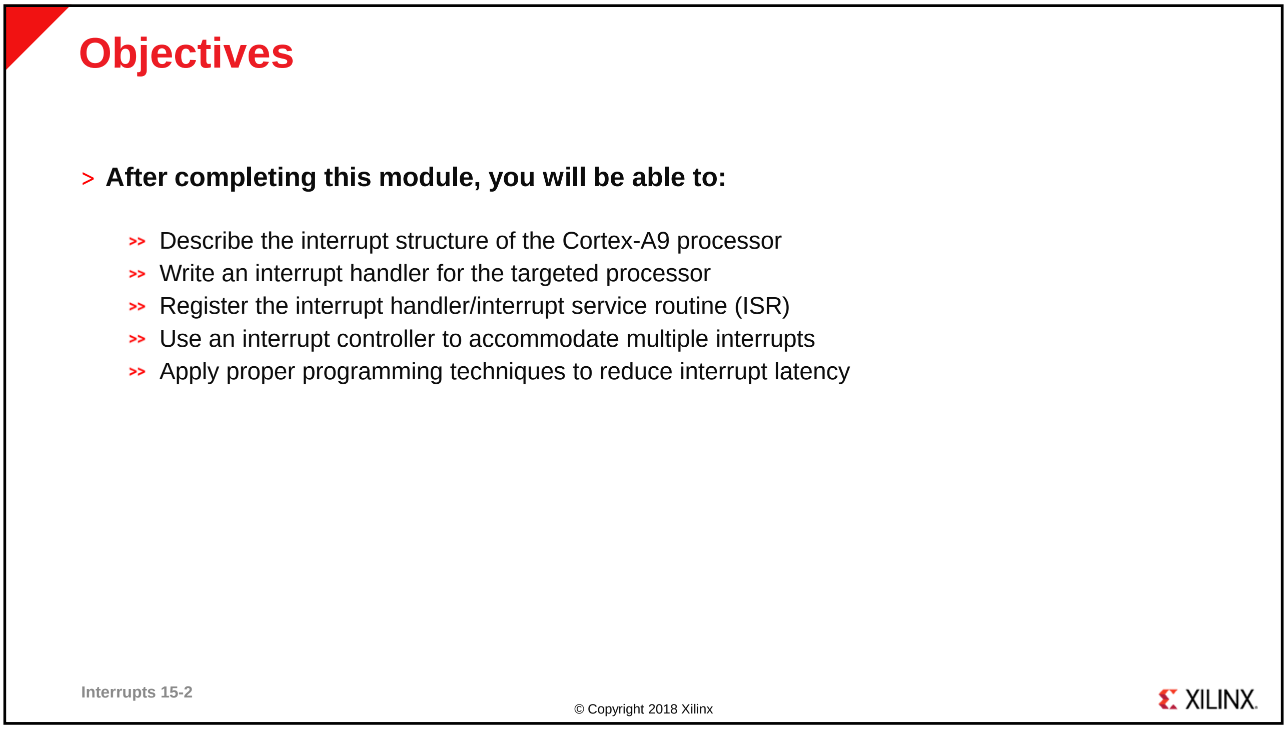
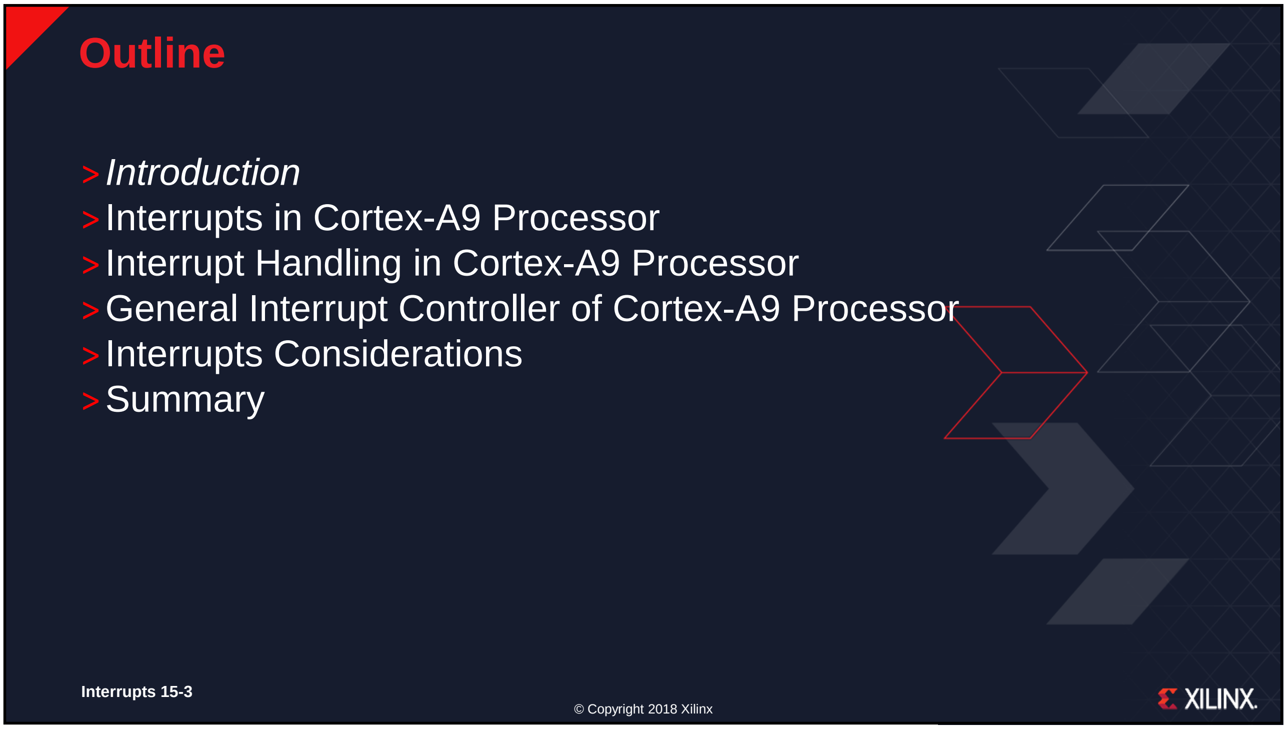
Interrupts



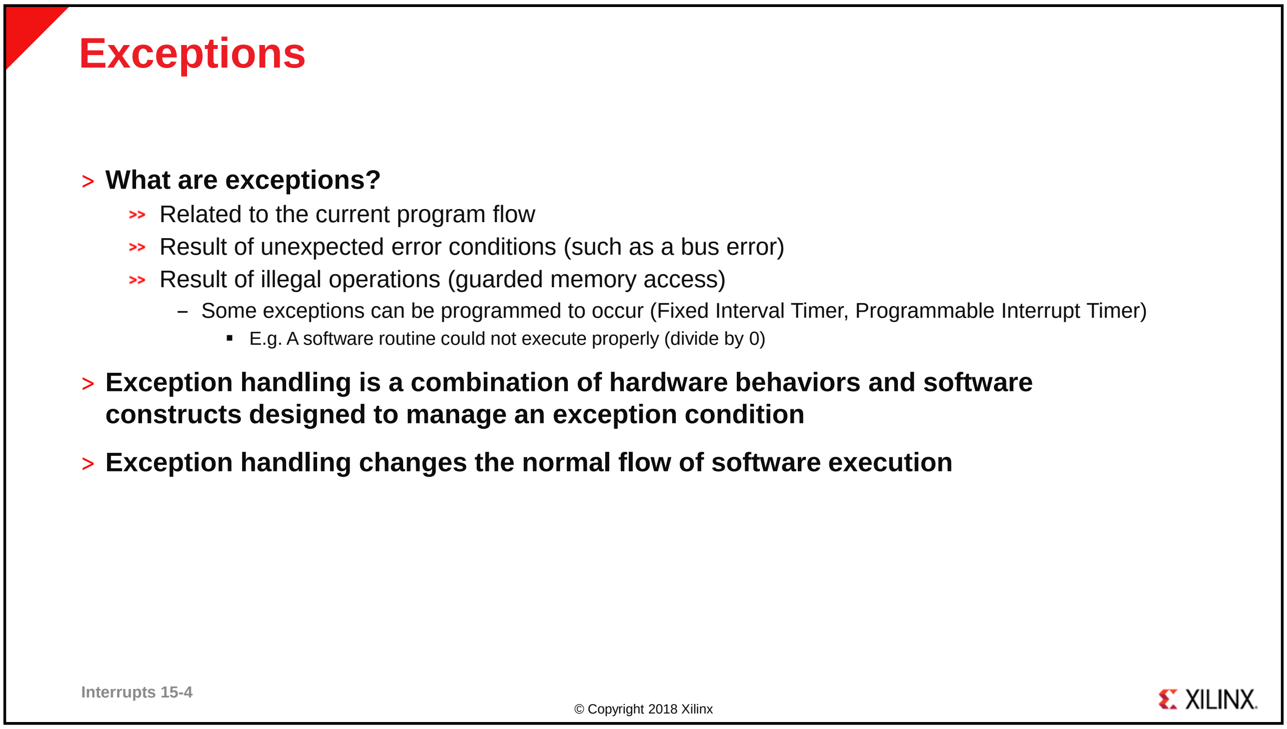
Objectives

- After completing this module, you will be able to:
  - Describe the interrupt structure of the Cortex-A9 processor
  - Write an interrupt handler for the targeted processor
  - Register the interrupt handler/interrupt service routine (ISR)
  - Use an interrupt controller to accommodate multiple interrupts
  - Apply proper programming techniques to reduce interrupt latency



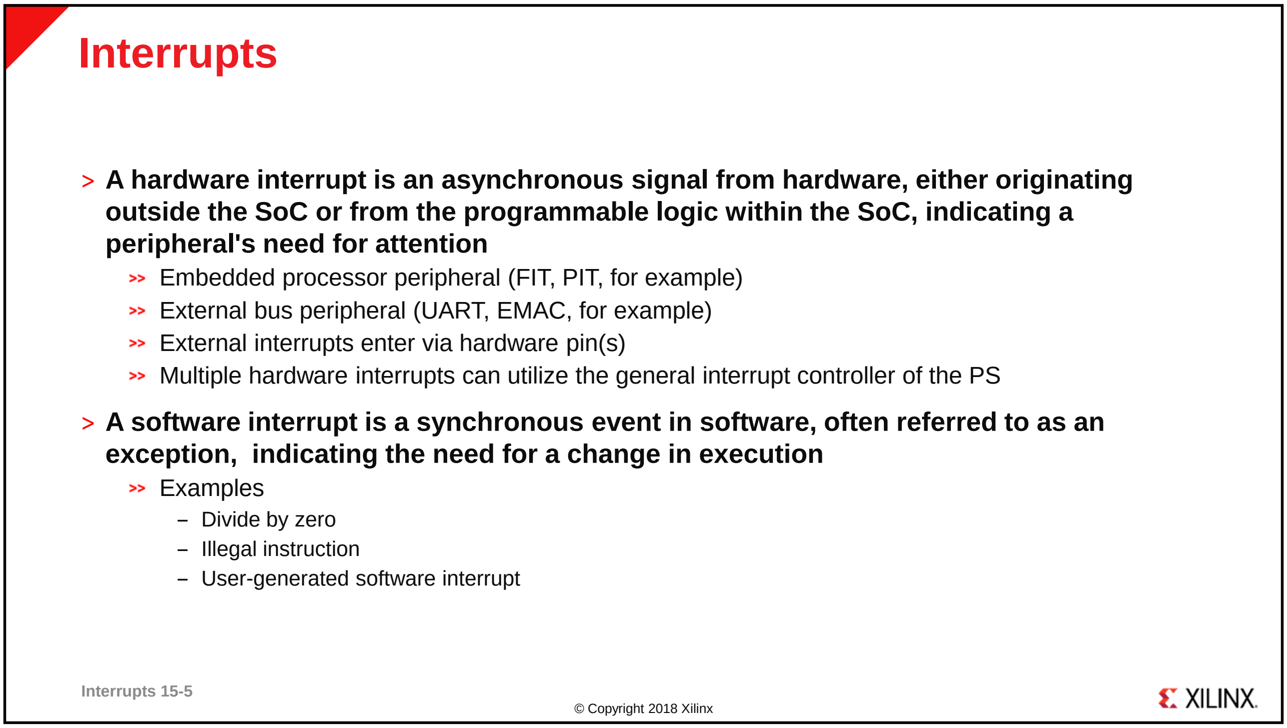
Outline

- > *Introduction*
- > Interrupts in Cortex-A9 Processor
- > Interrupt Handling in Cortex-A9 Processor
- > General Interrupt Controller of Cortex-A9 Processor
- > Interrupts Considerations
- > Summary



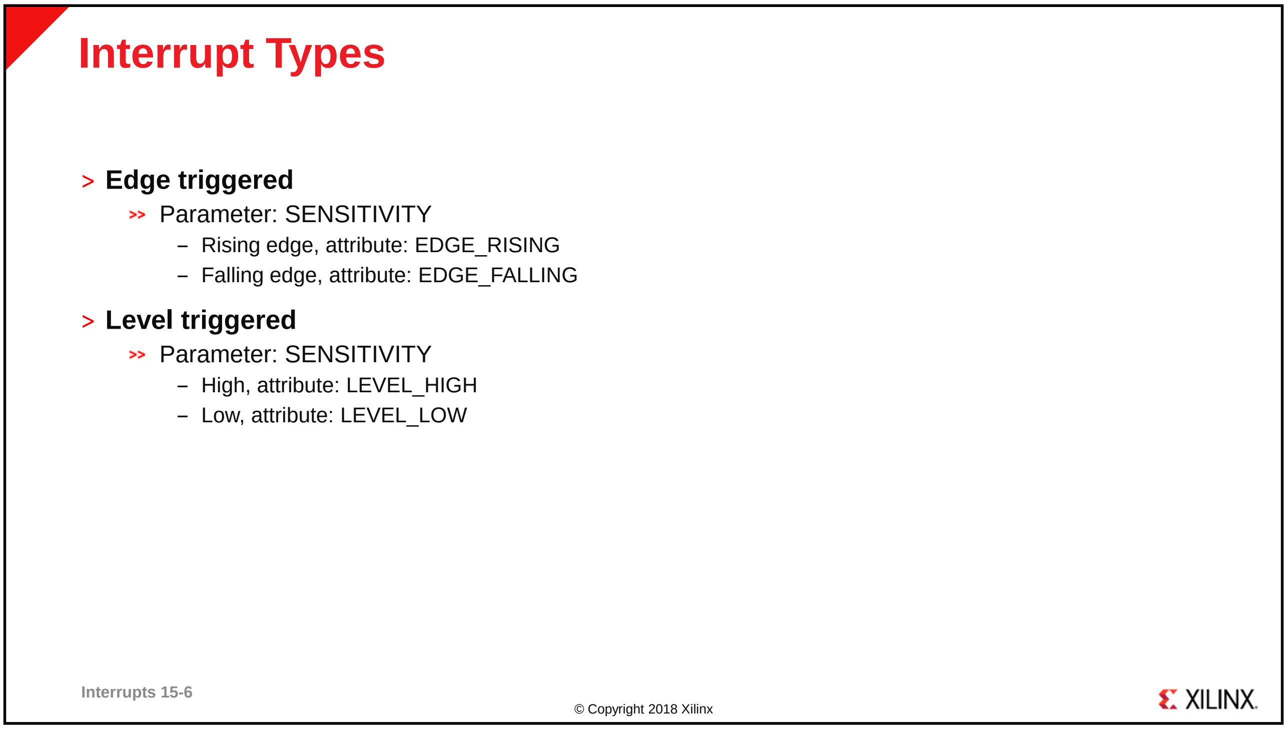
Exceptions

- > **What are exceptions?**
  - >> Related to the current program flow
  - >> Result of unexpected error conditions (such as a bus error)
  - >> Result of illegal operations (guarded memory access)
    - – Some exceptions can be programmed to occur (Fixed Interval Timer, Programmable Interrupt Timer)
      - ▪ E.g. A software routine could not execute properly (divide by 0)
- > **Exception handling is a combination of hardware behaviors and software constructs designed to manage an exception condition**
- > **Exception handling changes the normal flow of software execution**



Interrupts

- A hardware interrupt is an asynchronous signal from hardware, either originating outside the SoC or from the programmable logic within the SoC, indicating a peripheral's need for attention**
  - Embedded processor peripheral (FIT, PIT, for example)
  - External bus peripheral (UART, EMAC, for example)
  - External interrupts enter via hardware pin(s)
  - Multiple hardware interrupts can utilize the general interrupt controller of the PS
- **A software interrupt is a synchronous event in software, often referred to as an exception, indicating the need for a change in execution**
  - Examples
    - Divide by zero
    - Illegal instruction
    - User-generated software interrupt



Interrupt Types

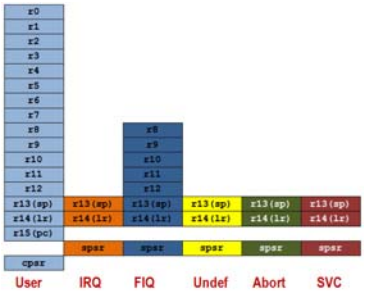
- **Edge triggered**
  - Parameter: SENSITIVITY
    - Rising edge, attribute: EDGE_RISING
    - Falling edge, attribute: EDGE_FALLING
- **Level triggered**
  - Parameter: SENSITIVITY
    - High, attribute: LEVEL_HIGH
    - Low, attribute: LEVEL_LOW

Interrupts in Cortex-A9 Processor

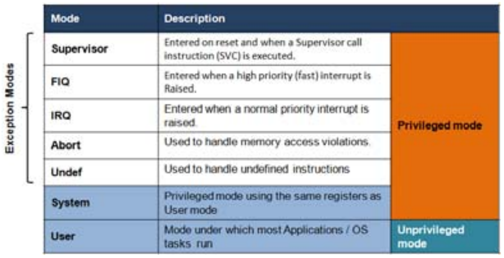
XILINX

Cortex-A9 Modes and Registers

- Cortex-A9 has seven execution modes
  - Five are exception modes
  - Each mode has its own stack space and different subset of registers
  - System mode will use the user mode registers
- Cortex-A9 has 37 registers
  - Up to 18 visible at any one time
  - Execution modes have some private registers that are banked in when the mode is changed
  - Non-banked registers are shared between modes



User	IRQ	FIQ	Undef	Abort	SVC
r0					
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8		r8			
r9		r9			
r10		r10			
r11		r11			
r12		r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
r15 (pc)					
	spsr	spsr	spsr	spsr	spsr
cpsr					

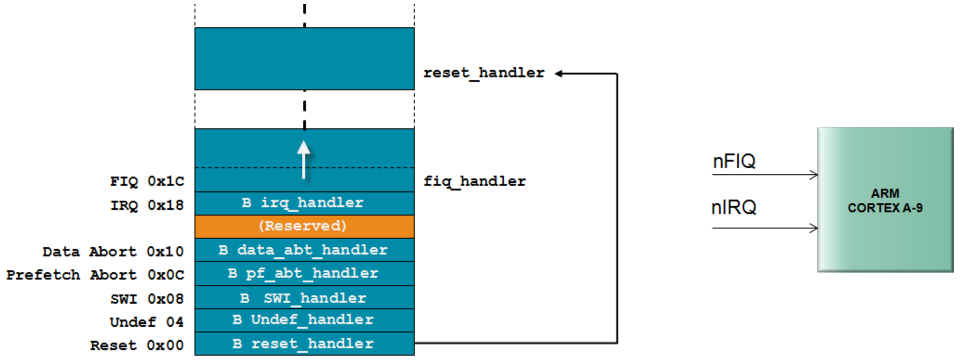


	Mode	Description	
Exception Modes	Supervisor	Entered on reset and when a Supervisor call instruction (SVC) is executed.	Privileged mode
	FIQ	Entered when a high priority (fast) interrupt is Raised.	
	IRQ	Entered when a normal priority interrupt is raised.	
	Abort	Used to handle memory access violations.	
	Undef	Used to handle undefined instructions	
	System	Privileged mode using the same registers as User mode	
	User	Mode under which most Applications / OS tasks run	Unprivileged mode

XILINX

Cortex-A9 Exceptions

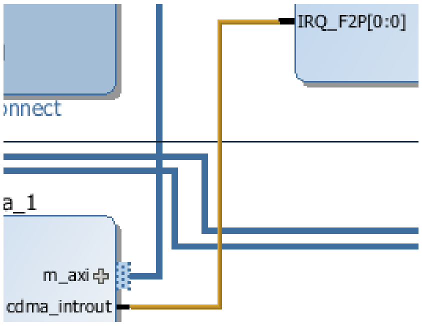
- **In Cortex-A9 processor interrupts are handled as exceptions**
  - Each Cortex-A9 processor core accepts two different levels of interrupts
    - nFIQ interrupts from secure sources (serviced first)
    - nIRQ interrupts from either secure sources or non-secure sources



Interrupt Handling in Cortex-A9 Processor

Interrupt Handling

- **In order to support interrupts,**
  - They must be connected
    - Interrupt signals from on-core peripherals are already connected
    - Interrupt signals from PL must explicitly be connected
  - They must be enabled in software
    - Use peripheral's API
  - Interrupt service routine must be developed



Interrupts 15-11

Interrupt Service Process

- **Service both interrupts and exceptions**
  - Interrupt Service Routine (ISR) must be registered
  - Current program execution is suspended after the current instruction
  - Context information is saved so that execution can return to the current program
  - Execution is transferred to an interrupt handler to service the interrupt
    - Interrupt handler calls an ISR
    - For simple situations, the handler and ISR can be combined operations
    - Each ISR is unique to the task at hand
      - UART interrupt to process a character
      - Divide-by-zero exception to change program flow
  - When finished
    - Normal: Returns to point in program where interrupt occurred
    - Exception: Branches to error recovery

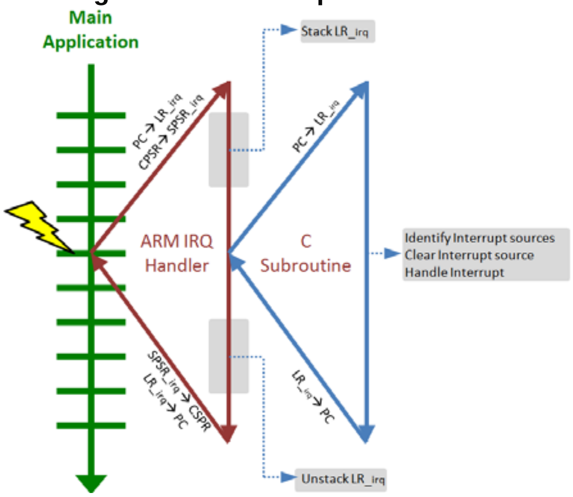
Interrupts 15-12

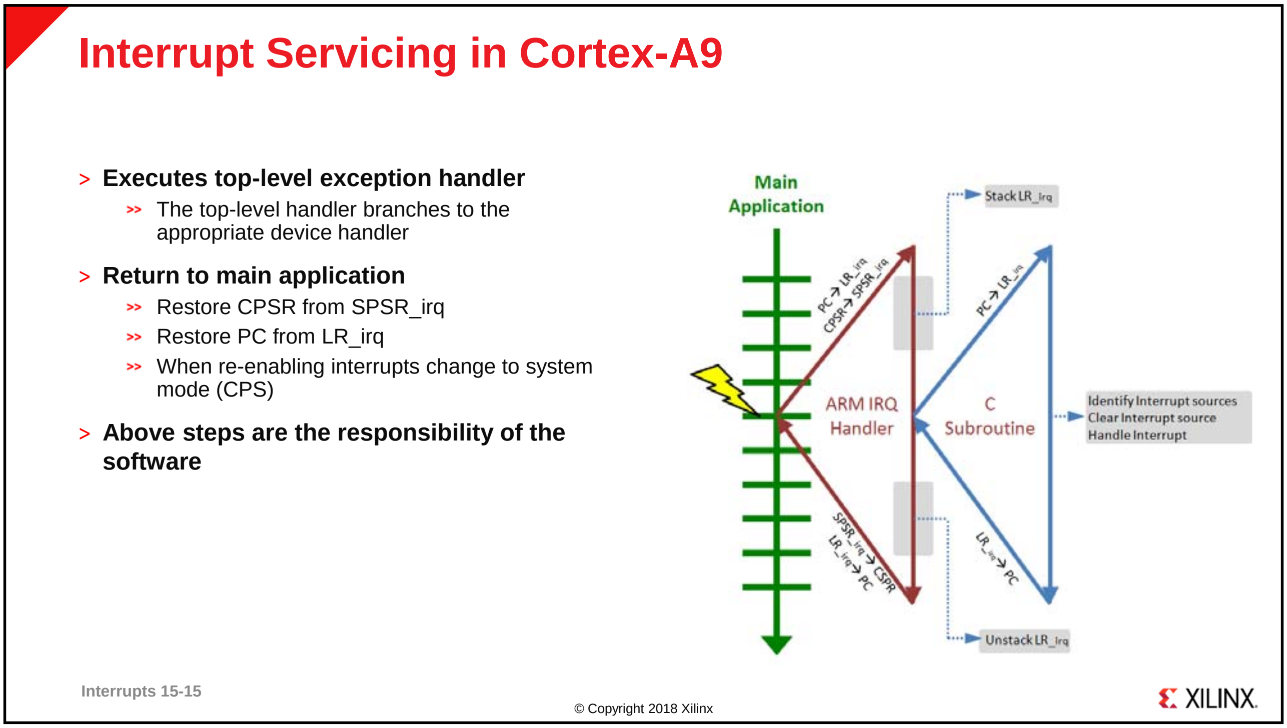
Interrupt Inclusion – Software

- **Requirements for including an interrupt into the application**
  - Write a *void* software function that services the interrupt
  - Use the provided device IP routines to facilitate writing the ISR
    - Clear interrupt
    - Perform the interrupt function
    - Re-enable interrupt upon exit
  - Register the interrupt handler by using an appropriate function
    - Single external interrupt registers with the processor function
    - Multiple external interrupts register with the interrupt controller
    - The call back registration argument is optional

Interrupt Servicing in Cortex-A9

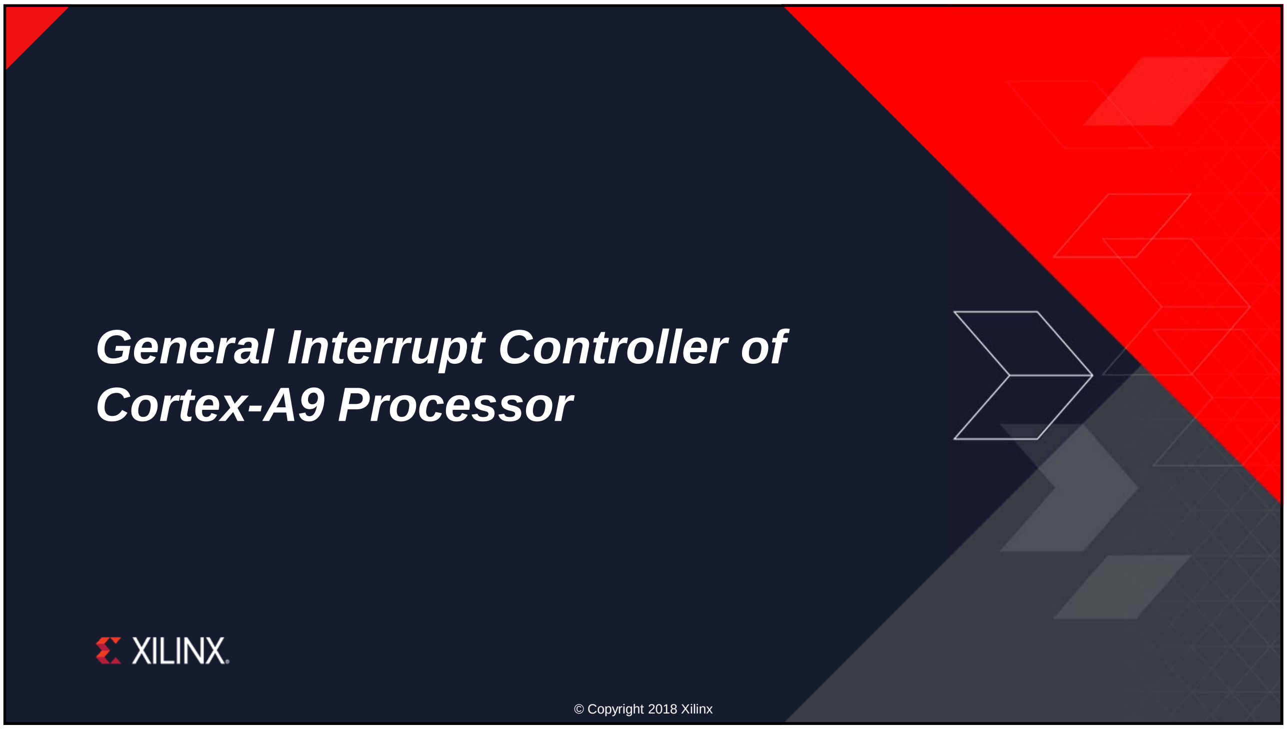
- **When an interrupt is received, the current executing instruction completes**
- **Save processor status**
  - Copies CPSR into SPSR_irq
  - Stores the return address in LR_irq
- **Change processor status for exception**
  - Mode field bits
  - ARM or thumb (T2) state
  - Interrupt disable bits (if appropriate)
  - Sets PC to vector address (either FIQ or IRQ)
- **The above steps are performed automatical**





Interrupt Servicing in Cortex-A9

- Executes top-level exception handler
  - The top-level handler branches to the appropriate device handler
- Return to main application
  - Restore CPSR from SPSR_irq
  - Restore PC from LR_irq
  - When re-enabling interrupts change to system mode (CPS)
- Above steps are the responsibility of the software



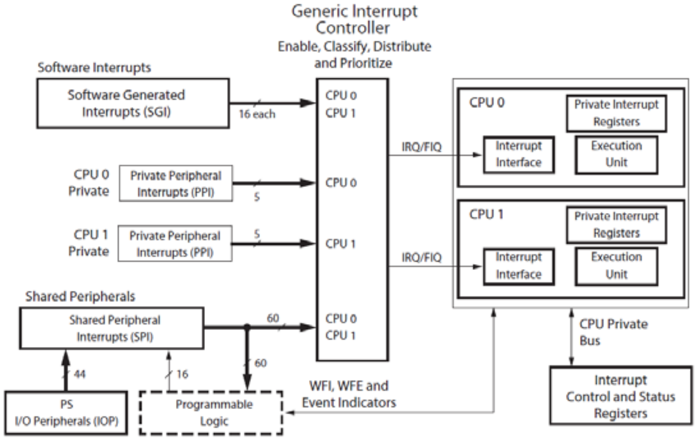
General Interrupt Controller of Cortex-A9 Processor

General Interrupt Controller (GIC)

- **Each processor has its own configuration space for interrupts**
  - Ability to route interrupts to either or both processors
  - Separate mask registers for processors
- **Supports interrupt prioritization**
- **Handles up to 16 software-generated interrupts (SGI)**
- **Supports 64 shared peripheral interrupts (SPI) starting at ID 32**
  - Shared between both cores
- **Processes both level-sensitive interrupts and edge-sensitive interrupts**
- **Five private peripheral interrupts (PPI)**
  - Dedicated for each core (no user-selectable PPI)

System-level Block Diagram

- **PPI includes**
  - The global timer, private watchdog timer, private timer, and FIQ/IRQ from the PL
  - IRQ IDs 16-26 reserved, global timer 27, nFIRQ 28, private timer 29, watchdog timer 30, nIRQ 31
- **SPI includes interrupts**
  - Generated by the various I/O and memory controllers in the PS and PL
- **SGI are generated by writing to the registers in the GIC**
  - IRQ IDs 0-15

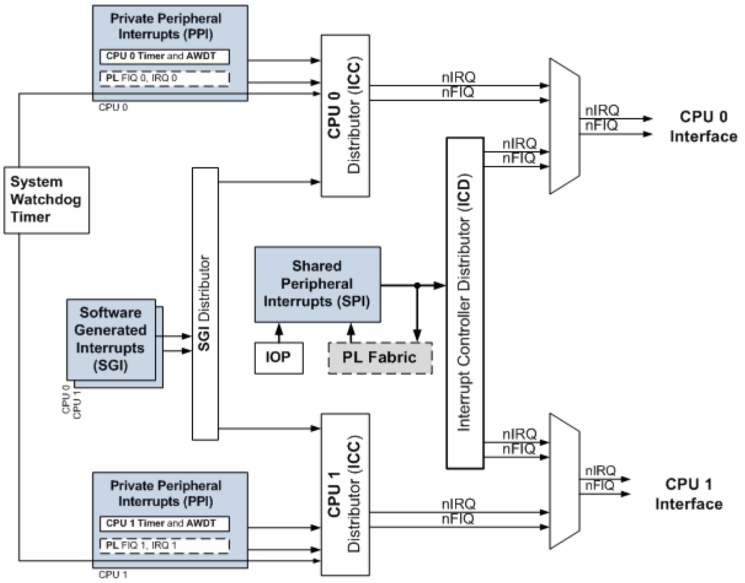

Generic Interrupt Controller
Enable, Classify, Distribute and Prioritize
Software Interrupts
Software Generated Interrupts (SGI)
16 each
CPU 0
CPU 1
CPU 0 Private
Private Peripheral Interrupts (PPI)
5
CPU 0
CPU 1 Private
Private Peripheral Interrupts (PPI)
5
CPU 1
Shared Peripherals
Shared Peripheral Interrupts (SPI)
60
CPU 0
CPU 1
44
16
60
PS I/O Peripherals (IOP)
Programmable Logic
WFI, WFE and Event Indicators
IRQ/FIQ
CPU 0
Private Interrupt Registers
Interrupt Interface
Execution Unit
IRQ/FIQ
CPU 1
Private Interrupt Registers
Interrupt Interface
Execution Unit
CPU Private Bus
Interrupt Control and Status Registers

Shared Peripherals Interrupts

Source	Interrupt Name	IRQ ID#
APU	CPU 1, 0 (L2, TLB, BTAC)	33:32
	L2 Cache	34
	OCM	35
Reserved	~	36
PMU	PMU [1,0]	38, 37
XADC	XADC	39
DVI	DVI	40
SWDT	SWDT	41
Timer	TTC 0	43:42
Reserved	~	44
DMAC	DMAC Abort	45
	DMAC [3:0]	49:46
Memory	SMC	50
	Quad SPI	51
Debug	CTI	~
IOP	GPIO	52
	USB 0	53
	Ethernet 0	54
	Ethernet 0 Wakeup	55
	SDIO 0	56
	I2C 0	57
	SPI 0	58
	UART 0	59
	CAN 0	60
PL	FPGA [2:0]	63:61
	FPGA [7:3]	68:64
Timer	TTC 1	71:69
DMAC	DMAC[7:4]	75:72
IOP	USB 1	76
	Ethernet 1	77
	Ethernet 1 Wakeup	78
	SDIO 1	79
	I2C 1	80
	SPI 1	81
	UART 1	82
	CAN 1	83
PL	FPGA [15:8]	91:84
SCU	Parity	92
Reserved	~	95:93

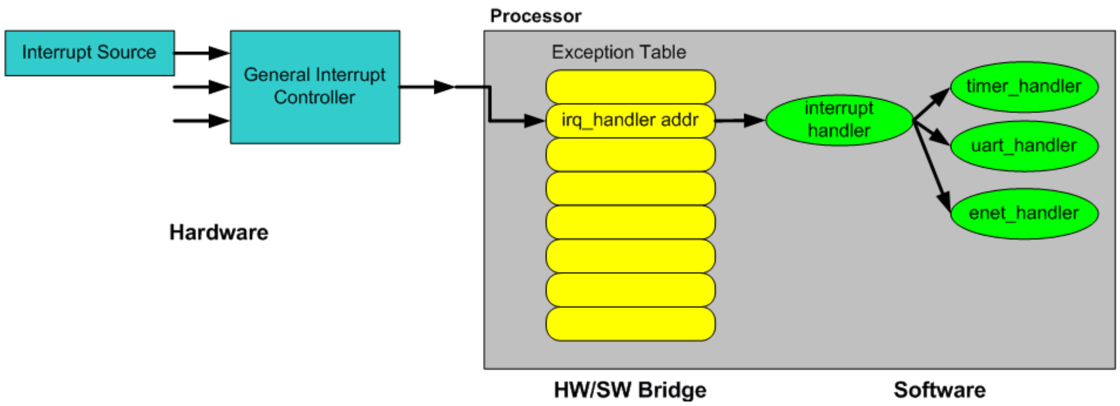
Interrupts 15-19

GIC Block Diagram



Interrupts 15-20

When Interrupt Request Comes to GIC

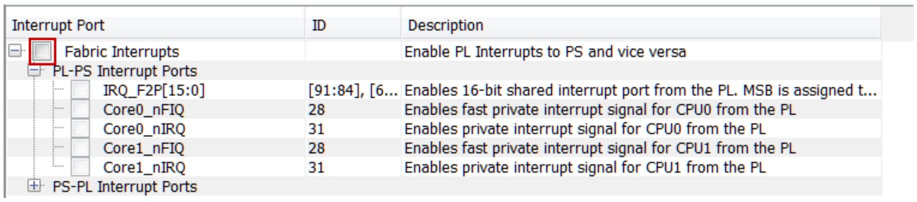


Connecting Interrupt Source to GIC

- Connect the interrupting net to the GIC
- Set the interrupt type at the interrupt generating source
  - The interrupt type for the GIC is programmed through software
- Interrupt source
  - Another AXI peripheral in the programmable logic
  - Any of the selected hard blocks present in the processing system
  - External net

Connecting Interrupt Source to GIC

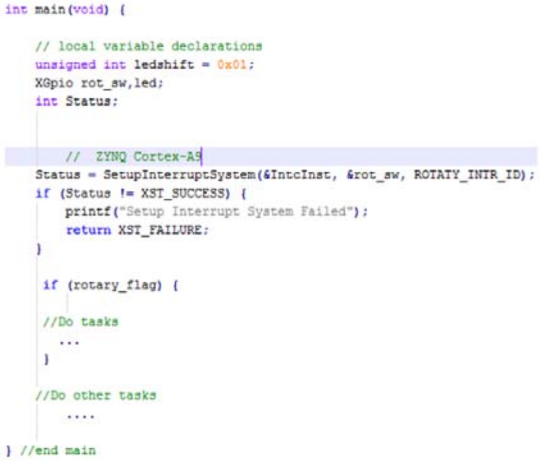
- The GIC also provides access to the private peripheral interrupts from the programmable logic
  - Basically a direct connection to the CPU's interrupt input
    - Bypasses the GIC
  - Corex_nFIQ (ID 28)
  - Corex_nIRQ (ID 31)



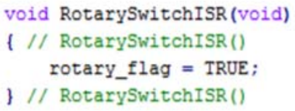
Interrupt Port	ID	Description
⊟ □ Fabric Interrupts		Enable PL Interrupts to PS and vice versa
⊟ PL-PS Interrupt Ports		
□ IRQ_F2P[15:0]	[91:84], [6...	Enables 16-bit shared interrupt port from the PL. MSB is assigned t...
□ Core0_nFIQ	28	Enables fast private interrupt signal for CPU0 from the PL
□ Core0_nIRQ	31	Enables private interrupt signal for CPU0 from the PL
□ Core1_nFIQ	28	Enables fast private interrupt signal for CPU1 from the PL
□ Core1_nIRQ	31	Enables private interrupt signal for CPU1 from the PL
⊞ PS-PL Interrupt Ports		

Setting Up Interrupt System in Cortex-A9

- Software application: Setting up the interrupt system in the main code and ISR function



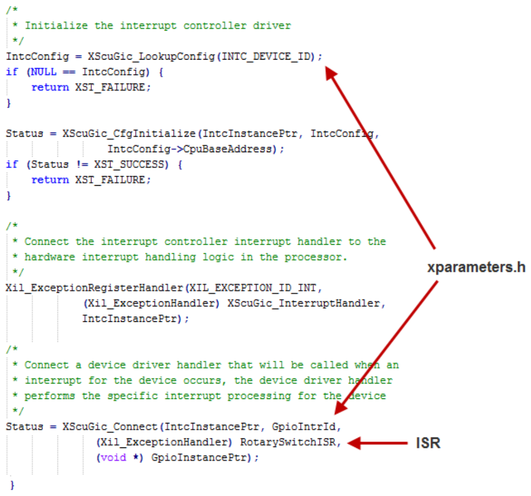
```
int main(void) {

    // local variable declarations
    unsigned int ledshift = 0x01;
    XGpio rot_sw,led;
    int Status;


        //  ZYNQ Cortex-A9
    Status = SetupInterruptSystem(&IntcInst, &rot_sw, ROTATY_INTR_ID);
    if (Status != XST_SUCCESS) {
        printf("Setup Interrupt System Failed");
        return XST_FAILURE;
    }

     if (rotary_flag) {

     //Do tasks
       ...
     }

    //Do other tasks
        ....

} //end main
```



```
void RotarySwitchISR(void)
{ // RotarySwitchISR()
     rotary_flag = TRUE;
} // RotarySwitchISR()
```

Interrupt Controller Software Requirements

- Initialize the interrupt controller
- Register the interrupt controller interrupt handler to the hardware interrupt handling logic in the processor
- Connect the device driver handler that will be called when an interrupt for the device occurs
- Enable the interrupts

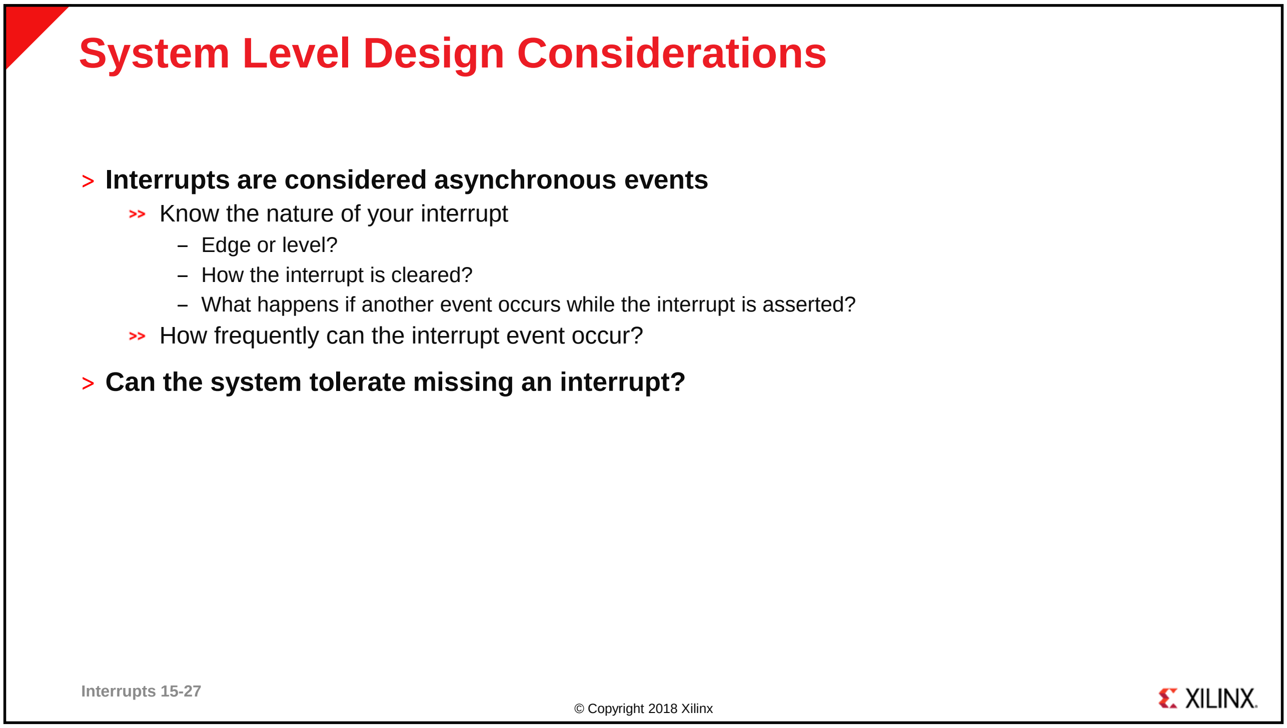


```
/*
 * Initialize the interrupt controller driver
 */
IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
if (NULL == IntcConfig) {
    return XST_FAILURE;
}

Status = XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig,
                IntcConfig->CpuBaseAddress);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}

/*
 * Connect the interrupt controller interrupt handler to the
 * hardware interrupt handling logic in the processor.
 */
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler) XScuGic_InterruptHandler,
            IntcInstancePtr);

/*
 * Connect a device driver handler that will be called when an
 * interrupt for the device occurs, the device driver handler
 * performs the specific interrupt processing for the device
 */
Status = XScuGic_Connect(IntcInstancePtr, GpioIntrId,
             (Xil_ExceptionHandler) RotarySwitchISR,
             (void *) GpioInstancePtr);

 }
```

XILINX

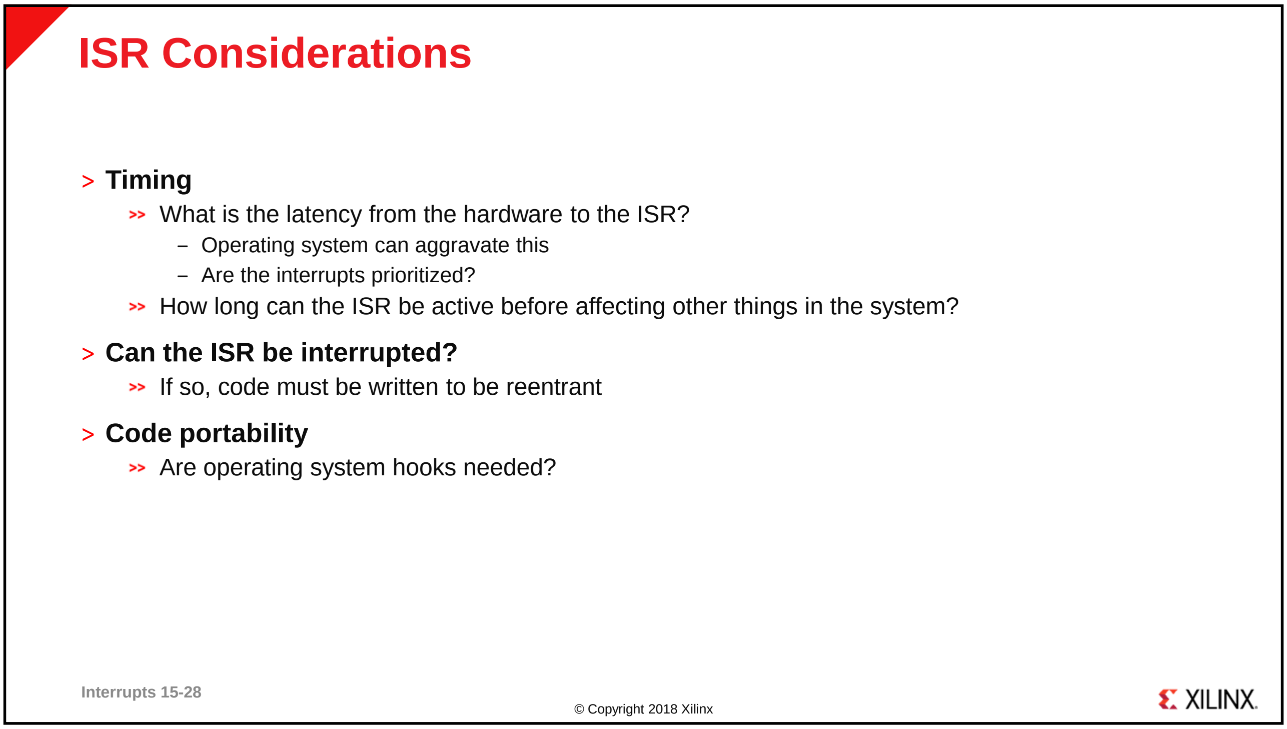
Interrupts Considerations

XILINX



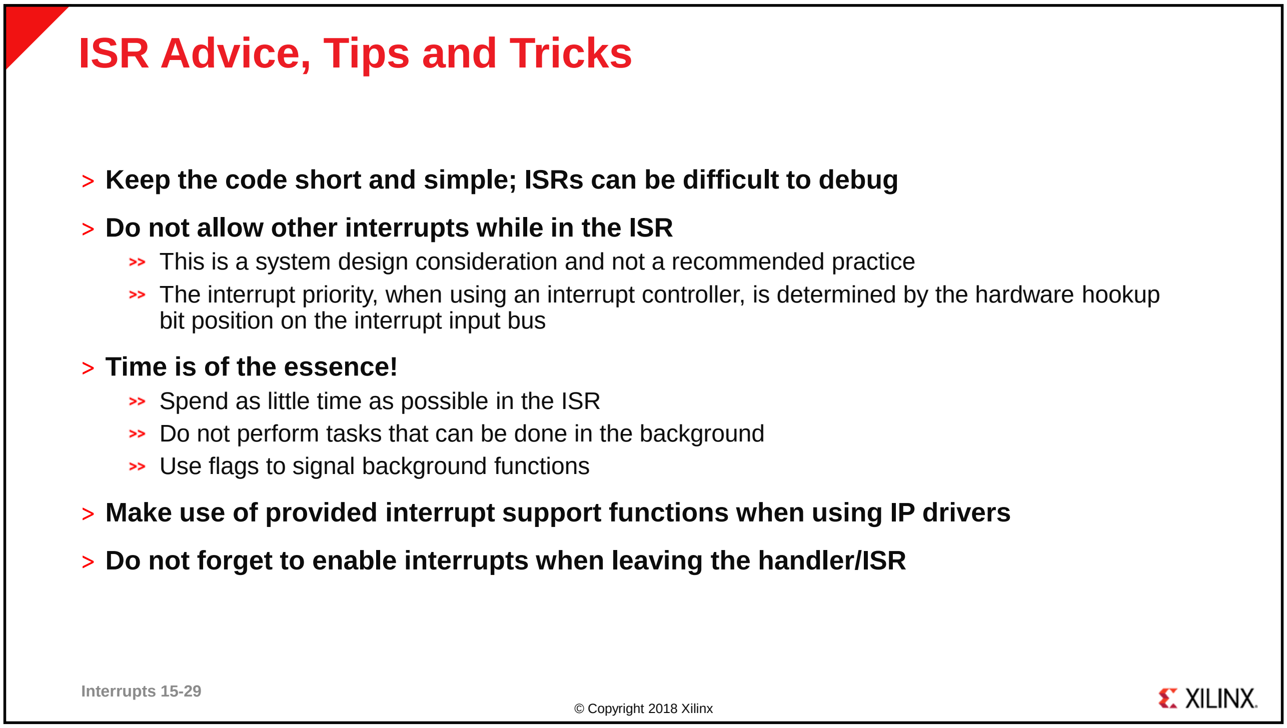
System Level Design Considerations

- Interrupts are considered asynchronous events**
  - Know the nature of your interrupt
    - Edge or level?
    - How the interrupt is cleared?
    - What happens if another event occurs while the interrupt is asserted?
  - How frequently can the interrupt event occur?
- **Can the system tolerate missing an interrupt?**



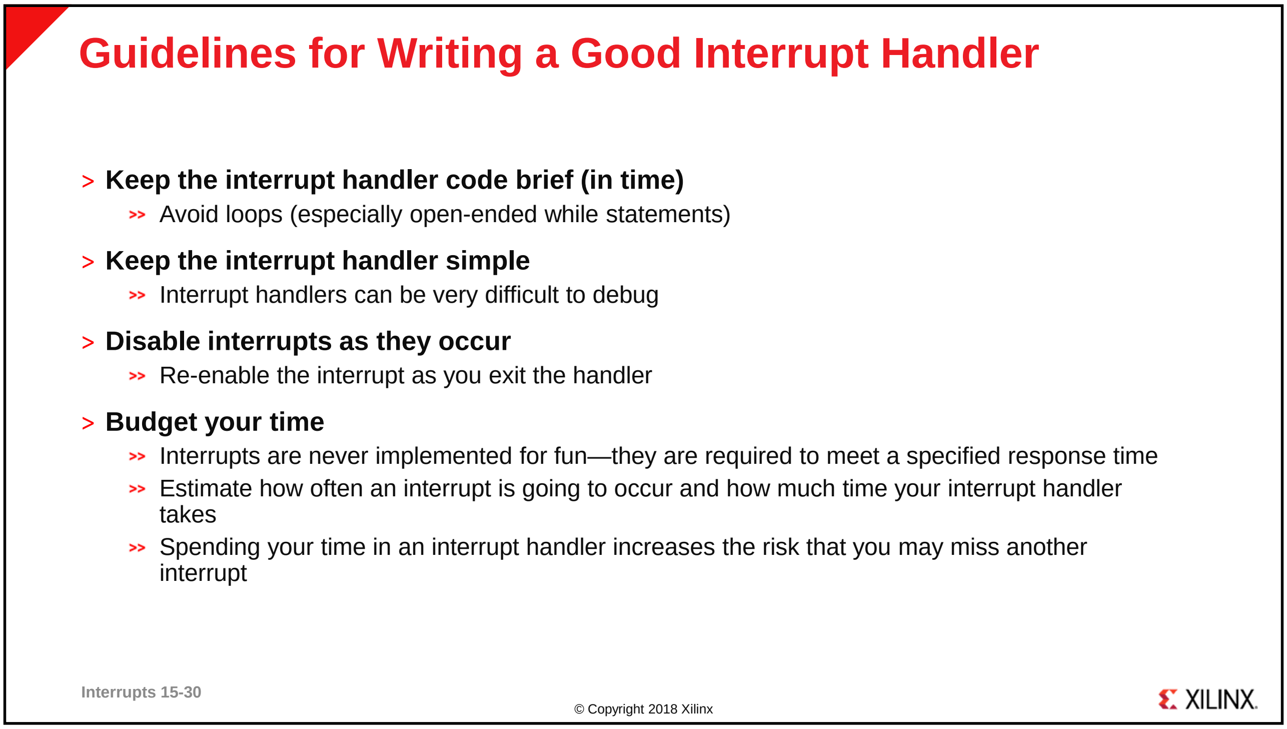
ISR Considerations

- **Timing**
  - What is the latency from the hardware to the ISR?
    - Operating system can aggravate this
    - Are the interrupts prioritized?
  - How long can the ISR be active before affecting other things in the system?
- **Can the ISR be interrupted?**
  - If so, code must be written to be reentrant
- **Code portability**
  - Are operating system hooks needed?



ISR Advice, Tips and Tricks

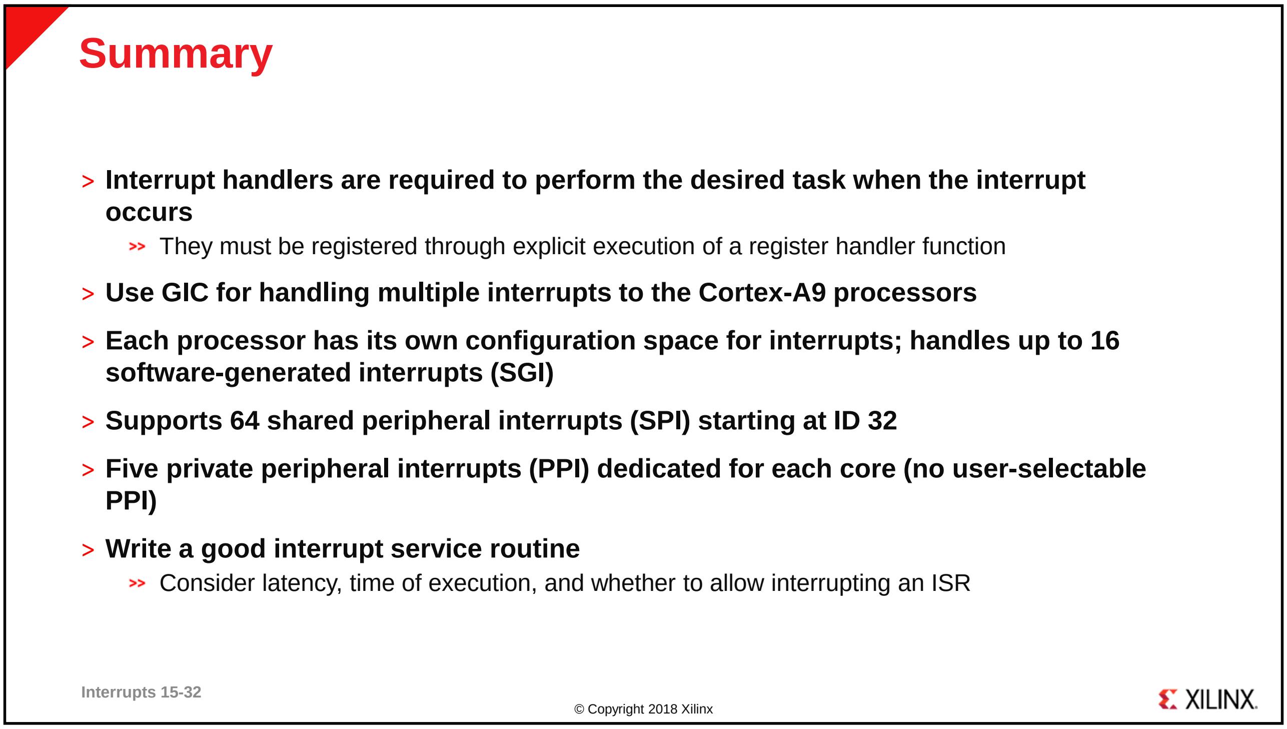
- **Keep the code short and simple; ISRs can be difficult to debug**
- **Do not allow other interrupts while in the ISR**
  - This is a system design consideration and not a recommended practice
  - The interrupt priority, when using an interrupt controller, is determined by the hardware hookup bit position on the interrupt input bus
- **Time is of the essence!**
  - Spend as little time as possible in the ISR
  - Do not perform tasks that can be done in the background
  - Use flags to signal background functions
- **Make use of provided interrupt support functions when using IP drivers**
- **Do not forget to enable interrupts when leaving the handler/ISR**



Guidelines for Writing a Good Interrupt Handler

- **Keep the interrupt handler code brief (in time)**
  - Avoid loops (especially open-ended while statements)
- **Keep the interrupt handler simple**
  - Interrupt handlers can be very difficult to debug
- **Disable interrupts as they occur**
  - Re-enable the interrupt as you exit the handler
- **Budget your time**
  - Interrupts are never implemented for fun—they are required to meet a specified response time
  - Estimate how often an interrupt is going to occur and how much time your interrupt handler takes
  - Spending your time in an interrupt handler increases the risk that you may miss another interrupt

Summary



Summary

- Interrupt handlers are required to perform the desired task when the interrupt occurs
  - They must be registered through explicit execution of a register handler function
- Use GIC for handling multiple interrupts to the Cortex-A9 processors
- Each processor has its own configuration space for interrupts; handles up to 16 software-generated interrupts (SGI)
- Supports 64 shared peripheral interrupts (SPI) starting at ID 32
- Five private peripheral interrupts (PPI) dedicated for each core (no user-selectable PPI)
- Write a good interrupt service routine
  - Consider latency, time of execution, and whether to allow interrupting an ISR

Adaptable.
Intelligent.