ARMv8 Translation Table System and Memory Model

ARM Architecture Group

Document number:	**PRD03-GENC-009612 42.0**
Date of Issue:	2 May 2014
Author:	
Authorised by:	

Abstract

Keywords

Distribution list

Name	Function	Name	Function

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

Beta Release

1.1.2 Change history

Change history managed under Domino

1.2 References

This document refers to the following documents.

Ref	Doc No	Author(s)	Title
[1]	PRD03-GENC-009432	R.Grisenthwaite	ARMv8 Exception Model
[2]	PRD03-PRDC-010486	M Williams	ARMv8 Debug Architecture
[3]	ARM DDI 0406 revC	ARM	ARM Architecture Reference Manual

1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning

2 INTRODUCTION

This document describes out how the concepts introduced in the Large Physical Address Extensions (LPAE) extend to define the translation system for the ARMv8 architecture when using AArch64. It is presented for review, and includes a number of notes to help explain the rationale. In addition, more detailed pieces of rationale and particular review points have been called out in the document.

As many of the concepts of the ARMv8 architecture are shared with the ARMv7 architecture, this document does not attempt to reproduce the details of those concepts. As a result, this document assumes a significant level of understanding of the ARMv7 architecture as extended by the Virtualization Extensions and Large Physical Address Extensions, and refers to it extensively.

3 OVERVIEW

3.1 Introduction

The translation system described in the Large Physical Address Extensions, defined in the ARM ARM revC[3] for the ARMv7 architecture was designed to be broadly applicable to the ARMv8 architecture when using AArch64, and as such the ARMv8 Translation System is presented as series of changes to the LPAE, rather than as a separate document.

When EL1 is using AArch32, the stage 1 translation systems described in the ARMv7 architecture, as extended by the Virtualization Extensions and Large Physical Address Extensions apply for stage 1 translations made at EL0 or EL1, and either the VMSAv7 or LPAE translation table format can be selected for the stage 1 translation.

When EL1 is using AArch64, the translation system described in this document apply for stage 1 translations made at EL0 or EL1

When EL2 is using AArch32:

- the translation system described in the ARMv7 architecture, as extended by the Virtualization Extensions and Large Physical Address Extensions apply for translations made at EL2.
- the translation system described in the ARMv7 architecture, as extended by the Virtualization Extensions and Large Physical Address Extensions apply for stage2 translations made at EL1 and EL0

When EL2 is using AArch64, the translation system described in this document apply for translations made at EL2 and for second stage translations made at EL1 and EL0.

When EL3 is using AArch32, the translation systems described in the ARMv7 architecture, as extended by the Virtualization Extensions and Large Physical Address Extensions apply for translations made at EL3 and either the VMSAv7 or LPAE translation table format can be selected.

When EL3 is using AArch64, the translation system described in this document apply for translations made at EL3.

In all cases, translations made at an exception level include (where applicable) translations caused by :

- Explicit memory accesses and cache maintenance operations in instructions executed at that exceptions level
- Stage 1 translation table walks caused with translations made at EL0 or EL1, which are subject to stage 2 translations
- Instruction fetches at that exception level
- Address Translation instructions using that exception level

The ARMv8 architecture introduces support for a large physical address size than is described in the ARMv7 architecture, as extended by the Virtualization Extensions and Large Physical Address Extensions, though it can only be accessed when using AArch64. This does have an impact on the translation system, as described in section 3.7.2

3.2 Memory translation granule size in AArch64 state

The ARMv8 architecture, when using AArch64, makes the size of translation tables and of the block size at the lowest level translation table the same, and this size is referred to as the *memory translation granule*.

The ARMv8 architecture supports three different memory translation granule sizes when executing in AArch64:

- 4KBytes – this is consistent with the translation table walking system described in the Large Physical Address Extensions
- 16Kbytes – this provides a medium sized granule that can be used to lead to fewer levels of translation tables to translate a particular sized virtual address while reducing memory wastage.
- 64KBytes – this provides a larger granule that can be used to lead to fewer levels of translation tables to translate a particular sized virtual address.

The Translation Granule size can be set independently for each translation table pointed to by a translation table base register, as defined by bits in the translation control register. It can also be set independently for stage 1 translations and for stage 2 translations.

All three of these translation granules are architecturally optional. Implementations must follow the requirements of the operating system and hypervisor vendors that the implementation is to use as to which sizes are required, and in the absence of any guidance to the contrary must implement at least 4Kbyte and 64Kbyte translation granules.

Note: ARM recommends that memory mapped peripherals are separated by an integer multiple of the largest granule size supported by the operating system or hypervisor to allow the peripherals to be independently managed.

3.2.1 Effects on 4KByte Granule on the Translation Tables

The basic translation table walking system described in the Large Physical Address Extensions is modified in the following ways to support a larger input address

- For stage 1 translations, a level 0 table is supported for an input address of greater than 39 bits. The format of the level 0 table is the same as the format of the level 1 table described in the Large Physical Address Extensions, except that a block translation is not supported – programming Bit[1] ==0 in the translation table descriptor gives a Translation Fault.
- For stage 2 translations, a level 0 table is supported for an input address of greater than 39 bits, though the starting level is configured in the VTCR_EL2.SL0 field. The VTCR_EL2.SL0 field is encoded as follows:

SL0	**Starting Level for a 4KByte granule**
0	Level 2
1	Level 1
2	Level 0
3	RESERVED (see section 3.7.1)

- The T*SZ fields in the various translation control registers increase to be 6 bit registers, holding an unsigned number. The size of input address supported is encoded in AArch64 as $2^{64-T*SZ}$.

Therefore the translation tables using the 4 KByte Granule have the following form:

Level of Table	Input Address Bits used for Index for a full table	Size of Block Entry	Permitted T*SZ values for starting at this level (see note1)			
			Stage 1 table		Stage 2 table	
			Min T*SZ	Max T*SZ	Min T*SZ	Max T*SZ
0	47:39 (see note2)	N/A	16	24	16	24
1	38:30 (see note2)	1 GBytes	25	33	21	33
2	29:21 (see note2)	2 MBytes	34	39	30	39
3	20:12	4 KBytes	NA		NA	

Note1: for stage 2 translation, the actual starting level is defined by the SL0 field

Note2: for stage 2 translations, up to 16 tables can be concatenated at the starting level, which means that up to an additional 4 bits (above the highest size bit indicated in the table) can be used to index the concatenated tables.

For stage1 translation if T*SZ is smaller than 16, then an implementation can do one of the following things:

- T*SZ behaves as if it is 16 (other than for a read-back of the value held in the register).
- Use of the T*SZ will cause a Stage1 Level 0 translation Fault

For stage1 translation if T*SZ is larger than 39, then an implementation can do one of the following things:

- T*SZ it behaves as if it is 39 (other than for a read-back of the value held in the register) .
- Use of the T*SZ will cause a Stage1 Level 0 translation Fault

For stage2 translation if T0SZ is larger than 39, then an implementation can do one of the following things:

- T0SZ behaves as if it is 39 (other than for a read-back of the value held in the register).
- Use of the T0SZ will cause a Stage2 Level 0 translation Fault

If T0SZ is smaller than the minimum size shown in section 3.7.1, then it behaves as described in section 3.7.1 .

For stage 2 translation, if the T0SZ value is not consistent with the programmed SL0 field at the time of a translation walk, a second stage level 0 translation fault is generated.

If the address range translated by a set of blocks marked as contiguous (using the contiguous bit) is larger than the size of the input address supported at a stage of translation (as defined by the T*SZ field) to translate that address at that stage of translation, then this is a programming error. An implementation is permitted, but not required, to:

- Treat such a block within a contiguous set of blocks as causing a translation fault, even though the block is valid, and the address accessed within that block is within the size of the input address supported at a stage of translation (as define by the T*SZ field)
- Treat such a block within a contiguous set of blocks as causing not causing translation fault even though the address accessed within that block is outside the size of the input address supported at a stage of translation (as define by the T*SZ field) , provided that both the following apply:
    - the block is valid

- At least one address within the block, or contiguous set of blocks, is within the size of the input address supported at a stage of translation

The following table shows the algorithm for determining the address of a translation table entry:

Table level	**Addressed by**
Level 0	BaseAddress[PAMax-1:x],IA[y:39],000 x = 12 if T{0,1}SZ < 16 x = 28-T{0,1}SZ if 16 =< T{0,1}SZ =< 24 y = x+35 For the Stage1 translation 16 =< T{0,1}SZ =< 24 For the Stage2 translation 16 =< T0SZ =< 24
Level 1	BaseAddress[PAMax-1:x],IA[y:30],000 For the Stage1 translation x = 37-T{0,1}SZ if 25=< T{0,1}SZ =< 33 x = 12 otherwise For the Stage2 translation If SL0 = 2 then x=12 If SL0 = 1 then x=37-T0SZ 21 =< T0SZ =< 33 y = x+26
Level 2	BaseAddress[PAMax-1:x],IA[y:21],000 For the Stage1 translation x = 46-T{0,1}SZ if T{0,1}SZ >= 34 x = 12 otherwise For the Stage2 translation If SL0 > 0 then x=12 If SL0 = 0 then : x=46-T0SZ 30 =< T0SZ =< 39 y = x+17
Level 3	BaseAddress[PAMax-1:12],IA[20:12],000

3.2.1.1 Identification for use of a 4Kbyte Granule

In future, it is possible that not all implementations support a 4 Kbyte granule, so the support of a 4 Kbyte Granule is identified using the field ID_AA64MMFR0_EL1, bits [31:28]:

0 – 4 KB granule supported

F – 4 KB granule not supported

All other values are RESERVED.

3.2.2 Effects on 16KByte Granule on the Translation Tables

The basic translation table walking system described in the Large Physical Address Extensions is modified in the following ways to support the 16KByte Granule:

- All pointers to translation tables found in translation table entries have bits [13:12] treated as RES0

- The maximum bit position of the least significant bit of the translation table base field at stage 1 is 14, and for stage 2 is 18
- The number of bits of the input address used to index a translation table is increased from 9 to 11 bits
- A Level 1 or Level 0 block entry is not supported - programming Bit[1] ==0 in the level 1 or level 0 translation table descriptor gives a Translation Fault
- A Level 0 table is not supported for Stage2 – to support a full 48 bit IPA, 2 contiguous level 1 tables must be used.
- For stage 2 translations, the SL0 field is encoded differently:

SL0	**Starting Level for a 16KByte granule**
0	Level 3
1	Level 2
2	Level 1
3	RESERVED (see section 3.7.1)

Therefore the translation tables using the 16 KByte Granule have the following form:

Level of Table	**Input Address Bits used for Index for a full table**	**Size of Block Entry**	**Permitted T*SZ values for starting at this level (see note1)**			
			Stage 1 table		**Stage 2 table**	
			Min T*SZ	**Max T*SZ**	**Min T*SZ**	**Max T*SZ**
0	47 (v small table)	N/A	16	16	N/A	
1	46:36 (see note2)	N/A	17	27	16	27
2	35:25 (see note2)	32 MBytes	28	38	24	38
3	24:14	16 KBytes	39	39	35	39

Note1: for stage 2 translation, the actual starting level is defined by the SL0 field

Note2: for stage 2 translations, up to 16 tables (if starting at Level2 or Level3) or 2 tables (if starting at Level1) can be concatenated at the starting level, which means that up to an additional 4 or 1 bits (above the highest size bit indicated in the table) can be used to index the concatenated tables.

For stage1 translation if T*SZ is smaller than 16, then an implementation can do one of the following things:

- T*SZ behaves as if it is 16 (other than for a read-back of the value held in the register).
- Use of the T*SZ will cause a Stage1 Level 0 translation Fault

For stage1 translation if T*SZ is larger than 39, then an implementation can do one of the following things:

- T*SZ it behaves as if it is 39 (other than for a read-back of the value held in the register) .
- Use of the T*SZ will cause a Stage1 Level 0 translation Fault

For stage2 translation if T0SZ is larger than 39, then an implementation can do one of the following things:

- T0SZ behaves as if it is 39 (other than for a read-back of the value held in the register).
- Use of the T0SZ will cause a Stage2 Level 0 translation Fault

If T0SZ is smaller than the minimum size shown in section 3.7.1, then it behaves as described in section 3.7.1 .

For stage 2 translation, if the T0SZ value is not consistent with the programmed SL0 field at the time of a translation walk, a second stage level 0 translation fault is generated.

When a 16 KByte granule is selected, the contiguousness bit applies to 128 contiguous aligned entries at level3, and 32 contiguous aligned entries at level2.

If the address range translated by a set of blocks marked as contiguous (using the contiguous bit – which was named Contiguous Hint in ARMv7) is larger than the size of the input address supported at a stage of translation (as define by the T*SZ field) to translate that address at that stage of translation, then this is a programming error. An implementation is permitted, but not required, to:

- Treat such a block within a contiguous set of blocks as causing a translation fault, even though the block is valid, and the address accessed within that block is within the size of the input address supported at a stage of translation (as define by the T*SZ field)
- Treat such a block within a contiguous set of blocks as causing not causing translation fault even though the address accessed within that block is outside the size of the input address supported at a stage of translation (as define by the T*SZ field) , provided that both the following apply:
    - the block is valid
    - At least one address within the block, or contiguous set of blocks, is within the size of the input address supported at a stage of translation

The following table shows the algorithm for determining the address of a translation table entry:

Table level	**Addressed by**
Level 0	BaseAddress[PAMax-1:4],IA[47],000 Only applies to stage 1 and only when T0SZ < 17
Level 1	BaseAddress[PAMax-1:x],IA[y:36],000 For the Stage1 translation x = 14 if T{0,1}SZ < 17 x = 31-T{0,1}SZ if 17=< T{0,1}SZ =< 27 For the Stage2 translation If SL0 = 2 then x=31-T0SZ T0SZ =< 27 y = x+32
Level 2	BaseAddress[PAMax-1:x],IA[y:25],000 For the Stage1 translation x = 42-T{0,1}SZ if 28 =<T{0,1}SZ =< 38 x = 14 otherwise For the Stage2 translation If SL0 = 2 then x=14 If SL0 = 1 then x=42-T0SZ 24 =< T0SZ =< 38 y = x+21
Level 3	For Stage1 translation: BaseAddress[PAMax-1:14],IA[24:14],000 For Stage2 translation: BaseAddress[PAMax-1:x],IA[y:14],000

	If SL0 = 0 then x = 53 – T0SZ; 35 =<T0SZ =<39 If SL0 >0 then x=14; y=x +10;

3.2.2.1 Identification for use of a 16Kbyte Granule

Not all implementations support a 16 Kbyte granule, so the support of a 16 Kbyte Granule is identified using the field ID_AA64MMFR0_EL1, bits [23:20]:

0 – 16 KB granule not supported

1 – 16 KB granule supported

All other values are RESERVED.

3.2.3 Effects on 64KByte Granule on the Translation Tables

The basic translation table walking system described in the Large Physical Address Extensions is modified in the following ways to support the 64KByte Granule:

- All pointers to translation tables and in translation table entries have bits [15:12] treated as RES0
- The maximum bit position of the least significant bit of the translation table base field at stage 1 is 16, and for stage 2 is 20
- The number of bits of the input address used to index a translation table is increased from 9 to 13 bits
- A Level 1 block entry is not supported - programming Bit[1] ==0 in the level 1 translation table descriptor gives a Translation Fault
- For stage 2 translations, the SL0 field is encoded differently:

SL0	**Starting Level for a 64KByte granule**
0	Level 3
1	Level 2
2	Level 1
3	RESERVED (see section 3.7.1)

Therefore the translation tables using the 64 KByte Granule have the following form:

Level of Table	**Input Address Bits used for Index for a full table**	**Size of Block Entry**	**Permitted T*SZ values for starting at this level (see note1)**			
			Stage 1 table		**Stage 2 table**	
			Min T*SZ	**Max T*SZ**	**Min T*SZ**	**Max T*SZ**
1	47:42 (small	N/A	16	21	16	21

	table)					
2	41:29 (see note2)	512 MBytes	22	34	18	34
3	28:16	64 KBytes	35	39	31	39

Note1: for stage 2 translation, the actual starting level is defined by the SL0 field

Note2: for stage 2 translations, up to 16 tables can be concatenated at the starting level, which means that up to an additional 4 bits (above the highest size bit indicated in the table) can be used to index the concatenated tables.

For stage1 translation if T*SZ is smaller than 16, then an implementation can do one of the following things:

- T*SZ behaves as if it is 16 (other than for a read-back of the value held in the register).
- Use of the T*SZ will cause a Stage1 Level 0 translation Fault

For stage1 translation if T*SZ is larger than 39, then an implementation can do one of the following things:

- T*SZ it behaves as if it is 39 (other than for a read-back of the value held in the register) .
- Use of the T*SZ will cause a Stage1 Level 0 translation Fault

For stage2 translation if T0SZ is larger than 39, then an implementation can do one of the following things:

- T0SZ behaves as if it is 39 (other than for a read-back of the value held in the register).
- Use of the T0SZ will cause a Stage2 Level 0 translation Fault

If T0SZ is smaller than the minimum size shown in section 3.7.1, then the behaviour is as described in section 3.7.1

For stage 2 translation, if the T0SZ value is not consistent with the programmed SL0 field at the time of a translation walk, a second stage level 0 translation fault is generated.

When a 64 KByte granule is selected, the contiguousness bit applies to 32 contiguous aligned entries, as opposed to the 16 contiguous entries that it describes for a 4 KByte granule. **Rationale:** This ensures that the lowest level contiguous region for the 64 KByte granule is 2 Mbytes, so aligning with likely TLB matching sizes.

If the address range translated by a set of blocks marked as contiguous (using the contiguous bit – which was named Contiguous Hint in ARMv7) is larger than the size of the input address supported at a stage of translation (as define by the T*SZ field) to translate that address at that stage of translation, then this is a programming error. An implementation is permitted, but not required, to:

- Treat such a block within a contiguous set of blocks as causing a translation fault, even though the block is valid, and the address accessed within that block is within the size of the input address supported at a stage of translation (as define by the T*SZ field)
- Treat such a block within a contiguous set of blocks as causing not causing translation fault even though the address accessed within that block is outside the size of the input address supported at a stage of translation (as define by the T*SZ field) , provided that both the following apply:
    - the block is valid
    - At least one address within the block, or contiguous set of blocks, is within the size of the input address supported at a stage of translation

The following table shows the algorithm for determining the address of a translation table entry:

Table level	**Addressed by**
Level 1	BaseAddress[PAMax-1:x],IA[y:42],000

	For the Stage1 translation x = 9 if T{0,1}SZ < 16 x = 25-T{0,1}SZ if 16=< T{0,1}SZ =< 21 For the Stage2 translation If SL0 = 2 then x=25-T0SZ 16 =< T0SZ =< 21 y = x+38
Level 2	BaseAddress[PAMax-1:x],IA[y:29],000 For the Stage1 translation x = 38-T{0,1}SZ if 22 =<T{0,1}SZ =< 34 x = 16 otherwise For the Stage2 translation If SL0 = 2 then x=16 If SL0 = 1 then x=38-T0SZ 18 =< T0SZ =< 34 y = x+25
Level 3	BaseAddress[PAMax-1:x],IA[y:16],000 For the Stage1 translation x = 51-T{0,1}SZ if T{0,1}SZ >= 35 x = 16 otherwise For the Stage2 translation If SL0 > 0 then x=16 If SL0 = 0 then x=51-T0SZ 31 =< T0SZ =< 39 y = x+ 12

3.2.3.1 Identification for use of a 64Kbyte Granule

In future, it is possible that not all implementations support a 64 Kbyte granule, so the support of a 64 Kbyte Granule is identified using the field ID_AA64MMFR0_EL1, bits [27:24]:

0 – 64 KB granule supported

F – 64 KB granule not supported

All other values are RESERVED.

3.3 First Stage of Translation under 64-bit Exception Level control

The first stage of translation, that from Virtual Address (VA) to Intermediate Physical Address (IPA) is altered in the ARMv8 architecture when using AArch64 when compared with the translation system described in the Large Physical Address Extensions (LPAE) defined in the ARM ARM revC[3] . This is because the input address is described as a result of a calculation involving a 64-bit register, as opposed to the 32-bit register used in the LPAE. In addition, the ability to support a16-Kbyte or a 64-Kbyte Granule is provided, as described in section 3.2.

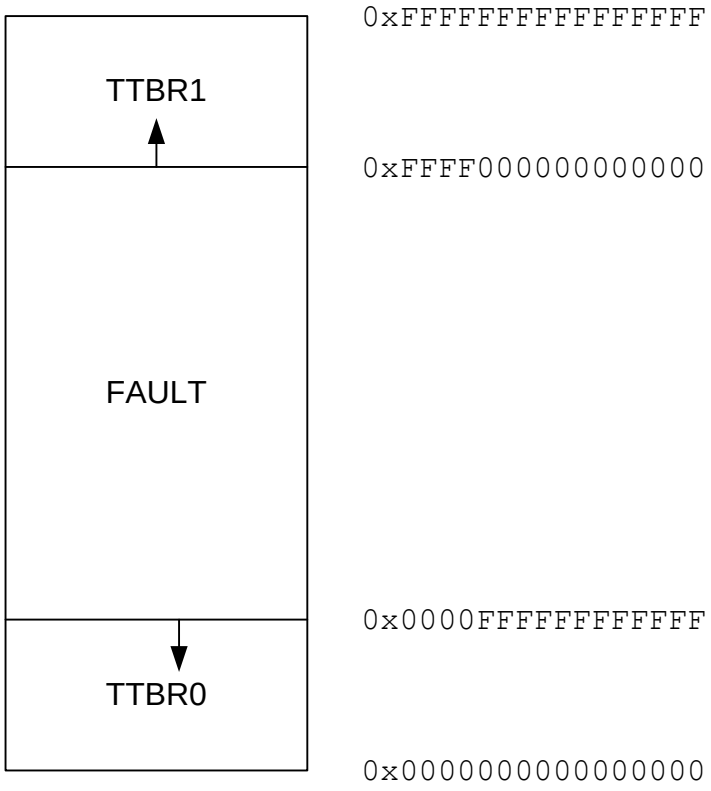
While the Virtual Address comes from a 64-bit register, only a smaller address range is supported, with each TTBR being able to translate up to 48 bits of Virtual Address. Other addresses will generate translation faults.

Note: While the input address size is programmable by software up to 48 bits in size for each TTBR, a hardware implementation is required to support all 48 bits for each TTBR.

Note: This is proposed to avoid the need to build (and validate) a system with up to 5 levels of translation table walk, so this sets an upper limit to the Virtual Address that can be translated.

The basic form of the First Stage of translation used by EL1/EL0, when EL1 is 64-bit, is as follows:

- Two different 64-bit TTBRx registers are provided: TTBR0 and TTBR1. These registers are the same as described in the LPAE (with the EAE bit set in the TTBCR – this bit becomes UNK/SBOP).
- The size of the IPA supported increases to up to 48 bits (see section 3.7.1).
- TTBR0 is selected when the upper bits of the VA are 0. TTBR1 is selected when the upper bits of the VA are 1. The number of uppers bits that are checked is a function of the TCR_ELx.TxSZ fields. The maximum address map is therefore as shown in the following diagram.



- Configuration bits are provided in the TCR_EL1 to determine whether bits[63:56] of the virtual address is considered for the address comparison for the TTBR0 region and for the TTBR1 region. These bits allow the top 8 bits to be used to hold additional information about an address that can be used by the software, provided that the software ensures that during manipulation of the address these top bits are correctly preserved (see also section 3.19)

  Note: Ordinary arithmetic on an address will tend to preserve this information.

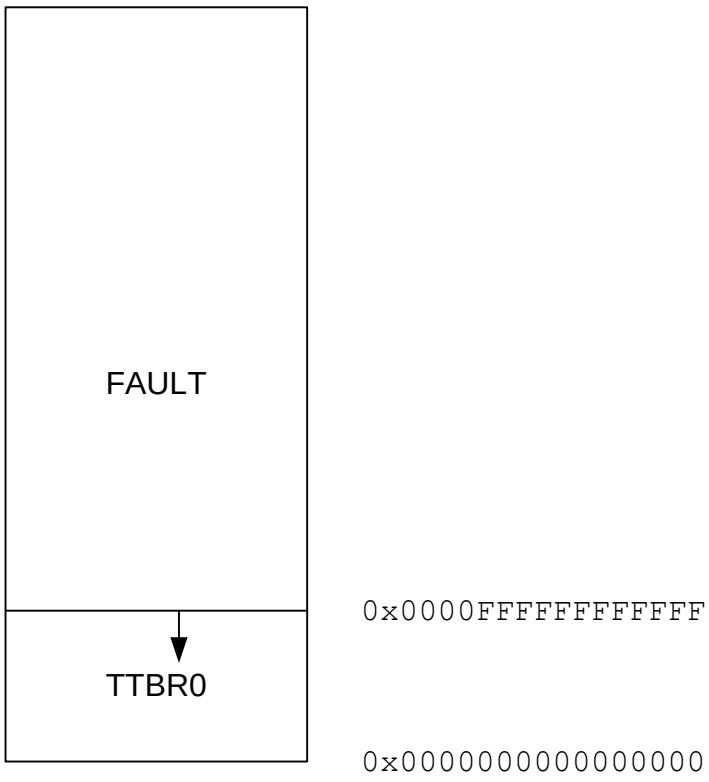
- TTBCR is renamed as the TCR_EL1 and is extended to be 64-bit register to allow additional control state to be added. The lower 32 bits follows broadly the same encoding as the stage 1 TTBCR in the LPAE. The key differences are:
  - The EAE bit is UNK/SBOP.
  - The T*SZ fields are extended by 3 bits to become 6 bit unsigned numbers as discussed in section 3.2, taking some of the UNK/SBZP fields defined in the LPAE, and the size of the input address space described is defined as $2^{64-T*SZ}$ Any input address outside the appropriate address space causes a Translation Fault.
    - T0SZ[5:0] occupies TCR_EL1[5:0]
    - T1SZ[5:0] occupies TCR_EL1[21:16]

The full layout of the TCR_EL1 is described in section 3.10.1

- The number of levels of translation table walk depends on the value of T*SZ and whether the 4KByte granule size, a 16Kbyte or the 64KByte granule size is used. This is discussed in section 3.2.
- There is a single FAR_EL1, which becomes a 64-bit value, holding the Faulting 64-bit address. The FAR_EL1 becomes UNKNOWN when executing in EL0. See section 3.5
- When EL0 is using AArch32, all addresses generated by the 32-bit architecture are 0-extended.
- The interpretation of the Access Permission bits, including PXN and XN, is changed when EL1 is using AArch64 as described in section 3.13

The First stage of translation used by EL2 when EL2 is using AArch64 and by EL3 when EL3 is using AArch64, is as follows:

- A single 64-bit TTBR0 (for each Exception Level), which requires that the upper bits of the VA are 0. The address map is therefore as shown in the following diagram:



- The TCR_ELx (for each EL2 and EL3) remains a 32-bit register that follows broadly the same encoding as the stage 1 HTCR in the LPAE. The key difference is that the T0SZ field is extended by 3 bits to become 6 bit unsigned numbers as discussed in section 3.2, and a field in the upper half of the register holds details of the supported output address size.

  The full layout of the TCR_EL2 and TCR_EL3 is described in section 3.10.2.

- A configuration bit is provided in the TCR_ELx (for each of EL2 and EL3) to determine whether bits[63:56] of the virtual address is considered for the address comparison. This bit allows the top 8 bits to be used to hold additional information about an address that can be used by the software, provided that the software ensures that during manipulation of the address these top bits are correctly preserved (see also section 3.19)

Note: Ordinary arithmetic on an address will tend to preserve this information.

- The number of levels of translation table walk depends on the value of T0SZ and whether the 4KByte granule size, a 16Kbyte granule size or the 64KByte granule size is used. This is discussed in section 3.2.
- There is an FAR at EL2 and an FAR at EL3, each of which becomes a 64-bit value, holding the Faulting 64-bit address. The FAR_EL2 becomes UNKNOWN when executing in EL1 or EL0. The FAR_EL3 becomes UNKNOWN when executing in EL2, EL1 or EL0.
- When executing in EL2 or EL3 using AArch64 some bits in the translation tables have reserved values:
    - nG field is ignored by hardware and is treated as if it is Zero
    - APTable[0] bit is ignored by hardware and is treated as if it is Zero
    - AP[1] bit is ignored by hardware and is treated as if it is One
    - PXNTable bit is ignored by hardware and is treated as if it is Zero
    - PXN field is ignored by hardware and is treated as if it is Zero
- When executing in EL2 using AArch64 some bits in the translation tables have reserved values:
    - NSTable and NS are is ignored by hardware and treated as if they are one.
- The interpretation of the Access Permission bits, including XN, is changed when EL2 is using AArch64 and when EL3 is using AArch64 as described in section 3.13
- When executing in EL3 using AArch64, the NSTable bit has no effect in making translation table entries non-global.

In all cases, the Fault Status information is held in the Exception Syndrome Register applicable to each exception level (other than Level 0), as described in the ARMv8 Exception model.

If the first stage of translation is disabled, then the Output Address from the first stage is equal to the input address. If this outside the range of addresses supported, this can lead to data aborts, either from the input address range of the second stage translation if there is a second stage of translation enabled, or as a result of the physical address size check on the output address (see section 3.7).

The output address of the first stage of translation is limited in the LPAE to 40bits, though the descriptors in the Large Physical Address Extensions reserve sufficient space to make extending the output address to support bits straightforward. In the ARMv8 architecture, when using AArch64, the address size supported is IMPLEMENTATION DEFINED between certain allowed values as discussed in section 3.7. As a result the Output Address field in all descriptors and the translation table base registers is extended to 48 bits.

3.4 Second Stage of Translation under 64-bit Exception Level control

The second stage of translation, that from Intermediate Physical Address (VA) to Physical Address (PA), applies only to execution in Non-secure EL0 and EL1 and is based on the Large Physical Address Extensions. This stage of translation is altered in the ARMv8 architecture when using AArch64 when compared with the translation system described in the Large Physical Address Extensions (LPAE) defined in the ARM ARM revC[3] . This is because the input address, the IPA, can be up to 48 bits in size, as opposed to the 40 bits supported in the LPAE.

In addition, the ability to support either a 16-Kbyte Granule or a 64-Kbyte Granule is provided, as described in section 3.2.

The output address fields of the descriptors are extended to 48 bits in the same way as with the stage 1 translation system, though the actual Physical Address size supported is IMPLEMENTATION DEFINED between certain allowed values as discussed in section 3.7.

The number of levels of translation table walk depend on the value of VTCR_EL2.T0SZ and VTCR_EL2.SL0 and whether the 4KByte granule size, a 16Kbyte granule or the 64KByte granule size. This is discussed in section 3.2.

The AArch64 VTCR_EL2 remains a 32-bit register that follows the same encoding as VTCR in the LPAE. The key difference is that the T0SZ field is extended by 3 bits to become 6 bit unsigned numbers as discussed in section 3.2, and a field in the upper half of the register holds details of the supported output address size.

The full layout of the VTCR_EL2 is described in section 3.10.3

The starting level of translation table walk depends on the VTCR_EL2.SL0 field, and the legal starting level values depends on T0SZ field and whether the 4 KByte granule, 16Kbyte or 64 KByte granule is used, as shown in section 3.2

Stage 2 Fault Status information is presented in the EL2 ESR register as described in the Virtualization Extensions.

Stage 2 Fault Address information is presented as both VA and IPA, using 2 64-bit registers, though the IPA is invalid in the same cases as it is invalid in the Virtualization Extensions. These registers are UNKNOWN when executing in EL0 and EL1. See section 3.5.

The interpretation of the Stage 2 Access Permission bits, including XN, is changed when EL2 is using AArch64 as described in section 3.13

3.5 Fault Address Information

The Fault Address Registers are organised in a similar way as with the virtualization extensions, as defined in the ARM ARM revC [3] and as described in the ARMv8 Exception Model documentation [1]. For exceptions taken in an Exception Level using AArch64:

- FAR_EL1 holds the VA of
    - Faults generated at Stage1 by execution in EL0 or EL1
    - Synchronous external aborts taken in EL1
    - The data address of Watchpoints taken in EL1
- HPFAR_EL2 holds the IPA for Second stage Faults generated by execution in EL0 and EL1 in the cases that this is valid (as defined in the Virtualization Extensions)
- FAR_EL2 holds the VA of:
    - Second stage Faults generated by execution in EL0 or EL1
    - First stage Faults generated by execution in EL2
    - Synchronous external aborts taken in EL2
    - The data address of Watchpoints taken in EL2
- FAR_EL3 holds the VA of:
    - First stage Faults generated by execution in EL3
    - Synchronous external aborts taken in EL3

In all cases, there is no distinction between a Data FAR and the Instruction FAR, a single register is used.

3.6 Fault Status Information

In AArch64, fault status information is reported within the Exception Syndrome Register of the target exception level.

The use of a Level 0 translation means that the LL code for distinguishing which level of the translation system caused the Fault is extended to use the value 0 to indicate a fault at Level 0 of the translation.

Translation Faults associated with the EPD0/1 bits or input addresses which are out of range are reported as Level 0 Translation Faults.

3.6.1 Handling of External Aborts

The ARMv7 architecture offers support for both synchronous and asynchronous external aborts, leaving it as an implementation defined feature whether any particular cause of an abort is treated synchronously or asynchronously.

Instruction fetches, data reads, data writes and translation table walks can all give rise to external aborts, and it is IMPLEMENTATION DEFINED for all of these cases whether the abort is reported synchronously or asynchronously. The architecture does not mandate that any external aborts are reported synchronously/

Synchronous Aborts

Handling of synchronous external aborts, if they are supported by an implementation, is the same as is described in the LPAE and use the appropriate abort code in the Exception Syndrome Register.

Asynchronous Aborts

Asynchronous aborts are basically a specific form of interrupt.

For the ARMv8 architecture, the Exception Model renames the asynchronous external abort when taken in an Exception Level using AArch64 as the SError interrupt, and handles it using a specific vector. An asynchronous external abort taken in an Exception Level using AArch64 also causes an update to the Exception Syndrome Register at that level using a dedicated Exception Class.

Note: Mechanisms for provide detailed syndrome information for the error interrupt is IMPLEMENTATION DEFINED. For exceptions taken in AArch64, the ESR has an IMPLEMENTATION DEFINED field that can be used for characterising the reasons for the Error interrupt.

3.6.2 Prioritization of synchronous aborts from a stage of translation

For a single stage of translation, the priority of the memory management faults on a memory access is as shown below (ordered highest priority to lowest priority). For memory accesses that undergo 2 stages of translation, the italic entries show where the faults from second stage translations can occur. Second stage faults within second stage translations are follow the same priority of faults.

1. Alignment fault not caused by memory type (stage 1 only)
2. Translation fault due to the input address being out of the address range to be translated or being to a TTBR that is disabled. This includes the VTCR_EL2.T0SZ being inconsistent with VTCR_EL2.SL0
3. Address Size fault on TTBR caused by either:
    - the TCR_EL1.IPS/TCR_ELx.PS/VTCR_EL2.PS check
    - the physical address being out of the range implemented
4. *Second stage abort on stage 1 level 0 table walk (including address size check of the physical address being out of the range implemented if the second stage is enabled)* - this is second stage abort during a first stage translation table walk

5. Synchronous parity fault on level 0 table walk
6. Synchronous external abort on level 0 table walk
7. Translation fault on level 0 table entry
8. Address Size fault on level 0 table entry caused by either:
    - the TCR_EL1.IPS/TCR_ELx.PS/VTCR_EL2.PS check
    - the output address being out of the range implemented
9. *Second stage abort on stage 1 level 1 table walk (including address size check of the physical address being out of the range implemented if the second stage is enabled)* - this is second stage abort during a first stage translation table walk
10. Synchronous parity fault on level 1 table walk
11. Synchronous external abort on level 1 table walk
12. Translation fault on level 1 table entry
13. Address Size fault on level 1 table entry caused by either:
    - the TCR_EL1.IPS/TCR_ELx.PS/VTCR_EL2.PS check
    - the output address being out of the range implemented
14. *Second stage abort on stage 1 level 2 table walk (including address size check of the physical address being out of the range implemented if the second stage is enabled)* - this is second stage abort during a first stage translation table walk
15. Synchronous parity fault on level 2 table walk
16. Synchronous external abort on level 2 table walk
17. Translation fault on level 2 table entry
18. Address Size fault on level 2 table entry caused by either:
    - the TCR_EL1.IPS/TCR_ELx.PS/VTCR_EL2.PS check
    - the output address being out of the range implemented
19. *Second stage abort on stage 1 level 3 table walk (including address size check of the physical address being out of the range implemented if the second stage is enabled)* - this is second stage abort during a first stage translation table walk
20. Synchronous parity fault on level 3 table walk
21. Synchronous external abort on level 3 table walk
22. Translation fault on level 3 table entry
23. Address Size fault on level 3 table entry caused by either:
    - the TCR_EL1.IPS/TCR_ELx.PS/VTCR_EL2.PS check
    - the output address being out of the range implemented
24. AccessFlag Fault
25. Alignment fault caused by the memory type
26. Permission Fault

27. *Faults arising from Second stage translation of the memory access (including address size check of the physical address being out of the range implemented if the second stage is enabled)*

28. Synchronous parity fault on the memory access

29. Synchronous External Abort on the memory access

Note: The prioritisation of TLB Conflict aborts is implementation defined, as the exact cause of these aborts depends on the form of TLBs implemented.

3.7 Physical Address Size

The ARMv8 architecture is designed to support a range of different applications, and the range of Physical Address required by these applications varies significantly. As a result, the ARMv8 architecture provides the controls and identification mechanisms as follows when using AArch64:

- The actual physical address size that is implemented is discoverable using the ID_AA64MMFR0 register described in [1].

- For each enabled stage of translation when the associated exception level is using AArch64, the maximum output address size is specified by a 3-bit code held in the relevant TCR_ELx. This size is used to check that translation table entries and the translation table base register being used in that stage of translation have the appropriate bits of the address in the translation tables as zero and causes a new class of Data or Instruction Abort, Address Size Fault, if this is not the case. The Fault encoding 0000LL is used to encode this fault, where LL is the translation table level that caused the fault. Address Size faults from the translation table base register are encoded using level 0.

  An Address Size Fault cannot be held in a TLB or other caching structure, and all statements about not caching Translation Faults in the TLB apply also the Address Size Faults.

  Address Size Faults are generated by address based Instruction Cache and Data Cache maintenance instructions if the output address is out of range as specified by the PS or IPS field in the TCR_ELx or VTCR_EL2, or outside the address range that has been implemented.

  Address Size Faults can be reported in the PAR_EL1 for Address Translation instructions if the output address is out of range as specified by the PS or IPS field in the TCR_ELx or VTCR_EL2, or outside the address range that has been implemented.

For the AArch64 translation system, if the output address specified by either stage of translation is too large to access the implemented physical address space of the processor then the address is treated as an Address Size fault for the translation level and stage that generated the output Address. If the second stage of translation is disabled, then the second stage does not generate a Physical Address, and so in this case the Address Size fault is treated as a stage1 fault.

Note: The Output Address Field in the translation table entries and translation table base registers are extended to 48 bits from that described in the ARMv7 Large Physical Address Extensions for both AArch32 and AArch64 execution; bits above bit[47] in the descriptors or translation table base registers have no effect on the Output Address.

Note: If the first stage of translation is disabled, then if the input address is larger than the actual physical address size that is implemented, then stage 1 will generate a level 0 Address Size fault that is treated as a Stage 1 fault. In this case, if the address was generated while executing in EL0, then if HCR.TGE==0, it is taken in EL1, otherwise it is taken in EL2.

> **Note:** Where two stages of translation are being used, and the second stage of translation is enabled, then if the output address from a stage 1 translation does not generate a stage 1 Address Size fault and is larger than the input address size specified for the second stage translation then this causes a second stage translation fault as described in the Large Physical Address Extensions. This is a standard

consequence of the input address size check for the second stage of translation and is not affected by the final address size check.

The TCR_ELx size field has the following values

TCR_EL1<34:32> **TCR_EL2/3<18:16>** **VTCR_EL2<18:16>**	**Output Address size in bits**	**Corresponding Address size**
000	32 bits	4 GBytes
001	36 bits	64 GBytes
010	40 bits	1 TByte
011	42 bits	4 TByte
100	44 bits	16 TByte
101	48 bits	256 TByte
Other values	RESERVED	RESERVED

The RESERVED values behave in the same way as the 101 encoding, but software should not rely on this property as the behaviour of the RESERVED values might change in a future revision of the architecture.

Note: The provision of an encoding for 42-bit output address corresponds to a natural alignment with the 64 KByte granule walking system and so is included for this reason.

3.7.1 IPA Size supported

The maximum value of the IPA size, which is configured by the VTCR_EL2.T0SZ and which determines the number of levels of translation table walk, is limited by the Physical Address size supported by the implementation, as reported in the ID_AA64MMFR0[3:0], according the following table:

Physical Address Size supported	**Effective Minimum value of VTCR_EL2.T0SZ**	**Maximum Value of the SL0 field for 4Kbyte Granule**	**Maximum Value of the SL0 field for 16Kbyte Granule**	**Maximum Value of the SL0 field for 64Kbyte Granule**
32 bits	32 for EL1 using AArch64, 24 for EL1 using AArch32	1	1	1
36 bits	28 for EL1 using AArch64, 24 for EL1 using AArch32	1	1	1
40 bits	24	1	1	1
42 bits	22	1	2	1
44 bits	20	2	2	2
48 bits	16	2	2	2

If the VTCR_EL2.SL0 value is programmed to a larger value than the maximum value shown above, this results in a second stage level 0 translation fault on any memory access that uses the second stage of translation.

If the VTCR_EL2.T0SZ field is programmed to a smaller value than the Effective Minimum Size shown in this table, then an implementation can do one of the following things:

- The value in the VTCR_EL2.T0SZ field is treated as if it is that minimum value for all purposes other than for a read-back of the value held in the register
- The value in the VTCR_EL2.T0SZ field is treated as if it is that minimum value for all purposes other than :
    - a read-back of the value held in the register, and
    - for checking whether the VTCR_EL2.SL0 field value is consistent with the programmed value of the VTCR_EL2.T0SZ
- Any use of the second stage translation gives rise to a second stage level 0 translation fault

Note: As any stage 1 output address from that is larger than:

- the PASize for a stage 1 AArch64 translation system, or
- 40 bits for a stage 1 AArch32 translation system

will give an Address Size Fault, programming the VTCR_EL2.T0SZ to a smaller value than the Effective Minimum value of VTCR_EL2.T0SZ will not cause a larger address range to be supported than can be achieved with the minimum Effective Minimum value.

3.7.2 Effect of a larger Physical Address on AArch32 execution

For the ARMv8 Architecture, when executing in an Exception Level whose accesses will be translated by a translation system specified in the ARMv7 architecture (as described in section 3.1) the output address field in translation table entries and in the translation table base registers used for that translation are extended to 48 bits. If bits[47:40] of an output address are not all 0 then an address size fault is generated in the same way as for Exception Levels using AArch64 when they have an output address that is out of range as specified by the PS or IPS field in the TCR_ELx or VTCR_EL2, or outside the address range that has been implemented.

Note: This implies that the Large Physical Address extensions gains the encoding 0000 for an Address Size Fault encoding, and uses the encoding LL==00 for such a fault from the TTBRx, in both the DFSR and HSR.

If an implementation supports 40 or more bits of physical address, the ID_MMFR3[27:24] Cached Memory size supported field takes the value 0010, indicating 40 bits.

3.8 Misprogramming of the Contiguous Bit

The block translation table entries for AArch64 execution and LPAE AArch32 execution in the ARMv8 architecture contain a Contiguous Bit (previously named Contiguous Hint) that allows

- 16 adjacent translation table entries (aligned such that the upper 5 bits of the Input address range to index the table are all the same) for a 4KB translation granule, or
- 128 adjacent translation table entries (aligned such that the upper 4 bits of the Input address range to index the table are all the same) for a 16KB translation granule at level3, or
- 32 adjacent translation table entries (aligned such that the upper 6 bits of the Input address range to index the table are all the same) for a 16KB translation granule at level2 or above, or

- 32 adjacent translation table entries (aligned such that the upper 8 bits of the Input address range to index the table are all the same) for a 64KB translation granule

point to a contiguous Output Address range. The contiguous Output Address range must be aligned to the size described by the 16 or 32 translation table entries at the same level of translation table.

This bit can then allow the TLB to cache a single entry in the TLB covering those multiple entries. This provides a capability similar to supersections and large pages, in a systematic manner

If:

- each of the adjacent entries does not have the contiguous bit set, or
- the output address of any of the entries is not consistent all of the entries pointing to the same aligned output address range, or
- the attributes or permissions of each of the entries are not the same

then this is a programming error that can result in the TLB containing overlapping entries, and as a result, the output address, memory permissions or attributes for a lookup might be corrupted, and might be equal to values that are not consistent with any of the programmed translation table values. It might also give rise to TLB Conflict aborts in some implementations.

The architecture guarantees that misprogramming of the contiguous bit cannot provide a mechanism for:

1. code running at an EL1 or EL0 to access regions of physical memory that are not accessible by programming of the translation table entries with arbitrary chosen values from EL1 without misprogramming of the contiguous hint
2. code running at an EL1 or EL0 to access regions of physical memory with attributes or permissions that are not accessible by programming of the translation table entries with arbitrary chosen values from EL1 without misprogramming of the contiguous hint
3. code running in the non-secure state to access secure physical memory

Note: Hardware implementations must ensure that use of the contiguous bit cannot provide a mechanism for avoiding output address range checking, as might occur if the contiguous bit is used with block sizes of 0.5GB or 1GB in a system with an output address configured to be 4GB. This might involve suppression of the contiguous bit for entries such entries in such systems.

Note: the name Continguous Bit has been used to replace “Contiguous Hint” which was thought to be potentially misleading, as while it is a hint to hardware caching, it should only be set by software in situations where it is being used to indicate guaranteed contiguousness, and so is not really a hint.

3.9 ASID Size

The ARMv7architecture as extended by the Large Physical Address Extensions defines that the ASID size is 8-bits. For the ARMv8 architecture, the ASID size is an implementation defined choice between 8 and 16 bits of ASID while executing using AArch64. When an implementation has 16 bits of ASID, a configuration bit in the TCR_EL1 can select whether the upper 8 bits of the TTBRx_EL1 are ignored by hardware for every purpose except reading back the register (and are treated as if they are all zeros when used for allocation and matching entries in the TLB), or whether they are used for allocation and matching in the TLB; this enables a more strict backwards compatibility to be maintained on an implementation with 8-bits of ASID.

The LPAE Translation System has ensured that all ASID fields in registers have an 8 bit reserved field above them. In all cases these fields are extended to be 16 bit fields for the AArch64 registers and instructions. If an implementation has only 8 bits of ASID, then the upper 8 bits of these fields are treated as RES0.

Rationale: The size of the ASID determines the expected frequency of ASID roll-over, and feedback to date has indicated that the frequency of ASID rollover is sufficiently low with 8-bits of ASID for existing systems. For the 64-bit architecture, systems involving large numbers of processors in a coherent arrangement are likely to cause an increase ASID use so creating an incentive for increasing the ASID size for such systems.

The ASID size when executing using an AArch32 translation system remains at 8 bits. In implementations that have 16 bits of ASID, the ASID used is zero-extended to 16 bits.

3.9.1 ASID Size ID mechanism

The ID _AA64MMFR0 register provides identification for the ASID size as described in [1].

3.10 TCR_ELx and VTCR_EL2 Register formats

The AArch64 TCR_ELx and VTCR_EL2 are based on the LPAE specification of the equivalent registers for AArch32. The AArch64 registers are extended to accommodate the selection of different sized granules, and to select ASID size. The layout of these registers is shown in this section.

3.10.1 TCR_EL1

Bits[63:39] : RES0

Bit[38]: TBI1: Top Byte ignored – indicates whether the top byte of the input address should be used for address match for the TTBR1 region

0 – Top Byte used in the address calculation

1 – Top Byte ignored in the address calculation

See section 3.19 for the function of this bit

Bit[37]: TBI0: Top Byte ignored – indicates whether the top byte of the input address should be used for address match for the TTBR0 region

0 – Top Byte used in the address calculation

1 – Top Byte ignored in the address calculation

See section 3.19 for the function of this bit

Bit[36]: AS : ASID Size

0 – 8 bit

1 – 16 bit

If only 8 bits are implemented, then this field is RES0

Bit[35]: RES0

Bits[34:32]: IPS: IPASize – see section 3.7

Bit[31:30] : TG1: TTBR1_EL1 Granule size

10 – 4KByte

11 – 64Kbyte

01 – 16K byte

00 – Reserved

If the value is programmed to either a reserved value, or to a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of one of the sizes that has been implemented for all purposes other than the value read back for this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

Note: The encoding here is different from TG0 for backwards compatibility reasons

Bit[29:28]: SH1 – Shareability attributes for TTBR1 as described in LPAE

Bit[27:26]: ORGN1 – Outer Cacheability attributes for TTBR1 as described in LPAE

Bit[25:24]: IRGN1 – Inner Cacheability attributes for TTBR1 as described in LPAE

Bit[23]: EPD1 – Translation Table walk disable for TTBR1 as described in LPAE

Bit[22]: A1 – ASID selection from TTBR1 as described in LPAE

Bit[21:16]: T1SZ – Size of virtual address for TTBR1

Bit[15:14] : TG0: TTBR0_EL1 Granule size

00 – 4KByte

01 – 64Kbyte

10 – 16Kbyte

11 - reserved

If the value is programmed to either a reserved value, or to a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of one of the sizes that has been implemented for all purposes other than the value read back for this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

Bit[13:12]: SH0 – Shareability attributes for TTBR0 as described in LPAE

Bit[11:10]: ORGN0 – Outer Cacheability attributes for TTBR0 as described in LPAE

Bit[9:8]: IRGN0 – Inner Cacheability attributes for TTBR0 as described in LPAE

Bit[7]: EPD0 – Translation Table walk disable for TTBR0 as described in LPAE

Bit[6]: RES0

Bit[5:0]: T0SZ – Size of virtual address for TTBR0

3.10.2 TCR_EL2, TCR_EL3

Bit[31]: RES1

Bit[30:24]: RES0

Bit[23]: RES1

Bit[22:21]: RES0

Bit[20]: TBI: Top Byte ignored – indicates whether the top byte of the input address should be used for address match

0 – Top Byte used in the address calculation

1 – Top Byte ignored in the address calculation

Bit[19]: RES0

Bit[18:16]: PS: PASize – see section 3.7

Bit[15:14]: TG0: TTBR0_ELx Granule size

00 – 4KByte

01 – 64Kbyte

10 – 16Kbyte

11 – RESERVED

If the value is programmed to either a reserved value, or to a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of one of the sizes that has been implemented for all purposes other than the value read back for this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

Bit[13:12]: SH0 – Shareability attributes for TTBR_ELx as described in LPAE

Bit[11:10]: ORGN0 – Outer Cacheability attributes for TTBR_ELx as described in LPAE

Bit[9:8]: IRGN0 – Inner Cacheability attributes for TTBR_ELx as described in LPAE

Bit[7:6]: RES0

Bit[5:0] T0SZ – Size of virtual address for TTBR_ELx.

3.10.3 VTCR_EL2

Bit[31]: RES1

Bit[30:24]: RES0

Bit[23]: RES0

Bit[22:19]: RES0

Bit[18:16]: PS: PASize – see section 3.7

Bit[15:14] : TG0: VTTBR0_EL2 Granule size

00 – 4KByte

01 – 64Kbyte

10 – 16Kbyte

11 – Reserved

If the value is programmed to either a reserved value, or to a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION

DEFINED choice of one of the sizes that has been implement for all purposes other than the value read back for this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

Bit[13:12]: SH0 – Shareability attributes for VTTBR_EL2 as described in LPAE

Bit[11:10]: ORGN0 – Outer Cacheability attributes for VTTBR_EL2 as described in LPAE

Bit[9:8]: IRGN0 – Inner Cacheability attributes for VTTBR_EL2 as described in LPAE

Bit[7:6] SL0 – Starting level of the VTCR_EL2 addressed region. The encoding of this depends on the memory allocation size as shown in section 3.2

Bit[5:0] T0SZ – Size of virtual address for VTTBR_EL2.

3.11 Alignment handling

The alignment handling of ARMv8 AArch64 follows the principles of AArch32, as described in ARMv7, in a more regular fashion:

- All instructions other than load exclusive/store exclusive, load with acquire and store with release (see section 4.2.2) that load or store a single or multiple registers can be configured using the SCTLR.A bit to have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed, or otherwise perform misaligned accesses.

  Note; The ARMv8 architecture therefore introduces the requirement for a 64-bit and 128-bit unaligned access.

  Note: the alignment check is based on element size, not overall access size, which has a material difference for Advanced SIMD element and structure loads and stores, as well as load/store pair instructions.

- All load exclusive/store exclusive and load with acquire and store with release memory accesses that access a single element have an alignment check that the address being accessed is aligned to the size of the data elements being accessed.

- All load exclusive/store exclusive memory accesses that access a pair of elements have an alignment check that the address being accessed is aligned to the size of the pair of data elements being accessed.

In keeping with the Virtualization Extensions, an access to any type of Device memory will cause an alignment check.

Where an alignment check is performed, a failure results in an Alignment fault, which is taken as a Data Abort exception. These exceptions are taken at the lowest exception level that can handle the exception, consistent with the basic requirement that the exception level never decreases on an exception. Thus alignment faults at EL0/EL1 will be taken at EL1 unless redirected by the HCR.TGE, alignment faults at EL2 will be taken at EL2, alignment faults at EL3 will be taken at EL3.

An exception to this principle applies for alignment exceptions from EL0 or EL1 that are only caused as a result of the memory attribute being marked as any type of Device memory . In this case: if the first stage of translation marked the location as any type of Device memory, then the exception is taken at EL1 otherwise if the second stage of translation marked the location as any type of Device memory, the exception is taken at EL2.

3.12 Endianness

The size of an element that is loaded or stored is the size that is used for the purpose of endian conversion for Advanced SIMD element and structure loads and stores.

The size of a data value that is loaded or stored is the size that is used for the purpose of endian conversion for Floating-point, Advanced SIMD and General purpose register loads and stores.

> **Note;** The ARMv8 architecture therefore introduces the requirement for a 128-bit endian conversion.

3.13 Modified Access Permission interpretation

The interpretation of the access permission bits in the translation tables is modified when using a translation table from an Exception Level using AArch64, to allow for the provision of an Execute Only permission in the following cases:

- For execution in EL0, as designated in the stage 1 translation tables, controlled from EL1 using AArch64
- For execution in Non-secure EL0 or EL1, as designated in the stage 1 translation tables, controlled from EL2 using AArch64.

The interpretation of the AP, XN and PXN bits is when using a translation table from an Exception Level using AArch32, is the same as described in the ARMv7 architecture as extended by the Virtualization Extensions and the Large Physical Address Extensions.

3.13.1 First Stage Execute-Only Provision

For the first stage translation when EL1 is using AArch64, Execute-only provision is based on a redefinition of the translation table XN access permission to become “User XN” (UXN), and so only prevents user execution permission. In addition, the checking for user execution permission is made independently of having read permission or having privileged execution permission. The redefinition of XN applies to UXN applies also to the XNTable bit, which becomes UXNTable.

The global override SCTLR.WXN forces both PXN and UXN if a location is Writeable from EL0, and forces PXN if the location is Writeable from EL1 but not writeable from EL0.

The table below shows the effect of the 4 access permission bits when in EL1 is usingAArch64 (which provides Execute only) and when in AArch32 using the LPAE format.

				EL1 using AArch64 behaviour		**EL1 using AArch32 behaviour**	
AP[2]	**AP[1]**	**UXN / XN**	**PXN**	**EL1**	**EL0**	**EL1**	**EL0**
0	0	0	0	R W (X)	X	R W (X)	None
0	0	0	1	R W	X	R W	None
0	0	1	0	R W (X)	None	R W	None
0	0	1	1	R W	None	R W	None
0	1	0	0	R W [note1]	R W (X)	R W (**X**)	R W (X)
0	1	0	1	R W	R W (X)	R W	R W (X)

0	1	1	0	R W [note 1]	R W	R W	R W
0	1	1	1	R W	R W	R W	R W
1	0	0	0	R X	X	R X	None
1	0	0	1	R	X	R	None
1	0	1	0	R X	None	R	None
1	0	1	1	R	None	R	None
1	1	0	0	R X	R X	R X	R X
1	1	0	1	R	R X	R	R X
1	1	1	0	R X	R	R	R
1	1	1	1	R	R	R	R

R = Read permission granted , W = Write permission granted X = Execute permission granted

Permissions in brackets indicate those permissions removed when SCTLR_EL1.WXN is set

Permission in bold indicate those permissions removed when SCTLR.UWXN is set - when EL1 is using AArch64 all regions that are writeable at EL0 are treated as being PXN at EL1.

Note 1: These cases are not executable as the AArch64 execution treats all regions that are writeable at EL0 as being PXN at EL1.

Note 2: All capabilities in the AArch32 behaviour other than the ability to support EL1 execution from regions that EL0 writeable, are supported in AArch64.

When executing in EL2 or EL3, the behaviour is unchanged between using AArch32 and using AArch64, and is as shown in the following table:

AP[2]	UXN / XN	EL2/ EL3 using AArch64 behaviour	EL2/EL3 using AArch32 behaviour
0	0	R W (X)	R W (X)
0	1	R W	R W
1	0	R X	R X
1	1	R	R

R = Read permission granted , W = Write permission granted X = Execute permission granted

Permissions in brackets indicate those permissions removed when SCTLR_EL2/EL3.WXN is set

This behaviour is consistent with the behaviour that when executing in EL2 or EL3, the following fields in the translation tables are treated as they had fixed values:

- APTable[0] bit is ignored by hardware and is treated as if it is Zero
- AP[1] bit is ignored by hardware and is treated as if it is One
- PXNTable bit is ignored by hardware and is treated as if it is Zero
- PXN field is ignored by hardware and is treated as if it is Zero

3.13.2 Second Stage Execute-Only Provision

Second stage Execute-Only provision when EL2 is using AArch64 comes separating execution permission from the read permission in the second stage translation tables. Thus the second stage permissions are as shown in the following table:

			EL2 using AArch64	**EL2 using AArch32**
HAP[2]	**HAP[1]**	**XN**	**EL1 and EL0**	**EL1 and EL0**
0	0	0	X	None
0	0	1	None	None
0	1	0	R X	R X
0	1	1	R	R
1	0	0	W X	W
1	0	1	W	W
1	1	0	W R X	W R X
1	1	1	W R	W R

R = Read permission granted

W = Write permission granted

X = Execute permission granted

The behaviour of the second stage translation when EL2 is using AArch32 is unchanged from the definition in the ARMv7 architecture as extended by the Virtualization Extensions and Large Physical Address Extensions.

3.14 Shareability of Device, and Normal Non-cacheable Memory

In the ARMv8 architecture, when using either AArch64 translation tables or AArch32 translation tables, for the following memory types, any shareability attributes described in the translation tables are ignored, as the memory is always regarded as being Outer Shareable:

- Any Device memory type
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable

3.15 Multi-register Loads and Stores to Strongly-ordered and Device Memory

For all instructions that load or store more than one general purpose register from an exception level using either AArch32 or AArch64, there is no requirement for the memory system beyond the processor to be able to identify the size of the elements accessed by these load/store instructions, even for accesses to any type of Device memory.

For all instructions that load or store more than one general purpose register from an exception level using AArch64, the order in which the registers are accessed is not defined by the architecture even for accesses to any type of Device memory.

Note: this is consistent with the treatment of LDRD/STRD in ARMv7.

For all instructions that load or store one or more Advanced SIMD registers from an exception level using either AArch32 or AArch64, there is no requirement for the memory system beyond the processor to be able to identify the size of the elements accessed by these load/store instructions, even for accesses to any type of Device memory.

3.16 AArch64 Memory Attributes for Access to Peripherals

The ARMv8 architecture provides a revised set of memory types for accessing peripherals, to replace the Strongly-ordered and Device memory.

Four different Memory Types are supported:

Device-nGnRnE: Device non-Gathering, non-Reordering, no Early Write Acknowledgement

Device-nGnRE: Device non-Gathering non-Reordering, Early Write Acknowledgement

Device-nGRE: Device non-Gathering Reordering, Early Write Acknowledgement

Device-GRE: Device Gathering Reordering, Early Write Acknowledgement

Collectively, these are referred can be referred to as “any Device memory type”. Going down this least, the memory types are described as getting “weaker”; conversely going up the list the memory types are described as getting “stronger”.

Note: it is recognised that not all implementations will make a distinction between each of these memory types, and it is not required that implementations do so. For each of the attributes, the implementation rules are described.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to a memory location of any Device memory type; that is each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program. There are three exceptions to this rule:

    1. Reads generated by the Advanced SIMD instructions are permitted to access bytes that are not explicitly accessed by the instruction, provided the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

    2. Where a load or store instruction performs a sequence of memory accesses as opposed to a single single-copy atomic access, as defined under the single-copy atomicity rules, then these accesses might occur multiple times as a result of executing the load or store instruction.

    3. Reads generated by a LDNP for memory locations with the Gathering attributes that are permitted to access bytes that are not explicitly accessed by the instruction, provided the bytes accessed are in a 128-byte window, aligned to 128-bytes, that contains at least one byte that is explicitly accessed by the instruction (see section 3.16.1)

    Note: write speculation visible to other observers is prohibited for all memory types.

- a writes to a memory location of any Device memory type completes (is globally observed) in finite time.

- where a location within a peripheral changes without an explicit write by an observer, as might be the case for a peripheral location that holds status information, such changes must also be globally observed in finite time.

- a completed write to a memory location of any Device memory type is observed by all observers in finite time without the need of explicit cache maintenance
- all accesses are treated as Outer-shareable.
- Writes to a memory location of any Device memory type must reach the endpoint for that address in the memory system – typically a peripheral or some physical memory – in finite time.
- For any Device memory type (and also for Normal Non-cacheable memory), a DMB will ensure that all members of Group A (defined by the DMB) will reach a single peripheral or block of memory of IMPLEMENTATION DEFINED size, as defined by the peripheral or block of memory, before any members of Group B (defined by the DMB). This applies for all types of DMB.
- All accesses to any Device memory type must be aligned – an unaligned access will generate an Alignment fault at the first stage of translation that defined the location as being Device.

  Note: in the non-secure EL1 translation regime in systems where HCR_EL2.TGE=1 and HCR_EL2.DC=0, any resulting Alignment faults from fact that all locations are treated as Device are stage 1 faults. This causes the ESR_EL2.ISS.ISV field to be 0 in this case.

  Note: this is handled in the same was as alignment faults are handled for Strongly-ordered or Device memory types in the ARMv7 Virtualization extensions.

Hardware does not prevent speculative instruction fetches to memory location of any Device memory type unless they are marked as Execute-never for all exception levels.

Note: In general, not marking a to memory location of any Device memory type as Execute-never for all exception levels is regarded as a programming error

The memory types for Translation table walks cannot be defined as any Device memory type within the TCR_ELx, but as a result of the second stage of translation, first stage translation table walks can be made to memory locations with any Device memory type, and these can be made speculatively. The HCR_EL2.PTW bit will cause a second stage permission fault if a first stage translation table walk is made to any Device memory type.

Note: In general, making a translation table walk to any Device memory is regarded as a programming error

These four memory types have three additional attributes:

- Gathering
- Re-ordering
- Early Write Acknowledgement Hint

3.16.1 Gathering

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses (either reads or writes) to the same memory location can be merged into a single transaction
- Multiple memory accesses (either reads or writes) to different memory locations can be merged into a single memory transaction on an interconnect

Note: This also applies to writebacks from the cache, whether caused by a natural eviction or as a result of a cache maintenance instruction.

For the memory types with the non-Gathering attribute, neither of these behaviours are permitted, and so the number memory accesses made corresponds to the number that would be generated by a simple sequential execution of the program.

Writes to memory types with the non-Gathering attribute that are single-copy atomic are also multi-copy atomic.

A read from a memory location with the non-Gathering attribute cannot come from a cache or buffer, but must come from the endpoint for that address in the memory system – typically a peripheral or some physical memory.

For the memory types with the Gathering attribute, either of these behaviours is permitted provided that the ordering and coherency rules of the memory location are obeyed. Gathering between memory accesses separated by a memory barrier that affects those memory accesses, such that one memory access is in Group A and one memory access is in Group B, is not permitted. Gathering between two memory accesses generated by load-acquire and/or store-release is not permitted.

Writes to memory types with Gathering attribute are not required to be multi-copy atomic .

Note: A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that :

- the accesses are not separated by barrier that affects both of the accesses (that is, one is in Group A and one is in Group B), or
- the accesses are not separated by other ordering constructions (such as the combination of a store-release and a load-acquire) that requires that the accesses are observed in order, or
- the accesses were not generated by store-release

In addition, for memory types with the Gathering attribute, reads generated by the LDNP instructions are permitted to access bytes that are not explicitly accessed by the instruction, provided the bytes accessed are in a 128-byte window, aligned to 128-bytes, that contains at least one byte that is explicitly accessed by the instruction.

Note: in keeping with the general principles of the ARM architecture, which only defines programmer visible behaviour, where it is impossible for a programmer to tell such gathering has occurred, then it is permissible to perform gathering.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute. An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute

3.16.2 Reordering

The Reordering attribute determines whether it is permissible to reorder the memory accesses to this memory type within the same rules that apply for Normal Non-cacheable memory accesses (see section 4.2) or whether a stricter order must be applied.

For the all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral must be the same as occur in a simple sequential execution of the program (that is, the accesses appear in program order). This ordering applies for all accesses using any of the memory types with the non-Reordering attribute (and so Device-nGnRE accesses are ordered with respect to Device-nGnRnE accesses to the same peripheral). If the memory accesses are not to a peripheral,then there are no restrictions from this attribute.

Note: the size of the single peripheral is the same as applies for the ordering guarantee provided by DMB instructions.

For the all memory types with the Reordering attribute, the ordering rules are the same as for accesses to Normal Non-cacheable memory (see section 4.2)

Note: in keeping with the general principles of the ARM architecture, which only defines programmer visible behaviour, where it is impossible for a programmer to tell re-ordering has occurred, then it is permissible to perform re-ordering.

Note: Some interconnect fabrics, such as PCI/PCIe perform very limited re-ordering which are not important for the usage by software. It is outside the scope of the ARM architecture to prohibit the use of a Non-reordering memory type with these interconnects.

An implementation is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute. An implementation is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute

3.16.2.1 Limitations on Non-Reordering

The non-Reordering attribute does not require any additional ordering (other than which applies to Normal Memory) between accesses with the non-Reordering attribute and accesses with the Reordering attribute.

The non-Reordering attribute does not require any additional ordering (other than which applies to Normal Memory) between accesses with the non-Reordering attribute and accesses to Normal memory.

The non-Reordering attribute does not require any additional ordering (other than which applies to Normal Memory) between accesses with the non-Reordering attribute where the accesses are to different peripherals of IMPLEMENTATION DEFINED size.

The non-Reordering attribute has no effect to the ordering of cache maintenance instructions, even if the memory location specified in the instruction has the non-Reordering attribute.

3.16.3 No Early Write Acknowledgement

The Early Write Acknowledgement is a hint to the platform memory system such that if the No Early Write Acknowledgement attribute is set, then it is recommended that the Write Acknowledgement is returned from the endpoint of the memory system, rather than being returned earlier in the memory system. This would allow a subsequent DSB barrier executed by the same processor that did a write to a memory location with the No Early Write Acknowledgement to complete after the write has reached the endpoint of the memory system – typically a peripheral or some physical memory.

When the Early Write Acknowledgement attribute is set, there is no such recommended behaviour.

Note: The Early Write Acknowledgement has no effect on the ordering rules. The purpose of signalling no Early Write Acknowledgement is to signal to the interconnect that the peripheral requires the ability to signal the acknowledgement, and in doing so give it an additional semantic that can be interpreted by the driver that is accessing to the peripheral

Note: This attribute is treated as a Hint, as the exact nature of interconnects attached to a processor is outside the scope of the ARM architecture to define, and not all interconnects provide a mechanism to ensure that a write has reached the physical endpoint of the memory system.

Note: ARM recommends that writes with the No Early Write Acknowledgement hint is used for PCIe configuration writes, though the mechanisms by which PCIe configuration writes are identified are IMPLEMENTATION DEFINED.

Note: ARM recommends that the Early Write Acknowledgement hint is not ignored within a CPU, but is made available to the system to use as appropriate.

As the Early Write Acknowledgement attribute is a hint:

- an implementation is permitted to perform an access with the Early Write Acknowledgement attribute in a manner consistent with the requirements specified by the non- Early Write Acknowledgement attribute, and

- an implementation is permitted to perform an access with the non- Early Write Acknowledgement attribute in a manner consistent with the relaxations allowed by the Early Write Acknowledgement attribute

3.16.4 Relationship with AArch32 Strongly-ordered or Device Memory type

In the ARMv8 architecture:

- the ARMv7 Strongly-ordered memory type is treated as Device-nGnRnE when using the AArch32 translation table formats.
- the ARMv7 Device memory type is treated as Device-nGnRE when using the AArch32 translation table formats.

The Device-nGRE and Device-GRE memory types are introduced into AArch32 translation table formats as part of the ARMv8 architecture when the Large Descriptor format is being used, by using the following MAIR encodings for stage1:

- MAIR[7:0] = 00001000 Device-nGRE
- MAIR[7:0] = 00001100 Device-GRE

And the following MemAttr encodings for stage 2:

- MemAttr[3:0] = 0010 Device-nGRE
- MemAttr[3:0] = 0011 Device-GRE

3.16.5 Combining of Memory attributes in two stages of translation.

Where the two stages of translation have different attributes, they are combined in a manner that gives the “strongest” memory type. Normal memory is treated as Gathering, Reordering Early Write Acknowledge

Thus:

- Combining Normal and Device memory gives Device
- Combining Non-Gathering and Gathering gives Non-Gathering
- Combining Non-Reordering and Reordering gives Non-Reordering
- Combining No Early Write Acknowledge and Early Write Acknowledge gives No Early Write Acknowledge.

3.16.6 Data Memory Attributes when the stage 1 MMU is disabled

When the stage 1 MMU is disabled, the default memory attribute from stage 1 for Data accesses is Device-nGnRnE.

For Non-secure EL0 and EL1 access, this can be overridden by the HCR_EL2.DC bit, which makes the default memory type from stage 1 Normal Non-shareable Inner WB Read-Write Allocate, Outer WB Read-Write Allocate

3.16.7 Encoding for AArch64 Device memory types

The stage1 encodings for the AArch64 Device memory types in the MAIR fields are:

- MAIR[7:0] = 00000000 Device-nGnRnE
- MAIR[7:0] = 00000100 Device nGnRE
- MAIR[7:0] = 00001000 Device-nGRE
- MAIR[7:0] = 00001100 Device-GRE

And the following MemAttr encodings for stage 2:

- MemAttr[3:0] = 0000 Device-nGnRnE
- MemAttr[3:0] = 0001 Device-nGnRE
- MemAttr[3:0] = 0010 Device-nGRE
- MemAttr[3:0] = 0011 Device-GRE

3.16.8 Instruction Fetches from Device memory

Software should avoid performing instruction fetches from Device memory, and software should mark areas of memory mapped as any form of device will also be marked as execute never for any exception level that defines the memory location has having the Device attribute.

If a region of memory has the Device attribute, and is not marked as execute never, then an implementation might perform speculative instruction accesses to this memory location at times when the MMU is enabled.

If branches cause the program counter to point at an area of memory with the Device attribute for instruction fetches which is not marked as execute never for the current exception level, an implementation can perform one of the following behaviours:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cachable attribute.
- Take a permission fault

3.17 Addition of Transient Hint to the MAIR

The ARMv8 Translation System adds support for a Transient hint for the cacheable attributes in the memory types. This transient attribute indicates that for memory access that are recommended to be allocated into the cache, the benefit of caching is for a relatively short period, and it might be beneficial to performance to restrict allocation to avoid casting out other, less transient, elements.

Note: typically use of such attributes is likely to be restricted to particular code sequences.

This is encoded using the MAIR field as shown below:

Bits[7:4]	**Meaning**
0000	Device Memory (see section 3.16
00RW (RW != 00)	Normal Memory, Outer Write-through transient
0100	Normal Memory, Outer Non-Cacheable
01RW (RW != 00)	Normal Memory, Outer Write-back transient
10RW (all values of RW)	Normal Memory, Outer Write-through non-transient

11RW (all value of RW)	Normal Memory, Outer Write-back non-transient

R= Outer Read Allocate Policy; W=Outer Write Allocate Policy

All other values UNPREDICTABLE.

Bits[3:0]	**Meaning when Bits[7:4] == '0000'**	**Meaning when Bits[7:4] != '0000'**
0000	Device-nGnRnE	UNPREDICTABLE
00RW (RW != 00)	UNPREDICTABLE	Normal Memory, Inner Write-through transient
0100	Device-nGnRE	Normal Memory Inner Non-Cacheable
01RW (RW != 00)	UNPREDICTABLE	Normal Memory, Inner Write-back transient
1000	Device-nGRE	Normal Memory Inner Write-through non-transient RW=00
10RW (RW !=00)	UNPREDICTABLE	Normal Memory Inner Write-through non-transient
1100	Device-GRE	Normal Memory Inner Write-back non-transient RW = 00
11RW (RW !=00)	UNPREDICTABLE	Normal Memory Inner Write-back non-transient

3.18 Load-exclusive and store exclusives to all types of Device, and to Normal Non-cacheable memory

For the ARMv8 architecture, Load exclusive and store exclusive instructions can be performed to any type of Device Memory and to Normal memory and perform the exclusive functionality described in the ARMv7 architecture.

In some implementations and for some memory types, the properties of the global monitor can be met only by functionality outside the processor. Some system implementations might not implement this functionality for all regions of memory. In particular, this can apply to:

- any type of memory in the system implementation that does not support hardware cache coherency
- Non-cacheable memory, or memory treated as non-cacheable, in an implementation that does support hardware cache coherency.

In such systems, it is defined by the system:

- whether the global monitor is implemented
- if the global monitor is implemented, which address ranges or memory types it monitors.

Note: To facilitate the use of load exclusive/store exclusive during times when memory translation is disabled, it can be beneficial for systems to provide at least one region of memory of at least 4 Kbytes in size within the system memory map which supports the global monitor for all ARM cores within an Inner Shareable domain. However, this is not an architectural requirement, and so architecturally compliant code that requires mutual

exclusion must not rely on having load exclusive/store exclusive to achieve such exclusion, and should use software algorithms such as Lamport's Bakery to support this.

As implementations have freedom to choose which memory types are treated as non-cacheable, the only memory types for which it is architecturally guaranteed that a global exclusive monitor is implemented are:

- Inner shareable, Inner Write-Back, Outer Write-Back Normal Memory with read-write allocation hints and not transient
- Outer shareable, Inner Write-Back, Outer Write-Back Normal Memory with read-write allocation hints and not transient

If the global monitor is not implemented for an address range or memory type, then performing an LDREX or STREX instruction to such a location can have one or more of the following effects:

- The instruction generates an external abort
- The instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Fault Status code of:
    - ESR_ELx.DFSC = 110101 for AArch64
    - DFSR.STATUS = 110101 for AArch32 when using the Long Descriptor Format (which can also be reported in the HSR.ISS[5:0] field for exceptions to Hyp mode)
    - DFSR.FS = 10101 for AArch32 when using the Short Descriptor Format
- The instruction is treated as a NOP
- The load exclusive instruction is treated as if it is to a non-shareable location but the local monitor becomes UNKNOWN
- The store exclusive instruction is treated as if it is to a non-shareable location, but with the local monitor being in an UNKNOWN state. In this case, if the store exclusive instruction is a store exclusive pair of 64-bit quantities, then the two quantities being stored might not be stored atomically
- The value held in the result register of the store exclusive instruction becomes UNKNOWN

In addition, for write transactions generated by non-processor observers that do not implement exclusive accesses or other atomic access mechanisms, it is IMPLEMENTATION DEFINED the effect that writes have on the architected monitors used by ARM processors such that it might not clear the global monitors of other processors for:

- some address ranges
- some memory types

3.19 Tagged Address Handling

The ARMv8 architecture when using AArch64, supports tagged addresses for data values, where the top 8 bits of the virtual address are ignored for the purpose of determining:

- that the address causes a Translation Fault from being out of range if the translation system is enabled,
- that the addresses causes an Address Size Fault from being out of range if the translation system is not enabled,
- the address to be invalidated for performing a TLB invalidation instruction by address.

This is controlled by:

- TCR_EL1.TBI0 for addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL1
- TCR_EL1.TBI1 for addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR1_EL1
- TCR_EL2.TBI for addresses generated in EL2 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL2
- TCR_EL3.TBI for addresses generated in EL3 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL3

These bits have an additional effect on any branch or procedure return within an exception level, exception to an exception level or exception return to an exception level, or debug state exit to an exception level where:

- For EL0 or EL1, if the associated TBI bit that would be selected by bit[55] is set, then bits[63:56] of the program counter are forced to be a sign extension of bit[55]
- For EL2 or EL3, if the TBI bit for that exception level is set, then bits[63:56] of the program counter are forced to 0.

The algorithm for this is as is shown in the following pseudo-code:

```
if (target_exception_level == EL0) || (target_exception_level == EL1) then
    if NewAddress<55> == '1' && TCR_EL1.TBI1== '1' then
        NewAddress<63:56> = '11111111';
    if NewAddress<55> == '0' && TCR_EL1.TBI0== '1' then
        NewAddress<63:56> = '00000000';
if (target_exception_level == EL2) then
    if TCR_EL2.TBI== '1' then
        NewAddress<63:56> = '00000000';
if (target_exception_level == EL3) then
    if TCR_EL3.TBI== '1' then
        NewAddress<63:56> = '00000000';
PC = NewAddress;
```

Where

`NewAddress` is the address being branched to, or being returned to

`target_exception_level` is

The current exception level for a branch or procedure return

The exception level being returned to for an exception return. If the exception return triggers the illegal exception return mechanism, it is IMPLEMENTATION DEFINED whether the target_exception_level is the exception level that was described in the SPSR at the time of the exception return, or the exception level that the exception return was executed from.

The exception level the exception is taken to for an exception entry

Note: This behaviour is added to prevent a tagged address being propagated to the program counter, though it should be noted that in the case of illegal exception return, if the implementation treats the target_exception_level as being the exception level that was described in the SPSR at the time of the exception return, and that level does not have the TCR_ELx.TBI bit set, but the exception level that the exception return was taken from has the TCR_ELx.TBI bit set, then the tagged address can be propagated to the program counter.

Note: the TCR_ELx.TBIx bits have an effect whether that translation regime is enabled or not.

Note: The tag will appear as part of the virtual address in the FAR as a result of a Data Abort or Watchpoint.

3.20 Virtual Address Incrementing and Virtual Address Space overflow

When a processor performs normal sequential execution of instructions, it effectively calculates:

(address_of_current_instruction) + (size_of_executed_instruction)

after each instruction to determine which instruction to execute next.

If this address calculation overflows 0xFFFF_FFFF_FFFF_FFFF,the program counter becomes UNKNOWN.

Note: as tagged addresses are not propagated to the program counter, there is no need to factor in the use of tagged pointers in this case.

Where an instruction accesses a sequential set of bytes which crosses the following boundary:

0xFFFF_FFFF_FFFF_FFFF and tagged addresses are not being used, or

0xxxFF_FFFF_FFFF_FFFF and tagged addresses are being used

The virtual addresses accessed for the bytes above this boundary is UNKNOWN, and, if tagged addresses are being used, the value of the tag associated with the address becomes UNKNOWN.

3.21 Non-cacheable accesses and Instruction Caches

For translation regimes controlled from AArch64 state, the ARMv8 architecture permits that non-cacheable accesses can be held in an instruction cache.

Correspondingly, the sequence for ensuring that modifications to instructions are available to be executed must include invalidation of the modified locations from the instruction cache, even if the instructions are held in non-cacheable space.

Therefore the code sequence for self-modified code to non-cacheable space in a uniprocessor system is:

```
; Enter this code with <Wt> containing the new 32-bit instruction
; to be held at a location pointed to by Xn in non-cacheable space

STR <Wt>, [Xn]
DSB             ; Ensure visibility of the data stored
IC IVAU, Xn     ; Invalidate instruction cache by VA to PoU
DSB             ; Ensure completion of the invalidations
ISB             ;
```

In a multi-processor system, the IC IVAU will be broadcast to all processors within the Inner shareable domain of the processor running this sequence, but additional software steps may need to be taken to synchronise the threads on the different processors such that the processors to execute the modified instructions can execute their ISB after the completion of the invalidation, and to avoid issues associated with concurrent modification and execution of instruction sequences.

Where larger blocks of instructions are modified, this can be achieved using the IC IALLU instruction for a uniprocessor system, or ICIALLUIS or a multiprocessor system.

Note: This section applies even when the Instruction Cache is disabled in AArch64 as described in section 3.22

3.22 Definition of Cache Disabled

When the data/unified cache is disabled for a translation regime, as determined by the SCTLR_ELx.C bit, the effect is to make data accesses and translation table walks from that translation regime to all Normal Memory types behave as Non-cacheable for all levels of data and unified cache.

For the EL0/EL1 translation regime:

- SCTLR_EL1.C==0 has an effect on the first stage of translation, by making all Normal memory data accesses non-cacheable, as well as having an effect on the EL0/EL1 stage 1 translation table walks by making those accesses Non-cacheable.
- A new bit in the HCR_EL2, HCR_EL2.CD, when HCR_EL2.CD ==1 has an effect on the second stage of translation in the non-secure state, by making all Normal memory data accesses to non-cacheable, as well as having an effect on the EL0/EL1 stage 2 translation table walks by making those accesses Non-cacheable.

  Note: The HCR_EL2.CD bit is not listed in v13 of [1] but is at bit[32] and will be added in a later revision.

  Note: The stage 1 and stage 2 cacheability attributes are combined as described in as part of the virtualization extensions in the ARM ARM revC [3]
- The SCTLR_EL1.C, HCR_EL2.DC and HCR_EL2.CD bits have no effect on the EL2 or EL3 translation regimes.
- If the HCR_EL2.DC bit is set, then the non-secure stage 1 EL1/EL0 translation regime is Cacheable regardless of the value of the SCTLR_EL1.C bit.

The SCTLR_EL2.C bit has no effect on the EL0/EL1 or EL3 translation regimes but when SCTLR_EL2.C==0, it makes all EL2 Normal memory data accesses non-cacheable, as well as having an effect on the EL2 translation table walks by making those accesses Non-cacheable.

The SCTLR_EL3.C bit has no effect on the EL0/EL1 or EL2 translation regimes but when SCTLR_EL3.C==0, it makes all EL3 Normal memory data accesses non-cacheable, as well as having an effect on the EL3 translation table walks by making those accesses Non-cacheable

The effect of SCTLR_ELx.C, HCR_EL2.DC and HCR_EL2.CD bits are reflected in the result of address translation operations in the PAR when those bits have an effect on the stages of translation being reported in the PAR.

When the instruction cache is disabled for a translation regime, as determined by the SCTLR_ELx.I bit, the effect is to make instruction accesses to all Normal Memory types behave as Non-cacheable for all levels of instruction or unified cache.

For the EL0/EL1 translation regime:

- SCTLR_EL1.I==0 has an effect on the first stage of translation, by making all Normal memory instruction accesses non-cacheable
- A new bit in the HCR_EL2, HCR_EL2.ID when HCR_EL2.ID ==1 has an effect on the second stage of translation in the non-secure state, by making all Normal memory instruction accesses to non-cacheable.

  Note: The HCR_EL2.ID bit is not listed in v13 of [1] but is at bit[33] and will be added in a later revision.

  Note: The stage 1 and stage 2 cacheability attributes are combined as described in as part of the virtualization extensions in the ARM ARM revC [3]
- The SCTLR_EL1.I and HCR_EL2.ID bits have no effect on the EL2 or EL3 translation regimes.
- If the HCR_EL2.DC bit is set, then the non-secure stage 1 EL1/EL0 translation regime is Cacheable regardless of the value of the SCTLR_EL1.I bit.

The SCTLR_EL2.I bit has no effect on the EL0/EL1 or EL3 translation regimes but when SCTLR_EL2.I==0, it makes all EL2 Normal memory instruction accesses non-cacheable.

The SCTLR_EL3.I bit has no effect on the EL0/EL1 or EL2 translation regimes but when SCTLR_EL3.I==0, it makes all EL3 Normal memory instruction accesses non-cacheable.

In addition, for the EL2, EL3 and Secure EL1 translation regimes, when the SCTLR_ELx.M==0, indicating that the stage1 translations are disabled for that translation regime, the SCTLR_ELx.I bit has the following effect:

- SCTLR_ELx.I ==0 to make instruction accesses from the stage 1 of that translation regime be to Normal memory, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- SCTLR_ELx.I ==1 to make instruction accesses from the stage 1 of that translation regime be to Normal memory, Outer Shareable, Inner Write-Through, Outer Write-Through.

For the Non-secure EL1 translation regime, when the SCTLR_EL1.M==0, indicating that the stage1 translations are disabled for that translation regime, and HCR_EL2.DC==0, the SCTLR_EL1.I bit has the following effect:

- SCTLR_EL1.I ==0 to make instruction accesses from the stage 1 of that translation regime be to Normal memory, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- SCTLR_EL1.I ==1 to make instruction accesses from the stage 1 of that translation regime be to Normal memory, Outer Shareable, Inner Write-Through, Outer Write-Through.

Note: in conjunction with section 3.21, this means that the architecturally required effect of the SCTLR_ELx.I bit is limited to its effect on caching instruction accesses in unified caches.

Note: This specification can give rise to different cacheability attributes between instruction and data accesses to the same location. Where this occurs, the measures for mismatch memory attributes described in section 3.24 must be followed to manage the corresponding loss of coherency.

3.22.1 Cache Disabled in AArch32

In the ARMv8 Architecture, for AArch32 operation, the same principles as described in section 3.22 apply for the equivalent cache enable bits in the SCTLR, HSCTLR, HCR and HCR2, namely:

SCTLR.C , SCTLR.I

HSCTLR.C, HSCTLR.I

HCR.DC

HCR2.CD, HCR2.ID

3.23 Pseudo-code for a translation table walk

The translation table walk pseudo-code in the LPAE is replaced by the following pseudo-code when using the ARMv8 translation system and translation is enabled

```
// PAMax is the implemented maximum size of the address space and is a constant for the
// implementation. It is encoded as the number of bits in the address
BaseAddress<47:0> = Zeros();
BaseFound = FALSE;
if Stage1Translation then
    if InEL3() then
        LookupSecure = TRUE;
        TSize = 64-UInt(TCR_EL3.T0SZ);
        if TSize > 48 then
            c = ConstrainUnpredictable();
```

```
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then
                TSize = 48;
            else // Constraint_FAULT
                TranslationFault(0);
        if TSize < 25 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then
                TSize = 25;
            else // Constraint_FAULT
                TranslationFault(0);
        LargeGrain = TCR_EL3.TG0 ==’01’;
        MidGrain = TCR_EL3.TG0 == ‘10’;
        BaseRegister = TTBR0_EL3;
        if TCR_EL3.TBI==’1’ then AddrTop =  55; else AddrTop = 63;
        BaseFound = IsZero(IA<AddrTop:TSize>);
        PS = TCR_EL3.PS;
        BigEndian = SCTLR_EL3.EE == ‘1’:
    elsif InEL2() then
        LookupSecure = FALSE;
        TSize = 64-UInt(TCR_EL2.T0SZ);
        if TSize > 48 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then
                TSize = 48;
            else // Constraint_FAULT
                TranslationFault(0);
        if TSize < 25 then
            c = ConstrainUnpredictable();
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then
                TSize = 25;
            else // Constraint_FAULT
                TranslationFault(0);
        LargeGrain = TCR_EL2.TG0 ==’01’;
        MidGrain = TCR_EL2.TG0 == ‘10’;
        BaseRegister = TTBR0_EL2;
        if TCR_EL2.TBI==’1’ then AddrTop =  55; else AddrTop = 63;
        BaseFound = IsZero(IA<AddrTop:TSize>);
        PS = TCR_EL2.PS;
        BigEndian = SCTLR_EL2.EE == ‘1’;
    else
        LookupSecure = SCR.NS == ‘0’;
        AddrTop = 63;
        if TCR_EL1.TBI1 == ‘1’ && IA<55> == ‘1’ then AddrTop = 55;
        if TCR_EL1.TBI0 == ‘1’ && IA<55> == ‘0’ then AddrTop = 55;
        if IA<AddrTop> == ‘1’ then
            TSize = 64-UInt(TCR_EL1.T1SZ);
            if TSize > 48 then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then
                    TSize = 48;
                else // Constraint_FAULT
                    TranslationFault(0);
            if TSize < 25 then
                c = ConstrainUnpredictable();
```

```
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then
                    TSize = 25;
                else // Constraint_FAULT
                    TranslationFault(0);
            LargeGrain = TCR_EL1.TG1 ==’11’;
            MidGrain = TCR_EL1.TG1 == ‘01’;
            BaseRegister = TTBR1_EL1;
            BaseFound = IsOnes(IA<AddrTop:TSize)>) && TCR_EL1.EPD1 == ‘0’;
        else
            TSize = 64-UInt(TCR_EL1.T0SZ);
            if TSize > 48 then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then
                    TSize = 48;
                else // Constraint_FAULT
                    TranslationFault(0);
            if TSize < 25 then
                c = ConstrainUnpredictable();
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then
                    TSize = 25;
                else // Constraint_FAULT
                    TranslationFault(0);
            LargeGrain = TCR_EL1.TG0 ==’01’
            MidGrain = TCR_EL1.TG0 == ‘10’;
            BaseRegister = TTBR0_EL1;
            BaseFound = IsZero(IA<AddrTop:TSize>) && TCR_EL1.EPD0 == ‘0’;

        PS = TCR_EL1.IPS;
        BigEndian = SCTLR_EL1.EE == ‘1’;

    if LargeGrain then
        GrainSize = 16;
        Stride = GrainSize – 3;
        if TSize > (GrainSize+2*Stride) then CurrentLevel = 1;
        elsif TSize > (GrainSize+Stride) then CurrentLevel = 2;
        else CurrentLevel = 3;
    elsif MidGrain then
        GrainSize = 14;
        Stride = GrainSize – 3;
        if TSize > (GrainSize+3*Stride) then CurrentLevel = 0;
        elsif TSize > (GrainSize+2*Stride) then CurrentLevel = 1;
        elsif TSize > (GrainSize+Stride) then CurrentLevel = 2;
        else CurrentLevel = 3;
    else
        GrainSize = 12;
        Stride = GrainSize-3;
        if TSize > (GrainSize+3*Stride) then CurrentLevel = 0;
        elsif TSize > (GrainSize+2*Stride) then CurrentLevel = 1;
        else CurrentLevel = 2;
else
    TSize = 64-UInt(VTCR_EL2.T0SZ);

    if TSize < 25 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_FORCE, Constraint_FAULT};
```

```
    if c == Constraint_FORCE then
        TSize = 25;
    else // Constraint_FAULT
        TranslationFault(0);

BaseRegister = VTTBR_EL2;
LargeGrain = VTCR_EL2.TG0 ==’01’;
MidGrain = VTCR_EL2.TG0 == ‘10’;
PS = VTCR_EL2.PS;
if FirstStageTranslationWasFromAArch32() then
    assert IA<47:40> == Zeros();
BaseFound = IsZero(IA<63:TSize>);
StartLevel = UInt(VTCR_EL2.SL0);
BigEndian = SCTLR_EL2.EE == ‘1’;


// Limits on IPA controls based on implemented PA size
if MidGrain then
    if PAMax < 41 && StartLevel == 2 then BaseFound = FALSE;
else
    if PAMax < 43 && StartLevel == 2 then BaseFound = FALSE;

// Force the TSize not to exceed:
//   the PAMax value, for an IPA from AArch64
//   the greater of the PAMax value and 40, for an IPA from AArch32
TSizeSLCheck = TSize;
if TSize > PAMax && ( !FirstStageTranslationWasFromAArch32() || TSize > 40) then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_FORCE, Constraint_FORCENoSLCheck, Constraint_FAULT};
    if c == Constraint_FORCE then
        TSize = PAMax;
        TSizeSLCheck = PAMax;
    elsif c == Constraint_FORCENoSLCheck then
        TSize = PAMax;
    else // Constraint_FAULT
        TranslationFault(0);


if StartLevel == 3 then BaseFound = FALSE;
if LargeGrain then
    CurrentLevel = 3- StartLevel;
    GrainSize = 16;
    Stride = GrainSize-3;
elsif MidGrain then
    CurrentLevel = 3 - StartLevel;
    GrainSize = 14;
    Stride = GrainSize-3;
else
    CurrentLevel = 2 - StartLevel;
    GrainSize = 12;
    Stride = GrainSize-3;

// Make illegal settings take a Translation Fault
// Translation Table of less than 2 entries or more than 16*(2^GrainSize/8) entries
// num entries in start table level =
// (Address Size)/((Address per level)^ Num of levels after start level + Size of Table)
//
```

```
    // Log2(Address per level) = Stride
    // Number of Levels after start level = 3-CurrentLevel
    // Log2(Address Size) = TSize;
    // Log2(Size of Table) = GrainSize
    // Upper bound check is
    // ((TSize) - Stride*(3-CurrentLevel) - GrainSize > GrainSize -3 + 4)
    // Lower bound check is
    // ((TSize) - Stride*(3-CurrentLevel) - GrainSize < 1
    if TSizeSLCheck >  Stride*(3-CurrentLevel) + 2*GrainSize + 1 then
        BaseFound = FALSE;
    if TSizeSLCheck <  Stride*(3-CurrentLevel) + GrainSize + 1  then
        BaseFound = FALSE;


// Bottom bound of the Base address is:
// log2(8 bytes per entry)+log2(num of entries in start table level )
// num entries in start table level =
// (Address Size)/((Address per level)^ Num of levels after start level  + Size of Table)
//
// Log2(Address per level) = Stride
// Number of Levels after start level = 3-CurrentLevel
// Log2(Address Size) = TSize;
// Log2(Size of Table) = GrainSize

BALowerBound = 3 + TSize - Stride*(3-CurrentLevel) - GrainSize ;
BaseAddress<47:0> = BaseRegister<47:BALowerBound>:Zeros(BALowerBound);
StartBit = TSize-1;


case PS of
    when '000'  CPAMax = 32;
    when '001'  CPAMax = 36;
    when '010'  CPAMax = 40;
    when '011'  CPAMax = 42;
    when '100'  CPAMax = 44;
    when '101'  CPAMax = 48;
    when '110'  CPAMax = 48;
    when '111'  CPAMax = 48;

if !BaseFound then
    TranslationFault(0);



if CPAMax > PAMax then
    CPAMax = PAMax;

if CPAMax != 48 then
    if BaseAddress<47:CPAMax> != Zeros() then AddressSizeFault(0);

FirstIteration = TRUE;
TableRW = TRUE;
TableUser = TRUE;
TableXN = FALSE;
TablePXN = FALSE;

repeat
    LookUpFinished = TRUE;
```

```
BlockTranslate = FALSE;

AddrSelectBottom = (3-CurrentLevel)*Stride + GrainSize;

if FirstIteration then
    AddrSelectTop = StartBit;
else
    AddrSelectTop = AddrSelectBottom + Stride - 1;

IASelect = ZeroExtend(IA<AddrSelectTop:AddrSelectBottom>:'000', PAMax);
LookupAddress = BaseAddress OR IASelect;
FirstIteration = FALSE;

// If there are two stages of translation, then the first stage table walk addresses
// are themselves subject to translation
if LastStageOfTranslation() then
    Descriptor = _Mem[LookupAddress,LookupSecure,8];
else
    LookupAddress = SecondStageTranslate(LookupAddress);
    Descriptor = _Mem[LookupAddress, FALSE, 8] ;

if BigEndian then
    Descriptor = BigEndianReverse(Descriptor, 8);

if Descriptor<0> == '0' then
    TranslationFault(CurrentLevel);
else
    if Descriptor<1> == '0' then
       if CurrentLevel == 3 then
           TranslationFault(CurrentLevel);
       else
           BlockTranslate = TRUE;
    else
        if CurrentLevel == 3 then
            BlockTranslate = TRUE;
        else // table translation
            BaseAddress = Descriptor<47:GrainSize>:Zeros(GrainSize);
            if CPAMax != 48 then
                if !IsZero(BaseAddress<47:CPAMax>) then AddressSizeFault(CurrentLevel);

            LookupSecure = LookupSecure && (Descriptor<63> == '0');
            TableRW = TableRW && (Descriptor<62> == '0');
            TableUser = TableUser && (Descriptor<61> == '0');
            TablePXN = TablePXN || (Descriptor<59> == '1');
            TableXN = TableXN || (Descriptor<60> == '1');
            LookUpFinished = FALSE;

if BlockTranslate then
    // not supporting extremely large blocks
    if LargeGrain then
        if CurrentLevel == 1 then TranslationFault(CurrentLevel);
    elsif MidGrain then
        if CurrentLevel < 2 then TranslationFault(CurrentLevel);
    else
        if CurrentLevel == 0 then TranslationFault(CurrentLevel);

    if LargeGrain then
        if CurrentLevel == 2 &&
```

```
                (TSize < 34 && Descriptor<52> == '1') then
                 IMPLEMENTATION DEFINED whether there is a TranslationFault(CurrentLevel);
         elsif MidGrain then
             if CurrentLevel == 2 && (TSize < 30 && Descriptor<52> == '1') then
                 IMPLEMENTATION DEFINED whether there is a TranslationFault(CurrentLevel);
        else
             if CurrentLevel == 1 &&
                 (TSize < 34 && Descriptor<52> == '1') then
                 IMPLEMENTATION DEFINED whether there is a TranslationFault(CurrentLevel);



         OutputAddress = Descriptor<47:AddrSelectBottom > : IA<AddressSelectBottom-1:0>;
         if CPAMax != 48 then
             if !IsZero(OutputAddress<47:CPAMax>) then AddressSizeFault(CurrentLevel);
         Attrs = Descriptor<54:52>: Descriptor<11:2>;



         if Stage1Translation then
             if TableXN then Attrs<12> = '1';
             if TablePXN then Attrs<11> = '1';
             if IsSecure() && !(LookupSecure) then Attrs<9> = '1';
             if !(TableRW) then Attrs<5> = '1';
             if !(TableUser) then Attrs<4> = '0';
             if !(LookupSecure) then Attrs<3> = '1';
    else
        CurrentLevel = CurrentLevel + 1;
until LookUpFinished
// final Attrs<> bus contains:
// 12:    XN
// 11:    PXN
// 10:    Contiguous Bit
// 9:     nG
// 8:     AccessFlag
// 7:6:   Shareability
// 5:4:   Stage1: AP[2:1]
// 5:4    Stage2: HAP[2:1]
// 3:0:   Stage2: Memory Type
// 3:     Stage1: Non-Secure
// 2:0:   Stage1: Memory Type Index
```

The pseudo-code for the translation table walk includes an addition pseudo-code function:

```
SecondStageTranslate(Address)
```

This function performs a second stage translation, from IPA to PA, of the supplied address. The second stage translation might hit in a TLB, or might involve a translation table walk using the algorithm described in this section.

When translation is disabled, the following pseudo-code applies for that stage of translation:

```
IAMax = 63;
if InEL3() then
    if TCR_EL3.TBI == '1' then IAMax = 55;
elsif InEL2() then
    if TCR_EL2.TBI == '1' then IAMax = 55;
else
```

```
    if (IA[55] == '1') && TCR_EL1.TBI1 == '1' then IAMax = 55;
    if (IA[55] == '0') && TCR_EL1.TBI0 == '1' then IAMax = 55;

if !IsZero(IA<IAMax:PAMax>) then
    AddressSizeFault(0);
else
    OutputAddress = IA<47:0>;
```

3.24 Mismatched Memory Attributes

A memory location is accessed with mismatched attributes when all memory accesses to the location do not have a common definition of one or more of the following properties for that memory location:

Shareability

Memory Type (Device, Normal)

Cacheability (excluding allocation hint) for the same level of cache (inner or outer)

Collectively these are referred as the memory attributes

In the ARMv8 Architecture, as a general principle, where a memory location is accessed with mismatched attributes the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory location might be lost; this is:
    - Reads of the memory location by one agent might not return the most recently written value to the memory location by the same agent.
    - Multiple writes to the memory location by one agent with different memory attributes might not be ordered in program order
- There might be a loss of coherency when multiple agents attempt to access a memory location
- There might be a loss of properties derived from the memory type
- If all load exclusive and store exclusive instructions executed across all threads do access a given memory location do not use consistent memory attributes, the exclusive monitor state becomes UNKNOWN.
- Bytes written without writeback cacheable attributes within the same writeback granule as bytes written with writeback cacheable attribute might have their values reverted to old values as a result of a cache writeback.

The following rules apply to the exact situations where the same physical memory location has different memory attributes associated with it:

1) The loss of properties associated with memory type refer to the following properties that are additional for Device memory types compared with Normal memory:
    - prohibition on read speculation
    - prohibition on re-ordering
    - prohibition on gathering
    - no Early acknowledgement hint

If the only mismatch in memory type associated with a memory location across all users of the memory location is between the different types of Device memory, then all accesses might take the properties of the weakest Device memory type.

2) If all aliases to a memory location with write permission use:

- a common definition of the shareability and cacheability attributes for a memory location, and
- Have the inner cacheability attribute be the same as the outer cacheability attribute

and all aliases to a memory location use a definition of the shareability attributes which encompass all the agents with permission to access the location, then any agent that reads that memory location using the same common definition of the shareability and cacheability attributes will access it coherently (to the extent required by that common definition of the memory attributes).

3) Where all accesses to a memory location treat the memory location as one of :

- Any Device memory type
- Normal Inner Non-cacheable, Outer Non-cacheable

The only permissible software visible effects can be one or more of the following

1) A potential loss of properties derived from the memory type when multiple agents attempt to access a memory location.
2) A potential re-ordering of memory transactions to the same memory location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Such loss of coherency or uniprocessor semantics can be avoided by inserting DMB barriers between accesses to the same memory location with different attributes.

Where there are losses of the uniprocessor semantics, ordering or coherency, the following approaches can be used:

1) Any loss of uniprocessor semantics, ordering or coherency within a shareability domain caused by the use of mismatched attributes where all accesses to the memory location have common shareability attributes can be avoided by employing the same software techniques that must applied for managing ordering and managing coherency in software for cacheable locations between agents in different shareability domains:

- invalidating (or cleaning) the location from the caches before the writing of a location not using the writeback attribute if any user to the location could have written to the location with the writeback attribute

  Note: this avoids any stale dirty overwriting the locations or overwriting of dirty bits in caches

- cleaning the location from the caches after the writing of a location with the writeback attribute

  Note: this ensures that the write is made visible to external memory before accessing it with a different attribute

- invalidating the location from caches before reading the location with the cacheable attribute

  Note: this ensures that any values held in caches reflect the last version made visible in external memory.

- Placing a DMB barrier applying to the common shareability of the accesses between any accesses to the same memory location that use different attributes

In all cases:

- the term "location" refers to any byte within the coherency granule being employed.
- Clean & invalidation can be used in place of either clean or invalidation.
- All cache maintenance and memory transactions must be completed or ordered by the use of barriers to ensure coherency.

Note: in keeping with all software management of coherency race conditions, where different bytes with the same location are written simultaneously by different agents can lead to the loss of data. This can occur when the invalidate/clean->write->clean sequence of one agent accessing a location overlaps with the invalidate/clean->write->clean sequence of another agent.

2) Any loss of uniprocessor semantics, ordering or coherency caused by the use of mismatched attributes where multiple cacheable accesses to the location could be made with different shareability attributes can only be avoided by requiring :

   - each core that accesses the location with cacheable attributes performs a clean and invalidation of the location with mismatched attributes.
   - Placing a DMB barrier applying of full shareability of the accesses between any accesses to the same memory location that use different attributes

   Note: in keeping with all software management of coherency race conditions, where different bytes with the same location are written simultaneously by different agents can lead to the loss of data. This can occur when the invalidate/clean->write->clean sequence of one agent accessing a location overlaps with the invalidate/clean->write->clean sequence of another agent.

3) Software is discouraged from using mismatched attributes, and it is recognised that implementations might not optimise the performance of systems that use mismatched attributes.

Note: in this text, the terms “location” and “memory location” are used synonymously, and refers to any byte within the coherency granule being employed

3.25 Exclusive Accesses Restrictions

In the A64 instruction, a single thread of execution where

- a LDXP instruction of two 32–bit quantities is followed by an STXR instruction of 1 64 bit quantity at the same address, or
- a LDXR instruction of one 64–bit quantity is followed by an STXP instruction of 2 32 bit quantities at the same address

can lead to UNPREDICTABLE behaviour. In this case:

- the STXP/STXR instruction might give rise to an external Data Abort
- The STXR/STXR generates an IMPLEMENTATION DEFINED MMU fault reported using the Fault Status code of ESR_ELx.DFSC = 110101
- the STXP/STXR instruction might always have an exclusive fail
- the STXP/STXR instruction might always have an exclusive pass
- the STXP/STXR instruction might have the same exclusive pass/fail behaviour that it would have had if the instruction was the same size and number of registers as the preceding LDXR/LDXP instruction.

4 ARMV8 MEMORY ORDER MODEL

4.1 Introduction

The ARMv8 memory order model is based on the ARMv7 memory order model as defined in [3]. The salient points of this are as follows:

For Normal Memory:

- Reads and writes to different locations can be observed in any order provided the following constraints are met:
    - Where an address dependency (as defined in [1]) exists between two reads or between a read and a write, then those memory accesses will be observed in program order by all observers within the shareability domain of the memory addresses being accessed

      Note: this is relaxed in the ARMv8 architecture for execution in an Exception Level using AArch64 where the second read is generated by a Load Non-temporal Pair instruction (see section 4.2.4).
    - Writes that that would not occur in a simple sequential execution of the program cannot be observed by other observers. This implies that where a control, address or data dependency exists between a read and a write, those memory accesses will be observed in program order by all observers within the shareability domain of the memory addresses being accessed
    - Ordering can be achieved by the use of DMB or DSB barriers according to the definitions of those barriers defined in [1]
- Reads and writes to the same location will be coherent within the shareability domain of the memory address being accessed
- Two reads to the same location by the same observer will be observed in program order by all observers within the shareability domain of the memory address being accessed
- Writes are not required to be multi-copy atomic; that is, in the absence of barriers, the observation of a store by one observer does not imply the observation of the store by any other observer

For Device memory – see section 3.16:

4.2 Memory order model for ARMv8

The ARMv8 architecture provides two new mechanisms set to provide memory order.

4.2.1 Load-Load/Store barriers

DMB and DSB are enhanced by a variant of the barrier that creates order between loads in Group A and loads or stores in Group B; that is

the required access type for accesses in Group A is load

the required access type for accesses in Group B is load or store

This is occasionally referred to as a Load-Load/Store barrier.

This applies in both AArch32 and AArch64 state.

4.2.2 Load with Acquire, Store with Release

A set of instructions with Acquire semantics for loads and Release semantics for stores are defined in the A64, T32 and A32 instruction sets. These instructions have, for all memory types, additional ordering requirements:

- A Store with Release followed by a Load with Acquire will be observed in program order by each observer within the shareability domain of both the memory address being accessed by the Store with Release and the memory address being accessed by the Load with Acquire
- Load with Acquire is a read where all reads and writes caused by loads and stores appearing in program order after the Load with Acquire will be observed as required by the shareability domains of the memory addresses being accessed by those loads and stores by each observer within the shareability domain of the memory address being accessed by the Load with Acquire after that observer observes the read caused by the Load with Acquire.
- The Load with Acquire places no additional ordering constraints on any loads or stores appearing before the Load with Acquire.
- Store with Release is a write where :
    - the reads and writes caused by loads and stores appearing in program order before the Store with Release
    - any writes that have been observed by the processor executing the Store with Release before the Store with Release

  will be observed as required by the shareability domains of the memory addresses being accessed by those loads and stores by each observer within the shareability domain of the memory address being accessed by the Store with Release before that observer observes the write caused by the Store with Release.
- The Store with Release places no additional ordering constraints on any loads or stores appearing after the Store with Release.
- All Stores with Release will be multi-copy atomic when observed with Load with Acquire instructions
- A Store with Release to an address in a memory-mapped peripheral of arbitrary system defined size using Device memory types will ensure that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed before the Store with Release will arrive at the memory-mapped peripheral before the memory access of the Store with Release.
- A Load with Acquire to an address in a memory-mapped peripheral of arbitrary system defined size using Device memory types will ensure that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed after the Load with Acquire will arrive at the memory-mapped peripheral after the memory access of the Load with Acquire.
- A memory access to a memory-mapped peripheral of arbitrary system defined size using Device memory types that are architecturally required to be ordered before the memory access of a Store with Release will arrive at the memory-mapped peripheral before any memory access to the same memory-mapped peripheral using Device memory types that are architectural required to be ordered after the memory access of a Load with Acquire to the same memory location as the Store with Release, where the Load with Acquire has observed the value stored by the Store with Release.

Load with Acquire and Store with Release, other than load-acquire exclusive pair and store-release exclusive pair, access only a single data element which is accessed single-copy atomically, and the address of the data object is required to be aligned to size of the element being accessed or else an Alignment Fault is generated.

Load-acquire exclusive pair and store-release exclusive pair, access two data elements. The address supplied to these instructions must be aligned to twice the element size being loaded or else an Alignment Fault is generated.

A Store-release exclusive instruction only has the Release semantics if the store is successful.

Note: In keeping with the ARMv7 architecture, an Observer is a master in the memory system that is capable of observing memory accesses. The definition of observe is as described in [3]

Note: Each Load-acquire exclusive and Store-release exclusive instructions is essentially a variant of the equivalent Load exclusive or Store exclusive instructions, and all usage restrictions and single-copy atomicity properties that apply the Load exclusive or Store exclusive instructions also apply to the Load-acquire exclusive or Store-release exclusive instructions.

4.2.3 Address Dependencies and order.

In the ARMv8 architecture for both AArch32 and AArch64, order is created between load and a subsequent memory transaction where there is a register data dependency between the data value returned from the load and the address used by the subsequent memory transaction.

An register data dependency exists between a first data value and a second data value when either

- the register, excluding the zero register (XZR/WZR), used to hold the first data value is used in the calculation of the second data value and the calculation between the first data value and the second data value does not consist of either:
    - A conditional branch whose condition is determined by the first data value
    - A conditional selection, move or computation whose condition is determined by the first data value, but the input data values for the selection, move or computation does not have a data dependency on the first data value
- there is a register data dependency between the first data value and a third data value, and between the third data value and the second data value

Note: this is a more formal definition of the concept of an address dependency used in ARMv7, and is intended to cover the same situations. In particular, a register data dependency can exist even if the value of the first data value is discarded as part of the calculation, as might be the case if it is ANDed with 0x0 or arithmetic using the first data value cancels out its contribution.

Thus, for example, both the following code snippets will exhibit order between the memory transactions

```
LDR X1, [X2]
AND X1, X1, XZR
LDR X4, [X3, X1]

LDR X1, [X2]
ADD X3, X3, X1
SUB X3, X3, X1
STR X4, [X3]
```

4.2.4 Address dependencies to Load Non-temporal Pair instructions

Where an address dependency (as defined in [1]) exists between two reads and the second read was generated by a Load Non-temporal Pair instruction, then, in the absence of any other barrier mechanism to achieve order, those memory accesses can be observed in any order by other observers within the shareability domain of the memory addresses being accessed

4.3 Single-copy Atomicity Considerations for ARMv8 Architecture

The ARMv7 rules for single-copy atomicity of memory accesses apply for all memory accesses generated from an Exception Level using AArch32.

For explicit memory accesses generated from an Exception Level using AArch64, the following rules apply:

- All reads generated by load instructions that load a single general purpose register, and that are aligned to the size of the read in that instruction are single-copy atomic
- All writes generated by store instructions that store a single general purpose register, and that are aligned to the size of the write in that instruction are single-copy atomic
- Reads to general purpose registers generated by load pair instructions that are aligned to size of the quantity being loaded into each register are treated as two single-copy atomic reads, one for each register being loaded.
- Writes from general purpose registers generated by store pair instructions that are aligned to size of the quantity being stored into each register are treated as two single-copy atomic writes, one for each register being stored.
- Load-exclusive Pair of two 32-bit quantities and Store-exclusive pair of two 32-bit quantities are single-copy atomic.
- The Load-exclusive Pair/Store-exclusive Pair instructions using two 64-bit quantities, when the Store-exclusive succeeds will cause a single-copy atomic update of the entire memory location being updated

  Note: the way to atomically load two 64-bit quantities is to perform a Load-exclusive Pair/Store-exclusive Pair sequence reading and writing the same value where the Store-exclusive succeeds, and use the read values.
- Each reads of a translation table entry caused by a translation table walks is single copy atomic.
- For the atomicity considerations of instruction fetches see section 4.3.1
- Reads to FP and SIMD registers of a single 64 bit or smaller quantity that is aligned to size of the quantity being loaded are treated as single-copy atomic reads.
- Writes from FP and SIMD registers of a single 64 bit or smaller quantity that is aligned to size of the quantity being stored are treated as single-copy atomic writes.
- Element or Structure Reads to FP and SIMD registers of 64 bit or smaller elements, where each element is aligned to size of the element being loaded, have each element treated as a single-copy atomic read.
- Element or Structure Writes from FP and SIMD registers of 64 bit or smaller elements, where each element is aligned to the size of the element being stores, have each element treated as a single-copy atomic store.

All other memory accesses are regarded as streams of accesses to bytes, and no atomicity is between accesses to different bytes is ensured by the architecture.

All accesses to any byte are single-copy atomic.

Note: For AArch64**,** no memory accesses involving FP and SIMD Registers, or from Data cache zero instructions have single-copy atomicity of any quantity greater than individual bytes.

Note: The Load-exclusive pair and store-exclusive pair instructions take an Alignment fault if they are not aligned to the size of the pair of elements being loaded/stored.

4.3.1 Concurrent Modification and Execution of Instructions

The ARMv8 architecture, for both AArch32 and AArch64, limits the set of instructions which can be executed by one thread of execution at they are being modified by a different thread of execution.

Except where the instruction before modification and the instruction after modification lie in the list below, concurrent modification and execution of instructions can lead to resulting instruction performing any behaviour that could be achieved by executing any sequence of instructions that could be executed the same Exception Level. Where the modification of the instruction is from one of these instructions to a different instruction, the behaviour will be consistent with either the original instruction being fetched, or the new instruction being fetched.

The list of instructions is:

For A64/A32 instruction

- Direct Branch and Direct Branch & Link
- NOP
- BKPT
- System calls SVC, HVC, SMC

For T32 instructions, 16-bit instructions for:

- Direct Branches
- NOP
- BKPT
- SVC

In addition having one thread of execution change an instruction between two conditional branches (including those with the always condition) where both the condition field and target is different between the two conditional branches when they are being executed by another thread of execution before the change of instruction has been synchronised, can lead to the execution of the instruction having either:

- The old condition being associated with the new target, or
- The new condition being associated with the old target

In addition, for T32 instructions:

Also

- The hw1 of a 32-bit BL immediate instruction can be modified to be another hw1 of the BL immediate instruction with a different immediate (ie the upper bits of the immediate can be changed).
- The hw1 of a 32-bit BLX immediate instruction can be modified to be another hw1 of the BLX immediate instruction with a different immediate (ie the upper bits of the immediate can be changed).
- The hw1 of a 32-bit BL immediate instruction or BLX immediate instruction can be modified to be a T32 16-bit instruction in the list above other than NOP (or vice versa)
- The hw2 of a 32-bit BL immediate instruction can be modified to be another hw2 of the BL immediate instruction with a different immediate (ie the lower bits of the immediate can be changed).

- The hw2 of a 32-bit BLX immediate instruction can be modified to be another hw2 of the BLX immediate instruction with a different immediate (ie the lower bits of the immediate can be changed).
- The hw2 of a 32-bit B immediate instruction without condition can be modified to be another hw2 of the B immediate instruction without condition with a different immediate (ie the lower bits of the immediate can be changed).
- The hw2 of a 32-bit B immediate instruction with condition can be modified to be another hw2 of the B immediate instruction with condition with a different immediate (ie the lower bits of the immediate can be changed).

The memory accesses caused by instruction fetches are not required to be observed in program order, unless they are separated by an ISB or other context synchronising event.

5 REPORTING MEMORY ATTRIBUTES ON INTERCONNECT

5.1 Introduction

The ARM architecture defines the architectural interface between the software and the processor hardware, and so the mechanisms by which different memory type and cacheability attributes are presented on the interface to an interconnect fabric such as AXI strictly lies outside the scope of the architecture.

To that end this section describes an approach that is strongly recommended for the interface between processor implementations and interconnect fabrics, rather than forming part of the ARMv8 architecture.

5.2 Effect of Micro-architectural choices on Memory attributes

In ARM implementations, there has been considerable variability in the presentation of such attributes on the interconnect fabric, particularly in cases where the processor implementation has not provided optimised support for a memory type. For example, an implementation might treat Write-through locations as Non-cacheable at some level of cache, since functionally this is consistent with the definition of Write-through, but the performance trade-off does not merit the hardware directly providing Write-through capability. However, in such an implementation, the memory attributes are not changed by these micro-architectural choices; the micro-architecture simply chooses different ways to handle the memory attributes.

As a result, it is strongly recommended that where any or all of the following memory attributes are presented on the interface between a processor and the interconnect fabric, the attributes that are presented are completely consistent with the attributes defined by the translation system:

- Memory type: Normal, Device
- Early Write Acknowledgement attribute
- Ordering requirements
- Shareability
- Cacheability, including where practical, the allocation hints

5.2.1 Effect when the cache is disabled

The effects of the caches being disabled, as described in section 3.22 are also reflected on the interconnect by the memory type used for memory transactions generated while the cache is disabled.