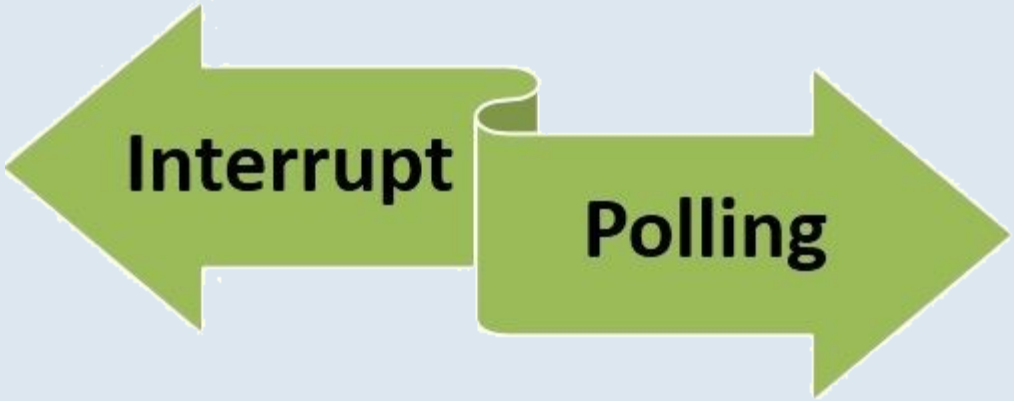
Part 2: Polling vs. Interrupts



Data transfer from PL to PS

- How can the PS know when new data is available in the PL?
- Classic CPU – peripheral interaction: **polling** and **interrupt**

- Analogy: pizza order at home

Polling



Interrupt



Polling

- Software routine used to identify a peripheral requesting service
- At certain intervals: check if new data is available

↑ Efficient when events arrives very often

↑ It requires no special hardware

↑ Quick response time (less 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 one 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

Interrupt

- **Special** software routine that attends peripherals in need for attention

↓ **Inefficient** when events arrives very often

↓ **It does** require special hardware

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

↑ It **does not take 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

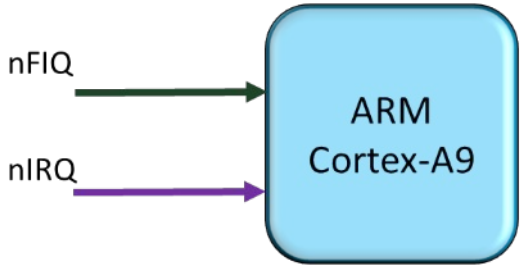
↑ priority of the peripherals is determined by the hardware

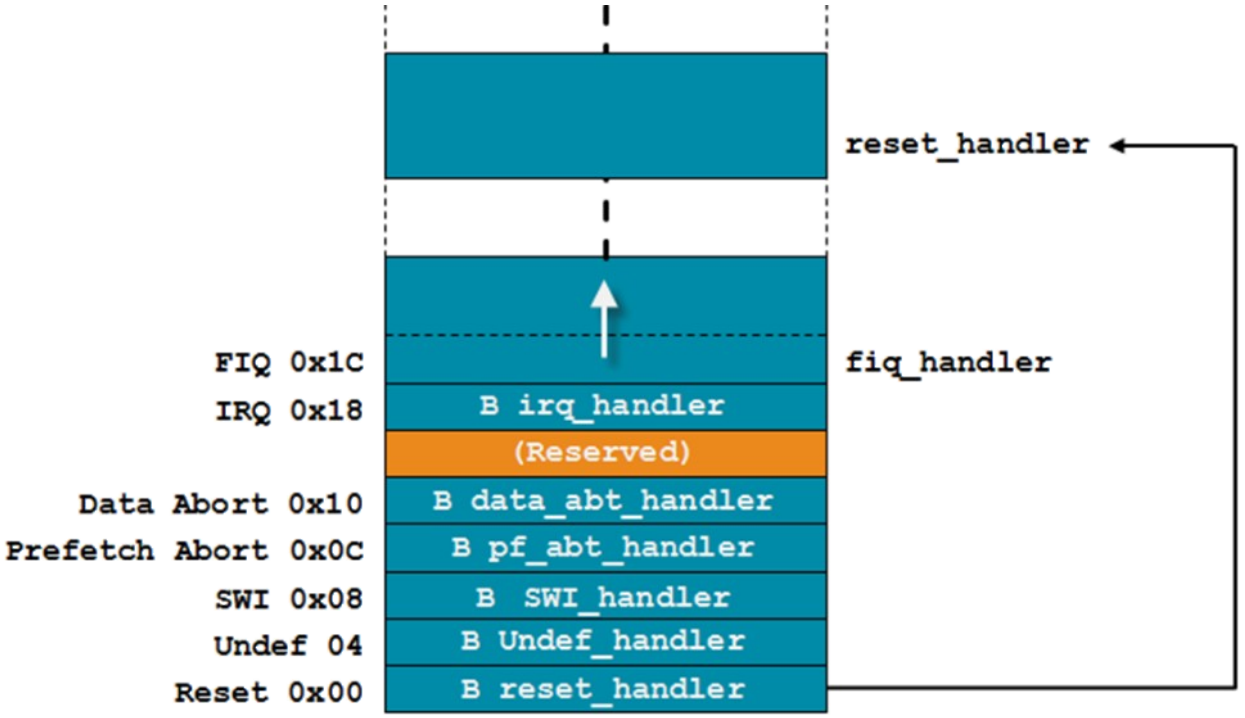
Interrupts in ARM Cortex-A9

CPU interrupts are handled as exceptions

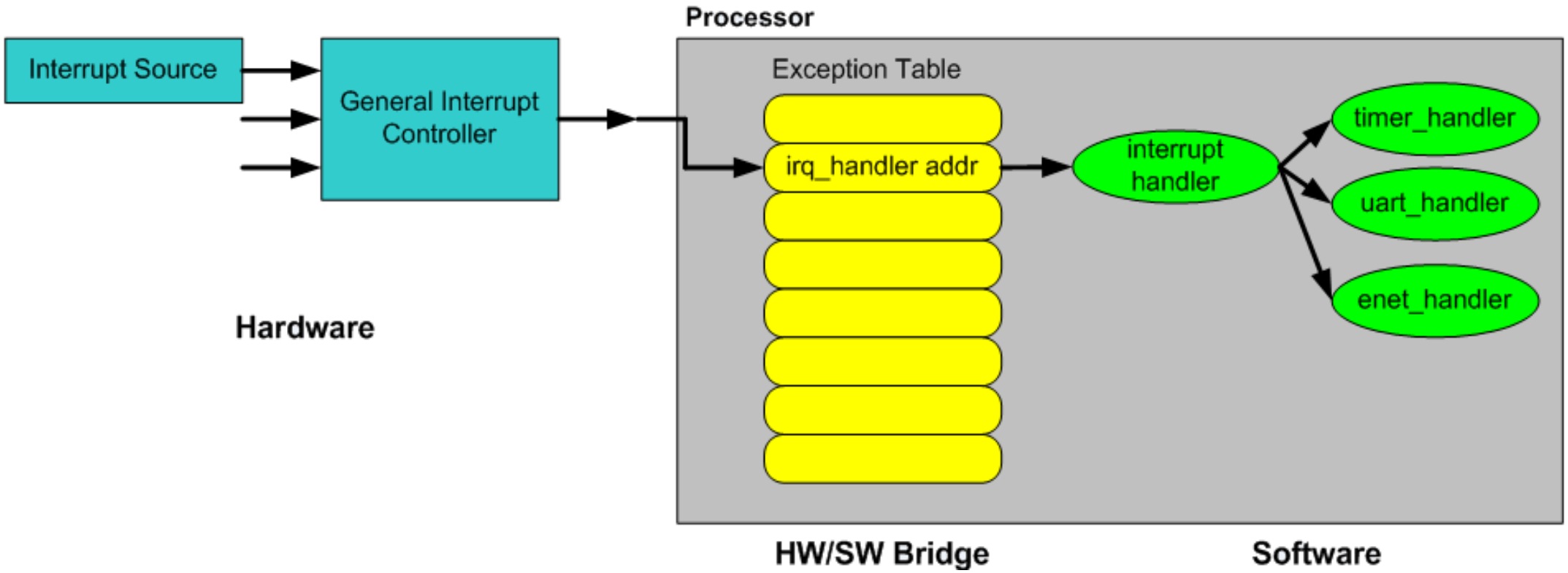
- Two levels of interrupts:

1. nFIQ: secure sources, serviced first
2. nIRQ: secure/non-secure sources

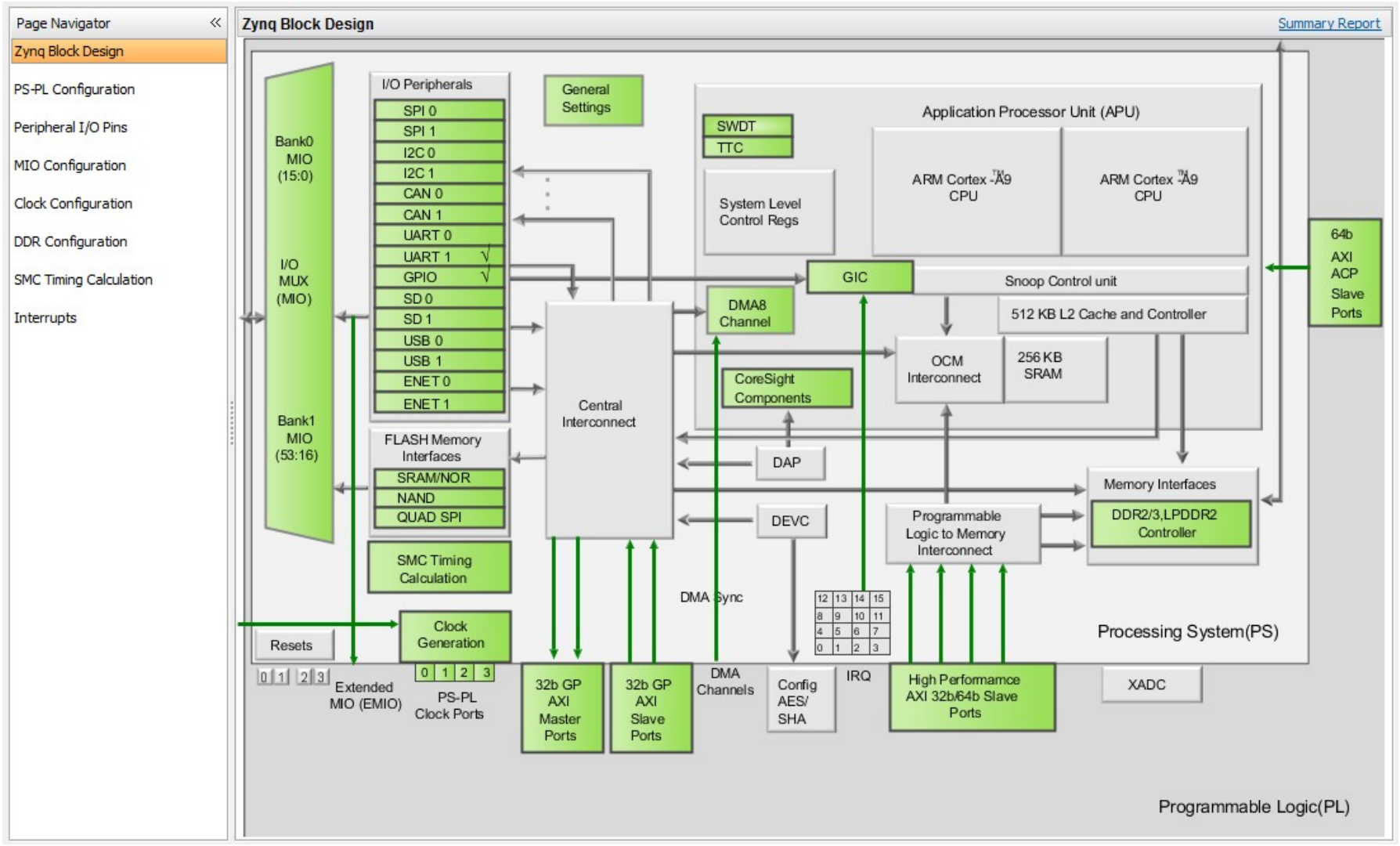




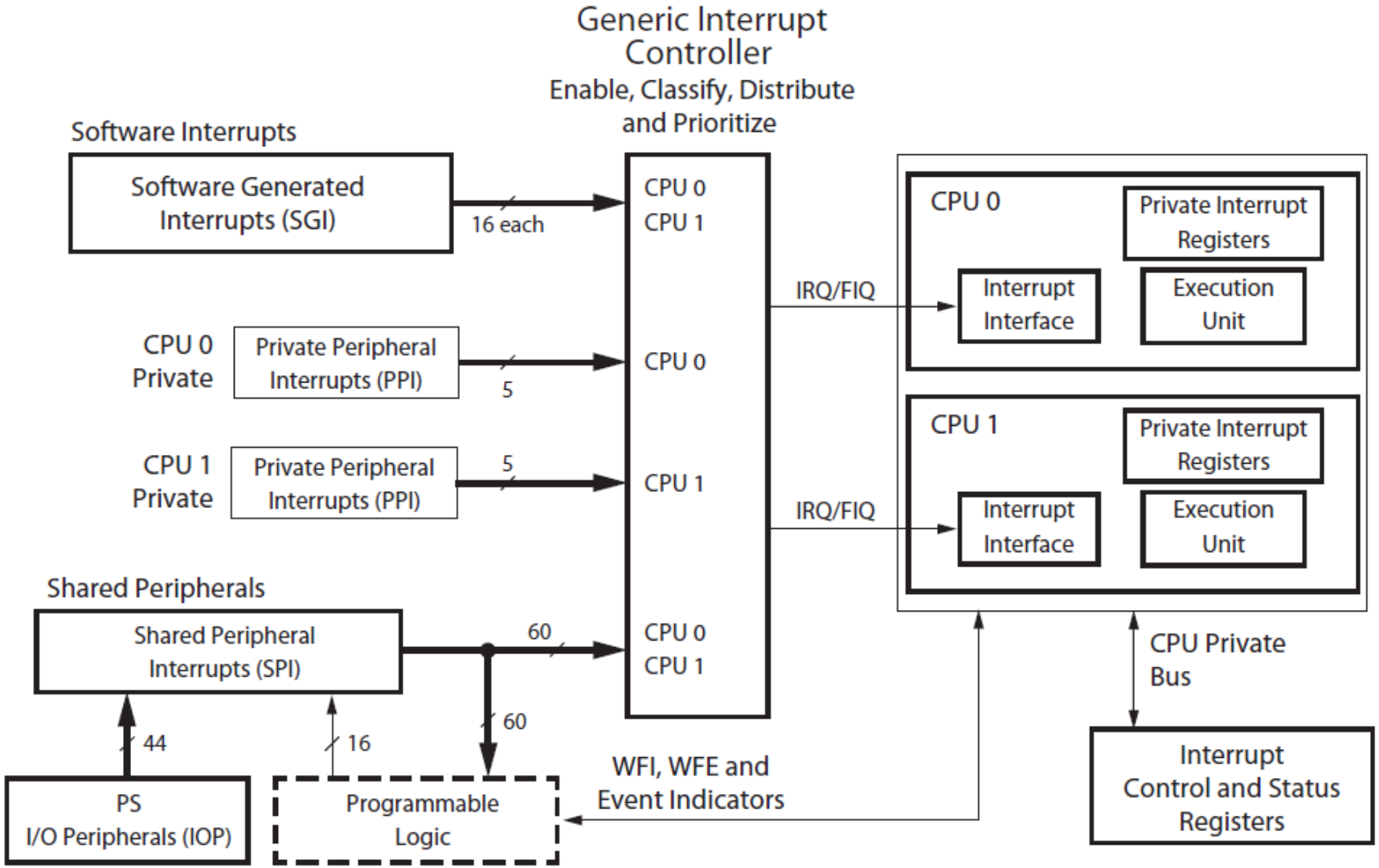
Interrupt Flow



Generic Interrupt Controller (GIC)



GIC – Block Diagram



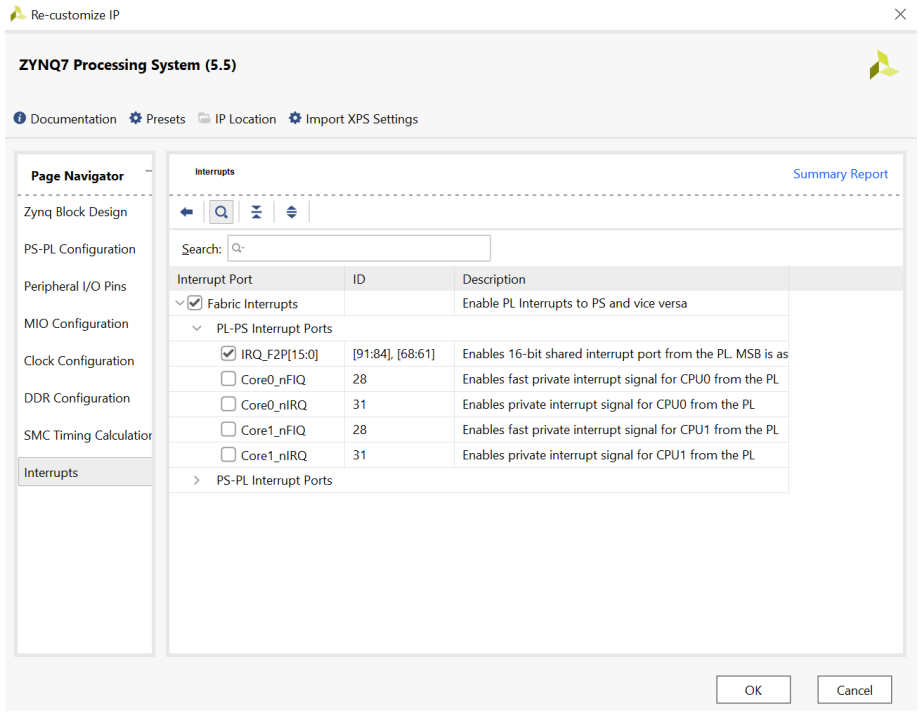
Shared Peripheral Interrupts (SPI)

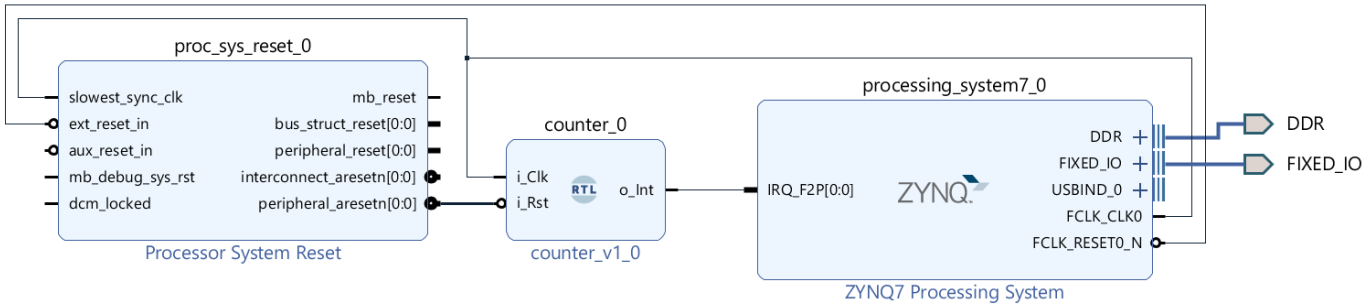
Source	Interrupt Name	IRQ ID#	Status Bits (mpcore Registers)	Required Type	PS-PL Signal Name
PL	PL [2:0]	63:61	spi_status_0[31:29]	Rising edge/ High level	IRQF2P[2:0]
	PL [7:3]	68:64	spi_status_1[4:0]	Rising edge/ High level	IRQF2P[7:3]
Timer	TTC 1	71:69	spi_status_1[7:5]	High level	~
DMAC	DMAC[7:4]	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]

Enabling Interrupts on Zynq-7000 – Vivado

Create a **block design** with a ZYNQ7 PS and a Processor System Reset

- The IRQ_F2P input of the PS is not enabled by default
- To enable it:
  - Double-click on the PS block
  - Select the tab “Interrupts”
  - Tick the “Fabric Interrupts” box
  - Tick the *IRQ_F2P[15:0]* box
- It can then be driven by (for instance) a timer





Enabling Interrupts on Zynq-7000 – Vitis

- Drivers to interface/configure the Global Interrupt Controller provided by **xscugic.h**
- Templates provided in Vitis (under BSP: Board Support Package)
- Easy to configure (see code)

```
// Interrupt header files
#include "xscugic.h"
#include "xil_exception.h"

// IRQ_F2P[0:0]
#define INTC_INTERRUPT_ID_0 61

// Function declaration
int setup_interrupt_system();
void isr0 (void *intc_inst_ptr);

// Global instance of interrupt controller
static XScuGic intc;
```

```
// sets up the interrupt system and enables interrupts for IRQ_F2P[0:0]
int setup_interrupt_system() {

    int result;
    XScuGic *intc_instance_ptr = &intc;
    XScuGic_Config *intc_config;

    // get config for interrupt controller
    intc_config = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
    if (NULL == intc_config) {
        return XST_FAILURE;
    }

    // initialize the interrupt controller driver
    result = XScuGic_CfgInitialize(intc_instance_ptr, intc_config, intc_config->CpuBaseAddress);

    if (result != XST_SUCCESS) {
        return result;
    }

    // set the priority of IRQ_F2P[0:0] to 0xA0 (highest 0xF8, lowest 0x00) and a trigger for a rising edge 0x3.
    XScuGic_SetPriorityTriggerType(intc_instance_ptr, INTC_INTERRUPT_ID_0, 0xA0, 0x3);

    // connect the interrupt service routine isr0 to the interrupt controller
    result = XScuGic_Connect(intc_instance_ptr, INTC_INTERRUPT_ID_0, (Xil_ExceptionHandler)isr0, (void *)&intc);

    if (result != XST_SUCCESS) {
        return result;
    }

    // enable interrupts for IRQ_F2P[0:0]
    XScuGic_Enable(intc_instance_ptr, INTC_INTERRUPT_ID_0);

    // initialize the exception table and register the interrupt controller handler with the exception table
    Xil_ExceptionInit();

    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT, (Xil_ExceptionHandler)XScuGic_InterruptHandler, intc_instance_ptr);

    // enable non-critical exceptions
    Xil_ExceptionEnable();

    return XST_SUCCESS;
}
```

Enabling Interrupts on Zynq-7000 – Vitis

- Drivers to interface/configure the Global Interrupt Controller provided by **xinterrupt_wrap.h**
- Templates provided in Vitis (under BSP: Board Support Package)
- Easy to configure (see code)

```
/*
   Interrupt Controller Device ID for IRQ_F2P[0:0] can be found in the SPI interrupts table:
   https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM/Shared-Peripheral-Interrupts-SPI?tocId=LGXkQnG10uJNPCT_l2ZhDQ&section=XREF_72962_PS_and_PL_Shared
*/
#define INTC_DEVICE_INT_ID  61 /* original ID was 0x0E for a software interrupt from the CPU */

#define IRQ_TYPE_EDGE_RISING    0x01U  /**< Interrupt on Rising edge */
#define IRQ_TYPE_EDGE_FALLING   0x02U  /**< Interrupt Falling edge */
#define IRQ_TYPE_EDGE_BOTH  0x03U  /**< Interrupt on both edges */
#define IRQ_TYPE_LEVEL_HIGH 0x04U  /**< Interrupt on high level */
#define IRQ_TYPE_LEVEL_LOW  0x08U  /**< Interrupt on low level */
```

```
void DeviceDriverHandler(void *CallbackRef)
{
    /*
     * Indicate the interrupt has been processed using a shared variable
     */
    xil_printf(ctrl1: "Processing interrupt!\r\n");
    InterruptProcessed = TRUE;
}
```

```
u32 IntrId;
UINTPTR IntcBaseAddr;

/* Here we define what type of interrupt we are going to configure,
   the second parameter defines the edge/level type
   the third parameter defines the SGI / *SPI* / PPI interrupt

   Read the documentation:
   TRM: https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM/Interrupts?tocId=GjIIAQFElp9aCgqYQifuZQ
   API: https://docs.amd.com/r/en-US/oslib_rm/Interrupt-Wrapper-APIs?tocId=NoHhBZrQRMb42VMlYcrz6g

*/
Status = XGetEncodedIntrId(LegacyIntrId: INTC_DEVICE_INT_ID,TriggerType: IRQ_TYPE_EDGE_RISING,IntrType: XINTR_IS_SPI,IntcType: XINTC_TYPE_IS_SCUGIC, IntrId: &IntrId);

if (Status != XST_SUCCESS) {
    return XST_FAILURE;
    }

IntcBaseAddr = XGetEncodedIntcBaseAddr(XPAR_XSCUGIC_0_BASEADDR, XINTC_TYPE_IS_SCUGIC);

Status =  XSetupInterruptSystem(DriverInstance: &InterruptController, IntrHandler: DeviceDriverHandler, IntrId, IntcParent: IntcBaseAddr, Priority: XINTERRUPT_DEFAULT_PRIORITY);

if (Status != XST_SUCCESS) {
    return XST_FAILURE;
    }
```

Way Forward

- Interrupts will be explored in hands-on 3

- **Vivado**: you will use a configurable timer to trigger interrupts from the PL to the PS
- **Vitis**: you will configure the interrupt controller and create a custom ISR (Interrupt Service Routine)

- Idea: every X milliseconds, an interrupt is used to bring data from the PL to the PS
- In the hands-on: state of push buttons

```
// interrupt service routine for IRQ_F2P[0:0]
void isr0 (void *intc_inst_ptr)
{
    xil_printf("Serving interrupt\n\r");
}
```

