Using Interrupts

OBJECTIVES

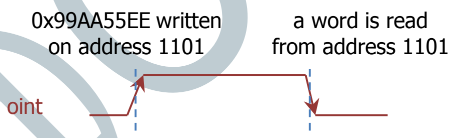
- Implement an embedded project (PS + PL) where a hardware component inside the PL can generate an interrupt to the processor (Vivado 2019.1).
- Learn to develop software routines to handle PL interrupts.

PL INTERRUPTS - DOCUMENTATION

- UG585: Zynq-7000 AP SoC Technical Reference Manual – Chapter 7.
- XPLANATION: FPGA 101, "How to Use Interrupts on the Zynq SoC", Xcell Journal, Second Quarter 2014.

PL INTERRUPT TEST

- Test Project: AXI-4 Full Pixel Processor peripheral with interrupt signal. This peripheral (from Unit 7) was modified by including an interrupt signal *oint* and by updating the FSM@ S_AXI_ACLK.
- A PL interrupt is forced by writing onto a specific register in the AXI4-Full Peripheral. This is not a software interrupt. The process goes as follows:
  - ✓ Write a specific word (`0x99AA55EE`) on address `1101` on the AXI4-Full Pixel processor peripheral. This will assert the interrupt signal *oint*.
  - ✓ Let the software routine wait until the PS detects the interrupt signal.
  - ✓ When the interrupt signal is detected, the ISR de-asserts the interrupt signal (and prints a message) by reading a word from address `1101`.


0x99AA55EE written on address 1101
a word is read from address 1101
oint

CUSTOM AXI4-FULL PERIPHERAL: PIXEL PROCESSOR WITH INTERRUPT OUTPUT

CONSIDERATIONS

- We will use the Pixel Processor with NC=4, NI=NO=8.
- AXI4-Full Peripheral: Custom FIFO-based interface that includes a user-generated output signal *oint*. All writes/reads to any of the 16-word memory positions are treated equally (writing/reading on the FIFO). The exception is the write/read at register 13, which is used to assert/de-assert the interrupt signal.
- List of files to use:
  - ✓ `mypixfullintr_v1_0.vhd`: AXI4-Full Peripheral (top file) with interrupt output.
  - ✓ `mypixfullintr_v1_0_S00_AXI.vhd`: AXI4-Full Interface description.
  - ✓ `myAXI_IP.vhd, my_AXI_fifo.vhd, my_gen_pulse_sclr.vhd`: Ancillary files for the AXI4-Full Peripheral.
  - ✓ `pixfull_rp.vhd`: wrapper file for the Pixel Processor IP. Here, we can modify the parameter F (1..5).
  - ✓ `pixfull_fifointf.vhd`: top file for the Pixel Processor IP.
  - ✓ `LUT_group.vhd, LUT_NItoNO.vhd, LUTNIto1.vhd, pack_xtras.vhd`.
  - ✓ `LUT_values8to8.txt`: LUT values.
- AXI4-Full Pixel Processor Peripheral (we make `S_AXI_CLK=CLK_FX`) with interrupt output *oint*: In Figure 1, see the circuit that generates *oint*. To assert and de-assert this interrupt signal, we need to write and read to/from address `1101`. Writing on address `1101` still writes data on the iFIFO, and the reading retrieves resulting data from oFIFO.

IP GENERATION

- Create a new Vivado project: `pixfull_dr_intr_sys`
  - ✓ Make sure the default language is VHDL, so that the system wrapper and template files are created in VHDL.
  - ✓ At Default Part, go to Boards, and select the **Zybo** (or **Zybo Z7-10**) board.
- From the menu bar, select **Tools → Create and Package IP**. A new Vivado project will open.
  - ✓ Create a new AXI4 Peripheral. Name: `mypixfullintr`. Location: `/ip_repo`.
    Peripheral Repositories tip: To add a previously-generated IP into a new project, go to: Project Settings → IP → IP repositories and point to the associated repository folder.
  - ✓ Add Interface: Full, 32 bits. Interface Mode: Slave. Memory Size: 64 bytes.
  - ✓ Select Edit IP. A New project appears: look for `<peripheral name>.vhd` and `<peripheral name>_S00_AXI.vhd` files (in this case it will be `mypixfullintr_v1_0.vhd mypixfullintr_v1_0_S00_AXI.vhd`). Modify the project:
    - ▫ Replace these two files with our edited files `mypixfullintr_v1_0_S00_AXI.vhd` and `mypixfullintr_v1_0.vhd`. This is necessary as our peripheral includes an output *oint*.
    - ▫ Add the extra files to the folder `/hdl` in `/ip_repo/mypixfullintr_1.0` and add these source files (including the .txt file) to the Vivado project. *Vivado 2019.1: by default, the files will also be added to the folder `/src`.
  - ✓ There is no need to add external ports as our peripheral does not include external I/Os.
  - ✓ Synthesize (just to double-check everything is ok): You should've simulated the code in a different project.

- ✓ Go to Package IP - mypixfullintr: Identify areas that need refresh:
  - Important: When replacing the top file (`mypixfullintr_v1_0.vhd`), we must make sure that Vivado detects the inclusion of the interrupt output signal (it is not enough to just replace the file in the folder, so you can just add an extra space in the file and save it). Then, you should see that in Package IP, you also need to click on Merge Changes for Ports and Interfaces. This will enable Vivado to add the extra interrupt port.
  - Click on Merge changes from File Group Wizard.
- ✓ Go to Review and Package → Re-Package IP.

- Your custom IP with interrupt output is now ready to be used as an AXI4-Full Peripheral
- You will return to the original Vivado Project.

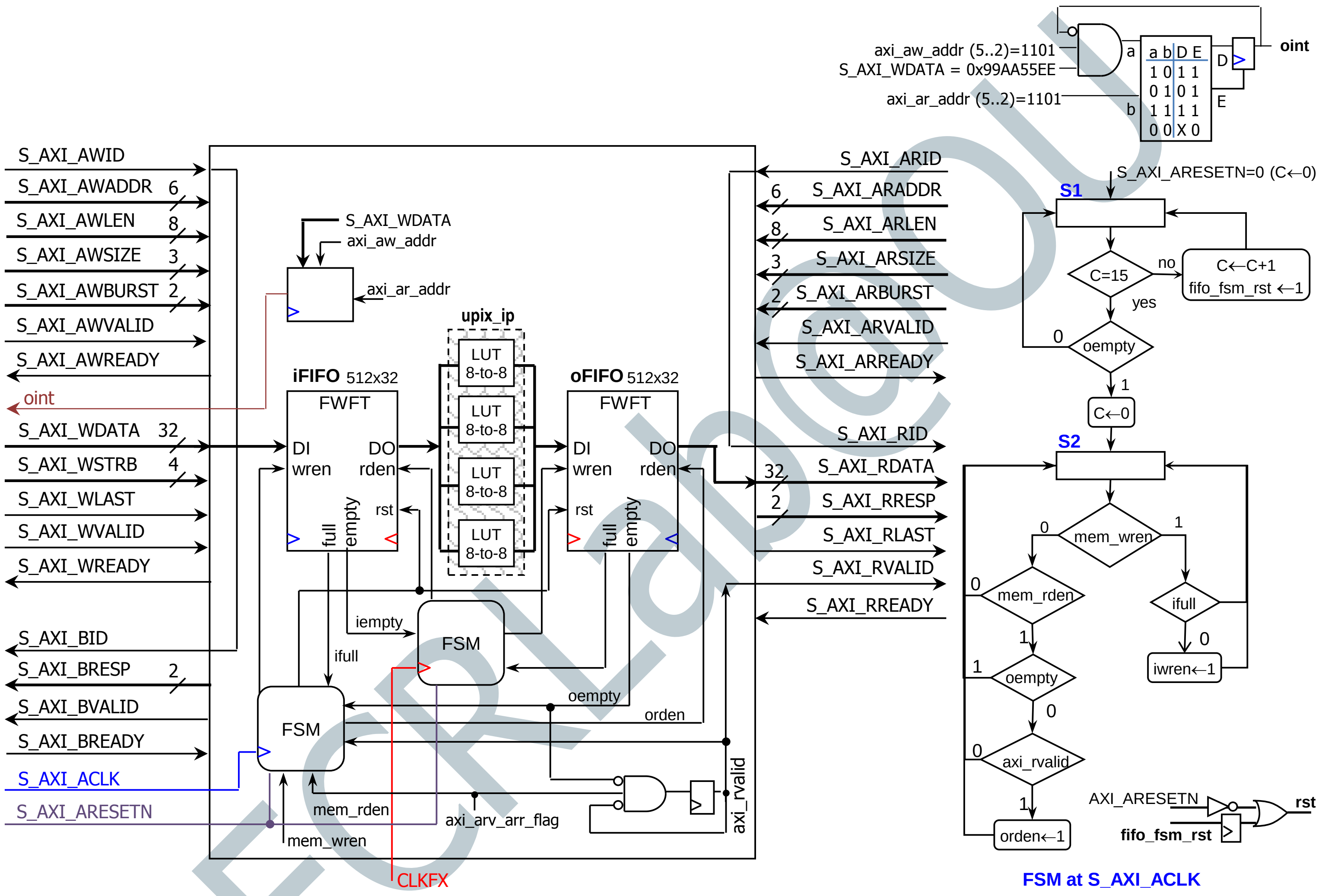


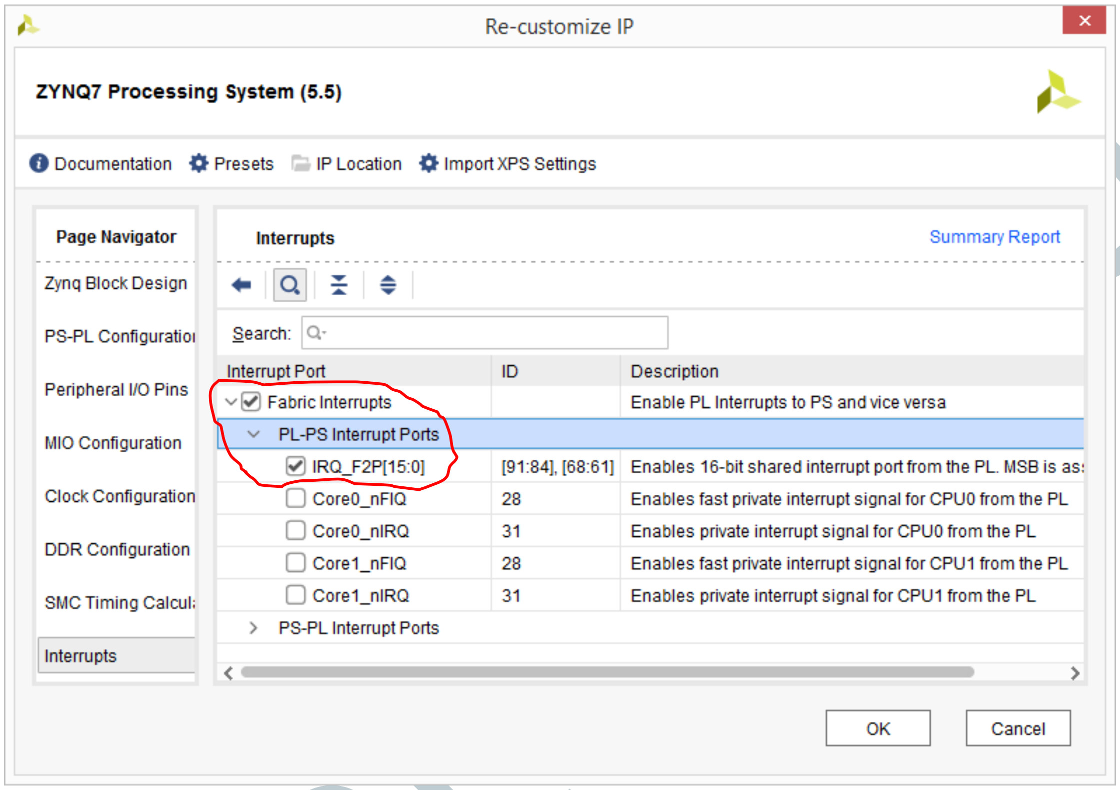
Figure 1. AXI4-Full Pixel Processor Peripheral with interrupt output (*oint*)

CREATING A BLOCK DESIGN PROJECT IN VIVADO

- Click on Create Block Design and instantiate the Zynq PS and the AXI MYPIXFULLINTR peripheral.
- Click on Run Block Automation and Run Connection Automation.
- There is no need to add an `.xdc` file as our peripheral does not use external ports.
- Connect to interrupt signal (*oint*) to the PS:
  - ✓ Go to PS → Interrupts. Check Fabric interrupts. Expand PL-PS Interrupt Ports. Then, check `IRQ_F2P[15..0]` (see Figure 2). This enables the 16 PL interrupts.
  - ✓ Connect the *oint* signal (see Figure 3). By default, if it is only one bit, it will connect to `IRQ_F2P[0..0]`. This is the interrupt `IRQ ID #61`. This has to be properly set up in the software routine as `'XPS_FPGA0_INT_ID'`.
  - ✓ If we have more than one interrupt signal, we must use the *concat* IP.
- Click on Validate Design to ensure the interrupt connection is correct.
- Create the VHDL wrapper (Sources Window → right click on the top-level system design → Create HDL Wrapper)
- Synthesize, implement, and generate the bitstream.

- ✓ An error will be reported at Synthesis. Vivado only copies VHDL files from the IP folder to the embedded project folder (located inside the `/<peripheral name>.srcs/…/ipshared` folder). As a result, the `LUT_NItoNO.vhd` file cannot find the `LUT_values8to8.txt`. We need to place this text file in the same folder as the `LUT_NItoNO.vhd` file.
- ✓ This folder location is available by opening the `LUT_NItoNO.vhd` file. You can find this file in the design structure or via the Vivado error which will point to the `LUT_NItoNO.vhd` file. After copying the .txt file, you can Synthesize again.
- ✓ In general, this procedure is to be followed for any ancillary file (e.g. text file) used by the VHDL files.

- Export hardware (with bitstream) and launch SDK.


Re-customize IP

ZYNQ7 Processing System (5.5)

Documentation | Presets | IP Location | Import XPS Settings

Page Navigator: Zynq Block Design; PS-PL Configuratio; Peripheral I/O Pins; MIO Configuration; Clock Configuration; DDR Configuration; SMC Timing Calcul; Interrupts

Interrupts — Summary Report

Search:

Interrupt Port	ID	Description
✓ Fabric Interrupts		Enable PL Interrupts to PS and vice versa
PL-PS Interrupt Ports		
✓ IRQ_F2P[15:0]	[91:84], [68:61]	Enables 16-bit shared interrupt port from the PL. MSB is as:
☐ 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		

OK | Cancel

Figure 2. PS Customization. 16 PL Interrupt signals enabled

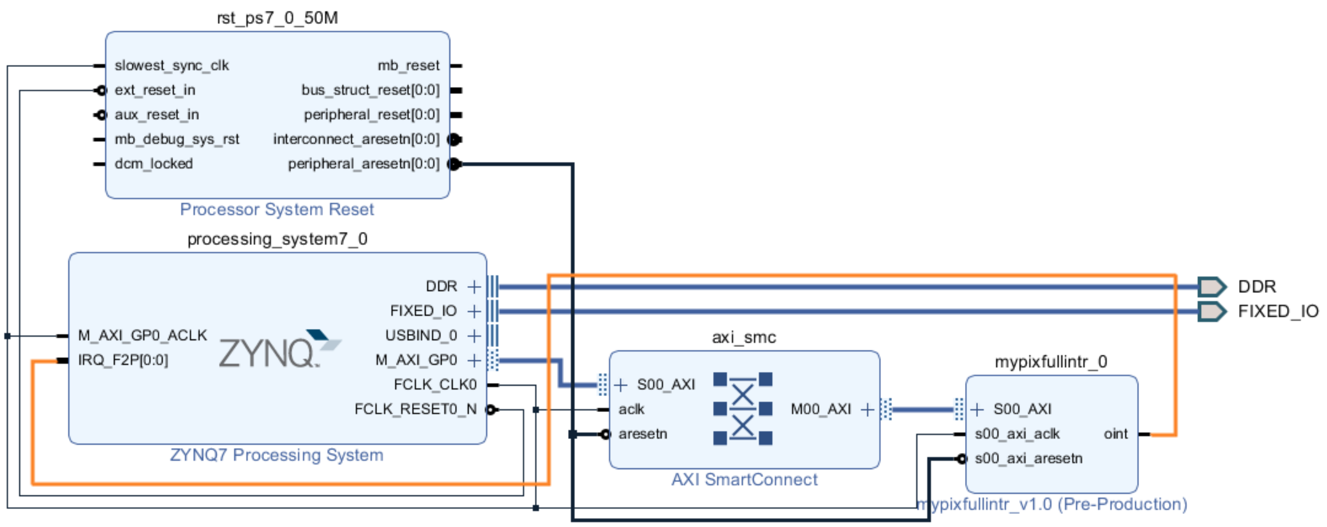


Figure 3. Connecting the *oint* interrupt signal to the PS (signal `IRQ_F2P[0..0]` with `IRQ ID #61`)

TESTING WITH SDK

- See Tutorial Unit 2 for details on how to create and test a software application on SDK.
- Create a new SDK application: pixtest_intr. Then, copy the following file into the `/src` folder: `pix_plintr.c`
- The software routine will write and read 32-bits word to/from the AXI-4 Full peripheral (Pixel Processor with interrupt output) and will verify the assertion, detection, and de-assertion of a PL interrupt.

- Each PL interrupt has its associated Interrupt Service Routine (ISR). We specify this inside the following function: `SetupInterruptSystem (XScuGic *GicPtr)`
  - ✓ We connect the ISR to the Generic Interrupt Controller (via `XScuGic_Connect`) by assigning the `IRQ ID #61` (`XPS_FPGA0_INT_ID`).
  - ✓ We specify a user-defined function to be executed inside the ISR. This is done by specifying the function `DeviceDriverHandler` in `XScuGic_connect`.
  - ✓ In the `DeviceDriverHandler` function, we can specify the instructions we want to be executed once an interrupt hits. In this example, this function reads a word from address `1101` and prints the retrieved data.

- Once the program is compiled, connect the ZYBO (or ZYBO Z7-10) Board to the USB port of your computer.
- Download the bitstream on the PL: Xilinx Tools → Program FPGA
- Go to the SDK Terminal and connect to the proper COM port.
- Select the project you created. Right-click and select Run As → Launch on Hardware (GDB).

- **Testing strategy**:
  - ✓ The software routine writes the following 4 words and retrieve the following words from the peripheral (Pixel Processor with parameter F=1):

Input	Output
0xDEADBEEF	0xEED2DDF7
0xDEAFBEAD	0xEED4DDD2
0xFADEDEAD	0xFDEEEED2
0xFACEB00C	0xFDE6D437

  - ✓ The software routine then writes the word `0x99AA55EE` on address `1101`. This will assert the interrupt. The routine waits until the interrupt is asserted.
  - ✓ Then, the ISR is executed: a message is printed (‘`PL Interrupt occurred`’), and a word is read from address `1101`. The word read should be `0xC6D194F7` (Pixel Processor with F=1).
  - ✓ To double-check that the assertion and de-assertion of the interrupt does not affect the correct operation of our circuit, we write again the words in the table above. We are supposed the retrieve the same output.
  - ✓ At this moment, we have successfully tested the interrupt.

- Note: Do not assert the interrupt and then de-assert it immediately. The PS needs to detect the interrupt first, otherwise the ISR is never activated and the software routine will remain forever waiting for the PL interrupt.

NOTE

- There is another file `pix_dma_plintr.c` that tests both the DMA (with interrupts) and the PL interrupts. It essentially merges the application from Tutorial # 8 (DMA with interrupts) and the software application in this Tutorial # 9 (PL interrupts). Feel free to test this code by creating a new SDK application.