计算机组成原理与接口技术
——基于 MIPS 架构

Chapter 8
INTERRUPT

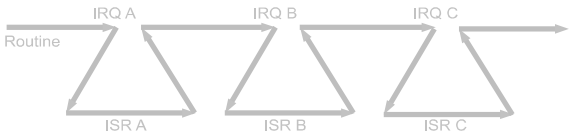
张江山
zhangjs@hust.edu.cn
信息工程系

Content & Objectives

- **Content**
  - The Concept of Interrupt
  - Interrupt System of Xilinx MicroBlaze
  - Design of IO Interface in Interrupt Mode
- **Objectives**
  - Understanding the Concepts of Interrupt
  - Understanding the Principles of Interrupt Controller
  - Master the Design Method of Interface in Interrupt Mode

1 The Concept of Interrupt

- **Interrupt & Exception Source**
  - Interrupt Source: Reset, Timer, Keyboard, Mouse, ADC, etc.
  - Exception Source: Overflow, Undefined instructions, etc.
- **IRQ** (Interrupt ReQuest)
  - The signals are emitted by device to processor
- **ISR** (Interrupt Service Routine, aka Interrupt Handler)
  - Processor pauses current task to process the ***IRQ***
  - The current status is saved before executing the ***ISR***
  - Returns to the last breakpoint after the ***ISR*** is completed

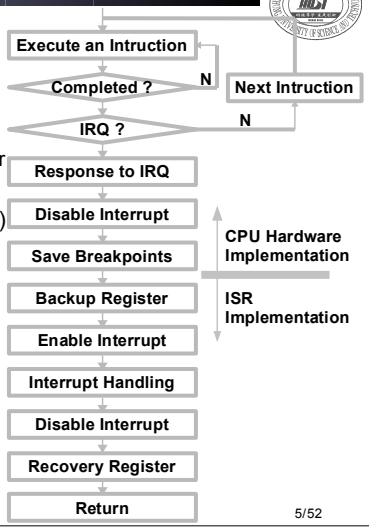


1 The Concept of Interrupt

- **INUM** (Interrupt Vector/Type NUMber)
  - Index number of the interrupt
- **IVEC** (Interrupt VECtor)
  - Address Description of ***ISR***
- **IVT** (Interrupt Vector Table)
- **NMI** (Non-Maskable Interrupt Request)
- **INTR** (Maskable INTerrupt Request)

1 The Concept of Interrupt

- **Interrupt Response Flow**
  - Identify Interrupt Source
    - ➢The status flags in register
      - e.g. MIPS
    - ➢Interrupt (vector/type) number
      - e.g. x86
  - Interrupt Acknowledge (***INTA***)
    - ➢Interrupt Priority
      - Polling Order
      - Priority Encoder
  - Save Breakpoint
    - ➢Register (e.g. MIPS)
    - ➢Stack (e.g. x86)



2 Interrupt System of Xilinx MicroBlaze

- **Interrupt / Exception Type (8-byte Vector)**
  - Hardware jumps to **C_BASE_VECTORS + Offset**
    - ➢Normal interrupt: Compiler allocates vector address for labels
    - ➢Fast interrupt: Interrupt controller supply vector address for labels

Event	Offset	Software jumps to Labels	Register Return Address	Priority
Reset	0x0	`imm _start[31:16]` `bri _start[15:0]`	-	1
User Vector (Exception)	0x8	`imm _exception_handler[31:16]` `bri _exception_handler[15:0]`	r15	7
Interrupt (User maskable)	0x10	`imm _interrupt_handler[31:16]` `bri _interrupt_handler[15:0]`	r14	6
Non-maskable Hardware Break	0x18	`imm _break_handler[31:16]` `bri _break_handler[15:0]`	r16	3
Maskable Hardware Break				4
Software Break				5
Hardware Exception	0x20	`imm _hw_exception_handler[31:16]` `bri _hw_exception_handler[15:0]`	r17	2

- **Xilinx SDK for the Interrupt on Microblaze**
  - Set custom routine as normal / fast ***ISR*** at precompile time

```
void customHandler(void) __attribute__ ((interrupt_handler));
void customHandler(void) __attribute__ ((fast_interrupt));
```

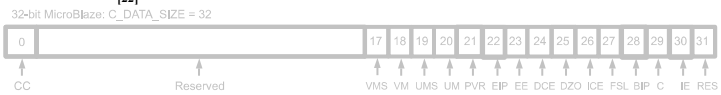
  - Set custom routine as ***ISR*** at run time

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

  - Set MSR (Machine Status Register)

```
#include "xparameters.h"  // hardware config constants
#include "mb_interface.h" // microblaze processor API
void microblaze_enable_interrupts(void);  // MSR[IE] = 1
void microblaze_disable_interrupts(void); // MSR[IE] = 0
```

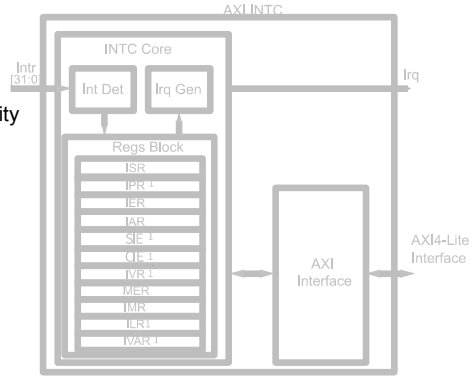

- ***MSR*** **(Machine Status Register)**
  - $MSR_{[30]}$: **1** → Interrupt enable
  - $MSR_{[28]}$: **0** → No break in progress
  - $MSR_{[22]}$: **0** → No hardware exception in progress



- **Interrupt Response Flow**

```
if (MSR & 0x20A == 2) {           // if({MSR[22], MSR[28], MSR[30]} == 3'b001)
  MSR = MSR & 0xFFFFFFFD;         // MSR[30] = 0, disable interrupt
  r14 = PC + 4;                   // save the breakpoints to r14
  if (C_USE_INTERRUPT == 2) {     // fast mode:
    PC = Interrupt_Address;       // ISR address is provided by interrupt_address input
    Interrupt_Ack = 01;           // output Interrupt_Ack
  }
  else PC = C_BASE_VECTORS + 0x10; // normal mode: call TOP ISR of user interrupt
}
```

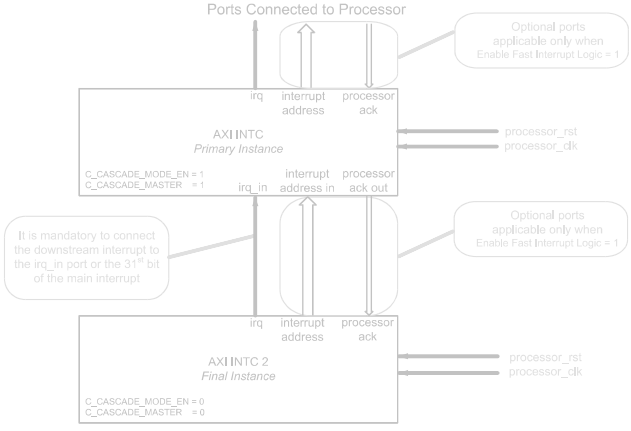

- **AXI INTC Features**
  - Register access through the AXI4-Lite interface
  - Supports 1~32 ***Intr*** Inputs
  - Configurable
    - Interrupt Priority
    - Interrupt Enable
    - Interrupt Sensitivity
      - Edge
      - Level
    - Interrupt Mode
      - Normal
      - Fast
      - Cascade



- **I/O Signals**

Name	I/O	Initial	Description
irq	O	1	Interrupt request output
$interrupt\_address_{[w-1:0]}$	O	0	Interrupt address output (Fast Mode)
$processor\_ack\_out_{[1:0]}$	O	0	Cascade mode as well as fast mode
$intr_{[N-1:0]}$	I	-	Interrupt input, $intr_{[0]}$ (the highest priority)
$interrupt\_address\_in_{[w-1:0]}$	I	-	Cascade mode as well as fast mode
irq_in	I	-	Cascade mode
$processor\_ack_{[1:0]}$	I	-	Interrupt acknowledge input (Fast Mode) 00 - No interrupt received 01 - Set when CPU branches to ISR 10 - Set when CPU returns from ISR 11 - Set when CPU enables interrupts
processor_clk	I	-	Processor clock
processor_rst	I	-	Processor reset, active-high
s_axi_aresetn	I	-	AXI Reset, active-low
s_axi_aclk	I	-	AXI Clock
s_axi_*	I	-	AXI4, AXI4-Lite, AXI-Stream signals

- **Cascade and Fast Interrupt mode**



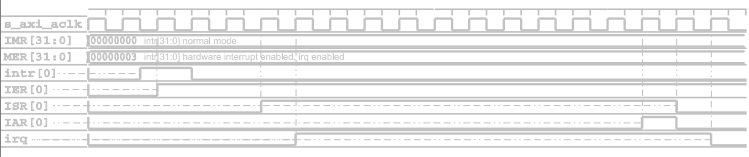
- **Register Address Mapping**

Address Offset	Name	Description
0x0	ISR	Interrupt Status Register, R/W
0x8	IER	Interrupt Enable Register, R/W
0xC	IAR	Interrupt Acknowledge Register, TOW (Toggle on Write)
0x1C	MER	Master Enable Register, R/W
0x20	IMR	Interrupt Mode Register, R/W
0x100 ~ 0x17C	IVAR	Interrupt Vector Address Register (Fast Mode, Optional)
0x200 ~ 0x2FC	IVEAR	Interrupt Vector Extended Address Register (Fast Mode)

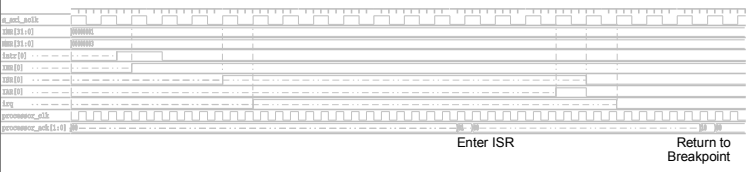
  - $ISR_{[i]}$ = **1 / 0** → Active / Not Active interrupt signal on $intr_{[i]}$ input
  - $IER_{[i]}$ = **1 / 0** → Unmask / Mask $ISR_{[i]}$
  - $IAR_{[i]}$ = **1** → Clear $IAR_{[i]}$ and $ISR_{[i]}$ for ***irq*** (generated by $intr_{[i]}$)
    - Toggle on Write: The bit written to 1 will be cleared to 0 in the next clock cy cle
    - Fast Interrupt: $IAR_{[i]}$ is cleared by signalling from $processor\_ack_{[1:0]}$
  - $MER_{[1:0]}$ = **{1 / 0, 1 / 0}** → Enable / Disable HW $intr_{[31:0]}$ inputs, ***irq*** output
  - $IMR_{[i]}$ = **1 / 0** → $intr_{[i]}$ in Fast / Normal mode

2.2 Interrupt Controller (AXI INTC) for MicroBlaze

- **Normal Interrput Timing (Input - Rising Edge Sensitive)**

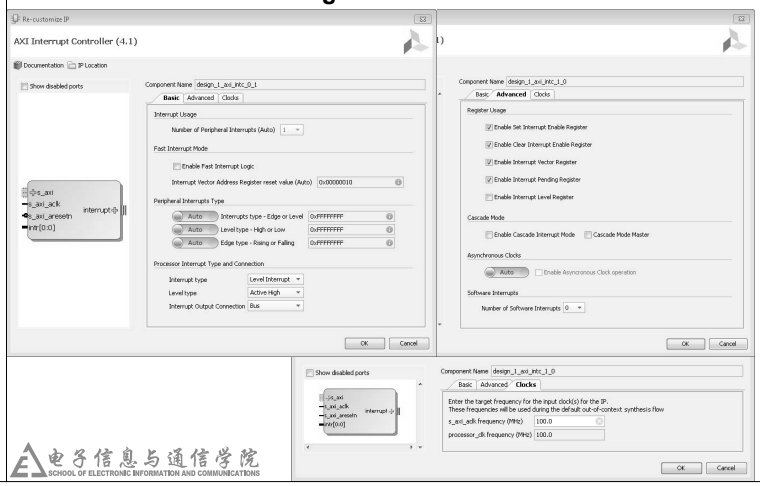


- **Fast Interrput Timing (Input - Rising Edge Sensitive)**



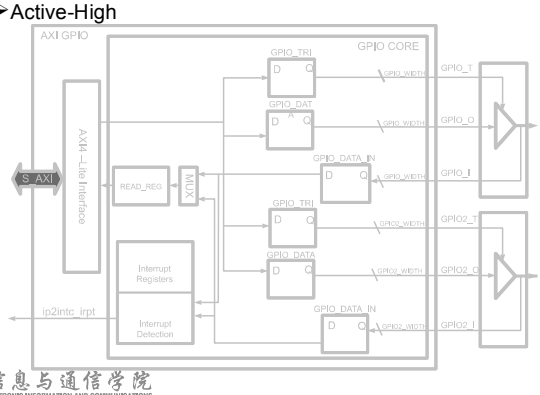
2.2 Interrupt Controller (AXI INTC) for MicroBlaze

- **GUI for AXI INTC Configuration**



3 Design of IO Interface in Interrupt Mode

- **AXI GPIO data transmission in interrupt mode**
  - Data incoming will output an interrupt request: ***ip2intc_irpt***
    - Level sensitive signal
    - Active-High



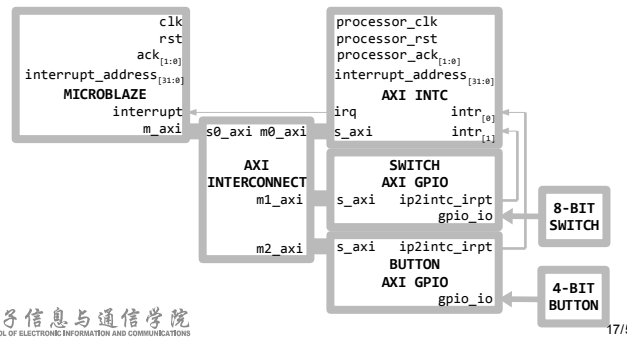
3.1 AXI GPIO Interrupt Interface

- **Registers for AXI GPIO Interrupt Interface**
  - ***GIER*** (Global Interrupt Enable Register, 32-bit, Offset: **0x11C**)
    - $GIER_{[31]}$**: 1 / 0** → Enabled / Disabled
  - ***IPIER*** (IP Interrupt Enable Register, 32-bit, Offset: **0x128**)
    - $IPIER_{[0]}$**: 1 / 0** → Channel 0 Interrupt Enabled / Disabled
    - $IPIER_{[1]}$**: 1 / 0** → Channel 1 Interrupt Enabled / Disabled
  - ***IPISR*** (IP Interrupt Status Register, 32-bit, Offset: **0x120**)
    - $IPISR_{[0]}$**: 1** → Channel 0 input interrupt
    - $IPISR_{[1]}$**: 1** → Channel 1 input interrupt
    - Toggle on Write

      Writing **1** to a bit position within the register causes the corresponding bit to toggle

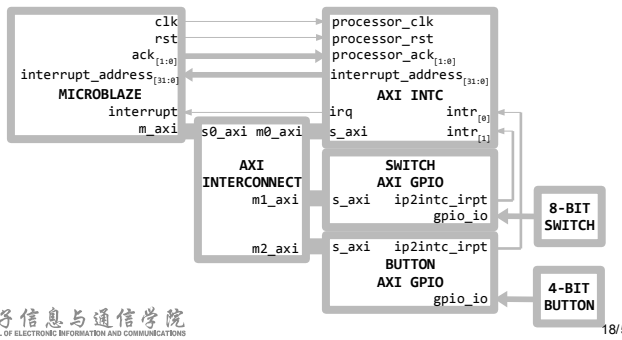
3.1 AXI GPIO Interrupt Interface

- **Ex.1:** Read 8 switches and 4 buttons code in interrupt mode
  - Define 2 GPIO cores for connecting Switches and Buttons
  - ***switch_GPIO.ip2intc_irpt*** → $INTC.intr_{[1]}$
  - ***btn_GPIO.ip2intc_irpt*** → $INTC.intr_{[0]}$
  - Normal interrupt mode: ***INTC.irq*** → MicroBlaze



3.1 AXI GPIO Interrupt Interface

- **Ex.1:** Read 8 switches and 4 buttons code in interrupt mode
  - Fast interrupt mode
    - MicroBlaze → $INTC.\{processor\_clk, processor\_rst, processor\_ack_{[1:0]}\}$
    - $INTC.irq$ → MicroBlaze
    - $INTC.interrupt\_address_{[31:0]}$ → MicroBlaze



- **Ex.1:** Read 8 switches and 4 buttons code in interrupt mode

```
// interrupt_gpio.h
#include "xparamters.h"
#define btn_GPIO_BASE  0x40000000
#define sw_GPIO_BASE   0x40100000
#define INTC_BASE      0x41200000
#define btn_GPIO_DATA  (btn_GPIO_BASE + 0x0)   //btn_GPIO DATA Register
#define btn_GPIO_TRI   (btn_GPIO_BASE + 0x4)   //btn_GPIO Tri-State Control Register
#define btn_GPIO_GIER  (btn_GPIO_BASE + 0x11C) //btn_GPIO Global Interrupt Enable Register
#define btn_GPIO_IPIER (btn_GPIO_BASE + 0x128) //btn_GPIO IP Interrupt Enable Register
#define btn_GPIO_IPISR (btn_GPIO_BASE + 0x120) //btn_GPIO IP Interrupt Status Register
#define sw_GPIO_DATA   (sw_GPIO_BASE + 0x0)    //sw_GPIO DATA Register
#define sw_GPIO_TRI    (sw_GPIO_BASE + 0x4)    //sw_GPIO Tri-State Control Register
#define sw_GPIO_GIER   (sw_GPIO_BASE + 0x11C)  //sw_GPIO Global Interrupt Enable Register
#define sw_GPIO_IPIER  (sw_GPIO_BASE + 0x128)  //sw_GPIO IP Interrupt Enable Register
#define sw_GPIO_IPISR  (sw_GPIO_BASE + 0x120)  //sw_GPIO IP Interrupt Status Register
#define INTC_ISR       (INTC_BASE + 0x0)       //INTC Interrupt Status Register
#define INTC_IER       (INTC_BASE + 0x8)       //INTC Interrupt Enable Register
#define INTC_IAR       (INTC_BASE + 0xC)       //INTC Interrupt Acknowledge Register
#define INTC_MER       (INTC_BASE + 0x1C)      //INTC Master Enable Register
#define INTC_IMR       (INTC_BASE + 0x20)      //INTC Interrupt Mode Register
#define INTC_IVAR      (INTC_BASE + 0x100)     //INTC Interrupt Vector Address Register
```

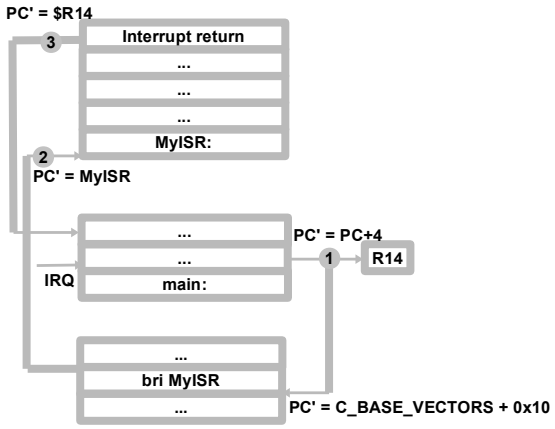

- **Ex.1:** Read 8 switches and 4 buttons code in interrupt mode

```
// normal mode
#include "sleep.h"
#include "interrupt_gpio.h"
void main(void) {
  Xil_Out32(btn_GPIO_TRI, 0xF);         //set btn_GPIO IO[3:0] as input
  Xil_Out32(btn_GPIO_IPISR, 0x1); //clear btn_GPIO Interrupt Status Register (TOW)
  Xil_Out32(btn_GPIO_GIER, 0x80000000); //enable btn_GPIO global interrupt
  Xil_Out32(btn_GPIO_IPIER, 1);         //enable btn_GPIO channel 0 interrupt
  Xil_Out32(sw_GPIO_TRI, 0xFF);         //set sw_GPIO IO[7:0] as input
  Xil_Out32(sw_GPIO_IPISR, 1);    //clear sw_GPIO Interrupt Status Register (TOW)
  Xil_Out32(sw_GPIO_GIER, 0x80000000);  //enable sw_GPIO global interrupt
  Xil_Out32(sw_GPIO_IPIER, 1);          //enable sw_GPIO channel 0 interrupt
  Xil_Out32(INTC_IAR, 0xFFFFFFFF);      //set INTC_IAR, to clear INTC_ISR
  Xil_Out32(INTC_IER, 0x3);             //enable INTC_ISR[1:0] and intr[1:0] input
  Xil_Out32(INTC_MER, 0x3);             //enable INTC interrupt input and irq output
  Xil_Out32(INTC_IMR, 0x0);             //INTC normal interrupt mode
  microblaze_enable_interrupts();       //enable microblaze interrupt
  while(1);
}
```


- **Ex.1:** Read 8 switches and 4 buttons code in interrupt mode

```
void MyISR(void) __attribute__ ((interrupt_handler)) {
  u32 status = Xil_In32(INTC_ISR);//read INTC Interrupt Status Register
  u8 code = 0;
  if (status & 2) {               //switch GPIO interrupt
    code = Xil_In8(sw_GPIO_DATA); //read sw_GPIO DATA Register
    Xil_Out32(sw_GPIO_IPISR, 1);  //clear sw_GPIO Interrupt Status Register (TOW)
    xil_printf("Switch Code = 0x%X\n", code);
  }
  else if (status & 1) {              //btn_GPIO interrupt
    code = Xil_In8(btn_GPIO_DATA) & 0xF; //read btn_GPIO DATA Register
    Xil_Out32(btn_GPIO_IPIER, 0x0);     //disable btn_GPIO interrupt
    usleep(50000);                      //delay 50ms
    Xil_Out32(btn_GPIO_IPISR, 0x1); //clear btn_GPIO Interrupt Status Register (TOW)
    Xil_Out32(btn_GPIO_IPIER, 0x1);     //enable btn_GPIO interrupt
    xil_printf("Button Code = 0x%X\n", code);
  }
  Xil_Out32(INTC_IAR, status); //set INTC_IAR (TOW), to clear INTC_ISR and INTC_IAR
}
```


- **Ex.1:** Read 8 switches and 4 buttons code in interrupt mode



- **Ex.1:** Read 8 switches and 4 buttons code in interrupt mode

```
// fast mode
#include "sleep.h"
#include "interrupt_gpio.h"
void MyISR0(void) __attribute__ ((fast_handler));
void MyISR1(void) __attribute__ ((fast_handler));
void main(void) {
  Xil_Out32(btn_GPIO_TRI, 0xF);         //set btn_GPIO IO[3:0] as input
  Xil_Out32(btn_GPIO_IPISR, 0x1); //clear btn_GPIO Interrupt Status Register (TOW)
  Xil_Out32(btn_GPIO_GIER, 0x80000000); //enable btn_GPIO global interrupt
  Xil_Out32(btn_GPIO_IPIER, 0x1);       //enable btn_GPIO channel 0 interrupt
  Xil_Out32(sw_GPIO_TRI, 0xFF);         //set sw_GPIO IO[7:0] as input
  Xil_Out32(sw_GPIO_IPISR, 1);    //clear sw_GPIO Interrupt Status Register (TOW)
  Xil_Out32(sw_GPIO_GIER, 0x80000000);  //enable sw_GPIO global interrupt
  Xil_Out32(sw_GPIO_IPIER, 0x1);        //enable sw_GPIO channel 0 interrupt
  Xil_Out32(INTC_IAR, 0xFFFFFFFF);      //set INTC_IAR, to clear INTC_ISR
  Xil_Out32(INTC_IER, 0x3);             //enable INTC_ISR[1:0] and intr[1:0] input
  Xil_Out32(INTC_MER, 0x3);             //enable INTC interrupt input and irq output
  Xil_Out32(INTC_IMR, 0x3);             //set INTC_IMR[1:0] to fast interrupt mode
  Xil_Out32(INTC_IVAR, (int)MyISR0);    //set MyISR0 Address to INTC_IVAR0
  Xil_Out32(INTC_IVAR + 4, (int)MyISR1);//set MyISR1 Address to INTC_IVAR1
  microblaze_enable_interrupts();       //enable microblaze interrupt
  while(1);
}
```

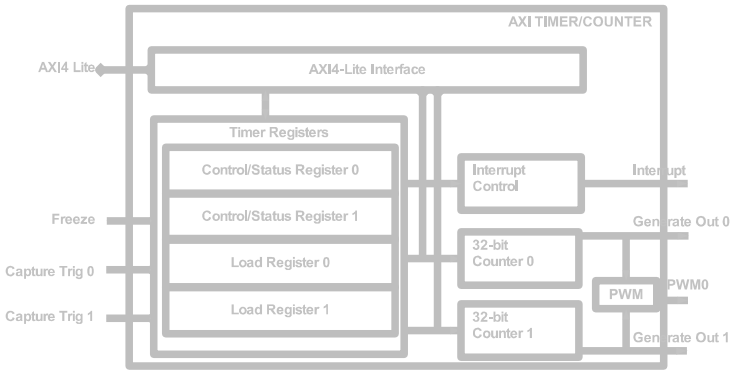

- **Ex.1:** Read 8 switches and 4 buttons code in interrupt mode

```
void MyISR0(void) __attribute__ ((fast_handler)) {
  u32 status = Xil_In32(INTC_ISR);       //read INTC Interrupt Status Register
  u8 code = 0;
  if (status & 1) {                      //btn_GPIO interrupt
    code = Xil_In8(btn_GPIO_DATA) & 0xF; //read btn_GPIO DATA Register
    Xil_Out32(btn_GPIO_IPIER, 0);        //disable btn_GPIO interrupt
    usleep(50000);                       //delay 50ms
    Xil_Out32(btn_GPIO_IPISR, 1); //clear btn_GPIO Interrupt Status Register (TOW)
    Xil_Out32(btn_GPIO_IPIER, 1);        //enable btn_GPIO interrupt
    xil_printf("Button Code = 0x%X\n", code);
  }
  Xil_Out32(INTC_IAR, status);           //set INTC_IAR, to clear INTC_ISR
}
void MyISR1(void) __attribute__ ((fast_handler)) {
  u32 status = Xil_In32(INTC_ISR);       //read INTC Interrupt Status Register
  u8 code = 0;
  if (status & 2) {                      //switch GPIO interrupt
    code = Xil_In8(sw_GPIO_DATA);        //read sw_GPIO DATA Register
    Xil_Out32(sw_GPIO_IPISR, 1);  //clear sw_GPIO Interrupt Status Register (TOW)
    xil_printf("Switch Code = 0x%X\n", code);
  }
  Xil_Out32(INTC_IAR, status);           //set INTC_IAR, to clear INTC_ISR
}
```

- **AXI Timer**
  - 2 counters with interrupt
    - Generate mode
      - If counter roll over, the pulse signal is generated, generating interrupt
    - Capture mode
      - If capture event is valid, the count value is latched, generating interrupt
  - 1 Pulse Width Modulation (PWM) output a pulse signal on $pwm0$
    - The pulse period / width be set by $Counter0$ / $Counter1$
  - Supports Cascade mode
    - $\{ TCR1_{[31:0]}, TCR0_{[31:0]} \}$

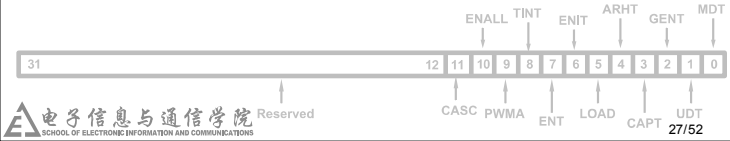
Register	Address Offset	Access	Default	Description
TCSR0, TCSR1	0x00, 0x10	R/W	0x0	Control/Status Register
TLR0, TLR1	0x04, 0x14	R/W	0x0	Load Register
TCR0, TCR1	0x08, 0x18	R	0x0	Counter Register

- **AXI Timer**



- $TCSR$ (2 Control/Status Registers, 32-bit, Reset Value $0x0$)

Bit	Name	Description
0	MDT	0 / 1: Generate / Capture mode
1	UDT	0 / 1: Up / Down count
2	GENT	0 / 1: Disables / Enables generate signal (Generate mode)
3	CAPT	0 / 1: Disables / Enables capture event (Capture mode)
4	ARHT	0 / 1: Hold / Reload TLR (Generate mode); Hold / Overwrite TLR (Capture mode)
5	LOAD	0 / 1: No-load / Loads counter with value in TLR
6	ENIT	0 / 1: Disables / Enables interrupt signal
7	ENT	0 / 1: Disables / Enables (Generate mode); Halts / Runs (Capture mode)
8	TINT	0 / 1: No-interrupt / Interrupt has occurred, Toggle-On-Write 1, clear
9	PWMA	0 / 1: Disables / Enables pulse width modulation
10	ENALL	0 / 1: Disables / Enable all counters
11	CASC	0 / 1: Disables / Enables cascaded operation



- TLR (2 Load Registers, 32-bit, Reset Value $0x0$)
  - Generate mode
    - Load the initial value from TLR before the timer starts ticking
  - Capture mode
    - If capture event is valid, the value of counter is written to TLR
- TCR (2 Counter Registers, 32-bit, Reset Value $0x0$)
  - Cascade Mode
    - $TCR0$: for lower 32-bit of the 64-bit counter
    - $TCR1$: for upper 32-bit of the 64-bit counter

- **I/O Signals**

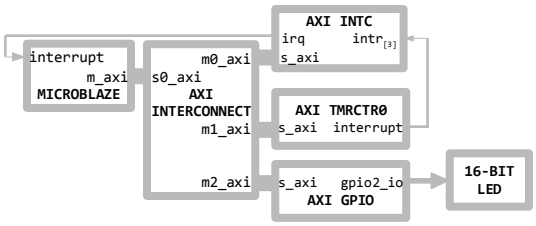
Name	I/O	Initial	Description
generateout0	O	0	Counter0 roll over, output pulse (1 clock period in width)
generateout1	O	0	Counter1 roll over, output pulse (1 clock period in width)
pwm0	O	0	Pulse width modulation output
interrupt	O	0	Capture event has occurred (Capture Mode) Counter has rolled over (Generate Mode) 0: No interrupt has occurred 1: Interrupt has occurred
capturetrig0	I	-	Counter0 capture trigger input
capturetrig1	I	-	Counter1 capture trigger input (not used in cascade mode)
freeze	I	-	Freeze input for halting counters during software debug
s_axi_aresetn	I	N/A	AXI reset, active-low
s_axi_aclk	I	N/A	AXI clock
s_axi_*	I/O	N/A	AXI4, AXI4-lite, AXI-stream signals

- **Programming Sequence**
  - Generate Mode
    - $TCSR_{[LOAD]} \leftarrow 1$, TLR is loaded
    - $TCSR_{[LOAD]} \leftarrow 0$, and $TCSR_{[ENT]} \leftarrow 1$
      - The timer starts ticking
    - When the timer rolls over, if ($TCSR_{[GENT]}$)
      - The $generateout$ signal is enabled, output pulse is generated
      - If interrupt is enabled, generate interrupt
      - If ($TCSR_{[ARHT]}$), auto reload, generating a repetitive pulse
      - If ($\sim TCSR_{[ARHT]}$), does not reload, generating a one-shot pulse
    - If ($TCSR_{[UDT]}$), count down
      - $TIMING\_INTERVAL = (TLR + 2) \times AXI\_CLOCK\_PERIOD$
    - If ($\sim TCSR_{[UDT]}$), count up
      - $TIMING\_INTERVAL = (MAX\_COUNT - TLR + 2) \times AXI\_CLOCK\_PERIOD$

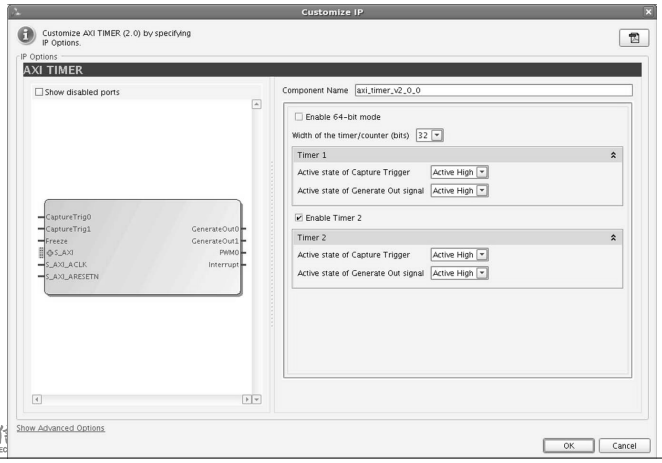
- **Programming Sequence**
  - Capture Mode
    - The capture signal can be configured to be low-true or high-true
    - If ($TCSR_{[UDT]}$), count down, else count up
    - If capture event is valid, the counter value is written to ***TLR***
      - If interrupt is enabled, generate interrupt
      - If ($\sim TCSR_{[ARHT]}$), ***TLR*** holds the capture value until it is read
      - If ($TCSR_{[ARHT]}$), The capture value is always written to ***TLR***

- **Programming Sequence**
  - Pulse Width Modulation Mode
    - If ($(\sim TCSR_{[MDT]})\ \&\&\ TCSR_{[PWMA]}\ \&\&\ TCSR_{[GENT]}$)
      - i.e., Generate Mode & PWM be enabled
      - The ***generateout*** signals must be active high-level
      - ***pwm0*** can output the PWM signal
    - If ($\sim TCSR_{[UDT]}$), i.e., count up
      - $PWM\_PERIOD = (MAX\_COUNT - TLR0 + 2) \times AXI\_CLOCK\_PERIOD$
      - $PWM\_HIGH\_TIME = (MAX\_COUNT - TLR1 + 2) \times AXI\_CLOCK\_PERIOD$
    - If ($TCSR_{[UDT]}$), i.e., count down
      - $PWM\_PERIOD = (TLR0 + 2) \times AXI\_CLOCK\_PERIOD$
      - $PWM\_HIGH\_TIME = (TLR1 + 2) \times AXI\_CLOCK\_PERIOD$

- **Ex.2:** 16 LEDs, cyclic lighting, 1 second interval
  - AXI INTC: $intr_{[3]}$
  - AXI ***TmrCtr0***: Generate mode
  - AXI GPIO: ***GPIO2_IO***
  - 100 MHz AXI clock frequency



- **Ex.2:** 16 LEDs, cyclic lighting, 1 second interval



- **Ex.2:** 16 LEDs, cyclic lighting, 1 second interval

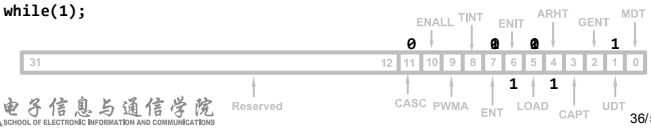
```
#include "xil_io.h"  // Use I/O functions, not API functions
#include "stdio.h"

#define INTC_BASE   0x41200000
#define GPIO_BASE   0x40000000
#define TIMER_BASE  0x41C00000

#define INTC_ISR    (INTC_BASE + 0x0)
#define INTC_IER    (INTC_BASE + 0x8)
#define INTC_IAR    (INTC_BASE + 0xC)
#define INTC_MER    (INTC_BASE + 0x1C)
#define GPIO2_DATA  (GPIO_BASE + 0x8)
#define GPIO2_TRI   (GPIO_BASE + 0xC)
#define TIMER_TCSR0 (TIMER_BASE + 0)
#define TIMER_TLR0  (TIMER_BASE + 4)
#define TIMER_TCR0  (TIMER_BASE + 8)
```


- **Ex.2:** 16 LEDs, cyclic lighting, 1 second interval

```
void main() {
  Xil_Out32(GPIO2_TRI, 0x0);   // set GPIO2_TRI, GPIO2_IO as output
  Xil_Out32(TIMER_TCSR0, 0x0); // reset TIMER_TCSR0
  Xil_Out32(TIMER_TLR0, 99999998); // 100MHz clock, dec count to 0 in 1sec.
  // set TCSR[LOAD]=1, so that counter loads the value from TLR
  Xil_Out32(TIMER_TCSR0, Xil_In32(TIMER_TCSR0) | 0x20);
  // TCSR[LOAD]=0, TCSR[ENT]=TCSR[ENIT]=TCSR[ARHT]=TCSR[UDT]=1
  // start counter, enable timer, intr enable, auto load, count down
  Xil_Out32(TIMER_TCSR0, Xil_In32(TIMER_TCSR0) & 0x7DF | 0xD2);
  Xil_Out32(INTC_IER, 0x8);    // set INTC_IER, enable INTC ISR[3]
  Xil_Out32(INTC_MER, 0x3);    // set INTC_MER, enable INTC irq and intr
  microblaze_enable_interrupts();
  while(1);
}
```



- **Ex.2:** 16 LEDs, cyclic lighting, 1 second interval

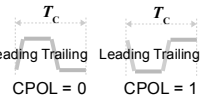
```
u32 ledBits = 0;

// Top interrupt handler
void MyISR(void) __attribute__ ((interrupt_handler)) {
  int status = Xil_In32(INTC_ISR);                  //read INTC status reg
  if (status & 0x8) {                               //if intr[3]
    Xil_Out32(GPIO2_DATA, 1 << ledBits);            //set GPIO2_DATA reg
    Xil_Out32(TIMER_TCSR0, Xil_In32(TIMER_TCSR0)); //clear TCSR[TINT] (TOW)
    Xil_Out32(INTC_IAR, Xil_In32(INTC_ISR));        //clear INTC_ISR
    ledBits = (ledBits < 15) ? ledBits + 1 : 0;
  }
}
```

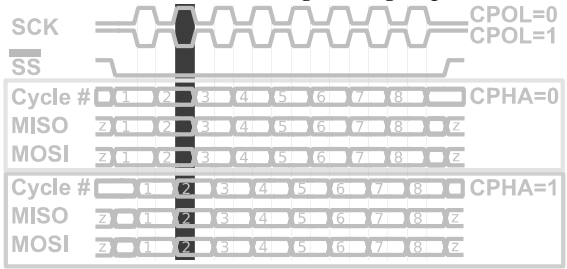

- **SPI (Serial Peripheral Interface)**
  - A Synchronized Serial Internal Bus between ICs
  - Full duplex mode
  - Master-slave architecture, with a single master
  - Four-wire serial bus
    - SCLK: Serial Clock (output from master)
    - MOSI: Master Output Slave Input (data output from master)
    - MISO: Master Input Slave Output (data output from slave)
    - /SS: Slave Select (Low level active, sent by master)

- **DSPI (Dual SPI), QSPI (Quad SPI)**

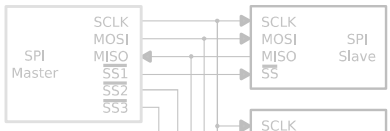
- **Clock polarity (CPOL = 0/1)**
  - Leading edge: ***Posedge/Negedge***
  - Trailing edge: ***Negedge/Posedge***



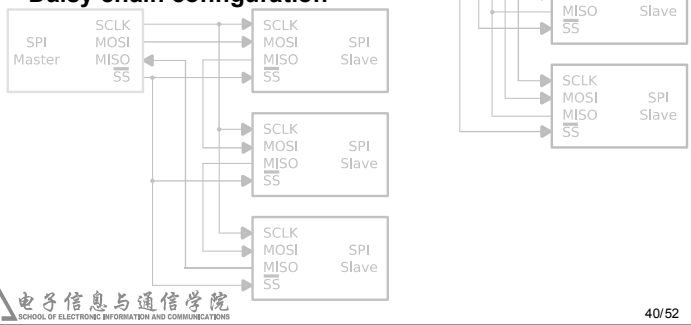
- **Clock phase (CPHA = 0/1)**
  - Out-side: Set data on the ***Trailing/Leading*** edge
  - In-side: Get data on the ***Leading/Trailing*** edge



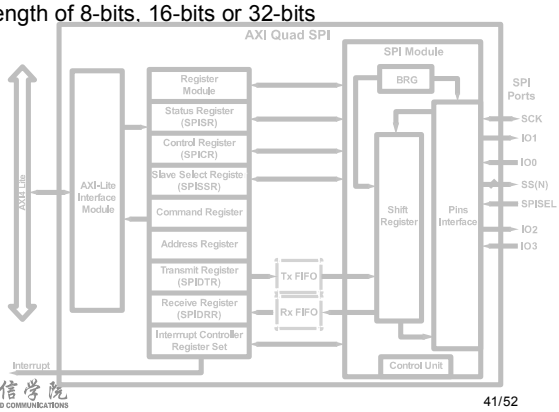
- **Independent slave configuration**



- **Daisy chain configuration**



- **AXI Quad SPI (Serial Peripheral Interface)**
  - Master and slave SPI modes
  - MSB / LSB first transactions
  - Transfer length of 8-bits, 16-bits or 32-bits



- **AXI Quad SPI IP Core I/O Signals**

Name	I/O	Initial	Description
ip2intc_irpt	O	-	Interrupt control signal from SPI
sck_o	O	0	SPI bus clock output
sck_t	O	1	3-state enable for SPI bus clock, Active low
$ss\_o_{[(N-1):0]}$	O	1	Output one-hot encoded
ss_t	O	1	3-state enable for slave select, Active low
$io\_o_{[3:0]}$	O	-	MOSI output
$io\_t_{[3:0]}$	O	1	3-state enable master output slave input, Active low
sck_i	I	-	SPI bus clock input
$io\_i_{[3:0]}$	I	-	MOSI input
$ss\_i_{[(N-1):0]}$	I	-	Input one-hot encoded
spisel	I	1	Slave mode: Slave select active low input Master mode: Must be set to 1
s_axi_*	-	-	AXI4, AXI4-Lite, AXI-Stream signals

3.3 AXI Quad SPI Interrupt Interface

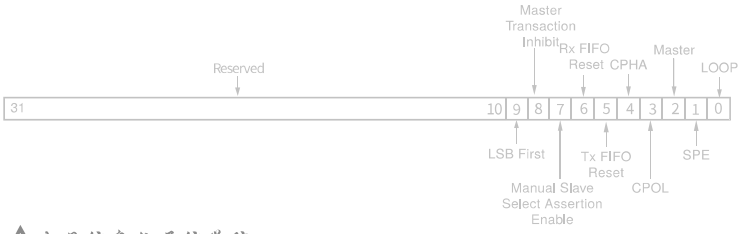
- **AXI Quad SPI IP Core Registers**

Register	Offset	Access	Default	Description
SRR	0x40	W	N/A	0xA: Reset SPI IP Core
SPICR	0x60	R/W	0x180	SPI Control Register
SPISR	0x64	R	0x25	SPI Status Register
SPIDTR	0x68	W	0x0	Data Transmit Register (8/16/32-bit)
SPIDRR	0x6C	R	NA	Data Receive Register (8/16/32-bit)
SPISSR	0x70	R/W	No SS	Slave Select Register
...	...	...	...	...
DGIER	0x1C	R/W	0x0	DGIER[31] = 1, Enable Global Interrupt
IPISR	0x20	R/TOW	0x0	Interrupt Status Register
IPIER	0x28	R/W	0x0	Interrupt Enable Register

3.3 AXI Quad SPI Interrupt Interface

- ***SPICR*** (SPI Control Register, Offset: **0x60**)

Bit	Name	Default	Description
0	LOOP	0	0 / 1: Normal operation / Loopback mode (Only Master)
1	SPE	0	0 / 1: Disabled / Enable SPI system
2	Master	0	0 / 1: Slave / Master configuration
3	CPOL	0	0 / 1: Active High / Low clock, SCK idles low / high
4	CPHA	0	Clock Phase, transfer formats
...	...	...	...



3.3 AXI Quad SPI Interrupt Interface

- ***SPISR*** (SPI Status Register, Offset: **0x64**)

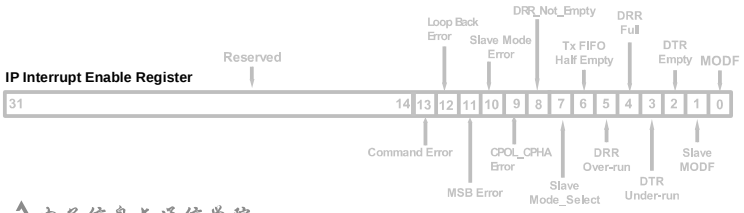
Bit	Name	Default	Description
0	Rx_Empty	1	1: receive FIFO is empty
1	Rx_Full	0	1: the receive FIFO is full
2	Tx_Empty	1	1: the transmit FIFO is empty
3	Tx_Full	0	1: the transmit FIFO is full
4	MODF	0	0 / 1: No error / Error condition detected
...	...	...	...

- ***SPISSR*** (SPI Slave Select Register, Offset: **0x70**)

Bit	Name	Default	Description
N-1:0	Selected Slave	1	Active Low, slave select vector of length N-bits
31:N	Reserved	N/A	Reserved

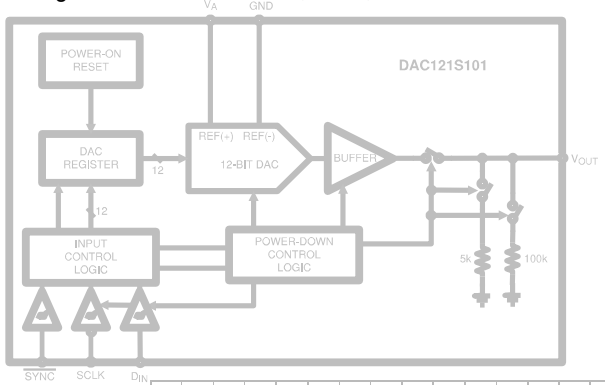
3.3 AXI Quad SPI Interrupt Interface

- ***DGIER*** (Device Global Interrupt Enable, Offset: **0x1C**)
  - ***DGIER***$_{[31]}$: **1** = Global Interrupt Enable (GIE)
- ***IPISR*** (IP Interrupt Status Register, Offset: **0x20**)
  - Toggle on Write
    - Writing a **1** to a bit position within the register causes the corresponding bit to toggle
- ***IPIER*** (IP Interrupt Enable Register, Offset: **0x28**)
  - Interrupt Source Enable



3.3 AXI Quad SPI Interrupt Interface

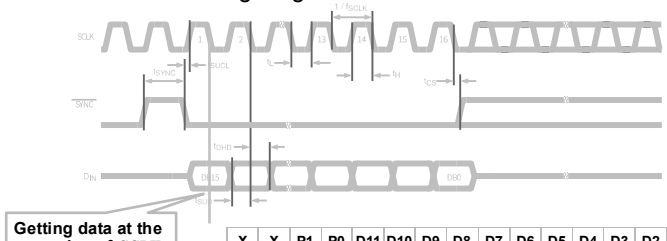
- **Ex.3:** Sawtooth signal generator
  - Using DAC121s101, AXI SPI, INTC, MicroBlaze



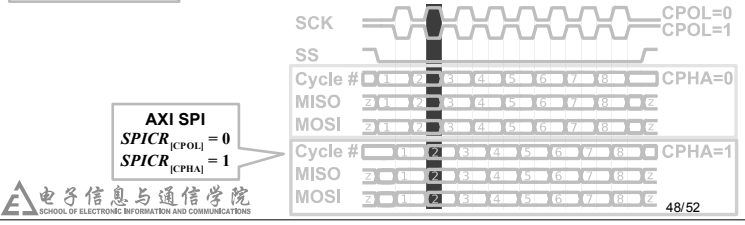
X	X	P1	P0	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

3.3 AXI Quad SPI Interrupt Interface

- **Ex.3:** Sawtooth signal generator

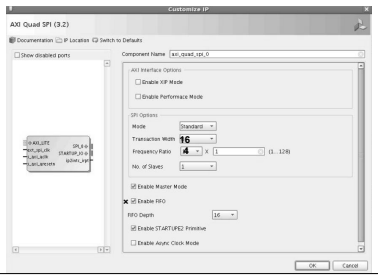


X	X	P1	P0	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0



3.3 AXI Quad SPI Interrupt Interface

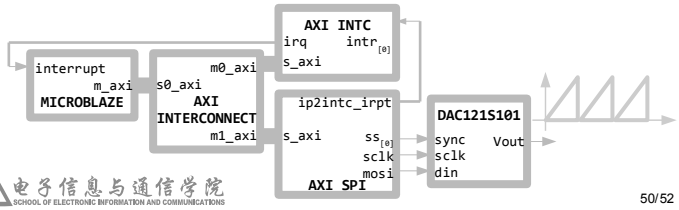
- **Ex.3:** Sawtooth signal generator
  - DAC121S101 (Slave)
    - Resolution: 12 Bits
    - Max clock frequency: 30MHz
  - SPI Design Parameters (Master)
    - **C_NUM_TRANSFER_BITS = 16**
    - **C_SCK_RATIO = 4**
      - AXI clock frequency 100MHz
      - DAC *SCK* 25MHz
    - **C_NUM_SS_BITS = 1**



3.3 AXI Quad SPI Interrupt Interface

- **Ex.3:** Sawtooth signal generator

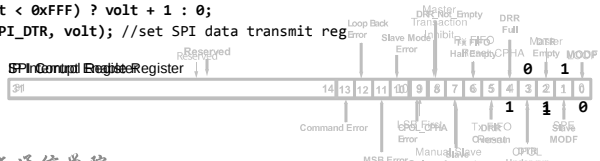
```
#include "xli_io.h"  // Use I/O functions, not API functions
#define INTC_BASE 0x41200000
#define SPI_BASE  0x44A00000
#define INTC_ISR  (INTC_BASE + 0x0)
#define INTC_IER  (INTC_BASE + 0x8)
#define INTC_IAR  (INTC_BASE + 0xC)
#define INTC_MER  (INTC_BASE + 0x1C)
#define SPI_DGIER (SPI_BASE + 0x1C) // SPI Device Global Interrupt Enable Register
#define SPI_IISR  (SPI_BASE + 0x20) // SPI IP Interrupt Status Register
#define SPI_IIER  (SPI_BASE + 0x28) // SPI IP Interrupt Enable Register
#define SPI_CR    (SPI_BASE + 0x60) // SPI Control Register
#define SPI_DTR   (SPI_BASE + 0x68) // SPI Data Transmit Register
#define SPI_SSR   (SPI_BASE + 0x70) // SPI Slave Select Register
u16 volt = 0;
```



3.3 AXI Quad SPI Interrupt Interface

- **Ex.3:** Sawtooth signal generator

```
void main() {
  Xil_Out32(SPI_CR, 0x16);          // CPHA=1, CPOL=0, master, SPI enable
  Xil_Out32(SPI_SSR, 0xFFFFFFFE);   // set slave select register
  Xil_Out32(SPI_IIER, 0x4);         // enable DTR empty intr
  Xil_Out32(SPI_DGIER, 0x80000000); // DGIER[31] = 1, enable SPI global intr
  Xil_Out32(INTC_IER, 0x1);         // enable INTC ISR[0]
  Xil_Out32(INTC_MER, 0x3);         // enable INTC irq and intr
  microblaze_enable_interrupts();
  Xil_Out16(SPI_DTR, volt);         //set SPI data transmit reg, start
  while(1);
}
void MyISR(void) __attribute__ ((interrupt_handler)) {
  Xil_Out32(SPI_IISR, Xil_In32(SPI_IISR)); //clear SPI status reg (TOW)
  Xil_Out32(INTC_IAR, Xil_In32(INTC_ISR)); //clear INTC ISR and IAR
  volt = (volt < 0xFFF) ? volt + 1 : 0;
  Xil_Out16(SPI_DTR, volt); //set SPI data transmit reg
}
```



Homework

- P368~369
  - 19
  - 23
  - 24
  - 29