**ARM GIC（一） cortex-A 处理器中断简介**

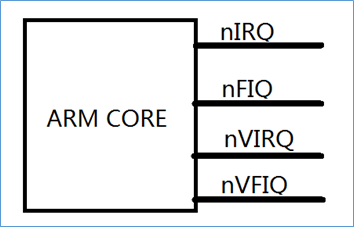
发表于[2018年7月11日](http://www.lujun.org.cn/?p=3811)由[卢 骏](http://www.lujun.org.cn/?author=3)

对于ARM的处理器，中断给处理器提供了触觉，使处理器能够感知到外界的变化，从而实时的处理。本系列博文，是以ARM cortex-A系列处理器，来介绍ARM的soc中，中断的处理。

ARM cortex-A系列处理器，提供了4个管脚给soc，实现外界中断的传递。分别是：

* nIRQ ：物理普通中断
* nFIQ : 物理快速中断
* nVIRQ: 虚拟普通中断
* nVFIQ: 虚拟快速中断

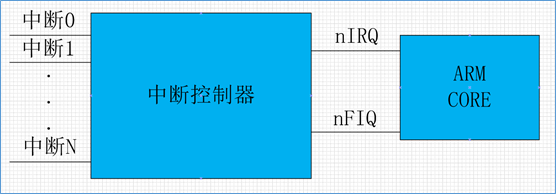
如下图所示：



其中虚拟中断，是为了实现虚拟化而加入的，在这个系列中，不讨论虚拟中断，只介绍物理中断的相关知识。

在arm的soc系统中，会有多个外设，均有可能会产生中断发送给arm cpu，等待cpu处理。

而arm cpu对中断，只提供了2根信号，一个nIRQ，一个是nFIQ。因此就需要有一个中断控制器来作为中间的桥接，收集soc的所有中断信号，然后仲裁选择合适的中断，再发送给CPU，等待CPU处理。如下图所示：



这中间的桥接器件，就是arm公司推出大名鼎鼎的gic，general interrupt controller。

gic其实是一个架构，版本历经了gicv1（已弃用），gicv2，gicv3，gicv4。对于不同的gic版本，arm公司设计了对应的gic IP。

* gic400，支持gicv2架构版本。
* gic500，支持gicv3架构版本。
* gic600，支持gicv3架构版本。

gic的核心功能，就是对soc中外设的中断源的管理，并且提供给软件，配置以及控制这些中断源。

当对应的中断源有效时，gic根据该中断源的配置，决定是否将该中断信号，发送给CPU。如果有多个中断源有效，那么gic还会进行仲裁，选择最高优先级中断，发送给CPU。

当CPU接受到gic发送的中断，通过读取gic的寄存器，就可以知道，中断的来源来自于哪里，从而可以做相应的处理。

当CPU处理完中断之后，会告诉gic，其实就是访问gic的寄存器，该中断处理完毕。gic接受到该信息后，就将该中断源取消，避免又重新发送该中断给cpu以及允许中断抢占。

之后，会先介绍下gicv2的相关知识，然后介绍目前主流使用的gicv3。

# ARM GIC（二）中断术语

发表于[2018年7月12日](http://www.lujun.org.cn/?p=3834)由[卢 骏](http://www.lujun.org.cn/?author=3)

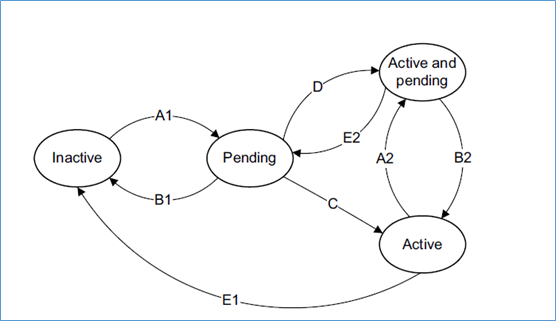
ARM在GIC中，对于中断，定义了如下的一些术语。

# 一、中断状态

对于每一个中断而言，有以下4个状态：

* inactive：中断处于无效状态
* pending：中断处于有效状态，但是cpu没有响应该中断
* active：cpu在响应该中断
* active and pending：cpu在响应该中断，但是该中断源又发送中断过来

以下是中断状态的转移图。至于图中的转移条件，在gic架构文档中，有介绍。



# 二、中断触发方式

    中断触发方式，包含以下两种方式：

* edge-triggered: 边沿触发，当中断源产生一个边沿，中断有效
* level-sensitive：电平触发，当中断源为指定电平，中断有效

# 三、中断类型

    中断类型分为以下几类：

* PPI：（private peripheral interrupt），私有外设中断，该中断来源于外设，但是该中断只对指定的core有效。
* SPI：（shared peripheral interrupt），共享外设中断，该中断来源于外设，但是该中断可以对所有的core有效。
* SGI：（software-generated interrupt），软中断，软件产生的中断，用于给其他的core发送中断信号
* virtual interrupt：虚拟中断，用于支持虚拟机

# 四、中断优先级

因为soc中，中断有很多，为了方便对中断的管理，对每个中断，附加了中断优先级。在中断仲裁时，高优先级的中断，会优于低优先级的中断，发送给cpu处理。

当cpu在响应低优先级中断时，如果此时来了高优先级中断，那么高优先级中断会抢占低优先级中断，而被处理器响应。

# 五、中断号

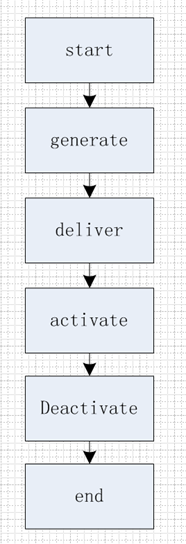
为了方便对中断的管理，gic为每个中断，分配了一个中断号，也就是interrupt ID。对于中断号，gic也进行了分配：

* ID0-ID15，分配给SGI
* ID16-ID31，分配给PPI
* ID32-ID1019分配给SPI
* 其他

在具体的arm的cpu中，对于PPI，又进行了详细的分配。这个，就得看arm cpu的TRM了。

# 六、中断生命周期

一个中断，是有生命周期的。以下是流程图：



* start：中断开始
* generate：中断源产生中断，发送给gic
* deliver：gic将中断发送给cpu
* activate：cpu响应该中断
* deactivate： cpu响应完中断，告诉gic，中断处理完毕，gic更新该中断状态
* end：中断结束

# 七、banking

banking（不知翻译成啥比较合适）功能，包括以下两个：

## 1、中断banking

    对于PPI和SGI，gic可以有多个中断对应于同一个中断号。比如在soc中，有多个外设的中断，共享同一个中断号。

## 2、寄存器banking

对于同一个gic寄存器地址，在不同的情况下，访问的是不同的寄存器。例如在secure和non-secure状态下，访问同一个gic寄存器，其实是访问的不同的gic的寄存器。

具体，更多的信息，得看gic spec以及arm spec。

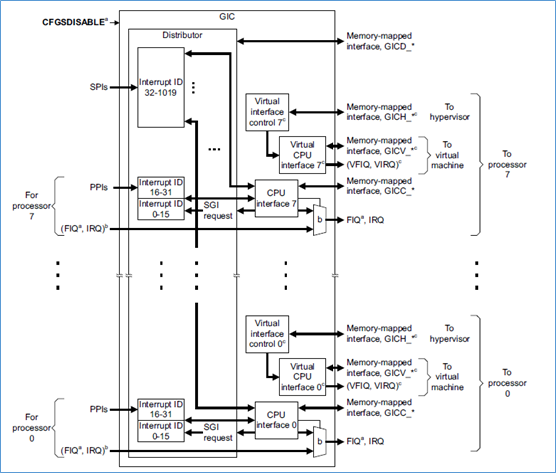
# ARM GIC（三） gicv2架构

发表于[2018年7月18日](http://www.lujun.org.cn/?p=3861)由[卢 骏](http://www.lujun.org.cn/?author=3)

ARM的cpu，特别是cortex-A系列的CPU，目前都是多core的cpu，因此对于多core的cpu的中断管理，就不能像单core那样简单去管理，由此arm定义了GICv2架构，来支持多核cpu的中断管理。

# 一、gicv2架构

GICv2，支持最大8个core。其框图如下图所示：



在gicv2中，gic由两个大模块组成：

* distributor：实现中断分发，对于PPI,SGI是各个core独有的中断，不参与目的core的仲裁，SPI，是所有core共享的，根据配置决定中断发往的core。最后选择最高优先级中断发送给cpu interface。寄存器使用 GICD\_ 作为前缀。一个gic中，只有一个GICD。
* cpu interface：将GICD发送的中断信息，通过IRQ,FIQ管脚，传输给core。寄存器使用 GICC\_ 作为前缀。每一个core，有一个cpu interface。
* virtual cpu interface：将GICD发送的虚拟中断信息，通过VIRQ，VFIQ管脚，传输给core。每一个core，有一个virtual cpu interface。而在这virtual cpu interface中，又包含以下两个组件：
  + virtual interface control：寄存器使用 GICH\_ 作为前缀
  + virtual cpu interface：寄存器使用 GICV\_ 作为前缀

图中的virtual interface，是用于支持虚拟中断，本系列不讨论虚拟中断。

GICv2支持中断旁路模式，也就是gic外部的FIQ,IRQ直接接到core的FIQ，IRQ上，相当于gic是不使能的。也就是CFGSDISABLE是有效的，将GIC给无效掉。

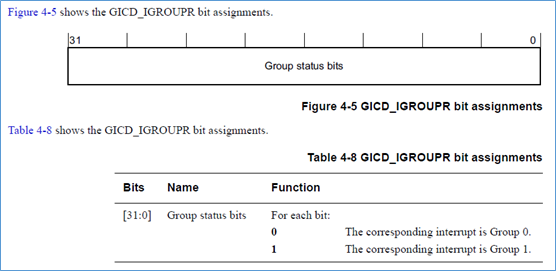
gicv2，定义了自己的一些寄存器，这些寄存器，都是使用memory-mapped的方式去访问的，也就是在soc中，会留有一片空间，给gic。cpu通过访问这部分空间，来对gic进行操作。

寄存器，分为以下：

* GICD\_\*: distributor的寄存器
* GICH\_\*: 虚拟interface的控制寄存器
* GICV\_\*：虚拟interface的控制寄存器
* GICC\_\*: 虚拟cpu interface的寄存器

# 二、中断分组

givc2，将中断，分成了group0和group1。使用寄存器GICD\_IGROUPRn来对每个中断，设置组。



* group0：安全中断，由nFIQ驱动
* group1：非安全中断，由nIRQ驱动

# 三、中断号

gicv2，支持最大1020个中断。其中断号分配如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 中断号 | 分配 | 中断来源 | 寄存器 |
| ID0-ID7 | 非安全软中断 | 软件 | GICD\_SGIR |
| ID8-ID15 | 安全软中断 | 软件 | GICD\_SGIR |
| ID16-ID31 | 私有中断 | 外设 | no |
| ID32-ID1019 | 共享中断 | 外设 | no |

# 四、gic结构

gic主要包括以下两个组件：

* distributor
* cpu interface

## 1、distributor

中断分发器，用来收集所有的中断来源，并且为每个中断源设置中断优先级，中断分组，中断目的core。当有中断产生时，将当前最高优先级中断，发送给对应的cpu interface。

distributor对中断提供以下的功能：

* 全局中断使能
* 每个中断的使能
* 中断的优先级
* 中断的分组
* 中断的目的core
* 中断触发方式
* 对于SGI中断，传输中断到指定的core
* 每个中断的状态管理
* 提供软件，可以修改中断的pending状态

## 2、cpu interface

cpu interface，将GICD发送的中断信息，通过IRQ，FIQ管脚，发送给连接到该cpu接口的core。

cpu interface提供了一下的功能：

* 将中断请求发送给cpu
* 对中断进行认可（acknowledging an interrupt）
* 中断完成识别(indicating completion of an interrupt)
* 设置中断优先级屏蔽
* 定义中断抢占策略
* 决定当前处于pending状态最高优先级中断

# 五、中断认可

中断认可，是指cpu响应该中断。此时中断状态从pending状态，变为active状态。通过访问GICC\_IAR寄存器，来对中断进行认可。

* GICC\_IAR： 认可group0的中断
* GICC\_AIAR: 认可group1的中断

# 六、中断完成

中断完成，是指cpu处理完中断。此时中断状态从active状态，变为inactive状态。gic中，对中断完成，定义了以下两个stage：

* **优先级重置（priority drop）**：将当前中断屏蔽的最高优先级进行重置，以便能够响应低优先级中断。group0中断，通过写GICC\_EOIR寄存器，来实现优先级重置，group1中断，通过写 GICC\_AEOIR 寄存器，来实现优先级重置。
* **中断无效（interrupt deactivation）**：将中断的状态，设置为inactive状态。通过写 GICC\_DIR 寄存器，来实现中断无效。

这里为什么要对中断完成，定义2个stage，其实是有考虑的。对于中断来说，我们是希望中断处理程序越短越好，但是有些中断处理程序，就是比较长，在这种情况下，就会使其他中断得到响应，从而影响实时性。

比如当前cpu在响应优先级为4的中断A，但是这个中断A的中断处理程序比较长，此时如果有优先级为5的中断B到来，那么cpu是不会响应这个中断的。

在软件上，会将中断处理程序分为两部分，分为上半部分，和下半部分。在上半部分，完成中断最紧急的任务，然后就可以通知GIC，降低当前的中断处理优先级，以便其他中断能够得到响应。在下半部分，处理该中断的其他事情。

在这种机制下，低优先级的中断，不用等待高优先级的中断，完全执行完中断处理程序后，就可以被cpu所响应，提高实时性。

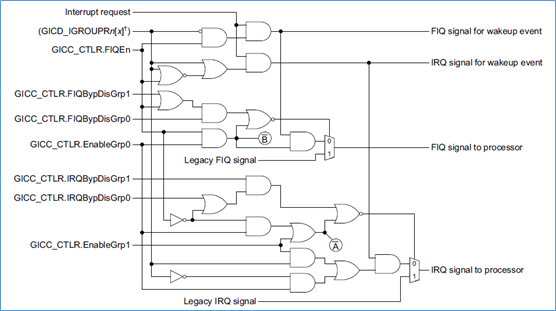
为了实现上述机制，就将中断完成分成了2步。还是刚刚的例子，cpu在响应优先级为4的中断A，当中断A的上半部分完成后，通知GIC，优先级重置（drop priority），GIC将当前的最高优先级中断重置，重置到响应中断A之前的优先级，比如优先级6，那么此时优先级为5的中断B，就可以被cpu响应。最后中断A的下半部分完成后，通知GIC，将该中断A的状态，设置为inactive状态，此时中断A就真正的完成了。

当然，也可以不将中断完成分成2步，就1步。通过控制 GICC\_CTLR寄存器的EOImode比特，来决定是否将中断完成分成2步。

# 七、bypass功能

gicv2支持bypass功能，这样gic就不起作用了，core的中断管脚，直接由soc的其他部分信号驱动。

如下图所示，通过控制 GICC\_CLTR 寄存器的一些比特位，来实现bypass功能。不过这个功能一般不使用，不然何必要在arm的soc中，加入gic呢？



# 八、中断处理流程

中断处理流程，包含了以下几步：

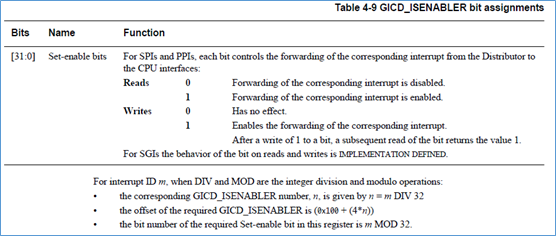
* GIC决定每个中断的使能状态，不使能的中断，是不能发送中断的
* 如果某个中断的中断源有效，GIC将该中断的状态设置为pending状态，然后判断该中断的目标core
* 对于每一个core，GIC将当前处于pending状态的优先级最高的中断，发送给该core的cpu interface
* cpu interface接收GIC发送的中断请求，判断优先级是否满足要求，如果满足，就将中断通过nFIQ或nIRQ管脚，发送给core。
* core响应该中断，通过读取 GICC\_IAR 寄存器，来认可该中断。读取该寄存器，如果是软中断，返回源处理器ID，否则返回中断号。
* 当core认可该中断后，GIC将该中断的状态，修改为active状态
* 当core完成该中断后，通过写 EOIR （end of interrupt register）来实现优先级重置，写 GICC\_DIR 寄存器，来无效该中断

# 九、中断使能和禁止

通过设置GICD\_ISENABLERn寄存器，来使中断使能，通过设置GICD\_ICENABLERn寄存器，来使中断禁止。

这两个寄存器，都是bit有效的寄存器，也就是一个bit，关联一个中断。

比如对于GICD\_ISENABLER寄存器，描述如下：



# 十、中断pending

通过设置GICD\_ISPENDRn或GICD\_ICPENDRn寄存器，可以读取和修改中断的pending状态。这两个寄存器，也是bit有效的寄存器，一个bit，关联一个中断。

# 十一、中断active

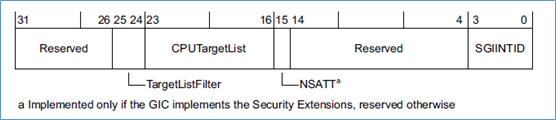
通过设置GICD\_ISACTIVERn或GICD\_ICACTIVERn寄存器，可以读取和修改中断的active状态。这两个寄存器，也是bit有效的寄存器，一个bit，关联一个中断。

# 十二、产生软中断

通过写 GICD\_SGIR 寄存器，来产生软中断。软中断，可以指定产生中断，发往执行的core，也可以发送多个core。

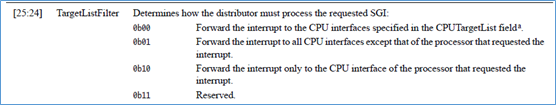
对于软中断，这个是软件产生的中断。比如软件，想给执行自己的core，发送一个中断，就可以通过软中断来产生。或者软件，想起其他的core，发送一个中断，也可以通过软中断来产生。

寄存器如下：



## 1、TargetListFileter

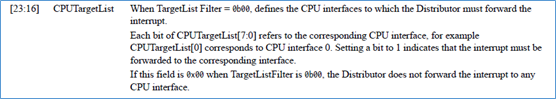
对于TargetListFileter：决定distributor将软中断，如何发送给cpu interface。



* 0b00：按照CPUTargetList的指定，来发送软中断
* 0b01：按照CPUTargetList的指定，来发送软中断，但是不能发送给自己
* 0b10：软中断，只能发送给自己

## 2、CPUTargetList

描述如下：



对于多core的系统，会给每个core一个编号。gicv2支持最多8个core，因此core的编号就是0-7，刚好8个bit，可以表示。

这样的CPUTargetList，就和8个cpu相对应。第0bit，表示core0，第7bit，表示core7。

如果想给core1，core2，core7发送软中断，那么此时这个位域要填入0x84。

## 3、NSATT

描述如下：



这个用来支持安全扩展，在gicv2中，将中断进行了分组

* group0：安全中断
* group1：非安全中断

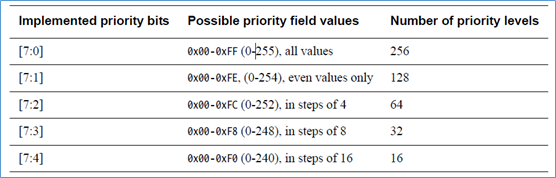
这个bit，用来表示发送的软中断，是安全中断，还是非安全中断。而且这个bit，只有core处于安全状态的时候，才能写。如果core是处于非安全状态，那么这个bit被忽略，也就是只能发非安全的软中断。

## 4、SGIINTID

发送的软中断的中断号。

# 十三、中断优先级

gicv2，支持最小16个，最大256个中断优先级，如下图所示：



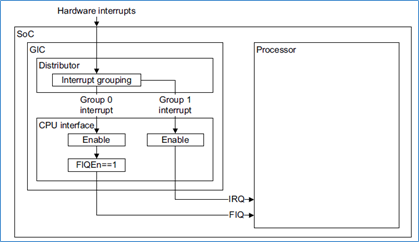
如果实现的中断优先级小于256个，那么最低的几个bit，是为0的。

通过设置GICD\_IPRIORITYRn寄存器，来设置中断的优先级。这个寄存器是字节有效的，也就是一个字节，对应一个中断的优先级。优先级数值越小，那么这个中断的优先级越高。

高优先级的中断，是可以抢占低优先级的中断。

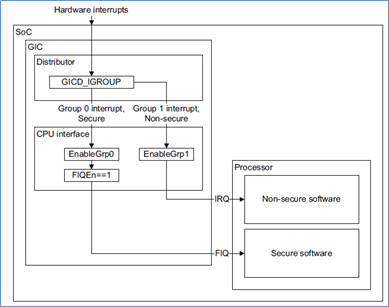
# 十四、gic使用例子

下图是gic的使用例子：



外部的中断，连接到gic。由distributor进行中断分组。中断请求，由distributor发送给cpu interface，cpu interface再发送给处理器。

对于支持安全扩展，其应用如下：



安全中断，处于group0，非安全中断处于group1。

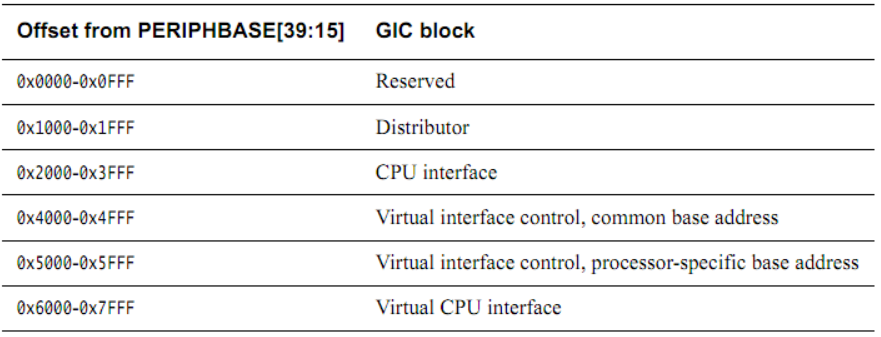
# 十五、gic寄存器

gic寄存器，分为两部分，一部分是distributor的寄存器，另一部分是cpu interface的寄存器。

两部分的寄存器，均是通过memory-mapped的方式来访问。

【

GIC的相关寄存器是存储器映射的，它的基地址是由PERIPHBASE[39:15]所指定的，PERIPHBASE的值可以通过读取CP15协处理器的CBAR寄存器可以得到，GIC的内存地址映射图如下所示：

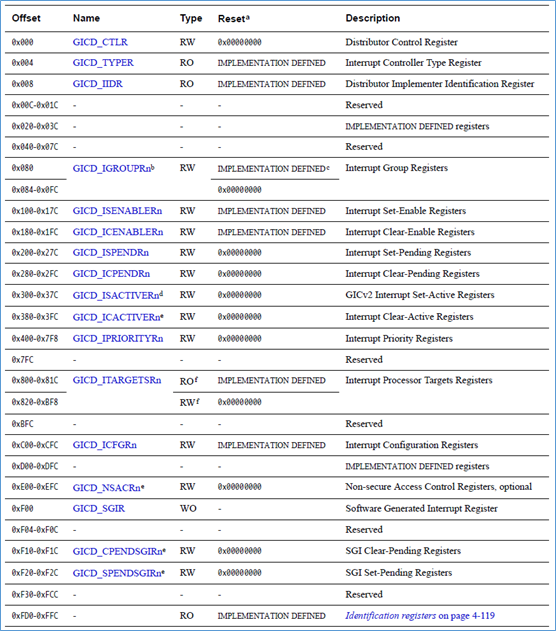


从上图可以看到，相对于GIC的基地址偏移0x0000~0x0FFF为系统保留，地址偏移0x1000~0x1FFF为GIC的Distributor相关寄存器地址，地址偏移0x2000~0x3FFF为GIC的CPU Interface相关寄存器地址

<http://t.zoukankan.com/Cqlismy-p-12549087.html>

】

下图是distributor的寄存器：



下图是cpu interface的寄存器：



# 十六、总结

以上就是GICv2的介绍，更多的内容，需要查看gicv2的文档。

gicv2比较简单，最多只能支持8个core，超过了8个core，那么就不能使用gicv2了。不过这也不是大问题，对于手机的arm处理器来说，最多也就8个core。但是对于服务器，桌面级的arm处理器，那么就可能会超过8个core，此时gicv2就不适用了，所以ARM后面又加入GICv3，v4架构。

GICv2的寄存器，都是通过memory-mapped的方式访问。但是中断在一个soc系统中，是经常会产生的，那么处理器就会经常的读取gic的寄存器，而使用memory-mapped的方式去访问，就会影响中断响应速度。在之后的GICv3，v3中，就加入了使用系统寄存器来进行访问，加快中断处理。

GICv2只是一个gic的架构，其实现的对应的IP是gic400。

# ARM GIC（四） gicv3架构基础

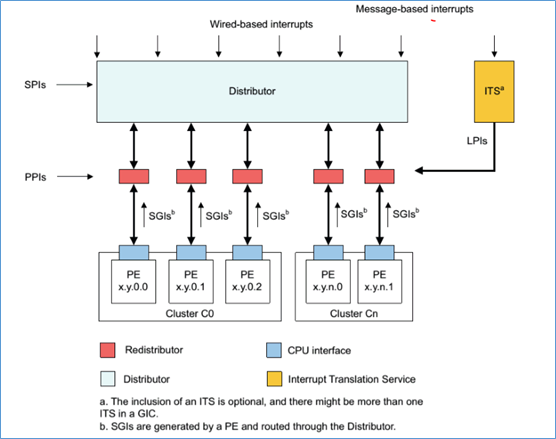
发表于[2018年7月22日](http://www.lujun.org.cn/?p=3874)由[卢 骏](http://www.lujun.org.cn/?author=3)

GICv3架构是GICv2架构的升级版，增加了很多东西。变化在于以下：

* 使用**属性层次**（affinity hierarchies），来对core进行标识，使gic支持更多的core
* 将cpu interface独立出来，用户可以将其设计在core内部
* 增加redistributor组件，用来连接distributor和cpu interface
* 增加了LPI，使用ITS来解析
* 对于cpu interface的寄存器，增加系统寄存器访问方式

# 一、gicv3结构

下图是gicv3的架构。



包含了以下的组件：

* distributor：SPI中断的管理，将中断发送给redistributor
* redistributor：PPI，SGI，LPI中断的管理，将中断发送给cpu interface
* cpu interface：传输中断给core
* ITS：用来解析LPI中断

其中，cpu interface是实现在core内部的，distributor，redistributor，ITS是实现在gic内部的。

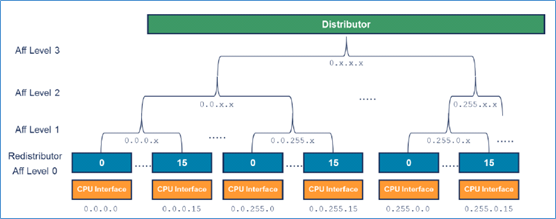
cpu interface和gic的redistributor通信，通过AXI-Stream协议，来实现通信。

# 二、属性层次

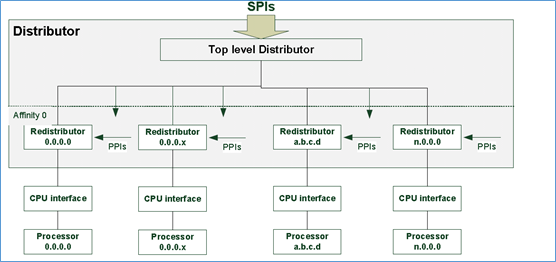
gicv3的一大变化，是对core的标识。对core不再使用单一数字来表示，而是使用属性层次来标识，和arm core，使用MPIDR\_EL1系统寄存器来标识core一致。

每个core，根据属性层次的不同，使用不同的标号来识别。如下图所示，是一个4层结构，那么对于一个core来说，就可以用 xxx.xxx.xxx.xxx 来识别。

这种标识方式，和ARMv8架构的使用MPIDR\_EL1寄存器，来标识core是一样的。



每个core，连接一个cpu interface，而cpu interface会连接gic中的一个redistributor。redistributor的标识和core的标识一样。

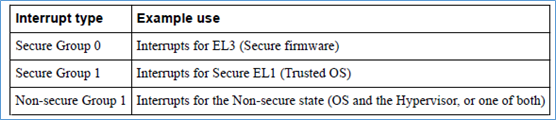


# 三、中断分组

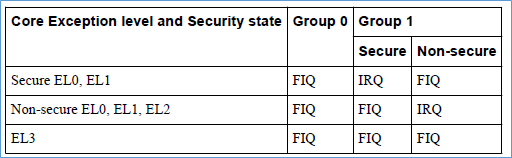
gicv3，将中断分成了2个大组，group0和group1。

* group0：提供给EL3使用
* group1：又分为2组，分别给安全中断和非安全中断使用

如下图所示:

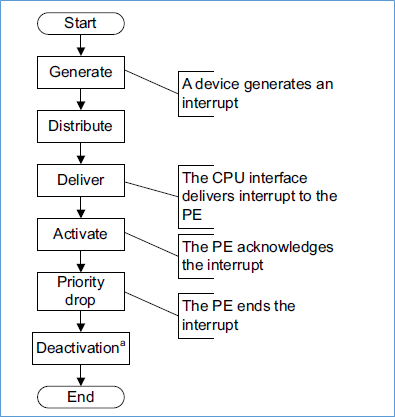


以下是IRQ,FIQ与组的对应关系。



# 四、中断生命周期

    中断生命周期，如下图所示：



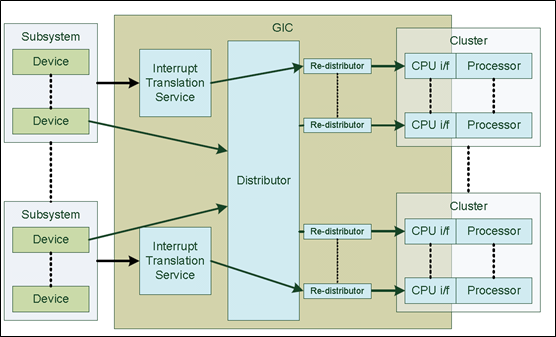
* generate：外设发起一个中断
* distribute：distributor对收到的中断源进行仲裁，然后发送给对应的cpu interface
* deliver：cpu interface将中断发送给core
* activate：core通过读取 GICC\_IAR 寄存器，来对中断进行认可
* priority drop: core通过写 GICC\_EOIR 寄存器，来实现优先级重置
* deactivation：core通过写 GICC\_DIR 寄存器，来无效该中断

这个中断生命周期，和gicv2的中断生命周期是一样的。

# 五、中断流程

下图是gic的中断流程，中断分成2类：

* 一类是中断要通过distributor，比如SPI中断
* 一类是中断不通过distributor，比如LPI中断



中断要通过distributor的中断流程

* 外设发起中断，发送给distributor
* distributor将该中断，分发给合适的re-distributor
* re-distributor将中断信息，发送给cpu interface。
* cpu interface产生合适的中断异常给处理器
* 处理器接收该异常，并且软件处理该中断

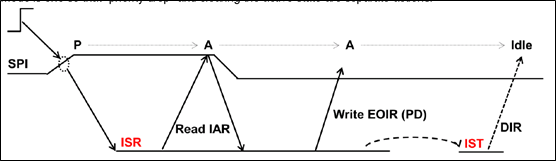
LPI中断的中断流程

* 外设发起中断，发送给ITS
* ITS分析中断，决定将来发送的re-distributor
* ITS将中断发送给合适的re-distributor
* re-distributor将中断信息，发送给cpu interface。
* cpu interface产生合适的中断异常给处理器
* 处理器接收该异常，并且软件处理该中断

# 六、中断处理

中断处理，分为边沿触发处理和电平触发处理

## 1、边沿触发处理

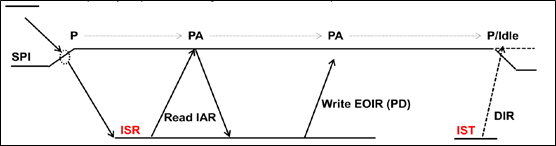


外部边沿中断到达，中断状态被置为pending状态。

软件读取IAR寄存器值，表示PE认可该中断，中断状态被置为active状态

软件中断处理完毕后，写EOIR寄存器，表示优先级重置。过一段时间后，写DIR寄存器，中断状态被置为idle状态。

## 2、电平触发处理



外部高电平中断到达，中断状态置为pending状态。

软件读取IAR寄存器，表示PE认可该中断。但中断依然为高，中断状态进入pending and active状态。

软件中断处理完毕后，写EOIR寄存器，表示优先级重置。过一段时间后，写DIR寄存器，中断状态被置为idle状态。

# 七、寄存器

gicv3中，多了很多寄存器。而且对寄存器，提供了2种访问方式，一种是memory-mapped的访问，一种是系统寄存器访问：

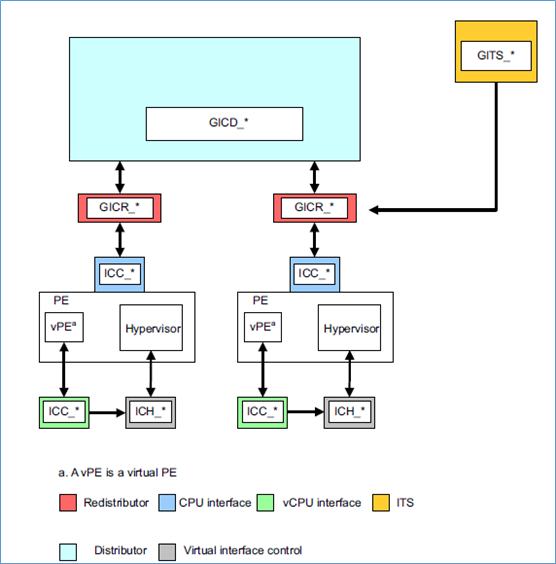
memory-mapped访问的寄存器：

* GICC: cpu interface寄存器
* GICD: distributor寄存器
* GICH: virtual interface控制寄存器，在hypervisor模式访问
* GICR: redistributor寄存器
* GICV: virtual cpu interface寄存器
* GITS: ITS寄存器

系统寄存器访问的寄存器：

* ICC: 物理 cpu interface 系统寄存器
* ICV: 虚拟 cpu interface 系统寄存器
* ICH： 虚拟 cpu interface 控制系统寄存器

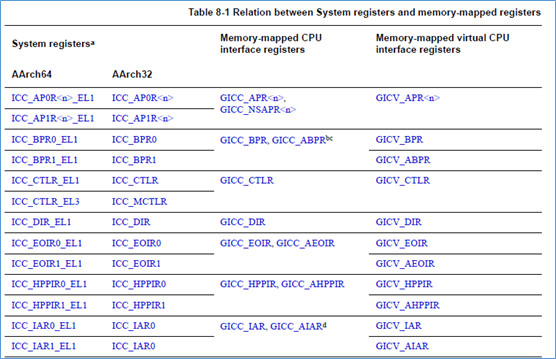
下图是gicv3中，各个寄存器，所在的位置。



对于系统寄存器访问方式的gic寄存器，是实现在core内部的。而memory-mapped访问方式的gic寄存器，是在gic内部的。

gicv3架构中，没有强制，系统寄存器访问方式的寄存器，是不能通过memory-mapped方式访问的。也就是ICC, ICV, ICH寄存器，也是可以实现在gic内部，通过memory-mapped方式去访问。但是一般的实现中，是没有这样的实现的。

下图是ICC的系统寄存器，和memory-mepped方式寄存器的对应关系的一部分，更多的就要查看gicv3的spec。



那么，问题来了，gicv3中，为什么选择将cpu interface，从gic中抽离，实现在core内部？为什么要将cpu interface的寄存器，增加系统寄存器访问方式，实现在core的内部？这样做，是有什么好处？

我认为，gicv3的上述安排，第一是为了软件编写能够简单，通用，第二是为了让中断响应能够更快。

首先要先了解，在gic的寄存器中，哪一些寄存器，是会频繁被core所访问的，哪一些寄存器，是不会频繁被core所访问的。毫无疑问，cpu interface的寄存器，是会频繁被core所访问的，因为core需要访问cpu interface的寄存器，来认可中断，来中断完成，来无效中断。而其他的寄存器，是配置中断的，只有在core需要去配置中断的时候，才用访问得到。有了这个认识，那么理解之后我所讲述的，就比较容易了。

在gicv2中，cpu interface的寄存器，是实现在gic内部的，因此当core收到一个中断时，会通过axi总线（假设memory总线是axi总线），去访问cpu interface的寄存器。而中断在一个soc系统中，是会频繁的产生的，这就意味着，core会频繁的去访问gic的寄存器，这样会占用axi总线的带宽，总而会影响中断的实时响应。而且core通过axi总线去访问cpu interface寄存器，延迟，也比较大。

在gicv3中，将cpu interface从gic中抽离出来，实现在core内部，而不实现在gic中。core对cpu interface的访问，通过系统寄存器方式访问，也就是使用msr，mrs访问，那么core对cpu interface的寄存器访问，就加速了，而且还不占用axi总线带宽。这样core对中断的处理，就加速了。

cpu interface与gic之间，是通过专用的AXI-stream总线，来传输信息的，这样也不会占用AXI总线的带宽。

gicv3架构的内容，比较多，我这边会拆成几个部分。

下一部分，会介绍gic stream协议。也就是cpu interface与gic之间的通信。

# ARM GIC（五）gicv3架构-gic stream协议

发表于[2018年7月22日](http://www.lujun.org.cn/?p=3896)由[卢 骏](http://www.lujun.org.cn/?author=3)

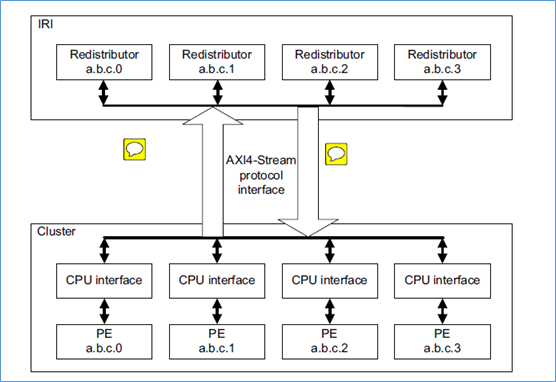
gic stream协议，是基于AXI-stream协议。用于gic的IRI组件（interrupt routing infrastructure），和cpu interface之间，传输信息。

distributor，redistributor和ITS，统称为IRI组件。

gic stream协议，包含以下2个接口：

* 下行AXI-stream接口：用于IRI向cpu interface传递信息，连接
* 上行AXI-stream接口：用于cpu interface向IRI传递信息

如下图所示：

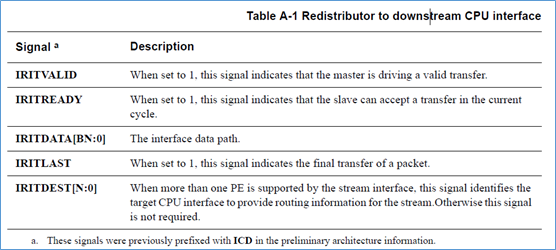


# 一、接口信号

    接口信号，包含下行接口信号，和上行接口信号。无论是上行还是下行，都是基于AXI-stream协议，通过data信号，来传输数据，data信号的位宽，也是固定的，为16bit。

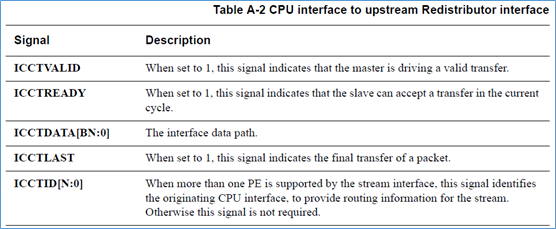
## 1、下行接口信号

下行接口信号如下表所示，接口协议是基于AXI-stream协议。



## 2、上行接口信号

上行接口信号如下表所示，接口协议是基于AXI-stream协议。



# 二、包

IRI与cpu interface通过gic stream协议传输信息，传输的信息，是以包为单位。包，分为两类包：

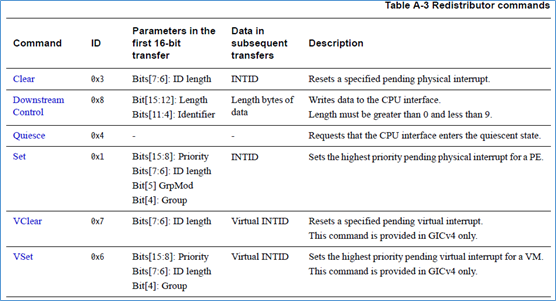
* 命令包，分为redistributor命令包，cpu interface命令包
* 响应包，分为redistributor响应包，cpu interface响应包

AXI-stream协议，每次传输2个字节，多次传输，组成一个包。不同的包，大小不是一样的，比如有的是16个字节，有的是8个字节。包传输的第一个16bit数据，表示包的类型。

如果一个组件，发送命令包，那么另一个，需要回应响应包。

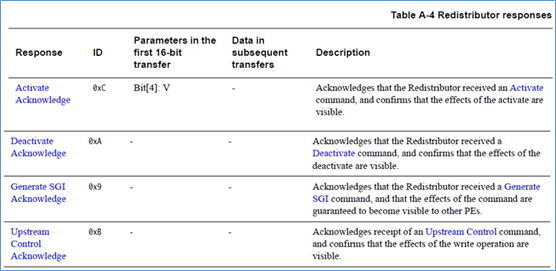
## 1、redistributor命令包

下图是下行redistributor命令包。



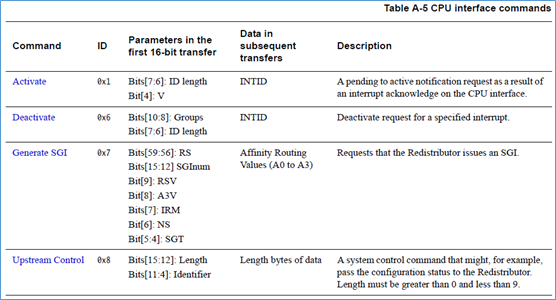
## 2、redistributor响应包

下图是上行redistributor响应包。



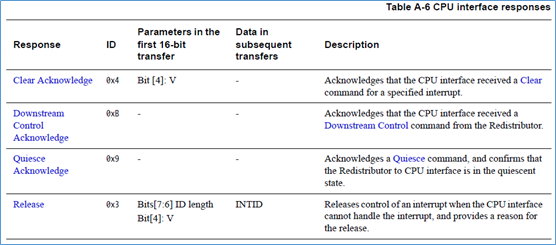
## 3、cpu interface命令包

下图是cpu interface命令包。



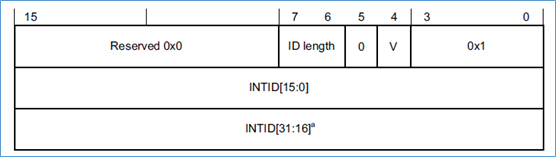
## 4、cpu interface响应包

下图是cpu interface响应包。



## 5、例子

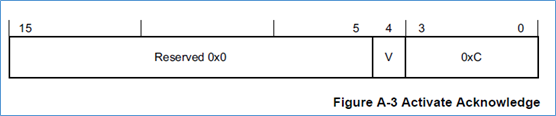
    以下是cpu interface的activate命令包，格式如下：



这个activate命令，由cpu interface发送给IRI，表示认可中断，中断号，由包中的INTID字域来表示。

IRI在收到cpu interface的activate命令后，会回发activate acknowledge响应包。表示，IRI接收到cpu interface的activate命令。

其格式，如下所示：

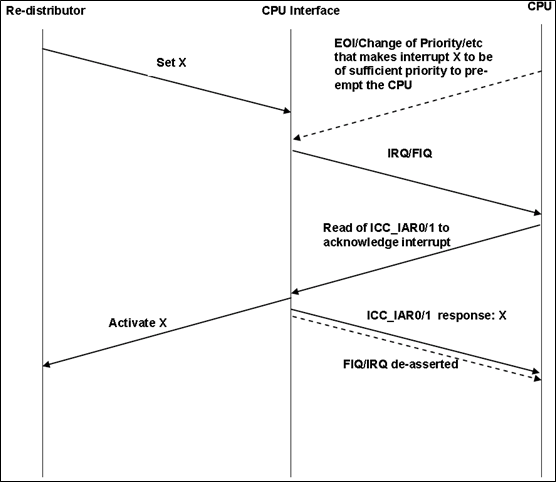


包在传输的过程中，先发第一个数据的低16bit数据，再发第一个数据的高16bit数据，如果还有下一个数据，按照上述流程发送。所以，命令的类型，是最先发送的。

# 三、包传输流程

## 1、中断发送

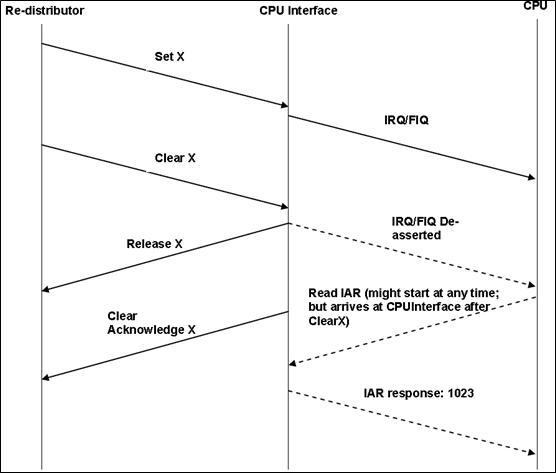
如下图，redistributor，要发送一个中断给cpu。包的传输的流程，如下：



* redistributor给cpu interface发送set命令，发送中断X请求，cpu interface接收到该命令后，如果该中断X符合当前优先级要求，CPU interface通过IRQ/FIQ给cpu发送中断。
* CPU响应CPU interface发送的中断，于是去读取ICC\_IAR寄存器，表示认可该中断，得到中断号。之后cpu interface给redistributor发送activate响应。然后把IRQ/FIQ给取消掉。

## 2、中断取消

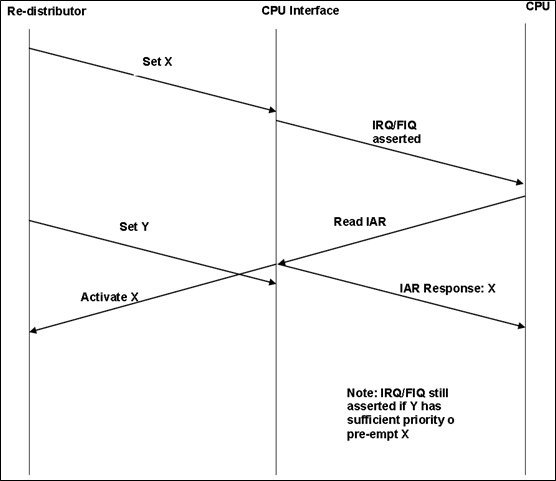
CPU在读取ICC\_IAR寄存器前，redistributor取消中断。包的传输流程，如下：



* redistributor给cpu interface发送set命令，cpu interface接收到该命令后，通过IRQ/FIQ给cpu发送中断。
* redistributor给cpu interface发送clear命令，清除该中断，cpu interface将IRQ/FIQ拉低。然后回release响应。
* cpu interface给redistributor，回clear acknowledge响应。
* 如果此时，cpu读取IAR寄存器，CPU会获取到一个假的中断号。

## 3、两个中断

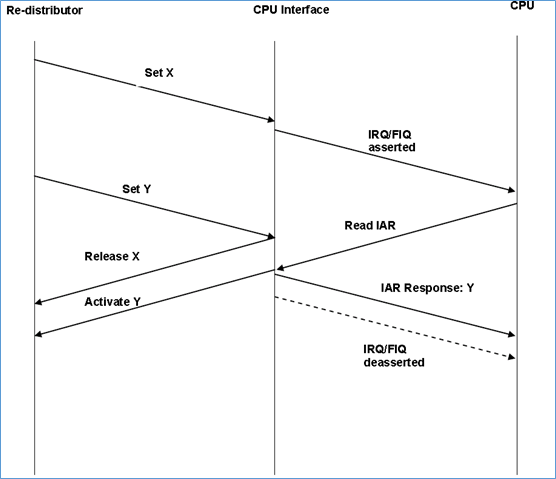
redistributor给cpu interface发送两个中断。包的传输流程，如下图所示：



* redistributor，首先发送set x命令，发送中断x。cpu interface接收该命令，将IRQ/FIQ拉高，向CPU发送中断请求。
* cpu读取ICC\_IAR寄存器，认可该中断x，开始处理该中断x。cpu interface给redistributor回activate x响应。
* 之后，redistributor给CPU interface发送set y命令，发送中断y。在cpu interface中，如果y的优先级符合要求，那么IRQ/FIQ会一直有效。等待CPU处理。

## 4、中断抢占

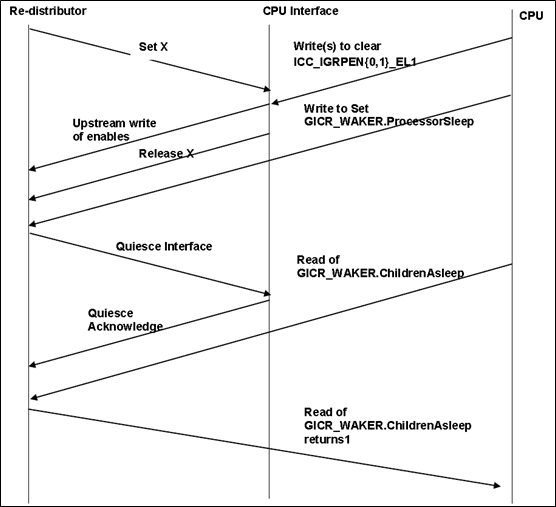
redistributor给cpu interface发送2个中断，第二个中断抢占第一个中断。包的流程如下：



* redistributor，首先发送set x命令，发送中断x。cpu interface接收该命令，将IRQ/FIQ拉高，向CPU发送中断请求。
* 在cpu读取ICC\_IAR寄存器之前，redistributor，又给cpu interface发送了set y命令，发送中断y。并且y的优先级比x高。
* cpu interface给redistributor回release x响应，表示cpu interface暂时不处理中断x，中断x，将来重新发送。
* CPU读取ICC\_IAR寄存器，认可中断y。cpu interface给redistributor回activate y响应。

## 5、电源断电

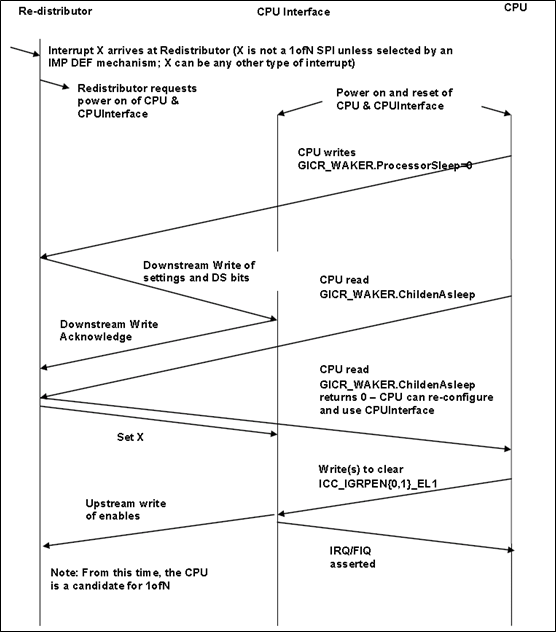
gic给CPU发送中断，使CPU断电。



这个，就比较复杂了。就不解析了。

## 6、电源上电

redistributor，请求将CPU和cpu interface上电。



这个也比较复杂，这里不解析。

# 四、总结

gicv3中，IRI与cpu interface之间，是通过包，来传输信息。传输的接口协议，使用AXI-stream。通过包的各种组合，来实现gic的中断操作与中断管理。

之后，会介绍gicv3中，引入的一种新的中断类型，消息中断。

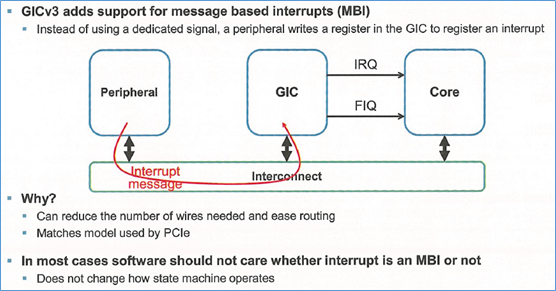
# ARM GIC（六）gicv3架构-LPI

发表于[2018年7月29日](http://www.lujun.org.cn/?p=3921)由[卢 骏](http://www.lujun.org.cn/?author=3)

在gicv3中，引入了一种新的中断类型。message based interrupts，消息中断。

# 一、消息中断

外设，不在通过专用中断线，向gic发送中断，而是写gic的寄存器，来发送中断。



这样的一个好处是，可以减少中断线的个数。

为了支持消息中断，gicv3，增加了LPI，来支持消息中断。并且为他分配了特别多的中断号，从8192开始，移植到16777216。

LPI，locality-specific peripheral interrupts。spec中，用了一章，来介绍这个LPI。

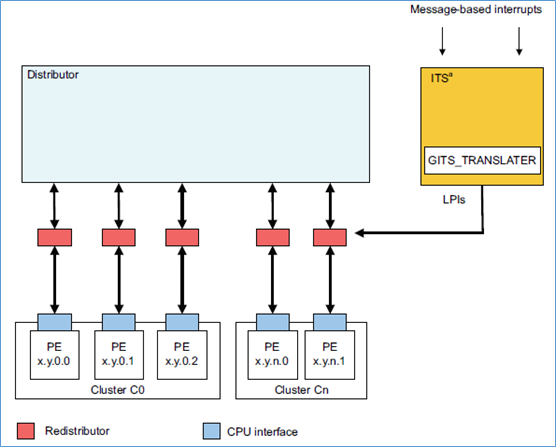
# 二、LPI介绍

LPI是一种基于消息的边沿中断。也就是，中断信息，不在通过中断线，进行传递，而是通过memory。gic内部，提供一个寄存器，当外设往这个地址，写入数据时，就往gic发送了一个中断。

在soc系统中，外设想要发送中断给gic，是需要一根中断线的。如果现在一个外设，需要增加一个中断，那么就要增加一根中断线，然后连接到gic。这样，就需要修改设计。而引入了LPI之后，当外设需要增加中断，只需要使用LPI方式，传输中断即可，不需要修改soc设计。

引入了LPI之后，gicv3中，还加入了ITS组件，interrupt translation service。ITS将接收到的LPI中断，进行解析，然后发送到对应的redistributor，再由redistributor将中断信息，发送给cpu interface。

以下是带LPI的框图。外设，通过写 GITS\_TRANSLATER 寄存器，来传递消息中断。



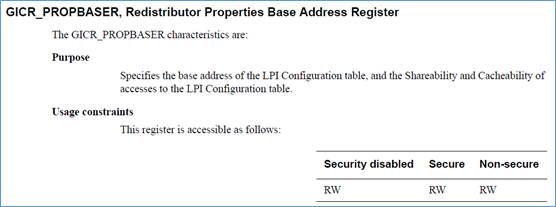
LPI，和SPI,PPI,SGI有些差别，LPI的中断的配置，以及中断的状态，是保存在memory的表中，而不是保存在gic的寄存器中的。

* GICR\_PROPBASER：保存LPI中断配置表的基地址
* GICR\_PENDBASER: 保存LPI中断状态表的基地址

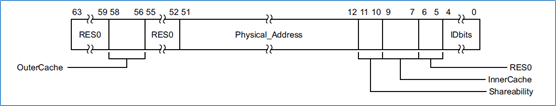
这里，就涉及到两个表：

## 1、LPI中断配置表

该表，保存在memory中。基地址，由GICR\_PROPBASER寄存器决定。

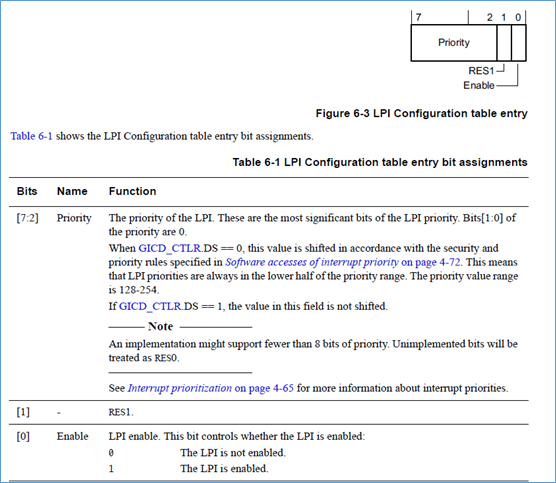


该寄存器描述如下：



其中的Physical\_Address字段，指定了LPI中断配置表的基地址。

对于LPI配置表，每个LPI中断，占用1个字节，指定了该中断的使能和中断优先级。



当外部发送LPI中断给redistributor，redistributor首先要查该表，也就是要访问memory来获取LPI中断的配置。为了加速这过程，redistributor中可以配置cache，用来缓存LPI中断的配置信息。

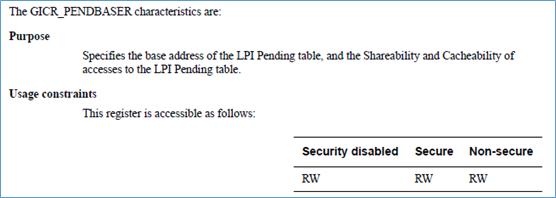
因为有了cache，所以LPI中断的配置信息，就有了2份拷贝，一份在memory中，一份在redistributor的cache中。如果软件修改了memory中的LPI中断的配置信息，需要将redistributor中的cache信息给无效掉。

## 2、LPI中断状态表

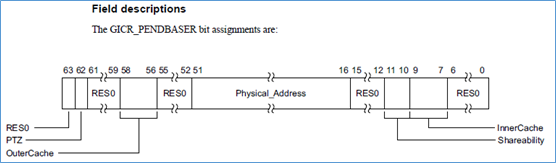
该表，处于memory中，保存了LPI中断的状态，是否pending状态。

LPI中断的状态，不是保存在寄存器中，而是保存在memory中的pending表中。该状态表，由redistributor来进行更改。而该table的基地址，是由软件来设置的。

软件通过设置 GICR\_PENDBASER 寄存器来设置。



该寄存器，设置LPI状态表的基地址，该状态表的memory的属性，如shareability，cache属性等。



每个LPI中断，占用一个bit空间

* 0： 该LPI中断，没有处于pending状态
* 1： 该LPI中断，处于pending状态

该状态表，由redistributor来设置。软件如果修改该表，会引发unpredictable行为。

# 三、LPI的实现方式

为了实现LPI，gicv3定义了以下两种方法来实现：

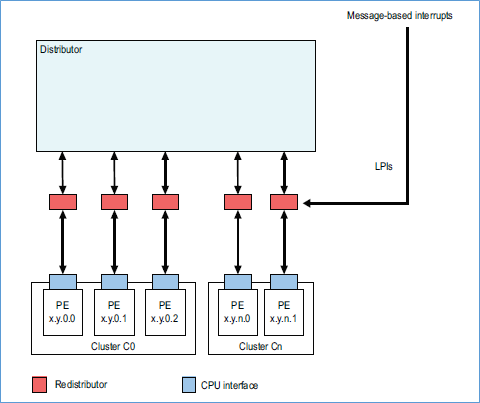
* 使用ITS，将外设发送到eventID，转换成LPI     中断号
* forwarding方式，直接访问redistributor的寄存器GICR\_SERLPIR，直接发送LPI中断

## 1、forwarding方式

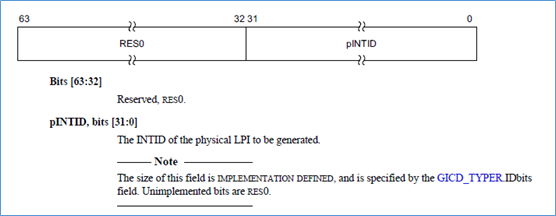
这种方式，比较简单，主要由下面几个寄存器来实现：

* GICR\_SERLPIR
* GICR\_CLRLPIR
* GICR\_INVLPIR
* GICR\_INVALLR
* GICR\_SYNCR

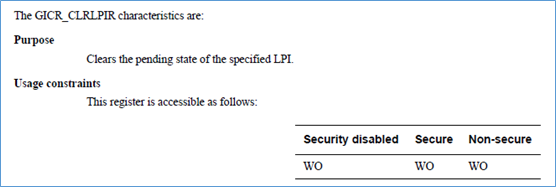
其gic框图如下所示：



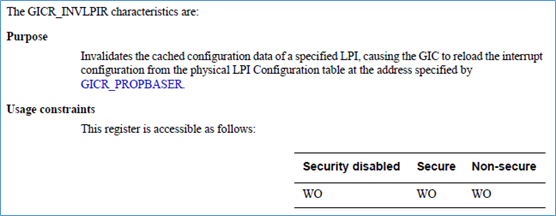
GICR\_SERLPIR，将指定的LPI中断，设置为pending状态。



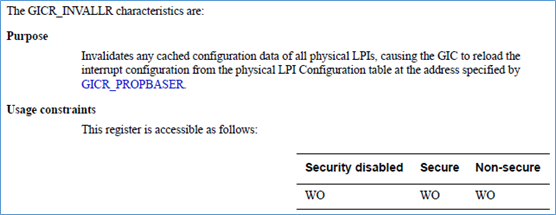
GICR\_INVLPIR，将指定的LPI中断，清除pending状态。寄存器内容和GICR\_SERLPIR一致。



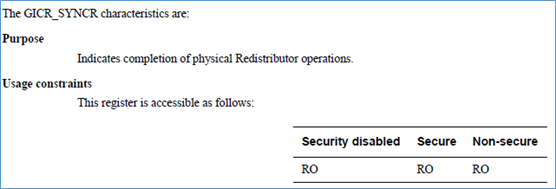
GICR\_INVLPIR，将缓存中，指定LPI的缓存给无效掉，使GIC重新从memory中载入LPI的配置。



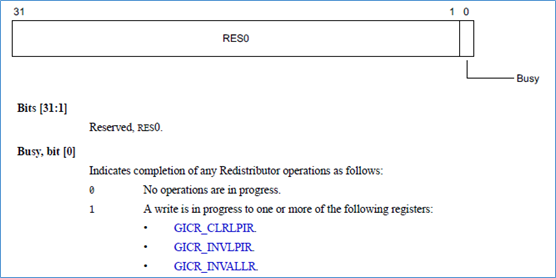
GICR\_INVALLR，将缓存中，所有LPI的缓存给无效掉，使GIC重新从memory中，载入LPI中断的配置。



GICR\_SYNCR，对redistributor的操作是否完成。



寄存器，只有第0bit是有效的。如果为0，表示当前对redistributor的操作是完成的，如果为1，那么是没有完成的。

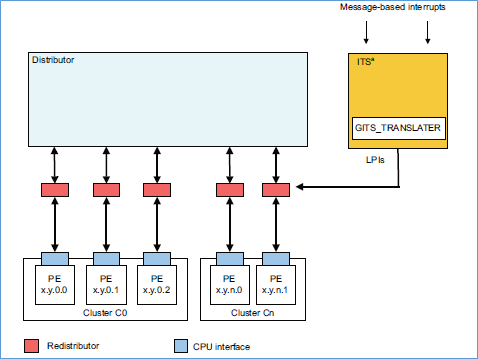


## 2、使用ITS方式

理解了forwarding方式，那么理解ITS方式，就要容易了。forwarding方式，是直接得到了LPI的中断号。

但是对于ITS方式，是不知道LPI的中断号的。需要将外设发送的DeviceID，eventID，通过一系列查表，得到LPI的中断号以及该中断对应的target redistributor，然后将LPI中断，发送给对应的redistributor。

下图是带有ITS的gic框图：



外设，通过写GITS\_TRANSLATER寄存器，发起LPI中断。写操作，给ITS提供2个信息：

* EventID：值保存在GITS\_TRANSLATER寄存器中，表示外设发送中断的事件类型
* DeviceID：表示哪一个外设发起LPI中断。该值的传递，是实现自定义，例如，可以使用AXI的user信号来传递。

ITS将DeviceID和eventID，通过一系列查表，得到LPI中断号，再使用LPI中断号查表，得到该中断的目标cpu。

ITS将LPI中断号，LPI中断对应的目标cpu，发送给对应的redistributor。redistributor再将该中断信息，发送给CPU。

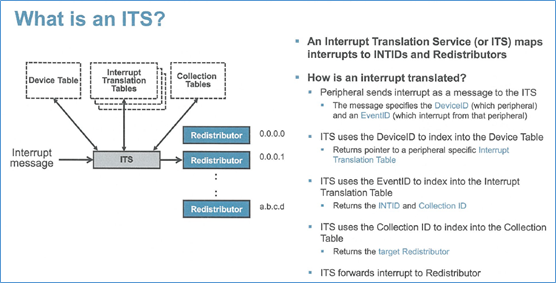
# 四、ITS

ITS是一个组件，用来提供给外设，发送LPI中断的，然后将LPI中断，发送给redistributor。

## 1、ITS处理流程

ITS使用三类表格，实现LPI的转换和映射：

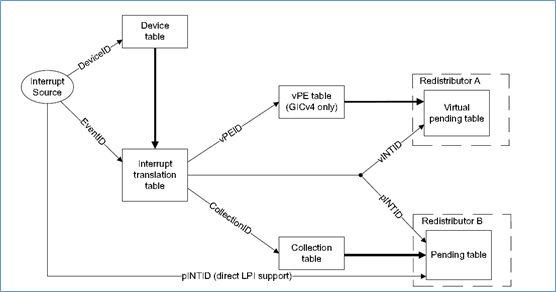
* device table： 映射deviceID到中断转换表
* interrupt translation table：映射EventID到INTID。以及INTID属于的collection组
* collection table：映射collection到redistributor



当外设往GITS\_TRANSLATER寄存器中写数据后，ITS做如下操作：

* 使用DeviceID，从设备表（device table）中选择索引为DeviceID的表项。从该表项中，得到中断映射表的位置
* 使用EventID，从中断映射表中选择索引为EventID的表项。得到中断号，以及中断所属的collection号
* 使用collection号，从collection表格中，选择索引为collection号的表项。得到redistributor的映射信息
* 根据collection表项的映射信息，将中断信息，发送给对应的redistributor

以上是物理LPI中断的ITS流程。虚拟LPI中断的ITS流程与之类似。以下是处理流程图：

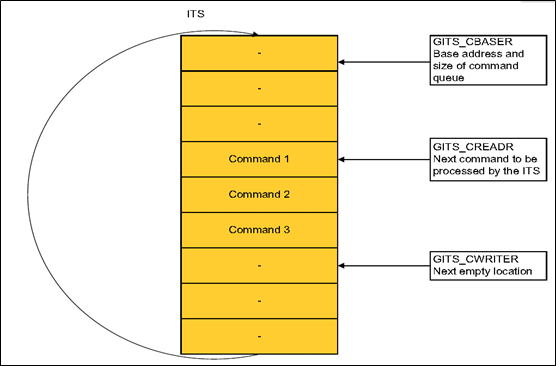


## 2、ITS命令

ITS操作，会涉及到很多表，而这些表的创建，维护是通过ITS命令，来实现的。虽然这些表，是在内存中的，但是GICv3和GICv4，不支持直接访问这些表，而是要通过ITS命令，来配置这些表。

ITS的操作，是通过命令，来控制的。外部通过发送命令给ITS，ITS然后去执行命令，每个命令，占32字节。

ITS有command队列，命令写在这个队列里面。ITS会自动的按照队列顺序，一一执行。

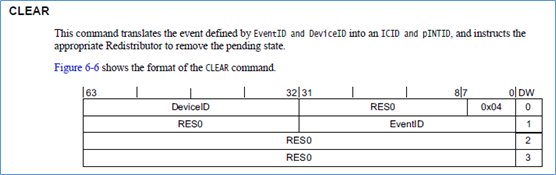


每个命令占32个字节。

命令，存放在内存中，GITS\_CBASE，保存命令的首地址。GITS\_CREADR，是由ITS控制，表示下一个命令的地址。GITS\_CWRITER，是下一个待写命令的地址。软件往GITS\_CWRITER地址处，写入命令，之后ITS就会执行这个命令。

ITS提供的命令，有很多，可以查阅GIC手册获取更多。

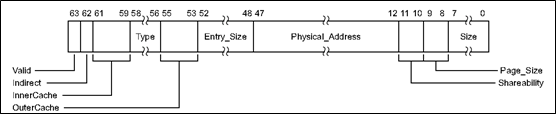
以下是CLEAR命令。



## 3、ITS table

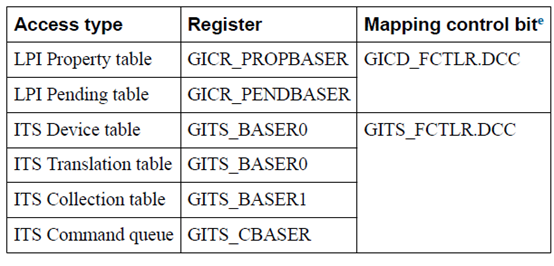
ITS包括很多个表，这些表均处于 non-secure区域。

GITS\_BASER<n>，指定ITS表的基地址和大小。软件，在使用ITS之前，必须要配置。



其中的Physical\_Address字段，就指定了表的基地址所在位置。

以下是各个表的基地址，对应的寄存器。



# 五、总结

gicv3中，引入了消息中断，并且为之，支持了LPI。分配了大量的中断号，用于LPI。对于LPI的实现，有2种方式，一种是访问redistributor提供的寄存器，一种是使用ITS。

不过对于手机arm cpu来说，其实是不需要LPI的，因为现有的中断，已经符合要求，加入了LPI，让gic更复杂，让软件操作，也更复杂。但是对于服务器arm cpu，这个，就需要了，因为这个可以和PCIE相连，实现消息中断。个人感觉，这个LPI中断，是为arm服务器cpu，所使用的。

对于提供的gic IP来说，比如gic600，是有配置选项，决定，是否是否支持LPI中断，以及是否需要ITS。

# ARM GIC（七）gicv3架构-power控制

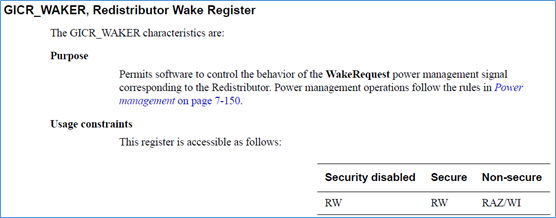
发表于[2018年7月29日](http://www.lujun.org.cn/?p=3926)由[卢 骏](http://www.lujun.org.cn/?author=3)

从gic3开始，cpu interface放到了PE中，因此cpu interface和PE是同一个power domain。而属于gic的其他组件，如redistributor，distributor，是另外一个power domain。因此就有如下一种情况，PE和cpu interface的电源给断掉了，而gic的电源并没有断掉。此时gic给cpu interface发送数据，cpu interface是不会响应的。

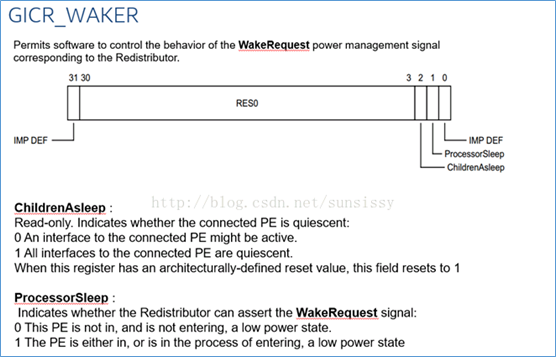
在这种情况下，gic提供了power管理功能。

# 一、GICR\_WAKER寄存器

gic中，提供了如下的 GICR\_WAKER 寄存器，来支持power功能。



其寄存器描述如下：



# 二、断电cpu interface和PE

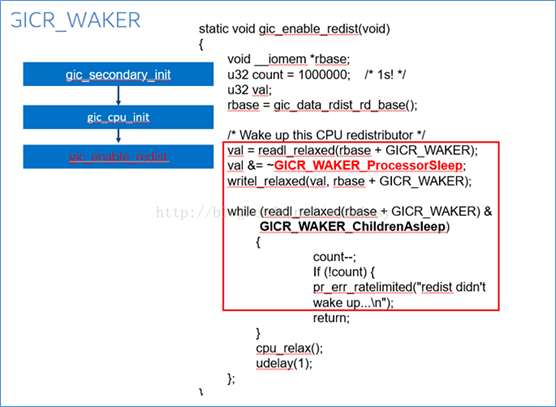
在cpu interface和PE要断电之前，软件要保证，通知redistributor，cpu interface和PE要进入low-power状态。软件，要往GICR\_WAKER寄存器的ProcessorSleep字段，写入1，表示PE要进入到low-power状态。cpu interface之后将自己置为low-power状态之后，就将ChildrenAsleep字段，设置为1。

当GICR\_WAKER.ChildrenAsleep为1之后，redistributor，不会在将中断，发送给cpu interface，distributor，在中断仲裁时，也不会考虑该PE。

# 三、唤醒cpu interface和PE

当gic要唤醒cpu interface和PE时，也是操作这个 GICR\_WAKER寄存器。

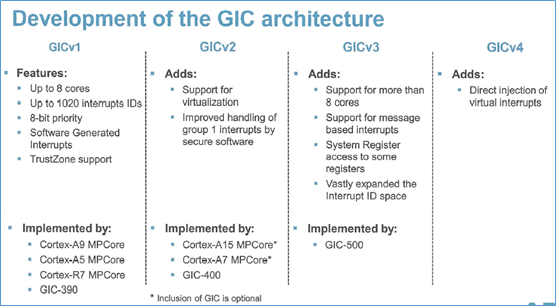
将processorsleep，写入0，然后去唤醒该cpu，最后读取childrenasleep，判断PE是否唤醒成功，



# ARM GIC（八）总结

发表于[2018年7月29日](http://www.lujun.org.cn/?p=3941)由[卢 骏](http://www.lujun.org.cn/?author=3)

GIC，是arm为了实现复杂的中断控制，而定义的一套架构。版本也历经了多个变化，从最初的GICv1到现在最新的GICv4。每一个新的版本，都增加了一些新的功能。

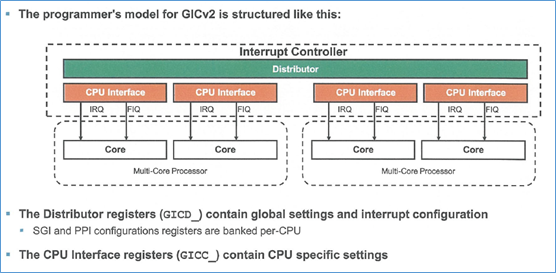


目前最新的GIC-600 IP，支持GICv4。

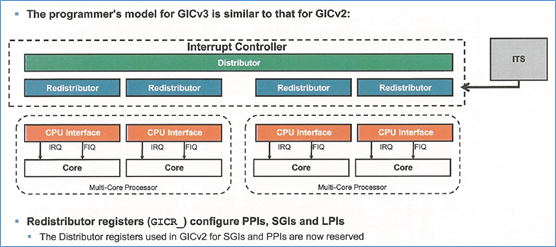
不过从GICv3开始，架构就和之前的架构，变化就比较大了。

# 一、变化一：cpu interface

下图是GICv2架构，cpu interface是实现在gic内部，而且gic的寄存器，都是memory-mapped方式访问。

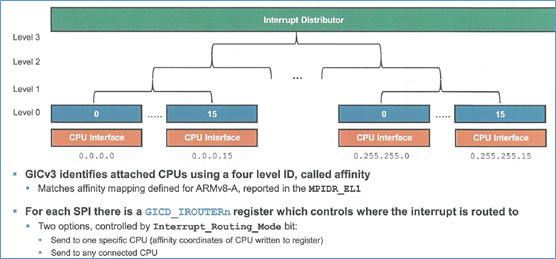


下图是GICv3架构，cpu interface从gic内部剥离，实现在PE的内部。并且将cpu interface的寄存器，提供了系统寄存器访问方式，从而实现中断的快速响应。



## 二、变化二：core的标识

GICv3中，对于core的标识，使用了属性层次的方式，来进行标识，从而可以支持更多的core。



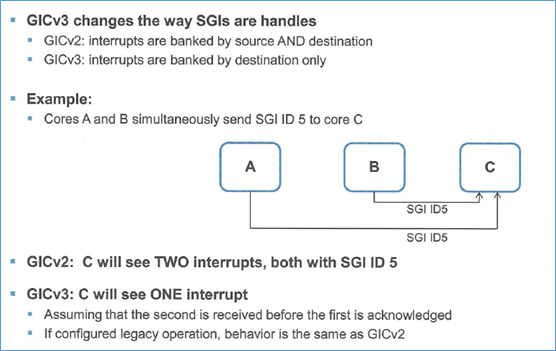
而GICv2中，支持最大8个core。

# 三、变化三：消息中断

GICv3中，加入了LPI中断类型，来实现消息中断。并且提供了ITS，来实现中断的转换。

# 四、变化四：SGI处理

对于SGI的处理，有如下的变化。



## 五、总结

gicv3/v4，架构，比gicv2架构，增加了很多的特性，从而支持更复杂的中断管理，支持更多的cpu。

自此，本系列博文到此就要结束了，基本上，除了虚拟中断的相关内容，我将GIC的内容都进行了介绍。希望大家看完这系列博文，能够对GIC有所认识。当初，自己也是看了很多的文档，外加上代码，才对这个理解的。

后面，如果我有去了解过虚拟中断，会在写一系列博文，来介绍虚拟中断。

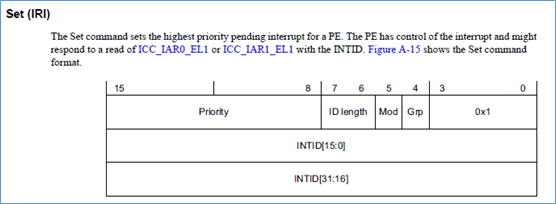
**ARM GIC（九） gicv3的中断分组**

发表于[2018年10月15日](http://www.lujun.org.cn/?p=3988)由[卢 骏](http://www.lujun.org.cn/?author=3)

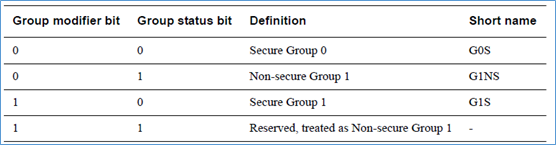
GICv3架构中，对中断进行了分组。分成了以下三个组：

* group0，用于EL3处理的中断
* secure group1：用于secure EL1处理的中断
* non-secure group1：用于non-secure的EL2和non-secure的EL1。

对于redistributor的set命令，带有Mod和Grp参数。



Mod与Grp共同表示，中断所属的组。其组合如下图所示：



对于每一组中断，有一个系统寄存器，来控制该组中断是否有效。

* ICC\_IGRPEN0\_EL1：针对group0的中断
* ICC\_IGRPEN1\_EL1：针对group1的中断，该寄存器分为non-secure和secure访问，不同的secure下，是访问当前secure下的寄存器

而每个中断的分组，由以下两个寄存器来决定：

* GICR\_IGROUPR<n>: interrupt group registers
* GICR\_IGRPMODR<n>: interrupt group modifier registers

每个中断，占寄存器中的1个bit，使用中断号进行索引。

当gic给cpu interface通过set命令发送中断，cpu能够响应该组中断，会回发activate命令，认可该中断。

如果cpu不能响应该组中断，会回发release响应。如下图所示：



gic给cpu interface通过set命令发送中断，中断号为93，优先级为0x40，Mod和Grp均为1，表示non-secure的group1。

cpu interface不能响应该中断，回发release响应。

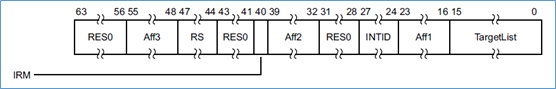
**ARM GIC（十） GICv3软中断**

发表于[2018年10月16日](http://www.lujun.org.cn/?p=4000)由[卢 骏](http://www.lujun.org.cn/?author=3)

软中断（software generated interrupts），用来多个核之间的通信（inter-processor communication）。软件通过写SGI寄存器来产生。

* 软件写ICC\_SGI1R\_EL1产生对应当前secure状态的group1软中断
* 软件写ICC\_ASGI1R\_EL1产生secure状态的group1软中断
* 软件写ICC\_SGI0R\_EL1产生secure状态的group0软中断

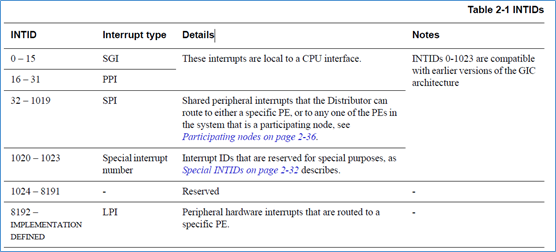
这三个寄存器的位域是一样的，如下图：



* Aff3.Aff2.Aff1：表示软中断目的CPU的属性层次。
* TargetList：表示要发给哪些CPU。
* INTID：表示软中断号
* IRM：软中断发送，是按照属性层次发送，还是发起其他所有的cpu
* RS：range selector，和TargetList结合，表示发送哪些CPU

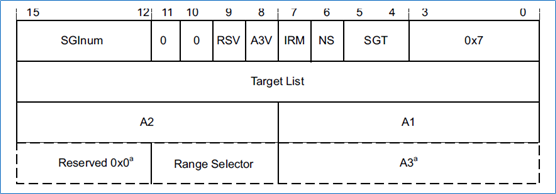
例如，IRM为0，INTID为1，Aff3.Aff2.Aff1为0.0.0，TargetList为0xf，RS为0，就表示，软中断发送给属性层次为0.0.0.[0-3]的cpu。

GICv3对软中断的中断号，进行了规定，只能是0-15。



一、cpu interface SGI命令

软件写SGI寄存器后，cpu interface会通过gic stream接口，发送SGI命令。



* SGT：表示写的哪一个SGI寄存器

0b00: ICC\_SGI0R\_EL1

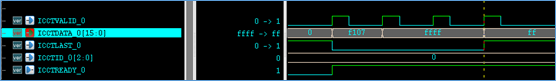
0b01: ICC\_SGI1R\_EL1

0b10: ICC\_ASGI1R\_EL1

0b11: Reserved

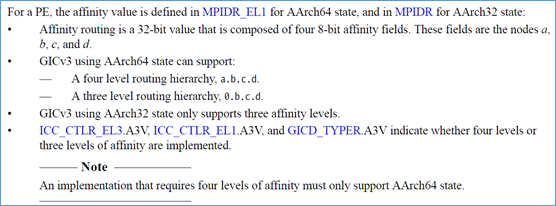
* NS: 当前的secure状态，0表示secure，1表示non-secure
* IRM: SGI寄存器的IRM bit
* A3V: aff3是否有效，如果有效，需要发送A3数据。
* RSV: range selector域是否有效
* SGInum: 软中断中断号
* A3,A2,A1: 对应Aff3.Aff2.Aff1
* TargetList: 对应SGI寄存器中的TargetList
* Range Selecto: 对应SGI寄存器中的RS域

比如，往ICC\_SGI0R\_EL1寄存器写0xffffef\_ffffffff，那么cpu interface会发送如下波形：



二、A3V

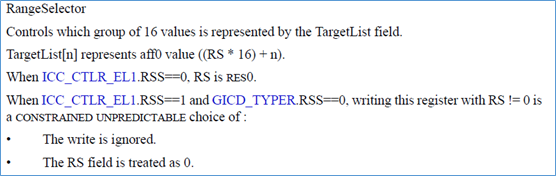
GICv3中，使用属性层次对CPU进行编号，属性层次最多有4层，最高层为Aff3，这一层可以通过cpu interface系统寄存器来控制，是否使能。



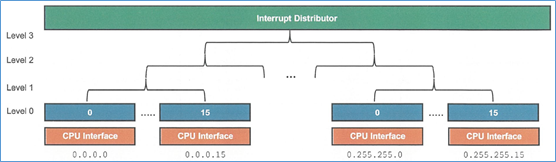
ICC\_CTLR\_EL3和ICC\_CTLR\_EL1的A3V表示cpu interface是否支持Aff3，而GICD\_TYPER.A3V表示gic ip是否支持Aff3。

三、关于range selector的理解。

gicv3的spec对range selector的解释如下：



gic3中，使用属性层次，来对CPU进行标识，这样可以精确的将中断发送指定的cpu。属性层次最高有4层，为aff3.aff2.aff1.aff0。每一个属性层次，用8bit来表示。如下图所示：



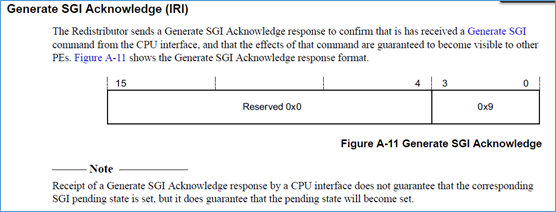
SGI中断，是可以同时发送给多个cpu的中断，通过TargetList来表示要发送给哪一些CPU。而TargetList只有16个bit，也就是只能发送给16个cpu，但是Aff0最多可以表示256个cpu，那怎么发送给其他的cpu的了？

这个时候range selector就派上用场了，RS域共4个bit，可以表示16个范围，16×16=256，刚好表示256个cpu。所以spec里面，说TargetList[n]表示的aff0的值为RS\*16 + n。

比如要给65-68号CPU发送软中断，首先判断这些cpu属于的range，为5，那么给这些CPU发送软中断，RS为5，TargetList为0x1E。

四、GIC的SGI认可响应

当GIC收到cpu interface发的SGI命令，需要回SGI认可响应。



# ARM GIC（十一） gicv3架构-two secure state

发表于[2018年11月20日](http://www.lujun.org.cn/?p=4046)由[卢 骏](http://www.lujun.org.cn/?author=3)

gicv3中，引入了支持2种安全状态（secure state），也就是对于中断，根据secure状态，分为安全中断和非安全中断。当然也可以只支持一种安全状态。。

这里的2种安全状态和1种安全状态，主要是影响中断分组，所使用IRQ和FIQ管脚的映射，以及gic中的寄存器访问。

# 一、中断线的映射

当gic架构，使用gicv3后，中断的传递，和gicv2有所区别。

gicv3中，将cpu interface从gic中抽离，放入到了cpu中，cpu interface通过gic stream接口，与gic进行通信。

当gic要发送中断，gic通过gic stream接口，给cpu interface发送中断命令，cpu interface收到中断命令后，根据中断线映射配置，决定是通过IRQ还是FIQ管脚，向cpu发送中断。

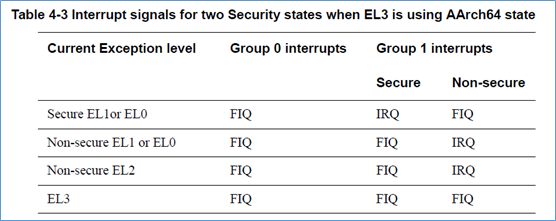
而中断线映射配置，要根据中断的分组以及当前cpu所处的EL以及seucre状态，来决定。

# 二、2种安全状态中断线映射

当gic支持2种安全状态，EL3是AArch64和AArch32，映射情况不同

## 1、EL3是AArch64

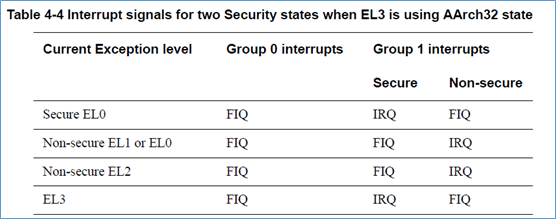
当EL3是AArch64时，映射如下：



* 对于group0中断，中断线均映射到FIQ
* 对于group1安全中断，secure EL1或EL0，中断线映射到IRQ，其他EL映射到FIQ
* 对于group1非安全中断，secure EL1或EL0以及EL3，中断线映射到FIQ，其他EL映射到IRQ

## 2、EL3是AArch32

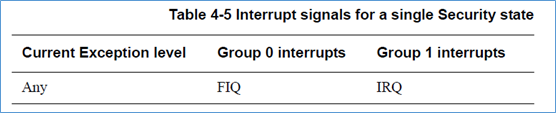
当EL3是AArch32时，映射如下：



* 对于group0中断，中断线均映射到FIQ
* 对于group1安全中断，secure EL0和EL3，中断线映射到IRQ，其余EL映射到FIQ
* 对于group1非安全中断，secure EL0和EL3，中断线映射到FIQ，其余EL映射到IRQ

# 三、1种安全状态中断线映射

映射如下：

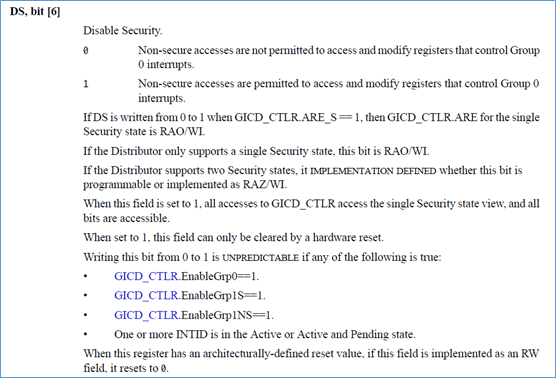


* group0中断线，直接映射到FIQ
* group1中断线，直接映射到IRQ

# 四、GICD寄存器

在GICD中的GICR\_CTLR寄存器的DS bit，表示是否支持2种安全模式。

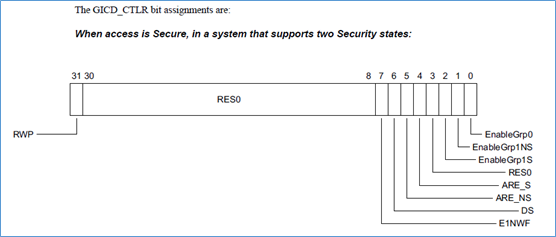
该bit描述如下，如果0，表示支持2种安全状态，为1，表示不支持。



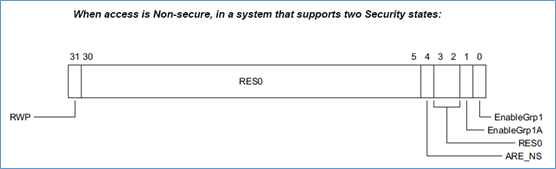
## 1、支持2种安全模式下GICD\_CTLR

在支持2种安全模式下，GICD中寄存器会进行备份成2份，一份提供给secure访问，。一份提供给non-secure访问。

比如对于GICD\_CTLR寄存器，secure访问，寄存器描述如下：

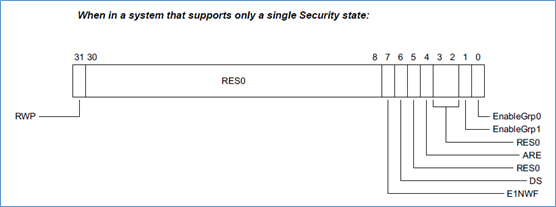


而如果是non-secure访问，其寄存器描述如下：



## 2、支持1种安全模式下GICD\_CTLR

在1种安全模式下，寄存器描述如下，此时不论是non-secure访问，还是secure访问，都访问的同一个寄存器。

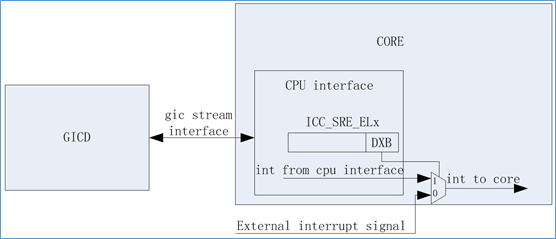


# ARM GIC（十二） 中断bypass

发表于[2018年12月7日](http://www.lujun.org.cn/?p=4061)由[卢 骏](http://www.lujun.org.cn/?author=3)

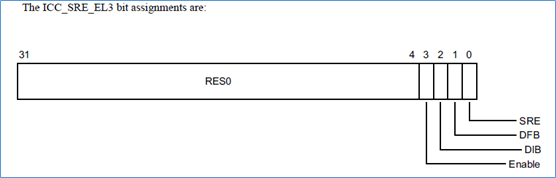
在GICv2架构中，GIC与core之间，是直接通过irq，fiq管脚，传递中断信号。但是在GICv3架构中，GIC通过gic stream接口向cpu interface传递中断信息，然后由cpu interface向core传递中断信息，而且，cpu interface被设计在了core当中。

GICv3支持中断bypass功能，以下是我画的一个框图：

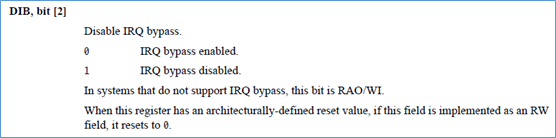


对于core而言，中断有2种（不考虑虚拟中断），分别是IRQ和FIQ。

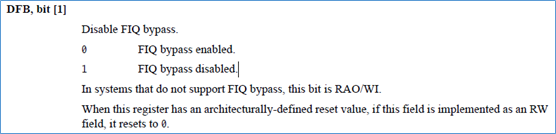
在ICC\_SRE\_ELx寄存器，有DFB和DIB bit，分别控制FIQ，IRQ是否bypass。



对于DIB bit的描述：



对于DFB bit的描述：



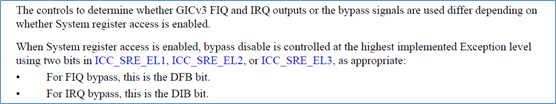
这2个bit，复位值都是0，表示系统在复位后，中断bypass功能是开启的，此时外部的中断信号，是直接输入到core中的。

当软件，关闭中断bypass后，此时，才是由cpu interface给core发送中断，而忽略掉外部发送的中断。

在gicv3架构描述中，提到了中断bypass功能，会受到下面几个配置的影响：

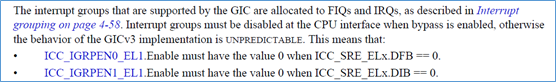
一个是，是否运行系统寄存器访问icc寄存器，也就是ICC\_CTLR.SRE bit是否为高，如果允许系统寄存器访问icc寄存器，那么中断bypass功能是禁止的。

一个是，ICC\_SRE寄存器，由当前设计中支持的最高EL级的ICC\_SRE寄存器的DFB和DIB bit来控制。

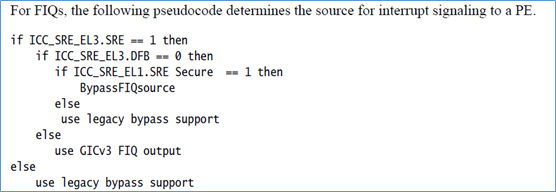


最后一个，就是中断分组使能，也就是ICC\_IGPREN0\_EL1和ICC\_IGRPEN1\_EL1的enable bit。

如果需要开启中断bypass功能，那么需要将中断分组使能给关掉。



以下是FIQ中断bypass的伪代码：



IRQ的处理，和FIQ是一致的。

**ARM GIC（十三） 波形为例，介绍gic600与cpu interface通信**

发表于[2018年12月7日](http://www.lujun.org.cn/?p=4088)由[卢 骏](http://www.lujun.org.cn/?author=3)

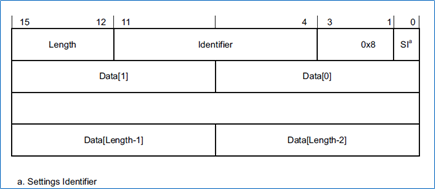
以下以gic600与cpu interface之间传递的包，来说明，他们之间是如何通信的。这里以波形进行介绍。这样比较直观。

一、downstream control命令包

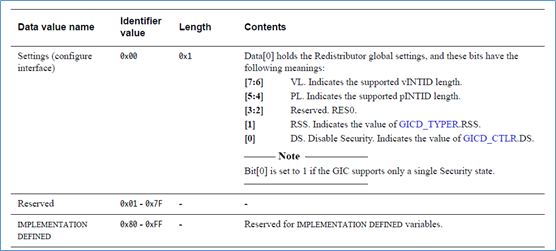
gic600发送downstream control命令包给cpu interface。命令包的数据为0。



downstream control包格式如下：



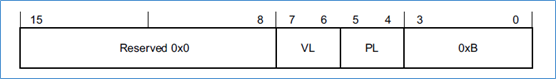
解释如下：



从表中可以解析，gic600告诉cpu interface做如下配置：

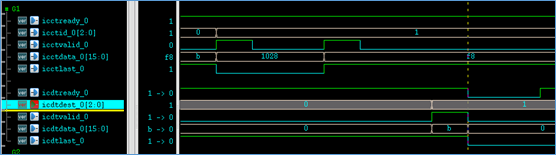
* VL设置为0，表示vINTID位宽是16bit
* PL设置为0，表示pINTID位宽是16bit
* RSS设置为1，表示SGI的aff0，支持0-15
* DS设置为0，表示支持两种secure模式

cpu interface接收到该命令包后，回一个downstream control acknowledge响应包。包格式如下：

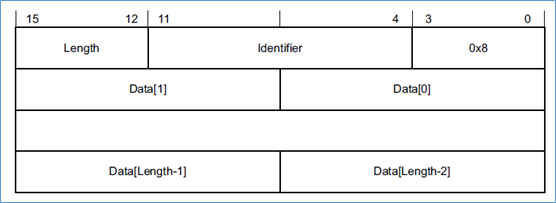


upstream control命令包，设置PMR

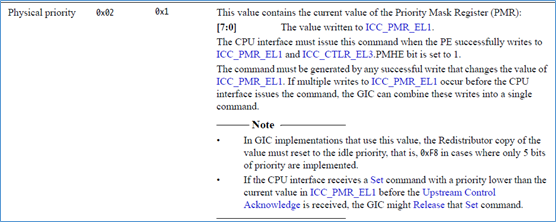
cpu interface给gic600发送upstream control命令包，数据为f8。设置PMR。



该包的格式如下：



对于f8，解析如下：



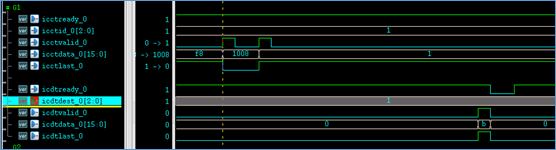
cpu interface告诉gic600，当前的ICC\_PMR\_EL1值设置为f8，也就是将来优先级比这个低的，gic600就不要发送中断给cpu interface。

gic600收到upstream control命令包后，回upstream control acknowledge包，格式如下：

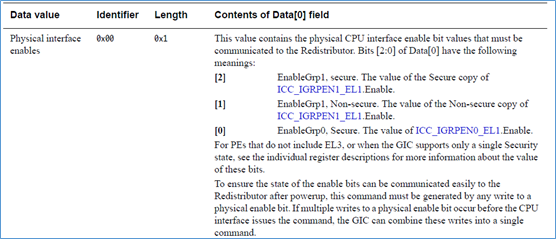


二、upstream control命令包，设置group enable

cpu interface给gic600发送upstream control命令包，数据为1。设置group enable。



对于数据1，解析如下：



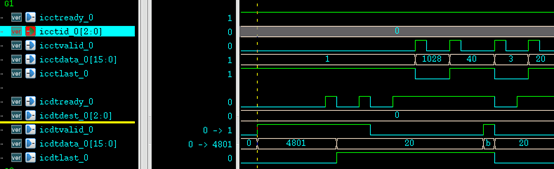
cpu interface告诉gic600，将grou0的中断使能，group1的secure和non-secure中断不使能。

gic600收到upstream control命令包后，回upstream control acknowledge包，格式如下：

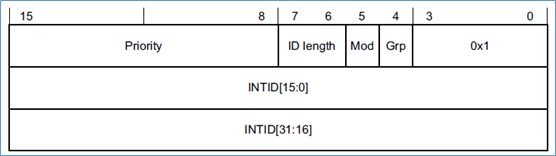


三、set命令包，发送中断

gic600发送set命令包给cpu interface。数据为20。



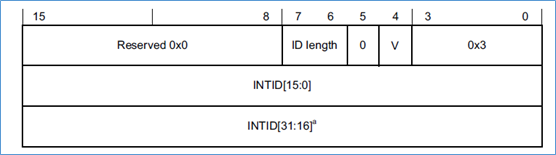
set命令包格式如下：



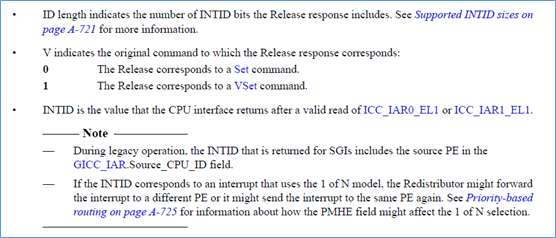
gic600向cpu interface发送了一个中断：

* Grp为0，Mod为0，表示中断属于group0中断组
* ID length为0，表示INTID位宽为16bit
* Priority为0x48，表示中断优先级
* INTID为0x20，表示中断号为32（SPI中断从32开始）

cpu interface收到该set命令后，发现不能处理该中断，因此发送release响应包，并且INTID指定刚刚gic600发送的中断号。



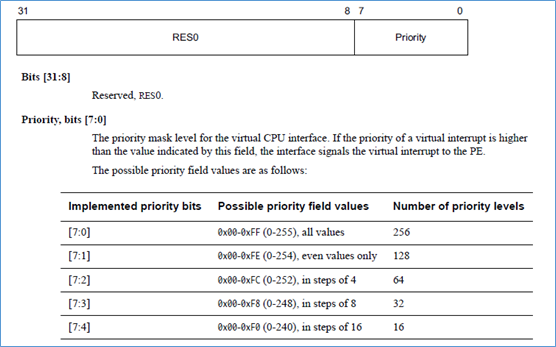
该release响应包的各个参数解释如下：



gic600收到cpu interface的release响应包后，知道之前发送的中断，不能得到cpu interface的响应，在之后，会重新发送该中断。

这里为什么cpu interface不能处理该中断，而是要回release响应包。原因在于中断优先级不符合要求。

对于ICC\_PMR\_EL1寄存器，gicv3 spec描述如下：



对于中断，gic架构，最多使用8个bit，来表示中断优先级。当然8个bit可以全用，也可以只用一部分，所以架构提供了5种配置方式。设计可以根据自己的需要，选择一种配置即可。

对于使用的bit不一样，对应的中断优先级值不一样。比如对于使用5个bit，那么就是[7:3]，共32种中断优先级。这个选择多少个bit，是硬件决定好了，软件是不能更改的。

软件可以设置这个寄存器，来表示，符合要求的优先级，可以被cpu响应。优先级值越小，表示优先级越高。

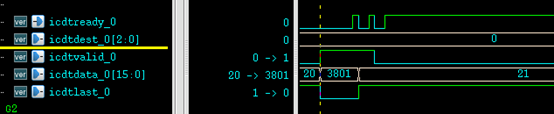
假设这个时候，软件往这个寄存器写了8，表示优先级高于8的，都不能被core响应。

假设设计，使用[7:3]，表示中断优先级。

此时，发送了优先级为0x48的中断，按照[7:3]bit，进行选择，得到中断优先级为9，高于设置的8，因此这个中断不能被core响应，所以cpu interface直接给gic600回release响应。

四、set命令包，发送中断

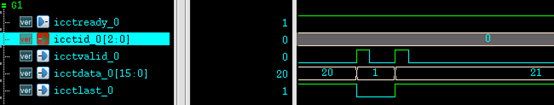
gic600发送set命令包给cpu interface。数据为21。



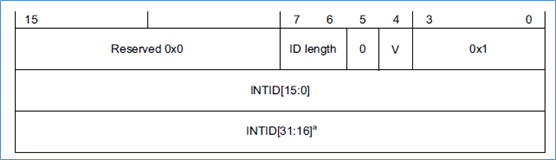
gic600向cpu interface发送了一个中断：

* Grp为0，Mod为0，表示中断属于group0中断组
* ID length为0，表示INTID位宽为16bit
* Priority为0x38，表示中断优先级
* INTID为0x21，表示中断号为33

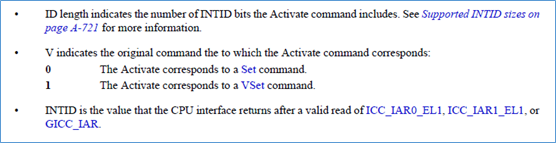
cpu interface接收到set命令包，将中断发送给core，core响应中断，读取icc\_iar寄存器，表示对该中断认可。中断被core认可后，cpu interface就会给gic600回activate命令包，表示core认可了该中断。gic600就可以把该中断状态，更新为active或者active and pending状态。



activate就命令包格式如下：

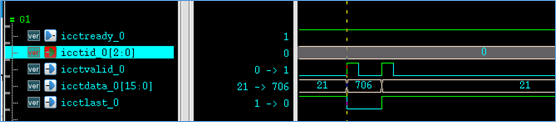


各个域解析如下：

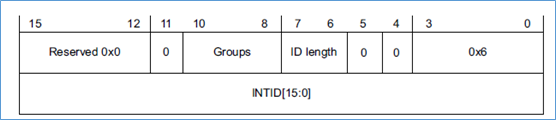


五、deactivate命令包，无效中断

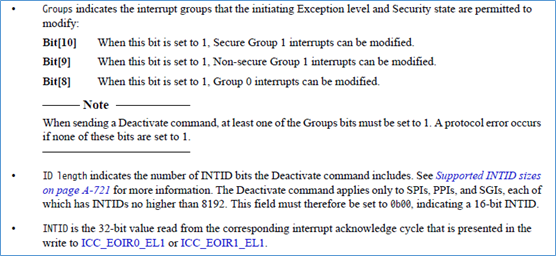
core在处理完中断后，会写ICC\_EOIR0\_EL1或者ICC\_EOIR1\_EL1寄存器，来无效中断，cpu interface接受core的这个写操作，会向gic600发送deactivate命令包，表示core对中断处理完毕。



deactivate命令包，格式如下；



各个域解析如下:

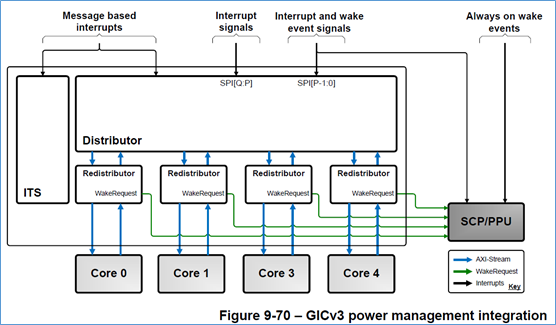


gic600收到cpu interfae发送的deactivate命令后，将自己内部对该中断维护的状态，修改为pending 或者 inactive。

# ARM GIC（十四）gicv3架构-power控制详解

发表于[2019年1月3日](http://www.lujun.org.cn/?p=4115)由[卢 骏](http://www.lujun.org.cn/?author=3)

在带有gicv3的soc架构中，其框图如下所示：



gicv3中的redistributor与core中的cpu interface通过AXI-Stream进行通信。

# 一、connection

当core上电之后，需要将core中cpu interface与gic中的redistributor进行connect，这样将来gic才可以将中断发送给core。

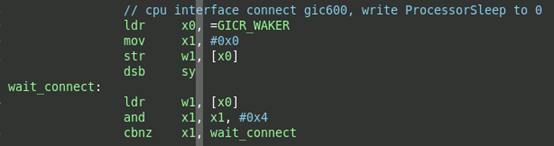
connection的流程如下所示：



描述如下：

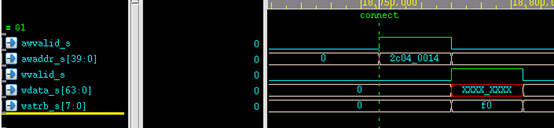
* 执行在core的程序，将GICR\_WAKER.ProcessorSleep位给置低，表示要connect redistributor
* redistributor在完成connect之后，将GICR\_WAKER.ChildrenAsleep位给置低，表示connect完成
* 执行在core的程序，查询GICR\_WAKER.ChildAsleep位是否为0，如果不是，表示redistributor还没有完成connect操作，就继续查询
* 如果查询到0，表示connect完成，接着做之后的初始化工作

其汇编代码如下：



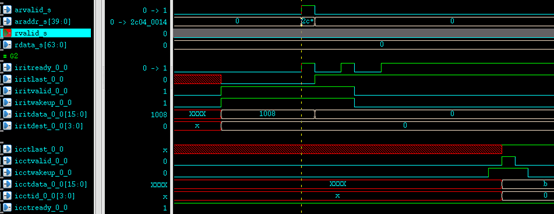
波形如下：

首先写GICR\_WAKER寄存器，将ProcessorSleep位给置低。

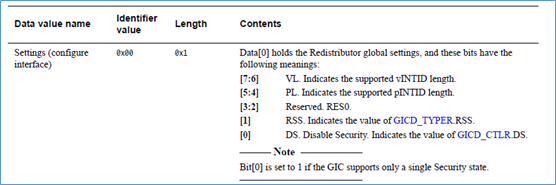


然后读取GICR\_WAKER寄存器，判断ChildAsleep位是否为0。在波形中，有读取到0，表示connect成功。

connect成功后，gic600会给cpu interface发送downstream包，设置vINTID，pINTID,RSS,DS这几个域。



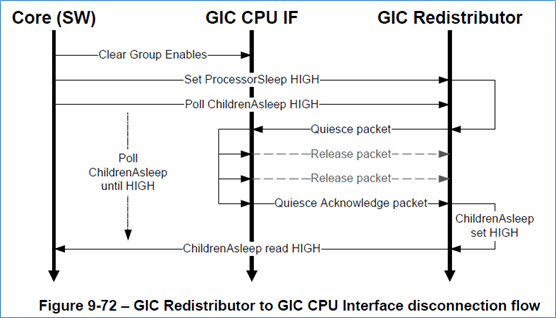
downstream包，对于identifier为0，length为1的解释如下：



# 二、disconnection

当core下电之后，需要将core中cpu interface与gic中的redistributor进行disconnect，这样将来gic才不会将中断发送给core。

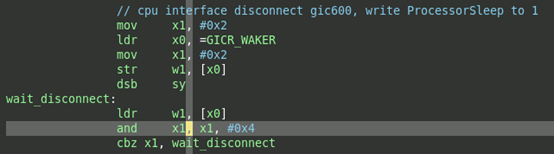
disconnection的流程如下所示：



描述如下：

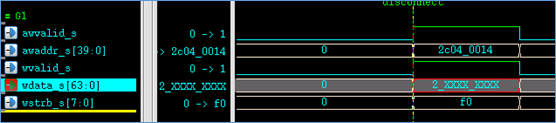
* 执行在core的程序，先将cpu interface的中断组使能给disable
* 执行在core的程序，将GICR\_WAKER.ProcessorSleep位给置高，表示要disconnect redistributor
* redistributor给cpu interface发送 Quiesce包
* cpu interface清掉内部所有pending的中断
* 清除完毕后，cpu interface回发Quiesce Acknowledge包给redistibutor
* redistributor收到cpu interface回发的响应之后，将GICR\_WAKER.ChildrenAsleep位给置高，表示disconnect完成
* 执行在core的程序，查询GICR\_WAKER.ChildAsleep位是否为1，如果不是，表示redistributor还没有完成connect操作，就继续查询
* 如果查询到1，表示disconnect完成

其汇编代码如下：

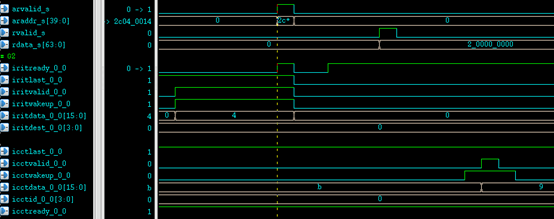


其波形如下：

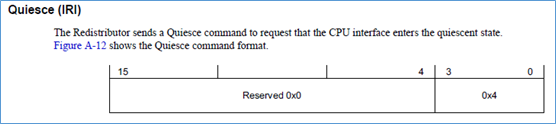
首先写GICR\_WAKER寄存器，将ProcessorSleep位给置高，表示要disconect。



然后读取GICR\_WAKER寄存器，判断ChildAsleep位是否为0。在波形中，有读取到0，表示disconnect不成功，需要再次读取判断。



gic600给cpu interface发送Quiesce包。



cpu interface收到该命令包，完成内部的操作后，回发quiesce acknowledge响应包。



至此，完成了disconnect操作，此时再读取GICR\_WAKER寄存器，ChildAsleep位就为1了。

http://www.lujun.org.cn/wordpress/wp-content/uploads/2019/01/010319_0536_ARMGICgicv13.png

# 三、中断唤醒core

当core下电之后，gic就不再会给core发送中断。如果此时有一个中断是唤醒core的，那么其处理流程应该如何了？

在gicv3，为每一个redistributor，提供了WakeRequest输出信号。当GICR\_WAKER的ProcessorSleep为1，此时外部有唤醒该core的中断请求，那么WakeRequest信号会被置高。

WakeRequest信号，会被连接到SCP或者PMU，也就是下图中的红色连线。当SCP或者PMU接收到WakeReqeust请求，就会将对应core给上电，然后core再connect redistributor，gic在将中断发送给该core，core再响应中断。

