Nailgun Defense

Chenxu Wang and Fengwei Zhang

Outline

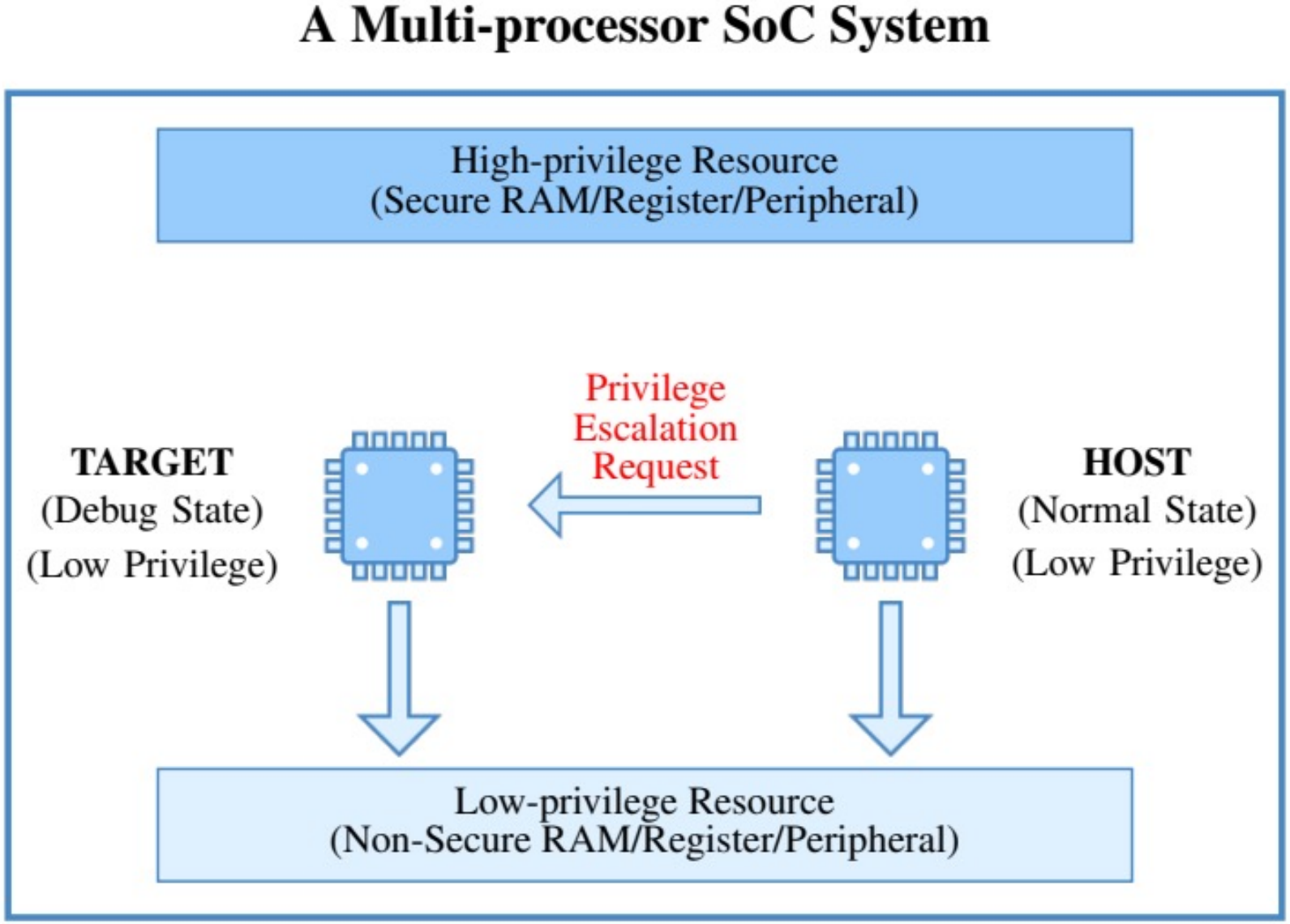
1. Introduction
2. Background
3. Design and Implementation
4. Possible Q&A
5. Necessary materials

Introduction: Nailgun Attack

An attack to break the privilege isolation via debugging

HOST processor escalates the privilege of TARGET processor by sending instructions (e.g., dcps3) via memory-mapped debug registers.

Introduction: Nailgun Attack



Source: Nailgun Attack Lecture slides

Introduction: How to defend?

Disabling the debug authentication signals? Great challenges!

1. Heavily influence most debug tools
2. Management of debug authentication is not publicly available

…

Introduction: How to defend?

Strict restriction in the inter-processor debugging?

Hardware-based access control to the debug registers?

Manufactures can implement it on the future devices.

But a great callback of current devices is expensive.

How about some software-based access control?

Introduction: Our defense

Implement the defense on higher privilege.

Control the access of the debug registers via memory mapping.

Fortunately, Arm provides Stage-2 translation.

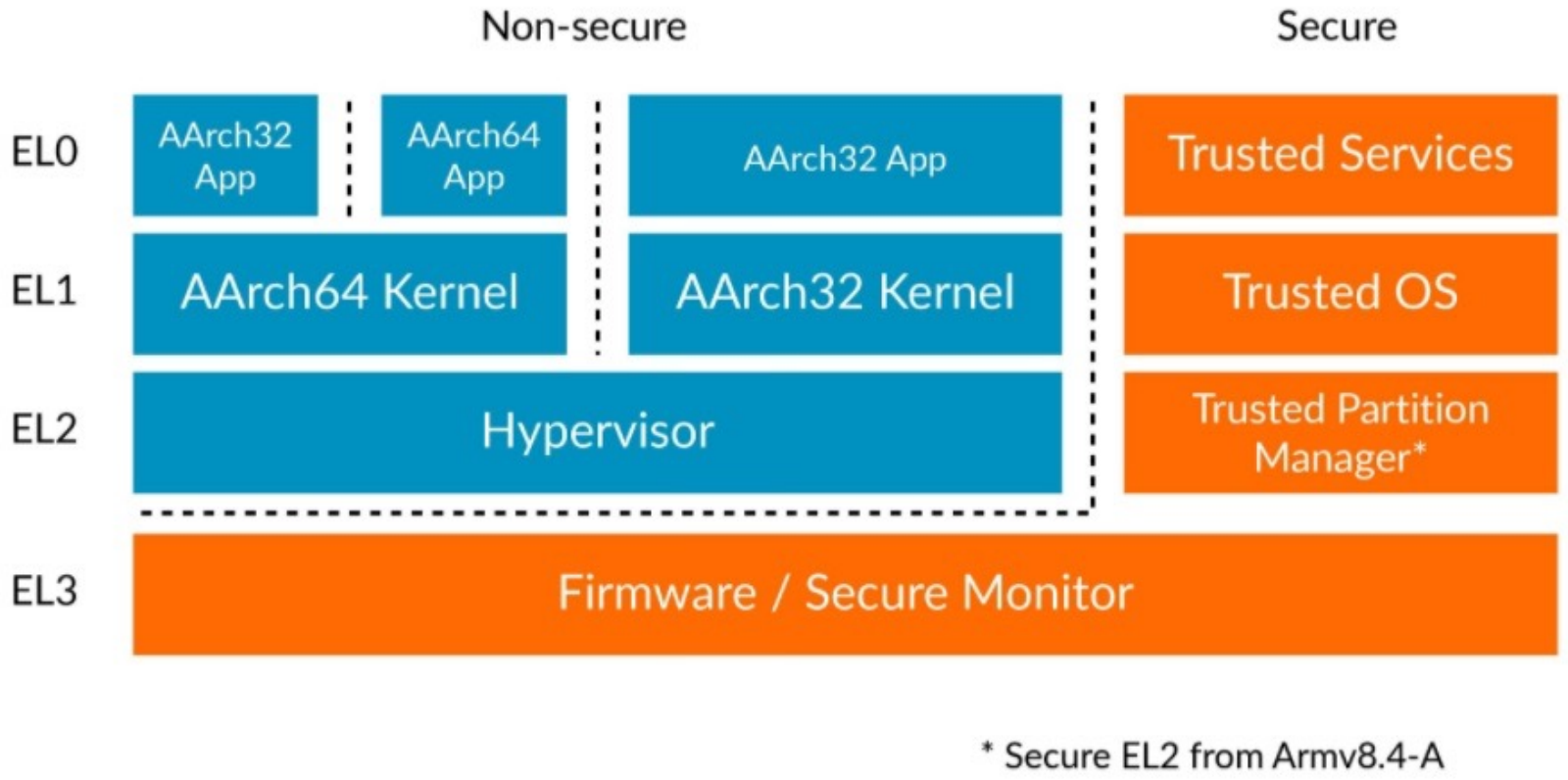
It is controlled by EL2, and restrict the memory access in EL1&0.

Background

Background: Arm Exception Levels

Two worlds, Four ELs with different privileges.

Different CPUs can be in different ELs.



*Source: https://developer.arm.com/documentation/102412/0100/Execution-and-Security-states

Background: Arm Address Translation

Arm introduces three types of address:

Virtual Address (VA)

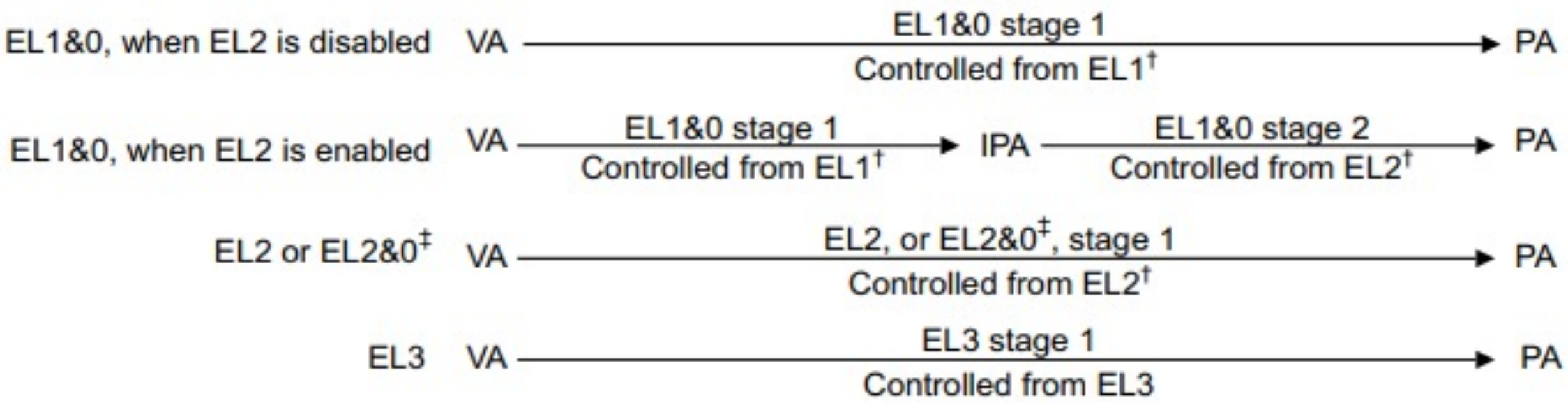
Intermediate Physical Address (IPA)

Physical Address (PA)

It can be used to isolate the process' address space, access control, etc.

Background: Arm Address Translation

Arm also defines the address translation regimes.



*Source: D5-2684 (64-bit address translation), also you can see G5-6264 (32-bit address translation)

Background: Arm Address Translation

Stage 1 translation exists in all ELs.

But EL1&0 contains an additional translation: Stage-2 translation.

It is controlled by EL2.

Hypervisor in Arm architecture can use it to handle the memory management for VMs.

Similarly, we can use it in access control for EL1&0 CPUs.

Background: Stage-2 translation

How to provide the table?

VTTBR register defines the address of the S-2 translation table.

Put the address of translation table into this register.

BADDR, bits [47:1]

Translation table base address, bits[47:x], Bits [x-1:1] are RES0, with the additional requirement that if bits[x-1:3] are not all zero, this is a misaligned translation table base address, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

*Source: G8-6942

Background: Stage-2 translation

How to use it?

We (1) Enable the S-2 translation, and (2) Provide the table of S-2 translation, then Memory Management Unit (MMU) will perform the S-2 translation after the S-1 translation.

You may ask these questions:

How to enable it?

How to provide the table?

How to create the table?

Background: Stage-2 translation

How to enable it?

HCR register controls the S-2 translation.

Set the bit[0] of HCR register as 0x1.

VM, bit [0]

Virtualization enable. Enables stage 2 address translation for the Non-secure EL1&0 translation regime.

0b0	Non-secure EL1&0 stage 2 address translation disabled.
0b1	Non-secure EL1&0 stage 2 address translation enabled.

*Source: G8-6599

Background: Stage-2 translation

Actually, it does not need the specific table address.

It cuts the offset of the table address, which is provided in VTCR register.

We will configure this register later.

x is determined from the value of VTCR.SL0 and VTCR.T0SZ as follows:

- If VTCR.SL0 is `0b00`, meaning that lookup starts at level 2, then x is 14 - VTCR.T0SZ.
- If VTCR.SL0 is `0b01`, meaning that lookup starts at level 1, then x is 5 - VTCR.T0SZ.
- If VTCR.SL0 is either `0b10` or `0b11` then a stage 2 level 1 Translation fault is generated.

*Source: G8-6942

Background: Stage-2 translation

How to create the table?

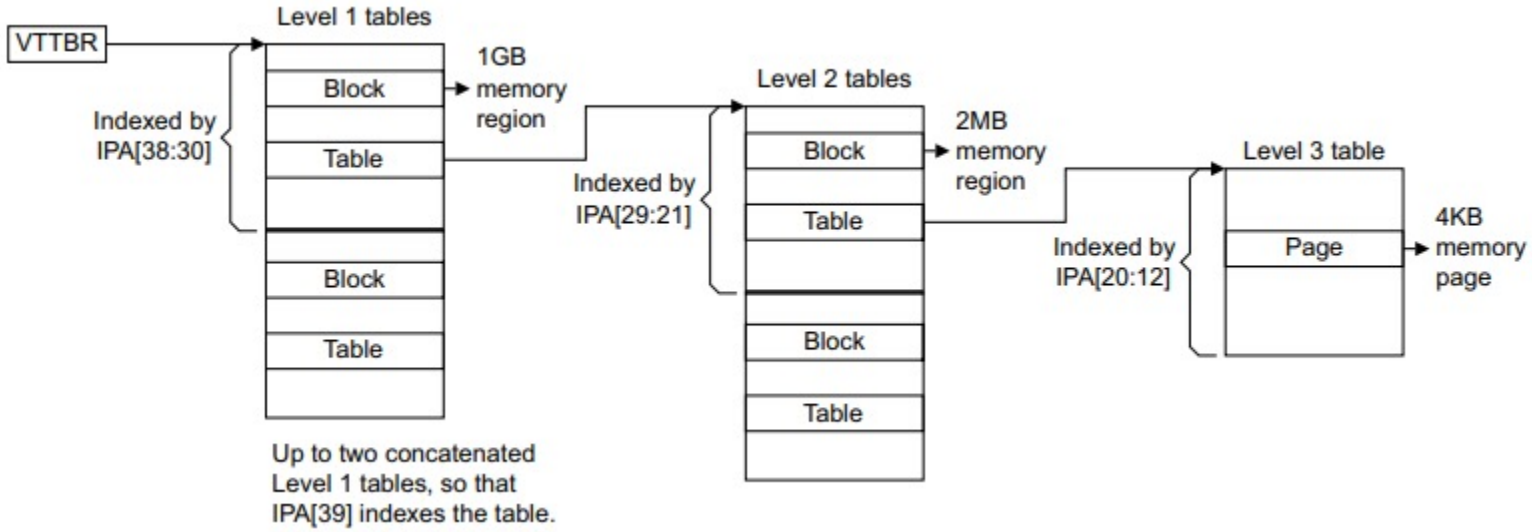
The translation table consists of several levels and page entries (descriptors).
Each descriptor indicates the attributes of a region.

Attributes: Read, Write, Execute, Cacheable…

Region: the size is decided by the level of the descriptor.
Level1: 1GB region
Level2: 2MB region
Level3: 4KB region

Background: Stage-2 translation

It shows the architecture of the translation table, the “Block” “Table” “Page” are different types of the descriptors.

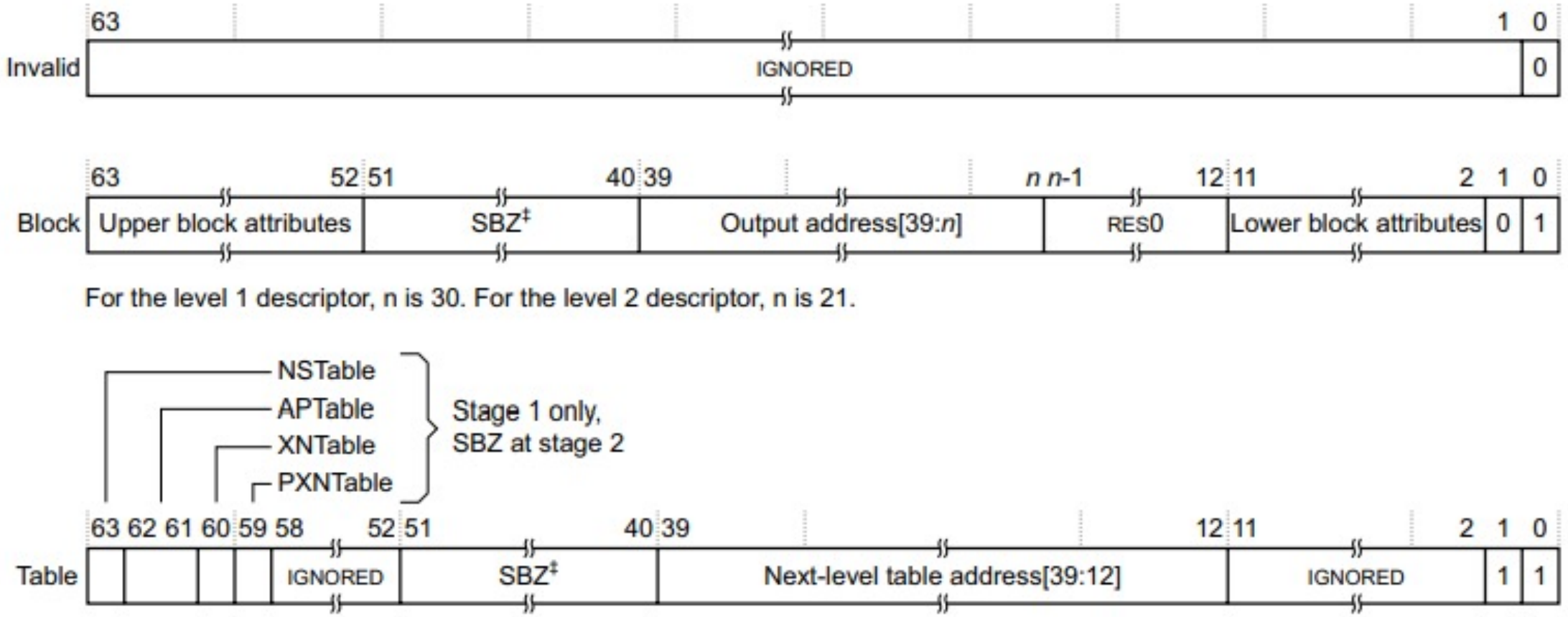


*Source: G5-6289

Background: Stage-2 translation

When we get a descriptor, it looks like this:

(Stage-2 translation uses Long descriptor, be careful when read manual.)



*Source: G5-6290

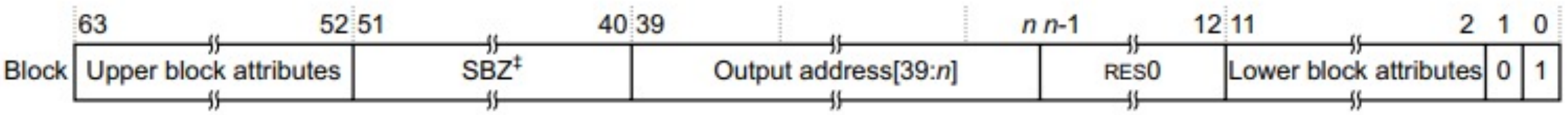
Background: Stage-2 translation

In a descriptor, the last two bits indicate the type of the descriptor.

Block (Level 1,2)/Page (Level 3): translation end, directly get the output

Invalid: translation fault

Table (Level 1,2): translation continue, go to next level.



For the level 1 descriptor, n is 30. For the level 2 descriptor, n is 21.

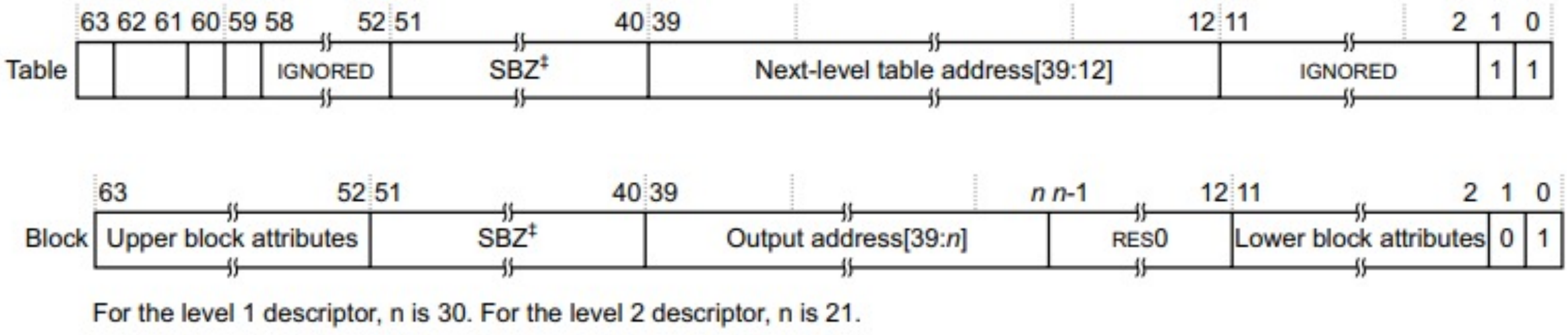
*Source: G5-6290

Background: Stage-2 translation

It contains some attributes (which is not important in this Lab).

It indicates the output address, or the address of the next descriptor.

To get them, add the provided address (e.g., bit[39:12]) and the offset of your input address (e.g., bit[11:0]).



*Source: G5-6290, SBZ=should be zero, means fill 0; RES0: reserved as 0, means fill 0; IGNORED: fill anything you like

Background: Stage-2 translation

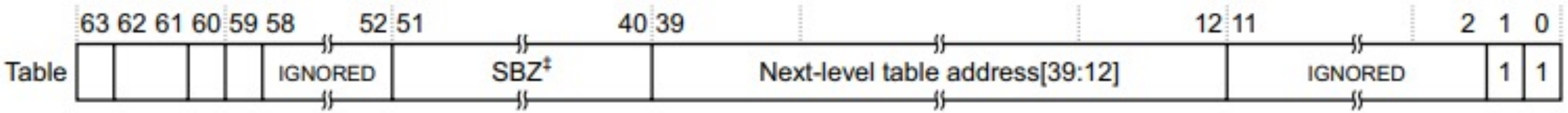
Example:

Your Input address is 0x12345678

Bit[39:12] (Next-level table address) is 0xdeadb

Then next address (0xdeadb<<12)+0x678=0xdeadb678.

Read it to get the value of the descriptor.



*Source: G5-6290, SBZ=should be zero, means fill 0; RES0: reserved as 0, means fill 0; IGNORED: fill anything you like

Design

Design

Goals in this Lab:

Enable the Stage-2 translation.

Invalid the mapping of the debug registers.

Invalid the mapping of S-2 translation table.

Design: Enable Stage-2 translation

Modified the device tree files to reserve a memory for the table.

In kernel booting stages, creating the S-2 translation table.

And configure related registers.

Replace the kernel image and compiled device tree files (.dtb) in “/boot” directory.

Boot your Raspberry Pi.

Design: Invalidate Mapping

The mapped memory of the debug registers is 0x4003_0000 ~ 0x4003_0FFF.

It is a 4KB region, you should split your table to 4KB-size granule.

Also, do not forget to invalidate the mapping of your table.

It just affect the EL1 attacker to read it, but not affect the EL2.

Design: Example

Here is one example of the table layout in 0x0 ~ 0xFFFF_FFFF (only invalid dbg)

VTTBR: point to area0

area0:
0x0000_0000 ~ 0x3FFF_FFFF: 1GB block
0x4000_0000 ~ 0x7FFF_FFFF: table, point to area1
0x8000_0000 ~ 0xBFFF_FFFF: 1GB block
0xC000_0000 ~ 0xFFFF_FFFF: 1GB block

area1:
0x4000_0000 ~ 0x401F_FFFF: table, point to area2
0x4020_0000 ~ 0x403F_FFFF: 2MB block
0x4040_0000 ~ 0x405F_FFFF: 2MB block
...
0x7E00_0000 ~ 0x7FFF_FFFF: 2MB block

area2:
0x4000_0000 ~ 0x4000_0FFF: 4KB Page
...
0x4002_F000 ~ 0x4002_FFFF: 4KB Page
0x4003_0000 ~ 0x4003_0FFF: Invalid (0x0)
0x4003_1000 ~ 0x4003_1FFF: 4KB Page
0x4003_2000 ~ 0x4003_2FFF: 4KB Page
...
0x401F_0000 ~ 0x401F_FFFF: 4KB Page

Design: Example Translation Table Walk

Here we give an example of a successful address translation.

You can see the whole translation process in K7-8498.

We recommend you configure these bits of VTCR registers

SL0[7:6]: 0x1 (Starting at Level 1)

T0SZ[3:0]: 0x0 (Indicating the PA size is 2^32-1)

SL0, bits [7:6]

Starting level for translation table walks using VTTBR.

0b00	Start at level 2
0b01	Start at level 1

T0SZ, bits [3:0]

The size offset of the memory region addressed by VTTBR. The region size is $2^{(32-T0SZ)}$ bytes.

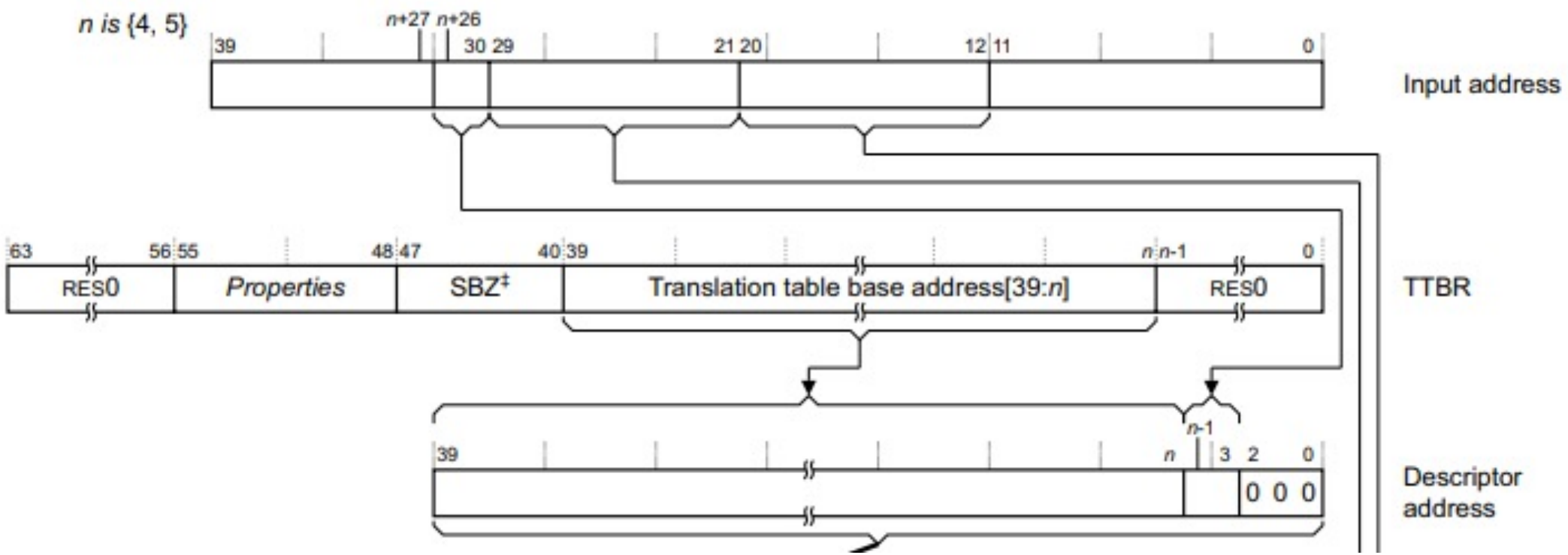
This field holds a four-bit signed integer value, meaning it supports values from -8 to 7.

Design: Example Translation Table Walk

When we translate the IPA 0x4003_1234, starting at Level 1.

(1) We do Level 0 translation, and VTTBR tells us to go to area 0.

(2) We go to area 0 and get the descriptor of 0x4000_0000 ~ 0x7FFF_FFFF



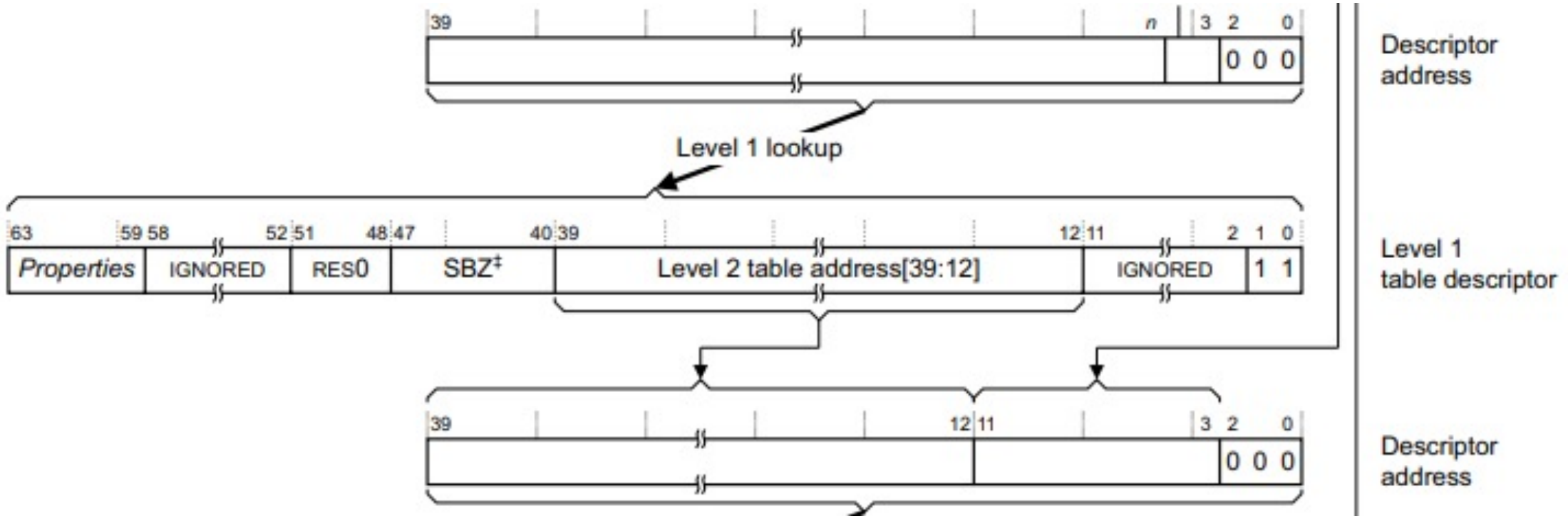
*Source: K7-8498

Design: Example Translation Table Walk

When we translate the IPA 0x4003_1234, starting at Level 1.

(3) We do Level 1 lookup of the descriptor 0x4000_0000 ~ 0x7FFF_FFFF, it is a table, and we should go to area2 to find the next descriptor.

(4) We go to area2 and get the descriptor of 0x4000_0000 ~ 0x401F_FFFF



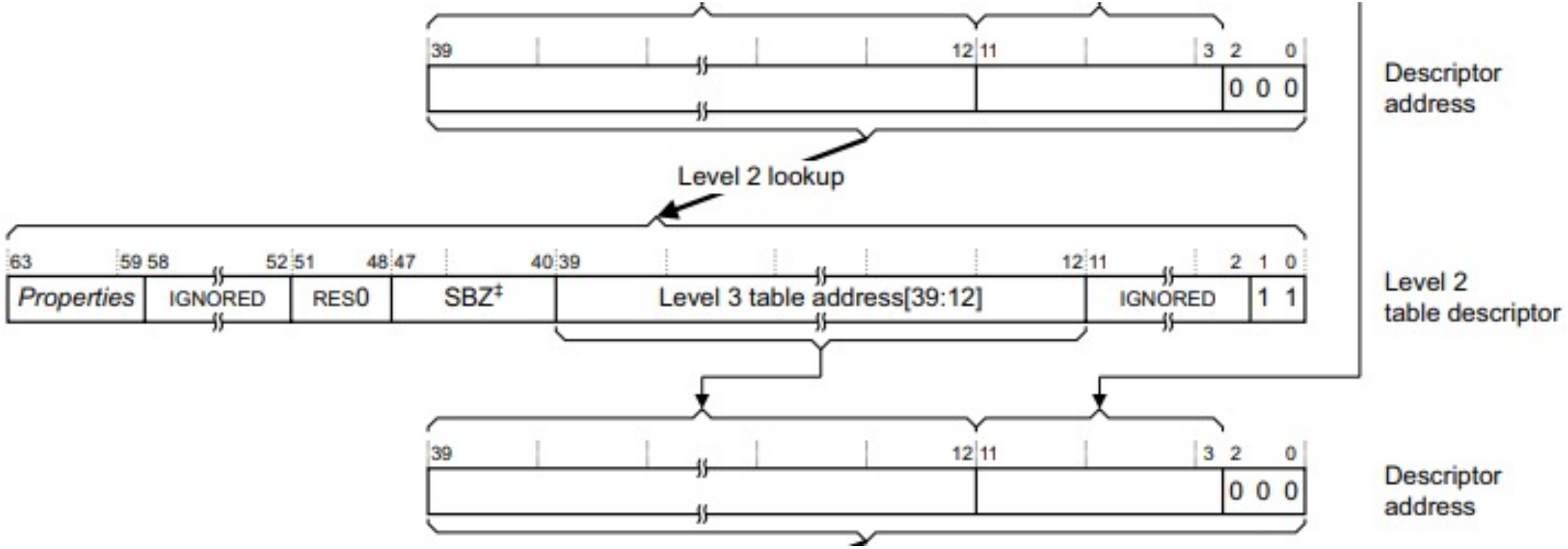
*Source: K7-8498

Design: Example Translation Table Walk

When we translate the IPA 0x4003_1234, starting at Level 1.

(5) We do Level 2 lookup of the descriptor 0x4000_0000 ~ 0x401F_FFFF, it is a table, and we should go to area3 to find the next descriptor.

(6) We go to area3 and get the descriptor of 0x4003_0000 ~ 0x4003_0FFF



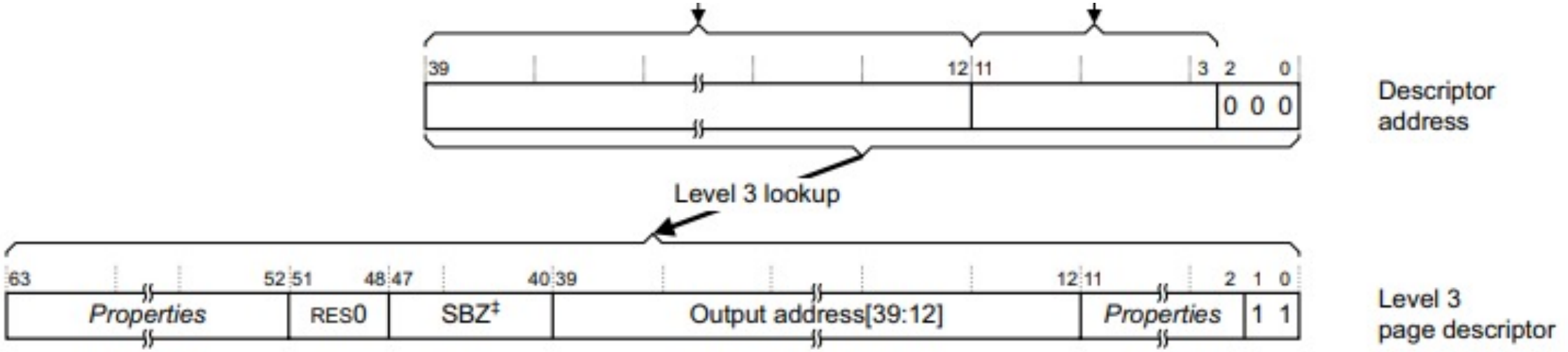
*Source: K7-8498

Design: Example Translation Table Walk

When we translate the IPA 0x4003_1234, starting at Level 1.

(7) We do Level 3 lookup of the descriptor 0x4003_0000 ~ 0x4003_0FFF, it is a page.

(8) Our translation result is (OA[39:12]<<12)+0x123



*Source: K7-8498

Possible Q&A

Possible Q&A

The system crashes when I load the Nailgun attack, why?

When we access the invalid region, it will trigger a translation fault.

But we have not designed the corresponding exception handler.

So the system cannot jump to the correct PC to exit the attack.

Possible Q&A

How about the cache and TLB attacks?

Actually, this Lab have not considered these attacks.

You can call them as “side-channel attack”, which is a popular topic.

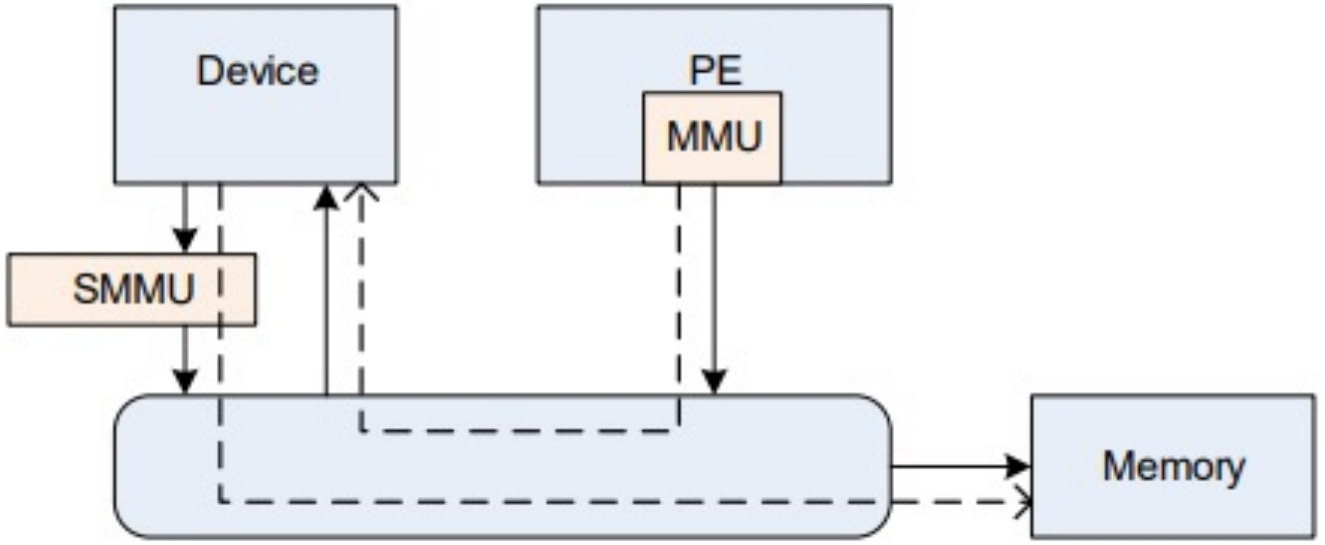
We may extend the discussion in the future.

Possible Q&A

How about peripheral attacks?

Attacker can use the peripherals, such as GPU and DMA, to bypass the MMU (Specifically, the MMU for CPU).

But we can config the MMU for them with S-2 translation, which is called as SMMU (or IOMMU).



*Source: https://developer.arm.com/documentation/ihi0070/latest

Possible Q&A

Do we have technical support?

The biggest support is the reference manual.

Look up it and solve your problem.

Also, you can contact partial designers of this Lab.

12150073@mail.sustech.edu.cn (WANG Chenxu)

Necessary materials

1. Armv8-A Reference Manual

Download: https://developer.arm.com/documentation/ddi0487/latest

A dictionary of Armv8-A architecture.

Important part in this Lab: G5.1-G5.7, K7.2

(They are reference about 32-bit Armv8-A architecture)

Necessary materials

2. Linux Kernel Source file

Raspberry Pi provides the source codes for its kernel.

Download: https://github.com/raspberrypi/linux

Here we use the branch rpi-4.14.y or rpi-4.14.y-rt, you can also use the other branches.

Necessary materials

3. Cross-compile Tools

Raspberry Pi provide it: https://github.com/raspberrypi/tools

We use it because the environment of our VM and Raspberry Pi are different.

Mostly we use Ubuntu, which is based on Intel x86 architecture.

But Raspberry Pi uses Cortex-A53 cores, which support Armv8-A architecture.

*If you build Kernel files on Raspberry Pi, you will not use it.