

## **ProNoC**

### **User Manual**

### Copyright ©2014-2018 Alireza Monemi

This file is part of ProNoC

ProNoC (stands for Prototype Network-on-Chip) is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 2 of the License, or (at your option) any later version.

ProNoC is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with ProNoC. If not, see <a href="http://www.gnu.org/licenses/">http://www.gnu.org/licenses/</a>>.

This document may include technical inaccuracies or typographical errors.

## **Contents**

1	Inst	allation Manual for the Ubuntu Linux Environment	3
	1.1	Installation	3
2	Inte	rface Generator	6
	2.1	Introduction	6
	2.2	Generate New Interface	8
	2.3	Defined Interfaces	8
		2.3.1 interrupt_cpu	9
		2.3.2 interrupt _peripheral	9
		2.3.3 clk	10
		2.3.4 reset	10
		2.3.5 Enable	10
		2.3.6 Wb_master	10
		2.3.7 Wb_slave	11
3	IP (	Senerator	12
	3.1	Introduction	12
	3.2	Generate a New IP	12
	3.3	List of available Variables in ProNoC	16
	3.4	List of available IP cores in ProNoC	17
		3.4.1 Bus	17
		3.4.2 Communication	17
		3.4.3 DMA	17
		3.4.4 Display	17
		3.4.5 GPIO	17
		3.4.6 Interrupt	17
		3.4.7 NI	18
		3.4.8 Processor	18
		3.4.9 RAM	18
		3.4.10 Source	18
		3.4.11 Timer	18
4	Pro	cessing Tile Generator	19

1

### 4 Processing Tile Generator

5	Pro	cessing Tile Generator Hello World Tutorial	21
	5.1	System Requirements:	21
	5.2	Objectives:	21
	5.3	Desired SoC	21
		5.3.1 Schematic	21
		5.3.2 Application Software	22
	5.4	Create New SoC Using ProNoC Processing Tile Generator	22
	5.5	Software Development	33
	5.6	Simulate the generated RTL code using Modelsim software	36
	5.7	Simulate the generated RTL code using Verilator software	38
	5.8	Compile the generated RTL code using Quartus II software	42
6	Add	Custom IP to Processing Tile Generator Tutorial	46
	6.1	System Requirements:	46
	6.2	Objectives:	46
	6.3	Greatest Common Divisor (GCD) Algorithm	46
	6.4	GCD RTL code	47
		6.4.1 GCD Simulation	50
	6.5	Add Wishbone bus interface to GCD	54
	6.6	Add custom wishbone-based IP core to ProNoC Library	57
	6.7	Generate a new SoC enhanced with new IP core (GCD)	64
	6.8	Software Development	68
7	NoC	<b>C Verilog File Parameters Description</b>	71
8	NoC	Simulator	74
	8.1	System Requirements:	74
	8.2	Simulation Example:	74
		8.2.1 Generate first NoC simulation model with XY routing	74
		8.2.2 Generate the second NoC simulation model with fully adaptive	
		routing	75
		8.2.3 Run simulation under Matrix Transposed traffic pattern	75
9	NoC	Emulator	79
	9.1	summary	79
	9.2	System Requirements	79
	9.3	Emulation Example:	79
		9.3.1 Generate first NoC emulation model with XY routing	79
		9.3.2 Generate the second NoC emulation model with fully adaptive	
		routing	81
		9.3.3 Run Emulation models under Matrix Transposed traffic pattern	81

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-lowlatency-noc-based-mpsoc/an-fpga-implementation-of-low-latencynoc-based-mpsoc/trunk

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

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
```

#### 4. Install required Perl modules:

```
sudo cpanm ExtUtils::Depends
sudo cpanm ExtUtils::PkgConfig
sudo cpanm Glib
sudo cpanm Pango
sudo cpanm Gtk2
sudo cpanm Gtk2::Ex::Graph::GD
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. Add mpsoc\_work path to the PATH variable in .bashrc file:

gedit ~/.bashrc

Add PRONOC\_WORK variable to .bashrc file then save and close it:

export PRONOC\_WORK={path\_to\_mpsoc\_work\_directory}
# e.g export PRONOC\_WORK=/home/alireza/Mywork/mpsoc\_work

Now run the following command in terminal to update the variables

```
ProNoC homepage
```

source ~/.bashrc

6. Install Verilator simulator.

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

7. Add QUARTUS\_BIN variable to .bashrc file. (This stage is optional, do it only if you are going to use Altera FPGAs for implementation or emulation):

```
export QUARTUS_BIN={path_to_quartus_bin_directory}
#e.g. export QUARTUS_BIN=/home/alireza/intelFPGA_lite/17.1/quartus
    /bin
```

8. Add MODELSIM\_BIN variable to .bashrc file. (This stage is optional, do it if you have installed Modelsim simulator and you want ProNoC to auto-generate the simulation models using Modelsim software):

```
export MODELSIM_BIN={path_to_Modelsim_bin_directory}
#e.g. export MODELSIM_BIN=/home/alireza/altera/modeltech/bin
```

Now run the following command in terminal to update the PATH variable

source ~/.bashrc

- 9. Download soft-core processors' GNU toolchain:
  - (a) aeMB
  - (b) Lm32 or from Lm32
  - (c) orlk-elf for morlk and orl200 OpenRISC CPUs.

Unzip the files and copy them in mpsoc\_work/toolchain directory:

```
mv lm32 mpsoc_work/toolchain/lm32
mv aemb mpsoc_work/toolchain/aemb
mv morlk mpsoc_work/toolchain/morlk
```

10. Give execution permission to GNU toolchains. Open terminal in mpsoc\_work/toolchain and run

sudo chmod +x -Rf ./

11. Open /mpsoc/src\_c in terminal and run

make

12. Now you can run the GUI application by

cd mpsoc/perl\_gui ./ProNoC.pl



Figure 1.1: 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 *socket* or *plug* 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.

Select soket type:	single connection	0
	multi connection	
< swap	📂 plug	

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:



```
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 ,
  •
  .
```



Figure 2.3: (a) Select Verilog parameters  $\mathbf{M}$  and  $\mathbf{S}$  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.

Select module:	dual_port_rar	n	: 🐣 Impor	l Ports	Select Category:	memory	1 Description
Interface name:		😑 🗇 💿 Import P	orts			ingle connection	:
			Select the ports incl	uded in the interface		socket	
Type		Type	Range	Name	Select	Name	
Input	; (Aw-1):0					addr_a	
Input	; (Dw-1):0	Input	[(Aw-1):0]	addr_a	<b>S</b>	data_a	
output	: (Dw-1):0	Input	[(Aw-1):0]	addr_b		a a	
innit		input		dk	_	we a	
		input	[(Dw-1):0]	data_a		1.00	
		input	[(Dw-1):0]	data_b	_		
		output	[(Dw-1):0]	q_a			
		output	[(Dw-1):0]	q_b			
		input		we_a			
		input	o at	we_p			
			UK				
		Input	💽 ok	we_b			

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.

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

interruptThis is the interrupt interface connection between CPUs having multiple interrupt pins\_peripheralthat 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.

The clock pin interface.

clk



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 <i>prowse</i> 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 mod- ule in Select Module field.
	5. Using <b>1</b> P 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.
	<b>Note:</b> In order to make the copy of your ProNoC software portable palace the documentation files somewhere inside mpsoc folder.
	<ul> <li>6. The h Add Software files button allows the addition of the necessarily files and folders to the generated processing tile software directory (mpsoc/SOC/[PT-name]/sw). By pressing this button you will have three notebook pages:</li> </ul>
	• 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  $f_{00}$  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 button. All files listed here will be copied in the generated processing tile inside mpsoc/SOC/[PT-name]/src\_verilog folder.
- 8. By pressing **O** 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 🕜	Туре 🕜	0	info 🕜	add/remove
М	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 predefined 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.

- 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,

ProNoC homepage

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.

8	😑 🗉 Interface pa	iramete	0xa100_0000	0xa1ff_ffff	HDLC Controller	
			0x9100_0000	0x91ff_ffff	General-Purpose I/O	
	interface name		0x9600_0000	0x96ff_ffff	PWM/Timer/Counter Ctrl	block address width 😮
			0xa400_0000	0xa4ff_ffff	Digital Camera Ctrl	
	wb		0xa600_0000	0xb7ff_ffff	Reserved1	Fixed 2 32 Bytes 5
			0xa500_0000	0xa5ff_ffff	Debug	
			0x9300_0000	0x93ff_ffff	Memory Controller	
			0xa300_0000	0xa3ff_ffff	I2C Controller	
			0.0000 0000	0.000 000 000 1		

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 ) 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 **(3)** Generate you can generate the IP. You can also modify the existing IPs by using *(p)* Load IP button.

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

ProNoC homepage

ProNoC				
Interface generator	P generator 🛛 🥞 Processing tile generator	NoC based MPSoC generator		
Interfaces list	Exact flag	s/crs. pariphara//hus/wishhapa, hus v		hus
<ul> <li>interrupt</li> </ul>	Select nie. [/nonre/aii/eza//riywork/mpso	c/sic_peripreta/bus/wishbone_bus.v	P browse IP name. Wishbolie	_bus
<ul> <li>source</li> <li>wishbone</li> <li>wb_addr_map</li> </ul>	Select wishbone_bus : Paran settle	ng Select Bus Category:	• O IP Description	dd Software files files
wb_master wb_slave	Interface name Type	Interface Num		
	wb_master socket		Concatenate	Remove
	wb_slave socket to socket		Concatenate	Remove Remove
	Type Port name	Interface name	Interface port	Port Range
	output s_adr_o_all	socket:wb_slave \$	adr_o ‡ Aw*S-1 :	0
	output s_dat_o_all	eoclotuub elavo *	dat_o ‡ Dw*S-1 :	0
	output s_sel_o_all	socket:wb_master	sel_o ‡ SELw*S-1	0
	output s taq o all	socket:wb_slave	taq o 🏥 TAGw*S-1	: 0
lease select the verilog file conta	inig the ip module	socket:wb_addr_map plug:reset	Community of the second	

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:

$$CORE_{-ID} = ((y * NX) + x)$$
(3.1)

where (x,y) are the node location in x and y axes and NX is the number of node in x dimension. If the generated tile is used as top level module <code>CORE\_ID</code> 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 read- ing/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 us- ing mpsoc/src_c/jtag/jtag_libusb via USB Blaster I and mpsoc/src_c/jtag /jtag_quartus_stp via USB Blaster II.
DMA	
	• <b>dma</b> : 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.
	• <b>int_ctrl</b> : 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.

ľ	V	]	[

NI	
·	<b>ni_master</b> : 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.
·	<b>ni_slave</b> : 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	
	• <b>Or1200:</b> OR1200 is the original implementation of the OpenRISC 1000 architecture. Its source code has been adopted from github at openrisc/or1200.
	<b>aeMB:</b> the EDK3.2 compatible Microblaze core. This IP core is adopted from OpenCores/aemb.
	<b>Im32:</b> LatticeMico32 is a soft processor originally developed by Lattice Semi- conductor. The source code of this IP core is adopted from github/soc-lm32.
	<b>mor1kx</b> : The mor1kx is a replacement for the original or1200 processor. The source code is adopted from github at openrisc/mor1kx
RAM	
	• <b>single_port_ram</b> : A Wishbone bus-based single port Random Access Memory (RAM).
•	dual_port_ram: A Wishbone bus-based dual port RAM.
Source	
	• <b>clk_source</b> : 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.

💉 Interface generator 🛛 🌵 IP generato	or 🕞 Processing tile generator	NoC based MPSoC generator			
IP list	mor1kx	Instance name	clk	ss:clk	
Bus     Communication	Setting	cpu	enable	ю	
▶ DMA					
Display     GPID	Remove		reset	ss:reset 2	
▶ Interrupt			iwb	bus:wb_master[0]	
NoC     Other			dwb	[bus:wb_master[1]	
Processor	gpo	Instance name	clk	ss:clk	
KAM     Source	Setting	led	reset	ss:reset	
Timer			_		_
Parameter setting for Wishbo	Remove		wb 🔍	Step 1: Select Compiler	
Parameter name Value Description	single port ram	Instance name	clk	Compiler tool Quartus	sII
м 2 🗘 😮	Wishbone slave porc addres	20 Secting		Verilato	or
s 3 🗘 😯	Instance name Interface name Bus nar	me Base address End	address	Modelsi	im
Dw 32 🗘 🕐	0: led wb bus	0x91000000 0x9100001	r q	Quartus bin: //home/alireza/intelFPGA_lit	te, 🔎
Aw 32 1	1: sim_uart wb_slave bus	0xa5000000	1		
		[0.0000000]			
🗹 ок 3	2: ram wb bus	0x0000000		=	Next
//home/alliesza/mywork/mpsoc_work/SOC/m Makefile     RAM	nor1				) RegE
/home/allreza/mywork/mpsoc_work/SOC/m Makefile RAM README	nor1 2 #include "mor1k_soc.h" 3 // a simple deby function 4 void deby (unsigned leby function	х			) RegE
/home/alireza/mywork/mpsoc_work/SOC/m Makeflie     RAM     README     defice_printLh     image	nor1 2 #indude "mor1k_soc.h" 3 // a simple deby function 4 void deby (unsigned int rum ) 5 white (num >0)( 7 mum-:	х			] RegE
//home/allireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_print.h image image.hex image.het	nor1 2 #indude "mor1k_soc.h" 3 / a simple deby function 4 void deby (unsigned int.um) 5 while (num>)( 7 um <sup>-</sup> ; 8 nop(); // sam voistile ("nop"); 9	х			] RegE
//home/allireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_print/.h image image.ikex image.ikex image.ap	nor1 2 #indude "mor1k_soc.h" 3 / a simple deby function 4 void deby (unsigned int.um) 5 while (num>0)( 7 uum~; 8 nop(); // sam voistile ("nop"); 10 return; 11	х:			) RegE>
//home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_printf.h image image.ikex image.ist image.j.itfc.sh link.id	nor1 2 #indude "mor1k_soc.h" 3 / a simple deby function 4 void deby (unsigned int.um) 5 while (num>0)( 7 uum~; 8 nop(); // sam voistile ("nop"); 10 return; 11 12 } 13 char i=0; 14 / 22 / 22 / 22 / 22 / 22 / 22 / 22 /	х			] RegE
//home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_printf.h image image.hex image.hex image.ist image.intc.sh limate.	<pre>nor1</pre>	х			) RegE
/home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_print/.h image_ image_ihex image_ist imag	<pre>nor1</pre>	X ; ;,			) RegE>
//home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_print/.h image image.ikex ima	nor1 2 #include "mor1k_soc.h" 3 // a simple deby function 4 void deby (unsigned int rum ) 5 while (num>0){ 7 num-; 8 nop(); // aum voiatile ("nop"); 10 return; 11 12 bhar i= "0; 13 char i= "0; 14 dir th main(){ 15 while(1){ 17 viet1; 18 man_uar_putsting ("helloh,n") 19 deby(100); 20 sim_uar_putsting ("helloh,n") 20 sim_uar_putsting ("helloh,n") 20 sim_uar_putsting ("helloh,n") 20 sim_uar_putsting ("helloh,n")	X ; ; ; ;			] RegE:
<pre>//home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_print/h image_ image_ihex image_ist imade_ist image_ist image_ist image_ist imade_ist image_ist imade_ist image_ist imade_ist image_ist imade_ist ist ist ist ist ist ist ist ist ist</pre>	<pre>mor1</pre>	X ; ; ; ; ; ; ; ;			] RegE:
<pre>//home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_print/h image_ image_ihex image_ist imade_ist image_ist imade_ist image_ist imade_ist ist imade_ist ist ist ist ist ist ist ist ist ist</pre>	<pre>mor1</pre>	X ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;			] RegE:
<pre>//home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_print/.h image_ image_ihex image_ist imade_ist image_ist image_ist imade_ist image_ist image_ist image_ist imade_ist image_ist image_ist imade_ist image_ist imade_ist image_ist imade ist imade ist imade ist imade ist imade ist imade ist imade ist imade ist imade ist imade ist imade ist ist ist ist ist ist ist ist ist ist</pre>	<pre>mor1</pre>	X ; **8); );			RegE:
<pre>//home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_print/h image_ image_ihex image_ist imade_ist image_ist imade_ist image_ist imade_ist ist imade_ist ist ist ist ist ist ist ist ist ist</pre>	<pre>mort</pre>	X( ; ; ;; ;; ;; ;; ;; ;; ;; ;;			] RegE:
<pre>//ome/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_printf.h image_ image_hex image_hex image_hex image_hex image_ist imade_ist imade_ist imal.c im</pre>	<pre>mort</pre>	X ; ;(8); );			] RegE:
//home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_printf.h image image.hex image.hex image.hex image.nep jtag_intfc.sh imain.c mortk_soc.h > mortks program.sh > simple-printf write_memory.sh	<pre>mort</pre>	X( ; *(8); );			) RegE
//home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_printf.h image image.hex image.hex image.hex image.hex image.intc.sh image.intc.sh imain.c mortk_soc.h > mortks program.sh > simple-printf write_memory.sh	mor1	X( ; *,6); ); pe > image.ist			) RegE>
//ome/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_print/h image.inex	mor1	x( ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;			] RegEx
//tome/alireza/mywork/mpsoc_work/SOC/m Makefile RAM README define_print/.h image.inage.inage image.inage.inage image.inage.inage.inage.inage.inage.inage.inage.inage.inage.inage.inage.inage.inage.ina	mor1	x( ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;			] RegE>
//home/alireza/mywork/mpsoc_work/SOC/m Makefile RAM RAM RADME define_print/h image_ina	mor1	<pre>&gt; image.ixt ge image.ixt ge image.ixt guinage.inex Miran0.mif bin</pre>			) RegEx

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 <b>ProNoC Processing Tile Genera-</b> <b>tor</b> . 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 In- stallation 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.
<b>Objectives:</b>	
	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 mod- ule.
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.

Interface generator	IP generator	Processing tile generator	NoC based MPSoC generator	
Select file:				Browse Description
Select	module:		Import Ports	Select Category:
Inter	ace name:		Se	elect soket type: single connection 🛟
lease select the verilog f	le containig the interfa	ace		
🔎 Load	Interface		(	Generate

Figure 5.2: ProNoC GUI first page snapshot.

Then select the *Processing Tile Generator* **Processing tile generator**:

Interface generator	F IP generator Processing tile generator
P list	
Bus	
Communication	
DMA	
Display	
GPIO	
Interrupt	
NoC	
Other	
Processor	
RAM	
Source	
clk_source	
Timer	

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 10.

	-		
P list	clk_source	Instance name	clk IO
Communication		alk course	recet IO
DMA			reset 10
Display	Remove		
GPIO			
Interrupt			
NoC			
Other			
Processor			
RAM			
Source			
clk_source			
Timer			

Figure 5.4: Adding clock source.

### Add Wishbone Bus:

- 1. Click on  ${\tt Bus}\xspace$  category and double click on  ${\tt Wishbone\_bus}.$
- 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.

F Interface generator	IP generator	Processing tile general	tor Soc based MPSoC generator		
P list		clk_source	Instance name	clk	10
wishbone_bus 1		Setting	source	reset	I0 ;
DMA Display		Remove			
GPIO		wishbone_bus	Instance name	clk	source:clk ;
NoC	🖲 🖲 Parameter	setting for wishbone_b	3 bus	reset	source:reset
Other Processor	Parameter name	Value Description		4	
RAM	м	2 🗘 🕄			
Source					
Timer	S	4 🗘 😯			
ishbone bus	Dw 2	32 🗘 💡			
	A	32 + 🗿			

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]

S S ProNoC	ah - shi			
= 🙀 ·				
💉 Interface generator 🛛 🐗 IP	generator 🛛 🧊 Processing tile generator	NoC based MPSoC generator		
IP list Bus	Remove			
Communication     DMA	wishbone_bus	Instance name	clk	source:clk
<ul> <li>Display</li> <li>GPIO</li> </ul>	Setting	bus	reset	source:reset
gpi gpio	Remove			
gpo 1 • Interrupt	gpo	3 Instance name	clk 4	source:clk
<ul> <li>NoC</li> <li>Other</li> </ul>	Setting	led	reset	source:reset
😣 🖱 🖲 Parameter setting for g	po Remove		wb 5	bus:wb_slave[0]
Parameter name Value D PORT_WIDTH	escription			
ОК 2	Wishbone-	bus addr Diagram 🧭 Generate	RTL Softw	are Ocompile RTL

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  ${\tt Processor}$  category and then double click on  ${\tt morlkx}.$
- 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.

Pliet	gpo Processing the general		Courseren
Interrupt	Setting led	reset	source:reset
NoC Other	Remove	wb	bus:wb_slave[0]
Altor	mor1kx Insta	nce name clk	3 source:clk
Or1200 aeMB	Setting Cpu	enable	4 10
Im32	Remove 2	reset	5 source: reset
mor1kx RAM		iwb	6 bus:wb_master[0]
Source Timer		dwb	bus:wb_master[1]
Source Timer		dwb	bus:wb_master[1]

Figure 5.7: Adding Processor.

### Add Timer:

- 1. Click on  ${\tt Timer}$  category and then double click on  ${\tt timer}.$
- 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].

Interface generator	IP generator Processi	ng tile generator	NoC based MPSoC g	enerator	
list Bus	Remove		reset		source:reset
DMA			iwb		bus:wb_master[0]
Display			dwb		bus:wb_master[1]
Interrupt NoC	imer 🙋	Instance name	clk	3	source: clk
Other Processor	Setting	timer	intrp	4	cpu:interrupt_peripheral[0]
RAM Source	Remove	2	reset	5	source:reset
timer 1			wb	6	bus:wb_slave[1]
simple, general purpose, V	Wishbone bus-based, 32-bit timer.	111			

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].

Thterface generator	🖡 IP generator 🏾 🍘 Proce	ssing tile generator	NoC based MPSoC generator		
IP list ▶ Bus ▼ Communication	Setting	timer	intrp		cpu:interrupt_peripheral[0]
ethmac_100 jtag_uart 1	Remove		reset		source: reset
jtag_wb			wb		bus:wb_slave[1]
Display	jtag_uart	2 Instance name	clk	3	source: clk
GPIO Interrupt	Setting	uart	interrupt_peripheral	4	NC
NoC Other	Remove		reset	5	source: reset
Processor RAM			wb_slave		IO
he Altera JTAG UART core wit	h Wishbone bus interface.				bus:wb_slave[0]->gpo0:wb_slave[0]
				6	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 content 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 Modelsim 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].

Interface generate	pr 👘 IP generator	Processing tile generator	NoC based MPSoC generator			
P list	Second Parameter Settin	g for single_port_rain		reset		source:reset
Bus Communication	Parameter name	Value		wb		bus:wb_slave[1]
DMA Display	Aw	2 14	Instance name	clk		source:clk
Interrupt NoC	BYTE_WR_EN	"YES" ‡	uart	interrupt_peripheral		NC
Other Processor	FPGA_VENDOR	3 altera"		reset		source:reset
RAM dual_port_ram	JTAG_CONNECT	4 "JTAG_WB"		wb_slave		bus.wb_siave[2]
single_port_ram Source	JTAG_INDEX	CORE_ID	6	CIK .	7	source: cik
Timer	BURST_MODE	"ENABLED" ‡	Iram	reset		source:reset
_	MEM_CONTENT_FILE_NAME	"ram0"		wb	8	bus:wb_slave[3]
	INITIAL_EN	5 "YES" ‡	11			

Figure 5.10: Adding Single port RAM.
Check wishboneAfter adding all required IP cores, now you can check the auto-assigned Wishbone busbus(es)addresses by clicking on Wishbone-bus addr button. Note that the assigned addressesaddresses:are also modifiable.

😣 🗏 🔍 Wishbone slave	port address setting				
Instance name	Interface name	Bus name	Base address	End address	Size (Bytes)
0: led	wb	bus	0x91000000	0x9100001f	32
1: uart	wb_slave	bus	0x900þ0000	0x9000001f	32
2: ram	wb	bus	0x0000000	0x0000ffff	64 к ⊘
3: timer	wb	bus	0x96000000	0x9600001f	32
		Revert	💽 ок		

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.



Figure 5.12: The tutorial SoC diagram.

## **Generate SoC**

RTL Code:

- 1. Set Tile name as tutorial.
- 2. Press Generate RTL button.

······································	Frocessing the generator		
t		wb	bus:wb_slave[1]
is mmunication	jtag_uart Instance name	clk	source: clk
1A splay	Setting uart	interrupt_peripheral	NC
чЮ terrupt	8	reset	source:reset
DC her	Processing Tile "tutorial" has been created successfully at /home/alireza/mywork/mpsoc_work/ SOC/tutorial/.	wb_slave	bus:wb_slave[2]
ocessor M		clk	source:clk
urce ner		reset	source:reset
	Remove	wb	bus:wb_slave[3]
e port ram with wishbone bus interface.			

Figure 5.13: Generating the tutorial SoC.

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

- 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.

ProNoC homepage

#### Software Development

- 1. Click on Software button Software 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 content. This file contain all generated SoC functions and WB addresses.



Figure 5.14: The software edit window snapshot.

3. Now click on main.c file. Replace the content of the main.c 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 assert 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.

```
#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 ( 5000 );
   int_init();
   //assume hw interrupt pin is connected to cpu intrrupt pin \ensuremath{\mathsf{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 is 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 run successfully you must have ram0.bin, ram0.hex, and ram0.mif files in your sw/RAM directory.

/home/alireza/myw	Q	
Makefile	1 #include "tutorial.b"	0 0
README	2 // a simple delay function	
define printf b	3 void delay ( unsigned int num ){	
uenne_prind.n	4 while (num>0){	
link.ld	5 num; 6 non(); // asm volatile ("non");	
main.c	7 }	
▶ mor1kx	8 return;	
program.sh	9 }	
▶ simple-printf	10 11 char i=1:	
tutorial b	12 void timer isr(void){	
cuconann	13 //write your interrupt code here	
write_memory.sn	14 i*=2;	
	15 if((i&0xFF)==0) i=1; 16 lod WDITE(i);	
	17 timer TCSR=timer TCSR: //ack int	
	18 return;	
	19 }	
	20 21 July main () (	
	21 Int main(){ 22 printf("bello world()p");	
	23 delay(500);	
	24 int_init ( 5000 );	
	25 int_init();	
	26 //assume hw interrupt pin is connected to 10th 27 int add(0, times is: 0);	n cpu intrrupt pin
	27 Int_add(0, timer_isr, 0); 28 // Enable this interrupt	
	29 int_enable(0);	
	<pre>30 cpu_enable_user_interrupts();</pre>	
	<pre>31 timer_int_init(500);</pre>	
	32 while(1){	
from main	c:1:	
or1ky/system h:9:13	: note: expected void (*)(void *) but argument is of	type void (*)(void)
extern int int add/ur	signed long vert void (* handler)(void *) void *arg)	
A v		
ompliation finished si	Iccessfully.	

Figure 5.15: Compile the software code.

Simulate the generated RTL code using Modelsim software If you have installed Modelsim software you can simulate your SoC when running your developed software. To do this follow these instructions:

1. Press Compile RTL button open "select compiler window" as shown in Figure 5.16.



Figure 5.16: Compile the software code.

- 2. Select Modelsim as compiler tool.
- 3. Enter the path to your installed Modelsim bin directory.
- 4. Press Next.

- 5. Now you must have the testbech.v opened in software code edit window as shown in Figure 5.17. 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.
- 6. Press the run button to run the simulation in Modelsim software.



Figure 5.17: Select Modelsim compiler.

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

😣 🖱 💿 ModelSim SE-64 6.5c	
<u>File Edit View Compile Simulate A</u>	dd Wave Tools Layout Window Help
🗋 • 😅 🖬 % 🚭   🐒 🖻 🛍	요 그 : 🗛 문 锡   Help 💦 🔗 - 🥵 🤢 🥥 🚳 💭 🕱   Layout Simulate 🛛 🚽 볼 늘 관 관 권 군 도 교 🤇 🤧 : ( 錄
🖉 🛧 ሩ 🛸   📑 🛛 100 pu 🚽	L L L M M M M M M M M M M M M M M M M M
🛺 sim + 🗗 🗙 😩 + 🗗	Wave:
* Instance	Messages
<ul> <li>○ c</li> <li>○ c</li></ul>	■         ■
	とまる Now 1152690 ms 160000 ms 200000 ms
	Sec. 9 Cursor 1 1152689.608 ns
🧸 Library 👪 sim < 🕨 🔽 🖂	
Transcrint # heilo world! # Break key hit # Break in Statement instruction # Simulation Breakpoint: Break i	****     ***     ***     ***     ***     ***     ***     ***     ***     ***     ***     ***     ***     ***     ***     ***     ***     ***     ***
Now: 1,152,690 ns Delta: 4	sim /testbench/uu/topu/mort kx0/mort kx_cpu/cappuccino/mort kx_cpu/mort kx_ctrl_cappuccino/instructioncache_ctrl/#ASSIGN#1345

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 Compile RTL button for a shown in Figure 5.19. Select Verilator as compiler tool then press Next.

😣 🖱 🔍 Step 1: Select Compiler	
	QuartusII
Compiler tool	Verilator
	Modelsim
	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.

🕽 🖨 💿 Step 2: Compile
<pre>www.ming.combolit.moritx_cut_cappuccho.v.1004. belayed assignments (&lt;-) in non-clocked (non-nop or later) block, suggest blocking assignments (=).</pre>
%Warning-COMBDLY: mor1kx_ctrl_cappuccino.v:1067: Delayed assignments (<=) in non-clocked (non flop or latch) block; suggest blocking assignments (=).
%Warning-COMBDLY: mor1kx_ctrl_cappuccino.v:1070: Delayed assignments (<=) in non-clocked (non flop or latch) block; suggest blocking assignments (=).
%Waming-UNOPTFLAT: mor1kx_icache.v:162: Signal unoptimizable: Feedback to dock or drcular logic: v.cpu.mor1kx0.mor1kx_cpu.cappuccino.mor1kx_cpu->mor1kx_fetch_cappuccino.icache_gen.mor1kx_icache.next_lru_history
%Warning-UNOPTFLAT: Example path: mor1kx_icache.v:162; v.cpu.mor1kx0.mor1kx_cpu.cappuccino.mor1kx_cpu- >mor1kx_fetch_cappuccino.icache_gen.mor1kx_icache.next_Iru_history
%Warning-UNOPTFLAT: Example path: mor1kx_icache.v:324: ALWAYS
%Warning-UNOPTFLAT: Example path: mor1kx_icache.v:157: v.cpu.mor1kx0.mor1kx_cpu.cappuccino.mor1kx_cpu- >mor1kx_fetch_cappuccino.icache_gen.mor1kx_icache.access
%Warning-UNOPTFLAT: Example path: mor1kx_cache_iru.v:173: ALWAYS
%Warning-UNOPTFLAT: Example path: mor1kx_icache.v:162: v.cpu.mor1kx0.mor1kx_cpu.cappuccino.mor1kx_cpu- >mor1kx_fetch_cappuccino.icache_gen.mor1kx_icache.next_iru_history
%Waming-UNOPTFLAT: mor1kx_dcache.v:181: Signal unoptimizable: Feedback to clock or circular logic: v.cpu.mor1kx0.mor1kx_cpu.cappuccino.mor1kx_cpu->mor1kx_lsu_cappuccino.dcache_gen.mor1kx_dcache.next_lru_history
%Warning-UNOPTFLAT: Example path: mor1kx_dcache.v:181: v.cpu.mor1kx0.mor1kx_cpu.cappuccino.mor1kx_cpu- >mor1kx_lsu_cappuccino.dcache_gen.mor1kx_dcache.next_Iru_history
%Warning-UNOPTFLAT: Example path: mor1kx_dcache.v:461: ALWAYS
%Warning-UNOPTFLAT: Example path: mor1kx_dcache.v:176: v.cpu.mor1kx0.mor1kx_cpu.cappuccino.mor1kx_cpu- >mor1kx_su_cappuccino.dcache_gen.mor1kx_dcache.access
%Warning-UNOPTFLAT: Example path: mor1kx_cache_Iru.v:173: ALWAYS
%Warning-UNOPTFLAT: Example path: mor1kx_dcache.v:181: v.cpu.mor1kx0.mor1kx_cpu.cappuccino.mor1kx_cpu- >mor1kx_lsu_cappuccino.dcache_gen.mor1kx_dcache.next_nu_history
%Error: Exiting due to 126 warning(s)
%Error: Command Failed /usr/bin/verilator_bincc tutorial.vprofile-dfuncsprefix Vtop -03 -CFLAGS -03
Veriator model has been generated successfully!
Previous 🕑 Next

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 testbench code:





6. Press Compile button to generate the executable binary file. If the file is generated successfully you must see the "compilation finished successfully" message as shown in Figure 5.22.

*RETCODE == 0		
1		
ar: creating Vtop ALL.a		
Compilation finished successfully.	6	7
🖕 Previous 💦 Regenerate Testbench.cpp	Compile	Run

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 botton 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<br/>generated RTL<br/>code usingIf you have installed Quartus II software on your system and an Altera FPGA devel-<br/>opment board, you can implement your SoC on your target FPGA and also change its<br/>software code at runtime using following instructions:Quartus II<br/>software1. Press Compile RTL buttonImage: Compile RTL button

1. Press Compile RTL button Compile RTL 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 Verilator compiler.

2. Enter the path to your installed QuartusII  ${\tt bin}$  directory.

- 3. In Targeted Board search for your FPGA board name. If the board exist select it, press Next button and continue from step 5. Otherwise, select Add New Board and then press 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 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 Auto Fill button to auto-fill the JTAG configuration setting.
  - (e) Press Add button.

	0	DE10_Nano_VB2	а
FPGA board golden top QSF file:	0	DE10_Nano_golden_top.qsf 🔎	b
FPGA board golden top verilog file	0	DE10_nano_VB2.v 🔎	с
FPGA Board JT/	AG Conf	iguration	
FPGA Borad USB Blaster PID:	0	6010	
FPGA Borad Programming Hardware Name	•	DE-SoC	
FPGA Borad Device location in JTAG chain:	0	2	
etected PID: 6010 home/alireza/intelFPGA_lite/17.1/quartus	/bin/jta	gconfig	
etected PID: 6010 home/alireza/intelFPGA_lite/17.1/quartus ) DE-SoC [1-2] 4BA00477 SOCVHPS 02D020DD SCSEBA6(.JES)/SCSEMA6/ *RETCODE == 0	/bin/jta	gconfig	

Figure 5.25: Add new FPGA board to ProNoC.

5. Assign your SoC pins to your FPGA boards defined pins as shown in Figure 5.26.

Port Direction	Port Range	Port name	Assigment Type	Board Port name	Board Port Range
nput		cpu_cpu_en	Direct ‡	*VCC ‡	
output	[4-1:0]	led_port_o		LED ‡	3 ‡:0 ‡
nput		source_clk_in	Direct *	FPGA_CLK1_50 ‡	
nput		source_reset_in	Negate(~) ‡	KEY ‡	0 ‡

Figure 5.26: SoC pin assignment.

Here, we have connected the enable pin to logic 1, led\_port [3:0] to LED[3:0], the clk signal to FPGA\_CLK1\_50 and reset to negate KEY[0]. KEY[1:0] are push-button switches and are active high.

6. Press Next button.

ſ

7. Press Compile button. Then wait until QuartusII compilation tasks to be finished.

/home/alireza/mywork/mpsoc_work	Q	RegExp Car
Top.v	10 anosis neverecerves a copy or one	ono ceser ocneror ruone
▶ lib	20 ** License along with ProNoC. If not, see	< <u>nttp://www.gnu.org/licenses/</u> >.
testhensh u	21	,
teschench.v	23 module Top (	
tutorial.v	24 FPGA_CLK1_50,	😠 🖨 🛞 sh
tutorial_top.v	25 KEY,	To find a second design of the first second time and second to set the
	26 LED	Infot to the terms and conditions of the Intel Program License
	28 input EPGA CLK1 50	Info: Subscription Agreement, the Intel Quartus Prime License Agreement,
	29 input [1:0] KEY:	Info: the Intel FPGA IP License Agreement, or other applicable license
	30 output [7:0] LED;	Info: agreement, including, without limitation, that your use is for
	31	Info: Intel and sold by Intel on its authorized distributors. Please
	32	Info; refer to the applicable agreement for further details.
	33 tutorial_top uut(	Info: Processing started: Non Apr 9 14:49:28 2018
	34 .cpu_cpu_en(1 b1), 35 led port o( LED [3 : 01)	Info: Command: quartus_map ==54bit tutorial ==read_settings_Files=on
	36 source dk in( EPGA CLK1 50 ).	version 16.0.2. Dreated Quartus Prime Befault Settinos File /howe/aliceza/wu
	37 .source_reset_in(~ KEY [ 0])	/wpsoc_work/SOC/tutorial/tutorial_assignment_defaults.qdf, which contains th
	38 );	fault assignment setting information from Quartus II software version 16.0.2
	39	Info (125063): Default assignment values were changed in the current version
	40	d in file /home/aliceza/intelEPSe lite/17.1/quantus/linux64/assignment defau
	41 endmodule	qdf
	16	Warning (18236): Number of processors has not been specified which way cause
erm -e sh -c '/home/alireza/intelFPC	34 lite/17 1/quartus/bip/quartus fit64bit tute	ricading on shared machines. Set the global assignment NUT_PHMALLEL_PKULESS
Theme falings a low such low page was	te / COC/haterial / "	Info (20030): Parallel compilation is enabled and will use 2 of the 2 proces
realized and a line an	The local distances of the local data and t	detected
enni -e sii -c ynomeyaniezayinten -c	34_inte/17.1/quartus/bin/quartus_asino4bic tu	
/nome/airreza/mywork/mpsoc_wor	k/SOC/tutonal/	
erm -e sh -c '/home/alireza/intelFPC	3A_lite/17.1/quartus/bin/quartus_sta64bit tut	prial;echo \$? > status'
artus compilation is done successful	ly in /home/alireza/mywork/mpsoc_work/SOC/tu	torial!

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 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 software button and change the timer interrupt time from 500 to 5000000. Then press compile button. By clicking on group button you can reprogram your SoC memory using the new code and you should observe the LEDs blinking now.

😣 🖱 💿 /home/alireza/mywork/m	psoc_work/SOC/tutorial/sw/main.c - Otec
/home/alireza/mywork/mpsoc_work	Q RegExp Case
Makefile	15 ff((8007)=0) (=1;
▶ RAM	10 itel_WRITC(I); 17 itimer TCSR=itimer TCSR: //ark int
README	18 return;
define_printf.h	19 }
image	21 int main(){
image.ihex	22 printf("hello world!\n");
image.lst	23 delay(500);
image.map	25 int_ini();
itag intfc.sh	26 //assume hw interrupt pin is connected to cpu intrrupt pin 0
link.ld	2/ int_ada(0, time_isr, 0); 28 // Enable this internut
main.c	29 int_enable(0);
▶ mor1kx	30 cpu_enable_user_interrupts();
program.sh	32 while(1){
simple-printf	33 delay(500);
tutorial b	34 ) 35 return 0:
write memory sh	36 }
write is verified	
index num=127	
Initial Vitag for DE-SoC & @2*	
send I:1,D:2:0,I:0 to itag	
*RETCODE == 0	
Memory is programed successfully!	
Regenerate main.c	Compile 🗘 Program the memory

Figure 5.29: Increase timer interrupt time.

**CHAPTER 6** 

# Add Custom IP to Processing Tile Generator Tutorial

Summary	This tutorial teaches how to add a custom intellectual property (IP) core to <b>ProNoC</b> <b>Processing Tile Generator</b> using <b>IP Generator</b> . 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 In- stallation 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.
<b>Objectives:</b>	
	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 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: <b>Data</b> : $(p, q)$ : A pair of 8-bit binary positive numbers. <b>Result</b> : $gcd$ : greatest common divisor INITIALIZE; while $p \neq q$ do <b>if</b> $p > q$ <b>then</b>   p = p - q; <b>end</b> <b>else if</b> $p < q$ <b>then</b>   q = q - p; <b>end</b> <b>else</b>   gcd = p; <b>end</b> <b>end</b>

Algorithm 1: Greatest Common Divisor algorithm.

The GCD flow chart:



Figure 6.1: GCD flow chart.

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

```
Listing 6.1: gcd.v
/ * * * *
* GCD
* * *
                    ******/
module gcd #(
parameter GCDw=32
)( clk, reset, enable, in1, in2, done, gcd);
   input clk, reset;
   input [GCDw-1 : 0] in1, in2;
   output [GCDw-1 : 0] gcd;
   input enable;
   output done;
   wire ldG, ldP, ldQ, selP0, selQ0, selP, selQ;
   wire AeqB, AltB;
   gcd_cu CU(
      .clk (clk),
      .reset (reset),
      .AeqB (AeqB),
      .AltB (AltB),
      .enable (enable),
      .ldG (ldG),
      .ldP (ldP),
      .ldQ (ldQ),
      .selP0 (selP0),
```

```
.selQ0 (selQ0),
      .selP (selP),
      .selQ (selQ),
      .done (done)
   );
   gcd_dpu #(
      .GCDw(GCDw)
   )DPU(
      .clk (clk),
      .reset (reset),
      .in1 (in1),
      .in2 (in2),
      .gcd (gcd),
      .AeqB (AeqB),
      .AltB (AltB),
      .ldG (ldG),
      .ldP (ldP),
      .ldQ (ldQ),
      .selP0 (selP0),
      .selQ0 (selQ0),
      .selP (selP),
      .selQ (selQ)
      );
endmodule
/ * * * * * * * *
              * * * * * * * * * * *
* gcd_cu
            ************
*****
module gcd_cu (clk, reset, ldG, ldP, ldQ, selP0, selQ0, selP, selQ, AeqB,
    AltB, done, enable);
   input clk, reset;
   input AeqB, AltB, enable;
output ldG, ldP, ldQ, selP0, selQ0, selP, selQ, done;
   reg ldG, ldP, ldQ, selP0, selQ0, selP, selQ, done;
   //State encoding
   parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10;
   reg [1:0] y;
   always @ (posedge reset or posedge clk) begin
      if (reset == 1) y <= S0;</pre>
      else begin
         case (y)
            S0: begin if (enable == 1) y <= S1;
             else y <= SO;
            end
            S1: begin if (AeqB == 1) y <= S2;
               else y <= S1;</pre>
```

```
end
            S2: begin if (enable == 0) y <= S0;
            else y <= S2;
           end
           default: y <= S0;</pre>
        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
          1dG = 1;
           done = 1;
        end
        else if (AltB == 1) begin
          ldQ = 1;
        end
        else begin
          1dP = 1; selP = 1; selQ = 1;
        end
     end
     S2: begin
        1dG = 1;
        done = 1;
     end
     default: ;
     endcase
     end
   endmodule
* gcd_dpu
module gcd_dpu #(
 parameter GCDw=32
)( clk, reset, in1, in2, gcd, ldG, ldP, ldQ, selP0, selQ0, selP, selQ,
```

```
AeqB, AltB);
   input clk, reset;
   input [GCDw-1:0] in1, in2;
   output [GCDw-1:0] gcd;
   input ldG, ldP, ldQ, selP0, selQ0, selP, selQ;
  output AeqB, AltB;
   reg [GCDw-1:0] reg_P, reg_Q;
  wire [GCDw-1:0] wire_ALU;
  reg [GCDw-1:0] gcd;
  wire AeqB, AltB;
   //RegP with Multiplex 2:1
   always @ (posedge clk or posedge reset)begin
      if (reset == 1) reg_P <= 0;</pre>
      else begin
         if (ldP == 1)begin
            if (selP0==1) reg_P <= in1;</pre>
            else reg_P <= wire_ALU;</pre>
         end
      end
  end
      //RegQ with Multiplex 2:1
   always @ (posedge clk or posedge reset) begin
      if (reset == 1) reg_Q <= 0;</pre>
      else begin
         if (ldQ == 1)begin
            if (selQ0==1) reg_Q <= in2;</pre>
            else reg_Q <= wire_ALU;</pre>
         end
      end
   end
   //RegG with enable signal
  always @ (posedge clk or posedge reset)begin
      if (reset == 1) gcd <= {GCDw{1'b0}};</pre>
      else begin
        if (ldG == 1) gcd <= reg_P;</pre>
      end
   end
   //Comparator
  assign AeqB = (reg_P == reg_Q)? 1'b1 : 1'b0;
  assign AltB = (reg_P < reg_Q) ? 1'b1 : 1'b0;</pre>
   //Subtractor
   assign wire_ALU = ((selP == 1) & (selQ == 1)) ? (reg_P - reg_Q) : (
       reg_Q - reg_P);
endmodule
```

 $Create \verb"mpsoc/src_peripheral/other" directory and then copy the above \verb"gcd.v" file inside it.$ 

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

ProNoC homepage

command in terminal

sudo apt-get install verilator

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

verilator --cc gcd.v

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

3. Open obj\_dir folder and create testbench.cpp inside it:

```
Listing 6.2: testbench.cpp
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <verilated.h>
#include "Vgcd.h" // From Verilating "gcd.v"
unsigned int input1[10] ={136, 25, 33220, 3627, 3450, 9375, 199317,
   157620, 5694235, 199307 };
unsigned int input2[10] ={248, 50, 2200, 4581, 6540, 61575, 103443,
    238844, 239871, 903443};
unsigned int expt_gcd[10] ={8, 25, 220, 9, 30, 75, 2523, 284, 2161,
    1};
Vgcd *gcd // Instantiation of module
unsigned int main_time = 0; // Current simulation time
int run;
unsigned int i=0,passed=1;
int main(int argc, char** argv) {
   Verilated::commandArgs(argc, argv); // Remember args
   gcd = new Vgcd;
   /**
   * initialize input
   ******
   gcd->reset=1;
   gcd->enable=0;
     gcd->in1=0;
     gcd->in2=0;
   main_time=0;
   run=0;
   while (!Verilated::gotFinish() && i<10) {</pre>
      if (main_time & 0x1) {
         gcd \rightarrow 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 ***************");
   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 -03 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.

ProNoC homepage

#### 6. To run the simulation run:

./testbench

#### Expected output:

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 vari-	Write-only
		able. Writing on this register will trigger	
		the GCD's enable port	
3	GCD	Holds the generated GCD value	Read-only

 $Create \ the \ following \ file \ inside \ {\tt mpsoc/src\_peripheral/other} \ directory$ 

```
Listing 6.4: gcd_ip.v
module gcd_ip#(
    parameter GCDw=32,
    parameter Dw =GCDw,
    parameter Aw =5,
    parameter TAGw =3,
    parameter SELw =4
)
(
    clk,
    reset,
    //wishbone bus interface
    s_dat_i,
    s_sel_i,
    s_addr_i,
```

```
s_tag_i,
   s_stb_i,
  s_cyc_i,
  s_we_i,
  s_dat_o,
   s_ack_o,
   s_err_o,
   s_rty_o
);
   input clk;
   input reset;
   //wishbone bus interface
   input [Dw-1 : 0] s_dat_i;
   input [SELw-1 : 0] s_sel_i;
   input [Aw-1 : 0] s_addr_i;
   input [TAGw-1 : 0] s_tag_i;
   input s_stb_i;
   input s_cyc_i;
   input s_we_i;
   output [Dw-1 : 0] s_dat_o;
   output reg s_ack_o;
   output s_err_o;
   output s_rty_o;
//Wishbone bus registers address
   localparam DONE_REG_ADDR=0;
   localparam IN_1_REG_ADDR=1;
   localparam IN_2_REG_ADDR=2;
   localparam GCD_REG_ADDR=3;
   assign s_err_o = 1'b0;
   assign s_rty_o = 1'b0;
   wire[GCDw-1 :0] gcd;
   reg [GCDw-1 :0] readdata,in1,in2;
   wire done;
   assign s_dat_o =readdata;
   always @ (posedge clk or posedge reset) begin
     if(reset) begin
        s_ack_o <= 1'b0;
      end else begin
        s_ack_o <= (s_stb_i & ~s_ack_o);
      end //reset
   end//always
   always @ (posedge clk or posedge reset) begin
      if(reset) begin
        readdata <= 0;
        in1 <= 0;
        in2 <= 0;
      end else begin
```

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

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.

#### ProNoC 📑 🖪 💉 Interface generator Processing tile generator Interfaces list Select file: Browse IP name: ▶ interrupt Select Select Select Bus h Add Software ddd HDL ▶ source ▶ wishbone Interface name Туре Interface Num Type Port name Interface name Interface port Port Range ease select the verilog file containig the ip module 🔎 Load IP 🙆 Generate

## 2. Then select the regenerator and 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.

🗲 Interface gen	erator 🏼 🖗 IP generator	Processing tile genera	tor Soc based MPSoC generation	ator	
nterfaces list	Select file: //hom	e/alireza/mywork/mpsoc/src_l	peripheral/Other/gcd_ip.v 3	P Browse	IP name: gcd 5
interrupt source	Select module: gcd_ip	Parameter Sel setting Cate	ect Other 4	• 🛛 🕖	IP Description h Add Software ddd HD
wishbone		ace name	Туре		Interface Num
	Interr				
	Type	Port name	Interface name	Interface port	Port Range
	Type	Port name cik	Interface name	Interface port	Port Range
	Type input input	Port name dk reset	Interface name	Interface port IO ‡ IO ‡	Port Range
	Type input input input	Port name dk reset s_dat_i	Interface name 10 ¢ 10 ¢ 10 ¢	Interface port IO \$ IO \$ IO \$	Port Range
	Type input input input input	Port name cik reset s_dat_i s_sel_i	Interface name 10 ¢ 10 ¢ 10 ¢ 10 ¢	Interface port IO 10 10 10 10 10 10 10 10	Port Range

Figure 6.4: Select gcd.v file.

- 6. The gcd\_ip.v file has one parameter named as GCDw which we want to be redefined by the end user during IP call time. To define the appropriate GUI interface for this parameter click on parameter setting.
- 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 select  ${\tt Localparam}.$
- 10. Click on *(i)* 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 add your setting.

			GCD's In	Add description	11	
Interface generator interfaces list NoC interrupt source Sel Sel Sel Sel Sel Sel Sel Se	P generator ct file: /home/alire elect odule: gcd_ip	Processing tile generator za/mywork/mpsoc/src_periph Parameter setting 6 Select Category:	eral/Other/gcd_i		Ø OK	
Define paramet Parameter name	ers detail Default va	ue Widget type	Widget content	Туре	info 🧲	add/remove
GCDw	32	Combo-box	\$ 8,16,32	Localparam 🌲	👿 Redefine 10 🚺	remove
Dw	GCDw	Fixed	: 8	Localparam ‡	🗹 Redefine 🕡	remove
Aw	5	Fixed	•	Localparam 🛟	👿 Redefine 🕡	remove
TAGw	3	Fixed	•	Localparam ‡	🗑 Redefine 🛛 🕡	remove
SELw	4	Fixed	•	Localparam 🛟	👿 Redefine 🕡	remove
		Fixed	•	Don <sup>9</sup> include 🛟	👿 Redefine 🕡	🔂 add
			💽 ок			

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 for wishbone address range.
- 16. Set block address range as 5. This results in allocating 32 Bytes for each instances 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 register which is equal to 16 Bytes).
- 17. Press ok.

Now we need to map each module individual port to its appropriate interface port. By selecting the interface name, the application automatically selects the nearest port which match with module port name. For this example it 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 port.
- 19. Select plug:reset for reset port.
- 20. Connect all other ports to plug:wb\_slave. port.

Interface generator	🛛 🙁 🗇 🕕 Interface parai	meter setting						
nterfaces list		15b						
NoC	interface name		address ra	ange: (start end name)			block address widt	h
interrupt		-			_			
source	wb_slave	0xb800_0000	0xbfff_	ffff custom device	s 15 ÷	Fixe	d 2 8	Bytes 5
enable								16
reset 13				17 🔗 ок				
vishbone								
wb_addr_map	Interface name	Туре		Interface Nu	m	_		
wb_master	reset	plug ±		1:0	0		Remove	
wb_slave								
	clk	plug 💲		1 2	0		🐼 Remove	
	wb_slave	piug 🛫			<b>1</b> 4		Remove	V
	Type	Port name		Interface name	Interface r	ort	Port Ba	nge
	input	clk	10	plug:clk +	clk I		Foreita	iige
	mput	CIR	10	plug.cik +	CIK_I			
	input	reset	19	plug:reset ‡	reset_i	-		
	input	s_dat_i		plug:wb_slave 💲	dat_i	:	Dw-1 : 0	
	input	s_sel_i		plug:wb_slave :	sel_i	: 5	SELw-1 : 0	
	logut	a adde i		pluguub clavo	ade I			
	mpuc	s_auur_r		plug.wb_slave +	aui_i		W-1 . 0	
	input	s_tag_i		plug:wb_slave 💲	tag_i	: 1	'AGw-1 : 0	
	input	s_stb_i		plug:wb_slave 💲	stb_i	:		
	Input	s cvc l	20	plug.wb_slave_*	CVC I			
	inpac	0_0/0		plaging_slave +	c/c_1			
	input	s_we_i		plug:wb_slave 💲	we_i	-		
		100 March 100						

Figure 6.6: GCD Core interface setting.

- 21. Click on Add HDL Files button.
- 22. In front of Select file(s) click on Browse.
- 23. Select gcd.v and gcd\_ip.v files and press ok.

Select file: //ho	ome/alireza/Mywork/mpsoc_doc/usermanual/tutorial2/gcd_p.v Prowse IP name: gcd 21
Select gcd_lp ‡	⊘ Parameter     Select Setting     GCD     ▼      IP     Add Software files     Image: Add HDL files
1	See add HDL file(s)
Interface nar clk	Global variables
reset	23     Selecet file(s):     Prowse       22     Selecet folder(s):     Prowse
wb_slave	File path //mpsoc/src_peripheral/GCD/gcd.v
Туре	/mpsoc/src_peripheral/GCD/gcd_lp.v
Input	

Figure 6.7: Adding GCD core HDL files.

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

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

25. Click on Add to tile.h tab.

26. Copy following text on the new tab, then click on Save button.

```
#define ${IP}_DONE_ADDR (*((volatile unsigned int *) ($BASE)))
#define ${IP}_IN_1_ADDR (*((volatile unsigned int *) ($BASE+4)))
#define ${IP}_IN_2_ADDR (*((volatile unsigned int *) ($BASE+8)))
#define ${IP}_GCD_ADDR (*((volatile unsigned int *) ($BASE+12)))
#define ${IP}_IN1_WRITE(value) ${IP}_IN_1_ADDR=value
#define ${IP}_IN2_WRITE(value) ${IP}_IN_2_ADDR=value
#define ${IP}_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  $p[variable_name]$  format here such as all IP core parameters and IP core Verilog instance name. These variables will be replaced with their exact values during SoC generation time). In this example, we used variable q[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 distinguished 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 tools facilitate 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 first then continue with the next step.
  - (c) Click on Setting button to open the IP core parameter setting window.
  - (d) Adjust IP core parameters according to Table 6.2.
  - (e) Rename the IP core instance name according to Table 6.2.
  - (f) Connect IP cores interfaces as listed in Table 6.2.
| Category      | IP name         | Paran  | neter  |   | Instance name | Inte                                 | rface   | connection   |
|---------------|-----------------|--|--|---|---------------|--------------------------------------|---|--|
| Source        | clk source      |  | _  |   | source        | clk                                  | $\rightarrow$   | IO   |
| Source        | CIK_SOULCE      |  | -  |   | source        | reset                                | $\rightarrow$   | IO   |
| Bus           | wishbone_bus    | M<br>S<br>Dw<br>Aw   | $\begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \end{array}$   | 3<br>5<br>32<br>32  | bus           | clk<br>reset                         | $\stackrel{\rightarrow}{\rightarrow}$   | source:clk<br>source:reset   |
| Processor     | aeMB            | STACK_SIZE<br>HEAP_SIZE  | $\rightarrow \rightarrow$  | 0X400<br>0x400  | aeMB          | clk<br>reset<br>iwb<br>dwb<br>enable | $\begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \end{array}$ | source:clk<br>source:reset<br>bus:wb_master[0]<br>bus:wb_master[1]<br>IO |
| RAM           | single_port_ram | Dw<br>Aw<br>BYTE_WR_EN<br>FPGA_VENDOR<br>JTAG_CONNECT<br>JTAG_INDEX<br>BURST_MODE<br>MEM_CONTENT_<br>FILE_NAME<br>INITIAL_EN | $\begin{array}{c} \rightarrow \\ \rightarrow \end{array}$ | 32<br>12<br>"YES"<br>"ALTERA"<br>"DISABLED"<br>CORE_ID<br>"DISABLED"<br>"ram0"<br>"YES" | ram           | clk<br>reset<br>wb                   | $\begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \end{array}$  | source:clk<br>source:reset<br>bus:wb_slave[0]                            |
| Interrupt     | int_ctrl        | INT_NUM  | $\rightarrow$  | 1   | int_ctrl      | clk<br>reset<br>interrupt_cpu<br>wb  | $\begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \end{array}$ | source:clk<br>source:reset<br>aeMB:interrupt_cpu<br>bus:wb_slave[1]      |
| Timer         | timer           | PRESCALE_WIDTH   | $\rightarrow$  | 8   | timer         | clk<br>reset<br>wb<br>intrp          | $\begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \end{array}$                               | source:clk<br>source:reset<br>bus:wb_slave[2]<br>in_ctrl:int_periph[0]   |
| Communication | jtag_wb         | Dw<br>VJTAG_INDEX  | $\rightarrow$ $\rightarrow$  | 32<br>CORE_ID   | jtag_wb       | clk<br>reset<br>wbm                  | $\rightarrow$<br>$\rightarrow$<br>$\rightarrow$   | source:clk<br>source:reset<br>bus:wb_master[2]                           |
| Communication | jtag_uart       | FPGA_VENDOR<br>SIM_BUFFER_SIZE<br>SIM_WAIT_COUNT   | ightarrow<br>ightarrow<br>ightarrow  | "ALTERA"<br>100<br>1000   | uart          | clk<br>reset<br>intrpt<br>wb         | $\rightarrow$<br>$\rightarrow$<br>$\rightarrow$<br>$\rightarrow$  | source:clk<br>source:reset<br>NC<br>bus:wb_slave[3]                      |

## Table 6.2: GCD SoC IP core list and setting.

2. Add the new GCD IP to SoC.

Table 6.3:	GCD SoC IP	core list	and setting.

Category	IP name	Para	meter	r	Instance name	In	terfac	ce connection
						clk	$\rightarrow$	source:clk
Other	gcd	GCDw	$\rightarrow$	32	gcd	reset	$\rightarrow$	source:reset
						wb	$\rightarrow$	bus:wb_slave[4]

© © ProNoC						
💉 Interface generator	IP generator	Processing tile generator	NoC based MPSoC generator			
IP list		umer 🐸	instance name	CIK		source:cik *
<ul> <li>Bus</li> <li>Communication</li> </ul>		Setting	timer	intrp		int_ctrl:int_periph[0]
<ul> <li>DMA</li> <li>Display</li> </ul>		Remove	Parameter setting for gcd	-		source:reset ‡
GPIO     Interrupt     NoC     Other		jtag_uart	8 Parameter name 16 GCDw 32	Description	i	GCD's Input/output width in bits
gcd sim_uart ▶ Processor ▶ RAM		Setting	🔗 ок	pe	riphe	OK :
<ul> <li>Source</li> <li>Timer</li> </ul>		gcd	Instance name	clk		source:dk ‡
		Setting	gcd	reset		source: reset
		📚 🔞 Remove		wb_slave		bus:wb_slave[4]
gcd module						
P Load Tile	Tile name: gcd_	soc	() Wishbone-bus addr	Diagram	🙆 Generat	e RTL Software Compile RTL

Figure 6.11

- 3. Set the tile name as gcd\_soc.
- 4. Press generate button. This must generate a folder in mpsoc\_work/SOC/gcd\_soc.

list	timer	Instance name	CIK	source: cik
Bus Communication	Setting	timer	intrp	int_ctrl:int_periph[0]
DMA Display	Remove		reset	source: reset
GPIO	8			bus:wb_slave[2]
NoC	jt 👔 Su	ocessing Tile "gcd_soc" has be occessfully at /home/alireza/my	en created work/mpsoc_work/	source: clk
dummy_module		JC/gca_soc/.	heral	NC
gcd sim_uart	₩		ок	source: reset
Processor RAM			wb_slave	bus:wb_slave[3]
Source Timer	gcd	Instance name	clk	source: clk

Figure 6.12

## Software Development

1. Click on Software button **Software** to open the software development

window. Now click on main.c file. Replace the content 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.

```
#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");
   }
</pre>
```

```
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.

Port Direction	Port Range	Port name	Assigment Type	Board Port name	Board Port Range
nput		aeMB_sys_ena_i	Direct	* VCC *	
nput		source_clk_in	Direct	FPGA_CLK1_50	
nput		source_reset_in	Negate(~)	¢ Key ¢	0 ‡

Figure 6.13: DE10-Nano FPGA board pin assignment.

## 😑 💿 alireza@alireza: ~/mywork/mpsoc/perl\_gui/lib/perl alireza@alireza:~/mywork/mpsoc/perl\_gui/lib/perl\$ \$QUARTUS\_BIN/nios2-terminal nios2-terminal: connected to hardware target using JTAG UART on cable nios2-terminal: "DE-SoC [1-2]", device 2, instance 0 nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate) GCD test application Enter number #1: 25684 Enter number #2: 36 GCD\_hardware (25684,36) = 4 GCD\_software (25684,36) = 4 spead up=12.0 times clock\_num=842 clock\_num=10182 Enter number #1: 45585 Enter number #2: 75 GCD\_hardware (45585,75) = 15 clock\_num=722 GCD\_software (45585,75) = 15 spead up=12.0 times clock\_num=8672 Enter number #1: 311 Enter number #2: 222 GCD\_hardware (311,222) = 1 GCD\_software (311,222) = 1 spead up=6.1 times clock\_num=158 clock\_num=966

Figure 6.14: Nios2-terminal output snapshots.

CHAPTER 7

## NoC Verilog File Parameters Description

r		
V	$\forall \in \mathbb{N}, \forall \ge 1.$	Number of VC per router port. Defining V as 1 results in a simple non-VC based router.
В	$\mathbf{B} \in \mathbb{N}, \mathbf{B} \geqslant 1$	Buffer size per VC in flit .
NX	$NX \in \mathbb{N}, NX \geqslant 2$	The number of node in x axis of a mesh or torus topology. For ring and line topolo- gies, it is total number of nodes in the ring.
NY	$\texttt{NY} \in \mathbb{N}, \texttt{NY} \geqslant 2$	The number of node in y axis of mesh or torus. Not used in ring and line topolo- gies.
С	$c \in \mathbb{N}$	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 de- fined using CLASS_SETTING parameter.
Fpay	Fpay $\in \mathbb{N}$ , Fpay $\geqslant 32$	Flit payload size in bit.
MUX_TYPE	"BINARY", "ONE_HOT"	Crossbar's multiplexer type in a NoC router. Binary and one-hot multiplexers are preferable for FPGA and ASIC im- plementation, 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.
COMBINATION_ TYPE	"COMB_NONSPEC", "COMB_SPEC1", "COMB_SPEC2", "BASELINE"	VC/SW combination type. Note that us- ing "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 regis- ters are updated only if they also receive the second level arbitration grants.
TOPOLOGY	"MESH", "TORUS" "RING" "LINE"	The NoC topology.

ROUTE_NAME	"XY", "DUATO", "WEST_FIRST", "NORTH_LAST", "NEGETIVE_FIRST", "ODD_EVEN"	NoC routing algorithm for mesh topol- ogy. "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 topol- ogy. See ? for more information.
CONGESTION_ INDEX	congestion_index $\in \mathbb{N}$ , $0 \leqslant$ congestion_index $\leqslant 7$	Define how congestion metrics is se- lected. See Table 7.2 for more informa- tion.
DEBUG_EN	0,1	If is defined as 1, the simulation will be run using extra debugging codes. The debugger dose several faults detec- tion 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 im- plementation 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 with asserted bits rep- resent the VCs 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. Yoiu must make sure that each message class have access to at least one EVC to prevent deadlock.

SSALEN	"YES", "NO"	If set as "YES", packets which are travel- ing to the same dimension bypass router pipeline stages using Static straight allo- cator.
SWA_ARBITER_ TYPE	"RRA", "WRRA"	Switch allocator's output ports arbiters type: RRA: Round Robin Arbiter. Pro- vides only local fairness in a router. WRRA: Weighted Round Robin Arbiter. Results in global fairness in the NoC. Switch allocation requests are grated ac- cording to their weight which increases due to contention
WEIGHTw	WEIGHTW $\in \mathbb{N},$ $2 \leqslant$ WEIGHTW $\leqslant 7$	WRRA weights' maximum width in bits.

Table 7.2: Congestion metrics. The adaptive router can be configured bypassing the congestion index as parameter to the to level RTL code.

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 neighbor-	-
	ing router adjacent input port.	
2	Number of active switch allocation requests in all ports	2-bit
	of the neighboring router.	
3	Number of active switch allocation requests in all ports	3-bit
	of the neighboring router.	
4	Number of active switch allocation requests in all ports	2-bit
	of the neighboring router that are not granted.	
5	Number of active switch allocation requests in all ports	3-bit
	of the neighboring router that are not granted.	
6	Number of unavailable VC in all ports of the neighbor-	2-bit
	ing router	
7	Number of unavailable VC in all ports of the neighbor-	3-bit
	ing router	

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:	<ul> <li>You will need a computer system running Linux OS with:</li> <li>1. Installed the ProNoC GUI software and its dependency packages.</li> <li>2. Installed Verilator simulator.</li> <li>For more information about the ProNoC and GNU toolchain installation please refer to the ProNoC system installation file located in /DoC folder.</li> </ul>
Simulation Example:	In this example we simulate two $8 \times 8$ Mesh NoCs, one with fully adaptive routing and another with DoR routing algorithms.
Generate first NoC simulation model with XY routing	<ol> <li>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.</li> </ol>
	2. Click on 🕂 to open ProNoC simulator tabs.

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

2 💼 💼	3				
🔫 Trace general	tor NoC simulator	DoC emulator			
Run simulator Generate NoC Simulation Model	Router Type Topology	?         "VC_BASED"           ?         "MESH"	:		
4	Routers per row	2		Avg. throughput/latency Injected Packet Worst-Case Delay Executaion Time	
	VC number per port	2		875,9 - 750,8 -	Latency
	payload width	32		8625,0 2 588,0 2	•
	Routing Algorithm		•	g 375.0 259.0 125.0 0.0 Desired Rvg. Injected Load Per Router (filts/clock (%)) Pro_name	
۶	oad	Save as:		Run all	

Figure 8.1

- 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.

Trace genera	tor NoC simulator	NoC emulator	
Run simulator	payload widdi	<b>U</b> 32	verilatorcc traffic_gen_verilator.vprofile-cfuncsprefix "Vtraffic" -O3 -CFLAGS *RETCODE == 0
Generate NoC Configuration	Routing Algorithm	() XY"	
	SSA Ebable	() "NO"	Veriator model has been generated successfully!
	NoC Parameters		cd "/home/alireza/mywork/mpsoc_work/simulate/venlator/processed_rtl/obj_dir///" make lib
	Advance Parameters	5	make -f Vnoc.mk
	Pck. injector FIFO Width:	16	The circulation binant file has been successfully sensetted in (
	Save as:	@ mesh_8x8_xy 6	home/alireza/mywork/mpsoc_work/simulate!
	Project directory	() /home/alireza/mywork/m	рхо
		Generate 7	5 375.8
	1		259,8
	10-		125.0
	100-1-		0.0
			0 Desired Out - To lasted Load Des Destes (Clifts (

Figure 8.2: Generate NoC model

Generate the second NoC simulation model with fully adaptive routing

Run simulation under Matrix Transposed traffic pattern

- 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.
- 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		

	Search Path:	0	/home/alireza/m	iywork/mpso 🔎
Trace generator	Verilated Model:	0	mesh_8x8_xy	13 :
Run simulator 11 Generate NoC Name Add/Remov	Traffic Type	0	Synthetic	÷
Simulation Model 12	Configuration name:	0	ху	
	Traffic name	0	transposed 1	:
	Min pck size :	0	2	
	Max pck size :	0	10	
	Avg. Packet size:	0	6	:
	Total packet number limit:	0	200000	:
	Simulator clocks limit:	0	100000	;
201	Injection ratios:	0	2:20:2	Check

Figure 8.3

14. Click on  $\bigcirc$  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  $\bigotimes$  Run all button. You can also run each individual simulation by pressing the  $\bigotimes$  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.

ProNoC homepage

- 🔫 Trace NoC sin ۰ 0 Ο 3 0 0 0 0 C Avg. t 119. 0 182.4 (clock) -68.3 0 51.3 . 34. 17. ß 18 20 22 30 32 :k(%)) 🖶 full 🖶 xy 🕑 Run all 21 Save 🔎 Load Save as
- 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 per router for different injection ratios
- (d) send/received worst-case delay per router for different injection ratios
- (e) Simulation execution clock cycles



ratio.

Core ID

0.0

■xy ■full

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

Core ID

Contraction of



0.0

■xy ■full

(e) Simulation time in clock cycles.



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. These modules inject/sink packets to the prototype NoC according 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 $5 \times 5$ 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	<ol> <li>Open mpsoc/perl_gui in terminal and run ProNoC GUI application:         ./ProNoC.pl</li> </ol>
	It should open The GUI interface as illustrated in Figure 9.1.
	2. Click on <b>T</b> to open ProNoC simulator tabs.

3. Click on NoC  $\,\, {\tt Emulator} \,\, tab$  to open the emulator GUI interface:



Figure 9.1

 $4. \ Click \ on \ {\tt Generate NoC \ Emulation \ Model \ tab \ to \ open \ NoC \ configuration \ setting \ page.}$ 

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.

Pck. injector FIFO Width:	16
Save as:	@ mesh_5x5_xy 6
Project directory	(home/alireza/mywork/mpso)
	Generate 7

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.

Port Direction	Port Range	Port name	Assigment Type	Board Port name	Board Port Range
nput		clk	Direct ‡	FPGA_CLK1_50	÷
output		done_led		LED	÷ 0 ÷
output		jtag_reset_led		LED	÷ 1 ÷
output		noc_reset_led		LED	÷ 2 ÷
nput		reset	Negate(~) 📫	KEY	÷ 0 ÷

Figure 9.3: DE10-Nano FPGA board pin assignment.

Generate the second NoC emulation model with fully adaptive routing

Run Emulation models under Matrix Transposed traffic pattern

- 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.
- 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	ху	Traffic name	transposed 2
Packet size in flits	5	Packet number limit per node	1000000
Emulation clock limits	2500000	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	2500000	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.



(c) Injected packets per router at 50% injection (d) Worst-case delay per router at 50% injecratio.

0.0

■xy ■full



tion ratio.



(e) Emulation time in clock cycles.

Figure 9.4: Emulator sample results.