Interrupts in Zynq Systems
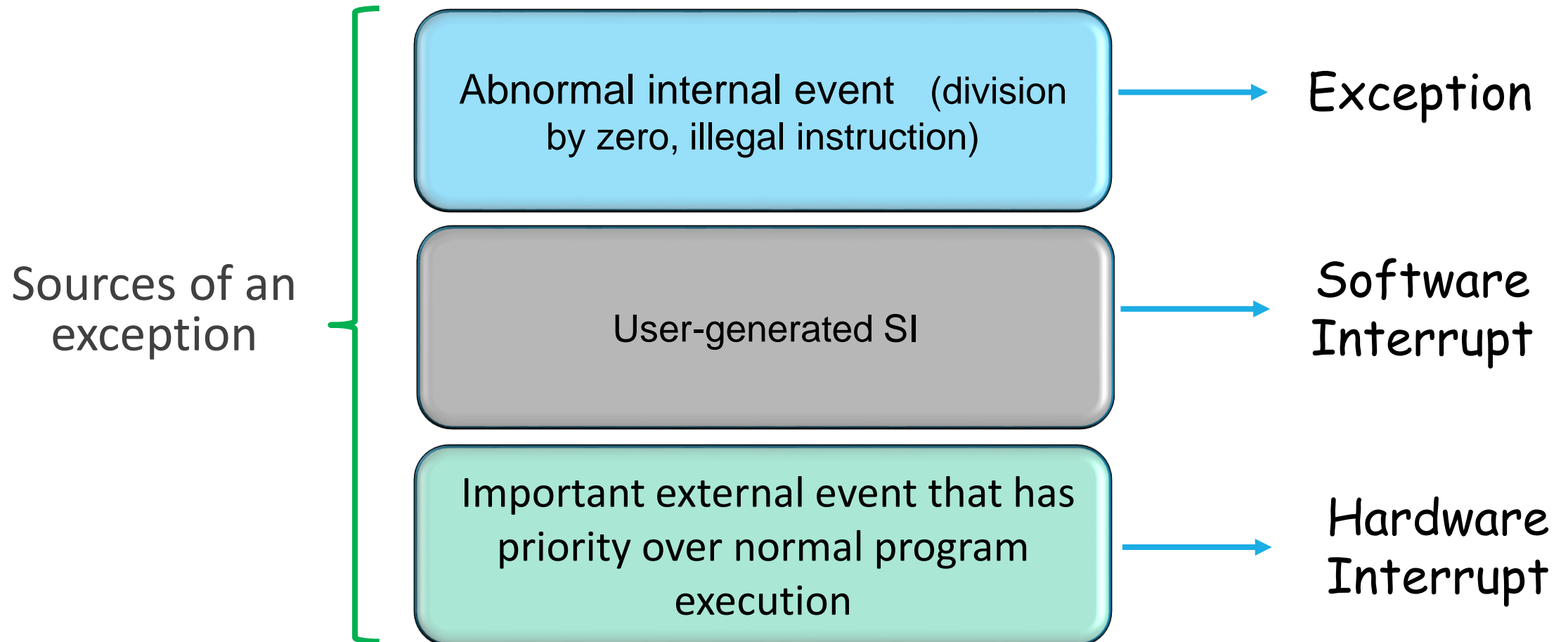
Cristian Sisterna

Universidad Nacional de San Juan

Argentina

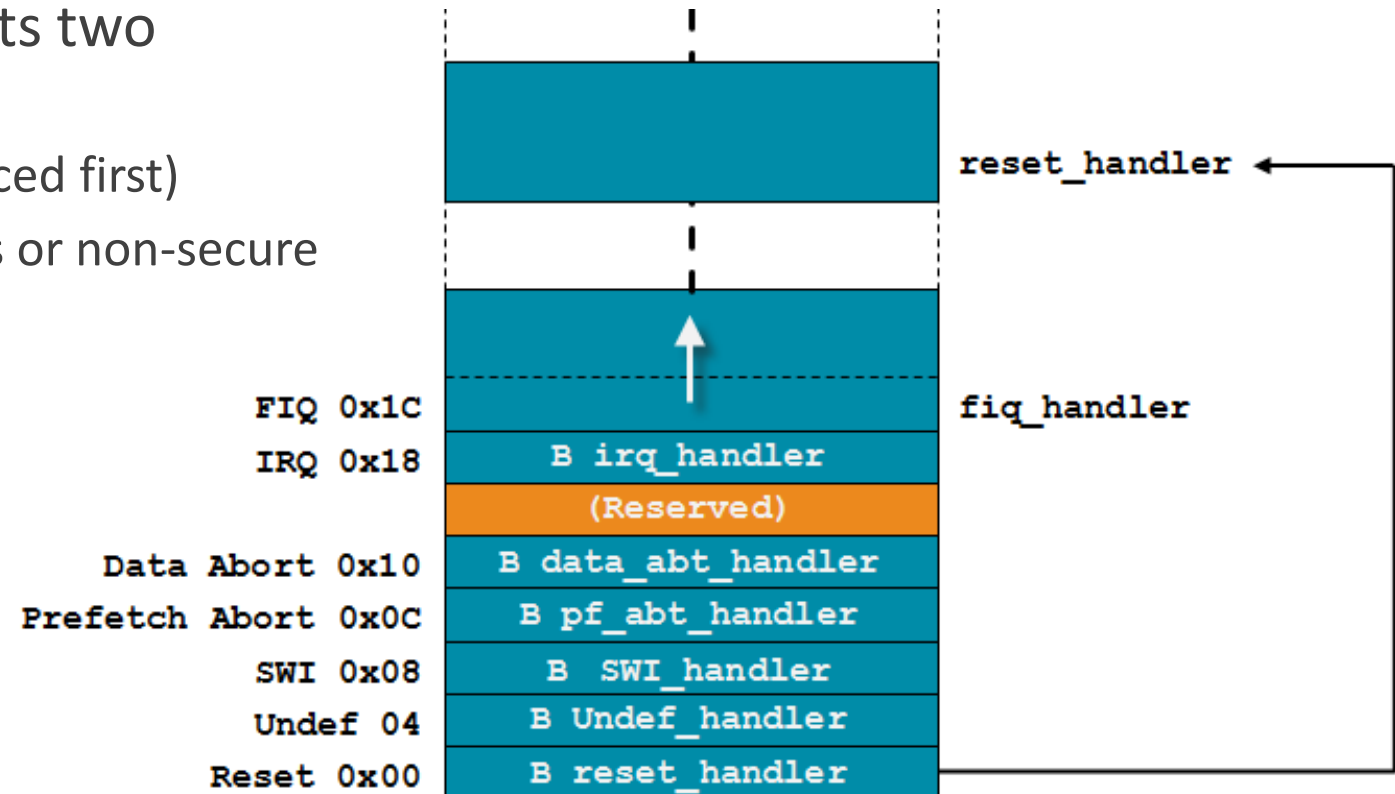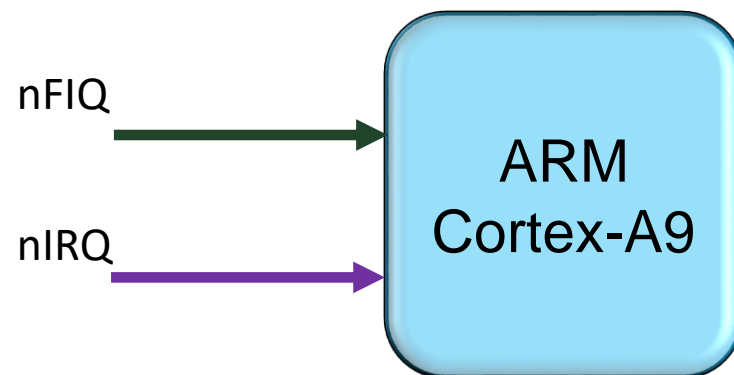# Exception / Interrupt

Special condition that requires a processor's immediate attention

Sources of an exception

Abnormal internal event (division by zero, illegal instruction)	→ Exception
User-generated SI	→ Software Interrupt
Important external event that has priority over normal program execution	→ Hardware Interrupt

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

nFIQ

nIRQ

ARM Cortex-A9

	reset_handler
FIQ 0x1C	fiq_handler
IRQ 0x18	B irq_handler
	(Reserved)
Data Abort 0x10	B data_abt_handler
Prefetch Abort 0x0C	B pf_abt_handler
SWI 0x08	B SWI_handler
Undef 04	B Undef_handler
Reset 0x00	B reset_handler

Pooling or Hardware Interrupt ?

Hardware
Interrupt

Polling

pizza

Interrupt Terminology

o **Interrupts Pins**: set of pins used as input of hardware interrupts

o **Interrupt Service Routine (ISR)**: 'C' code written to answer a specific interrupt. It can be hardware or software interrupt

o **Interrupt Priority**: In systems with more tan one source of interrupt, some interrupt have higher priority of attendance than other.

o **Interrupt Vector**: code loaded in the bus by the interrupting peripheral that contains the address of the specific ISR

o **Interrupt Mask**: interrupt that can be enable or disable by software

o **Non-Maskable Interrupt (NMI)**: source of interrupt that can not be ignored

o **Interrupt Latency**: The time interval from when the interrupt is first asserted to the time the CPU recognizes it. This will depend much upon whether interrupts are disabled, prioritized and what the processor is currently executing

o **Interrupt Response Time**: The time interval between the CPU recognizing the interrupt to the time when the first instruction of the interrupt service routine is executed. This is determined by the processor architecture and clock speed

Polling

A software routine is used to identify the peripheral requesting service. A simple polling technique is used, each peripheral is checked to see if it was the one needing service.

↑ Efficient when events arrives very often

↑ It requires no special hardware

↑ Quick response time (lees overhead)

↓ For fast peripherals, polling may simply not be fast enough to satisfy the minimum service requirements

↓ It takes CPU time even when there is peripheral looking for attention; as it needlessly checks the status of all devices all the time

○ priority of the peripherals is determined by the order in the polling loop

Polling is like picking up your phone every few seconds to find out whether someone has called you

Hardware Interrupt

A special software routine is used to attend an specific peripheral when it rise the need for attention

↑ It **does not takes CPU time** even when there is no peripheral looking for attention

↑ Good for **fast peripherals**, polling may simply not be fast enough to satisfy the minimum service requirements

↓ **Inefficient** when events arrives very often

↓ **It does** requires special hardware

↓ **Slow** response time (more overhead)

o priority of the peripherals is determined by the hardware

Hardware Interrupt is like picking up your phone
ONLY when it rings

Polling vs Interrupt

EVENT	
Asynchronous	Synchronous (i.e. you know when to expect it within a small window)
Urgent	Not urgent (i.e. a slow polling interval has not bad effects)
Infrequent	Frequent (i.e. majority of your polling cycles create a 'hit')

Hardware
Interrupt

Polling

Hardware Interrupt in the Zynq

Hardware Interrupts

A hardware interrupt is an asynchronous signal from hardware, originating either from outside the SoC, from any of the PS peripherals or from the logic implemented in the PL, indicating that a peripheral needs attention

Source of Hardware Interrupts:

- Embedded processor peripheral (FIT, PIT, for example)
- External bus peripheral (UART, EMAC, for example)
- External interrupts enter via hardware pin(s)
- PL block
- Multiple hardware interrupts can utilize general interrupt controller of the PS

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

How is the Interrupt Flow ? Part 1

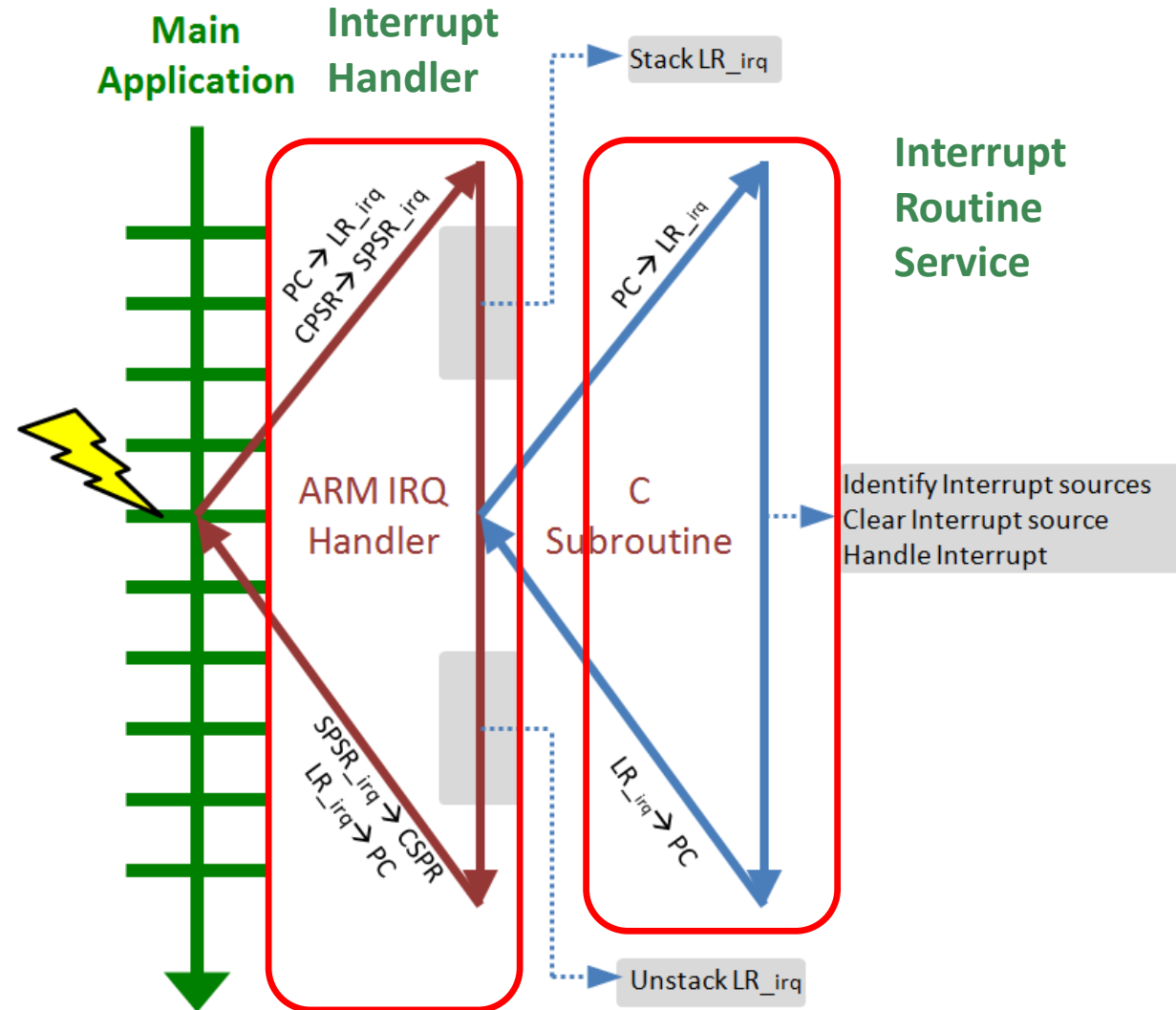**When an interrupt occurs, the current executing instruction completes**

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)

(This steps are performed automatically by the core)
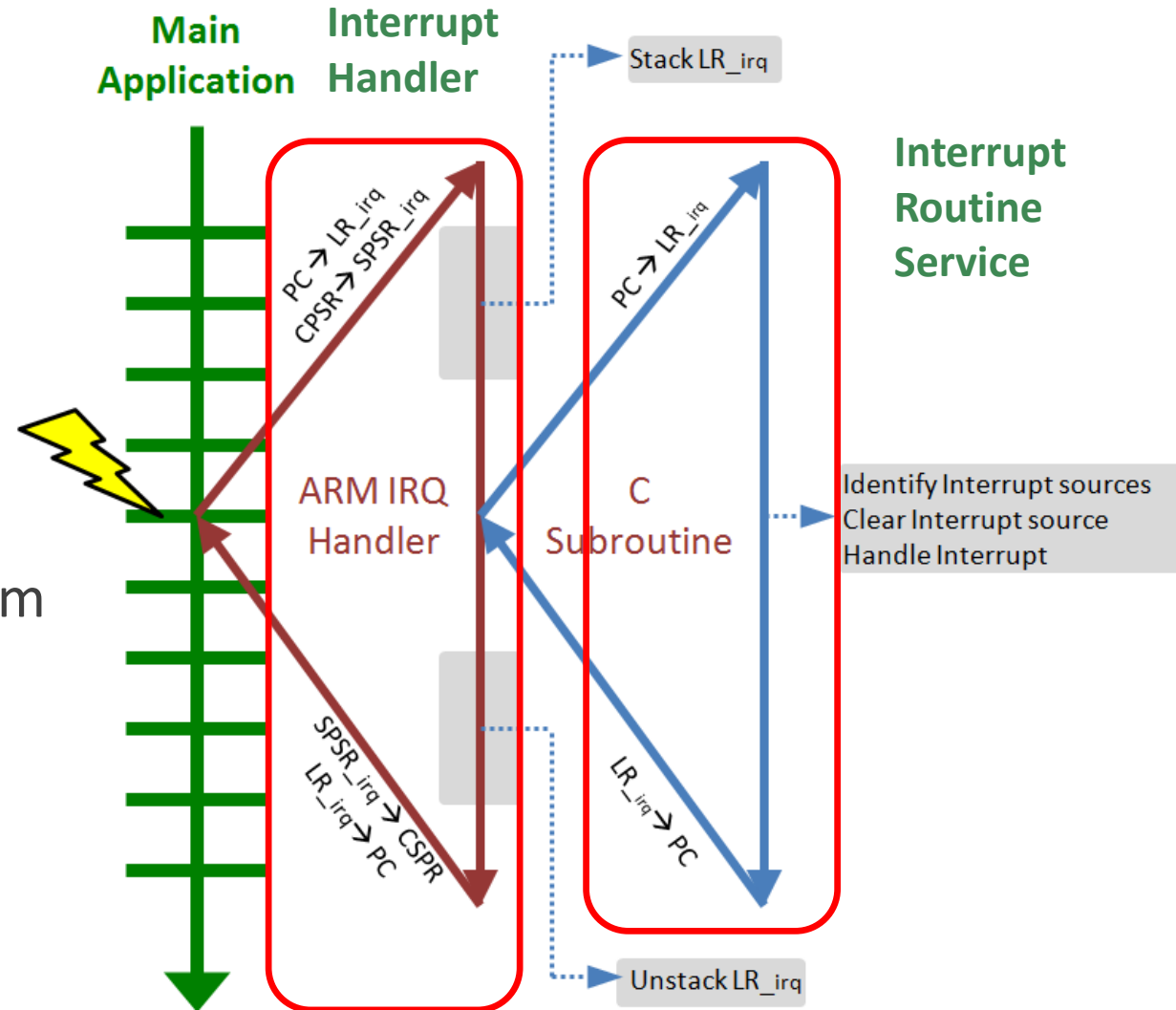
Executes top-level exception handler

◦ The top-level handler branches to the appropriate device handler
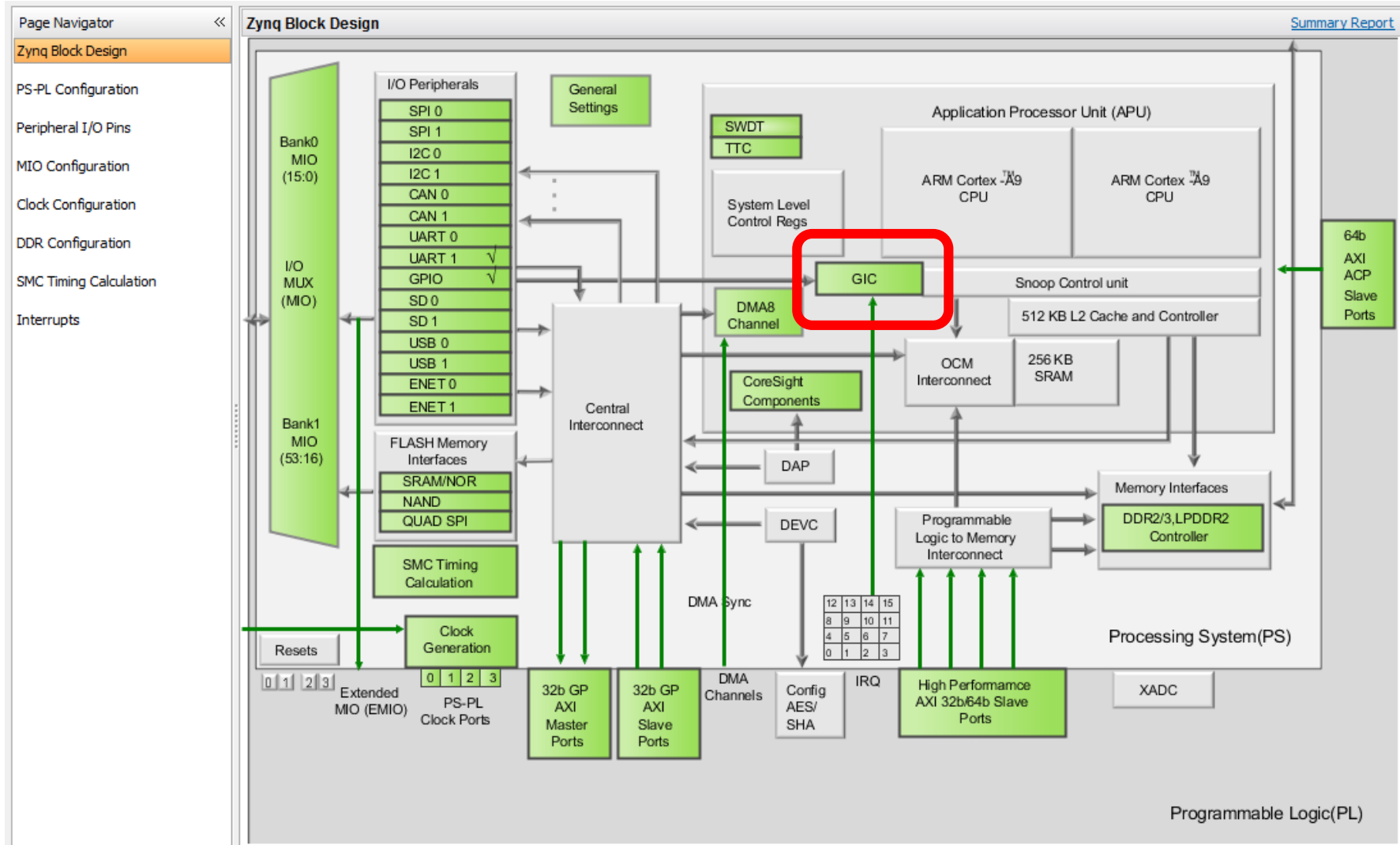
Return to main application

◦ Restore CPSR from SPSR_irq

◦ Restore PC from LR_irq

◦ When re-enabling interrupts change to system mode (CPS)

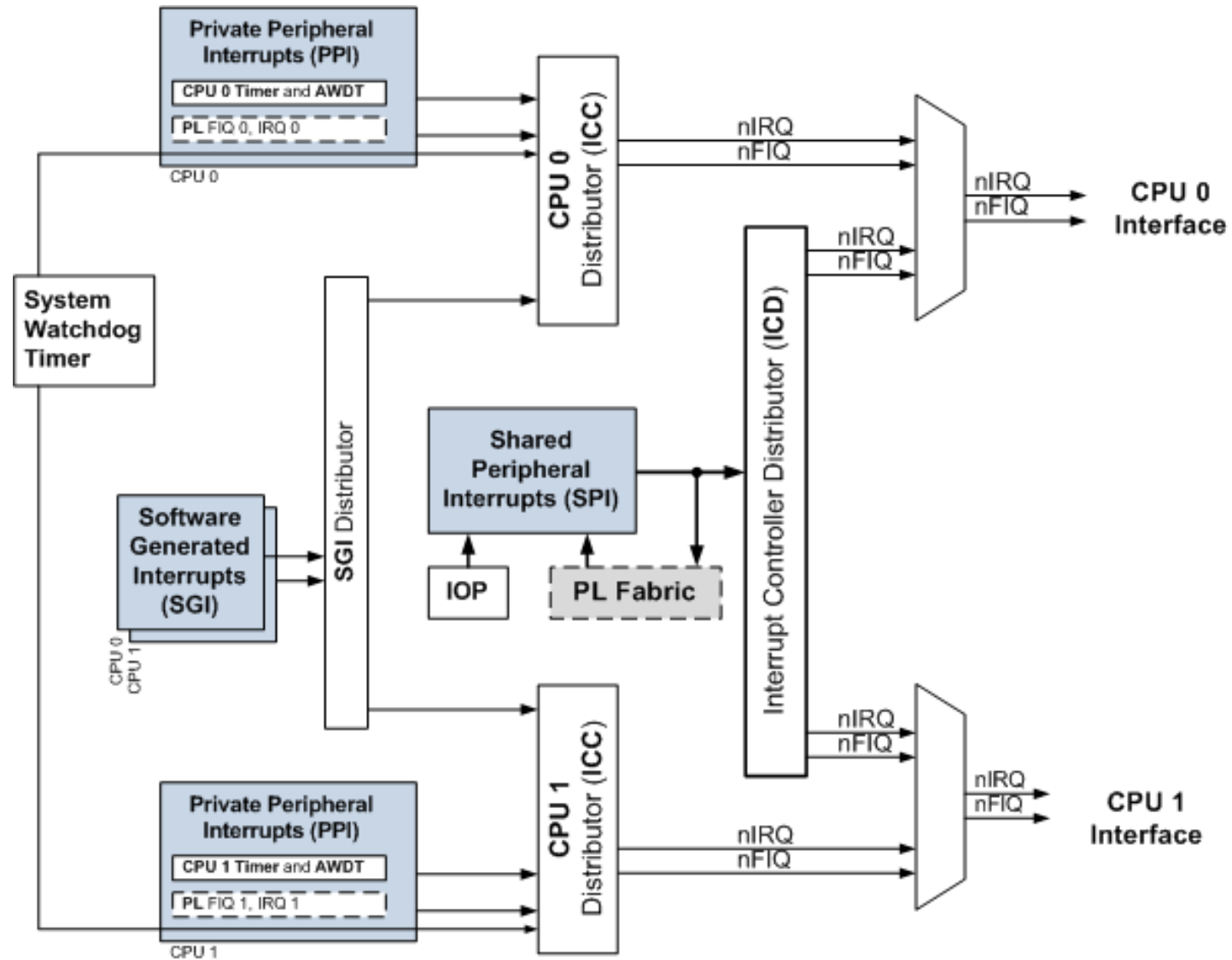(Above steps are the responsibility of the software)

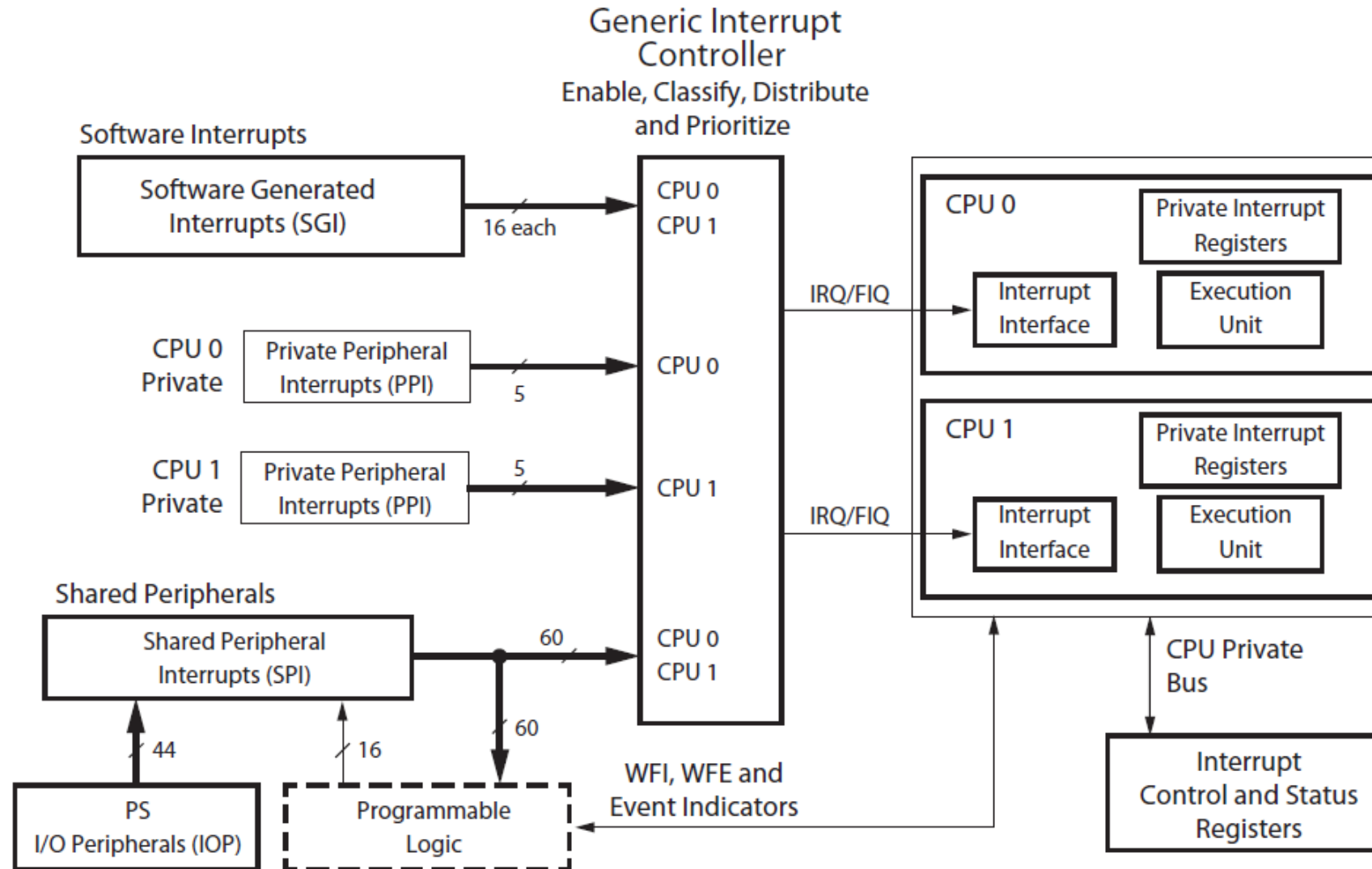# Generic Interrupt Controller (GIC)

Generic Interrupt Controller (GIC)

Generic Interrupt Controller (PL390)

GIC Block Diagram

System Level Block Diagram

Generic Interrupt Controller (GIC)

o Each processor has its own configuration space for interrupts

- o Ability to route interrupts to either or both processors
- o Separate mask registers for processors

o Supports interrupt prioritization

o Handles up to 16 software-generated interrupts (SGI)

o Supports 64 shared peripheral interrupts (SPI) starting at ID 32

o Processes both level-sensitive interrupts and edge-sensitive interrupts

o Five private peripheral interrupts (PPI) dedicated for each core (no user-selectable PPI)

Interrupt Flow in the GIC
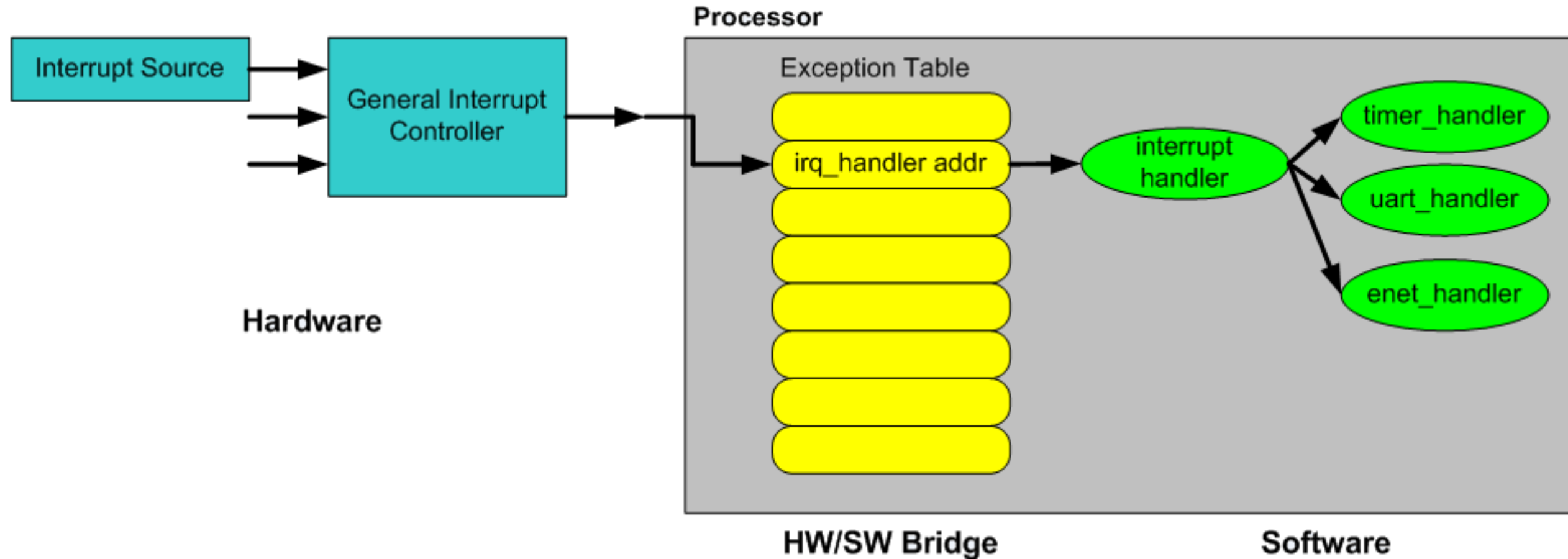
System Interrupt in the Zynq
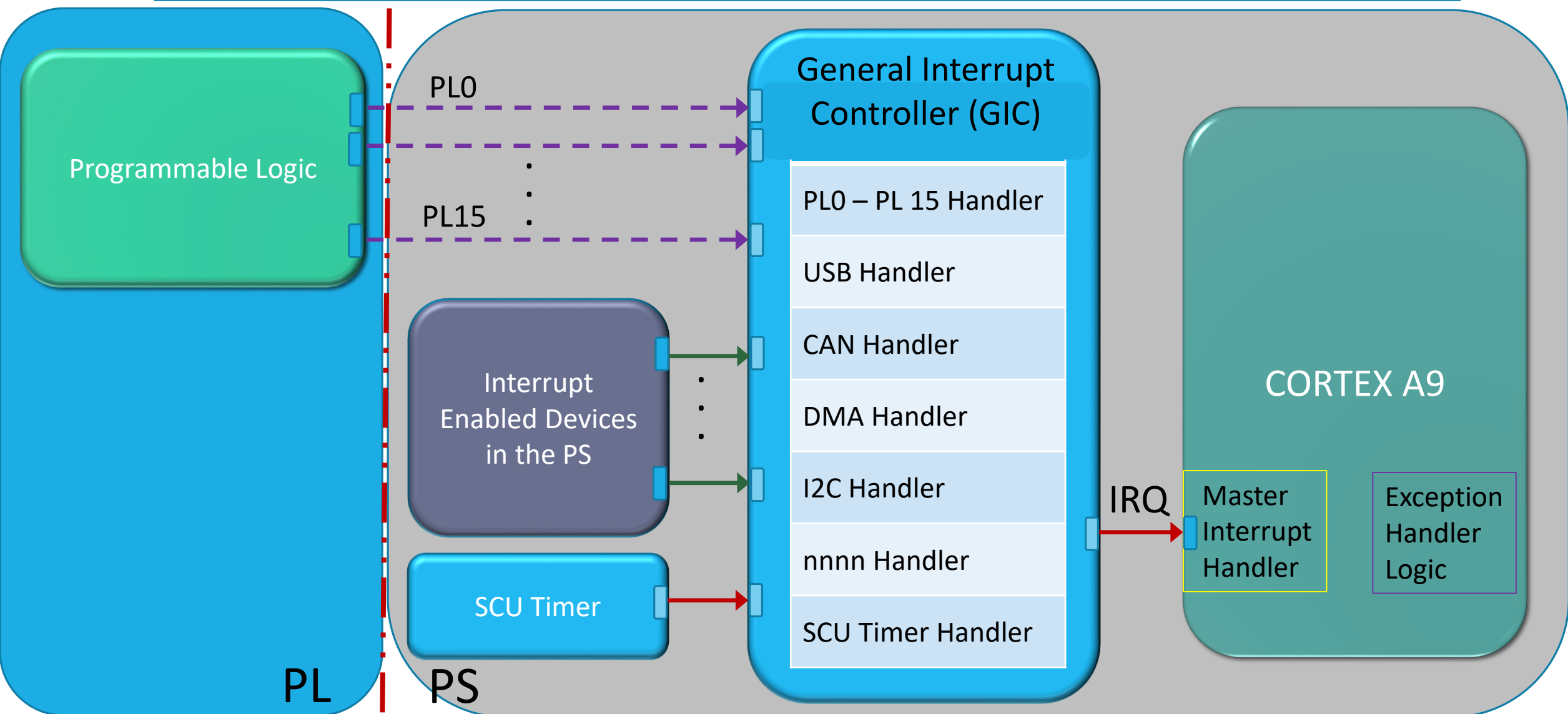
Software Generated Interrupts (SGI)

Each CPU can interrupt itself, the other CPU, or both CPUs using a software generated interrupt (SGI)

❖ There are 16 software generated interrupts
  ❖ An SGI is generated by writing the SGI interrupt number to the ICDSGIR register and specifying the target CPU(s).
  ❖ All SGIs are edge triggered.

CPU Private Peripheral Interrupts (PPI)

Each CPU connects to a private set of five peripheral interrupts

IRQ ID#	Name	PPI#	Type	Description
26:16	Reserved	~	~	Reserved
27	Global Timer	0	Rising edge	Global timer
28	nFIQ	1	Active Low level (active High at PS-PL interface)	Fast interrupt signal from the PL: CPU0: IRQF2P[18] CPU1: IRQF2P[19]
29	CPU Private Timer	2	Rising edge	Interrupt from private CPU timer
30	AWDT{0, 1}	3	Rising edge	Private watchdog timer for each CPU
31	nIRQ	4	Active Low level (active High at PS-PL interface)	Interrupt signal from the PL: CPU0: IRQF2P[16] CPU1: IRQF2P[17]

PPI Includes
- The global timer, private watchdog timer, private timer, and FIQ/IRQ from the PL
- IRQ IDs 16-26 reserved, global timer 27, nFIQ 28, private timer 29, watchdog timer 30, nIRQ 31

Shared Peripheral Interrupts (SPI)

A group of approximately 60 interrupts from various peripherals can be routed to one or both of the CPUs or the PL. The interrupt controller manages the prioritization and reception of these interrupts for the CPUs.

Source	Interrupt Name	IRQ ID#	Status Bits (mpcore Registers)	Required Type	PS-PL Signal Name
PL	PL [2:0]	63:61	spi_status_0[31:29]	Rising edge/ High level	IRQF2P[2:0]
	PL [7:3]	68:64	spi_status_1[4:0]	Rising edge/ High level	IRQF2P[7:3]
Timer	TTC 1	71:69	spi_status_1[7:5]	High level	~
DMAC	DMAC[7:4]	75:72	spi_status_1[11:8]	High level	IRQP2F[27:24]
IOP	USB 1	76	spi_status_1[12]	High level	IRQP2F[7]
	Ethernet 1	77	spi_status_1[13]	High level	IRQP2F[6]
	Ethernet 1 Wake-up	78	spi_status_1[14]	Rising edge	IRQP2F[5]
	SDIO 1	79	spi_status_1[15]	High level	IRQP2F[4]
	I2C 1	80	spi_status_1[16]	High level	IRQP2F[3]
	SPI 1	81	spi_status_1[17]	High level	IRQP2F[2]
	UART 1	82	spi_status_1[18]	High level	IRQP2F[1]
	CAN 1	83	spi_status_1[19]	High level	IRQP2F[0]
PL	PL [15:8]	91:84	spi_status_1[27:20]	Rising edge/ High level	IRQF2P[15:8]

Shared Peripheral Interrupts (SPI)

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	~

Source	Interrupt Name	IRQ ID#
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

Source	Interrupt Name	IRQ ID#
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

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 (ID31)

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		

Processing a Harwdware Interrupt

Detailed Procedure

1. The processor completes the current instruction

2. The processor disables further interrupts, saves the content of the status register, and saves the content of the program counter, which is the next address of normal program execution

3. The processor transfers execution to the top-level exception handler by loading the program counter with the predetermined exception handler address

4. The exception handler saves the contents of the processor's registers

5. The exception handler determines the cause of the interrupt

6. The exception handler calls the proper ISR according to the cause

Detailed Procedure (cont')

7. The ISR clears the associated interrupt condition

8. The ISR performs the designated device-specific function and then returns the control to the exception handler

9. The exception handler restores the contents of the processor's registers and enables future interrupts

10. The exception handler exits by issuing the `eret` (exception return) instruction

11. The processor executes the `eret` instruction, which restores the status register and loads the previous saved return address to the program counter

12. The processor resumes the normal program execution from the interrupted point

Interrupt Handling in Cortex-A9 Processor

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 enable in the Zynq PS
- They must be enabled in software
  - Use peripheral's API
- Interrupt service routine must be developed
  - 'C' defined functions and user code
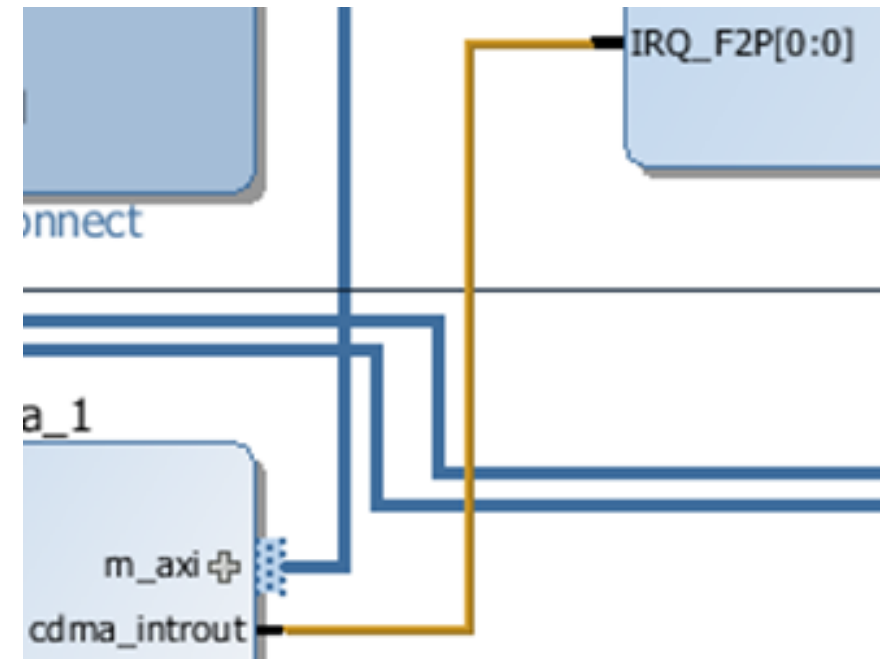
'C' Code for Hardware Interrupt

Pseudo Code for Hardware Interrupt

```c
int main (void)
{

  assign_interrupt_handler(&my_doorbell, front_door_interrupt_handler);

  while(1)
        {
         print("I'm watching TV\n\r");
        }
return(0);
}


void front_door_interrupt_handler (void)
        {
         print("My pizza has arrived!\n\r");
        }
```

USING INTERRUPTS IN Xilinx SDK

Interrupts are supported and can be implemented on a bare-metal system using the standalone board support package (BSP) within the Xilinx Software Development Kit (SDK).

The BSP contains a number of functions that greatly ease this task of creating an interrupt-driven system. They are provided within the following header files:

- *Xparameters.h* – This file contains the processor's address space and the device IDs.
- *Xscugic.h* – This file holds the drivers for the configuration and use of the GIC.
- *Xil_exception.h* – This file contains exception functions for the Cortex-A9.

Requirements for an Interrupt Routine Service

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

Hardware Interrupt - Example Code

SCU Timer Interrupt: Detailed Study

The Zynq documentation states that an interrupt is generated by the timer whenever it reaches zero, so we can use this feature to generate a hardware.

To configure the ARM processor to be able to answer to an interrupt request from the SCU Timer the following steps have to be implemented:

1. Enable interrupt functionality on the SCU Timer

2. Enable interrupts on the ARM Processor

3. Set up the Interrupt Controller

Main Code for the SCU Timer Interrupt – Step 1

```c
// Declare two structs related to the Timer
XScuTimer my_Timer;              // Timer instanc
XScuTimer_Config *Timer_Config; // timer's config information

// Declare two structs related to the GIC
XScuGic my_Gic;                  // General Interrupt Controller (GIC) instance
XScuGic_Config *Gic_Config;      //  GIC's config information

    // ---------------- end declarations part ------------------//

// Look up the config information for the GIC
Gic_Config = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);

// Initialise the GIC using the config information
Status = XScuGic_CfgInitialize(&my_Gic, Gic_Config, Gic_Config->CpuBaseAddress);

// Look up the the config information for the timer
Timer_Config = XScuTimer_LookupConfig(XPAR_PS7_SCUTIMER_0_DEVICE_ID);

// Initialise the timer using the config information
Status = XScuTimer_CfgInitialize(&my_Timer, Timer_Config, Timer_Config->BaseAddr);

// Initialize Exception handling on the ARM processor
Xil_ExceptionInit();
```

Step 2

Next we need to initialize the exception handling features on the ARM processor. This is done using a function call from the "xil_exception.h" header file.

```c
// Initialize Exception handling on the ARM processor
Xil_ExceptionInit();
```

Step 3

When an interrupt occurs, the processor first has to interrogate the interrupt controller to find out which peripheral generated the interrupt. Xilinx provide an interrupt handler to do this automatically, and it is called "XScuGic_InterruptHandler".

```
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
        (Xil_ExceptionHandler)XScuGic_InterruptHandler, &my_Gic);
```

Step 4

We now need to assign our interrupt handler, which will handle interrupts for the timer peripheral. In our case, the handler is called "my_timer_interrupt_handler".

It's connected to a unique interrupt ID number which is represented by the "XPAR_SCUTIMER_INTR" ("xparameters.h").

```c
// Assign (connect) the interrupt handler (that you wrote) for our Timer
Status = XScuGic_Connect(&my_Gic, XPAR_SCUTIMER_INTR,
        (Xil_ExceptionHandler)my_timer_interrupt_handler, (void *)&my_Timer);
```

Step 5

The next task is to enable the interrupt input for the timer on the interrupt controller. Interrupt controllers are flexible, so you can enable and disable each interrupt to decide what gets through and what doesn't.

```
// Enable the interrupt *input* on the GIC for the timer's interrupt
XScuGic_Enable(&my_Gic, XPAR_SCUTIMER_INTR);
```

Step 6

Next, we need to enable the interrupt output on the timer.

```
// Enable the interrupt *output* in the timer.
XScuTimer_EnableInterrupt(&my_Timer);
```

Step 7

Finally, we need to enable interrupt handling on the ARM processor. Again, this function call can be found in the "xil_exception.h" header file.

```
// Enable interrupts in the ARM Processor
Xil_ExceptionEnable();
```

Interrupt Steps in the Zynq
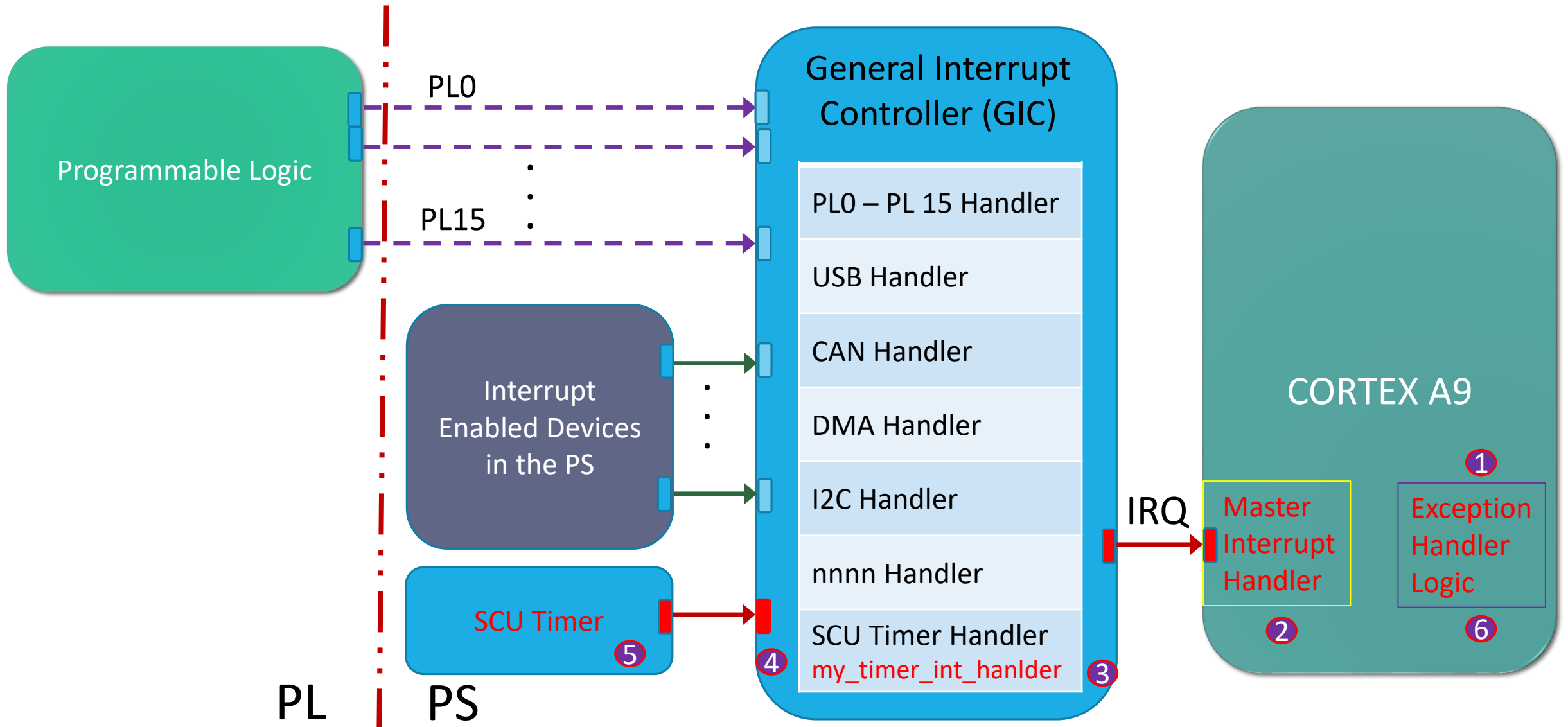
IRS - Interrupt Handler - 1

```c
// ------------------------------------------------//
// IRS: Interrupt Routine Service//
// ------------------------------------------------//
static void my_timer_interrupt_handler(void *CallBackRef)
{
```

IRS - Interrupt Handler - 2

We can declare a local copy of the "*my_Timer*" instance, and assign it the information provided by the "*CallBackRef*"

```
// The Xilinx drivers automatically pass an instance of
// the peripheral which generated in the interrupt into this
// function, using the special parameter called "CallBackRef".
// We will locally declare an instance of the timer, and assign
// it to CallBackRef.
XScuTimer *my_Timer_LOCAL = (XScuTimer *) CallBackRef;
```

we now want to check to make sure that the timer really did generate this interrupt.

```
// Here we'll check to see if the timer counter has expired.
// Technically speaking, this check is not necessary.
// We know that the timer has expired because that's the
// reason we're in this handler in the first place!
// However, this is an example of how a callback reference
// can be used as a pointer to the instance of the timer
// that expired.
if (XScuTimer_IsExpired(my_Timer_LOCAL))
{
```

IRS Interrupt Handler - 4

The last task is to clear the interrupt condition in the timer peripheral. If we don't do this, then the interrupt condition will still be active.

```
// Clear the interrupt flag in the timer, so we don't service
// the same interrupt twice.
XScuTimer_ClearInterruptStatus(my_Timer_LOCAL);
```

Interrupt Considerations

Interrupt 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
- A practice to be avoided

Code portability
- Are operating system hooks needed?

ISR Tips and Tricks

o **Keep the code short and simple; ISRs can be difficult to debug**

o **Do not allow other interrupts while in the ISR**
  o This is a system design consideration and not a recommended practice
  o The interrupt priority, when using an interrupt controller, is determined by the hardware hookup bit position on the interrupt input bus

o **Time is of the essence!**
  o Spend as little time as possible in the ISR
  o Do not perform tasks that can be done in the background
  o Use flags to signal background functions

o **Make use of provided interrupt support functions when using IP drivers**

o **Do not forget to enable interrupts when leaving the handler/ISR**

How to write a Good ISR

o **Keep the interrupt handler code brief (in time)**
  o Avoid loops (especially open-ended while statements)

o **Keep the interrupt handler simple**
  o Interrupt handlers can be very difficult to debug

o **Disable interrupts as they occur**
  o Re-enable the interrupt as you exit the handler

o **Budget your time**
  o Interrupts are never implemented for fun—they are required to meet a specified response time
  ◦ Predict 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

ISR: Precautions

Despite its simplistic appearance, scheduling with interrupt can be very involved and may complicate the software development.

❖ ISRs are the most error-prone portion in embedded software. Since an ISR can be invoked any time during program execution, it is difficult to develop, debug, test, and maintain.

❖ ISRs complicate the timing analysis of the main polling loop. We must consider the frequency of occurrence and duration of the ISRs to ensure that normal tasks within the polling loop can be executed in a timely manner. When multiple interrupts are used, an ISR may effect and interfere with other ISRs and the complexity multiplies.

To ease the problem, it is good practice to *keep ISRs simple and fast.* We should use an ISR to perform the most essential functionalities, such as setting event flags, updating counters, sending a message to a queue, etc., and leave the non-critical computation to a task within the main polling loop. This simplifies the ISR development and timing analysis.

Performance

The top-level exception handler timing includes three parts

✓ *Hardware interrupt latency:* from the time when an interrupt is asserted to the time when the processor executes the instruction at the exception address.

✓ *Response time:* from the time when an interrupt is asserted to the time when the processor executes the first instruction in the ISR. It includes the time for the exception handler to determine the cause of the interrupt and save the content of register file.

✓ *Recovery time:* the time taken from the last instruction in the ISR to return to normal processing

ISR performance enhancement

Some techniques are:

o Adjust the buffer size or implement advanced features, such as double buffering.

o Utilize DMA (direct memory access) controllers to transfer data between memory modules.

o Implement custom hardware accelerator to perform computation intensive processing.

The SoC platform introduces a new dimension of flexibility. We can examine the performance criteria and design complexity and find the best trade-off between software and hardware resources.

Apendix

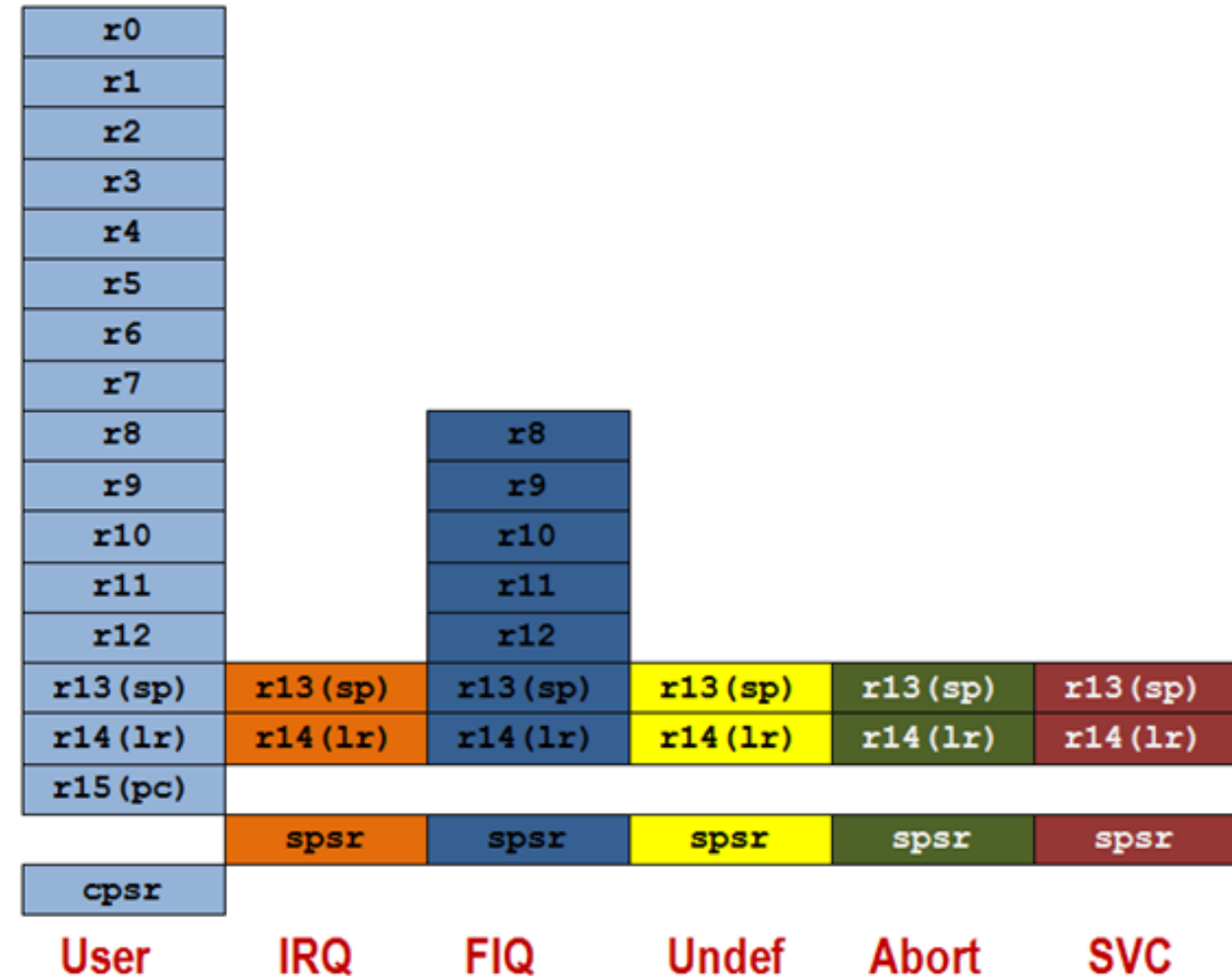# Cortex A9 – Modes and Registers

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

Mode	Description	
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**

Exception Modes bracket spans Supervisor, FIQ, IRQ, Abort, Undef.

Cortex A9 – Modes and 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					

Application Software for HI

1. Set a direction control for the AXI GPIO pin as an input pin, which is connected with BTNU push button on the board. The location is fixed via LOC constraint in the user constraint file (UCF) during system creation.

2. Initialize the AXI TIMER module with device ID 0.

3. Associate a timer callback function with AXI timer ISR.

4. This function is called every time the timer interrupt happens. This callback switches on the *LED 'LD9'* on the board and sets the interrupt flag.

5. The *main()* function uses the interrupt flag to halt execution, wait for timer interrupt to happen, and then restarts the execution.

6. Set the reset value of the timer, which is loaded to the timer during reset and timer starts.

7. Set timer options such as Interrupt mode and Auto Reload mode.
   Zedboard_refdoc_Vivado_2014-2_cs.pdf

Application Software for HI

8. Initialize the PS section GPIO.

9. Set the PS section GPIO, channel 0, pin number 10 to the output pin, which is mapped to the MIO pin and physically connected to the LED 'LD9' on the board.

10. Set PS Section GPIO channel number 2 pin number 0 to input pin, which is mapped to PL side pin via the EMIO interface and physically connected to the BTNR push button switch.

11. Initialize Snoop control unit Global Interrupt controller. Also, register Timer interrupt routine to interrupt ID '91', register the exceptional handler, and enable the interrupt.

12. Execute a sequence in the loop to select between AXI GPIO or PS GPIO use case via serial terminal.