Integrating Flopoco operators into Zynq's ZedBoard using GPIO

Antoine Martinet

June 13, 2014

Contents

Introduction

This tutorial aims to help you integrating custom peripherals into Zynq's ZedBoard, and especially peripherals generated using INRIA's Flopoco framework. I will tell you about how to build the hardware, then export it to a driver form, and program it using c routines generated by the Xilinx tools. Please note that I am not going to introduce you to AXI-compatible peripherals (which seems to be much better when your process grows in size and complexity). The only thing we are going to perform is plugging peripherals directly on the GPIO port.

1 Main requirements

A background in VHDL should be fine to follow this tutorial.
You first need a working Xilinx ISE suite, with the right license that comes for the PlanAhead tool. I used version 14.7, note that the GUI layouts may vary through versions, so you might be a little bit lost if you don't have this one. You might have to install usb-jtag drivers properly, according to your Linux distribution. The last version of Flopoco is also required (and remember that it comes with sollya, gmp, and gmpxx c++ libs).

2 Build the hardware

First, you need to design your hardware. For this purpose, we will use Flopoco to design the peripheral we want (for this example, I will use a simple 16 bits adder). Then, we will use Xilinx's PlanAhead utility to interface our peripheral with ZedBoard.

2.1 Building a peripheral using Flopoco

If you want to integrate your own handbuild peripheral or to generate a peripheral using other software, you can skip this section.

I will only tell you about the few commands I use from Flopoco, for further information, see Flopoco's documentation. The only one command you need to use is:

```
$ flopoco -outputfile=int_adder_16.vhdl -pipeline=no IntAdder 16
```

Then, we will have to modify the vhdl produced by Flopoco in order to make it compatible with XPS integration. Indeed, you can't plug a peripheral on GPIO unless it's input and output ports are the same width. That is, if you have 2*16 bits input and 16 bits output (in our situation), you will have to pad 16 bits of zeros in front of your 16 actual bits of output.

So, when you edit int_adder_16.vhdl with your favorite text editor, you will see this:

```
--------------------------------------------------------------------------------
--                            IntAdder_16_f400_uid2
--                       (IntAdderClassical_16_f400_uid4)
-- This operator is part of the Infinite Virtual Library FloPoCoLib
-- All rights reserved
-- Authors: Bogdan Pasca, Florent de Dinechin (2008-2010)
--------------------------------------------------------------------------------
-- combinatorial

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
library std;
use std.textio.all;
library work;

entity IntAdder_16_f400_uid2 is
   port ( X : in  std_logic_vector(15 downto 0);
          Y : in  std_logic_vector(15 downto 0);
          Cin : in std_logic;
          R : out  std_logic_vector(15 downto 0)   );
end entity;

architecture arch of IntAdder_16_f400_uid2 is
begin
   --Classical
    R <= X + Y + Cin;
end architecture;
```

Figure 1: int_adder_16.vhdl

So you want to have two ports of the same width. To have this, just transform the two ports X and Y in one port XY of width 32. Change the size of port R too. Then, in the process of architecure section, replace the line:

```
R <= X+Y+Cin;
```

by the line:

```
R <= "0000000000000000" & ( XY(31 downto 16) + XY(15 downto 0) + Cin );
```

What you performed is splitting the port XY and adding the 16 MSBs with the 16 LSBs, and padding the rest of the port (the 16 R port MSBs) with zeros using the concatenation operator &

Now your file should look like this:

```
--------------------------------------------------------------------------------
--                            IntAdder_16_f400_uid2
--                       (IntAdderClassical_16_f400_uid4)
-- This operator is part of the Infinite Virtual Library FloPoCoLib
-- All rights reserved
-- Authors: Bogdan Pasca, Florent de Dinechin (2008-2010)
--------------------------------------------------------------------------------
-- combinatorial

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
library std;
use std.textio.all;
library work;

entity IntAdder_16_f400_uid2 is
   port ( XY : in  std_logic_vector(31 downto 0);
          --Y : in  std_logic_vector(15 downto 0);
          Cin : in std_logic;
          R : out  std_logic_vector(31 downto 0)   );
end entity;

architecture arch of IntAdder_16_f400_uid2 is
begin
   --Classical
R <= "0000000000000000" & ( XY(31 downto 16) + XY(15 downto 0) + Cin );
end architecture;
```

Figure 2: int_adder_16.vhdl

Now your peripheral is ready to be plugged to ZedBoard's GPIO.

2.2 Create a new PlanAhead project

Now you've got a peripheral with an input port and an output port of the same width, we can start building our hardware. First, you need to launch PlanAhead and create a new project.

1. Click on "Create New Project".
2. In the first window, click next.
3. You're now asked to name and locate your new project. May I advise you to have a directory in which to put all your PlanAhead projects. Select your project location, and name your project, for example, I use "int_adder". The box "Create project subdirectory" is checked. Leave it be.
4. Click "Next". The button "RTL Project" is selectionned. Leave everything as it is. Click "Next".
5. You get now to "Add Sources" screen. Set "Target language" to VHDL, as your entire design will be in VHDL, so it gives a better consistency. Actually, you could leave it to Verilog, as it will be recompiled and remixed by PlanAhead. As I don't know verilog very well, I prefer VHDL in case of improbable bug issues to fix.

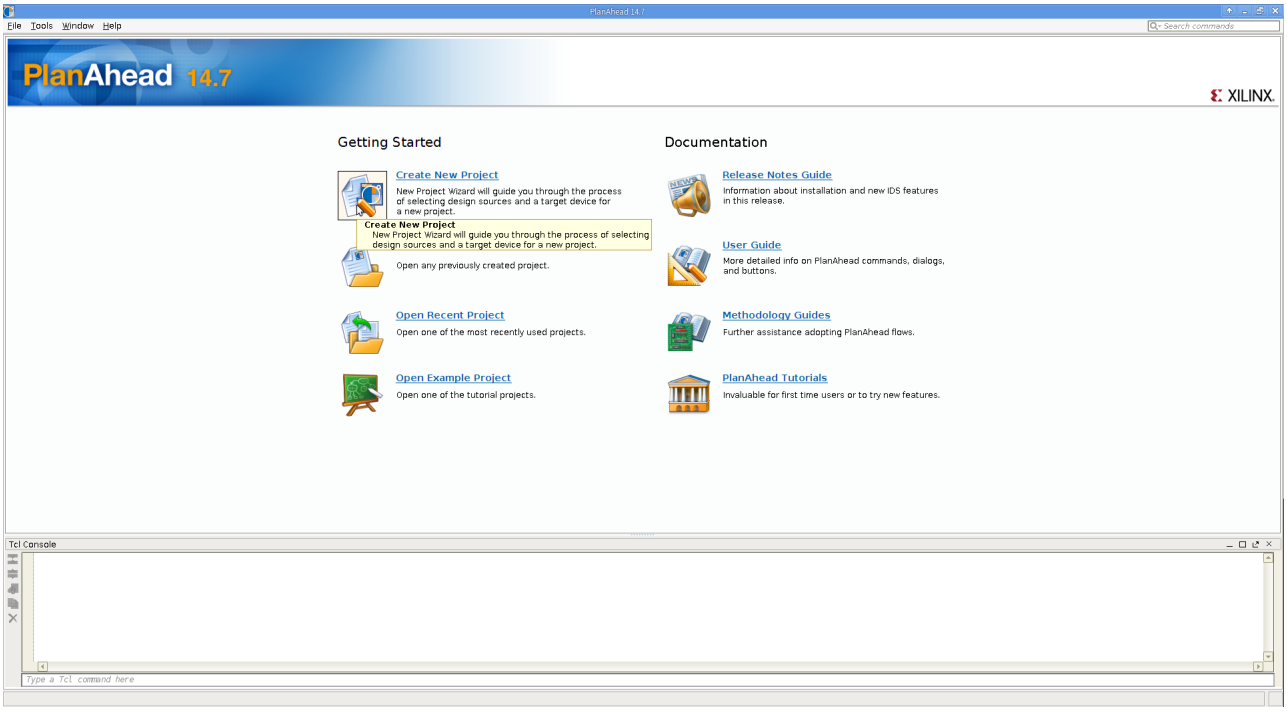

Figure 3: PlanAhead main window

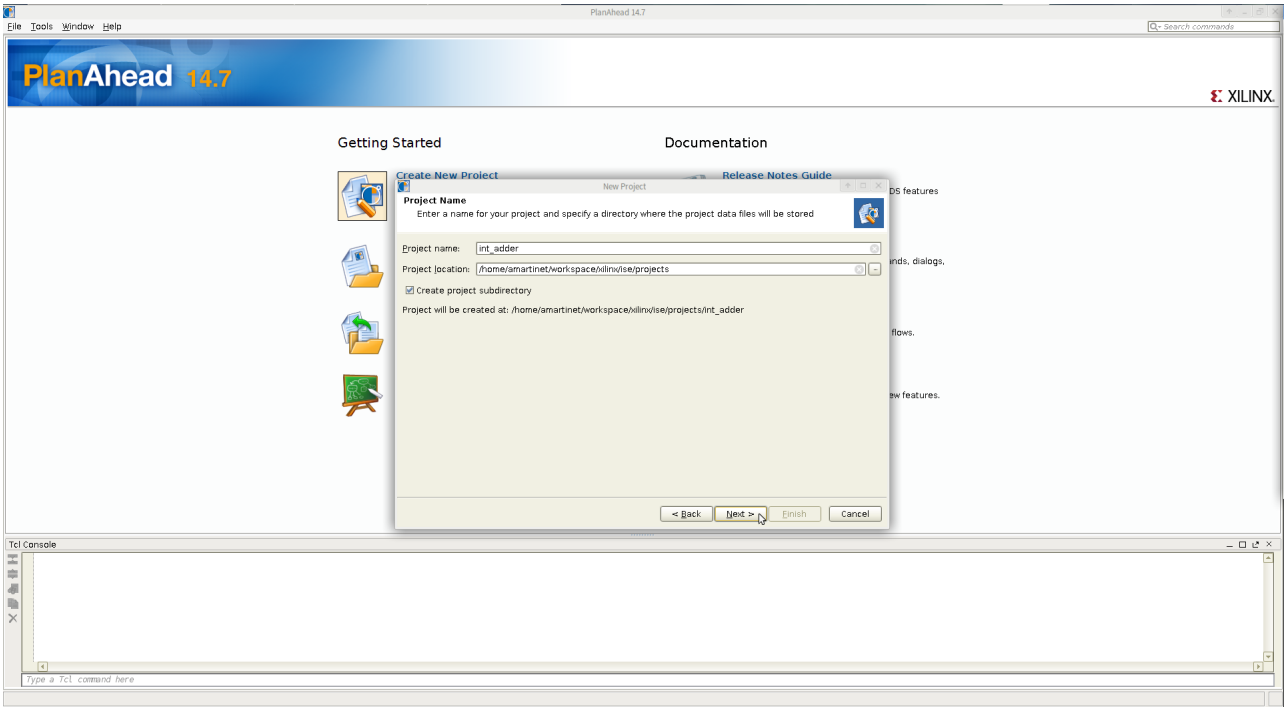

Figure 4: Create a new project

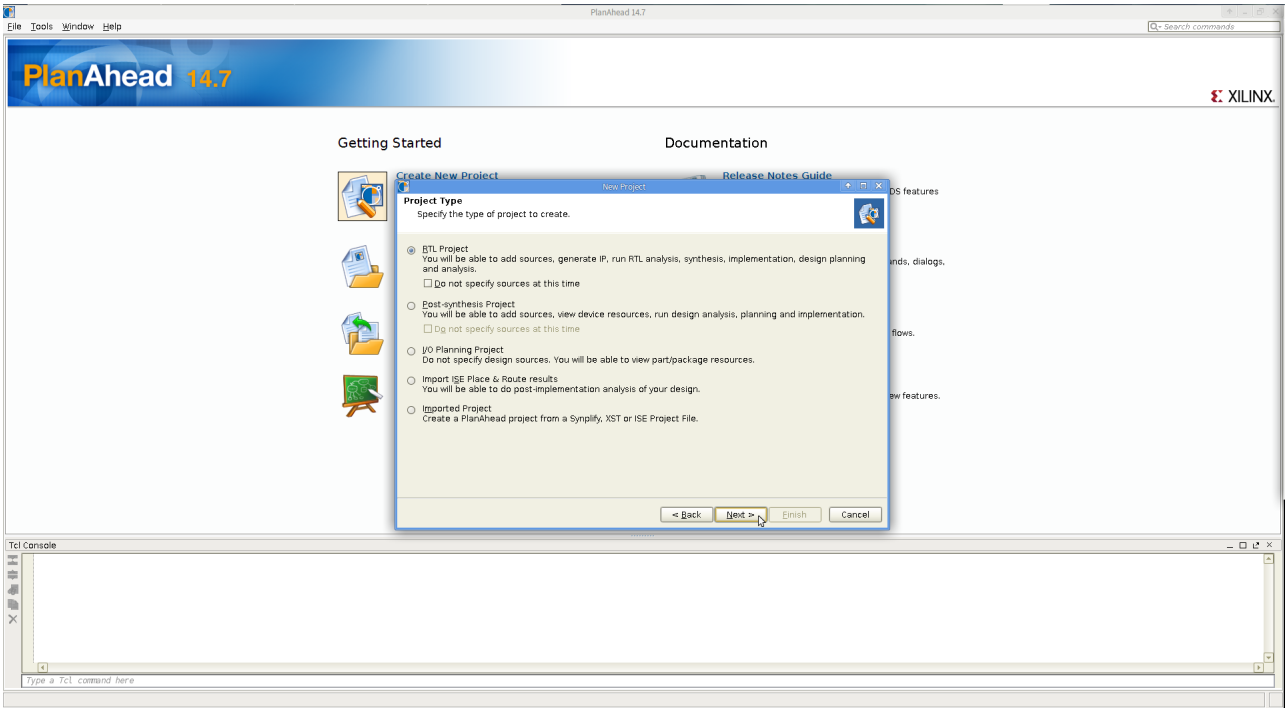

Figure 5: Create a new project

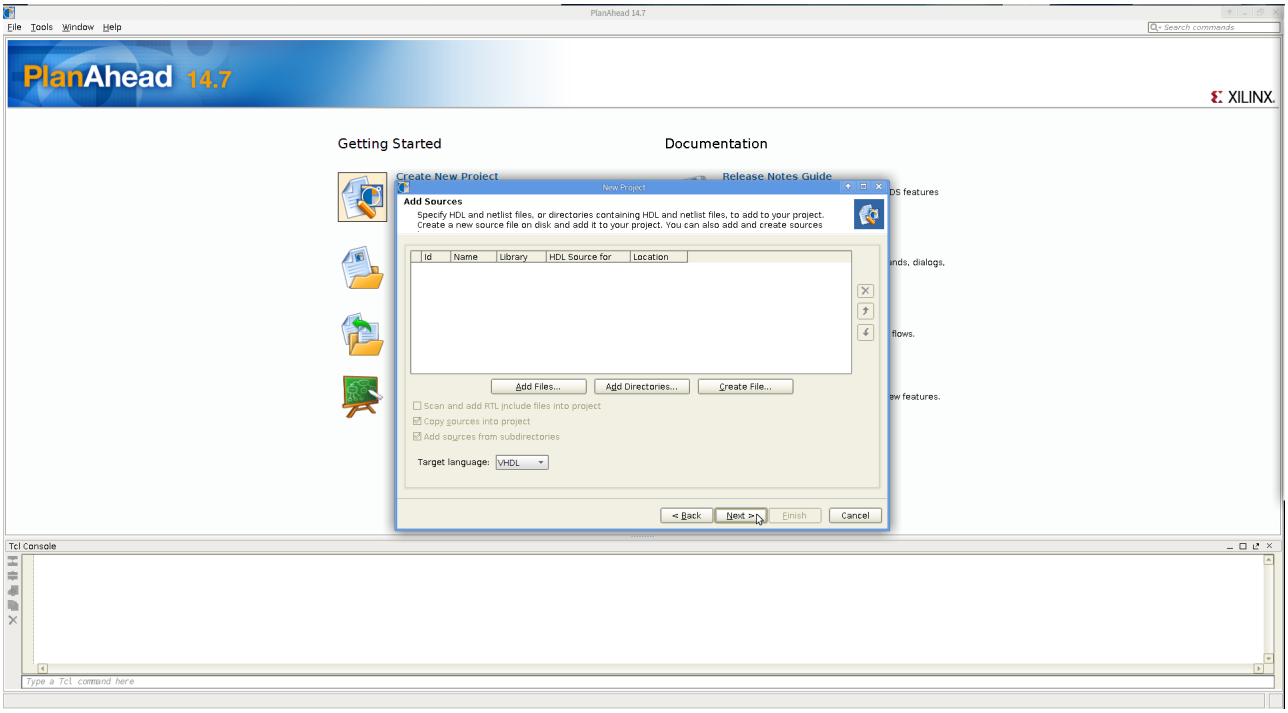

Figure 6: Create a new project

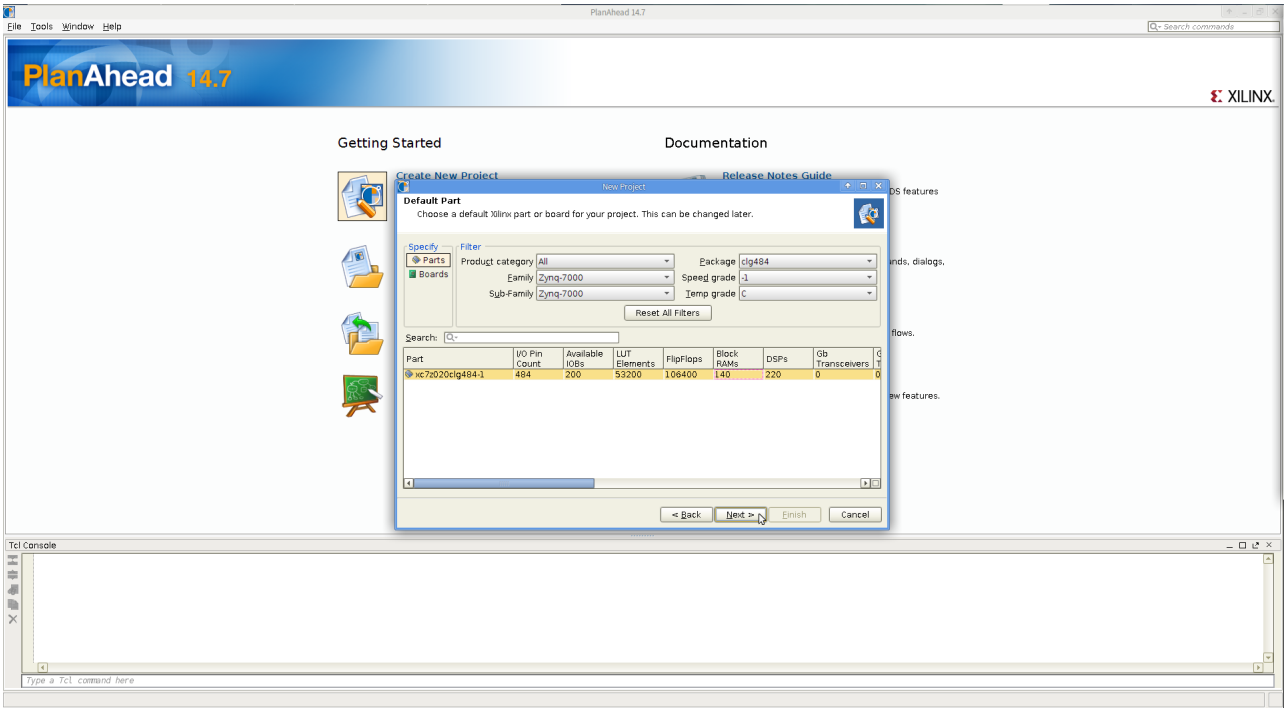

Figure 7: Create a new project

6. Click "Next".

7. You get to "Add Existing IP" screen. Nothing to do. Click "Next".

8. You can here add constraints. But we do not have any constraint file to add for the moment. Click "Next".

9. Then you get to "Default Part" screen. Here you will choose the hardware specifications. To choose your part, you may need some filters as the names are not user-readable.

   - In "Family" field, select "Zynq-7000"
   - In "Sub-Family" field, select "Zynq-7000"
   - In "Package" field, select "clg484" as it is the package type of your chip (see zedboard documentation).
   - In "Speed grade" field, select "-1" as you don't need to get in speed details for the moment.
   - Leave the "Temp grade" field for the moment.

   Then only one part should remain, xc7z020clg484-1. You will see on ZedBoard documentation that this is actually the name of the chip. Double-click on it or click "Next".

10. You get now to the project summary. Click "Finish" to create the new project. The project creation process may take a short time to complete.

Then we will get to the hardware design on XPS.

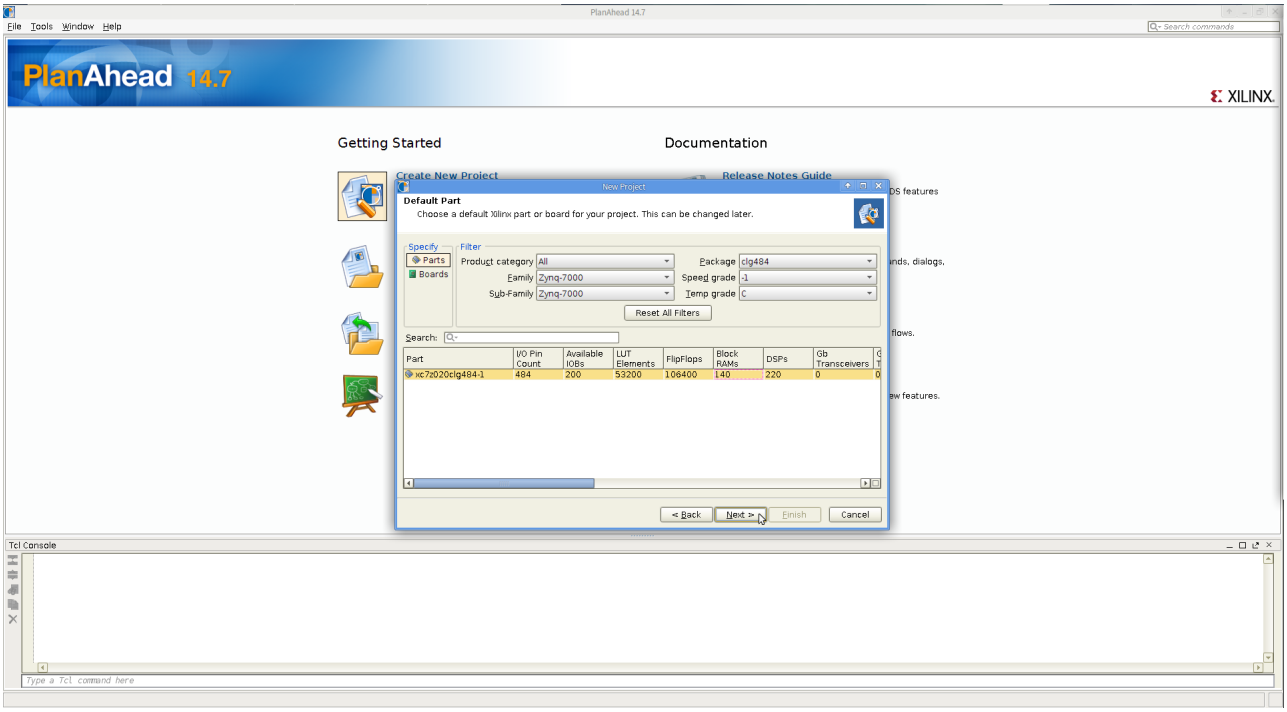

Figure 8: Create a new project

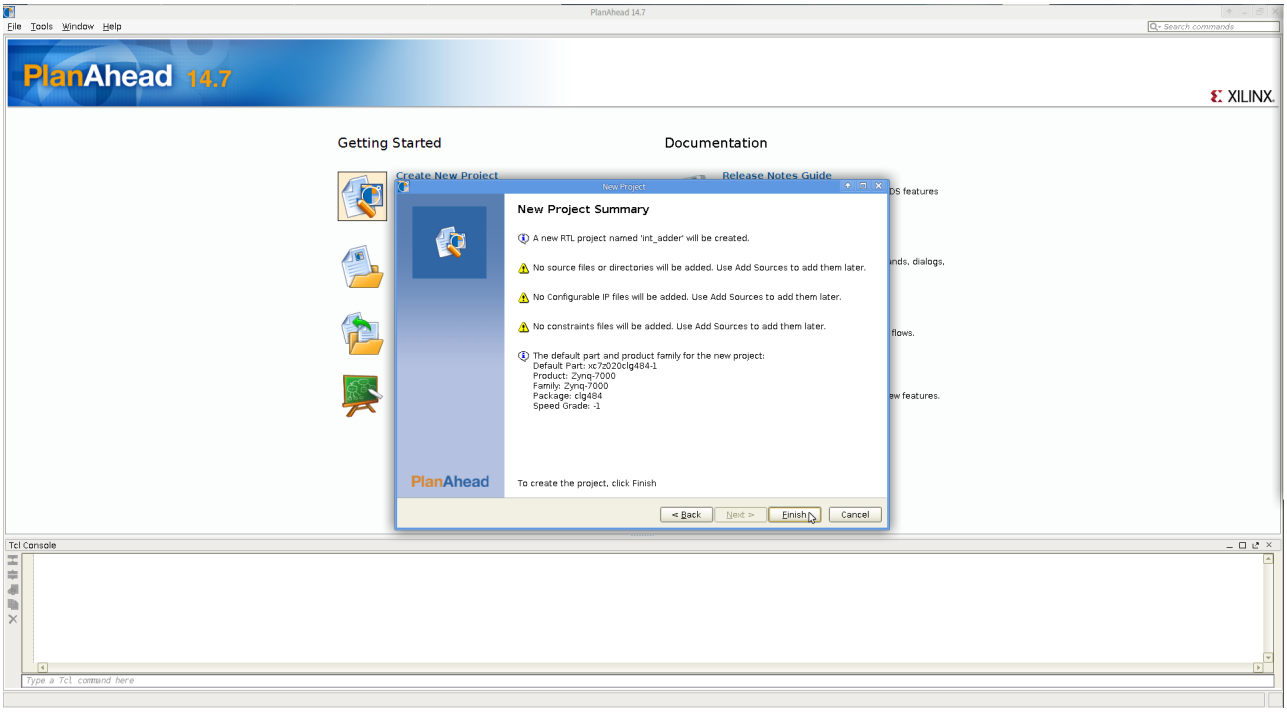

Figure 9: Create a new project

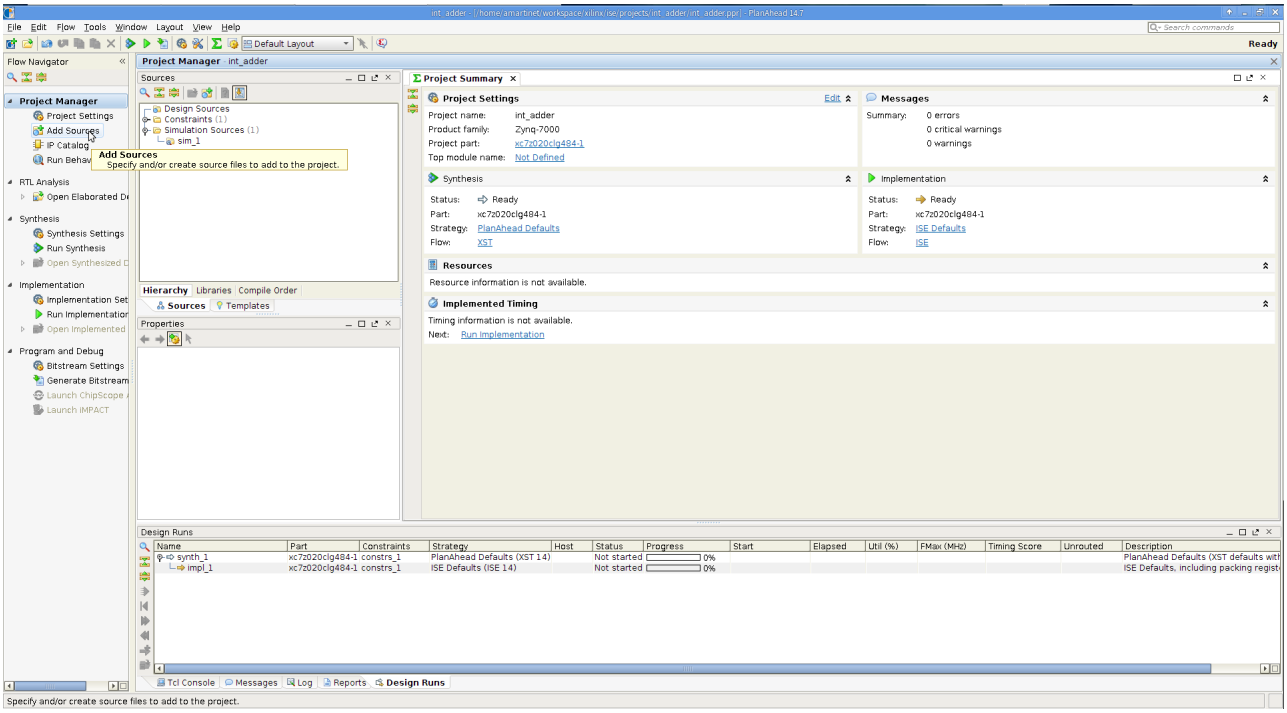

Figure 10: Add sources: create sub-design

2.3 Hardware design on XPS

Now you have to create an embedded source to design the hardware.

1. In the flow navigator (by default on the left panel), click "Add Sources" in the "Project Manager" menu.

2. In the "Add Sources" window that popped up, select "Add or Create Embedded Sources" and click "Next".

3. Then click "Create Sub-Design". Name your embedded source, for example, I used "system". Click "OK".

4. You will see the new file system.xmp pop as first line of the embedded sources table. Click "Finish". XPS Launches after a while.

5. XPS will ask you if you want to add a Processing System 7 instance to the system. Click "Yes". Wait for a while.

6. Then you have to load the basic configuration of the PS. Note that every settings are note important for the job we are performing, but some configuration features, as UART and USB, are useful to see the displayed results of preograms on a tty for example. In the zynq tab, click "Import".

7. In the User Template list, click on the "+" button, browse the file zedboard_RevC_v2.xml. You can download it here:
   http://zedboard.org/sites/default/files/documentations/zedboard_RevC_v2_XML.zip

8. click "OK"

9. XPS may ask you if you want to continue importing and updating MIO configuration. Click "Yes".

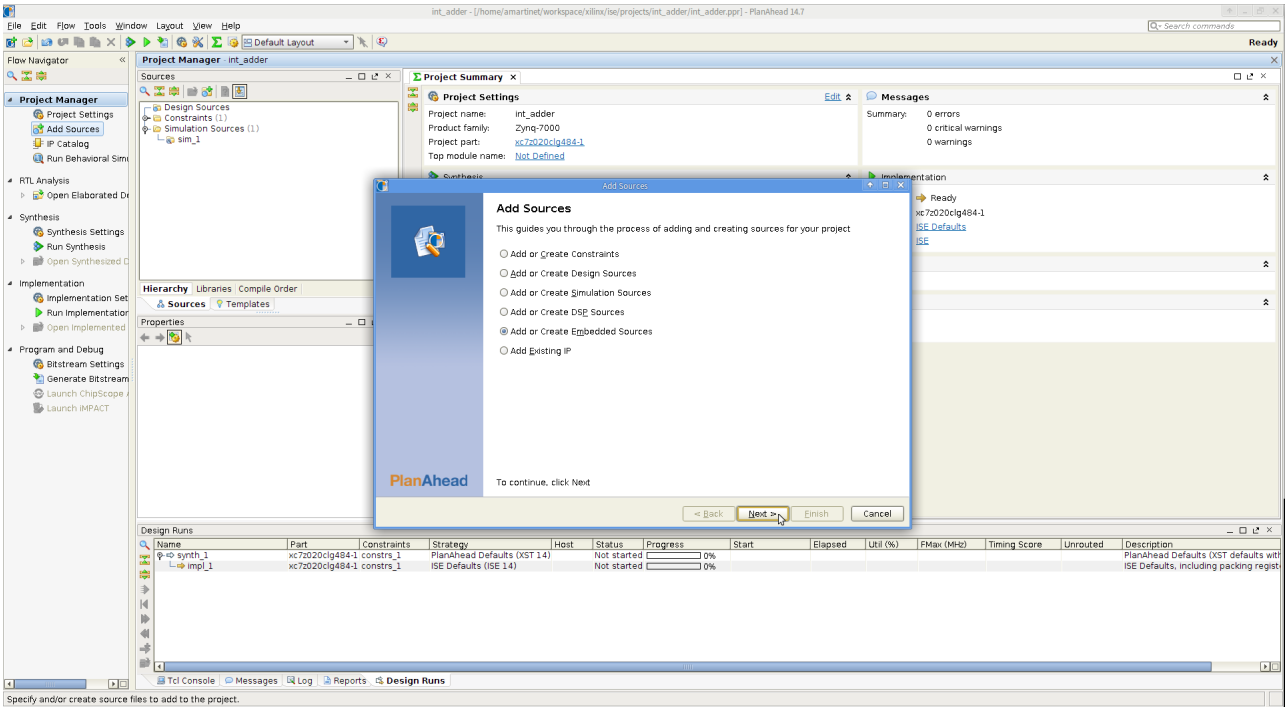

Figure 11: Add sources: create sub-design

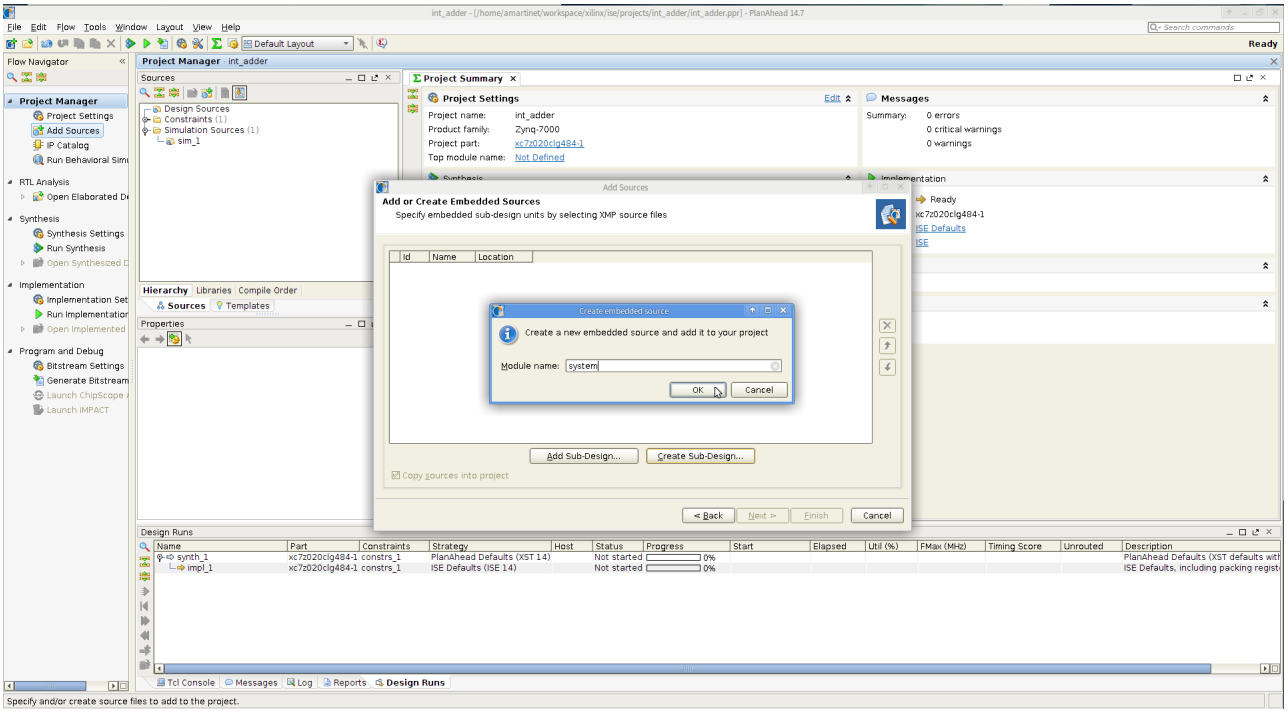

Figure 12: Add sources: create sub-design

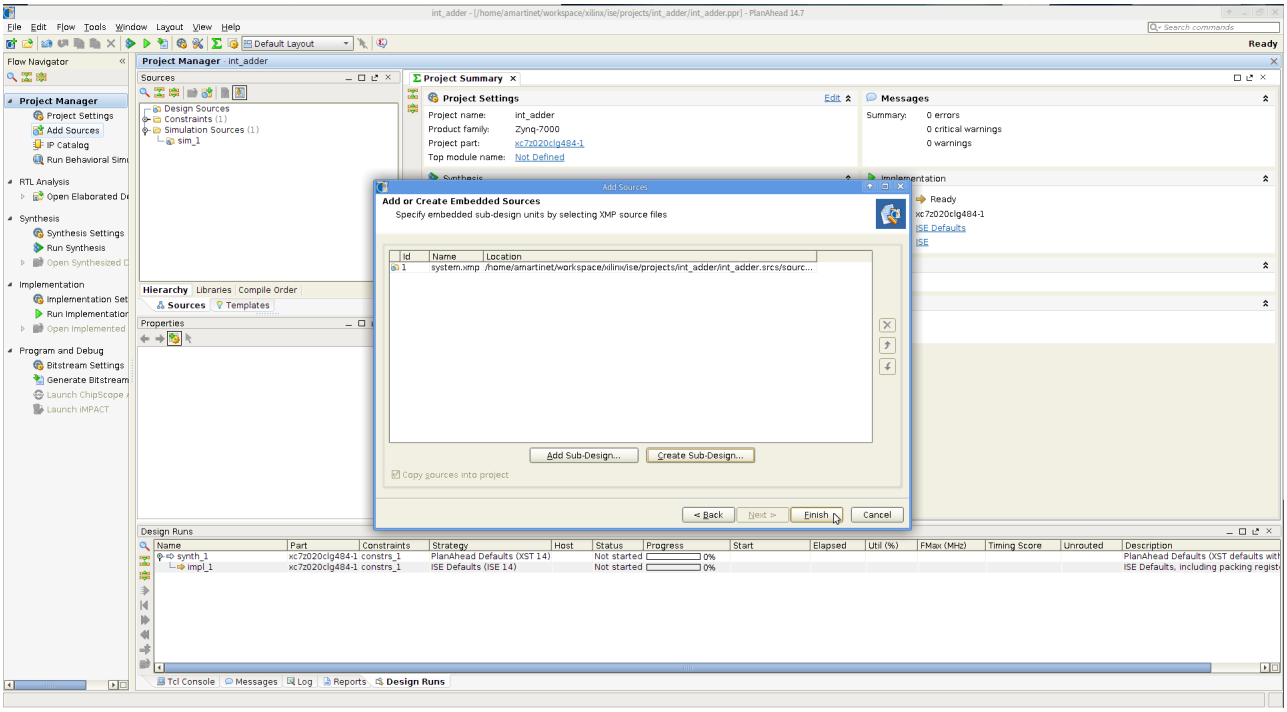

Figure 13: Add sources: create sub-design

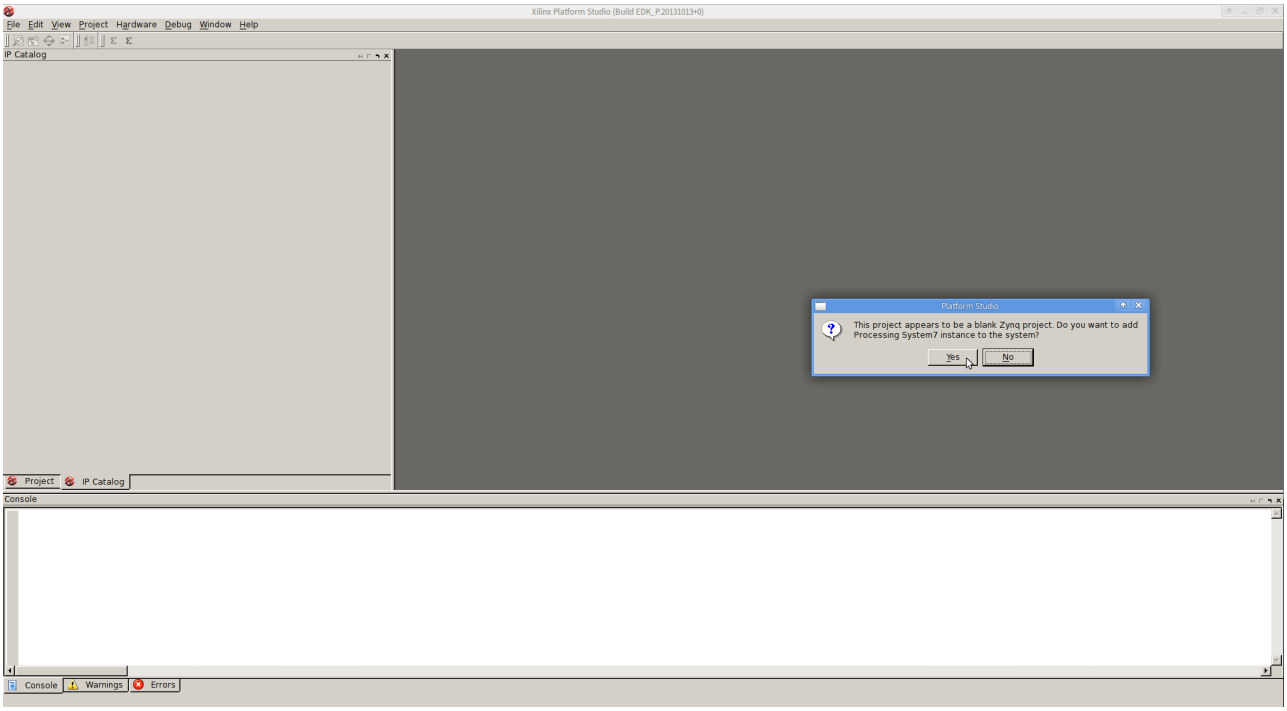

Figure 14: XPS launch

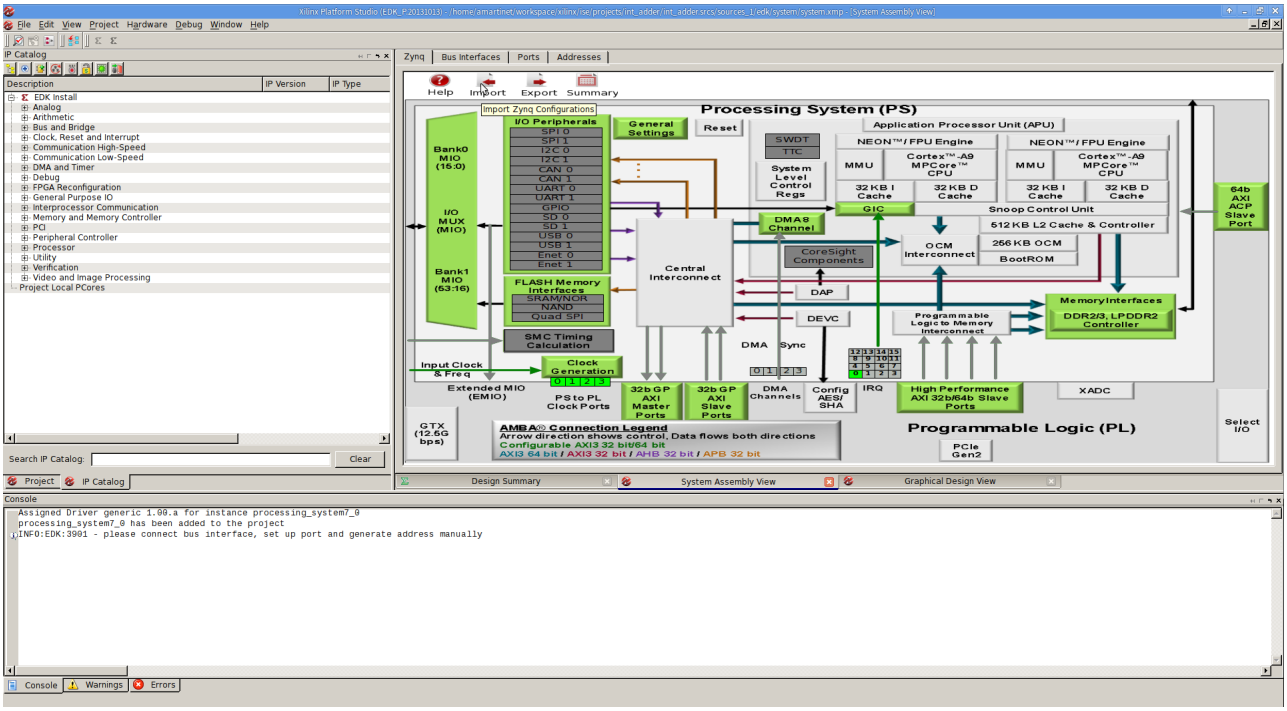

Figure 15: Import Configuration

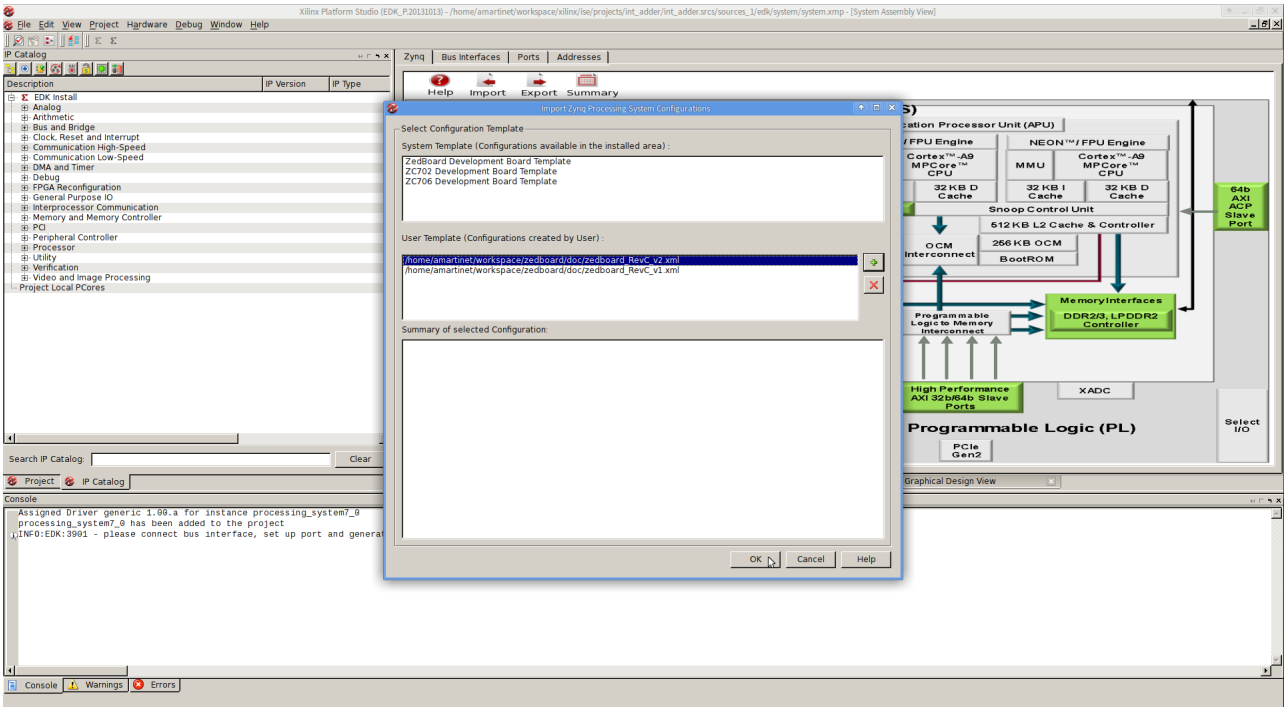

Figure 16: Import Configuration

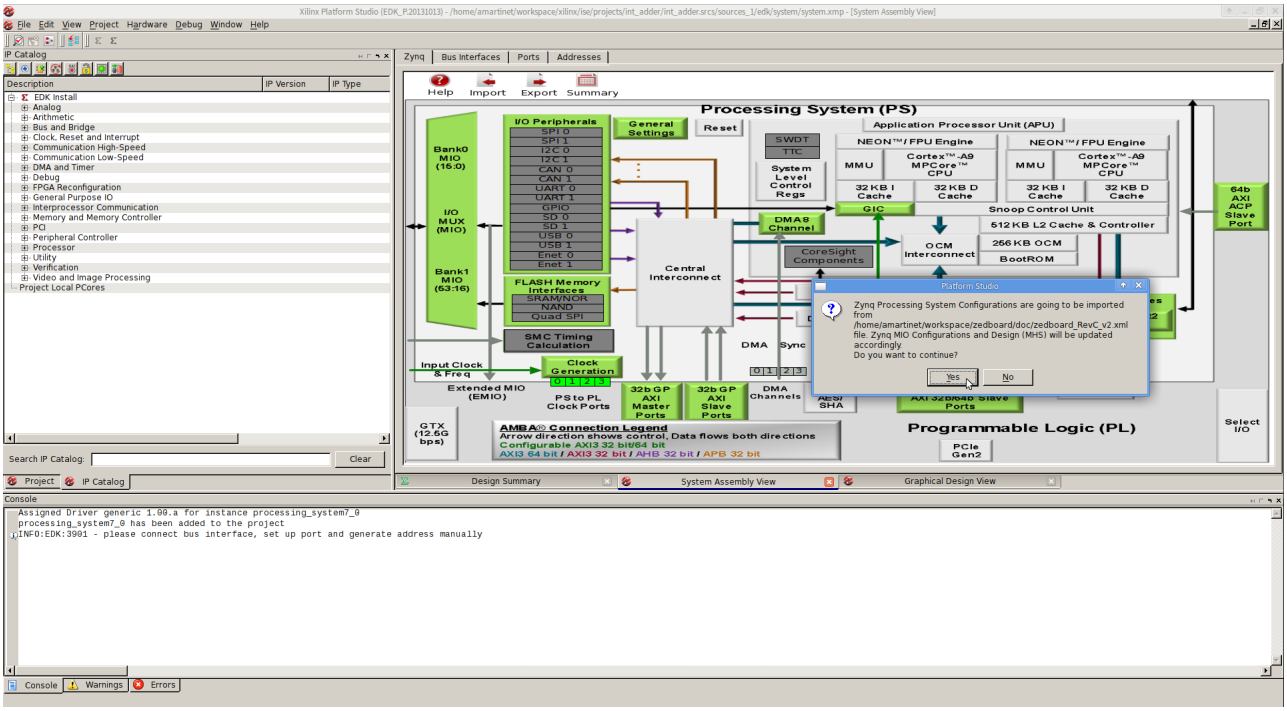

Figure 17: Import Configuration

10. Now, we are going to set the width of GPIO we are using. As we can read in the ZedBoard documentation, Custom PL peripherals have to be mapped through EMIO on the GPIO banks 2 and 3. For more info about GPIOs, you can go to appendix B: GPIO start guide .

    - So, in the Zynq tab, click on "I/O Peripherals" green box.
    - Then expand GPIO in Zynq PS Configuration Pannel.
    - check "EMIO GPIO (with)" checkbox and set the width to 32 bits (as your custom peripheral has 32 bits ports)
    - then click "Close".

11. Back to main XPS window, here you can already do a design rule check to see if everything is fine before breaking all the configuration. Go to the "Project" menu and click "Design rule check". You should have no warning, no error.

12. Go to menu "Hardware" -> "Create or Import Peripheral".

13. Getting in the wizard, click "Next" (there is nothing of interest on the first page).

14. In the "Select Flow" box, select "Import existing peripheral". Click "Next".

15. It's important to now that SDK will copy the source file of your peripheral into a location before using it. Indeed, you might want to modify it if you did a mistake, or if you want to change some features. So you can modify the project file without touching to the original peripheral (that could be a troubling point of view, but that's not the point of the discussion). So the wizard wants to know where to store the peripheral. By default, it selects the current project as a repository. Leave it do it well and click "Next".

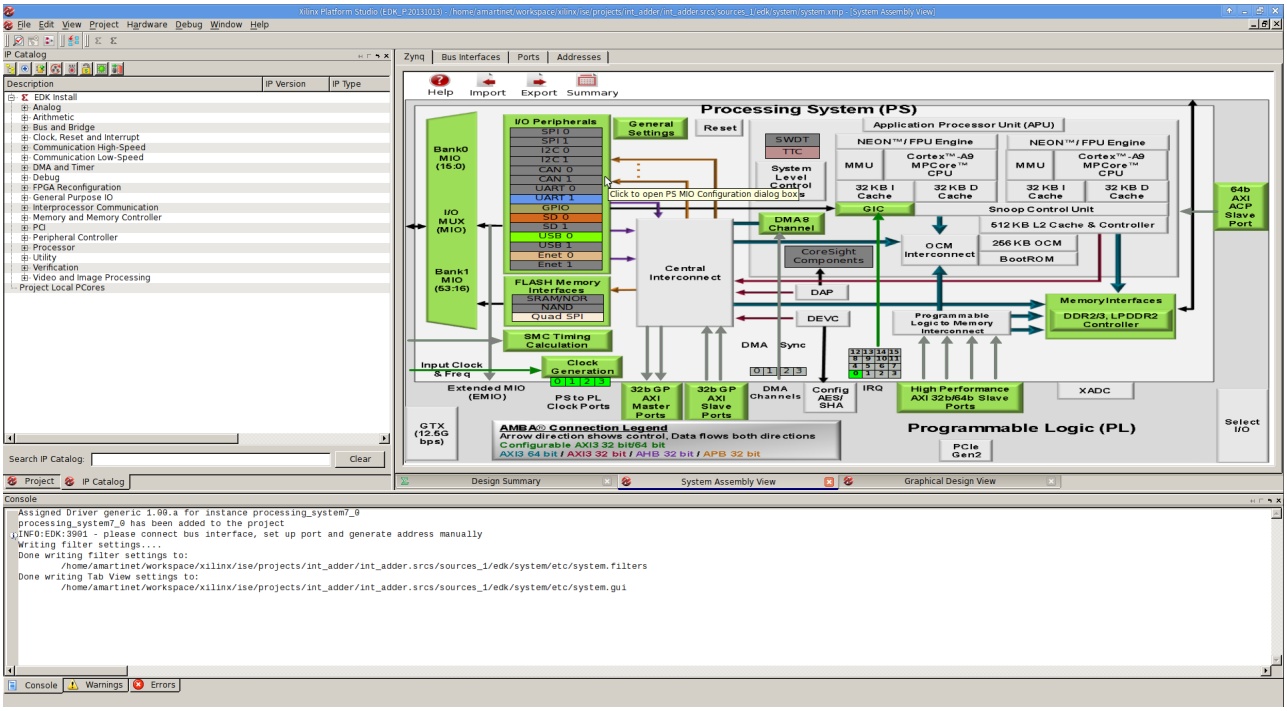

Figure 18: MIO/GPIO Configuration

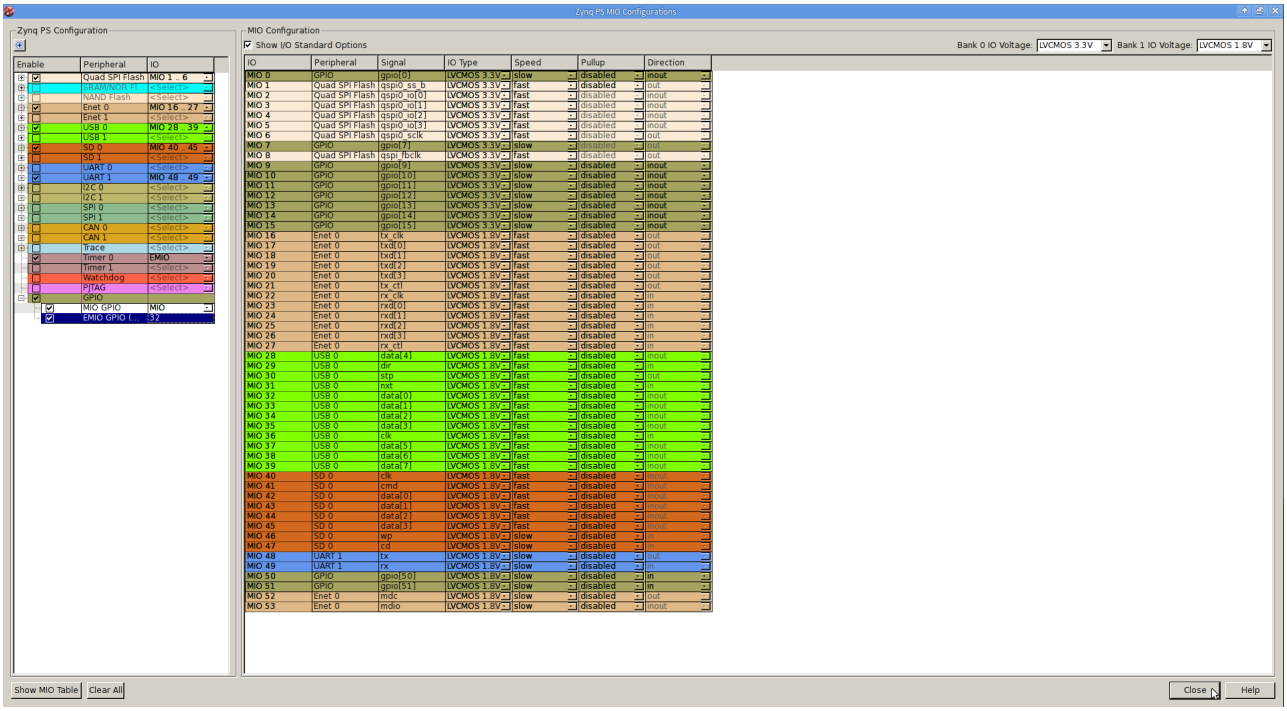

Figure 19: Import Configuration

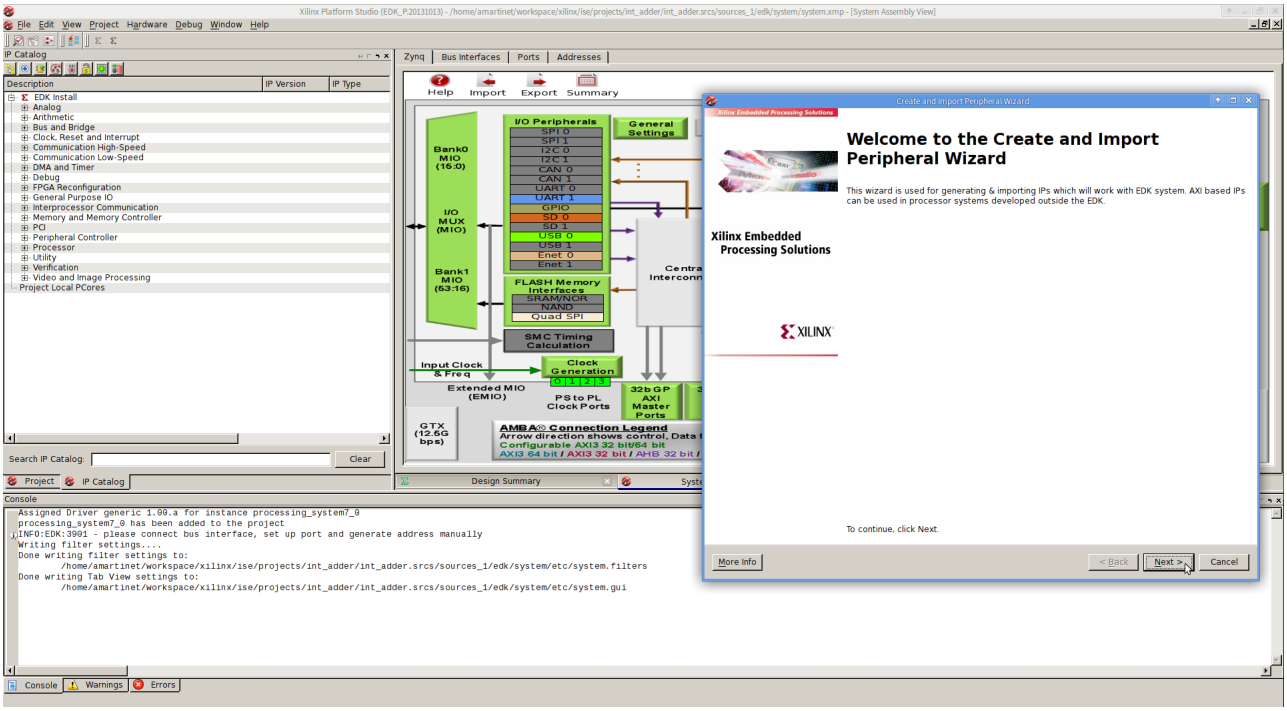

Figure 20: Import Peripheral

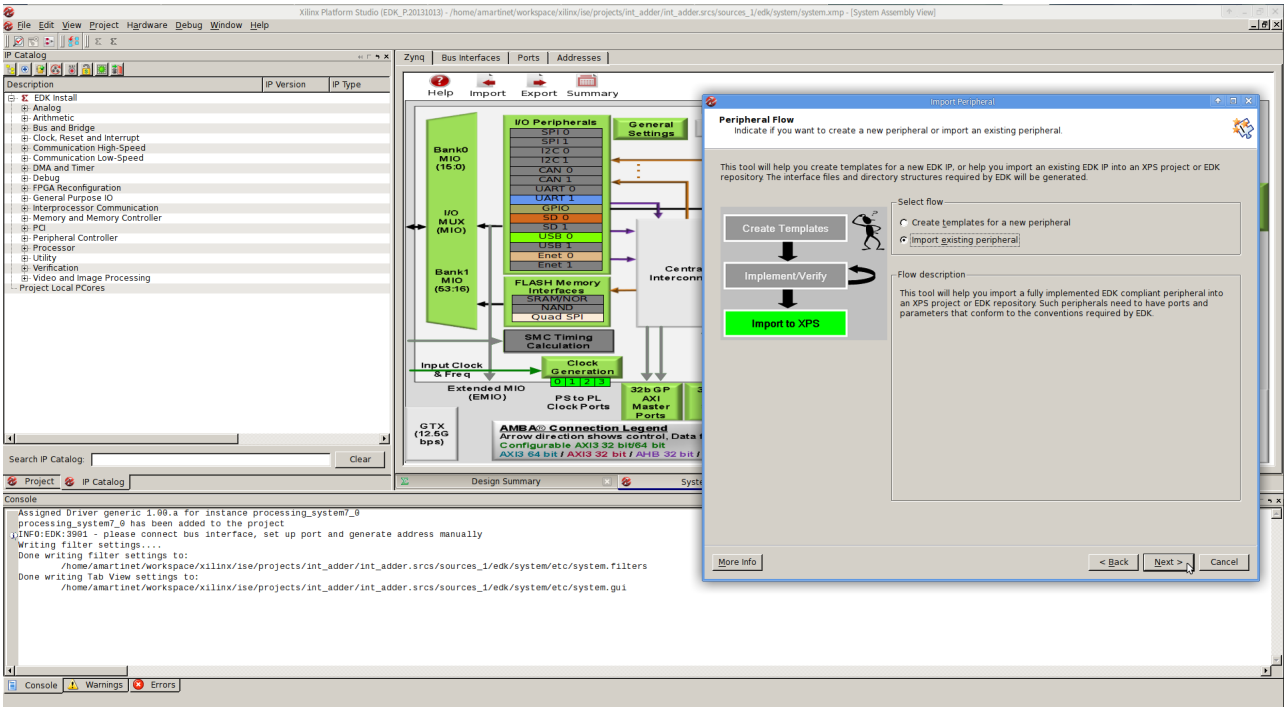

Figure 21: Import Peripheral

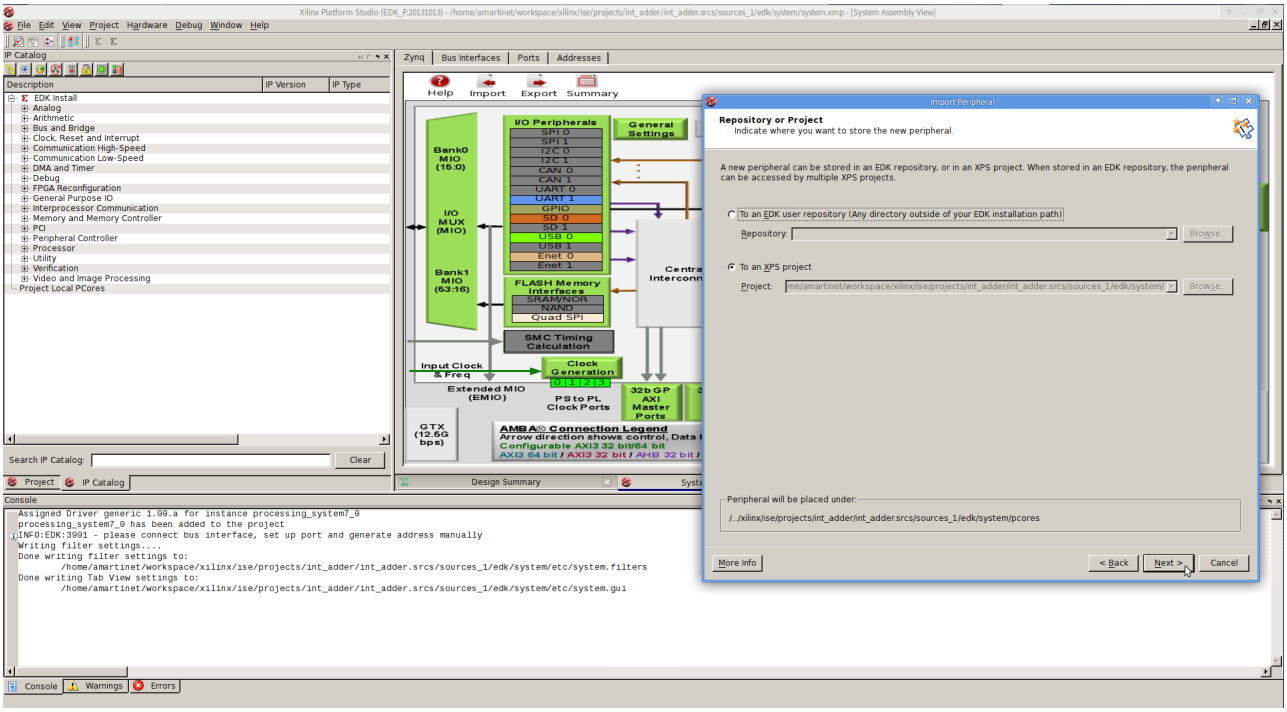

Figure 22: Import Peripheral

16. The wizard will now ask you the name of your new peripheral. Moreover, it will not be able to complete the wrapper synthesis if the name of your peripheral doesn't match the name described in the peripheral source file, and won't allow you to get further than the file loading. Fortunately, VHDL is non-case-sensitive, so you just have to put the flopoco-generated name and replace uppercase letters. Put "intadder_16_f400_uid2" as name for your new peripheral. The wizard provides also versionning features. Leave them be for the moment. Click "Next".

17. The wizard asks now for the source file type. Leave the "HDL source files" checked (default) and click "Next".

18. Now specify to the wizard where to find the peripheral. Select the last option, "Browse to your HDL source..." and click "Next".

19. Then click on the "Add Files" button and browse your vhdl component, so int_adder_16.vhdl. It appears as first line of the table. Click "Next".

20. The wizard will tell you that it doesn't manage to find the top design. This seems to happen because it uses the name of the peripheral (library) to infer the name of the top module. Here we've no problem, because we have only one file. In case of several files, this could easily be worked around moving up or down top module files. Indeed, as you can see it takes the first file of the list as top design file. Click "OK".

21. Now you're going to specify bus interfaces compatibilities. As flopoco's int_adder_16 doesn't follow any standard protocol (we don't need it for a simple adder), uncheck the main "Select bu interface(s)" box. Click "Next".

22. Now the wizard wants to know if your peripheral generates interrupts. Actually it doesn't, so uncheck "Select and configure interrupt(s)" box. Click "Next".

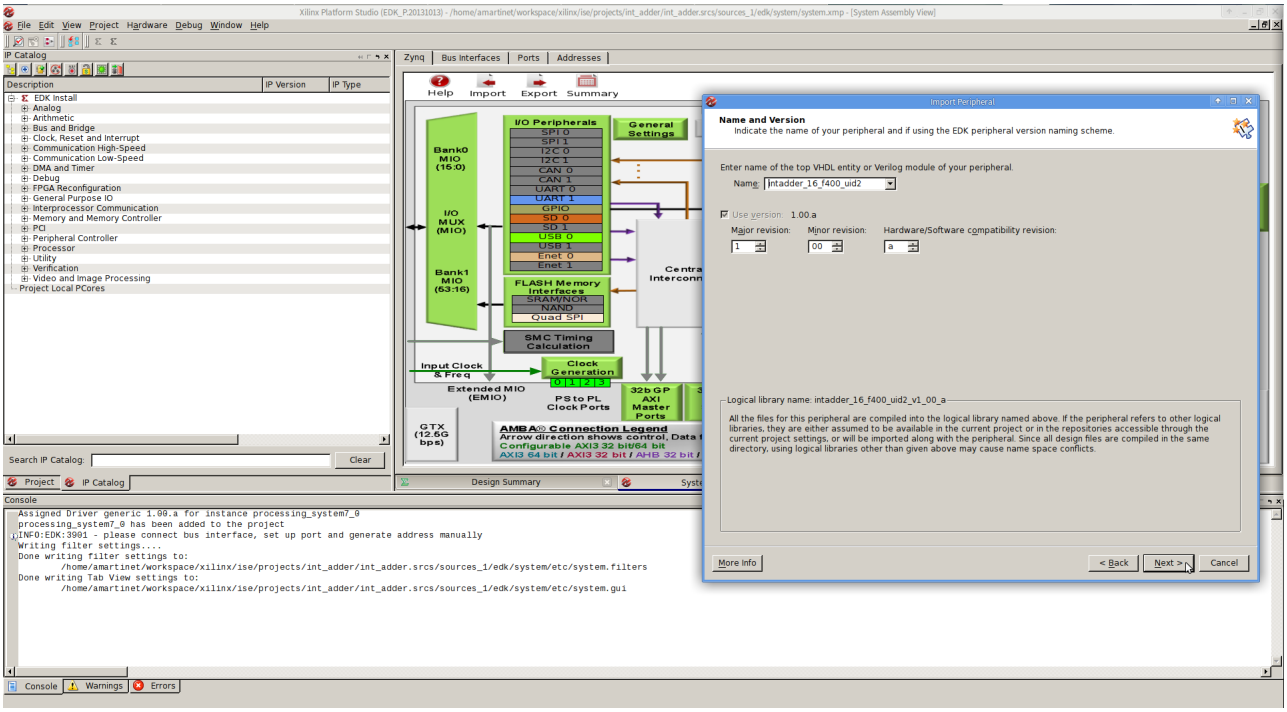

Figure 23: Import Peripheral

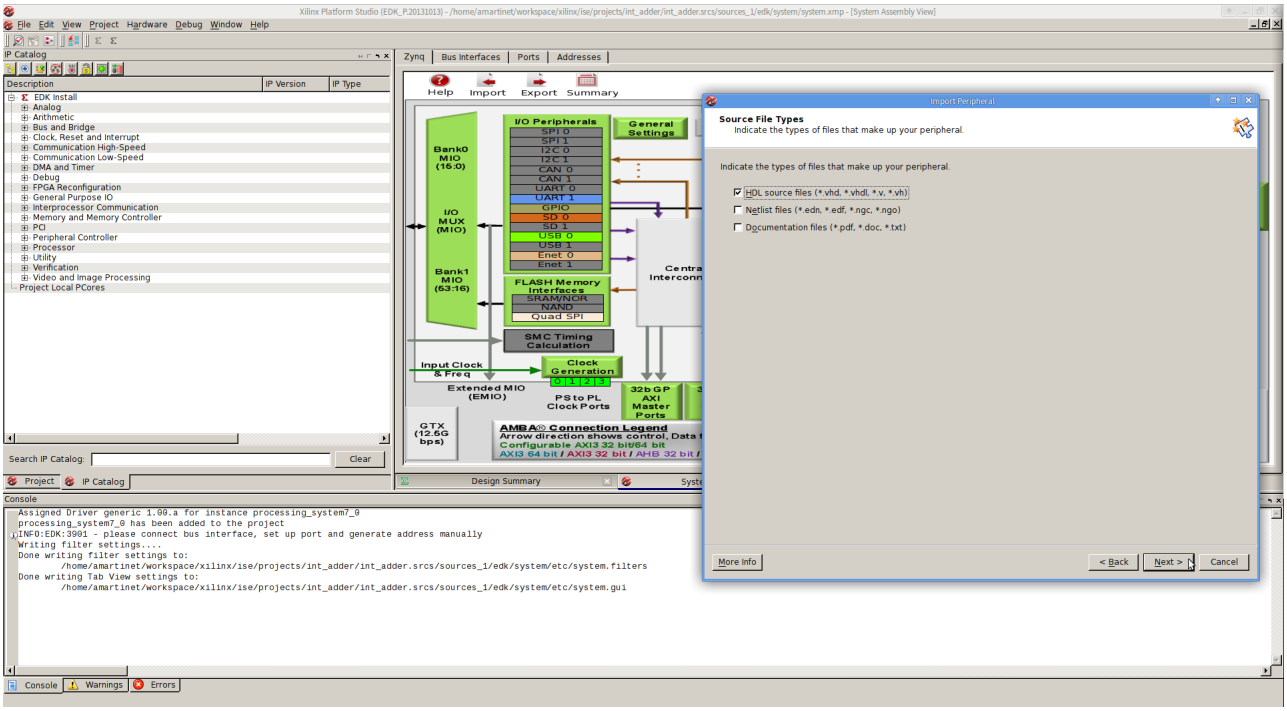

Figure 24: Import Peripheral

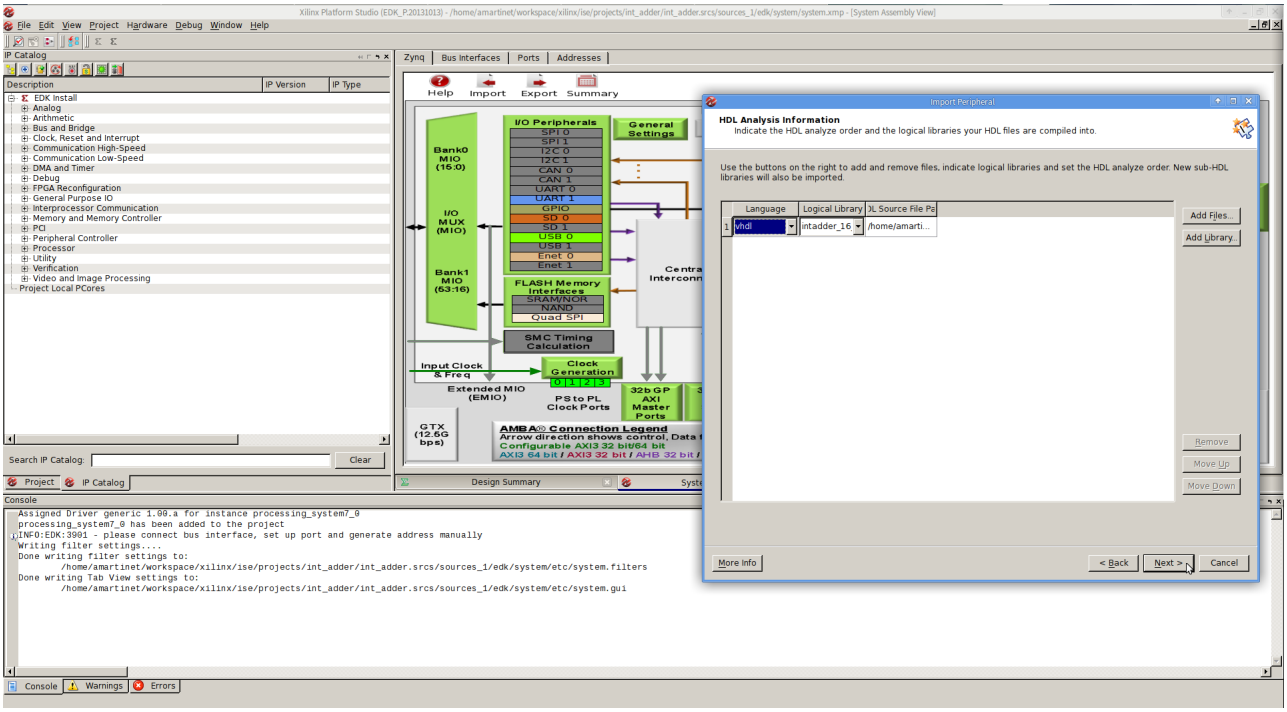

Figure 25: Import Peripheral

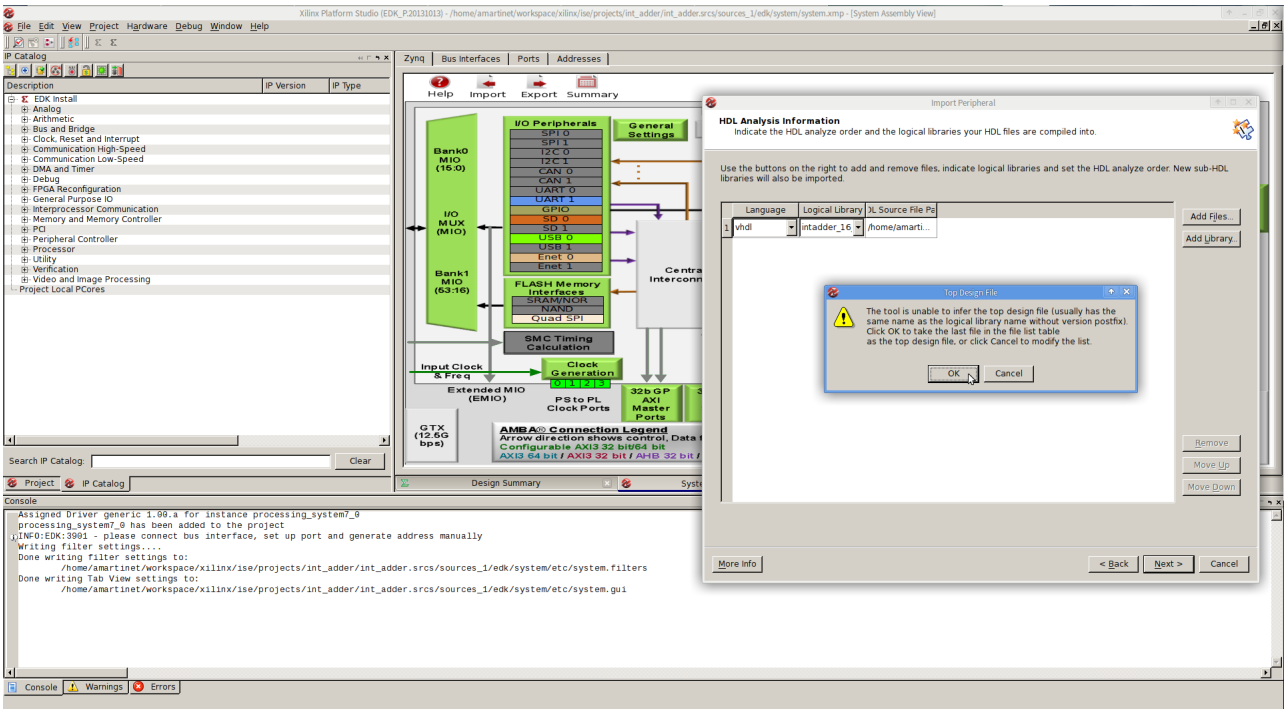

Figure 26: Import Peripheral

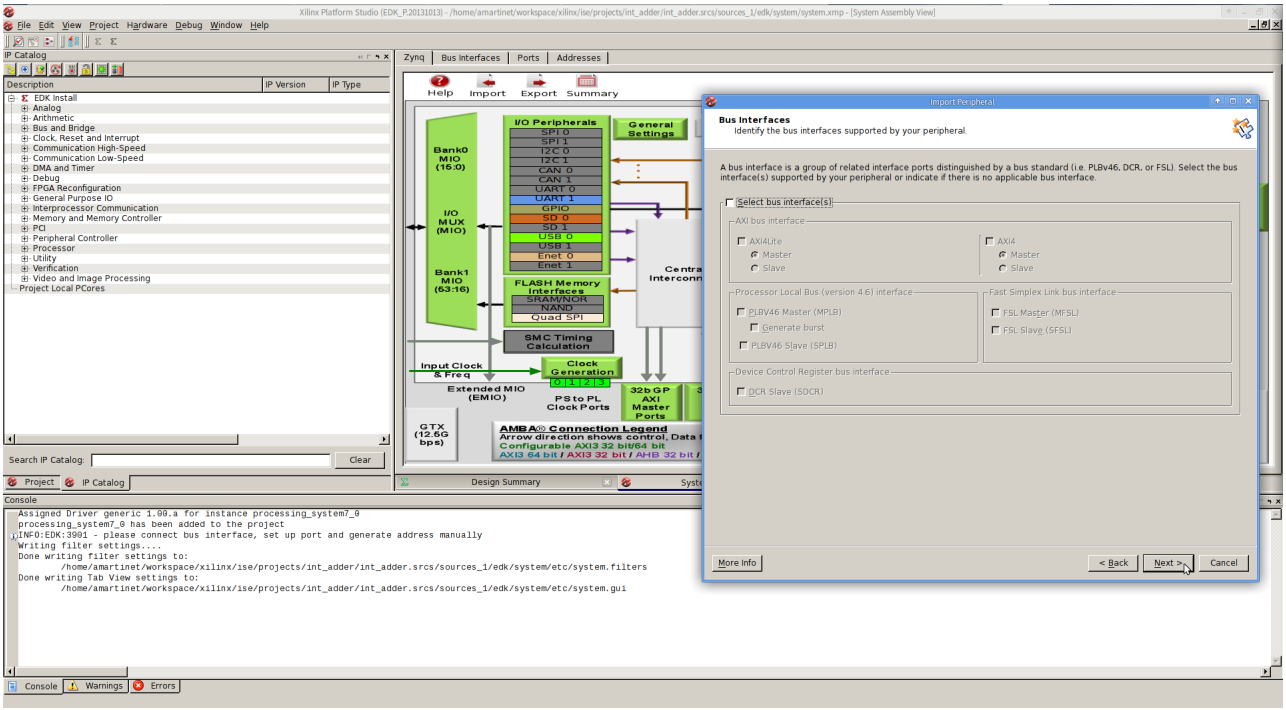

Figure 27: Import Peripheral

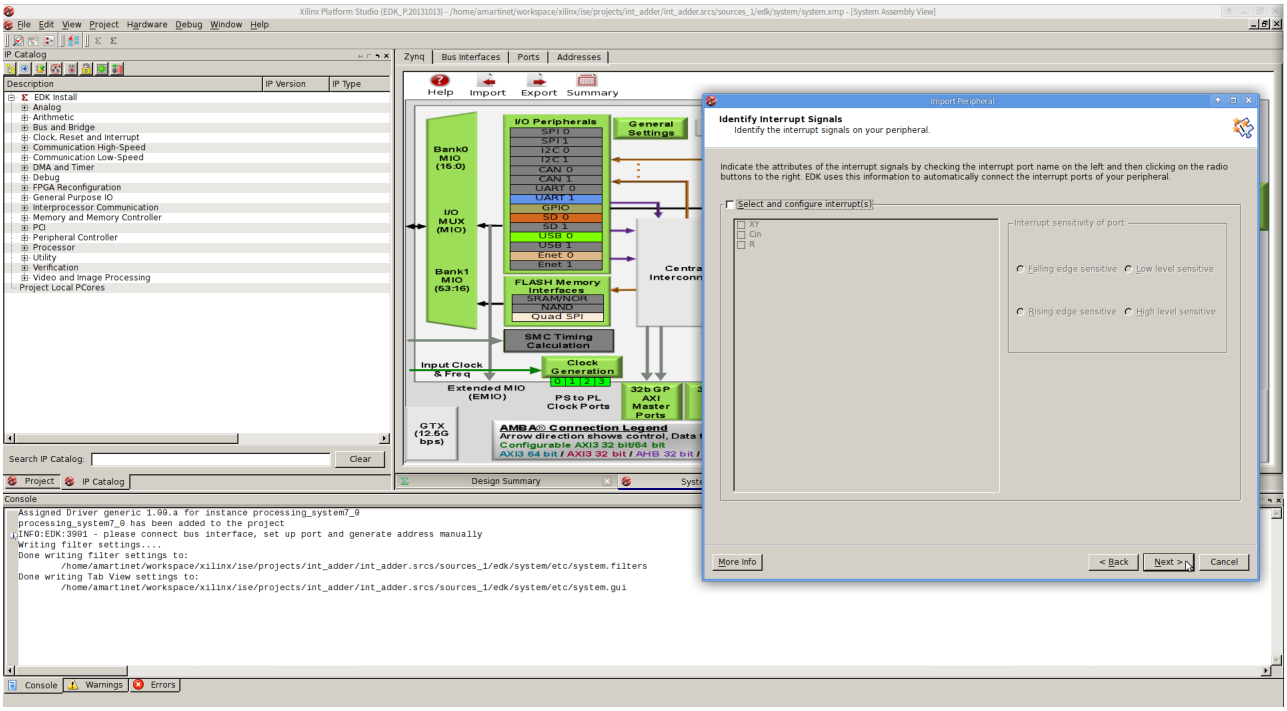

Figure 28: Import Peripheral

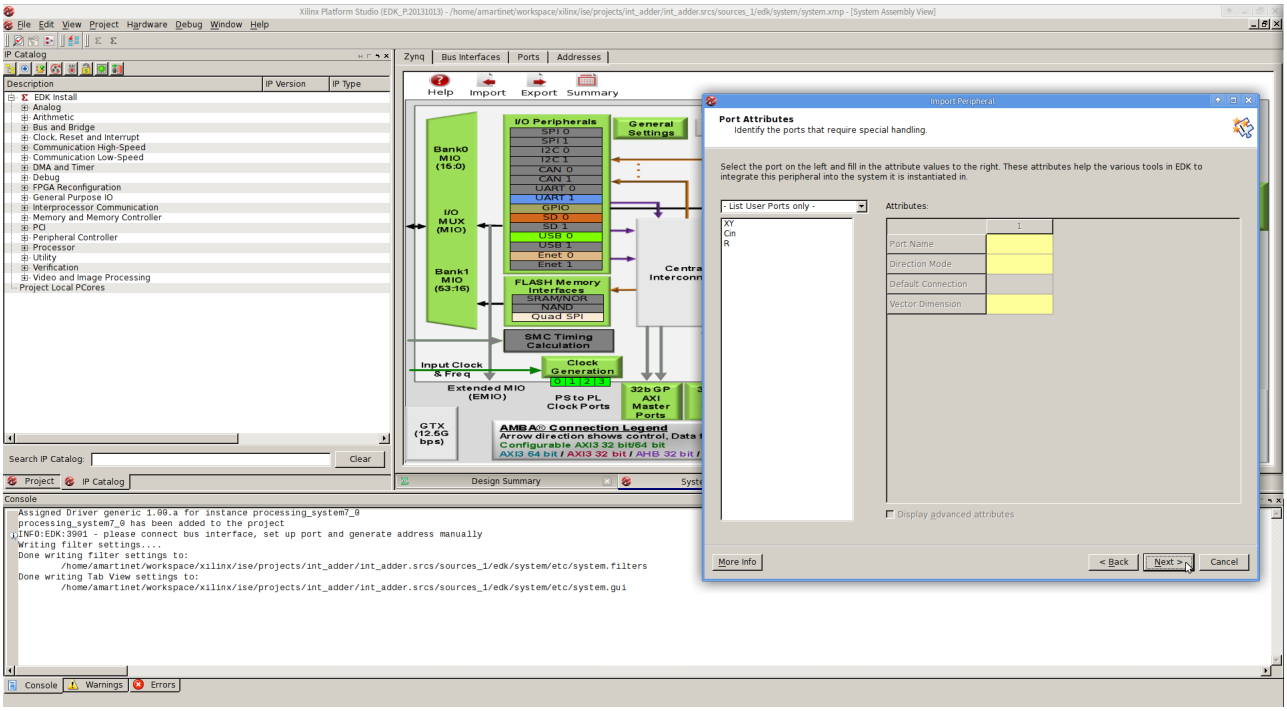

Figure 29: Import Peripheral

23. Then you can predefine where ports will be connected. This could be interesting when you use clock, reset pins, and some common standard features, which may not be automatically detected. As we don't follow any standard protocol, we are going to plug the peripheral manually. Leave everything be and click "Next".

24. Now you get into the last summary page when you can check if the wizard well understood what the hell you are doing. Click "Finish" to create your peripheral or get back to the previous steps if you feel unsatisfied. Wait for the GUI to entire rebuild itself.

25. You will see that a new field "USER" appeared in the "IP Catalog" Pannel. Expand it and double click your new peripheral to add it to the hardware design. Click "Yes" to the confirmation message. Then Click "OK" in the "Core Config" window that poped up.

26. Now we are going to plug the adder on the GPIO ports.

    - click the "Ports" tab.
    - expand "intadder_16_f400_uid2_0"
    - click on the "XY" Connected Port cell (pencil)
    - select "processing_system7_0" on the left drop down menu and "[GPIO_0]::GPIO_O" on the right one. Click the ok button.
    - click on the "R" Connected Port cell
    - select "processing_system7_0" on the left drop down menu and "[GPIO_0]::GPIO_I" on the right one. Click the ok button.
    - Cin is not important for this job. (we don't have any remainder to add). Leave it be for now.

27. Run a design rule check and try to fix any error (this should not occur).

28. Close XPS.

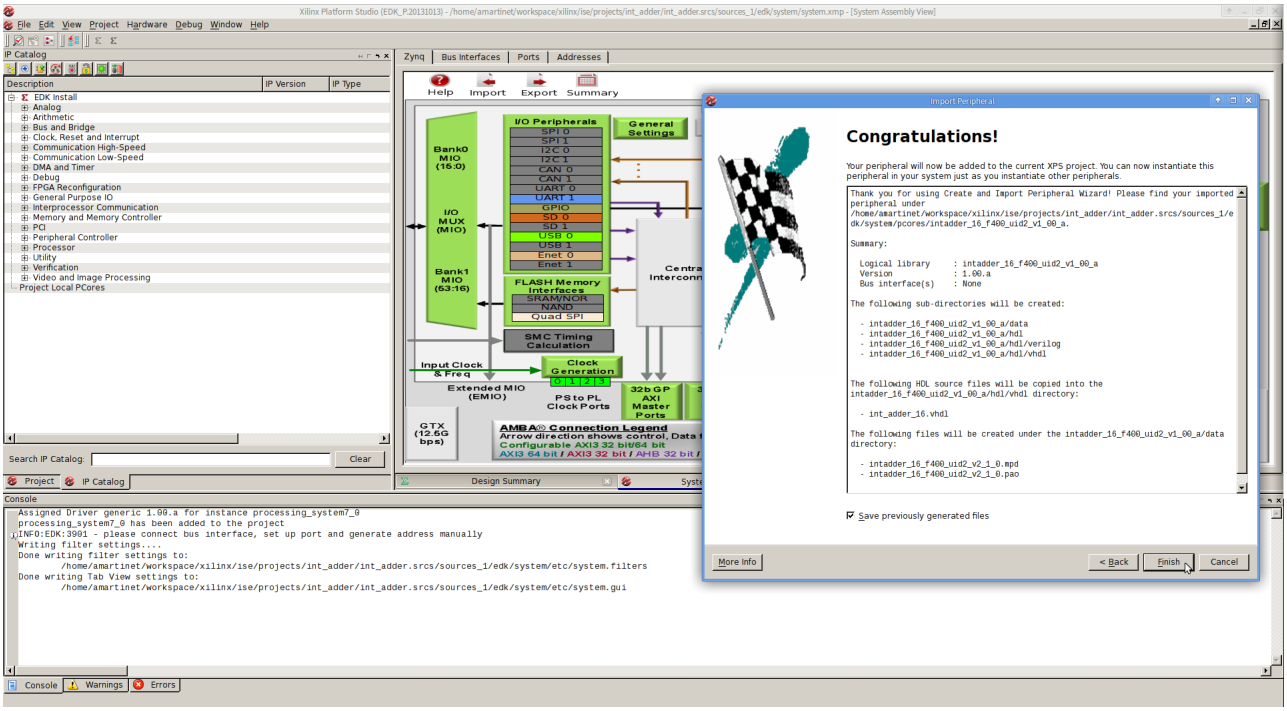

Figure 30: Import Peripheral

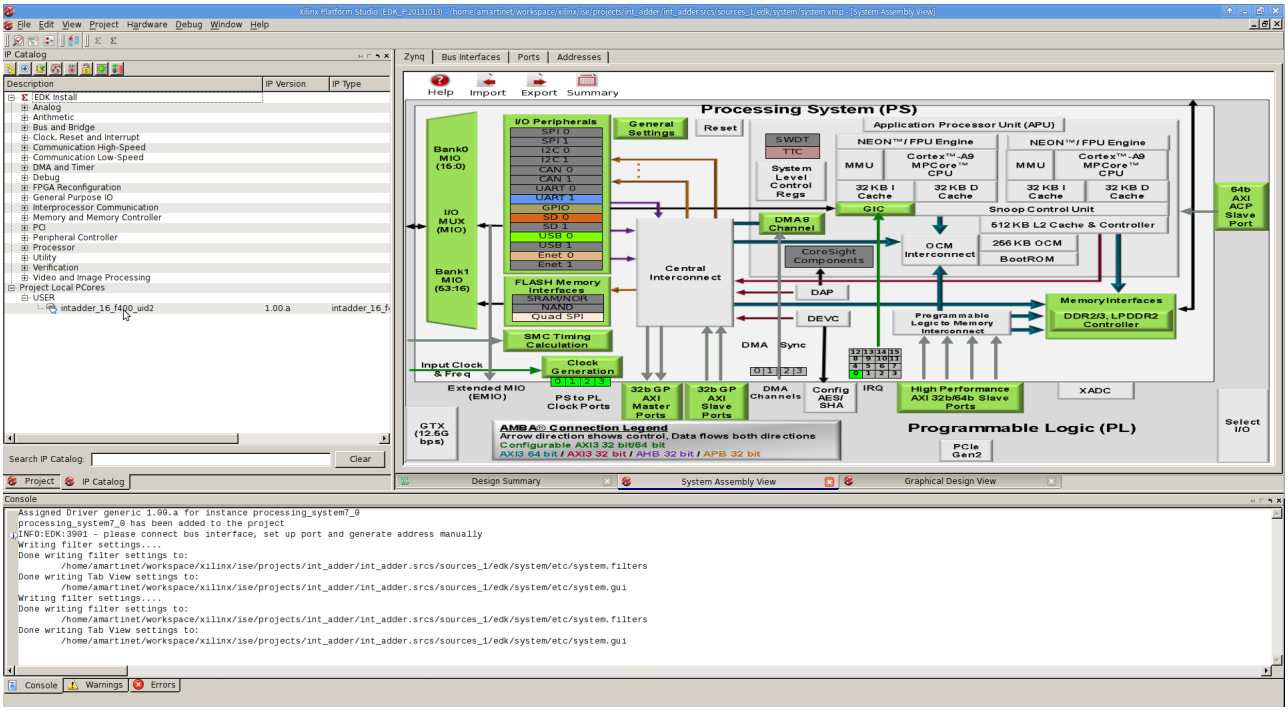

Figure 31: Add custom IP

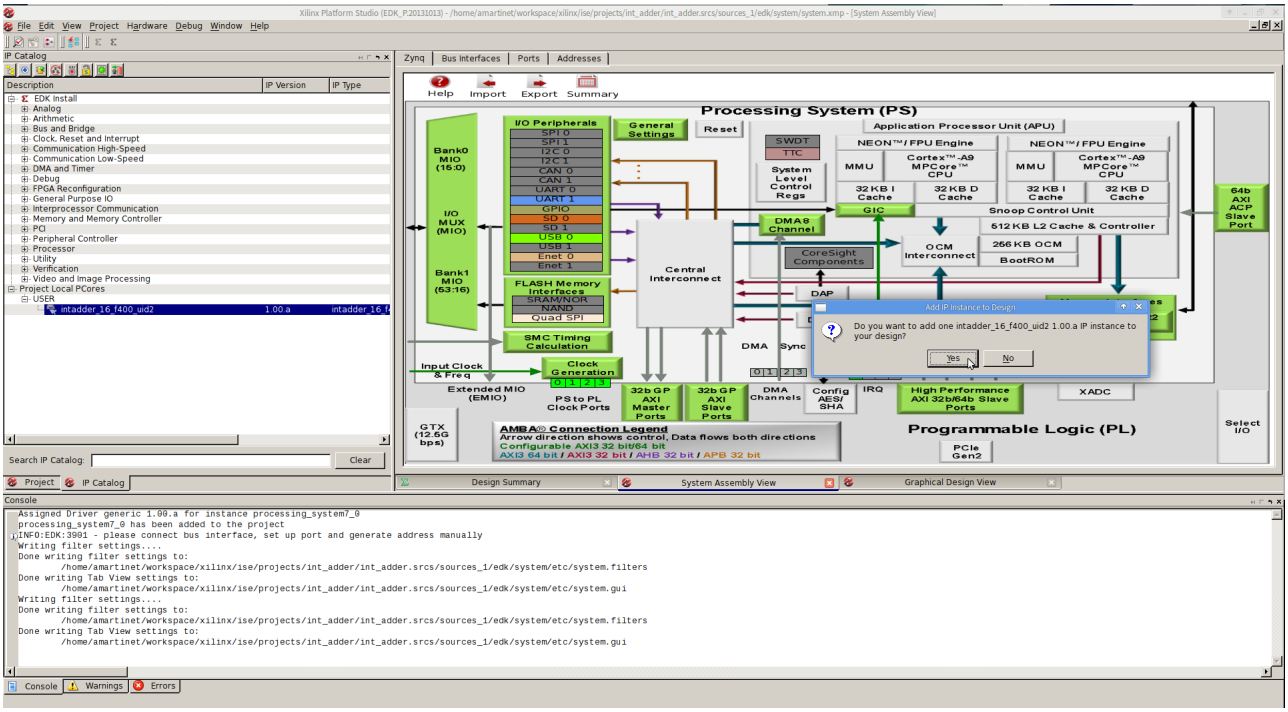

Figure 32: Add custom IP

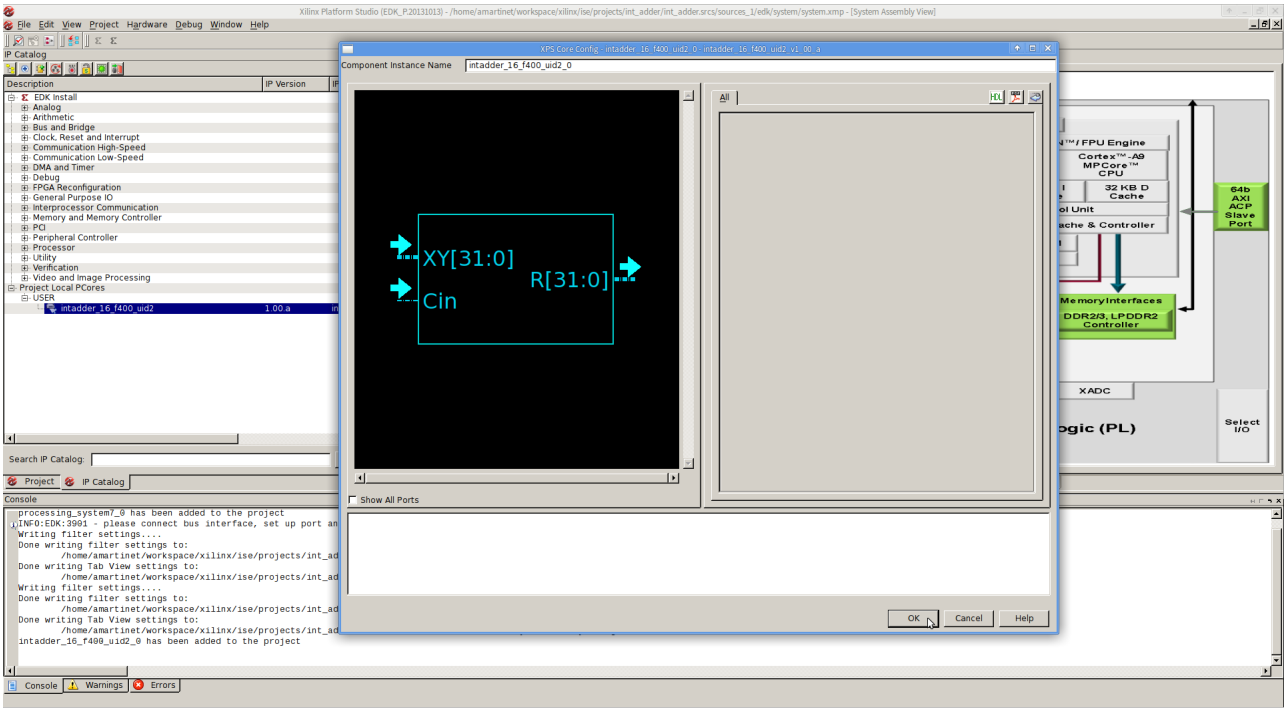

Figure 33: Add custom IP

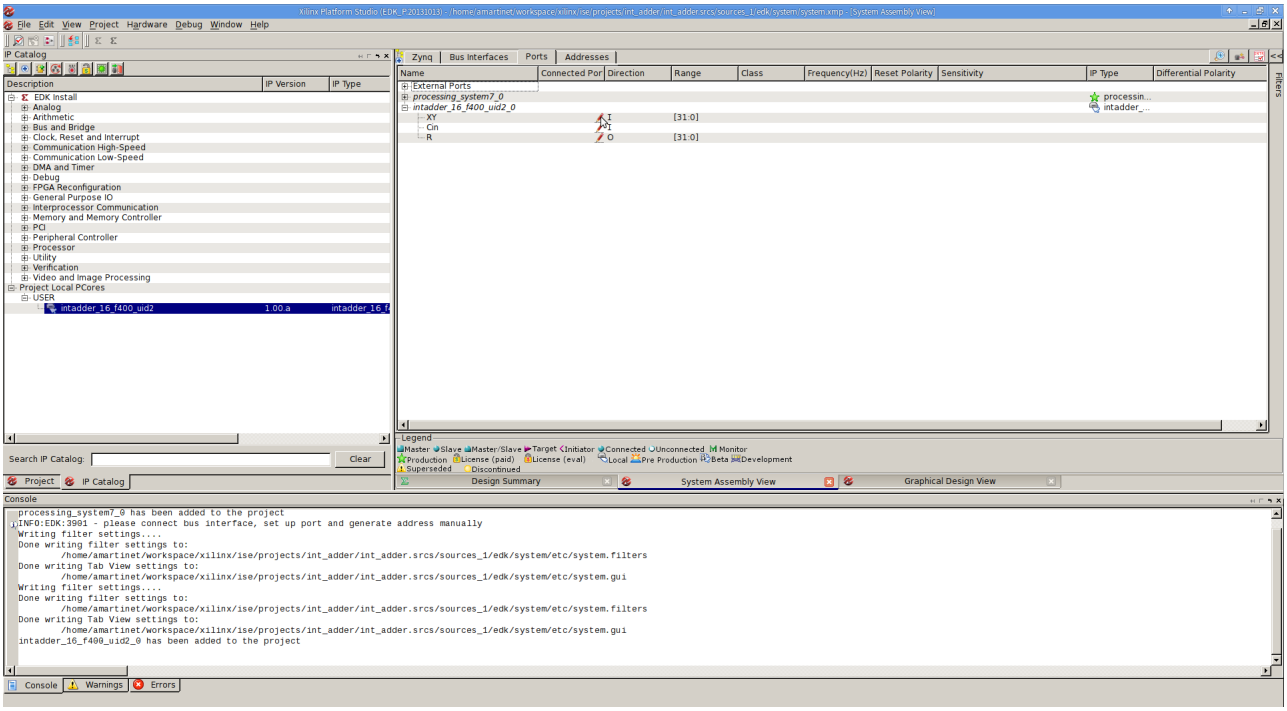

Figure 34: Add custom IP

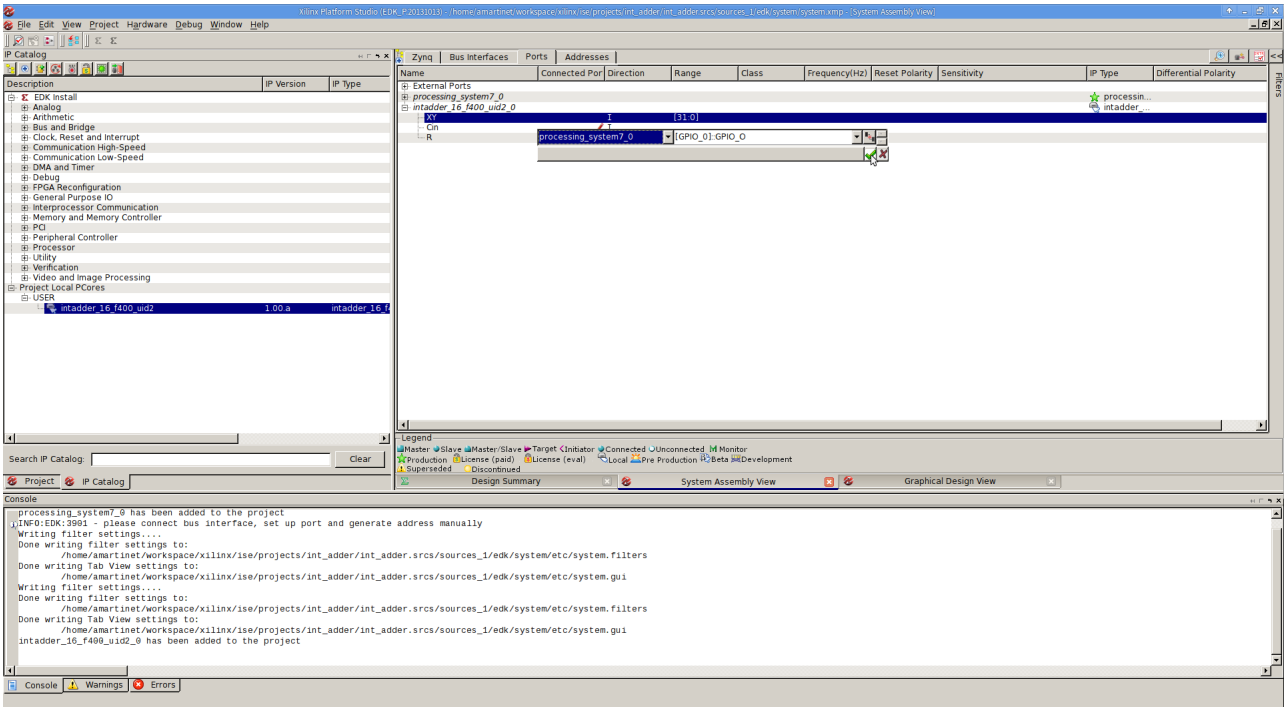

Figure 35: Add custom IP

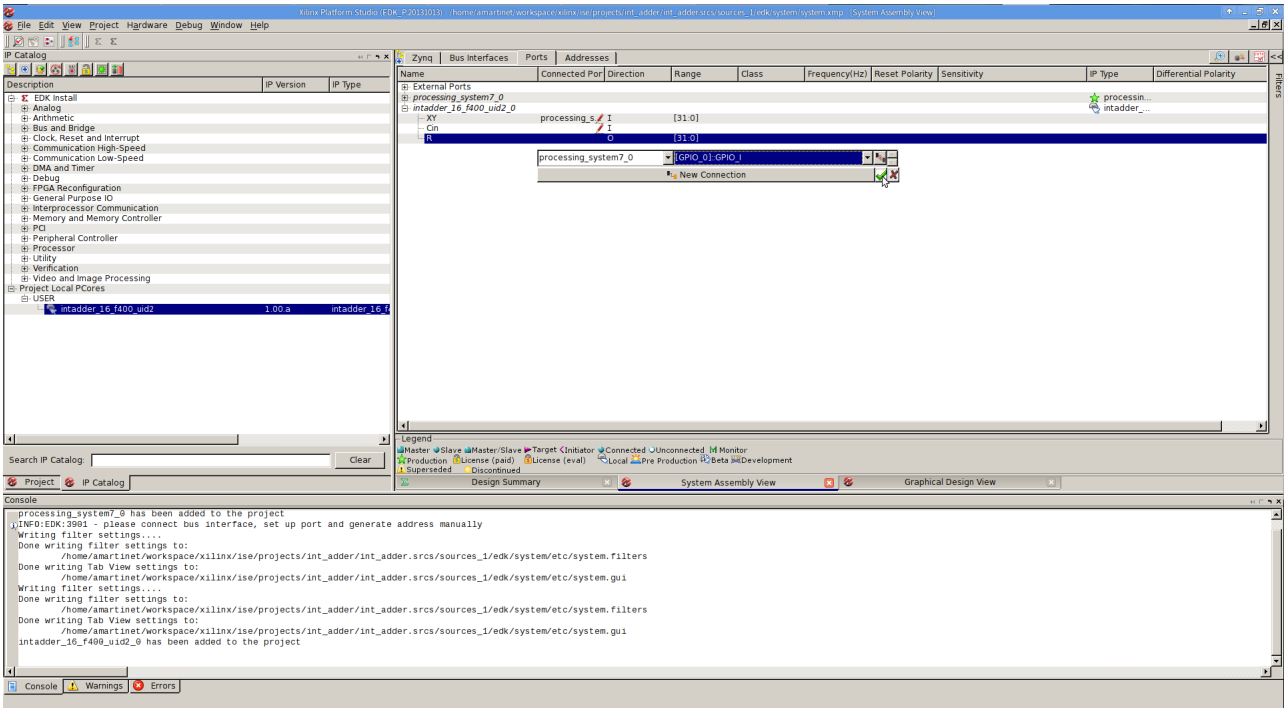

Figure 36: Add custom IP

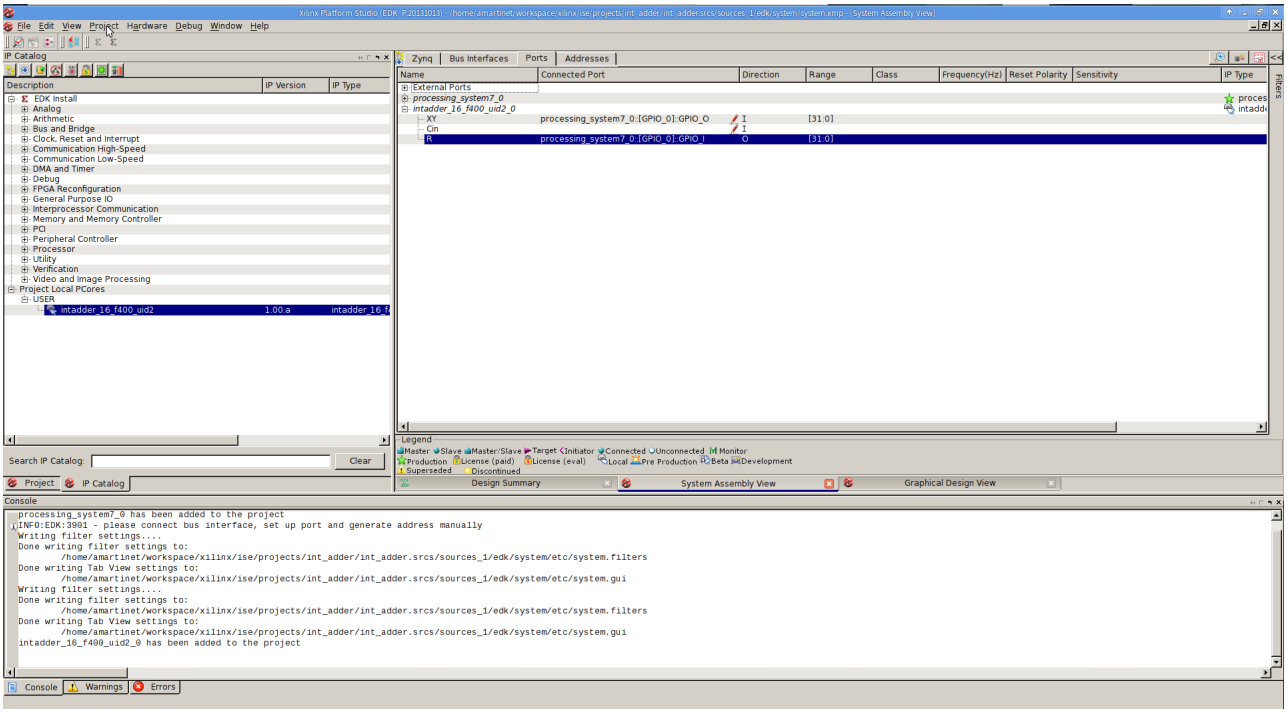

Figure 37: Add custom IP

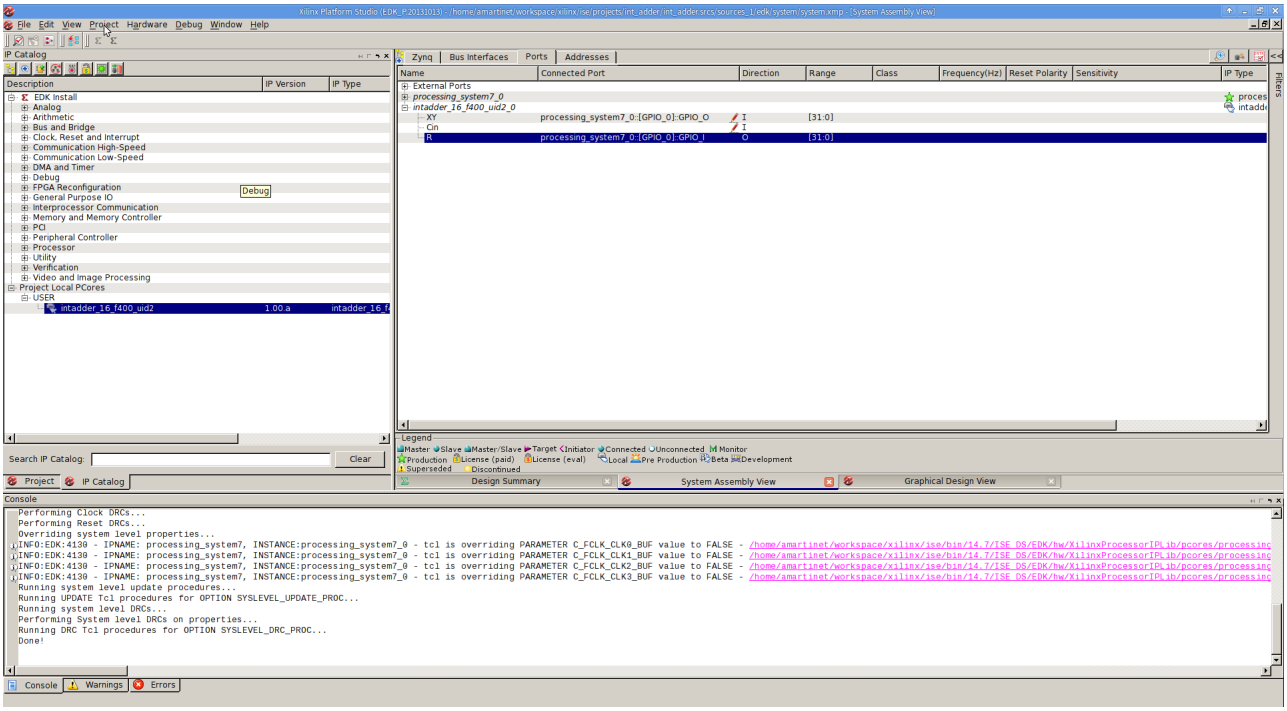

Figure 38: Design rule check

3 Generating hardware and drivers

3.1 Generate Hardware

Now you know how to design a hardware, let's take a look on the compilation chain.

1. First, you need to finish the design sources generation. You will see that a new design source popped in the "Sources" panel. Right click on "System (system.xmp)", and select "Create Top HDL". It will take a short time before the system_stub.vhd come up.

2. Then click on "Run Synthesis" in the "Synthesis" menu of the "Flow Navigator" panel.

3. Go make coffee while the synthesis runs.

4. you are then proposed to perform various actions. Choose the default "Run Implementation" and click "OK".

5. Go drink your coffee.

6. You will then get (usually 3) critical warnings about IBUF pins. Don't care of them and click ok to the boxes.

7. Then you will be asked again to perform various actions. Here I will choose "Open Implemented Design". You can also choose "Generate Bitstream", but remember you'll have to open the implemented design to be able to perform the later "Export Hardware" operation. This will be done using the "Open Implemented Design" option in the Flow Navigator. So for now, choose the default "Open Implemented Design". Wait for it to be read and displayed. Don't take care of the warnings, actually, you have already seen those ones at the synthesis step.

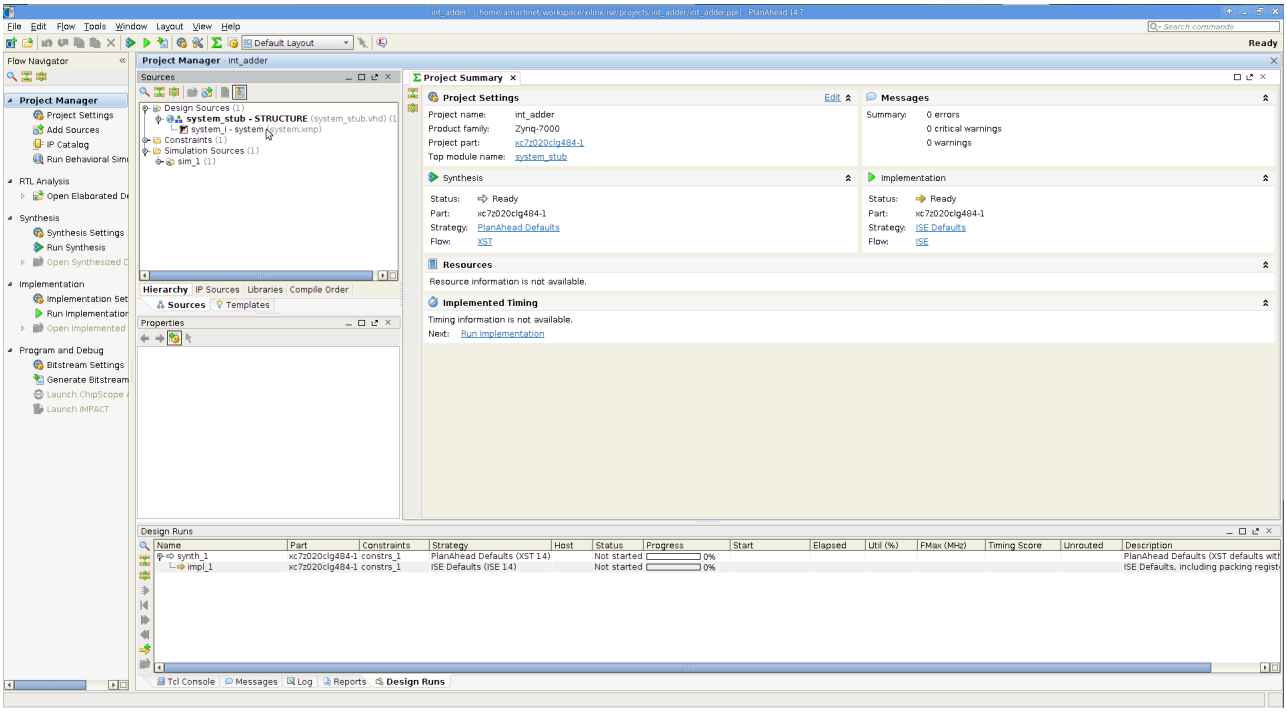

Figure 39: Create top HDL

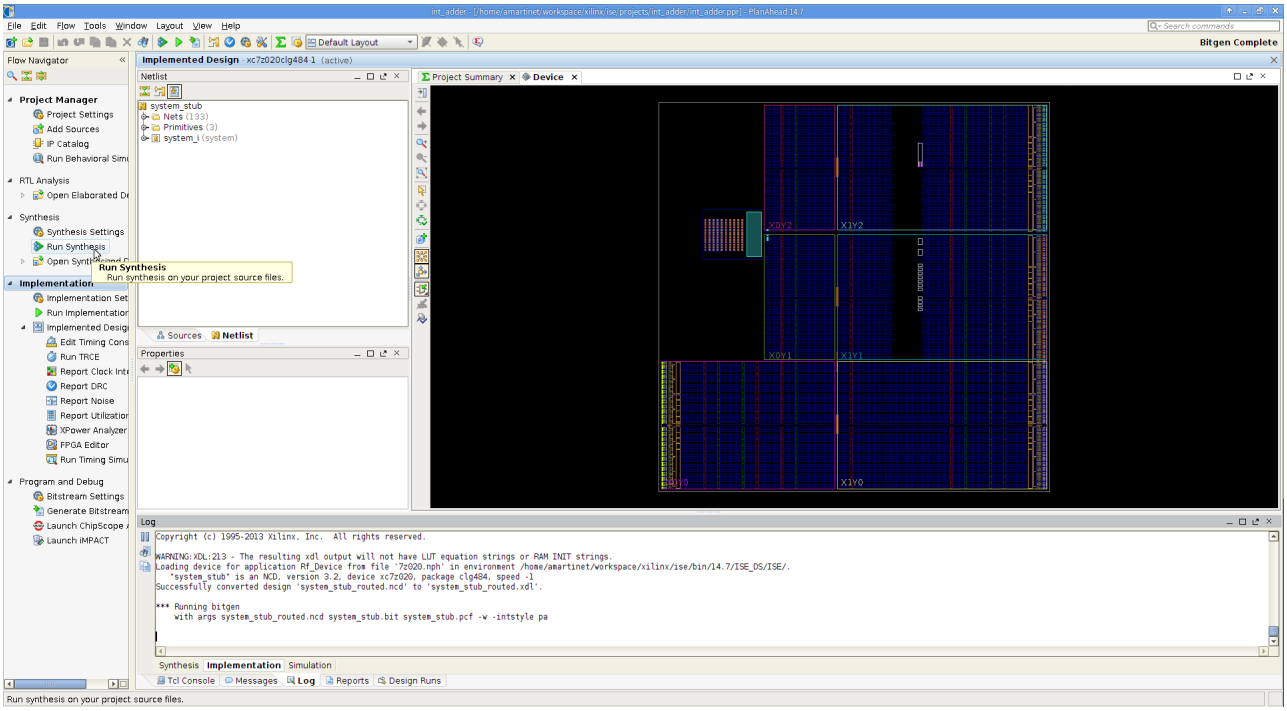

Figure 40: Run synthesis

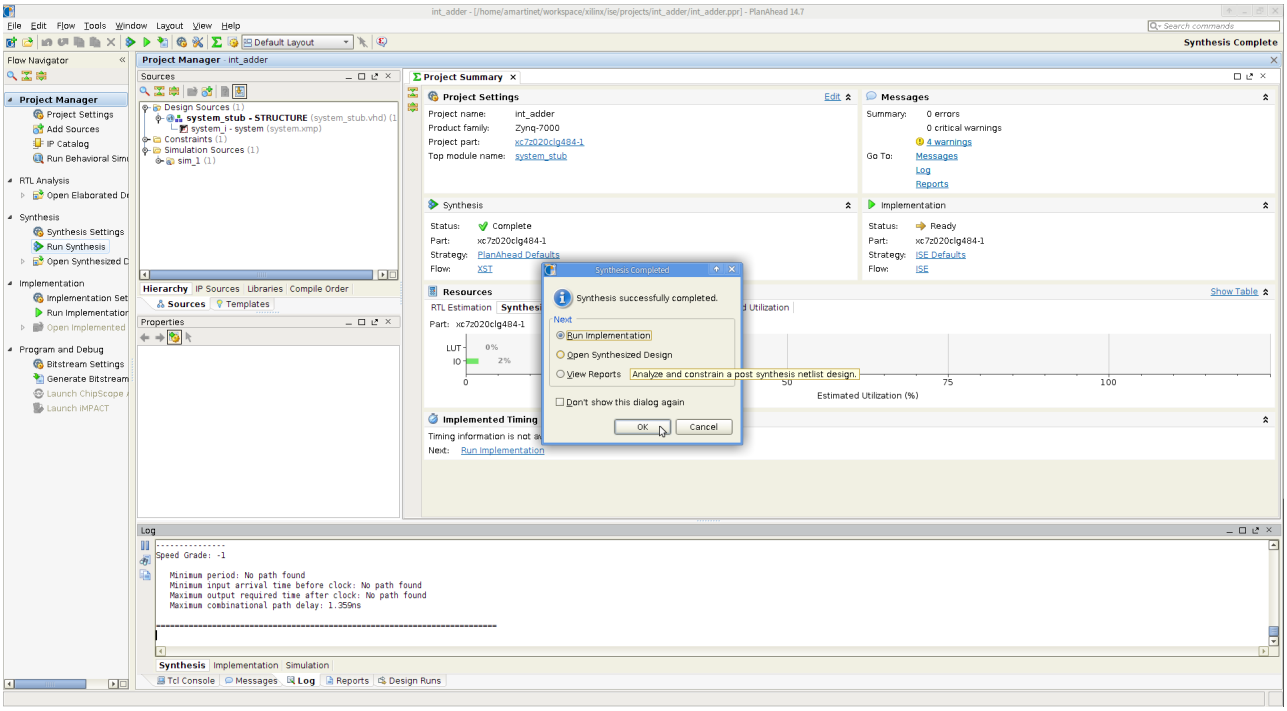

Figure 41: Run implementation

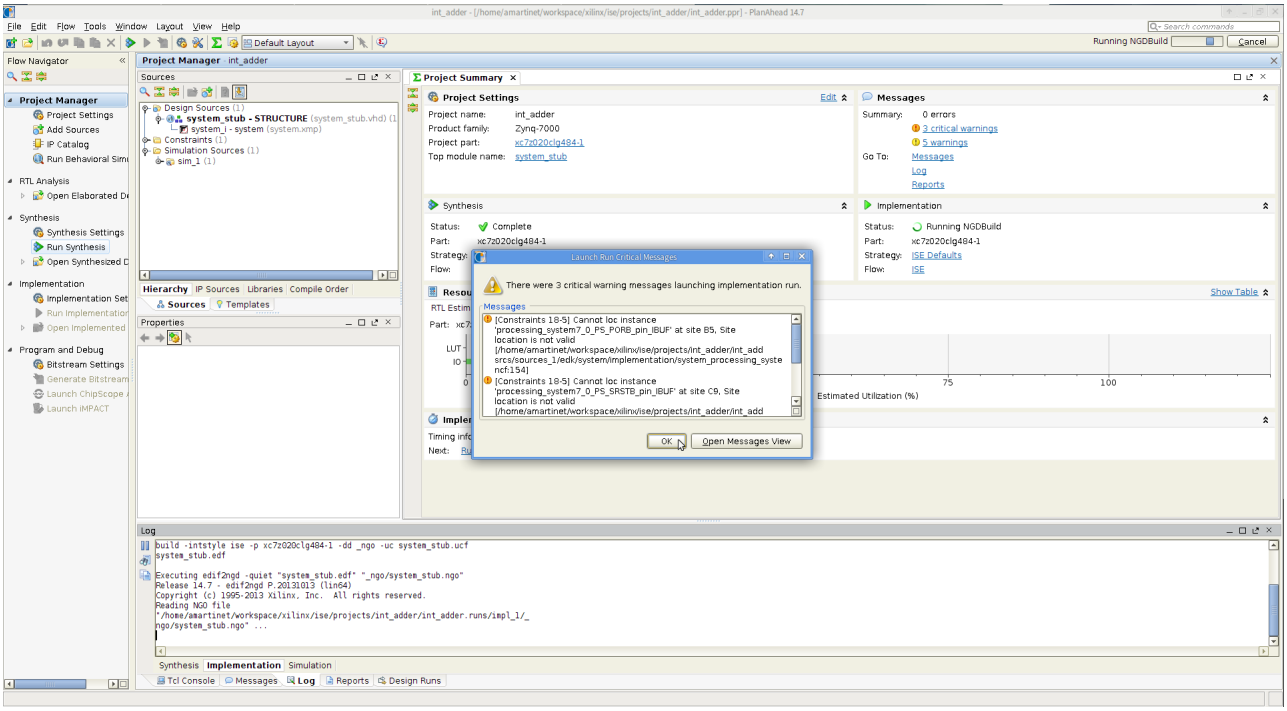

Figure 42: Critical warnings

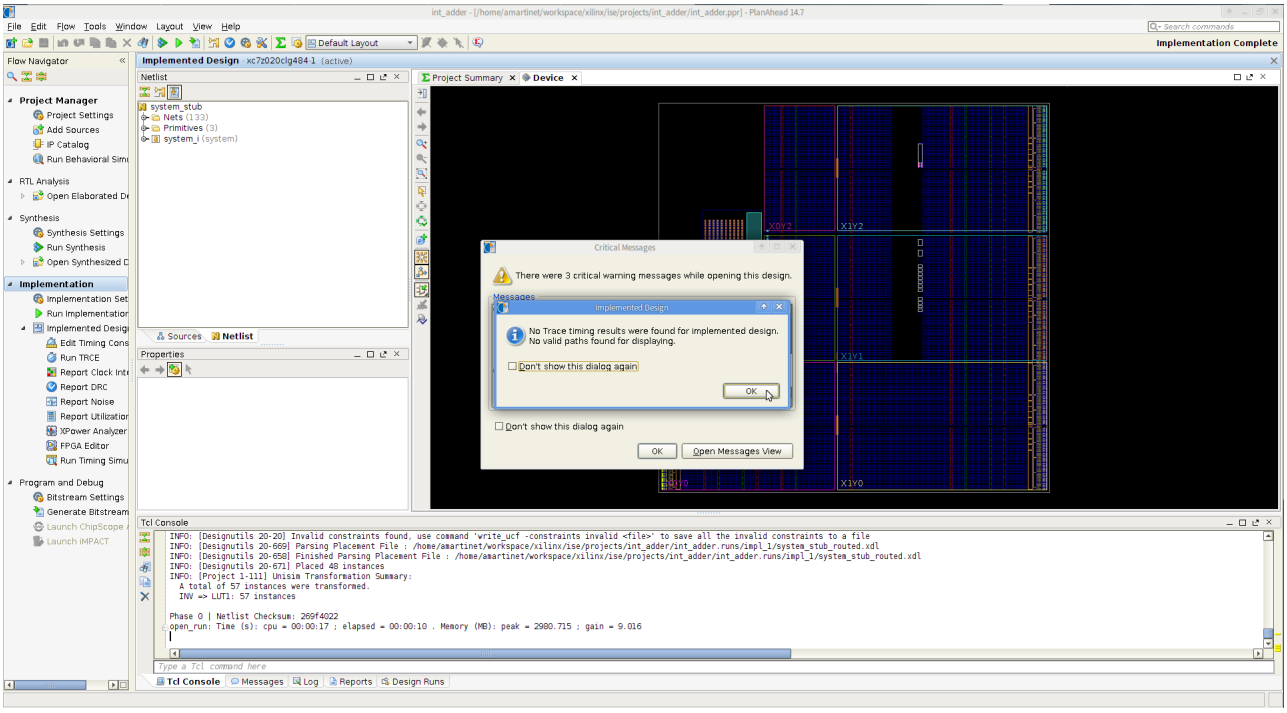

Figure 43: Critical warnings on implementation openning

8. Then click "Generate Bitstream" in the Flow Navigator. For further runs, I have to tell you that clicking it direct after the top HDL generation will result in doing all the precedent actions but the implementing design opening.

9. Take a little rest waiting Bitgen script to complete.

10. Then click the menu "File" -> "Export" -> "Export Hardware for SDK".

11. Check the three boxes ("Include Bitstream", "Export Hardware" and "Launch SDK"). Click "OK". Wait for SDK to launch.

4 Peripheral full test

Now the peripheral is configured and "drived", Let's take a look about it's good functionnality. First, you will create a new project and create code for getting information from your new peripheral.

4.1 Creating a new C project

You first need to create a new project. For this purpose, get through the following steps:

1. Go to "File" -> "New" -> "Project"

2. Expand "Xilinx" and select "Application Project". Click "Next".

3. Give a name to your project. For example, I use int_add_16. Leave everything as it is and click "Next".

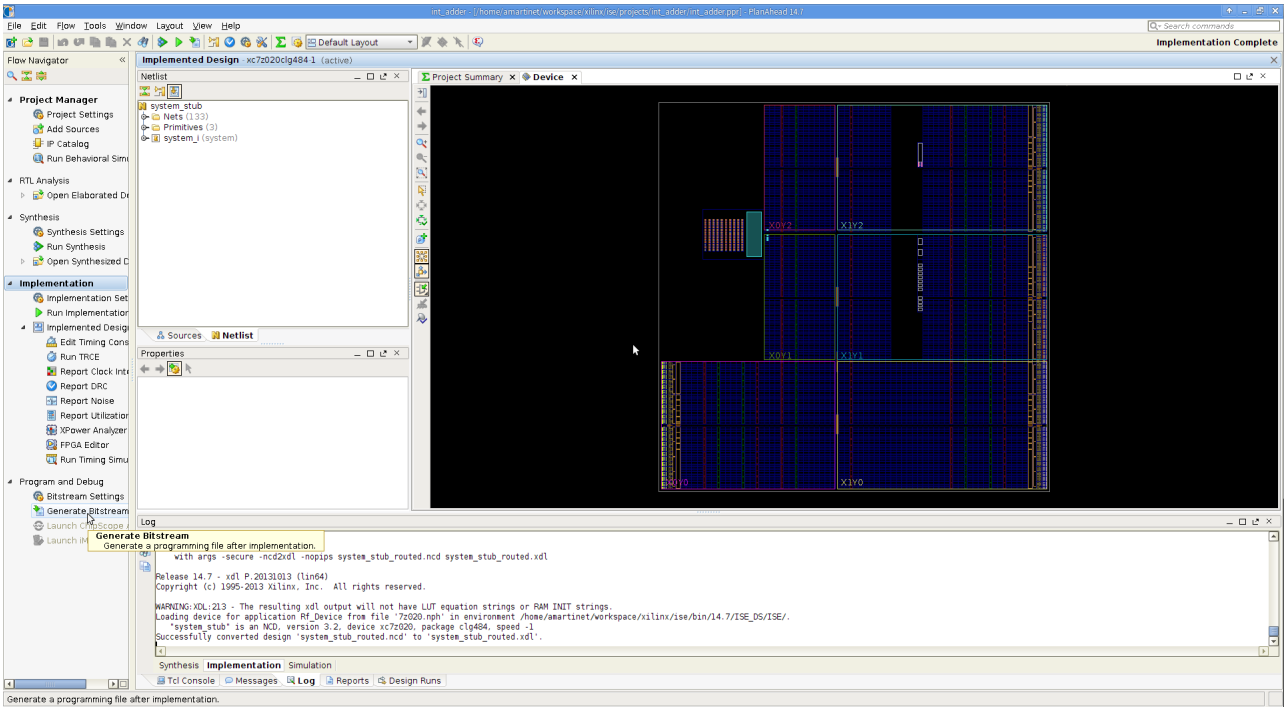

Figure 44: Generate bitstream

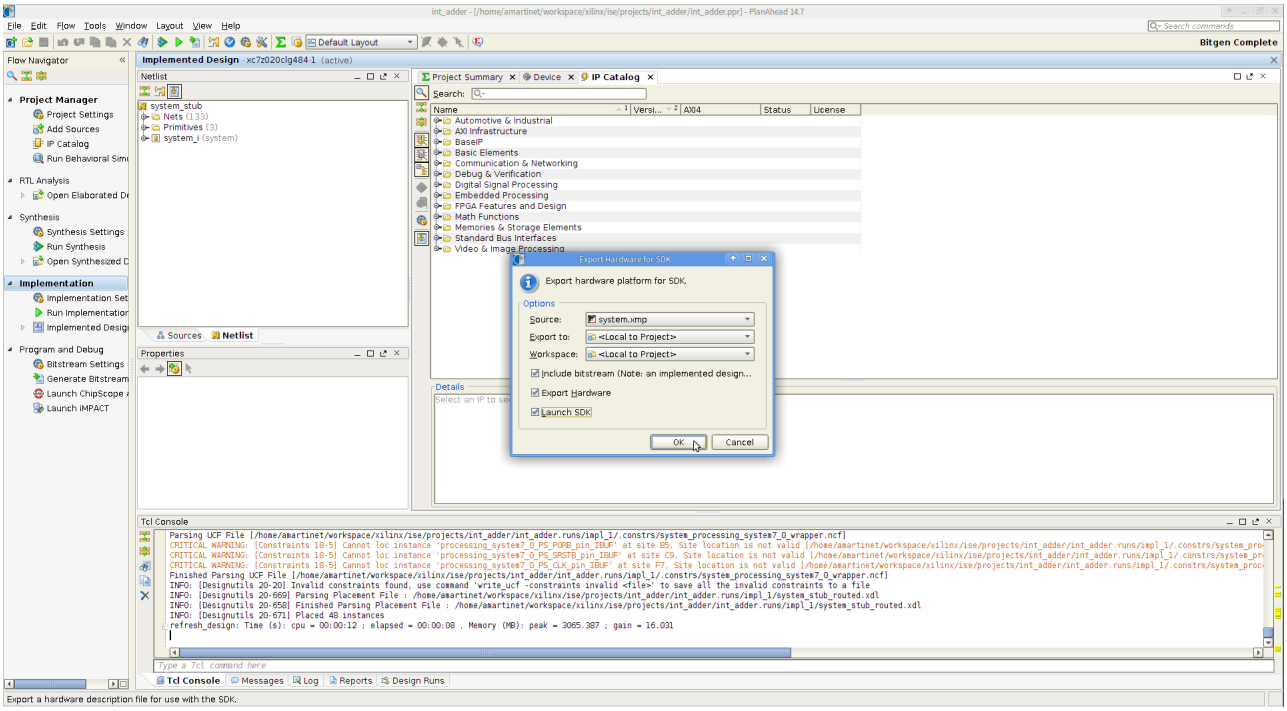

Figure 45: Export hardware

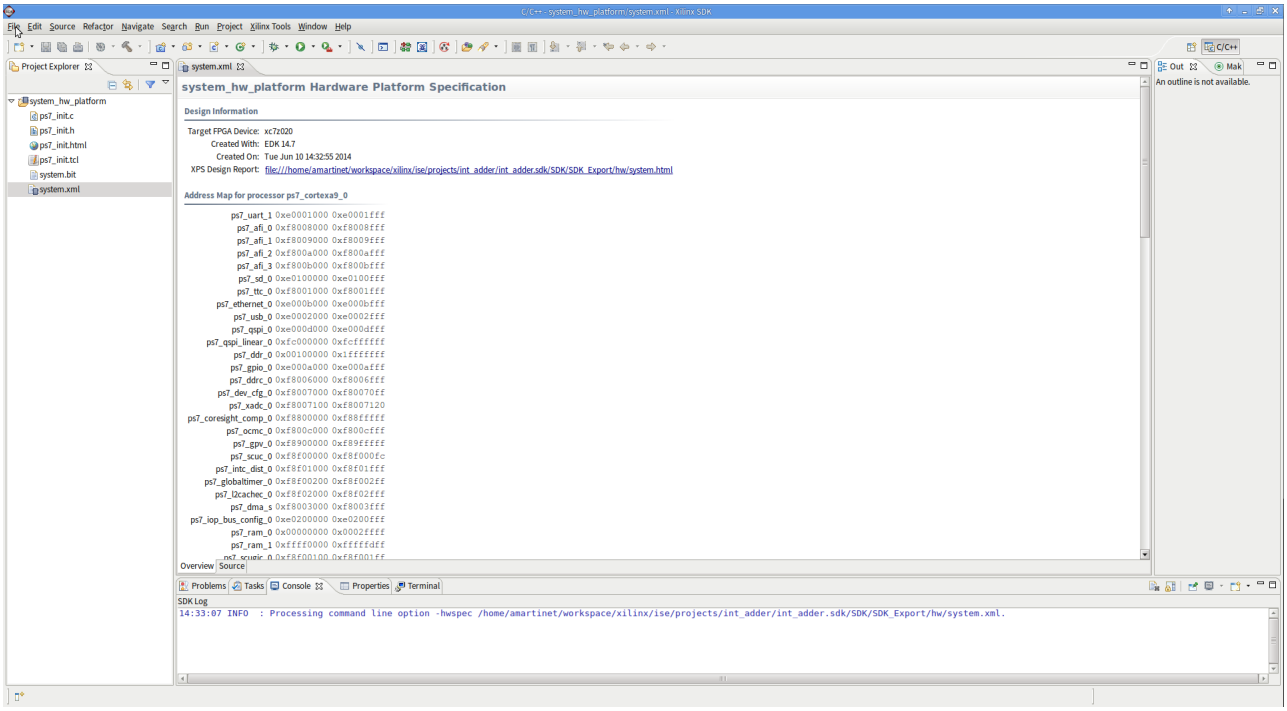

Figure 46: Xilinx SDK start window

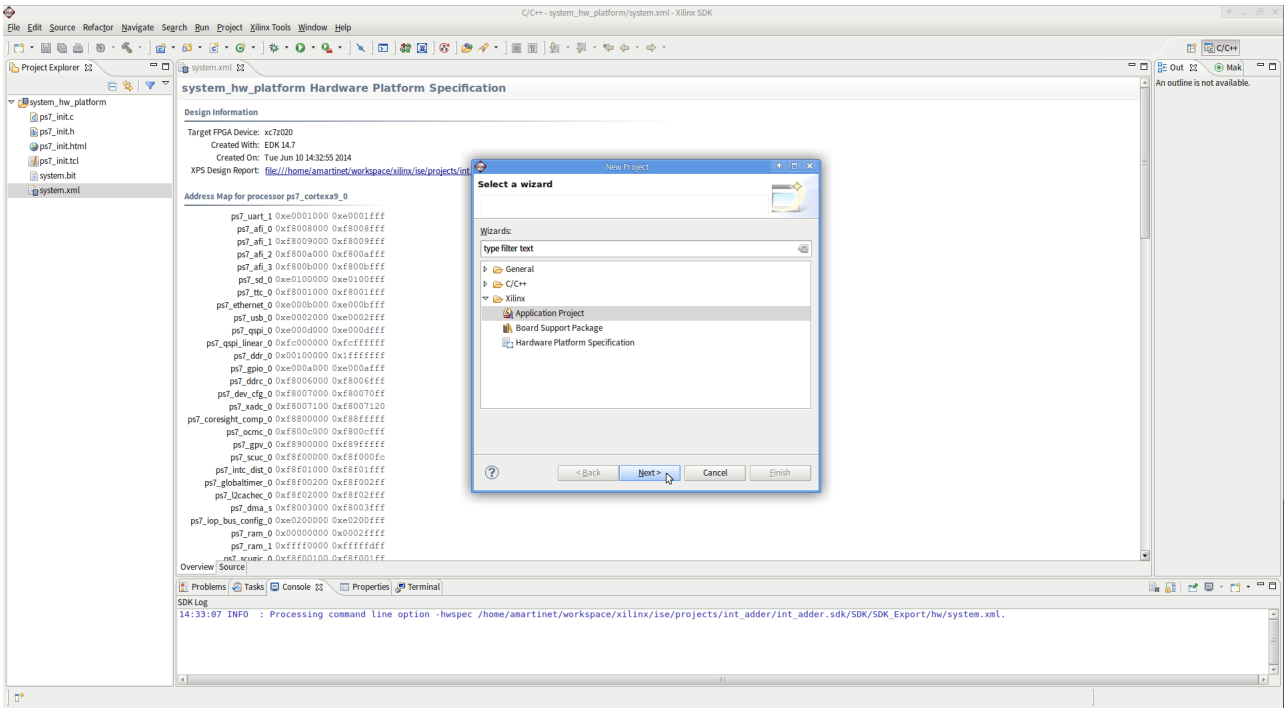

Figure 47: Export hardware

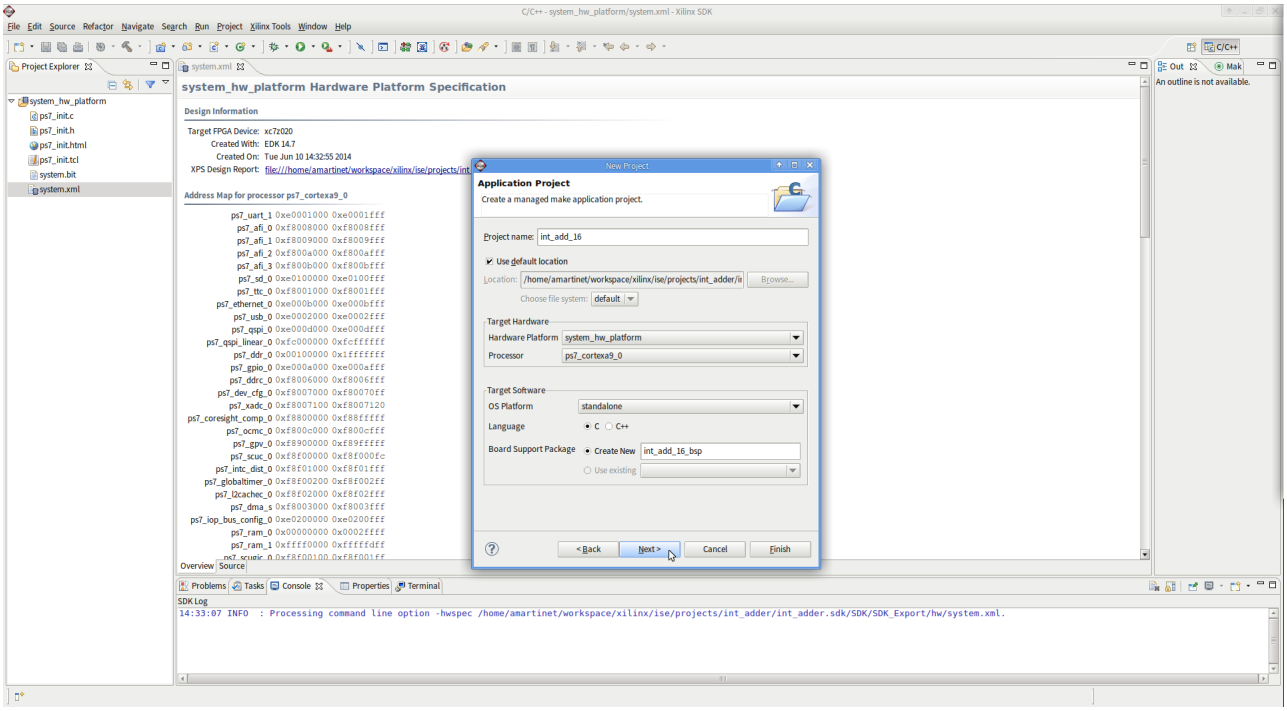

Figure 48: Export hardware

4. Select the default "Hello World" template and click "Finish". Your project has been added in the project explorer.

5. Expand it, expand the "src" folder and open "helloworld.c" double-clicking on it.

6. You see that you can now display "Hello World". That's fine, but we have to get more deeper.

4.2 Peripheral communication

The only thing you need to know about peripheral communication is that GPIO commuication is done through registers. You can find full documentation here at http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

So what you need to use is writing/reading routines, that fortunately are provided by drivers in the int_add_16_bsp/ps7_cortexa9_0/include directory. Here I will use the XGpioPs_ReadReg() and XGpioPs_WriteReg() macros from the xgpiops_hw.h driver. These macros need a base address, which will be the base gpio address (0xE000A000, also defined as XPAR_PS7_GPIO_0_BASEADDR in xparameters.h), and an offset address, to compute the register to write/read on/from. Read macro returns the value read and Write macro needs of course the data (32 bit width) to write.

Now we know how to read/write in a register, let's take a look on where to read/write. According to the documentation, ug585-Zynq-7000-TRM.pdf, the register to write is here DATA_2 (we are on Bank 2, input register, so output from the chip point of view) (offset 0x48, absolute address 0xE000A048). You can also perform writing using the maskable output registers, but we don't use masks, as it is useless for this job. But first, need to enable output (so FPGA output) writing 1 to the DIRM_2 register. You will the write 0xFFFFFFFF to it, and will be able to write on the full DATA_2 register. Reading is a much more simple. You just need to read using the routine and will get the full DATA_2_RO register value. (offset 0x68, absolute address 0xE000A068). Now you can perform the full adder testing, and check the addition is well done. You will find the code I used to test in appendix A: Code listing: helloworld.c .

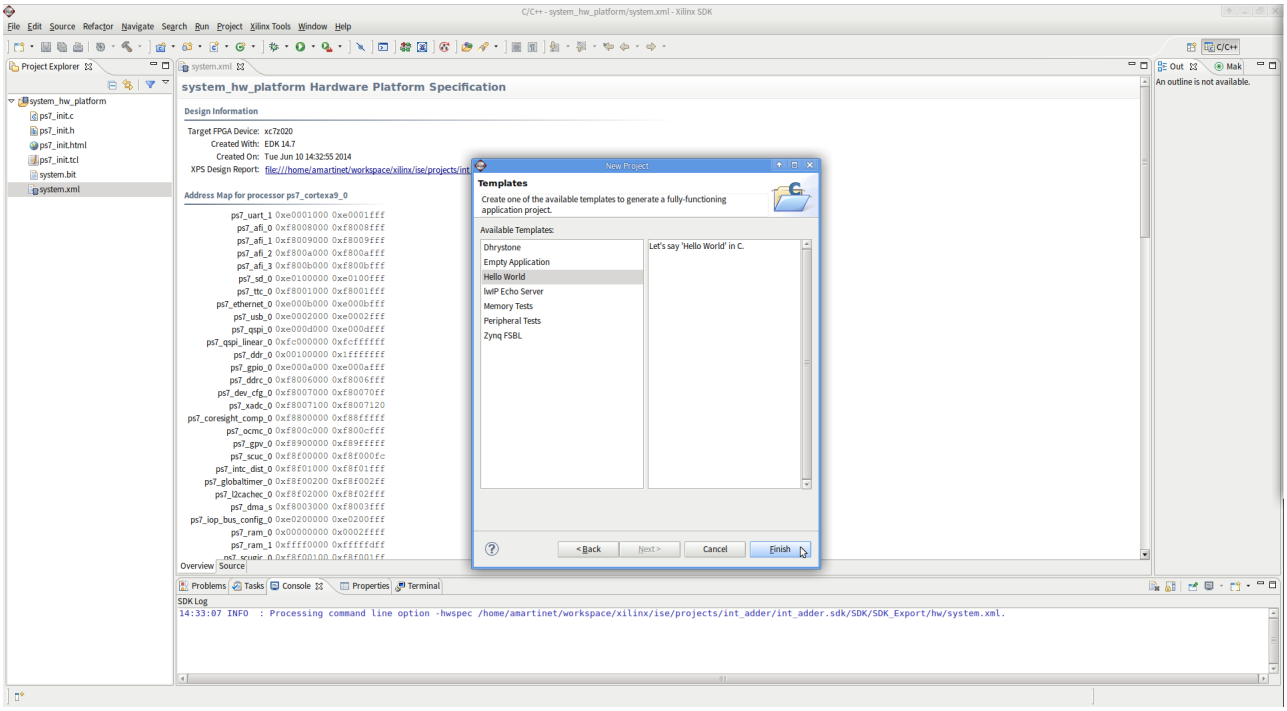

Figure 49: Export hardware

4.3 Board programming and program running

Now you have a good and useful code, you need to program the board and run your application.

4.3.1 Program the board

To program the board and read from it, you need your 2 usb/micro-usb cables plugged on the PROG (J17) and UART (J14) usb ports.

1. Connect them and the AC power supply. Power the zedboard using the POWER (SW8) switch.

2. Then you can connect a console through the JTAG usb cable (UART), using the shell command:

   ```
   $ screen /dev/serial/by-id/usb-2012_Cypress_Semiconductor_
   Cypress-USB2UART-Ver1.0G_027211C91A19-if00 115200
   ```

   You obtain a black screen because no application is running.

3. Back to SDK, you can put the bitstream into the FPGA. Click "Xilinx Tools" -> "Program FPGA". SDK wants to know what file to input. Keep the default one, which is the one that PlanAhead gived to SDK when you performed the "Export Hardware" action.

4. Click "Program". Wait for the process to complete. The blue led wakes up on the board, telling you that you can now run your application with a functionnal FPGA.

5. Before running your application, you must set a run configuration. There are several methods to do that. You can pass throught the run button, the "Run" menu, or right click on the project in the "Project Explorer" panel and choose the "Run As" option. Following one of these methods, click "Run Configurations". The wizard pops up.

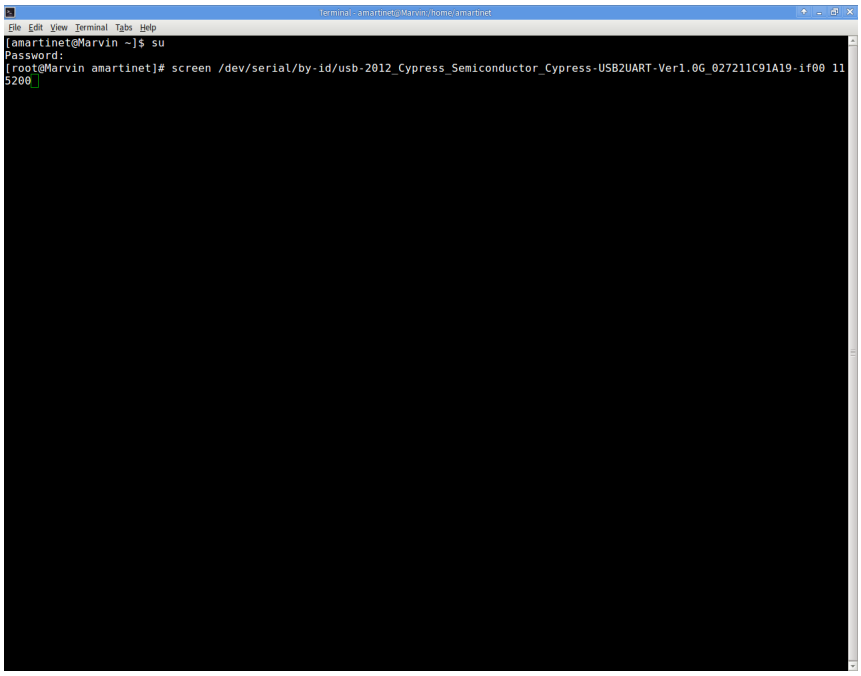

Figure 50: Listening to ZedBoard's port

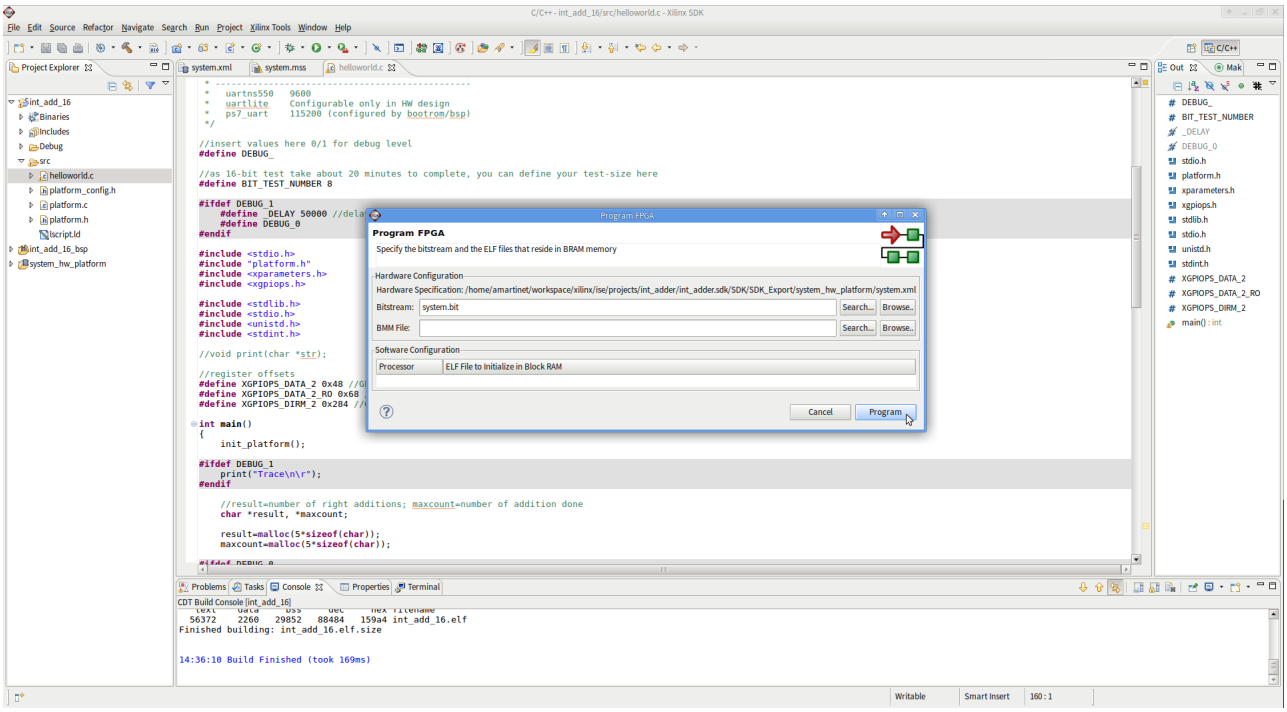

Figure 51: Program FPGA

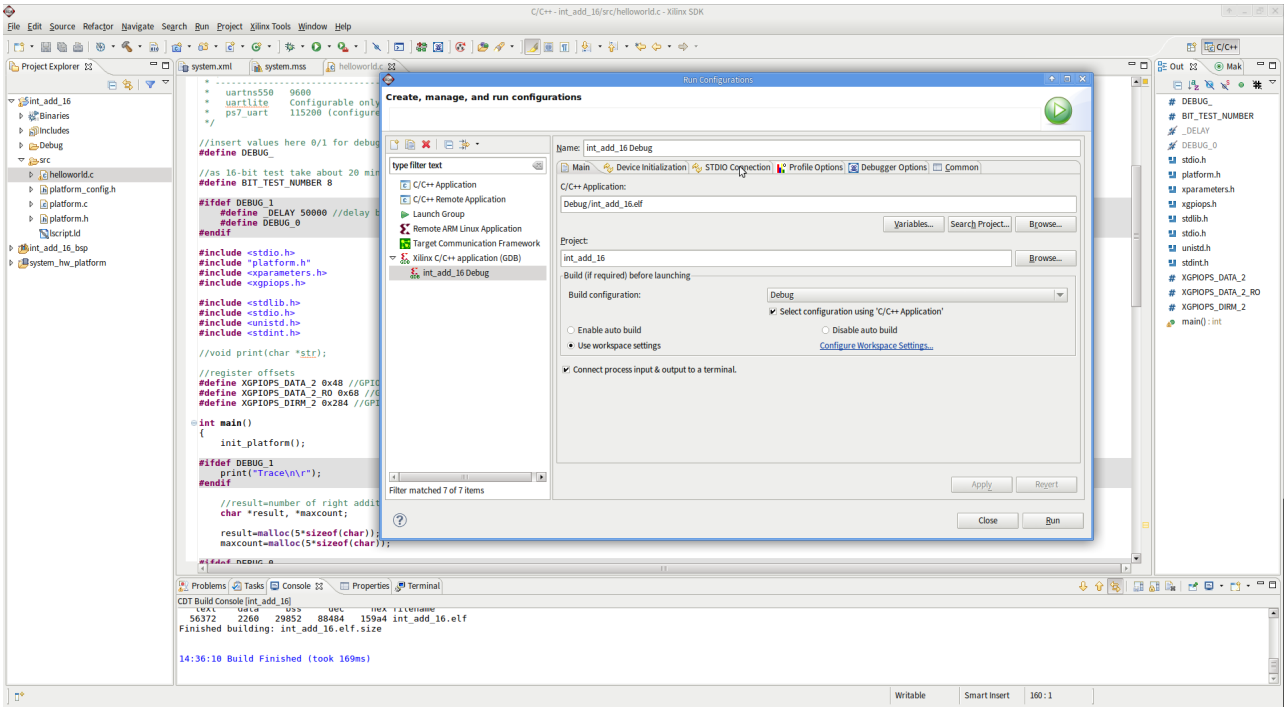

Figure 52: Run configuration

6. Right click on "Xilinx C/C++ application (GDB)", and click new.
7. In the "Device Initialization" tab, check that the default "Reset Type" is set to "Reset Processor Only".
8. In the "STDIO Connection" tab, check the "Connect STDIO to Console" and leave the defaults: "Port: /dev/ttyS0", "BAUD Rate: 9600", so that the board connects to the tty you will be looking at, and so you will be able to see outputs.
9. Click "Apply".
10. Then click "Run". Check the output. You will now see if you know how to add on 16 bits or not. (and actually, you do know!)

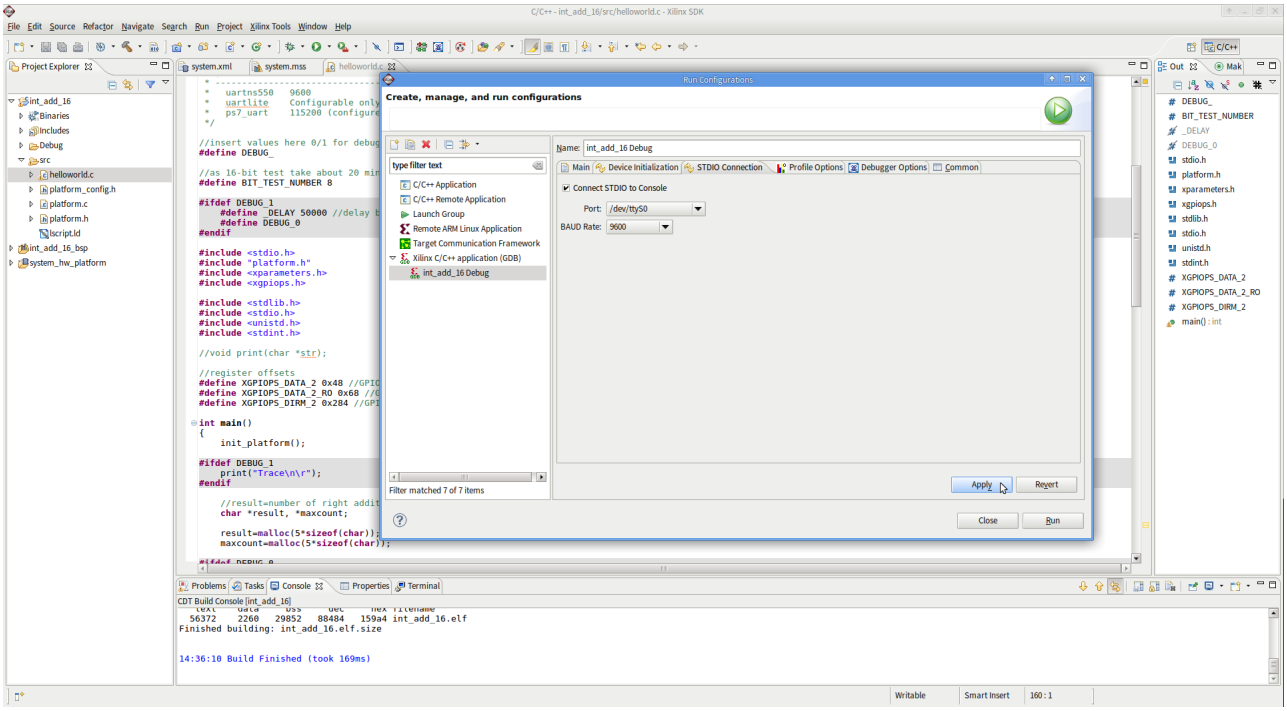

Figure 53: Run configuration

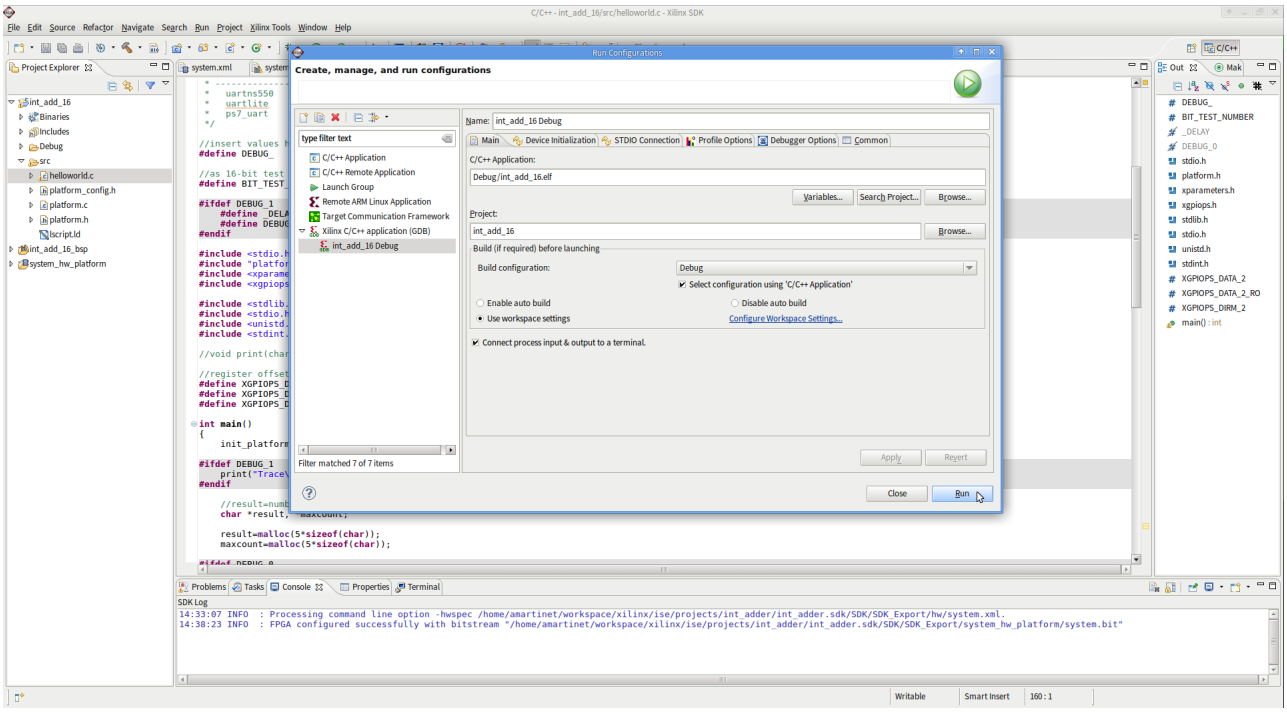

Figure 54: Run application

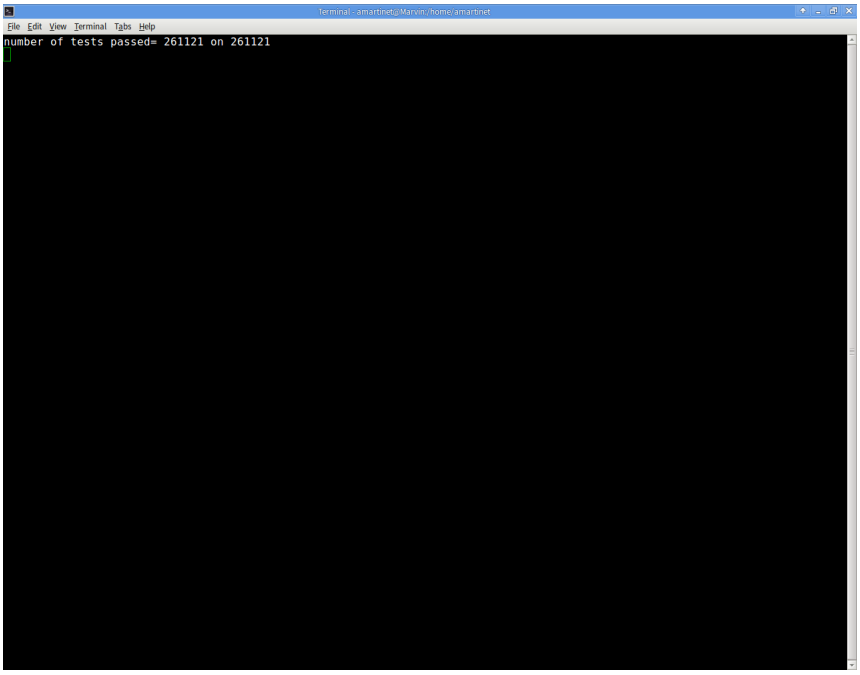

Figure 55: Run result on 8 bits test

Appendices

A Code listing: helloworld.c

```
showtabs
/*
 * Copyright (c) 2009-2012 Xilinx, Inc.  All rights reserved.
 *
 * Xilinx, Inc.
 * XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS A
 * COURTESY TO YOU.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS
 * ONE POSSIBLE   IMPLEMENTATION OF THIS FEATURE, APPLICATION OR
 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION
 * IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE
 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION.
 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO
 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO
 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE
 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 */

/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
```

```
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

//insert values here 0/1 for debug level
#define DEBUG_

//as 16-bit test take about 20 minutes to complete, you can define your test-size here
#define BIT_TEST_NUMBER 8

#ifdef DEBUG_1
 #define _DELAY 50000 //delay between two results display, in microseconds
 #define DEBUG_0
#endif

#include <stdio.h>
#include "platform.h"
#include <xparameters.h>
#include <xgpiops.h>

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdint.h>

//void print(char *str);

//register offsets
#define XGPIOPS_DATA_2 0x48 //GPIO input register (DATA_2, GPIO Bank 2, EMIO)
#define XGPIOPS_DATA_2_RO 0x68 //GPIO output register (DATA_RO, GPIO Bank 2, EMIO)
#define XGPIOPS_DIRM_2 0x284 //GPIO direction mode register (DRIM_2, GPIO Bank 2, EMIO)

int main()
{
 init_platform();

#ifdef DEBUG_1
 print("Trace\n\r");
#endif

 //result=number of right additions; maxcount=number of addition done
 char *result, *maxcount;

 result=malloc(5*sizeof(char));
 maxcount=malloc(5*sizeof(char));

#ifdef DEBUG_0
 //debug display section
 char *InputToPrint, *OutputToPrint, *baseone, *basetwo;
 InputToPrint=malloc(4*sizeof(char));
 OutputToPrint=malloc(4*sizeof(char));
 baseone=malloc(4*sizeof(char));
 basetwo=malloc(4*sizeof(char));
#endif

 //unmask all bits to be able to read output
 XGpioPs_WriteReg( XPAR_PS7_GPIO_0_BASEADDR, XGPIOPS_DIRM_2 , 0xFFFFFFFF );

 uint32_t Xinput;
 uint16_t base1=0;
```

```
 uint16_t base2=0;
 uint32_t success=0;
 uint32_t testnumber=0;

 while ( base1 < ( (2<<BIT_TEST_NUMBER) - 1 ) ) {

  while ( base2 < ( (2<<BIT_TEST_NUMBER) - 1 ) ){

   Xinput=(base1<<16)+base2;
   //writing into the XY input
   XGpioPs_WriteReg( XPAR_PS7_GPIO_0_BASEADDR, XGPIOPS_DATA_2 , (Xinput) );

#ifdef DEBUG_0
   sprintf(InputToPrint, "%u\n\r", Xinput);
   sprintf(baseone,"%i", base1);
   sprintf(basetwo,"%i\n\r", base2);
   print("input=");
   print(InputToPrint);
   print("as=");
   print(baseone);
   print("+");
   print(basetwo);
#endif

   //reading R output
   int Xoutput=XGpioPs_ReadReg( XPAR_PS7_GPIO_0_BASEADDR, XGPIOPS_DATA_2_RO  );

#ifdef DEBUG_0
   sprintf(OutputToPrint, "%i\n\r", Xoutput);
   print("output=");
   print(OutputToPrint);
#endif
   //check sum (we are adding on 8 bits so check is on 8 bits)
   if (Xoutput==(uint16_t)(base1+base2)) {

#ifdef DEBUG_1
    print("Addition ok!\n\r");
#endif
    success++;


   }
#ifdef DEBUG_1
    else {
     print("You failed, moron!\n\r");
    }
#endif
    testnumber++;

#ifdef DEBUG_1
    usleep(_DELAY);
#endif


   base2++;
  }
  base1++;
  base2=0;
 }

 sprintf(result,"%u", success);
 sprintf(maxcount,"%u", testnumber);
 print("number of tests passed= ");
 print(result);
 print(" on ");
```

```
 print(maxcount);
 print("\n\r");

 return 0;
}
```

B GPIO start guide

What you need to know about GPIO is pretty simple. There are 4 GPIO banks on the zedboard. The GPIO banks 0 and 1 are connected to MIO pins that are 54 bits large. So GPIO Bank 0 has a width of 32 bits and GPIO bank 1 22 bits. These are not useful as they are not connected to the Programable Logic (PL). GPIO Banks 2 and 3 are both 32 bits large and are connected to the EMIO interface to the PL. Note that they are plugged contiguesly (Bank 2 is 31:0 pins of EMIO and Bank 3 is 63:32 pins of EMIO). The following schematic tells you how GPIO banks are organized.

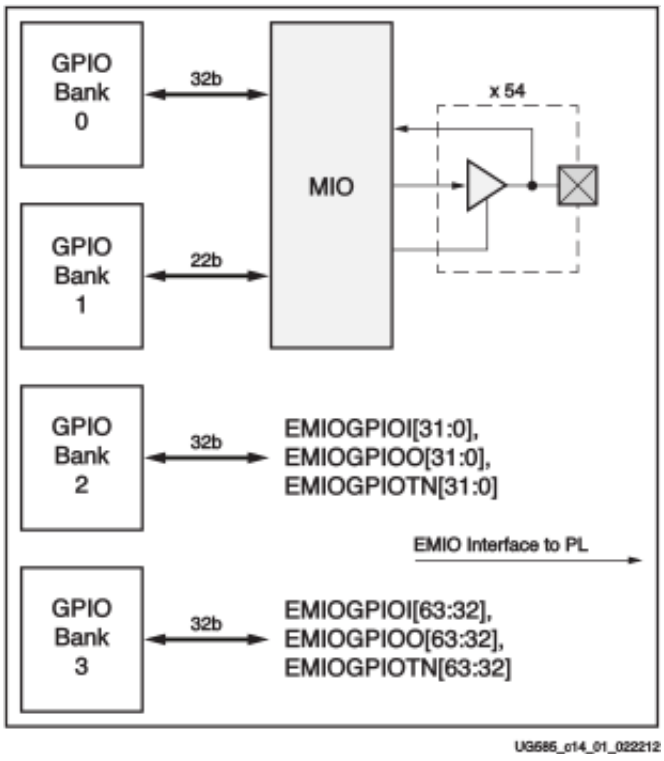

Figure 56: GPIO Block Diagram

Then, you will want to know how to access GPIO. This is pretty simple as you just access the registers decribed on the picture below. As you can see, the DATA_RO register is used to gather ingormation from GPIO devices. so we use it as input. You can see that the DATA, MASK_DATA_LSW, and MASK_DATA_MSW registers correspond to the same output register (chip point of view taken). Then you can see the DIRM and OEN register which can enable output and so writing operations on GPIO devices. As you can see, there is a bitwise and between this two registers. You might pay attention to that if you are

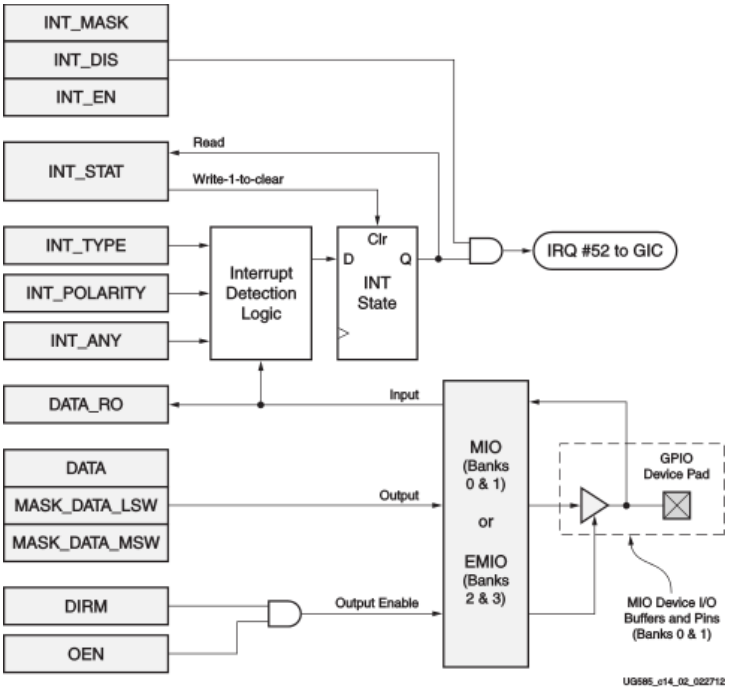

Figure 57: GPIO Channel

using regular GPIO. You can also see various interrupts configuration registers that I don't use in this tutorial.

Of course, you will get much better information using the original documentation. You will find it here, as I mentionned earlier:
http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

C Known bugs

C.1 JTAG connection

Though your USB-JTAG driver may be well installed and configured on your computer, the JTAG connection might be instable. This sometimes could be resolved by different ways:

- connect after programming FPGA
- connect just right after launching the application (with risk to miss some outputs)
- recompile all the hardware and re-import bitstream
- close all the project and re-open software
- reboot the pc

C.2 Manjaro/Arch Linux issues

Issues have been found on Manjaro 3.10.40-1, when output to console fails when computer AC/DC power suppply converter is not plugged. This could be workaround by hard-resetting the zedboard using the power

switch, then downloading the bitstream, strarting the application and then connecting your console to the board. Of course, you might loose some of the results, but for this test it should be fine.