Debugging Using Hardware Analyzer

Introduction

Software and hardware interact with each other in an embedded system. The SDK includes System Debugger as a software debugging tool. The hardware analyzer tool has different types of cores that allow hardware debugging by providing access to internal signals without requiring the signals to be connected to package pins. These hardware debug cores may reside in the programmable logic (PL) portion of the device and can be configured with several modes that can monitor signals within the design. In this lab you will be introduced to the various debugging cores.

Objectives

After completing this lab, you will be able to:

- Add a VIO core in the design
- Use a VIO core to inject stimulus to the design and monitor the response
- *Mark nets* as debug so AXI transactions can be monitored
- Add an ILA core in Vivado
- Perform hardware debugging using the hardware analyzer
- Perform software debugging using the SDK

Procedure

This lab is separated into steps that consist of general overview statements providing information on the subsequent detailed instructions. Follow the (step-by-step) detailed instructions to progress through the

Design Description

In this lab, you will add a custom IP core that performs a simple addition function. The IP has been developed using the IP Packager capability of Vivado and is provided as part of the lab source files. The core has additional ports so that stimuli can be brought in and the response can be monitored. This way the core can be tested independently without using the PS or software application. The following block diagram represents the completed design (**Figure 1**).

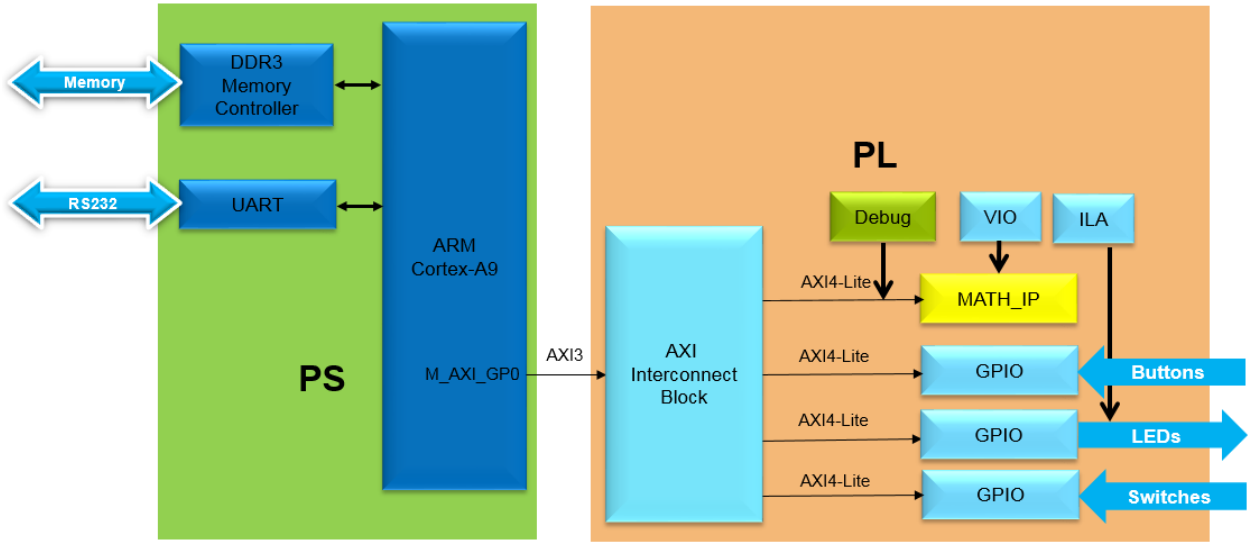
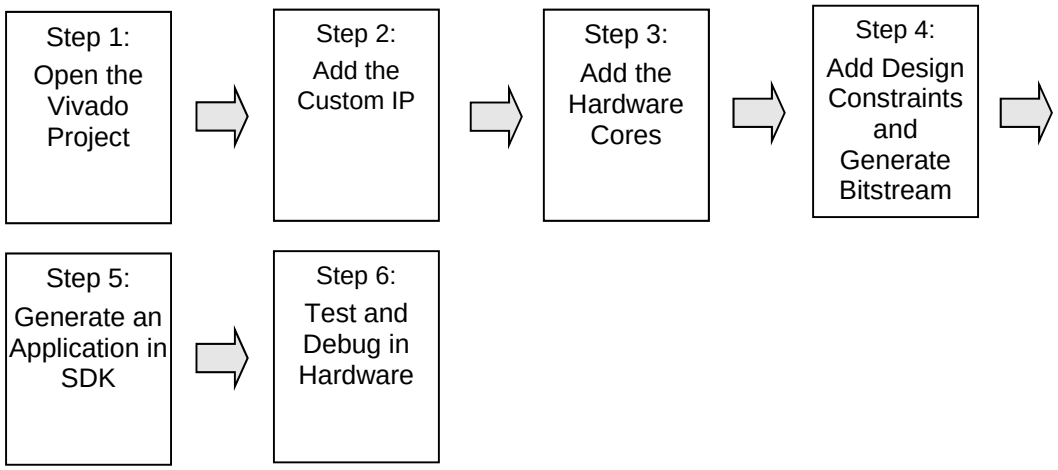


Figure 1. Completed Design

General Flow for this Lab



In the instructions below;
{***sources***} refers to: C:\xup\adv_embedded\2018_2_zynq_sources
{***labs***} refers to: C:\xup\adv_embedded\2018_2_zynq_labs

Board support for PYNQ-Z1 and PYNQ-Z2 are not included in Vivado 2018.2 by default. The relevant zip file need to be extracted and saved to: {Vivado installation}\data\boards\board_files\.

These files can be downloaded from the XUP webpage (http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-adv-embedded-design-zynq.html) where this material is also hosted.

Open the Project — Step 1

1-1. Open the Vivado program. Open the *lab1* project you created in the previous lab or use the *lab1* project from the labsolution directory, and save the project as lab2. Set Project Settings to point to the IP repository provided in the *sources* directory.

1-1-1. Start Vivado if necessary and open either the lab1 project (lab1.xpr) you created in the previous lab or the lab1 project in the labsolutions directory using the **Open Project** link in the Getting Started page.

1-1-2. Select **File > Project > Save As …** to open the *Save Project As* dialog box. Enter **lab2** as the project name. Make sure that the *Create Project Subdirectory* option is checked, the project directory path is {labs} and click **OK**.

This will create the lab2 directory and save the project and associated directory with lab2 name.

1-1-3. Click **Settings** in the *Flow Navigator* pane.

1-1-4. Expand **IP** in the left pane of the *Project Settings* form and select **Repository**.

1-1-5. Click on the *plus* button of the IP Repositories panel, browse to **{sources}\lab2\math_ip** and click **Select**.

The directory will be scanned and one IP will be detected and reported.

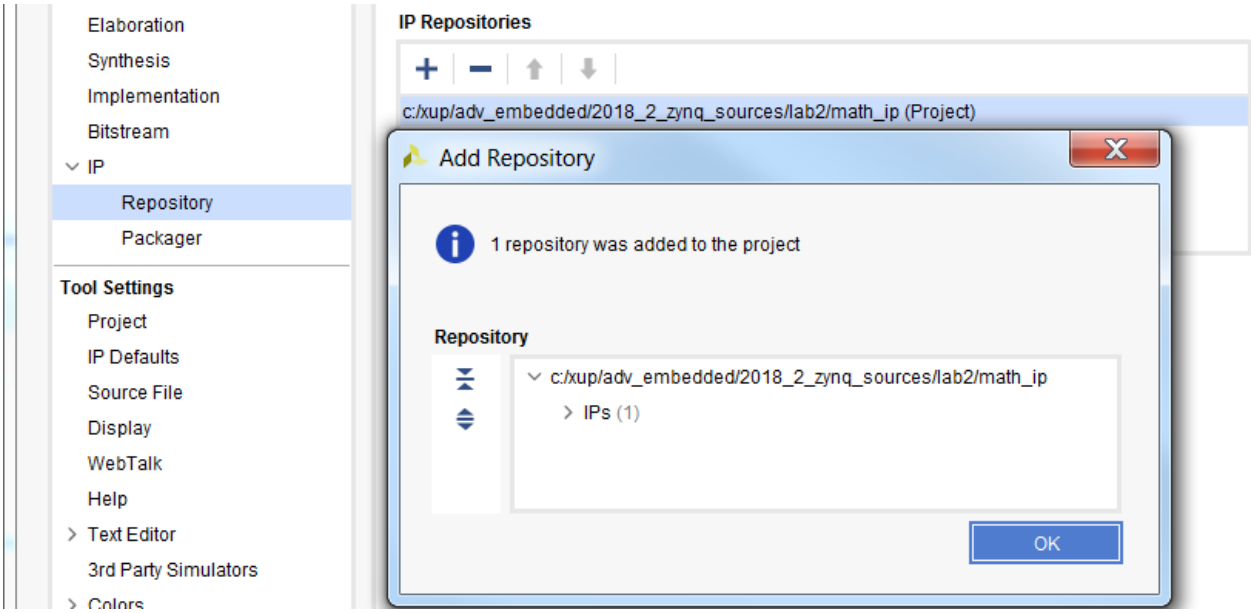


Figure 2. Specify IP Repository

1-1-6. Click **OK** twice to close the window.

Add the Custom IP — Step 2

2-1. Open the Block Design and add the custom IP to the system.

2-1-1. Click **Open Block Design** in the *Flow Navigator* pane to open the block diagram.

2-1-2. Click the + button and search for **math** in the catalog.

2-1-3. Double-click the **math_ip_v1_0** to add an instance of the core to the design.

2-1-4. Click on **Run Connection Automation**, (ensure math_ip_0 and S_AXI are selected) and click **OK.**

The *Math IP* consists of a hierarchical design with the lower-level module performing the addition. The higher-level module includes the two slave registers.

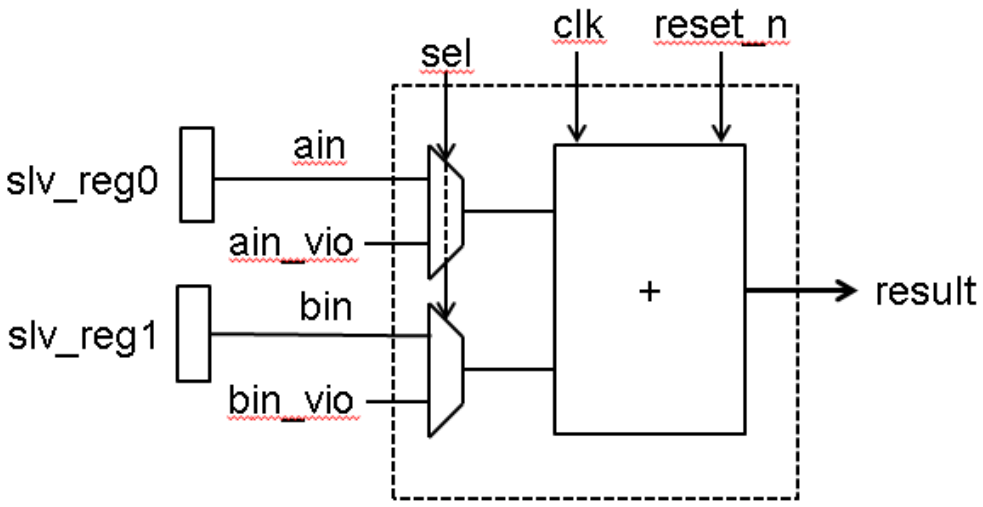


Figure 3. Custom Core's Main Functional Block

Add the ILA and VIO Cores Step 3

We want to connect the ILA core to the LED interface. Vivado prohibits connecting ILA cores to interfaces. In order to monitor the LED output signals, we need to convert the LED interface to simple output port.

3-1. Disable LEDs interface.

3-1-1. Double-click the *leds* instance to open its configuration form.

3-1-2. Click **Clear Board Parameters** and click **OK** to close the configuration form.

3-1-3. Select *leds_4bit* port and delete it.

3-1-4. Expand the *gpio* interface of the **leds** instance to see the associate ports.

3-2. Make the gpio_io_o port of the leds instance external and rename it as *leds*.

3-2-1. Move the mouse close to the end of the *gpio_io_o* port, left-click to select (do not select the main GPIO port), and then right click and select **Make External**.

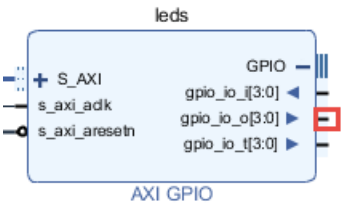


Figure 4. Select the gpio_io_o port

The port connector named gpio_io_o will be created and connected to the port.

3-2-2. Select the port *gpio_io_o* and change its name to **leds** by typing it in the properties form.

3-3. Enable cross triggering between the PL and PS

3-3-1. Double click on the *Zynq* block to open the configuration properties.

3-3-2. Click on PS-PL Configuration, and enable the *PS-PL Cross Trigger interface*.

3-3-3. Expand *PS-PL Cross Trigger interface* > *Input Cross Trigger*, and select **CPU0 DBG REQ** for *Cross Trigger Input 0.*

3-3-4. Similarly, expand Output Cross Trigger, and select **CPU0 DBG ACK** for *Cross Trigger Output 0* and click **OK.**

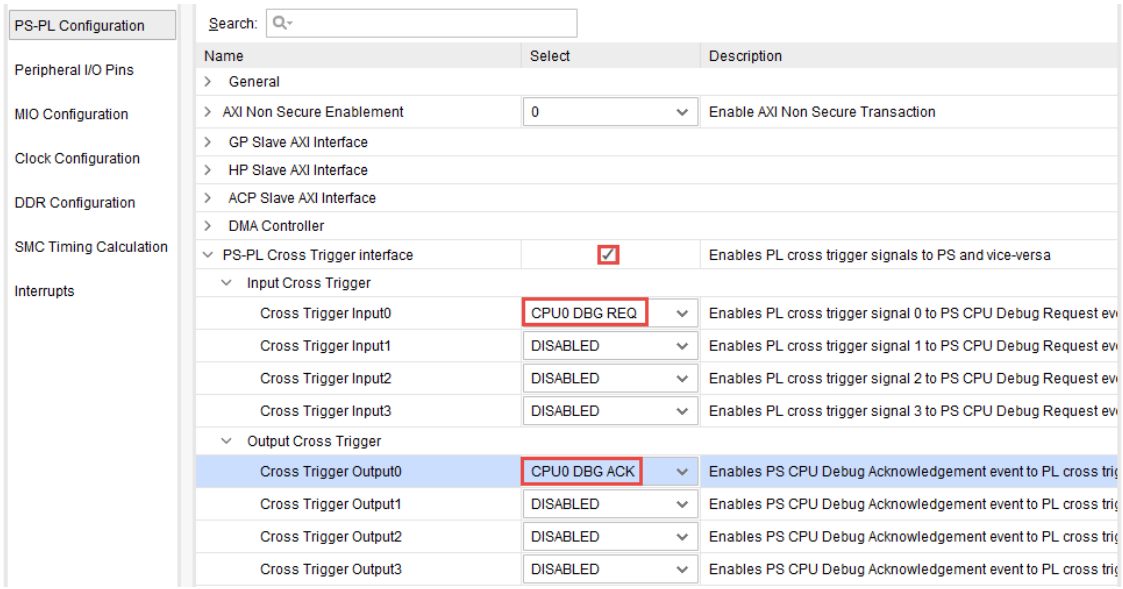


Figure 5. Enabling cross triggering in the Zynq processing system

3-4. Add the ILA core and connect it to the LED output port.

3-4-1. Click the + button and search for **ila** in the catalog.

3-4-2. Double-click on the **ILA (Integrated Logic Analyzer)** to add an instance of it. The *ila_0* instance will be added.

3-4-3. Double-click on the *ila_0* instance.

3-4-4. Select **Native** as the *Monitor type*.

3-4-5. Enable *Trigger Out Port*, and *Trigger In port.*

3-4-6. Select the **Probe Ports** tab, and set the **Probe Width** of *PROBE0* to **4** and click **OK**.

3-4-7. Using the drawing tool, connect the **PROBE0** port of the *ila_0* instance to the **gpio_io_o** port of the *leds* instance.

3-4-8. Connect the **clk** port of the *ila_0* instance to the **FCLK_CLK0** port of the Zynq subsystem.

3-4-9. Connect **TRIGG_IN** of the ILA to **TRIGGER_OUT_0** of the Zynq processing system, and **TRIG_OUT** of the ILA to the **TRIGGER_IN_0**.

3-5. Add the VIO core and connect it to the math_ip ports.

3-5-1. Click the + button and search for **vio** in the catalog.

3-5-2. Double-click on the **VIO (Virtual Input/Output)** to add an instance of it.

3-5-3. Double-click on the *vio* instance to open the configuration form.

3-5-4. In the *General Options* tab, leave the *Input Probe Count* set to **1** and set the *Output Probe Count* to **3**

3-5-5. Select the *PROBE_IN Ports* tab and set the *PROBE_IN0* width to **9**.

3-5-6. Select the *PROBE_OUT Ports* tab and set *PROBE_OUT0* width to **1**, *PROBE_OUT1* width to **8**, and *PROBE_OUT2* width to **8**.

3-5-7. Click **OK**.

3-5-8. Connect the VIO ports to the math instance ports as follows:

PROBE_IN -> result
PROBE_OUT0 -> sel
PROBE_OUT1 -> ain_vio
PROBE_OUT2 -> bin_vio

3-5-9. Connect the **CLK** port of the *vio_0* to FCLK_CKL0 net.

3-5-10. The block diagram should look similar to shown below.

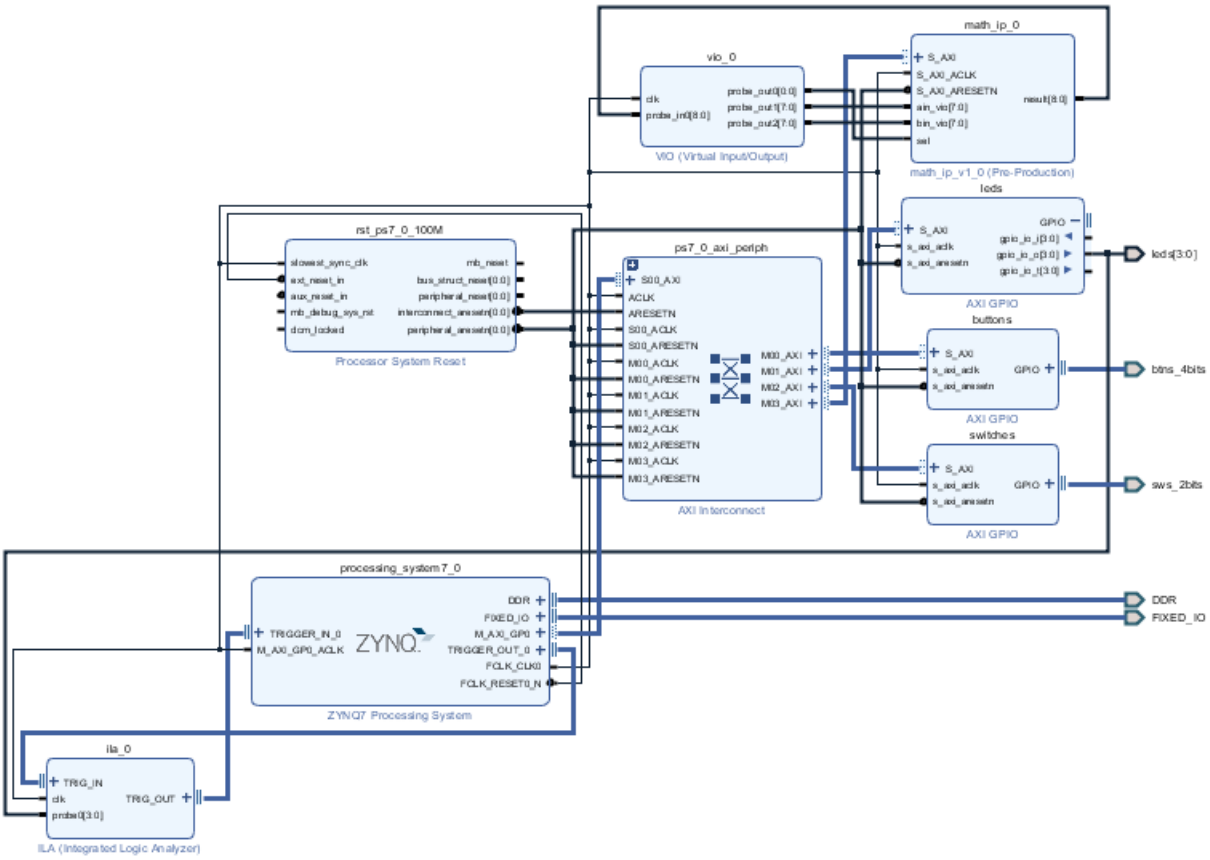


Figure 6. VIO added and connections made

3-6. Mark Debug the S_AXI connection between the AXI Interconnect and math_0 instance. Validate the design.

3-6-1. Select the **S_AXI** connection between the AXI Interconnect and the *math_ip_0* instance**.**

3-6-2. Right-click and select **Debug** to monitor the AXI4Lite transactions.

Notice that a system_ila IP instance got added and the M03_AXI <-> S_AXI connection is connected to its SLOT_0_AXI interface.

The block diagram should look as shown below.

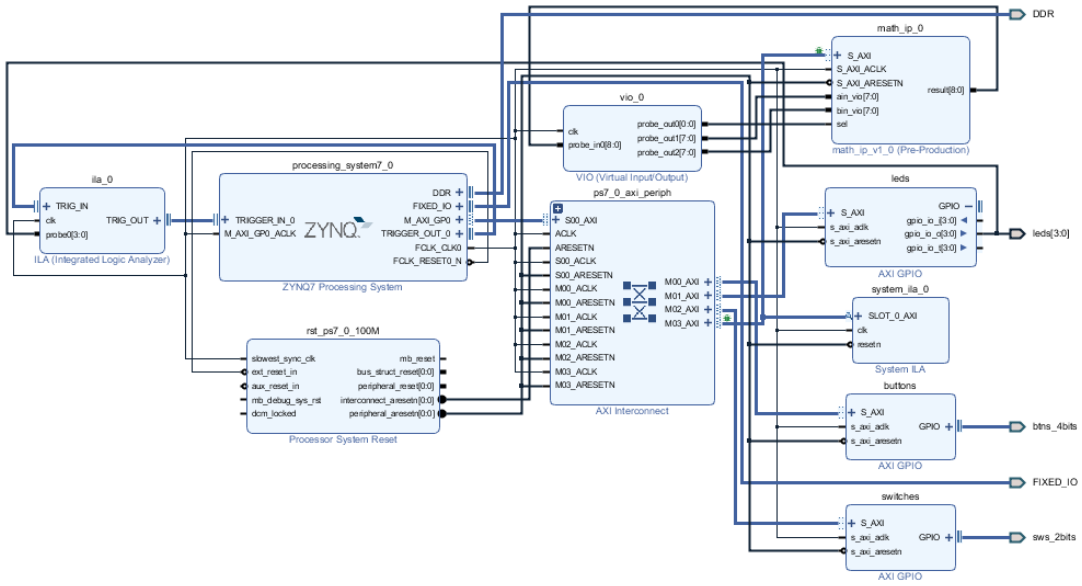


Figure 7. Block diagram of the design after marking AXI connection to the math_ip_0 instance for debugging

3-6-3. Click the **Run Connection Automation** link to see the form where you can select the desired channels to monitor.

3-6-4. Change *AXI Read Address* and *AXI Read Data* channels to **Data** since we will not trigger any signals of those channels.

This saves resources being used by the design.

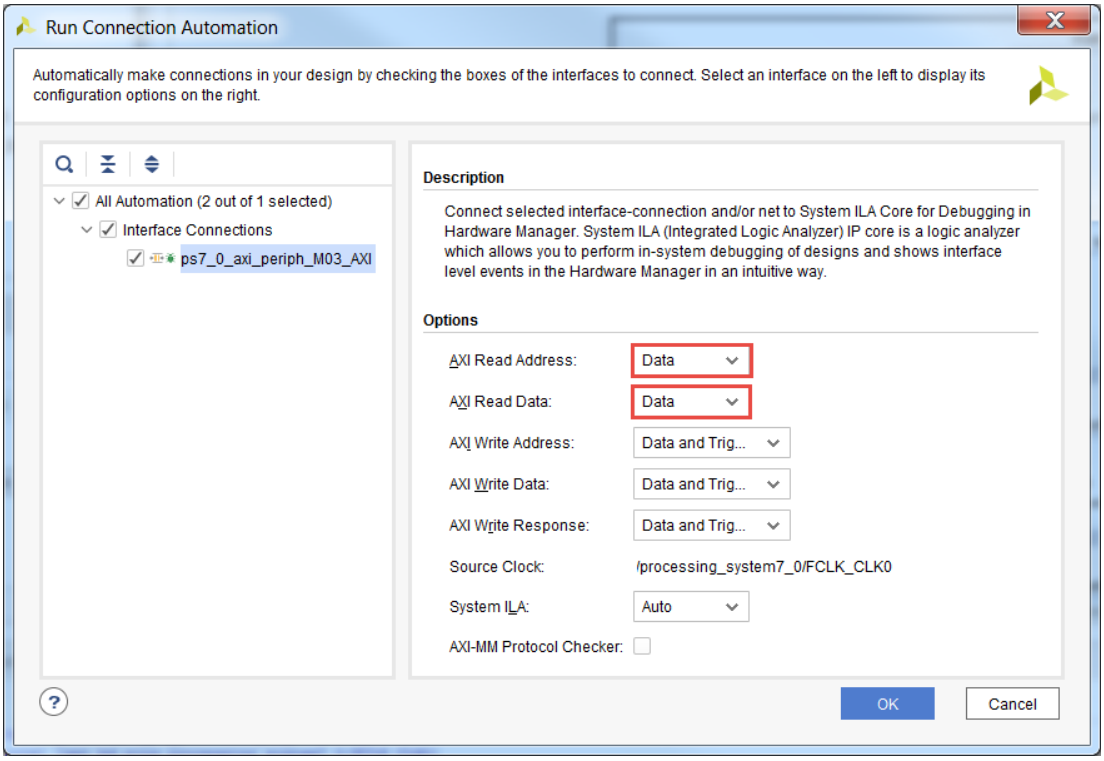


Figure 8. Selecting channels for debugging

3-6-5. Select the *Diagram* tab, and click on the (Validate Design) button to make sure that there are no errors.

3-6-6. Verify that there are no unmapped addresses shown in the *Address Editor* tab.

Add Design Constraints and Generate Bitstream — Step 4

4-1. Add the provided lab2_<board>.xdc from the sources\lab2 directory. Generate bitstream.

4-1-1. Right click in the *Sources* panel, and select **Add Sources.**

4-1-2. Select *Add or Create Constraints* and click **Next**.

4-1-3. Click the *Plus* button then **Add Files**, and browse to **{sources}\lab2** and select **lab2_pynz1.xdc,** or **lab2_pynqz2.xdc** depending on the board you are using**.**

4-1-4. Click **OK** and then click **Finish.**

4-1-5. Click on the **Generate Bitstream** to run the implementation and bit generation processes.

4-1-6. Click **Save** to save the project (if prompted), **OK** to ignore the warning (if prompted), and **Yes** to launch Implementation (if prompted). Click **OK** to launch the runs.

4-1-7. When the bitstream generation process has completed successfully, click **Cancel**.

Generate an Application in SDK — Step 5

5-1. Export the implemented design and launch SDK.

5-1-1. Export the hardware configuration by clicking **File > Export > Export Hardware…**, click the box to *Include Bitstream*

5-1-2. Click **OK** to export and **Yes** to overwrite the previous project created by lab1.

5-1-3. Launch SDK by clicking **File > Launch SDK** and click **OK.**

5-1-4. Right-click on the **lab1** and **standalone_bsp_0** and **system_wrapper_hw_platfrom_0** projects in the Project Explorer view and select **close project**.

5-2. Create an empty application project named lab2, and import the provided lab2.c file.

5-2-1. Select **File > New** > **Application Project.**

5-2-2. In the *Project Name* field, enter **lab2** as the project name, leave all other settings to their default's and click **Next** (a new BSP will be created)**.**

5-2-3. Select the **Empty Application** template and click **Finish.**

The lab2 project will be created in the Project Explorer window of the SDK.

5-2-4. Select **lab2 > src** in the project view, right-click, and select **Import.**

5-2-5. Expand the **General** category and double-click on **File System.**

5-2-6. Browse to the **{sources}\lab2** folder.

5-2-7. Select **lab2.c** and click **Finish.**

A snippet of the part of the source code is shown in the following figure. It shows that two operands are written to the custom core, the result is read, and printed out. The write transaction will be used as a trigger condition in the Vivado Logic Analyzer.

```
    xil_printf("-- Press any of BTN0-BTN3 to see corresponding output on LEDs --\r\n");
    xil_printf("-- Set slide switches to 0x03 to exit the program --\r\n");

    Xil_Out32(XPAR_MATH_IP_0_BASEADDR, 0x12);
    Xil_Out32(XPAR_MATH_IP_0_BASEADDR+4, 0x34);
    i=Xil_In32(XPAR_MATH_IP_0_BASEADDR);
    xil_printf("result=%x\r\n",i);

    while (1)
    {
        btns_check = XGpio_DiscreteRead(&btns, 1);
        XGpio_DiscreteWrite(&leds, 1, btns_check);
        sws_check = XGpio_DiscreteRead(&sws,1);
        if((sws_check & 0x03)==0x03)
            break;
        for (i=0; i<9999999; i++); // delay loop
     }
    xil_printf("-- End of Program --\r\n");

    return 0;
}
```

Figure 9. Source Code snippet

5-2-8. Right click on *lab2*, and select **Debug As > Debug Configurations**

5-2-9. Double click on Xilinx C/C++ application (System Debugger) to create a new configuration (*lab2 Debug will be created*), and in the *Target Setup* tab, check the **Enable Cross-Triggering** option**,** and click the Browse button.

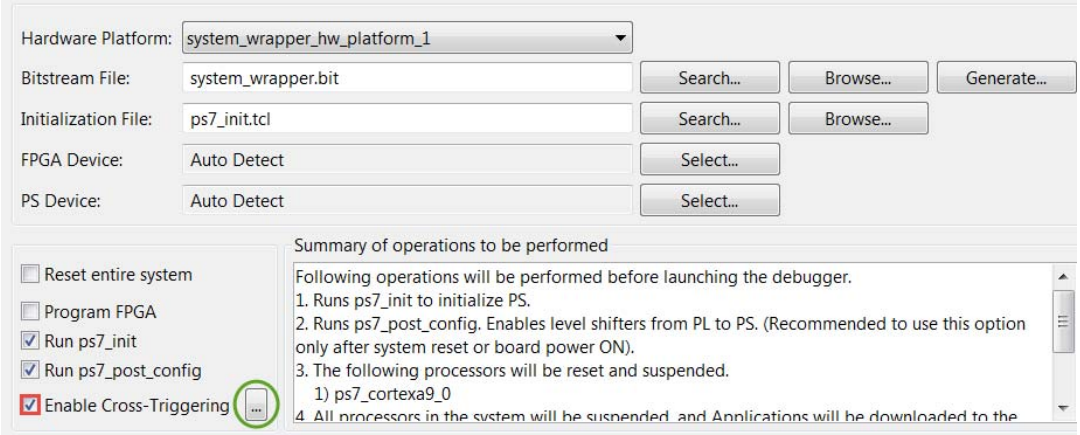


Figure 10. Enable cross triggering in the software environment

5-2-10. When the *Cross Trigger Breakpoints* dialog box opens, click **Create**

5-2-11. Select the options as shown in *Figure 11* and click **OK** to set up the cross-trigger condition for Processor to Fabric.

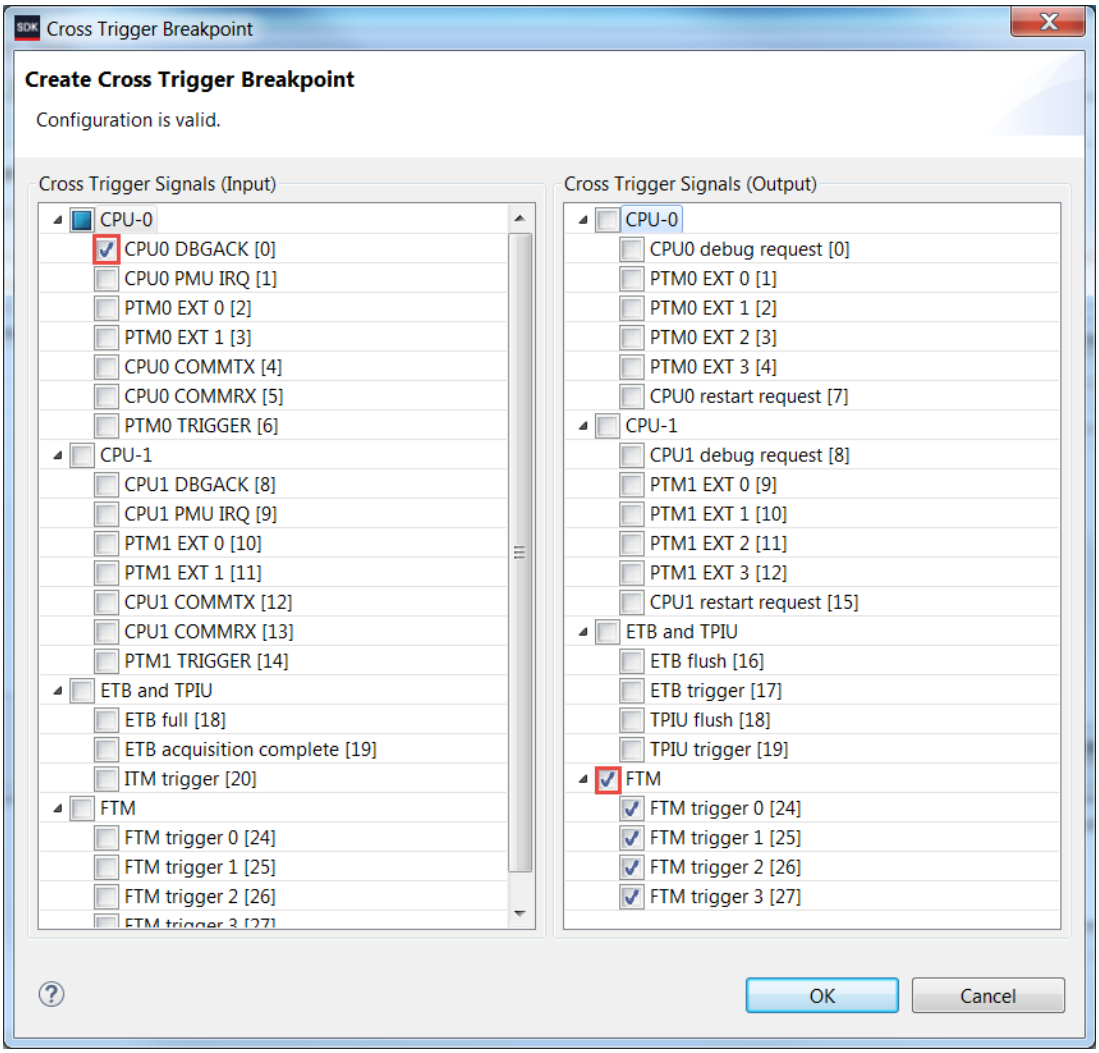


Figure 11. Enabling CPU0 for request from PL

5-2-12. In the *Cross Trigger Breakpoints* dialog box click **Create** again.

5-2-13. Select the options as shown in *Figure 12* and click **OK** to set up the cross trigger condition for Fabric to Processor.

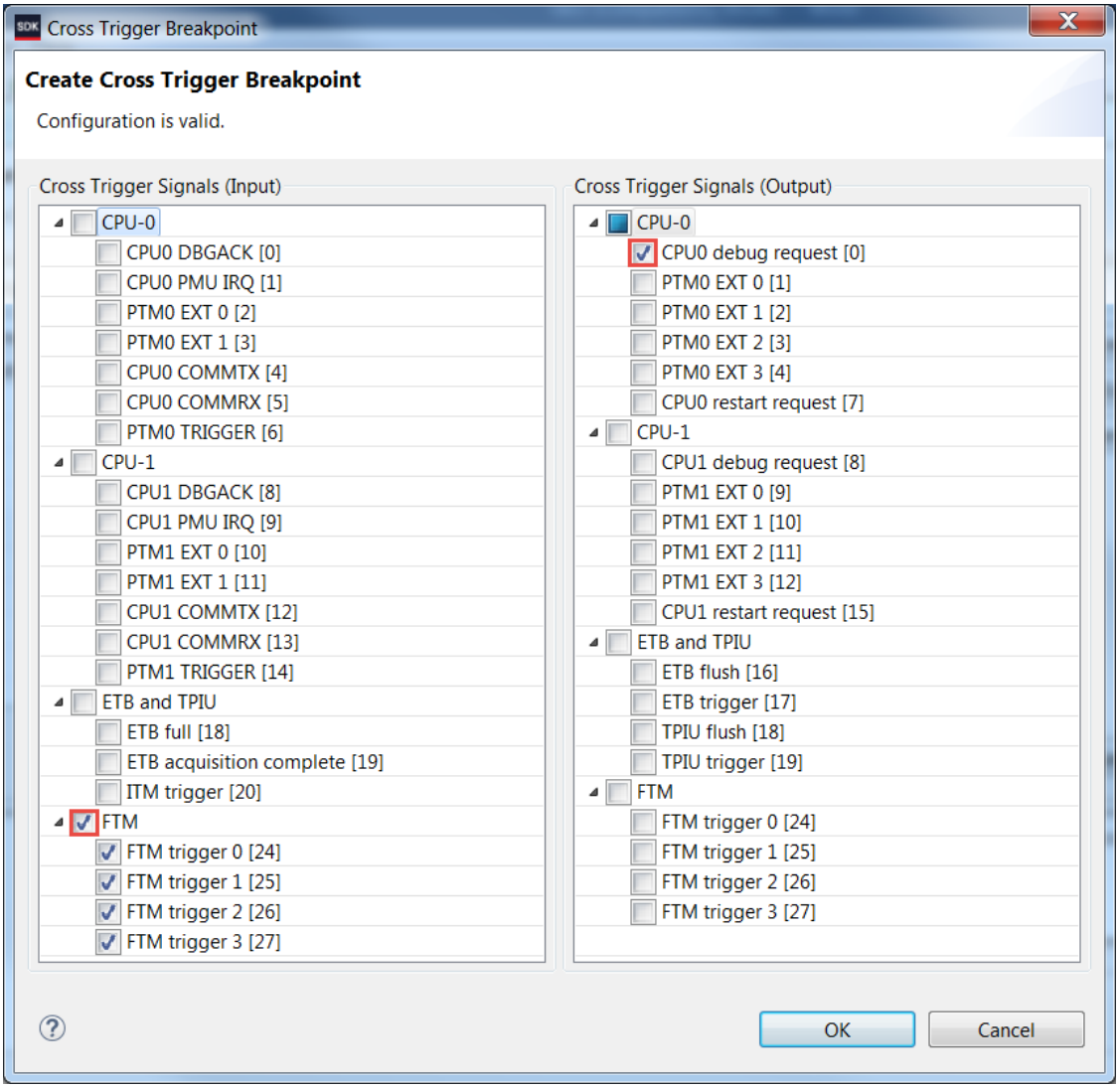


Figure 12. Enabling CPU0 for request to PL

5-2-14. Click **OK**, then click **Apply,** then **Close**

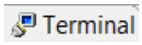
Test in Hardware Step 6

6-1. Connect and power up the board. Download the bitstream into the target device. Start the debug session on lab2 project. Switch to the Debug perspective and establish serial communication.

6-1-1. Connect and power up the board.

6-1-2. Select **Xilinx** > **Program FPGA** and click **Program**

6-1-3. Select the **lab2** project in *Project Explorer*, right-click and select **Debug As > Launch on Hardware** (System Debugger) to download the application, execute ps7_init. (If prompted, click **Yes** to switch to the Debug perspective.) The program execution starts and suspends at the entry point.

6-1-4. Select the 

tab. If it is not visible then select **Window > Show view > Terminal**.

6-1-5. Click on  and select the appropriate COM port (depending on your computer), and configure it as you did it in Lab 1.

6-2. Start the hardware session from Vivado.

6-2-1. Switch to Vivado.

6-2-2. Click on **Open Hardware Manager** from the *Program and Debug* group of the *Flow Navigator* pane to invoke the analyzer.

6-2-3. Click on the **Open Target > Auto connect** to establish the connection with the board.

6-2-4. Select **Window > Debug Probes**

The hardware session will open showing the **Debug Probes** tab in the **Console** view.

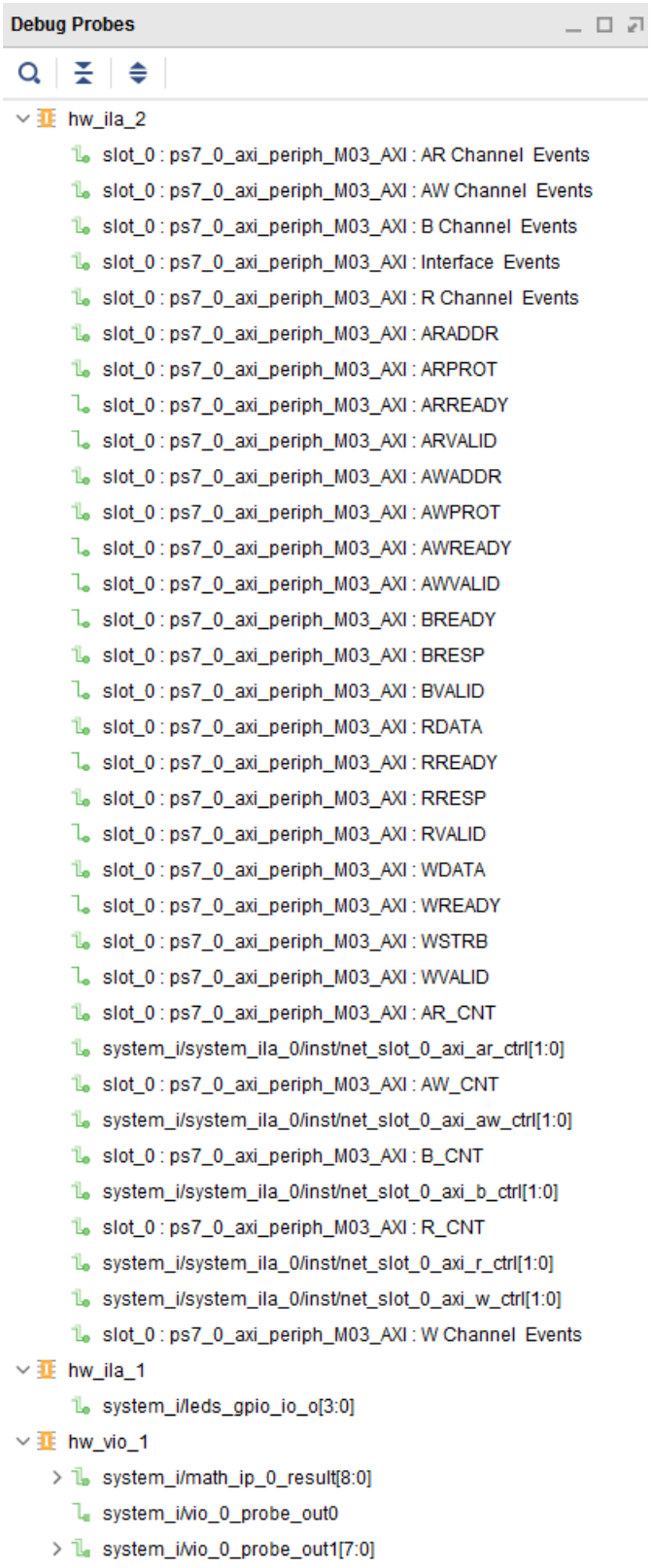


Figure 13. Debug probes

The hardware session status window also opens showing that the FPGA is programmed (we did it in SDK), there are three cores out of which the two ila cores are in the idle state.

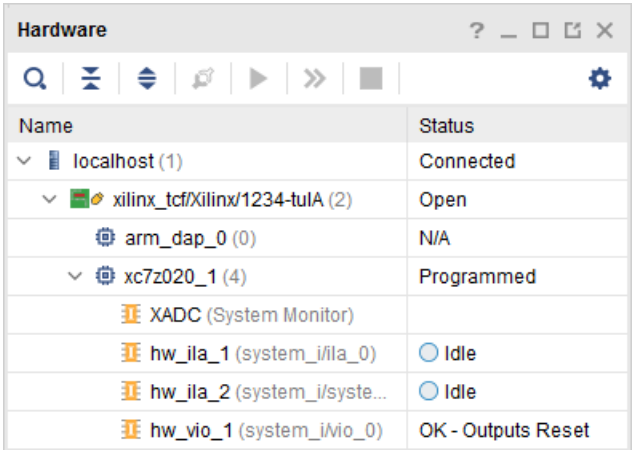


Figure 14. Hardware session status

6-2-5. Select the XC7Z020, and click on the **Run Trigger Immediate** button to see the signals in the waveform window.

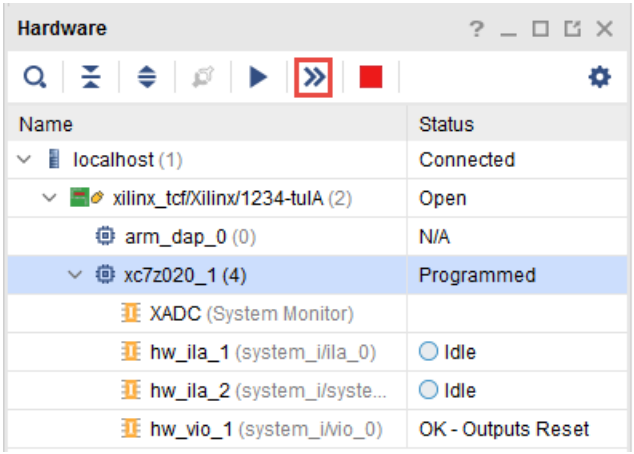


Figure 15. Opening the waveform window

6-3. Setup trigger conditions to trigger on a write transaction (WSTRB) when the desired data (WDATA) of XXXX_XX12 is written. The transaction takes place when WVALID and WREADY are equal to 1.

6-3-1. Click on the *hw_ila_2* tab to select it. In the **Debug Probes** window, under *hw_ila_2*, drag and drop the **WDATA** signal to the *ILA Basic Trigger setup* window**.**

6-3-2. Set the value to **XXXX_XX12** (HEX) (the value written to the math_0 instance at line 24 of the program).

6-3-3. Similarly, add **WREADY, WSTRB,** and **WVALID** signals to the *ILA Basic Trigger setup* window.

6-3-4. Change the radix to binary for *WSTRB*, and change the value from **xxxx** to **xxx1**

6-3-5. Change the value of **WVALID** and **WREADY** to 1.

6-3-6. Set the trigger position of the *hw_ila_2* to **512** in the *Settings – hw_ila_2* tab.

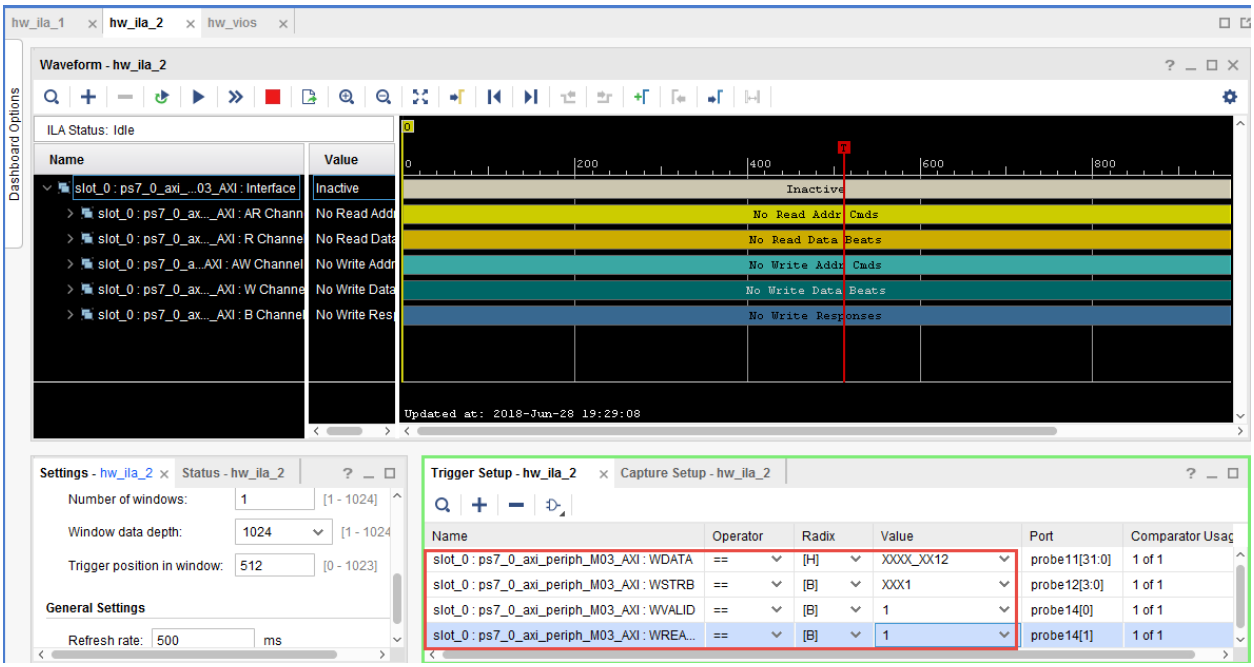


Figure 16. Setting up the ILA

6-3-7. Similarly, set the trigger position in the *Settings – hw_ila_1* tab to **512**.

6-3-8. Select **hw_ila_2** in the *Hardware* window and click on the **Run Trigger** button and observe that the *hw_ila_2* core is armed and showing the status as **Waiting For Trigger**.

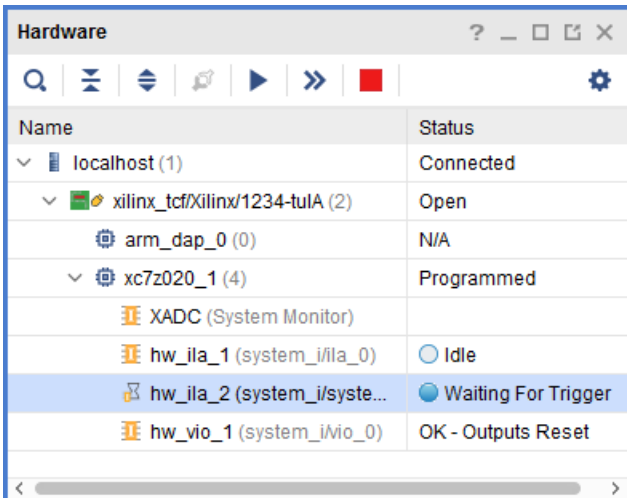


Figure 17. Hardware analyzer running and in capture mode

6-3-9. Switch to SDK.

6-3-10. Near line 27 (right click in the margin and select *Show Line Numbers* if necessary*)*, double click on the left border on the line where xil_printf statement is (before the while (1) statement) is defined in the lab2.c window to set a breakpoint.

```
24      Xil_Out32(XPAR_MATH_IP_0_BASEADDR, 0x12);
25      Xil_Out32(XPAR_MATH_IP_0_BASEADDR+4, 0x34);
26      i=Xil_In32(XPAR_MATH_IP_0_BASEADDR);
27      xil_printf("result=%x\r\n",i);
```

Figure 18. Setting a breakpoint

6-3-11. Click on the **Resume** ( ) button to execute the program and stop at the breakpoint.

6-3-12. In the Vivado program, notice that the **hw_ila_2** status changed from *Waiting for Trigger* to *Idle*, and the waveform window shows the triggered output (select the *hw_ila_data_2.wcfg* tab if necessary).

6-3-13. Move the cursor to closer to the trigger point and then click on the button to zoom at the cursor. Click on the **Zoom In** button couple of times to see the activity near the trigger point. Similarly, you can see other activities by scrolling to right as needed.

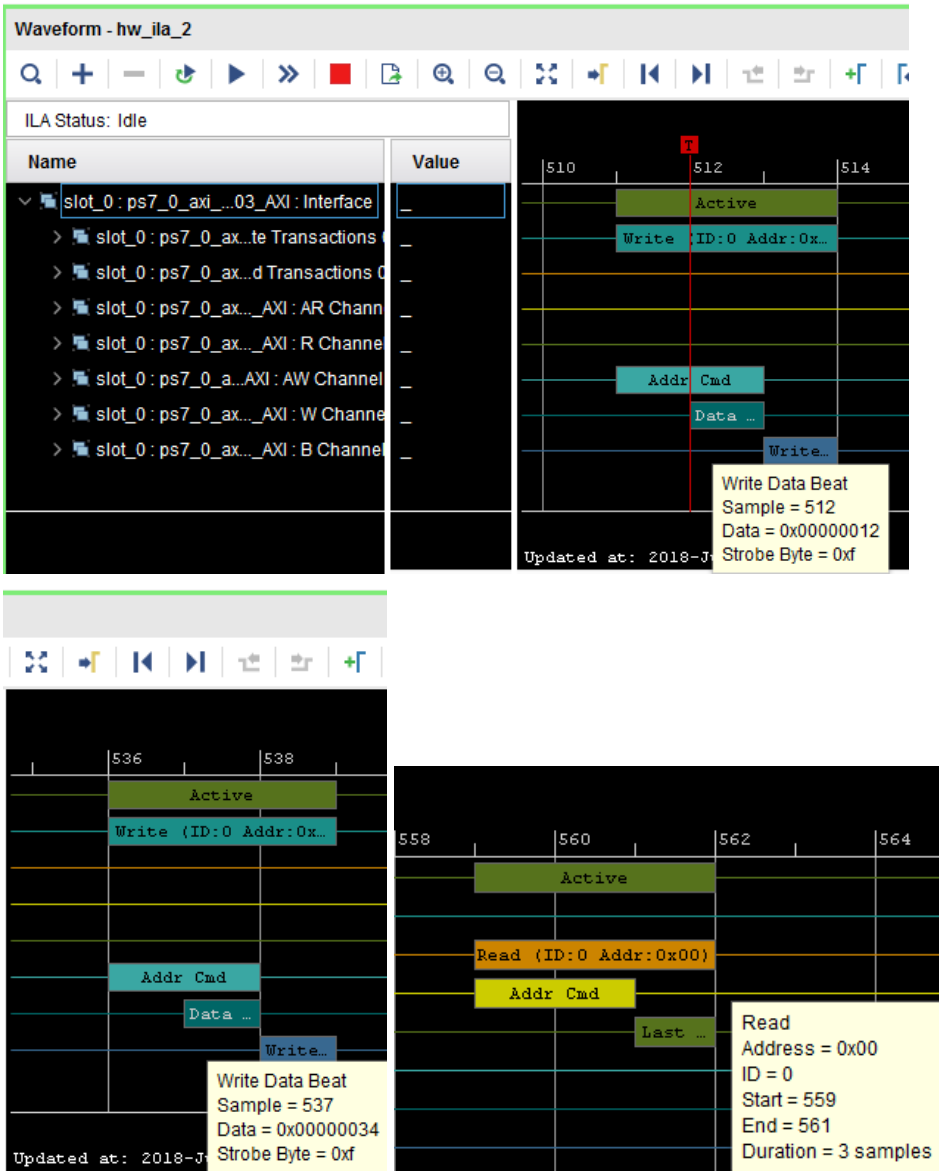


Figure 19. Zoomed waveform view of the three AXI transactions

Observe the following:

Around the 512th sample WDATA being written is 0x012 at offset 0 (AWADDR=0x0).

At the 536th sample, offset is 0x4 (AWADDR), and the data being written is 0x034.

At the 559th sample, data is being read from the IP at the offset 0x0 (ARADDR), and at 561st mark the result (0x46) is on the RDATA bus.

6-3-14. You also should see the following output in the SDK Terminal console.

```
-- Start of the Program --
-- Press any of BTN0-BTN3 to see corresponding output on LEDs --
-- Set slide switches to 0x03 to exit the program --
```

Figure 20. Terminal Output

6-4. In Vivado, select the VIO Cores related from the Dashboard Options windows, set the vio_1_probe_out0 so math_ip's input can be controlled manually through the VIO core. Try entering various values for the two operands and observe the output on the math_ip_1_result port in the Console pane.

6-4-1. Select the **hw_vio_1** core in the *Dashboard Options* panel.

6-4-2. Click on the + button and select all signals to stimulate and monitoring. Change the **vio_0_probe_out0** value to **1** so the math_ip core input can be controlled via the VIO core.

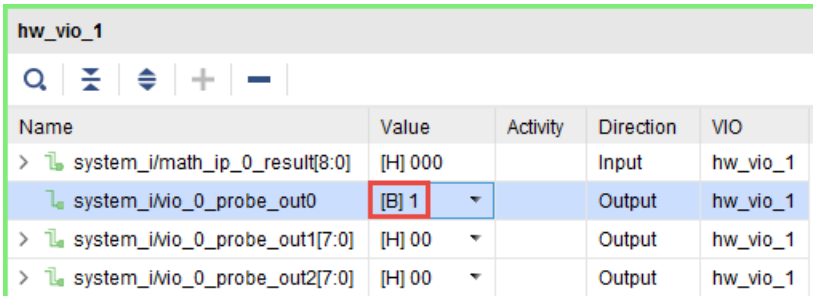


Figure 21. VIO probes

6-4-3. Change **vio_0_probe_out1** value to **55** (in Hex), and similarly, **vio_0_probe_out2** value to **44** (in Hex). Notice that for a brief moment a blue-colored up-arrow will appear in the Activity column and the result value changes to **099** (in Hex).

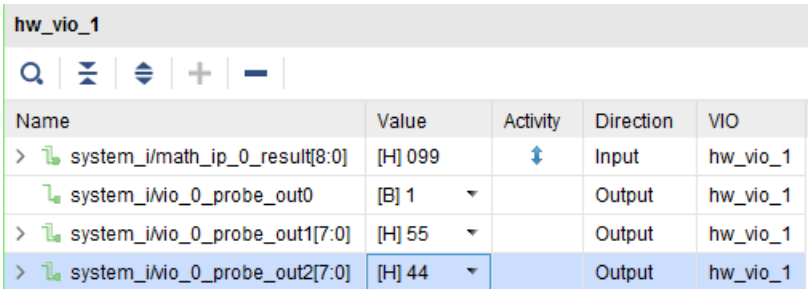


Figure 22. Input stimuli through the VIO core's probes

www.xilinx.com/university
xup@xilinx.com

6-4-4. Try a few other inputs and observe the outputs.

6-4-5. Once done, set the *vio_0_probe_out0* to **0** to isolate the vio interactions with the math_ip core.

6-5. Setup the ILA core (hw_ila_1) trigger condition to 0x2 for the PYNQ-Z1/PYNQ-Z2. Make sure that the switches on the board are not set at x3 (for PYNQ-Z1/PYNQ-Z2). Set the trigger equation to be ==, and arm the trigger. Click on the Resume button in the SDK to continue executing the program. Change the switches and observe that the hardware core triggers when the preset condition is met.

6-5-1. Select the **hw_ila_1** in the *Dashboard Options* panel.

6-5-2. Add the LEDs to the *Basic Trigger Setup*, and set the trigger condition of the *hw_ila_1* to trigger at LED output value equal to **0x5** for the PYNQ-Z1/PYNQ-Z2.

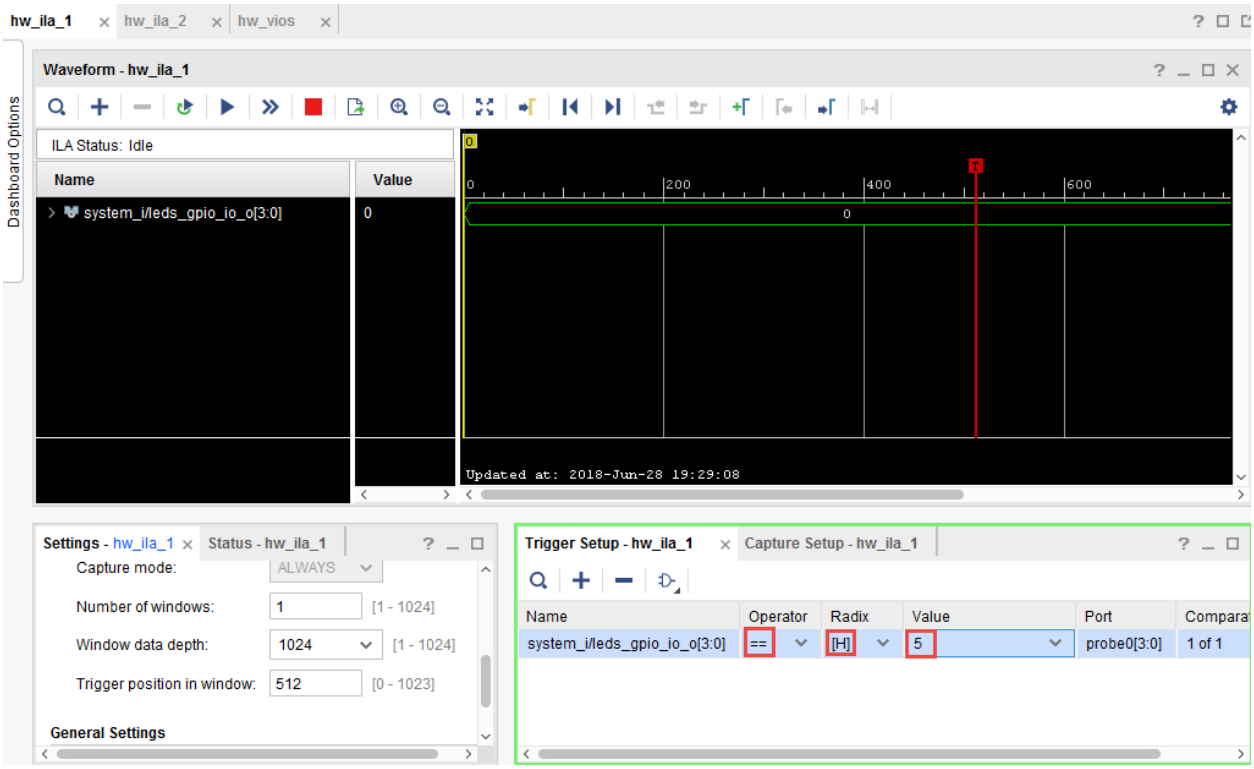


Figure 23. Setting up Trigger for hw_ila_1

6-5-3. Ensure that the trigger position for the *hw_ila_1* is set to **512**.

Make sure that the switches are not set to 11 (PYNQ-Z1/PYNQ-Z2) as this is the exit pattern.

6-5-4. Right-click on the *hw_ila_1* in the *hardware* window, and arm the trigger by selecting **Run Trigger.**

The hardware analyzer should be waiting for the trigger condition to occur.

6-5-5. In the SDK window, click on the *Resume* button.

6-5-6. Press the push-buttons and see the corresponding LED turning ON and OFF.

6-5-7. When the condition is met, the waveform will be displayed.

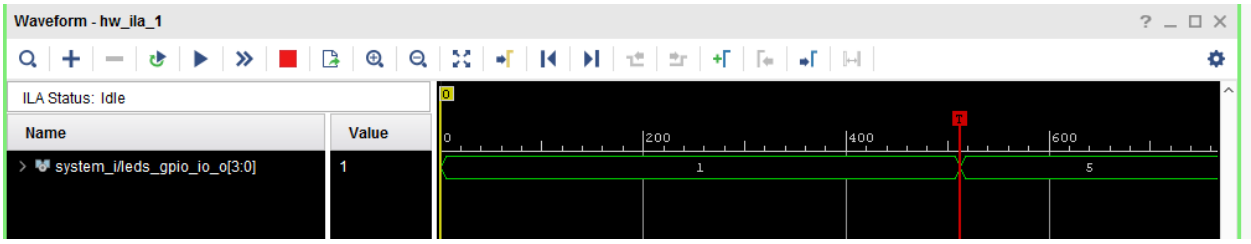


Figure 24. ILA waveform window after Trigger

6-6. Cross trigger a debug session between the hardware and software

6-6-1. In Vivado, select *hw_ila_1*

6-6-2. In the ILA properties, set the *Trigger mode* to **BASIC_OR_TRIGG_IN**, and the *TRIG_OUT* mode to **TRIGGER_OR_TRIG_IN**

6-6-3. In SDK, in the C/C++ view, relaunch the software by right clicking on the lab2 project, and selecting *Debug As > Launch on Hardware (System Debugger)* (Click **OK** if prompted to reset the processor)

The program will be loaded and the excution will suspend at the entry point

6-6-4. Arm the *hw_ila_1* trigger

6-6-5. In SDK continue execution of the software to the next breakpoint (line 27)

When the next breakpoint in SDK is reached, return to Vivado and notice the ILA has triggered

6-7. Trigger the ILA and cause the software to halt

6-7-1. Click Step Over (F6) button twice to pass the current breakpoint

6-7-2. Arm the *hw_ila_1* trigger

6-7-3. Resume the software (F8) until it enters the while loop

6-7-4. Verify it is executing by toggling the dip switches

6-7-5. In Vivado, arm the *hw_ila_1* trigger

6-7-6. Press the push-buttons to 0x5, and notice that the application in SDK will break at some point (This point will be somewhere within the while loop)

6-7-7. Click on the **Resume** button

The program will continue execution. Flip switches until it is *0x03*.

6-7-8. Click the Disconnect button ( ) in the SDK to terminate the execution.

6-7-9. Close the SDK by selecting **File > Exit**.

6-7-10. Close the hardware session by selecting **File > Close Hardware Manager**. Click **OK**.

6-7-11. Close Vivado program by selecting **File > Exit**.

6-7-12. Turn OFF the power on the board.

Conclusion

In this lab, you added a custom core with extra ports so you can debug the design using the VIO core. You instantiated the ILA and the VIO cores into the design. You used Mark Debug feature of Vivado to debug the AXI transactions on the custom peripheral. You then opened the hardware session from Vivado, setup various cores, and verified the design and core functionality using SDK and the hardware analyzer.

.