计算机组成原理与接口技术——基于MIPS架构

Apr, 2022

第8讲 中断技术

杨明
华中科技大学电信学院
myang@hust.edu.cn

School of EIC, Huazhong University of Science and Technology, Wuhan, Hubei 430074, P.R. China

Agenda

- 内容
  - 中断的基本概念，中断响应过程
  - 典型微处理器中断系统简介
  - Xilinx的中断控制器-AXI INTC
  - GPIO中断方式接口设计
  - AXI Timer接口
  - AXI SPI接口
- 目的
  - 理解Interrupt 的含义，优点、分类；
  - 理解中断源、中断请求、中断类型码、中断优先级、中断向量入口地址(Interrupt Vector Address)、中断向量表等术语的含义和作用；
  - 理解CPU响应中断的过程；
  - 理解X86和Microbalze系统的中断处理过程；
  - 掌握AXI INTC原理，学会Microbalze系统中断程序设计；
  - 掌握AXI Timer接口设计；
  - 掌握AXI SPI接口设计。

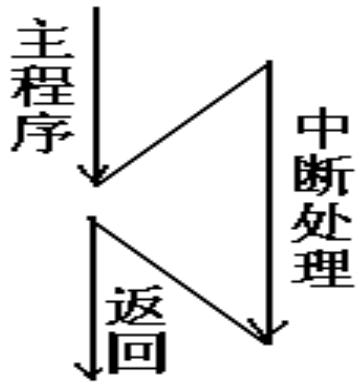
8.1 中断的基本概念

▶ 中断含义

- 中断（Interrupt）是指CPU在执行当前程序的过程中，由于某种随机出现的外设请求或CPU内部的异常事件，使CPU暂停正在执行的程序而转去执行相应的中断服务处理程序；当服务处理程序运行完毕后，CPU再返回到暂停处继续执行原来程序的过程。

- 主程序被中断的地方称为断点。
  - 断点处的指令是CPU完成中断处理后返回主程序时恢复执行的第一条指令，该指令的存储地址称为中断返回地址。

- Interrupt 的好处
  - 减轻CPU的负荷；
  - 提高系统的实行性；
  - 使CPU脱离顺序处理，实现多任务(分时)成为可能；



回顾：3.6 微处理器异常处理原理

▶ 异常处理机制

- 异常处理
  - 计算机在正常运行过程中，由于种种原因，使CPU暂时停止当前程序的执行，转而去处理临时发生的异常事件，处理完毕后，再返回去继续执行暂停的程序。
- 微处理器要能够实现异常处理需要完成以下几方面的功能：
  - 记录异常发生的原因
  - 记录程序断点处的指令在存储器中的地址
  - 记录不同种类的异常处理程序在内存中的地址
  - 建立异常种类与异常处理程序地址之间的对应关系。

才能在遇到某个特定的异常事件时去执行相应的异常处理程序。

▶ 断点保存和返回

- 寄存器法（嵌入式）
  - 在微处理器中设计一个寄存器EPC，当微处理器出现异常时，就将PC的值保存到EPC中。异常处理完之后，再把EPC的值赋给PC，这样就可以实现中断的返回
- 栈（PC）
  - 微处理器直接将PC的值压入栈中，异常处理完之后，再从栈顶把值弹出来赋给PC

8.1 中断的基本概念

► 分类

- 微机系统的中断可分为两大类：
  - 软中断
    - 指由CPU内部原因引起的中断，也叫内中断，统称为异常。又分为两大类：
      - ► ① 指令引起的异常
      - ► ② 处理器检测的异常
  - 硬中断
    - 指由CPU外部事件引起的中断，又叫外中断，简称中断。又分为：
      - ►①非屏蔽中断NMI（ Non Maskable Interrupt ）
      - ►②可屏蔽中断INTR
    - 前者不受CPU内部的中断允许标志IF的控制，而后者受控制。

8.1 中断的基本概念

▶ 中断源

- 引起中断的原因或能发出中断请求的来源（通常是硬件设备）称为中断源(Interrupt Sources)。如：
  - 上电复位中断；
  - 断点中断；
  - Timer中断；
  - 键盘/鼠标/打印机中断；
  - A/D中断；
- 中断请求(IRQ—Interrupt Request)：中断源向CPU发出的中断服务申请信号；
  - 如：0808的EOC信号
- 为便于管理各种不同的Sources/IRQ，系统通常给每种不同的Sources/IRQ都规定一个唯一的中断号，如0号中断、1号中断。

回顾：3.6 微处理器异常处理原理

▶ 异常事件

异常种类	来源	MIPS处理器命名
I/O设备	外部	中断
用户程序唤醒操作系统	内部	异常
计算结果溢出	内部	异常
未定义的指令（非法指令）	内部	异常
硬件出错	两者	异常或中断

▶ 异常事件识别机制

- 状态位法（MIPS)：
  - 在微处理器中利用一个寄存器对每种异常事件确定一个标志位，当有异常事件发生时，寄存器中对应的位置1，一个32位的寄存器可以表示32种不同类型的异常事件
- 向量法(Intel）：
  - 对不同类型的异常事件进行编码，这个编码叫中断类型码或异常类型码。

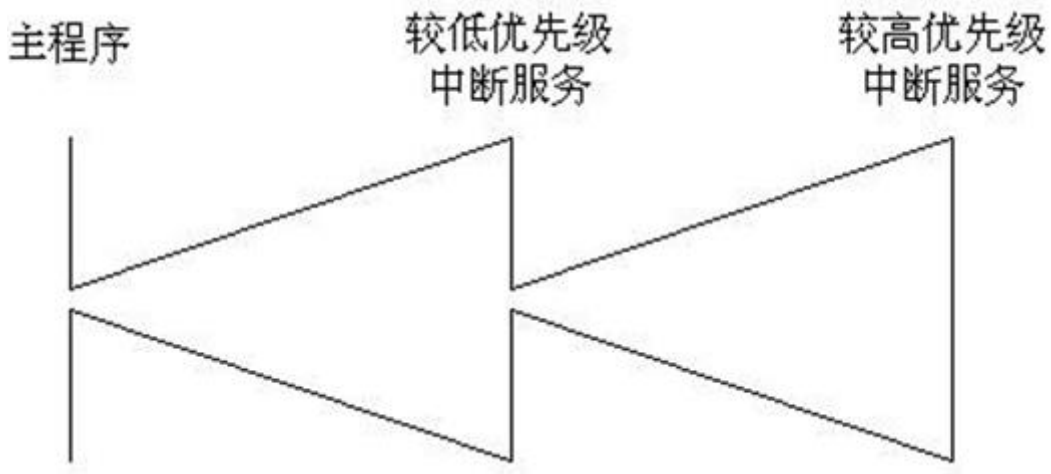
▶ 中断优先级（Interrupt Priority）

- CPU同一时刻只能响应一个中断源的申请
- 通常系统中具有多个Interrupt Sources，当多个不同的Interrupt Sources同时向CPU发出IRQ时，CPU最先响应哪个IRQ、最后响应哪个IRQ需要按照何种规则？——中断优先级(Interrupt Priority)。
- 应按各中断源的轻重缓急程度来确定它们的优先级别。
  - 系统事先给各种不同Interrupt Sources设定不同级别的Priority，同时出现不同IRQ时，Interrupt Priority高的IRQ将先得到CPU的响应，Interrupt Priority低的IRQ只有CPU处理完优先级别高的IRQ之后才会得到相应。
- 不同的系统(8088/Intel-51/Motorola-08/TI-MSP438)具有各自不同的Interrupt Sources，视各自外围模块的多少而定；同一个系统不同IRQ的Interrupt Priority的高低通常可以通过软件设置相应的寄存器来改变。

8.1 中断的基本概念

▶ 中断嵌套

- 又称多重中断处理，是指在处理一个中断请求服务的过程中，允许再去响应另一个中断请求。

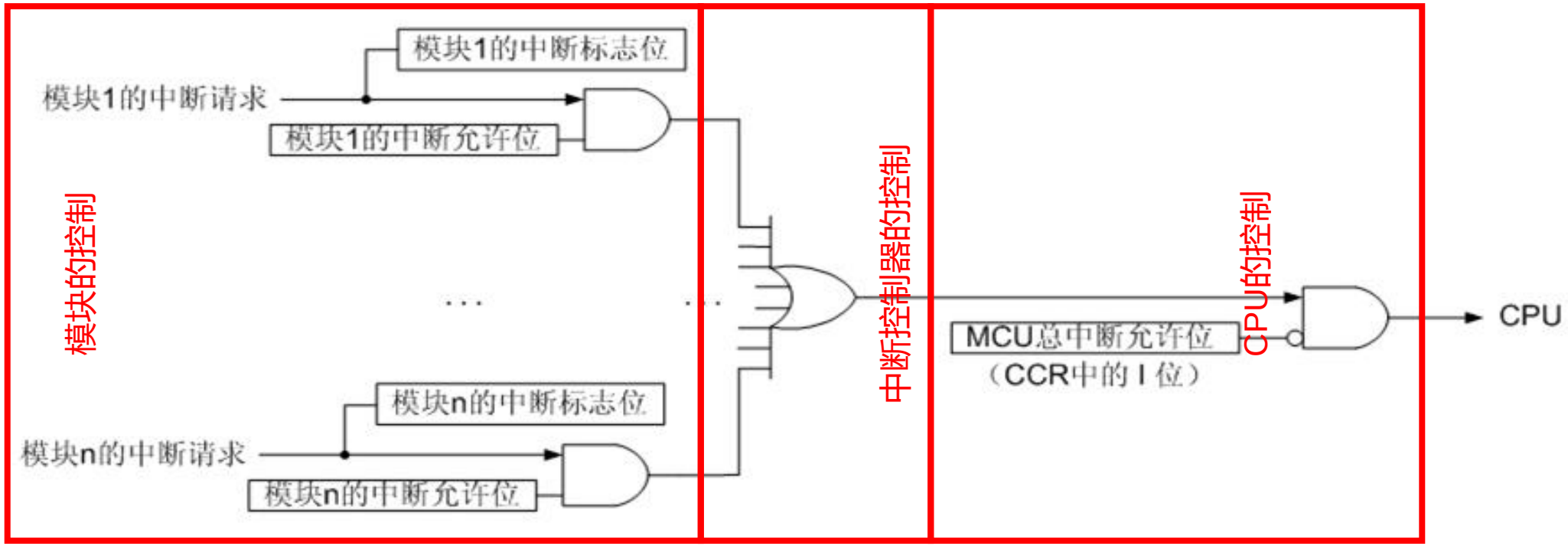


- 中断嵌套的基本原则：
  - 高优先级的中断请求可以打断低级中断的服务过程；
  - 同级或低一级别的中断不能打断同级别或高一级别的正在进行的中断服务过程。

8.1 中断的基本概念

▶ 中断模型

- CPU是否能够收到可屏蔽中断，一般受三个控制位控制
  - 模块的中断控制
  - 中断控制器的控制
  - CPU的控制

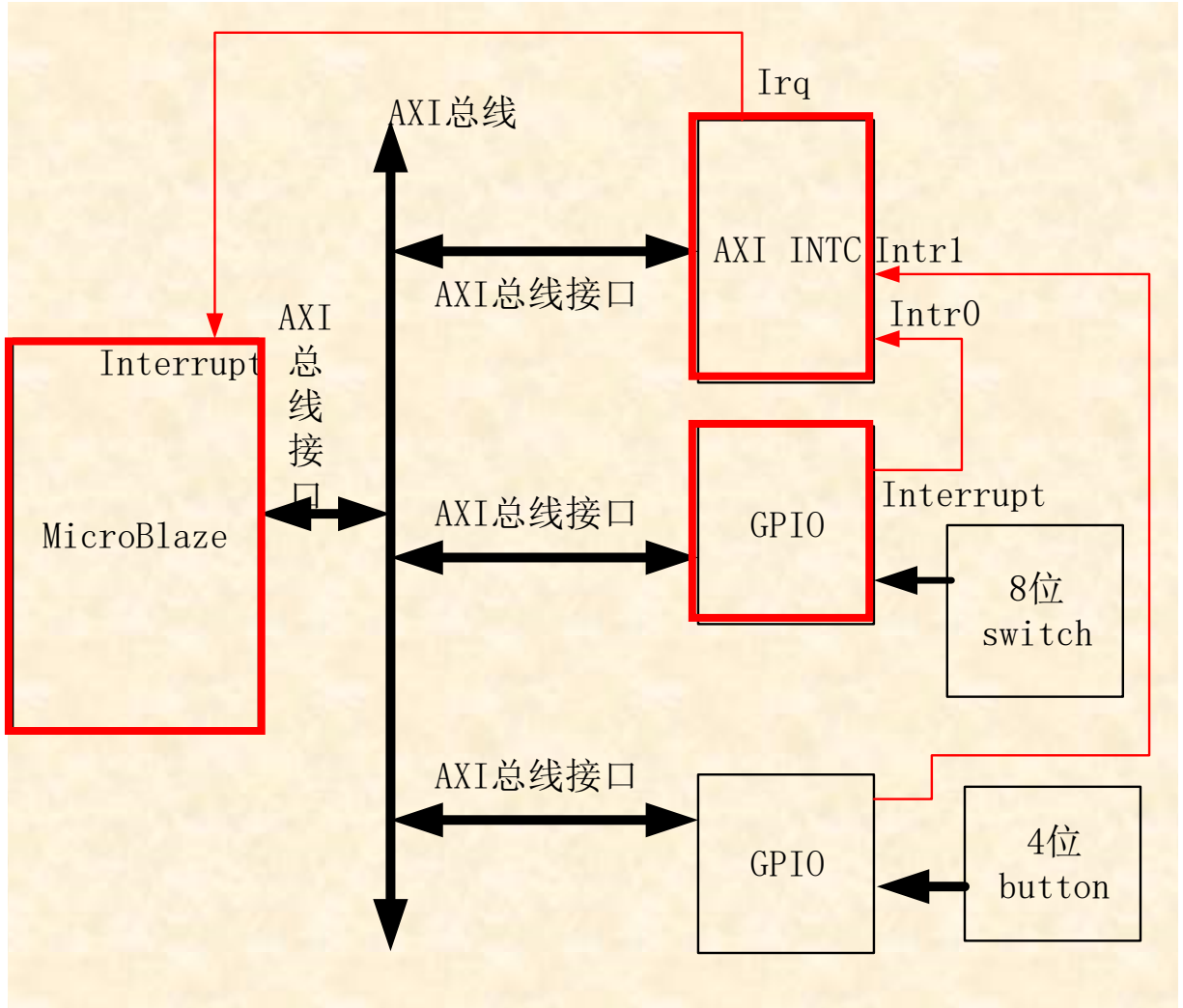


8.1 中断的基本概念

中断模型

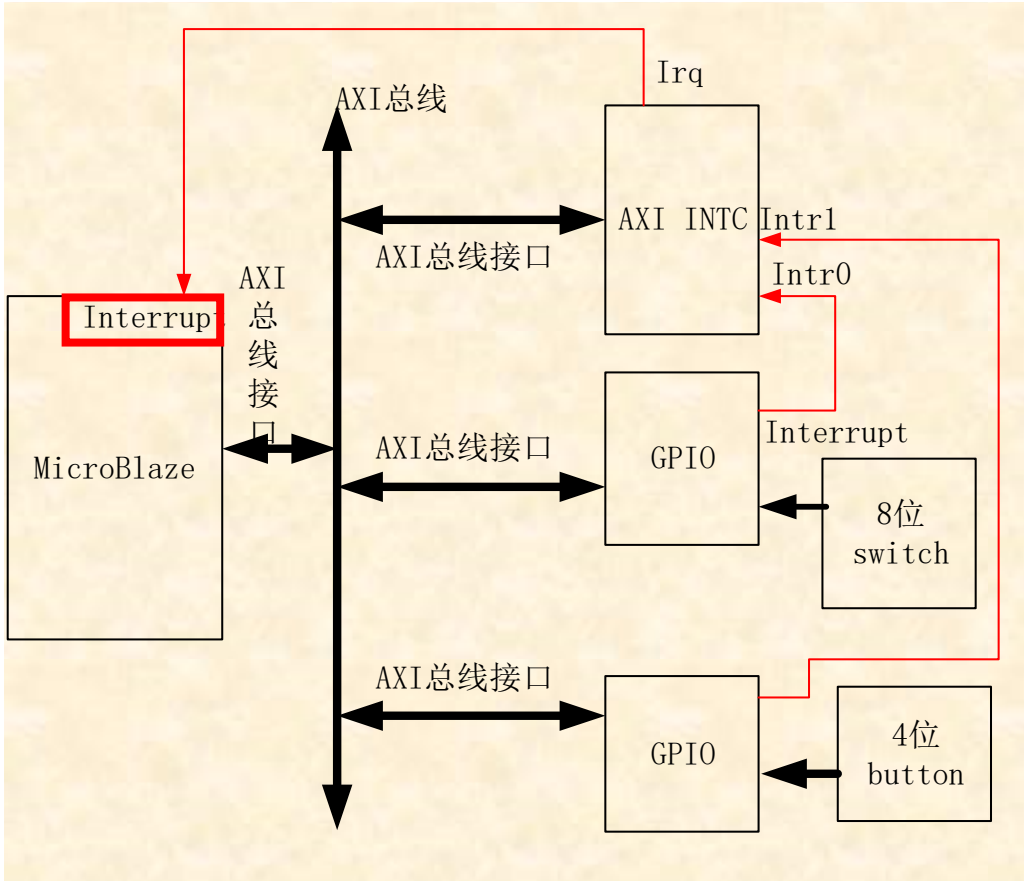
- MicroBlaze中断系统硬件电路框图
- 中断的三个层次
  - 模块(GPIO)
  - 中断控制器(INTC)
  - CPU(MicroBlaze)

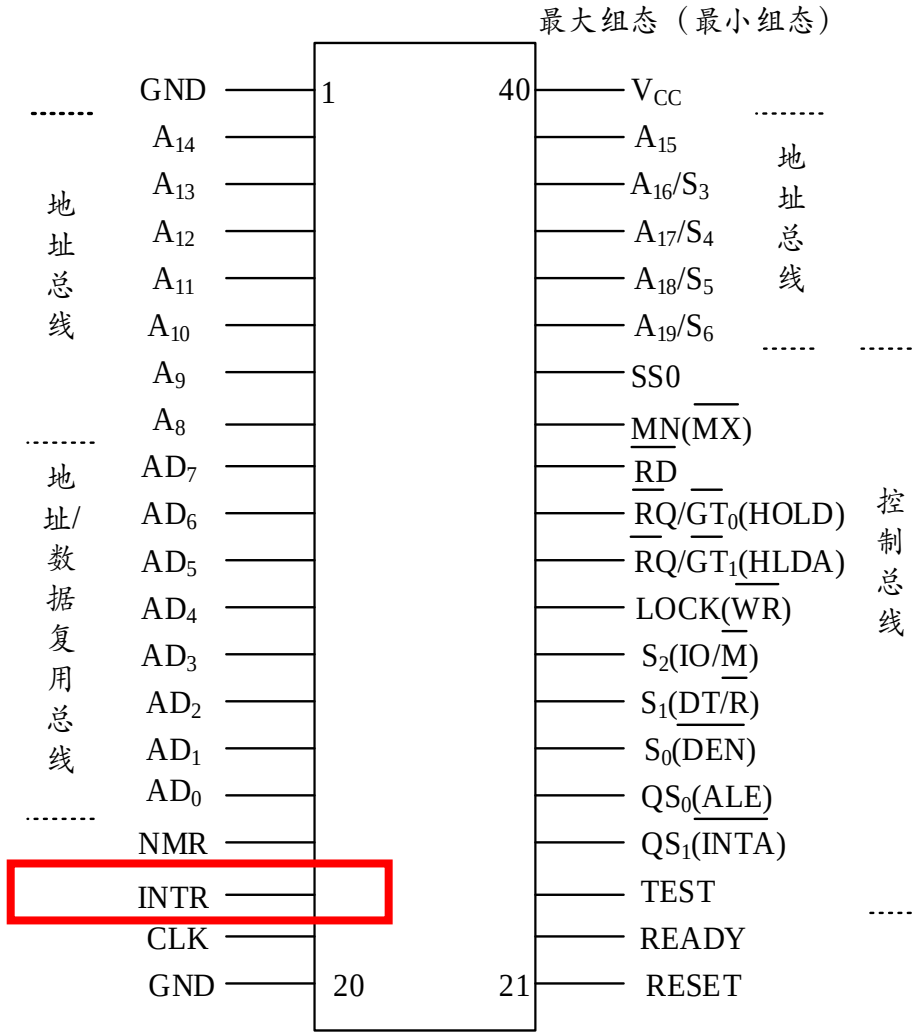
- CPU能否收到模块的中断请求取决于
  - 模块的中断控制
  - 中断控制器的控制
  - CPU的控制



8.1 中断的基本概念

- 微处理器中断信号的引入
  - X86
  - MIPS



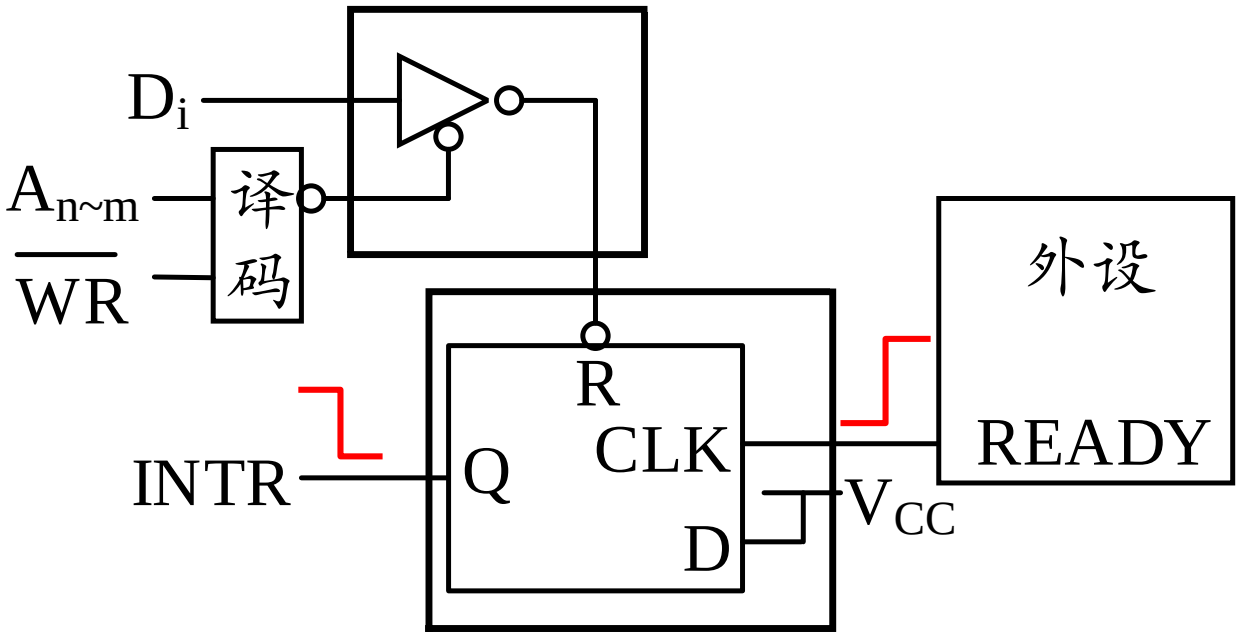


8.1 中断的基本概念

- 中断系统一般应具有以下功能：
  - 中断请求信号保持与清除，
  - 中断源识别，
  - 中断允许控制，
  - 中断优先级设置。

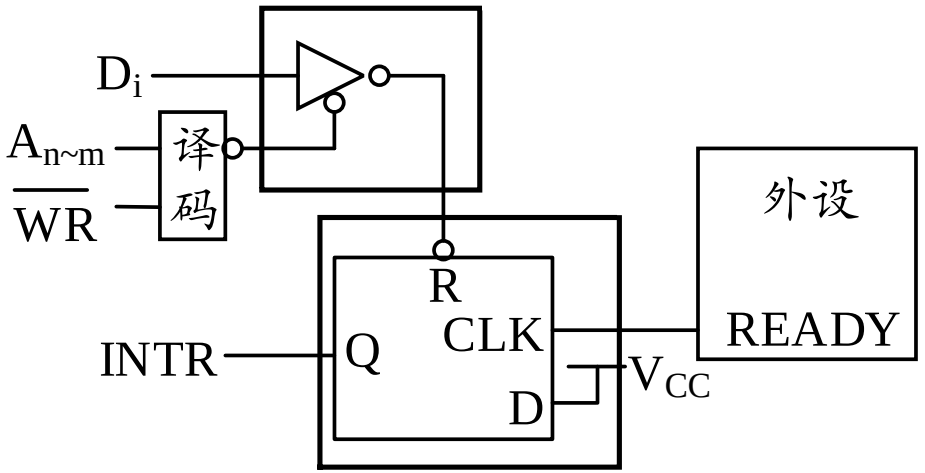
8.1 中断的基本概念

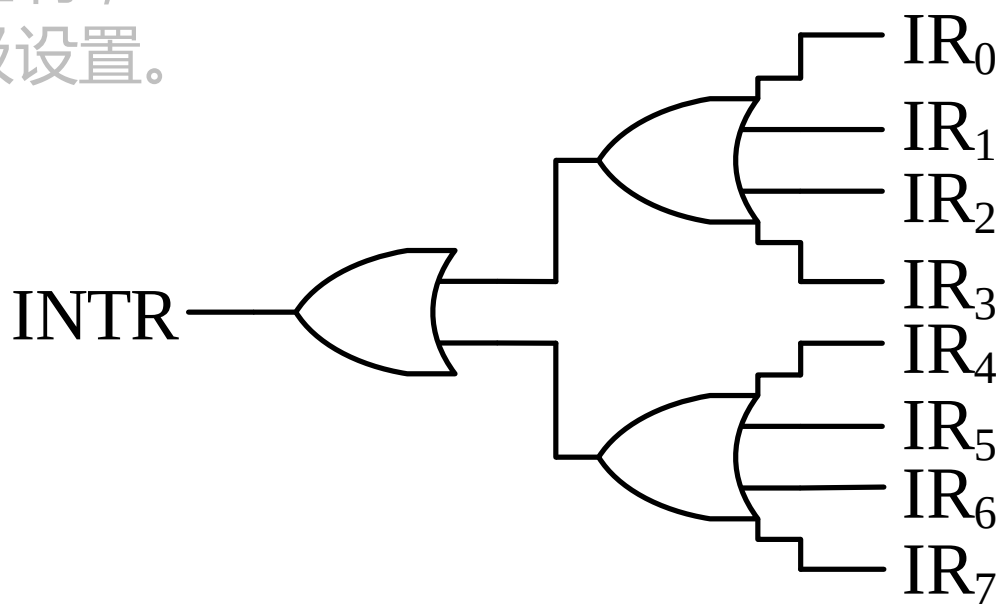
- 中断系统一般应具有以下功能：
  - 中断请求信号保持与清除
    - 带复位功能的锁存器/触发器
    - 写操作清除中断状态
  - 中断源识别，
  - 中断允许控制，
  - 中断优先级设置。



8.1 中断的基本概念

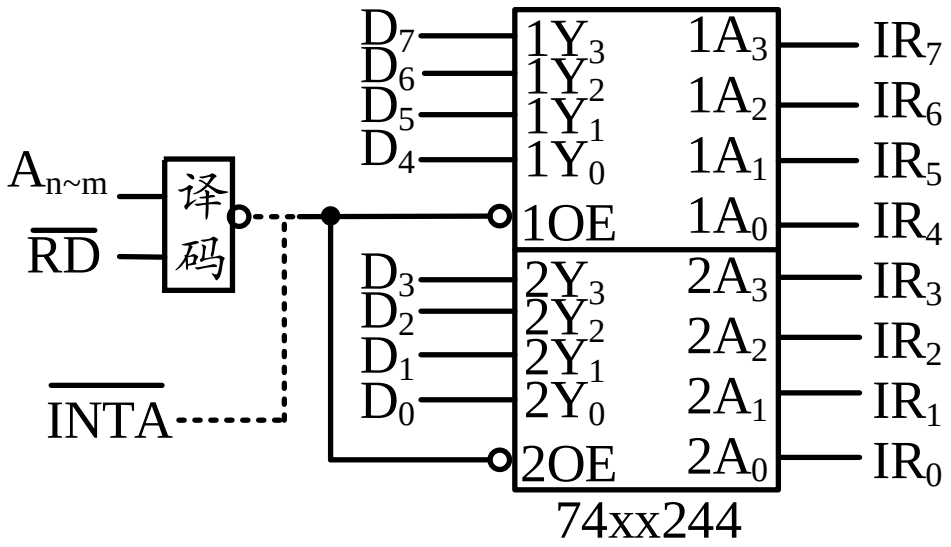
- 中断系统一般应具有以下功能：
  - 中断请求信号保持与清除，
  - 中断源识别
    - 多个中断请求信号合成
  - 中断允许控制，
  - 中断优先级设置。





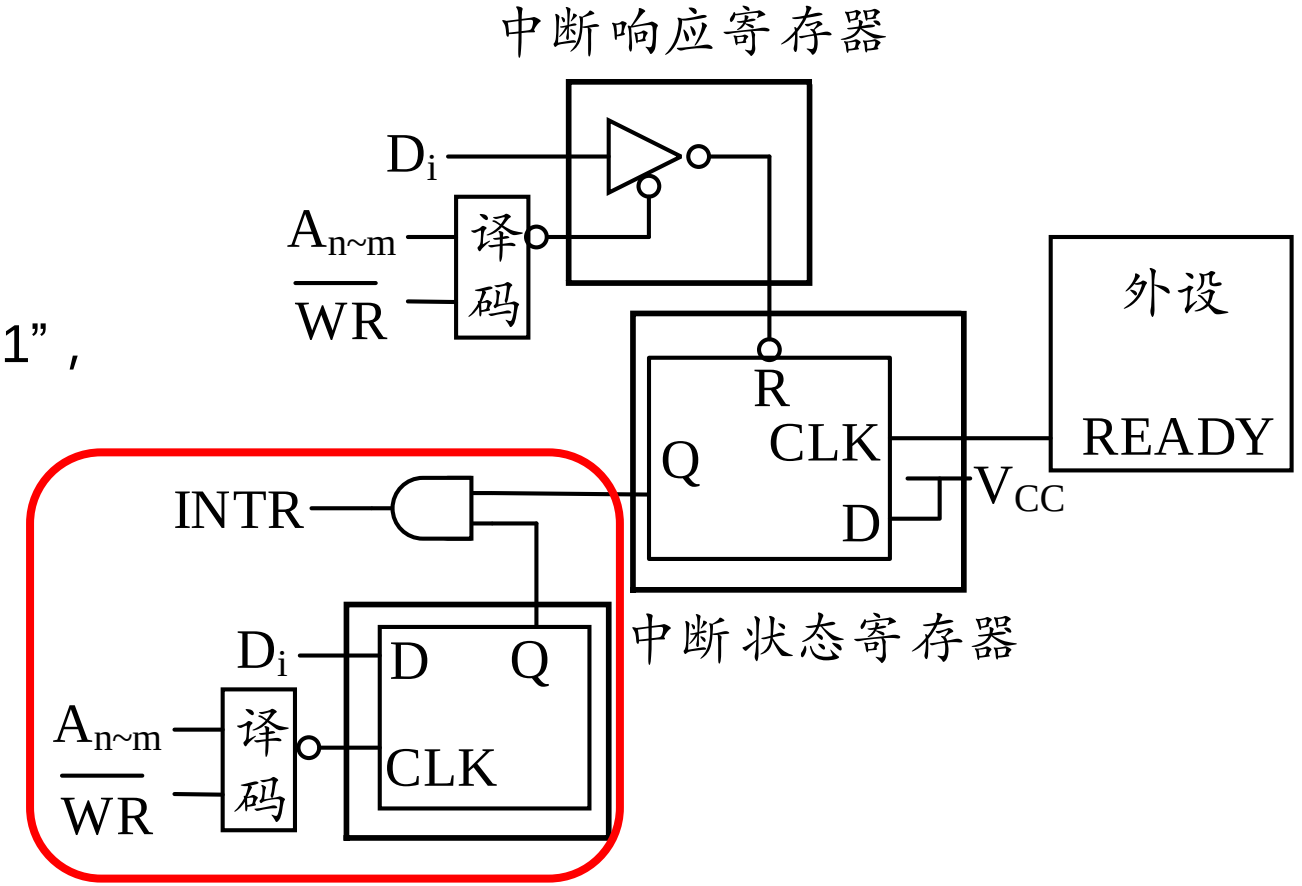
- 中断系统一般应具有以下功能：
  - 中断请求信号保持与清除，
  - 中断源识别
    - 多个中断源、多个中断类型码识别
  - 中断允许控制，
  - 中断优先级设置。

	中断类型码								
中断源	D7	D6	D5	D4	D3	D2	D1	D0	值
IR0	0	0	0	0	0	0	0	1	0x01
IR1	0	0	0	0	0	0	1	0	0x02
IR2	0	0	0	0	0	1	0	0	0x04
IR3	0	0	0	0	1	0	0	0	0x08
IR4	0	0	0	1	0	0	0	0	0x10
IR5	0	0	1	0	0	0	0	0	0x20
IR6	0	1	0	0	0	0	0	0	0x40
IR7	1	0	0	0	0	0	0	0	0x80



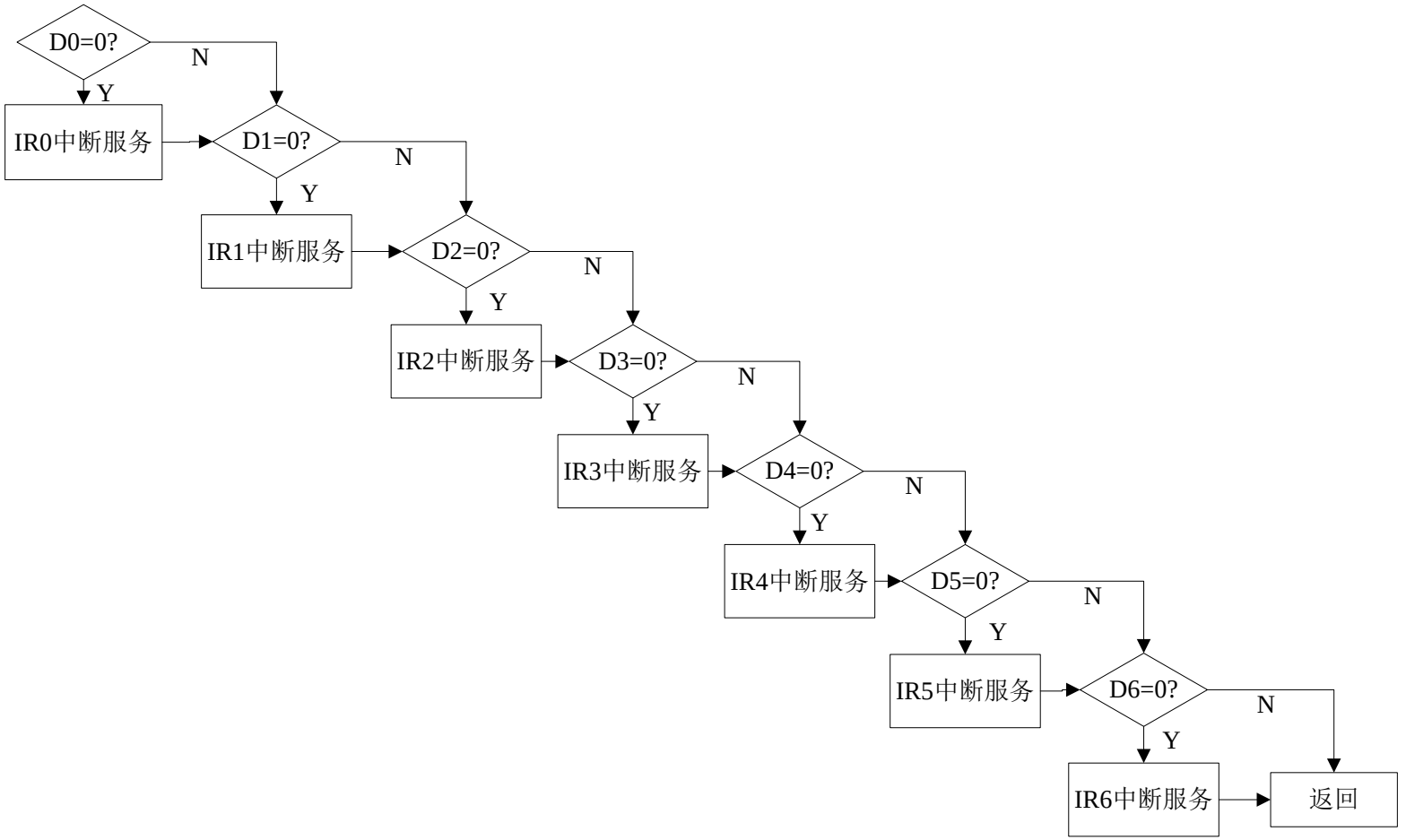
8.1 中断的基本概念

- 中断系统一般应具有以下功能：
  - 中断请求信号保持与清除，
  - 中断源识别，
  - 中断允许控制，
    - 模块的中断允许
    - CPU的中断允许
    - 只有当其（中断允许位）为“1”，即开中断，才会允许产生中断
  - 中断优先级设置。



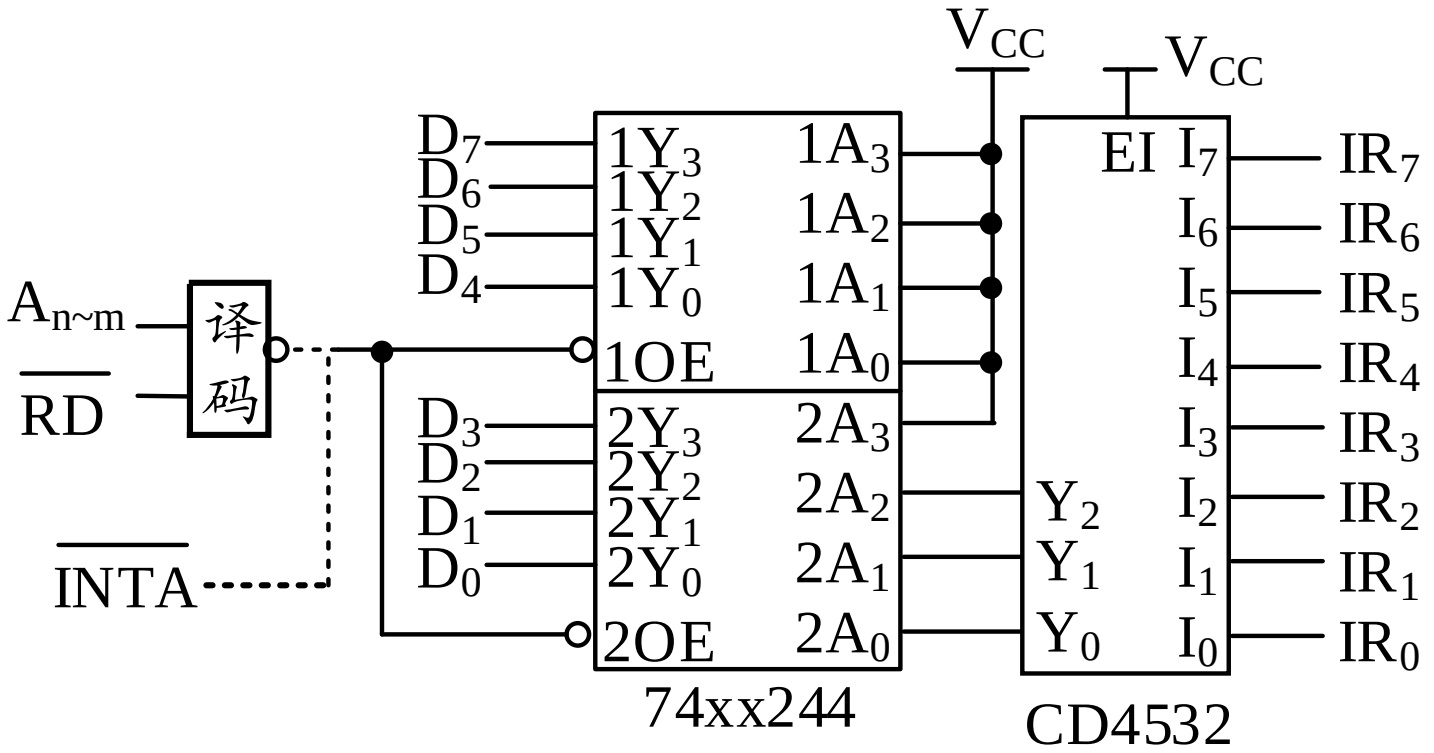
8.1 中断的基本概念

- 中断系统一般应具有以下功能：
  - 中断请求信号保持与清除，
  - 中断源识别，
  - 中断允许控制，
  - 中断优先级设置。
    - 软件查询顺序决定



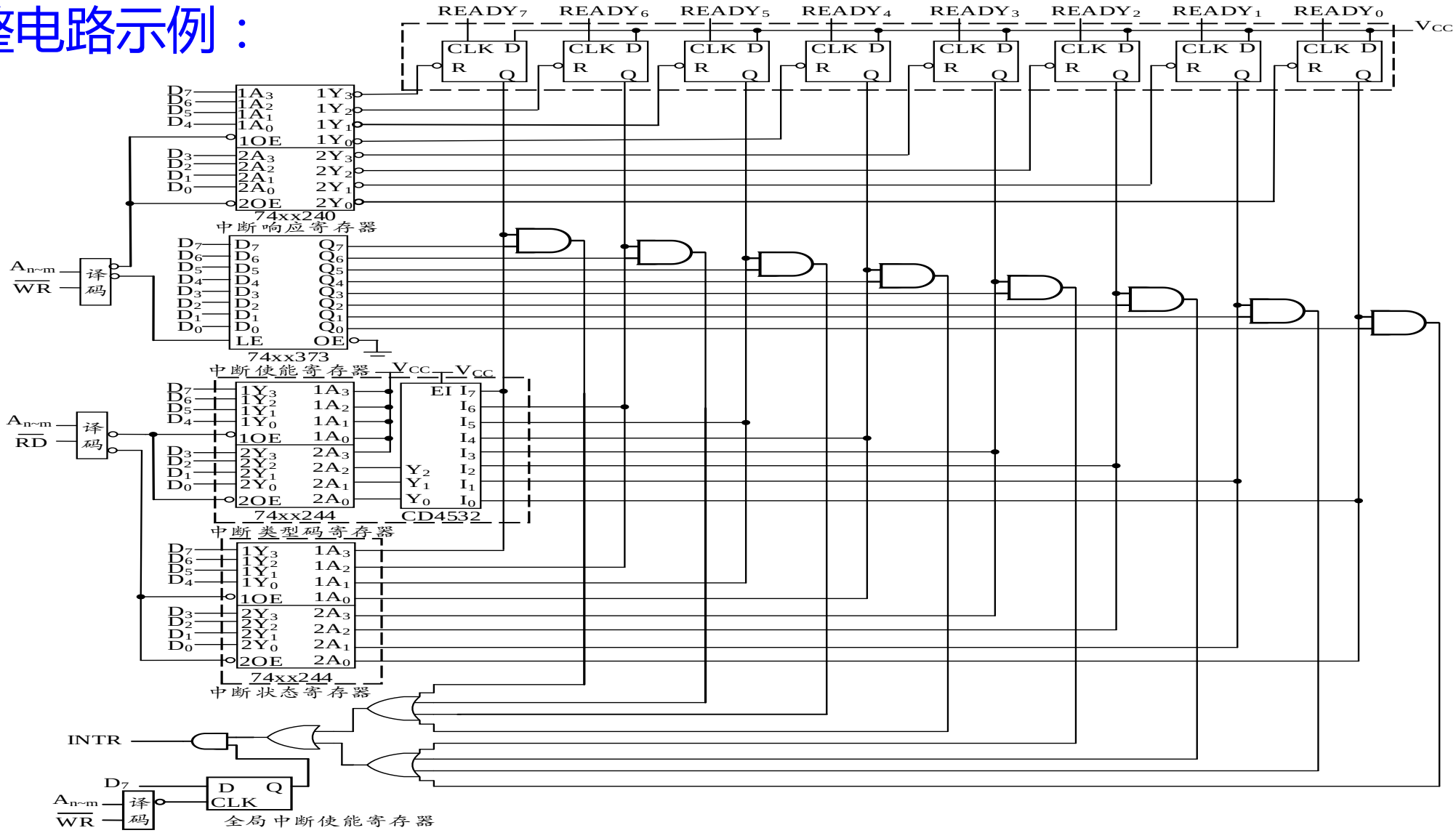
8.1 中断的基本概念

- 中断系统一般应具有以下功能：
  - 中断请求信号保持与清除，
  - 中断源识别，
  - 中断允许控制，
  - 中断优先级设置。
    - 硬件：优先编码器



8.1 中断的基本概念

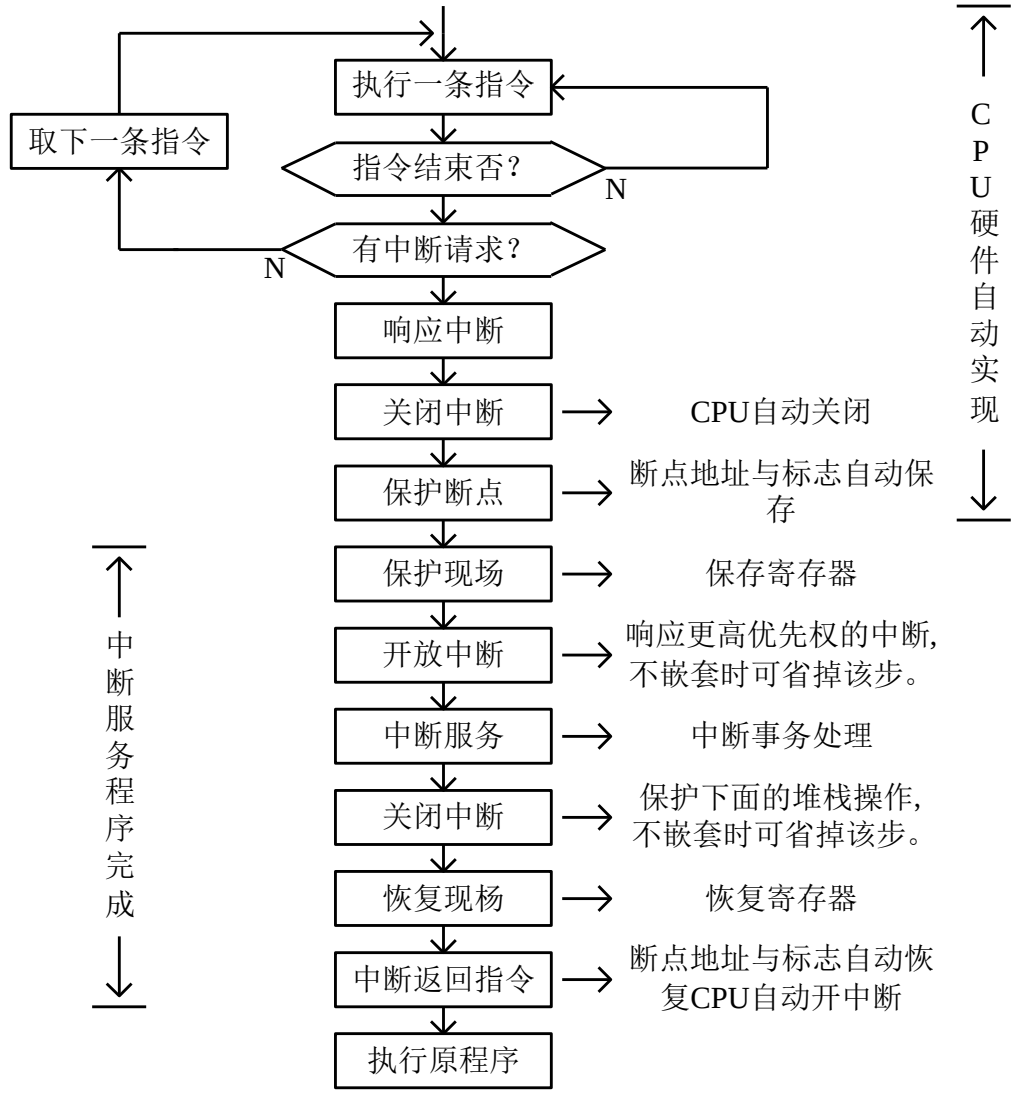
- 中断系统完整电路示例：



8.2 中断响应过程

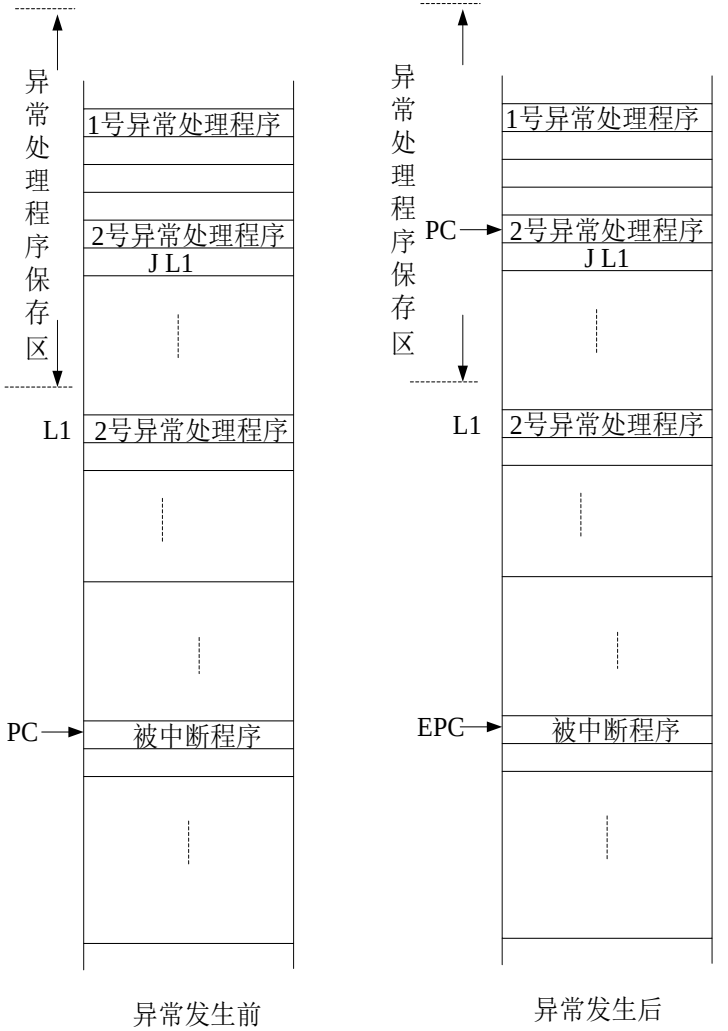
▶ 微处理器响应中断的一般过程

- CPU响应中断后，一个关键的问题是如何根据中断号/中断类型码得到相应的中断服务程序的入口地址，转向中断服务程序。



回顾：3.6 微处理器异常处理原理

- 异常处理程序进入方式
  - 1）专门的内存区域保存异常处理程序
    - 在这块内存区域中为每个异常处理程序分配固定长度的空间如32个字节或8条指令长度的空间，而且针对每个异常事件其异常处理程序的存放地址是固定的。
  - 2）仅提供一个异常处理程序存放地址
  - 3）分配一块专门的内存区域保存异常处理程序的入口地址



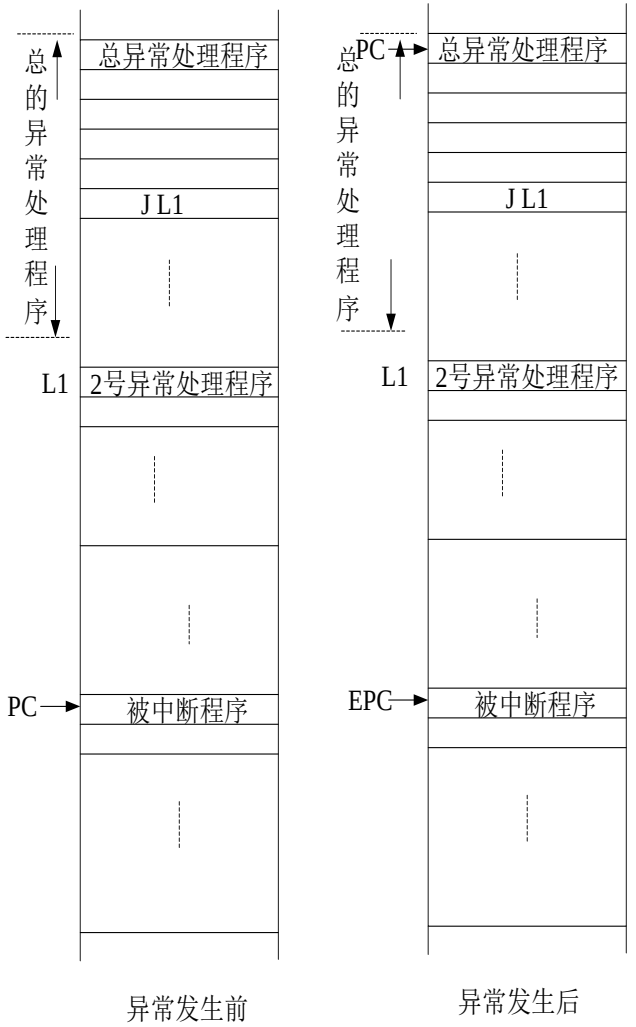
回顾： 3.6 微处理器异常处理原理

▶ 异常处理程序进入方式

- 1）专门的内存区域保存异常处理程序

- 2）仅提供一个异常处理程序存放地址
  - 发生任何异常事件都首先转移到该地址执行总的异常处理，并在总异常处理程序中分析异常事件的原因，然后再根据异常的原因通过子程序调用的方式去执行相应的异常处理。

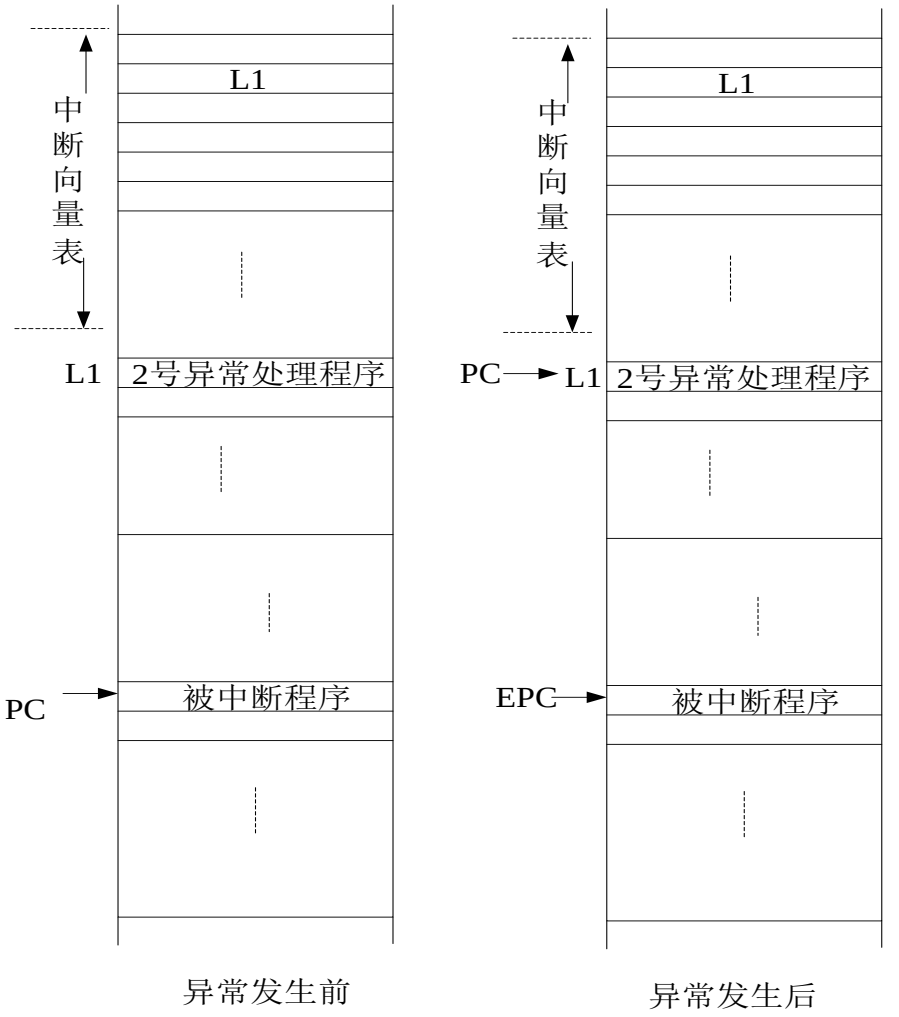
- 3）分配一块专门的内存区域保存异常处理程序的入口地址



回顾： 3.6 微处理器异常处理原理

▶ 异常处理程序进入方式

- 1）专门的内存区域保存异常处理程序
- 2）仅提供一个异常处理程序存放地址
- 3）分配一块专门的内存区域保存异常处理程序的入口地址
  - 异常处理程序的入口地址叫中断向量
  - 保存异常处理程序的入口地址的内存区域叫做中断向量表
  - 异常处理程序可以存放在内存中的任意位置，只需要把该异常处理程序的入口地址保存到中断向量表中正确的地址中，当异常发生时，微处理器就可以通过中断向量表查找到中断服务程序的入口地址。



Agenda

- 内容
  - 中断的基本概念，中断响应过程
  - 典型微处理器中断系统简介
  - Xilinx的中断控制器-AXI INTC
  - GPIO中断方式接口设计
  - AXI Timer接口
  - AXI SPI接口
- 目的
  - 理解Interrupt 的含义，优点、分类；
  - 理解中断源、中断请求、中断类型码、中断优先级、中断向量入口地址(Interrupt Vector Address)、中断向量表等术语的含义和作用；
  - 理解CPU响应中断的过程；
  - 理解X86和Microbalze系统的中断处理过程；
  - 掌握AXI INTC原理，学会Microbalze系统中断程序设计；
  - 掌握AXI Timer接口设计；
  - 掌握AXI SPI接口设计。

8.3 典型微处理器中断系统简介

- intel 80X86中断系统介绍
- MicroBlaze中断系统介绍

8.3 典型微处理器中断系统简介

- intel 80X86中断系统介绍
  - 80x86的中断类型码及中断种类
    - 中断优先权次序为：内部中断(单步中断除外)优先权最高、其次是NMI、再次是INTR，优先权最低的是内部中断中的单步中断
  - x86获取中断服务程序入口地址的方法
    - 实地址方式使用中断向量表，
    - 虚地址保护方式使用中断描述符表。

中断类型码	中断种类
0	除法错误中断
1	单步中断
2	非屏蔽中断
3	断点中断
4	INTO指令溢出中断
5	越界(超出了BOUND范围)中断
6	非法操作码中断
7	浮点单元不可用中断
8	双重故障中断
9	保留
10	无效任务状态段中断
11	段不存在中断
12	堆栈异常中断
13	一般保护中断
14	页故障中断
15	保留
16	浮点错误中断
17	对准检查中断
18 ~ 31	保留
32 ~ 255	INT N指令中断和INTR可屏蔽中断

8.3 典型微处理器中断系统简介

- intel 80X86中断系统介绍
  - 80x86的可屏蔽中断INTR
    - 现代PC系统中使用两片中断控制器8259A管理IRQ0～IRQ7（主片）、IRQ8 ～ IRQ15 （从片）共16个外部中断源。
      - IRQ0～7的中断类型码为：08H～0FH；
      - IRQ8～15的中断类型码为：70H ～ 77H

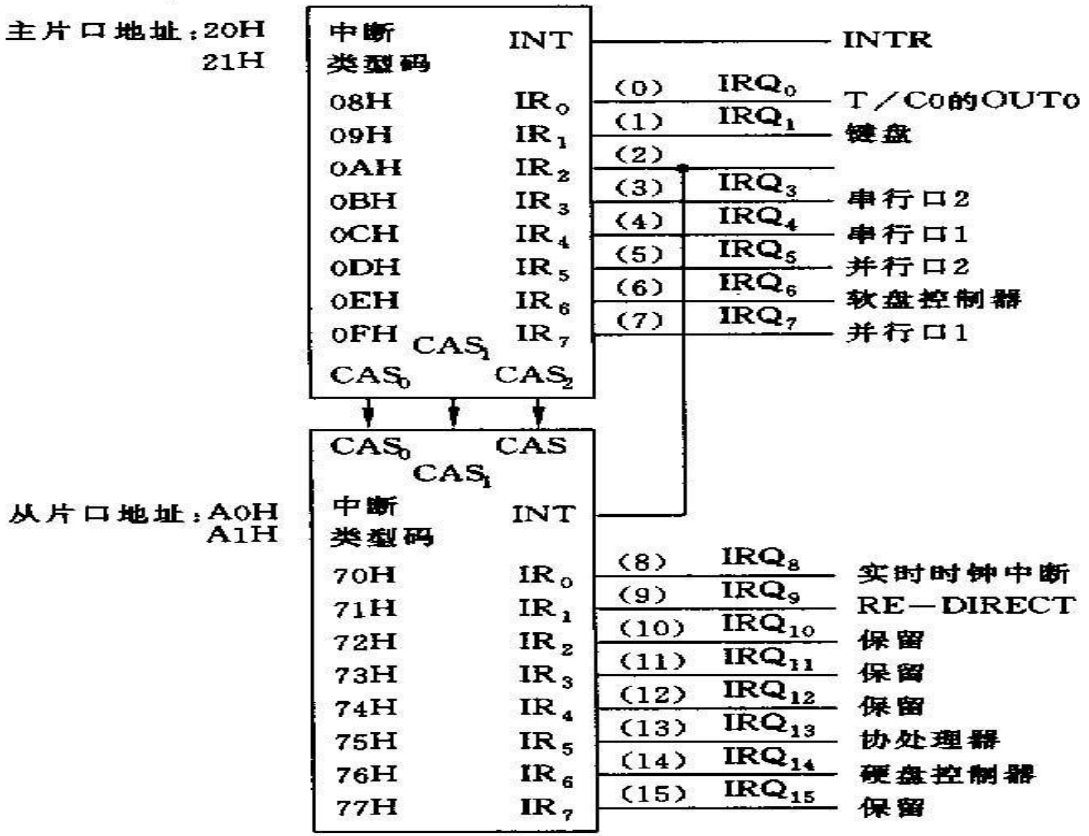
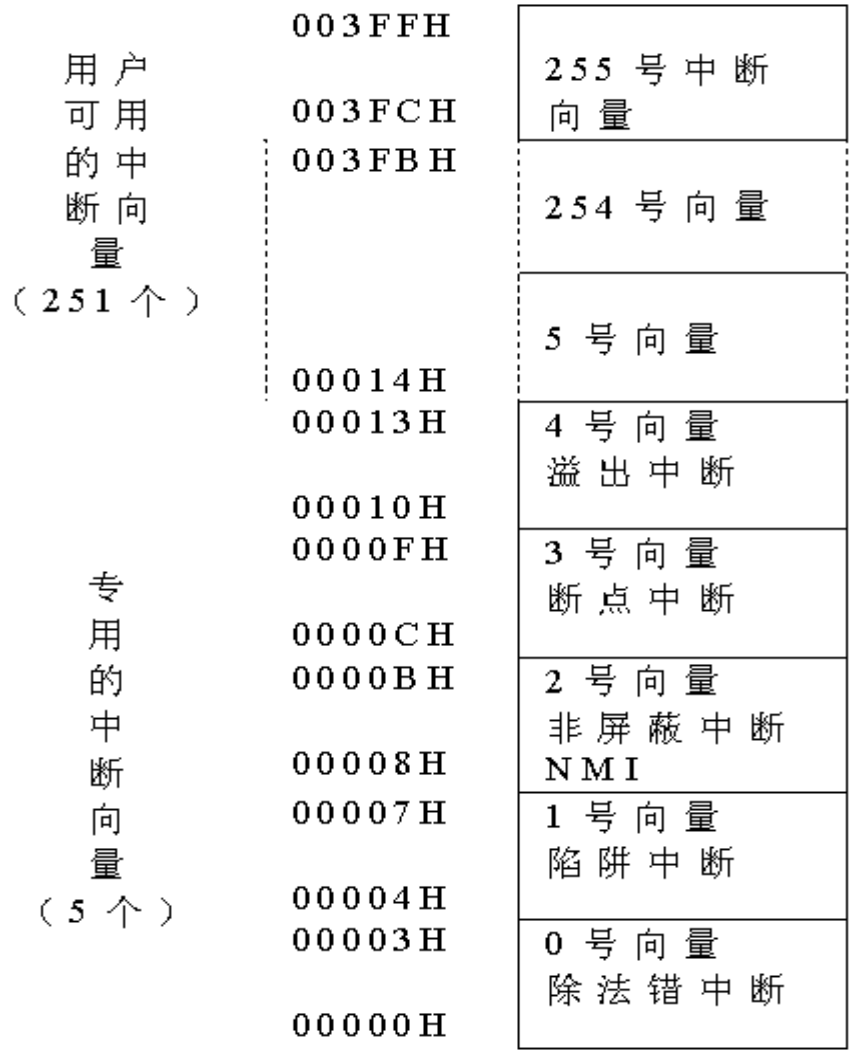


图 7-12　80x86 PC 机的硬中断结构

8.3 典型微处理器中断系统简介

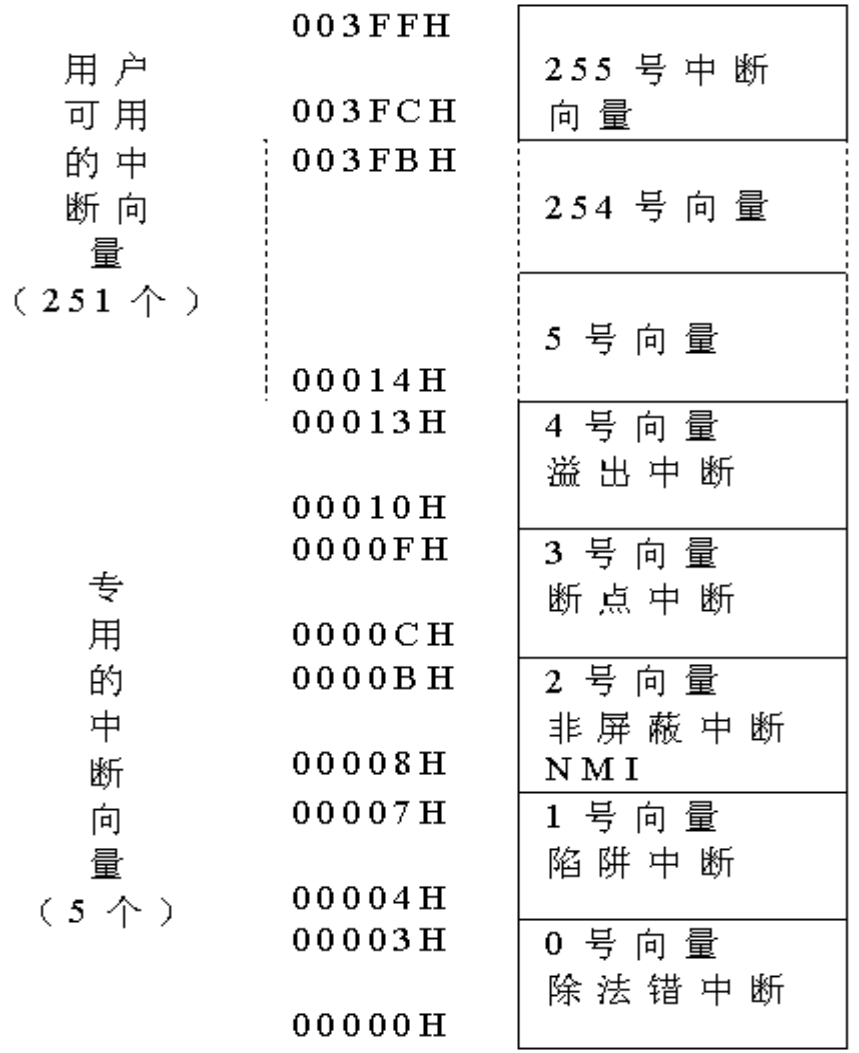
- intel 80X86中断系统介绍
  - 实模式下的中断向量表
    - CPU响应中断后，一个关键的问题是如何根据中断号得到相应的中断服务程序的入口地址(Interrupt Vector Address)，转向中断服务程序。
    - 80X86系统采用的办法是将256种类型的中断服务子程序建立一张中断服务程序入口地址表——中断向量表。该中断向量表安排在内存的前1KB，即00000H~003FFH；
    - 每一个中断服务子程序的入口地址CS:IP占用4个字节存储单元。
    - 一个中断类型码n所对应的中断向量表占有4n、4n+1和4n+2、4n+3四个字节单元或4n和4n+2两个半字单元。每个中断向量表规定：低半字存放偏移地址(IP)，高半字存放段地址(CS)，按中断号顺序存放。



8.3 典型微处理器中断系统简介

▶ intel 80X86中断系统介绍

- 实模式下的中断向量表
  - 每一个中断服务子程序的入口地址CS:IP占用4个字节存储单元。
  - 一个中断类型码n所对应的中断向量表占有4n、4n+1和4n+2、4n+3四个字节单元或4n和4n+2两个半字单元。每个中断向量表规定：低半字存放偏移地址(IP)，高半字存放段地址(CS)，按中断号顺序存放。
    - 【例】若80x86系统采用的8259A的中断类型码为71H，试问中断服务程序的地址填入哪个半字单元？
    - 中断服务程序偏移地址填入4n半字单元，而4×71H=01C4H，故填入001C4H半字单元。
    - 中断服务程序段地址填入4n+2半字单元，而4×41H+2=01C6H，故填入001C6H半字单元。

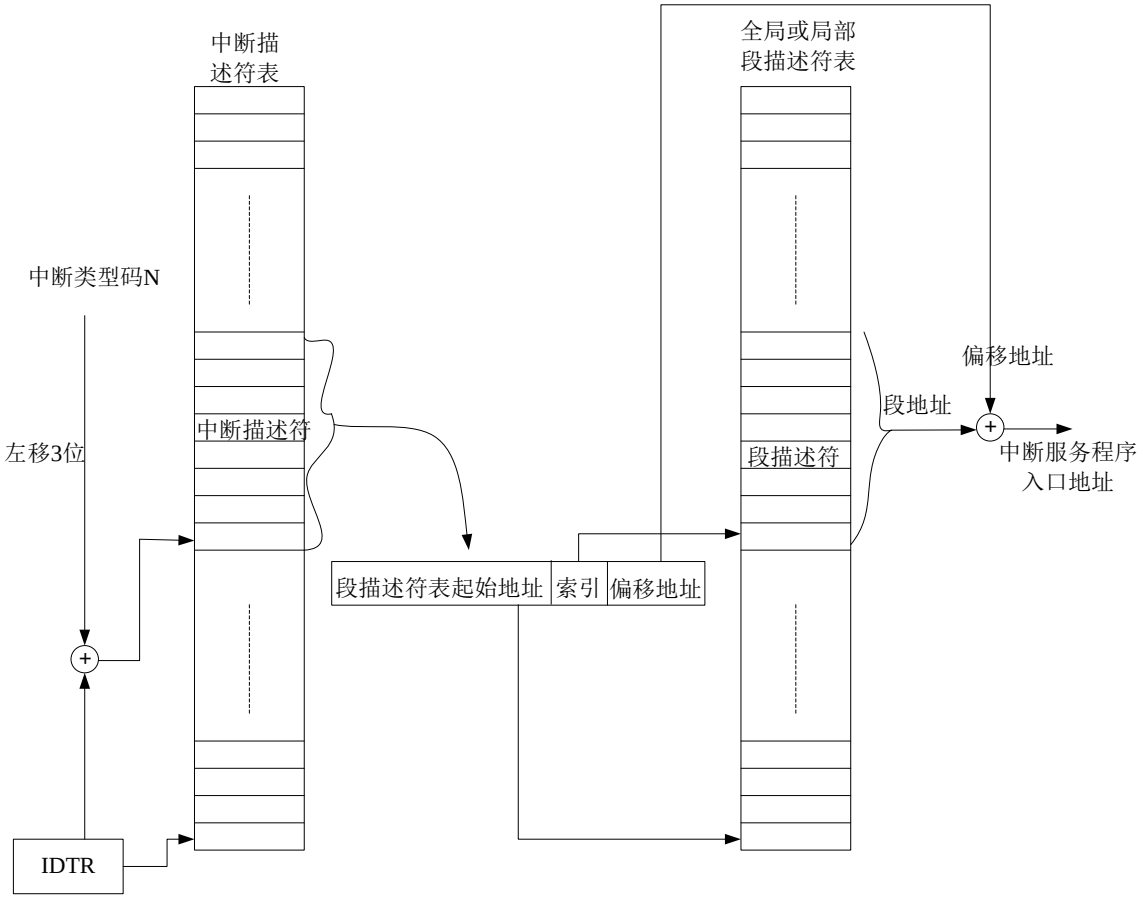


8.3 典型微处理器中断系统简介

► intel 80X86中断系统介绍

- 保护模式下的中断描述符表
  - 中断描述符表最多可包含256个中断描述符，每个中断描述符为8字节，中断描述符表长为8×256=2K字节，
  - 中断描述符表在内存中存放的起始地址由中断描述符表地址寄存器IDTR指定。IDTR是一个48位的寄存器，它的高32位保存中断描述符表的基地址，低16位保存中断描述符表的界限值即表长度。
  - 中断描述符包含3个内容，一是描述符索引DI，由此可以获得段基址等；二是32位的偏移地址；三是相关段的参数，这些参数指示引起中断的原因属于哪一类。

- 保护模式下中断服务程序入口地址获取过程



8.3 典型微处理器中断系统简介

▶ MicroBlaze中断系统介绍

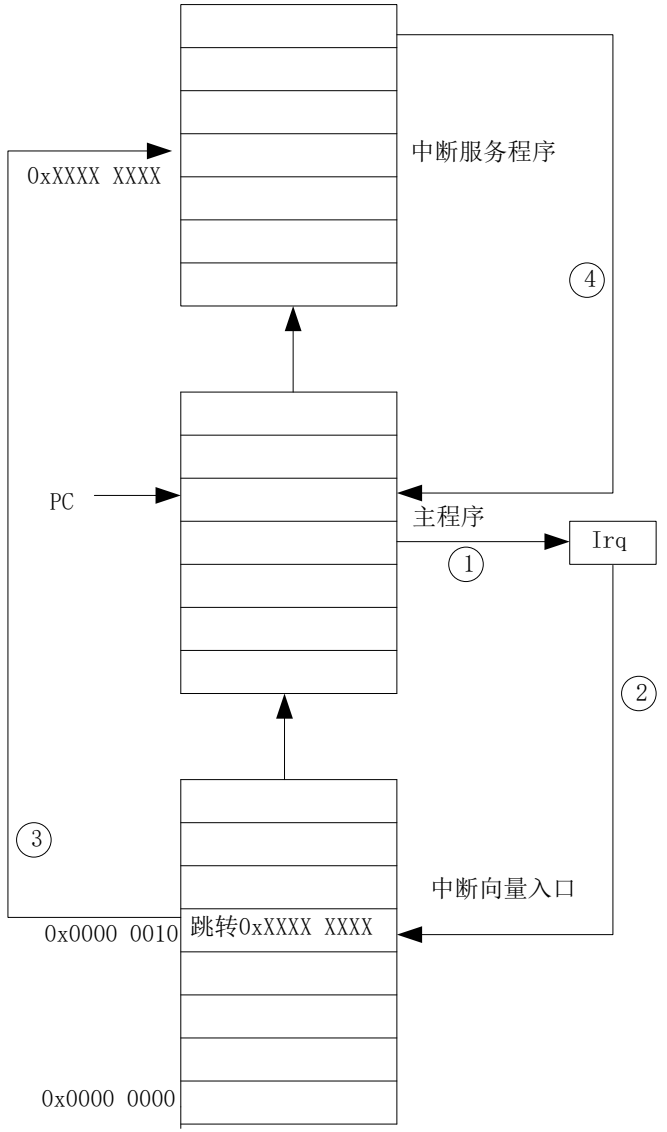
- 软核处理器
- 中断类型

中断类型	中断向量地址	保存断点的寄存器	中断优先级
复位	0x00000000-0x00000004	-	1
用户异常	0x00000008-0x0000000C	-	7
中断	0x00000010-0x00000013	R14	6
不可屏蔽硬件打断	0x00000018-0x0000001C	R16	3
硬件打断（break）			4
软件打断（break）			5
硬件异常	0x00000020-0x00000024	R17	2

- 中断向量的地址都是固定的，微处理器响应某个特定类型的中断时，不需要查询中断向量表，就可以直接根据固定的中断向量地址进入中断服务程序；
  - 中断向量预留的只有8个字节存储空间，不足以保存中断服务程序；
    - ▶4个字节保存真正的中断服务程序入口地址，4个字节保存间接寻址跳转指令的机器码
  - 真正的中断服务程序不是保存在中断向量地址处，而是在内存其它位置；中断向量地址处仅通过一条跳转指令转入真正的中断服务程序
- 保存断点的方式也是有别于X86处理器，不是利用堆栈，而是直接利用寄存器保存断点

8.3 典型微处理器中断系统简介

- MicroBlaze中断系统介绍
  - MicroBlaze中断处理过程
    - 微处理器对所有外设仅提供一个中断服务程序跳转地址（0x0000 0010），因此该中断服务程序也叫做主中断服务程序。
    - 如果系统需要支持多个外设中断请求
      - 硬件上就需要通过一个中断控制器如AXI INTC对多个中断信号进行管理；
      - 软件上，需要在主中断服务程序中读取中断控制器的中断请求寄存器，以确定是哪个中断源发生了中断请求，并且需要进一步调用针对该中断源的中断服务程序。因此主中断服务程序需要维护一个中断向量表（中断向量数组），并根据中断源查找相应中断服务程序



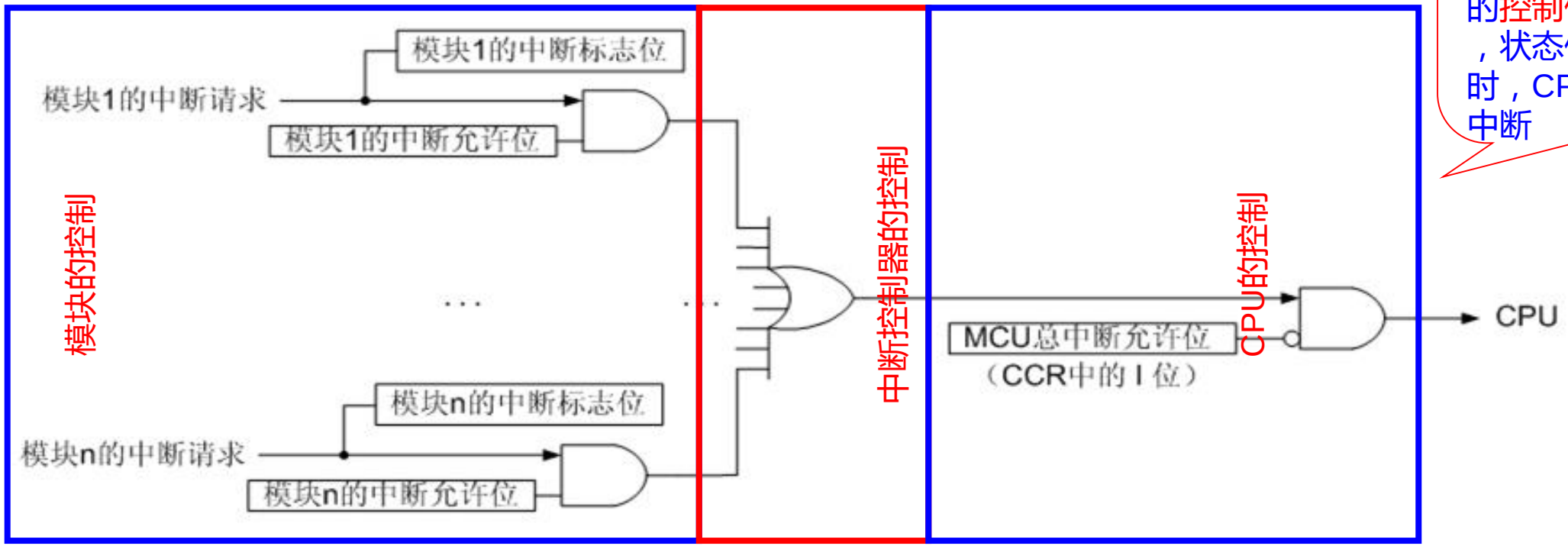
回顾：8.1 中断的基本概念

▶ 中断模型

- CPU是否能够收到可屏蔽中断，一般受三个控制位控制
  - 模块的中断控制
  - 中断控制器的控制
  - CPU的控制

80X86系统中用中断控制器8259A多多个外部中断进行管理
Microblaze系统中用中断控制器AXI INTC对多个中断源进行管理

MicroBlaze有一个机器状态寄存器MSR，该寄存器中的控制位IE位为1，状态位BIP位为0时，CPU可以响应中断



8.3 典型微处理器中断系统简介

▶ MicroBlaze中断系统介绍

- standalone操作系统中断相关系统调用
  - MicroBlaze微处理器中断系统调用
    - void microblaze_enable_interrupts(void)
      - ▶该函数的功能是使得MSR中的IE位为1，从而MicroBlaze微处理器可以响应外部中断。
    - void microblaze_disable_interrupts(void)
      - ▶该函数的功能是使得MSR中的IE位为0，从而MicroBlaze微处理器不响应外部中断。
    - void microblaze_register_handler(XInterruptHandler Handler, void *DataPtr)
      - ▶该函数的功能是将中断控制器主中断服务程序的地址与跳转指令结合后填入中断向量地址0x00000010处，并且将中断服务程序需要处理的参数地址传给中断服务程序。

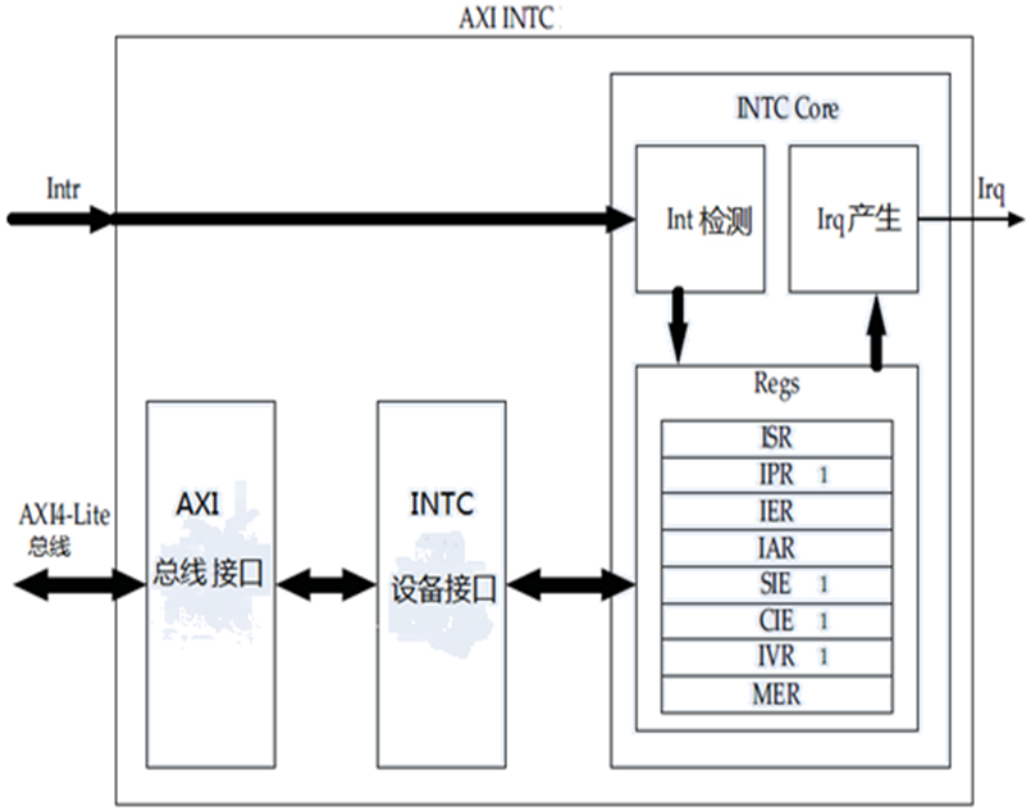
Agenda

- 内容
  - 中断的基本概念，中断响应过程
  - 典型微处理器中断系统简介
  - Xilinx的中断控制器-AXI INTC
  - GPIO中断方式接口设计
  - AXI Timer接口
  - AXI SPI接口
- 目的
  - 理解Interrupt 的含义，优点、分类；
  - 理解中断源、中断请求、中断类型码、中断优先级、中断向量入口地址(Interrupt Vector Address)、中断向量表等术语的含义和作用；
  - 理解CPU响应中断的过程；
  - 理解X86和Microbalze系统的中断处理过程；
  - 掌握AXI INTC原理，学会Microbalze系统中断程序设计；
  - 掌握AXI Timer接口设计；
  - 掌握AXI SPI接口设计。

8.4 Xilinx的中断控制器-AXI INTC

- 中断控制器AXI INTC特征
  - 支持32个中断源输入Intr[31:0] ，每个中断源都可以配置为4种中断触发方式中的任意一种
  - 一个中断请求信号输出Irq，可配置为4种中断触发方式中的任意一种
  - 可以级联
  - 中断请求输入端Intr[31:0]的优先级根据所处位置决定，bit0具有最高优先级，bit31优先级最低
  - 每个中断源可以单独屏蔽或开放，也可以同时屏蔽所有中断源

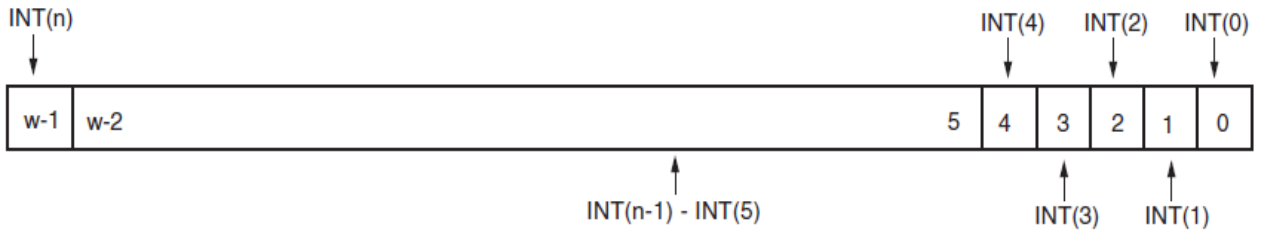
——所有功能配置通过AXI INTC寄存器

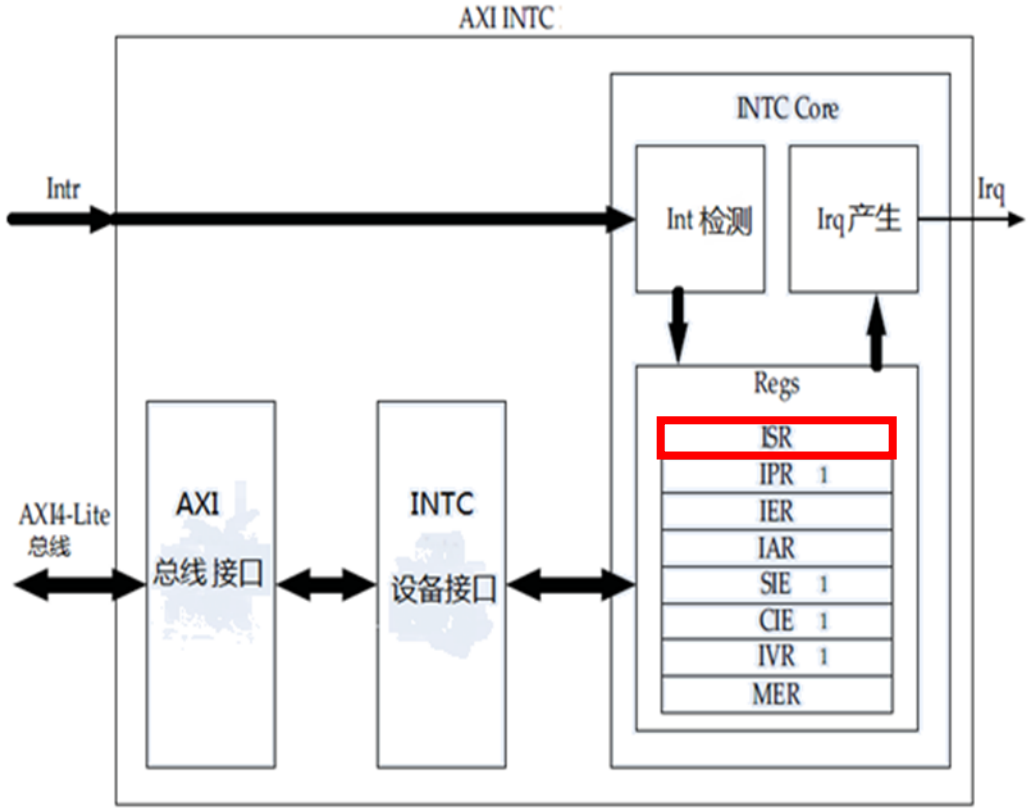


8.4 Xilinx的中断控制器-AXI INTC

▶ AXI INTC的寄存器

寄存器名称	偏移地址	允许操作	初始值	含义
ISR	0x0	Read / Write	0x0	中断请求状态寄存器
IPR（可选）	0x4	Read	0x0	中断悬挂寄存器
IER	0x8	Read / Write	0x0	中断屏蔽寄存器
IAR	0xC	Write	0x0	中断响应寄存器
SIE（可选）	0x10	Write	0x0	中断允许设置寄存器
CIE（可选）	0x14	Write	0x0	中断允许清除寄存器
IVR（可选）	0x18	Read	0x0	中断类型码寄存器
MER	0x1C	Read / Write	0x0	主中断屏蔽寄存器



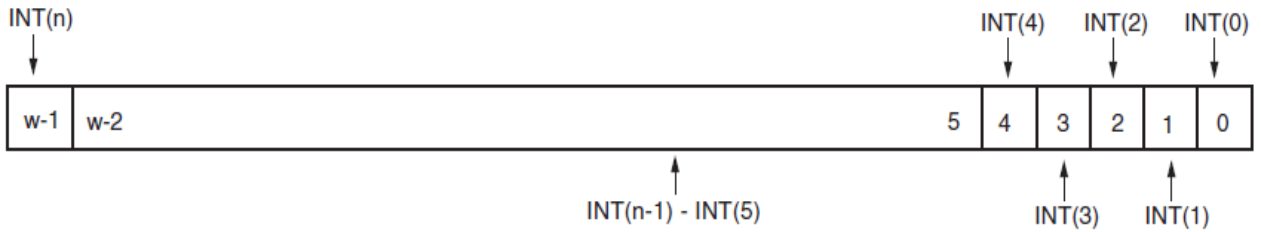


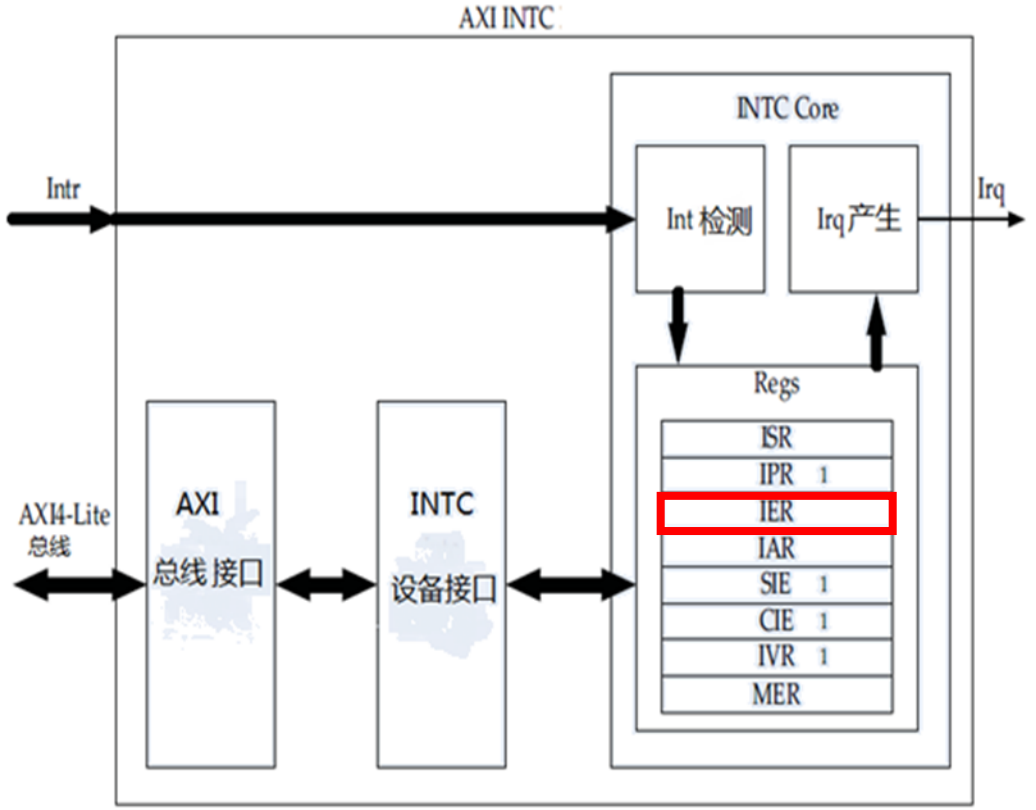
ISR：Intr[31:0]输入线上有中断请求，则ISR[31:0]对应位为1。软件中通过查询ISR[31:0]为1的位可以获知是哪个中断源产生了中断。

8.4 Xilinx的中断控制器-AXI INTC

▶ AXI INTC的寄存器

寄存器名称	偏移地址	允许操作	初始值	含义
ISR	0x0	Read / Write	0x0	中断请求状态寄存器
IPR（可选）	0x4	Read	0x0	中断悬挂寄存器
IER	0x8	Read / Write	0x0	中断屏蔽寄存器
IAR	0xC	Write	0x0	中断响应寄存器
SIE（可选）	0x10	Write	0x0	中断允许设置寄存器
CIE（可选）	0x14	Write	0x0	中断允许清除寄存器
IVR（可选）	0x18	Read	0x0	中断类型码寄存器
MER	0x1C	Read / Write	0x0	主中断屏蔽寄存器



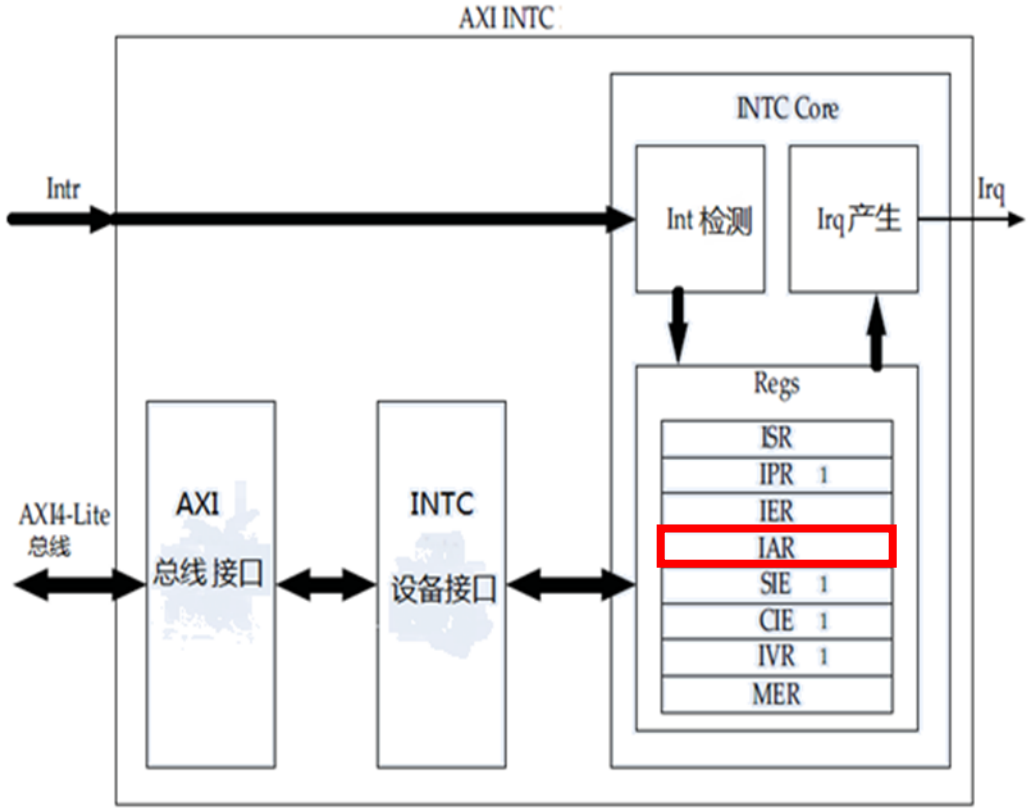


IER：控制ISR[31:0]为1的位是否产生Irq请求
1：允许产生Irq请求
0：禁止产生Irq请求

8.4 Xilinx的中断控制器-AXI INTC

▶ AXI INTC的寄存器

寄存器名称	偏移地址	允许操作	初始值	含义
ISR	0x0	Read / Write	0x0	中断请求状态寄存器
IPR（可选）	0x4	Read	0x0	中断悬挂寄存器
IER	0x8	Read / Write	0x0	中断屏蔽寄存器
IAR	0xC	Write	0x0	中断响应寄存器
SIE（可选）	0x10	Write	0x0	中断允许设置寄存器
CIE（可选）	0x14	Write	0x0	中断允许清除寄存器
IVR（可选）	0x18	Read	0x0	中断类型码寄存器
MER	0x1C	Read / Write	0x0	主中断屏蔽寄存器

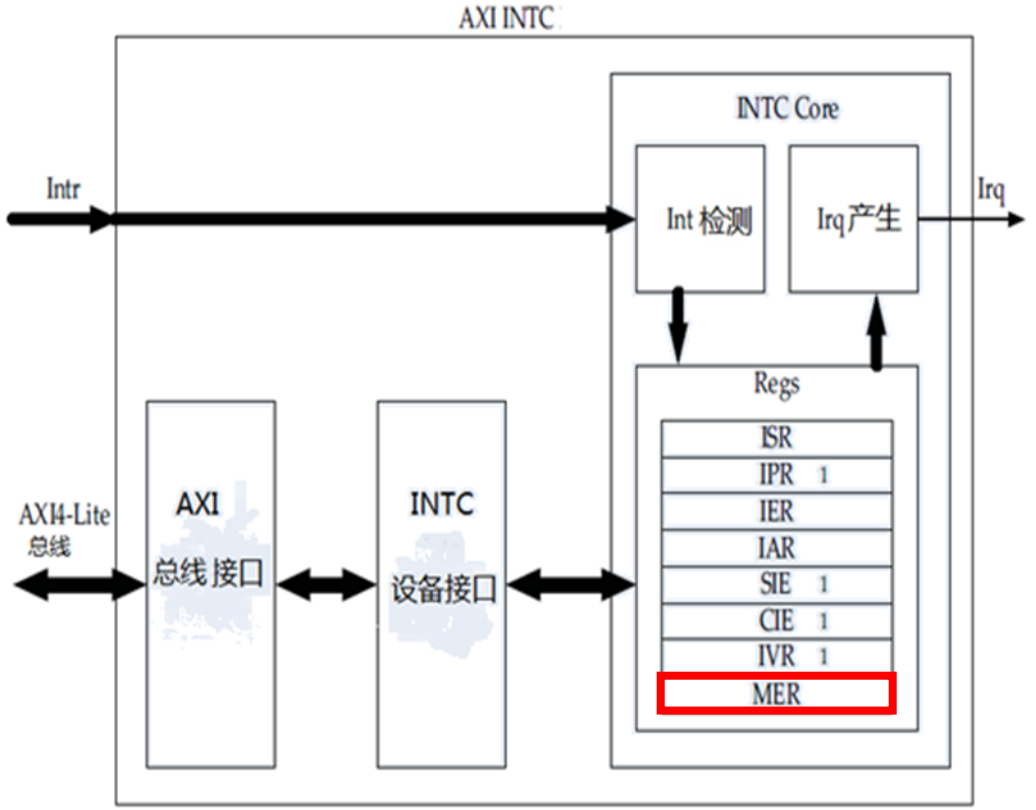


IAR：只写寄存器，通过向该寄存器的某一位写1，来清零ISR[31:0]中对应为1的位。在通过IER开放某位中断之前，应该先清零ISR中的标志位。在中断服务程序中，响应完中断之后，一定要写IAR相关的位为1来清零对应的ISR中的中断标志位。

8.4 Xilinx的中断控制器-AXI INTC

▶ AXI INTC的寄存器

寄存器名称	偏移地址	允许操作	初始值	含义
ISR	0x0	Read / Write	0x0	中断请求状态寄存器
IPR（可选）	0x4	Read	0x0	中断悬挂寄存器
IER	0x8	Read / Write	0x0	中断屏蔽寄存器
IAR	0xC	Write	0x0	中断响应寄存器
SIE（可选）	0x10	Write	0x0	中断允许设置寄存器
CIE（可选）	0x14	Write	0x0	中断允许清除寄存器
IVR（可选）	0x18	Read	0x0	中断类型码寄存器
MER	0x1C	Read / Write	0x0	主中断屏蔽寄存器

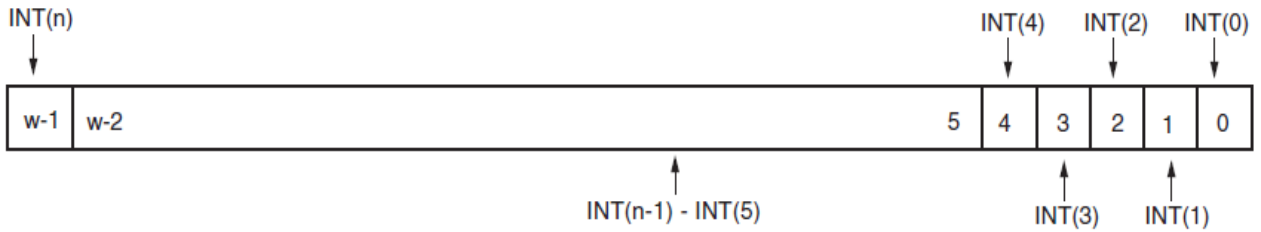


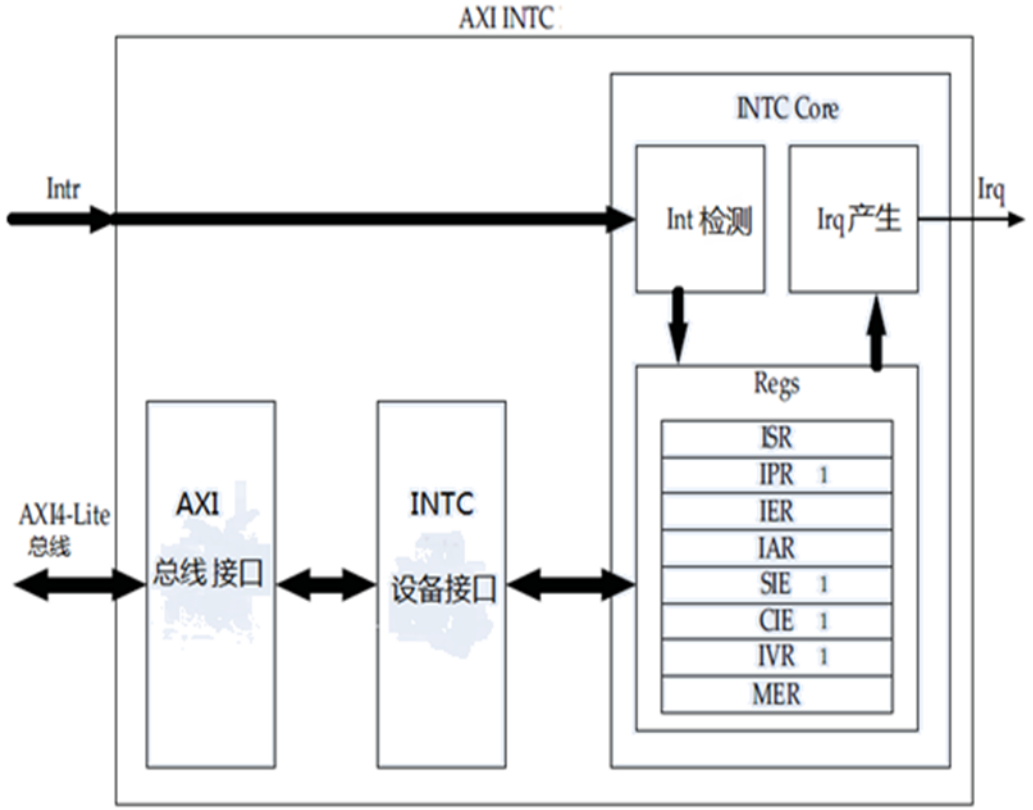
MER：控制INTC模块的允许和禁止
HIE=1：允许INTC模块接收Intr[31:0]硬件中断
ME=1：允许rq请求向CPU产生中断请求

8.4 Xilinx的中断控制器-AXI INTC

▶ AXI INTC的寄存器

寄存器名称	偏移地址	允许操作	初始值	含义
ISR	0x0	Read / Write	0x0	中断请求状态寄存器
IPR（可选）	0x4	Read	0x0	中断悬挂寄存器
IER	0x8	Read / Write	0x0	中断屏蔽寄存器
IAR	0xC	Write	0x0	中断响应寄存器
SIE（可选）	0x10	Write	0x0	中断允许设置寄存器
CIE（可选）	0x14	Write	0x0	中断允许清除寄存器
IVR（可选）	0x18	Read	0x0	中断类型码寄存器
MER	0x1C	Read / Write	0x0	主中断屏蔽寄存器



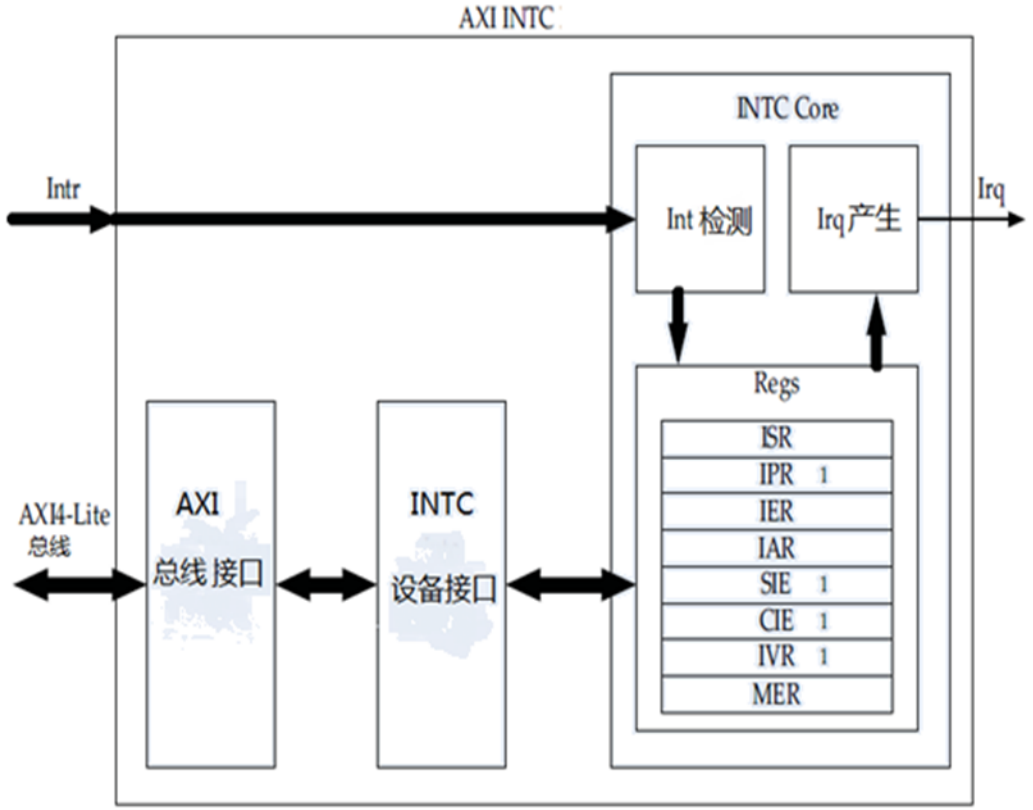


ISRIPR,SIE,CIE, IVR：
可选，设定额外功能。

8.4 Xilinx的中断控制器-AXI INTC

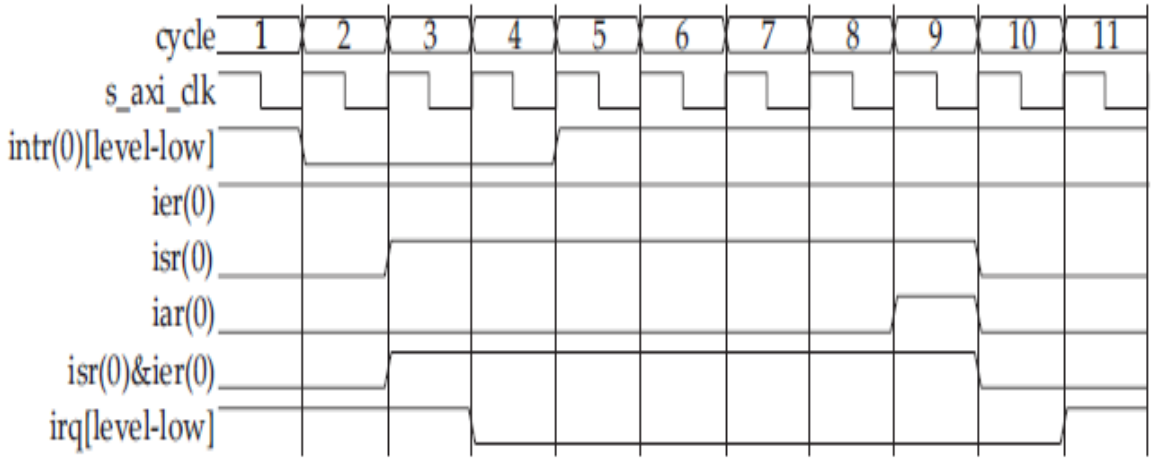
AXI INTC中断处理过程

- (1) 在中断请求输入端Intr上接受中断请求。
  - 前提是MER中的HIE=1
- (2) 中断请求锁存在ISR中，并与IER相“与”，若使用优先级判断电路，那么将未屏蔽的中断送给优先级判定电路。
- (3) 控制逻辑接受中断请求，输出Irq信号。
  - 前提是IER中对应的控制位=1
- (4) 若使用优先级判断电路，优先级判定电路检出优先级最高的中断请求位，并将IVR设置为相应的值。
- (5) 进入微处理器中断响应过程。若使用优先级判断电路，微处理器读取IVR识别当前优先级最高的中断请求源。若没有使用优先级判断电路，微处理器就需要读取ISR，识别产生中断的请求源。
  - 前提是MER中的ME=1
- (6)微处理器向中断响应寄存器（IAR）对应的位写入1，使ISR相应位复位从而结束中断。

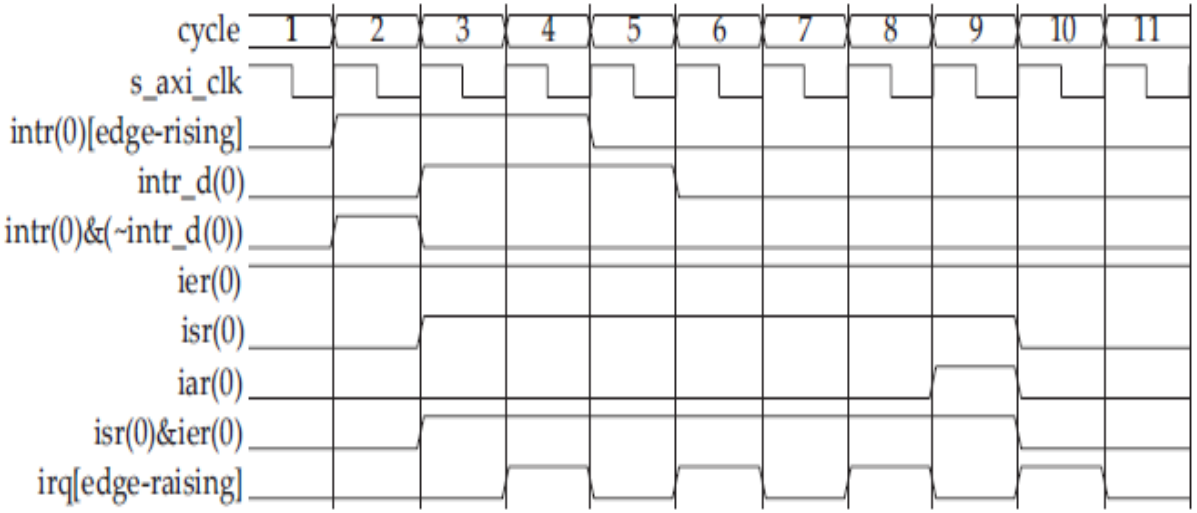


8.4 Xilinx的中断控制器-AXI INTC

- 中断信号时序(4*4=16种)
  - Intr和Irq都为低电平中断触发方式



  - Intr和Irq都为上升沿中断触发方式



8.4 Xilinx的中断控制器-AXI INTC

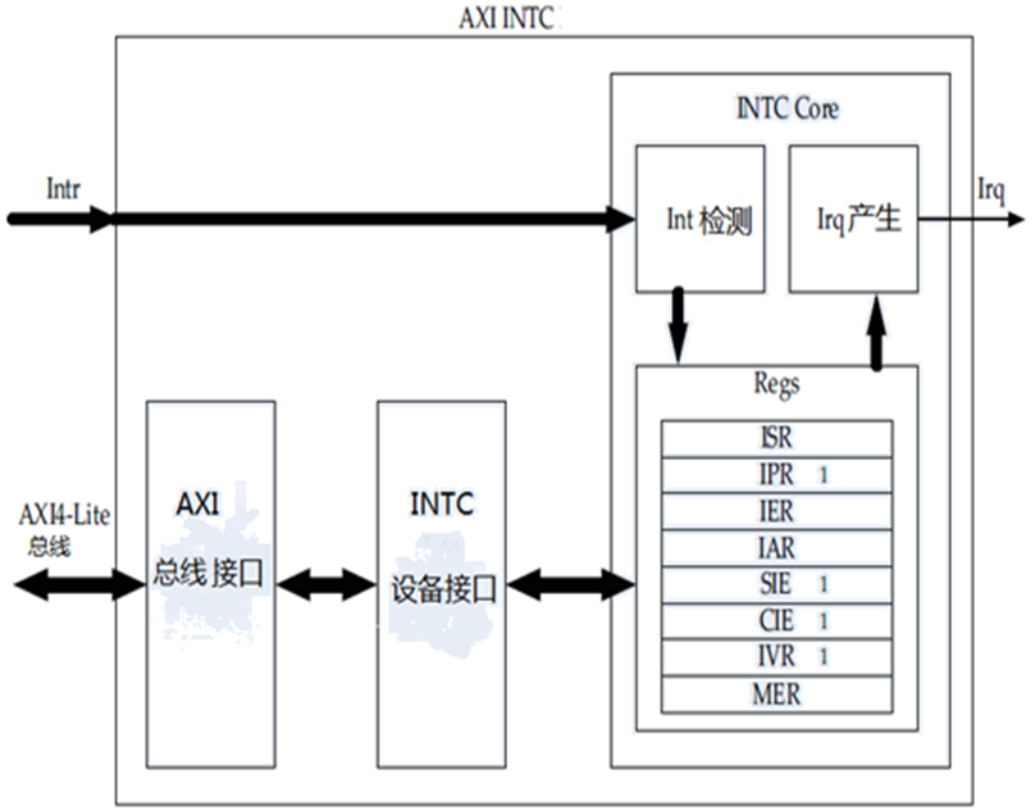
▶ AXI INTC的编程控制

- 初始化，通常包括两个方面：
  - 修改MER，使得HIE和ME都为1；
  - 修改IER，使得连接了中断请求源的相应位为1，允许中断请求
- 中断结束
  - 写IAR，使服务了的中断请求位清零

某小字节序计算机系统利用AXI INTC作为中断控制器，该中断系统可以接受5个中断源的中断申请，分别将它们连接到中断请求端Intr4~0，请编写对AXI INTC进行初始化的程序段。假定AXI INTC的基地址为0x80000000。

IER的地址为0x80000008，MER的地址为0x8000001c。
允许Intr4~0的中断请求，在小字节序的计算机系统中，IER的值为0x0000001f。MER的值为0x00000003。采用Xilinx C语言控制的程序段为：

```
Xil_Out32(0x80000008, 0x0000001f);
Xil_Out32(0x8000001c, 0x00000003);
```

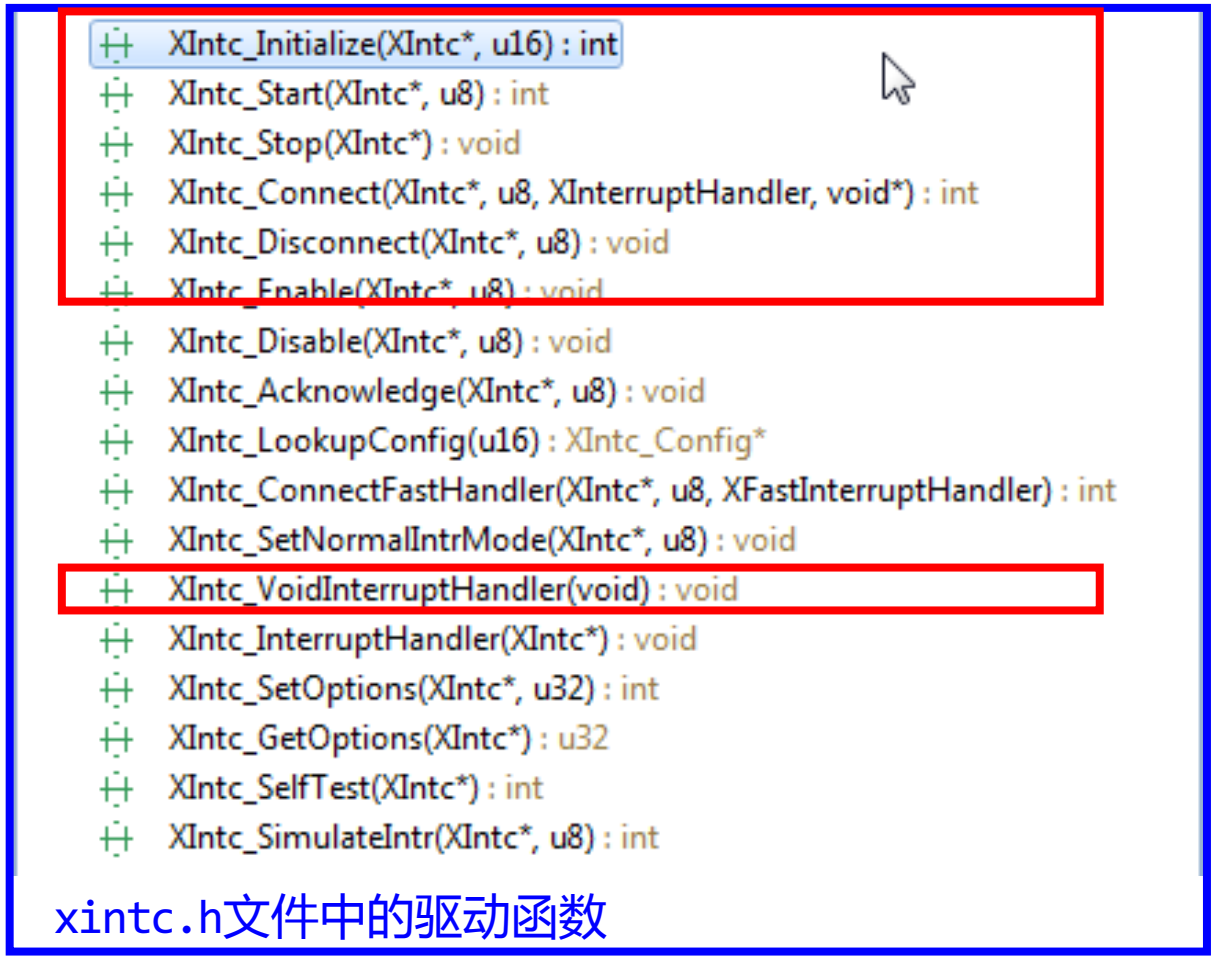


对MER、IER、IAR等寄存器的操作除了使用Xil_Out32函数操作之外，还可以使用AXI INTC的API函数

8.4 Xilinx的中断控制器-AXI INTC

- AXI INTC的编程控制
  - 初始化，通常包括两个方面：
    - 修改MER，使得HIE和ME都为1；
    - 修改IER，使得连接了中断请求源的相应位为1，允许中断请求
  - 中断结束
    - 写IAR，使服务了的中断请求位清零

- AXI INTC中断控制器驱动API
  - 详情见P242-P243



8.4 Xilinx的中断控制器-AXI INTC

▶ AXI INTC中断控制器驱动API

- AXI INTC中断控制器驱动相关数据结构
  - 中断向量表

```
typedef struct {
    XInterruptHandler Handler;          //中断服务程序入口地址
    void *CallBackRef;                  //中断服务程序参数地址
} XIntc_VectorTableEntry;
```

  - INTC配置项

```
typedef struct {
    u16 DeviceId;                       //操作系统维护的设备ID号
    u32 BaseAddress;                    //设备对应的基地址
    u32 AckBeforeService;               //何时写中断响应寄存器选项
    u32 Options;                        //总中断服务程序处理方式
    XIntc_VectorTableEntry HandlerTable[XPAR_INTC_MAX_NUM_INTR_INPUTS]; //中断向量表
} XIntc_Config;
```

  - INTC数据结构

```
typedef struct {
    u32 BaseAddress;                    //控制器基地址
    u32 IsReady;                        //控制器是否已初始化标志
    u32 IsStarted;                      //控制器是否已开启标志
    u32 UnhandledInterrupts;            //统计未处理的中断请求数目
    XIntc_Config *CfgPtr;               //中断控制器配置项
} XIntc;
```

Agenda

- 内容
  - 中断的基本概念，中断响应过程
  - 典型微处理器中断系统简介
  - Xilinx的中断控制器-AXI INTC
  - GPIO中断方式接口设计
  - AXI Timer接口
  - AXI SPI接口
- 目的
  - 理解Interrupt 的含义，优点、分类；
  - 理解中断源、中断请求、中断类型码、中断优先级、中断向量入口地址(Interrupt Vector Address)、中断向量表等术语的含义和作用；
  - 理解CPU响应中断的过程；
  - 理解X86和Microbalze系统的中断处理过程；
  - 掌握AXI INTC原理，学会Microbalze系统中断程序设计；
  - 掌握AXI Timer接口设计；
  - 掌握AXI SPI接口设计。

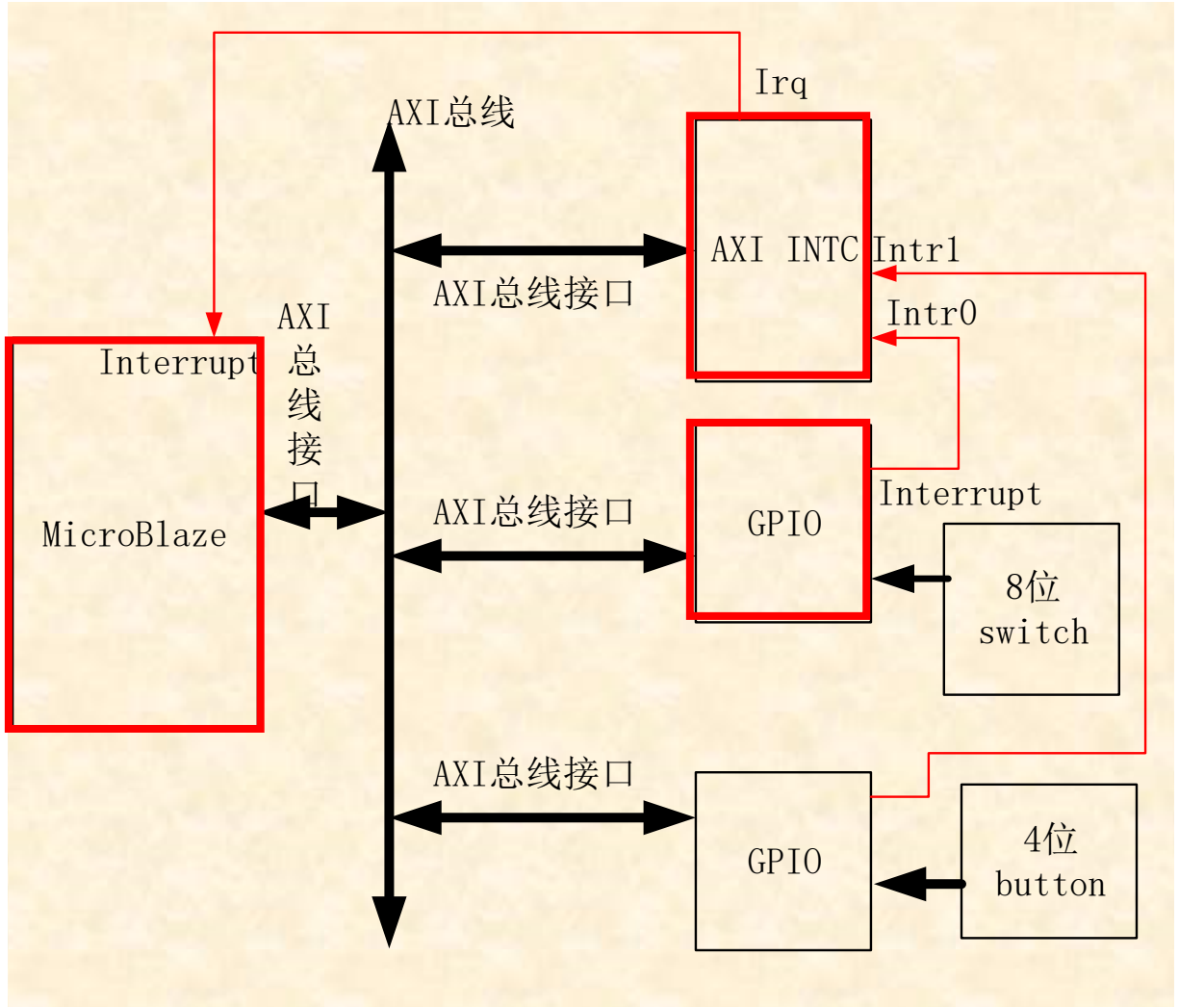
8.5 GPIO中断方式接口设计

▶ 电路框图

- 如何在Xinlinx FPGA中搭建硬件电路？
  ——详见实验课件和实验教材

- 中断的三个层次
  - 模块(GPIO)
  - 中断控制器(INTC)
  - CPU(MicroBlaze)

- CPU能否收到模块的中断请求取决于
  - 模块的中断控制
  - 中断控制器的控制
  - CPU的控制



8.5 GPIO中断方式接口设计

▶ AXI GPIO并行接口控制器中断原理简介

- GPIO内部中断相关寄存器

名称	偏移地址	含义	读写操作
GIER	0x11C	全局中断屏蔽寄存器	最高位bit31控制GPIO是否输出中断信号Irq
IP IER	0x128	中断屏蔽寄存器	控制各个通道是否允许产生中断 bit0-通道1；bit1-通道2
IP ISR	0x120	中断状态寄存器	各个通道的中断请求状态，写1将清除相应位的中断状态： bit0-通道1；bit1-通道2

  - GPIO中断产生逻辑模块当检测到GPIO_DATA_IN输入数据发生变化时，就可以产生中断信号，但是是否输出中断信号，受中断允许控制寄存器控制。中断控制逻辑与AXI INTC类似，但是没有优先级判断，仅2个中断源。

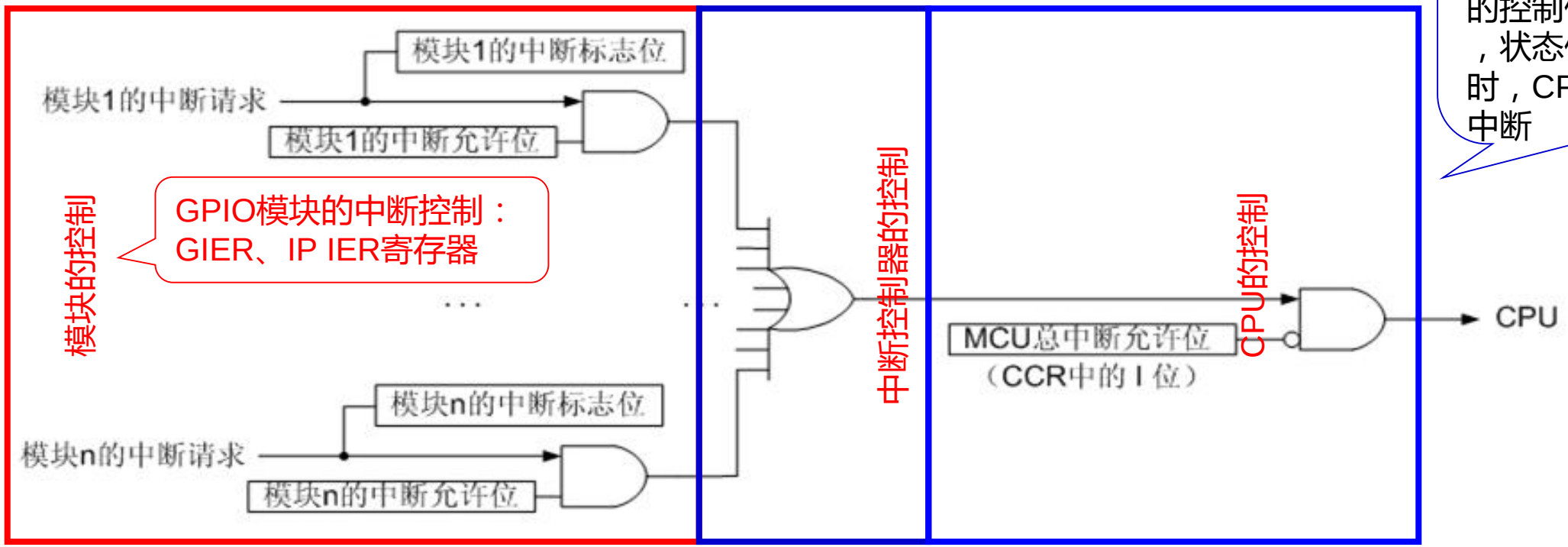
回顾：8.1 中断的基本概念

▶ 中断模型

- CPU是否能够收到可屏蔽中断，一般受三个控制位控制
  - 模块的中断控制
  - 中断控制器的控制
  - CPU的控制

80X86系统中用中断控制器8259A多多个外部中断进行管理
Microblaze系统中用中断控制器AXI INTC对多个中断源进行管理

MicroBlaze有一个机器状态寄存器MSR，该寄存器中的控制位IE位为1，状态位BIP位为0时，CPU可以响应中断

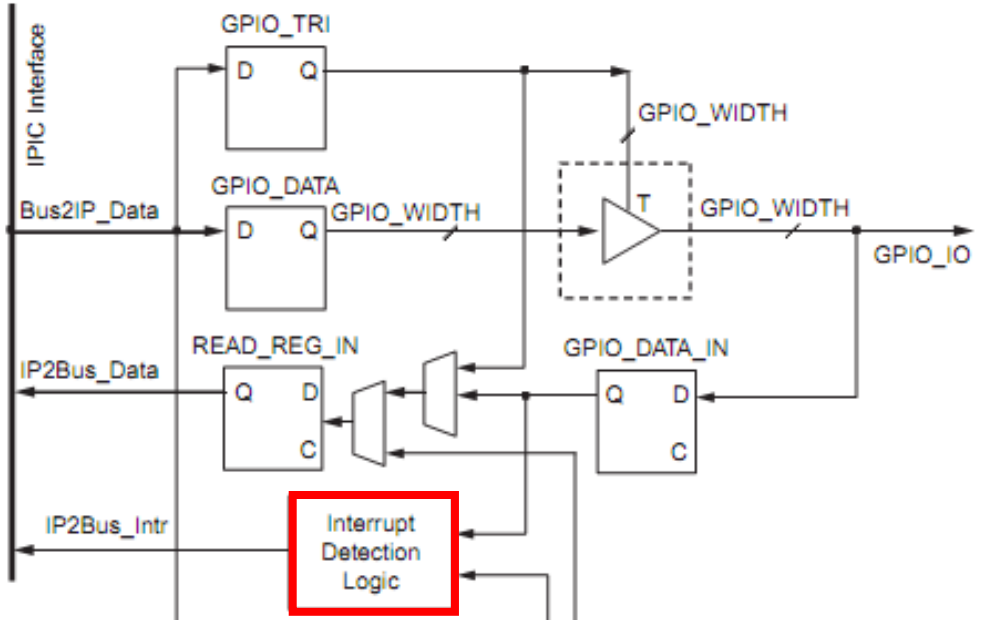


8.5 GPIO中断方式接口设计

- CPU能否收到模块的中断请求取决于：
  - 模块的中断控制
    - 如果要允许I/O模块的中断，则需要对GIER、IP IER进行编程。
      - I/O的API函数
      - Xil_Out(addr, data), Xil_In()

```
XGpio_Initialize(XGpio*, u16) : int
XGpio_LookupConfig(u16) : XGpio_Config*
XGpio_CfgInitialize(XGpio*, XGpio_Config*, u32) : int
XGpio_SetDataDirection(XGpio*, unsigned, u32) : void
XGpio_GetDataDirection(XGpio*, unsigned) : u32
XGpio_DiscreteRead(XGpio*, unsigned) : u32
XGpio_DiscreteWrite(XGpio*, unsigned, u32) : void
XGpio_DiscreteSet(XGpio*, unsigned, u32) : void
XGpio_DiscreteClear(XGpio*, unsigned, u32) : void
XGpio_SelfTest(XGpio*) : int
XGpio_InterruptGlobalEnable(XGpio*) : void
XGpio_InterruptGlobalDisable(XGpio*) : void
XGpio_InterruptEnable(XGpio*, u32) : void
XGpio_InterruptDisable(XGpio*, u32) : void
XGpio_InterruptClear(XGpio*, u32) : void
XGpio_InterruptGetEnabled(XGpio*) : u32
XGpio_InterruptGetStatus(XGpio*) : u32
```

xgpio.h文件中的驱动函数

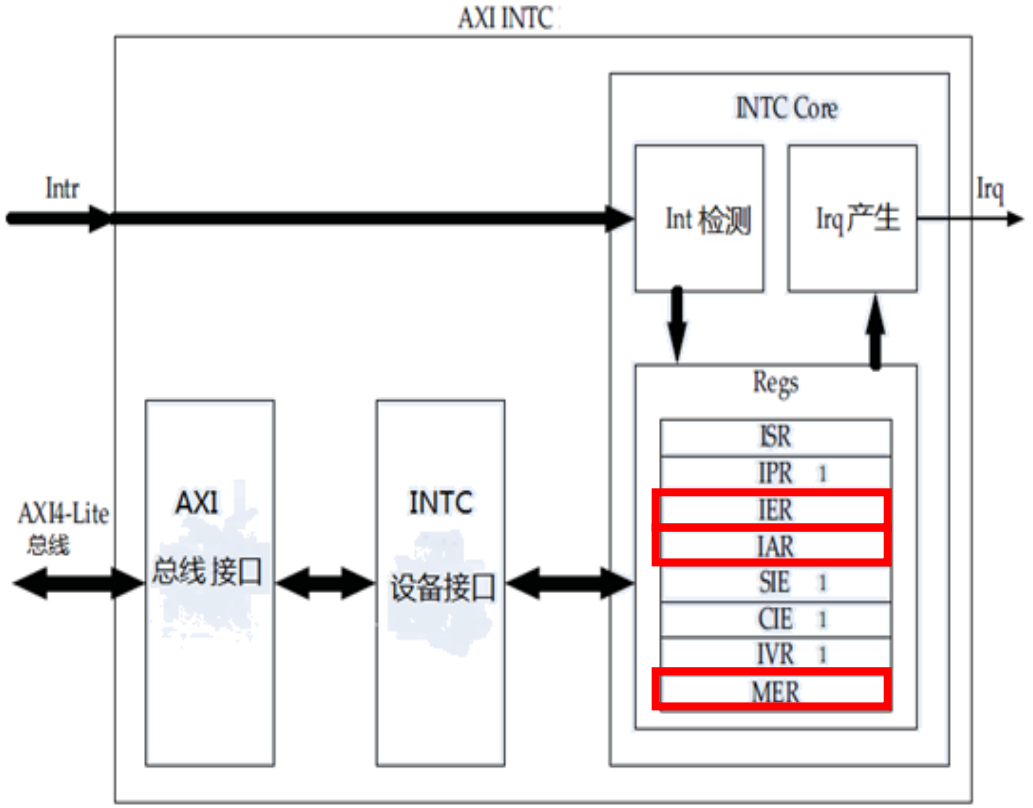


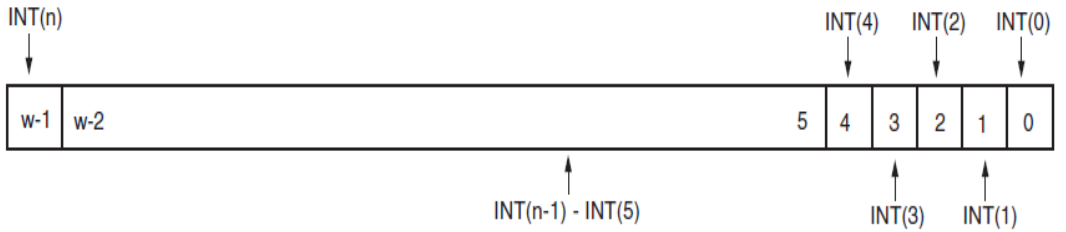
名称	偏移地址	含义	读写操作
GIER	0x11C	全局中断屏蔽寄存器	最高位bit31控制GPIO是否输出中断信号Irq
IP IER	0x128	中断屏蔽寄存器	控制各个通道是否允许产生中断 bit0-通道1；bit1-通道2
IP ISR	0x120	中断状态寄存器	各个通道的中断请求状态，写1将清除相应位的中断状态： bit0-通道1；bit1-通道2

若IP IER某位为1，则GPIO的中断检测逻辑模块检测到对应通道的GPIO_DATA_IN输入数据发生变化时，就可以产生IO模块中断信号，但是否输出中断信号，受中断允许控制寄存器GIER控制。

8.5 GPIO中断方式接口设计

- CPU能否收到模块的中断请求取决于：
  - 模块的中断控制
  - 中断控制器的控制
    - 则需要对IER、MER进行编程。
      - INT的API函数
      - Xil_Out(addr, data) , Xil_In()
  - CPU的控制

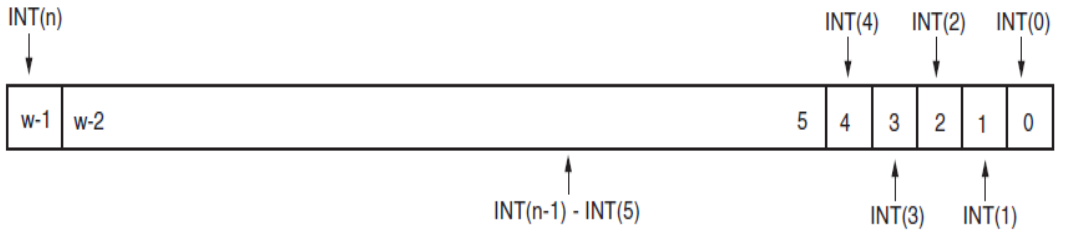




8.5 GPIO中断方式接口设计

- CPU能否收到模块的中断请求取决于：
  - 模块的中断控制
  - 中断控制器的控制
    - 则需要对IER、MER进行编程。
      - INT的API函数
      - Xil_Out(addr, data) , Xil_In()
  - CPU的控制

寄存器名称	偏移地址	允许操作	初始值	含义
ISR	0x0	Read / Write	0x0	中断请求状态寄存器
IPR（可选）	0x4	Read	0x0	中断悬挂寄存器
IER	0x8	Read / Write	0x0	中断屏蔽寄存器
IAR	0xC	Write	0x0	中断响应寄存器
SIE（可选）	0x10	Write	0x0	中断允许设置寄存器
CIE（可选）	0x14	Write	0x0	中断允许清除寄存器
IVR（可选）	0x18	Read	0x0	中断类型码寄存器
MER	0x1C	Read / Write	0x0	主中断屏蔽寄存器

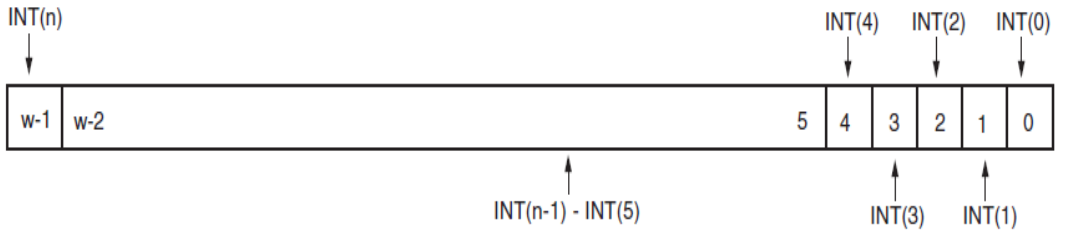


8.5 GPIO中断方式接口设计

- CPU能否收到模块的中断请求取决于：
  - 模块的中断控制
  - 中断控制器的控制
    - 则需要对IER、MER进行编程。
      - INT的API函数
      - Xil_Out(addr, data) ，Xil_In()
  - CPU的控制

```
XIntc_Initialize(XIntc*, u16) : int
XIntc_Start(XIntc*, u8) : int
XIntc_Stop(XIntc*) : void
XIntc_Connect(XIntc*, u8, XInterruptHandler, void*) : int
XIntc_Disconnect(XIntc*, u8) : void
XIntc_Enable(XIntc*, u8) : void
XIntc_Disable(XIntc*, u8) : void
XIntc_Acknowledge(XIntc*, u8) : void
XIntc_LookupConfig(u16) : XIntc_Config*
XIntc_ConnectFastHandler(XIntc*, u8, XFastInterruptHandler) : int
XIntc_SetNormalIntrMode(XIntc*, u8) : void
XIntc_VoidInterruptHandler(void) : void
XIntc_InterruptHandler(XIntc*) : void
XIntc_SetOptions(XIntc*, u32) : int
XIntc_GetOptions(XIntc*) : u32
XIntc_SelfTest(XIntc*) : int
XIntc_SimulateIntr(XIntc*, u8) : int
```

xintc.h文件中的驱动函数

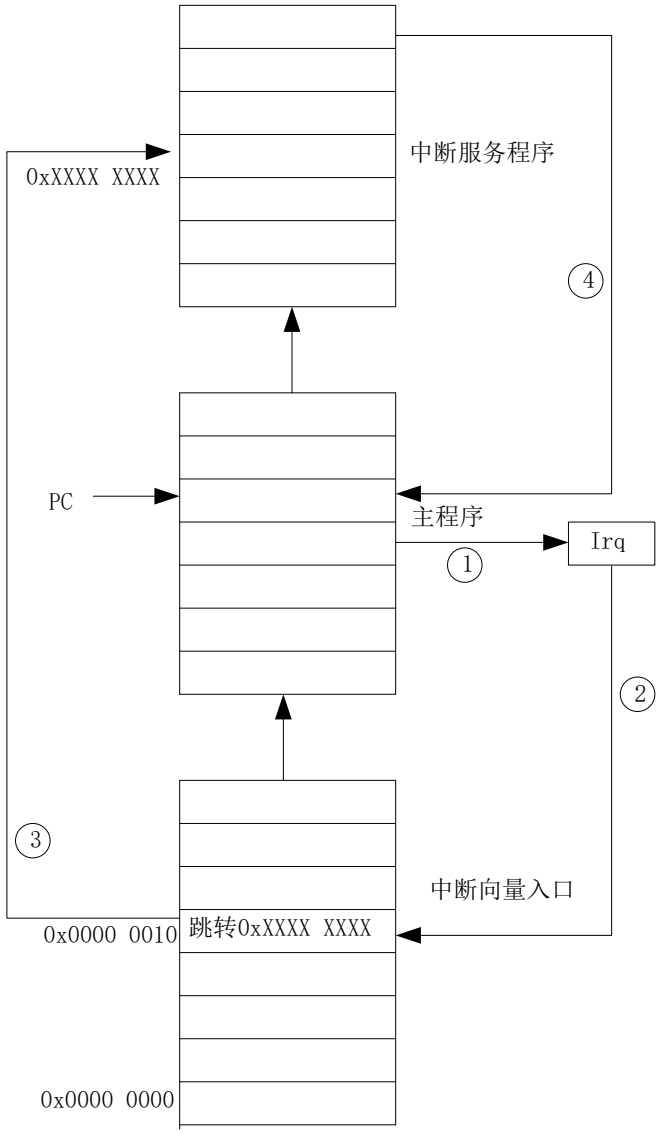


8.5 GPIO中断方式接口设计

- CPU能否收到模块的中断请求取决于：
  - 模块的中断控制
  - 中断控制器的控制
  - CPU的控制
    - 则需要对MSR中的I位进行编程、需要设定中断服务函数入口。
      - CPU的API函数

mb_interface.h文件中的驱动函数

```
microblaze_enable_interrupts(void) : void
microblaze_disable_interrupts(void) : void
microblaze_enable_icache(void) : void
microblaze_disable_icache(void) : void
microblaze_enable_dcache(void) : void
microblaze_disable_dcache(void) : void
microblaze_enable_exceptions(void) : void
microblaze_disable_exceptions(void) : void
microblaze_register_handler(XInterruptHandler, void*) : void
```



8.5 GPIO中断方式接口设计

- CPU能否收到模块的中断请求取决于：
  - 模块的中断控制
  - 中断控制器的控制
  - CPU的控制

- 对模块、中断控制器、CPU的编程可以采用两种不同方法：
  - IO/INTC等模块的API函数
  - 直接地址读、写(Xil_In、Xil_Out)

8.5 GPIO中断方式接口设计

实验书例子采用，需要知道这些函数的具体含义。程序模块化程度高。

► 函数接口

- 可以基于已有驱动提供的API —— GPIO位于头文件xgpio.h中，INTC位于头文件xintc.h中，CPU位于头文件mb_interface.h中

```
XGpio_Initialize(XGpio*, u16) : int
XGpio_LookupConfig(u16) : XGpio_Config*
XGpio_CfgInitialize(XGpio*, XGpio_Config*, u32) : int
XGpio_SetDataDirection(XGpio*, unsigned, u32) : void
XGpio_GetDataDirection(XGpio*, unsigned) : u32
XGpio_DiscreteRead(XGpio*, unsigned) : u32
XGpio_DiscreteWrite(XGpio*, unsigned, u32) : void
XGpio_DiscreteSet(XGpio*, unsigned, u32) : void
XGpio_DiscreteClear(XGpio*, unsigned, u32) : void
XGpio_SelfTest(XGpio*) : int
XGpio_InterruptGlobalEnable(XGpio*) : void
XGpio_InterruptGlobalDisable(XGpio*) : void
XGpio_InterruptEnable(XGpio*, u32) : void
XGpio_InterruptDisable(XGpio*, u32) : void
XGpio_InterruptClear(XGpio*, u32) : void
XGpio_InterruptGetEnabled(XGpio*) : u32
XGpio_InterruptGetStatus(XGpio*) : u32
```

xgpio.h文件中的驱动函数

```
XIntc_Initialize(XIntc*, u16) : int
XIntc_Start(XIntc*, u8) : int
XIntc_Stop(XIntc*) : void
XIntc_Connect(XIntc*, u8, XInterruptHandler, void*) : int
XIntc_Disconnect(XIntc*, u8) : void
XIntc_Enable(XIntc*, u8) : void
XIntc_Disable(XIntc*, u8) : void
XIntc_Acknowledge(XIntc*, u8) : void
XIntc_LookupConfig(u16) : XIntc_Config*
XIntc_ConnectFastHandler(XIntc*, u8, XFastInterruptHandler) : int
XIntc_SetNormalIntrMode(XIntc*, u8) : void
XIntc_VoidInterruptHandler(void) : void
XIntc_InterruptHandler(XIntc*) : void
XIntc_SetOptions(XIntc*, u32) : int
XIntc_GetOptions(XIntc*) : u32
XIntc_SelfTest(XIntc*) : int
XIntc_SimulateIntr(XIntc*, u8) : int
```

xintc.h文件中的驱动函数

mb_interface.h文件中的驱动函数

```
microblaze_enable_interrupts(void) : void
microblaze_disable_interrupts(void) : void
microblaze_enable_icache(void) : void
microblaze_disable_icache(void) : void
microblaze_enable_dcache(void) : void
microblaze_disable_dcache(void) : void
microblaze_enable_exceptions(void) : void
microblaze_disable_exceptions(void) : void
microblaze_register_handler(XInterruptHandler, void*) : void
```

8.5 GPIO中断方式接口设计

▶ 函数接口

- 也可以直接通过libc提供的输入输出语句对硬件进行直接控制 —— 位于xil_io.h中

```
/****************************************************************************/
/**
*
* Perform an input operation for an 8-bit memory location by reading from the
* specified address and returning the value read from that address.
*
* @param    Addr contains the address to perform the input operation at.
*
* @return   The value read from the specified input address.
*
* @note     None.
*
****************************************************************************/
#define Xil_In8(Addr)  (*(volatile u8  *)(Addr))

/****************************************************************************/
/**
*
* Perform an input operation for a 16-bit memory location by reading from the
* specified address and returning the value read from that address.
*
* @param    Addr contains the address to perform the input operation at.
*
* @return   The value read from the specified input address.
*
* @note     None.
*
****************************************************************************/
#define Xil_In16(Addr) (*(volatile u16 *)(Addr))

/****************************************************************************/
/**
*
* Perform an input operation for a 32-bit memory location by reading from the
* specified address and returning the value read from that address.
*
* @param    Addr contains the address to perform the input operation at.
*
* @return   The value read from the specified input address.
*
* @note     None.
*
****************************************************************************/
#define Xil_In32(Addr)  (*(volatile u32 *)(Addr))
```

```
/****************************************************************************/
/**
*
* Perform an output operation for an 8-bit memory location by writing the
* specified value to the specified address.
*
* @param    Addr contains the address to perform the output operation at.
* @param    value contains the value to be output at the specified address.
*
* @return   None
*
* @note     None.
*
****************************************************************************/
#define Xil_Out8(Addr, Value)  \
    (*(volatile u8  *)((Addr)) = (Value))

/****************************************************************************/
/**
*
* Perform an output operation for a 16-bit memory location by writing the
* specified value to the specified address.
*
* @param    Addr contains the address to perform the output operation at.
* @param    value contains the value to be output at the specified address.
*
* @return   None
*
* @note     None.
*
****************************************************************************/
#define Xil_Out16(Addr, Value) \
    (*(volatile u16 *)((Addr)) = (Value))

/****************************************************************************/
/**
*
* Perform an output operation for a 32-bit memory location by writing the
* specified value to the specified address.
*
* @param    addr contains the address to perform the output operation at.
* @param    value contains the value to be output at the specified address.
*
* @return   None
*
* @note     None.
*
****************************************************************************/
#define Xil_Out32(Addr, Value) \
    (*(volatile u32 *)((Addr)) = (Value))
```

8.5 GPIO中断方式接口设计

- 查询程序设计思路
  - 主程序不停的读取GPIO的ISR寄存器，当其对应的位为1时，读取GPIO的数据寄存器并输出到console（xil_printf函数实现，stdio.h），并写ISR相应位
    - XIL_IN
    - XIL_OUT
- 中断程序设计思路（详见实验书&& “chap7——第十七讲 GPIO中断输入接口.pdf” ）
  - 主程序开放microblaze，INTC ,GPIO中断，不停的检测输出标志是否为1，是则输出数据到console，并将输出标志设置为0
  - 中断服务程序读取数据（或输出数据）并设立输出标志位为1
- 延时方式
  - For循环
    - For(i=0;i<constant;i++);
    - Constant的值决定延时的时间长短

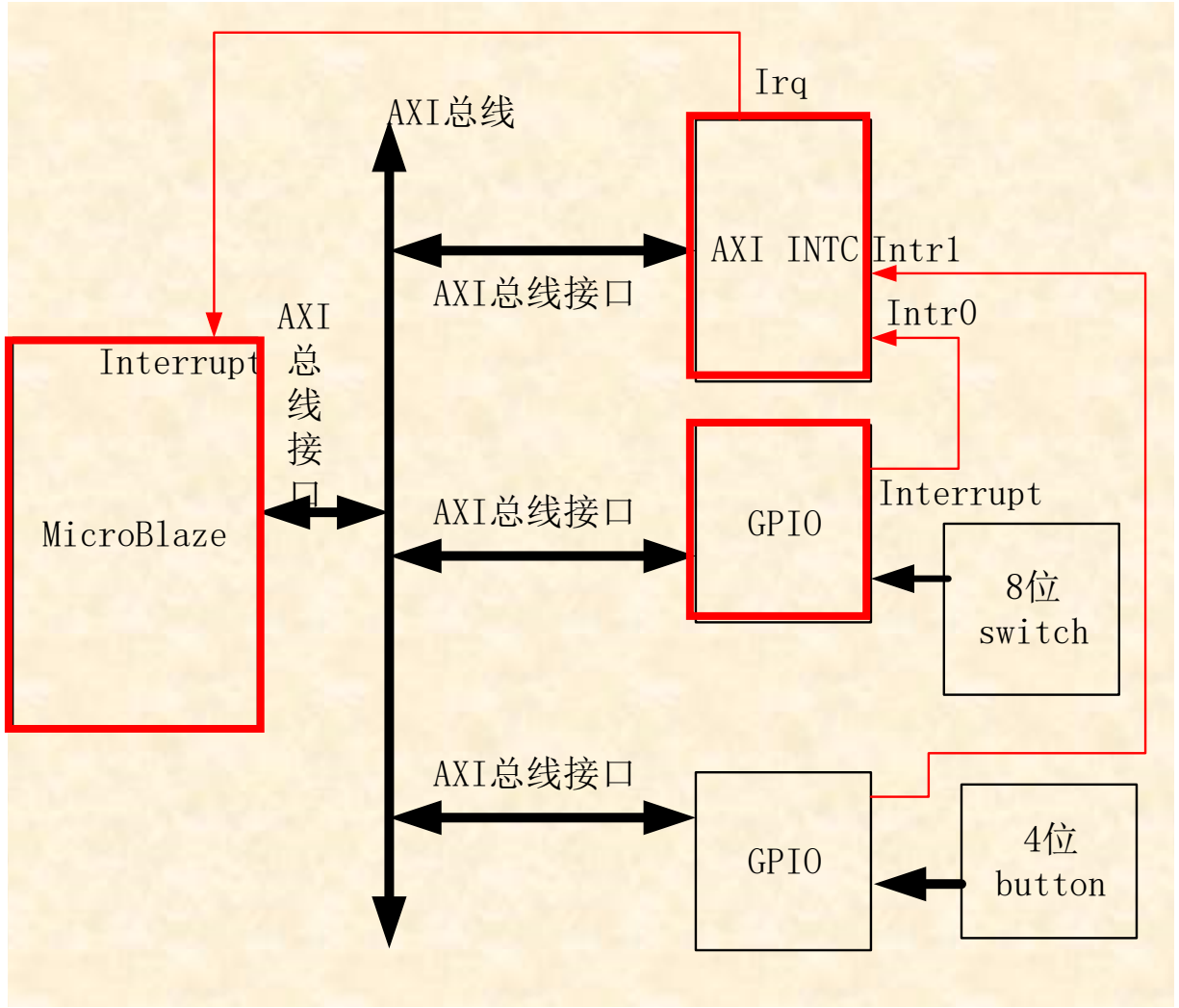
8.5 GPIO中断方式接口设计

▶ 电路框图

- 如何在Xinlinx FPGA中搭建硬件电路？
  ——详见实验课件和实验教材

- 中断的三个层次
  - 模块(GPIO)
  - 中断控制器(INTC)
  - CPU(MicroBlaze)

- CPU能否收到模块的中断请求取决于
  - 模块的中断控制
  - 中断控制器的控制
  - CPU的控制



8.5 GPIO中断方式接口设计

▶ 查询程序代码(参考)

```c
// 查询方式button按键以及switch输入的测试程序
#include "xparameters.h"                //The hardware configuration describing constants
#include "stdio.h"
#include "xil_io.h"                     // IO functions
#include "xil_types.h"

#define btn_DATA        0x40000000      // button 数据寄存器地址
#define btn_TRI         0x40000004      // button 控制寄存器地址
#define btn_ISR         0x40000120      // button 中断状态寄存器地址

#define sw_DATA         0x40040000      // switch 数据寄存器地址
#define sw_TRI          0x40040004      // switch 控制寄存器地址
#define sw_ISR          0x40040120      // switch 中断状态寄存器地址

// 按键标志位
short   pshBtn,pshSw;

int main(void)
{
        short btn,sw;
        xil_printf("\r\nRunning GpioInput Test(Po11)!\r\n");
        Xil_Out8(btn_TRI,0xff);         // btn is used as input
        Xil_Out8(sw_TRI,0xff);          // sw is used as input
        pshBtn = 0x00;
        pshSw  = 0x00;
        while(1)
        {
        }
        return 0;
}
```

```c
        while(1)
        {
                pshBtn = Xil_In8(btn_ISR);
                pshSw  = Xil_In8(sw_ISR);
                if(pshBtn)              //若按下按键，则打印相关信息
                {
                        btn = Xil_In8(btn_DATA);
                        Xil_Out8(btn_ISR,0x01);
                        xil_printf("Button Pushed!!!the state is 0x%X\n\r",btn);
                }
                if(pshSw)               //若拨动Switch开关，则打印相关信息
                {
                        sw  = Xil_In8(sw_DATA);
                        Xil_Out8(sw_ISR,0x01);
                        xil_printf("Switch Pushed!!!the state is 0x%X\n\r",sw);
                }
        }
```

▶ 中断程序（No API）代码(参考)

```
// 中断方式button按键以及switch输入的测试程序

#include "xparameters.h"       //The hardware configuration describing constants
#include "xintc.h"             //Interrupt Controller API functions
#include "stdio.h"
#include "xil_io.h"            // IO functions
#include "xil_types.h"
#include "mb_interface.h"
#include "xgpio.h"             //GPIO API functions

#define btn_DATA        0x40000000      // button 数据寄存器地址
#define btn_TRI         0x40000004      // button 控制寄存器地址
#define btn_GIER        0x4000011c      // button 中断全局允许寄存器地址
#define btn_IER         0x40000128      // button 中断通道允许寄存器地址
#define btn_ISR         0x40000120      // button 中断状态寄存器地址

#define sw_DATA         0x40040000      // switch 数据寄存器地址
#define sw_TRI          0x40040004      // switch 控制寄存器地址
#define sw_GIER         0x4004011c      // switch 中断全局允许寄存器地址
#define sw_IER          0x40040128      // switch 中断通道允许寄存器地址
#define sw_ISR          0x40040120      // switch 中断状态寄存器地址

#define intc_ISR        0x41200000
#define intc_IER        0x41200008
#define intc_IAR        0x4120000C
#define intc_MER        0x4120001C

// 注册总中断服务程序地址
void My_ISR (void) __attribute__ ((interrupt_handler));

void Initialize();                      //初始化函数（包含中断初始化）
```

▶ 中断程序（No API）代码(参考) — 续

```
32  void PushBtnHandler();                  //按键的处理函数
33  void SwitchHandler();                   //拨动开关的处理函数
34  void Delay_50ms();                      //延时函数
35
36  short   flag_Sw,flag_Btn;               // 按键标志位
37  short   sw,btn;                         // 按键键值
38
39  int main(void)
40  {
41          xil_printf("\r\nRunning GpioInput Interrupt Test(No APP)!\r\n");
42          Initialize();
43          while(1)
44          {
45                  if(flag_Sw)             //若拨动Switch开关，则打印相关信息
46                  {
47                          xil_printf("Switch Interrupt Triggered!!!the result is 0x%X\n\r",sw);
48                          flag_Sw=0;
49                  }
50                  if(flag_Btn)            //若按下按键，则打印相关信息
51                  {
52                          xil_printf("Button Interrupt Triggered!!!the result is 0x%X\n\r",btn);
53                          flag_Btn=0;
54                  }
55                  int status;
56                  status = Xil_In32(intc_ISR);          // 读取ISR
57                  status = Xil_In32(sw_ISR);            // 读取ISR
58                  status = Xil_In32(btn_ISR);           // 读取ISR
59
60          }
61          return 0;
62  }
```

▶ 中断程序（No API）代码(参考) — 续

```
63
64  void Initialize()
65  {
66          flag_Sw = 0x00;
67          flag_Btn  = 0x00;
68          sw = 0x00;
69          btn = 0x00;
70
71          Xil_Out8(sw_TRI,0xff);                  // sw is used as input
72          Xil_Out8(sw_IER,0x01);                  // channel 0 inter enable
73          Xil_Out32(sw_GIER,0x80000000);          // sw Interrupt enable
74
75          Xil_Out8(btn_TRI,0xff);                 // btn is used as input
76          Xil_Out8(btn_IER,0x01);                 // channel 0 inter enable
77          Xil_Out32(btn_GIER,0x80000000);         // btn Interrupt enable
78
79          Xil_Out32(intc_IAR,0xffffffff);         // claer all irq requests
80          Xil_Out32(intc_IER,0x03);               // Intr[1:0] Interrupt enable
81          Xil_Out32(intc_MER,0x03);               // INTC Interrupt enable
82
83          microblaze_enable_interrupts();         // CPU interrupt enable
84  }
```

8.5 GPIO中断方式接口设计

▶ 中断程序（No API）代码(参考) — 续

```
85
86  void My_ISR(void)
87  {
88          int status;
89          status = Xil_In32(intc_ISR);          // 读取ISR
90          if(status&0x02)                       // ISR[1]=1,说明是Switch中断
91          {
92                  SwitchHandler();
93          }
94          if(status&0x01)                       // ISR[0]=1,说明是PshButton中断
95          {
96                  PushBtnHandler();
97          }
98
99          Xil_Out32(intc_IAR,status);           // 写IAR清INTC中断标志
100 }
101
102 void SwitchHandler()
103 {
104         sw = Xil_In8(sw_DATA);                //读取Switch开关的状态值
105         flag_Sw=1;
106         int isr_status;
107         isr_status = Xil_In32(sw_ISR);
108         Xil_Out32(sw_ISR, 0x01);              //清除中断标志位
109 }
```

```
110
111 void PushBtnHandler()
112 {
113
114         btn = Xil_In8(btn_DATA);              //读取Switch开关的状态值
115         flag_Btn=1;
116         Xil_Out8(btn_IER,0x00);               // channel 0 inter disable
117         Delay_50ms();
118         int isr_status;
119         isr_status = Xil_In32(btn_ISR);
120         Xil_Out32(btn_ISR, 0x01);             //清除中断标志位
121         Xil_Out8(btn_IER,0x01);               // channel 0 inter enable
122 }
123
124 void Delay_50ms()
125 {
126         int i;
127         for(i=0;i<5000000;i++);
128 }
```

Agenda

- 内容
  - 中断的基本概念，中断响应过程
  - 典型微处理器中断系统简介
  - Xilinx的中断控制器-AXI INTC
  - GPIO中断方式接口设计
  - AXI Timer接口
  - AXI SPI接口
- 目的
  - 理解Interrupt 的含义，优点、分类；
  - 理解中断源、中断请求、中断类型码、中断优先级、中断向量入口地址(Interrupt Vector Address)、中断向量表等术语的含义和作用；
  - 理解CPU响应中断的过程；
  - 理解X86和Microbalze系统的中断处理过程；
  - 掌握AXI INTC原理，学会Microbalze系统中断程序设计；
  - 掌握AXI Timer接口设计；
  - 掌握AXI SPI接口设计。

▶ 定时器和计数器

- 计数
  - 对脉冲的个数的个数进行计数，关心计数脉冲的个数，而非脉冲的时间间隔；
  - 脉冲的出现频率不一定一成不变；
  - 可由硬件计数器实现计数，以计数脉冲作为计数器的时钟（74LS90/92/93：数字钟）；

- 定时：
  - 取得给定事件发生的时间间隔，关心前后前后两个事件发生时的时间间隔长短；
  - 可由硬件计数器来实现定时，以固定频率的信号作为计数器的时钟，前一事件发生时开始计数，后一事件发生时计数终止，计数次数*时钟周期即为前后时延。

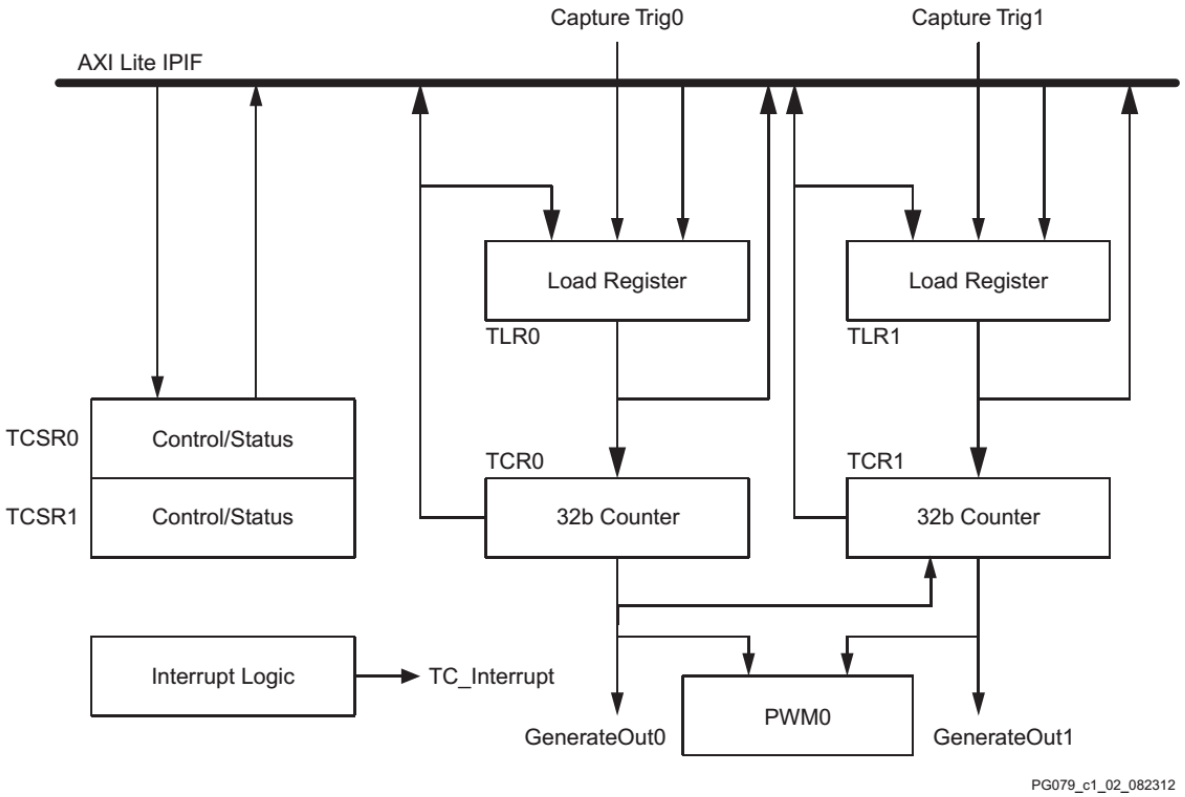
8.6 AXI Timer接口设计

定时方法

- 软件定时
  - 利用指令的执行时间，设计循环程序，使CPU执行延迟子程序来产生定时，
  - 缺点：
    - 执行延迟时，CPU一直被占用，降低了CPU的效率；
    - 不同CPU的时钟不一，要重新计算CX；
    - 时间精度受限于CPU时钟频率
- 硬件定时
  - 用计数器/定时器作为主要硬件，在软件简单指令的控制下产生精确的时间延迟。
  - 优点：
    - 计数时不占用CPU时间，如利用定时器/计数器产生中断信号，可建立多任务环境，故提 高了CPU效率。
    - 时间精度高

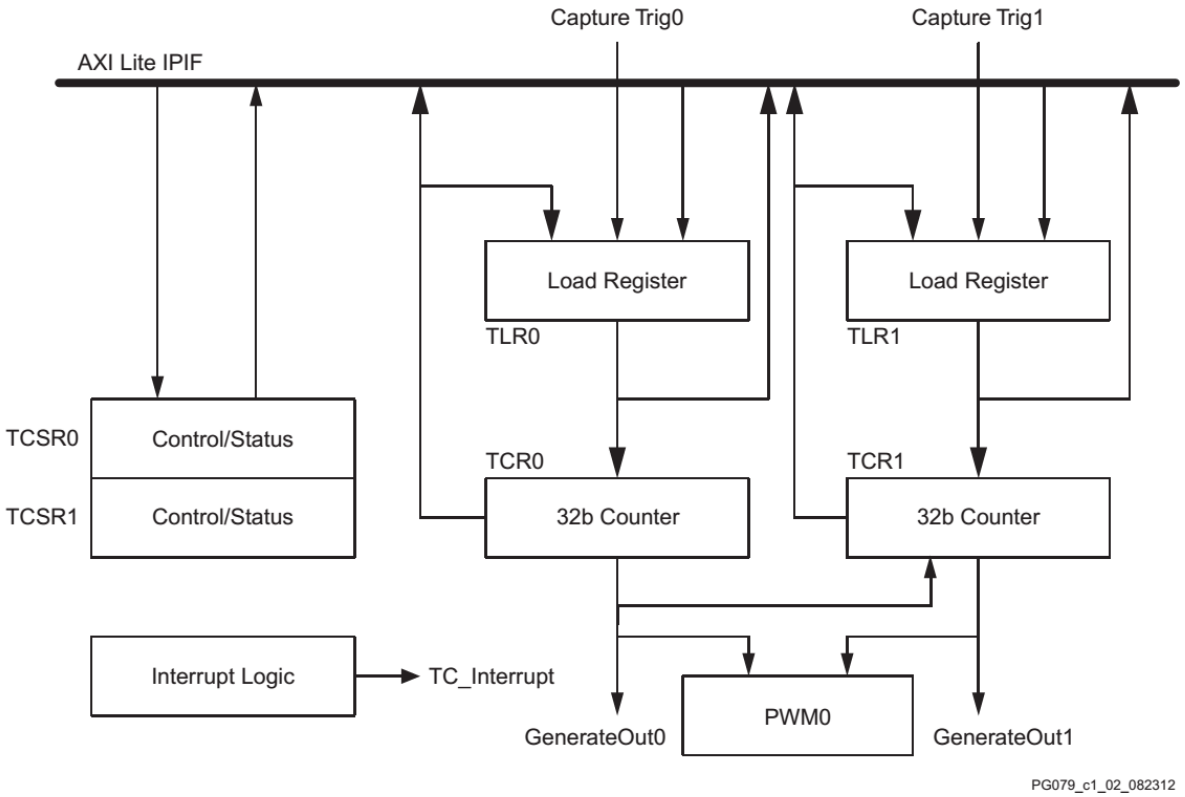
8.6 AXI Timer接口设计

- AXI Timer的结构
  - 每个AXI Timer含有2个独立的Timer模块
  - 每个模块具有四种工作模式
    - Generate mode（定时模式）
      - The generate value is used to generate a single interrupt at the expiration of an interval or
        a continuous series of interrupts with a programmable interval.
    - Capture mode
      - The capture value is the timer value that has been latched on detection of an external event.
    - Pulse Width Modulation (PWM) mode
    - Cascade mode
  - The clock rate of the timer modules is S_AXI_ACLK



8.6 AXI Timer接口设计

- AXI Timer的定时模式
  - 定时器可以分为两种计数方式：向上计数和向下计数；
  - 定时器可以工作在8位、16位、32位三种模式；
  - 采用小字节序
  - 两个定时器中的任意一个计数结束都会产生中断，如果允许输出中断信号，则使Interrupt输出高电平，从而产生中断请求。

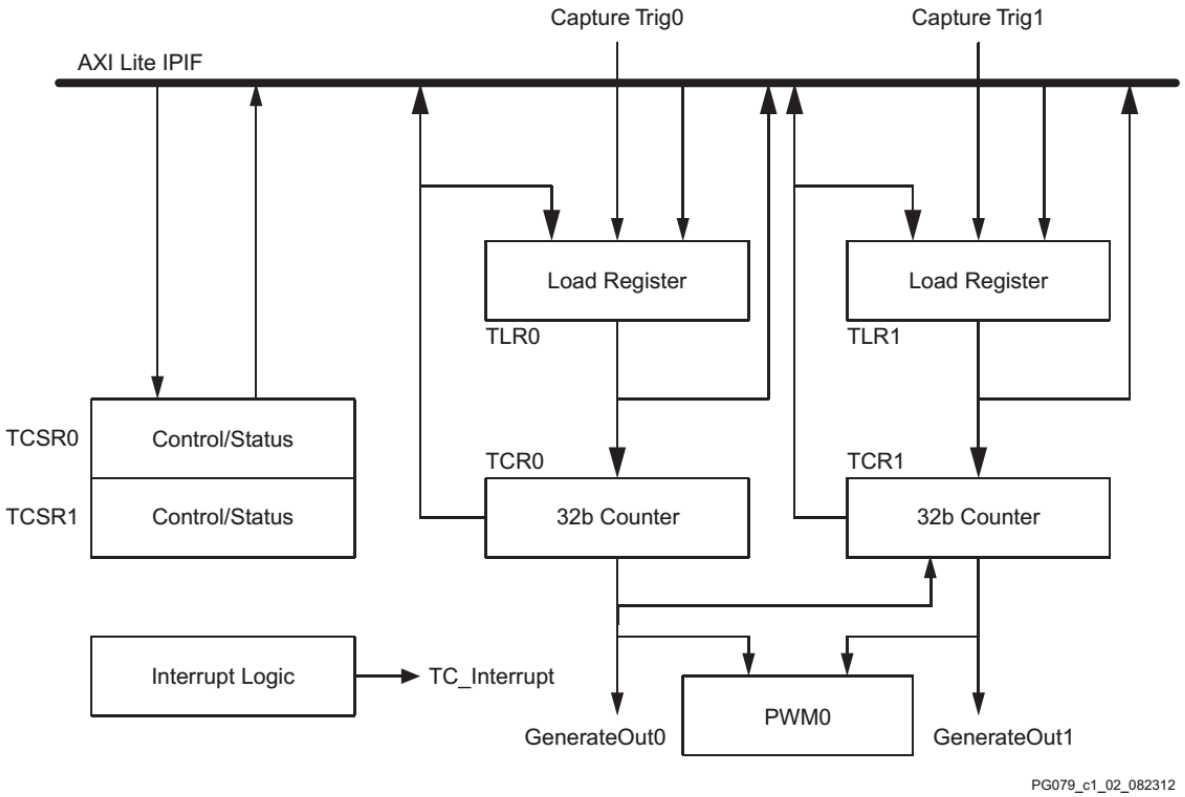


- 产生中断的间隔
  - 加计数：T=(TCRmax-TLR)*AXI_CLK_PERIOD
  - 减计数：T=TLR* AXI_CLK_PERIOD

8.6 AXI Timer接口设计

- AXI Timer的寄存器

寄存器名称	偏移地址	功能描述
TCSR0	0x00	定时器0控制寄存器
TLR0	0x04	定时器0预置数寄存器
TCR0	0x08	定时器0计数寄存器
TCSR1	0x10	定时器1控制寄存器
TLR1	0x14	定时器1预置数寄存器
TCR1	0x18	定时器1计数寄存器

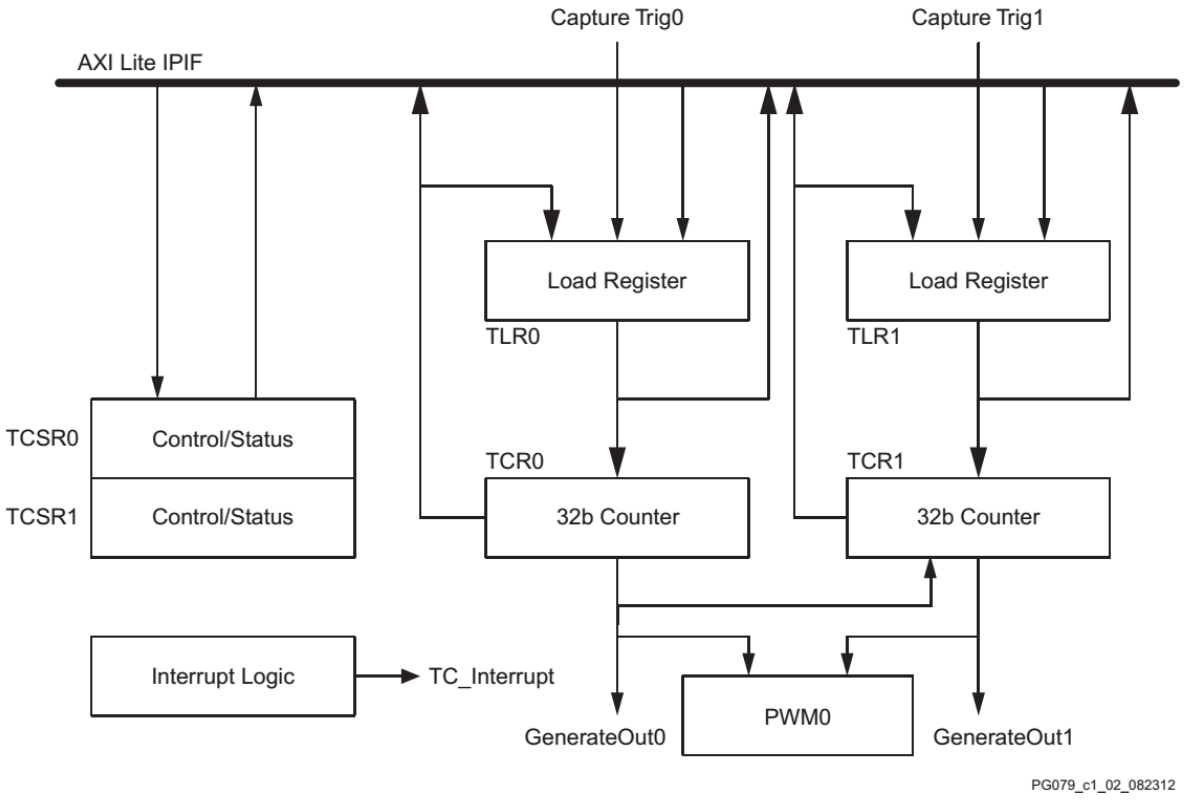


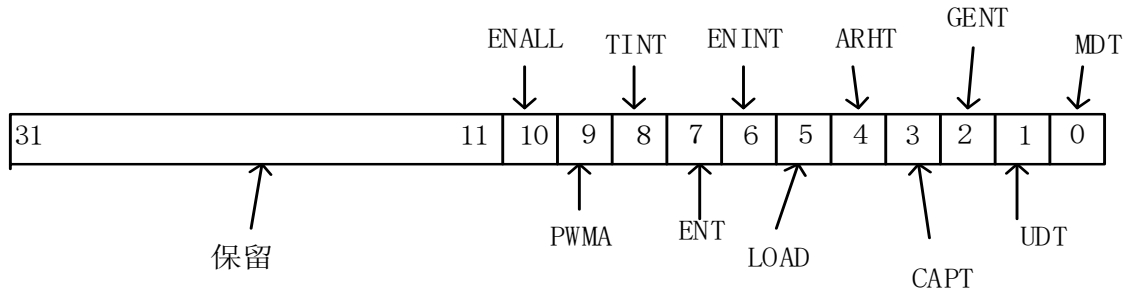
8.6 AXI Timer接口设计

▶ AXI Timer的寄存器

- TCSR寄存器

名称	含义	位置	读	写
MDT	工作模式	Bit0	设置值	1 capture 模式（计数）；0 Generate模式（定时）
UDT	计数方式	Bit1	设置值	1 减计数；0 加计数
GENT	使能GenerateOut输出	Bit2	设置值	0 不允许比较输出；1 允许比较输出
CAPT	Capture trig外部触发信号使能	Bit3	设置值	0关闭外部触发信号；1使能外部触发信号
ARHT	自动装载	Bit4	设置值	1 TCR自动装载TLR的值；0 TCR保持不变
LOAD	装载命令	Bit5	设置值	0不装载TLR到TCR；1装载TLR到TCR
ENINT	中断使能	Bit6	设置值	1产生中断输出；0不产生中断输出
ENT	定时器使能	Bit7	设置值	1定时器运行；0 定时器停止
TINT	定时器中断状态	Bit8	1，中断 0，无	1清除中断状态；0无影响
PWMA	脉宽调制使能	Bit9	设置值	0使脉宽调制输出无效 1且MDT0,MDT1必须为0，GENT0,GENT1也同时为1时，脉宽输出调致有效，
ENALL	所有定时器使能	Bit10	设置值	1使能所有定时器，写0则清除ENALL位，对ENT0,ENT1无影响
保留		其余位		





8.6 AXI Timer接口设计

▶ AXI Timer的寄存器

- TCSR寄存器

名称	含义	位置	读	写
MDT	工作模式	Bit0	设置值	1 capture 模式（计数）； 0 Generate模式（定时）
UDT	计数方式	Bit1	设置值	1 减计数；0 加计数
GENT	使能GenerateOut输出	Bit2	设置值	0 不允许比较输出；1 允许比较输出
CAPT	Capture trig外部触发信号使能	Bit3	设置值	0关闭外部触发信号；1使能外部触发信号
ARHT	自动装载	Bit4	设置值	1 TCR自动装载TLR的值；0 TCR保持不变
LOAD	装载命令	Bit5	设置值	0不装载TLR到TCR；1装载TLR到TCR
ENINT	中断使能	Bit6	设置值	1产生中断输出；0不产生中断输出
ENT	定时器使能	Bit7	设置值	1定时器运行；0 定时器停止
TINT	定时器中断状态	Bit8	1，中断 0，无	1清除中断状态；0无影响
PWMA	脉宽调制使能	Bit9	设置值	0使脉宽调制输出无效 1且MDT0,MDT1必须为0， GENT0,GENT1也同时为1时，脉宽输出调致有效，
ENALL	所有定时器使能	Bit10	设置值	1使能所有定时器，写0则清除ENALL位，对ENT0,ENT1无影响
保留		其余位		

▶ 控制定时器定时产生中断的基本流程

- 初始化定时器
  - 停止定时器，写TCSR使ENT=0;
  - 清除中断标志，写TINT=1;
  - 清除MDT,使其为0
  - 设置UDT为0或1，进行加或减计数；
  - 设置ARHT=1，控制定时器计数结束时自动装载预置值
  - 使能中断，写TCSR使ENINT=1;
  - 写TLR，配置计数初始值
  - 装载TCR，写TCSR使LOAD=1;
- 运行定时器，写TCSR使ENT=1，LOAD=0；这样定时器就以用户设置的初始值开始计数。

8.6 AXI Timer接口设计

► 控制定时器定时产生中断的基本流程

- 初始化定时器
  - 停止定时器，写TCSR使ENT=0;
  - 清除中断标志，写TINT=1;
  - 清除MDT,使其为0
  - 设置UDT为0或1，进行加或减计数；
  - 设置ARHT=1，控制定时器计数结束时自动装载预置值
  - 使能中断，写TCSR使ENINT=1;
  - 写TLR，配置计数初始值
  - 装载TCR，写TCSR使LOAD=1;
- 运行定时器，写TCSR使ENT=1，LOAD=0；这样定时器就以用户设置的初始值开始计数。

► 定时器API

- XTmrCtr_Initialize(XTmrCtr*, u16) : int
- XTmrCtr_Start(XTmrCtr*, u8) : void
- XTmrCtr_Stop(XTmrCtr*, u8) : void
- XTmrCtr_GetValue(XTmrCtr*, u8) : u32
- XTmrCtr_SetResetValue(XTmrCtr*, u8, u32) : void
- XTmrCtr_GetCaptureValue(XTmrCtr*, u8) : u32
- XTmrCtr_IsExpired(XTmrCtr*, u8) : int
- XTmrCtr_Reset(XTmrCtr*, u8) : void
- XTmrCtr_LookupConfig(u16) : XTmrCtr_Config*
- XTmrCtr_SetOptions(XTmrCtr*, u8, u32) : void
- XTmrCtr_GetOptions(XTmrCtr*, u8) : u32
- XTmrCtr_GetStats(XTmrCtr*, XTmrCtrStats*) : void
- XTmrCtr_ClearStats(XTmrCtr*) : void
- XTmrCtr_SelfTest(XTmrCtr*, u8) : int
- XTmrCtr_SetHandler(XTmrCtr*, XTmrCtr_Handler, void*) : void
- XTmrCtr_InterruptHandler(void*) : void

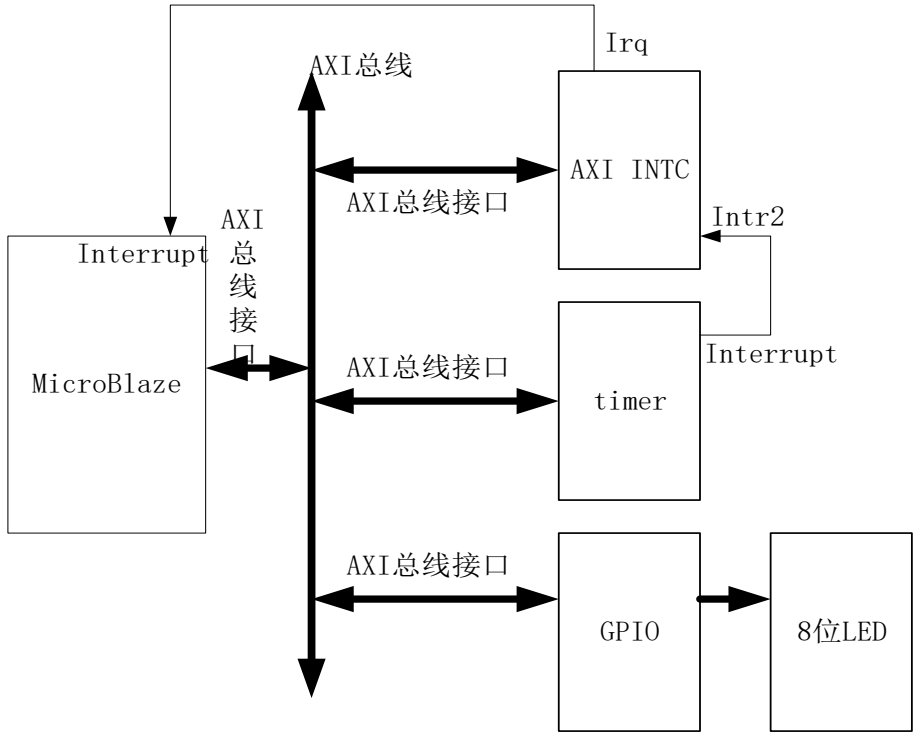
8.6 AXI Timer接口设计

▶ 定时器中断程序设计实例

- 基于MicroBlaze微处理器AXI总线设计硬件接口电路以及控制程序，要求微处理器控制8位LED灯轮流亮灭，且轮流间隔1秒钟。采用硬件定时器中断方式实现定时。
  - AXI timer时钟信号来自AXI总线时钟AXI_CLK。若AXI_CLK=100MHz，那么定时1s，就需要计100M个时钟脉冲。如果采用减计数，TLR的初始值就是100M；如果采用加计数，TLR的初始值就是2^32−1−100M。
  - 头文件

```
#include "xparameters.h"   // The hardware configuration describing
                           // constants
#include <stdio.h>
#include "xil_io.h"
#include "xil_printf.h"
#include "mb_interface.h"
#include "xintc_l.h"
#include "xtmrctr_l.h"

#define Led_DATA  0x40000008          // Leds数据寄存器地址
#define Led_TRI   0x4000000C          // Leds控制寄存器地址
```



8.6 AXI Timer接口设计

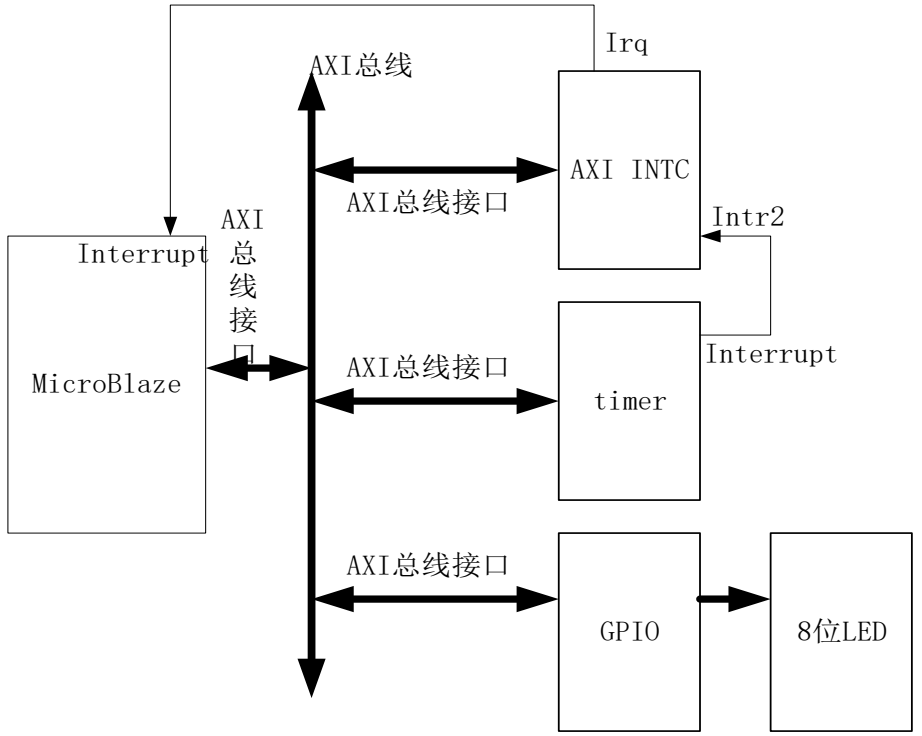
▶ 定时器中断程序设计实例

- 基于MicroBlaze微处理器AXI总线设计硬件接口电路以及控制程序，要求微处理器控制8位LED灯轮流亮灭，且轮流间隔1秒钟。采用硬件定时器中断方式实现定时。
  - AXI timer时钟信号来自AXI总线时钟AXI_CLK。若AXI_CLK=100MHz，那么定时1s，就需要计100M个时钟脉冲。如果采用减计数，TLR的初始值就是100M；如果采用加计数，TLR的初始值就是2^32−1−100M。
  - 符号常量

```
#define Led_DATA  0x40000008        // Leds数据寄存器地址
#define Led_TRI   0x4000000C        // Leds控制寄存器地址

#define intc_ISR     0x41200000
#define    intc_IER  0x41200008
#define intc_IAR     0x4120000C
#define    intc_MER  0x4120001C

#define T0_RESET_VALUE 100000000 //1s
// 减计数，TLR的初始值就是100M = 10e8 = 0x5F5E100
//注册总中断服务程序
void My_ISR(void) __attribute__((interrupt_handler));
```



8.6 AXI Timer接口设计

▶ 定时器中断程序设计实例

- 基于MicroBlaze微处理器AXI总线设计硬件接口电路以及控制程序，要求微处理器控制8位LED灯轮流亮灭，且轮流间隔1秒钟。采用硬件定时器中断方式实现定时。
  - AXI timer时钟信号来自AXI总线时钟AXI_CLK。若AXI_CLK=100MHz，那么定时1s，就需要计100M个时钟脉冲。如果采用减计数，TLR的初始值就是100M；如果采用加计数，TLR的初始值就是2^32−1−100M。
  - 函数及全局变量声明

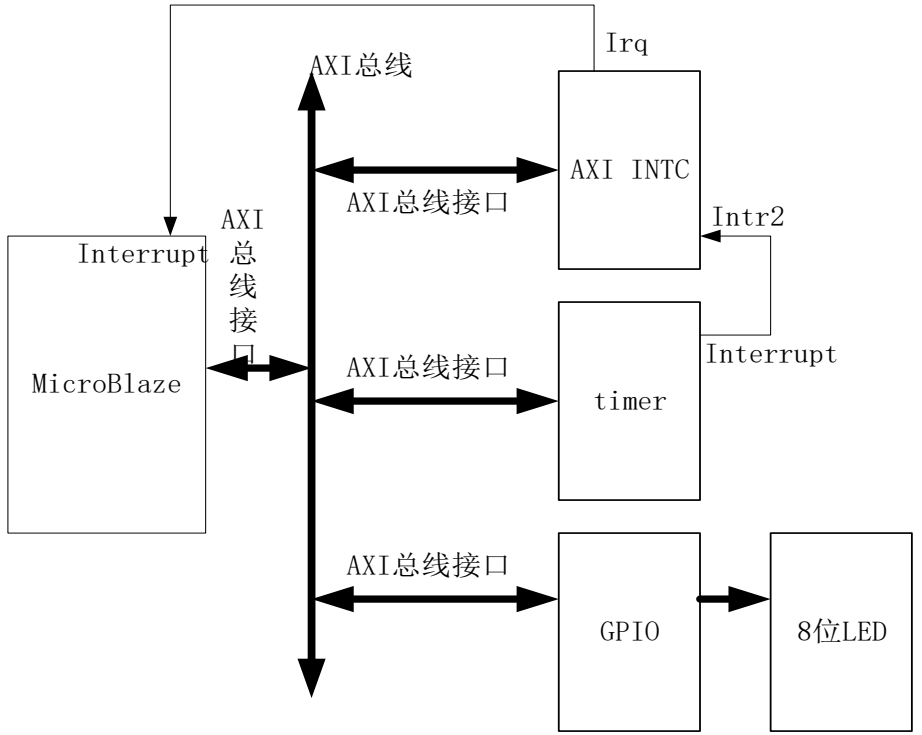
```
#define T0_RESET_VALUE 100000000 //1s

//注册总中断服务程序
void My_ISR(void) __attribute__((interrupt_handler));

void TimerCh0Handler(void);   // 定时器通道0中断服务子函数
void Initialize(void);        // 硬件初始化子函数

short int t_1s_Flag;

int main(void)
{
```



▶ 定时器中断程序设计实例

- 主函数

```
int main(void)
{
    unsigned char LedBits;
    xil_printf("\r\nRunning Timer Test(No API)\r\n");
    Initialize();
    LedBits = 0;
    while(1)
    {
            if(t_1s_Flag == 0x01)                                   // 分支处理
            {
                    t_1s_Flag = 0x00;
                    Xil_Out16(Led_DATA,1<<LedBits);         // //产生中断时，输出LED显示值
                    LedBits++;
                    if(LedBits==8)          LedBits = 0;
            }
    }
    return 0;
}
```

▶ 定时器中断程序设计实例

- GPIO、Timer、CPU初始化函数

```
void Initialize(void)
{
  t_1s_Flag = 0;
  int tcsr0;
  Xil_Out16(Led_TRI,0x0000);                    // Led is used as output;

  tcsr0=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET);
  Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,tcsr0&(~XTC_CSR_ENABLE_TMR_MASK));  // ENT = 0, 停止T
  Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TLR_OFFSET,T0_RESET_VALUE);      // 写LDR, 配置计数初始值

  tcsr0=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET);
  Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,tcsr0|XTC_CSR_LOAD_MASK);     // LODA = 1，装载TLR到TCR
  Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
          tcsr0| ((XTC_CSR_ENABLE_TMR_MASK|XTC_CSR_DOWN_COUNT_MASK              // ENT =1，运行定时器，UDT = 1, 减计数
                  |XTC_CSR_AUTO_RELOAD_MASK)&(~XTC_CSR_LOAD_MASK)));    // 自动装载：ARHT=1,LOAD =0;

  Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
          Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET )|
          XTC_CSR_INT_OCCURED_MASK| XTC_CSR_ENABLE_INT_MASK);  // 清除中断标志：TINT=1;允许TIMER中断：ENINT=1；
```

▶ 定时器中断程序设计实例

- GPIO、Timer、CPU初始化函数
-

```
void Initialize(void)
{
  …
  …
  Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
            Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET )|
            XTC_CSR_INT_OCCURED_MASK| XTC_CSR_ENABLE_INT_MASK);  // 清除中断标志：TINT=1;允许TIMER中断：ENINT=1；

    Xil_Out32(intc_IAR,0xffffffff);          // 清零所有中断标志
    Xil_Out32(intc_IER,0x04);                // Intr[2]通道中断允许
    Xil_Out32(intc_MER,0x03);                // 允许INTC模块中断

    microblaze_enable_interrupts();          // 允许CPU中断
}
```

▶ 定时器中断程序设计实例

- 总中断服务函数、定时器中断函数

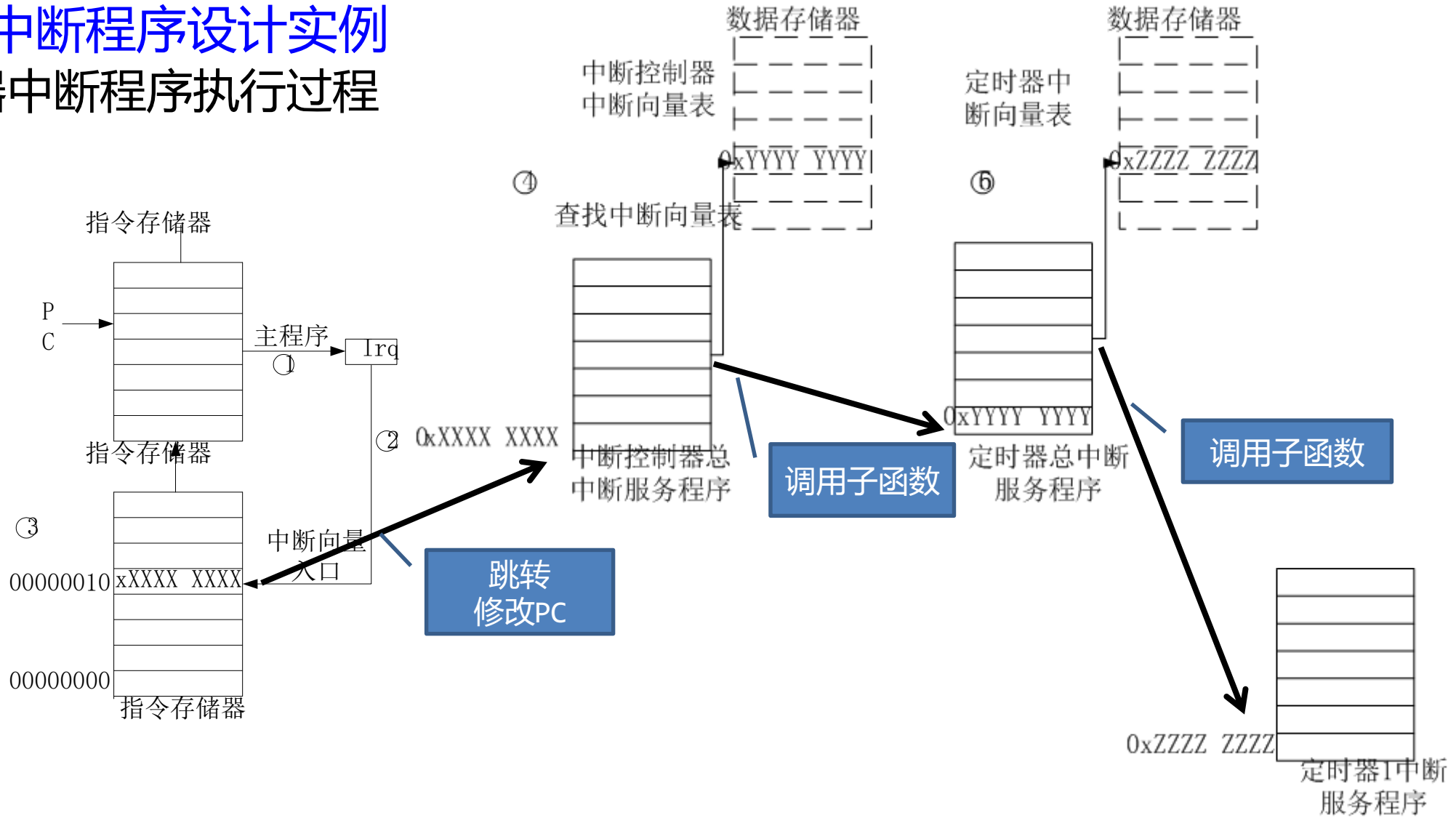
```
void My_ISR(void)
{
        int intc_Status;
        intc_Status = Xil_In32(intc_ISR);          // 读取INTC的ISR
        xil_printf("\r\nintc_Status = %X",intc_Status);
        if(intc_Status&0x04)                       // ISR[2]=1，说明有Timer中断
        {
                TimerCh0Handler();
        }
        Xil_Out32(intc_IAR,intc_Status);           // 写INTC的IAR，清零INTC的ISR
}

void TimerCh0Handler(void)
{
        t_1s_Flag = 0x01;
        Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
                Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET )|
                XTC_CSR_INT_OCCURED_MASK| XTC_CSR_ENABLE_INT_MASK); // 清除中断标志：写TINT=1;
}
```

【思考】若Timer的两个通道Ch0定时1s、Ch1定时4ms：
（1）如何写Timer的初始化代码？
（2）在总中断服务函数中，如何判断Timer的Ch0、Ch1中断源？

8.6 AXI Timer接口设计

- 定时器中断程序设计实例
  - 定时器中断程序执行过程



Agenda

- 内容
  - 中断的基本概念，中断响应过程
  - 典型微处理器中断系统简介
  - Xilinx的中断控制器-AXI INTC
  - GPIO中断方式接口设计
  - AXI Timer接口
  - AXI SPI接口
- 目的
  - 理解Interrupt 的含义，优点、分类；
  - 理解中断源、中断请求、中断类型码、中断优先级、中断向量入口地址(Interrupt Vector Address)、中断向量表等术语的含义和作用；
  - 理解CPU响应中断的过程；
  - 理解X86和Microbalze系统的中断处理过程；
  - 掌握AXI INTC原理，学会Microbalze系统中断程序设计；
  - 掌握AXI Timer接口设计；
  - 掌握AXI SPI接口设计。

8.7 AXI SPI接口

▶ SPI总线

- SPI，英文全称为Serial Peripheral Interface，串行外围接口的意思，是Motorola公司开发的一种全双工同步串行总线。
- 有很多外围芯片，例如存储器、A/D或D/A变换器，LCD等均采用SPI接口。
- SPI使用4个信号来收发数据。

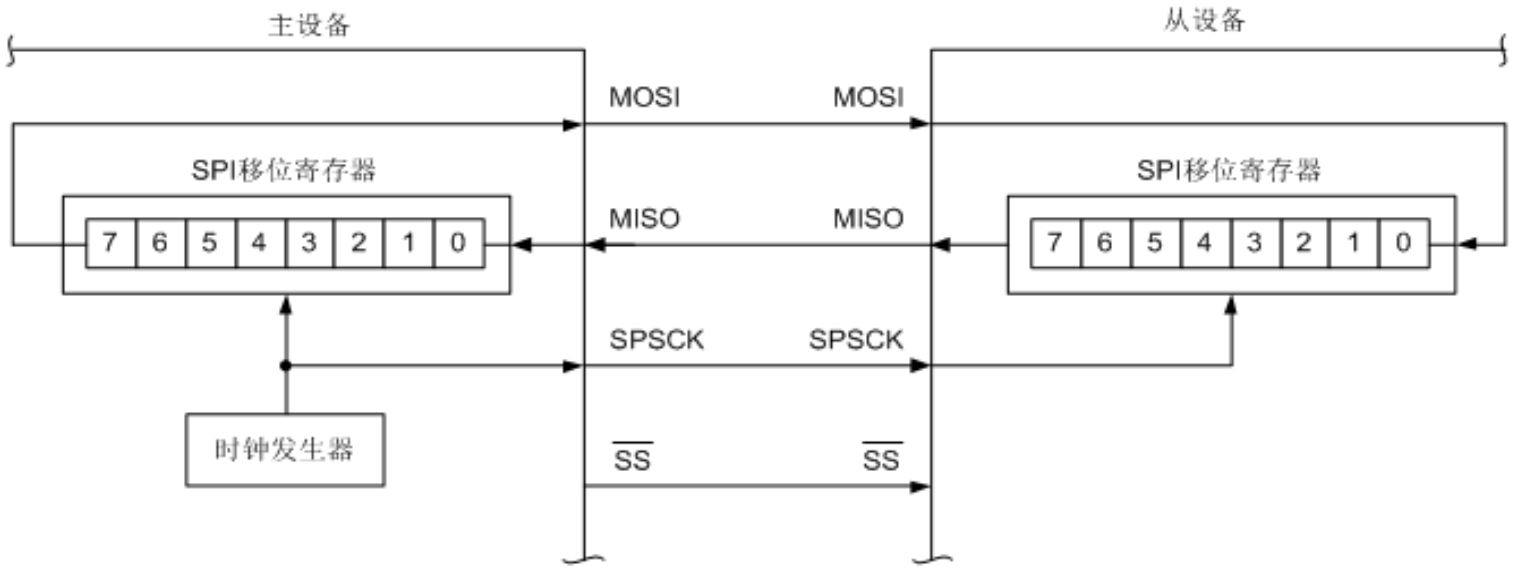
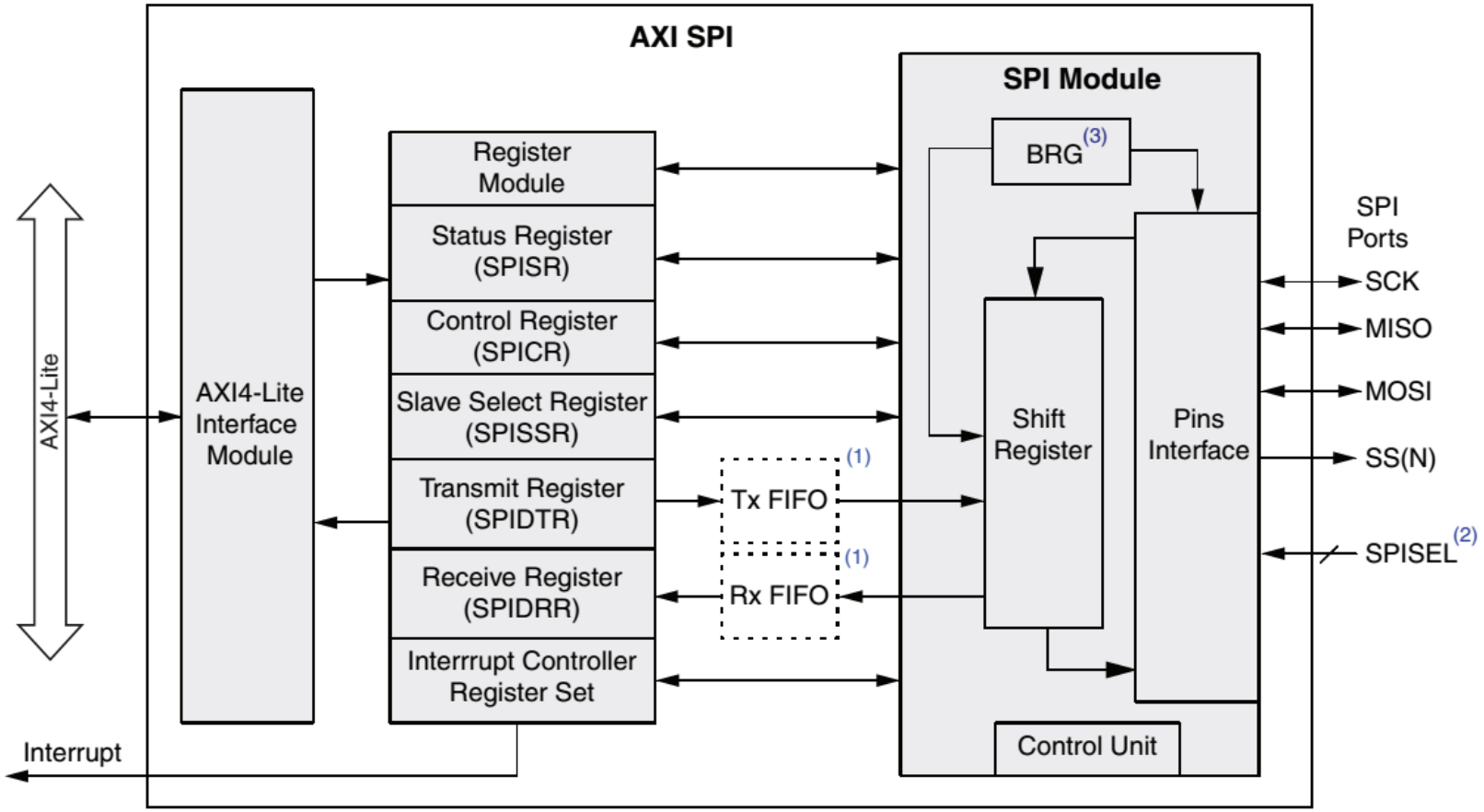


图 12-1　SPI 系统连接

8.7 AXI SPI接口

▶ AXI SPI总线接口简介

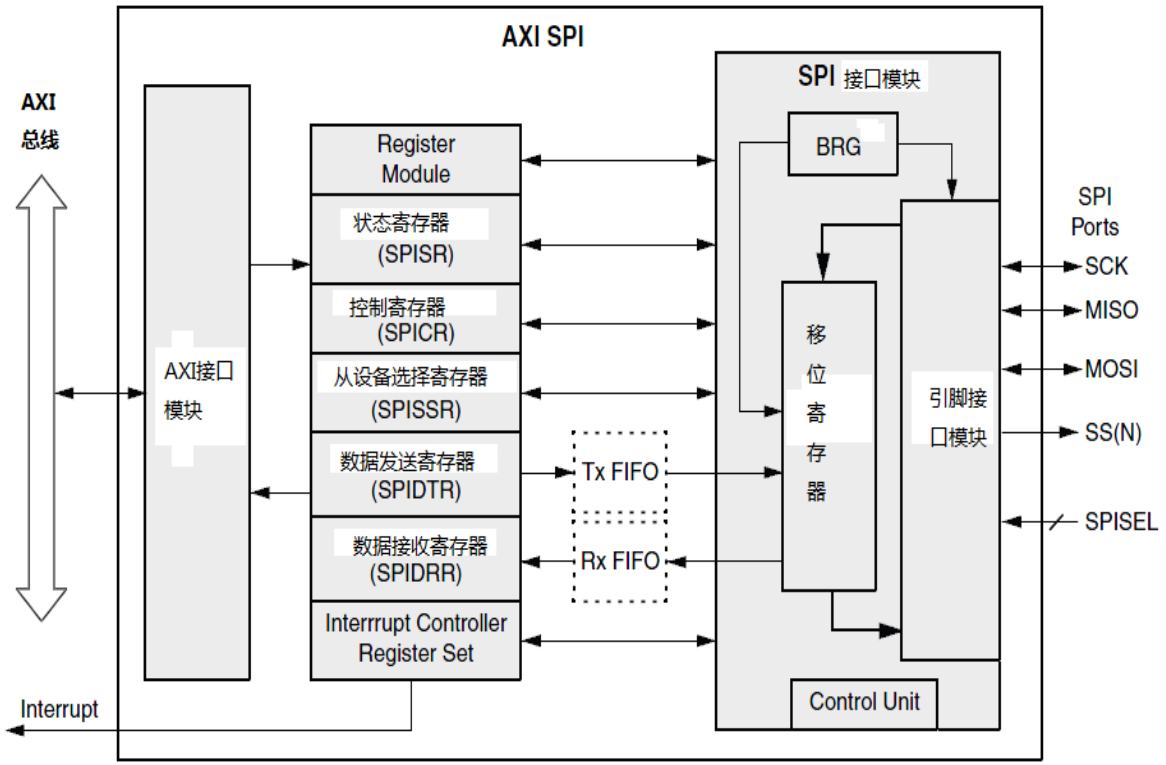


Notes:
1. The width of Tx FIFO, Rx FiFO, and Shift Register depends on the value of the generic, C_NUM_TRANSER_BITS.
2. The width of SS depends on the value of the generic C_NUM_SS_BITS.
3. BRG (Buad Rate Generator)

DS742_01

▶ AXI SPI 寄存器

寄存器名称	偏移地址	含义	操作类型	初始值
SRR	40	软件复位寄存器，向该寄存器写0x0000000A，复位接口	写	N/A
SPICR	60	控制寄存器	读写	0x180
SPISR	64	状态寄存器	读	0x25
SPIDTR	68	发送寄存器或FIFO（可为8，16，32位）	写	0x0
SPIDRR	6C	接收寄存器或FIFO（可为8，16，32位）	读	N/A
SPISSR	70	从设备选择寄存器	读写	未选中
Tx_FIFO_OCY	74	发送FIFO占用长度指示，低4位的值+1表示FIFO有效数据的长度	读	0x0
Rx_FIFO_OCY	78	接收FIFO占用长度指示，低4位的值+1表示FIFO有效数据的长度	读	0x0
DGIER	1C	设备总中断请求使能寄存器，仅最高位有效，bit31=1使能设备中断	读写	0x0
IPISR	20	中断状态寄存器	读/写	0x0
IPIER	28	中断使能寄存器	读写	0x0



8.7 AXI SPI接口

▶ AXI SPI 寄存器

- 控制寄存器SPICR

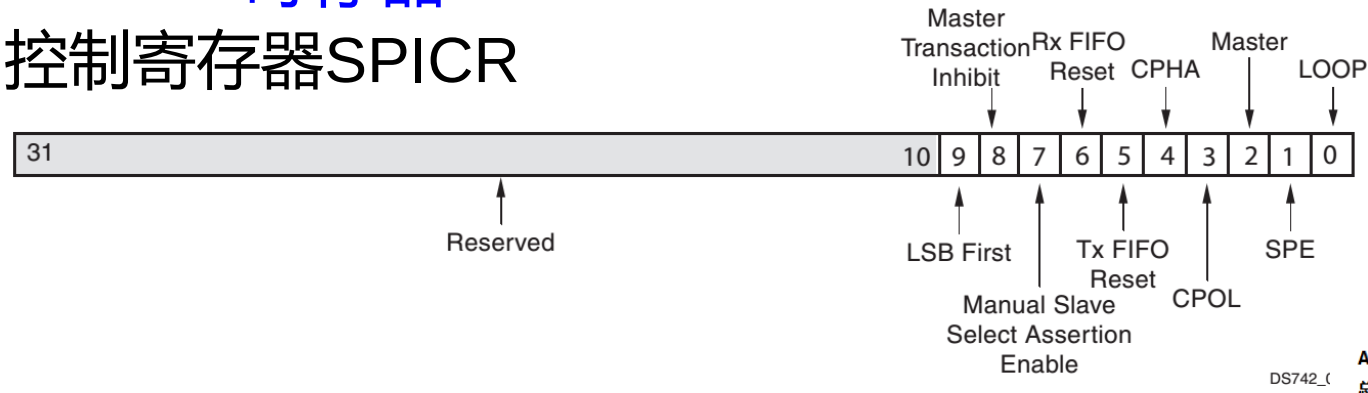
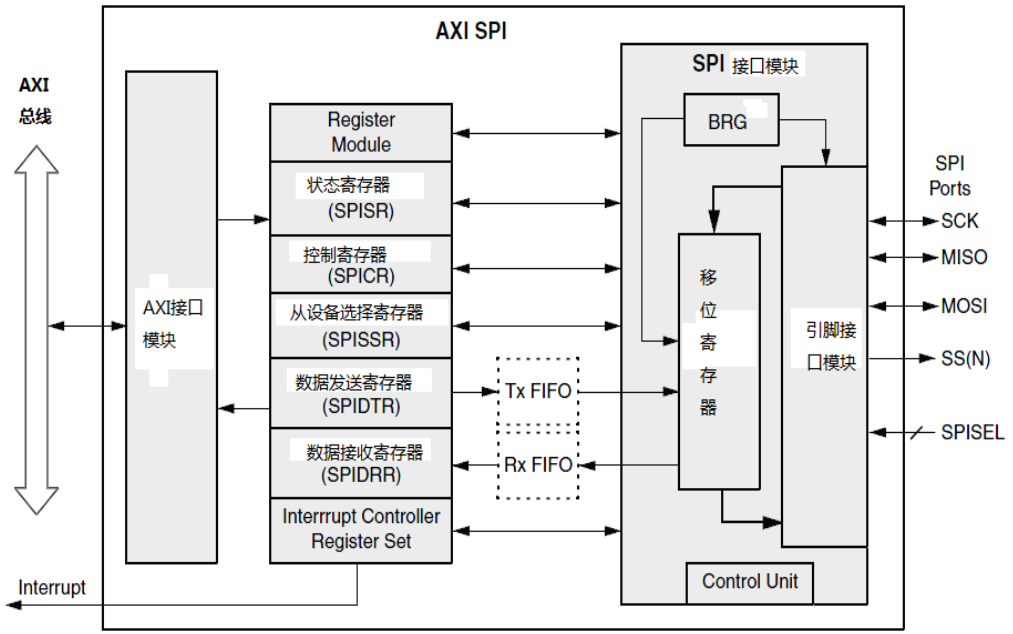


Figure 3: **SPI Control Register (C_BASEADDR + 0x60)**

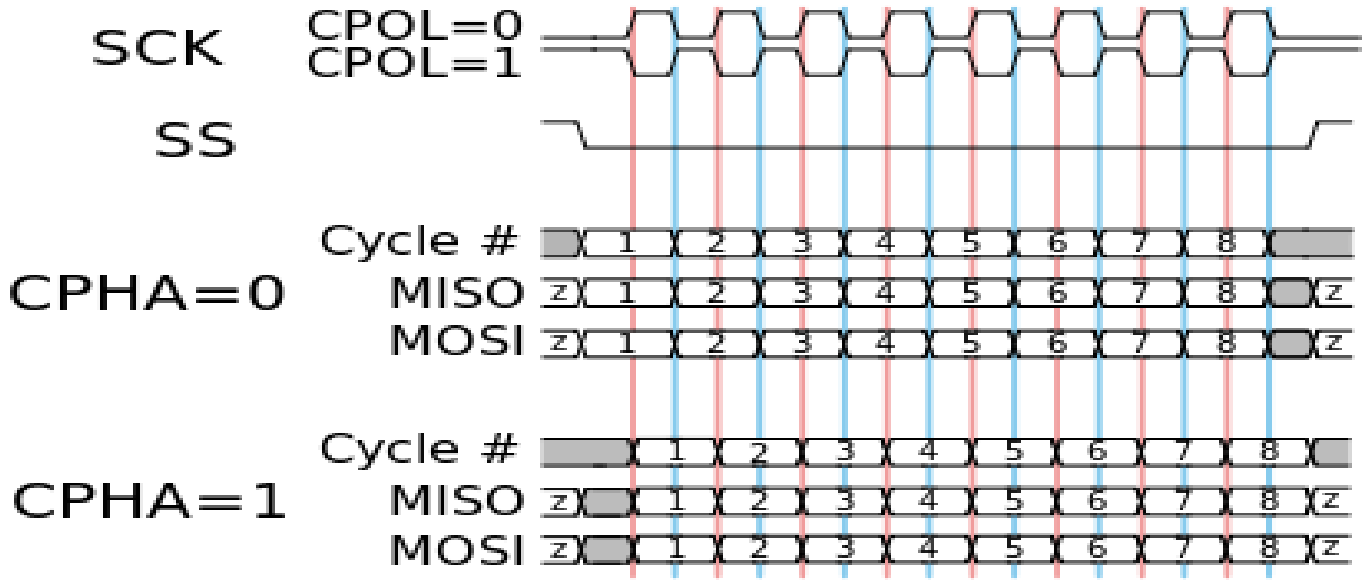
Bit 位置	写1	写0
0	SPI发送端与接收端在内部形成环路	正常工作
1	使能SPI接口	停止SPI接口
2	配置为主设备	配置为从设备
3	空闲时时钟为高电平	空闲时时钟为低电平
4	数据在第二个时钟周期开始有效	数据在第一个时钟开始有效
5	复位发送FIFO指针	无影响
6	复位接收FIFO指针	无影响
7	配置为手动控制，根据SPISSR寄存器的值输出SS	根据内部逻辑自动输出SS
8	禁止主设备事务；若为从设备则无影响	使能主设备事务
9	串行数据低位优先传送	串行数据高位优先传送



8.7 AXI SPI接口

AXI SPI 寄存器

- 控制寄存器SPICR：CPOL以及CPHA组合可以设定SPI总线时序
  - 当CPOL=0，CPHA=1时，空闲时SCLK为低电平，数据输出端在SCLK的上升沿转换数据，数据输入端在SCLK下降沿采样数据（即第二个时钟周期才采样数据）
  - 当CPOL=0，CPHA=0时，空闲时SCLK为低电平，数据输出端在SCLK下降沿转换数据，数据输入端在SCLK上升沿采样数据（即第一个时钟周期采样数据）
  - 当CPOL=1，CPHA=1时，空闲时SCLK为高电平，数据输出端在SCLK的下降沿转换数据，数据输入端在SCLK上升沿采样数据（即第二个时钟周期才采样数据）
  - 当CPOL=1，CPHA=0时，空闲时SCLK为高电平，数据输出端在SCLK上升沿转换数据，数据输入端在SCLK下降沿采样数据（即第一个时钟周期采样数据）



8.7 AXI SPI接口

▶ AXI SPI 寄存器

- 状态寄存器SPISR

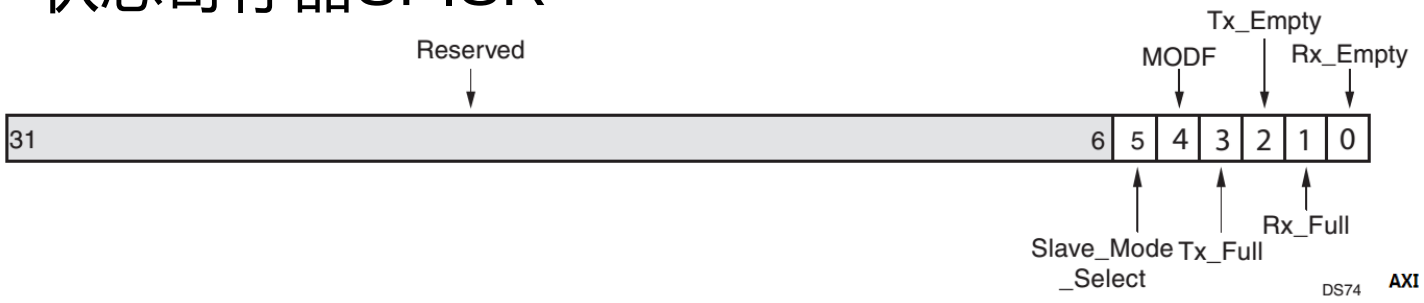
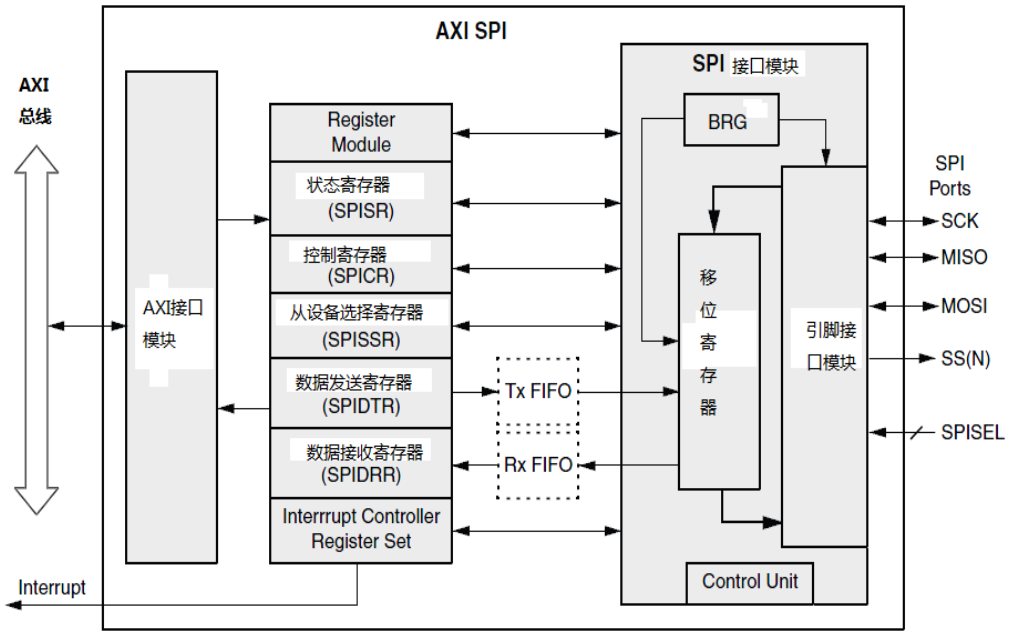


Figure 4: **SPI Status Register (C_BASEADDR + 0x64)**

Bit 位置	写1	写0
0	接收寄存器/FIFO非空	接收寄存器/FIFO空
1	接收寄存器/FIFO未满	接收寄存器/FIFO满
2	发送寄存器/FIFO非空	发送寄存器/FIFO空
3	发送寄存器/FIFO未满	发送寄存器/FIFO满
4	没有模式错误	SPI设备配置为主设备，但是SS引脚输入低电平
5	从设备选中	默认值，未被选中



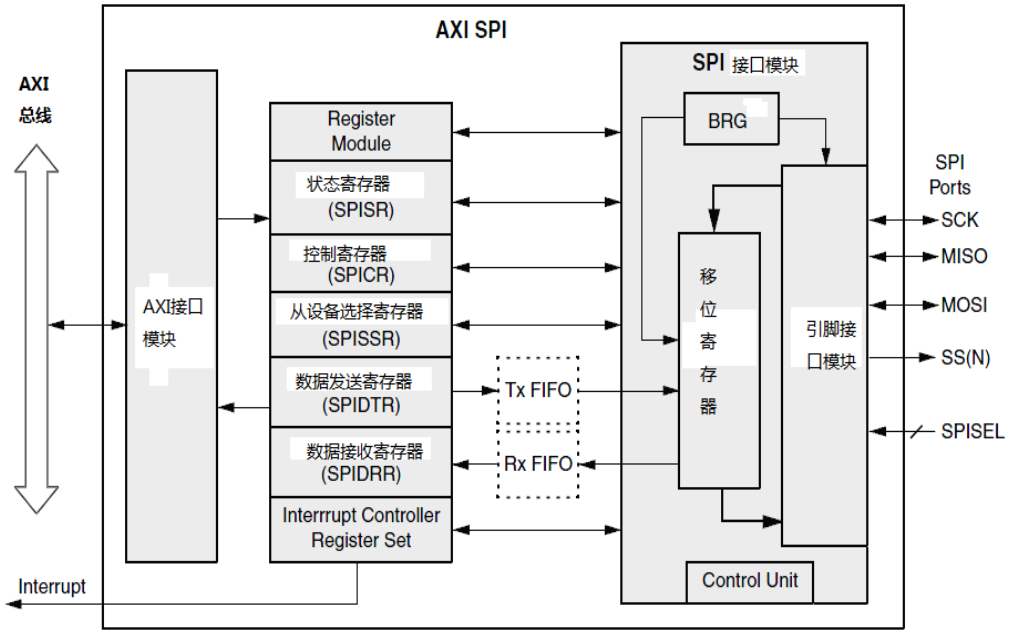
8.7 AXI SPI接口

► AXI SPI 寄存器

- 从设备选择寄存器SPISSR

Figure 7: **SPI Slave Select Register (C_BASEADDR + 0x70)**

  - SPISSR寄存器bit0~bitn-1分别对应控制SS(0~n-1)的输出



- AXI SPI 寄存器
  - 中断控制寄存器组
    - 中断状态寄存器IPISR

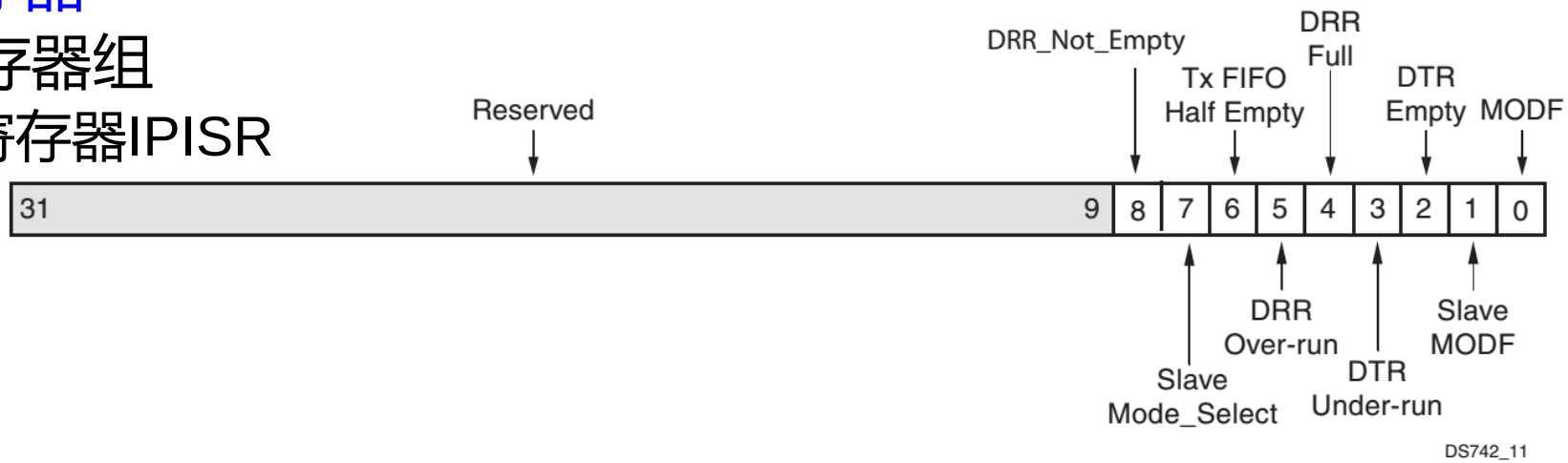


Figure 11: **IP Interrupt Status Register (IPISR) (C_BASEADDR + 0x20)**

    - 中断允许寄存器IPIER

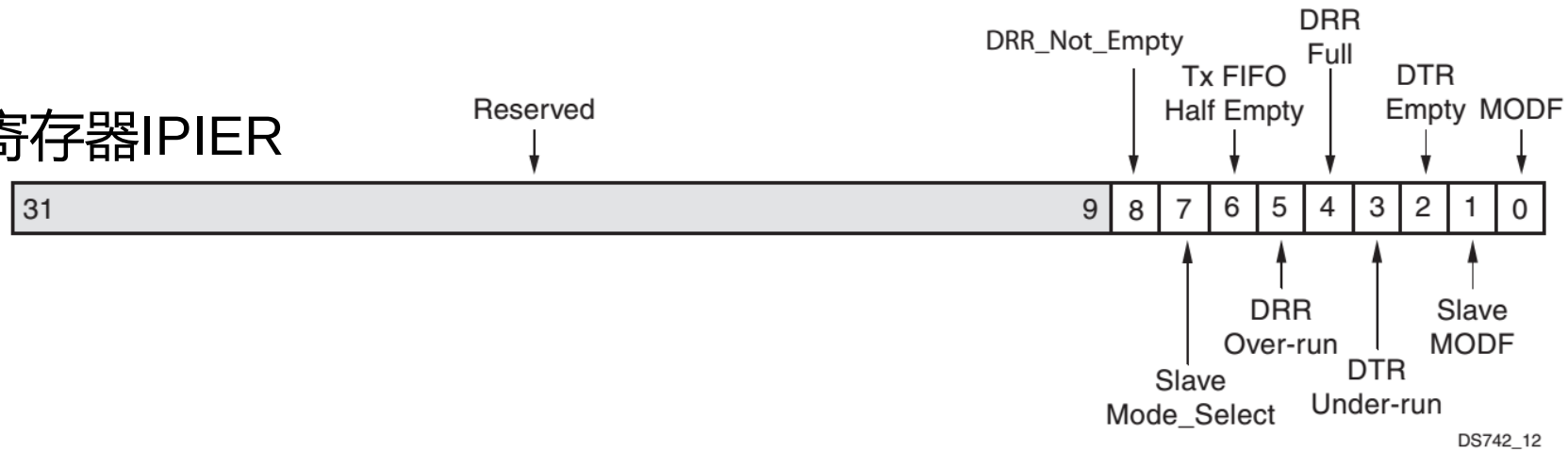


Figure 12: **IP Interrupt Enable Register (IPIER) (C_BASEADDR + 0x28)**

▶ SPI 典型控制流程

- SPI总线接口作为主设备并采用手动控制SS（N），应用时序1）发送数据的流程
  - 根据应用需要配置DGIER以及IPIER，实现中断使能控制；
  - 将要传输的数据写入SPIDTR寄存器或FIFO；
  - 将SPISSR预置为全1
  - 配置SPICR，初始化SCLK以及MOSI，但禁止数据传输
    - SPICR（bit7）=1，手动控制SS（N）
    - SPICR（bit1）=1，使能SPI
    - SPICR（bit8）=1，禁止主设备事务
    - SPICR（bit2）=1，配置为主设备
    - SPICR（bit3）=0，空闲时时钟为低电平
    - SPICR（bit4）=1，上升沿输出数据，下降沿采样数据
    - SPICR（bit0）=0，非内部循环
  - 写SPISSR，控制SS（N）输出信号的使能信号
  - 修改SPICR（bit8）=0，使能主设备事务，从而开始SPI数据传输
  - 等待中断或查询SPI状态

SPI 典型控制流程（续）

- SPI总线接口作为主设备并采用手动控制SS（N），应用时序1）发送数据的流程
  - 根据应用需要配置DGIER以及IPIER，实现中断使能控制；
  - 将要传输的数据写入SPIDTR寄存器或FIFO；
  - 将SPISSR预置为全1
  - 配置SPICR，初始化SCLK以及MOSI，但禁止数据传输
  - 写SPISSR，控制SS（N）输出信号的使能信号
  - 修改SPICR（bit8）=0，使能主设备事务，从而开始SPI数据传输
  - 等待中断或查询SPI状态
  - 进入中断服务，修改SPICR（bit8）=1，禁止主设备事务，将新的数据写入数据寄存器或FIFO之后，再修改SPICR（bit8）=0，使能主设备事务，从而开始SPI数据传输
  - 重复6），7）直到所有数据传送完毕
  - 写SPISSR，控制SS（N）输出全1
  - 禁止SPI设备。

8.7 AXI SPI接口

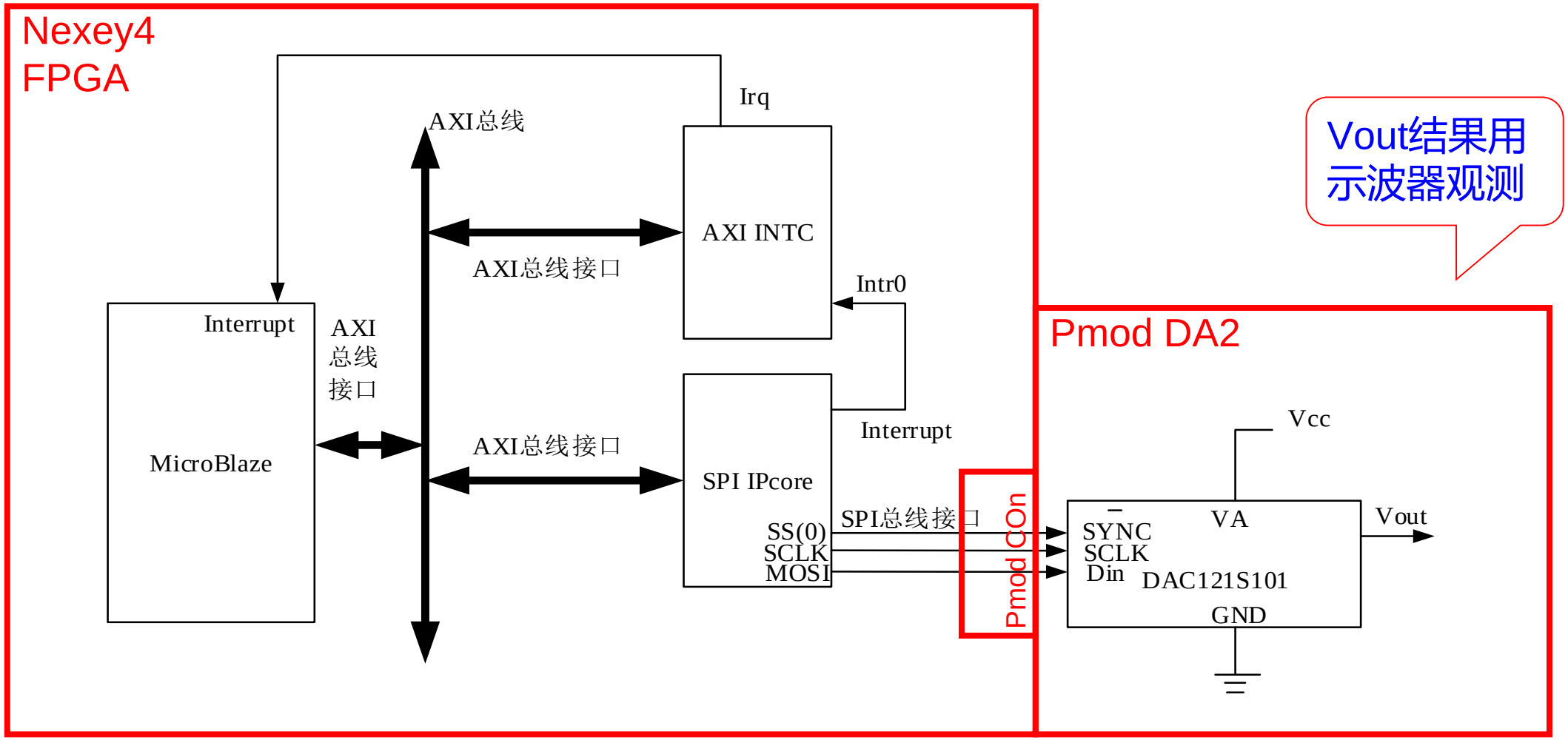
▶ SPI API

XSpi_IntrGlobalEnable
XSpi_IntrGlobalDisable
XSpi_IsIntrGlobalEnabled
XSpi_IntrGetStatus
XSpi_IntrClear
XSpi_IntrEnable
XSpi_IntrDisable
XSpi_IntrGetEnabled
XSpi_SetControlReg
XSpi_GetControlReg
XSpi_GetStatusReg
XSpi_SetSlaveSelectReg
XSpi_GetSlaveSelectReg
XSpi_Enable
XSpi_Disable

XSpi_Initialize(XSpi*, u16) : int
XSpi_LookupConfig(u16) : XSpi_Config*
XSpi_CfgInitialize(XSpi*, XSpi_Config*, u32) : int
XSpi_Start(XSpi*) : int
XSpi_Stop(XSpi*) : int
XSpi_Reset(XSpi*) : void
XSpi_SetSlaveSelect(XSpi*, u32) : int
XSpi_GetSlaveSelect(XSpi*) : u32
XSpi_Transfer(XSpi*, u8*, u8*, unsigned int) : int
XSpi_SetStatusHandler(XSpi*, void*, XSpi_StatusHandler) : void
XSpi_InterruptHandler(void*) : void
XSpi_SelfTest(XSpi*) : int
XSpi_GetStats(XSpi*, XSpi_Stats*) : void
XSpi_ClearStats(XSpi*) : void
XSpi_SetOptions(XSpi*, u32) : int
XSpi_GetOptions(XSpi*) : u32

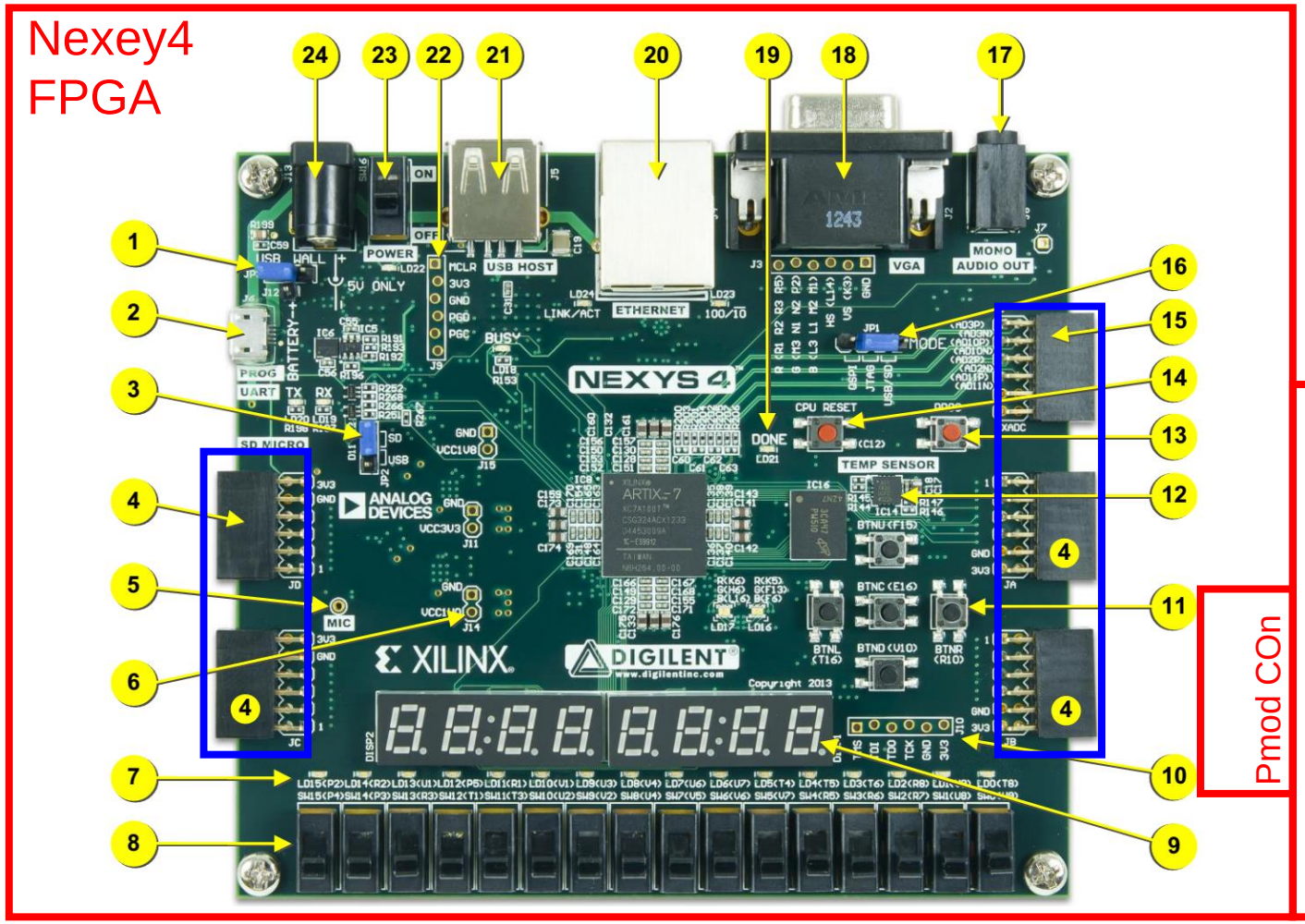
8.7 AXI SPI接口

- 利用DAC121S101DA转换芯片，基于SPI总线控制其Dout输出锯齿波

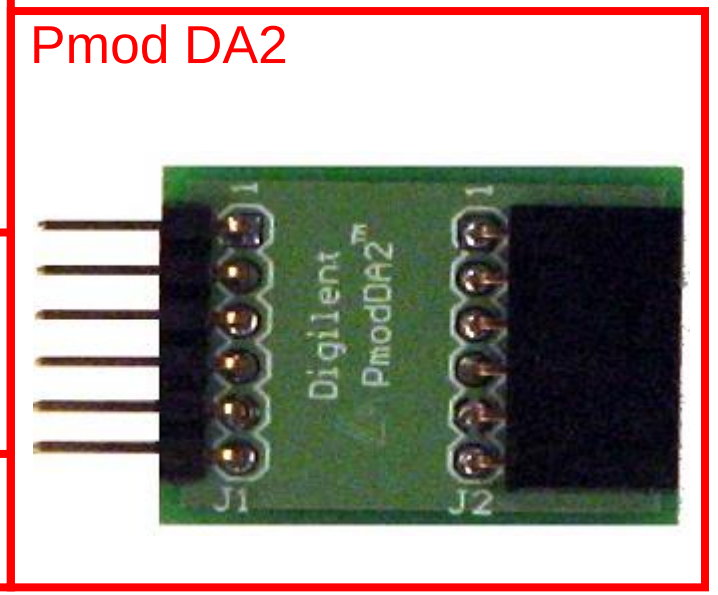


8.7 AXI SPI接口

- 利用DAC121S101DA转换芯片，基于SPI总线控制其Dout输出锯齿波



Vout结果用
示波器观测

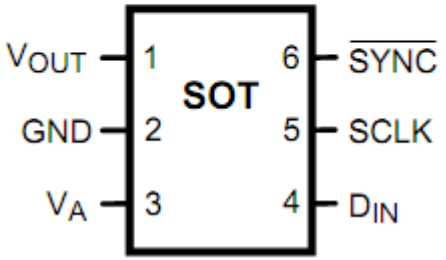


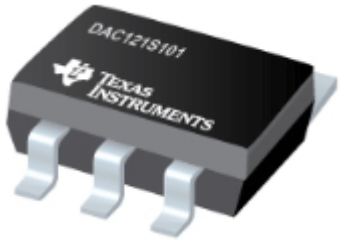
8.7 AXI SPI接口

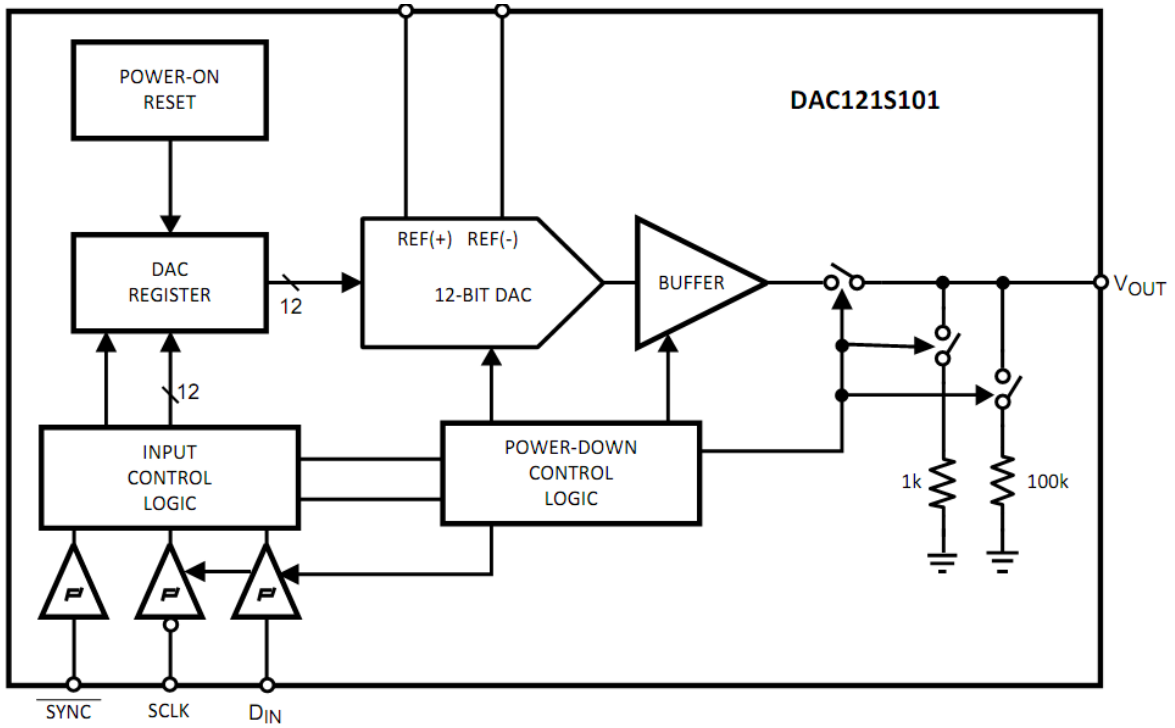
▶ DAC121S101 概述

管脚	含义
V_{OUT}	模拟电压输出
GND	地
V_A	模拟参考电压
$\overline{SYNC}$	帧数据同步，当该引脚为低电平时，数据在SCLK的下降沿输入，并且16个时钟周期之后，移位寄存器的数据进入DAC寄存器，开始DA转换；若该引脚在16个时钟周期之前变为高电平，那么之前传入的数据都将被忽略。
SCLK	SPI总线时钟，数据在该时钟的下降沿采样。时钟频率最高为30MHZ
D_{IN}	SPI总线从设备数据输入线，相当于MOSI

DDC Package
6-Pin SOT
Top View





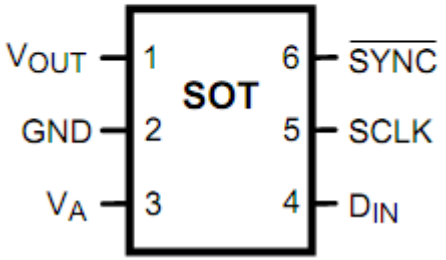


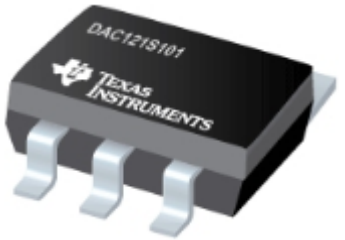
8.7 AXI SPI接口

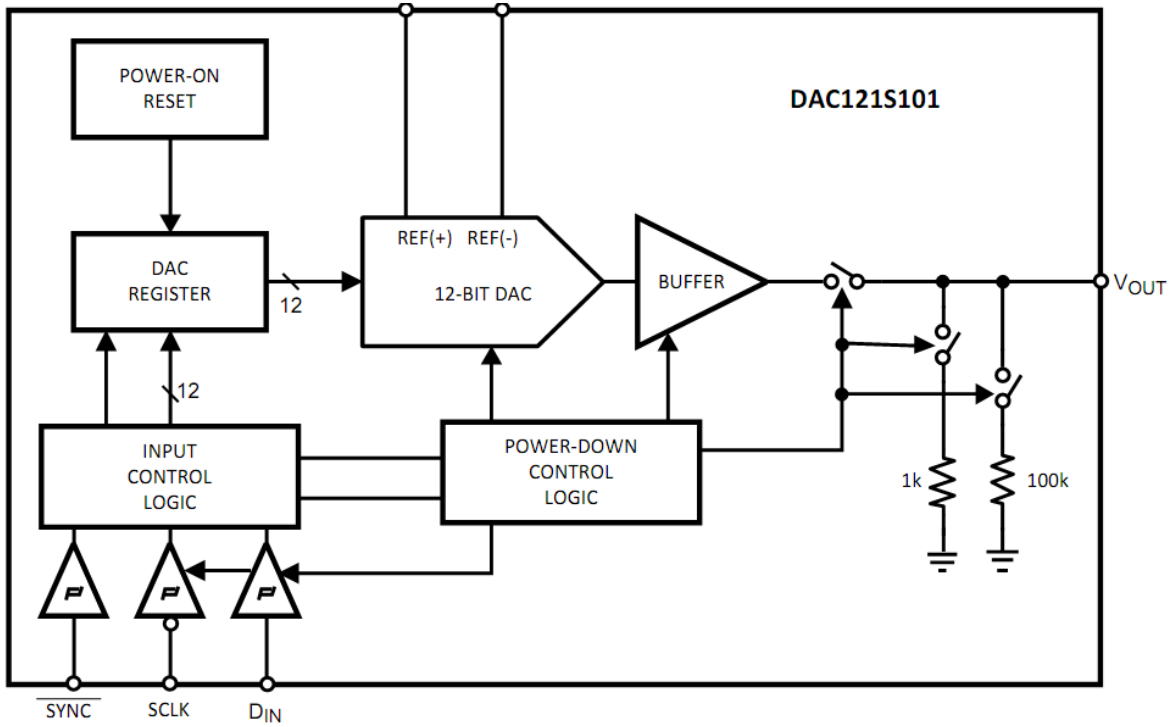
▶ DAC121S101 概述

- The input coding is straight binary with an ideal output voltage of:
  VOUT = VA × (D / 4096)
  - where D is the decimal equivalent of the binary code that is loaded into the DAC register and can take on any value between 0 and 4095.

DDC Package
6-Pin SOT
Top View

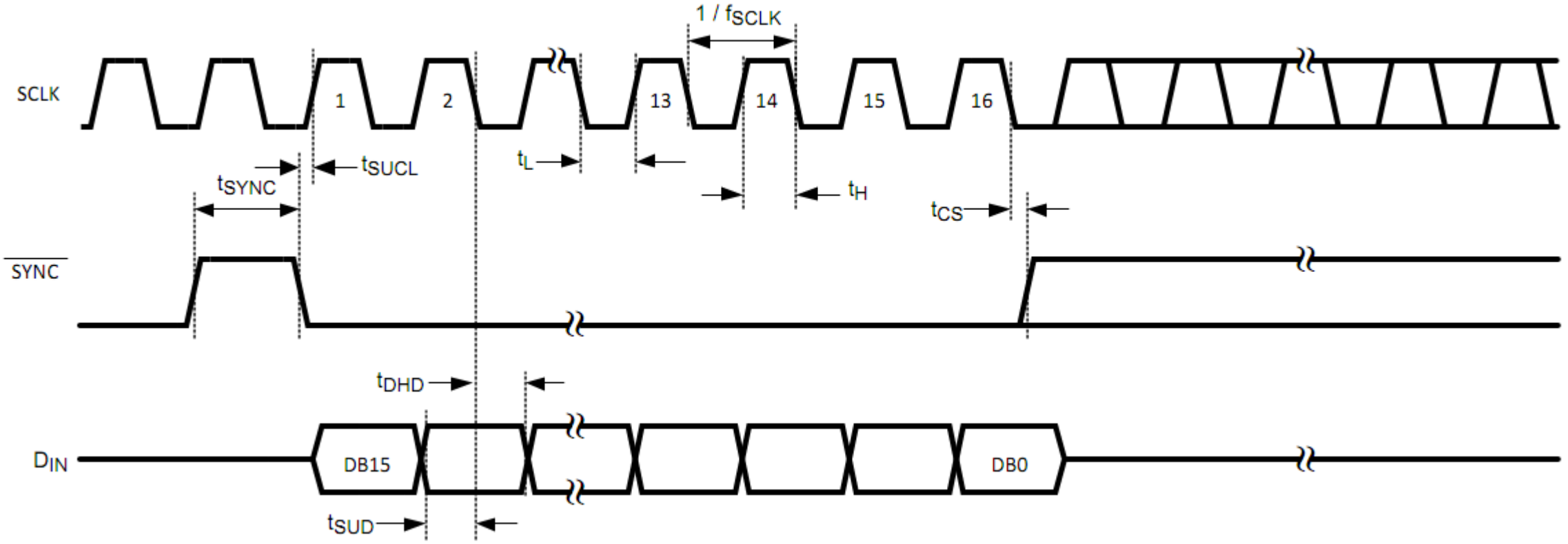






8.7 AXI SPI接口

▶ DAC121S101 DA转换接口时序



- 任何两次写操作之间必须使(SYNC#) 维持一段时间的高电平，以便启动下一次数据传输。该芯片支持SCLK的最高时钟频率为30MHz

▶ DAC121S101 DA转换接口时序

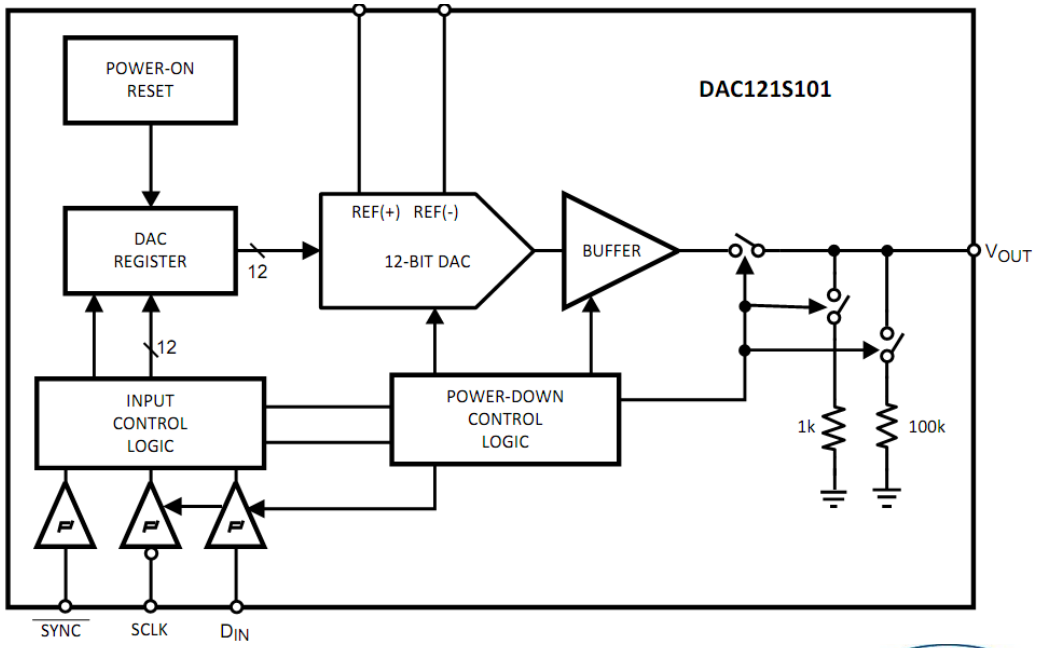
- 16位串行数据的含义

DB15 (MSB) DB0 (LSB)

X	X	PD1	PD0	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

- D0~D11为12位DA转换的数字量，
- PD0~PD1为电源下拉控制逻辑的输入，控制电源下拉模块的工作方式，改变输出Vout的输出连接方式

PD1	PD0	电源下拉模块工作方式
0	0	正常工作（不下拉），Vout正常输出
0	1	Vout通过1K电阻下拉
1	0	Vout通过100K电阻下拉
1	1	Vout为高阻状态



8.7 AXI SPI接口

▶ DAC121S101 SPI 时序控制要求

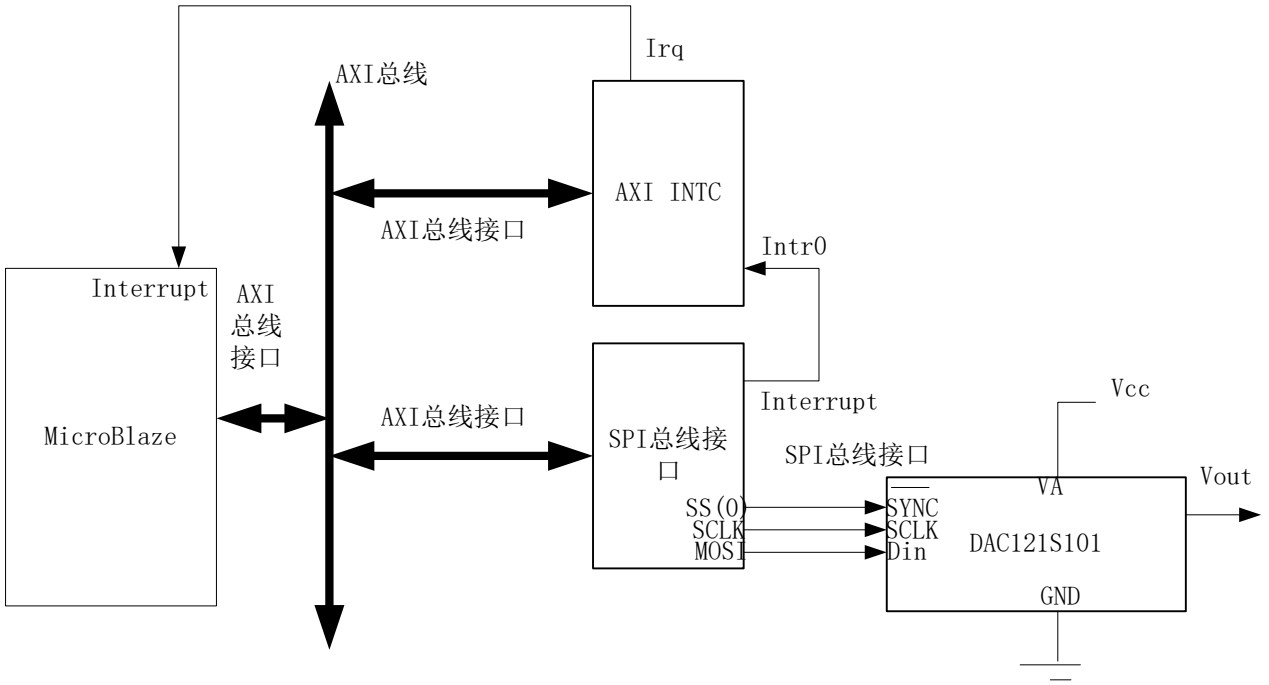
- DAC121S101要求SPI总线时序空闲时SCLK为低电平，并且在SCLK的下降沿采样数据，因此SPI总线接口控制寄存器SPICR的CPOL需设置为0，CPHA需设置为1。
- DAC121S101要求SPI总线高位优先传送，因此SPI总线接口控制寄存器的LSB优先需设置为0。 此SPI总线接口需设置为主设备，并且使能SPI接口。
- 由于DAC121S101要求每次传送16位数据，而且在两次数据的传送过程中必须使(SYNC)维持一定时间的高电平，因此可以配置SPI总线接口采用16位数据、自动控制SS（N）的数据传送方式，其中N=1。AXI SPI接口通过硬件配置为采用16位数据传送，并且不使用FIFO的模式。
- 由于DAC121S101支持的最高时钟频率为30MHz，如果AXI总线时钟为100MHz，那么可以将该频率4分频，得到SPI输出时钟频率为25MHz。

8.7 AXI SPI接口

▶ 基于SPI总线的DA转换器接口设计实例

- 利用DAC121S101DA转换芯片，基于SPI总线控制其Dout输出锯齿波。要求采用中断控制方式实现Microblaze微处理器与DAC121S101之间的通信。

  - DAC121S101要求SPI总线时序空闲时SCLK为低电平，并且在SCLK的下降沿采样数据，因此SPI总线接口控制寄存器SPICR的CPOL需设置为0，CPHA需设置为1。
  - DAC121S101要求SPI总线高位优先传送，因此SPI总线接口控制寄存器的LSB优先需设置为0。 此SPI总线接口需设置为主设备，并且使能SPI接口。
  - 由于DAC121S101要求每次传送16位数据，而且在两次数据的传送过程中必须使SYNC维持一定时间的高电平，因此可以配置SPI总线接口采用16位数据、自动控制SS（N）的数据传送方式，其中N=1。AXI SPI接口通过硬件配置为采用16位数据传送，并且不使用FIFO的模式。
  - 由于DAC121S101支持的最高时钟频率为30MHz，如果AXI总线时钟为100MHz，那么可以将该频率4分频，得到SPI输出时钟频率为25MHz。



▶ 基于SPI总线的DA转换器接口设计实例

- 主函数

```
volatile int  TransferInProgress;
int main(void)
{
    int Status;
    u32 Count;
    Status = SpiIntrExample(&IntcInstance, &SpiInstance, SPI_DEVICE_ID, SPI_IRPT_INTR);
    while(1)
    {
        WriteBuffer[0] = (u8)(Count);                    //SPI输出数据的低8位
        WriteBuffer[1] = (u8)(Count>>8)&0x0f;            // SPI输出数据的高8位，其中最高4位清0，使得Vout正常输出电压
        Count++;
        if (Count==4096)  Count=0;                       //12位DAC转换数据到达最大值时，恢复到0
        TransferInProgress = TRUE;                       //设置传输状态标志为1
        XSpi_Transfer(SpiInstancePtr, WriteBuffer, (void*)0, 2);          //一次传输2个字节
        while (TransferInProgress);                                       //等待传输结束
    }
    return XST_SUCCESS;
}
```

▶ 基于SPI总线的DA转换器接口设计实例

- SPI初始化函数

```
int SpiIntrExample(XIntc *IntcInstancePtr, XSpi *SpiInstancePtr, u16 SpiDeviceId, u16 SpiIntrId)
{   int Status;
    XSpi_Config *ConfigPtr;
    ConfigPtr = XSpi_LookupConfig(SpiDeviceId);                  // 查找SPI接口配置项
    Status = XSpi_CfgInitialize(SpiInstancePtr, ConfigPtr, ConfigPtr->BaseAddress);    //根据配置项参数，初始化SPI参数

    //配置中断控制器，以及针对中断控制器Intr引脚的SPI设备中断处理函数
    Status = SpiSetupIntrSystem(IntcInstancePtr,     SpiInstancePtr, SpiIntrId);

    //设置SPI接口用户中断服务函数
    XSpi_SetStatusHandler(SpiInstancePtr, SpiInstancePtr, (XSpi_StatusHandler) SpiIntrHandler);

    //配置SPI接口工作模式
    Status = XSpi_SetOptions(SpiInstancePtr, XSP_MASTER_OPTION |XSP_CLK_PHASE_1_OPTION);

    Status = XSpi_SetSlaveSelect(SpiInstancePtr, 1);             //设置从设备选择信号
    XSpi_Start(SpiInstancePtr);                                  //使能SPI接口
    return XST_SUCCESS;
}
```

▶ 基于SPI总线的DA转换器接口设计实例

- 中断系统初始化函数

```
static int SpiSetupIntrSystem(XIntc *IntcInstancePtr, XSpi *SpiInstancePtr, u16 SpiIntrId)    //配置中断系统
{
    int Status;
    Status = XIntc_Initialize(IntcInstancePtr, INTC_DEVICE_ID);          //初始化中断控制器

    //配置相应中断输入引脚的中断处理函数
    Status = XIntc_Connect(IntcInstancePtr, SpiIntrId, (XInterruptHandler) XSpi_InterruptHandler, (void *)SpiInstancePtr);

    Status = XIntc_Start(IntcInstancePtr, XIN_REAL_MODE);                //使能中断中断请求输出端，硬件中断
    XIntc_Enable(IntcInstancePtr, SpiIntrId);                            //使能SPI接口对应的中断输入端

    //注册中断控制器的总中断处理函数
    microblaze_register_handler((XInterruptHandler)XIntc_InterruptHandler, IntcInstancePtr);

    microblaze_enable_interrupts();                                      //开放微处理器中断
    return XST_SUCCESS;
}
```

▶ 基于SPI总线的DA转换器接口设计实例

- SPI中断服务函数

```
void SpiIntrHandler(void *CallBackRef, u32 StatusEvent, u32 ByteCount)
{
    TransferInProgress = FALSE;                    // 进入中断表示传输结束，修改传输状态标志为0
    if (StatusEvent != XST_SPI_TRANSFER_DONE)
    {
        Error++;
    }
}
```

- 基于SPI总线的DA转换器接口设计实例
  - 扩展
    - 锯齿波的周期
    - 锯齿波的电压范围
    - 输出波形

第8章 作业（一）

- 作业题（P368）
  - 19
  - 20
  - 21

- 要求
  - 不用抄题目
  - 白纸作业，写好班级、名字
  - 严格要求自己
  - 实验课验证

Thanks