Debugging Using Hardware Analyzer

Introduction

Software and hardware interact with each other in an embedded system. The Xilinx SDK includes both GNU and the Xilinx Microprocessor Debugger (XMD) as software debugging tools. 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 lab.

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**).

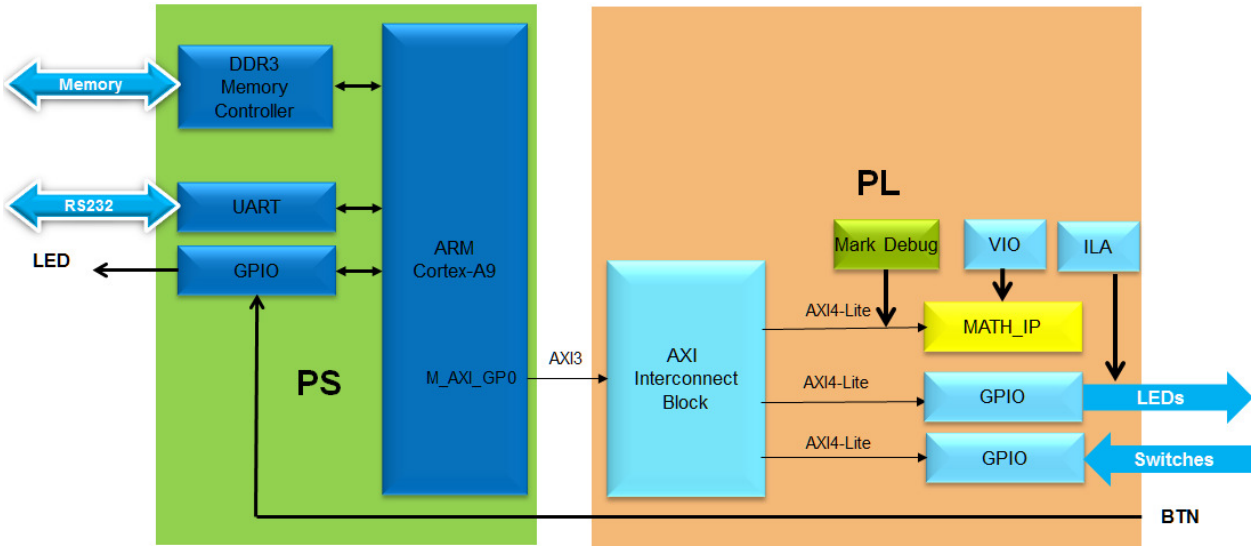
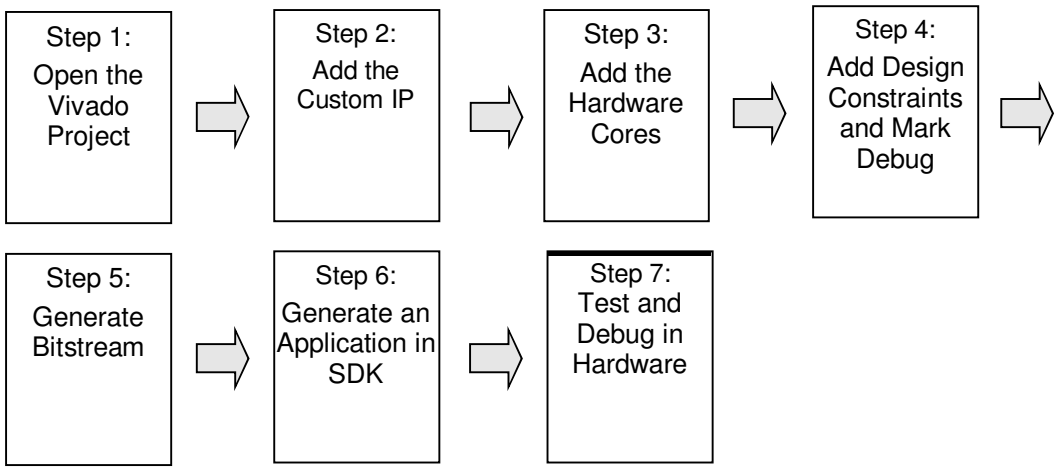


Figure 1. Completed Design

General Flow for this Lab



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 > Save Project 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 c:\xup\adv_embedded\labs\ and click **OK**.

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

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

1-1-4. Select **IP** in the left pane of the *Project Settings* form.

1-1-5. Click on the **Green Plus** button, browse to **c:\xup\adv_embedded\sources\lab2\math_ip** and click **Select**.

1-1-6. The math_ip_v1_0 IP will appear the **IP in the Selected Repository** window.

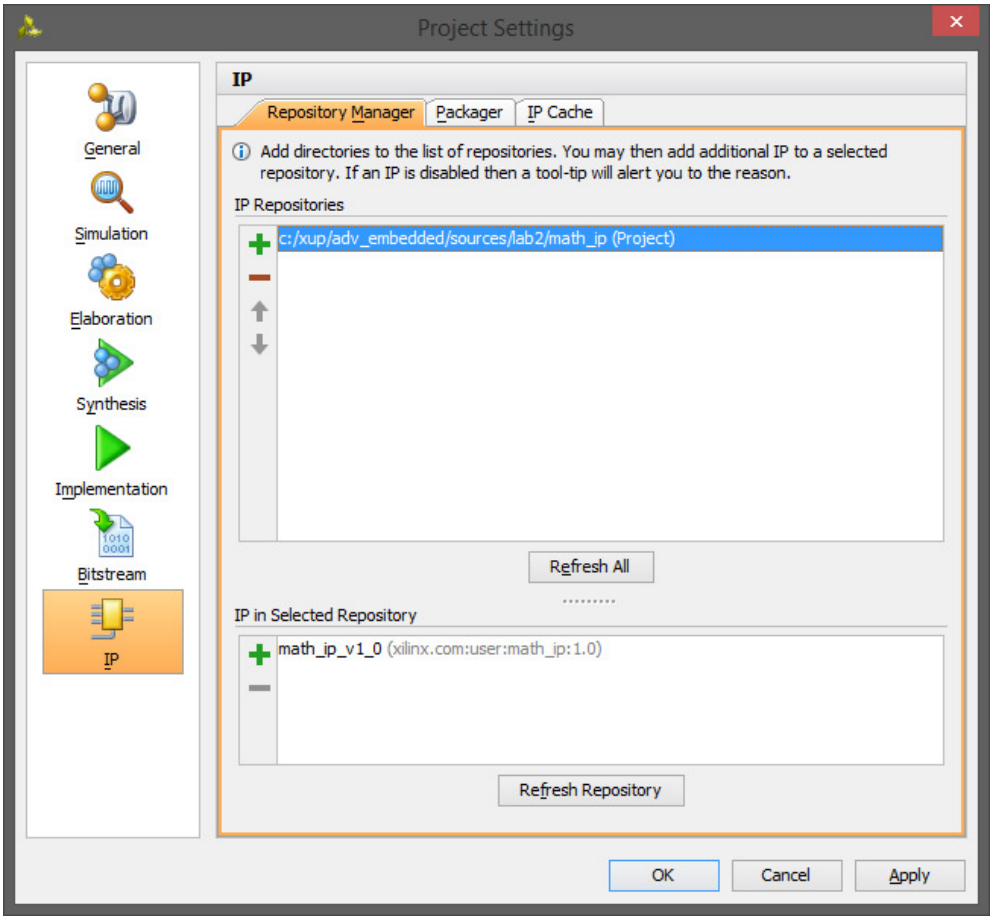


Figure 2. Specify IP Repository

1-1-7. Click **OK**.

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, and select **system.bd** to open the block diagram.

2-1-2. Click the Add IP icon 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.

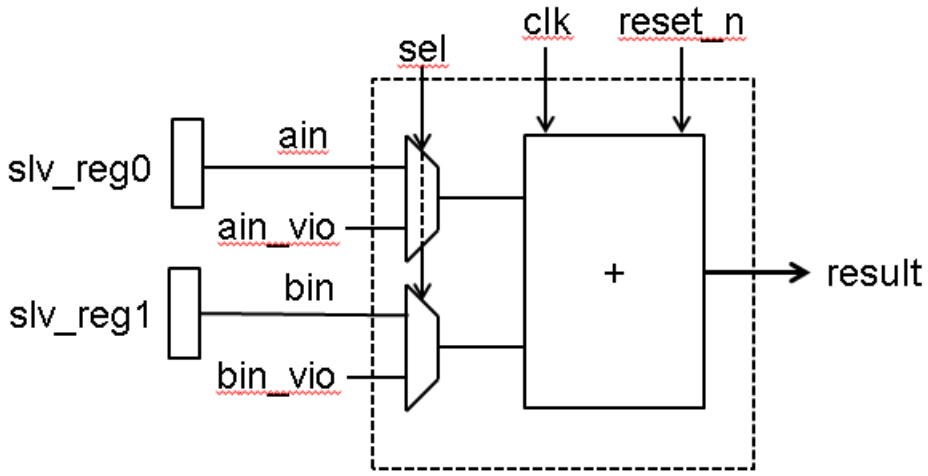


Figure 3. Custom Core's Main Functional Block

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. Expand the *gpio* interface of the **leds** instance to see the gpio_io_o port.

3-1-4. Delete the associated port, which will also disconnect the connection, by selecting it, right-clicking on it and selecting **Delete**.

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 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 clock **OK.**

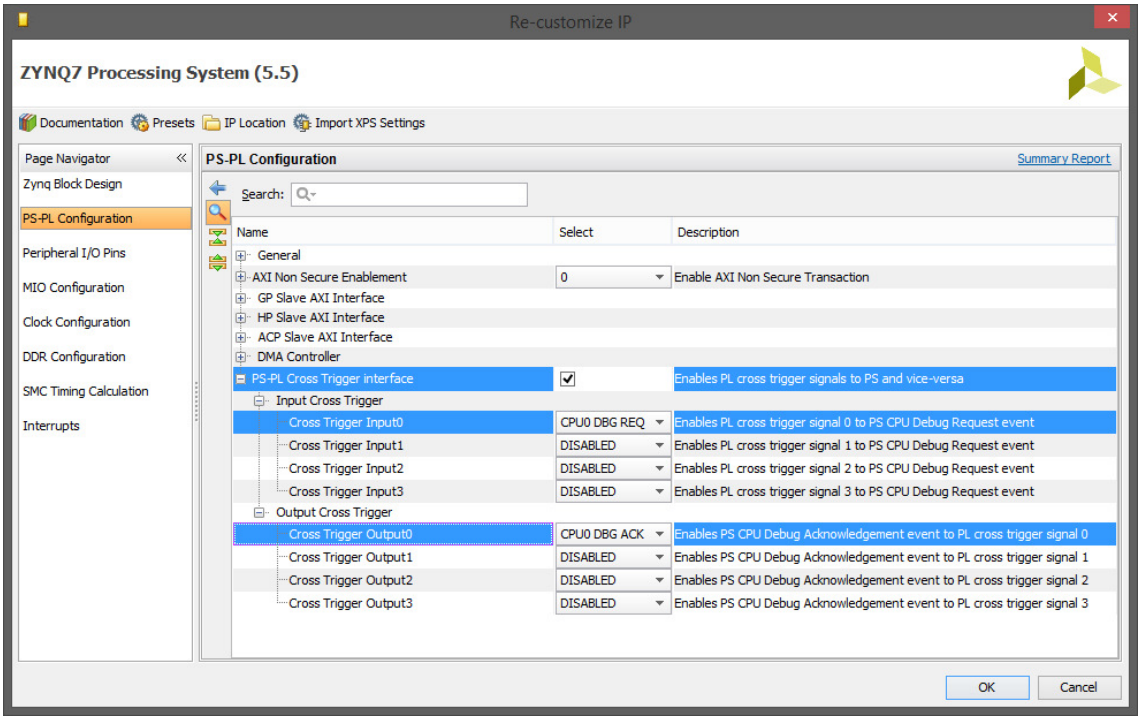


Figure 5. Enabling cross triggering in the Zynq processing system

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

3-4-1. Click the Add IP icon 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 **8** for the Zedboard and **4** for the Zybo 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 Add IP icon 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.

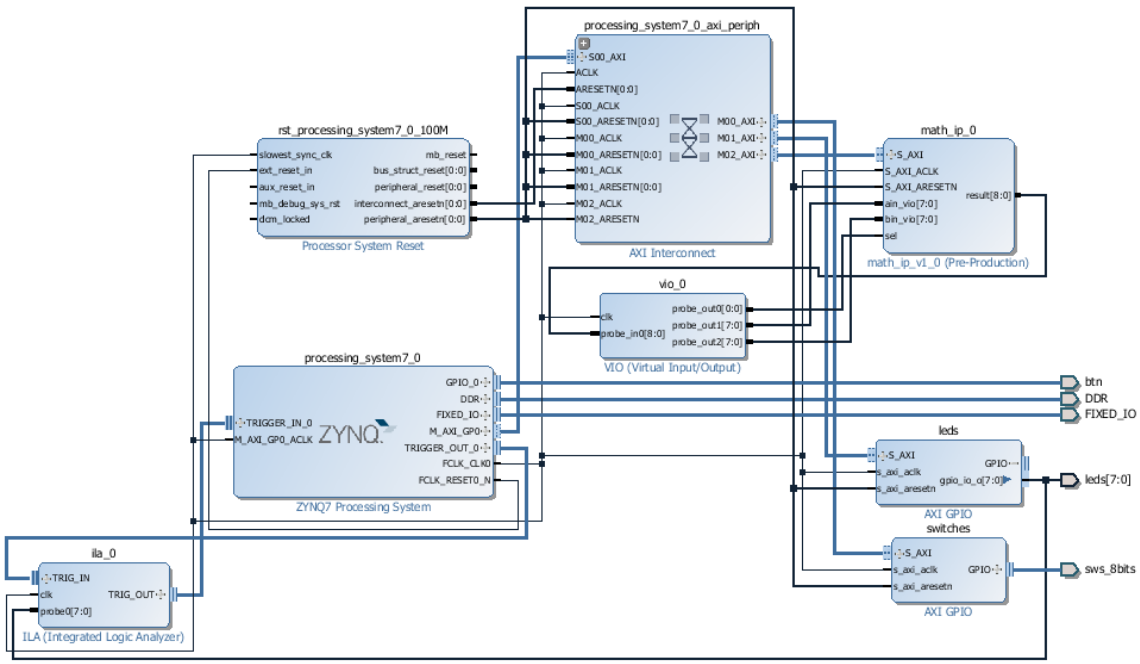


Figure 6. VIO added and connections made

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 **Mark Debug** to monitor the AXI4Lite transactions.

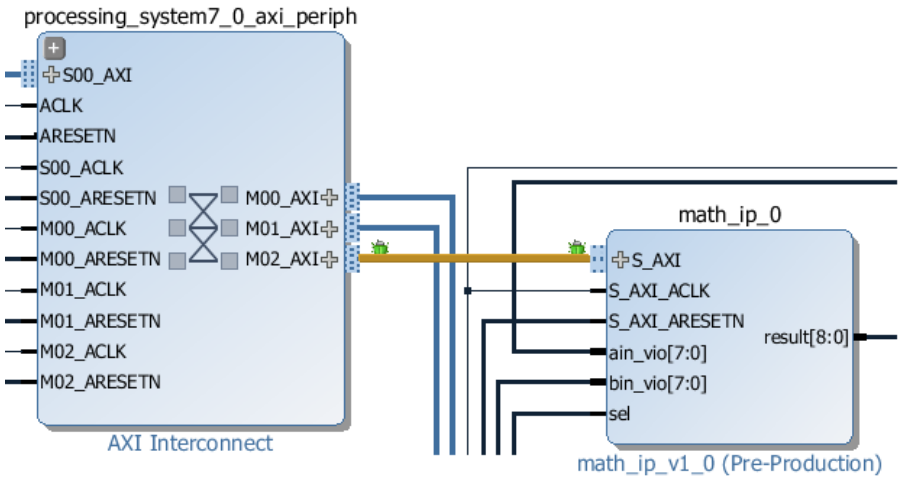


Figure 7. Mark Debug on S_AXI interface

3-6-3. Select **Tools > Validate Design** to run the design rules checker.

3-6-4. Verify that there are no unmapped addresses shown in the *Address Editor* tab.

Add Design Constraints and Assign Nets for Debugging Step 4

4-1. Add the provided .xdc from the sources\lab2 directory. Run the synthesis.

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 **Green Plus** then **Add Files**, and browse to **c:\xup\adv_embedded\sources\lab2** and select **lab2_zybo.xdc** or **lab2_zedboard.xdc** depending on the board you are using**.**

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

4-1-5. Click **Run Synthesis**, and click **OK**, and **Save** (if prompted) to save the design and run the synthesis process.

4-1-6. When the synthesis is completed, select the *Open Synthesized Design* option and click **OK**.

4-2. Assign nets for debugging.

4-2-1. The synthesized design will be opened in the Auxiliary pane and the **Debug** tab will be opened in the *Console* pane.

If the Debug tab is not open then select **Window > Debug**.

Notice that the nets which can be debugged are grouped into the signals connected to the *ila*, and *vio*, and *Unassigned nets*. The Unassigned nets are associated with the AXI interface that was marked for debug.

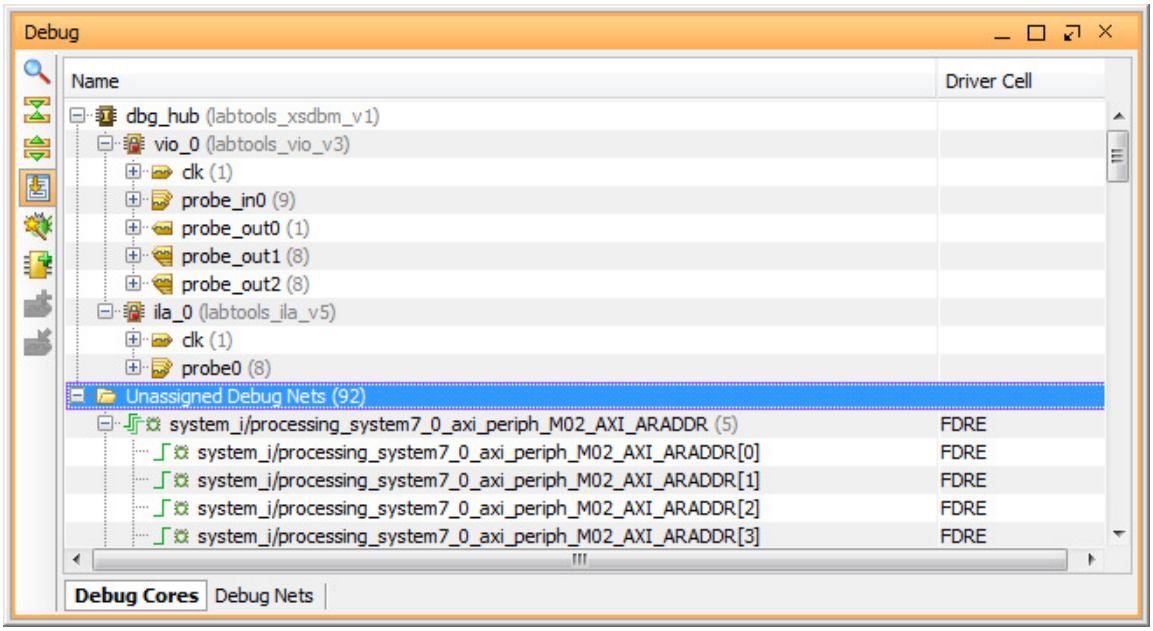


Figure 8. The Debug tab

4-2-2. Right-click on the *Unassigned Debug Nets* and select **Set up Debug…** and click **Next.**

The nets are listed. There are 92 nets to debug.

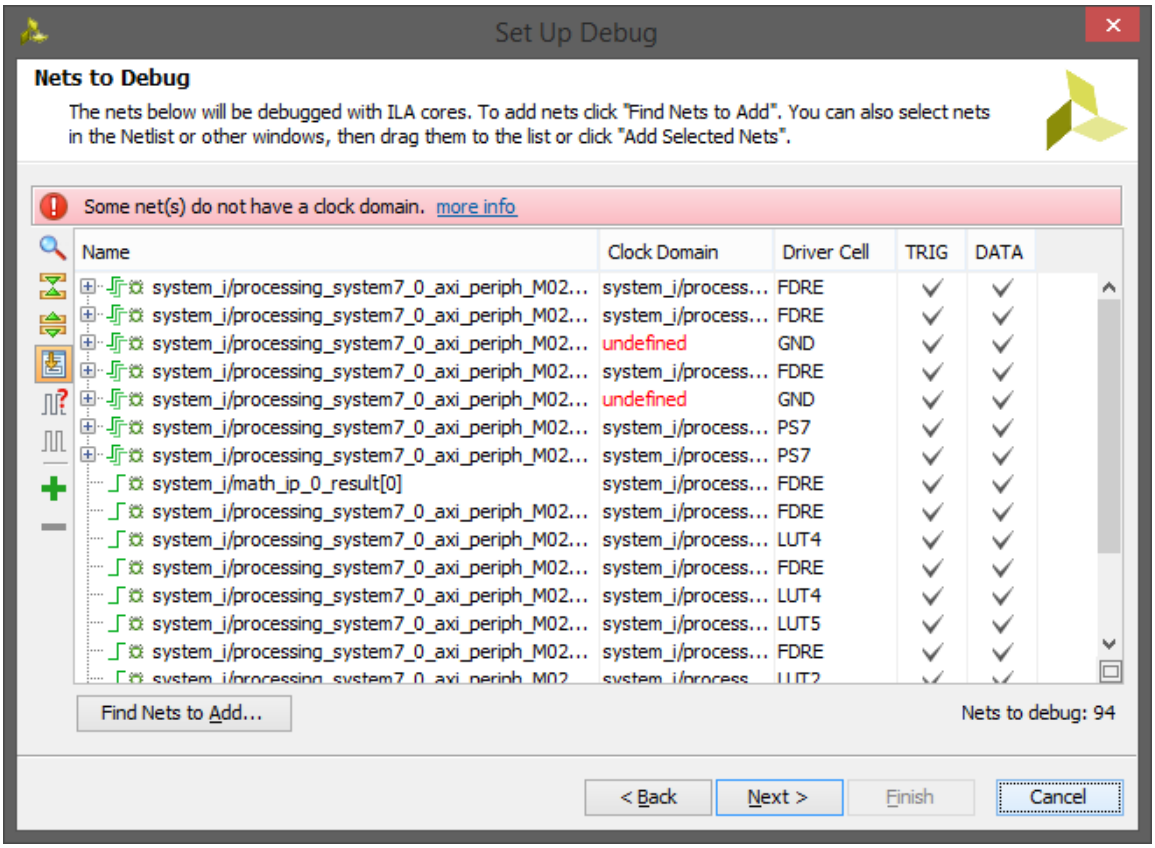


Figure 9. Nets to debug

4-2-3. Right click on the **BRESP** and **RRSEP** (which are driven by GND and marked in red as undefined) and select **Remove Nets**.

4-2-4. Click **Next,** leave the default values for the *ILA core,* click **Next,** and then **Finish**.

Generate Bitstream Step 5

5-1. Generate the bitstream.

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

5-1-2. Click **Save** to save the project (if prompted), **OK** to ignore the warning (if prompted), and **Yes** to launch Implementation (if prompted).

5-1-3. When the bitstream generation process has completed successfully, click **Cancel**.

Generate an Application in SDK Step 6

6-1. Export the implemented design and launch SDK.

6-1-1. Export the hardware configuration by clicking **File > Export > Export Hardware…**, click the box to *Include Bitstream*

6-1-2. Click **OK** to export and **Yes** to overwrite the previous project created by lab1.

6-1-3. Launch SDK by clicking **File > Launch SDK** and click **OK.**

6-1-4. 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.

6-2. Create an empty application project named lab2, and import the provided lab2.c file.

6-2-1. Select **File > New** > **Application Project.**

6-2-2. 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)**.**

6-2-3. Select the **Empty Application** template and click **Finish.**

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

6-2-4. Select **lab2 > src** in the project view, right-click, and select **Import.**

6-2-5. Expand the **General** category and double-click on **File System.**

6-2-6. Browse to the **c:\xup\adv_embedded\sources\lab2** folder.

6-2-7. 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 BTNR (Zedboard) or BTN3 (Zybo) to see the LED light --\r\n");
xil_printf("-- Change slide switches to see corresponding output on LEDs --\r\n");
xil_printf("-- Set slide switches to 0x0F 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)
{
    sw_check = XGpio_DiscreteRead(&sw, 1);
    XGpio_DiscreteWrite(&led, 1, sw_check);
    if ((sw_check & 0x0f) == 0xf)
      break;
    pshb_check = XGpioPs_ReadPin(&psGpioInstancePtr,iPinNumberEMIO);
    XGpioPs_WritePin(&psGpioInstancePtr,iPinNumber,pshb_check);
    for (i=0; i<9999999; i++); // delay loop
 }
```

Figure 10. Source Code snippet

6-2-8. Right click on *lab2*, and select **Debug As > Debug Configurations**

6-2-9. Double click on Xilinx C/C++ application (GDB) to create a new configuration (*lab2 Debug will be created*), and in the *Target Setup* tab, **Enable Cross-Triggering,** click **Apply,** then **Close**

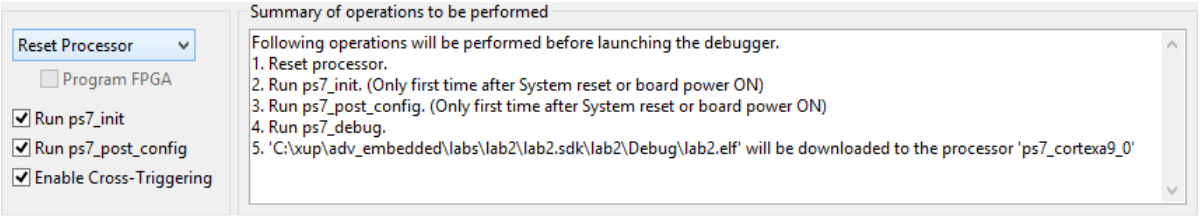


Figure 11. Enable cross triggering in the software environment

Test in Hardware — Step 7

7-1. Connect and power up the board. Establish serial communications using the SDK's Terminal tab. Download the bitstream into the target device. Start the debug session on lab2 project.

7-1-1. Connect and power up the board.

7-1-2. Select the Terminal tab. If it is not visible then select **Window > Show view > Terminal**.

7-1-3. Click on and select the appropriate COM port (depending on your computer), and configure it as you did it in Lab 1.

7-1-4. Select **Xilinx Tools** > **Program FPGA** and click **Program**

7-1-5. Select the **lab2** project in *Project Explorer*, right-click and select **Debug As > Launch on Hardware** (GDB) 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.

7-2. Start the hardware session from Vivado.

7-2-1. Switch to Vivado.

7-2-2. Click on **Open Hardware Manager** from the *Program and Debug* group of the *Flow Navigator* pane to invoke the analyzer.

7-2-3. Click on the **Open Target > Auto connect** to establish the connection with the board.

The hardware session will open showing the **Debug Probes** tab in the **Console** view. (If the probes are not displayed, right click on the XC7Z010/ XC7Z020 and select **Window > Debug Probes**)

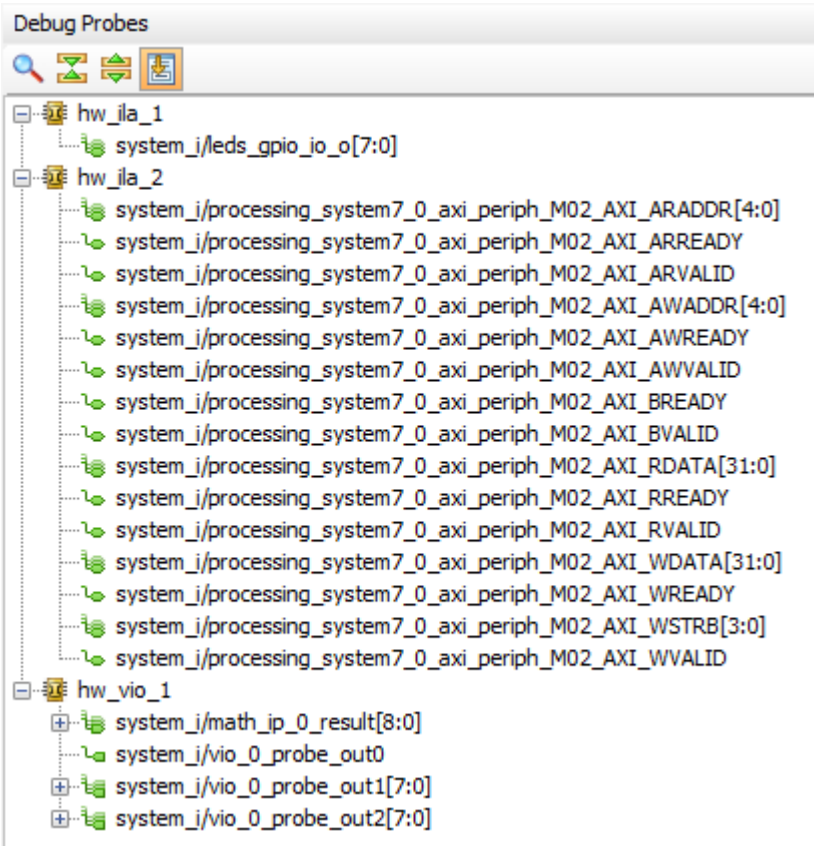


Figure 12. Debug probes

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

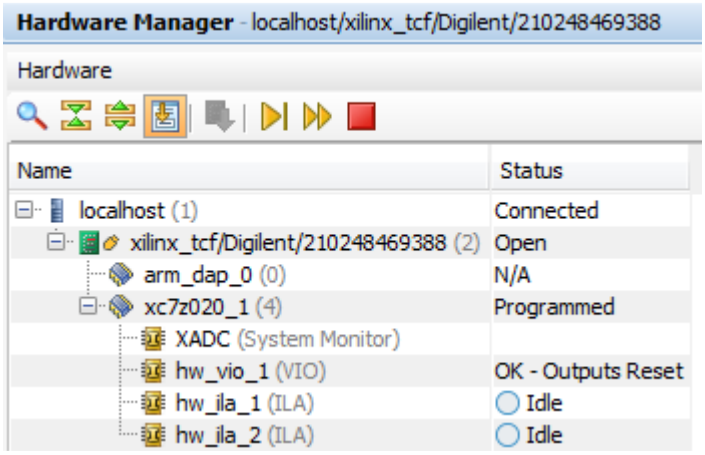


Figure 13 Hardware session status (for Zedboard - XC7Z020)

7-2-4. Select the XC7Z010/ XC7Z020, and click on the **Run Trigger Immediate** button to see the signals in the waveform window.

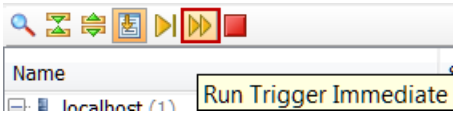


Figure 14. Opening the waveform window

7-3. Setup trigger conditions to trigger on a write transaction (WSTRB) when the valid address (AWADDR) is written with data (WVALID equal to 1).

7-3-1. Click on the hw_ila_2 tab to select it. In the **Debug Probes** window, under *hw_ila_2*, drag and drop the **AWADDR[4:0]** signal to the *ILA Basic Trigger setup* window.

7-3-2. Change the value from xx to 04 (HEX) (the slave_reg2 address of the math_0 instance).

7-3-3. Add **WSTRB[3:0]** and **WVALID** signals to the *ILA Basic Trigger setup* window.

7-3-4. Change the radix to binary for *WSTRB*, and change the value from **xxxx** to **xxx1**

7-3-5. Change the value of **WVALID** to 1.

7-3-6. Set the trigger position of the *hw_ila_2* to **512**.

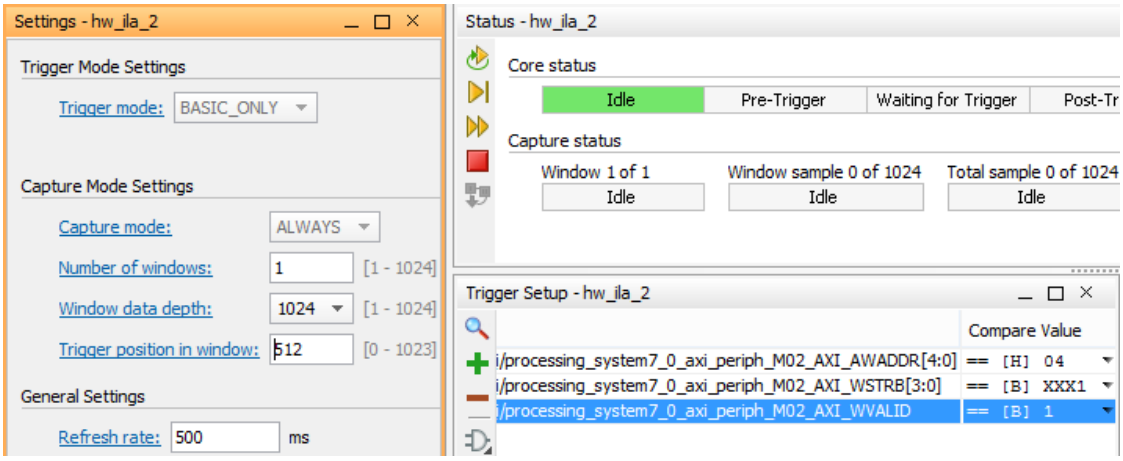


Figure 15. Setting up the ILA

7-3-7. Similarly, set the trigger position of the *hw_ila_1* to **512**.

7-3-8. 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**.

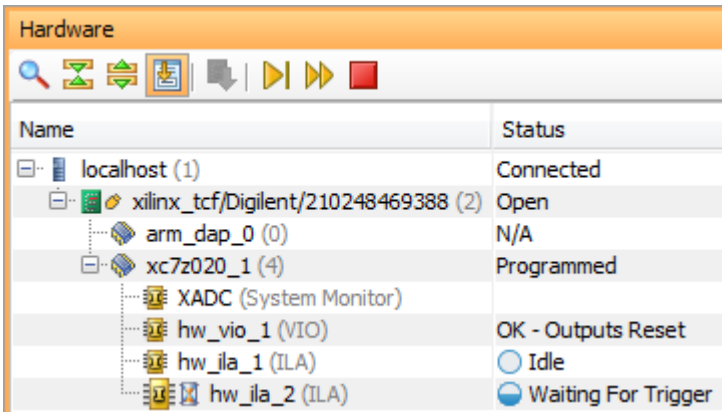


Figure 16. Hardware analyzer running and in capture mode

7-3-9. Switch to SDK.

7-3-10. Near line 54 (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.

```
51      Xil_Out32(XPAR_MATH_IP_0_BASEADDR, 0x12);
52      Xil_Out32(XPAR_MATH_IP_0_BASEADDR+4, 0x34);
53      i=Xil_In32(XPAR_MATH_IP_0_BASEADDR);
54      xil_printf("result=%x\r\n",i);
55
56      while (1)
57      {
58          sw_check = XGpio_DiscreteRead(&sw, 1);
```

Figure 17. Setting a breakpoint

7-3-11. Click on the **Resume** ( ) button to execute the program and stop at the breakpoint.

7-3-12. In the Vivado program, notice that the **hw_ila_2** status changed from *capturing* to *Idle*, and the waveform window shows the triggered output (select the *hw_ila_data_2.wcfg* tab if necessary).

7-3-13. 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.

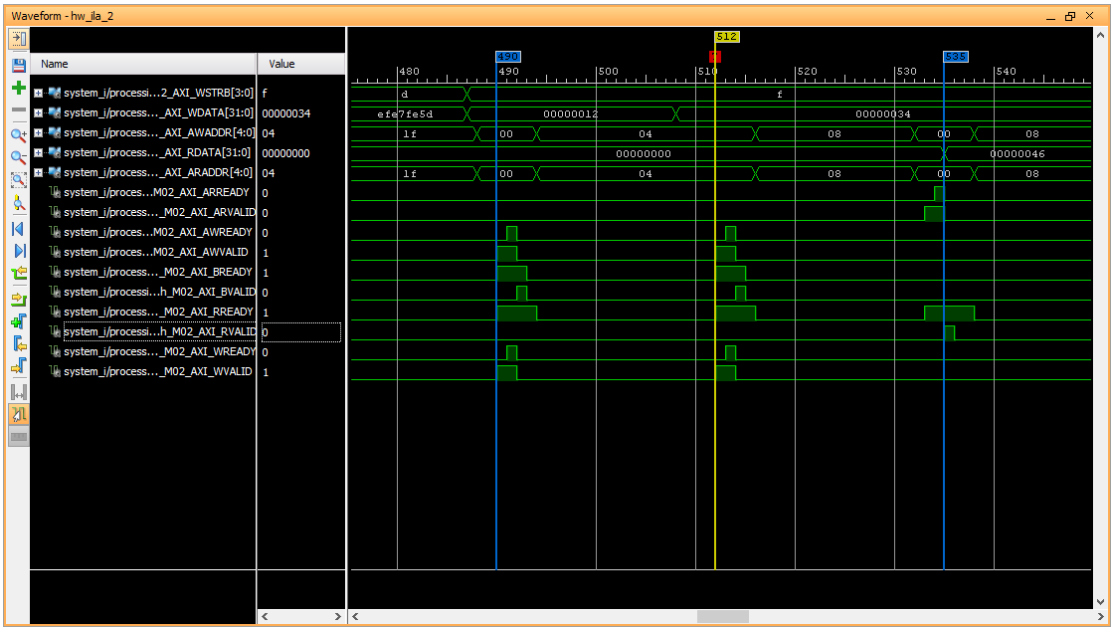


Figure 18. Zoomed waveform view of the three AXI transactions

Observe the following:

Around the 490th sample the RDATA value is 0x00, WDATA being written is 0x012 at offset 0 (AWADDR=0x0), WVALID is '1', WREADY '1' indicating the data is being written into the IP.

At the 512th sample, WVALID is '1' WSTRB is 0xf, offset is 0x4 (AWADDR), and the data being written is 0x034.

At the 535th sample, RREADY and RVALID are '1' indicating data (result=0x046) is being read from the IP at the offset 0x0 (ARADDR).

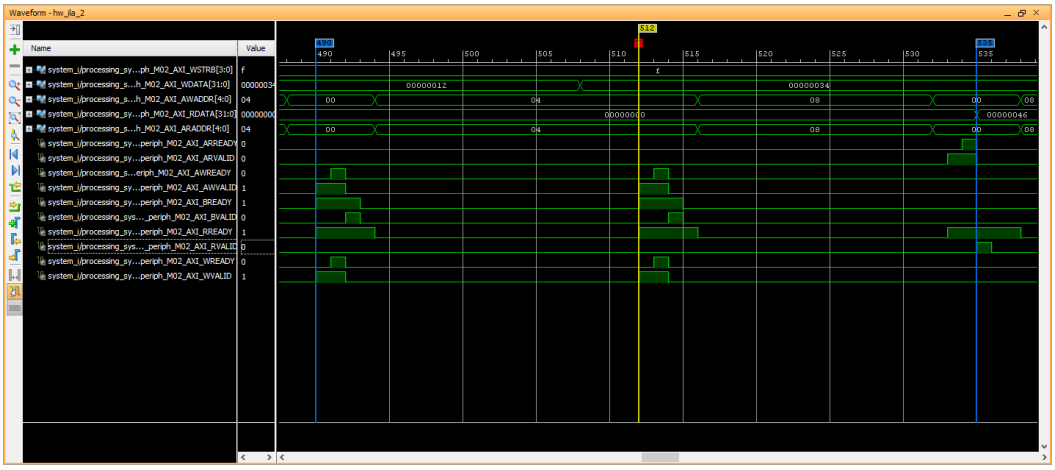


Figure 19. Zoomed view of the hw_ila_2 waveform window

7-3-14. You also should see the following output in the SDK Terminal console.

```
-- Press BTNR (Zedboard) or BTN3 (Zybo) to see the LED light --
-- Change slide switches to see corresponding output on LEDs --
-- Set slide switches to 0x0F to exit the program --
```

Figure 20. SDK Terminal Output

7-4. In Vivado, select the VIO Cores in Console, 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.

7-4-1. Select the **hw_vio_1** core in the Debug Probes panel

7-4-2. Drag and drop **vio_0_probe_out0** to the pane on the right, and change its value to **1** so the math_ip core input can be controlled via the VIO core.

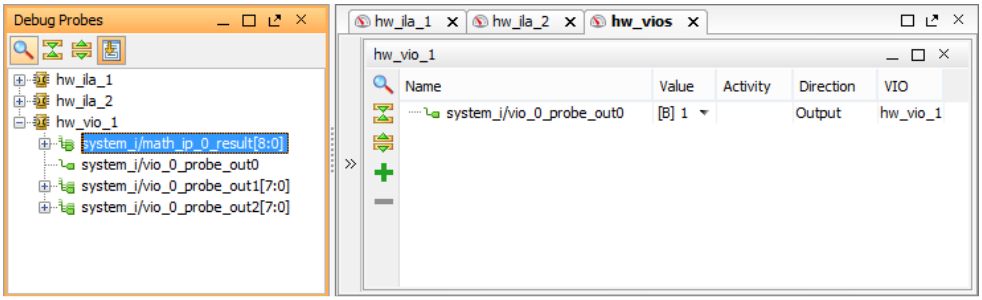


Figure 21. VIO probes

7-4-3. Also add **math_ip_0_result**, then add **vio_0_probe_out1** and change its value to **55** (in Hex), and similarly, add **vio_0_probe_out2** and change its 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).

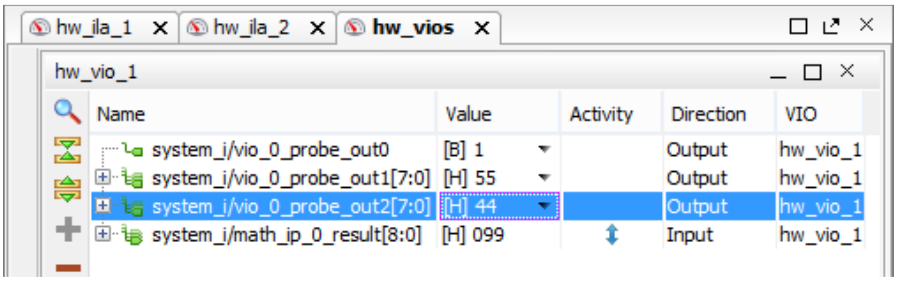


Figure 22. Input stimuli through the VIO core's probes

7-4-4. Try a few other inputs and observe the outputs.

7-4-5. Once done, set the *vio_0_probe_out0* to **0** to isolate the vio interactions with the math_ip core.

7-5. Setup the ILA core (hw_ila_1) trigger condition to 0101_0101 (0x55) for the Zedboard or 0101 (0x5) for the Zybo. Make sure that the switches on the board are not set at x55. Set the trigger equation to be ==, and arm the trigger. Click on the Resume button in the SDK to continue executing th eprogram. Change the switches and observe that the hardware core triggers when the preset condition is met.

7-5-1. Select the **hw_ila_1** in the *Debug Probes* panel.

7-5-2. 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 0x**55** for the Zedboard or **0x5** for the Zybo.

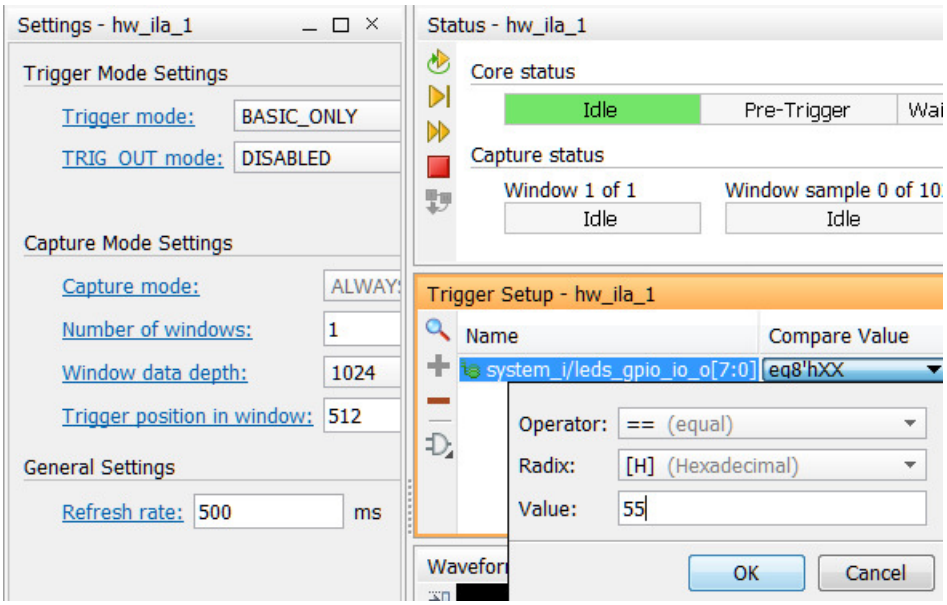


Figure 23. Setting up Trigger for hw_ila_1

7-5-3. Ensure that the trigger position for the *hw_ila_1* is set to **512**.

Make sure that the switches are not set to 01010101 (Zedboard) or 0101 (Zybo) as this is the exit pattern.

7-5-4. 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.

7-5-5. In the SDK window, click on the *Resume* button.

7-5-6. Change the slide switches and see the corresponding LED turning ON and OFF.

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

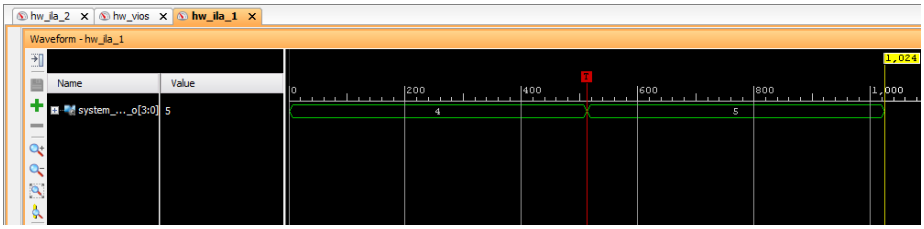


Figure 24. ILA waveform window after Trigger

7-6. Cross trigger a debug session between the HW and software

7-6-1. In Vivado, select *hw_ila_1*

7-6-2. In the ILA properties, set the *Trigger mode* to **BASIC_OR_TRIGG_IN**, and the *TRIG_OUT* mode to **TRIGGER_OR_TRIG_IN** (Hover over these values to display a description of the options)

7-6-3. In SDK, in the C/C++ view, relaunch the software by right clicking on the lab2 project, and selecting Debug As > Launch on Hardware (GDB) (Click **OK** if prompted to reset the processor)

7-6-4. In SDK continue execution of the software to the next breakpoint

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

7-6-6. In SDK continue execution of the software to the next breakpoint

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

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

7-7-1. Run the software (F8) until it enters the while loop

7-7-2. Verify it is executing by toggling the dip switches

7-7-3. In Vivado, ARM the *hw_ila_1* trigger (from earlier, the trigger condition is 0x55 on the LEDS)

7-7-4. Toggle the dip switches to 0x55, and notice that the application in SDK will break at some point (This point will be somewhere within the while loop)

7-7-5. Click the Terminate button ( ) in the SDK to terminate the execution.

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

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

7-7-8. Close Vivado program by selecting **File > Exit**.

7-7-9. 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.

.