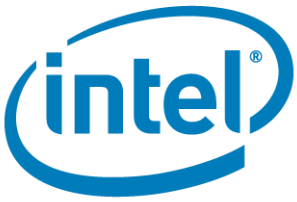

CFG Orion

IP Integration Specification

Version: ORION-19.07L

Revision: 0.0

Intel Confidential

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

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 Orion IP Integration Specification

About This Document

This document describes the guidelines for seamless integration of CFG Orion 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
- ARM AMBA 4 interconnect standard

Related Documents

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

- CFG NocStudio Orion User Manual
- CFG Orion 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

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.

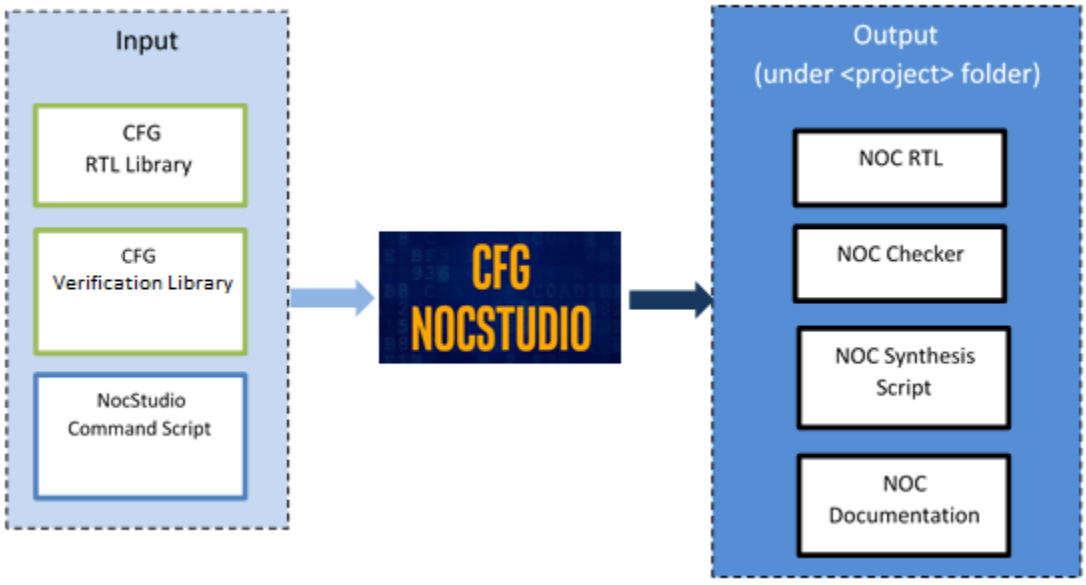

Figure 1 NoC IP generation flow

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.

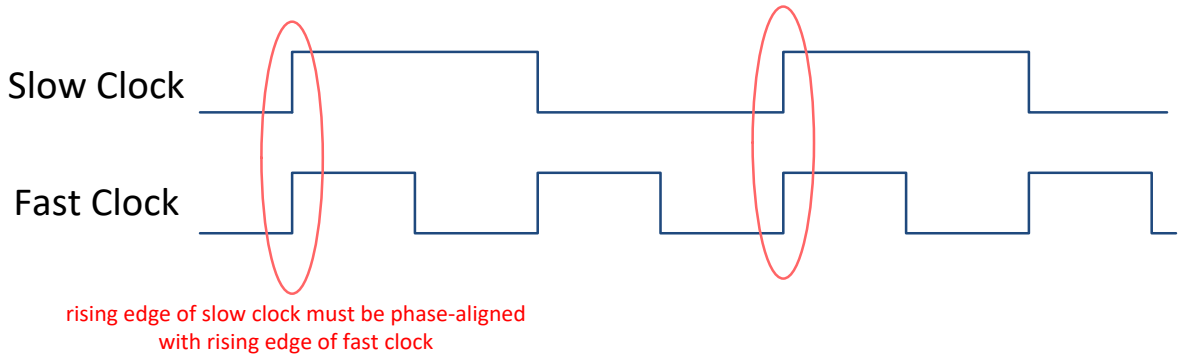

Figure 2 Synchronous clock crossers require alignment of rising edge.

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.

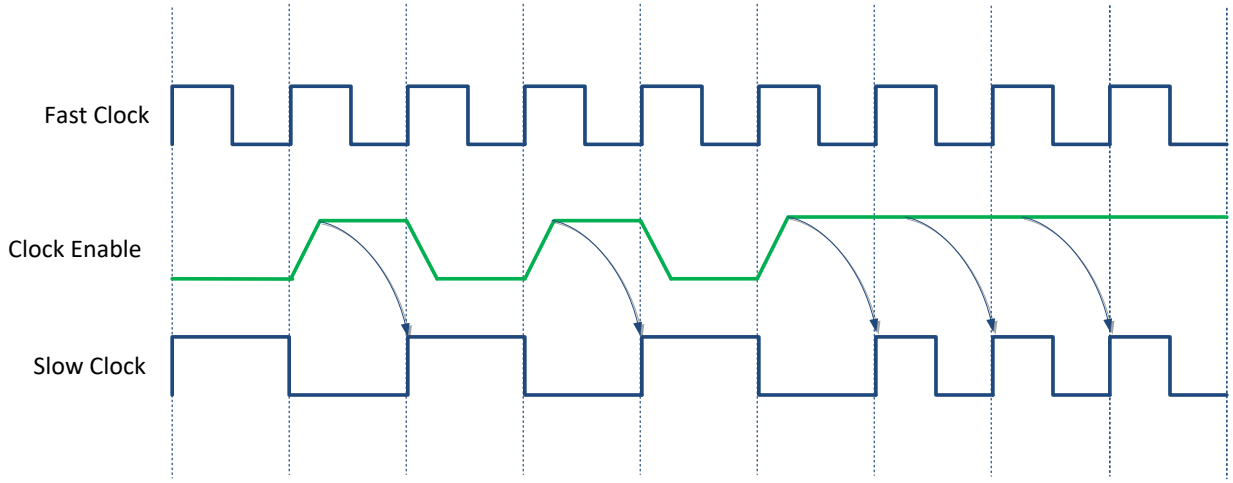

Figure 3 A clock enable input indicates when the rising edges are shared by the clocks.

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.

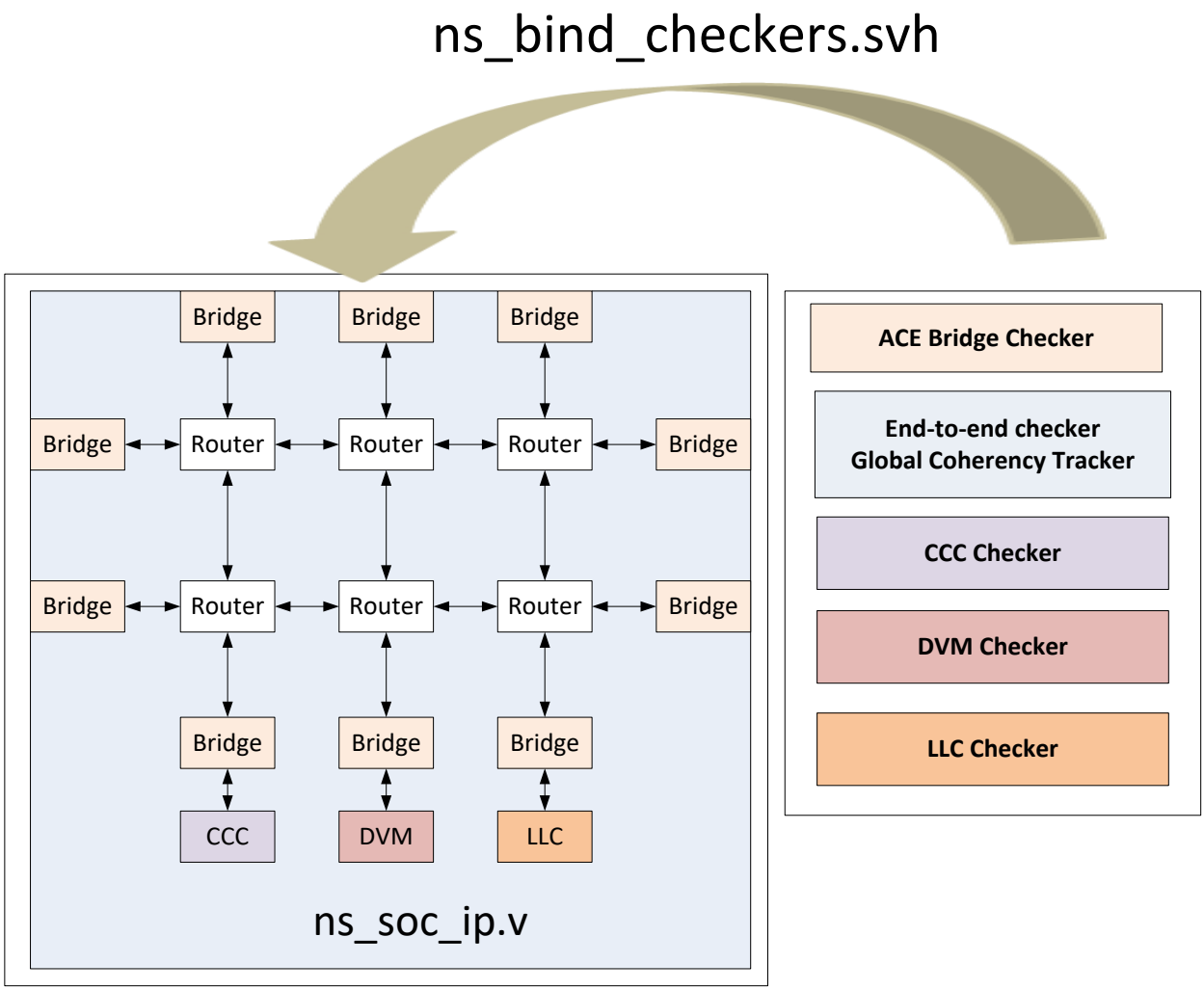

Figure 4 NoC checkers binding to RTL

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> _pp_to_pd_map.txt	ns_ pp_to_pd_map.txt for each NoC. NOTE: Only generated for Low Power NoCs.	Verification
ns_<project> _pd_to_pdnum_map.txt	ns_pd_to_pdnum_map.txt for each NoC. NOTE: Only generated for Low Power NoCs.	Verification
ns_<project>_preloader_declare.sv	ns_preloader_declare.sv for each NoC. NOTE: Only present for Pegasus.	Verification

ns_<project> _preloader_init.sv	ns_preloader_init.sv for each NoC. NOTE: Only present for Pegasus.	Verification
ns_<project> _preloader_set.sv	ns_preloader_set.sv for each NoC. NOTE: Only present for Pegasus.	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 the installation 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 8 Directory and files with tag_project_name 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 checkers IP library	Verification
noc_rtl_agents/	NoC RTL agents library.	RTL

noc_verif_agents/	NoC verification agents IP library	Verification
cpfs/	NoC hierarchical CPF files. NOTE: Only generated for Low Power NoCs.	CPF

- 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_routing_table.sv file in the combined project directory by concatenating the contents of ns_routing_table.sv from each NoC.
- Create a combined ns_power_map_table.sv file in the combined project directory by concatenating the contents of ns_power_map_table.sv from each NoC. NOTE: Only present for Low Power NoCs.
- Create a combined ns_global_defines.vh by taking noc_verif_cust/ns_global_defines.vh from the first NoC project directory then adding any `defines from noc_verif_cust/ns_global_defines.vh of subsequent NoCs that are not already present. There should be no duplicate `defines. 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_E2E_CHECKER_TOP \
            `NS_<PROJECT1>_NOC_TOP.ns_amba_noc_e2e_checker
`define NS_REGBUS_E2E_CHECKER_TOP \
            `NS_<PROJECT1>_NOC_TOP.ns_regbus_e2e_checker

`define NS_<PROJECT2>_NOC_TOP       <Hierarchical path to second NoC top module>
`define NS_ASYNC_FIFO_CHECKER_TOP \
    `NS_<PROJECT2> _NOC_TOP.ns_noc_async_fifo_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+$COMBINED_PATH/noc_rtl
+incdir+$COMBINED_PATH/noc_rtl_agents/tunnel
+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_rtl_agents/tunnel
-y $COMBINED_PATH/noc_verif_ip
-y $COMBINED_PATH/noc_verif_ip_proj
$COMBINED_PATH/noc_verif_ip/ns_amba_struct.sv
$COMBINED_PATH/noc_verif_ip/ns_brdg_cfg.sv
$COMBINED_PATH/noc_verif_ip/ns_mstrbrdg_cfg.sv
$COMBINED_PATH/noc_verif_ip/ns_aximpabrdg_cfg.sv
$COMBINED_PATH/noc_verif_ip/ns_regbus_struct.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
-f $COMBINED_PATH/noc_verif_bench/noc_verif_bench.vc
$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 verification checkers

Checkers	Instantiated
NoC End-to-End Checker	One instance per NoC
Regbus End-to-End Checker	One instance per regbus layer
AMBA Master Bridge Checker	One instance per AMBA Master bridge RTL
AMBA Slave Bridge Checker	One instance per AMBA Slave Bridge RTL
Shared Interface Bridge Checker	One instance per Shared Interface bridge RTL
Regbus Ring Slave Checker	One instance per Regbus Ring Slave RTL
Link Clock Cross FIFO Checker	One instance per link clock cross FIFO
Asynchronous FIFO Checker	One instance per voltage domain crossing FIFO or link clock cross FIFO
LLC Checker	One instance per LLC
CSR Checker	One instance per CSR
SRAM Checkers	One instance per LLC RAM

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 ns_soc_ip instance within user testbench.	None.
`NS_END_OF_SIM	Map to a 1-bit signal with single 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 AMBA NoC End-to-End Checker.	Recommend to map 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 AMBA NoC End-to-End Checker.	Recommend to map to a plusarg variable to allow run-time control of value.
`NS_REGBUS_END2END_CHECKER_EN	Set to 1 to enable, 0 to disable Regbus End-to-End Checker.	Recommend to map to a plusarg variable to allow run-time control of value.
`NS_REGBUS_E2E_LOG	Set to 1 to enable, 0 to display traffic logging by the Regbus End-to-End Checker.	Recommend to map to a plusarg variable to allow run-time control of value.
`NS_ACEMSTRBRDG_CHECKER_EN	Set to 1 to enable, 0 to disable AMBA Master Bridge Checker.	Recommend to map to a plusarg variable to allow run-time control of value.
`NS_ACESLVBRDG_CHECKER_EN	Set to 1 to enable, 0 to disable AMBA Slave Bridge Checker.	Recommend to map to a plusarg variable to allow run-time control of value.
`NS_REGBUS_RING_SLV_CHECKER_EN	Set to 1 to enable, 0 to disable Regbus Ring Slave Checker.	Recommend to map to a plusarg variable to allow run-time control of value.

`NS_LINK_CLK_CROSS_FIFO_CHECKER_EN	Set to 1 to enable, 0 to disable Link Clock Cross FIFO Checker.	Recommend to map to a plusarg variable to allow run-time control of value.
`NS_ASYNC_FIFO_CHECKER_EN	Set to 1 to enable, 0 to disable Async FIFO Checker.	Recommend to map to a plusarg variable to allow run-time control of value.
NS_ASYNC_FIFO_LOG_EN	Set to 1 to enable, 0 to display traffic logging by the Async FIFO checker.	Recommend to map to a plusarg variable to allow run-time control of value.
`NS_LLC_CHECKER_EN	Set to 1 to enable, 0 to disable LLC checker.	Recommend to map 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 to map to a plusarg variable to allow run-time control of value.
`NS_LLC_TAG_SRAM_CHECKER_EN	Set to 1 to enable, 0 to disable SRAM checker for LLC tag RAM.	Recommend map to a plusarg variable to allow run-time control
`NS_LLC_DATA_SRAM_CHECKER_EN	Set to 1 to enable, 0 to disable SRAM checker for LLC data RAM.	Recommend map to a plusarg variable to allow run-time control
`NS_CSLV_CHECKER_EN	Set to 1 to enable, 0 to disable the Configurable Slave Checker	Recommend map to a plusarg variable to allow run-time control

4.3 FAST INITIALIZATION FOR LLC

By default, hardware explicitly initializes LLC RAMs after reset. This can slow down simulation time. If the user wants to avoid the extra simulation time, CFG provides the `define in the following table that will immediately initialize LLC RAMs with a backdoor mechanism. If user does preloading using the preload_pegasus mechanism, this `define should be used in conjunction.

Table 12 `define variables for fast back-door initialization

`define Variable Name	Description
`NS_FORCE_RTL_LLC_SHORT_INIT	If defined, immediately initializes LLC RAMs using a backdoor mechanism. If not defined, hardware will explicitly perform initialization.

4.4 Fast Initialization for Configurable Slave

By default, hardware explicitly initializes Configurable Slaves after reset. This can slow down simulation time. If the user wants to avoid the extra simulation time, CFG provides the `define in the following table that will immediately initialize Configurable Slave Blocks with a backdoor mechanism.

Table 13 `define variables for fast back-door initialization

`define Variable Name	Description
`NS_FORCE_RTL_CSLV_SHORT_INIT	If defined, immediately initializes Configurable Slave using a backdoor mechanism. If not defined, hardware will explicitly perform initialization.

4.5 Usage Modes

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

Table 14 Recommended checker settings

Usage mode	Bringup 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_ACEMSTRBRDG_CHECKER_EN	1	1	0
`NS_ACESLVBRDG_CHECKER_EN	1	1	0
`NS_REGBUS_RING_SLV_CHECKER_EN	1	1	0
`NS_LINK_CLK_CROSS_FIFO_CHECKER_EN	1	1	0
`NS_ASYNC_FIFO_CHECKER_EN	1	1	0
`NS_ASYNC_FIFO_LOG_EN	1	1	0
`NS_LLC_CHECKER_EN	1	1	1
`NS_CSR_CHECKER_EN	1	1	0
`NS_LLC_TAG_SRAM_CHECKER_EN	1	1	1
`NS_LLC_DATA_SRAM_CHECKER_EN	1	1	1
`NS_CSLV_CHECKER_EN	1	1	0

4.6 PERFORMANCE DEBUG

At the end of simulation, a set of performance statistics are printed to aid performance debug. These performance statistics are collected from RTL registers which can be used to extract the same information post-silicon.

When ` NS_NOC_END2END_CHECKER_EN is set to 1, a set of performance statistics messages are displayed for each master bridge at the end of every simulation. These messages can shed light on root causes for any performance bottlenecks for the simulation, such as serialization events, percentages of narrow requests, time spent waiting for grant, etc. Search for 'SIMULATION PERFORMANCE STATS' in the simulation log.

The following is an example of such messages:

```
SIMULATION PERFORMANCE STATS: AR channel total number of requests =
       200.
SIMULATION PERFORMANCE STATS: AR channel total number of narrow
       requests = 0 (0%).
SIMULATION PERFORMANCE STATS: AR channel total number of split requests
       = 9 (4%).
SIMULATION PERFORMANCE STATS: AR channel total number of cycles waiting
       for serialization due to request split = 2955.
SIMULATION PERFORMANCE STATS: AR channel total number of cycles waiting
       for serialization due to ARID reuse = 0.
SIMULATION PERFORMANCE STATS: AR channel total number of cycles waiting
       for max_outstanding = 698.
SIMULATION PERFORMANCE STATS: AR channel total number of cycles waiting
       for autowake power domains to power up = 0.
SIMULATION PERFORMANCE STATS: AR channel total number of cycles waiting
       for decode err processing = 0.
SIMULATION PERFORMANCE STATS: AR channel total number of cycles waiting
       for grant from noc = 240.
SIMULATION PERFORMANCE STATS: AR channel total number of requests =
       306.
SIMULATION PERFORMANCE STATS: AR channel total number of narrow
       requests = 0 (0%).
SIMULATION PERFORMANCE STATS: AR channel total number of split requests
       = 12 (3%).
SIMULATION PERFORMANCE STATS: AW channel total number of cycles waiting
       for serialization due to request split = 1323.
SIMULATION PERFORMANCE STATS: AW channel total number of cycles waiting
       for serialization due to AWID reuse = 0.
SIMULATION PERFORMANCE STATS: AW channel total number of cycles waiting
       for max_outstanding = 194.
SIMULATION PERFORMANCE STATS: AW channel total number of cycles waiting
       for autowake power domains to power up = 0.
SIMULATION PERFORMANCE STATS: AW channel total number of cycles waiting
       for decode err processing = 0.
SIMULATION PERFORMANCE STATS: AW channel total number of cycles waiting
       for grant from noc = 1723.
```

4.7 CHECKERS

4.7.1 Terminology

The types of checks performed are as follows:

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

Unsupported – These checks flag violations of AMBA interface protocol features 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.

4.7.2 AMBA NoC End-to-End Checker

The AMBA NoC End-to-End Checker to verifies the correct operation of the NoC design. All requests and responses transmitted into the NoC are tracked to construct a global reference database of expected results at the output interfaces. The traffic transmitted out of each NoC interface is then compared against the global reference database to ensure all relevant data content and command fields arrive with the correct values and in the correct order. Unexpected requests or responses, any traffic sent to an incorrect destination, lost commands or responses, extra commands or responses, unexpected command modification, traffic received that mismatches against expected traffic, any traffic 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 15 AMBA NoC end-to-end checks

Description of check	Present	Type of check
Every AR request is tracked with its corresponding R response for the roundtrip check to ensure correct propagation of command fields, propagation of data size and content and ordering within the NoC.	Always	Functional
Every AR request that enters the NoC arrives at the correct destination NoC slave bridge, with the correct ARADDR, ARID, ARCACHE, ARBURST, ARPROT, ARQOS, ARLOCK, ARUSER and ARREGION, in the correct order.	Always	Functional
ARCACHE, ARPROT and ARQOS overrides configured in the NoC on master and slave bridges are transported correctly	Always	Functional
Every AR request marked as non-modifiable remains unmodified except for the cases where NoC is	Always	Functional

expected to force modification for functional correctness.		
Every R response that enters the NoC arrives at the correct destination NoC master bridge with the correct RID, RDATA, RRESP, RUSER, RLAST, in the correct order.	Always	Functional
Every AR request that fails address map lookup receives a decode error in its R response.	Always	Functional
Every AR request that results in slave error receives the corresponding slave error in its R response.	Always	Functional
Every pair of AW and W requests 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 NoC.	Always	Functional
Every AW and W request that enters the NoC arrives at the correct destination NoC slave bridge with the correct AWADDR, AWID, AWBURST, AWCACHE, AWPROT, AWQOS, AWLOCK, AWUSER, AWREGION, WID, WUSER, WDATA, WSTRB and WLAST, in the correct order.	Always	Functional
Every AW and AR request with an address that has relocation arrives at the correct destination NoC slave bridge with the correct relocated AWADDR/ARADDR	Always	Functional
AWCACHE, AWPROT and AWQOS overrides configured in the NoC on master and slave bridges are transported correctly.	Always	Functional
Every AW request marked as non-modifiable remains unmodified except for the cases where NoC is expected to force modification for functional correctness.	Always	Functional
In AXI3 mode, WID arrives at the slave destination as sent from the master.	Always	Functional

Every B response that enters the NoC arrives at the correct destination NoC master bridge with the correct BID, BRESP and BUSER, in the correct order.	Always	Functional
Every AW request that fails address map lookup receives a decode error in its B response.	Always	Functional
Every AW request that results in slave error receives the corresponding slave error in its B response.	Always	Functional
Additional checks within the NoC for early detection of corrupt or misrouted traffic for ease of debugging.	Always	Functional
On AXI4 Master Bridge, if 'axi4_unique_aid' is set and there is no reordering buffer, outstanding ARIDs to the same slave must be unique.	Always	Functional
On AXI4 Master Bridge, if 'axi4_unique_aid' is set and there is no reordering buffer, outstanding AWIDs to the same slave must be unique.	Always	Functional
On AXI4 Slave Bridge, if 'axi4_unique_aid' is set then outstanding ARIDs issued onto the slave bus must be unique.	Always	Functional
On AXI4 Slave Bridge, if 'axi4_unique_aid' is set then outstanding AWIDs issued onto the slave bus must be unique.	Always	Functional
On AXI4 and ACE-lite Slave Bridge, if ‘axi4_ar_org_id_enb ' is set then, id of the master issuing the AR transaction should be propagated to the slave.	Always	Functional
On AXI4 and ACE-lite Slave Bridge, if ‘axi4_aw_org_id_enb ' is set then, id of the master issuing the AW transaction should be propagated to the slave.	Always	Functional
No request or response is in flight in the NoC.	Always	Exit
All requests and responses during the simulation are accounted for.	Always	Exit
All verification checkers indicate no error.	Always	Exit
Every AR request that should travel via the Fast Tap does take the Fast Tap.	Fast Tap	Functional
Every AR request using the Fast Tap is tracked together with the corresponding R response for	Fast Tap	Functional

roundtrip check to ensure correct propagation of command fields, propagation of data and ordering within the NoC.		
Fast Tap and the regular AR path must not have the same ARID outstanding at the same time.	Fast Tap	Functional
Requests must not attempt to travel through Fast Tap when there is no Fast Tap present.	Fast Tap	Functional
No inflight requests in Fast Tap. All Fast Tap requests are accounted for and have completed correctly.	Fast Tap	Exit
Correct propagation of AXI4/AXI3 transactions from all the enabled ports in SIB	Shared Interface Bridge	Functional
Port check error assertion is correct when port check is enabled.	Shared Interface Bridge	Functional
Check X or Z on all incoming axi ports during valid transaction	Shared Interface Bridge	Functional
All AR requests are accounted for with respect to R responses.	Shared Interface Bridge	Functional
All AW and W requests are accounted for with respect to B responses.	Shared Interface Bridge	Functional
No out-of-order or extra commands exiting SIB.	Shared Interface Bridge	Functional
No pending transactions.	Shared Interface Bridge	Exit
All requests on AR, AW, W buses are correctly processed and propagated.	AXI3 Slave Agent*	Functional
All responses on R and B buses are correctly processed and propagated.	AXI3 Slave Agent*	Functional
All AR requests are accounted for with respect to R responses.	AXI3 Slave Agent*	Functional
All AW and W request are accounted for with respect to B responses.	AXI3 Slave Agent*	Functional
WDATA is not interleaved across different WIDs.	AXI3 Slave Agent*	Functional

WID must match AWID for same request.	AXI3 Slave Agent*	Functional
AR, AW, W, R, B control signals must not be X or Z when corresponding VALID and READY are high.	AXI3 Slave Agent*	Protocol
No request is in-flight at end of simulation. All verification structures are empty.	AXI3 Slave Agent*	Exit
Correct conversion of AXI4 AxREGION to APB PSEL.	APB Agent*	Functional
Pass through of AXI4 AxPROT to APB PPROT.	APB Agent*	Functional
Correct conversion of AXI4 AxADDR to APB PADDR.	APB Agent*	Functional
Pass through of AXI4 AWREGION, AWPROT, WDATA and WSTRB to respective APB slave interface.	APB Agent*	Functional
Pass through of AXI4 ARREGION and ARPROT to respective APB slave interface.	APB Agent*	Functional
Pass through of APB slave responses for APB write requests depending on slave version to AXI4 BRESP.	APB Agent*	Functional
Pass through of APB slave responses for APB read requests depending on slave version to AXI4 RDATA and RRESP.	APB Agent*	Functional
No out-of-order or extra commands exiting converter.	APB Agent*	Functional
Correct error response for unsupported requests entering converter.	APB Agent*	Functional
PWRITE, PENABLE must not be X or Z when out of reset.	APB Agent*	Protocol
APB control signals must not be X or Z during valid accesses.	APB Agent*	Protocol
No pending AXI4 or APB transactions.	APB Agent*	Exit
All AXI4 requests on AR, AW, W channels are correctly translated and propagated to AXI4-Lite slave.	AXI4-Lite Slave Agent*	Functional
All AXI4-Lite single beat responses on B and R channels are correctly translated and propagated to AXI4 response interface.	AXI4-Lite Slave Agent*	Functional

Pass through of AXI4 Request AxREGION and AxPROT, and AxUSER conversion to AXI4-Lite slave.	AXI4-Lite Slave Agent*	Functional
Correct conversion of AXI4-Lite response BUSER/RUSER to AXI4 interface.	AXI4-Lite Slave Agent*	Functional
All AR requests are accounted for with respect to R responses.	AXI4-Lite Slave Agent*	Functional
All AW and W requests are accounted for with respect to B responses.	AXI4-Lite Slave Agent*	Functional
No out-of-order or extra commands exiting converter.	AXI4-Lite Slave Agent*	Functional
Correct error response for unsupported requests entering converter.	AXI4-Lite Slave Agent*	Functional
AR, AW, W, R, B control signals must not be X or Z when corresponding VALID and READY are high.	AXI4-Lite Slave Agent*	Protocol
No pending AXI4 and AXI4-Lite transactions.	AXI4-Lite Slave Agent*	Exit
HMASTLOCK must be low.	AHB-Lite Master Agent*	Functional
WRAP burst size must be 16B, 32B or 64B.	AHB-Lite Master Agent*	Functional
HSIZE must not be 128B.	AHB-Lite Master Agent*	Functional
AWQOS, AWLOCK, ARQOS, ARLOCK must be low.	AHB-Lite Master Agent*	Functional
HSIZE must not be greater than data width.	AHB-Lite Master Agent*	Protocol
HADDR must be aligned with respect to HSIZE.	AHB-Lite Master Agent*	Protocol
AHB-Lite control signals must not be X or Z during SEQ or NONSEQ transfers.	AHB-Lite Master Agent*	Protocol
Correct conversion of AHB-Lite write transfer to AXI4 AWADDR, AWBURST, AWSIZE, AWLEN, AWPROT and AWCACHE.	AHB-Lite Master Agent*	Functional

Correct conversion of AHB-Lite read transfer to AXI4 ARADDR, ARBURST, ARSIZE, ARLEN, ARPROT and ARCACHE.	AHB-Lite Master Agent*	Functional
Correct conversion of AXI4 RDATA to AHB-Lite read request.	AHB-Lite Master Agent*	Functional
Correct conversion of AXI4 WDATA and WSTRB to AHB-Lite write request.	AHB-Lite Master Agent*	Functional
AxBURST must not be ‘h0 or 'h3.	AHB-Lite Master Agent*	Functional
AHB-Lite read response must not return for at least 1 cycle after AHB-Lite read request is issued.	AHB-Lite Master Agent*	Functional
No write and read FIFO overflow or underflow.	AHB-Lite Master Agent*	Functional
Correct conversion of AHB-Lite HRESP to AXI4 RRESP, BRESP.	AHB-Lite Master Agent*	Functional
All valid signals low, FSM in idle state, all FIFOs empty.	AHB-Lite Master Agent*	Exit
No outstanding transactions. All verification structures are empty	AHB-Lite Master Agent*	Exit
Correct conversion of AXI4 AW request and W data to AHB-Lite write request and write data.	AHB-Lite Slave Agent*	Functional
Correct conversion of AHB-Lite write response to AXI4 B response.	AHB-Lite Slave Agent*	Functional
Correct conversion of AXI4 AR request to AHB-Lite read request.	AHB-Lite Slave Agent*	Functional
Correct conversion of AHB-Lite read data to AXI4 R data.	AHB-Lite Slave Agent*	Functional
Correct conversion of AHB-Lite read response to AXI4 R response.	AHB-Lite Slave Agent*	Functional
HMASTLOCK is always low.	AHB-Lite Slave Agent*	Functional
HSIZE is not greater than AHB-Lite slave bus width.	AHB-Lite Slave Agent*	Protocol

AWADDR is aligned with respect to AWSIZE.	AHB-Lite Slave Agent*	Protocol
AWSIZE is not greater than bus width.	AHB-Lite Slave Agent*	Protocol
ARSIZE is not greater than bus width.	AHB-Lite Slave Agent*	Protocol
Total transfer data size of AR commands does not cross 1KB boundary.	AHB-Lite Slave Agent*	Protocol
Total transfer data size of AW commands does not cross 1KB boundary.	AHB-Lite Slave Agent*	Protocol
WSTRB has contiguous bits set for all bytes in a write (no holes).	AHB-Lite Slave Agent*	Protocol
WSTRB is aligned with respect to AWSIZE.	AHB-Lite Slave Agent*	Protocol
AHB-Lite control signals must not be X or Z when corresponding HREADY and HREADY_IN are high.	AHB-Lite Slave Agent*	Protocol
Internal RTL micro-architectural checks.	AHB-Lite Slave Agent*	Functional
Bridge RTL FSM is restored to idle state.	AHB-Lite Slave Agent*	Exit
All bridge RTL FIFOs are empty.	AHB-Lite Slave Agent*	Exit
All bridge RTL outstanding transaction counts are zero.	AHB-Lite Slave Agent*	Exit
No outstanding transactions. All verification structures are empty	AHB-Lite Slave Agent*	Exit

* Present if agent of the specified protocol type is present in NocStudio configuration.

4.7.2.1 AMBA NoC End-to-End Fine-Grained User Control

For a subset of the above checks, fine-grained user control is provided to individually enable or disable the checks. For each check listed in the following table, setting the corresponding `define to 1 disables the check; setting it to 0 enables 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 16 Fine-grained user control of AMBA NoC end-to-end checks

Description of check	`define to control	Present	Default value

WSTRB has contiguous bits set for all bytes in a write (no holes) and is aligned with respect to AWSIZE.	`NS_AXI2AHB_LEGAL_WSTRB_CHECK_DISABLE	AHB-Lite Slave Agent*	0
AWADDR is aligned with respect to AWSIZE.	`NS_AXI2AHB_LEGAL_AWADDR_CHECK_DISABLE	AHB-Lite Slave Agent*	0

* Present if agent of the specified protocol type is present in NocStudio configuration.

4.7.2.2 AMBA NoC End-to-End Traffic Logs

The AMBA NoC End-to-End Checker has the capability of generating a set of end-to-end traffic log files during the simulation to provide visibility of the traffic on each master bridge and each slave bridge within the NoC. The following table lists the settings required to enable the end-to-end logs.

Table 17 Settings to enable AMBA NoC End-to-End Checker traffic logs

`define to control	Value
`NS_NOC_END2END_CHECKER_EN	1
`NS_E2E_LOG	1

The file names of the logs have the following format with <bridge_id> corresponding to the bridge id of each bridge assigned by NocStudio.

Table 18 AMBA NoC End-to-End Checker log files

File name	Description
ns_noc_acemstrbrdg_<bridge_id>_ar_r.log	AXI or ACE master bridge traffic log for AR and R channels

ns_noc_acemstrbrdg_<bridge_id>_aw_w_b.log	AXI or ACE master bridge traffic log for AW, W and B channels
ns_noc_aceslvbrdg_<bridge_id>_ar_r.log	AXI or ACE slave bridge traffic log for AR and R channels
ns_noc_aceslvbrdg_<bridge_id>_aw_w_b.log	AXI or ACE slave bridge traffic log for AW, W and B channels
ns_noc_acemstrbrdg_<bridge_id>_ac.log	ACE master bridge traffic log for AC channel
ns_noc_acemstrbrdg_<bridge_id>_cr_cd.log	ACE master bridge traffic log for CR and CD channels
ns_noc_aceslvbrdg_<bridge_id>_ac.log	ACE slave bridge traffic log for AC channel
ns_noc_aceslvbrdg_<bridge_id>_cr_cd.log	ACE slave bridge traffic log for CR and CD channels

The above table shows the traffic log files for all major channels produced for each bridge in the NoC. Each log file records Noc-level information for all the traffic seen by each bridge.

4.7.2.2.1 AMBA NoC End-to-End Traffic Logs for Reads

The following is an example of ns_noc_acemstrbrdg_<bridge_id>_ar_r.log:

<sim_time> : request_id : 0x8 : split_cnt : 0 : split_size : 1024 : master_id : 0x0 : slave_id : 0x1 : AR_sent_time : 585 : unknown_dest : 0 : original_master_bridge_id : 0x0 : seq_num: 0x1 : mprt_ARADDR : 0x10000070 : mprt_ARADDR_to_slv : 0x10000070 : sys_address : 0x10000070 : mprt_ARID : 0x27 : mprt_ARLEN : 0x1

: mprt_ARSIZE : 0x3 : mprt_ARBURST : 0x1 : mprt_ARCACHE : 0x0 :

mprt_ARPROT : 0x5 : mprt_ARQOS : 0x6 : mprt_ARLOCK : 0x0 : mprt_ARUSER : 0x0 : mprt_ARSNOOP : 0x0 : mprt_ARDOMAIN : 0x3 : mprt_ARBAR : 0x0 : mprt_ARREGION : 0x0 : slave_region : 0x0 : fast_tap : 0

<sim_time> : request_id : 0x8 : split_cnt : 0 : split_size : 1024 : master_id : 0x0 : slave_id : 0x1 : R_received_time : 675 : seq_num: 0x1 : acmb_RID : 0x27 : acmb_RRESP : 0x0 : acmb_RUSER(per transaction) : 0x0 : acmb_RUSER_CL(per beat) : 0x0 : acmb_RLAST: 0x1 : derived_mprt_ARADDR : 0x10000070 :

derived_mprt_ARADDR_to_slv : 0x10000070 : sys_address : 0x10000070 : derived_sys_ARID : 0x27 : acmb_RDATA : 0x0 : fast_tap : 0

Table 19 Nomenclature of ns_noc_acemstrbrdg_<bridge_id>_ar_r.log

Field name	Description
<sim_time>	Simulation time when master bridge traffic collected and printed.
<request_id>	A unique id associated with the request (AR and R) in the master and slave bridge logs.
<master_id>	The bridge ID of the master bridge that sent the AR request, or received R response.
<split_cnt>	Each ingress request has a unique <request_id>. When a request splits into multiple transactions based on <split_size> alignment, each transaction is shown with the same request_id but incrementing <split_cnt>.
<split_size>	The address alignment in bytes that ingress requests split on.
<slave_id>	The bridge ID of the slave bridge that received the AR request or sent the R response.
<AR_sent_time>	Simulation time when the AR request was sent by the master bridge.
<unknown_dest>	Whether the request has a known destination. If unknown_dest is 1, the request failed address look-up and would get a decode error in its R response.
<original_master_bridge_id>	The bridge ID of the master bridge that sent the AR request originally. In multi-hop request, this ID represents the bridge ID of the master bridge in the first hop.

<seq_num>	The internal NoC sequence number, used for tracking ordering.
<mprt_AR*>	Fields of AR channel as seen by the master bridge.
<mprt_ARADDR_to_slv>	AR Address sent to the slave. In the case of address range relocation, mprt_ARADDR represents ARADDR sent to the master bridge and mprt_ARADDR_to_slv represents the relocated address
<sys_address>	This address represents unique system address.
<ARREGION>	Region field for the AR request assigned by the NoC. This is meaningful for slaves that use ARREGION for decoding address.
<R_received_time>	Simulation time when R response was received by the master bridge.
<acmb_R*>	Fields of R channel as seen by the master bridge.
<acmb_RUSER(per transaction)>	Per transaction portion of the RUSER.
<acmb_RUSER_CL(per beat)>	Per beat portion of RUSER, packed into 512-bit vector, with RUSER for each byte taking up 8 bits.
<derived_mprt_ARADDR>	The ARADDR that corresponds to the R response. This value is internally calculated and displayed along with each R response for ease of traffic association.
<derived_sys_ARID>	The system ARID shows the full ARID for the request. If the slave bridge has a narrower ARID width than the master bridge, it may see a truncated version of the original ARID sent by the master bridge. The system ARID is printed here for ease of traffic association.

acmb_RDATA	RDATA as seen by the master bridge.
<fast_tap>	Indicates whether the request is traveling through Fast Tap.

4.7.2.2.2 AMBA NoC End-to-End Traffic Logs for Writes

The following is an example of ns_noc_acemstrbrdg_<bridge_id>_aw_w_b.log:

<sim_time> : request_id : 0x100000002 : split_cnt : 0 : split_size : 1024 : master_id : 0x0 : slave_id : 0x1 : AW_sent_time : 445 : W_sent_time : 475 : unknown_dest : 0 : original_master_bridge_id : 0x0 : seq_num: 0x1 : mprt_AWADDR : 0x10 : mprt_AWADDR_to_slv : 0x10 sys_address : 0x10 : mprt_AWID : 0x21 : mprt_AWLEN : 0x1 : mprt_AWSIZE : 0x3 : mprt_AWBURST : 0x1 : mprt_AWCACHE : 0x3 : mprt_AWPROT : 0x5 : mprt_AWQOS : 0x8 : mprt_AWLOCK : 0x0 : mprt_AWUSER : 0x0 : mprt_AWSNOOP : 0x0 : mprt_AWDOMAIN : 0x3 : mprt_AWBAR : 0x0 : mprt_AWREGION : 0x0 : mprt_AWUNIQUE : 0x0 : slave_region : 0x0 : mprt_WDATA_CL :

0x38302820181008001c1814100c0804000000000000000000000000000000000000 : mprt_WSTRB_CL : 0xffff0000 : mprt_WUSER_CL : 0x0

<sim_time> : request_id : 0x100000002 : split_cnt : 0 : split_size : 1024 : master_id : 0x0 : slave_id : 0x1 : B_received_time : 605 : bid : 0x21 : bresp : 0x0 : buser : 0x0 : derived_mprt_AWADDR : 0x10 :derived_mprt_AWADDR_to_slv : 0x10 : sys_address : 0x10 : derived_sys_AWID : 0x21, seq_num : 0x1

Table 20 Nomenclature of ns_noc_acemstrbrdg_<bridge_id>_aw_w_b.log

Field name	Description
<sim_time>	Simulation time when master bridge traffic collected and printed.
<request_id>	A unique id associated with the request (AW/W and B) in the master and slave bridge logs.
<master_id>	The bridge ID of the master bridge that sent the AR request, or received R response.

<split_cnt>	Each ingress request has a unique <request_id>. When a request splits into multiple transactions based on <split_size> alignment, each transaction is shown with the same request_id but incrementing <split_cnt>.
<split_size>	The address alignment in bytes that ingress requests split on.
<slave_id>	The bridge ID of the slave bridge that received the AR request or sent the R response.
<AW_sent_time>	Simulation time when the AW request was sent by the master bridge.
<W_sent_time>	Simulation time when the W request was sent by the master bridge.
<unknown_dest>	Whether the request has a known destination. If unknown_dest is 1, the request failed address look-up and would get a decode error in its R response.
<original_master_bridge_id>	The bridge ID of the master bridge that sent the AW request originally. In multi-hop request, this ID represents the bridge ID of the master bridge in the first hop.
<seq_num>	The internal NoC sequence number, used for tracking ordering.
<mprt_AW*>	Fields of AW channel as seen by the master bridge.
<mprt_AWADDR_to_slv>	AW Address sent to the slave. In the case of address range relocation, mprt_AWADDR represents AWADDR sent to the master bridge and mprt_AWADDR_to_slv represents the relocated address

<sys_address>	This address represents unique system address.
<AWREGION>	Region field for the AW request assigned by the NoC. This is meaningful for slaves that use AWREGION for decoding address.
<mprt_W*>	Fields of W channel as seen by the master bridge.
<mprt_WDATA_CL>	WDATA displayed as 512-bit write data, aligned to 64B address.
<mprt_WSTRB_CL>	WSTRB displayed as 64-bit write strobe, matching the 512-bit mprt_WDATA_CL, with each bit indicating whether the corresponding WDATA_CL byte is valid.
<mprt_WUSER_CL>	WUSER packed into 512-bit, with 8 bits per byte of data in mprt_WDATA_CL.
<B_received_time>	Simulation time when B response was received by the master bridge.
<b*>	Fields of B channel as seen by the master bridge.
<derived_mprt_AWADDR>	The AWADDR that corresponds to the B response. This value is internally calculated and displayed along with each B response for ease of traffic association.
<derived_sys_AWID>	The system AWID shows the full AWID for the request. If the slave bridge has a narrower AWID width than the master bridge, it may see a truncated version of the original AWID sent by the master bridge. The system AWID is printed here for ease of traffic association.

The slave bridge logs, ns_noc_aceslvbrdg_<bridge_id>_ar_r.log and ns_noc_aceslvbrdg_<bridge_id>_aw_w_b.log, follow the same format as the master bridge logs above.

4.7.2.3 AMBA NoC End-to-End Request ID

The AMBA NoC End-to-End Checker provides a mechanism to allow the user to track requests from the time they enter the NoC to when they exit the NoC. Each transaction that enters the NoC is assigned a unique request ID. Each transaction that leaves the NoC will have a request_id that corresponds to the original request. The same request ID is used for corresponding requests and responses (i.e. AR and R, or AWW and B). The AMBA NoC End-to-End Checker reports transaction information including request ID, bridge ID and request type to one of the following SystemVerilog mailboxes depending on the type of request:

1. For requests entering the NoC: `NS_E2E_CHECKER_TOP.ns_transaction_src_mbox
2. For requests leaving the NoC: `NS_E2E_CHECKER_TOP.ns_transaction_dst_mbox

By polling these mailboxes, the user can track requests that enter and exit the NoC.

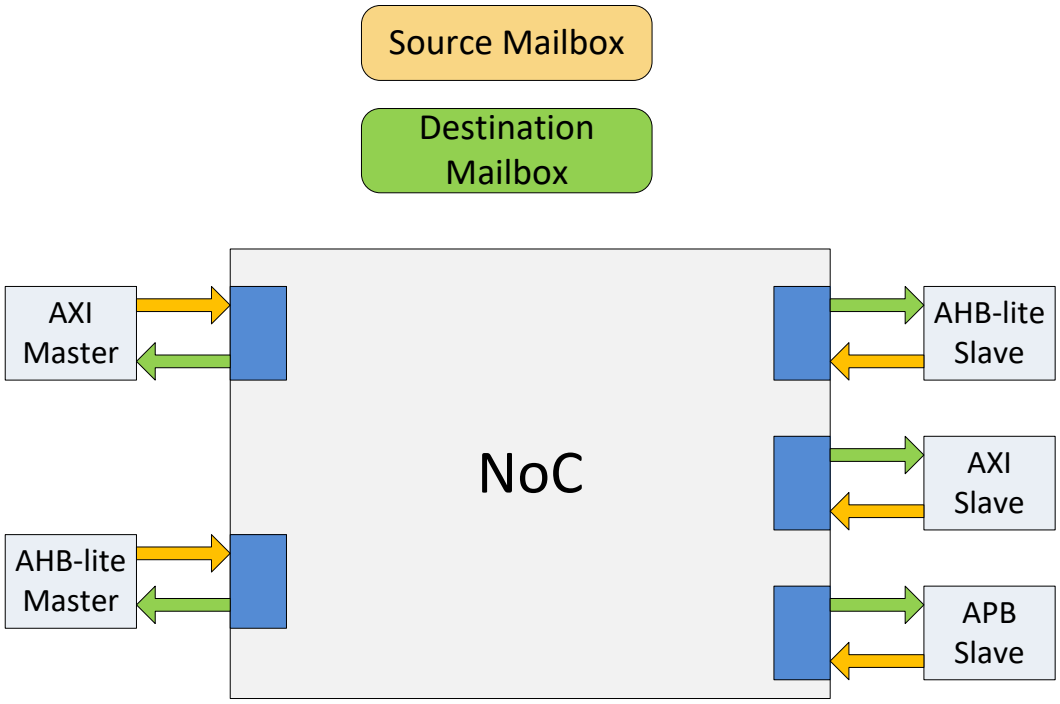

Figure 5 AMBA NoC end-to-end checking architecture

The mailboxes use the ns_transaction_info SystemVerilog struct type. The following table describes the fields of the ns_transaction_info data structure:

Table 21 ns_transaction_info struct

Field name	Description
bridge_id	The bridge ID of the immediate single-hop master or slave bridge that sent the request.

original_master_id	The bridge ID of the original master bridge that sent the request, preserved across multiple hops of the NoC, if original master ID propagation is enabled in the NocStudio command script.
request_id	A unique id associated with the request (AR and R, or AWW and B) in the master and slave bridge logs.
request_type	One of the following request types as defined in ns_mailbox_defines.vh : `define NS_REQUEST_TYPE_AR 0 `define NS_REQUEST_TYPE_AW 1 `define NS_REQUEST_TYPE_R 2 `define NS_REQUEST_TYPE_B 3 `define NS_REQUEST_TYPE_AHB_NONPOSTED_WRITE 4 `define NS_REQUEST_TYPE_AHB_POSTED_WRITE 5 `define NS_REQUEST_TYPE_AHB_READ 6 `define NS_REQUEST_TYPE_AHB_NONPOSTED_WRITE_RESP 7 `define NS_REQUEST_TYPE_AHB_RD_RESP 8 `define NS_REQUEST_TYPE_APB_READ 9 `define NS_REQUEST_TYPE_APB_WRITE 10 `define NS_REQUEST_TYPE_APB_READ_RESP 11 `define NS_REQUEST_TYPE_APB_WRITE_RESP 12

To use the AMBA NoC End-to-End mailbox mechanism:

- Add the following file to the file list. This file contains the definition of the ns_transaction_info module.

noc_verif_ip/ns_amba_struct.sv

- In the testbench, declare two instances of ns_transaction_info, for example:

ns_transaction_info ns_src_transaction;

ns_transaction_info ns_dst_transaction;

- Add code to poll transaction information from the AMBA NoC end-to-end mailboxes as needed, for example:

```
//=====================================================
// Set proj_tag to one of the following,
//   - The project name if 'tag_project_name' is used in NocStudio script.
//   - Empty string if 'tag_project_name' is used instead
//=====================================================
string proj_tag = "";
//=====================================================
// Example code to poll request id from E2E mailboxes.
//=====================================================
if(`NS_NOC_MAILBOX_LOG == 1) begin
    if(`NS_E2E_CHECKER_TOP.ns_transaction_src_mbox[proj_tag].try_get(ns_src_transaction))
begin
        $display("%t NS SRC MAILBOX: bridge_id=0x%0x,
                    request_id=0x%0x,
                    request_type=0x%0x",
                    $time,
                        ns_src_transaction.bridge_id,
                        ns_src_transaction.request_id,
                        ns_src_transaction.request_type);
    end
      if(`NS_E2E_CHECKER_TOP.ns_transaction_dst_mbox[proj_tag].try_get(ns_dst_transaction))
begin
        $display("%t NS DST MAILBOX: bridge_id=0x%0x,
                    request_id=0x%0x,
                    request_type=0x%0x",
                    $time,
                        ns_dst_transaction.bridge_id,
                        ns_dst_transaction.request_id,
                        ns_dst_transaction.request_type);
    end
end
```

4.7.3 AMBA Bridge Checkers

The AMBA bridge checkers are responsible for monitoring AMBA bridge RTL during simulation. Each instance of AMBA Master Bridge and AMBA Slave Bridge RTL has a corresponding AMBA bridge checker monitoring its behavior. Each bridge has a read and read-response checker, and a write and write-response checker. Additionally, based on whether the NocStudio configuration file has the read reordering option enabled in any of the master bridges, the read reordering checker bind is present in the generated checkers binds file. The same holds true for the write reordering option. Another option in the NocStudio configuration file are whether a slave supports read response data interleaving. If any slave supports read response data interleaving, the data interleaving checker bind is present in the generated checkers binds file. If any of the bridges are specified as asynchronous in the NocStudio configuration file, the asynchronous FIFO checker bind is present in the generated checkers binds file.

The AMBA bridge checkers perform micro-architectural checks to ensure functional correctness of the AMBA 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 AMBA bridge RTL is in a proper idle state.

The following table describes the checks performed by the AMBA bridge checkers. Violation of any check triggers an error in simulation.

Table 22 AMBA bridge checks

Description of check	Instantiated (per bridge or interface)	Type of check
ARVALID, AWVALID, WVALID, RVALID, BVALID are low when in reset.	AMBA Master Bridge	Protocol
ARVALID, AWVALID, WVALID, RVALID, BVALID are never x or z.	AMBA Master Bridge	Protocol
AR, AW, W, R, and B control signals must not be x or z when corresponding VALID and READY are high.	AMBA Master Bridge	Protocol
Reset high for at least 16 cycles. If the bridge is an asynchronous bridge, this check makes sure that reset is high for at least 16 clock cycles of the slower clock.	AMBA Master Bridge	Protocol

Narrow transfers on AR and AW interfaces are not permitted when support for narrow transfers is disabled on the bridge.	AMBA Master Bridge	Protocol
WRAP bursts on AR and AW interfaces must have total transfer data size of 16, 32 or 64 bytes. All other payload sizes are currently not supported.	AMBA Master Bridge	Unsupported
INCR burst total transfer data size must be no more than the maximum size (4KB).	AMBA Master Bridge	Unsupported
In Orion AMBA mode, coherent, cache maintenance and DVM commands are not permitted.	AMBA Master Bridge	Unsupported
ARREADY, AWREADY low when in reset.	AMBA Master Bridge	Functional
RDATA is never interleaved across different RIDs.	AMBA Master Bridge	Functional
Internal RTL micro-architectural checks.	AMBA Master Bridge	Functional
WDATA is never interleaved across different WIDs.	AMBA Master Bridge	Functional
WID must match AWID for same request.	AMBA Master Bridge	Functional
ARVALID, AWVALID, WVALID, RVALID, BVALID are low when in reset.	AMBA Slave Bridge	Protocol
ARVALID, AWVALID, WVALID, RVALID, BVALID are never x or z.	AMBA Slave Bridge	Protocol
AR, AW, W, R, B control signals must not be x or z when corresponding VALID and READY are high.	AMBA Slave Bridge	Protocol
Reset high for at least 16 cycles. If the bridge is an asynchronous bridge, this check makes sure that reset is high for at least 16 clock cycles of the slower clock.	AMBA Slave Bridge	Protocol
RREADY and BREADY are low when in reset.	AMBA Slave Bridge	Functional

RDATA is not interleaved across different RIDs if RDATA de-interleaving logic is not enabled.	AMBA Slave Bridge	Functional
Internal RTL micro-architectural checks.	AMBA Slave Bridge	Functional
All bridge RTL FIFOs are empty.	AMBA Master Bridge	Exit
All bridge RTL tracking tables are empty.	AMBA Master Bridge	Exit
All bridge RTL buffers are freed up.	AMBA Master Bridge	Exit
ARVALID, AWVALID, WVALID, RVALID, BVALID are low. Internal NoC valid signals are low.	AMBA Master Bridge	Exit
All bridge RTL FIFOs are empty.	AMBA Slave Bridge	Exit
All bridge RTL tracking tables are empty.	AMBA Slave Bridge	Exit
All bridge RTL buffers are freed up.	AMBA Slave Bridge	Exit
ARVALID, AWVALID, WVALID, RVALID, BVALID are low. Internal NoC valid signals are low.	AMBA Slave Bridge	Exit

For a subset of the above checks, fine-grained user control is provided to individually enable or disable the checks. For each check listed in the following table, setting the corresponding `define to 1 disables the check; setting it to 0 enables 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 23 Fine-grained user-control of AMBA bridge checks

Description of check	`define to control	Default value
No unknown (x or z) data packets.	`NS_STRBRDG_DATA_XZ_CHECK_DISABLE	0
Narrow transfers on AR and AW interfaces are not permitted when support for narrow	`NS_ACEBRDG_NARROWS_CHECK_DISABLE	0

transfers is disabled on the bridge.		
Set to 0 to inhibit some checks while injecting errors for SAFETY feature validation.	`NS_SAFETY_CHECKALL_EN	1

4.7.4 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 24 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	Regbus Ring Master	Functional

content and ordering within each Regbus Ring Master.		
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 25 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 26 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.

4.7.5 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 27 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

4.7.6 Clock Control Signal Checks

Table 28 Clock control signal checks

Description of check	Instantiated (per bridge or interface)	Type of check
scan_mode pin can toggle only when AMBA master bridge is idle.	AMBA Master Bridge	Functional

system_cg_or pin can toggle only when AMBA master bridge is idle.	AMBA Master Bridge	Functional
system_clk_en pin can toggle only when AMBA master bridge is idle.	AMBA Master Bridge	Functional
scan_mode pin can toggle only when AMBA slave bridge is idle.	AMBA Slave Bridge	Functional
system_cg_or pin can toggle only when AMBA slave bridge is idle.	AMBA Slave Bridge	Functional
system_clk_en pin can toggle only when AMBA slave bridge is idle.	AMBA Slave Bridge	Functional

4.7.7 Link Clock Cross FIFO Checker

The Link Clock Cross FIFO Checker is responsible for monitoring and ensuring functional correctness of the link clock cross FIFO RTL during simulation. The following checks are performed.

Table 29 Link Clock Cross FIFO checks

Description of check	Instantiated (per bridge or interface)	Type of check
Arbitration must only select a vc that has data, credit, and a downstream link available.	Link Clock Cross FIFO	Functional
Arbitration must grant to a vc that has valid data, credits, and downstream link available to maintain full throughput.	Link Clock Cross FIFO	Functional
Grant must be one hot or zero.	Link Clock Cross FIFO	Functional
system_cg_or pin can toggle only when ACE master bridge is idle.	Link Clock Cross FIFO	Functional
At any given arbitration must follow the priority architecture.	Link Clock Cross FIFO	Functional
No credit overflow or underflow.	Link Clock Cross FIFO	Functional
No X or Z on credit signals.	Link Clock Cross FIFO	Functional

Flit entering link clock cross FIFO on write side passes through on link from read side.	Link Clock Cross FIFO	Functional
Outputs low during reset.	Link Clock Cross FIFO	Functional
Credit counts returned to reset value.	Link Clock Cross FIFO	Exit

4.7.8 Asynchronous FIFO Checker

The Asynchronous FIFO Checker is responsible for monitoring and ensuring functional correctness of the Asynchronous FIFO RTL during simulation. The following checks are performed.

Table 30 Asynchronous FIFO checks

Description of check	Instantiated (per bridge or interface)	Type of check
Capacity checks: FIFO occupancy >=0 and <= fifo capacity	Voltage Domain Crossing FIFO Link Clock Cross FIFO	Functional
Setup and Hold Checks: Register array values must be stable for one read clock before and after read cycle	Voltage Domain Crossing FIFO Link Clock Cross FIFO	Functional
Read Valid: Register array entry must be valid at the time of read, cannot be read more than once	Voltage Domain Crossing FIFO Link Clock Cross FIFO	Functional

4.7.9 LLC Checker

The LLC checker tracks behavior of LLC RTL during simulation. The LLC checker is present if Pegasus IP is instantiated.

Table 31 LLC checker

Description of Check	Type of Check
No X or Z on credit signals.	Functional

No credit overflow or underflow.	Functional
No X or Z on registers when out of reset.	Functional
Credit counts returned to reset value.	Exit

4.7.10 LLC Performance and Debug features

When Pegasus IP is instantiated, the Pegasus mailbox provides user visibility of the content of each LLC for gathering performance statistics and ease of debugging. By polling this mailbox, the user can track the LLC content as lines are allocated, evicted and updated.

The Pegasus mailbox uses the ns_pegasus_transaction_info SystemVerilog struct type. The following table describes the fields of the ns_pegasus_transaction_info data structure:

Table 32 LLC mailbox request types encoding

Field name	Description
bridge_id	The bridge ID of the master bridge of each LLC instance.
request_type	One of the following request types as defined in ns_mailbox_defines.vh : `define NS_PEGASUS_REQUEST_TYPE_ALLOCATE_LINE 8'h30 `define NS_PEGASUS_REQUEST_TYPE_UPDATE_TAG 8'h31 `define NS_PEGASUS_REQUEST_TYPE_EVICT_LINE 8'h32 `define NS_PEGASUS_REQUEST_TYPE_INVALIDATE_LINE 8'h33
addr	Address of the cache line.
valid	Valid bit of the cache line.
dirty	Dirty bit of the cache line.
ns_bit	Non-Secure bit of the cache line, corresponding to AxProt[1].

To use the Pegasus mailbox mechanism:

- In the testbench, declare an instance of ns_pegasus_transaction_info, for example:

```
ns_pegasus_transaction_info ns_pegasus_transaction;
```

- Add code to poll transaction information from the mailbox as needed, for example:

```
//=============================================================
// Set proj_tag to one of the following,
//   - The project_name if 'tag_project_name' is used in NocStudio script.
//   - Empty string if 'tag_project_name' is not used.
//=============================================================
string proj_tag = "";

//=============================================================
// Define string names for readable encoding (Optional)
//=============================================================
string mbox_request_type_num2str[int];

initial begin

    mbox_request_type_num2str[`NS_PEGASUS_REQUEST_TYPE_ALLOCATE_LINE]  =
        "NS_PEGASUS_REQUEST_TYPE_ALLOCATE_LINE";
    mbox_request_type_num2str[`NS_PEGASUS_REQUEST_TYPE_UPDATE_TAG]     =
        "NS_PEGASUS_REQUEST_TYPE_UPDATE_TAG";
    mbox_request_type_num2str[`NS_PEGASUS_REQUEST_TYPE_EVICT_LINE]     =
        "NS_PEGASUS_REQUEST_TYPE_EVICT_LINE";
    mbox_request_type_num2str[`NS_PEGASUS_REQUEST_TYPE_INVALIDATE_LINE] =
        "NS_PEGASUS_REQUEST_TYPE_INVALIDATE_LINE";

end

//=============================================================
// Example code to poll Pegasus mailbox
//=============================================================
if (`NS_PEGASUS_MAILBOX_LOG == 1) begin
        if(`NS_E2E_CHECKER_TOP.ns_pegasus_transaction_mbox[proj_tag].try_get(ns_
    pegasus_transaction)) begin
        $display ("%t NS PEGASUS MAILBOX: bridge_id=0x%0x,
            addr=0x%0x,
```

```
                valid=%0b,
                ns_bit=%0b,
            dirty=%0b,
                request_type=0x%0x(%s)",
                $time,
                ns_pegasus_transaction.bridge_id,
                ns_pegasus_transaction.addr,
             ns_pegasus_transaction.valid
                ns_pegasus_transaction.ns_bit,
             ns_pegasus_transaction.dirty,
                ns_pegasus_transaction.request_type,
                mbox_request_type_num2str[ns_pegasus_transaction.request_type]);
          end
end
```

4.7.11 LLC Preloading for Fast Initialization

When Pegasus is instantiated, the user can perform zero-time preloading of LLCs for fast backdoor initialization in RTL simulation by calling `preload_pegasus`, a Verilog task with the following syntax:

Table 33 Syntax of preload_pegasus task

Field type	Field name	Description
input	cache_mode	LLC mode for the preload operation. Set to 1'b1 to indicate cache mode, and 1'b0 to indicate ram mode.
input	ns_bit	Non-secure bit of the cache line, corresponding to AxProt[1]. Set to 1'b1 to indicate line is non-secure, and 1'b0 to indicate line is secure.
input[63:0]	addr	Address of the cache line.
input	dirty	Set to 1'b1 if line should be preloaded as dirty, 1'b0 if line is clean.
input[511:0]	data	Full cache line worth of data. All 64B will be preloaded.
input	abort_on_error	When set to 1'b1, simulation aborts when preload_status indicates failure. This mode can be used if the user expects the preload command to succeed and prefers simulation to abort if the operation fails for unexpected reasons.

		When set to 1'b0, simulation continues when preload_status indicates failure while reporting the failure with NS_INFO messages. This mode can be used if the user does not guarantee that the current preload command will succeed and prefers to simulate all the preload commands to gather failure status reported by the simulation.
output	preload_status	Pass or fail status for the preload command. 1'b1 indicates pass, and 1'b0 indicates failure. Whether simulation aborts upon a failed preload status depends on the setting of abort_on_error.

As part of the Pegasus preload mechanism, the following checks are performed before a cache line can be preloaded successfully:

Table 34 Checks performed for Preloading Pegasus

Check name	Description
PRELOAD_PEGASUS_CACHE_LINE_ALIGNMENT	Address must be cache-line aligned. No partial-line preload is allowed.
PRELOAD_PEGASUS_DUPLICATE_ENTRY	User must not preload the same address twice.
PRELOAD_PEGASUS_NO_VALID_DESTINATION_LLC	No valid destination LLC found. This could be due to: • Mismatch of cache mode versus RAM mode. • Address mismatch.
PRELOAD_PEGASUS_FOUND_NO_ROOM	No room to allocate in LLC (after finding a valid destination LLC). This could be due to: If preloading in cache mode: • Waygroup is not in the right mode. LLC is either disabled or in RAM mode. • Waygroup found in the right mode, but all 4 ways are full. If preloading in RAM mode: • Waygroup is not in the right mode. LLC is either

	disabled or in cache mode. • ns_bit doesn't match waygroup property (specified by NocStudio prop) in RAM mode.

The Pegasus preloading mechanism is a Verilog task, `preload_pegasus`. To use it the user must:

- Wait for the relevant LLC to come out of reset.
- Start zero-time preloading by calling `preload_pegasus` for every line that needs to be preloaded.

Example code as follows:

```
localparam CACHE_MODE = 1;
localparam RAM_MODE = 0;
reg preload_status;

initial begin // {
  // Wait for llc llc_0_0 to come out of reset
  wait(~`NS_NOC_TOP.u_ns_fabric.u_ns_llc_llc_0_0.u_ns_llc.reset);
  // Wait for SHORT_INIT to be done which happens at de-assertion of llc reset
  @(posedge `NS_NOC_TOP.u_ns_fabric.u_ns_llc_llc_0_0.u_ns_llc.clk);
  @(posedge `NS_NOC_TOP.u_ns_fabric.u_ns_llc_llc_0_0.u_ns_llc.clk);
  `NS_NOC_TOP.ns_preloader.preload_pegasus(CACHE_MODE, 1'b0, 32'h20000000, 1'b0,
          512'h6d1dafb9d60623ed40c0921a161cfd3990a8cb9d9a705c2b68e771d549dda28d5a04f
          8a03b2007685fa0186ff69716af29b4cf709e99fc2c594386449f5a391f, 1'b1,
          preload_status);
  `NS_NOC_TOP.ns_preloader.preload_pegasus(CACHE_MODE, 1'b1, 32'h20000000, 1'b1,
          512'h7da483691d803522123fde3cbffd7f2b3951cfab8cd79173dedbbb8d849d0a339cde3
          38d9c1761a18401455ea8a50b0b7b96cb6e55f7bfb56980518f81ea1a79, 1'b1,
          preload_status);
end //}
```

If address map is re-programmed through regbus to set base, mask or hash functions to different values, call get_reprog_addr_map task to get the newly programmed values before calling

preload tasks. get_reprog_addr_map() gets the base,mask,hash function values from rtl at the time of the call.

Sample code is shown below

```
<After the address map is reprogrammed call the task as shown below followed by normal
        preloading calls>
 `NS_NOC_TOP.ns_preloader.get_reprog_addr_map();
 `NS_NOC_TOP.ns_preloader.preload_pegasus(CACHE_MODE, 1'b1, 32'h20000000, 1'b1,
        512'h7da483691d803522123fde3cbffd7f2b3951cfab8cd79173dedbbb8d849d0a339cde3
        38d9c1761a18401455ea8a50b0b7b96cb6e55f7bfb56980518f81ea1a79, 1'b1,
        preload_status);
```

4.7.12 SRAM checker

The SRAM checker binds to every instance of LLC RAM. It tracks each write to RAM, and then checks the data correctness of every read from RAM.

Table 35 SRAM checker

Description of check	Instantiated	Type of check
Actual data read from RAM matches expected data.	LLC RAM	Functional

4.7.13 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 36 CSR checks

Description of check	Instantiated	Type of check
Bridge registers are not X or Z when out of reset.	AMBA Master Bridge	Functional
Bridge registers are not X or Z when out of reset.	AMBA Slave Bridge	Functional

5 LOW POWER INTEGRATION OVERVIEW

5.1 NOC IP COMPONENTS

This section describes additional content generated for a low power enabled NoC.

- NocStudio LP usage examples
- User manuals and documentation

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

- NoC LP aware RTL
- NoC CPFs
- NoC UPFs
- NoC LP verification checkers

5.2 DIRECTORY STRUCTURE

Table 37 Low Power NoC IP directory structure

Name	Description
noc_verif_ip/*_lp_checker.sv	NoC verification library for LP

5.3 NOCSTUDIO FLOW TO GENERATE LOW POWER NOC IP

This section describes the Low Power NoC IP generation flow using NocStudio. The user specifies a NocStudio command script that describes the user system requirements. The following files are generated by NocStudio for a Low Power NoC:

- NoC RTL
- NoC CPFs
- NoC UPFs
- 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.

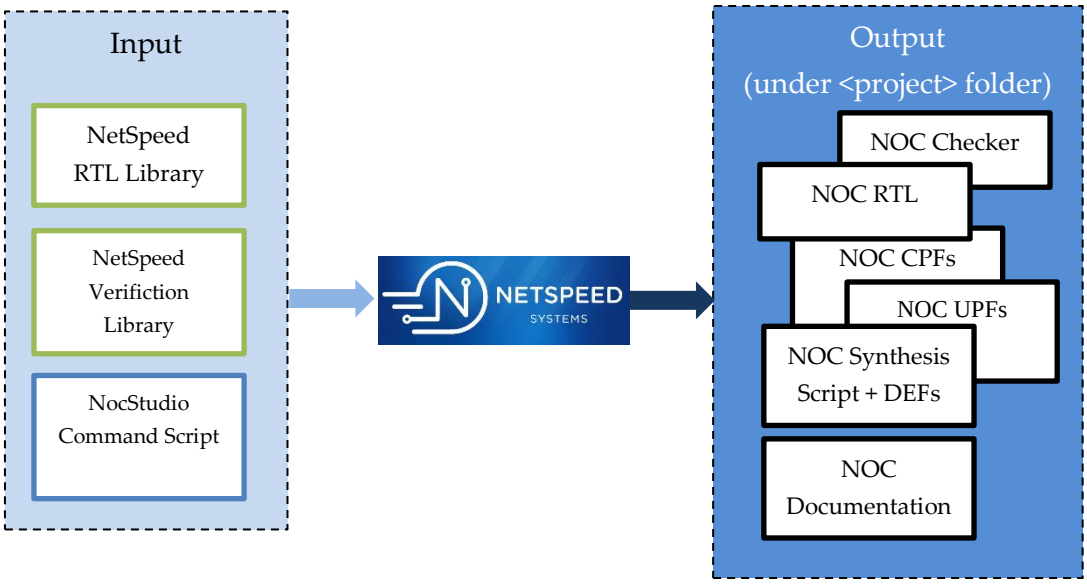

Figure 6 Low Power NoC IP generation flow

5.3.1 Generating RTL, CPFs and UPFs from NocStudio

To generate NoC RTL, CPF and UPF files, include gen_ip command at the end of the NocStudio command script, and then process the script with NocStudio. Once the gen_ip command executes, a project directory is created which contains all the files and directories generated by NocStudio. Below is a list of additional files (CPF, UPF and verification IP) that are generated for a low power enabled configuration. For a complete list with detailed descriptions, please refer to CFG NocStudio User Manual.

Table 38 Key low power sanity testbench files generated by NocStudio in project directory

Name	Description	Type
ns_power_map_table.sv	Support file for AXI NoC LP functionality Checker generated for the NoC by NocStudio.	Verification
run_test.sh	Run command to launch LP sanity bench for the generated NoC using Cadence Incisive simulator.	Sanity testbench
cpfs/*	NoC hierarchical CPF files	CPF
cpfs/ns_soc_ip.cpf	NoC design top cpf file having isolation rules and power modes defined based on user	CPF

	power intent specification in command script file	
cpfs/top.cpf	Testbench cpf for low power simulations which instantiates NoC design top CPF file, ns_soc_ip.cpf	CPF
upf/1.0/	NoC hierarchical UPF files	UPF
upf/1.0/ns_soc_ip.upf	NoC design top upf file having isolation rules and power modes defined based on user power intent specification in command script file	UPF
upf/1.0/top.upf	Testbench upf for low power simulations which instantiates NoC design top UPF file, ns_soc_ip.upf. ***(Still in Development)***	UPF

5.3.2 Low Power NoC Sanity Testbench

NocStudio generates the following files in a low power configuration project directory:

Common Power Format (CPF) files. These are located in the cpfs/ directory. They are version 1.1 compatible and are generated in a hierarchical method from power domain group level to NoC top. The low power sanity testbench top file, top.cpf, instantiates the NoC design top file, ns_soc_ip.cpf.

Unified Power Format (UPF) files. These are located in the upf/1.0/ directory. They are version 1.1 compatible and are generated in a hierarchical method from power domain group level to NoC top. The low power sanity testbench for UPF model is still in development.

(a) Low Power Verification Checker files. These are located in the noc_verif_ip/ directory. They verify the low power functionality.

To run the Low Power NoC sanity test, change to the project directory and invoke the following run script (only Cadence Incisive Simulator is supported):

```
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:

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

The waveform database will be generated inside the simulation trace/ or trace_regbus/ directory. On a successful compile and simulation, the following will appear in the log file:

```
Passing to irun for build ns_soc_ip
BUILD SOC SUCCESSFUL
Passing to irun for build
NON LOW POWER BUILD SUCCESSFUL
Passing to irun for build
LOW POWER BUILD SUCCESSFUL
Passing to irun for simulation
REGBUS SIMULATION PASSED
Passing to irun for simulation
SIMULATION FAILED
```

After the completion of simulation run, the presence of a file named SIM_FAILED indicates a failure in the simulation run. The presence of SIM_PASSED in the project directory indicates a successful simulation run. The log file run_test_incisiv.log will list any errors encountered during the build and simulation phase. Additional information for the build is recorded in build.log, located in the model/ directory. Additional information for the simulation run is recorded in run.log and/or regbus_run.log, located in the trace/ and trace_regbus/ directories, respectively. With a successful simulation from the low power NoC sanity testbench, the generated NoC RTL and Low Power Verification IP are ready to be integrated into the user's verification environment.

The IP Integration Specification describes the general NoC integration process. This section highlights differences and additional considerations when integrating a low power enabled NoC.

5.4 INTEGRATION OF NOC RTL

5.4.1 Hierarchical RTL generation

For a low power enabled configuration, NocStudio creates a new layer of hierarchy in the RTL that groups elements by power domain. Elements are placed inside a module that is named by the power domain to which they belong (e.g. elements belonging to the system power domain exist in the system module). These power domain modules are instantiated as the first level of hierarchy inside ns_fabric. Any optional user specified RTL grouping applies below the power domain RTL module boundary. Where user defined RTL groups cross power domain boundaries, the user defined group is implemented within each power domain module.

Within each power domain RTL group, a *nsps_grp* may be created which contains all CFG Power Supervisor logic. This is always present in the system power domain group, where it contains the state machines that implement the Q-channel interfaces. In other power domain groups, *nsps_grp*, may or may not be present, where if necessary, it will contain small amounts of local signal aggregation logic. All interface signals of each *nsps_grp* is treated as asynchronous to allow for relaxed timing constraints for the long distances these signals may need to travel, and within the RTL, the signal names include the string "_async_" to highlight this fact.

The position of each *nsps_grp* module may be controlled via the –nsps_pos argument of the *add_power_domain* command and the related *set_power_domain_nsps_pos* command. The clock used for logic within each *nsps_grp* may be controlled via the –nsps_clk argument of *add_power_domain* and the related *set_power_domain_nsps_clk* command. If these are not specified in the configuration, NocStudio will attempt to choose reasonable values during NoC construction.

5.4.2 Integration of Q-Channel interface ports

The low power interface between NoC and PMU follows the ARM Q-Channel Low Power Interface Specification. The Low Power Support chapter of the Technical Reference Manual covers details of the ports generated at NoC RTL top ns_soc_ip.v per power domain present in the NoC.

5.4.3 Reset

In a low power enabled configuration, per power domain reset signals (*reset_n_<power_domain>*) are present. Details of the behavior of these reset signals and usage requirements are described further in the Reset section of chapter 2 – Integration of NoC.

5.4.4 Clocks

For low-power enabled designs, clock pins are provided per power domain to facilitate clock gating at the power domain level, and the naming of clock pins is modified accordingly as described in the following table.

Table 39 NoC clock signals

Signal name	Description
clk__<power_domain>__<clock_domain>	Clock pins are named with the power domain and clock domain. - Each clock pin is connected to one power domain. - If a clock domain spans multiple power domains, each power domain will have a separate clock pin for that clock domain

5.5 INTEGRATION OF CPF FILES

To instantiate NoC CPFs in the user environment:

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

```
setenv NS_CPF_DIR /absolute/path/of/project/cpfs/created/
```

- Include the following lines in the ns_soc_ip parent hierarchy CPF file to instantiate NoC CPFs. All the isolation and power switch control ports per power domain referred to are virtual and need to be mapped to real controls as described below:

```
set_instance ns_soc_ip0 -port_mapping {
        {<virtual port mapping of power domain (0) isolation control>}
        {< virtual port mapping of power domain (1) isolation control >}
        {< virtual port mapping of power domain (n) isolation control >}

        {<virtual port mapping of power domain (0) switch control>}
        {<virtual port mapping of power domain (1) switch control >}
        {<virtual port mapping of power domain (n) switch control >}
}
source $::env(NS_CPF_DIR)/ns_soc_ip.cpf
```

  The top-level design is ns_soc_ip, specified in ns_soc_ip.cpf.

5.5.1 Hierarchical CPF generation

NocStudio creates cpf for each power domain group. These modules are hierarchically instantiated in the design top level cpf file, ns_soc_ip.cpf.

5.6 INTEGRATION OF UPF FILES

To instantiate NoC UPFs in the user environment:

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

```
setenv NS_UPF_DIR /absolute/path/of/project/upfs/created/
```

- Include the following lines in the ns_soc_ip parent hierarchy UPF file to instantiate NoC UPFs. All the isolation and power switch control ports per power domain referred to are virtual and need to be mapped to real controls as described below:

```
set_instance ns_soc_ip0 -port_mapping {
        {<virtual port mapping of power domain (0) isolation control>}
        {< virtual port mapping of power domain (1) isolation control >}
        {< virtual port mapping of power domain (n) isolation control >}

        {<virtual port mapping of power domain (0) switch control>}
        {<virtual port mapping of power domain (1) switch control >}
        {<virtual port mapping of power domain (n) switch control >}
}
source $::env(NS_UPF_DIR)/ns_soc_ip.upf
```

The top-level design is ns_soc_ip, specified in ns_soc_ip.upf.

5.6.1 Hierarchical UPF generation

NocStudio creates upf for each power domain group. These modules are hierarchically instantiated in the design top level cpf file, ns_soc_ip.upf

5.7 Integration of Low Power Verification Checkers

The steps described in Section Integration of NoC Verification Checkers will automatically bind low power checkers for a low power NoC configuration.

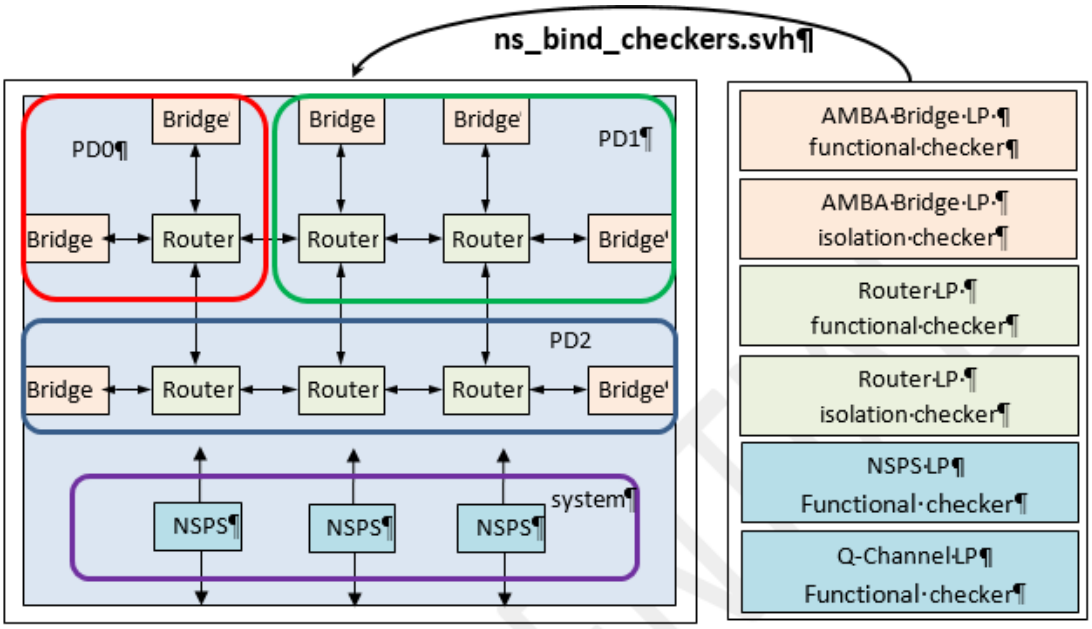

Figure 7 Low Power Checker Bind to RTL

5.8 Overview of Checkers

NocStudio provides the following low power checker files:

Table 40 CFG LP checkers

Checkers	Instantiated
Q-Channel protocol LP Checker	One instance per NSPS RTL
AMBA Master Bridge isolation LP Checker	One instance per AMBA Master bridge RTL
AMBA Slave Bridge functional LP Checker	One instance per AMBA Slave Bridge RTL
AMBA Slave Bridge isolation LP Checker	One instance per AMBA Slave Bridge RTL
AMBA Router functional LP Checker	One instance per Router RTL
AMBA Router isolation LP Checker	One instance per Router RTL
NSPS functional LP Checker	One instance per NSPS RTL
Multi-Voltage FIFO Checker	One instance per half-FIFO RTL

5.9 Environment Setup for Integration

CFG verification checkers use Verilog preprocessor macros to enable the checkers at runtime. When integrating the CFG verification components, the following macros must be defined.

Table 41 `define variables for model build

`define variable name	Description	Notes
`NS_AXI_LP_CHECKER_EN	Set to 1 to enable, 0 to disable AMBA Master Bridge, Slave Bridge and Router LP functionality Checkers, NSPS interface and Q-channel protocol checks.	Recommend mapping to a plusarg variable to allow run-time control of value.
`NS_AXI_ISO_LP_CHECKER_EN	Set to 1 to enable, 0 to disable AMBA Master Bridge, Slave Bridge and Router low power state Checkers.	Recommend mapping to a plusarg variable to allow run-time control of value.
`NS_NSPS_LP_CHECKER_EN	Set to 1 to enable, 0 to disable NSPS Element Checkers.	Recommend mapping to a plusarg variable to allow run-time control of value.
`NS_QCHANNEL_LP_CHECKER_EN	Set to 1 to enable, 0 to disable ARM Q-Channel Protocol LP Checker.	Recommend mapping to a plusarg variable to allow run-time control of value.

5.10 Usage Modes

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

Table 42 Recommended checker settings

Usage mode	Bringup Mode	Heavy Debug Mode	Code Stable Mode
`NS_AXI_LP_CHECKER_EN	1	1	0

`NS_AXI_ISO_LP_CHECKER_EN	1	1	0
`NS_NSPS_LP_CHECKER_EN	1	1	0
`NS_QCHANNEL_LP_CHECKER_EN	1	1	0

5.11 CHECKERS

5.11.1 Terminology

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

Protocol – These checks enforce adherence to ARM Q-Channel interface protocol.

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

Isolation – These checks verify that control output signals have reset state values during low power state of power domain elements.

All general and low power checkers are disabled during the power gated state for power-aware simulations. This is to avoid false assertions due to X-corruption in the power-gated state. The checker enable to these checks is dynamically controlled by detecting power gated state using Incisive system tasks $lps_enabled and $lps_link_power_domain_powerdown. As these system tasks are applicable only to Incisive NCSIM simulator, this checker uses the Incisive NCSIM tool default macro INCA to guard their use.

5.11.2 AMBA Bridge and Router LP Functionality Checker

The AMBA bridge LP functionality checkers verify that the transaction-level protocols and responses match the low-power state during the simulation. The Router LP functionality checkers are responsible for verifying flit and credit transfers in low power state. This includes checking of the CFG low power handshake protocol and timing.

The AMBA Master Bridge LP checker uses a global reference database that is generated by the AMBA NoC End-to-End Checker for verifying outstanding and pending transactions to connected elements during low power states. AMBA master bridge LP checkers are common to all types of master bridges: AXI4, AXI3, AXI4-Lite, ACE-Lite and AHB-Lite. Similarly, AMBA slave bridge LP checkers are common to AXI4, AXI3, AXI4-Lite, APB, AHB-Lite slave bridges.

The following table describes the checks performed by the AMBA Bridge and router LP checkers. Violation of any check triggers an error in simulation.

Table 43 Low Power functionality checks

Description of check	Instantiated (per bridge)	Type of check
All transactions to the slave get DECERR when the power domain(s) present in the respective link is in sleep state unless autowake is enabled	AMBA Master Bridge	Functional
The outstanding transaction count to connected slaves should be 0 when master bridge enters sleep mode and stays 0 till master bridge exits sleep mode	AMBA Master Bridge	Functional
All the outstanding transactions are drained before power domain(s) present in the respective link enters to sleep state	AMBA Master Bridge AMBA Slave Bridge	Functional
No new input and output credit valid when in sleep state	AMBA Master Bridge AMBA Slave Bridge Router	Functional
No input and output flit valid when in sleep state	AMBA Master Bridge AMBA Slave Bridge Router	Functional
Output link_available is 0 and stable when bridge is in sleep state	AMBA Master Bridge AMBA Slave Bridge Router	Functional
LP signaling protocol checks	AMBA Master Bridge AMBA Slave Bridge Router	Functional
Output auto_wake signal should always be 0 when autowake mode is disabled	AMBA Master Bridge	Functional
Output sleep_ack_n protocol check	AMBA Master Bridge AMBA Slave Bridge Router	Functional
Timeout check from connected power domain state change request to fence acknowledge	AMBA Master Bridge	Functional

Timeout check from sleep state change request to sleep acknowledge	AMBA Master Bridge AMBA Slave Bridge Router	Functional

5.11.3 NoC Element LP Isolation Checkers

The NoC Element LP isolation checkers verify that output control signals have stable reset values throughout sleep, reset and power gated states. All NoC elements, including all AMBA master and slave bridges, e.g. AXI4, AXI3, AXI4-Lite and ACE-Lite, AHB-Lite and APB, have checkers for the AMBA protocol interface to the host. In addition, these checkers also have checks on low power control output signals and all NoC element ring slave control output signals.

The following table describes the microarchitectural level checks performed by the NoC Element isolation checkers. Violation of any check triggers an error in simulation.

Table 44 Low Power isolation checks

Description of control output check (during sleep, reset and power gated states)	Instantiated (per bridge or interface)	Type of check
Low power control outputs fence_ack_n, fence_done_n, auto_wake and sleep_ack_n is low	All AMBA Master Bridges	Isolation
Low power control output sleep_ack_n is low	All AMBA Slave Bridges	Isolation
Low power control output autowake and sleep_ack_n is low	Router	Isolation
Low power control output autowake, idle_status and sleep_ack_n is low	AHB-Lite Master Bridge converter interface	Isolation
Bridge to router interface outputs flit_valid, credit_inc and link_available are low	All AMBA Bridges interface to Router	Isolation

Master bridge outputs ARREADY, AWREADY, WREADY, RVALID and BVALID are low	All AMBA AXI Master Bridges	Isolation
Slave bridge outputs ARVALID, AWVALID and WVALID are low	All AMBA AXI Slave Bridges	Isolation
AHB-Lite slave bridge HTRANS is all 0s	AHB-Lite Slave Bridge	Isolation
APB slave bridge PADDR, PPROT, PWDATA, PSTRB, PENABLE, PWRITE and PSELx are all 0s	APB Slave Bridge	Isolation
Bridge ring slave interface ring_data_out_valid, ring_credit_out and ring_wakeup_out are low	All AMBA Bridges	Isolation
Bridge interrupt is low	All AMBA Bridges	Isolation
Router outputs flit_valid, credit_inc and link_available are low	Router	Isolation
Router ring slave interface ring_data_out_valid, ring_credit_out and ring_wakeup_out are low	Router	Isolation
Router interrupt is low	Router	Isolation

5.11.4 NSPS Element Checkers

The NetSpeed Power Supervisor (NSPS) element checkers are responsible for monitoring the NSPS interface to NoC elements and verifying low power behavior during low power state transitions. Three modules are covered, each with a separate checker: the NetSpeed Power Supervisor, the NSPS Fence/Ack Proxy, and the NSPS Aggregator.

Every instance of each NSPS element has a corresponding NSPS LP checker monitoring the NetSpeed low power protocol interface.

The following table describes the microarchitectural level checks performed at NSPS RTL. Violation of any check triggers an error in simulation.

Table 45 NSPS LP checks

Description of check	Instantiated	Type of check

LP signaling protocol checks	NSPS	Functional
Q-Channel TIMEOUT	NSPS	Functional
Sleep signaling TIMEOUT	NSPS	Functional

5.11.5 ARM Q-Channel Protocol LP Checker

The ARM Q-Channel protocol LP checker is bound at the NSPS interface to the Power Management Unit (PMU) and is responsible for verifying all Q-Channel handshake rules specified in section 2.1.2 of the ARM Low Power Interface Specification.

The following table describes the Q-Channel protocol checks performed at NSPS RTL. Violation of any check triggers an error in simulation.

Table 46 ARM Q-Channel LP checks

Description of check	Instantiated	Type of check
Q-Channel handshake rules	NSPS	Protocol
Q-Channel illegal handshake state - QDENY high when QACCEPTn is low	NSPS	Protocol
Power domain reset to be asserted only during Q_STOPPED state	NSPS	Protocol

5.11.6 Multi-voltage FIFO Checker

The Multi-voltage FIFO checker is an AMBA transaction FIFO split into two halves, each in a separate voltage domain. The Multi-voltage FIFO checker is bound to each half of the FIFO. The following table describes the FIFO functional checks performed at each half FIFO. Violation of any check triggers an error in simulation.

Table 47 Multi-Voltage FIFO LP checks

Description of check	Instantiated	Type of check
NSPS Signaling power sequencing checks	Slave Half-FIFOs	Protocol
Sleep and Reset FIFO state checks	All Half-FIFOs	Functional
Sleep and Reset Transaction idle checks	All Half-FIFOs	Functional
Reset Isolation checks	All Half-FIFOs	Isolation

Gray-Code pointer checks	All Half-FIFOs	Functional
Async-FIFO checks – See Asynchronous FIFO Checker	All Half-FIFOs	Functional

5.12 SUPPORTED TOOLS

Supported 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 13.20.036
- Synthesis: Cadence Genus 15.12 – 15.10-s019_1

The current release does not support low power simulations using Synopsys VCS, nor does it support low power synthesis using Synopsys DC.

6 PHYSICAL DESIGN GUIDELINES

NocStudio, when enabled, generates RTL for a given configuration. Generated RTL can then be synthesized using the reference Synthesis Environment. Figure 8 shows the different stages of the flow from Verilog RTL generation to Synthesis.

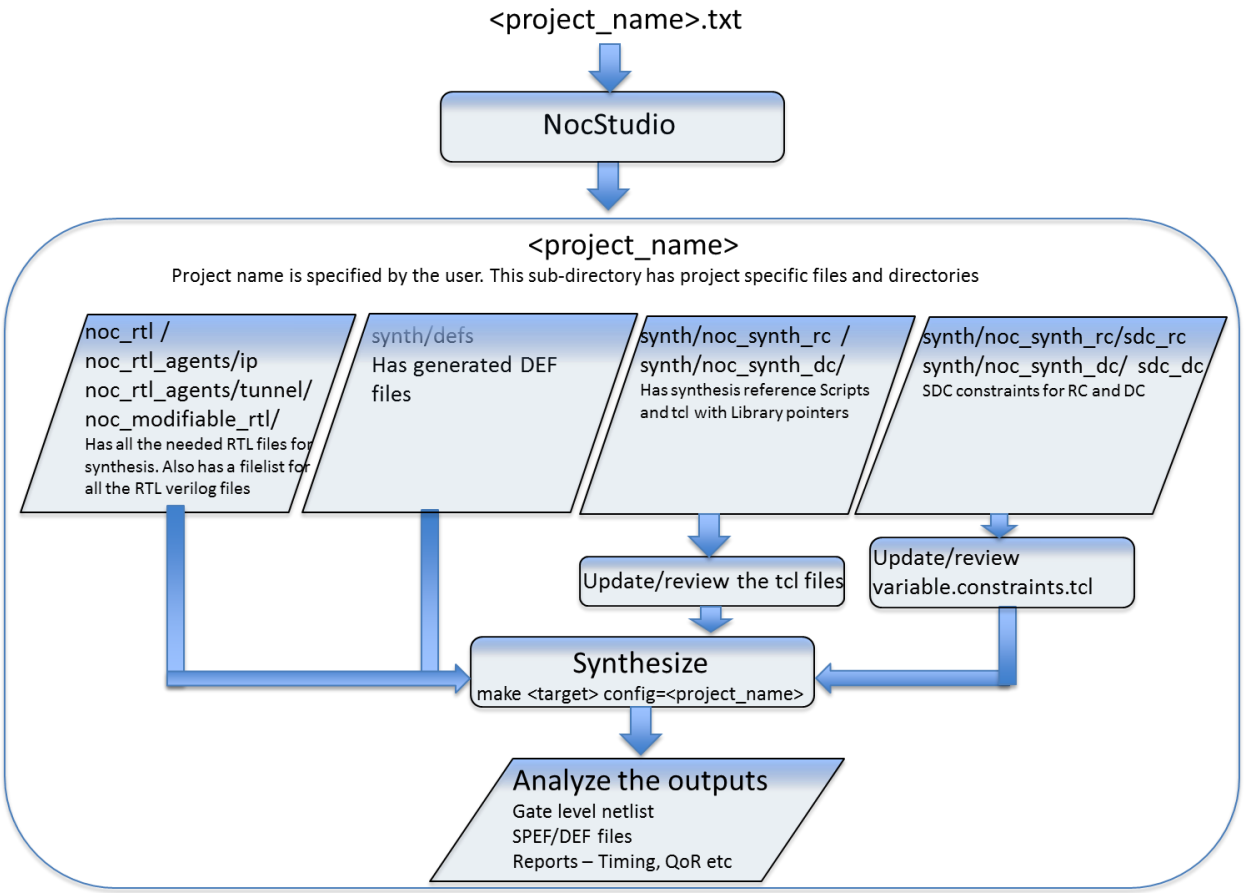

Figure 8: An example flow chart explaining steps from NocStudio to Synthesis output

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

6.1 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.

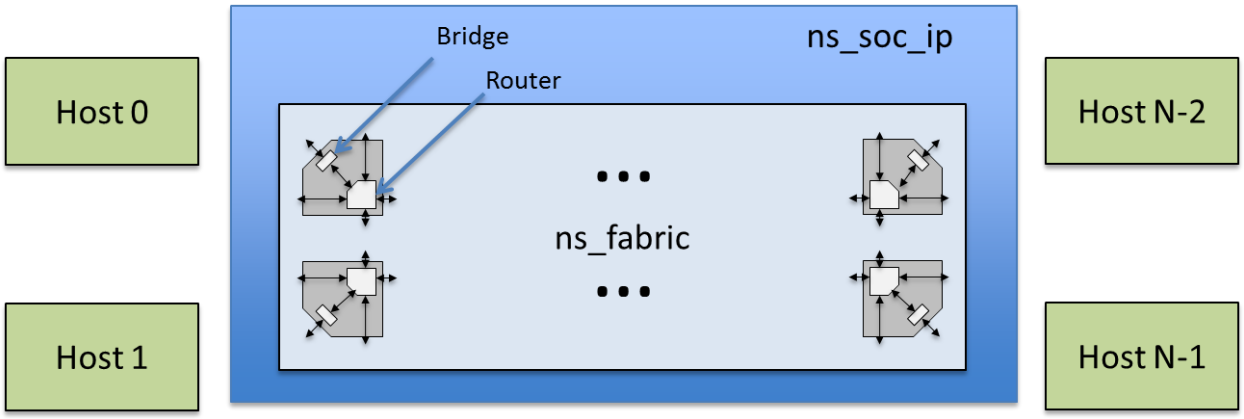

Figure 9: NoC RTL Netlist Structure

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).

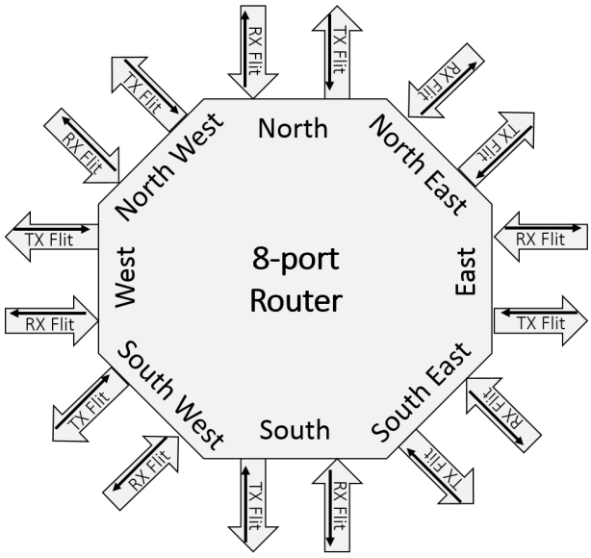

Figure 10: 8 Port Router with 4 directional ports & 4 host ports

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

6.1.1 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.

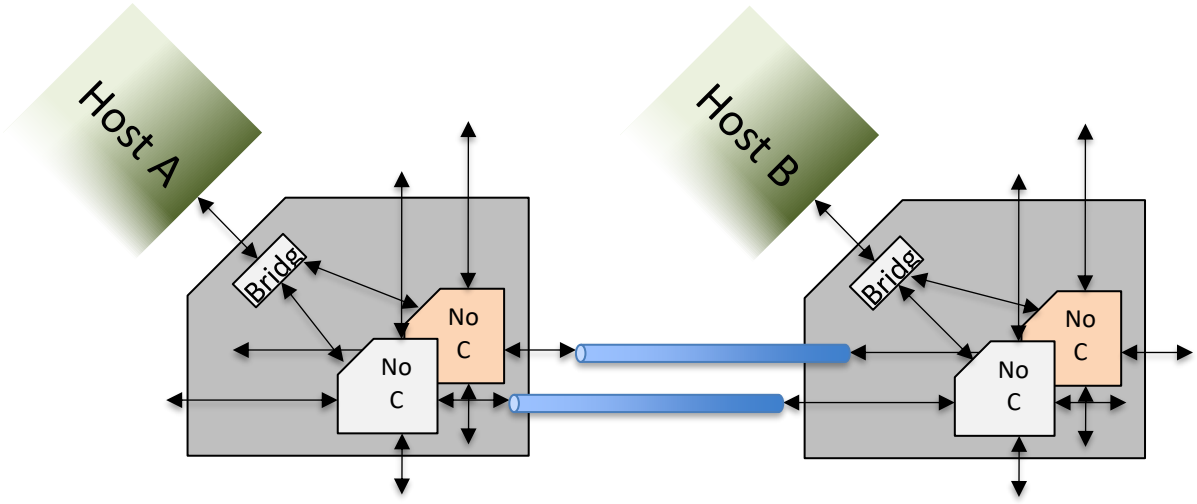

Figure 11: Dual NoC layers

6.1.2 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>

6.1.3 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.

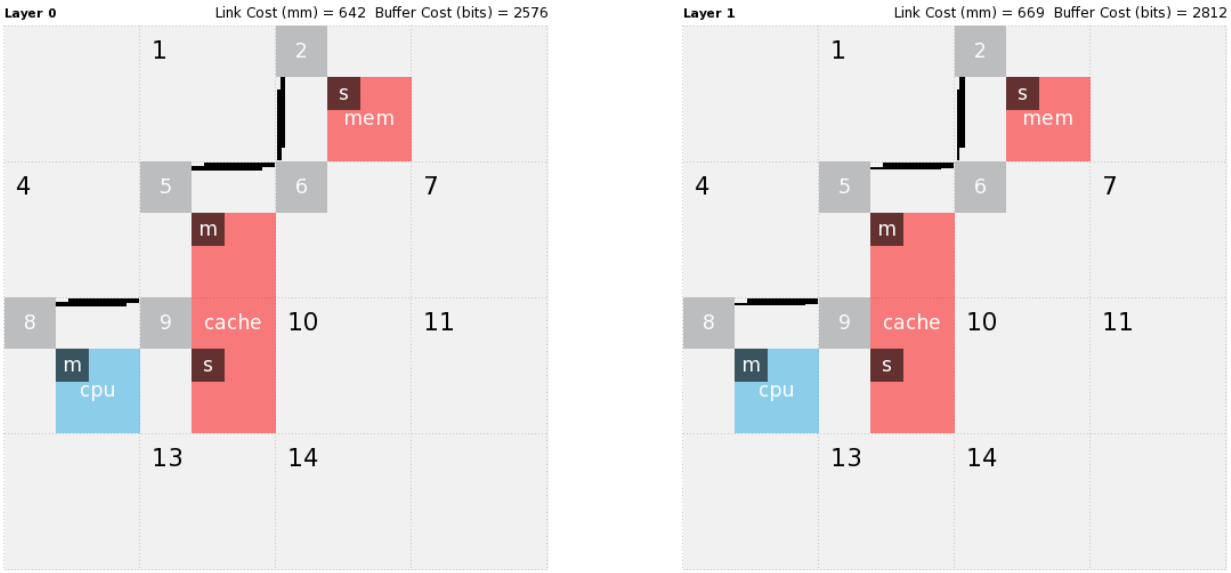

Figure 12: NocStudio GUI snapshot

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”

6.2 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

o **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

6.2.1 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.

6.2.2 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.

6.2.3 Running Synthesis

The following tools are needed to run the reference synthesis flow

6.2.4 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

6.2.5 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.

6.2.6 Analyzing outputs

6.2.6.1 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.

6.2.6.2 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.

6.3 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

6.4 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.

6.4.1 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.

6.4.2 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.

6.4.3 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.

6.4.4 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

7 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.

7.1 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.

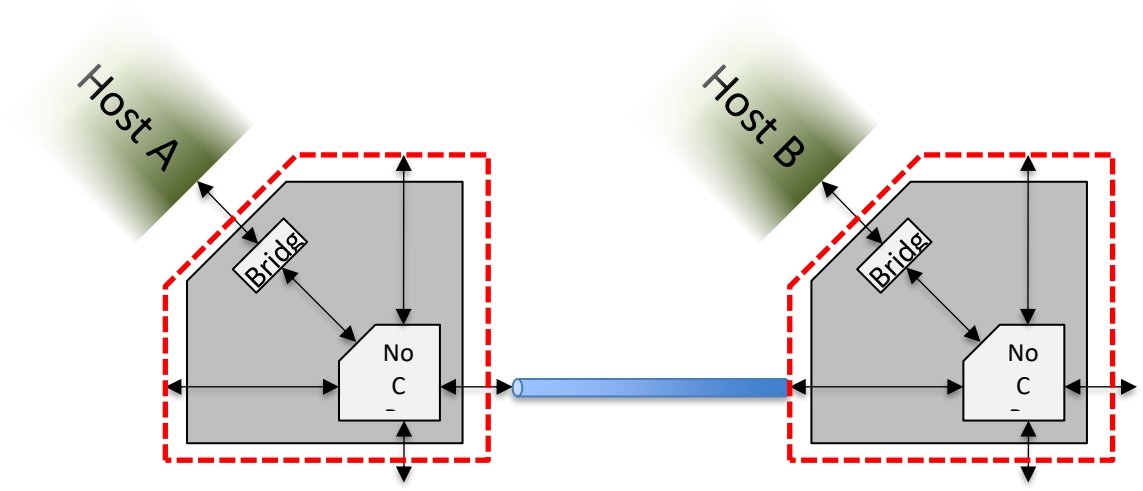

Figure 13: NoC Elements in stand-alone P&R regions

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

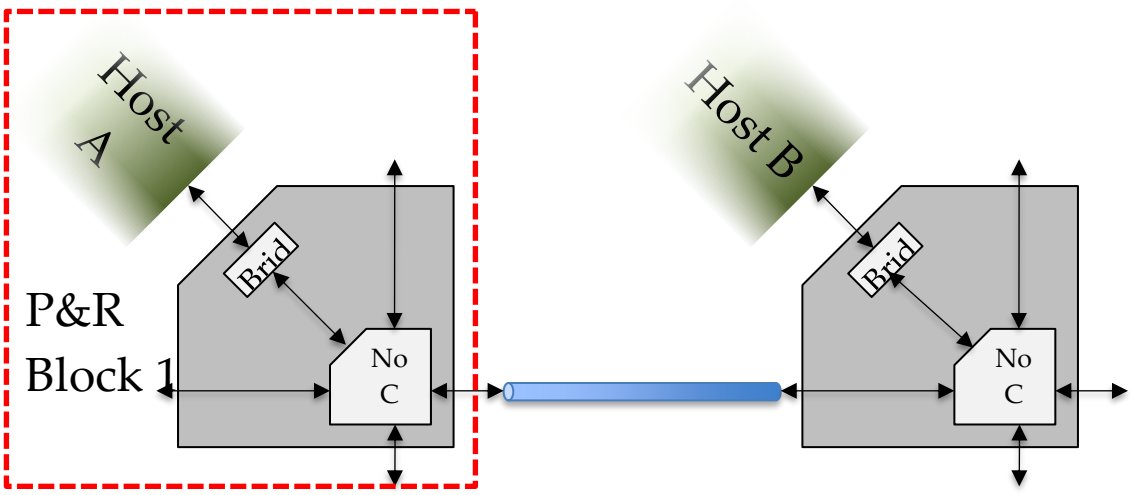

Figure 14: Merge all NoC Elements with Host P&R Region

7.3 NoC Quick Start Information

7.3.1 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.

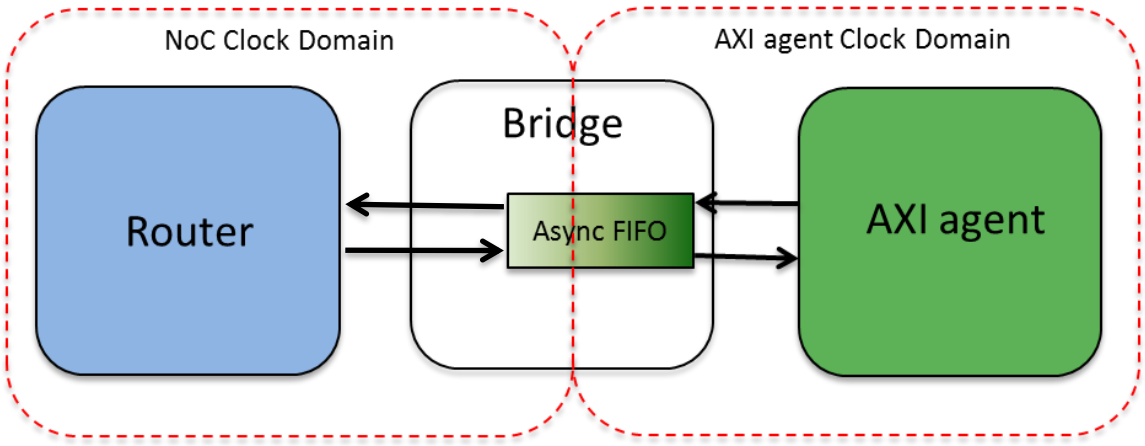

Figure 15: Bridge and AXI agent with async clock

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

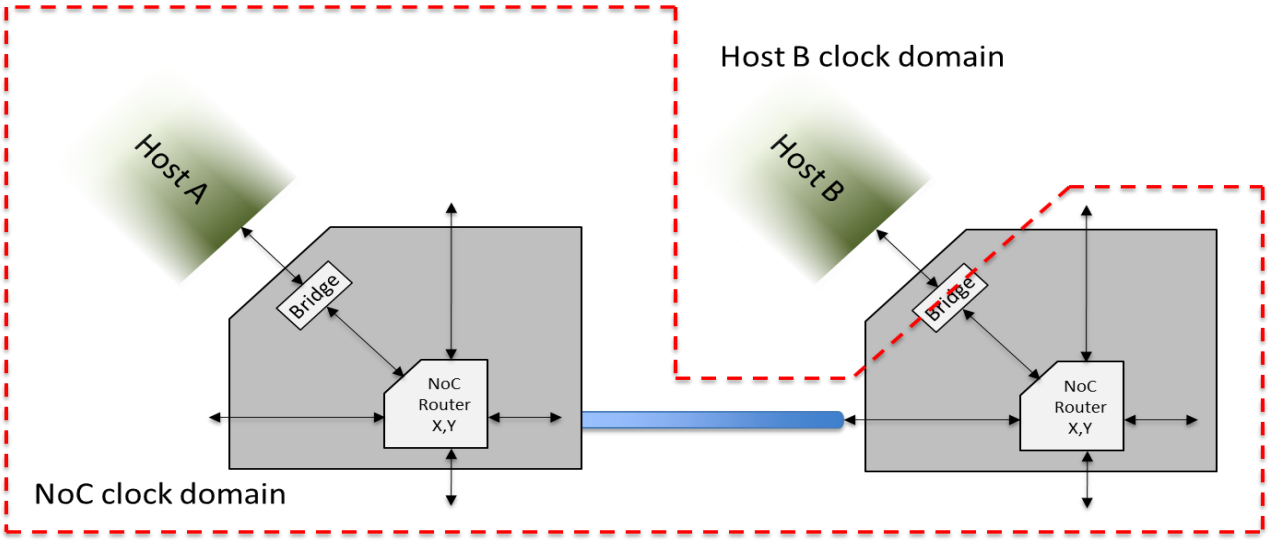

Figure 16: Host A and NoC clock domain different than Host B clock

Figure 17 is another implementation where Host A and Host B clock domains are different from NoC clock domain.

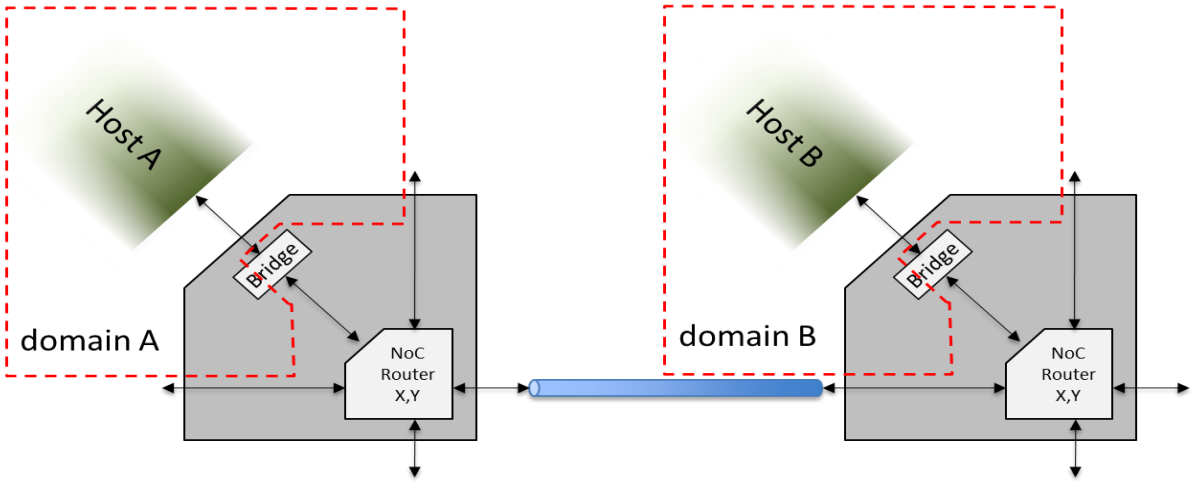

Figure 17: Host A & B with different clock domains than NoC clock

7.3.2 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.

7.3.3 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.

7.3.4 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 18 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.

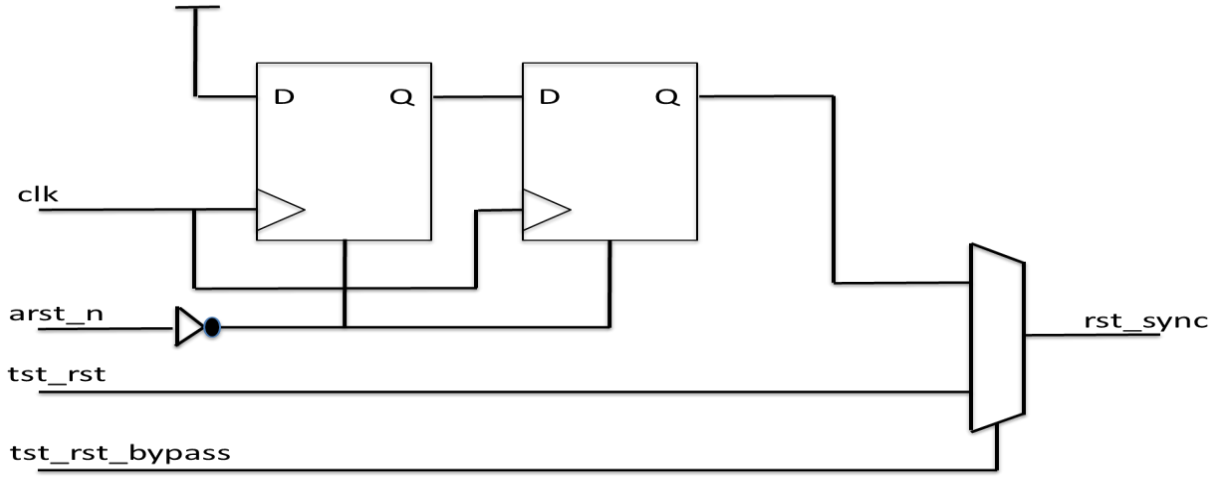

Figure 18: Reset logic

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.

7.3.5 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 18), 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.

7.3.6 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.

7.3.7 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.

7.3.8 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.

7.3.9 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.

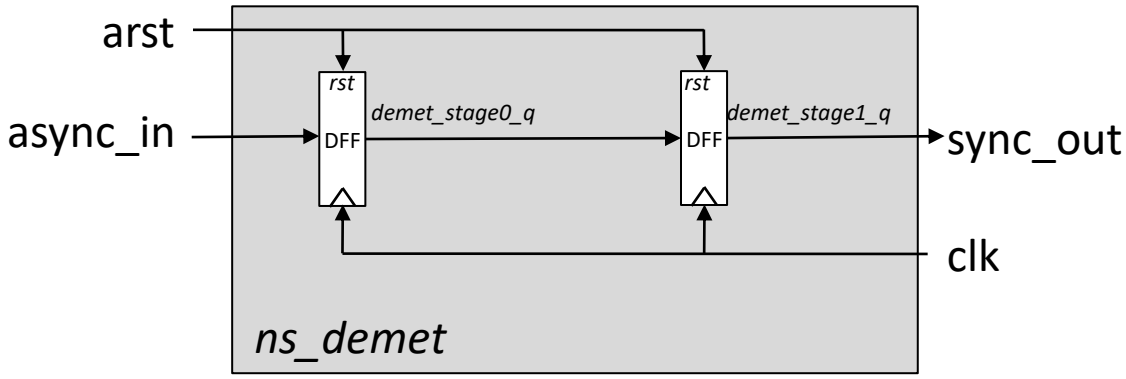

Figure 19: CFG synchronizer module

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*".

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

Below is the block diagram of VDC/ILDC:

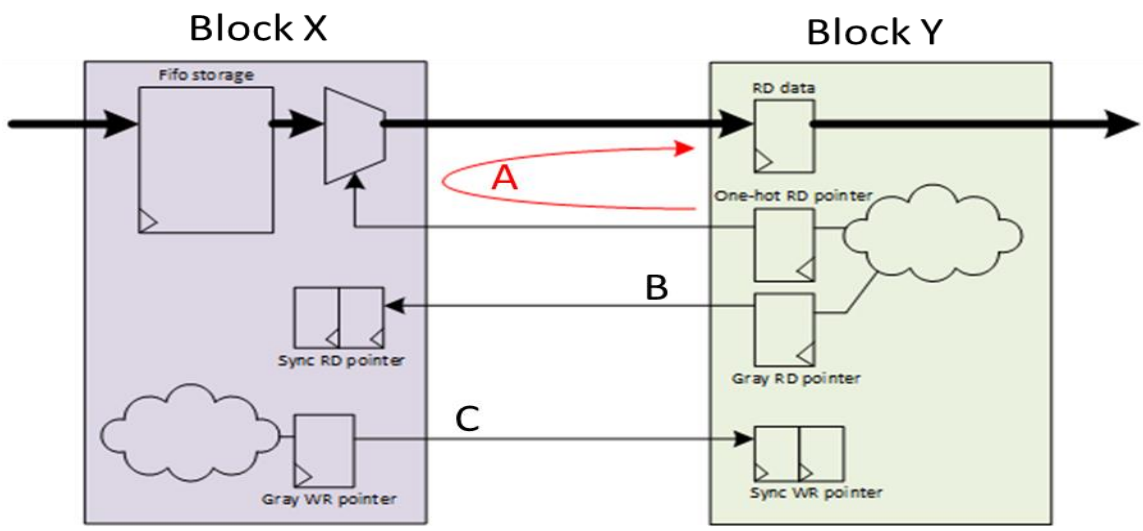

Figure 20: VDC and ILDC block diagram

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.

8 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.

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.