EGR680
Fall 2018
Laboratory #9

HW/SW System Debug

Objectives

After completing this laboratory, you will be able to:

- Mark nets as debug so AXI transactions and GPIO port can be monitored
- Perform hardware debugging using the hardware analyzer
- Perform software debugging using the SDK

Introduction

Software and hardware interact with each other in an embedded system. The Xilinx SDK includes System Debugger as software debugging tools. The hardware analyzer tool has various 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.

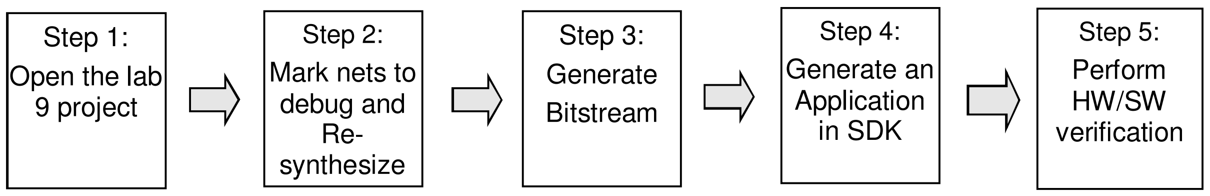
Procedure

This lab is separated into steps that consist of general overview statements that provide information on the detailed instructions that follow. Follow these detailed instructions to progress through the lab.

Design Description

You will extend the system created in one of the previous lab by adding a debug hub (ILA Core) to the design using the mark nets feature in Vivado (see Figure 1). The debug core will be added to monitor the AXI4-Lite bus of the GPIO control signals going to the on-board Switches and Buttons. You will set trigger conditions in the Hardware Manager under the ILA settings (running on PC) to capture bus transactions when the value of the count variable is written to the LEDs. When the hardware trigger condition is met, you will see that the software debugger stops at the line of code that was last executed.

General Flow for this Lab


Step 1: Open the lab 9 project
Step 2: Mark nets to debug and Re-synthesize
Step 3: Generate Bitstream
Step 4: Generate an Application in SDK
Step 5: Perform HW/SW verification

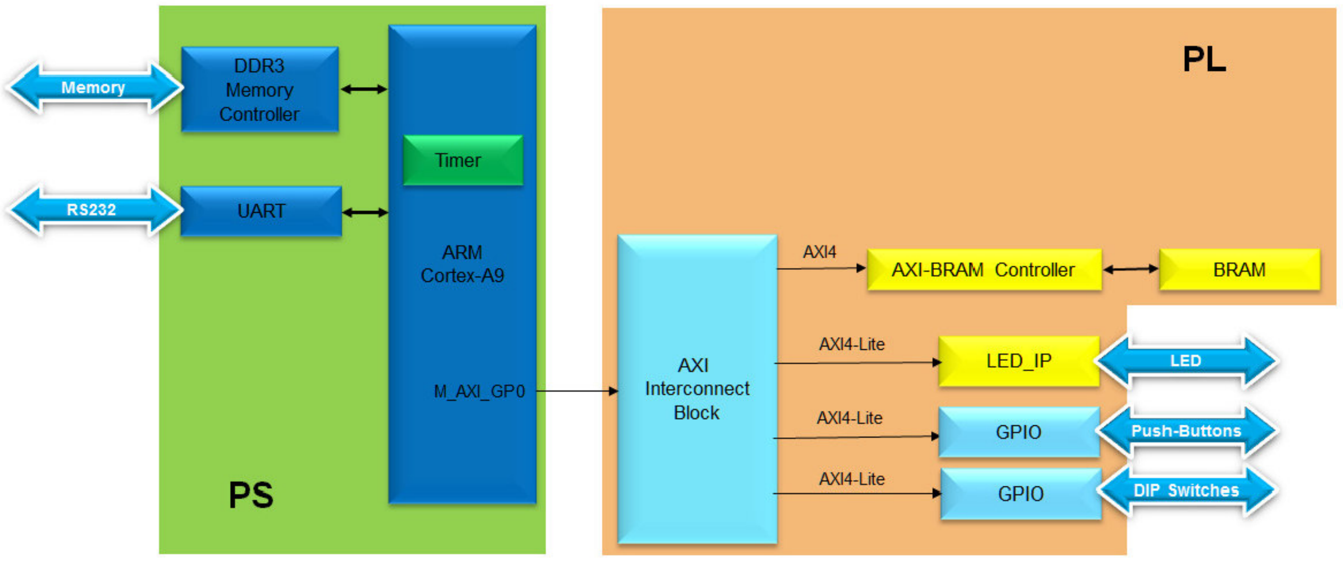


Figure 1. Design from the previous lab

Step 1: Open the project

1. Download ***GPIO_App*** compressed folder from the course website. Unzip it.

2. Start **Vivado** and open **GPIO_App** project (*GPIO_App.xpr*).

3. Click on ***Open Block Design*** under **IP Integrator**.

Step 2: Add ILA core

1. Double-click the ***leds*** instance to open its configuration form. Click on **Clear Board Parameters** followed by **OK** to close the configuration form.

2. Select ***leds*** port and delete it. Expand the **GPIO** interface of ***leds*** instance to see the associate ports.

3. 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**. See Figure 2.

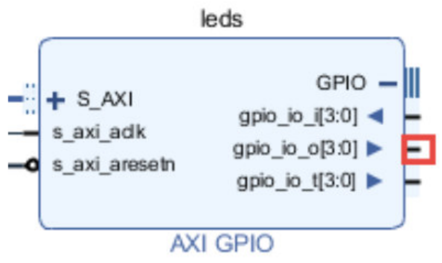
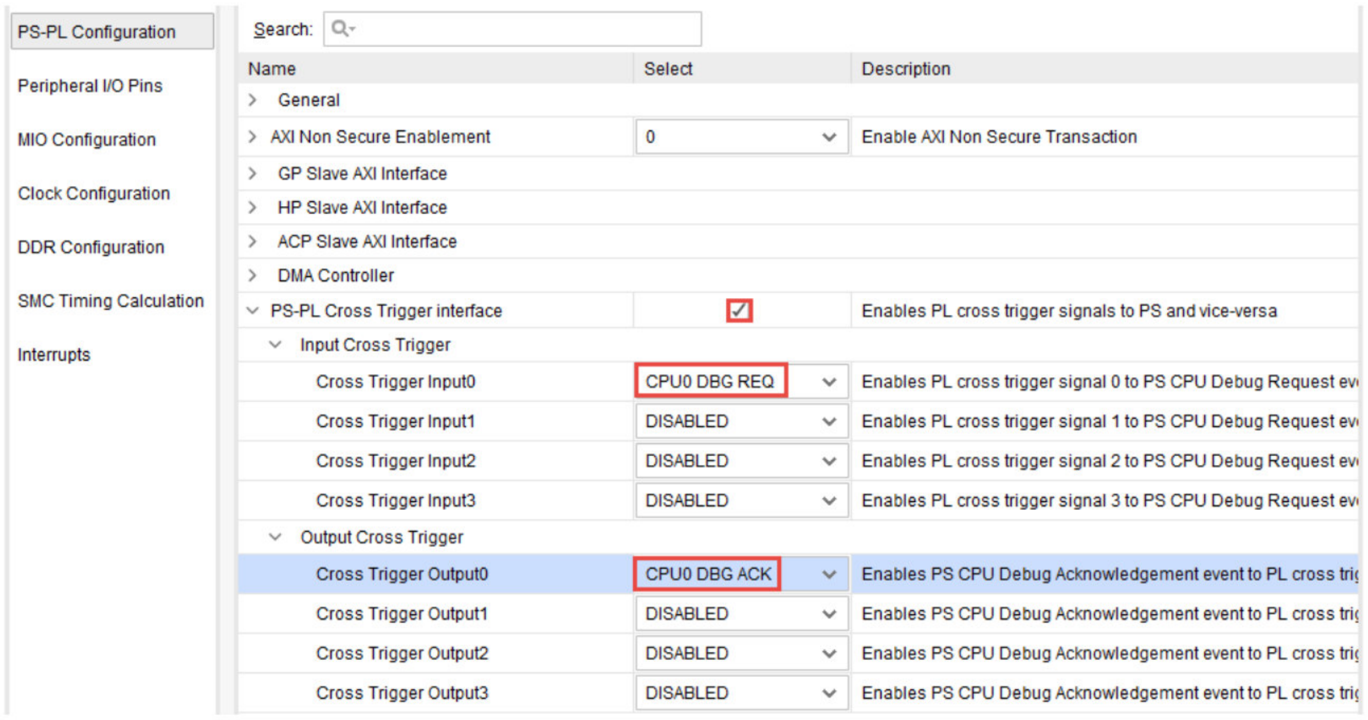


Figure 2: Select the gpio_io_o port

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

5. Double-click on ***ZYNQ*** block to open the configuration properties. Now click on **PS-PL Configuration** and enable the **PS-PL Cross Trigger Interface**.

6. Expand ***PS-PL Cross Trigger interface → Input Cross Trigger***, and select **CPU0 DBG REQ** for ***Cross Trigger Input 0***. Similarly, expand **Output Cross Trigger**, and select ***CPU0 DBG ACK*** for **Cross Trigger Output 0** and click ***OK***. See Figure 3.


PS-PL Configuration
Peripheral I/O Pins
MIO Configuration
Clock Configuration
DDR Configuration
SMC Timing Calculation
Interrupts

Search:

Name	Select	Description
> General		
> AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
> GP Slave AXI Interface		
> HP Slave AXI Interface		
> ACP Slave AXI Interface		
> DMA Controller		
⌄ PS-PL Cross Trigger interface	✓	Enables PL cross trigger signals to PS and vice-versa
⌄ Input Cross Trigger		
Cross Trigger Input0	CPU0 DBG REQ	Enables PL cross trigger signal 0 to PS CPU Debug Request ev
Cross Trigger Input1	DISABLED	Enables PL cross trigger signal 1 to PS CPU Debug Request ev
Cross Trigger Input2	DISABLED	Enables PL cross trigger signal 2 to PS CPU Debug Request ev
Cross Trigger Input3	DISABLED	Enables PL cross trigger signal 3 to PS CPU Debug Request ev
⌄ Output Cross Trigger		
Cross Trigger Output0	CPU0 DBG ACK	Enables PS CPU Debug Acknowledgement event to PL cross tri
Cross Trigger Output1	DISABLED	Enables PS CPU Debug Acknowledgement event to PL cross tri
Cross Trigger Output2	DISABLED	Enables PS CPU Debug Acknowledgement event to PL cross tri
Cross Trigger Output3	DISABLED	Enables PS CPU Debug Acknowledgement event to PL cross tri

Figure 3: Enabling cross triggering in the ZYNQ processing system

7. Click the **+** button and search for ***ila*** in the catalog. Double-click on the ***ILA (Integrated Logic Analyzer)*** to add an instance of it. The **ila_0** instance will be added.

8. Double-click on the ***ila_0*** instance. Select ***Native*** as the **Monitor** type**.** Enable ***Trigger Out Port***, and ***Trigger In port***. Select the ***Probe Ports*** tab, and set the ***Probe Width*** of **PROBE0** to ***4*** and click **OK**.

9. Using the drawing tool, connect the ***PROBE0*** port of the **ila_0** instance to the ***gpio_io_o*** port of the **leds** instance.

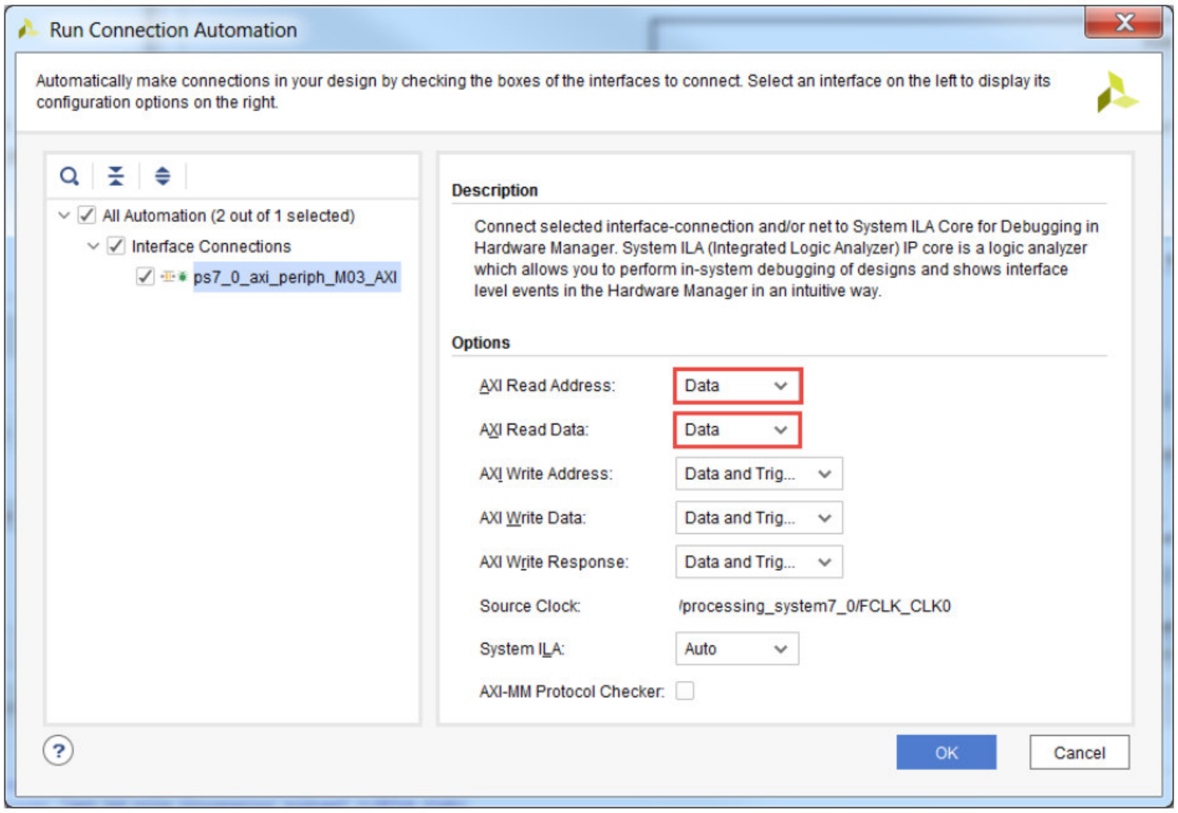
10. Connect the ***clk*** port of the **ila_0** instance to the ***FCLK_CLK0*** port of the Zynq subsystem.

11. 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***.

12. Select the **S_AXI** connection between the AXI Interconnect and the ***leds*** instance. Right-click and select **Debug** to monitor the AXI4-Lite transactions.

13. Now select **Run Connection Automation**.

14. Change ***AXI Read Address*** and ***AXI Read Data*** channels to **Data** since we will not trigger any signals of those channels. <u>**This saves resources being used by the design**</u>. See Figure 4.


Run Connection Automation

Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

- ✓ All Automation (2 out of 1 selected)
  - ✓ Interface Connections
    - ✓ ps7_0_axi_periph_M03_AXI

Description

Connect selected interface-connection and/or net to System ILA Core for Debugging in Hardware Manager. System ILA (Integrated Logic Analyzer) IP core is a logic analyzer which allows you to perform in-system debugging of designs and shows interface level events in the Hardware Manager in an intuitive way.

Options

AXI Read Address:	Data
AXI Read Data:	Data
AXI Write Address:	Data and Trig...
AXI Write Data:	Data and Trig...
AXI Write Response:	Data and Trig...
Source Clock:	/processing_system7_0/FCLK_CLK0
System ILA:	Auto
AXI-MM Protocol Checker:	☐

OK Cancel

Figure 4: Selecting channels for debugging

15. Select **Tools → Validate Design** to run the design rules checker. Verify that there are no errors or critical warnings. Your final design should look similar to Figure 5.

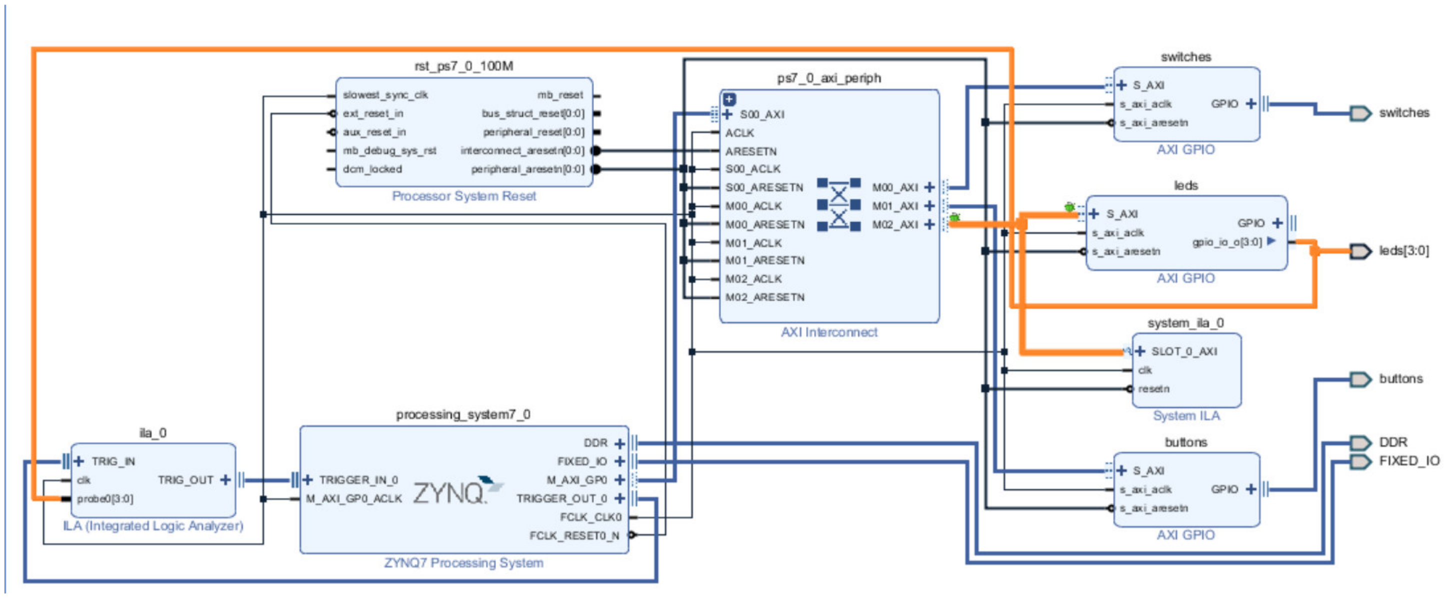
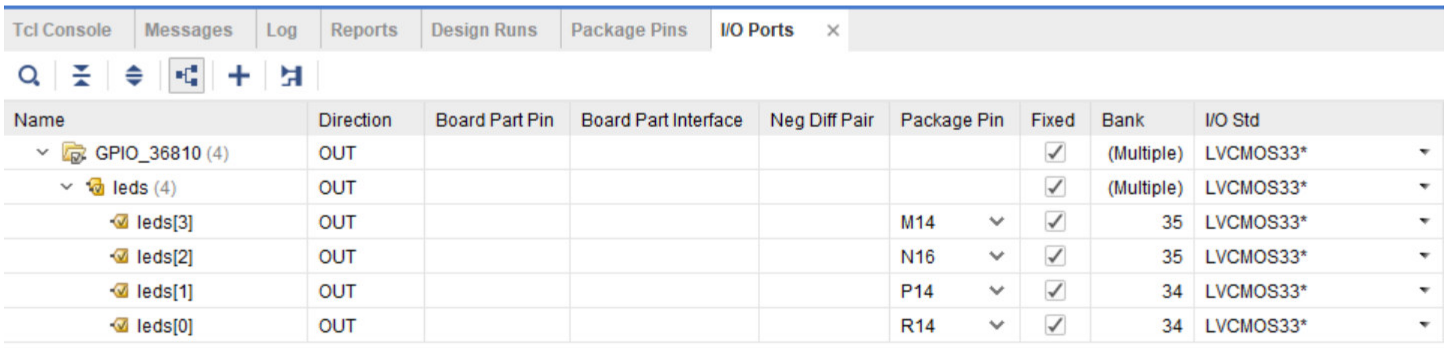


Figure 5: Finalized version of Block design

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

17. When the synthesis is completed, select the ***Open Synthesized Design*** option and click **OK.**

18. The synthesized design will be opened in the Auxiliary pane. Click on **I/O Ports** tab and enter the following pin assignments for ***leds*** as shown in Figure 6.



Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std
GPIO_36810 (4)	OUT					✓	(Multiple)	LVCMOS33*
leds (4)	OUT					✓	(Multiple)	LVCMOS33*
leds[3]	OUT				M14	✓	35	LVCMOS33*
leds[2]	OUT				N16	✓	35	LVCMOS33*
leds[1]	OUT				P14	✓	34	LVCMOS33*
leds[0]	OUT				R14	✓	34	LVCMOS33*

Figure 6: Pin assignments for leds

19. Save and close the Synthesized design.

Step 3: Generate Bitstream

1. Now click on the **Generate Bitstream** to run the implementation and bit generation processes.

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

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

4. Now go to your project folder ***GPIO_App*** and rename **GPIO_App.sdk** folder to **GPIO_App.sdk1**. This will allow you to create a new SDK folder and still be able to use the software code that is already written.

Step 4: Generate an Application in SDK

1. Export the hardware configuration by clicking **File → Export → Export Hardware…**, click the box ***Include Bitstream.***

2. Click **OK** to export.

3. Launch SDK by clicking **File → Launch SDK** and click **OK.**

4. Select **File → New → Application Project**

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

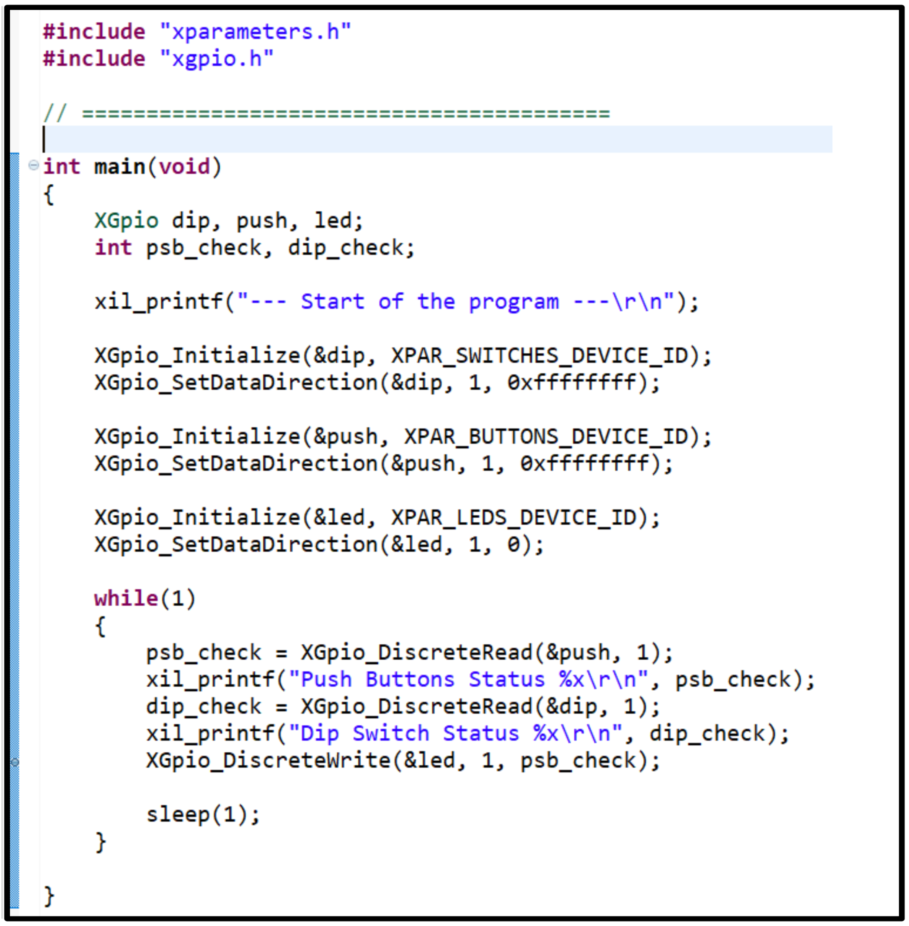
6. Select the **Empty Application** template and click **Finish.**

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

7. Expand **GPIO_Debug** in the project view and right-click in the ***src folder*** and select **New → Source File.**

8. Under **Source file** field, provide name as ***GPIO_Status.c*** and click **Finish**.

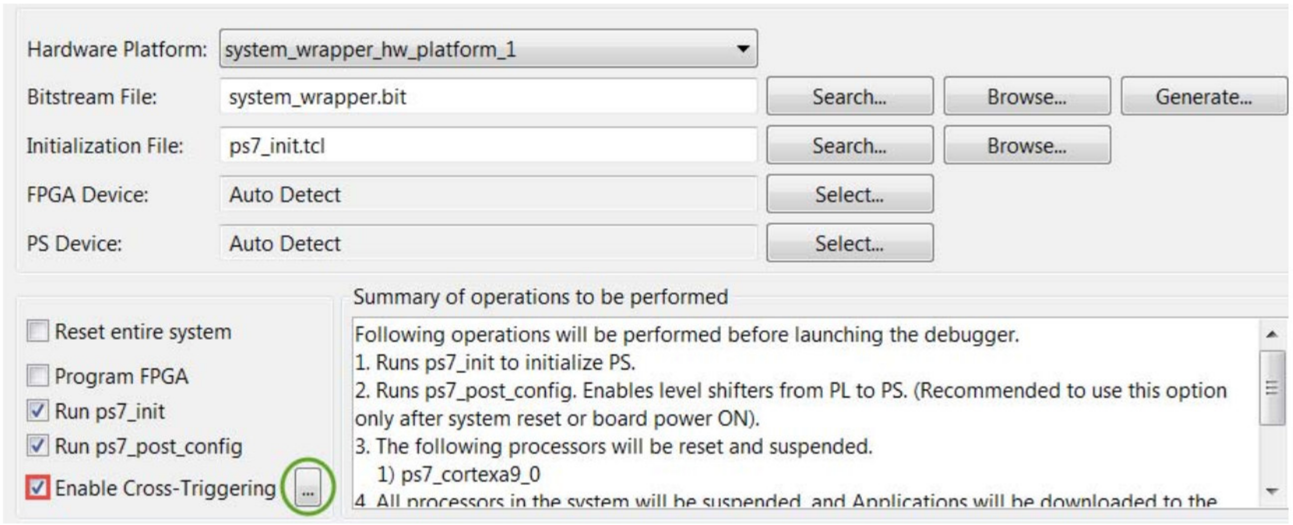
9. Type the following code in ***GPIO_Status.c*** file and then **Save** the file.



```
#include "xparameters.h"
#include "xgpio.h"

// ========================================

int main(void)
{
    XGpio dip, push, led;
    int psb_check, dip_check;

    xil_printf("--- Start of the program ---\r\n");

    XGpio_Initialize(&dip, XPAR_SWITCHES_DEVICE_ID);
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    XGpio_Initialize(&push, XPAR_BUTTONS_DEVICE_ID);
    XGpio_SetDataDirection(&push, 1, 0xffffffff);

    XGpio_Initialize(&led, XPAR_LEDS_DEVICE_ID);
    XGpio_SetDataDirection(&led, 1, 0);

    while(1)
    {
        psb_check = XGpio_DiscreteRead(&push, 1);
        xil_printf("Push Buttons Status %x\r\n", psb_check);
        dip_check = XGpio_DiscreteRead(&dip, 1);
        xil_printf("Dip Switch Status %x\r\n", dip_check);
        XGpio_DiscreteWrite(&led, 1, psb_check);

        sleep(1);
    }

}
```

Figure 7: C Program for checking and displaying button status on leds

10. Right click on ***GPIO_Debug*** folder and select **Debug As → Debug Configurations.** Double click on **Xilinx C/C++ application (System Debugger)** to create a new configuration (***GPIO_Debug Debug*** *will be created*), and in the ***Target Setup*** tab, check the **Enable Cross-Triggering** option**,** and click the **Browse** button as shown in Figure 8.



Hardware Platform: system_wrapper_hw_platform_1
Bitstream File: system_wrapper.bit
Initialization File: ps7_init.tcl
FPGA Device: Auto Detect
PS Device: Auto Detect

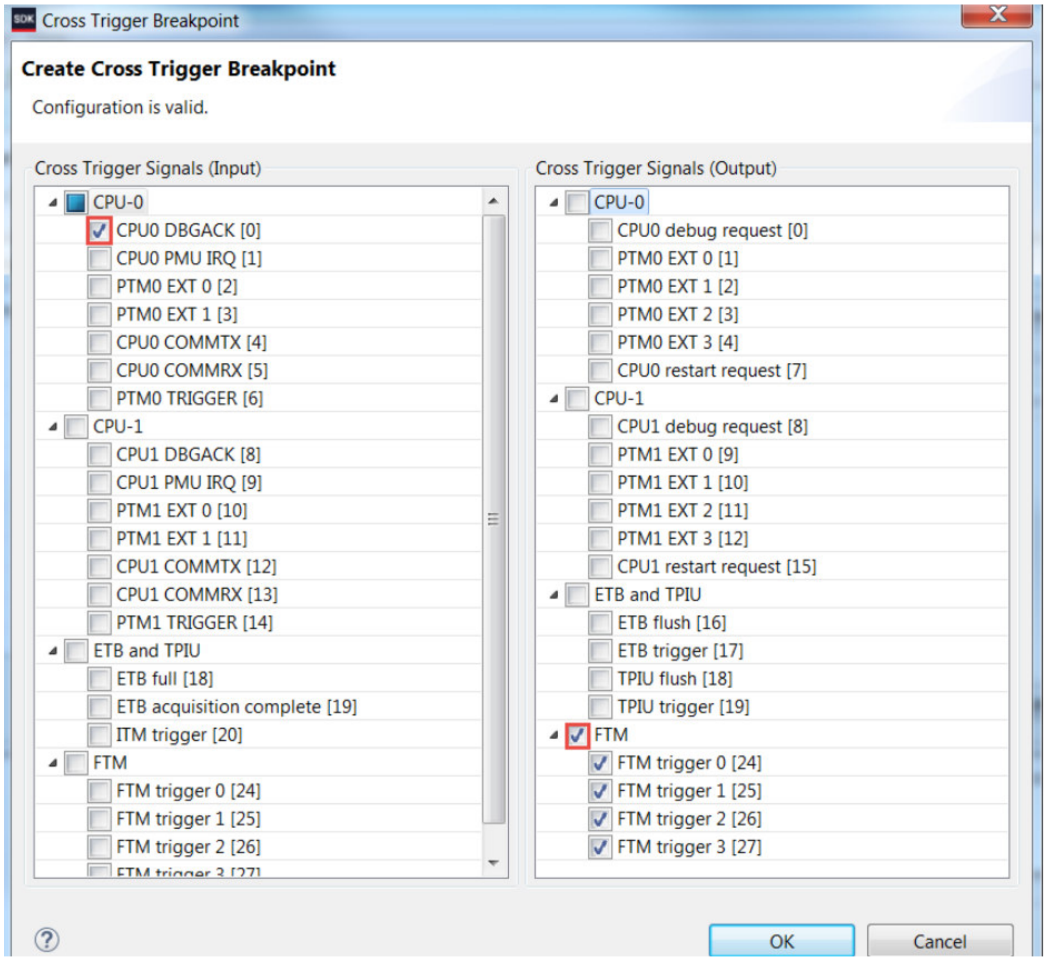
- [ ] Reset entire system
- [ ] Program FPGA
- [x] Run ps7_init
- [x] Run ps7_post_config
- [x] Enable Cross-Triggering

Summary of operations to be performed

Following operations will be performed before launching the debugger.
1. Runs ps7_init to initialize PS.
2. Runs ps7_post_config. Enables level shifters from PL to PS. (Recommended to use this option only after system reset or board power ON).
3. The following processors will be reset and suspended.
   1) ps7_cortexa9_0
4. All processors in the system will be suspended, and Applications will be downloaded to the

Figure 8: Enable cross triggering in the software environment

11. When the ***Cross Trigger Breakpoints*** dialog box opens, click **Create.** Select the options as shown in Figure 9 and click **OK** to set up the cross-trigger condition for Processor to Fabric.



Create Cross Trigger Breakpoint

Configuration is valid.

Cross Trigger Signals (Input)

- ■ CPU-0
  - ✓ CPU0 DBGACK [0]
  - CPU0 PMU IRQ [1]
  - PTM0 EXT 0 [2]
  - PTM0 EXT 1 [3]
  - CPU0 COMMTX [4]
  - CPU0 COMMRX [5]
  - PTM0 TRIGGER [6]
- CPU-1
  - CPU1 DBGACK [8]
  - CPU1 PMU IRQ [9]
  - PTM1 EXT 0 [10]
  - PTM1 EXT 1 [11]
  - CPU1 COMMTX [12]
  - CPU1 COMMRX [13]
  - PTM1 TRIGGER [14]
- ETB and TPIU
  - ETB full [18]
  - ETB acquisition complete [19]
  - ITM trigger [20]
- FTM
  - FTM trigger 0 [24]
  - FTM trigger 1 [25]
  - FTM trigger 2 [26]
  - FTM trigger 3 [27]

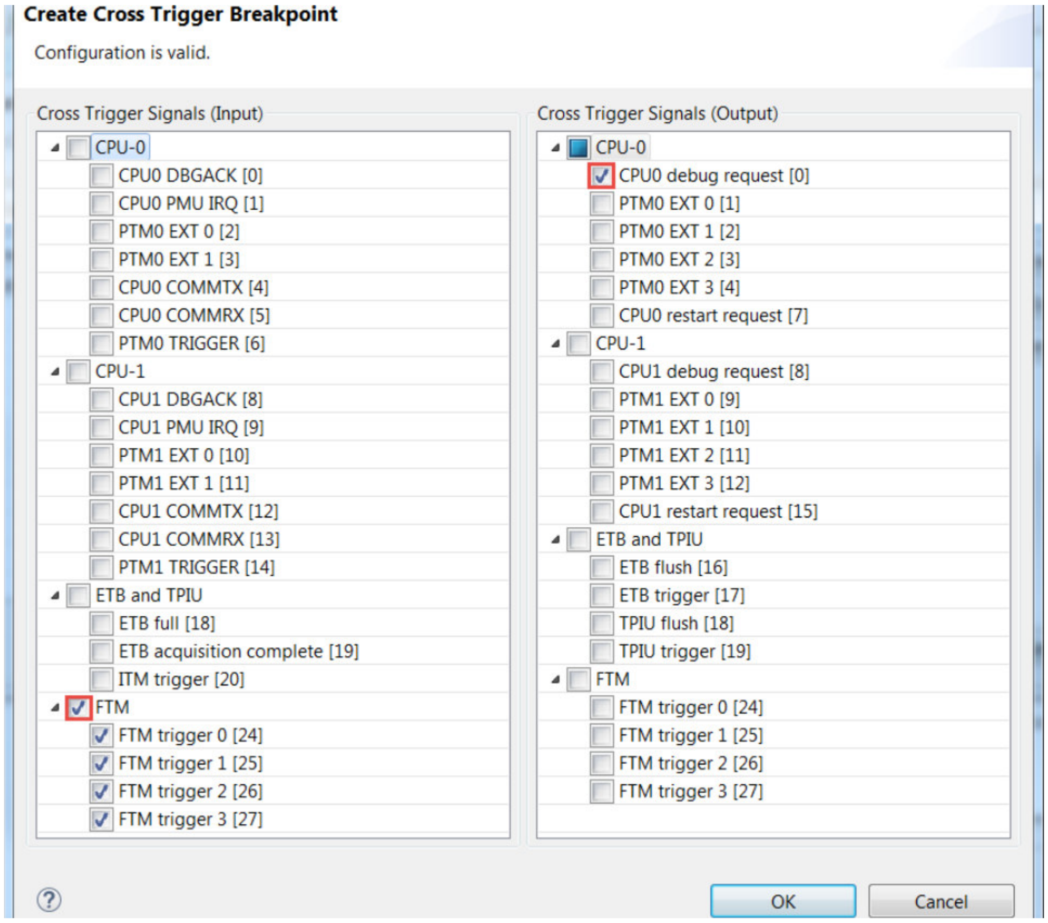
Cross Trigger Signals (Output)

- CPU-0
  - CPU0 debug request [0]
  - PTM0 EXT 0 [1]
  - PTM0 EXT 1 [2]
  - PTM0 EXT 2 [3]
  - PTM0 EXT 3 [4]
  - CPU0 restart request [7]
- CPU-1
  - CPU1 debug request [8]
  - PTM1 EXT 0 [9]
  - PTM1 EXT 1 [10]
  - PTM1 EXT 2 [11]
  - PTM1 EXT 3 [12]
  - CPU1 restart request [15]
- ETB and TPIU
  - ETB flush [16]
  - ETB trigger [17]
  - TPIU flush [18]
  - TPIU trigger [19]
- ✓ FTM
  - ✓ FTM trigger 0 [24]
  - ✓ FTM trigger 1 [25]
  - ✓ FTM trigger 2 [26]
  - ✓ FTM trigger 3 [27]

Figure 9: Enabling CPU0 for request from PL

12. In the ***Cross Trigger Breakpoints*** dialog box click **Create** again. Select the options as shown in Figure 10 and click **OK** to set up the cross trigger condition for Fabric to Processor.

13. Click **OK**, then click **Apply**, then **Close**.



Create Cross Trigger Breakpoint

Configuration is valid.

Cross Trigger Signals (Input)

- CPU-0
  - CPU0 DBGACK [0]
  - CPU0 PMU IRQ [1]
  - PTM0 EXT 0 [2]
  - PTM0 EXT 1 [3]
  - CPU0 COMMTX [4]
  - CPU0 COMMRX [5]
  - PTM0 TRIGGER [6]
- CPU-1
  - CPU1 DBGACK [8]
  - CPU1 PMU IRQ [9]
  - PTM1 EXT 0 [10]
  - PTM1 EXT 1 [11]
  - CPU1 COMMTX [12]
  - CPU1 COMMRX [13]
  - PTM1 TRIGGER [14]
- ETB and TPIU
  - ETB full [18]
  - ETB acquisition complete [19]
  - ITM trigger [20]
- [✓] FTM
  - [✓] FTM trigger 0 [24]
  - [✓] FTM trigger 1 [25]
  - [✓] FTM trigger 2 [26]
  - [✓] FTM trigger 3 [27]

Cross Trigger Signals (Output)

- CPU-0
  - [✓] CPU0 debug request [0]
  - PTM0 EXT 0 [1]
  - PTM0 EXT 1 [2]
  - PTM0 EXT 2 [3]
  - PTM0 EXT 3 [4]
  - CPU0 restart request [7]
- CPU-1
  - CPU1 debug request [8]
  - PTM1 EXT 0 [9]
  - PTM1 EXT 1 [10]
  - PTM1 EXT 2 [11]
  - PTM1 EXT 3 [12]
  - CPU1 restart request [15]
- ETB and TPIU
  - ETB flush [16]
  - ETB trigger [17]
  - TPIU flush [18]
  - TPIU trigger [19]
- FTM
  - FTM trigger 0 [24]
  - FTM trigger 1 [25]
  - FTM trigger 2 [26]
  - FTM trigger 3 [27]

Figure 10: Enabling CPU0 for request to PL

Step 5: Perform HW/SW Verification

1. Make sure that the **JP5** is set to select **USB** power and **JP4** is set to **JTAG** mode. Connect the board with a micro-usb cable and power it ON.

2. Open the SDK Terminal tab or **Tera-Term**.

3. Select **Xilinx Tools → Program FPGA.** Then click the **Program** button to program the FPGA.

   This will execute ***Data2Mem*** program to combine the ***ELF/MEM*** file with hardware ***bitstream***, generate the ***system_wrapper.bit*** file, and configure the FPGA.

4. Select **GPIO_Debug** in ***Project Explorer***, right-click and select **Debug As → Launch on Hardware (System Debugger)**.

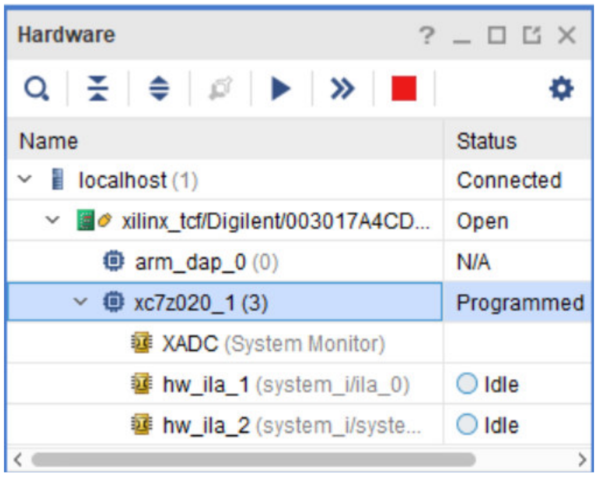
   The ***ELF*** file will be downloaded and if prompted, click **OK** to switch to the ***Debug perspective.***

5. Switch to Vivado.

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

7. Click on the **Open Target → Auto connect** to establish the connection with the board.

8. The hardware session status window also opens (Figure 11) showing that the FPGA is programmed (we did it in SDK).



Name	Status
localhost (1)	Connected
xilinx_tcf/Digilent/003017A4CD...	Open
arm_dap_0 (0)	N/A
xc7z020_1 (3)	Programmed
XADC (System Monitor)	
hw_ila_1 (system_i/ila_0)	Idle
hw_ila_2 (system_i/syste...	Idle

Figure 11: Hardware Session status

9. Select the ***xc7z020_1*** and click on the **Run Trigger Immediate** button to see the signals in the waveform window (Figure 12).

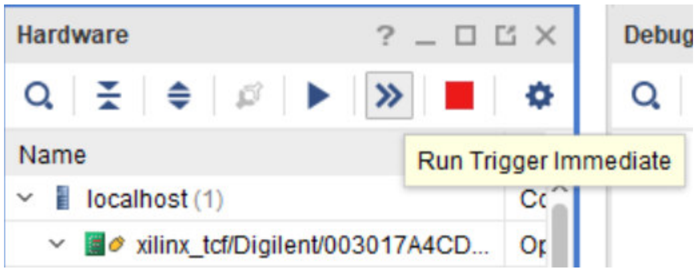
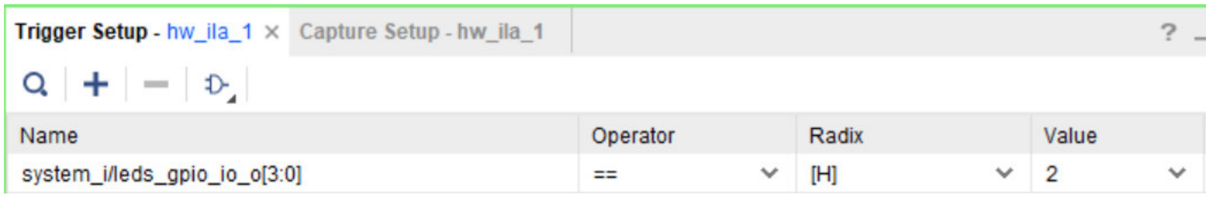


Figure 12: Opening the waveform window

10. Click on the ***hw_ila_1*** tab to select it. In the **Trigger Setup** window, click the **+** button to add Debug probes. Once presented with the list of signals, select ***leds_gpio_io_o[3:0]*** and click **OK**. These signals will be added to the **Trigger Setup** window.

11. Set compare value parameters **leds_gpio_io_o[3:0] == 2** which is in radix Hexadecimal and See Figure 13.

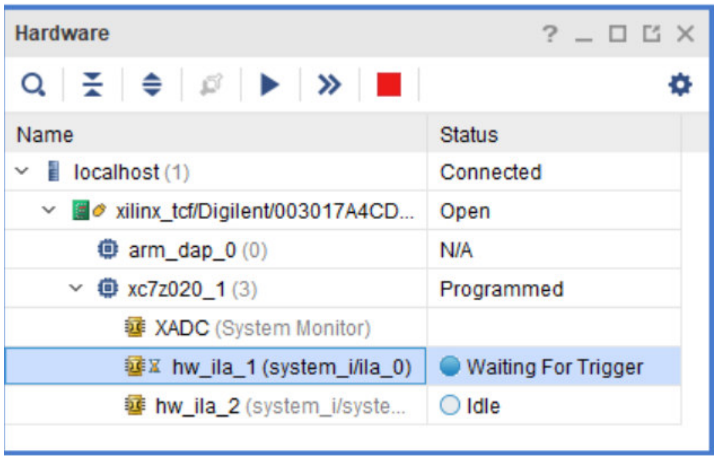


Name	Operator	Radix	Value
system_i/leds_gpio_io_o[3:0]	==	[H]	2

Figure 13: hw_ila_1 trigger settings

12. Set the trigger position of the ***hw_ila_1*** to **512** in the ***Settings – hw_ila_1*** tab.

13. Select **hw_ila_1** in the ***Hardware*** window and click on the **Run Trigger** button and observe that the ***hw_ila_1*** core is armed and showing the status as **Waiting For Trigger**. See Figure 14.

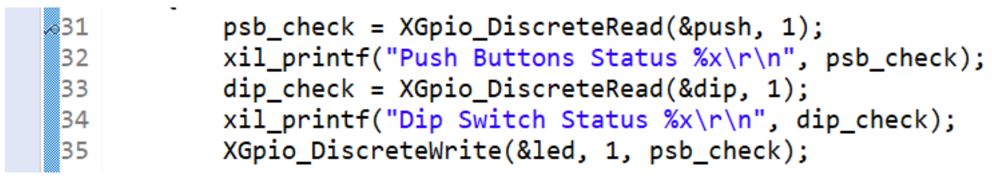


Name	Status
localhost (1)	Connected
xilinx_tcf/Digilent/003017A4CD...	Open
arm_dap_0 (0)	N/A
xc7z020_1 (3)	Programmed
XADC (System Monitor)	
hw_ila_1 (system_i/ila_0)	Waiting For Trigger
hw_ila_2 (system_i/syste...	Idle

Figure 14: Hardware analyzer running and in capture mode

14. Switch to SDK.

15. Near line 31 (right click in the margin and select ***Show Line Numbers*** if necessary*)*, double click on the left border on line 31 to set a breakpoint (Figure 14).



```
31          psb_check = XGpio_DiscreteRead(&push, 1);
32          xil_printf("Push Buttons Status %x\r\n", psb_check);
33          dip_check = XGpio_DiscreteRead(&dip, 1);
34          xil_printf("Dip Switch Status %x\r\n", dip_check);
35          XGpio_DiscreteWrite(&led, 1, psb_check);
```

Figure 14: Setting a breakpoint

16. Click on the **Resume** ( ) button to execute the program and stop at the breakpoint.

17. Now keep BTN0 pressed and click on Resume ( ) button. You will see that ***psb_check*** has a value of 1 on the terminal window. In Vivado program, ILA core does not trigger as it is waiting for a value of 2 on **leds**.

18. Switch to SDK. Now keep BTN1 pressed and click on the **Resume** ( ) button to execute the program till it encounters the break point again.

19. The ILA core will trigger this time since a value of 2 is written to the **leds**. A snapshot of the activity on **leds** is captured and displayed in the simulation window as shown in Figure 15.

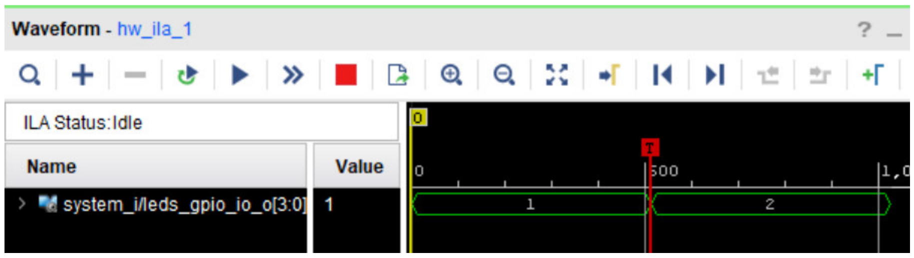


Figure 15: Zoomed view of the hw_ila_1 waveform window

20. Click the Disconnect button in the SDK to terminate the execution.

21. If you want, you could go back and look for a different value on leds or do something similar for buttons. Once satisfied, yu can close SDK by selecting **File → Exit**

22. Close the hardware session by selecting **File → Close Hardware Manager**. Click **OK.**

23. Close Vivado program by selecting **File → Exit**

24. Turn OFF the power on the board.

Conclusion

In this lab, you added an ILA core by using the 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.

Laboratory Deliverables

None

You need to demonstrate your output to your instructor.