|  |
| --- |
| NetSpeed Gemini IP Integration Specification  Version: Gemini-15.11  November 8, 2015 |

NetSpeed Gemini IP Integration Specification

About This Document

This document describes the guidelines for seamless integration of NetSpeed Gemini 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 NetSpeed 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
* NoC Designers
* NoC Verification Engineers
* SoC Architects
* SoC Designers
* SoC Verification Engineers

Prerequisite

Before proceeding, you should generally understand:

* Basics of Network on Chip technology
* ARM AMBA 4 interconnect standard
* ARM ACE interconnect standard

Related Documents

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

* NetSpeed NocStudio Gemini User Manual
* NetSpeed Gemini Physical Design Guidelines
* NetSpeed Register Bus Protocol
* NetSpeed Gemini Protocol Support

Customer Support

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

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

**Contents**

[About This Document 2](#_Toc434427799)

[Audience 2](#_Toc434427800)

[Prerequisite 2](#_Toc434427801)

[Related Documents 2](#_Toc434427802)

[Customer Support 3](#_Toc434427803)

[1 NoC IP Overview 10](#_Toc434427804)

[1.1 NoC IP Components 10](#_Toc434427805)

[1.2 Directory Structure 10](#_Toc434427806)

[1.3 Documentation 11](#_Toc434427807)

[1.4 NocStudio Flow to Generate NoC IP 12](#_Toc434427808)

[1.4.1 Generating RTL from NocStudio 13](#_Toc434427809)

[2 Integration of NoC 18](#_Toc434427810)

[2.1 Integration of NoC RTL 18](#_Toc434427811)

[2.1.1 Hierarchical RTL generation 18](#_Toc434427812)

[2.1.2 Reset 18](#_Toc434427813)

[2.1.3 Clocks 19](#_Toc434427814)

[2.1.4 Clock Gating 21](#_Toc434427815)

[2.1.4.1 Clock Gating for NoCs without Regbus 22](#_Toc434427816)

[2.1.4.2 Clock Gating for NoCs with Regbus 22](#_Toc434427817)

[2.1.5 Register Bus 23](#_Toc434427818)

[2.1.6 Regbus Master Tunnel 23](#_Toc434427819)

[2.1.7 Interrupts 24](#_Toc434427820)

[2.1.8 Event Counters 25](#_Toc434427821)

[2.2 Integration of NoC Verification Checkers 25](#_Toc434427822)

[2.3 Synthesis 27](#_Toc434427823)

[2.4 Supported Tools 27](#_Toc434427824)

[3 NoC Verification Components 28](#_Toc434427825)

[3.1 Overview of Checkers 28](#_Toc434427826)

[3.2 Environment Setup for Integration 28](#_Toc434427827)

[3.3 Fast Initialization for CCC Directory 30](#_Toc434427828)

[3.4 Usage Modes 30](#_Toc434427829)

[3.5 Checkers 31](#_Toc434427830)

[3.5.1 Terminology 31](#_Toc434427831)

[3.5.2 AMBA NoC End-to-End Checker 32](#_Toc434427832)

[3.5.2.1 AMBA NoC End-to-End Fine-Grained User Control 39](#_Toc434427833)

[3.5.2.2 AMBA NoC End-to-End Traffic Logs 39](#_Toc434427834)

[3.5.2.2.1 AMBA NoC End-to-End Traffic Logs for Reads 40](#_Toc434427835)

[3.5.2.2.2 AMBA NoC End-to-End Traffic Logs for Writes 43](#_Toc434427836)

[3.5.2.2.3 AMBA NoC End-to-End Traffic Logs for AC channel 45](#_Toc434427837)

[3.5.2.3 AMBA NoC End-to-End Traffic Logs for CR/CD channels 46](#_Toc434427838)

[3.5.2.4 AMBA NoC End-to-End Request ID 46](#_Toc434427839)

[3.5.3 ACE Bridge Checkers 49](#_Toc434427840)

[3.5.4 Regbus End-to-End Checker 52](#_Toc434427841)

[3.5.5 Regbus Ring Slave Checker 55](#_Toc434427842)

[3.5.6 Clock Control Signal Checks 56](#_Toc434427843)

[3.5.7 Router Checker 56](#_Toc434427844)

[3.5.8 CCC Checkers 58](#_Toc434427845)

[3.5.9 DVM Checker 59](#_Toc434427846)

[3.5.10 Global Coherency Tracker (GCT) 60](#_Toc434427847)

[4 Document Changes/Revisions 63](#_Toc434427848)

List of Figures

[Figure 1 NoC IP generation flow 13](#_Toc434427205)

[Figure 2 Synchronous clock crossers require alignment of rising edge. 20](#_Toc434427206)

[Figure 3 A clock enable input indicates when the rising edges are shared by the clocks. 21](#_Toc434427207)

[Figure 4 NoC checkers binding to RTL 26](#_Toc434427208)

[Figure 5 AMBA NoC end-to-end checking architecture 47](#_Toc434427209)

List of Tables

[Table 1 NoC IP directory structure 10](#_Toc434427768)

[Table 2 NoC IP document list 11](#_Toc434427769)

[Table 3 Files and directories generated by NocStudio in the project directory 14](#_Toc434427770)

[Table 4 NoC reset signals 18](#_Toc434427771)

[Table 5 NoC clock signals 19](#_Toc434427772)

[Table 6 Regbus layer clock gating signal 22](#_Toc434427773)

[Table 7 NoC interrupt signals 24](#_Toc434427774)

[Table 8 NetSpeed checkers 28](#_Toc434427775)

[Table 9 `define variables for model build 28](#_Toc434427776)

[Table 10 `define variables for fast back-door initialization 30](#_Toc434427777)

[Table 11 Recommended checker settings 30](#_Toc434427778)

[Table 12 AMBA NoC end-to-end checks 32](#_Toc434427779)

[Table 13 Fine-grained user control of AMBA NoC end-to-end checks 39](#_Toc434427780)

[Table 14 Settings to enable AMBA NoC End-to-End Checker traffic logs 39](#_Toc434427781)

[Table 15 AMBA NoC End-to-End Checker log files 40](#_Toc434427782)

[Table 16 Nomenclature of ns\_noc\_acemstrbrdg\_<bridge\_id>\_ar\_r.log 41](#_Toc434427783)

[Table 17 Nomenclature of ns\_noc\_acemstrbrdg\_<bridge\_id>\_aw\_w\_b.log 43](#_Toc434427784)

[Table 18 Nomenclature of ns\_noc\_acemstrbrdg\_<bridge\_id>\_aw\_w\_b.log 45](#_Toc434427785)

[Table 19 Nomenclature of ns\_noc\_acemstrbrdg\_<bridge\_id>\_cr\_cd.log 46](#_Toc434427786)

[Table 20 ns\_transaction\_info struct 47](#_Toc434427787)

[Table 21 ACE bridge checks 49](#_Toc434427788)

[Table 22 Fine-grained user-control of ACE bridge checks 51](#_Toc434427789)

[Table 23 Register bus end-to-end checks 52](#_Toc434427790)

[Table 24 Settings to enable register bus end-to-end traffic logs 54](#_Toc434427791)

[Table 25 Register Bus End-to-End Checker log files 54](#_Toc434427792)

[Table 26 Regbus ring slave checks 55](#_Toc434427793)

[Table 27 Clock control signal checks 56](#_Toc434427794)

[Table 28 Router checks 56](#_Toc434427795)

[Table 29 CCC checker 58](#_Toc434427796)

[Table 30 DVM checker 59](#_Toc434427797)

[Table 31 Global Coherency Tracker 60](#_Toc434427798)

# NoC IP Overview

## 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

## 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\_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\_deines.vh | `defines file used for integration of NetSpeed verification IP into customer environment. |
| noc\_verif\_ip/\* | NoC verification checkers IP library. |
| scripts/\* | Scripts for sanity bench. |
| synth/\* | NoC synthesis environment. For more information, please refer to the NetSpeed Gemini Physical Design Guidelines document. |
| tutorials/\* | NocStudio tutorials. |
| user\_manual\_files/\* | Support files for NocStudio manuals. |

## Documentation

A separate directory is included with the following documents.

Table 2 NoC IP document list

|  |  |
| --- | --- |
| **Name** | **Description** |
| NetSpeed NocStudio Gemini User Manual.pdf | Overview, architecture, usage and examples. |
| NetSpeed Gemini IP Integration Spec.pdf | How to use NocStudio generated IP (this document). |
| NetSpeed Gemini Protocol Support.pdf | Details of AMBA protocol support in NetSpeed Gemini. |
| NetSpeed Gemini Physical Design Guidelines.pdf | Synthesis and placement recommendations. |
| NetSpeed Register Bus Protocol.pdf | Protocol and usage details for the register bus interface. |

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. |

## 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, whose name corresponds to the project name specified in the “new\_mesh” command in the NocStudio command script.



Figure 1 NoC IP generation flow

### Generating RTL from NocStudio

To generate NoC RTL, include “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

The last command in the above example script is “gen\_ip”. Once the command 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 NetSpeed NocStudio Gemini User Manual.

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

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Type** |
| archive/\* | Location into which old run output is placed when a new run is started. | 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 | Writing usage estimate. | NoC property |
| noc\_modifiable\_rtl/\* | RTL modules, such as RAM, that can be replaced by user implementation. | 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\_deines.vh | `defines file used for integration of NetSpeed verification IP into customer environment. | Verification |
| noc\_verif\_ip/\* | NoC verification checkers IP library. | Verification |
| ns\_agent\_modules.v | Additional host IP modules. | RTL |
| ns\_bind\_checkers.svh | Binds 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\_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\_test\_incisiv.sh | Run command to launch sanity bench for the generated NoC using Cadence Incisive simulator. | Sanity testbench |
| run\_test\_vcs.sh | Run command to launch sanity bench for the generated NoC using Synopsys VCS simulator. | Sanity testbench |
| scripts/\* | Scripts for sanity testbench. | Sanity testbench |
| synth/\* | NoC synthesis environment. For more information, please refer to the NetSpeed Gemini Physical Design Guidelines document. | 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 |

Once the NoC RTL has been generated, 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 basic operation of the NoC in simulation.

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

If you are using Synopsys VCS Simulator, run:

run\_test\_vcs.sh

If you are using Cadence Incisive Simulator, run:

run\_test\_incisiv.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\_vcs.sh -waves=1

Or, for Cadence Incisive Simulator:

run\_test\_incisiv.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. 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 build logs are named “build.log” and are located in the model directory. The log from the NoC traffic test is named “run.log” and is located in the trace directory. The log file from the register bus test is named “regbus\_run.log” and is located in the trace\_regbus directory.

The presence of the file SIM\_PASSED in the project directory after the completion of the simulation run indicates a successful simulation run.

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

# 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.

## 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:

$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.

### 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, NetSpeed RTL IP modules or NoC nodes. When grouped by NoC node, all routers and bridges at that node are added to the group module.

### Reset

Table 4 NoC reset signals

|  |  |
| --- | --- |
| **Signal name** | **Description** |
| reset\_n\_<voltage\_domain> | Reset pins are named according to voltage domain to which they belong. The reset pins are active low.   * Default voltage domain is “system”, and hence default reset pin is “reset\_n\_system” * All other voltage domains are added via “add\_voltage\_domain”, and are named accordingly. |

One reset pin per voltage domain is present at the NoC level. The reset pin is an active low reset pin. This reset pin is distributed to the different NoC elements belonging to that voltage domain. All reset signals are asynchronous at the NoC level. Internally to the NoC elements, reset may be asynchronously asserted, but are synchronously de-asserted

The customer must ensure that the active-low reset pin is driven low long enough so that all NoC elements 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. The reset logic within each module uses an asynchronous assert, synchronous de-assert mechanism. This involves registering the reset and ORing the registered version with the reset input. Because of this, de-assertion of the reset in the NoC modules will take additional time. Traffic to the NoC should be delayed until 2 cycles of the slowest clock 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 then synchronizes the reset to its clock before using it internally. The reset is de-asserted synchronously. Each NoC element can come out of reset at different times. The bit[0] of the link\_not\_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 NetSpeed Gemini Physical Design Guidelines provided with the release.

### Clocks

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

Table 5 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 “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/ratio\_fast clock crosser interface. The corresponding clk\_<clock\_domain> pin at the NoC level will be internally connected to the host\_clk 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 NetSpeed 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 happens within a bridge. 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. “ratio\_slow” refers to a clock crosser where the host is running slower than the NoC. The “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 Synchronous clock crossers require alignment of rising edge.

The N:1 and 1:N clock crossers can actually support a range of ratios. N can be 1, 2, 3 or 4. This allows the faster clock to run at either the same frequency, or up to 4 times faster.

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.

### Clock Gating

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

### 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.

### 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 s*ystem\_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 6 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. |

### Register Bus

The NoC has an optional, distributed register network consisting of registers used for debug visibility, performance status collection, and error logging. This 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 NetSpeed 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.

**Regbus Master Tunnel option**: NocStudio instantiates a Regbus Master Tunnel, which connects to the NoC regbus layer port for reading or writing registers of the NoC.

### Regbus Master Tunnel

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: “host\_prop rbm tunnel\_slv1\_enabled yes”. The second port is connected directly to the regbus master agent.
  + 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.

### Interrupts

Every NoC router and bridge has an interrupt output signal. Interrupt is asserted when a fatal error is encountered in a NoC element. The errors are also logged in interrupt status registers of the NoC element. Interrupts from different NoC elements are combined within the NoC by a specialized network. A single combined interrupt is brought out on the NoC external interface. The errors are also logged within the NoC registers, which are described in the HTML documentation generated by NocStudio.

Table 7 NoC interrupt signals

|  |  |
| --- | --- |
| **Signal name** | **Description** |
| Interrupt | Single interrupt pin which combines all interrupt events inside the NoC |

The interrupt signals are outputs of individual NoC elements, and will exist as local pins at the physical boundary of those elements. Interrupt network handles pipelined routing of the interrupt signals along NoC channels. Combining of the interrupt signals across different clock domains in the NoC is also handled internally. Currently the combined interrupt pin is transported to the register bus master’s physical grid location if register bus is enabled in a configuration. When register bus is disabled, NocStudio picks a position to transport the combined reset to.

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.

### 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.

## Integration of NoC Verification Checkers

For details on the NetSpeed IP verification checkers, see Section 3. 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:

$NS\_PROJ\_PATH = /absolute/path/of/project/created/

* Adjust the `defines settings in the following file according to the recommended usage inError! Reference source not found.and Table 22,

noc\_verif\_cust/ns\_global\_defines.vh

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.

## Synthesis

Please refer to NetSpeed Gemini Physical Design Guidelines.

## Supported Tools

Supported versions of tools and languages:

* NoC RTL uses Verilog-2001 (IEEE Std 1364™-2001) syntax and its support must be enabled in the tool flow.
* NoC simulation environment uses SystemVerilog IEEE Std 1800-2009
* Simulator: Cadence Incisiv 13.20.004
* Simulator: Synopsys J-2014.12-SP3-2
* Synthesis: Cadence Encounter RTL Compiler version RC12.23 - v12.20-s027\_1 (64-bit)

# NoC Verification Components

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

## Overview of Checkers

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

Table 8 NetSpeed 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 |
| Regbus Ring Slave Checker | One instance per Regbus Ring Slave RTL |
| Router Checker | One instance per Router RTL |
| CCC checkers | One instance per CCC |
| DVM checker | One instance per DVM |
| Global Coherency Tracker(GCT) | One instance per NoC |

## Environment Setup for Integration

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

Table 9 `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 mapping to a plusarg variable to allow run-time control of value. |
| `NS\_E2E\_LOG | Set to 1 to enable, 0 to disable traffic logging by AMBA NoC End-to-End Checker. | Recommend mapping to a plusarg variable to allow run-time control of value. |
| `NS\_REGBUS\_END2END\_CHECKER\_EN | Set to 1 to enable, 0 to disable Regbus End-to-End Checker. | Recommend mapping to a plusarg variable to allow run-time control of value. |
| `NS\_REGBUS\_E2E\_LOG | Set to 1 to enable, 0 to display traffic logging by the Regbus End-to-End Checker. | Recommend mapping 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 mapping 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 mapping 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 mapping to a plusarg variable to allow run-time control of value. |
| `NS\_ROUTER\_CHECKER\_EN | Set to 1 to enable, 0 to disable Router Checker. | Recommend mapping to a plusarg variable to allow run-time control of value. |
| `NS\_ROUTER\_CHECKER\_COMPILE\_EN | Define to enable compilation of Router Checker. | Provides fine-grained checks at the expense of performance. |
| `NS\_CCC\_CHECKER\_EN | Set to 1 to enable, 0 to disable CCC checkers. | Recommend to map to a plusarg variable to allow run-time control of value. |
| `NS\_DVM\_CHECKER\_EN | Set to 1 to enable, 0 to disable DVM checker. | Recommend map to a plusarg variable to allow run-time control of value. |
| `NS\_GCT\_EN | Set to 1 to enable, 0 to disable Global Coherency Tracker. | Recommend map to a plusarg variable to allow run-time control |

## Fast Initialization for CCC Directory

By default, the coherency directory explicitly initializes hardware after reset. This can slow simulation time. If the user wants to avoid the extra simulation time, NetSpeed provides the `define in the following table that will immediately initialize the directory with a backdoor mechanism.

Table 10 `define variables for fast back-door initialization

|  |  |
| --- | --- |
| `define Variable Name | Description |
| `NS\_FORCE\_RTL\_DIRECTORY\_SHORT\_INIT | If defined, immediately initializes directory using a backdoor mechanism.  If not defined, hardware will explicitly perform directory initialization. |

## Usage Modes

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

Table 11 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\_ROUTER\_CHECKER\_EN | 0 | 1 | 0 |
| `NS\_ROUTER\_CHECKER\_COMPILE\_EN | Do not define | Define | Do not define |
| `NS\_CCC\_CHECKER\_EN | 1 | 1 | 1 |
| `NS\_DVM\_CHECKER\_EN | 1 | 1 | 1 |
| `NS\_GCT\_EN | 1 | 1 | 1 |

## Checkers

### Terminology

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

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

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

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

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

### AMBA NoC End-to-End Checker

The AMBA NoC End-to-End Checker is used to verify 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 12 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 expected to force modification for functional correctness. | Always | Functional |
| 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 |
| 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 |
| 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 roundtrip check to ensure correct propagation of command fields, propagation of data and ordering within the NoC. | Fast Tap | Functional |
| 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 |
| 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.

### 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 13 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.

### 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 14 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 the each bridge assigned by NocStudio.

Table 15 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 |

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

### 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 : 0x200000008 : master\_id : 0x0 : split\_cnt : 0 : split\_size : 64 : slave\_id : 0x4 : AR\_sent\_time : 2065 : unknown\_dest : 0 :   seq\_num: 0x0 : mprt\_ARADDR : 0x70000f80 mprt\_ARID : 0x5 : mprt\_ARLEN : 0x3 : mprt\_ARSIZE : 0x4 : mprt\_ARBURST : 0x2 : mprt\_ARCACHE : 0x2 : mprt\_ARPROT :   0x1 : mprt\_ARQOS : 0xb : mprt\_ARLOCK : 0x0 : mprt\_ARUSER : 0x0 : mprt\_ARSNOOP : 0x0 : mprt\_ARDOMAIN : 0x0 :   mprt\_ARBAR : 0x0 : ARREGION : 0x0 : fast\_tap : 0

<sim\_time> : request\_id : 0x200000008 : master\_id : 0x0 : split\_cnt : 0 : split\_size : 64 : slave\_id : 0x4 : R\_received\_time : 3425 : seq\_num: 0x1 : acmb\_RID : 0x1 : acmb\_RRESP : 0x0 : acmb\_RUSER(per transaction) : 0x0 : acmb\_RUSER\_CL(per beat) : 0x0 : acmb\_RLAST: 0x1 : derived\_mprt\_ARADDR : 0x40000b0b : derived\_sys\_ARID : 0x1 : acmb\_RDATA : 0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef : fast\_tap : 0

Table 16 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. |
| <seq\_num> | The internal NoC sequence number, used for tracking ordering. |
| <mprt\_AR\*> | Fields of AR channel as seen by the master bridge. |
| <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. |

### 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 : 0x200000009 : master\_id : 0x0 : split\_cnt : 0 : split\_size : 64 : slave\_id : 0x4 : AW\_sent\_time : 1975 : W\_sent\_time : 1975 : unknown\_dest : 0 :     seq\_num: 0x2 : mprt\_AWADDR : 0x50000bfc mprt\_AWID : 0x3 : mprt\_AWLEN : 0x0 : mprt\_AWSIZE : 0x4 : mprt\_AWBURST : 0x1 : mprt\_AWCACHE : 0x0 : mprt\_AWPROT :     0x5 : mprt\_AWQOS : 0x8 : mprt\_AWLOCK : 0x0 : mprt\_AWUSER : 0x0 : mprt\_AWSNOOP : 0x0 : mprt\_AWDOMAIN : 0x0 :     mprt\_AWBAR : 0x0 : AWREGION : 0x0 : mprt\_WDATA\_CL : 0xb2a758e8000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 : mprt\_WSTRB\_CL : 0xf000000000000000 : mprt\_WUSER\_CL : 0x0

<sim\_time> : request\_id : 0x200000009 : master\_id : 0x0 : split\_cnt : 0 : split\_size : 64 : slave\_id : 0x7 : B\_received\_time : 2285 : bid : 0x0 : bresp : 0x0 : buser : 0x0 : derived\_mprt\_AWADDR : 0x3c00028e : derived\_sys\_AWID : 0x0, seq\_num : 0x0

Table 17 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. |
| <seq\_num> | The internal NoC sequence number, used for tracking ordering. |
| <mprt\_AW\*> | Fields of AW channel as seen by the master bridge. |
| <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.

### AMBA NoC End-to-End Traffic Logs for AC channel

The following is an example of “ns\_noc\_acemstrbrdg\_<bridge\_id>\_ac.log”:

<sim\_time> : master\_id : 0x3 : slave\_id : 0x11 : AC\_sent\_time : 4524 : acmb\_ACADDR : 0x40000120 acmb\_ACcrtid : 0x0 acmb\_ACPROT : 0x2 : acmb\_ACSNOOP : 0x2

Table 18 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. |
| <master\_id> | The bridge ID of the master bridge that sent the AR request, or received R response. |
| <slave\_id> | The bridge ID of the slave bridge that received the AR request or sent the R response. |
| <AC\_sent\_time> | Simulation time when the AC request was sent by the master bridge. |
| acmb\_AC\* | Fields of AC channel as seen by the master bridge |

### AMBA NoC End-to-End Traffic Logs for CR/CD channels

<sim\_time> : master\_id : 0x2 : slave\_id : 0x11 : CR\_sent\_time : 18588 : mprt\_CRcrtid : 0x7 mprt\_CRRESP : 0x0

Table 19 Nomenclature of ns\_noc\_acemstrbrdg\_<bridge\_id>\_cr\_cd.log

|  |  |
| --- | --- |
| **Field name** | **Description** |
| <sim\_time> | Simulation time when master bridge traffic collected and printed. |
| <master\_id> | The bridge ID of the master bridge that sent the AR request, or received R response. |
| <slave\_id> | The bridge ID of the slave bridge that received the AR request or sent the R response. |
| <CR\_sent\_time> | Simulation time when the CR response was sent by the master bridge. |
| mprt\*CR\* | Fields of CR channel as seen by the master bridge |

### 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 20 ns\_transaction\_info struct

|  |  |
| --- | --- |
| **Field name** | **Description** |
| bridge\_id | The bridge ID of the master or slave bridge that sent the request. |
| 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\_global\_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 in your environment:

* Add the following file to your file list. This file contains the definition of ns\_transaction\_info.

noc\_verif\_ip/ns\_amba\_struct.sv

* In your environment, 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:

//==========================================================

// Example code to poll request id from E2E mailboxes.

//==========================================================

// if(`NS\_E2E\_CHECKER\_TOP.ns\_transaction\_src\_mbox.try\_get(ns\_src\_transaction)) begin

// $display("%t NS SRC: 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.try\_get(ns\_dst\_transaction)) begin

// $display("%t NS DST: 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

### ACE Bridge Checkers

The ACE bridge checkers are responsible for monitoring ACE bridge RTL during simulation. Each instance of ACE Master Bridge and ACE Slave Bridge RTL has a corresponding ACE 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 ACE bridge checkers perform micro-architectural checks to ensure functional correctness of the ACE 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 ACE bridge RTL is in a proper idle state.

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

Table 21 ACE bridge checks

|  |  |  |
| --- | --- | --- |
| **Description of check** | **Instantiated (per bridge or interface)** | **Type of check** |
| ARVALID, AWVALID, WVALID, RVALID, BVALID, ACVALID, CRVALID, CDVALID are low when in reset. | ACE Master Bridge | Protocol |
| ARVALID, AWVALID, WVALID, RVALID, BVALID, ACVALID, CRVALID, CDVALID are never x or z. | ACE Master Bridge | Protocol |
| AR, AW, W, R, B, AC, CR, CD control signals must not be x or z when corresponding VALID and READY are high. | ACE 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. | ACE Master Bridge | Protocol |
| Narrow transfers on AR and AW interfaces are not permitted when support for narrow transfers is disabled on the bridge. | ACE Master Bridge | Protocol |
| WRAP bursts on AR and AW interfaces must have total transfer data size of 16, 32 or 64 bytes. Other total transfer data sizes are currently not supported. | ACE Master Bridge | Unsupported |
| INCR burst total transfer data size must be no more than the maximum size (4KB). | ACE Master Bridge | Unsupported |
| ARREADY, AWREADY low when in reset. | ACE Master Bridge | Functional |
| RDATA is never interleaved across different RIDs. | ACE Master Bridge | Functional |
| INCR ReadOnce request from an ACE master is not allowed to cross cache line boundary. | ACE master bridge | Protocol |
| INCR WriteUnique request from an ACE master is not allowed to cross cache line boundary. | ACE master bridge | Protocol |
| WDATA is never interleaved across different WIDs. | ACE Master Bridge | Functional |
| WID must match AWID for same request. | ACE Master Bridge | Functional |
| ARVALID, AWVALID, WVALID, RVALID, BVALID, ACVALID, CRVALID, CDVALID are low when in reset. | ACE Slave Bridge | Protocol |
| ARVALID, AWVALID, WVALID, RVALID, BVALID, ACVALID, CRVALID, CDVALID are never x or z. | ACE Slave Bridge | Protocol |
| AR, AW, W, R, B, AC, CR, CD control signals must not be x or z when corresponding VALID and READY are high. | ACE 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. | ACE Slave Bridge | Protocol |
| RREADY and BREADY are low when in reset. | ACE Slave Bridge | Functional |
| RDATA is not interleaved across different RIDs if RDATA de-interleaving logic is not enabled. | ACE Slave Bridge | Functional |
| Internal RTL micro-architectural checks. | ACE Slave Bridge | Functional |
| All bridge RTL FIFOs are empty. | ACE Master Bridge | Exit |
| All bridge RTL tracking tables are empty. | ACE Master Bridge | Exit |
| All bridge RTL buffers are freed up. | ACE Master Bridge | Exit |
| ARVALID, AWVALID, WVALID, RVALID, BVALID are low. Internal NoC valids are low. | ACE Master Bridge | Exit |
| All bridge RTL FIFOs are empty. | ACE Slave Bridge | Exit |
| All bridge RTL tracking tables are empty. | ACE Slave Bridge | Exit |
| All bridge RTL buffers are freed up. | ACE Slave Bridge | Exit |
| ARVALID, AWVALID, WVALID, RVALID, BVALID are low. Internal NoC valid signals are low. | ACE 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 22 Fine-grained user-control of ACE 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 transfers is disabled on the bridge. | `NS\_ACEBRDG\_NARROWS\_CHECK\_DISABLE | 0 |

### 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 23 Register bus end-to-end checks

|  |  |  |
| --- | --- | --- |
| **Description of check** | **Instantiated (per bridge or interface)** | **Type of check** |
| ARVALID, AWVALID, WVALID, RREADY are never x or z. | Regbus Master Bridge | Protocol |
| ARLEN and AWLEN are either 0 or 1. | Regbus Master Bridge | Protocol |
| Every AR request that enters the Regbus Master Bridge is tracked with its corresponding R response for the roundtrip check to ensure correct propagation of command fields, propagation of content and ordering within the regbus layer. | NoC | Functional |
| Every AR request that enters the Regbus Master Bridge arrives at the correct destination Regbus Ring Master, with the correct ARADDR, ARPROT, ARLEN, node\_id, ring\_id, seqnum, in the correct order. | NoC | Functional |
| Every read request packet that enters the Regbus Ring Master is tracked with its corresponding response packet for the roundtrip check to ensure correct propagation of command fields, propagation of content and ordering within each Regbus Ring Master. | Regbus Ring Master | Functional |
| Every R response that leaves the Regbus Ring Master arrives at the Regbus Master Bridge with the correct RDATA, RRESP, RLAST, in the correct order. | NoC | Functional |
| Every pair of AW and W requests that enters the Regbus Master Bridge is tracked together with the corresponding B response for roundtrip check to ensure correct propagation of command fields, propagation of data and ordering within the regbus layer. | NoC | Functional |
| Every AW and W request that enters the Regbus Master Bridge arrives at the correct destination Regbus Ring Master with the correct AWADDR, AWPROT, AWLEN, WDATA, WSTRB and WLAST, node\_id, ring\_id, seqnum, in the correct order. | NoC | Functional |
| Every write request packet that enters the Regbus Ring Master is tracked with its corresponding response packet for the roundtrip check to ensure correct propagation of command fields, propagation of content and ordering within each Regbus Ring Master. | Regbus Ring Master | Functional |
| Every B response that leaves a Regbus Ring Master arrives at the Regbus Master Bridge with the correct BRESP, in the correct order. | NoC | Functional |
| There is one and only one SOP per Regbus Ring Master packet. | Regbus Ring Master | Protocol |
| There is one and only one EOP per Regbus Ring Master packet. | Regbus Ring Master | Protocol |
| Valid is high only when a Regbus Ring Master packet is in-flight. | Regbus Ring Master | Protocol |
| No regbus request or response is in-flight. | NoC | Exit |
| All regbus requests and responses during the simulation are accounted for. | NoC | Exit |

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

Table 24 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 25 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.

### 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 26 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 |

### Clock Control Signal Checks

Table 27 Clock control signal checks

|  |  |  |
| --- | --- | --- |
| **Description of check** | **Instantiated (per bridge or interface)** | **Type of check** |
| scan\_mode pin can toggle only when ACE master bridge is idle. | ACE Master Bridge | Functional |
| system\_cg\_or pin can toggle only when ACE master bridge is idle. | ACE Master Bridge | Functional |
| system\_clk\_en pin can toggle only when ACE master bridge is idle. | ACE Master Bridge | Functional |
| scan\_mode pin can toggle only when ACE slave bridge is idle . | ACE Slave Bridge | Functional |
| system\_cg\_or pin can toggle only when ACE slave bridge is idle. | ACE Slave Bridge | Functional |
| system\_clk\_en pin can toggle only when ACE slave bridge is idle. | ACE Slave Bridge | Functional |

### Router Checker

The Router Checker is responsible for monitoring router RTL during simulation. Each instance of router RTL has a corresponding router checker monitoring its behavior. The Router Checker enforces adherence to protocol on all ports. In addition, it ensures functional correctness of router RTL during simulation. At the end of simulation, when there should be no traffic in the NoC, this checker performs exit checks to ensure each instance of router RTL is in a proper idle state.

Table 28 Router checks

|  |  |  |
| --- | --- | --- |
| **Description of check** | **Instantiated** | **Type of check** |
| Control signals must not be unknown (x or z) when out of reset. | Each Router Input Port | Protocol |
| Reset has to be active for at least 16 clocks. | Each Router Input Port | Protocol |
| VC valid is one-hot. | Each Router Input Port | Protocol |
| Output port should be 0-4. | Each Router Input Port | Protocol |
| Output port should not be the same as input port. | Each Router Input Port | Protocol |
| Credit overflow per VC. | Each Router Input Port | Protocol |
| Credit underflow per VC. | Each Router Input Port | Protocol |
| Packets received on disabled VC. | Each Router Input Port | Protocol |
| SB info should remain constant throughout the packet. | Each Router Input Port | Protocol |
| Packet type should remain constant throughout the packet. | Each Router Input Port | Protocol |
| Output port should remain constant throughout the packet. | Each Router Input Port | Protocol |
| 2 SOPs without EOP in between for a VC. | Each Router Input Port | Protocol |
| 2 EOPs without SOP in between for a VC. | Each Router Input Port | Protocol |
| BV is non-zero only at EOP. | Each Router Input Port | Protocol |
| Control signals must not be unknown (x or z) when out of reset. | Each Router Output Port | Protocol |
| Reset must be active for at least 16 clocks. | Each Router Output Port | Protocol |
| VC valid is one-hot. | Each Router Output Port | Protocol |
| Output port should be 0-4. | Each Router Output Port | Protocol |
| Credit overflow per VC. | Each Router Output Port | Protocol |
| Credit underflow per VC. | Each Router Output Port | Protocol |
| Packets received on disabled VC. | Each Router Output Port | Protocol |
| SB info should remain constant throughout the packet. | Each Router Output Port | Protocol |
| Packet type should remain constant throughout the packet. | Each Router Output Port | Protocol |
| Output port should remain constant throughout the packet. | Each Router Output Port | Protocol |
| 2 SOPs without EOP in between for a VC. | Each Router Output Port | Protocol |
| 2 EOPs without SOP in between for a VC. | Each Router Output Port | Protocol |
| BV is non-zero only at EOP. | Each Router Output Port | Protocol |
| All credits have restored to initial value. | Each Router | Exit |
| All router FIFOs are empty. | Each Router | Exit |

### CCC Checkers

The CCC checkers monitor different RTL structures within the CCC and perform micro-architectural checks for illegal RTL states and exit checks that each structure is in an idle state at end of simulation. The CCC directory control checker is an architectural reference model of the CCC directory control RTL that dynamically monitors and checks directory functionality during simulation.

Table 29 CCC checker

|  |  |
| --- | --- |
| **Description of Check** | **Type of Check** |
| No illegal traffic combinations on every interface. | Functional |
| No X or Z on control signals. | Functional |
| FIFO overflow and underflow. | Functional |
| Self-snoop never happens to a master who initiates the request. | Protocol |
| Micro-architectural checks. | Functional |
| Track every incoming request to be processed by the coherency directory in the reference model | Functional |
| Check every output of the directory is architecturally correct. | Functional |
| Check every snoop is consistent with the directory content. | Functional |
| Micro-architectural checks. | Functional |
| CCC RTL structures are empty. | Exit |
| The entire content of the coherency directory must be identical to the content of the architectural reference model. | Exit |

### DVM Checker

The DVM checker tracks behavior of DVM RTL during simulation.

Table 30 DVM checker

|  |  |
| --- | --- |
| Description of Check | Type of Check |
| Multi-part requests must arrive and get sent atomically. | Protocol |
| Multi-part snoops are sent atomically. | Protocol |
| Multi-part DVM messages must have ARADDR[0] set to 0, and the same master ID and ARID. | Protocol |
| Multi-part DVM snoops must have ACADDR[0] set to 0, and the same destination ID. | Protocol |
| Track DVM synchronization requests and make sure all the corresponding snoops and snoop responses complete in the correct order. | Protocol |
| Track AID ordering to make sure responses for requests with the same AID are turned in order. | Protocol |
| For SYNC command, all slave agents should have received SYNC command. | Protocol |
| When agent that originates the SYNC receives completion, all slave agents should have completed processing SYNC. | Protocol |
| SYNC message in AC channel must have AxADDR[15] and AxADDR[11:0] set to 0 | Protocol |
| SYNC message in AR channel must have AxADDR[15] and AxADDR[11:0] set to 0. | Protocol |
| No RTL FIFO overflow | Functional |
| Check no illegal traffic shows up on the DVM | Functional |
| DVM RTL structures are empty. | Exit |
| No DVM requests outstanding. | Exit |

### Global Coherency Tracker (GCT)

The Global Coherency Tracker globally checks data correctness of every address in the coherent domain. It has a dynamic memory reference model that is used to ensure data correctness for traffic on major interfaces of the NoC.

Table 31 Global Coherency Tracker

|  |  |  |
| --- | --- | --- |
| Description of check | Instantiated | Type of check |
| AR request: check R data against reference memory. | ACE master bridge | Functional |
| AW request: if copyback is from an agent without unique ownership, check WDATA against reference memory. | ACE master bridge | Functional |
| AW request: if copyback is from an agent with unique ownership, update reference memory with WDATA. | ACE master bridge | Functional |
| AC request: if CD bus is used for snoop data and from an agent without unique ownership, check snoop data against reference model | ACE master bridge | Functional |
| AC request: if CD bus is used and from an agent with unique ownership, update reference memory with CDDATA. | ACE master bridge | Functional |
| Check that address is cache-line aligned. | ACE master bridge | Protocol |
| Check that at most one agent has unique ownership of a line. | ACE master bridge | Protocol |
| Check that at most one agent has a line in dirty state. | ACE master bridge | Protocol |
| Check that a line with unique ownership by one agent does not have cached copies in any state in the system. | ACE master bridge | Protocol |
| Check that upon arrival or RRESP of a MakeInvalid request, the line is not cached anywhere in the system. | ACE master bridge | Protocol |
| IO coherent write: update reference memory with WDATA | IOCB | Functional |
| IO coherent read: check data against reference model | IOCB | Functional |
| AR request: check RDATA against reference memory. | CCC | Functional |
| AC request: check CDDATA against reference memory | CCC | Functional |
| AW request: check copyback WDATA against reference memory | CCC | Functional |
| All coherent requests and responses are account for with no request or response outstanding. | NoC | Exit |

# Document Changes/Revisions

*Documentation Changes* include additions, deletions, and modifications made to this document. This section identifies the changes made in each release of the document.

2670 Seely Avenue

Building 11

San Jose CA 95134

[www.netspeedsytems.com](file:///C:\Users\Ranjeet\Documents\Netspeed\NocStudio\netspeed_noc\trunk\doc\release_docs\www.netspeedsytems.com)