Armv8-A Address Translation

Version 1.1

Revision Information

The following revisions have been made to this User Guide.

Date	Issue	Confidentiality	Change
28 February 2017	0100	Non-Confidential	First release
3 July 2019	0101	Non-Confidential	Section 6.1: corrected TCR_EL2.TG0 reference to VTCR_EL2.TG0

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of Arm® in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by Arm in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Arm shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term Arm is used it means “Arm or any of its subsidiaries as appropriate”.

Confidentiality Status

This document is Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

http://www.arm.com

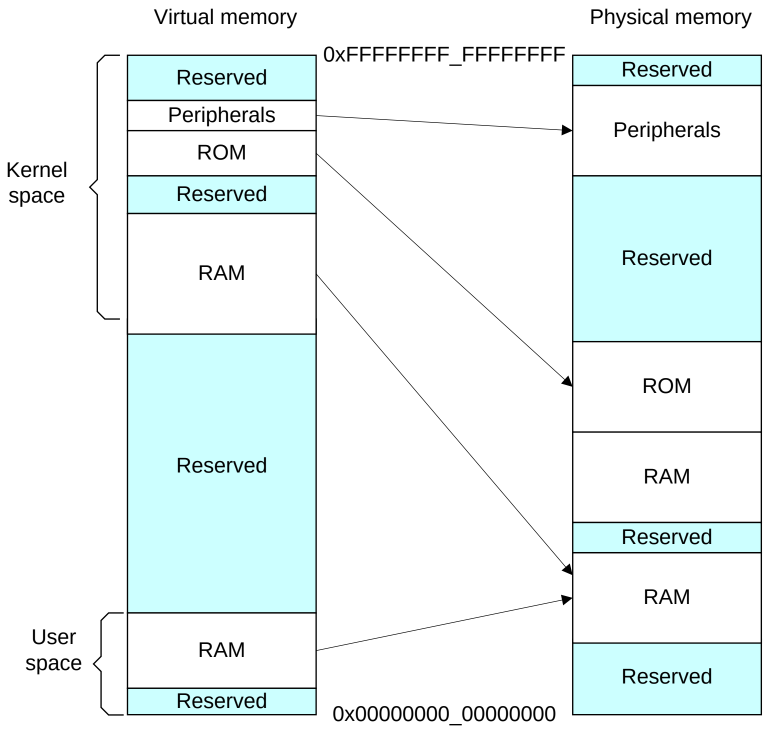
Contents

1 Armv8-A Address translation

Armv8-A uses a *Virtual Memory* system where the addresses used by code (*virtual addresses*) are translated into *physical addresses* which are used by the memory system. This translation is performed by a part of the processor that is called a *Memory Management Unit* (MMU). MMUs in the Arm architecture use translation tables stored in memory to translate virtual addresses to physical addresses. The MMU will automatically read the translation tables when necessary, this process is known as a Table Walk.

An important function of the MMU is to enable the system to run multiple tasks, as independent programs running in their own private virtual memory space. They do not need any knowledge of the physical memory map of the system, that is, the addresses that are used by the hardware, or about other programs that might execute at the same time.

You can use the same virtual memory address space for each program. You can also work with a contiguous virtual memory map, even if the physical memory is fragmented. This virtual address space is separate from the actual physical map of memory in the system. You can write, compile, and link applications to run in the virtual memory space. Different processors and devices in a single system might have different virtual and physical address maps. Privileged software, such as an Operating System, programs the MMU to translate between these two views of memory as the following figure shows.

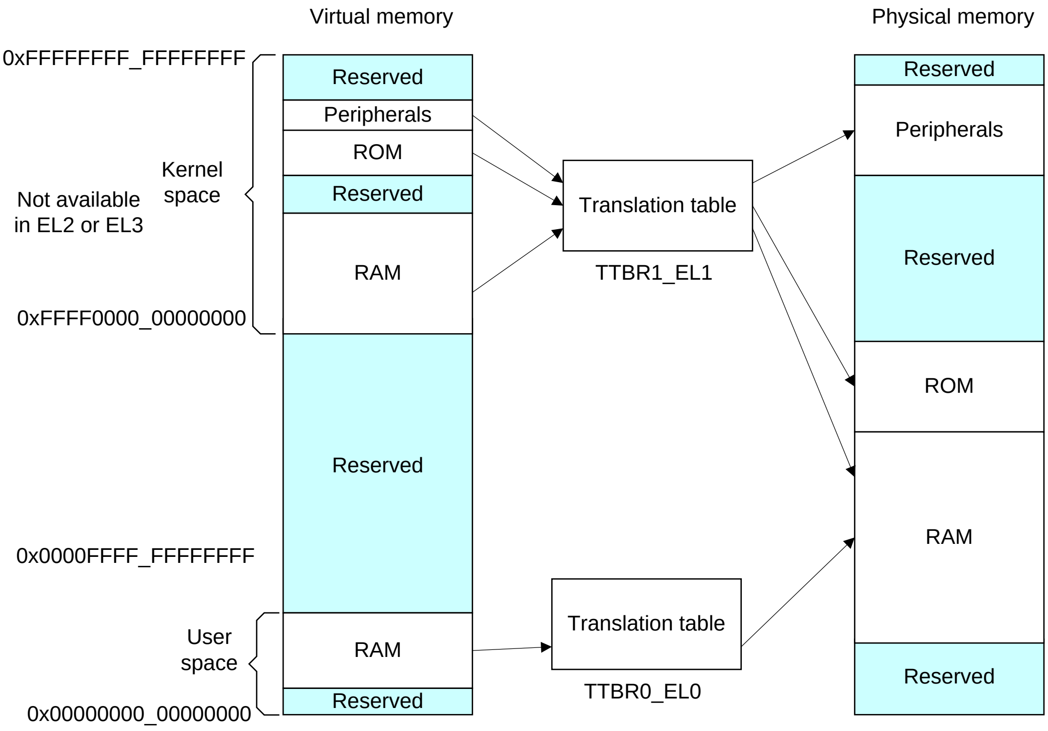


To do this, the hardware in a virtual memory system must provide address translation, which is the translation of the virtual address that is issued by the processor to a physical address in the main memory.

Virtual addresses are those used by you, and the compiler and linker, when placing code in memory. Physical addresses are those used by the actual hardware system.

The MMU uses the most significant bits of the virtual address to index entries in a translation table and establish which block is being accessed. The MMU translates the virtual addresses of code and data to the physical addresses in the actual system. The translation is carried out automatically in hardware and is transparent to the application. In addition to address translation, the MMU controls memory access permissions, memory ordering, and cache policies for each region of memory.

Address translation using translation tables is shown in the following figure:



The MMU enables tasks or applications to be written in a way that requires them to have no knowledge of the physical memory map of the system, or about other programs that might be running simultaneously. This enables the same virtual memory address space to be used for each program. It also works with a contiguous virtual memory map, even if the physical memory is fragmented. This virtual address space is separate from the actual physical map of memory in the system. Applications are written, compiled, and linked to run in the virtual memory space.

2 Separation of kernel and application virtual address spaces

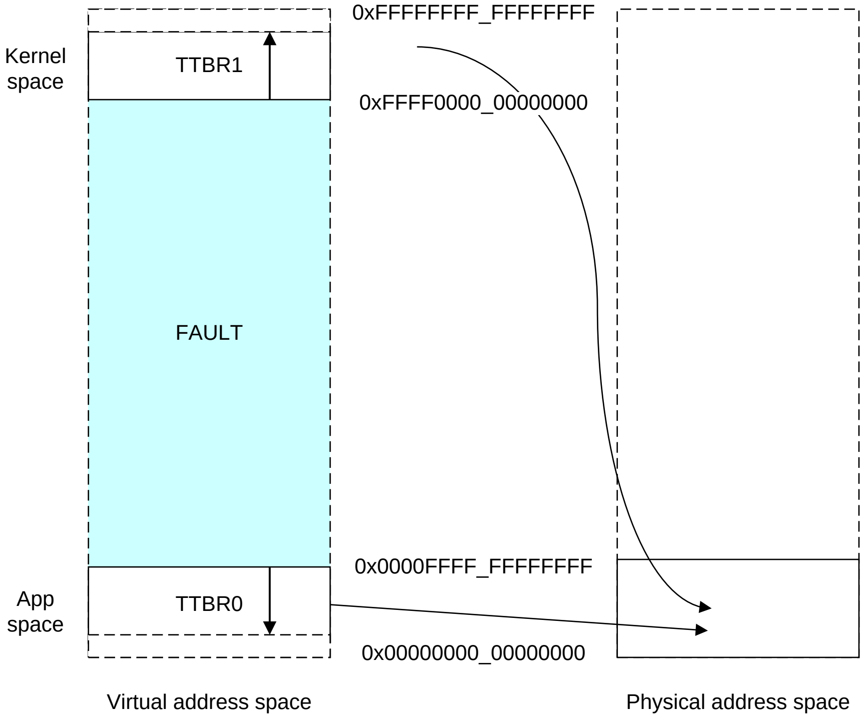
Operating systems typically have several applications or tasks running concurrently. Each of these has its own unique set of translation tables and the kernel switches from one to another as part of the process of context switching between one task and another. However, much of the memory system is used only by the kernel and has fixed virtual to physical address mappings where the translation table entries rarely change. The Armv8-A architecture provides several features to efficiently handle this requirement.

The table base addresses are specified in the *Translation Table Base Registers* (TTBR0_EL1) and (TTBR1_EL1). The translation table pointed to by TTBR0 is selected when the upper bits of the virtual address (VA) are all set to 0. TTBR1 is selected when the upper bits of the VA are all set to 1. You can enable VA tagging to exclude the top 8 bits from the check.

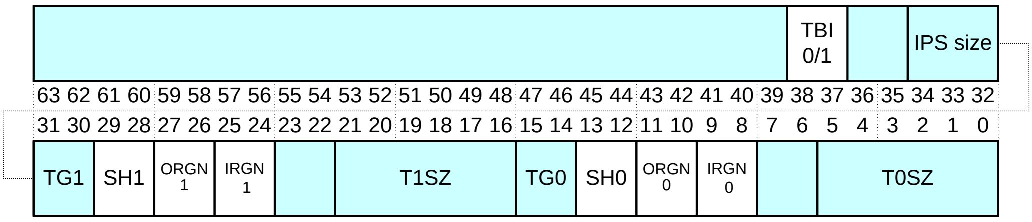
The virtual address from the processor of an instruction fetch or data access is 64 bits. However, you must map both of the two regions that are defined within a single 48-bit physical address memory map.

EL2 and EL3 have a TTBR0, but no TTBR1. This means that is either EL2 or EL3 is using AArch64, they can only use virtual addresses in the range `0x0` to `0x0000FFFF_FFFFFFFF`.

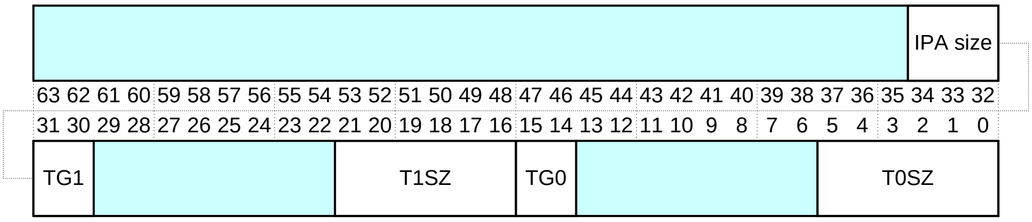
The following figure shows an example of how the kernel space can be mapped to the most significant area of memory while the virtual address space associated with each application can be mapped to the least significant area of memory. However, both of these can be mapped to a much smaller physical address space, as the following figure shows:



The *Translation Control Register* TCR_EL1 defines the exact number of most significant bits that are checked. TCR_EL1 contains the size fields T0SZ[5:0] and T1SZ[5:0]. The integer in the field gives the number of the most significant bits that must be either all 0s or all 1s. There are specified minimum and maximum values for these fields, which vary with granule size and starting table level. Therefore, you must always use both spaces and at least two translation tables are required in all systems. A simple bare metal system without an OS still requires a small upper table that contains only fault entries. This is shown in the following figure:



TCR_EL1 controls other memory management features at EL1 and EL0. The following figure shows only those fields that control address ranges and granule size.



The *Intermediate Physical Address Size* (IPS) field controls the maximum output address size. If translations specify output addresses outside this range, then access is faulted, 000=32 bits of physical address, 101=48 bits. The two-bit *Translation Granule* (TG) TG1 and TG0 fields give the granule size for kernel or user space respectively, 00=4KB, 01=16KB, 11=64KB. The size of the Translation Granule indicates the smallest block of memory that can be independently mapped in the translation tables.

You can configure the level of translation table that is used for the first lookup. The full translation process can require three or four levels of tables. You need not implement all levels. The first level of lookup is, in effect, determined by the granule size and TCR_EL*n*.TxSZ fields. You can configure it separately for TTBR0_EL1 and TTBR1_EL1.

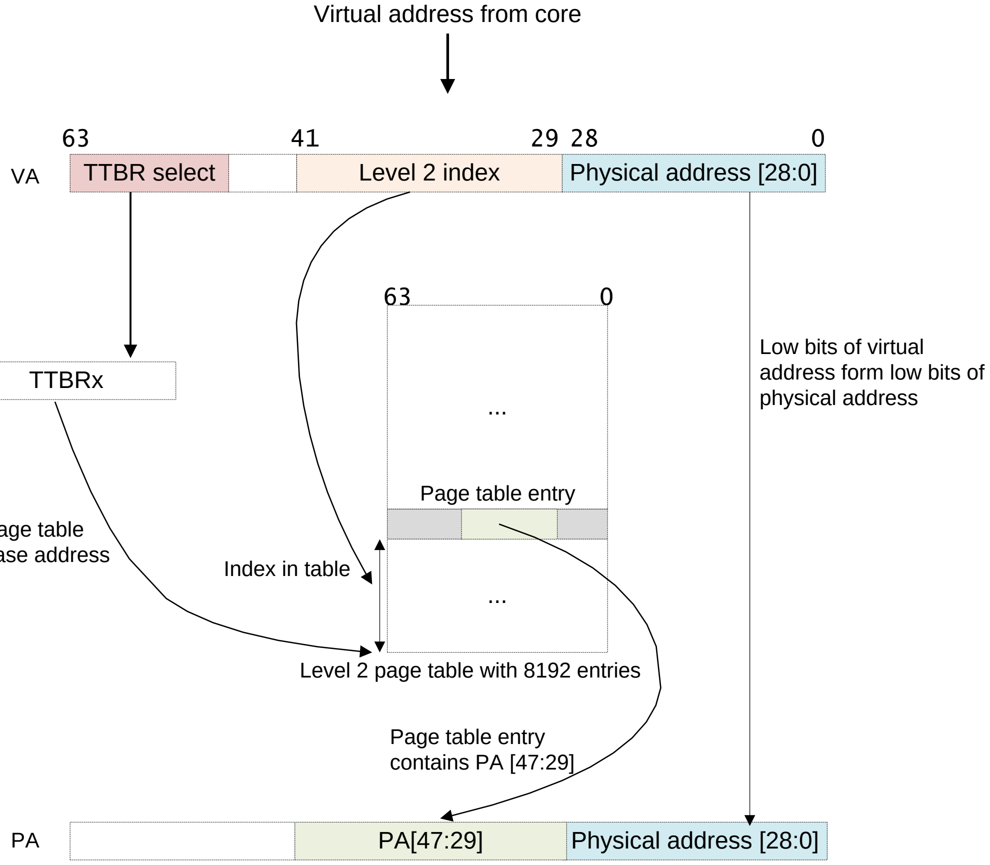
3 Translating a virtual address to a physical address

When the processor issues a virtual address for an instruction fetch, or data access, the MMU hardware translates the virtual address to the corresponding physical address. For a virtual address in an n -bit address space, the top 64-*n* bits VA[63:n] must be all 0s or 1s, otherwise the address triggers a fault.

The least significant bits are then used to give an offset within the selected section, so that the MMU combines the physical address bits from the block table entry with the least significant bits from the original address to produce the final address.

In a simple address translation involving only one level of look-up, and assumes that we are using a 64KB granule with a 42-bit virtual address space. The MMU translates a virtual address as follows:

1. If VA[63:42] = 1 then TTBR1 is used for the base address for the first translation table. When VA[63:42] = 0, TTBR0 is used for the base address for the first translation table.
2. The translation table contains 8192 ×64-bit translation table entries, and is indexed using VA[41:29]. The MMU reads the pertinent Level 2 translation table entry from the table.
3. The MMU checks the translation table entry for validity and whether the requested memory access is allowed. Assuming it is valid, the memory access is allowed.
4. In the following figure, the translation table entry refers to a 512MB page (it is a block descriptor).



Bits [47:29] are taken from this translation table entry and form bits [47:29] of the physical address.

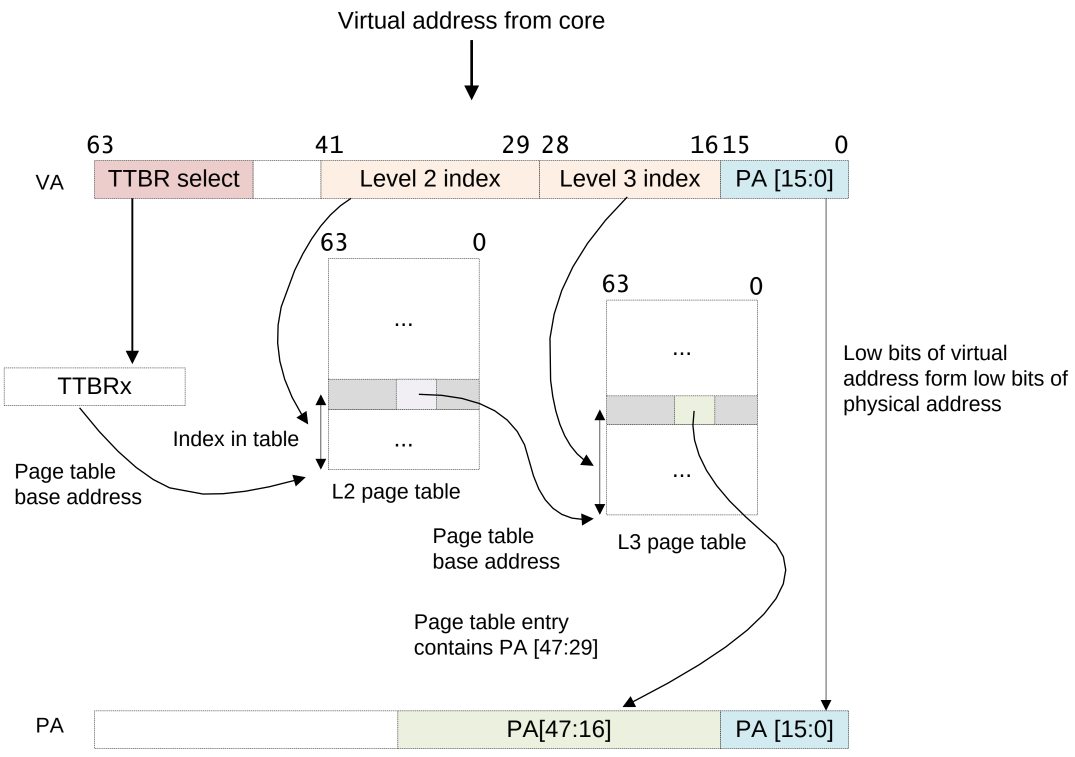
5. Because we have a 512MB page, bits [28:0] of the VA are taken to form PA[28:0].

The full PA[47:0] is returned, along with additional information from the translation table entry.

In practice, such a simple translation process severely limits how finely the address space can be divided.

Instead of using only this first-level translation table, a first-level table entry can also point to a second-level translation table. In this way, an OS can further divide a large section of virtual memory into smaller pages. For a second-level table, the first-level descriptor contains the physical base address of the second-level translation table. The physical address that corresponds to the virtual address requested by the processor is found in the second-level descriptor.

The following figure shows an example of translation for a 64KB granule starting at stage 1, level 2 for a normal 64KB page. It describes a situation where there are two levels of look-up. Again, this assumes a 64KB granule and 42-bit virtual address space.



Each second-level table is associated with one or more first-level entries. You can have multiple first-level descriptors that point to the same second-level table, which means you can alias several virtual locations to the same physical address.

1. If VA[63:42] = 1 then TTBR1 is used for the base address for the first translation table. When VA[63:42] = 0, TTBR0 is used for the base address for the first translation table.

2. The translation table contains 8192 64-bit translation table entries, and is indexed via VA[41:29]. The MMU reads the pertinent level 2 translation table entry from the table.

3. The MMU checks the level 2 translation table entry for validity and whether the requested memory access is allowed. Assuming it is valid, the memory access is allowed.

4. In the figure, the level 2 translation table entry refers to the address of the level 3 translation table (it is a table descriptor).

5. Bits [47:16] are taken from the level 2 translation table entry and form the base address of the level 3 translation table.

6. Bits [28:16] of the VA are used to index the level 3 translation table entry. The MMU reads the pertinent level 3 translation table entry from the table.

7. The MMU checks the level 3 translation table entry for validity and whether the requested memory access is allowed. Assuming it is valid, the memory access is allowed.

8. In the figure, the level 3 translation table entry refers to a 64KB page (it is a page descriptor).

9. Bits [47:16] are taken from the level 3 translation table entry and used to form PA[47:16].

10. Because there is a 64KB page, VA[15:0] is taken to form PA[15:0].

11. The full PA[47:0] is returned, along with additional information from the translation table entries.

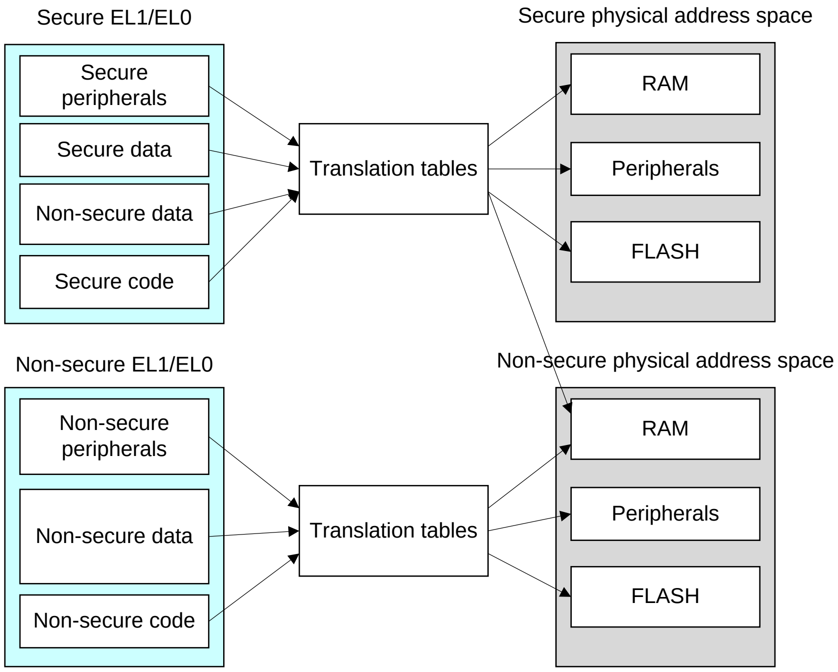
3.1 Secure and Non-secure addresses

The Arm architecture defines two physical address spaces. A Secure address space and a Non-secure address space. In theory the Secure and Non-secure physical address spaces are independent of each other, and exist in parallel. A system could be designed to have two entirely separate memory systems. However, most real systems treat Secure and Non-secure as an attribute for access control. The Normal (Non-secure) world can only access the Non-secure physical address space. The Secure world can access both physical address spaces when the MMU is enabled. This is controlled through translation tables.

This also has cache coherency implications. For example, because Secure 0x8000 and Non-secure 0x8000 are, technically speaking, different physical addresses, they could both be in the cache at the same time.

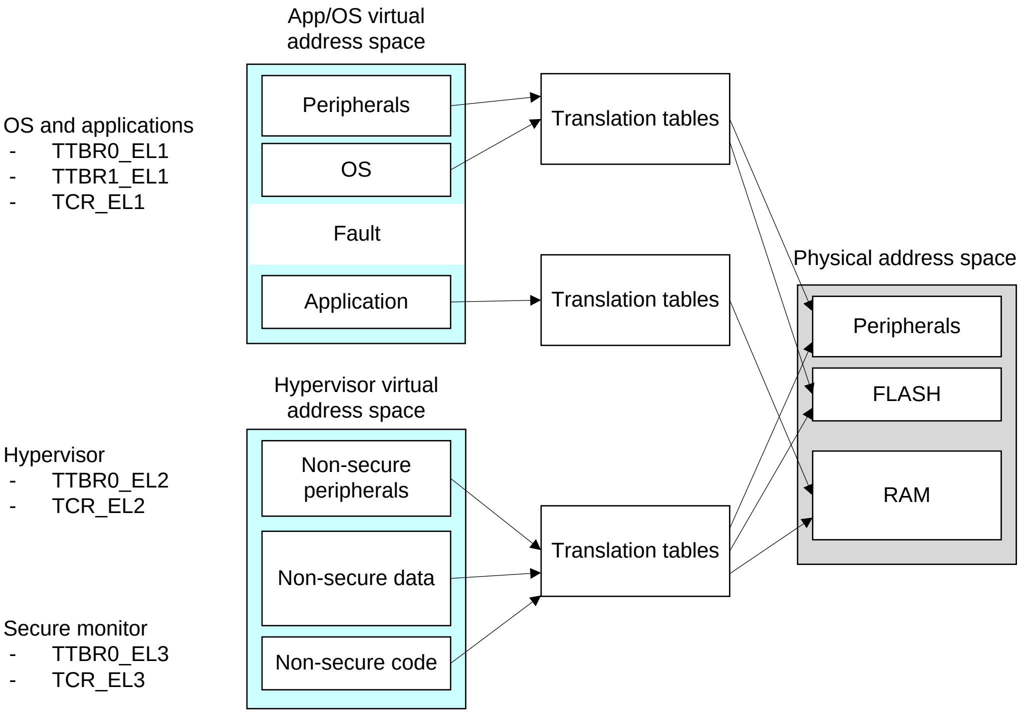
In a system where Secure and Non-secure memory are in different locations, there would be no problem. It is more likely that they would be in the same location. Ideally a memory system would block Secure accesses to Non-secure memory and Non-secure accesses to Secure memory.

In practice most only block Non-secure access to Secure memory. Again, this means you could end up with the same physical memory in the cache twice, Secure, and Non-secure. This is always a programming error. To avoid this the Secure world must always use Non-secure accesses to Non-secure memory.



3.2 Multiple virtual address spaces

At any one time, only one virtual address space is being used (that for the current security state Exception level). However, conceptually, because there are three different TTBRs, there are three parallel virtual address spaces (EL0/1, EL2, and EL3).



You can also have a Secure and Non-secure EL1/0. However, there is only one physical set of TTBR0_EL1, TTBR1_EL1 and TCR_EL1. So when switching between worlds, the Secure monitor has to save and restore these registers.

3.3 Operation when the Memory Management Unit is disabled

When the MMU is enabled the translation tables control the memory type and attributes of memory. When the MMU is not enabled, such as immediately after reset, memory takes a default type.

The default type used for instruction fetches is Normal memory with a cacheability attribute controlled by SCTLR_EL*x*.I:

- When I=0, fetches use the Non-cacheable and Outer Shareable attributes.
- When I=1, the Cacheable, Inner Write-Through Read-Allocate No Write-Allocate, Outer Write-Through Read-Allocate No Write-Allocate Outer Shareable attribute is used.

In systems that use virtualization it might be necessary to allow the Hypervisor to override the default memory types used by guests. When the stage 1 MMU is disabled, for Non-secure EL0 and EL1 accesses when the HCR_EL2.DC bit is set to enable the data cache, the default memory type is Normal, Non-shareable, Inner Write-Back, read and Write Allocate, Outer Write-Back read and Write Allocate. This might be useful in situations where caches have already been configured by a Hypervisor.

3.4 Configuring and enabling the MMU

Writes to the system registers controlling the MMU are context-changing events and there are no ordering requirements between them. The results of these events are not guaranteed to be seen until a context synchronization event.

```
MSR TTBR0_EL1, X0                   // Set TTBR0
MSR TTBR1_EL1, X1                   // Set TTBR1
MSR TCR_EL1, X2                     // Set TCR
ISB                                 // The ISB forces these changes to be
                                    // seen before the MMU is enabled.
MRS X0, SCTLR_EL1                   // Read System Control Register
                                    // configuration data
ORR X0, X0, #1                      // Set [M] bit and enable the MMU.
MSR SCTLR_EL1, X0                   // Write System Control Register
                                    // configuration data
ISB                                 // The ISB forces these changes to be
                                    // seen by the next instruction
```

This is separate from the requirement for flat mapping, which is to make sure that we know which instruction is executed directly after the write to SCTLR_EL1.M. If we see the result of the write it is the instruction at VA+4 using the new translation regime. If the result is not seen, it is still the instruction at VA+4 but where the VA = PA. The ISB does not help here as it cannot be guaranteed to be the next instruction that is executed unless memory is flat mapped.

4 Translation tables in Armv8-A

Armv8-A supports three different sets of translation table format:

- The Armv8-A AArch64 Long Descriptor format.
- The Armv7-A Long Descriptor format such as the *Large Physical Address Extension* (LPAE) to the Armv7-A architecture, found in, for example, the Arm Cortex-A15 processor.
- The Armv7-A Short Descriptor format.

In AArch32 state, the existing Armv7-A long and short descriptor formats can be used to run existing guest operating systems and existing application code without modification. The Armv7-A short descriptors can only be used at EL0 and EL1 stage 1 translations. They cannot be used by hypervisors or Secure monitor code.

The Armv8-A long descriptor format is always used in AArch64 Execution state. This is similar to the Armv7-A long descriptor format with Large Physical Address Extensions. It uses the same 64-bit long-descriptor format, but with some changes. It introduces a Level 0 table index, which uses the same descriptor format as the level 1 table. There is added support for up to 48-bit input and output addresses. The input virtual address is now 64-bit.

However, as the architecture does not support full 64-bit addressing, bits [63:48] of the address must all be the same, that is, all 0s or all 1s, or the top 8 bits can be used for VA tagging.

AArch64 supports three different translation granules. These define the block size at the lowest level of translation table and control the size of translation tables in use. Larger granule sizes reduce the number of levels of translation table required and this can become an important consideration in systems using a hypervisor to provide virtualization.

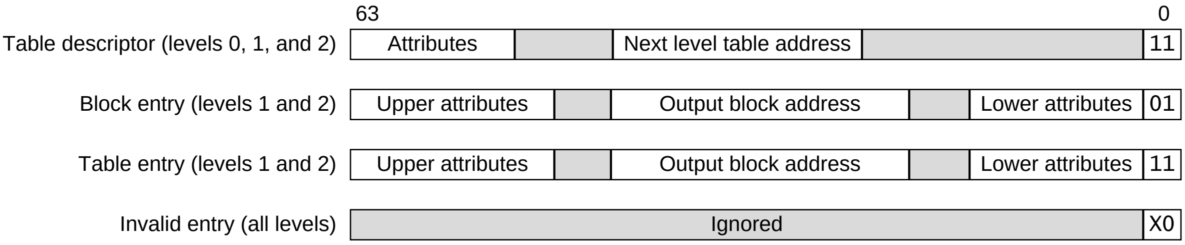
The supported granule sizes are 4KB, 16KB, and 64KB, and it is IMPLEMENTATION DEFINED which of the three are supported. Code that creates translation tables is able to read the Memory Model Feature Register 0 system register (ID_AA64MMFR0_EL1), to find out which are the supported sizes. The size is configurable for each translation table within the Translation Control Register (TCR_EL1).

4.1 AArch64 descriptor format

The AArch64 descriptor format is used in all levels of table, from Level 0 to Level 3. Level 0 descriptors can only output the address of a Level 1 table. Level 3 descriptors cannot point to another table and can only output block addresses. The format of the table is therefore slightly different for Level 3.

The following figure shows that the table descriptor type is identified by bits 1:0 of the entry and can refer to either:

- The address of a next level table, in which case memory can be further subdivided into smaller blocks.
- The address of a variable sized block of memory.
- Table entries, which can be marked Fault, or Invalid.



Note

For clarity, this diagram does not specify the width of bit fields.

4.2 Effect of granule sizes on translation tables

The three different granule sizes can affect the number and size of translation tables required.

Note

- In all cases, the first level of table is omitted if the VA input range is restricted to 39 bits.
- Depending on the size of the possible VA range, there can be even fewer levels. With a 4KB granule, for example, if the TTBCR is set so that low addresses span only 1GB, then levels 0 and 1 are not required and the translation starts at level 2, going down to level 3 for 4KB pages.

4KB

In the case of a 4kB granule, the hardware can use a 4-level look up process. The 48-bit address has nine address bits for each level translated (that is, 512 entries each), with the final 12 bits selecting a byte within the 4kB coming directly from the original address.

Bits [47:39] of the virtual address index into the 512 entry L0 table. Each of these table entries spans a 512GB range and points to an L1 table. Within that 512 entry L1 table, bits [38:30] are used as index to select an entry and each entry points to either a 1GB block or an L2 table.

Bits [29:21] index into a 512 entry L2 table and each entry points to a 2MB block or next table level. At the last level, bits [20:12] index into a 512 entry L2 table and each entry points to a 4kB block.

VA bits [47:39]	VA bits [38:30]	VA bits [29:21]	VA bits [20:12]	VA bits [11:0]
Level 0 Table Index Each entry contains: Pointer to L1 table (No block entry)	Level 1 Table Index Each entry contains: Pointer to L2 table Base address of 1GB block (IPA)	Level 2 Table Index Each entry contains: Pointer to L3 table Base address of 2MB block (IPA)	Level 3 Table Index Each entry contains: Base address off 4KB block (IPA)	Block offset and PA [11:0]

16KB

In the case of a 16kB granule, the hardware can use a 4-level look up process.

The 48-bit address has 11 address bits per level that is translated, that is 2048 entries each, with the final 14 bits selecting a byte within the 4kB coming directly from the original address.

The level 0 table contains only two entries. Bit [47] of the virtual address selects a descriptor from the two entry L0 table. Each of these table entries spans a 128TB range and points to an L1 table. Within that 2048 entry L1 table, bits [46:36] are used as an index to select an entry and each entry points to an L2 table. Bits [35:25] index into a 2048 entry L2 table and each entry points to a 32MB block or next table level.

At the final translation stage, bits [24:14] index into a 2048 entry L2 table and each entry points to a 16kB block.

VA bit [47]	VA bits [46:36]	VA bits [35:25]	VA bits [24:14]	VA bits [13:0]
Level 0 Table Index Each entry contains: Pointer to L1 table (No block entry)	Level 1 Table Index Each entry contains: Pointer to L2 table	Level 2 Table Index Each entry contains: Pointer to L3 table Base address of 32MB block (IPA)	Level 3 Table Index Each entry contains: Base address off 16KB block (IPA)	Block offset and PA [13:0]

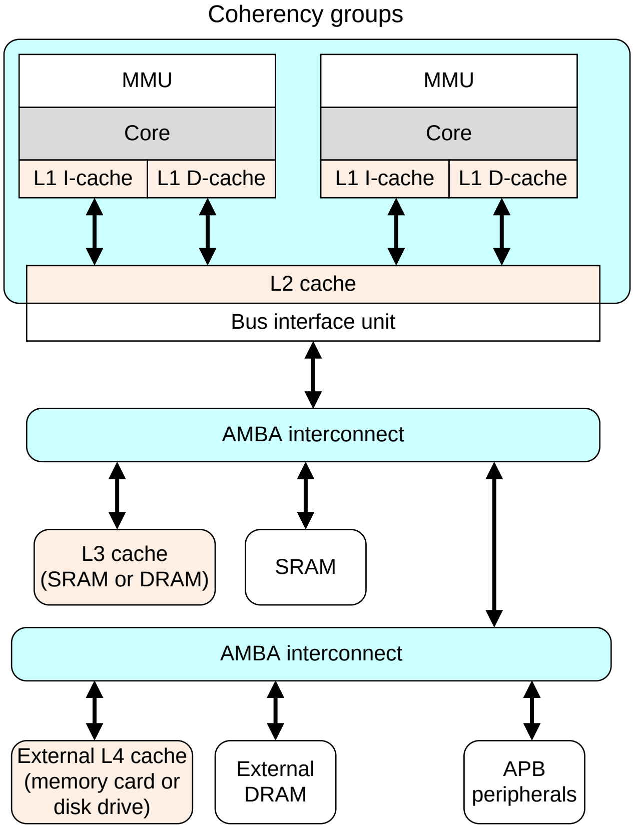
64KB

In the case of a 64kB granule, the hardware can use a 3-level look up process. The level 1 table contains only 64 entries. Bits [47:42] of the virtual address select a descriptor from the 64 entry L1 table. Each of these table entries spans a 4TB range and points to an L2 table. Within that 8192 entry L2 table, bits [41:29] are used as index to select an entry and each entry points to either a 512MB block or an L2 table. At the final translation stage, bits [28:16] index into an 8192 entry L3 table and each entry points to a 64kB block.

VA bit [47:42]	VA bits [41:29]	VA bits [28:16]	VA bits [15:0]
Level 1 Table Index Each entry contains: Pointer to L2 table (No block entry)	Level 2 Table Index Each entry contains: Pointer to L2 table Base address of 512MB block (IPA)	Level 3 Table Index Each entry contains: Base address of 64KB block (IPA)	Block offset and PA [15:0]

4.3 Cache configuration

The MMU uses translation tables and translation registers to control which memory locations are Cacheable. The MMU controls the cache policy, memory attributes, and access permissions, and provides Virtual to physical address translation.

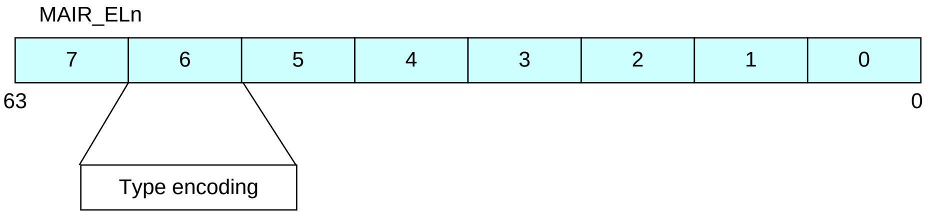


Software configuration is performed by system registers.

In some designs, the external memory system might contain further implementation-specific caches of external memories.

4.4 Cache policies

The memory type is not directly encoded in the translation table entry. Instead, each block entry specifies a 3-bit index into a table of memory types. This table is stored in the *Memory Attribute Indirection Register* MAIR_EL*n*. This table has eight entries and each of those entries has 8 bits, as shown in the following figure.



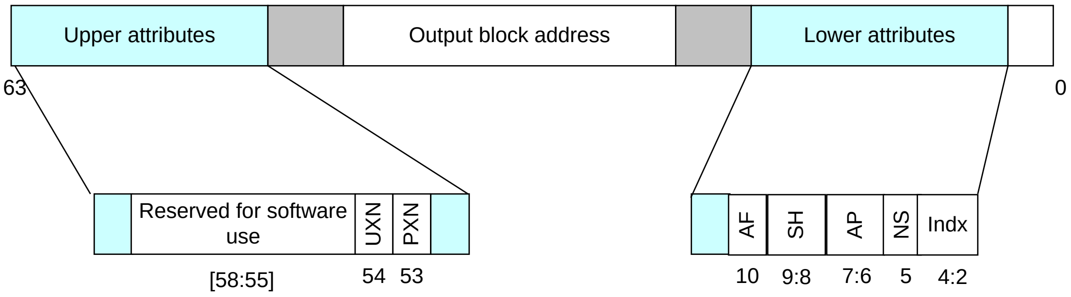
Although the translation table block entry itself does not directly contain the memory type encoding, the TLB entry inside the processor usually stores this information for a specific entry. Therefore, changes to MAIR_EL*n* might not be observed until after both an ISB instruction barrier and a TLB invalidate operation.

The MMU translation tables also define the cache policy for each block within the memory system. Memory regions that are defined as Normal might be marked as Cacheable or Non-cacheable. Bits [4:2] from the translation table entry refer to one of the eight memory attribute encodings in the

MAIR_EL*n*. The memory attribute encodings then specify the cache policies to use when accessing that memory. These are hints to the processor and it is IMPLEMENTATION DEFINED whether all cache policies are supported in a particular implementation and which cache data is regarded as coherent. A memory region can be defined in terms of its *shareability* property.

4.5 Memory attributes

The following figure shows how memory attributes are specified in a stage 1 block entry. The block entry in the translation table defines the attributes for each memory region. Stage 2 entries have a different layout.



- Unprivileged eXecute Never (UXN) and Privileged eXecute Never (PXN) are execution permissions.
- AF is the access flag.
- SH is the shareable attribute.
- AP is the access permission.
- NS is the security bit, but only at EL3 and Secure EL1.
- Indx is the index into the MAIR_EL*n*.

  Note

  For clarity, not all bits are shown in the figure.

The descriptor format supports hierarchical attributes, so that an attribute set at one level can be inherited by lower levels. It means that a table entry in an L0, L1, or L2 table can override one or more attributes that are specified in the table that it points to. This can be used for access permissions, security, and execution permissions. For example, an entry in the L1 table that has NSTable = 1 means that the NS bits in the L2 and L3 tables that it points to are ignored and all the entries are treated as having NS = 1. This feature only restricts subsequent levels of look-up for the same stage of translation.

5 Translation table configuration

In addition to storing individual translations within the TLB, the MMU can be configured to store translation tables in Cacheable memory. This usually provides much faster access to tables than always reading from external memory. TCR_EL1 has fields that control this. These fields specify the cacheability and shareability of translation tables for TTBR0 and TTBR1. The relevant fields are called SH0/1 Shareability, IRGN0/1 Inner Cacheable, and ORGN0/1 Outer Cacheable.

The following table shows the permitted settings for cacheability.

IRGN/ORGN bits for TTBR0/TTBR1	Cacheable Property
00	Normal memory, Inner Non-cacheable
01	Normal memory, Inner Write-Back Write-Allocate Cacheable
10	Normal memory, Inner Write-Through Cacheable
11	Normal memory, Inner Write-Back no Write-Allocate Cacheable

The corresponding table for shareability of memory is associated with translation table walks. For a device region, the value is ignored.

SH0 bits[13:12]	Shareability
00	Non-shareable
01	UNPREDICTABLE
10	Outer Shareable
11	Inner Shareable

The attributes that are specified in the TCR_EL1 must be the same as those specified for the virtual memory region in which the translation tables are stored. Caching the translation tables is the normal default behavior.

5.1 Virtual address tagging

The *Translation Control Register* (TCR_ELn) has an extra field that is called *Top Byte Ignore* (TBI) that provides tagged addressing support. The most significant 16 bits of an address in a 64-bit general-purpose register must be 0xFFFF or 0x0000. Any attempt to use a different bit value triggers a fault.

When tagged addressing support is enabled, the top eight bits [63:56] of the virtual address are ignored by the processor. It internally sets bit [55] to sign-extend the address to 64-bit format. The top 8 bits can then be used to pass data. These bits are ignored for addressing and translation faults. The TCR_EL1 has separate enable bits for EL0 and EL1.

One example use case might be in support of object-oriented programming languages. In addition to having a pointer to an object, you might have to keep a reference count of the number of references or pointers or handles that refer to the object, for example, so that automatic garbage collection code can de-allocate objects that are no longer referenced. This reference count can be stored as part of the tagged address, rather than in a separate table, speeding up the process of creating or destroying objects.

6 Multiple Address Spaces

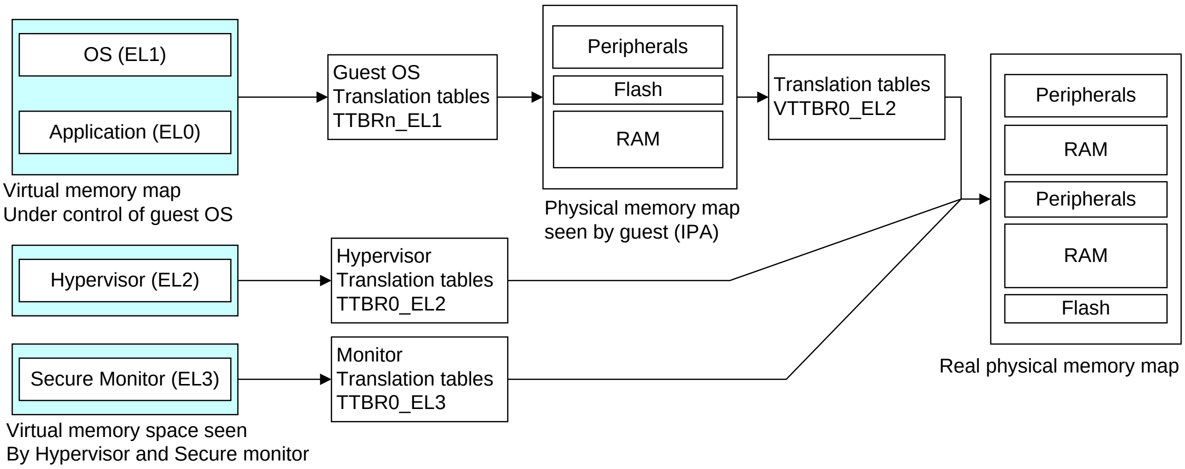
6.1 Two Stage Translations

Armv8-A virtualization introduces a second stage of translation. When a hypervisor is present in the system, there can be one or more guest operating systems present. These use TTBRn_EL1 as previously described and MMU operation appears unchanged.

In a two-stage process, the hypervisor must perform some extra translation steps to share the physical memory system between the different guest operating systems. In the first stage, the VA is translated to an *Intermediate Physical Address* (IPA). This is usually under OS control. A second stage, which is controlled by the hypervisor, translates the IPA to the final physical address (PA).

The following figure summarizes this two stage translation process.

The hypervisor and Secure monitor also have their set of stage 1 translation tables for their own code and data, which perform mapping directly from VA to PA.



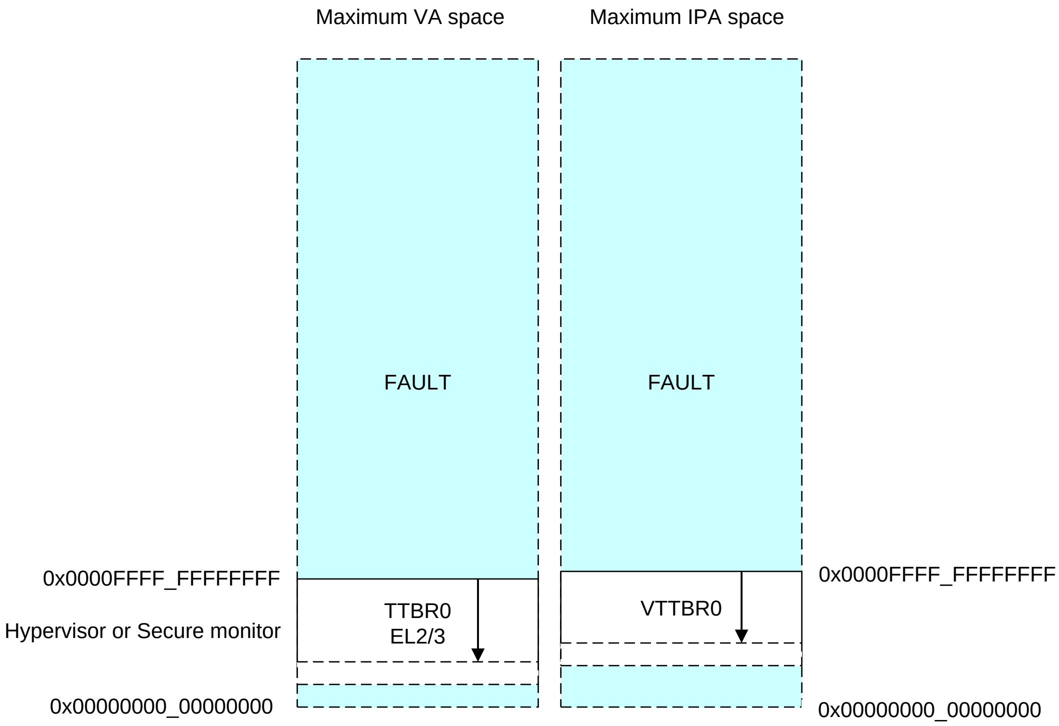
Stage 2 translations, which convert an intermediate physical address into a physical address, use an extra set of tables under control of the hypervisor. For Non-secure EL1/0 accesses, these must be explicitly enabled by writing to the *Hypervisor Configuration Register* HCR_EL2.

The base address of the Stage 2 translation table is specified in the Virtualization Translation Table Base Register (VTTBR0_EL2). It specifies a single contiguous address space at the bottom of memory. The size of the supported address space is specified in the T0SZ[5:0] field of the *Virtualization Translation Control Register*, VTCR_EL2.

31	30 19	18 16	15 14	13 12	11	9 8	7	5 0
		PS	TG0	SH0			SL0	T0SZ

The TG0 field of VTCR_EL2 specifies the granule size while the SL0 field controls the first-level of table lookup. Any access outside the defined address range causes a translation fault.

6.2 EL2 and EL3 Address Spaces



The hypervisor EL2 and Secure monitor EL3 have their own level 1 tables, which map directly from virtual to physical address space. The table base address is specified in TTBR0_EL2 and TTBR0_EL3 respectively, enabling a single contiguous address space of variable size at the bottom of memory. The TG field specifies the granule size and the SL0 field controls the first level of table lookup. Any access outside the defined address range causes a translation fault.

The Secure monitor EL3 also has its own dedicated translation tables. The table base address is specified in TTBR0_EL3 and configured via TCR_EL3. Translation tables are capable of accessing both Secure and Non-secure physical addresses. TTBR0_EL3 is used only in Secure monitor EL3 mode, not by the trusted kernel itself.

When the transition to Secure world has completed, the trusted kernel uses the EL1 translations, that is, the translation tables pointed to by TTBR0_EL1 and TTBR1_EL1. As these registers are not banked in AArch64, Secure monitor code must configure new tables for the Secure world and save and restore copies of TTBR0_EL1 or TTBR1_EL1.

The EL1 translation regime behaves differently in Secure state, compared to its normal operation in Non-secure state. The second stage of translation is disabled and the EL1 translation regime now points to both Secure and Non-secure physical addresses. There is no virtualization in the Secure world, so the IPA is always the same as the final PA.

Entries in the TLB are tagged as Secure or Non-secure, so that no TLB maintenance is ever required when moving between Secure and Normal worlds.

7 Access permissions

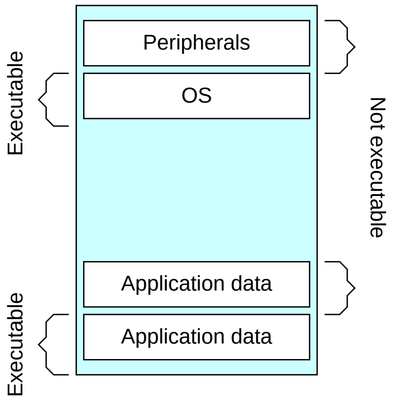
Access permissions are controlled through translation table entries. Access permissions control whether a region is readable or writeable, or both, and can be set separately to EL0 for unprivileged and access to EL1, EL2, and EL3 for privileged accesses, as shown in the following table.

AP	Unprivileged (EL0)	Privileged (EL1/2/3)
00	No access	Read and write
01	Read and write	Read and write
10	No access	Read-only
11	Read-only	Read-only

The operating system kernel, as normal, runs in EL1. The OS defines the translation table mappings, which are used by the kernel itself and by the applications that run at EL0. Some distinction between unprivileged and privileged access permissions is required as the kernel specifies different permissions for its own code and for applications.

The hypervisor, which runs at EL2, and the EL3 Secure monitor only have translation schemes for their own use and there is no need for a split in permissions between privileged and unprivileged.

Another kind of access permission is the executable attribute. Blocks can be marked as *executable* or *non-executable* (*Execute Never* (XN)). The *Unprivileged Execute Never* (UXN) and *Privileged Execute Never* (PXN) attributes can be set separately. This is used to prevent, for example, application code running with kernel privilege, or attempts to execute kernel code while in an unprivileged state. Setting these attributes prevents the processor from performing speculative instruction fetches to the memory location and ensures that speculative instruction fetches do not accidentally access locations that might be perturbed by such an access, for example, a *First in, First out* (FIFO) page replacement queue. As a result, device regions must always be marked as XN.



You can configure the processor to treat writeable regions as Execute Never, using the following bits in the SCTLR registers:

- SCTLR_EL1.WXN. Regions writeable at EL0 are treated as XN at EL0 and EL1. Regions writeable at EL1 are treated as XN at EL1.
- SCTLR_EL2 and 3.WXN. Regions writeable at ELn are treated as XN at ELn.

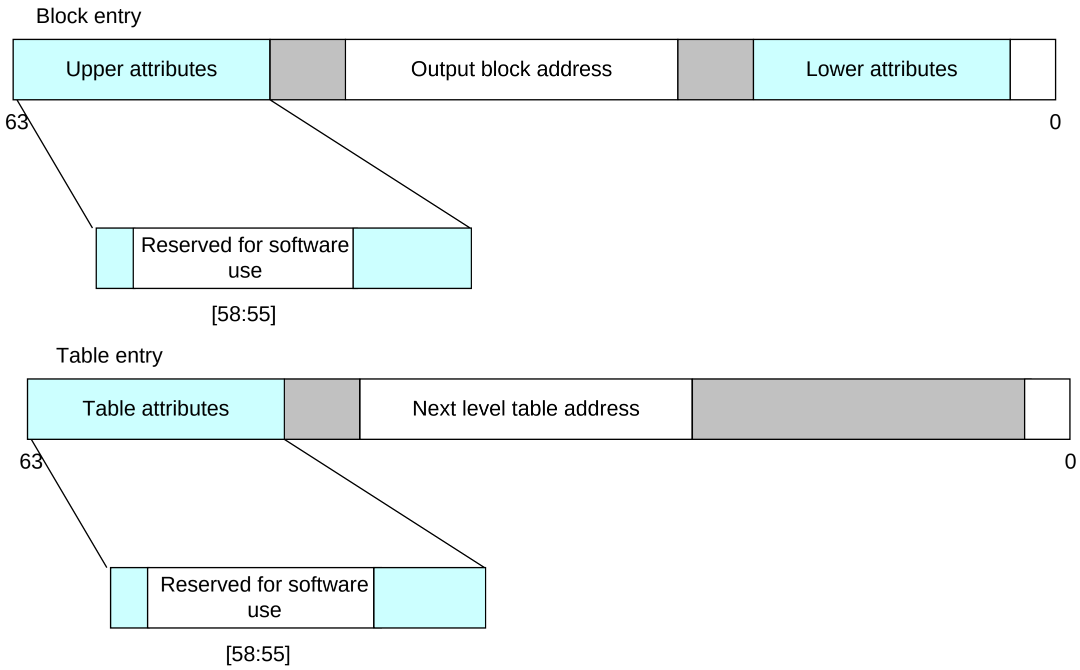
- SCTLR.UWXN. Regions writeable at EL0 are treated as XN at EL1. This is for AArch32 only.

The SCTLR_ELn bits can be cached in a TLB entry. Changing the bit in the SCTLR might not affect entries already in the TLBs. When modifying these bits, a TLB invalidate and ISB sequence is necessary.

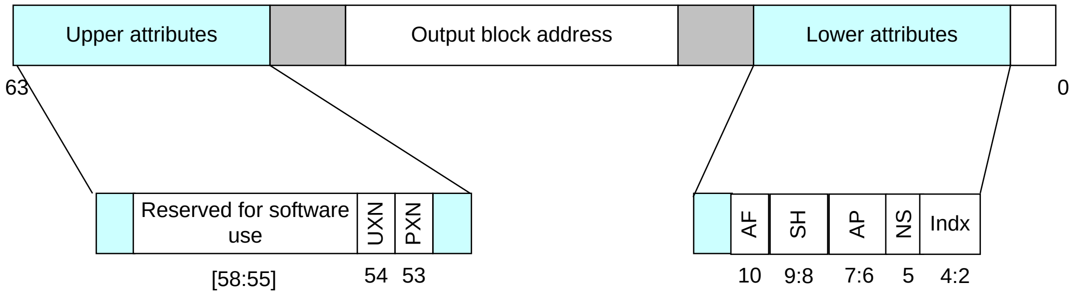
8 OS use of translation table descriptors

Operating systems use an access flag bit to keep track of which pages are being used. Software manages the flag. When the page is first created, its entry has AF set to 0. The first time the page is accessed by code, if it has AF at 0, this triggers an MMU fault. The Page fault handler records that this page is now being used and manually sets the AF bit in the table entry.

For example, the Linux kernel uses the [AF] bit for PTE_AF on ARM64 (the Linux kernel name for AArch64), which is used to check whether a page has ever been accessed. This influences some of the kernel memory management choices. For example, when a page must be swapped out of memory, it is less likely to swap out pages that are being actively used.



The following figure shows how memory attributes are specified in a stage 1 block entry.



A memory attribute bit in the descriptor, the *Access Flag* (AF), indicates when a block entry is used for the first time.

- AF = 0: This block entry has not yet been used.
- AF = 1: This block entry has been used.

Bits [58:55] of the descriptor are marked as reserved for Software Use and can be used to record OS-specific information in the translation tables. For example, the Linux kernel uses one of these bits to mark an entry as clean or dirty. The dirty status records whether the page has been written to. If the page is later swapped out of memory, a clean page can simply be discarded, but a dirty page must have its contents saved first.

9 Security and the MMU

In Non-secure state, the NS bits and NSTable bits in translation tables are ignored. Only Non-secure memory can be accessed. In Secure state, the NS bits and NSTable bits control whether a virtual address translates to a Secure or Non-secure physical address. You can use SCR_EL3.CIF to prevent the Secure world from executing from any virtual address that translates to a Non-secure physical address. Also, when in the Secure world, you can use the SCR.CIF bit to control whether Secure instruction fetches can be made to Non-secure physical memory.

9.1 Kernel access with user permissions

The LDTR or STTR instructions allow code executing at EL1 (for example, an OS) to perform memory accesses with EL0 or application permissions. This can be used, for example, to de-reference pointers that are provided with system calls, and enable the OS to check that only data accessible to the application is accessed. When executed at EL1, these instructions perform the load or store as if executed at EL0. At all other Exception levels, `LDTR`, and `STTR` behave like regular `LDR` or `STR` instructions. These are the usual size and have the same signed and unsigned variants as normal load and store instructions, but with smaller offset and restricted indexing options.

10 The Translation Lookaside Buffer

The *Translation Lookaside Buffer* (TLB) is a cache of recently accessed page translations in the MMU. For each memory access performed by the processor, the MMU checks whether the translation is cached in the TLB. If the requested address translation causes a hit within the TLB, the translation of the address is immediately available.

Each TLB entry typically contains not only physical and virtual addresses, but also attributes such as memory type, cache policies, access permissions, the *Address Space ID* (ASID), and the *Virtual Machine ID* (VMID). If the TLB does not contain a valid translation for the virtual address that is issued by the processor, which is known as a TLB miss, an external translation table walk or lookup is performed. Dedicated hardware within the MMU enables it to read the translation tables in memory. The newly loaded translation can then be cached in the TLB for possible reuse if the translation table walk does not result in a page fault. The exact structure of the TLB differs between implementations of the Arm processors.

If the OS modifies translation entries that have been cached in the TLB, it is the responsibility of the OS to invalidate these stale TLB entries.

When executing A64 code, there is a TLBI, which is a TLB invalidate instruction. It has the form:

```
TLBI <type><level>{IS} {, <Xt>}
```

The following list gives some of the more common selections for the type field.

ALL	All TLB entries.
VMALL	All TLB entries. This is stage 1 for current guest OS.
VMALLS12	All TLB entries. This is stage 1 and 2 for current guest OS.
ASID	Entries that match ASID in Xt.
VA	Entry for virtual address and ASID specified in Xt.
VAA	Entries for virtual address that is specified in Xt, with any ASID.

Each Exception level, that is EL3, EL2, or EL1, has its own virtual address space that the operation applies to. The IS field specifies that this is only for Inner Shareable entries.

The `<level>` field simply specifies the Exception level virtual address space (can be 3, 2 or 1) that the operation must apply to.

The `IS` field specifies that this is only for Inner Shareable entries.

The following table lists TLB configuration instructions:

TLB invalidate	Variant	Description
TLBI	ALLEn	TLB invalidate All, ELn.
	ALLEnIS	TLB invalidate All, ELn, Inner Shareable.
	ASIDE1	TLB invalidate by ASID, EL1.
	ASIDE1IS	TLB invalidate by ASID, EL1, Inner Shareable.
	IPAS2E1	TLB invalidate by IPA, Stage 2, EL1.

IPAS2E1IS	TLB invalidate by IPA, Stage 2, EL1, Inner Shareable.
IPAS2LE1IS	TLB invalidate by IPA, Stage 2, Last level, EL1, Inner Shareable.
VAAE1	TLB invalidate by VA, All ASID, EL1.
VAAE1IS	TLB invalidate by VA, All ASID, EL1, Inner Shareable.
VAALE1IS	TLB invalidate for the Last level, by VA, All ASID, EL1, Inner Shareable.
VAEn	TLB invalidate by VA, ELn.
VAEnIS	TLB invalidate by VA, ELn, Inner Shareable.
VALEn	TLB invalidate by VA, Last level, ELn.
VALEnIS	TLB invalidate by VA, Last level, ELn, Inner Shareable.
VMALLE1	TLB invalidate by VMID, All at stage 1, EL1.
VMALLE1IS	TLB invalidate by VMID, EL1, Inner Shareable.
VMALLS12E1	TLB invalidate by VMID, All at Stage 1 and 2, EL1.
VMALLS12E1	TLB invalidate by VMID, All at Stage 1 and 2, EL1.
VMALLS12E1IS	TLB invalidate by VMID, All at Stage 1 and 2, EL1 Inner Shareable.
VMALLS12E1IS	TLB invalidate by VMID, All at Stage 1 and 2, EL1 Inner Shareable.

The following code example shows a sequence for writes to translation tables backed by Inner Shareable memory:

```
<< Writes to translation tables >>
DSB ISHST                    // ensure write has completed
TLBI ALLE1                   // invalidate all TLB entries
DSB ISH                      // ensure completion of TLB invalidation
ISB                          // synchronize context and ensure that no
                             // instructions are fetched using the old
                             // translation
```

For a change to a single entry, for example, use the instruction:

```
TLBI VAE1, X0
```

Which invalidates an entry that is associated with the address that is specified in the register X0.

The TLB can hold a fixed number of entries. You can achieve best performance by minimizing the number of external memory accesses caused by translation table traversal and obtaining a high TLB hit rate. The Armv8-A architecture provides a feature known as contiguous block entries to efficiently use TLB space. Translation table block entries each contain a contiguous bit. When set, this bit signals to the TLB that it can cache a single entry covering translations for multiple blocks. A lookup can index anywhere into an address range covered by a contiguous block. The TLB can therefore cache one entry for a defined range of addresses, making it possible to store a larger range of virtual addresses within the TLB than is otherwise possible.

To use a contiguous bit, the contiguous blocks must be adjacent, that is they must correspond to a contiguous range of virtual addresses. They must start on an aligned boundary, have consistent attributes, and point to a contiguous output address range at the same level of translation. The required alignment is that VA[20:16] for a 4KB granule or VA[28:21] for a 64KB granule, are the same for all addresses. The following numbers of contiguous blocks are required:

- 16 × 4KB adjacent blocks giving a 64KB entry with 4KB granule.
- 32 ×32MB adjacent blocks giving a 1GB entry for L2 descriptors.
- 128 ×16KB giving a 2MB entry for L3 descriptors when using a 16KB granule.
- 32 ×64Kb adjacent blocks giving a 2MB entry with a 64KB granule.

If these conditions are not met, a programming error occurs, which can cause TLB aborts or corrupted lookups. Possible examples of such an error include:

- One or more of the table entries do not have the contiguous bit set.
- The output of one of the entries points outside the aligned range.

With the Armv8-A architecture, incorrect use does not allow permissions checks outside of EL0 and EL1 valid address space to be escaped, or to erroneously provide access to EL3 space.

11 Context switching

Processors that implement the Armv8-A architecture are typically used in systems running a complex operating system with many applications or tasks that run concurrently. When an application starts, the operating system allocates it a set of translation table entries that map both the code and data that is used by the application to physical memory. Each application therefore has its own unique translation tables residing in physical memory. These tables can be modified later by the kernel, for example, to map in extra memory space, and are removed when the application is no longer running.

Normally, it is likely that there are multiple tasks present in the memory system. The kernel scheduler periodically transfers execution from one task to another. This is called a context switch and requires the kernel to save all Execution state that is associated with the process and to restore the state of the process to be run next. The kernel also switches translation table entries to those of the next process to be run. The memory of the tasks that are not currently running is protected from the task that is running.

Exactly what has to be saved and restored varies between different operating systems, but typically a process context switch includes saving or restoring some or all of the following elements:

- General-purpose registers (X0 - X30).
- Advanced SIMD and floating-point registers (V0 - V31).
- Some status registers.
- TTBR0_EL1 and TTBR0.
- Thread Process ID (TPIDxxx) Registers.
- Address Space ID (ASID).

For EL0 and EL1, there are two translation tables. TTBR0_EL1 provides translations for the bottom of the virtual address space, which is typically application space and TTBR1_EL1 covers the top of the virtual address space, typically kernel space. This split means that the OS mappings do not have to be replicated in the translation tables of each task.

Translation table entries contain a non-global (nG) bit. If the nG bit is set for a particular page, it is associated with a specific task or application. If the bit is marked as 0, then the entry is global and applies to all tasks.

For non-global entries, when the TLB is updated and the entry is marked as non-global, a value is stored in the TLB entry in addition to the normal translation information. This value is called the Address Space ID, which is a number that is assigned by the OS to each individual task.

Subsequent TLB look-ups only match on that entry if the current ASID matches with the ASID stored in the entry. This enables multiple valid TLB entries to be present for a particular page that is marked as non-global, but with different ASID values. In other words, it is not necessary to clean or invalidate the TLBs when context switching takes place.

This ASID value can be specified as either an 8-bit or 16-bit value, which is controlled by the TCR_EL1.AS bit. The current ASID value is specified in either TTBR0_EL1 or TTBR1_EL1. TCR_EL1 controls which TTBR holds the ASID, but typically, it is TTBR0_EL1, as this corresponds to application space.

Note

Having the current value of the ASID stored in the translation table register means that both the translation tables and the ASID can be atomically modified in a single instruction. This simplifies the process of changing the table and ASID when compared with the Armv7-A architecture.

Additionally, the Armv8-A architecture provides Thread ID registers for use by operating system software. These have no hardware significance and are typically used by threading libraries as a base pointer to per-thread data. This is often referred to as Thread Local Storage (TLS). For example, the Pthreads library uses this feature and includes the following registers:

- *User Read and Write Thread ID Register* (TPIDR_EL0).
- *User Read-Only Thread ID Register* (TPIDRRO_EL0).
- *Thread ID Register, privileged accesses only* (TPIDR_EL1).

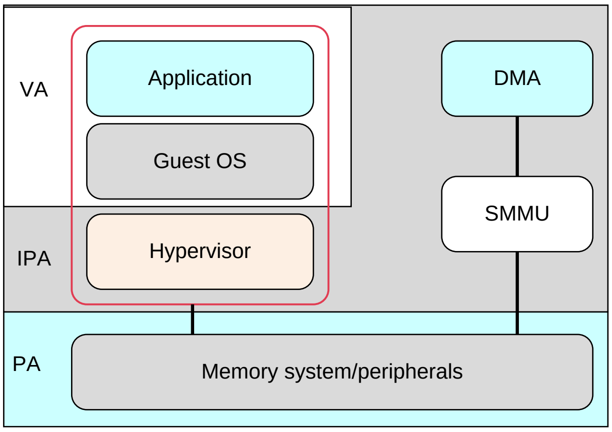
12 System MMU

Devices like a DMA or GPU in a compute system see the physical address space, so when they are programmed, you must use the PA to specify the source and destination addresses for a DMA, or the frame buffer location for a GPU. This is normally handled by kernel level code, which makes calls into the kernel to get the VA to PA mappings.

When second stage translation is added, the kernel no longer sees 'real' physical addresses. It sees IPAs instead. This means that if a pointer is passed from the kernel module to the GPU or DMA it could be the wrong address.

One solution to this is for a hypervisor to intercept all communication between the OS and the device, with the Hypervisor translating passed addresses from the IPA to the PA. This approach is potentially expensive, as it would mean that you had to take an exception (to enter the Hypervisor) every time you wrote to one of the memory mapped registers of the device.

The alternative approach is to make the device see the same IPA space as the kernel, which is where the *System MMU* (SMMU) comes in.



The SMMU is effectively an external copy of the MMU inside the processor. It can be placed in your system between a device (like a DMA or GPU) and the interconnect. Any transaction passing through the SMMU can then be translated, meaning that the DMA or GPU sees a translated address space.

The SMMU architecture uses the same translation table formats as Armv7-A and Armv8-A. So an SMMU is typically pointed at the same set of tables in memory as the processor is using. This means that the DMA or GPU can have the same view of memory as the guest OS, removing the need for the costly trapping by the Hypervisor in software. The SMMU architecture allows for designs that do stage 1 translation (VA to IPA), stage 2 (IPA to PA) or both (VA to IPA to PA). Not all implementations support all these options.

Second stage translation only gives the DMA or GPU the same view of memory as the guest OS, so that the same drive code can be used in a virtualized or non-virtualized system.