ProNoC

User Manual

Copyright ©2014–2019 Alireza Monemi

Contents

CHAPTER 1

Installation Manual for the Ubuntu Linux Environment

Installation

1. 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
```

Copy the downloaded folder (`trunk/`) somewhere in your home directory. Make sure that there is **no space** in selected path.

2. To give execute permission, open `trunk/mpsoc` in terminal and run

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

3. Install required package dependencies

```
sudo apt-get install build-essential
sudo apt-get install libgtk2.0-dev libglib2.0-dev
sudo apt-get install libpango1.0-dev
sudo apt-get install clang
sudo apt-get install lib32z1
sudo apt-get install libgd-graph-perl
sudo apt-get install cpanminus
sudo apt-get install libusb-1.0
sudo apt-get install graphviz
sudo apt-get install libgtksourceview2.0-dev
sudo apt-get install libcanberra-gtk-module
```

4. Install required Perl modules:

```
sudo cpanm ExtUtils::Depends
sudo cpanm ExtUtils::PkgConfig
sudo cpanm Glib
sudo cpanm Pango
sudo cpanm Gtk2
sudo cpanm String::Similarity
sudo cpanm Gtk2::Ex::Graph::GD
sudo cpanm GD::Graph::bars3d
sudo cpanm IO::CaptureOutput
sudo cpanm Proc::Background
sudo cpanm List::MoreUtils
sudo cpanm File::Find::Rule
sudo cpanm Gtk2::SourceView2
sudo cpanm Verilog::EditFiles
```

5. Install Verilator simulator.

```
sudo apt-get install verilator
sudo cpanm install Verilog::Language
```

6. Download soft-core processors' GNU toolchain:

   (a) aeMB

(b) [Lm32](#) or from [Lm32](#)

(c) [or1k-elf](#) for mor1k and or1200 OpenRISC CPUs.

Unzip the files and copy the `lm32`, `or1k-elf`, and `aemb` folders in `trunk/mpsoc_work/toolchain` directory.

7. Give execution permission to GNU toolchains. Open terminal in `mpsoc_work/toolchain` and run

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

8. Open `/mpsoc/src_c` in terminal and run

```
make
```

9. Now you can run the GUI application by

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

10. If it is the first time you are running the ProNoC software, it asks you to set the following paths:

(a) **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 compulsory.

(b) **QUARTUS_BIN**: The path to QuartusII compiler bin directory. Setting of this variable is optional and is needed if you are going to use Altera FPGAs for implementation or emulation.

(c) **MODELSIM_BIN**: The path to Modelsim simulator bin directory. Setting of this variable is optional and is needed if you have installed Modelsim simulator and you want ProNoC to auto-generate the simulation models using Modelsim software.

You can modify these variables at any time later via `File->setting` menu:
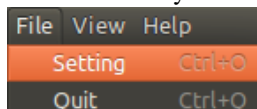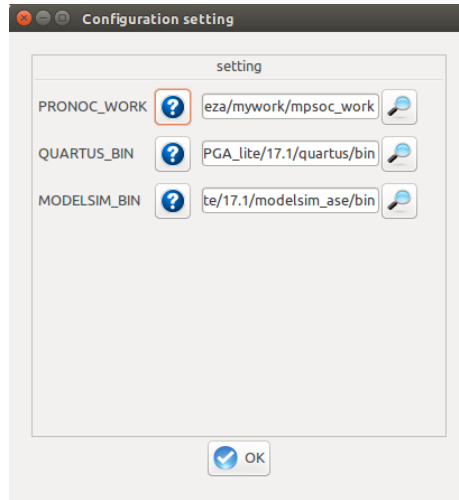
Figure 1.1: ProNoC path variables setting.
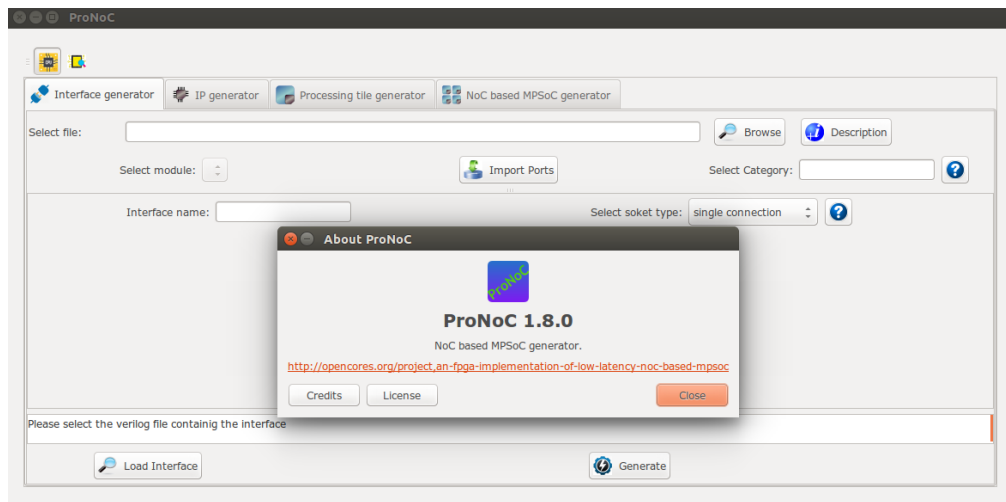
Figure 1.2: ProNoC GUI snapshot.

CHAPTER 2

Interface Generator

Introduction   The interface generator allows the addition of new interfaces to ProNoC software. An interface is a port or a group of ports that are common in different IP cores which are used for doing a specific task. The most common interfaces in ProNoC are the shared bus (wishbone bus) master/slave, clk and reset interfaces. Each individual interface is divided into two types of *socket* and *plug* interfaces. Two different IP cores can be connected when one has the *socket* type of an interface and another one has the *plug* type of that interface. While it is optional to select any side of the connection as *socket* or *plug* interface, bellow are some differences between them that help to select an appropriate type of interface for each IP core:

1. In processing tile generator only the *plug* interfaces of an IP are shown in the IP box. The user can select the connection interface from the list of all IP cores having the *socket* type of that interface as shown in Figure 2.1.
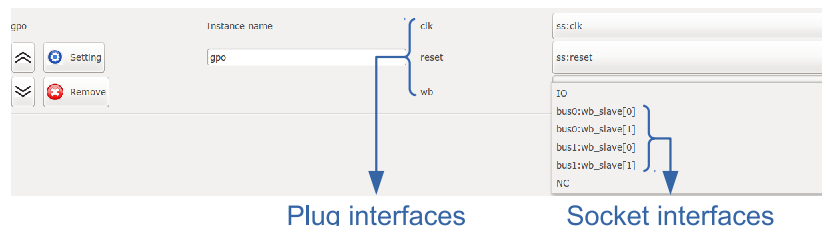
Figure 2.1: GPO IP box snapshot.

2. The *socket* interfaces can be defined as single or multi-connection. A socket interface can be defined as multi-connection only when it consists of only output ports. As a result, it can be connected to multiple IPs having the *plug* type of that interface. Examples of multi-connection *socket* in PoNoC are clk and reset interfaces.
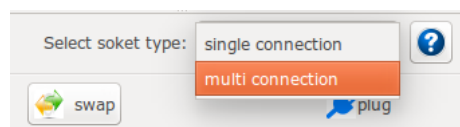
Figure 2.2: multi-connection selection snapshot.

3. The number of a *socket* interface in an IP core can be parameterizable. To do this, the interfaces' ports that having the same name must be concatenated as a single port in the IP core Verilog file. This feature provides flexibility to the ProNoC Processing tile generator as an IP core now can have variable number of an interface which can be defined by the user at the generation time. As an example the interfaces of the Wishbone bus and the interrupt controller are defined as *socket* with parameterizable number of interfaces. Below is an example which shows how the interfaces are defined in a Wishbone Bus IP core module:

Listing 2.1: bus.v

```verilog
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 ,
    .
    .
```
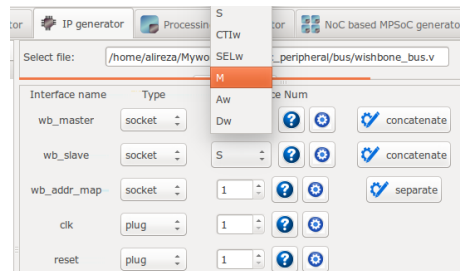

(a)
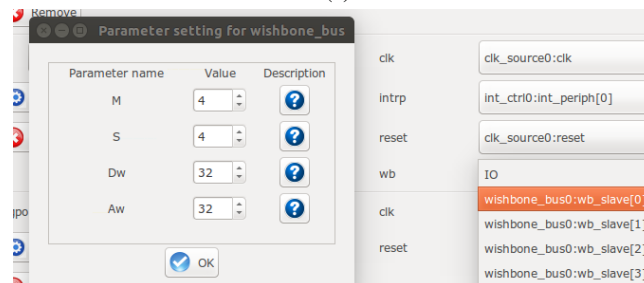
(b)

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

Generate New Interface

In order to add a new interface to ProNoC, press the `browse` button and select the Verilog file containing a module with the desired interface. If there are multiple modules inside that file, you can select the desired one from `Select module` menu. To add ports to the interface press `Import Ports` button. It opens a pop-up window as shown in Figure 2.4 where you can select and add the required ports.
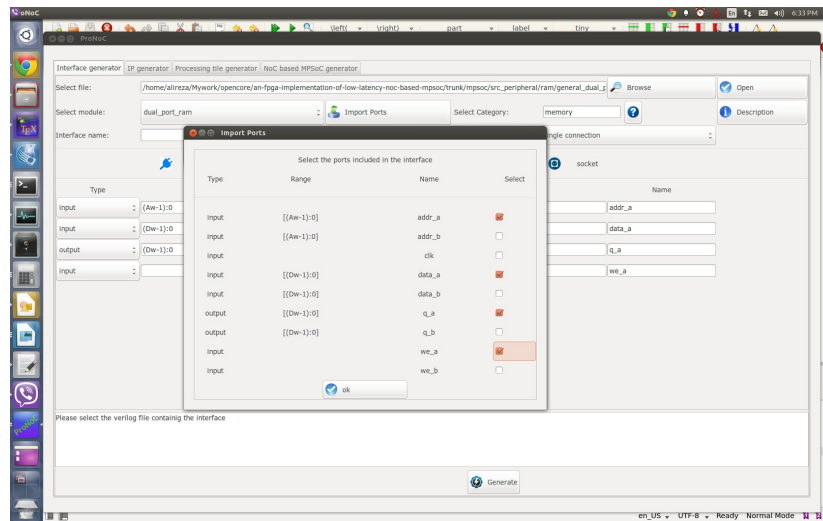
Figure 2.4: Interface generator snapshot.

Using `swap` button, you can define if the selected ports belong to the *socket* or *plug* type of an interface. You are only needed to define one type of an interface, the other type will be defined automatically. The width of each port can also be a Verilog code parameter. 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 or multi connection. If a socket is defined as single connection, by connecting a new IP to the socket, the last connected *plug* to that *socket* will be disconnected automatically.

Defined Interfaces

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

NI

This is the interface connection between Network-on-chip (NoC) router and the NoC interface adapter module (NI). Figure 2.5 shows this interface.
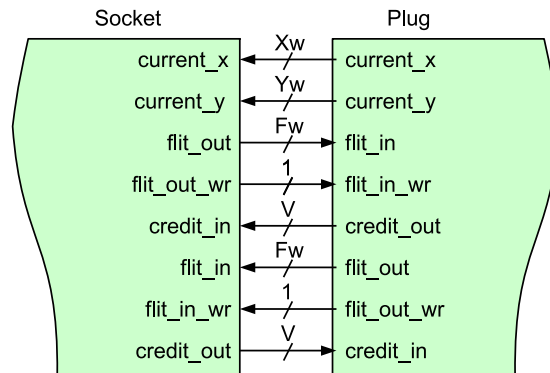
Figure 2.5: NI socket/plug interfaces.

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

interrupt_cpu   CPUs that have only one single interrupt pin must be connected to an interrupt controller module to allow combination of several sources of interrupt. The interface between these CPUs and Interrupt controller is called interrupt_cpu.
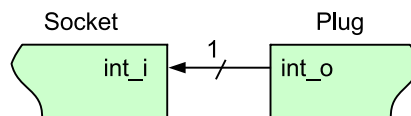
Figure 2.6: interrupt_cpu socket/plug interfaces.

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

interrupt _peripheral   This is the interrupt interface connection between CPUs having multiple interrupt pins that can directly be connected to multiple the peripheral devices.
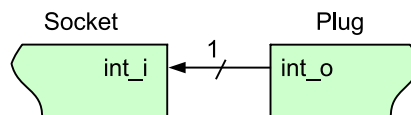
Figure 2.7: interrupt_peripheral socket/plug interfaces.

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.
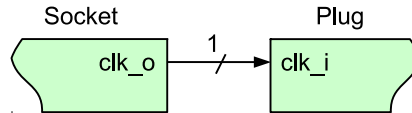
Figure 2.8: clk socket/plug interfaces.

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.
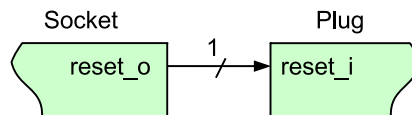
Figure 2.9: reset socket/plug interfaces.

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

Enable   The enable pin interface. The enable pin is used for disabling any active module in a processing tile (e.g CPUs). The Processing tile and NoC-based MCSoC generators automatically connect all enable plug interfaces to each other and used them for disabling CPUs during programming mode. The enable pin for each CPU must be defined as IO in processing tile generator.
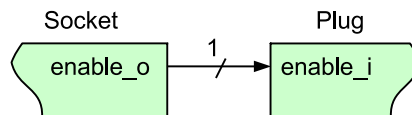
Figure 2.10: Enable socket/plug interfaces.

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

Wb_master   The wishbone bus master interface. The Wb_master socket interface is mapped to wishbone bus module. All IP cores' WB master interface must be mapped to the plug interface.
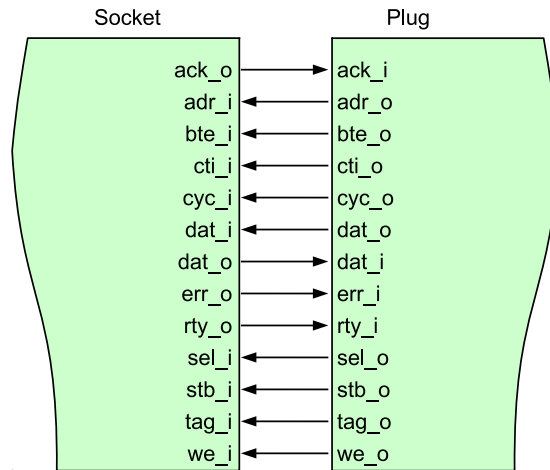
Figure 2.11: WB master socket/plug interfaces.

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 The wishbone bus slave interface. The Wb_slave socket interface is mapped to wishbone bus module. All IP cores' WB slave interface(s) 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
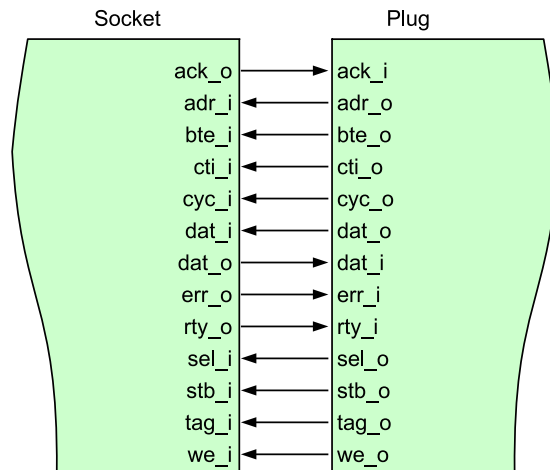
Figure 2.12: WB slave socket/plug interfaces.

CHAPTER 3
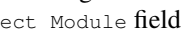
IP Generator
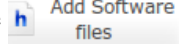
Introduction   The IP generator allows adding new intellectual properties (IPs) to the ProNoC's library. It provides a GUI interface for mapping the IP's ports to the interfaces, defining how the IP parameters must be collected from the user at tile generation time, and getting the location of IP cores' source files.

Generate a New IP   For adding a new IP to ProNoC, first you need to have the Verilog file(s) describing the RTL code of that IP.

1. Click on 🔍 `Browse` button and select the Verilog file containing the top level module.

2. Select a category which this new IP core is belonging to. You can eighther select it form the list of available categories or define a new category by typing its name in `Select Category:` `Bus` `▾`. All IPs belonging to the same category are listed under the same tree branch in processing tile generator.

3. Define an `IP name` for this module. The IP name will be shown in IP list below its category name in Processing tile generator.

4. In case the Verilog file contains several Verilog module select the top level module in `Select Module` field.

5. Using ⓘ `IP Description` button you can add a short description about the IP. This description will be shown when the IP is selected in processing tile generator. You can also add the IP-core documentation in PDF format here. This generate a short key for opening the IP documentation in processing tile generator.

   Note: In order to make the copy of your ProNoC software portable palace the documentation files somewhere inside `mpsoc` folder.

6. The **h** `Add Software files` button allows the addition of the necessarily files and folders to the generated processing tile software directory (`[PRONOC_WORK]/SOC/[PT -name]/sw`). By pressing this button you will have three notebook pages:

   - **Add existing files/folders**: In this page you can add the list of files and folders which you want to copy them exactly into the `mpsoc/SOC/[PT-name ]/sw` folder.

   - **Add files contain variables**: In this page you can add the list of files which contain some variables that can be replaced at the processing tile generation time. Variables must be written in the source file with `${variable_name}` format. You can use any of available variables in ProNoC as variable name.

   - **Add to tile.h**: You can add the definition and functions for this peripheral device here. These definitions are added to the processing tile header file at generation time. You can use any of available variables in ProNoC with `${variable_name}` format. A header file example 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)

void ${IP}_initial (unsigned int v) {
    ${IP}_WRITE_REG1(v);
}
```

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 `0
X96000000` by ProNoC automatically is 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)

  void foo_initial (unsigned int v) {
   foo_WRITE_REG1(v);
  }
```
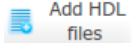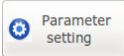
7. Add the list of all required designed HDL files for the new IP core by using
   Add HDL files button. All files listed here will be copied in the generated process-
   ing tile inside `mpsoc/SOC/[PT-name]/src_verilog` folder.

8. By pressing Parameter setting button, all parameters inside the top module Verilog
   file are extracted. This menu allows you to add, remove or define how to get the
   parameter values from the user. Below is an example for setting parameter **M** in
   wishbone bus.

Parameter name	Default value	Widget type	Widget content	Type		info	add/remove
M	4	Spin-button	1,256,1	Localparam	☑ Redefine		✖ remove

Figure 3.1: Parameter setting window snapshot.

- **Parameter name**: It is the parameter name which has been read from the
  Verilog file.

- **Default value**: when an IP is selected for the first time in processing tile generator, the parameters are loaded by their default values.

- **Widget type**: defines how the parameter value must be taken from the user when calling the IP in processing tile generator. There are four ways to define a widget type:

  - **Fixed**: The parameter is a fixed value and get the default value. User will not see the parameter and cannot change it in GUI.

  - **Entry**: The parameter value is received via `entry` widget. The user can type anything as parameter value.

  - **Combo-box**: The parameter value can be selected from a list of pre-defined values.

  - **Spin-box**: The parameter is a numeric value and is taken using `spin-box` widget.

- **Widget content**: For Fixed and Entry leave it empty. For Combo box define the parameters which must be shown in combo box. Use following format: `"VALUE1","VALUE2",...,"VALUEn"`. For Spin box define it with this format `minimum,maximum,step` (e.g 0,10,1).

- **Type**: Here you can define that how any specific IP-core parameter is defined in the generated processing tile Verilog file. You have three options `localparam`, `Parameter`, and `Don't include`. If you select it as `Parameter` then all processing tile parameters are also defined as parameter in the processing tile Verilog file. Hence, they can be changed during NoC-based MPSoC generation time. This allows calling same tile in different places with different parameter values. In case the parameter is a software parameter which must be used in software code variables define it as `Don't include`.

- **Redefine**: If it is check marked, the defined parameter/localparam in processing tile Verilog file will be passed to the IP core during instantiating. Remove the check mark if you only have added a parameter using parameter setting GUI which 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**: The parameter description for the user can be added here.

9. **Add interface**: You can add interfaces to the IP library by double clicking on an interface name located at the left top corner. After adding the interface, it appears in the interface box where you can adjust the interface setting such as,

interface name, type, and the number of that interface which appears in the new IP core.

For wishbone slave interface you can select the wishbone address setting by pressing ⚙ button and do the following settings:

- **Interface name**: define a name for this interface.
- **Address Range**: select the address range for WB slave port. These addresses are defined in `mpsoc/perl_gui/lib/perl/wb_addr.pm` file. You can add your own address range by modifying this file.
- **Block address width**: define the maximum memory size required for this interface in byte which is defined as 2 power of `block address width` (see Figure 3.2 caption as an example). The width can be defined as a fixed number when the number of memory mapped registers inside the interface is predefined as a fixed number. In case, that the number of required registers is dependent on a Verilog parameter (e.g. a memory block that its size is parameterizable) and it is aimed to be defined by the user at processing tile generation time then you can define it as `parametrizable` then select the corresponding parameter as address width.

Figure 3.2: Slave WB address setting snapshot. The size of memory mapped registers in this example is $2^5 = 32$ bytes. For a 32-width WB it is equal to $32/4 = 8$ individual registers. In case, you have parameterizable number (e.g. **M**) to indicate memory mapped register width in words in your IP module Verilog file, you need to add another parameter such as **N=M+2** in `parameter setting` window and select its type as `Don't include` to be used as address width parameter in bytes.

For socket interfaces, there is an option to define the interface number as parameter by selecting ✔ `concatenate` condition or a fixed number by selecting ✔ `separate` condition. See socket interface specification for more information.

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

11. Finally by pressing ⚡ Generate you can generate the IP. You can also modify the existing IPs by using 🔍 Load IP button.

See Add Custom IP Tutorial for observing an example of adding a custom IP core to the ProNoC library.
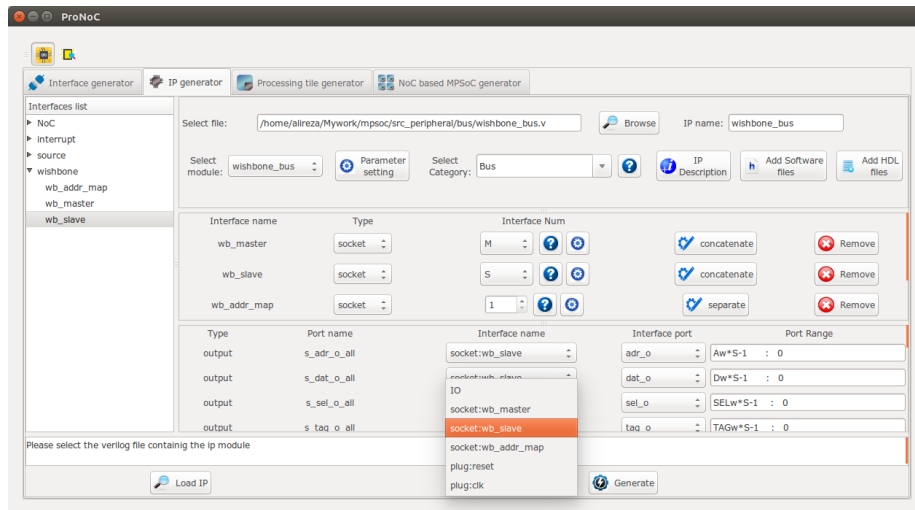
Figure 3.3: Wishbone Bus module interface mapping snapshot.

List of available Variables in ProNoC

- `${[parameter_name]}`: The IP core parameter value. The actual value is defined by the user when calling IP core at processing tile generation time. The parameter had to be added in GUI parameter using `parameter setting` button.

- `${CORE_ID}`: Each Wishbone bus-based processing tile will have a unique `CORE_ID` that represents its location in NoC topology.

  If the generated tile is used as top level module `CORE_ID` will take the default value of zero.

- `${IP}`: is the peripheral device instance name which is defined by the user when calling IP core using Processing tile generator.

- `${CORE}`: is the peripheral device IP core name.

- `${BASE}`: is the wishbone base address(es) and will be added during processing tile generation to processing tile C header file (`mpsoc/SOC/[PT-name]/sw/[Tile_name].h`). If more than one slave wishbone bus exist in the IP core, the variables are define as `${BASE0}`, `${BASE1}`... .

List of available IP cores in ProNoC

This section provides a brief description about the available IP core modules in ProNoC library. Most of IP cores that are developed with ProNoC software come with a separate documentation PDF file. Theses files are accessible by clicking on the IP core modules' name in following section. For the other IP cores which are adopted from OpenCores website the project homepage URL address is linked to the IP core name.

Bus

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

Communication

- **Etmach_100:** The Ethernet MAC (Media Access Control) 10/100 Mbps. This IP core is adopted from OpenCores/ethmac.

- **jtag_uart**: The Altera JTAG UART core with Wishbone bus interface.

- **jtag_wb**: Altera VJTAG to Wishbone bus interface. This module allows reading/writing data to the IP cores connected to the wishbone bus (e.g. memory cores). The communication between the host PC and the VJTAG 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.

DMA

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

Display

- **lcd_2×16**: 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 combination of several sources of interrupt.

NI

- **ni_master**: ni_master is 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**: ni_slave is an extension of NI_master module connected to two input and output buffers. There are three WB slave interfaces in this module, one for writing on output buffer, one for reading 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) connecting via interfaces. Figure 4.1 illustrate a snapshot of PT generator. PT generator facilitates the RTL code generation of a custom PT by providing following features:

1. Allows addition of any arbitrary number of IP cores to the PT.

2. Provides a simple GUI for connection 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 or QuartusII compilers.

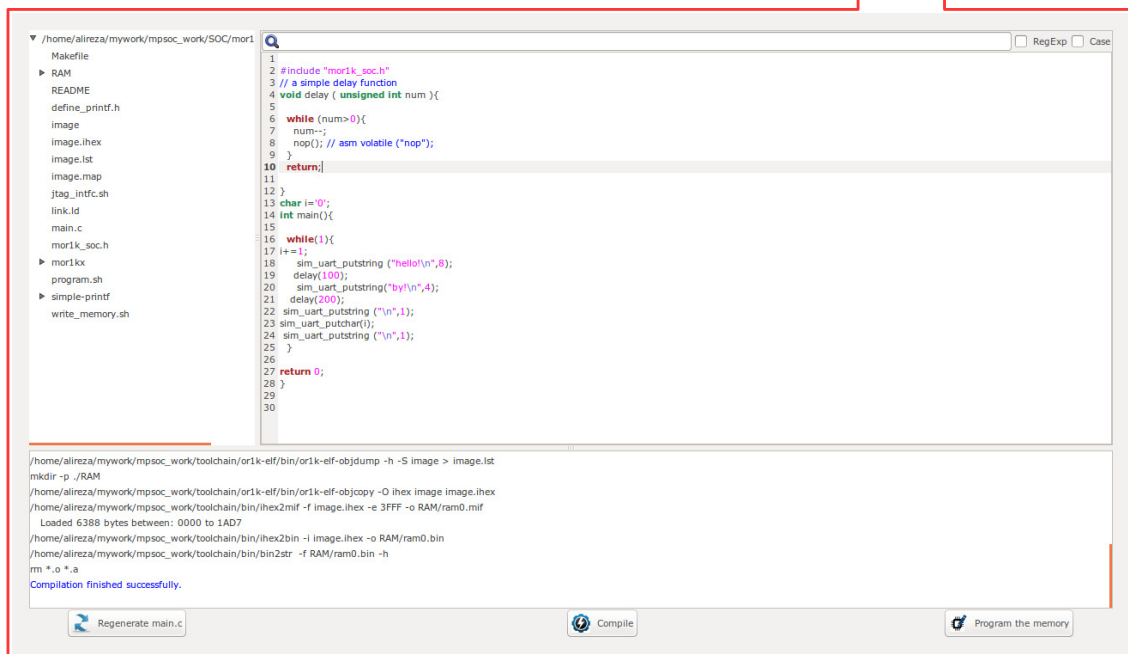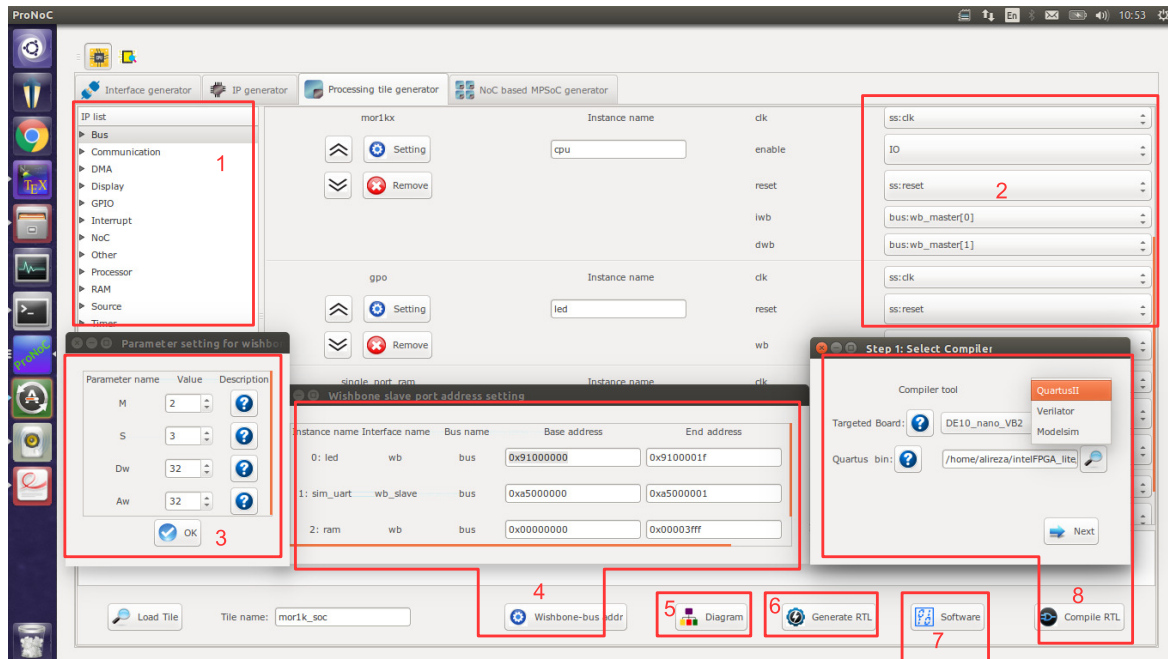For more information about PT generator, please refer to Processing Tile Generator Tutorial.

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 **ProNoC Processing Tile Generator**. The desired 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 the ProNoC FPGA board list please follow the instructions given in Adding a New Altera FPGA Board to ProNoC, to add your board to the ProNoC library.

Objectives:

1. To design a Wishbone bus-based system-on-chip hardware architecture using ProNoC Electronic Design Automation (EDA) software.

2. To develop a simple software application running on generated SoC.

3. To interact with on-board memory units using 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 4-bit general purpose output (GPO)

2. A 32-bit timer.

3. A mor1kx processor (You can use any of 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).
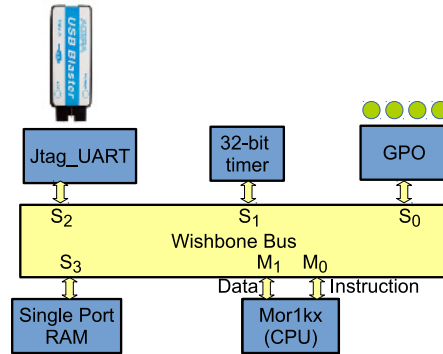
Figure 5.1: The schematic of desired SoC in this tutorial.

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

```
./ProNoC.pl
```

It should open The GUI interface as illustrated in Figure 5.2.
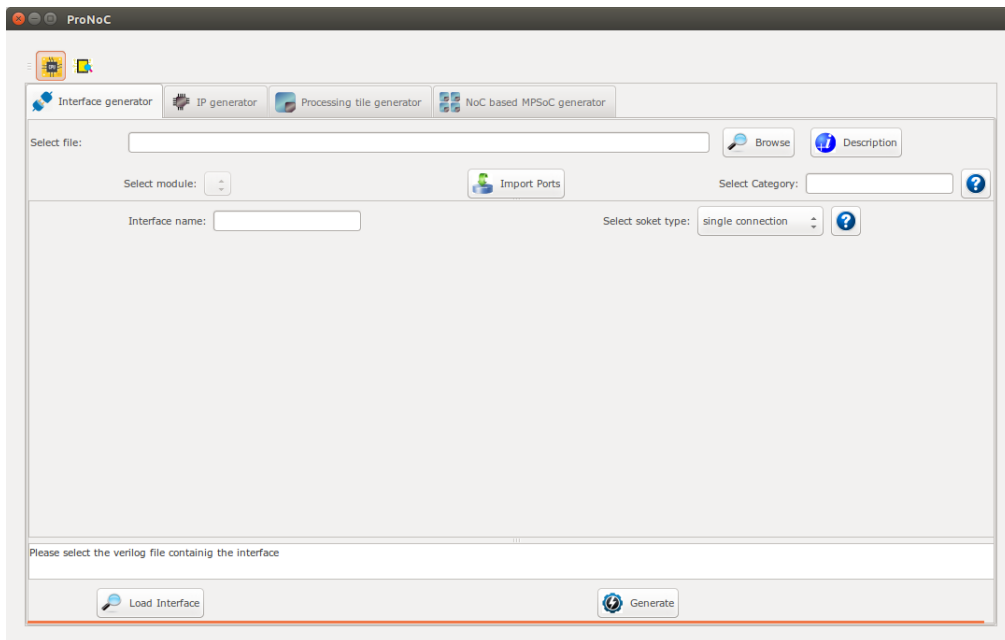
Figure 5.2: ProNoC GUI first page snapshot.

Then select the Processing Tile Generator tab:
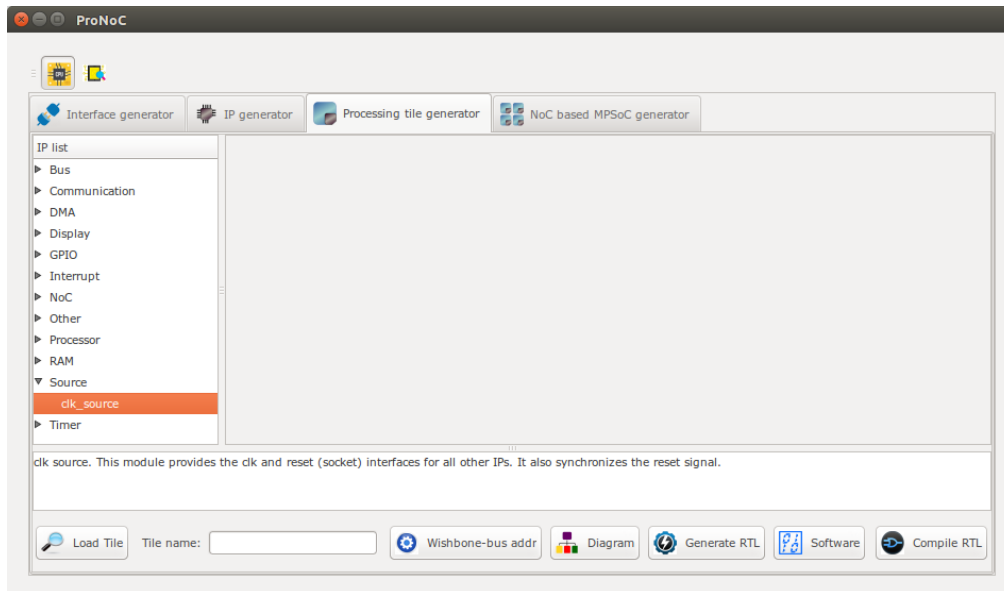
Figure 5.3: ProNoC New Processing Tile generator snapshot.

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

(a) In first column you can shift IP core box position up/down in 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) 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 Interface Generator chapter for more information about interfaces. You can also export the interface as SoC's input/output (IO) ports here.

Now let start calling required IPs. We start with clk_source:

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

1. Click on `Source` category, then double click on `clk_source`.

2. Rename the `clk_source` instance name as source. leave the interfaces as `IO`.
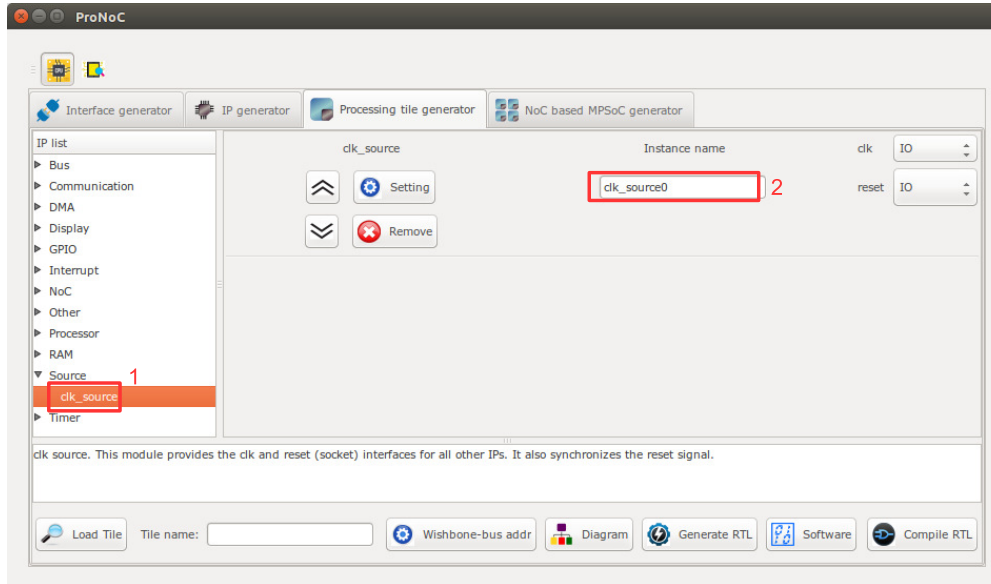
Figure 5.4: Adding clock source.

Add Wishbone Bus:

1. Click on `Bus` category and double click on `Wishbone_bus`.

2. In 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 changed 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 `clk_source` module.
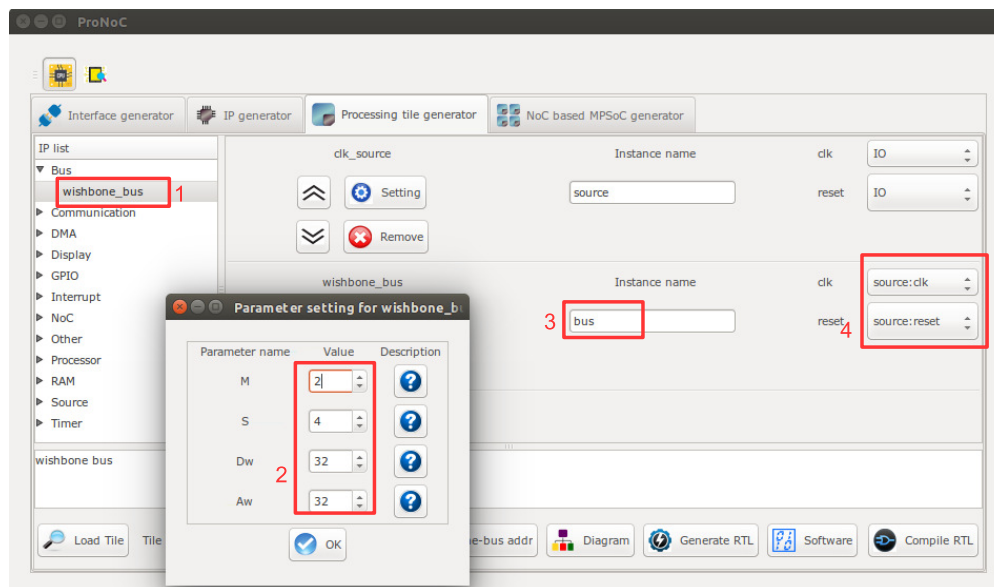
Figure 5.5: Adding Wishbone bus.

Add GPO:

1. Click on `GPIO` category and then double click on `gpo`.

2. In parameter setting set `PORT_WIDTH` as 4.

3. Rename the instance name as `led`.

4. Connect the clock and source interfaces to `clk_source` module.

5. In interface connection column connect `wb` (Wishbone bus) interface to `bus:wb_slave[0]`
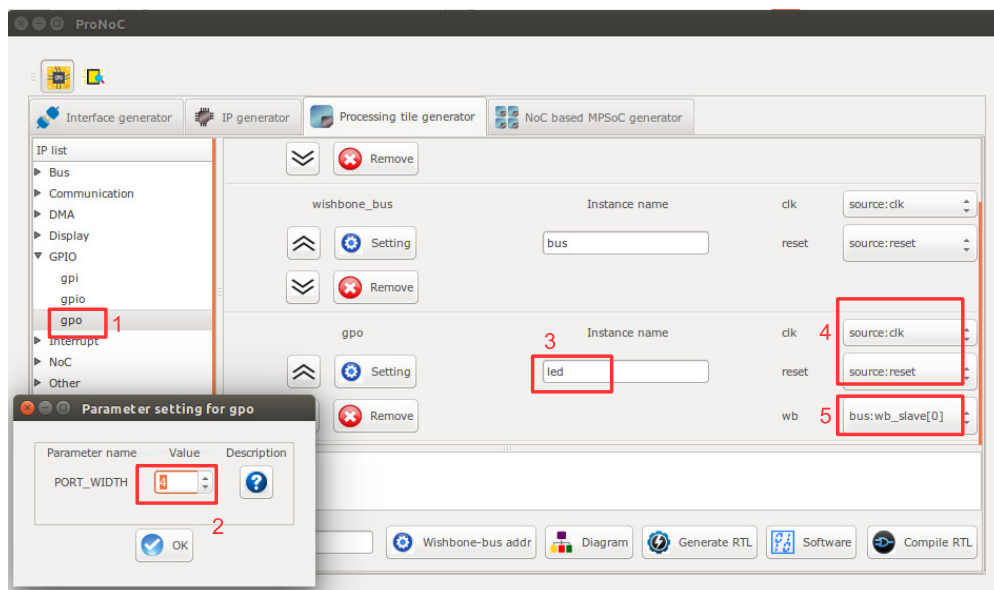
Figure 5.6: Adding GPO.

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 zeroth `wb` interface. Note that you can optionally connect it to any of other `wb` interfaces number as WB has a round-robin arbitration scheduler.

Add Processor:

1. Click on `Processor` category and then double click on `mor1kx`.

2. Rename the instance name as `cpu`.

3. Connect the clock interface to `clk_source:clk` module.

4. Connect `enable` interface to `IO`

5. Connect the reset interface to `clk_source:reset` interface.

6. Connect `iwb` (instruction wishbone bus) and `dwb` (data wishbone bus) interfaces to `bus:wb_master[0]` and `bus:wb_master[1]`, respectively.
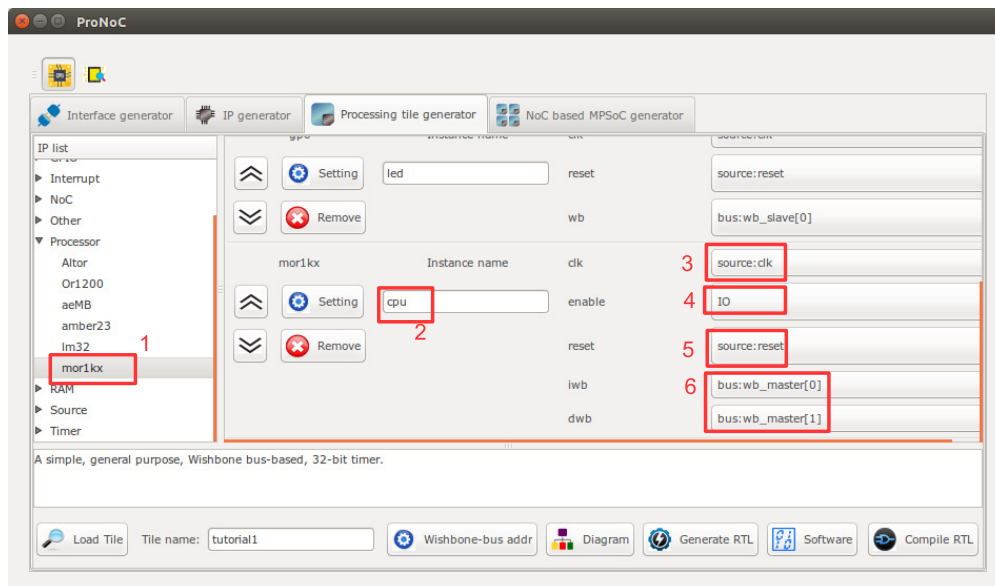
Figure 5.7: Adding Processor.

Add Timer:

1. Click on `Timer` category and then double click on `timer`.

2. Rename the instance name as `timer`.

3. Connect the clk interface to `clk_source:clk` interface.

4. Connect `interrupt` interface to `cpu:int_periph[0]`.

5. Connect the reset interface to `clk_source:reset` interface.

6. Connect `wb` (Wishbone bus) interface to `bus:wb_slave[1]`.
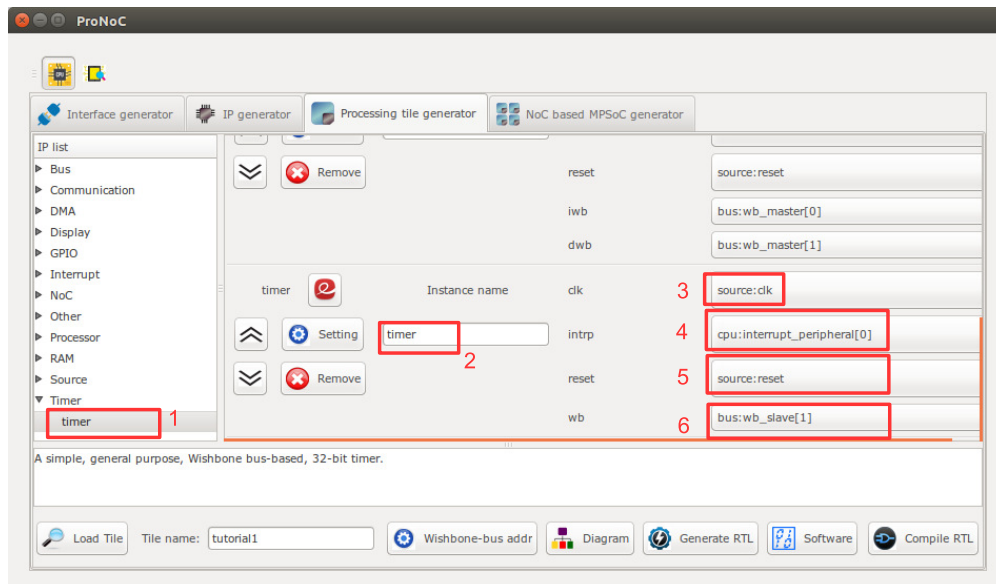
Figure 5.8: Adding Timer.

**Add JTAG
UART:**

1. Click on `communication` category and then double click on `jtag_wb`.

2. Rename the instance name as `jtag_uart`.

3. Connect the clk interfaces to `clk_source:clk`.

4. Leave `interrupt_peripheral` unconnected (NC).

5. Connect the reset interface to `clk_source:reset`.

6. Connect `wb_slave` interface to `bus:wb_slave[2]`.
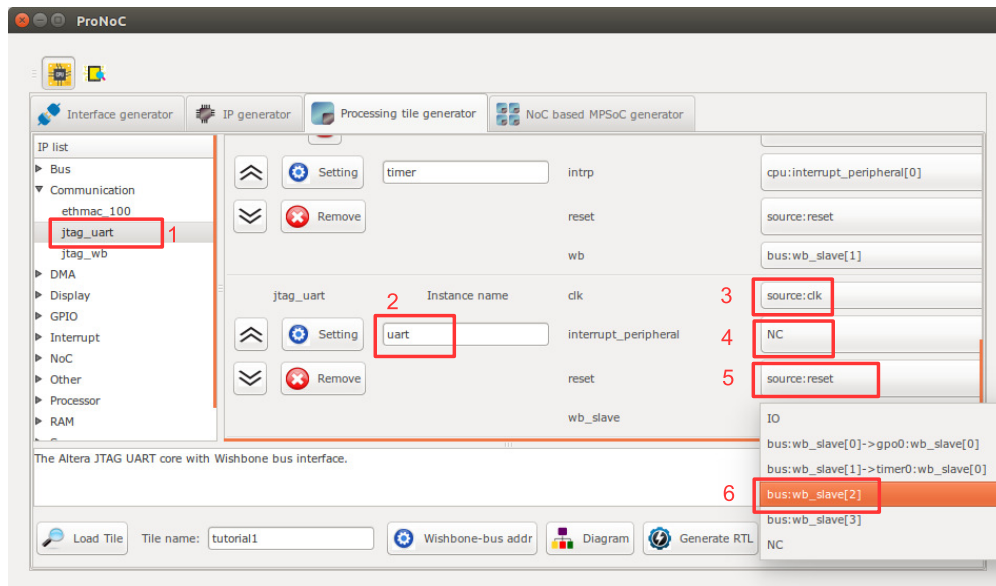
Figure 5.9: Adding JTAG UART.

Add Single port RAM:

1. Click on `RAM` category and then double click on `single_port_ram`.

2. In parameter setting set `Aw` as 14. AW is the memory address width. Hence, this results in a $2^{14} \times 32$ bit= 500kb memory. Make sure your FPGA board has sufficient on-chip BRAM to be allocated. Otherwise decrees the AW to fit with your target device.

3. Select `ALTERA` for `FPGA_VENDOR`.

4. Connect `JTAG_CONNECT` to `JTAG_WB`. This allows the editing of memory contents at run time using JTAG interface.

5. Set `INITIAL_EN` as `"YES"`. This enable the memory initialization at compilation time. This configuration is also required for simulating the system using Model-sim or Verilator softwares. Leave the rest of parameters as their default.

6. Rename the instance name as `ram`.

7. Connect the clk and reset interfaces to `clk_source` module.

8. Connect `wb` interface to `bus:wb_slave[3]`.
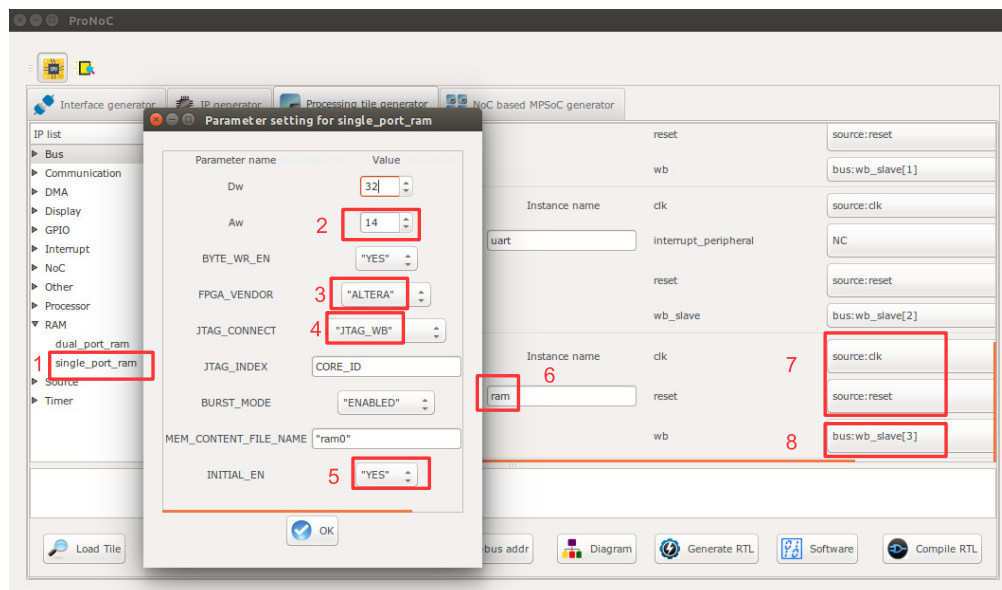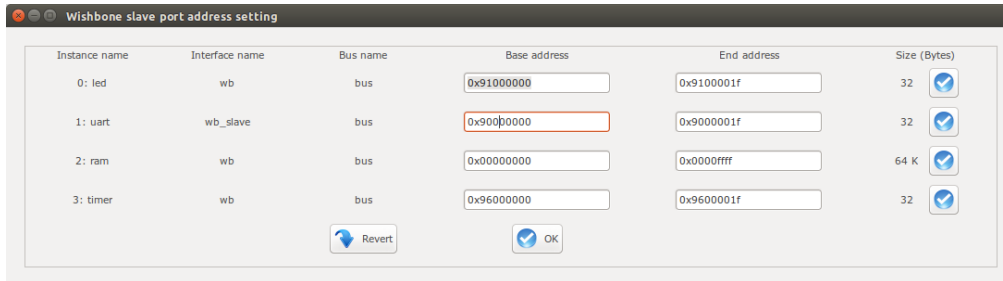
Figure 5.10: Adding Single port RAM.

Check wishbone bus(es) addresses: After adding all required IP cores, now you can check the auto-assigned Wishbone bus addresses by clicking on ⚙ `Wishbone-bus addr` button. Note that the assigned addresses are also modifiable.

Instance name	Interface name	Bus name	Base address	End address	Size (Bytes)	
0: led	wb	bus	0x91000000	0x9100001f	32	✓
1: uart	wb_slave	bus	0x90000000	0x9000001f	32	✓
2: ram	wb	bus	0x00000000	0x0000ffff	64 K	✓
3: timer	wb	bus	0x96000000	0x9600001f	32	✓

Figure 5.11: Wishbone bus addresses of the tutorial SoC.

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

View SoC functional block diagram: Press the ▦ `Diagram` button to observe the SoC functional block diagram.
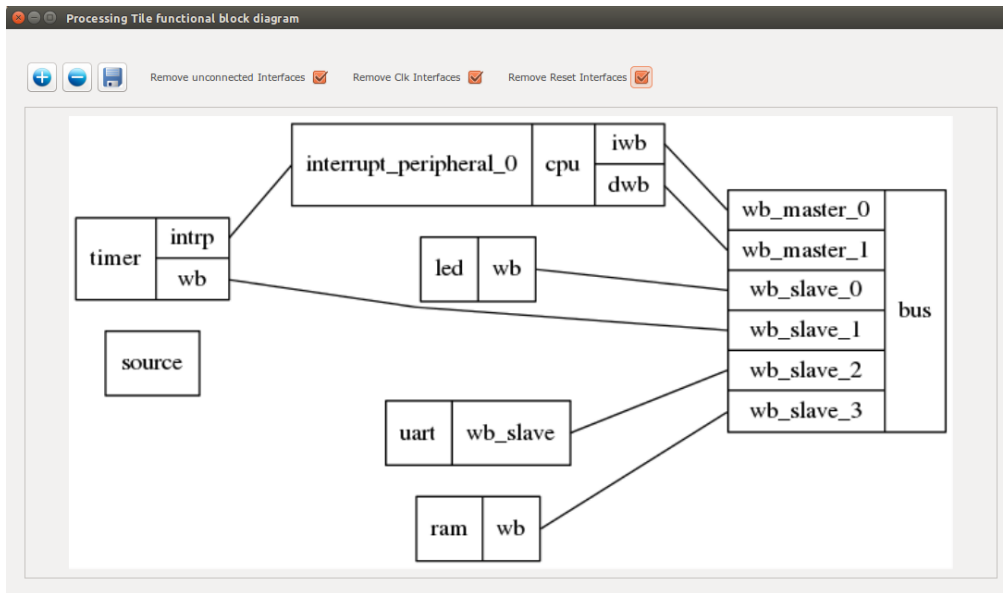
Figure 5.12: The tutorial SoC diagram.

Generate SoC RTL Code:

1. Set Tile name as `tutorial`.

2. Press ⚡ `Generate RTL` button.
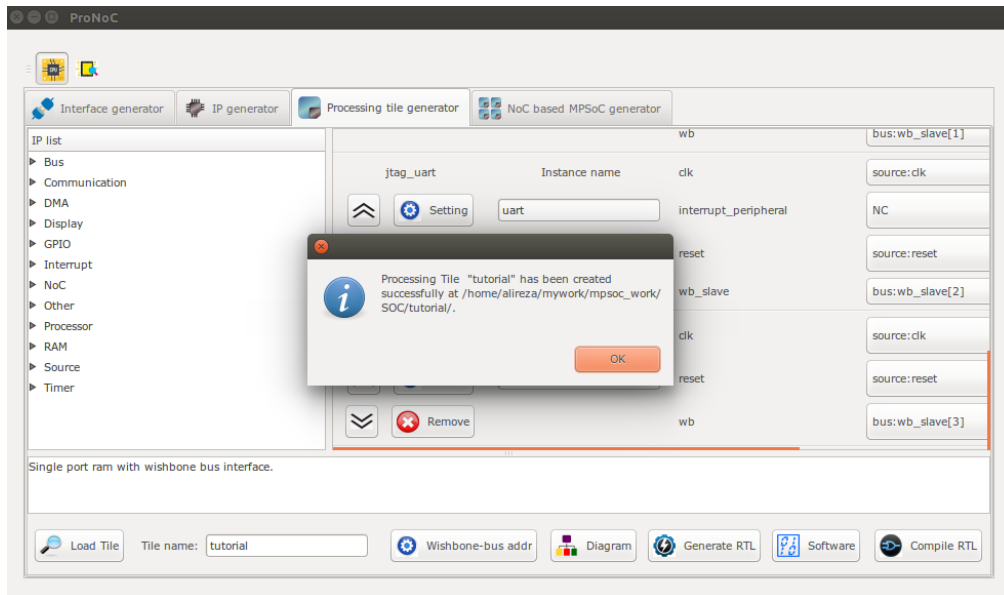
Figure 5.13: Generating the tutorial SoC.

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 all peripheral devices' WB addresses and functions (some IPs may have additional header files).
  - `README`: This file contains SoC parameters, IP connection and wishbone bus addresses. This file also explain how to work with `Jtag_wb` IP core.
  - `program.sh`: A sample bash file that can be used for programing the SoC RAMs at run time using JTAG interface.

- `src_verilog`: contains two Verilog files and a folder:

  - `tutorial.v`: the generated SoC RTL code. This file contains all IPs instances and connections.
  - `tutorial_top.v`: this file contains the tutorial SoC module instance connected to a JTAG-based remote enable/reset controller which disable the SoC during programming time.

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

Software Development

1. Click on the ![Software icon] `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. Click on `tutorial.h` file to see the file contents. This file contains all generated SoC functions and WB addresses.
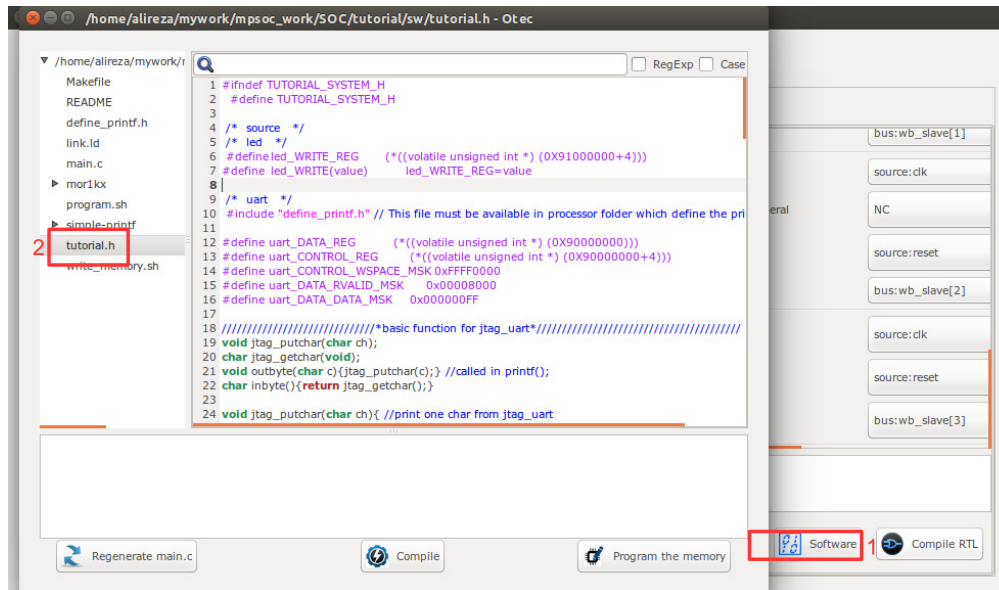
Figure 5.14: The software edit window snapshot.

3. Now click on `main.c` file. Replace the contents of this file with the following C code. This code writes the "Hello worlds!" on Altera JTAG UART port once, and then controls the LEDs using the timer interrupt service routine. Each time an interrupt happens the LED which is on is turned off and the neighboring one is turned on. The timer asserts an interrupt in every 500 clock cycles. The interrupt time is deliberately chosen too small to speed up the simulation. In 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);
    int_init();
    //assume hw interrupt pin is connected to cpu intrrupt pin 0
    int_add(0, timer_isr, 0);
    // Enable this interrupt
    int_enable(0);
    cpu_enable_user_interrupts();
    timer_int_init(500);
    while(1){
        delay(500);
    }
    return 0;
}
```

4. Now press the ⚙ `Compile` button. This will compile the C code using Mor1kx GNU toolchain. If everything runs ok, you must see "compilation finished successfully" message as shown in Figure 5.15. Otherwise, check the error message to fix your code and press the compile button again. If every thing runs successfully you must have `ram0.bin`, `ram0.hex`, and `ram0.mif` files in your `sw/RAM` directory.
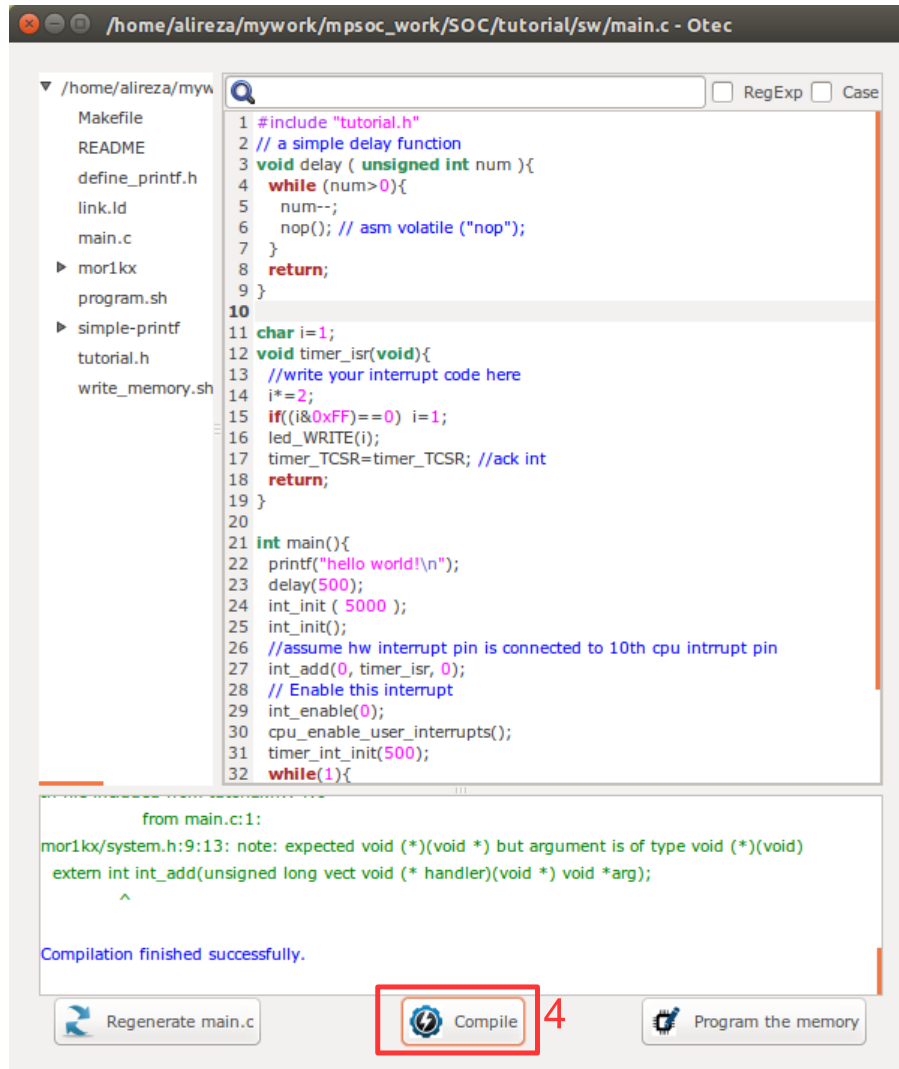
Figure 5.15: Compile the software code.

Simulate the generated RTL code using Modelsim software

If you have installed Modelsim software on your system, you can simulate your SoC working with 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 5.16.

2. Select Modelsim as compiler tool.

3. Enter the path to your installed Modelsim bin directory.

4. Press the ➡ Next button.
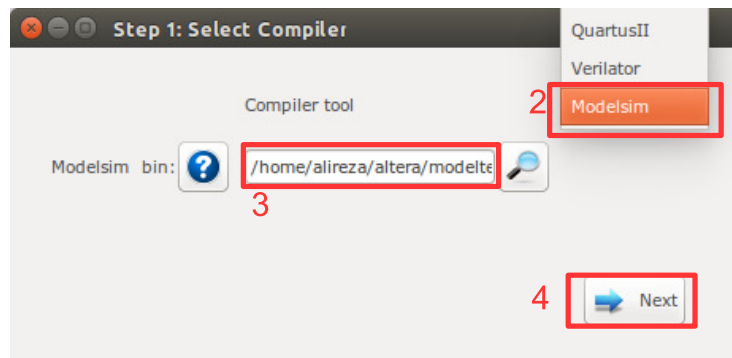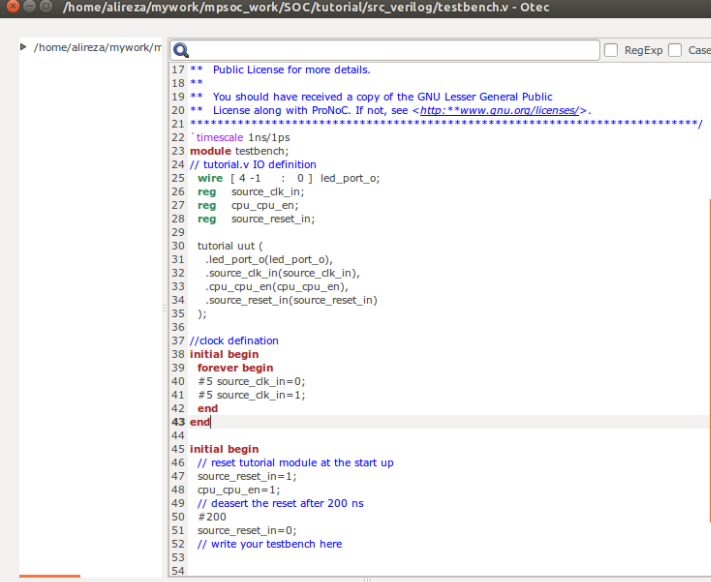
Figure 5.16: Select Modelsim as simulator.

5. Now you must have the testbech.v opened in text editor window as shown in Figure 5.17. This is the minimum testbench file for running the simulation of the generated SoC in Modelsim software. It has the SoC instance module connected to the clock and reset signals. You can edit this file as you wish.

6. Press the ▶ run button to run the simulation in Modelsim software.

Figure 5.17: testbech.v file snapshot.

7. Figure 5.18 shows the Modelsim simulation output snapshot. You must see the "hello world!" expression in the Modelsim terminal. The LEDs outputs also must be seen as cyclic shift to the left of a one-hot code in the Signal Waveform Window.
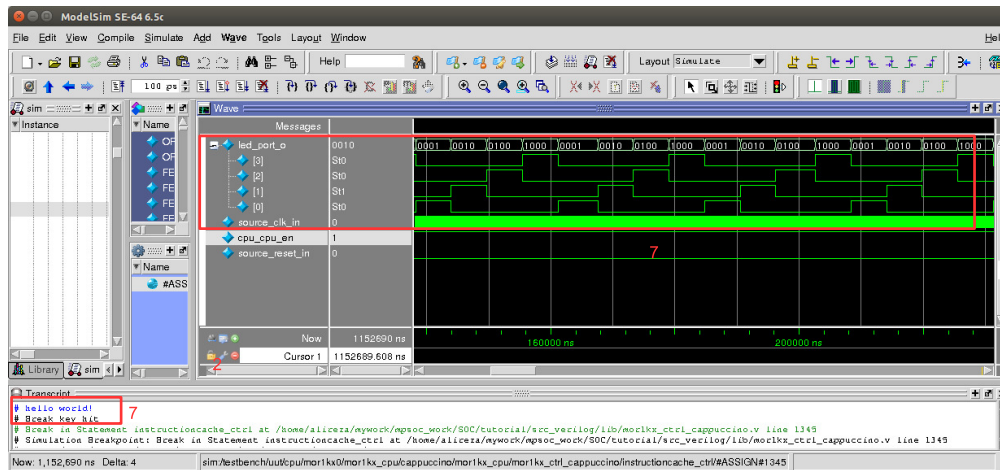
Figure 5.18: Modelsim output snapshot.

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 5.19. Select Verilator as compiler tool then press ➡ Next.
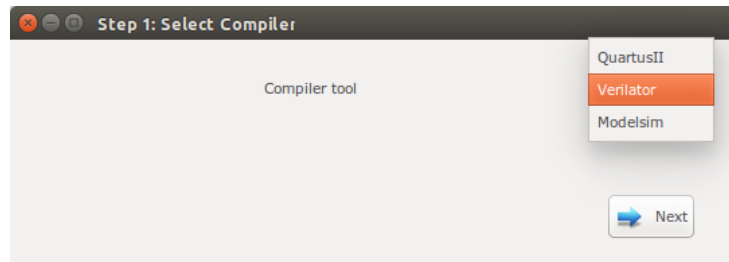
Figure 5.19: Select Verilator compiler.

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 5.20.
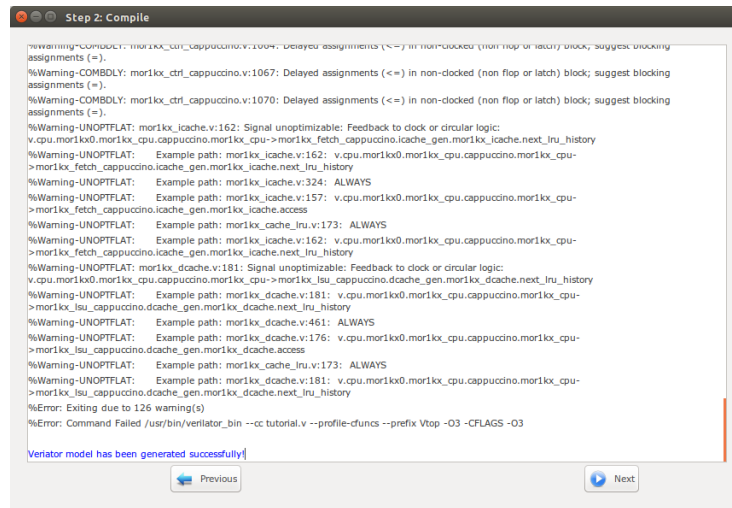
Figure 5.20: Verilator model generation snapshot.

3. Press Next.

4. Now you must have the `testbech.c` opened in software code edit window as shown in Figure 5.21. 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.
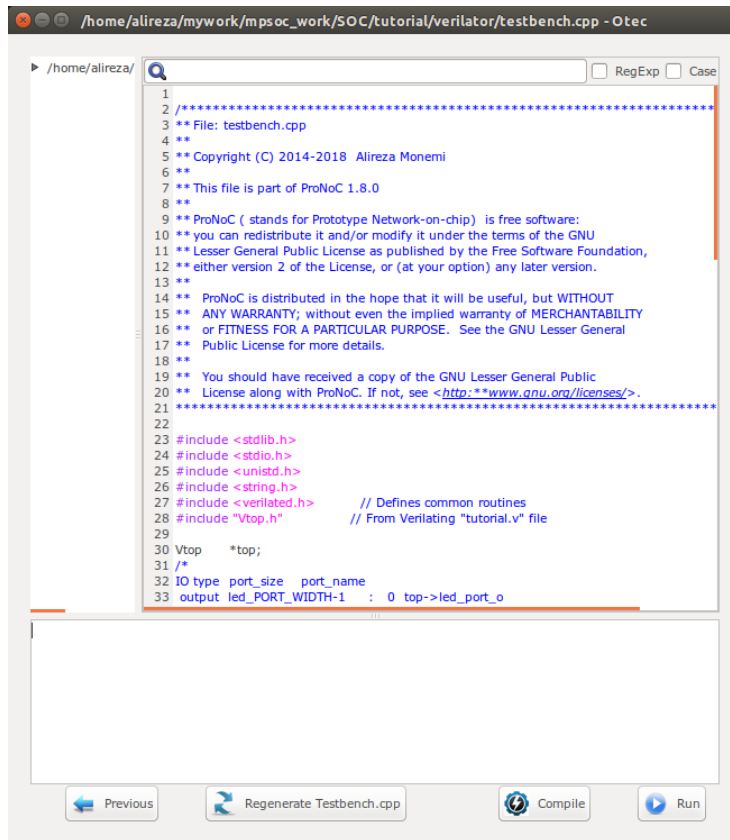
Figure 5.21: Verilator model testbench edit snapshot.

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
```

```
65
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 gener-
   ated successfully you must see the "compilation finished successfully" message
   as shown in Figure 5.22.
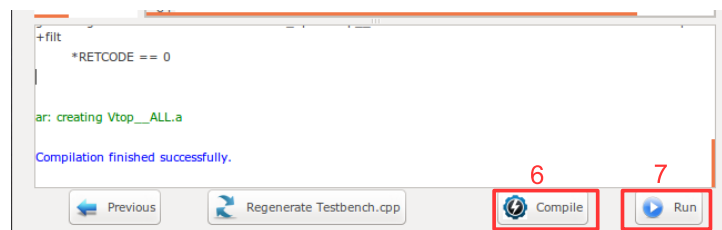
Figure 5.22: Verilator compilation successful snapshot.

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 5.23.

Figure 5.23: Verilator simulation results snapshot.

Compile the generated RTL code using Quartus II software

If you have installed Quartus II software on your system and you have an Altera 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 5.24. Select QuartusII as compiler tool.
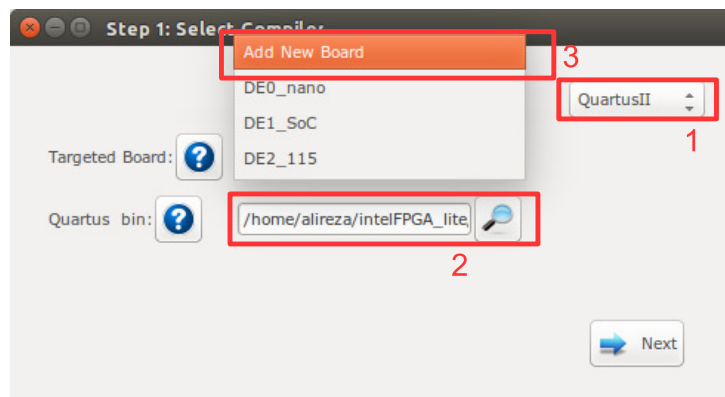
Figure 5.24: Select QuartusII as compiler.

2. Enter the path to your installed QuartusII `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 and then press the ➡ Next button.

4. If you selected Add New Board, a new window as shown in Figure 5.25 must be appear. Fill the required fields as follows:

   (a) Enter your board name. Do not use any space in the given name

   (b) 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).

   (c) 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).

   (d) Power on your FPGA board and connect it to your PC then press the ✨ Auto Fill button to auto-fill the JTAG configuration setting.

   (e) Press the ➕ Add button.
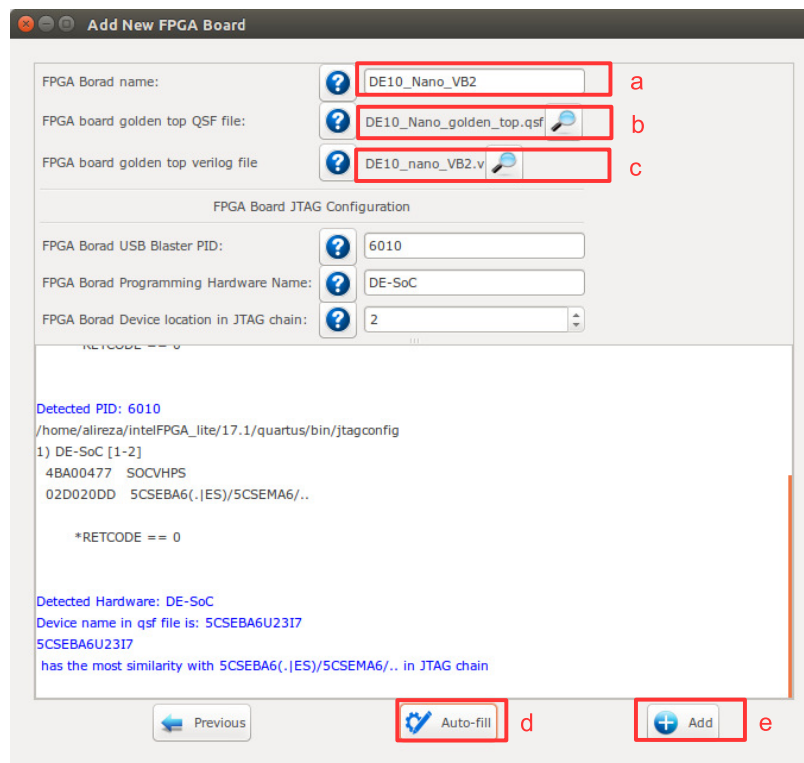
Figure 5.25: Add new FPGA board to ProNoC.

5. Assign your SoC pins to your FPGA boards pins as shown in Figure 5.26.
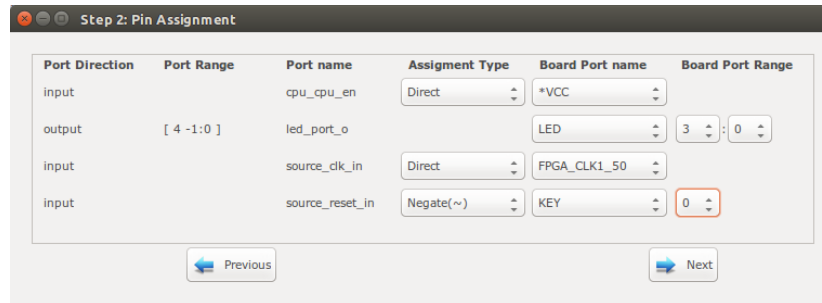
Figure 5.26: SoC pin assignment.

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.
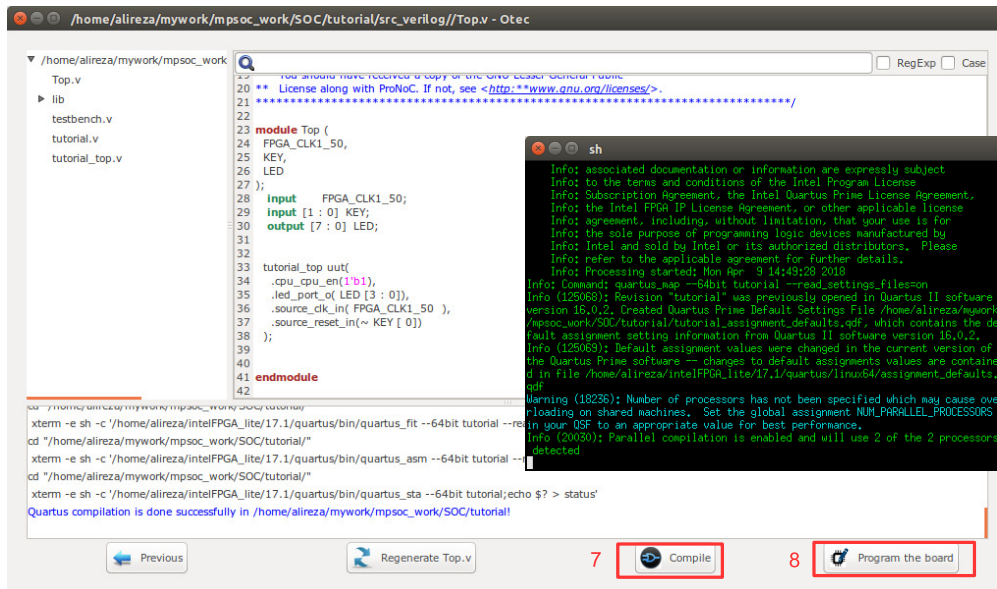
Figure 5.27: QuartusII compilation snapshot.

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

9. 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 5.28.
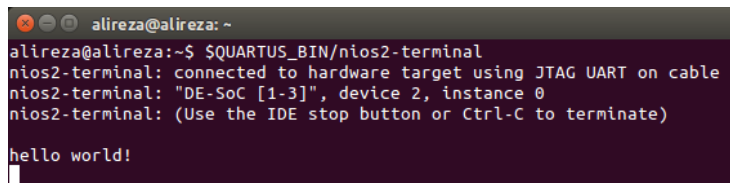
Figure 5.28: nios2-terminal output snapshot.

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 ![icon] Software button and change the timer interrupt time from 500 to 5000000. Then press the ![icon] compile button. By clicking on ![icon] Program the Board button you can reprogram your SoC memory contents at run time. You should be able to observe the blinking LEDs now.
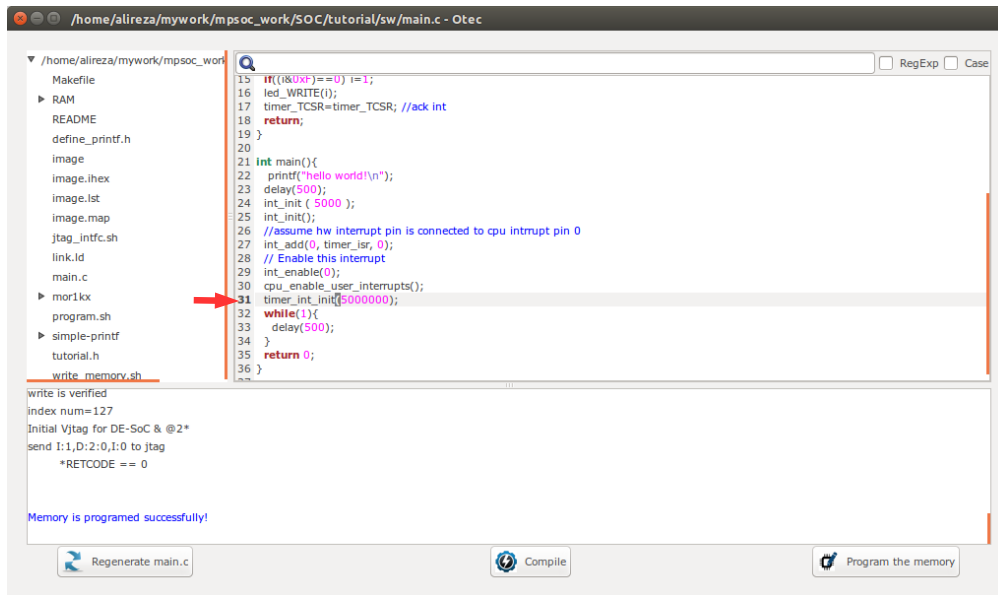
Figure 5.29: Increase timer interrupt time.

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**
$\quad$ **if** $p > q$ **then**
$\quad\quad |\quad p = p - q$;
$\quad$ **end**
$\quad$ **else if** $p < q$ **then**
$\quad\quad |\quad q = q - p$;
$\quad$ **end**
$\quad$ **else**
$\quad\quad |\quad gcd = p$;
$\quad$ **end**
end

Algorithm 1: Greatest Common Divisor algorithm.

The GCD flow chart:
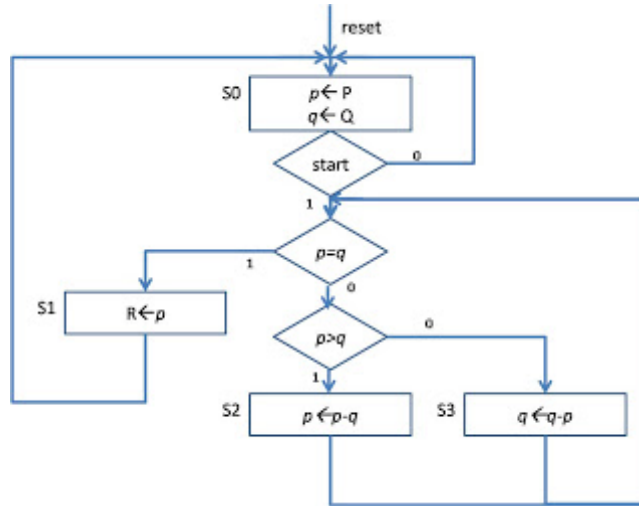
Figure 6.1: GCD flow chart.

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

Listing 6.1: gcd.v

```verilog
/************************
 * 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),
```

```verilog
      .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;
```

```verilog
               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,
```

```verilog
 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

```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

```verilog
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,
```

```verilog
    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
```

```verilog
        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
    ```

    It should open The GUI interface as follows:
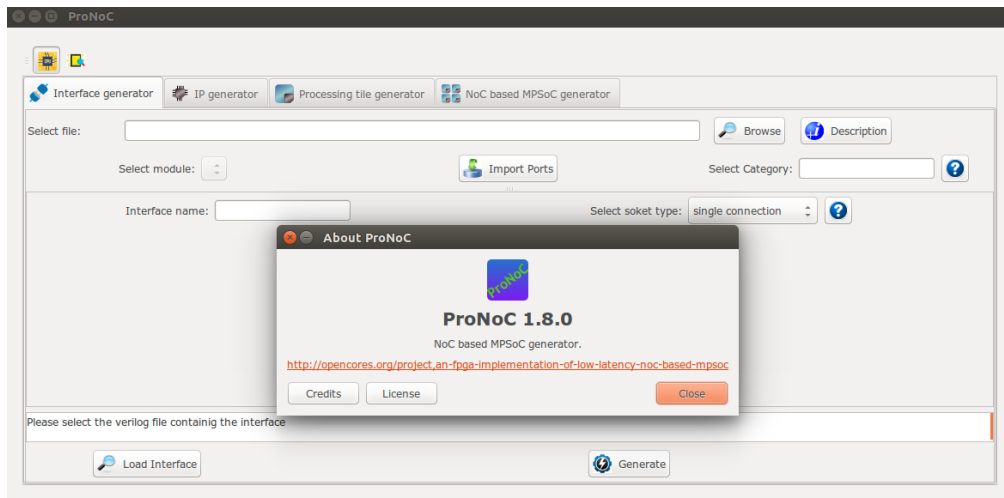
Figure 6.2: ProNoC GUI first page snapshot.

2. Then select the ⊞ IP generator. The IP Generator snapshot is shown in Figure 6.3.
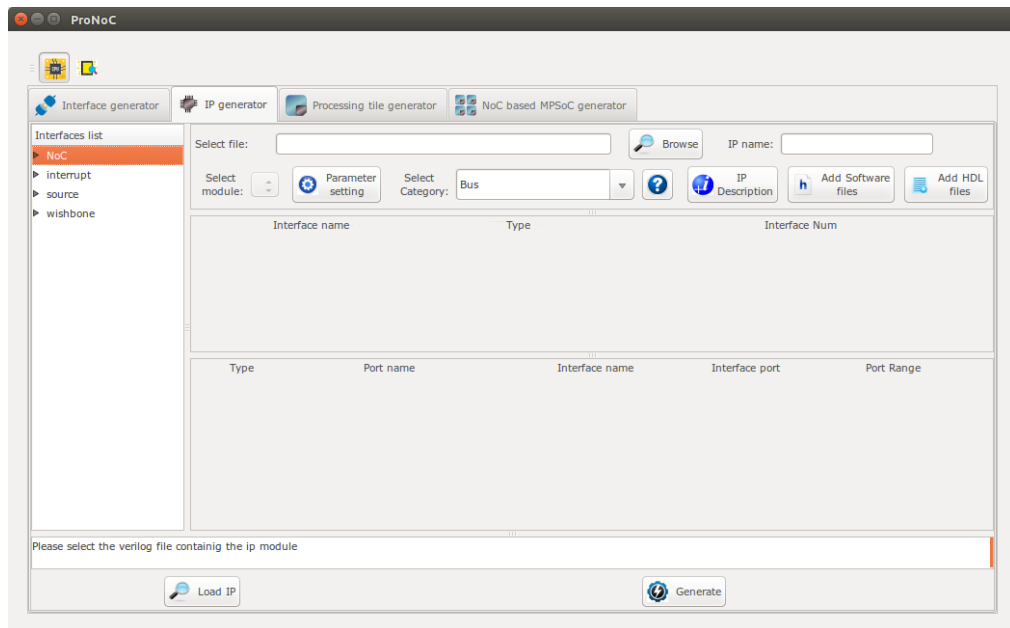
Figure 6.3: ProNoC New IP Generator snapshot.

3. Click on 🔍 Browse and select gcd_ip.v file.

4. Enter Other as category name.

5. Enter gcd as IP name.
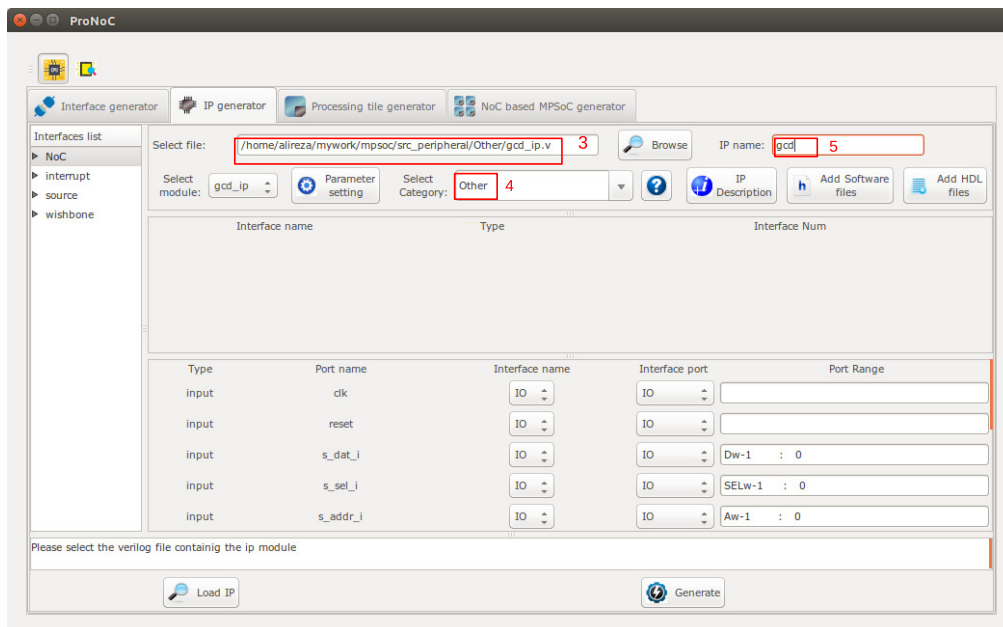
Figure 6.4: Select `gcd_ip.v` file.

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.
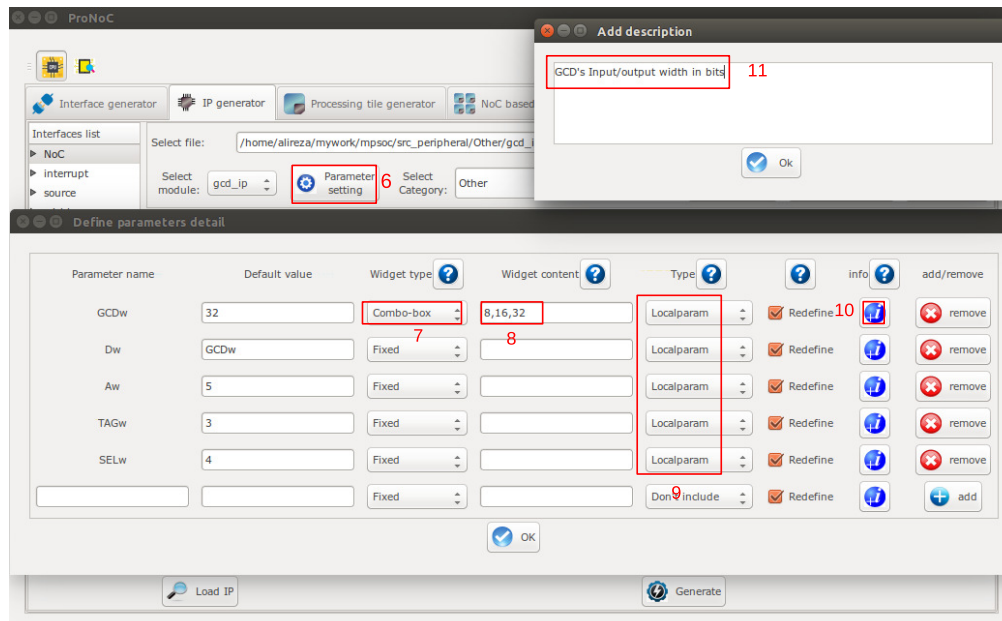
Figure 6.5: GCD IP core parameter setting.

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.
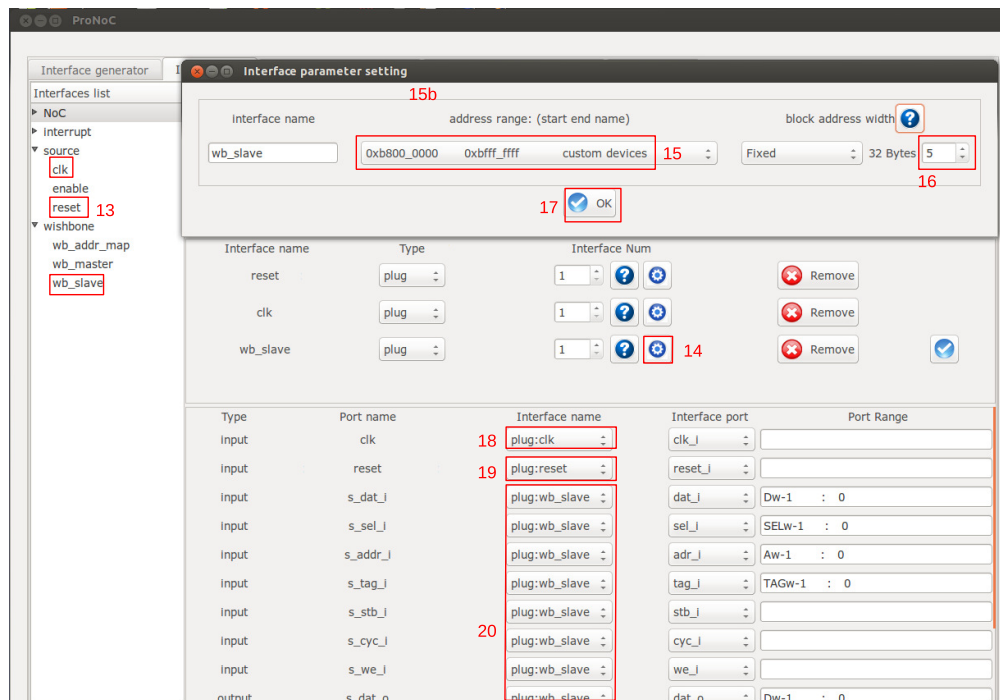
Figure 6.6: GCD Core interface setting.

21. Click on ![Add HDL Files icon] Add HDL Files button.

22. In front of Select file(s) click on ![Browse icon] Browse button.

23. Select gcd.v and gcd_ip.v files and press ![ok icon] ok.
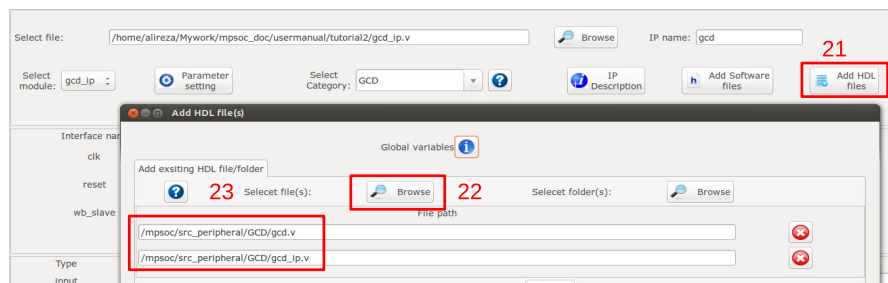
Figure 6.7: Adding GCD core HDL files.

24. Click on ![h icon] 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 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` file. This file contains 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.

27. Click on 🌀 `Generate` to add the GCD IP core to the library.
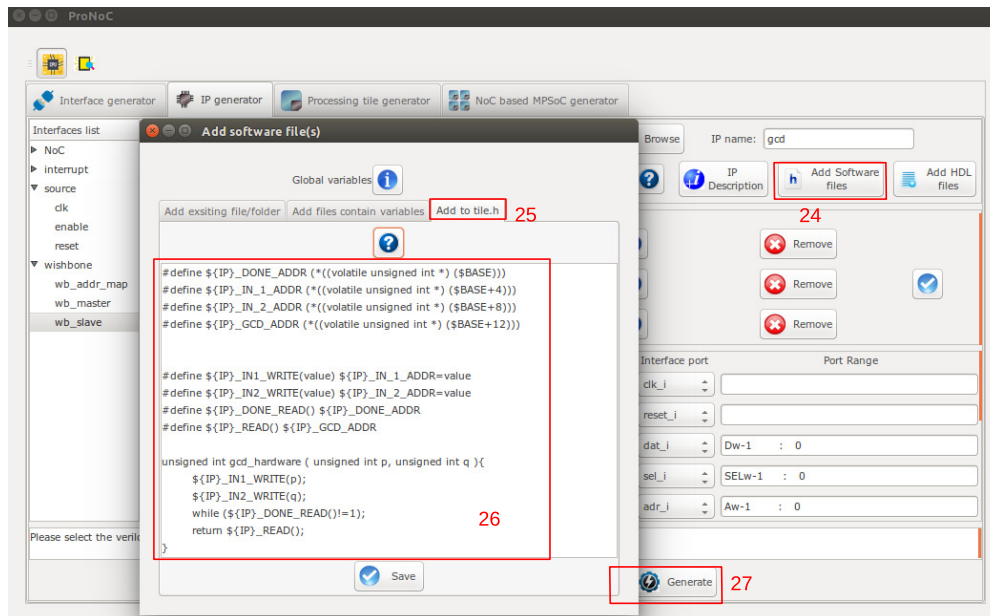
Figure 6.8: Add GCD software files.

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 6.9.
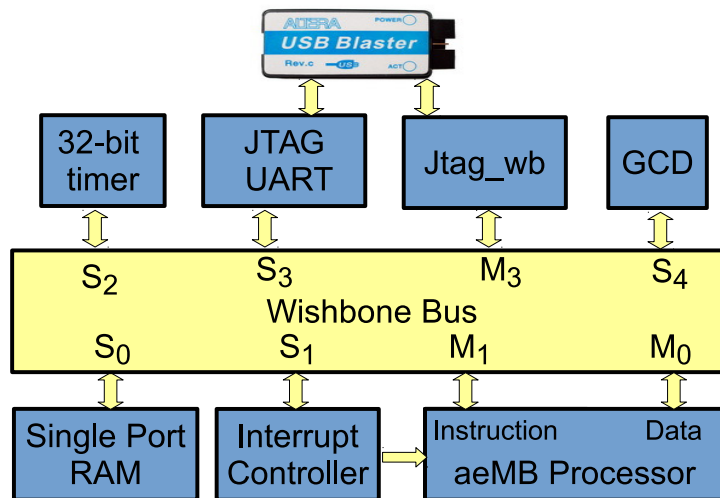
Figure 6.9: Desired SoC with GCD IP core.
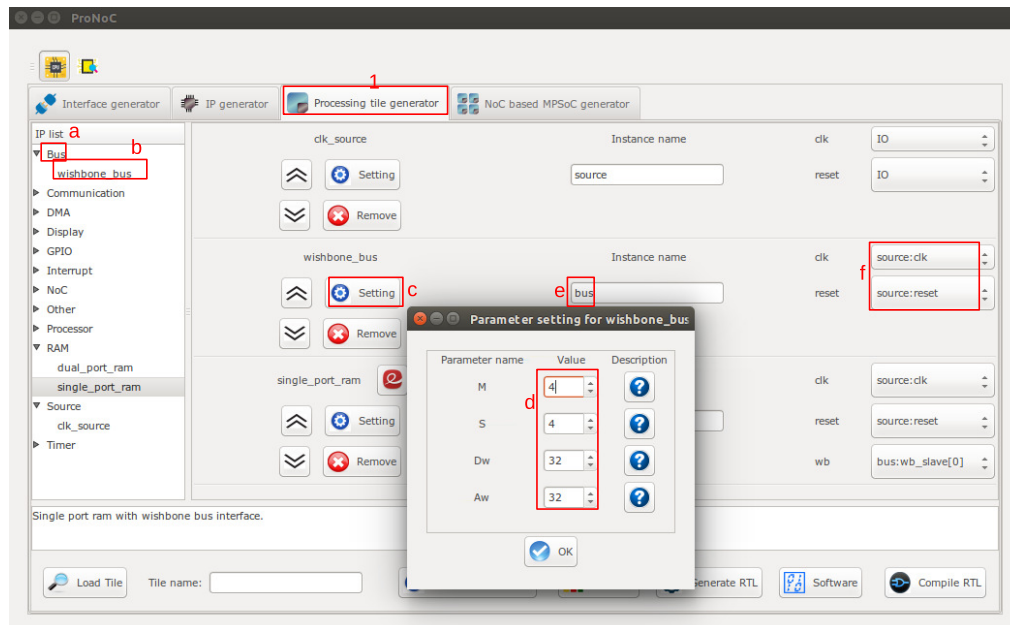
Figure 6.10

1. In ProNoC GUI Click on `Processing Tile Generator`. This tool facilitates the generation of a custom SoC using a list of available IP cores. Add all required IP cores according to the following stages:

   (a) Click on IP core category name to see the list of its containing IP cores.

   (b) Double click on each IP core name to add the IP core to the SoC. Add all IP cores listed in Table 6.2 at first, then continue with the next step.

   (c) Click on ⚙ `Setting` button to open each IP core parameter setting window.

   (d) Adjust the IP core parameters according to Table 6.2.

   (e) Rename the IP core instance name according to Table 6.2.

   (f) Connect the IP cores interfaces as listed in Table 6.2.

Table 6.2: GCD SoC IP core list and setting.

Category	IP name	Parameter			Instance name	Interface connection		
Source	clk_source	-			source	clk	→	IO
						reset	→	IO
Bus	wishbone_bus	M	→	3	bus	clk	→	source:clk
		S	→	5		reset	→	source:reset
		Dw	→	32				
		Aw	→	32				
Processor	aeMB	STACK_SIZE	→	0X400	aeMB	clk	→	source:clk
		HEAP_SIZE	→	0X400		reset	→	source:reset
						iwb	→	bus:wb_master[0]
						dwb	→	bus:wb_master[1]
						enable	→	IO
RAM	single_port_ram	Dw	→	32	ram	clk	→	source:clk
		Aw	→	12		reset	→	source:reset
		BYTE_WR_EN	→	"YES"		wb	→	bus:wb_slave[0]
		FPGA_VENDOR	→	"ALTERA"				
		JTAG_CONNECT	→	"DISABLED"				
		JTAG_INDEX	→	CORE_ID				
		BURST_MODE	→	"DISABLED"				
		MEM_CONTENT_FILE_NAME	→	"ram0"				
		INITIAL_EN	→	"YES"				
Interrupt	int_ctrl	INT_NUM	→	1	int_ctrl	clk	→	source:clk
						reset	→	source:reset
						interrupt_cpu	→	aeMB:interrupt_cpu
						wb	→	bus:wb_slave[1]
Timer	timer	PRESCALE_WIDTH	→	8	timer	clk	→	source:clk
						reset	→	source:reset
						wb	→	bus:wb_slave[2]
						intrp	→	in_ctrl:int_periph[0]
Communication	jtag_wb	Dw	→	32	jtag_wb	clk	→	source:clk
		VJTAG_INDEX	→	CORE_ID		reset	→	source:reset
						wbm	→	bus:wb_master[2]
Communication	jtag_uart	FPGA_VENDOR	→	"ALTERA"	uart	clk	→	source:clk
		SIM_BUFFER_SIZE	→	100		reset	→	source:reset
		SIM_WAIT_COUNT	→	1000		intrpt	→	NC
						wb	→	bus:wb_slave[3]

2. Add the new GCD IP to SoC.

Table 6.3: GCD SoC IP core list and setting.

Category	IP name	Parameter	Instance name	Interface connection		
Other	gcd	GCDw $\rightarrow$ 32	gcd	clk	$\rightarrow$	source:clk
				reset	$\rightarrow$	source:reset
				wb	$\rightarrow$	bus:wb_slave[4]
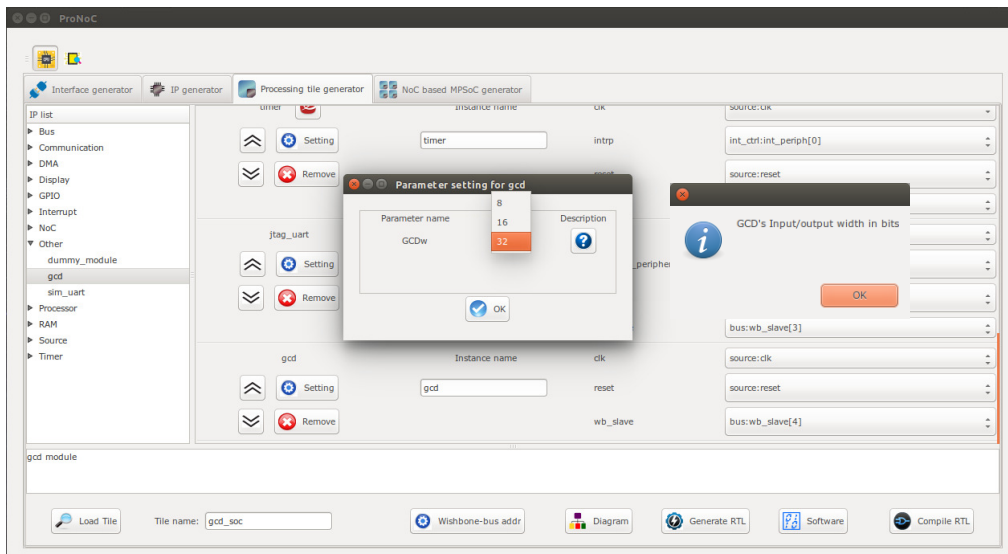
Figure 6.11: Add the generated GCD IP core to gcd_soc.

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.
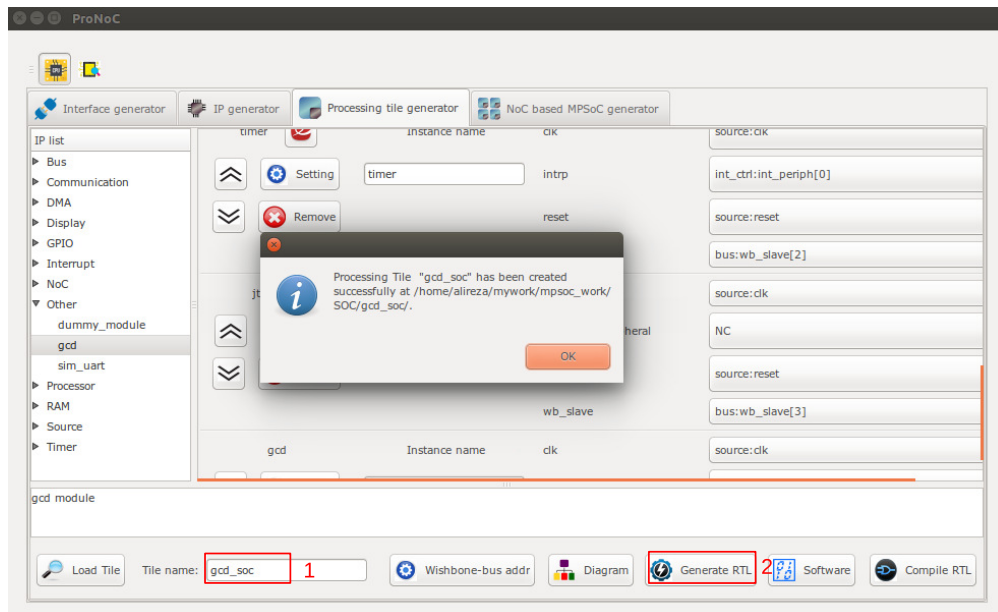
Figure 6.12: Generate the `gcd_soc` RTL codes.

Software Development

1. Click on [icon] `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.

```c
#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 Altera FPGA board. The DE10-Nano FPGA board pin assignment and a snapshot of a sample result on UART terminal is shown in Figures 6.13 and 6.14, respectively. You can test the GCD IP core by entering different values.

Figure 6.13: DE10-Nano FPGA board pin assignment.

Figure 6.14: Nios2-terminal output snapshots.

CHAPTER 7

Simple message passing demo on 4×4 MPSoC

Summary   This chapter demonstrates a simple message passing on a 4×4 NoC based MPSoC. This includes developing a custom shared bus (Wishbone bus) based processing tile using ProNoC Proc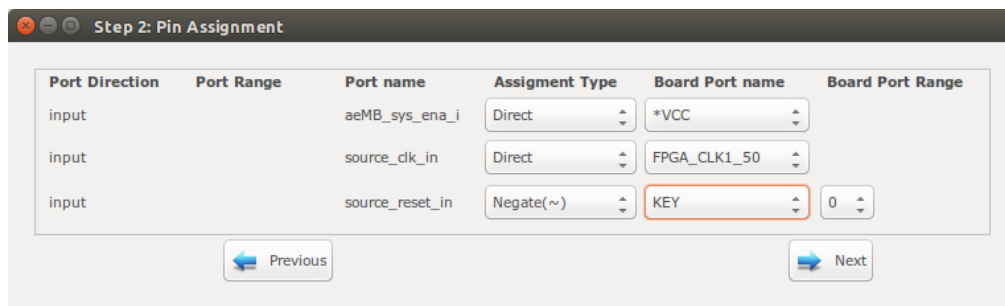essing Tile Generator. The generated tile is used then for generating a multicore using ProNoC NoC based MPSoC generator.

System Requirements: You will need an Altera 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 (or1k-elf) for compiling the Mor1kx soft-core processor software code.

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 the ProNoC FPGA board list please follow the instructions given in Adding a New Altera 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		-		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	→ → → → → → → → →	32 14 "YES" "ALTERA" "DISABLED" CORE_ID "ENABLED" "ram0" "YES"	ram	clk reset wb	→ → →	source:clk source:reset bus:wb_slave[0]
NoC	ni_master	MAX_TRANSACTION_WIDTH MAX_BURST_SIZE Dw CRC_EN	→ → → →	13 16 32 "NO"	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	jtag_uart	FPGA_VENDOR SIM_BUFFER_SIZE SIM_WAIT_COUNT	→ → →	"ALTERA" 1000 1000	uart	clk reset intrpt wb	→ → → →	source:clk source:reset NC bus:wb_slave[3]

Figure 7.1 illustrates the functional block diagram the Mor1k_tile module.
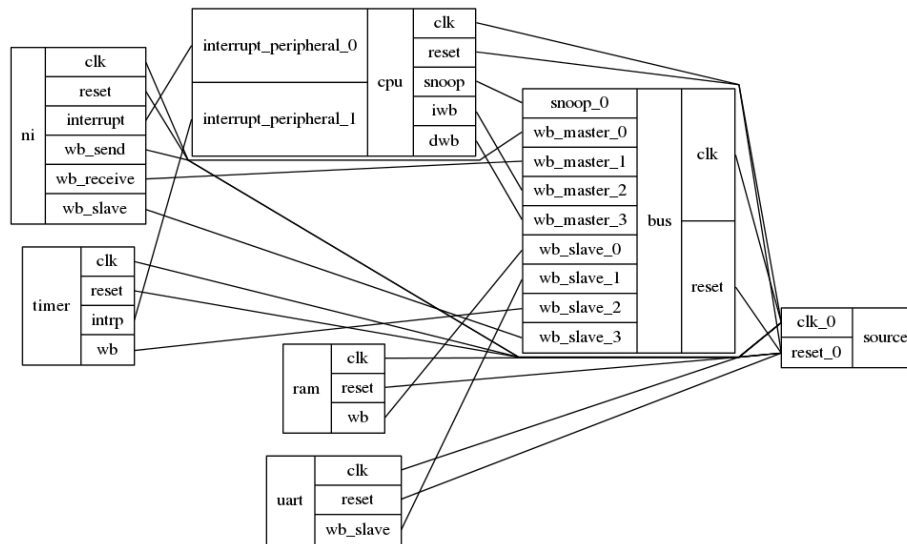
Figure 7.1: Mor1k_tile functional block diagram.

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"		
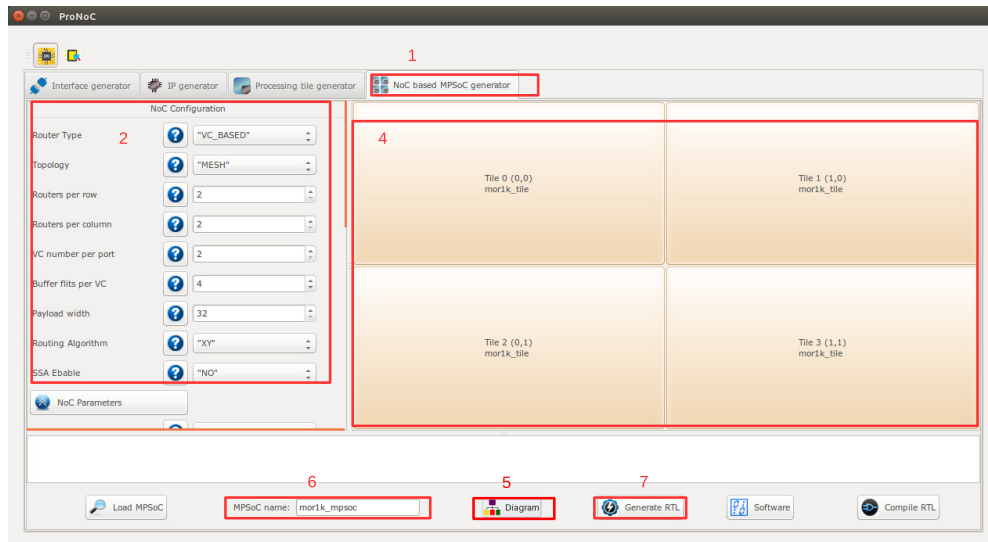
Figure 7.2: NoC-based MPSoC generator snapshot.

3. In `Tile configuration` setting, you should be able to see the list of all process-ing 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 sim-ply as `1:3`.
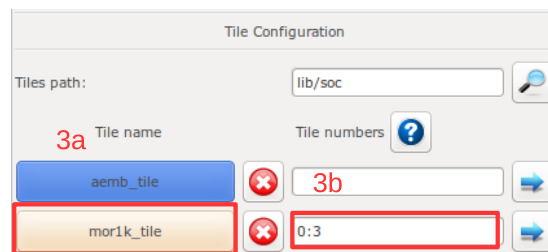
Figure 7.3: Tile Configuration snapshot.

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.
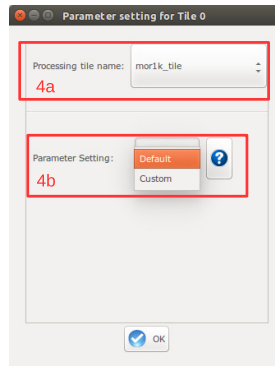
Figure 7.4: Custom Tile setting.

5. You can press the `Diagram` to see the actual location of each tile in the selected topology. In this example:
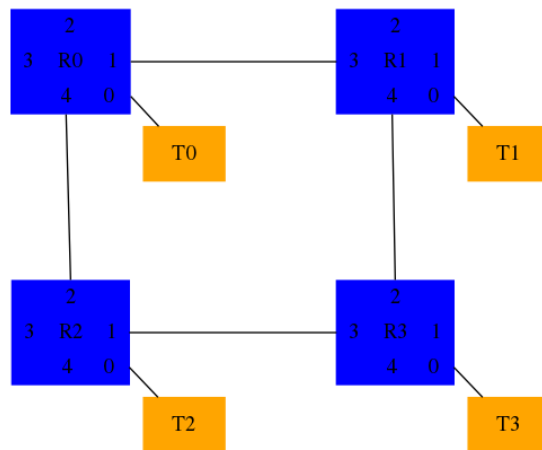
Figure 7.5: 4x4 mesh topology.

6. Set MPSoC name as `mor1k_mpsoc`.

7. Press `Generate RTL` button to generate the MPSoC RTL code.

Software Development

1. Press ⊞ `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 tile 0,1 and 2 with the following first and second C codes, respectively.

   In this example, tile 0 to 2 send each 3 packets to tile 3. The NI interrupt function is enabled in tile 3 which is responsible for packet collection from NoC and showing the packet content using serial port.

main.c file of tile 0,1 & 2

```c
//This code send packets to tile 3
#include "mor1k_tile.h"

unsigned char pck1[10]={"first data"};
unsigned char pck2[11]={"second data"};
unsigned char pck3[6]={"123456"};
unsigned char recive_buffer[15];

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

void error_handelling_function(){
    unsigned int i;
    for (i=0;i<ni_NUM_VCs;i++){
        if(ni_ERROR_FLAGS_REG(i)){
            printf ("Error in vc %u\n",i);
            if(ni_ERROR_FLAGS_REG(i) & BUFF_OVER_FLOW_ERR) printf ("
                The receiver allocated buffer size is smaller than
                the received packet size in core%u\n",COREID);
            if(ni_ERROR_FLAGS_REG(i) & SEND_DATA_SIZE_ERR) printf ("
                the send data size is not set in core%u\n",COREID);
            if(ni_ERROR_FLAGS_REG(i) & BURST_SIZE_ERR) printf (" the
                burst size is not set in core%u\n",COREID);
            if(ni_ERROR_FLAGS_REG(i) & ILLEGAL_SEND_REQ) printf( "A
                new send request is received while the DMA is still
                busy sending previous packet in core%u\n",COREID);
            if(ni_ERROR_FLAGS_REG(i) & CRC_MISS_MATCH) printf( "CRC
                missmatch in core%u\n",COREID);

        }
    }
}

unsigned int reseived_counter=0;
void got_packet_funtion(){
    unsigned int i;
    for (i=0;i<ni_NUM_VCs;i++){
        if(ni_got_packet(i)) {
            ni_receive (i, (unsigned int)recive_buffer, 4);
            reseived_counter++;
        }
    }
}

// NI interrupt function
void ni_isr(void){
    //place your interrupt code here
    if( ni_STATUS2_REG & ERRORS_ISR ){
```

```c
    // An error ocures
        error_handelling_function();
        ni_ack_errors_isr();
    }

    return;
}

int main(){
    int i;
    //void ni_initial (unsigned int burst_size,unsigned char
        send_int_en,unsigned char save_int_en,unsigned char
        got_pck_int_en)
    printf("Hi from core %u\n",COREID);
    int_init();
    int_add(0, ni_isr, 0);
    // Enable ni interrupt (its connected to inttruupt pin 0)
    int_enable(0);
    cpu_enable_user_interrupts();
    // hw interrupt enable function:
    // ni_initial ( burst_size, errors_int_en, send_int_en,
        save_int_en, got_pck_int_en)
    ni_initial (16,1,0,0,1); //enable the intrrupt when a packet is
        recived or when there is an error
    //ni_transfer (w, v, class_num, data_start_addr, data_size,
        dest_x, dest_y){
    ni_transfer (1,0, 0, (unsigned int)&pck1[0], 3, PHY_ADDR_ENDP_3);
    ni_transfer (1,1, 1, (unsigned int)&pck2[0], 3, PHY_ADDR_ENDP_3);
    ni_transfer (1,0, 0, (unsigned int)&pck3[0], 2, 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;
}
```

main.c file of tile 3

```c
//This code just recives packets and shows the packet contents
#include "mor1k_tile.h"

//if there is a data chach make sure the snoop protocol is supported
     by CPU to invalidate the
// the local copy of the recive_buffer once the NI update the main
    memory
volatile unsigned char recive_buffer[2][16];

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

void error_handelling_function(){
   unsigned int i;
   for (i=0;i<ni_NUM_VCs;i++){
         if(ni_ERROR_FLAGS_REG(i)){
             printf ("Error in vc %u\n",i);
             if(ni_ERROR_FLAGS_REG(i) & BUFF_OVER_FLOW_ERR) printf ("
                 The receiver allocated buffer size is smaller than
                 the received packet size in core%u\n",COREID);
             if(ni_ERROR_FLAGS_REG(i) & SEND_DATA_SIZE_ERR) printf ("
                 the send data size is not set in core%u\n",COREID);
             if(ni_ERROR_FLAGS_REG(i) & BURST_SIZE_ERR) printf (" the
                  burst size is not set in core%u\n",COREID);
             if(ni_ERROR_FLAGS_REG(i) & ILLEGAL_SEND_REQ) printf( "A
                 new send request is received while the DMA is still
                 busy sending previous packet in core%u\n",COREID);
             if(ni_ERROR_FLAGS_REG(i) & CRC_MISS_MATCH) printf( "CRC
                 missmatch in core%u\n",COREID);
      }
   }
}

unsigned int reseived_counter=0;
void got_packet_funtion(){
//this function sends packet saving command to the NI. It doese not
    wait until the packet saving is finished. Once the packet is
    completely saved, the software will be notified using
    SAVE_DONE_ISR intrrupt or it can check by using
    ni_packet_is_saved(v) function.
   unsigned int i;
   for (i=0;i<ni_NUM_VCs;i++){
      if(ni_got_packet(i)) {
         ni_receive (i, (unsigned int)(&recive_buffer[i][0]), 4);
            reseived_counter++;
      }
   }
}
```

```c
void check_packet_funtion(){// in this example we just print the
     packet content
    unsigned int i,size,j;
    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);
            printf("A message of %u words is recived from core (%x) in
                 vc%u:", size,src_info.addr,i);
            for (j=0;j<size*4;j++){
                    printf("%c", recive_buffer[i][j]);
            }
                printf("\n");
        }
    }
}

// NI interrupt function
void ni_isr(void){
    //place your interrupt code here
    if( ni_STATUS2_REG & ERRORS_ISR ){
    // An error ocures
        error_handelling_function();
        ni_ack_errors_isr();
    }
    if( ni_STATUS2_REG & SAVE_DONE_ISR ){
    //check which VC has finished saving the packet. This function
         must be called before got_packet_funtion
        check_packet_funtion();
        ni_ack_save_done_isr();
    }
    if( ni_STATUS2_REG & GOT_PCK_ISR ){
    //check which VC got a packet and send the save command to NI to
         start saving the packet.
        got_packet_funtion();
    //Please note that the whole of the packet may not yet be in the
         memory when the code reaches here. Once the packet is
         completely saved the software will be notified using
         SAVE_DONE_ISR flag
        ni_ack_got_pck_isr();
    }
    return;
}

int main(){
    int i,j;
    unsigned int send_counter=0;
    printf("Hi from core %u \n",COREID);
    int_init();
    int_add(0, ni_isr, 0);
    // Enable ni interrupt (its connected to inttruupt pin 0)
    int_enable(0);
    cpu_enable_user_interrupts();
    // hw interrupt enable function:
```

```
    // ni_initial (burst_size, errors_int_en, send_int_en,
        save_int_en, got_pck_int_en)
    ni_initial (16,1,0,1,1); //enable the intrrupt when a packet is
        recived, saved or got any error
    delay(2000+COREID);
    printf("total received packets by core%u is %u\n",COREID,
        reseived_counter);
    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 suc-
   cessfully" message. Otherwise, check the error message to fix your code and
   press the compile button again. 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

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 8.1.

2. Click on 🔳 to open ProNoC simulator tabs.

3. Click on `NoC Simulator` tab to open simulator GUI interface:
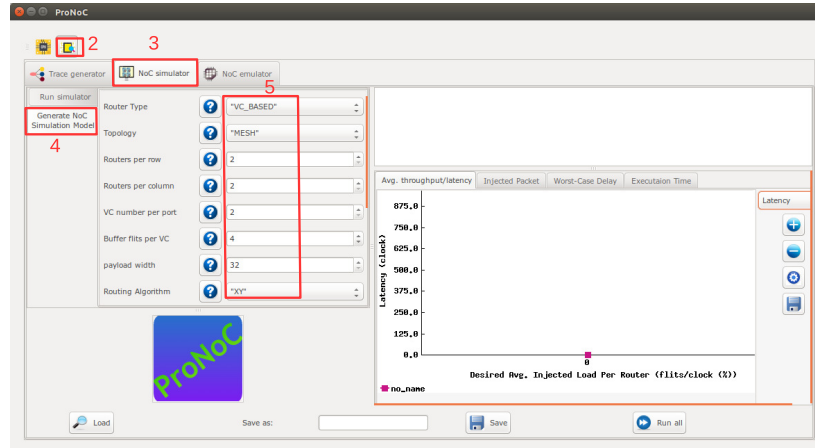
Figure 8.1: NoC simulator snapshot.

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.
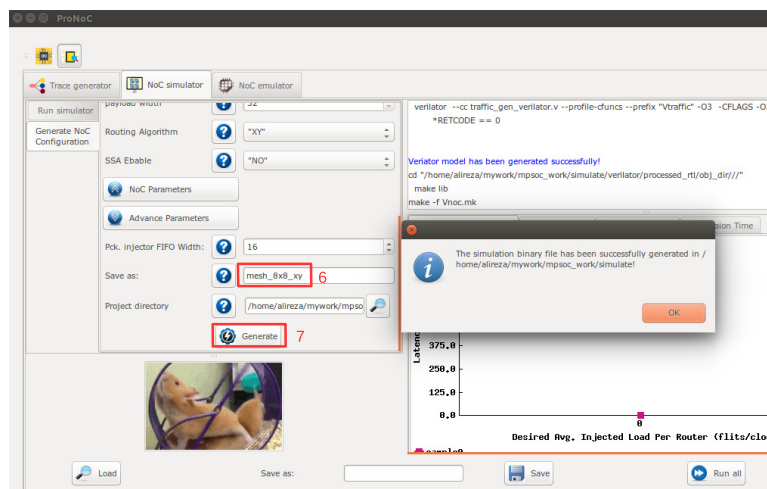
Figure 8.2: Generate NoC simulation model.

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		
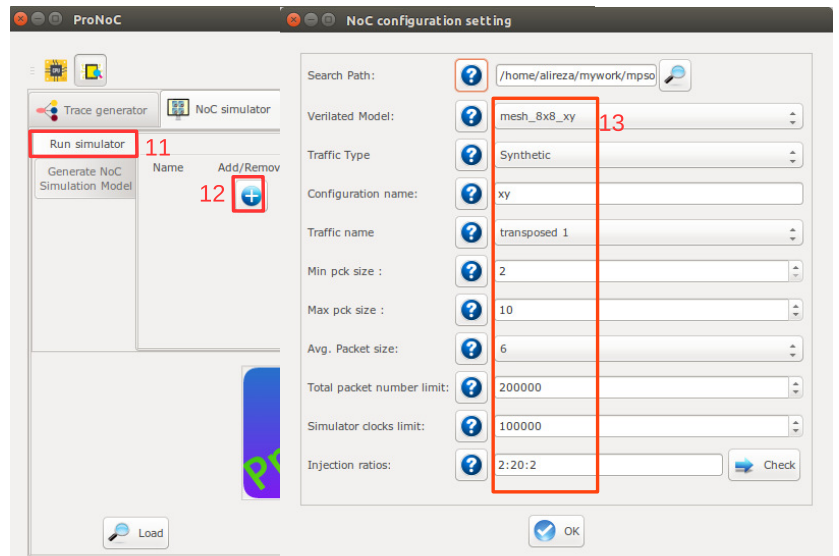
Figure 8.3

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.
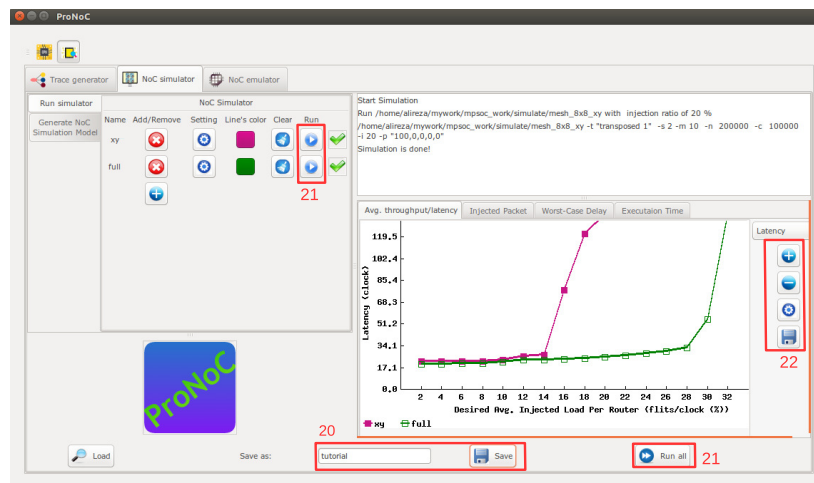
Figure 8.4

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
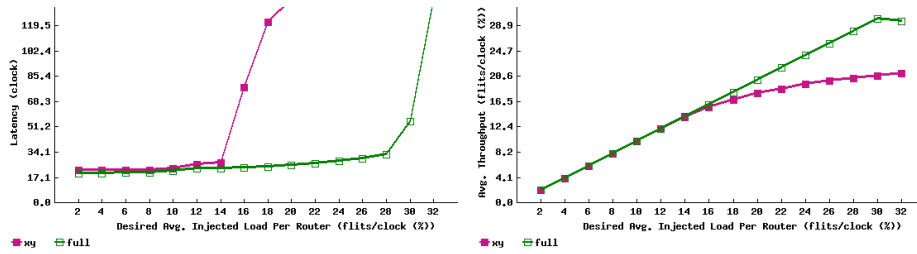
(a) Load-latency

(b) Load-throughput
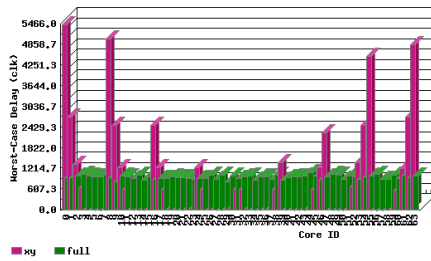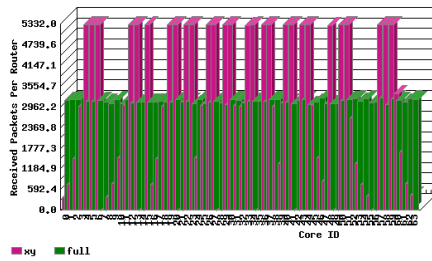
(c) Injected packets per router at 32% injection ratio.

(d) Worst-case delay per router at 32% injection ratio.
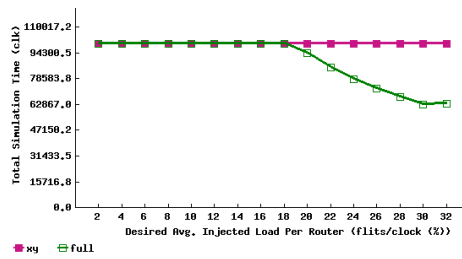
(e) Simulation time in clock cycles.

Figure 8.5: Simulation sample results.

CHAPTER 9

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 Adding a New Altera 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 9.1.

2. Click on ![icon] to open ProNoC simulator tabs.

3. Click on `NoC Emulator` tab to open the emulator GUI interface:
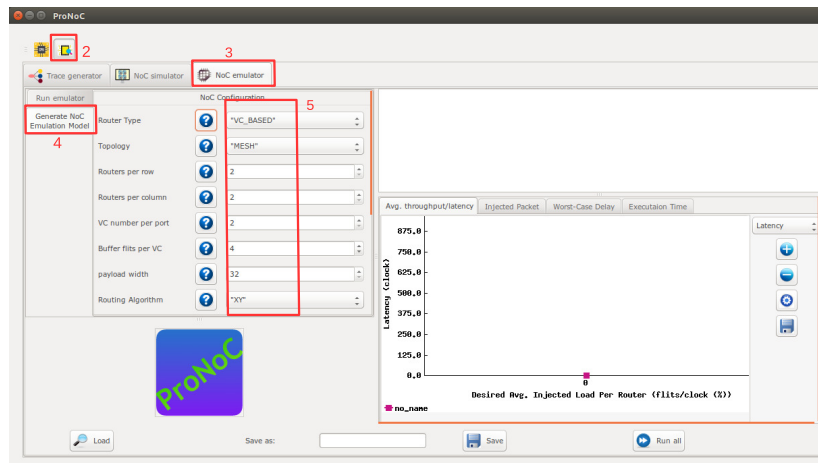
Figure 9.1

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.
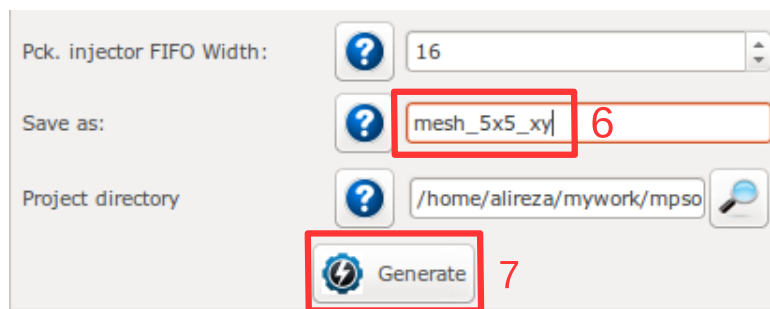
Figure 9.2: Generate NoC model

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 9.3.
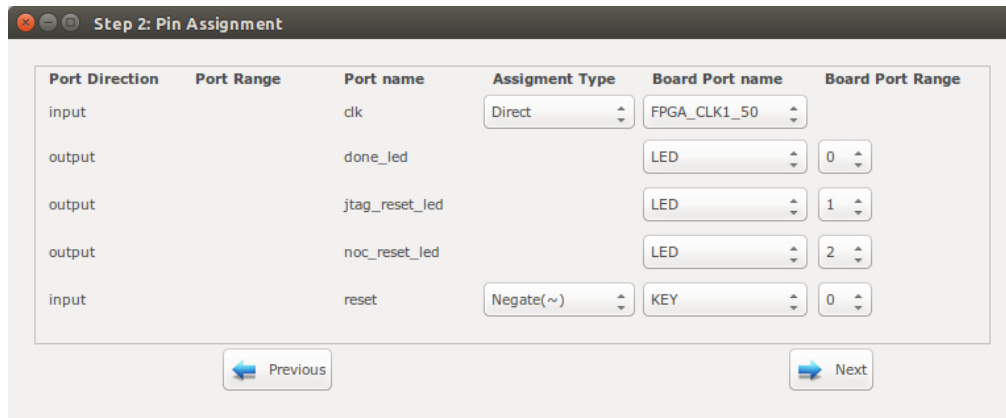
Figure 9.3: DE10-Nano FPGA board pin assignment.

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.
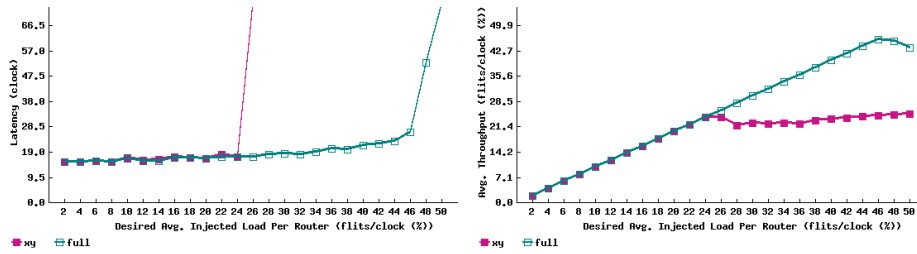
(a) Load-latency

(b) Load-throughput
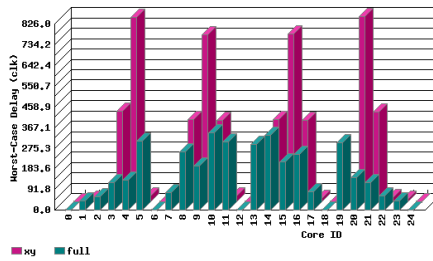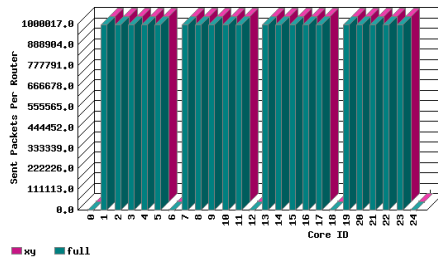
(c) Injected packets per router at 50% injection ratio.

(d) Worst-case delay per router at 50% injection ratio.
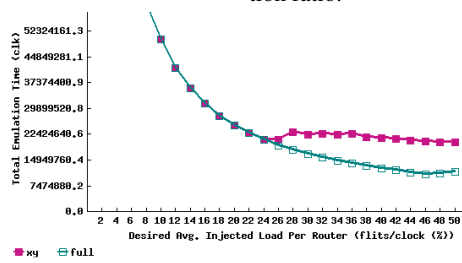
(e) Emulation time in clock cycles.

Figure 9.4: Emulator sample results.

Appendices

NoC Verilog File Parameters Description

V	$\mathtt{V} \in \mathbb{N}, \mathtt{V} \geqslant 1.$	Number of VC per router port. Defining V as 1 results in a simple non-VC based router.
B	$\mathtt{B} \in \mathbb{N}, \mathtt{B} \geqslant 2$	Buffer size per VC in flit .
TOPOLOGY	`"MESH"` `"TORUS"` `"RING"` `"LINE"` `"FATTREE"` `"TREE"`	The NoC topology.
T1,T2,T3,T4	$\mathtt{T1},\mathtt{T2},\mathtt{T3},\mathtt{T4} \in \mathbb{N}$	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 ∈ 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 ∈ ℕ, Fpay ⩾ 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", "BASELINE"	VC/SW combination type. Note that using "BASELINE" is not recommended.
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.
CONGESTION_INDEX	CONGESTION_INDEX ∈ ℕ, 0 ⩽ CONGESTION_INDEX ⩽ 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.
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.
WEIGHTw	WEIGHTw ∈ ℕ, $2 \leqslant$ WEIGHTw $\leqslant 7$	WRRA weights' maximum width in bits.

MIN_PCK_SIZE	MIN_PCK_SIZE $\in \mathbb{N}$, 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.

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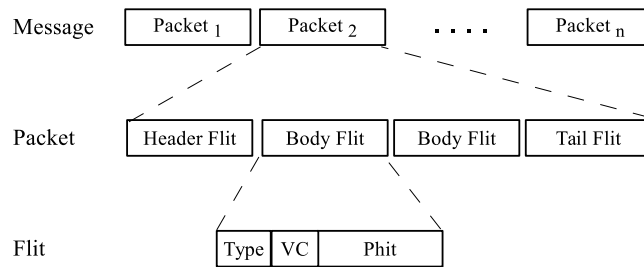
Figure B.1: Units of resource allocation in NoC.

Flit type	Type is 2-bit signal indicate the type of the respective flit (header,body or tail). The first bit is asserted for header flits. The second flit is asserted if the flit is 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.
Header flit 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
DSTPw	EAw	EAw

Size (bits)

Fpay

header-data	weight	class
HDw	Ww	Cw

Size (bits)

| 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.2: 2×2 CMESH endpoint coding example.

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

$$NXw = \log 2NX;$$
$$NYw = (NY > 1)? \ \log 2NY : 0;$$
$$NLw = (NL > 1)? \ \log 2NL : 0;$$
$$EAw = NLw + NYw + NXw; \qquad \text{(B.1)}$$

where NX is the maximum number of routers in first dimension, NY is maximum number of router in second dimension,NL is the 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 nth layer which can receives a packet from that endpoint (note that n_{root}=0). Figure B.3 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.
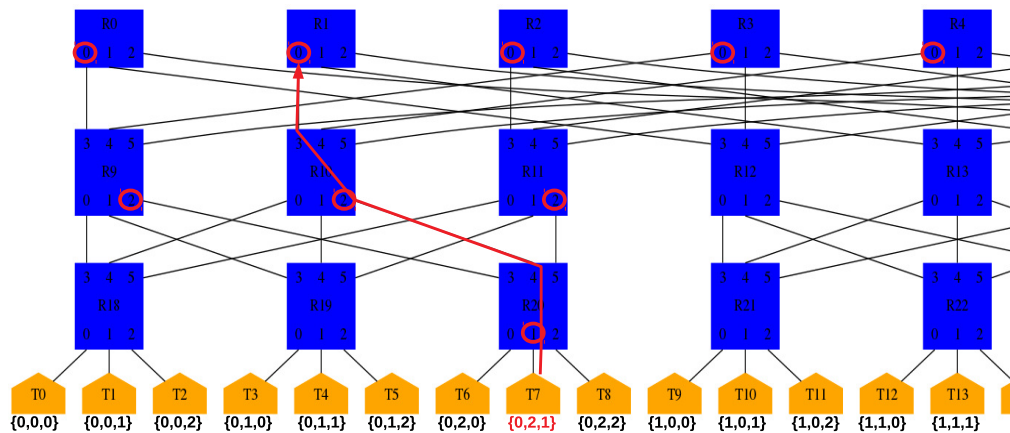
Figure B.3: Fattree (k=3,l=3) endpoint coding example.

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

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 9.1.

Table 9.1: 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.

$$Cw = (C > 0)?\ \log 2C : 0 \qquad (9.3)$$

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.

$$Ww = (\texttt{SWA\_ARBITER\_TYPE} == \texttt{"WRAA"})?\ \texttt{WEIGHTw} : 0 \qquad (9.4)$$

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):

$$HDw = Fpay - 2EAw - DSPTw - Ww \qquad (9.5)$$