LABORATORY EXERCISE 3
EDK Introduction

Prerequisites

1. previous labs

2. Basic knowledge of digital design

What will be learned?

After completing this exercise, students will be able to Design a embedded processor system in EDK and to create the custom peripheral.

The Embedded Development Kit (EDK)

Embedded systems are complex. Getting the hardware and software portions of an embedded design to work are projects in themselves. Merging the two design components so they function as one system creates additional challenges. Add an FPGA design project to the mix, and the situation has the potential to become very complicated indeed. To simplify the design process, Xilinx offers several sets of tools.

The Xilinx® Embedded Development Kit (EDK) is a suite of tools and Intellectual Property (IP) that enables you to design a complete embedded processor system for implementation in a Xilinx Field Programmable Gate Array (FPGA) device. It consist of:

1. Xilinx Platform Studio (XPS) is the development environment used for designing the hardware portion of your embedded processor system. You can run XPS in batch mode or using the GUI, which is what we will be demonstrating in this guide.
2. The Software Development Kit (SDK) is an integrated development environment, complementary to XPS, that is used for C/C++ embedded software application creation and verification. SDK is built on the Eclipse open-source framework and might appear familiar to you or members of your design team.
3. Other EDK components include:
    - Hardware IP for the Xilinx embedded processors
    - Drivers and libraries for the embedded software development
    - GNU compiler and debugger for C/C++ software development targeting the
    - MicroBlaze™ and PowerPC® processors
    - Documentation
    - Sample projects

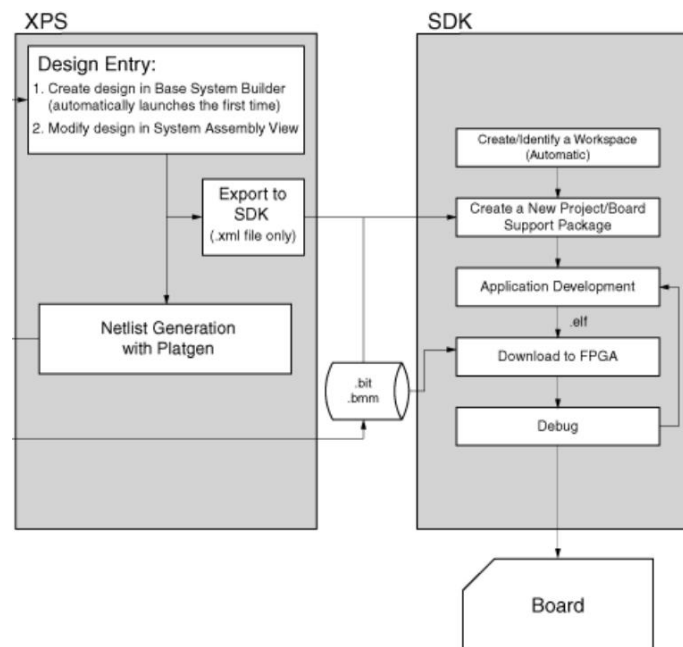EDK is designed to assist in all phases of the embedded design process;

Figure 1: Basic Embedded Design Process Flow

Hardware Construction with the Base System Builder

This lab guides you through the process of using the Xilinx Platform Studio (XPS) tools (included in the ISE® Design Suite) to create a simple processor system. 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.

The purpose of the lab exercises is to complete the process of implementing an embedded system using a XPS embedded system. The processor system will be the top-level design structure within the XPS tool project.

This lab comprises three primary steps: You will create a project using the Base System Builder; analyze the created project; and, finally, generate the processor and hardware IP netlists. A bitstream will also be generated. The software component of the design will be covered in the second lab part.

The following diagram represents the completed design Figure 2 which will be realized during this tutorial.
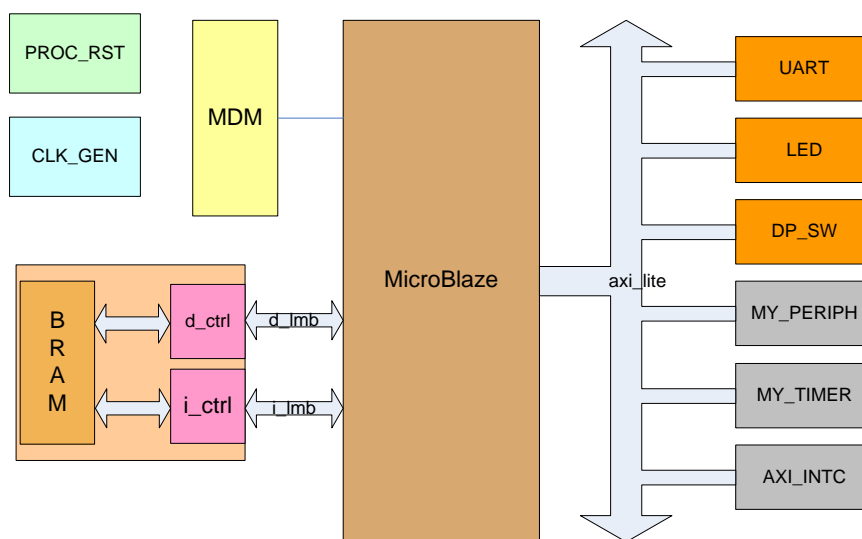

Figure 2:Simple XPS design

Creating the Project Using Base System Builder

Begin the lab by opening a new XPS software project and creating the embedded project using Base System Builder (BSB).

1. Launch the XPS tool, and pick **Create New Project Using Base System Builder** Figure 3.
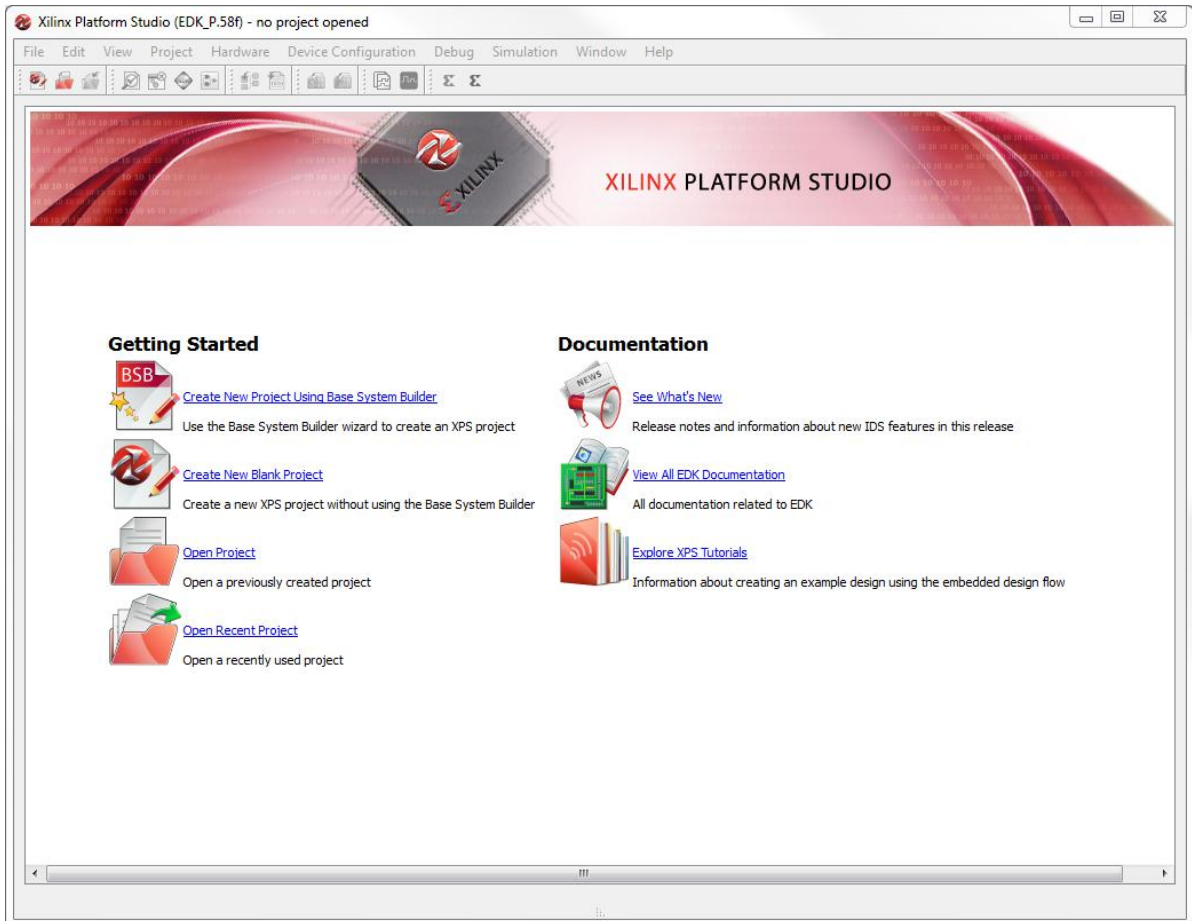

Figure 3: Xilinx Platform Studio

2. You will be asked to specify which folder to place the project. Click **Browse** and create a new folder for the project (Figure 4). Click **OK**.

3. The first page of the wizard will ask us to choose between using a Processor Local Bus (PLB) or an Advanced Extensible Interface (AXI). Pick the AXI system, and others fields left blank. Click **OK**.
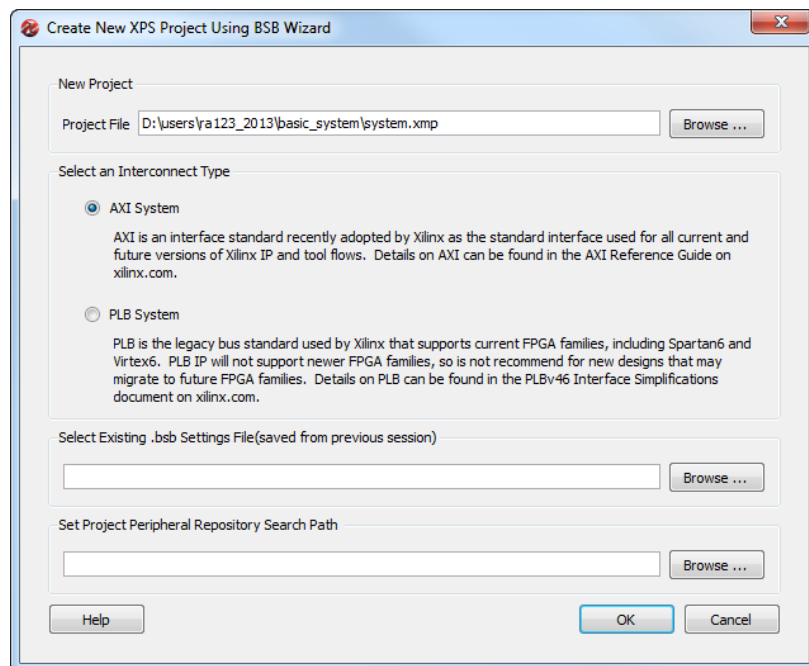
Figure 4: BSP Wizard, Interconnection Type Selection

4. On the Select Board page, select Create a System for Custom Board. Then fill fields with platform details (Figure 5). Click Next.

- *Architecture*:   **Spartan6**
- *Device*:   **XC6SLX45**
- *Reference Clock Freq*:   **50.00 MHz**
- *Package*:   **FGG676**
- *Speed*:   **-2**
- *Reset Polarity*:   **Active Low**
- *Select a System*:   **Single MicroBlaze Processor System**
- *Optimization Strategy*:   **Throughput**

Figure 5: BSB Wizard, Create System for Custom Board

5. On the **Processor and peripheral Configuration**(Figure 6) select:

- Local Memory Size select 32 KB.

- Because this is custom board, peripherals should be added manually. Click on Add Device.
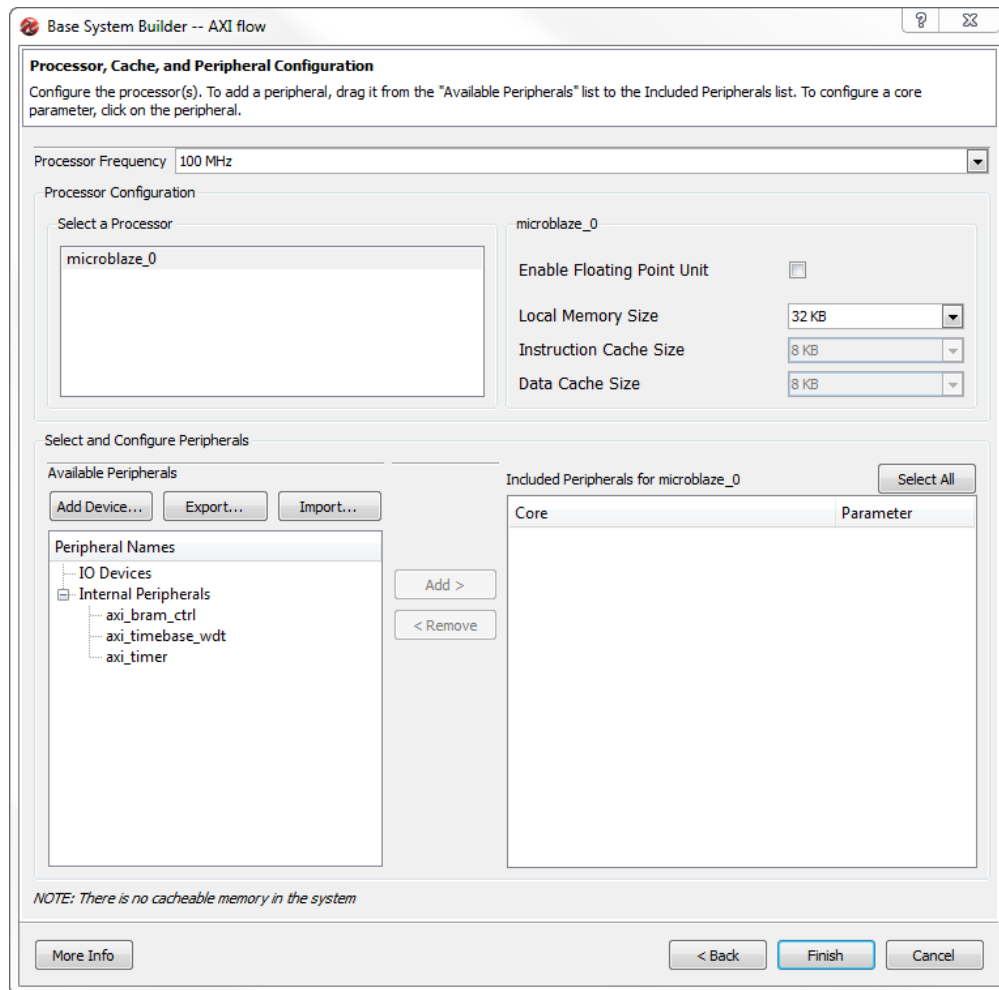
Figure 6: BSP Wizard, Select and Configure Peripherals

3. For first peripheral select **IO Interface Type** select *GPIO*. In **Device** *DIP_Swithces.* Leave filed **Add Device to the system** checked (Figure 7). Click on **OK**. In included peripherals for MicroBleze will appear DIP_Switches peripheral with **Core** name (*axi_gpio*) Default **Data Width** *8*, and unchecked filed **Use Interrupt (**Figure 8**).** Leave all defaults values.

4. Second, add also **GPIO** peripheral, but now add *LEDS* **Devices**. Leave all defaults values.

5. Then add **IO Interface Type** *UART,* **Device** *RS232.* Leave all defaults values.

6. Click **Finish**. Pop-up menu will appear with massage *Generating Project & Design Files...*

7. Finally, Figure 9 shows lunched XPS window with System Assembly View tab.

Figure 7: Wizard, Add DIP_Switches and LEDs Peripherals
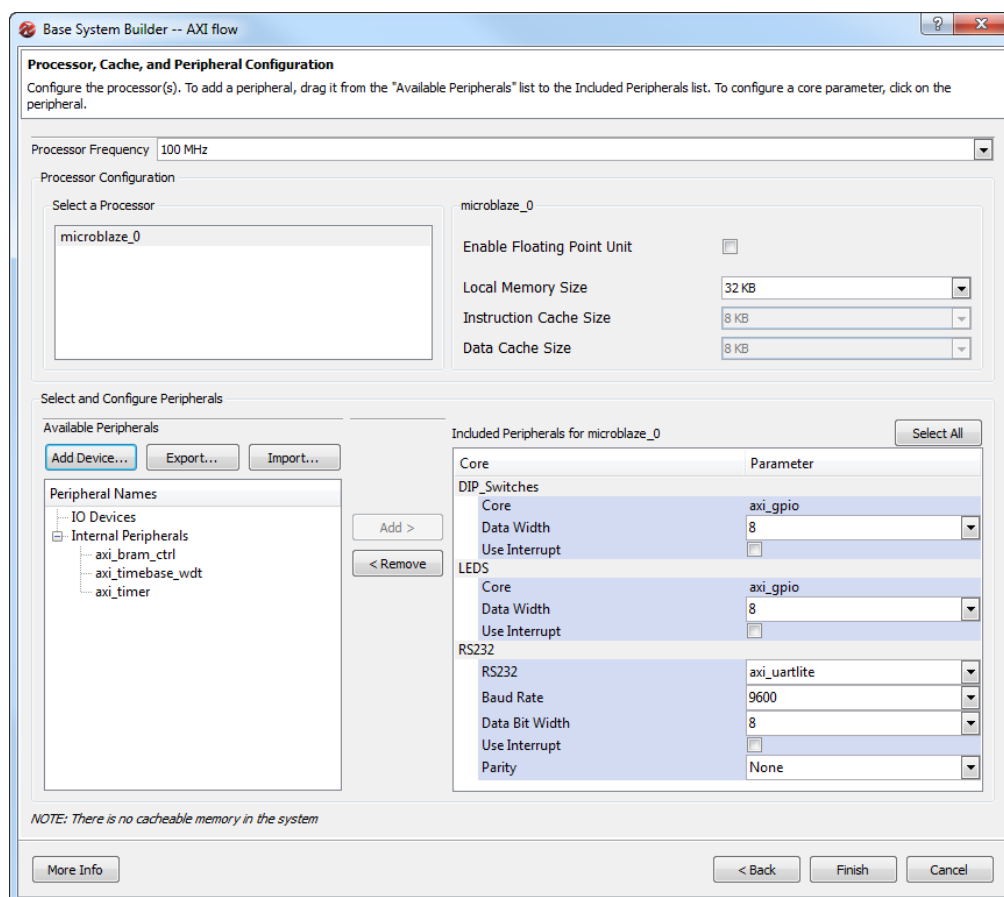
Figure 8: BSB Wizard, with Peripherals Configuration Default Values
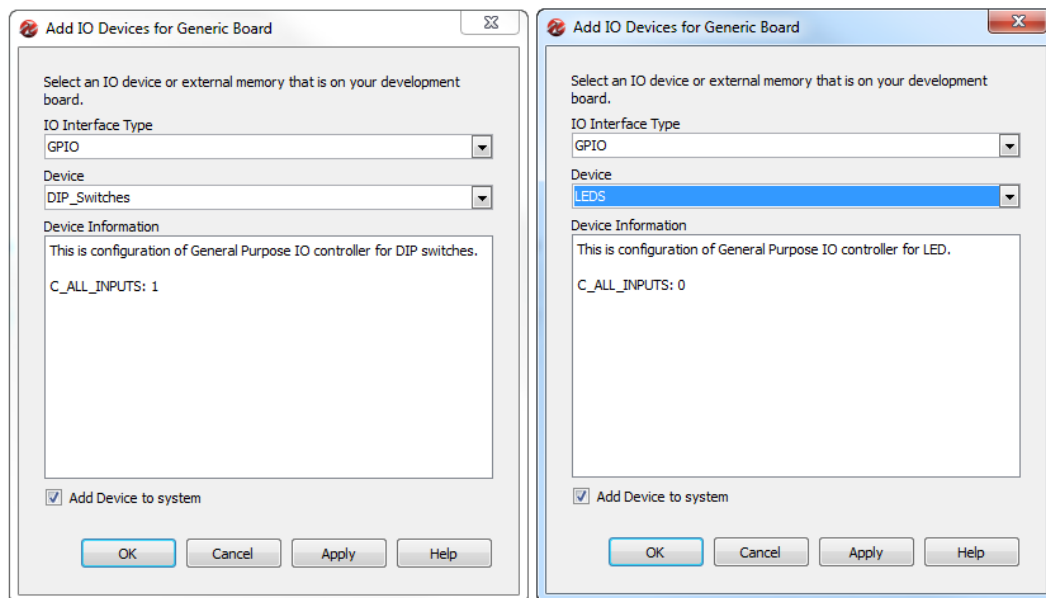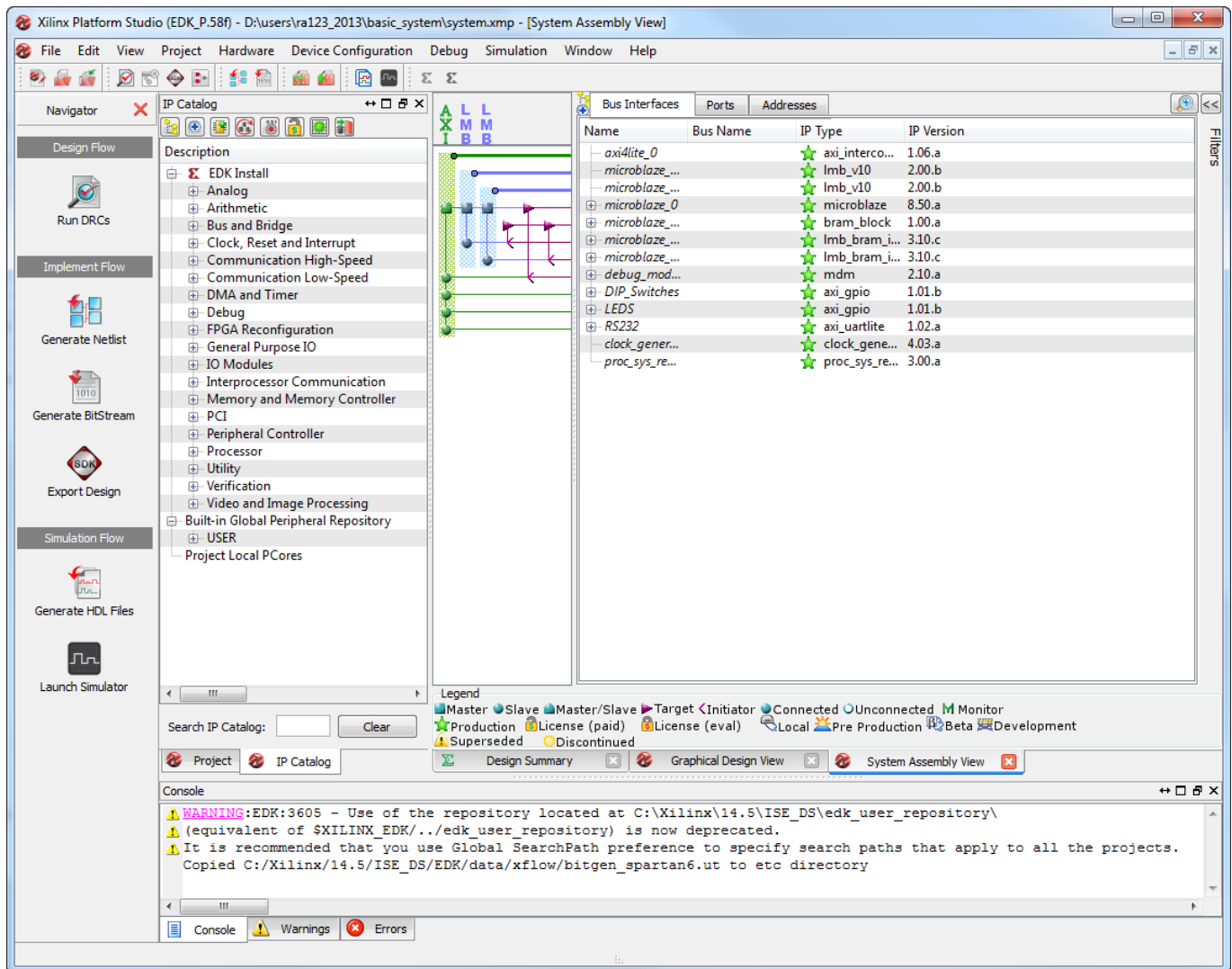
Figure 9: Launched XPS

Analyzing the Created Project

Review the information in the System Assembly View tab, which shows each peripheral used and the connections between the peripherals when the Bus Interface tab is selected.

1. Examine the hardware design - In the Graphical Design view Select the **Graphical Design View** tab and browse the various components that are used in the design by using the mouse gestures to zoom in for closer examination (Click drag up and right to zoom out; click drag down and left to zoom in. Also try click drag down right to zoom an area and click drag left up to zoom to fit). The block diagram provides a graphical schematic of the embedded processor platform. You will see the MicroBlaze processor, AXI interconnect, LMB bus, and the various peripherals (Figure 10).
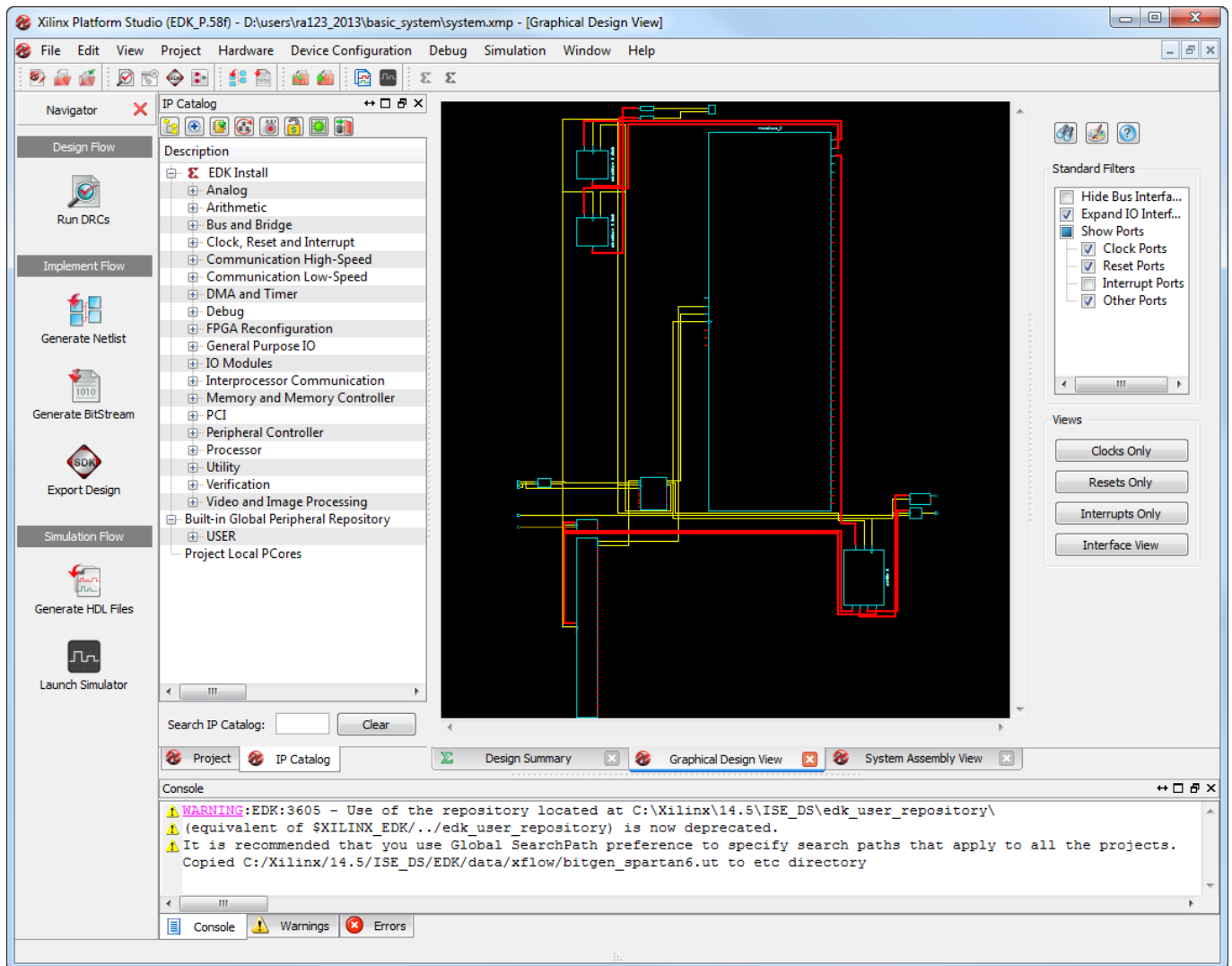
Figure 10: Graphical Design View

2. Connections between the peripherals - Select the **Ports** tab and observe the expanded view similar to the figure below. The Ports view lists the connections for signals that are not part of any bus. These are typically signals that are specific to a peripheral. The formal port name of the peripheral is on the left; the actual net name is in the Net column.

The direction column indicates the signal direction relative to the component.
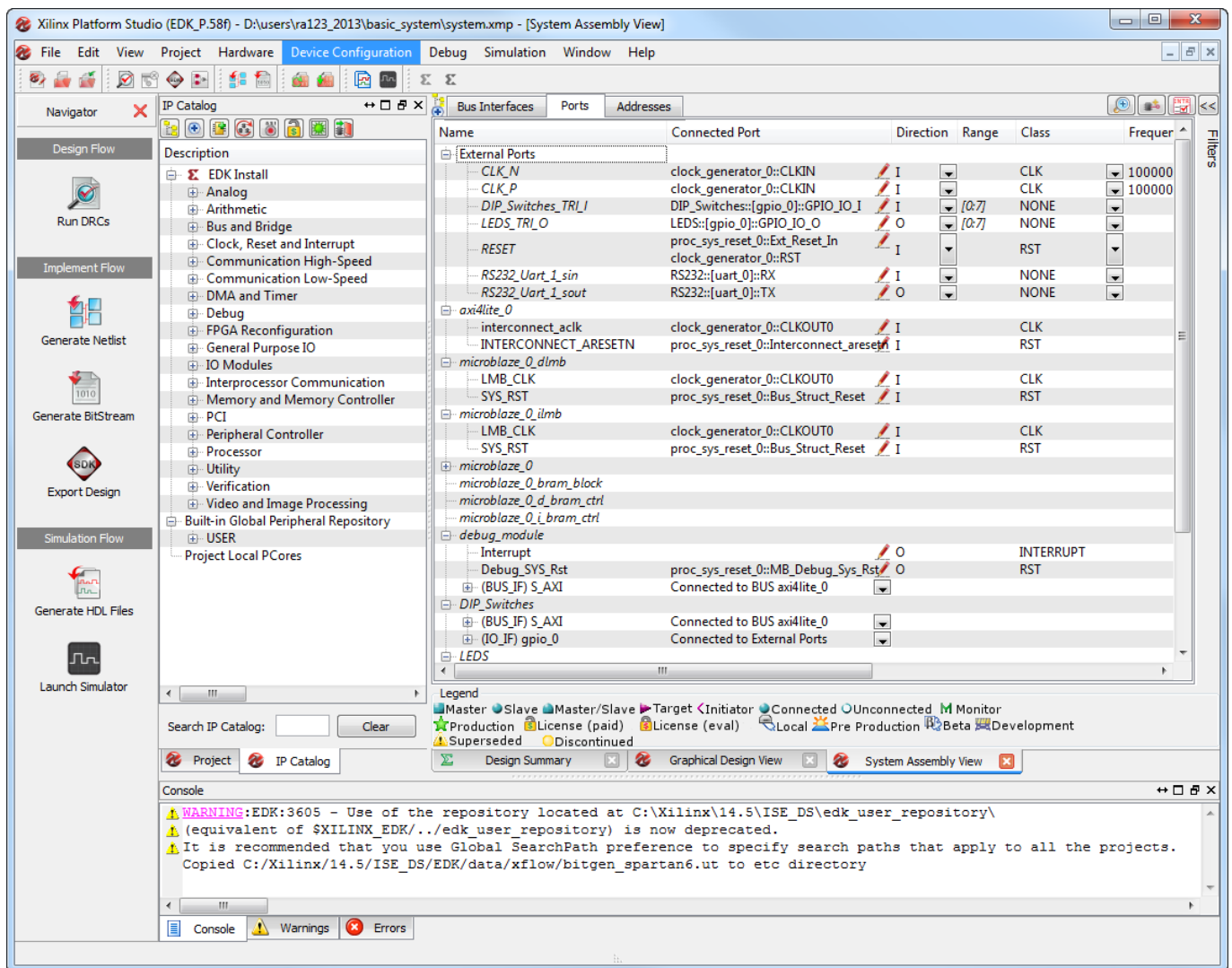
Figure 11: XPS Ports Tab

3. View the address map for this processor. Select the **Addresses** tab. Expand **microblaze_0's Address Map** to view the address map for this processor.

Figure 12: XPS Addresses Tab

Design Implementation

Before of design implementation you can run a design rule check to verify that the design has no errors, by selecting **Project**, **Design Rule Check**, or **Run DRCs** in Design Flow Navigator toolbars. View the results of the DRC in the Console window.

Because our platform E2LP is not supported with BSB we should update *.ucf file with constraints. These constraints affect how the logical design is implemented in the target device. Example of *.ucf file for our system.
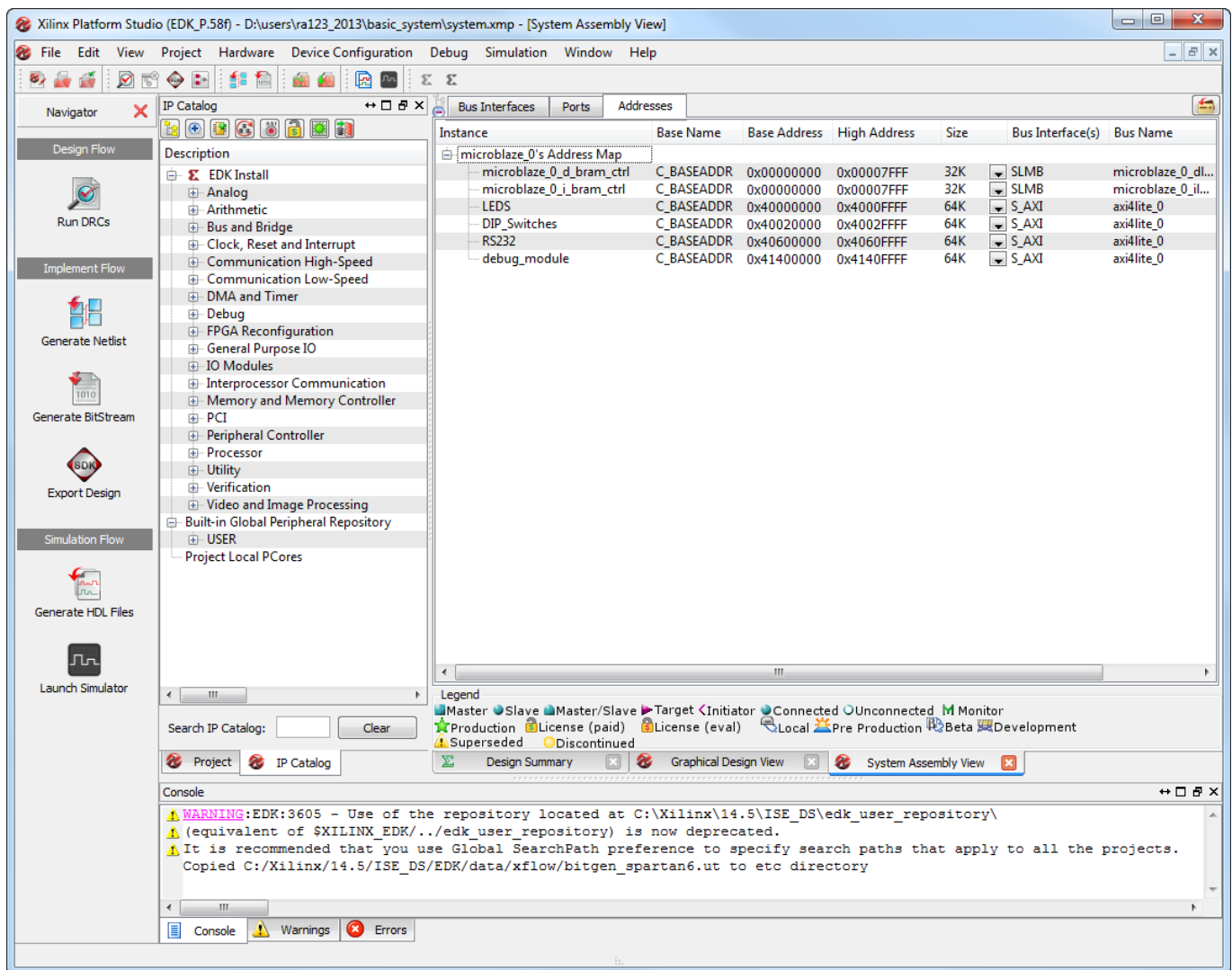
```
NET CLK_N                LOC = AF14 | IOSTANDARD = LVDS_33;
NET CLK_P                LOC = AD14 | IOSTANDARD = LVDS_33;
NET RESET                LOC = AE24 | IOSTANDARD = LVCMOS33;
NET RS232_Uart_1_sin     LOC = A21  | IOSTANDARD = LVTTL;
NET RS232_Uart_1_sout    LOC = A20  | IOSTANDARD = LVTTL;
NET DIP_Switches_TRI_I(0) LOC = W19  | IOSTANDARD = LVTTL;
NET DIP_Switches_TRI_I(1) LOC = Y24  | IOSTANDARD = LVTTL;
NET DIP_Switches_TRI_I(2) LOC = K19  | IOSTANDARD = LVTTL;
NET DIP_Switches_TRI_I(3) LOC = V24  | IOSTANDARD = LVTTL;
NET DIP_Switches_TRI_I(4) LOC = U20  | IOSTANDARD = LVTTL;
NET DIP_Switches_TRI_I(5) LOC = U23  | IOSTANDARD = LVTTL;
NET DIP_Switches_TRI_I(6) LOC = U24  | IOSTANDARD = LVTTL;
NET DIP_Switches_TRI_I(7) LOC = U19  | IOSTANDARD = LVTTL;
NET LEDS_TRI_O(0)        LOC = N24  | IOSTANDARD = LVTTL;
NET LEDS_TRI_O(1)        LOC = N23  | IOSTANDARD = LVTTL;
NET LEDS_TRI_O(2)        LOC = M24  | IOSTANDARD = LVTTL;
NET LEDS_TRI_O(3)        LOC = L24  | IOSTANDARD = LVTTL;
```

```
NET LEDS_TRI_O(4)          LOC = L23  | IOSTANDARD = LVTTL;
NET LEDS_TRI_O(5)          LOC = K24  | IOSTANDARD = LVTTL;
NET LEDS_TRI_O(6)          LOC = H24  | IOSTANDARD = LVTTL;
NET LEDS_TRI_O(7)          LOC = D24  | IOSTANDARD = LVTTL;
```

Run PlatGen by selecting **Hardware** > **Generate Netlist**. You can also find **Generate Netlist** in the Implement Flow section of the Navigator window. It will generate implementation, HDL, and synthesis directories. The generated peripherals and system netlists will be placed in the implementation directory.

Run implementation tools (xflow) by selecting **Hardware** > **Generate BitStream**. You can also find **Generate BitStream** in the Implement Flow section of the Navigator window

It will take more than ten minutes (depends on your computer).

Click on the Design Summary tab (Figure 13). The Design Summary allows you to quickly access design overview information, reports, and messages. It displays information specific to your targeted device and software tools. The pane on the left side of the Design Summary allows you to control the information displayed in the right pane.

Figure 13: XPS Design Summary Tab

Adding and Downloading Software

This lab guides you through the process of adding software and downloading the design onto the evaluation board using the Xilinx Software Development Kit (SDK). It presupposes the presence of a hardware design which is generated in previous lab part.

There are three main steps in this lab that will take an existing hardware design (provided for you) to conclusion. The first step is to launch the SDK tool from the XPS tool and create a new workspace. The second step will walk you through the process of creating a new software project that will be added to the workspace. This software project will be generated using a template from a selection of sample applications provided by SDK. Finally, you will download the software application to E2LP board.

This lab part 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.

1. Lunch XPS project and run **Project** > **Export Hardware Design to SDK** or run **Export Design** from Implementation Flow.
2. In pop-up menu select **Export & Lunch SDK.**
3. Select a workspace. We recommend you to select directory below the embedded project in *SDK/SDK_Workspace. Click **OK**. SDK will be lunched.*
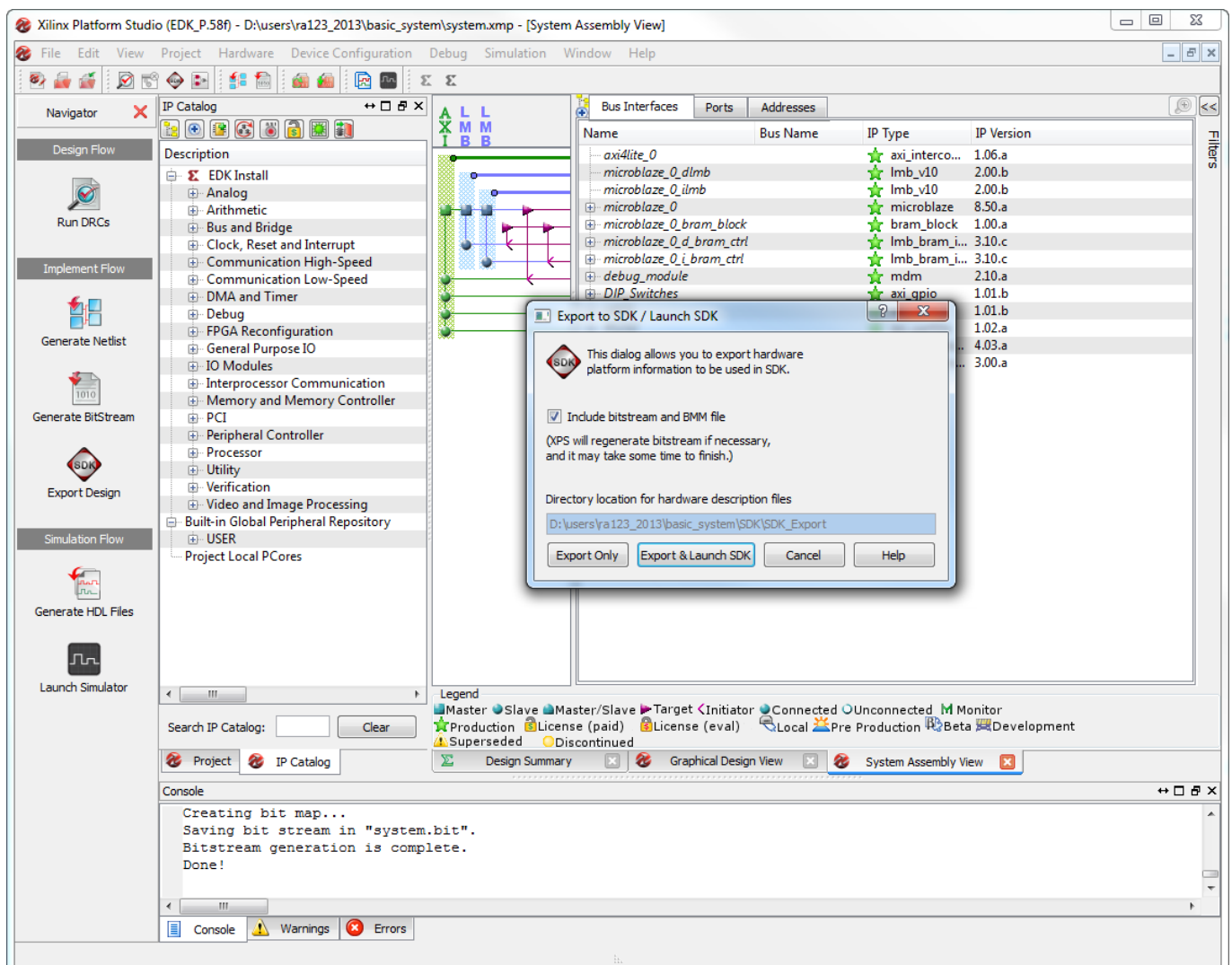
Figure 14: XPS, Export Design
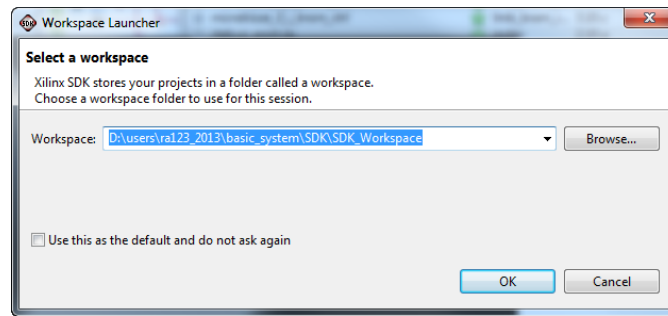
Figure 15: SDK, Select Workspace

SDK takes the hardware description from the XPS tools and generates a model of the hardware from which other software is based. This hardware description project takes the name of *basic_system_hw_platform (like name of folder which contain the project).* From here, you will construct a number of other projects based on this hardware platform description.

Upon automatic launch when exporting embedded hardware, SDK creates a hardware platform project in the new workspace that you specified in the last step. Based on this hardware platform description, you will build a board support package (BSP) and a software application. All of these projects will be members of the SDK workspace.

The BSP is a collection of files that support the processor and peripherals listed in the hardware platform description.

Next, you will create a software project by using one of the available project templates ("Hello World") that SDK provides. When creating a simple software application project, SDK can auto-create the BSP project and automatically build the software application project and produce an Executable and Load Format (ELF) file.
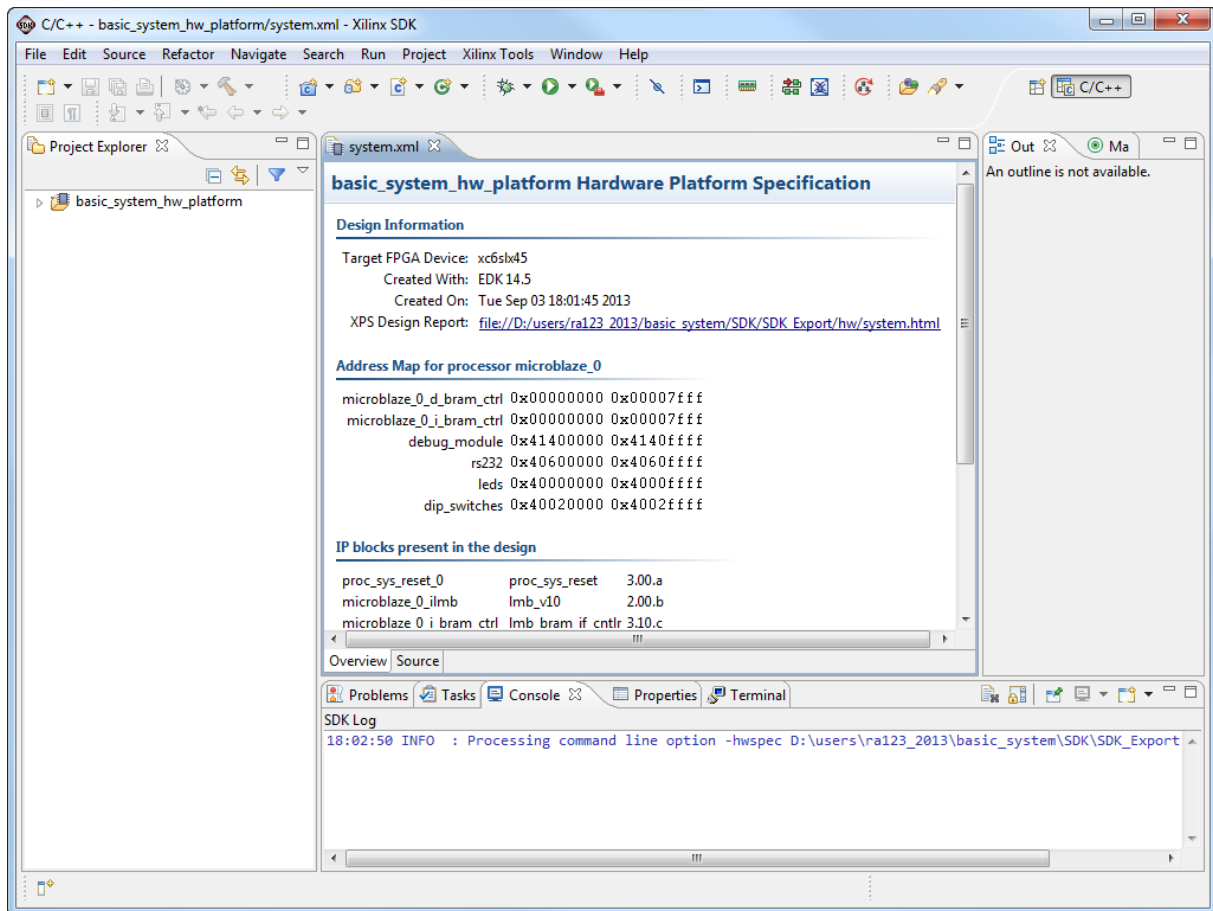
Figure 16: SDK, Main Window

1. Add a C language application project. Although SDK supports multiple software projects, you will only be adding one in this lab. There are different types of software application templates, the simplest being a Xilinx C project. In this type of project, a software designer will typically write code beginning with the main{} C function.

2. In SDK, select **File** > **New** > **Other**. **Select Xilinx > Application Project.** Click **Next**.
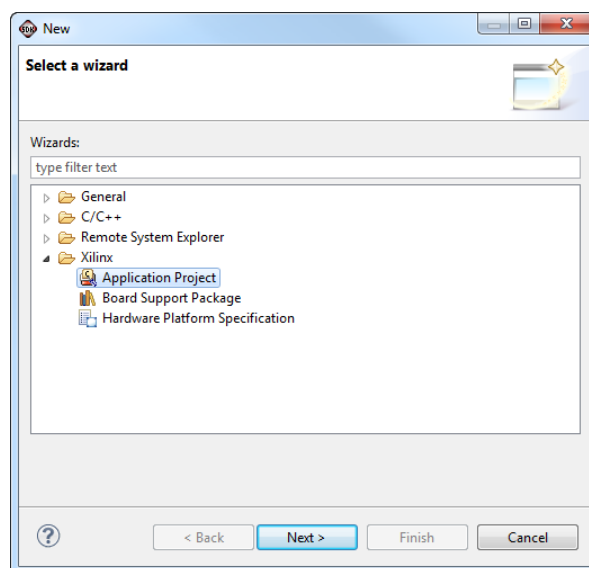
Figure 17: SDK, Select Xilinx Application Project

3. Enter a name for project. It will be the simple Hello World Application so **Project Name** can be *hello_world*.

4. Target Hardware is previously created

    Hardware Platform: *basic_system_hw_platform*

    Processor: *microblaze_0*

5. Target Software:

    OS Platform: standalone

    Language: C

    Board Support Package: Create New (with default name *hello_world_bsp*)

6. Click **Next**. From available templates pick **Hello World**. Click Finish.

7. Also, You can manually create BSP. Choosing select **File** > **New** > **Other**. **Select Xilinx > Board support Package**. And on application project (*hello_world*) right click and choose **Change Referenced BSP**.
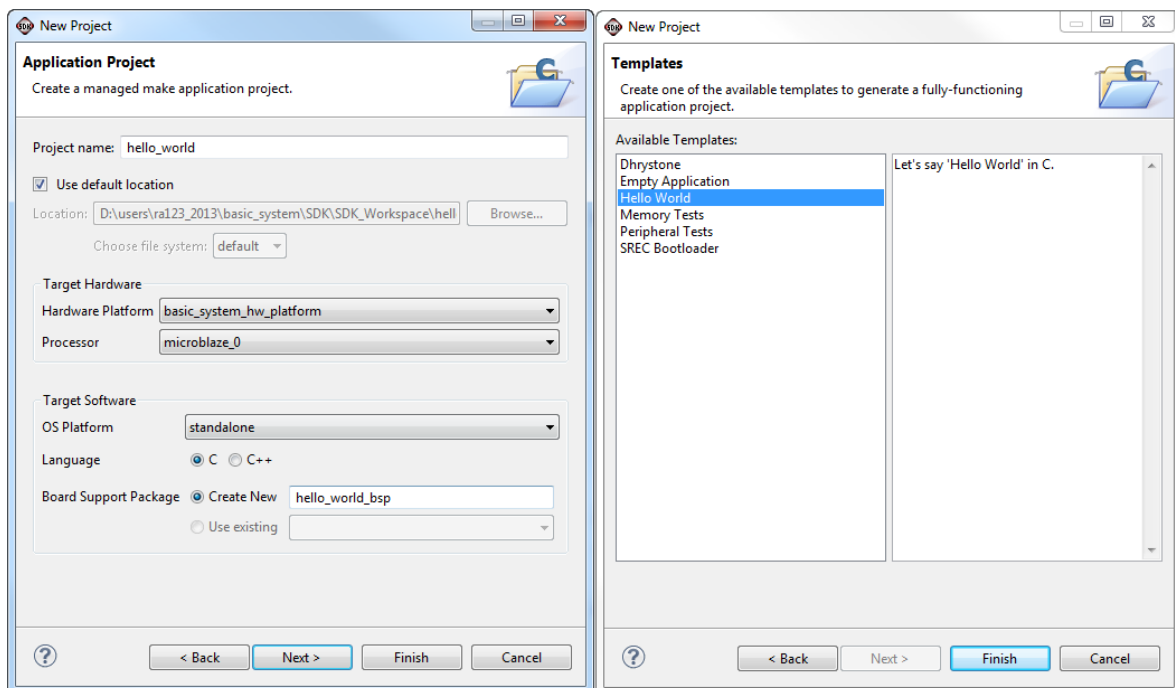
Figure 18: SDK, Add Project, from Template and BSP

Follow the status info at the bottom right of SDK window to identify when the build completes. You may also view the status in the Console window.

The *hello_world* application contains all the source files needed to build the object *executable.elf* file. The project will automatically build without errors. A successful build is indicated in the Console tab when the sizes of all of the program segments are displayed before the build complete message.
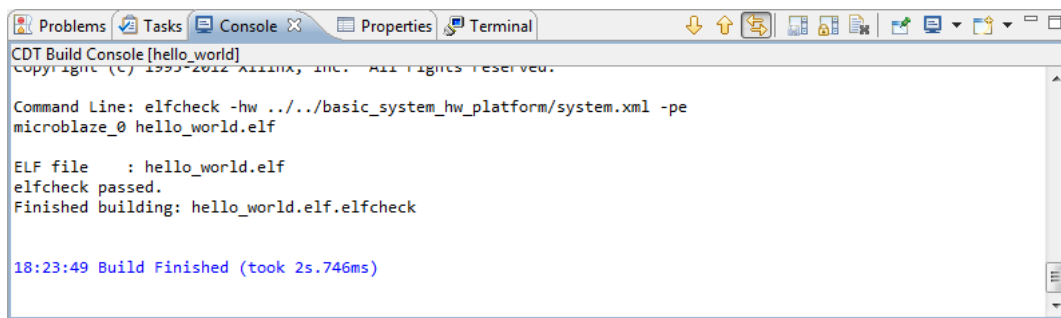
Figure 19: SDK, Successful Software Application Build

The Project Explorer tab shows a tree structure for the software application under the hardware system. Although it may appear as if there are several software applications, this is not the case. For just the *hello_world* application, there are several entries: the hardware platform description, the board support package, and the application itself. Other entries may be present depending on how the workspace is shared. For this lab, *hello_world* is the application, the *hello_world_bsp* project is the BSP that contains all of the supporting software that bridges the hardware to the application software, and the *basic_system_hw_platform* project is the description of the hardware platform.

1. Expand **hello_world** > **src** to see all of the source files that are part of this project by clicking the plus ('+') sign next to src.
2. Double-click the **helloworld.c** application file to open it.

Note the print statement that should be printed after the application is downloaded and executed. Besides opening source files, the software developer can use this view to open other resource files and set tool options for software applications and software platforms.
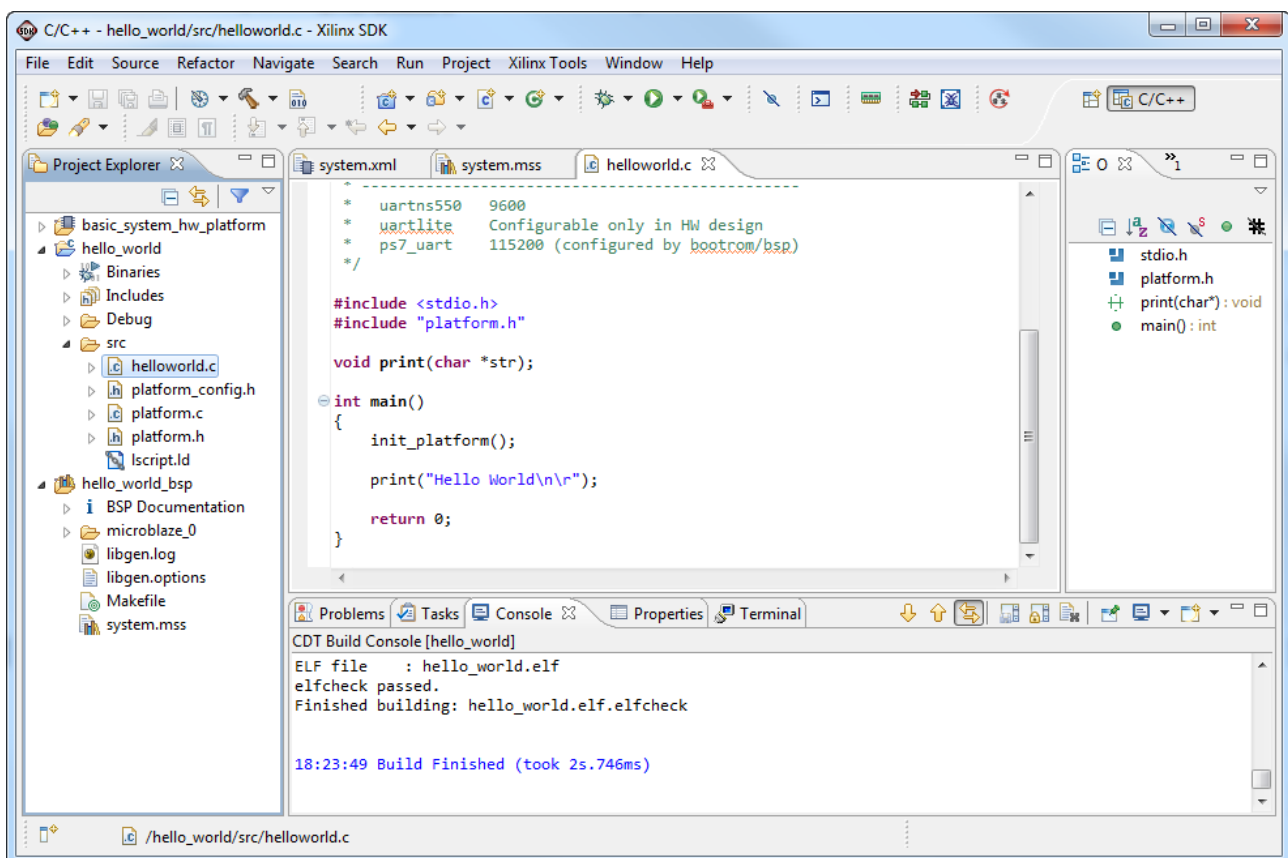
Figure 20: SDK, Expanding Source Structure

Exercise:

Change the code for **hello_world** example with following one:

```c
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xio.h"


int main()
{
    unsigned int DataRead;
    init_platform();

    print("Hello World\n\r");
    while(1){
        DataRead = XIo_In32(XPAR_DIP_SWITCHES_BASEADDR);
        xil_printf("\n\rDataRead = %x", DataRead);
    }

    return 0;
}
```

1. What will be the effect of this code?
2. What is xparameters.h? It is placed in hello_world_bsp/include.
3. What is platform.h?
4. How would you print SW status on LEDs?

Downloading the Design

You will now take the final step and download bitstream and the ELF file to the hardware and monitor the output of the system. Prepare the lab environment for test. You will need to verify the proper connections and open a terminal window on the PC to monitor the output of the *hello_world* application. You may need to change the assigned COM port based on your computer's configuration.

1. Connect the E2lp board to your machine.
2. Connect USB UART and USB PROG cable.
3. Power up the board.
4. Select **Xilinx Tools** > **Program FPGA**.
5. Make sure that the bitstream path is pointing to correct **.bit* file.
6. Select the software application ELF from the ELF file to Initialize in Block RAM drop-down list (use *hello_world.elf*).
7. Click **Program** to download the hardware bitstream.
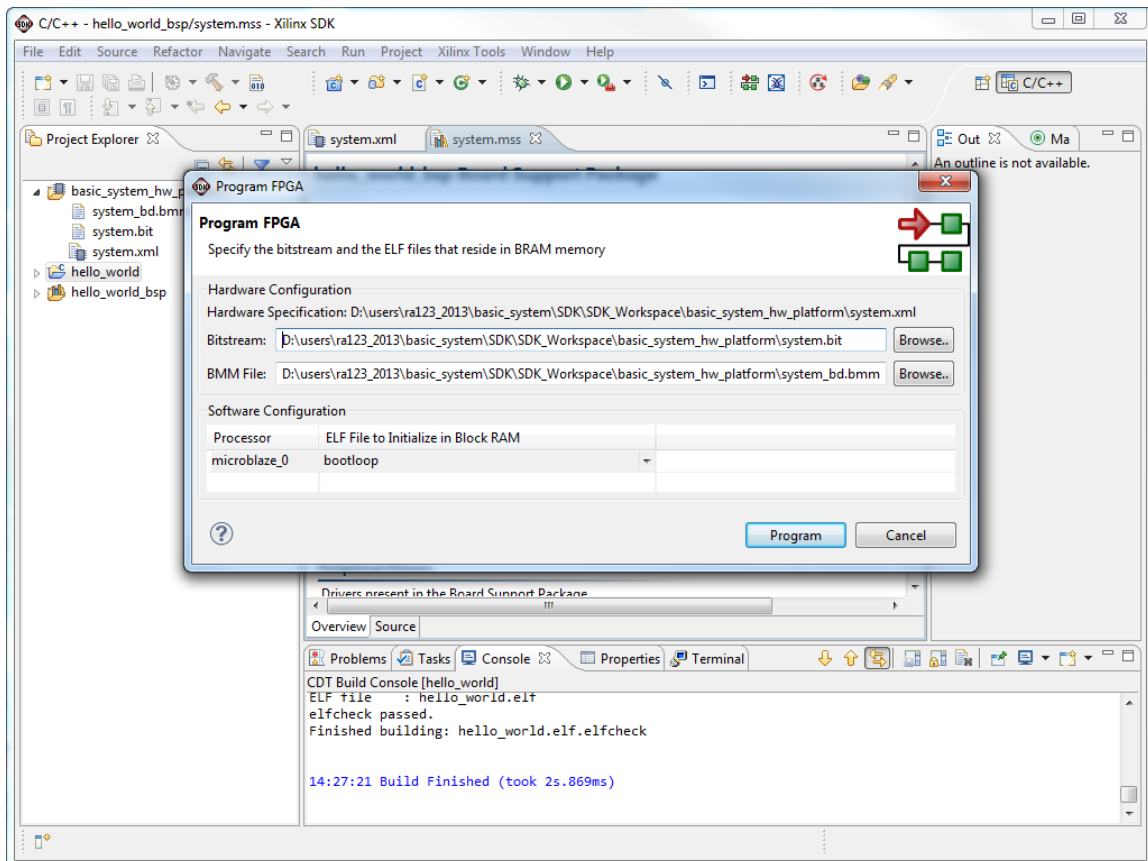
Figure 21: SDK, Programming the FPGA

The system monitoring

1. Open a **Terminal** window in SDK, if not already open.
2. terminal tab would appear next to the Problems, Tasks, Console, and Properties tabs under the Edit window.
3. If you do not see a Terminal tab, select **Window** > **Show View** > **Terminal**.

4. Click the **Connect** icon (   ) to open the Terminal Settings dialog box.

5. Configure the settings as shown in the following figure.

6. **Note:** The COM port setting is specific to the computer being used and may need to be different than shown. Your instructor will provide you with the proper port number.

7. Click **OK**.
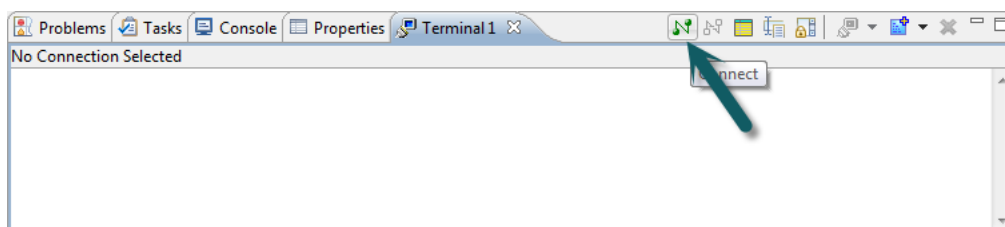
Figure 22: SDK, Terminal Tab
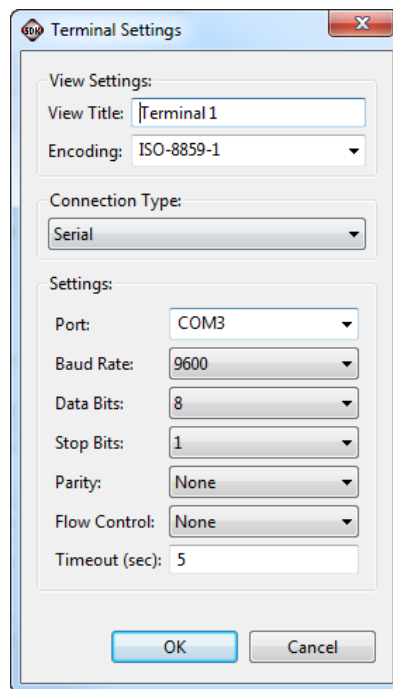
Figure 23: SDK, Terminal Settings

Create Run configuration

Create a new Run configuration named *hello_world Debug* and test the application. A run configuration contains a collection of information describing how the application is to operate on the hardware, including how stdio is used.

1. In the Project Explorer tab, right-click the **hello_world** software application and select **Run As** > **Run Configurations.**
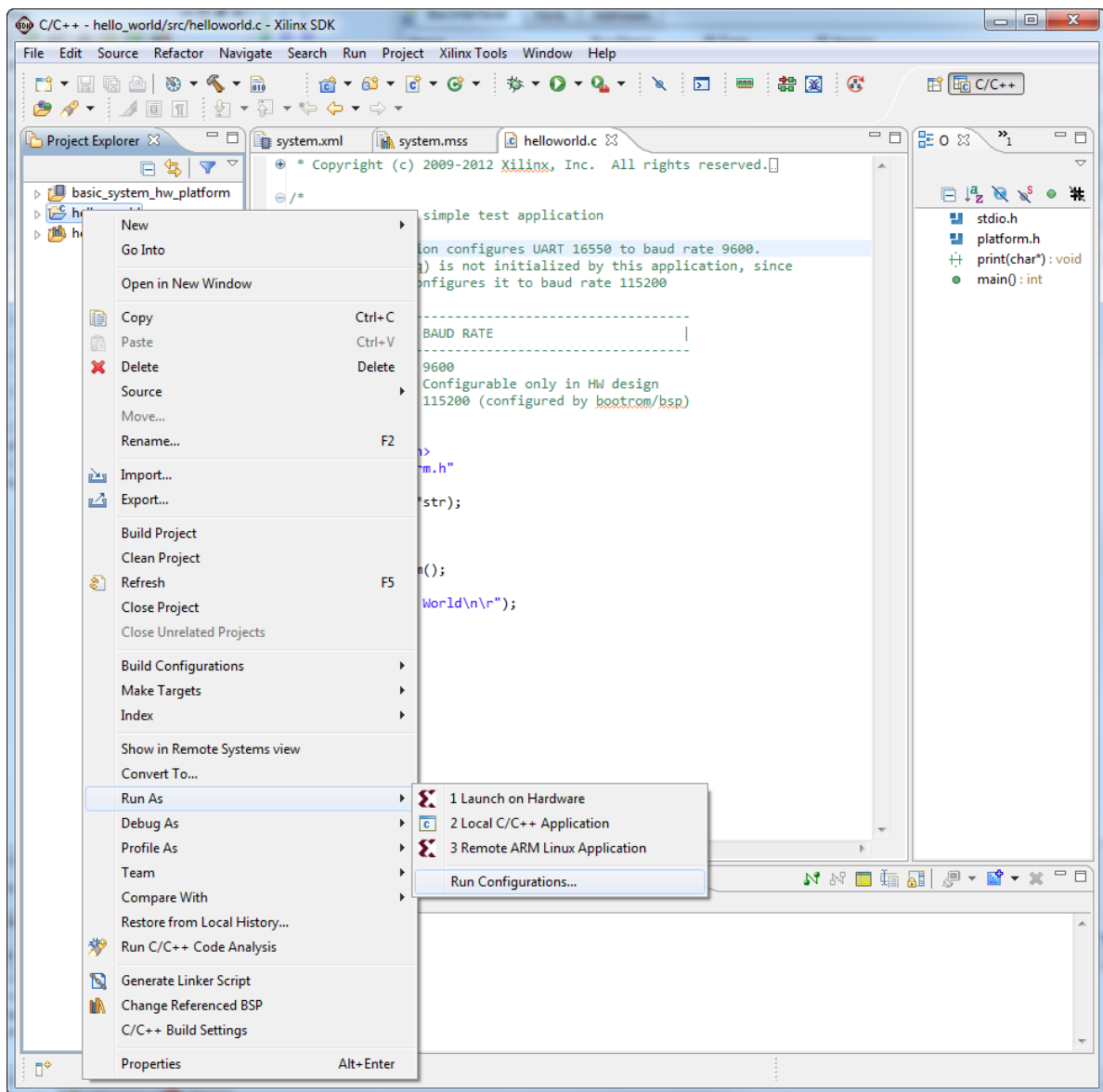
Figure 24: SDK, Selecting Run Configuration

2. When the Run Configurations dialog box opens, double-click **Xilinx C/C++ ELF** in the left pane. A default Run configuration named *hello_world Debug* will automatically be created.

3. Click **Run**.

4. Select the **Terminal** tab and view the output of the application.

Figure 25: SDK, Application Output in Terminal

Custom AXI IP

This lab guides you through the process of creating a simple IP peripheral (instead of using the XPS General Purpose IO peripheral provided by Xilinx) to read from the DIP switches and write to the LEDs. The software application will display the DIP switches values on the LED outputs and also send values to the UART.

Any custom logic (IP) that you design must connect to the AXI to communicate with the MicroBlaze processor. When connected to the AXI, the IP becomes part of the memory map accessible to the MicroBlaze. The IP will have a base address and a high address signifying where in the memory map it resides, and how much of the memory map it occupies. The MicroBlaze interacts with the IP as though it were part of the memory.

In this example, we will use the Peripheral Wizard to create a basic IP peripheral that connects to the AXI and implements one 32 bit register. Writing to the peripheral through the AXI will allow us to change the contents of the register. The outputs of the register will drive the LEDs. We will modify the core so that reading from it through the AXI will return the DIP switch settings.

This lab guides you through the process of creating a simple IP peripheral (instead of using the XPS General Purpose IO peripheral provided by Xilinx) to read from the DIP switches and write to the LEDs. The software application will display the DIP switches values on the LED outputs and also send values to the UART.
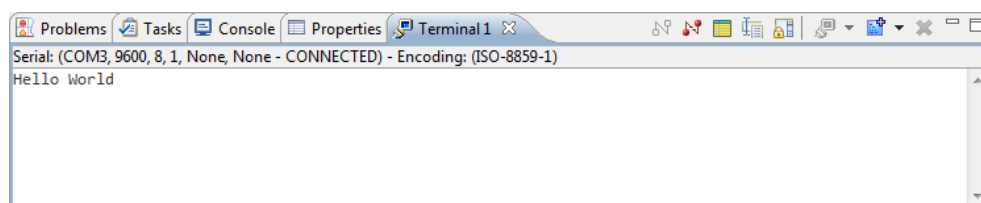
Any custom logic (IP) that you design must connect to the AXI to communicate with the MicroBlaze processor. When connected to the AXI, the IP becomes part of the memory map accessible to the MicroBlaze. The IP will have a base address and a high address signifying where in the memory map it resides, and how much of the memory map it occupies. The MicroBlaze interacts with the IP as though it were part of the memory.

In this example, we will use the Peripheral Wizard to create a basic IP peripheral that connects to the AXI and implements one 32 bit register. Writing to the peripheral through the AXI will allow us to change the contents of the register. The outputs of the register will drive the LEDs. We will modify the core so that reading from it through the AXI will return the DIP switch settings.

Creating a Custom AXI IP Using the Wizard

We will use the Create/Import Peripheral (CIP) wizard in XPS to create a new custom IP for the existing system. The custom IP will read from the DIP switches and write to the LEDs controlled using a software mapped register.

1. Launch the XPS tool and load previously created project.
2. Go to **Hardware > Create or Import Peripheral…** Click on **Next**.
3. CIP wizard will appear. Click on **Next**.
4. Make sure that **Create templates for a new peripheral** is selected then click **Next**.
5. We must now decide where to place the files for the peripheral. They can be placed within this project, or they can be made accessible to other projects. Select **To an XPS project**. Click on **Next**.
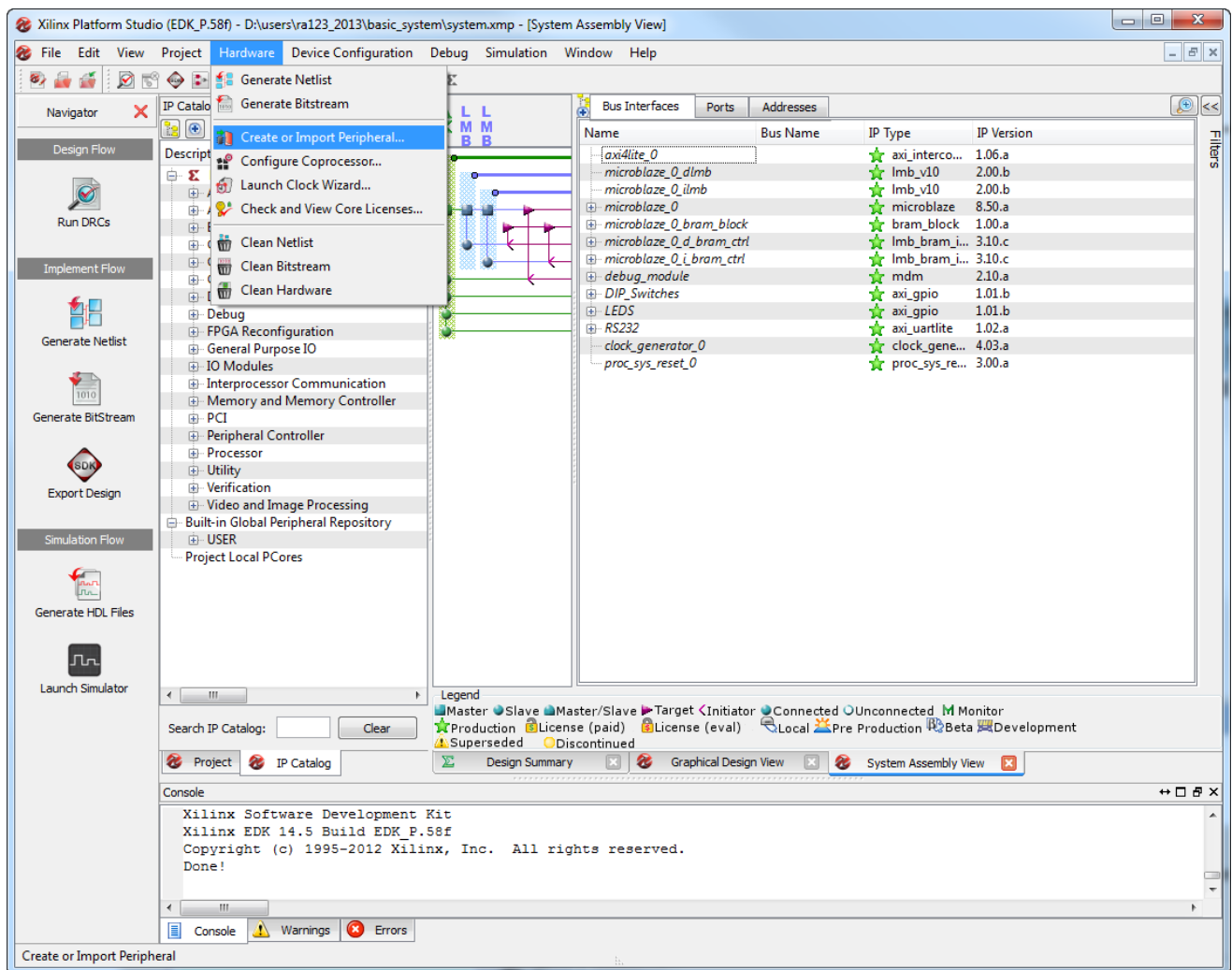
Figure 26: Xilinx Platform Studio
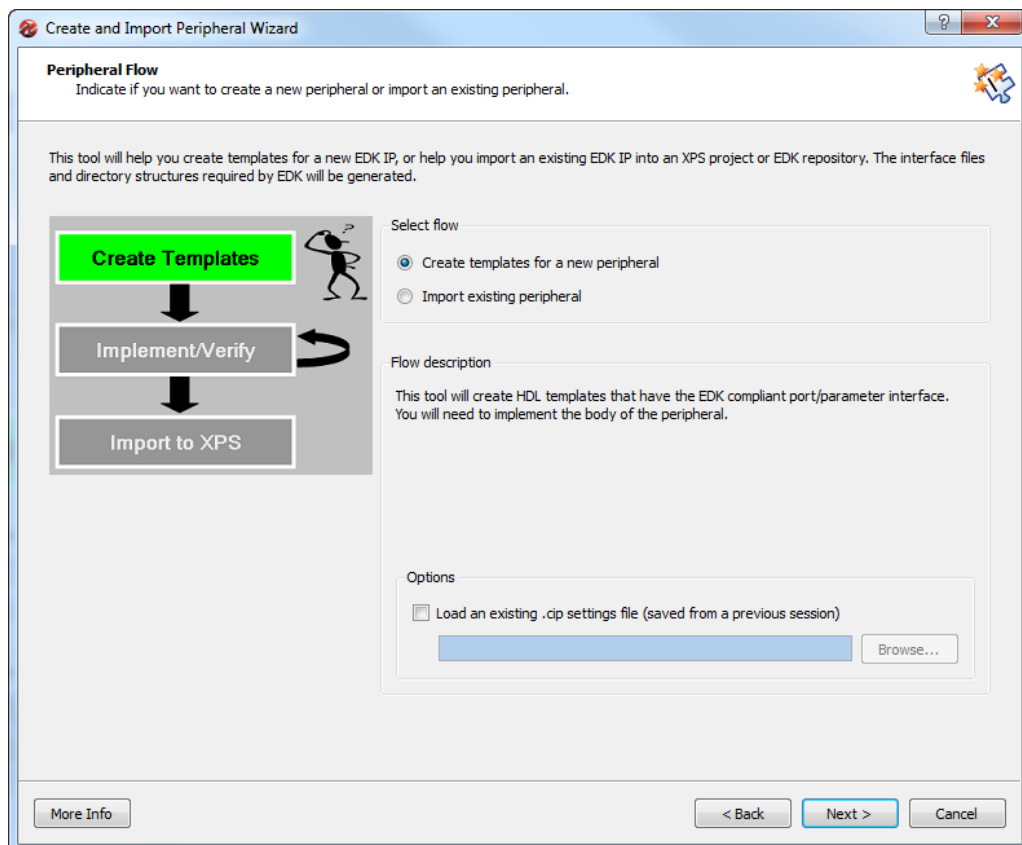
Figure 27: Create Peripheral Wizard
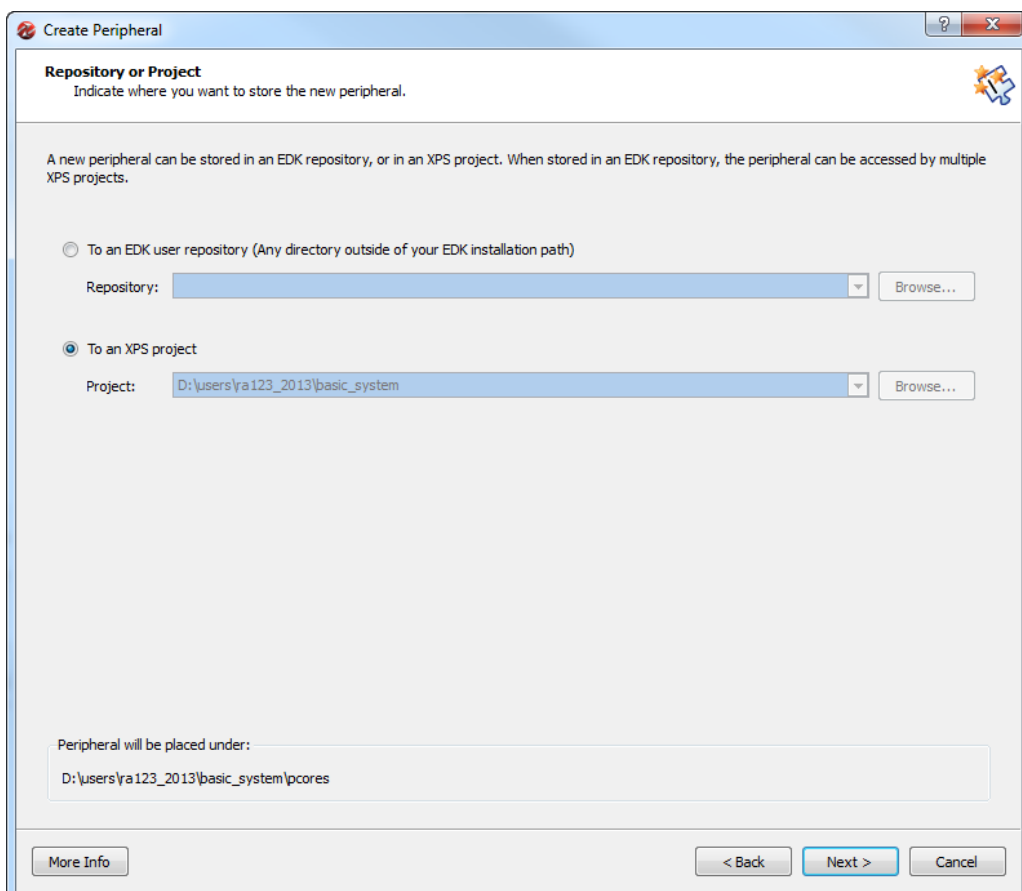
Figure 28: Where to Store Repository

1. On the **Name and Version** page, type *my_peripheral* for the peripheral name. Click **Next**.

2. Select **AXI4-Lite: Simpler, non-burst control register style interface**. Click **Next**.
   The IPIF is a module isolating the user interface from the bus. In addition to facilitating bus attachment, the IPIF provides additional optional services. The services include software registers, user address ranges, FIFOs, software reset, interrupt support and bus-master access. You can click on the **More Info** button, then select **AXI Bus Interface IPIF Features for AXI** to see a description of each feature.

3. We will be using software registers to control the peripheral. Select **User logic software register**. We'll choose how many registers in the next screen. **Unselect** all other choices. Click **Next.**

4. On the **User S/W Register** page, choose *1* for the number of software accessible registers. As the MicroBlaze is a 32-bit processor, we will choose 32-bits for the size of the register, even though we only need 8-bits for the 8 LEDs. Click on **Next**

5. The IP Interconnect (IPIC) uses a set of signals between the user logic and the AXI bus. We will use the default signals already selected. Click on **Next**.

6. Bus Functional Models can be generated to accelerate the IP verification. This tutorial does not cover the BFM simulation of the peripheral. Click on **Next**.

7. The wizard can also generate custom drivers for the peripheral and an ISE project. We will be using XPS and writing our own code to test the peripheral. This is also where you can select a Verilog template versus the default, VHDL. Leave it as VHDL for this tutorial. Click on **Next**.

8. Click on **Finish** to create the peripheral.

Figure 29: CIP Wizard, Step 1
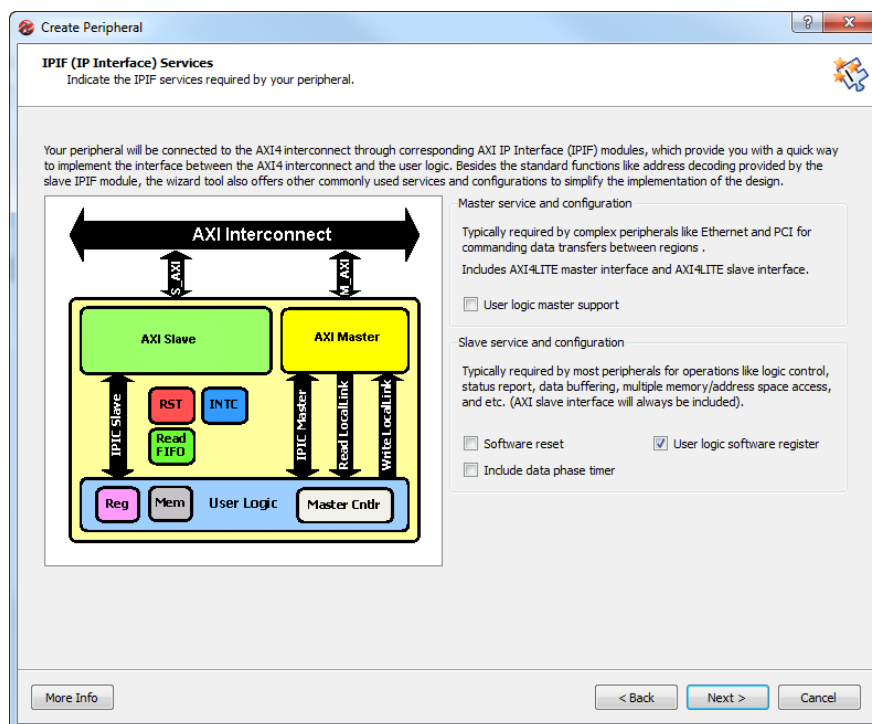
Figure 30: CIP Wizard, Step 2
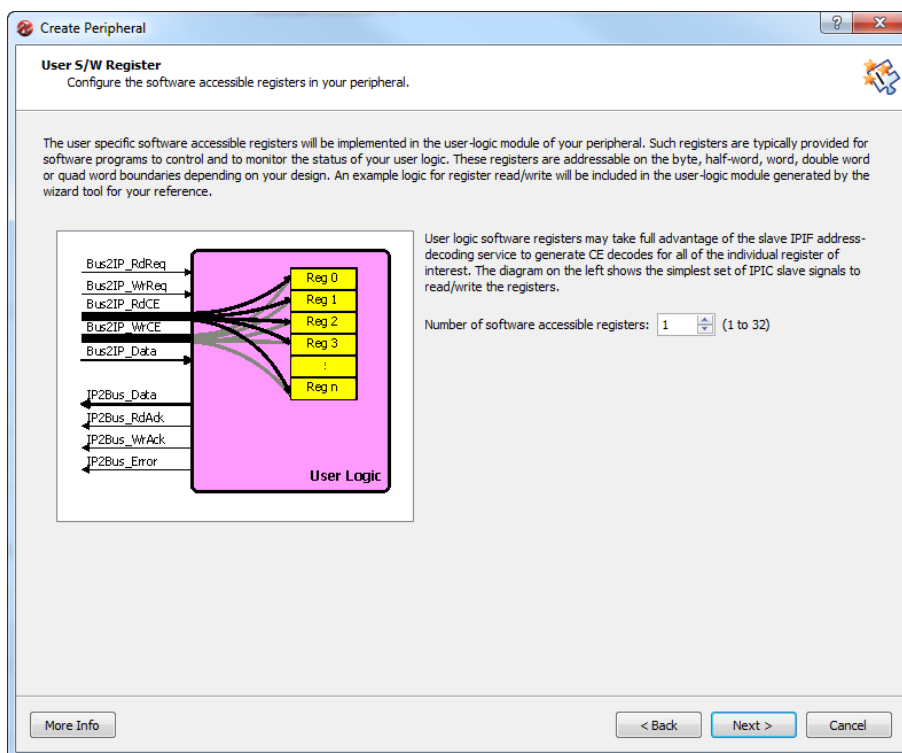
Figure 31: CIP Wizard, Step 3
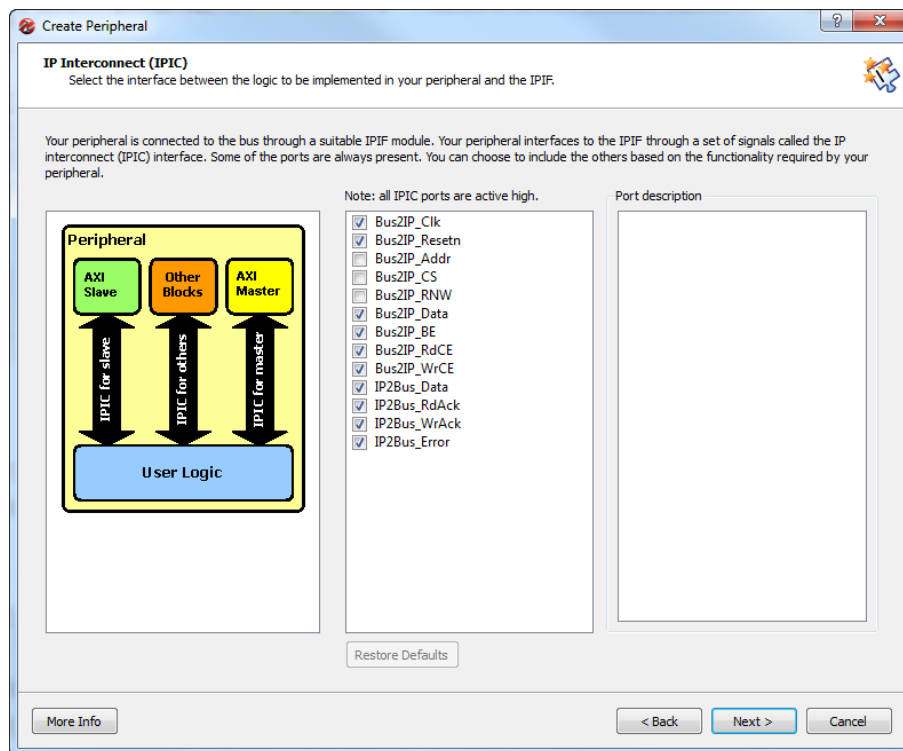
Figure 32: CIP Wizard, Step 4
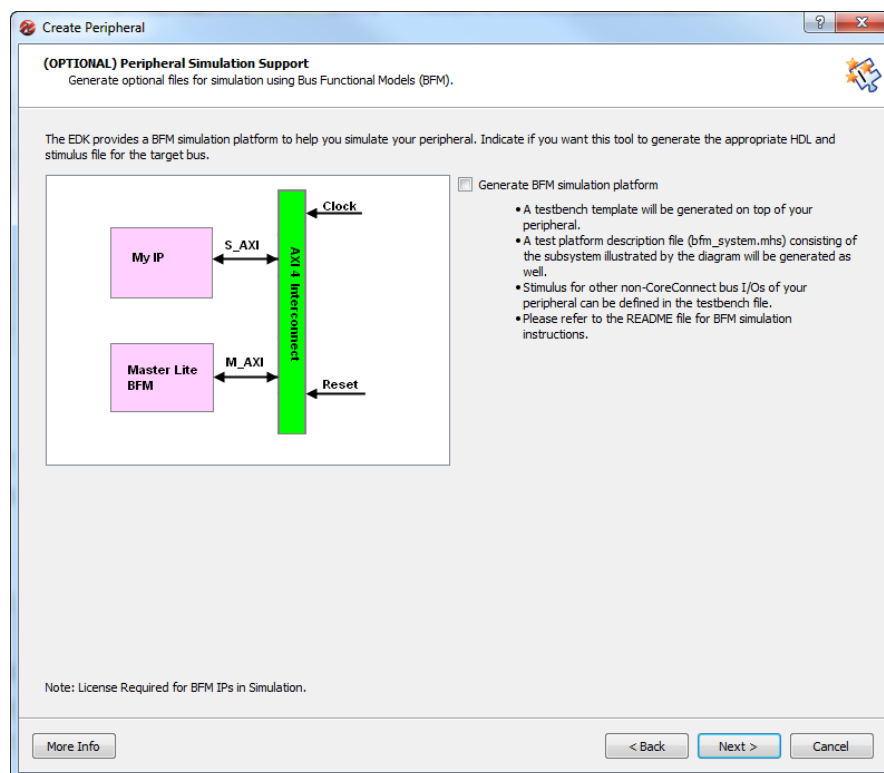
Figure 33: CIP Wizard, Step 5
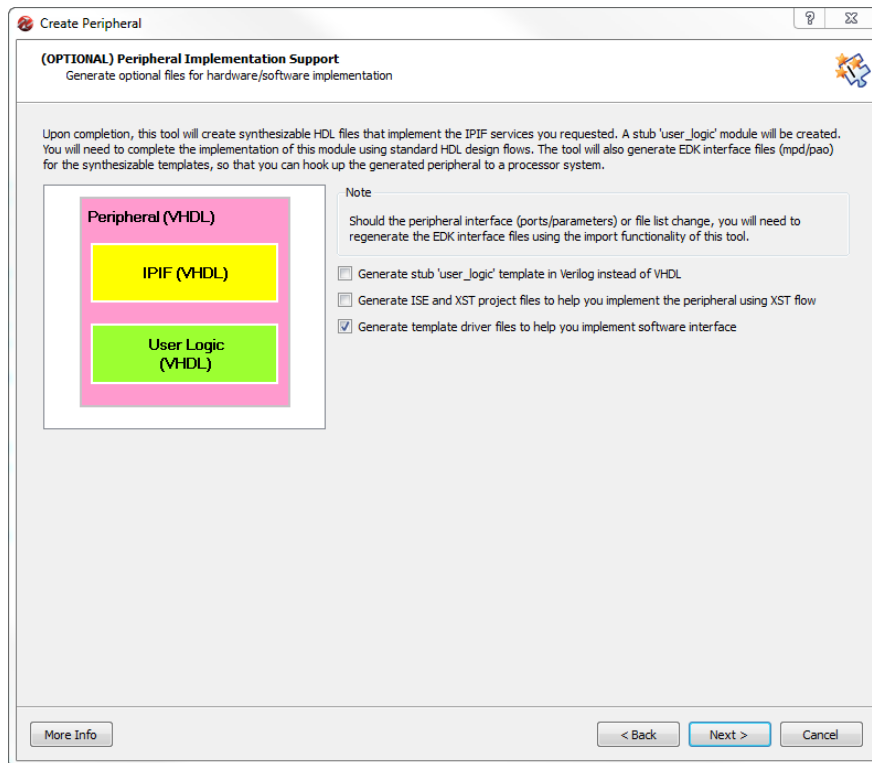
Figure 34: CIP Wizard, Step 6
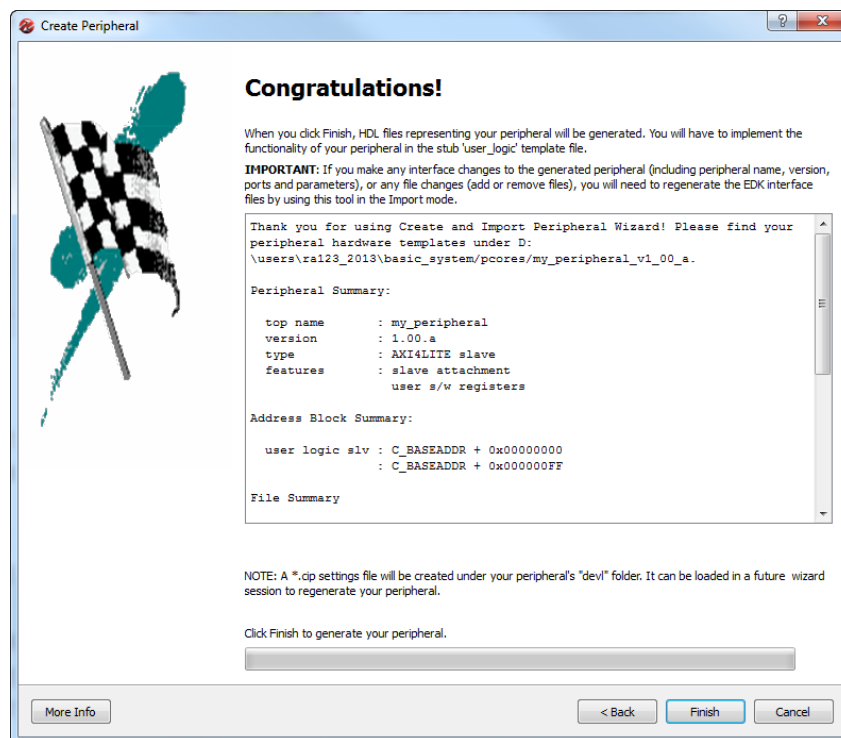
Figure 35: CIP Wizard, Step 7

Customizing the new Peripheral

The template created by the Peripheral Wizard implements a 32-bit register that we can read and write to via the AXI. When the peripheral is instantiated, we could access the register by reading and writing to the base address of the peripheral. We need to modify this functionality slightly, because of two things:

1. We want to connect the outputs of the register to the LEDs – to achieve this we need to create an output port on the peripheral and connect it to the output of the register.

2. We want to make all reads from the peripheral return the values of the DIP switches – to achieve this we need to create an input port on the peripheral and make it return these values on every read request.

Follow these steps to modify the peripheral:

Select from the menu **File**->**Open** and browse to *pcores\my_peripheral_v1_00_a\hdl\vhdl* from the project folder. This folder contains two source files that describe our peripheral: *my_peripheral.vhd* and *user_logic.vhd*. The first file is the top level VHDL entity for the new IP. It contains two instances, AXI_LITE_IPIF and *User Logic*. The AXI_LITE_IPIF creates a proxy between the user logic and the AXI interface based on the settings we selected in the Create Templates for a New Peripheral Wizard The second file is where we place our custom logic to make the peripheral do what we need it to do. This part is instantiated by the first file.

1. Open the file *my_peripheral.vhd*. We will need to add two ports to this source code, one for the LEDs and one for the DIP switches.

2. Find the line of code that says –ADD USER PORTS BELOW THIS LINE and add these two lines of code:

   ```
   LED_Data : out std_logic_vector(7 downto 0);
   DIP_Data : in std_logic_vector(7 downto 0);
   ```

3. Find the line of code that says "–MAP USER PORTS BELOW THIS LINE" and add these two lines of code:

   ```
   LED_Data => LED_Data,
   DIP_Data => DIP_Data,
   ```

4. Save and close the file.

5. Open the file *user_logic.vhd*. We will need to modify this source code to include the two new ports and to modify the behavior.

6. Find the line of code that says "–ADD USER PORTS BELOW THIS LINE" and add the following two lines of code.

   ```
   LED_Data : out std_logic_vector(7 downto 0);
   DIP_Data : in std_logic_vector(7 downto 0);
   ```

7. Find the line of code that says "–USER logic implementation added here" and add the following line of code below. This connects the LED port to the first 8 bits of the register output.

   ```
   LED_Data <= slv_reg0(7 downto 0);
   ```

8. Find the line of code that says:

   ```
   IP2Bus_Data <= slv_ip2bus_data when slv_read_ack = '1' else (others => '0')
   ```

9. and replace it with the code below. Now, when the peripheral is read on the bus, it will always return the DIP switch settings. As the bus is 32 bits long, we just fill the unused bits with zeros using the "others" keyword.

   ```
   IP2Bus_Data(31 downto 8) <= (others=>'0');
   IP2Bus_Data(7 downto 0) <= DIP_Data(7 downto 0);
   ```

10. Save and close the file.

11. The new external port needs to be added to the definition file for the peripheral in order to be used in XPS. Open the **my_peripheral_v1_00_a\data** directory and open the file **my_peripheral _v2_1_0.mpd**.

12. Add the following ports:

```
## Ports
PORT DIP_Data = "", DIR = I, VEC = [7:0]
PORT LED_Data = "", DIR = O, VEC = [7:0]
```

13. Save and close the file.

14. If you and additional files in project it should be included in *.pao file placed in same place as ***.mpd**. In **my_peripheral_v1_00_a\data.**

```
lib my_peripheral_v1_00_a user_logic vhdl
lib my_peripheral_v1_00_a my_peripheral vhdl
lib my_peripheral_v1_00_a vhdl_file_name vhdl
```

15. Save and close the file.

16. If you project contains any ***.ngc** file it should be place in directory **my_peripheral_v1_00_a\netlist.** Usually this folder don't exist (wizard will not generate this folder) so you should create it. Also you should create **my_peripheral.bbd** file in folder **my_peripheral_v1_00_a\data** with names of used *.ngc files. Similar to the following content:

```
Files
ngc_file_name.ngc
```

17. Save and close the file.

18. In XPS, rescan the user IP directories. **Project > Rescan User Repositories**.

19. The new core will now be available.

Adding the Custom Peripheral to the System

We will add and connect the new custom IP to the existing system following the same instructions as in the previous lab. We will remove the GPIO peripheral for the LEDs and connect the custom made peripheral to the LEDs.

1. In XPS, click on the **IP Catalog** tab in the Project Information Area.
2. Expand the **Project Local Pcores/USER** list to view the custom IP.
3. Select **my_peripheral** then drag and drop it to the **System Assembly** V**iew** window. Click **OK**.
4. Click **OK** to connect this IP to MicroBlaze.
5. Click on the **Addresses** tab to view the address range for the new IP.
6. Delete the GPIO peripheral instance for the LEDs. In the **System Assemble View**, right-click on the **LEDS** instance and select **Delete Instance**. Select **Delete instance but do not remove the nets**. Click **OK**.
7. Click on the **Ports** tab. Expand **my_peripheral_0** from the list. It will show the connections available for the peripheral.
8. For **DIP_Data** click on the Net column and select **New Connection** from the left drop-down list select **External Ports** and right **DIP_Switches_TRI_I**.
9. For **LED_Data** click on the Net column and select **New Connection** from the left drop-down list select **External Ports** and right select existing **LEDS_TRI_O.**
10. Close **XPS**.
11. There is no need to update the constraints file since we are reusing the existing external ports.

12. Double-Click on **Export Hardware Design to SDK** to update the bit file with the new peripheral. This may take several minutes.
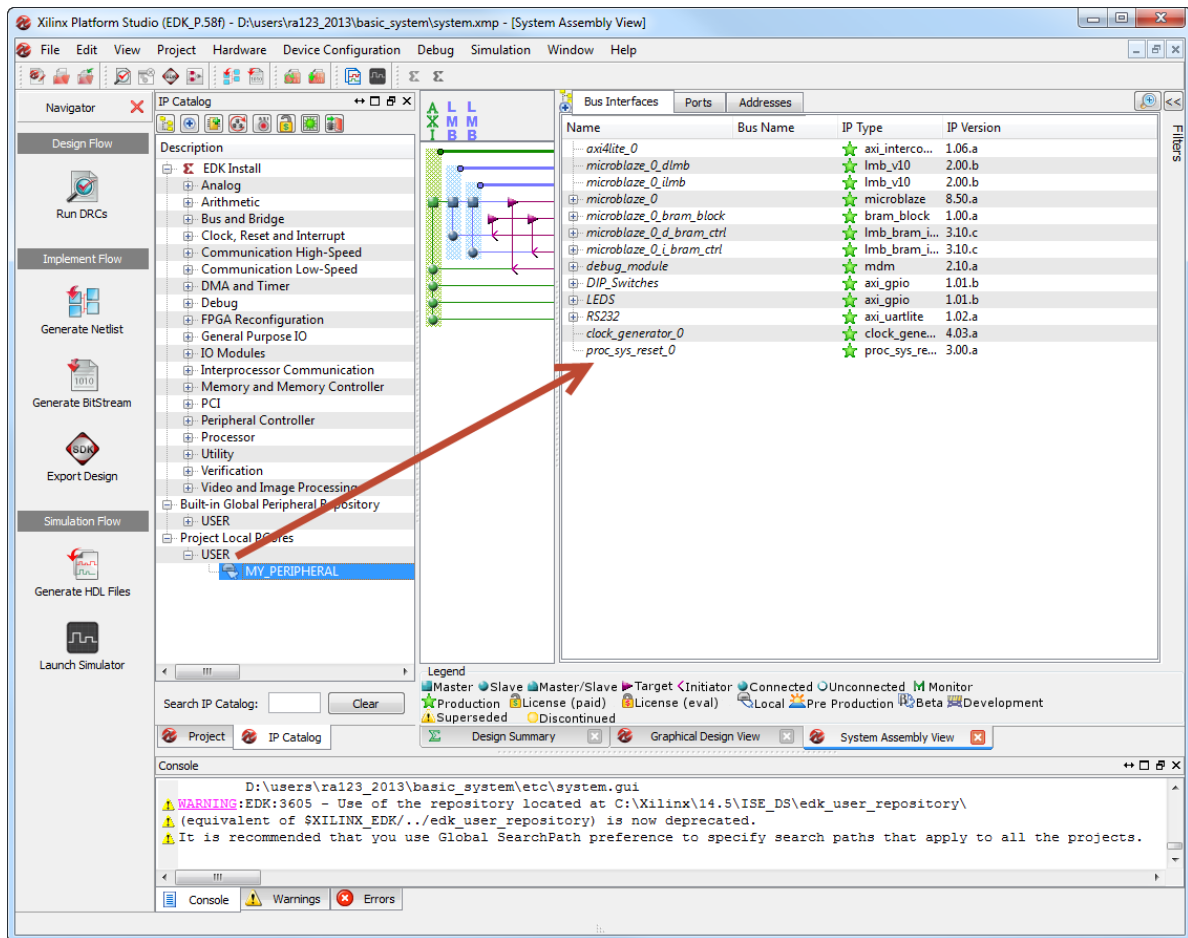
Figure 36: Add peripheral to the system
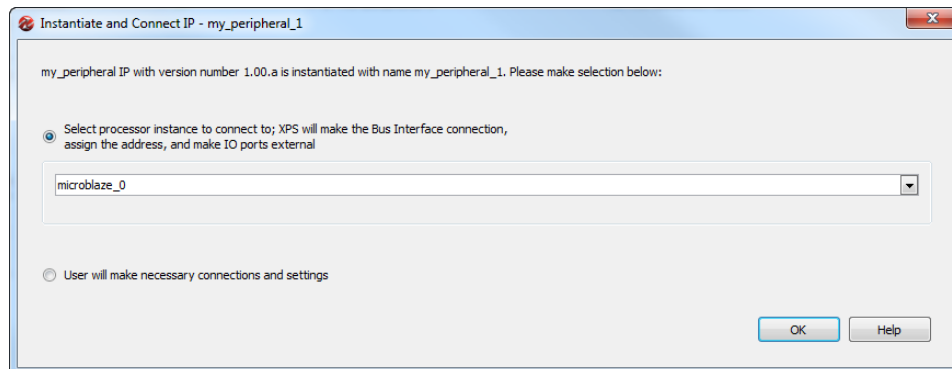
Figure 37: Select Processor Instance
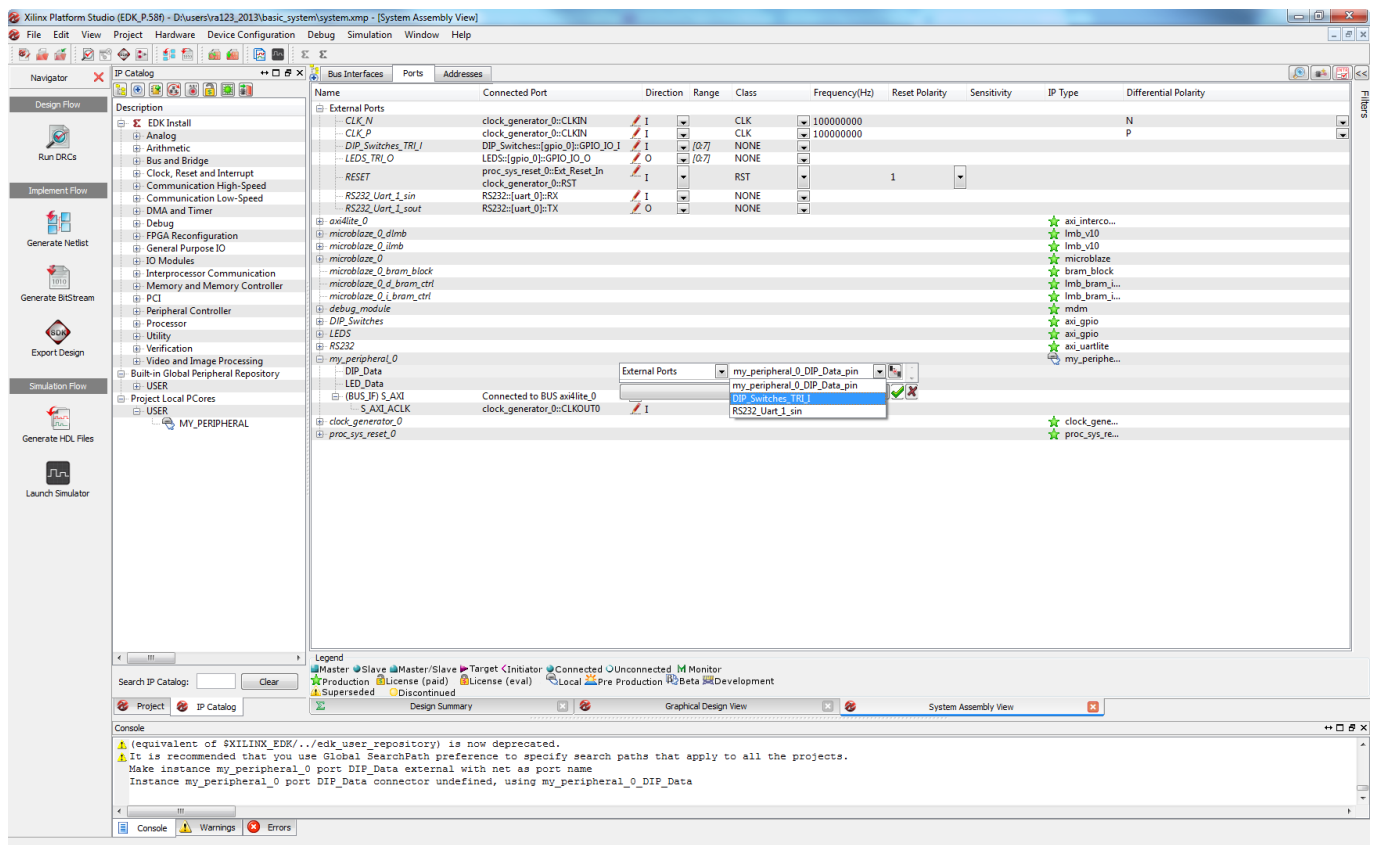
Figure 38: Connect DIP_Switches

Writing Code for Custom Peripheral

To test the new peripheral we will add code to the **Tutorial_Test** project created with Platform Studio SDK. We will make use of the DIP switches to control the LEDS. The 8 DIP switches will be used to select the 8 LEDS.

1. Since, the last step was to export Design to SDK, our hardware platform is updated to the current hardware design.
2. Start Xilinx SDK and select the previous used Workspace.
3. Expand the **hello_world_bsp** project then **microblaze_0** in the **Project Explorer** window. Expand the **include** directory. Double click on the **xparameters.h** file to view the driver parameters for the custom peripheral:
4. Scroll down to view the address for custom peripheral.

```
/* Definitions for peripheral AXI_PWM_0 */
#define XPAR_MY_PERIPHERAL_0_BASEADDR 0x7DE00000
#define XPAR_MY_PERIPHERAL_0_HIGHADDR 0x7DE0FFFF
```

We can modified existing *hello_world* application, or using the same stapes to create new for example *my_peripheral_test* (for the new application we can use Hello World template to setup platform, and then changed according to the our requirements). In main file you should include *platform.h* and call *init_platform()* procedure.

```
#include <stdio.h>
#include "platform.h"

int main()
{
    init_platform();
```

```
        // Add application code

        return 0;
    }
```
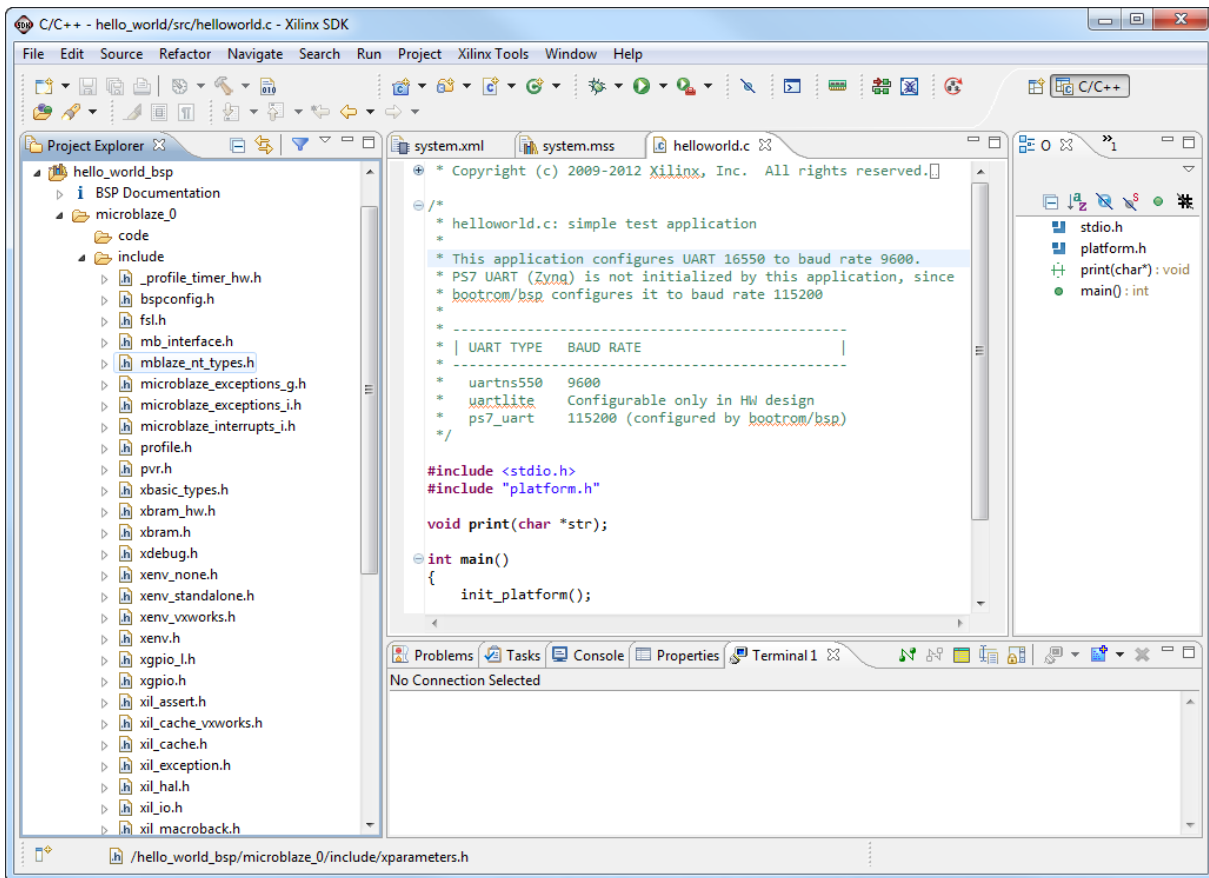

Figure 39: SDK Window

1. Copy the code below into this new file and save the file as "*my_peripheral_test.c*" in a new folder called *my_peripheral_test* within the project folder. We use the *XPAR_MY_PERIPHERAL_0_BASEADDR* definition given in the *xparameters.h* file which is created automatically by XPS. By using this definition, instead of a fixed value, we don't have to modify our code each time we modify the hardware and addresses.

```c
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xio.h"

int main () {
    unsigned int DataRead;
    unsigned int OldData;

    init_platform();

    // Clear the screen
    xil_printf("%c[2J",27);

    OldData = (unsigned int) 0xffffffff;
    while(1){
        // Read the state of the DIP switches
        DataRead = XIo_In32(XPAR_MY_PERIPHERAL_0_BASEADDR);
        // Send the data to the UART if the settings change
        if(DataRead != OldData){
            xil_printf("DIP Switch settings: 0x%2X\r\n", DataRead);
            // Set the LED outputs to the DIP switch values
            XIo_Out32(XPAR_MY_PERIPHERAL_0_BASEADDR, DataRead);
```

```
                    // Record the DIP switch settings
                OldData = DataRead;
            }
        }
    }
```

2. Save and close the file. Verify that the code compiled without errors
3. The system is ready to be downloaded to the board.

Implementing SW drivers for custom peripheral

Software driver is a user-friendly interface between hardware component and software application, instead of operating on the memory mapped hardware component's registers directly, you call certain user-friendly driver functions in your software application to control and use the hardware component. It's not required to write a software driver for every hardware component, especially those simple hardware like our *my_peripheral* custom peripheral, but it's always a good approach to supply each hardware component with a software driver thus to hide the hardware complexity from end user.

So far we've completed the hardware system specification including our custom peripheral *my_peripheral*, in this section we will show you how to implement a very simple software driver for our *my_peripheral* custom peripheral. You will notice that there's a drivers directory (stores software drivers) created under our lab project (parallel to the pcores directory which stores hardware peripherals). A software driver *my_peripheral_v1_00_a* is already created below drivers by the wizard as we indicated in the **Create and Import Peripheral wizard** - create mode that we want the wizard to generate the software driver template for us (Figure 34). This auto generated driver contains the correct directory structure and interface files (.mdd/.tcl) that make it recognizable by XPS, as well as the driver header/source template files. Open the **my_peripheral.h** and **my_peripheral**.c files under ..\drivers\my_peripheral_v1_00_a\src directory and take a while to read the software driver header/source template that the wizard generated. In the header file, some constants are defined for your convenience to access applicable registers, two simple register peek/poke macros are defined, as well as some useful macros and a default interrupt handler declaration with its simple implementation in the corresponding source file. It's totally up to you as how friendly you want the driver to be.

You may also want to take a look at the self-test function that the wizard generated for you. It's defined in the **my_peripheral_selftest.c** file. Depending on the features you selected in the wizard, this self-test function tries to access each individual feature of the hardware component sequentially, thus to verify that the hardware component functioning properly.

As an example, we will define some simple driver functions in this section to clear display in convenient way. And include defined driver in our test application.

1. In..\drivers\my_peripheral_v1_00_a\src\my_peripheral.h header file, add the following function declarations below the self-test function my_peripheral_SelfTest as shown in Figure 40.

   ```
   void MY_PERIPHERAL_ClearScreen();
   ```

2. In ..\drivers\my_peripheral_v1_00_a\src\my_peripheral.c source file, add the following function definitions at the bottom of the file as shown in figure 10-2.

   ```
   void MY_PERIPHERAL_ClearScreen(){
       xil_printf("%c[2J",27);
   }
   ```

3. In open SDK, run Xilinx Tools/Repositories.

4. At filed Local Repositories Click at New....

5. Browse to the current project path where the CIP Wizard generate driver template Figure 41. Click OK.

6. In Project Explorer, right click on **Hello_world_bsp** Click Board Support Package Settings.

7. In Driver Table for component *my_peripheral* instead Generic pick *my_peripheral* Driver Figure 42.

8. Now, driver for custom peripheral is available.

9. In main file hello_world.c include driver header file and make driver function call.

```
#include "my_peripheral.h"
...
//xil_printf("%c[2J",27);
MY_PERIPHERAL_ClearScreen();
```

10. Run Project, Clean.

11. Known Issue: In generated template for drive in file my_peripheral_selftest.c is made mistake. Instead TEST_AXI_LITE_USER_NUM_REG (defined in header file *my_peripheral.h)* is written MY_PERIPHERAL_USER_NUM_REG. You should change it .
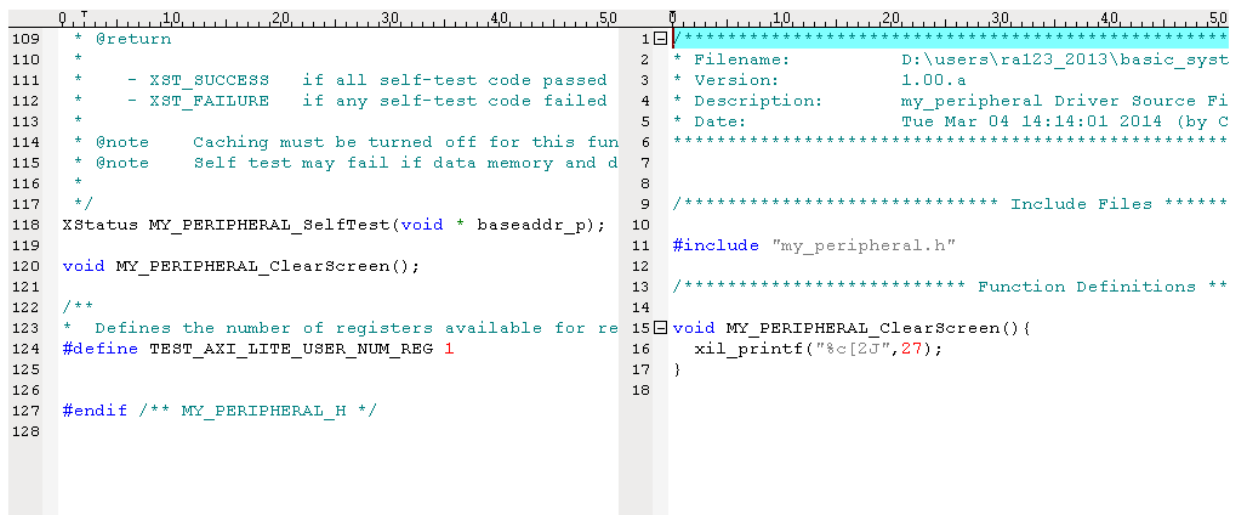
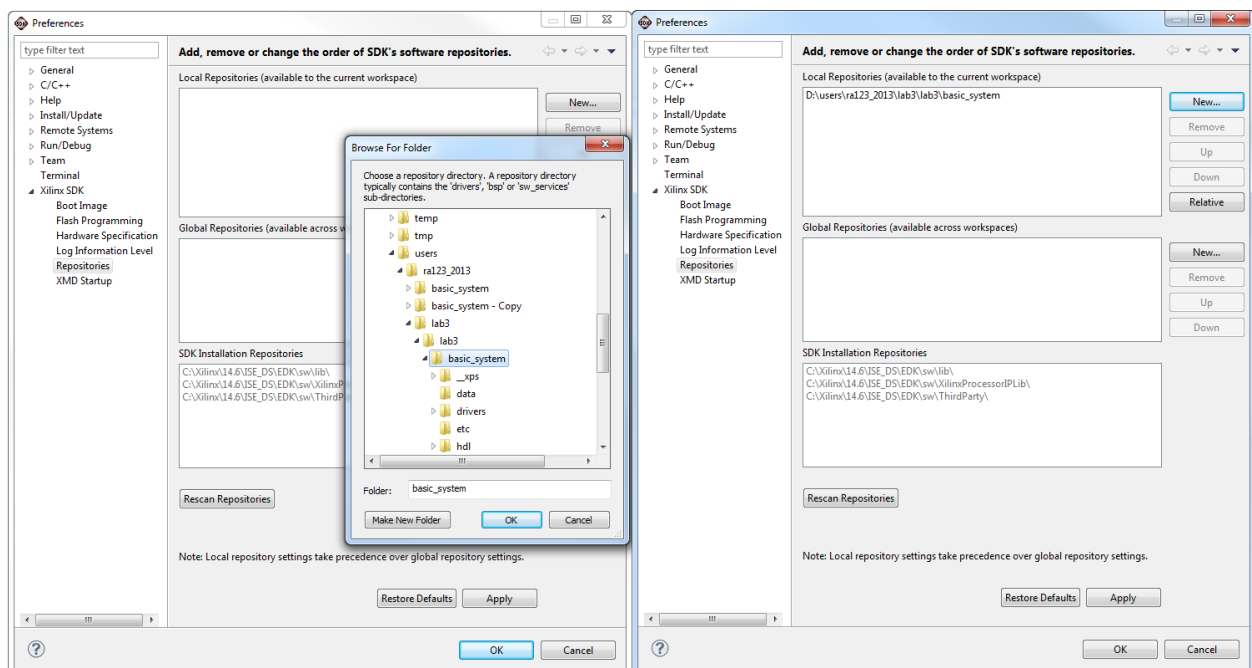12. Run example.

Figure 40: Add Driver Function
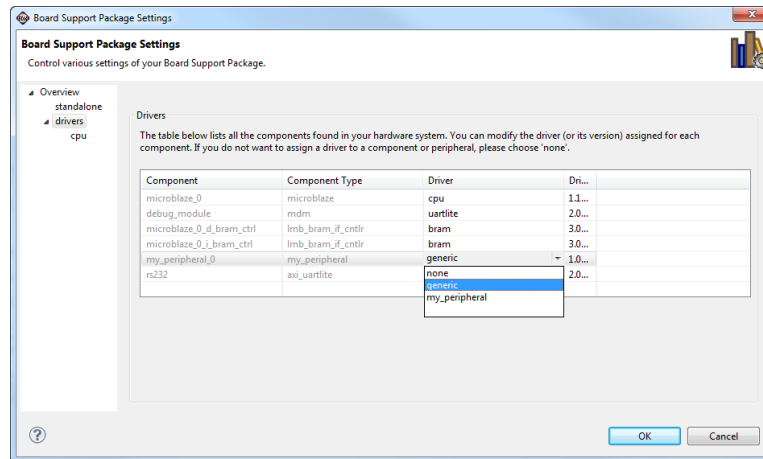
Figure 41: SDK, Add SW Repository Path

Figure 42: SDK, Pick SW Driver for Custom Peripheral

Using and Creating Interrupt-Based Systems

Interrupts are automatic control transfers that occur as a result of an exception. An interrupt occurs when the processor suspends execution of a program after detecting an exception. The processor saves the suspended-program machine state and a return address into the suspended program. This information is stored in a pair of special registers, called save/restore registers. A predefined machine state is loaded by the processor, which transfers control to an interrupt handler. An interrupt handler is a system-software routine that responds to the interrupt, often by correcting the condition causing the exception. System software places interrupt handlers at predefined addresses in physical memory and the interrupt mechanism automatically transfers control to the appropriate handler based on the exception condition. Throughout this Application Note, the term exception and interrupts can be used interchangeably

Create the new custom peripheral *my_timer*

As it described in previous chapters, create simple peripheral *my_timer* with 4 accessible registers. Later, we will add a timer to peripheral template and connect it up with the registers that the Peripheral Wizard created for us. Peripheral registers (described in Table 1) will be used to control timer as described on Figure 43.
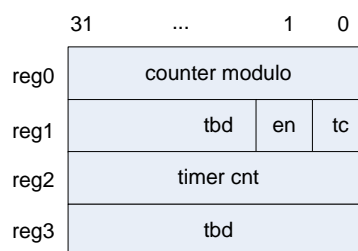

Figure 43: my_timer register map

Table 1

name	Read/write	Description
counter modulo	r/w	value to count to
en	r/w	timer count enable

tc	r	terminal count used as interrupt source
timer_cnt	r	current timer counter value

To edit generated template, select from the menu **File**-**Open** and look in the project folder. Open the folder:

../pcores/my_peripheral_v1_00_a/hdl/vhdl

This folder contains two source files that describe our peripheral **my_ timer.vhd** and **user_logic.vhd**. The first file is the main part of the peripheral and it implements the interface to the AXI. The second file is where we place our custom logic to make the peripheral do what we need it to do. This part is instantiated by the first file.

1. Open the file **user_logic.vhd.** We will need to modify this source code to include our timer code. Find the line of code that says –USER signal declarations added here and add the following lines of code just below:

    ```
    -- Timer signals and components
      signal timer_count : std_logic_vector(C_SLV_DWIDTH-1 downto 0);
      signal timer_count_tc : std_logic;
      signal timer_count_en : std_logic;
    ```

2. Find the line of code that says –USER logic implementation added here and add the following lines of code just below:

    ```
    -- timer counter
    process (Bus2IP_Clk, Bus2IP_Resetn)
    begin
      if Bus2IP_Resetn = '0' then
        timer_count <= (others => '0');
      elsif rising_edge(Bus2IP_Clk) then
        if (timer_count_en = '1') then
          if (timer_count_tc = '1')  then
            timer_count <= (others => '0');
          else
            timer_count <= timer_count + 1;
          end if;
        end if;
      end if;
    end process;
    timer_count_en <= slv_reg1(1);
    timer_count_tc <= '1' when (timer_count >= (slv_reg0 - 1))  else '0';

    process( Bus2IP_Clk ) is
    begin
      if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
        if Bus2IP_Resetn = '0' then
          my_timer_irq <= '0';
        else
          my_timer_irq <= timer_count_tc;
        end if;
      end if;
    end process;
    ```

3. Find the line of code that says SLAVE_REG_READ_PROC : process and replace it with the three lines of code below. Now, when the timer control register is read from, bit 0 will represent the timer_expired signal.

    ```
    -- implement slave model software accessible register(s) read mux
      SLAVE_REG_READ_PROC  :  process( slv_reg_read_sel,  slv_reg0,  slv_reg1,  slv_reg2,  slv_reg3,
    timer_count, timer_count_tc, timer_count_en) is
      begin
        case slv_reg_read_sel is
          when "1000" => slv_ip2bus_data <= slv_reg0;
          when "0100" => slv_ip2bus_data <= slv_reg1(31 downto 2) & timer_count_en & timer_count_tc;
          when "0010" => slv_ip2bus_data <= timer_count;
          when "0001" => slv_ip2bus_data <= slv_reg3;
          when others => slv_ip2bus_data <= (others => '0');
    ```

```
    end case;
  end process SLAVE_REG_READ_PROC;
```

4. Save and close the file.

5. Modify *user_logic.vhd, my_timer.vhd* file and *my_timer_v2_1_0.mpd* file to add new port which will be use as interrupt source (as described in previous chapters). When you modify *.mpd file add code below to indentify port as interrupt source.

```
    PORT o_system_start_irq = "", DIR = O, SIGIS = INTERRUPT, SENSITIVITY = EDGE_RISING
```

6. Run Project/Rescan User Repositories.

7. Add Peripheral to the system like on Figure 36 for my_peripheral.

Create an Instance of the Interrupt Controller

Now we need to include an interrupt controller into our design to detect the interrupt requests from the *my_timer* peripheral and relay them to the MicroBlaze. We could of course, not use the interrupt controller and directly connect the *my_timer* interrupt request signal (IRQ) to the MicroBlaze. The benefit of using the interrupt controller is that we are able to manage multiple interrupt sources if we wish to do so in the future.

1. From the **IP Catalog** find the **AXI Interrupt Controller** IP core in the **Clock, Reset and Interrupt** group. Right click on the core and select **Add IP**.

2. Wizard will as you: *Do you want to add...* Click **Yes**.

3. Click on the **Addresses** tab from **System Assembly View**. Change the **Size** for *axi_intc_0* to 64K. Then click **Generate Addresses**(Figure 44)**.**

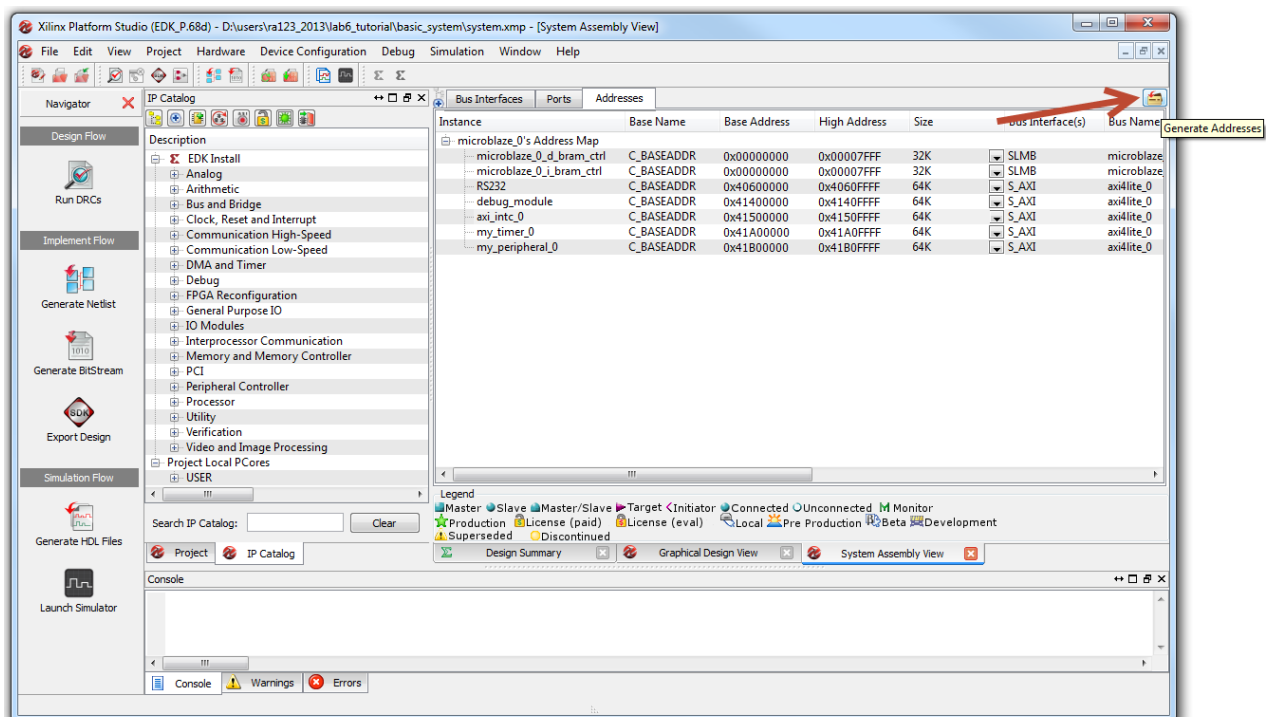Now we have created an instance of the Interrupt Controller peripheral in our design.

Figure 44: Generate Address

Connect the Interrupt Request (IRQ) signals

We now need to connect the **my_timer_0** IRQ signal to the interrupt controller, and then connect the interrupt controller IRQ signal to the MicroBlaze.

1. Click on the **Ports** filter. Click on the "+" for **my_timer_0** to view its ports. Click on the **Connected Port** field for the *my_timer_irq* port. Dialog on Figure 45 will show.

2. Select **Instances Name**: **my_timer_0** and **Port Name** *my_timer_irq* and then click on **Add to Connected Interrupts like on** Figure 45**.**
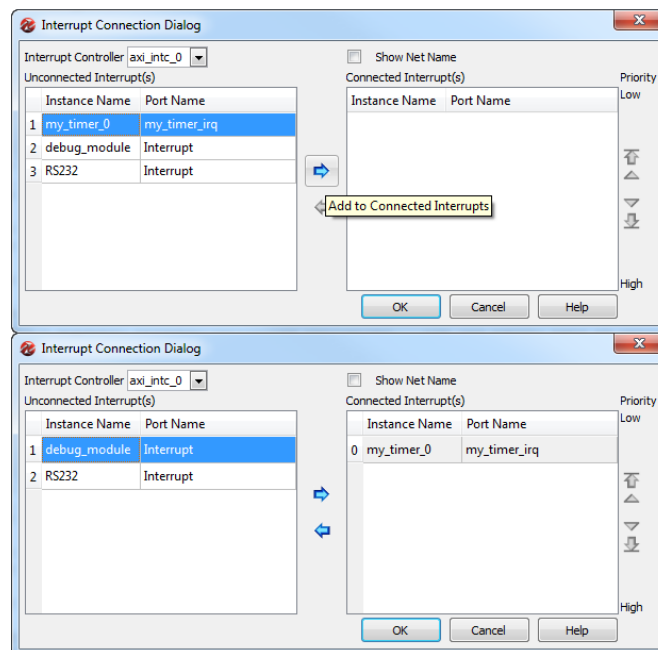

Figure 45: Interrupt Connection Dialog

3. Click OK. Now *my_timer_irq* from **my_timer_0** is connected to the **axi_intc_0** port *Intr*.

4. Go back to the Bus Interface. Click on "+" for **microblaze_0** and **axi_intc_0** instances. You can see that Port from **axi_int_0** INTTERRUPT is not connected. You can connect it to the **microbmaze_0** port INTERUTPT by clicking on Red arrow. This connects the interrupt controller IRQ signal to the IRQ input of the MicroBlaze.

5. Now you can Implement design, then Export Design to SDK, and ran SDK to write simple Interrupt test software test like in previous chapters.

Now we have connected the **my_timer_0** IRQ signal to the interrupt controller, and the interrupt controller's IRQ signal to the IRQ input of the MicroBlaze. This completes our hardware connections for interrupt capability.
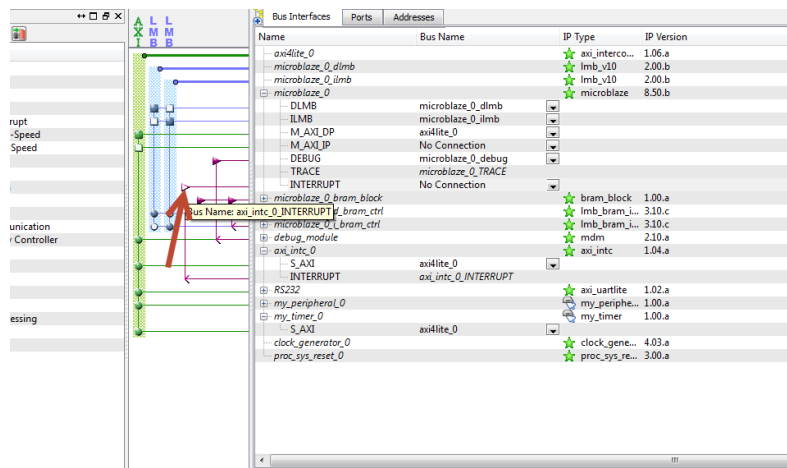
Figure 46: Connecting Interrupt

Modify the Software Application

The software application will contain the main program and the interrupt handler function to be called when an interrupt occurs. It must create the interrupt vector table which tells the MicroBlaze what code to run when an interrupt is generated. It also must enable the Timer peripheral to generate interrupts, and the MicroBlaze to respond to interrupts.

1. Run SDK.
2. Create new Application Project for interrupt test..
3. Replace all the code in this file with the following source and save the file.

```c
#include <stdio.h>
#include "platform.h"
#include "xintc.h"
#include "xparameters.h"
#include "xio.h"
#include "xil_exception.h"

XIntc Intc;

void my_timer_interrupt_handler(void * baseaddr_p) {
        xil_printf("\n\rtimer timeout.");
}

int main()
{
    init_platform();
    XStatus Status;
    Xuint32 value1, value2, value3;

    xil_printf("Interrupt example\n\r");

    //Set Terminal count for my_timer
    XIo_Out32(XPAR_MY_TIMER_0_BASEADDR + 0x0, 0x5F5E100);

    // Run my_timer
    XIo_Out32(XPAR_MY_TIMER_0_BASEADDR + 0x4, 0x2);

    // Read my_timer register to verify that TC value is written
    value1 = XIo_In32(XPAR_MY_TIMER_0_BASEADDR + 0x0);
    xil_printf("\n\rvalue1 = %x.", value1);

    //initialize interrupt controller
    Status = XIntc_Initialize (&Intc, XPAR_INTC_0_DEVICE_ID);
    if (Status != XST_SUCCESS) xil_printf ("\r\nInterrupt controller initialization failure");
    else xil_printf("\r\nInterrupt controller initialized");

    // Connect my_timer_interrupt_handler
    Status = XIntc_Connect (&Intc, XPAR_INTC_0_MY_TIMER_0_VEC_ID,
                        (XInterruptHandler) my_timer_interrupt_handler,
```

```
                          (void *)0);

        if (Status != XST_SUCCESS) xil_printf ("\r\nRegistering MY_TIMER Interrupt Failed");
        else xil_printf("\r\nMY_TIMER Interrupt registered");

        //start the interrupt controller in real mode
        Status = XIntc_Start(&Intc, XIN_REAL_MODE);

        //enable interrupt controller
        XIntc_Enable  (&Intc, XPAR_INTC_0_MY_TIMER_0_VEC_ID);

        microblaze_enable_interrupts();

        while (1){
            /*value3 = XIo_In32(XPAR_MY_TIMER_0_BASEADDR + 0x0);
            value1 = XIo_In32(XPAR_MY_TIMER_0_BASEADDR + 0x4);
            value2 = XIo_In32(XPAR_MY_TIMER_0_BASEADDR + 0x8);
             xil_printf("\n\rvalue1 = %x, value2 = %x, value3 = %x.", value1, value2, value3);*/
        }
        cleanup_platform();

        return 0;
}
```

4. Save and close the file.
5. Generate a new linker script by right clicking on project and clicking "Generate Linker Script". Click "OK".
6. Run test application.

Literature

[1] http://www.fpgadeveloper.com/2011/06/creating-a-project-using-the-base-system-builder.html

[2] http://www.fpgadeveloper.com/2011/07/read-dip-switches-from-a-microblaze-application.html

[3] http://www.fpgadeveloper.com/2014/02/creating-a-project-using-the-base-system-builder-2.html

[4] http://www.fpgadeveloper.com/2014/02/create-an-application-using-the-sdk.html

[5] http://www.fpgadeveloper.com/2008/10/timer-with-interrupts.html

[6] http://forums.xilinx.com/xlnx/attachments/xlnx/EDK/22489/1/xapp778_ver3_4.pdf

[7] http://www.xilinx.com/support/documentation/ip_documentation/axi_intc/v1_04_a/ds747_axi_intc.pdf

[8] http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_6/edk_ctt.pdf