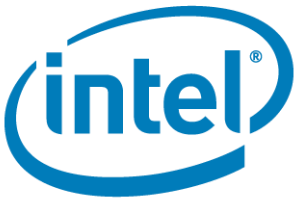

CFG Crux

IP Integration Specification

Version: CRUX-19.07L

Revision: 0.0

Intel Confidential

This document contains information on products in the design phase of development.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED OR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your Intel account manager or distributor to obtain the latest specifications and before placing your product order.

Copies of documents that have an order number and are referenced in this document or in other Intel literature can be obtained from your Intel account manager or distributor.

CFG Crux IP Integration Specification

About This Document

This document describes the guidelines for seamless integration of CFG Crux IP. This includes details of the IP components and instructions on how to integrate them into customer SoC. The registers in NoC RTL can be accessed via the CFG configuration bus which is described in the HTML documentation generated by NocStudio for each user input configuration.

Audience

This document is intended for users of NocStudio:

- NoC Architects, Designers and Verification Engineers
- SoC Architects, Designers and Verification Engineers

Prerequisite

Before proceeding, you should generally understand:

- Basics of Network on Chip technology

Related Documents

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

- CFG NocStudio Crux User Manual
- CFG Crux Technical Reference Manual

Customer Support

For technical support about this product and general information about CFG products, contact CFG Support.

Revision History

Revision	Date	Updates
0.0	Jul 19, 2019	Initial Version

Contents

About This Document **3**

Audience **3**

Prerequisite **3**

Related Documents **3**

Customer Support **3**

1 NoC IP Overview **7**

1.1 NoC IP Components 7

1.2 Directory Structure 7

1.3 Documentation 8

1.4 NocStudio Flow to Generate NoC IP 9

2 Integration of NoC **19**

2.1 Integration of NoC RTL 19

2.2 Integration of NoC Verification Checkers 28

2.3 Supported Tools 30

3 Integration of Multiple NoCs **31**

3.1 Automated Integration of Multiple NoCs 32

3.2 Manual Integration of Multiple NoCs 33

4 NoC Verification Components **36**

4.1 Overview of Checkers 36

4.2 Environment Setup for Integration 36

4.3 Usage Modes 37

4.4 Checkers 38

5 Physical Design Guidelines **49**

5.1 RTL Netlist Structure 49

5.2 Synthesis 53

5.3 DEF 57

5.4 CDC 57
6 Place and Route Guidelines 60
6.1 P&R keeping the NoC elements together 60
6.2 P&R merging NoC Elements with Host 60
6.3 NoC Quick Start Information 61
7 DFT 70
8 C++ NoC Model Integration 71
8.1 How to Simulate Model 71
8.2 How to Include Model 75
8.3 How to Configure Model 76
8.4 Simulation Performance 76
8.5 Utilities 78
8.6 Summary of NoC Model APIs 80
9 Waivers 83
9.1 Lint Waivers 83
9.2 CDC Waivers 87

1 NoC IP Overview

1.1 NoC IP Components

The NoC IP release package contains the following main components:

- NocStudio binary
- NocStudio usage examples
- RTL library
- Verification library
- User manuals and documentation

In addition, NocStudio generates the following for every user-specified system described in a NocStudio command script:

- NoC RTL
- NoC verification checkers
- Sanity testbench for the generated NoC
- Comprehensive html specification for the generated NoC

1.2 Directory Structure

Table 1 NoC IP directory structure

Name	Description
custom_header.txt	Custom header content, modifiable by the user, which is inserted in all auto-generated NoC files
examples/*	Example NocStudio command scripts from user manual and demonstrating feature usage.
lib/*	NocStudio dynamic libraries.
noc_doc_images/*	Support files for NoC html documentation generation.
noc_help_images/*	Support files for NoC help manual
noc_modifiable_rtl/*	RTL modules, such as RAM, that can be replaced by user implementation.

noc_rtl/*	NoC RTL library.
noc_rtl_agents/*	NoC RTL agents IP library.
NocStudio	NocStudio executable.
noc_verif_agents/*	NoC verification agents IP library.
noc_verif_bench/*	NoC sanity testbench library.
noc_verif_cust/ns_global_defines.vh	`defines file used for integration of CFG verification IP into customer environment.
noc_verif_ip/*	NoC verification checkers IP library.
scripts/*	Scripts for sanity bench.
synth/*	NoC synthesis environment.
tutorials/*	NocStudio tutorials.
user_manual_files/*	Support files for NocStudio manuals.

1.3 DOCUMENTATION

A separate package will be delivered with the following documents.

Table 2 NoC IP document list

Name	Description
NocStudio User Manual.pdf	The User Guide describes how to set up a system using NocStudio and how to use it to generate CFG IP
IP Integration Spec.pdf	The Integration Manual describes how to integrate a configured network into a larger subsystem (this document).
Technical Reference Manual.pdf	The Technical Reference Manual describes how the functionality of the various NoC elements, the features and functions available, and how to dynamically change the functions using the programmers mode.

NocStudio generated documents:

Name	Description
NocStudio Command Reference	Available in two forms: 1. NocStudio toolbar help. 2. Generated HTML document.

noc_reference_manual.html	Per project reference manual containing NoC project architecture details, registers, etc.

1.4 NocStudio Flow to Generate NoC IP

Figure 1 describes the NoC IP generation flow using NocStudio. The user specifies a NocStudio command script that describes the user system requirements. NocStudio processes this script to construct a deadlock-free NoC that meets all the system requirements. The following files are generated by NocStudio for the NoC:

- NoC RTL
- NoC verification IP
- Sanity testbench
- Synthesis scripts
- HTML specification for the generated NoC

All the generated files are output to the project directory. The name of the project directory corresponds to the project name specified in the new_mesh command in the NocStudio command script.

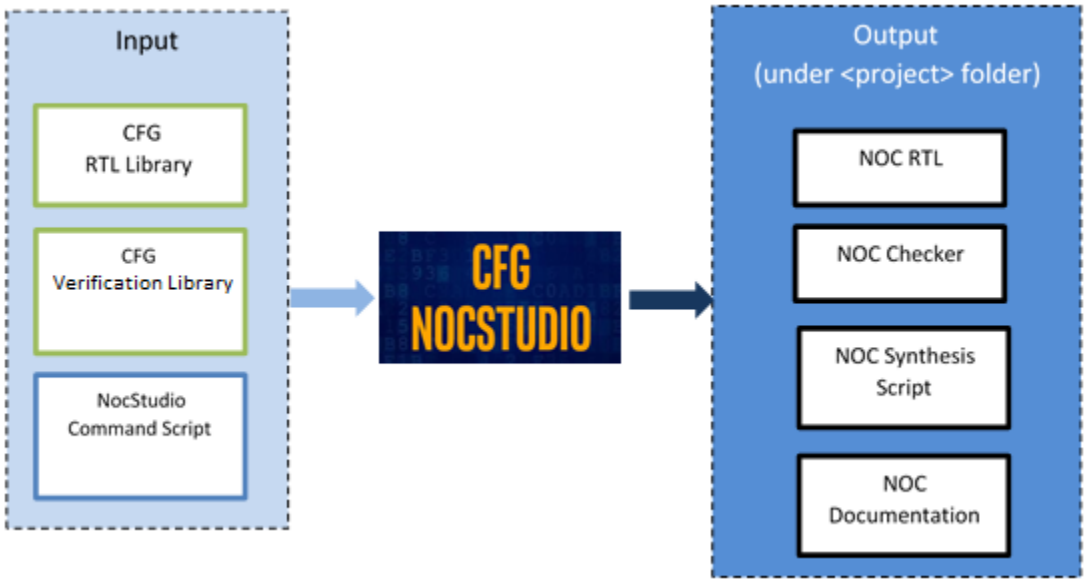

Figure 1 NoC IP generation flow

1.4.1 Generating RTL from NocStudio

To generate NoC RTL, include the gen_ip command at the end of the NocStudio command script, and then process the script with NocStudio.

For example, from the IP root directory, run the following command for GUI mode:

```
./NocStudio examples/example_cache1.txt
```

Or, the following command for batch mode:

```
./NocStudio examples/example_cache1.txt  -nogui
```

Once the gen_ip executes, a project directory called example_cache1/ is created which contains all the files and directories generated by NocStudio. Below is a list of key files related to RTL and verification component integration. For a complete list with detailed descriptions please refer to CFG NocStudio Gemini User Manual.

Table 3 Files and directories generated by NocStudio in the project directory

Name	Description	Type
archive/*	Location into which old project directory output is placed when a new project directory of same name is created.	Archive
bridge_prop.csv	Information about the bridges in the NoC.	NoC property
buffer_costs.csv	Flip-flop count estimate.	NoC property
commands.log	Command file from the NocStudio run.	Log file
doc_files/*	Support files for HTML documentation.	HTML documentation
link_costs.csv	Wiring cost estimate.	NoC property
noc_modifiable_rtl/*	RTL modules, such as RAM, that user can replace with own implementation. See Table 4 Files in the noc_modifiable_rtl directory.	RTL
noc_reference_manual.html	HTML specification for the generated NoC with information such as layer diagrams, traffic dependencies, full register specification and more.	HTML documentation
noc_registers.csv	List of NoC registers.	NoC property
noc_rtl/*	NoC RTL library.	RTL
noc_rtl_agents/*	NoC RTL agents IP library.	RTL
noc_verif_agents/*	NoC verification agents IP library.	RTL
noc_verif_bench/*	NoC sanity testbench library.	Sanity testbench
noc_verif_cust/ns_global_defines.vh	`defines file used for integration of CFG verification IP into customer environment.	Verification
noc_verif_ip/*	NoC verification checkers IP library.	Verification

ns_bind_checkers.svh	Bind file of verification checkers to corresponding RTL modules based on the generated ns_fabric.v.	Verification
ns_fabric_modules.v	Support RTL modules for the generated NoC.	RTL
ns_fabric.v	RTL module for the generated NoC.	RTL
ns_group_modules.v	Hierarchical RTL group modules for the generated NoC. This file will be empty if user did not create any groups.	RTL
ns_noc_files.f	File list to compile NoC RTL, NoC RTL agents and NoC verification IP; Intended for integration of RTL and NoC verification components into customer environment.	RTL and Verification
ns_node_id_table.sv	Support file for Register Bus End-to-End Checker generated for the NoC by NocStudio.	Verification
ns_preloader_declare.sv	Support file for CFG Pegasus Preloader generated for NoC by NocStudio. NOTE: Only present for Pegasus.	Verification
ns_preloader_init.sv	Support file for CFG Pegasus Preloader generated for NoC by NocStudio. NOTE: Only present for Pegasus.	Verification
ns_preloader_set.sv	Support file for CFG Pegasus Preloader generated for NoC by NocStudio. NOTE: Only present for Pegasus.	Verification
ns_routing_table.sv	Support file for NoC End-to-End Checker generated for the NoC by NocStudio.	Verification
ns_soc_ip.v	Top level RTL module for the generated SoC IP.	RTL
protocol_dependencies.csv	List of dependencies for the generated NoC.	NoC property
run_sramtest.sh	Run script to launch sanity bench for the SRAM instances present in the system (if any) using either Synopsys VCS simulator or Cadence Incisive simulator. NOTE: Only generated if SRAM instances are present.	Sanity testbench

run_test.sh	Run script to launch sanity bench for the generated NoC using either Synopsys VCS simulator or Cadence Incisive simulator. Please invoke scripts/run_test.pl -h for a set of options.	Sanity testbench
scripts/*	Scripts for sanity testbenches.	Sanity testbench
sram_verif/*	SRAM sanity bench directory. NOTE: Only generated if SRAM instances are present.	Sanity testbench
synth/*	NoC synthesis environment.	Synthesis
system.f	File list for full RTL compile (no System Verilog).	Sanity testbench
tb_system_rtl.f	File list for register bus test compile. Includes NoC RTL, NoC RTL IP agents (CCC, IOCB, LLC), NoC verification tunnel agent, NoC verification IP and NoC sanity testbench directories.	Sanity testbench
tb_system_verif.f	File list for NoC traffic test compile. Includes NoC RTL, NoC verification IP agents (CCC, IOCB, LLC), NoC verification tunnel agent, NoC verification IP and NoC sanity testbench directories.	Sanity testbench
top.v	Top file for the sanity testbench.	Sanity testbench
trace/*	Trace files for the NoC traffic test.	Sanity testbench
trace_regbus/*	Trace files for the register bus test.	Sanity testbench
traffic/*	Traffic information specified by the user.	NoC property
transcript.log	Log file from the NocStudio run.	Log file

Table 4 Files in the noc_modifiable_rtl directory.

Name	Description	Type

ns_cmn_2p_rf.v	An RTL wrapper over a 2-port register file model. Contents can be replaced with 2-port register file primitive from user library. If left untouched, this would be synthesized into flip-flops. The register file does not depend on validity of read data in a cycle where write enable is active and read and write addresses are the same.	RTL
ns_ccc_rf256_byen.v	An RTL wrapper over a 2-port register file model that includes byte-enables for the write-port. Contents can be replaced with 2-port register file primitive from user library. If left untouched, this would be synthesized into flip-flops. The register file does not depend on validity of read data in a cycle where write enable is active and read and write addresses are the same.	RTL
ns_ccc_sram1p_mdl.v	This wrapper contains a single port SRAM model. This is the CCC RAM model and can be replaced with appropriate single port RAM from user's library. NOTE: Only present for Gemini.	RTL
ns_cg_cell.v	Clock gating cell model. Contents can be replaced by suitable clock gating cell from user's library.	RTL
ns_demet.v	Async domain crossing synchronization module.	RTL
ns_llc_data_ram.v	This wrapper contains a single port SRAM model. This is the LLC data RAM model and can be replaced with appropriate single port RAM from user's library. NOTE: Only present for Pegasus.	RTL
ns_llc_tag_ram.v	This wrapper contains a single port SRAM model. This is the LLC tag RAM model and can be replaced with appropriate single port RAM	RTL

	from user's library. NOTE: Only present for Pegasus.	
ns_rst_n.v	Reset synchronization module. Converts an active low asynchronous reset to async assert and sync de-assert for each clock domain.	RTL

1.4.2 NoC Sanity Testbench

Once NocStudio generates NoC RTL, the next step is running the sanity testbench to perform a sanity check on the generated NoC RTL in simulation. This is a push-button method of instantiating the generated NoC RTL in a sanity testbench along with verification checkers and running a sanity traffic pattern on the generated NoC RTL to validate connectivity and basic operation of the NoC in simulation.

To run the NoC sanity test, change to the project directory and invoke the following run script

If you are using Synopsys VCS Simulator, run:

```
run_test.sh -VCS
```

If you are using Cadence Incisive Simulator, run:

```
run_test.sh
```

The run script compiles the sanity testbench and launches the simulation.

To enable waveform dumping, use the command line option -waves=1. For example:

```
run_test.sh -VCS -waves=1
```

Or, for Cadence Incisive Simulator:

```
run_test.sh -waves=1
```

The waveform database will be generated inside the simulation trace directory. On a successful compile and simulation, the following will appear at the prompt:

```
Passing to irun for RTL-only build
BUILD SOC SUCCESSFUL
Passing to irun for regbus sanity bench build
BUILD SUCCESSFUL
Passing to irun for simulation
**************************
* REGBUS SIMULATION PASSED *
**************************
Passing to irun for NoC sanity bench build
BUILD SUCCESSFUL
Passing to irun for simulation
********************
* SIMULATION PASSED *
********************
```

After the completion of the simulation, the presence of a file named SIM_FAILED in the project directory indicates that the simulation failed. The presence of the file SIM_PASSED in the project directory indicates a successful simulation. Depending on the simulator, the log file run_test_vcs.log or run_test_incisiv.log will list any errors encountered during the build and run phase. The log file, named build.log, is located in the model/ directory. The simulation log from the NoC traffic test, named run.log, is located in the trace/ directory. The simulation log file from the register bus test, named regbus_run.log, is located in the trace_regbus/ directory.

After a successful NoC sanity testbench simulation, the generated NoC RTL and verification IP are ready for integration into the user's environment.

1.4.3 SRAM Sanity Testbench

If the design has instances of SRAM, the SRAM sanity testbench provides the basic test for RTL with the SRAM checker, which runs a fixed set of traffic patterns on each SRAM instance to validate basic operation of the SRAM in simulation. The intent of this standalone testbench is to pre-qualify any SRAM modules that the customer may swap into their design. The SRAM sanity test executes four sub-tests on each SRAM instance in the configuration: a latency test, a data bus walking one test, an address bus walking one test and a bandwidth test. For the 2-ported regfile RAMs, concurrent read and write test has been added in addition to the standard tests.

Example usage:

1. User replaces noc_modifiable_rtl folder with design specific implementation of SRAM model. The new file must have the same file name, module name and must be pin compatible with the file it is replacing.
2. Run SRAM sanity test to confirm that latency, size and bandwidth match user specification from the NocStudio command script.

To run the SRAM sanity test, change to the project directory and invoke the following run script

If you are using Synopsys VCS Simulator, run:

```
run_sramtest.sh -VCS
```

If you are using Cadence Incisive Simulator, run:

```
run_sramtest.sh
```

The run script compiles the sanity testbench, and then launches the simulation for each SRAM instance in the configuration.

To enable waveform dumping, use the command line option -waves=1. For example:

```
run_sramtest.sh -VCS -waves=1
```

Or, for Cadence Incisive Simulator:

```
run_sramtest.sh -waves=1
```

The waveform database will be generated inside the sram_verif/<sram_instance_name>/sim/ directory. On a successful compile and simulation, the following will appear in the log for each SRAM instance:

```
Changing to directory sram_verif/llc0_tag
Passing to irun for build
**********
* BUILD PASSED (llc0_tag) *
**********
Passing to irun for llc0_tag simulation
**********
* SIMULATION PASSED (llc0_tag) *
**********
```

After the completion of the simulation, the presence of a file named SIM_FAILED in the sram_verif/<sram_instance_name>/sim/ directory indicates that the simulation failed. The presence of the file SIM_PASSED indicates a successful simulation. Depending on the simulator, the log file run_sramtest_vcs.log or run_sramtest_incisiv.log will list any errors encountered during the build and run phase. The build log, named build.log, is located in the sram_verif/<sram_instance_name>/model/ directory. The simulation log, named run.log, is located in the sram_verif/<sram_instance_name>/sim/ directory.

2 INTEGRATION OF NOC

In the NocStudio project directory, ns_noc_files.f contains all the file references for NoC RTL and verification checkers. The following sections describe the integration process in detail.

2.1 INTEGRATION OF NOC RTL

To instantiate NoC RTL in your environment:

- Set the environment variable $NS_PROJ_PATH to point to the project directory that was created by NocStudio, for example:

```
setenv NS_PROJ_PATH /absolute/path/of/project/created/
```

- Include the following line in the file list for the project which instantiates the NoC.

```
-f ns_noc_files.f
```

- The top-level module is ns_soc_ip, specified in ns_soc_ip.v.

2.1.1 Hierarchical RTL generation

By default, NocStudio creates modules for each NoC element and these modules are interconnected to create the NoC. An option is available to create an additional level of hierarchy in the generated RTL by grouping certain NoC elements into their own hierarchy. This creates group modules interconnecting NoC modules belonging to that group. Group modules along with ungrouped NoC modules are interconnected at the next higher level to create the NoC.

Groups can comprise of any combination of routers, bridges, CFG RTL IP modules or NoC nodes. When grouped by NoC node, all routers and bridges at that node are added to the group module.

2.1.2 Reset

Table 5 NoC reset signals

Signal name	Description
reset_n_<power_domain>	Reset pins are named according to the power domain to which they belong. The reset pins are active-low. - Default power domain is system, and hence the default reset pin is reset_n_system.

	- All other power domains are added via the command add_power_domain, and they are named accordingly.
tst_rst_<power_domain>	Per power domain test reset input, active-high.
tst_rst_bypass_<power_domain>	Per power domain test reset bypass control, active-high.

One active low reset pin per power domain is present at the NoC level. Additionally, all NoCs provide a pair of DFT reset bypass pins per power domain (both active-high) that provide explicit control over reset in test modes. If test mode control over reset is not required in a particular application, these pins should be tied low.

Each set of reset pins is distributed asynchronously to all the NoC elements belonging to the associated power domain. Internally within the NoC elements, reset_n is captured in synchronizing structures that propagate the assertion edge asynchronously but force synchronous de-assertion of reset to local logic. The test reset signal, tst_rst_<power_domain>, locally overrides reset_n via multiplexors controlled by tst_rst_bypass_<power_domain> that sit at the outputs of the synchronizers. Hence test reset is fully asynchronously distributed to the local logic.

The customer must ensure that the reset pins are asserted long enough so that all NoC elements being powered up together are in reset at the same time. This is equal to the number of cycles for reset to propagate across the NoC + 16 (number of clock cycles that reset must be asserted at each NoC element). Since cold reset is typically a long operation, a safe way to do this is to assert reset for a large number of slow clocks – 100 ticks of the slowest clock in NoC system. Note that the synchronization of the de-assertion of reset in the NoC modules will take additional time. Traffic to the NoC should be delayed a number cycles of the slowest clock equal to the synchronizer depth after reset is de-asserted.

Physical distribution of reset to the various components of the NoC is the responsibility of the customer.

Each NoC element can come out of reset at different times. The link_available signal from a NoC element, if asserted, indicates to its neighbors that it has not yet come out from reset.

Additional details on the clocks, resets and physical integration and design guidelines are available in the Physical Design Guidelines document provided with the release.

2.1.3 Clocks

The following table lists the clock signal in the generated RTL.

Table 6 NoC clock signals

Signal name	Description
clk_<clock_domain>	Clock pins are named as per the clock domain they belong to - Default clock domain is noc, and hence default clock pin is clk_noc - Register bus clock domain is regbus and the corresponding clock pin is clk_regbus - All other clock domains are added via command add_clock_domain, and are named accordingly For example, a certain clock domain can be defined for the host clock of a bridge with an async, ratio_slow or ratio_fast clock crosser interface. The corresponding clk_<clock_domain> pin at the NoC level will be internally connected to the host clock pin of the bridge (NoC element)

The NoC IP may have different clock domains that run asynchronously to each other. Instructions for adding clock domains can be found in the CFG NocStudio Gemini User Manual. In addition, host interfaces can operate at a clock asynchronous to the NoC clocks. The register bus layer can also operate on a clock asynchronous to NoC clocks. The physical fan-out and distribution of the clock is the responsibility of the customer.

Clock crossing between hosts and the NoC can happen within a bridge or on link between the bridge and the connected router. A list of clock crossings that exist in the NoC is generated by NocStudio in noc_reference_manual.html. There are different kinds of clock crossings. The async clock crossing refers to an asynchronous clock, where the frequency and phase alignment of the clocks have no necessary relationship. The ratio_slow and ratio_fast are phase-aligned synchronous clock crossers with an N:1 or 1:N ratio. The ratio_slow refers to a clock crosser where the host is running slower than the NoC. ratio_fast refers to a clock crosser where the host is running faster than the NoC.

The synchronous clock crossers require a frequency relationship as well as a phase relationship. To achieve phase alignment, it is expected that the source of the ratio clocks will be the same.

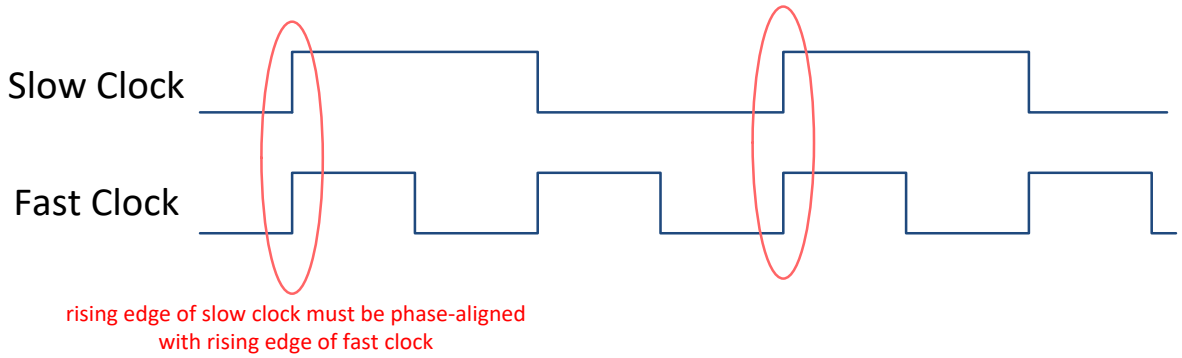

Figure 2 Synchronous clock crossers require alignment of rising edge.

To enable the communication, a clock enable control signal must be driven by the clock divider logic to identify when the fast and slow clocks share a rising edge.

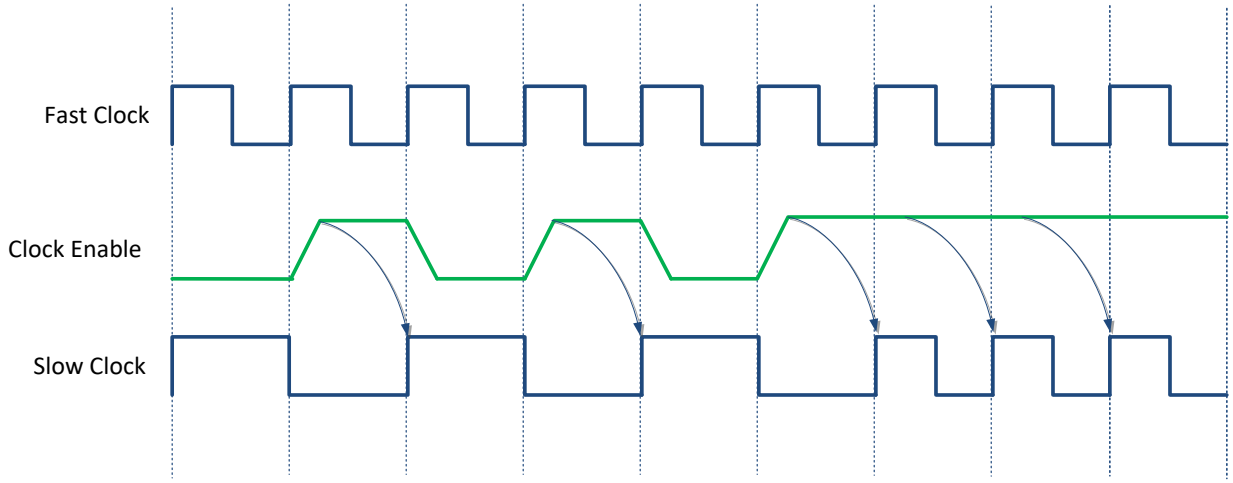

Figure 3 A clock enable input indicates when the rising edges are shared by the clocks.

The figure above shows a 1:2 clock ratio with a transition to 1:1 ratio. The clock enable signal should be driven for a full fast clock cycle, and should be driven high during the cycle before the shared rising edge. In the 1:1 case, the clock enable will stay asserted.

The interface with the clock crossing does not need to be quiesced when a ratio change occurs. The only requirement is that the clock enable is only asserted during the fast clock cycle before the shared rising edge.

2.1.4 Clock Gating

All NoC elements support activity-based coarse clock gating. Coarse clock gating can be enabled or disabled through NocStudio programming.

2.1.4.1 Clock Gating for NoCs without Regbus

In absence of the NoC regbus layer, after programming is done through NocStudio, there is no further option to control clock gating. Coarse clock gating is either always enabled or disabled based on NocStudio programming.

2.1.4.2 Clock Gating for NoCs with Regbus

For NoCs with the register bus and coarse clock gating enabled via NocStudio, control is provided to disable or enable clock gating at the granularity of each NoC element through register programming. There is one system_cg_or pin for each NoC element. The system_cg_or pin of a NoC element allows the coarse clock gating feature implemented in hardware to be overridden by software control. This is done by writing to a dedicated register RBSLVCG (details in noc_reference_manual.html and noc_registers.csv) residing in the Regbus Ring Master on that node. While changing the value on system_cg_or pin of a NoC element, it should be in an idle state and there should be no traffic flowing through it. Failure to observe this restriction will result in unpredictable or unrecoverable errors at system level. In addition, an important requirement is to wait the requisite number of cycles to allow the value written in RBSLVCG, to propagate to the target NoC element.

The Regbus Ring Master on each node contains 32 registers, each mapping to one of 32 slaves on that node. These register outputs will drive output pins from Regbus Ring Master and connect to system_cg_or pins of the appropriate NoC elements on that node.

The clock gating of NoC elements on the regbus layer is controlled through an external pin system_cg_or_regbus. The pin description is given in Table 6.

Table 7 Regbus layer clock gating signal

Signal name	Description
system_cg_or_regbus	Overrides coarse clock gating feature for Regbus Ring Master and Regbus Master Bridge (logic 1 implies coarse clock gating feature will not be used by hardware). This signal originates from a system-level clock controller (from external to NoC fabric) and connects to the system_cg_or pin of all Regbus Ring Master and Regbus Master Bridge.

2.1.5 Register Bus

The NoC has an optional, distributed register network allowing for general debugging, performance statistics gathering, and error recovery. The register bus (regbus) is built as an independent NoC layer (regbus layer) with a single access port. This port uses a modified AXI4-Lite protocol.

Please refer to the CFG Register Bus Protocol document for in-depth details on the register bus.

Some properties of the register bus master interface are listed below:

- 32-bit data width
- Supports AxLEN 0 or 1 for accessing 32-bit and 64-bit registers, respectively
- Up to 16 outstanding read/write requests can be issued to the NoC regbus layer

NoC registers are automatically created by NocStudio and placed in a fixed register bus address map. This address map is unrelated to any address map within the main NoC design.

For details of the registers and register address map, refer to noc_reference_manual.html and noc_registers.csv (which only appears if register bus is enabled) generated by NocStudio in the project directory.

NocStudio provides two configuration options through which read and write commands can be sent to the NoC regbus layer:

Default option: A master agent connects directly to the regbus layer port for reading or writing registers of the NoC. Usually this port is connected with a house keeping processor which is highly secure managing all system related functions.

Regbus Master Tunnel option: For design which doesn't have a separate housekeeping processor, the regbus can be enabled with the Regbus Master Tunnel mode. Any processor on the NoC layer can access the NoC registers through the tunnel.

2.1.6 Regbus Master Tunnel

Inside the NoC, the Regbus Master Tunnel is implemented as a host with two input slave ports and an output master port.

- By default, only one slave input port (port 0) is instantiated and is an AXI4 interface. This slave port is used to connect to a particular AXI4 Slave Bridge from the non-regbus NoC layers. Register reads and writes are issued on the non-regbus NoC layers and their corresponding addresses are mapped to the AXI4 Slave Bridge. The AXI4 Slave Bridge then sends those requests to the regbus layer via the Regbus Master Tunnel.

- The second slave input port (port 1) can be enabled through a NocStudio property and is connected directly to the regbus master agent.

```
host_prop rbm tunnel_slv1_enabled yes
```

- The output master port is connected to the Regbus Master Bridge on the NoC regbus layer. The Regbus Master Tunnel processes register read and write requests from the two slave ports and sends them to the regbus layer via the Regbus Master Bridge.

2.1.7 SRAM/Register File Instantiation

Many NoC designs contain rams or register files--either for NoC components which must be implemented as RAMs, or for NoC elements where the integrator can choose to either use a flopped implementation or to use a register file, as suits each instance best. Integrators must instantiate their own RAMs at the appropriate place in the ram wrapper files created in the noc_modifiable_rtl directory in the project directory.

For some designs, this is straightforward because there may be only one instance of a given ram in the design. In that case, the integrator should simply remove the sample instantiation and replace it with their own ram implementation. Each ram wrapper file in noc_modifiable_rtl contains a section like this:

```
`else
  // Please remove our ram model and instantiate your ram for ns_cmn_2p_rf here.
  wire cs;
  assign cs = (read_enable | write_enable);


ns_sram2p_cmn_rf u_ns_cmn_2p_rf_mdl (
                .CK(clk),
                .CS(cs),
                .WE(write_enable),
                .RE(read_enable),
                .RDPTR(read_ptr),
                .WRPTR(write_ptr),
                .DIN(data_in),
                .DOUT(data_out));
```

`endif

However, for other NoCs, this may be less straightforward, because they may contain, for example, multiple instances of a given NoC element, like CCC with its directory rams, or master bridges with reorder buffers. In that case, each might need a different size of the ram in question, requiring unique instantiations. In those cases, NocStudio will generate a comment statement like this inside each ram wrapper, immediately following the instantiation of the placeholder RAM module:

```
//If you have multiple unique ram instances for ns_cmn_2p_rf please
//uncomment the structure below and populate it with the correct
//ram instantiations.

  //generate
    //if (P_RROB_RAM_NAME == ns_h6_m_cmn_2p_rf) begin: G_NS_H6_M_CMN_2P_RF
    // Please instantiate your ram for ns_h6_m_cmn_2p_rf (width=261, depth=128) here
    //end
    //else if (P_RDATA_RF0_RAM_NAME == ns_iocb0_cmn_2p_rf_rd_0) begin: G_NS_IOCB0_CMN_2P_RF_RD_0
    // Please instantiate your ram for ns_iocb0_cmn_2p_rf_rd_0 (width=256, depth=128) here
    //end
    //else if (P_RDATA_RF1_RAM_NAME == ns_iocb0_cmn_2p_rf_rd_1) begin: G_NS_IOCB0_CMN_2P_RF_RD_1
    // Please instantiate your ram for ns_iocb0_cmn_2p_rf_rd_1 (width=256, depth=128) here
    //end
    //else if (P_RDATA_RF0_RAM_NAME == ns_llc0_cmn_2p_rf0_sprt0) begin: G_NS_LLC0_CMN_2P_RF0_SPRT0
    // Please instantiate your ram for ns_llc0_cmn_2p_rf0_sprt0 (width=256, depth=128) here
    //end
  //endgenerate
```

Implementors should uncomment the generate statement in the relevant ram wrapper files and modify it such that each RAM of that type in the design is properly instantiated for the instance name listed.

2.1.8 Interrupts

Every NoC router and bridge has an interrupt output signal. Interrupt is asserted when a fatal error or other interesting condition is encountered in a NoC element. The errors are also logged in interrupt status registers of the NoC element. If the mesh_prop *single_interrupt_for_noc* is set to "yes," interrupts from different NoC elements are logically ORed together (combinatorically) within the NoC and a single combined interrupt is brought out on the NoC external interface. This combined interrupt single should be treated as asynchronous relative to all clocks. This combined interrupt single should be treated as asynchronous relative to all clocks. If the mesh_prop *single_interrupt_for_noc* is set to "no," interrupts from different NoC elements are exposed directly on the NoC external interface.

If the register bus is instantiated in the NoC, it can be used to access interrupt control and status registers. Interrupt mask registers can be set to enable or disable some interrupts. When an interrupt occurs, a status register can be accessed to determine what the cause of the interrupt was. This status can be cleared in order to de-assert the interrupt signal. If an interrupt mask is modified to enable an event to trigger an interrupt, the status for that interrupt should be cleared by the user before changing the mask or the interrupt will trigger immediately.

If the register bus is not present in the NoC, the interrupt signals will still exist. Since there is no way to vary status or to change the interrupt mask, the mask will be set to only enable fatal error conditions. If the interrupt is ever triggered, there will be no way to de-assert the interrupt. This can still be useful to indicate a fatal error.

2.1.9 Event Counters

When the register bus is present in the NoC, it is possible to configure the NoC elements to count events for either performance measurement or for debug purposes. The routers and bridges each have a set of control registers and counters. The control registers allow the user to specify which event(s) they would like to have counted. As soon as the user programs the control registers, the counter(s) will begin to count the specified event(s). The counter register is readable and writeable. It can be read to see the current count and it can be written to in order to clear the count, or to initialize the count to a specific value. The counter will keep counting even when it hits its maximum value, which will cause it to overflow and start over at count zero.

The event counters can be set up to trigger an interrupt when the counter overflows. The overflow condition updates the interrupt status register. The interrupt mask can be used to enable or disable that interrupt. The user can trigger an interrupt after N number of events within the count window by initializing the counter to a value where incrementing N times will cause an overflow.

The registers controlling the event counters are independent, so intelligent use of the registers is required. Before switching from one set of counts to another, the user may want to program the control registers to not count any event. At that time, the user should clear the counter and the status bit in the interrupt register and program the interrupt mask. Once all of these registers are set correctly, the user can program the event control register to start counting a specified event.

2.1.10 Functional safety

ECC or parity-based error detection and correction of transport over a NoC can be enabled on a per-flow basis. Following guidelines must be followed when a NoC with these safety features is integrated in an environment for simulation.

External modules can inject X state into NoC data inputs for invalid data bits. A second source of X may come from non-reset flip-flops present in the internal data path on NoC elements. When ECC is computed by hardware over a data bus, any X can propagate to the end point and cause spurious X on error check interrupts. This may trigger verification X checkers on the interrupt pins. To overcome this, when building and running simulations with ECC/Parity enabled on data flows, variable NS_ECC_X_SQUASH must be defined. This causes X on any invalid data bit to be randomly changed to 1'b1 or 1'b0 before ECC is generated on the data bus.

```
+define+NS_ECC_X_SQUASH
```

Note that this is a special case of simulation only behavior implemented in RTL under the control of a *`define* variable. This variable should only be defined for simulations on ECC/parity enabled NoCs and must remain undefined in synthesis or emulation environments.

2.2 Integration of NoC Verification Checkers

For details on the CFG IP verification checkers, see Section **4**. Each checker binds to the corresponding RTL instance as shown in Figure 4 NoC checkers binding to RTL.

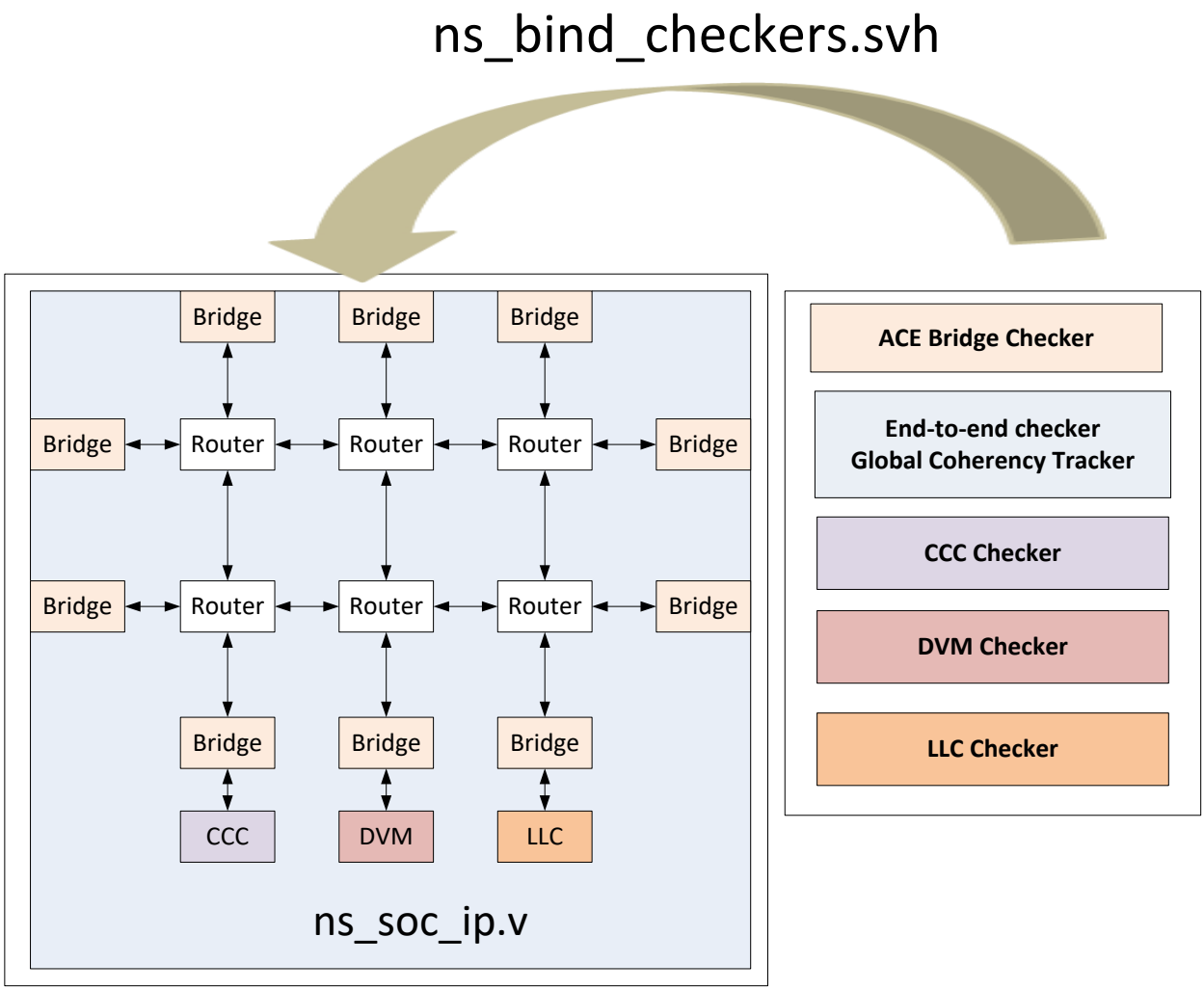

Figure 4 NoC checkers binding to RTL

To integrate NoC verification checkers into your environment:

- Add the following line to your testbench:

```
`include "ns_bind_checkers.svh"
```

  The ns_bind_checkers.svh file is located in the project directory created by NocStudio. It binds all checkers provided by NocStudio IP to their respective RTL instances regardless of testbench hierarchy. This file can be used without any changes.

- Set the environment variable $NS_PROJ_PATH to point to the directory created by NocStudio, for example:

```
setenv NS_PROJ_PATH /absolute/path/of/project/created/
```

- Adjust the `defines settings in noc_verif_cust/ns_global_defines.vh according to the recommended usage Section 3.2 **Environment Setup for Integration**. This file contains the `defines used by the verification checkers.

- Set the NS_NOC_TOP `define to the hierarchical path to the ns_soc_ip instance in your environment.

- Set the NS_END_OF_SIM `define to the hierarchical path to the end_of_sim signal from your testbench. It should be mapped to a 1-bit signal with single rising edge when the sim ends. This `define is used to trigger exit checks in checkers. Set the NS_END_OF_SIM `define to 1'b0 if you do not desire exit checks.

2.3 SUPPORTED TOOLS

Supported 64-bits versions of tools and languages:

- NoC RTL uses IEEE Std 1364™-2005 syntax and its support must be enabled in the tool flow.
- NoC simulation environment uses SystemVerilog IEEE Std 1800-2009
- Simulator: Cadence Incisiv 13.20.s036
- Simulator: Synopsys VCS-MX J-2014.12-SP3-3
- Synthesis: Cadence GENUS 15.20-p004_1

3 INTEGRATION OF MULTIPLE NOCS

To integrate RTL from multiple NoCs into the same design for simulation the following steps must be followed:

- The NocStudio configuration file for each NoC must have a unique project name and contain the command:

```
prop_default tag_project_name yes
```

  This allows each NoC to have unique top-level module names and log file names.
- Run NocStudio on each configuration file to generate project directories for multiple NoCs. For each configuration file, NocStudio generates the following:

Table 8 Directory and files with tag_project_name

Name	Description	Type
noc_verif_ip_proj/	Custom verification files specific to each NoC	Verification
ns_<project>_soc_ip.v	ns_soc_ip.v for each NoC	RTL
ns_<project>_fabric.v	ns_fabric.v for each NoC	RTL
ns_<project>_fabric_modules.v	ns_fabric_modules.v for each NoC	RTL
ns_<project>_group_modules.v	ns_group_modules.v for each NoC	RTL
ns_<project>_stream_noc_end_to_end_checker.sv	ns_stream_noc_end_to_end_checker.sv for each NoC	Verification
ns_<project>_stream_struct.sv	ns_stream_struct.sv for each NoC	Verification
ns_<project>_bind_checkers.svh	ns_bind_checkers.svh for each NoC	Verification

- Once multiple NoC project directories exist, the next step is to construct a combined NoC project directory. This can be done either manually, or using the sample script,

$NOCSTUDIO_HOME/scripts/ns_multi_noc.pl where $NOCSTUDIO_HOME is the installation directory.

3.1 AUTOMATED INTEGRATION OF MULTIPLE NOCS

- The ns_multi_noc.pl sample script constructs a combined NoC project directory, stitches a combined NoC testbench and performs a sanity check by compiling and elaborating the generated NoC RTL.
- To get a list of options, invoke the following command:

```
$NOCSTUDIO_HOME/scripts/ns_multi_noc.pl -h
```

- To run ns_multi_noc.pl, change to extracted tarball directory and invoke the following:

  If you are using Cadence Incisive Simulator, run:

```
$NOCSTUDIO_HOME/scripts/ns_multi_noc.pl \
-i <project1> <project2> ... <projectN> \
-o <combined_project>
```

  If you are using Synopsys VCS, run:

```
$NOCSTUDIO_HOME/scripts/ns_multi_noc.pl \
-i <project1> <project2> ... <projectN> \
-o <combined_project> \
-VCS
```

  Where,

  - <project1> <project2> ... <projectN> denote the paths to input NoC project directories.
  - <combined_project> denotes the desired name of the output combined NoC project directory.

On a successful compile and elaboration, the following will appear at the prompt:

```
***************
* BUILD PASSED *
***************
```

A file named SIM_SKIPPED will be generated in the combined NoC project directory indicating that only the build phase was performed. The log file ns_multi_noc.log will log the steps performed by the ns_multi_noc.pl script and any errors encountered. Depending on the simulator, the log file run_test_vcs.log or run_test_incisiv.log will also list any errors encountered during the build phase. The build logs, named build.log, are located in the model/model_*/ directory.

After a successful NoC sanity testbench build, the generated combined NoC RTL and verification IP are ready for integration into the user's environment.

3.2 MANUAL INTEGRATION OF MULTIPLE NOCS

- Create a combined NoC project directory and recursively copy the directories and files listed in table below from each NoC project directory, along with the following directories, into the combined project directory.

Table 9 Additional directories to copy into combined NoC project directory

Name	Description	Type
noc_modifiable_rtl/	RTL modules, such as RAM, that can be replaced by user implementation	RTL
noc_rtl/	NoC RTL library	RTL
noc_verif_cust/	Holds ns_global_defines.vh file used for integration of CFG verification IP into customer environment	Verification
noc_verif_ip/	NoC verification agents IP library	Verification

- Edit the ns_<project>_bind_checkers.svh files in the combined project directory. For multiple NoCs, there will be bind statements for the same module in more than one file. Resolve this by removing redundant bind statements for the same module.
- Create a combined ns_node_id_table.sv file in the combined project directory by concatenating the contents of ns_node_id_table.sv from each NoC.

- Create a combined ns_global_defines.vh by taking noc_verif_cust/ns_global_defines.vh from the first NoC project directory then adding in any `defines from noc_verif_cust/ns_global_defines.vh of subsequent NoCs that are not already present. Remove any `defines pointing to checker instances that do not have a bind statement in ns_<project>_bind_checkers.svh. For example,

```
`define NS_<PROJECT1>_NOC_TOP   <Hierarchical path to first NoC top module>
`define NS_<PROJECT1>_STR_E2E_CHECKER_TOP \
        `NS_<PROJECT1>_NOC_TOP.ns_<project1>_stream_noc_end_to_end_checker

`define NS_<PROJECT2>_NOC_TOP   <Hierarchical path to second NoC top module>
`define NS_<PROJECT2>_STR_E2E_CHECKER_TOP \
        `NS_<PROJECT2>_NOC_TOP.ns_<project2>_stream_noc_end_to_end_checker
```

  Where,

  - <project1> denotes the name of the project for the first NoC in lower case.
  - <PROJECT1> denotes name of the project for the first NoC in upper case.
  - <project2> denotes name of the project for the second NoC in lower case.
  - <PROJECT2> denotes name of the project for the second NoC in upper case.
- Consolidate ns_noc_files.f from each NoC to create final simulation file list for multiple NoCs. For example,

```
+libext+.v+.sv
+incdir+$COMBINED_PATH
+incdir+$COMBINED_PATH/noc_rtl
+incdir+$COMBINED_PATH/noc_verif_ip
+incdir+$COMBINED_PATH/noc_verif_cust
-y $COMBINED_PATH/noc_rtl
-y $COMBINED_PATH/noc_modifiable_rtl
-y $COMBINED_PATH/noc_verif_ip
-y $COMBINED_PATH/noc_verif_ip_proj
$COMBINED_PATH/noc_verif_ip/ns_regbus_struct.sv
$COMBINED_PATH/ns_<project1>_stream_struct.sv
$COMBINED_PATH/ns_<project1>_stream_noc_end_to_end_checker.sv
$COMBINED_PATH/ns_<project1>_fabric_modules.v
$COMBINED_PATH/ns_<project1>_fabric.v
$COMBINED_PATH/ns_<project1>_group_modules.v
$COMBINED_PATH/ns_<project1>_soc_ip.v
$COMBINED_PATH/ns_<project2>_stream_struct.sv
$COMBINED_PATH/ns_<project2>_stream_noc_end_to_end_checker.sv
$COMBINED_PATH/ns_<project2>_fabric_modules.v
$COMBINED_PATH/ns_<project2>_fabric.v
$COMBINED_PATH/ns_<project2>_group_modules.v
$COMBINED_PATH/ns_<project2>_soc_ip.v
```

Where,

- <project1> denotes the project name of the first NoC.
- <project2> denotes the project name of the second NoC.
- $COMBINED_PATH denotes the path to the combined NoC project directory.
- For each additional NoC, simply replicate the same lines of <project2> to point to the unique files of the new NoC.

This mechanism can be used to integrate more than two NoCs with no upper limit to the number of NoCs.

4 NoC Verification Components

The NoC IP package contains both unit-level and NoC-level checkers to ensure functional correctness of the NoC during simulation.

4.1 Overview of Checkers

If NocStudio is enabled with **verification components generation privileges**, it would provide the following checker files:

Table 10 CFG checkers

Checkers	Instantiated
NoC end-to-end checker	One instance per NoC
Streaming bridge checker	One instance per streaming bridge RTL
Regbus end-to-end checker	One instance per regbus layer

4.2 Environment Setup for Integration

In order to integrate in NoC verification components, a list of `define variables need to be defined.

Table 11 `define variables for model build

`define variable name	Description	Notes
`NS_NOC_TOP	Map to hierarchical path to NoC within user testbench	None.
`NS_END_OF_SIM	Map to a signal with one rising edge transition at end of simulation to trigger exit checks in the various NoC checkers	This signal goes high once and stays high for at least 2 cycles at the end of simulation, when all traffic in NoC is expected to have quiesced.
`NS_NOC_END2END_CHECKER_EN	Set to 1 to enable, 0 to disable NoC end-to-end checker	Recommend mapping to a plusarg variable to allow run-time control of value.

`NS_E2E_LOG	Set to 1 to enable, 0 to disable traffic logging by NoC end-to-end checker	Recommend mapping to a plusarg variable to allow run-time control of value.
`NS_STRBRDG_CHECKER_EN	Set to 1 to enable, 0 to disable NoC streaming bridge checker	Recommend mapping to a plusarg variable to allow run-time control of value.
`NS_REGBUS_END2END_CHECKER_EN	Set to 1 to enable, 0 to disable register bus end-to-end checker	Recommend mapping to a plusarg variable to allow run-time control of value
`NS_REGBUS_E2E_LOG	Set to 1 to enable, 0 to disable traffic logging by register bus end-to-end checker	Recommend mapping to a plusarg variable to allow run-time control of value.
`NS_CSR _CHECKER_EN	Set to 1 to enable, 0 to disable CSR checker	Recommend mapping to a plusarg variable to allow run-time control of value

4.3 USAGE MODES

The following table describes a set of recommended usage modes for enabling the NoC checkers. The tradeoff is made between debug visibility and simulation performance penalty for increased visibility.

Table 12 Recommended checker settings

Usage mode	Bring up Mode	Heavy Debug Mode	Code Stable Mode
`NS_NOC_END2END_CHECKER_EN	1	1	1
`NS_E2E_LOG	1	1	0
`NS_REGBUS_END2END_CHECKER_EN	1	1	1

`NS_REGBUS_E2E_LOG	1	1	0
`NS_STRBRDG_CHECKER_EN	1	1	0
`NS_REGBUS_CHECKER_EN	1	1	0
`NS_REGBUS_E2E_LOG	1	1	0
`NS_CSR_CHECKER_EN	1	1	0

4.4 CHECKERS

4.4.1 Terminology

The types of checks that are performed are divided into the following categories:

Protocol – These checks enforce adherence to AMBA or NoC interface protocol.

Unsupported – These checks flag violations of AMBA interface protocol features that are currently not supported.

Functional – These checks verify the functionality of the NoC RTL.

Exit – These checks are performed at the end of simulation to verify that the NoC is in a proper idle state at the end of simulation.

4.4.2 Streaming Bridge Checkers

The streaming bridge checkers are responsible for monitoring streaming bridge RTL during simulation. Each instance of streaming bridge RTL has a corresponding streaming bridge checker monitoring its behavior. The streaming bridge checkers enforce adherence to interface protocol on all bridge interfaces. In addition, they also perform micro-architectural checks to ensure functional correctness of the streaming bridge RTL. At end of simulation, when there should be no traffic in the NoC, these checkers perform exit checks to ensure each instance of streaming bridge RTL is in a proper idle state. The following is a list of the checks performed by the streaming bridge checker. Violation of any one of these will trigger an error in simulation.

Table 13 Streaming bridge checks

Description of check	Instantiated (per bridge or interface)	Type of check
Each transaction begins with one and only one SOP	Host port TX interface	Protocol
Each transaction ends with one and only one EOP	Host port TX interface	Protocol

QoS field must be constant during transmission of the transaction	Host port TX interface	Protocol
Destination hostport id field must be constant during transmission of the transaction	Host port TX interface	Protocol
Destination interface id must be constant during transmission of the transaction	Host port TX interface	Protocol
Credit overflow	Host port TX interface	Protocol
Credit underflow	Host port TX interface	Protocol
No unknown (x or z) control signals when not in reset	Host port TX interface	Protocol
No unknown (x or z) data packets	Host port TX interface	Functional
Payload size is no more than maximum size(64KB)	Host port TX interface	Protocol
Protocol violation of double EOP will raise interrupt	Host port TX interface	Functional
Protocol violation of packet transmission without SOP will raise interrupt	Host port TX interface	Functional
Packet with illegal destination	Host port TX interface	Functional
Packet with illegal destination will raise interrupt	Host port TX interface	Functional
Each transaction begins with one and only one SOP	Host port RX interface	Protocol
Each transaction ends with one and only one EOP	Host port RX interface	Protocol
Credit overflow	Host port RX interface	Protocol
Credit underflow	Host port RX interface	Protocol
Payload size is no more than maximum size	Host port RX interface	Protocol
No unknown (x or z) control signals when not in reset	Host port RX interface	Protocol
No unknown (x or z) data packets	Host port RX interface	Functional
Internal micro-architectural checks	Streaming bridge	Functional
Should not have any interrupt	Streaming bridge	Functional
No packets in flight	Host port RX interface	Exit
All credits have restored to initial value	Host port RX interface	Exit
No packets in flight	Host port TX interface	Exit

All credits have restored to initial value	Host port TX interface	Exit
All bridge FIFOs in RTL are empty	Streaming bridge	Exit

For a subset of the above checkers, fine-grained user-control is provided to individually enable or disable the checkers. For each check listed in the following table, setting the corresponding `define to 0 enables the check; setting it to 1 disables the check. They should be set to the default value in all cases except for error testing that may require these to be set otherwise.

Table 14 Fine-grained user-control of streaming bridge checkers

Description of check	`define to control	Default value
Packet with illegal destination	`NS_STRBRDG_INVALID_DEST_CHECK_DISABLE	0
Should not have any interrupt	`NS_STRBRDG_INTERRUPT_CHECK_DISABLE	0
No unknown (x or z) data packets	`NS_STRBRDG_DATA_XZ_CHECK_DISABLE	0
Set to 0 to inhibit some checks while injecting errors for SAFETY feature validation.	`NS_SAFETY_CHECKALL_EN	1

4.4.3 Regbus End-to-End Checker

The Regbus End-to-End Checker is a scoreboard that tracks register traffic on the register bus layer to ensure every register access is properly routed to the correct destination register ring, and every response is propagated back to the master in the correct order with the correct data content. The following checks are performed.

Table 15 Register bus end-to-end checks

Description of check	Instantiated (per bridge or interface)	Type of check
ARVALID, AWVALID, WVALID, RREADY are never X or Z.	Regbus Master Bridge	Protocol

ARLEN and AWLEN are either 0 or 1.	Regbus Master Bridge	Protocol
Every AR request that enters the Regbus Master Bridge is tracked with its corresponding R response for the roundtrip check to ensure correct propagation of command fields, propagation of content and ordering within the regbus layer.	NoC	Functional
Every AR request that enters the Regbus Master Bridge arrives at the correct destination Regbus Ring Master, with the correct ARADDR, ARPROT, ARLEN, node_id, ring_id, seqnum, in the correct order.	NoC	Functional
Every read request packet that enters the Regbus Ring Master is tracked with its corresponding response packet for the roundtrip check to ensure correct propagation of command fields, propagation of content and ordering within each Regbus Ring Master.	Regbus Ring Master	Functional
Every R response that leaves the Regbus Ring Master arrives at the Regbus Master Bridge with the correct RDATA, RRESP, RLAST, in the correct order.	NoC	Functional
Every pair of AW and W requests that enters the Regbus Master Bridge is tracked together with the corresponding B response for roundtrip check to ensure correct propagation of command fields, propagation of data and ordering within the regbus layer.	NoC	Functional
Every AW and W request that enters the Regbus Master Bridge arrives at the correct destination Regbus Ring Master with the correct AWADDR, AWPROT, AWLEN, WDATA, WSTRB and WLAST, node_id, ring_id, seqnum, in the correct order.	NoC	Functional
Every write request packet that enters the Regbus Ring Master is tracked with its corresponding response packet for the roundtrip check to ensure correct propagation of command fields, propagation of content and ordering within each Regbus Ring Master.	Regbus Ring Master	Functional

Every B response that leaves a Regbus Ring Master arrives at the Regbus Master Bridge with the correct BRESP, in the correct order.	NoC	Functional
There is one and only one SOP per Regbus Ring Master packet.	Regbus Ring Master	Protocol
There is one and only one EOP per Regbus Ring Master packet.	Regbus Ring Master	Protocol
Valid is high only when a Regbus Ring Master packet is in-flight.	Regbus Ring Master	Protocol
No regbus request or response is in-flight.	NoC	Exit
All regbus requests and responses during the simulation are accounted for.	NoC	Exit

The Regbus End-to-End Checker has the capability of generating a set of traffic log files during the simulation to provide visibility of the traffic on the Regbus Master Bridge and on each Regbus Ring Master connected to the Regbus Master Bridge. The following table lists the settings required to enable the traffic logs.

Table 16 Settings to enable register bus end-to-end traffic logs

`define to control	Value
`NS_REGBUS_END2END_CHECKER_EN	1
`NS_REGBUS_E2E_CHECKER_LOG	1

The file names of the logs are of the following format with <node_id> corresponding to the node id of the Regbus Master Bridge and each Regbus Ring Master assigned by NocStudio,

Table 17 Register Bus End-to-End Checker log files

File name	Description
ns_regbus_mbrdg_<node_id>.log	Regbus Master Bridge traffic log for AW, W, B, AR and R channels.
ns_regbus_ring_master_<node_id>.log	Regbus Ring Master traffic log for request (Regbus

	Master Bridge to Regbus Ring Master) and response (Regbus Ring Master to Regbus Master Bridge) channels.

As shown in the above table, two types of log files are created for every NoC with regbus layer. The ns_regbus_mbrdg<node_id>.log displays transactions received on the modified AXI4-Lite interface of the Regbus Master Bridge. Each ns_regbus_ring_master_<node_id>.log displays transactions received on the Regbus Ring Master interface with the Regbus Master Bridge.

4.4.4 Regbus Ring Slave Checker

The Regbus Ring Slave Checker is responsible for monitoring register bus ring slave RTL during simulation. Each instance of register bus ring slave RTL has a corresponding register bus ring slave checker monitoring its behavior. The following checks are performed.

Table 18 Regbus ring slave checks

Description of check	Instantiated (per bridge or interface)	Type of check
regslv_req_valid is low when in reset.	Regbus Ring Slave	Protocol
regslv_rsp_valid is low when in reset.	Regbus Ring Slave	Protocol
regslv_req_valid is not X or Z when out of reset.	Regbus Ring Slave	Protocol
regslv_rsp_valid is not X or Z when out of reset.	Regbus Ring Slave	Protocol
regslv_req_valid must not transition low until regslv_req_ready is asserted.	Regbus Ring Slave	Protocol
regslv_rsp_valid must not transition low until regslv_rsp_ready is asserted.	Regbus Ring Slave	Protocol
When regslv_req_valid is high, request bus control signals must not be X or Z.	Regbus Ring Slave	Protocol
While regslv_req_valid is high, request bus signals must remain constant until after regslv_req_ready is asserted.	Regbus Ring Slave	Protocol

While regslv_rsp_valid is high, response bus signals must remain constant until after regslv_rsp_ready is asserted.	Regbus Ring Slave	Protocol

4.4.5 Clock Control Signal Checks

Table 19 Clock control signal checks

Description of check	Instantiated (per bridge or interface)	Type of check
scan_mode pin can only toggle when streaming bridge is idle	Streaming bridge	Functional
system_cg_or pin can only toggle when streaming bridge is idle	Streaming bridge	Functional
system_clk_en pin can only toggle when streaming bridge is idle	Streaming bridge	Functional

4.4.6 NoC End-to-End Checker

NoC end-to-end checker is used to verify the correct operation of the NoC design. All packets transmitted into the NoC are tracked to construct a global reference database of expected results at the output interfaces. The packets transmitted out of each NoC interface are then compared against the global reference database to ensure packets arrive with correct content and in the correct order. Unexpected packets, packets sent to an incorrect destination, lost packets, extra packets, corrupt packets or packets received out of order would all be detected and flagged as errors. At end of simulation, when there should be no traffic in-flight in the NoC, the end-to-end checker ensures that the NoC is in a proper idle state.

Table 20 NoC end-to-end checks

Description of check	Instantiated	Type of check
Each packet is routed to the correct port with the correct content and size	Per NoC	Functional
Ordering of traffic is maintained for the same source, source interface, destination, destination interface and QoS value	Per NoC	Functional

Every packet is accounted for, no extra packet or missing packet at destinations	Per NoC	Functional
No traffic is in-flight within the NoC	Per NoC	Exit
All verification FIFOs and queues are empty	Per NoC	Exit
All verification checkers indicate no error	Per NoC	Exit

4.4.7 NoC End-to-End Traffic Logs

The NoC end-to-end checker has the capability of generating a set of end-to-end traffic log files as each packet successfully exits the NoC and passes all the checks in the end-to-end checker. The following table has the setting to enable the end-to-end logs.

Table 21 Settings to enable end-to-end traffic logs

`define to control	Value
`NS_NOC_END2END_CHECKER_EN	1
`NS_E2E_LOG	1

The file names of the logs are of the format,

ns_noc_packets_to_<*destination_hostport*>_<*destination_interface*>.log

One log file is produced for each destination NoC interface. Each log file records the end-to-end information for every packet as it's received on the destination NoC interface, in the following format:

<sim_time> : pkt_sent_sop_time=<pkt_sent_sop_time>,
pkt_sent_eop_time=<pkt_sent_eop_time>,
pkt_received_sop_time=<pkt_received_sop_time>,
pkt_received_eop_time=<pkt_received_eop_time>,
pkt_length=<pkt_length>, src_id=<src_id>, src_intf=<src_intf>,
qos=<qos>, dst_id=<dst_id>, dst_intf=<dst_intf>,
pkt_dataQ=<first_data_segment>...<last_data_segment>,
match_unique=<match_unique>, match_cnt=<match_cnt>

Table 22 Nomenclature of end-to-end traffic logs

Field name	Description
<sim_time>	Simulation time when the packet passed all end-to-end checker checks at destination NoC interface
<pkt_sent_sop_time>	Simulation time of when the packet's SOP was sent into the NoC
<pkt_sent_eop_time>	Simulation time of when the packet's EOP was sent into the NoC
<pkt_received_sop_time>	Simulation time of when the packet's SOP exited the NoC
<pkt_received_eop_time>	Simulation time of when the packet's EOP exited the NoC
<src_id>	Source hostport id of the streaming bridge from which the packet was sent
<src_intf>	Source interface id of the streaming bridge from which the packet was sent
<qos>	QoS value of the packet
<dst_id>	Destination hostport id of the streaming bridge to which the packet was sent
<dst_intf>	Destination interface id of the streaming to which the packet was sent
<first_data_segment>	Data content at the beginning of the packet

<last_data_segment>	Data content the end of the packet
<match_unique>	When the packet was received, whether the packet was a unique match in the reference database in end-to-end checker or whether there were multiple matches of identical packets. If this is 1, then the packet can be uniquely identified and has satisfied the end-to-end check. If this is 0, then this packet is not a unique packet but has satisfied one of the possible legal outcomes without violating ordering requirements
<match_cnt>	If <match_unique> is 0, this field indicates the total number of expected packets that are identical which are possible matches for the packet at the time of arrival at the destination NoC interface

Example traffic flow from a NoC end-to-end log file:

```
4135: pkt_sent_sop_time=3775, pkt_sent_eop_time=3795,
pkt_received_sop_time=4125, pkt_received_eop_time=4135,
pkt_length=105 bits, src_id=0, src_intf=0, qos=3, dst_id=1,
dst_intf=1, pkt_dataQ=0x705fa05fa...705fa05fa, match_unique=1,
match_cnt=1
```

4.4.8 CSR Checker

The CSR Checker monitors the configuration status register module and ensures registers are never X or Z out of reset. Each instance of configuration status register RTL has a corresponding CSR checker monitoring its behavior. The following checks are performed.

Table 23 CSR checks

Description of check	Instantiated (per bridge or interface)	Type of check

No unknown (x or z) on streaming bridge registers	Host port TX interface	Functional
No unknown (x or z) on streaming bridge registers	Host port RX interface	Functional

5 PHYSICAL DESIGN GUIDELINES

NocStudio, when enabled, generates RTL for a given configuration. Generated RTL can then be synthesized using the reference Synthesis Environment. Figure 5 shows the different stages of the flow from Verilog RTL generation to Synthesis.

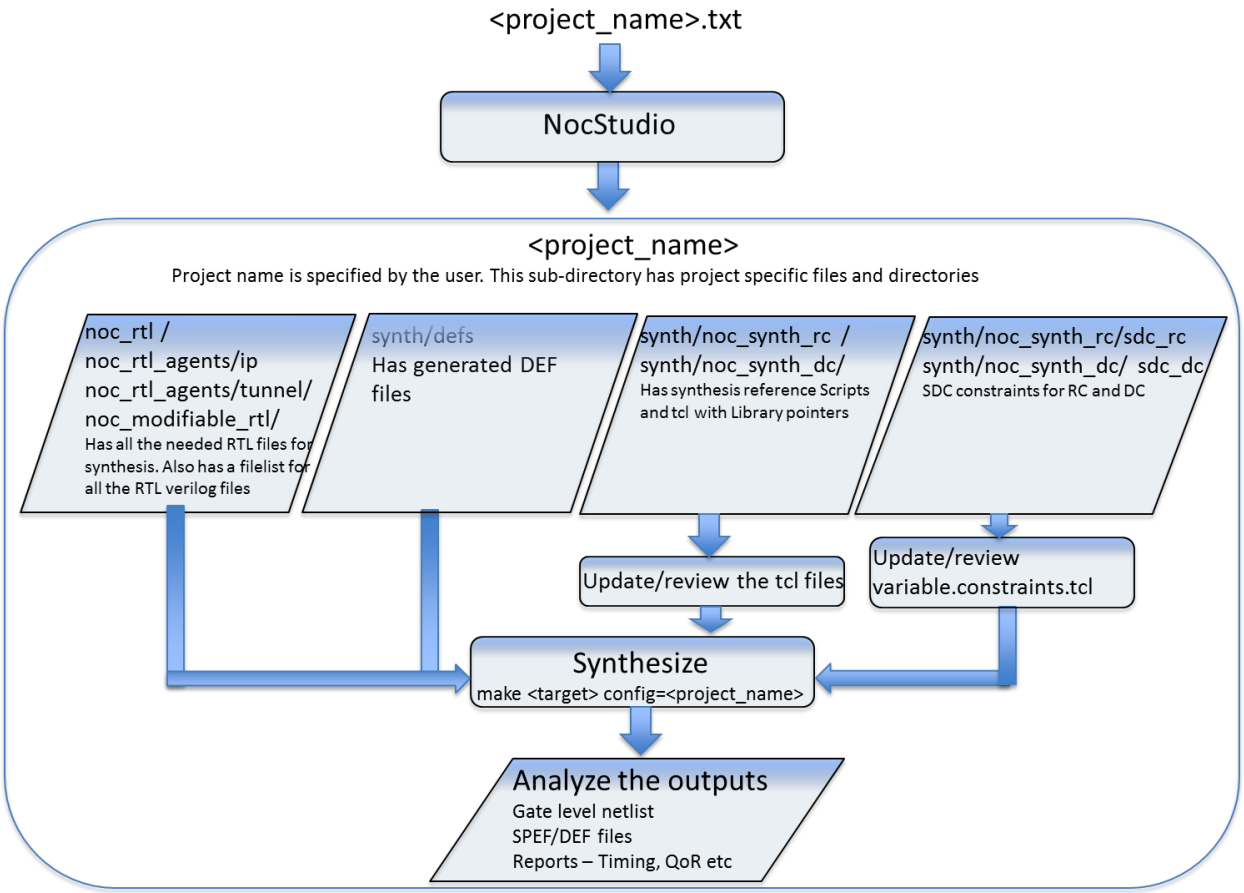

Figure 5: An example flow chart explaining steps from NocStudio to Synthesis output

A particular NoC configuration is defined in "<project_name>.txt". This file is the input to NocStudio. NocStudio then generates a <project_name> subdirectory.

5.1 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 the 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.

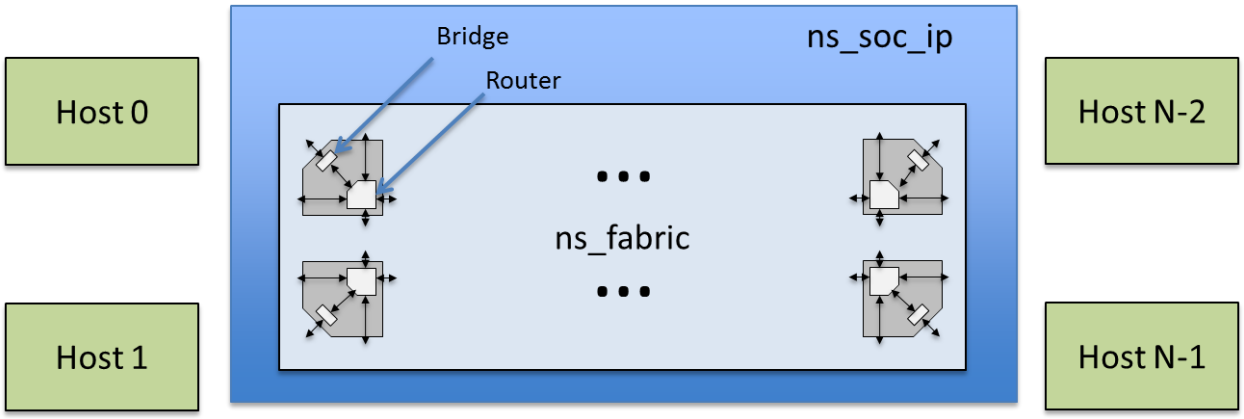

Figure 6: NoC RTL Netlist Structure

Each host port interfaces to the NoC through a bridge. The widths of the host port interface bus and the NoC link are configurable in NocStudio. NoC routers contain 8 ports: 4 for the interconnect, named North, East, South, West, and 4 for hosts, H (shown in the figure as South East), I (shown in the figure as South West), J (shown in the figure as North West), and K (shown in the figure as North East).

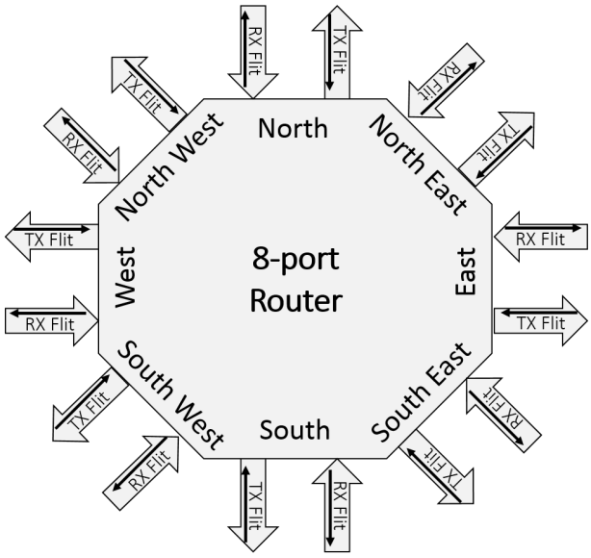

Figure 7: 8 Port Router with 4 directional ports & 4 host ports

Router and bridge elements are created such that the NoC top-level file ns_soc_ip.v contains no parameters and no logic elements.

5.1.1 Multiple NoC Router Layers

To support additional bandwidth and virtual channels, CFG'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 exist in the design 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.

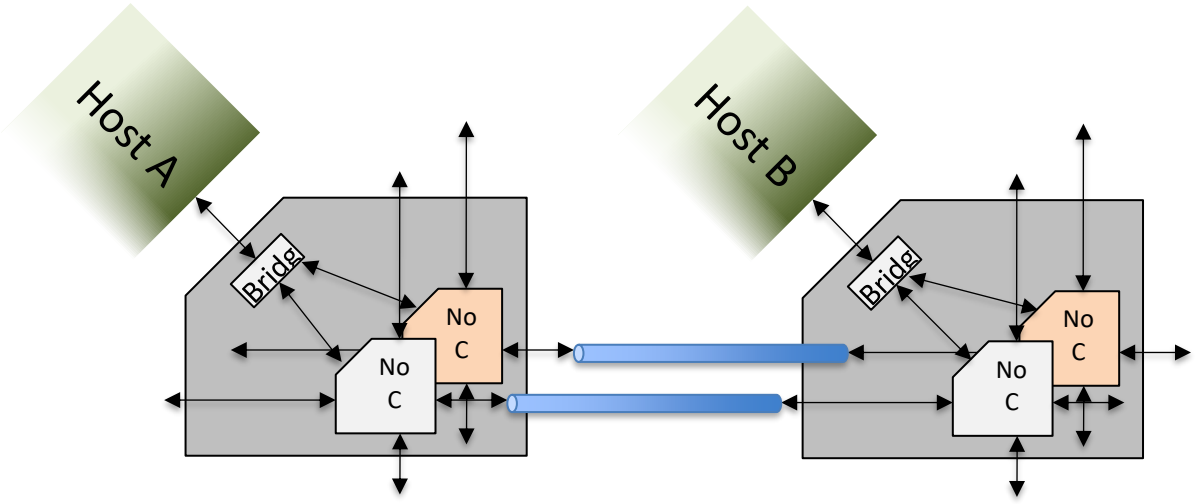

Figure 8: Dual NoC layers

5.1.2 Naming convention

NocStudio uses a naming convention that concatenates the bridge type, the component function, and appends interconnect information to each instance. Examples of bridge types are axi, ace, ocp, etc. Sample NoC component names are:

1. Router – ns_router_<layer_number>_<node_number>
2. Master bridges – ns_<bridge type>mstrbrdg_<host_name>_<port_name>
3. Slave bridges – ns_<bridge type>slvbrdg_<host_name>_<port_name>

5.1.3 Example

Following is the example for example_synth.txt config in “examples” directory. This example is for a 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.

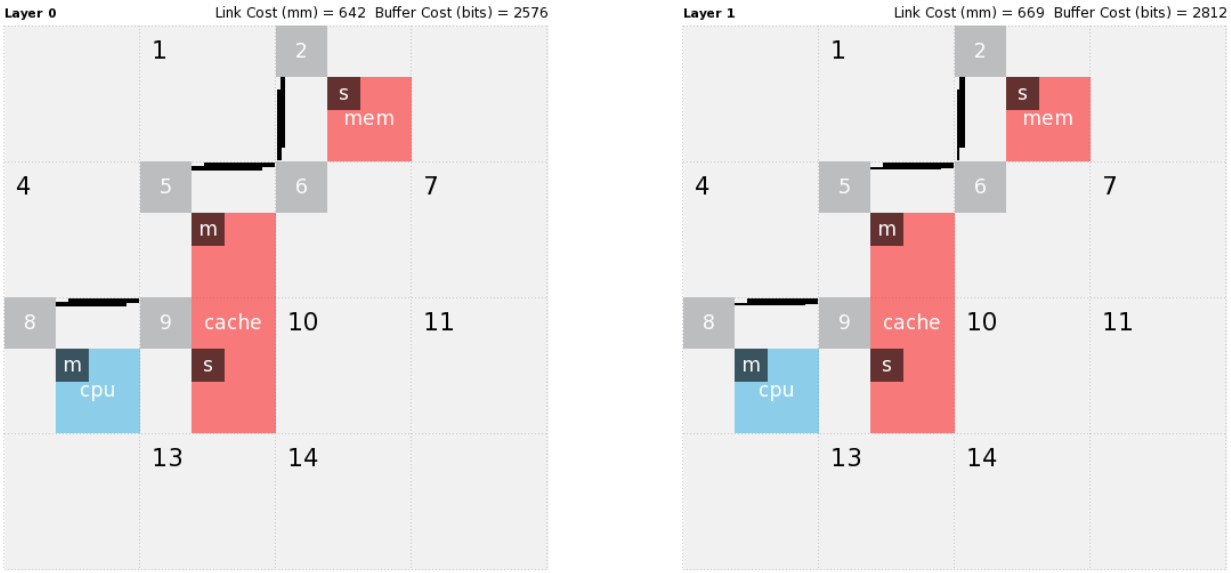

Figure 9: NocStudio GUI snapshot

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”

5.2 SYNTHESIS

NocStudio, when enabled, generates RTL and reference synthesis files for the given noc configuration. Generated RTL can be synthesized using this 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 IEEE Std 1364™-2005 format
    - noc_rtl.vc // Manifest file containing path to library
    - noc_rtl.gemini.vc // Manifest file containing path to Gemini library
  - **noc_rtl_agents**
    - ip/*.v, ip/*.h // NoC Agents library RTL in IEEE Std 1364™-2005 // format
    - tunnel/*.v, tunnel/*.h // NoC Agents library RTL in IEEE Std 1364™-2005 // format
    - noc_rtl_agents.vc // Manifest file containing path to NoC agent library
    - noc_rtl_agents.gemini.vc // Manifest file containing path to Gemini 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)
- library_domain.tcl // Tcl file used define Voltage domain(s) used being
  // used for 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

5.2.1 Synthesis Methodology

The reference synthesis environment supports top-down and hierarchical synthesis flows. In a top-down synthesis flow, the NoC is synthesized as a single block. In a hierarchical synthesis flow, the submodules are synthesized separately, and then are stitched together at the top level. Synthesis can be done with wire load models or with a more physically aware flow.

Any synthesis tool can synthesize the NoC and its components. We provide reference synthesis scripts for Cadence and Synopsys synthesis tools.

The RC reference synthesis scripts are intended for Cadence RTL Compiler (RC) or the newer Genus. Both these tools use wire load models to estimate loads. Alternatively, for a more physically aware synthesis, the RCP scripts can be used with Cadence RTL Compiler Physical (RCP) or Genus in physical mode. These tools use the LEF and CAPTABLE or the QRC technology file provided by the foundry for parasitic extraction.

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

5.2.2 Setting Up Synthesis

NocStudio generates a reference sdc file. It includes timing constraints on inputs and outputs, as well as setting max delay values for asynchronous areas of the design—for example, on the grey-coded pointer values in the asynchronous fifos.

Designers should carefully review their designs, especially signals that cross clock domain boundaries, and modify the NocStudio-generated timing constraints as needed to ensure timing correctness for their systems. Specifically, designers should address these parts of the generated sdc file:

- The NocStudio generated sdc file treats *reset_n* inputs as ideal nets. Depending on a customer's reset methodology, designers may choose to change this constraint.
- Top-level point-to-point (non-IO) signals that travel longer distances exist in the NOC. All these signals are sourced from a flop, and end in a synchronizer circuit (ns_demet). The regular expression *async* will find them. (Note that these signals may not be truly asynchronous but are treated as such due to the distance they may traverse.) Designers could choose to treat these as multicycle paths, or depending on customer methodology, even as false paths. **Only top-level *async* nets should be treated as multicycle or false paths. This should not be done recursively**.
- **Designers must ensure that the NocStudio-generated set_max_delay statements appear at the end of the sdc files**, to ensure that they are not overwritten by any pattern matching on the regular expressions described above. NocStudio-generated set_max_delay statements exist to ensure proper timing in gray-coded pointers crossing clock domain boundaries in async fifos. More detail on these paths and a list of specific signal names and circuits is below in section 2.3.

Finally, designers should take into account the ultimate physical floorplan of the noc. For example, for a particular cell/node of the mesh, if the bridges and routers will be placed physically close/adjacent to each other, then the synthesis PPA results (power, performance, and area) will be optimal if they are kept in the same hierarchy and synthesized together.

5.2.3 Running Synthesis

The following tools are needed to run the reference synthesis flow

5.2.4 For Cadence flow:

- Cadence RTL Compiler (RC) or Genus – For doing synthesis with wire load models
- Cadence RTL Physical Compiler (RCP) or Genus – 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
  - Pointer for QRC file – QRC files can be used instead of CAPTABLEs
- 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 for non-LP NoCs
- Do "make hier config=<project_name>" to do hierarchical synthesis for non-LP NoCs
- Do "make lptop config=<project_name>" to do top level synthesis for LP/Multi-voltage NoCs
  Do "make lphier config=<project_name>" to do hierarchical synthesis for LP/Multi-voltage NoCs

5.2.5 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.

5.2.6 Analyzing outputs

5.2.6.1 Analyzing RC/RCP/Genus 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.

5.2.6.2 Analyzing DC/DCT synthesis output

Top level and hierarchical synthesis creates the following files and directories

- <module>_outputs – This directory stores generated outputs.
- <module>_reports - This directory stores generated reports.

5.3 DEF

When enabled NocStudio generates top level def file(s) for a NoC. Two files are generated by NocStudio

- ns_soc_ip.def – This file contains “UNITS DISTANCE” in microns, “REGION” and “GROUP” definition
- ns_soc_ip_afp.def – This file contains X-Y co-ordinates, “DIEAREA”, “PINS” definition

5.4 CDC

This section details the IP support for the Spyglass CDC flow. It covers the files included with our IP to support this flow, how to run the flow, and the outputs generated. It also includes some known warnings that are waivable.

5.4.1 Files Included to Support Spyglass CDC

When NocStudio runs on an NCF file, it generates a project directory for the config file. Inside that config directory, the following files are created by NocStudio:

- ./cdc/spyglass_cdc.tcl: Tcl script used as input to sg_shell
- ./cdc/cdc_waiver.swl: Waiver file for warnings, not errors
- ./synth/noc_synth_dc/sdc_dc/spyglass_constraints.sgdc: NocStudio generated config-specific constraints to help Spyglass run correctly on the IP.

In addition, when the spyglass_cdc.tcl script runs, it creates a constraints file, ./synth/noc_synth_cd/sdc_dc/sg_cdc.constraints.tcl based on the variables.constraints.tcl and ns_soc_ip.constraints.tcl files found in the same directory.

All these files are necessary input to the Spgylass tool.

5.4.2 How to Run Spyglass

To run Spyglass CDC:

- cd to the NocStudio-generated project directory.
- Type "sg_shell –tcl ./cdc/spyglass_cdc.tcl"

This tcl script runs the following spyglass goals:

- cdc/cdc_setup_check: checks input constraints.
- cdc/cdc_verify_struct: actual CDC verification
- Ac_abstract01: generates abstract view of IP, for use in SOC-level spyglass runs, when the goal is to blackbox our IP and run at the sytem-level.

5.4.3 Spyglass Flow Outputs

The spyglass flow generates the following sets of outputs:

- spyglass/consolidated_reports/ns_soc_ip_Design_Read: a report on the design read portion of the cdc flow.
- spyglass/consolidated_reports/ns_soc_ip_cdc_cdc_setup_check: a report on the setup and constraints checking portion of the CDC run.
- spyglass/consolidated_reports/ns_soc_ip_cdc_cdc_verify_struct: a report on the CDC results.
- spyglass/ns_soc_ip/cdc/cdc_verify_struct/spyglass_reports/abstract_view/ ns_soc_ip_cdc_abstract.sgdc: the Spyglass-generated abstract view of the Noc IP, to be used by SOC-level Spyglass runs which want to blackbox our Noc IP.

5.4.4 Known CDC Warnings

There should not be any errors generated by a Spyglass CDC run on our IP. In the past we have seen and fixed Ac_unsync0*/Ac_glitch* errors in our rtl. Also, Spyglass itself sometimes has an

issue recognizing a qualifier, and we've been working through those as they come up with Spyglass AEs and getting Spyglass bug fixes from them.

There may be some Ac_conv0*, or Ac_coherency06 warnings generated. These are waivable. Specifically:

- In our low-power logic, warnings that multiple asynchronous signals converge on a signal in a new clockdomain, for instance power domain signals like *QACTIVE*, *autowake*, *sleep_req* and *sleep_ack*.
- Warnings about fanout to multiple clock domains for *sleep_req_n* signals
- Warnings about fanout to multiple clock domains from flops like *u_ns_nsps_agg_8phase_nsps_proxy_async_fence_ack_n*.u_ns_nsps_agg_8phase.G_SLV_ASYNC.mst_demet.demet_stage_q*
- Warnings about signals being synchronized multiple times into the same clock domain. This is common on master and slave bridges, because each channel (R/W) synchronizes certain system-level signals and uses them internally. Also, each inblock and outblock of router will synchronize and use internally certain router-level signals for local consumption only

6 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. NocStudio supports the "show_noc_density" command to extract number of wires horizontally and vertically. It outputs to the NocStudio console windows so it's part of the transcript.log. In addition, the command extracts the local wires (among bridges and routers at a grid) as well as local flops. The information can be used to predict potential congestion point and SOC architect may move certain bridge port to the next grid to avoid such a congestion.

6.1 P&R KEEPING THE NOC ELEMENTS TOGETHER

The bridge and router at single mesh grid point may be combined into 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.

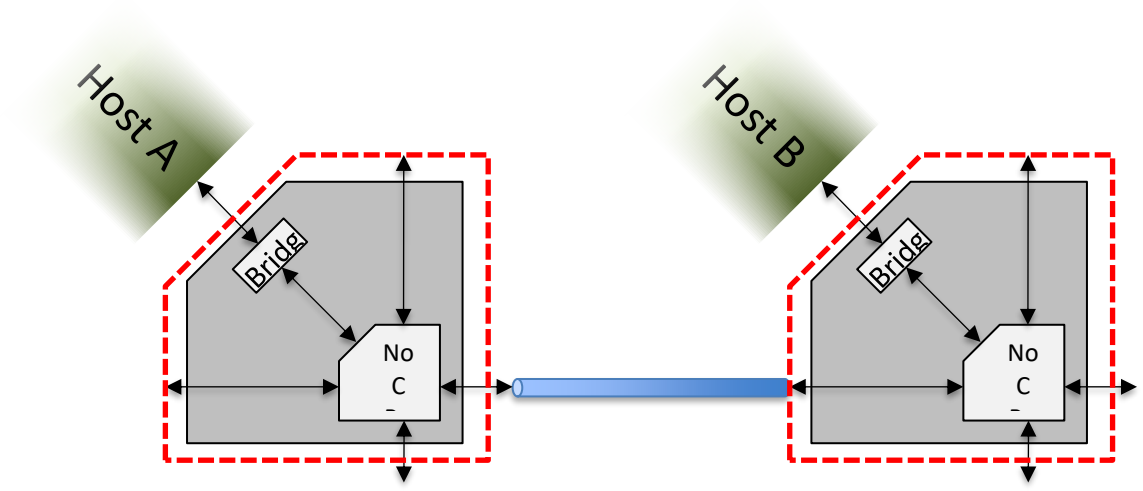

Figure 10: NoC Elements in stand-alone P&R regions

6.2 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 11. This may reduce the number of top-level regions and/or macros and the number of top-level signals.

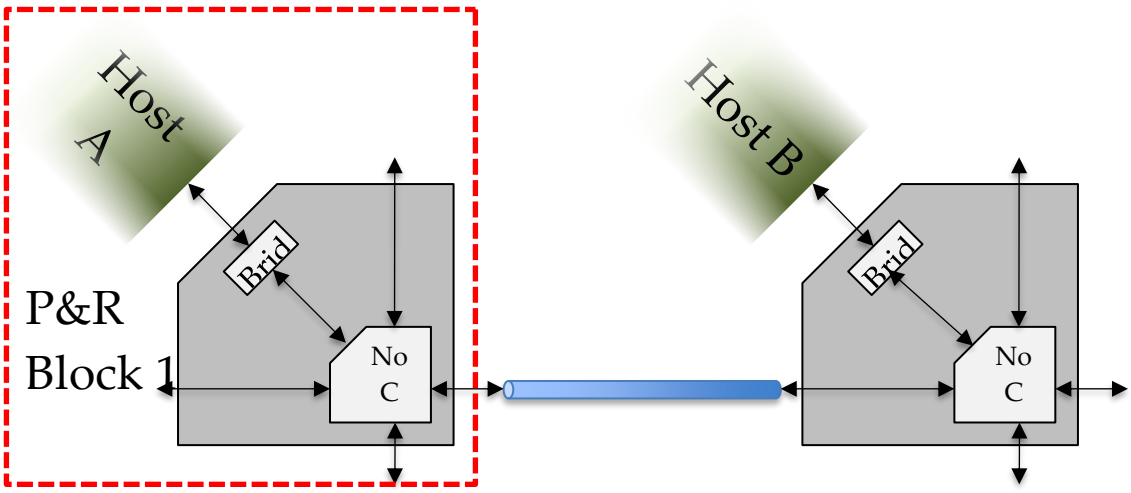

Figure 11: Merge all NoC Elements with Host P&R Region

6.3 NoC Quick Start Information

6.3.1 NoC clock domains

NoC elements may share a common clock, use multiple clocks that have a phase relationship (ratio synchronous), or use multiple clocks that are completely asynchronous (there is no phase relationship between the various clocks). If clocks are asynchronous, the NoC instantiates asynchronous FIFOs between the clock domains. An example for a clock domain boundary between a bridge and an axi agent is shown in figure.

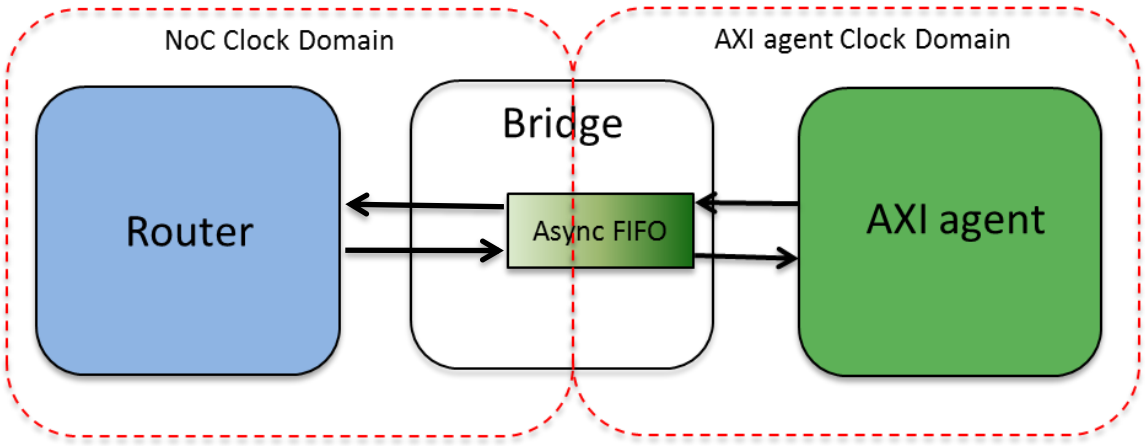

Figure 12: Bridge and AXI agent with async clock

In asynchronous designs, for timing closure, designers must constrain the skew between the bits in the gray-coded async FIFO read and write pointers, to ensure that correct values are received into the destination clock domain. To be conservative, constrain the skew between the bits of the gray-coded pointers is to be less than one period of the faster of the two clocks.

The skew must be constrained in both directions, as the gray-coded rd_pntr goes to the write domain and the gray-coded wr_pntr goes to the read domain inside the asynchronous fifo. The constraint should be of the form:

set_max_delay -from <source FFs> <value less than the faster of the two clock periods>

Regular expressions for the source FFs are:

- *rd_pntr_gray_q* (src clock is rd_clk, dest clock is wr_clk)
- *wr_pntr_gray_q* (src clock is wr_clk, dest clock is rd_clk)

In addition, for link clock crossing fifos (ILDCs), for leaf level synthesis, the read clock domain (via the read address) accesses the flop arrays (written on wr_clk) directly. This is alright because the time required to synchronize the pointers across the clock boundary guarantees the stability of the write data. For these fifos, the skew between the rd_addrs bits should be constrained to be less than one period of the write clock. For example:

set_max_delay -from *rd_addrs_one_hot* <time value less than 1 period of source clock>

The skew between the data bits can be constrained to be less than one cycle of the destination clock (rd clk).

set_max_delay –from *ns_async_reg_array_q* <time value less than 1 period of dest clock>

These ILDC constraints are required only for leaf-level synthesis, because the two sides of the fifo will be synthesized individually. They are not necessary for timing closure

Figure 13 is an example where Host A has the same clock as NoC clock and Host B clocks with an independent clock domain.

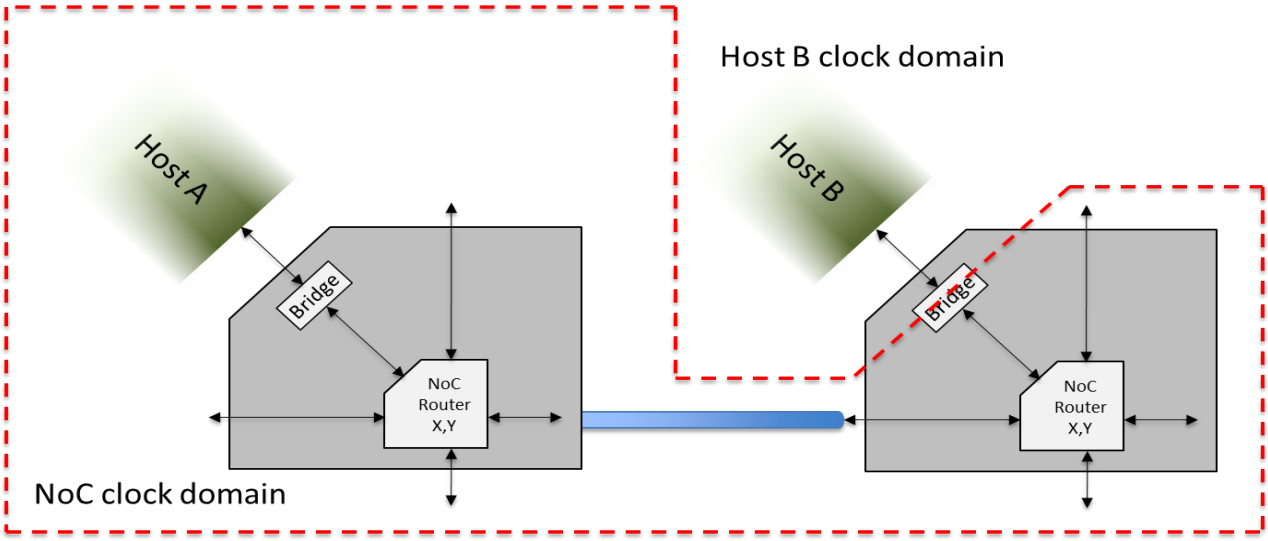

Figure 13: Host A and NoC clock domain different than Host B clock

Figure 14 is another implementation where Host A and Host B clock domains are different from NoC clock domain.

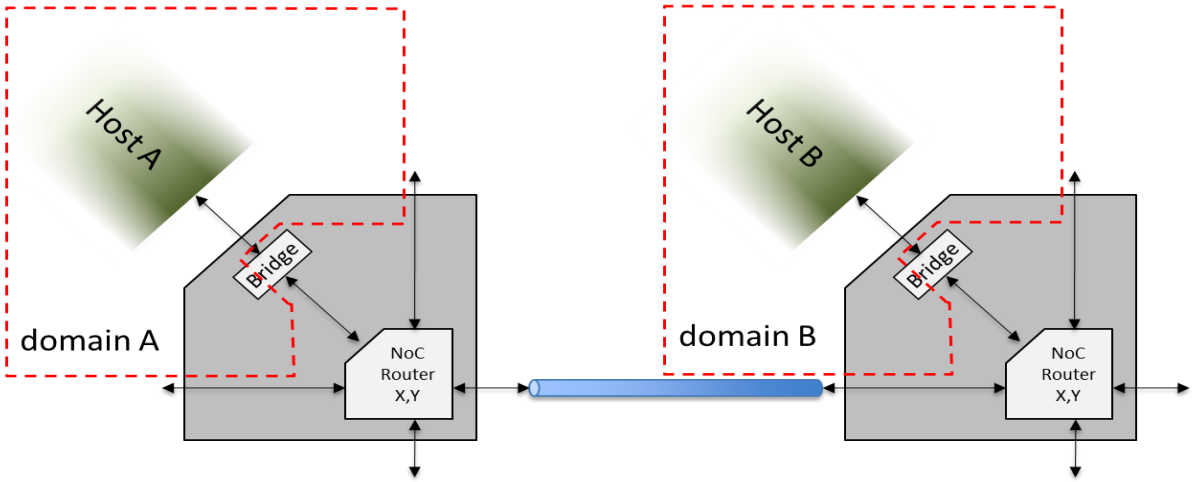

Figure 14: Host A & B with different clock domains than NoC clock

6.3.2 Clock Gating Cell

CFG 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 is in the noc_modifiable_rtl directory of the tarball. Before synthesis, the user should replace the clock-gating cell with one 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. We recommend using higher metal layers (thicker) for these signals when possible. Non-minimum spacing and shielding will reduce crosstalk.

6.3.3 NoC Clock tree

NoC implementations can span an entire chip. This raises the concern of the overall clock skew budget. To address this, split the overall chip clock skew into two skew groups—a global clock skew group and a local clock skew group. The NoC is architecturally independent of global clock skew. It is extremely important to minimize local clock skew.

6.3.4 Reset

The NoC elements are reset using an asynchronous assertion, synchronous de-assertion reset scheme. Our reference reset synchronizer is in the file ns_rst_n.v in the noc_modifiable_rtl

directory. This module is provided as a reference only; users can and should replace it with a module or library cell that matches their chip reset methodology.

For ns_rst_n, note that the number of stages present is programmable by NocStudio, on a per-clock-domain basis. Figure 15 shows the reset logic inside ns_rst_n, using the NocStudio default value of two flop stages per synchronizer. The output of the reset synchronizer cell is either the synchronized reset if tst_rst_bypass is low, or tst_rst, if tst_rst_bypass is high.

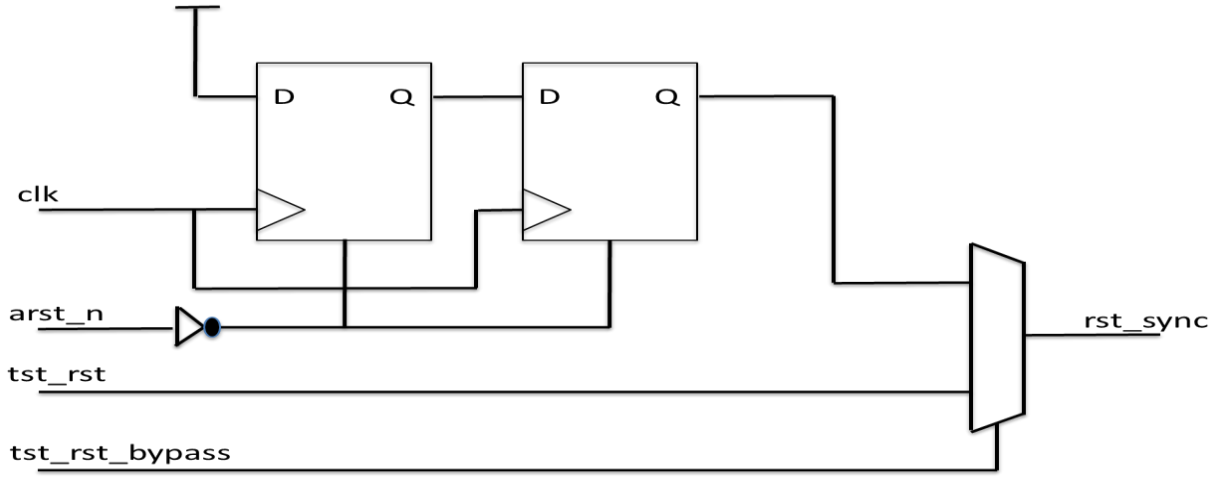

Figure 15: Reset logic

Top-level reset timing requirements are:

- Reset must be 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.

Also, designers should not allow synthesis or physical design flows to place buffers between the flops in the synchronizer chain, as it increases the failure rate of the synchronizer.

6.3.5 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 15), basic two and three 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. The resulting noc design is fully scan compliant and easy to constrain using an SDC file.

6.3.6 Channel Routing

The busses between noc routers can potentially be very wide, depending on the design. As with most full chip routing we recommend using higher metal layers (thicker) for longer routes of these wide parallel interconnects. Also, to reduce crosstalk-related delays and errors, we recommend using non-minimum wire spacing.

6.3.7 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
- Use non-minimal 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, NocStudio parameters must be updated in order to ensure design correctness.

6.3.8 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. Frequently 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.

6.3.9 Synchronizers

There is one standard module used for all synchronization in RTL generated by NocStudio: ns_demet.v. It uses fixed names (*demet_stage*) for its DFFs, making it easy to identify them in physical design and verification flows. The depth of the flop chain inside ns_demet.v is programmable, on a per-clock-domain basis. The synchronizer module ns_demet.v is in the

noc_modifiable rtl directory. Users should replace it with a synchronizer module or library cell that meets their own chip's synchronization methodology.

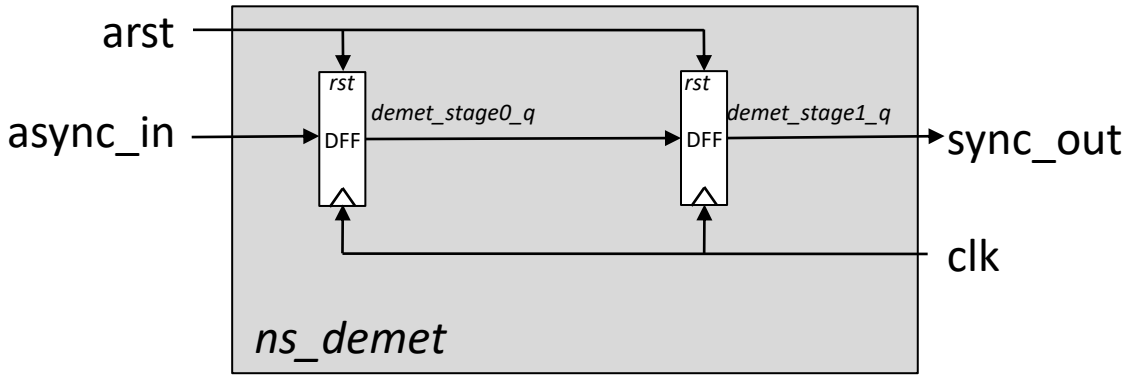

Figure 16: CFG synchronizer module

As mentioned above in 2.3.1, all data asynchronous clock domain crossings occur in async FIFOs, where the grey-coded control signals pass through synchronizers with a programmable number of flop stages. Physical design flows should avoid placing buffers between the flip-flops in the synchronization chain, as this increases the failure rate exponentially. This is especially important for high frequency designs.

All synchronizer flip-flops may be located using the following regular expression: "*demet_stage*".

6.3.10 Timing for Voltage Domain Crossers (VDC) and In-Link Domain Crossers (ILDC)

Below is the block diagram of VDC/ILDC:

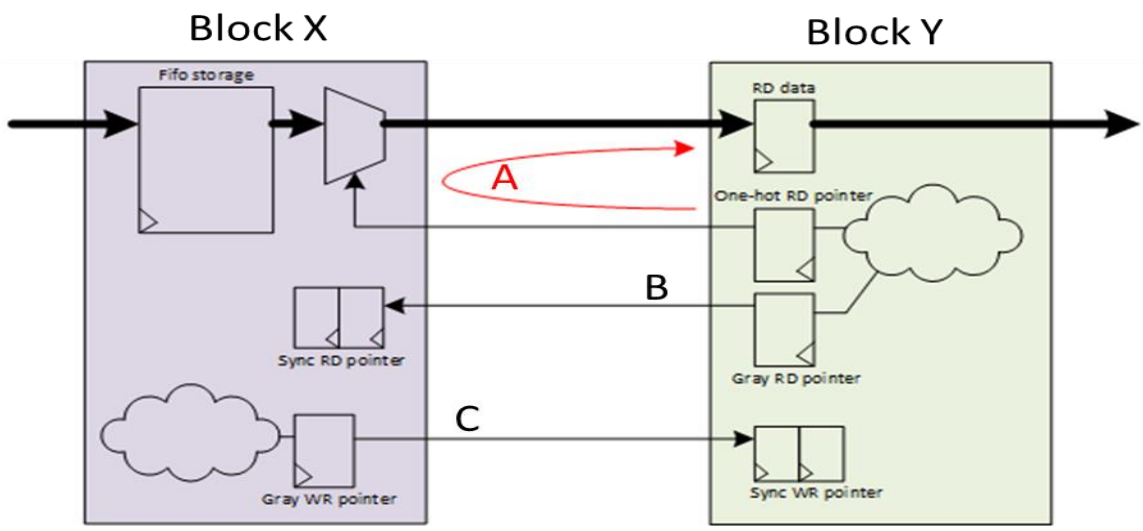

Figure 17: VDC and ILDC block diagram

There are three types of timing arcs between Block X and Block Y

- Timing arc A: The output of a flop in Block Y drives logic in Block X, whose output is flopped by Block Y. The NocStudio SDC file for Block X uses a set_max_delay to constrain timing on this path. Design-appropriate constraints of "set_max_delay" and "set_min_delay" should be applied during synthesis and static timing analysis, using the examples in our reference SDC files. For ILDC, Block X is "ns_link_clk_cross_fifo_wr*" and Block Y is "ns_link_clk_cross_fifo_rd*". The regular expression for A's input to Block X is: "*rd_addrs_one_hot*". The regular expression for A's output from Block X is: "*rd_data*". For VDC, both Blocks X and Y blocks are instantiated in "ns_vdc_slv*". There are 3 pairs of signals—3 separate A arcs—in the VDC fifo. The A arc is from every bit of the input to every bit of the output for each of the 3 paths. They are:
    - input [P_FIFO_DEPTH-1:0] vdc_mst_AR_ch_rd_addrs, output [P_AR_CH_INFO_WIDTH-1:0] vdc_slv_AR_ch_rd_data.
    - input [P_FIFO_DEPTH-1:0] vdc_mst_AW_ch_rd_addrs, output [P_AW_CH_INFO_WIDTH-1:0] vdc_slv_AW_ch_rd_data.
    - input [P_FIFO_DEPTH-1:0] vdc_mst_W_ch_rd_addrs, output [P_W_CH_INFO_WIDTH-1:0] vdc_slv_W_ch_rd_data ;
- Timing arc B: The NocStudio SDC file has output constraints of Block Y and input constraints of Block X. The regular expression is "*rd_pntr_g*".
- Timing arc C: The NocStudio SDC file has output constraints of Block X and input constraints of Block Y. The regular expression is "*wr_pntr_g*".

For both ILDC and VDC, the read and write sides should be placed close to each other to avoid timing violations. These two blocks may be instantiated in the top level (ns_fabric level) or in rtl groups.

7 DFT

Testability is a design attribute that measures how easy it is to create a program that comprehensively tests a manufactured design's quality. Traditionally, design and test processes were separate, with test considered only at the end of the design cycle. However, 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 NoC elements are DFT compliant and yield high coverage, ensuring higher test quality.

8 C++ NoC Model Integration

The C++ NoC model consists of a statically linked library containing all functions of the model, and a header file containing the APIs that are used to run the model. The C++ model is named nocstudio.a for Linux and nocstudio.lib for Windows. The C++ header files are named nocstudio.h.

8.1 How to Simulate Model

A number of APIs are defined in the nocstudio.h file. These APIs are used to simulate the model. During simulation, flits are injected into the model and ejected from the model at various bridge interfaces.

Flit Construction for Injection

A base NocFlit header class will be provided in the nocstudio.h file. Its attributes are as follows:

```
enum FlitPos { SopEop, Sop, Eop, Middle };

struct NocFlit_base {
   brif_t src;              // Source bridge interface id
   int qos;                 // QoS
   FlitPos pos;             // Position of flit in packet
   void* payload;           // Pointer to payload
};

struct NocFlit : NocFlit_base {
   brif_t dest;                          // Dest bridge interface id
};
struct MCNocFlit : NocFlit_base {
   std::vector<brif_t> dest;  // Vector of dest bridge interface id
};
```

A flit object of type NocFlit or MCNocFlit must be constructed for injection into the NoC, depending on whether the Flit is Unicast or Multicast. The NocFlit is unicast and has one destination, while the MCNocFlit supports multicast transmissions through the NoC, having a vector of destinations. Flit injection process must adhere to the protocol requirements of the interface. The FlitPos field determines the position of the flit in the packet i.e. Start of Packet (Sop), End of Packet (Eop), Middle of packet (Neither Sop nor Eop) or single flit packet (SopEop).

The source and destination of the NocFlit/MCNocFlit are represented as brif_t (typedef int), encoding the bridge id and the interface id. The brif_t (bridge_interface ID) of a source or destination is derived as (bridge_id << 2) | interface_id.

Streaming bridge interfaces a, b, c, d are mapped to integers 0, 1, 2, 3, respectively.

The payload field is a generic (void) pointer that is disregarded by the internals of the model and returned unchanged upon flit ejection.

8.1.1 Flit Injection

The following API is used for flit injection.

```
bool inject_flit(Sim* s, const NocFlit& flit,
        string* error_string = NULL);
bool inject_flit(Sim* s, const MCNocFlit& flit,
        string* error_string = NULL);
```

For an injection to be successful, the flit provided must match a valid traffic hop specified in the imported script file with which the model was configured. Flit injections must adhere to following protocol:

1. SOP/EOP logic
    a. A series of flits must be started with an SOP flit
    b. An EOP flit must come at some point after an SOP to complete the series
    c. The first injection from any interface must always be an SOP flit
2. Flits for different traffic flows cannot be interleaved when injecting into the same interface
3. Only one flit may be injected per interface per cycle

During each cycle, a flit may be injected at every input interface of all bridges of the NoC by calling the appropriate inject_flit() API depending on whether the injected flit is unicast or multicast. The injection API returns true if flit injection is successful, and false if injection fails. Injection may fail because of protocol errors, invalid message, or invalid fields in the flow. It may also fail if there is flow control asserted at the injection interface. In this case, the same flit may be injected again in the next cycle.

Table 24 Injection error cause and string values

Cause	Error message
Invalid src_h	Source bridge ID <bridge id> not valid.
Invalid dest_h	Destination bridge ID <bridge id> not valid.
Invalid src_i	Source interface ID <interface id> must be in range [0-3].
Invalid dest_i	Destination interface ID <interface id> must be in range [0-3].
Invalid src_br	Cannot get a valid bridge from source bridge ID <bridge id>.
Invalid dest_br	Cannot get a valid bridge from destination bridge ID <bridge id>.
Invalid src_ifce	Cannot get a valid transmitting interface from Bridge ID <bridge id> and Interface ID <interface id>.
Invalid dest_ifce	Cannot get a valid receiving interface from Bridge ID <bridge id> and Interface ID <interface id>.
Invalid src_br (regbus)	Source bridge is Regbus Master. Currently not supported.
Invalid dest_br (regbus)	Destination bridge is Regbus Master. Currently not supported.
Invalid message	Cannot create a valid packet. No traffic flows match flit attributes.
Invalid message	No packet is being sent; need SOP to start a new packet.
Invalid message	Current packet is still transmitting; need EOP to complete.
Invalid message	Flits are being interleaved. QoS and destination of flits from same packet must match.
Invalid message	Flit offset mismatch.
Flow control	Can inject only one flit per interface every cycle.
Flow control	Cannot inject; rate limit exceeded.
Flow control	Cannot inject; fifo full.
Flow control	Flow Control.
Other reasons	Failed to inject flit into the NoC.

The following API is used for credit return by the NoC.

```
bool set_credit_return_callback(Sim* s,
        const brif_t src,
        function<void(const brif_t src)> cb,
        string* error_string = NULL);
```

set_credit_return_callback() sets the callback function to be called upon the transmitting bridge returning a credit to the host via the given interface. The NocFlit should be destroyed inside this callback function.

Table 25 Credit return callback setting error cause and string values

Cause	Error message
Invalid src bridge id	Source bridge ID <bridge id> not valid.
Invalid src ifce id	Source interface ID <bridge id> must be in range [0-3].
Invalid src_br	Tx bridge for Bridge ID <bridge id> not found.
Invalid src_ifce	Tx interface for Bridge ID <bridge id> and Interface ID <interface id> not found.
Undefined callback	Callback must be defined.

8.1.2 Flit Ejection

The following event-based API is used for flit ejection. Note: The original polling-based eject_flit() and can_eject_flit() APIs have been removed.

```
bool set_eject_callback(Sim* s, brif_t dest,
    function<void(const NocFlit&)> cb,
        string* error_string = NULL);
bool send_credit_rxif(Sim* s, brif_t dest,
        string* error_string = NULL);
```

set_eject_flit_callback() sets the callback function to be called upon the receiving bridge receiving a flit at the given interface. When it is called, the ownership of the NocFlit returns to the user code. The callback function is responsible for calling send_credit_rxif() to indicate to the receiving bridge that it will be able to send one more packet to the host without overflowing the host. There is no MCNocFlit version of eject callbacks because only NocFlit is ejected.

The following properties hold:

1. Flits injected from a given source to a given destination arrive (are ejected) in the order that they were injected
2. As long as the payload pointers of all injected flits of a packet are consistent with each other (i.e. they all point to the same payload), the payload pointers of all ejected flits of the packet will be consistent with each other

Table 26 Flit ejection callback setting and credit issuing error cause and string values

Cause	Error message
Invalid dbridge	Destination bridge ID <bridge id> not valid.
Invalid difce	Destination interface ID <interface id> must be in range [0-3].
Invalid dest_br	Rx bridge for Bridge ID <bridge id> not found.
Invalid dest_ifce	Rx interface for Bridge ID <bridge id> and Interface ID <interface id> not found.
Undefined callback	Callback must be defined. (Only for set_eject_flit_callback())

8.1.3 Advance Clock

The following API is used to advance the internal clock of the NoC model.

```
void advance_time(Sim* s);
```

advance_time() advances the simulation time by one clock cycle, consuming all inputs of all blocks and producing outputs during that clock. The simulation is clocked at the highest clock frequency of all the clocks domains in the NoC.

8.2 How to Include Model

To use the NoC model, users must include the .h header file and link the static library in the project. Then the model must be configured with the same command script file that was used to generate the NoC IP in NocStudio. Once model is configured, the model can be simulated by using the flit injection and flit ejection APIs defined in the header files. The Flit data structure is also defined in the same library. The model is clock cycle aware; therefore, the clock must be advanced in the model cycle by cycle by calling the advance_time() API.

An example of how to include various model files and use the NoC model APIs is provided in APITest.cpp in the NocStudio release package.

8.3 How to Configure Model

To configure the model, a command script file must be created. This file must contain the same set of commands that were used to generate the NoC IP. Alternatively, the command log file generated in NocStudio can be used.

The command script must be scrubbed, as the NoC model does not allow standard NocStudio simulator commands such as run, cont, etc. Therefore, all such commands should be removed from the command script. If not removed, these commands will be ignored while configuring the NoC model. The command that generates NoC IP, gen_rtl, is ignored, as the model library is disabled from generating RTL and verification files. Currently, all commands that write to disk are disabled, including sim_stats, gen_image, create_trace_files, etc., and are ignored during model configuration.

Once the script file is ready, the following API is used to configure the model.

```
Sim* create_sim(istream& commands, string* error_string = NULL);
```

The function takes the input stream of the commands from the script file as an argument and returns a NULL pointer in case of error(s) in script processing, mapping of the NoC, or Sim creation, and writes a description of the error in error_string if the pointer is not NULL.

Table 27 NoC model configuration error causes and string values

Cause	Error message
Invalid commands in script	Error: Failed processing commands from input
Mesh cannot be created	Error: Cannot extract a NocStudio grid from Console; new_mesh not called?
Traffic is not mapped	Error: Cannot run sim unless traffic is mapped.

8.4 Simulation Performance

The NoC model features adjustable knobs that enable users to make tradeoffs between simulation speed and statistics data collection and error checking.

8.4.1 Performance statistics logging

The following API is used to set the performance statistics logging level.

```
void set_perf_logging(api_mode_stats_collection_level_t stat_level);
```

set_perf_logging() adjusts the level of logging performed by the simulator for purposes of performance statistics generation. Lower values consume fewer resources per simulation cycle, increasing simulation speed.

Table 28 Performance statistics logging levels

Level	Logging performed
LvlNone (default)	No logging
LvlLow	Includes interface and bridge level logging
LvlMedium	Includes channel buffer level logging w/ blocked causes
LvlHigh	Includes detailed logging of tx/rx times on each packet

8.4.2 Simulation safety checks

The following API is used to set the simulation safety check level.

```
void set_sim_safety_check_level(Sim* s,
        api_mode_sim_safety_check_level_t level);
```

set_sim_safety_check_level() adjusts the extent of safety checks performed by simulation routines. Levels are per Sim object. Lower values consume fewer resources per flit injection, increasing simulation speed.

Table 29 Simulation safety check levels

Level	Checks performed
ChkNone (default)	No checking
ChkMedium	Includes flit injection flow control checks
ChkHigh	Includes protocol checks on injected flits: • Invalid source or destination bridge/interface ID • Source or destination bridge is a regbus master (currently not supported) • No traffic flows match flit attributes • Mismatch of QoS and/or destination between two flits of same packet

	• More than one flit injected per cycle • SOP of next packet sent before EOP of current packet • Flit begins a new packet, but is not SOP

8.5 UTILITIES

The NoC model also includes utilities for modifying the NoC and querying information.

8.5.1 Run commands

The following API is used to run arbitrary commands on the NoC, as if they were typed into the NocStudio console.

```
bool run_commands(Sim* s,
        const vector<string>& cmds,
        ostream& cmd_out = cout,
        string* error_string = NULL);
```

run_commands() takes a vector of strings and runs them sequentially on the NoC of the given simulator. It also accepts an optional cmd_out parameter containing a reference to an output stream, to enable users to redirect the output of the commands to a file or a parser, for instance to inspect error messages to determine why a command failed. If no output stream is given, output is written to standard output cout.

It is intended for each set of commands to either:

- Not modify NoC construction, such as merely changing the FIFO depth of a link
- End with a map command after modifying the construction of the NoC by, for instance, adding a host, changing the skip_enable property of a router, or adding traffic

Consequently, it returns true only if all commands succeeded *and* the NoC remains in a 'Mapped' state after executing them.

Table 30 Run command error cause and string values

Cause	Error message
One or more commands failed	Command(s) failed; see standard output / 'cmd_out' stream for details.
Given set of commands modified NoC	NoC is no longer in 'Mapped' state after running commands.

construction without re-running map	

8.5.2 Query statistics

The following API is used to query the occupancy, throughput or latency of a channel.

```
struct EventStats {
        double minimum;      // Minimum value of the stat gathered
        double maximum;      // Maximum value of the stat gathered
        double average;      // Average value of the stat gathered
        double count;        // Number of samples of the event stat
        void merge(const EventStats& e); // To merge two stats
};


EventStats query_end_to_end_latency(Sim* s,
        brif_t src, brif_t dest,
        string* error_string = NULL);

EventStats query_stats(Sim* s,
        const string& channel_name,
        api_mode_stats_t type,
        string* error_string = NULL);
```

query_end_to_end_latency() is used to get the end to end latency of the messages from a source to a destination. If either or both of the src and dest are -1, then the function returns the event stats associated with all the src/dest.

query_stats()is used to get the occupancy, throughput or latency of a specified channel. The type of statistics to collect is controllable via the third argument to the function. The allowed types are Occupancy, Throughput and Latency.

Table 31 Statistics query error cause and string values

Cause	Error message
Invalid channel name	Invalid channel name <channel name> specified.
Channel name contains a wildcard	Specified channel name must match exactly 1 channel. Wildcards are not supported.

8.5.3 Reset stats

The following API is used to reset the stats collected on a channel(s).

```
bool reset_stats(Sim* s,
                 string& channel_name,
                 api_mode_stats_t type,
                 string* error_string = NULL);
```

reset_stats() allows users to reset the logging structures associated with the Occupancy, Throughput or Latency of channel(s). The channels can be VCs of routers or Interfaces of bridges and can include wildcards. The function returns true if the resetting of stats was successful.

8.5.4 Performance simulator verbose logging to file

The following API is used to enable or disable the writing of detailed performance simulator messages to an output stream.

```
bool set_perf_sim_log_file(Sim* s,
                 bool en,
                 ostream* os,
                 string* error_string = NULL);
```

set_perf_sim_log_file() allows users to write the verbose output displayed in NocStudio console output upon issuing a step command to a file or an output stream instead. It accepts a Boolean value of whether to enable or disable logging, and a mandatory pointer to an output stream to which the messages will be written (if enabling).

The function returns false if enabling logging, but the given output stream is null or not in a "good" state.

8.6 SUMMARY OF NOC MODEL APIS

The NoC model uses the following commands, summarized below:

1. **create_sim**
    a. *input:* input stream of the script file
    b. *output:* pointer to model simulator
    c. *about:* creates the sim pointer from a script
2. **advance_time**
    a. *input:* simulator pointer

b. *about:* runs the simulator for one clock cycle

3. **inject_flit**
   a. *input:* simulator pointer, NocFlit/MCNocFlit reference, payload
   b. *output:* bool value of successful injection
   c. *about:* injects a flit
4. **set_credit_return_callback**
   a. *input*: simulator pointer, source bridge_interface ID, pointer to user-implemented callback function
   b. *output*: bool value of successful setting of callback function
   c. *about*: sets the callback function that handles a credit issued by a transmitting bridge to a host
5. **set_eject_flit_callback**
   a. *input*: simulator pointer, destination bridge_interface ID, pointer to user-implemented callback function
   b. *output*: bool value of successful setting of callback function
   c. *about*: sets the callback function that handles an ejected flit and issues a credit
6. **send_credit_rxif**
   a. *input*: simulator pointer, destination bridge_interface ID
   b. *output*: bool value of successful issuing of credit
   c. *about*: to be called by user-implemented 'eject flit' callback function to issue a credit to the receiving bridge
7. **set_perf_logging**
   a. *input*: desired level
   b. *about*: sets the performance statistics logging level
8. **set_sim_safety_check_level**
   a. *input*: desired level
   b. *about*: sets the extent of safety checks performed by simulation routines
9. **run_commands**
   a. *input*: simulator pointer, vector of commands, output stream
   b. *output*: bool value of whether all commands succeeded and NoC remains in 'Mapped' state
   c. *about*: runs a list of commands on the NoC
10. **query_stats**
    a. *input*: simulator pointer, channel name as string, logging type
    b. *output*: EventStats containing the min, max, avg and number of events of the requested statistic.
    c. *about*: returns the throughput, occupancy, or latency of a channel
11. **query_end_to_end_latency**
    a. *input*: simulator pointer, src bridge_interface ID, dest bridge_interface ID
    b. *output*: EventStats containing the min, max, avg and number of events of the requested statistic.
    c. *about*: returns the end to end latency of the messages from source(s) to destinations(s).

12. **reset_stats**
    a. *input*: simulator pointer, channel name as string, logging type
    b. *output*: true if the reset of stats was successful
    c. *about*: resets the stat logger associated with the channel(s)
13. **set_perf_sim_log_file**
    a. *input*: simulator pointer, bool value of whether to enable or disable logging, output stream
    b. *output*: bool value of whether given output stream is good
    c. *about*: enables or disables writing of simulator step output to an output stream

9 WAIVERS

9.1 LINT WAIVERS

9.1.1 Waivers for Cadence HAL RTL Lint Check

Following rules are waived based on review of occurrences in the design.

CONSTC: "Constant conditional expression" There are instances in the code where a parameter may be directly used in a logic expression. There are localparam definitions using expressions based on other parameters.

MEMSIZ: "Memory declaration for '%s' defines a single bit memory word". Certain parameterized vector arrays may become array of 1-bit vectors based on the parameter value in that configuration

SHFTNC: "Shift by non-constant" As a coding practice we allow use of non-constant shifts to represent multiplexers

CLKUCL: "The clock '%s' drives a combinational logic" There are clock gating modules built into the design. These are allowed to violate this rule

UNCONO: "Port '%s' (which is being used as an output) of entity/module ' %s' is being driven inside the design but not connected (either partially or completely) in its instance '%s'" Design modules are parameterized. In some configurations of these parameterized blocks, some output ports may be left unconnected when there is no functional path downstream of that port. These are usually driven by constants in the module.

BSINTT: "Bit/part select of integer or time variable '%s' encountered" Range select of integer variable used as loop index is allowed in our designs.

IPRTEX: "Integer is used in port expression" Subset configurations may be derived from parameterized template designs by driving unused inputs with constants and leaving used outputs unloaded.

URDWIR: "Wire '%s' defined in module '%s' does not drive any object but is assigned at least once" Some internal signals are defined as tap points for functional checkers. Some internal logic

clouds may also be unused when certain parameter values remove logic using them. Synthesis tools are allowed to optimize these.

URAWIR: Wire '%s' defined in module '%s' is unused (neither read nor assigned)" Some software generated files may contain unused wires

URDREG: "Local register variable '%s' is not read but assigned at least once in the module '%s'" Due to parameterized code for fine grained logic optimization. Some registers may remain unused when certain parameter values remove or disable logic using them. Synthesis tools are allowed to optimize these.

VERREP: "Repeated usage of identifier or label name '%s'" local scope loop variables and genvars are allowed to be reused

USEPAR: "Parameter '%s' is unused" Parameters to be used by certain code sections may remain unused if those code sections are removed/disabled due to parameters

FDTHRU: "Feedthrough detected from input '%s' to output '%s'" In certain parameterized configuration of blocks there may be an input to output feedthrough

POOBID: "Variable index/range selection of '%s' is potentially outside of the defined range" Based on values of parameters, non-power-of-two arrays may exist in the design. When indexed with pointers, this warning may occur. However functionally it is ensured that the pointers do not take values outside the defined range of the array

UCCONN: "Use upper case letters for names of constants and user-defined Types" All of our Parameters are in Upper Case. This was done according to our guidelines

SEPLIN: "Use a separate line for each HDL statement". This coding style is allowed according to our guidelines

PRMVAL: "Bit width not specified for parameter '%s'". These are the parameters in the code that are defined as integers

PRMBSE: "Base not specified for parameter '%s". Our integer parameters are in decimal

LOGAND: "Bitwise AND in a conditional expression. Logical AND may have been intended". Correct operators have been used in the design.

LOGORP: "Bitwise OR in a conditional expression. Logical OR may have been intended. Correct operators have been used in the design.

LOGNEG: "Bitwise negation in a conditional expression. Logical NOT may have been intended". Correct operators have been used in the design.

CNSTLT: "Literal '%s' should be replaced with a constant". This is allowed by our RTL coding guidelines.

POIASG: "The result of operation may lead to a potential overflow". Verified by our verification environment to ensure that there will never be an overflow. In some case if there is an overflow, it is intended.

URDPRT: "The Input/inout port '%s' defined in the %s '%s' is unread but assigned" Some output ports may remain unused depending on the spec of the NoC.

BOUINC: "Lower bound of '%s' is not '%d". This style of signal declaration is allowed according to our coding guidelines

Other rules which are waived and follow are design guidelines are

AMSKWD, OPRNUM, MXUANS, NOBLKN, ONPNSG, TROPCC, MPCMPE, BITUSD, ALOWID, AUTOBX, CDNOTE, CLKINF, FSMIDN, IDLENG, INFNOT, INSYNC, NUMDFF, REDOPR, OPRCAT, PRTCNT, FTNNAS, DIFFMN, ALOWNM, FFWNSR, SYNPRT, DALIAS, FFWASR, MULBAS, SYNASN, DIFRST, UNCONN, REVROP, TROPCZ, INDXOP, PRMNAM, RDOPND, OPLVNC, LRGOPR, NFCASE, MICAWS, CSTEXP, HASLEX, HASPGM, IFSMCD, IMPTYP, OBMEMI, MAXLEN, LCVARN, EXPIPC, STYVAL, SIGLEN, NUMSUF, NBGEND, CDWARN, RPTVAR, IMPDTC, CONSBS

9.1.2 Waivers for Spyglass lint check

Following rules are waived based on review of occurrences in the design. These are global waivers; other code specific waivers are placed in the RTL files as pragmas.

NoBusPartClock-ML: " Clock name '%s' is not a simple name". CFG IP RTL may concatenate clocks into a bus in some configurations.

ComplexExpr-ML: “Complex expressions". CFG IP RTL may implement complex logical expression in some RTL blocks.

UnloadedNet-ML: “Unloaded net”. Due to high configurability, CFG IP RTL may have unloaded nets in a design, especially used with “generate” statement.

LineLength: “Line-length exceeds defined limit". For complex logical expression, CFG IP RTL’s line length may exceed defined limit.

STARC-2.1.1.1: “function+assign”. CFG IP RTL may use both assign statement and always block to describe combinational logic.

STARC-2.6.2.2: "Signal set and read in the same block”. CFG IP RTL coding guidance permits such design practice.

WRN_47: “Improper repetition multiplier %s in concatenation". CFG IP RTL may pad the signals with constants in some configurations.

W111: “Not all elements of an array are read". CFG IP RTL may implement registers in an “always” block partially due to configurability.

W120: “Variable ‘%s’ declared but not used". CFG IP RTL may have non-used wire declaration due to configurability.
W175: “Parameter ‘%s’ declared but not used ". CFG IP RTL may have non-referenced parameter declaration due to configurability.
W191: “Function declared but not used". CFG IP RTL may not use all the functions in commonly shared function file.

W240: “Input ‘%s’ declared but not read". CFG IP RTL may declare non-used input ports in a design due to configurability.

W328: “NS – Truncation in constant conversion, without loss of data". CFG IP RTL may implement truncation during constant conversion due to configurability.

W456: “A signal is included in the sensitivity list of a process/always block but not all of its bits are read in that block". CFG IP RTL may have non-used signals declared in sensitivity list of an “always” block.

W488: “A bus/array appears in the sensitivity list but not all bits of it are read in the contained block/process". Similar to W111, CFG IP RTL may implement the registers partially due to configurability.

W497: “Not all bits of a bus are set". CFG IP RTL may implement un-assigned but dont_care signals due to configurability.

W498: "Not all bits of a bus are read". CFG IP RTL may implement non-used signals due to configurability.

W528: "Variable '%s' set but not read". CFG IP RTL may implement non-used variable due to configurability.

W563: "Reduction of a single-bit expression is redundant". CFG IP RTL coding guideline permits such design practice.

9.2 CDC WAIVERS

9.2.1 Tool Details

- Tool: Cadence Conformal Constraint Designer CDC
- Version: Ver15.10-D182

9.2.2 False CDC Violations – Design Intent

False Violation #1

The tool reports false CDC violations on following clock cross modules, when *fast_clk* and *slow_clk* signals are specified in different clock domains by CFG auto-generated SDC files. The leaf level module is ns_clkcross_buffer, instanced in all four of these modules where the false CDC violations are reported

- ns_clkcross_fast_to_slow
- ns_clkcross_slow_to_fast
- ns_clkcross_rwack_slow_to_fast.v
- ns_clkcross_rwack_fast_to_slow.v

***Resolution*:** *fast_clk* and *slow_clk* clocks need to be specified to be in the same clock domain manually in the sdc constraint file**.** The clock cross modules are phase-aligned synchronous clock crossers with an N:1 or 1:N ratio

***Work around*:** The customer can post-edit the auto-generated SDC files to specify *fast_clk* and *slow_clk* clocks in the same clock domain.

Example:

```
set clock_clk_A_sync_group [get_clocks [list fast_clk slow_clk]]
set_clock_groups -name async_clock_group -asynchronous \
                 -group clock_clk_A_sync_group \
```

-group [get_clocks clk_C]

9.2.3 False CDC Violations – Tool Issues

False Violation #1

False violation paths occur occasionally due to Conformal Constraint Designer not being able to identify asynchronous FIFOs across two clock domains

from_instance:
u_ns_fabric/*/u_ns_router_*/u_ns_router/G_*_INBLK_ENB.u_*_inblk/G_IVCBUF_IVCCTRL[*].G_IVC_ENB.G_IVCBUF_ASYNC.u_ivcbuf_async/reg_array_reg*

Resolution: CDC violations from above starting path should be waived. The CDC tool fails to identify *ns_ivcbuf_1rp_a_c* module inside ns_router as asynchronous FIFO. The *to_instance* of these paths are various destination registers in the design.

from_instance:
u_ns_fabric/*/*/*/*/*/u_ns_strrxswitch/ns_strrxbrdg_layeriflogic0/strrxbrdg_lay[*].strrxfifologicvc*.IVCFIFO_ASYNC_*.vc*_fifo_async/reg_array_reg*

Resolution: CDC violations from above starting path should be waived. The CDC tool fails to identify *ns_ivcbuf_1rp_a_c* module inside ns_strrxswitch as asynchronous FIFO. The *to_instance* of these paths are various destination registers in the design.

9.2.4 Known CDC Violations

There are currently no known CDC violations in our design

Intel Corporation

2200 Mission College Blvd,

Santa Clara, CA - 95054.