

DEEP IN ARM VIRTUALIZATION



DETAILED IN TWO STAGES ADDRESS TRANSLATION

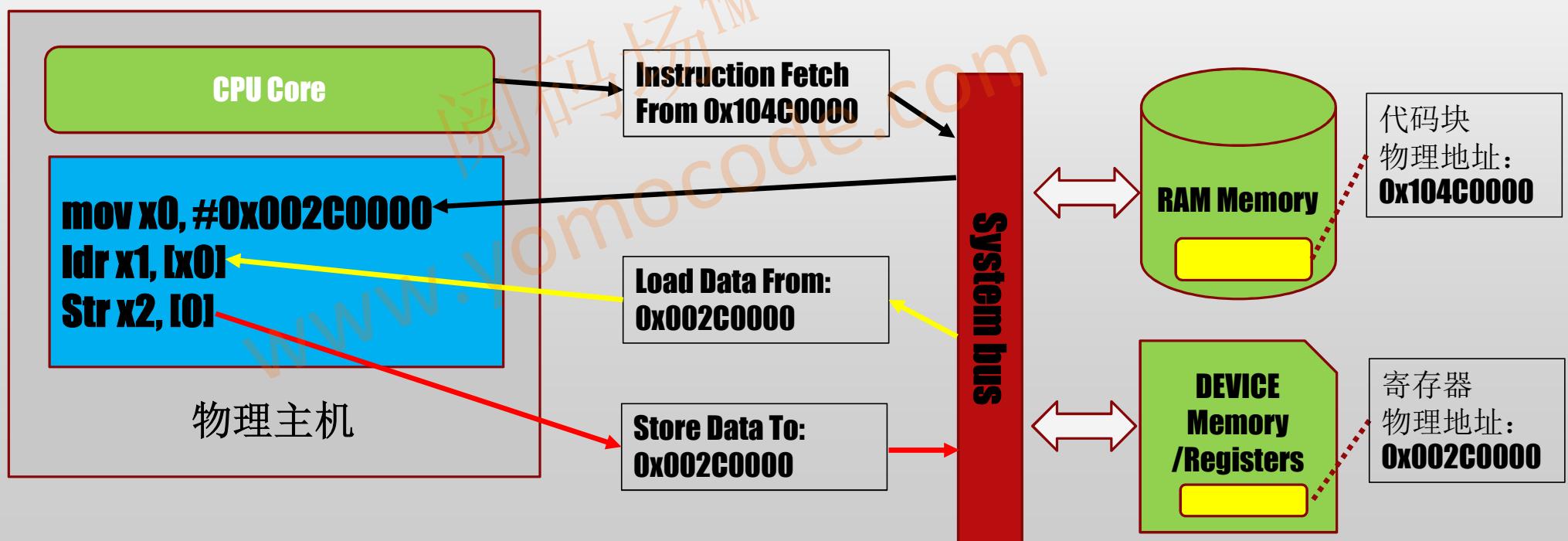
深入理解**2**级地址翻译

MEMORY ACCESS MODEL

- 程序的执行就是围绕着一序列地址访问进行的，下图一个物理机上的程序地址访问模型：

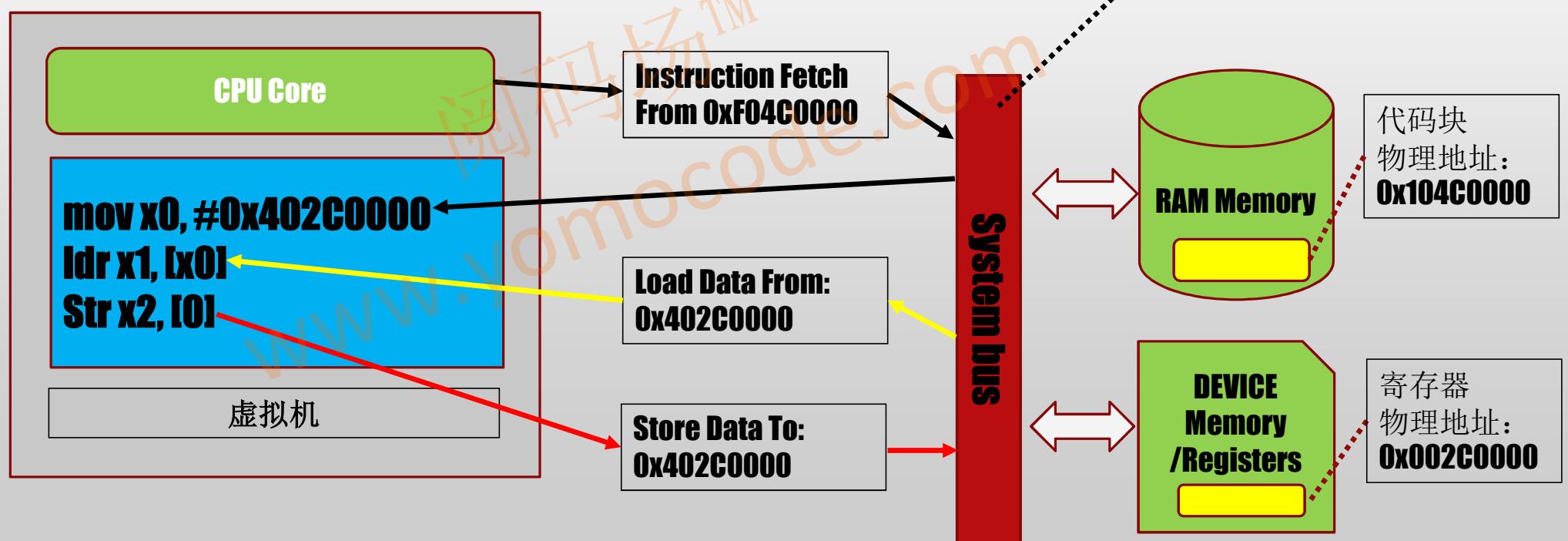
- CPU先要将程序指令从**0X104C0000**加载到**CPU**内部
- 程序执行时会从外部寄存器**0X002C0000**读取数据到**X1**寄存器
- 然后会在保存在**X2**寄存器的值写回**0X002C0000**

任何一步的地址访问错误
都会触发异常，造成程序
无法运行



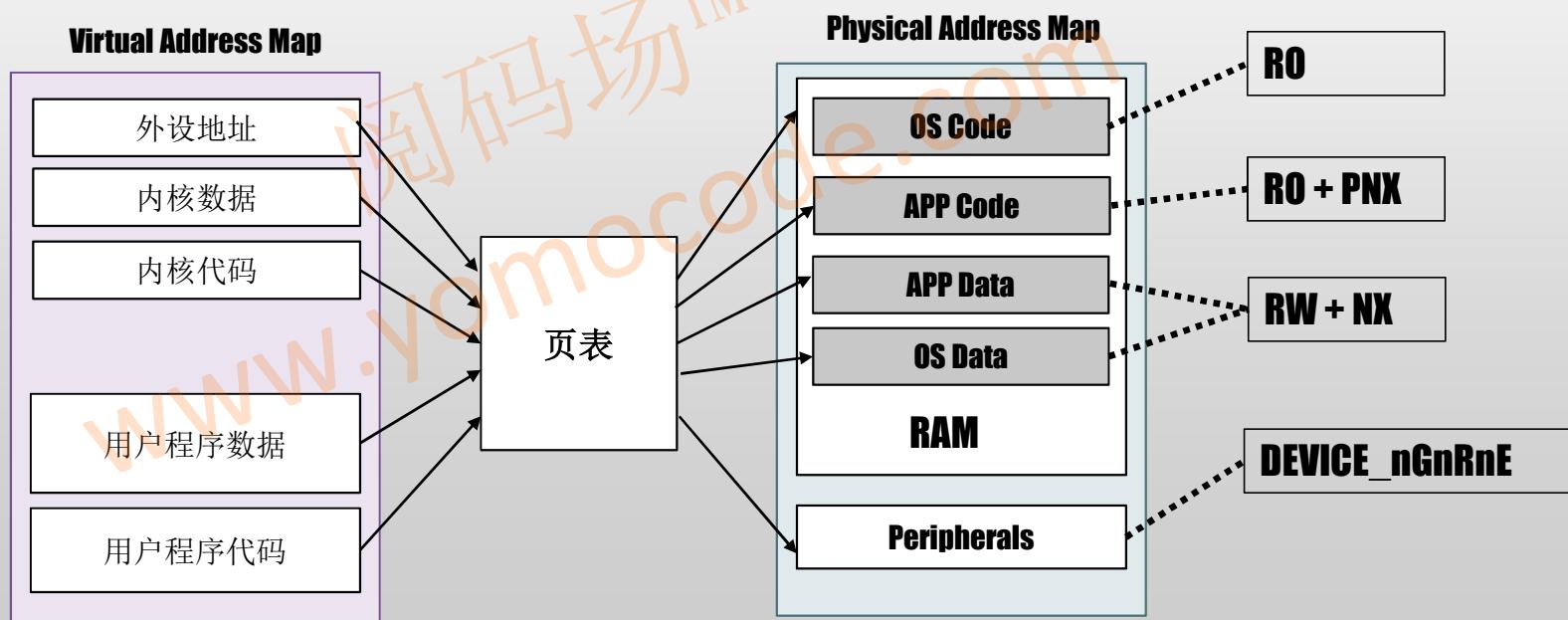
VM MEMORY ACCESS MODEL

- 虚拟机中的地址空间通常和物理主机是不一致
- 指令存放的物理内存**0x104C0000**, 在虚拟机中看到的地址可能是**0xF04C0000**
- 物理寄存器**0x002C0000**在虚拟机中可能被映射为**0x402C0000**
- 代码块的执行就会变成下图, **CPU**会按照虚拟机的地址进行存取操作



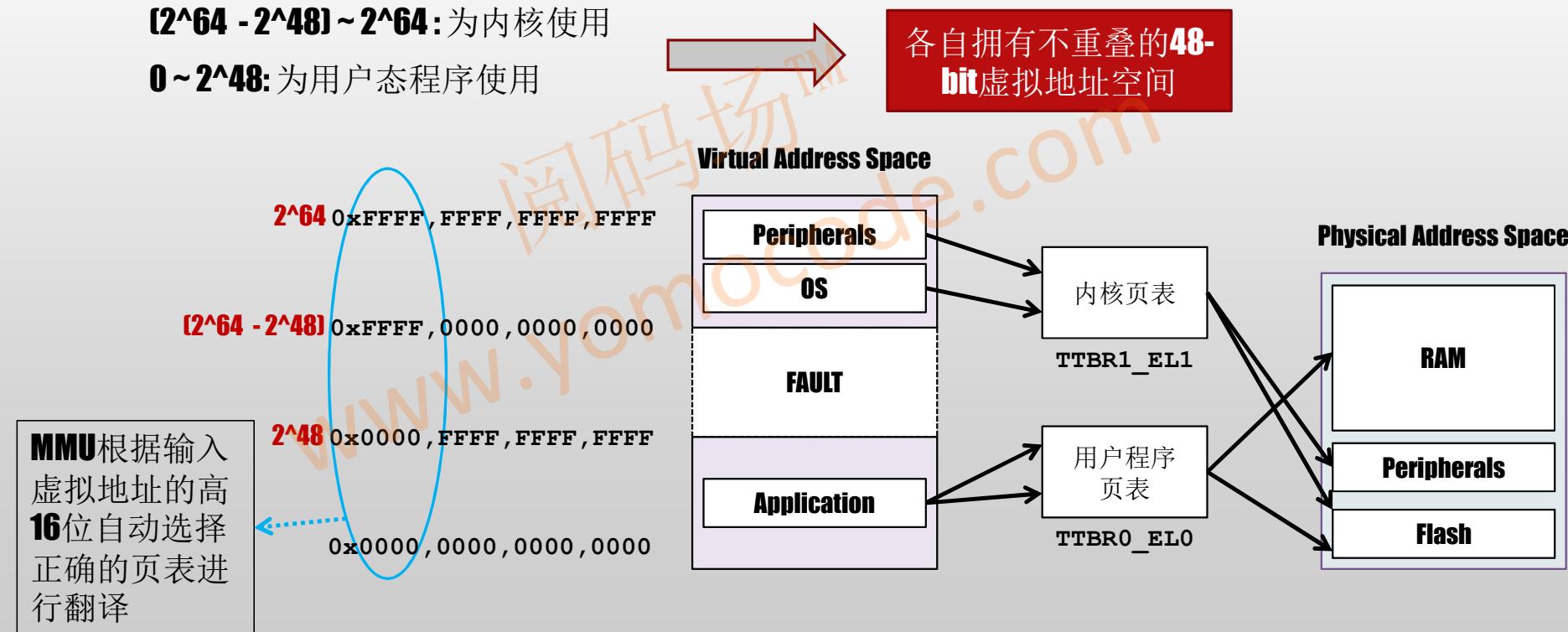
ARMV8 VMSA

- VMSA是ARMV8的虚拟内存系统架构(VIRTUAL MEMORY SYSTEM ARCHITECTURE)的缩写：
 - VMSA的核心是利用MMU(MEMORY MANAGEMENT UNIT)，来负责处理虚拟地址到物理地址的翻译
 - MMU页表定义了虚拟地址到物理地址的映射关系，同时还定义了内存的属性，比如有些内存只能执行用户态程序，有些内存不可写，有些内存是设备专用



VIRTUAL ADDRESS SPACE

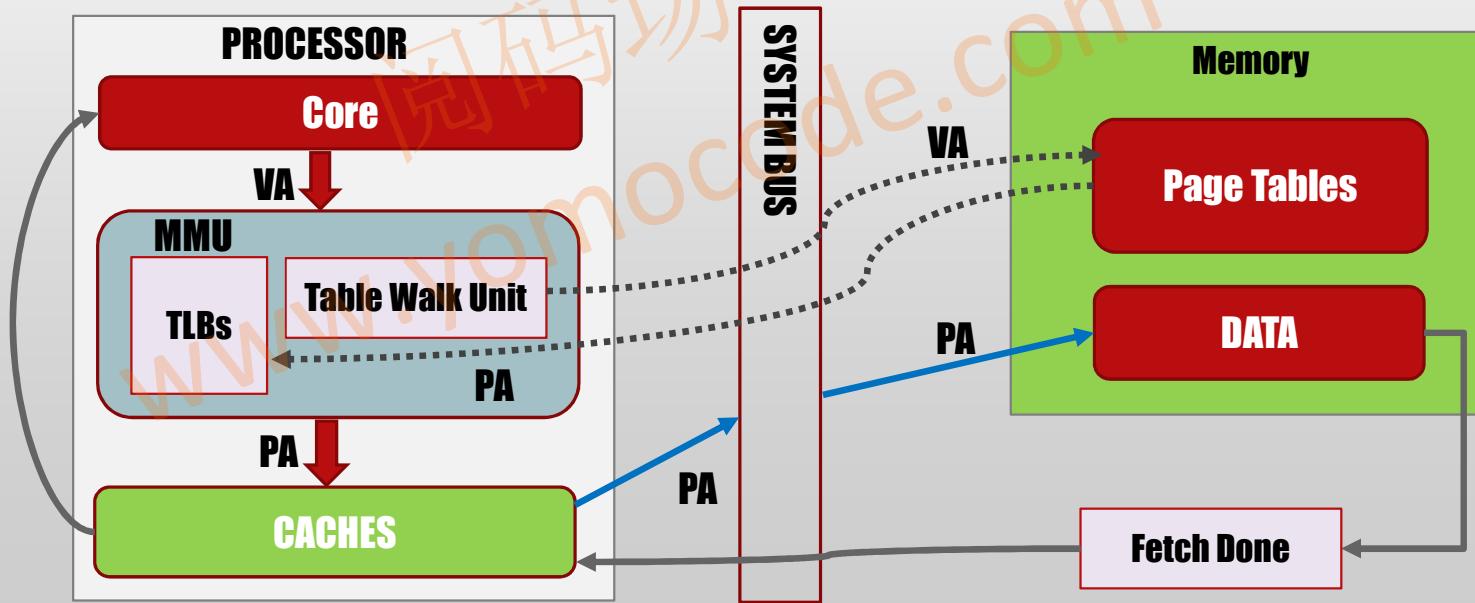
- 在ARM64中，虚拟地址宽度是**64**位的，但并不是所有的**64**位地址空间都可以访问
 - 每个系统的设计都有它最大支持的虚拟地址宽度，常用为**40-BIT**和**48-BIT** (ARMV8.2可扩展到**52-BIT**)
 - 在一个支持**48-BIT**虚拟地址的系统中，通常将虚拟地址空间分割成两段：
[2⁶⁴ - 2⁴⁸] ~ 2⁶⁴: 为内核使用
0 ~ 2⁴⁸: 为用户态程序使用



WHAT IS A MMU?

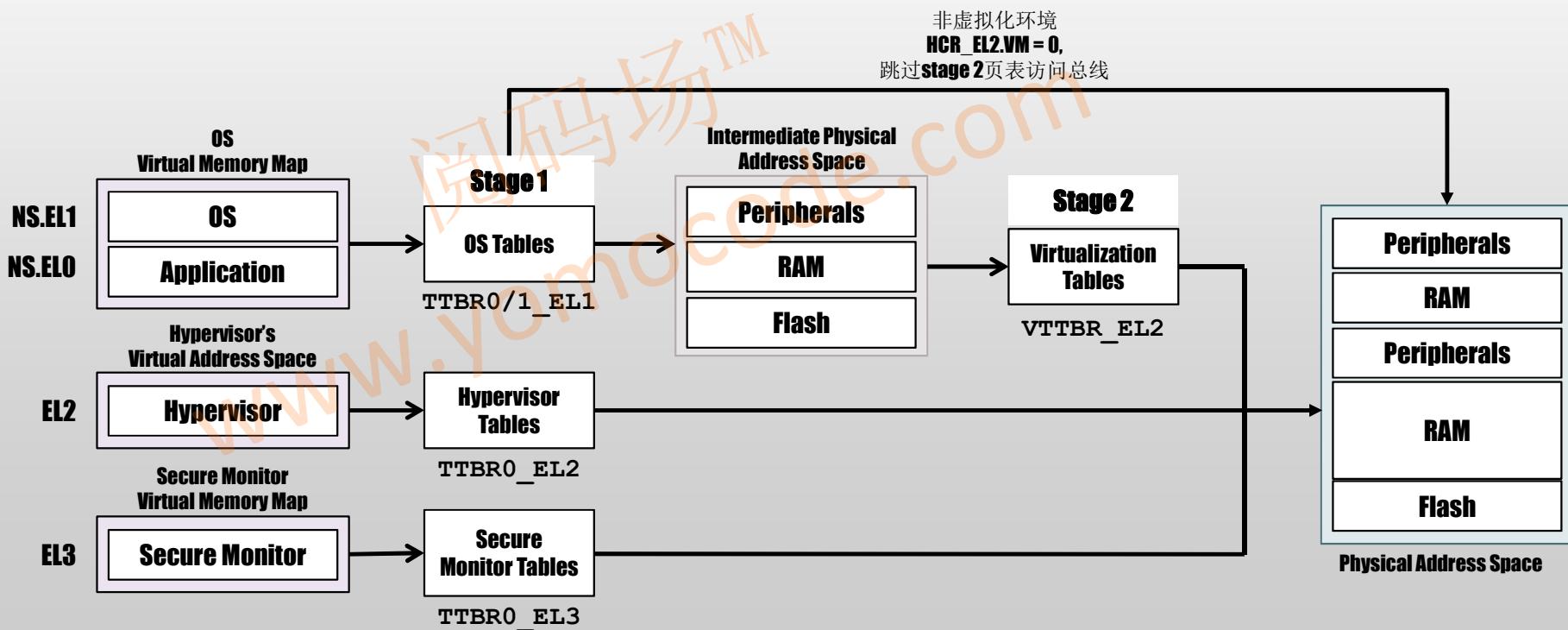
Mmu介于**CPU**核心和系统总线直接，负责将**VA**转换为**PA**:

- **TLB**用于缓存**VA**到**PA**的转换结果，**MMU**会先检查**TLB**，如果有之前保存好的结果会直接返回**PA**
- **TLB**不命中时，**Table Walk Unit**通过查询页表得到**VA**对应的**PA**，并保存到**TLB**
- **CPU**通过**MMU**得到的**PA**查询**Cache**，如果命中会直接返回数据
- **Cache**不命中，会用**PA**到系统总线进行设备访问，从内存得到真实数据并返回

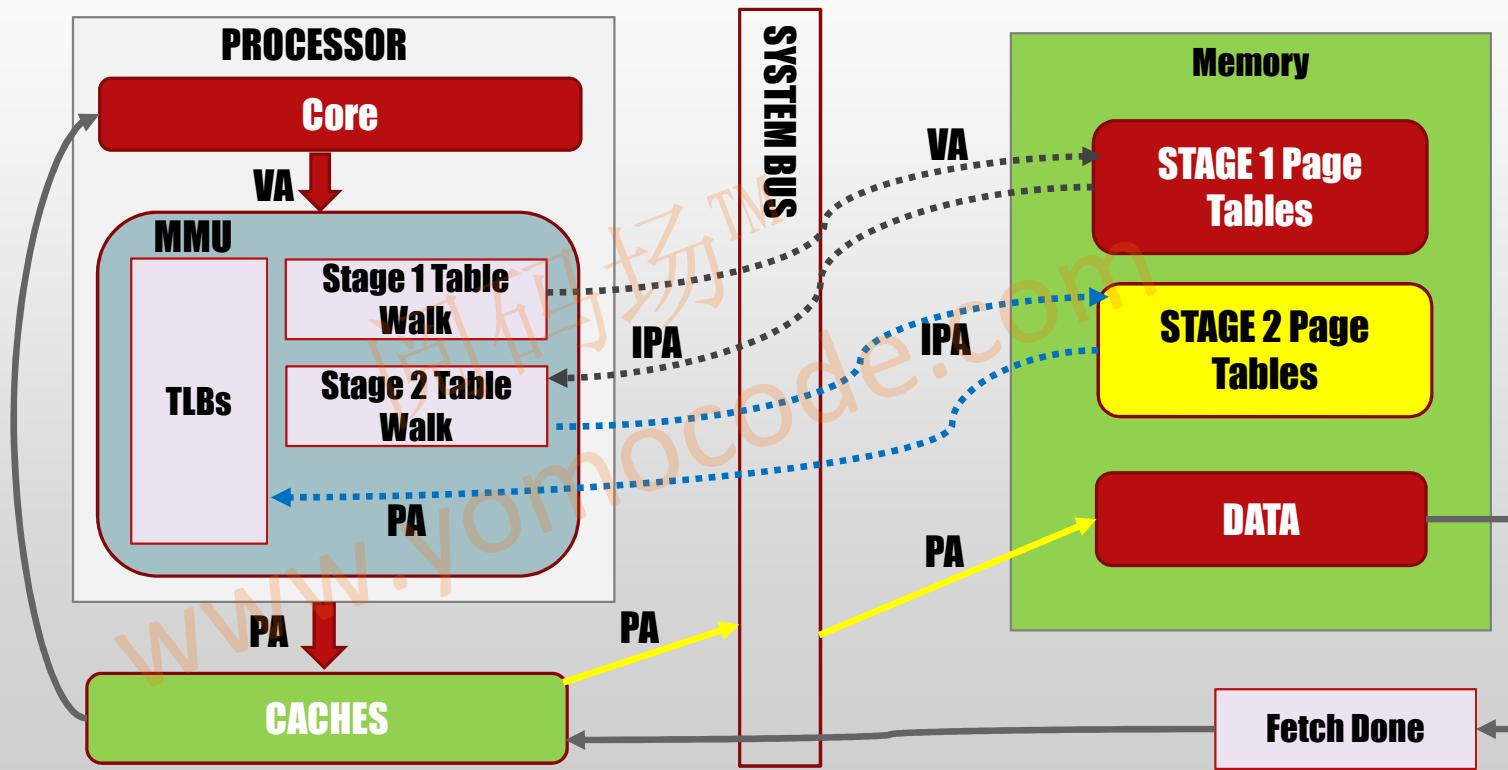


MULTIPLE TRANSLATION REGIMES

- ARM64 系统中每个 **EXCEPTION LEVEL** 都是一个独立的转换空间，都有自己独立的虚拟地址，如 **LINUX** 中：
 - **EL0** 虚拟地址转换由 **OS** 中指向 **TTBRO_EL1** 的页表控制
 - **EL1** 虚拟地址转换由 **OS** 中指向 **TTBR1_EL1** 的页表控制
 - **EL2 HYPERVISOR** 的虚拟地址转换由指向 **TTBR_EL2** 的页表控制
 - **EL3 SECURE MONITOR** 的虚拟地址转换由指向 **TTBR_EL3** 的页表控制



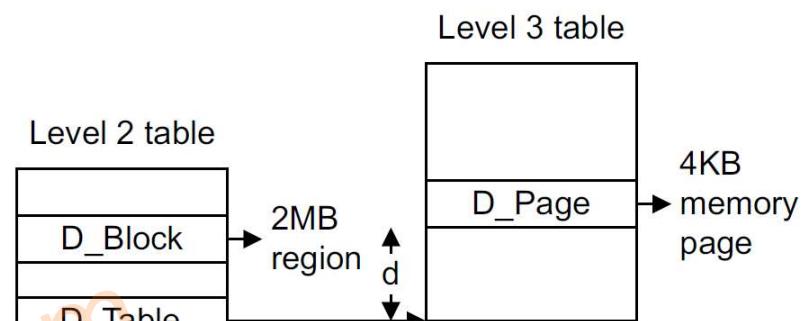
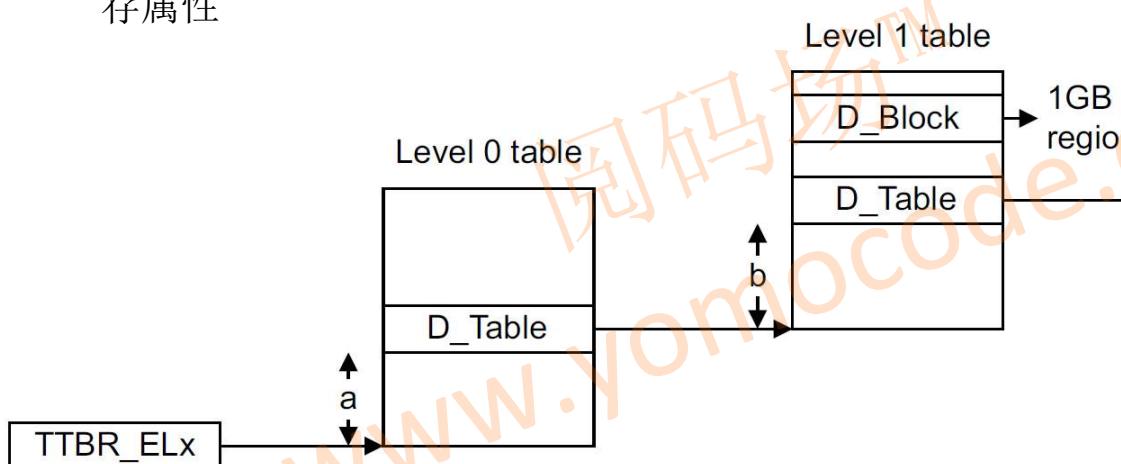
MMU STAGE2 TRANSLATION



PAGE TABLE STRUCTURE

页表由多级构成：

1. 可以避免线性页表在不连续的地址空间中造成的内存浪费
2. 由虚拟地址对每级页表进行索引
3. 页表中的**Table entry**描述了下一级页表的物理地址
4. 页表中的**BLOCK**和**PAGE Entry**，描述了对应的物理地址和内存属性



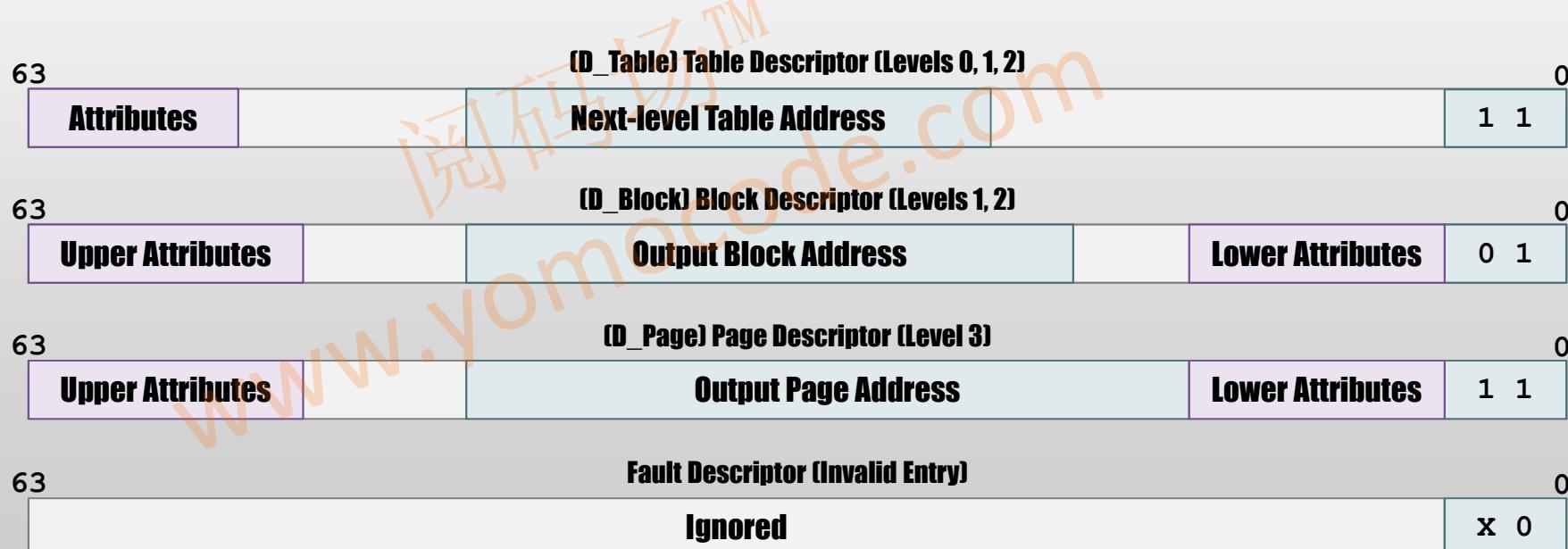
Key:

- D_Table is a Table descriptor
 - D_Block is a Block descriptor
 - D_Page is a Page descriptor
- a Indexed by IA[$n:39$], where IA width is $(n+1)$ bits
 - b Indexed by IA[38:30]
 - c Indexed by IA[29:21]
 - d Indexed by IA[20:12]

PAGE TABLE DESCRIPTOR

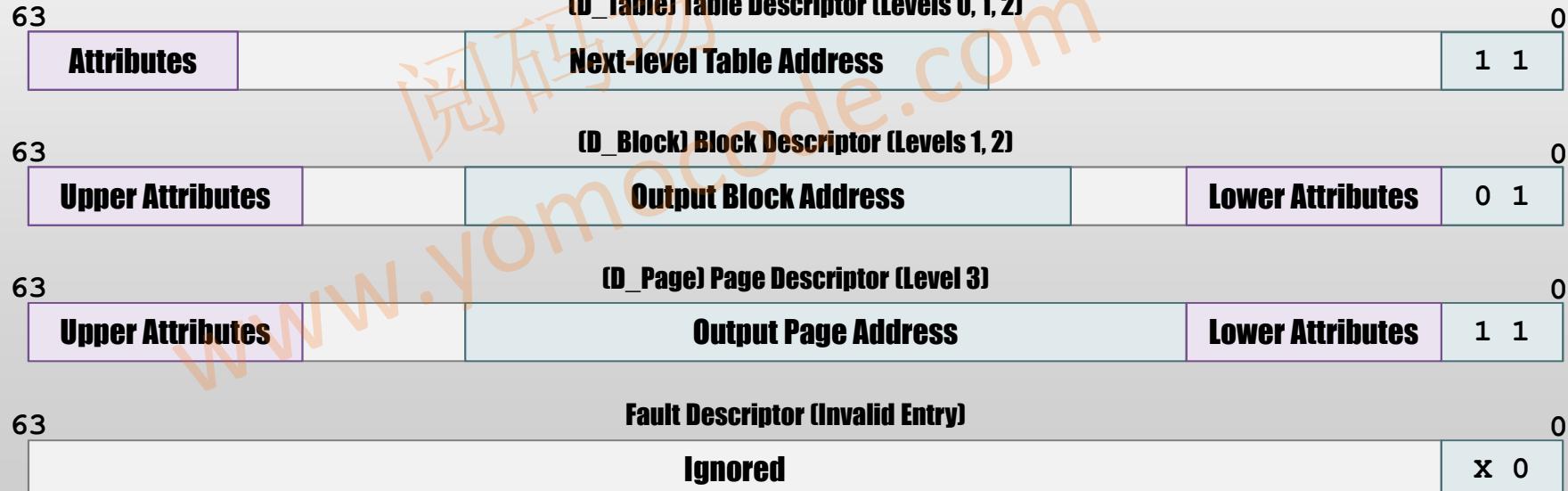
页表中内容我们称之为描述符：

1. **Table descriptor** 只有**level 0, 1, 2**级页表才有，它的输出地址是下一级页表的**IPA/PA**
2. **Block descriptor** 只有**level 1, 2**级页表才有，当虚拟内存以**2MB**或者**1GB**的大块进行映射时，它可以跳过后续的页表，直接返回。它的输出地址时映射物理块的起始地址。
3. **Table descriptor** 只有**level 3**才拥有，它的输出地址就是映射的最小单元**page**的物理地址



PAGE TABLE DESCRIPTOR

- Lower Attributes:** 结合MAIR寄存器决定所映射的物理空间是RAM还是IO，是否只读，是否是全局地址(忽略VMID/ASID)
- Upper Attributes:** 管理所映射物理空间的执行权限，完全可执行，特权可执行，非特权可执行等。
- Table的Attributes**只有在**Stage 1**中才可以决定下一级页表的执行权限和安全属性。



4K PAGE TABLE DESCRIPTOR

由虚拟地址对每级页表进行索引的示例，**48-bit**地址空间，**4KB PAGE**:

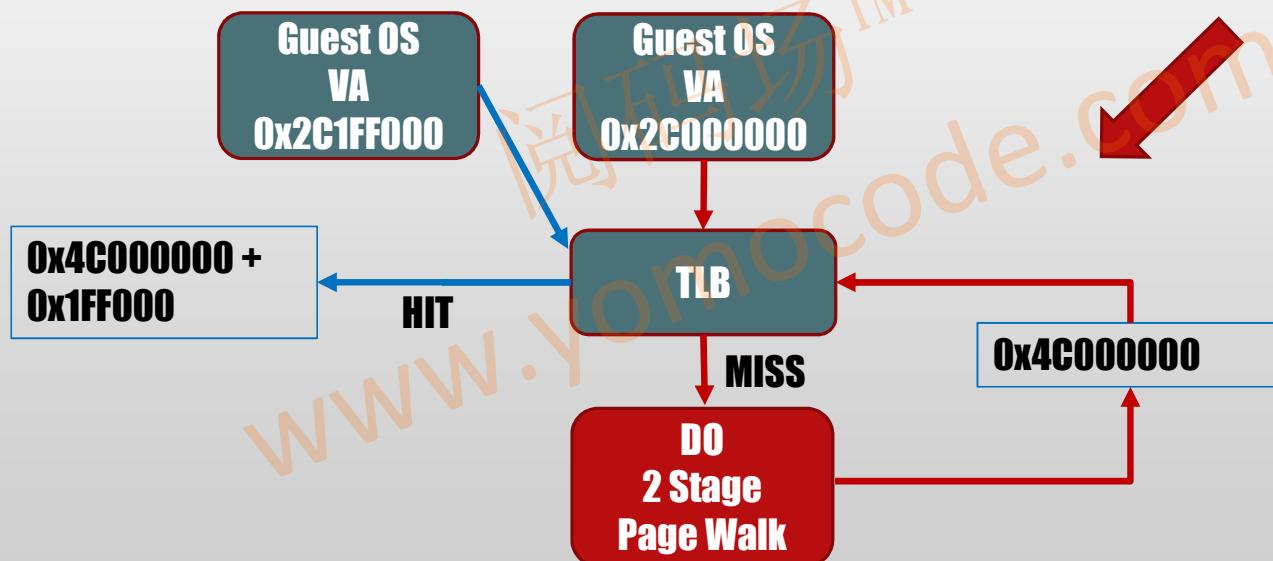
1. 从虚拟地址的最高位**bit-47**开始，每一级取**9bits**作为当前级别页面的索引
2. 因此每级页表，单个页面只支持**512**个**entry**

Virtual Address bits				
[47 : 39]	[38 : 30]	[29 : 21]	[20 : 12]	[11 : 0]
Level 0 Table 索引 每个 entry 只能: <ul style="list-style-type: none">指向一个 L1 Table	Level 1 Table 索引 每个 entry 可以: <ul style="list-style-type: none">指向一个 L2 Table指向一个 1GB Block	Level 2 Table 索引 每个 entry 可以: <ul style="list-style-type: none">指向一个 L3 Table指向一个 2MB Block	Level 3 Table 索引 每个 entry 可以: <ul style="list-style-type: none">指向一个 4KB page	Page Offset

PAGE WALK TUNING

Arm在页表上也提供减少**2 Stage Page Walk** 次数的设计：

1. 尽量使用大页内存，提高**TLB**的命中率减少**Page Wake**



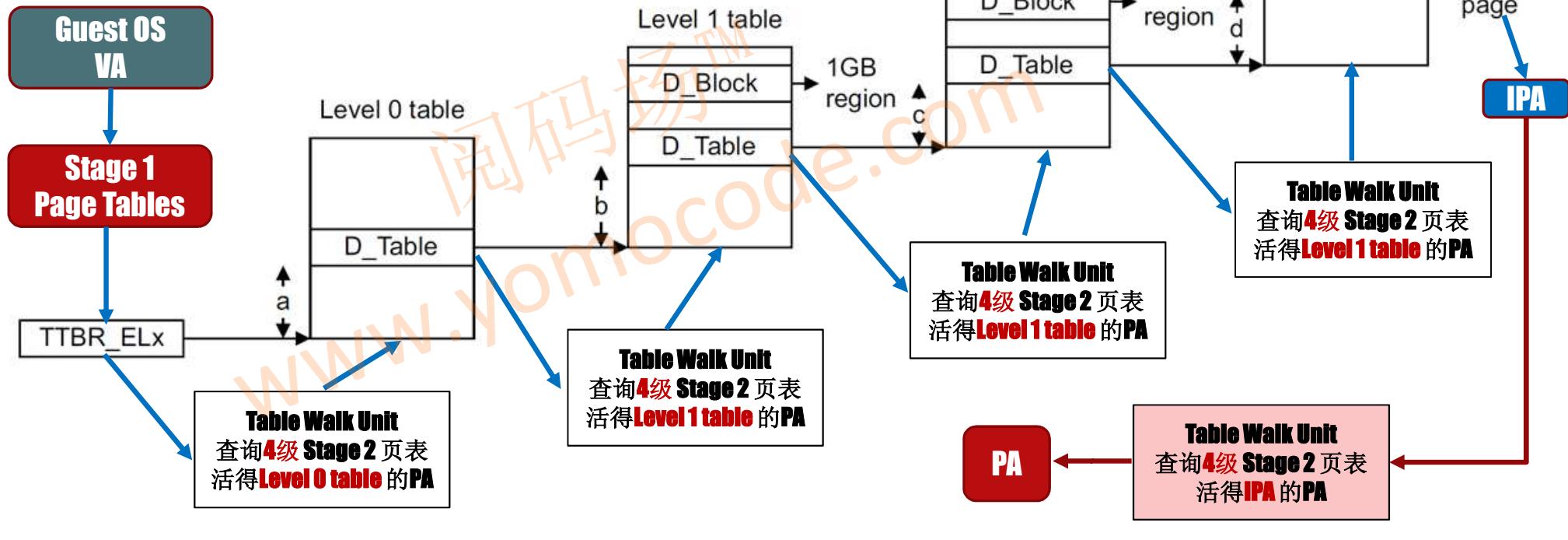
例如使用**2MB**大页：

1. 第一次访问时**0x2C000000**对应的**TLB**没有命中，**MMU**进行了**Page Walk**. 将得到的转换结果存入**TLB**
2. 第二次访问时**0x2C1FF000**在**0x2C000000 + 0x200000**的区间内，**mmu**判断**TLB**有命中，直接返回加上偏移的地址，结束转换

WORST CASE IN 2 STAGES TRANSLATION

Stage 1 和 Stage 2 页表拥有相同的结构：

1. 都支持最大4级页表
2. 在TLB都不命中的情况下，**GUEST VA => IPA**需要查询
 $4[\text{stage1}] + 4 * 4 [\text{stage2}] = 20$ 级页表，**24**级页表才能得到**PA**

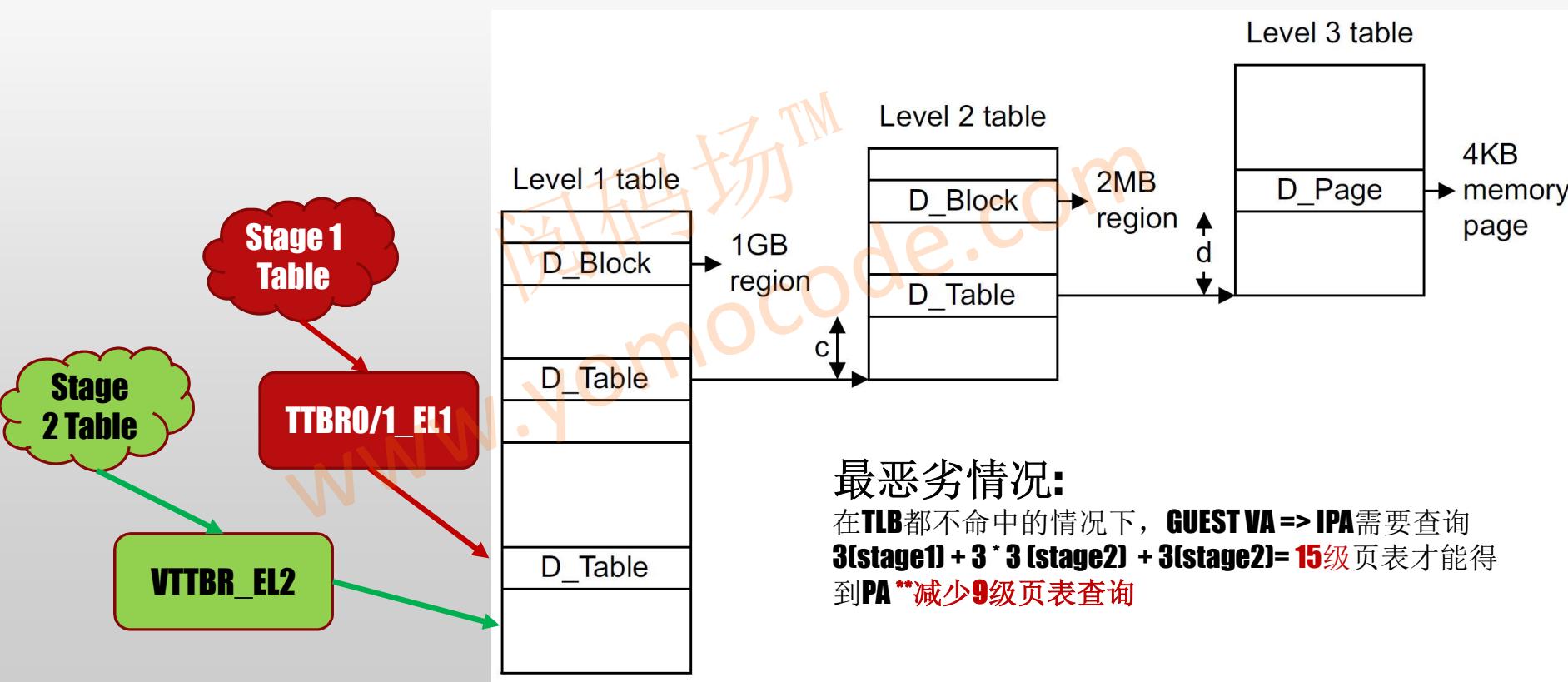


PAGE WALK TUNING

Arm在页表上也提供减少**2 Stage Page Walk** 次数的设计：

2. 设计虚拟机的地址空间时尽量减少空洞，缩减虚拟机的地址范围：

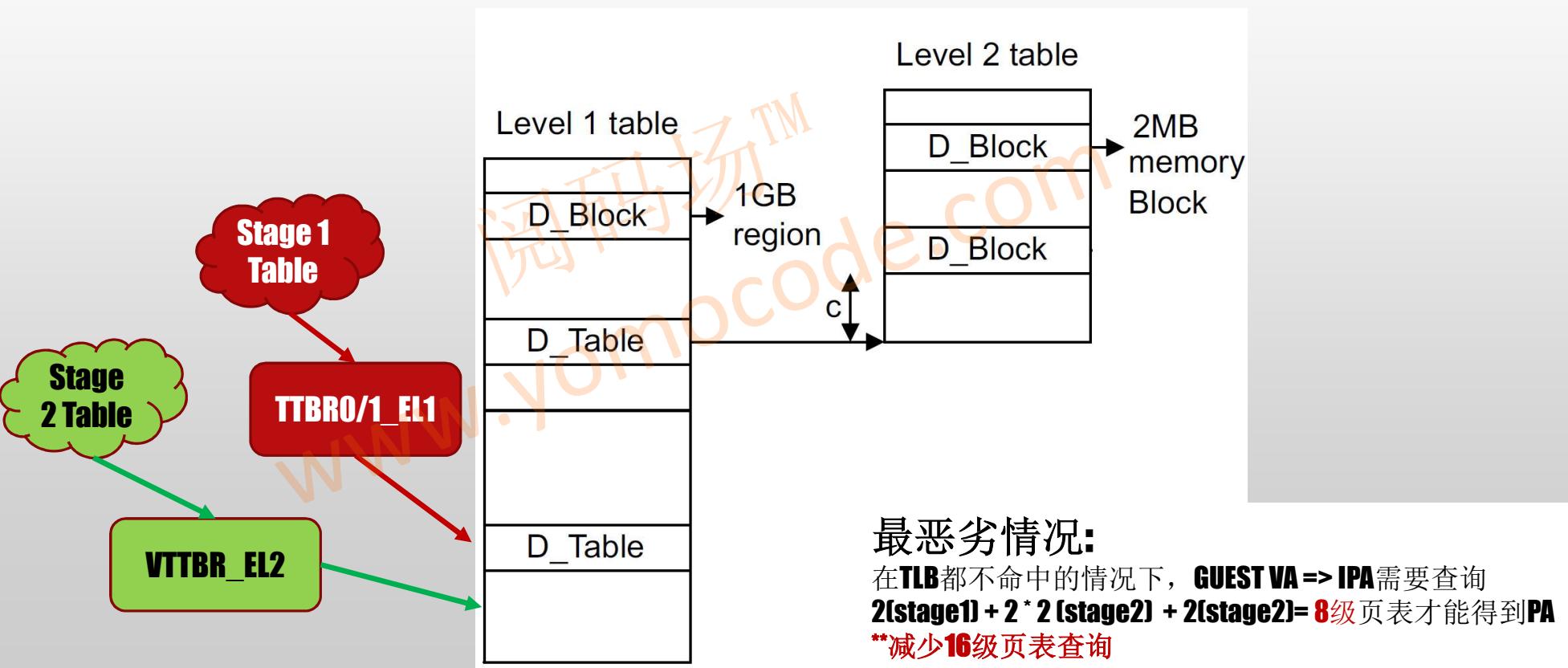
例如使用**39位**地址空间，我们就可以将**stage 1**和**stage 2**的**Level 0**页表省略



PAGE WALK TUNING

Arm在页表上也提供减少**2 Stage Page Walk** 次数的设计：

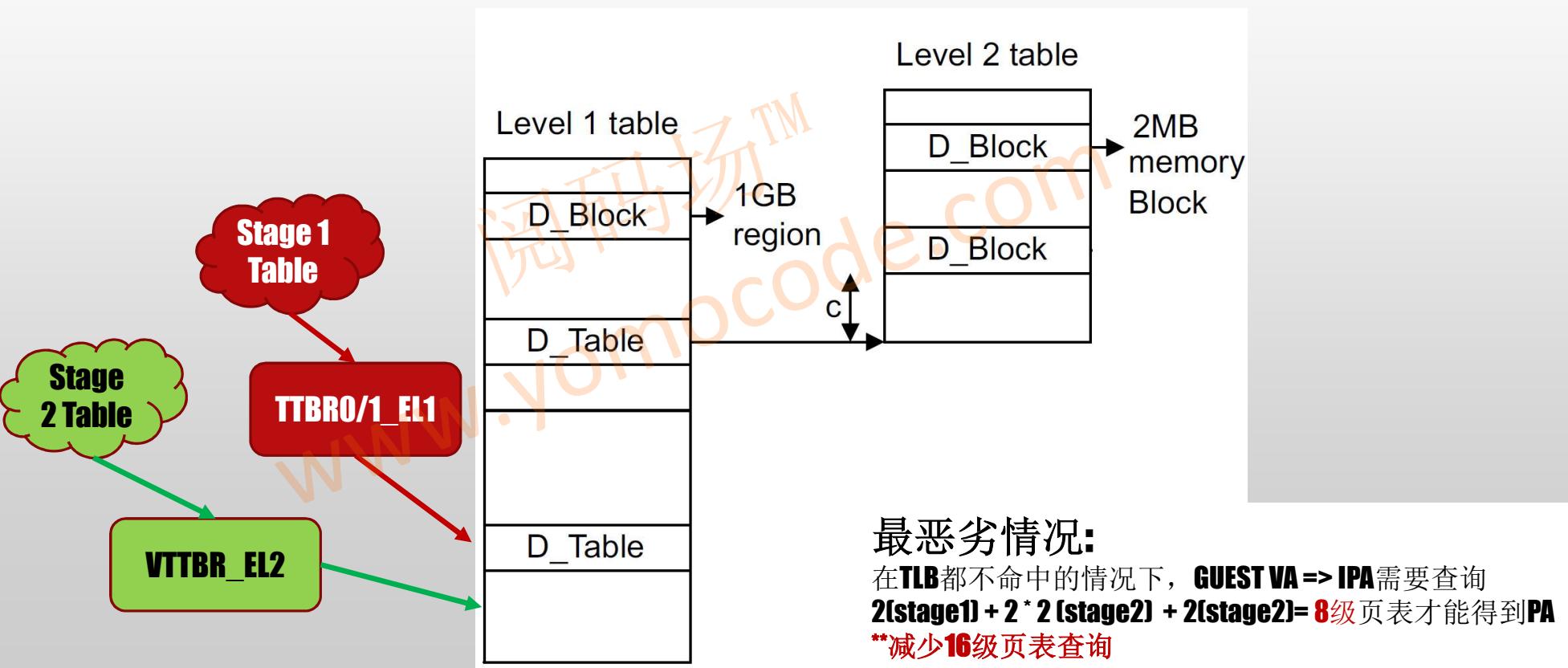
3. 在使用39位地址空间，将GuestOS以2MB大小作为page的对齐和管理单位，可以将**stage1** 和 **Stage2**页表降低为2级。



PAGE WALK TUNING

Arm在页表上也提供减少**2 Stage Page Walk** 次数的设计：

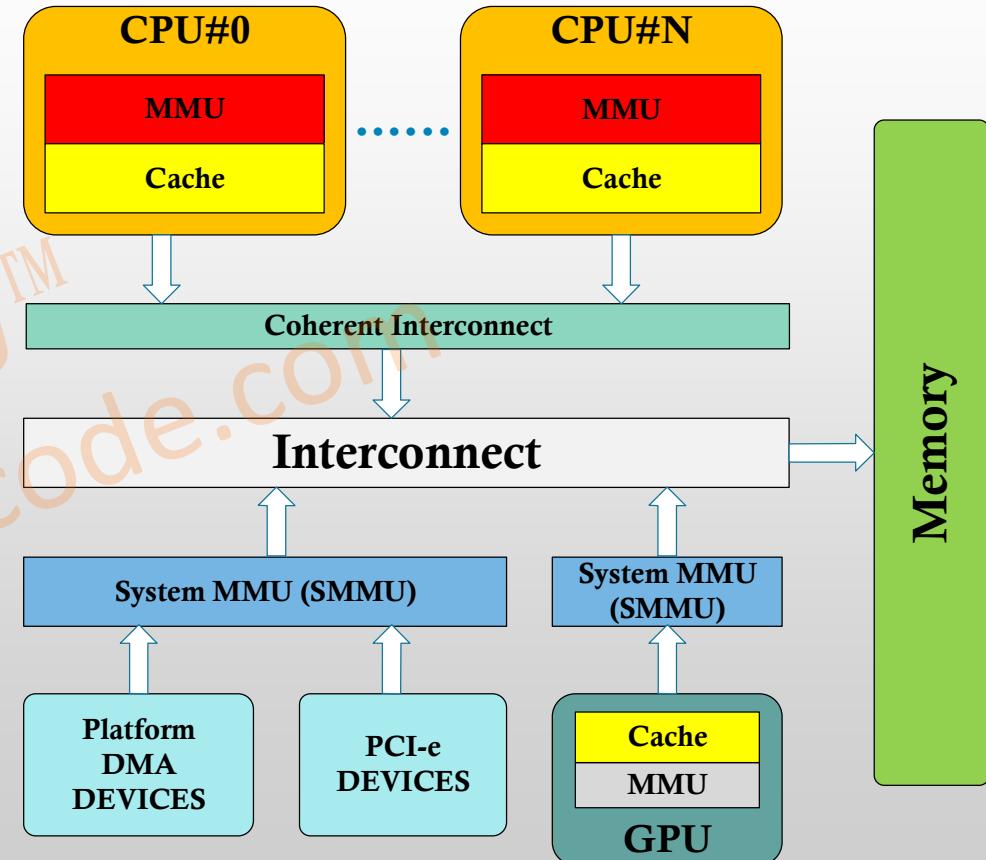
3. 在使用39位地址空间，将GuestOS以2MB大小作为page的对齐和管理单位，可以将**stage1** 和 **Stage2**页表降低为2级。



SMMU AND MMU

MMU 为**CPU**提供地址翻译，**SMMU**为设备提供地址翻译：

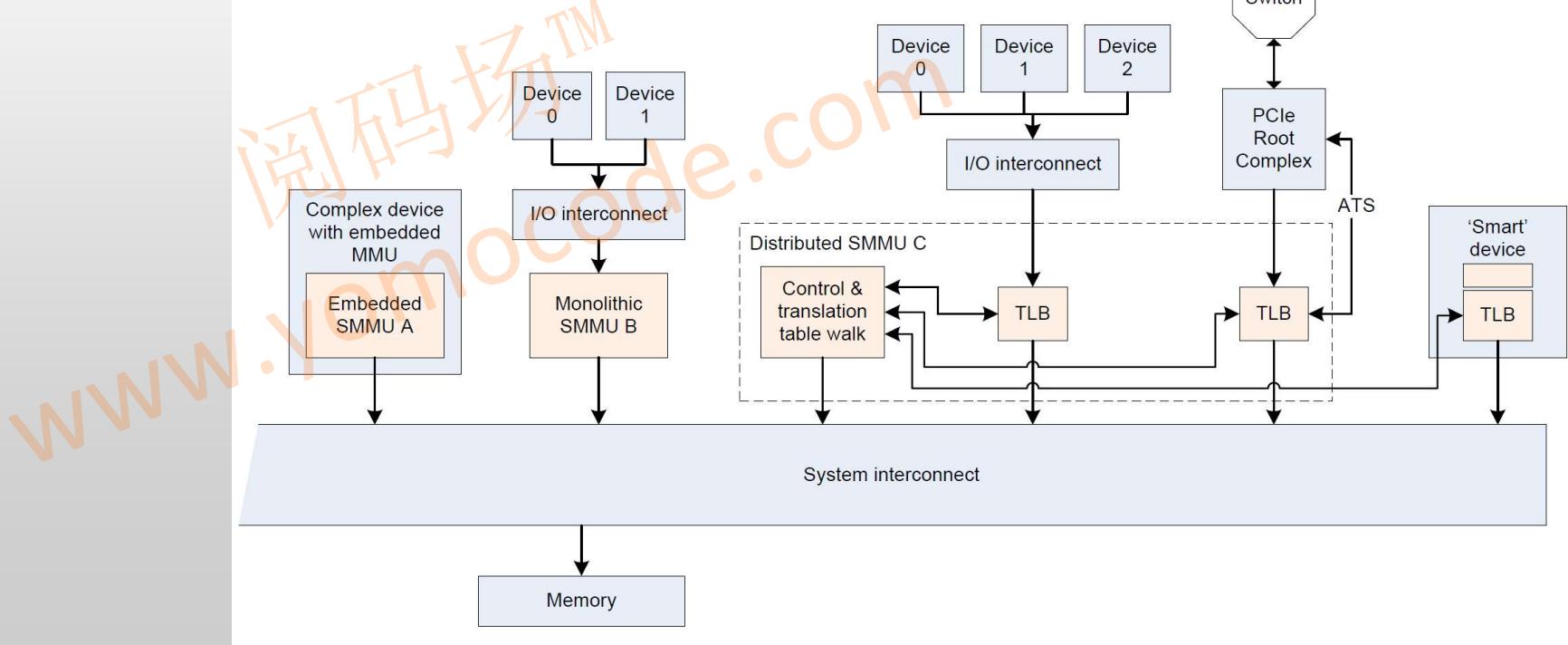
1. 当**DMA**设备通过**device passthrough**直通给虚拟机使用时，它和**CPU**一样都具备地址空间的访问能力
2. **DMA**设备和**CPU**都是内存的**master**，**SMMU**和**MMU**必须给**DMA**设备和**CPU**提供相同的地址空间视野
3. **SMMU**具有和**MMU**一致的**page table entry descriptor**



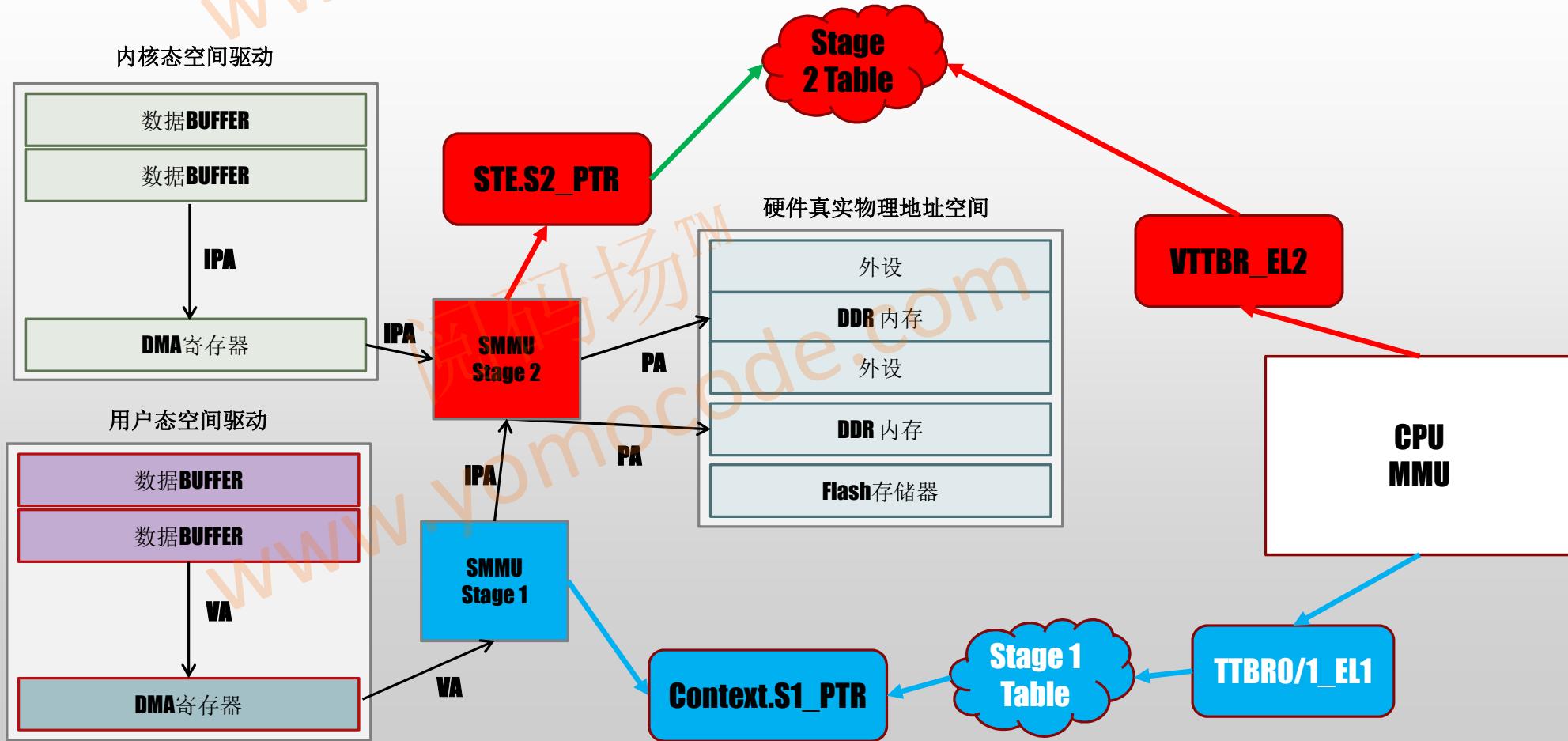
SMMU ON DMA PATH

SMMU有4种实现：

- SMMU作为一个复杂设备的一部分，集成到这个设备中
- SMMU实现是将地址翻译、编程接口作为一个单一模块，放置在两个或多个设备进行DMA访问的路径上。
- SMMU实现是分布式的，它提供多个DMA访问路径，以获得更高的带宽
- 仅集成TLB，TLB miss时向远程分布式SMMU请求翻译，降低了设备的复杂度。



SMMU AND MMU SHARE PAGE TABLES

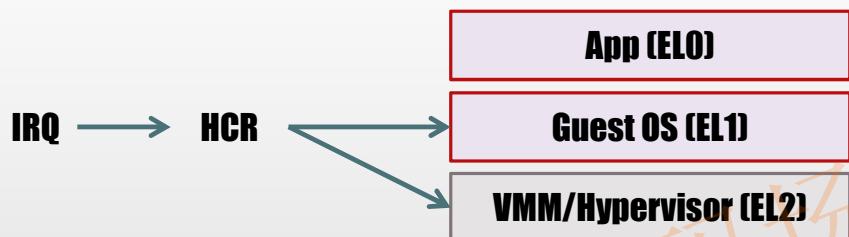


DETAILED IN INTERRUPT VIRTUALIZATION

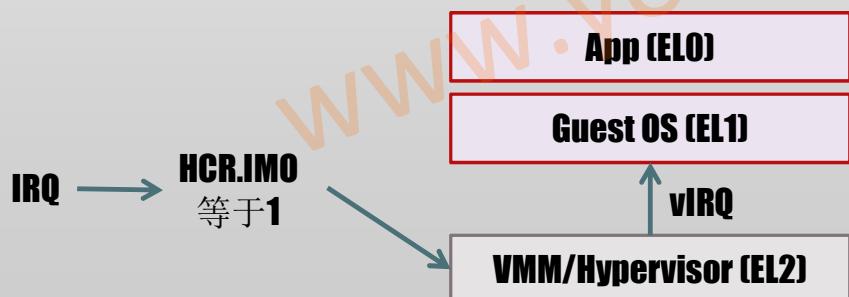
深入理解中断虚拟化

VIRTUAL EXCEPTIONS

在**Armv8**中，中断可以通过**HCR**寄存器的配置，选择让哪一个**Exception Level (EL)**来处理中断。



在典型的**Arm**虚拟化系统里，一般都是让**VMM**所在的**EL2**来处理来自真实设备的中断，然后通过虚拟化异常(**virtual exception**)向**GuestOS**注入中断。



1. Armv8 支持3种 virtual exceptions:

- 虚拟系统错误 (**virtual Serror**)
- 虚拟中断 (**virtual IRQ**)
- 虚拟快速中断 (**virtual FIQ**)

2. **Virtual IRQ** 只能由**EL2 hypervisor/VMM** 发起，然后由**EL1 GuestOS**接受。

3. **Hypervisor**本身并不会收到**virtual IRQ**

4. Arm 在硬件上提供了两种机制来发起**virtual IRQ**:

- 通过**GIC**中断控制器 (**Generic Interrupt Controller**)
- 通过**CPU EL2**的**hypervisor** 配置寄存器 **HCR_EL2.VI**

GIC VIRTUALIZATION EXTENSION

GIC 通用模块:

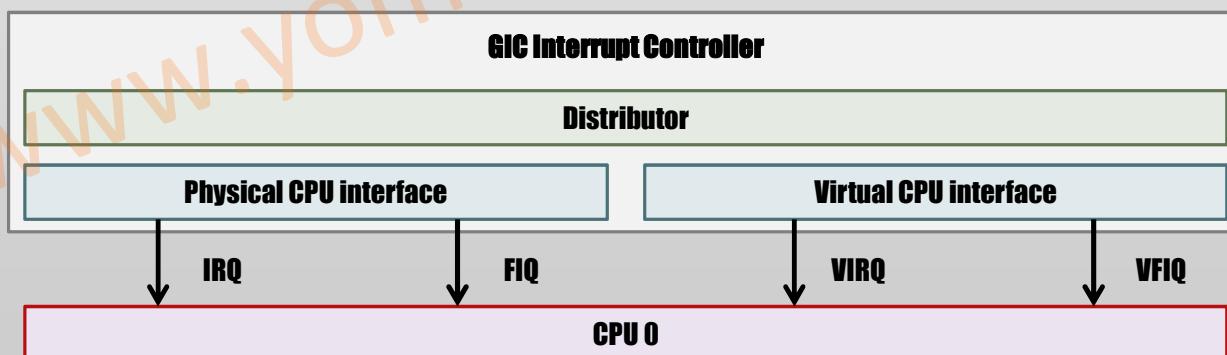
- **Distributor** – 和设备连接，用来配置中断的开关、属性等
- **Physical CPU interface** – 和**CPU**接口，触发**CPU**进入中断异常

GIC 虚拟化扩展模块:

- **Virtual CPU interface** – 和**CPU**的接口，触发虚拟中断
- **Hypervisor interface** – **VMM**用来给虚拟机配置**Virtual CPU interface**

在虚拟化系统中，**physical CPU interface** 和 **hypervisor interface** 属于 **VMM** 控制

- **VMM** 通过 **physical CPU interface** 接收物理中断
- **VMM** 通过 **hypervisor interface** 配置 **virtual CPU interface** 向虚拟机注入虚拟中断
- **GuestOS** 通过 **virtual CPU interface** 接收虚拟中断并处理



GIC VERSIONS

GICv1

- 最高支持**8CORE**
- 支持高达**1020**个中断**ID**
- **8-BIT** 中断优先级
- 提供软件接口触发**SGI**
- 首次支持**TRUSTZONE**
- **Implemented by:**
 - Cortex-A9 MPCore
 - Cortex-A5 MPCore
 - Cortex-R7 MPCore
 - CoreLink™ GIC-390

GICv2

在**V1**基础上增加了：

- 虚拟化扩展支持
- 改进了**SECURE SOFTWARE** 对**GROUP1**中断的处理
- ****SBSA** 通过**GICV2M** 通过扩展增加了对**MSI**中断的支持用于服务器。

▪ **Implemented by:**

- Cortex-A15 MPCore*
- Cortex-A7 MPCore*
- CoreLink GIC-400

GICv3

在**V2**基础上增加了：

- 支持**8**个以上的**CORE**
- 原生支持**MSI**中断
- 支持通过系统寄存器方式访问部分**GIC**寄存器
- 最高支持**2^24**个中断号

▪ **Implemented by:**

- Cortex-R52
- CoreLink GIC-500
- CoreLink GIC-600

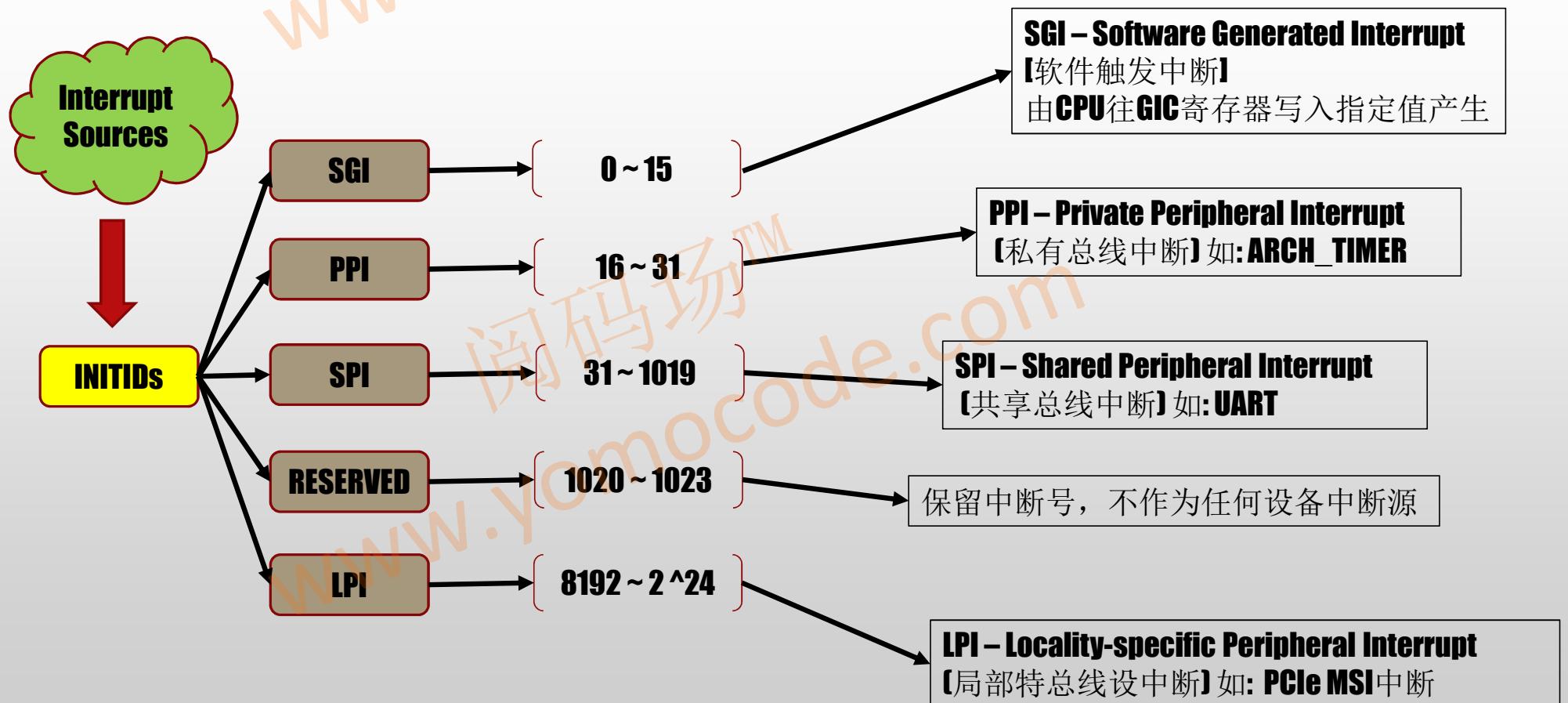
GICv4

在**V3**基础上增加了：

- **DIRECT INJECTION OF VIRTUAL INTERRUPTS**

[中断直接注入当前运行的虚拟机，无需**HYPERVISOR**介入，即使配置了**HCR_EL2.IM0=1**]

INTERRUPT TYPES



INTERRUPT STATES

- **Inactive**

- 该中断既没有新的信号产生，也没有在处理中的信号。

- **Pending**

- 中断信号已经产生，但是还没有中断服务程序来响应。

- **Active**

- 已经有中断服务程序响应该信号，但是还没有处理完成。

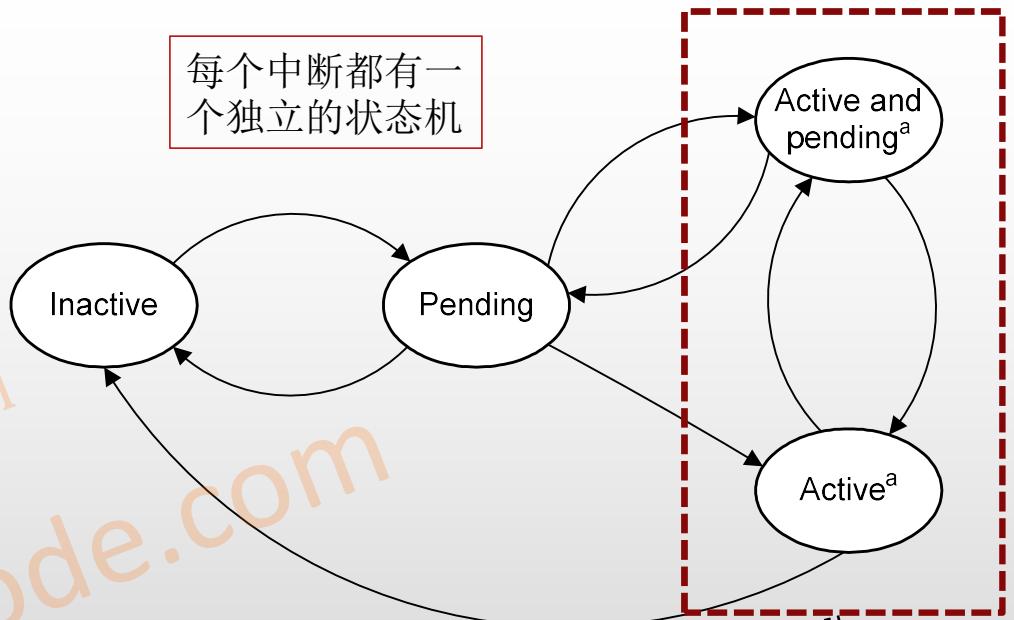
- **Active & Pending**

- 当前信号在处理时又有一个新的信号到达
 - 只有边缘触发型[**edge-triggered**]的中断才有这个状态

- **状态变迁:**

- 中断信号被触发时， **Inactive → Pending**
 - 中断服务程序响应该信号时【中断服务程序读取**GIC_IAR**寄存器】 **Pending → Active**
 - 中断服务程序完成处理【中断服务程序写入**GIC_EOIR**寄存器】 **Active → Inactive**

每个中断都有一个独立的状态机

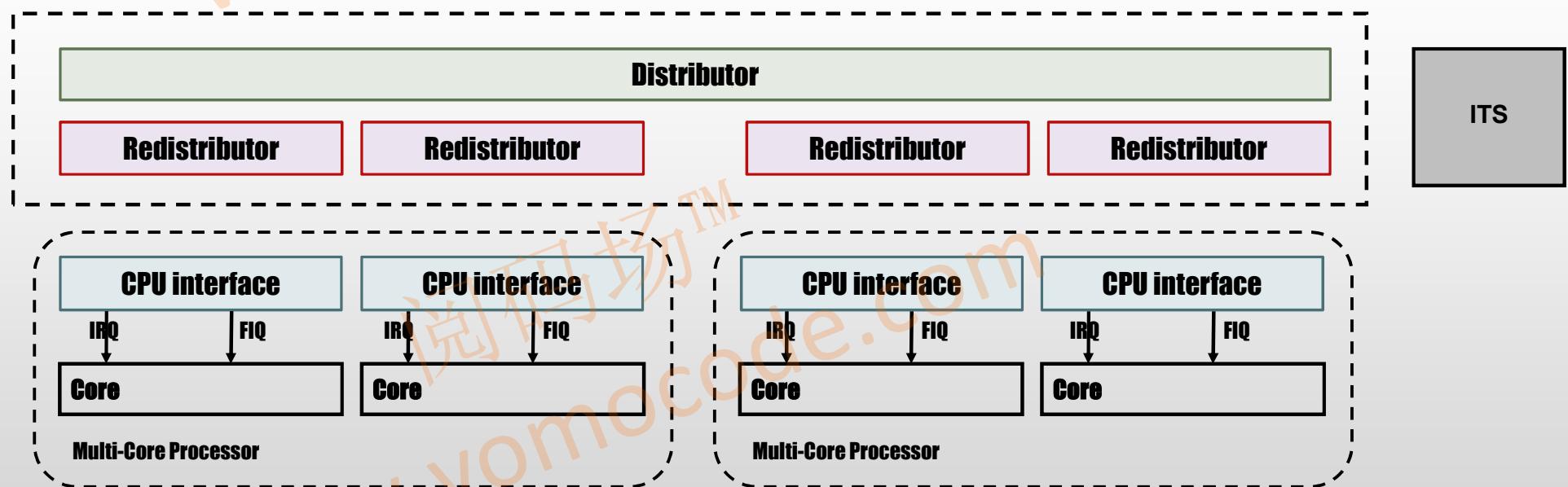


LPI没有这两个状态

INTERRUPT SECURITY

- GICV3 SUPPORTS THREE GROUP/SECURITY SETTINGS
 - SET INDIVIDUALLY FOR EACH INTERRUPT
- GROUP 0
 - GROUP 0 INTERRUPTS ARE ALWAYS SECURE
 - SIGNALLED AS FIQ, REGARDLESS OF CURRENT SECURITY STATE
 - TYPICALLY USED FOR INTERRUPTS FOR THE FIRMWARE RUNNING AT EL3
- SECURE GROUP 1
 - SIGNALLED AS FIQ IF CORE IS IN NON-SECURE STATE
 - SIGNALLED AS IRQ IF CORE IS IN SECURE STATE
 - TYPICALLY USED FOR INTERRUPTS FOR THE TRUSTED OS
- NON-SECURE GROUP 1
 - SIGNALLED AS FIQ IF CORE IS IN SECURE STATE
 - SIGNALLED AS IRQ IF CORE IS IN NON-SECURE STATE
 - TYPICALLY USED FOR INTERRUPTS FOR THE RICH OS OR HYPERVISOR

INTERFACES



- **CPU INTERFACE** 集成在**CPU**内部，可以通过系统寄存器方式访问
 - **ICC_**, **ICH_** 和 **ICV_** 前缀的系统寄存器 **[ICH_]** 是给**HYPERVERISOR**访问的专有寄存器，当**OS**运行在虚拟机内部时**ICC_**会自动映射成**[ICV_]**
- **DISTRIBUTOR, REDISTRIBUTOR** 和 **ITS INTERFACES ARE** 实现在**CPU**外部
 - 只能同过**GICD_**, **GICR_** 和 **GITS_** 前缀的MMIO寄存器访问

SPI, SGI AND PPI CONFIGURATION

• ENABLE

- 使能中断，中断使能后，硬件信号才能在让**CPU**进入中断异常
- 被关闭的中断仍然能够进入**PENDING**状态，只是**CPU**不会主动进入中断异常，当该中断再次被使能时，**CPU**会进入异常。

• PRIORITY

- 最高**8-BIT**优先级设置，**0**最高~**255**最低

• CONFIGURATION

- 配置中断的触发类型水平触发还是边缘触发
- SGI**都是边缘触发

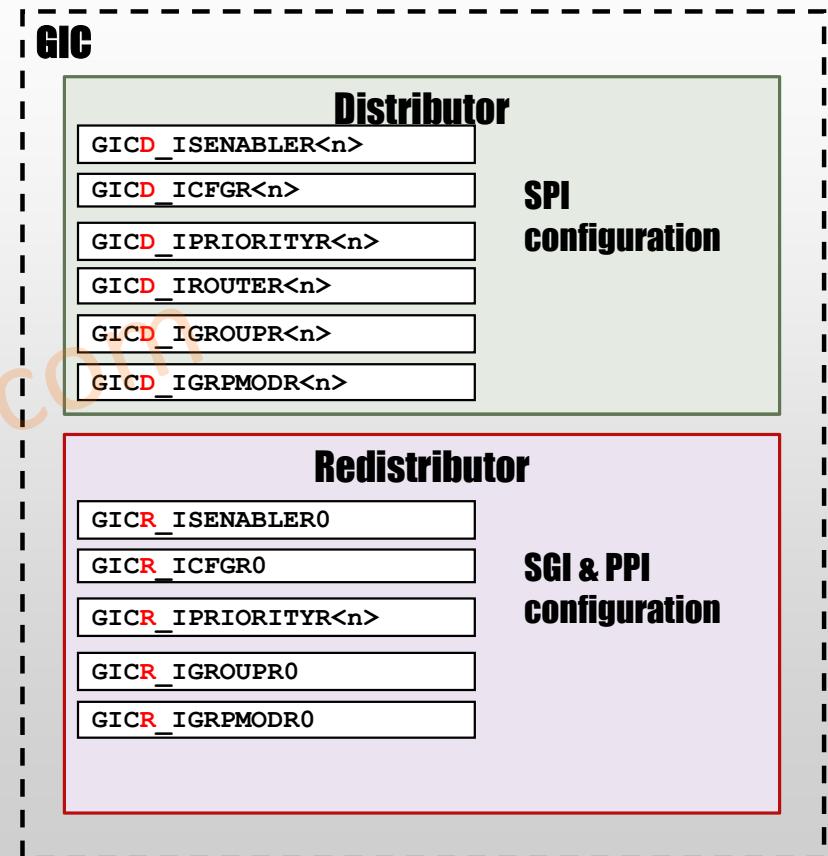
• ROUTING

- 配置发送的目标**CPU**，可以是一个或者多个**CPU**.

• SECURITY/GROUP

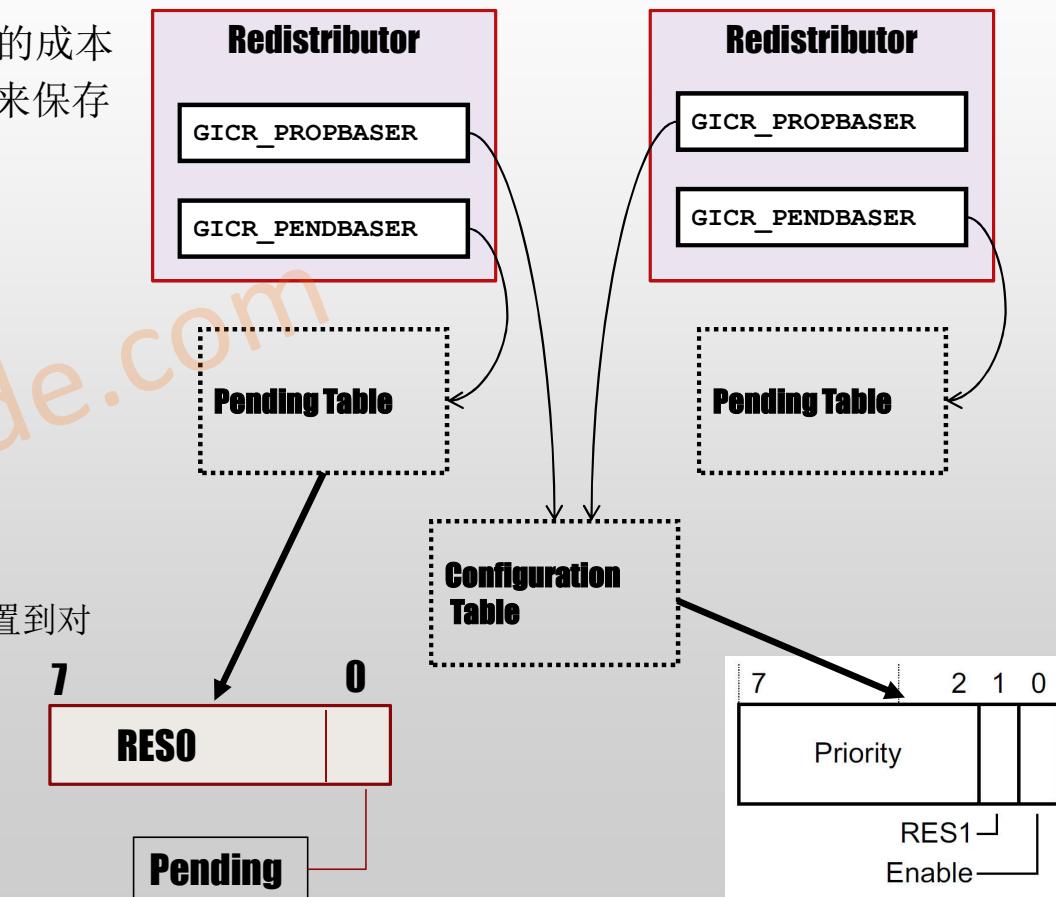
- 可以配置中断所属的分组(**IRQ/FIQ/SECURE/NON-SECURE**)

***不能在中断**active**状态重新配置中断

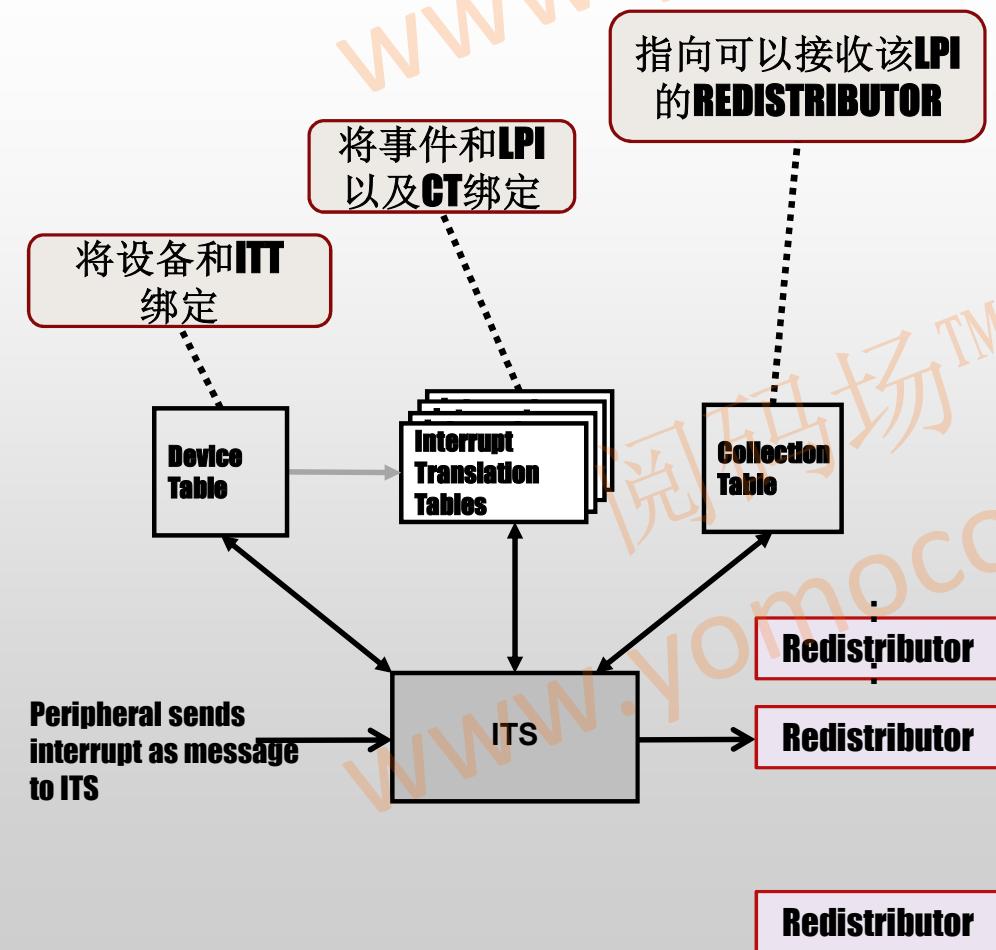


LPI CONFIGURATION

- LPI的INITID因为范围广，它的配置和状态保存在寄存器中的成本太高。所以通过内存中的**LPI CONFIG TABLE**和**PENDING TABLE**来保存
- GICR_PROPBASER指向保存**LPI**配置表的内存块
 - **LPI CONFIGURATION**是全局的
 - 所有的**REDISTRIBUTORS**都共享一张**CONFIG TABLE**
- GICR_PENDBASER指向**LPI**状态表的内存块
 - 每个**REDISTRIBUTOR**都有一张独立的**LPI**状态表
- 系统初始化时：
 - 需要在内存中分配所有的**LPI**配置表和状态表，并将它们设置到对应的**GICR_PROPBASER**和**GICR_PENDBASER**寄存器。
 - 帮助初始化**LPI**配置表和状态表
 - 在**REDISTRIBUTOR**上使能**LPI**(**GICR_CTLR.ENABLELPI**)



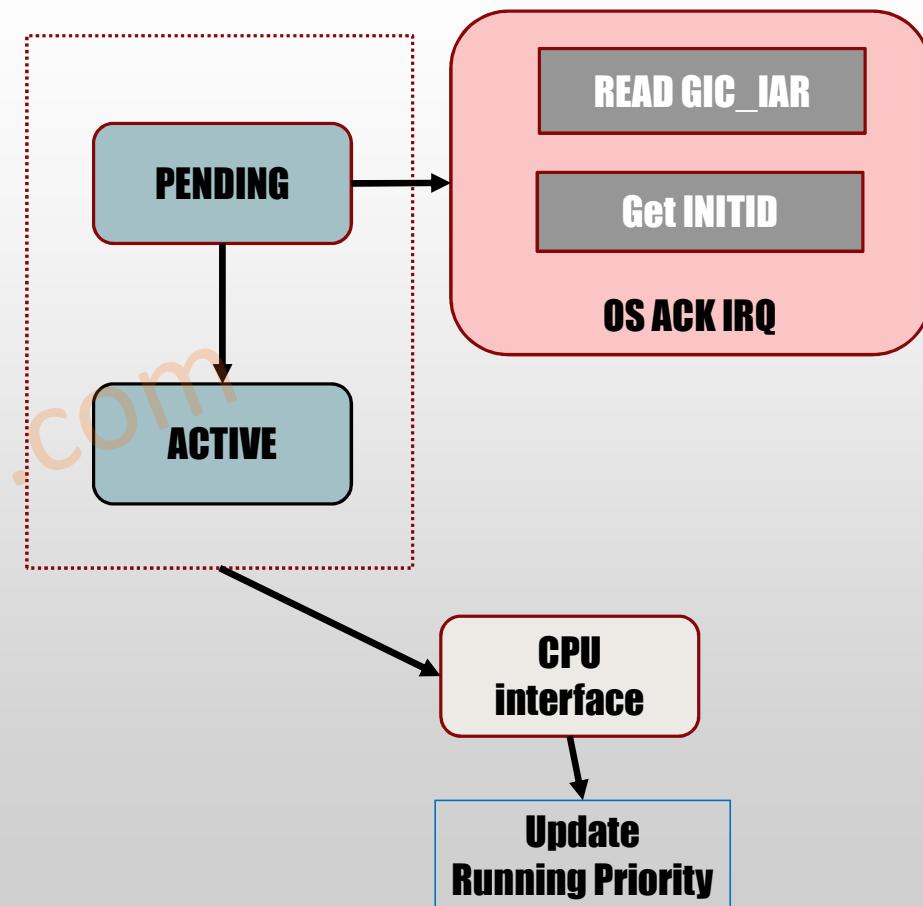
ITS



- **LPI**s are message-based, and peripherals usually convert their own interrupt messages into **LPI** interrupts through the **ITS (INTERRUPT TRANSLATION SERVICE)**.
- **ITS** conversion process:
 - Peripherals send interrupt messages to **ITS**. The message consists of the device ID and event ID.
 - **ITS** finds the corresponding **ENTRY** in the **DEVICE TABLE** based on the device ID.
 - **ITS** returns the **ENTRY** address to the peripheral.
 - **ITS** finds the corresponding **ENTRY** in the **INTERRUPT TRANSLATION TABLE** based on the event ID.
 - **ITS** returns the **LPI INTID** and **COLLECTION ID** to the peripheral.
 - **ITS** finds the corresponding **ENTRY** in the **COLLECTION TABLE** based on the **COLLECTION ID**.
 - **ITS** returns the **TARGET REDISTRIBUTOR** to the peripheral.
 - **ITS** forwards the interrupt to the **REDISTRIBUTOR**.
 - **REDISTRIBUTOR** triggers the **CPU** to enter an interrupt state.

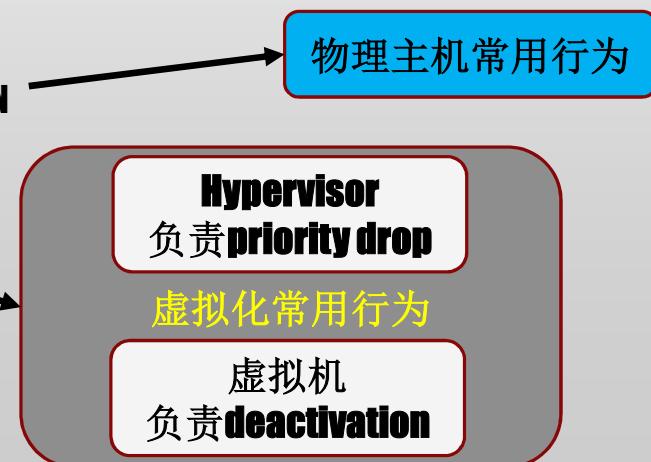
ACKNOWLEDGING INTERRUPTS

- 操作系统必须读取**GIC_IAR** (**INTERRUPT ACKNOWLEDGE REGISTERS**) 寄存器来响应中断
 - 这个过程会改变中断的状态，从**PENDING** 变成**ACTIVE**
 - 同时返回该中断的**INITID**
 - CPU INTERFACE**会将该中断的优先级作为**运行中的优先级**任何低于该优先级的中断都无法抢占该中断的处理过程。可以通过**ICC_RPR_EL1**查看该优先级



AT THE END OF YOUR INTERRUPT HANDLER

- 当一个中断被处理完成后，操作系统必须执行结束处理告诉**GIC**中断以及处理完成，这是为了
 - PRIORITY DROP** 运行优先级降权
 - 我们在处理中断时，将运行优先级设为该中断的优先级，并禁止低于这个优先级的中断抢占
 - 处理完中断后需要及时恢复，让低优先级中断可以正常被服务
 - DEACTIVATION**
 - 处于**ACTIVE**状态的中断无法再次进入**PENDING**[边缘触发除外]，需要通过**DEACTIVATION**将**ACTIVE**变成**INACTIVE**.
- END OF INTERRUPT**有两种方式，由**GIC DISTRIBUTOR**的**CTLR**存器**EOIMODE**位决定
 - EOIMODE == 0**
 - 写入**ICC_EOIRN_EL1**完成**PRIORITY DROP**和**DEACTIVATION**
 - EOIMODE == 1**
 - 写入**ICC_EOIRN_EL1**只会完成**PRIORITY DROP**
 - 写入**ICC_DIR_EL1**才会完成**DEACTIVATION**



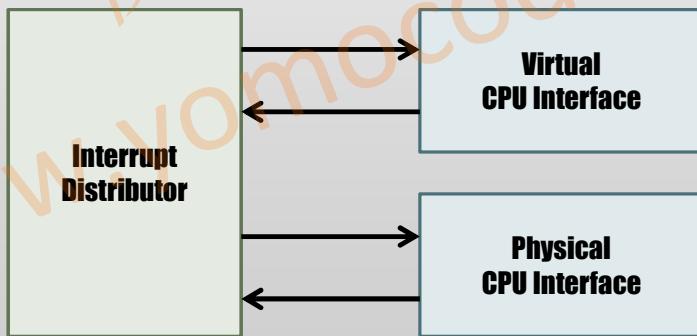
RESERVED INTIDS

- 当我们响应中断读取**GIC_IAR**时，有时会返回一些**INITID**，内容就是我们之前没细说的保留**INITID**
- 1020, 1021**只有在**EL3**可以看到。一般发生在因为错误的中断安全组配置。
- 1022**-这是一个历史异常，我们可以
- 1023**
 - 表示当前**CPU**上没有**PENDING**的中断需要处理
 - 或者表示我们在**NON-SECURE**尝试处理一个**SECURE**中断

VIRTUAL INTERRUPT INJECT WITH GIC

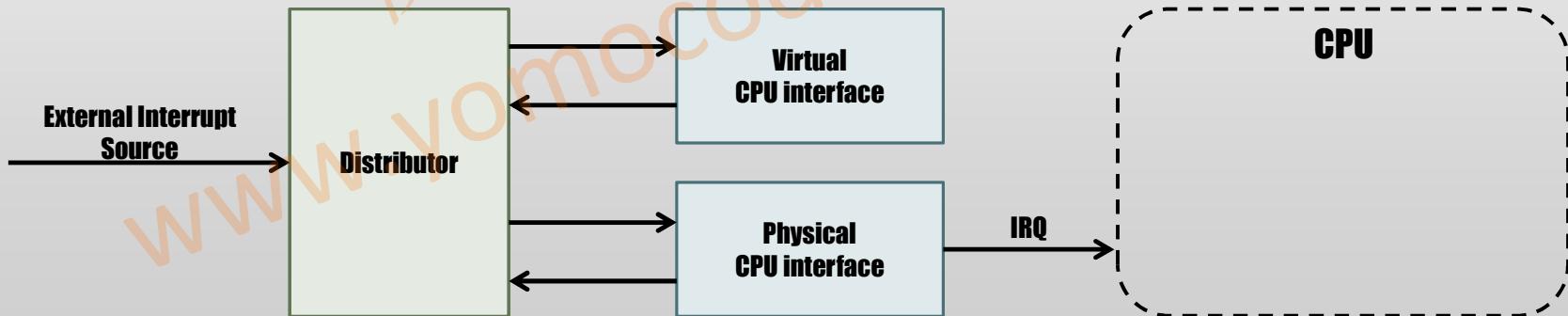
通过**GIC**的虚拟中断注入

VIRTUAL INTERRUPT SIGNALING (GIC)



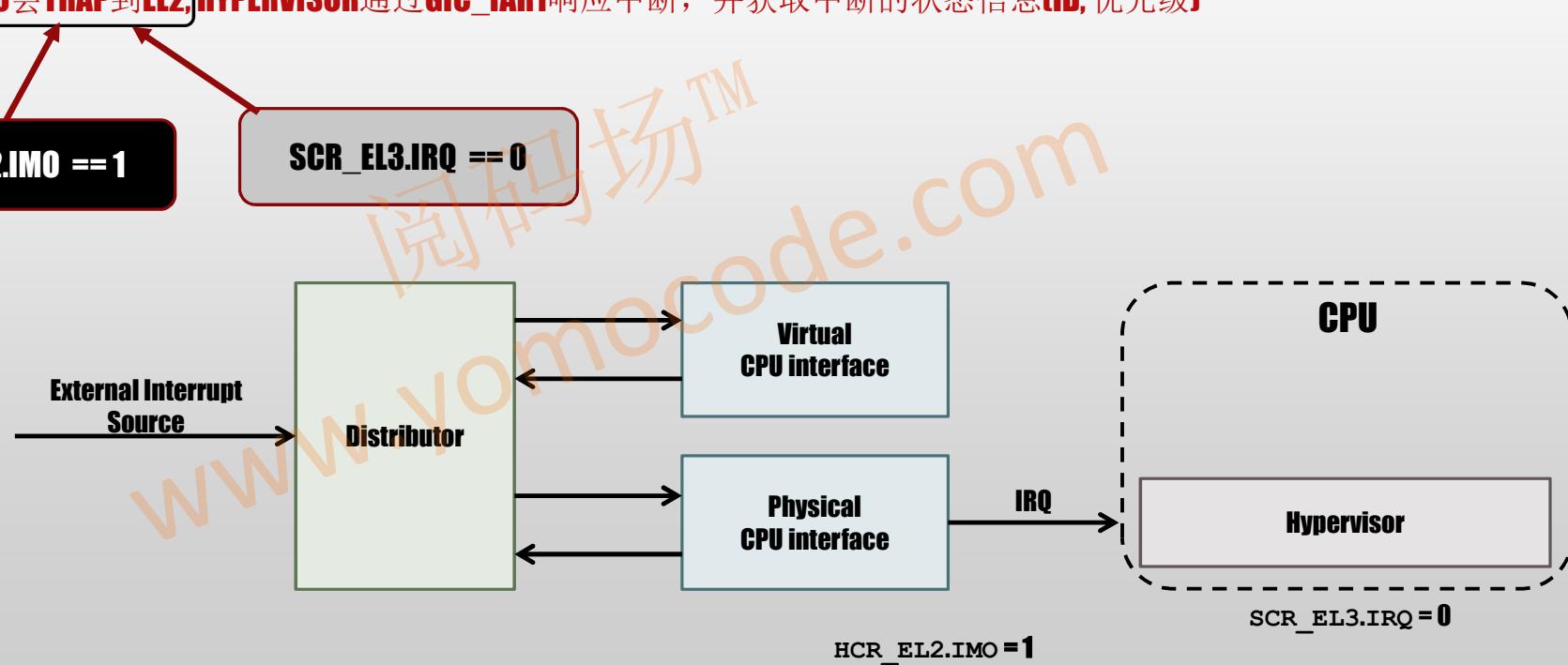
VIRTUAL INTERRUPT SIGNALLING (GIC)

1. 外设中断到达**GIC**
2. 并通过**PHYSICAL CPU INTERFACE** 发送一个物理中断到**CPU**



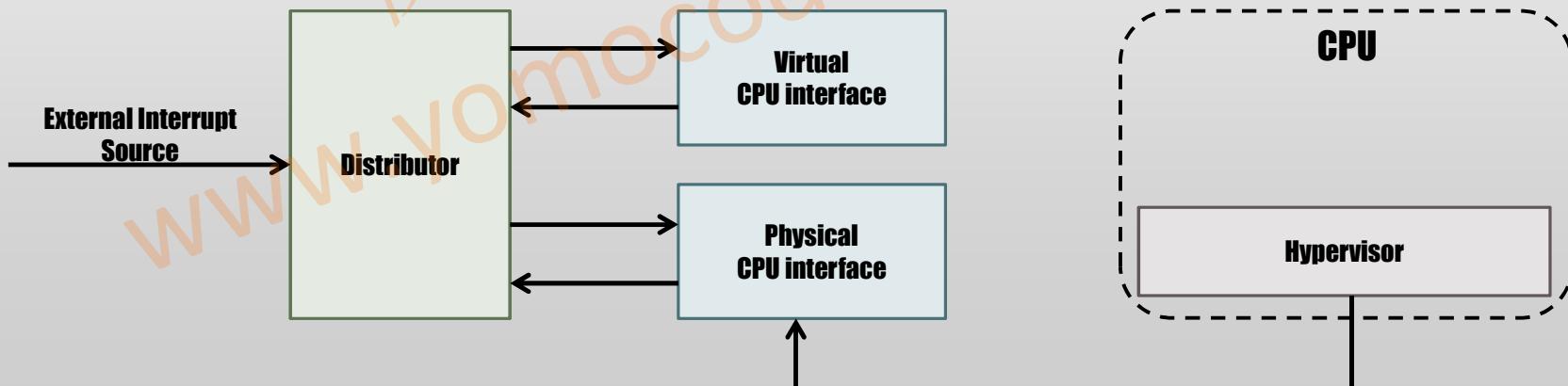
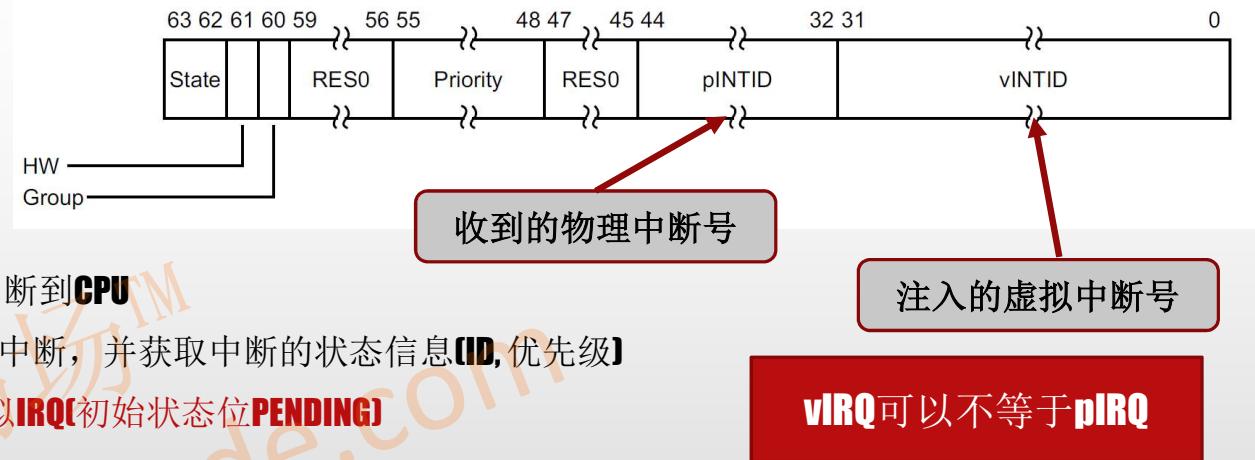
VIRTUAL INTERRUPT SIGNALLING (GIC)

1. 外设中断到达**GIC**
2. 并通过**PHYSICAL CPU INTERFACE**发送一个物理中断到**CPU**
3. **CPU会TRAP到EL2, HYPERVISOR通过GIC_IAR1响应中断，并获取中断的状态信息(ID,优先级)**



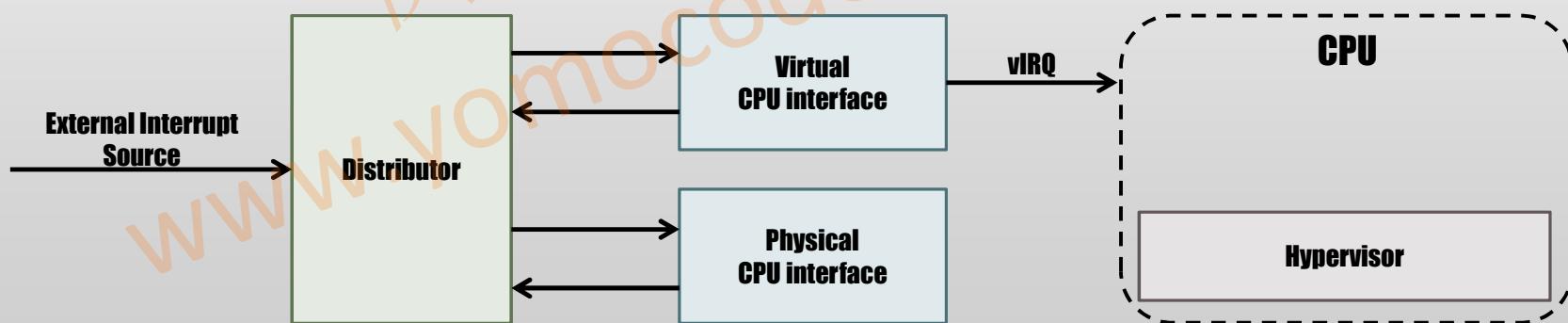
VIRTUAL INTERRUPT SIGNALLING (GIC)

1. 外设中断到达**GIC**
2. 并通过**PHYSICAL CPU INTERFACE**发送一个物理中断到**CPU**
3. **CPU**会**TRAP**到**EL2, HYPERVISOR**通过**GIC_IAR1**响应中断，并获取中断的状态信息**[ID, 优先级]**
4. **HYPervisor**往**GIC LIST REGISTER TO**注册一个虚拟**IRQ**[初始状态位**PENDING**]



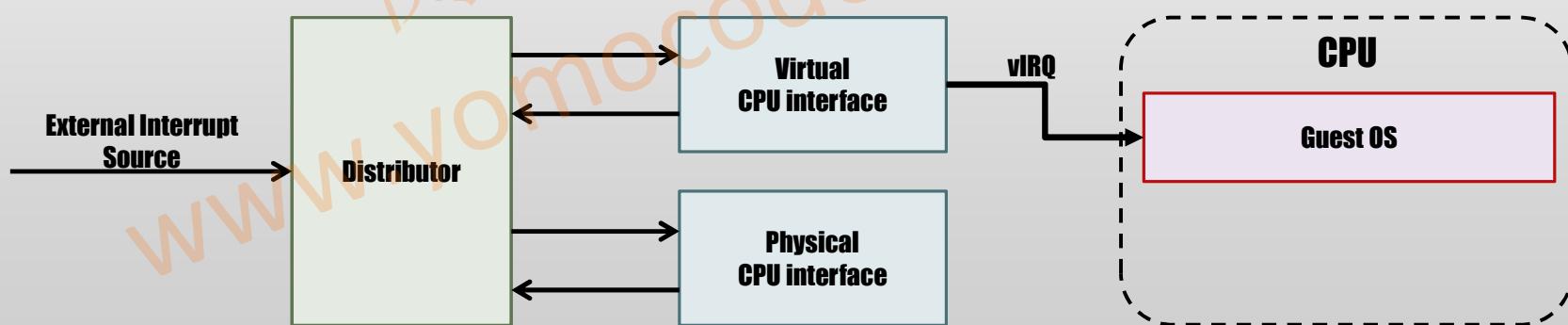
VIRTUAL INTERRUPT SIGNALLING (GIC)

1. 外设中断到达**GIC**
2. 并通过**PHYSICAL CPU INTERFACE**发送一个物理中断到**CPU**
3. **CPU**会**TRAP**到**EL2, HYPERVISOR**通过**GIC_IAR1**响应中断，并获取中断的状态信息**(ID, 优先级)**
4. **HYPERVISOR**往**GIC LIST REGISTER TO**注册一个虚拟**IRQ****[初始状态位 PENDING]**
5. **GIC VIRTUAL CPU INTERFACE**会触发一个虚拟**IRQ**信号到**CPU**



VIRTUAL INTERRUPT SIGNALLING (GIC)

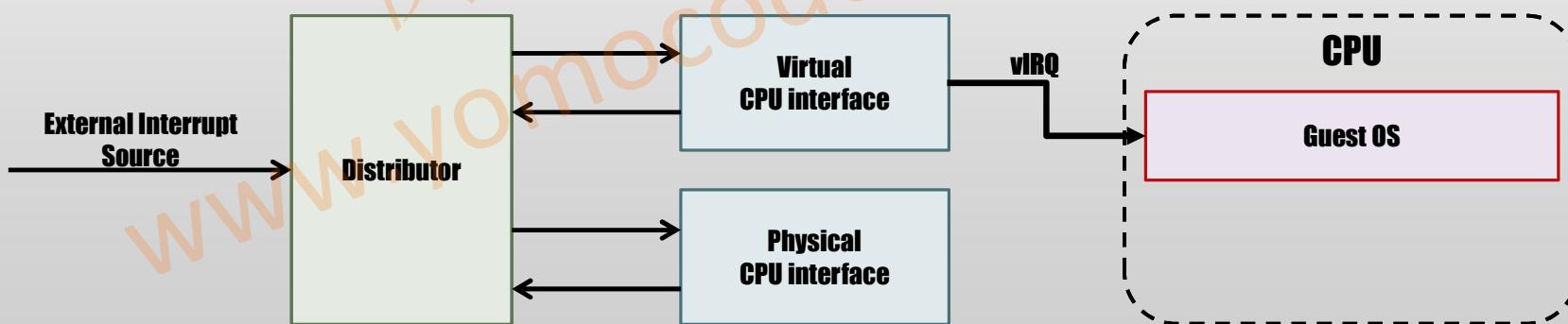
1. 外设中断到达**GIC**
2. 并通过**PHYSICAL CPU INTERFACE**发送一个物理中断到**CPU**
3. **CPU**会**TRAP**到**EL2, HYPERVISOR**通过**GIC_IAR1**响应中断，并获取中断的状态信息**[ID, 优先级]**
4. **HYPERVISOR**往**GIC LIST REGISTER TO**注册一个虚拟**IRQ****[初始状态位 PENDING]**
5. **GIC VIRTUAL CPU INTERFACE**会触发一个虚拟**IRQ**信号到**CPU**
6. **CPU**退出**HYPERVISOR**返回虚拟机所在的**EL1**或**EL0**



VIRTUAL INTERRUPT SIGNALLING (GIC)

1. 外设中断到达**GIC**
2. 并通过**PHYSICAL CPU INTERFACE**发送一个物理中断到**CPU**
3. **CPU**会**TRAP**到**EL2, HYPERVISOR**通过**GIC_IAR1**响应中断，并获取中断的状态信息(**ID, 优先级**)
4. **HYPERVISOR**往**GIC LIST REGISTER TO**注册一个虚拟**IRQ**[初始状态位**PENDING**]
5. **GIC VIRTUAL CPU INTERFACE**会触发一个虚拟**IRQ**信号到**CPU**
6. **CPU**退出**HYPERVISOR**返回虚拟机所在的**EL1**或**EL0**
7. 因为**VIRQ**的信号，**CPU**会进入**VIRTUAL IRQ EXCEPTION, GUESTOS**通过**VIRTUAL CPU INTERFACE**完成**ACK/EOI**等动作

VIRTUAL CPU INTERFACE由硬件实现，虚拟机访问不会造成**trap**



VIRTUAL INTERRUPT INJECT

WITHOUT GIC

不通过**GIC**的虚拟中断注入

VIRTUAL INTERRUPT SIGNALLING

- **SCR_EL3.IRQ = 0, SECURE MONITOR** 不处理物理IRQ
- **HCR_EL2.IM0 = 1, HYPERVISOR** 处理物理IRQ
- **EL1 CPU** 上的中断状态位没有 MASK, 可以正常接收虚拟中断

Non-secure state
SCR_EL3.I IRQ = 0
HCR_EL2.I IM0 = 1
EL1 PSTATE.I I = 0 (EL1 IRQs not masked)

App (EL0)

Guest OS (EL1)

IRQ handler

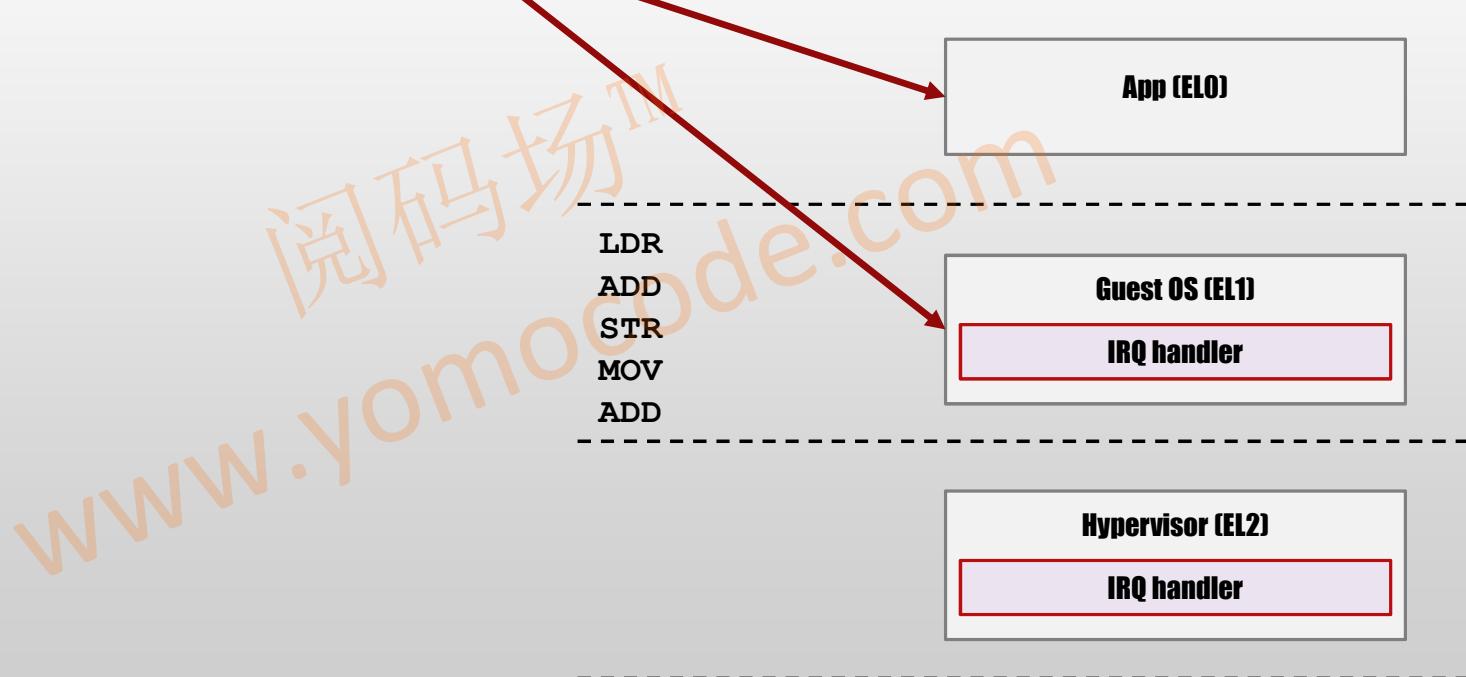
Hypervisor (EL2)

IRQ handler

VIRTUAL INTERRUPT SIGNALLING

1. THE GUEST OS 在 EL1 执行或者 GUEST OS 用
户态 APP 在 EL0 执行

Non-secure state
SCR_EL3.I IRQ = 0
HCR_EL2.I IMO = 1
EL1 PSTATE.I I = 0 (EL1 IRQs not masked)



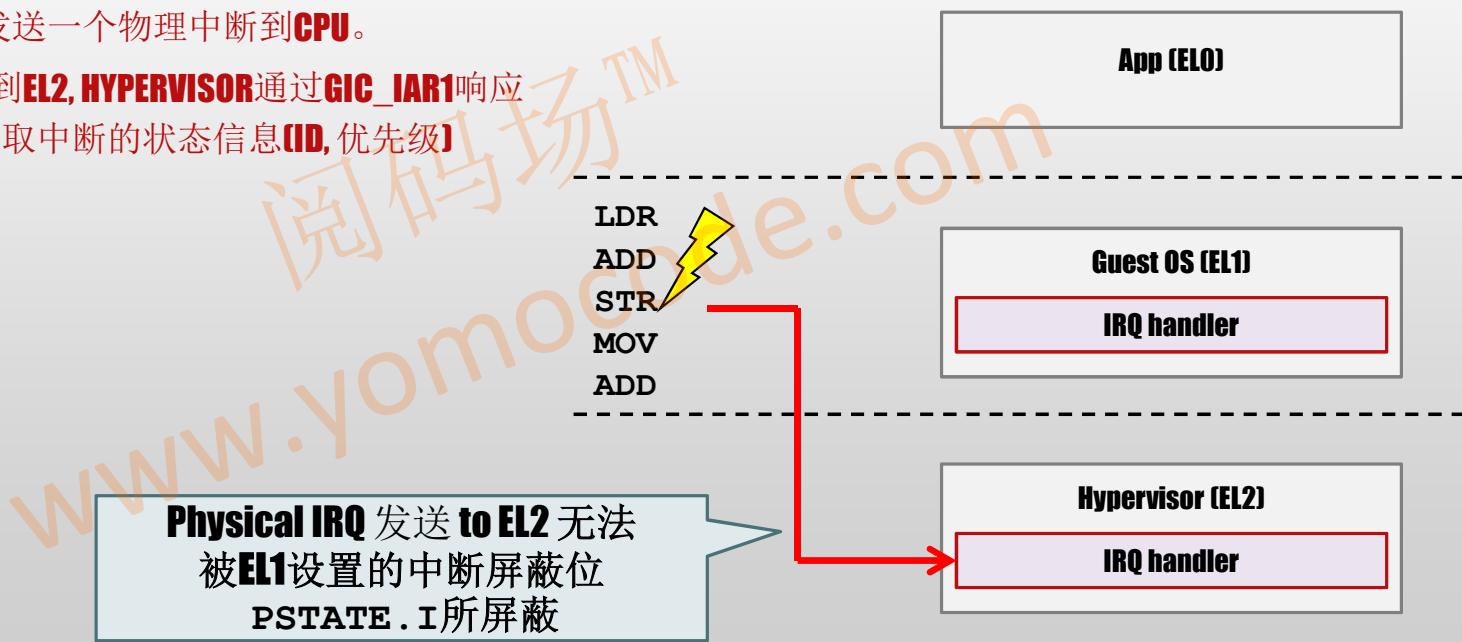
VIRTUAL INTERRUPT SIGNALLING

1. THE GUEST OS 在 EL1 执行或者 GUEST OS 用户态 APP 在 EL0 执行

2. 外设中断到达 GIC，并通过 PHYSICAL CPU INTERFACE 发送一个物理中断到 CPU。

3. CPU 会 TRAP 到 EL2，HYPERVISOR 通过 GIC_IAR1 响应中断，并获取中断的状态信息 [ID, 优先级]

Non-secure state
SCR_EL3.I IRQ = 0
HCR_EL2.I IMO = 1
EL1 PSTATE.I = 0 (EL1 IRQs not masked)



VIRTUAL INTERRUPT SIGNALLING

1. THE GUEST OS 在EL1执行或者**GUEST OS** 用户态**APP**在**EL0**执行

2. 外设中断到达**GIC**, 并通过**PHYSICAL CPU INTERFACE**发送一个物理中断到**CPU**。

3. **CPU**会**TRAP**到**EL2, HYPERVISOR**通过**GIC_IAR1**响应中断，并获取中断的状态信息**[ID, 优先级]**

4. **EL2 IRQ HANDLER**判断该中断是**HYPERVERISOR**自己处理还是同虚拟中断注入虚拟机处理

Non-secure state
SCR_EL3.I IRQ = 0
HCR_EL2.I IMO = 1
EL1 PSTATE.I I = 0 (EL1 IRQs not masked)

LDR
ADD
STR
MOV
ADD

App (EL0)

Guest OS (EL1)

IRQ handler

Hypervisor (EL2)

IRQ handler

VIRTUAL INTERRUPT SIGNALLING

1. THE GUEST OS 在 EL1 执行或者 GUEST OS 用户态 APP 在 EL0 执行

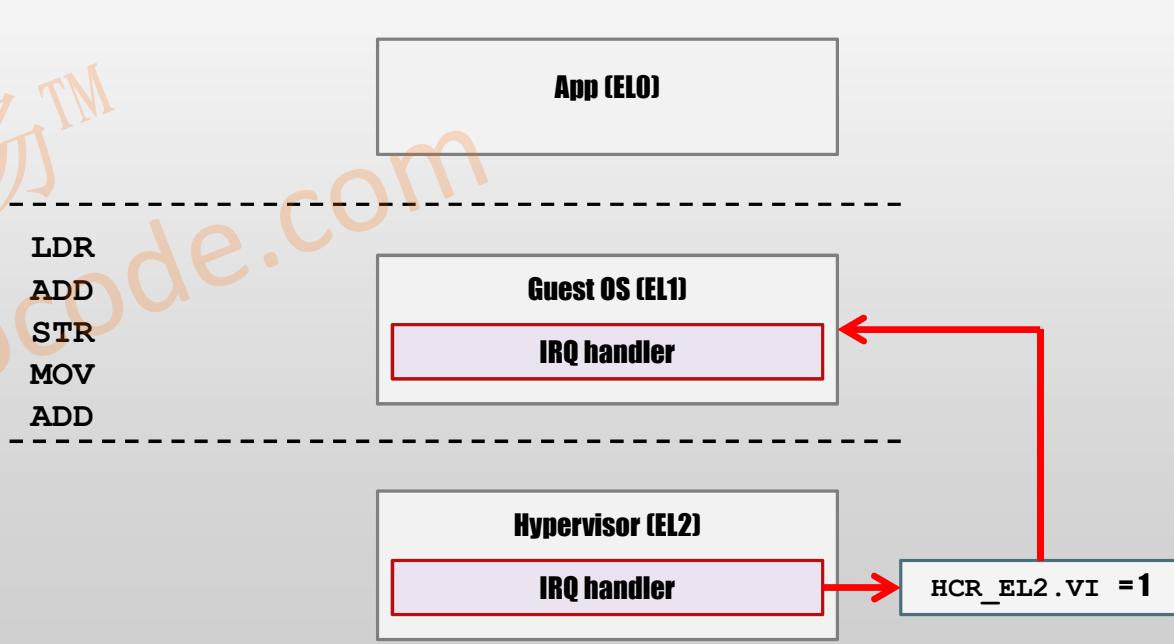
2. 外设中断到达 GIC，并通过 PHYSICAL CPU INTERFACE 发送一个物理中断到 CPU。

3. CPU 会 TRAP 到 EL2, HYPERVISOR 通过 GIC_IAR1 响应中断，并获取中断的状态信息 [ID, 优先级]

4. EL2 IRQ HANDLER 判断该中断是 HYPERVISOR 自己处理还是同虚拟中断注入虚拟机处理

5. HYPERVISOR 通过 HCR_EL2.VI 向虚拟机注入一个虚拟中断，没有使用 GIC LIST REGISTER

Non-secure state
SCR_EL3.I IRQ = 0
HCR_EL2.I IMO = 1, VI = 1
EL1 PSTATE.I = 0 (EL1 IRQs not masked)



VIRTUAL INTERRUPT SIGNALLING

1. THE GUEST OS 在 EL1 执行或者 GUEST OS 用户态 APP 在 EL0 执行

2. 外设中断到达 GIC，并通过 PHYSICAL CPU INTERFACE 发送一个物理中断到 CPU。

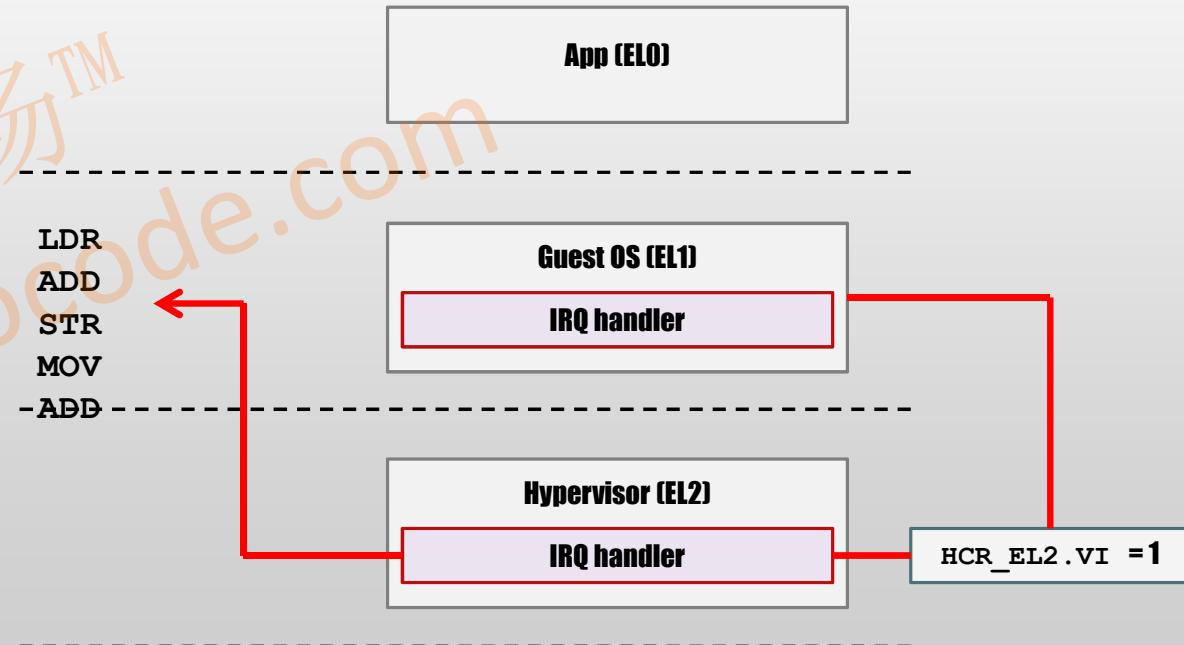
3. CPU 会 TRAP 到 EL2，HYPERVISOR 通过 GIC_IAR1 响应中断，并获取中断的状态信息 [ID, 优先级]

4. EL2 IRQ HANDLER 判断该中断是 HYPERVISOR 自己处理还是同虚拟中断注入虚拟机处理

5. HYPERVISOR 通过 HCR_EL2.VI 向虚拟机注入一个虚拟中断，没有使用 GIC LIST REGISTER

6. CPU 退出 HYPERVISOR 返回虚拟机所在的 EL1 或 EL0

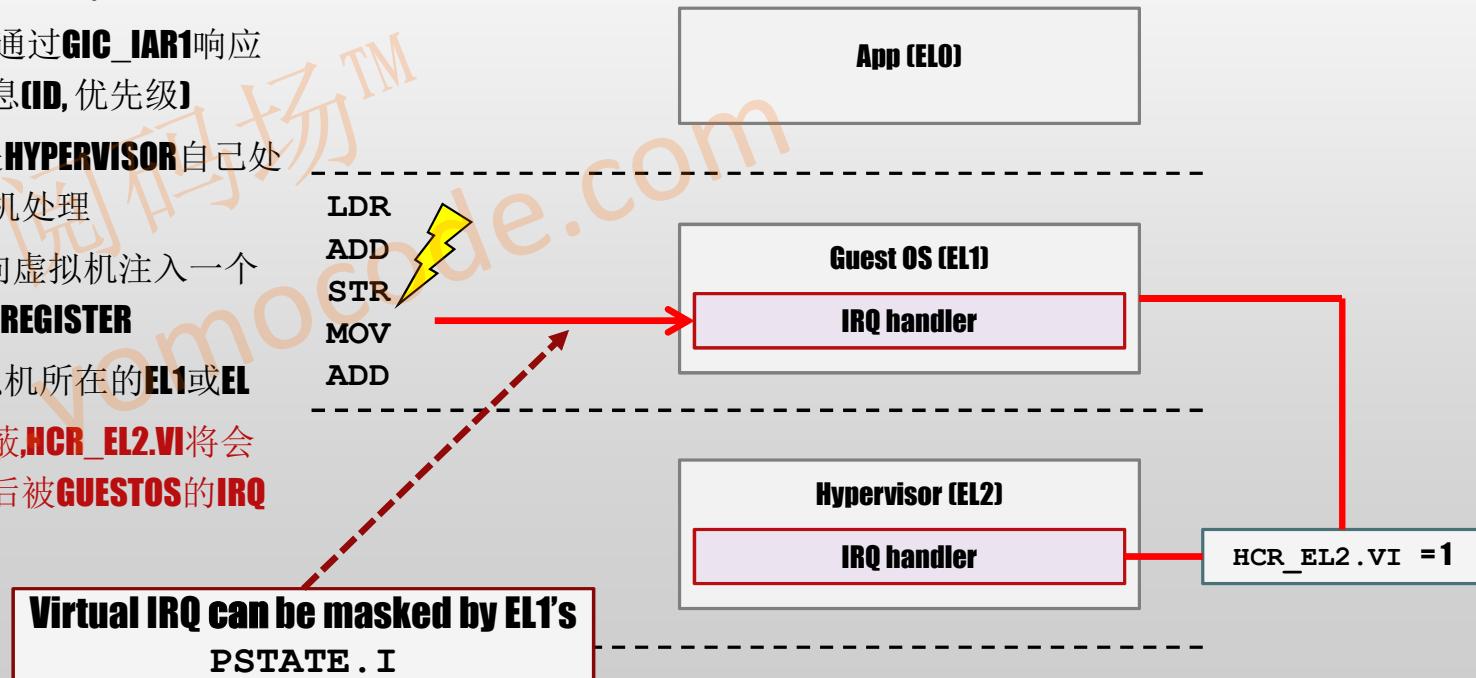
Non-secure state
SCR_EL3.I IRQ = 0
HCR_EL2.I IMO = 1, VI = 1
EL1 PSTATE.I = 0 (EL1 IRQs not masked)



VIRTUAL INTERRUPT SIGNALLING

1. THE GUEST OS 在 EL1 执行或者 GUEST OS 用户态 APP 在 EL0 执行
2. 外设中断到达 GIC，并通过 PHYSICAL CPU INTERFACE 发送一个物理中断到 CPU。
3. CPU 会 TRAP 到 EL2, HYPERVISOR 通过 GIC_IAR1 响应中断，并获取中断的状态信息 [ID, 优先级]
4. EL2 IRQ HANDLER 判断该中断是 HYPERVISOR 自己处理还是同虚拟中断注入虚拟机处理
5. HYPERVISOR 通过 HCR_EL2.VI 向虚拟机注入一个虚拟中断，没有使用 GIC LIST REGISTER
6. CPU 退出 HYPERVISOR 返回虚拟机所在的 EL1 或 EL
7. 因为 EL1 的 PSTATE.I 没被屏蔽，HCR_EL2.VI 将会触发一个虚拟中断异常，然后被 GUESTOS 的 IRQ HANDLER 处理

Non-secure state
SCR_EL3.I IRQ = 0
HCR_EL2.I IMO = 1, VI = 1
EL1 PSTATE.I = 0 (EL1 IRQs not masked)



VIRTUALIZATION IMPROVEMENT IN GICV4

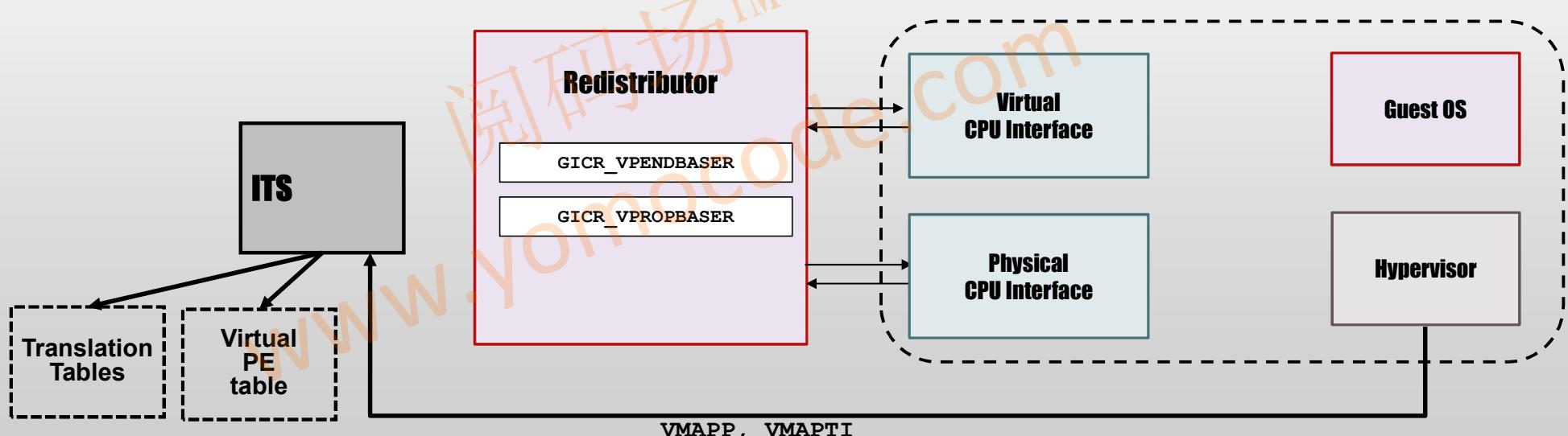
GICV4中的虚拟化改进

VIRTUALIZATION IN GICV4

- **GICV4** 新增了虚拟中断的直接注入(**DIVL: DIRECT INJECT VIRTUAL LPI**)
 - 在一些场景下，物理中断可以不需要先进入**HYPERVERISOR**，通过事先的配置，硬件直接完成虚拟中断注入
 - 只适用于**LPI**，需要**ITS**支持
- **HYPERVERISOR**事先在 **ITS** 中完成了虚拟中断和物理中断的映射
 - 映射包括：
 - **EVENT_ID/DEVICE_ID** 到物理中断的映射
 - 分配虚拟中断号
 - 该虚拟中断将要发送给那几个**VCPU**
 - 那几个物理**CPU** 将会运行这些**VCPU**
- 当**VCPU**被调度运行时，如果该物理中断触发了，硬件直接注入虚拟中断，虚拟机不退出到**HYPERVERISOR**
 - 如果**VCPU**没有被调度，一个**DOOR-BELL**物理中断会触发，**HYPERVERISOR**会尽快调度**VCPU**来处理中断
- **GIC LIST REGISTERS** 仍然可用
 - 因为**SPI**中断还是需要**HYPERVERISOR**通过上述两种方式注入

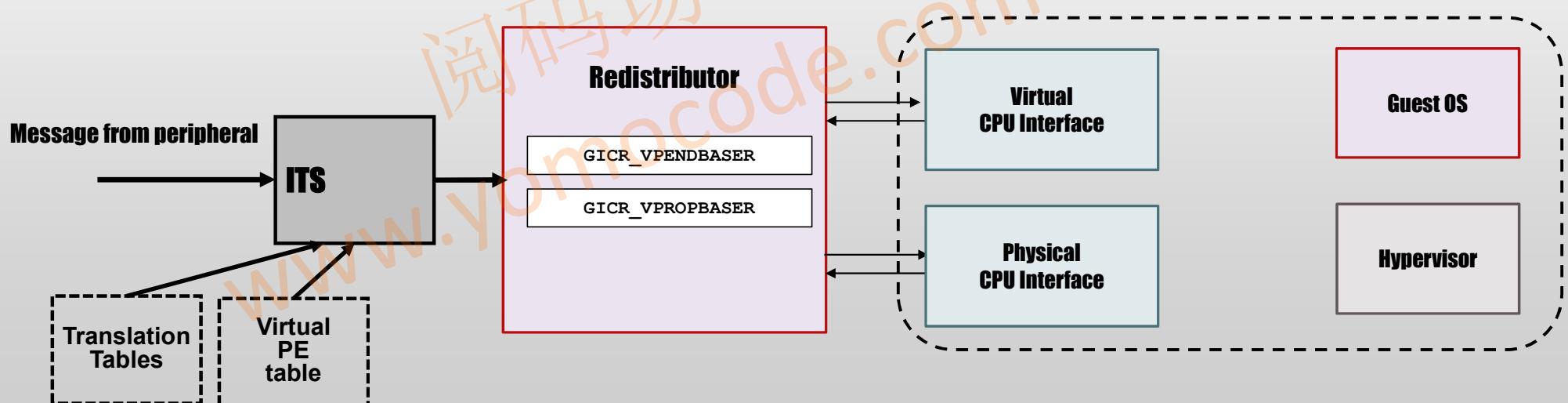
GICV4 EXAMPLE (1)

- **HYPervisor** 通过**ITS**命令来完成中断映射
 - **VMAPI**和**VMAPTI**将**EVENTID/DEVICEID**映射到**VIRQ**和**VCPU**
 - 也可以将**DOORBELL**中断绑定到指定物理**CPU**上
 - **VMAPP** 将**VCPU**映射到物理**CPU**上



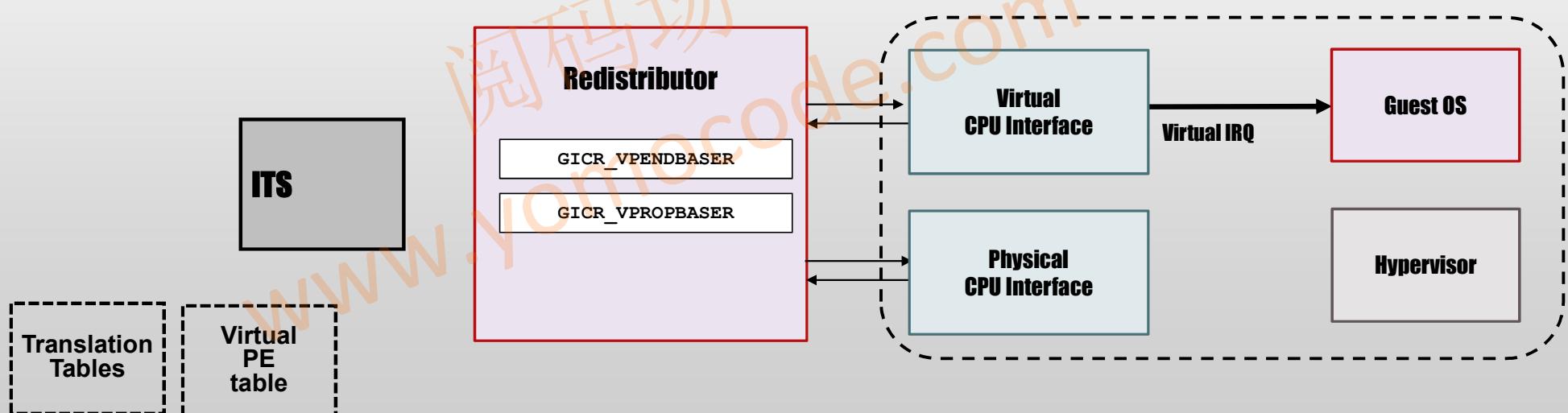
GICV4 EXAMPLE [2]

- 当中断发生时, **ITS** 通过 **EVENTID/DEVICEID** 获取转换信息
 - 包括 **VIRQ** 和 **VCPU** 信息
 - DOORBELL** 的物理中断信息
 - VCPU** 映射的物理 **CPU** 所在的 **REDISTRIBUTOR** 信息



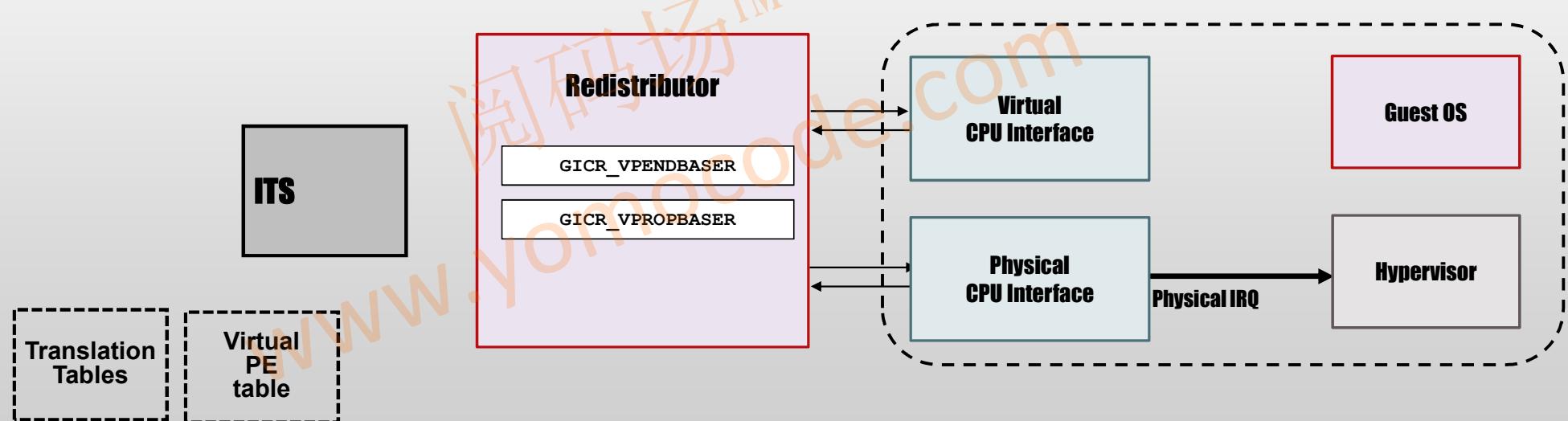
GICV4 EXAMPLE [3]

- **REDISTRIBUTOR** 检查当前被调度在物理**CPU**上的**VCPUs**是不是该中断所对应的
 - 通过**GICR_VPENDBASER**的值来检查，每个**VCPUs**值不同
- 如果是该中断对应的**VCPUs**, **REDISTRIBUTOR** 将中断转发到 **CPU INTERFACE**
 - **VIRTUAL CPU INTERFACE**会触发一个虚拟中断



GICV4 EXAMPLE (4)

- 如果当前**VCPU**不是该中断对应的:
 - **VIRQ**会被记录在**VIRTUAL PENDING TABLE**, 标记位**PENDING**
 - **DOORBELL** 中断会通知**HYPERVISOR**尽快调度



实践环境



X86_64 QEMU + Linux：通过**QEMU**来模拟一台**arm64**的计算平台**[ARM64 HOST]**



Ubuntu 18.04 arm64 作为**ARM64 HOST**的基础操作系统 (**Linux + KVM = hypervisor**)



AARCH64 QEMU: 在**ARM64 Host** 上用来起虚拟机



Ubuntu 16.04 arm64 作为虚拟机的操作系统**[ARM64 GUEST OS]**

谢谢!
THANK YOU!

