|  |
| --- |
| NetSpeed Gemini NoC Physical Design Guidelines  Version: GEMINI-15.10  November 5, 2015 |

NetSpeed Gemini NoC Physical Design Guidelines

About This Document

This document describes physical design guidelines for a NoC implementation. Using these guidelines the Physical Design and Integration teams can integrate and implement NoC design from a RTL which is auto generated by NocStudio to Synthesized netlist.

Audience

This document is intended for users of NocStudio:

* NoC Integration and Physical Desisn teams
* NoC Designers
* SoC Designers

Prerequisite

Before proceeding, you should generally understand:

* Basics of SoC Physical Design Concepts

Related Documents

The following documents can be used as a reference to this document.

* NetSpeed NocStudio Gemini User Manual
* NetSpeed Gemini IP Integration Spec

Customer Support

For technical support about this product, please contact [support@netspeedsystems.com](mailto:support@netspeedsystems.com)

For general information about NetSpeed products refer to: [www.netspeedsystems.com](http://www.netspeedsystems.com)

**Contents**

About This Document 2

Audience 2

Prerequisite 2

Related Documents 2

Customer Support 2

1 Introduction to NetSpeed NoC 6

1.1 Heterogeneous Mesh 7

1.2 Bridges 8

1.3 Microarchitecture for Rapid Timing Closure 9

2 From RTL generation to Synthesis 12

3 RTL Netlist Structure 13

3.1 Multiple NoC Router Layers 14

3.2 Naming convention 15

3.3 Example 15

4 Synthesis 17

4.1 Synthesis Methodology 18

4.2 Running Synthesis 19

4.2.1 For Cadence flow: 19

4.2.2 For Synopsys flow: 20

4.3 Analyzing outputs 20

4.3.1 Analyzing RC/RCP synthesis output 20

4.3.2 Analyzing DC/DCT synthesis output 20

5 Place and Route Guidelines 22

5.1 P&R keeping the NoC elements together 22

5.2 P&R merging NoC Elements with Host 22

5.3 NoC Quick Start Information 23

5.3.1 NoC clock domains 23

5.3.2 Clock Gating Cell 25

5.3.3 NoC Clock tree 26

5.3.4 Reset 26

5.3.5 Standard Cells 26

5.3.6 Channel Routing 27

5.3.7 Repeaters 27

5.3.8 Repeaters & Power Grid 27

5.3.9 Synchronizers 27

6 DFT 29

List of Figures

Figure 1: Example SoC with mesh NoC 6

Figure 2: Example Heterogeneous Mesh 7

Figure 3: A bridge sits between the host and the router 9

Figure 4: Communication is never combinatorial 9

Figure 5: Adding pipeline stages is functionally allowed 10

Figure 6: An example flow chart explaining steps from NocStudio to Synthesis output 12

Figure 7: NoC RTL Netlist Structure 13

Figure 8: 8 Port Router with 4 directional ports & 4 host ports 14

Figure 9: Dual NoC layers 15

Figure 10: NocStudio GUI snapshot 16

Figure 11: NoC Elements in stand-alone P&R regions 22

Figure 12: Merge all NoC Elements with Host P&R Region 23

Figure 13: Bridge and AXI agent with async clock 23

Figure 14: Host A and NoC clock domain different than Host B clock 24

Figure 15: Host A & B with different clock domains than NoC clock 24

Figure 16: Reset logic 26

Figure 17: NetSpeed synchronizer module 28

# Introduction to NetSpeed NoC

The NetSpeed NoC has been designed to meet complex interconnect architectural requirements while enabling ease of physical implementation and rapid timing closure. The NoC is based on a mesh topology, which allows simple point-to-point communication.



Figure 1: Example SoC with mesh NoC

The diagram above shows an example SoC with the SoC components interconnected with a mesh-based NoC. The various components in the system, called hosts, connect locally to a router in the northwest corner of the host. The routers are then connected to neighboring routers through point-to-point connections.

One of the significant implementation advantages of a mesh-based NoC is that routing the global signals is straightforward. Routers are placed in a grid formation, and the global connections follow the lines of the grid. This avoids large buses crossing paths with one another. It also avoids routing many different buses to a central location which can create significant congestion, such as in a crossbar design.

## Heterogeneous Mesh

The NetSpeed NoC topology is customizable to support heterogeneous host shapes and sizes and to meet architectural requirements including bandwidth, latency, area, and power consumption. One of the key ways this is supported is by creating a heterogeneous mesh.



Figure 2: Example Heterogeneous Mesh

In addition to topology, NetSpeed heterogeneous mesh supports significant variability. Link widths can vary at every interface. Links can be removed if they are not needed. Routers can be removed if they are not needed. The number of ports on each router can vary, including the port connecting the router to the host. The size and aspect ratio of the hosts can change as well.

For physical design, this means that individual components of the NoC can vary significantly. Topology may be heterogeneous and individual links may vary in width and length. NetSpeed heterogeneous mesh is however designed in NocStudio such that the resulting topology is based on the SoC floor plan and host shapes and sizes. This will avoid any significant surprises in terms of timing or wiring congestion during the physical implementation of the SoC components and the NoC.

While a heterogeneous mesh may differ from a standard full mesh topology significantly, it still retains some of the key implementation advantages of a mesh such as simplified signaling, distributed wiring and arbitration, and high performance. Connections remain point-to-point, and global routing still avoids congestion points.

## Bridges

Hosts communicate with each other through ports; a host may have multiple ports. Each port may connect to a NoC router’s port however the host ports do not directly communicate with the routers. Routers use a simple packetized communication protocol to exchange information quickly and efficiently, however the host ports are often designed to communicate using a standard bus protocol such as AMBA. In order for the host ports to communicate to the router, a bridge is used which converts the host port’s protocol to NoC’s internal protocol.

The bridge is a light-weight component that sits between the host port and the router it connects to and it is part of the NoC generated by NocStudio.

NoC

Router

X,Y

Bridge

Host A

Figure 3: A bridge sits between the host and the router

The NoC as a whole consists of the routers, the bridges, and the links between the routers and between the routers and bridges. Physical implementation must deal with all three of these components.

## Microarchitecture for Rapid Timing Closure

Since links in the NoC is always point-to-point between neighboring routers and between bridges and bridges, rapid timing closure can be relatively straightforward.

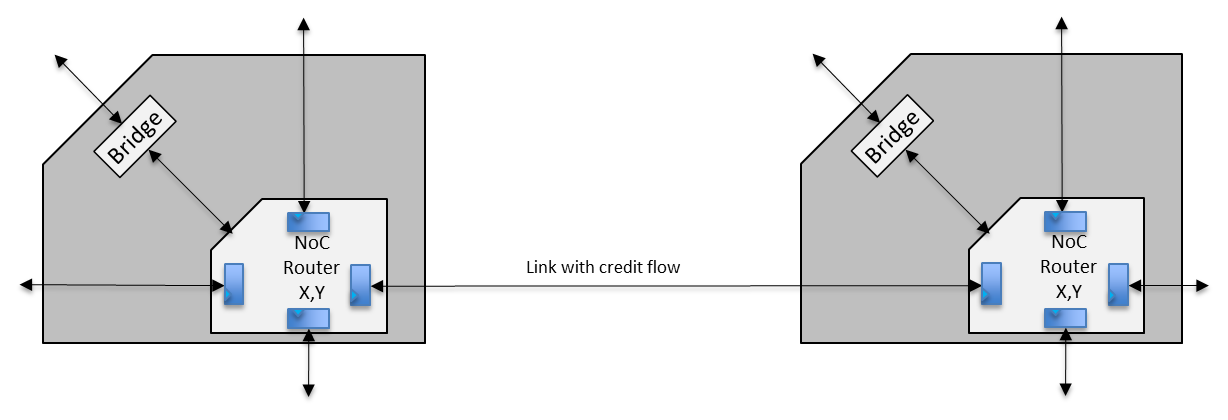


Figure 4: Communication is never combinatorial

Signals sent from one router to another are always registered at the receiving router upon arrival. There are no combinatorial paths connecting one link of the NoC with another. This ensures that timing closure only deals with paths starting in one router or bridge and ending in another router.

In addition, each link uses credit-based flow control with the associated flow control buffers at the receiving end. A link that is long with high latency may be pipelined by inserting register stages in the links adding to the latency; in such cases the flow control buffers are provisioned to account for the increased latency for better link utilization. Both credit exchanges and data exchange are registered at the corresponding receivers and at the inserted pipeline stages. Therefore there are no architectural critical paths at the global timing closure stage.

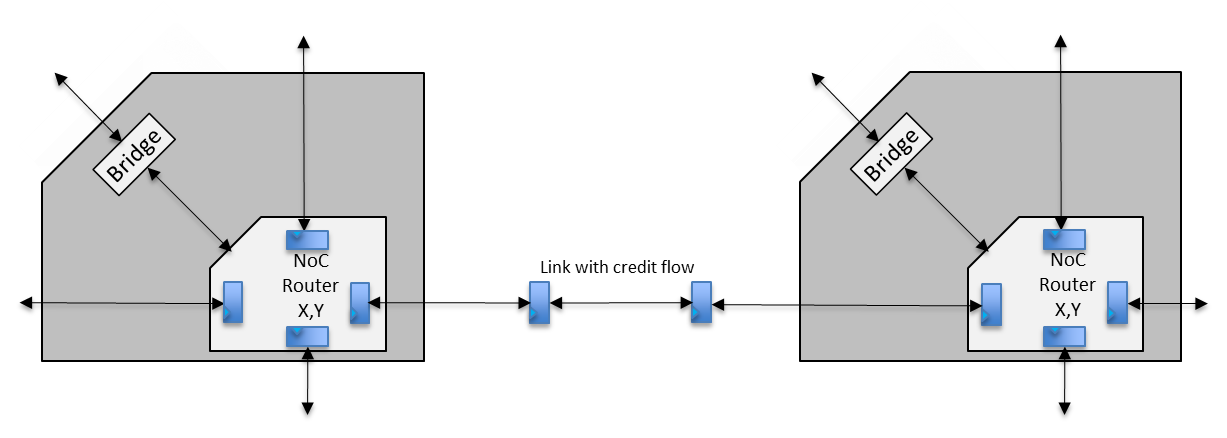


Figure 5: Adding pipeline stages is functionally allowed

If pipeline stages need to be added in order to make a timing path converge, this information should be provided back to the architect who designed the NoC in NocStudio. The flow control buffers depths are optimized for an expected round-trip latency by NocStudio. If pipeline stages are added, NocStudio automatically provisions additional buffers to handle increased latency.

This communication method also allows the frequency of the NoC to be limited by the internal router timing. The latency through the routing channel can be removed from the frequency limit by placing flop stages at the output boundary of the routers. Any number of stages can be added at the links to increase the frequency within the routing channel. Alternatively, the output register at the routers may be omitted, and link pipeline registers may be inserted at certain position along the routing channels that balances the routing channel frequency with the router’s internal logic frequency; such approaches are useful when the NoC is intended to operate at a lower clock frequency lower than the one given by the router’s internal timing. Ultimately, the NoC frequency limit will be the internal timing paths within the routers or bridges.

* The credit based flow control and point to point links has one additional benefit. Since the mesh spans a significant area on the chip and runs synchronously, significant skew is possible between various routers. However, since communication only occurs between neighboring routers, hence only the local skew between adjacent routers/bridges, as opposed to the global skew, matters for timing closure. Since additional pipeline stages are easily added, these increased requirements will not jeopardize timing closure.

# From RTL generation to Synthesis

NocStudio, when enabled generates the RTL for the given configuration. Generated RTL can be synthesized using reference Synthesis Environment.

Figure 6 shows the different stages of the flow from Verilog RTL generation to Synthesis.

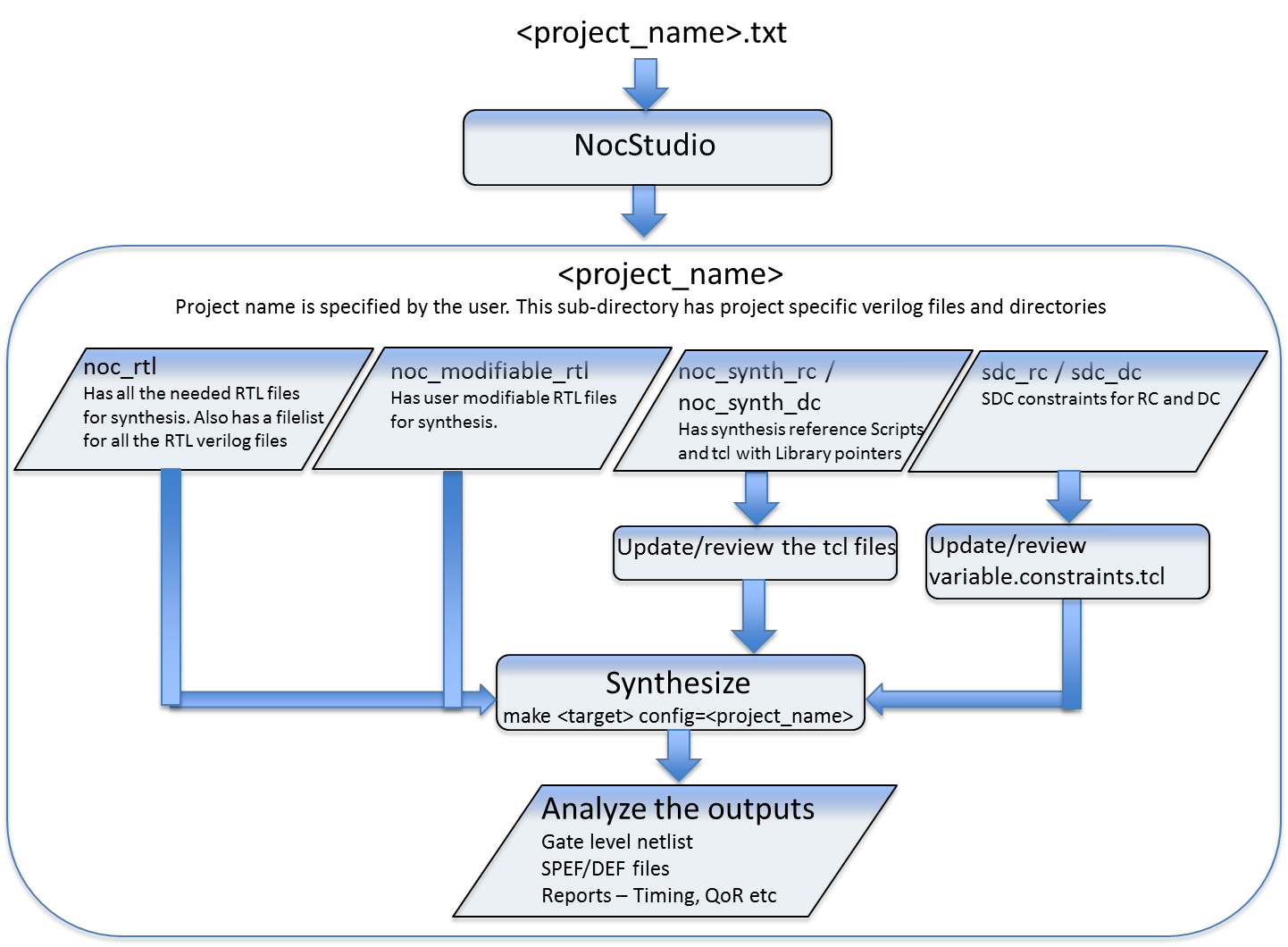


Figure 6: An example flow chart explaining steps from NocStudio to Synthesis output

A particular NoC configuration is defined in “<project\_name>.txt”. This file is the input file to NocStudio. NocStudio then generates <project\_name> subdirectory.

For more details

* About NocStudio, please refer to NocStudio User Manual.
* On sub-directories and Synthesis environment, please refer to Section 4.

# RTL Netlist Structure

NocStudio produces a Top level NoC Verilog RTL netlist (ns\_soc\_ip.v) which instantiates NoC fabric hierarchy (ns\_fabric.v) which further instantiates Router and Bridge modules for the NoC with a proper interconnect. If RTL grouping is enabled, RTL groups (Bridges/Routers) will be declared in a separate file (ns\_group\_modules.v) and will be instantiated in NoC fabric hierarchy (ns\_fabric.v). The number and type of each sub-module is assigned by NocStudio and is dependent on the NoC design.

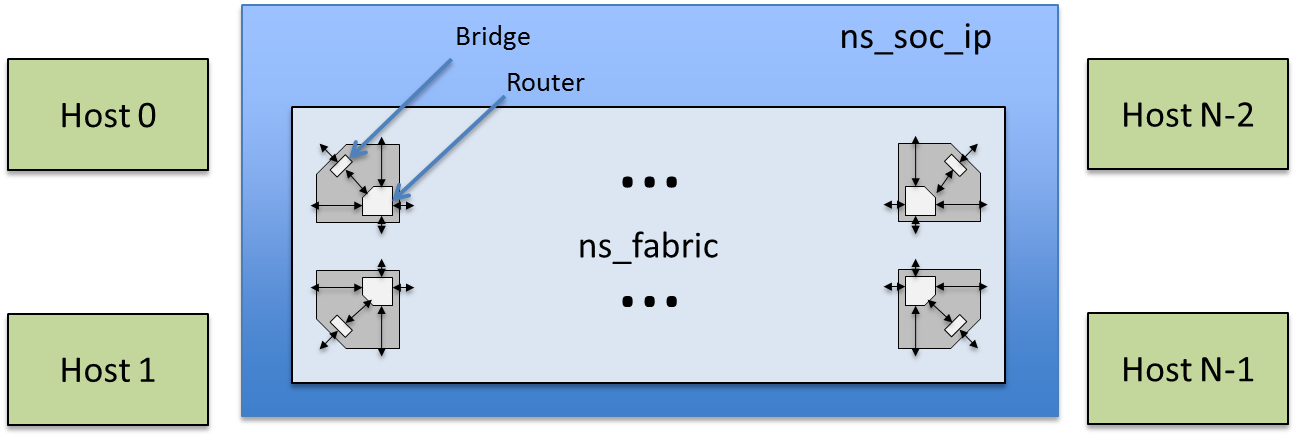


Figure 7: NoC RTL Netlist Structure

Each host port interfaces to the NoC through a Bridge. The width of both the host port interface bus and NoC link are configurable in NocStudio. The NoC Router contains 8 ports, which are named North, East, South, West and Host ports which are H (shown in the figure as South East), I (shown in the figure as South West), J (shown in the figure as North West), K (shown in the figure as North East).

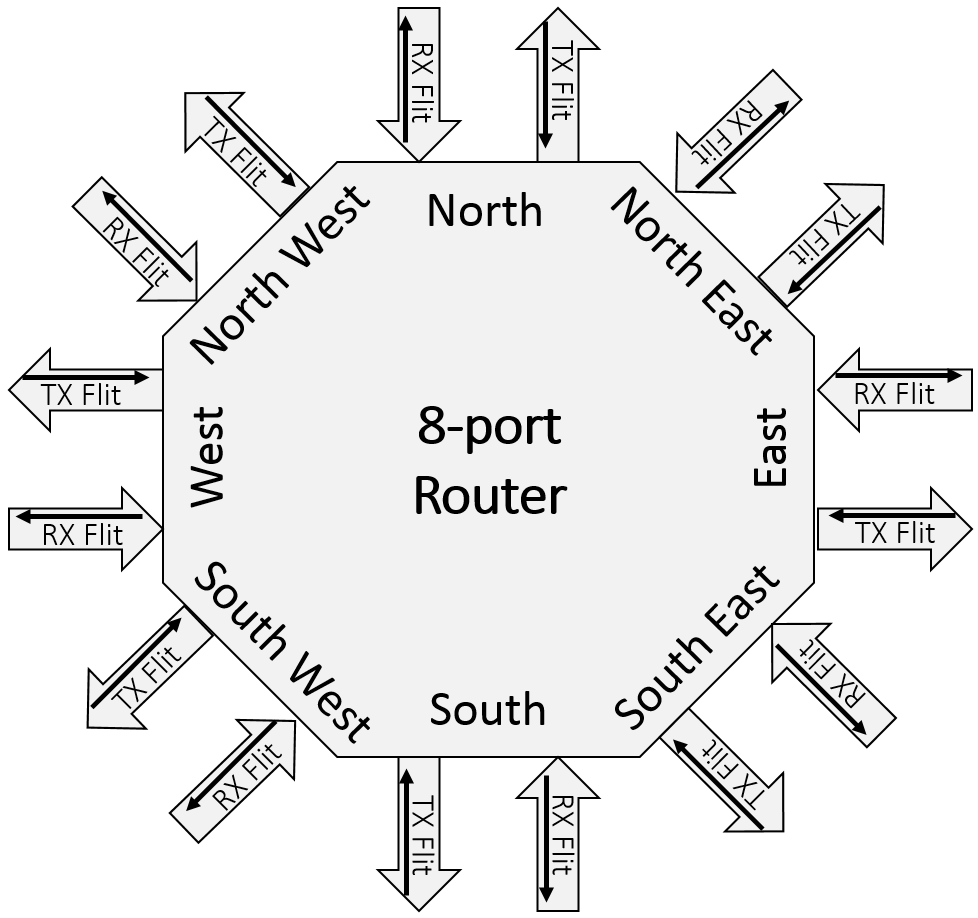


Figure 8: 8 Port Router with 4 directional ports & 4 host ports

The Router and Bridge elements are created so that the NoC Top level file contains no parameters and no logic elements.

## Multiple NoC Router Layers

To support additional bandwidth and virtual channels, NetSpeed’s NoC solution allows a single Bridge to connect to multiple NoC layers. Each layer operates as an independent NoC; the bridge connects them at the boundary. A Bridge at a grid point can connect to all routers at the grid point, one router for each layer.

The following diagram shows an example of this connectivity with 2 NoC layers. For physical design, this means that more wires are needed in order to connect the Bridges to the Routers. Also note that multiple layers may use different bus widths, so the connections to the routers of different layers may vary in width.

NoC

Router

X,Y

NoC

Router

X,Y

Bridge

Host A

Host B

NoC

Router

X,Y

NoC

Router

X,Y

Bridge

Figure 9: Dual NoC layers

## Naming convention

The naming convention of the various NoC components in the generated NoC RTL is as follows

1. Router – ns\_router\_<layer\_number>\_<node\_number>
2. Master bridges – ns\_aximstrbrdg\_<host\_name>\_<port\_name>
3. Slave bridges – ns\_axislvbrdg\_<host\_name>\_<port\_name>

## Example

Following is the example for example\_synth.txt config in “examples” directory. This example is of 2-layer NoC with two AXI Master and two AXI Slave hosts. The first layer is used for load data and store transactions, while the second is used for load command and store response.

The NocStudio GUI snapshot of this example is shown below.

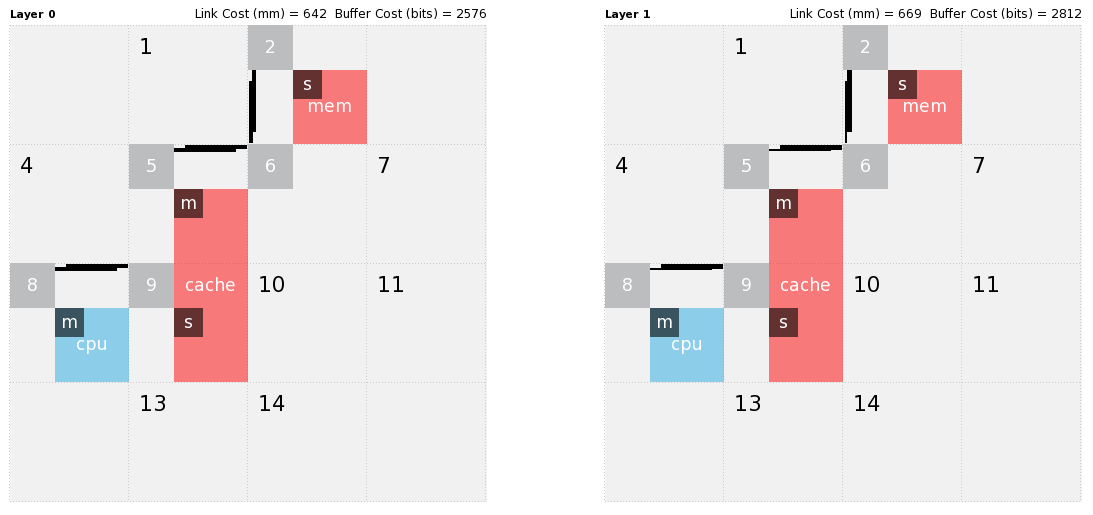


Figure 10: NocStudio GUI snapshot

NocStudio generates the top level RTL file “ns\_soc\_ip.v” in directory “example\_synth” which is the project name specified by user. The top level module name is “ns\_soc\_ip” and it instantiates “ns\_fabric” which further instantiates following NoC elements

* Two master bridges “ns\_aximstrbrdg\_cache\_m” and “ns\_aximstrbrdg\_cpu\_m”
* Two slave bridges “ns\_axislvbrdg\_cache\_s” and “ns\_axislvbrdg\_mem\_s”
* Six routers “ns\_router\_0\_5”, “ns\_router\_0\_6”, “ns\_router\_0\_9”, “ns\_router\_1\_5”, “ns\_router\_1\_6” and “ns\_router\_1\_9”

NocStudio also generates “ns\_fabric\_modules.v” in directory “example\_synth” which has parameterized instance definition of routers and bridges. In this example this file will have module definition of “ns\_aximstrbrdg\_cache\_m”, “ns\_aximstrbrdg\_cpu\_m”, “ns\_axislvbrdg\_cache\_s”, “ns\_axislvbrdg\_mem\_s”, “ns\_router\_0\_5”, “ns\_router\_0\_6”, “ns\_router\_0\_9”, “ns\_router\_1\_5”, “ns\_router\_1\_6” and “ns\_router\_1\_9”

# Synthesis

NocStudio, when enabled generates the RTL for the given configuration. Generated RTL can be synthesized using reference Synthesis Environment. Directories are highlighted **as bold**.

* **<project\_name>** // User defined Project name
* ns\_soc\_ip.v // NoC Top level instance that instantiates NoC Fabric, Group and Agent modules
* ns\_fabric.v // NoC Fabric that instantiates unique bridges and routers
* ns\_fabric\_modules.v // Parameterized module definition of NoC elements
* ns\_group\_modules.v // Group modules definitions
* **noc\_rtl**
  + \*.v, \*.vh // NoC library RTL in Verilog-2001 format
  + noc\_rtl.vc // Manifest file containing path to library
* **noc\_rtl\_agents**
  + ip/\*.v, ip/\*.h // NoC Agents library RTL in Verilog-2001 format
  + tunnel/\*.v, tunnel/\*.h // NoC Agents library RTL in Verilog-2001 format
  + noc\_rtl\_agents.vc // Manifest file containing path to NoC agent library
  + noc\_rtl\_agents.llc.vc // Manifest file containing path to Pegasus (LLC) library
* **noc\_modifiable\_rtl** // Users can modify these RTL files for RAM

**//** models, clock gating cells, register files

**//** and synchronizer units

* + \*.v // NoC library RTL that can be modified by user
* **synth** // Synthesis scripts for RC/RCP & DC/DCT
  + **defs** // NocStudio generated DEF files
    - ns\_soc\_ip\_afp.def // DEF file with Bridge pins
    - ns\_soc\_ip.def // DEF file with NoC regions
  + **noc\_synth\_rc** // Synthesis scripts for RC/RCP
    - **sdc\_rc**  // Contains RC constraints
      * variables.constraints.tcl // Common variables
      * ns\_soc\_ip.constraints.tcl // Top level constraints
      * <fabric\_modules>.constraints.tcl // Block level constraints
    - **rm\_rc\_scripts** // RC/RCP scripts
      * rc.tcl // Synthesis script for Cadence RC/RCP
    - **rm\_setup**
      * tech.tcl // Pointers to Technology files
      * vars.tcl // User defined variables set for Synthesis
      * dont\_use.sdc // List of don’t\_use cells
    - Makefile // Makefile to launch synthesis runs
    - synth\_rc.sh // Shell script that for RC Hier Synthesis runs
    - synth.tcl // Tcl file used for NoC level stitch (used only for

// RC hierarchical Synthesis)

* + **noc\_synth\_dc** // Synthesis scripts for DC/DCT
    - **sdc\_dc**  // Contains DC constraints
      * variables.constraints.tcl // Common variables
      * ns\_soc\_ip.constraints.tcl // Top level constraints
      * <fabric\_modules>.constraints.tcl // Block level constraints
    - **rm\_dc\_scripts** // Contains DC/DCT and Formality scripts
    - **rm\_setup** // Contains setup files
    - Makefile // Makefile to Synthesize NoC
    - synth\_dc.sh // Shell script that for DC Hier Synthesis runs

## Synthesis Methodology

The reference synthesis environment supports top down and hierarchical synthesis flow. In top level synthesis flow, the NoC is synthesized as a block. In hierarchical synthesis flow, the sub-blocks are synthesized first and then are stitched together at the top level. Synthesis can be done using wire load models or can have more physical awareness. NoC and its components can be synthesized by any synthesis tool.

The RC/RCP reference synthesis scripts can either be run using Cadence RTL Compiler (RC) which uses wire load models for estimating the loads or can be run using Cadence RTL Compiler Physical (RCP) which uses LEF and CAPTABLE or QRC technology file provided by the foundry for parasitic extraction.

The DC/DCT reference synthesis scripts can either be run using Synopsys DC Compiler (DC) which uses wire load models for estimating the loads or can be run using Synopsys DC Topographical Compiler (DCT) which uses TLU plus library files provided by the foundry for parasitic extraction

It is recommended to carefully partition the design according to physical locality of the logic. For example, in a particular Cell/Node of the mesh, if the bridge(s) and router(s) are physically placed close/adjacent to each other, then Fmax, Area and Power will be optimum if they are kept in the same hierarchy.

## Running Synthesis

Following tools are needed to run the reference synthesis flow

### For Cadence flow:

* Cadence RTL Compiler (RC) – For doing synthesis with wire load models
* Cadence RTL Physical Compiler (RCP) – For doing physical synthesis

Following is an example procedure for running Synthesis using Cadence tool set

* Update tech.tcl file – tech.tcl file sets attributes for
  + Liberty library search path
  + Liberty library file names
  + Pointer for LEF files – These are needed for Physical Aware Synthesis
  + Pointer for CAPTABLE file – This is also needed for Physical Aware Synthesis
* Update vars.tcl file – vars.tcl file sets the variables for
  + VT Usage – Selects the VT library to use, lvt|rvt|hvt
  + RC vs RCP – Whether to run RC or to run RCP (rcp 0/1)
  + DFT - Whether to insert scan chains or not (dft 0/1)
  + Number of Scan chains to insert
  + Synthesis effort - Whether to synthesis with low|medium|high effort
* Review and update “sdc\_rc/variables.constraints.tcl” for clock periods, clock uncertainties, clock groups, input/output delay margins, and output load. Input/output delay margins should be updated keeping in mind the technology node and (for hierarchical synthesis) if a bridge/router has output registering enabled or disabled
* Do “make top config=<project\_name>” to do top level synthesis
* Do “make hier config=<project\_name>” to do hierarchical synthesis

### For Synopsys flow:

* Synopsys DC Compiler (DC) – For doing synthesis with wire load models
* Synopsys DC Topographical Compiler (DCT) – For doing physical synthesis

Following is an example procedure for running Synthesis using Synopsys tool set

* Update rm\_setup/common\_setup.tcl file – This file sets attributes for
  + Search path
  + Liberty library file names
  + Pointer for TLUplus, Milkyway Physical library, Tech and MAP files – These are needed for Physical Aware Synthesis
* Review and update “sdc\_dc/variables.constraints.tcl” for clock periods, clock uncertainties, clock groups, input/output delay margins, and output load. Input/output delay margins should be updated keeping in mind the technology node and (for hierarchical synthesis) if a bridge/router has output registering enabled or disabled
* Do “make top config=<project\_name>” to do top level synthesis
* Do “make hier config=<project\_name>” to do hierarchical synthesis

Note: Clock gating can be can be implemented on NoC elements using Synthesis tools. Currently it is not supported in reference Synthesis scripts.

## Analyzing outputs

### Analyzing RC/RCP synthesis output

Top level and hierarchical synthesis creates the following files and directories

* <module>.gv – Gate level file generated by the tool
* <module>.spef – Parasitic file generated by the tool. This is only generated when physical synthesis is enabled
* <module>.def – DEF file generated by the tool. This is only generated when physical synthesis is enabled
* <module>\_outputs - This directory has sub-directories to store generated outputs at each stage i.e. synthesis stage, mapping stage, placement stage and final stage. The outputs that are saved are gate level files, DEF file, Scandef file, and the Encounter files.
* <module>\_reports - This directory has sub-directories to store generated reports at each stage.

### Analyzing DC/DCT synthesis output

Top level and hierarchical synthesis creates the following files and directories

* <module>\_outputs - This store generated outputs.
* <module>\_reports - This store generated reports.

# Place and Route Guidelines

Place and Route options greatly depend on the chip overall design methodology. This section gives basic guidelines for NoC physical implementation.

## P&R keeping the NoC elements together

Bridge and Router at single mesh grid point may be combined in to a single P&R region. This may make sense when the Host is a monolithic hard macro. This P&R region would reside outside the Host.

NoC

Router

X,Y

Bridge

Host A

NoC

Router

X,Y

Bridge

Host B

Figure 11: NoC Elements in stand-alone P&R regions

## P&R merging NoC Elements with Host

When multiple hosts are soft macros, it may be advantageous to simply merge the associated NoC elements with the host into a single P&R region, as shown on the left “P&R Block 1” in Figure 12. This may reduce the number of top-level regions and/or macros and the number of top-level signals.

NoC

Router

X,Y

Bridge

Host A

NoC

Router

X,Y

Bridge

Host B

P&R

Block 1

Figure 12: Merge all NoC Elements with Host P&R Region

## NoC Quick Start Information

### NoC clock domains

NoC instantiates groups of NoC bridges and corresponding hosts/agents.

AXI NoC operates with a common clock or a clock that is completely asynchronous. If clock is asynchronous, NoC provides an asynchronous mode by putting an asynchronous FIFO between the bridge and the agent as shown in Figure 13. A minimum clock skew should be maintained in each domain.

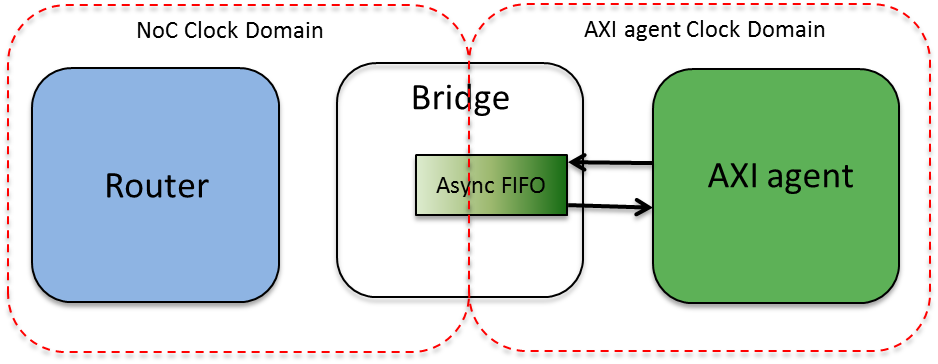


Figure 13: Bridge and AXI agent with async clock

Figure 14 is one of the examples of implementation where Host A has the same clock as NoC clock and Host B clocks with an independent clock domain.

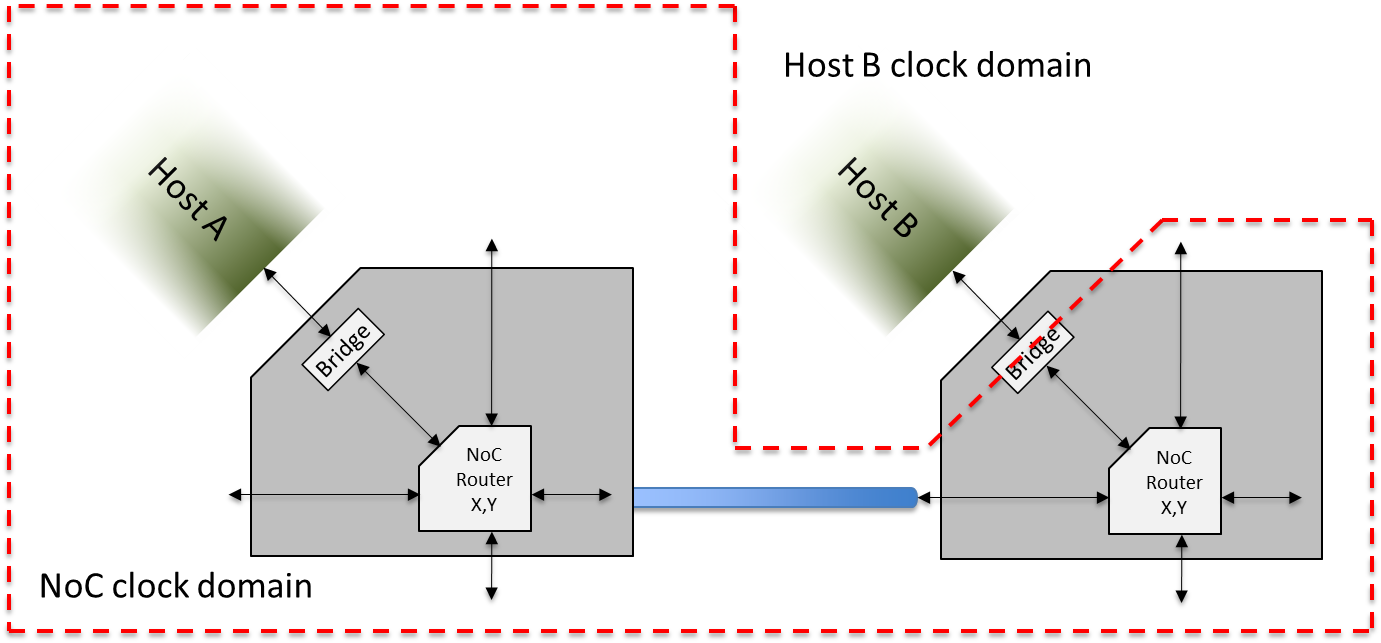


Figure 14: Host A and NoC clock domain different than Host B clock

Figure 15 is another implementation where Host A and Host B clock domains are different than NoC clock domain.

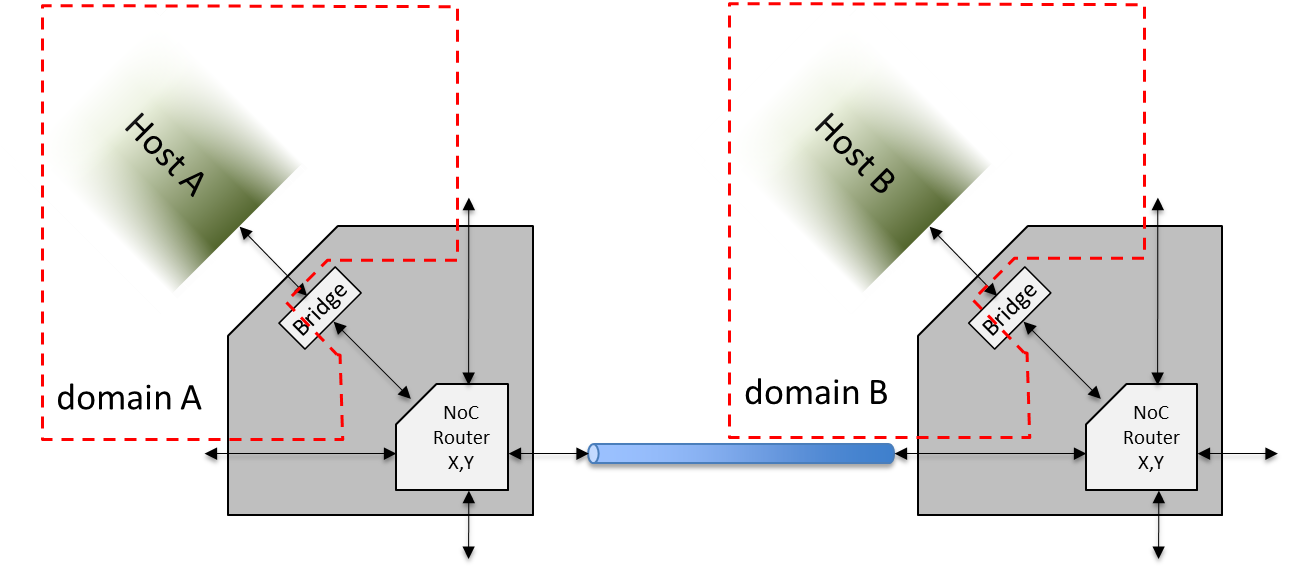


Figure 15: Host A & B with different clock domains than NoC clock

### Clock Gating Cell

NetSpeed NoC RTL uses “ns\_cg\_cell” as the clock gating cell. The definition of the cell is given below

module ns\_cg\_cell

(

input en,

input clk,

input scan\_mode,

output eclk

);

`ifndef GATES

reg d\_latch;

wire cg\_en;

assign cg\_en = scan\_mode | en;

always @(cg\_en or clk) begin

if (~clk)

d\_latch <= cg\_en;

end

assign eclk = d\_latch & clk;

`else

PREICG\_X2B\_A9TR(.CK(clk), .E(en), .ECK(eclk), .SE(scan\_mode));

`endif

endmodule

This module can be found in noc\_modifiable\_rtl directory of the tarball. For synthesis purposes user can define the clock gating cell from their library instead of “PREICG\_X2B\_A9TR” mentioned above.

Special attention should be given while routing “\*\_cg\_busy” signals between NoC elements. These are timing critical clock gating signals. It is recommended to use higher layers (thicker) for these signals when possible. Non-minimum spacing and shielding will reduce crosstalk related

### NoC Clock tree

NoC implementations can span the entire chip; however the concern can be the overall clock skew budget. To address this concern implementer can break up overall chip clock skew into two skew groups, global clock skew group and local clock skew group. NoC is architecturally independent of global clock skew. It is really important to keep local clock skew as minimum as possible.

### Reset

The NoC elements are reset using an asynchronous assert, synchronous de-assert reset scheme. Figure 16 shows the reset logic used in each NoC component.



Figure 16: Reset logic

Top level reset timing requirements are:

* Reset is asserted for at least 16 NoC core clock cycles.
* Reset should be identical to all modules (bridge, router, pipeline and register bus modules) i.e. reset falling edge should be in the same cycle for each NoC element.

### Standard Cells

All NoC elements are synthesizable using generally available standard cell libraries. The required elements can minimally be expressed as rising-edge triggered flip-flops, async set flip-flops (for reset logic in Figure 16), basic 2 and 3 input logic functions (nand, nor, and, or, xor), multiplexors, and buffers and inverters. No special cells like one hot muxes or any other pass-gate type of cells are required and the resulting design is fully scan compliant and easy to constrain using SDC.

### Channel Routing

Routing between NoC Routers are potentially wide parallel buses. As with most full chip routing it is recommended to use higher layers (thicker) for longer routes when possible. Non-minimum spacing will reduce crosstalk related signal delays.

### Repeaters

All NoC interconnections are point to point which should work well with whatever top-level repeater flow is in place. As usual, good repeater methodology is encouraged. Design frequencies and targets vary so some rule of thumb guidelines are presented.

* Keep edge rates below 20% of the cycle time
* Non minimum metal spacing to reduce crosstalk (1.5-2.0x if possible)
* Favor the higher (thicker) metal layers for long signal runs

Interconnect from one router to another is designed/generated by NocStudio according to assumptions about wire delays (RC & repeater delays).  If the physical design does not match those assumptions, the architect should be informed so that NocStudio parameters can be updated.

### Repeaters & Power Grid

If the selected NoC design uses very wide buses to achieve high data bandwidth, care should be taken to analyze the standard cell power grid. Very often the power mesh for standard cells is determined using metrics corresponding to generalized logic blocks where the percentage mix of device drive strengths favors smaller devices. In these generalized logic blocks switching of these smaller devices is spread throughout the cycle (though biased just after the rising edge of clock) as logic cascades down subsequent stages. In the NoC router, the number of logic levels is kept to a minimum to provide low latency. When the NoC design generated by NocStudio possesses very wide buses, it is likely that the number of large drivers simultaneously switching within a small area exceeds that of a standard logic mix. It is suggested that early validation of dynamic IR drop be performed to assure adequate design margin.

### Synchronizers

All synchronizers within RTL generated by NocStudio are referenced from a common module. This module uses fixed names for the DFFs which comprise the synchronizer making it easy to identify this in PD (and in verification flows). If a synchronizer cell exists in the library, providing a gate level netlist for the single **ns\_demet** module can provide a simple way to assure the proper gate mapping.

DFF

DFF

*ns\_demet*

clk

sync\_out

async\_in

*demet\_stage1\_q*

*demet\_stage0\_q*

*rst*

*rst*

arst

Figure 17: NetSpeed synchronizer module

All asynchronous clock domain crossings are handled via FIFOs where the control signal is synchronized through a 2-stage synchronizer. It is encouraged that PD flows avoid placing buffers between the 1st and 2nd stage flip-flops of the synchronizers. The amount of apparent slack from the 1st stage to the 2nd stage will increase the MTBF exponentially. This is more important for high frequency designs. All synchronizer flip-flops may be located by using the following regular expression “.\*demet\_stage.\*”.

# DFT

*Testability* is a design attribute that measures how easy it is to create a program to comprehensively test a manufactured design’s quality. Traditionally, design and test processes were kept separate, with test considered only at the end of the design cycle. But in contemporary design flows, test merges with design much earlier in the process, creating what is called a *design-for-test (DFT)* process flow.

All the NoC elements are DFT compliant and yield high test coverage ensuring higher Test quality.

2670 Seely Ave

Building 11

San Jose, CA 95134

(408) 914-6962

<http://www.netspeedsystems.com>