GIC-based Interrupt handling mechanism on AArch64

Dennis Chen @ Arm

CLK17

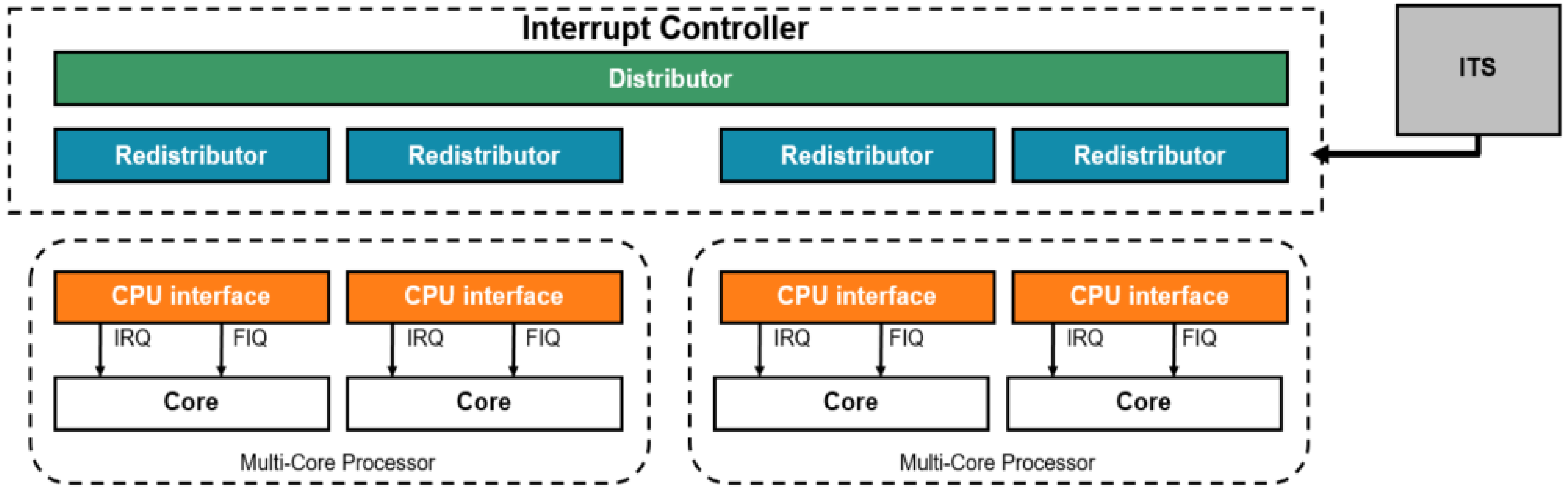
Oct 21, 2017 BEIJING

Agenda

- Part 1 - GICv3 Introduction
- Part 2 - Interrupt handling on AArch64
- Q & A

GICv3 Architecture

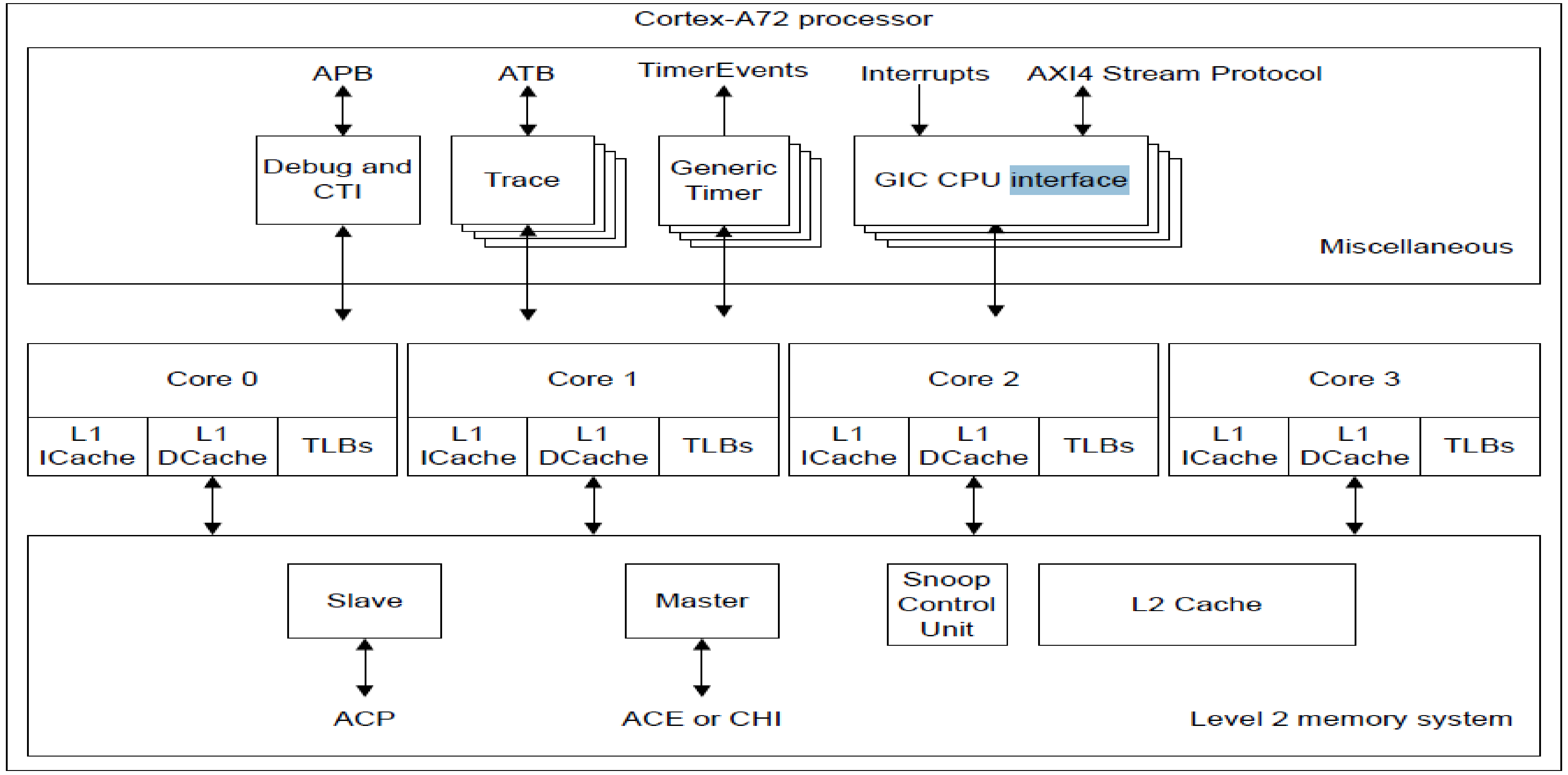
- The Generic Interrupt Controller (GIC) Architecture defines a programmers model for interrupt controllers



- Distributor and Redistributor interfaces are memory mapped
- CPU interface accessed as system registers

GICv3 Architecture

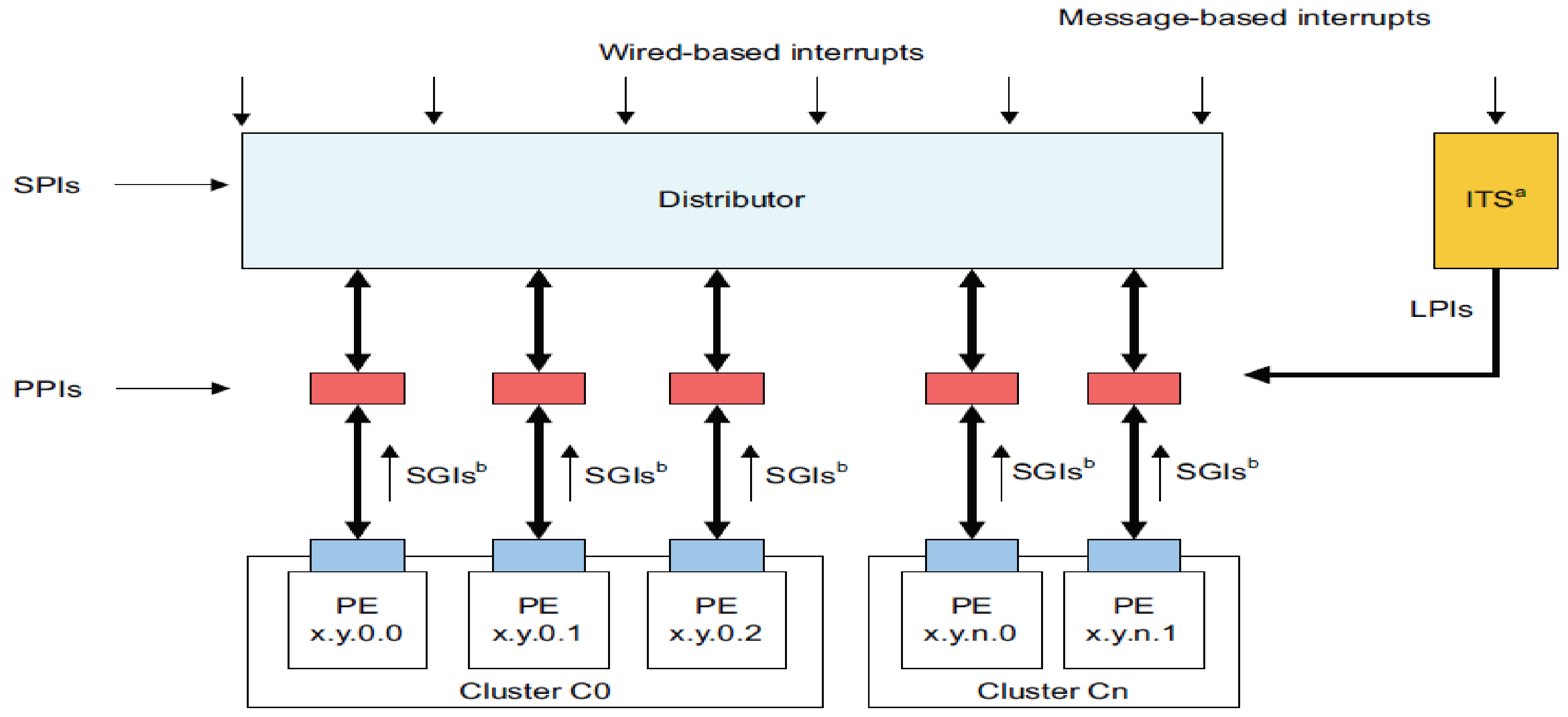
- **The GIC CPU interface in an Arm MPCore Processor**



GIC interrupt types

- **SPI – Shared Peripheral Interrupt**
  - Peripheral interrupt, available to all the cores using the interrupt controller
  - INTIDs 32 to 1019
- **PPI – Private Peripheral Interrupt**
  - Peripheral interrupt which is private to an individual core
  - INTIDs 16 to 31
- **LPI – Locality-specific Peripheral Interrupt** ***(new in GICv3)***
  - Peripheral interrupt, typically routed by an ITS
  - INTIDs 8192+
- **SGI – Software Generated Interrupt**
  - Triggered by writing to a register within the interrupt controller
  - INTIDs 0 to 15
- **INTIDs 1020 to 1023 are reserved and have special purposes**
  - Take 1023 as an example. INTID=1023 means no pending interrupt.

GIC logic partition

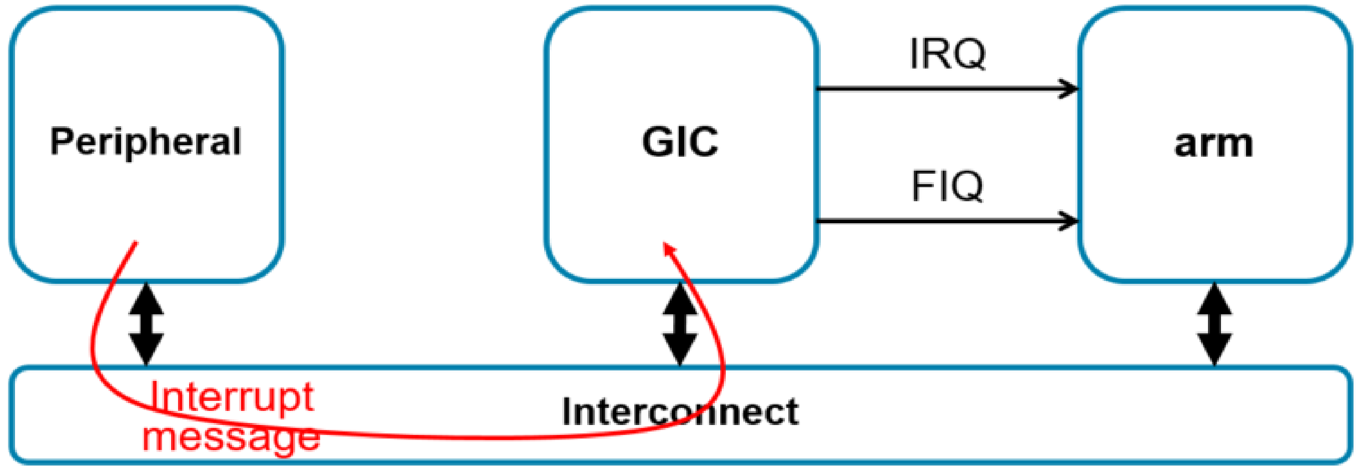


🟥	Redistributor	🟦	CPU interface
⬜	Distributor	🟨	Interrupt Translation Service

a. The inclusion of an ITS is optional, and there might be more than one ITS in a GIC.
b. SGIs are generated by a PE and routed through the Distributor.

Message based interrupt – new in GICv3

- **GICv3 adds support for message based interrupts (MBI)**
  - Instead of using a dedicated signal, a peripheral writes a register in the GIC to trigger an interrupt

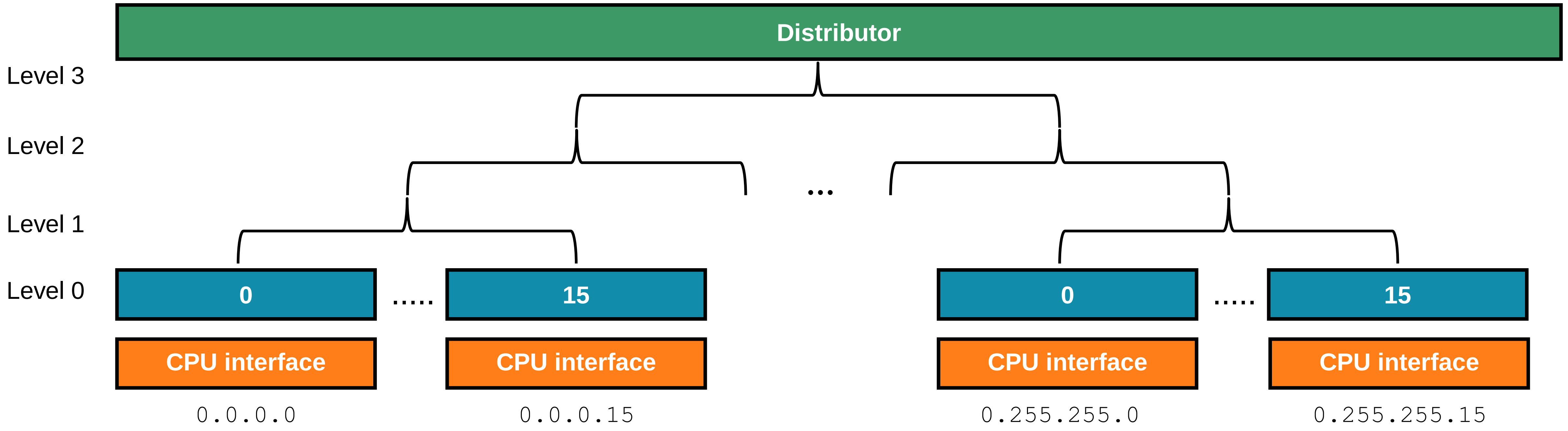


- **Why?**
  - Can reduce the number of wires needed and ease routing
  - Matches model used by PCIe
- **In most cases software should not care whether interrupt is an MBI or not**
  - Does not change how state machine operates

Interrupt configuration

- **Each INTID has a number of settings associated with it:**
- **Enable**
  - Only enabled interrupts can be forwarded to a core
  - However, disabled interrupts can still become pending if the source is asserted
- **Priority**
  - Each interrupt has an 8-bit priority associated with it
  - `0x00` is the highest priority, `0xFF` is the lowest priority
  - A GIC can implement less than 8 bits of priority, if so only the most significant bits are used
- **Configuration**
  - Whether the interrupt is level-sensitive or edge-triggered
- **Security/Group and Target**
  - Covered on the next few slides

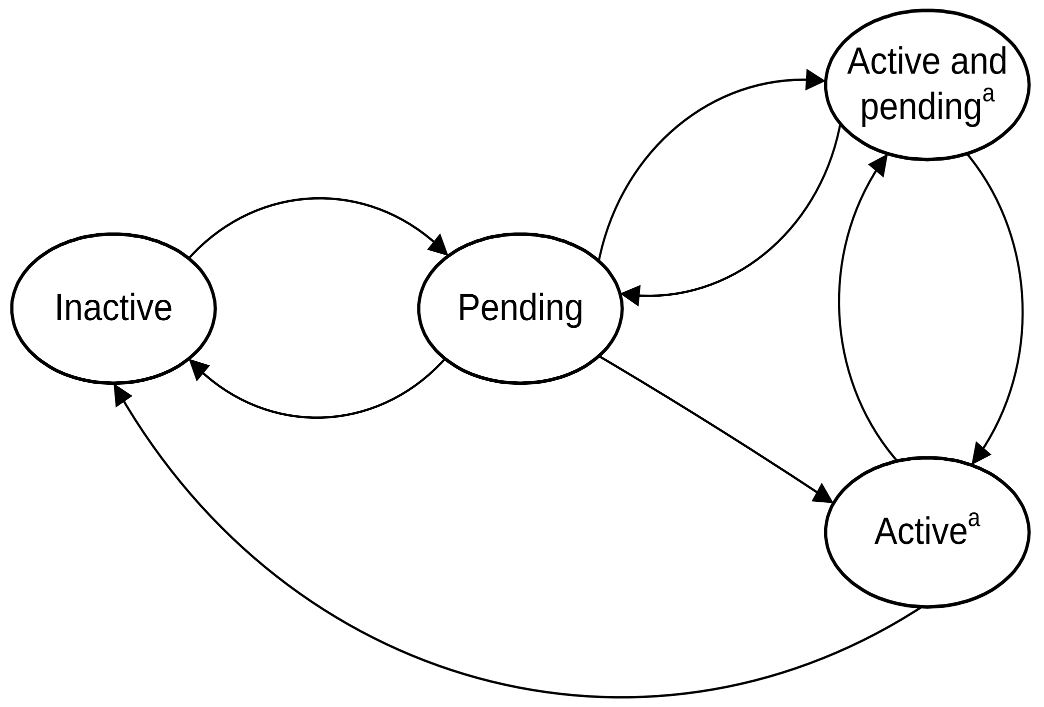
SPI routing in GICv3 – Affinity levels



- **GICv3 identifies the attached CPU using a four level ID, called affinity**
  - Matches affinity mapping defined for ARMv8-A, reported in the `MPIDR_EL1`
- **For each SPI there is a `GICD_IROUTERn` register which controls where the interrupt is routed to**
  - Two options:
    - Send to any connected CPU (referred to as "1 of N")
    - Send to one specific CPU (affinity coordinates of CPU written to register)

Interrupt states

- **Inactive**
  - interrupt is not active and not pending
- **Pending**
  - interrupt is asserted but not yet being serviced
- **Active**
  - interrupt is being serviced but not yet complete
- **Active & Pending**
  - interrupt is both active and pending



a. Not applicable for LPIs.

- **Interrupt goes:**
  - Inactive → Pending when the interrupt is asserted
  - Pending → Active when a CPU acknowledges the interrupt by reading the Interrupt Acknowledge Register (IAR)
  - Active → Inactive when the same CPU deactivates the interrupt by writing the End of Interrupt Register (EOIR)

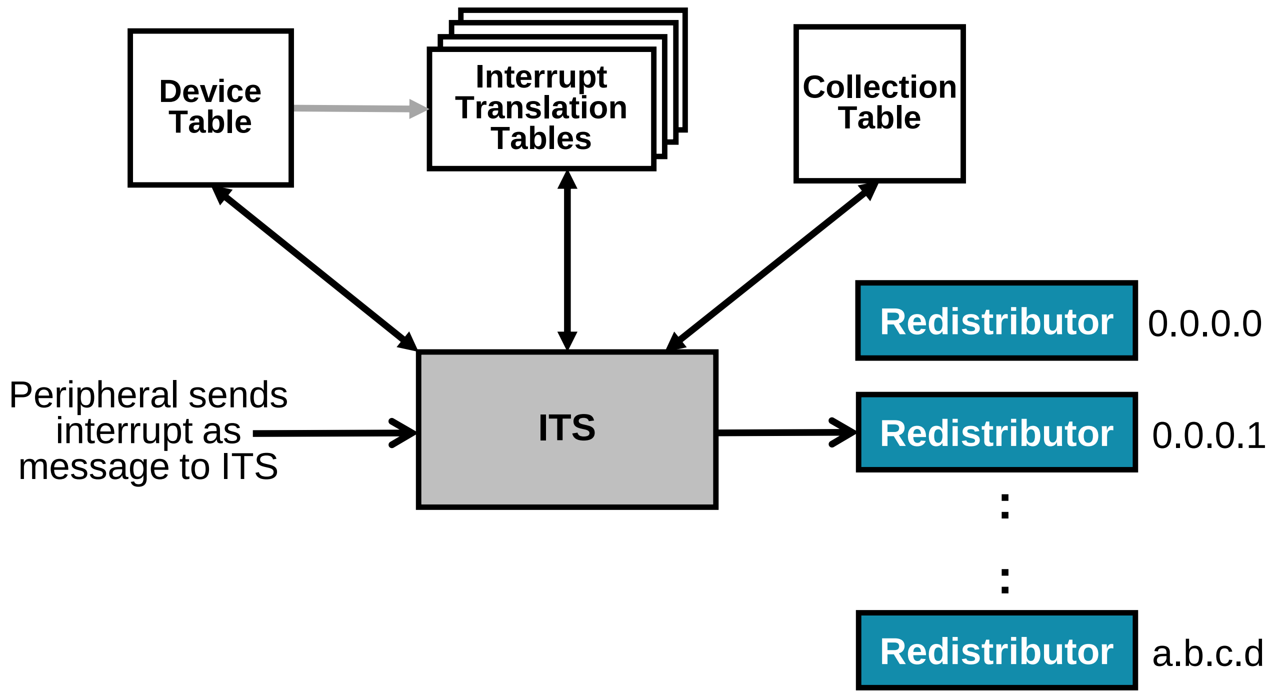
Interrupt Acknowledging

- **On taking an interrupt software must read one of the Interrupt Acknowledge registers**
  - This returns the INTID of the interrupt and updates the state machine

- **There are different registers for Group 0 and Group 1 interrupts**
  - `ICC_IAR0_EL1` – for Group 0 interrupts
  - `ICC_IAR1_EL1` – for Group 1 interrupts

- **When an interrupt is acknowledged the Running Priority of the CPU interface takes on the priority of the interrupt**
  - Current value reported by `ICC_RPR_EL1`
  - Matters for pre-emption, covered later

ITS – PCI MSI/MSIx



- **An Interrupt Translation Service (or ITS) maps interrupts to INTIDs and Redistributors**
- **How is an interrupt translated?**
  - Peripheral sends interrupt as a message to the ITS
    - The message specifies the DeviceID (which peripheral) and an EventID (which interrupt from that peripheral)
  - ITS uses the DeviceID to index into the Device Table
    - Returns pointer to a peripheral specific Interrupt Translation Table
  - ITS uses the EventID to index into the Interrupt Translation Table
    - Returns the INTID and Collection ID
  - ITS uses the Collection ID to index into the Collection Table
    - Returns the target Redistributor
  - ITS forwards interrupt to Redistributor

Interrupt entry on AArch64

- **Vector Table – A number of consecutive word-aligned addresses in memory, starting at the *Vector Base Address.* Each EL has an associated *Vector Base Address Register* (VBAR)**

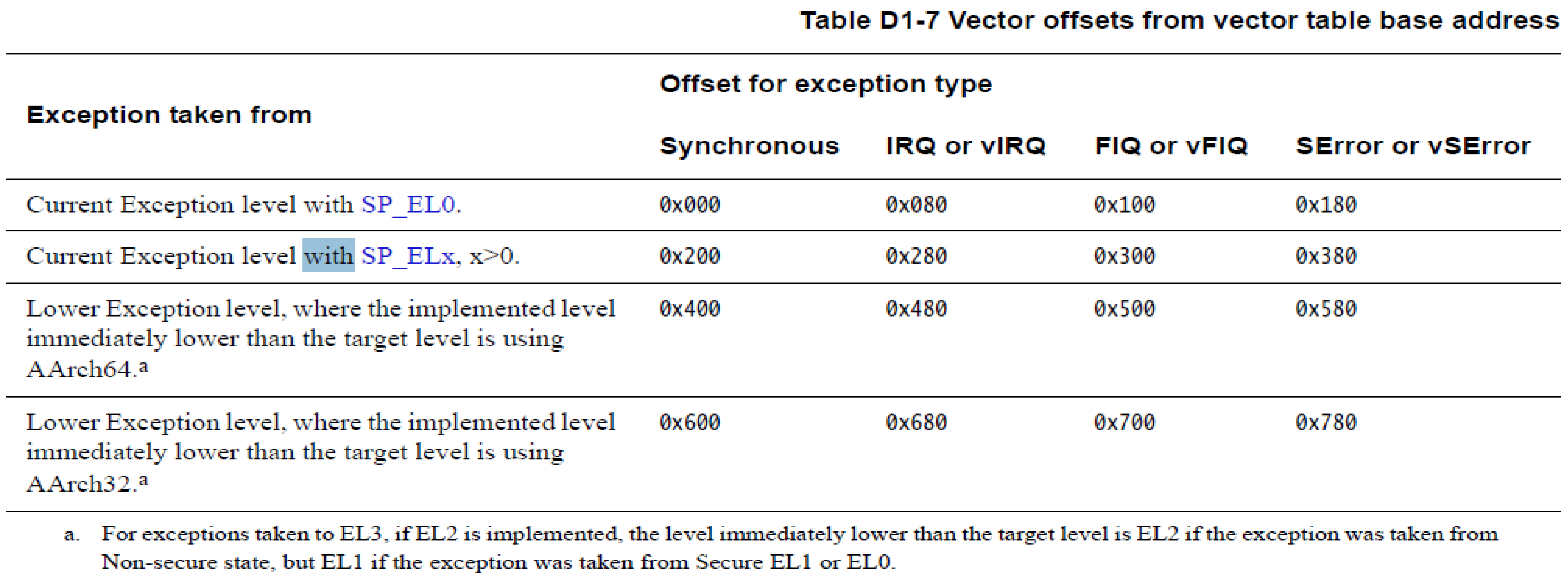


Table D1-7 Vector offsets from vector table base address

Exception taken from	**Offset for exception type**			
	Synchronous	**IRQ or vIRQ**	**FIQ or vFIQ**	**SError or vSError**
Current Exception level with SP_EL0.	0x000	0x080	0x100	0x180
Current Exception level with SP_ELx, x>0.	0x200	0x280	0x300	0x380
Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.[a]	0x400	0x480	0x500	0x580
Lower Exception level, where the implemented level immediately lower than the target level is using AArch32.[a]	0x600	0x680	0x700	0x780

a. For exceptions taken to EL3, if EL2 is implemented, the level immediately lower than the target level is EL2 if the exception was taken from Non-secure state, but EL1 if the exception was taken from Secure EL1 or EL0.

EL0 has no corresponding VBAR

Interrupt entry on AArch64

- **Vector Table establishment in kernel**

arch/arm64/kernel/entry.S:

```
.align      11
ENTRY(vectors)
  …
  ventry    el1_irq
  …
END(vectors)
```

ventry el1_irq is:

```
.align 7
b    el1_irq
```

The IRQ handling chain:

el1_irq → irq_handler →handle_arch_irq

drivers/irqchip/irq-gic-v3.c:

set_handle_irq(gic_handle_irq), so – handle_arch_irq = gic_handle_irq

The main interrupt handling entry function is:

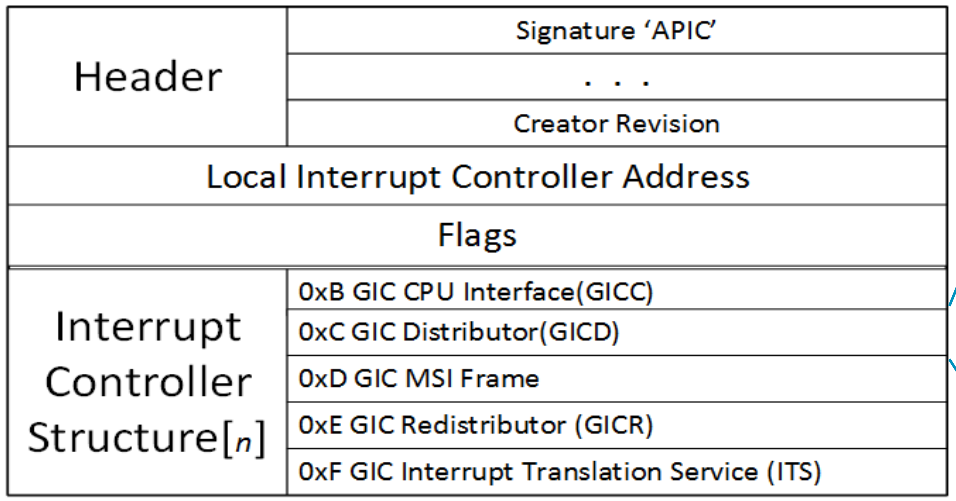
```
static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
{
        u32 irqnr;  //INTID
        do {
                irqnr = gic_read_iar();  // Read ICC_IAR1_EL1, get INTID & ACK interrupt
                gic_write_eoir(irqnr);
                handle_irq();
        } while (irqnr != ICC_IAR1_EL1_SPURIOUS);  // ICC_IAR1_EL1_SPURIOUS = 1023
}
```

1023 is the special INTID indicating no pending interrupt.

MADT of GIC

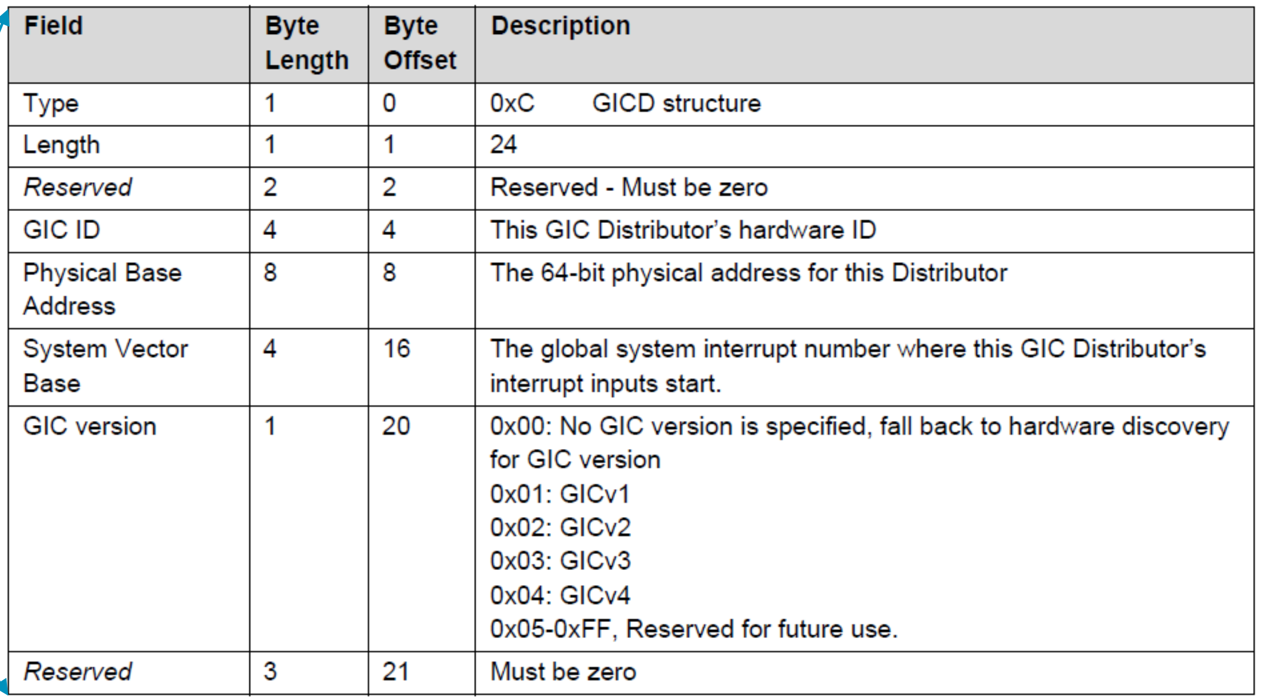
- **ACPI MADT (Multiple APIC Description Table) – Firmware Description of GIC topology/info**

MADT



Header
Signature 'APIC'
. . .
Creator Revision
Local Interrupt Controller Address
Flags
Interrupt Controller Structure[n]
0xB GIC CPU Interface(GICC)
0xC GIC Distributor(GICD)
0xD GIC MSI Frame
0xE GIC Redistributor (GICR)
0xF GIC Interrupt Translation Service (ITS)

Table 5-63 GICD Structure

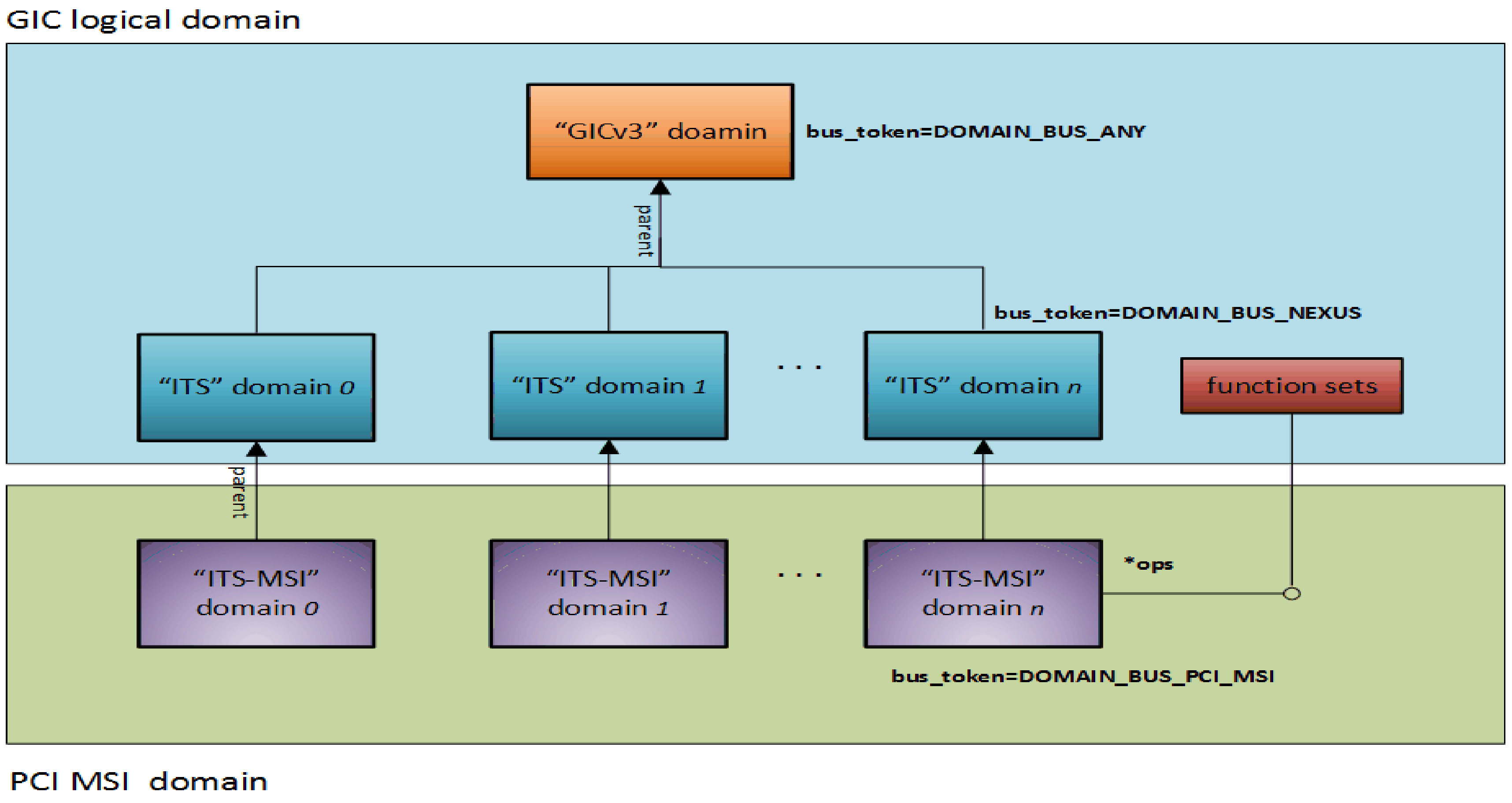


Field	Byte Length	Byte Offset	Description
Type	1	0	0xC GICD structure
Length	1	1	24
Reserved	2	2	Reserved - Must be zero
GIC ID	4	4	This GIC Distributor's hardware ID
Physical Base Address	8	8	The 64-bit physical address for this Distributor
System Vector Base	4	16	The global system interrupt number where this GIC Distributor's interrupt inputs start.
GIC version	1	20	0x00: No GIC version is specified, fall back to hardware discovery for GIC version 0x01: GICv1 0x02: GICv2 0x03: GICv3 0x04: GICv4 0x05-0xFF, Reserved for future use.
Reserved	3	21	Must be zero

gic_acpi_init() is used to parse the MADT and create the irq_domain hierarchy of GIC

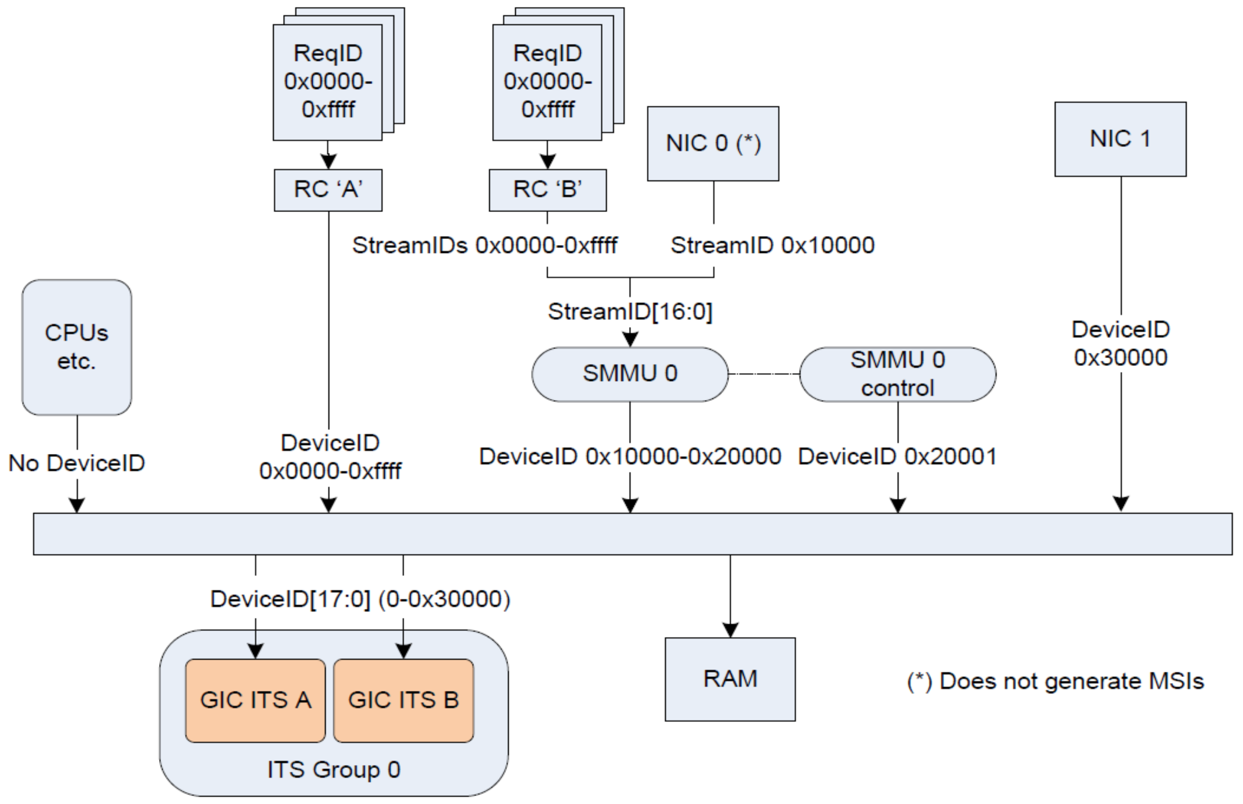
irq_domain hierarchy of GIC

- **irq_domain level setup during GIC initialization**



GIC driver in kernel setups the above hierarchy by parsing the MADT

DeviceID and IORT

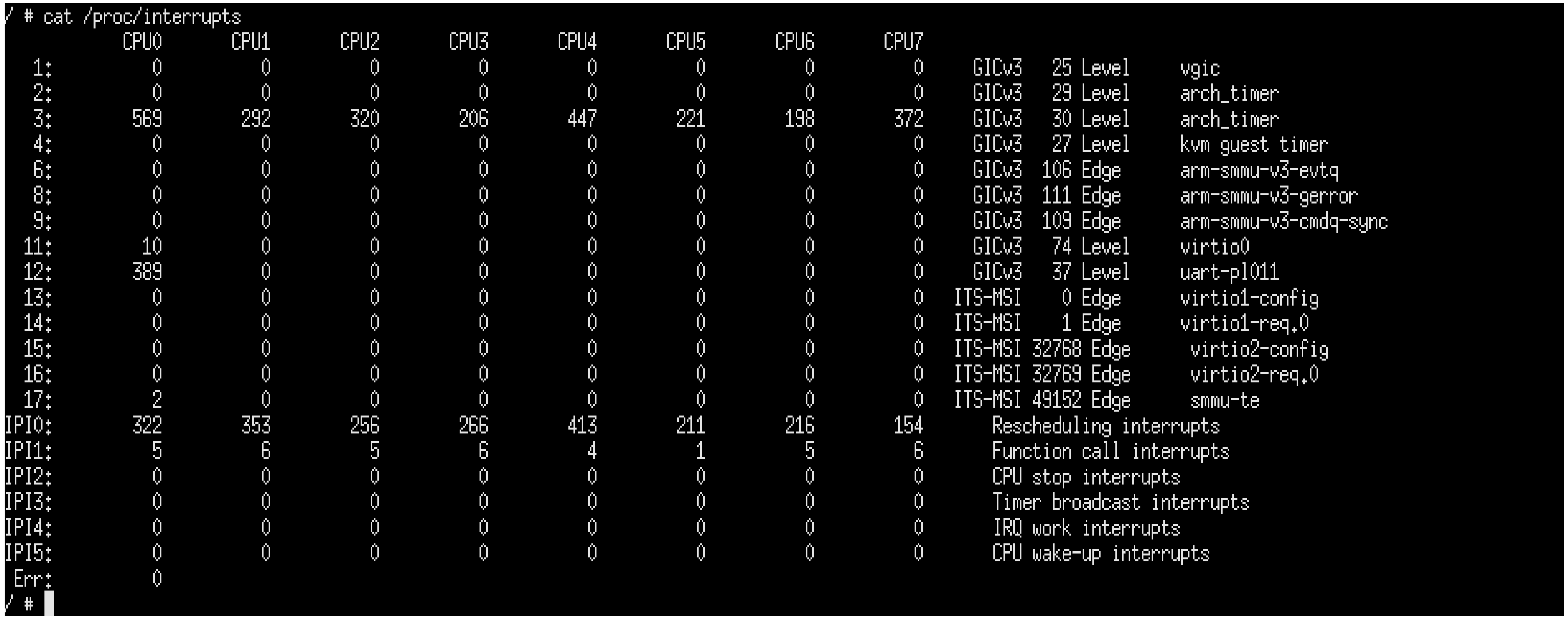


Devices topology on AArch64 platform

- IORT is ACPI table to describe this kind of topology.
- GIC subsystem in kernel parses the IORT and generate the DeviceID.
- The mapping of the ReqID, StreamID and DeviceID is baked in Hardware, the IORT in firmware just describe this mapping, it can't modify the mapping relationship
- iort_msi_map_rid() used to map a ReqID to a DeviceID, then the GIC driver in kernel uses this DeviceID to setup a DTE for interrupt translation.

Interrupts on AArch64

- **Screenshot of interrupts on a 8-core AArch64 Fast Model**



```
/ # cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
  1:          0          0          0          0          0          0          0          0     GICv3  25 Level     vgic
  2:          0          0          0          0          0          0          0          0     GICv3  29 Level     arch_timer
  3:        569        292        320        206        447        221        198        372     GICv3  30 Level     arch_timer
  4:          0          0          0          0          0          0          0          0     GICv3  27 Level     kvm guest timer
  6:          0          0          0          0          0          0          0          0     GICv3 106 Edge      arm-smmu-v3-evtq
  8:          0          0          0          0          0          0          0          0     GICv3 111 Edge      arm-smmu-v3-gerror
  9:          0          0          0          0          0          0          0          0     GICv3 109 Edge      arm-smmu-v3-cmdq-sync
 11:         10          0          0          0          0          0          0          0     GICv3  74 Level     virtio0
 12:        389          0          0          0          0          0          0          0     GICv3  37 Level     uart-pl011
 13:          0          0          0          0          0          0          0          0   ITS-MSI   0 Edge      virtio1-config
 14:          0          0          0          0          0          0          0          0   ITS-MSI   1 Edge      virtio1-req.0
 15:          0          0          0          0          0          0          0          0   ITS-MSI 32768 Edge      virtio2-config
 16:          0          0          0          0          0          0          0          0   ITS-MSI 32769 Edge      virtio2-req.0
 17:          2          0          0          0          0          0          0          0   ITS-MSI 49152 Edge      smmu-te
IPI0:       322        353        256        266        413        211        216        154       Rescheduling interrupts
IPI1:         5          6          5          6          4          1          5          6       Function call interrupts
IPI2:         0          0          0          0          0          0          0          0       CPU stop interrupts
IPI3:         0          0          0          0          0          0          0          0       Timer broadcast interrupts
IPI4:         0          0          0          0          0          0          0          0       IRQ work interrupts
IPI5:         0          0          0          0          0          0          0          0       CPU wake-up interrupts
 Err:         0
/ #
```

Take the smmu-te device as an example, irq =17 hwirq=49152, irq_domain='ITS-MSI'.

Example of PCI MSI/MSI-X usage

- **struct msix_entry**

```
struct msix_entry {
        u32     vector;     // Filled by PCI MSI/MSIX subsystem, the irq
        u16     entry;      // Filled by pci device driver, generally 0,1,… N. N is the total MSI-X vectorS numbers
};
```

- **Enable the MSI/MSI-X**

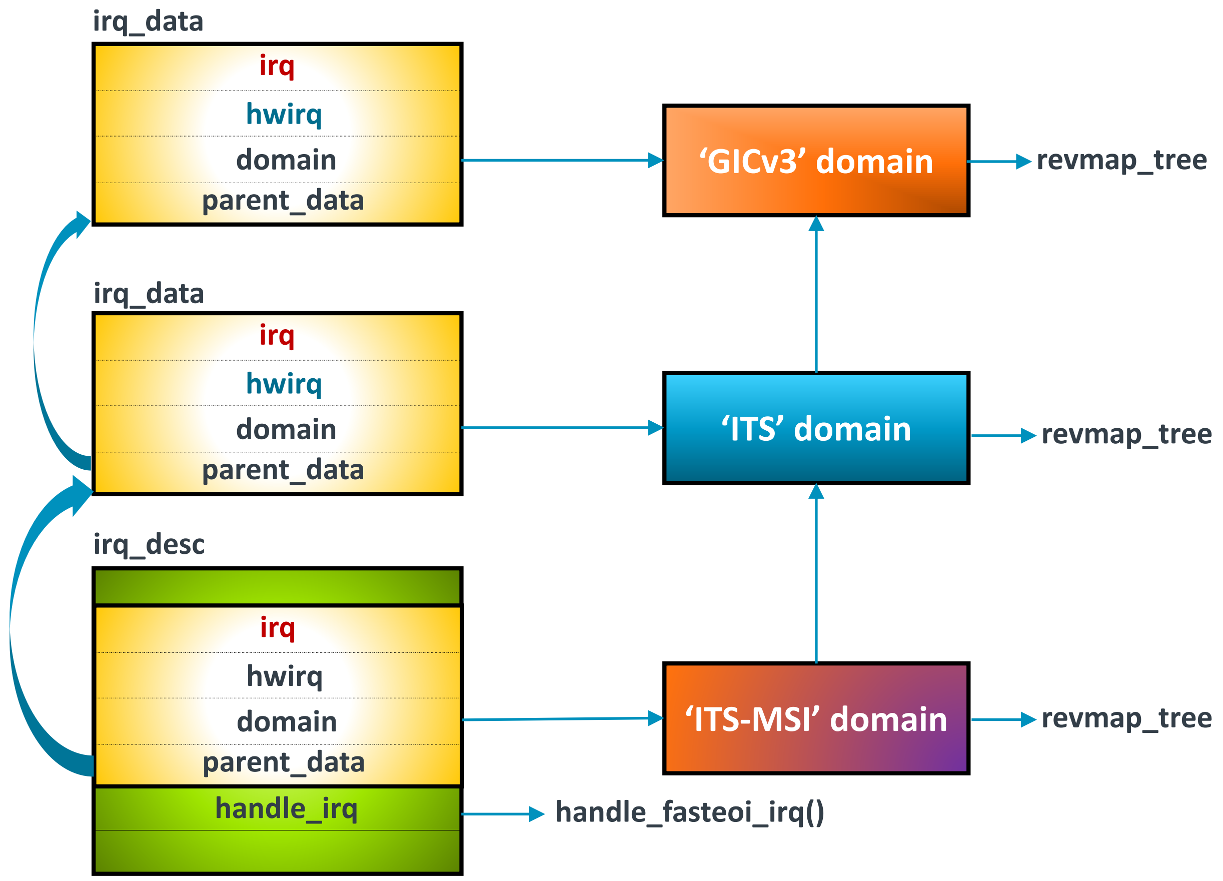
```
struct msix_entry *entries = devm_kmalloc(&pdev->dev, sizeof(struct msix_entry) * numvecs, GFP_KERNEL);
for (i = 0; i < numvecs; i++)
        entries[i].entry = i;
…
ret = pci_enable_msix_range(pdev, entries, 1, numvecs);

if (ret <= 0)
        return ret;
```

- **Register the interrupt handler**

```
for (i = 0; i < nr_rx; i++)
        request_irq(entries[i].vector,rx_pkg_handler, 0, netdev->name, adapter);
```

irq & hwirq Allocation

- **Runtime allocation and mapping**



- For a given pci device:

  pci_msi_get_device_domain(pdev)→

  dom = iort_get_device_domain(&pdev->dev, rid) will find the "ITS-MSI" domain according to the rid of the pdev

- The "ITS-MSI" domain is bound to a pci device(pdev) which happens in:

  pci_device_add() -> pci_set_msi_domain(pdev)

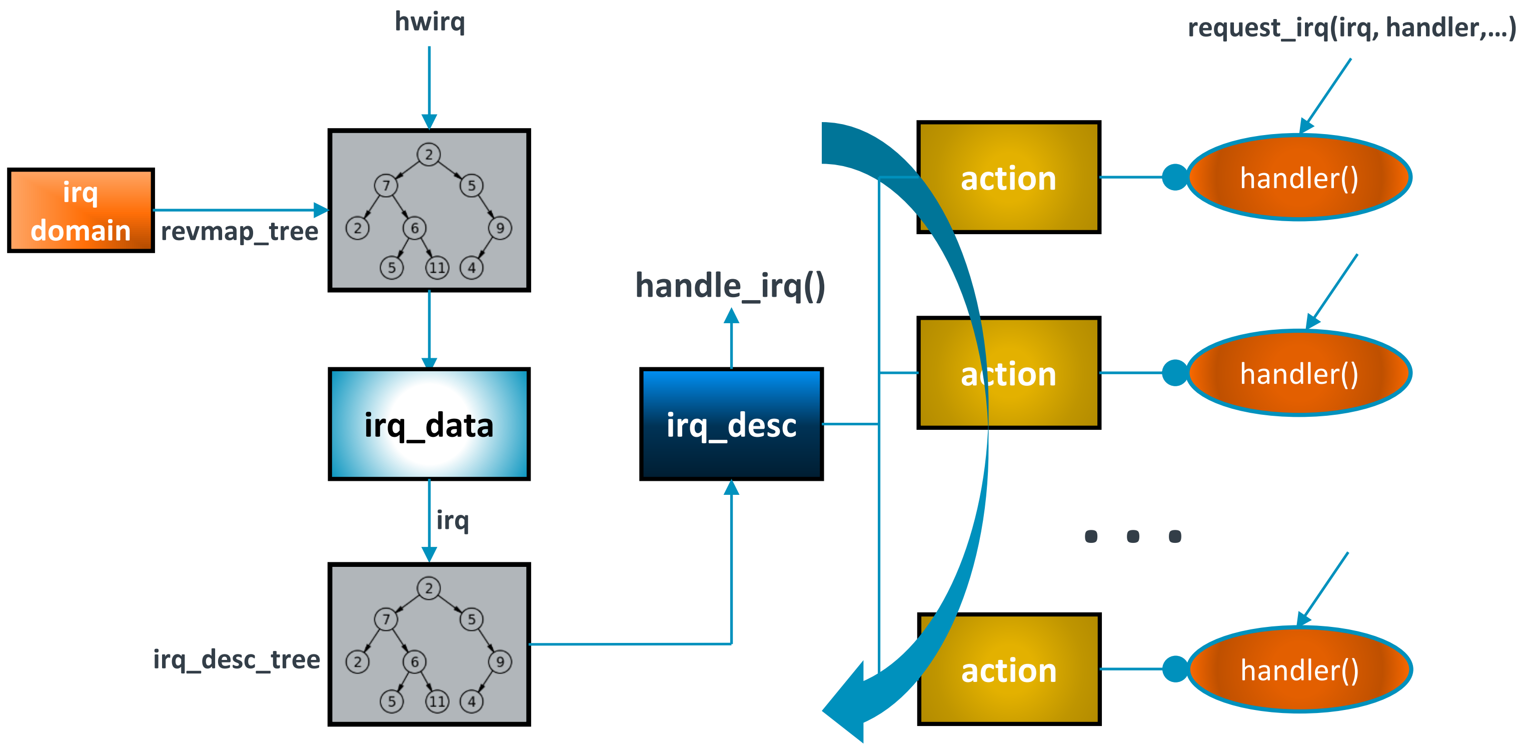
- The irq is global in the irq_domain space, hwirq is not. each domain will allocate a hwirq.

- each domain has a hwirq used as an index of the radix tree:

  radix_tree_insert(&domain->revmap_tree, hwirq,irq_data)

Installation of interrupt handler

- **How the interconnection setup**



- Read the hwirq from the GIC (ICC_IAR)
- Lookup the irq_data from the revmap_tree (indexed by hwirq)
- Lookup the irq_desc from the irq_desc_tree (indexed by irq)
- Invoke irq_desc->handle_irq()
- Call the action->handler() literately

HardIrq & SoftIrq

- **HardIrq part**
  - Can’t receive the external interrupt when in HardIrq – Behavior of the hardware
    PSTATE.{I, F} will be set 1 to mask external IRQ and FIQ
  - Software need to enable the IRQ/FIQ at the end of HardIrq

- **SoftIrq part**
  - Kernel unmask the external interrupt before invoking the softirq
  - Core can receive external interrupt during the softirq handling
  - Both hardirq and softirq are in the interrupt context

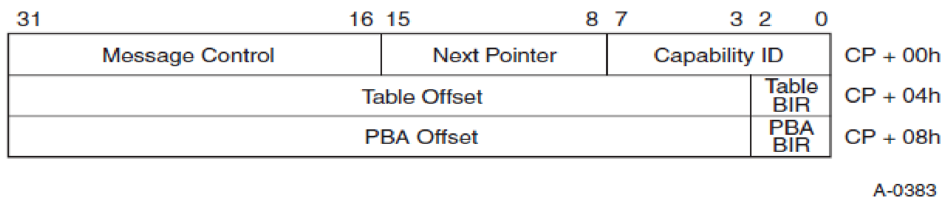
- **Arm instruction enable/disable IRQ**

```
static inline void arch_local_irq_enable(void)
{
        asm volatile("msr daifclr, #2"
                :: "memory");
}
```

```
static inline void arch_local_irq_disable(void)
{
        asm volatile("msr  daifset, #2"
                :: "memory");
}
```

PCI(e) MSI/MSI-X implementation

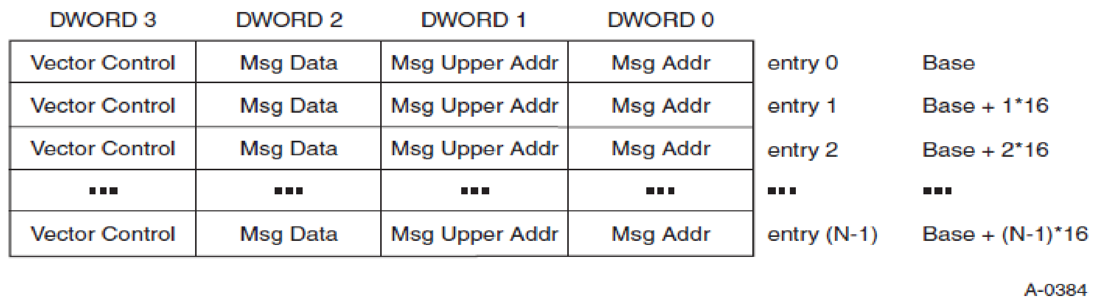
- **MSI-X Capability & Table structure**



31 16	15 8	7 3	2 0	
Message Control	Next Pointer	Capability ID		CP + 00h
Table Offset			Table BIR	CP + 04h
PBA Offset			PBA BIR	CP + 08h

A-0383

Figure 6-10: MSI-X Capability Structure



DWORD 3	DWORD 2	DWORD 1	DWORD 0		
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 0	Base
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 1	Base + 1*16
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 2	Base + 2*16
...	...	...	...	...	...
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry (N-1)	Base + (N-1)*16

A-0384

Figure 6-11: MSI-X Table Structure

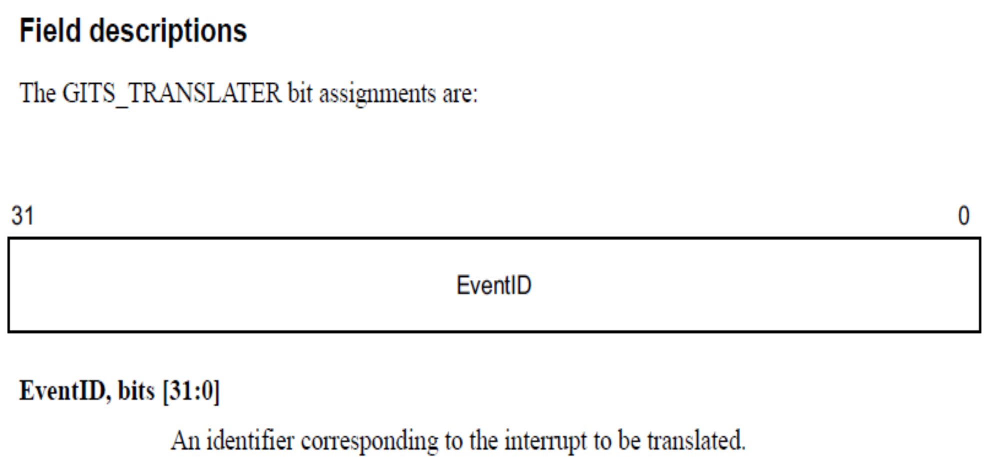
(Table Base) bir = (u8)table_offset & 0x7;

(Table Base) base = dev->resource[bir].start;

Table Phys Addr = base + table_offset & FFFFFFF8;

PCI(e) MSI/MSI-X implementation

- **GIC ITS Translation Register**



Field descriptions

The GITS_TRANSLATER bit assignments are:

31
0
EventID

EventID, bits [31:0]

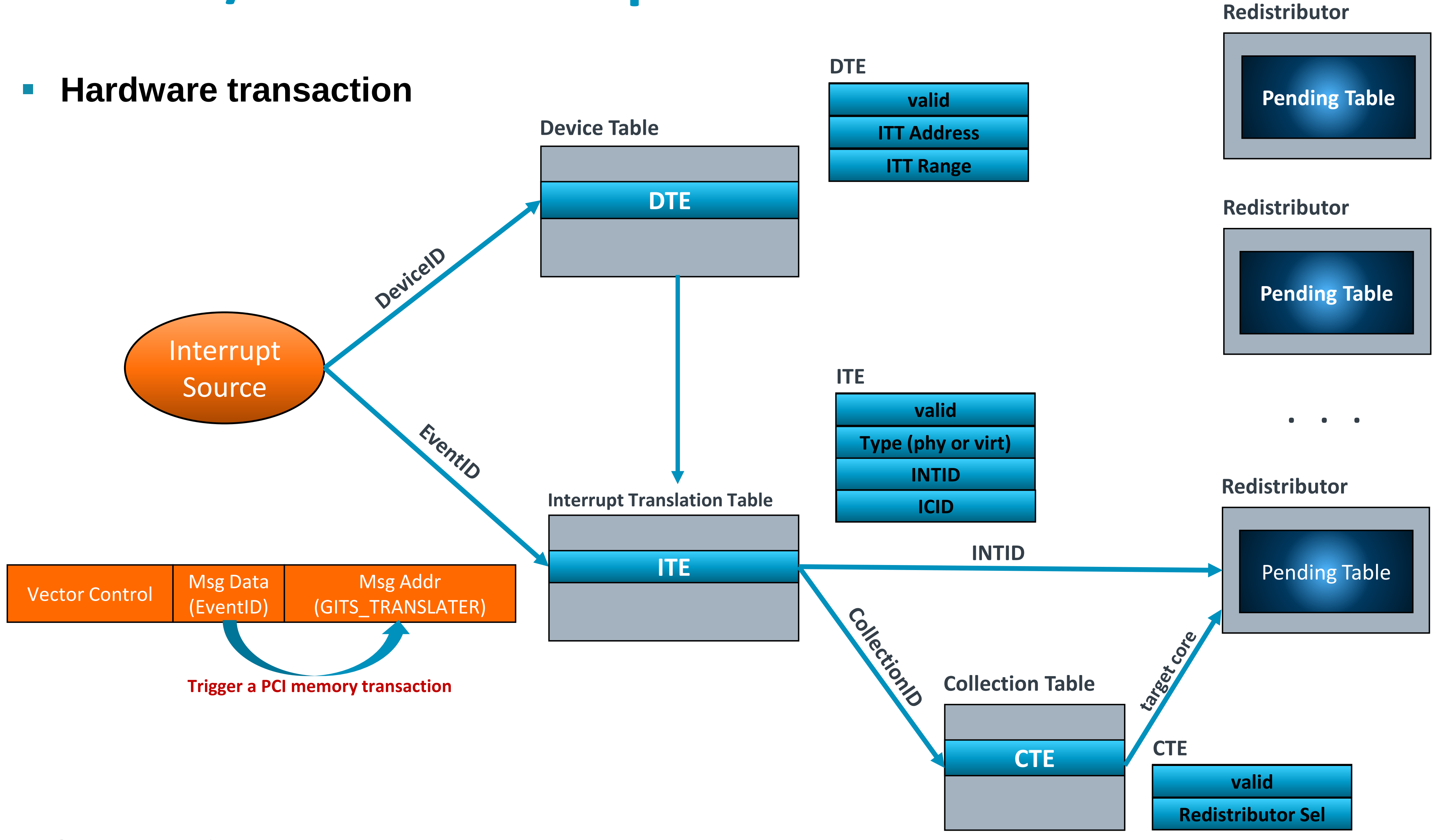
An identifier corresponding to the interrupt to be translated.

pci_enable_msix_range() → msi_domain_activate() → irq_chip_compose_msi_msg() → its_irq_compose_msi_msg()

```
static void its_irq_compose_msi_msg(struct irq_data *d, struct msi_msg *msg)
{
        struct its_device *its_dev = irq_data_get_irq_chip_data(d);
        struct its_node *its;
        u64 addr;

        its = its_dev->its;
        addr = its->phys_base + GITS_TRANSLATER;

        msg->address_lo         = lower_32_bits(addr);
        msg->address_hi         = upper_32_bits(addr);
        msg->data       = its_get_event_id(d);

        iommu_dma_map_msi_msg(d->irq, msg);
}
```

GIC MSI/MSI-X interrupt flow

- **Hardware transaction**



Thank You

arm