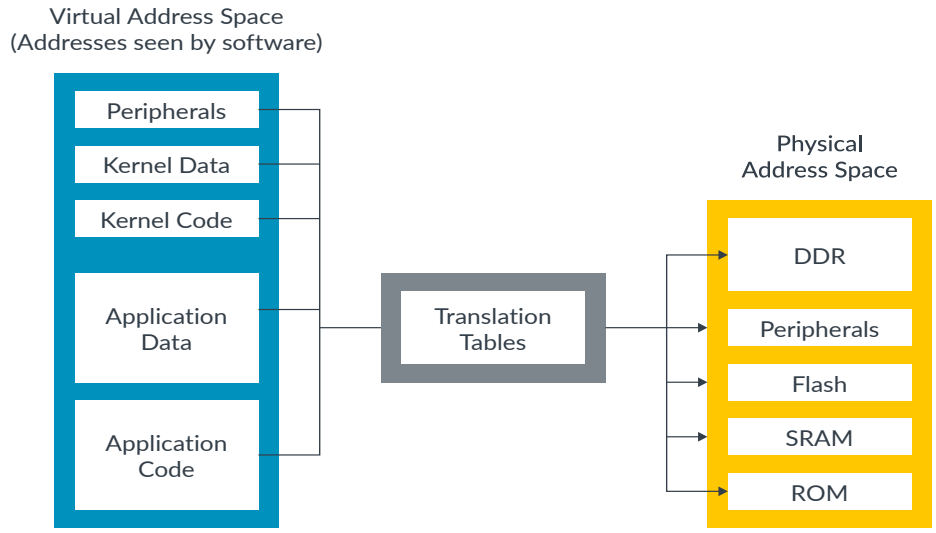
Another benefit of using virtual addresses is that an OS can present multiple fragmented physical regions of memory as a single, contiguous virtual address space to an application.

Virtual addresses also benefit software developers, who will not know a system's exact memory addresses when writing their application. With virtual addresses, software developers do not need to concern themselves with the physical memory. The application knows that it is up to the OS and the hardware to work together to perform the address translation.

In practice, each application can use its own set of virtual addresses that will be mapped to different locations in the physical system. As the operating system switches between different applications it re-programs the map. This means that the virtual addresses for the current application will map to the correct physical location in memory.

Virtual addresses are translated to physical addresses through mappings. The mappings between virtual addresses and physical addresses are stored in translation tables (sometimes referred to as page tables) as this diagram shows:

Figure 3-1: A diagram showing virtual and physical addresses



Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Translation tables are in memory and are managed by software, typically an OS or hypervisor. The translations tables are not static, and the tables can be updated as the needs of software change. This changes the mapping between virtual and physical addresses.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4. The Memory Management Unit (MMU)

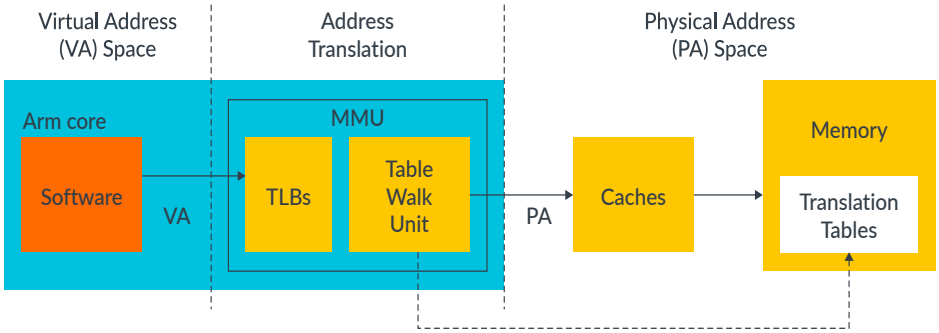
The Memory Management Unit (MMU) is responsible for the translation of virtual addresses used by software to physical addresses used in the memory system.

The MMU contains the following:

- The table walk unit, which contains logic that reads the translation tables from memory.
- Translation Lookaside Buffers (TLBs), which cache recently used translations.

All memory addresses that are issued by software are virtual. These memory addresses are passed to the MMU, which checks the TLBs for a recently used cached translation. If the MMU does not find a recently cached translation, the table walk unit reads the appropriate table entry, or entries, from memory, as shown here:

Figure 4-1: A diagram showing The MMU



A virtual address must be translated to a physical address before a memory access can take place (because we must know which physical memory location we are accessing). This need for translation also applies to cached data, because on Armv6 and later processors, the data caches store data using the physical address (addresses that are physically tagged). Therefore, the address must be translated before a cache lookup can complete.

Note

Architecture is a behavioural specification. The caches must behave as if they are physically tagged. An implementation might do something different, as long as this is not software-visible.

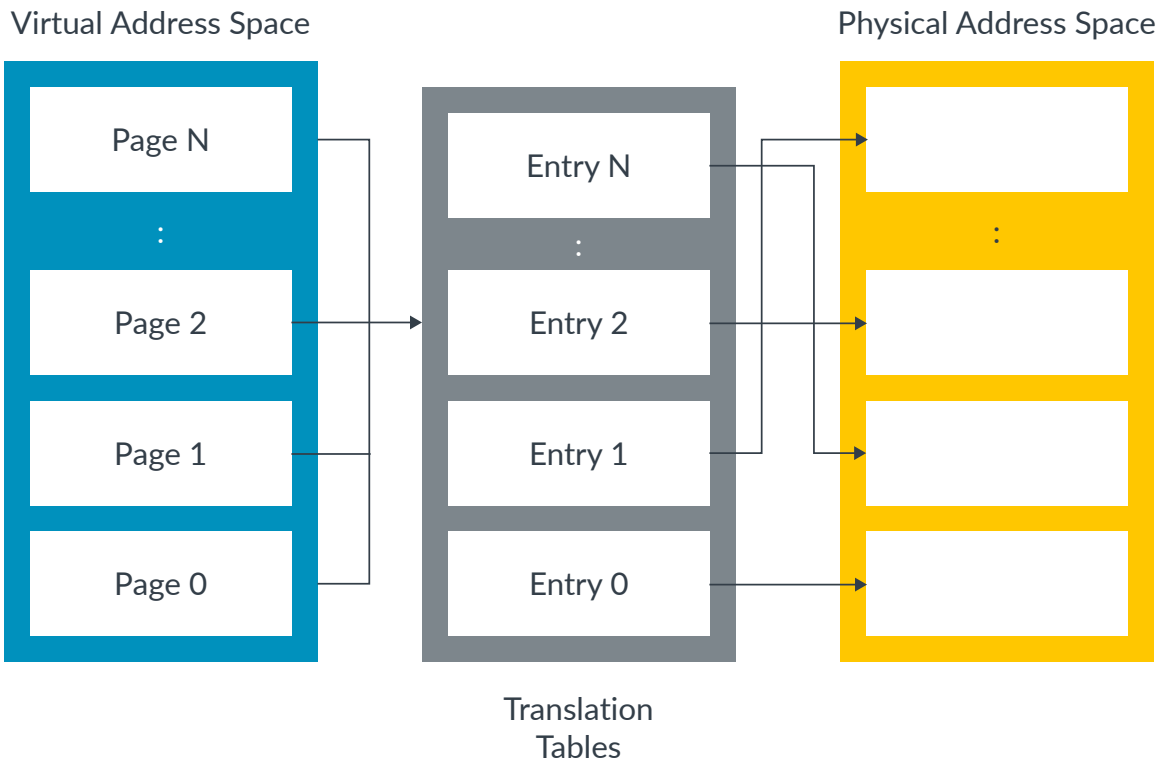
Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1 Table entry

The translation tables work by dividing the virtual address space into equal-sized blocks and by providing one entry in the table per block.

Entry 0 in the table provides the mapping for block 0, entry 1 provides the mapping for block 1, and so on. Each entry contains the address of a corresponding block of physical memory and the attributes to use when accessing the physical address.

Figure 4-2: A diagram showing table entry

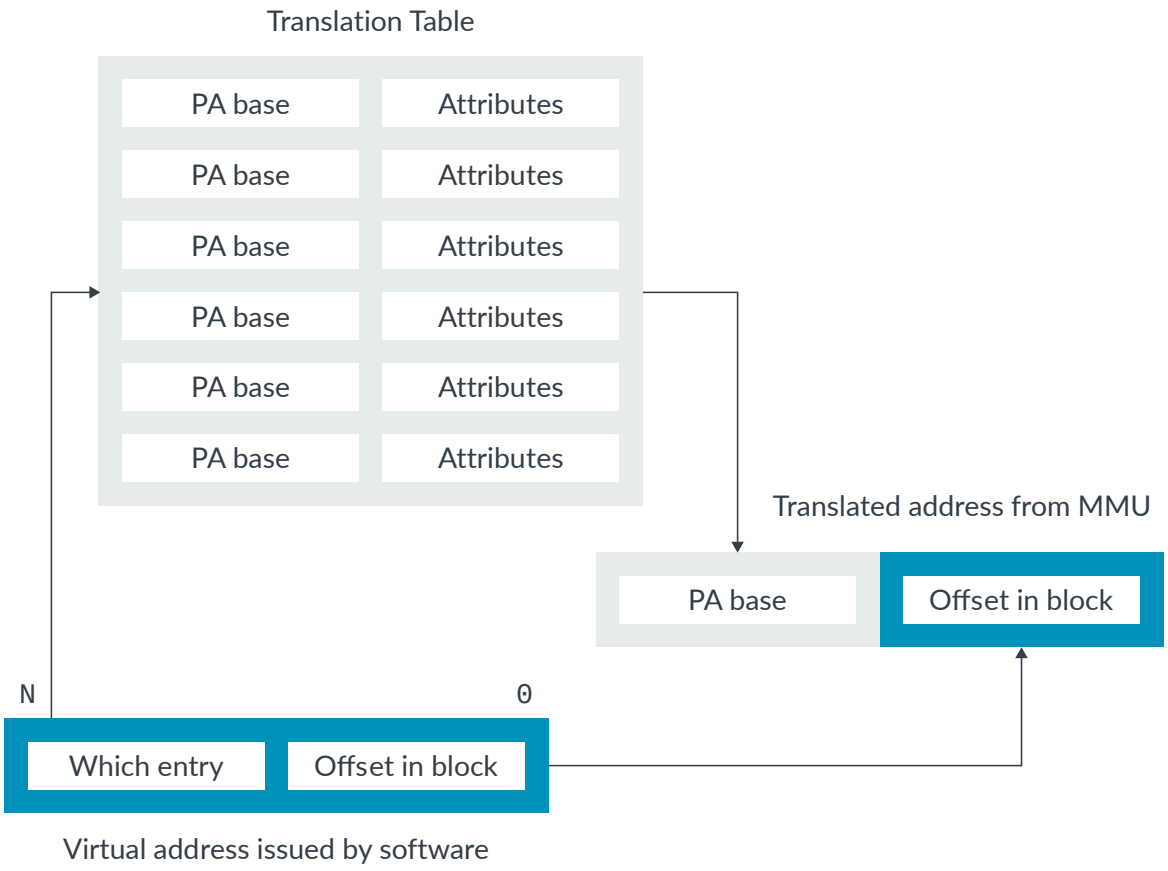


Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.2 Table lookup

A table lookup occurs when a translation takes place. When a translation happens, the virtual address that is issued by the software is split in two, as shown in this diagram:

Figure 4-3: A diagram showing table look-up



This diagram shows a single-level lookup.

The upper-order bits, which are labelled 'Which entry' in the diagram, tell you which block entry to look in and they are used as an index into the table. This entry block contains the physical address for the virtual address.

The lower-order bits, which are labelled 'Offset in block' in the diagram, are an offset within that block and are not changed by the translation.

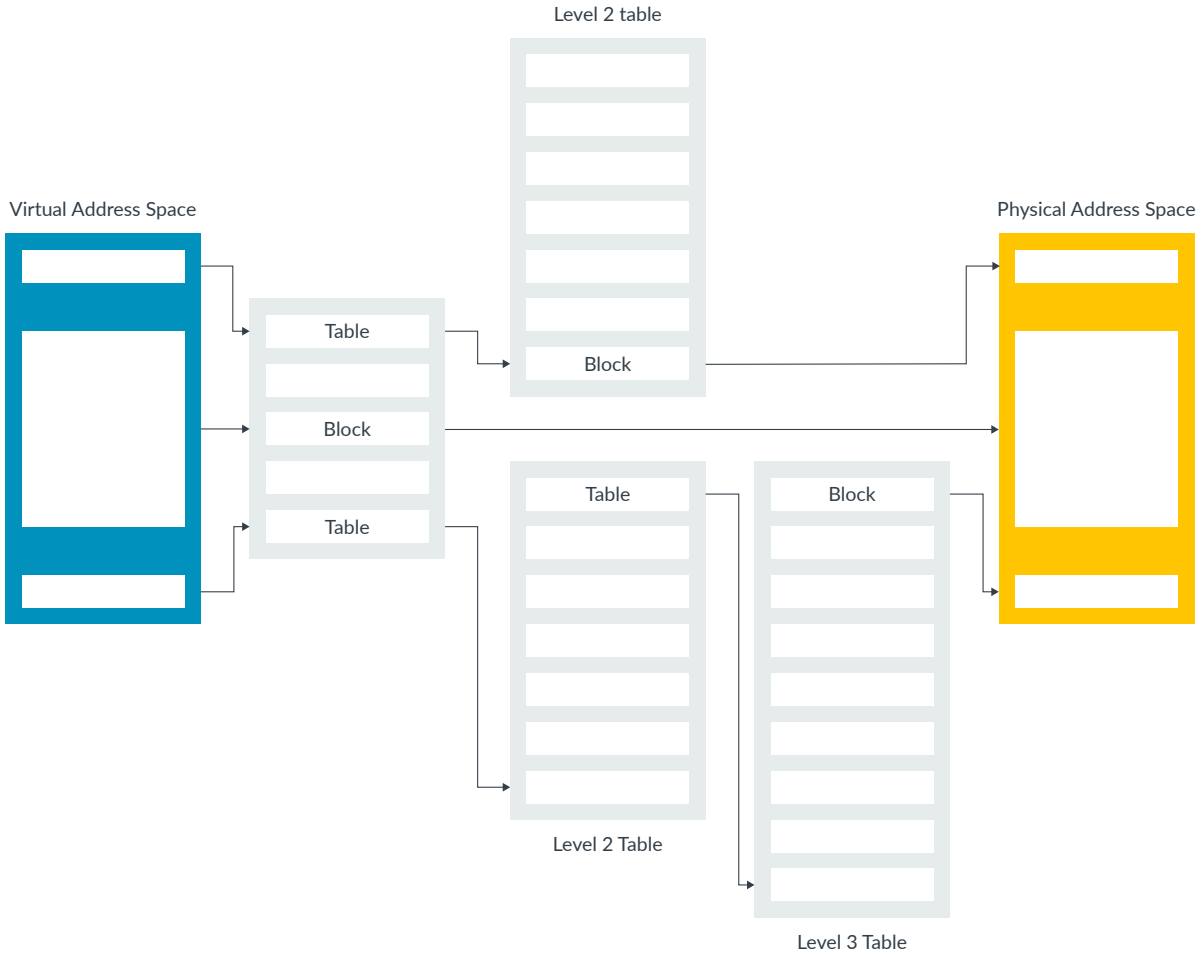
Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.3 Multilevel translation

In a single-level lookup, the virtual address space is split into equal-sized blocks. In practice, a hierarchy of tables is used.

The first table (Level 1 table) divides the virtual address space into large blocks. Each entry in this table can point to an equal-sized block of physical memory or it can point to another table which subdivides the block into smaller blocks. We call this type of table a 'multilevel table'. Here we can see an example of a multilevel table that has three levels:

Figure 4-4: A diagram showing multilevel translation



In Armv8-A, the maximum number of levels is six, and the levels are numbered -2 to 3.
This multilevel approach allows both larger blocks and smaller blocks to be described. The characteristics of large and small blocks are as follows:

- Large blocks require fewer levels of reads to translate than small blocks. Plus, large blocks are more efficient to cache in the TLBs.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Small blocks give software fine-grain control over memory allocation. However, small blocks are less efficient to cache in the TLBs. Caching is less efficient because small blocks require multiple reads through the levels to translate.

To manage this trade-off, an OS must balance the efficiency of using large mappings against the flexibility of using smaller mappings for optimum performance.


Note

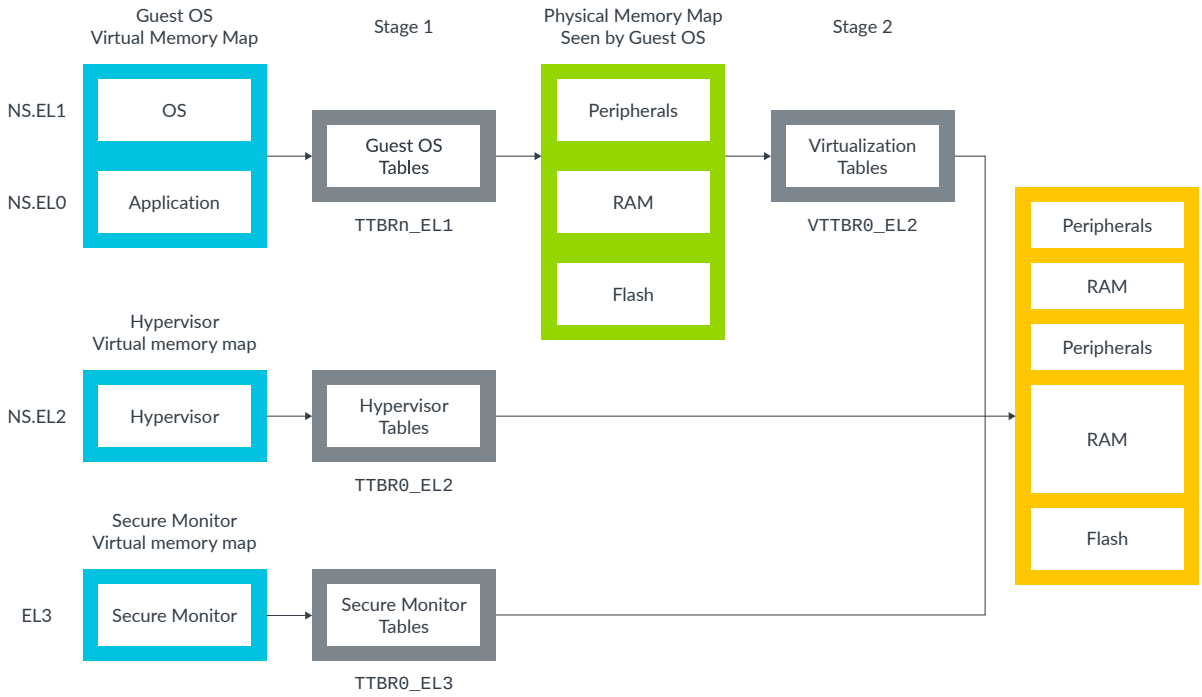
The processor does not know the size of the translation when it starts the table lookup. The processor works out the size of the block that is being translated by performing the table walk.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5. Address spaces

There are several independent virtual address spaces in AArch64. This diagram shows some of these virtual address spaces:

Figure 5-1: Address spaces in Armv8-A



The diagram shows three virtual address spaces:

- Non-secure EL0 and EL1.
- Non-secure EL2.
- EL3.

Each of these virtual address spaces is independent, and has its own settings and tables. We often call these settings and tables 'translation regimes'. There are also virtual address spaces for Secure EL0, Secure EL1 and Secure EL2, but they are not shown in the diagram.



Note

Support for Secure EL2 (FEAT_SEL2) was added in Armv8.4-A.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Because there are multiple virtual address spaces, it is important to specify which address space an address is in. For example, NS.EL2:0x8000 refers to the address 0x8000 in the Non-secure EL2 virtual address space.

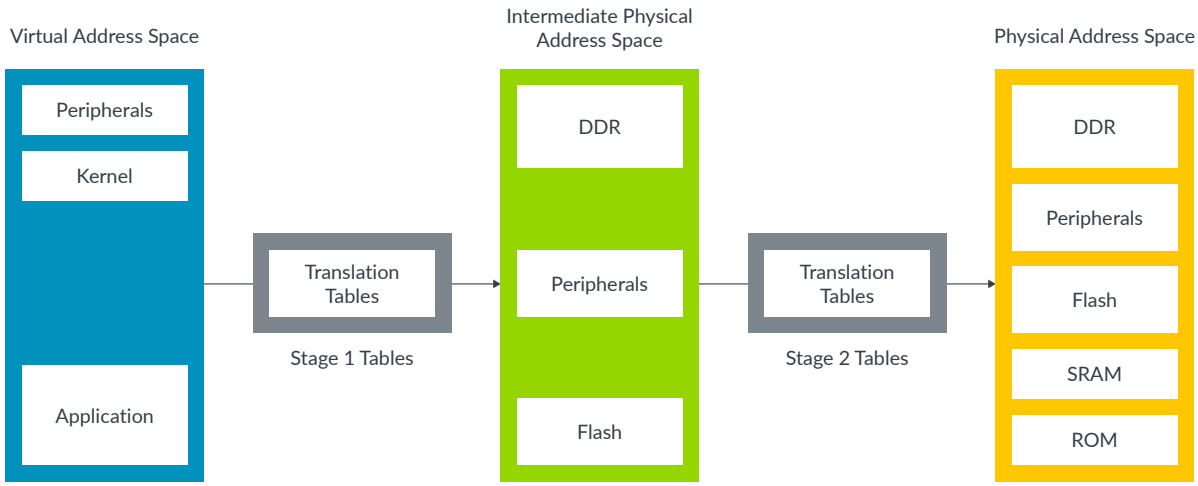
The diagram also shows that the virtual addresses from Non-secure EL0 and Non-secure EL1 go through two stages of translation. These tables support virtualization and allow the hypervisor to virtualize the view of physical memory that is seen by a virtual machine (VM).

Armv9-A supports all the virtual addresses spaces described above for Armv8-A. Armv9-A introduces the optional Realm Management Extension (RME). When RME is implemented, additional translation regimes are also present:

- Realm EL1 and EL0
- Realm EL2 and EL0
- Realm EL2

In virtualization, we call the set of translations that are controlled by the OS, Stage 1. The Stage 1 tables translate virtual addresses to intermediate physical addresses (IPAs). In Stage 1 the OS thinks that the IPAs are physical address spaces. However, the hypervisor controls a second set of translations, which we call Stage 2. This second set of tables translates IPAs to physical addresses. This diagram shows how the two sets of translations work:

Figure 5-2: Stage 2 IPAs to physical addresses



Although there are some minor differences in the table format, the process of Stage 1 and Stage 2 translation is usually the same.

Note

At Arm, we use the address 0x8000 in many of our examples. 0x8000 is also the default address for linking with the Arm linker, armlink. The address comes from an early microcomputer, the BBC Micro Model B, which had ROM (and sideways RAM)

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

at the address `0x8000`. The BBC Micro Model B was built by a company called Acorn, which developed the Acorn RISC Machine (ARM), and later became Arm.

5.1 Physical Addresses

As well as multiple virtual address spaces, AArch64 also has multiple physical address spaces (PAS):

- Non-secure PAS
- Secure PAS
- Realm PAS (Armv9-A only)
- Root PAS (Armv9-A only)

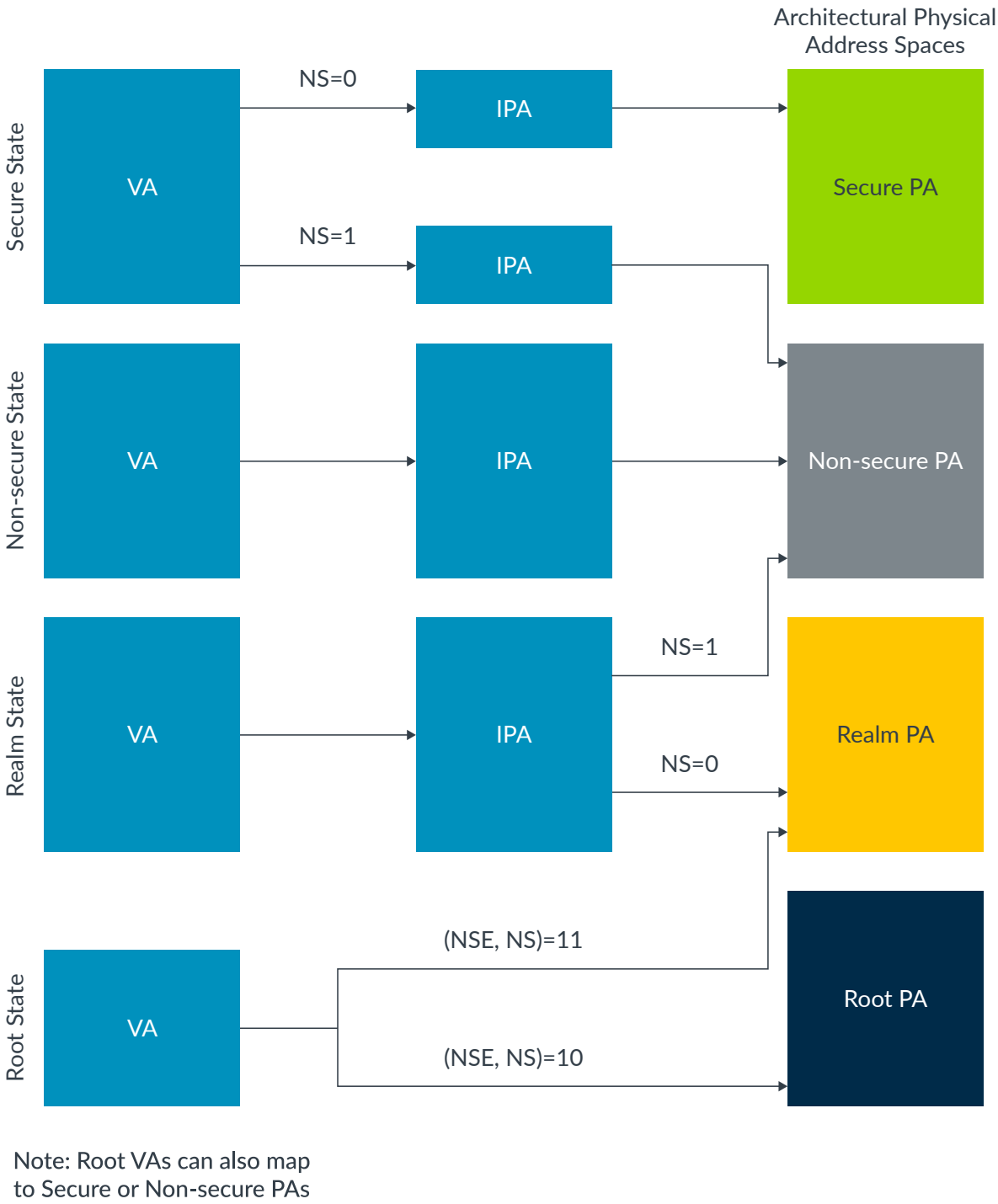
Which physical address space, or spaces, a virtual address can be mapped to depends on the current Security state of the processor. The following list shows the Security states with its corresponding virtual address mapping destinations:

- Non-secure state: virtual addresses can only map to Non-secure physical addresses.
- Secure state: virtual addresses can map to Secure or Non-secure physical addresses.
- Realm state: virtual addresses can map to Realm or Non-secure physical addresses.
- Root state: virtual address can map to any physical address space.

When the PE is in a Security state that can access multiple physical address spaces, the translation table entries control which output physical address space is used. The following diagram shows the mapping of multiple physical address spaces:

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 5-3: Mapping of multiple physical address spaces



Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.2 Address sizes

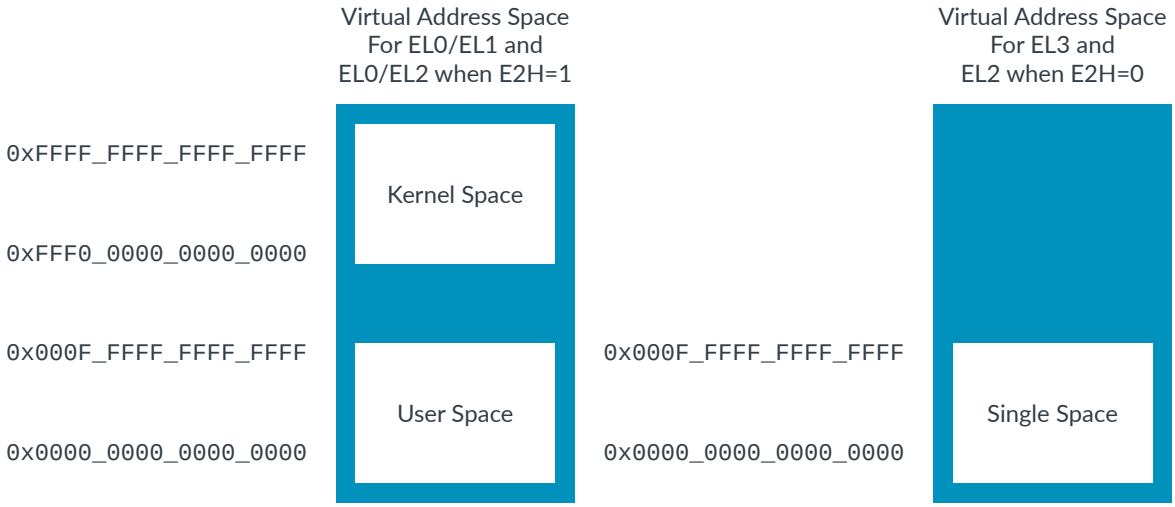
AArch64 is a 64-bit architecture, but this does not mean that all addresses are 64-bit.

5.3 Size of virtual addresses

Virtual addresses are stored in a 64-bit format. As a result, the address in load instructions (LDR) and store instructions (STR) is always specified in an X register. However, not all of the addresses in the X register are valid.

This diagram shows the layout of the virtual address space in AArch64:

Figure 5-4: Size of virtual addresses



There are two regions for the EL0/EL1 virtual address space: by convention, they are used as kernel space and application space. These two regions are shown on the left-hand side of the diagram, with kernel space at the top, and application space, which is labelled 'User space', at the bottom of the address space. Kernel space and user space have separate translation tables and this means that their mappings can be kept separate.

There is a single region at the bottom of the address space for all other Exception levels. This region is shown on the right-hand side of the diagram and is single space box.

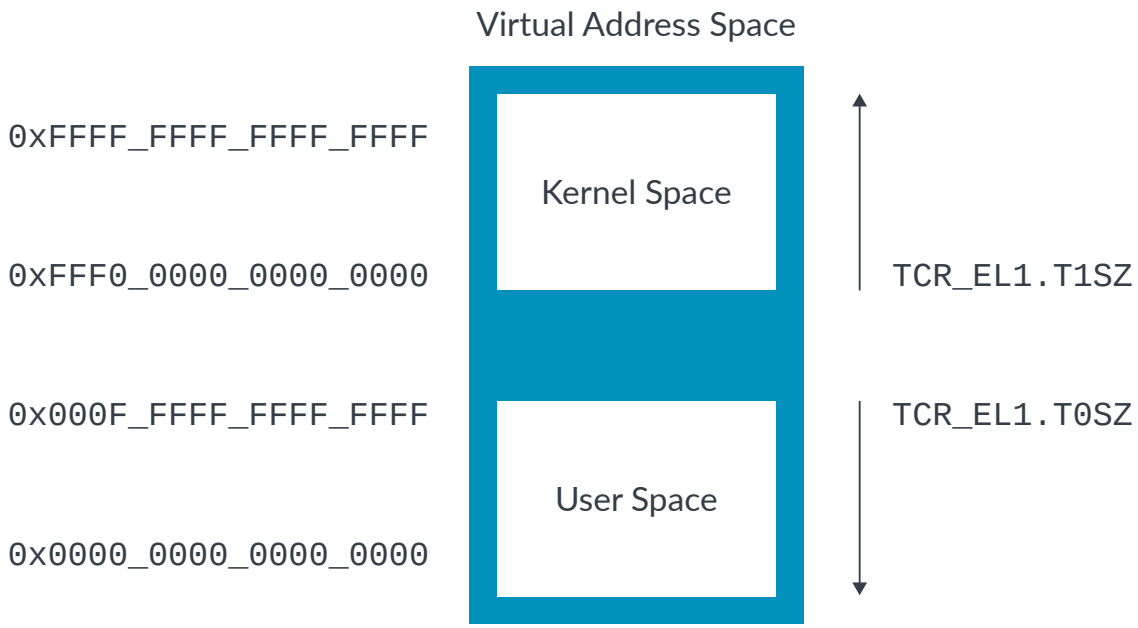

Note

If you set HCR_EL2.E2H to 1 it enables a configuration where a host OS runs in EL2, and the applications of the host OS run in EL0. In this scenario, EL2 also has an upper and a lower region.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Each region of address space has a size of up to 55-bits. However, each region can be independently shrunk to a smaller size. The TnSZ fields in the TCR_ELx registers control the size of the virtual address space. For example, this diagram shows that TCR_EL1 controls the EL0/EL1 virtual address space:

Figure 5-5: Virtual address space



The virtual address size is encoded as:

```
virtual address size in bytes = 2^(64-TCR_ELx.TnSZ)
```

The virtual address size can also be expressed as a number of address bits:

```
Number of address bits = 64 - TnSZ
```

Therefore, if `TCR_EL1.T1SZ` is set to 32, the size of the kernel region in the EL0/EL1 virtual address space is 2^{32} bytes (`0xFFFF_FFFF_0000_0000` to `0xFFFF_FFFF_FFFF_FFFF`). Any address that is outside of the configured range or ranges will, when it is accessed, generate an exception as a translation fault. The advantage of this configuration is that we only need to describe as much of the address space as we want to use, which saves time and space. For example, imagine that the OS kernel needs 1GB of address space (30-bit address size) for its kernel space. If the OS sets T1SZ to 34, then only the translation table entries to describe 1GB are created, as 64 - 34 = 30.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

All Armv8-A implementations support 48-bit virtual addresses. Support for 52-bit or 56-bit virtual addresses is optional and reported by ID_AA64MMFR2_EL1. At the time of writing, none of the Arm Cortex-A processors support 52-bit virtual addresses.

5.4 Size of physical addresses

The size of a physical address is **IMPLEMENTATION DEFINED**, up to a maximum of 56 bits. The ID_AA64MMFR0_EL1 register reports the size that is implemented by the processor. For Arm Cortex-A processors, this will usually be 40 bits or 44 bits.


Note

In Armv8.0-A, the maximum size for a physical address is 48 bits. This was extended to 56 bits in Armv9.3-A.

5.5 Size of intermediate physical addresses

If you specify an output address in a translation table entry that is larger than the implemented maximum, the *Memory Management Unit* (MMU) will generate an exception as an address size fault.

The size of the IPA space can be configured in the same way as the virtual address space. VTCR_EL2.T0SZ controls the size. The maximum size that can be configured is the same as the physical address size that is supported by the processor. This means that you cannot configure a larger IPA space than the supported physical address space.

5.6 Address Space Identifiers - Tagging translations with the owning process

Many modern OSs have applications that all seem to run from the same address region, this is what we have described as user space. In practice, different applications require different mappings. This means, for example, that the translation for VA `0x8000` depends on which application is currently running.

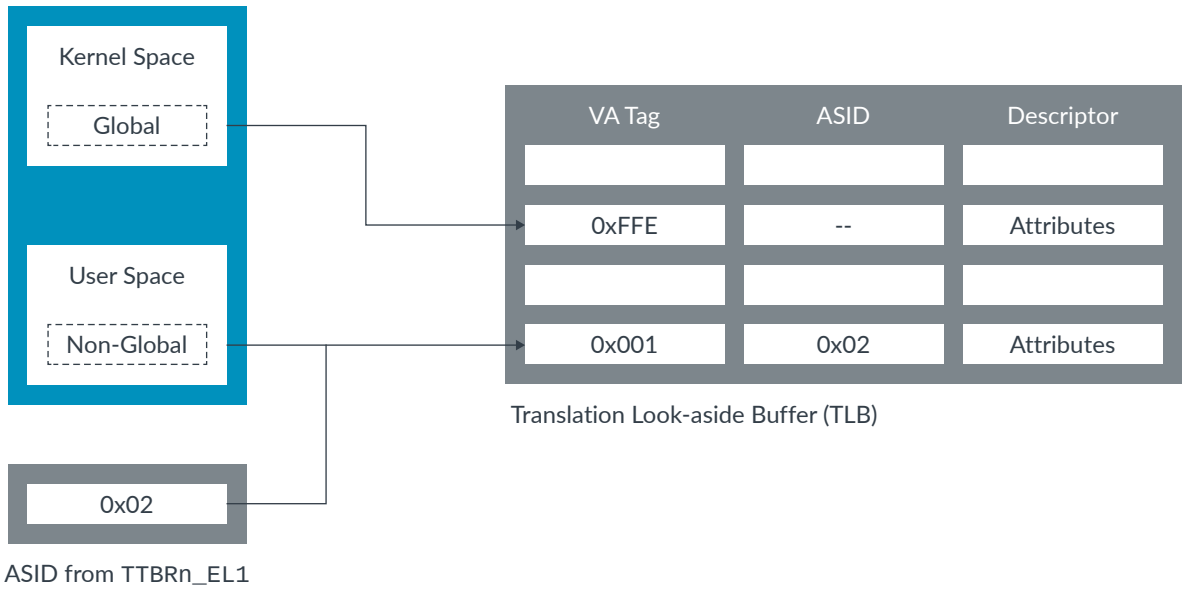
Ideally, we would like the translations for different applications to coexist within the Translation Lookaside Buffers (TLBs), to prevent the need for TLB invalidates on a context switch. But how would the processor know which version of the VA `0x8000` translation to use? In AArch64, the answer is Address Space Identifiers (ASIDs).

For the EL0/EL1 virtual address space, translations can be marked as Global (G) or Non-Global (nG) using the nG bit in the attributes field of the translation table entry. For example, kernel mappings are Global translations, and application mappings are Non-Global translations. Global translations apply to whichever application is currently running. Non-Global translations only apply to a specific application.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Non-Global mappings are tagged with an ASID in the TLBs. On a TLB lookup, the ASID in the TLB entry is compared with the currently selected ASID. If they do not match, then the TLB entry is not used. This diagram shows a Global mapping in the kernel space with no ASID tag and a non-Global mapping in user space with an ASID tag:

Figure 5-6: Translation look-aside buffer



The diagram shows that TLB entries for multiple applications are allowed to coexist in the cache, and the ASID determines which entry to use.

The ASID is stored in one of the two TTBRn_EL1 registers. Usually TTBR0_EL1 is used for user space. As a result, a single register update can change both the ASID and the translation table that it points to.

In 2023 Arm introduced the ability to specify two ASIDs simultaneously. From Armv9.5-A, software has the option to use the ASID field in both TTBRn_EL1 registers simultaneously. TTBR0_EL1.ASID is applied to addresses in the lower portion of the virtual address space and TTBR1_EL1.ASID to addresses in the upper portion of the virtual address space.



Note

ASID tagging is also available in EL2, when HCR_EL2.E2H==1.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.7 Virtual Machine Identifiers - Tagging translations with the owning VM

EL0/EL1 translations can also be tagged with a Virtual Machine Identifier (VMID). VMIDs allow translations from different VMs to coexist in the cache. This is similar to the way in which ASIDs work for translations from different applications. In practice, this means that some translations will be tagged with both a VMID and an ASID, and that both must match for the TLB entry to be used.


Note

When virtualization is supported for a security state, EL0/EL1 translations are always tagged with a VMID - even if Stage 2 translation is not enabled. This means that if you are writing initialization code and are not using a hypervisor, it is important to set a known VMID value before setting up the Stage 1 MMU.

5.8 Common not Private

If a system includes multiple processors, do the ASIDs and VMIDs used on one processor have the same meaning on other processors?

For Armv8.0-A the answer is that they do not have to mean the same thing. There is no requirement for software to use a given ASID in the same way across multiple processors. For example, ASID 5 might be used by the calculator on one processor and by the web browser on another processor. This means that a TLB entry that is created by one processor cannot be used by another processor.

In practice, it is unlikely that software will use ASIDs differently across processors. It is more common for software to use ASIDs and VMIDs in the same way on all processors in a given system. Therefore, Armv8.2-A introduced the *Common not Private* (CnP) bit in the *Translation Table Base Register* (TTBR). When the CnP bit is set, the software promises to use the ASIDs and VMIDs in the same way on all processors, which allows the TLB entries that are created by one processor to be used by another.


Note

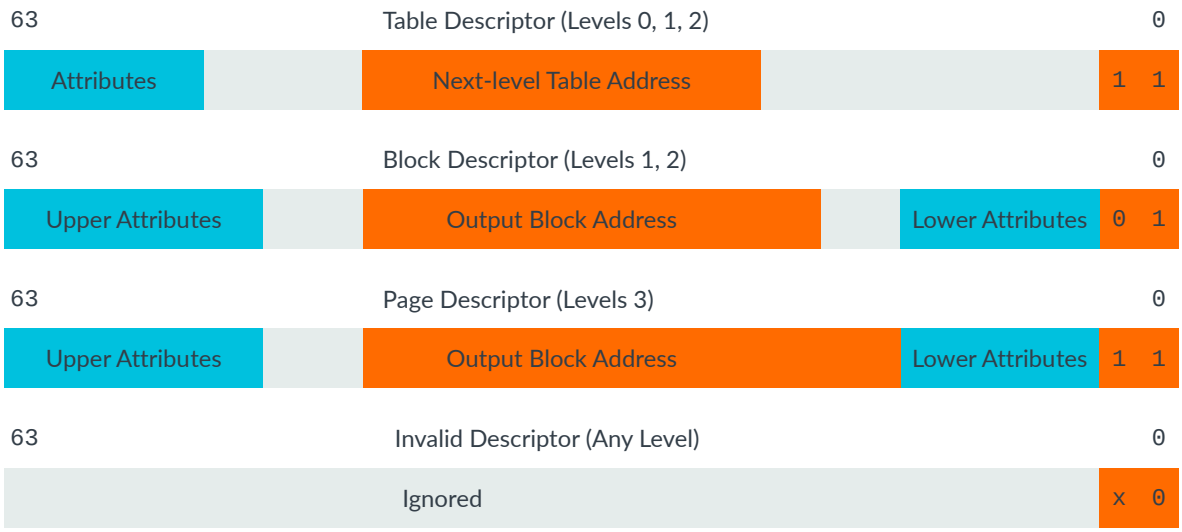
We have been talking about processors, however, technically, we should be using the term, *Processing Element* (PE). PE is a generic term for any machine that implements the Arm architecture. It is important here because there are microarchitectural reasons why sharing TLBs between processors would be difficult. But within a multithreaded processor, where each hardware thread is a PE, it is much more desirable to share TLB entries.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6. Controlling address translation Translation table format

Here we can see the different formats that are allowed for translation table entries:

Figure 6-1: Translation table format



Note

For purposes of clarity, this diagram does not specify the width of bit fields. You can find this information in the Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile: The VMSAv8-64 translation table format descriptors.

Each entry is 64 bits and the bottom two bits determine the type of entry.

Notice that some of the table entries are only valid at specific levels. The maximum number of levels of tables is four, which is why there is no table descriptor for level 3 (or the fourth level), tables. Similarly, there are no Block descriptors or Page descriptors for level 0. Because level 0 entry covers a large region of virtual address space, it does not make sense to allow blocks.

Note

The encoding for the Table descriptor at levels 0-2 is the same as the Page descriptor at level 3. This encoding allows 'recursive tables', which point back to themselves. This is useful because it makes it easy to calculate the virtual address of a particular page table entry so that it can be updated.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

7. Translation granule

A translation granule is the smallest block of memory that can be described. Nothing smaller can be described, only larger blocks, which are multiples of the granule.

AArch64 supports three different granule sizes: 4KB, 16KB, and 64KB.

The granule sizes that a processor supports are IMPLEMENTATION DEFINED and are reported by ID_AA64MMFR0_EL1. All Arm Cortex-A processors support 4KB and 64KB. The selected granule is the smallest block that can be described in the last level table. Larger blocks can also be described. This table shows the different block sizes for each level of table based on the selected granule:

Level of table	4KB granule	4KB granule	16KB granule	16KB granule	64KB granule	64KB granule
	Size per entry	Bits used to index	Size per entry	Bits used to index	Size per entry	Bits used to index
0	512GB	47:39	128TB	47	–	–
1	1GB	38:30	64GB	46:36	4TB	51:42
2	2MB	29:21	32MB	35:25	512MB	41:29
3	4KB	20:12	16KB	24:14	64KB	28:16

Before the introduction of Armv9.2-A and Armv8.7-A, there were restrictions on using 52-bit addresses. When the selected granule is 4KB or 16KB, the maximum virtual address region size is 48 bits. Similarly, output physical addresses are limited to 48 bits. It is only when the 64KB granule is used that the full 52 bits can be used.


Note

TCR_EL1 has two separate fields that control the granule size for the kernel space and the user space virtual address ranges. These fields are called TG1 for kernel space and TG0 for user space. A potential problem for programmers is that these two fields have different encodings.

7.1 The starting level of address translation

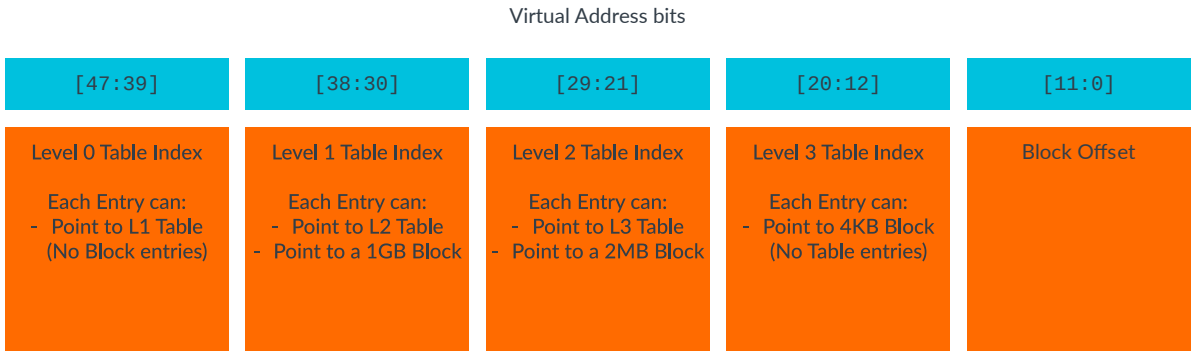
Together, the granule and the size of the virtual address space control the starting level of address translation.

The previous table summarized the block size (size of virtual address range covered by a single entry) for each granule at each level of table. From the block size, you can work out which bits of the virtual address are used to index each level of table.

Let us take the 4KB granule as an example. This diagram shows the bits that are used to index the different levels of table for a 4KB granule:

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 7-1: Starting level of translation



Imagine that, for a configuration, you set the size of the virtual address space, TCR_ELx.T0SZ, to 32. Then the size of the virtual address space, in address bits, is calculated as:

64 - T0SZ = 32-bit address space (address bits 31:0)

If we look at the previous 4KB granule diagram again, level 0 is indexed by bits 47:39. With a 32-bit address space the bits are present but they are zero. Therefore, the starting level of translation for your configuration is level 1.

Next, imagine you set T0SZ to 34:

64 - T0SZ = 30-bit address space (address bits 29:0)

This time, you do not have any other bits that are used to index the level 0 table or the level 1 table, so the starting level of translation for your configuration is level 2.

As the previous diagram shows, when the size of the virtual address space reduces, you need fewer levels of tables to describe it.

These examples are based on using the 4KB granule. The same principle applies when using 16KB and 64KB granules, but the address bits change.

7.2 Registers that control address translation

Address translation is controlled by a combination of system registers:

- SCTLR_ELx

  M - Enable Memory Management Unit (MMU).

  C - Enable for data and unified caches.

  EE - Endianness of translation table walks.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- TTBR0_ELx and TTBR1_ELx

  BADDR - Physical address (PA) (or intermediate physical address, IPA, for EL0/EL1) of start of translation table.

  ASID - The Address Space Identifier for Non-Global translations.
- TCR_ELx

  PS/IPS - Size of PA or IPA space, the maximum output address size.

  TnSZ - Size of address space covered by table.

  TGn - Granule size.

  SH/IRGN/ORGN - Cacheability and shareability to be used by MMU table walks.

  EPDn - Disabling of table walks to a specific table.
- MAIR_ELx

  Attr - Controls the Type and cacheability in Stage 1 tables.

7.3 MMU disabled

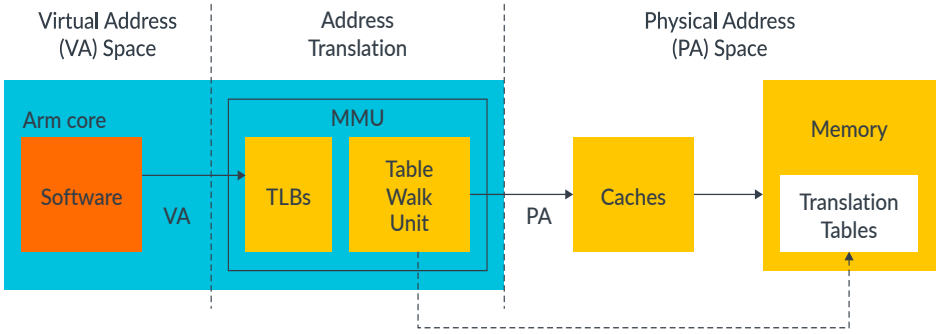
When the MMU is disabled for a stage of translation, all addresses are flat-mapped. Flat mapping means that the input and output addresses are the same.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

8. TLBs and Changing translations

While the MMU is enabled, all data accesses and instructions fetches are translated by the MMU and checked for permissions. To avoid having to re-walk the tables on every access, processors cache recently used translations in the Translation Lookaside Buffers (TLBs).

Figure 8-1: A diagram showing The MMU




Note

The TLBs are caches of translations, not caches of the translation tables. The difference is subtle. Several register fields control how the translation table entries are interpreted. What is in a TLB entry is the interpretation of the translation table entry given the configuration at the point that the tables were walked. In the Arm Architecture Reference Manual (Arm ARM), such register fields are described as 'permitted to be cached in a TLB'.

If you change a translation table entry, or the controls that affect how entries are interpreted, then you need to invalidate the affected entries in the TLB. If you do not invalidate those entries, then the processor might continue to use the old translation.

8.1 Updating translations

The processor is not permitted to cache a translation into the TLBs that results in any of the following faults:

- A translation fault (unmapped address).
- An address size fault (address outside of range).
- An access flag fault. This generally means that when making an invalid translation valid, not TLBI maintenance is necessary.

When un-mapping a valid translation or changing a currently valid translation, TLBI maintenance is required.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Here is an example of un-mapping:

```
STR X11, [X1]  ; update the translation table entry, marking location as invalid
DSB ISH        ; ensures visibility of the update to translation table walks
TLBI VAE1, X10 ; assumes we are in the EL1
DSB ISH        ; ensures completion of the TLB invalidation
ISB            ; synchronise context on this PE

; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using the old mapping have been completed
```

Here is an example of changing a valid translation to be a different valid translation:

```
STR X11, [X1]  ; remove previous mapping, marking location as invalid
DSB ISH        ; ensures visibility of the update to translation table walks
TLBI VAE1, X10 ; assumes we are in the EL1
DSB ISH        ; ensures completion of the TLB invalidation

; No cached copies of the old translation now exist in any observer in
; the inner shareability domain.
; Any attempt to access the virtual address at this point would result in
; a translation fault, as there is no valid mapping.

STR X11, [X2]  ; write the new mapping to the translation table entry
DSB ISH        ; ensures visibility of the update to translation table walks

; No TLBI required as TLBIs are not permitted to cache translations
; that result in a translation fault
```

This approach is called Break Before Make (BBM). It first unmaps the old translation, performs the necessary TLBI maintenance as in the previous example, then creates the new translation. This approach ensures that at no point in time can you have a mixture of observers using the old and new mapping.

If software does not manage updates to translation tables correctly, it could lead to the TLBs containing multiple cached translations for the same address. Or, different observers in the system using different versions of the translation.

There are cases where BBM is not desirable. For example, devices connected via System MMU (SMMU) can be less tolerant of taking faults and re-attempting the access.

FEAT_BBMx adds support for changing the size of a translation without the need to use BBM. This ability is useful when fusing multiple page mappings into one block mapping, or shattering a block mapping into individual page mappings.

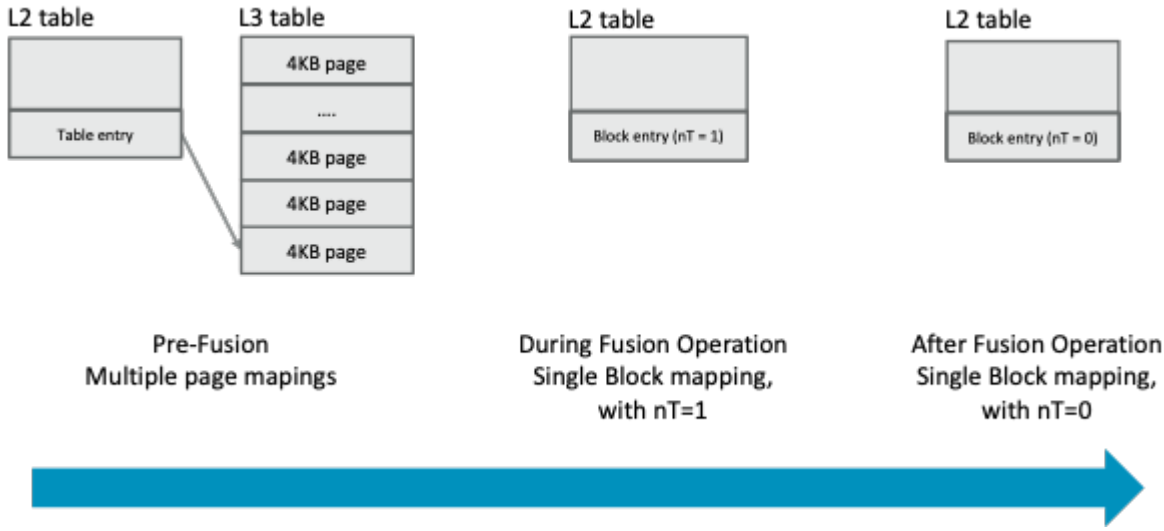
Different levels of support are possible, with support reported via ID_AA64MMFR2_EL1:

- Level 0: Software must use Break Before Make.
- Level 1: Adds support for the nT property.
- Level 2: Adds a guarantee that changing table or block size does not break coherency, ordering guarantees or uniprocessor semantics, or fail to clear the Exclusives monitors.
- Level 3: As level 2, but with a guarantee that changing a Block or Table size does not cause TLBI to be reported.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

For BBM Level 1 and above, the nT property can be used as part to make process of fusing more efficient. To illustrate the use of the nT bit, we will look at an example of page fusing.

Figure 8-2: A diagram showing page fusing



Initially, the virtual address region is mapped as a set of Page mappings. The mapped memory is contiguous in both the input and output address spaces, meaning that a single Block mapping is possible and likely more efficient. We are going to fuse the Page mappings into a single Block mapping. To do this we will:

1. Overwrite the existing Table descriptor in the L2 table with a Block descriptor, with nT=1
    a. Setting the nT bit tells the processor that it might encounter multiple TLB entries for this region.
2. Invalidate the TLBs to remove any remaining cached pre-fusion translations
3. Clear the nT bits
    a. At this point there should be no risk of conflicting entries in the TLB for this region.
4. Invalidate the TLBs to remove any cached translation that still have the nT bit set.

The advantage of this flow compared to traditional break-before-make is that at no time is the translated region invalid.

With BBM L2 and above, the sequence with nT is not needed, the processor can handle changes in the size of translation (such as fusing or shattering).

8.2 Format of a TLB operation

The TLBI instruction is used to invalidate entries in the TLBs. The syntax of this instruction is:

```
TLBI{P} {R}<type><level>{IS|OS} {, <Xt>, <Xt2>}
```

Where:

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- {P} Whether the argument is specified in a single register or a pair of registers
  - Xt2 is only used for the P variants of operations.
- {R} Whether the operation is on an address range, specified as a base address and size.
- < type > Which entries to invalidate.

  All - All entries

  VA - Entry matching VA and ASID in *Xt*

  VAA - Entry matching VA in *Xt*, for any ASID

  ASID - Any entry matching the ASID in *Xt*

  and many more
- < level > Which address space to operate on.

  E1 = EL0/1 virtual address space

  E2 = EL2 virtual address space

  E3 = EL3 virtual address space
- < IS|OS > Whether an operation is Inner Shareable (IS) or Outer Shareable (OS).

  When IS is added to the operation, it is broadcast to the other cores in the Inner Shareable domain.

  When OS is added to the operation, it is broadcast to the other cores in the Outer Shareable domain (Added in Armv8.4-A).
- < Xt > Which address or ASID to operate on.

  Only used for operations by address or ASID.

Consider, for example, an Operating System (OS) that is updating an entry in its kernel translation tables. A typical TLB invalidate sequence would look like this:

```
STR  X1, [X5]        // Write to translation table entry
DSB  ISH             // Barrier instructions - not covered in this guide
TLBI VAAE1IS  , X0   // Invalidate VA specified by X0, in EL0/1
                     // virtual address space for all ASIDs
DSB  ISH             // Barrier instructions - not covered in this guide
ISB                  // Synchronize context on this processor
```

8.3 TLBI Domains

This chapter introduces TLBI domains (FEAT_TLBID), which were added as part of Armv9.7-A.

As described earlier, when an existing translation is changed often a TLB invalidate operation is needed to removed cached copies of the old translation. These invalidate operations need to

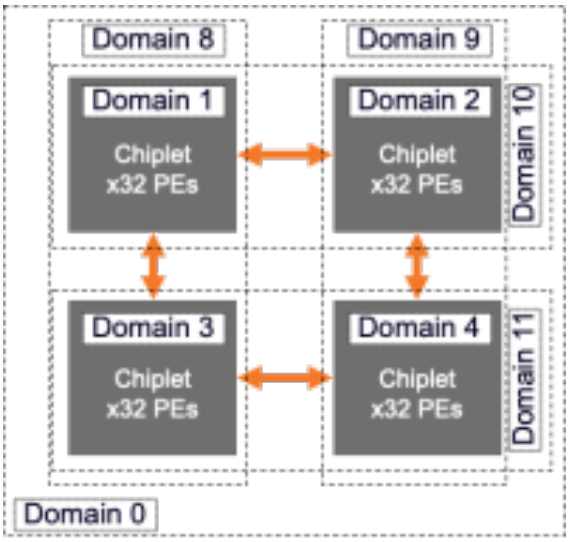
Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

happen on any core, or SMMU, that could have a copy of the old translation cached. To efficiently enable this, the architecture allows TLB invalidate operations to be broadcast to either the Inner or Outer share ability domain.

A given workload, such as an application or virtual machine, might only have run on a known subset of the cores. TLBI domains allow software to further fine tune where a TLB invalidation is broadcast. System designers can divide a system into a set Domains. When executing a TLBI operation, software specifies which Domain the operation should be broadcast to. Meaning that only the cores that are part of that Domain will receive the TLB invalidate operation. The number and composition of TLBI Domains is up to the system designer, and would typically be discoverable via firmware tables.

For example, below is a system that is compromised of four chiplets, with each chiplet containing 64 cores.

Figure 8-3: A diagram showing a system of four chiplets



The system has been divided into a set of Domains:

- Domain 0: On all systems Domain 0 represents all the cores in the Inner Shareability Domain.
- Domains 1 to 4: A Domain per chiplet.
- Domains 8, 9, 10 and 11: A set of domains grouping pairs of chiplets.

The above is just an example. However, in this example software would have option to target invalidation more precisely at the places where the affected workload had operated. This fine grain targeting can reduce the cost and latency of TLB invalidate operations.


Note

In some cases to be able make use of TLBI Domains software must use the TLBIP variant of the operation. See the individual instruction descriptions in the Arm Architecture Reference Manual for more information.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

9. Contiguous mappings

Contiguous mappings enable the MMU to represent a group of adjacent Page or Block descriptors as a single contiguous region for the translation stage. An implementation can cache the group using fewer Translation Lookaside Buffer (TLB) entries, and sometimes reduce to a single TLB entry. This can improve TLB efficiency for that range.

Contiguous mappings give kernels an extra mapping size at each level of the translation tables. These sizes can provide more flexibility when software maps memory and can reduce translation overhead. However, a larger contiguous mapping can waste memory if it covers memory that software does not need to map. The choice depends on the memory mapping requirements. Larger mappings can improve TLB efficiency, while smaller mappings provide finer control over memory allocation.

9.1 Contiguous bit

To enable a contiguous mapping, software sets the Contiguous bit to 1 in each Block or Page descriptor in the contiguous mapping group. The contiguous bit is a hint to the MMU. An implementation is not required to cache the group as a single TLB entry. The contiguous bit was introduced in Armv8-A.

Software must maintain the following constraints when it creates the group:

- Each adjacent entry in the group must set the Contiguous bit to 1.
- The input address (IA) range and output address (OA) range must both be aligned to the total size of the contiguous mapping.
- Each adjacent entry in the group must describe a different adjacent block-sized or page-sized portion of the same contiguous OA range.
- Each adjacent entry in the group must use consistent attributes and permissions.
- The group of entries as a whole must be aligned within the translation table to the total byte size of the descriptors in the group.

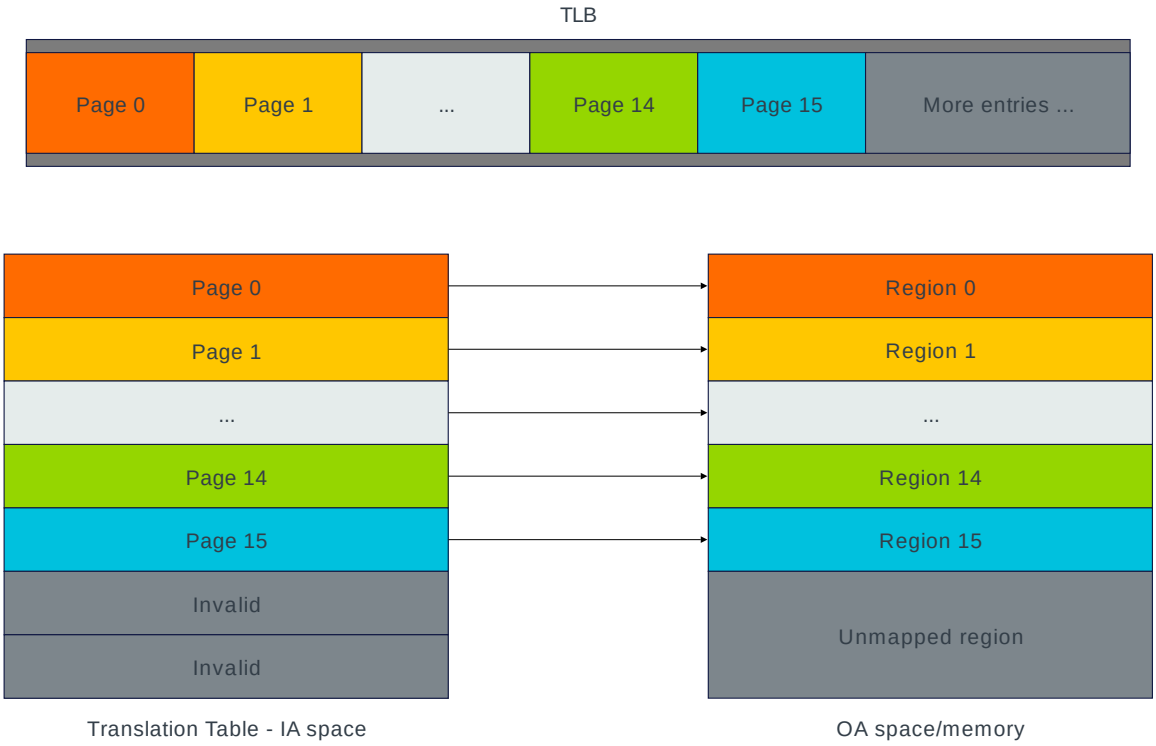

Note

If software does not meet these constraints, the contiguous mapping is misprogrammed. A misprogrammed contiguous mapping can cause TLB conflict aborts or inconsistent translation results.

The diagrams below show how contiguous bit affect the TLB entries:

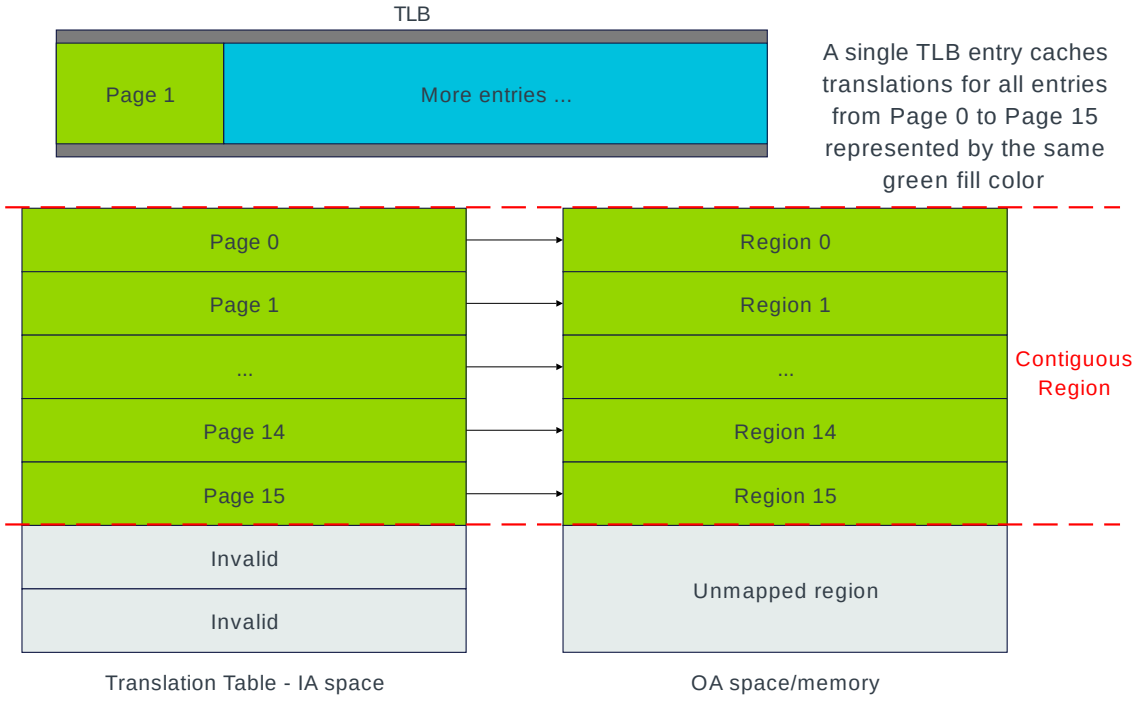
Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 9-1: A diagram showing translation table entries and their respective TLB entries when contiguous bit is set to 1



Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 9-2: A diagram showing translation table entries and their respective TLB entries when contiguous bit is set to 0



The following tables show the contiguous mapping alignment requirements for VMSAv8-64 and VMSAv9-128 translation systems:

Table 9-1: Permitted properties of caching a translation in a TLB when the effective value of the Contiguous bit is 1 for VMSAv8-64 translation system

Translation granule	Block or Page lookup level	Number of adjacent translation table entries	Alignment boundary of adjacent translation table entries within the translation table	Alignment of contiguous OA range
4KB	1	16	128 bytes	16GB
	2	16	128 bytes	32MB
	3	16	128 bytes	64KB
16KB	2	32	256 bytes	1GB
	3	128	1KB	2MB
64KB	2	32	256 bytes	16GB
	3	32	256 bytes	2MB

Table 9-2: Permitted properties of caching a translation in a TLB when the effective value of the Contiguous bit is 1 for VMSAv9-128 translation system

Translation granule	Block or Page lookup level	Number of adjacent translation table entries	Alignment boundary of adjacent translation table entries within the translation table	Alignment of contiguous OA range
4KB	1	4	64 bytes	1GB

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Translation granule	Block or Page lookup level	Number of adjacent translation table entries	Alignment boundary of adjacent translation table entries within the translation table	Alignment of contiguous OA range
	2	16	256 bytes	16MB
	3	16	256 bytes	64KB
16KB	1	4	64 bytes	64GB
	2	16	256 bytes	256MB
	3	64	1KB	1MB
64KB	2	64	1KB	16GB
	3	16	256 bytes	1MB

9.2 FEAT_CCONT

Linux folios show that grouping physically contiguous memory can improve performance. Folios simplify kernel memory management and can increase TLB efficiency. However, the contiguous hint introduced in Armv8-A that we have covered so far supports only fixed contiguous region sizes. Fixed contiguous region sizes limit how often OS kernels can use contiguous mappings without wasting memory.

FEAT_CCONT, configurable contiguous region sizes, is an architecture feature. FEAT_CCONT enables supervisory software, including Linux, to configure the MMU to use a wider range of contiguous region sizes. The Linux kernel can use these sizes to describe common contiguous physical memory ranges in page table entries. These sizes can improve translation latency and TLB efficiency while limiting memory waste. FEAT_CCONT also enables software to dynamically change the contiguous size of a region without causing TLB conflict aborts.



Note

FEAT_CCONT is planned for release as part of Future Arm Technologies.

Usage of FEAT_CCONT

Software can check if FEAT_CCONT is implemented by reading the register ID_AA64MMFR3_EL1.CCONT.

A new control register field (V)TCR(2)_ELx.CC is introduced for each stage of each translation regime. The term (V)TCR(2)_ELx.CC means all of:

- TCR2_EL1.CC.
- If EL2 is implemented, TCR2_EL2.CC and VTCR_EL2.CC.
- If EL3 is implemented, TCR_EL3.CC.

For simplicity, this section uses the term CC to refer to the relevant (V)TCR(2)_ELx.CC field. To further clarify, the contiguous hint that was introduced in Armv8-A that was covered in the earlier chapter will now be referred to as “fixed size contiguous mapping”.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

When CC is set to 0, the behavior of fixed size contiguous mappings apply. When CC is set to 1, the meaning of the Contiguous bit is changed such that instead of a fixed number of adjacent Block or Page entries, the number of entries in the group is derived from the least-significant bits of the OA field in the descriptor.

The following tables show the bit indices within the descriptor that derive the contiguous region size, when CC is 1. The bit indices differ for each level of translation walk.

Table 9-3: Descriptor bits used to indicate region size in VMSAv8-64.

Level of walk	TGx=4KB	TGx=16KB	TGx=64KB
1	[37:30]	N/A	N/A
2	[28:21]	[34:25]	[40:29]
3	[19:12]	[23:14]	[27:16]

Table 9-4: Descriptor bits used to indicate region size in VMSAv9-128.

Level of walk	TGx=4KB	TGx=16KB	TGx=64KB
1	[34:28]	[42:34]	N/A
2	[26:20]	[32:24]	[38:28]
3	[18:12]	[22:14]	[26:16]


Note

For each bit range shown, the bit range identifies both the OA slice and the corresponding descriptor bits. Therefore, `OA[A:B] == descriptor[A:B]`.

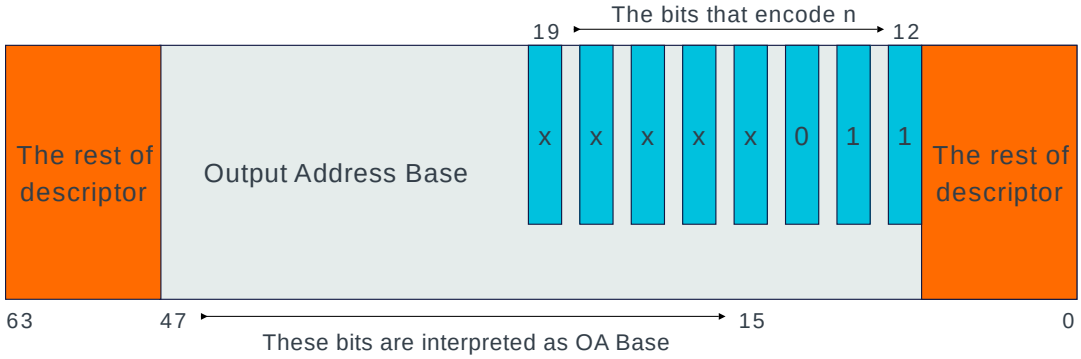
The Contiguous region is naturally aligned by hardware. Therefore, for a contiguous group of adjacent descriptors, software should program the OA field of each descriptor to the same value, rather than incrementing the values as it would when CC is 0.

When CC is 1, the respective bits shown in the tables above determine the size of the contiguous region. The number n of the slice bits that are 1 is counted, starting from the least-significant bit and stopping at the first bit that is 0. If all bits of the slice are 1, the most-significant bit of the slice is treated as 0. The size of the Contiguous region is $2^{(n+1)}$ times the size that would be described by a Block or Page descriptor at the level of the page table walk.

The diagram below illustrates how the contiguous region size (n) bits are positioned and counted.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 9-3: A diagram showing the encoding of n in OA field



n = 2,
mapping size is 2^(n+1) * Page descriptor size,
mapping size = 32KB for 4KB granule

The Contiguous region is aligned to its total size in IA space and OA space. The maximum size of a contiguous region at level of walk L is up to half the size of Block mapping at level L-1.

For example, given a 4KB granule, at level 3 page table, for VMSAv8-64, and n = 0:

$2^{(n+1)}$ x (page/block size) = contiguous region size

$2^{(2+1)}$ x 4KB = contiguous region size

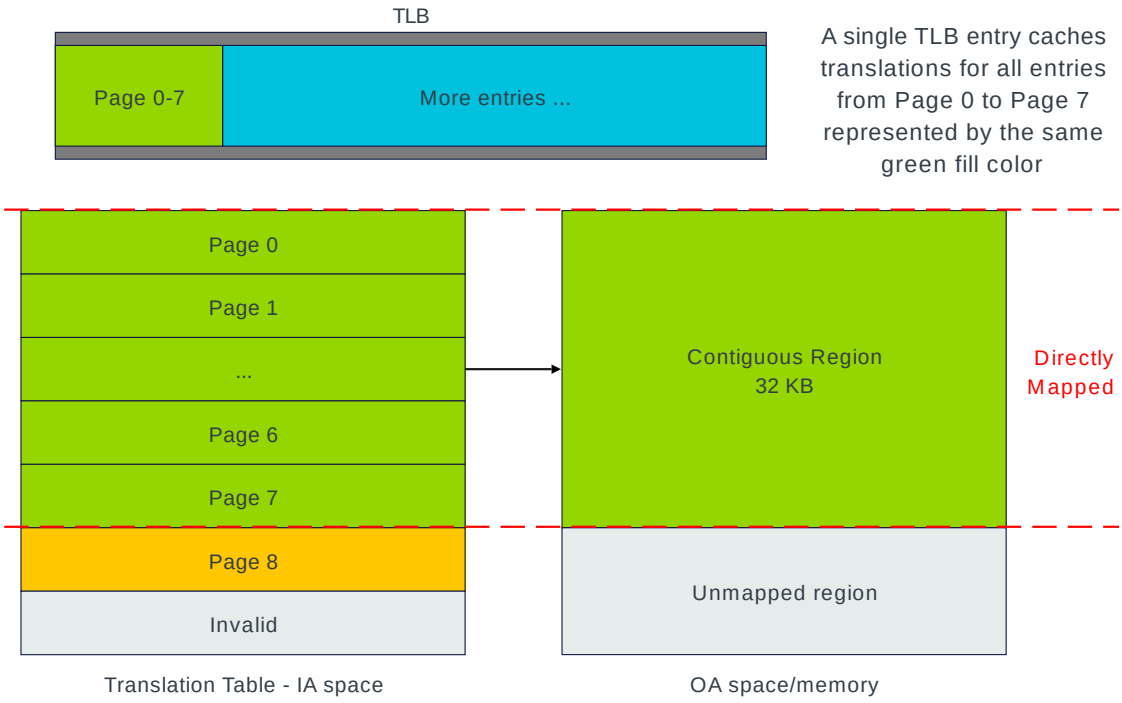
32KB = contiguous region size

For a level 3 page walk, the contiguous region size grows as follows: 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1024MB. The number of contiguous page entries grows as follows: 2, 4, 8, 16, 32, 64, 128, 256.

The constraints mentioned for the fixed size contiguous mapping mostly remains the same, except that the OA base of the adjacent entries now points to the base or the start of the contiguous region as a whole rather than the respective portion of the region. The following diagram illustrates the difference.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The diagram below shows how contiguous mapping is structured in the translation table and is cached in the TLB when FEAT_CCONT is enabled:



Note

When CC is 1, the Contiguous bit is no longer a hint to the MMU. When CC is 1 and the Contiguous bit is 1, the change in behavior of the MMU is guaranteed. This is because FEAT_CCONT changes how the Block and Page descriptors are interpreted and handled during address translation.

Performance guide

Mapping size is a trade-off between average translation latency and memory-allocation granularity. Larger mapping sizes reduce average translation latency, while smaller mapping sizes reduce memory waste by providing finer allocation granularity. To help software choose an efficient mapping size, an implementation can advertise preferred contiguous region sizes in the register CCONTIDR_EL1. Preferred sizes provide a better balance between translation latency and memory usage than non-preferred sizes. However, regardless of preferred sizes, a larger mapping size always provides average translation latency that is less than or equal to that of a smaller mapping size.

To further demonstrate:

```
Preferred Sizes: { 32KB, 64KB, 128KB }

latency(128) <= latency(64) <= latency(32)
```

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
latency(512) <= latency(128)
```

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

10. Address translation instructions

An *Address Translation* (AT) instruction lets the software query the translation for a specific address. The translation that results, including the attributes, is written to the *Physical Address Register*, PAR_EL1.

The syntax of the AT instruction lets you specify which translation regime to use. For example, EL2 can query the EL0/EL1 translation regime. However, EL1 cannot use the AT instruction to query the EL2 translation regime, as this is a breach of privilege.

If the requested translation would have caused a fault, no exception is generated. Instead, the type of fault that would have been generated is recorded in PAR_EL1.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

11. Check your knowledge

Q: What is the difference between a stage and a level in address translation?

A: A stage is the process of translating an input address to an output address. For Stage 1 this is the process of going from VA to IPA and for Stage 2 going from IPA to PA.

A level refers to the tables in a given stage of translation. It is also how a larger block can be subdivided into smaller blocks.

Q: What is the maximum size of a physical address?

A: The maximum size of the physical address space is IMPLEMENTATION DEFINED, and up to 52 bits (since Armv8.2-A).

Q: Which register field controls the size of the virtual address space?

A: `TCR_ELx.TnSZ`, or `VTCR_EL2.T0SZ` for Stage 2.

Q: What is a translation granule, and what are the supported sizes?

A: It is the smallest block of memory that can be described.

The supported sizes are 4KB, 16KB, and 64KB.

Q: What does the TLBI ALLE3 do?

A: It invalidates all the TLB entries for the EL3 virtual address space.

Q: Can a translation table entry that causes a Translation Fault be cached in the TLBs?

A: No, it cannot be stored in the TLBs.

Q: How are addresses mapped when the MMU is disabled?

A: Addresses are flat mapped, so that the input and output addresses are the same.

Q: What is an ASID and when does a TLB entry include an ASID?

A: An ASID is an Address Space Identifier, it identifies which application a translation is associated with. Non-Global mappings (nG=1) are tagged with an ASID in the TLBs.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

12. Related information

Here are some resources related to material in this guide:

Virtualization:

- For Armv8-A and Armv9-A, this topic is covered in Virtualization.

Barriers:

- Learn the architecture - Memory Systems, Ordering, and Barriers

Useful links to training:

- Introduction to Armv8-A
- Memory model overview
- Memory types overview
- What does architecture consist of?

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

13. Next steps

This guide has introduced the concept of memory management, explaining the mapping of virtual to physical addresses. Understanding this information will help you to create your own bare-metal page tables and understand the processes your OS performs when allocating memory.

Memory types and attributes, such as access permissions, are covered in the AArch64 Memory Attributes and Properties guide.

As well as the Memory Management Unit (MMU) in the processor, it is increasingly common to have MMUs for non-processor masters, such as Direct Memory Access (DMA) engines. These are referred to as SMMUs (System MMUs) in Arm systems, and elsewhere as IOMMU.

To keep learning about the Armv8-A architecture, see more in our series of guides.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported,

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm’s trademark usage guidelines at https://www.arm.com/company/policies/trademarks. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

The information in this document is Final, that is for a developed product.

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
0106-00	6 October 2026	Non-Confidential	Updates to support 2026 extensions
0105-00	29 September 2025	Non-Confidential	Updates to support 2025 extensions
0104-00	15 October 2024	Non-Confidential	Minor updates
0103-03	22 November 2023	Non-Confidential	Minor update
0103-02	7 November 2023	Non-Confidential	Minor updates
0103-01	14 June 2023	Non-Confidential	Minor update
0103-00	6 June 2023	Non-Confidential	Minor Correction
0102-00	23 November 2021	Non-Confidential	Armv9-A Content Updates
0101-00	4 March 2021	Non-Confidential	Minor corrections
0100-00	20 June 2019	Non-Confidential	Initial release

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change history

The Change history tables describe the technical changes between released issues of this document in reverse order. Issue numbers match the revision history in Document release information on page 46.

Table 2: Changes between v1.5 and v1.6

Changes	Topics affected
Added a chapter on contiguous mappings	• Contiguous mappings

Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential



We recommend the following. If you do not follow these recommendations your system might not work.



Your system requires the following. If you do not follow these requirements your system will not work.



You are at risk of causing permanent damage to your system or your equipment, or harming yourself.



This information is important and needs your attention.



A useful tip that might make it easier, better or faster to perform a task.



A reminder of something important that relates to the information you are reading.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Arm documents are available on developer.arm.com/documentation.

Confidential documents are only available to licensees, when logged in. Each document link in the following tables provides direct access to the online version of the document.

Copyright © 2019, 2021, 2023–2026 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential