Tutorial

Creating a Custom Peripheral and adding it to a Microblaze Embedded System

Introduction

In the previous tutorials you learned how to create embedded design using EDK, and connect it to a personal computer over the standard network interface. In these tutorials you only used peripherals provided by Xilinx in its IP core library. In this tutorial you experiment creating your own IP core, add it to Xilinx library and integrate it into your design. You will be able to add your own VHDL code into the new design to implement the required function of your projects. The new peripheral will be connected to the Microblaze system using the PLB bus.

Pre-requisites

1. Complete Tutorial: Building an Embedded Processor System on FPGA.

Objectives:

1. Create an IP core with PLB interface.
2. Connect the new IP core to Microblaze embedded system.
3. Perform communication with the new core.

Equipment and Tools

1. NEXYS 3 Spartan-6 Board.
2. Xilinx Embedded Development Kit (v13.3).
3. Personal Computer with RS232 cable.

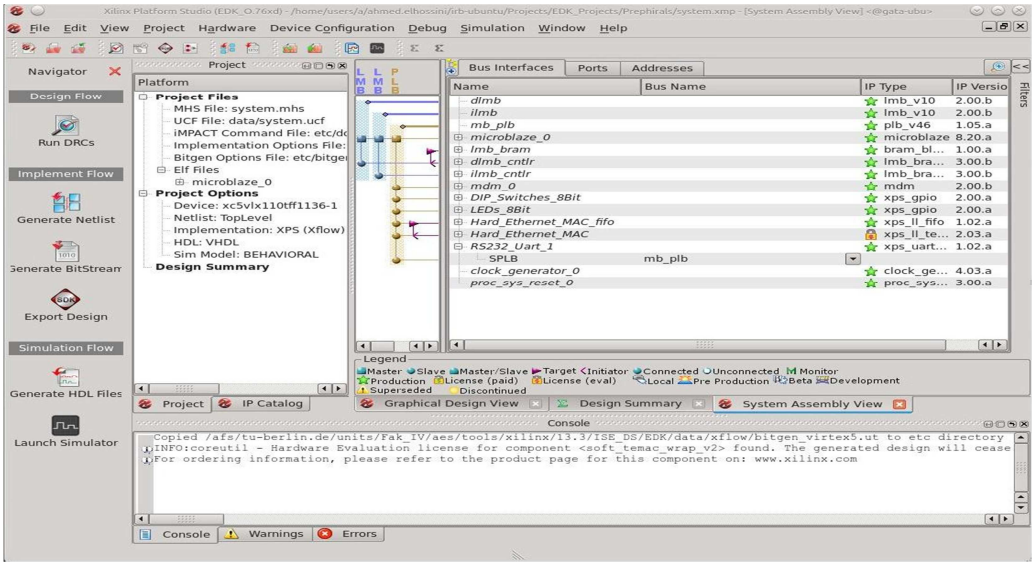

Figure 1: Platform Studio Window

Tutorial Steps

Part 1: Creating a Custom IP Core

In this part of the tutorial we will create a new IP core using EDK to integrate it with the Microblaze embedded system.

Step 1: Starting from the end of tutorial 2: you should end up with the Platform Studio window shown in **Figure 1**. In this window select the menu item "**Hardware-> Create or Import Peripheral...**". This will display the "Create or Import Peripheral Wizard" window shown in **Figure 2**. This wizard will guide you through the required steps to create a new peripheral. Click **"Next"** to continue.

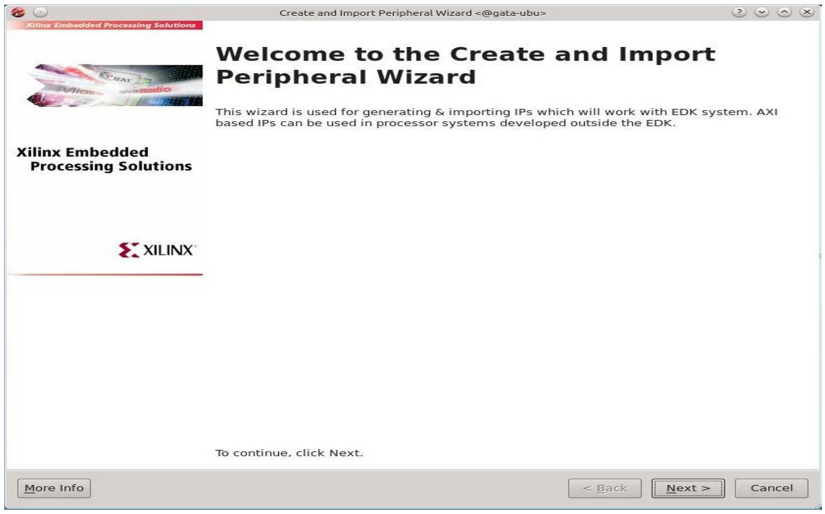

Figure 2: Create and Import Peripheral Wizard (Step 1)

Step 2: The second page of the wizard asks if you want to create a new peripheral or import an already created core. Select **"Create Template for a new Peripheral"** and then press **"Next"** as shown in **Figure 3**. The wizard will guide you with the necessary steps to build the required files to build and connect your new peripheral (both hardware and software).

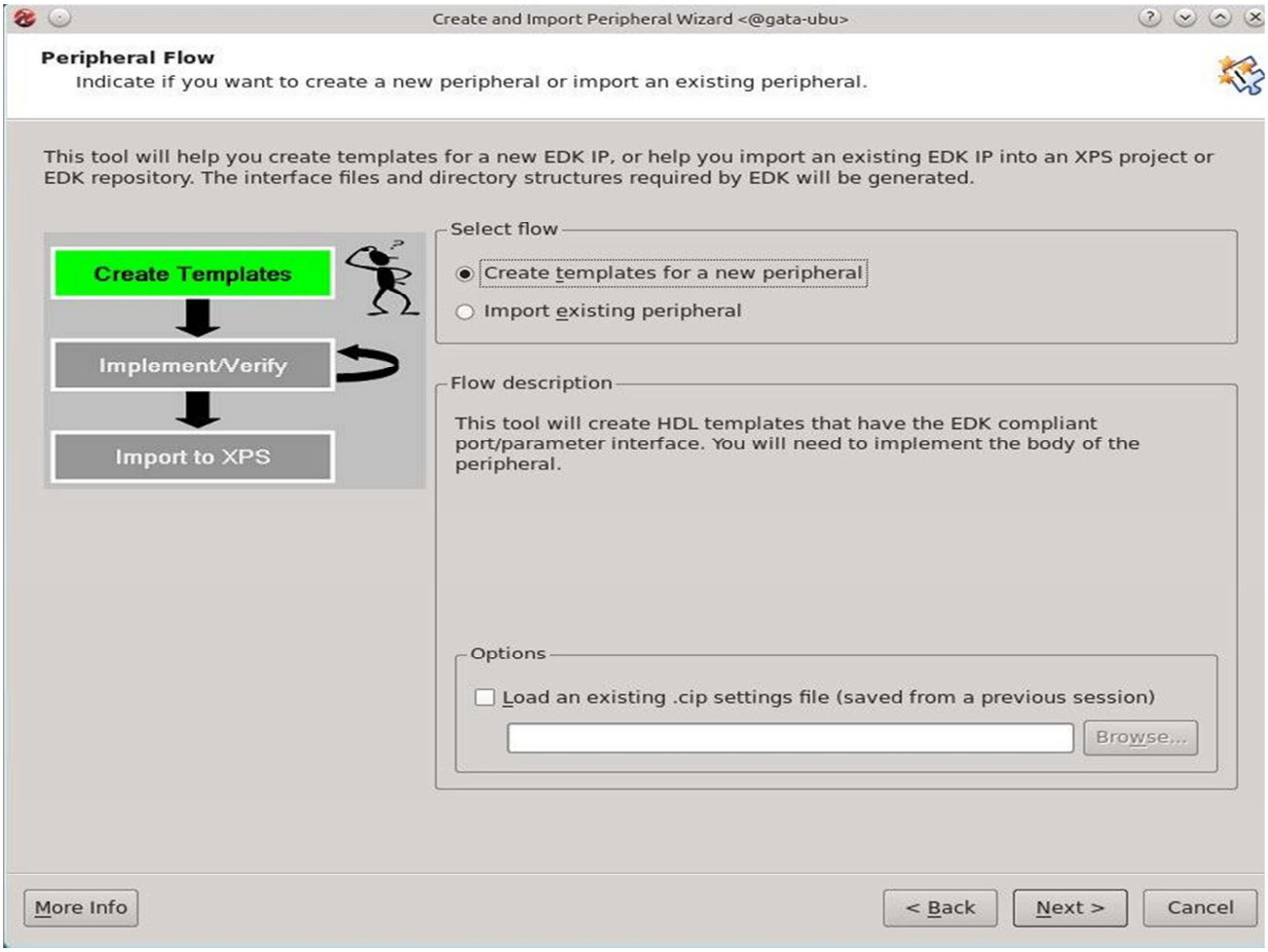

Figure 3: Create and Import Peripheral Wizard (Step 2)

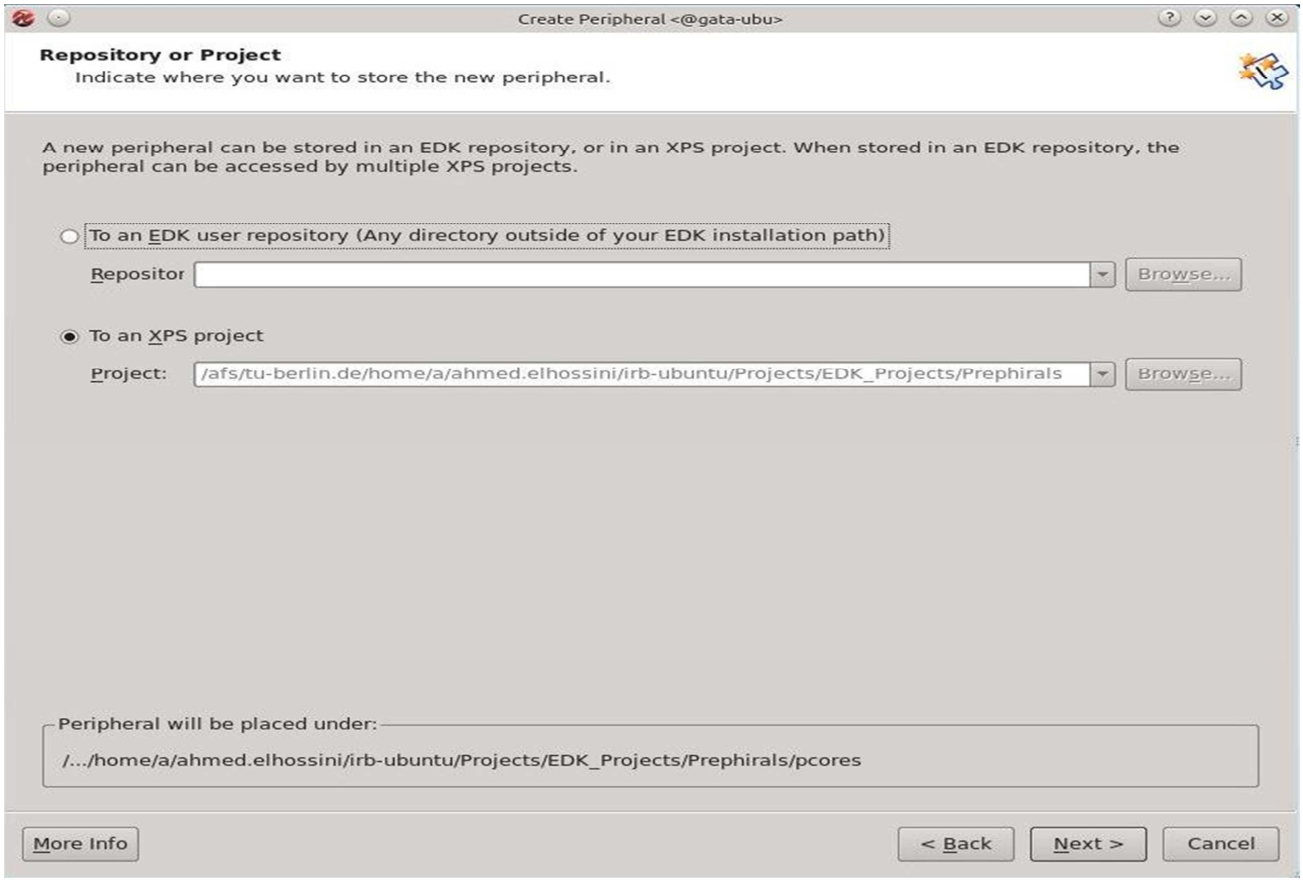

Figure4: Create and Import Peripheral Wizard (Step #3)

Step 3: The following window asks about the location where you want to store the files for your new peripheral. Here you have two options as show in **Figure 4**. The first option stores the core in a global repository so that it will be accessible in any project you create. The second option stores the new peripheral in your project local directory and so it will be accessible only in your project. Since this is just a tutorial select the second option and then press **"Next"**.

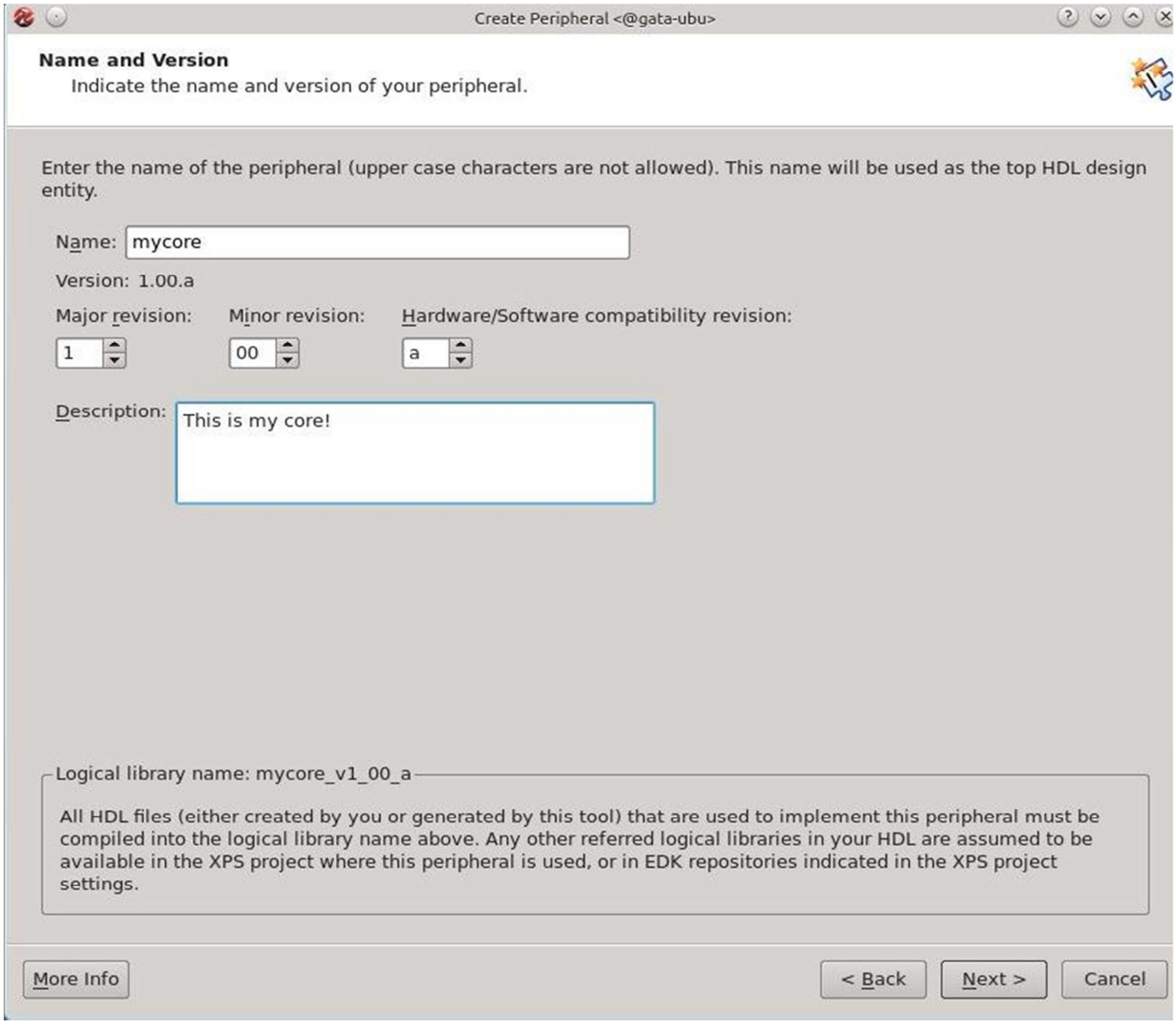

Figure 5: Create and Import Peripheral Wizard (Step 4)

Step 4: In the window shown in **Figure 5** you are required to enter the peripheral name and revision. The wizard allows you to create different revisions of the same core as well as starting from scratch. Note that the core name should consist only of lower case letters and numbers. In this screen you can also add some description to your core. Give your new IP core any name, for example **"mycore"**. Type the name and give description if required then press **"Next"** to continue.

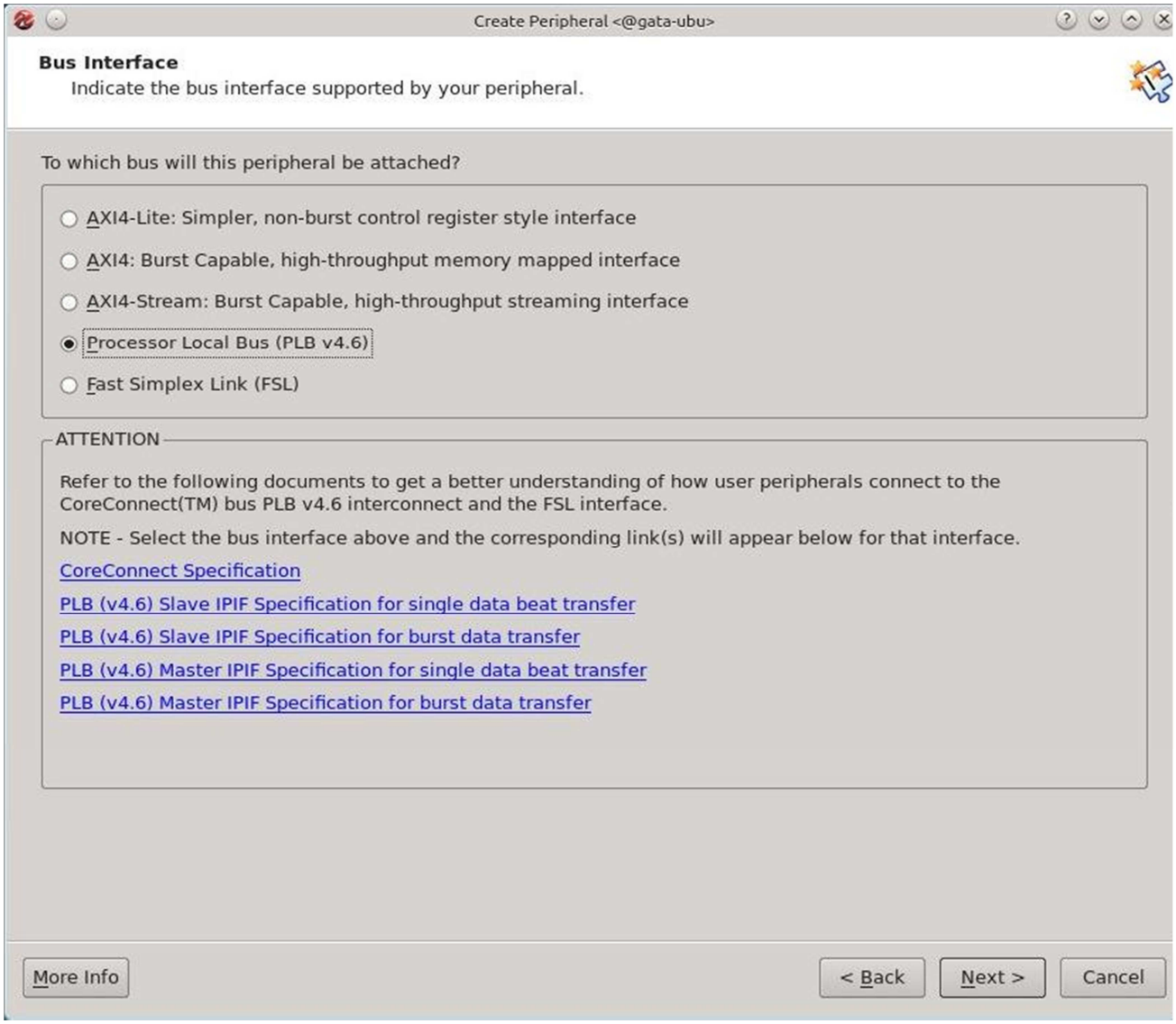

Figure 6: Create and Import Peripheral Wizard (Step 5)

Step 5: In this step you are required to select the interface type for your peripheral. Xilinx EDK supports 3 types of interface standards in its embedded systems. The first and the newest one is the AXI4 architecture which is designed for various types of communication strategies. The second type is the **"Processor Local Bus (PLB 4.0)"** which is a standard bus architecture to connect Microblaze and its peripheral in a shared bus structure. This bus standard will be used in this tutorial. The last interface type is the **"Fast Simplex Link (FSL)"** which is a dedicated FIFO link between the processor and the peripheral. The Microblaze supports a limited number of FSL links but this communication channel guarantees the fastest communication between the peripheral and the processor. Select the **"Processor Local Bus (PLB 4.0)"** as shown in **Figure 6** and then press **"Next"**.

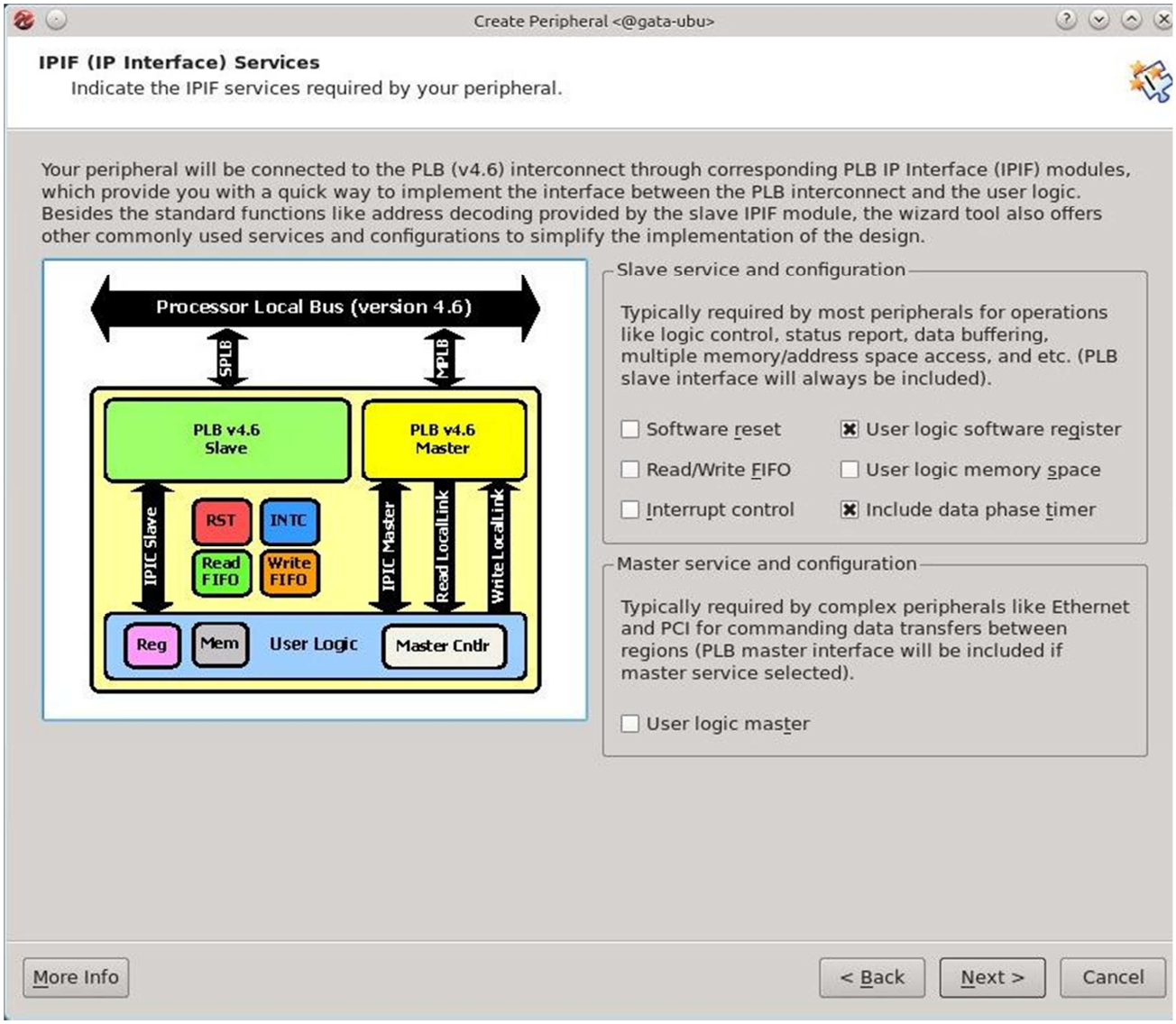

Figure 7: Create and Import Peripheral Wizard (Step 6)

Step 6: The PLB bus is based on a Master/Slave configuration. The bus can have multiple masters and slaves. In the design created in the previous Tutorial, Microblaze acted as the master of the PLB bus while all other peripheral acted as slaves. The master initiates the transactions and controls the data flow, while the slave peripheral waits for a transaction from the master. The master of the bus sends requests to the a specific address space, and the slave peripheral decodes that address and responds to the request if the address is recognized. The address of each peripheral is unique and all transaction are memory mapped. In this step we specify the configuration of the core we are about to create. The wizard will add VHDL logic block for slave, master, or both if required. You have to specify the components in the slave logic and if you want a master logic you can also do so. As you can see in **Figure 7**, you can add interrupt control, software reset, Read/Write FIFO, user registers, and more. Later you can modify these components and add more depending on the function of your peripheral. For the purpose of this tutorial select the options shown in **Figure 7**, then press **"Next"**.

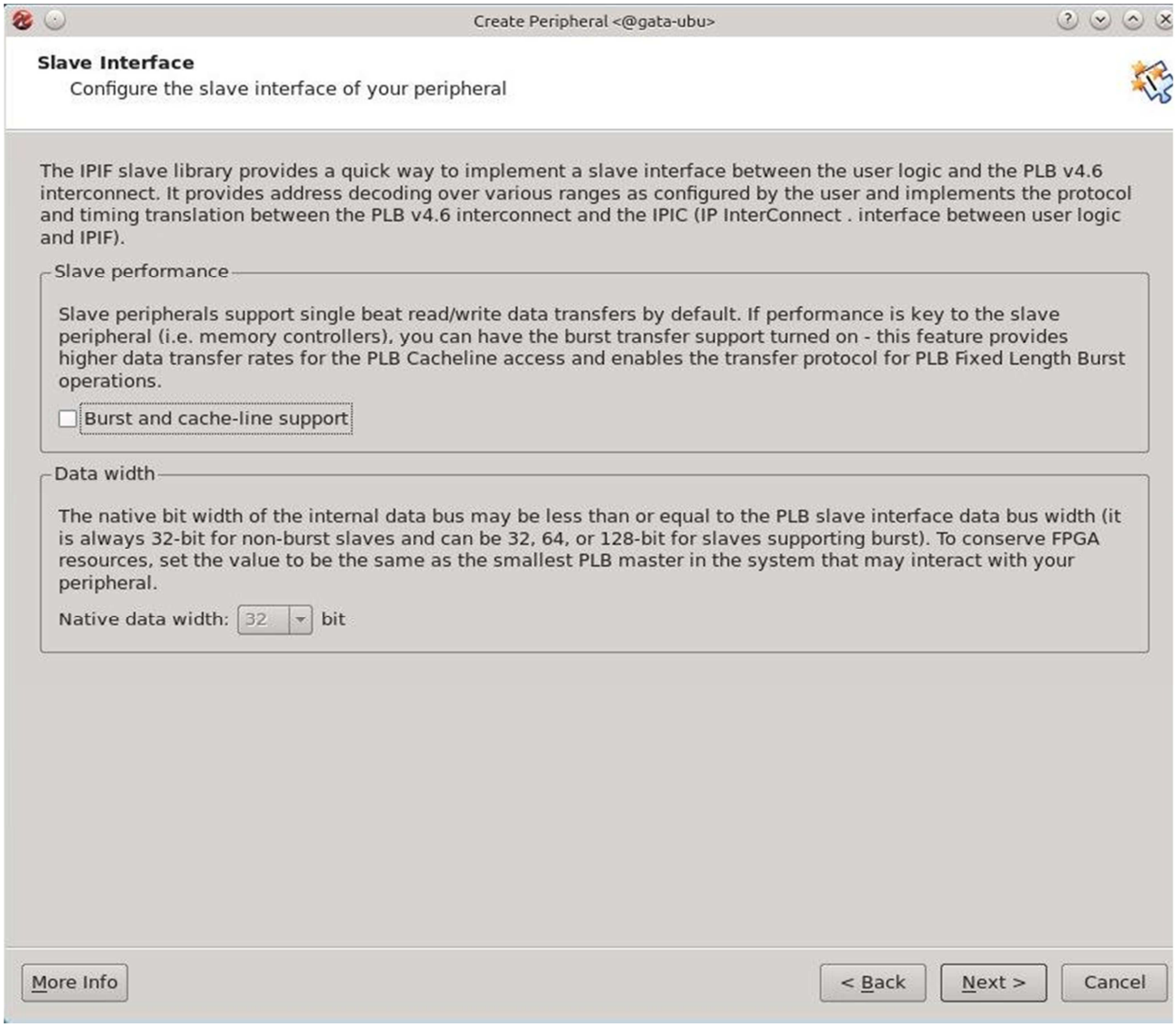

Figure 8: Create and Import Peripheral Wizard (Step 7)

Step 7: The following page of the wizard is shown in **Figure 8**. In this window you have the option to add cache support for your device as well as changing the data bus width. You do not need to do any changes in this screen. Press **"Next"** to continue.

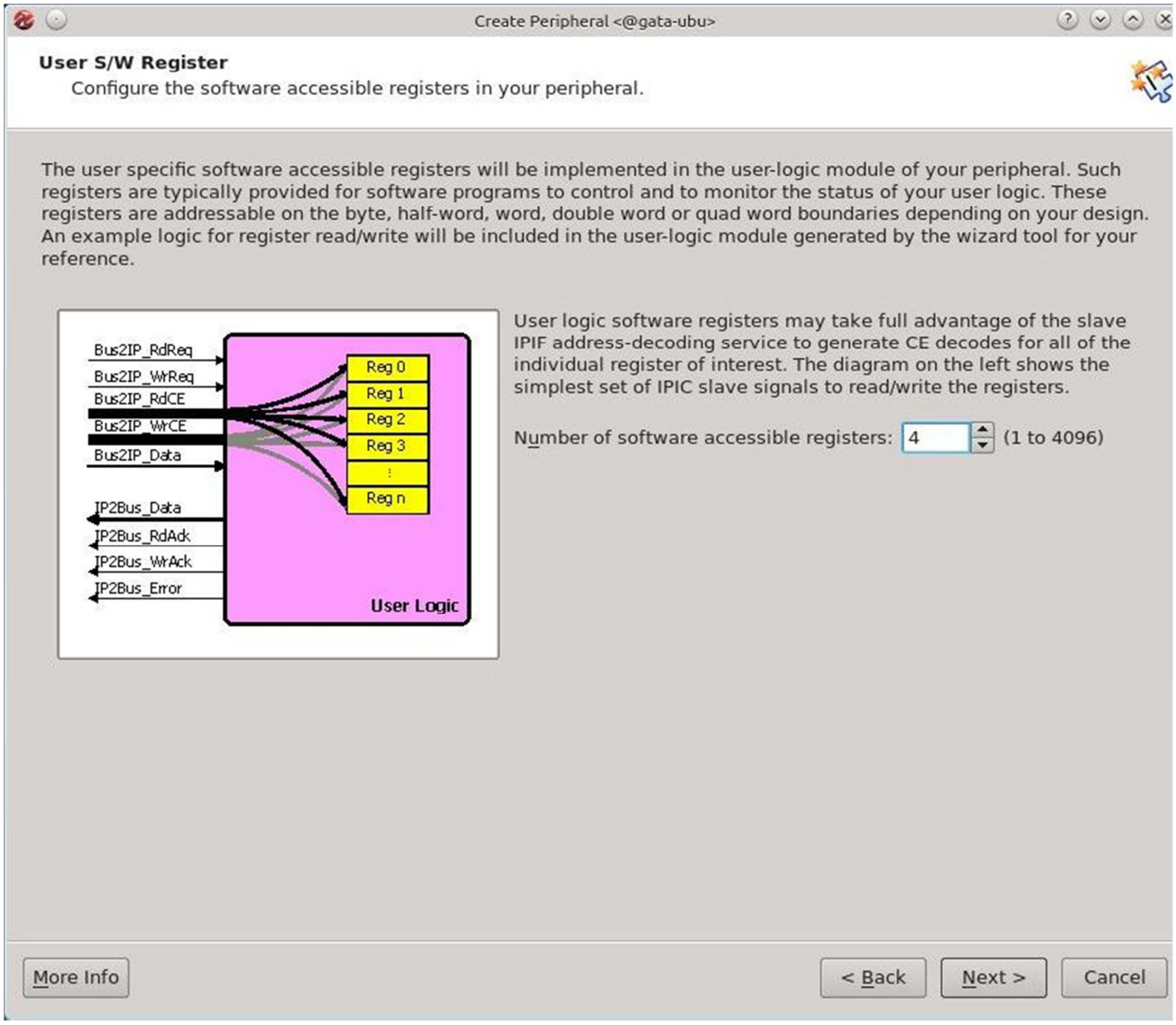

Figure 9: Create and Import Peripheral Wizard (Step 8)

Step 8: In the previous step you configured the new core to have built in software registers. The wizard will create the necessary logic inside your core to have read/write registers. In this step you are asked to specify the number of registers in your design as seen in **Figure 9**. Select ***"4"*** to add for registers to your design and then press **"Next"** to continue.

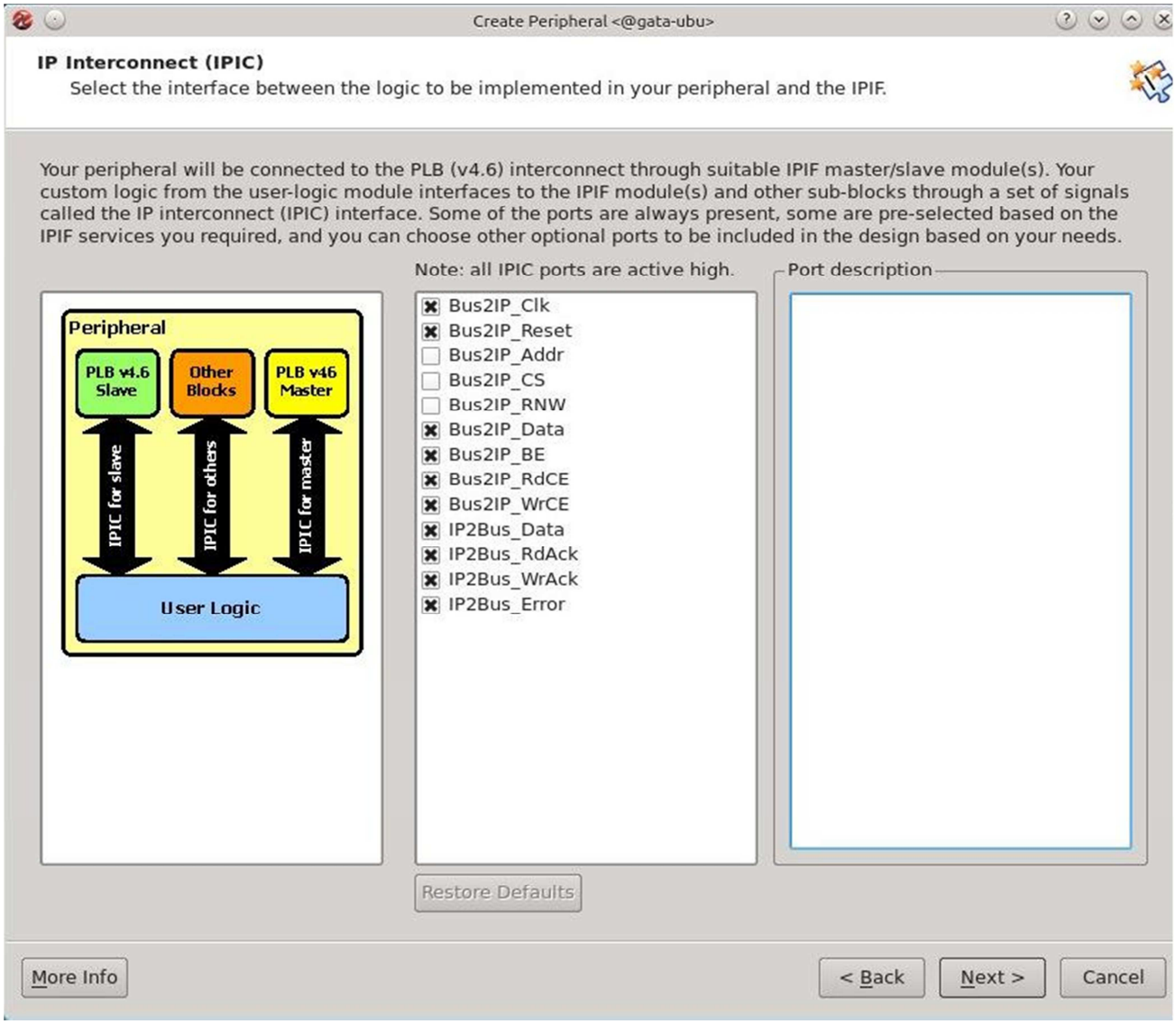

Figure 10: Create and Import Peripheral Wizard (Step 9)

Step 9: After setting the core configuration and internal components, the wizard window shown in **Figure 10** allows you to change the interface between your new added logic (IP) and the bus interface. The wizard will create the bus interface for you and then create a **usr_logic** module that contains all the logic selected by you in the wizard. This screen allows you to change the signals between the bus interface and your user logic. Press **"Next"** to continue.

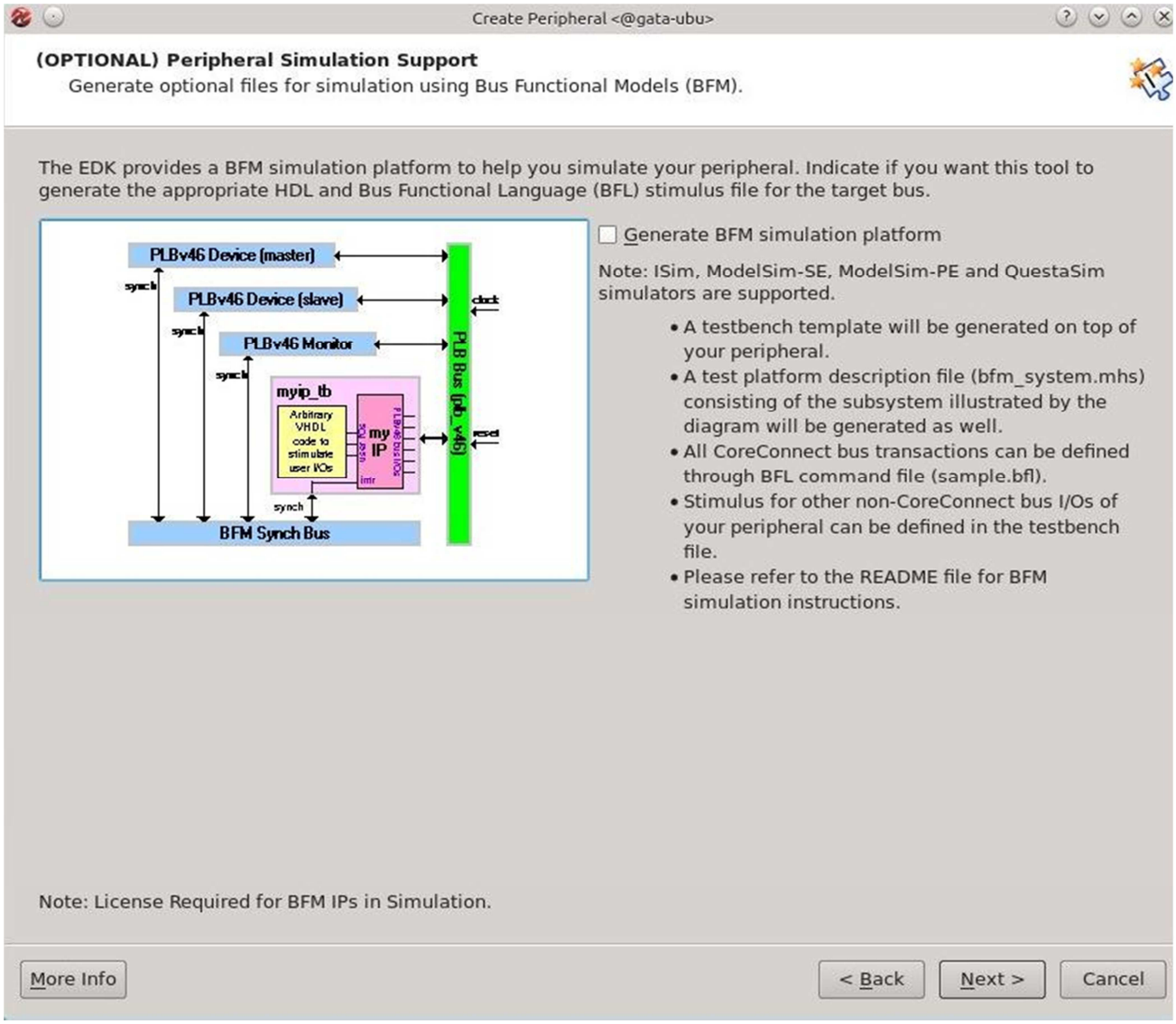

Figure 11: Create and Import Peripheral Wizard (Step 10)

Step 10: EDK designs can be simulated for functional and timing verification. In order to perform simulation, extra files will be needed and added for the simulation of the bus components. If you plan to perform simulations for your design then select **"Generate BFM simulation platform"** in the wizard page shown in **Figure 11**.

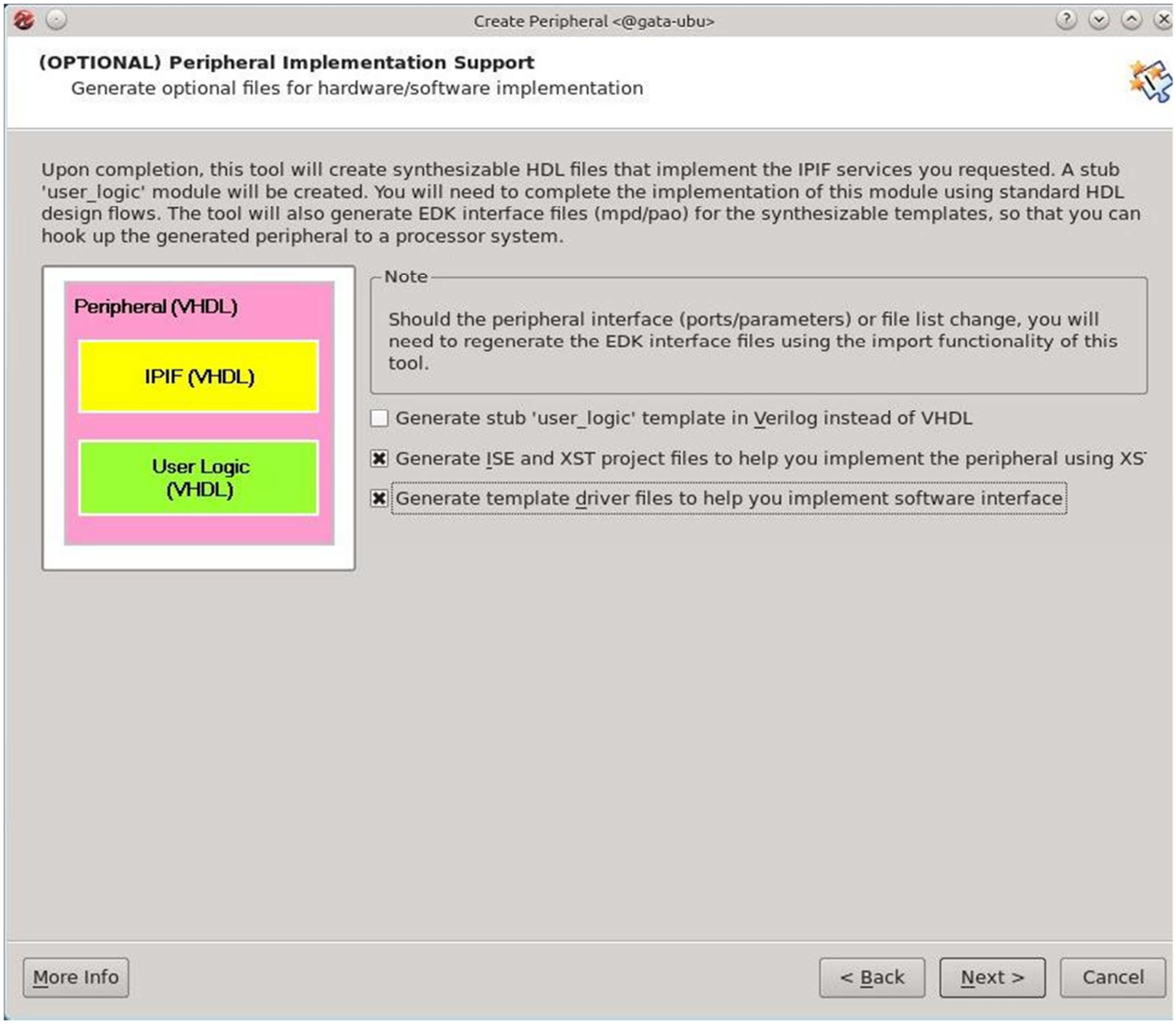

Figure 12: Create and Import Peripheral Wizard (Step 11)

Step 11: The final step of the wizard contains three final options:

1. The option to generate the user_logic module in Verilog rather than VHDL.
This option is important when you prefer Verilog for hardware development.
2. The option to generate an ISE project to help in the implementation of your core.
The generated ISE project can be used to verify the core before integration in the EDK.
3. The final option is to create a software driver for core to be used during software development.

Perform the selection as shown in **Figure 12** (unless you want to write in Verilog) and then press **"Next"** to continue.

The wizard steps are now completed and the template for your design is created.

Part 2: Modifying the generated IP core

In Part 1 of this tutorial, the wizard created all necessary files required to integrate your core into an EDK design. These files (as specified in step 3 of the wizard) are located in the local directory of your projects as shown in **Figure 13**. The files created are stored in two main directories **"pcores"** and **"drivers"**. The first includes all the files for the hardware build while the second contains the software drivers created by the wizard. In both directories you will find another directory for your core with the name **"mycore_v1_00_a"**.

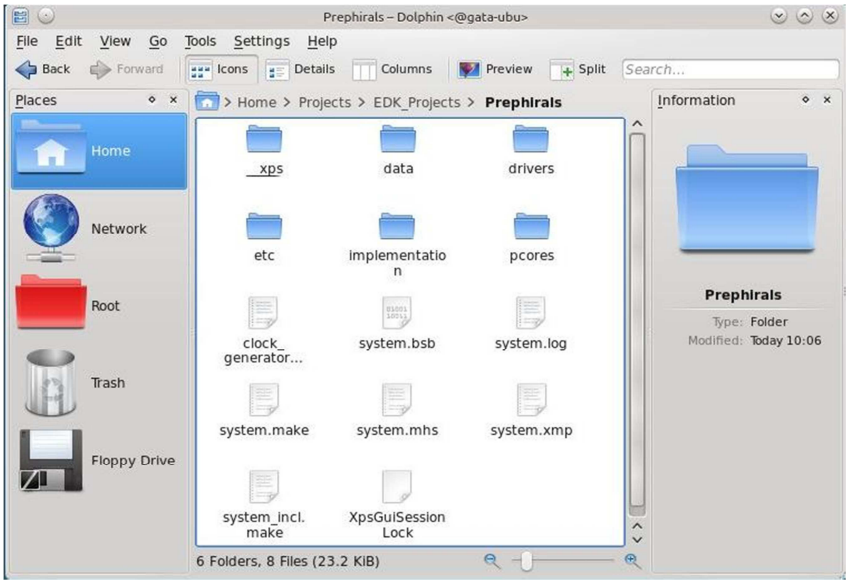

Figure 13: Project Directory Listing

Open the folder **“pcores->mycore_v1_00_a”** it should be as shown in **Figure 14**. Three directories are found:

- **“data”**: this directory contains the files required to define the core to the EDK. The EDK reads the files inside this directory to identify the bus connections of the core, and the hardware files required to build the core (library deceleration).
- **“devl”**: this directory contains the files for the ISE project.
- **“hdl”**: this directory contains the HDL files for the core.

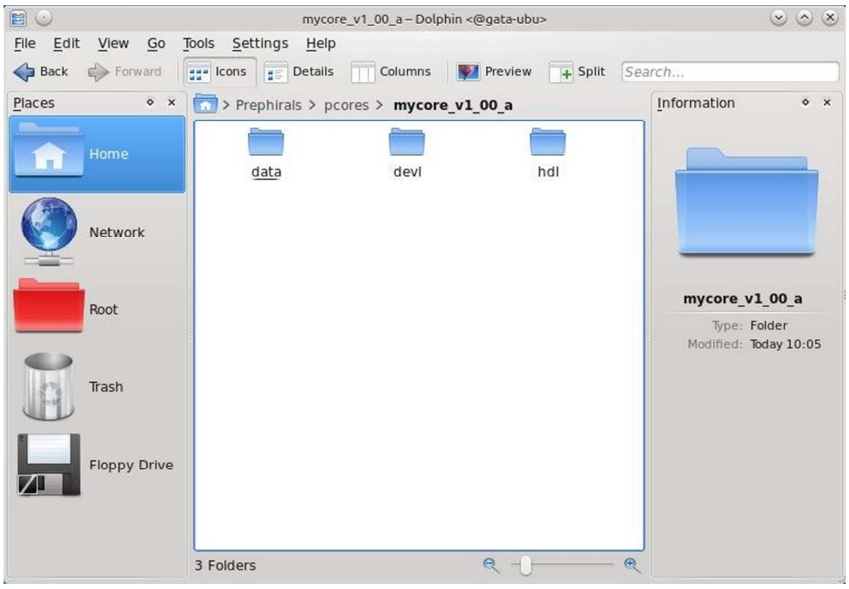

Figure 14: "pcores->mycore_v1_00_a" directory listing

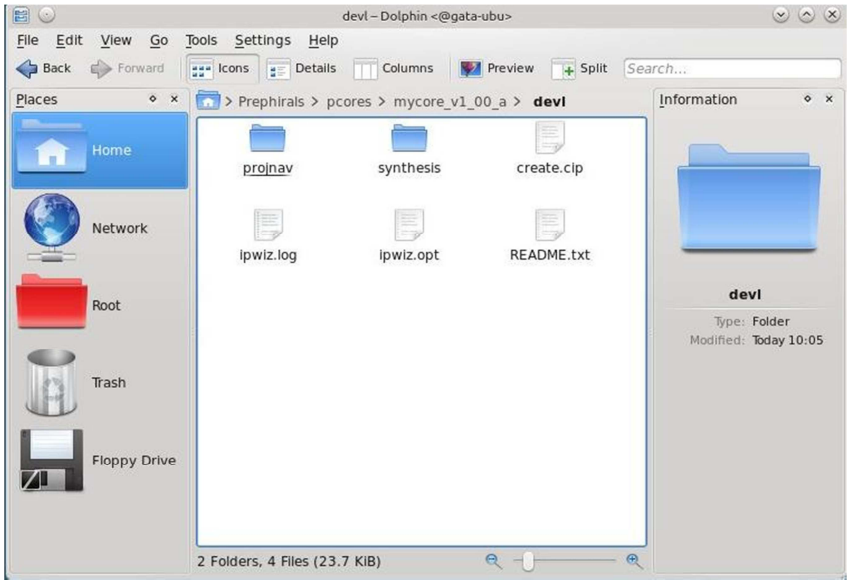

Figure 15: "pcores->mycore_v1_00_a->devl" directory listing

Open the folder **"devl"** as shown in **Figure 15**. The directory **"projnav"** contains the files of the ISE project navigator to edit the source files of your core.

Open that directory as shown in **Figure 16**.

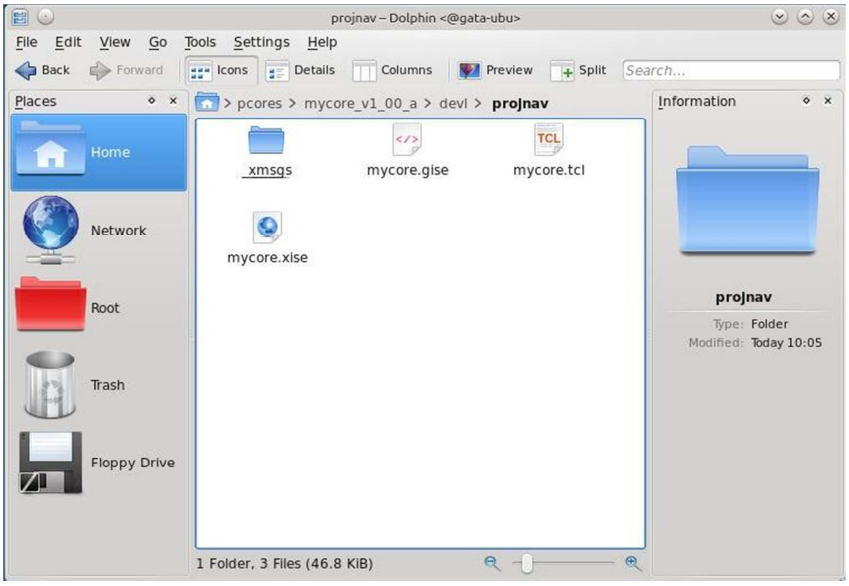

Figure 16: "pcores->mycore_v1_00_a->devl->projnav"

Here you will find the file **"mycore.xise"** which is the ISE project navigator file.
You need to open this file using **"ISE"**.

Start ISE and the The ISE project navigator window will open. Select **"File->Open Project"** and select the file **"mycore.xise"**.

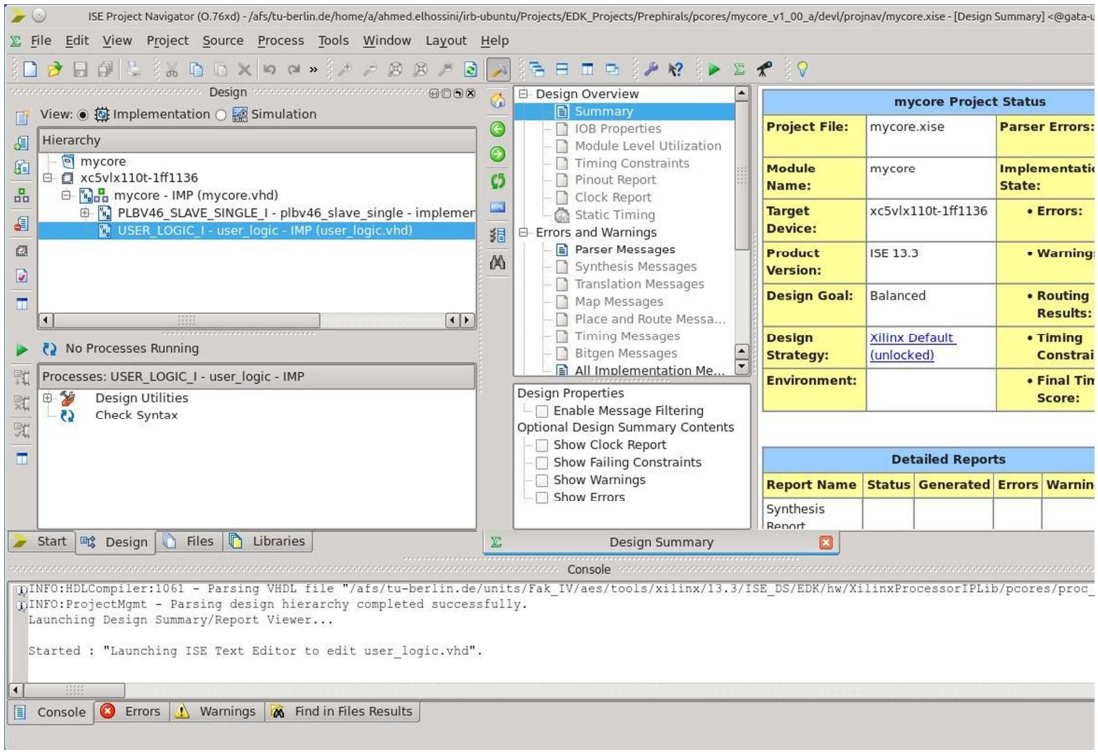

Figure 18: ISE project for "mycore"

xxxxxx

This will open the ISE navigator for the "mycore" project as shown in **Figure 18**.

Open the file **"user_logic.vhd"** and inspect it. The wizard created an entity called **"user_logic"**. The declaration for that entity is shown in **Figure 19**.

```
84  entity user_logic is
85    generic
86    (
87      -- ADD USER GENERICS BELOW THIS LINE ---------------
88      --USER generics added here
89      -- ADD USER GENERICS ABOVE THIS LINE ---------------
90
91      -- DO NOT EDIT BELOW THIS LINE ----------------------
92      -- Bus protocol parameters, do not add to or delete
93      C_SLV_DWIDTH                    : integer              := 32;
94      C_NUM_REG                       : integer              := 4
95      -- DO NOT EDIT ABOVE THIS LINE ----------------------
96    );
97    port
98    (
99      -- ADD USER PORTS BELOW THIS LINE ------------------
100     --USER ports added here
101     -- ADD USER PORTS ABOVE THIS LINE ------------------
102
103     -- DO NOT EDIT BELOW THIS LINE ----------------------
104     -- Bus protocol ports, do not add to or delete
105     Bus2IP_Clk                      : in  std_logic;
106     Bus2IP_Reset                    : in  std_logic;
107     Bus2IP_Data                     : in  std_logic_vector(0 to C_SLV_DWIDTH-1);
108     Bus2IP_BE                       : in  std_logic_vector(0 to C_SLV_DWIDTH/8-1);
109     Bus2IP_RdCE                     : in  std_logic_vector(0 to C_NUM_REG-1);
110     Bus2IP_WrCE                     : in  std_logic_vector(0 to C_NUM_REG-1);
111     IP2Bus_Data                     : out std_logic_vector(0 to C_SLV_DWIDTH-1);
112     IP2Bus_RdAck                    : out std_logic;
113     IP2Bus_WrAck                    : out std_logic;
114     IP2Bus_Error                    : out std_logic
115     -- DO NOT EDIT ABOVE THIS LINE ----------------------
116   );
117
118   attribute MAX_FANOUT : string;
119   attribute SIGIS : string;
120
121   attribute SIGIS of Bus2IP_Clk    : signal is "CLK";
122   attribute SIGIS of Bus2IP_Reset  : signal is "RST";
123
124 end entity user_logic;
```

Figure 19: Entity declaration for "user_logic" component

The entity declarations contains the required ports for data and control signals. The signals **"Bus2IP_Data"** and **"IP2Bus_Data"** are 32-bits data signals that are used to transfer data from/to the core. The signals **"Bus2IP_RdCE"** and **"Bus2IP_WeCE**" are the read enable and write enable signals for each register of the created core. As we selected four registers in the Wizard window, four select signals are included in these control signals. Other signals are used for data acknowledgment and error detection.

The architecture of the **"user_logic"** is composed of several blocks:

1. Signal declaration:
   As shown in **Figure 20**, signals are defined for each of the four registers **"slv_reg0"** to **"slv_reg3"**. Also signals are defined for read/write selection signals, and read/write acknowledgement.

```
134   ---------------------------------------------
135   -- Signals for user logic slave model s/w accessible register example
136   ---------------------------------------------
137   signal slv_reg0                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
138   signal slv_reg1                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
139   signal slv_reg2                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
140   signal slv_reg3                       : std_logic_vector(0 to C_SLV_DWIDTH-1);
141   signal slv_reg_write_sel              : std_logic_vector(0 to 3);
142   signal slv_reg_read_sel               : std_logic_vector(0 to 3);
143   signal slv_ip2bus_data                : std_logic_vector(0 to C_SLV_DWIDTH-1);
144   signal slv_read_ack                   : std_logic;
145   signal slv_write_ack                  : std_logic;
146
```

Figure 20: Signal Declaration

2. Register selection signals:
   The bus signals are decoded here to generate read/write select and read/write acknowledgement signals as shown in **Figure 21**.

```
162   --    Bus2IP_WrCE/Bus2IP_RdCE   Memory Mapped Register
163   --                        "1000"   C_BASEADDR + 0x0
164   --                        "0100"   C_BASEADDR + 0x4
165   --                        "0010"   C_BASEADDR + 0x8
166   --                        "0001"   C_BASEADDR + 0xC
167   --
168   ---------------------------------------------
169   slv_reg_write_sel <= Bus2IP_WrCE(0 to 3);
170   slv_reg_read_sel  <= Bus2IP_RdCE(0 to 3);
171   slv_write_ack     <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2) or Bus2IP_WrCE(3);
172   slv_read_ack      <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2) or Bus2IP_RdCE(3);
```

Figure 21: Signal Assignment

3. Register-Write Process:

This segment of code describes a process that is used to write into each of the four registers. The process checks the signals **"slv_reg_write_sel"** signal and decode it to select the proper register and transfer the data signal **"Bus2IP_Data"** into the selected register. The write operation is synchronized with the bus clock **"Bus2IP_Clk"** and the registers contents are cleared using the bus reset signal **"Bus2IP_Reset"**. This segment of code is shown in **Figure 22**. This process models the hardware required to transfer data from the PLB data bus into the contents of the selected register.

```
174   -- implement slave model software accessible register(s)
175   SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
176   begin
177
178     if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
179       if Bus2IP_Reset = '1' then
180         slv_reg0 <= (others => '0');
181         slv_reg1 <= (others => '0');
182         slv_reg2 <= (others => '0');
183         slv_reg3 <= (others => '0');
184       else
185         case slv_reg_write_sel is
186           when "1000" =>
187             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
188               if ( Bus2IP_BE(byte_index) = '1' ) then
189                 slv_reg0(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
190               end if;
191             end loop;
192           when "0100" =>
193             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
194               if ( Bus2IP_BE(byte_index) = '1' ) then
195                 slv_reg1(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
196               end if;
197             end loop;
198           when "0010" =>
199             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
200               if ( Bus2IP_BE(byte_index) = '1' ) then
201                 slv_reg2(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
202               end if;
203             end loop;
204           when "0001" =>
205             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
206               if ( Bus2IP_BE(byte_index) = '1' ) then
207                 slv_reg3(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
208               end if;
209             end loop;
210           when others => null;
211         end case;
212       end if;
213     end if;
214
215   end process SLAVE_REG_WRITE_PROC;
```

Figure 22: Register Write Process

4. Register-Read Process:

This process models the hardware required to transfer the contents of each register into the PLB data bus. The process decodes the signal **"slv_reg_read_sel"** and transfer the contents of the required register into the signal **"slv_ip2bus_data"** as shown in **Figure 23**.

```
217   -- implement slave model software accessible register(s) read mux
218   SLAVE_REG_READ_PROC : process( slv_reg_read_sel, slv_reg0, slv_reg1, slv_reg2, slv_reg3 ) is
219   begin
220
221     case slv_reg_read_sel is
222       when "1000" => slv_ip2bus_data <= slv_reg0;
223       when "0100" => slv_ip2bus_data <= slv_reg1;
224       when "0010" => slv_ip2bus_data <= slv_reg2;
225       when "0001" => slv_ip2bus_data <= slv_reg3;
226       when others => slv_ip2bus_data <= (others => '0');
227     end case;
228
229   end process SLAVE_REG_READ_PROC;
```

Figure 23: Register Read Process

5. Bus driver signals:

The final part of the architecture connects the acknowledgement signals and data signals into the bus signals as shown in **Figure 24**.

```
231   ------------------------------------------
232   -- Example code to drive IP to Bus signals
233   ------------------------------------------
234   IP2Bus_Data  <= slv_ip2bus_data when slv_read_ack = '1' else
235                   (others => '0');
236
237   IP2Bus_WrAck <= slv_write_ack;
238   IP2Bus_RdAck <= slv_read_ack;
239   IP2Bus_Error <= '0';
```

Figure 24: Bus Driver Signals

Now after investigating each component of the **"user_logic"** module, we are ready to modify it. We will modify this logic to perform simple subtraction and addition operations. We will simply subtract the contents of the first two registers and store the results in Register #2. We will also add the contents of the first two registers and store the results in Register#3 as shown in **Figure 25**.

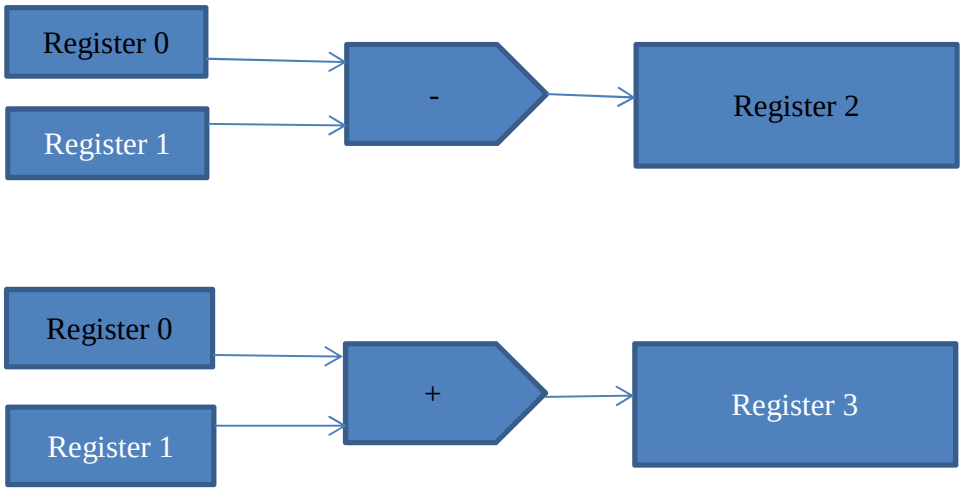

Figure 25: Hardware Modification

Add the following two lines before the end of the architecture body

```
slv_reg2 <= slv_reg0 - slv_reg1;
slv_reg3 <= slv_reg0 + slv_reg1;
```

This will build the hardware shown previously in **Figure 25**. Now as the wizard created a logic to write into registers 2 and 3, we need to modify this logic so that these registers will only have a single source. Comment the following code shown in **Figure 26**. When you complete these modifications implement the design using the ISE flow to make sure that it builds correctly before connecting the core to Microblaze in the EDK.

```
174   -- implement slave model software accessible register(s)
175   SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
176   begin
177
178     if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
179       if Bus2IP_Reset = '1' then
180         slv_reg0 <= (others => '0');
181         slv_reg1 <= (others => '0');
182 --        slv_reg2 <= (others => '0');
183 --        slv_reg3 <= (others => '0');
184       else
185         case slv_reg_write_sel is
186           when "1000" =>
187             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
188               if ( Bus2IP_BE(byte_index) = '1' ) then
189                 slv_reg0(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
190               end if;
191             end loop;
192           when "0100" =>
193             for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
194               if ( Bus2IP_BE(byte_index) = '1' ) then
195                 slv_reg1(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
196               end if;
197             end loop;
198 --          when "0010" =>
199 --            for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
200 --              if ( Bus2IP_BE(byte_index) = '1' ) then
201 --                slv_reg2(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
202 --              end if;
203 --            end loop;
204 --          when "0001" =>
205 --            for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
206 --              if ( Bus2IP_BE(byte_index) = '1' ) then
207 --                slv_reg3(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
208 --              end if;
209 --            end loop;
210           when others => null;
211         end case;
212       end if;
213     end if;
214
215   end process SLAVE_REG_WRITE_PROC;
```

Figure 26: Modified Version of the Register Write Process

Build the design in the ISE project navigator by selecting **“mycore.vhd”** and then double click **“Implement Design”** in the **“Design Tab”.**

Part 3: Adding the generated IP core the Microblaze system

Now as we are done modifying the new IP core to work as described in Figure 25. We are ready to add the newly created peripheral to our Microblaze system created in the previous tutorial. The IP core we created in Part 1 and Part 2 supports the PLB bus which is the common bus used in the Microblaze system. Now we will add the new core to the system, connect it to the bus, give it an address space and then rebuild the system to generate a new bit-stream (file). In Part 4 of this tutorial we will modify the software of the Microblaze to access this core. Now we will switch back to the EDK by starting the platform studio using the command **"xps".**

Step 1: Add the core to the system:

When you open the EDK after creating the new core, select the tap "IP Catalog" in the left side of the Platform Studio main window as shown in **Figure 27**.

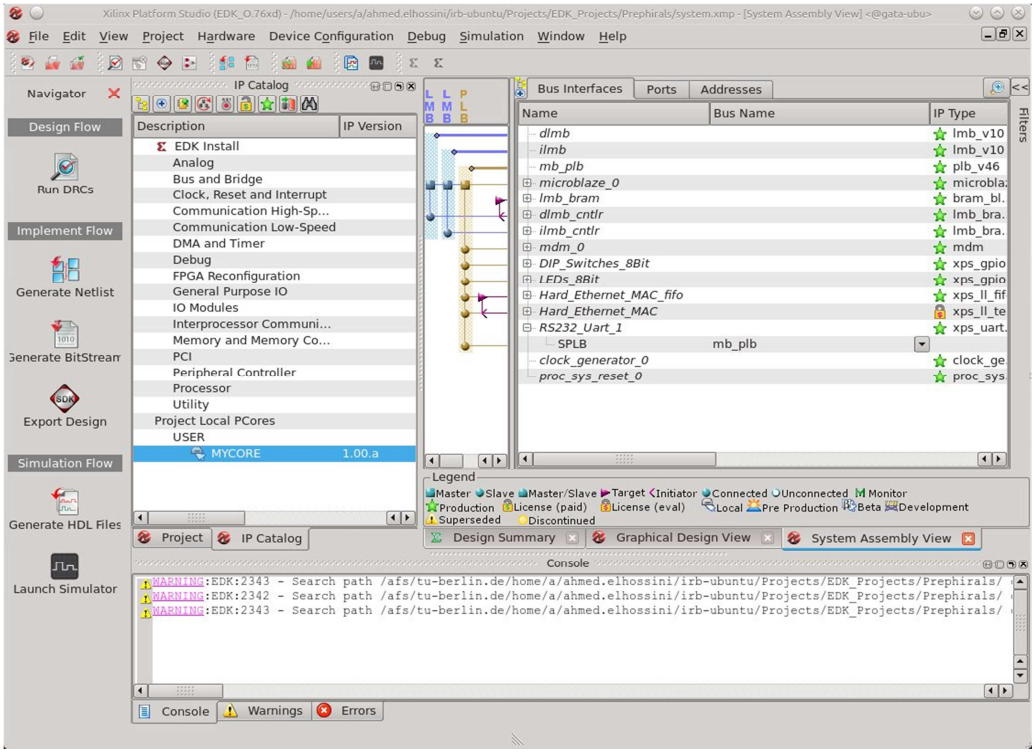

Figure 27: IP Catalog in the Platform Studio Main Window

The **"IP Catalog"** is used to explore the library of components available for you to add to your design, either created by Xilinx (under EDK Install), or by the user (under Project Local Pcores). You will notice that the core we created appears under the following catalog path: **"Project Local Pcores-> MYCORE"**. Double-clicking the core will direct the tool to add an instant of this component to your design.

First it will ask you if you want to add **"mycore"** to your design. Press "**Yes**". Then it will display a window as shown in **Figure 28**.

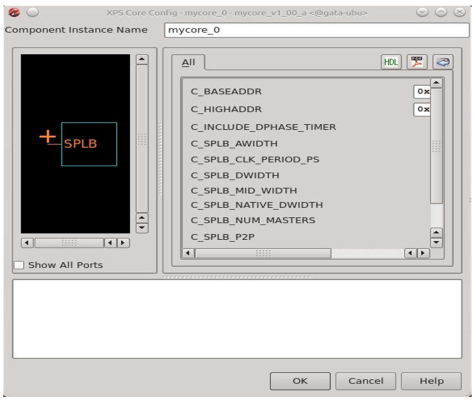

Figure 28: Instant name

When you are done editing the instant name press **"OK"** to add the component to your design as shown in **Figure 29**. This new component is now ready to be connected to the MicroBlaze common PLB bus (shown in gold). Click the "+" sign beside the name of the core (**"mycore_0")** you will notice one connection named **"SPLB"** which stands for **"slave PLB"**. This simply indicates that this component can be connected to the PLB bus as a slave component. Click the **"No Connection"** list and choose **"mb_plb"** from the list. This will add a gold circle marking the connection of the new component with the PLB bus as a slave. (Master components are shown with rectangle connections – such as the Microblaze_0 connection). **By completing this step, the new component is now connected to the PLB bus.**

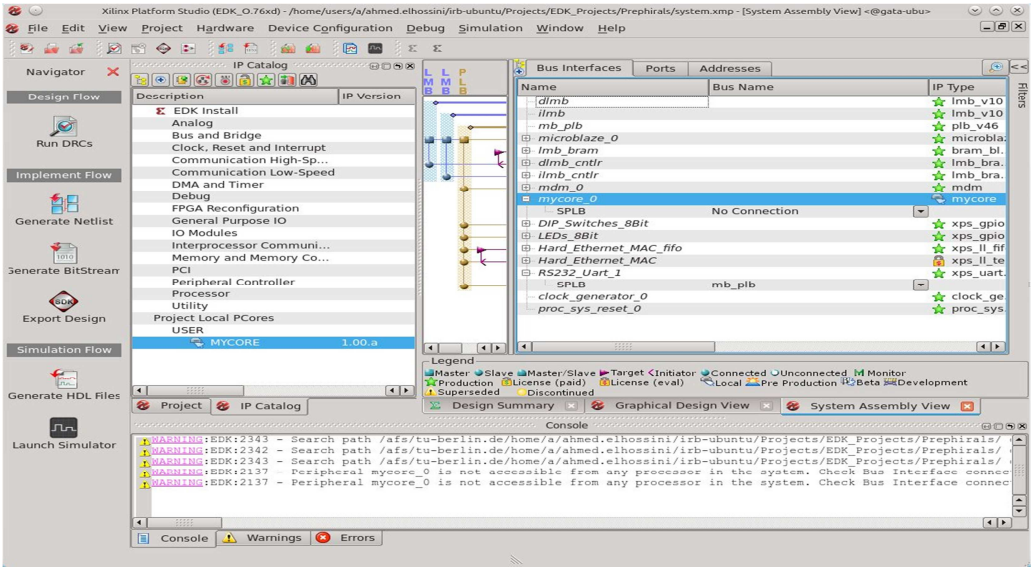

Figure 29: New component added to the system

Step 2: Modify the address space of the new core:

As explained earlier, each component connected to the PLB bus is given an address. This address is specified by the designer so that the software (user code) can access the peripheral to perform a certain computation. We need to give our newly added component an address space to be used to access its registers. To do so, click the address tab in the right side of the platform studio window.

Not that our component **"mycore_0"** will be shown under **"unmapped components"**. Select the component and then change the size of the address space to 64k. This will move the component to MicroBlaze_0 addresses list. There you will find all components connected to the Microblaze. You will also be able to list, and modify the address space of each component. In this stage we only need to give our new component a starting address. This can be any address as long as it is not used by any other component. For example, as shown in **Figure 30**, we use the address 0x81500000. Now we are done adding the new core and assigning it an address space. We are now ready to build the new hardware. Click **"Generate BitStream"**.

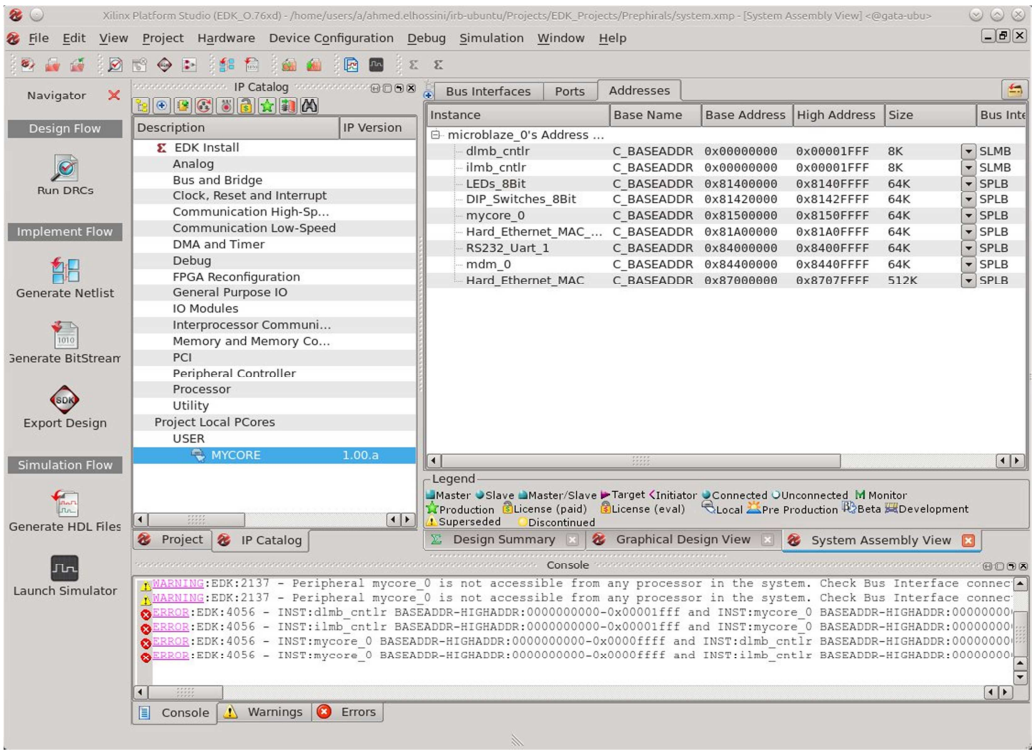

Figure 30: Address space modification

Part 4: Modifying the Software

From the software point of view, the new core contains 4 registers as described in Table 1. The wizard created a driver that will allow us to access the four registers. The driver contains several functions that will allow us to access each of those registers.

Register	Function	Address	Access	Width
Register 0	First Operand	Base Address + 0x00	Read/Write	32-bits
Register 1	Second Operand	Base Address + 0x04	Read/Write	32-bits
Register 2	Subtraction Result	Base Address + 0x08	Read Only	32-bits
Register 3	Addition Result	Base Address + 0x0C	Read Only	32-bits

Table 1: Registers of the Created IP Core

These functions are listed below:

- **MYCORE_mWriteReg(BaseAddress, RegOffset, Data): Writes data to a specific register.**
- **Data = MYCORE_mReadReg(BaseAddress, RegOffset): Reads data from a specific register.**

The tool generates various constants based on the hardware configuration to enable software access to system constants such as peripheral addresses. For example the constant XPAR_MYCORE_0_BASEADDR is defined to hold the base address of our component (in this case 0x81500000).

Step 1: Adding the Driver to SDK

Start the SDK as described in Tutorial 2. Then select **"Xilinx Tools->Repositories"**, the window shown in **Figure 31**. This will allow us to add the project local directory as a source for drivers. In the **"Local Repositories"** click **"New"** and then select the project local directory.

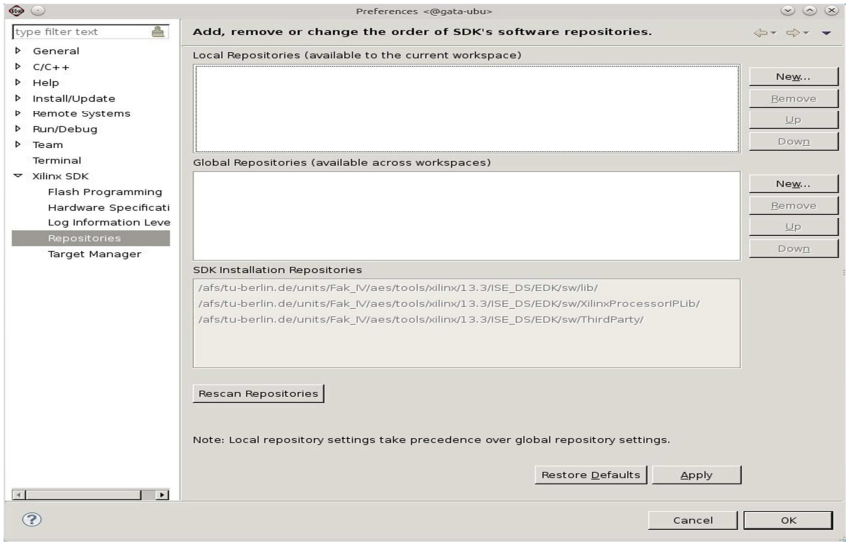

Figure 31: Add Repositories

Step 2: Create a software project:

Create a software project as described in Tutorial 2. In this project we will write software to access the core and test its functionality. Modify the source code of your **"helloworld.c"** (or the name you choose for your project) as shown in the code segment below. The values MYCORE_SLV_REGX_OFFSET are defined by the driver for each register. Note that the function **"xil_printf"** is a light version of the famous **"printf"** function. This function requires smaller memory size and runs faster than the regular function. We use this function to print out data. The output is directed to the standard output. This standard output can be the RS232 serial port if it exists. The settings of the standard output, operating system, and drivers can be changed by clicking **"Modify this BSP's Setting"** in the **"System.mss"** file found in the BSP project (for example **"Hello_world_bsp")**.

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

void print(char *str);

int main()
{
    int a = 0, b = 0;

    init_platform();

    print("Hello World¥n¥r");

    MYCORE_mWriteReg(XPAR_MYCORE_0_BASEADDR, MYCORE_SLV_REG0_OFFSET, 0x8);
    // Write the value 8 into register 0
    MYCORE_mWriteReg(XPAR_MYCORE_0_BASEADDR, MYCORE_SLV_REG1_OFFSET, 0x5);
    // Write the value 5 into register 1

    a = MYCORE_mReadReg(XPAR_MYCORE_0_BASEADDR,MYCORE_SLV_REG2_OFFSET);
    // Read the value of Register 2 into variable a
    // Should be 3 (0x03)

    b = MYCORE_mReadReg(XPAR_MYCORE_0_BASEADDR,MYCORE_SLV_REG3_OFFSET);
    // Read the value of Register 3 into variable b
    // Should be 13 (0x0D)

    xil_printf("Output = %d, %d",a , b);
    // Print out the result using the serial port
    // Xil_printf is a light version of printf

    cleanup_platform();

    return 0;
}
```

Compile the code, and then select **"Xilinx Tools-> Program FPGA"** to download the design on the FPGA and execute it.

Final Statement

After completing this tutorial you will be able to build an embedded system, add your own design to that system, and execute the complete system on the FPGA. To gain more knowledge about the EDK and SDK tools along with adding an IP to a Microblaze you need to build your own design from scratch.