

*CE 435*  
*Embedded Systems*

Spring 2018

**Lab 3**

*Adding Custom IP to the SoC*  
*Hardware Debug*

## Introduction

The first part of this lab guides you through the process of creating and adding a custom peripheral to the embedded system. You will write part of the Verilog code to create the hardware for the peripheral and you will connect it to the AXI4 bus. You will also update the application code to communicate to this peripheral, and download and test on the Zedboard. The application software reads the 8 switches of the Zedboard and turns on/off the corresponding LEDs. You will verify that the design operates as expected on the Zedboard.

The second step introduces you to the software and hardware debug, a powerful tool to control and monitor internal signals of an FPGA in conjunction with a running program in the ARM Cortex-A9 CPU. You will execute the program of the first step in the ARM CPU and, at the same time, monitor the activity of the hardware in the PL section.

The third step combines all the work that you did on the previous labs. You will add the Gray Code hardware IP as an AXI4 peripheral and add code to control/monitor its functionality on the Zedboard.

## Step 1 – Adding Custom IP

In this lab, you will use the Create and Import Peripheral Wizard of Xilinx Platform Studio (XPS) to create a user peripheral from an HDL module, add an instance of the imported peripheral, and provide an interface to the on-board LED module. The peripheral is a simple GPIO module.

This lab comprises several steps involving the creation of an AXI4 custom IP (a simple 5-bit output to drive LEDs) and its addition to the system. Although the change to the hardware is simple, this lab illustrates the integration of a user peripheral through the *Create and Import Peripheral Wizard*.

### Create the new hardware IP

Create a *lab3\_customIP* folder (use *mkdir*) and copy the contents of the *lab2\_simple* folder into the *lab3\_customIP* folder (using *Save As* from within *lab2\_simple* project) if you wish to continue with the design you created in the previous lab.

In the new project invoke the *Tools* → *Create and Package IP*. The wizard will take you through a pre-defined set of steps to configure the hardware and software drivers of your new peripheral and to generate template HDL files that you can use to describe the functionality of your peripheral. Click *Next* to continue and select *Create new AXI4 peripheral* and click *Next* again. In the *Peripheral Details* panel, change the following settings (indicative):

- Name: *led\_ip*
- Description: *My new LED IP*

Click *Next* to go to the *Add Interfaces* panel. Change the name of the interface to *S\_AXI* and make sure that the Interface Type is *Lite* (use AXI4 Lite protocol), the peripheral is a *AXI4 Slave*, the width of the Data Bus of the AXI4 Lite is 32 bits, and there are 4 registers in the peripherals (the registers are memory-mapped and can be accessed by the ARM CPU to control and monitor the functionality of the peripheral). Click *Next*, select *ADD IP to repository*, and click *Finish*.

Click on *IP Catalog* tab in *Flow Navigator* and search for “led”. The *led\_ip\_v1.0* peripheral has appeared the IP repository to show that the peripheral which you just added becomes part of the available cores list. Moreover, *Vivado* has generated a new *ip\_repo* directory which contains all your own IP, a *led\_op\_1.0* subdirectory, and a number of other subdirectories which include HDL files, and C source code and header files for the drivers. These are template files to help the designer integrate the functionality of the peripheral to the rest of the system. For example, `LED_IP_Reg_SelfTest(void *baseaddr)`, is a generic Read/Write test to the new peripheral.

Most of the work for designing a new peripheral involves filling up the details for these template files to define both the functionality (HDL code), as well as the software drivers that the peripheral should offer to the CPU programmer. This last point is very important: as designers of a new hardware component, you should provide not only the fully verified hardware, but also a set of driver functions to control and monitor the functionality of the component from a CPU program. These drivers abstract out the complexity of the hardware functionality and the interface and should be cover all different capabilities of a hardware component.

Right-click on the *led\_ip\_v1.0* peripheral and select *Edit in IP Packager*. This will create a **new** Vivado project, named *led\_ip\_v1.0\_project*, where the functionality of the peripheral can be modified in the HDL code and then packaged for future use. In the *Design Sources*, you can see two Verilog source files (remember, we designated Verilog as our target language, so all template HDL files are automatically generated in Verilog).

The following two files have been generated (organized as shown in the Figure at the end of the page)

- *led\_ip\_v1\_0.v* is a wrapper file.
- *led\_ip\_v1\_0\_S00\_AXI.v* file. This template file contains the AXI4-Lite interface functionality which handles the interaction between the peripheral and the AXI4-Lite bus. The user can include code to define the functionality of the peripheral. It is a very good idea to study the Verilog code to understand the functionality of the AXI4-Lite bus and its interface to a standard peripheral.

We will add a new verilog file (e.g. *led\_user\_logic.v*) which will describe the functionality of the GPIO peripheral called LED and we will integrate it to the *led\_ip\_v1\_0\_S00\_AXI.v* file. First, open the wrapper file *led\_ip\_v1\_0.v* and add a new output in the designated place (around line 18):

```
// Users to add ports here
output wire [7:0] LED,
// User ports end
```

This port should also be added to the instantiation of module *led\_ip\_v1\_0\_S\_AXI* (around line 50).

Then, scroll down to the end of the file and instantiate the *led\_user\_logic* module with the name *U1*:

// Users user logic here

```
led_user_logic # (  
    .C_S_AXI_ADDR_WIDTH(C_S_AXI_ADDR_WIDTH), // parameters  
    .ADDR_LSB(ADDR_LSB)  
) U1 (  
    .S_AXI_ACLK(S_AXI_ACLK),  
    .S_AXI_ARESETN(S_AXI_ARESETN),  
    .slv_reg_wren(slv_reg_wren),  
    .axi_awaddr(axi_awaddr[C_S_AXI_ADDR_WIDTH-1 : ADDR_LSB]),  
    .S_AXI_WDATA(S_AXI_WDATA),  
    .LED(LED)  
);
```

// User logic ends

We still need to write the code for the *led\_user\_logic.v*. Click on *Add Sources* and follow the procedure we described in lab1 to add the Verilog file shown below. As you see, the main functionality of this peripheral is to pass the data from the Write Data Bus to the LED output.

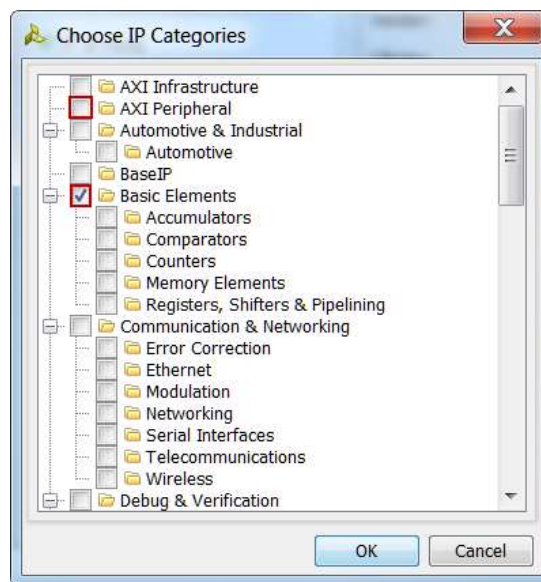
```
`timescale 1ns / 1ps  
/////////////////////////////////////  
// Module Name: led_user_logic  
/////////////////////////////////////  
module led_user_logic(  
    input S_AXI_ACLK,  
    input S_AXI_ARESETN,  
    input slv_reg_wren,  
    input [C_S_AXI_ADDR_WIDTH-1:0] axi_awaddr,  
    input [31:0] S_AXI_WDATA,  
    output reg [7:0] LED  
);  
  
always @( posedge S_AXI_ACLK )  
begin  
    if ( S_AXI_ARESETN == 1'b1 )  
        LED <= 7'b0;  
    else  
        if ((slv_reg_wren == 1) && (axi_awaddr == 0))  
            LED <= S_AXI_WDATA[7:0];  
    end  
endmodule
```

Click *Run Synthesis* and *Save* if prompted. Make sure that there are no synthesis errors by checking the Messages Tab.

## Package the new hardware IP

For the *led\_ip* to appear in the IP catalog and be reused in various designs, it has to follow standardized specifications (IP-XACT) which are industry-wide. The IP-XACT standard is an eXtensible Markup Language (XML) schema for documenting IP using metadata, which is both human readable and machine accessible, along with an Application Programming Interface (API) which allows software tools access to the stored meta-data. IP-XACT does not describe the hardware functionality of IP (this is your job), but instead describes the interface. It is therefore not a replacement for RTL languages such as VHDL or Verilog, embedded software or documentation, but is instead complementary: an IP-XACT component provides key information about the IP.

Start by clicking *Package IP* in the *Project Manager* panel to open up the *IP Packaging Steps* panel. In *IP Identification* panel, go to *Categories*, check the *Basic Elements* category, uncheck the *AXI Peripheral* category and click *OK* as shown in the figure.



Next, select IP Compatibility. This shows the different Xilinx FPGA Families that the IP supports. The value is inherited from the device selected for the project.

You can also customize the the address space using the using the *IP Addressing and Memory* category. We won't make any changes.

Click on *IP Ports and Interfaces*. Notice that the ports of the our new IP do not contain the LED output port. To fix that, click *Merge changes from IP File Groups Wizard*. This is to update the IP Packager with the changes that were made to the IP and the *led\_user\_logic.v* file that was added to the project.

After you take a look at all steps, select *Review and Package* and click *Re-Package IP*. This will close the *led\_ip\_v1.0\_project* project in Vivado.


## Modify the Project settings

Open the *lab3\_custom* project and click *Project Settings* in the *Flow Navigator* panel. Select *IP* in the left panel of the *Project Settings*. Click on the *Add Repository...* button, browse to the *led\_ip\_1.0* and click *Select*. The *led\_ip\_v1.0* IP will appear in the *Selected Repository* window at the bottom. Click *OK*.


This set of actions adds directories to the list of repositories (in this case the *ip\_repo*). We can then add later additional IP to the selected repository.

## Use the new IP in the lab3 design

We will now add *led\_ip* to the *lab3\_custom* design and connect to the AXI4-Lite interconnect in the IP Integrator. Then, we will make internal and external port connections and establish the LED port as external FPGA pins.

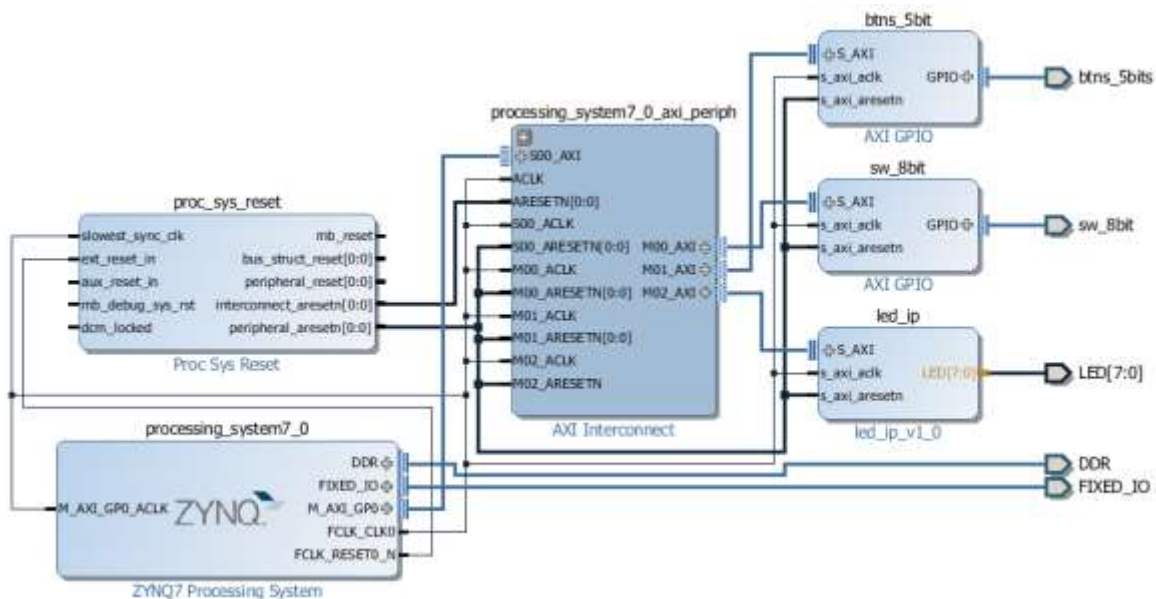
Click *Open Block Design* under *IP Integrator* in the *Flow Navigator* panel, and select *system.bd* to open IP Integrator. Click the *Add IP* icon  and search for *led\_ip\_v1\_0* in the catalog by typing “led” in the search field. Double-click *led\_ip\_v1\_0* to add the core to the design. Select the IP in the block diagram and change the instance name to *led\_ip* in the properties view

Click on *Run Connection Automation*, select */led\_ip/S\_AXI* and click *OK* to automatically make the connection from the AXI Interconnect to the IP.

Select the *LED[7:0]* port on the *led\_ip* instance (by clicking on its pin), right-click and select *Make External*. Click the regenerate button (  ) to redraw the diagram which should be similar to the one below:

Select the **Address Editor** tab and verify that an address has been assigned to *led\_ip*. The standard address is *0x43C00000* to *0x43C0FFFF*. Validate your new design using *Tools* → *Validate Design*.

The next step is to update the wrapper file: In the *sources* → *IP sources* view, right-click on the block diagram file, *system.bd*, and select *Create HDL Wrapper* to update the HDL wrapper file. When prompted, select *Let Vivado Manage Wrapper and auto-update* and click *OK*. The *system\_wrapper.v* file will be updated to include the new IP and ports. Check the *system.v* file to verify that LED port has been added.



The next step is to add Constraints before we implement the design. This is something you can do on your own by looking at the pin assignments of the Zedboard. You only need to assign the extra LED IP that you have created:

```
set_property PACKAGE_PIN T22 [get_ports LED[0]]  
set_property IOSTANDARD LVCMOS33 [get_ports LED[0]]
```

Continue here...

Click on the *Generate Bitstream* in the Flow Navigator to run the synthesis, implementation, and bitstream generation processes. (Click *Save* if prompted.). Click *Yes* to run the synthesis process again as the design has changed. When the build completes, click *OK* if prompted to open the Implemented design.

### Create the new Software IP and test your design on the Zedboard

Export your hardware along with the generated bitstream and Open *SDK* as we have described in lab2. The *system\_wrapper\_hw\_platform\_0* contains the *system.hdf* platform description including the *led\_ip* with memory ranges *[0x43C00000, 0x43C0FFFF]* (or whatever you have changed it to). Create a new empty application (e.g. *lab3\_custom*) with the corresponding BSP package.

You can use the provided *lab3\_custom.c* file as a template and complete the code to output the readings of the 8 switches to the corresponding LEDs. Use the header files provided (*xparameters.h* and *led\_ip.h* contain particularly useful driver functions) to complete the code. When you are done, download the bitstream to the FPGA, open a minicom session and Run your code on the Zedboard.

## Step 2 – Debugging in Software and Hardware

Software and hardware interacts 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 allows for hardware debugging by providing access to the internal signals without necessarily bringing them out via the package pins using several types of cores. These powerful cores reside in the programmable logic (PL) portion of the device and can be configured with several modes that monitor signals within the design. Vivado provides capabilities to mark any net at several stages of the design flow and can combine both software and hardware co-debug sessions.

### New hardware platform


Save the *lab3\_custom* project as *lab3\_HwDebug*. We will enhance the new platform with a number of hardware components to assist us with hardware debug:

- ILA (Integrated Logic Analyzer) is a customizable IP core used to monitor the internal signals of a design. The ILA core includes many advanced features of modern logic analyzers, including Boolean trigger equations, and edge transition triggers. Because the ILA core is synchronous to the design being monitored, all design clock constraints that are applied to your design are also applied to the components inside the ILA core.




- Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real time.


We will now add the ILA core and connect it to the LED output port to monitor the output of the LED component.

Click the Add IP icon  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. Double-click on the *ila\_0* instance, select *Native* tab (not AXI) and select the Probe Ports tab. Set the *Probe Width* of *PROBE0* to 8 and click OK.

Using the drawing tool, connect the *PROBE0* port of the *ila\_0* instance to the LED port of the *led\_ip* instance. Connect the *CLK* port of the *ila\_0* instance to the *S\_AXI\_ACLK* port of one of the other instances.

We will now add a second ILA core and connect it to the input AXI interface of the LED core. Select the *S\_AXI* connection between the AXI Interconnect and the *led\_ip* instance. Right-click and select *Mark Debug*. This much simpler operation will implicitly create a second ILA component that monitors the AXI transactions to the LED core.

The final step is to add the VIO core and connect it to the system. Before we add the VIO core, we will add a math core that will be controlled and monitored by the VIO. Click the Add IP icon  and search for “add” in the catalog. Double-click on the *addsub* IP core to add an instance of it. Double-click on the adder/subtractor instance, set the output width to 16 bits, and the two input width to 15 bits. Include also a *CLK* as an input to the component.

The math component will only be controlled by the VIO; it does not have an output to the board, and it does not receive any input from anywhere else in the platform. Once more, click the Add IP icon  and search for *vio* in the catalog. Double-click on the VIO (Virtual Input/Output) to add an instance of it. The *vio\_0* instance will be added. Double-click on the *vio\_0* instance to open the configuration form. Set the *Output Probe Count* to 3 and the *Input Probe Count* to 1 in the *General Options* tab. Select the *PROBE\_IN* Ports tab and set the *PROBE\_IN0* width to 16. Select the *PROBE\_OUT* Ports tab and set *PROBE\_OUT0* width to 1, *PROBE\_OUT1* width to 15, and *PROBE\_OUT2* width to 15. Click OK.

Connect the VIO instance’s ports to the *addsub* instance ports as follows:

- *PROBE\_IN0* -> output *S* of adder/subtractor
- *PROBE\_OUT0* -> port *ADD*
- *PROBE\_OUT1* -> port *A*
- *PROBE\_OUT2* -> port *B*
- Connect the *CLK* port of the *vio\_1* to *S\_AXI\_ACLK* port of one of the other instances.

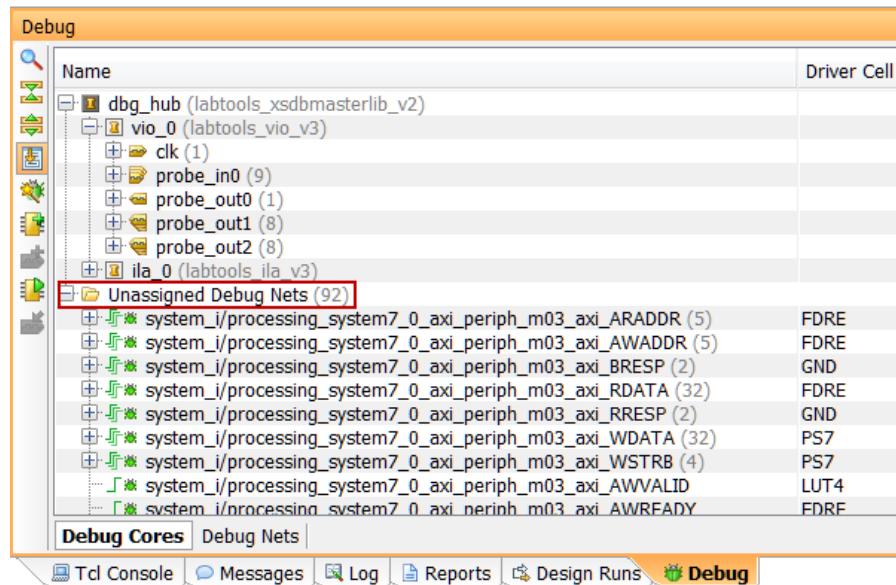
Select *Tools* → *Validate Design* to run the design rules checker. Verify that there are no unmapped addresses shown in the *Address Editor* tab. Click *OK*.

Select the *Sources* > *Hierarchy* tab. Expand the *Design Sources*, right-click the *system.bd* and select *Create HDL Wrapper*.

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



We now assign nets for debugging: The synthesized design will be opened in the *Auxiliary panel* and the *Debug* tab will be opened in the *Console* panel. If the *Debug* tab is not open then select *Window* → *Debug*. Notice that the nets which can be debugged are grouped into *Assigned* and *Unassigned* groups. The assigned net groups include nets associated with the VIO and ILA cores, whereas the unassigned nets group includes S\_AXI related nets.



Right-click on the *Unassigned Debug Nets* and select the *Set up Debug...* option. Click *Next*. The nets are listed. Right click on the *BRESP* and *RRESP* (which are driven by GND) and select *Remove Nets*. Click *Next* twice, and then *Finish*.

Click on the *Generate Bitstream* to run the implementation and bit generation processes. Click *Save* to save the constraint file, and *Yes* to run the processes.

## Software development

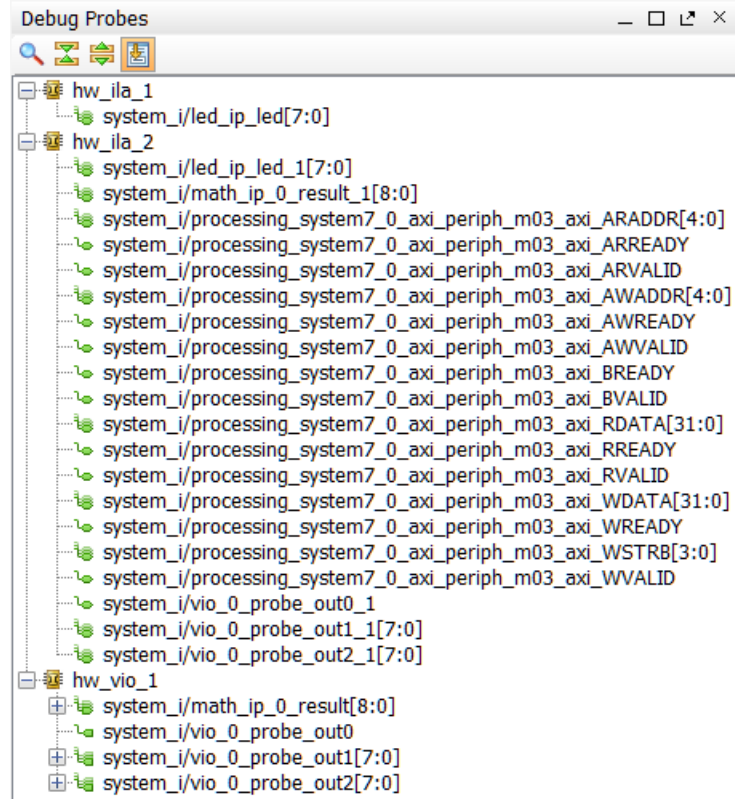
Export your hardware along with the generated bitstream. Open *SDK*. The *system\_wrapper\_hw\_platform* contains the *system.hdf* platform description including the new *ILA* (2), *VIO* and *addsub* components. Create a new empty application (e.g. *lab3\_HwDebug*) with the corresponding BSP package. Import the *lab3\_custom.c* file of the first step (rename it first to *lab3\_HwDebug.c*). This code will be debugged at the same time as it triggers transactions in the Vivado Hardware Analyzer.

## Hardware/Software Debugging

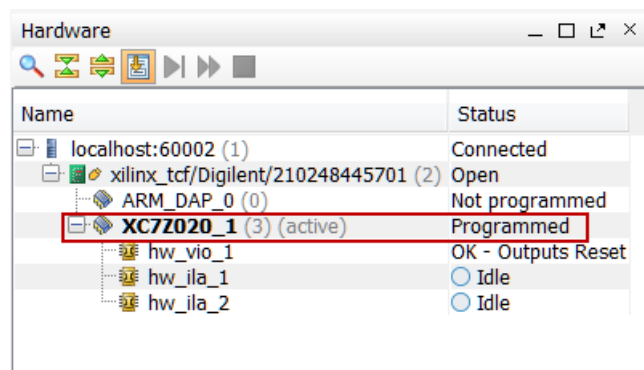
We will have designed and implemented the hardware and software platform required for debug. So let us try it. Connect and power up the ZedBoard. Download the bitstream into the target device *XilinxTools* → *Program FPGA*.

Select the *lab3\_HwDebug* project in *Project Explorer*, right-click and select *Debug As* → *Launch on Hardware (GDB)* to download the application, execute *ps7\_init*, and display a dialog box asking to switch the perspective to the *Debug* perspective. Click *Yes* and the perspective will change. The program execution starts and suspends at the entry point (beginning of *main*).

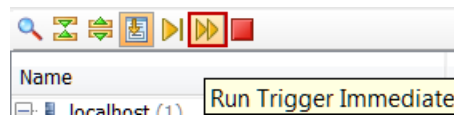
**The next step is to start a Hardware session.** Switch to Vivado. Click on *Open Hardware Manager* from the *Program and Debug* group of the *Flow Navigator* panel to invoke the analyzer. Click on the *Open a New Hardware Target* link to establish the connection with the board. Click *Next* twice to use the default settings. The JTAG chain will be scanned and the device will be detected. Click *Next* twice and then *Finish*. The hardware session will open showing the *Debug Probes* tab as shown in the Figure.



The hardware session status window also opens showing that the FPGA is programmed (we did it in SDK), there are three cores (2 *ilas* and one *vio*). The two *ila* cores are in the idle state. These components will be used separately in our experiments of *lab3*.



Select *XC7Z020\_1*, and click on the *Run Trigger Immediate* button to see the signals in the waveform window.

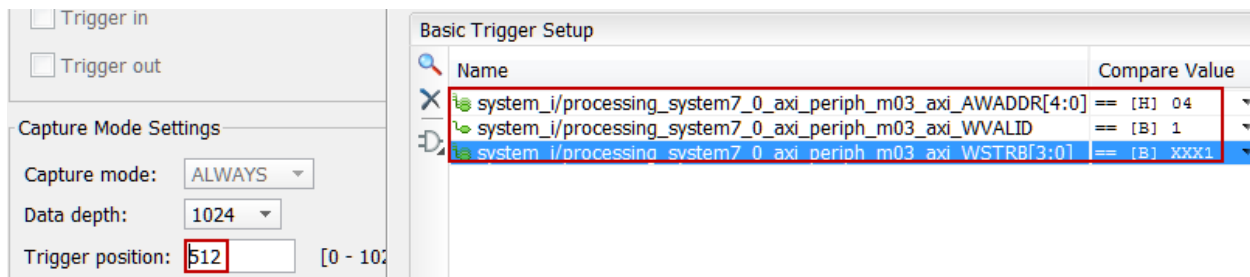



Two waveform windows will be created, one for each *ila*; one *ila* (*hw\_ila\_1*) is of the instantiated ILA core and another (*hw\_ila\_2*) for the *MARK\_DEBUG* method (captures the AXI bus).


Once the hardware session has opened, **the next step is to set up trigger conditions**. These are boolean conditions on the I/O signals of *ilas*, and *vios* that, when evaluated to true, cause the dumping of the values of internal FPGA signals to the waveform windows. This way you can monitor the signal activity of components in the real hardware.

**Our first experiment** concerns the *hw\_ila\_2* and triggers on values of the AXI bus. In the *Debug Probes* window, select the *AWADDR* bus and drag it into the *ILA-hw\_ila\_2* window and release to add it to set a trigger condition on it. Change the value from *xx* to *00*. Similarly, add the *WSTRB* and *WVALID* signals, and change the condition from *xxxx* to *xxx1* and to *1*. When all these three AXI signals take the trigger values in the actual hardware the values of all internal signals of the *hw\_ila\_2* will be dumped to the waveforms.

Set the trigger position of the *hw\_ila\_2* to 512. Since the *Data Depth* is 1024, the value of 512 means that the Trigger cursor will appear exactly at in the middle of the waveforms.

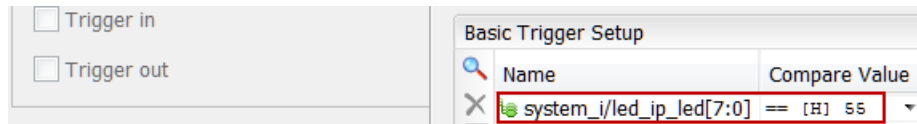


Select *hw\_ila\_2* in the *Hardware* panel. Click on the *Run Trigger* () button and observe that the *hw\_ila\_2* core is armed and showing the status as *Waiting For Trigger*. This means that the *ila* core is waiting for these three conditions to become simultaneously TRUE in order to dump the AXI values to the waveforms. But for that to happen, we need to resume the CPU code and exercise the values of the AXI bus.

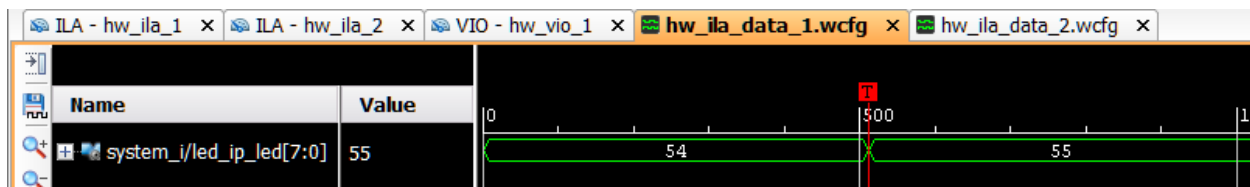
Switch back to SDK. Click on the *Resume* () button to execute the program. In the Vivado program, notice that the *hw\_ila\_2* status changed from *capturing* to *Idle*, and the waveform window shows the triggered output. Move the cursor to closer to the trigger point and then

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.

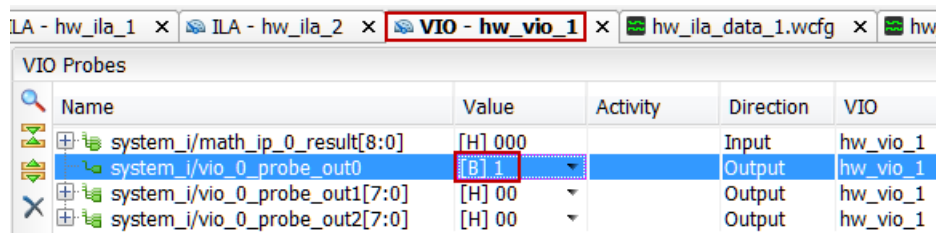
**Our second experiment** concerns *hw\_ila\_1* core which tracks the output of the LED component. Setup the ILA core (*hw\_ila\_1*) trigger condition to 0101\_0101 (x55) as shown in the following Figure. Make sure that the switches on the ZedBoard are not set at x55. Set the trigger equation to be ==, verify that the trigger position for the *hw\_ila\_1* is set to 512, and arm the trigger by selecting *Run Trigger*.



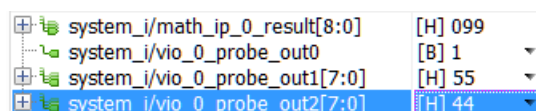
Click on the Resume button in the SDK to continue executing the program. Change the switches to 0x55 and observe that the hardware core triggers when the preset condition is met. That happens, the waveform will be displayed.




**Our third experiment** concerns the VIO functionality. Select the all the signals of the VIO Cores in the Debug Probes, drag them into the VIO-hw\_vio\_1 window. Select *vio\_0\_probe\_out0* and change its value to 1 so the *addsub* core input performs an addition.



Click on the Value field of *vio\_0\_probe\_out1* and change the value to 55 (in Hex). Similarly, click on the Value field of *vio\_0\_probe\_out2* and change the 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). Try a few other inputs and observe the outputs.



Click on the Terminate button (  ) in the SDK to terminate the execution. Close the SDK by selecting *File* → *Exit*. In Vivado, close the hardware session by selecting *File* → *Close Hardware Manager*. Click *OK*. Close Vivado program by selecting *File* → *Exit*. Click *OK*. Turn OFF the power on the board.

## Step 3 – Adding Gray Counter as an IP

In step 3 you will add the Gray Code hardware of lab1 as an AXI4-Lite peripheral and you will write some simple drivers and application code to control and monitor its functionality. You will use two push buttons on Zedboard for control and you will use the LEDs to monitor the functionality of your design.

The specifications of the functionality are shown in the following Table. A Button is 1 when pressed, and 0 when unpressed. You can use the switches instead of the buttons.

Button0	Button1	Description
0	0	Continue with what you were doing
0	1	Freeze without reset.
1	0	Start Counting in Gray Code from the current value
1	1	Reset Gray counter to 0 and freeze.

As a final task, you should print out the 8-bit value of the Gray counter each time the counter changes its value. For example, when the LEDs values are equal to

ON OFF ON ON OFF ON ON ON

the value 0xB8 appears in the minicom. Think of the hardware and software changes needed to achieve this functionality.