고려대학교
KOREA UNIVERSITY

COSE321 Computer Systems Design

# Lecture 11. ARMv7 Address Translation - MMU & TLB -

Prof. Taeweon Suh

Computer Science & Engineering

Korea University

# Virtual Memory Support in ARMv7-A

- Address translation
  - Virtual address to physical address
- On page basis,
  - Access permission & Protection & Security
    - Readable / Writable
    - Executable / Non-executable
    - Domain
    - Secure / Non-secure access
  - Cacheability & Shareability
    - Cacheable / non-cacheable
    - Shareable / non-shareable

# 2 Translation Tables

When using the Short-descriptor translation table format, two levels of translation tables are held in memory:

**First-level table**

Holds *first-level descriptors* that contain the base address and

- translation properties for a Section and Supersection
- translation properties and pointers to a second-level table for a Large page or a Small page.

**Second-level tables**

Hold *second-level descriptors* that contain the base address and translation properties for a Small page or a Large page. With the Short-descriptor format, second-level tables can be referred to as *Page tables*.

A second-level table requires 1KByte of memory.

| | |
|---|---|
| Virtual Address | **32-bit** |
| Physical Address | **32-bit** |

| | Size | Address translation |
|---|---|---|
| Supersection | **16MB** | 1st-level page table |
| Section | **1MB** | |
| Large Page | **64KB** | 2-level page tables (1st-level & 2nd-level) |
| Small Page | **4KB** | |

- **Supersection: 16MB**
- **Section: 1MB**
- **Large page: 64KB**
- **Small page : 4KB**

# Address Translation using Short Descriptor Tables

- **Supersection: 16MB**
- **Section: 1MB**
- **Large page: 64KB**
- **Page : 4KB**

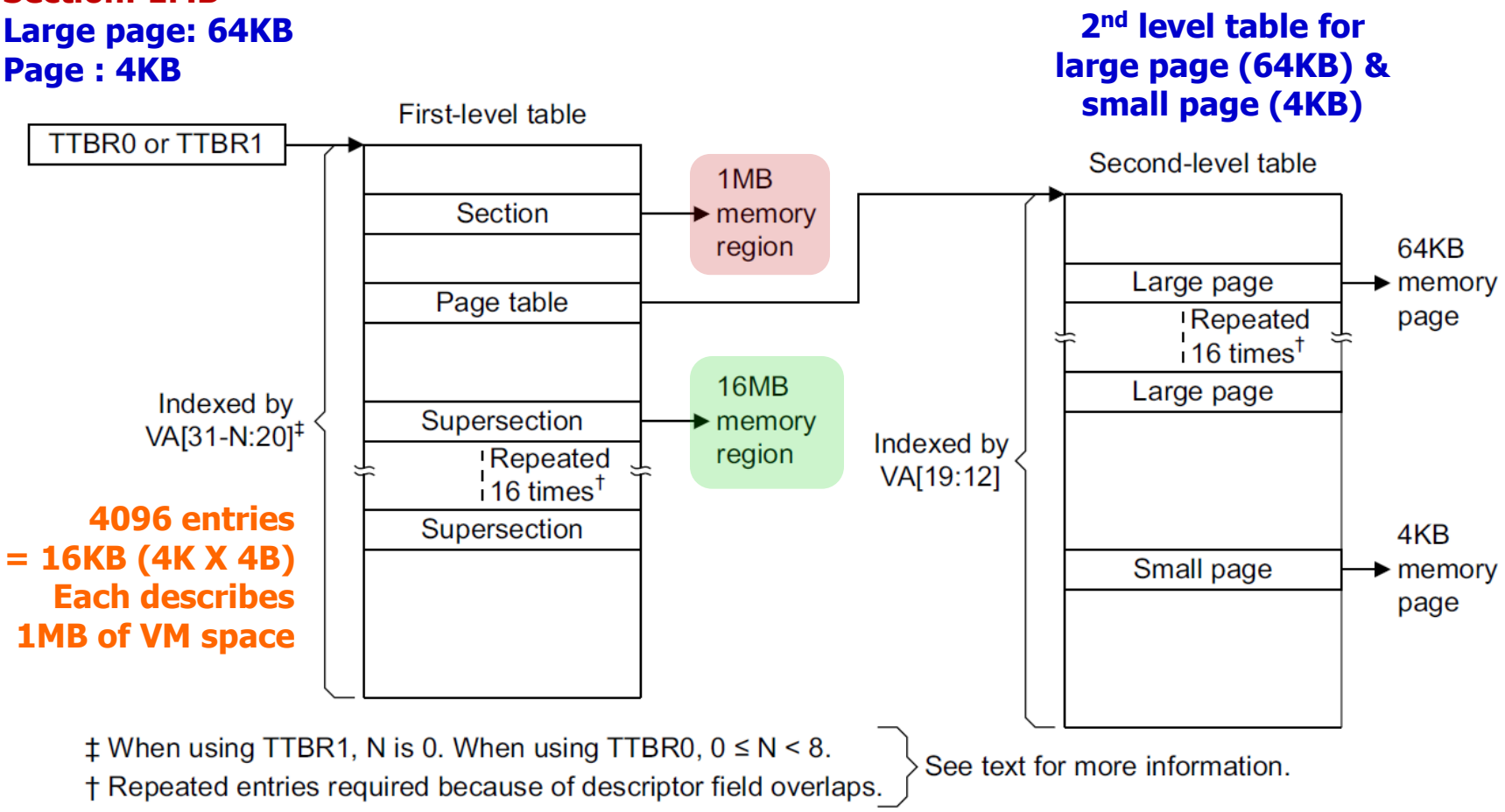
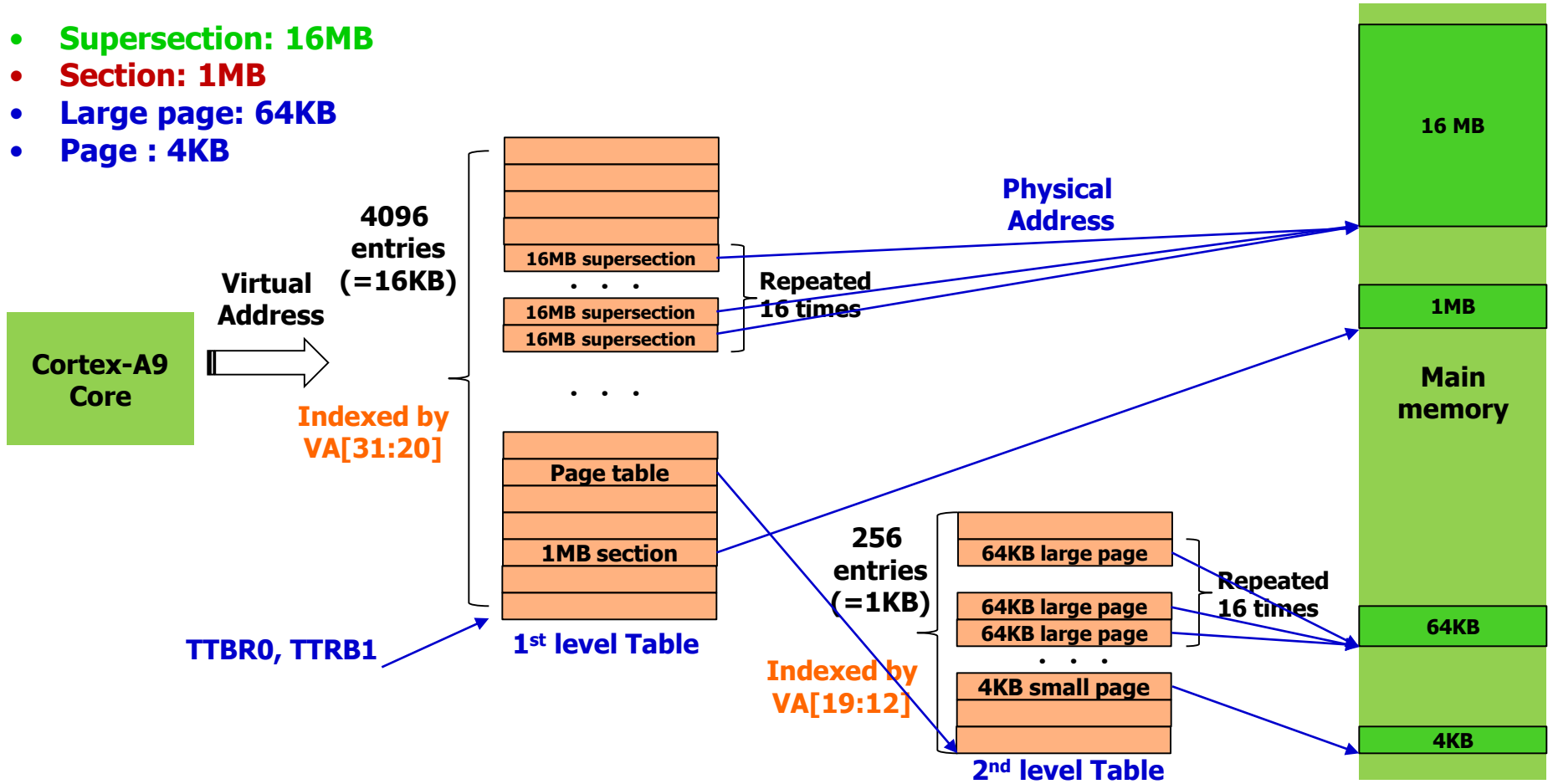



**Figure B3-3 General view of address translation using Short-descriptor format translation tables**

# Page Tables (Translation Tables)

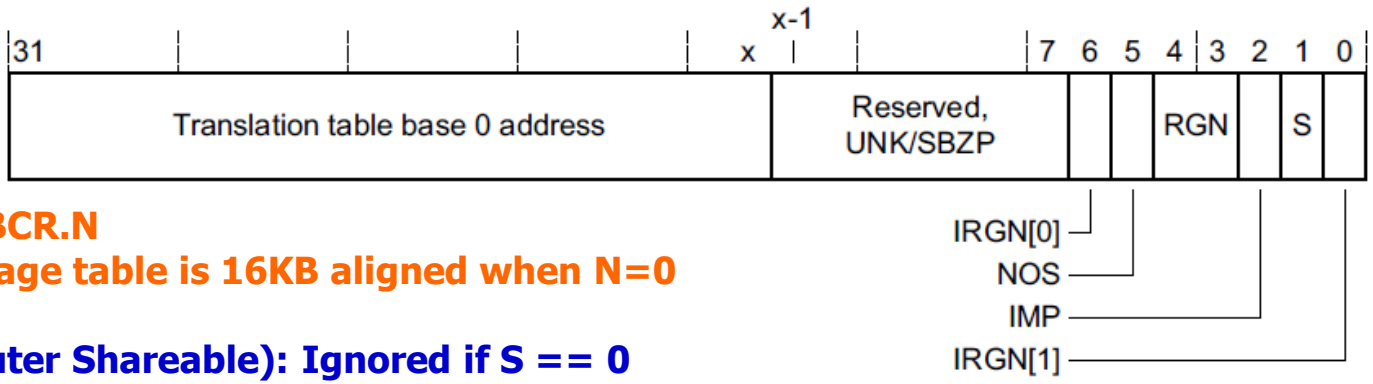
- Each entry of the 1st level table describes 1MB of VM space (total 4GB space)



# TTBR0

## 32-bit TTBR0 format

In an implementation that includes the Multiprocessing Extensions, the 32-bit TTBR0 bit assignments are:




**x = 14 – TTBCR.N**
*** 1st level page table is 16KB aligned when N=0**

**NOS (Not Outer Shareable): Ignored if S == 0**
- **0 : Outer Shareable**
- **1 : Inner Shareable**

**RGN[4:3]: Region bits**

**S: Shareable bit**
- **0 : Non-Shareable**
- **1 : Shareable**

**IRGN[6][0]: Inner Region bits**

**RGN, bits[4:3]**

Region bits. Indicates the Outer cacheability attributes for the memory associated with the translation table walks:

| | |
|---|---|
| 0b00 | Normal memory, Outer Non-cacheable. |
| 0b01 | Normal memory, Outer Write-Back Write-Allocate Cacheable. |
| 0b10 | Normal memory, Outer Write-Through Cacheable. |
| 0b11 | Normal memory, Outer Write-Back no Write-Allocate Cacheable. |

**IRGN, bits[6, 0], in an implementation that includes the Multiprocessing Extensions**

Inner region bits. Indicates the Inner Cacheability attributes for the memory associated with the translation table walks. The possible values of IRGN[1:0] are:

| | |
|---|---|
| 0b00 | Normal memory, Inner Non-cacheable. |
| 0b01 | Normal memory, Inner Write-Back Write-Allocate Cacheable. |
| 0b10 | Normal memory, Inner Write-Through Cacheable. |
| 0b11 | Normal memory, Inner Write-Back no Write-Allocate Cacheable. |

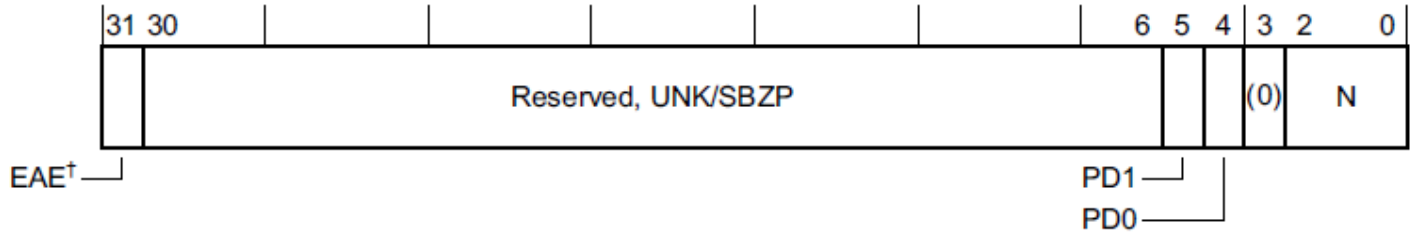
- TTBR0 holds the base address of translation table 0, and **information about the memory it occupies**. This is one of the translation tables for the stage 1 translation of memory accesses from modes other than Hyp mode.

# Translation Table Base Control Register (TTBCR)

- Determines which one (TTBR0 or TTBR1) defines the base address for table walk for address translation

**TTBCR format when using the Short-descriptor translation table format**

In an implementation that includes the Security Extensions and is using the Short-descriptor translation table format, the TTBCR bit assignments are:




† Reserved, UNK/SBZP, if the implementation does not include the Large Physical Address Extension.

**EAE (Extended Address Enable): 0 :32-bit translation system w/ short-descriptor table format**
**1 :40-bit translation system w/ long-descriptor table format**
**PD1: Translation table walk disable for translation using TTBR1**
**0: Perform translation table walk using TTBR1**
**1: a TLB miss generates a translation fault**
**PD0: Translation table walk disable for translation using TTBR0**
**N: Indicate the width of base address in TTBR0**
**address [31:14-N]**

# Selecting between TTBR0 and TTBR1

### B3.5.4 Selecting between TTBR0 and TTBR1, Short-descriptor translation table format

As described in *Determining the translation table base address* on page B3-1320, two sets of translation tables can be defined for each of the PL1&0 stage 1 translations, and TTBR0 and TTBR1 hold the base addresses for the two sets of tables. When using the Short-descriptor translation table format, the value of TTBCR.N indicates the number of most significant bits of the input VA that determine whether TTBR0 or TTBR1 holds the required translation table base address, as follows:

- If N == 0 then use TTBR0. Setting TTBCR.N to zero disables use of a second set of translation tables.
- if N > 0 then:
  - if bits[31:32-N] of the input VA are all zero then use TTBR0
  - otherwise use TTBR1.

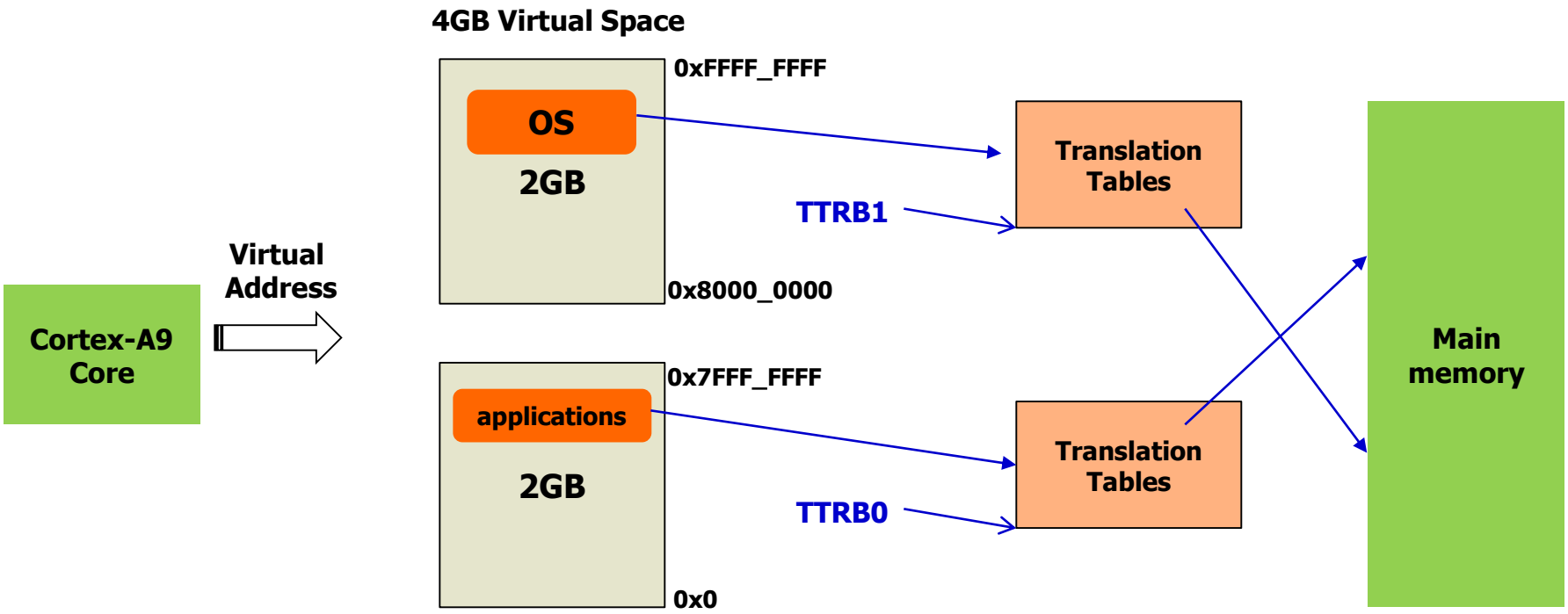
Table B3-1 shows how the value of N determines the lowest address translated using TTBR1, and the size of the first-level translation table addressed by TTBR0.

**Table B3-1 Effect of TTBCR.N on address translation, Short-descriptor format**

| TTBCR.N | First address translated with TTBR1 | TTBR0 table | |
|---|---|---|---|
| | | Size | Index range |
| 0b000 | TTBR1 not used | 16KB | VA[31:20] |
| 0b001 | 0x80000000 | 8KB | VA[30:20] |
| 0b010 | 0x40000000 | 4KB | VA[29:20] |
| 0b011 | 0x20000000 | 2KB | VA[28:20] |
| 0b100 | 0x10000000 | 1KB | VA[27:20] |
| 0b101 | 0x08000000 | 512 bytes | VA[26:20] |
| 0b110 | 0x04000000 | 256 bytes | VA[25:20] |
| 0b111 | 0x02000000 | 128 bytes | VA[24:20] |

# Example (N=1)

- VM space is split into 2 parts
  - Upper part is controlled by the tables pointed to by TTBR1
  - Lower part is controlled by the tables pointed to by TTBR0
- Typical usage case
  - TTBR1 is used by OS
  - TTRB0 is used by applications



# Page Table Descriptors

- **Short-descriptor**
  - Used in implementations that do **not** include large physical address extension
  - Up to 2 levels of address lookup
  - 32-bit input addresses
  - Up to 40-bit output addresses
  - 4B table entries

- **Long-descriptor**
  - Large physical address extension adds support for long descriptor format
  - Up to 3 levels of address lookup
  - Up to 40-bit input addresses, when used for stage 2 translation
  - Up to 40-bit output addresses
  - 8B table entries

# Short-descriptor Translation Table
# L1 Descriptor Formats

**NX: Execute-never bit**
**PXN: Privileged execute-never bit**
**NS: Non-secure bit**
**AP[2:0]: Access Permissions bits**
**S: Shareable**
- **0 : Non-Shareable**
- **1 : Shareable**

**nG: Not Global**
**Domain: specifies one of 16 domains**
- **No access**
- **Clients**
- **Managers**

**Short-descriptor translation table first-level descriptor formats**

Each entry in the first-level table describes the mapping of the associated 1MB MVA range.

Figure B3-4 shows the possible first-level descriptor formats.

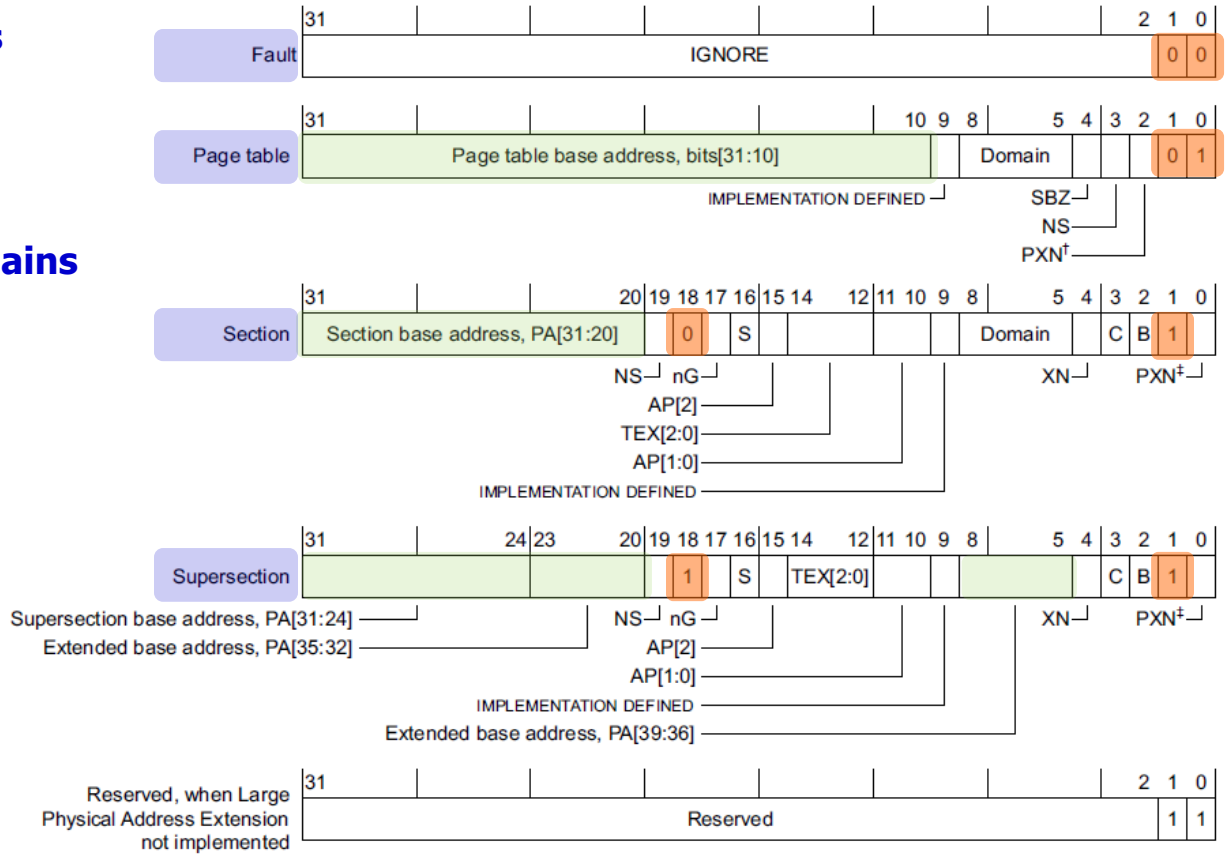



**Figure B3-4 Short-descriptor first-level descriptor formats**

# Short-descriptor Translation Table
# L1 Descriptor Formats

**NX: Execute-never bit**
**PXN: Privileged execute-never bit**
**NS: Non-secure bit**
**AP[2:0]: Access Permissions bits**
**S: Shareable**
- **0 : Non-Shareable**
- **1 : Shareable**

**nG: Not Global**
**Domain: specifies one of 16 domains**
- **No access**
- **Clients**
- **Managers**

Access permission bits in a translation table descriptor control access to the corresponding memory region. The Short-descriptor translation table format supports two options for defining the access permissions:

- three bits, AP[2:0], define the access permissions
- two bits, AP[2:1], define the access permissions, and AP[0] can be used as an Access flag.

SCTLR.AFE selects the access permissions option. Setting this bit to 1, to enable the Access flag, also selects use of AP[2:1] to define access permissions.

**Table B3-8 VMSAv7 MMU access permissions**

| AP[2] | AP[1:0] | PL1 and PL2 access | Unprivileged access | Description |
|---|---|---|---|---|
| 0 | 00 | No access | No access | All accesses generate Permission faults |
| | 01 | Read/write | No access | Access only at PL1 or higher |
| | 10 | Read/write | Read-only | Writes at PL0 generate Permission faults |
| | 11 | Read/write | Read/write | Full access |
| 1 | 00 | - | - | Reserved |
| | 01 | Read-only | No access | Read-only, only at PL1 or higher |
| | 10 | Read-only | Read-only | Read-only at any privilege level, deprecated[a] |
| | 11 | Read-only | Read-only | Read-only at any privilege level[b] |

a. From VMSAv7, ARM strongly recommends use of the 0b11 encoding for Read-only at any privilege level.
b. This mapping is introduced in VMSAv7, and is reserved in VMSAv6.

**Table B3-6 VMSAv7 AP[2:1] access permissions model**

| AP[2], disable write access | AP[1], enable unprivileged access | Access |
|---|---|---|
| 0 | 0[a] | Read/write, only at PL1 |
| 0 | 1 | Read/write, at any privilege level |
| 1 | 0[a] | Read-only, only at PL1 |
| 1 | 1 | Read-only, at any privilege level |

a. Not valid for Non-secure PL2 stage 1 translation tables. AP[1] is SBO in these tables.

# Short-descriptor Translation Table
# L2 Descriptor Formats

**Short-descriptor translation table second-level descriptor formats**

Figure B3-5 shows the possible formats of a second-level descriptor.

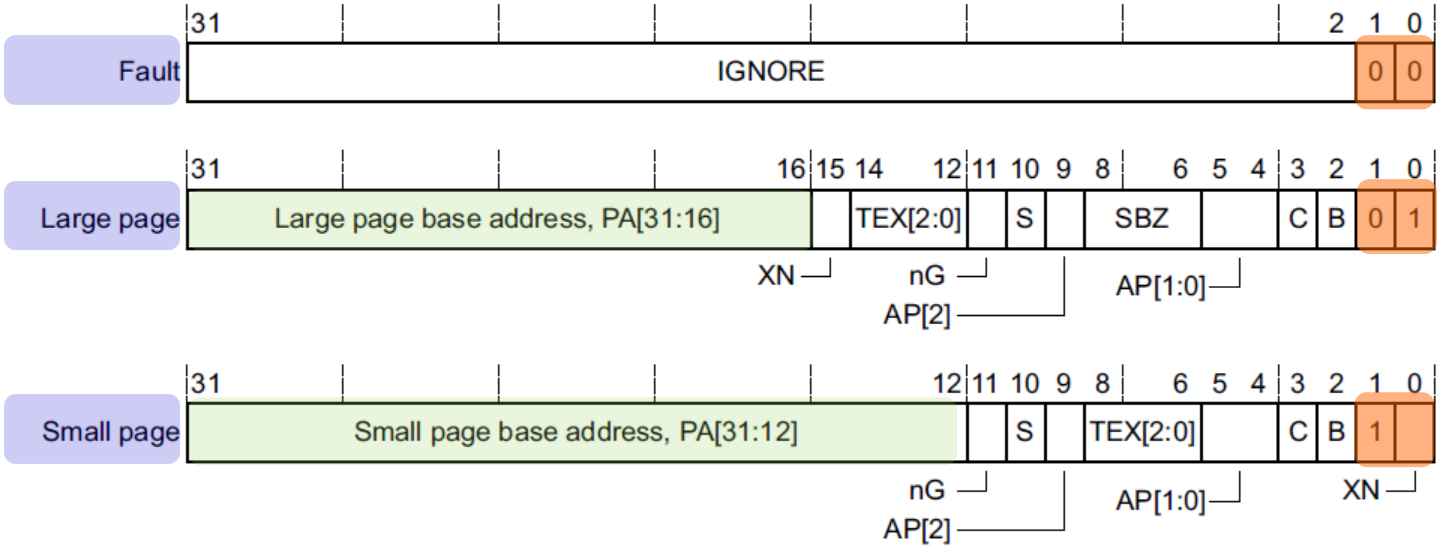
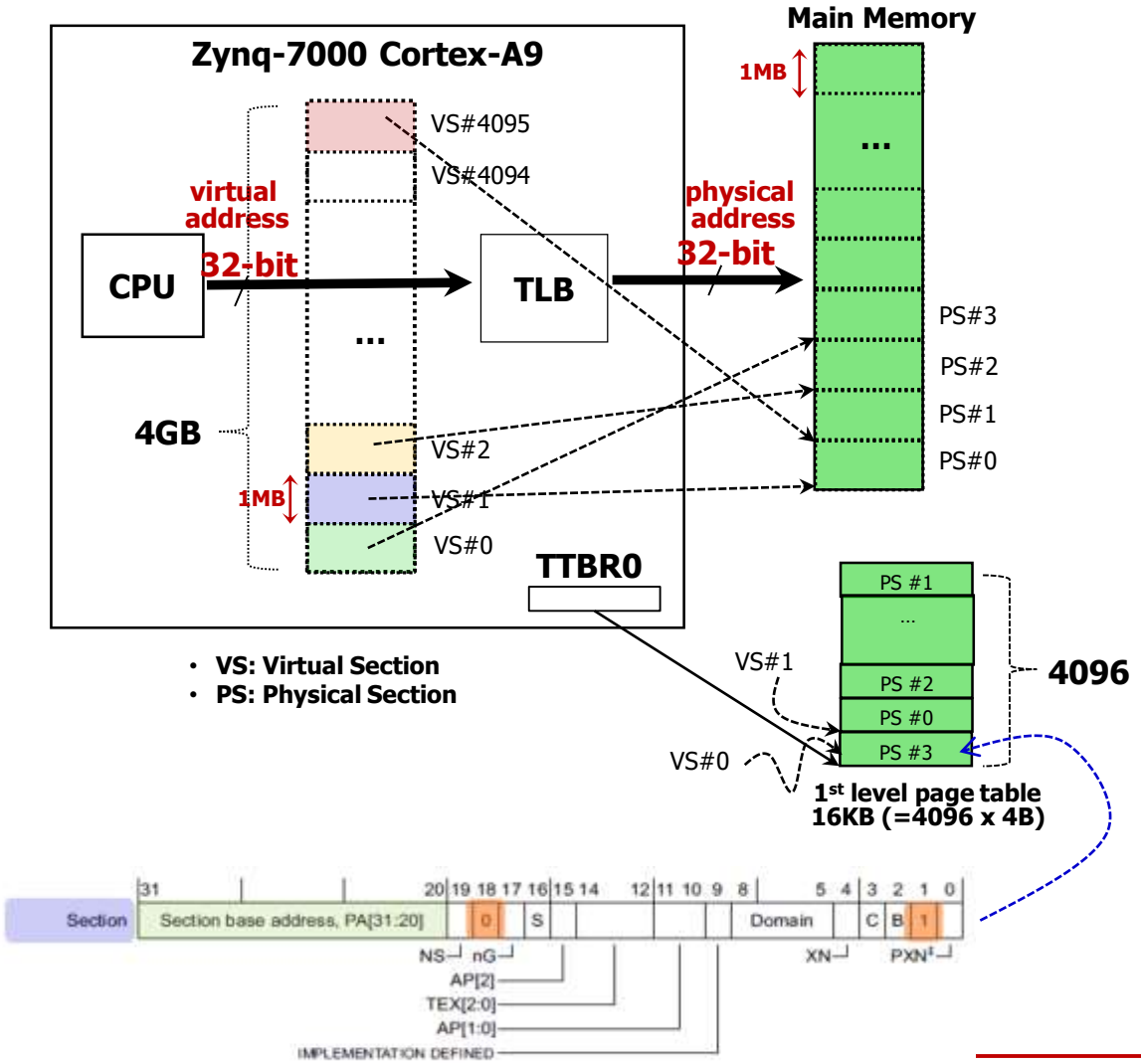



**Figure B3-5 Short-descriptor second-level descriptor formats**

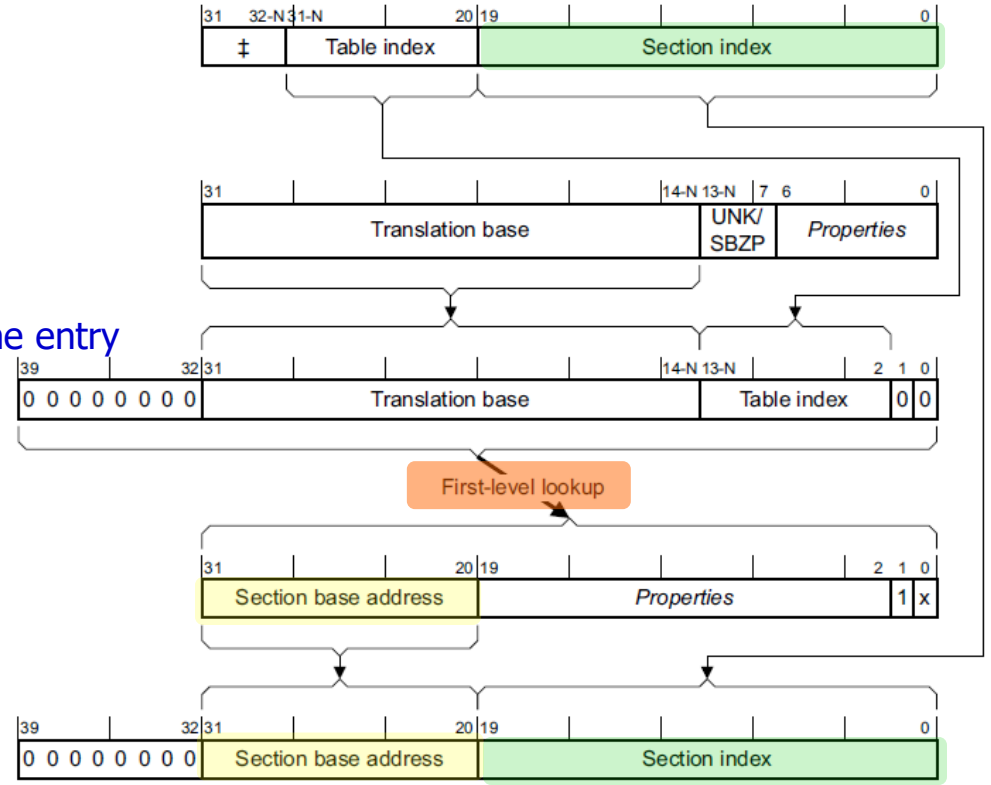
# Section (1MB) translation



# Translation Flow for a Section (1MB)

***Translation flow for a Section***

Figure B3-9 shows the complete translation flow for a Section. For more information about the fields shown in this figure see *The address and Properties fields shown in the translation flows* on page B3-1333.




‡ This field is absent if N is 0
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description.

**Figure B3-9 Section address translation**

# Example - Page table

```
.globl  csd_MMUTable
.section .csd_mmu_tbl,"a"
csd_MMUTable:

                /* Each table entry occupies one 32-bit word and there are
                 * 4096 entries, so the entire table takes up 16KB.
                 * Each entry covers a 1MB section.
                 *
                 * The following defines only three 1MB sections
                 *      1st 1MB: 0x0000_0000 (VA) -> 0x0000_0000 (PA)
                 *      2nd 1MB: 0x0010_0000 (VA) -> 0x0020_0000 (PA)
                 *      3rd 1MB: 0x0020_0000 (VA) -> 0x0040_0000 (PA)
                 */
.set SECT, 0
.word           SECT + 0x15de6  /* S=b1 TEX=b101 AP=b11, Domain=b1111, C=b0, B=b1
*/
.set            SECT, SECT+0x200000
.word           SECT + 0x15de6  /* S=b1 TEX=b101 AP=b11, Domain=b1111, C=b0, B=b1
*/
.set            SECT, SECT+0x200000
.word           SECT + 0x15de6  /* S=b1 TEX=b101 AP=b11, Domain=b1111, C=b0, B=b1
*/
.end
```

1st-level Page Table

1st entry

2nd entry

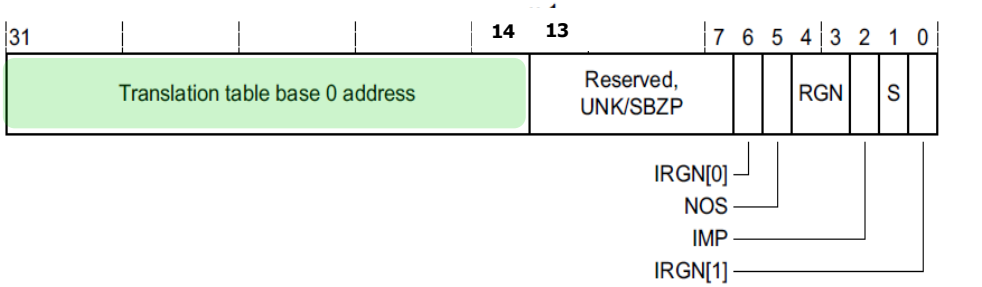
3rd entry

```
.csd_mmu_tbl (ALIGN(16384)) : {
   __csd_mmu_tbl_start = .;
   *(.csd_mmu_tbl)
   __csd_mmu_tbl_end = .;
} > ps7_ram_0_S_AXI_BASEADDR
```

Linker Script

# TTBR0 Setup

In an implementation that includes the Multiprocessing Extensions, the 32-bit TTBR0 bit assignments are:




```
MRC p15, 0, <Rt>, c2, c0, 0    ; Read 32-bit TTBR0 into Rt
MCR p15, 0, <Rt>, c2, c0, 0    ; Write Rt to 32-bit TTBR0
```

```
main:

    /* Load MMU translation table base */
1   ldr       r0,=csd_MMUTable
    orr       r0, r0, #0x5B   /* Outer-cacheable, WB */
    mcr       15, 0, r0, c2, c0, 0  /* TTBR0 */

    // Clear PD0 in TTBCR (Translation Table Base Control Register)
    mrc p15, 0, r2, c2, c0, 2
    bic r2, r2, #(1<<4)    // Set PD0 to 0 = HW-based page table walk
    mcr p15, 0, r2, c2, c0, 2

    /* Enable MMU, I$ and D$ */
    // Set M-bit in SCTLR (System Control Register)
    mrc p15, 0, r0, c1, c0, 0
    bic r0, r0, #(1<<12)  // I-bit = I$ disable
    bic r0, r0, #(1<<2)    // C-bit = $ disable
    orr r0, r0, #(1<<0)    // M-bit = MMU enable
    mcr       p15, 0, r0, c1, c0, 0 /* Enable MMU */
```

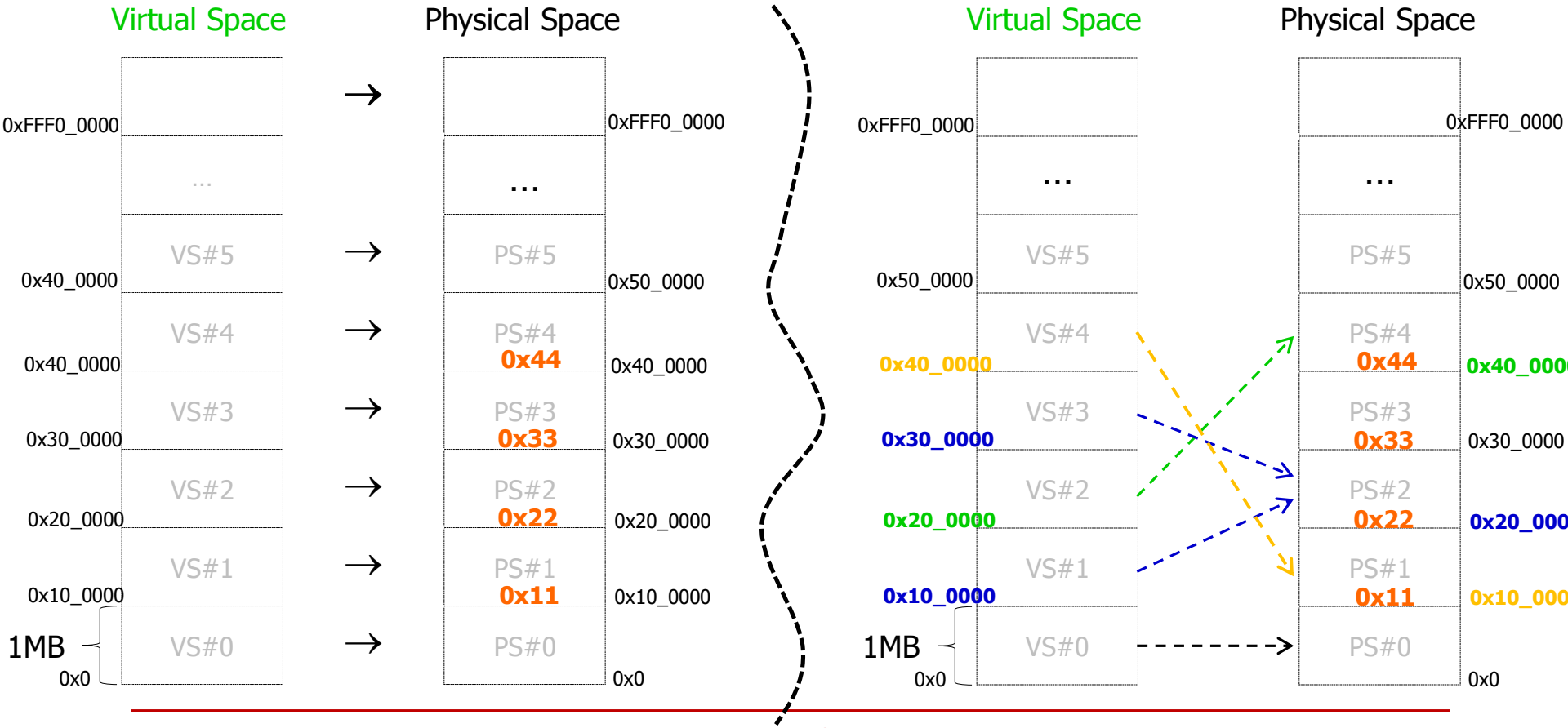
# Demo

**Before**

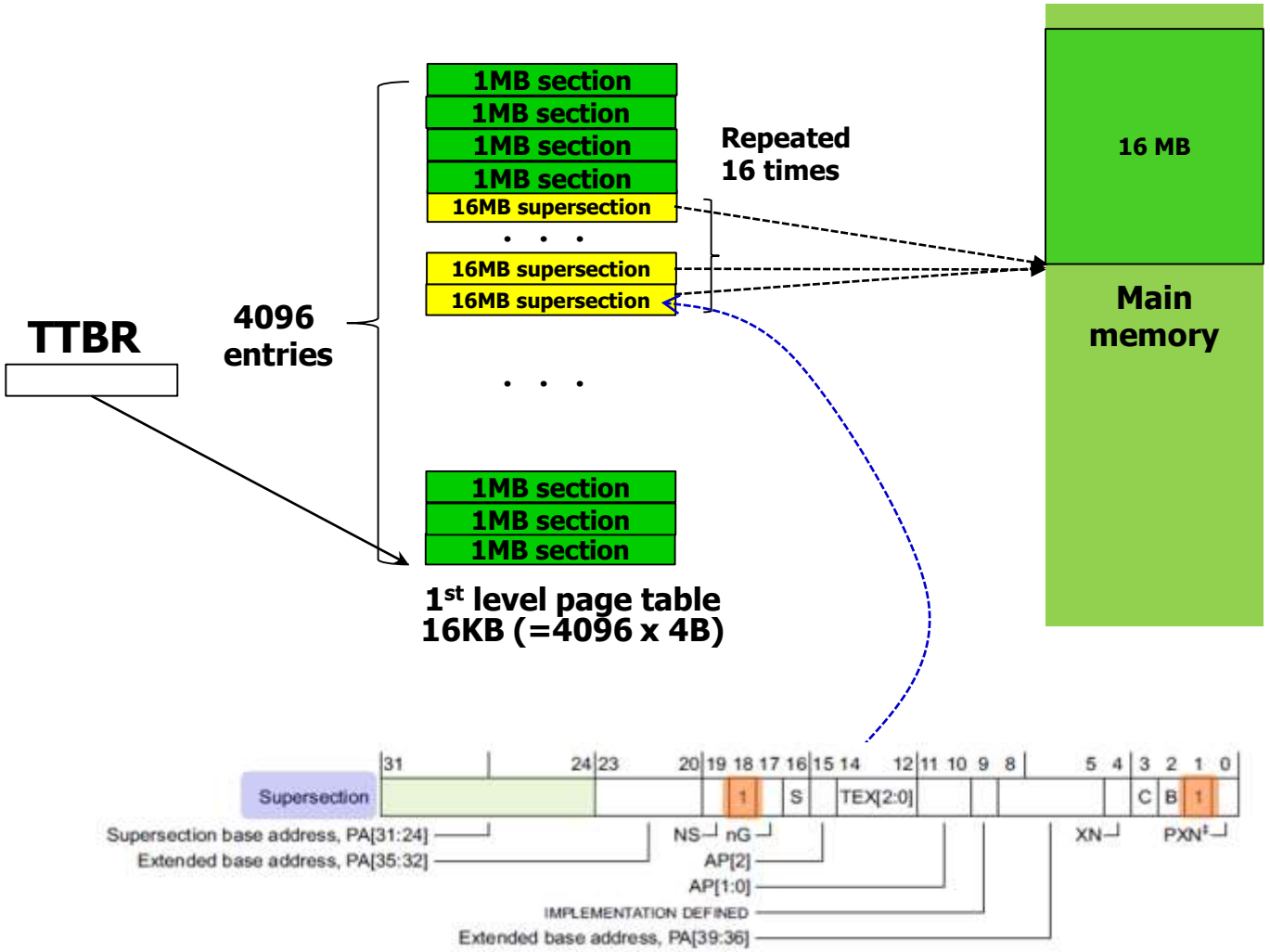
- TTBR0: base address of page table
- TTBCR: HW-based page table walk
- SCTLR: MMU enable

**After**

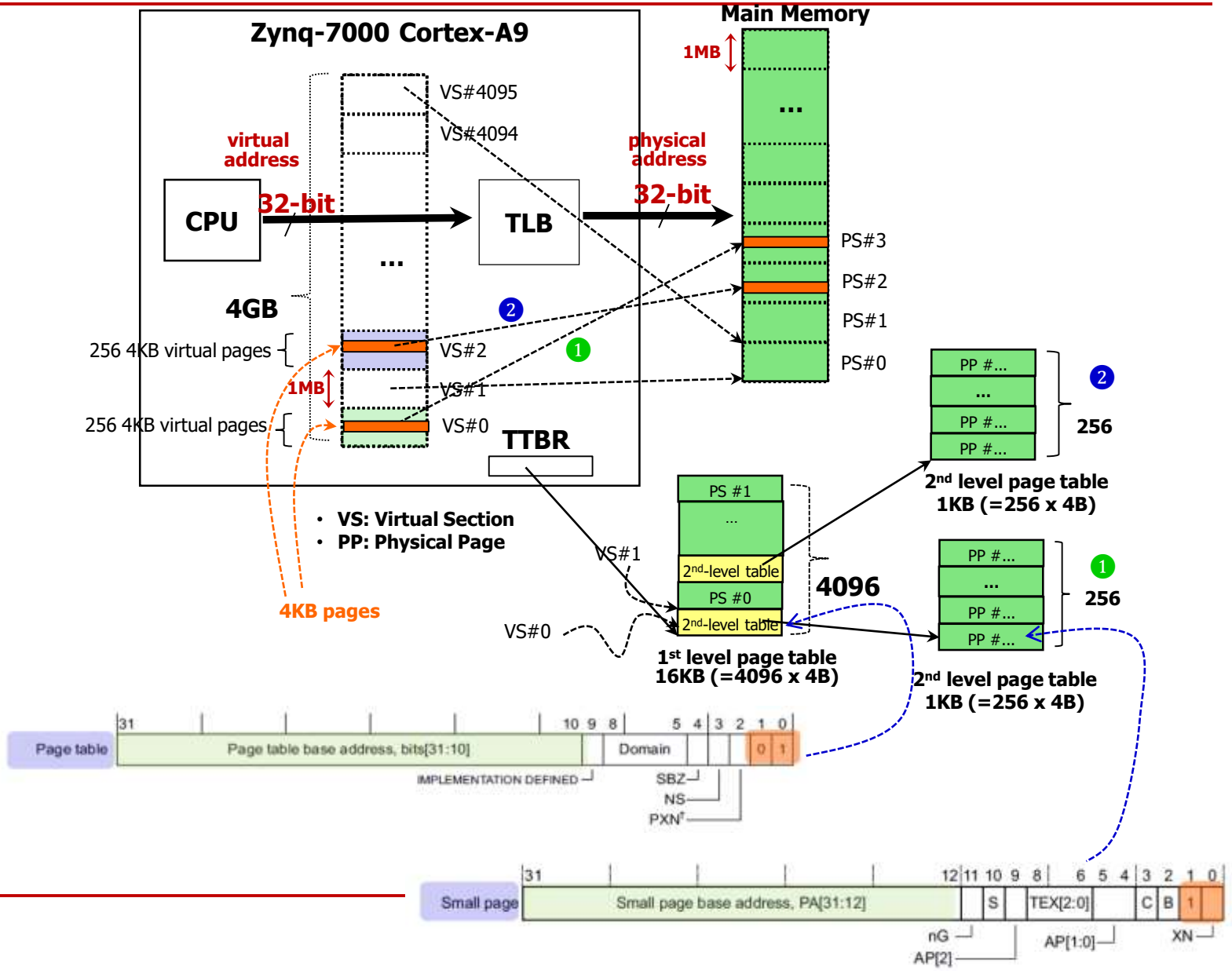
**Flat mapping** ⟶ **Mapping by Page Table**



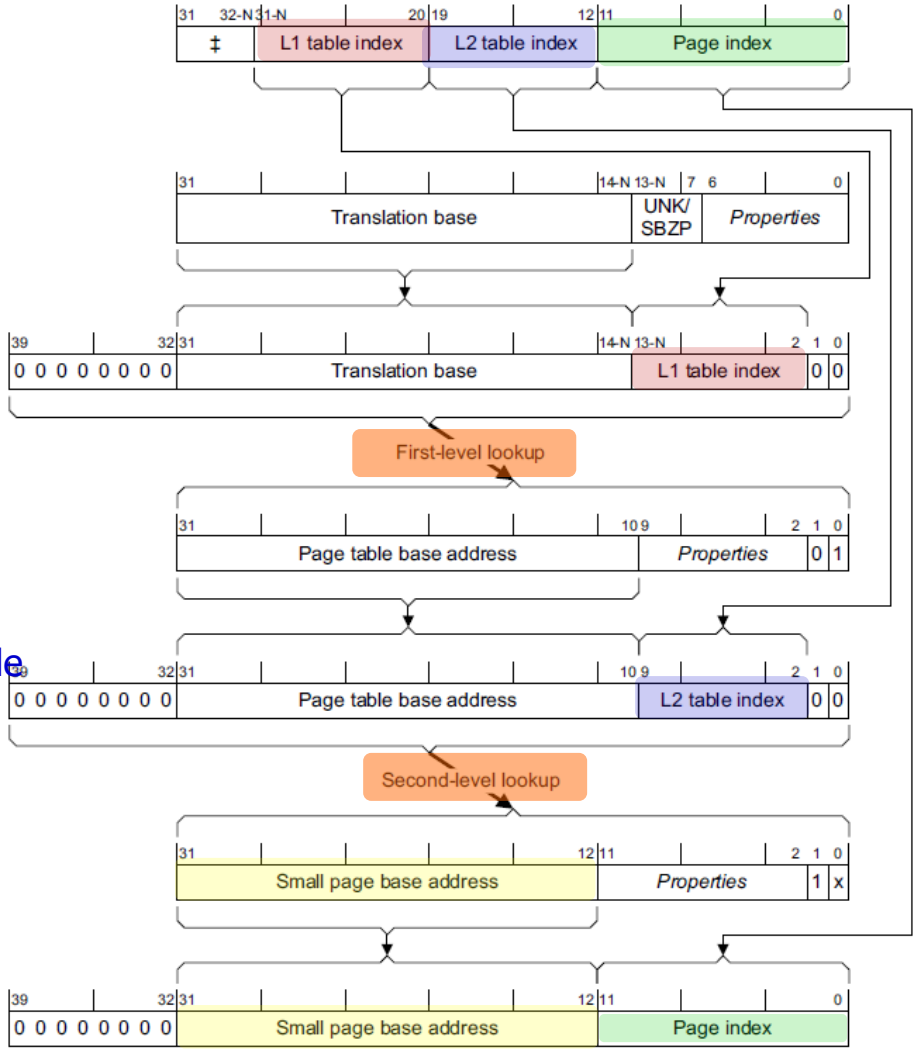
# Supersection (16MB) translation



# Small Page (4KB) translation

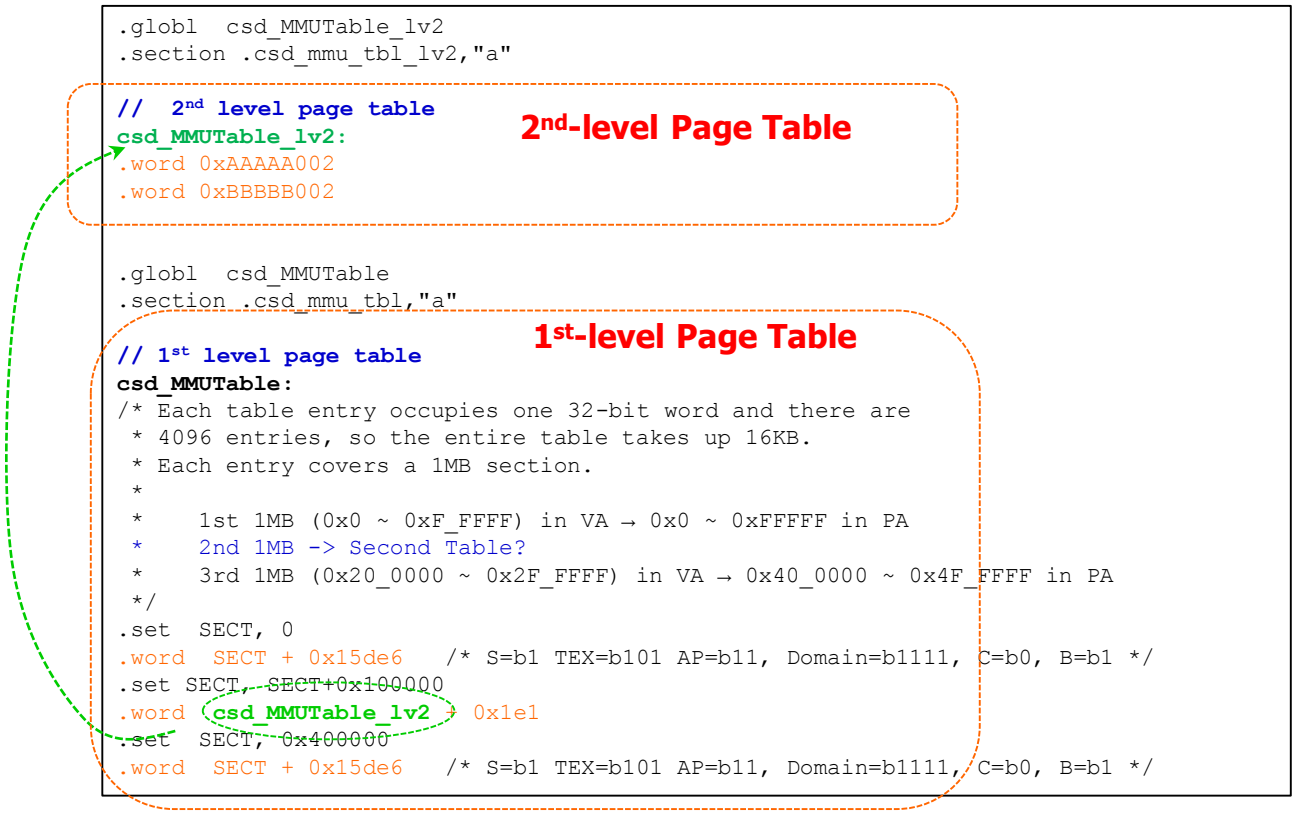


# Translation Flow for a Small Page (4KB)

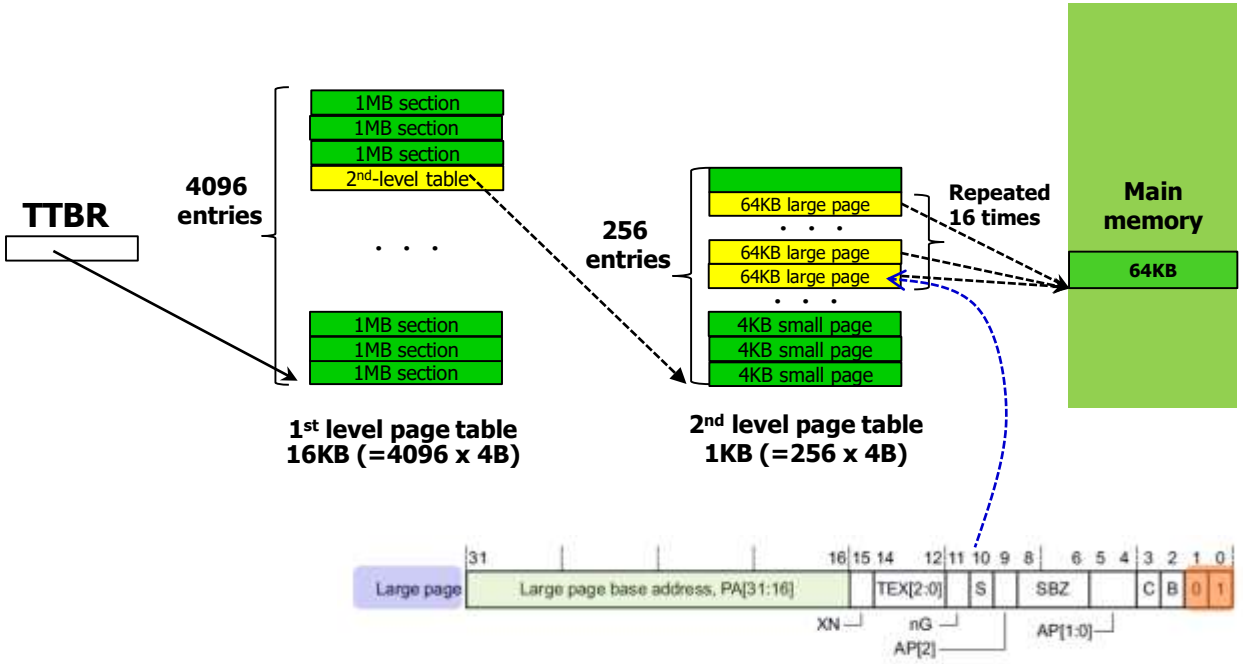



‡ This field is absent if N is 0
L1 = First-level, L2 = Second-level
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description.

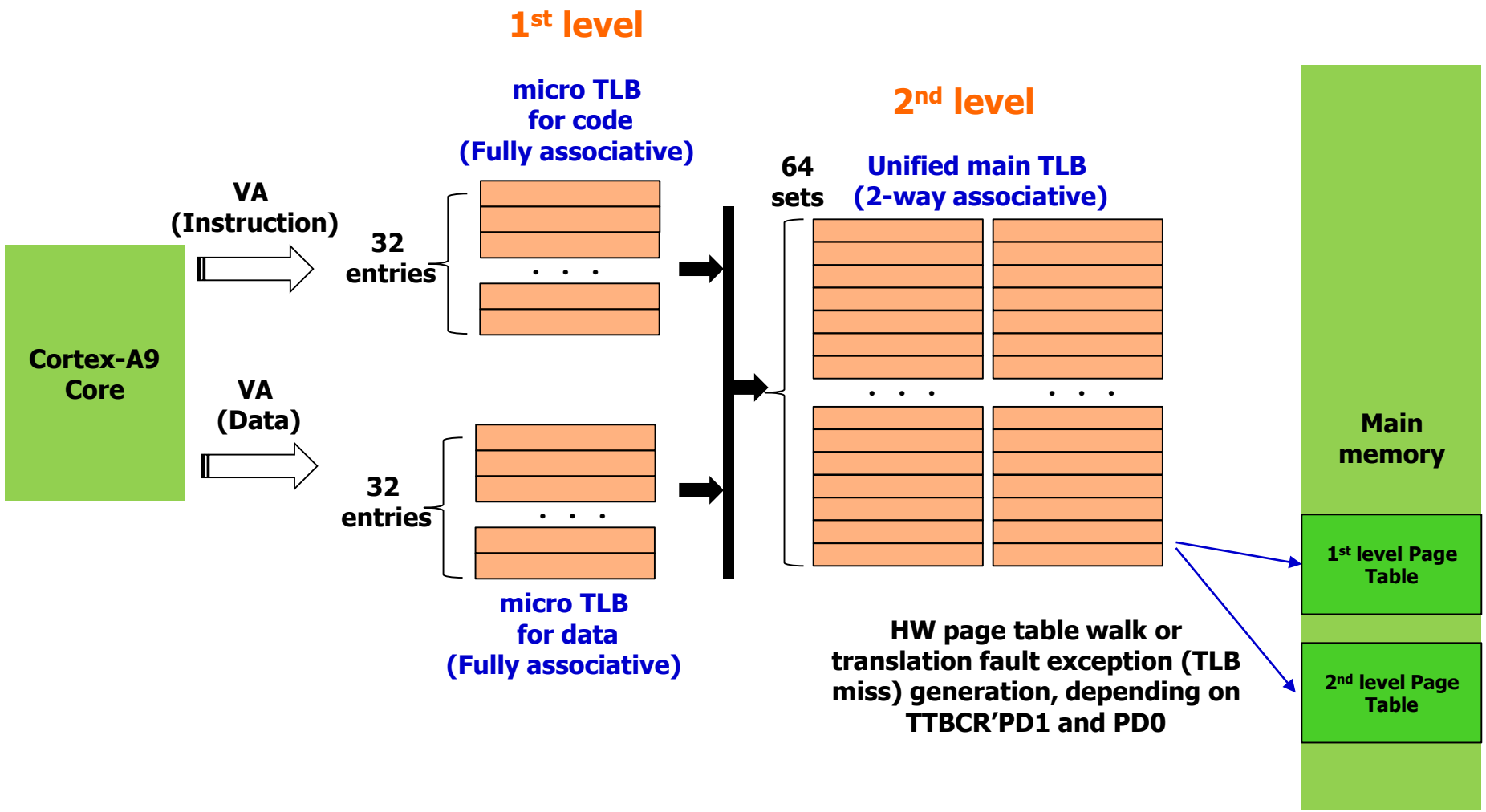
# Example - 2nd level page table



# Large Page (64KB) translation

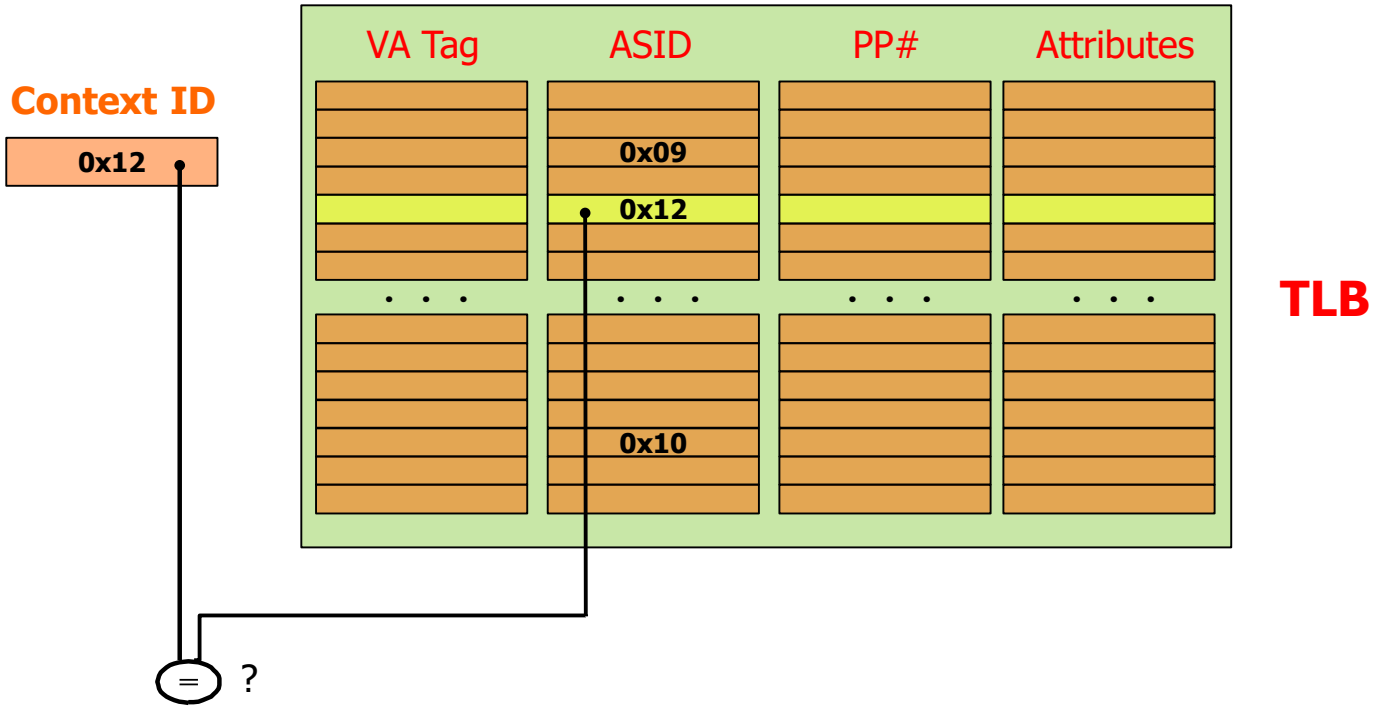


# TLBs in Cortex-A9



# ASIDs (Address Space Identifiers)

- ASID is used to identify the currently running task
  - Each application has its own page table and ASID
  - ASID is stored in Context ID register as well
- No need to flush TLBs upon context-switching (task switching)



# Non-Deterministic MMU Behavior

- Assume that a page is 4KB in size

| Code | Worst-case scenario in $ | Worst-case scenario in TLB |
|---|---|---|
| 0x091C: add r0, r1, r2 | I$ miss<br>• 8-word line fill from 0x0900 | ITLB miss |
| 0x0920: mov r12, #0xA000 | I$ miss<br>• 8-word line fill from 0x0920 | |
| 0x0924: ldr r3, [r12, #0] | D$ miss<br>• D$ dirty line eviction<br>• 8-word line fill from 0xA000 | DTLB miss |
| 0x0928: ldm sp!, {r0-r9} | 3 D$ misses<br>• 3 D$ dirty line evictions<br>• 3 Line fills (miss for r0, miss for r1~r8, and miss for r9) | 2 DTLB misses |

# TLB Maintenance Operations

### VMSA CP15 c8 register summary, TLB maintenance operations

On an ARMv7-A implementation, the CP15 c8 registers provide TLB maintenance functions. Figure B3-34 shows the CP15 c8 registers.

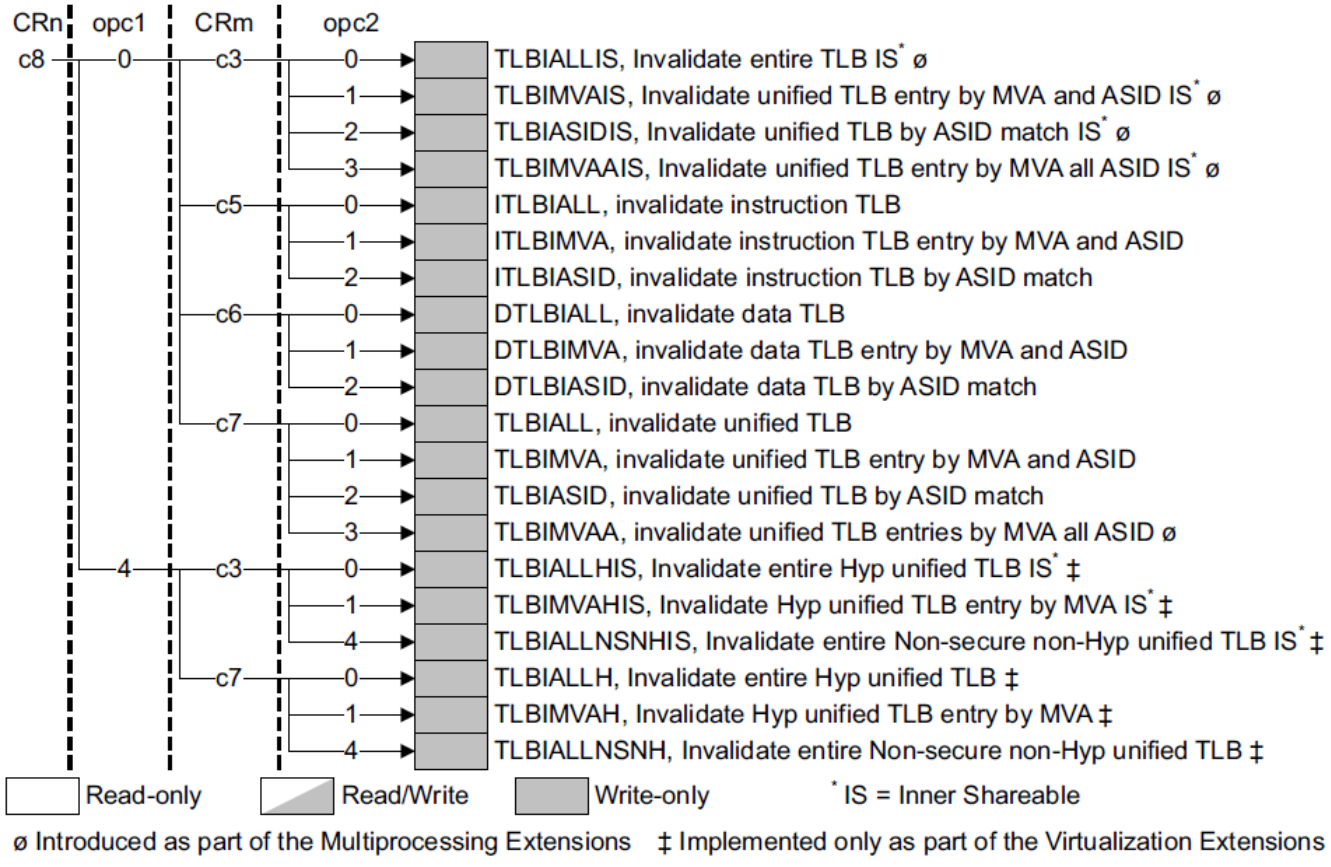
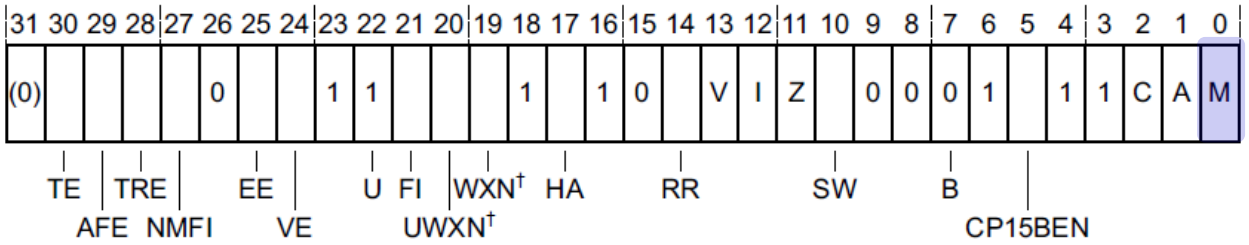



**Figure B3-34 CP15 c8 registers in a VMSA implementation**

# System Control Register (SCTLR)

- SCTLR provides the top level control of the system

In a VMSAv7 implementation, the SCTLR bit assignments are:




† Reserved before the introduction of the Virtualization Extensions, see text for more information.

- **TE: Thumb Exception Enable**
- **AFE: Access Flag Enable**
  **0: In the translation table descriptors, AP[0] is an access permission bit**
  **1: In the translation table descriptors, AP[0] is an access flag**
- **TRE: TEX remap enable**
  **0: TEX remap disabled. TEX[2:0] are used with the C and B bits to describe memory region attributes**
  **1: TEX remap enabled. TEX[2:1] are reassigned for use as bits managed by OS. The TEX[0], C and B bits, with the MMU remap registers describe the memory region attributes**
- **VE: Interrupt Vectors Enable**
- **RR: Round Robin select. Cache replacement policy**
- **V: Vectors bit. 0: Low vectors 0x0, 1: High vectors (Hivecs): 0xFFFF0000**
- **I: I$ enable**
- **Z: Branch prediction enable**
- **CP15BEN: CP15 barrier enable**
- **C: $ enable**
- **A: ~~Alignment check enable~~**
- **M: MMU enable**

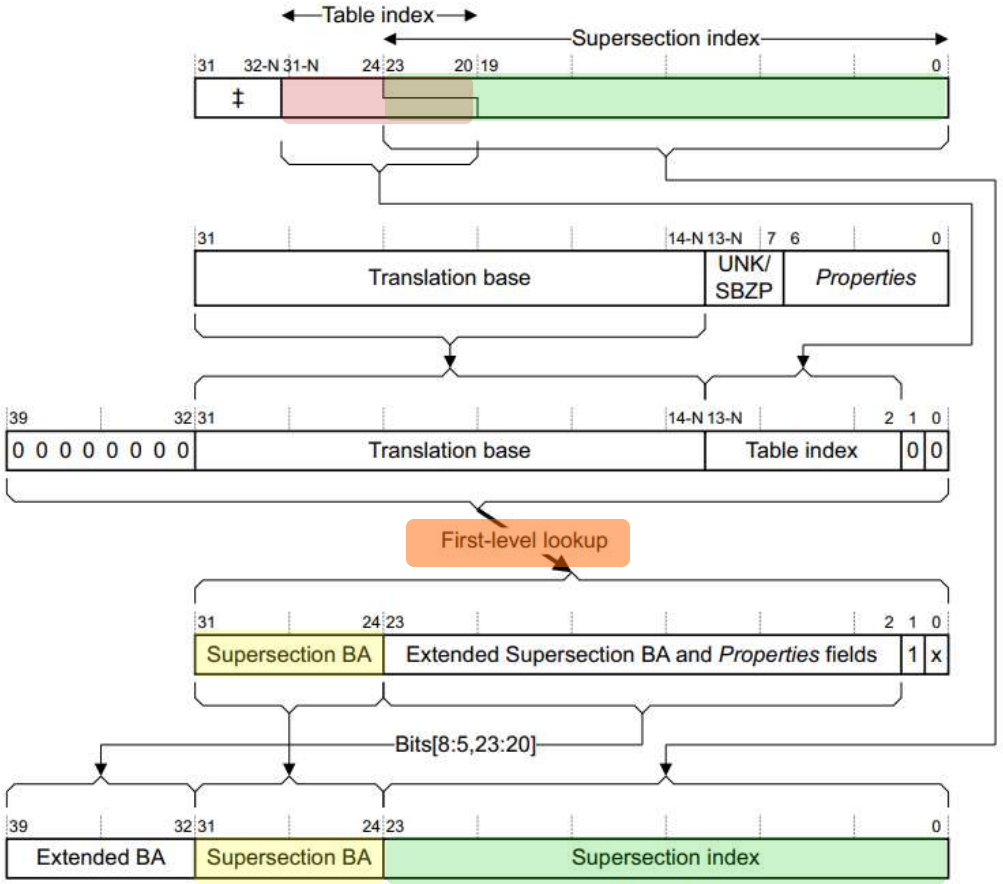
# Backup Slides

# Privilege Levels

- ARMv7 defines different levels of execution privilege:
  - In Secure state, PL1 and PL0
  - In Non-secure state, PL2, PL1, and PL0

  - PL0
    - Lowest privilege (unprivileged level)
    - User applications run in **User** mode
  - PL1
    - Software execution in **all modes other than User or Hyp** mode
  - PL2
    - Software executing in **Hyp** mode

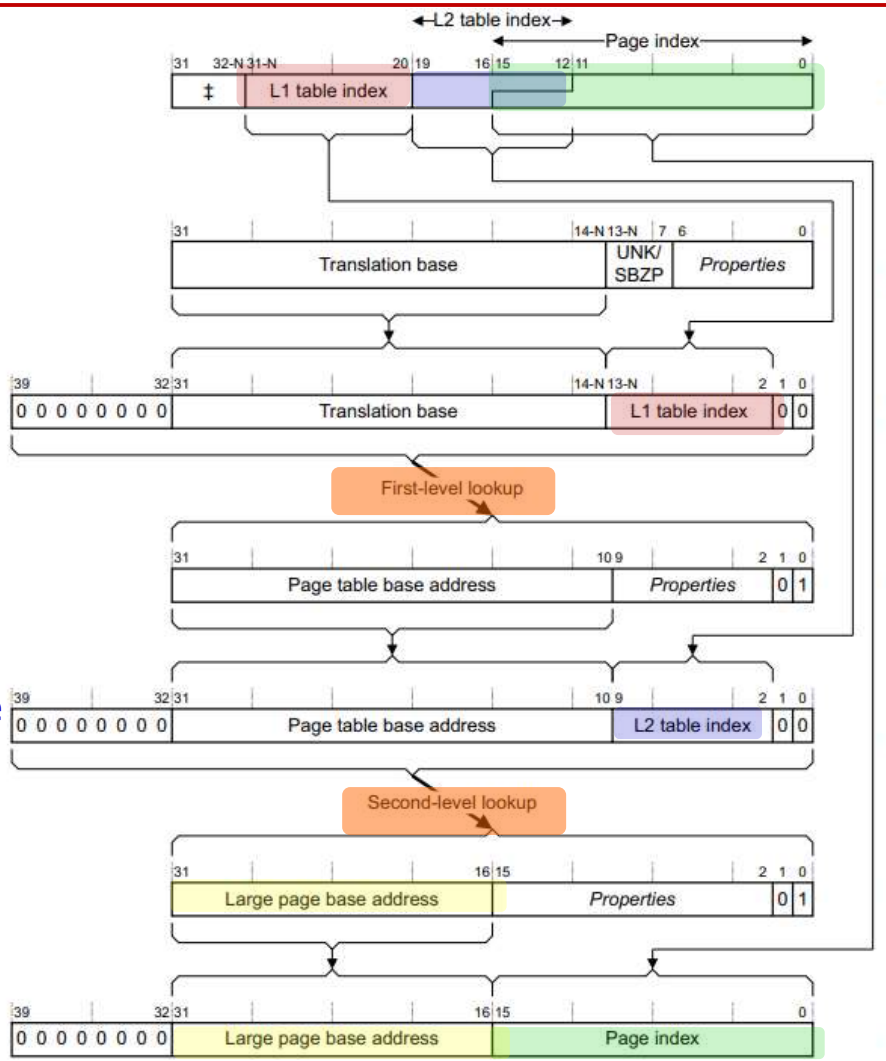
# Translation Flow for a SuperSection (16MB)




‡ This field is absent if N is 0
BA = Base address
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description

**Figure B3-8 Supersection address translation**

# Translation Flow for a Large Page (64KB)




‡ This field is absent if N is 0
L1 = First-level, L2 = Second-level
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description

Figure B3-10 Large page address translation

# Domains

- Domains provide a mechanism to override permissions for an associated group of memory regions
- DACR (Domain Access Control Register) must be configured before enabling the MMU
- Each of 16 domains can be configured to the following states:
  - **Client (b'01)**: Obey access permissions in the section or page descriptor
  - **Manager (b'11)**: **Ignore access permissions in the section or page descriptor**
  - **No access (b'00)**: Any access will generate a domain fault
- Each 1st level descriptor specifies a domain number
  - All entries in a 2nd level table inherit the domain from the associated 1st level descriptor

* Excerpts from AAA

# Domain

### B4.1.43 DACR, Domain Access Control Register, VMSA

The DACR characteristics are:

**Purpose** DACR defines the access permission for each of the sixteen memory domains.

This register is part of the Virtual memory control registers functional group.

**Usage constraints** Only accessible from PL1 or higher.

**Configurations** If the implementation includes the Security Extensions, this register:

- is Banked
- has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

In an implementation that includes the Large Physical Address Extension, this register has no function when TTBCR.EAE is set to 1, to select the Long-descriptor translation table format.

**Attributes** A 32-bit RW register with an UNKNOWN reset value. For more information see *Reset behavior of CP14 and CP15 registers* on page B3-1450.

Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

The DACR bit assignments are:

| 31 30 | 29 28 | 27 26 | 25 24 | 23 22 | 21 20 | 19 18 | 17 16 | 15 14 | 13 12 | 11 10 | 9 8 | 7 6 | 5 4 | 3 2 | 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |

**Dn, bits[(2n+1):2n]**

Domain n access permission, where n = 0 to 15. Permitted values are:

| | |
|---|---|
| 0b00 | No access. Any access to the domain generates a Domain fault. |
| 0b01 | Client. Accesses are checked against the permission bits in the translation tables. |
| 0b10 | Reserved, effect is UNPREDICTABLE. |
| 0b11 | Manager. Accesses are not checked against the permission bits in the translation tables. |

For more information, see *Domains, Short-descriptor format only* on page B3-1362.

# Inner & Outer Cacheable?

**Note**

The terms Inner and Outer in this document represent the levels of caches that can be built in a system. Inner refers to the innermost caches, including level one. Outer refers to the outermost caches. The boundary between Inner and Outer caches is defined in the implementation of a cached system. Inner must always include level one. In a system with three levels of caches, an example is for the Inner attributes to apply to level one and level two, while the Outer attributes apply to level three. In a two-level system, it is envisaged that Inner always applies to level one and Outer to level two.

In the processor, Inner refers to level one and the **ARSIDEBAND[4:1]**, for read, and **AWSIDEBAND[4:1]**, for writes, signals show the Inner Cacheable values.

**ARCACHE**, for reads, and **AWCACHE**, for writes, show the Outer Cacheable properties.

Table 6.2. TEX field, and C and B bit encodings used in pag

| Page table encodings | | | Description | |
|---|---|---|---|---|
| TEX | C | B | | |
| b000 | 0 | 0 | Strongly Ordered | |
| b000 | 0 | 1 | Shared Device | Device |
| b000 | 1 | 0 | Outer and Inner Write-Through, No Allocate on Write | Normal |
| b000 | 1 | 1 | Outer and Inner Write-Back, No Allocate on Write | Normal |
| b001 | 0 | 0 | Outer and Inner Noncacheable | Normal |
| b001 | 0 | 1 | Reserved | - |
| b001 | 1 | 0 | Reserved | - |
| b001 | 1 | 1 | Outer and Inner Write-Back, Allocate on Write(c) | Normal |
| b010 | 0 | 0 | Non-Shared Device | Device |
| b010 | 0 | 1 | Reserved | - |
| 010 | 1 | X | Reserved | - |
| 011 | X | X | Reserved | - |
| 1BB | A | A | Cached memory. BB = Outer policy, AA = Inner policy. See Table 6.3. | Normal |

**Cacheable memory attributes, without TEX remap**

When TEX[2] == 1, the translation table entry describes Cacheable memory, and the rest of the encoding defines the Inner and Outer cache attributes:

| | |
|---|---|
| TEX[1:0] | Define the Outer cache attribute. |
| C, B | Define the Inner cache attribute. |

The translation table entries use the same encoding for the Outer and Inner cache attributes, as Table B3-11 shows.

**Table B3-11 Inner and Outer cache attribute encoding**

| Encoding | Cache attribute |
|---|---|
| 00 | Non-cacheable |
| 01 | Write-Back, Write-Allocate |
| 10 | Write-Through, no Write-Allocate |
| 11 | Write-Back, no Write-Allocate |

RM11 processors

ARM1176JZ-S Technical Reference Manual Revision: r0p7

Home > Memory Management Unit > Memory region attributes > C and B bit, and type extension field encodings

### 6.6.1. C and B bit, and type extension field encodings

# TTBCR

## B4.1.153 TTBCR, Translation Table Base Control Register, VMSA

The TTBCR characteristics are:

**Purpose**

TTBCR determines which of the Translation Table Base Registers, TTBR0 or TTBR1, defines the base address for a translation table walk required for the stage 1 translation of a memory access from any mode other than Hyp mode.

If the implementation includes the Large Physical Address Extension, the TTBCR also:

- Controls the translation table format.
- When using the Long-descriptor translation table format, holds cacheability and shareability information for the accesses.

**Note**

When using the Short-descriptor translation table format, TTBR0 and TTBR1 hold this cacheability and shareability information.

This register is part of the Virtual memory control registers functional group.

**Usage constraints**

Only accessible from PL1 or higher.

**Configurations**

The Large Physical Address Extension adds an alternative format for the register. If an implementation includes the Large Physical Address Extension then the current translation table format determines which format of the register is used.

If the implementation includes the Security Extensions, this register:

- is Banked
- has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

**Attributes**

A 32-bit RW register that resets to zero. If the implementation includes the Security Extensions this defined reset value applies only to the Secure copy of the register, except for the EAE bit in an implementation that includes the Large Physical Address Extension. For more information see the field descriptions. See also *Reset behavior of CP14 and CP15 registers* on page B3-1450.

Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

# TTBR0

## B4.1.154 TTBR0, Translation Table Base Register 0, VMSA

The TTBR0 characteristics are:

**Purpose**

TTBR0 holds the base address of translation table 0, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses from modes other than Hyp mode.

This register is part of the Virtual memory control registers functional group.

**Usage constraints**

Only accessible from PL1 or higher.

Used in conjunction with the TTBCR. When the 64-bit TTBR0 format is used, cacheability and shareability information is held in the TTBCR, not in TTBR0.

**Configurations**

The Multiprocessing Extensions change the TTBR0 32-bit register format.

The Large Physical Address Extension extends TTBR0 to a 64-bit register. In an implementation that includes the Large Physical Address Extension, TTBCR.EAE determines which TTBR0 format is used:

**EAE==0** 32-bit format is used. TTBR0[63:32] are ignored.

**EAE==1** 64-bit format is used.

If the implementation includes the Security Extensions, this register:

- is Banked
- has write access to the Secure copy of the register disabled when the **CP15SDISABLE** signal is asserted HIGH.

**Attributes**

A 32-bit or 64-bit RW register with a reset value that depends on the register implementation. For more information see the register bit descriptions. See also *Reset behavior of CP14 and CP15 registers* on page B3-1450.

Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

# Context ID Register

### B4.1.36 CONTEXTIDR, Context ID Register, VMSA

The CONTEXTIDR characteristics are:

**Purpose** CONTEXTIDR identifies the current *Process Identifier* (PROCID) and, when using the Short-descriptor translation table format, the *Address Space Identifier* (ASID).

This register is part of the Virtual memory control registers functional group.

**Usage constraints** Only accessible from PL1 or higher.

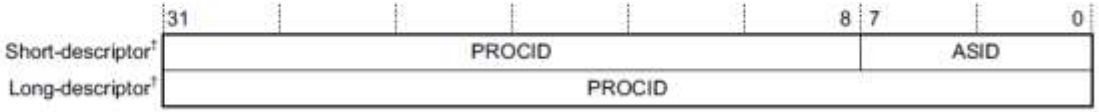
**Configurations** The register format depends on whether address translation is using the Long-descriptor or the Short-descriptor translation table format.

In an implementation that includes the Security Extensions, this register is Banked.

**Attributes** A 32-bit RW register with an UNKNOWN reset value. See also *Reset behavior of CP14 and CP15 registers* on page B3-1450.

Table B3-45 on page B3-1493 shows the encodings of all of the registers in the Virtual memory control registers functional group.

In a VMSA implementation, the CONTEXTIDR bit assignments are:




† Current translation table format

**PROCID, bits[31:0], when using the Long-descriptor translation table format**

**PROCID, bits[31:8], when using the Short-descriptor translation table format**

Process Identifier. This field must be programmed with a unique value that identifies the current process. See also *Using the CONTEXTIDR*.

**ASID, bits[7:0], when using the Short-descriptor translation table format**

Address Space Identifier. This field is programmed with the value of the current ASID.

**Note**

When using the Long-descriptor translation table format, either TTBR0 or TTBR1 holds the current ASID.

# Context ID Register

### Using the CONTEXTIDR

The value of the whole of this register is called the *Context ID* and is used by:

- the debug logic, for Linked and Unlinked Context ID matching, see *Breakpoint debug events* on page C3-2039 and *Watchpoint debug events* on page C3-2057
- the trace logic, to identify the current process.

The ASID field value is an identifier for a particular process. In the translation tables it identifies entries associated with a process, and distinguishes them from global entries. This means many cache and TLB maintenance operations take an ASID argument.

For information about the synchronization of changes to the CONTEXTIDR see *Synchronization of changes to system control registers* on page B3-1461. There are particular synchronization requirements when changing the ASID and Translation Table Base Registers, see *Synchronization of changes of ASID and TTBR* on page B3-1386.

### Accessing the CONTEXTIDR

To access the CONTEXTIDR, software reads or writes the CP15 registers with <opc1> set to 0, <CRn> set to c13, <CRm> set to c0, and <opc2> set to 1. For example:

```
MRC p15, 0, <Rt>, c13, c0, 1    ; Read CONTEXTIDR into Rt
MCR p15, 0, <Rt>, c13, c0, 1    ; Write Rt to CONTEXTIDR
```

# Access Permission Encodings

- The access permission bits in the page table entry give the access permission for a page

*Table 3-2:* **Access Permission Encodings**

| APX | AP1 | AP0 | Privileged | Unprivileged | Description |
|---|---|---|---|---|---|
| 0 | 0 | 0 | No access | No access | Permission fault |
| 0 | 0 | 1 | Read/Write | No access | Privileged access only |
| 0 | 1 | 0 | Read/Write | Read | No user-mode write |
| 0 | 1 | 1 | Read/Write | Read/Write | Full access |
| 1 | 0 | 0 | ~ | ~ | Reserved |
| 1 | 0 | 1 | Read | No access | Privileged Read only |
| 1 | 1 | 0 | Read | Read | Read only |
| 1 | 1 | 1 | ~ | ~~ | Reserved |

# Memory Attributes

- TEX, C, and B bits within the page table entry are used to set the memory attributes of a page and the cache policies

*Table 3-3:* **Memory Attributes Encodings**

| TEX [2:0] | C | B | Description | Memory Type |
|---|---|---|---|---|
| 0 | 0 | 0 | Strongly-ordered | Strongly ordered |
| 0 | 0 | 1 | shareable device | Device |
| 0 | 1 | 0 | Outer and Inner write-through, no allocate on write | Normal |
| 0 | 1 | 1 | Outer and Inner write-back, no allocate on write | Normal |
| 1 | 0 | 0 | Outer and Inner non-cacheable | Normal |
| 1 | - | - | Reserved | - |
| 10 | 1 | 0 | Non-Shareable device | Device |
| 10 | - | - | Reserved | - |
| 11 | - | - | Reserved | - |
| 1XX | Y | Y | Cached memory<br>XX - Outer Policy<br>YY - Inner Policy | Normal |

*Table 3-4:* **Memory Attributes Encodings**

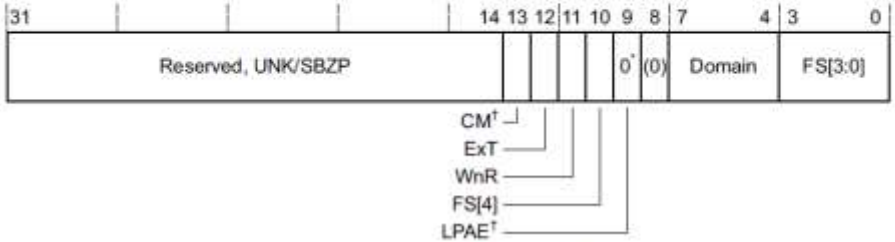
| Encoding Bits | | Cache Attribute |
|---|---|---|
| C | B | |
| 0 | 0 | Non-Cacheable |
| 0 | 1 | Write-Back, Write-Allocate |
| 1 | 0 | Write-Through, no Write-Allocate |
| 1 | 1 | Write-Back, no Write-Allocate |

# Data Abort Exceptions

- On taking a data abort exception to a PL1 mode, DFSR is updated with details of the fault

**DFSR format when using the Short-descriptor translation table format**

In a VMSAv7 implementation that does not include the Large Physical Address Extension, or in an implen that includes the Large Physical Address Extension when address translation is using the Short-descriptor translation table format, the DFSR bit assignments are:




† Only on an implementation that includes the Large Physical Address Extension. For more information, see the field description.

* Returned value, but might be overwritten, because the bit is RW.

**Bits[31:14]** Reserved, UNK/SBZP.

**CM, bit[13], if implementation includes the Large Physical Address Extension**

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maint operation generated the fault. The possible values of this bit are:

**0** Abort not caused by a cache maintenance operation.

**1** Abort caused by a cache maintenance operation.

On an asynchronous fault, this bit is UNKNOWN.

**Bit[13], if implementation does not include the Large Physical Address Extension**

Reserved, UNK/SBZP.

**ExT, bit[12]** External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

In an implementation that does not provide any classification of external aborts, this bit is UNK/SBZP.

**WnR, bit[11]** Write not Read bit. On a synchronous exception, indicates whether the abort was caused by a write or a read access. The possible values of this bit are:

**0** Abort caused by a read access.

**1** Abort caused by a write access.

For synchronous faults on CP15 cache maintenance operations, including the address translation operations, this bit always returns a value of 1.

This bit is UNKNOWN on:

- an asynchronous Data Abort exception
- a Data Abort exception caused by a debug exception.

**FS, bits[10, 3:0]**

Fault status bits. For the valid encodings of these bits when using the Short-descriptor translation table format, see Table B3-23 on page B3-1415. All encodings not shown in the table are reserved.

**LPAE, bit[9], if the implementation includes the Large Physical Address Extension**

On taking a Data Abort exception, this bit is set to 0 to indicate use of the Short-descriptor translation table formats.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation. Unless the register has been updated to report a fault, a subsequent read of the register returns the value written to it.

**Domain, bits[7:4]**

The domain of the fault address.

ARM deprecates any use of this field, see *The Domain field in the DFSR* on page B3-1415.

This field is UNKNOWN on a Data Abort exception:

- caused by a debug exception
- caused by a Permission fault in an implementation includes the Large Physical Address Extension.

**Accessing the DFSR**

To access the DFSR, software reads or writes the CP15 registers with <opc1> set to 0, <CRn> set to c5, <CRm> set to c0, and <opc2> set to 0. For example:

```
MRC p15, 0, <Rt>, c5, c0, 0    ; Read DFSR into Rt
MCR p15, 0, <Rt>, c5, c0, 0    ; Write Rt to DFSR
```

# FS[4:0]

Table B3-23 Short-descriptor format FSR encodings

| FS | Source | | Notes |
|---|---|---|---|
| 00001 | Alignment fault | | DFSR only. Fault on first lookup |
| 00100 | Fault on instruction cache maintenance | | DFSR only |
| 01100<br>01110 | Synchronous external abort on translation table walk | First level<br>Second level | - |
| 11100<br>11110 | Synchronous parity error on translation table walk | First level<br>Second level | - |
| 00101<br>00111 | Translation fault | First level<br>Second level | MMU fault |
| 00011[a]<br>00110 | Access flag fault | First level<br>Second level | MMU fault |
| 01001<br>01011 | Domain fault | First level<br>Second level | MMU fault |
| 01101<br>01111 | Permission fault | First level<br>Second level | MMU fault |
| 00010 | Debug event | | See *About debug events* on page C3-2036 |
| 01000 | Synchronous external abort | | - |
| 10000 | TLB conflict abort | | See *TLB conflict aborts* on page B3-1380 |
| 10100 | IMPLEMENTATION DEFINED | | Lockdown |
| 11010 | IMPLEMENTATION DEFINED | | Coprocessor abort |
| 11001 | Synchronous parity error on memory access | | - |
| 10110 | Asynchronous external abort[b] | | DFSR only |
| 11000 | Asynchronous parity error on memory access[c] | | DFSR only |

a. Previously, this encoding was a deprecated encoding for Alignment fault. The extensive changes in the memory model in VMSAv7 mean there should be no possibility of confusing the new use of this encoding with its previous use

b. Including asynchronous data external abort on translation table walk or instruction fetch.

c. Including asynchronous parity error on translation table walk.

# DFAR (Data Fault Address Register)

### B4.1.51 DFAR, Data Fault Address Register, VMSA

The DFAR characteristics are:

**Purpose** The DFAR holds the VA of the faulting address that caused a synchronous Data Abort exception.

This register is part of the PL1 Fault handling registers functional group.

**Usage constraints** Only accessible from PL1 or higher.

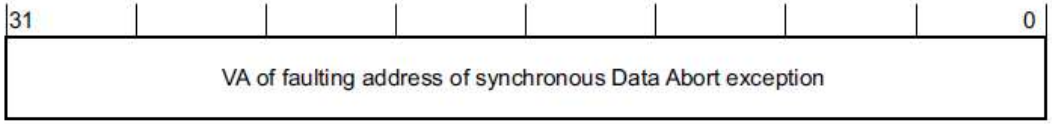
**Configurations** If the implementation includes the Security Extensions, this register is Banked.

Before ARMv7 the DFAR was called the Fault Address Register (FAR).

**Attributes** A 32-bit RW register with an UNKNOWN reset value. See also *Reset behavior of CP14 and CP15 registers* on page B3-1450.

Table B3-46 on page B3-1494 shows the encodings of all of the registers in the PL1 Fault handling registers functional group.

The DFAR bit assignments are:




For information about using the DFAR, and when the value in the DFAR is valid, see *Exception reporting in a VMSA implementation* on page B3-1409.

A debugger can write to the DFAR to restore its value.

#### Accessing the DFAR

To access the DFAR, software reads or writes the CP15 registers with <opc1> set to 0, <CRn> set to c6, <CRm> set to c0, and <opc2> set to 0. For example:

```
MRC p15, 0, <Rt>, c6, c0, 0    ; Read DFAR into Rt
MCR p15, 0, <Rt>, c6, c0, 0    ; Write Rt to DFAR
```

# IFAR (Instruction Fault Address Register)

### B4.1.95 IFAR, Instruction Fault Address Register, VMSA

The IFAR characteristics are:

**Purpose** The IFAR holds the VA of the faulting access that caused a synchronous Prefetch Abort exception.

This register is part of the PL1 Fault handling registers functional group.

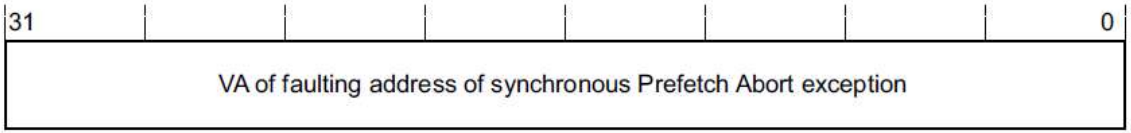
**Usage constraints** Only accessible from PL1 or higher.

**Configurations** If the implementation includes the Security Extensions, this register is Banked.

**Attributes** A 32-bit RW register with an UNKNOWN reset value. See also *Reset behavior of CP14 and CP15 registers* on page B3-1450.

Table B3-46 on page B3-1494 shows the encodings of all of the registers in the PL1 Fault handling registers functional group.

The IFAR bit assignments are:




For information about using the IFAR see *Exception reporting in a VMSA implementation* on page B3-1409.

A debugger can write to the IFAR to restore its value.

#### Accessing the IFAR

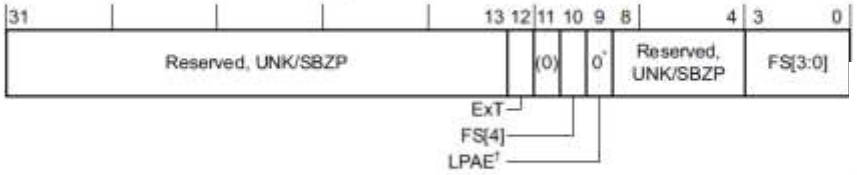
To access the IFAR, software reads or writes the CP15 registers with <opc1> set to 0, <CRn> set to c6, <CRm> set to c0, and <opc2> set to 2. For example:

```
MRC p15, 0, <Rt>, c6, c0, 2    ; Read IFAR into Rt
MCR p15, 0, <Rt>, c6, c0, 2    ; Write Rt to IFAR
```

# IFSR (Instruction Fault Status Register)

**IFSR format when using the Short-descriptor translation table format**

In a VMSAv7 implementation that does not include the Large Physical Address Extension, or in an implementation that includes the Large Physical Address Extension when address translation is using the Short-descriptor translation table format, the IFSR bit assignments are:




† Only on an implementation that includes the Large Physical Address Extension. For more information, see the field description.

* Returned value, but might be overwritten, because the bit is RW.

**Bits[31:13]** Reserved, UNK/SBZP.

**ExT, bit[12]** External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of external aborts.

For aborts other than external aborts this bit always returns 0.

In an implementation that does not provide any classification of external aborts, this bit is UNK/SBZP.

**Bit[11]** Reserved, UNK/SBZP.

**FS, bits[10, 3:0]**

Fault status bits. For the valid encodings of these bits when using the Short-descriptor translation table format, see Table B3-23 on page B3-1415. All encodings not shown in the table are reserved.

**LPAE, bit[9], if the implementation includes the Large Physical Address Extension**

On taking an exception, this bit is set to 0 to indicate use of the Short-descriptor translation table format.

Hardware does not interpret this bit to determine the behavior of the memory system, and therefore software can set this bit to 0 or 1 without affecting operation. Unless the register has been updated to report a fault, a subsequent read of the register returns the value written to it.

**Bits[9], if the implementation does not include the Large Physical Address Extension**

Reserved, UNK/SBZP.

**Bits[8:4]** Reserved, UNK/SBZP.

## Accessing the IFSR

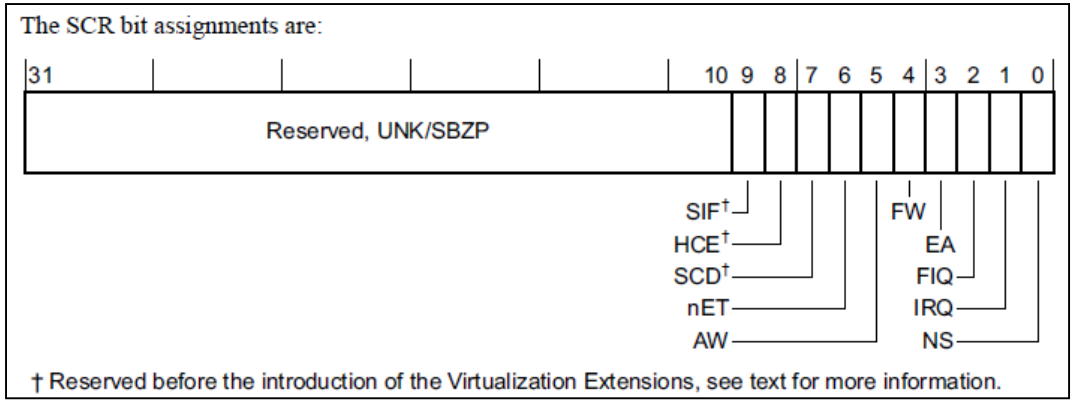
To access the IFSR, software reads or writes the CP15 registers with <opc1> set to 0, <CRn> set to c5, <CRm> set to c0, and <opc2> set to 1. For example:

```
MRC p15, 0, <Rt>, c5, c0, 1    ; Read IFSR into Rt
MCR p15, 0, <Rt>, c5, c0, 1    ; Write Rt to IFSR
```

# Secure Configuration Register (SCR)

- SCR defines the configuration of the current security state
  - Security state of the CPU: Secure or Non-secure
  - What mode CPU branches to if IRQ, FIQ or external abort occurs
  - Whether the CPSR.{F,A} can be modified when SCR.NS == 1

- **SIF: Secure Instruction Fetch**
- **HCE: Hyp Call Enable**
- **SCD: Secure Monitor Call disable**
- **nET: Not Early Termination**
- **AW: A bit writable**
- **FW: F bit writable**
- **EA: External Abort handler**
- **FIQ: FIQ handler**
- **IRQ: IRQ handler**
- **NS: Non-Secure bit**



# Example - Page table

```
. .globl  csd_MMUTable
.section .csd_mmu_tbl,"a"

csd_MMUTable:

                /* Each table entry occupies one 32-bit word and there are
                 * 4096 entries, so the entire table takes up 16KB.
                 * Each entry covers a 1MB section.
                 *
                 * The following defines only 3 1MB sections
                 *     1st 1MB: 0x0000_0000 (VA) -> 0x0000_0000 (PA)
                 *     2nd 1MB: 0x0010_0000 (VA) -> 0x0020_0000 (PA)
                 *     3rd 1MB: 0x0020_0000 (VA) -> 0x0040_0000 (PA)
                 */

.set SECT, 0
.word       SECT + 0x15de6                  /* S=b1 TEX=b101 AP=b11, Domain=b1111, C=b0, B=b1 */
.set        SECT, SECT+0x200000
.word       SECT + 0x15de6                  /* S=b1 TEX=b101 AP=b11, Domain=b1111, C=b0, B=b1 */
.set        SECT, SECT+0x200000
.word       SECT + 0x15de6                  /* S=b1 TEX=b101 AP=b11, Domain=b1111, C=b0, B=b1 */
.end
```

# Example - 2nd level page table

```
.globl  csd_MMUTable_lv2
.section .csd_mmu_tbl_lv2,"a"

csd_MMUTable_lv2:
 .word 0xAAAAA002                  // Create your 2nd level page table



.globl  csd_MMUTable
.section .csd_mmu_tbl,"a"

csd_MMUTable:
/* Each table entry occupies one 32-bit word and there are
 * 4096 entries, so the entire table takes up 16KB.
 * Each entry covers a 1MB section.
 *
 *    1st 1MB (0x0 ~ 0xF_FFFF) in VA → 0x0 ~ 0xFFFFF in PA
 *    2nd 1MB -> Second Table?
 *    3rd 1MB (0x20_0000 ~ 0x2F_FFFF) in VA → 0x40_0000 ~ 0x4F_FFFF in PA
 */
// 00(0000_0000) 01(0000_0001) 5d(0101_1101) e6(1110_0110)

.set  SECT, 0
.word  SECT + 0x15de6                  /* S=b1 TEX=b101 AP=b11, Domain=b1111, C=b0, B=b1 */
.set SECT, SECT+0x100000
.word  csd_MMUTable_lv2 + 0x1e1
.set  SECT, 0x400000
.word  SECT + 0x15de6                  /* S=b1 TEX=b101 AP=b11, Domain=b1111, C=b0, B=b1 */
```