第7章 中断技术

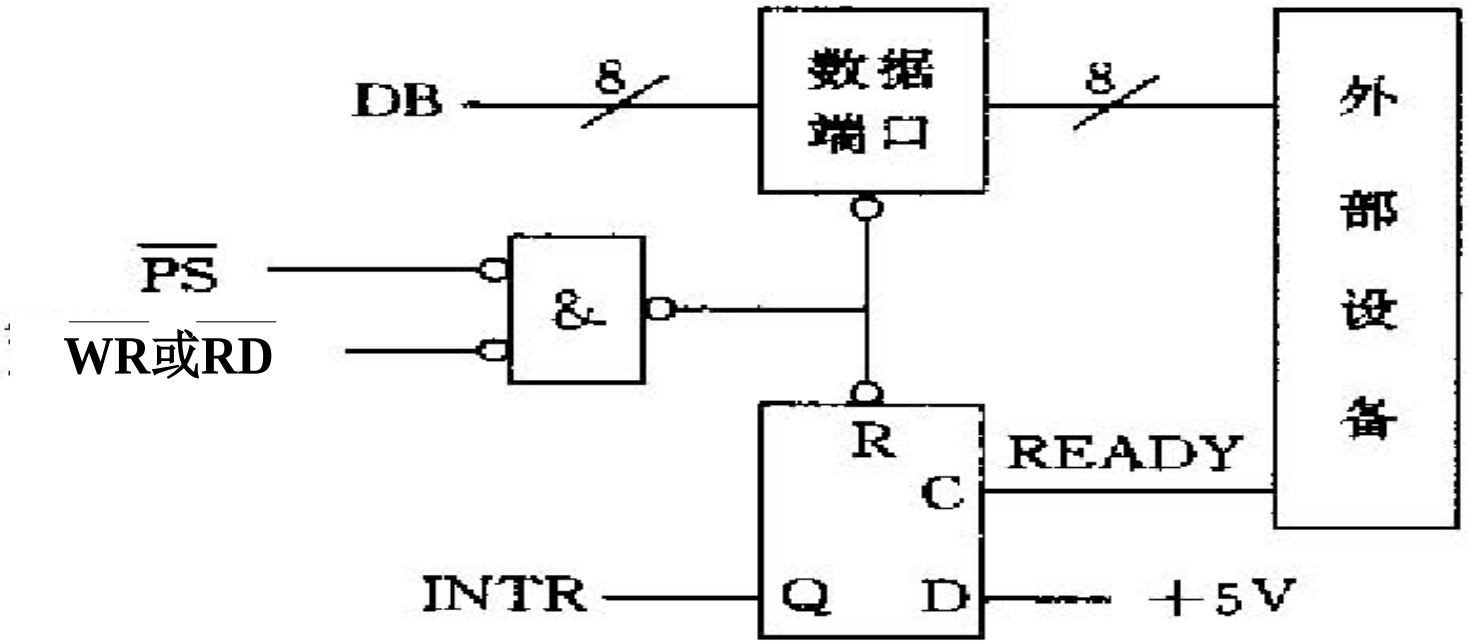
学习目标

- 了解中断控制器的基本构成
- 熟悉AXI INTC的具体应用
- 理解80x86微处理器实模式下中断处理过程
- 理解MicroBlaze微处理器的中断处理过程
- 熟悉基于MicroBlaze微处理器AXI总线的中断方式接口设计
- 熟悉MicroBlaze中断处理程序设计
- 掌握SPI串行总线接口技术
- 理解GPIO接口中断原理
- 掌握硬件时钟中断原理和应用

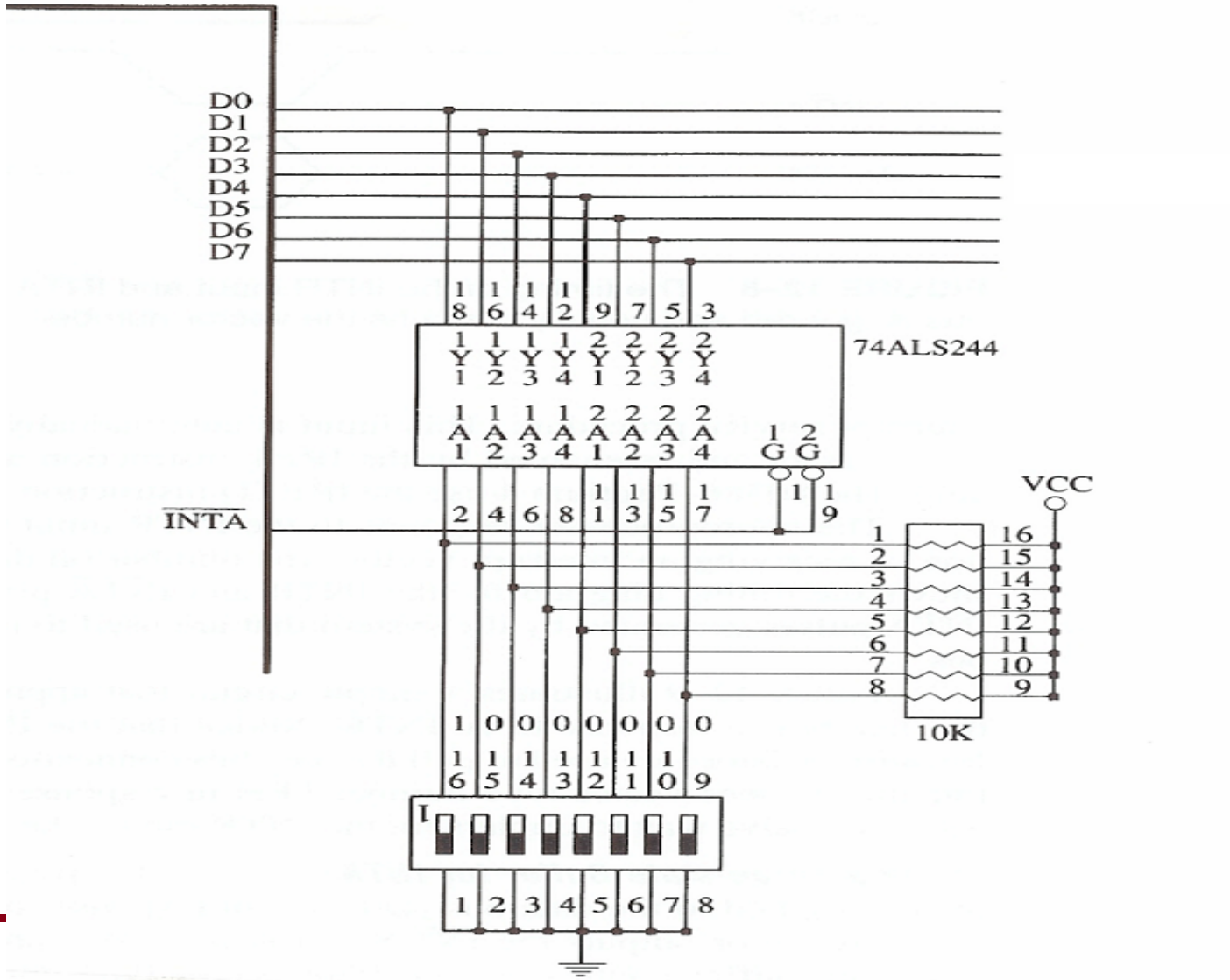
中断控制器构成

- 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。

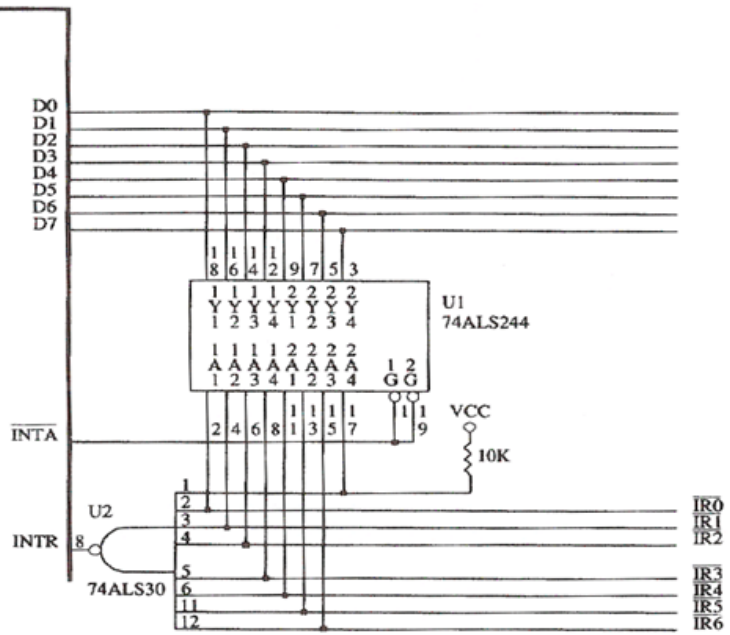
中断请求信号保持与清除



中断源识别——读取中断类型码



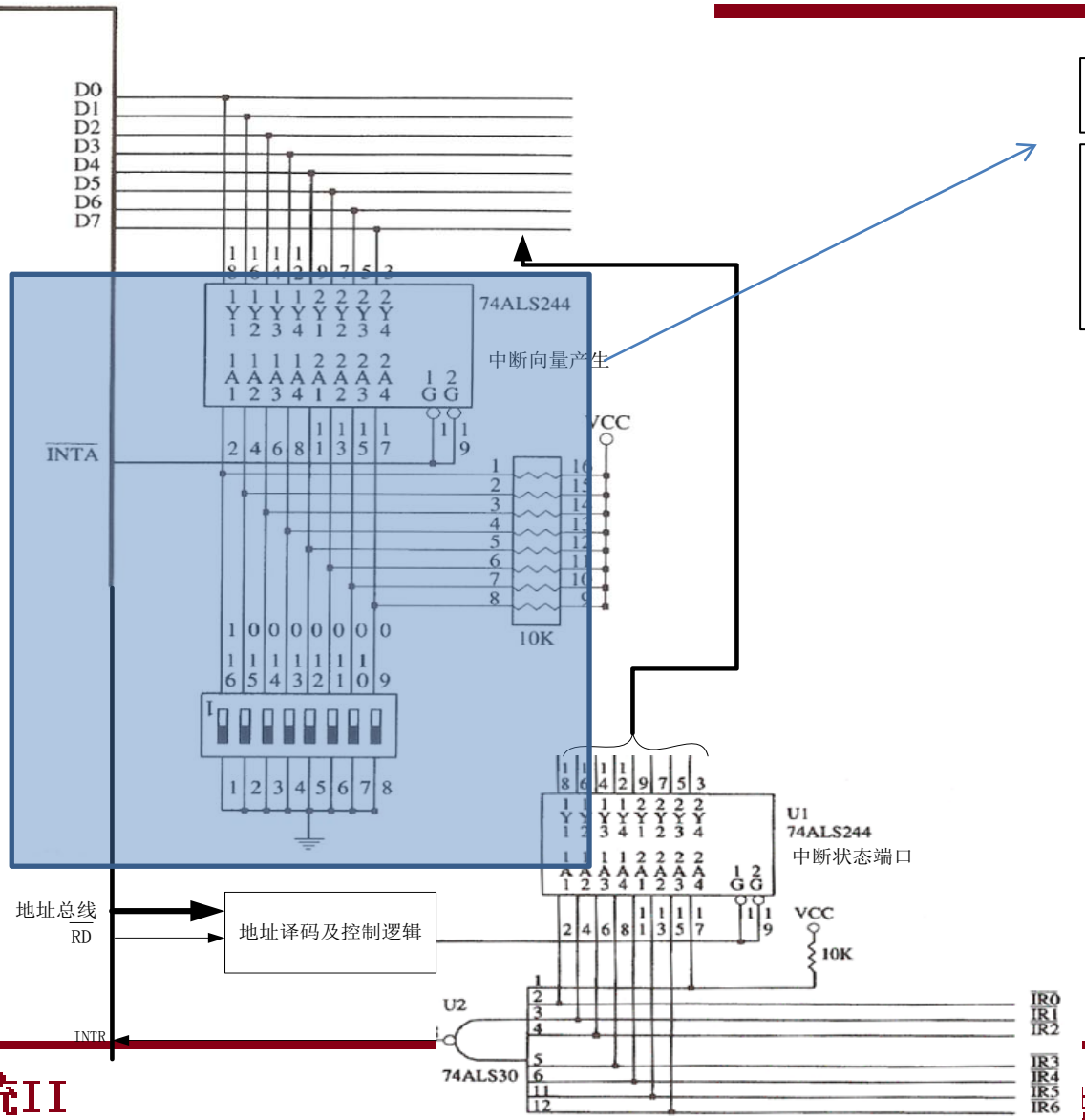
多中断源



同时产生多个中断，中断类型码种类太多（256）

	中断类型码							
中断源	D7	D6	D5	D4	D3	D2	D1	D0
IR0	1	1	1	1	1	1	1	0
IR1	1	1	1	1	1	1	0	1
IR2	1	1	1	1	1	0	1	1
IR3	1	1	1	1	0	1	1	1
IR4	1	1	1	0	1	1	1	1
IR5	1	1	0	1	1	1	1	1
IR6	1	0	1	1	1	1	1	1

软件查询中断源-单一向量

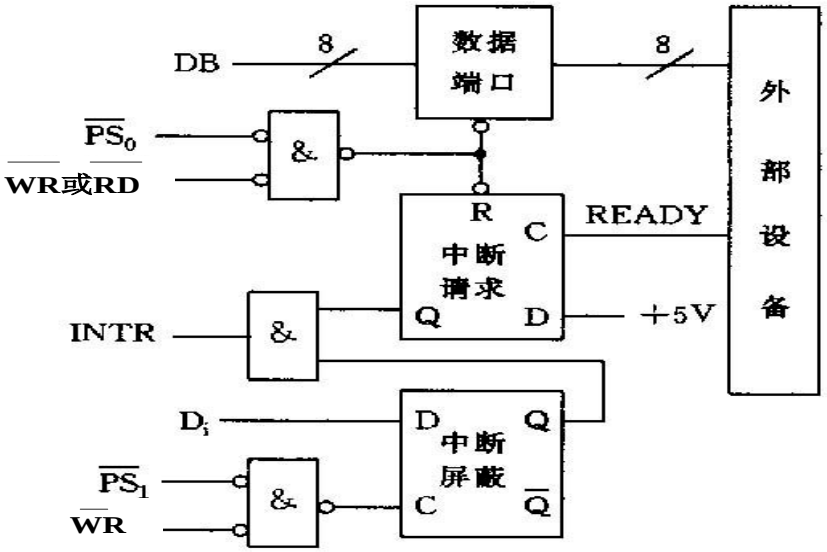


嵌入式处理器

固化，所有外设采用一个固定的中断类型码

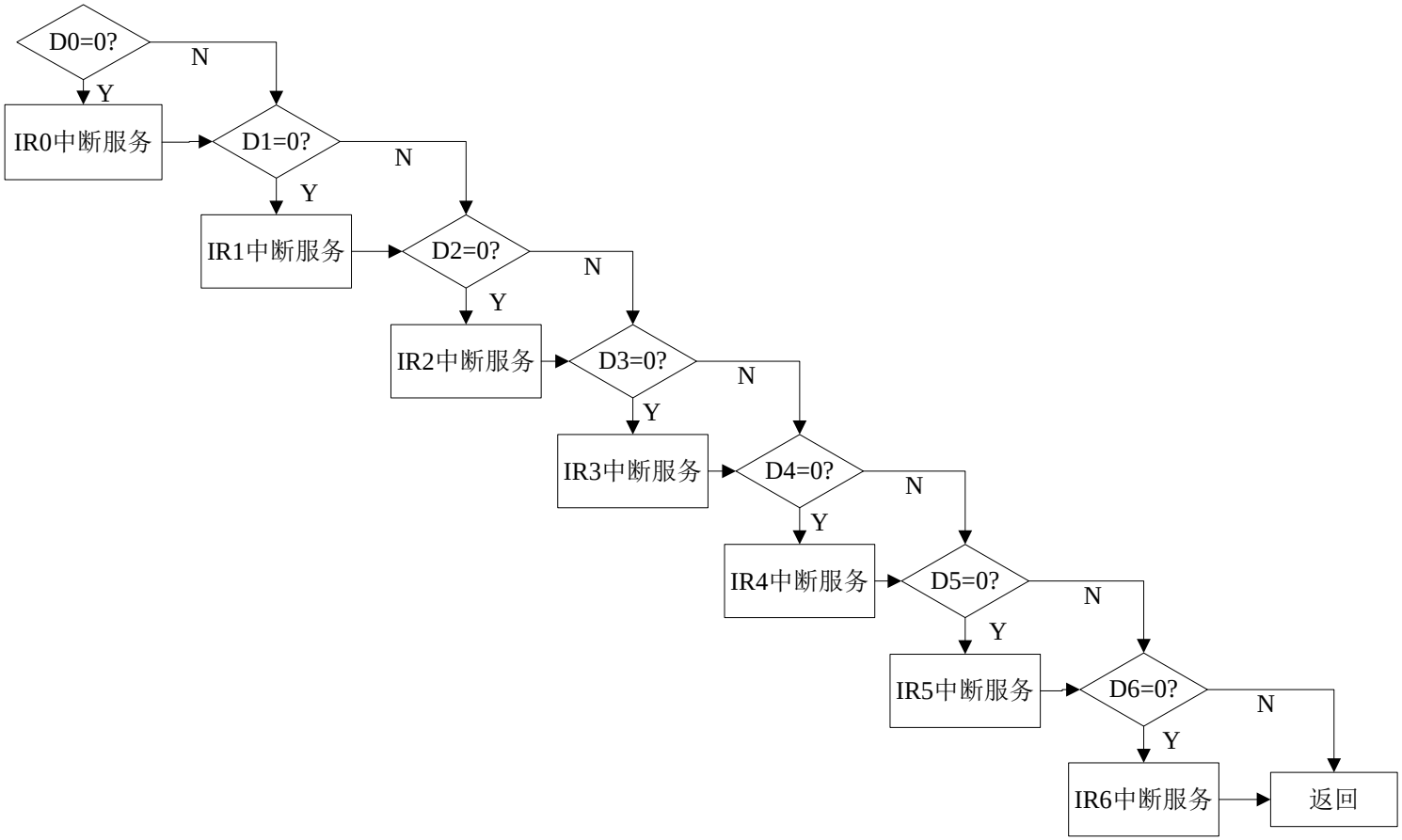
中断控制

- 微处理器用输出指令来控制中断屏蔽触发器的状态，从而控制是否接受某个特殊外设的中断请求
- 微处理器内部也有一个中断允许触发器，只有当其为“1”（即开中断），CPU才能响应外部中断

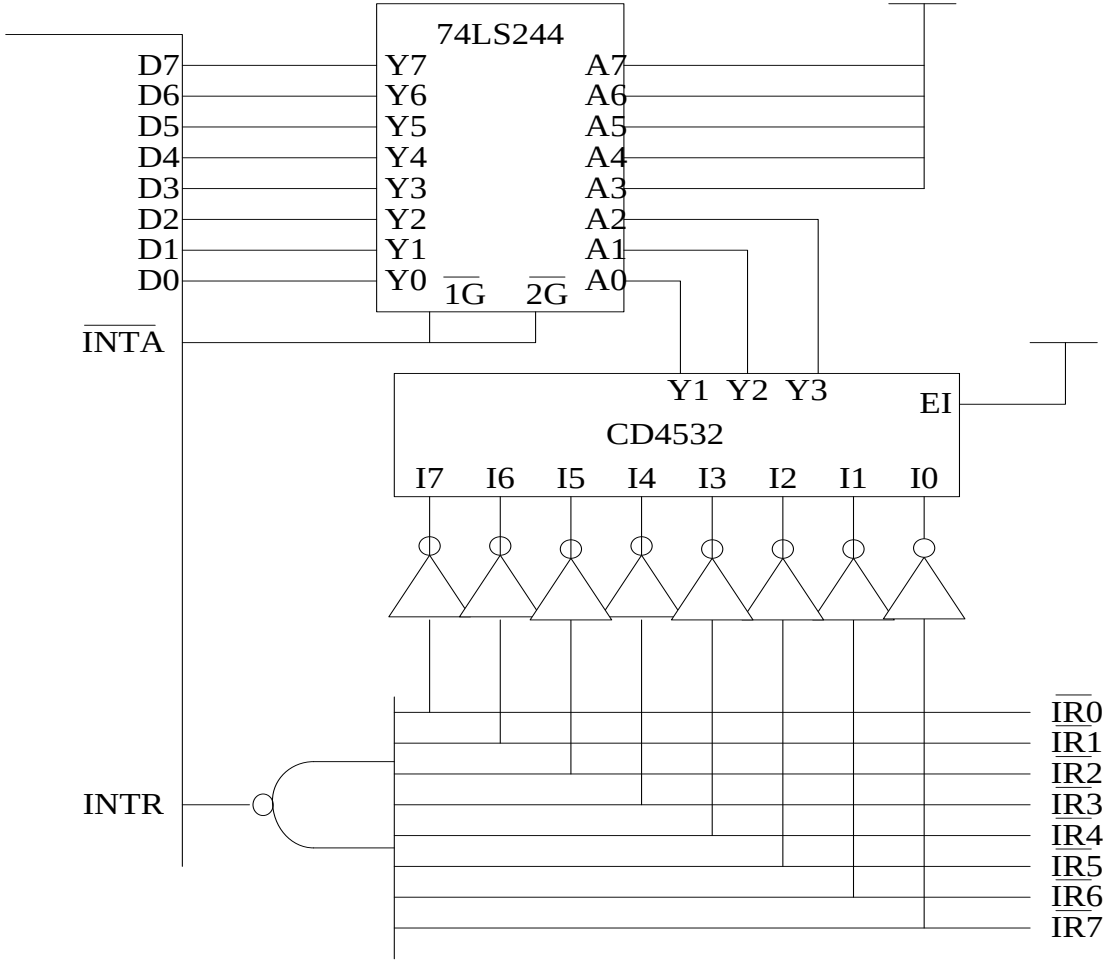


中断优先级

- 软件查询顺序决定

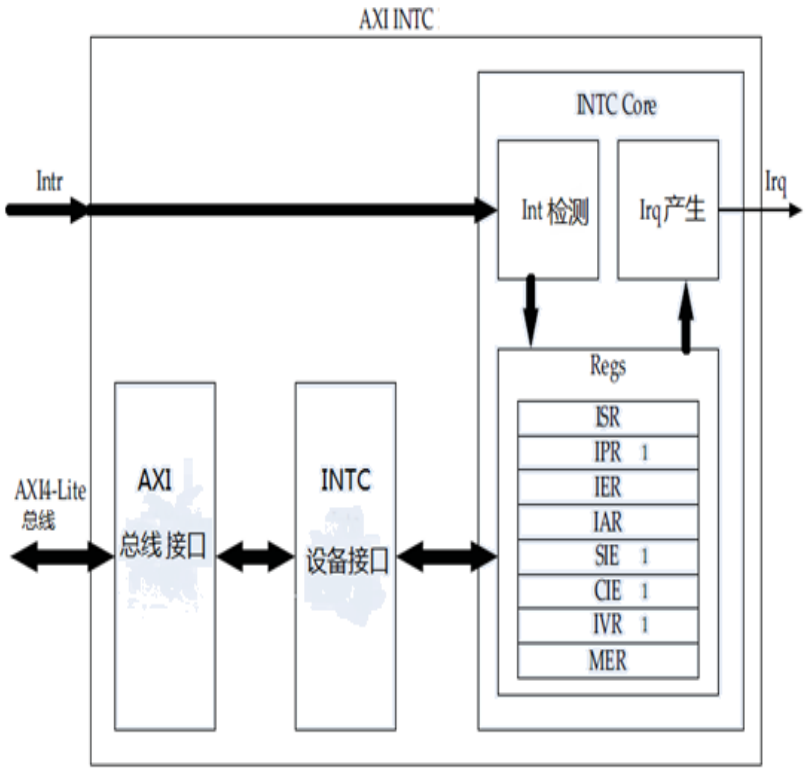


优先编码器


74LS244
D7
D6
D5
D4
D3
D2
D1
D0
Y7
Y6
Y5
Y4
Y3
Y2
Y1
Y0
1G
2G
A7
A6
A5
A4
A3
A2
A1
A0
INTA
Y1 Y2 Y3
EI
CD4532
I7 I6 I5 I4 I3 I2 I1 I0
IR0
IR1
IR2
IR3
IR4
IR5
IR6
IR7
INTR

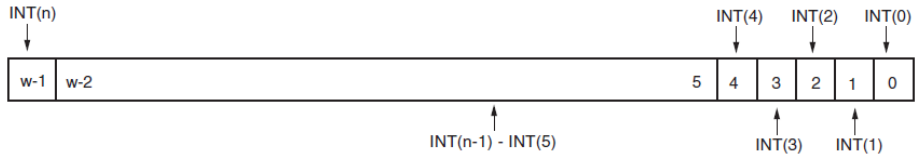
中断控制器举例-AXI INTC

- 支持32个中断源输入，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，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



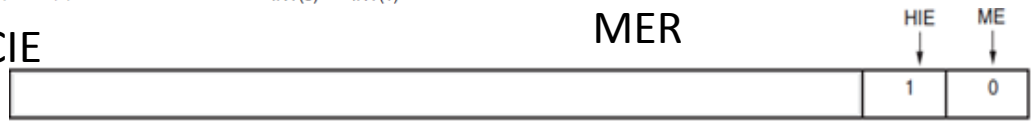
寄存器偏移地址

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,IPR,IER,IAR,SIE,CIE

MER



中断处理过程

(1) 在中断请求输入端Intr上接受中断请求。

(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。

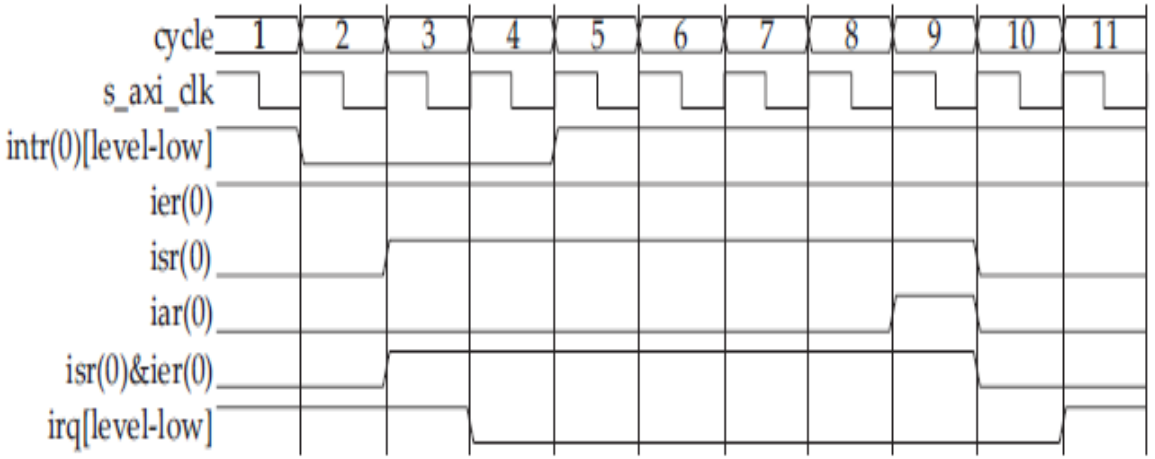
(4) 若使用优先级判断电路， 优先级判定电路检出优先级最高的中断请求位，并将IVR设置为相应的值。

(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。

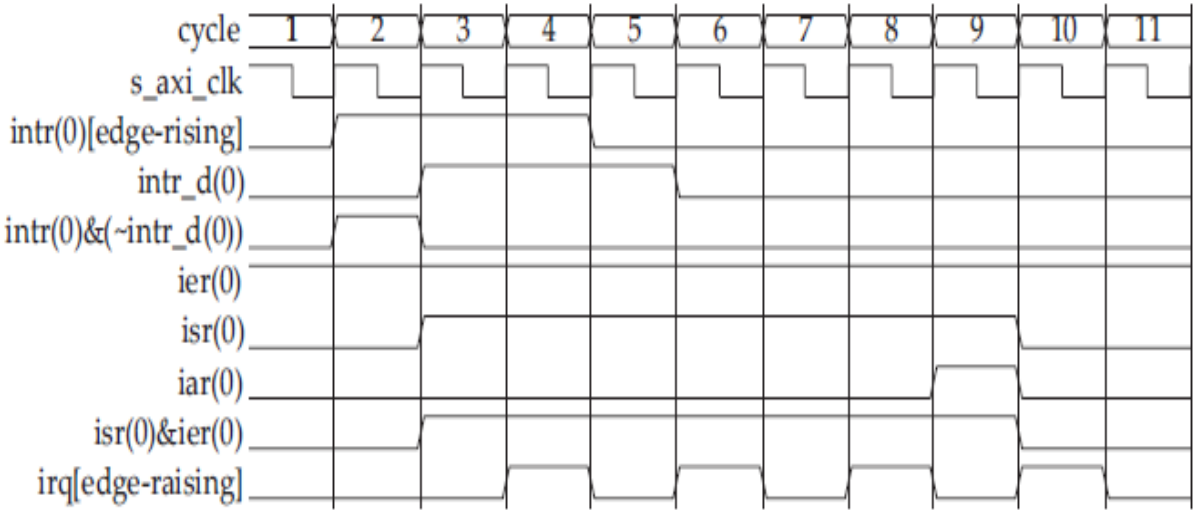
(6)微处理器向中断响应寄存器（IAR）对应的位写入1，使ISR相应位复位从而结束中断。

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



编程控制

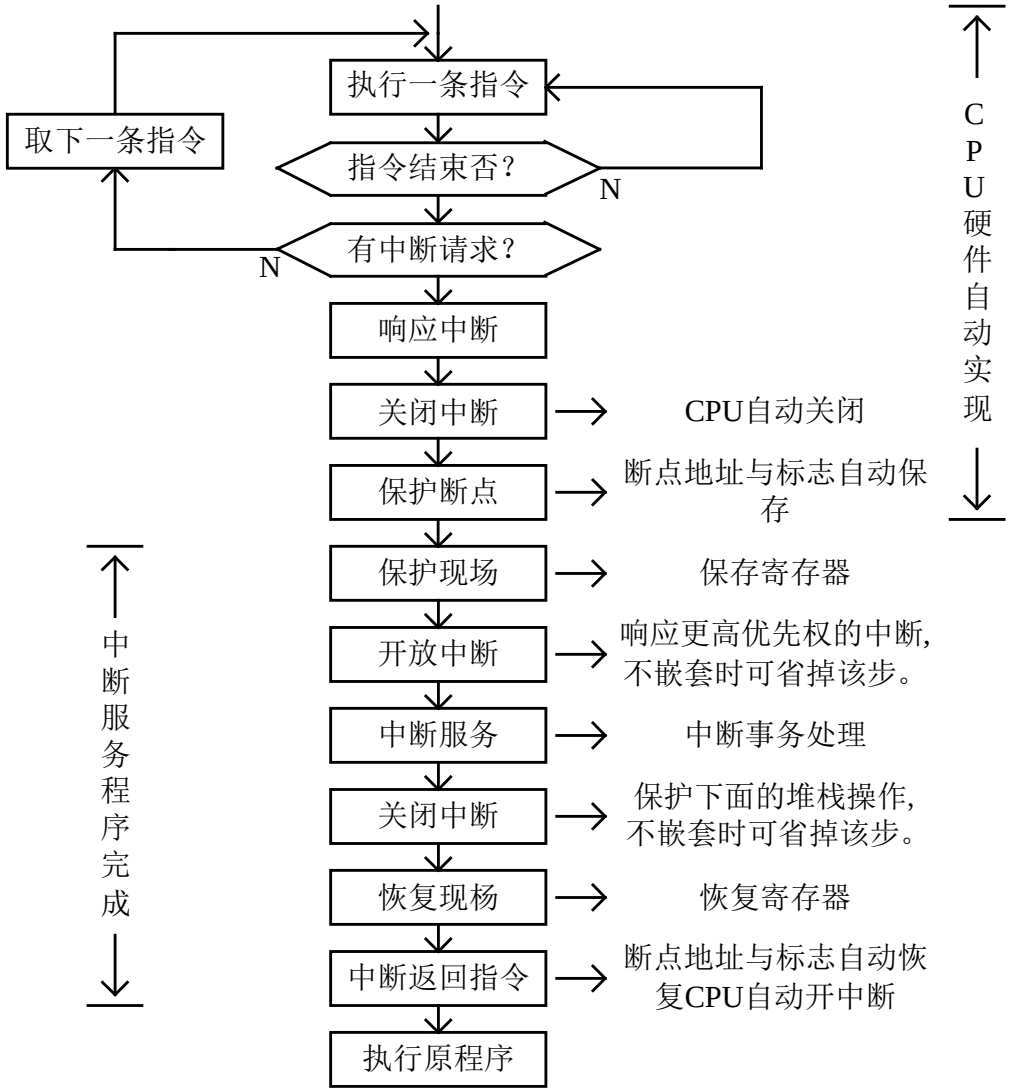
- 初始化，通常包括两个方面：
  - 修改MER，使得HIE和ME都为1；
  - 修改IER，使得连接了中断请求源的相应位为1，允许中断请求
- 中断结束
  - 写IAR,使服务了的中断请求复位

- 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Intr0~4，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Intr0~4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);
```

微处理器响应中断的一般过程



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

intel 80X86中断系统介绍

- 80x86的中断类型码及中断种类

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

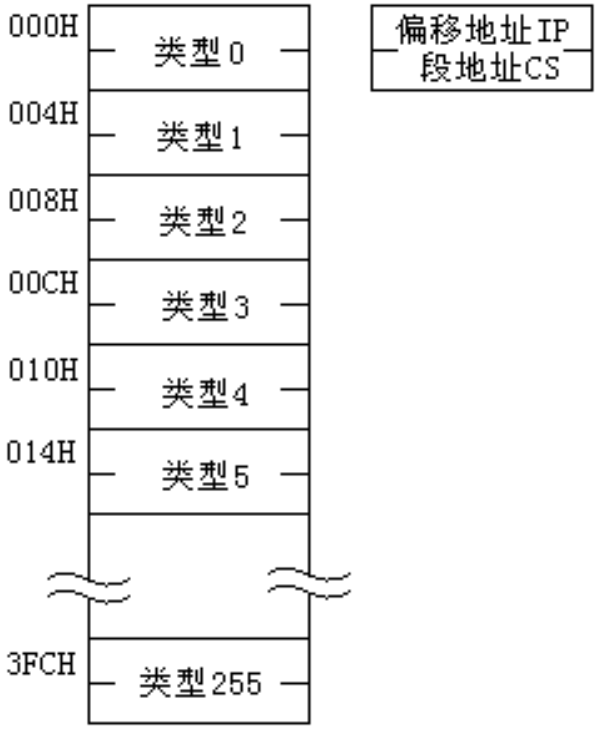
中断优先权次序为：内部中断(单步中断除外)优先权最高、其次是NMI、再次是INTR，优先权最低的是内部中断中的单步中断

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。

中断向量表

一个中断类型码n占有4n、4n+1和4n+2、4n+3四个字节单元或4n和4n+2两个字单元。在这4个字节中，存放着中断向量对应的中断源服务程序入口地址



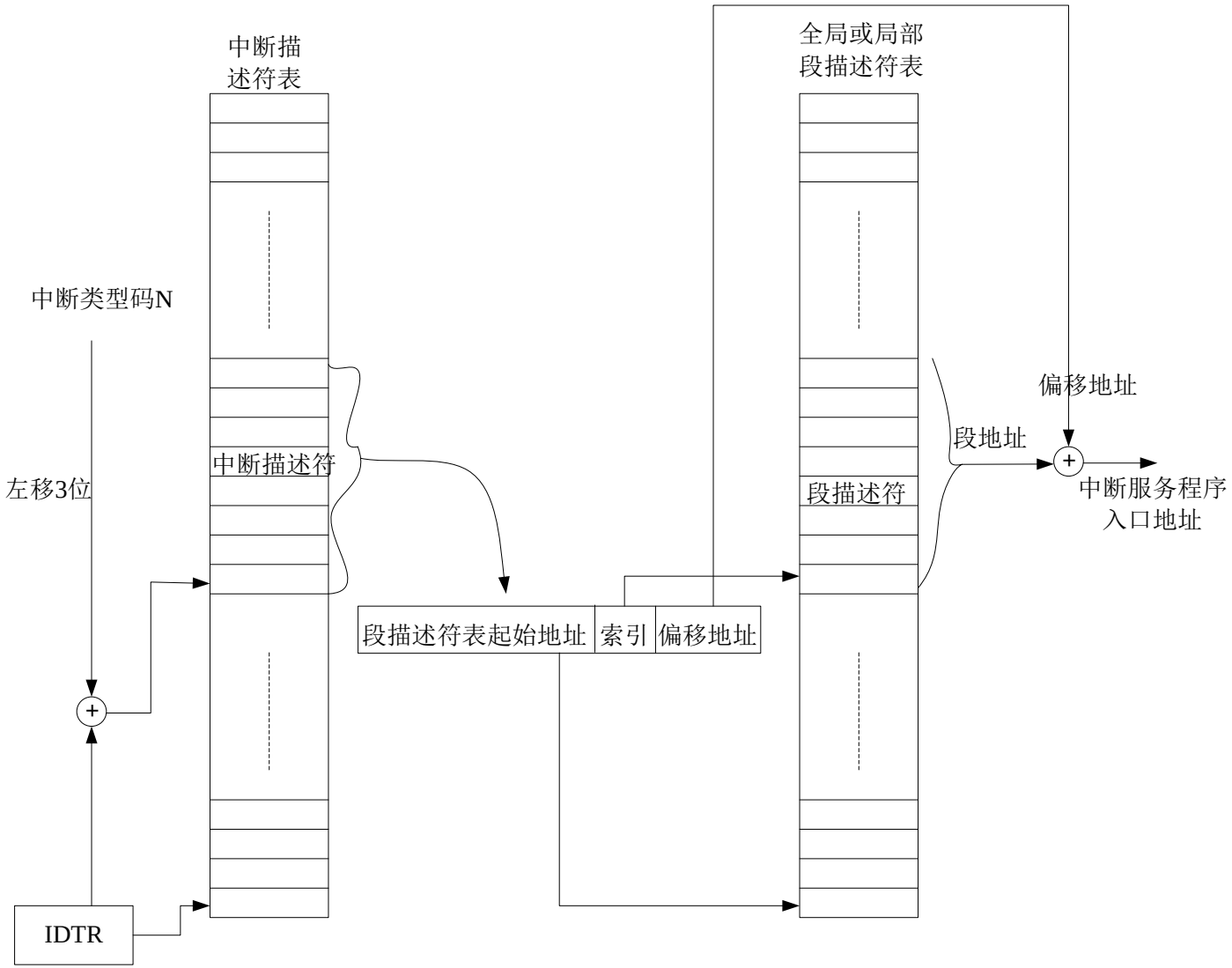
- 若80x86系统采用的8259A的中断类型码为88H，试问中断服务程序的地址填入哪个字单元？

中断服务程序偏移地址IP填入4n字单元，而4×88H=220H，故填入00220H字单元。
中断服务程序段地址CS填入4n+2字单元，而4×88H+2=222H，故填入00222H字单元。

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



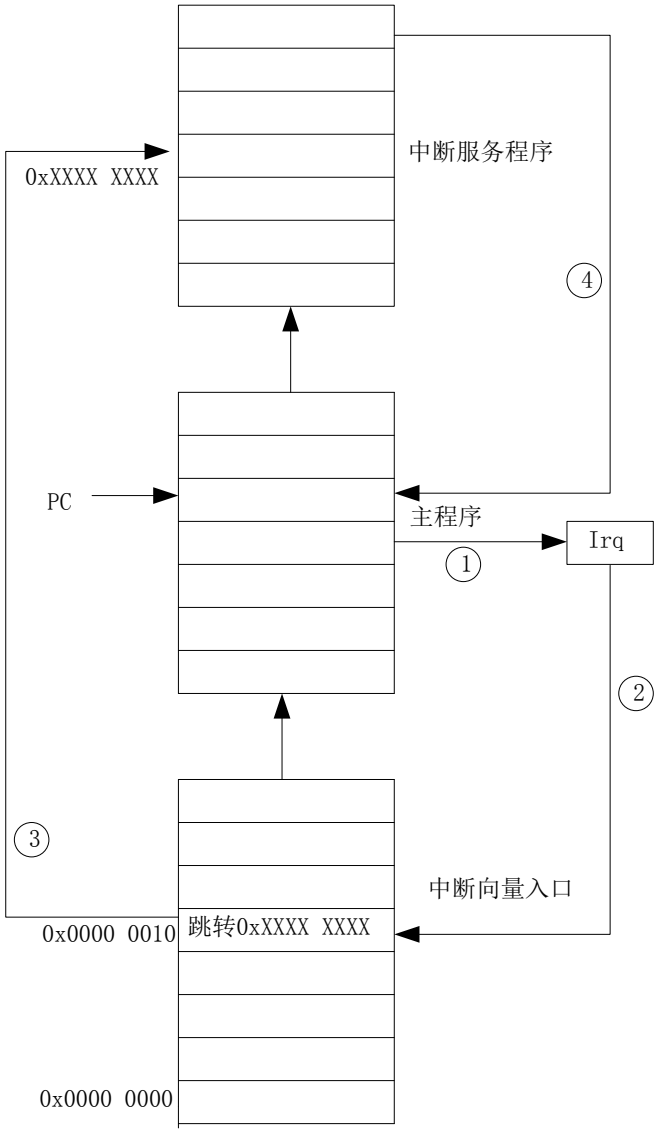
MicroBlaze中断系统介绍

中断类型	中断向量地址	保存断点的寄存器
复位	0x00000000-0x00000004	-
用户异常	0x00000008-0x0000000C	-
中断	0x00000010-0x00000014	R14
不可屏蔽硬件中断	0x00000018-0x0000001C	R16
硬件打断（break）		
软件打断（break）		
硬件异常	0x00000020-0x00000024	R17

MicroBlaze中断处理过程

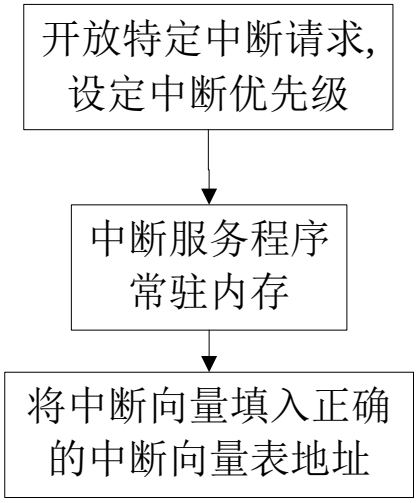
- 微处理器对所有外设仅提供一个中断服务程序跳转地址，因此该中断服务程序也叫做主中断服务程序。如果系统中存在不同种类中断源，该中断服务程序应该读取中断控制器的中断请求寄存器，以确定是哪个中断源发生了中断请求，并且需要进一步调用针对该中断源的中断服务程序。因此主中断服务程序需要维护一个中断向量表（中断向量数组），并根据中断源查找相应中断服务程序

软件识别不同外部中断源，并实现程序控制

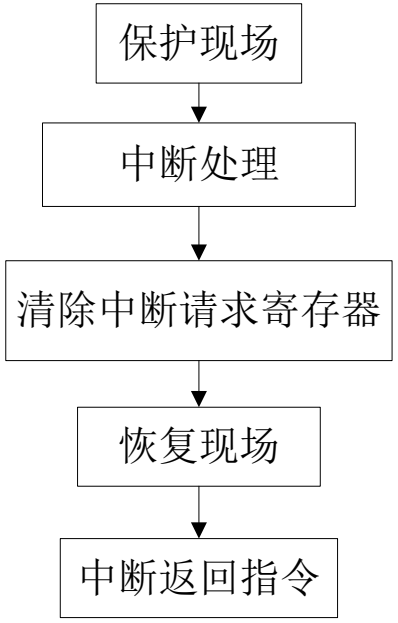


中断方式接口设计

- 中断程序构成



(A)启用中断传送



(B)中断服务程序

(C) 停止中断传送

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0x0000 0010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。

- AXI INTC中断控制器驱动API

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_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

AXI INTC中断控制器驱动相关数据结构

```
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;
```

```
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;
```

```
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;
```

AXI GPIO并行接口控制器中断原理简介

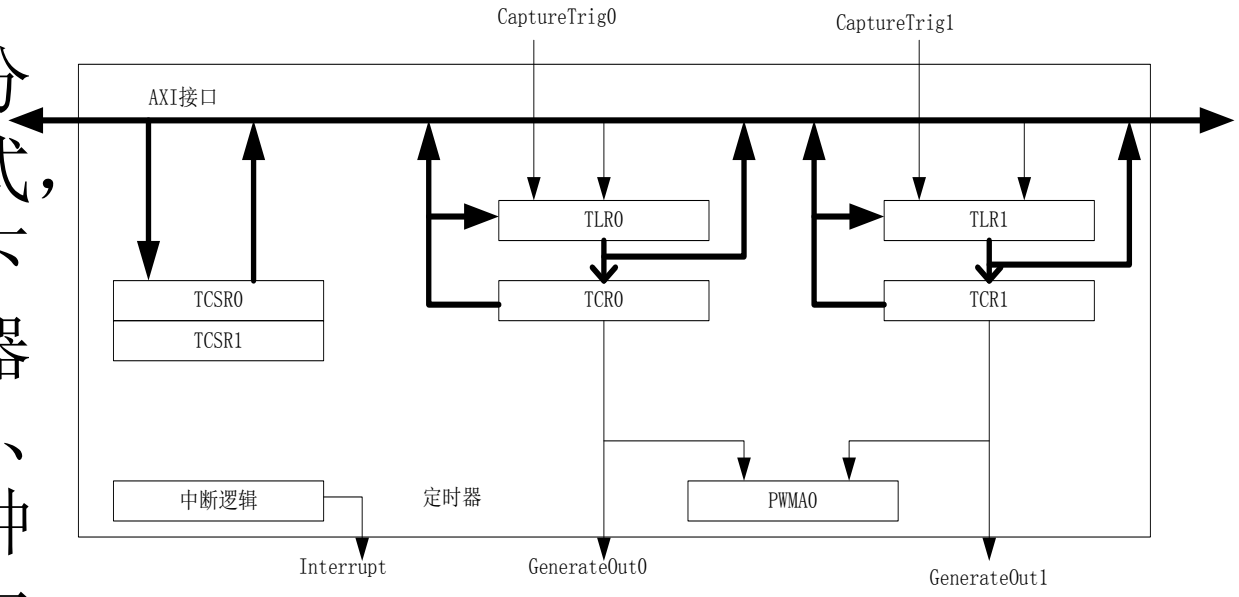
- GPIO内部中断相关寄存器

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。

AXI Timer定时器简介

- 定时模式可以分为两种计数方式，向上计数和向下计数。该定时器可以工作在8位、16位、32位三种模式，采用小字节序



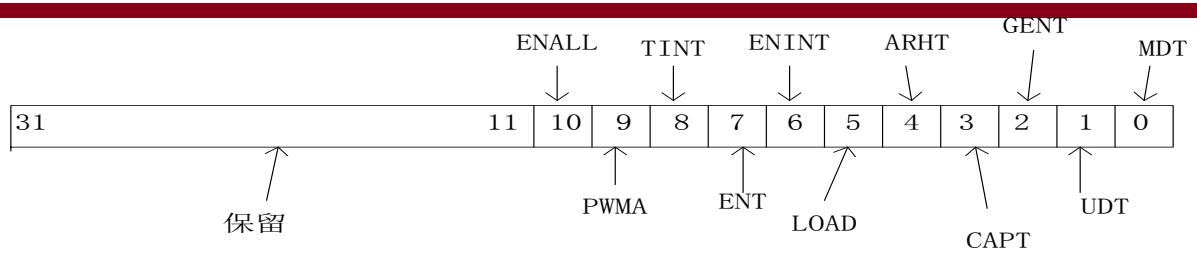
两个定时器中的任意一个计数结束都会产生中断，如果允许输出中断信号，则使Interrupt输出高电平，从而产生中断请求。

产生中断信号的时间间隔

- 加计数：T=(TCRmax-TLR)*AXI_CLK_PERIOD
- 减计数：T=TLR* AXI_CLK_PERIOD

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

TCSR寄存器各位的定义



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		

控制定时器定时结束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

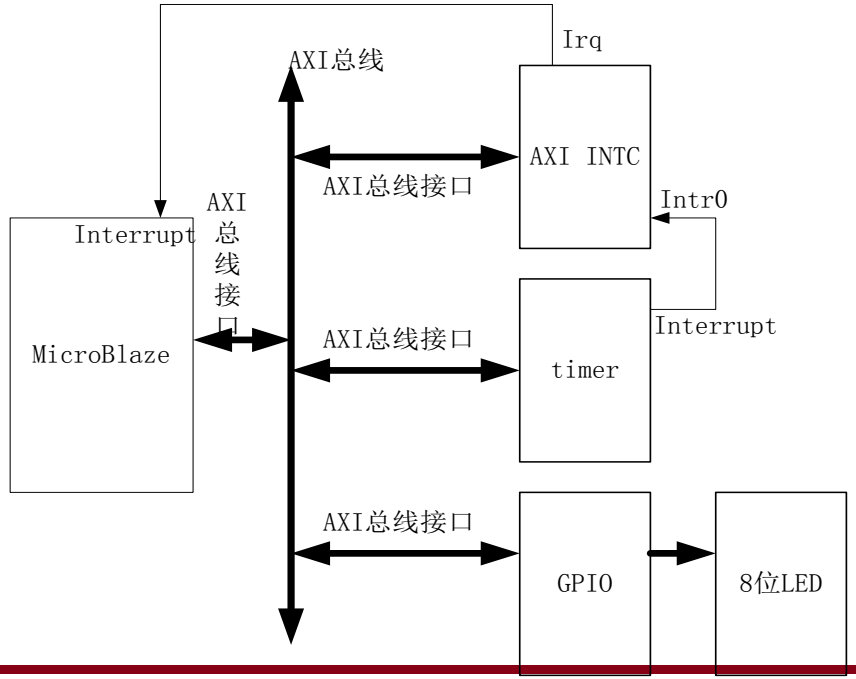
- XTmrCtr_Initialize(XTmrCtr*, u16) : int
- XTmrCtr_Start(XTmrCtr*, u8) : void
- XTmrCtr_Stop(XTmrCtr*, u8) : void
- XTmrCtr_GetValue(XTmrCtr*, u8) : u32
- XTmrCtr_SetResetValue(XTmrCtr*, u8, u32) : void
- XTmrCtr_GetCaptureValue(XTmrCtr*, u8) : u32
- XTmrCtr_IsExpired(XTmrCtr*, u8) : int
- XTmrCtr_Reset(XTmrCtr*, 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

GPIO API

- 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

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1秒钟更换一个。并采用硬件定时器中断方式实现延时控制。



- 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$2^{32}-1-100M$。

定时器中断服务程序

```
void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber)
{
        Xil_Out32(XPAR_LEDS_8BITS_BASEADDR,1<<LedBits);
//产生中断时，输出LED显示值
        LedBits++;//修改显示位置指向下一位
        if(LedBits==8)
//由于只有8位LED灯，因此位置不能大于等于8，继续从bit0开始循环
                LedBits=0;
}
```

建立定时器中断系统函数

```
static int TmrCtrSetupIntrSystem(XIntc* IntcInstancePtr,XTmrCtr* TmrCtrInstancePtr,
u16 DeviceId, u16 IntrId, u8 TmrCtrNumber)
{                int Status;
Status = XIntc_Initialize(IntcInstancePtr, INTC_DEVICE_ID);//初始化中断控制器数据结构
                if (Status != XST_SUCCESS) {
                                return XST_FAILURE;
                }
Status = XIntc_Connect(IntcInstancePtr, IntrId,
                (XInterruptHandler)XTmrCtr_InterruptHandler,(void *)TmrCtrInstancePtr);
//将定时器主中断服务程序注册到相应Intr引脚对应的中断向量
                if (Status != XST_SUCCESS) {
                                return XST_FAILURE;       }
                Status = XIntc_Start(IntcInstancePtr, XIN_REAL_MODE);
//设置中断控制器接受硬件中断，并发出中断请求
                if (Status != XST_SUCCESS) {
                                return XST_FAILURE;       }
                XIntc_Enable(IntcInstancePtr, IntrId);//使能定时器对应Intr的中断请求
microblaze_register_handler((XInterruptHandler)XIntc_InterruptHandler, IntcInstancePtr);
//将中断控制器的总中断服务程序注册到0X0000 0010地址处
                microblaze_enable_interrupts();//使能微处理器的中断控制位
                return XST_SUCCESS;
}
```

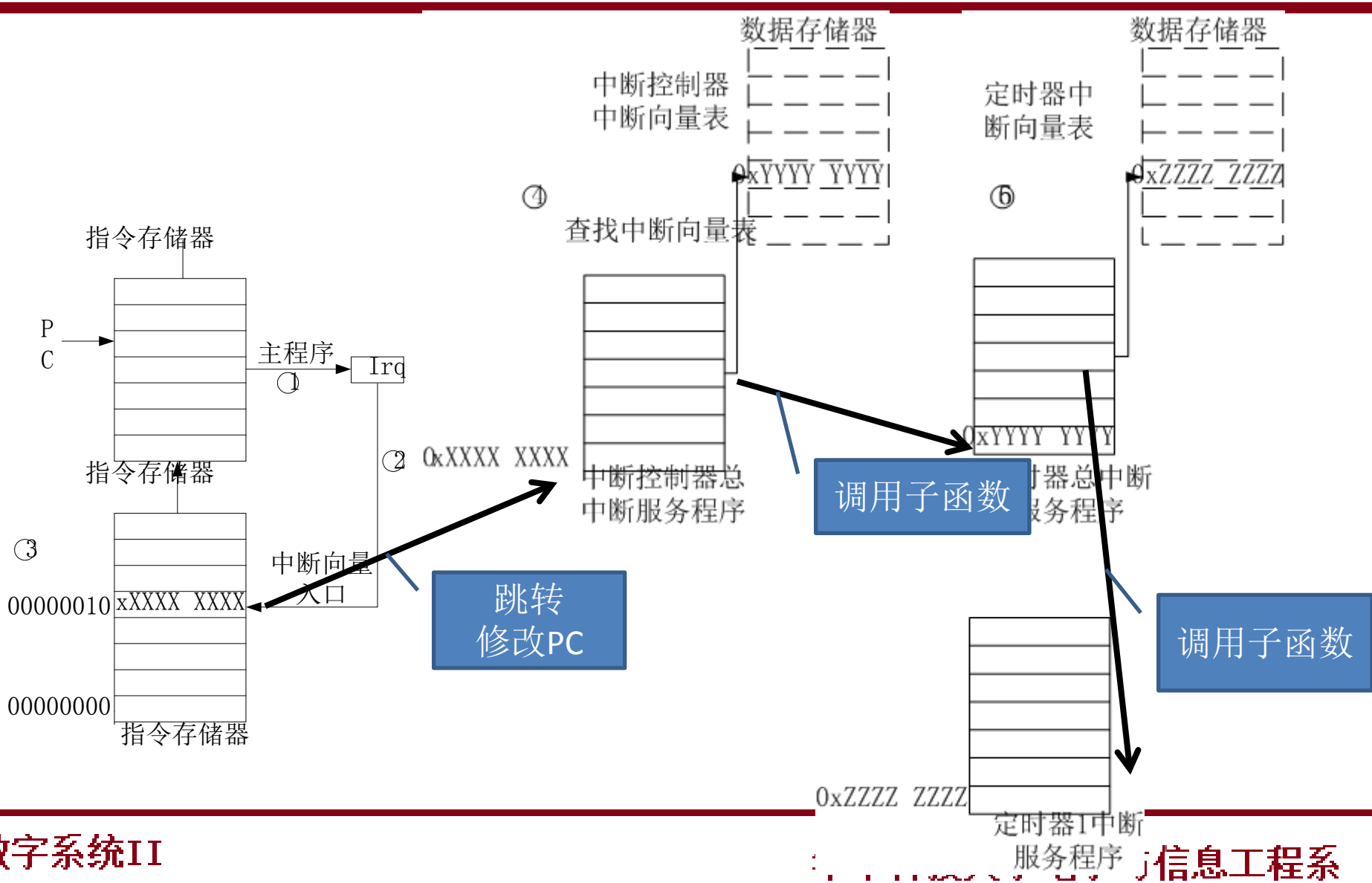
定时器初始化函数

```
int TmrCtrIntrExample(XIntc* IntcInstancePtr,XTmrCtr* TmrCtrInstancePtr,u16 DeviceId,u16 IntrId,u8 TmrCtrNumber)
{
        int Status;
        Status = XTmrCtr_Initialize(TmrCtrInstancePtr, DeviceId);
//初始化定时器数据结构，并清除定时器中断产生标志以及装载标志
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }
        Status = TmrCtrSetupIntrSystem(IntcInstancePtr,TmrCtrInstancePtr,DeviceId,IntrId,TmrCtrNumber);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }
        XTmrCtr_SetHandler(TmrCtrInstancePtr, TimerCounterHandler,  TmrCtrInstancePtr);//注册定时器中断服务程序
        XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber,
        XTC_INT_MODE_OPTION|XTC_AUTO_RELOAD_OPTION| XTC_DOWN_COUNT_OPTION);
//设置定时器0：允许中断、自动装载、减计数
        XTmrCtr_SetResetValue(TmrCtrInstancePtr, TmrCtrNumber, RESET_VALUE);
//设置定时器初始值到TLR
        XTmrCtr_Start(TmrCtrInstancePtr, TmrCtrNumber);
//首先设置LOAD标志为1，然后在清除load标志的同时，设置定时器使能标志ENT=1
        return XST_SUCCESS;
}
```

主函数

```
int main(void)
{
        int Status;
        LedBits=0;
        Xil_Out32(XPAR_LEDS_8BITS_BASEADDR+0x4,0x0);
//控制通道1 LED GPIO为输出
        Status = TmrCtrIntrExample(&InterruptController,
                                   &TimerCounterInst,
                                   TMRCTR_DEVICE_ID,
                                   TMRCTR_INTERRUPT_ID,
                                   TIMER_CNTR_0);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }
        While(1);//死循环
        return XST_SUCCESS;
}
```

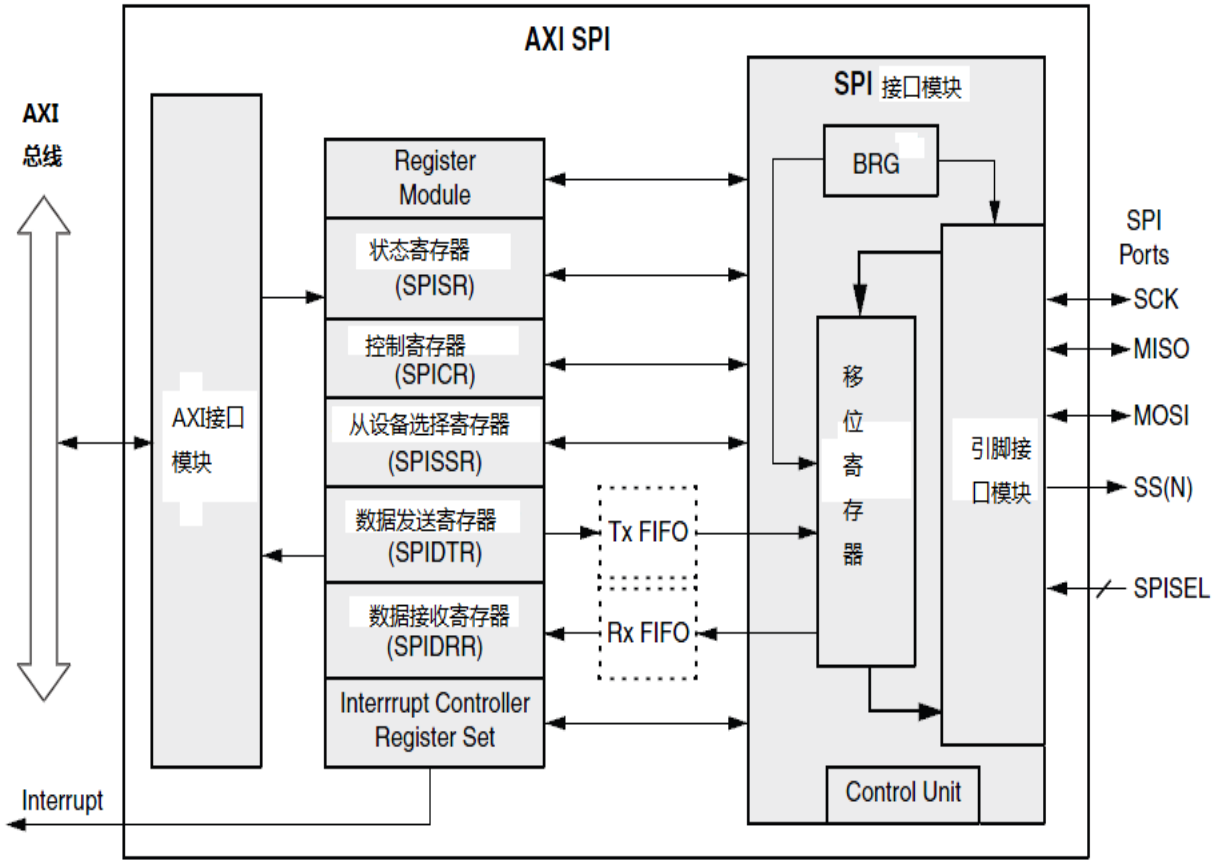
定时器中断程序执行过程



SPI DAC

DAC121S101

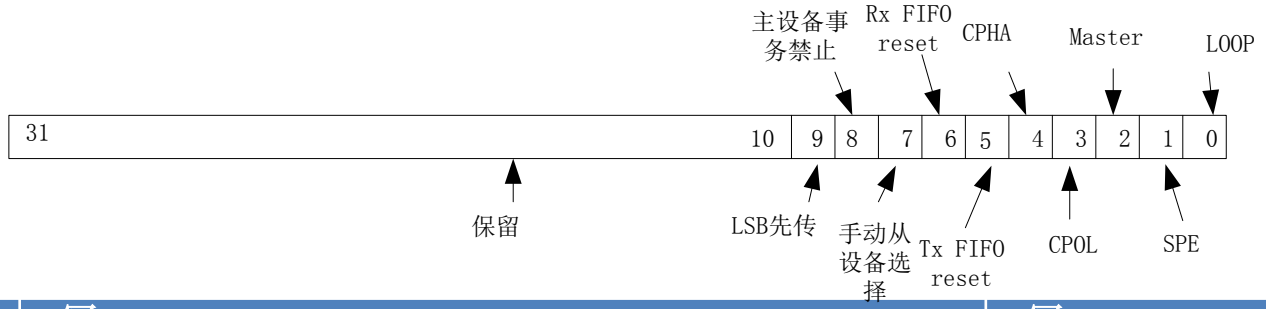
AXI SPI总线接口简介



SPI接口寄存器

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

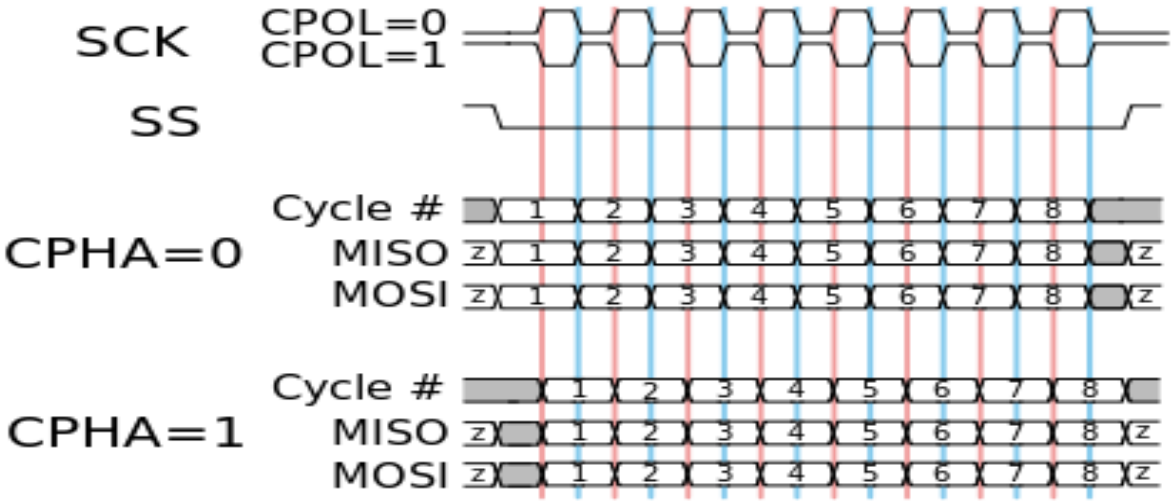
SPICR寄存器



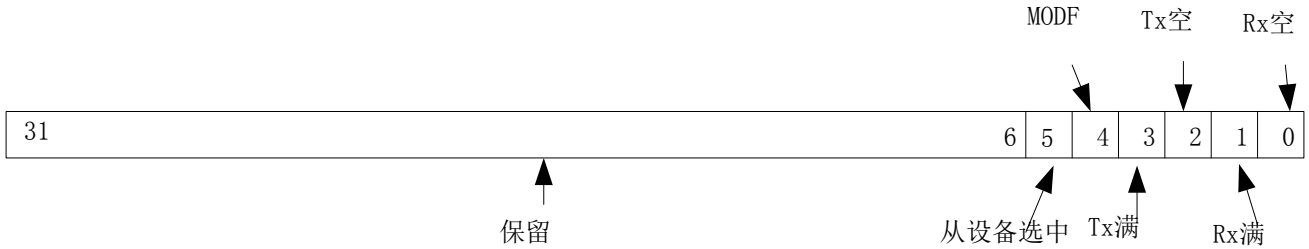
Bit位置	写1	写0
0	SPI发送端与接收端在内部形成环路	正常工作
1	使能SPI接口	停止SPI接口
2	配置为主设备	配置为从设备
3	空闲时时钟为高电平	空闲时时钟为低电平
4	数据在第二个时钟周期开始有效	数据在第一个时钟开始有效
5	复位发送FIFO指针	无影响
6	复位接收FIFO指针	无影响
7	配置为手动控制，根据SPISSR寄存器的值输出SS	根据内部逻辑自动输出SS
8	禁止主设备事务；若为从设备则无影响	使能主设备事务
9	串行数据低位优先传送	串行数据高位优先传送

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）

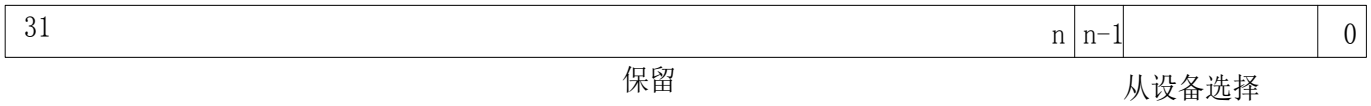


SPISR寄存器



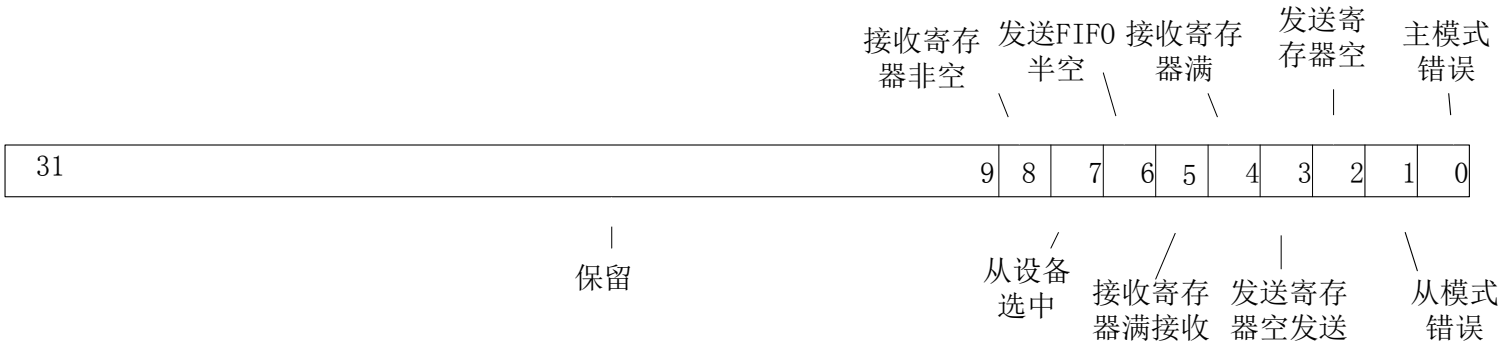
Bit位置	读0	读1
0	接收寄存器/FIFO非空	接收寄存器/FIFO空
1	接收寄存器/FIFO未满	接收寄存器/FIFO满
2	发送寄存器/FIFO非空	发送寄存器/FIFO空
3	发送寄存器/FIFO未满	发送寄存器/FIFO满
4	没有模式错误	SPI设备配置为主设备，但是SS引脚输入低电平
5	从设备选中	默认值，未被选中

SPISSR寄存器



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

IPISR寄存器


接收寄存器非空
发送FIFO半空
接收寄存器满
发送寄存器空
主模式错误
31
9
8
7
6
5
4
3
2
1
0
保留
从设备选中
接收寄存器满接收
发送寄存器空发送
从模式错误

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

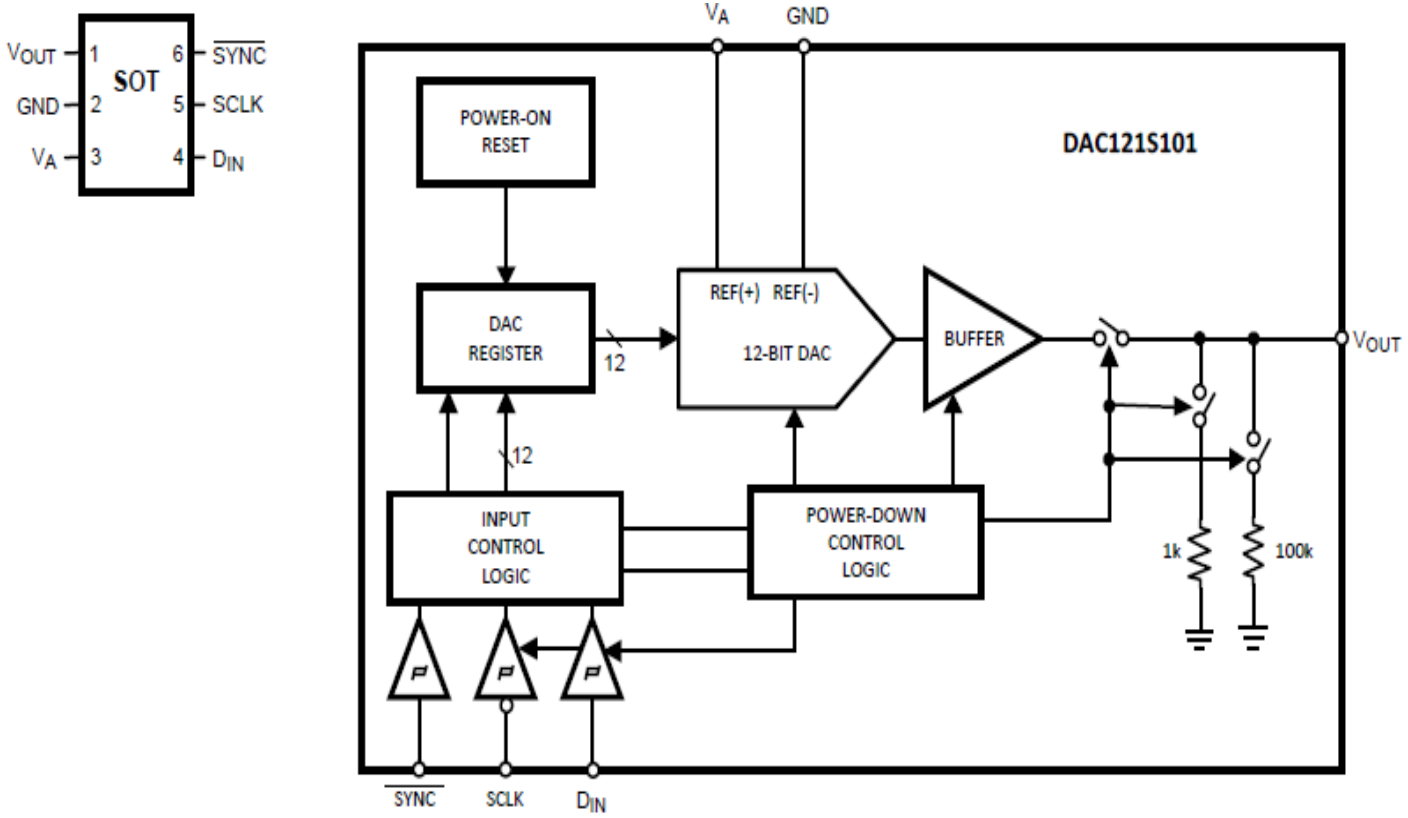
- 根据应用需要配置DGIER以及IPIER，实现中断使能控制；
- 将要传输的数据写入SPIDTR寄存器或FIFO；
- 将SPISSR预置为全1
- 配置SPICR，初始化SCLK以及MOSI，但禁止数据传输
  - SPICR（bit7）=1，手动控制SS（N）
  - SPICR（bit1）=1，使能SPI
  - SPICR（bit8）=1，禁止主设备事务
  - SPICR（bit2）=1，配置为主设备
  - SPICR（bit3）=0，空闲时时钟为低电平
  - SPICR（bit4）=1，上升沿输出数据，下降沿采样数据
  - SPICR（bit0）=0，非内部循环
- 写SPISSR，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
- 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。

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

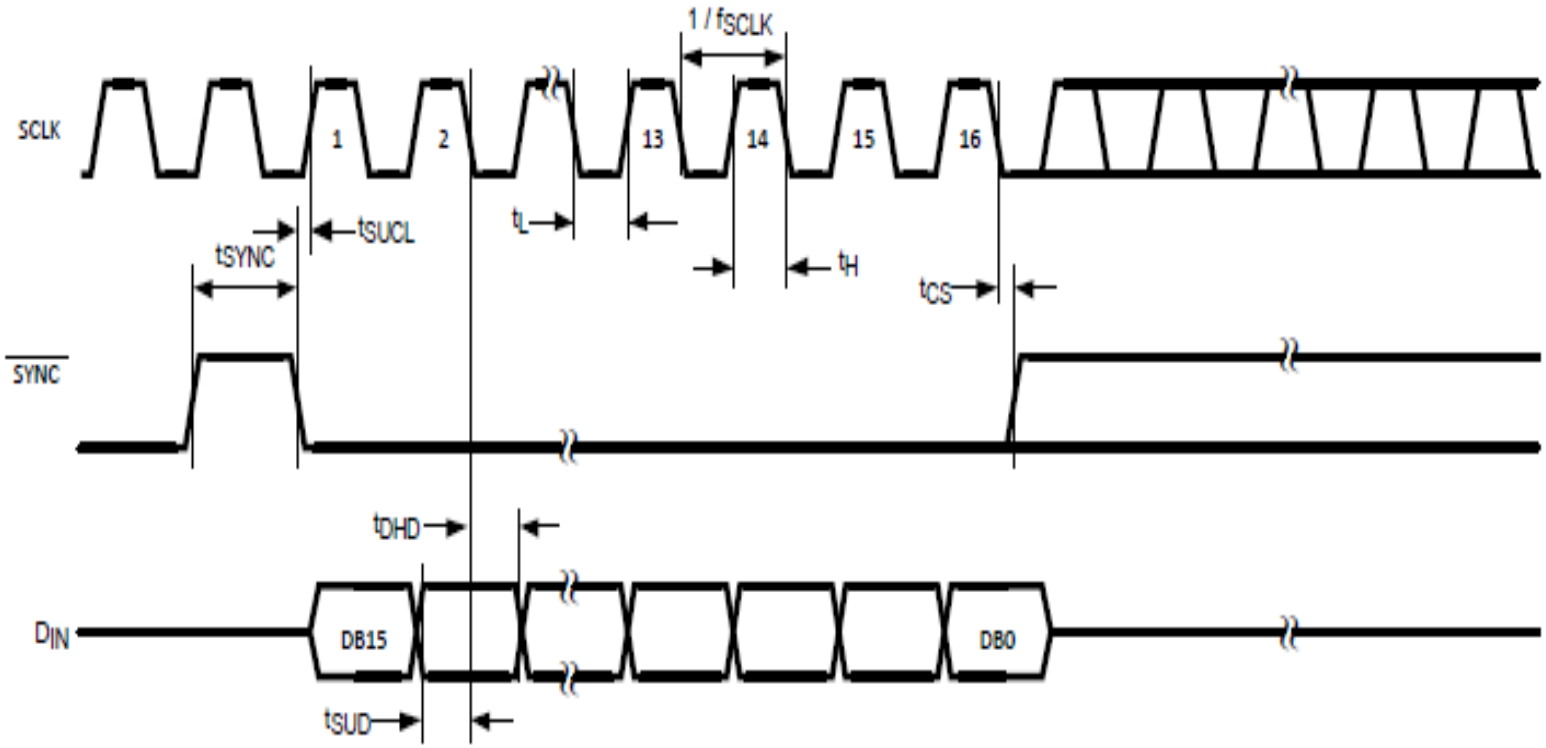
DAC121S101简介



DAC121S101引脚含义

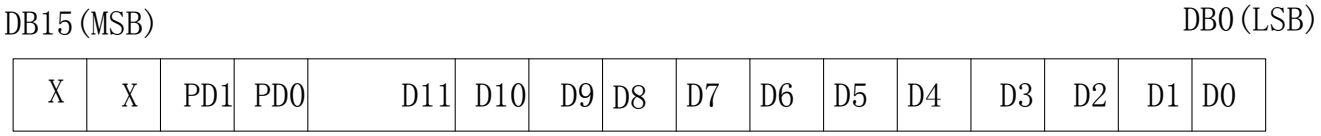
引脚名称	含义
V_{OUT}	模拟电压输出
GND	地
V_A	模拟参考电压
$\overline{SYNC}$	帧数据同步，当该引脚为低电平时，数据在SCLK的下降沿输入，并且16个时钟周期之后，移位寄存器的数据进入DAC寄存器，开始DA转换；若该引脚在16个时钟周期之前变为高电平，那么之前传入的数据都将被忽略。
SCLK	SPI总线时钟，数据在该时钟的下降沿采样。时钟频率最高为30MHZ
D_{IN}	SPI总线从设备数据输入线，相当于MOSI

时序



任何两次写操作之间必须使$\overline{\text{SYNC}}$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

16位串行数据的具体含义

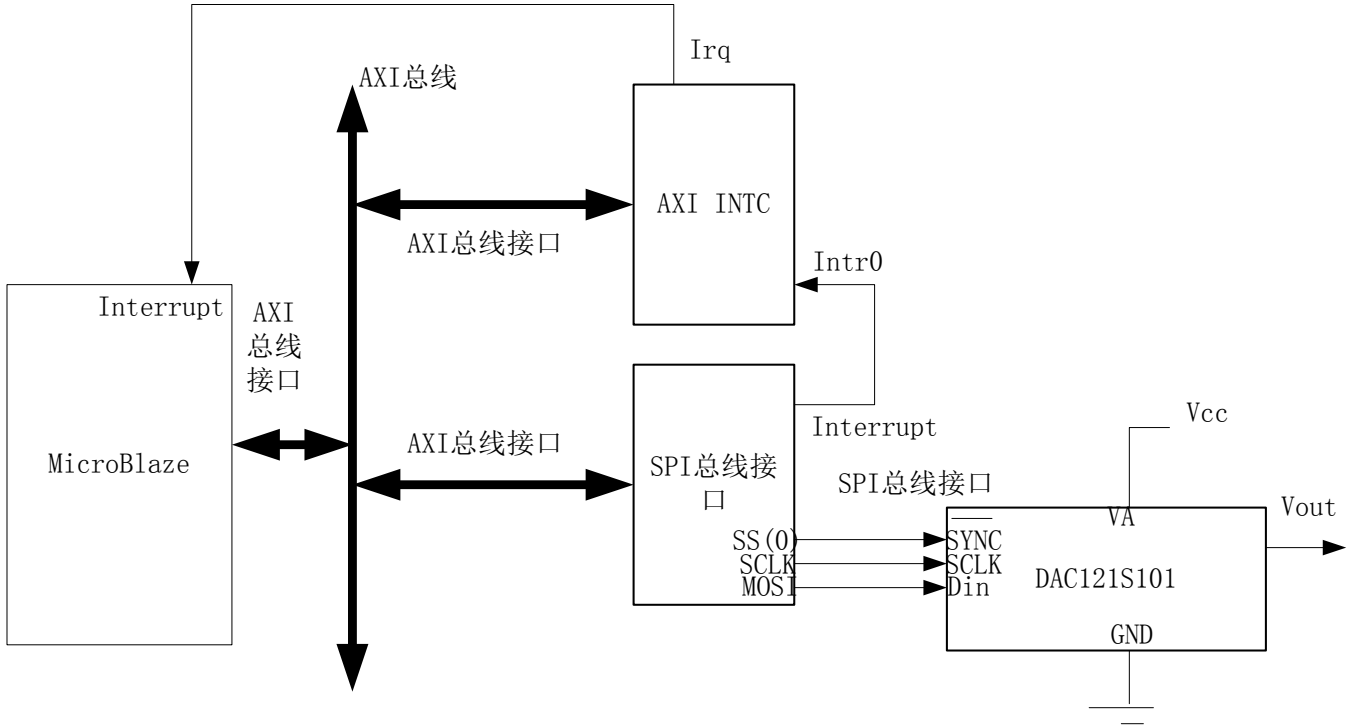


D0~D11为12位DA转换的数字量，
PD0~PD1为电源下拉控制逻辑的输入，控制电源下拉模块的工作方式，改变输出Vout的输出连接方式

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

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$\overline{\text{SYNC}}$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。

SPI初始化函数

```
int SpiIntrExample(XIntc *IntcInstancePtr, XSpi *SpiInstancePtr, u16 SpiDeviceId, u16 SpiIntrId)
{
        int Status;
        u32 Count;
        XSpi_Config *ConfigPtr;
//      查找SPI接口配置项
        ConfigPtr = XSpi_LookupConfig(SpiDeviceId);
        //根据配置项参数，初始化SPI接口参数
        Status = XSpi_CfgInitialize(SpiInstancePtr, ConfigPtr, ConfigPtr->BaseAddress);
        //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);
        //设置从设备选择信号
        Status = XSpi_SetSlaveSelect(SpiInstancePtr, 1);
        //使能SPI接口
        XSpi_Start(SpiInstancePtr);
        return XST_SUCCESS;
}
```

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
Status = XIntc_Connect(IntcInstancePtr, SpiIntrId, (XInterruptHandler) XSpi_InterruptHandler, (void
*)SpiInstancePtr);
//配置相应中断输入引脚的中断处理函数
        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
microblaze_register_handler((XInterruptHandler)XIntc_InterruptHandler, IntcInstancePtr);
//注册中断控制器的总中断处理函数
        microblaze_enable_interrupts();
//开放微处理器中断
        return XST_SUCCESS;
}
```

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++;
        }
}
```

主函数

```
int main(void)
{               int Status;
                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;
}
```

扩展

- 锯齿波的周期
- 锯齿波的电压范围
- 输出波形

作业

- 7， 8, 10