
The TLUT tool flow: An introduction

The TLUT tool flow is a tool flow that generates FPGA configurations. Its biggest difference with the traditional FPGA flow is the use of a new technology mapper, the TLUT mapper. The TLUT tool flow was developed to implement Dynamic Circuit Specialization (DCS) on FPGAs. A DCS implementation of an application specializes its circuit for the current values of a number of specific inputs, called parameters. This specialized circuit is smaller, and in some cases faster, than the original circuit. However, it is only correct for one set of parameter values. Each time the parameters change value, a new specialized circuit is generated. This new specialized circuit is then loaded into the FPGA using partial run-time reconfiguration.

The TLUT tool flow offers an efficient implementation of DCS because it uses the concept of parameterized configurations. A parameterized configuration is a configuration in which some bits are expressed as Boolean functions of the parameters. Before the FPGA can be configured, the parameter values are used to evaluate the Boolean functions. This generates the specialized configuration. Several papers on the academic underpinnings of the TLUT tool flow are listed in the wiki of this project or are contained in the documentation directory. There you can also find more information on our current research on extending and improving the TLUT tool flow. The TLUT tool flow itself and how to use it, is described in much more detail in this user guide.

1.1 What can I do with this tool flow?

First, you can evaluate our adapted TLUT technology mapper. This allows you to compare its results with a conventional technology mapper and see for yourself if Dynamic Circuit Specialization can be used to optimize your application. We have included a framework to make this comparison very easy. The following pages provide clear examples and show how to adapt the framework for your own uses.

Second, our tool flow has been integrated with the Xilinx FPGA tool flow so that you can perform DCS on a commercial Virtex 2 Pro or Virtex 5 FPGA. A number of examples that you can run right away on either the XUPV2P board (Virtex 2 Pro) or ML507 board (Virtex 5) are included in this repository. Information on creating your own project is also included in this document. The tool flow may be extended in the future to support more FPGAs, such as the Virtex 6 and 7.

1.2 Contact us

The TLUT tool flow is released by Ghent University, ELIS department, Hardware and Embedded Systems (HES) group (<http://hes.elis.ugent.be>).

If you encounter bugs, want to use the TLUT tool flow but need support or want to tell us about your results, feel free to contact us. We can be reached at hes@elis.ugent.be.

1.3 Referencing the TLUT tool flow

If you use the TLUT tool flow in your work, please reference in your publications the following paper:

Karel Bruneel, Wim Heirman, and Dirk Stroobandt. 2011. "Dynamic Data Folding with Parameterizable FPGA Configurations." *ACM Transactions on Design Automation of Electronic Systems* 16 (4).

You may also refer to one of our other papers if you think it is more related.

1.4 Contents of this document

Chapter 2: *Setup*: The setup manual

Chapter 3: *Standalone TLUT tool flow*: The manual for running your own experiments without implementation on an FPGA

Chapter 4: *Xilinx-integrated TLUT tool flow*: The manual for creating your own DCS project using the Xilinx XUPV2P board

Chapter 5: *Contents*: Description of the contents of this package

Setup

```
> make
```

This downloads and builds the dependencies listed below, builds the TLUT technology mapper and creates a 'source' file which sets environment variables. Make sure you agree with the respective licences of the dependencies before running make.

Set your environment variables using:

```
> . source
```

Section 3.1 explains how to test the TLUT tool flow, while in Section 4.1 you can find information on how to test the integration with the Xilinx tool flow and the XUPV2P or ML507 board.

2.1 Dependencies

- Dependencies to be provided by the user:
 - A UNIX operating system (tools used: gcc (tested with v4.1.2), bash, minicom, curl, stty, ...)
 - Quartus II (tested with Web Edition version 11.0 SP1), Altera Corporation, <http://www.altera.com>
 - Java (tested with version 1.6.0), Oracle, <http://www.java.com>
 - Python 2.7, <http://www.python.org>
- Optional dependencies to be provided by the user:
 - A Xilinx XUP Virtex-II Pro Development System (XUPV2P board) and Xilinx Design Suite 9 (tested with version 9.1 SP2), Xilinx Inc., <http://www.xilinx.com>
Version 9 is old, but newer versions don't work well with the Virtex-II Pro.
 - A Xilinx ML507 Virtex-5 Development System and Xilinx Design Suite (tested with version 13.4), Xilinx Inc.
- Dependencies automatically downloaded and installed:
 - Aiger 1.9.4, JKU Institute for Formal Models and Verification, <http://fmv.jku.at/aiger>
 - ABC (version 810ba683c042, 5 October 2012), Berkeley Logic Synthesis and Verification Group, <http://www.eecs.berkeley.edu/~alanmi/abc>
 - RapidSmith 0.5.1, BYU RapidSmith Project, <http://rapidsmith.sourceforge.net>
And the dependencies of Rapidsmith itself, such as Hessian 4.0.6.

Standalone TLUT tool flow

The standalone TLUT tool flow can be used to evaluate the impact of DCS on your design. This tool flow will map your annotated VHDL or Verilog design and tell how much smaller or faster (LUT depth) your design can become when using parameterized configurations.

A number of examples are included in the ‘examples’ directory that you can run using the included Python scripts. E.g.

```
> cd examples/treeMult4b
> ./treeMult4b.py
```

3.1 Testing

Your installation of the TLUT tool flow can be tested using the script in the directory ‘tests’.

```
> cd tests
> ./examples_test.sh
```

If you encounter errors, troubleshooting information can be found in Section 3.5.

3.2 Creating your own project: A step-by-step approach

Below, we have provided a step-by-step approach to set up your own Python script starting from the *treeMult4b* example:

1. Make a folder for your design and copy the *treeMult4b* folder from the examples directory. You need at least the files *abc.rc* and *treeMult4b.py*

```
> mkdir yourDesign
> cp -r examples/treeMult4b/* yourDesign/
```

2. Replace *treeMult4.vhd* by copying your VHDL/Verilog files, describing your design, into the *yourDesign*-folder

```
> rm treeMult4b.vhd
```

3. Annotate the parameters in your top level VHDL/Verilog file.

Any input signal, or combination of input signals, of your top level module can be chosen as parameter. Preferably the designer will choose the slowly changing input signals as parameters, because a change in the value of the parameters results in a reconfiguration of the FPGA. Different combinations can be tested easily by changing the annotations. In the following example a simple multiplexer is described in VHDL and the ‘sel’ input signal is annotated as a parameter.

```
entity multiplexer is
port (
  --PARAM
  sel : in  std_logic_vector(1 downto 0);
  --PARAM
  in  : in  std_logic_vector(3 downto 0);
  out : out std_logic
```

```

);
end multiplexer;

architecture behavior of multiplexer is
begin
    out <= in(conv_integer(sel));
end behavior;

```

The Verilog annotations are quite similar. The following Verilog example, a simple multiplier, can be found in the examples folder;

```

module mult(x,y,z);
parameter N = 16;
input [N-1:0] x;
//PARAM
input [N-1:0] y;
//PARAM
output [2*N-1:0] z;
assign z = x * y;
endmodule

```

4. Modify the Python script

```

> mv treeMult4b.py yourDesign.py
> nano yourDesign.py

```

Edit the 6th line, in which the *run* function is called, according to the documentation in Section 3.3. The most important change you have to make is filling in the name of the VHDL file of your design.

5. Run the modified Python script

```

> ./yourDesign.py

```

6. Check the '.par' file in the work directory, to ensure the parameters were correctly recognised. It should list all the signal names of the parameters, i.e. one for each line of a bus signal. The tool for extracting parameters from VHDL and Verilog is still limited and can not yet parse all VHDL/Verilog constructs.

7. Analyze the results. As an example, the output of the Python script of the treeMult4b project is given;

```

Stage: TLUT mapper
Luts (TLUTs)      depth      check
12 (12)           1          PASSED
Stage: SimpleMAP
Luts              depth      check
67                10         PASSED
Stage: ABC fpga
Luts              depth      check
67                10         PASSED

```

The first column shows the number of LUTs in each mapping solution. These can be compared directly and represents the number of K-LUTs needed to implement the design. In this case, the TLUT mapper needs 12 LUTs, and SimpleMAP and ABC's fpga need 67.

SimpleMAP and ABC fpga both perform mapping for a conventional, not parameterised, configuration. The same results could be expected for both tools because they implement basically the same algorithm. However, ABC fpga contains some optimisations, such as 'area recovery', which haven't been (completely) implemented in SimpleMAP. While the number of LUTs may differ between the two tools, you can expect the depth to be always equal.

The number between brackets '()' is only relevant for the TLUT mapper. It shows the number of LUTs that will actually be reconfigured at run-time. We call these LUTs TLUTs. In this case all 12 of the LUTs are TLUTs and will be reconfigured at run-time.

The third column shows the depth of each mapping solution. This is the number of LUTs in the longest path, a measure for the speed of the circuit.

Finally, column 4 shows if the mapping solution has passed the equivalence test. This check is performed to see if the resulting circuit still has the same functionality as the original input.

3.3 The *run* function

To test the TLUT mapper on your own design (described in VHDL or Verilog), import the *run* function from *fast_tlutmap* in your own Python script and call it with the following arguments:

```
run(module, submodules, K, virtexFamily, performCheck,
     verboseFlag, generateImplementationFilesFlag)
```

- **module** - String
The location of the top level of your design. If your design consists of only one VHDL/Verilog module, then you only have to pass the location of this module as the first argument and you can ignore the second argument.
E.g. `run('yourDesign.vhd')`
- **submodules** - list of Strings - *optional*
If your design consists of a top level module and several submodules, you can add a list of submodules here, for example if you have one top level module and two submodules.
E.g. `run('yourTopLevelModule.vhd', ['yourFirstSubModule.vhd', 'yourSecondSubModule.vhd'])`
- **K** - integer - *default 4*
You can choose the number of inputs a LookUp Table has on your target FPGA
E.g. `run('yourDesign.vhd', 6)` or `run('yourDesign.vhd', K=6)`
- **performCheck** - boolean - *default True*
When checks are turned on, the resulting mapping is verified using a miter and satisfiability solver. This ensures that the mapped circuit implements the same functionality as the input circuit.
E.g. `run('yourDesign.vhd', performCheck=False)`
- **verboseFlag** - boolean - *default False*
This activates the verbose mode in which more information gets printed regarding the execution of the flow.
E.g. `run('yourDesign.vhd', verboseFlag=True)`
- **virtexFamily** - String - *default undefined*
Set the Virtex family to "virtex2pro" or "virtex5". This sets "K" for you and is a required argument in combination with "generateImplementationFilesFlag=True". This option is only used when implementing DCS on a real FPGA.
- **generateImplementationFilesFlag** - boolean - *default False*
Generate VHDL files for implementation of DCS on a real FPGA. This option requires "virtexFamily" to be set.

3.4 Advanced

It is also possible to start from an '.aag' or '.blif' file. As a reference for this we provide examples *AES* and *tripleDES*. Advanced experiments can be set up by copying *fast_tlutmap* and using this file as a template for your experiment.

3.5 Troubleshooting

Try running one of the examples in the 'examples' folder to verify that your environment is set up correctly.

These are a few common errors and their solution:

- *Throws "Python ImportError: No module named fast_tlutmap" (or others):*
The environment variables probably aren't set up correctly. Try running '`. source`' in the main directory. If you have moved the main directory, the paths in the 'source' file are no longer correct. Update it using '`make source`' and do '`. source`' again.

- *Throws “java.lang.OutOfMemoryError”:*

The Java technology mappers don't have enough memory available. Try to use the *setMaxMemory* function. It can be imported from the *fast_tlutmap* Python script and should be called before the *run* function. This function sets the maximum memory usage for the Java tools in megabytes.

```
from fast_tlutmap import run, setMaxMemory

setMaxMemory(4096)
run('treeMult4b.vhd', K=4, performCheck=True, verboseFlag=False)
```

- *The number of TLUTs is 0, or the reduction in number of LUTs is lower than expected:*
Check if all parameters were correctly extracted from your VHDL/Verilog. See Section 3.2, step 6.

Xilinx-integrated TLUT tool flow

The Xilinx-integrated TLUT tool flow can be used to build an application with DCS on the XUPV2P or ML507 board (Currently only designs in VHDL, not Verilog). A number of examples for the XUPV2P are included in the 'examples' directory.

- 'examples/xorExample/xps'
- 'examples/xorExample/xps_mi'
- 'examples/treeMult4b/xps'

An example for the ML507 is also included in the 'examples' directory.

- 'examples/xorExample/xpsV13'

4.1 Testing

This section explains how you can run all these examples to test your setup.

First, connect your board to the computer using a USB cable (for bitstream configuration) and a serial cable (for debugging). Then, open each of the projects (see above) in XPS (using the version 9 for the XUPV2P and version 13.4 for the ML507) and choose "Generate bitstream". You may immediately terminate this process or it may produce an error but this is ok, this creates a number of files for the project so that the automated test scripts can run.

Now you can run all tests on these projects by doing:

```
> cd tests
> ./virtex2pro_test.sh
or
> ./virtex5_test.sh
```

Every project is now being built, loaded onto the FPGA and DCS is tested using the embedded PowerPC (you can find the debug output from the FPGA in the 'received.txt' file in the project directory). This will take a long time. Make sure minicom is not running at the same time, because this will make it impossible to read the debug output from the FPGA. If everything works fine you will get the following result.

```
Testing xorExample/xps
...
xorExample/xps succeeded
Testing xorExample/xps_mi
...
xorExample/xps_mi succeeded
Testing treeMult4b/xps
...
treeMult4b/xps succeeded
```

You can manually run one example by going to the 'pcores/opb_<design_name>/design' subdirectory of the example and running 'generateTMAPMake.py virtex2pro' from there (this has to be done only the first time). After this you can choose "Download bitstream" from within XPS (in version 13.4 you also need to run the application from Xilinx SDK). The debug output of the FPGA can be read with minicom.

If you encounter errors, troubleshooting information can be found in Section 4.3.

4.2 Your own project

This tutorial guides you through the setup and implementation of a design with DCS. You can use it to start your own project or test it with the 'xorExample' design.

Step 1: Creating the project for the XUPV2P, p. 9

Step 2: Adding the HWICAP, p. 14

Step 3: Creating the DCS peripheral, p. 15

Step 4: Adding the DCS peripheral, p. 21

Step 5: Writing the DCS peripheral, p. 23

Step 6: Creating the software application, p. 22

Step 7: Creating the custom makefile, p. 23

Step 8: Writing the software application, p. 25

Step 9: Testing, p. 26

Differences for the Virtex 5, p. 26.

Creating a project for the XUPV2P

- Start XPS v9. Make sure that you have sourced the 'source' file from the base directory of this repository in the same terminal as you start XPS in.
- Create a new project
- Chose the "Base System Builder wizard"
- Chose the project directory
- Follow these steps:

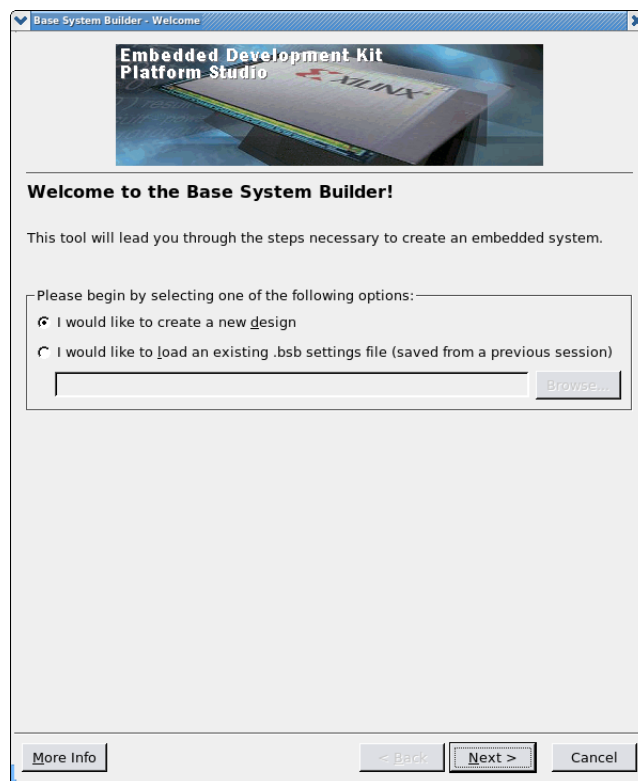


Figure 4.1: Choose to create a new design. Click Next

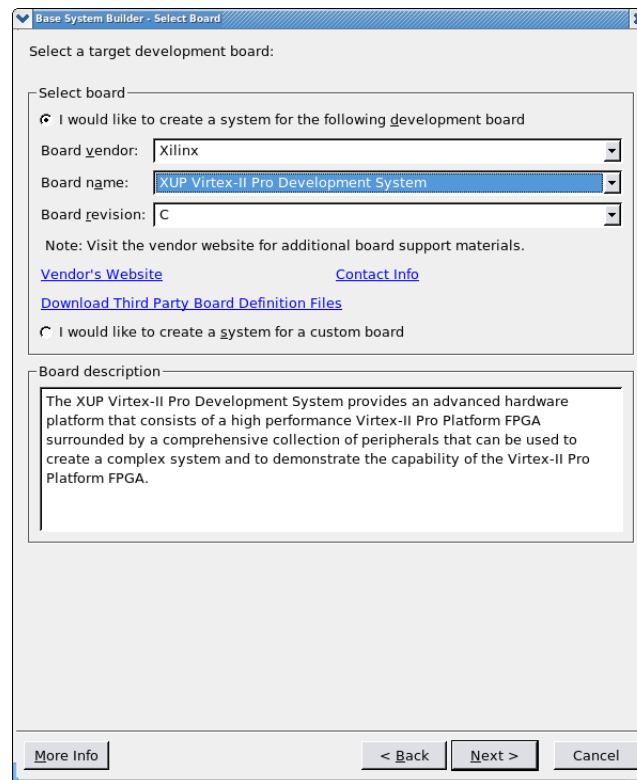


Figure 4.2: Choose Xilinx as board vendor. Choose the XUP board. Click Next

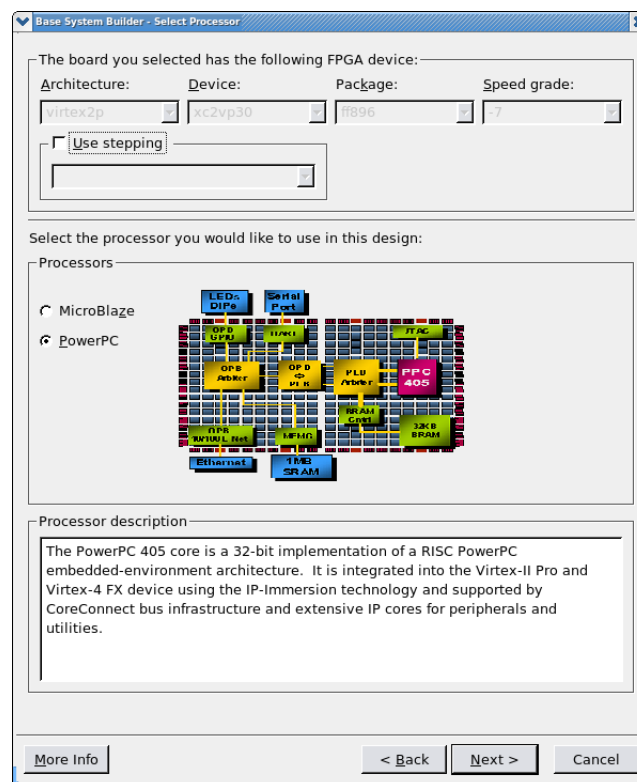


Figure 4.3: Choose to use a the PowerPc. Click Next

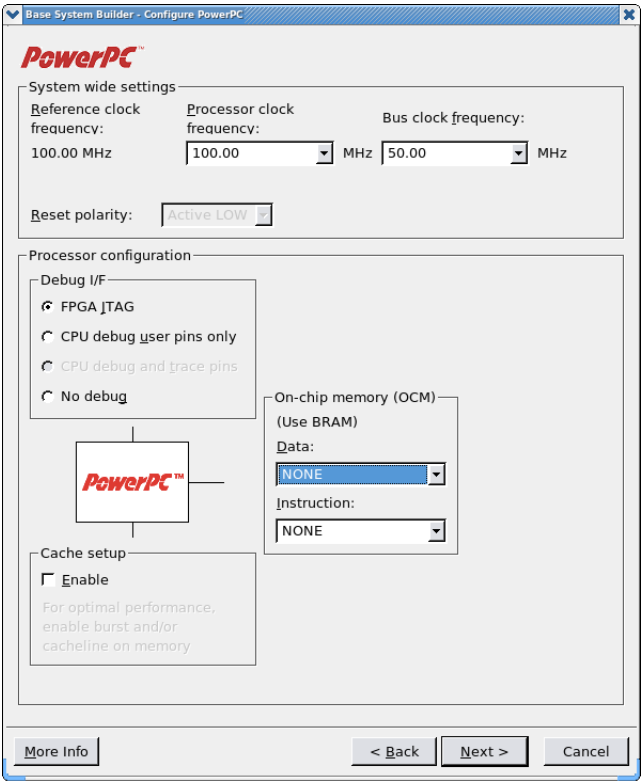


Figure 4.4: Choose 50MHz as Bus Clock Frequency (The ICAP can maximally handle 66MHz). Click Next

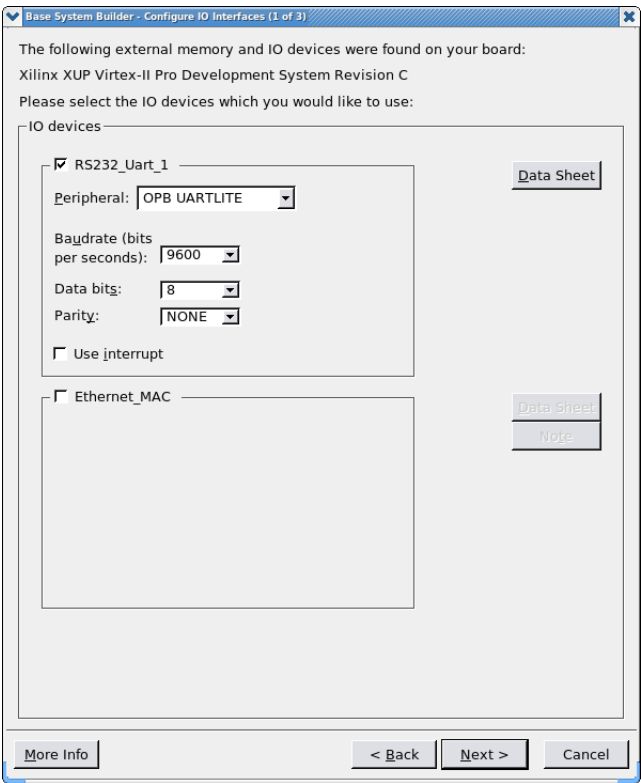


Figure 4.5: Retain only the OPB_UARTLITE in the next three windows

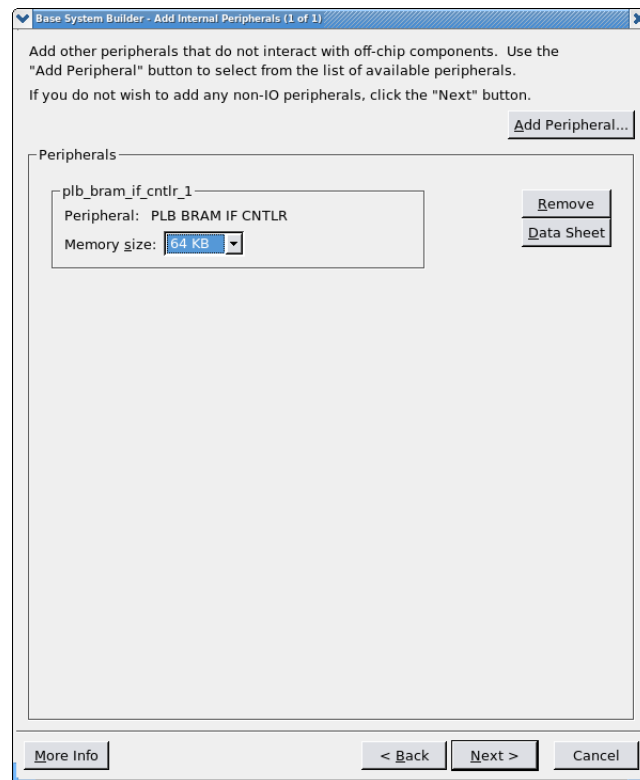


Figure 4.6: Use 64kB of memory. Click Next

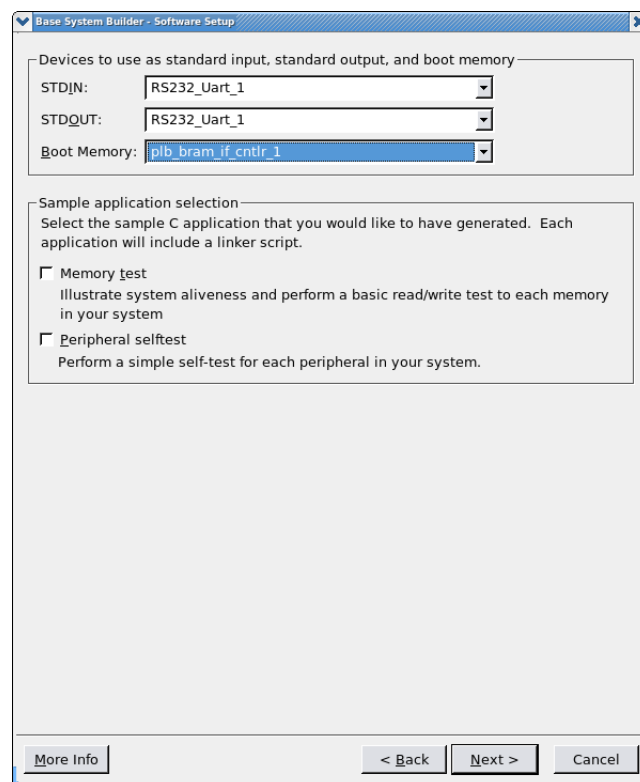


Figure 4.7: Choose the UART as STDIN and STDOUT. Don't generate sample applications. Click Next

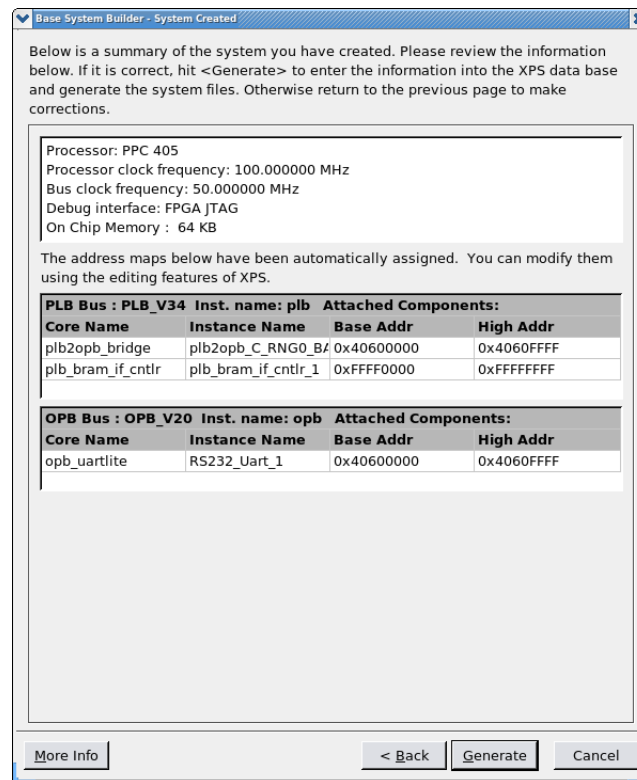


Figure 4.8: Generate the project. Click Generate

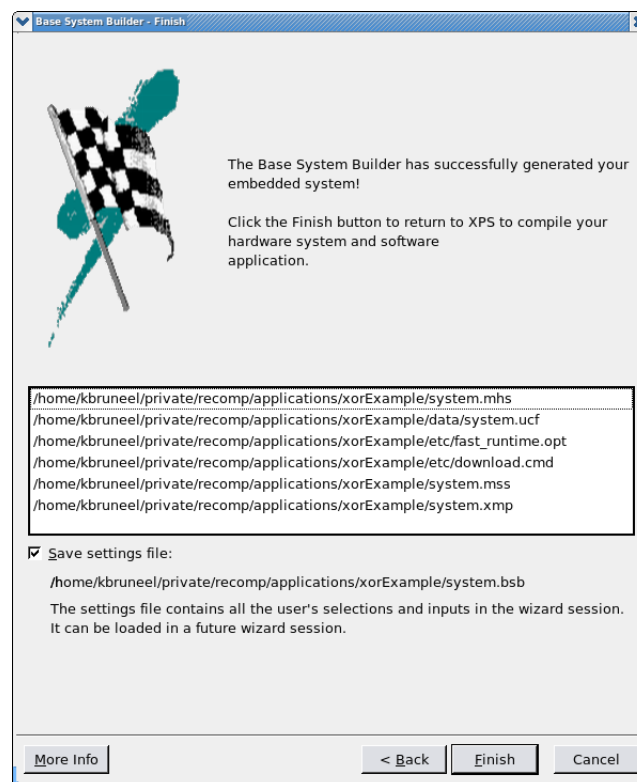


Figure 4.9: Click Finish

Adding the HWICAP

The HWICAP is used for run-time reconfiguration of the FPGA. This peripheral must be added to the project and connected to the OPB.

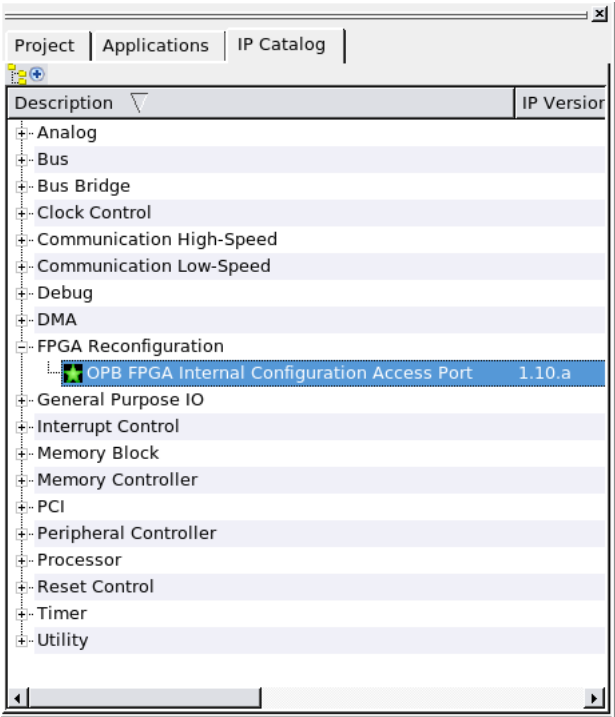


Figure 4.10: In the IP Catalog, double click “OPB FPGA Internal Configuration Access Port”. Click Yes

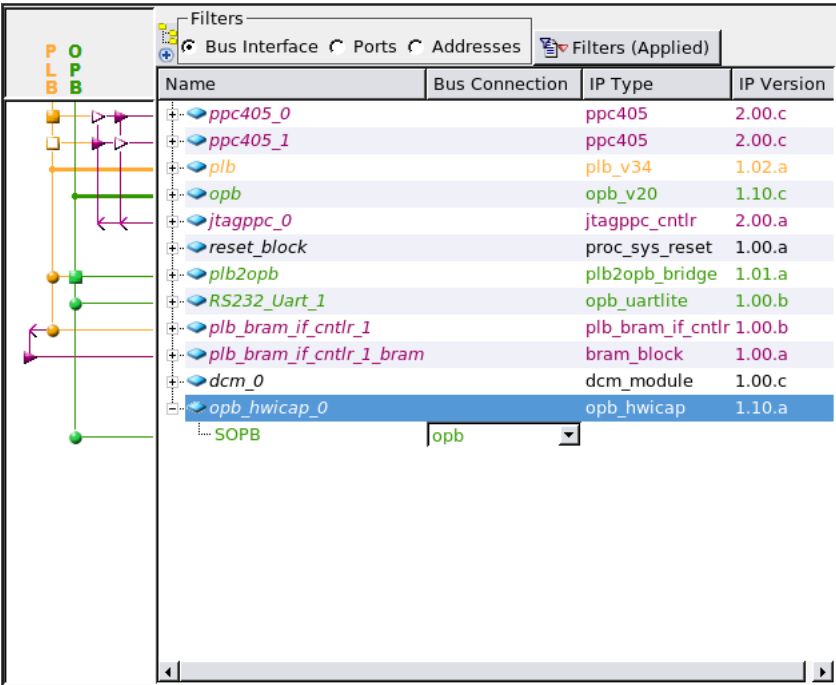


Figure 4.11: An instance of the opb_hwicap appears in the System Assembly View. Connect the hwicap to the OPB

```
156  
157  
158 BEGIN opb_hwicap  
159 PARAMETER INSTANCE = opb_hwicap_0  
160 PARAMETER HW_VER = 1.00.b  
161 BUS_INTERFACE SOPB = opb  
162 END  
163  
164
```

Figure 4.12: Open the MHS file by double clicking it in the Project tab. Search the opb_hwicap instance and change the version from *1.10.a* to *1.00.b*

Creating the DCS peripheral

Currently, DCS can only be done on exactly one peripheral of the OPB bus. In this section you create this OPB peripheral with memory mapped registers.

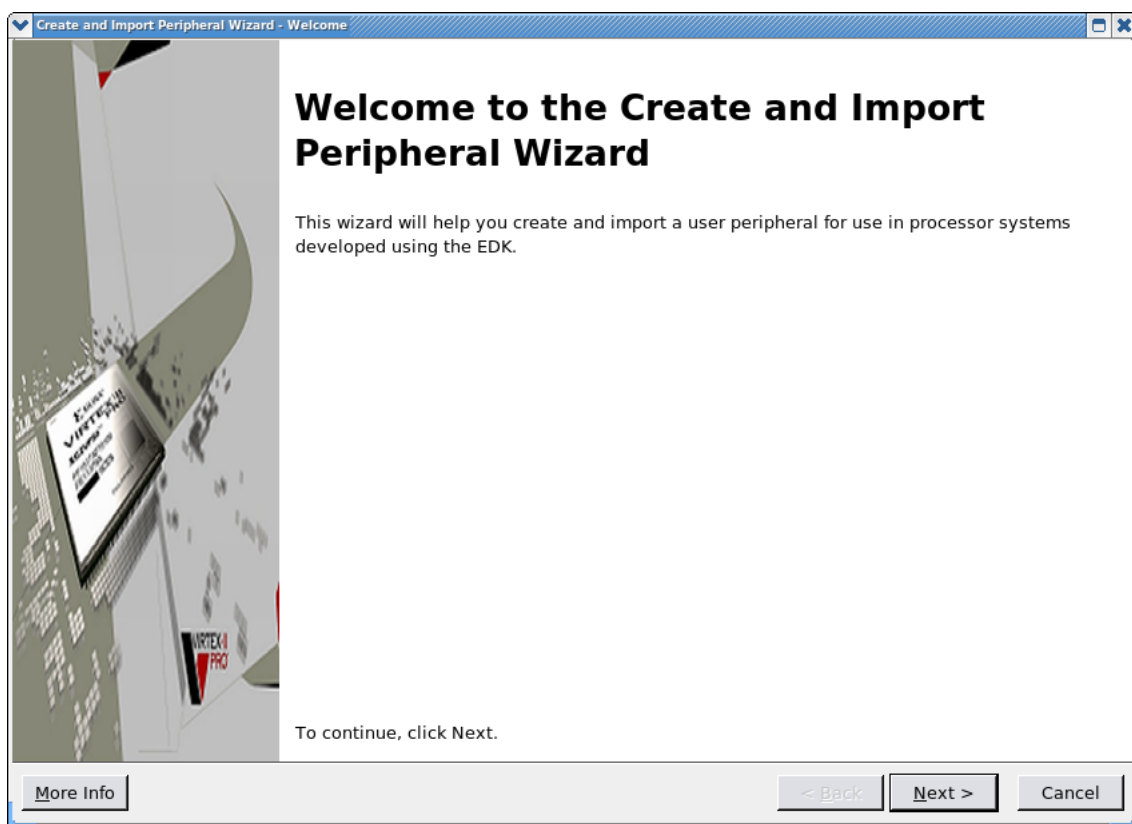


Figure 4.13: Start the Create and Import Peripheral Wizard. Start the wizard from the Hardware menu. Click Next

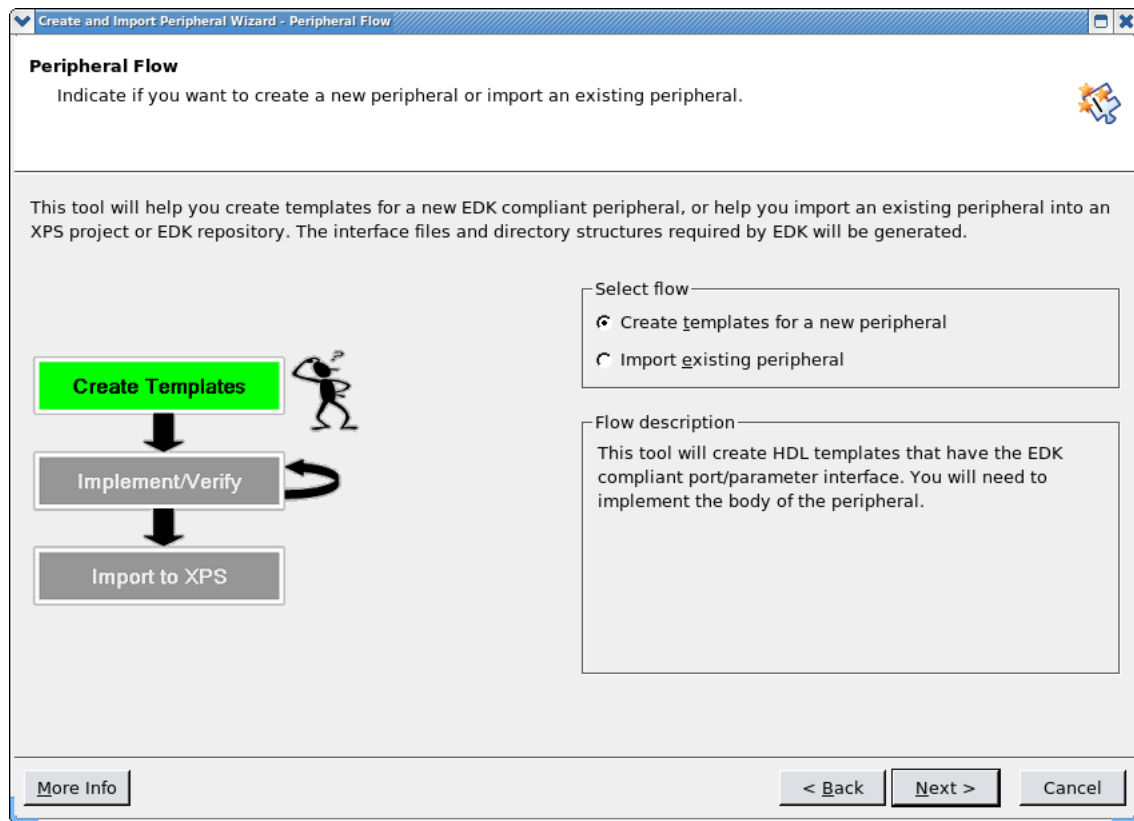


Figure 4.14: Choose Create templates for a new peripheral. Click Next

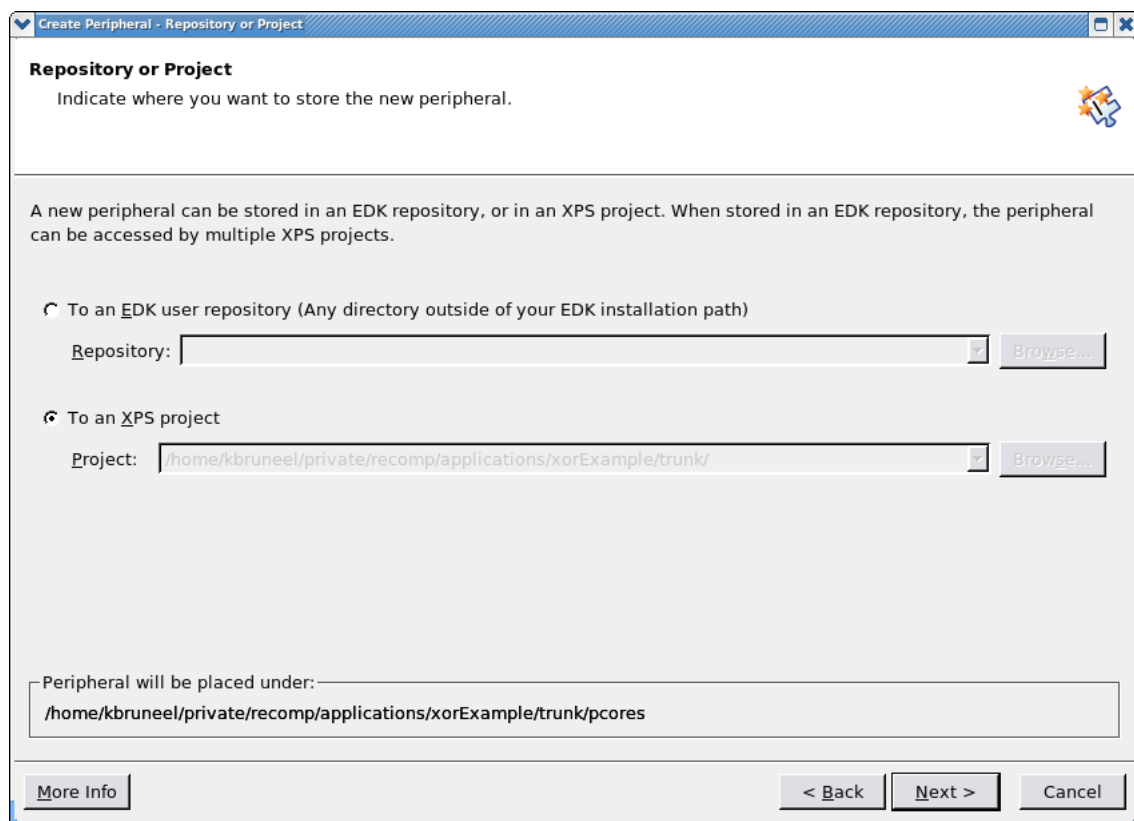
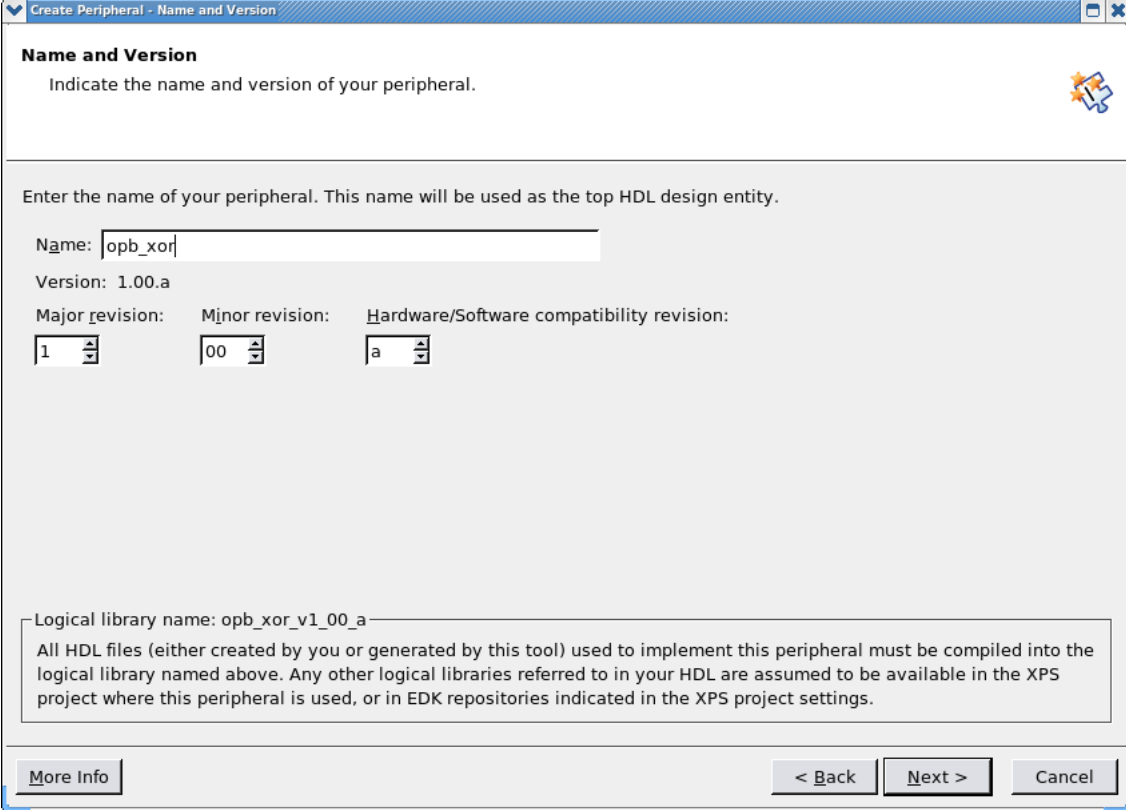


Figure 4.15: Choose to store the peripheral in the XPS project. Click Next



Create Peripheral - Name and Version

Indicate the name and version of your peripheral.

Enter the name of your peripheral. This name will be used as the top HDL design entity.

Name:

Version: 1.00.a

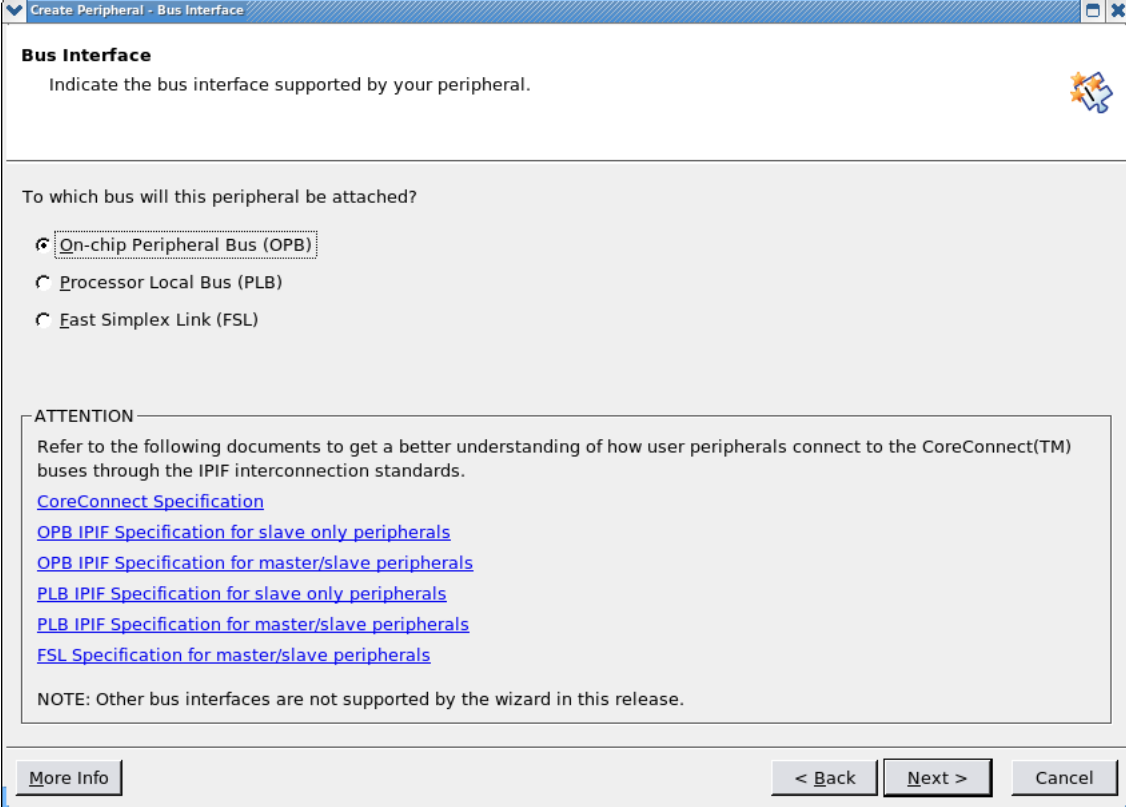
Major revision: Minor revision: Hardware/Software compatibility revision:

Logical library name: opb_xor_v1_00_a

All HDL files (either created by you or generated by this tool) used to implement this peripheral must be compiled into the logical library named above. Any other logical libraries referred to in your HDL are assumed to be available in the XPS project where this peripheral is used, or in EDK repositories indicated in the XPS project settings.

[More Info](#) [< Back](#) [Next >](#) [Cancel](#)

Figure 4.16: Choose a name for the peripheral (E.g. 'opb_xor'). Click Next



Create Peripheral - Bus Interface

Indicate the bus interface supported by your peripheral.

To which bus will this peripheral be attached?

☒ On-chip Peripheral Bus (OPB)

☐ Processor Local Bus (PLB)

☐ Fast Simplex Link (FSL)

ATTENTION

Refer to the following documents to get a better understanding of how user peripherals connect to the CoreConnect(TM) buses through the IPIF interconnection standards.

[CoreConnect Specification](#)

[OPB IPIF Specification for slave only peripherals](#)

[OPB IPIF Specification for master/slave peripherals](#)

[PLB IPIF Specification for slave only peripherals](#)

[PLB IPIF Specification for master/slave peripherals](#)

[FSL Specification for master/slave peripherals](#)

NOTE: Other bus interfaces are not supported by the wizard in this release.

[More Info](#) [< Back](#) [Next >](#) [Cancel](#)

Figure 4.17: Choose to create an OPB peripheral. Click Next

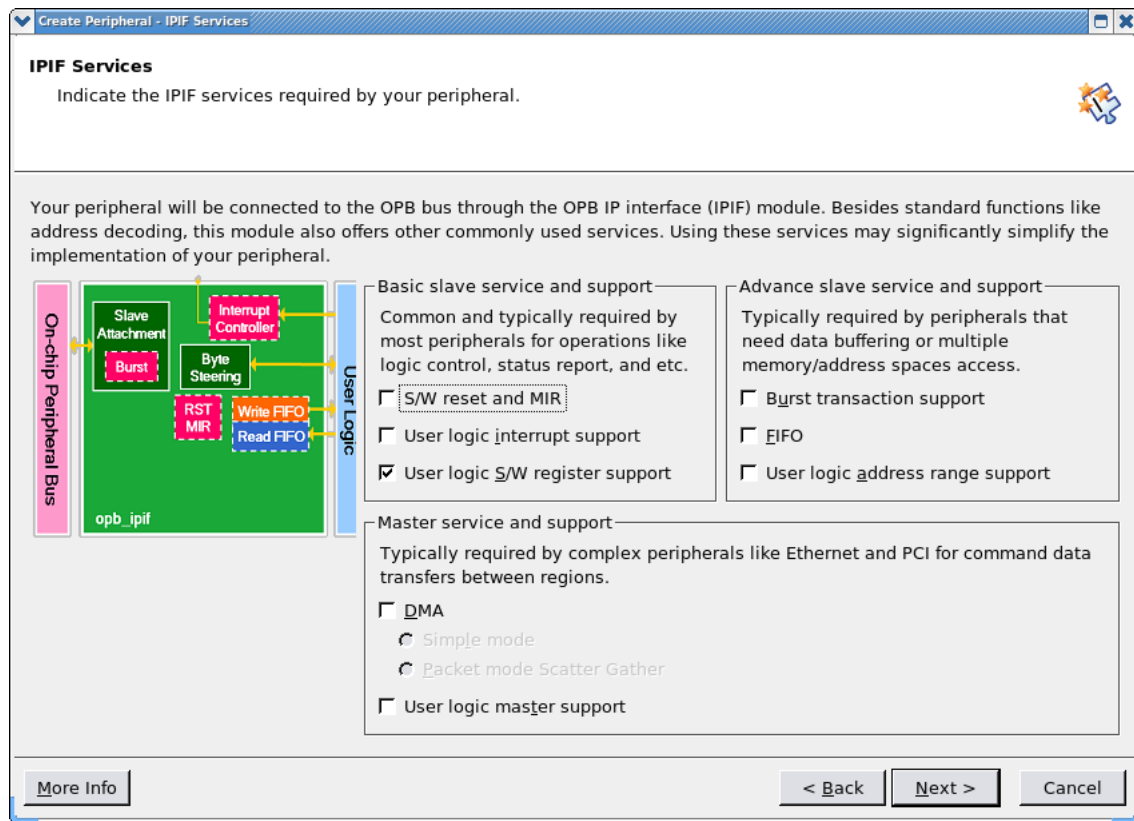


Figure 4.18: Choose for User logic S/W register support deselect S/W reset and MIR. Click Next

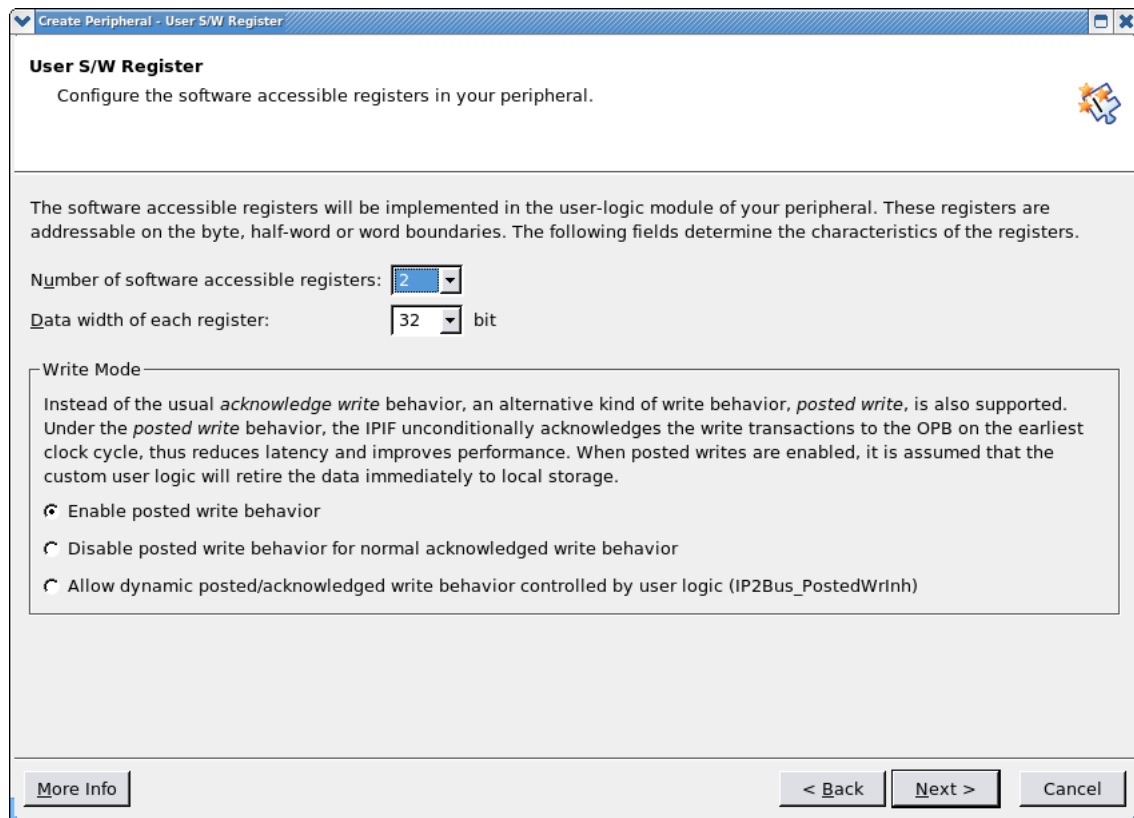


Figure 4.19: Choose the required number and width of the registers. For the 'xorExample' choose two 32-bit registers, one for the input and one for the output. Click Next

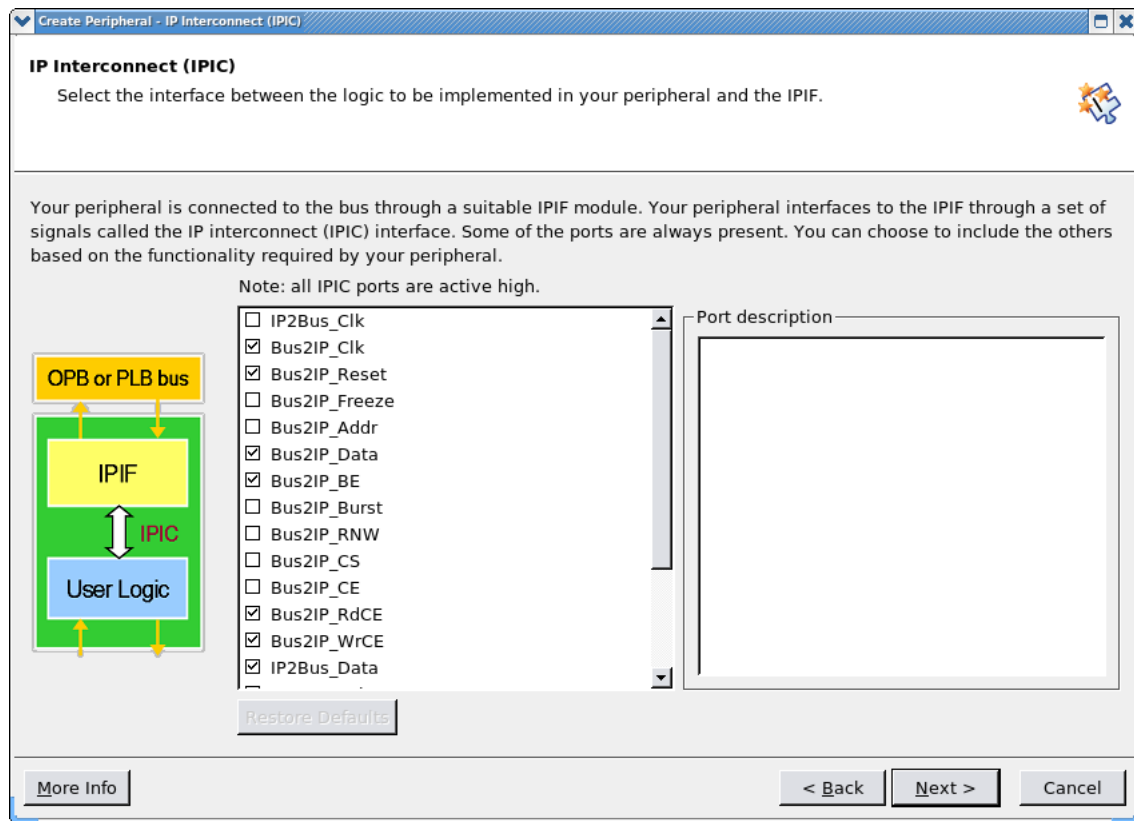


Figure 4.20: Don't change anything. Click Next

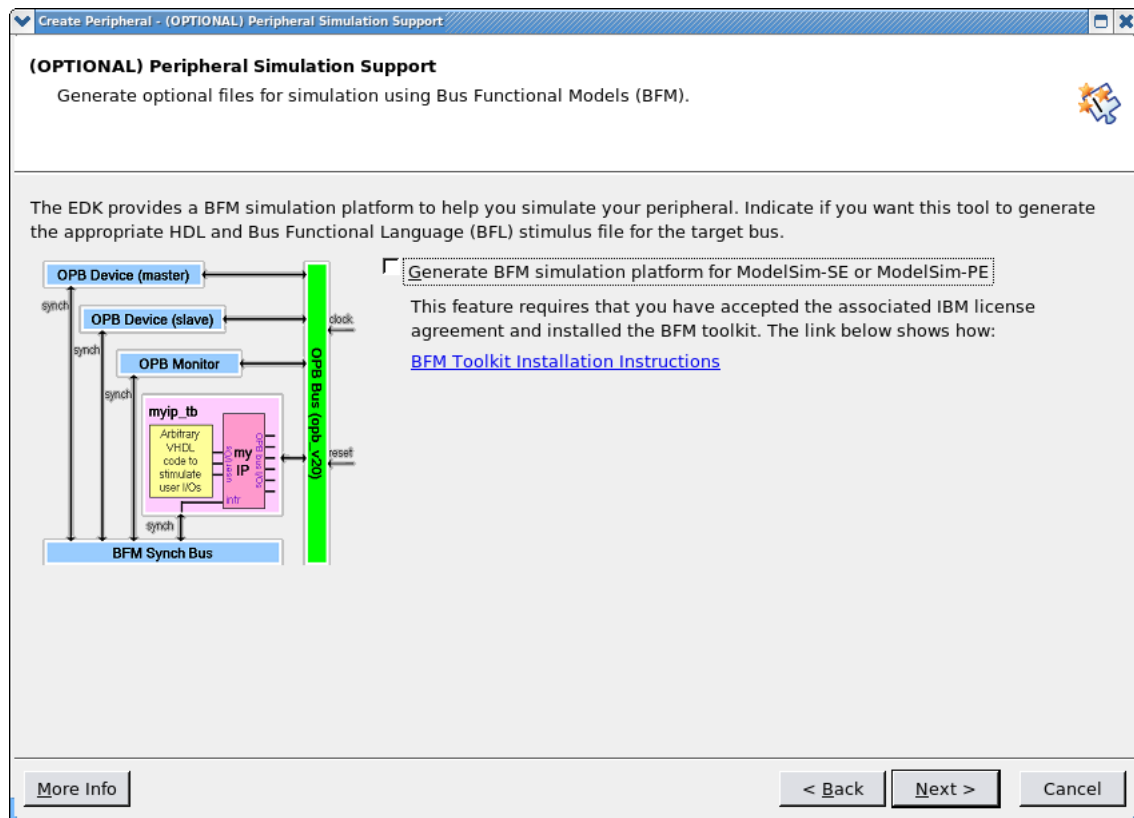


Figure 4.21: Don't change anything. Click Next

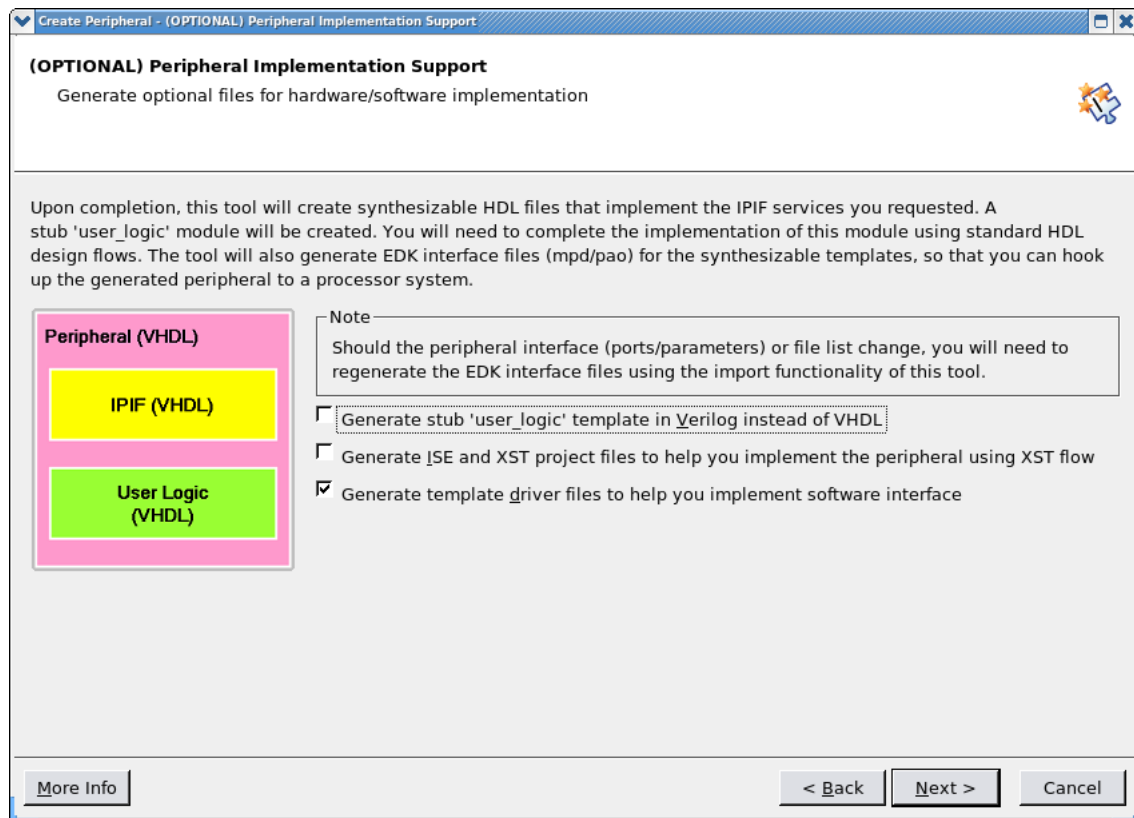


Figure 4.22: Deselect Generate ISE and XPS project files. Click Next

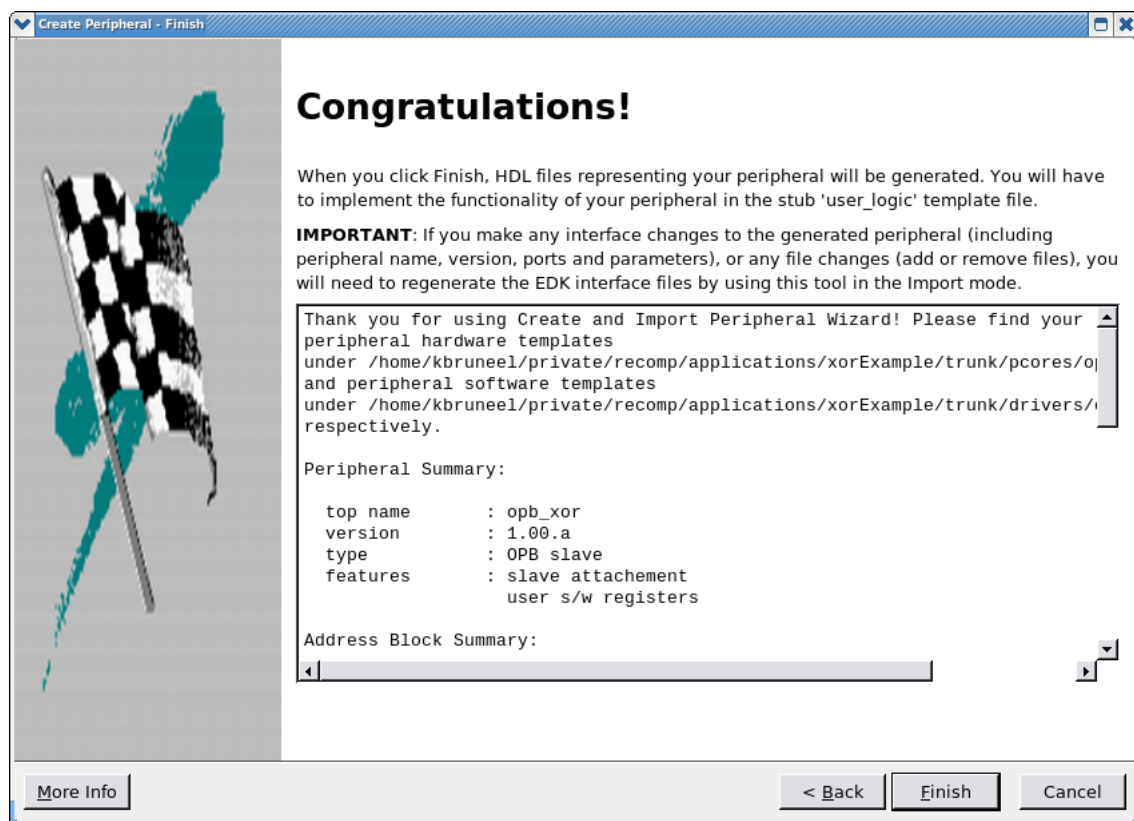


Figure 4.23: Click Finish

Adding the DCS peripheral

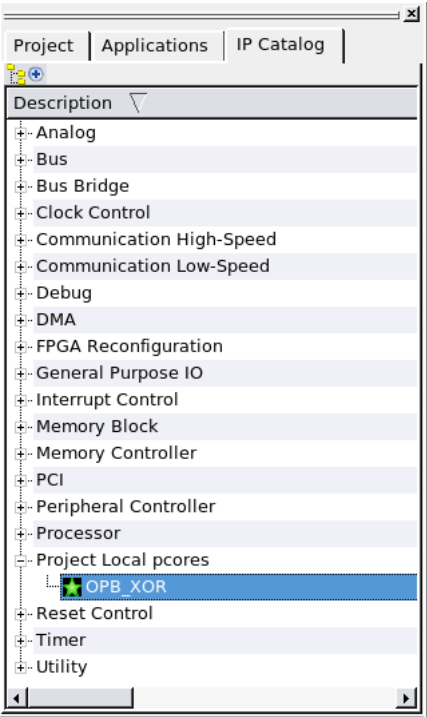


Figure 4.24: In the IP Catalog, double click your OPB peripheral (E.g. 'OPB_XOR'). Click Yes.

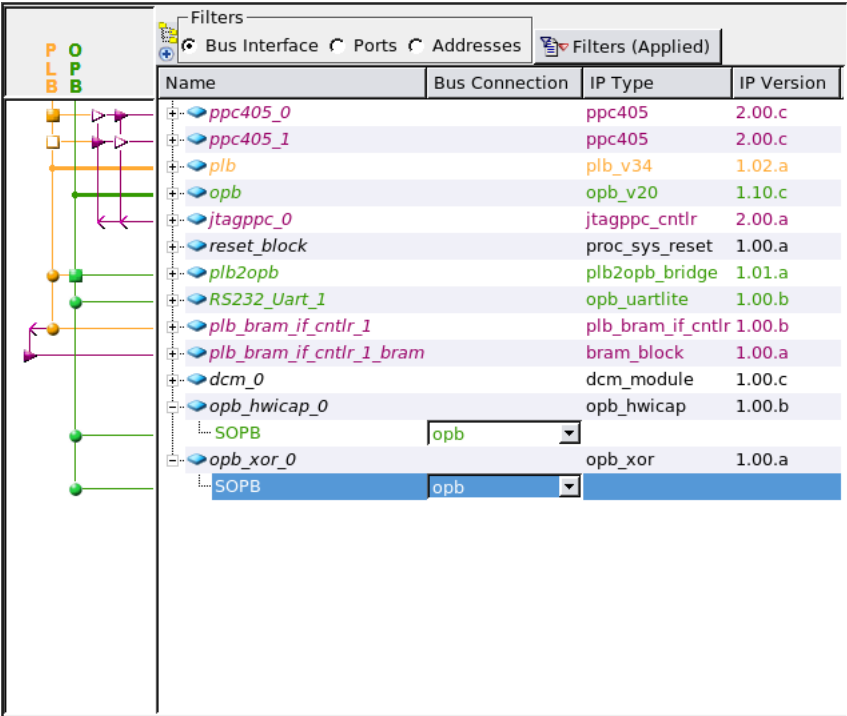
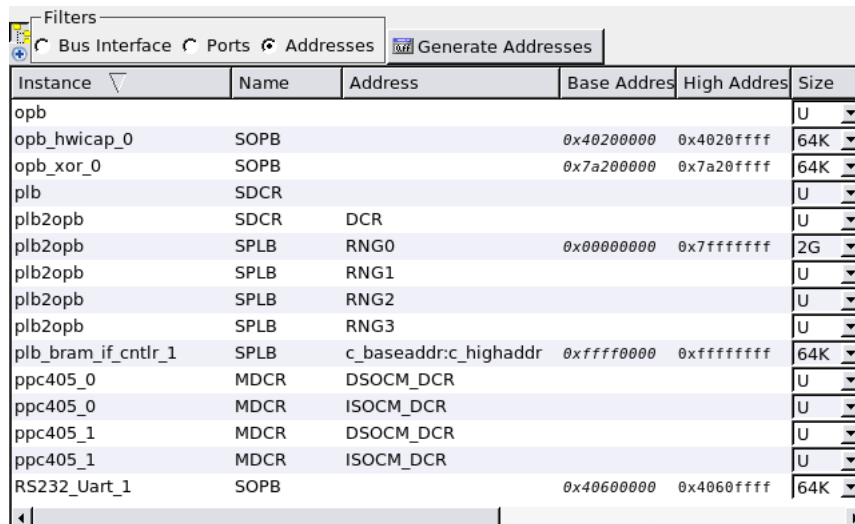


Figure 4.25: An instance of the peripheral appears in the System Assembly View. Connect the peripheral to the opb



Instance	Name	Address	Base Address	High Address	Size
opb					U
opb_hwicap_0	SOPB		0x40200000	0x4020ffff	64K
opb_xor_0	SOPB		0x7a200000	0x7a20ffff	64K
plb	SDCR				U
plb2opb	SDCR	DCR			U
plb2opb	SPLB	RNG0	0x00000000	0x7fffffff	2G
plb2opb	SPLB	RNG1			U
plb2opb	SPLB	RNG2			U
plb2opb	SPLB	RNG3			U
plb_bram_if_cntlr_1	SPLB	c_baseaddr:c_highaddr	0xffff0000	0xffffffff	64K
ppc405_0	MDCR	DSOCM_DCR			U
ppc405_0	MDCR	ISOCM_DCR			U
ppc405_1	MDCR	DSOCM_DCR			U
ppc405_1	MDCR	ISOCM_DCR			U
RS232_Uart_1	SOPB		0x40600000	0x4060ffff	64K

Figure 4.26: Select the Addresses view in the System Assembly View. Click the Generate Addresses button

Creating the software application

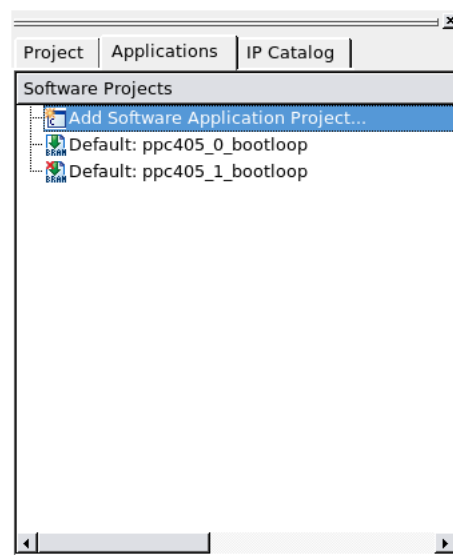


Figure 4.27: Double Click “Add Software Application Project...” in the Applications tab

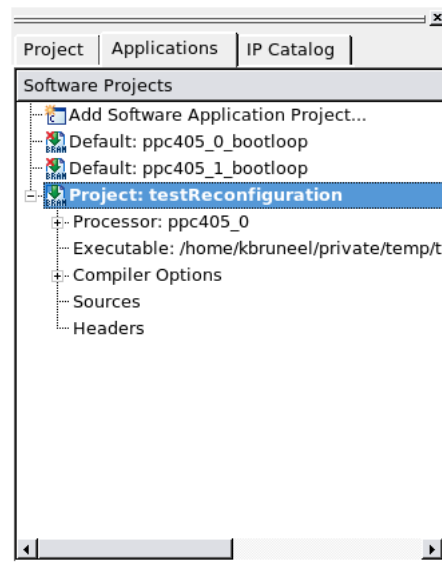


Figure 4.28: Right click on the project. Click “Mark to Initialize BRAMs” in the pop down menu. Also unmark ppc405_0_bootloop

Creating the custom makefile

- Choose “Generate netlist”. You may immediately terminate it once started.
- Copy ‘system.make’ to ‘custom.make’ in your project directory.
- In ‘custom.make’, add the line “include tmap.make” right after “include system_incl.make”.
- Click “Project Options...” in the “Project” menu and choose ‘custom.make’ as the Custom Makefile (Figure 4.29).

Make sure that you copy ‘system.make’ after creating the software application. Do this again when you add new software applications.

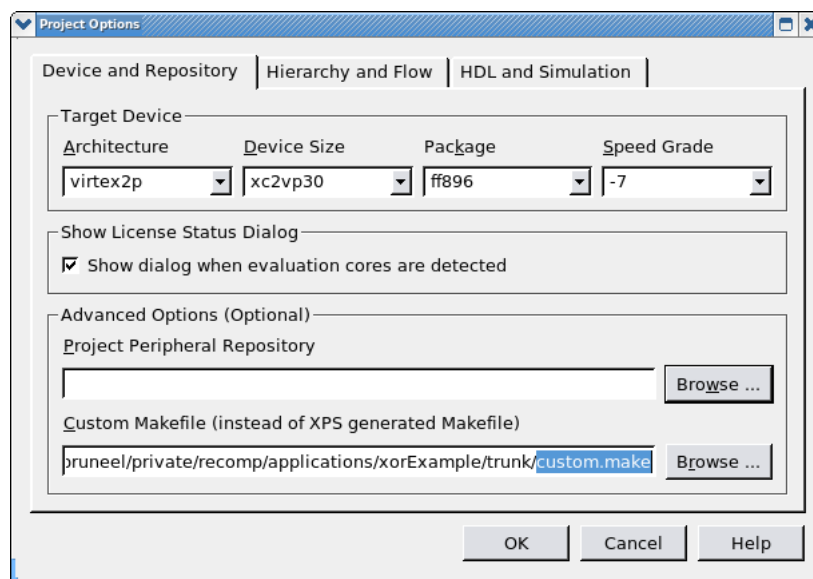


Figure 4.29: Choose ‘custom.make’ as the Custom Makefile. Click Ok

Writing the DCS peripheral

- Go to the subdirectory corresponding to your DCS peripheral in the ‘pcores’ directory of the project.

- Copy the 'hdl/vhdl' directory to the 'design' directory. It should contain two files 'opb_<peripheral_name>.vhd' and 'user_logic.vhd'.
- Add the VHDL code of your DCS design to the 'design' directory. It is advised that you run your design through the standalone TLUT tool flow first (Section 3). Add the line '--TMAP' to the top of the file that you want to perform DCS on. This does not have to be the top level module of the peripheral.
- Add the names of your VHDL modules to the '.pao' file in the 'data' directory. Modules that are only used as a submodule of your DCS module should not be included. Note that if a module depends on another module it should be listed after it.
- Add the line "OPTION CORE_STATE = development" to the '.mpd' file in the 'data' directory to ensure that your peripheral is synthesized again if changes are made to it.
- Instantiate your module in the architecture description of 'user_logic.vhd' and connect the inputs and outputs to the memory-mapped registers of the OPB peripheral. Assign the parameter inputs a dummy value.
E.g. for 'xorExample':

```
EXORS: entity work.exorw32
port map (
    a => slv_reg0,
    x => slv_reg1,
    p => (others => '0')
);
```

- Remove the write functionality for read-only registers. For the 'xorExample', 'slv_reg1' is read-only (Figure 4.30).

```
181 SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
182 begin
183     if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
184         if Bus2IP_Reset = '1' then
185             slv_reg0 <= (others => '0');
186             -- slv_reg1 <= (others => '0');
187         else
188             case slv_reg_write_select is
189                 when "10" =>
190                     for byte_index in 0 to (C_DWIDTH/8)-1 loop
191                         if ( Bus2IP_BE(byte_index) = '1' ) then
192                             slv_reg0(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8
193                             end if;
194                         end loop;
195                 when "01" =>
196                     for byte_index in 0 to (C_DWIDTH/8)-1 loop
197                         if ( Bus2IP_BE(byte_index) = '1' ) then
198                             -- slv_reg1(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index
199                             -- end if;
200                         end loop;
201                 when others => null;
202             end case;
203         end if;
204     end if;
205 end process SLAVE_REG_WRITE_PROC;
```

Figure 4.30: Removing the write functionality for read-only registers

- Enter the 'design' directory and run 'generateTMAPMake.py <your_fpga>' with <your_fpga> either "virtex2pro" or "virtex5". This creates the custom makefile 'tmap.make'. You have to rerun 'generateTMAPMake.py' if you add files to the DCS peripheral or change the '--TMAP' annotation.
- Choose "Generate bitstream" in XPS to build your peripheral.
- Look near the end of the output of this command for "Group path: " followed by something similar to "opb_xor_0/opb_xor_0/USER_LOGIC_I/EXORS/*". Edit 'data/system.ucf' and add the following lines

before the IO constraints, replacing `<group_path>` with the path found before. Enlarge the “RANGE” of the group as needed. This area constraint makes sure that run-time reconfiguration does not break the rest of your design. This is necessary because reconfiguring a frame resets the contents of the SRLs and RAM LUTs in it.

```
INST "<group_path>" AREA_GROUP=group1;
AREA_GROUP "group1" COMPRESSION=0;
AREA_GROUP "group1" RANGE=SLICE_X0Y*:SLICE_X51Y*;
```

Writing the software application

- Choose “Generate bitstream” if you haven’t already done this. This will create a number of C files in ‘swReconfiguration’ that contain the functions needed to perform run-time reconfiguration.
- Under the applications tab, right click “Sources” and choose “Add existing files...”. Add ‘swReconfiguration/locations.c’ and ‘swReconfiguration/<TMAPmodule>.c’ with TMAPmodule the name of the VHDL module that has the “--TMAP” annotation.
- Right click “Headers” choose “Add existing files...”. Add ‘swReconfiguration/locations.h’ and ‘swReconfiguration/<TMAPmodule>.h’.
- Right click “Sources” and choose “Add new file...”. Call it ‘main.c’ and save it.
You can find a template for your ‘main.c’ file below or at the bottom of ‘swReconfiguration/<TMAPmodule>.h’ (only for Virtex 2 Pro). Replace “XPAR_OPB_XOR_0_BASEADDR” with the name of the base address of your peripheral. You can find this in ‘Generated Header: ppc405_0/include/xparameters.h’ under ‘Processor: ppc405_0’.
- Choose “Build all user applications” to build your software application.

```
#include "<TMAPmodule>.h"

int main(void) {
    xil_printf("Starting EXOR test...\n\r\n\r");
    //Initialization
    static XHwIcap HwIcap;
    XHwIcap_Initialize(&HwIcap, HWICAP_DEVICEID, XHI_TARGET_DEVICEID);
    //Run-time reconfiguration
    Xuint8 i;
    Xuint8 parameter[NUMBER_OF_PARAMETERS];
    Xuint8 output[NUMBER_OF_INSTANCES][16];
    xil_printf("Configuring the LUTs for p=0...\n\r");
    for (i=0;i<NUMBER_OF_INSTANCES;i++) {
        //Reconfigure one instance
        parameter[0]=0;
        evaluate(parameter,output);
        reconfigure(&HwIcap,output,location_array[i]);
    }
    xil_printf("Configuration Complete!\n\r\n\r");
    //Testing configuration
    xil_printf("Writing 0xDEADBEAF to input register...\n\r");
    XIo_Out32(XPAR_OPB_XOR_0_BASEADDR,0xDEADBEAF);
    xil_printf("Reading output register: %x\n\r\n\r",
        XIo_In32(XPAR_OPB_XOR_0_BASEADDR+4));
    xil_printf("End EXOR test.\n\r\n\r");
    return 1;
}
```

Explanation of the functions:

evaluate(parameter,output) calculates the new truth table contents of the TLUTs for the new values of the parameters. The values of the parameter signals have to be stored as boolean values in the *parameter* array (one array element per signal line). The parameters are sorted alphabetically. You can check this ordering in ‘pcores/opb_<your_peripheral>/design/work/<TMAPmodule>.par’.

`reconfigure(&HwIcap,output,location_array[i])` reconfigures the TLUTs of instance *i* with the previously computed truth table contents. The instances are also sorted alphabetically. You can find this ordering in 'swReconfiguration/locations.c'.

Testing the design

The project can now be tested. To do this, connect your XUPV2P board to the computer using a USB cable (for bitstream programming) and a serial cable (for debugging). Start the program minicom to read the debug output from your XUPV2P board. Then, choose "Download bitstream" in XPS to run your project on the FPGA.

Differences for the Virtex 5

For the Virtex 5 FPGA the steps are similar but might look different because of the newer Xilinx Design Suite that is used. Summarised, you need to create a project (for the ML507 board) with a PowerPC, then add the HWICAP and connect it to the PLB bus (you don't have to change the version of the HWICAP this time) and finally create a DCS peripheral in the same way as for the Virtex 2 Pro.

In the new Design Suite you use Xilinx SDK to create software applications for your board. To start SDK, choose "Export Hardware Design to SDK" under "Project" from within XPS. Create a new software project and add all the files in "swReconfiguration" to your project. You should link to the files but don't copy them because they are updated by the TLUT flow when you change your design. Copy the file "xhw-icap_clb_lut_replacement.h" from "examples/xorExample/xpsV13/swReconfiguration" to your project. This fixes a bug in the HWICAP drivers for Virtex 5. You may want to increase heap and stack size in the linker script of your software project.

4.3 Troubleshooting

It is assumed that you have already successfully run the tests for the standalone TLUT tool flow (Section 3).

- *Throws "tmapXilinx.py: Command not found":*
Make sure you have set the environment variables correctly by doing '. source'. You must do this in the same terminal as you start XPS in and before starting XPS.
- *The output of minicom is garbage:*
Reboot your computer to fix the corrupted state the serial port is in. The corrupted state may be caused by running minicom and 'virtex2pro_test.sh' at the same time.
- *Xilinx Design Suite Errors:*
Make sure you are using a compatible version of the Design Suite. Version 9 for the Virtex 2 Pro and a newer one for the Virtex 5 (For the recommended versions see Section 2).
- *Reconfiguration does not happen, even for the example designs:*
Check the jumper settings of the XUPV2P or ML507 board.

Contents

The 'tlut_flow' folder contains a number of folders and three files: 'Makefile', 'README.md' and 'LICENSE'. The 'Makefile' is used to initially set up the tool flow. It will download some necessary third party tools and compile the used programs. More information about the license under which the TLUT tool flow is released can be found in 'LICENSE'.

The 'tlut_flow' folder contains the following folders: 'examples', 'tests', 'documentation', 'java', 'python' and 'third_party'.

- In the 'examples' folder you can find a number of designs that use the TLUT tool flow. You can run each of them by executing the Python script contained in the respective subfolder. You can run all examples at once (except AES) by executing the 'run_all.sh' script.
- In the 'tests' folder you can currently find one test, namely 'examples_test.sh'. Executing this script will run the 'examples/run_all.sh' script and compare the output to the expected output.
- The 'documentation' folder contains this document, and its Latex source files, and the Ph.D. thesis by Karel Bruneel which contains more information about the academic underpinnings of the TLUT tool flow.

The remaining folders contain binaries, source code and wrapper scripts. You may want to edit these when doing advanced experiments.

- The 'java' folder contains the Java source and binary files of a simple technology mapper and an adapted parameterized version of this technology mapper, the TLUT technology mapper.
- After setup, the 'third_party' folder will contain ABC, the logic synthesis and technology mapping tool of the university of Berkeley, and Aiger, a tool used to handle textual (.aag) and binary representations of and-inverter-graphs (.aig).
- The 'python' folder contains the high level Python scripts that are used to interface with the tools in the other folders. It contains three files: 'fast_tlutmap.py', 'genParameters.py' and 'mapping.py'. The 'genParameters.py' script will extract the parameters from the annotated VHDL file. The 'mapping.py' script contains high level wrappers that call the different mappers and a number of utility programs. These 2 last scripts are best used as given.

The 'fast_tlutmap.py' script contains the main code in the 'run' function. This function can easily be adjusted to meet the user's needs. The run function mainly consists of 2 steps: synthesis and technology mapping. The synthesis step converts the VHDL file into a logic circuit in blif format using Quartus II. The technology mapping step will map the logic circuit to a circuit with LUTs. Three different technology mapping tools are included to allow an easy comparison: the simple mapper, the TLUT technology mapper and the ABC mapper. The first two are not as optimized as ABC. For example 'area recovery' is not yet completely implemented. The ABC tool flow is an academic framework that does include a large number of such optimizations. Several commercial tools are based on this technology mapper.

The 'run' function also contains some intermediate steps:

- the conversion of the blif format into the aag format, needed for the Java mappers
- extraction of the parameters out of the annotated VHDL file