ProNoC

User Manual

Contents

CHAPTER 1

Installation Manual for the Ubuntu Linux Environment

ProNoC Source Code

You can download the ProNoC source code from ProNoC homepage or optionally open the *terminal* and run:

```
svn co http://opencores.org/ocsvn/an-fpga-implementation-of-low-
    latency-noc-based-mpsoc/an-fpga-implementation-of-low-latency
    -noc-based-mpsoc/trunk
```

Or using git

```
git clone https://github.com/amonemi/ProNoC.git
```

Figure 1.1 shows the organization of important directories in ProNoC source code:

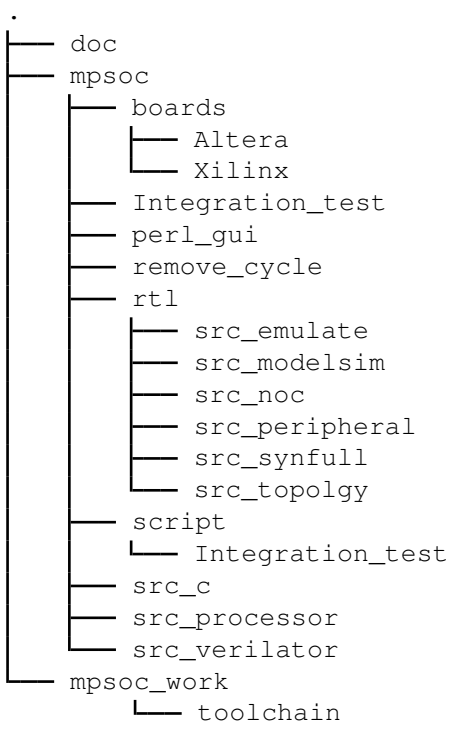


Figure 1.1: ProNoC Directory Structure.

- `doc/`: Contains the ProNoC documentation.
- `mpsoc/`: This is the main ProNoC source code directory where all projects' source codes are placed.
- `board/`: This folder contains the FPGA boards' configuration setting files. ProNoC supports both Altera and Xilinx FPGAs.
- `Integration_test/`: This folder contains Perl scripts that verify different NoC configurations using Veriltor simulator.

- `perl_gui/`: It contains ProNoC's Graphical User Interface (GUI) source codes.
- `remove_cycle/`: This directory contains a third party source code for Breaking Cycles in Noisy Hierarchies. This code removes cyclic turns in custom typologies' routing algorithms.
- `rtl/`: This folder contains all ProNoC's developed HDL codes, including the RTL code for NoC, peripheral devices, NoC simulator and NoC emulator.
- `script/`: Contains some bash scripting language source codes.
- `src_c/`: Contains source codes written in C language for converting memory files to different formats. This folder also contains codes for communicating with the JTAG interface.
- `src_processor/`: This folder contains third party open-source soft-processors' source codes.
- `mpsoc_work/`: This is a working directory for ProNoC. All generated files by ProNoC GUI will be placed here. This folder also is used as target directory for FPGA implementation and RTL simulation. The user requiters to install the soft-core CPUs' Toolchain in this folder.

Installation

Note: Please ensure that you have the necessary permissions and dependencies before proceeding with the installation.

1. Copy the downloaded folder (`trunk/`) to a location in your home directory. Ensure that there are **no spaces** in the selected path.
2. To grant execute permission, open the terminal in `trunk/mpsoc` folder and run the following command:

   ```
   sudo chmod +x -Rf ./
   ```

3. Install the required package dependencies. First, ensure that your operating system is updated by running the following commands in the terminal:

   ```
   sudo apt-get update
   sudo apt-get upgrade
   ```

   Then, in the mpsoc folder, run the following command:

   ```
   sudo sh install.sh
   ```

4. You should now be able to run the ProNoC GUI by executing the following command:

   ```
   cd mpsoc/perl_gui
   perl ./ProNoC.pl
   ```

5. If it is your first time running the ProNoC software, you will see the settings window shown in Figure 1.2.

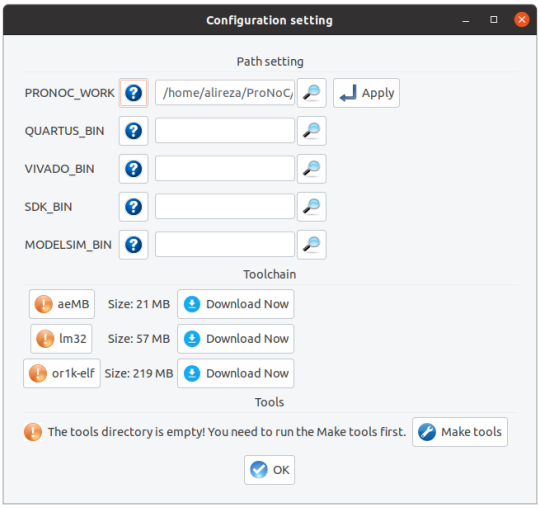


Figure 1.2: ProNoC configuration setting window snapshot.

(a) **Path Setting:** In this section, you can set the following path variables:

i. **PRONOC_WORK**: The working directory where the projects' files will be created and the toolchains are located. The default location is the `trunk/mpsoc_work` folder. Setting this variable is mandatory.

ii. **QUARTUS_BIN**: The path to the Quartus II compiler bin directory. The setting of this variable is optional. It is needed only if you will use Altera FPGAs for implementation or emulation.

iii. **VIVADO_BIN**: The path to the `Xilinx/Vivado/bin` compiler directory. Setting this variable is optional and only required if you use Xilinx FPGAs for implementation or emulation.

iv. **SDK_BIN**: The path to `Xilinx/SDK/bin` directory. Setting this variable is optional and is only needed if you use Xilinx FPGAs for implementation or emulation.

v. **MODELSIM_BIN**: The path to ModelSim/Questasim simulator bin directory. The setting of this variable is optional. You should set this variable if you have ModelSim/Questasim simulator installed on your machine and wish to auto-generate the simulation models using one of these simulators.

(b) **Toolchain:** You can download the soft-core processors' GNU toolchain by clicking on the Download Now button. Once the toolchain installation is completed successfully, you will be notified by the ✔ icon displayed in front of each toolchain name.

However, if there is a problem with downloading the files (toolchain status is still marked by ! icon), you can manually download the toolchains from the following links:

i. aeMB
ii. Lm32 or from Lm32
iii. or1k-elf for mor1k and or1200 OpenRISC CPUs.

If you have downloaded the files manually, unzip them and copy the `lm32`, `or1k-elf`, and `aemb` folders to the `trunk/mpsoc_work/toolchain` directory. Additionally, you should provide the execution permission to the GNU toolchains by running the command `sudo chmod +x -Rf ./` in the terminal inside the `mpsoc_work/toolchain` directory.

(c) **Tools:** The required tools can be compiled by clicking on the `Make tools` button. This button runs the Makefile inside `/mpsoc/src_c` directory.

You can modify these settings at any time later by navigating to `File->setting` menu in the ProNoC GUI:

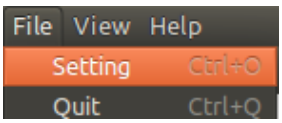


Figure 1.3: ProNoC setting menu.

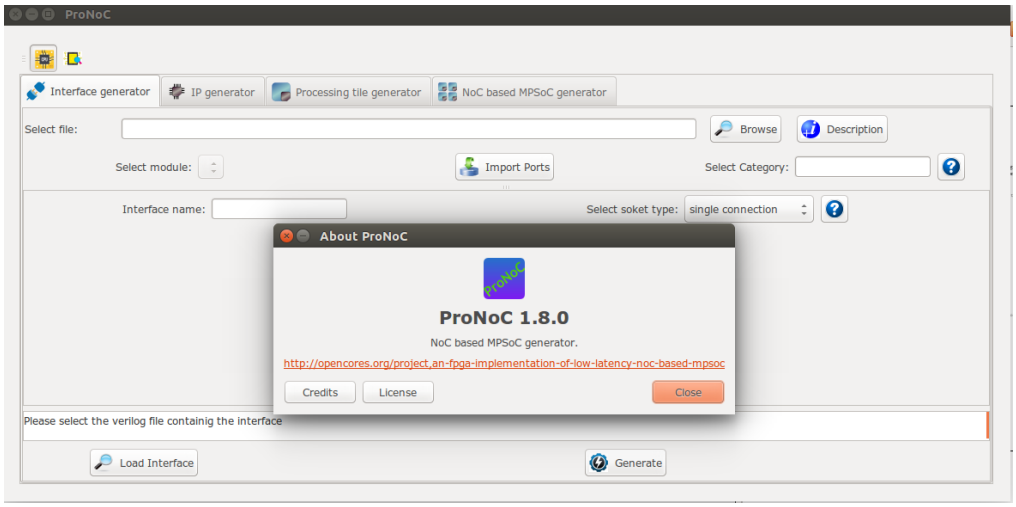


Figure 1.4: ProNoC GUI snapshot.

CHAPTER 2

Interface Generator

Introduction

The interface generator allows for the addition of new interfaces to the ProNoC. An interface refers to a port or a group of ports that are common in different IP cores used to perform specific tasks. The most frequently used interfaces in ProNoC are the shared bus (wishbone bus) master/slave, interfaces, as well as the clock (clk) and reset interfaces. Each individual interface is divided into two types: *socket* and *plug* interfaces. When connecting two different IP cores, one can have the *socket* type of an interface while the other has the *plug* type of that interface. Although it is optional to select any side of the connection either as *socket* or *plug* interface, the following differences can help in choosing the appropriate type of interface for each IP core:

- In the processing tile generator only the *plug* interfaces of an IP are displayed in the IP box. The user can select the connection interface from the list of all IP cores that have the *socket* type of that interface, as shown in Figure 2.1.

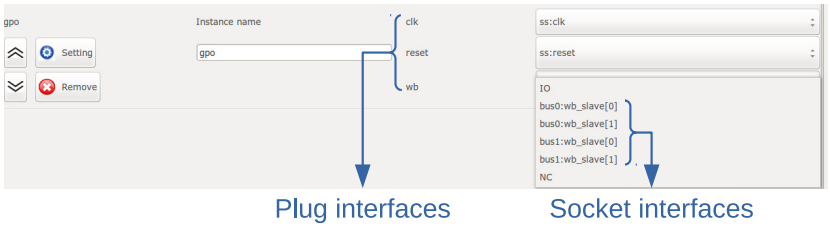


Figure 2.1: GPO IP box snapshot.

- *Socket* interfaces can be defined as single-connection or multi-connection. A socket interface can only be defined as multi-connection when it consists of only output ports exclusively. Consequently, it can be connected to multiple IPs that have the *plug* type of that interface. Examples of multi-connection *sockets* in ProNoC are the clk and reset interfaces.

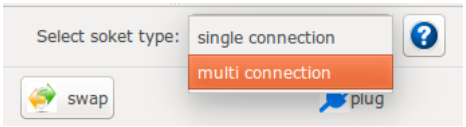
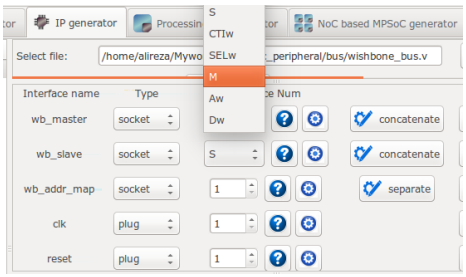


Figure 2.2: multi-connection selection snapshot.

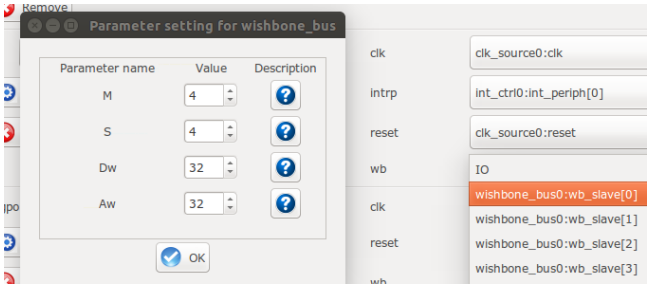
- The number of a specific *socket* interface in an IP core can be parameterized. To achieve this, the ports of interfaces with the same name must be concatenated as a single port in the IP core's Verilog file. This feature provides flexibility to the ProNoC Processing tile generator, as an IP core can now have a variable number of an interface that can be defined by the user during generation. For instance, the interfaces of the Wishbone bus and the interrupt controller have been defined as *sockets* with a parameterizable number of interfaces. The following example illustrates how the interfaces are defined in a Wishbone Bus IP core module:

Listing 2.1: bus.v

```
module wishbone_bus #(
  parameter M = 4, //number of master port
  parameter S = 4, //number of slave port
  parameter Dw = 32,// maximum data width
  parameter Aw = 32 // address width
  parameter DwS= Dw * S,
  parameter AwS= Aw * S,
    .
    .
)(
  //Slaves interface
  output [AwS-1 : 0] s_adr_o_all ,
  output [DwS-1 : 0] s_dat_o_all ,
  input [DwS-1 : 0] s_dat_i_all ,
  output [S-1 : 0] s_we_o_all ,
  output [S-1 : 0] s_cyc_o_all ,
  output [S-1 : 0] s_stb_o_all ,
  .
  .
```



(a)



(b)

Figure 2.3: (a) Select Verilog parameters **M** and **S** as the number of Wishbone bus (WB) master and slave interfaces for generating the Wishbone Bus IP core. (b) The number of WB master/slave interfaces can be defined at SoC generation time via GUI.

Generate New Interface

To add a new interface to ProNoC, click on the `browse` button and select the Verilog file that contains a module with the desired interface. If there are multiple modules within the file, you can choose the desired one from the `Select module` menu. To add ports to the interface, click on `Import Ports` button to open a pop-up window, as shown in Figure 2.4 where you can select and add the required ports to the interface.

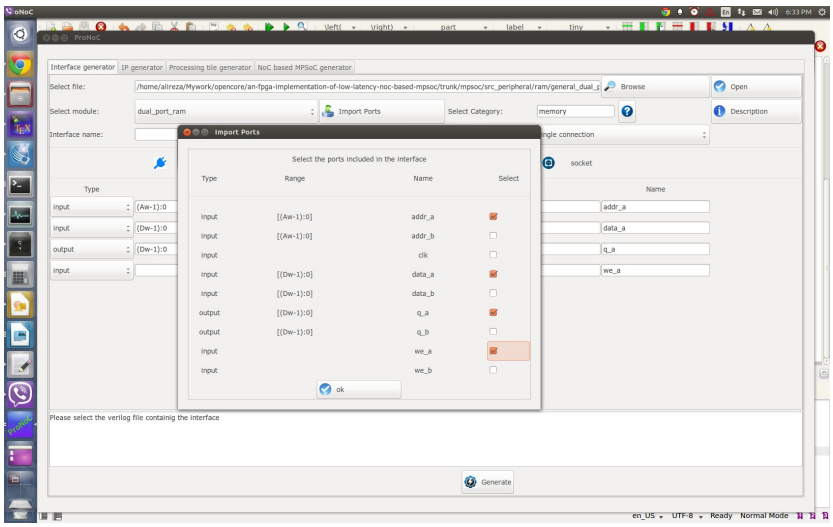


Figure 2.4: Interface generator snapshot.

Using the `swap` button, you can specify whether the selected ports belong to the *socket* or *plug* type of an interface. You only need to define one type of an interface, as the other type will be automatically defined. The width of each port can also be specified as a Verilog code parameter. It is important to note that any Verilog module using this interface must define the interface ports using the same parameter name.

The *socket* interfaces can be defined as single-connection or multi-connection. If a socket is defined as a single-connection, by connecting a new IP to the socket will automatically disconnect the previously connected *plug* to that *socket*.

Defined Interfaces

While it is optional to designate any side of an interface connection as a socket or plug when defining a new interface, once the definition is done for an IP core, all other IP cores that use that interface must follow the same convention. Therefore, it is important to understand how the defined interfaces (socket and plug) are mapped to the existing IP cores in the library. This section provides a list of defined interfaces and the IP cores that use these interfaces.

NI

This interface connects the Network-on-Chip (NoC) router and the NoC interface adapter module (NI). Figure 2.5 illustrates this interface.

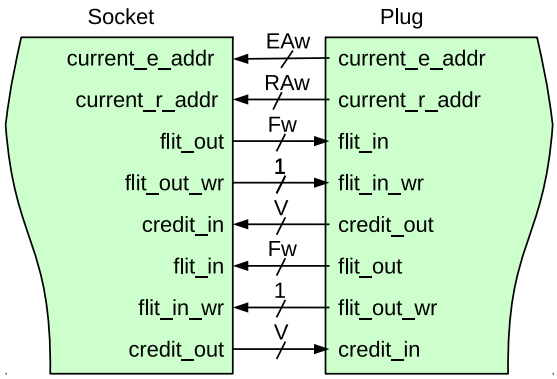


Figure 2.5: NI socket/plug interfaces.

IP cores having NI socket: ni_master, ni_slave
IP cores having NI plug: NoC

interrupt_cpu CPUs that have only a single interrupt pin must be connected to an interrupt controller module in order to handle multiple sources of interrupts. The interface between these CPUs and the interrupt controller is called interrupt_cpu.

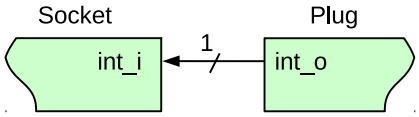


Figure 2.6: interrupt_cpu socket/plug interfaces.

IP core having interrupt_cpu socket: aeMB CPU
IP core having interrupt_cpu plug: int_ctrl (interrupt controller module)

interrupt_peripheral This interface handles interrupt connections between CPUs that have multiple interrupt pins and can be directly connected to multiple peripheral devices.

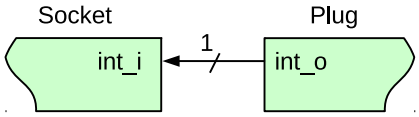


Figure 2.7: interrupt_peripheral socket/plug interfaces.

IP cores having interrupt_peripheral socket: int_ctrl, mor1kx, or1200, and lm32 CPUs.
IP cores having interrupt_peripheral plug: dma, timer, ni_master, ni_slave, ext_int (external interrupt), eth_mac100, jtag_uart.

clk The clock pin interface.

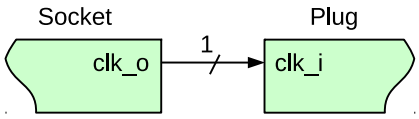


Figure 2.8: clk socket/plug interfaces.

IP core having clk socket: clk_source
IP cores having clk plug: All IP cores which have clk pin except clk_source

reset The reset pin interface.

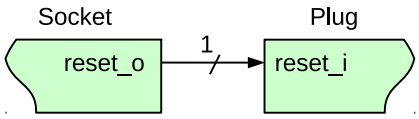


Figure 2.9: reset socket/plug interfaces.

IP core having reset socket: clk_source
IP cores having reset plug: All IP cores which have reset pin except clk_source

Enable This interface is used for the enable pin. The enable pin is responsible for disabling any active module in a processing tile, such as CPUs. The Processing Tile and NoC-based MCSoC generators automatically connect all enabled plug interfaces to each other and use them for disabling CPUs during programming mode. The enable pin for each CPU must be defined as I/O in the Processing Tile generator.

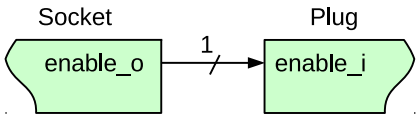


Figure 2.10: Enable socket/plug interfaces.

IP core that have enable socket: -
IP core that have enable plug: All CPUs

Wb_master This interface represents the Wishbone bus (WB) master interface. The Wb_master socket interface is mapped to the Wishbone bus module. The WB master interface of all IP cores must be mapped to the *plug* interface.

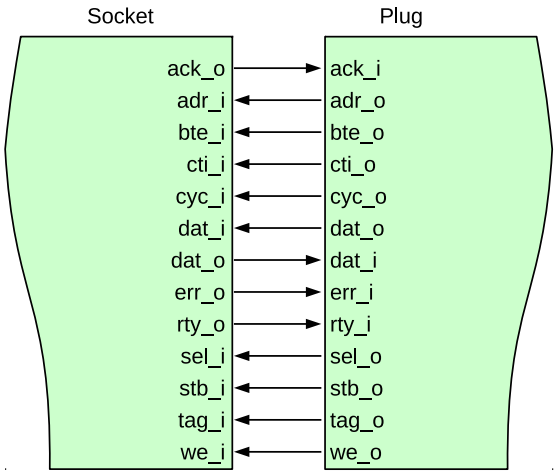


Figure 2.11: WB master socket/plug interfaces.

IP core having Wb_master socket interface: Wishbone Bus module
IP cores having Wb_master plug interface: All CPUs, ni_master, dma, eth_mac100, jtag_wb.

Wb_slave

This interface represents the Wishbone bus slave interface. The Wb_slave socket interface is mapped to the Wishbone bus module. The WB slave interface(s) of all IP cores must be mapped to the *plug* interface.
IP core having Wb_slave socket interface: Wishbone Bus module
IP core that have Wb_slave plug interface: ni_master, ni_slave, dma, eth_mac100, jtag_wb, jtag_uart, timer, gpio, gpi, gpo, single_port_ram, dual_port_ram, lcd_2x16, ext_int, int_ctrl

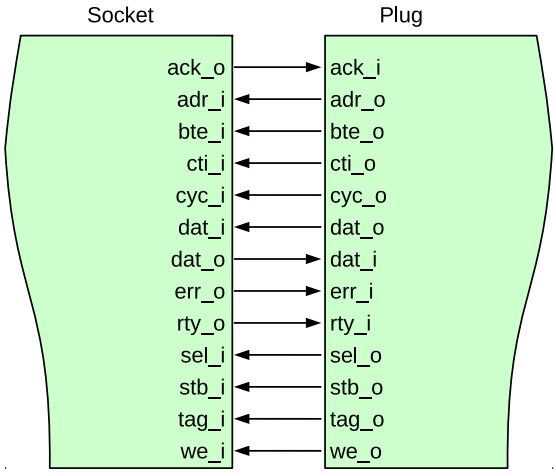


Figure 2.12: WB slave socket/plug interfaces.

CHAPTER 3

IP Generator

Introduction

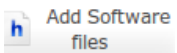
The IP generator enables the addition of new intellectual properties (IPs) to the ProNoC library. It provides a GUI interface for mapping the IP's ports to interfaces, defining how the IP parameters are collected from the user during Tile generation, and obtaining the location of the IP cores' source files.

Generate a New IP

To add a new IP to ProNoC, follow these steps:

1. Click on the `Browse` button and select the Verilog file that describes the RTL code of the IP's top-level module.
2. Select the category to which this new IP core belongs. You can either choose from the list of available categories or define a new category by typing its name in the 

. IPs belonging to the same category are listed under the same tree branch in the processing tile generator.
3. Define an `IP name` for this module. The IP name will be displayed in the IP list below its category name in the Processing tile generator.
4. If the Verilog file contains multiple Verilog modules select the top-level module in the `Select Module` field.
5. Use the `IP Description` button to add a short description of the IP. This description will be displayed when the IP is selected in the Processing Tile Generator. You can also add the IP-core documentation in PDF format here. This generates a short key for opening the IP documentation in the processing tile generator.

   Note: To ensure the portability of your ProNoC software copy the documentation files to a location within the `mpsoc` folder.
6. The 

button allows you to add necessary files and folders to the generated processing tile software directory (`[PRONOC_WORK]/SOC/[PT-name]/sw`). Clicking this button opens three notebook pages:

   - **Add existing files/folders**: Here you can add a list of files and folders that you want to copy exactly into the `mpsoc/SOC/[PT-name]/sw` folder.
   - **Add files containing variables**: On this page, you can add a list of files that contain some variables which can be replaced during processing tile generation. Variables must be written in the source file with the format `${variable_name}`. You can use any of the available variables in ProNoC as a variable name.
   - **Add to tile.h**: Define the header file for this peripheral device, including peripheral device function declarations, memory-mapped register address definitions, data type definitions, and C preprocessor commands. **Do not** include function definitions in the header file. Function definition should be

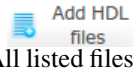
included in `Add to tile.c` section. These definitions are added to the processing tile header file during generation. Use the format `${variable_name}` for any of the available variables in ProNoC. An example of a header file is as follows:

```
#define ${IP}_REG_0 (*((volatile unsigned int *)(${BASE})))
#define ${IP}_REG_1 (*((volatile unsigned int *)(${BASE}+4)
    ))

#define ${IP}_WRITE_REG1(value) ${IP}_REG_1 = value
#define ${IP}_READ_REG1 ( ) ${IP}_REG_1

#define ${IP}_is_busy(n) ((${IP}_REG_0 >> n) & 0x1)
```

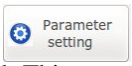
A sample generated header file by ProNoC, assuming the IP instance name is defined as `foo` by the user and the WB slave address is defined as `0X96000000` by ProNoC automatically, would be as follows:

```
/* foo */
  #define foo_REG_0 (*((volatile unsigned int *)(0X96000000)))
  #define foo_REG_1 (*((volatile unsigned int *)(0X96000000+4))
      )

  #define foo_WRITE_REG1(value) foo_REG_1 = value
  #define foo_READ_REG1 ( ) foo_REG_1

  #define foo_is_busy(n) ((foo_REG_0 >> n) & 0x1)
```

- **Add to tile.c**: Define the peripheral device's functions in this file. You can use any of available variables in ProNoC with format `${variable_name}`.

7. Click the 

button to add a list of all required HDL files for this new IP core. All listed files will be copied to the Generated Processing Tile inside the `mpsoc/SOC/[PT-name]/src_verilog` folder. If you select the `simulation only` option for any entered file/folder, they will be copied to the `mpsoc/SOC/[PT-name]/src_sim` folder instead and used only for simulation.

8. By clicking the 

button, all parameters inside the top module Verilog file are extracted. This menu allows you to add, remove, or define how to obtain parameter values from the user. Below is an example of setting the parameter **M** in the Wishbone bus.

Figure 3.1: Parameter setting snapshot.

- **Parameter name**: This is the parameter name extracted from the Verilog file.
- **Default value**: When an IP is selected for the first time in the processing tile generator, the parameters are loaded with their default values.
- **Widget type**: It defines how the parameter value should be obtained from the user when calling the IP in the Processing Tile Generator. There are four ways to define a widget type:
  - **Fixed**: The parameter is a fixed value and uses the default value. The user will not see the parameter and cannot change it in the GUI.
  - **Entry**: The parameter value is received via an `entry` widget. The user can type any value as the parameter value.
  - **Combo-box**: The parameter value can be selected from a list of predefined values.
  - **Spin-box**: The parameter is a numeric value and is selected using a `spin-box` widget.
- **Widget content**: Leaved it empty for Fixed and Entry. For Combo-box define the parameters that should be shown in the Combo-box using the following format: `"VALUE1","VALUE2",...,"VALUEn"`. For Spin-box, define it in the format `minimum,maximum,step` (e.g., 0,10,1).
- **Type**: Here, you can define how any specific IP core parameter is defined in the generated processing tile Verilog file. There are three options: `localparam`, `Parameter`, and `Don't include`. If you select `Parameter`, all processing tile parameters are also defined as parameters in the processing tile Verilog file. This allows for changing the parameters during NoC-based MPSoC generation time and enables calling the same tile in different places with different parameter values. If the parameter is a software parameter that should be used in software code as a variable define it as `Don't include`.
- **Redefine**: If checked, the defined parameter/localparam in the processing tile Verilog file will be passed to the IP core during instantiation. Uncheck it if you have only added a parameter using the parameter setting GUI that does not exist in the IP-core Verilog file.

```
parameter PARAM1= n; //redefined is on
localparam PARAM2=m; //redefined is off

ip_name #(
      // redefined parameters
      .PARAM1(PARAM1)
) instance_name(
      //ports definition starts here

);
```

- **info**: You can add a description of the parameter for the user here.

9. **Add Interface**: You can add interfaces to the IP library by double-clicking on an interface name located in the top left corner. Once the interface is added, it will appear in the interface box where you can adjust the interface settings such as the interface name, type, and the number of instances of that interface in the new IP core.

   For the Wishbone slave interface, you can select the Wishbone address by clicking on  button and configuring the following settings:

   - **Interface name**: Define a name for this interface.
   - **Address Range**: Select the address range for the Wishbone slave port. These addresses are defined in the `mpsoc/perl_gui/lib/perl/wb_addr.pm` file. You can add your own address range by modifying this file.
   - **Block address width**: This defines the maximum memory size required for this interface in bytes that is calculated as 2 raised to the power of the `block address width` (see Figure 3.2 caption for an example). The width can be set as a fixed number when the number of memory-mapped registers inside the interface is predefined. However, if the number of required registers depends on a Verilog parameter (e.g., a memory block with a parameterizable size) and you want to define it during the processing tile generation, you can make it `parametrizable` and select the corresponding parameter as the address width.

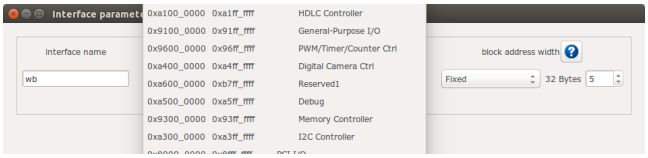


Figure 3.2: Snapshot of Wishbone slave address settings. In this example, the size of the memory-mapped registers is $2^5 = 32$ bytes. For a Wishbone bus with 32-bit width, it is equivalent to $32/4 = 8$ individual registers. If you have a parameterizable number (e.g., **M**) to indicate the memory-mapped register width in words in your IP module's Verilog file, you need to add another parameter such as **N=M+2** in the `parameter setting` window. Select its type as `Don't include` to use it as the address width parameter in bytes.

   For socket interfaces, you have the option to define the interface number as parameter by selecting the  `concatenate` condition or as a fixed number by selecting the  `separate` condition. Refer to the socket interface specification for more information.

10. After adding the interfaces, you need to map the top module ports to the interfaces ports. For each top-level module port, you need to select the corresponding interface name and interface port. Figure 3.3 illustrates a snapshot of interface mapping for the Wishbone Bus module.

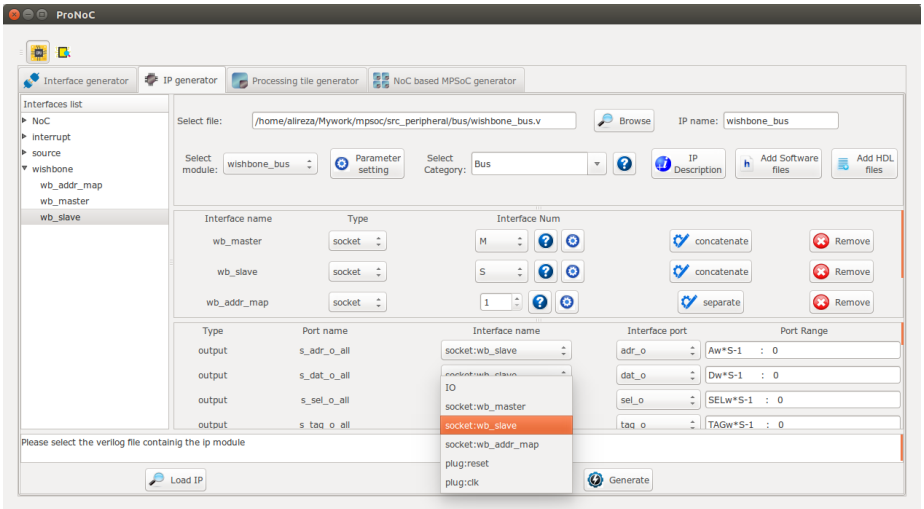


Figure 3.3: Wishbone Bus module interface mapping snapshot.

11. Finally, by clicking on Generate you can generate the IP. You can also modify existing IPs by using the Load IP button.

Refer to the Add Custom IP Tutorial for an example of adding a custom IP core to the ProNoC library.

List of Available Variables in ProNoC

- `${[parameter_name]}`: The IP core parameter value. The actual value is defined by the user when calling the IP core during processing tile generation. The parameter needs to be added in the GUI parameter using the `parameter setting` button.
- `${CORE_ID}`: Each Wishbone bus-based processing tile has a unique `CORE_ID` that represents its location in the NoC topology.

  If the generated tile is used as a top-level module, `CORE_ID` will take the default value of zero.
- `${IP}`: The instance name of the peripheral device, which is defined by the user when calling the IP core using the Processing tile generator.
- `${CORE}`: The name of the peripheral device IP core.
- `${BASE}`: The Wishbone base address(es) that will be added during processing tile generation to the processing tile's C header file (`mpsoc/SOC/[PT-name]/sw/[Tile_name].h`). If there are multiple slave Wishbone buses in the IP core, the variables are defined as `${BASE0}`, `${BASE1}`, and so on.

List of Available IP Cores in ProNoC

This section provides a brief description of the available IP core modules in the ProNoC library. Most IP cores developed with the ProNoC software come with a separate documentation PDF file. These files can be accessed by clicking on the names of the IP core modules in the following section. For other IP cores adopted from the OpenCores website, the project homepage URL address is linked to the IP core name.

Bus

- **Wishbone_bus (WB):** This is an open-source hardware computer bus released by OpenCores. ProNoC's WB is fully parameterizable in terms of the number of master/slave interfaces and data/address width.

Communication

- **ProNoC-jtag_uart**: A JTAG-based Universal Asynchronous Receiver-Transmitter (UART) module with a Wishbone bus interface. Communication with the host PC is handled using the Altera Virtual JTAG Tab or Xilinx BSCANE2 Tab. ProNoC has a built-in GUI with the ability to monitor multiple UART terminals simultaneously (See ProNoC_UART GUI).
- **ProNoC_jtag_wb**: The Altera JTAG to Wishbone bus interface. This module allows reading/writing data to the IP cores connected to the Wishbone bus (e.g., memory cores). For Altera FPGAs, communication with the host PC is done using `mpsoc/src_c/jtag/jtag_libusb` via USB Blaster I and `mpsoc/src_c/jtag/jtag_quartus_stp` via USB Blaster II. For Xilinx FPGAs, it is done using `mpsoc/src_c/jtag/jtag_xilinx_xsct`. Communication with the FPGA board can also be done using a GUI interface called Run-time JTAG Debugger.
- **altera_jtag_uart**: The Altera (Qsys) JTAG UART core with a Wishbone bus interface.
- **Etmach_100:** The Ethernet MAC (Media Access Control) 10/100 Mbps. This IP core is adopted from OpenCores/ethmac.

DMA

- **dma**: A Wishbone bus round-robin-based multi-channel DMA (no byte enable is supported yet). The DMA supports burst data transactions.

Display

- **lcd_2×16**: A 2×16 Character Alphabet Liquid Crystal Display (LCD) driver module.

GPIO

- **gpi**: General purpose Wishbone bus-based input port.
- **gpo**: General purpose Wishbone bus-based output port.
- **gpio**: General purpose Wishbone bus-based bidirectional port.

Interrupt

- **ext_int**: External interrupt module.
- **int_ctrl**: Interrupt controller. CPUs that have only one single interrupt pin (e.g., aeMB) must be connected to an interrupt controller module to allow the combination of several sources of interrupts.

NI

- **ni_master**: A Wishbone bus (WB)-based interface for the network-on-chip (ProNoC) router. This module has two WB master interfaces, one for sending and another for receiving data packets.
- **ni_slave**: An extension of the **ni_master** module connected to two input and output buffers. There are three WB slave interfaces in this module: one for writing to the output buffer, one for reading the input buffer, and one for controlling the NI.

Processor

- **Or1200:** OR1200 is the original implementation of the OpenRISC 1000 architecture. Its source code has been adopted from GitHub at openrisc/or1200.
- **aeMB:** The EDK3.2 compatible Microblaze core. This IP core is adopted from OpenCores/aemb.
- **lm32:** LatticeMico32 is a soft processor originally developed by Lattice Semiconductor. The source code of this IP core is adopted from github/soc-lm32.
- **mor1kx**: The mor1kx is a replacement for the original or1200 processor. The source code is adopted from GitHub at openrisc/mor1kx.

RAM

- **single_port_ram**: A Wishbone bus-based single port Random Access Memory (RAM).
- **dual_port_ram**: A Wishbone bus-based dual port RAM.

Source

- **clk_source**: This module provides the clk and reset (socket) interfaces for all other IPs. It also synchronizes the reset signal.

Timer

- **timer**: A simple, general purpose, Wishbone bus-based, 32-bit timer.

CHAPTER 4

Processing Tile Generator

A Processing Tile (PT) is a set of several IPs (processors and peripheral devices) connected via interfaces. Figure 4.1 illustrates a snapshot of the PT generator. The PT generator facilitates the RTL code generation of a custom PT by providing the following features:

1. Allows the addition of any arbitrary number of IP cores to the PT.
2. Provides a simple GUI for connecting IP cores.
3. Provides a GUI for setting IP core parameters.
4. Auto-generates the Wishbone Bus slave interface addresses.
5. PT functional block diagram viewer.
6. PT RTL code generator.
7. Comes with an in-built text editor for software development and compilation.
8. Facilitate RTL code synthesizing using one of the Verilator, Modelsim, Vivado or QuartusII compilers.

For more information about the PT generator, please refer to the Processing Tile Generator Tutorial.

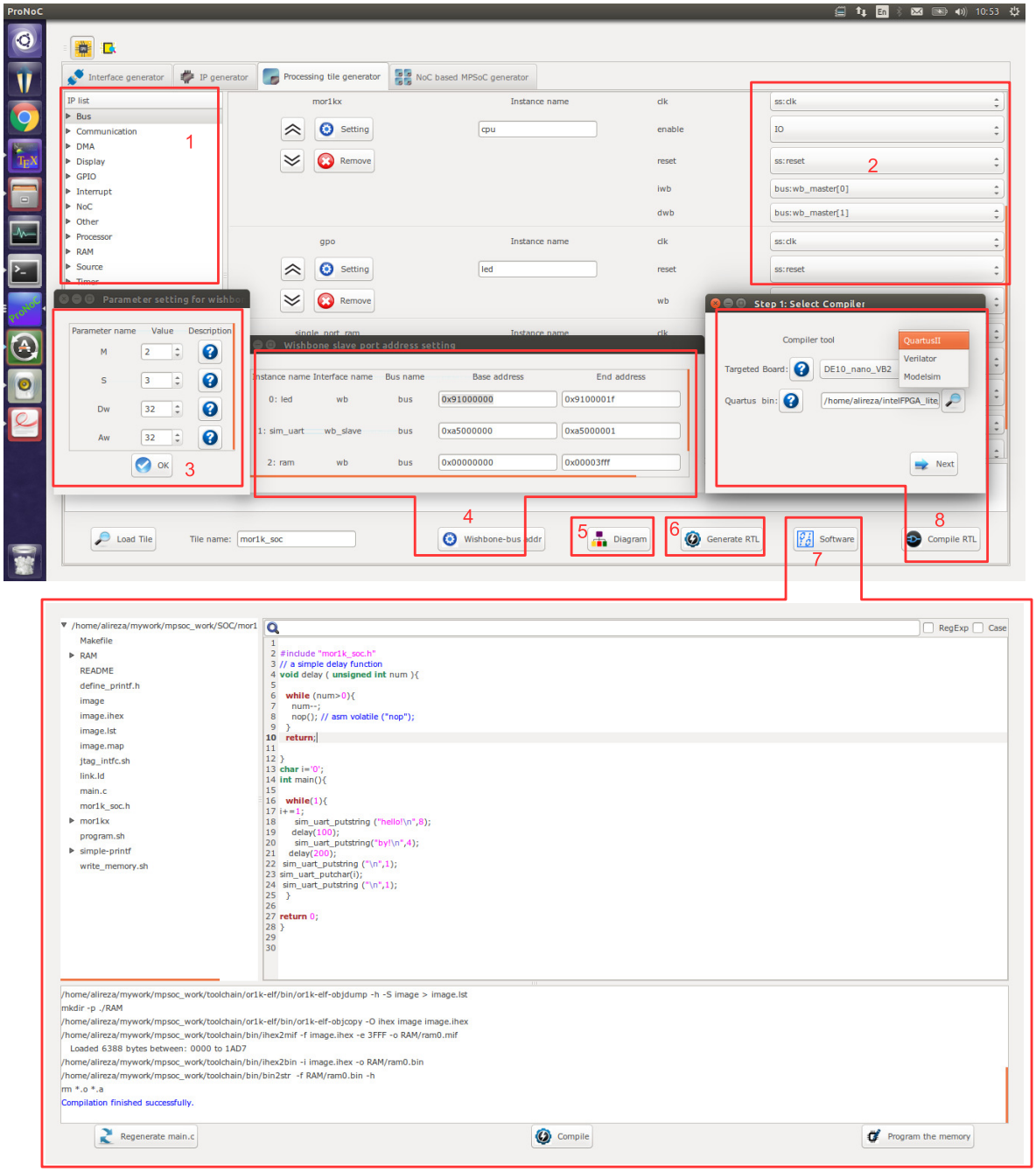


Figure 4.1: PT generator snapshot.

CHAPTER 5

Processing Tile Generator Hello World Tutorial

Summary

This tutorial teaches how to develop a shared bus (Wishbone bus) based system-on-chip (SoC) and a simple software implementation using the **ProNoC Processing Tile Generator**. The desired SoC will be generated by connecting open-source IP cores on an Altera or Xilinx FPGA board.

System Requirements

You will need an Altera or Xilinx FPGA development board and a computer system running Linux OS with:

1. ProNoC GUI software and its dependency packages installed.
2. Installed/pre-built GNU toolchain of the Mor1kx soft-core processor.
3. Installed Quartus II (Web-edition or full) compiler in case of having an Altera FPGA board or Vivado Design Suite compiler in case of using a Xilinx FPGA board.

For more information about the GNU toolchain installation, please refer to the Installation Manual for Ubuntu. If your FPGA board is not included in the ProNoC FPGA board list, please follow the instructions given in Adding a New Altera/Xilinx FPGA Board to ProNoC to add your board to the ProNoC library.

Objectives

1. Design a Wishbone bus-based system-on-chip hardware architecture using ProNoC Electronic Design Automation (EDA) software.
2. Develop a simple software application running on the generated SoC.
3. Interact with on-board memory units using the JTAG to Wishbone interface module.

Desired SoC

Schematic

Figure 5.1 illustrates the desired hardware architecture in this tutorial. This architecture consists of:

1. Four LEDs connected to a 4-bit general-purpose output (GPO).
2. A 32-bit timer.
3. A mor1kx processor (you can use any other available processors).
4. A single-port RAM.
5. A JTAG UART.
6. A Wishbone Bus.
7. A Clock source (not shown in Figure 5.1).

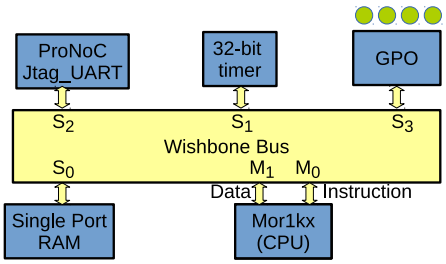


Figure 5.1: The schematic of the desired SoC in this tutorial.

Application Software

The aim of this tutorial is to design a simple SoC for running ”Hello world” and ”blinking LED” programs on the desired SoC.

Create New SoC Using ProNoC Processing Tile Generator

Open terminal and navigate to the `mpsoc/perl_gui` directory. Run ProNoC GUI application using the following command:

```
./ProNoC.pl
```

This will open the ProNoC GUI interface as shown in Figure 5.2.

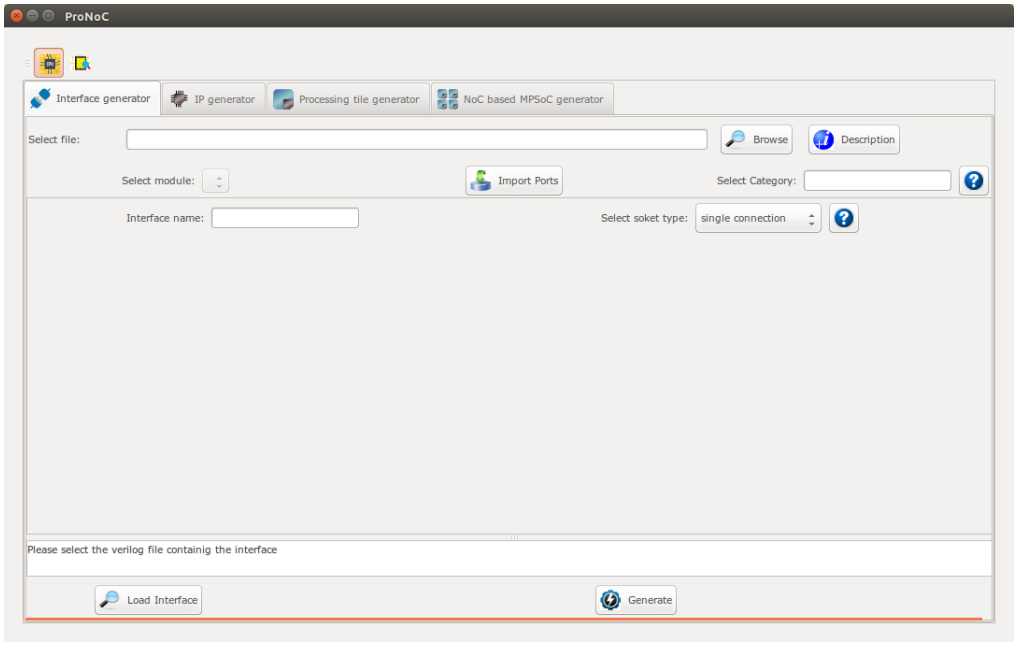


Figure 5.2: ProNoC GUI first page snapshot.

Select the  `Processing Tile Generator` tab as shown in Figure 5.3.

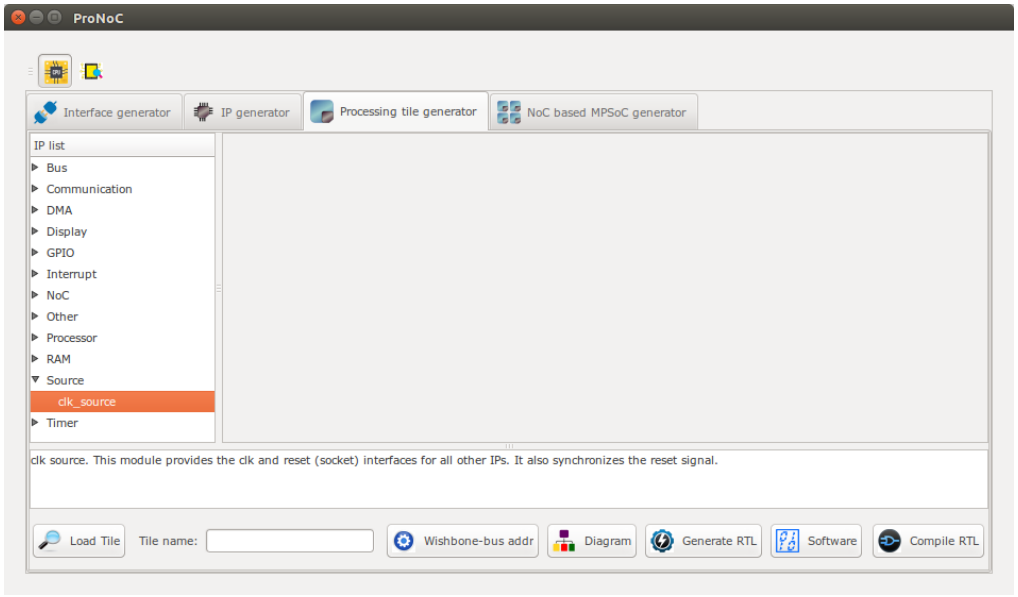


Figure 5.3: ProNoC New Processing Tile generator snapshot.

In the left Tree-View window, you can see the list of all available IP categories. Clicking on each category will expand the associated list of IP cores. Each IP core can be added to the GUI by double-clicking on its name. The added IP core has three setting columns:

(a) In the first column, you can shift the IP core box position up/down in the GUI interface, remove the IP core, or set its parameters (if any).

(b) In the second column, you can rename the IP core instance name.

(c) The third column shows all *(Plug)* interfaces of this module. Here you can connect each plug to one appropriate *(socket)* interface. Each interface is categorized into two types of plug and socket. See the Interface Generator chapter for more information about interfaces. You can also export the interface as the SoC's input/output (IO) ports here.

Let's start by adding the `clk_source`:

Add clk source

This module provides the clk and reset interfaces for all other IPs. It also synchronizes the reset signal.

1. Click on the `Source` category, then double-click on `clk_source`.
2. Rename the `clk_source` instance name as `source`. Leave the interfaces as `IO`.

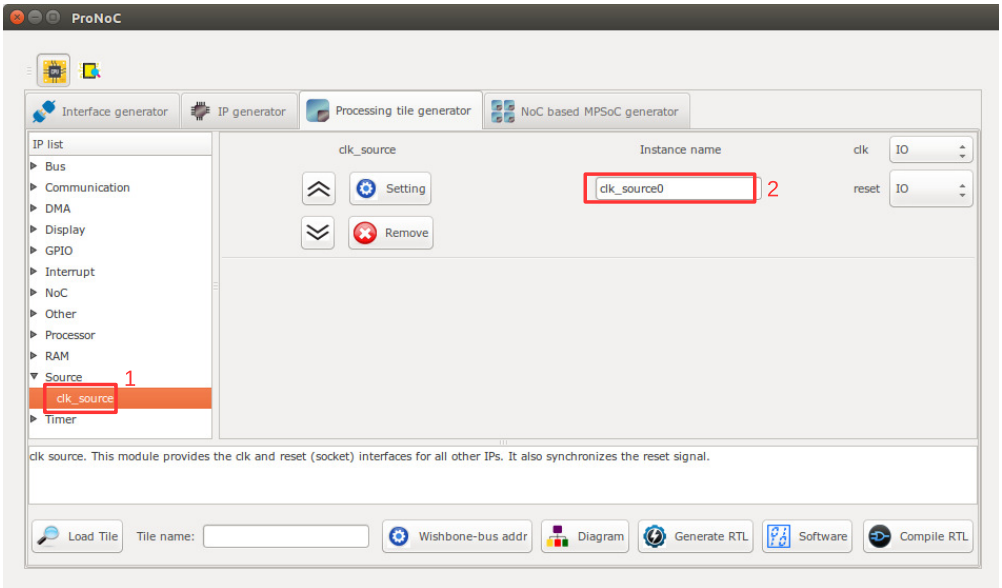


Figure 5.4: Adding the clock source.

Add Wishbone Bus

1. Click on the `Bus` category and double-click on `Wishbone_bus`.
2. In the parameter setting, set `M` (master interfaces number) as 2 and `S` (slave interfaces number) as 4. These values are obtained from Figure 5.1. You can change them later if you want to add/remove any IPs.
3. Rename the instance name as `bus`.
4. Connect the clock and source interfaces to the `clk_source` module.

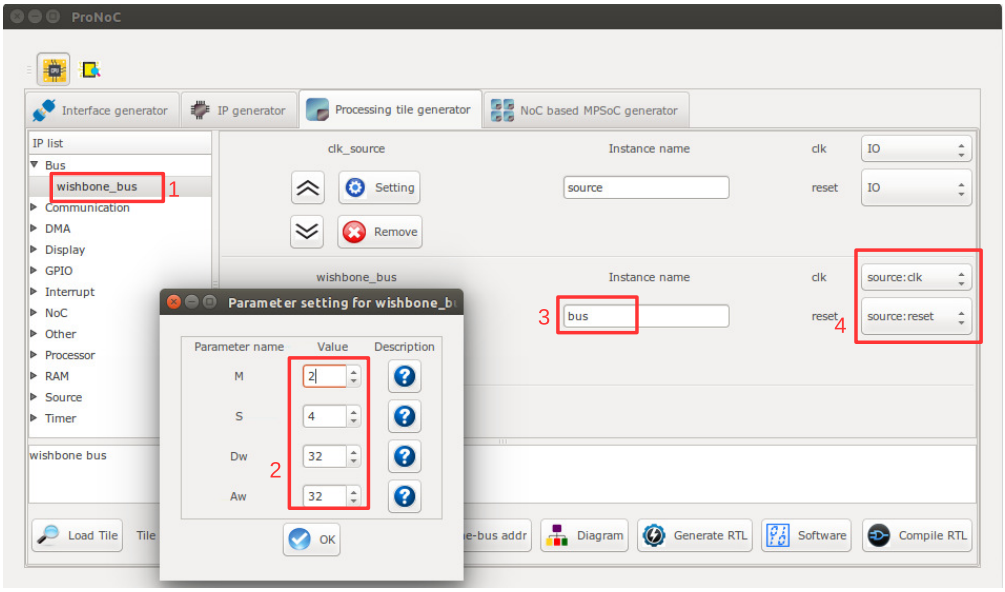


Figure 5.5: Adding the Wishbone bus.

Add the rest of IPs

Add the rest of the IP cores according to Table 5.1.

- Note that the parameters which are needed to be assigned differently for Altera & Xilinx FPGA boards are footnoted.
- Note that the socket interface has the following format:
  `connection-IP-instance-name : interface-name [interface number]`.
  Hence, `bus:wb_slave[0]` means that the `wb` interface of `GPO` IP is connected to the `bus` via the zeroth `wb` interface. Note that you can optionally connect it to any of the other `wb` interfaces numbers as WB has a round-robin arbitration scheduler.
- In case of using another processor, note that some softcore processors such as aeMB may need an interrupt controller. Table 5.2 lists the IP core setting for this CPU.
- It is not necessary to connect Wishbone bus Master/Slave interfaces according to the given port number in Table 5.1and 5.2. Any arbitrary order will work.

Table 5.1: IP core list and setting for Mor1kx SoC.

Category	IP name	Parameter			Instance name	Interface connection		
Source	clk_source	FPGA_VENDOR	→	"ALTERA"[1]	source	clk reset	→ →	IO IO
Bus	wishbone_bus	M S Dw Aw	→ → → →	2 4 32 32	bus	clk reset	→ →	source:clk source:reset
Processor	Mor1kx	OPTION_DCACHE_SNOOP FEATURE_INSTRUCTIONCACHE FEATURE_DATACACHE FEATURE_IMMU FEATURE_DMMU	→ → → → →	"NONE" "ENABLED" "ENABLED" "ENABLED" "ENABLED"	cpu	clk reset snoop iwb dwb enable	→ → → → → →	source:clk source:reset bus:snoop bus:wb_master[0] bus:wb_master[1] IO
RAM	single_port_ram	Dw Aw BYTE_WR_EN FPGA_VENDOR JTAG_CONNECT JTAG_INDEX BURST_MODE MEM_CONTENT_ FILE_NAME INITIAL_EN JTAG_CHAIN	→ → → → → → → → → →	32 14 "YES" "ALTERA"[1] "ALTERA_JTAG_WB"[2] CORE_ID "ENABLED" "ram0" "YES" 4	ram	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[0]
Timer	timer	PRESCALE_WIDTH	→	8	timer	clk reset wb intrp	→ → → →	source:clk source:reset bus:wb_slave[1] cpu:interrupt_peripheral[0]
Communication	ProNoC_jtag_uart	BUFF_Aw JTAG_INDEX JTAG_CHAIN JTAG_CONNECT INCLUDE_SIM_PRINTF	→ → → → →	4 126-CORE_ID 3 "ALTERA_JTAG_WB" [2] SIMPLE_PRINTF	uart	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[2]
GPIO	gpo	PORT_WIDTH	→	4	led	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[3]

[1] → "XILINX" For Xilinx FPGA
[2] → "XILINX_JTAG_WB" For Xilinx FPGA

Table 5.2: IP core list and setting for aeMB SoC.

Category	IP name	Parameter			Instance name	Interface connection		
Source	clk_source	FPGA_VENDOR	→	"ALTERA"[3]	source	clk reset	→ →	IO IO
Bus	wishbone_bus	M S Dw Aw	→ → → →	2 5 32 32	bus	clk reset	→ →	source:clk source:reset
Processor	aeMB	STACK_SIZE HEAP_SIZE	→ →	0X400 0X400	aeMB	clk reset iwb dwb enable	→ → → → →	source:clk source:reset bus:wb_master[0] bus:wb_master[1] IO
RAM	single_port_ram	Dw Aw BYTE_WR_EN FPGA_VENDOR JTAG_CONNECT JTAG_INDEX BURST_MODE MEM_CONTENT_ FILE_NAME INITIAL_EN JTAG_CHAIN	→ → → → → → → → → →	32 14 "YES" "ALTERA" [3] "ALTERA_JTAG_WB"[4] CORE_ID "ENABLED" "ram0" "YES" 4	ram	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[0]
Interrupt	int_ctrl	INT_NUM	→	1	int_ctrl	clk reset interrupt_cpu wb	→ → → →	source:clk source:reset aeMB:interrupt_cpu bus:wb_slave[4]
Timer	timer	PRESCALE_WIDTH	→	8	timer	clk reset wb intrp	→ → → →	source:clk source:reset bus:wb_slave[1] in_ctrl:int_periph[0]
Communication	ProNoC_jtag_uart	BUFF_Aw JTAG_INDEX JTAG_CHAIN JTAG_CONNECT INCLUDE_SIM_PRINTF	→ → → → →	4 126-CORE_ID 3 "ALTERA_JTAG_WB" [4] SIMPLE_PRINTF	uart	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[2]
GPIO	gpo	PORT_WIDTH	→	4	led	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[3]

Check Wishbone Bus Addresses

After adding all the required IP cores, you can check the auto-assigned Wishbone bus addresses by clicking on the `Wishbone-bus addr` button. Note that the assigned addresses are also modifiable.

[3] → "XILINX" For Xilinx FPGA
[4] → "XILINX_JTAG_WB" For Xilinx FPGA

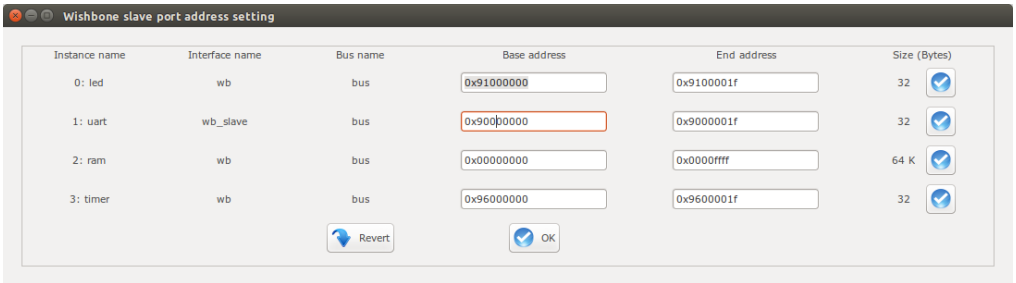


Figure 5.6: Wishbone bus addresses of the tutorial SoC.

These addresses are automatically set based on the IP cores library setting, inserted parameters, and numbers of repeating same IP cores in the system. However, you are free to adjust them to new values as long as there is no conflict in inserted addresses.

View SoC Functional Block Diagra

Click on the Diagram button to observe the SoC functional block diagram.

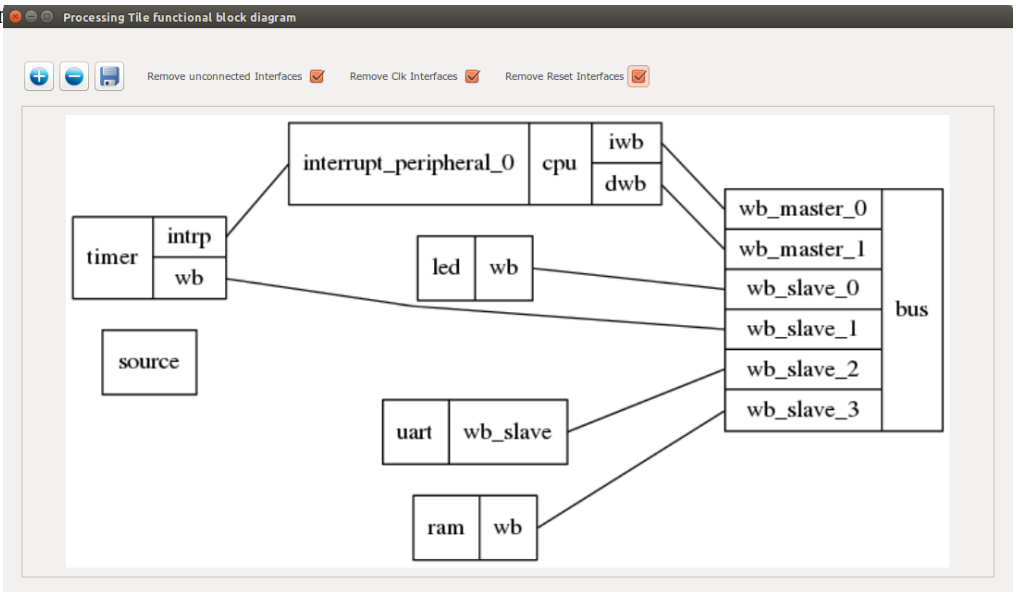


Figure 5.7: The tutorial SoC diagram.

CLK Setting

In case that the SoC is desired to be the top-level module in FPGA implementation (e.g., in this example), you may need to generate the SoC clock signal from the FPGA incoming reference clock.

To do that, click on the CLK Setting button. It will open a new window where you can connect your SoC top module to some clock sources. As an example, Xilinx Kintex-7 FPGA KC705 has a differential reference clock. The differential FPGA input

clocks are first needed to be converted into a single clock using the IBUFGDS module. The output of IBUFGDS (diff:clk) is 200 MHz, which is too high for the desired SoC to meet the timing constraints. This clock can be divided by half using a PLL. To do so, set the PLL multiplication (`CLKFBOUT_MULT`) and division factor (`CLKOUT0_DEVIDE`) to 9 and 18, respectively.

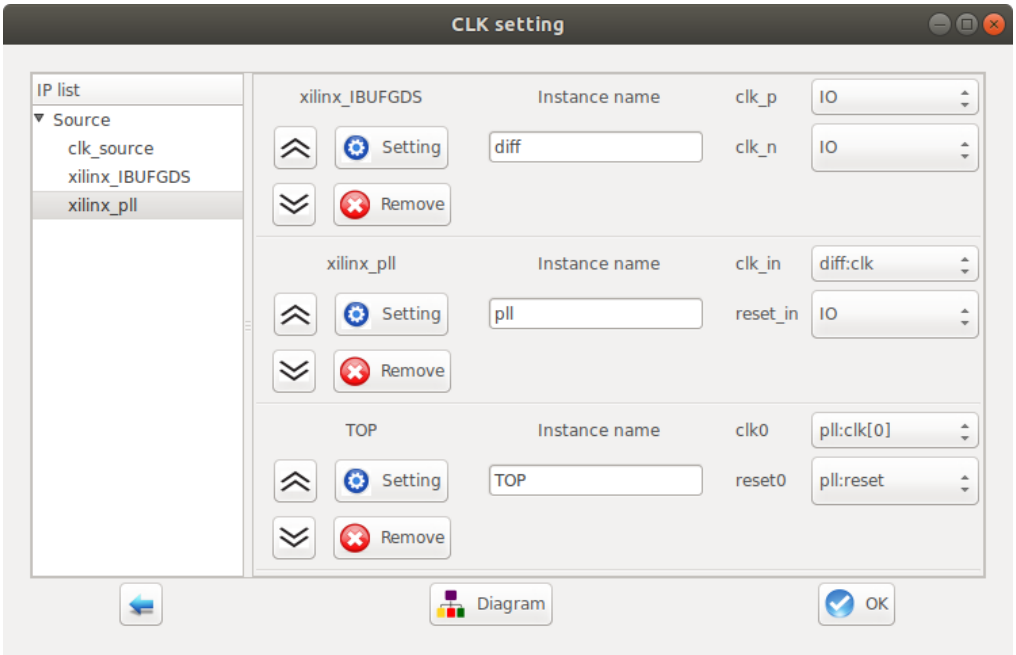


Figure 5.8: Example of clock setting for Xilinx Kintex-7 FPGA KC705 Evaluation Kit.

- Note that if you directly want to connect the FPGA reference clock to the SoC, you can omit this configuration. The TOP module clk and reset signals are left as output by default.
- If the processing tile is planned to be used in an internal module inside an MP-SoC, the clock setting can be ignored.

Generate SoC RTL Code

1. Set the Tile name as `tutorial`.
2. Click on the `Generate RTL` button.

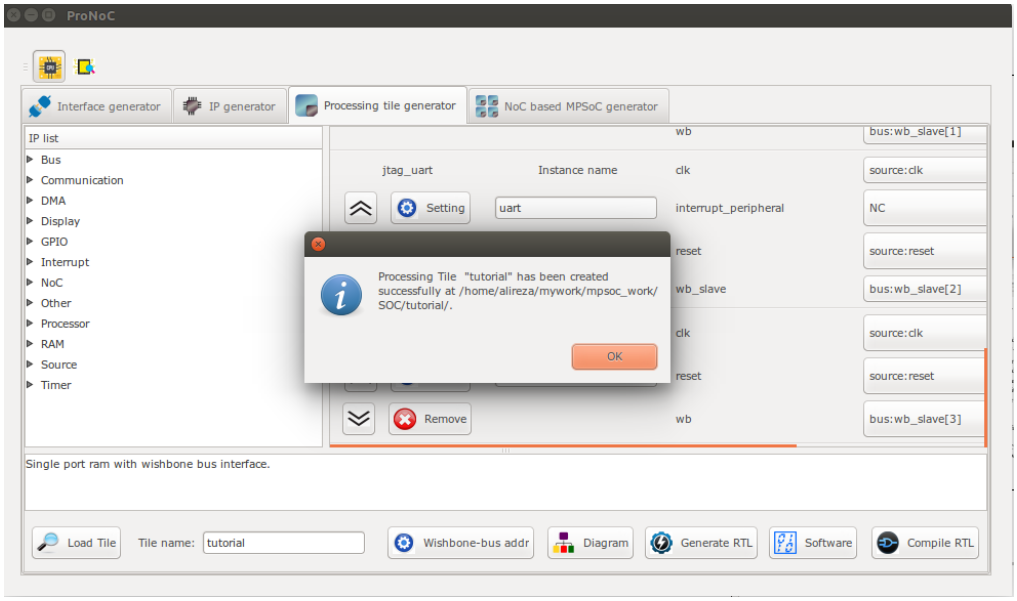


Figure 5.9: Generating the tutorial SoC.

If the generation is successful, you must have two new folders in your `mpsoc/soc/tutorial` path:

- `sw`: This folder contains the required software files, including the programming header files, in-system memory editing files, and Makefile.
  - `tutorial.h`: The SoC header file containing peripheral device functions' declaration, memory-mapped register address definition, definitions of data types, and C preprocessor commands (some IPs may have additional header files).
  - `tutorial.c`: Contains all peripheral device function definitions.
  - `README`: This file contains SoC parameters, IP connections, and wishbone bus addresses. This file also explains how to work with the `Jtag_wb` IP core.
  - `program.sh`: A sample bash file that can be used for programming the SoC RAMs at run time using the JTAG interface.
- `src_verilog`: This folder contains three Verilog files and a folder:
  - `tutorial.v`: The generated SoC RTL code. This file contains all IP instances and connections.
  - `tutorial_top.v`: This file contains the tutorial SoC module instance connected to JTAG controllers and clock converter modules.
  - `Top.v`: Contains the highest top module where the SoC ports are connected to the target FPGA pins.

– `lib`: This folder contains all IP core HDL files.

Software Development

1. Click on the `Software` button to open the software development window.

2. In the left Tree-View window, you can select any file in the project `sw` directory to open and edit it. Click on the `tutorial.h` file to view its contents. This file contains all the generated SoC functions and Wishbone addresses.See Figure 5.10 for a snapshot of the software edit window.

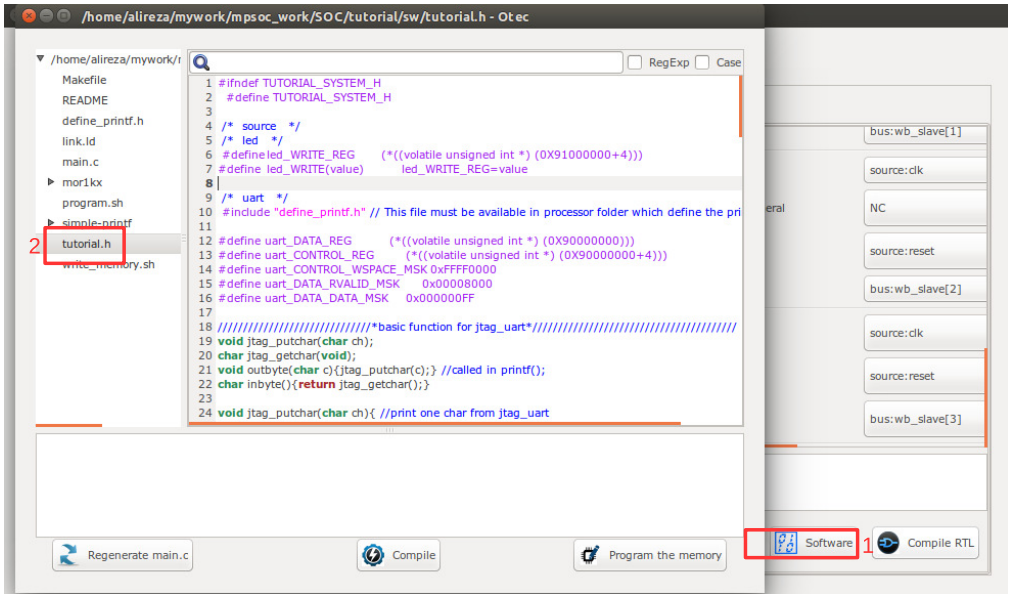


Figure 5.10: The software edit window snapshot.

3. Now click on `main.c` file. Replace the contents of this file with the following C code. This code writes ”Hello worlds!” on the JTAG UART port once and then controls the LEDs using the timer interrupt service routine. Each time an interrupt occurs, the currently lit LED is turned off, and the neighboring LED is turned on. The timer asserts an interrupt every 500 clock cycles. The interrupt time is deliberately chosen to be small to speed up the simulation. In the FPGA implementation which comes later, we will increase the interrupt time to observe the blinking LEDs on the target FPGA board.

```c
#include "tutorial.h"


// a simple delay function
void delay ( unsigned int num ){

   while (num>0){
      num--;
      nop(); // asm volatile ("nop");
   }
   return;
}


char i=1;
void timer_isr(void){
   //write your interrupt code here
   i*=2;
   if((i&0xF)==0) i=1;
   led_WRITE(i);
   timer_TCSR=timer_TCSR; //ack int
   return;
}


int main(){
   printf("hello world!\n");
   delay(500);

   general_int_init();
   general_int_add(timer_INT_PIN, timer_isr, 0);
   general_int_enable(timer_INT_PIN);
   general_cpu_int_en();

   timer_int_init(500);
   while(1){
      delay(500);
   }
   return 0;
}
```

4. Now press the Compile button. This will compile the C code using the Mor1kx GNU toolchain. If everything runs smoothly, you should see a "Compilation finished successfully" message, as shown in Figure 5.11. If there are any errors, check the error message to identify and fix the issues in your code. Once fixed, click the compile button again. Upon successful compilation you will find `ram0.bin, ram0.hex, and ram0.mif` files in your `sw/RAM` directory.

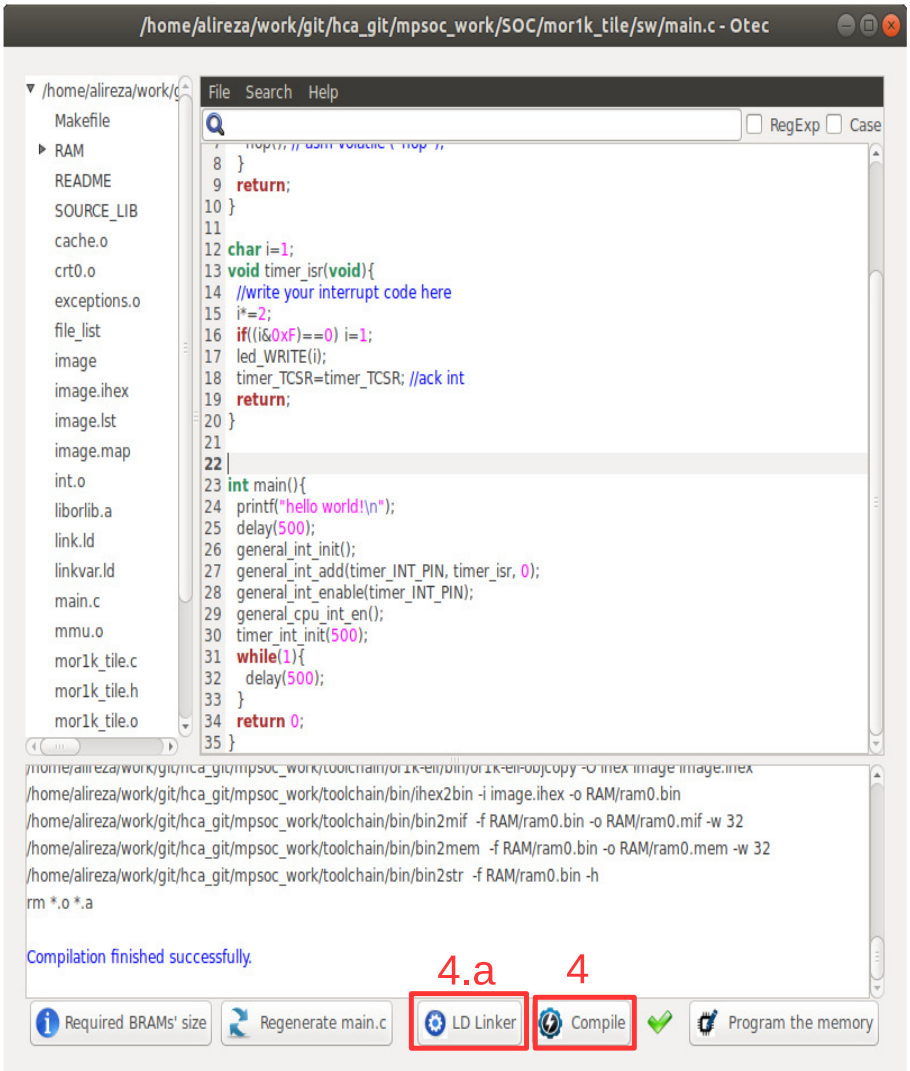


Figure 5.11: Compile the software code.

(a) In case you encounter a region (ram or rom) overflow erro, similar to ex-

ample shown in Figure 5.12, you need to configure the linker (LD) size variables. Click on the LD linker button.

```
/home/alireza/work/git/hca_git/mpsoc_work/toolchain/or1k-elf/bin/or1k-elf-ld: image section `.stack' will not fit in region `ram'
/home/alireza/work/git/hca_git/mpsoc_work/toolchain/or1k-elf/bin/or1k-elf-ld: region `ram' overflowed by 472 bytes
make: *** [image] Error 1

Compilation failed.
```

Figure 5.12: LD Linker error example.

This action will open a new window, as shown in Figure 5.13 where you can redefine the ROM/RAM regions in the executable output file. The file contains the following sections:

- `.vectors`: contains exception handling addresses. This section is mapped to the ROM region.
- `.text`: holds instructions and is mapped to the ROM region.
- `.rodata`: contains read-only data (constants) and is mapped to the ROM region.
- `.data`: holds initialized writable data. As the data in this section is writable, its value may change during execution. After resetting the CPU, the data should return to its initial value. To achieve this, the section has two addresses: the actual section is loaded in ROM at the load address (LMA), and the second address is reserved in RAM, known as the virtual address (VMA) region. During boot time, the entire `.data` section is copied from ROM to RAM, and the executable program only changes the values in the RAM section.
- `.bss`: contains uninitialized data and is mapped to RAM.
- `.stack`: contains the stack and is mapped to RAM.

Therefore, it is important to define the size of the RAM and ROM areas in a way that allows the mentioned sections to fit within them. By default, 75% of the memory is dedicated to the ROM area, while the remaing 25% is reserved for the RAM area. Depending on which area overflows, you can adjust these ratios and retry the compilation process. If both areas are overflowing, you will need to increase the memory size. Additionally, remember to increase the memory width parameter in the memory IP in the processing tile generator window later on (for example, the RAMs' Aw parameter in Table 5.1).

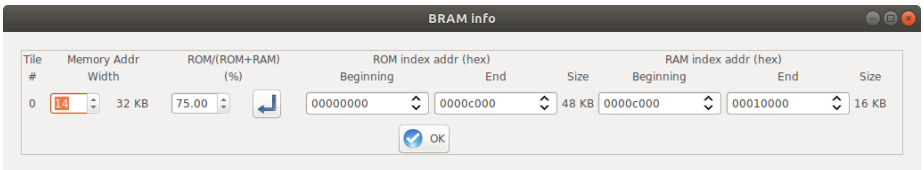


Figure 5.13: LD Linker configuration window snapshot.

Simulate the generated RTL code using Modelsim software

If you have installed Modelsim software on your system, you can simulate your SoC's operation with your developed software. Follow these instructions:

1. Click on the `Compile RTL` button in bottom right corner. This will open the "select compiler window" as shown in Figure 5.14.
2. Select Modelsim as the compiler tool.
3. Enter the path to your installed Modelsim bin directory.
4. Click on the `Next` button.

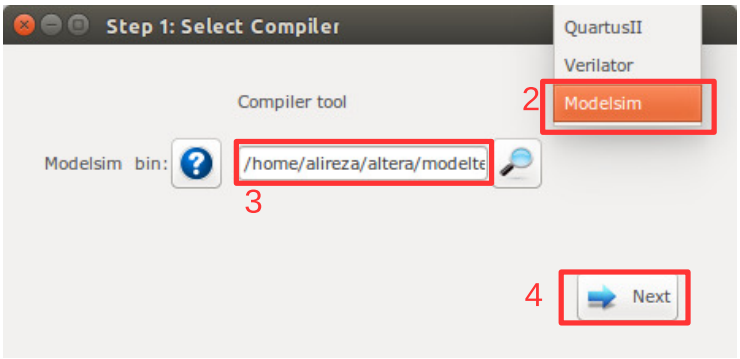


Figure 5.14: Select Modelsim as the simulator.

5. You should now have the `testbench.v` file open in the text editor window, as shown in Figure 5.15. This is the minimum testbench file required to run the simulation of the generated SoC in Modelsim software. It contains the SoC instance module connected to the clock and reset signals. You can edit this file as desired.

6. Click on the  `run` button to run the simulation in Modelsim software.

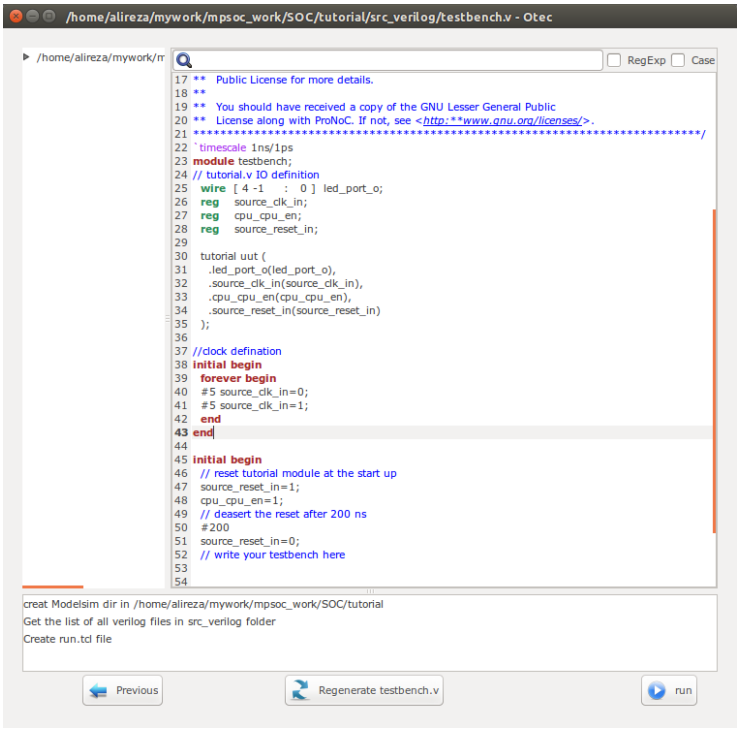


Figure 5.15: Snapshot of the `testbech.v` file.

7. Figure 5.16 displays a snapshot of the Modelsim simulation output. You should see ”hello world!” in the Modelsim terminal, and the output LEDs should cyclically shift to the left with a one-hot code in the Signal Waveform Window.

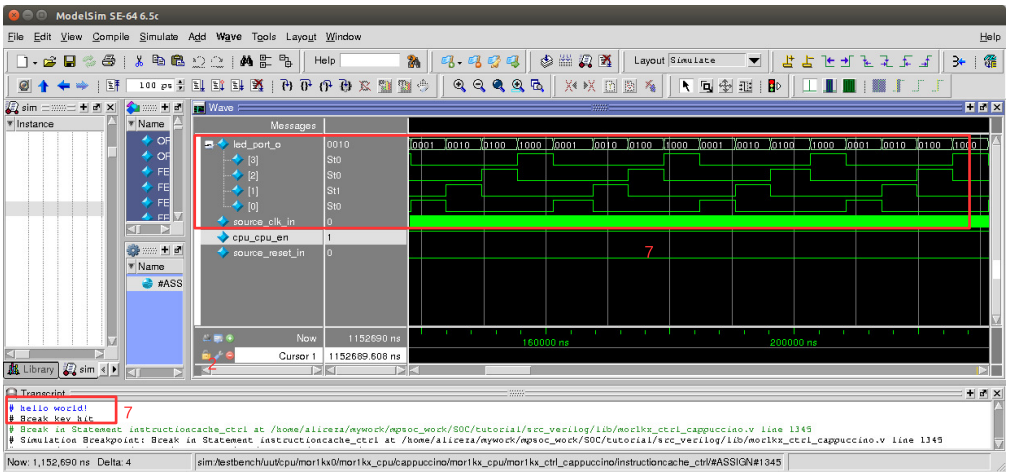


Figure 5.16: Snapshot of the Modelsim output.

Simulate the generated RTL code using Verilator software

If you have installed Verilator software on your system, you can simulate your SoC when it is running your developed software. To do this follow these instructions:

1. Press the `Compile RTL` button in right down corner. This should open "select compiler window" as shown in Figure 5.17. Select Verilator as compiler tool then press `Next`.

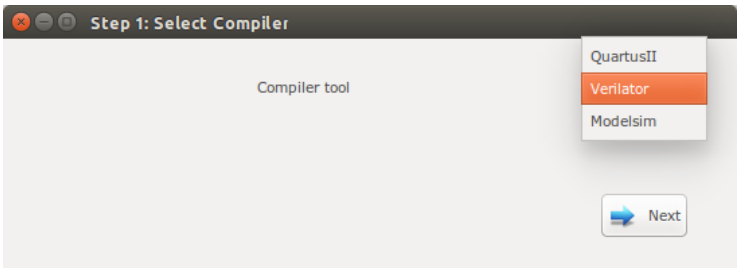


Figure 5.17: Select Verilator compiler.

2. The Verilator Model of your SoC should be generated now. If the model is generated successfully, you must see "Veriator model has been generated successfully!" in the Textview window as shown in Figure 5.18.

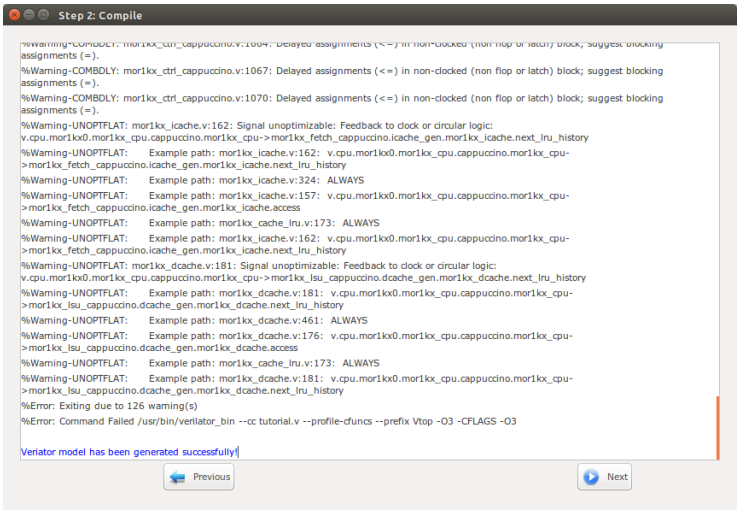


Figure 5.18: Verilator model generation snapshot.

3. Press Next.

4. Now you must have the `testbech.c` opened in software code edit window as shown in Figure 5.19. This is the minimum testbench file for running the generated SoC. It has the SoC instance module connected to the clock and reset signals. You can edit this file as you wish.

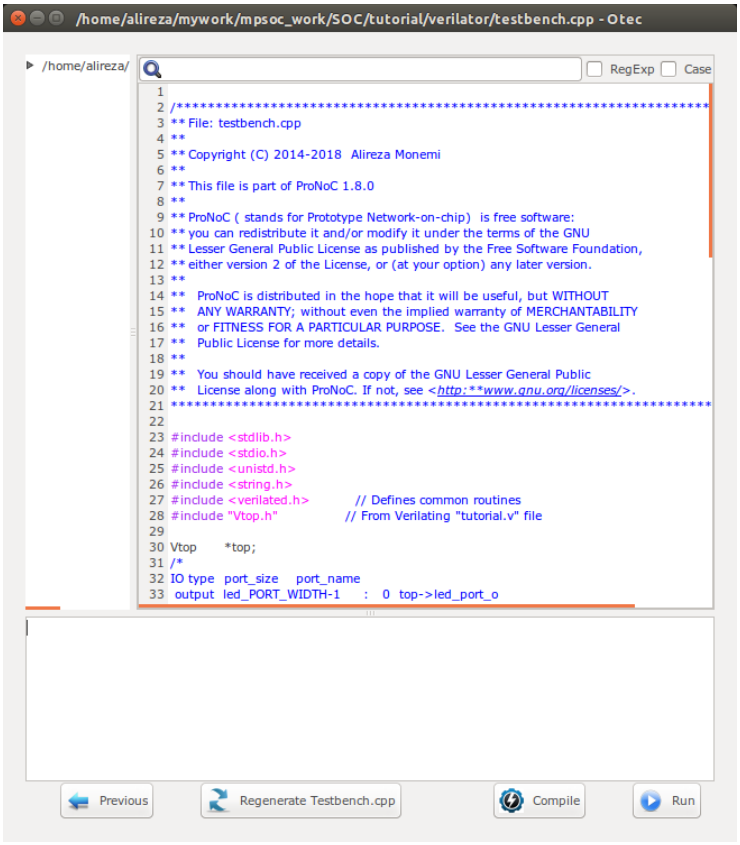


Figure 5.19: Verilator model testbench edit snapshot.

5. We would like to monitor the value of LEDs when running the simulation model. To do this, add the following lines to the `testbech.c` file:

```
43 int led=0;
44 int main(int argc, char** argv) {
45   Verilated::commandArgs(argc, argv);   // Remember args
46   top  = new Vtop;
47
```

```
66   if ((main_time & 1) == 0) {
67     top->source_clk_in=1;
68     // Toggle clock
69     // you can change the inputs and read the outputs here in case they are
70     // captured at posedge of clock
71
72
73  if(led!=top->led_port_o ){
74    led = top->led_port_o ;
75    printf("%X ",top->led_port_o );
76    getchar();
77  }
78
79
80    }//if
81    else
```

6. Press Compile button to generate the executable binary file. If the file is generated successfully you must see the ”compilation finished successfully” message as shown in Figure 5.20.

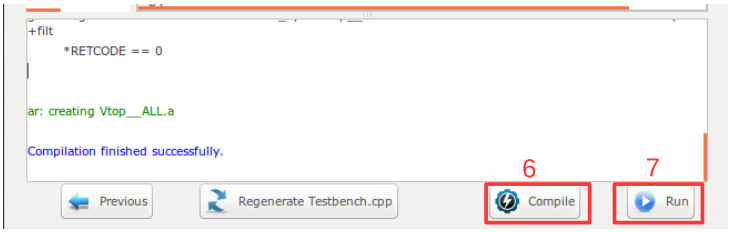


Figure 5.20: Verilator compilation successful snapshot.

7. Now press the Run button. In the successful simulation you must observe the ”Hello world!” sentence in terminal and each time you press the Enter button you must observe the printed value of LED output port change to one of ”1,2,4,8” numbers in order as show in Figure 5.21.

Figure 5.21: Verilator simulation results snapshot.

Compile the generated RTL code using Quartus II/Vivado software

If you have installed Quartus II/Vivado software on your system and you have an Altera/Xilinx FPGA development board, you can prototype your SoC on your target FPGA and change its software code at runtime using following instructions:

1. Press `Compile RTL` button in right down corner. This should open "select compiler window" as shown in Figure 5.22. Select QuartusII or Vivado as the compiler tool depend on your FPGA vendor.

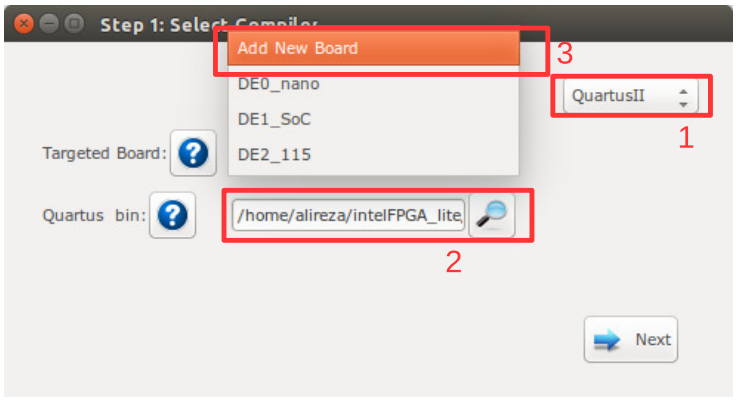


Figure 5.22: Select QuartusII as compiler.

2. Enter the path to your installed QuartusII/Vivado `bin` directory.

3. In `Targeted Board` search for your FPGA board name. If the board exist select it, press the `Next` button and continue from step 5. Otherwise, select `Add New Board`.

4. Press the `Next` button. If you selected `Add New Board`, follow instructions in Add new FPGA board to ProNoC to add your new board to ProNoC library.

5. Assign your SoC pins to your FPGA boards pins as shown in Figure 5.23.

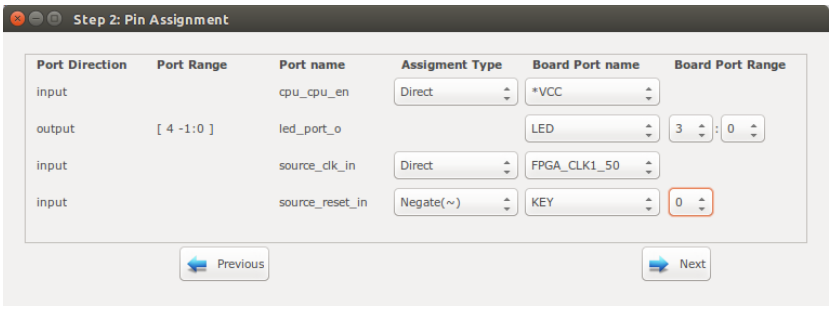


Figure 5.23: SoC pin assignment.

Here, we have a DE10_nano FPGA board which we have used its FPGA_CLK1_50, KEY[0], and LED[3:0] ports. The enable pin is connected to logic 1, led_port [3:0] to LED[3:0], the clk signal to FPGA_CLK1_50 and reset to negate KEY[0]. In DE10_nano FPGA board the KEY[1:0] are push-button switches and are active-high. Hence, to use them as active-high reset sources we have to negate their value.

6. Press the `Next` button.

7. Press the `Compile` button. Then wait for QuartusII compilation tasks to complete.

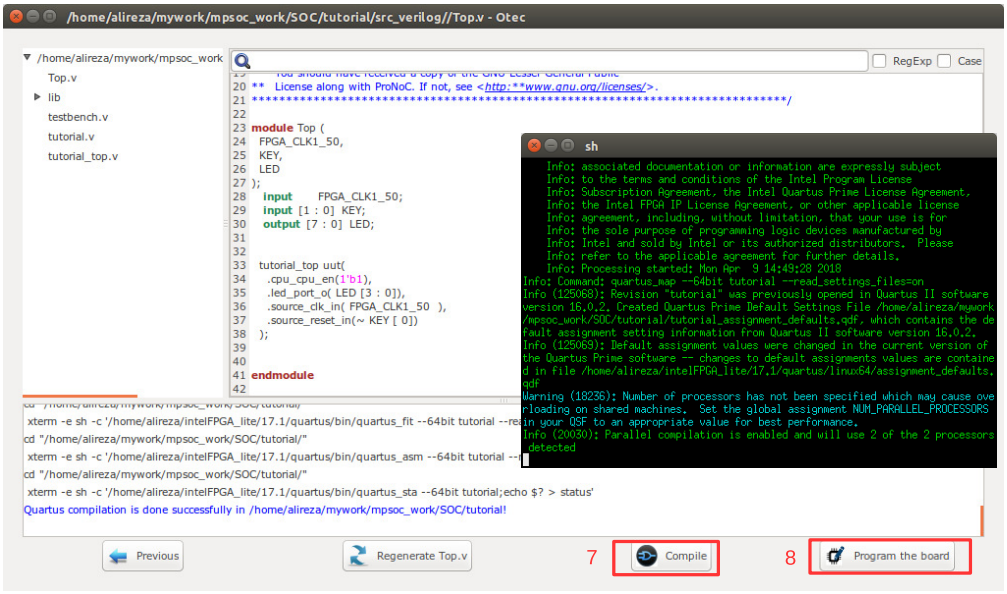


Figure 5.24: QuartusII compilation snapshot.

8. If Quartus compilation is finished successfully, power on your FPGA board and connect it to your PC then press `Program the Board` button to program your FPGA board using the generated sof file.

9. Press `Ctrl+U` to run the UART terminal GUI. Follow instructions on UART terminal GUI chapter to monitor the UART output.

   In case you have Altera FPGA and you have preferred to used `altera_jatag_uart` instead of `ProNoC_JTAG_UART`, open Terminal and type `$QUARTUS_BIN/nios2-terminal`. You must be able to observe the "Hello world!" Sentence in the terminal as shown in Figure 5.25.

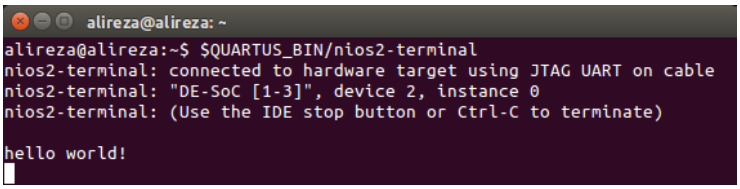


Figure 5.25: nios2-terminal output snapshot.

10. As we mentioned in step 3, the interrupt time is too short to observe the LEDs blinking. To change the interrupt time click on `Software` button and change the timer interrupt time from 500 to 5000000. Then press the `compile` button. By clicking on `Program the Board` button you can reprogram your

SoC memory contents at run time. You should be able to observe the blinking LEDs now.

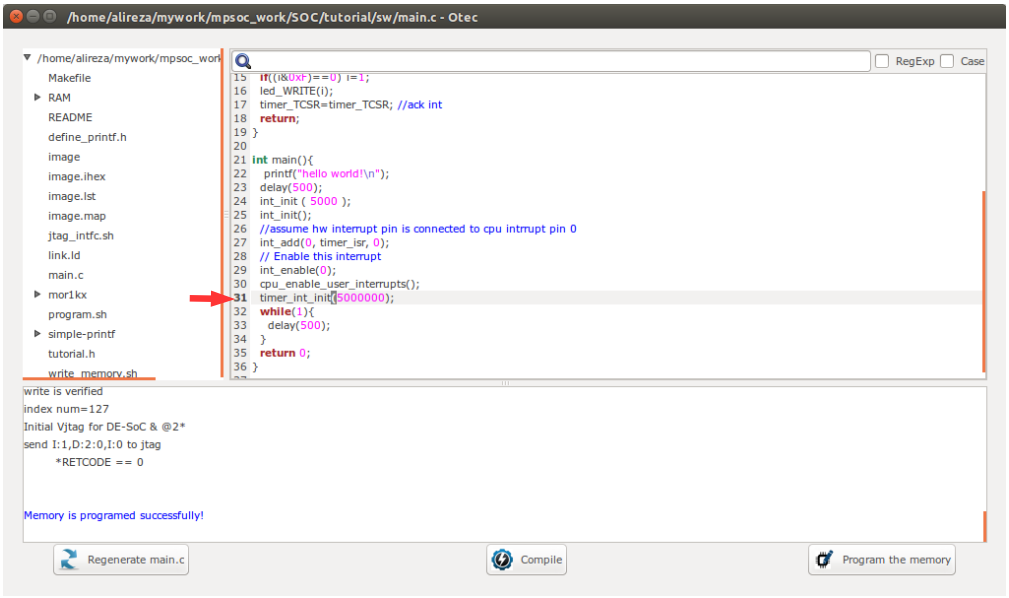


Figure 5.26: Increase timer interrupt time.

CHAPTER 6

Add Custom IP to Processing Tile Generator Tutorial

Summary

This tutorial teaches how to add a custom intellectual property (IP) core to **ProNoC Processing Tile Generator** using **IP Generator**. This tutorial uses a custom Verilog module for calculating the greatest common divisor (GCD) as an example hardware accelerator to be added to ProNoC IP library. The desired system is a Wishbone bus based SoC that is enhanced with GCD accelerator. This SoC will be generated by connecting open-source IP cores on Altera FPGA board.

System Requirements:

You will need an Altera FPGA development board having USB blaster I or II and a computer system running Linux OS with:

1. Installed the ProNoC GUI software and its dependency packages.
2. Installed/Pre-built GNU toolchain of the aeMB soft-core processor.
3. Installed Quarts II (Web-edition or full) compiler.

For more information about the GNU toolchain installation please refer to the Installation Manual for the Ubuntu. In case your FPGA board is not included in ProNoC FPGA board list please follow the instruction given in Adding a New Altera FPGA Board to ProNoC, to add your board to ProNoC.

Objectives:

1. To develop a Wishbone bus based custom Hardware Accelerator (HA) IP core.
2. To extend ProNoC IP core library with a new IP core and its required software header file.

Greatest Common Divisor (GCD) Algorithm

The Greatest Common Divisor (GCD) of two integers p and q, is the largest integer that divides both p and q. GCD can be obtained using Euclidean algorithm as follow:

Data: (p, q): A pair of 8-bit binary positive numbers.
Result: gcd: greatest common divisor
INITIALIZE;
while $p \neq q$ **do**
  if $p > q$ **then**
    $p = p - q$;
  end
  else if $p < q$ **then**
    $q = q - p$;
  end
  else
    $gcd = p$;
  end
end

Algorithm 1: Greatest Common Divisor algorithm.

The GCD flow chart:

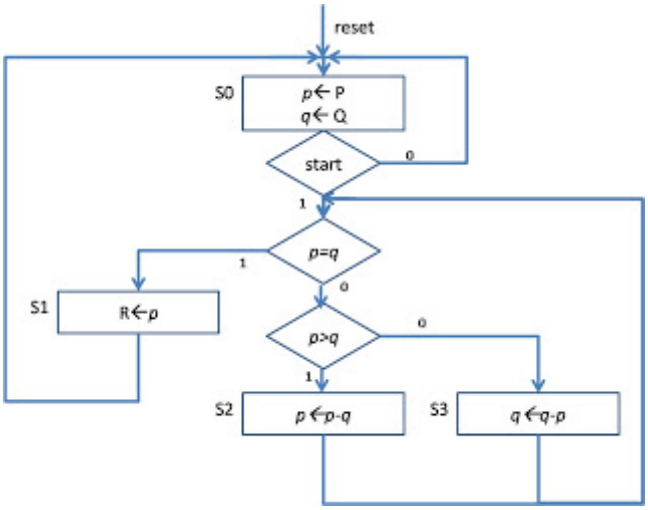


Figure 6.1: GCD flow chart.

GCD RTL code The GCD Verilog RTL code is as follows:

Listing 6.1: gcd.v

```
/************************
* GCD
************************/

module gcd #(
parameter GCDw=32

)( clk, reset, enable, in1, in2, done, gcd);
   input clk, reset;
   input [GCDw-1 : 0] in1, in2;
   output [GCDw-1 : 0] gcd;
   input enable;
   output done;
   wire ldG, ldP, ldQ, selP0, selQ0, selP, selQ;
   wire AeqB, AltB;

   gcd_cu CU(
      .clk (clk),
      .reset (reset),
      .AeqB (AeqB),
      .AltB (AltB),
      .enable (enable),
      .ldG (ldG),
      .ldP (ldP),
      .ldQ (ldQ),
      .selP0 (selP0),
```

```
      .selQ0 (selQ0),
      .selP (selP),
      .selQ (selQ),
      .done (done)
   );


   gcd_dpu #(
      .GCDw(GCDw)
   )DPU(
      .clk (clk),
      .reset (reset),
      .in1 (in1),
      .in2 (in2),
      .gcd (gcd),
      .AeqB (AeqB),
      .AltB (AltB),
      .ldG (ldG),
      .ldP (ldP),
      .ldQ (ldQ),
      .selP0 (selP0),
      .selQ0 (selQ0),
      .selP (selP),
      .selQ (selQ)
      );


endmodule



/************************
* gcd_cu
************************/

module gcd_cu (clk, reset, ldG, ldP, ldQ, selP0, selQ0, selP, selQ, AeqB,
     AltB, done, enable);
   input clk, reset;
   input AeqB, AltB, enable;
   output ldG, ldP, ldQ, selP0, selQ0, selP, selQ, done;
   reg ldG, ldP, ldQ, selP0, selQ0, selP, selQ, done;


   //State encoding
   parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10;
   reg [1:0] y;
   always @ (posedge reset or posedge clk) begin
      if (reset == 1) y <= S0;
      else begin
         case (y)
            S0: begin if (enable == 1) y <= S1;
               else y <= S0;
            end
            S1: begin if (AeqB == 1) y <= S2;
               else y <= S1;
```

```
            end
            S2: begin if (enable == 0) y <= S0;
               else y <= S2;
            end
            default: y <= S0;
         endcase
      end
   end


   always @ (y or enable or AeqB or AltB) begin
      ldG = 1'b0; ldP = 1'b0; ldQ = 1'b0;
      selP0 = 1'b0;
      selQ0 = 1'b0;
      selP = 1'b0;
      selQ = 1'b0;
      done = 1'b0;
      case (y)
      S0: begin
         done = 1'b1;
         if (enable == 1)begin
            selP0 = 1; ldP = 1; selQ0 = 1; ldQ = 1; done = 0;
         end
      end

      S1: begin
         if (AeqB == 1) begin
            ldG = 1;
            done = 1;
         end
         else if (AltB == 1) begin
            ldQ = 1;
         end
         else begin
            ldP = 1; selP = 1; selQ = 1;
         end
      end
      S2: begin
         ldG = 1;
         done = 1;
      end
      default: ;
      endcase
      end
   endmodule



/************************
* gcd_dpu
************************/

module gcd_dpu #(
   parameter GCDw=32

)( clk, reset, in1, in2, gcd, ldG, ldP, ldQ, selP0, selQ0, selP, selQ,
```

```
    AeqB, AltB);
   input clk, reset;
   input [GCDw-1:0] in1, in2;
   output [GCDw-1:0] gcd;
   input ldG, ldP, ldQ, selP0, selQ0, selP, selQ;
   output AeqB, AltB;
   reg [GCDw-1:0] reg_P, reg_Q;
   wire [GCDw-1:0] wire_ALU;
   reg [GCDw-1:0] gcd;
   wire AeqB, AltB;
   //RegP with Multiplex 2:1
   always @ (posedge clk or posedge reset)begin
      if (reset == 1) reg_P <= 0;
      else begin
         if (ldP == 1)begin
            if (selP0==1) reg_P <= in1;
            else reg_P <= wire_ALU;
         end
      end
   end

      //RegQ with Multiplex 2:1
   always @ (posedge clk or posedge reset) begin
      if (reset == 1) reg_Q <= 0;
      else begin
         if (ldQ == 1)begin
            if (selQ0==1) reg_Q <= in2;
            else reg_Q <= wire_ALU;
         end
      end
   end

   //RegG with enable signal
   always @ (posedge clk or posedge reset)begin
      if (reset == 1) gcd <= {GCDw{1'b0}};
      else begin
         if (ldG == 1) gcd <= reg_P;
      end
   end

   //Comparator
   assign AeqB = (reg_P == reg_Q)? 1'b1 : 1'b0;
   assign AltB = (reg_P < reg_Q) ? 1'b1 : 1'b0;

   //Subtractor
   assign wire_ALU = ((selP == 1) & (selQ == 1)) ? (reg_P - reg_Q) : (
       reg_Q - reg_P);
endmodule
```

Create `mpsoc/src_peripheral/other` directory and then copy the above `gcd.v` file inside it.

GCD Simulation

In order to verify GCD hardware module, we use Verilator simulator. Optionally you can use Modelsim as well.

1. If you have not yet installed Verilator simulator on your system run the following

command in terminal

```
sudo apt-get install verilator
```

2. Open terminal in the folder which you have created `gcd.v` file and run:

```
verilator --cc gcd.v
```

If your code is successfully verilated, you will have an `obj_dir` directory that includes all generated GCD object files.

3. Open `obj_dir` folder and create `testbench.cpp` inside it:

Listing 6.2: testbench.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <verilated.h>
#include "Vgcd.h" // From Verilating "gcd.v"

unsigned int input1[10] ={136, 25, 33220, 3627, 3450, 9375, 199317,
    157620, 5694235, 199307 };
unsigned int input2[10] ={248, 50, 2200, 4581, 6540, 61575, 103443,
    238844, 239871, 903443};
unsigned int expt_gcd[10] ={8, 25, 220, 9, 30, 75, 2523, 284, 2161,
    1};

Vgcd *gcd // Instantiation of module

unsigned int main_time = 0; // Current simulation time
int run;
unsigned int i=0,passed=1;

int main(int argc, char** argv) {
   Verilated::commandArgs(argc, argv); // Remember args
   gcd = new Vgcd;
   /********************
   * initialize input
   *********************/
   gcd->reset=1;
   gcd->enable=0;
      gcd->in1=0;
      gcd->in2=0;
   main_time=0;
   run=0;

   while (!Verilated::gotFinish() && i<10) {

      if (main_time & 0x1) {
         gcd-> clk = 0;
         if(gcd-> done==1 && run>6){
            printf("%u : GCD(%u,%u)= %d\t",main_time,gcd->in1, gcd->
                in2, gcd->gcd);
            if(gcd->gcd ==  expt_gcd[i]) printf(" Matched\n");
            else {passed=0; printf(" Error:Miss-matched\n");}
```

```
            i++;
            run=0;

         }
         if(gcd-> enable == 1 && run==5){
            gcd-> enable = 0;
         }
         if(run==4 && gcd->reset==0){
            gcd-> enable = 1;
            gcd-> in1 = input1[i];
            gcd-> in2 = input2[i];


         }
         if (main_time >= 10 ) {
            gcd->reset=0;
            run++;
         }


      }//if
      else {
         gcd-> clk = 1; // Toggle clock

      }//else

      gcd->eval();
      main_time++;


   }
   if(passed) printf( " ********** GCD Testing passed ************\n
       ") ;
   else printf( " ********** GCD Testing failed ************\n");
   gcd->final();

}

double sc_time_stamp () { // Called by $time in Verilog
   return main_time;
}
```

4. Now create a Makefile inside `obj_dir`:

Listing 6.3: Makefile

```
# -*- Makefile -*-


default: sim

MUDUL = Vgcd


include Vgcd.mk

lib:
   $(MAKE) -f $(MUDUL).mk



###################################
# Compile flags

CPPFLAGS += -DVL_DEBUG=1
ifeq ($(CFG_WITH_CCWARN),yes) # Local... Else don't burden users
CPPFLAGS += -DVL_THREADED=1
CPPFLAGS += -W -Werror -Wall
endif
#############################
# Linking final exe -- presumes have a sim_main.cpp


sim: testbench.o $(VK_GLOBAL_OBJS) $(MUDUL)__ALL.a
   $(LINK) $(LDFLAGS) -g $^ $(LOADLIBES) $(LDLIBS) -o testbench $(LIBS) -
       Wall -O3 2>&1 | c++filt

testbench.o: testbench.cpp $(MUDUL).h

clean:
   rm *.o *.a main
```

5. Now to compile the testbench code open terminal in `obj_dir` directory and run:

```
make
```

Sample output:

```
g++ -I. -MMD -I/usr/local/share/verilator/include -I/usr/local/
    share/verilator/include/vltstd -DVL_PRINTF=printf -DVM_TRACE=0
     -DVM_COVERAGE=0 -DVL_DEBUG=1 -c -o testbench.o testbench.cpp
g++ -g testbench.o verilated.o Vgcd__ALL.a -o testbench -lm -lstdc
    ++ -Wall -O3 2>&1 | c++filt
```

This must generate a binary executable file inside `obj_dir` named as `testbench`.

6. To run the simulation run:

```
./testbench
```

Expected output:

```
37 : GCD(136,248)= 8 Matched
51 : GCD(25,50)= 25 Matched
109 : GCD(33220,2200)= 220 Matched
177 : GCD(3627,4581)= 9 Matched
217 : GCD(3450,6540)= 30 Matched
263 : GCD(9375,61575)= 75 Matched
305 : GCD(199317,103443)= 2523 Matched
365 : GCD(157620,238844)= 284 Matched
445 : GCD(5694235,239871)= 2161 Matched
557 : GCD(199307,903443)= 1 Matched
 ********** GCD Testing passed ************
```

Add Wishbone bus interface to GCD

After the GCD core is functionality verified, next is to add Wishbone bus interface to GCD hardware. This interface module provides memory-mapped access of GCD module's input/output ports for the processor. The memory-mapped addresses are illustrated in Table 6.1:

Table 6.1: GCD_IP internal register addresses.

Offset Address	Name	Description	Mode
0	DONE	Holds the value of done output port	Read-only
1	IN1	Write on GCD's module first input variable	Write-only
2	IN2	Write on GCD's module second input variable. Writing on this register will trigger the GCD's enable port	Write-only
3	GCD	Holds the generated GCD value	Read-only

Create the following file inside `mpsoc/src_peripheral/other` directory

Listing 6.4: gcd_ip.v

```
module gcd_ip#(
   parameter GCDw=32,
   parameter Dw =GCDw,
   parameter Aw =5,
   parameter TAGw =3,
   parameter SELw =4
)
(
   clk,
   reset,
   //wishbone bus interface
   s_dat_i,
   s_sel_i,
   s_addr_i,
```

```
    s_tag_i,
    s_stb_i,
    s_cyc_i,
    s_we_i,
    s_dat_o,
    s_ack_o,
    s_err_o,
    s_rty_o

);
    input clk;
    input reset;

    //wishbone bus interface
    input [Dw-1 : 0] s_dat_i;
    input [SELw-1 : 0] s_sel_i;
    input [Aw-1 : 0] s_addr_i;
    input [TAGw-1 : 0] s_tag_i;
    input s_stb_i;
    input s_cyc_i;
    input s_we_i;

    output [Dw-1 : 0] s_dat_o;
    output reg s_ack_o;
    output s_err_o;
    output s_rty_o;

 //Wishbone bus registers address
     localparam DONE_REG_ADDR=0;
     localparam IN_1_REG_ADDR=1;
     localparam IN_2_REG_ADDR=2;
     localparam GCD_REG_ADDR=3;

    assign s_err_o = 1'b0;
    assign s_rty_o = 1'b0;

    wire[GCDw-1 :0] gcd;
    reg [GCDw-1 :0] readdata,in1,in2;
    wire done;

    assign s_dat_o =readdata;

    always @ (posedge clk or posedge reset) begin
       if(reset) begin
          s_ack_o <= 1'b0;
       end else begin
          s_ack_o <= (s_stb_i & ~s_ack_o);
       end //reset
    end//always

    always @ (posedge clk or posedge reset) begin
       if(reset) begin
          readdata <= 0;
          in1 <= 0;
          in2 <= 0;
       end else begin
```

```
        if(s_stb_i && s_we_i) begin //write regiters
           if(s_addr_i==IN_1_REG_ADDR[Aw-1: 0]) in1 <= s_dat_i;
           else if(s_addr_i==IN_2_REG_ADDR[Aw-1: 0]) in2 <= s_dat_i;
        end //sa_stb_i && sa_we_i
        else begin //read registers
           if (s_addr_i==DONE_REG_ADDR) readdata<={{GCDw{1'b0}},done};
           if (s_addr_i==GCD_REG_ADDR) readdata<=gcd;
        end
     end //reset
  end//always

  // start gcd calculation by writiing on in2 register
  wire start=(s_stb_i && s_we_i && (s_addr_i==IN_2_REG_ADDR[Aw-1: 0]));
  reg ps,ns;
  reg gcd_reset,gcd_reset_next;
  reg gcd_en,gcd_en_next;

  always @ (posedge clk or posedge reset) begin
     if(reset) begin
        ps<=1'b0;
        gcd_reset<=1'b1;
        gcd_en<=1'b0;
     end else begin
        ps<=ns;
        gcd_en<=gcd_en_next;
        gcd_reset<=gcd_reset_next;
     end
  end

  always @(*)begin
     gcd_reset_next=1'b0;
     gcd_en_next=1'b0;
     ns=ps;
     case(ps)
        1'b0:begin
           if(start) begin
              ns=1'b1;
              gcd_reset_next=1'b1;
           end
        end
        1'b1:begin
           gcd_en_next=1'b1;
           ns=1'b0;
        end
     endcase
  end


  gcd #(
     .GCDw(GCDw)
  ) the_gcd
  (
     .clk (clk),
     .reset (gcd_reset),
     .enable (gcd_en),
     .in1 (in1),
```

```
        .in2 (in2),
        .done (done),
        .gcd (gcd)
    );

endmodule
```

Add custom wishbone-based IP core to ProNoC Library

In this section, we show how to add previously generated GCD IP core to ProNoC library. However, this can be applied to any other wishbone based IP core.

1. Open `mpsoc/perl_gui` in the terminal and run ProNoC GUI application:

   ```
   ./ProNoC.pl
   ```

2. Then select the  `IP generator`. The IP Generator snapshot is shown in Figure 6.2.

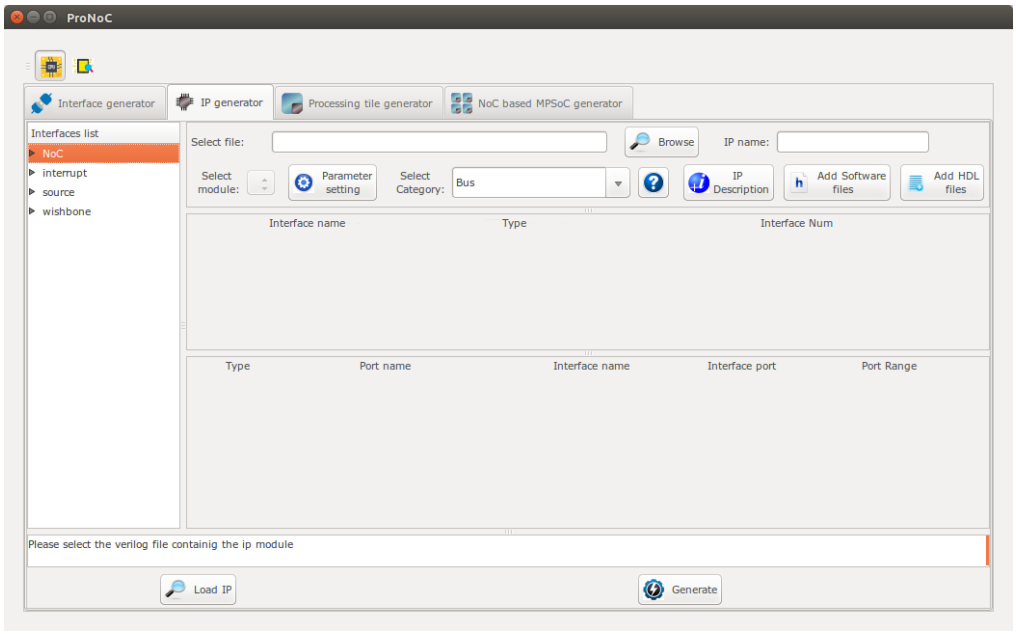


Figure 6.2: ProNoC New IP Generator snapshot.

3. Click on  `Browse` and select `gcd_ip.v` file.
4. Enter `Other` as category name.
5. Enter `gcd` as IP name.

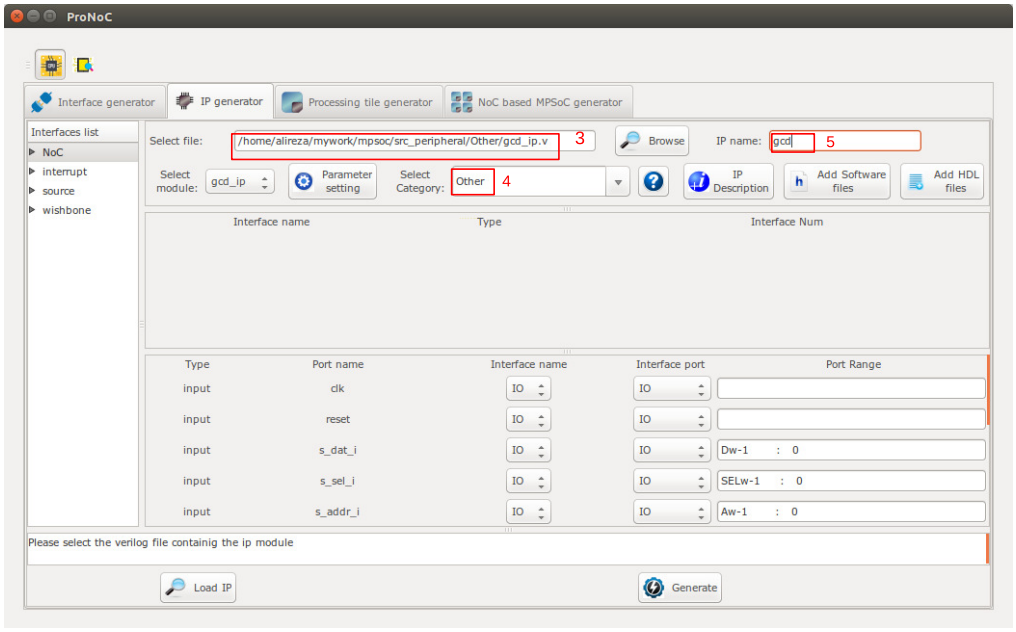


Figure 6.3: Select `gcd_ip.v` file.

6. The `gcd_ip.v` file has one parameter named as `GCDw` which we want to be redefined by the user during IP call time. To define the appropriate GUI interface for this parameter click on `parameter setting` button.
7. In the newly open window, select `Combo-box` as widget type.
8. Enter `8,16,32` as widget content. It will allow the user to select one of these three values for this parameter during Processing tile generation.
9. In the next Combo-box define it as `Localparam`. You can optionally select it as `Parameter`. See here to understand the differences.
10. Click on `IP Description` button to add parameter information.
11. Enter parameter information as `GCD's Input/output width in bits` then press `ok`.
12. In parameter setting window press `ok` to save your setting.

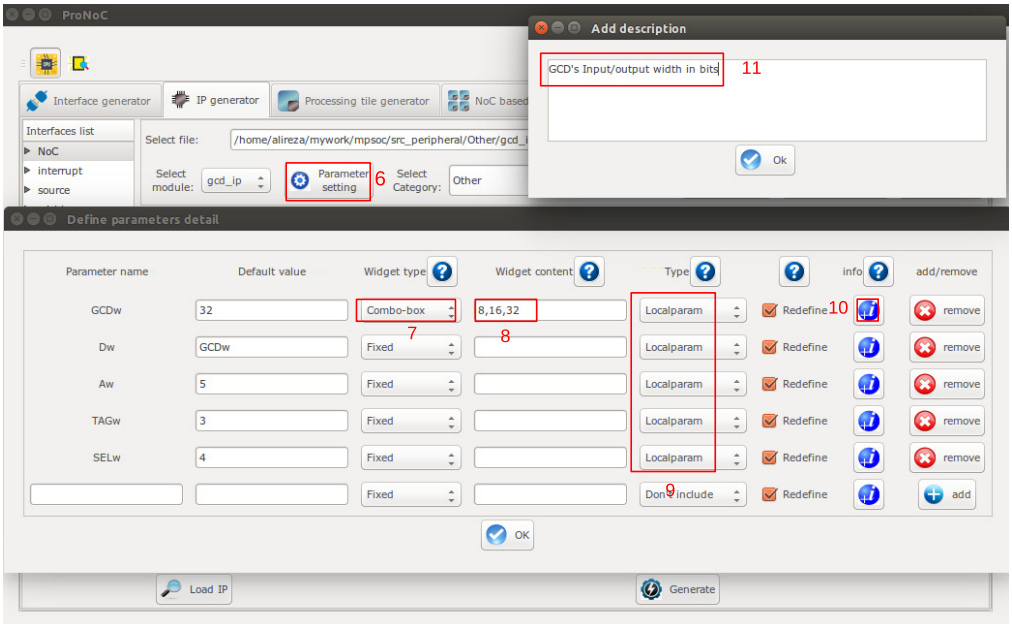


Figure 6.4: GCD IP core parameter setting.

13. In `Interface list` window expand `source` and `wishbone` categories. Then double click on `clk`, `reset` and `wishbone` to add them to the GCD IP library.

14. In Wishbone bus interface row, click on ⚙ button.

15. Select `custom devices` as wishbone address range category.

16. Set block address range as 5. This results in allocating 32 Bytes for each instance of this module. The memory size must be selected equal or greater than the actual IP's internal register size. (GCD has four 32-bit internal registers which are equal to 16 Bytes memory space).

17. Press ✔ `ok`.

    Now we need to map each module individual port to its appropriate interface port. By selecting the interface name, the port with the most similar name is matched with module port name, automatically . For this example the software can match all ports correctly. However in general, you may also needed to adjust the port name as well.

18. Select `plug:clk` for `clk` interface.

19. Select `plug:reset` for `reset` interface.

20. Connect all other ports to `plug:wb_slave` interface.

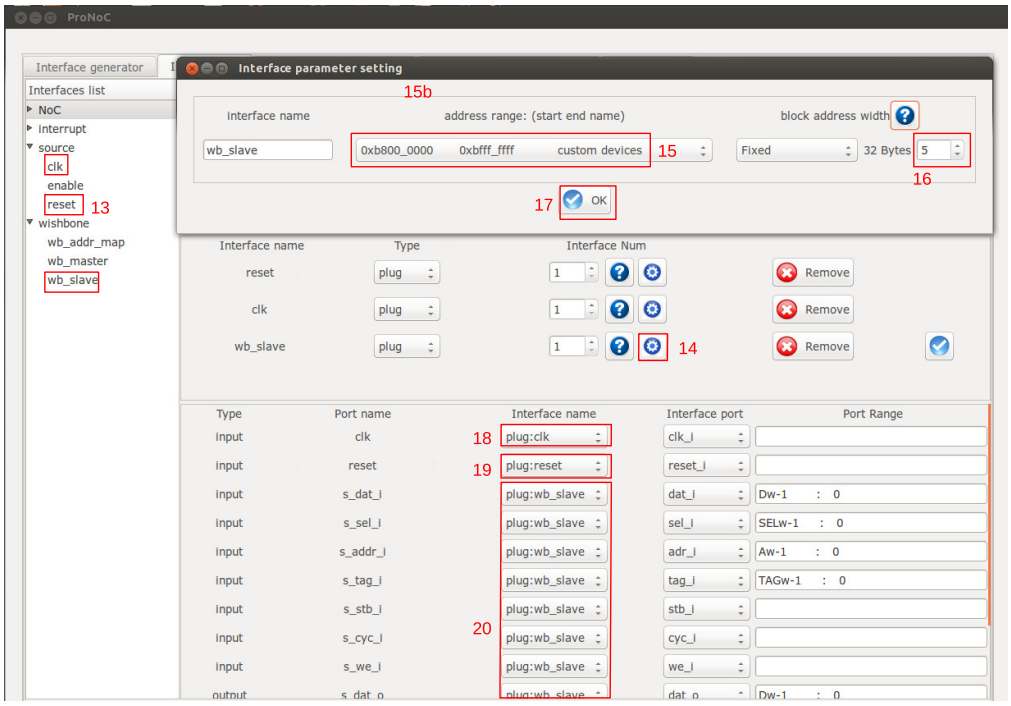


Figure 6.5: GCD Core interface setting.

21. Click on `Add HDL Files` button.

22. In front of `Select file(s)` click on `Browse` button.

23. Select `gcd.v` and `gcd_ip.v` files and press `ok`.

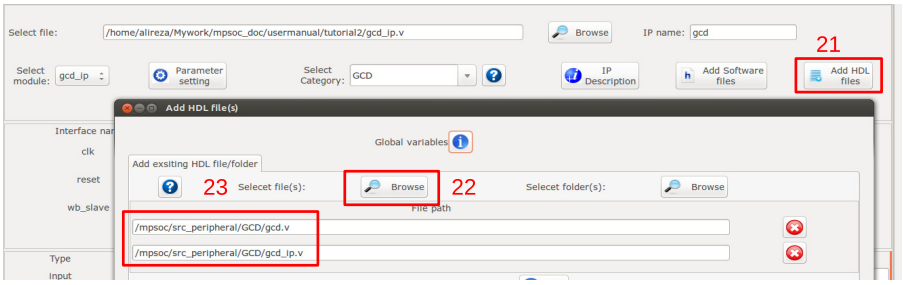


Figure 6.6: Adding GCD core HDL files.

24. Click on `Add software files` button. In the newly opened window, you

can add IP core's software library/header files. The listed files/folder here will be copied in generated SoC project folder inside `sw` directory.

25. Click on `Add to tile.h` tab.
26. Copy following text on the new tab, then click on `Save` button.

```
#define ${IP}_DONE_ADDR (*((volatile unsigned int *) ($BASE)))
#define ${IP}_IN_1_ADDR (*((volatile unsigned int *) ($BASE+4)))
#define ${IP}_IN_2_ADDR (*((volatile unsigned int *) ($BASE+8)))
#define ${IP}_GCD_ADDR (*((volatile unsigned int *) ($BASE+12)))

#define ${IP}_IN1_WRITE(value) ${IP}_IN_1_ADDR=value
#define ${IP}_IN2_WRITE(value) ${IP}_IN_2_ADDR=value
#define ${IP}_DONE_READ() ${IP}_DONE_ADDR
#define ${IP}_READ() ${IP}_GCD_ADDR

unsigned int gcd_hardware ( unsigned int, unsigned int);
```

27. Click on `Add to tile.c` tab.
28. Copy following text on the new tab, then click on `Save` button.

```
unsigned int gcd_hardware ( unsigned int p, unsigned int q ){
      ${IP}_IN1_WRITE(p);
      ${IP}_IN2_WRITE(q);
      while (${IP}_DONE_READ()!=1);
      return ${IP}_READ();
}
```

The entered text here will be added to the `[SoC_name].h` and `[SoC_name].c` file. These files contain all IP cores' wishbone bus addresses, functions and header files. You can use some global variables with `$[variable_name]` format here such as all IP core parameters and IP core Verilog instance name (see the list of complete available variables in ProNoC). These variables will be replaced with their exact values during SoC generation time. In this example, we used variable `${IP}` which is the IP core's instance name. Hence, in case this IP core is called more than once in any SoC, each instance has its own unique WB addresses and functions.

29. Click on `Generate` to add the GCD IP core to the library.

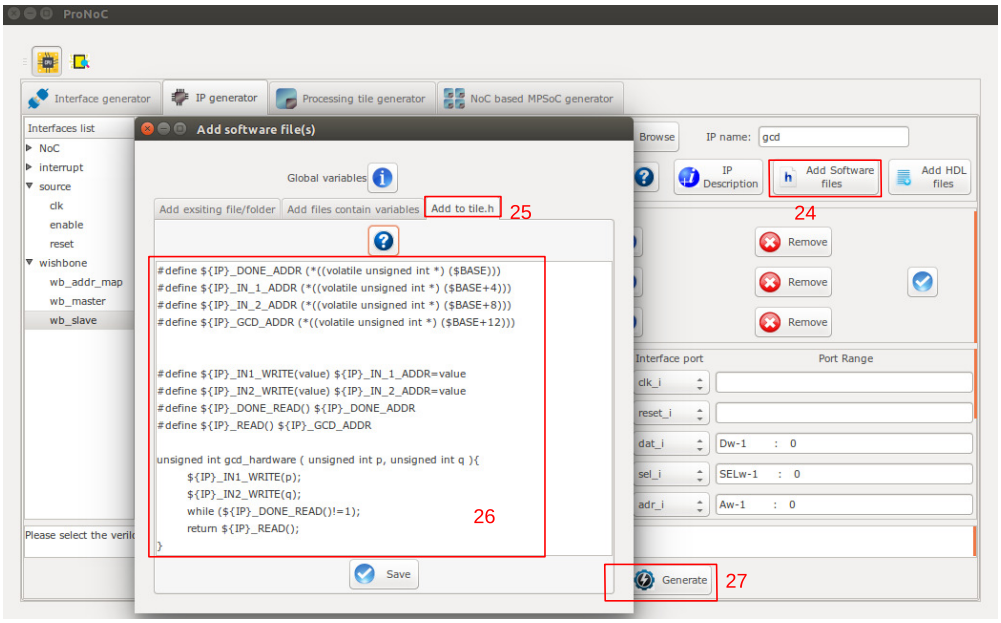


Figure 6.7: Add GCD software files.

Generate a new SoC enhanced with new IP core (GCD)

In this section, we aim to generate an embedded SoC enhanced using generated GCD IP core. The desired SoC schematic is shown in Figure 6.8.

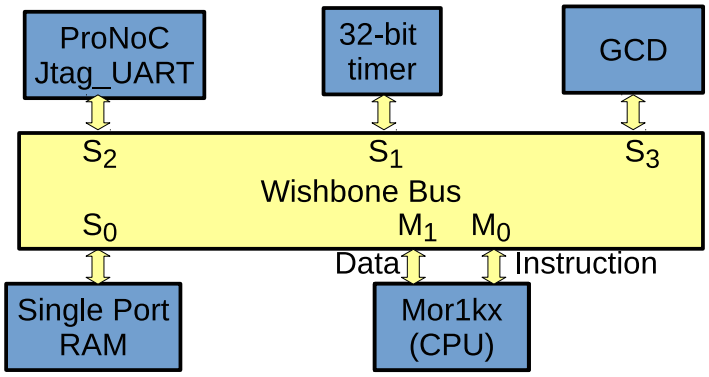


Figure 6.8: Desired SoC with GCD IP core.

1. In ProNoC GUI Click on `Processing Tile Generator`. Use Table 6.2 and follow instruction in Create New SoC Using ProNoC Processing Tile Generator to generate a processing tile.

Table 6.2: GCD SoC IP core list and setting.

Category	IP name	Parameter			Instance name	Interface connection		
Source	clk_source	FPGA_VENDOR	→	"ALTERA"[1]	source	clk reset	→ →	IO IO
Bus	wishbone_bus	M S Dw Aw	→ → → →	2 4 32 32	bus	clk reset	→ →	source:clk source:reset
Processor	Mor1kx	OPTION_DCACHE_SNOOP FEATURE_INSTRUCTIONCACHE FEATURE_DATACACHE FEATURE_IMMU FEATURE_DMMU	→ → → → →	"NONE" "ENABLED" "ENABLED" "ENABLED" "ENABLED"	cpu	clk reset snoop iwb dwb enable	→ → → → → →	source:clk source:reset bus:snoop bus:wb_master[0] bus:wb_master[1] IO
RAM	single_port_ram	Dw Aw BYTE_WR_EN FPGA_VENDOR JTAG_CONNECT JTAG_INDEX BURST_MODE MEM_CONTENT_FILE_NAME INITIAL_EN JTAG_CHAIN	→ → → → → → → → → →	32 14 "YES" "ALTERA" [1] "ALTERA_JTAG_WB"[2] CORE_ID "ENABLED" "ram0" "YES" 4	ram	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[0]
Timer	timer	PRESCALE_WIDTH	→	8	timer	clk reset wb intrp	→ → → →	source:clk source:reset bus:wb_slave[1] cpu:interrupt_peripheral[0]
Communication	ProNoC_jtag_uart	BUFF_Aw JTAG_INDEX JTAG_CHAIN JTAG_CONNECT INCLUDE_SIM_PRINTF	→ → → → →	4 126-CORE_ID 3 "ALTERA_JTAG_WB" [2] SIMPLE_PRINTF	uart	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[2]

[1] → "XILINX" For Xilinx FPGA
[2] → "XILINX_JTAG_WB" For Xilinx FPGA

2. Add the new GCD IP to the SoC.

Table 6.3: GCD SoC IP core list and setting.

Category	IP name	Parameter	Instance name	Interface connection
Other	gcd	GCDw → 32	gcd	clk → source:clk reset → source:reset wb → bus:wb_slave[3]

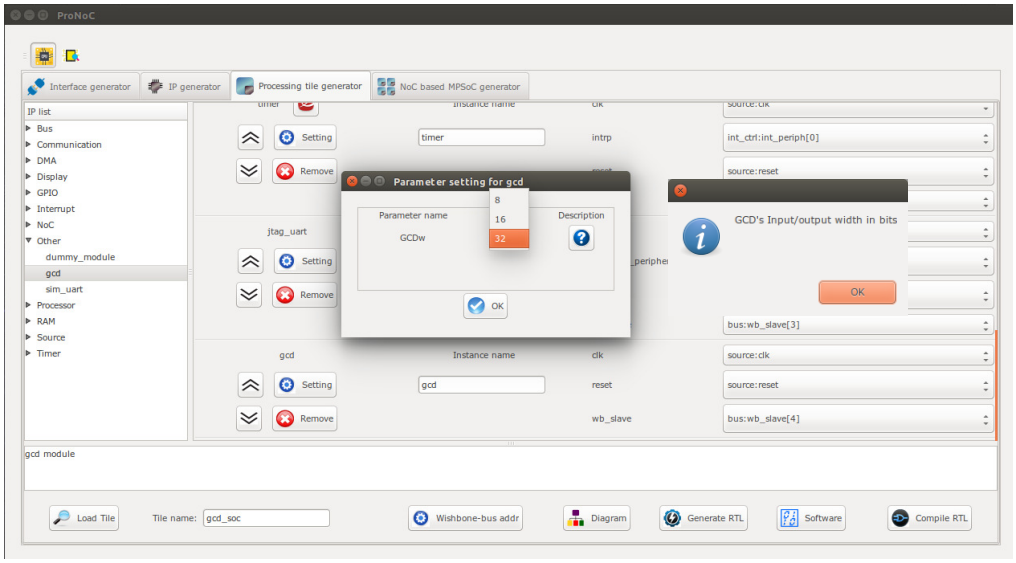


Figure 6.9: Add the generated GCD IP core to `gcd_soc`.

3. Set the tile name as `gcd_soc`.
4. Press the `Generate RTL` button. This must generate a new folder: `mpsoc_work/SOC/gcd_soc`.

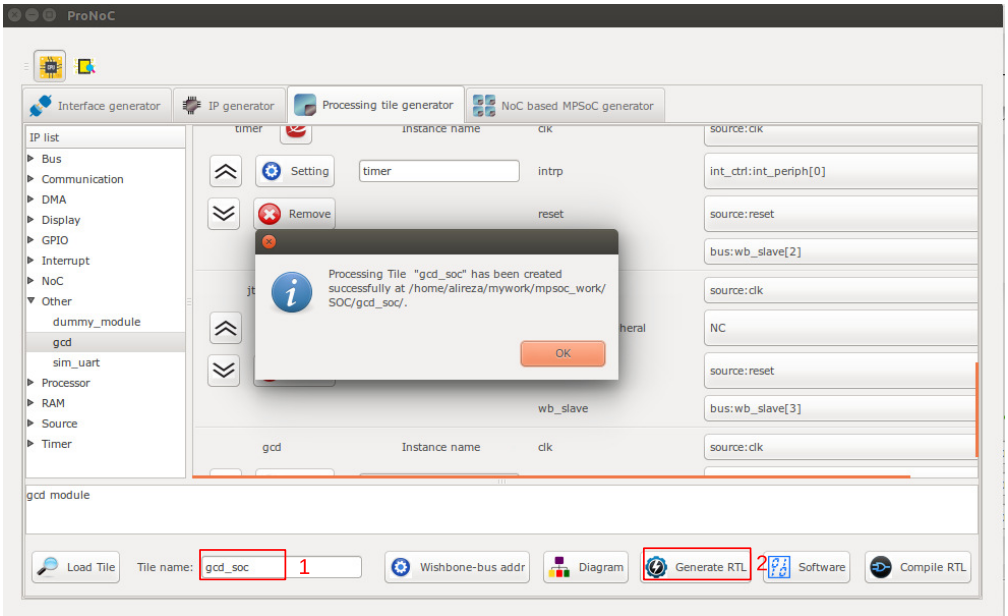


Figure 6.10: Generate the `gcd_soc` RTL codes.

Software Development

1. Click on `Software` button to open the software development window. Now click on `main.c` file. Replace the contents of `main.c` file with the following C code then press compile button. Check software edit terminal output to make sure that compilation ran successfully. If you got RAM or ROM overflow error follows instruction in linker LD setting to fix this error.

```
#include "gcd_soc.h"

unsigned int gcd_software ( unsigned int p, unsigned int q ){
   while (p != q) {
      if (p > q) p=p-q;
      else if (p < q) q=q-p;
   }
   return p;
}



int main(){
   int A,B,C,D;
   unsigned int t_hw,t_sw;
   unsigned int speed;
   printf ("GCD test application\n");
   while(1){
      printf ("Enter number #1:\n");
```

```
        jtag_scanint(&A);
        printf ("Enter number #2:\n");
        jtag_scanint(&B);
        timer_reset();
        timer_start();
        C=gcd_hardware ( A, B);
        timer_stop();
        t_hw=timer_read();
        timer_reset();
        timer_start();
        D=gcd_software ( A, B);
        timer_stop();
        t_sw=timer_read();
        speed=(t_sw*10)/(t_hw);
        printf ("GCD_hardware (%d,%d) = %d\t clock_num=%d\n",A,B,C,
            t_hw);
        printf ("GCD_software (%d,%d) = %d\t clock_num=%d\n",A,B,D,
            t_sw);
        printf ("spead up=%d.%d times\n",speed/10,speed%10);
    }
return 0;
}
```

2. Follow instructions in Compile the generated RTL code using Quartus II software to compile and run the desired SoC on an FPGA board. For instance the pin assignment on DE10-Nano FPGA and a snapshot of a sample result on UART terminal is shown in Figures 6.11 and 6.12, respectively. You can test the GCD IP core by entering different values.

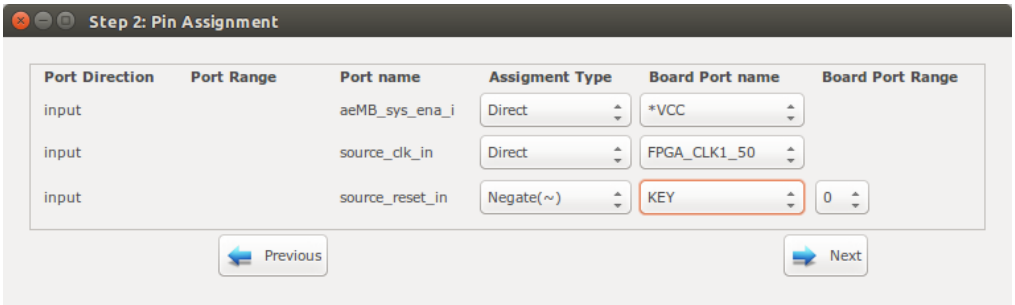


Figure 6.11: DE10-Nano FPGA board pin assignment.

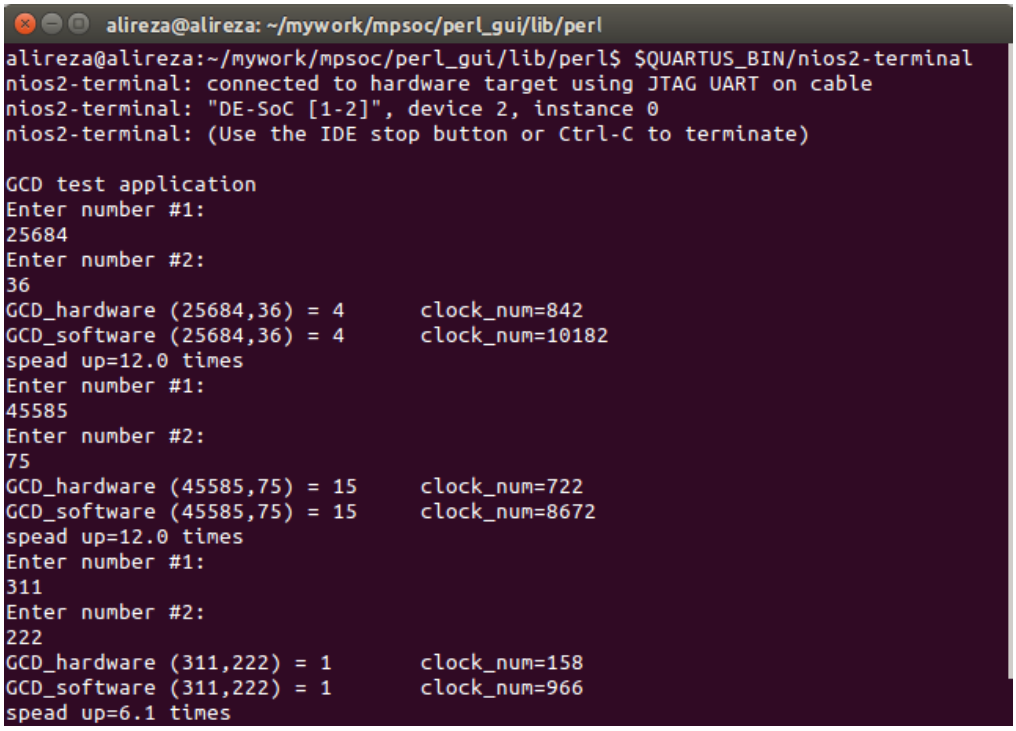


Figure 6.12: Nios2-terminal output snapshots.

CHAPTER 7

Simple message passing demo on 2×2 MPSoC

Summary

This chapter demonstrates a simple message passing on a 2×2 NoC based MPSoC. This includes developing a custom shared bus (Wishbone bus) based processing tile using ProNoC Processing Tile Generator. The generated tile is used then for generating a multicore using ProNoC NoC based MPSoC generator.

System Requirements:

You will need an Altera or Xilinx FPGA development board and a computer system running Linux OS with:

1. Installed the ProNoC GUI software and its dependency packages.
2. Installed/Pre-built GNU toolchain. (or1k-elf) for compiling the Mor1kx soft-core processor software code.
3. Installed Quarts II (Web-edition or full) or Vivado compiler.

For more information about the GNU toolchain installation please refer to the Installation Manual for the Ubuntu. In case your FPGA board is not included in the ProNoC FPGA board list please follow the instructions given in Adding a New FPGA Board to ProNoC, to add your board to the ProNoC library.

Generating a custom Processing tile

Follow the instructions in Processing tile generator tutorial up to the Software Development section and generate a processing tile according to the tile setting listed in Table 7.1. Set the tile name as `mor1k_tile`. Remember to press  `Generate RTL` button at the end to generate the processing tile RTL code.

- Note that the desired tile in this chapter has a network interface (NI) IP to be connected to a NoC.
- The NI has a master interface which can automatically write the arrived packets from the NoC to the main memory. Hence, for the CPUs with enabled Data cache, you need to have either the snoop support (to invalidate the Data cache memory location where the NI writes on it) or map the NI receiver buffer on an uncachebale memory location. For this example we have enabled the snoop support of Mor1kx processor. The snoop interface of the CPU must be connected to the WB to inform about the main memory data changes.

Table 7.1: Desired Processing core IP list and setting.

Category	IP name	Parameter			Instance name	Interface connection		
Source	clk_source	FPGA_VENDOR	→	"ALTERA"[1]	source	clk reset	→ →	IO IO
Bus	wishbone_bus	M S Dw Aw	→ → → →	4 4 32 32	bus	clk reset	→ →	source:clk source:reset
Processor	mor1kx	OPTION_DCACHE_SNOOP FEATURE_INSTRUCTIONCACHE FEATURE_DATACACHE FEATURE_IMMU FEATURE_DMMU	→ → → → →	"ENABLED" "ENABLED" "ENABLED" "ENABLED" "ENABLED"	cpu	clk reset snoop iwb dwb enable	→ → → → → →	source:clk source:reset bus:snoop bus:wb_master[0] bus:wb_master[1] IO
RAM	single_port_ram	Dw Aw BYTE_WR_EN FPGA_VENDOR JTAG_CONNECT JTAG_INDEX BURST_MODE MEM_CONTENT_FILE_NAME INITIAL_EN JTAG_CHAIN	→ → → → → → → → → →	32 14 "YES" "ALTERA" [1] "ALTERA_JTAG_WB"[2] CORE_ID "ENABLED" "ram0" "YES" 4	ram	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[0]
NoC	ni_master	MAX_TRANSACTION_WIDTH MAX_BURST_SIZE Dw CRC_EN HDATA_PRECAPw	→ → → → →	13 16 32 "NO" 0	ni	clk reset interrupt wb_send wb_receive wb_slave	→ → → → → →	source:clk source:reset cpu:interrupt_peripheral[0] bus:wb_master[2] bus:wb_master[3] bus:wb_slave[1]
Timer	timer	PRESCALE_WIDTH	→	8	timer	clk reset wb intrp	→ → → →	source:clk source:reset bus:wb_slave[2] cpu:interrupt_peripheral[1]
Communication	ProNoC_jtag_uart	BUFF_Aw JTAG_INDEX JTAG_CHAIN JTAG_CONNECT INCLUDE_SIM_PRINTF	→ → → → →	4 126-CORE_ID 3 "ALTERA_JTAG_WB" [2] SIMPLE_PRINTF	uart	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[2]

[1] → "XILINX" For Xilinx FPGA
[2] → "XILINX_JTAG_WB" For Xilinx FPGA

Figure 7.1 illustrates the functional block diagram the Mor1k_tile module.

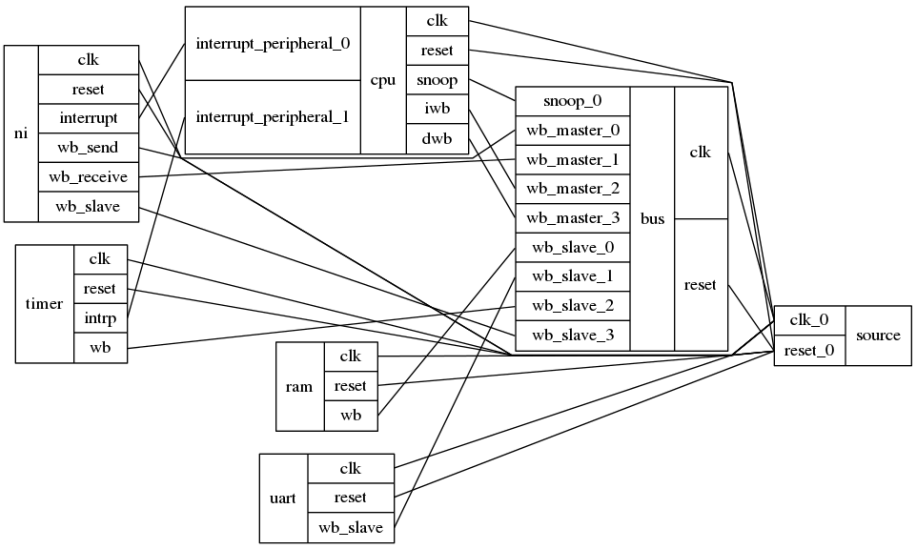


Figure 7.1: Mor1k_tile functional block diagram.

Generating a 4×4 NoC-based MPSoC

1. Click on `NoC based MPSoC generator`
2. Set the `NoC configuration` setting as stated in Table 7.2. Here we have defined two Virtual Networks (VNs) by defining two message classes and separating message class permitted VCs in such a way that each message class can only use its own dedicated VC. For more information regarding the NoC parameters please refer to NoC Verilog File Parameter Description.

Table 7.2: 4×4 NoC configuration setting.

Parameter	Value	Parameter	Value
Router Type	”VC_BASED”	Topology	”MESH”
Router per row	2	Router per column	2
VC number per port	2	Buffer flits per VC	4
payload Width	32	Routing Algorithm	”XY”
SSA Enable	”NO”	VC reallocation type	”NONATOMIC”
VC/SW combination type	”COMB_NONSPEC”	Crossbar mux type	”BINARY”
Class number	2	Class 0 Permitted VCs	☐ ☑
Class 1 Permitted VCs	☑ ☐	Debug enable	0
Add pipeline register after crossbar	0	Swich allocator first level arbiters external priority enable	1
SW allocator arbitration type	”RRA”	Byte Enable	1

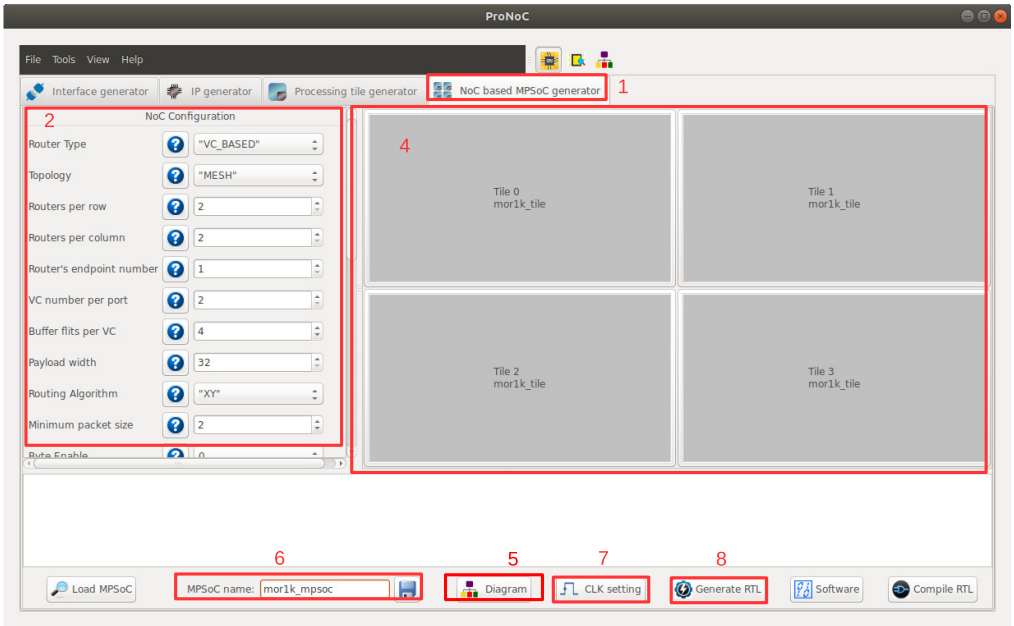


Figure 7.2: NoC-based MPSoC generator snapshot.

3. In `Tile configuration` setting, you should be able to see the list of all processing tile modules which have NI IP core in their shared bus.

   (a) You can change the processing tile default parameters by clicking on its tile name. For this example, we leave the default parameters values unchanged.

   (b) You can enter the tile numbers (location) where this processing tile should be placed in the NoC. Set the `Mor1k_tile` tile numbers as `0,1,2,3` or simply as `1:3`.

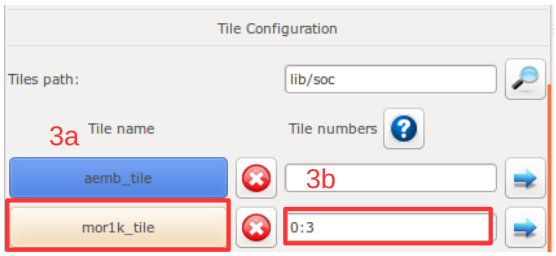


Figure 7.3: Tile Configuration snapshot.

4. You can also map the generated processing tiles on their locations by simply clicking on the tile location in the NoC.

(a) You can select the `Processing tile name` here.

(b) If you wish this processing tile has a custom parameter setting you can select it here. In case you select the `parameter setting` as `custom`, after pressing the OK button, it shows a window where you can change the default parameters values.The tile which has a custom parameter setting is marked by * on its name.

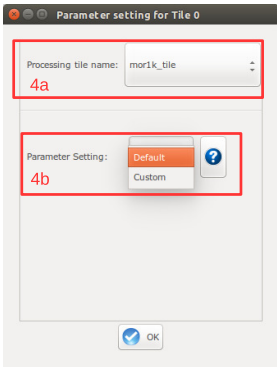


Figure 7.4: Custom Tile setting.

5. You can press the `Diagram` to see the actual location of each tile in the selected topology. In this example:

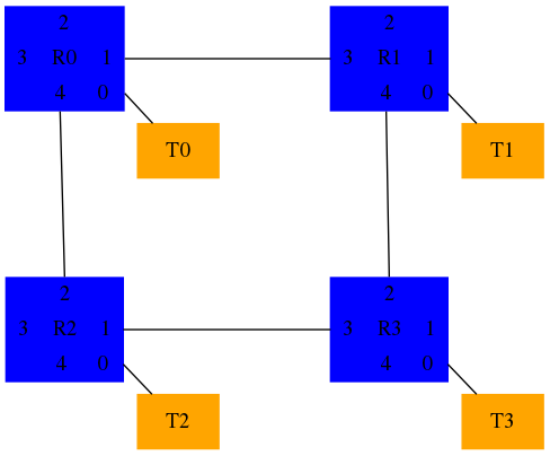


Figure 7.5: 4x4 mesh topology.

6. Set MPSoC name as `mor1k_mpsoc`.

7. In case the MPSoC clk should be generated using FPGA clk pin, click on the `CLK Setting` button then follow in CLK setting to generate the MPSoC clk.

8. Click on  `Generate RTL` button to generate the MPSoC RTL code.

Software Development

1. Click on  `Software` button to open the software development window.
2. In the left Tree-View window, you can select any file in project `sw` directory to open and then edit it. Replace the content of `main.c` files in all tiles with the following C codes.

   In this example, tile 0 to 2 send each 3 packets to tile 3. Tiles 3 shows the packets' content in serial port terminal.

main.c

```
#define MULTI_CORE
#include "mor1k_tile.h"

unsigned char pck1[10]={"first data"};
unsigned char pck2[11]={"second data"};
unsigned char pck3[6]={"123456"};
unsigned char receive_buff[ni_NUM_VCs][16];

// a simple delay function
void delay ( unsigned int num ){
   while (num>0){ num--; nop();}
}

void error_handelling_function(){
   unsigned int i;
   for (i=0;i<ni_NUM_VCs;i++){
      if(ni_got_buff_ovf(i)) {
         printf ("VC%u:The receiver allocated buffer size is smaller
              than the received packet size in core%u\n",i,COREID);
         ni_ack_buff_ovf_isr(i);
      }
      if(ni_got_send_dsize_err(i)) {
          printf ("VC%u:The send data size is not set in core%u\n",i
              ,COREID);
          ni_ack_send_dsize_err_isr(i);
      }
      if(ni_got_burst_size_err(i)){
          printf ("VC%u:The burst size is not set in core%u\n",i,
              COREID);
          ni_ack_burst_size_err_isr(i);
      }
      if(ni_got_invalid_send_req(i)){
          printf( "VC%u:A new send request is received while the DMA
               is still busy sending previous packet in core%u\n",i,
              COREID);
          ni_ack_invalid_send_req_isr(i);
      }
      if(ni_got_crc_mismatch(i)){
         printf( "VC%u:CRC miss-matched in core%u\n",i,COREID);
         ni_ack_crc_mismatch_isr(i);
```

```
      }
   }//for
}//error_handle

void got_packet_funtion(void){
   unsigned int i ;
   unsigned char iport;
   for (i=0;i<ni_NUM_VCs;i++){
      if(ni_got_packet(i)) {
         iport =ni_RECEIVE_PRECAP_DATA_REG(i);
         //different destination can be targeted according to iport
             value
         //E.g if(iport==0) ini_receive (i, (unsigned int)&
             receive_buff_p0[i][0], 16, 0);
         //E.g else if(iport==1) ini_receive (i, (unsigned int)&
             receive_buff_p1[i][0], 16, 0);
         ni_receive (i, (unsigned int)& receive_buff[i][0] , 16, 0);

         ni_ack_got_pck_isr(i);
      }//If ni got packet
   }//for
}// got_packet_funtion

void check_packet_funtion (void){
   unsigned char iport;
   unsigned int i,j ,size ;
   struct SRC_INFOS src_info;
   for (i=0;i<ni_NUM_VCs;i++){
      if(ni_packet_is_saved(i)) {
         src_info=get_src_info(i);
         size=ni_RECEIVE_DATA_SIZE_REG(i); //size in byte
         iport= src_info.r;
         //run a function on the recieved packet according to the
             destination port
         //E.G func_on_recived_packe (iport);
         // but here we just print the recived packet in terminal
         printf("A message of %u bytes is recived from core (%x) in
             vc%u:", size,src_info.addr,i);
         for (j=0;j<size;j++){
            printf("%c", receive_buff[i][j]);
         }
         printf("\n");
         ni_ack_save_done_isr(i);
      }//If ni_packet_is_saved
   }//for
}// check_packet_funtion

void sent_packet_done_funtion (void){
   unsigned char oport;
   unsigned int i;
   for (i=0;i<ni_NUM_VCs;i++){
      if(ni_packet_is_sent(i)) {
         ni_ack_send_done_isr(i);
      }//If ni_packet_is_sent
   }//for
}//sent_packet_done_funtion
```

```c
// NI interrupt function
void ni_isr(void){
   //place your interrupt code here
   if(ni_any_err_isr_is_asserted() ){
      // An error occure
      error_handelling_function();
   }
   if( ni_any_sent_done_isr_is_asserted() ){
      //check which VC has finished sending the packet.
      sent_packet_done_funtion();
   }
   if( ni_any_save_done_isr_is_asserted()){
      //check which VC has finished saving the packet. This function
           must be called before got_packet_funtion
      check_packet_funtion();
   }
   if(ni_any_got_pck_isr_is_asserted() ){
      //check which VC got packet
      got_packet_funtion();
   }
   return;
}


int main(){
   printf("Hi from core %u\n",COREID);
   general_int_init();
   general_int_add(ni_INT_PIN, ni_isr, 0); //ni_INT_PIN
   // Enable ni interrupt (its connected to inttruupt pin 0)
   general_int_enable(ni_INT_PIN);
   general_cpu_int_en();
   // hw interrupt enable function:
   // ni_initial (burst_size, errors_int_en, send_int_en,
       save_int_en, got_pck_int_en)
   ni_initial (16,1,1,1,1); //enable the intrrupt when a packet is
       recived, saved or got any error
   if(COREID == 3) while(1); //Core 3 only receives packets from
       other cores
   //ni_transfer (w, v, class_num, dest_port , start_addr_pointer,
       data_size, dest_phy_addr);
   ni_transfer (1,0, 0, 0,(unsigned int)&pck1[0], 10,
       PHY_ADDR_ENDP_3);
   ni_transfer (1,1, 1, 0,(unsigned int)&pck2[0], 11,
       PHY_ADDR_ENDP_3);
   ni_transfer (1,0, 0, 0,(unsigned int)&pck3[0], 6, PHY_ADDR_ENDP_3
       );
   //printf("core %u sent packet to (%u,%u)",CORID,rnd_dest_x[i],
       rnd_dest_y[i]);
   printf("total sent packets by core%u is %u\n",COREID,3);
   while(1){

   }
   return 0;
}
```

3. Now press the  `Compile` button. This compiles the C codes using Mor1kx GNU toolchain. If everything runs ok, you must see ”compilation finished successfully” message. Otherwise, check the error message to fix your code and press the compile button again. Note that in case you got RAM or ROM overflow errors you can fix them following linker LD setting. If every thing runs successfully you must have `ram0.bin, ram0.hex, and ram0.mif` files in your `sw/tile[n]/RAM` directory, where `n` is the tile number.

4. Follow bellows instruction to see the simulation/compilation results:
   Simulate the generated RTL code using Modelsim software
   Simulate the generated RTL code using Verilator software
   Compile the generated RTL code using Quartus II software

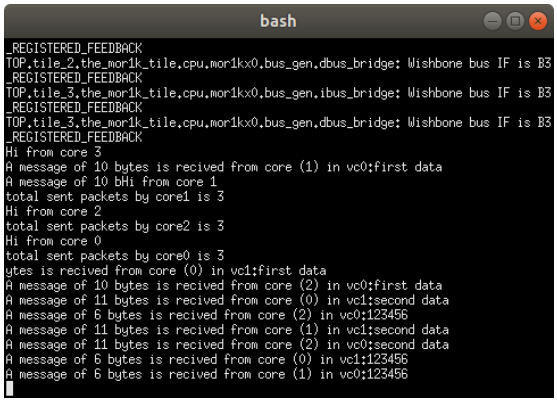


Figure 7.6: Verilator simulation output snapshot.

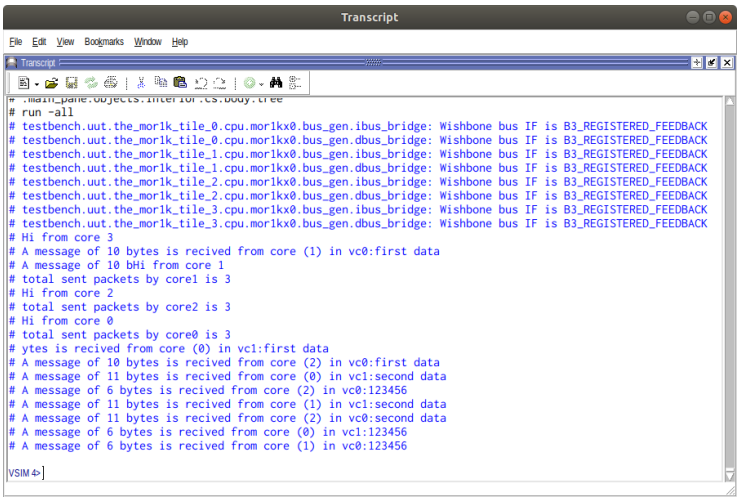


Figure 7.7: Modelsim simulation output snapshot.

CHAPTER 8

Software Auto-generation using CAL language (CAL2C)

Cal2C

CAL2C is a tool that can auto-generate application C code from that application CAL actor Data-flow model. The generated C/C++ code is multi-threaded and can be run in parallel on a multicore system. However, to compile/execute the generated C code requires an operating system (OS) to be installed on the target platform. Meanwhile, the generated multicore by PRONoC has the lack of OS and can only run the bare metal application.

PRONoC comes with a tool that takes the generated C/C++ codes and automatically converts them to several multi bare-metal C/C++ codes that each can be run on a single core. It provides the user a GUI to speed-up the actor mapping process. Several actors can be grouped to be run on a single core. Grouped actors then can be mapped on any custom-defined MPSoC using a drag-and-drop interface or optimally using an application mapping algorithm.

This chapter explains the steps needed to be taken for running an application written in CAL data flow language on a Custom MPSoC.

ORCC installation

To convert an application from CAL to C, you can use the Open RVC-CAL Compiler (ORCC). ORCC can be installed on eclipse IDE as a plugin.

1. Download the eclipse with pre-installed ORCC plugin from: eclipse-orcc. Download and unzip the eclipse in your home directory.

2. Download orc-apps by running the following command in terminal:

```
git clone https://github.com/orcc/orc-apps.git
```

3. This version of eclipse works fine with JDK 8. For higher JDK version it may raise an error. You can use SDKMAN to switch between different JDK versions on your system. To do so open terminal and type:

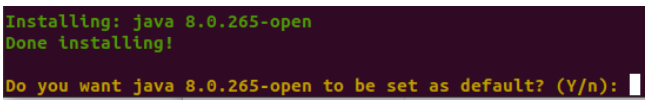
```
curl -s "https://get.sdkman.io" | bash
source "$HOME/.sdkman/bin/sdkman-init.sh"
```

   Now run following command to list all available different JDK versions for your system.

```
sdk list java
```

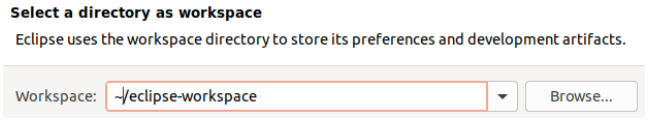
   From the given list, install a java 8. It is not needed to be defined as the default java version at the end of the installation operation. E.g:

```
sdk install java 8.0.265-open
```



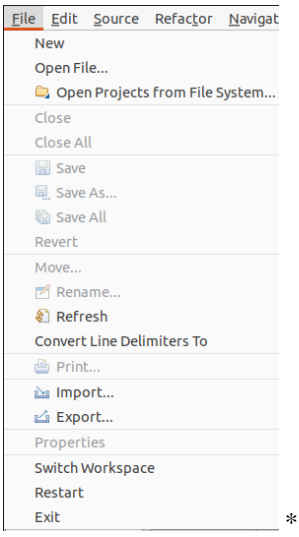
4. Now to run the Eclipse open `eclispe-orcc` folder in terminal then run the following commands. It should ask you to set a workplace directory then open the Eclipse IDE.

```
sdk use java 8.0.265-open
./eclipse
```

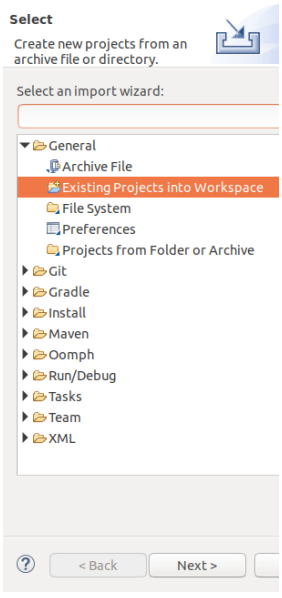


ORCC Hello word on ProNoC platform

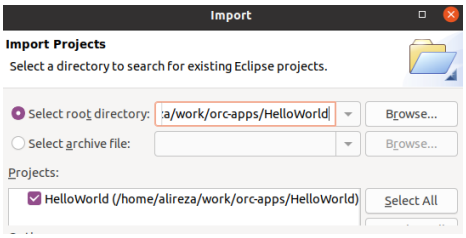
1. From Eclipse menu select `File->import->General->Exsiting Projects into Workspace`



*



2. Select the path to `orc-app/Hello word`:



Run ORCC inbuilt simulator

3. On the `Package Explore` click on `Example.xdf -> Run As -> 2 Orcc Simulation` .

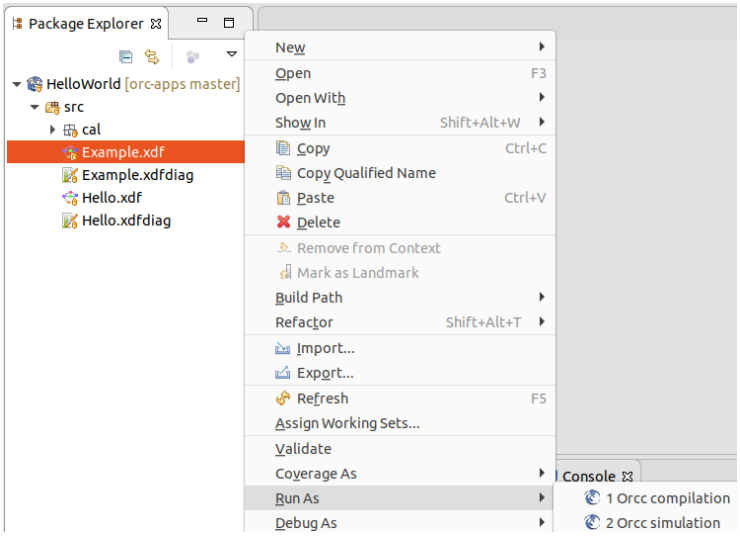


Figure 8.1: ORCC Run as Menu Snapshot.

4. Select a file as input stimulu

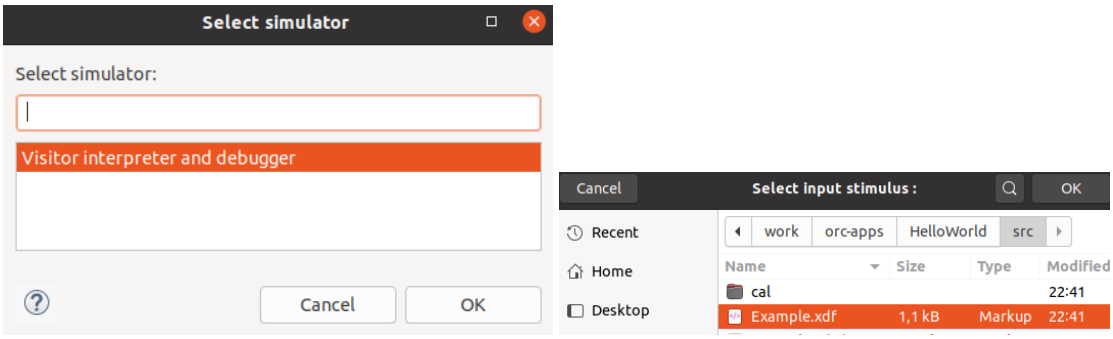


Figure 8.2: ORCC Run as Menu Snapshot.

5. By default, the FIFO width in ORCC is set to 512-bytes (the minimum page-size desktop PC OS). However, this value may be too large for the targeted FPGA platform due to limited memory resources. You can set a smaller value for FIFO width in `Simulation Options` as shown in Figure 8.3.

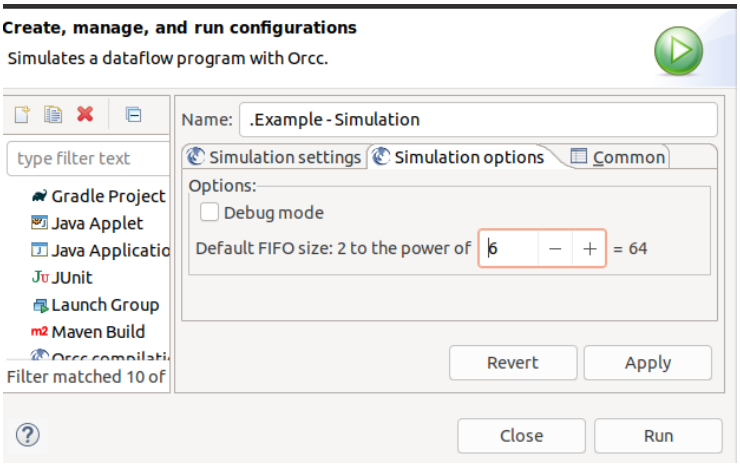


Figure 8.3: Reducing FIFO width.

6. Now you can click on ”Run” button and observe the simulation results in Eclipse console terminal:

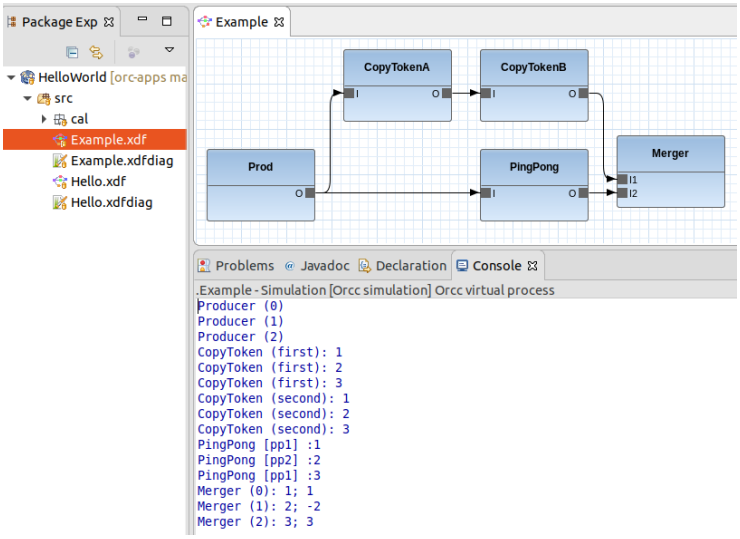


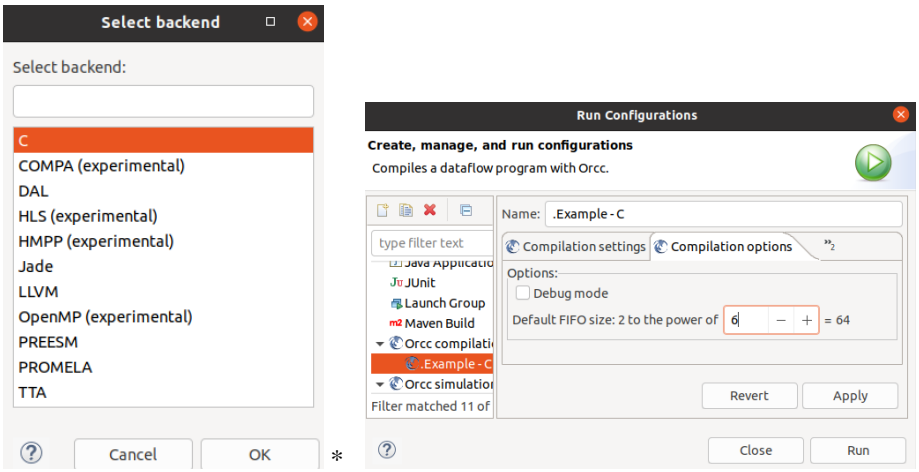
Figure 8.4: Reducing FIFO width.

Run ORCC Compilation

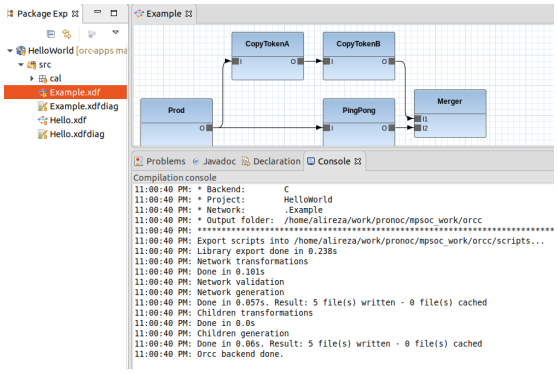
7. On the `Package Explore` click on `Example.xdf -> Run As -> 1 Orcc Compilation` . (see figure 8.1).

8. It now asks you to choose an output folder where the generated C codes will be stored there. Then select `C` as backend and reduce the FIFO width as shown in

the bellow figures: . Create a new folder with the name of `orcc-out` and define it as the output directory in this step.



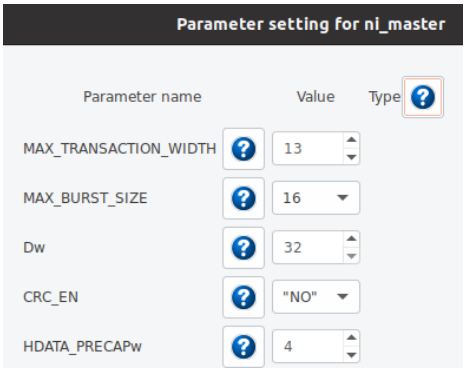
9. Click on the Run button to compile the code:



Modifying the generated C code using ProNoC

The generated C code by ORCC can only be run on the desktop PC having OS. To run them on the ProNoC custom embedded multicore hardware, they needed to be modified using ProNoC tools.

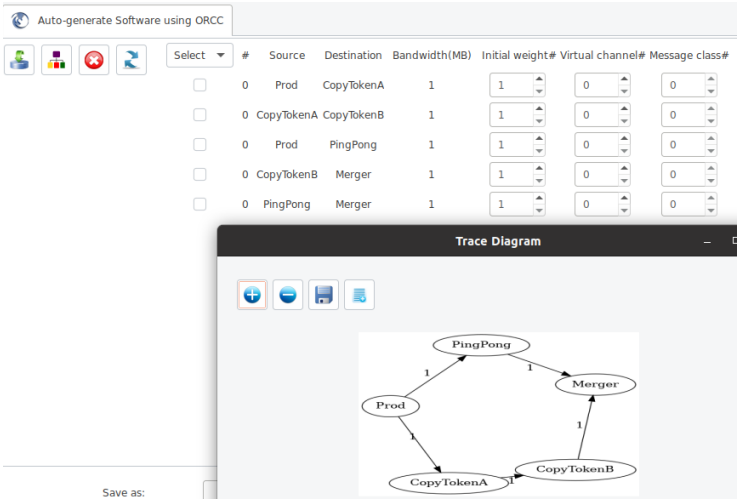
1. Follow instructions on Simple message passing demo on 2×2 MPSoC Chapter up to Software Development Section and generate a Multicore system in RTL.
    - Note that you need to set a value larger than zero (e.g. 4) for header data pre-captured width (HDATA_PRECAPw) parameter in NI configuration setting. This value defines the maximum number of input FIFOs in a processing tile source code that is $2^{(HDATA\_PRECAPw)}$. In case that the number of FIFOs exceeds this limitation, an error is asserted during the compilation time.



- You need to connect the NI interrupt pin to the processing core as the communication between actors located in different processing tiles is handled using interrupt service routine.
- Note that you can target a larger number of processing cores by changing NoC configuration setting.

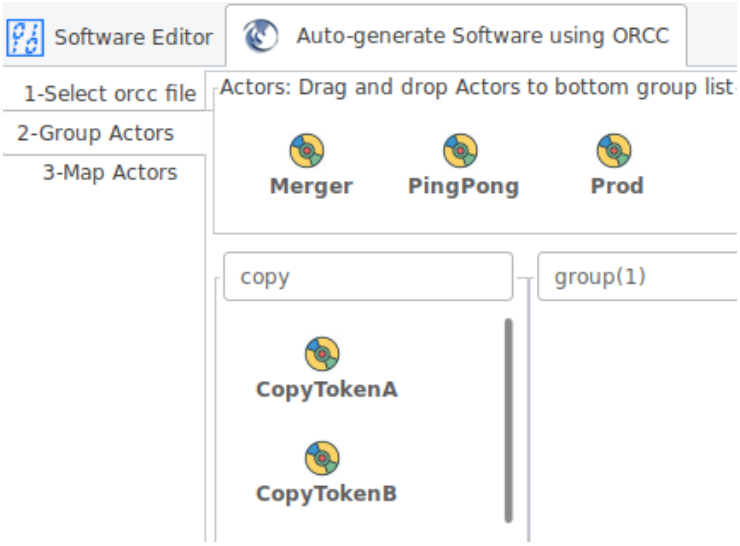
2. Click on the `Software` button to open the software development window. Then click on the `Auto-generate software using ORCC` tab to open ORCC converter page.

3. To load the generated C codes in step 8 on `select a file` page click on button and load `orcc-outdir/src/Example.csv`. It should load the actors in the GUI window as shown in the bellow figure. You can also click on button to see the actor communication graph.

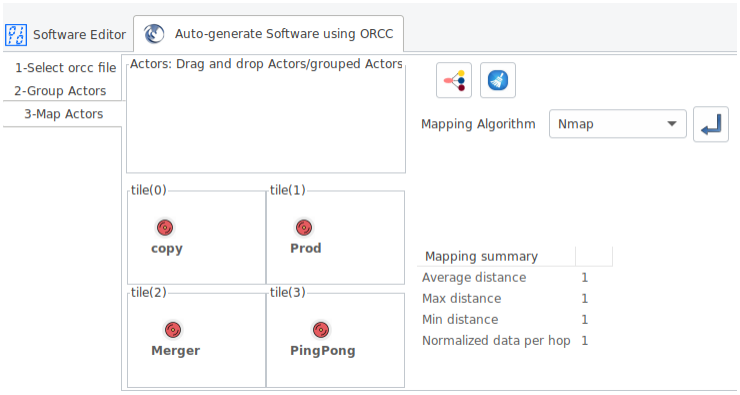


4. Now click on `Group Actors` page. Here you can make different actor groups and put several actors in one group. All actors that are in the same group will

be run on the same processing core. In this example, we have 5 actors and want to run them on a 4-cores processing platform. To do that, we need to make an actor-group containing at least two actors to fit in this target. You can drag any actor icon and drop it in any group window. Let's drag-and-drop `Copy_TokenA` and `Copy_TokenB` in the same group and rename the group as `Copy`, as shown in the bellow figure.

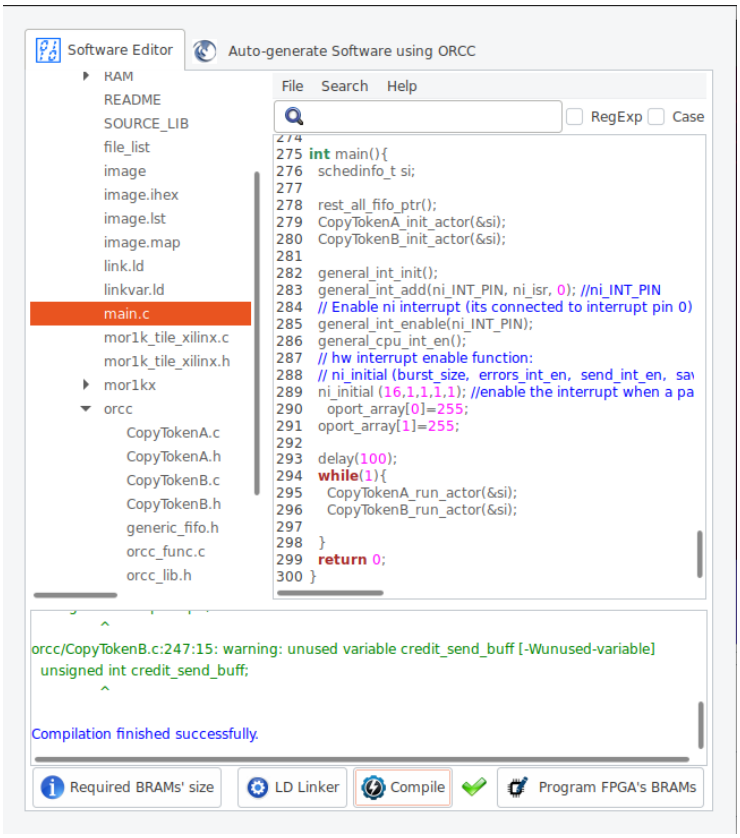


5. Now click on `Map Actors`. Here you can map actors/Grouped actors to processing tiles. Each processing tile can get only one actor or one actor-group. You can drag-and-drop actors to processing tiles or use a mapping algorithm (recommended Nmap) to do the mapping tasks automatically.



6. Click on `Generate` button to generate the bare-metal C codes for each processing core.

7. If everything goes correctly, you must have the new generated codes in your target MpSoC/SW folder. Click on `Software Editor` to compile the generated codes.



8. Now press the `Compile` button. This compiles the C codes using Mor1kx GNU toolchain. If everything runs ok, you must see the "Compilation finished successfully" message. Otherwise, check the error message to fix your code and press the compile button again. Note that in case you got RAM or ROM overflow errors, you can fix them by following the linker LD setting. If everything runs successfully, you must have `ram0.bin, ram0.hex, and ram0.mif` files in your `sw/tile[n]/RAM` directory, where `n` is the tile number.

9. Follow bellow instructions to see the simulation/compilation results:
   Simulate the generated RTL code using Modelsim software
   Simulate the generated RTL code using Verilator software
   Compile the generated RTL code using Quartus II software

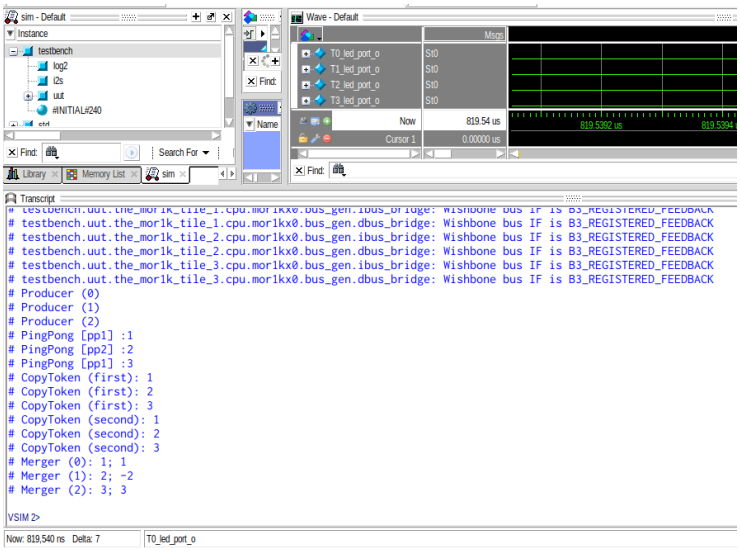


Figure 8.5: The Modelsim simulation snapshot of ORCC hello world example.

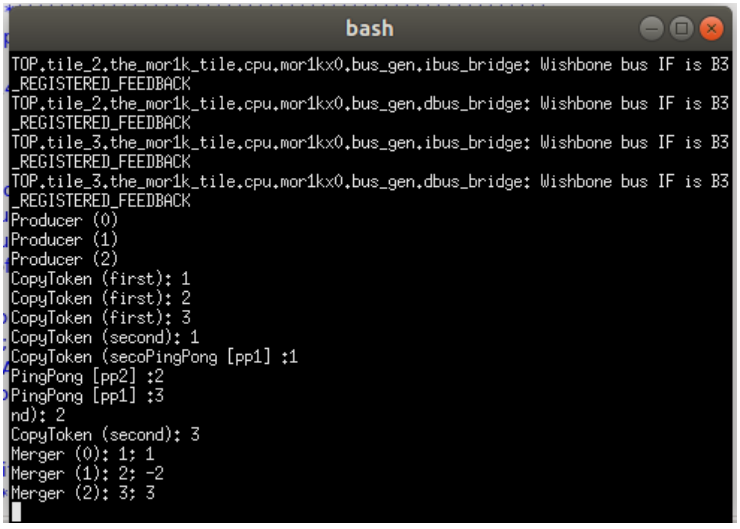


Figure 8.6: Verilaor simulation snapshot of ORCC hello wold example.

CHAPTER 9

NoC Simulator

Summary

The ProNoC NoC is developed in RTL using Verilog HDL and it can be simulated using Verilator simulator. The ProNoC simulator provides the graphical user interface (GUI) for simulating different NoC configuration under different synthetic traffic patterns.

System Requirements:

You will need a computer system running Linux OS with:

1. Installed the ProNoC GUI software and its dependency packages.
2. Installed Verilator simulator.

Simulation Example:

In this example we simulate two 8×8 Mesh NoCs, one with fully adaptive routing and another with DoR routing algorithms.

Generate first NoC simulation model with XY routing

1. Open `mpsoc/perl_gui` in terminal and run ProNoC GUI application:

   ```
   ./ProNoC.pl
   ```

   It should open The GUI interface as illustrated in Figure 9.1.

2. Click on  to open ProNoC simulator tabs.

3. Click on `NoC Simulator` tab to open simulator GUI interface:

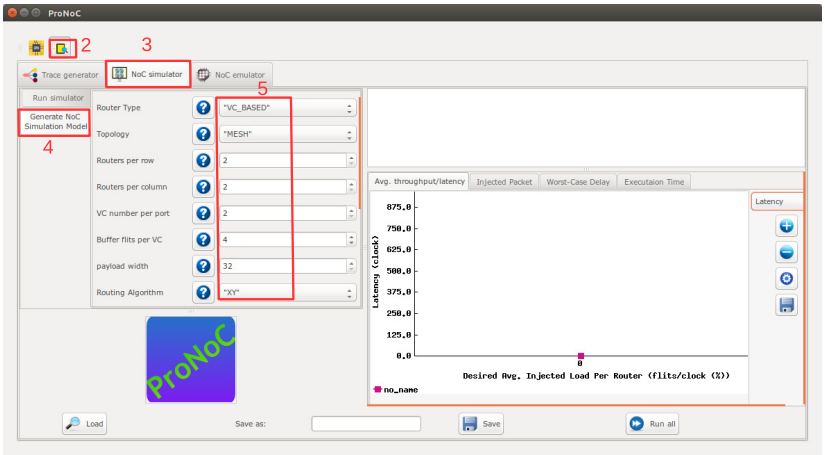


Figure 9.1: NoC simulator snapshot.

4. Click on `Generate NoC Simulation Model` tab to open NoC configuration setting page.

5. Change the default NoC parameters as shown in below table:

Parameter name	Value	Parameter Name	Value
Router Type	”VC_BASED”	Router per row	8
Router per column	8	VC number per port	2
Buffer Flits per VC	2	Payload width	32
Topology	”Mesh”	Routing Algorithm	”xy”
SSA Enable	”NO”	SW allocator arbitration type	”RRA”

6. Enter a name for this NoC configuration (e.g. `mesh_8x8_xy`).
7. Press the generate button.

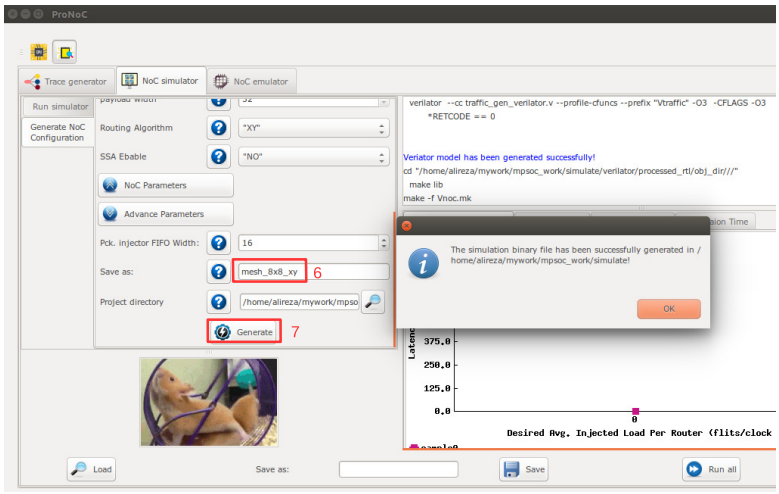


Figure 9.2: Generate NoC simulation model.

Generate the second NoC simulation model with fully adaptive routing

8. In NOC configuration tab, keep the previously set parameters and only change the routing algorithm to ”DUATO”.
9. Enter a new name for this NoC configuration (e.g. `mesh8x8_full`).
10. press Generate button and wait for compilation to be done.

Run simulation under Matrix Transposed traffic pattern

11. Click on `Run simulator` tab.
12. Click on  to add a NoC simulation model.
13. Set following configurations for the simulation model. For flit injection ratios, you can define individual ratios separated by comma (’,’) or optionally you can define a range of injection ratios with `[min]:[max]:[step]` format.

* Note that you can also add more injections ratios later. Each time you run the simulation the simulation results of new injection ratios are added to the previously plotted results.

Parameter name	Value	Parameter Name	Value
Verilated Model	”mesh_8x8_xy”	Traffic Type	Synthetic
Configuration Name	xy	Traffic name	transposed 1
Min pck size	2	Max ock size	10
Total packet number limit	200000	Simulation clock limits	100000
Injection ratios	2:32:2		

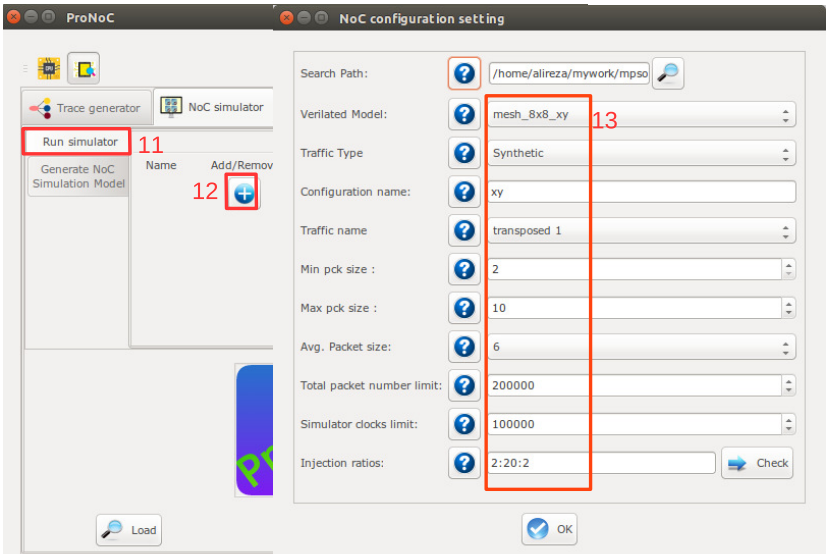


Figure 9.3

14. Click on  to add the second NoC simulation model. Fill the NoC configuration as shown in following table.

Parameter name	Value	Parameter Name	Value
Verilated Model	”mesh_8x8_full”	Traffic Type	Synthetic
Configuration Name	fully	Traffic name	transposed 1
Min pck size	2	Max ock size	10
Total packet number limit	200000	Simulation clock limits	100000
Injection ratios	2:32:2		

15. Save the simulation. You can save the simulation at any time during run time. Hence, later you can continue the rest of simulation.

16. To start the simulation press  `Run all` button. You can also run each individual simulation by pressing the  `Run` button in its simulation row.

17. After the simulation is done, if your graph is not yet completed you can enter a new injection ratio range and press the  `Run` key again.

18. You can edit the generated graph and then save it from graph editing toolbox. By saving the simulation graph, the simulation results is also provided in a text file as well.

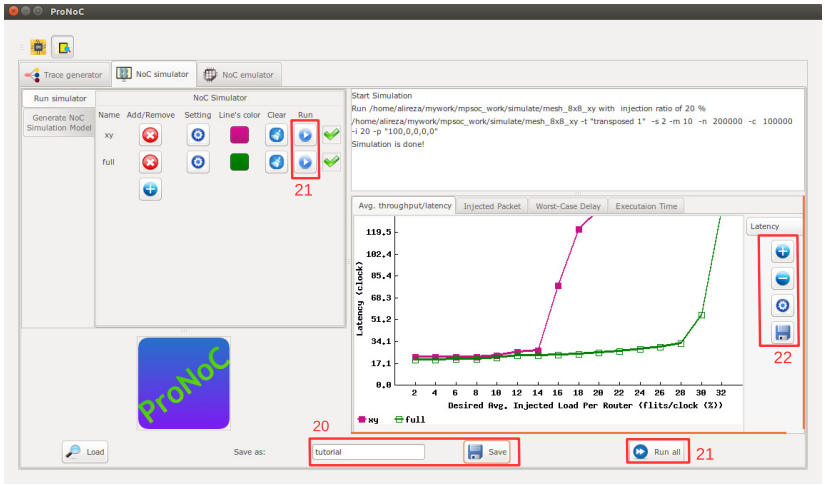


Figure 9.4

For each simulation experiment five simulation results are obtained:

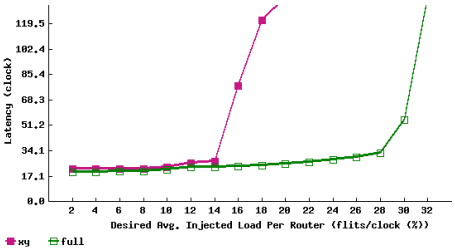
(a) Average latency per average desired flit injection ratio

(b) Average throughput per average desired flit injection ratio

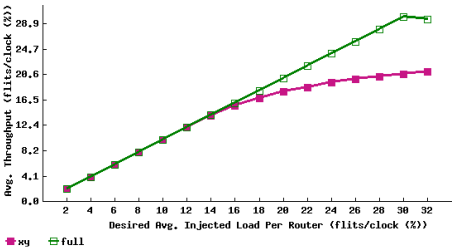
(c) send/received packets number for each router at different injection ratios

(d) send/received worst-case delay for each router at different injection ratios

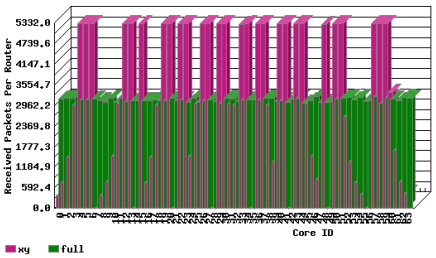
(e) Simulation execution clock cycles



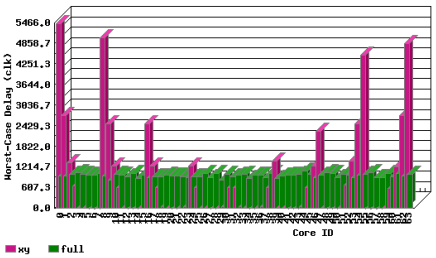
(a) Load-latency



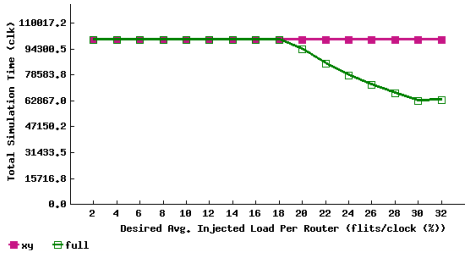
(b) Load-throughput



(c) Injected packets per router at 32% injection ratio.



(d) Worst-case delay per router at 32% injection ratio.



(e) Simulation time in clock cycles.

Figure 9.5: Simulation sample results.

CHAPTER 10

NoC Emulator

Summary

ProNoC comes up with a GUI for emulating an actual NoC on Altera FPGAs. The ProNoC emulator is a programmable packet injector module that can be programmed at run time using Altera JTAG interface. These modules inject/sink packets to the prototype NoC according to the traffic patterns.

System Requirements

You will need an Altera FPGA development board having USB blaster I or II and a computer system running Linux OS with:

1. Installed the ProNoC GUI software and its dependency packages.
2. Installed Quarts II (Web-edition or full) compiler.

For more information about the GNU toolchain installation please refer to the Installation Manual for the Ubuntu. In case your FPGA board is not included in ProNoC FPGA board list please follow the instruction given in Add new FPGA board to ProNoC, to add your board to ProNoC.

Emulation Example:

In this example we simulate two 5×5 Mesh NoCs, one with fully adaptive routing and another with DoR routing algorithms using DE10-nano Altera FPGA board.

Generate first NoC emulation model with XY routing

1. Open `mpsoc/perl_gui` in terminal and run ProNoC GUI application:

   ```
   ./ProNoC.pl
   ```

   It should open The GUI interface as illustrated in Figure 10.1.

2. Click on  to open ProNoC simulator tabs.

3. Click on `NoC Emulator` tab to open the emulator GUI interface:

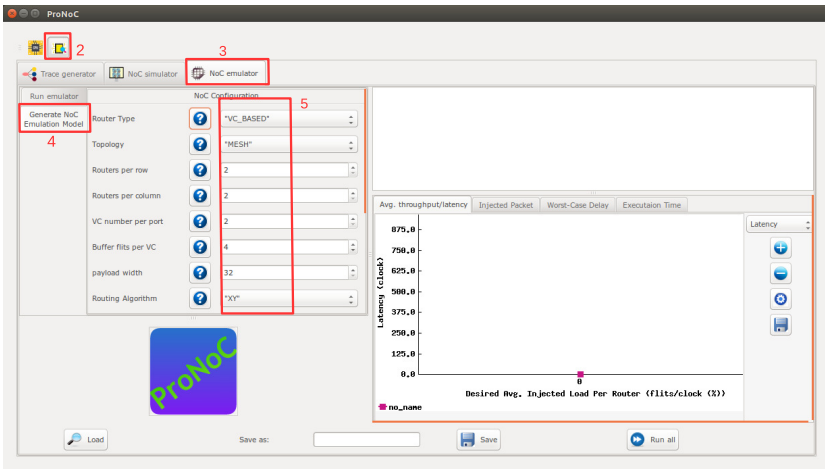


Figure 10.1

4. Click on `Generate NoC Emulation Model` tab to open NoC configuration setting page.

5. Change the default NoC parameters as shown in below table:

Parameter name	Value	Parameter Name	Value
Router Type	”VC_BASED”	Router per row	5
Router per column	5	VC number per port	2
Buffer Flits per VC	2	Payload width	32
Topology	”Mesh”	Routing Algorithm	”xy”
SSA Enable	”NO”	SW allocator arbitration type	”RRA”

6. Enter a name for this NoC configuration e.g. `mesh_5x5_xy`.

7. Press the generate button.

Figure 10.2: Generate NoC model

8. Follow instructions in Compile the generated RTL code using Quartus II software to compile the desired emulation model for an Altera FPGA board. For this example we used the DE10-Nano FPGA board which its pin assignment is shown in Figures 10.3.

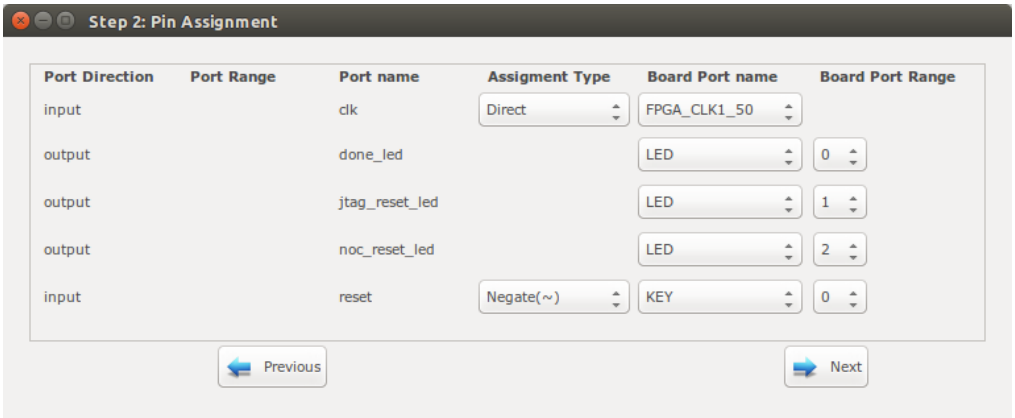


Figure 10.3: DE10-Nano FPGA board pin assignment.

Generate the second NoC emulation model with fully adaptive routing

9. In NOC configuration tab, keep the previously set parameters and only change the routing algorithm to ”DUATO”.
10. Enter a new name for this NoC configuration e.g. `mesh5x5_full`.
11. Generate the NoC emulation model in similar way to step 8.

Run Emulation models under Matrix Transposed traffic pattern

12. Click on `Run Emulator` tab.
13. Click on  to add a NoC emulation model.
14. Set following configurations for the emulation model. For flit injection ratios, you can define individual ratios separated by comma (’,’) or optionally you can define a range of injection ratios with `[min]:[max]:[step]` format.

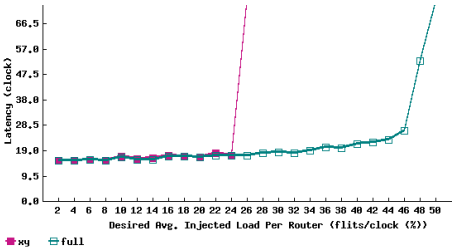
    * Note that you can also add more injections ratios later. Each time you run the emulation the emulation results of new injection ratios are added to the previously plotted results.

Parameter name	Value	Parameter Name	Value
FPGA Board	[Your FPGA board name]	Sram Object file	”mesh_5x5_xy”
Configuration Name	xy	Traffic name	transposed 2
Packet size in flits	5	Packet number limit per node	1000000
Emulation clock limits	25000000	Injection ratios	2:50:2

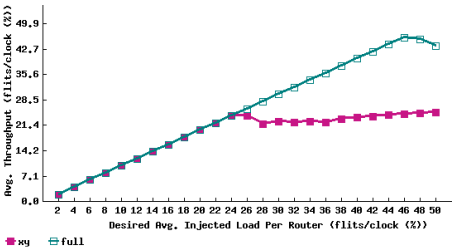
15. Click on  to add the second NoC emulation model. Fill the NoC configuration as shown in following table.

Parameter name	Value	Parameter Name	Value
FPGA Board	[Your FPGA board name]	Sram Object file	”mesh_5x5_full”
Configuration Name	fully	Traffic name	transposed 2
Packet size in flits	5	Packet number limit per node	1000000
Emulation clock limits	25000000	Injection ratios	2:50:2

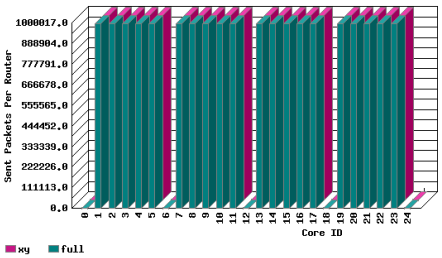
16. Save the emulation. You can save the emulation at any time during run time. Hence, later you can continue the rest of emulation.
17. To start the emulation, Power on your FPGA board and connect it to your PC then press  `Run all` button. You can also run each individual emulation by pressing the  `Run` button in its emulation row.
18. After the emulation is done, if your graph is not yet completed you can enter a new injection ratio range and press the  `Run` key again.
19. The emulator generates similar results as NoC simulator generates.



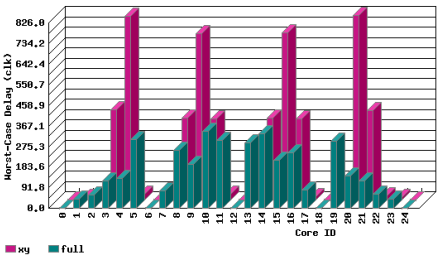
(a) Load-latency



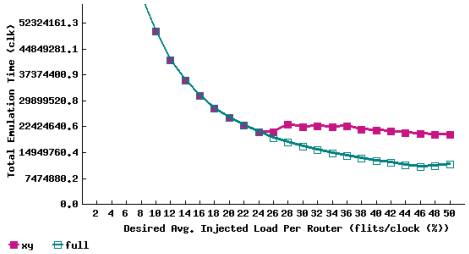
(b) Load-throughput



(c) Injected packets per router at 50% injection ratio.



(d) Worst-case delay per router at 50% injection ratio.



(e) Emulation time in clock cycles.

Figure 10.4: Emulator sample results.

CHAPTER 11

ProNoC Tools

JTAG UART

ProNoC support including several JTAG based Universal Asynchronous Receiver-Transmitter (UART) in a SoC/MPSoC. A unique `JTAG_INDEX` should be assigned to each UART to avoid conflict.

UART Terminal

ProNoC includes an in-built GUI for UARTs terminal. To run the UART terminal run ProNoC and press `Ctrl+U`. You can also directly run this tool by running following commands in in linux terminal.

```
cd mpsoc/perl_gui/lib/perl
perl uart.pl
```

Figure 11.1 illustrates the snapshot of the UART GUI.The following settings are required in order to connect to UART modules:

1. `UART name`: Select one of `ProNoC_XILINX_UART` or `ProNoC_AlTERA_UART` according to your FPGA device.
2. Define the total number of UART modules in your SoC/MPSoC design. Each UART module will have its own output window on the left side of the main window.
3. For each UART module set the corresponding `JTAG_INDEX`. This value is given as an input parameter to each UART module. The default value for each UART is `126`–CORE_ID. In case you have left the default values for an MPSoC where each of its tile has its own UART module, these indexes are 126,125,124 and so on.
4. You need to set the FPGA device configuration on JTAG chain now. Click on `browse` button will guide you about this task.
   (a) For Xilinx FPGAs you need to set the JTAG TAP chain number. This parameter has been passed to the UART module as global parameter. The default value is 3.
   (b) For Xilinx FPGAs you need to set the FPGA device target number in the JTAG chain. Figure 11.1 shows an example for Digilent Arty-Z7 XILINX FPGA board. The FPGA device is xc7z020 which is the 3^{rd} target in jtag chain.
   (c) For Altera FPGAs (see Figure 11.2) set the `Hardware name`.
   (d) For Altera FPGAs you need to set the FPGA device target number in the JTAG chain. Figure 11.2 shows an example for DE10-Nano Altera FPGA board. The FPGA device is 5CSEBA6U23I7 which is the 2^{nd} target in jtag chain.
5. you can now connect/disconnect the JTAG UART terminal by pressing clicking on `OFF`/`ON` button.

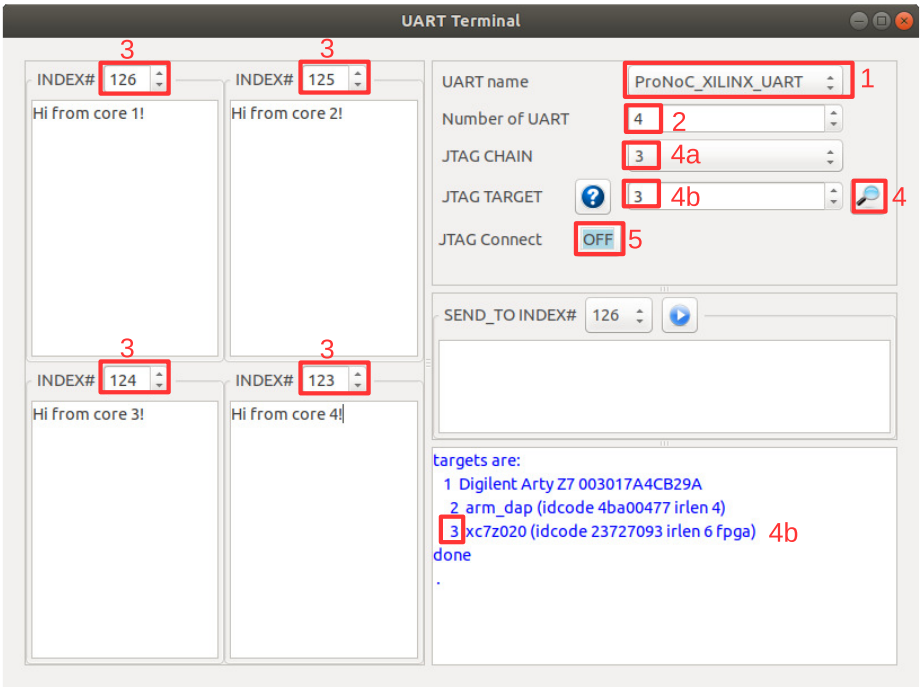


Figure 11.1: Uart terminal snapshot (Xilinx FPGA configuration).

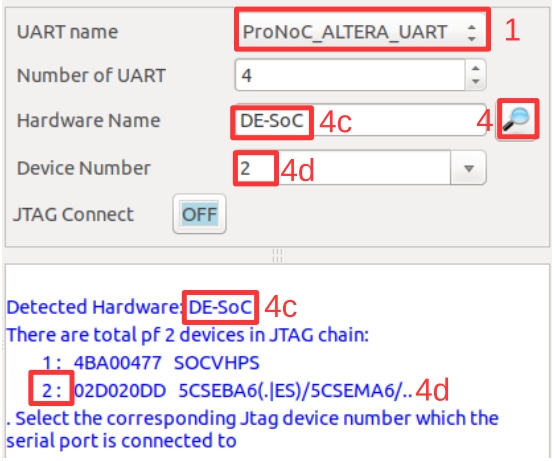


Figure 11.2: Altera FPGA configuration setting.

For Xilinx FPGAs you can also use `mpsoc/src_c/jtag/uart_xsct_terminal/uart` to monitor UART output ports on linux terminal.Remember you need to add the path to `xilinx/SDK/bin` to your PATH variable first. Run `./uart` without any option to see the usage info. Following is the example command for monitoring Digilent Arty-Z7 XILINX FPGA with four UART modules:

```
./uart -a 3 -b 36 -t 3 -n 126,125,124,123
```

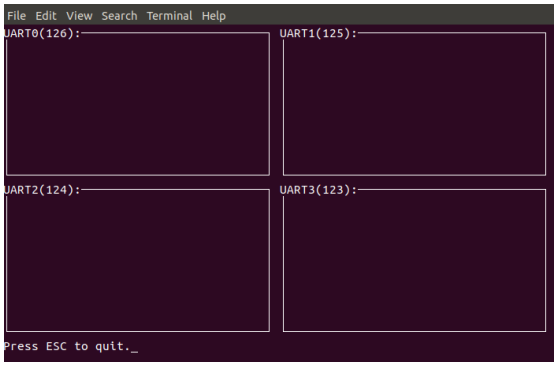


Figure 11.3: Linux terminal-based UART terminal.

Add new ALtera FPGA Board

On ProNoC GUI window click on `Tools` then select `Add new ALtera FPGA Board`. A new window as shown in Figure 11.4 must appear. Fill the required fields as follows:

1. Enter your board name. Do not use any space in the given name
2. Enter the path to FPGA board QSF file. In your Altera board installation CD or in the Internet search for a QSF file containing your FPGA device name with other necessary global project setting including the pin assignments (e.g `DE10_Nano_golden_top.qsf`).
3. Enter the path to [FPGA_board_top].v file. In your Altera board installation CD or in the Internet search for a Verilog file containing all your FPGA device IO ports (e.g `DE10_Nano_golden_top.v`).
4. Power on your FPGA board and connect it to your PC then press the `Auto Fill` button to auto-fill the JTAG configuration setting.
5. Press the `Add` button.

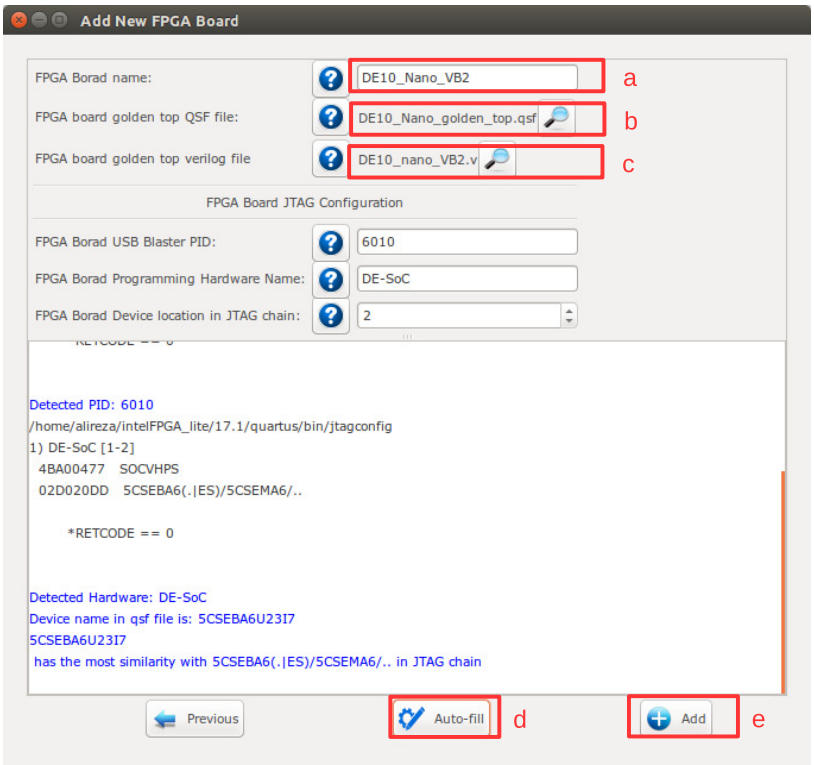


Figure 11.4: Add new FPGA board to ProNoC.

Add new Xilinx FPGA Board

On ProNoC GUI window click on `Tools` then select `Add new Xilinx FPGA Board`. A new window as shown in Figure 11.5 must appear. Fill the required fields as follows:

1. FPGA board display name: Enter a name for your FPGA Board. Do not use any space in the given name.
2. Set the path to Vivado board files repository.
   E.g. for ArtyZ7 FPGA board you can download its corresponding repo from https://github.com/Digilent/vivado-boards and save in `${ProNoC_work}/toolchain/board_files` folder.
3. FPGA board part name: Your Board name (Board PART). You can click on its adjacent to get the list of all available boards in your Vivado software.
4. FPGA part name: Enter your FPGA device name (PART). If you have selected Board PART in last step you can click on its adjacent to get this parameter.
5. FPGA Hardware device name: The target hardware name in JTAG chain. Connect your FPGA board to your PC and click on its adjacent to get the list of all available targets in your FPGA board.
6. Target device JTAG chain order number:The order number of target device in the jtag chain.Connect your FPGA board to your PC and click on its adjacent to get this value.
7. FPGA board xdc file: Path to FPGA board xdc file. In your Xilinx board installation CD or on the Internet, search for an xdc file containing your FPGA device pin assignment constrain.
8. FPGA board golden top Verilog file: (Path to FPGA_board_top.v file) A Verilog file containing all your FPGA device IO ports.

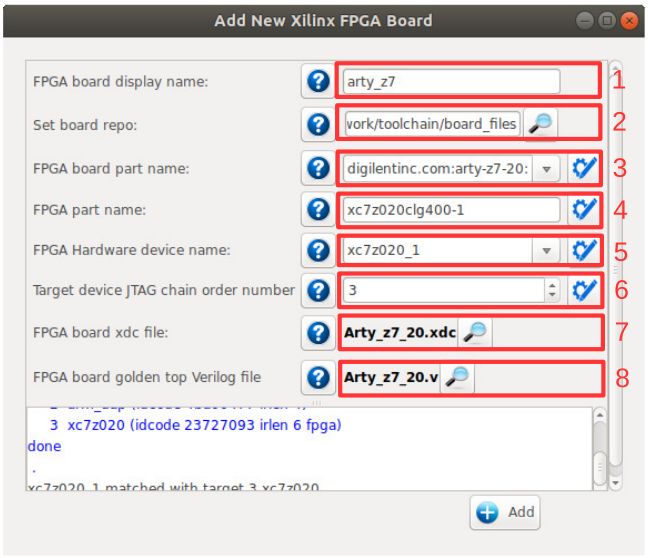


Figure 11.5: Add new Xilinx FPGA board to ProNoC.

Appendices

APPENDIX A

NoC Verilog File Parameters Description

V	$V \in \mathbb{N}, V \geqslant 1$.	Number of VC per router port. Defining V as 1 results in a simple non-VC based router.
B	$B \in \mathbb{N}, B \geqslant 2$	Buffer size per VC in flit for ports connected to other routers.
LB	$LB \in \mathbb{N}, LB \geqslant 2$	Buffer size per VC in flit for ports connected to endpoints (local ports).
PCK_TYPE	"MULTI_FLIT", "SINGLE_FLIT"	Refer to Packet type for more information.
TOPOLOGY	"MESH" "TORUS" "RING" "LINE" "FATTREE" "TREE" "CUSTOM"	The NoC topology.
T1,T2,T3,T4	$T1, T2, T3, T4 \in \mathbb{N}$	A desired topology can be defined using at most four parameters: e.g: in mesh: • T1: NX, number of node in x dimension. • T2: NY, number of node in y dimension. • T3: NL: number of individual router local ports • T4: is not used. e.g: in Tree, Fattree: • T1: K, umber of last level individual router's endpoints. • T2:L layer number. • T2,T3 are not used.
ROUTE_NAME	"XY", "DUATO", "WEST_FIRST", "NORTH_LAST", "NEGETIVE_FIRST", "ODD_EVEN"	NoC routing algorithm for mesh topology. "XY" is deterministic routing (DoR), "DUATO" is fully adaptive and the rest are partially adaptive routing algorithms.
	"TRANC_XY", "TRANC_DUATO", "TRANC_WEST_FIRST", "TRANC_NORTH_LAST", "TRANC_NEGETIVE_FIRST" "TRANC_ODD_EVEN"	NoC routing algorithm for torus topology. See [rahmati:2012] for more information.

	"NCA_RND_UP", "NCA_DST_UP", "NCA_STRAIGHT_UP",	NoC routing algorithm for Fatree topology. Nearest common ancestor (NCA) where the up port is selected randomly (RND), based on destination endpoint address (DST) or it is the top port that is located in front of the the port which has received the packet (STRAIGHT).
C	$C \in \mathbb{N}$	The number of message classes. Packets that belong to different message classes can have access to a different subset of VCs. The subset of VCs for each class is defined using CLASS_SETTING parameter.
Fpay	$Fpay \in \mathbb{N}$, $Fpay \geqslant 32$	Flit payload size in bit.
MUX_TYPE	"BINARY", "ONE_HOT"	Crossbar's multiplexer type in a NoC router. Binary and one-hot multiplexers are preferable for FPGA and ASIC implementation, respectively.
VC_REALLOCATION_TYPE	"ATOMIC", "NONATOMIC"	"ATOMIC": only an empty output VC can be reallocated for a new header flit. "NONATOMIC": A VC can be reallocated when it has received the tail flit of the last packet and has at least one empty buffer space. See [monemi:2016a] for more information.
COMBINATION_TYPE	"COMB_NONSPEC", "COMB_SPEC1", "COMB_SPEC2"	VC/SW combination type. None-Speculative or Speculative VC/SW combination.
FIRST_ARBITER_EXT_P_EN	0, 1	If it is set as 0, then the first level arbiters' priority registers in switch allocator are updated whenever any request is granted at first level otherwise the priority registers are updated only if they also receive the second level arbitration grants.
BYTE_EN	0,1	0: Disabled 1: Enabled. Adds a byte enable (BE) filed to header flit which shows the location of last valid byte in tail flit. It is needed once the sent data unit is smaller than Fpay.
CONGESTION_INDEX	$CONGESTION\_INDEX \in \mathbb{N}$, $0 \leqslant CONGESTION\_INDEX \leqslant 7$	Define how congestion metrics is selected. See Table A.2 for more information.

DEBUG_EN	0,1	If is defined as 1, the simulation will be run using extra debugging codes. The debugger dose several faults detection such as out of order flits receiving, packet miss-routing and VC status miss-matching.
ADD_PIPREG_AFTER_CROSSBAR	0,1	If is defined as 1, a pipeline register will be added after the crossbar switch which add one clock cycle latency for link traversal stage. It may be needed for ASIC NoC where routers are connected using long wires. However, in FPGA implementation it may not be required.
CLASS_SETTING	{V'bX,...,V'bX}	It defines how each message class can have access to VCs. For each class a V-bit access-VC value is defined in such a way that each asserted bit represents the VC which this message class can request for. The CLASS_SETTING is concatenate of all message class access-VC values.
ESCAP_VC_MASK	V'bX	It is a V-bit value and its asserted bit(s) represent the escape VC(s) (EVC). It is valid only for fully adaptive routing. You must make sure that each message class have access to at least one EVC to prevent deadlock in fully adaptive routing.
SSA_EN	"YES" , "NO"	If set as "YES", packets which are traveling to the same dimension bypass router pipeline stages using Static straight allocator. See [monemi:2016b] for more information.
SMART_MAX	SMART_MAX $\in \mathbb{N}$, SMART_MAX $\geqslant 0$	If Max Straight Bypass (SMART_MAX) is defined as $n > 0$ then packets are allowed to bypass Maximum of n routers in straight direction in single cycle. See [monemi:2021] for more information.

SWA_ARBITER_TYPE	"RRA", "WRRA"	Switch allocator's output ports arbiters type: RRA: Round Robin Arbiter. Provides only local fairness in a router. WRRA: Weighted Round Robin Arbiter. Results in global fairness in the NoC. Using WRRA the switch allocation requests are granted according to their weights which increases dynamically due to contention.Refer to [monemi:2020] for more information.
WEIGHTw	$\texttt{WEIGHTw} \in \mathbb{N}$, $2 \leqslant \texttt{WEIGHTw} \leqslant 7$	WRRA weights' maximum width in bits.
MIN_PCK_SIZE	$\texttt{MIN\_PCK\_SIZE} \in \mathbb{N}$, $\texttt{MIN\_PCK\_SIZE} \geqslant 1$	The minimum packet size in flits. In atomic VC re-allocation, it is just important to define if the single-flit sized packets are allowed to be injected to the NoC by defining this parameter value as one. Setting any larger value than one results in the same architecture and the NoC works correctly even if it receives smaller packets size as while as they are not single-flit sized packets. However, for non-atomic VC reallocation NoCs, you have to define the exact value as it defines the NoC control registers' internal buffers. The NoC may crash once it receives packets having smaller size than the defined minimum packet size.
SELF_LOOP_EN	"YES" , "NO"	If the self-loop is enabled, it allows a router input port sends packets to its own output port. Enabling it allows a tile to be able to send packets to itself too.;

CAST_TYPE	"UNICAST", "MULTICAST_PARTIAL", "MULTICAST_FULL", "BROADCAST_PARTIAL", "BROADCAST_FULL"	Configure a NoC as Unicast, Multicast or broadcast NoC. In Unicast NoC, a packet can be send to only one destination. In Multicast, a single packet can have multiple target destination nodes. The broadcast packets are sent to all other destination nodes. For Multicast and broadcast NoC only one single copy of a packet is needed to be injected to the source router, the routers then fork the packet to different output ports when it is necessary. Multicast and Broadcast can be selected as FULL, where all destinations can be included in packet destination list or PARTIAL where a user defined subset of nodes can be targeted in Muticasting. Note that the Multicast/broadcast packets size should be equal or smaller than router input port buffer size.
MCAST_ENDP_LIST	NE'bX	MCAST_ENDP_LIST is an NE bit one-hot coded number where NE is the total number of emdpoints and the asserted bit indicates that the corresponding destination ID can be targeted in multicast/broadcast packets. The corresponding destinations with zero bit can only receive unicast packets. This parameter is only valid for "BROADCAST_PARTIAL" and "MULTICAST_PARTIAL" parameters.

Table A.2: Congestion metrics.

Index	Description	pin overhead
0	Number of unavailable VCs in the neighboring router adjacent input port.	-
1	Number of consumed credit in all VCs of the neighboring router adjacent input port.	-
2	Number of active switch allocation requests in all ports of the neighboring router.	2-bit
3	Number of active switch allocation requests in all ports of the neighboring router.	3-bit
4	Number of active switch allocation requests in all ports of the neighboring router that are not granted.	2-bit
5	Number of active switch allocation requests in all ports of the neighboring router that are not granted.	3-bit
6	Number of unavailable VC in all ports of the neighboring router	2-bit
7	Number of unavailable VC in all ports of the neighboring router	3-bit

APPENDIX B

NoC Verilog File Signals Description

Resource allocation units

A **packet** is the unit of data that is routed between a source and a destination cores. Packets contain control signals such as routing data, destination address, message classes in addition to the data. **Flits** (flow control units) are the atomic units that form packets. A packet consists of three types of flit; it starts with a header flit, followed by an optional number of body flits, and ends with a tail flit. **Phit** (physical unit), is the smallest unit of data transmitted in a single cycle on a communication link. In ProNoC flits are single phit size. The NoC resource allocation units is shown in Figure B.1.

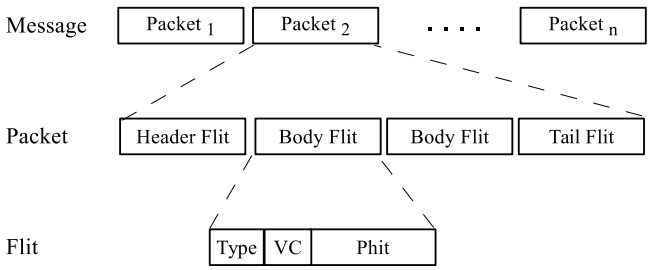


Figure B.1: Units of resource allocation in NoC.

Flit type

Type is a 2-bit signal indicates the type of its respective flit (header,body or tail). The first bit is asserted for header flits. The second flit is asserted if the flit is a tail. For body flits neither of these two bits are asserted. For single flit packets both of type bits are asserted.

VC filed

VC filed is coded in one-hot format and the asserted bit indicates the packet VC number.

Packet type

ProNoC supports two types of single or multi flit packet format. The packet format is defined by setting `PCK_TYPE` parameter to one of `SINGLE_FLIT` or `MULTI_FLIT` values.

single-flit

: In this configuration all packets injected to the NoC must consist of only single-flit. The packet control fields are added to packet Data filed as shown in Figure B.2.

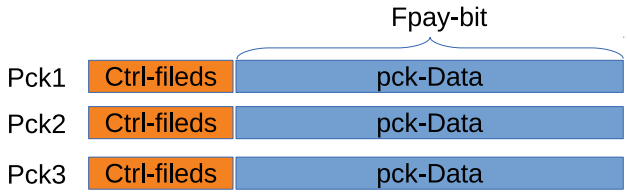


Figure B.2: Single-flit type packet format.

Multi-flit

: In multi-flit format a packet can consist any arbitrary number of flits. In this format the header flit carries the control fields as part of its payload filed as shown if Figure B.3.

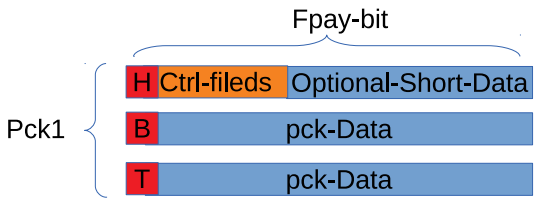


Figure B.3: Muti-flit type packet format.

Control fields format

The header flit carries some necessarily information which is required by the flow control. The header flit format varies depending on ho NoC parameter are selected.

Bit			0
	destport	dst-endpoint-addr	src-endpoint-addr
Size (bits)	DSTPw	EAw	EAw
	Fpay		
	header-data	weight	class
Size (bits)	HDw	Ww	Cw

Endpoint addressing format

ProNoC encodes the endpoint addresses according to the topology parameters:

Line,Ring : {L,X}
X: connected router index number.
L: index of router local port connected to the endpoint node.

Mesh,Torus : {L,Y,X}
X: Index of the first dimension (column) of the connected router.
Y: index of the second dimension (row) of the connected router.
L: index of a router local port connected to the endpoint node.

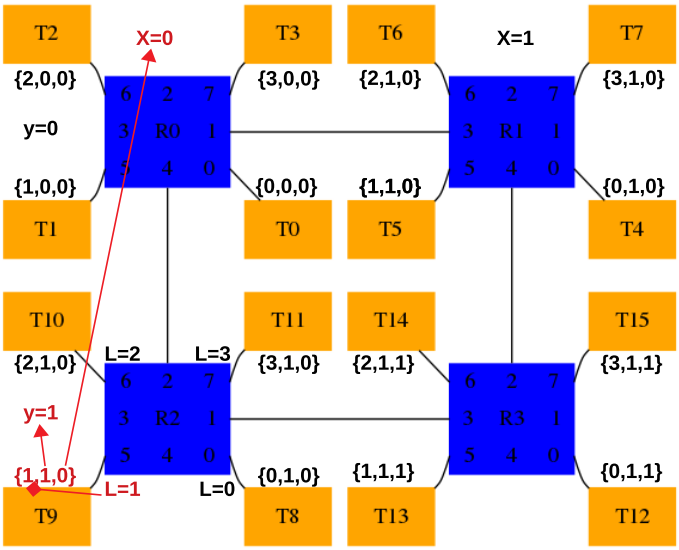


Figure B.4: 2×2 CMESH endpoint coding example.

The endpoint address size in Line,Ring,Mesh and, Torus is obtained using the following equations:

$$\begin{aligned} NXw &= \log 2NX; \\ NYw &= (NY > 1)?\ \log 2NY : 0; \\ NLw &= (NL > 1)?\ \log 2NL : 0; \\ EAw &= NLw + NYw + NXw; \end{aligned} \tag{B.1}$$

where NX is the maximum number of routers in first dimension, NY is maximum number of router in second dimension,NL is maximum number of individual router's local port, NXw is router first dimension size in bits, NYw is router second dimension size in bits, NLw is router local port index width in bits, and EAw is the endpoint address width.

In Fatree/tree each individual endpoint address is coded based on port number of parent routers: $\{P_0,P_1,...,P_{k-1}\}$
where K is the height of tree and P_n is the router's bellow port number located at the n^{th} layer which can receives a packet from that endpoint (note that n_{root}=0). Figure B.5 shows an example of endpoint encoding in a fattree (k=3,n=3). As an example a packet which sent from T_7 to any of root nodes will always received from port number 1,2,0 in each layer respectively.

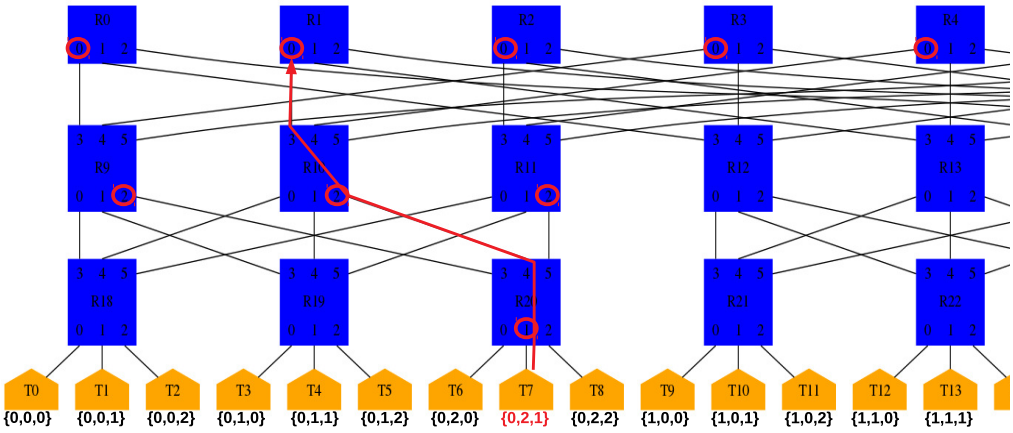


Figure B.5: Fattree (k=3,l=3) endpoint coding example.

$$Kw = \log 2K$$
$$EAw = L \times Kw \tag{B.2}$$

where L is length of three and K is the number of endpoints connected to each individual router in last tree level.

Once you press `Generate RTL` button in NoC-based MPSoC using `NoC based MPSoC generator`, the endpoint addresses are generated in `[PRONOC_WORK]/mpsoc/[PT-name]/sw/phy_addr.h` file.

phy_addr.h contains 2×2 CMESH endpoint adresses.

```
#ifndef PHY_ADDR_H
   #define PHY_ADDR_H

   #define PHY_ADDR_ENDP_0 0x0
   #define PHY_ADDR_ENDP_1 0x4
   #define PHY_ADDR_ENDP_2 0x8
   #define PHY_ADDR_ENDP_3 0xc
   #define PHY_ADDR_ENDP_4 0x1
   #define PHY_ADDR_ENDP_5 0x5
   #define PHY_ADDR_ENDP_6 0x9
   #define PHY_ADDR_ENDP_7 0xd
   #define PHY_ADDR_ENDP_8 0x2
   #define PHY_ADDR_ENDP_9 0x6
   #define PHY_ADDR_ENDP_10 0xa
   #define PHY_ADDR_ENDP_11 0xe
   #define PHY_ADDR_ENDP_12 0x3
   #define PHY_ADDR_ENDP_13 0x7
   #define PHY_ADDR_ENDP_14 0xb
   #define PHY_ADDR_ENDP_15 0xf
```

```
#endif
```

destport ProNoC routers use look-ahead routing algorithm i.e. the destination port is calculated one router ahead and the result is attached to the header flit in *destport* filed. The *destport* format varies dependent on topology and routing algorithm. Once a packet is injected into the NoC this field is automatically updated by each router at packet departure time. However, it is the duty of endpoints to update the *destport* filed of packets which are injected to the router's local ports. Endpoints are supposed to use `ni_conventional_routing` Verilog module to obtain the *destport* filed. The size of *destport* filed is shown in Table B.1.

Table B.1: DSTPw for different typologies. Note that K is the tree height.

TOPOLOGY	RING, LINE	MESH, TORUS	FATTREE	TREE
DSTPw (bits)	2	4	K+1	log2(K+1)

class This filed indicates the message class binary number. Each specific class can use different set of VCs. The permitted VCs which can be used by each individual class is given to the RTL code using `CLASS_SETTING` Verilog parameter.

$$Cw = (C > 0)?\ \ \log 2C : 0 \tag{B.3}$$

where C is number of defined class and Cw is class width filed in bits.

weight This filed carries packets weight which increases dynamically inside the NoC at presence of congestion. weight filed is only valid once the router is configured with weighted round robin arbitration.

$$Ww = (\texttt{SWA\_ARBITER\_TYPE} == \texttt{"WRAA"})?\ \texttt{WEIGHTw} : 0 \tag{B.4}$$

where C is number of defined class and Cw is class width filed in bits.

header-data The header flit can optionally carries some data. The size of data which a header flit can carries. The number of data bits which a header flit can carry (HDw):

$$HDw = Fpay - 2EAw - DSPTw - Ww \tag{B.5}$$

APPENDIX C

Multiple physical NoCs with different configurations

In ProNoC the NoC configuration parameters are defined in `noc_localparam.v` file in `mpsoc/rtl/src_noc/` path. This file is then used inside `pronoc_pkg.sv` file to define NoC structs, channels, and interfaces. In SystemVerilog it is not possible to override a parameter inside a package, so in case you needed to have multiple NoCs with different parameters in a system you may face a challenge.

Solution 1: Compiling Each NoC as a Separate Library

A practical workaround to address this issue is to compile each NoC as a separate library. Below are the steps required for Modelsim simulation. For this example, we assume there are three different NoCs (`NoC1`, `NoC2`, `NoC3`), where each NoC has a different flit payload width (512, 72, and 64 bits).

1. Modify `noc_localparam.v`
   Update the `noc_localparam.v` file as shown below. The `USE_LIB` macro is passed during compilation time and determines the flit payload width.

```
//NoCs' common parameters definition
   localparam V=1;
   localparam B=16;
   localparam PCK_TYPE="SINGLE_FLIT";
   localparam MIN_PCK_SIZE=1;
   localparam BYTE_EN=0;
   localparam SSA_EN="NO";
   localparam SMART_MAX=0;
// Rest of parameters ...

//NoC's unique parameter definition
   `define STRINGIFY(x) `"x`"
   `ifdef USE_LIB
     localparam LIB_STR =`STRINGIFY(`USE_LIB);
     localparam Fpay = (LIB_STR =="noc1_rtl_work")? 512 :
       (LIB_STR =="noc2_rtl_work")? 72 : 64 ;
   `else
     localparam Fpay=32 ;
   `endif
```

2. Create a Top Module for Each NoC
   For each NoC, create a dedicated top module in a separate file (`noc1.sv`, `noc2.sv`, `noc3.sv`) and instantiate the `noc_top` module within them. Ensure you include the `pronoc_def.v` file.

```
`include "pronoc_def.v"
module noc1
   import pronoc_pkg::*; // need `USELIB noc1_rtl_work
   (
   input clk,
   input reset,
   input smartflit_chanel_t pronoc_chan_in [NE-1 : 0],
   output smartflit_chanel_t pronoc_chan_out [NE-1 : 0]
  );

   noc_top the_noc1 (
      .reset(reset),
      .clk(clk),
```

```
      .chan_in_all (pronoc_chan_in),
      .chan_out_all(pronoc_chan_out));
endmodule
```

3. Prepare a Compilation Script
   Write a compilation script to compile `noc1.sv`, `noc2.sv`, and `noc3.sv` with different `USE_LIB` values. This will generate the desired NoCs.

```
#!/usr/bin/tclsh
set comp_path  [path_to_work_dir]

file mkdir $comp_path
cd $comp_path

set noc1_rtl_work $comp_path/noc1_rtl_work
set noc2_rtl_work $comp_path/noc2_rtl_work
set noc3_rtl_work $comp_path/noc3_rtl_work
set work_path $comp_path/work

if {[file exists $noc1_rtl_work]} {vdel -lib $noc1_rtl_work -all}
if {[file exists $noc2_rtl_work]} {vdel -lib $noc2_rtl_work -all}
if {[file exists $noc3_rtl_work]} {vdel -lib $noc3_rtl_work -all}
if {[file exists $work_path]} {vdel -lib $work_path -all}

vlib work

vlib $noc1_rtl_work
vlog -compile_uselibs -libmap_verbose +acc=rn -work noc1_rtl_work +
    define+USE_LIB=noc1_rtl_work -F noc_flist.f noc1.sv
vmap noc1_rtl_work $noc1_rtl_work

vlib $noc2_rtl_work
vlog -compile_uselibs -libmap_verbose +acc=rn -work noc2_rtl_work +
    define+USE_LIB=noc2_rtl_work -F noc_flist.f noc2.sv
vmap noc2_rtl_work $noc2_rtl_work

vlib $noc3_rtl_work
vlog -compile_uselibs -libmap_verbose +acc=rn -work noc3_rtl_work +
    define+USE_LIB=noc3_rtl_work -F noc_flist.f noc3.sv
vmap noc3_rtl_work $noc3_rtl_work

vlog +acc=rn -compile_uselibs -libmap_verbose -work work -L
    noc1_rtl_work -L noc2_rtl_work -L noc3_rtl_work testbench.sv

vsim -t ps -L noc1_rtl_work -L noc2_rtl_work -L noc3_rtl_work
    work.testbench

run 100 ms

quit
```

Solution 2: Avoiding `import pronoc_pkg.sv`

An alternative solution is to avoid importing the `pronoc_pkg.sv` file and instead include it directly in each module. The steps to implement this are as follows:

1. Modify `pronoc_def.v`
   Open the `mpsoc/rtl/pronoc_def.v` file in a text editor and remove or comment out the line containing `` `define IMPORT_PRONOC_PCK ``.
2. Update `noc_localparam.v`
   Modify the `noc_localparam.v` file to redefine the NoC local parameters based on the `NOC_ID` value. See the example below:

```
//NoCs' common parameters definition
   localparam V=1;
   localparam B=16;
   localparam PCK_TYPE="SINGLE_FLIT";
   localparam MIN_PCK_SIZE=1;
   localparam BYTE_EN=0;
   localparam SSA_EN="NO";
   localparam SMART_MAX=0;
// Rest of parameters ...

//NoC's unique parameter definition
   localparam Fpay =  (NOC_ID == 0 ) ? 512 :
                      (NOC_ID == 1 ) ? 72 : 64;
```

3. Use the `NOC_ID` Parameter
   You can now instantiate multiple physical NoCs by passing different values to the `NOC_ID` parameter in the `noc_top` module. For an example implementation, refer to the following:

```
   noc_top #(
      NOC_ID(0)
   )noc0(
      .reset(reset),
      .clk(clk),
      .chan_in_all  (chan_in0),
      .chan_out_all(chan_out0)
   );

   noc_top #(
      NOC_ID(1)
   )noc1(
      .reset(reset),
      .clk(clk),
      .chan_in_all  (chan_in1),
      .chan_out_all(chan_out1)
   );
```

Solution 3: Generating a Unique Physical NoC RTL Code

The following script generates a unique Physical NoC RTL code. This approach enables the creation of multiple Physical NoCs, each with a distinct package, parameters, and RTL modules. The `NoC_ID` is appended to parameters, functions, and module names to ensure uniqueness across the designs.

```
perl mpsoc/script/phy_noc_gen/phy_noc.pl [NoC_ID] [TARGET_DIR]
```