1 NoC Platform Generation Process

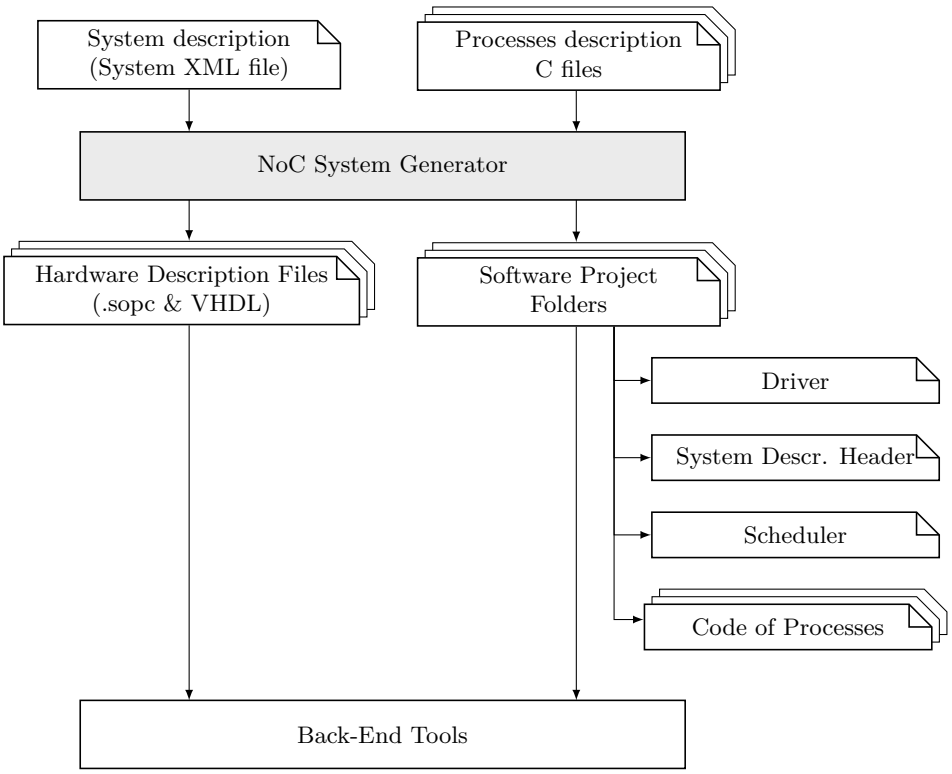


Figure 1.1.: *The Platform Generation Process of the NoC System Generator and the related files.*

Figure 1.1 gives an overview of the platform generation process. The NoC System Generator requires two inputs. The *system description XML file* (SD-XML) file, and a group of *processes description* C files (PD-C).

The SD-XML file contains the system-level description of the system we want to generate: a process network, representing the application, together with details regarding

the NoC-based MPSoC where we want to map the processes of the process network. It will be described in Section 1.1.2. The PD-C files describe the functionality of each process of the process network. The NoC System Generator requires at least one PD-C file for each process of the process network.

From the two inputs, the NoC System Generator produces two outputs: the *Hardware Description Files* (HDFs) and the *Software Projects*. These two outputs can be easily integrated in commercial FPGA synthesis tools (Altera QSYS and Xilinx XPS) to generate a working prototype of the system-level model, running on the selected FPGA.

1.1. Input files

1.1.1. processes description C files (PD-Cs)

The PD-C files are C or C++ files provided by the user, describing the functionality of the processes (actors) of the process network. PD-C files can be generated by from a system-level framework such as Simulink or ForSyDe.

1.1.2. System description XML file (SD-XML)

The structure of different parts of the system description XML file are shown in listing 1.1 and listing 1.2. The first section of listing 1.1, just after the leading `<system>` tag, specifies the system parameters and the target FPGA platform. These parameters define the folder path where to generate the output files, the system name, the target FPGA manufacturer, the FPGA board type, etc.

The configuration of the hardware is embedded into the `<hardware>` tag. The first part (within the `<noc>` tag) specifies the parameters of the network-on-chip, such as topology, size, dimension. The `framesize` parameter defines the maximum size of any packet of the input and output buffers of the system. Data through the NoC is currently sent with a bitwidth of 32 bits (bandwidth of the connection between switches). If all processes in the system communicate one single integer data (or a float single precision), a framesize of 3 is sufficient (2 headers and 1 flit composed by 32 bits each). If the processes communicate using only one double precision data, a framesize of 4 is sufficient (2 headers and 2 flits composed by 32 bits each). If the processes communicate using a vector of 4 double precision data, a framesize of 10 is sufficient (2 headers and 8 flits composed by 32 bits each). The `GlobalSync` parameter sets the heartbeat frequency.

The following part of the SD-XML file specifies details of the different NoC nodes. Each node has its own `node` element, which describes the hardware connected to that specific node. The main parameter here is the processing element (PE) connected to

Listing 1.1: *Structure of the System description file of the NoC generator*

```
<?xml version="1.0" encoding="UTF-8"?>
<system name="NoC_2x2">

  <!-- Platform specification -->
  <parameter name="param1" value="val_1" />
  <parameter name="param2" value="val_2" />
  ...

  <hardware>

    <noc>
      <parameter name="nocType" value="Mesh" />
      <parameter name="nocKind" value="2DNoC" />
      <parameter name="nrofCols" value="3" />
      <parameter name="nrofRows" value="3" />
      <parameter name="framesize" value="64"/>
      <parameter name="GlobalSync" value="1␣Hz"/>
      ...
    </noc>

    <node nr="0" cpu="{nios,tiny}" ... />
    <node nr="1" cpu="{nios,tiny}" ... />
    ...

  </hardware>

  <software>
    ...
  </software>
</system>
```

node. Further parameters specify additional hardware connected to the PE, such as memory, PIO, JTAG, IP blocks, and so on.

The SD-XML file part shown in listing 1.2, — embedded within the `<software>` tag — specifies the mapping of the actors of the process network (PD-C files a.k.a. software processes) to the MPSoC nodes. An example is shown in Listing 1.2, where four software processes are mapped to three MPSoC nodes. The directory where the functionlity of the actor (PD-C files) can be found is specified by the `Repository` tag. Then every actor (sofware process) is specified by a `process` tag. An actor (process) is configured by the process name (`name`), its model of computation (`moc`), the PE on which it runs (`node`), and the IF-C file name containing the process functionality. The software processes from which the process receives data and the processes it sends data to are specified by the `source` and `target` tags. This information is used to assign the input and output buffers to the associated process. The order in

which the processes are specified here, is also the order of execution on the nodes (scheduling within a node).

Listing 1.2: *Process Declaration within the Target Description File*

```
<software>
  <parameter name="Repository" value="D:/NoC/SW" />
  <process name="p0" moc="Synchronous" node="0"
          sources="{p3}" targets="{p1}"
          files="{process_0.c}" />
  <process name="p1" moc="Synchronous" node="0"
          sources="{p0}" targets="{p2}"
          files="{process_1.c}" />
  <process name="p2" moc="Synchronous" node="1"
          sources="{p1,p3}" targets="{p3}"
          files="{process_2.c}" />
  <process name="p3" moc="Synchronous" node="2"
          sources="{p2}" targets="{p0,p2}"
          files="{process_3.c}" />
</software>
```

1.2. Output files

Both the *Hardware Description Files* and the *Software Projects* are used by the back-end tools provided by the FPGA vendor to create a running system on the selected FPGA.

1.2.1. The Hardware Description Files (HDFs)

The *Hardware Description Files* consist of VHDL files that describe the Network-on-Chip interconnection and the whole system descriptions (PEs, memories, etc) described through different formats depending on the targeted FPGA vendor. In the case of Altera FPGAs, the system description is a `.qsys` file that is processed by the Altera QSYS tool. In the case of Xilinx FPGAs, the system description is composed by `.mhs` and `.mss` files that are processed by the Xilinx XPS tool. From the HDFs it is possible to automatically create the bitstream to configure the FPGA using the FPGA vendor tools.

1.2.2. The Software Projects

Beside the hardware description files, the NoC System Generator creates a software project for every PE in the NoC. The software project for a specific PE is composed

by:

- the IF-C files describing the functionality mapped to the specific PE;
- a scheduler, named `synchronous_MoC_main.c`;
- the System Description Header, named `software_configuration.h`;
- device drivers.

1.2.2.1. The IF-C files

The IF-C files mapped on the specific process have to be written in a certain way to work on the generated NoC-based system. The basic structure of a IF-C file is shown in Listing 1.3. For a detailed example, the examples of the network generator can be consulted.

Listing 1.3: *Structure of the process code provided to the network generator*

```
#include "software_configuration.h"

void p0_init(void)
{
  // Perform initializations

  // Write initial message to RNI
  while (NOC_RNI_STATUS(NOC_RNI_BASE)!=0);
  NOC_RNI_SEND( ... );
};

void p0_main(void)
{
  int recv_value = NOC_RNI_CHK_MSG( ... );
  if (recv_value > 0)
  {
     // Read message from RNI
     // Process message
     // Write message to RNI

     while (NOC_RNI_STATUS(NOC_RNI_BASE)!=0);

     NOC_RNI_SEND( ... );
     NOC_RNI_CLEAR( ... );
   }
   else
   {
     // Absent value
   };
};
```

Firstly, System Description Header (`software_configuration.h`) is included. As described in Section 1.2.2.3, this file loads the network driver and defines a set of helper functions and macros used in the IF-C file. The IF-C file code continues defining two functions, scheduled by the scheduler. The first function, `px_init(void)`, is called once when the process starts executing. It performs the necessary initialization to guarantee a proper start of the system. As the initialization can be seen as the first cycle of the synchronous execution, the initial values of delay elements also have to be sent to the network. This done by the three steps: (1) writing the message to the send buffer of the RNI, (2) wait until the RNI is ready to send, and (3) initiate transmission to the network. It is important that the execution time of the `init` function is not longer than the heartbeat period. The second function, `px_main(void)`, is executed once every heartbeat period. The structure of the `px_main(void)` file is slightly more complex. Firstly, the RNI is checked if a message was received. This is done by calling the driver macro `NOC_RNI_CHK_MSG`. If there was no message received, this is interpreted as an *absent* value in the synchronous model of computation (or a heartbeat violation occurred). In these cases the `else` branch is executed. In the case of a message was received, the message is read from the receive buffer of the RNI, processed and written back to the send buffer of the RNI. As soon as the RNI is not busy with sending any previous value (`NOC_RNI_STATUS`), the transmission can be started. Finally the flag signaling a received message must be reset by `NOC_RNI_CLEAR`.

1.2.2.2. The scheduler

The *Scheduler* (`synchronous_MoC_main.c`) contains the `main()` function of the node's software. It has two simple functions: Synchronization with the heartbeat and executing the processes in the correct order. Additionally to the generated files, the NoC System Generator copies the driver (`kth_avalon_noc_rni_regs.h`) and the target code files belonging to the node to the software project folder. (Should we put an example?)

1.2.2.3. System Description Header

The file `software_configuration.h` is generated by the network generator. One of these system descriptions exist for each node. It is a C header file that specifies the system specific parameters relevant to the software running on the node. To use the network functionality by a processes, this file must be included in the code. The device driver is included by the system description header, and it is not necessary to include it separately. It contains all the PIDs of the processes running in the system, and names are assigned to the receive and send channel numbers to make their usage readable, e.g. `recv_channel_p0_from_p1` for the channel on which process 0 receives data from process 1. The hardware address of the RNI is mapped to

NOC_RNI_BASE, which is used to access the NoC with the commands of the device driver. The names of the nodes' hardware resources are also mapped to names that are easier to read, as they contain the node name.

1.2.2.4. Device Driver

The network operations are abstracted by a device driver. In the actual version for the Altera Nios II processor the driver is a single .h-file called kth_avalon_noc_rni_regs.h. Before including this file to the source code the base address (NOC_RNI_BASE) of the RNI must be set. This section describes shortly the macros provided for the user. The parameter base – used by every command – is the base address of the RNI.

NOC_RNI_SEND(base,priority,spid,dpid,buf,msg_size) invokes the RNI to send the message stored in the channel buffer (buf) to its destination process defined by its PID (dpid). The PID of the sending process is indicated by the spid parameter. The message size is given by the msg_size parameter and a priority is assigned by the priority parameter.

NOC_RNI_STATUS(base) checks the status of a the preceding send command. Returns '1' as long the RNI is busy.

NOC_RNI_CHK_MSG(base, channel) checks for a received message channel. Returns '1' if there is a new message in the buffer.

NOC_RNI_CLEAR(base, channel_id) clears the 'message received' flag of the channel (channel_id). Should always be called after the buffer was read.

NOC_RNI_MSG_LENGTH(base, src) returns the message length of the received message in the channel indicated by src.

NOC_RNI_DEST_PID(base, src) returns the destination PID of a packet in the receive buffer src.

NOC_RNI_NODE_NR(base) returns the number of the node on which a process is running.

NOC_RNI_SRC_PID(base, src) returns the source PID of a packet in the the receive buffer src.

NOC_RNI_READ_SYNCHRONIZER_FLAG(base) reads the synchronization flag that indicates the occurrence of a synchronization event on the network.

NOC_RNI_CLEAR_SYNCHRONIZER_FLAG(base) clears the synchronization flag. This must be done before a new heartbeat occurrence can be registered.

1.3. Generated NoC performances, power and area consumption

The generated NoC connects switches with channels which are 32 bits wide. Each switches is clocked with a 50 MHz clock. Consequently, the NoC has the following *channel bandwidth* (b):

$$b = 32 \cdot 50 = 1600 \ \frac{\text{Mbits}}{\text{s}} = 200 \ \frac{\text{MBytes}}{\text{s}} \tag{1.1}$$

In a 2×2 NoC this would mean a *bisection bandwidth* (Bb) equal to $2 \cdot b = 400 \ \frac{\text{MBytes}}{\text{s}}$

However, if we include the limitation of the current RNI, which can inject 1 packet every 4 switch cycles (where each switch cycle is 4 clock cycles), we get the following channel bandwidth:

$$b_{withRNI} = \frac{32}{4 \cdot 4} \cdot 50 = 100 \ \frac{\text{Mbits}}{\text{s}} = 12.5 \ \frac{\text{MBytes}}{\text{s}} \tag{1.2}$$

This limitation has been included in order to reduce the injection rate below the deflection knee.

2 Getting Started Tutorial

2.1. Overview

When you start the GUI, the following window appears:

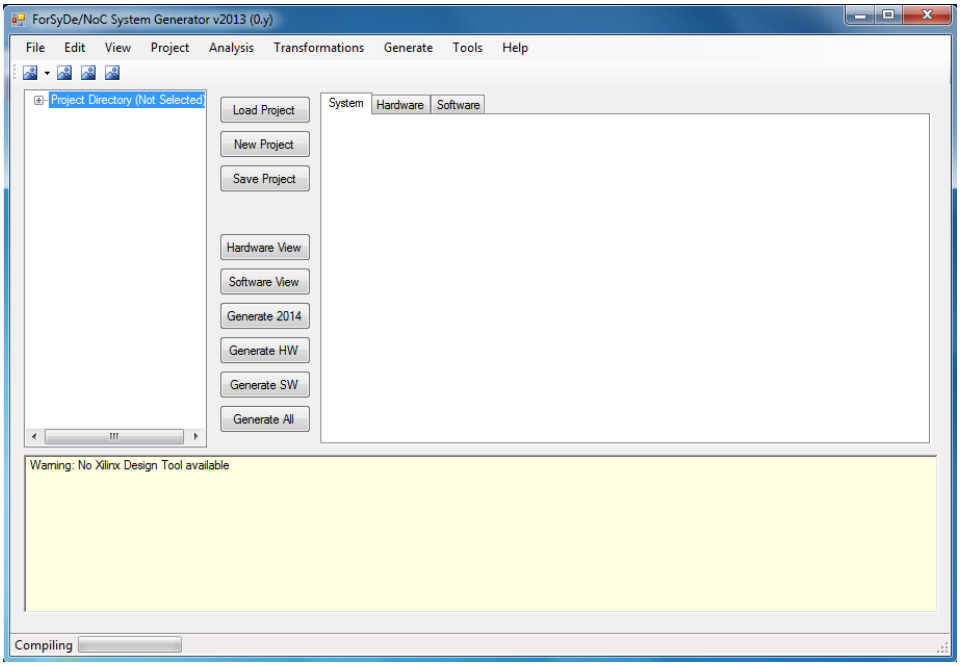


Figure 2.1.: *Main window*

2.1.1. Main window menus

A list of menus is available on the top. They are described later in the sections listed below.:

1. File – enable to save and load projects

2. Edit – not used in this tutorial
3. View – switch between SW and HW view
4. Project – not used in this tutorial
5. Analysis – not used in this tutorial
6. Transformations – configure hardware accelerators instead of PEs
7. Generate – generate HW and SW components
8. Tools – not used in this tutorial
9. Help – not used in this tutorial

2.1.2. Quick access buttons

The main window has the following *quick access buttons* that can be used to quickly access the most commonly used design steps:

1. Load Project – Opens an existing project
2. New Project – Creates a new project
3. Save Project – Saves the current project[1]
4. Hardware View – Opens the Hardware View
5. Software View – Opens the Software View
6. Generate HW – Generates all necessary HW files for synthesizing the project using the backend FPGA tool vendor (Altera Quartus or Xilinx ISE)
7. Generate SW – Generates all necessary SW files for compiling using the backend FPGA tools SDK
8. Generate All – Generates all necessary HW and SW files for backend synthesis/compilation using the target FPGA tool vendor

[1] It is suggested to use this button frequently, it should be used before running any Generate command.

2.1.3. Information windows

Besides the Buttons and Menus, there are also three *information windows*:

1. The project window listing all files associated with the project
2. The main information window, having three tabs:
   a) The System Tab - displaying the content of the current active system .xml file, used as intermediate representation of the whole NoC-based MPSoC system.
   b) The Hardware Tab – displaying information and properties (area estimates etc.) of the target HW implementation.
   c) The Software Tab – displaying information and properties (Worst Case Execution Times etc.) of the target SW implementation.
3. The console window – Displaying the result of the latest command.

2.1.4. A toy example: the Ring system

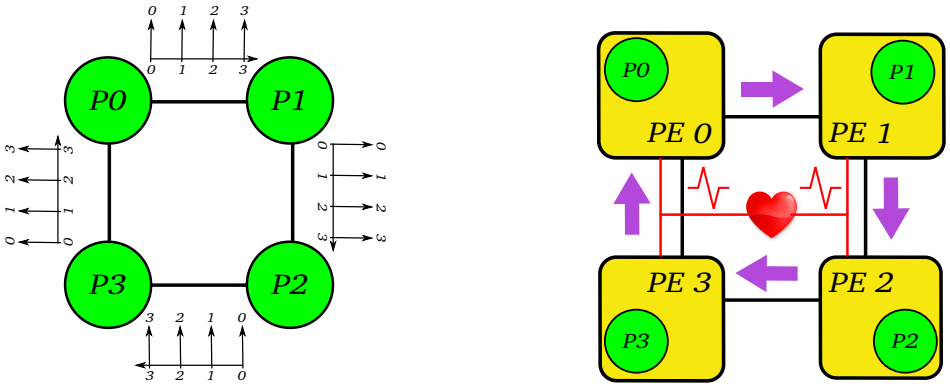


Figure 2.2.: *System-level model of the Ring and its implementation on MPSoC*

In this first tutorial we will create a very simple system, shown in Figure 2.2. The left picture represent the functionality of the system using a process network (system-level model). Each of the four processes sends to the following process an initial value 0. Then, the receiving process receive the value, increase it by 1 unit, and sends the result on the following event. So, we expect each process to send (and receive) in order the values 0,1,2,3,... This system is referred to as *Ring*.

The Ring example will be executed on a platform composed by 4 processors, connected through a 2×2 NoC. Each processor will run one of the processes composing the Ring example, as shown on the right picture in Figure 2.2.

2.2. Design flow

To create a design using ForSyDe NSG, the following design flow should be followed:

1. Create Project
    a) Select target technology
    b) Decide NoC properties (Size, Topology, RNI type etc.)
2. Open Hardware View - Edit HW/node properties
3. Open Software View - Add, place and edit SW processes
4. Generate HW and SW for target platform
5. Synthesize platform for target technology
6. Use target tool SDK (or online system) to compile, download and debug SW.
7. If necessary, repeat steps 3-7 until system performance is satisfactory

2.2.1. Creating a new project

Start the NSG tool double clicking on the Desktop icon, or double clicking on \Ring2x2\bin\Windows\GUITest.exe. To create a project, follow these steps:

1. click on the *New Project* quick access button. A new form asking for a project file name will open;
2. browse to a folder where you want to save the project files. From now on, we will refer to this folder with the term `workspace`. A typical workspace can be a folder such as \Ring2x2\Examples\test\. Before clicking on the Save button, insert the name of a XML file which will be used as project file, to save your project configurations. This step is shown in Figure 2.3. Note: the workspace folder will contain only the XML intermediate format file, not the output files of the NSG!
3. After pressing the next button, we have to configure some project settings in the project settings dialog. Configure your project as shown in Figure 2.4. A short explanation of each field is described in the following list:
    a) `System Name` – It is recommended that **the system name has the same name as the project XML file** (see step 2) to avoid problems during VHDL/FPGA backend compilation.
    b) Board – You may only use one of the boards listed in the dropbox menu

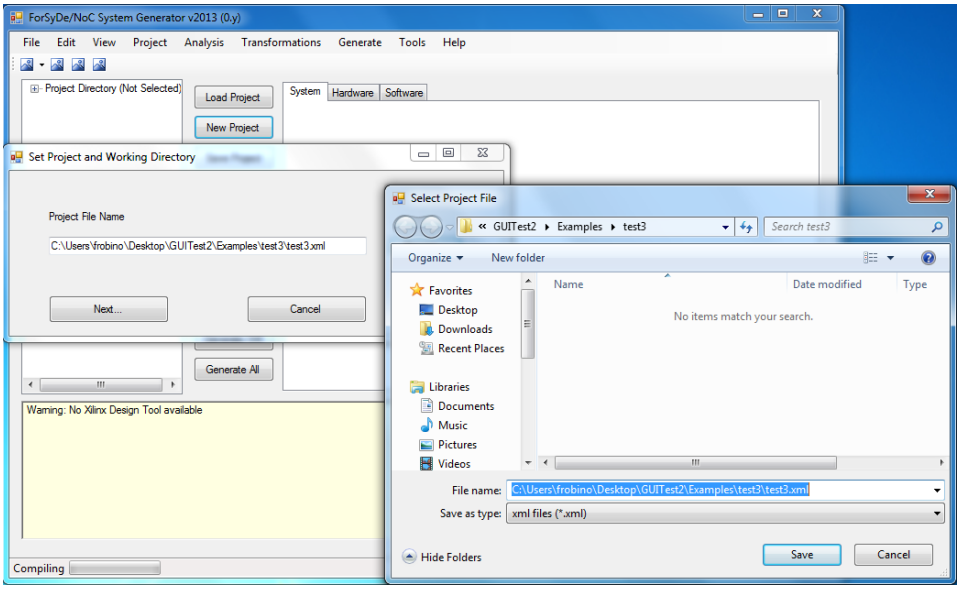


Figure 2.3.: *Creating a new project: workspace*

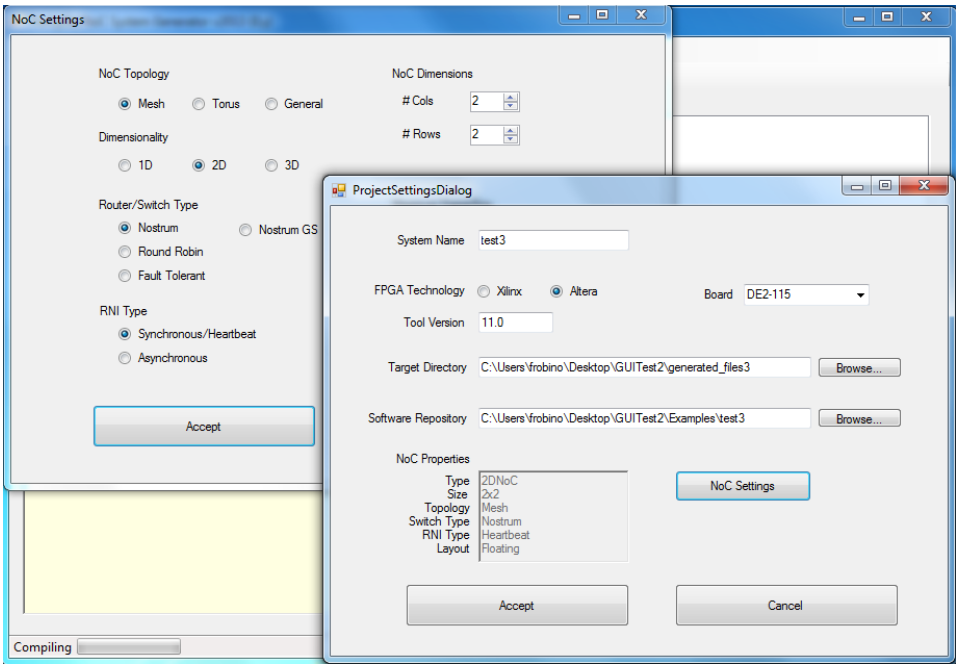


Figure 2.4.: *Creating a new project: project settings*

c) `Target Directory` – is the directory where the NSG tool will store/-generate all the necessary files to create a project for the target FPGA vendor

d) `Software Repository` – is the directory from which the NSG tool will search for the software files describing the functionality of your system. You could use a custom directory on you PC, but in this first tutorial we suggest to use the same `workspace` location you used earlier. Doing so, the XML file and the C files representing the functionality of the system will be saved in the same folder, in the `workspace`.

e) NoC Properties – shows the properties if the NoC interconnection. By default the properties are configured for a 2×2 mesh NoC, where a single PE (Nios or MicroBlaze) is connected. NB: both PEs and NoC properties can be edited later in the flow.

4. Optional - From the project settings dialog, if you click on the NoC settings button you can change the NoC interconnection characteristics. NB! Only 3D and 2D Mesh Type NoCs of Nostrum type are currently available. For this getting started tutorial, just double check that the settings are the same as Figure 2.5. Check especially the Maximum Frame Size field: it should be set to 64. If it is not, please change it to 64, so that we'll have a good margin when "playing around" with the system. A short description of the of the parameters

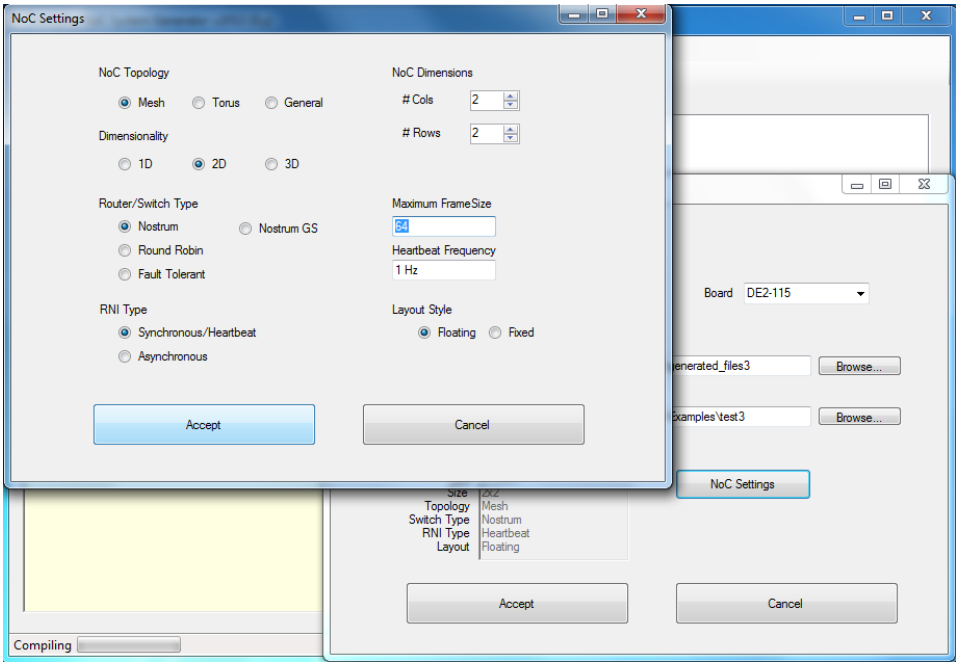


Figure 2.5.: *Creating a new project: NoC settings*

of the NoC which can be customized is in the following list:

a) Topology - Currently, only the Mesh NoC is available, with Dimensionality 2D and 3D.

b) Switch Type – Currently, only the Nostrum is available

c) Maximum FrameSize – The largest amount of information (in words) you ever intend to send over the NoC in this implementation. The minimum value is 3. The Maximum value is 512. Use 64 (or 42) if you do not know what value to set.

d) RNI Type – Select Synchronous for Real-Time Heartbeat applications, and give the Heartbeat Frequency for your system. A small value of 1 Hz is good for debugging onboard applications. A large value of 10-100 kHz is good when debugging IPs using VHDL simulations.

e) Layout Style – Currently, only Floating is available.

5. Save the project before you start with next step by pressing the *Save Project* button (if there is an asterisk in the System Tab – System* your project has been modified and needs saving). After you have saved, the main window of the NSG tool should look as the one in Figure 2.6.

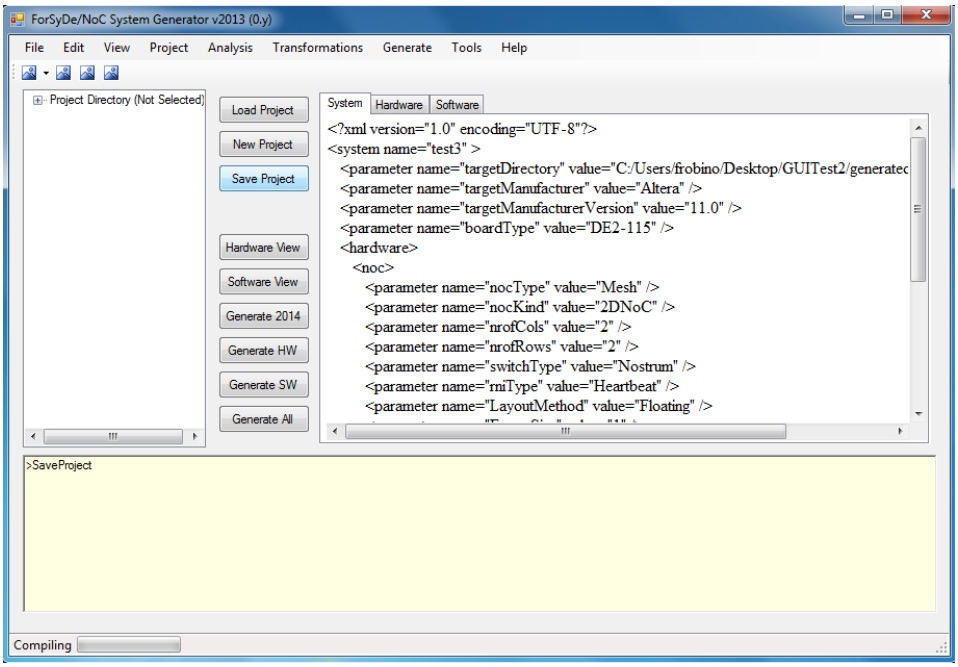


Figure 2.6.: *The main window after the project has been created*

2.2.2. The Hardware view

Once the project has been initialized we can customize HW and SW configuration of the NoC-based MPSoC we want to generate. In this part of the tutorial we will

configure the HW part. We recall from Section 2.1.4 and Figure 2.2 that **our goal is to create a multi processor system composed by 4 PEs, connected through a 2×2 NoC**. To do so, click on the Hardware view button in the main window. The hardware view form will open, and it will look like Figure 2.7.

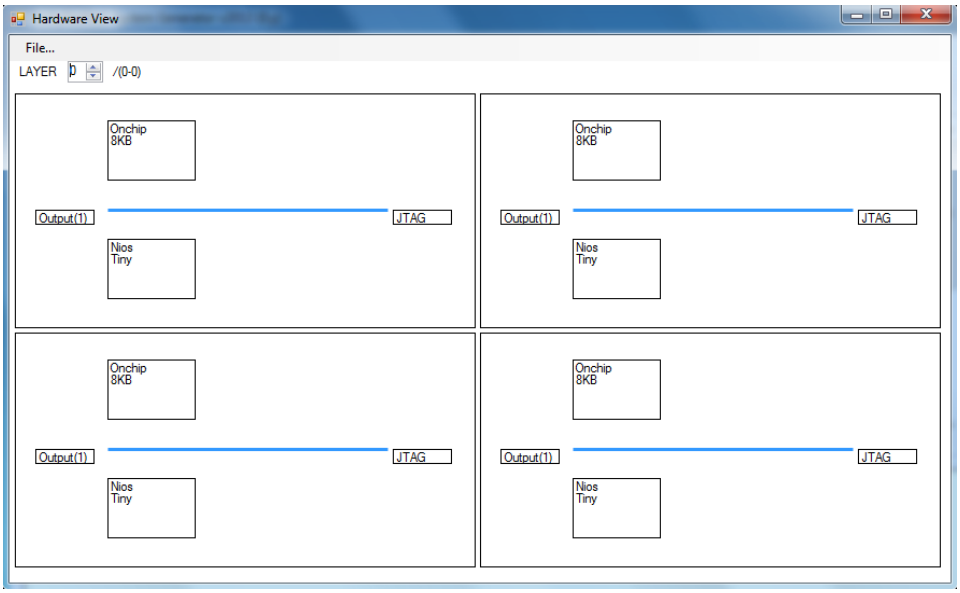


Figure 2.7.: *The Hardware view form*

The figure gives an abstract representation of the platform we want to create. It is composed by 4 rectangular areas, each of them modeling one of the 4 nodes of the 2×2 NoC. By default, each node is configured with a Nios/s core, a jtag unit, 8 KB of on-chip memory (which will contain the code of the application running on the Nios), and 1 IO, which can be connected to a LED.

We do not have to modify this configuration during this first tutorial, however, if you double click on one of the rectangular areas representing the node, you will enter the Node entry form, shown in Figure 2.8, which permits to customize the selected node.

Through the Node entry form it is possible to include more PEs in the same node, or to select different processor types (Nios2/s, Nios2/f, Microblaze or Leon3) to create an heterogeneous MPSoC. It is also possible to configure the size of the on-chip memory, to increase the number of IOs, or to add your custom IP block in the system. In this first tutorial we do not need to change the configuration here, so leave it as shown in Figure 2.8 and close the Node entry form and the Hardware view form, so that you come back to the main window.

When you are back in the main window, have a look at the Hardware Tab. As shown in Figure 2.9. This tab is constantly updated every time you change a parameter

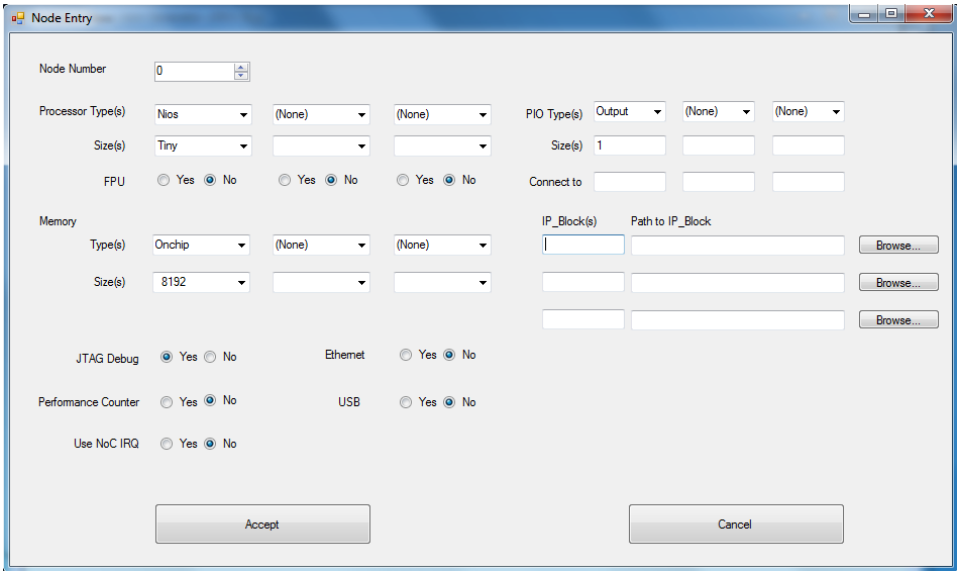


Figure 2.8.: *The Node entry form*

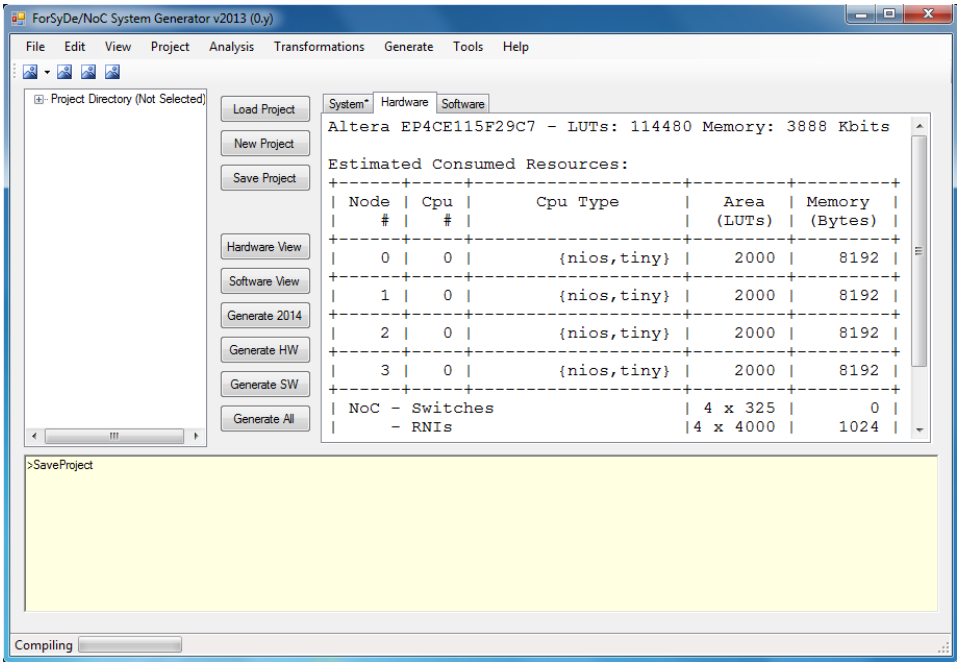


Figure 2.9.: *Hardware Tab, with estimation of consumed resources for the system modeled in the HW view*

in the Hardware view form, so that you can have an estimation of the resources required by your design, and avoid to try to compile designs which will not fit on your target FPGA.

2.2.3. The Software view

Once the configuration of the NoC-based MPSoC has been specified, we have to describe the functionality that the system should implement and specify how this functionality is distributed onto the PEs of the MPSoC. This is done using a system-level design entry, where the system is described in terms of abstract functions and interconnected blocks. Blocks will be used to specify the execution semantics of the functionality, which is described through C files. This is done using the Software view form, which can be accessed pressing the quick access button SW view, shown in Figure 2.10.

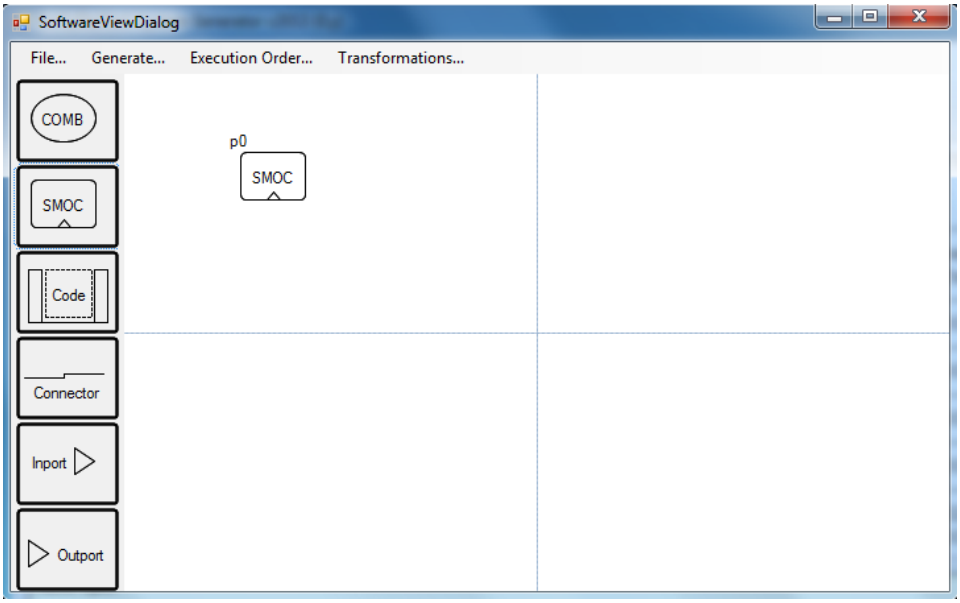


Figure 2.10.: *The SW view, where we describe a system in terms of abstract functions and interconnected blocks*

We recall from Section 2.1.4 and Figure 2.2 that **our goal is to create a system composed by four processes (tasks), connected in a circle (as a ring)**. Each of the four processes sends an initial value 0 to the next process. The receiveing process receives the value, increase it by 1 unit, and sends the result on the following event. In this example, the event is a Heartbeat tick, i.e., each processes will execute its functionality and send the result to the following process when triggered by a HeartBeat tick. For debugging purpose, we have selected a HeartBeat frequency of 1Hz, i.e. each process will execute once every second. See the HeartBeat Frequency setting in Figure 2.5.

The software view provides different blocks (called *process templates*) on the left column. Process templates are the blocks which provide the execution semantics to a specific function. In this case we want a process template which is stimulated

to execute on the synchronous HB tick event, and which delivers its result to the following process on the next HB tick event. This behavior (semantics) is given by the *SMOC* process template.

You instantiate a SMOC process by clicking on its template in the left column, then drag and drop it in the region you would like to have it. The SMOC is released when you click again. For example, click on the SMOC process template in the left column, move it to the upper left rectangular region, and click again to release the process template in that region, reaching the exact same situation shown in Figure 2.10.

What you are doing here is to map a specific process (task) on a specific node of the MPSoC platform. In fact, the 4 rectangular regions in the Software view represent the 4 nodes of the 2×2 NoC-based MPSoC created through the Hardware view. Repeat the previous procedure to add other SMOC process templates in the other three regions[2]. You should get the software view looking like Figure 2.11. The lines

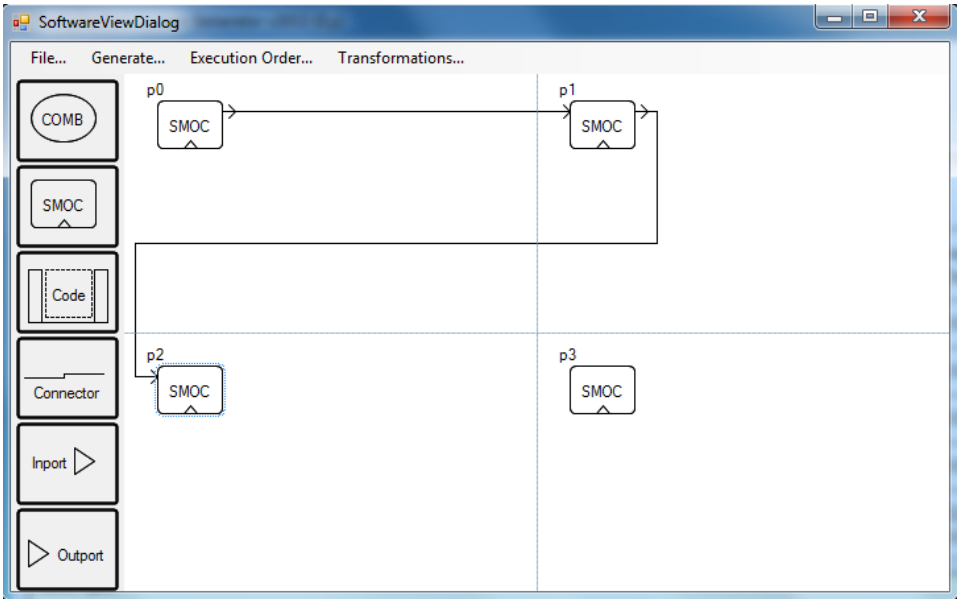


Figure 2.11.: *Adding SMOC process templates and connecting them*

between process templates shown in Figure 2.11 are connectors, showing the flow of the data through the processes. Click on the connector button on the left column of the software view, and then click on the right side of the SMOC placed on Node 0 (p0) and click on the left side of the SMOC placed on Node 1 (p1). Continue connecting p1 to p2, and you will reach the exact same situation as the one drawn in Figure 2.11. To complete the Ring, continue connecting p2 to p3 and p3 to p0.

[2] If you drop SMOC process templates wrongly, you can always click on them and press the Delete button on your keyboard to delete them.

Once this is done you can save your project clicking on File and then Save, as shown in Figure 2.12

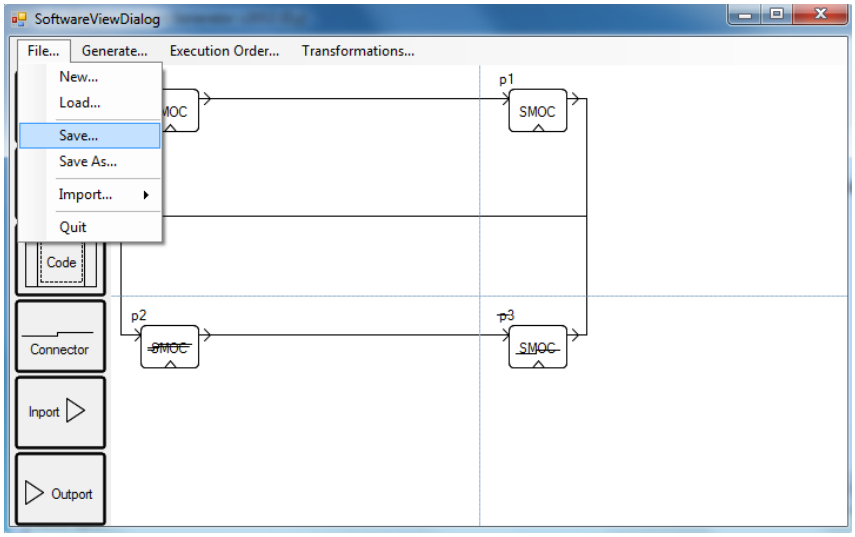


Figure 2.12.: *Saving the SW view*

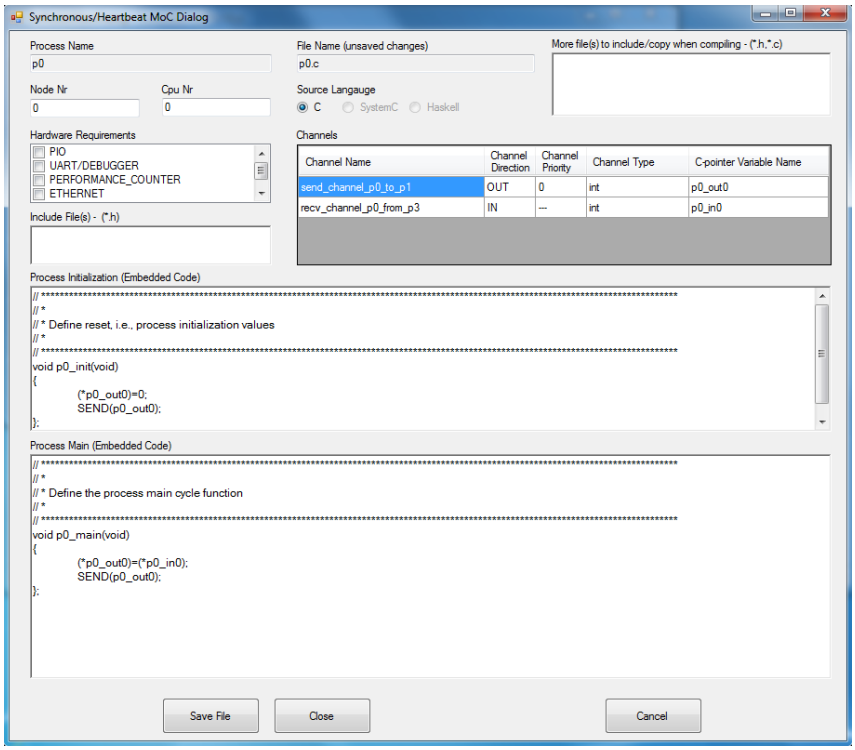


Figure 2.13.: *The dialog to describe the functionality of each process/task*

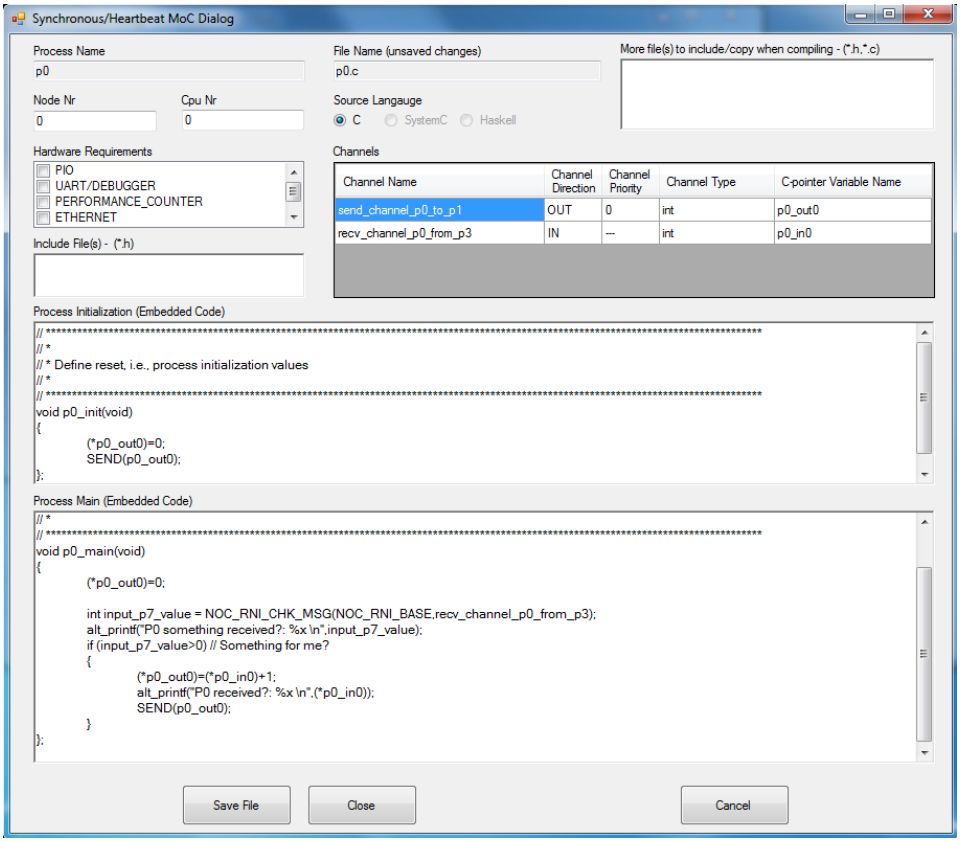


Figure 2.14.: *Customizing the functionality of each process/task*

Up to now, we have just described the mapping of processes/tasks onto the NoC-based MPSoC platform. However we have still to specify the functionality of each process, which is: receive the input, increase it by one unit, send it to the following process. This is done by double clicking on the SMOC process templates we have placed in the four regions of the Software view. For example, if you double click on the SMOC p0, you will get the form shown in Figure 2.13.

In this form you can specify the functionality of each task, adding your own C code. Note that the NoC system Generator has created for you already a lot of code, which is used to synchronize the process execution on the HeartBeat ticks, automatically send and receive data after being triggered from the HeartBeat tick, etc. What you need to do now is only to add your own custom functionality. For example, if you modify the code in the Process Main window as shown in Figure 2.14, you will add your own functionality (increase by one), plus some other debug print statements.

Once you have modified the functionality of p0 as shown in Figure 2.14, you can click on save file. This will save the functionality of the process as a C file in your

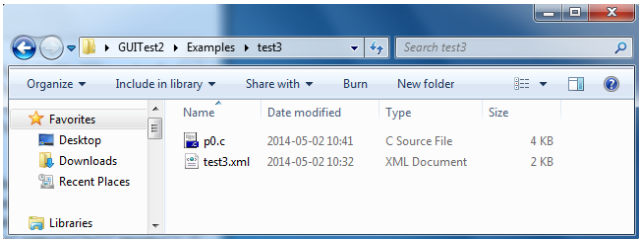


Figure 2.15.: *Functionality of p0 saved as C file in your workspace*

`workspace` folder, as shown in Figure 2.15.

Modify the other SMOC processes, and coustomize their main C file with the code provided in Appendix A.

Once you have saved the process functionality for all 4 processes (p0 to p3), close the Software view to return to the main window. Now you have described the overall system, and it is time to run the NoC system Generator tool to automatically generate a working prototype!

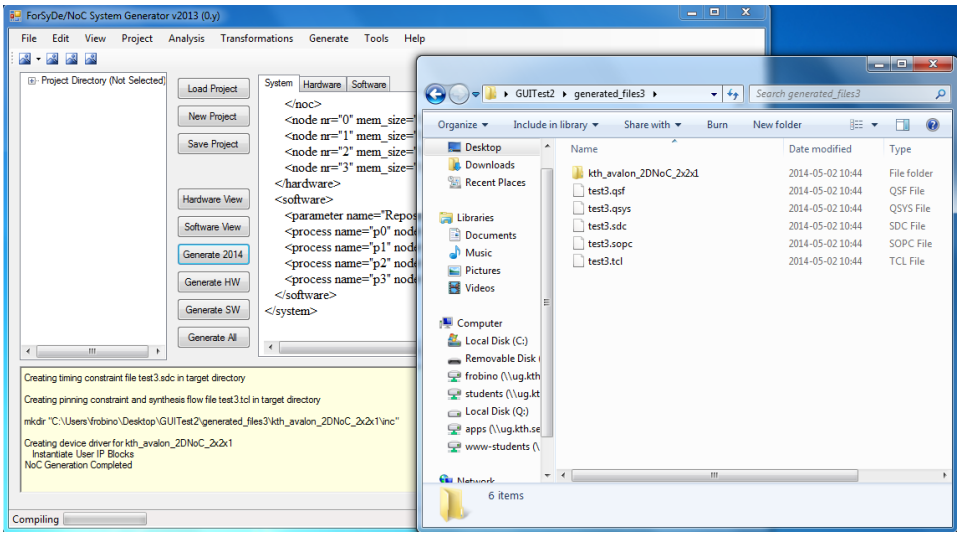


Figure 2.16.: *Automatically generate HW and SW with the NoC system Generator*

Before you move to the next step, please check that the XML file represented in the XML Tab of the main window is exactly the same as the one shown in Appendix A. If it is not, you can edit the XML file manually, or use the HW view and SW view of the tool to modify it following the steps presented in this guide. Then, from the main window click on the button Generate 2014. As shown in Figure 2.16, this will generate the Hardware description files in your `Target Directory` folder. Please, check that they have been generated properly. Then, from the main window press the button

Generate SW. In the same folder, a new folder containing the software files will be generated, Again, check that the folder is there before proceeding with the next step. You can close the NoC system Generator main window now.

2.2.4. Synthesize the platform to FPGA

After the completion of the previous step, we have generated a model/image of the HW and SW in the target platform's language. In this specific case it is composed by:

1. hardware:
   a) VHDL files, describing the on-chip interconnection (NoC) coherent with the user specification;
   b) `.sopc` and/or `.qsys` files, used by the Altera tool suite to interconnect resources (Nios2 processors) to peripherals and to the NoC. This files provide a high level model of the multi processor platform;
   c) `.qsf` file, providing the pin assignment for the selected FPGA board;
2. software:
   a) drivers for the multi-processor platform, providing to the user an API to access the platform services (e.g. send/receive messages)
   b) the schedulers (one for each Nios resource) scheduling the execution of the functionality on the HB event.

Feel free to browse the generated files in the `Target Directory`.

Initially you will configure the FPGA as a multi-processor platform. This can be done using the Quartus tool suite.

1. start the Quartus II software double clicking the icon on the Desktop of your computer. If the icon is not there, click on the Windows start button and search for Quartus II, then click on the icon;
2. when Quartus opens you can click on the New project wizard button, or simply select File – new project.
3. in the Directory, Name Top-Level Entity form, shown in Figure 2.17, specify as working directory your `Target Directory`, the one containing the output files of the NSG. As project name and top-level design entity, specify the `System Name` you specified in the NSG project (see Section 2.2.1), which is the same name of the .sopc or .qsys files in the `Target Directory`. Then press Next;
4. in the Family and Device settings form, select the exact same configuration as shown in Figure 2.18. Then press Finish;

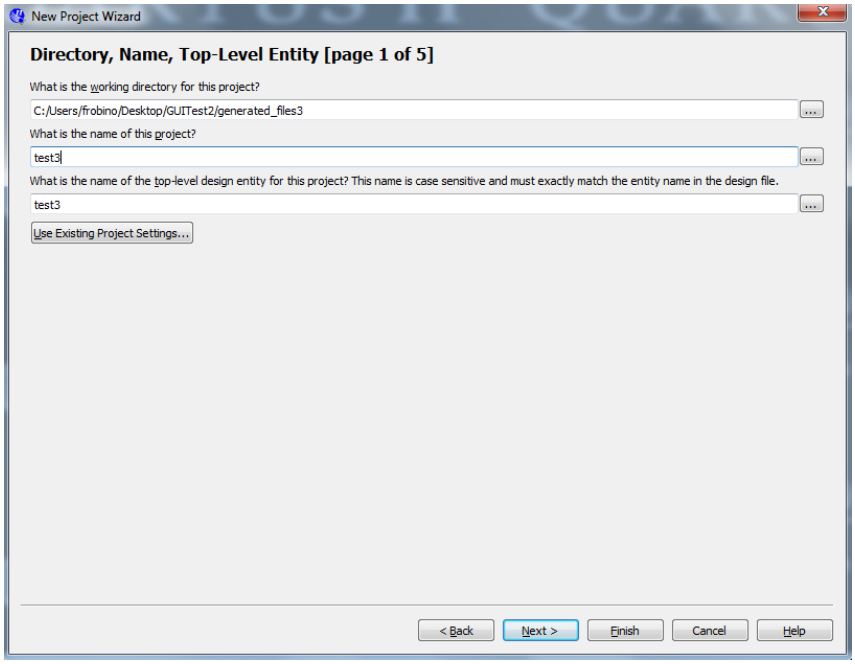


Figure 2.17.: *Specify directory and project name in Quartus II*

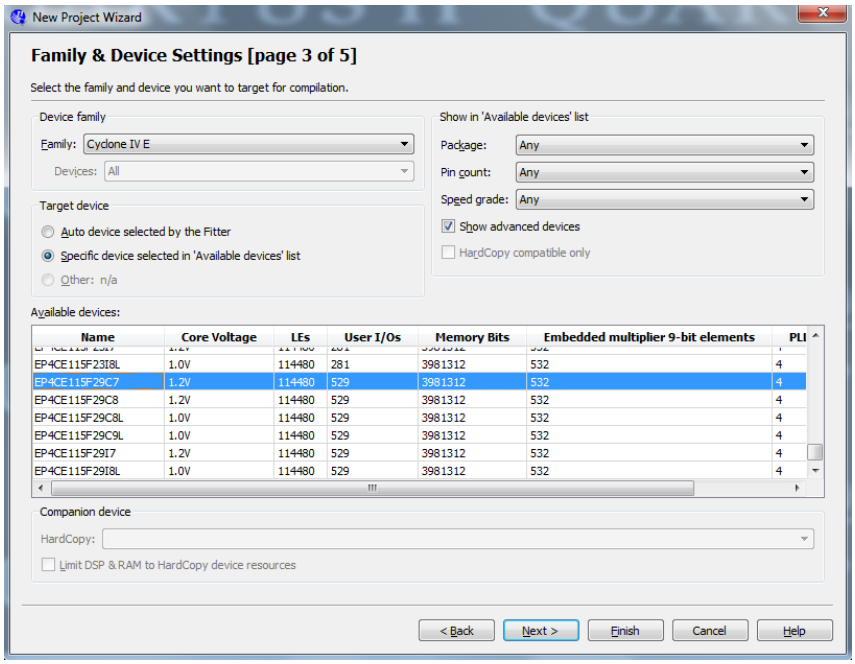


Figure 2.18.: *Target FPGA board settings in Quartus II*

5. once you are back in the Quartus main window, click on Tools – License setup to access the License manager. As shown in the top of Figure 2.19, add the

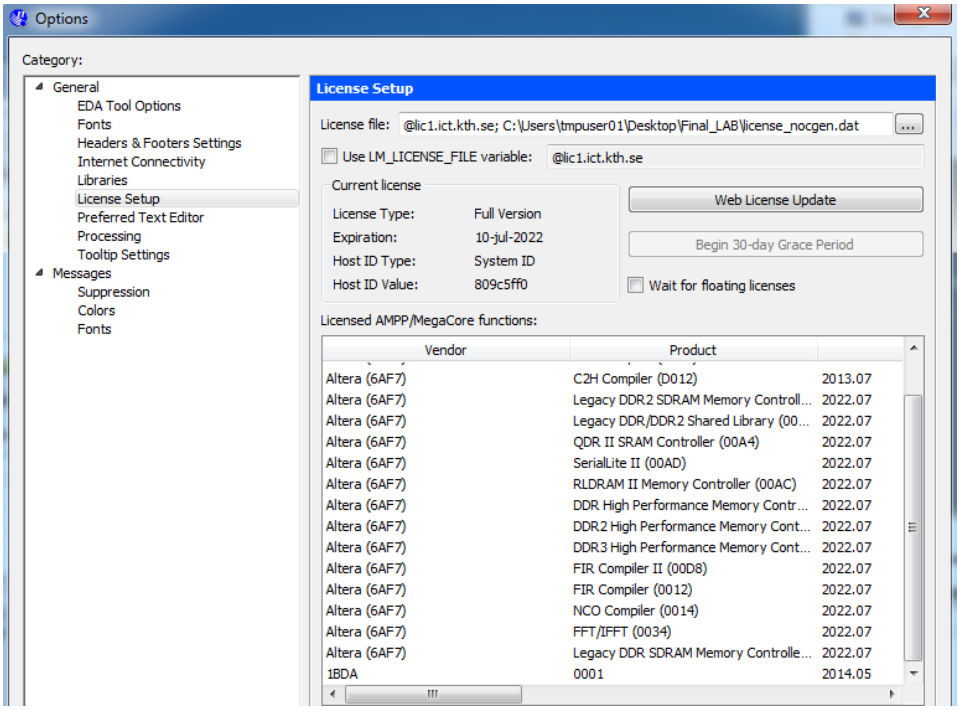


Figure 2.19.: *Setup the NSG license in Quartus*

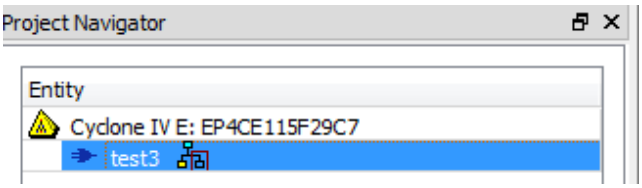


Figure 2.20.: *Open your project in SOPC Builder*

path to the license_nocgen.dat file in the License file field and check that the Vendor 1BDA Produce 0001 is in the list of licensed functions. You can find the license_nocgen.dat file in your Final_LAB folder;

6. from the Quartus main window, double click on the project in the window (shown in Figure 2.20) to open the SOPC Builder;

7. analyze the system (it is also shown in Figure 2.21): it is composed by 4 Nios processors, each one of them connected to a scratchpad memory and some peripherals, as specified through the GUI. A NoC component interconnects the resources. Everything has been created and connected automatically through the NSG tool in the previous step;

8. click on the Generate button to start the generation of HDL code describing the modeled system;

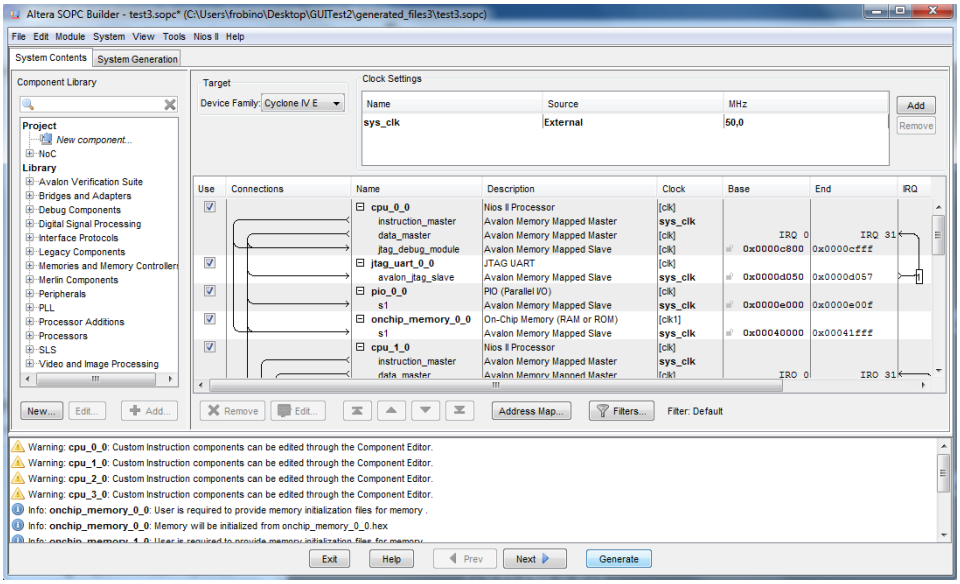


Figure 2.21.: *SOPC Builder representation of the system modeled through NSG*

9. once the generation is successfully terminated, close (and, if required, save) the sopc project;

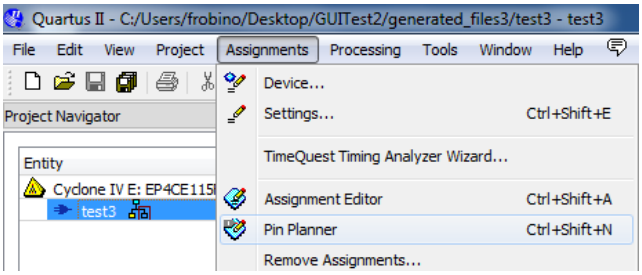


Figure 2.22.: *Access the pin planner*

10. back again in the Quartus II tool main window, click on Assignment – pin planner (Figure 2.22). In this window you can notice that the pin assignment has been done automatically for you, as shown in Figure 2.23. Close this window after you checked it matches with Figure 2.23;

11. back once again in the Quartus II tool, double click on "compile", shown in Figure 2.24. This will start the process of generating a bitstream file to configure the FPGA as a quad-processor platform, in the form of a `.sof` file. This operation will take around 10 minutes to complete, so take this occasion to ask questions or grab a coffee;

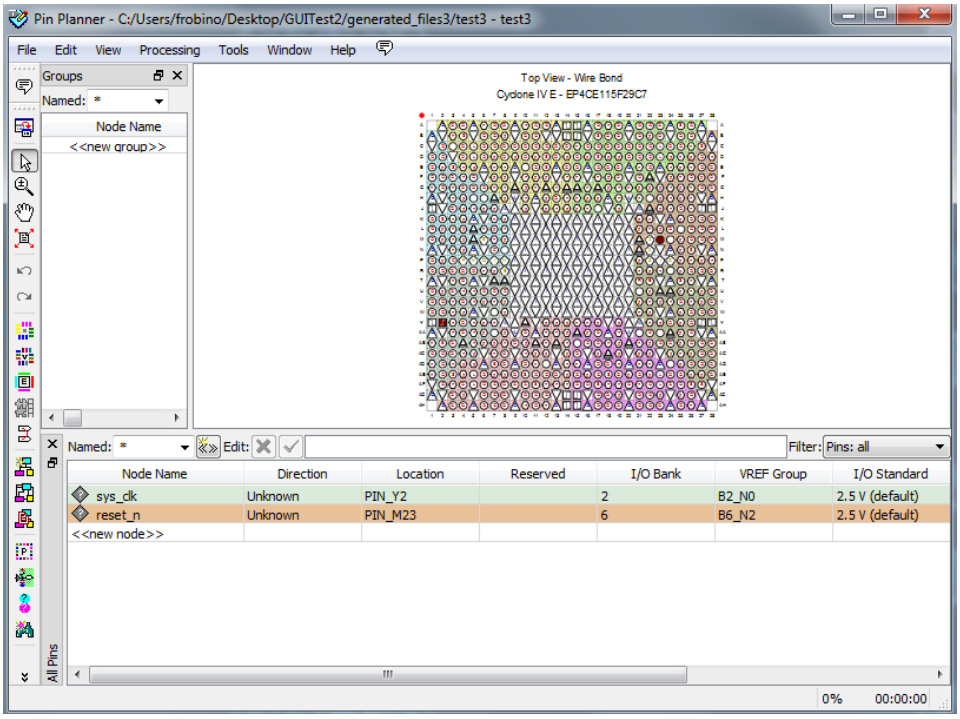


Figure 2.23.: *Pin assignment has been done automatically by NSG*

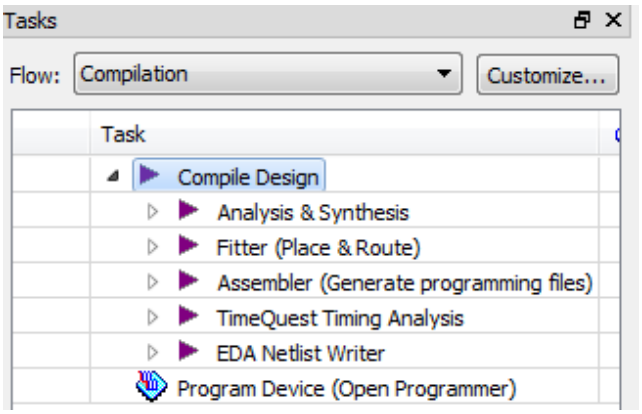


Figure 2.24.: *Compile the project with Quartus II*

12. once the compilation is successful, you can close the Quartus tool and get ready to program your FPGA. Check that your FPGA is ON and it is connected though USB to the PC. Check also that the USB cable is connected with the Blaster port of your FPGA, and not with the Device one. Then, from the desktop of your PC, click on the Windows start button and search for NiosII shell. Open one of this shell and browse to the `Target Directory` containing

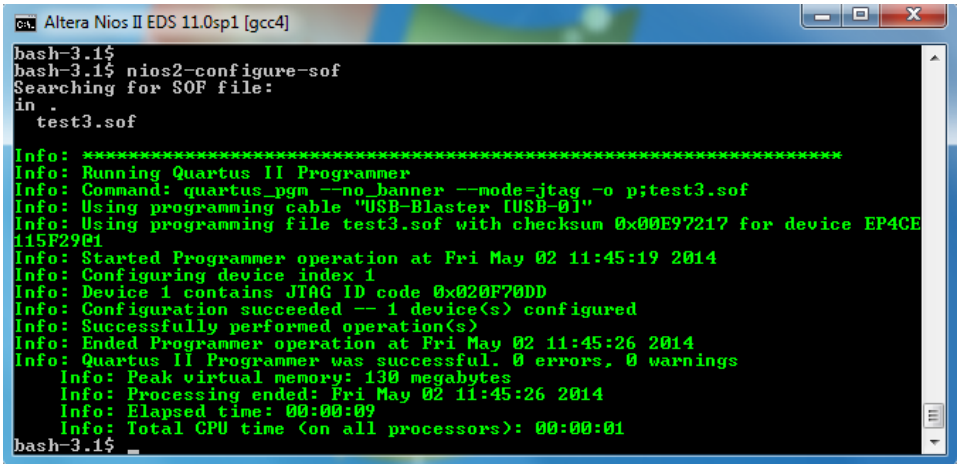


Figure 2.25.: *Configure your FPGA as 4 processors system using Nios shell*

the Quartus project and a .sof file. Run the command `nios2-configure-sof`[3], as shown in Figure 2.25. You have successfully configured your FPGA as a quad-processor system.

2.2.5. Compile, download and debug SW

In this part of tutorial you will use the generated SW (driver, schedulers and IF-C files). You will compile the C code targeting each processor and download the compiled code on each of the processors. Once the software is downloaded, the system is completed and becomes fully functional.

In order to accomplish this step, there are 2 alternatives:

1. use the Nios shell, a command line interface. This alternative is faster and can be efficiently integrated in a set of scripts, however does not have the benefits of a user interface;
2. use an Eclipse based environment, to configure the processors one by one;

2.2.5.1. Nios shell

The 4 processors platform generated through the Altera tools is configured using the following notation:

1. CPU_0_0, jtag_0_0, cpu_id 1, jtag_id 1
2. CPU_1_0, jtag_1_0, cpu_id 2, jtag_id 2

[3] Another way to complete this step is to use the programmer tool in Quartus II to program the FPGA with your brand new `.sof` file.

3. CPU_2_0, jtag_2_0, cpu_id 3, jtag_id 3
4. CPU_3_0, jtag_3_0, cpu_id 0, jtag_id 0

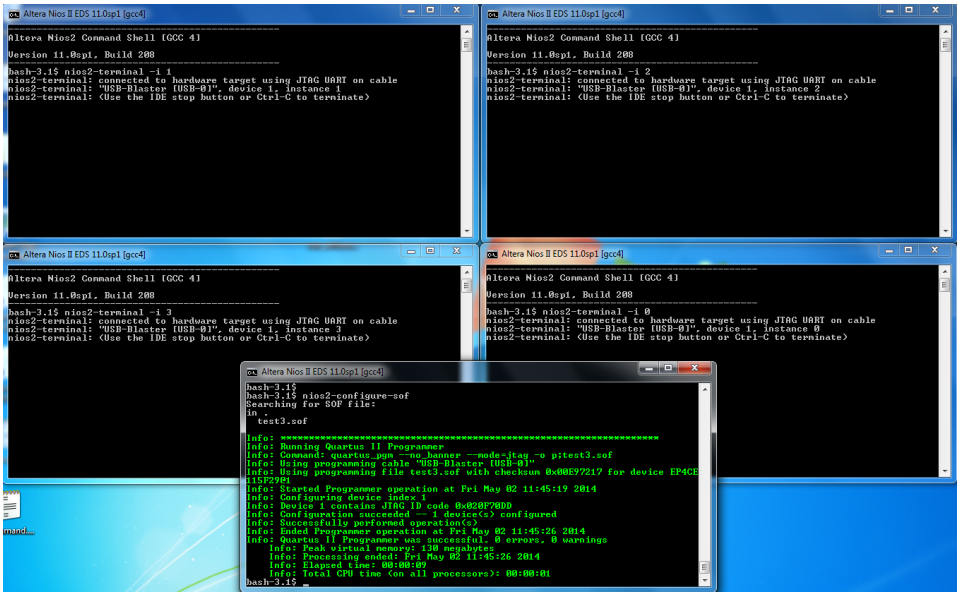


Figure 2.26.: *Use Nios shells to compile, download and debug the 4 processors platform*

Open 5 Nios shells (double click on the Desktop icon 5 times, or search it through the Windwos button). 4 shells will be used to connect to the 4 processors, 1 shell to execute the commands. As shown in Figure 2.26, for each of the 4 shell dedicated to one of the processors, run 1 of the following commands:

- to connect to CPU_0_0: `nios2-terminal -i 1`
- to connect to CPU_1_0: `nios2-terminal -i 2`
- to connect to CPU_2_0: `nios2-terminal -i 3`
- to connect to CPU_3_0: `nios2-terminal -i 0`

You are now connected to the 4 processors through 4 terminals. You will use the 5th shell to compile and download the generated `.elf` files (the compiled code) on each processor. First, compile the software files using the *run_software.sh* script. You find this file in the Ring_2x2 folder. Copy this file in the folder containing the generated files (`.sopc, .qsys, .sof`), which contains also the Software folder. Open the *run_software.sh* using a text editor, and check that the field `system name` matches your system name. If not, edit it so that it does. Then, use the Nios shell to browse to the folder, and execute the script through the command:
`./run_software.sh -b`
The compilation takes a while but it creates a new folder called Software_projects

containing the compiled software (.elf files). Then, to download the compiled software on the different Nios processors, you use the following commands, in sequence:

- to download the compiled software to CPU_0_0: `nios2-download -g -i 1 <path_to_file>/Node_0_0.elf`
- to download the compiled software to CPU_1_0: `nios2-download -g -i 2 <path_to_file>/Node_1_0.elf`
- to download the compiled software to CPU_2_0: `nios2-download -g -i 3 <path_to_file>/Node_2_0.elf`
- to download the compiled software to CPU_3_0: `nios2-download -g -i 0 <path_to_file>/Node_3_0.elf`

where $< path\_to\_file >$ is the folder path where the `.elf` files are contained (i.e. `C:\NsG\GeneratedFiles\Software_projects\Node_0_0\Node_0_0.elf`). You should see the system start to run on the 4 terminals.

2.2.5.2. Eclipse based environment

This part is not yet documented.

3 Synthesize a Simulink model

3.1. Design flow overview

Step 1

Create your system-level model with Simulink;
Simulate the system using Simulink;
Use the Simulink Embedded Coder to generate C code which models the functionality of the system;

Step 2

Use the user interface of the NoC System Generator to generate an XML description of the target platform;
Use the NoC System Generator to:

- generate the HDL and project files enabling Quartus to configure the FPGA as an HeartBeat compliant multi-processor system;
- generation of process wrappers (main C files scheduling the C code modeling the functionality of the system);

Step 3

Extraction of `rt_onestep` function from the Embedded Coder generated C files;
Embed the `rt_onestep` function in the process wrapper, scheduling its execution on the HB ticks;

Step 4

Compilation of the HDL for FPGA;
Compilation of the C code for each PE;

Step 5

Configure the FPGA;
Download and run the compiled SW for each PE;
Collect the prototype results and compare with the initial simulation;

3.2. Step 1

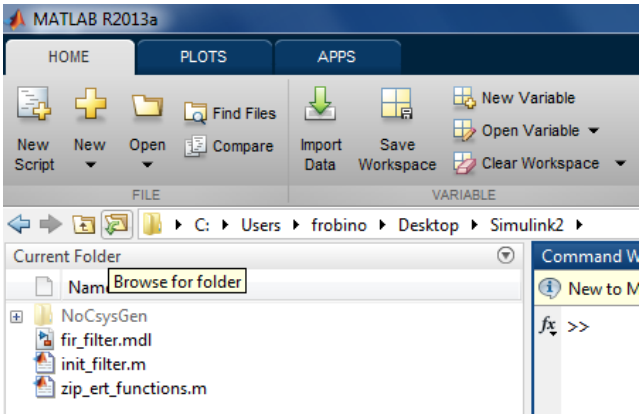


Figure 3.1.: *Browse to the right folder with MATLAB*

Open MATLAB (search for matlab using the window start button). Using the "browse for folder" button, indicated in Figure 3.1, browse to the folder named "Simulink", containing the .mdl and .m files. Double click on the .mdl file to open the Simulink model. The DSP application used in this case study is taken from a

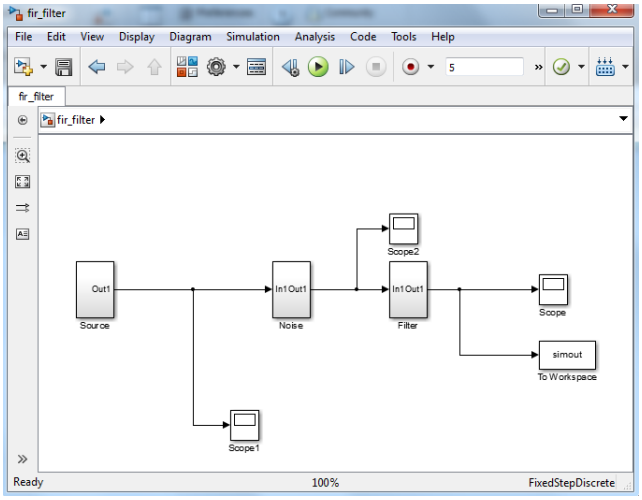


Figure 3.2.: *The Simulink system we will synthesize to a* 2×2 *NoC-based FPGA*

Simulink tutorial, and it is shown in Figure 3.2. A sinusoidal **source** block generates a sinusoidal signal. Then, some random **noise** (generated by a random source block and filtered through a digital high pass FIR filter) is added to the sinusoidal signal, creating a noisy signal. The noisy signal is then filtered by a low pass FIR **filter**, which removes the noise component.

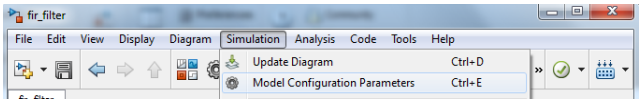


Figure 3.3.: *Access the configuration parameters in Simulink*

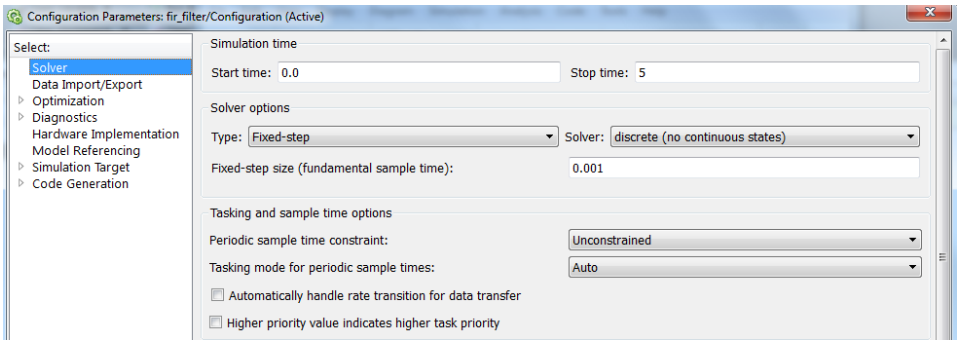


Figure 3.4.: *Solver configuration to enable automated synthesis through NSG*

In order to synthesize the Simulink model to a NoC-based platform through the NSG tool, the Simulink system should be simulated using the *discrete fixed step solver*. To check that the simulation is properly configured click on Simulation — configuration parameters, as shown in Figure 3.3. Make sure that everything is configured as in Figure 3.4, then close this form. Before running the simulation we must configure the

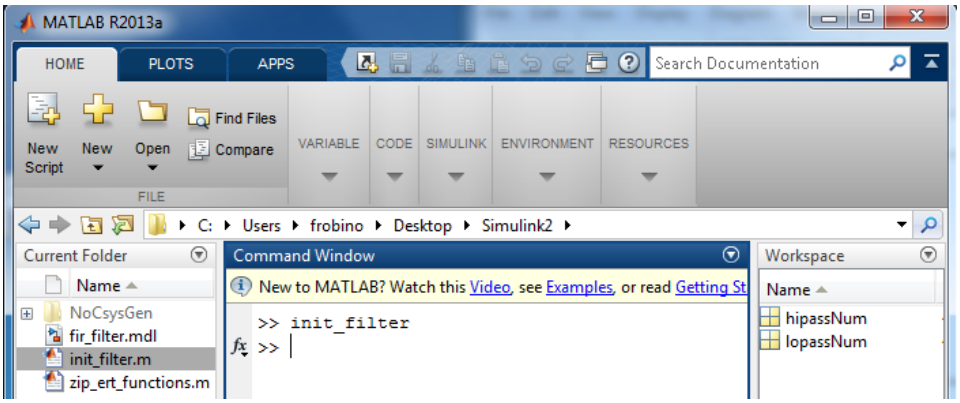
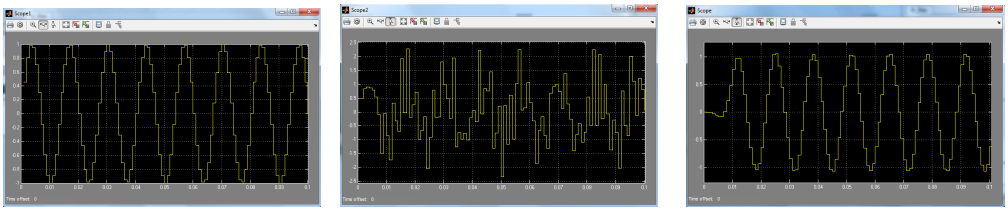


Figure 3.5.: *Configure the filters taps*

taps of the filter. This is done in MATLAB, writing in the command window (and consequently running) the `init_filter` command, as shown in Figure 3.5. Then we can come back to Simulink and run the simulation for 5 seconds, pressing on the round green button shown on the top of Figure 3.2.

Figure 3.6 shows the most significant signals of the modeled system. Figure 3.6a shows the signal produced by the sinusoidal source block. Figure 3.6b the sinusoidal signal disturbed with noise. Figure 3.6c shows the output signal of the Filter block



(a) *Source output* **(b)** *Noise output* **(c)** *Filter output*

Figure 3.6.: *Signals in the modeled system*

which is a the sinusoidal signal where the high frequency component (noise) has been filtered away. You can get the same graphs from your Simulink simulation clicking on the Scope blocks in the Simulink model.

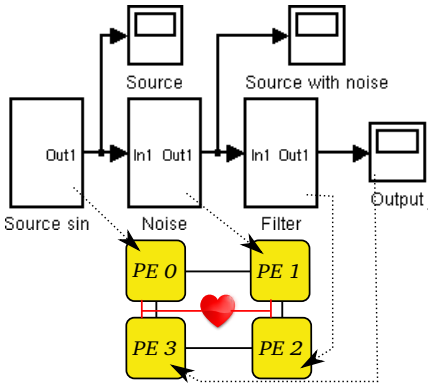


Figure 3.7.: *Map the Simulink model onto the 4 processors platform*

Once you are comfortable (and happy) with the simulation results we start the synthesis flow to NoC-based MPSoC on FPGA. **Our goal is to map the modeled system onto a** 2×2 **NoC-based MPSoC**, as shown in Figure 3.7. Each subsystem will be mapped on a different PE of the multi processor system. We recall that the first subsystem, *Source*, contains the sinusoidal source block. The second, *Noise*, contains the noise generator and the high pass FIR filter. The third, *Filter*, contains the low pass FIR filter, while the fourth is just printing out the results (and will be omitted in this tutorial).

The first step is to use the Embedded Coder to generate generates one `rt_onestep` C function for each subsystem, modeling its functionality. This can be done from the Simulink model, right clicking on each of the 3 subsystems (Source, Noise and Filter), and selecting C/C++ code — Build subsystem, as shown in Figure 3.8. When a new folder pop up, just press Build and wait for the process to be completed. Once you have done this *for each of the 3 subsystems* the tool will generate 3 folders called Source_ert_rtw, Noise_ert_rtw, and Filter_ert_rtw, containing the `rt_onestep` C function modeling the functionality of each subsystem, as shown in the right column

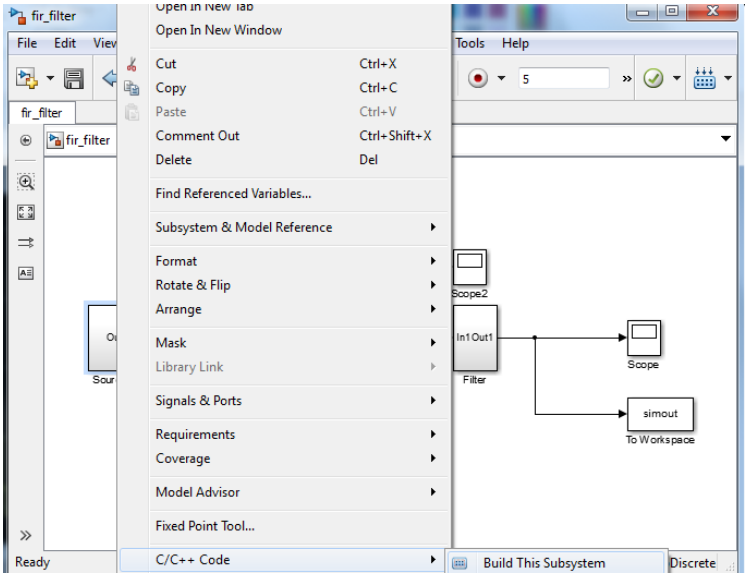


Figure 3.8.: *Using the Embedded Coder*

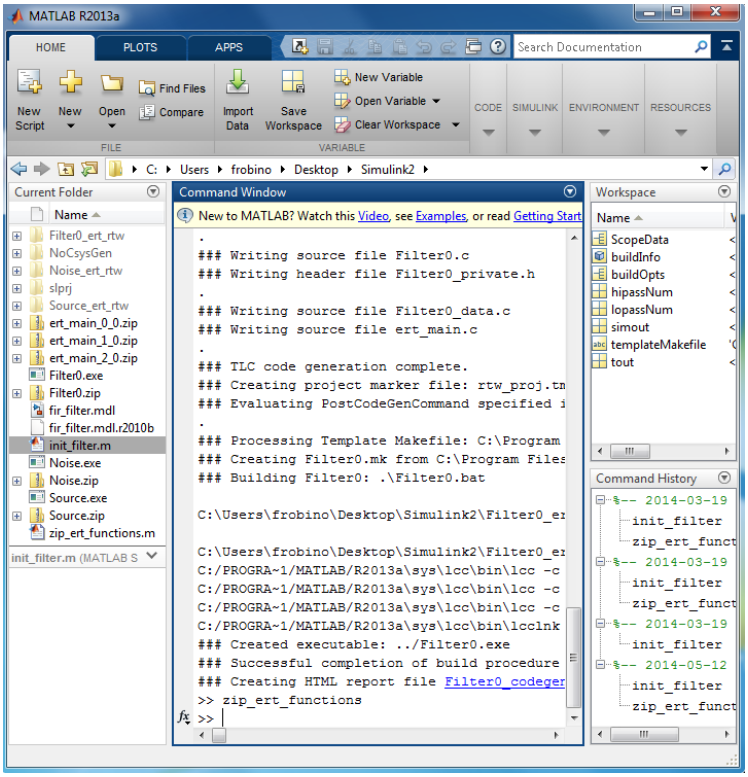


Figure 3.9.: *Embedded Coder results and zip_ert_functions command*

of Figure 3.9.

Before moving forward, use MATLAB to run the `zip_ert_functions` function, running the command in the MATLAB shell as shown in Figure 3.9. This function simply takes the _ert_rtw folders previously created and zip and rename them. For example, Source_ert_rtw is zipped and renamed to `ert_main_0_0.zip`, so that in a later stage of the flow this functionality is mapped to the PE0, i.e. Node 0 0. Check in your workspace folder that the 3 .zip files have been created, as shown in the right column in Figure 3.9. You can close Matlab and Simulink at this point and move on with the NSG tool to synthesize the system on FPGA.

3.3. Step 2

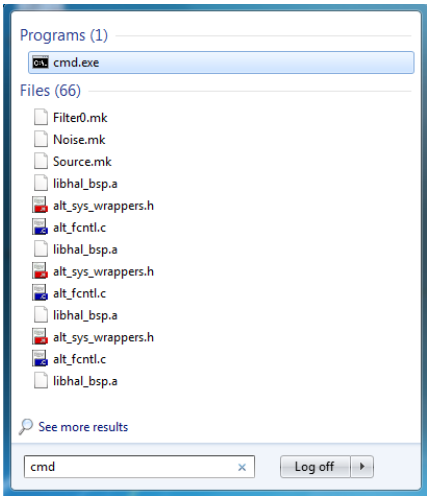


Figure 3.10.: *Open command shell in Windows*

Open a command shell (Windows start button, search for cmd), as shown in Figure 3.10, and browse through the `cd` command to the folder $Simulink\backslash NoCsysGen\backslash$. In this folder you will find an executable named `generate_noc_v2013.exe` and an intermediate representation XML file. Theoretically this XML file could have been created using the GUI of the NSG tool, as we did for the Ring tutorial. However, to shorten the tutorial time, we already provide you the XML file representing the 4 processors system where we want to map the Simulink model functionality. Open the XML file using a editor of your choice (Wordpad, etc.). Check that the `targetDirectory` parameter is pointing to the right folder. For example, if your computer username is `frobino` and your Simulink folder is on the Desktop, the right folder is:
$C:\backslash Users\backslash frobino\backslash Desktop\backslash Simulink\backslash NoCsysGen\backslash generated\_files$
The `Repository` parameter is not used in this case, so you can leave it pointing to

a non existing folder. In fact, what we want to do now, is just to generate a HW representation of the platform together with the process wrappers. Then we will provide the functions to wrap, but this is done in step 3.

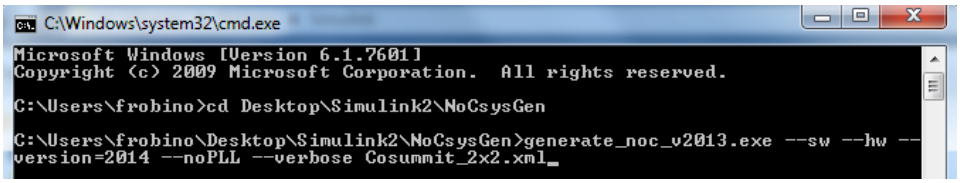


Figure 3.11.: *Run NSG in the shell*

As shown in Figure 3.11, with the shell in the *Simulink\NoCsysGen* folder, run the following command:

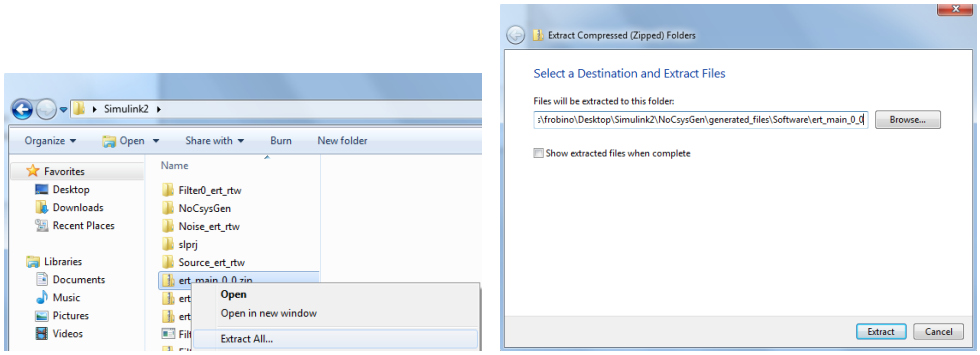
```
generate_noc_v2013.exe --sw --hw --version=2014 --noPLL --verbose Cosummit_2x2.xml
```

This will run the NSG tool and it will create all files needed to generate a 4 processors SoC on FPGA in the `target directory` (pointed by the `targetDirectory` parameter in the XML file). You can now close the command shell.

3.4. Step 3

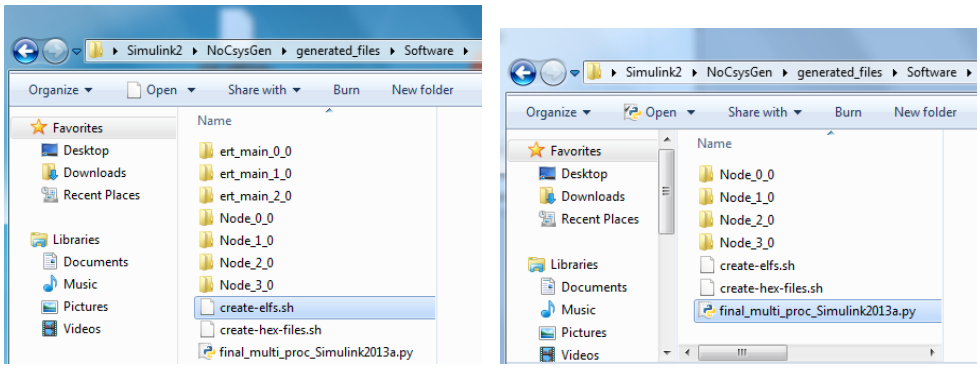
Check that all files have been generated in the *Simulink\NoCsysGen\generated_files* folder. It should look similar to the one shown in Figure 2.16, from the previous tutorial. If you browse in the *generated_files\Software* folder, you will find 4 folders where the SW part of the process wrappers are implemented. However they are not wrapping any functionality yet. What we have to do now is to include the functionality we generated earlier through Simulink and the Embedded Coder in the process wrappers.

To achieve this goal, browse to the folder containing the ert_main_X_X.zip files created at the end of Step 1. Following the commands shown in Figure 3.12, unzip the 3 files in the *generated_files\Software* folder. In addition, copy the provided `final_multi_proc_Simulink2013a.py` file in the same folder. You should reach the situation presented in Figure 3.13a. If the Software folder looks like the one in Figure 3.13a. double click on the .py file. This script will automatically include the Simulink C code in the process wrappers, leaving you with 4 folders left, as shown in Figure 3.13b.



(a) *Unzip ert_main_X_X.zip files...* **(b)** *...to the correct folder.*

Figure 3.12.: *Unzip ert_main_X_X.zip files to the right folder*



(a) *Software folder after unzipping files.* **(b)** *Software folder after running .py file.*

Figure 3.13.: *Wrap the Simulink functionality in the process wrappers*

3.5. Step 4 and 5

Now it is time to prototype the Simulink model onto FPGA. This step can be done using the Quartus tool and following the exact same instructions described in Sections 2.2.4 and compiling the SW using the instructions in Section 2.2.5. When compiling the software, be careful to use the `run_software_simulink.sh` script, instead of the previously provided `run_software.sh`. In addition, differently from the previous step, when downloading the code to each PE, it is sufficient to load only 3 PEs using the following 3 commands:

- `nios2-download -g -i 1 <path_to_file>/Node_0_0.elf` to download the compiled software to CPU_0_0;
- `nios2-download -g -i 2 <path_to_file>/Node_1_0.elf` to download the compiled software to CPU_1_0;

- `nios2-download -g -i 3 <path_to_file>/Node_2_0.elf` to download the compiled software to CPU_2_0;

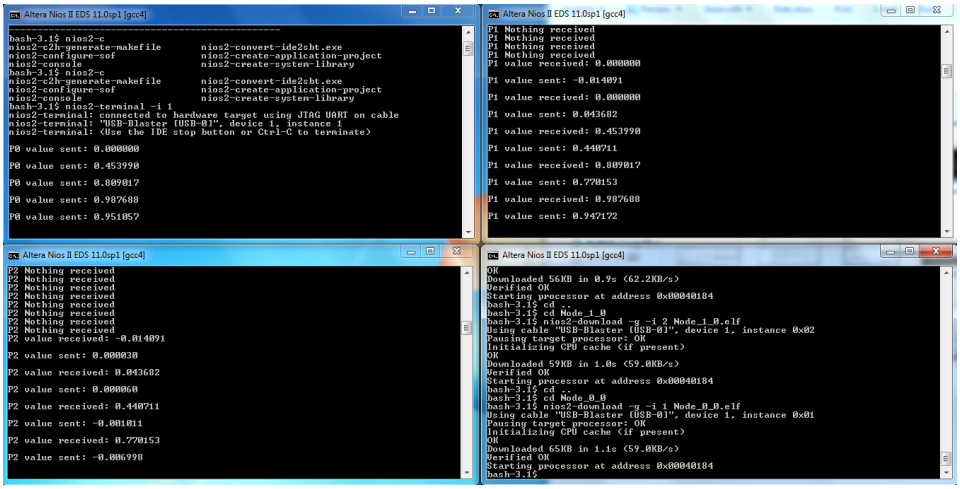


Figure 3.14.: *Run NSG in the shell*

The result of the emulation are shown in Figure 3.14. The upper left shell is Node_0_0, emulating the **Source** block of the Simulink model. As seen from the output, it generates a sinusoidal signal with values between -1 and +1. The upper right shell is Node_1_0, emulating the **Noise** block of the Simulink model. As seen from the output, the values received are the one sent from the **Source** block. The sent values represent the noisy signal, in fact the values are not representing anymore a sinusoidal signal and sometimes they exceed the values between -1 and +1. The lower left shell is Node_2_0, emulating the **Filter** block of the Simulink model. As seen from the output, the values received are the one sent from the **Noise** block. The sent values represent the filtered signal. In fact the values are representing again a sinusoidal signal and they are most of the time contained between between -1 and +1 (a bit of noise is still in the signal because of the low order of the filter).

The upper plot in Figure 3.15 shows the plotted values from Node_0_0 (Source) and Node_1_0 (Noise). The lower plot in Figure 3.15 shows a comparison between the output signal of the Simulink model (red line) and the output signal of the prototype running on 4 PEs (output of Node_3_0 ,Filter).

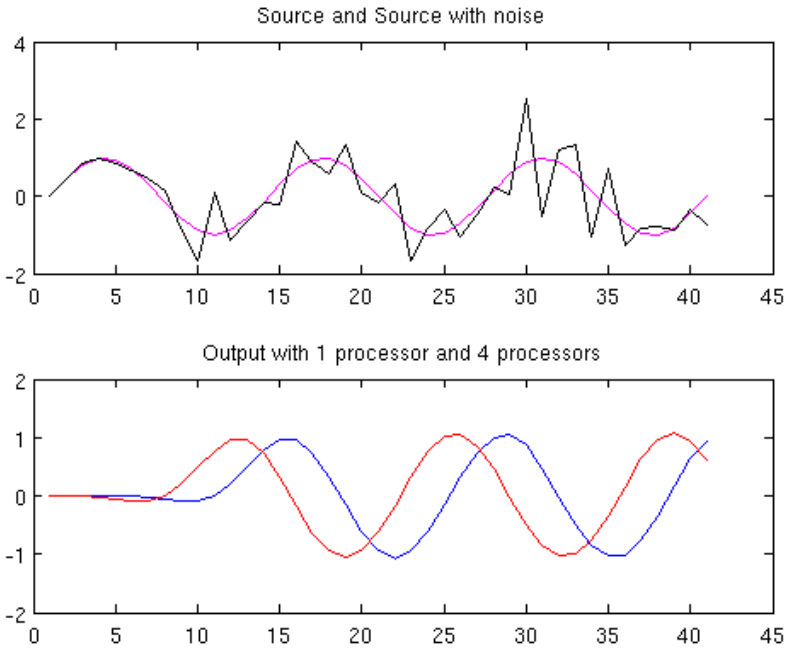


Figure 3.15.: *Simulated vs emulated results*

Appendices

A Ring Tutorial Appendix

Listing A.1: *XML description of the Ring system*

```
<?xml version="1.0" encoding="UTF-8"?>
<system name="test2" >
   <parameter name="targetDirectory" value="C:/Users/frobino/Desktop/GUITest2/generated_files2" />
   <parameter name="targetManufacturer" value="Altera" />
   <parameter name="targetManufacturerVersion" value="11.0" />
   <parameter name="boardType" value="DE2-115" />
   <hardware>
      <noc>
         <parameter name="nocType" value="Mesh" />
         <parameter name="nocKind" value="2DNoC" />
         <parameter name="nrofCols" value="2" />
         <parameter name="nrofRows" value="2" />
         <parameter name="switchType" value="Nostrum" />
         <parameter name="rniType" value="Heartbeat" />
         <parameter name="LayoutMethod" value="Floating" />
         <parameter name="FrameSize" value="64" />
         <parameter name="Heartbeat" value="1␣Hz" />
      </noc>
      <node nr="0" mem_size="8192" jtag="yes" perf_counter="no" pio="{o,1}" noc_irq="no" cpu="{nios,tiny}" />
      <node nr="1" mem_size="8192" jtag="yes" perf_counter="no" pio="{o,1}" noc_irq="no" cpu="{nios,tiny}" />
      <node nr="2" mem_size="8192" jtag="yes" perf_counter="no" pio="{o,1}" noc_irq="no" cpu="{nios,tiny}" />
      <node nr="3" mem_size="8192" jtag="yes" perf_counter="no" pio="{o,1}" noc_irq="no" cpu="{nios,tiny}" />
   </hardware>
   <software>
      <parameter name="Repository" value="C:/Users/frobino/Desktop/GUITest2/Examples/test2" />
      <process name="p0" node="0" cpu="0" moc="Synchronous" sources="{p3}" targets="{p1}" files="{p0.c}" />
      <process name="p1" node="1" cpu="0" moc="Synchronous" sources="{p0}" targets="{p2}" files="{p1.c}" />
      <process name="p2" node="3" cpu="0" moc="Synchronous" sources="{p1}" targets="{p3}" files="{p2.c}" />
      <process name="p3" node="2" cpu="0" moc="Synchronous" sources="{p2}" targets="{p0}" files="{p3.c}" />
   </software>
</system>
```

Listing A.2: *Process main P0*

```
void p0_main(void)
{
    (*p0_out0)=0;

    int input_p7_value = NOC_RNI_CHK_MSG(NOC_RNI_BASE,recv_channel_p0_from_p3);
    alt_printf("P0␣something␣received?:␣%x␣\n",input_p7_value);
    if (input_p7_value>0) // Something for me?
    {

  (*p0_out0)=(*p0_in0)+1;
  alt_printf("P0␣received?:␣%x␣\n",(*p0_in0));
  SEND(p0_out0);
    }
};
```

Listing A.3: *Process main P1*

```
void p1_main(void)
{
    (*p1_out0)=0;

    int input_p7_value = NOC_RNI_CHK_MSG(NOC_RNI_BASE,recv_channel_p1_from_p0);
    alt_printf("P1␣something␣received?:␣%x␣\n",input_p7_value);
    if (input_p7_value>0) // Something for me?
    {

      (*p1_out0)=(*p1_in0)+1;
  alt_printf("P1␣received?:␣%x␣\n",(*p1_in0));
  SEND(p1_out0);
    }
};
```

Listing A.4: *Process main P2*

```
void p2_main(void)
{
    (*p2_out0)=0;

    int input_p7_value = NOC_RNI_CHK_MSG(NOC_RNI_BASE,recv_channel_p2_from_p1);
    alt_printf("P2␣something␣received?:␣%x␣\n",input_p7_value);
    if (input_p7_value>0) // Something for me?
    {

  (*p2_out0)=(*p2_in0)+1;
  alt_printf("P2␣received?:␣%x␣\n",(*p2_in0));
  SEND(p2_out0);
    }
};
```

Listing A.5: *Process main P3*

```
void p3_main(void)
{
    (*p3_out0)=0;

    int input_p7_value = NOC_RNI_CHK_MSG(NOC_RNI_BASE,recv_channel_p3_from_p2);
    alt_printf("P3␣something␣received?:␣%x␣\n",input_p7_value);
    if (input_p7_value>0) // Something for me?
    {

  (*p3_out0)=(*p3_in0);
  alt_printf("P3␣received?:␣%x␣\n",(*p3_in0));
  SEND(p3_out0);
    }
};
```