Learn the architecture - AArch64 memory management examples

Version 1.0

Non-Confidential

Issue 01

102416_0100_01_en

Learn the architecture - AArch64 memory management examples

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	27 June 2019	Non-Confidential	Initial release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

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

Contents

1. Overview

This set of examples shows how to set up the Memory Management Unit (MMU) in a bare metal environment. The examples walk through sets of code, building on the overall explanation of the MMU and translation process that the Memory management guide provides.

The examples are useful if you need to interact with the MMU at a low level, typically in a bare metal environment like bring-up test code. The examples do not cover MMU usage in an operating system.

The virtual address spaces that are constructed here are not intended to be realistic. Instead, the examples demonstrate different ways to configure the MMU. The final example gives a more realistic configuration for a simple bare metal system.

At the end of these examples, you will be able to write or modify a sequence to set up a simple virtual address space.

Before you begin

This set of examples requires you to be familiar with the principles of memory translation and the MMU controls in the processor. These subjects are covered in the Memory management guide.

The examples use A64 assembler. A basic understanding of A64 assembler helps you to follow the descriptions of the code. For an introduction to A64 see our Armv8-A Instruction Set Architecture guide.

Example platform

This set of examples are available as a separate download.

The examples were developed for the Base Platform model. Here are details of the physical address maps for the Base Platform model.

To build and run the examples, you need Arm Development Studio. If you do not have a copy of Arm Development Studio, download an evaluation copy.

Building and running the examples

The examples package includes a `ReadMe.txt` file. This file gives instructions for building and running the examples. The command line arguments to launch the simulator is different for each example. Refer to the `ReadMe.txt` file for more information.

2. Single-level table at EL3

The first example covers the simplest scenario: A single level of translation in the EL3 translation regime. We are going to flat map the virtual addresses. This means that the input virtual address and output physical address are the same for all translations. The MMU is only being used to control attributes and permissions.

In the examples package, the files are in: `<example dir>\el3_stage1_l1only\`

Specify the location of the translation table

The code for the example is in startup.S. Looking at this file, the MMU code starts at line 159. Here you see the first interesting piece of code:

```
// Set the Base address
// ---------------------
LDR   x0, =tt_l1_base  // Get address of level 1 for TTBR0_EL3
MSR   TTBR0_EL3, x0    // Set TTBR0_EL3 (NOTE: There is no TTBR1 at EL3)
```

This code loads the address of the memory that is allocated for the translation table, and then writes that address into the Translation Table Base Register (`TTBR0_ELx`). This register tells the processor where the first level table is located when a table walk is required.

The symbol name indicates that the register points to a level 1 table. We see later in this section Configure the translation regime how the starting level of translation is configured.

The memory for the table is allocated at the end of the file, as you see here:

```
.section TT,"ax"
.align 12

.global tt_l1_base
tt_l1_base:
.fill 4096 , 1 , 0
```

The code defines a sensible label (`tt_l1_base`) to let us refer to the allocated memory. The `fill` directive then allocates a 4KB block that is pre-filled with zeros. This is useful because a value of 0 in a translation table entry means Fault. The value 0 in a descriptor.

Translation tables must be size aligned. In this example, we have a full level 1 table. With a 4KB granule, a full level 1 table includes 512 entries. Each entry is 8 bytes. This means that the table is 4KB in size, and must start on a 4KB boundary. The `align` directive sets the alignment as a power of 2. In this case, the alignment is `2^12=4096`.

Initialize the MAIR

Going back to the code, let's look at the next step, which you see here:

```
// Set up memory attributes
 // -------------------------
 // This equates to:
```

```
// 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
// 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
// 2 = b00000000 = Device-nGnRnE
MOV   x0, #0x000000000000FF44
MSR   MAIR_EL3, x0
```

We learned in the AArch64 Memory model guide that the Type, either Normal or Device, is not directly encoded with the translation table entries for stage 1 tables. Instead, the table entries contain an index into the Memory Attribute Indirection Register (`MAIR_ELx`). Each 8-bit entry is set by software to specify a different memory Type. The example populates only the first three entries within the MAIR:

- [0] = Normal, Inner and Outer Non-cacheable
- [1] = Normal, Inner and Outer Cacheable, with write-back and read/write allocation
- [2] = Device_nGnRnE

For this simple example, these three types are enough. We do not use the other index values.

Which Type is specified in which MAIR index is important later when we create the translation table entries.

Configure the translation regime

The next step is to configure the translation regime, as you see here:

```
// Set up TCR_EL3
// ---------------
MOV   x0, #0x19            // T0SZ=0b011001 Limits VA space to 39 bits,
                          // translation starts @ l1
ORR   x0, x0, #(0x1 << 8)  // IGRN0=0b01  Walks to TTBR0 are Inner WB/WA
ORR   x0, x0, #(0x1 << 10) // OGRN0=0b01  Walks to TTBR0 are Outer WB/WA
ORR   x0, x0, #(0x3 << 12) // SH0=0b11   Inner Shareable
                          // TBI0=0b0   Top byte not ignored
                          // TG0=0b00   4KB granule
                          // IPS=0     32-bit PA space
MSR   TCR_EL3, x0
```

The Translation Control Register (`TCR_ELx`) configures many aspects of the translation regime, including:

TnSZ

Controls the size of the virtual address space that is being described

TGn

Sets the granule, which is the smallest describable block, for the translation regime

IGRNn/ORGNn/SH

Specifies the cacheability and shareability that the MMU should use for table walks

TBIn

To byte ignore. Setting this bit causes the top 8 bits of the virtual address to be ignored by the processor when performing virtual to physical translation. Allowing software to store something else in those bits instead. In this exercise, we do not use this feature, so we leave it disabled.

Note

For an example that shows when the TBI feature is used, see the description of Memory Tagging in the Providing protection for complex software guide.

The selected granule (`TG0`) for all the examples in this guide is 4KB. As described in the AArch64 Memory management guide, the granule determines the different page and block sizes that are used. With a 4KB granule, the options are:

- L0 table: 512GB per entry
- L1 tables: Each table covers 512GB, 1GB per entry
- L2 tables: Each table covers 1GB, 2MB per entry
- L3 tables: Each table covers 2MB, 4KB per entry

The size of the virtual address space is configured as `64 – TnSZ`. In this example, `64 – 0x19` gives 39 bits of virtual address space. This equates to 512GB (2^{39}), which means that the entire virtual address space is covered by a single L1 table. Therefore, our starting level of translation is level 1.

The next part of the example is shown here:

```
// Invalidate TLBs
// ----------------
TLBI  ALLE3
DSB   SY
ISB
```

The state of the Translation Lookaside Buffers (TLB) are not guaranteed at reset. Therefore, the example invalidates the TLB before enabling the MMU. The command (`TLBI ALLE3`) invalidates all cached translations for the EL3 translation regime, which is the translation regime that the example is configuring.

Generate the translation tables

The next step is to generate the tables in memory. This example creates a minimal set of entries, as you see in the following code:

```
LDR   x1, =tt_l1_base              // Address of L1 table

 // [0]: 0x0000,0000 - 0x3FFF,FFFF
 LDR   x0, =TT_S1_DEVICE_nGnRnE     // Entry template
                                    // AP=0, RW
                                    // Don't need to OR in address, as it is 0
 STR   x0, [x1]

 // [1]: 0x4000,0000 - 0x7FFF,FFFF
 LDR   x0, =TT_S1_DEVICE_nGnRnE     // Entry template
                                    // AP=0, RW
 ORR   x0, x0, #0x40000000            // 'OR' template with base physical address
 STR   x0, [x1, #8]

 // [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
 LDR   x0, =TT_S1_NORMAL_WBWA       // Entry template
 ORR   x0, x0, #TT_S1_INNER_SHARED  // 'OR' with inner-shareable attribute
```

```
                                                    // AP=0, RW
ORR   x0, x0, #0x80000000           // 'OR' template with base physical address
```

As described in the previous section Configure the translation regime, L1 is the first level of translation in this example. With a 4K granule, this means that each entry in the table covers 1GB of address space. The example only populates the first three entries, covering the first three 3GB of the virtual address space.

In the previous section Specify the location of the translation table, we showed how to allocate the memory for the translation table. A 4KB region that is prefilled with zeros was allocated with a `fill` directive. A value of zero corresponds to a Fault in the translation table. Therefore, all the entries that are not written are faulting entries.

Note

In a real system, software would typically fill the table with zeros at run-time, instead of relying on allocating them in the source. However, pre-allocating the zeros can speed up some simulations or emulations.

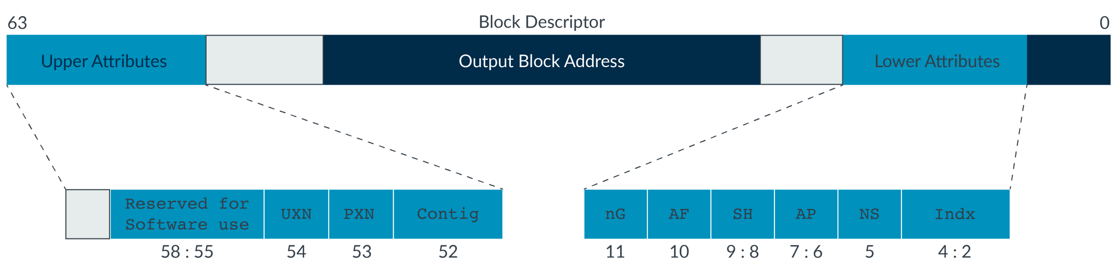
Understand how an entry is formed

The code uses symbols that are defined as templates at the start of the file. For example, `TT_S1_NORMAL_WBWA` is a template for a Normal, Write-back, Read/Write-allocate entry. The definition of this template is shown in this code:

```
.equ TT_S1_NORMAL_WBWA,    0x0000000000000405`
```

The following diagram shows the format of a stage 1 level 1 table entry:

Figure 2-1: Understand how an entry is formed diagram



Decoding the `TT_S1_NORMAL_WBWA` template gives:

- `Indx= b01`, take Type information from entry [1] in the `MAIR`
- `NS= b0`, output physical addresses are Secure
- `AP= b00`, address is readable and writeable

- `SH= b00`, Non-shareable
- `AF= b1`, Access Flag is pre-set. No Access Flag Fault is generated on access.
- `nG=` Not used at EL3
- `Contig = b0`, the entry is not part of a contiguous block
- `PXN= b0`, block is executable. This attribute is called `XN` at EL3.
- `UXN=` Not used at EL3

In the template, we see why knowing the configuration of the `MAIR` is important. The template relies on `MAIR` having entry [1] pre-set to Normal/Cacheable.

We want the region to be Inner-shareable, not Non-shareable as defined within the template. To fix this, the example combines the `TT_S1_NORMAL_WBWA` template with another template, `TT_S1_INNER_SHAREABLE`. This second template sets the correct value in the `SH` field.

Check your knowledge:

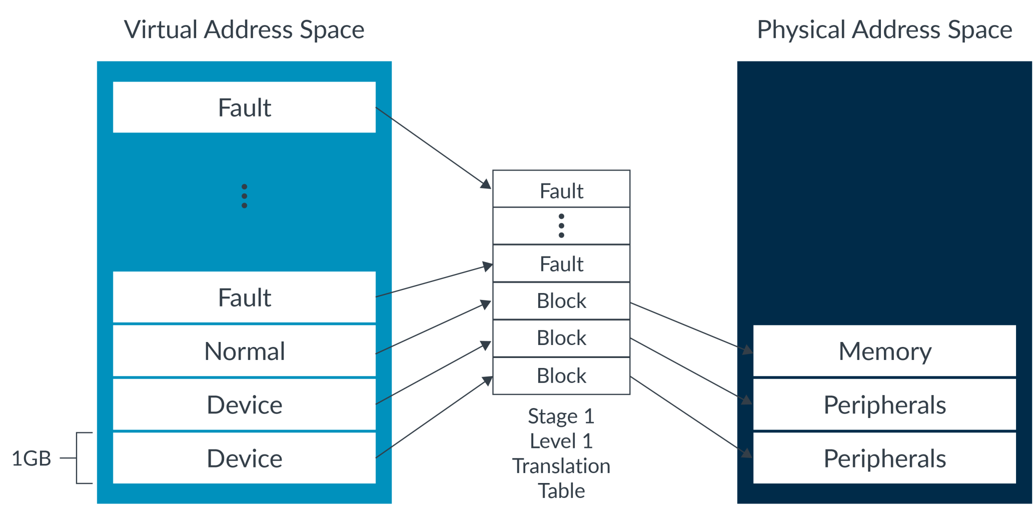
Question: Look at the other templates defined within the example. How would you modify the preceding example to map to a Non-secure physical address?

Answer: To map to a Non-secure Physical address requires setting the `NS` bit to 1. The example has a template for this, `TT_S1_NS`, which could be ORed like we did with for the Shareability attribute.

Overview of the configured virtual address space

With this set of translation table entries, the virtual address looks like what you see in the following diagram:

Figure 2-2: MMU virtual address space



Enable the MMU

At this point, the MMU is configured and the translation tables are created in memory. The next step is to enable the MMU, as you see in the following code:

```
// Enable MMU
// -----------
MOV   x0, #(1 << 0)      // M=1  Enable the stage 1 MMU
ORR   x0, x0, #(1 << 2)  // C=1  Enable data and unified caches
ORR   x0, x0, #(1 << 12) // I=1  Enable instruction fetches to allocate
                        //    into unified caches
                        // A=0  Strict alignment checking disabled
                        // SA=0 Stack alignment checking disabled
                        // WXN=0 Write permission does not imply XN
                        // EE=0 EL3 data accesses are little endian
MSR   SCTLR_EL3, x0
ISB
```

The example sets the `M`, `C`, and `I` bits in the System Control Register (`SCTLR_ELx`). Setting these bits enables the MMU and caches. The `ISB` after the write to the `SCTLR` ensures that the effect of enabling the MMU is visible to the next instruction.

The table walk

The examples in this exercise are developed to run on the Base Platform Fixed Virtual Platform (FVP). The Base Platform FVP is a model that is provided by Arm. FVP models trace the simulation, and provide detailed information on the execution of the simulated processor. The resulting trace is in the TARMAC format. Here is more information on TARMAC.

Tracing the entire example produces hundreds of lines of trace data. Instead, let's begin the trace at the point where the MMU is enabled, as you see here:

```
75 clk IT (75) 8000012c d51e1000 O EL3h_s : MSR   SCTLR_EL3,x0
75 clk R SCTLR_EL3 00000000:00001005
75 clk CACHE FVP_Base_AEMv8A_AEMv8A.cluster0.cpu0.l1dcache LINE 0100
 ALLOC 0x000080002000
75 clk CACHE FVP_Base_AEMv8A_AEMv8A.cluster0.l2_cache LINE 0800
 ALLOC 0x000080002000
75 clk TTW ITLB LPAE 1:1 000080002010 0000000080000705 : BLOCK ATTRIDX=1 NS=0 AP=0
 SH=3 AF=1 nG=0 16E=0 PXN=0 XN=0 ADDR=0x0000000080000000
75 clk TLB FILL FVP_Base_AEMv8A_AEMv8A.cluster0.cpu0.UTLB 1G 0x80000000 EL3_s,
 nG asid=0:0x0080000000 Normal InnerShareable Inner=WriteBackWriteAllocate
 Outer=WriteBackWriteAllocate xn=0 pxn=0 ContiguousHint=0
75 clk CACHE FVP_Base_AEMv8A_AEMv8A.cluster0.cpu0.l1icache LINE 0008
 ALLOC 0x000080000100
75 clk CACHE FVP_Base_AEMv8A_AEMv8A.cluster0.l2_cache LINE 0040
 ALLOC 0x000080000100
76 clk IT (76) 80000130 d5033fdf O EL3h_s : ISB
```

The trace is dense, so let's look at it one line at a time. This code shows the first section:

```
75 clk IT (75) 8000012c d51e1000 O EL3h_s : MSR   SCTLR_EL3,x0
75 clk R SCTLR_EL3 00000000:00001005
```

This code shows that the execution of the `MSR`, which enables the MMU. `0x8000_012C`, is the address of the instruction and `0xD51E_1000` is the opcode. The second line shows the value the instruction wrote to the register.

Note

By default, the trace shows the value that is written to the register, not the new value of the register. In many cases, but not all cases, the new value is the written value. For example, if the register includes read-only fields, the new value is not the written value.

Because this instruction enabled the MMU, the processor needs to implement a table walk for the page containing the next instruction. First, the trace shows this code:

```
75 clk CACHE FVP_Base_AEMv8A_AEMv8A.cluster0.cpu0.l1dcache LINE 0100
 ALLOC 0x000080002000
75 clk CACHE FVP_Base_AEMv8A_AEMv8A.cluster0.l2_cache LINE 0800
 ALLOC 0x000080002000
```

When we configured `TCR_EL3`, we configured the MMU to use cacheable accesses for the table walk. The preceding two lines show the cache line that contains the required table entry being fetched into the cache. Once the line is returned from the memory, the descriptor can be interpreted by the MMU. This interpretation is shown in the next line of code, as you can see here:

```
75 clk TTW ITLB LPAE 1:1 000080002010 0000000080000705 : BLOCK ATTRIDX=1 NS=0 AP=0
 SH=3 AF=1 nG=0 16E=0 PXN=0 XN=0 ADDR=0x0000000080000000
```

The preceding trace entry shows the MMU processing the table entry. This entry shows us the following things:

- `TTW`= Table walk
- `ITLB`= Table walk for the instruction interface. The I is for instruction.
- `1:1`= Stage 1, level 1 table entry
- `0x80002010`= Address the entry was fetched from
- `0x0000000080000705`= Entry returned from memory system
- `BLOCK`= The entry is a Block entry
- `ATTRIDX=1`= Uses MAIR entry 1.
- `NS`= Output physical address is Secure
- `AP`= Access permission bits
- `SH`= Shareability bits

Finally, the trace shows the TLB record that is being generated, as you see in this code:

```
75 clk TLB FILL FVP_Base_AEMv8A_AEMv8A.cluster0.cpu0.UTLB 1G 0x80000000 EL3_s,
 nG asid=0:0x0080000000 Normal InnerShareable Inner=WriteBackWriteAllocate
 Outer=WriteBackWriteAllocate xn=0 pxn=0 ContiguousHint=0
```

The trace shows that the TLB entry is created as follows:

- 1GB block
- `PA:0x8000_0000`
- `VA:0x8000_0000`, with ASID 0, although ASIDs are not used at EL3
- Translation regime: EL3
- Normal, Inner Shareable, Write-Back, Write-Allocate
- Execute-able

Check your knowledge: Look at the preceding code and find where all the settings that are shown in the trace come from.

3. EL3 Multiple levels of table

This section of the guide walks through an example with two levels of translation. The single-level table at EL3 example used a single level 1 table. This means that all mappings were using 1GB blocks. For a simple system, this kind of course grain mapping is appropriate. However, many systems need more fine grain mappings, which is achieved by using multiple levels of tables.

In the examples package, the files are in:

```
<example dir>\el3_stage1_l1andl2\
```

Generate the level 1 table

The first steps of this example are the same as the single-level table at EL3 example. As in the single-level table at EL3 example, the `MAIR` is populated with the three Types that the example uses. The `TCR` is configured to select a 4KB granule and a starting level of translation is L1.

This example differs from the single-level table at EL3 example at the point of table generation. The following code generates the L1 table:

```
//
 // Generate L1 table
 //

 LDR   x1, =tt_l1_base           // Address of L1 table

 // [0]: 0x0000,0000 - 0x3FFF,FFFF
 LDR   x0, =TT_S1_DEVICE_nGnRnE  // Entry template
                                 // AP=0, RW
                                 // Don't need to OR in address, as it is 0
 STR   x0, [x1]

 // [1]: 0x4000,0000 - 0x7FFF,FFFF
 LDR   x0, =TT_S1_DEVICE_nGnRnE  // Entry template
                                 // AP=0, RW
 ORR   x0, x0, #0x40000000         // 'OR' template with base physical address
 STR   x0, [x1, #8]

 // [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
 LDR   x2, =tt_l2_base           // Get address of L2 table
 LDR   x0, =TT_S1_TABLE          // Entry template for pointer to next level table
 ORR   x0, x0, x2                // Combine template with L2 table Base address
 STR   x0, [x1, #16]             // Write template into entry table[2]
```

As in the single-level table at EL3 example, this example uses templates that are defined at the start of the file to create each entry. The first two entries are the same as the single-level table at EL3 example. These entries create two 1GB block mappings with Device type.

The third entry is different. Instead of mapping a 1GB block, this entry points to a level 2 table. This level 2 table divides the 1GB block in to 512 2MB blocks. To do this, the example uses another template, `TT_S1_TABLE`.

Where does the address for the level 2 table (`tt_l2_base`) come from?

Like the level 1 table, the example uses a `fill` directive to allocate a 4KB region of memory to hold the table. Here is the code that allocates the level 1 and 2 tables:

```
 .align 12

.global tt_l1_base
tt_l1_base:
 .fill 4096 , 1 , 0

 .global tt_l2_base
tt_l2_base:
 .fill 4096 , 1 , 0
```

The example uses the `fill` directive to pre-fill the memory allocated for the tables with zeros. A value of zeros gives a fault. In a real system, code writes these zeros at run-time. However, pre-filling the zeros is useful for test code, to reduce simulation or emulation time.

Note

The level 2 table also needs to be size aligned, which is 4KB aligned in this example. In this example, the table is aligned because it immediately follows another size-aligned 4KB structure.

Generate the level 2 tables

The following code generates the level table 2:

```
//
// Generate L2 table
//
…
LDR   x0, =tt_l2_base             // Address of first L2 table

// The L2 table covers the address range:
// 0x8000_0000 - 0xBFFF_FFFF
//
// This example only populates entry 0, which covers:
// 0x8000_0000 - 0x801F_FFFF

LDR   x1, =tt_l2_base             // Address of L1 table

LDR   x0, =TT_S1_NORMAL_WBWA      // Entry template
ORR   x0, x0, #TT_S1_INNER_SHARED // 'OR' with inner-shareable attribute
                                  // AP=0, RW
ORR   x0, x0, #0x80000000           // 'OR' template with base physical address
STR   x0, [x1]

DSB   SY
```

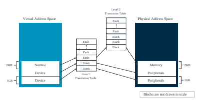
The level 2 table in this example covers the virtual address range `0x8000_0000` to `0xBFFF_FFFF`, which is the third gigabyte of the virtual address space. The table contains 512 entries, and each entry describes 2MB of virtual address space.

The example populates the first entry of the level 2 table with an entry for a Normal/Cacheable block. This entry corresponds to the first 2MB of address space that is covered by the L2 table, `0x8000_0000` to `0x801F_FFFF`.

Overview of the configured virtual address space

The result of the translation tables is shown in the following diagram:

Figure 3-1: Overview of the configured virtual address space



The table walk

Like in the single-level table at EL3 example, let's look at the TARMAC trace showing the first table walk after the MMU is enabled:

```
81 clk TTW ITLB LPAE 1:1 000080002010 0000000080003003 : TABLE PXN=0 XN=0 AP=0 NS=0
 ADDR=0x0000000080003000
81 clk TTW ITLB LPAE 1:2 000080003000 0000000080000705 : BLOCK ATTRIDX=1 NS=0 AP=0
 SH=3 AF=1 nG=0 16E=0 PXN=0 XN=0 ADDR=0x0000000080000000
81 clk TLB FILL FVP_Base_AEMv8A_AEMv8A.cluster0.cpu0.UTLB 2M 0x80000000 EL3_s,
 nG asid=0:0x0080000000 Normal InnerShareable Inner=WriteBackWriteAllocate
 Outer=WriteBackWriteAllocate xn=0 pxn=0 ContiguousHint=0
```

In this example, unlike the single-level table at EL3 example, there are now two ITLB lines. The first reports 1:1, which refers to a stage 1 table entry. The trace shows that level 1 table entry fetched from memory points to a level 2 table.

The next line in the trace reports 1:2, which refers to a stage 1 level 2 table entry. This is entry is a Block descriptor, like we saw in the single-level table at EL3 example.

The final line shows the TLB entry being recorded. Because the translation came from a level 2 block, this time the size of the entry is recorded as 2MB.

Check your knowledge

Question: This example shows a stage 1 translation, with two levels of table. What would you expect to see the trace for a stage 2 level 3 entry?

Answer: 2:3

4. EL1 Single-level table

In this section of the guide, we recreate the single-level table at EL3 example, this time running at EL1 in Non-secure state. The single-level table at EL3 and multiple-level table examples run at EL3.

In the examples package, the files are in `\el1_stage1\`

Enter NS.EL1

The Processing Element (PE) always comes out of reset in the highest implemented Exception level. For our test system, the highest implemented Exception level is EL3. The example therefore needs to include code to switch from EL3 to EL1. Before changing the Exception level, we need to carry out some configuration at EL3.

The first register the example configures is the Secure Configuration Register (`SCR_EL3`), as you see in the following code:

```
// Configure SCR_EL3
// ------------------
MOV   x0, #1                // NS=1
ORR   x0, x0, #(1 << 1)     // IRQ=1 IRQs routed to EL3
ORR   x0, x0, #(1 << 2)     // FIQ=1 FIQs routed to EL3
ORR   x0, x0, #(1 << 3)     // EA=1  SError routed to EL3
ORR   x0, x0, #(1 << 8)     // HCE=1 HVC instructions are enabled
ORR   x0, x0, #(1 << 10)    // RW=1  Next EL down uses AArch64
ORR   x0, x0, #(1 << 11)    // ST=1  Secure EL1 can access timers
MSR   SCR_EL3, x0
```

There are many settings in `SCR_EL3`. Two settings are important for this example:

`NS`

Controls whether lower Exception levels are Secure or Non-secure

`RW`

Controls whether the next Exception level uses AArch64 or AArch32 The example sets both bits. This means that lower Exception levels are Non-secure and that EL2 uses AArch64.

We also need to configure the Hypervisor Configuration Register (`HCR_EL2`). In a real software stack, code running in EL2 would do this. However, to keep the example simple, these registers are programmed from EL3 instead. The code to configure `HCR_EL2` is shown here:

```
// Configure HCR_EL2
// ------------------
ORR   w0, wzr, #(1 << 3)       // FMO=1
ORR   x0, x0, #(1 << 4)        // IMO=1
ORR   x0, x0, #(1 << 31)       // RW=1     NS.EL1 is AArch64
                               // TGE=0     Entry to NS.EL1 is possible
                               // VM=0     Stage 2 MMU disabled
MSR   HCR_EL2, x0
```

Like with the SCR_EL3, HCR_EL2 contains many controls. For this example, like with the single-level table at EL3 and multiple-level table examples, we are most interested in two settings:

RW

Controls whether EL1 uses AArch64 or AArch32

VM

Enables/disables stage 2 translation at EL1 and EL0

The example disables stage 2 and sets EL1 to use AArch64.

There are other settings in EL2 registers that we need to configure before switching to EL2. These are shown in the following code:

```
// Set up VMPIDR_EL2/VPIDR_EL1
// ---------------------------
MRS   x0, MIDR_EL1
MSR   VPIDR_EL2, x0
MRS   x0, MPIDR_EL1
MSR   VMPIDR_EL2, x0


// Set VMID
// ---------
// Although we are not using stage 2 translation, NS.EL1 still cares
// about the VMID
MSR   VTTBR_EL2, xzr


// Set SCTLRs for EL1/2 to safe values
// ------------------------------------
MSR   SCTLR_EL2, xzr
MSR   SCTLR_EL1, xzr
```

Reads of the MPIDR_EL1 and MIDR_EL2 registers at NS.EL1 return virtual values. The registers which hold these virtual values, VMPIDR_EL2 and VPIDR_EL2, do not have defined reset values. Software should initialize these registers before entering EL1 for the first time. For this example, we are not using virtualization. This means that we can copy the physical values.

Even though stage 2 is disabled, EL1 still uses a Virtual Machine Identifier (VMID). It is good practice to set this to a known value, before entering EL1. This is particularly important when working in a multi-core environment. All the PEs that run within the same NS.EL0/1 translation regime need to use the same VMID.

Finally, there are separate System Control Registers (SCTLR_ELx) for EL3, EL2, and EL1. Only the SCTLR_ELx for the highest implemented Exception level has a known reset value. Software must set the SCTLR_ELx registers for lower Exception levels to known or safe values before entering those Exception levels. This example sets them to 0, which ensures that the MMU for that Exception level is disabled.

Now that the minimum configuration is performed, control pass to NS.EL1, as you see in the following code:

```
 ADR   x0, el1_entry
 MSR   ELR_EL3, x0

 LDR   x0, =AArch64_EL1_SP1
 MSR   spsr_el3, x0

 ERET

// ---------------------------------------------------------------
// Enter EL1
// ---------------------------------------------------------------

el1_entry:
```

The only way to move to a lower Exception level is to perform an exception return. Normally, the exception return information is generated as part of taking an exception.. Because this is the part of boot the example instead creates the required information and populates the registers. The example sets the Saved Processor State Register (`SPSR_ELx`) to indicate EL1 using AArch64, and the Exception Link Register (`ELR_ELx`) to point to the start of the EL1 code. The `ERET` instruction then performs the Exception return.

Configure the MMU at EL1

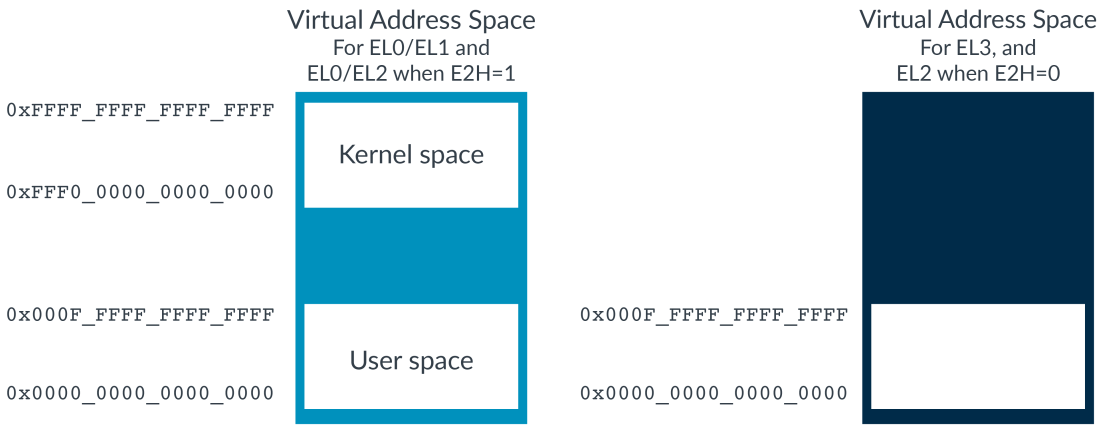
Now that execution has entered EL1, the next step is to configure the MMU. The steps are the same as in the first example, but this time we use `_EL1` registers instead of `_EL3` registers. For examples, let's look at the following code:

```
 // Set the Base address
 // ---------------------
 LDR   x0, =tt_l1_base
 MSR   TTBR0_EL1, x0


 // Set up memory attributes
 // -------------------------
 // This equates to:
 // 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
 // 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
 // 2 = b00000000 = Device-nGnRnE
 MOV   x0, #0x000000000000FF44
 MSR   MAIR_EL1, x0
```

Unlike EL3, in EL1/0 there are two virtual address regions: one at the bottom of the address space and another at the top of the address space. This is illustrated in the following diagram:

Figure 4-1: EL1 configuration diagram



By convention, the lower address region is called User space and the upper region is called Kernel space. However, this is only a convention and you will not see these names used in the Architecture Reference Manual.

For this example, we only configure the lower region. We disable the upper region, using a control in the Translation Control Register (`TCR_ELx`). The code for this is shown here:

```
// Set up TCR_EL1
// ---------------
MOV   x0, #0x19              // T0SZ=0b011001 Limits VA space to 39 bits
ORR   x0, x0, #(0x1 << 8)   // IGRN0=0b01  Walks to TTBR0 are Inner WB/WA
ORR   x0, x0, #(0x1 << 10)  // OGRN0=0b01  Walks to TTBR0 are Outer WB/WA
ORR   x0, x0, #(0x3 << 12)  // SH0=0b11   Inner Shareable
ORR   x0, x0, #(0x1 << 23)  // EPD1=0b1   Disable table walks from TTBR1
                            // TBI0=0b0
                            // TG0=0b00   4KB granule for TTBR0
                            // A1=0     TTBR0 contains the ASID
                            // AS=0     8-bit ASID
                            // IPS=0     32-bit IPA space
MSR   TCR_EL1, x0
```

The `EPDn` bits enable or disable walks from the lower region (`EPD0`) and the upper region (`EPD1`). The example only configures the lower region (`EPD1==0`). Walks to the upper region are disabled (`EPD1==1`). Because table walks to the upper region are disabled, the example does not need to provide a table pointer in `TTBR1_EL1`.

The code to generate the translation tables is unchanged from the single-level table at EL3 example.

Non-secure translation regimes

The example in this section of the guide runs in `NS.EL1`. The translation tables in this example are identical to the translation tables at EL3 in the single-level table at EL3 example. Does this mean that the resulting mappings are the same?

The answer is no. There is an important difference between Secure and Non-secure translation regimes. A Secure translation regime maps virtual addresses to Secure or Non-secure physical addresses that are controlled by the `NS` bit in the table entries. A Non-secure translation regime only maps to Non-secure physical addresses. The `NS` bit in the table entries is ignored.

In the single-level table at EL3 example and this example, the `NS` bit in the table entries is b0 (Secure). At EL3, this causes the outputted address to be Secure. In `NS.EL1` the `NS` bit is ignored, and the outputted address is Non-secure.

Note

On a real system, it is very unlikely that you could run both these two examples, because the memory at physical address `0x8000_0000` would either be Secure or Non-secure. The memory system of a real system would not allow both kinds of access to the memory. However, the FVP model that is used for these examples allows us to control which types of accesses are permitted to DRAM using model parameters.

The table walk

Tracing this example gives a very similar result to the single-level table at EL3 example, as you see in the following code:

```
93 clk IT (93) 80000174 d5181000 O EL1h_n : MSR   SCTLR_EL1,x0
93 clk R SCTLR_EL1 00000000:00001005
93 clk CACHE FVP_Base_AEMv8A_AEMv8A.cluster0.cpu0.l1dcache LINE 0100
 ALLOC 0x000080002000_NS
93 clk CACHE FVP_Base_AEMv8A_AEMv8A.cluster0.l2_cache LINE 0800
 ALLOC 0x000080002000_NS
93 clk TTW ITLB LPAE 1:1 000080002010 0000000080000705 : BLOCK ATTRIDX=1 NS=0 AP=0
 SH=3 AF=1 nG=0 16E=0 PXN=0 XN=0 ADDR=0x0000000080000000
93 clk TLB FILL FVP_Base_AEMv8A_AEMv8A.cluster0.cpu0.UTLB 1G 0x80000000_NS EL1_n
 vmid=0:0x0080000000_NS Normal InnerShareable Inner=WriteBackWriteAllocate
 Outer=WriteBackWriteAllocate xn=0 pxn=0 ContiguousHint=0
93 clk CACHE FVP_Base_AEMv8A_AEMv8A.cluster0.cpu0.l1icache LINE 000a
 ALLOC 0x000080000140_NS
93 clk CACHE FVP_Base_AEMv8A_AEMv8A.cluster0.l2_cache LINE 0050
 ALLOC 0x000080000140_NS
```

There are, however, some important differences between the single-level table at EL3 example and this example, starting with the `MSR`, as this code shows:

```
93 clk IT (93) 80000174 d5181000 O EL1h_n : MSR   SCTLR_EL1,x0
```

TARMAC records the Exception level and Security state that instructions were executed in. In the previous examples, this was `EL3h_s`, but this example reports `EL1h_n`. This means that EL1 is in Non-secure state.

The created TLB entry is also different, as this code shows:

```
93 clk TLB FILL FVP_Base_AEMv8A_AEMv8A.cluster0.cpu0.UTLB 1G 0x80000000_NS EL1_n
 vmid=0:0x0080000000_NS Normal InnerShareable Inner=WriteBackWriteAllocate
 Outer=WriteBackWriteAllocate xn=0 pxn=0 ContiguousHint=0
```

The trace shows the TLB entry recording the translation regime (`EL1_n`). The trace also shows the VMID being stored (`vmid=0`). As explained in Enter NS.EL1, even when stage 2 translation is disabled, the VMID is still recorded for the Non-secure EL1 translation regime.

5. A more complicated virtual address space

The example in this section shows a more complex set of mappings. Like with the single-level table at EL3 and multiple level table examples, this example runs at EL3.

So far, the examples that we have seen map a small number of blocks. For a simple test image, this might be enough. However, in a larger system we want to map more of the resources of the systems, and map these resources at a finer grain.

In the examples package, the files are in:

```
<example dir>\el3_stage1_full_mem_map\.
```

System physical address map

The example targets the Base Platform Model from Arm. The address map of the Base Platform model it typical of a modern Arm-based SoC and is summarized in the following table:

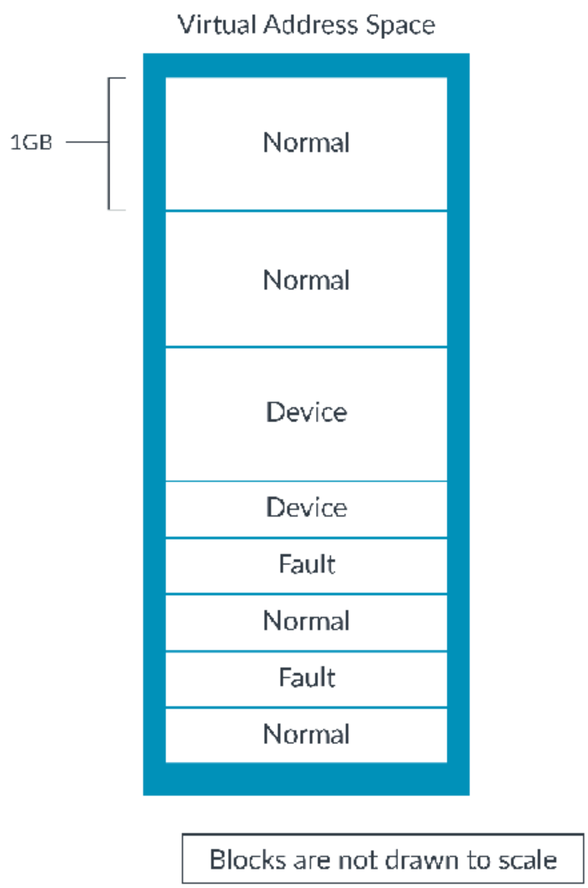
Physical address	Component	Secure or Non-secure	Attributes that the example assigns
0x0000_0000 to 0x03FF_FFFF	Trusted ROM	Secure	Normal, Cacheable, Shareable, Read-only, Executable
0x0400_0000 to 0x05FF_FFFF	–	–	Fault
0x0600_0000 to 0x07FF_FFFF	Trusted DRAM	Secure	Normal, Cacheable, Shareable, Read/Write, Executable
0x0800_0000 to 0x0FFF_FFFF	Flash	Non-secure	Normal, Cacheable, Shareable, Read-only, XN*
0x1000_0000 to 0x19FF_FFFF	–	–	Fault
0x1A00_0000 to 0x7FFF_FFFF	Peripherals	Secure	Device-nGnRnE, Read/Write, XN
0x8000_0000 to 0xFFFF_FFFF	DRAM	Non-secure**	Normal, Cacheable, Shareable, Read/Write, XN*

*These are Non-secure memories. This example runs in EL3, which is part of Secure state. Typically, we want to prevent execution from Non-secure locations while in Secure state. This can also be prevented using `SCR_EL3.CIF`.

**The FVP model that we are using can be configured so that the DRAM is either Non-secure, or both Secure and Non-secure. In the previous examples, we configured the model to allow both. In this example, we configure the model to allow only Non-secure accesses, which is a more realistic configuration.

The following diagram shows what the preceding memory map might look like as a set of MMU mappings:

Figure 5-1: Memory map diagram



Set the first level of translation

The address map that we just described is 4GB (`0x0..0xFFFF_FFFF`), or 32-bits, in total. As described in Configure the translation regime, the size of the virtual address space is specified as `64-T0SZ`. The example sets `TCR_EL3.T0SZ` to 32, to give a 32-bit virtual address space. This is shown in the following code:

```
// Set up TCR_EL3
// ---------------
MOV     x0, #32                // T0SZ=0b011001 Limits VA space to 32 bits
ORR     x0, x0, #(0x1 << 8)     // IGRN0=0b01    Walks to TTBR0 are Inner WB/WA
ORR     x0, x0, #(0x1 << 10)    // OGRN0=0b01    Walks to TTBR0 are Outer WB/WA
ORR     x0, x0, #(0x3 << 12)    // SH0=0b11      Inner Shareable
                               // TBI0=0b0      Top byte not ignored
                               // TG0=0b00      4KB granule
                               // IPS=0        32-bit PA space
MSR     TCR_EL3, x0
```

With a 4KB granule, 4GB is too big for a single level 2 table. Remember that each level 2 table covers 1GB. Therefore, the starting level of translation is level 1. Each entry in the level 1 table covers 1GB of virtual address space. Because we have configured a 4GB address space, the level 1 table in this example only requires four entries. We do not need to provide memory, or values, for the other entries.

In the previous examples, we always had a full (512 entry) level 1 table. A translation table must be size aligned. In the previous examples, this meant that the table had to 4KB aligned, with 512 entries * 8 bytes per entry. In this example, the alignment requirement is only 32-byte aligned, with 4 entries * 8 bytes per entry. However, for simplicity the example still allocates a 4KB aligned table.

The code below shows the reserving of memory for the level 1 and level 2 tables:

```
.align 12

  .global tt_l1_base
tt_l1_base:
  .fill 32 , 1 , 0

  .align 12
  .global tt_l2_base
tt_l2_base:
  .fill 4096 , 1 , 0
```

Level 1 table

Here is the level 1 table that is generated for the example:

```
  //
  // Generate L1 table
  //

  LDR      x1, =tt_l1_base             // Address of L1 table

  // [0]: 0x0000,0000 - 0x3FFF,FFFF
  LDR      x2, =tt_l2_base             // Get address of L2 table
  LDR      x0, =TT_S1_TABLE            // Entry template for pointer to next level
 table
  ORR      x0, x0, x2                  // Combine template with L2 table Base
 address
  STR      x0, [x1]

  // [1]: 0x4000,0000 - 0x7FFF,FFFF
  LDR      x0, =TT_S1_DEVICE_nGnRnE    // Entry template
                                       // AP=0, RW
  ORR      x0, x0, #0x40000000           // 'OR' template with base physical address
  STR      x0, [x1, #8]

  // [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
  LDR      x0, =TT_S1_NORMAL_WBWA      // Entry template
  ORR      x0, x0, #TT_S1_INNER_SHARED // 'OR' with inner-shareable attribute
  ORR      x0, x0, #TT_S1_NS           // 'OR' with NS==1
  ORR      x0, x0, #TT_S1_PXN          // 'OR' with XN==1
                                       // AP=0, RW
  ORR      x0, x0, #0x80000000           // 'OR' template with base physical address
  STR      x0, [x1, #16]

  // [3]: 0xC000,0000 - 0xFFFF,FFFF (DRAM on the VE and Base Platform)
  LDR      x0, =TT_S1_NORMAL_WBWA      // Entry template
  ORR      x0, x0, #TT_S1_INNER_SHARED // 'OR' with inner-shareable attribute
  ORR      x0, x0, #TT_S1_NS           // 'OR' with NS==1
  ORR      x0, x0, #TT_S1_PXN          // 'OR' with XN==1
                                       // AP=0, RW
  ORR      x0, x0, #0xC0000000           // 'OR' template with base physical address
```

```
  STR      x0, [x1, #24]
```

The level 1 table has only has only four entries. This is because we have a 4GB virtual address space and each entry in this table covers 1GB.

For the first 1GB of the virtual address space, `0x0000_0000` to `0x3FFF_FFFF`, we need to describe multiple regions. Therefore, the entry in the level 1 table must point to level 2 table, where we make more granular mappings.

For the next three entries, we use 1GB blocks. Although we can use smaller blocks, larger blocks are more efficient. This is because larger blocks require less memory for the translation tables, and it means that the TLB entries covers more addresses.

Level 2 table

Here is the level 2 table that is created for the first 1GB of the virtual address space:

```
//
  // Generate L2 table
  //
  …
  LDR      x1, =tt_l2_base              // Address of L1 table

  // [0..31]: 0x0000,0000 - 0x03FF,FFFF (Trusted Boot ROM)
  LDR      x0, =TT_S1_NORMAL_WBWA       // Entry template
  ORR      x0, x0, #TT_S1_INNER_SHARED  // 'OR' with inner-shareable attribute
  ORR      x0, x0, #TT_S1_PRIV_RO       // 'OR' in Read-only
  ORR      x0, x0, xzr                  // 'OR' template with base physical address
  MOV      x2, #32
1:
  STR      x0, [x1], #8
  ADD      x0, x0, #0x200000            // Increment the physical address field
  SUB      x2, x2, #1
  CBNZ     x2, 1b

  // [32..47]: 0x0400,0000 - 0x05FF,FFFF (Fault)
  LDR      x0, =TT_S1_FAULT             // Entry template
  ORR      x0, x0, #0x04000000          // 'OR' template with base physical
 address
  MOV      x2, #16
1:
  STR      x0, [x1], #8
  ADD      x0, x0, #0x200000            // Increment the physical address field
  SUB      x2, x2, #1
  CBNZ     x2, 1b


  // [48..63]: 0x0600,0000 - 0x07FF,FFFF (Trusted DRAM)
  LDR      x0, =TT_S1_NORMAL_WBWA       // Entry template
  ORR      x0, x0, #TT_S1_INNER_SHARED  // 'OR' with inner-shareable attribute
                                        // RW
  ORR      x0, x0, #0x06000000          // 'OR' template with base physical
 address
  MOV      x2, #16
1:
  STR      x0, [x1], #8
  ADD      x0, x0, #0x200000            // Increment the physical address field
  SUB      x2, x2, #1
  CBNZ     x2, 1b

  // [64..127]: 0x0800,0000 - 0x0FFF,FFFF (Flash)
  LDR      x0, =TT_S1_NORMAL_WBWA       // Entry template
```

```
  ORR      x0, x0, #TT_S1_INNER_SHARED  // 'OR' with inner-shareable attribute
  ORR      x0, x0, #TT_S1_PRIV_RO       // 'OR' in Read-only
  ORR      x0, x0, #TT_S1_NS            // 'OR' with NS==1
  ORR      x0, x0, #TT_S1_PXN           // 'OR' with XN==1
  ORR      x0, x0, #0x08000000          // 'OR' template with base physical
 address
  MOV      x2, #64
1:
  STR      x0, [x1], #8
  ADD      x0, x0, #0x200000            // Increment the physical address field
  SUB      x2, x2, #1
  CBNZ     x2, 1b

 // [128..511]: 0x1000,0000 - 0x3FFF,FFFF (Fault)
```