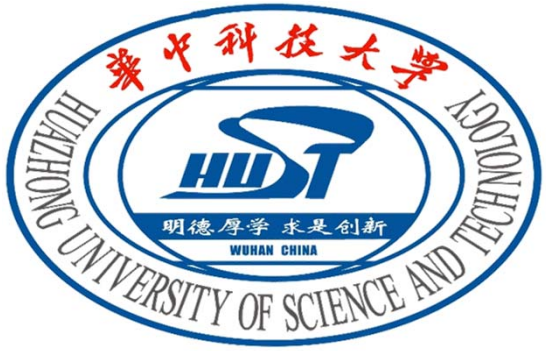
微机原理与接口技术

中断控制器构成

华中科技大学　左冬红



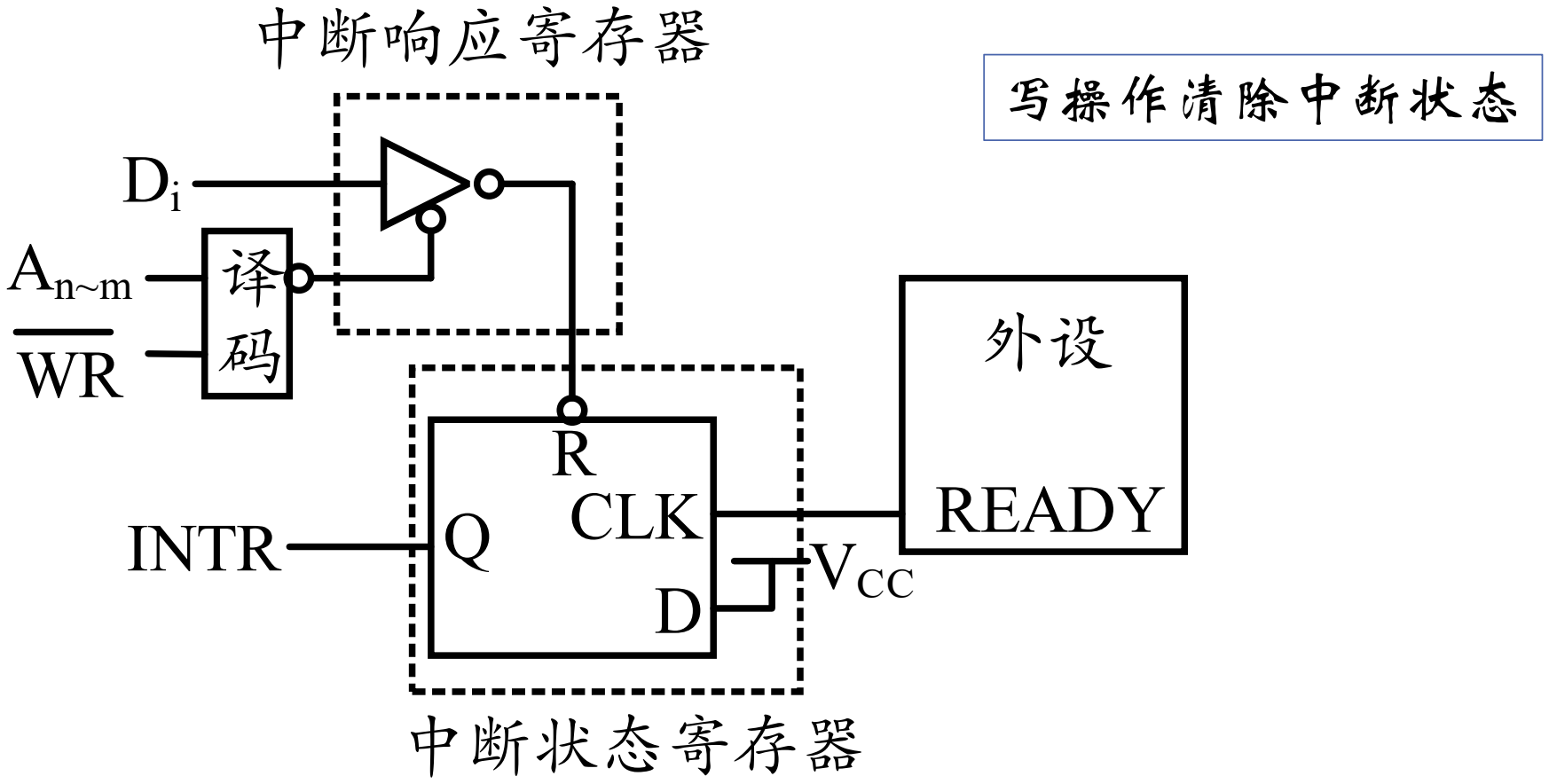
术语

- 中断源：引起中断的原因或发出中断请求的来源
- 中断类型码：中断源的编码
- 中断向量：中断服务程序入口地址
- 中断触发方式：有效中断信号
  - 边沿触发
    - 上升边沿触发
    - 下降边沿触发
  - 电平触发
    - 高电平触发
    - 低电平触发

中断控制器基本功能

- 中断请求信号保持与清除
- 中断源识别
- 中断使能控制
- 中断优先级设置

中断请求信号保持与清除电路



中断源识别

CPU一个中断请求信号线

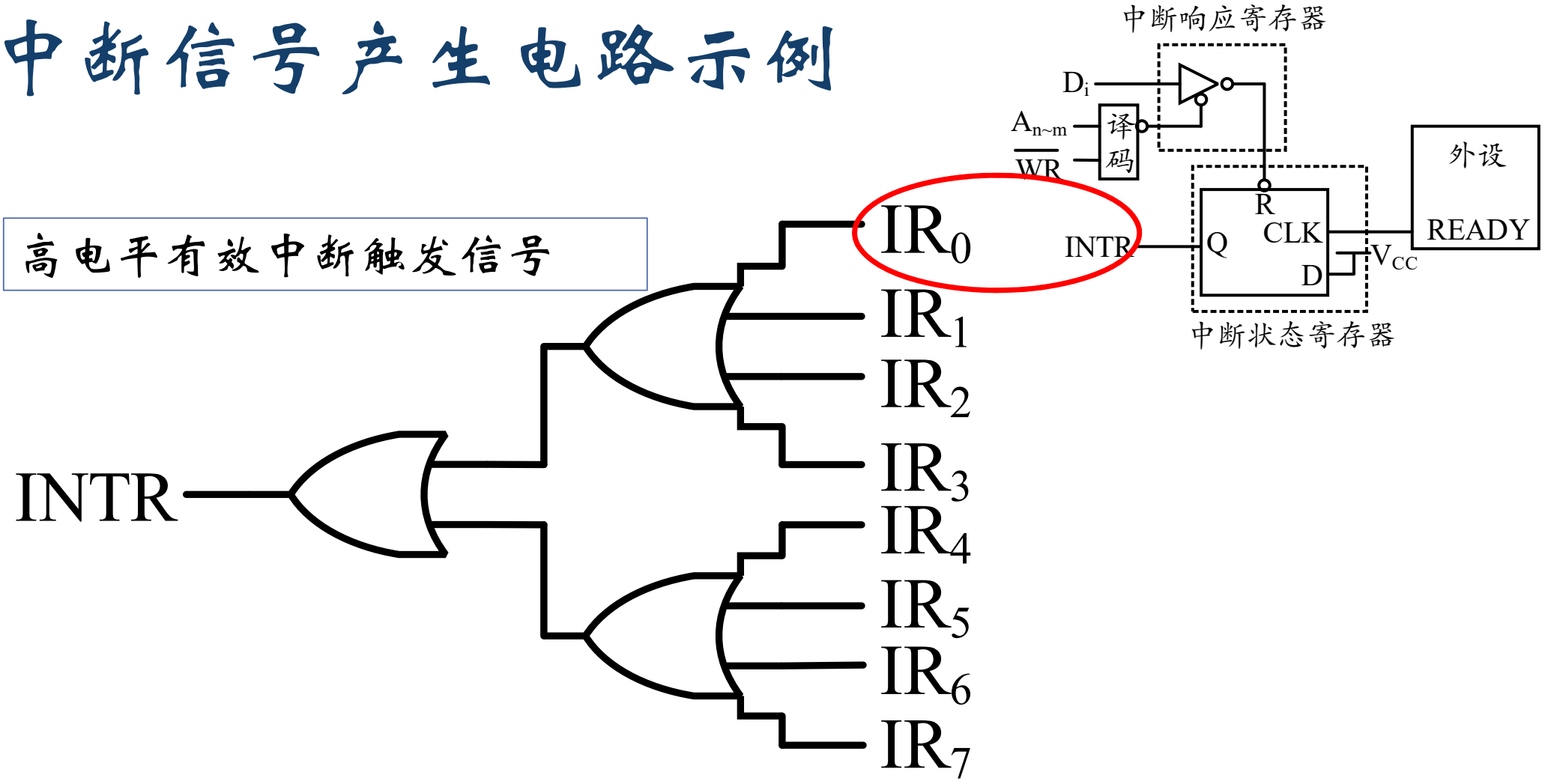
- 外设0一个中断请求输出
- 外设1一个中断请求输出
- 外设2一个中断请求输出
- 外设n一个中断请求输出

逻辑电路功能

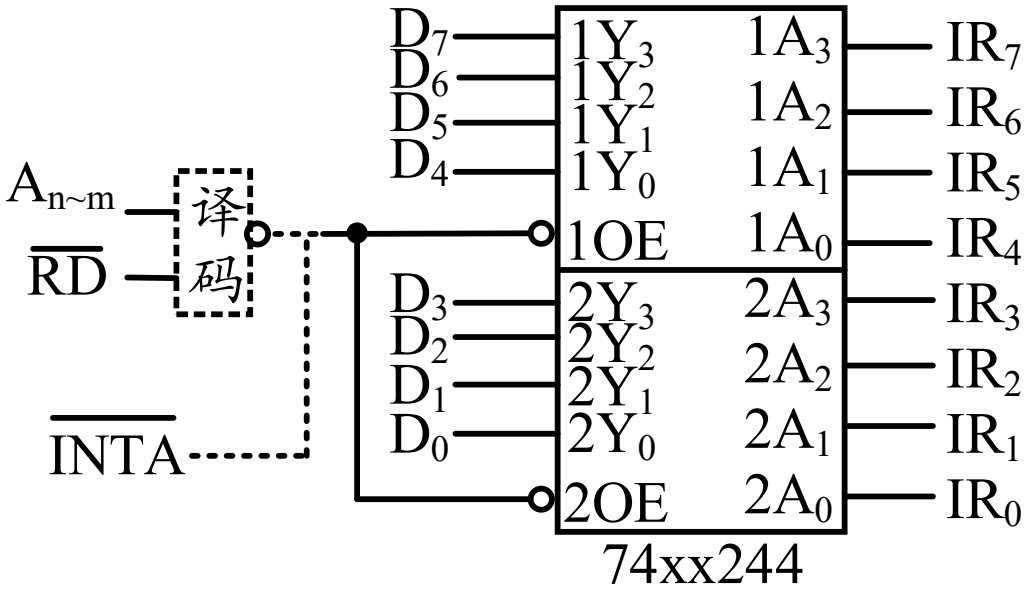
任何一个外设产生了中断请求，CPU都能接收到中断请求，并能识别出哪个外设产生中断请求

中断信号产生电路示例

高电平有效中断触发信号

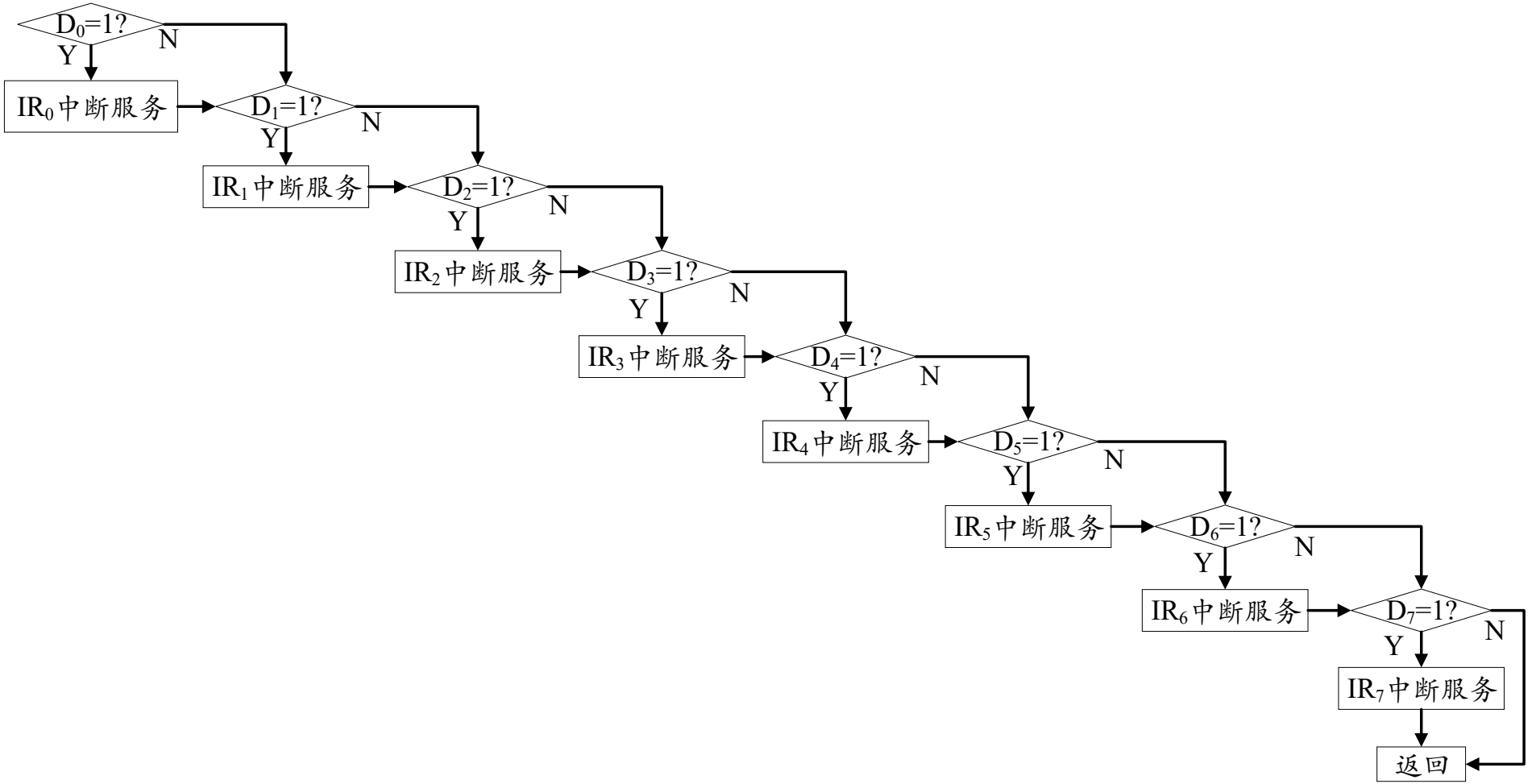


中断源识别——软件查询

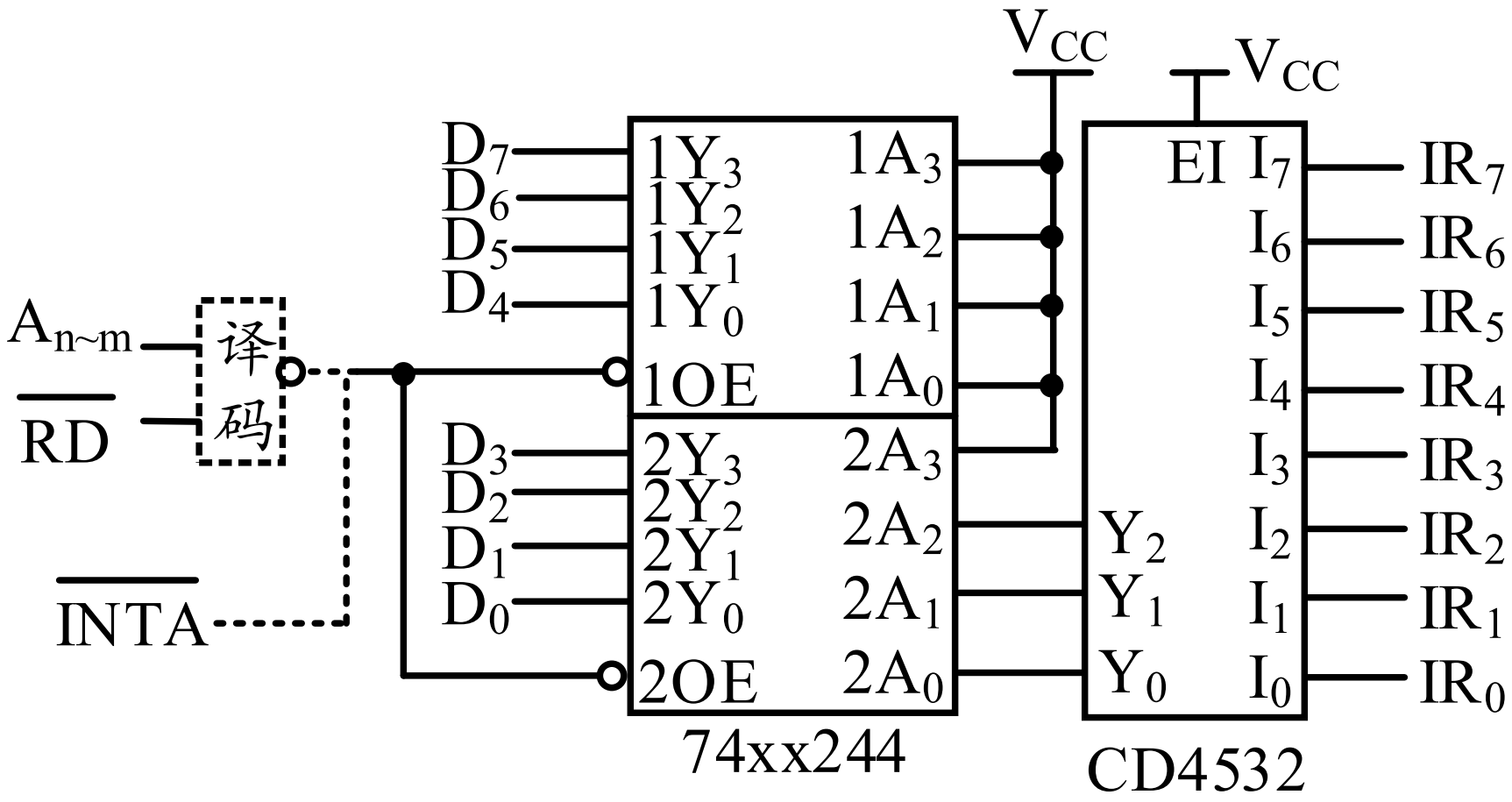


中断源	中断类型码								
	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	值
IR_0	0	0	0	0	0	0	0	1	0x1
IR_1	0	0	0	0	0	0	1	0	0x2
IR_2	0	0	0	0	0	1	0	0	0x4
IR_3	0	0	0	0	1	0	0	0	0x8
IR_4	0	0	0	1	0	0	0	0	0x10
IR_5	0	0	1	0	0	0	0	0	0x20
IR_6	0	1	0	0	0	0	0	0	0x40
IR_7	1	0	0	0	0	0	0	0	0x80

中断优先级


D0=1?
Y
N
IR0中断服务
D1=1?
Y
N
IR1中断服务
D2=1?
Y
N
IR2中断服务
D3=1?
Y
N
IR3中断服务
D4=1?
Y
N
IR4中断服务
D5=1?
Y
N
IR5中断服务
D6=1?
Y
N
IR6中断服务
D7=1?
Y
N
IR7中断服务
返回

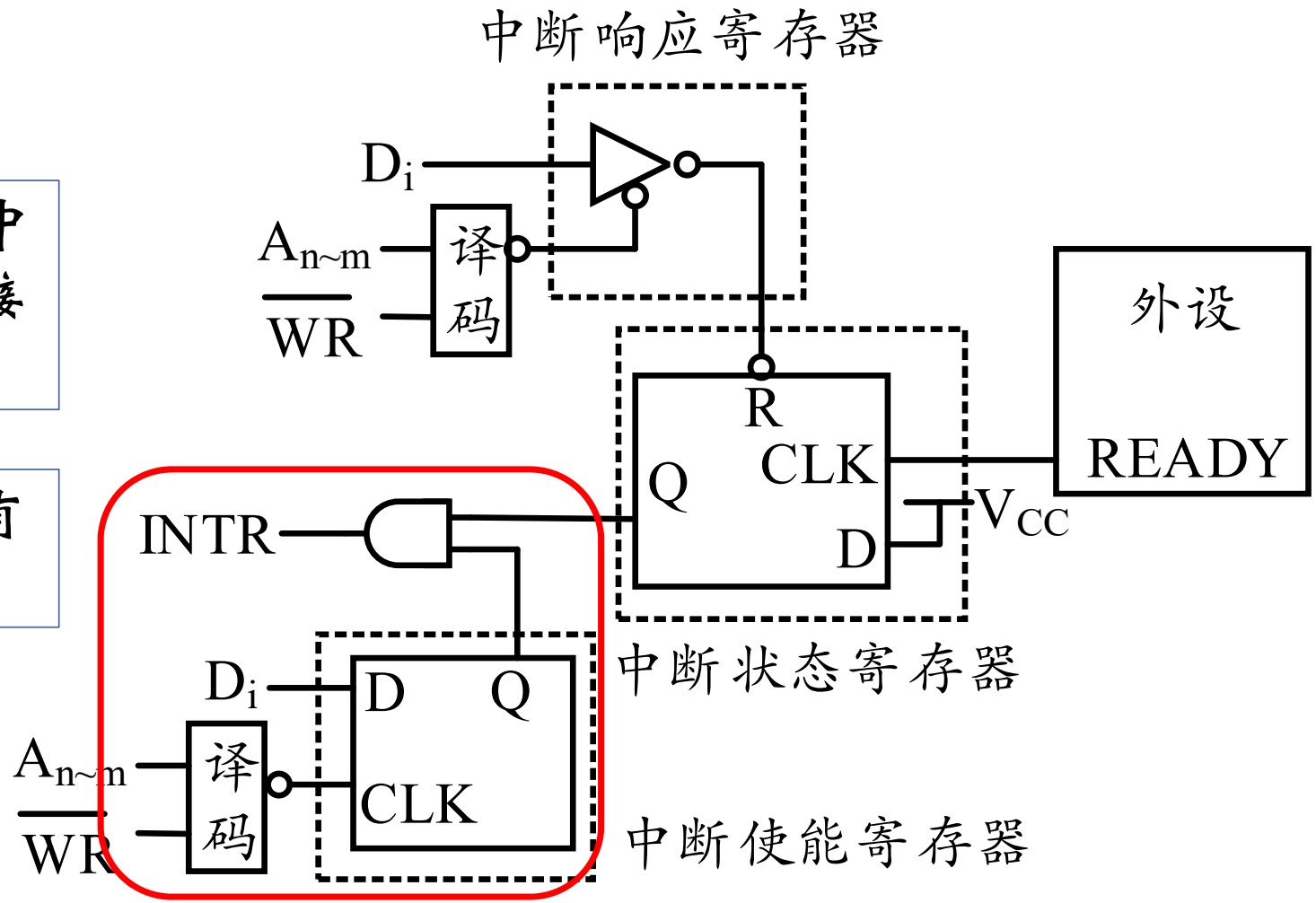
中断源识别、优先级编码


VCC
VCC
D7
D6
D5
D4
1Y3
1Y2
1Y1
1Y0
1OE
1A3
1A2
1A1
1A0
An~m
译
码
RD
D3
D2
D1
D0
2Y3
2Y2
2Y1
2Y0
2OE
2A3
2A2
2A1
2A0
INTA
74xx244
EI
I7
I6
I5
I4
I3
I2
I1
I0
Y2
Y1
Y0
IR7
IR6
IR5
IR4
IR3
IR2
IR1
IR0
CD4532

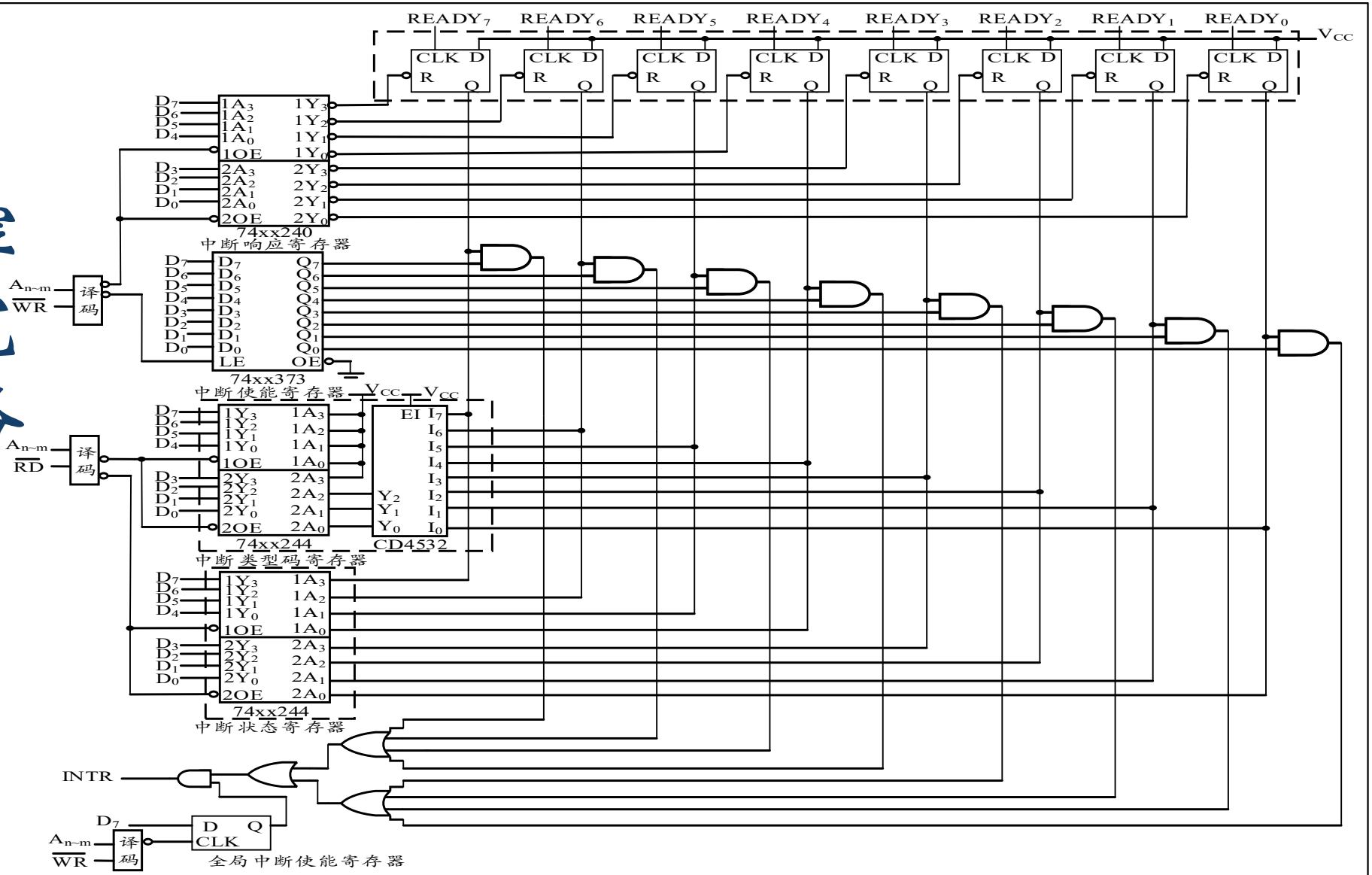
中断控制

控制各个外设的中断是否可被CPU接收到

CPU也可控制所有外设中断请求



中断控制器完整电路示例



小结

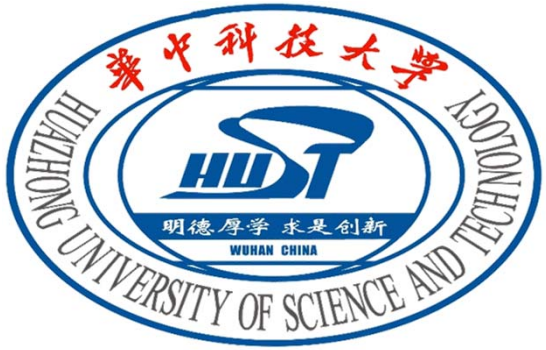
- 中断相关术语
  - 中断源
  - 中断类型码
  - 中断向量
- 中断控制器基本构成
  - 中断状态寄存器
  - 中断响应寄存器
  - 中断使能寄存器

下一讲： AXI INTC中断控制器

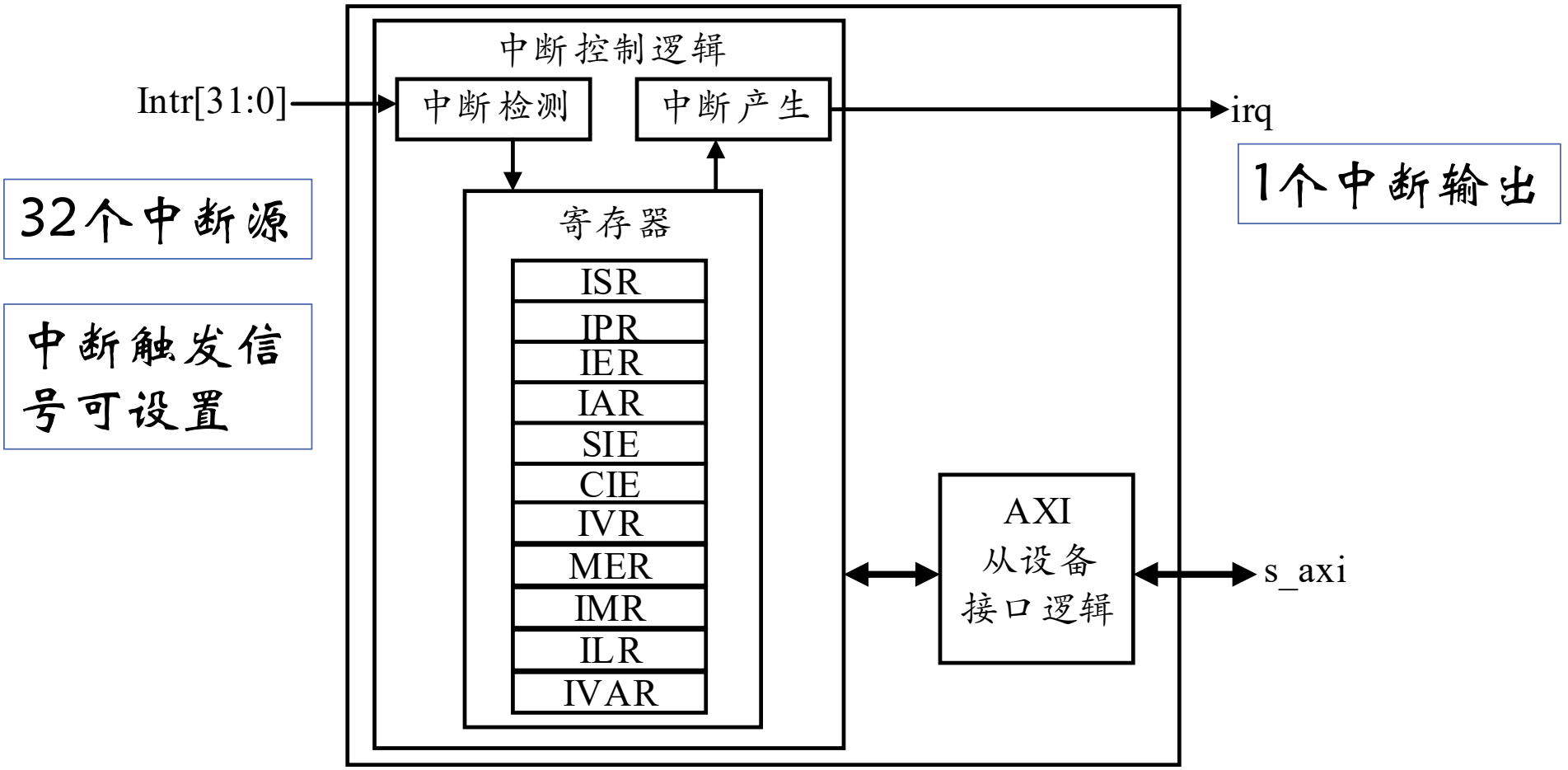
微机原理与接口技术

AXI INTC中断控制器

华中科技大学　左冬红



INTC基本结构框图


中断控制逻辑
Intr[31:0]
中断检测
中断产生
irq
寄存器
ISR
IPR
IER
IAR
SIE
CIE
IVR
MER
IMR
ILR
IVAR
AXI
从设备
接口逻辑
s_axi
32个中断源
中断触发信号可设置
1个中断输出

INTC工作模式

普通中断模式：由总中断服务程序软件查询中断状态寄存器识别中断源、由总中断服务程序软件调用具体中断源的中断服务程序，总中断服务程序软件维护中断向量表

快速中断模式：由硬件识别中断源，并在CPU的中断响应周期直接输出中断源的中断向量，INTC硬件提供中断向量表IVAR

INTC寄存器存储空间映射

寄存器名称	偏移地址	含义
ISR	0x0	中断状态寄存器
IPR*	0x4	中断悬挂寄存器
IER	0x8	中断使能寄存器
IAR	0xC	中断响应寄存器
SIE*	0x10	中断使能设置寄存器
CIE*	0x14	中断使能清除寄存器
IVR*	0x18	中断类型码寄存器
MER	0x1C	主中断使能寄存器
IMR*	0x20	中断模式寄存器
ILR*	0x24	中断级别寄存器
IVAR*	0x100~0x170	中断向量表寄存器

INTC寄存器含义

ISR\IER\IAR\IPR\SIE\CIE\IMR

寄存器的D_i对应中断源引脚$Intr_i$

都是1有效

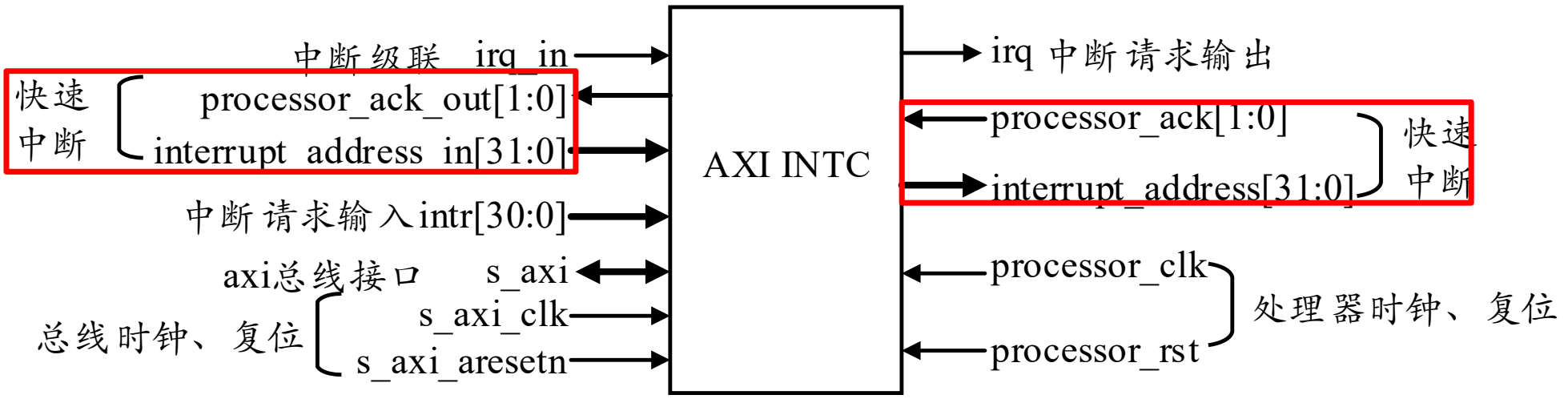
IVR 寄存器保存最高优先级中断源的编码

所有中断源使能，且中断请求输入引脚intr[0]、intr[1]、intr[2]都没有产生中断请求，intr[3]产生了中断请求，此时IVR的值为0x3；如果之后intr[0]产生中断请求，那么IVR的值变为0x0。

INTC寄存器含义

- ILR
  - 寄存器保存阻止的最高优先级中断源的编码
  - ILR为0x0阻止所有中断源
  - ILR为3表示阻止中断源intr[31:3]，仅允许中断源intr[2:0]
- IVAR
  - 共32个寄存器，每个寄存器都是4B
  - 中断请求引脚intr[i]与IVAR偏移地址EA之间关系为：EA=0x100+4×i
- MER
  - HIE(D1)为1表示使能硬件中断
  - ME(D0)为1表示允许Irq产生中断请求信号

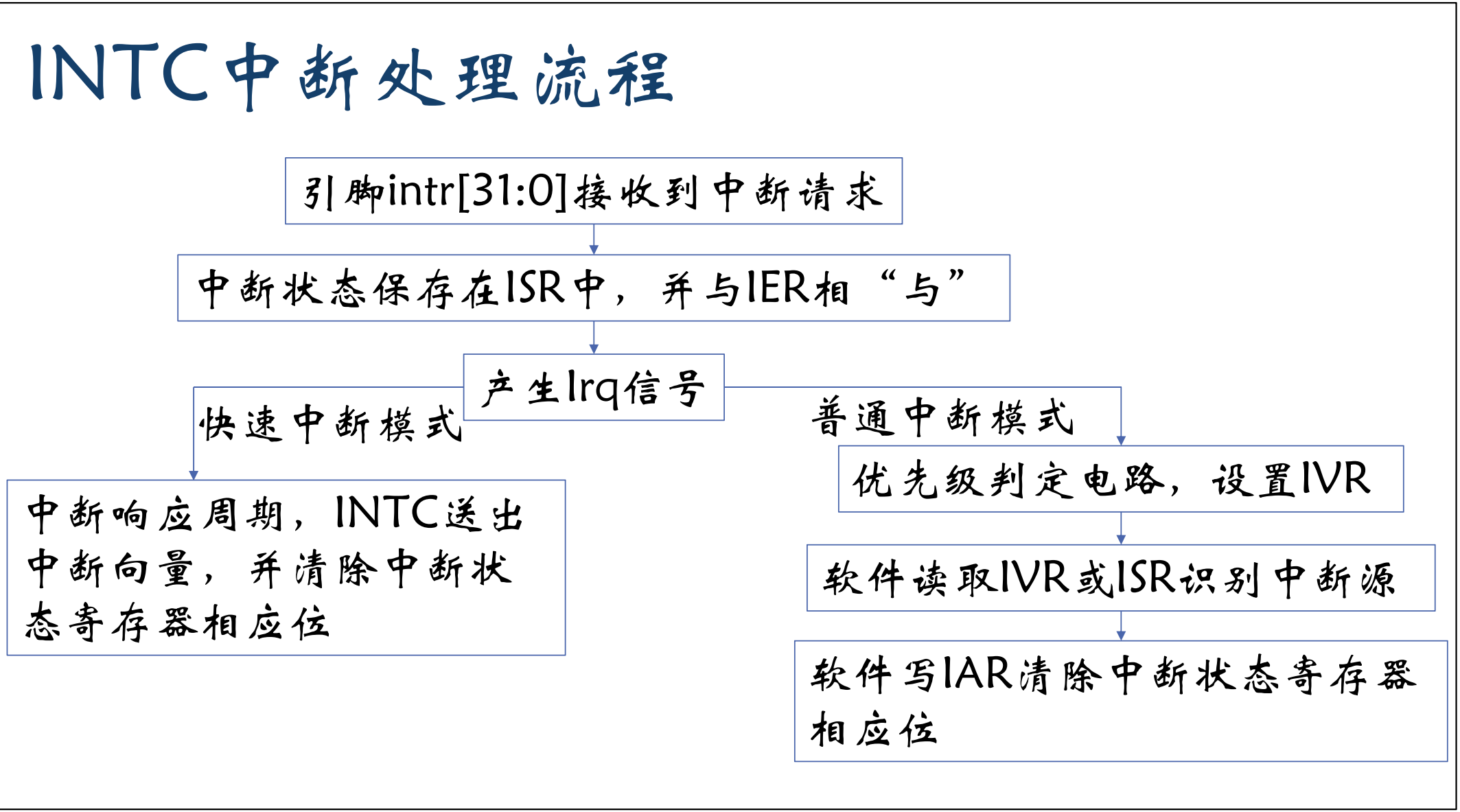
INTC外部引脚



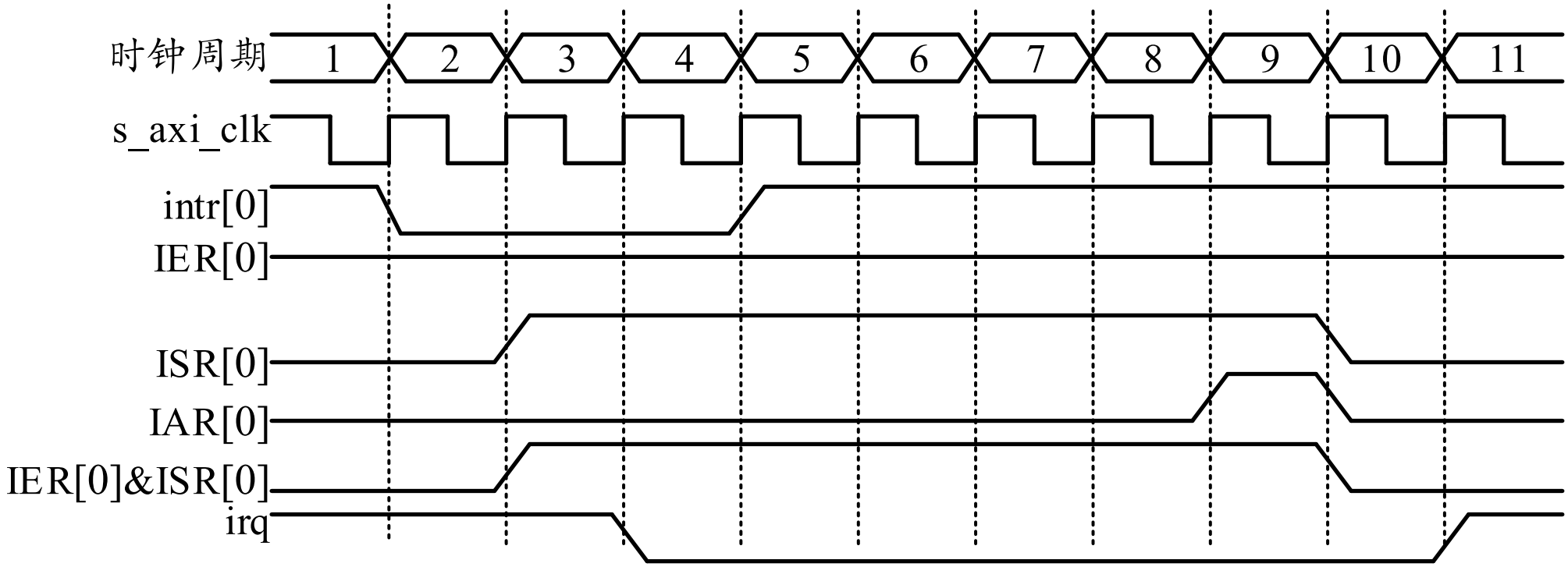
普通中断模式：由软件查询ISR识别中断源、由软件调用具体中断源的中断服务程序

快速中断模式：由硬件识别中断源，并通过Interrupt_address输出相应中断源的中断向量

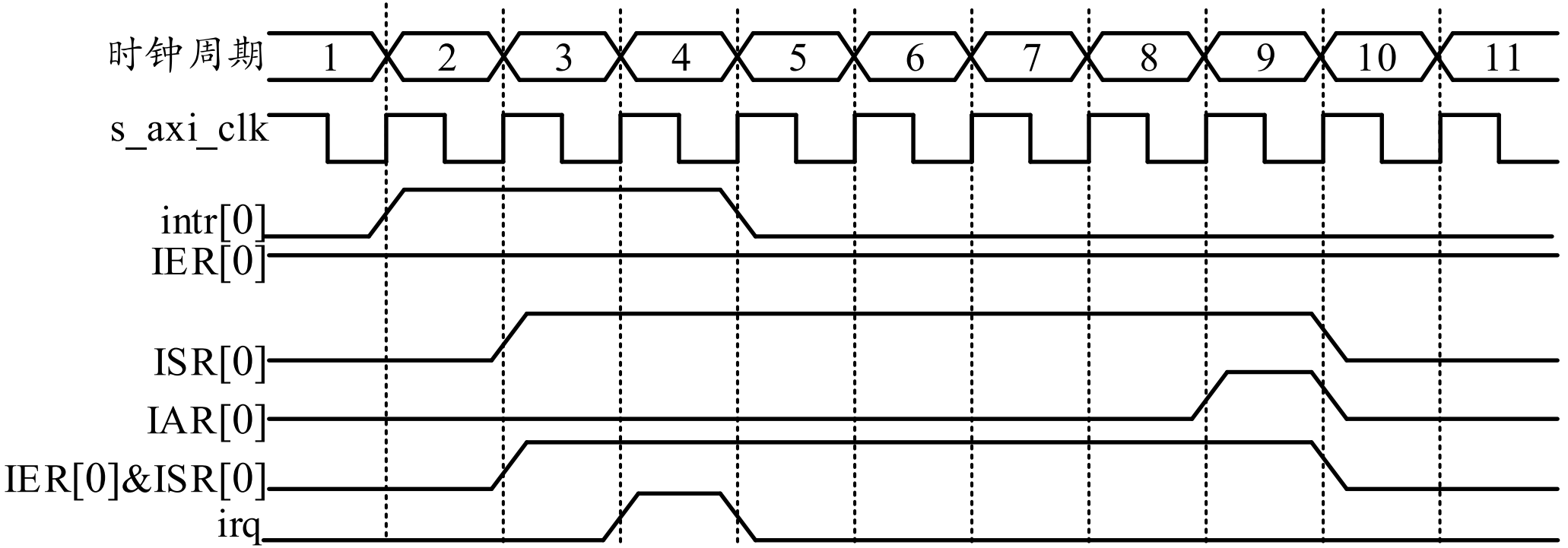
INTC中断处理流程


引脚intr[31:0]接收到中断请求
中断状态保存在ISR中，并与IER相“与”
产生Irq信号
快速中断模式
中断响应周期，INTC送出中断向量，并清除中断状态寄存器相应位
普通中断模式
优先级判定电路，设置IVR
软件读取IVR或ISR识别中断源
软件写IAR清除中断状态寄存器相应位

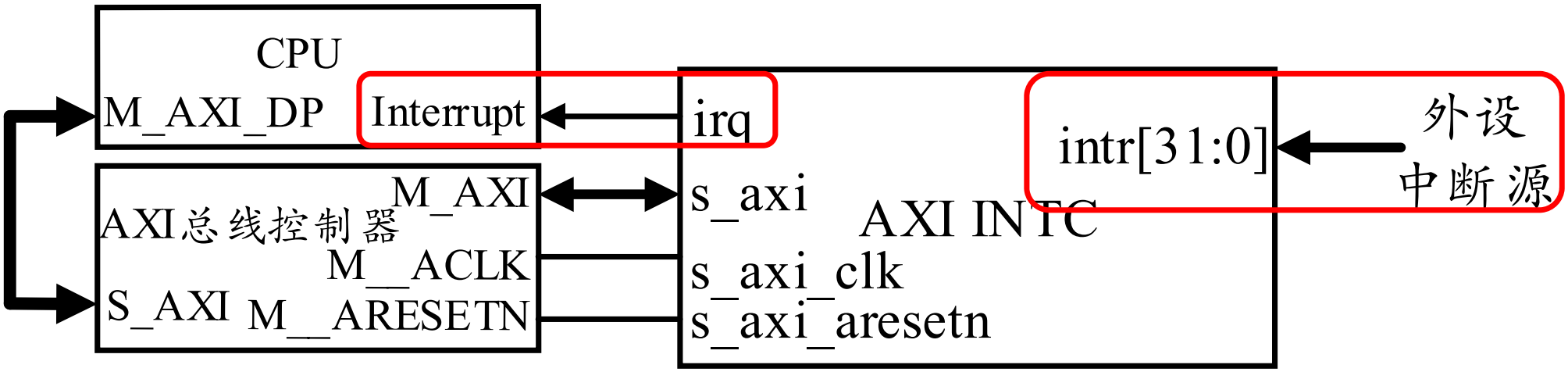
INTC中断信号产生时序——低电平有效



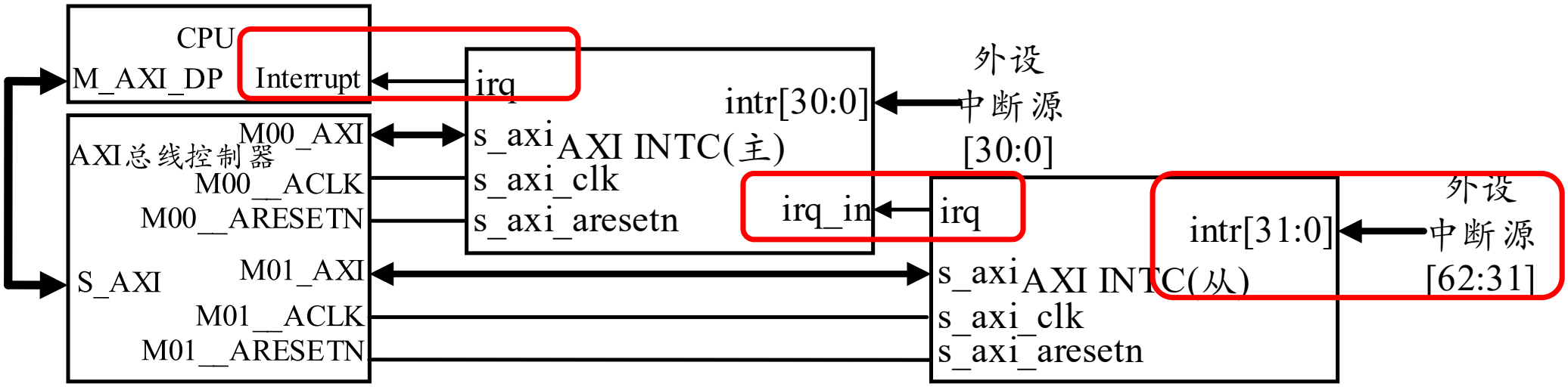
INTC中断信号产生时序——上升沿有效



INTC应用电路-单模块普通中断模式

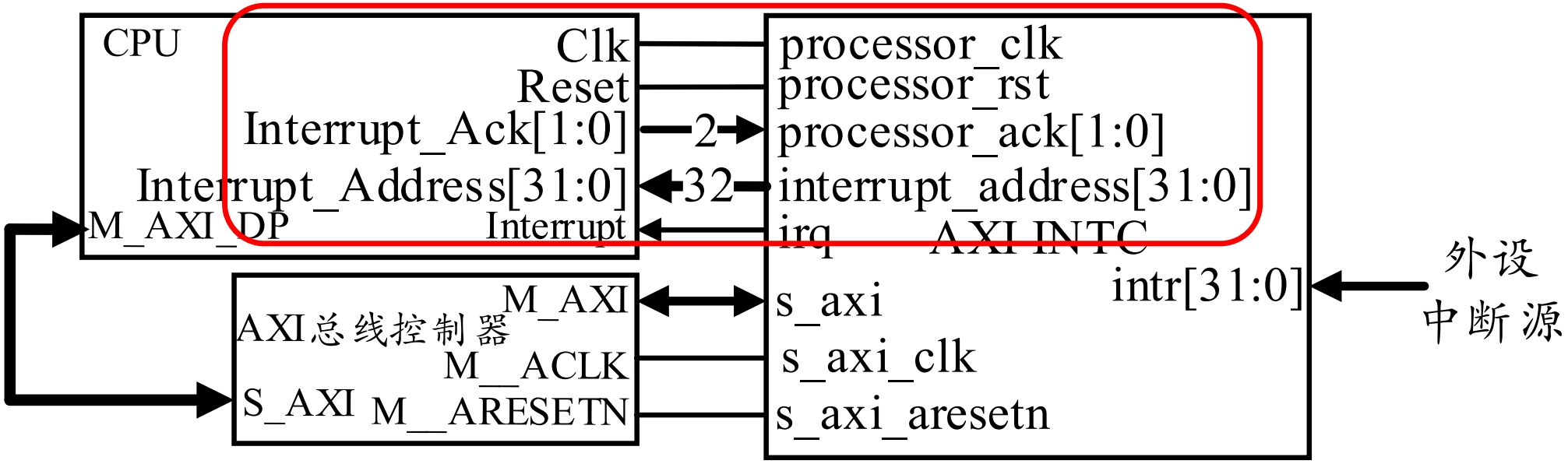


INTC应用电路-级联普通中断模式



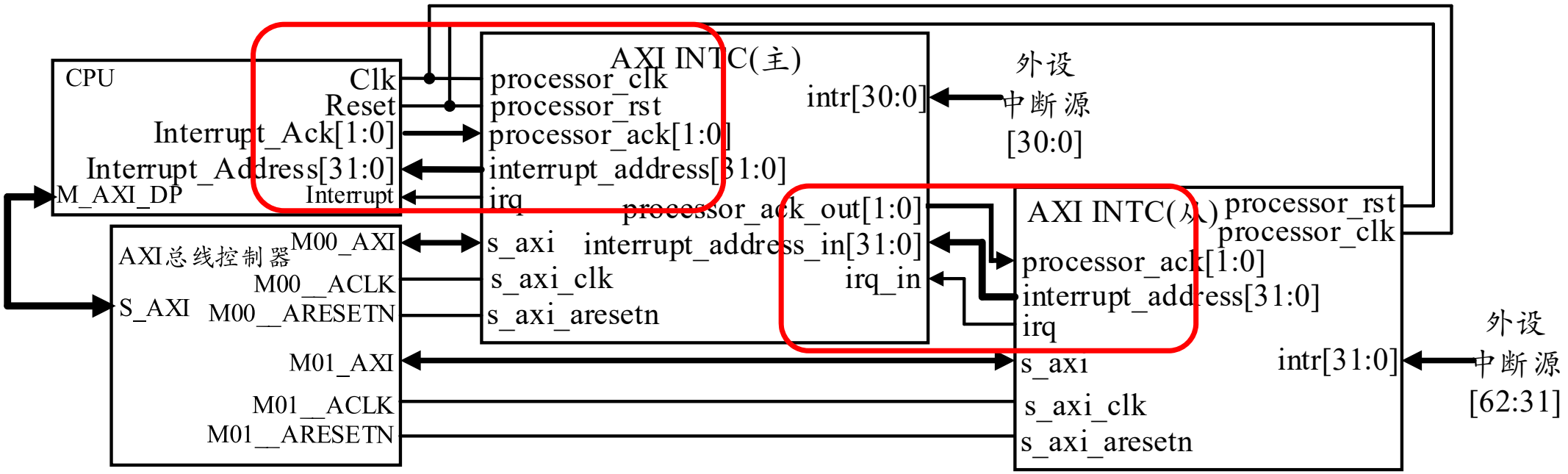
63个中断源

INTC应用电路-单模块快速中断模式

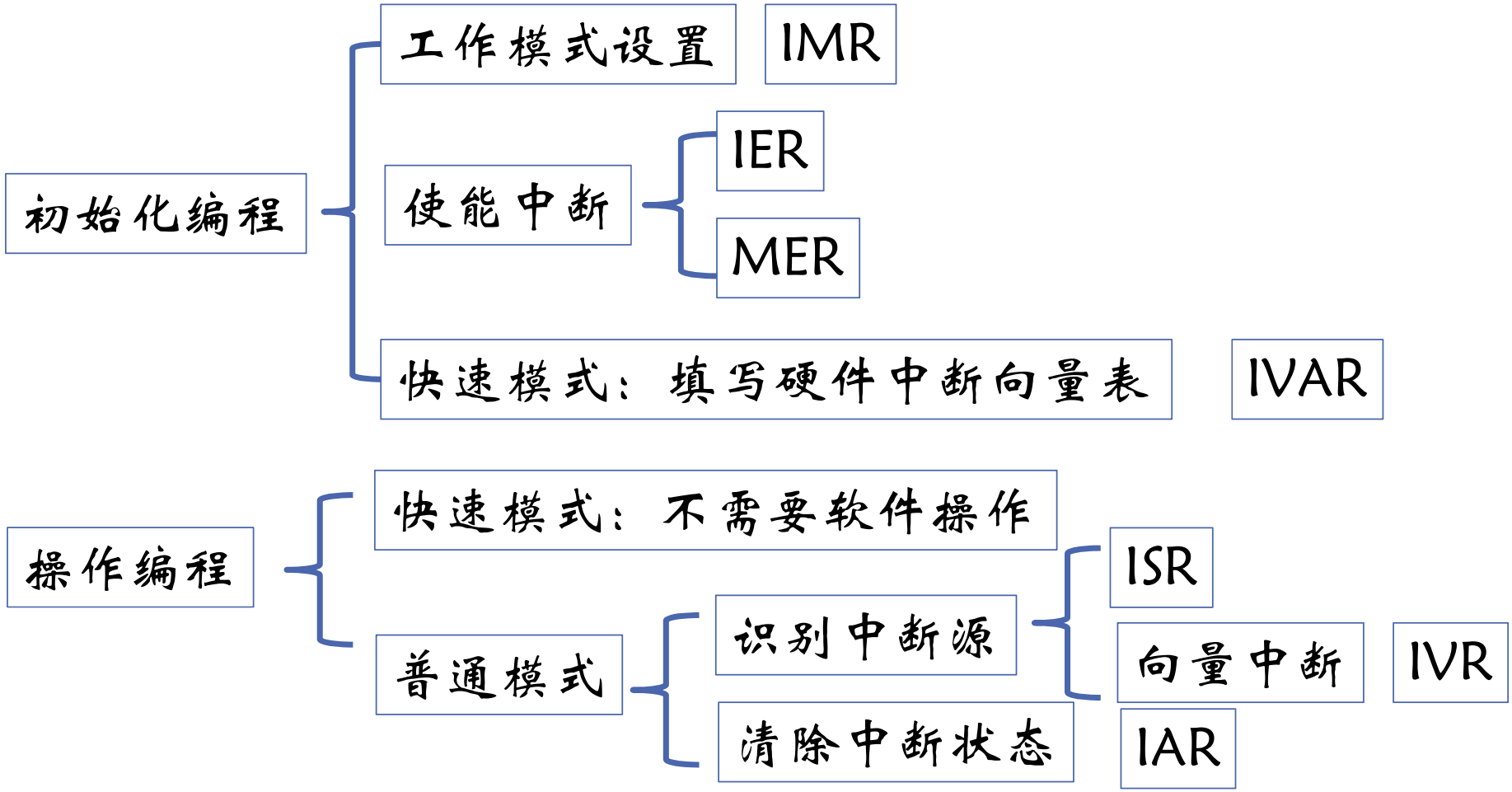


中断总线

INTC应用电路-级联快速中断模式



INTC编程控制


初始化编程
工作模式设置
IMR
使能中断
IER
MER
快速模式：填写硬件中断向量表
IVAR
操作编程
快速模式：不需要软件操作
普通模式
识别中断源
ISR
向量中断
IVR
清除中断状态
IAR

单模块INTC编程示例

INTC基地址为0x41200000，若INTC中断请求输入引脚intr[1:0]连接了中断源，编写INTC普通中断模式初始化控制程序段

寄存器名称	偏移地址	含义
ISR	0x0	中断状态寄存器
IPR*	0x4	中断悬挂寄存器
IER	0x8	中断使能寄存器
IAR	0xC	中断响应寄存器
SIE*	0x10	中断使能设置寄存器
CIE*	0x14	中断使能清除寄存器
IVR*	0x18	中断类型码寄存器
MER	0x1C	主中断使能寄存器
IMR*	0x20	中断模式寄存器
ILR*	0x24	中断级别寄存器
IVAR*	0x100~0x170	中断向量表寄存器

```
Xil_Out32(0x41200020, 0x0);
//普通中断模式
Xil_Out32(0x41200008, 0x3);
//使能intr[1:0]
 Xil_Out32(0x4120001c, 0x3);
//使能硬件中断输出
```

单模块INTC编程示例

INTC基地址为0x41200000，若INTC中断请求输入引脚intr[1:0]连接了中断源，编写INTC快速中断模式初始化控制程序段

寄存器名称	偏移地址	含义
ISR	0x0	中断状态寄存器
IPR*	0x4	中断悬挂寄存器
IER	0x8	中断使能寄存器
IAR	0xC	中断响应寄存器
SIE*	0x10	中断使能设置寄存器
CIE*	0x14	中断使能清除寄存器
IVR*	0x18	中断类型码寄存器
MER	0x1C	主中断使能寄存器
IMR*	0x20	中断模式寄存器
ILR*	0x24	中断级别寄存器
IVAR*	0x100~0x170	中断向量表寄存器

```
Xil_Out32(0x41200020, 0x3);//
Xil_Out32(0x41200008, 0x3);//
Xil_Out32(0x4120001c, 0x3);//
Xil_Out32(0x41200100, ISR0); //
Xil_Out32(0x41200104, ISR1); //
```

小结

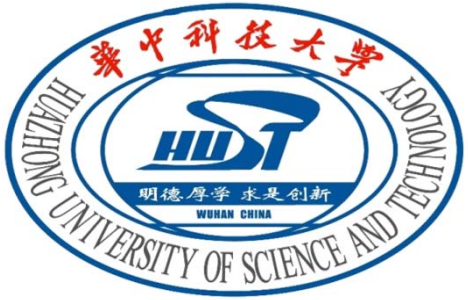
- INTC结构
- 工作模式
- 寄存器功能
- 应用电路
- 初始化编程

下一讲：微处理器中断控制

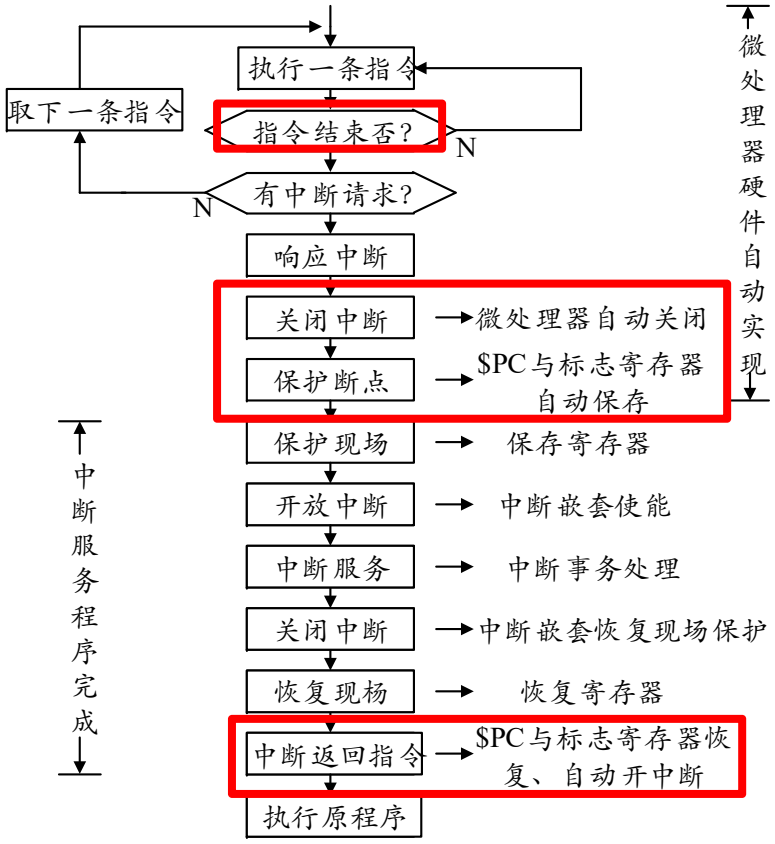
微机原理与接口技术

微处理器中断控制

华中科技大学　左冬红



微处理器中断响应一般流程



MicroBlaze微处理器开中断

机器状态寄存器MSR(Machine Status Register)位IE(D_1)

当IE为1时，MicroBlaze微处理器响应可屏蔽中断

MicroBlaze中断向量表

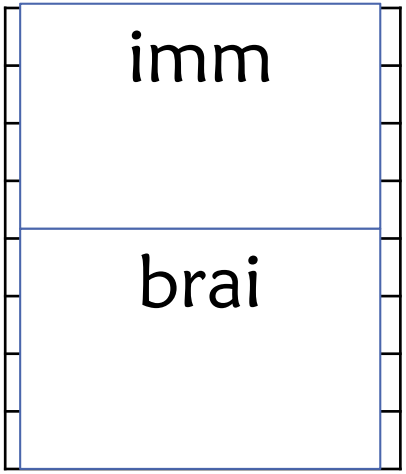
异常事件	中断向量	断点保存寄存器
复位	C_BASE_VECTORS+0x00000000	-
用户异常	C_BASE_VECTORS+0x00000008	Rx
可屏蔽中断	C_BASE_VECTORS+0x00000010 或INTC提供的中断向量	R14
打断：不可屏蔽中断、硬件打断、软件打断	C_BASE_VECTORS+0x00000018	R16
硬件异常	C_BASE_VECTORS+0x00000020	R17

MicroBlaze中断服务程序进入方式

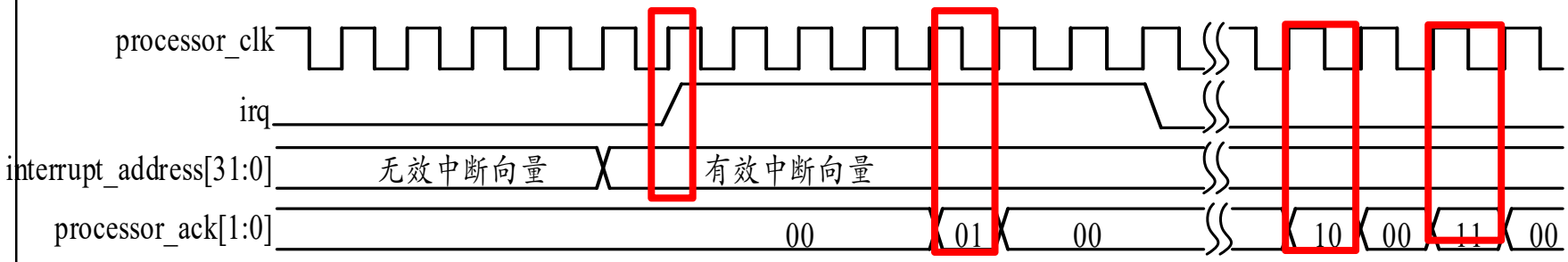
中断向量处8个字节保存两条MicroBlaze指令

imm指令：将立即数赋值到临时寄存器的指令

brai指令：立即数跳转指令



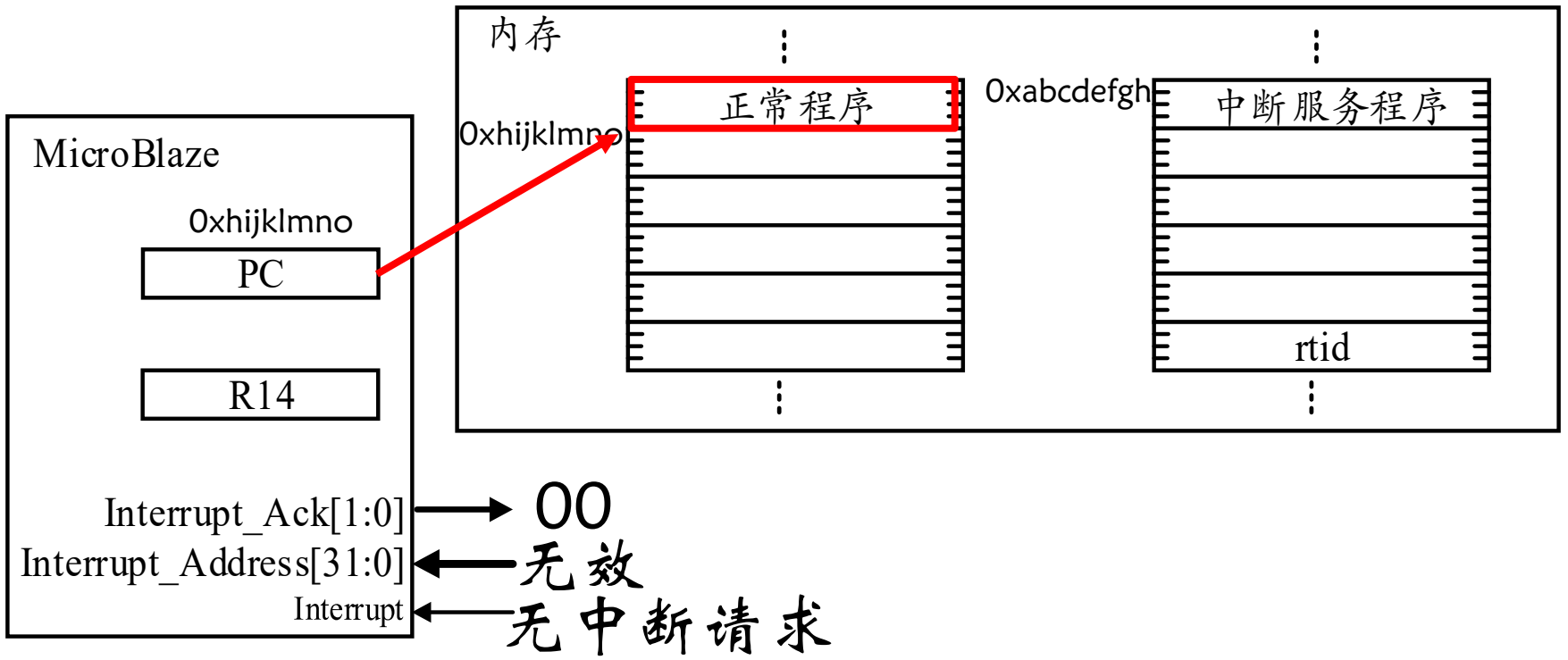
MicroBlaze中断响应周期



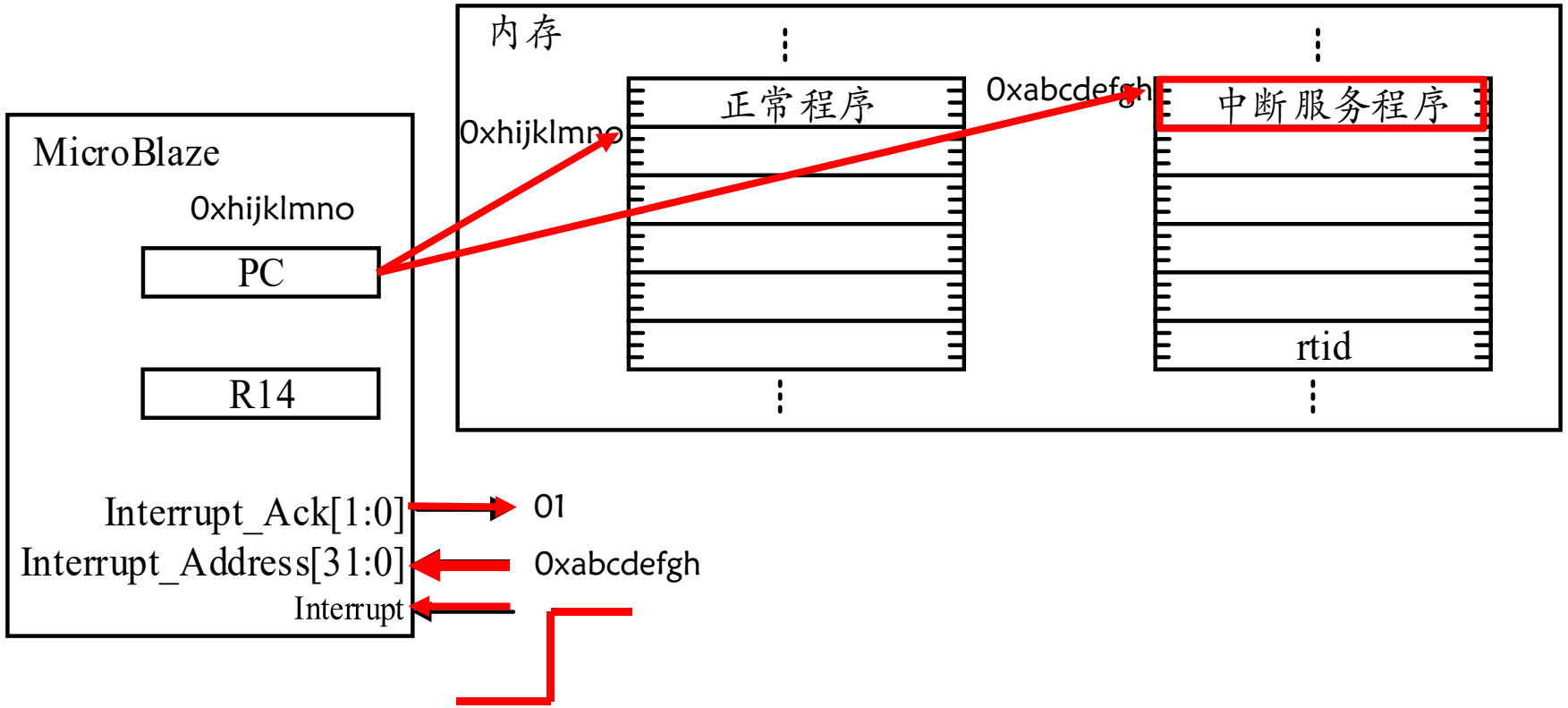
irq上升沿检测中断请求，并锁存interrupt_address[31:0]到寄存器PC

Processor_ack[1:0]	状态
01	跳转到中断服务程序
10	结束中断服务返回
11	再次开放中断

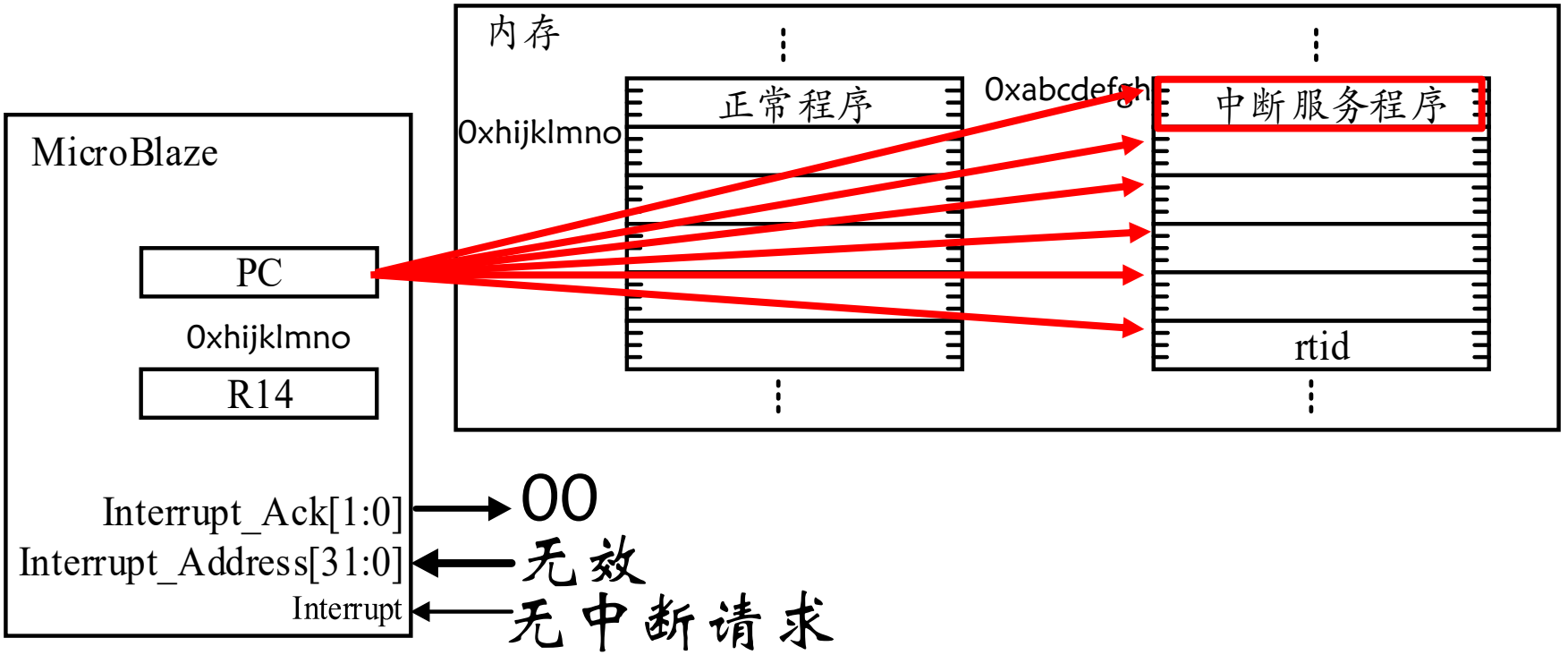
MicroBlaze中断响应过程-初始状态



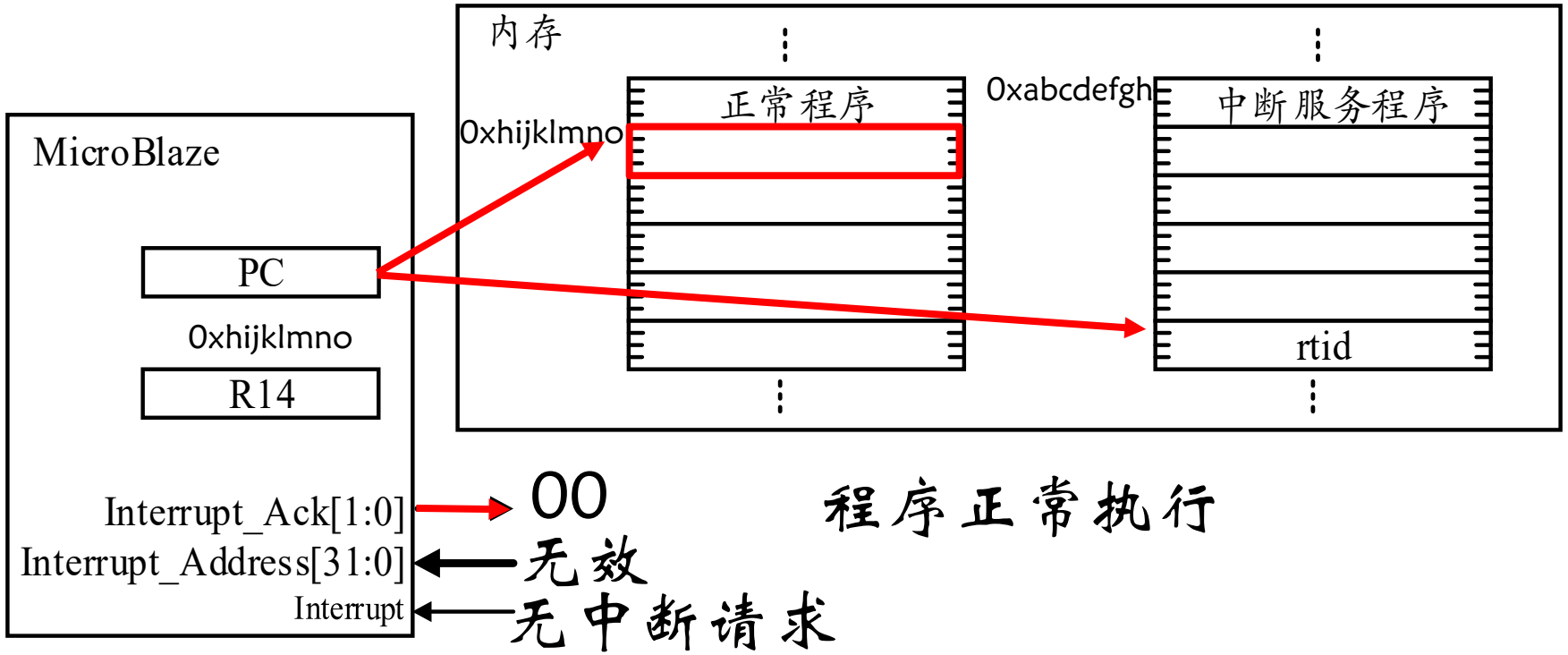
MicroBlaze中断响应过程-中断响应



MicroBlaze中断响应过程-中断服务



MicroBlaze中断响应过程-中断返回



小结

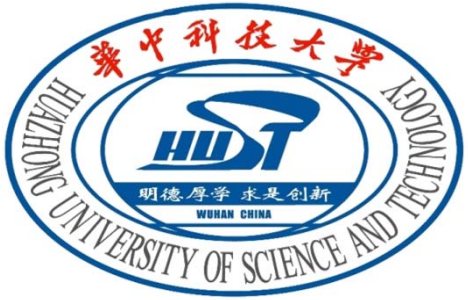
- 微处理器响应中断的条件
  - 现行指令执行结束
  - 微处理器开中断
- MicroBlaze中断响应过程
  - 中断响应周期
  - 中断向量构成
    - 立即数构成的跳转指令
  - 与INTC配合实现快速中断
    - 中断向量由INTC提供

下一讲：中断程序设计基础

微机原理与接口技术

中断方式IO程序设计基础

华中科技大学　左冬红



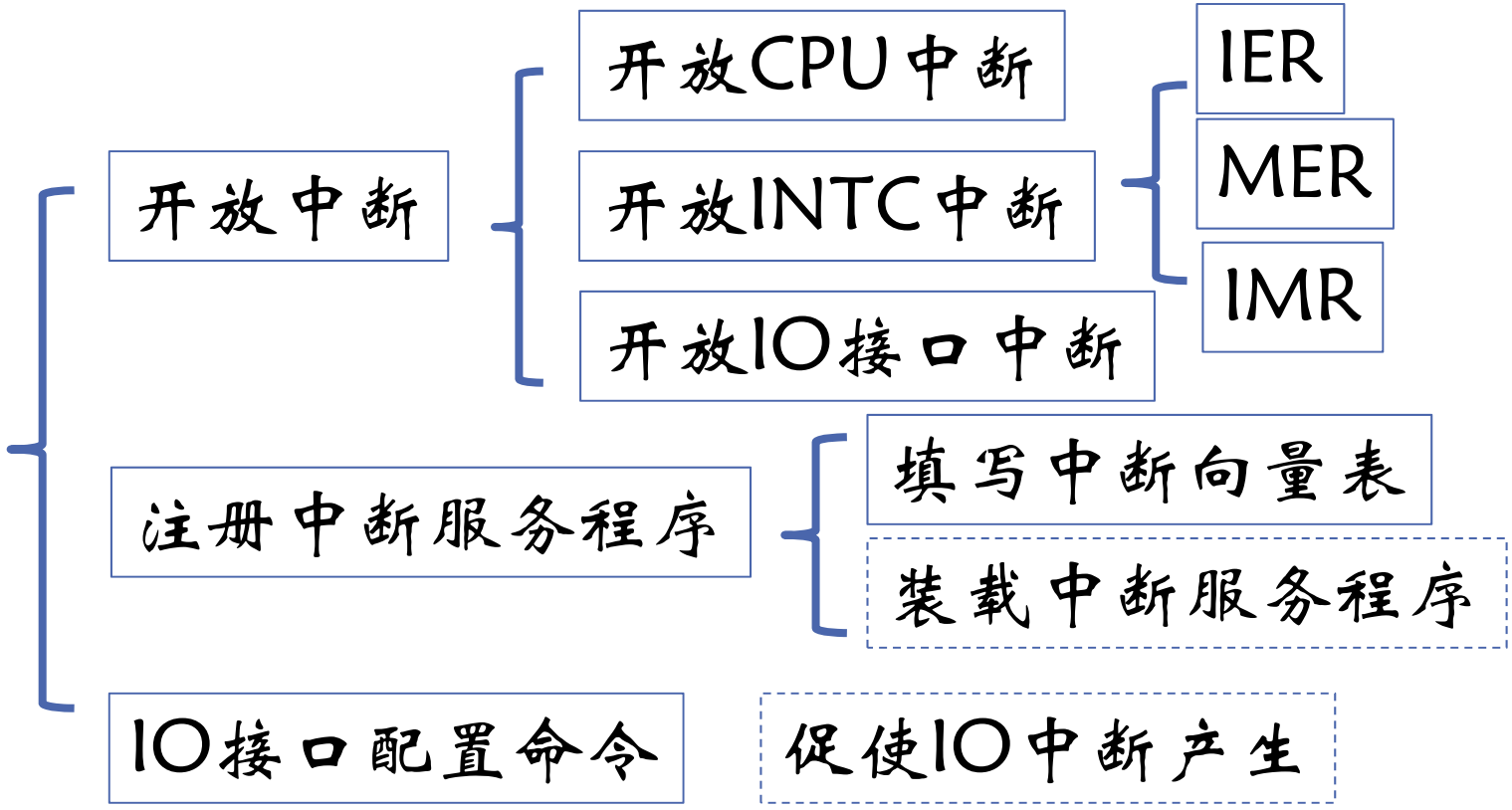
中断方式IO程序基本构成

中断系统初始化程序

中断服务程序

两个程序不存在父子关系

中断初始化程序功能


开放中断
开放CPU中断
开放INTC中断
开放IO接口中断
IER
MER
IMR
注册中断服务程序
填写中断向量表
装载中断服务程序
IO接口配置命令
促使IO中断产生

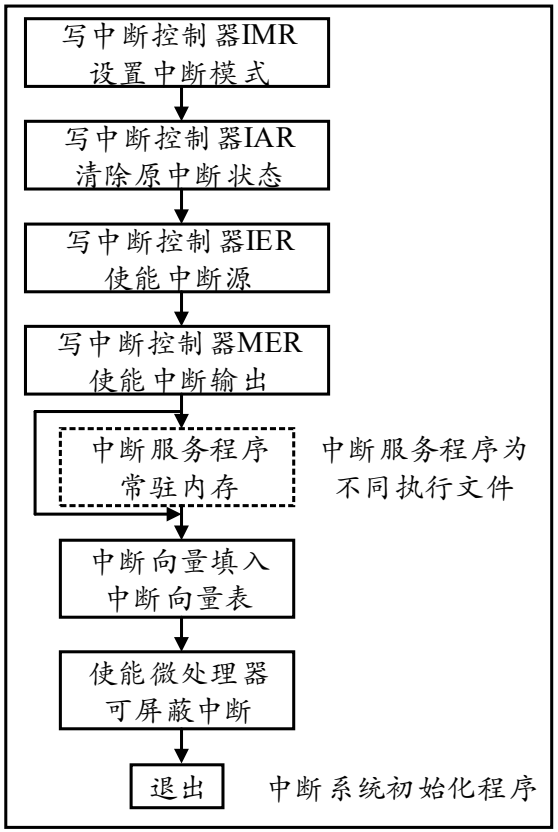
中断服务程序功能

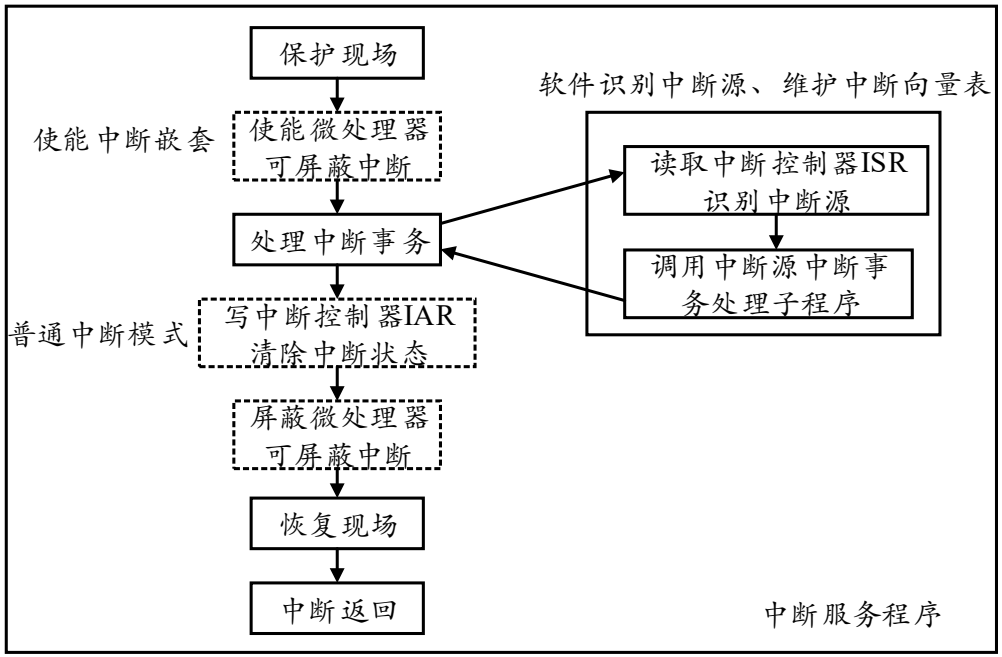
多中断源中断服务程序需识别中断源，调用相应中断事务处理函数

中断事务处理

清除中断状态

中断程序构成


写中断控制器IMR
设置中断模式
写中断控制器IAR
清除原中断状态
写中断控制器IER
使能中断源
写中断控制器MER
使能中断输出
中断服务程序
常驻内存
中断服务程序为
不同执行文件
中断向量填入
中断向量表
使能微处理器
可屏蔽中断
退出
中断系统初始化程序


保护现场
使能中断嵌套
使能微处理器
可屏蔽中断
处理中断事务
普通中断模式
写中断控制器IAR
清除中断状态
屏蔽微处理器
可屏蔽中断
恢复现场
中断返回
软件识别中断源、维护中断向量表
读取中断控制器ISR
识别中断源
调用中断源中断事
务处理子程序
中断服务程序

小结

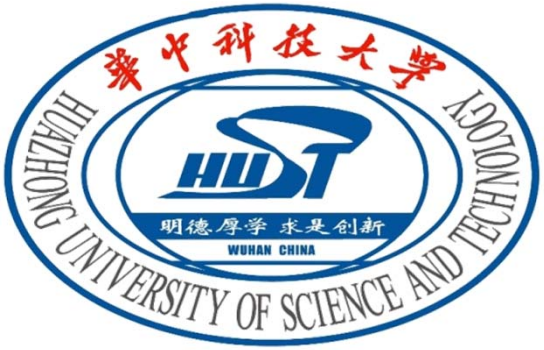
- 中断方式IO程序设计
  - 中断系统初始化程序
  - 中断服务程序
- 两个程序都是在执行完特定任务时，立即退出
  - 无需使用死循环
  - 中断服务程序在每次中断产生时由硬件调用

下一讲：中断API函数

微机原理与接口技术

MicroBlaze MB-GCC C语言中断控制API函数

华中科技大学　左冬红



中断服务程序申明

```
void function_name() __attribute__((interrupt_handler));
// 普通中断模式函数
```

```
void interrupt_handler_name() __attribute__((fast_interrupt));
//快速中断模式函数
```

共同点：

- 都不能带参数
- 编译器编译时函数最后插入中断返回指令

不同点：

- interrupt_handler整个系统只能有一个函数采用此申明，由编译器将该函数地址填写进MB的中断向量表
- fast_interrupt可以多个函数采用此类型，函数地址需由用户程序填入INTC硬件维护的中断向量表

StandAlone BSP API函数

注册中断事务处理函数

```
void microblaze_register_handler(XInterruptHandler Handler, void *DataPtr);
```

函数体

```
MB_InterruptVectorTable[0].Handler = Handler;
MB_InterruptVectorTable[0].CallBackRef = DataPtr;
```

```
MB_InterruptVectorTableEntry MB_InterruptVectorTable[] =
{
{       XIntc_DeviceInterruptHandler,
        (void*) XPAR_MICROBLAZE_0_AXI_INTC_DEVICE_ID}
};
```

MB中断向量表初始值

```
typedef struct
{
  XInterruptHandler Handler;
  void *CallBackRef;
} MB_InterruptVectorTableEntry;
```

MB中断向量表结构体

MicroBlaze开、关中断API函数

开中断

函数原型	汇编代码
void microblaze_enable_interrupts(void)	mfs r12, rmsr ori r12, r12, 2 mts rmsr, r12 rtsd r15, 8 or r0, r0, r0

关中断

函数原型
void microblaze_disable_interrupts(void)

小结

- MicroBlaze C语言中断程序设计
  - 注册中断服务程序
    - 编译器
    - 编译器+IO写INTC
    - StandAlone API （中断事务处理函数，非真正中断服务程序）
  - 开、关中断
    - StandAlone API

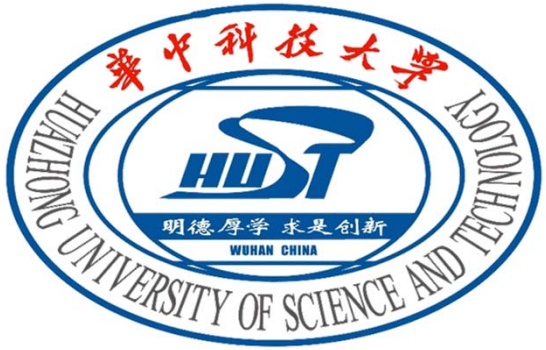
真正中断服务程序不能带参数

下一讲：快速中断应用

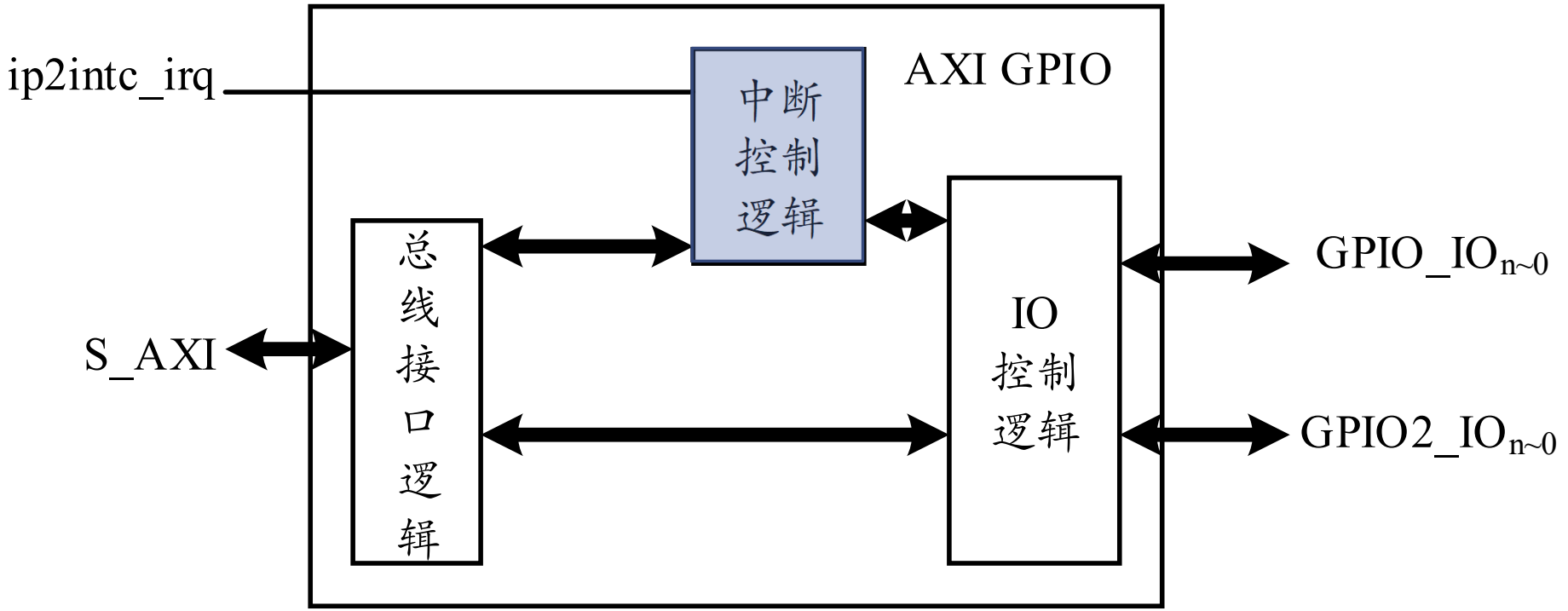
微机原理与接口技术

GPIO中断应用示例

华中科技大学　左冬红



GPIO结构框图



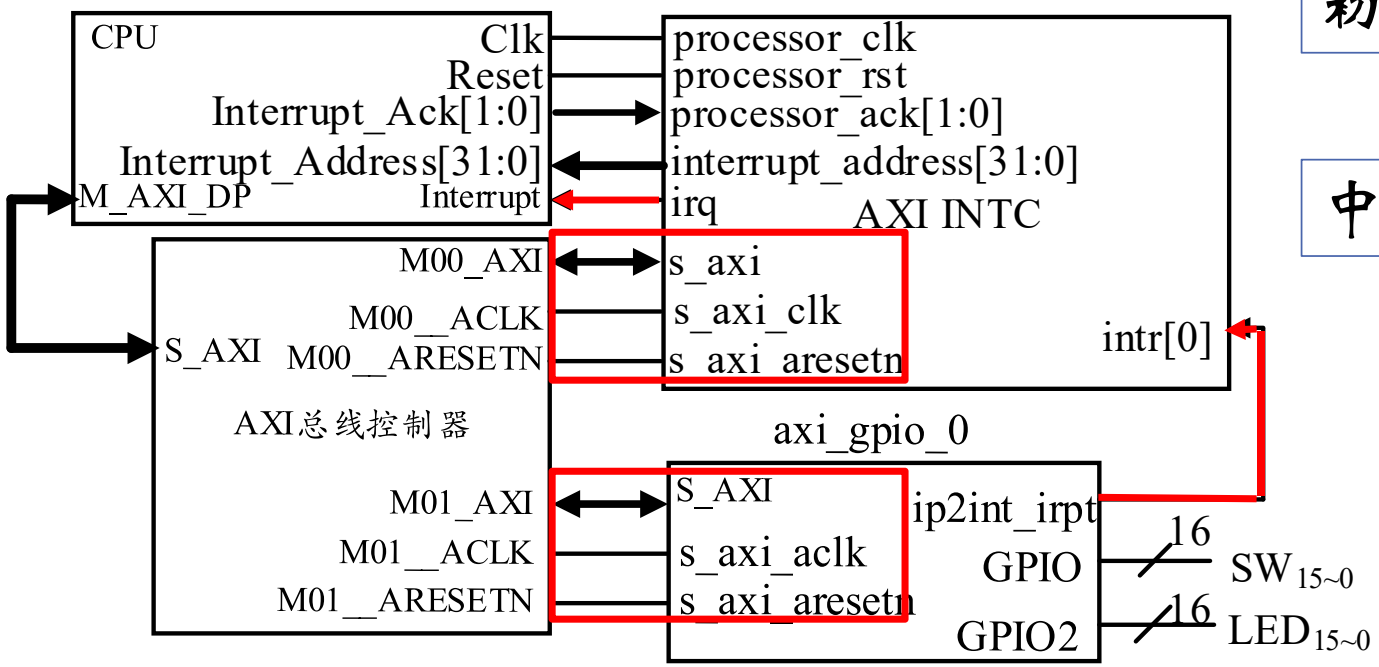
GPIOx_IO引脚输入时出现信号电平跳变，则可以产生中断

两个通道都可以产生中断，最后形成一个中断输出，类似INTC

GPIO中断相关寄存器

名称	偏移地址	含义
GIER	0x11c	D_{31}=1使能中断信号ip2intc_irpt输出
IPIER	0x128	D_0=1使能通道GPIO中断；D_1=1使能通道GPIO2中断，
IPISR	0x120	读：获取通道中断状态，写：清除中断状态 读：D_0=1GPIO产生了中断；D_1=1GPIO2产生了中断 写：D_0=1清除GPIO中断状态；D_1=1清除GPIO2中断状态

应用示例1



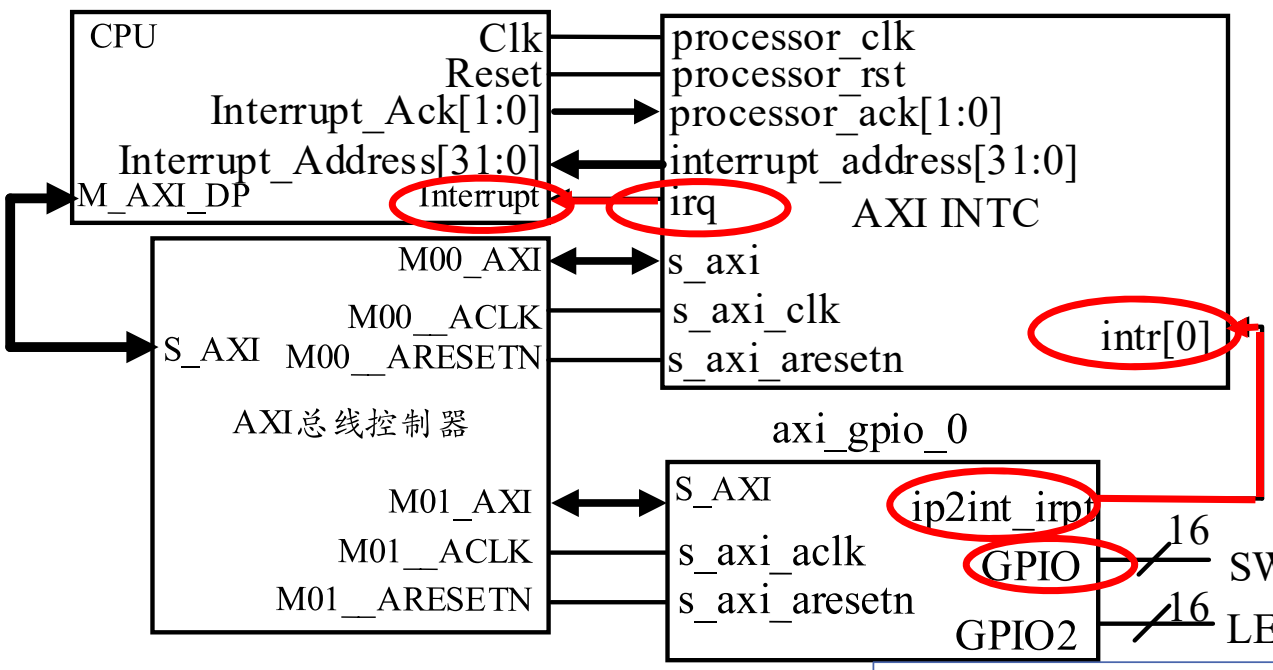
初始化程序

中断服务程序

GPIO 开关拨动产生中断

中断方式读取开关状态并输出到LED

应用示例1



开放中断

- gpio_0 GPIO通道
  - IPIER
  - GIER
- intc intr[0]
  - IER
  - MER
- MicroBlaze
  - MSR

初始化程序

填中断向量表

MB-GCC __attribute__((interrupt_handler))

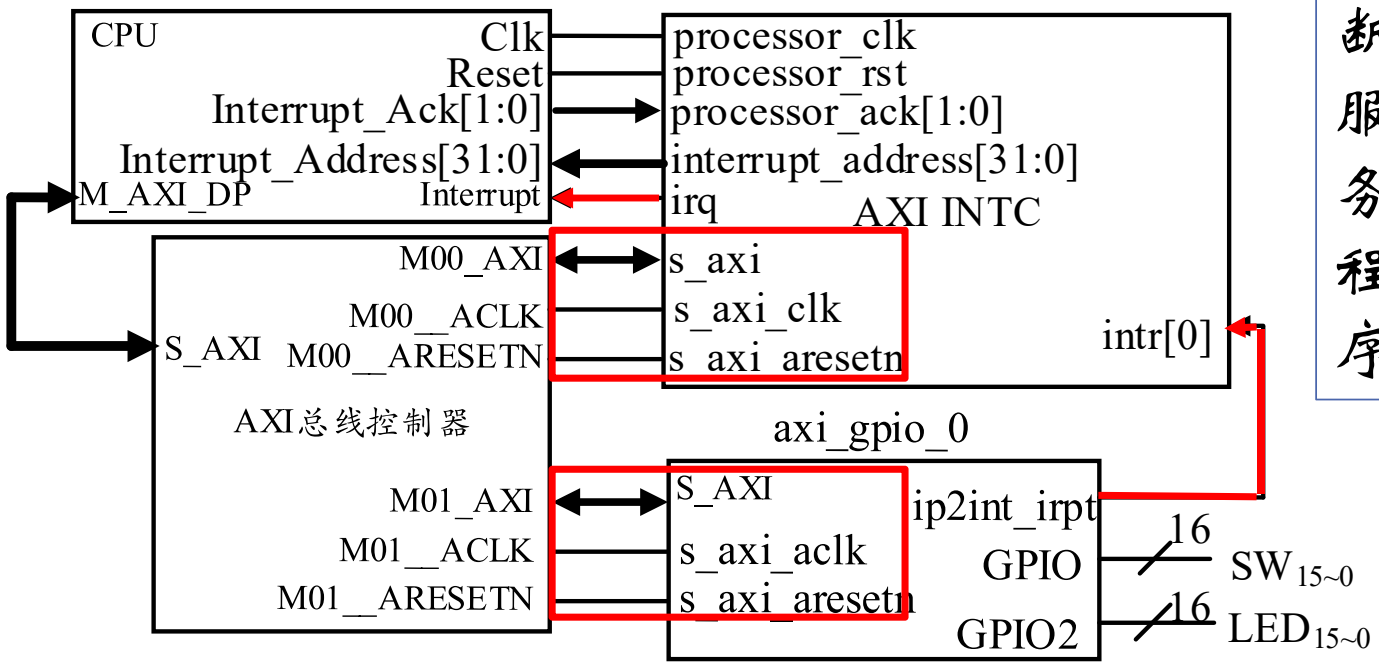
API microblaze_register_handler(,)

IO命令

GPIO工作方式

两者任选其一

应用示例1



中断服务程序

- 中断事务处理
  - 读一次开关
  - 写一次LED
- 清除中断状态
  - INTC intr[0]
    - IAR
  - Gpio_0 GPIO
    - IPISR

中断方式读取开关状态并输出到LED

应用示例1-初始化程序结构

开放中断

- gpio_0 GPIO通道 IPIER GIER
- intc intr[0] IER MER
- MicroBlaze MSR

Xil_Out32(addr,value)

填中断向量表

microblaze_enable_interrupts();

MB-GCC __attribute__(interrupt_handler)

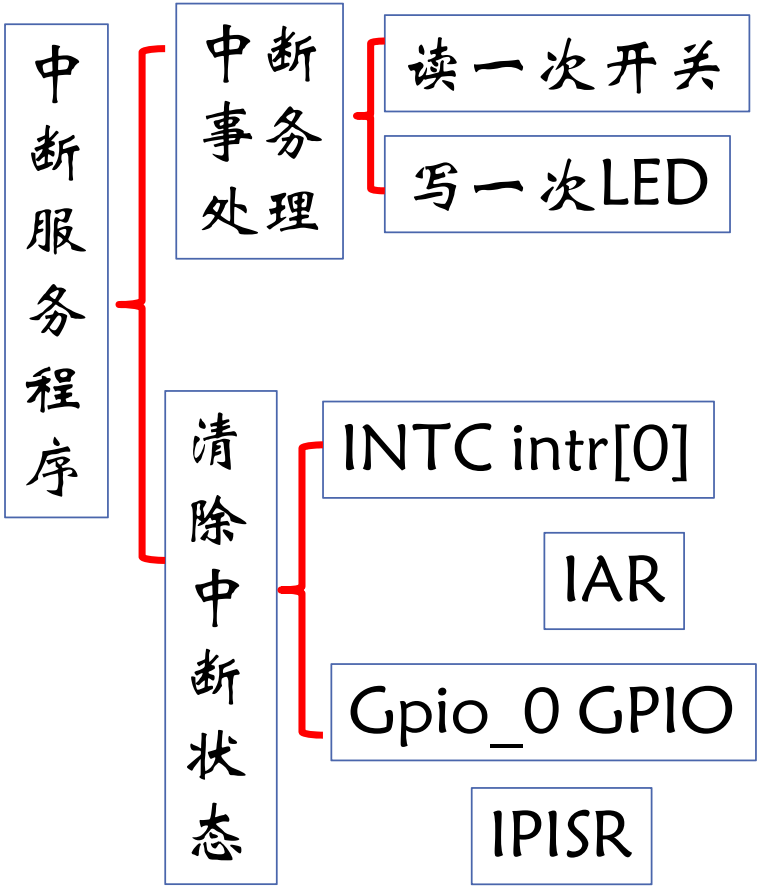
IO命令

GPIO工作方式

应用示例1-初始化程序

```
void main(void)
{
    Xil_Out32(0x40000120,0x1);//写AXI_GPIO_0 IPISR 清除中断
    Xil_Out32(0x40000128,0x1);//写AXI_GPIO_0 IPIER
    Xil_Out32(0x4000011C,0x80000000);//写AXI_GPIO_0 GIER
    Xil_Out32(0x4120000c,0xffffffff);//写INTC IAR清除中断状态
    Xil_Out32(0x41200008,0x1);//写INTC IER,使能intr[0]中断
    Xil_Out32(0x4120001c,0x3);//写INTC MER,使能硬件中断irq输出
    microblaze_enable_interrupts();//使能微处理器中断
    Xil_Out32(0x40000004,0xffff);//写GPIO_TRI通道GPIO输入
    Xil_Out32(0x4000000C,0x0); //写GPIO2_TRI通道GPIO2输出
}
```

应用示例1-中断服务程序



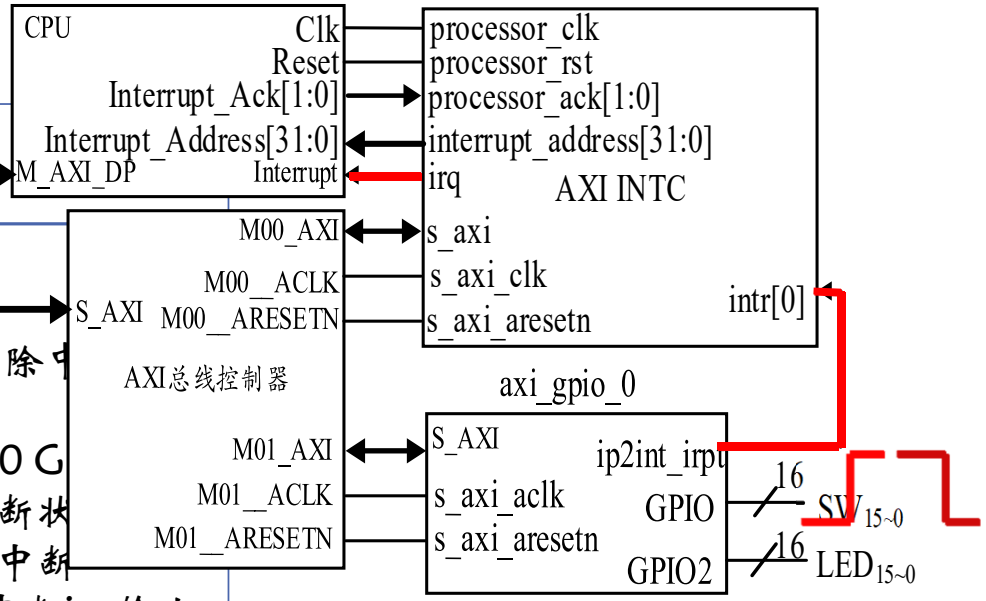

```
void swHandler(void) __attribute__((interrupt_handler));
```

```
void swHandler(void)
{
    int sw;
    sw=Xil_In32(0x40000000);
    Xil_Out32(0x40000008,sw)
    Xil_Out32(0x40000120,0x1);
    Xil_Out32(0x4120000c,0x1);
}
```

应用示例1

```
#include “xil_io.h”
void swHandler(void) __attribute__((interrupt_handler);

void main(void)
{
    Xil_Out32(0x40000120,0x1);//写AXI_GPIO_0 IPISR 清除中
    Xil_Out32(0x40000128,0x1);//写AXI_GPIO_0 IPIER
    Xil_Out32(0x4000011C,0x80000000);//写AXI_GPIO_0 G
    Xil_Out32(0x4120000c,0xffffffff);//写INTC IAR清除中断状
    Xil_Out32(0x41200008,0x1);//写INTC IER,使能intr[0]中断
    Xil_Out32(0x4120001c,0x3);//写INTC MER,使能硬件中断irq输出
    microblaze_enable_interrupts();//使能微处
    Xil_Out32(0x40000004,0xffff);//写GPIO_
    Xil_Out32(0x4000000C,0x0); //写GPIO2
}
```




```
void swHandler(void)
{
    int sw;
    sw=Xil_In32(0x40000000);
    Xil_Out32(0x40000008,sw)
    Xil_Out32(0x40000120,0x1);//写GPIO ISR
    Xil_Out32(0x4120000c,0x1);//写INTC IAR
}
```

小结

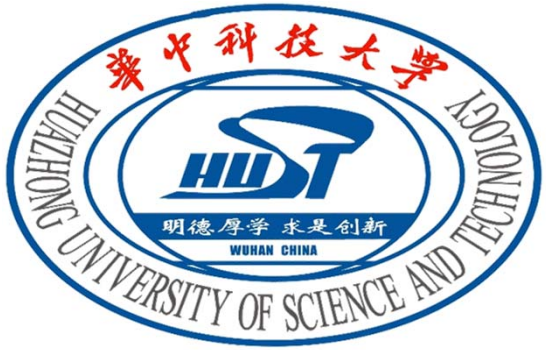
- GPIO产生中断的原因
  - GPIO输入引脚电平跳变
- 中断程序设计
  - 初始化程序
    - GPIO中断开放
    - GPIO工作方式设置
    - INTC开中断
    - MicroBlaze开中断
    - 填写中断向量表
  - 中断服务程序
    - 中断事务处理
    - 清除中断
      - GPIO\INTC都需清除

下一讲：定时器中断

微机原理与接口技术

定时器中断应用示例

华中科技大学　左冬红



硬件定时器

数字钟　由计数器实现

秒钟进位信号的时间间隔60秒
分钟进位信号的时间间隔60分
} 固定时长定时器
嵌入式计算机系统的看门狗

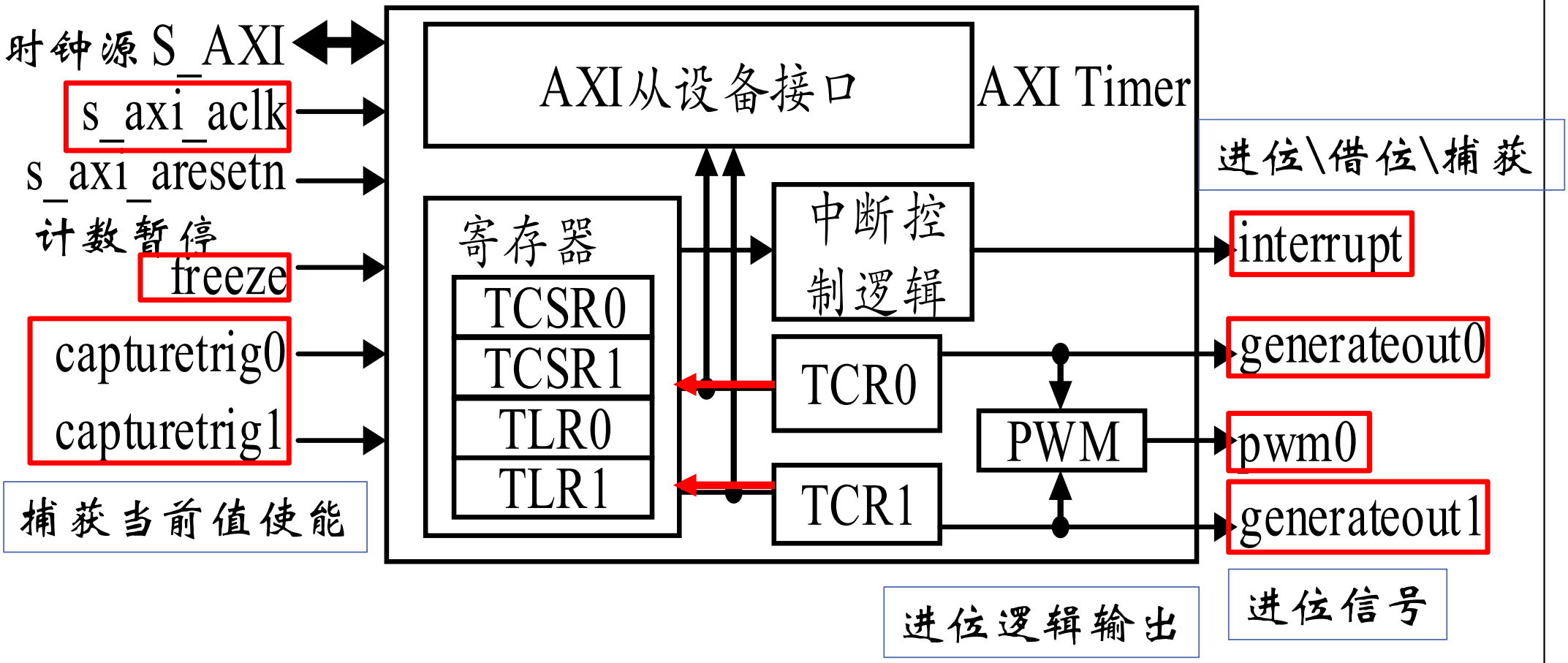
预置一次数值，之后多位计数器不停加计数　计算机系统时钟

若计数溢出之后自动装载新的预置值　定时间隔可调整

若将计数器的进位信号输出，则为周期性脉冲信号

若计数器的计数输出可锁存，则可测量外部信号周期（边沿触发）

AXI Timer定时器



Timer工作方式

- 定时
  - 单次定时：装载一次预置值；计数一轮；产生一个中断
  - 周期性定时：自动装载预置值；重复计数；周期性中断
- 定时模式：Generate
  - 周期脉冲
- 测频：无需预置；重复计数；中断时获取计数当前值
  - 两次中断计数值只差即为被测信号的周期
- 计时模式：Capture
- PWM波：周期性定时；自动装载预置值；重复计数；周期脉冲
- PWM模式：同时使用两个计数器

Timer定时间隔

加计数：T=(TCRmax-TLR+2)*AXI_CLK_PERIOD
减计数：T=(TLR+2)* AXI_CLK_PERIOD

Timer寄存器存储映像

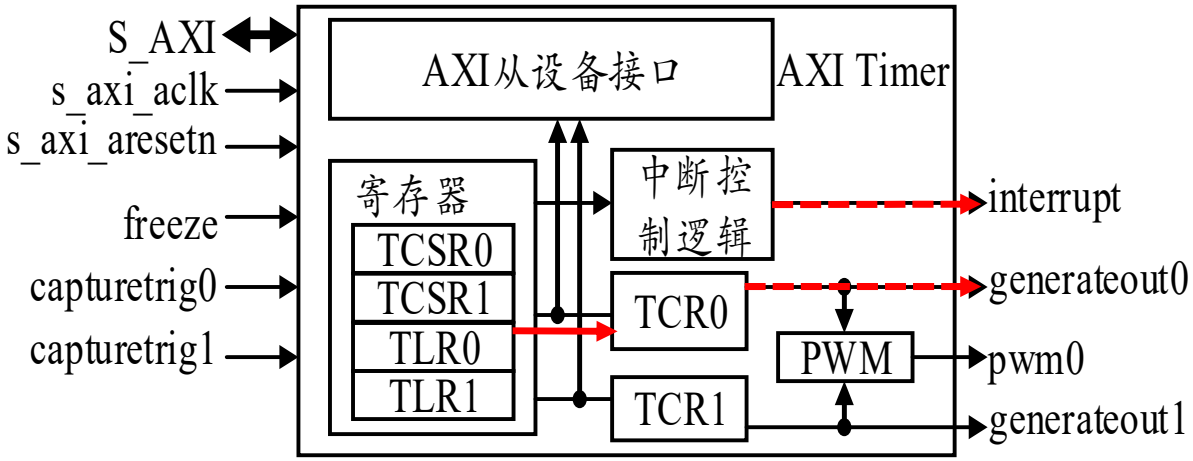
名称	偏移地址	功能描述	名称	偏移地址	功能描述
TCSR0	0x00	T0控制状态寄存器	TCSR1	0x10	T1控制状态寄存器
TLR0	0x04	T0装载寄存器	TLR1	0x14	T1装载寄存器
TCR0	0x08	T0计数寄存器	TCR1	0x18	T1计数寄存器

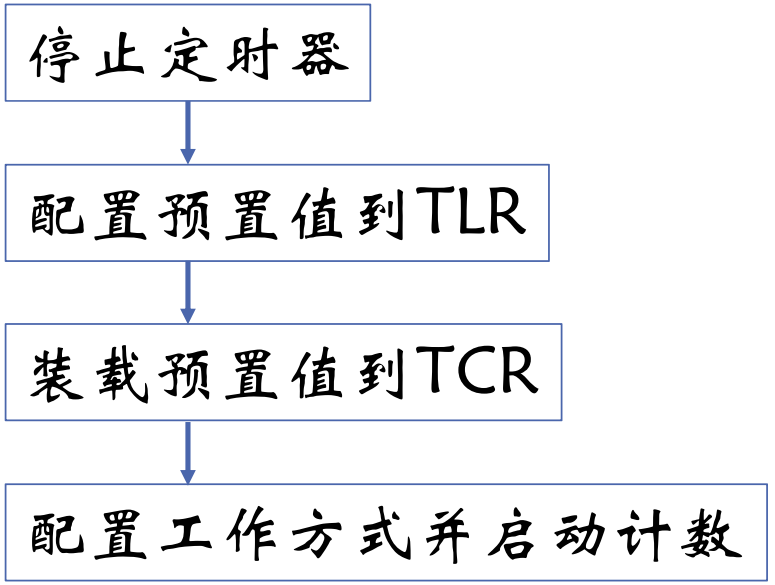
32位预置值或计数值

Timer TCSR寄存器

位	名称	含义
0	MDTx	工作模式：0-定时，1-计时
1	UDTx	计数方式：0-加计数，1-减计数
2	GENTx	generateout输出使能：0-禁止，1-使能
3	CAPTx	capturetrig触发使能：0-禁止，1-使能
4	ARHTx	自动重复装载：0-禁止，1-使能
5	LOADx	装载：0-不装载，1-装载
6	ENITx	中断使能：0-禁止，1-使能
7	ENTx	计数器运行使能：0-停止，1-启动
8	TxINT	计数器中断状态：读：0-无中断，1-有中断;写：0-无意义，1-清除中断状态
9	PWMAx	脉宽调制使能： 0-无效，1且MDTx=0、GENTx=1-脉宽输出
10	ENALL	所有计数器使能：0-清除ENALL位，对ENTx无影响；1-使能所有计数器
11	CASC	级联模式1-级联，0-独立 仅TCSR0此位有意义

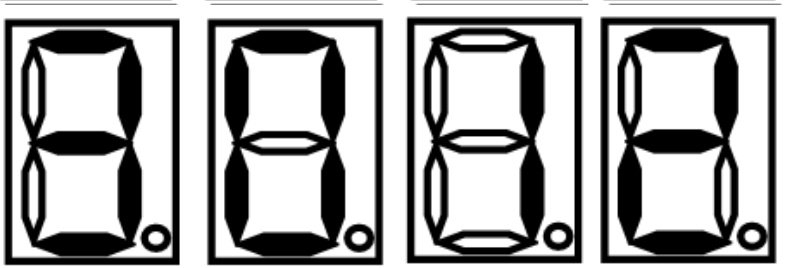
Timer工作原理-定时模式





Timer定时中断示例

Microblaze微处理器计算机系统采用AXI总线控制4位七段数码管动态显示接口滚屏显示数字0~3序列，要求采用AXI Timer定时中断实现，试设计接口电路和控制程序。



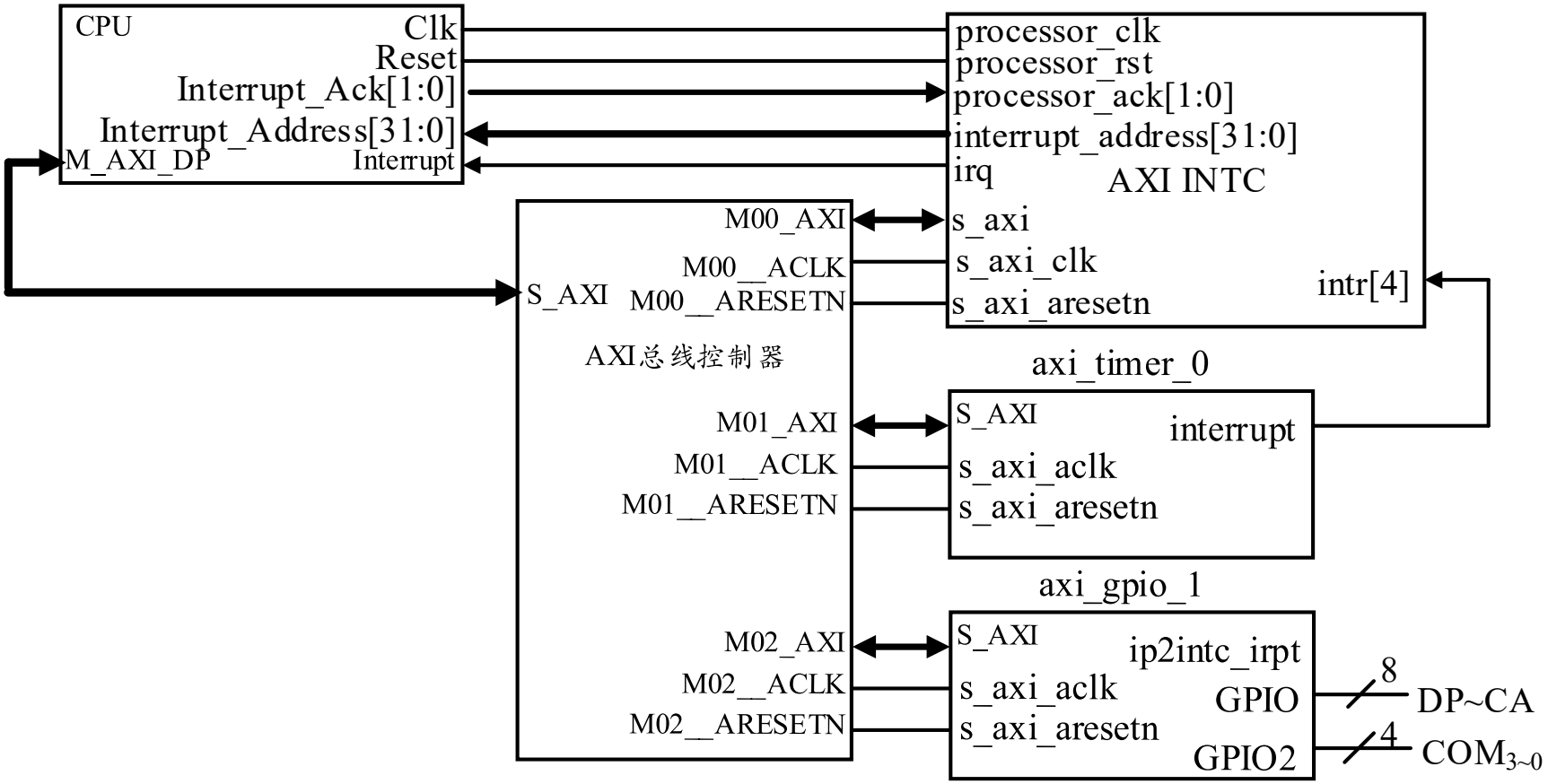
4位七段数码管滚屏显示数字0~3序列，即4位七段数码管首次显示数字串0123，之后每隔一段时间再依次更换显示数字串1230；2301；3012；并循环

两处定时：

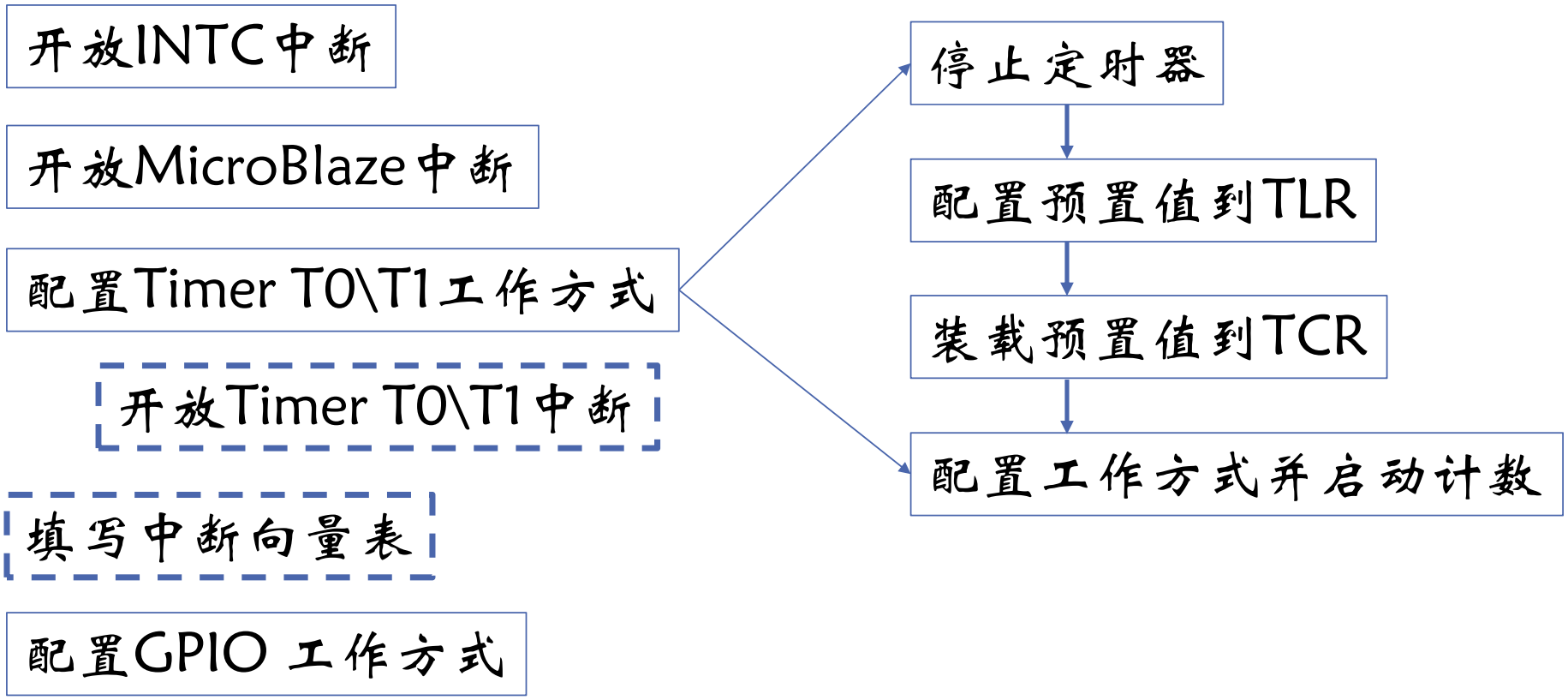
1）4位七段数码管动态显示扫描间隔延时；

2）显示数字串更新延时。

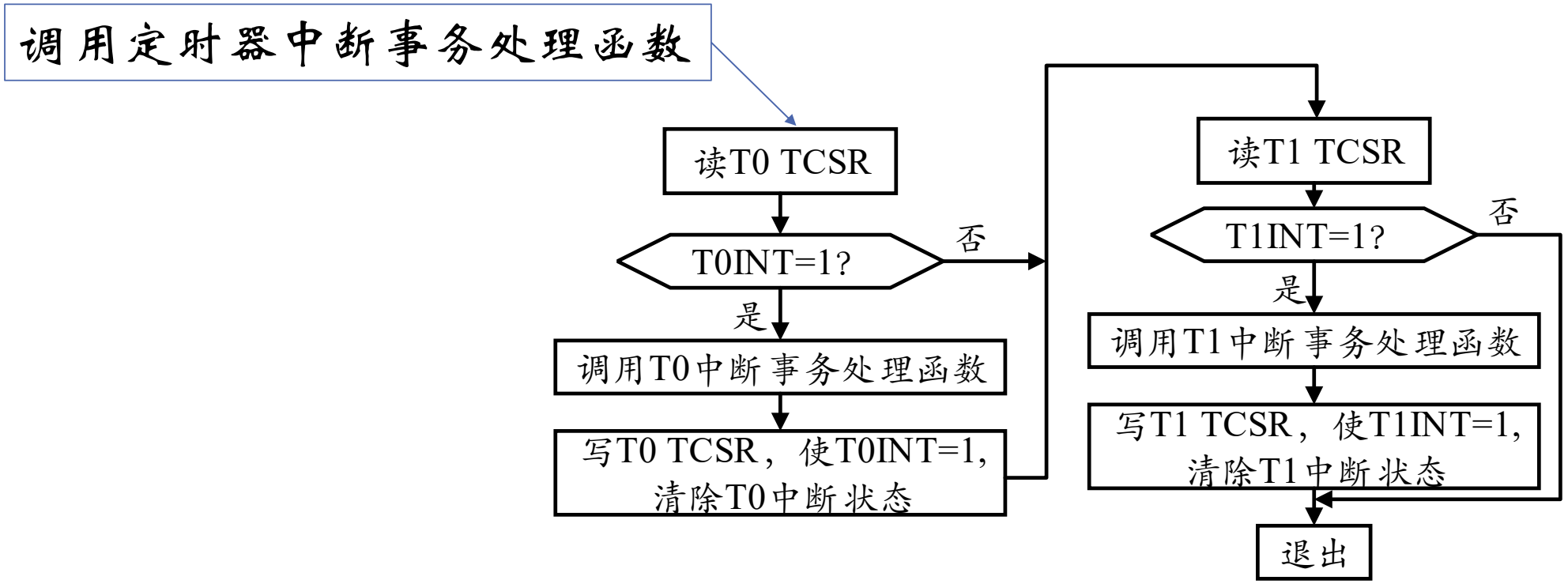
Timer定时中断示例接口电路-快速中断



中断初始化程序

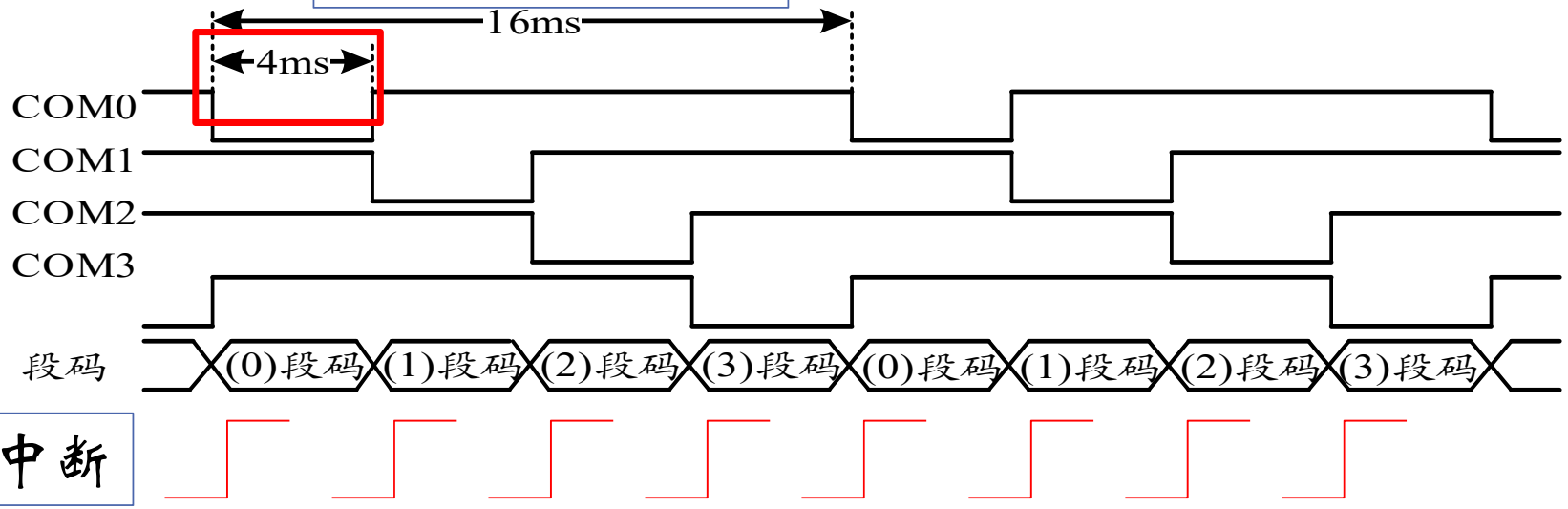

开放INTC中断
开放MicroBlaze中断
配置Timer T0\T1工作方式
开放Timer T0\T1中断
填写中断向量表
配置GPIO 工作方式
停止定时器
配置预置值到TLR
装载预置值到TCR
配置工作方式并启动计数

中断服务程序


调用定时器中断事务处理函数
读T0 TCSR
T0INT=1?
否
是
调用T0中断事务处理函数
写T0 TCSR，使T0INT=1，
清除T0中断状态
读T1 TCSR
T1INT=1?
否
是
调用T1中断事务处理函数
写T1 TCSR，使T1INT=1，
清除T1中断状态
退出

4位七段数码管动态显示扫描间隔延时

硬件定时器定时



定时中断

中断事务：

从显示缓冲区读取一位数字对应的段码、位码并输出

记录段码、位码对应值，中断次数%4

数字串移位更新延时间隔

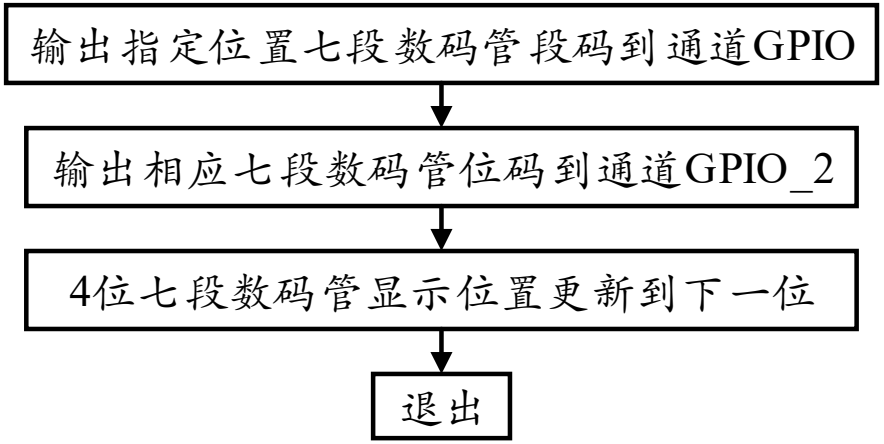
间隔时间应为数秒或更长

中断事务：

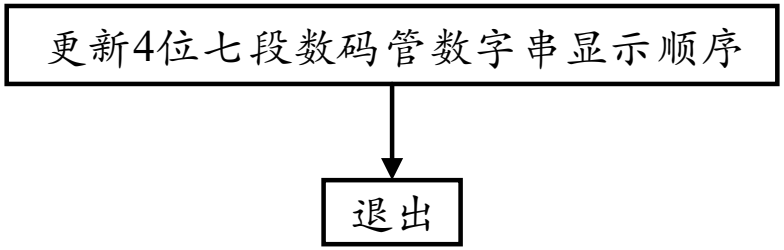
更新数码管显示缓冲器

各位数字段码在显示缓冲区中存储的位置向左循环移动一位

定时器中断事务处理



T0中断事务处理函数流程



T1中断事务处理函数流程

中断初始化程序代码——快速中断

```
void setup_interrupt_system(void)
{
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IAR_OFFSET,XPAR_AXI_TIMER_0_INTERRUPT_MASK);
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IER_OFFSET,XPAR_AXI_TIMER_0_INTERRUPT_MASK);
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IMR_OFFSET,XPAR_AXI_TIMER_0_INTERRUPT_MASK);
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_MER_OFFSET,
   XIN_INT_MASTER_ENABLE_MASK|XIN_INT_HARDWARE_ENABLE_MASK);
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IVAR_OFFSET+
          4*XPAR_INTC_0_TMRCTR_0_VEC_ID,(int)Timerhandler);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
                Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET )|XTC_CSR_INT_OCCURED_MASK);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
                  Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET )| XTC_CSR_ENABLE_INT_MASK);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
          Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET )
          | XTC_CSR_INT_OCCURED_MASK);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
          Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET )
          | XTC_CSR_ENABLE_INT_MASK);
    microblaze_enable_interrupts();
}
```

IO控制初始化程序

```
int main(void)
{
    int tcsr0,tcsr1;
    setup_interrupt_system();
    Xil_Out32(XPAR_GPIO_1_BASEADDR+XGPIO_TRI_OFFSET,0x0);
    Xil_Out32(XPAR_GPIO_1_BASEADDR+XGPIO_TRI2_OFFSET,0x0);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TLR_OFFSET,T0_RESET_VALUE);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TLR_OFFSET,T1_RESET_VALUE);
    tcsr0=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,tcsr0|XTC_CSR_LOAD_MASK);
    tcsr1=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,tcsr1|XTC_CSR_LOAD_MASK);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
  tcsr0|XTC_CSR_ENABLE_TMR_MASK|XTC_CSR_DOWN_COUNT_MASK
  |XTC_CSR_AUTO_RELOAD_MASK|XTC_CSR_EXT_GENERATE_MASK);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
tcsr1|XTC_CSR_ENABLE_TMR_MASK|XTC_CSR_DOWN_COUNT_MASK|XTC_CSR_AUTO_RELOAD_MASK
            |XTC_CSR_EXT_GENERATE_MASK);
    return 0;
}
```

全局变量及函数申明

```
#define T0_RESET_VALUE 100000-2 //0.001s
#define T1_RESET_VALUE 100000000-2 //1s
void T0handler(void);
void T1handler(void);
void Timerhandler(void) __attribute__ ((fast_interrupt));
void setup_interrupt_system(void);
char segcode[4]={0xc0,0xf9,0xa4,0xb0};
char poscode[4]={0xf7,0xfb,0xfd,0xfe};
int loop=0,pos=0;
```

宏定义头文件

```
#include "xintc_l.h"
#include "xtmrctr_l.h"
#include "xgpio_l.h"
#include "xparameters.h"
#include "xio.h"
#include "xil_exception.h"
```

定时器中断服务程序

```
void Timerhandler(void)
{
    int tcsr;
    tcsr=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET);
    if((tcsr&XTC_CSR_INT_OCCURED_MASK)==XTC_CSR_INT_OCCURED_MASK)
    {
        T0handler();
        Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
                tcsr|XTC_CSR_INT_OCCURED_MASK);
    }
    tcsr=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET
                +XTC_TCSR_OFFSET);
    if((tcsr&XTC_CSR_INT_OCCURED_MASK)==XTC_CSR_INT_OCCURED_MASK)
    {
        T1handler();
        Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET
            +XTC_TCSR_OFFSET,tcsr|XTC_CSR_INT_OCCURED_MASK);
    }
}
```

中断事务处理函数

```
void T0handler(void)
{
    Xil_Out32(XPAR_GPIO_1_BASEADDR+XGPIO_DATA_OFFSET,segcode[(loop+pos)%4]);
    Xil_Out32(XPAR_GPIO_1_BASEADDR+XGPIO_DATA2_OFFSET,poscode[pos]);
    pos++;
    if(pos==4)
        pos=0;
}
```

```
void T1handler(void)
{
    loop++;
    if(loop==4)
        loop=0;
}
```

显示缓冲区段码数字
0,0xc0
1,0xf9
2,0xa4
3,0xb0

	T0 pos			
	0	1	2	3
T1 loop	数码管显示的数字串			
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

普通中断方式

```
#define T0_RESET_VALUE 100000-2 //0.001s
#define T1_RESET_VALUE 100000000-2 //1s
void T0handler(void);
void T1handler(void);
void Timerhandler(void) __attribute__ ((interrupt_handler));
void setup_interrupt_system(void);
char segcode[4]={0xc0,0xf9,0xa4,0xb0};
char poscode[4]={0xf7,0xfb,0xfd,0xfe};
 int loop=0,pos=0;
```

定时器中断服务程序

```
void Timerhandler(void)
{
    int tcsr;
    tcsr=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET);
    if((tcsr&XTC_CSR_INT_OCCURED_MASK)==XTC_CSR_INT_OCCURED_MASK)
    {
        T0handler();
        Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
                tcsr|XTC_CSR_INT_OCCURED_MASK);
    }
    tcsr=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET
                +XTC_TCSR_OFFSET);
    if((tcsr&XTC_CSR_INT_OCCURED_MASK)==XTC_CSR_INT_OCCURED_MASK)
    {
        T1handler();
        Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET
            +XTC_TCSR_OFFSET,tcsr|XTC_CSR_INT_OCCURED_MASK);
    }
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IAR_OFFSET,
    XPAR_AXI_TIMER_0_INTERRUPT_MASK);//写INTC IAR
}
```

中断初始化程序代码——普通中断

```
void setup_interrupt_system(void)
{
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IAR_OFFSET,XPAR_AXI_TIMER_0_INTERRUPT_MASK);
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IER_OFFSET,XPAR_AXI_TIMER_0_INTERRUPT_MASK);
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IMR_OFFSET,XPAR_AXI_TIMER_0_INTERRUPT_MASK);
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_MER_OFFSET,
   XIN_INT_MASTER_ENABLE_MASK|XIN_INT_HARDWARE_ENABLE_MASK);
    Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IVAR_OFFSET+
          4*XPAR_INTC_0_TMRCTR_0_VEC_ID,(int)Timerhandler);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
                Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET )|XTC_CSR_INT_OCCURED_MASK);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
                  Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET )| XTC_CSR_ENABLE_INT_MASK);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
          Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET )
          | XTC_CSR_INT_OCCURED_MASK);
    Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
          Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET )
          | XTC_CSR_ENABLE_INT_MASK);
    microblaze_enable_interrupts();
}
```

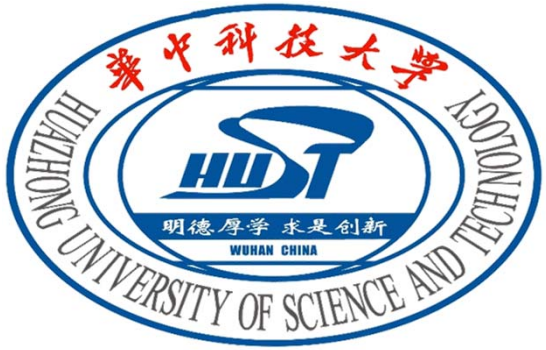
小结

- 定时器工作在循环定时方式的初始化控制流程
  - 停止定时器
  - 写计数初值
  - 装载计数初值
  - 启动定时器并配置工作模式、启用中断
- 快速中断与普通中断方式区别
  - 快速中断INTC无需查询(多中断源)、无需写IAR
  - 快速中断需填写INTC中断向量表

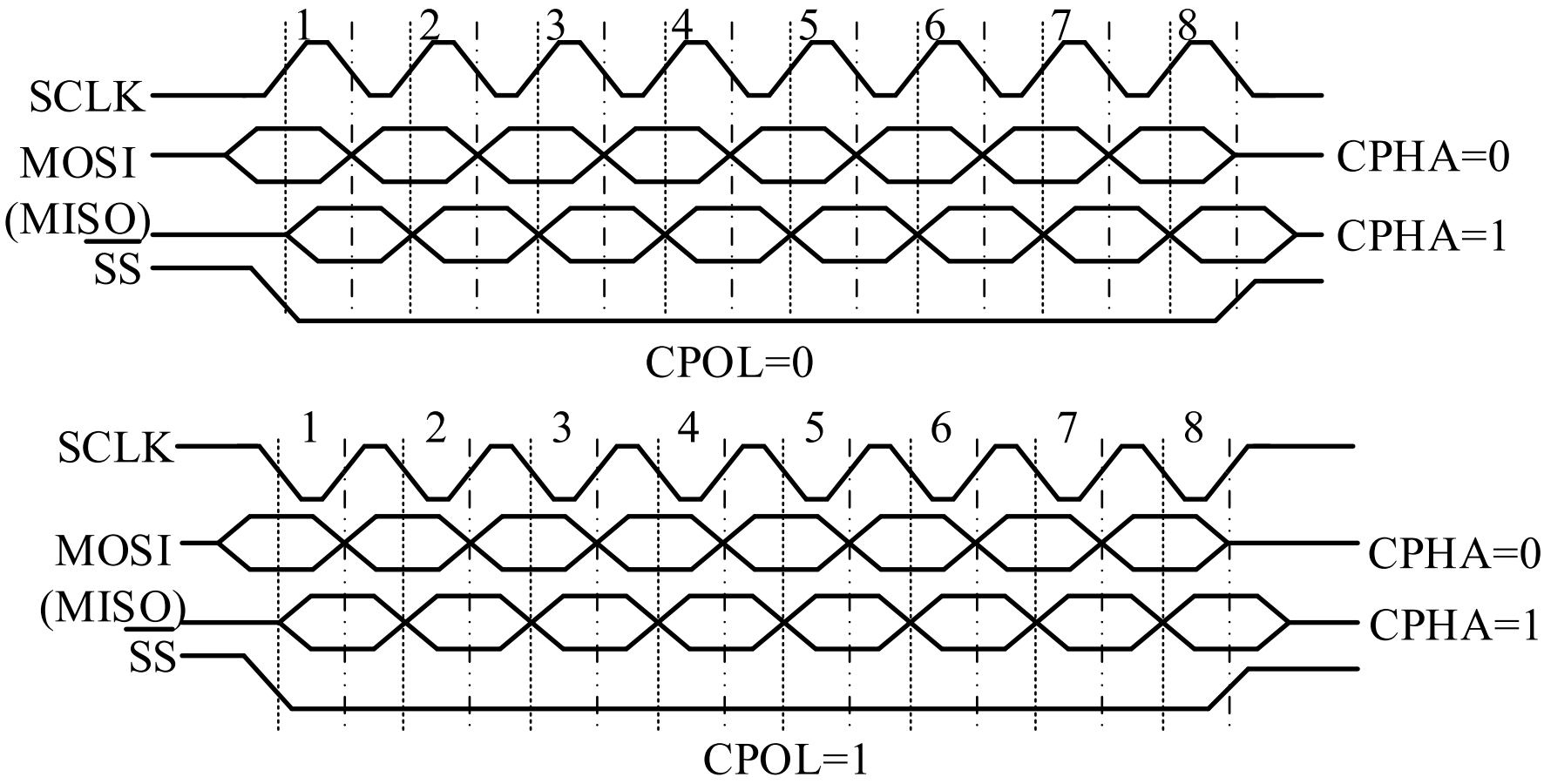
微机原理与接口技术

串行SPI接口中断程序设计

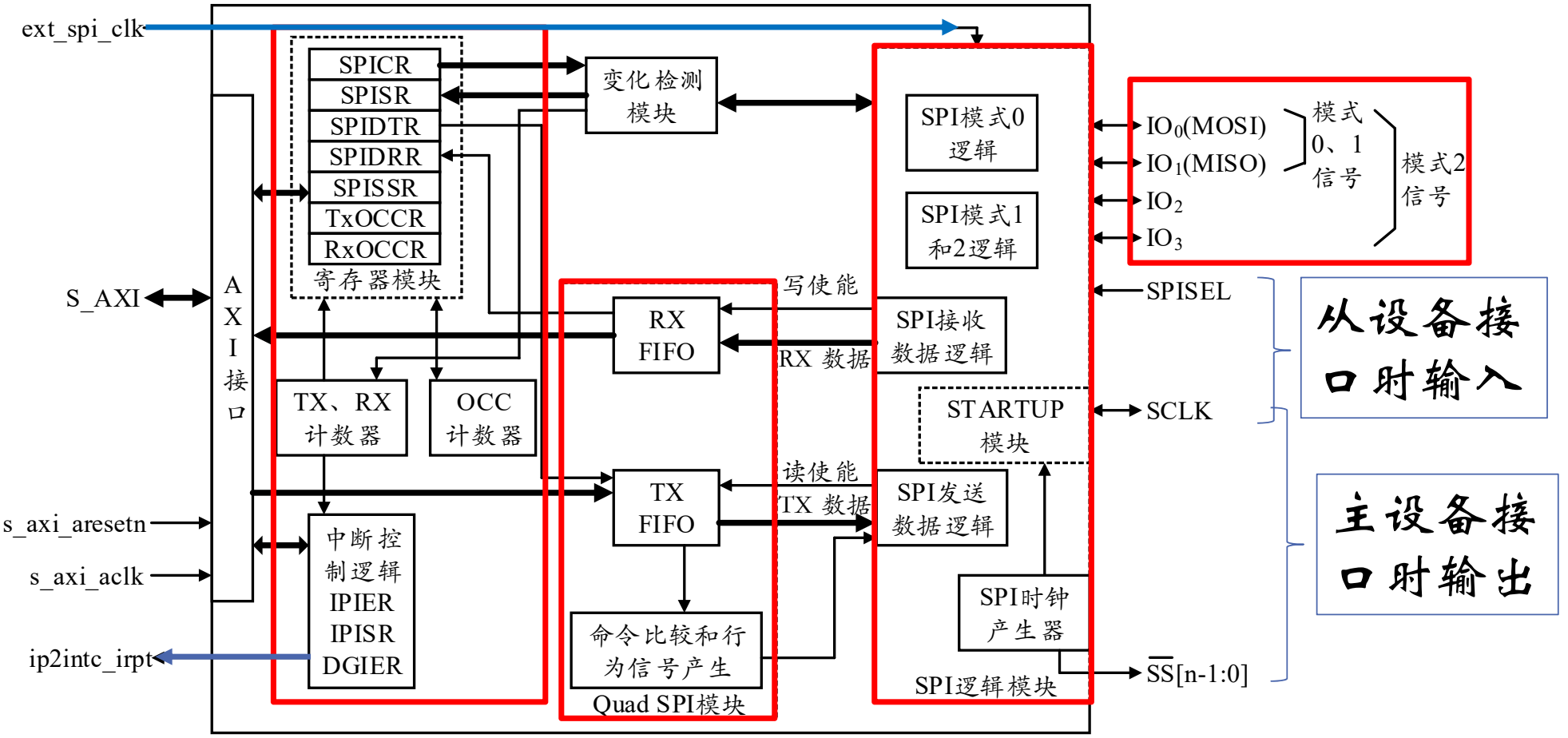
华中科技大学　左冬红



回顾SPI总线通信流程



AXI SPI IP核简介



SPI编程寄存器存储映像

寄存器	偏移地址	含义
SRR	0x40	软件复位寄存器，写0x0000000A复位接口
SPICR	0x60	控制寄存器，设定AXI Quad SPI IP核工作方式
SPISR	0x64	状态寄存器，指示AXI Quad SPI IP核工作状态
SPIDTR	0x68	发送数据寄存器或发送数据FIFO
SPIDRR	0x6C	接收数据寄存器或接收数据FIFO
SPISSR	0x70	从设备选择寄存器
TxOCCR	0x74	发送FIFO占用长度指示，值+1表示发送FIFO有效数据的长度
RxOCCR	0x78	接收FIFO占用长度指示，值+1表示接收FIFO有效数据的长度
DGIER	0x1C	设备总中断使能寄存器，仅最高位有效，D_{31}=1使能接口中断请求输出
IPISR	0x20	中断状态寄存器
IPIER	0x28	中断使能寄存器

SPICR寄存器含义

位	含义	1	0
0	回环	SPI发送端(MOSI)与接收端(MISO)内部连通形成环路	SPI发送端(MOSI)与接收端(MISO)独立工作
1	启用接口	启用SPI接口(与SPI总线连接)	禁用SPI接口(与SPI总线断开、高阻态)
2	主设备	SPI主设备接口	SPI从设备接口
3	CPOL	空闲时时钟为高电平	空闲时时钟为低电平
4	CPHA	数据信号在时钟第二个边沿(相位180°)稳定有效	数据信号在时钟第一个边沿(相位0°)稳定有效
5	Tx 复位	复位发送FIFO指针	无意义
6	Rx 复位	复位接收FIFO指针	无意义
7	手动控制从设备选择	程序控制从设备选择SS输出，即写SPISSR立即反映到$\overline{66}$[n-1:0]	协议逻辑控制SPISSR输出到$\overline{66}$[n-1:0]
8	禁止主设备事务	禁止主设备事务；若为从设备则无意义	使能主设备事务
9	低位优先	低位优先传送	高位优先传送
10~31	保留，无意义		

SPISR寄存器含义

位	含义	0	1
0	接收寄存器/FIFO空否	非空	空
1	接收寄存器/FIFO满否	未满	满
2	发送寄存器/FIFO空否	非空	空
3	发送寄存器/FIFO满否	未满	满
4	模式错误否	无错误	错误。主设备接口时，$\overline{66}$输入低电平有效信号则置位
5	从设备选中否	选中	未选中
6	CPOL_CPHA错误否	无	错误。DSPI或QSPI模式时，CPOL、CPHA仅支持配置为00或11，若配置为10或01则置位
7	从设备模式错误否	无	错误。DSPI或QSPI模式时，仅支持主设备模式，若配置为从设备模式则置位
8	高位优先错误否	无	错误。DSPI或QSPI模式时，仅支持高位优先传送，若配置为低位优先传送则置位
9	回环错误否	无	错误。DSPI或QSPI模式时，不支持回环，若配置为回环则置位
10	命令错误否	无	错误。DSPI或QSPI模式时，若复位后发送FIFO中的第一个数据不是支持的命令则置位
11~31	保留，无意义		

SPIDTR\DRR\FIFO有效数据位

SPIDTR、SPIDRR以及发送、接收FIFO有效数据位长度与SPI一帧数据位数n一致

n在标准SPI模式时可为8、16、32，其余模式时仅可为8

IPISR\IER寄存器含义

位	含义
31:14	保留，无意义
13	1：命令错误，DSPI或QSPI模式时，复位后发送FIFO中的第一个数据不是支持的命令；0：标准SPI或无错误
12	1：回环错误，DSPI或QSPI模式时，配置为回环；0：标准SPI或无回环错误
11	1：DSPI或QSPI模式时，配置为低位优先传送；0：标准SPI或无低位优先设置错误
10	1：DSPI或QSPI模式时，配置为从设备模式；0：标准SPI或为主设备模式
9	1：DSPI或QSPI模式时，CPOL、CPHA配置为10或01 0：标准SPI或CPOL、CPHA为00或11
8	1：数据接收FIFO非空，仅采用FIFO且从设备模式时有效；0：空或其他模式
7	1：从设备被选中，仅工作在从设备模式时有效；0：未被选中或其他模式
6	1：Tx FIFO半空，表示FIFO深度为16时，TxOCCR寄存器的值从8变为7或FIFO深度为256时，TxOCCR寄存器的值从128变为127；0：其他情况
5	1：数据接收寄存器/FIFO过载（满接收）；0：未溢出
4	1：数据接收寄存器/FIFO满；0：未满
3	1：数据发送寄存器/FIFO欠载（空发送）；0：未溢出
2	1：数据发送寄存器/FIFO空；0：非空
1	1：从设备模式错误，配置为从设备但未启动SPI接口时，$\overline{66}$引脚输入低电平； 0：无错
0	1：模式错误，配置为主设备但$\overline{66}$引脚输入低电平；0：未错

SPI主设备接口中断方式控制程序流程

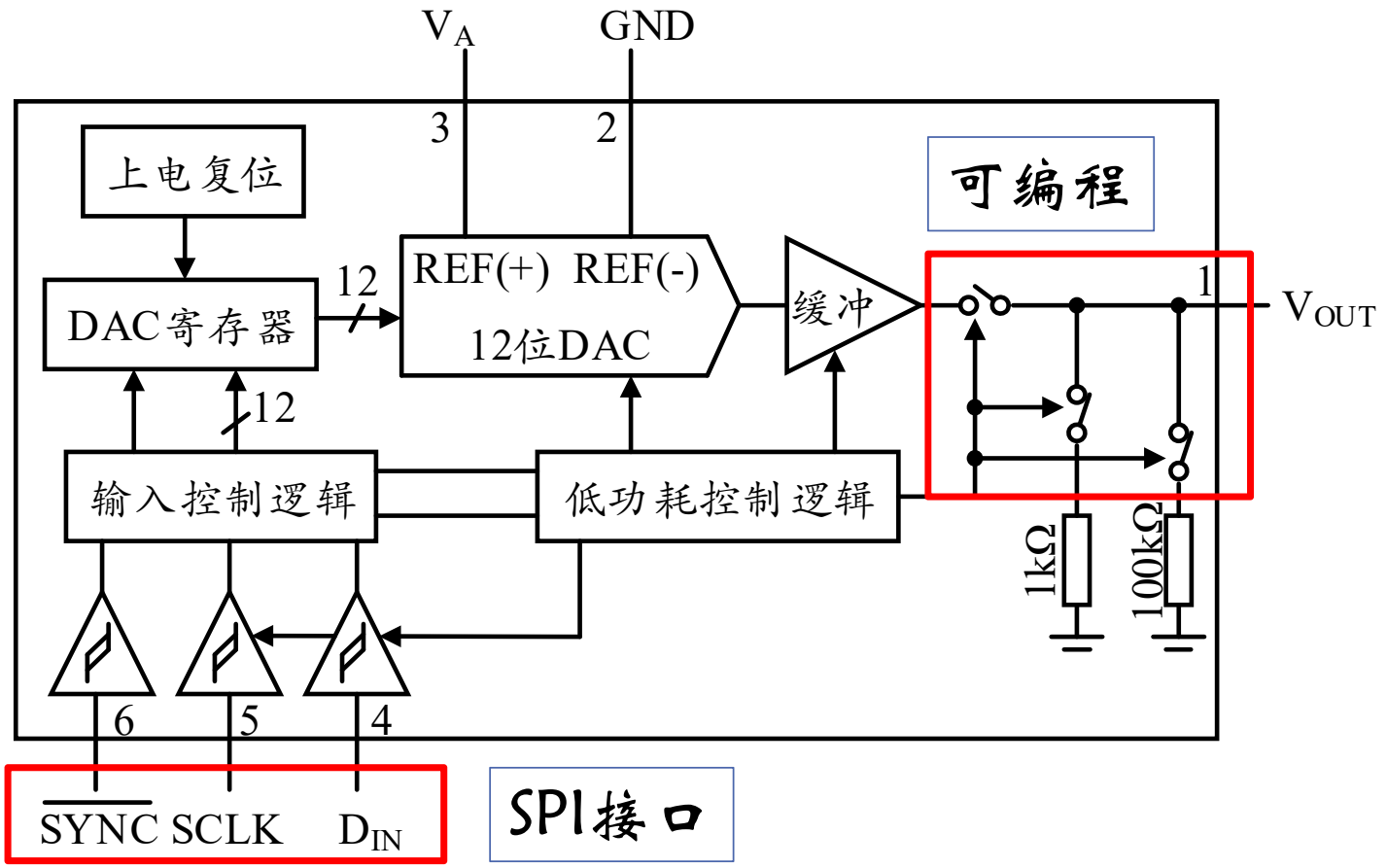
中断初始化

- 写SPICR设置SPI接口工作模式
- 自动输出SS
- 写SPISSR选择通信从设备
- 写IPIER\DGIER使能中断源
- 写SPIDTR发送数据，实现SPI总线全双工通信，触发中断源

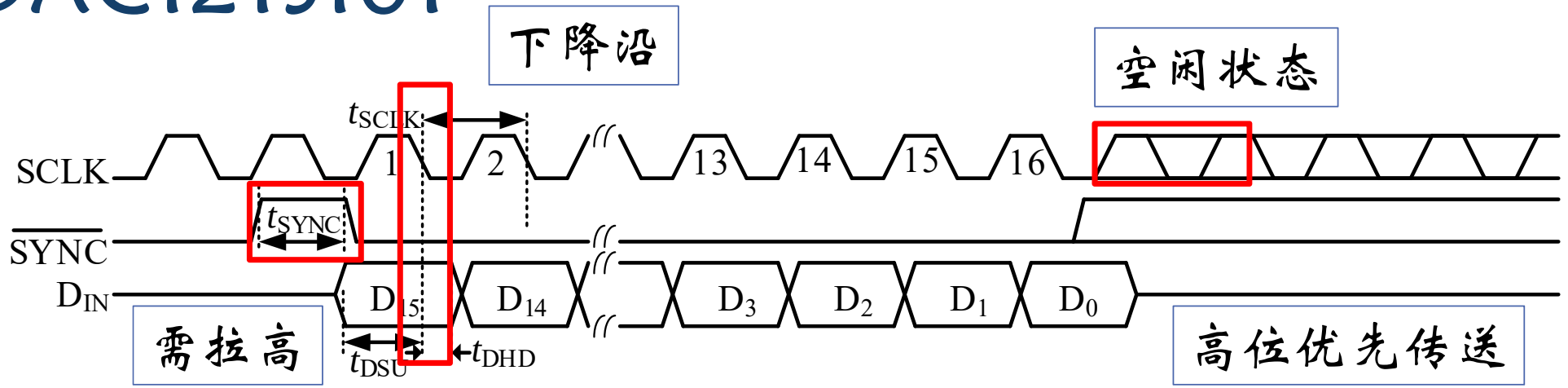
中断服务

- 读、写IPISR清除中断状态
- 读SPIDRR获取接收的数据
- 写SPIDTR发送下一个数据，触发下一次中断源

SPI应用示例——DAC121S101



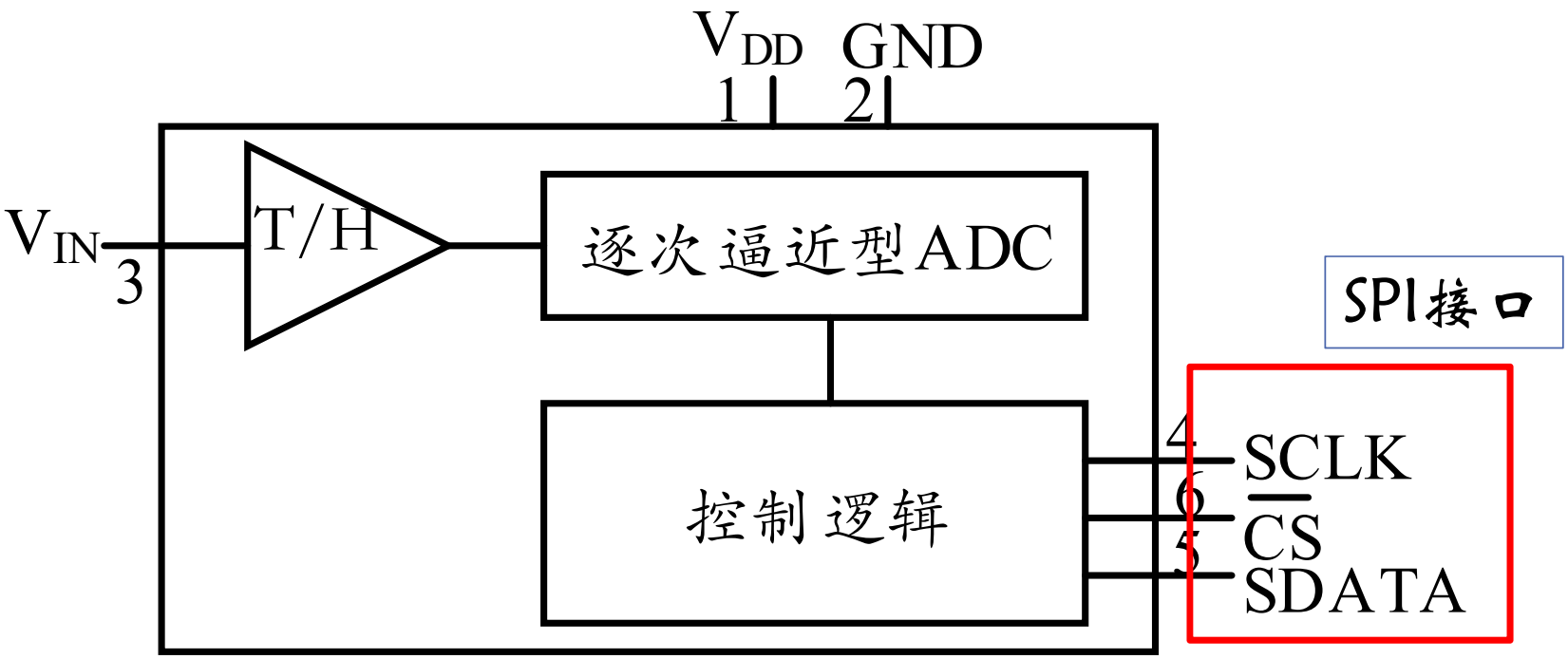
DAC121S101



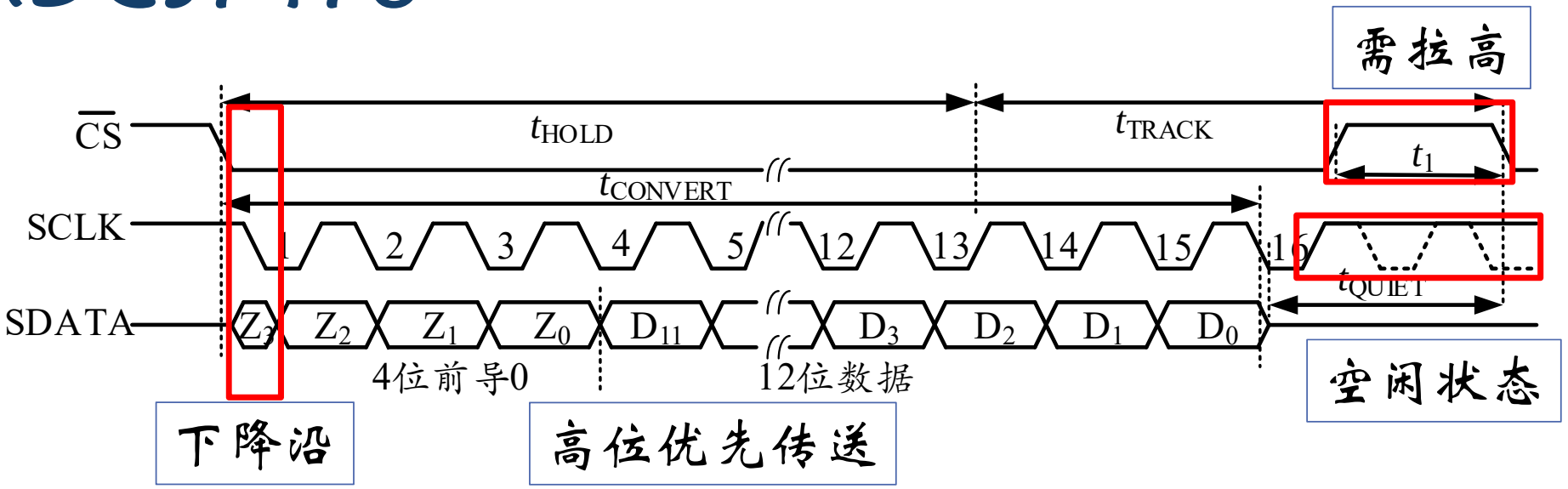
D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	……	D_0
无意义		PD_1	PD_0	D		

PD_1	PD_0	V_{OUT}输出方式
0	0	V_{OUT}正常输出（不下拉）
0	1	V_{OUT}通过1kΩ电阻下拉
1	0	V_{OUT}通过100kΩ电阻下拉
1	1	V_{OUT}高阻（无输出）

SPI应用示例——ADCS7476



ADCS7476


需拉高
$\overline{CS}$
SCLK
SDATA
t_{HOLD}
t_{TRACK}
$t_{CONVERT}$
t_1
t_{QUIET}
1 2 3 4 5 12 13 14 15 16
Z_3 Z_2 Z_1 Z_0 D_{11} D_3 D_2 D_1 D_0
4位前导0
12位数据
下降沿
高位优先传送
空闲状态

SPI应用示例

某计算机系统要求利用ADCS7476采集某电路输出电压（范围为0~3.3V），采样频率为5kHz，连续采集100个数据，然后再利用DAC121S101 DA转换芯片将采集到的100个数据作为一个周期的数据样本以最快速度转换为模拟电压信号输出。试设计该数据采集及转换系统接口电路和控制程序。

定时采样采用定时器中断

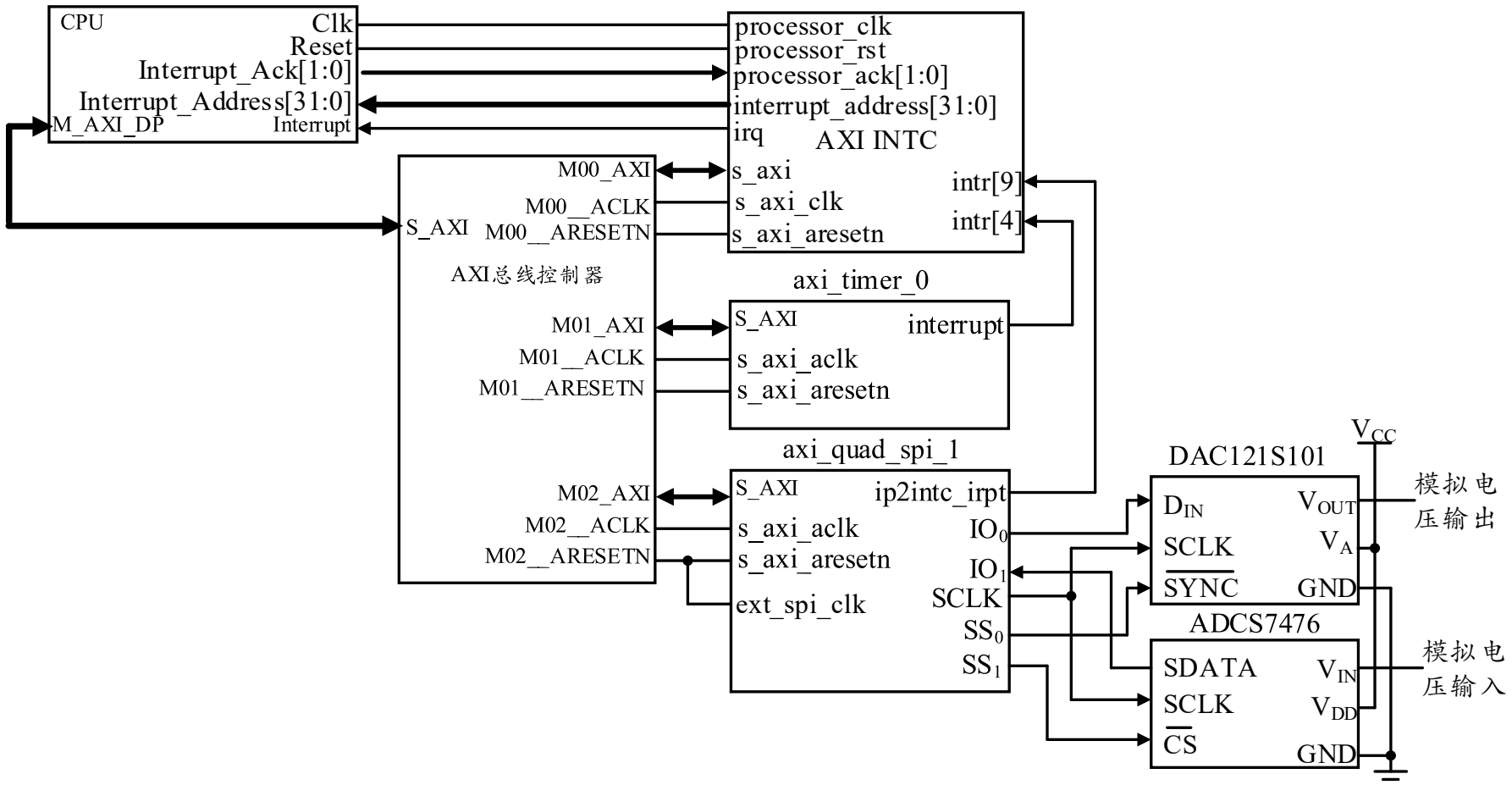
中断100次

数据发送完立即中断，继续发送下一个数据

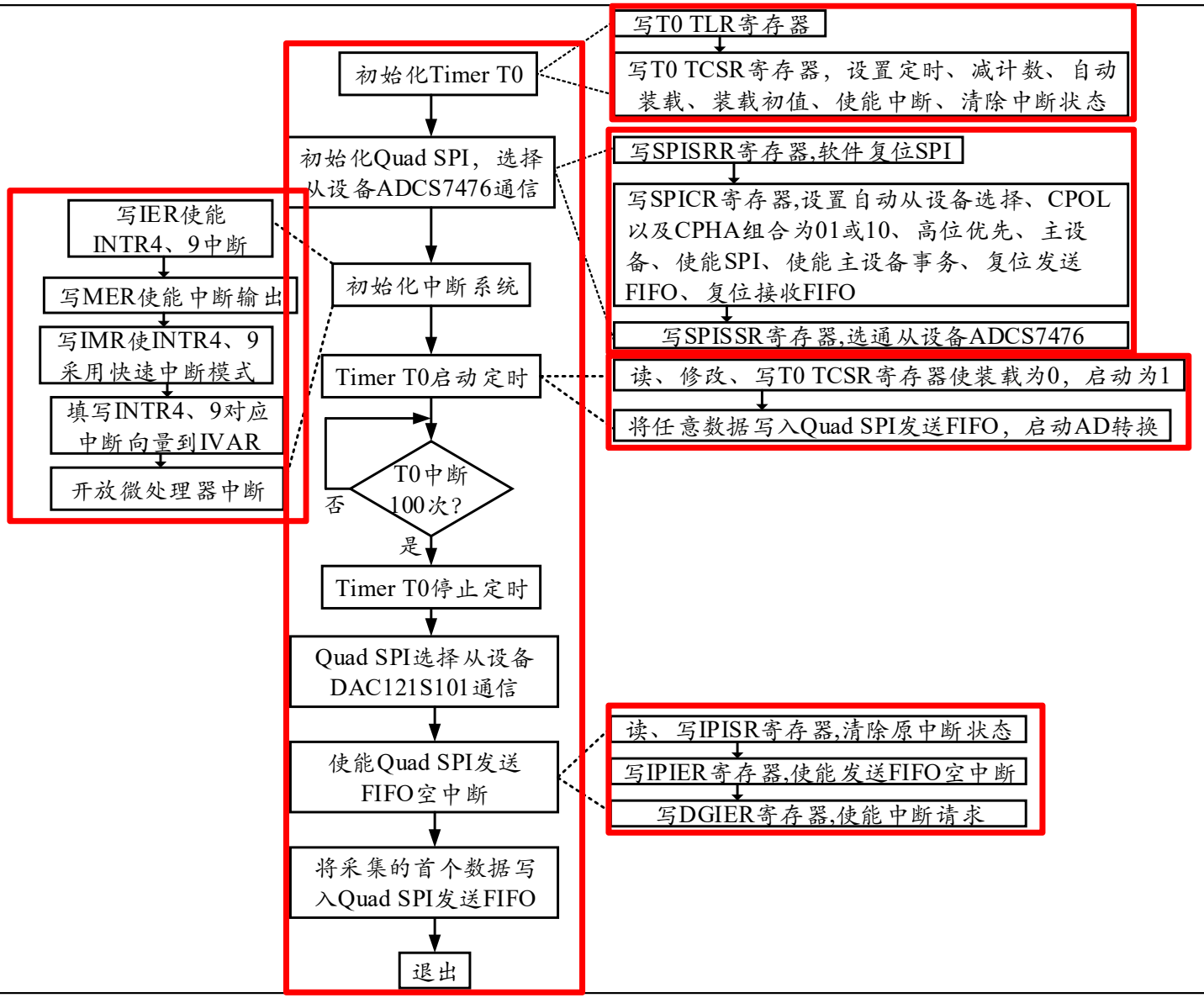
不停中断

定时器、中断控制器、SPI两个从设备（两个SPI）

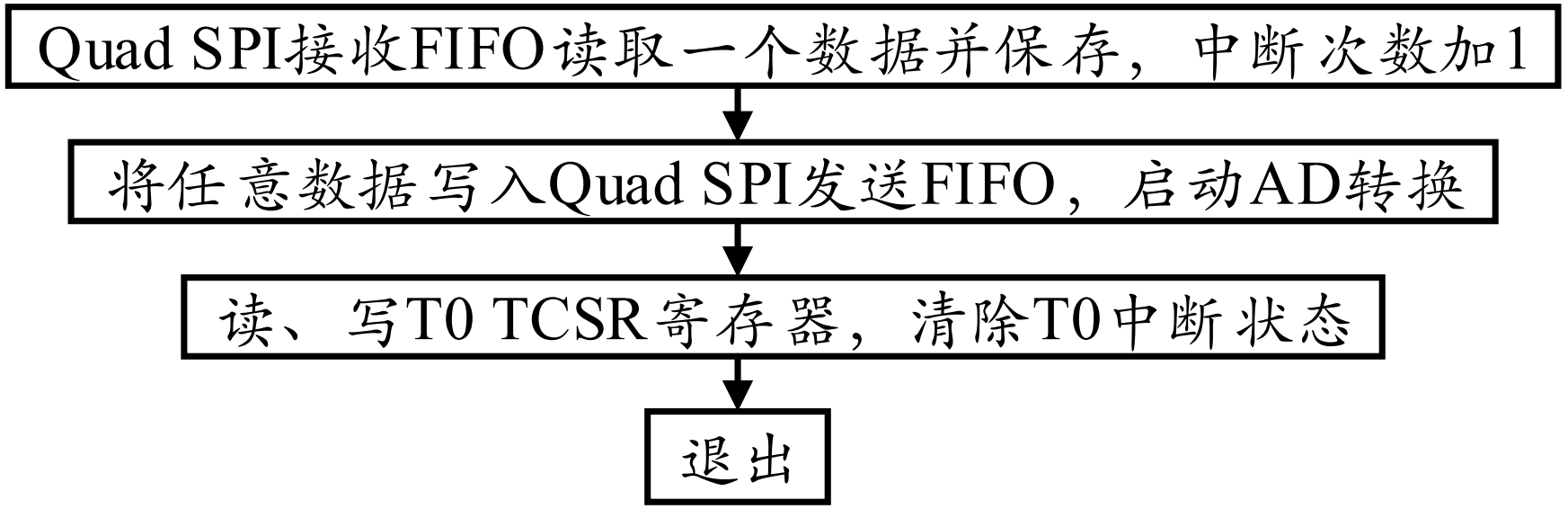
接口电路示例


CPU
Clk
Reset
Interrupt_Ack[1:0]
Interrupt_Address[31:0]
M_AXI_DP
Interrupt
processor_clk
processor_rst
processor_ack[1:0]
interrupt_address[31:0]
irq
AXI INTC
intr[9]
intr[4]
M00_AXI
M00__ACLK
S_AXI
M00__ARESETN
s_axi
s_axi_clk
s_axi_aresetn
AXI总线控制器
axi_timer_0
M01_AXI
M01__ACLK
M01__ARESETN
S_AXI
interrupt
s_axi_aclk
s_axi_aresetn
axi_quad_spi_1
M02_AXI
M02__ACLK
M02__ARESETN
S_AXI
ip2intc_irpt
s_axi_aclk
s_axi_aresetn
ext_spi_clk
IO0
IO1
SCLK
SS0
SS1
DAC121S101
DIN
SCLK
SYNC
VOUT
VA
GND
VCC
模拟电压输出
ADCS7476
SDATA
SCLK
CS
VIN
VDD
GND
模拟电压输入

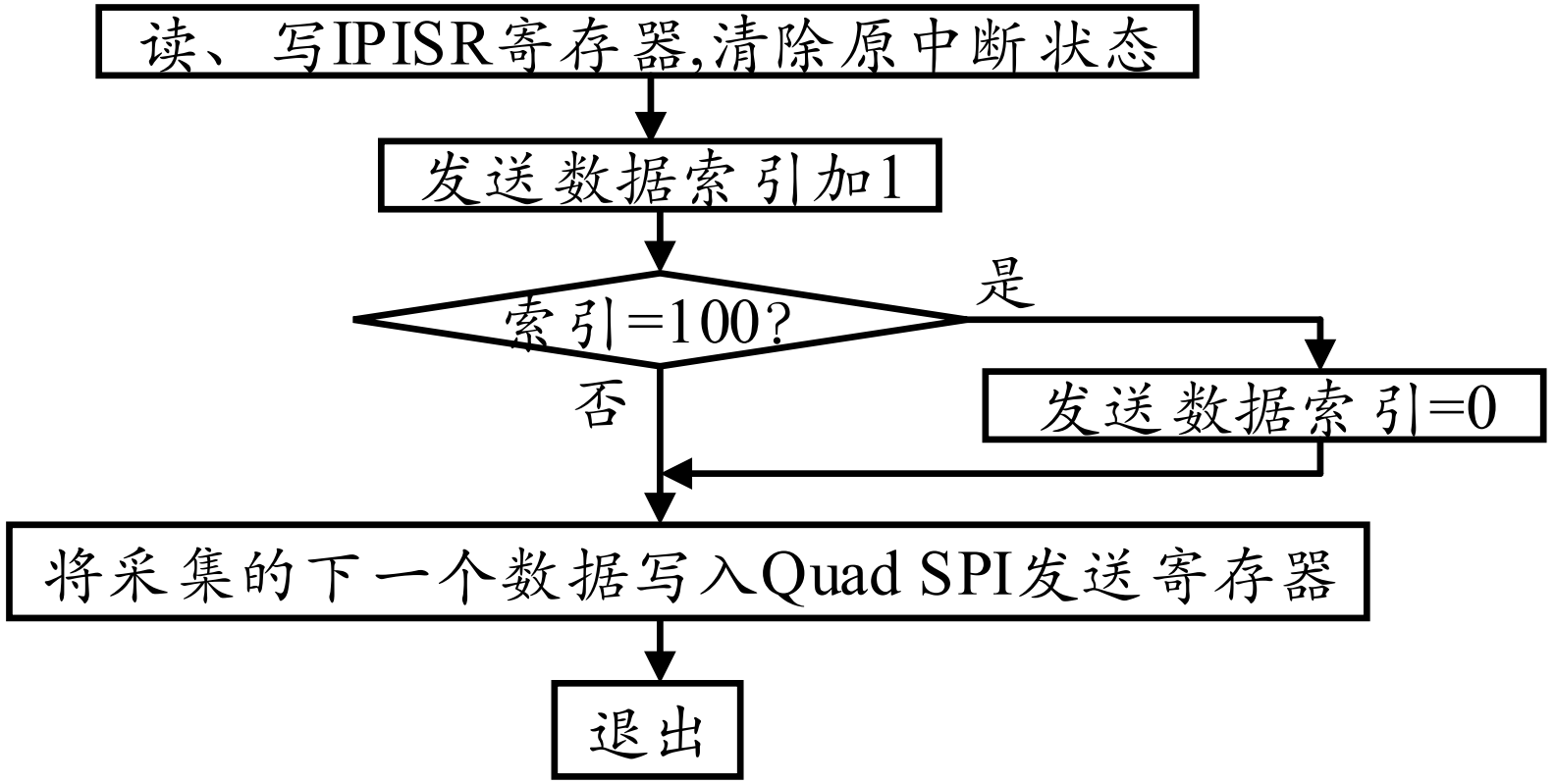
主程序流程


初始化Timer T0
写T0 TLR寄存器
写T0 TCSR寄存器，设置定时、减计数、自动装载、装载初值、使能中断、清除中断状态
初始化Quad SPI，选择从设备ADCS7476通信
写SPISRR寄存器,软件复位SPI
写SPICR寄存器,设置自动从设备选择、CPOL以及CPHA组合为01或10、高位优先、主设备、使能SPI、使能主设备事务、复位发送FIFO、复位接收FIFO
写SPISSR寄存器,选通从设备ADCS7476
写IER使能INTR4、9中断
写MER使能中断输出
写IMR使INTR4、9采用快速中断模式
填写INTR4、9对应中断向量到IVAR
开放微处理器中断
初始化中断系统
Timer T0启动定时
读、修改、写T0 TCSR寄存器使装载为0，启动为1
将任意数据写入Quad SPI发送FIFO，启动AD转换
T0中断100次？
否
是
Timer T0停止定时
Quad SPI选择从设备DAC121S101通信
使能Quad SPI发送FIFO空中断
读、写IPISR寄存器,清除原中断状态
写IPIER寄存器,使能发送FIFO空中断
写DGIER寄存器,使能中断请求
将采集的首个数据写入Quad SPI发送FIFO
退出

T0中断服务程序流程


Quad SPI接收FIFO读取一个数据并保存，中断次数加1
将任意数据写入Quad SPI发送FIFO，启动AD转换
读、写T0 TCSR寄存器，清除T0中断状态
退出

SPI发送中断服务程序流程



SPICR寄存器含义

位	含义	1	0
0	回环	SPI发送端(MOSI)与接收端(MISO)内部连通形成环路	SPI发送端(MOSI)与接收端(MISO)独立工作
1	启用接口	启用SPI接口(与SPI总线连接)	禁用SPI接口(与SPI总线断开、高阻态)
2	主设备	SPI主设备接口	SPI从设备接口
3	CPOL	空闲时时钟为高电平	空闲时时钟为低电平
4	CPHA	数据信号在时钟第二个边沿(相位180°)稳定有效	数据信号在时钟第一个边沿(相位0°)稳定有效
5	Tx 复位	复位发送FIFO指针	无意义
6	Rx 复位	复位接收FIFO指针	无意义
7	手动控制从设备选择	程序控制从设备选择SS输出，即写SPISSR立即反映到$\overline{66}$[n-1:0]	协议逻辑控制SPISSR输出到$\overline{66}$[n-1:0]
8	禁止主设备事务	禁止主设备事务；若为从设备则无意义	使能主设备事务
9	低位优先	低位优先传送	高位优先传送
10~31	保留，无意义		

中断初始化程序-快速中断

```
int main()
{    int status;
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_ISR_OFFSET,
Xil_In32(XPAR_INTC_0_BASEADDR+XIN_ISR_OFFSET));
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IER_OFFSET,
XPAR_AXI_TIMER_0_INTERRUPT_MASK|XPAR_AXI_QUAD_SPI_1_IP2INTC_IRPT_MASK);
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IMR_OFFSET,
XPAR_AXI_TIMER_0_INTERRUPT_MASK|XPAR_AXI_QUAD_SPI_1_IP2INTC_IRPT_MASK);
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_MER_OFFSET,XIN_INT_MASTER_ENABLE_MASK|XIN_INT_HARDW
ARE_ENABLE_MASK);
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IVAR_OFFSET+4*XPAR_INTC_0_TMRCTR_0_VEC_ID,(int)T0Handler);
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IVAR_OFFSET+4*XPAR_INTC_0_SPI_1_VEC_ID,(int)SPIHandler);
     microblaze_enable_interrupts();

     Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TLR_OFFSET,RESET_VALUE);
     Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,XTC_CSR_INT_OCCURED_MASK|XTC_CSR_AUT
O_RELOAD_MASK|XTC_CSR_DOWN_COUNT_MASK|XTC_CSR_LOAD_MASK  |XTC_CSR_ENABLE_INT_MASK);
     status=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET);
     status=(status&(~XTC_CSR_LOAD_MASK))|XTC_CSR_ENABLE_TMR_MASK;
     Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,status);
```

中断初始化程序

```
Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_SRR_OFFSET,XSP_SRR_RESET_MASK);
Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_CR_OFFSET,XSP_CR_ENABLE_MASK|
        XSP_CR_MASTER_MODE_MASK|XSP_CR_CLK_POLARITY_MASK
        |XSP_CR_TXFIFO_RESET_MASK|XSP_CR_RXFIFO_RESET_MASK);
Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_SSR_OFFSET,0x1);
Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_DTR_OFFSET,0x0);
while(int_times<100);
status=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET);
status=status&(~XTC_CSR_ENABLE_TMR_MASK);
Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,status);
Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_SSR_OFFSET,0x2);
Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_IISR_OFFSET,
        Xil_In32(XPAR_SPI_1_BASEADDR+XSP_IISR_OFFSET));
Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_IIER_OFFSET,
        XSP_INTR_TX_EMPTY_MASK);
Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_DGIER_OFFSET,
        XSP_GINTR_ENABLE_MASK);
int_times=0;
Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_DTR_OFFSET,samples[int_times]);
return 0;}
```

中断初始化程序

```
#include "xparameters.h"
#include "xtmrctr_l.h"
#include "xspi_l.h"
#include "xintc_l.h"
#include "xil_io.h"
#include "xil_exception.h"
#define RESET_VALUE 2000-2
void T0Handler() __attribute__((fast_interrupt));
void SPIHandler() __attribute__((fast_interrupt));
short samples[100];
int int_times;
```

T0中断服务程序-读取AD转换结果并启动AD转换

```
void T0Handler()
{
samples[int_times]=(short)(Xil_In32(XPAR_SPI_1_BASEADDR+XSP_DRR_OFFSET)
            &0xfff);
int_times++;
Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_DTR_OFFSET,0x0);
Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,
            Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET));
}
```

SPI发送结束中断服务程序

```
void SPIHandler()
{
    Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_IISR_OFFSET,
            Xil_In32(XPAR_SPI_1_BASEADDR+XSP_IISR_OFFSET));
    int_times++;
    if(int_times==100)
        int_times=0;
    Xil_Out32(XPAR_SPI_1_BASEADDR+XSP_DTR_OFFSET,samples[int_times]);
}
```

中断初始化程序-普通中断

```
#include "xparameters.h"
#include "xtmrctr_l.h"
#include "xspi_l.h"
#include "xintc_l.h"
#include "xil_io.h"
#include "xil_exception.h"
#define RESET_VALUE 2000-2
void T0Handler()  __attribute__((fast_interrupt));
void SPIHandler()  __attribute__((fast_interrupt));
short samples[100];
int int_times;
```

```
Void My_ISR __attribute__(interrupt_handler)
```

中断初始化程序-普通中断

```
int main()
{    int status;
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_ISR_OFFSET,
Xil_In32(XPAR_INTC_0_BASEADDR+XIN_ISR_OFFSET));
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IER_OFFSET,
XPAR_AXI_TIMER_0_INTERRUPT_MASK|XPAR_AXI_QUAD_SPI_1_IP2INTC_IRPT_MASK);
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IMR_OFFSET,
XPAR_AXI_TIMER_0_INTERRUPT_MASK|XPAR_AXI_QUAD_SPI_1_IP2INTC_IRPT_MASK);
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_MER_OFFSET,XIN_INT_MASTER_ENABLE_MASK|XIN_INT_HARDW
ARE_ENABLE_MASK);
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IVAR_OFFSET+4*XPAR_INTC_0_TMRCTR_0_VEC_ID,(int)T0Handler);
     Xil_Out32(XPAR_INTC_0_BASEADDR+XIN_IVAR_OFFSET+4*XPAR_INTC_0_SPI_1_VEC_ID,(int)SPIHandler);
     microblaze_enable_interrupts();

     Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TLR_OFFSET,RESET_VALUE);
     Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,XTC_CSR_INT_OCCURED_MASK|XTC_CSR_AUT
O_RELOAD_MASK|XTC_CSR_DOWN_COUNT_MASK|XTC_CSR_LOAD_MASK  |XTC_CSR_ENABLE_INT_MASK);
     status=Xil_In32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET);
     status=(status&(~XTC_CSR_LOAD_MASK))|XTC_CSR_ENABLE_TMR_MASK;
     Xil_Out32(XPAR_TMRCTR_0_BASEADDR+XTC_TCSR_OFFSET,status);
```

总中断服务程序

```
void My_ISR()
{
      int status;
      status=Xil_In32(XPAR_AXI_INTC_0_BASEADDR+XIN_ISR_OFFSET);//读取ISR
      if((status&XPAR_AXI_TIMER_0_INTERRUPT_MASK)==XPAR_AXI_TIMER_0_INTERRUPT_MASK)
            T0Handler();//调用T0中断服务程序
      if((status&XPAR_AXI_QUAD_SPI_1_IP2INTC_IRPT_MASK)==
                        XPAR_AXI_QUAD_SPI_1_IP2INTC_IRPT_MASK)
           SPIHandler ();//调用SPI中断
      Xil_Out32(XPAR_AXI_INTC_0_BASEADDR+XIN_IAR_OFFSET,status);//写IAR
}
```

小结

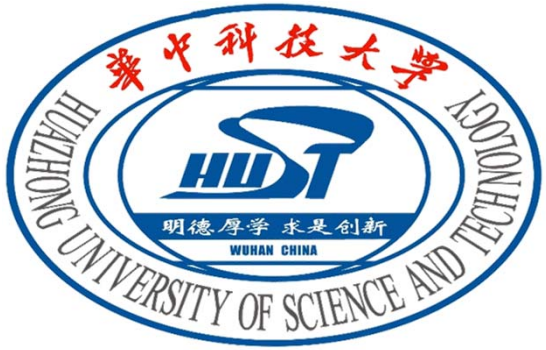
- SPI中断方式编程控制
  - 写控制寄存器控制工作方式
  - 写IPIER\IPGIER开放中断
    - 发送空、接收满中断
  - 促使产生中断
    - 先发送数据，收发全双工
- 再次回顾快速中断、普通中断编程差别
- 再次回顾定时器中断编程控制

下一讲：多中断源程序设计

微机原理与接口技术

多中断源程序设计示例

华中科技大学　左冬红



多中断源应用系统示例

基于嵌入式微处理器设计一个同时支持多种并行IO设备工作的嵌入式实时MIMO系统。基本输入输出设备有：16个独立LED灯，16个独立开关、2个独立按键，4个七段数码管。且都通过GPIO并行输入输出接口连接到单核微处理器计算机系统的同一总线上。

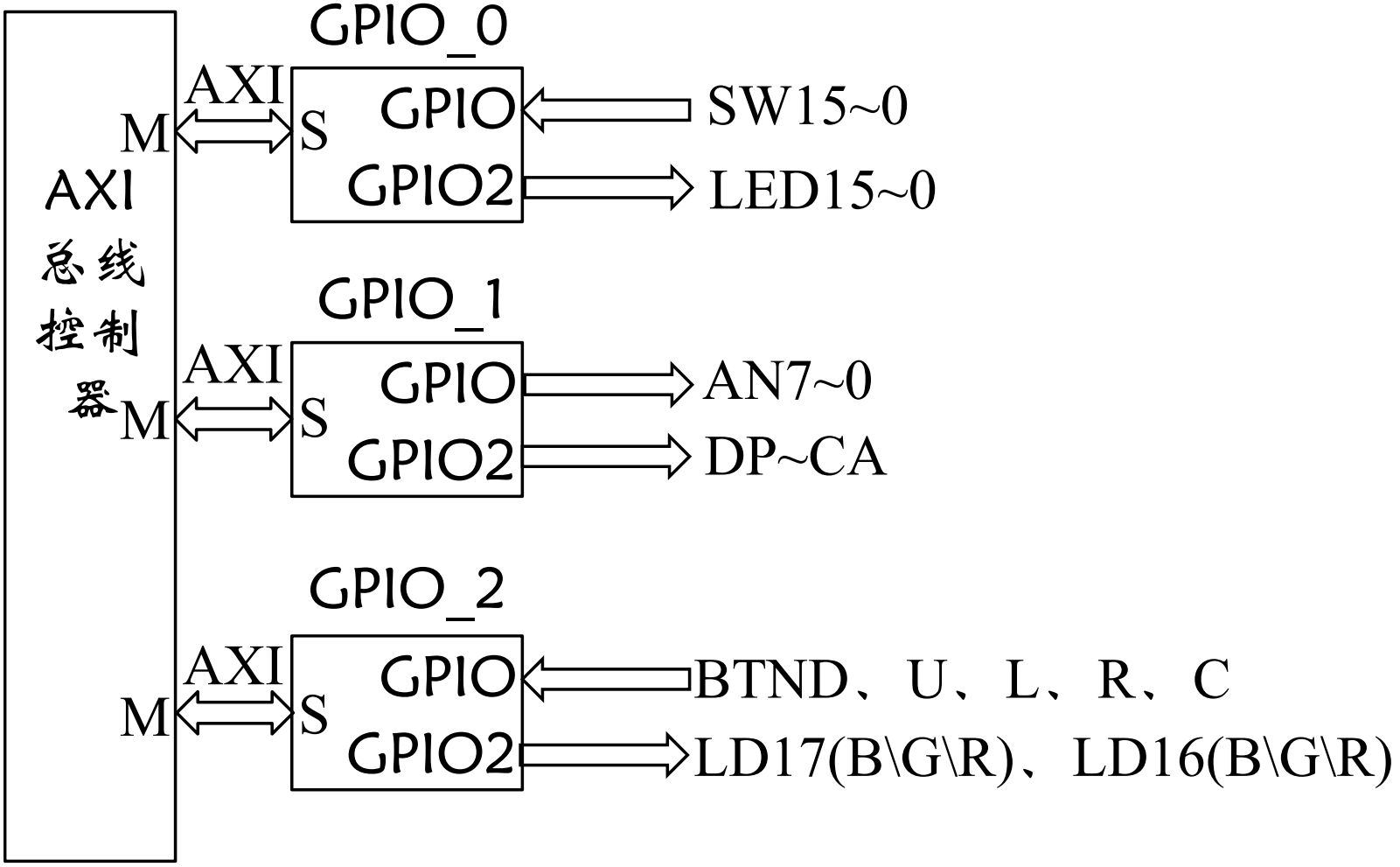
要求实现的基本功能为：

1) 16个LED灯走马灯式轮流循环亮灭。且循环速度可通过两个独立按键步进控制，其中一个按键每按下一次步进增速，另一个按键每按一次步进减速。

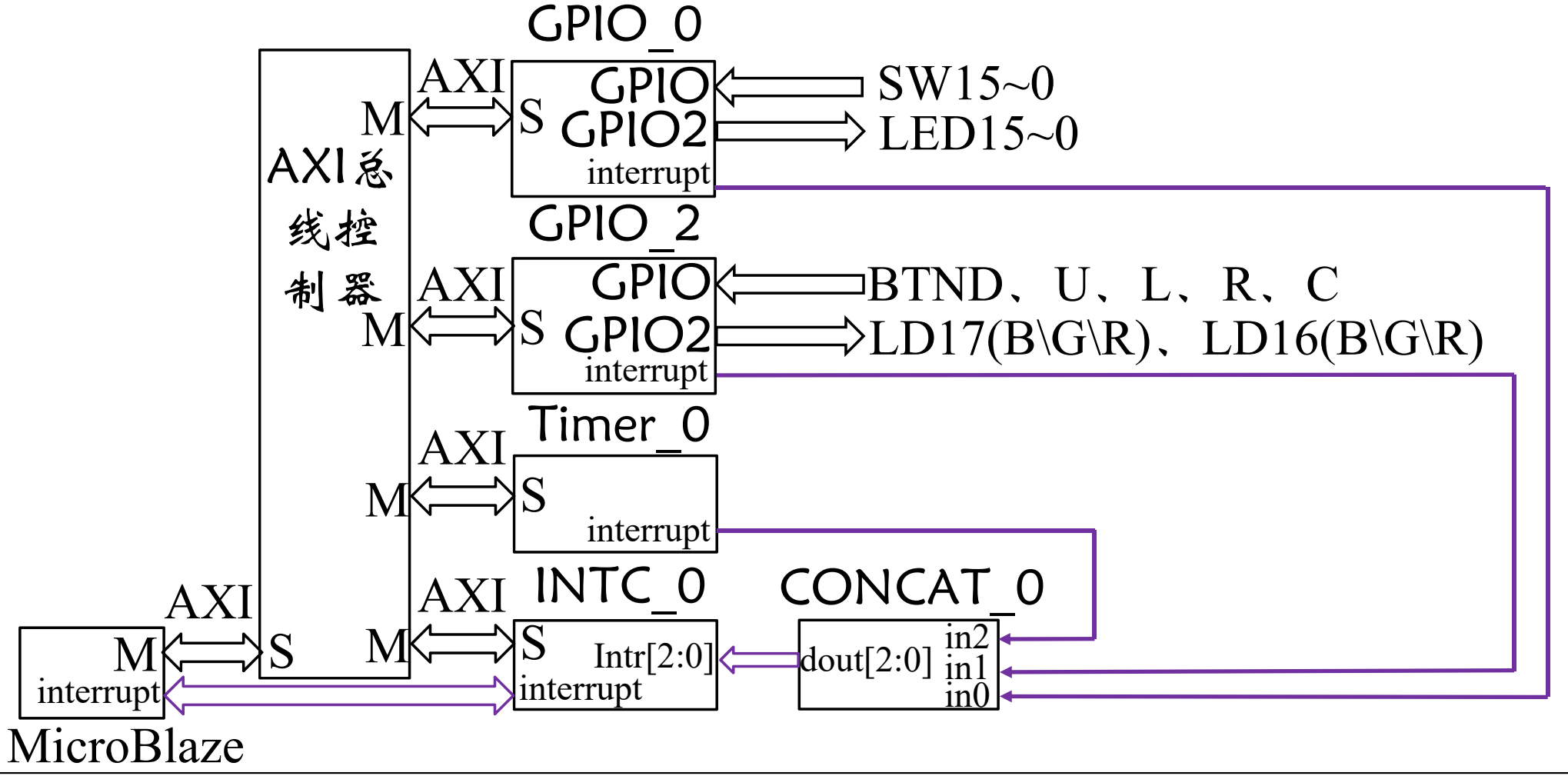
2) 4个七段数码管实时显示16位独立开关表示的十六进制数。

四个中断源
- 两个GPIO
- 两个Timer

GPIO与外设连接关系



并行IO中断系统


GPIO_0
AXI
M
S
GPIO
GPIO2
interrupt
SW15~0
LED15~0
AXI总线控制器
GPIO_2
AXI
M
S
GPIO
GPIO2
interrupt
BTND、U、L、R、C
LD17(B\G\R)、LD16(B\G\R)
Timer_0
AXI
M
S
interrupt
AXI
M
S
INTC_0
AXI
M
S
Intr[2:0]
interrupt
CONCAT_0
dout[2:0]
in2
in1
in0
AXI
M
S
interrupt
MicroBlaze

普通中断方式——初始化流程

- 各GPIO工作方式设定：输入、输出
- 各Timer工作方式设定
  - 停止Timer
  - 设定初值
    - 装载初值
  - 启动Timer、开中断、自动装载、减计数
- 开放中断系统
  - GPIO_0开中断
  - GPIO_2开中断
  - INTC_0开中断
  - MicroBlaze_0开中断
- 注册总中断服务程序

普通中断方式——总中断服务程序

读INTC ISR

判断intr0　　调用开关中断事务处理函数

判断intr1　　调用按键中断事务处理函数

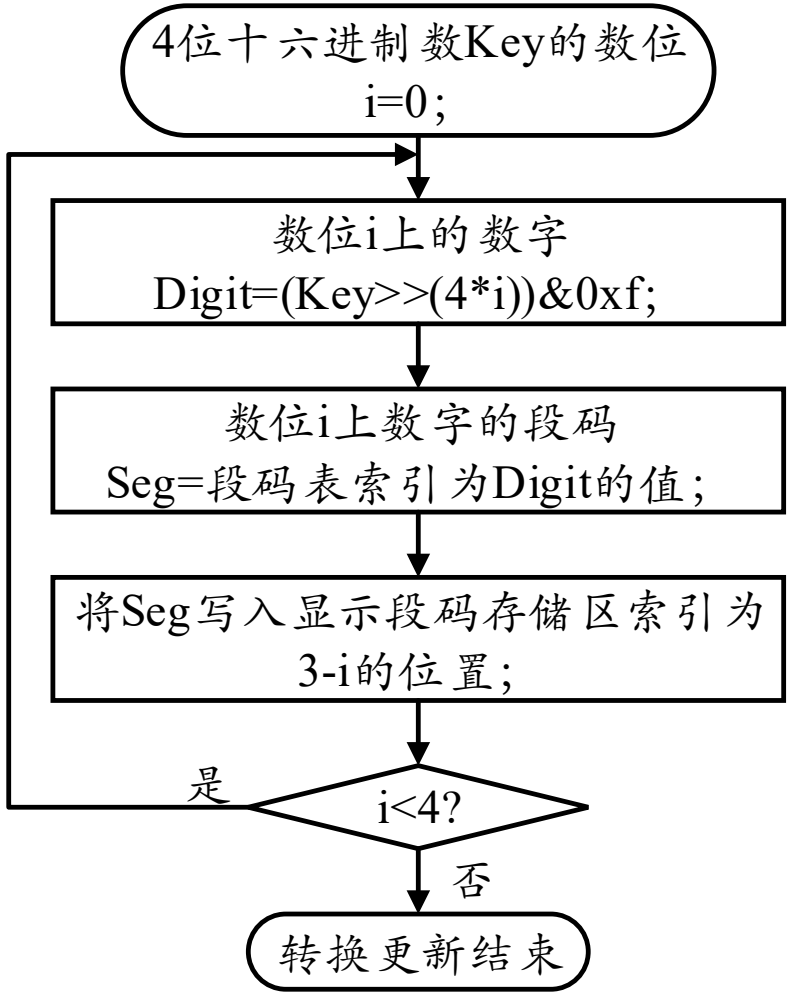
判断intr2　　调用定时器中断事务处理函数

写INTC IAR清除ISR状态

开关中断事务处理程序

读取开关值、更新数码管显示缓冲区

读后写GPIO_0 ISR清除ISR状态



按键中断事务处理程序

读取按键值，改变走马灯定时器计数初值

判断增速按键

将定时器初值减少一固定值

判断减速按键

将定时器初值增加一固定值

读后写GPIO_2 ISR清除ISR状态

定时器总中断事务处理程序

判断定时器0中断

调用定时器0中断事务处理函数

读后写定时器0 TCSR 清除定时器0中断状态

判断定时器1中断

调用定时器1中断事务处理函数

读后写定时器1 TCSR 清除定时器1中断状态

走马灯定时器中断事务处理程序

控制当前位置LED灯点亮，并将当前位置修改为下一位

数码管动态扫描定时器中断事务处理程序

输出当前位置数码管段码、位码，并将当前位置修改为下一位

普通中断方式初始化程序代码

```
int main()
{
Xil_Out8(XPAR_AXI_GPIO_2_BASEADDR+XGPIO_TRI_OFFSET,0x1f);
Xil_Out16(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_TRI_OFFSET,0xffff);
Xil_Out16(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_TRI2_OFFSET,0x0);
Xil_Out16(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_DATA2_OFFSET,0x1);
Xil_Out8(XPAR_AXI_GPIO_1_BASEADDR+XGPIO_TRI_OFFSET,0x0);
Xil_Out8(XPAR_AXI_GPIO_1_BASEADDR+XGPIO_TRI2_OFFSET,0x0);
Xil_Out32(XPAR_AXI_GPIO_2_BASEADDR+XGPIO_IER_OFFSET,XGPIO_IR_CH1_MASK);//
Xil_Out32(XPAR_AXI_GPIO_2_BASEADDR+XGPIO_GIE_OFFSET,XGPIO_GIE_GINTR_ENABLE_MASK);//
Xil_Out32(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_IER_OFFSET,XGPIO_IR_CH1_MASK);//
Xil_Out32(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_GIE_OFFSET,XGPIO_GIE_GINTR_ENABLE_MASK);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET,Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET)&~XTC_CSR_ENABLE_TMR_MASK);
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TLR_OFFSET,RESET_VALUE0);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET,Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET)|XTC_CSR_LOAD_MASK);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET,(Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET)&~XTC_CSR_LOAD_MASK)
|XTC_CSR_ENABLE_TMR_MASK|XTC_CSR_AUTO_RELOAD_MASK|XTC_CSR_ENABLE_INT_MASK|XTC_CSR_DOWN_COUNT_MASK);
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET)&~XTC_CSR_ENABLE_TMR_MASK);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TLR_OFFSET,RESET_VALUE1);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET)|XTC_CSR_LOAD_MASK);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
(Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET)&~XTC_CSR_LOAD_MASK)\
|XTC_CSR_ENABLE_TMR_MASK|XTC_CSR_AUTO_RELOAD_MASK|XTC_CSR_ENABLE_INT_MASK|XTC_CSR_DOWN_COUNT_MASK);
Xil_Out32(XPAR_AXI_INTC_0_BASEADDR+XIN_IER_OFFSET,XPAR_AXI_GPIO_0_IP2INTC_IRPT_MASK
|XPAR_AXI_GPIO_2_IP2INTC_IRPT_MASK|XPAR_AXI_TIMER_0_INTERRUPT_MASK);//
Xil_Out32(XPAR_AXI_INTC_0_BASEADDR+XIN_MER_OFFSET,XIN_INT_MASTER_ENABLE_MASK|XIN_INT_HARDWARE_ENABLE_MASK);
microblaze_enable_interrupts();
microblaze_register_handler((XInterruptHandler)My_ISR, (void *)0);
}
```

总中断服务程序代码

```
void My_ISR()
{
int status;
status=Xil_In32(XPAR_AXI_INTC_0_BASEADDR+XIN_ISR_OFFSET);//
if((status&XPAR_AXI_GPIO_0_IP2INTC_IRPT_MASK)==XPAR_AXI_GPIO_0_IP2INTC_IRPT_MASK)
switch_handle();
if((status&XPAR_AXI_GPIO_2_IP2INTC_IRPT_MASK)==XPAR_AXI_GPIO_2_IP2INTC_IRPT_MASK)
button_handle();
if ((status&XPAR_AXI_TIMER_0_INTERRUPT_MASK)==XPAR_AXI_TIMER_0_INTERRUPT_MASK)
timer_handle();
Xil_Out32(XPAR_AXI_INTC_0_BASEADDR+XIN_IAR_OFFSET,status);//
}
```

开关中断事务处理程序代码

```
void switch_handle()
{
short hex=Xil_In16(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_DATA_OFFSET);
int segcode_index=3;
for(int digit_index=0;digit_index<4;digit_index++)
{
segcode[segcode_index]=segtable[(hex>>(4*digit_index))&0xf];
segcode_index--;
}
Xil_Out32(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_ISR_OFFSET,
Xil_In32(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_ISR_OFFSET));//
}
```

按键中断事务处理程序

```
void button_handle()
{
    char button;
    button = Xil_In8(XPAR_AXI_GPIO_2_BASEADDR+XGPIO_DATA_OFFSET)&0x1f;
    if(button==0x2)
        Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TLR_OFFSET,
                Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TLR_OFFSET)-STEP_PACE);
    if (button==0x10)
        Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TLR_OFFSET,
                Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TLR_OFFSET)+STEP_PACE);
    Xil_Out32(XPAR_AXI_GPIO_2_BASEADDR+XGPIO_ISR_OFFSET,
                Xil_In32(XPAR_AXI_GPIO_2_BASEADDR+XGPIO_ISR_OFFSET));//
}
```

定时器总中断服务程序代码

```
void timer_handle()
{
int status;
status=Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET);//
if((status&XTC_CSR_INT_OCCURED_MASK)==XTC_CSR_INT_OCCURED_MASK)
timer0_handle();
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET,
Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET));
status=Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET);//
if((status&XTC_CSR_INT_OCCURED_MASK)==XTC_CSR_INT_OCCURED_MASK)
timer1_handle();
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET));
}
```

定时器中断事务处理程序代码

走马灯定时

```
void timer0_handle()
{
    ledbits++;
    if(ledbits==16)
        ledbits=0;
    Xil_Out16(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_DATA2_OFFSET,1<<ledbits);
}
```

数码管动态扫描定时

```
void timer1_handle()
{
    Xil_Out16(XPAR_AXI_GPIO_1_BASEADDR+XGPIO_DATA2_OFFSET,segcode[pos]);
    Xil_Out16(XPAR_AXI_GPIO_1_BASEADDR+XGPIO_DATA_OFFSET,poscode[pos]);
    pos++;
    if(pos==4)
            pos=0;
}
```

全部变量及函数申明

```
#include "xil_io.h"
#include "stdio.h"
#include "xintc_l.h"
#include "xtmrctr_l.h"
#include "xgpio_l.h"
#define RESET_VALUE0       100000000-2
#define RESET_VALUE1       100000-2
#define STEP_PACE 10000000
void My_ISR();
void switch_handle();
void button_handle();
void timer_handle();
void timer0_handle();
void timer1_handle();
char segtable[16]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x98,0x88,0x83,0xc6,0xa1,0x86,0x8e};
char segcode[4]={0xc0,0xc0,0xc0,0xc0};
short poscode[4]={0xf7,0xfb,0xfd,0xfe};
int ledbits=0;
int pos=0;
```

快速中断——初始化程序

```
int main()
{
Xil_Out8(XPAR_AXI_GPIO_2_BASEADDR+XGPIO_TRI_OFFSET,0x1f);
Xil_Out16(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_TRI_OFFSET,0xffff);
Xil_Out16(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_TRI2_OFFSET,0x0);
Xil_Out16(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_DATA2_OFFSET,0x1);
Xil_Out8(XPAR_AXI_GPIO_1_BASEADDR+XGPIO_TRI_OFFSET,0x0);
Xil_Out8(XPAR_AXI_GPIO_1_BASEADDR+XGPIO_TRI2_OFFSET,0x0);
Xil_Out32(XPAR_AXI_GPIO_2_BASEADDR+XGPIO_IER_OFFSET,XGPIO_IR_CH1_MASK);//
Xil_Out32(XPAR_AXI_GPIO_2_BASEADDR+XGPIO_GIE_OFFSET,XGPIO_GIE_GINTR_ENABLE_MASK);//
Xil_Out32(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_IER_OFFSET,XGPIO_IR_CH1_MASK);//
Xil_Out32(XPAR_AXI_GPIO_0_BASEADDR+XGPIO_GIE_OFFSET,XGPIO_GIE_GINTR_ENABLE_MASK);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET,Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET)&~XTC_CSR_ENABLE_TMR_MASK);
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TLR_OFFSET,RESET_VALUE0);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET,Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET)|XTC_CSR_LOAD_MASK);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET,(Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TCSR_OFFSET)&~XTC_CSR_LOAD_MASK)
|XTC_CSR_ENABLE_TMR_MASK|XTC_CSR_AUTO_RELOAD_MASK|XTC_CSR_ENABLE_INT_MASK|XTC_CSR_DOWN_COUNT_MASK);
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET)&~XTC_CSR_ENABLE_TMR_MASK);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TLR_OFFSET,RESET_VALUE1);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET)|XTC_CSR_LOAD_MASK);//
Xil_Out32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET,
(Xil_In32(XPAR_AXI_TIMER_0_BASEADDR+XTC_TIMER_COUNTER_OFFSET+XTC_TCSR_OFFSET)&~XTC_CSR_LOAD_MASK)\
|XTC_CSR_ENABLE_TMR_MASK|XTC_CSR_AUTO_RELOAD_MASK|XTC_CSR_ENABLE_INT_MASK|XTC_CSR_DOWN_COUNT_MASK);
Xil_Out32(XPAR_AXI_INTC_0_BASEADDR+XIN_IER_OFFSET,XPAR_AXI_GPIO_0_IP2INTC_IRPT_MASK
|XPAR_AXI_GPIO_2_IP2INTC_IRPT_MASK|XPAR_AXI_TIMER_0_INTERRUPT_MASK);//
Xil_Out32(XPAR_AXI_INTC_0_BASEADDR+XIN_MER_OFFSET,XIN_INT_MASTER_ENABLE_MASK|XIN_INT_HARDWARE_ENABLE_MASK);
microblaze_enable_interrupts();
microblaze_register_handler((XInterruptHandler)My_ISR, (void *)0);
}
```

快速中断-初始化程序

配置快速模式

```
Xil_Out32(XPAR_AXI_INTC_0_BASEADDR+XIN_IMR_OFFSET,XPAR_AXI_GPIO_0_IP2INTC_IRPT_MASK
|XPAR_AXI_GPIO_2_IP2INTC_IRPT_MASK|XPAR_AXI_TIMER_0_INTERRUPT_MASK);//
```

填写INTC IVAR

```
Xil_Out32(XPAR_AXI_INTC_0_BASEADDR+XIN_IVAR_OFFSET+
        4*XPAR_AXI_INTC_0_AXI_GPIO_0_IP2INTC_IRPT_INTR,(int)switch_handle);
Xil_Out32(XPAR_AXI_INTC_0_BASEADDR+XIN_IVAR_OFFSET+
        4*XPAR_AXI_INTC_0_AXI_GPIO_2_IP2INTC_IRPT_INTR,(int)button_handle);
Xil_Out32(XPAR_AXI_INTC_0_BASEADDR+XIN_IVAR_OFFSET+
        4*XPAR_AXI_INTC_0_AXI_TIMER_0_INTERRUPT_INTR,(int)timer_handle);
```

全部变量及函数申明

```
#include "xil_io.h"
#include "stdio.h"
#include "xintc_l.h"
#include "xtmrctr_l.h"
#include "xgpio_l.h"
#define RESET_VALUE0       100000000-2
#define RESET_VALUE1       100000-2
#define STEP_PACE 10000000
void My_ISR();
void switch_handle() __attribute__ ((fast_interrupt));
void button_handle() __attribute__ ((fast_interrupt));
void timer_handle() __attribute__ ((fast_interrupt));
void timer0_handle();
void timer1_handle();
char segtable[16]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x98,0x88,0x83,0xc6,0xa1,0x86,0x8e};
char segcode[4]={0xc0,0xc0,0xc0,0xc0};
short poscode[4]={0xf7,0xfb,0xfd,0xfe};
int ledbits=0;
int pos=0;
```

中断事务处理程序函数体不变

程序控制方式实现——请自行设计

跑马灯延时

数码管动态扫描延时

查询开关

查询按键

小结

- 多中断源
  - 产生中断时，中断服务程序处理各自相应业务
  - 中断事务尽可能简单，否则阻塞其他中断源
    - 开启MicroBlaze中断
    - 关闭MicroBlaze中断
- 快速模式-普通模式对比

下一讲：DMA技术