NocStudio Orion NSIP Tutorial

Revision 0.6

March 25th, 2015

NocStudio Orion NSIP Tutorial

REVISION HISTORY

Rev 0.6	March 25th, 2015	Q1'15 Release

About This Document

This document is a tutorial for NocStudio Orion Streaming IP. Using NocStudio, users can define NoC architectures, describe specifications and requirements, optimize the NoC design and finally generate the Orion NoC IP files such as RTL, testbench, C++ models, synthesis scripts, NoC IP documentation etc.

Audience

This document is intended for users of NocStudio:

- NoC Architects
- NoC Designers
- SoC Architects

Related Documents

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

- NetSpeed Orion User Manual
- NetSpeed Orion IP Integration Spec

Customer Support

For technical support about this product, please contact support@netspeedsystems.com

For general information about NetSpeed products refer to: www.netspeedsystems.com

Contents

1 TUTORIAL AND EXAMPLE SESSION

1.1 STARTING NOCSTUDIO

NocStudio GUI can be launched by running NocStudio.sh from the command prompt in Linux or by running NocStudio-GUI.exe on Windows.

A filename containing a list of NocStudio commands (NocStudio command script) can be provided as an argument in which case upon launching NocStudio the contents of the file would be executed by NocStudio. When a filename is provided, NocStudio can also be run in a nogui mode by providing a –nogui switch at the end in which case the GUI will not be launched, and the contents of the file would be executed silently by NocStudio.

```
Linux$ ./NocStudio.sh
Linux$ ./NocStudio.sh commands.txt
Linux$ ./NocStudio.sh commands.txt –nogui
```

```
Win> NocStudio-GUI.exe
Win> NocStudio-GUI.exe commands.txt
Win> NocStudio-GUI.exe commands.txt –nogui
```

1.2 NETSPEED STREAMING BRIDGE TUTORIAL AND EXAMPLE INTERACTIVE SESSION

```
$ prop_default cell_size 8
$ prop_default data_width 64
$ new_mesh 4 4 3 stream_test
$ mesh_prop virtual_ok yes
```

First we set the default cell size and data width at 9 and 72 respectively (if user is using cell size other than 9, then it is recommended to set these two props together before creating mesh or

before adding hosts so that all default interface widths will be initialized at correct values). Then we create a new project directory named "stream_test" to store all files related to the project. This command does not load an existing project that may exist in that directory, with the exception of traffic trace files. This command also clears any existing configuration and initializes a new mesh with 3 layers 4 columns and 4 rows. The next command lets NocStudio know that virtual nodes of the NoC can be used to place a host and hostport.

```
$ add_host host00 color blue bridge m stream
$ add_host host01 color blue bridge m stream
$ add_host host02 color blue bridge m stream
$ add_host host03 color blue bridge m stream
$ add_host host04 color blue bridge m stream
$ add_host host05 color blue bridge m stream
$ add_host host06 color blue bridge m stream
$ add_host host07 color blue bridge m stream
$ add_host host08 color blue bridge m stream
$ add_host host09 color blue bridge m stream
$ add_host host10 color blue bridge m stream
$ add_host host11 color blue bridge m stream
```

Next, 12 hosts are added, each with a single port named m of type stream. In this example, the streaming hostport supports all four interfaces, i.e. interface a, b, c, and d. The position, shape and size of the hosts are not described, so they are the default single-cell-sized hosts, and NocStudio adds them at the first free position in the mesh, including the virtual node positions. The resulting display is shown below. There are three figures because three NoC layers are being built.

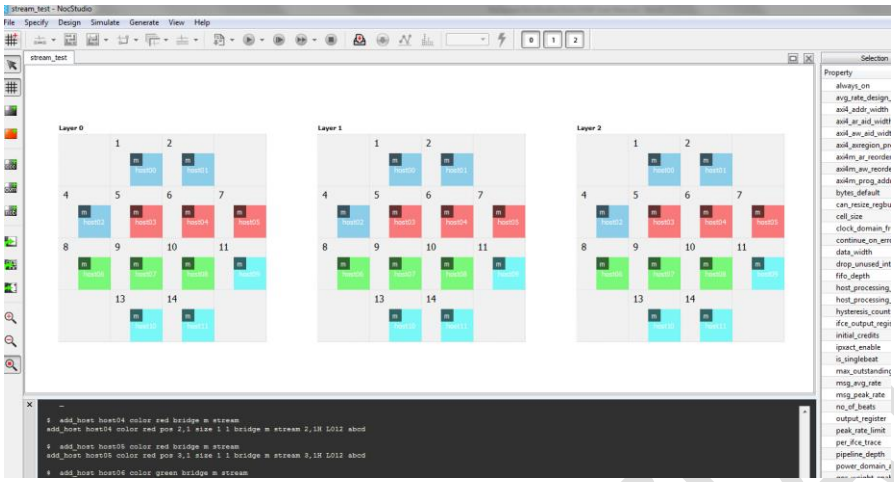


Figure 1: NoC display after adding 12 hosts with single streaming ports

```
$ add_alias hosts host00 host01 host02 host03 host04 host05 host06 host07
        host08 host09 host10 host11
```

Next, user creates an alias named hosts containing all 12 hosts.

```
$ add_traffic rates 0.025 0.01 hosts/m.a <-1 -1 4 64 0> hosts/m.a
$ add_traffic rates 0.01 0.01 hosts/m.b <-1 -1 4 64 1> hosts/m.b
$ add_traffic rates 0.01 0.01 host00/m.a host01/m.a <-1 -1 4 64 2> host10/m.b
        host11/m.b
$ map
```

Next, user adds traffic transaction between various hostports and interfaces. The first traffic transaction is from interface a of all hostports to interface a of all hostports. This traffic is given average rate 0.025, peak rate 0.01, number of beats 4, latency requirement 64 cycles, and is mapped to NoC layer 0. Note that each initiator sends 0.025 beat rate of traffic to a target. Therefore each intiator sends n x beat rate, and each target receives n x beat rate. The second traffic transaction is from interface b of all hostports to interface b of all hostports, and is mapped to NoC layer 1. The third traffic transaction is from interface a, of hostports host00/m and host01/m to interface b of hostports host10/m and host11/m on NoC layer 2. Finally, the **map** command maps the traffic.

```
$ add_traffic rates 0.025 0.01 hosts/m.b <-1 -1 1 64 0> hosts/m.a/m.a <-1 -1 4 64
        2> hosts/m.b
$ map
```

Next, more complex traffic is added. Here there is multi-hop traffic. The first hop is from interface **b** of all host m ports to interface **a** of all host m ports; with latency constraint of 64 cycles on NoC layer 0. The second hop is from interface **a** of all host m ports back to interface **b** of all host m ports;. **Because the alias hosts contains all hosts in the system, this traffic specification, when alias is expanded, will contain transactions from a host to itself; such transactions are omitted by NocStudio.**

Upon adding the multi-hop traffic transaction and mapping, NocStudio detects a protocol level deadlock and displays following message to the user.

Below, reporting the detected cyclic dependency

host01/m0.a.in <- host00/m0.a.out <- host00/m0.a.in <- host01/m0.a.out <- host01/m0.a.in

Error: Protocol level deadlock found when mapping flow src: host01/m.a.out, dest: host00/m.b.in, qos: 0. Please correct it

```
$ reset_traffic
$ add_traffic rates 0.025 0.01 hosts/m.a <-1 -1 4 64 0> hosts/m.a
$ add_traffic rates 0.01 0.01 hosts/m.b <-1 -1 4 64 1> hosts/m.b
$ add_traffic rates 0.01 0.01 host00/m.a host01/m.a <-1 -1 4 64 2> host10/m.b
        host11/m.b
$ map
```

Subsequently, the user can reset the traffic and add the first three traffic transactions again, which were mapped successfully. The resulting NocStudio display is shown below.

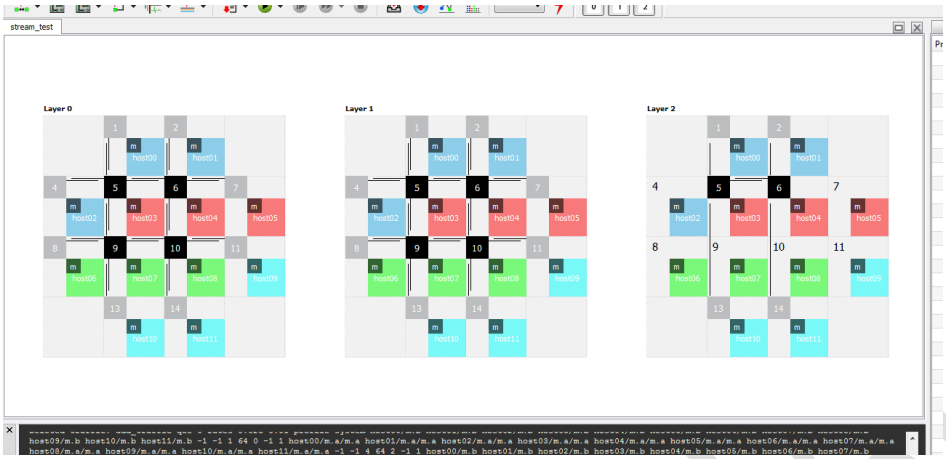


Figure 2: NoC display after adding traffic and mapping

```
$ tune_route 1 10 d
$ show_route hosts/m.* host00/m.* red
$ show_route host06/m.* host01/m.* blue
$ grid off
```

Next, optimizations are carried out. First, route optimizations are performed using the **tune_route** command. Next, the routes between all hosts' m ports to the host00 m port are highlighted in red, and the routes from host06/m to host01/m are highlighted in blue using the **show_route** command. The grid is also turned off for better visualization of routes. The resulting display is shown below.

Figure 3: NocStudio display highlighting certain routes in red and blue colors

```
$ tune_place
$ map
```

Next, the position of various hosts is optimized using the **tune_place** command; this should move the hosts that talk to each other more often closer to each other. In this example, most traffic is highly symmetric between all hosts; however, host00 and host01 communicate with host10 and host11 using an additional set of transactions in NoC layer 3, therefore these hosts are moved closer to each other when the cost function of the traffic is high enough. To illustrate, the rates between host00,01 and hosts10,11 were changed to 0.1. The resulting placement is below.

Figure 4: floorplan after tune_place

```
$ analyze_links p
$ tune_links p
```

Next, link bandwidths are analyzed (for the original traffic spec). NocStudio prints the bandwidth of various links on the console and also highlights the link bandwidth in the display in various colors. Subsequently, the link bandwidths are optimized to meet the traffic requirements using the **tune_links** command. The resulting NoC display after all these optimizations is shown below.

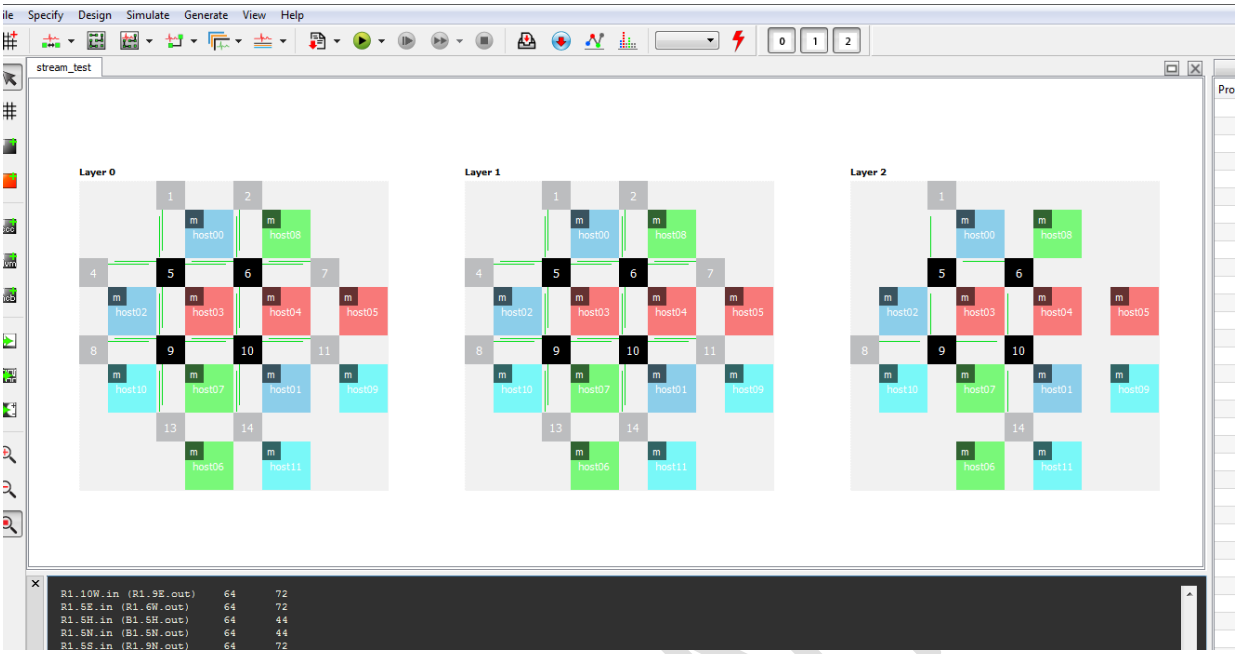


Figure 5: NocStudio display after link optimization

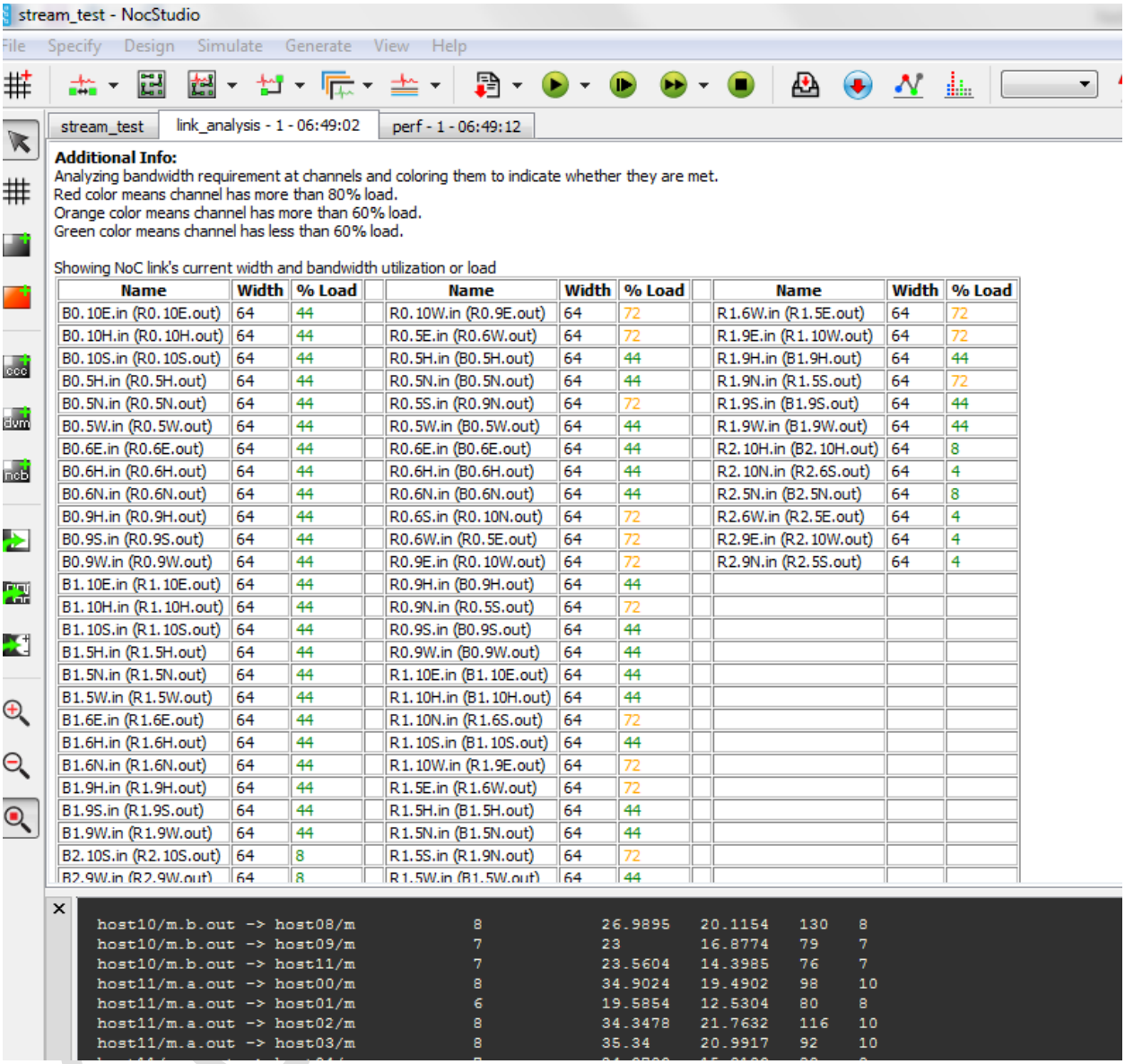


Figure 6 Results of link analysis showing loads

These windows are obtained by clicking o the link_analysis button at the top right of the GUI screen. The window is persistent and it is possible to do further changes to the design and come back to compare with this data stored as a timestamped tab on the GUI.

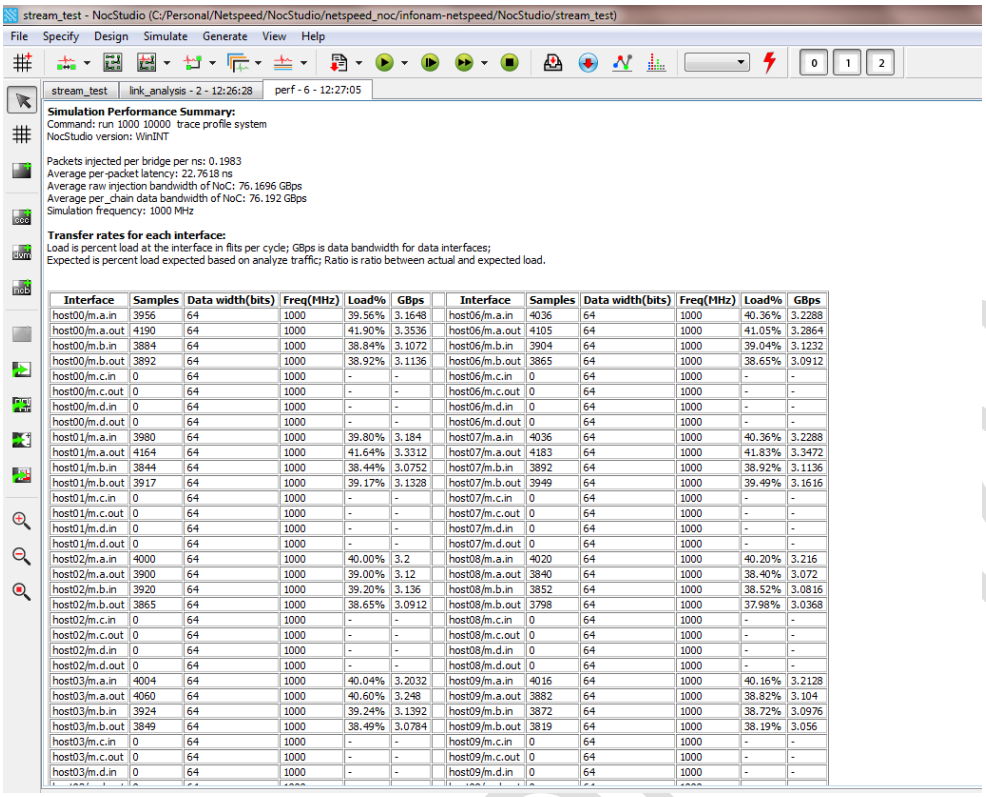
```
$ create_trace_files avg profile system

$ run 1000 10000 trace
```

Next, in a single command, user warms up the network by simulating it for 1000 cycles, then clears the statistics and run for another 10000 cycles.

Figure 7: Link congestion shown after dynamic simulation

Detailed performance statistics are obtained by clicking on the 'perf' button.

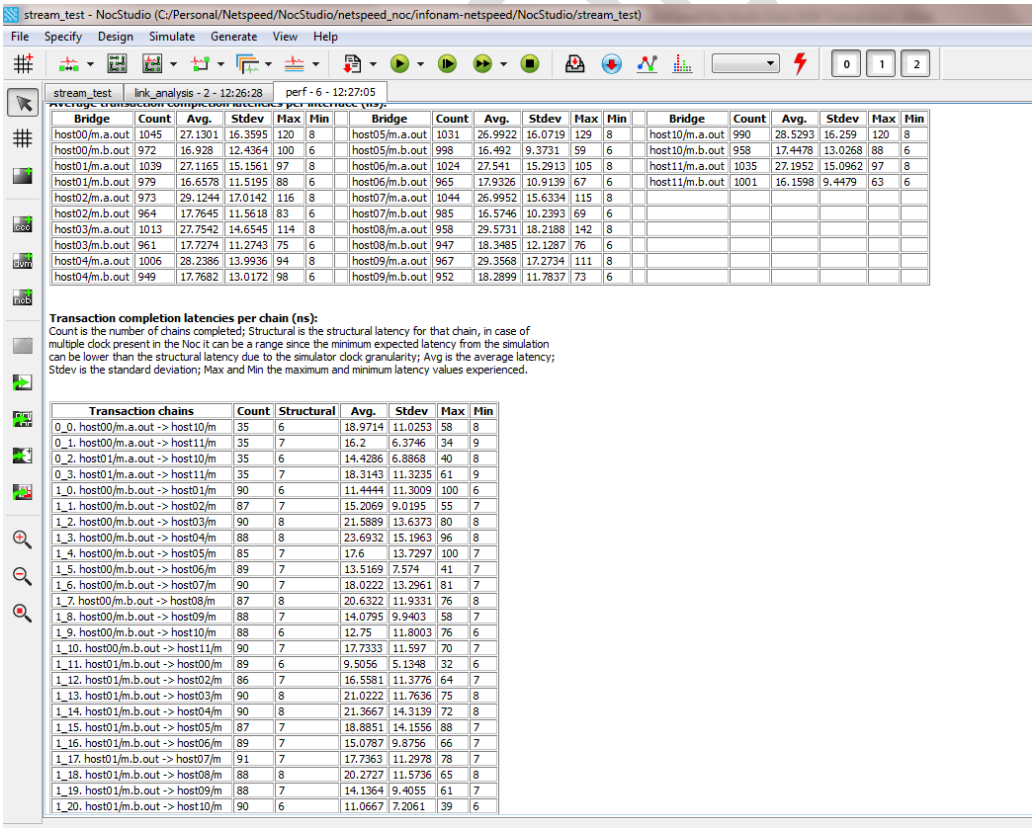


Simulation Performance Summary:
Command: run 1000 10000 trace profile system
NocStudio version: WinINT

Packets injected per bridge per ns: 0.1983
Average per-packet latency: 22.7618 ns
Average raw injection bandwidth of NoC: 76.1696 GBps
Average per_chain data bandwidth of NoC: 76.192 GBps
Simulation frequency: 1000 MHz

Transfer rates for each interface:
Load is percent load at the interface in flits per cycle; GBps is data bandwidth for data interfaces;
Expected is percent load expected based on analyze traffic; Ratio is ratio between actual and expected load.

Interface	Samples	Data width(bits)	Freq(MHz)	Load%	GBps	Interface	Samples	Data width(bits)	Freq(MHz)	Load%	GBps
host00/m.a.in	3956	64	1000	39.56%	3.1648	host06/m.a.in	4036	64	1000	40.36%	3.2288
host00/m.a.out	4190	64	1000	41.90%	3.3536	host06/m.a.out	4105	64	1000	41.05%	3.2864
host00/m.b.in	3884	64	1000	38.84%	3.1072	host06/m.b.in	3904	64	1000	39.04%	3.1232
host00/m.b.out	3892	64	1000	38.92%	3.1136	host06/m.b.out	3865	64	1000	38.65%	3.0912
host00/m.c.in	0	64	1000	-	-	host06/m.c.in	0	64	1000	-	-
host00/m.c.out	0	64	1000	-	-	host06/m.c.out	0	64	1000	-	-
host00/m.d.in	0	64	1000	-	-	host06/m.d.in	0	64	1000	-	-
host00/m.d.out	0	64	1000	-	-	host06/m.d.out	0	64	1000	-	-
host01/m.a.in	3980	64	1000	39.80%	3.184	host07/m.a.in	4036	64	1000	40.36%	3.2288
host01/m.a.out	4164	64	1000	41.64%	3.3312	host07/m.a.out	4183	64	1000	41.83%	3.3472
host01/m.b.in	3844	64	1000	38.44%	3.0752	host07/m.b.in	3892	64	1000	38.92%	3.1136
host01/m.b.out	3917	64	1000	39.17%	3.1328	host07/m.b.out	3949	64	1000	39.49%	3.1616
host01/m.c.in	0	64	1000	-	-	host07/m.c.in	0	64	1000	-	-
host01/m.c.out	0	64	1000	-	-	host07/m.c.out	0	64	1000	-	-
host01/m.d.in	0	64	1000	-	-	host07/m.d.in	0	64	1000	-	-
host01/m.d.out	0	64	1000	-	-	host07/m.d.out	0	64	1000	-	-
host02/m.a.in	4000	64	1000	40.00%	3.2	host08/m.a.in	4020	64	1000	40.20%	3.216
host02/m.a.out	3900	64	1000	39.00%	3.12	host08/m.a.out	3840	64	1000	38.40%	3.072
host02/m.b.in	3920	64	1000	39.20%	3.136	host08/m.b.in	3852	64	1000	38.52%	3.0816
host02/m.b.out	3865	64	1000	38.65%	3.0912	host08/m.b.out	3798	64	1000	37.98%	3.0368
host02/m.c.in	0	64	1000	-	-	host08/m.c.in	0	64	1000	-	-
host02/m.c.out	0	64	1000	-	-	host08/m.c.out	0	64	1000	-	-
host02/m.d.in	0	64	1000	-	-	host08/m.d.in	0	64	1000	-	-
host02/m.d.out	0	64	1000	-	-	host08/m.d.out	0	64	1000	-	-
host03/m.a.in	4004	64	1000	40.04%	3.2032	host09/m.a.in	4016	64	1000	40.16%	3.2128
host03/m.a.out	4060	64	1000	40.60%	3.248	host09/m.a.out	3882	64	1000	38.82%	3.104
host03/m.b.in	3924	64	1000	39.24%	3.1392	host09/m.b.in	3872	64	1000	38.72%	3.0976
host03/m.b.out	3849	64	1000	38.49%	3.0784	host09/m.b.out	3819	64	1000	38.19%	3.056
host03/m.c.in	0	64	1000	-	-	host09/m.c.in	0	64	1000	-	-
host03/m.c.out	0	64	1000	-	-	host09/m.c.out	0	64	1000	-	-
host03/m.d.in	0	64	1000	-	-	host09/m.d.in	0	64	1000	-	-



Bridge	Count	Avg.	Stdev	Max	Min	Bridge	Count	Avg.	Stdev	Max	Min	Bridge	Count	Avg.	Stdev	Max	Min
host00/m.a.out	1045	27.1301	16.3595	120	8	host05/m.a.out	1031	26.9922	16.0719	129	8	host10/m.a.out	990	28.5293	16.259	120	8
host00/m.b.out	972	16.928	12.4364	100	6	host05/m.b.out	998	16.492	9.3731	59	6	host10/m.b.out	958	17.4478	13.0268	88	6
host01/m.a.out	1039	27.1165	15.1561	97	8	host06/m.a.out	1024	27.541	15.2913	105	8	host11/m.a.out	1035	27.1952	15.0962	97	8
host01/m.b.out	979	16.6578	11.5195	88	6	host06/m.b.out	965	17.9326	10.9139	67	6	host11/m.b.out	1001	16.1598	9.4479	63	6
host02/m.a.out	973	29.1244	17.0142	116	8	host07/m.a.out	1044	26.9952	15.6334	115	8						
host02/m.b.out	964	17.7645	11.5618	83	6	host07/m.b.out	985	16.5746	10.2393	69	6						
host03/m.a.out	1013	27.7542	14.6545	114	8	host08/m.a.out	958	29.5731	18.2188	142	8						
host03/m.b.out	961	17.7274	11.2743	75	6	host08/m.b.out	947	18.3485	12.1287	76	6						
host04/m.a.out	1006	28.2386	13.9936	94	8	host09/m.a.out	967	29.3568	17.2734	111	8						
host04/m.b.out	949	17.7682	13.0172	98	6	host09/m.b.out	952	18.2899	11.7837	73	6						

Transaction completion latencies per chain (ns):
Count is the number of chains completed; Structural is the structural latency for that chain, in case of multiple clock present in the Noc it can be a range since the minimum expected latency from the simulation can be lower than the structural latency due to the simulator clock granularity; Avg is the average latency; Stdev is the standard deviation; Max and Min the maximum and minimum latency values experienced.

Transaction chains	Count	Structural	Avg.	Stdev	Max	Min
0_0. host00/m.a.out -> host10/m	35	6	18.9714	11.0253	58	8
0_1. host00/m.a.out -> host11/m	35	7	16.2	6.3746	34	9
0_2. host01/m.a.out -> host10/m	35	6	14.4286	6.8868	40	8
0_3. host01/m.a.out -> host11/m	35	7	18.3143	11.3235	61	9
1_0. host00/m.b.out -> host01/m	90	6	11.4444	11.3009	100	6
1_1. host00/m.b.out -> host02/m	87	7	15.2069	9.0195	55	7
1_2. host00/m.b.out -> host03/m	90	8	21.5889	13.6373	80	8
1_3. host00/m.b.out -> host04/m	88	8	23.6932	15.1963	96	8
1_4. host00/m.b.out -> host05/m	85	7	17.6	13.7297	100	7
1_5. host00/m.b.out -> host06/m	89	7	13.5169	7.574	41	7
1_6. host00/m.b.out -> host07/m	90	7	18.0222	13.2961	81	7
1_7. host00/m.b.out -> host08/m	87	8	20.6322	11.9331	76	8
1_8. host00/m.b.out -> host09/m	88	7	14.0795	9.9403	58	7
1_9. host00/m.b.out -> host10/m	88	6	12.75	11.8003	76	6
1_10. host00/m.b.out -> host11/m	90	7	17.7333	11.597	70	7
1_11. host01/m.b.out -> host00/m	89	6	9.5056	5.1348	32	6
1_12. host01/m.b.out -> host02/m	86	7	16.5581	11.3776	64	7
1_13. host01/m.b.out -> host03/m	90	8	21.0222	11.7636	75	8
1_14. host01/m.b.out -> host04/m	90	8	21.3667	14.3139	72	8
1_15. host01/m.b.out -> host05/m	87	7	18.8851	14.1556	88	7
1_16. host01/m.b.out -> host06/m	89	7	15.0787	9.8756	66	7
1_17. host01/m.b.out -> host07/m	91	7	17.7363	11.2978	78	7
1_18. host01/m.b.out -> host08/m	88	8	20.2727	11.5736	65	8
1_19. host01/m.b.out -> host09/m	88	7	14.1364	9.4055	61	7
1_20. host01/m.b.out -> host10/m	90	6	11.0667	7.2061	39	6

Figure 8: Performance Statistics after simulation

```
$ gen_ip
```

Clicking on the gen-ip button generates the RTL and other IP files.

1.2.1 Comparing with Map_opt

The same configuration has been built with map_opt below with a change to the traffic specifications, whereby explicit mapping of traffic to layers 0,1,2 has been removed.

```
$ add_traffic rates 0.025 0.01 hosts/m.a <-1 -1 4 64 0> hosts/m.a
$ add_traffic rates 0.01 0.01 hosts/m.b <-1 -1 4 64 1> hosts/m.b
$ add_traffic rates 0.01 0.01 host00/m.a host01/m.a <-1 -1 4 64 2> host10/m.b
        host11/m.b
$ map_opt 1 5 d
```

From the previous example, the network was seen to have been congested. In this example, map_opt is given 8 layers to use, but ends up using 4. The resulting network is different, with traffic being spread around layers in a different way.

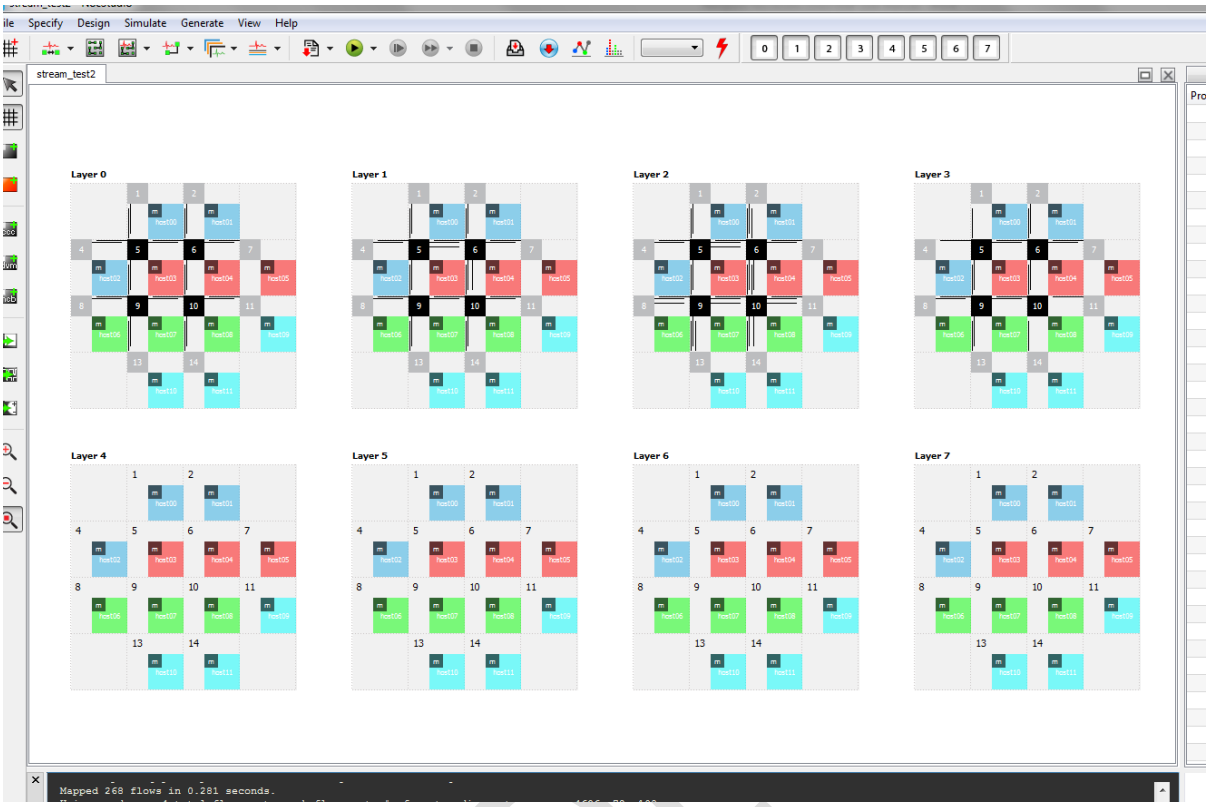


Figure 9 NoC generated using map_opt

1.2.2 Enabling Regbus

Regbus is a private layer used for accessing bridge and router registers. It has one single master device known as the Regbus Master which communicates to the regbus layer through a Regbus master bridge. The regbus master bridge speaks AXI-lite on the host side, and streaming protocol on the NoC side. Regbus is enabled by using the new_mesh command as shown below:

```
$ prop_default cell_size 8
$ prop_default data_width 64
$ new_mesh 4 4 3 stream_test regbus_enabled
$ mesh_prop virtual_ok yes
$ gui on regbus
```

Using the same example as before, but with one fewer host,

```
$ add_host host00 color blue bridge m stream
$ add_host host01 color blue bridge m stream
$ add_host host02 color blue bridge m stream
$ add_host host03 color blue bridge m stream
$ add_host host04 color blue bridge m stream
$ add_host host05 color blue bridge m stream
$ add_host host06 color blue bridge m stream
$ add_host host07 color blue bridge m stream
$ add_host host08 color blue bridge m stream
$ add_host host09 color blue bridge m stream
$ add_host host10 color blue bridge m stream
```

```
$ add_alias hosts host00 host01 host02 host03 host04 host05 host06 host07
        host08 host09 host10
$ add_traffic rates 0.025 0.01 hosts/m.a <-1 -1 4 64 0> hosts/m.a
$ add_traffic rates 0.01 0.01 hosts/m.b <-1 -1 4 64 1> hosts/m.b
$ add_traffic rates 0.01 0.01 host00/m.a host01/m.a <-1 -1 4 64 2> host09/m.b
        host10/m.b
$ map_opt 1 5 d
```

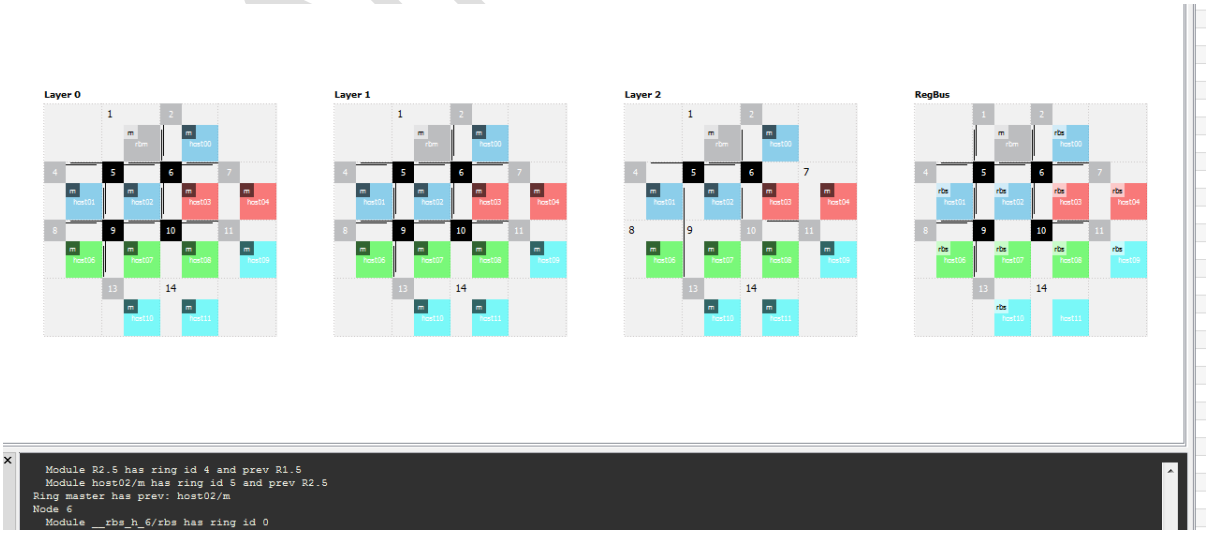


Figure 10 Noc layers with regbus enabled

The figure above shows the Regbus layer in the rightmost panel. A regbus slave bridge is generated for each node that has a streaming bridge or router on it. The noc protocol used for regbus is the same for the other layers. However usage of the master bridge should be by a privileged AXI-lite host.

The regbus host can be moved to a given position on the grid by the following command sequence:

```
host_prop rbm lock off

move_host rbm 10
```

2670 Seely Avenue
Building 11
San Jose CA 95134
www.netspeedsytems.com