CHAIN® Network Performance Closure and Verification User Guide

(CHAIN®works 2.2)

License

Disclaimer

Table of Contents

Performance Closure using Silistix NoC Performance Validation (NPV)

Overview

The Silistix NoC Performance Validation (NPV) feature augments and simplifies the existing EDA flow for System on Chip (SoC) design. Essentially, NPV provides an easier path to performance closure and sign-off compared to traditional methods. Performance closure is the process of verifying, at various points during the chip development flow, that the requirements specified in a CSL design are met, eventually leading to performance signoff prior to IC tape-out.

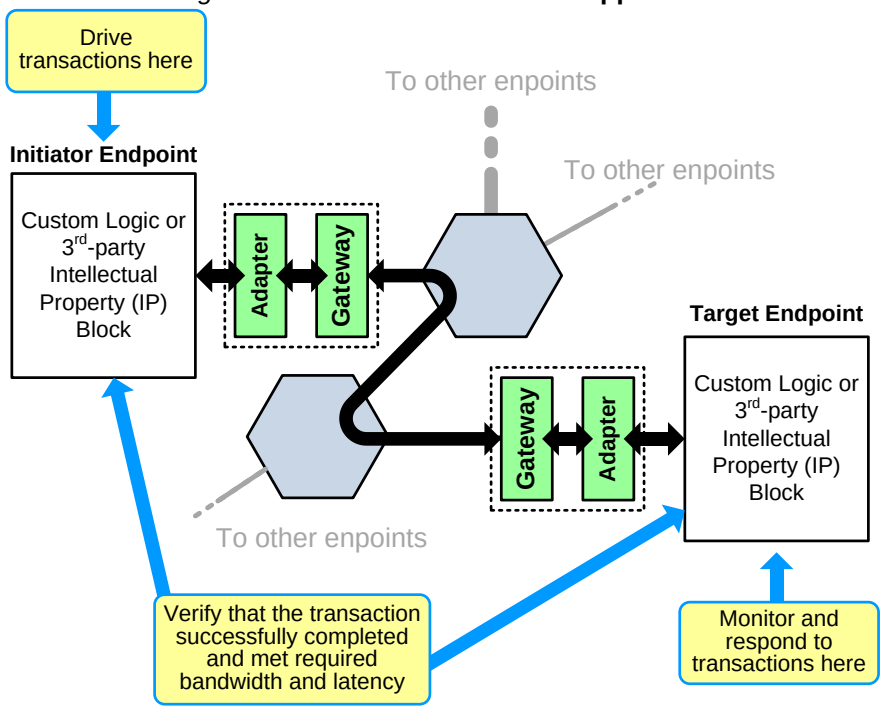
NPV generates worst case data traffic patterns that stress the Silistix network in order to guarantee that the design meets the specified bandwidth and latency requirements. Perhaps the easiest way to understand the benefits of NoC Performance Validation (NPV) is to compare the challenges when verifying an on-chip network with and without NPV.

The Challenge without NPV

First, consider the example design shown in Figure 1 that uses traditional verification techniques. In such an environment, the system designer or test engineer must laboriously create a testbench or test harness that …

- drives a transaction over the network using the initiator's native protocol,
- monitors the transaction at the target,
- correctly responds to the transaction using the target's native protocol,
- verifies that both the initiator and target have sent or received the correct data,
- drives network traffic with worst-case patterns, and
- checks that the network meets the application's bandwidth and latency requirements.

Figure 1: **Traditional Testbench Approach**



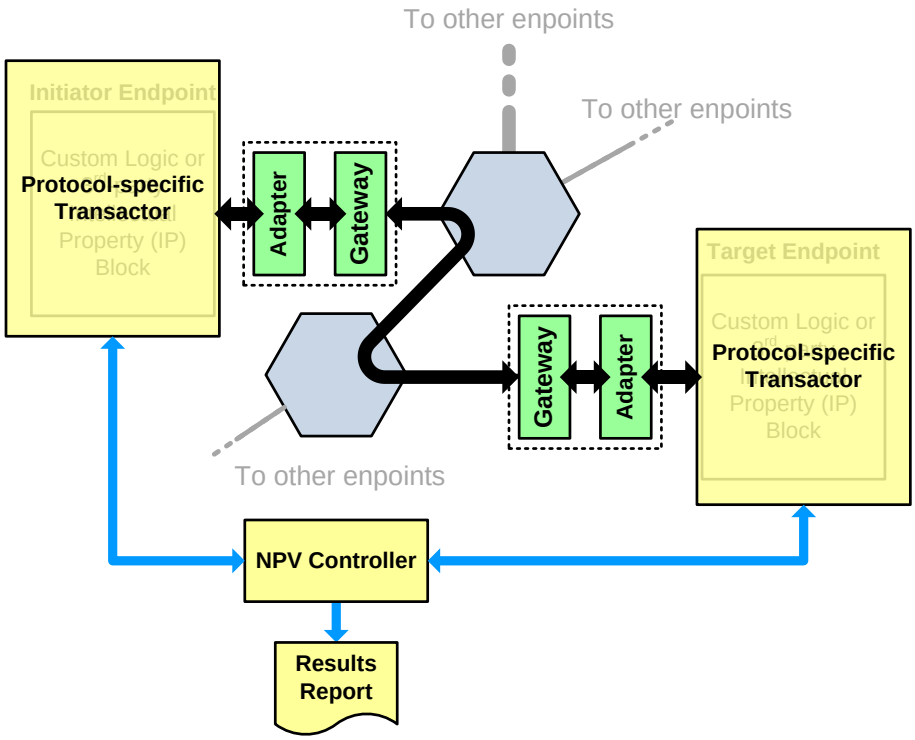
Such a testbench quickly becomes complex; a major design task by itself. Furthermore, the entire system design must be completed, including the IP blocks at both ends of the network, before validating the network. The testbench must also exercise both the initiator and target endpoints in their native protocol in order to generate worst-case network traffic. Finally, the testbench must include a means to check for successful transactions and check that the application's bandwidth and latency requirements are satisfied.

NPV Simplifies Performance Closure and Network Testing

Compare the traditional approach to the simpler, more automated NoC Performance Validation (NPV) approach shown in Figure 2. NPV automatically generates a test environment to verify the performance of a Silistix Network on Chip (NoC). The automated NPV test harness replaces the user-defined endpoints on the network with pre-built test transactors. The transactors connect to the NPV controller that automatically generates real-world network traffic. The controller also monitors each transaction and produces a report that shows the resulting bandwidth and latency performance for each connection. The NPV environment aids performance closure and augments the performance analysis within CHAINarchitect. Furthermore, NPV eliminates the need to have all endpoints in place before validating network performance.

The NPV testbench generates and consumes network traffic that is representative of the actual system when the system is in a particular mode of operation. CHAINarchitect or the CSL Compiler generates traffic files that are read by the NPV test harness. The NPV harness converts the test data into signals that stimulate data flow between adapters, mimicking transactions by the actual endpoint hardware and software.

Figure 2: **Automated NoC Performance Validation (NPV) Environment**



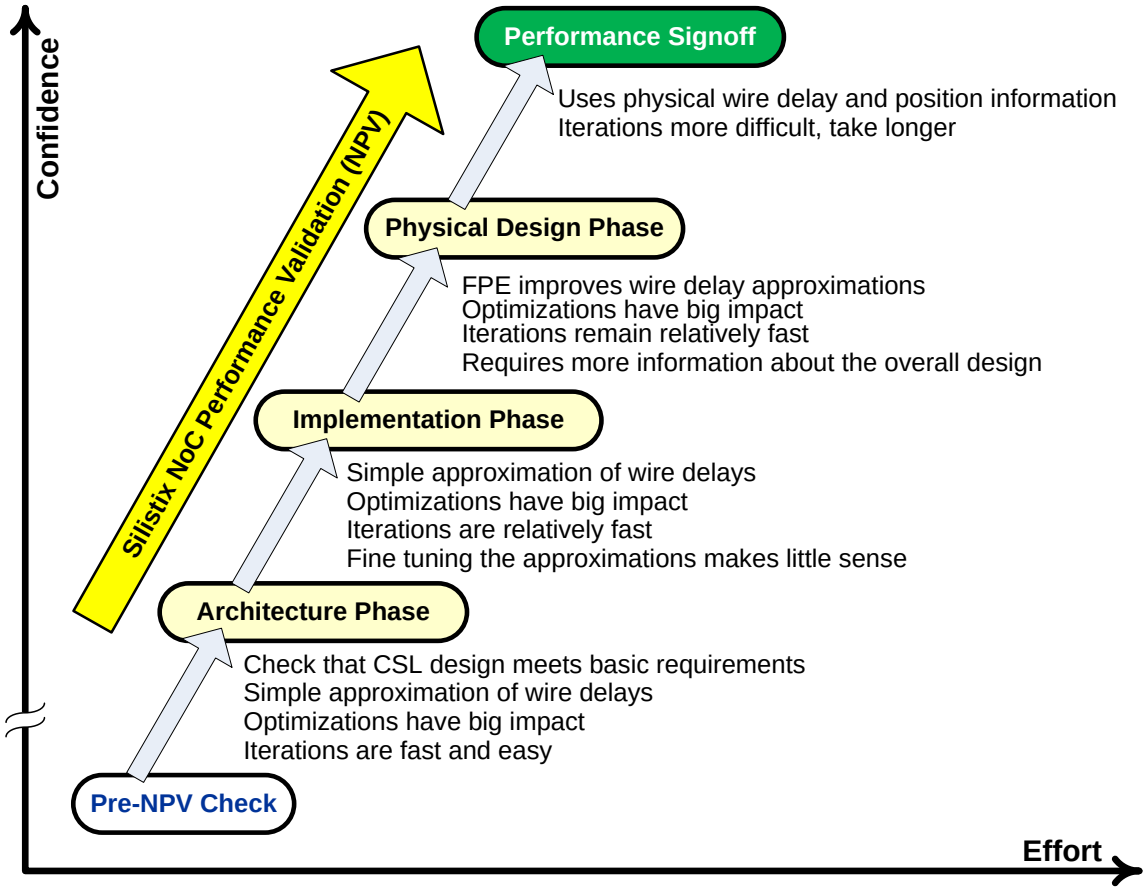
NPV provides a predictable view of the NoC during the entire development process. Network performance can be re-verified at each step of the development process, increasing the accuracy of the final result. This predictability increases confidence in the implemented results for the system.

[THIS PAGE INTENTIONALLY LEFT BLANK]

NPV Performance Sign-Off Process

Using NPV, the performance signoff process includes multiple steps, as shown in Figure 3. Before beginning the NPV validation process, check the performance requirements specified in the CSL file using CHAINarchitect or CSL Compiler. This pre-NPV check uses static modeling and simple wire delay approximations to check performance. Design iterations at this point are fast and easy.

Figure 3: **Silistix NPV Performance Closure Process**



The first step in the NPV validation process is called the Architecture Phase and provides a first level of confidence before progressing with the design. Here, the CSL design is checked against the specified performance requirements using a cycle-accurate simulation model, but using a simple approximation of wire delays.

The Implementation Phase is the second phase of the performance signoff process. This phase integrates more-detailed physical design parameters into the simulation model, resulting in a much higher degree of confidence before entering the physical design phase of the overall chip design. The physical information leverages the First Placement Estimator (FPE) tool within the Silistix CHAINarchitect software.

After successfully completing the Implementation Phase, there is sufficient confidence to begin the lengthy synthesis, verification and chip assembly phases of the design. Changes after this point become significantly more difficult and time consuming.

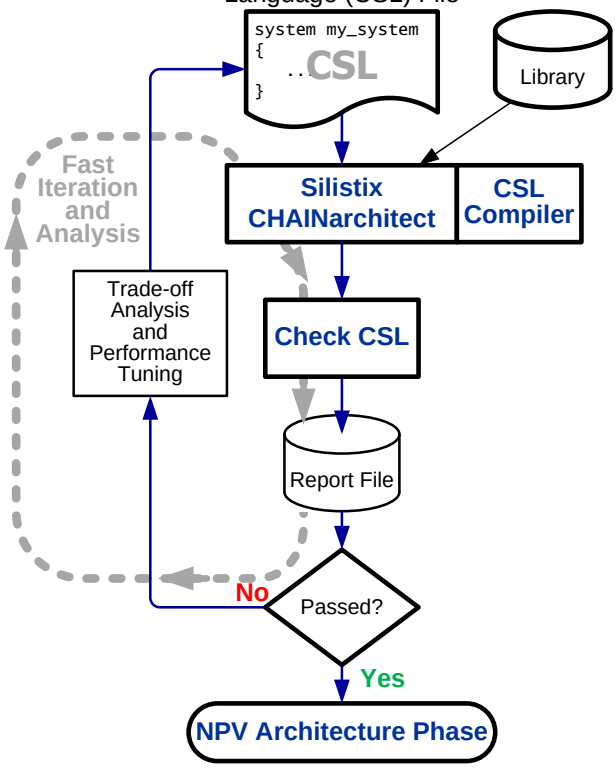
The third and final step in the NPV performance signoff process is the Post Layout Phase. This phase requires a fully laid out chip design. The extracted post-layout information ultimately provides the confidence that the resulting physical silicon meets the specified bandwidth and latency requirements.

Pre-NPV Check

Before starting NPV verification, ensure that the CSL file meets the specified performance requirements for the application, as shown in Figure 4. From within CHAINarchitect, run the Check operation and verify that no errors or performance-related warnings are reported. For more information using CHAINarchitect, please refer to the following document.

Building and Analyzing On-Chip Networks using CHAINarchitect
(chainarchitect-user-guide.pdf)

Figure 4: **Pre-NPV Check**



To process the design with CSL Compiler, use the following command line options.

```
cslc <csl_source_file>.csl
```

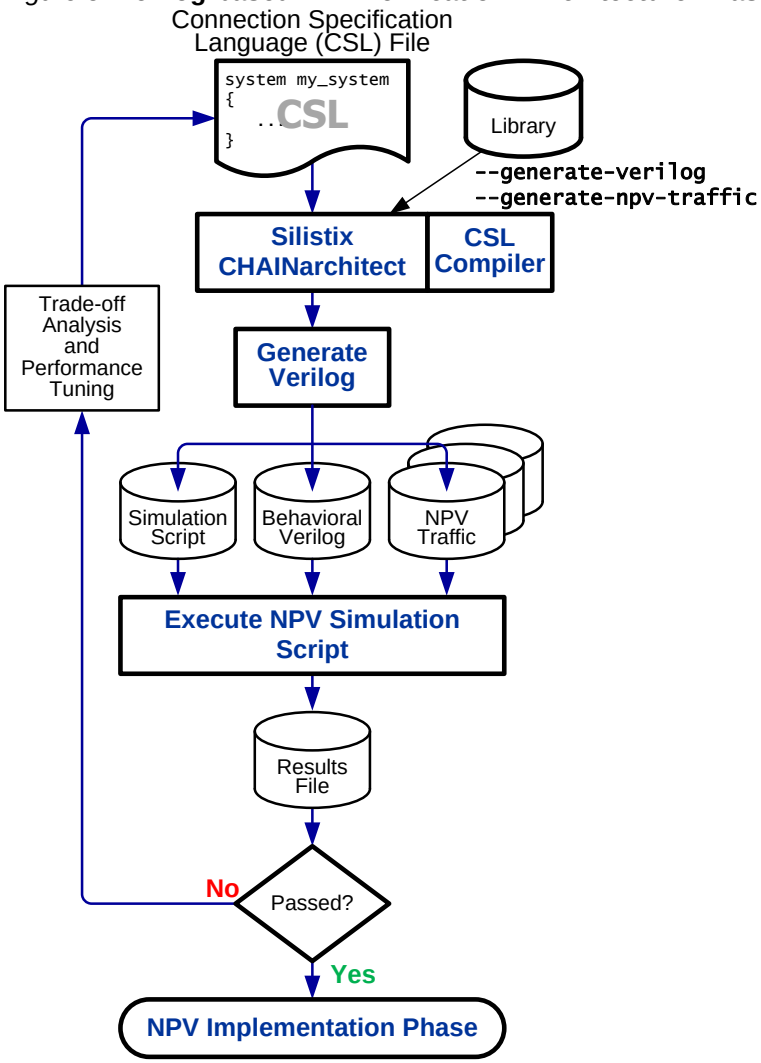
The CHAINarchitect Check operation provides a fast and relatively accurate performance check using a static timing model and a simple approximation of wire delay. If the Check operation is successful, proceed to NPV validation, starting with the Architecture Phase.

Architecture Phase

Figure 5 shows the Verilog-based NPV flow during the architectural phase. Using the proper settings, CHAINarchitect or CSL Compiler generates the following output files.

- Behavioral Verilog model of the Silistix network. A SystemC model is a separate option.
- Simulator-specific script file, `simulate.sh`
- Various `.npv` files, one per `mode` defined in the CSL source file. If no modes are specified in the CSL design, then only `default.npv` is created.

Figure 5: **Verilog-based NPV Verification: Architecture Phase**



To generate NPV files, include the **`--generate-npv-traffic`** CSL Compiler option. By default, the generated NPV files appear in the *<system_name>*`/npv` subdirectory. CSL Compiler generates one NPV file per `mode` statement declared in the source CSL file.

CHAINarchitect Option Settings

As shown in Figure 10 on page 18, set the following options in the Generate Verilog section.

```
--generate-report --generate-npv-traffic --generate-verilog
```

CSL Compiler Command Line

To process the design with CSL Compiler, use the following command line options.

```
cslc <csl_source_file>.csl --generate-report --generate-npv-traffic  \
     --generate-verilog
```

Run Simulation Script on Each NPV File

After generating the NPV files, execute the simulation script for each NPV file. See "Executing an NPV Test" on page 20 for more information.

Examine the summary at the end of the simulation log file for each NPV. Verify that the mode meets the current requirements. See "Example Report File Output" on page 23 for more information.

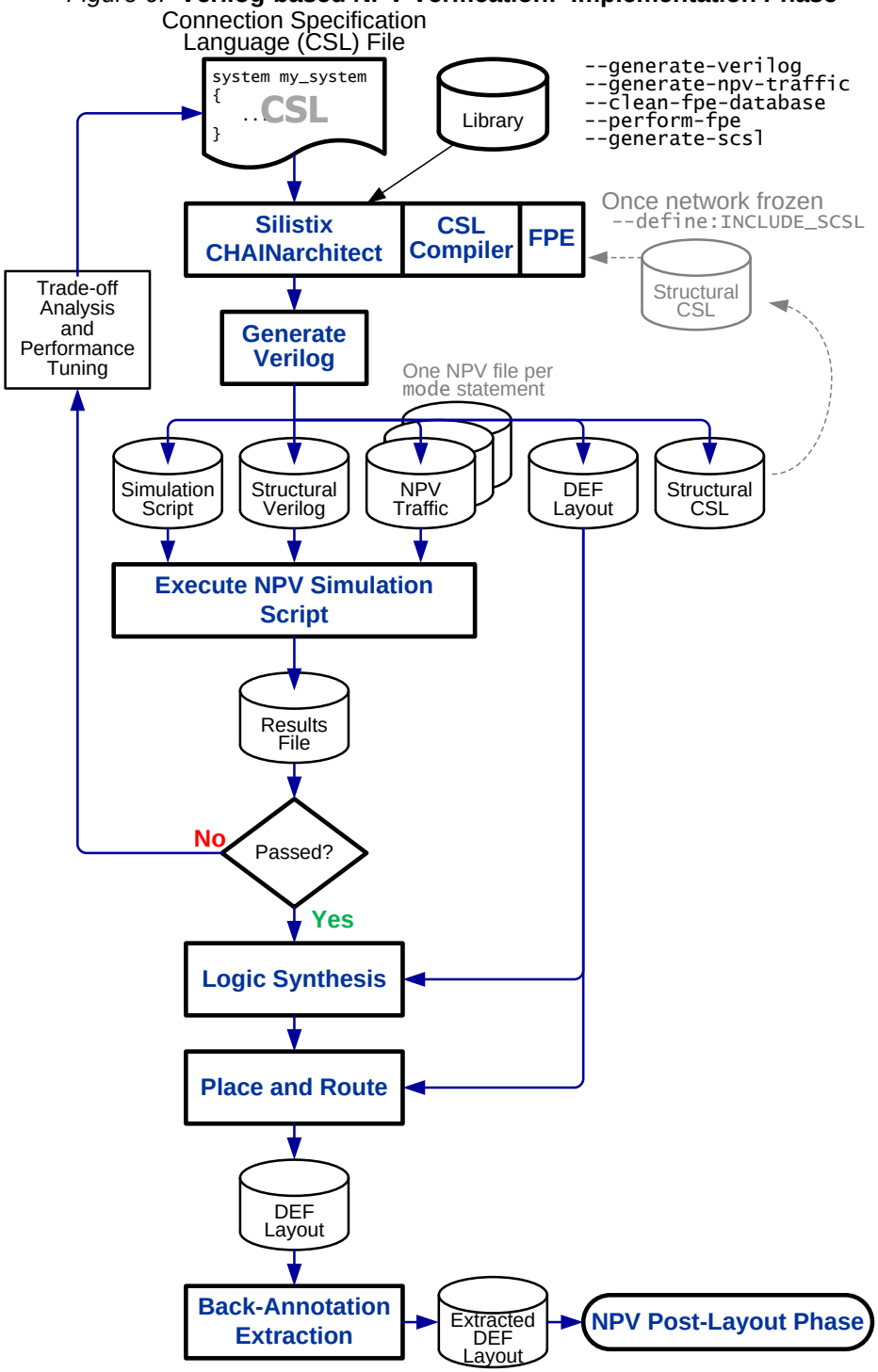
If all NPV simulations pass, continue to the Implementation Phase.

If an NPV simulation does not pass, then see Option 1 under "What If NPV Results Do Not Pass?" on page 17 for more information.

Implementation Phase

The Implementation Phase, shown in Figure 6, includes the placement-related affects on timing. The First Placement Estimator (FPE) tool uses pre-placement information to provide more realistic wire delay times. Because wire delay may have significant impacts on the actual latency and bandwidth, running NPV on a network ***after*** performing FPE is a mandatory step to guaranteed performance closure.

Figure 6: **Verilog-based NPV Verification: Implementation Phase**



Freezing a Network

After creating an acceptable initial physical placement that meets all the specified requirements, freeze the network so that it can be annotated and re-verified against those requirements later. To freeze the network, perform the following steps.

1. Generate a structural CSL file by including the `--generate-scsl` compiler option.
2. In the original CSL source file, add the following lines just inside the closing brace of the system declaration to reference the structural CSL file, as shown in Figure 7.

Figure 7: **Method to Include Structural CSL File**

```
system <system_name> {

    ...

    #if defined(INCLUDE_SCSL)
    #include "<system_name>/scsl/<scsl file name>.csl"
    #endif
}  // end system
```

3. To use the frozen structural CSL file, add the following command line option.

```
cslc <filename> <options> --define:INCLUDE_SCSL
```

CHAINarchitect Option Settings

As shown in Figure 10 on page 18, set the following options CSL Compilers.

FPE section

```
--generate-report --clean-fpe-database --perform-fpe  \
     --generate-scsl: <structural_csl_file>.csl
```

Generate Verilog section

```
--generate-report --generate-npv-traffic  \
     --generate-verilog --define:INCLUDE_SCSL
```

CSL Compiler Command Line

To process the design with CSL Compiler from the command line, use the following options.

First Placement Estimation (FPE)

```
cslc <csl_source_file>.csl --generate-report  --clean-fpe-database  \
     --perform-fpe --generate-scsl:<structural_csl_file>.csl
```

Generate NPV

```
cslc <csl_source_file>.csl --generate-report --generate-npv-traffic  \
     --generate-verilog --define:INCLUDE_SCSL
```

Run Simulation Script on Each NPV File

After generating the NPV files, execute the simulation script for each NPV file. See "Executing an NPV Test" on page 20 for more information.

Examine the summary at the end of the simulation log file for each NPV. Verify that the mode meets the current requirements. See "Example Report File Output" on page 23 for more information.

If all NPV simulations pass, continue to the Preparing for Post-Layout Phase.

Preparing for Post-Layout Phase

Any placement information generated by FPE is saved in a DEF file. This information is useful for Preparing for Post-Layout Phase. The resulting `*.def` file is saved under the **constraints** subdirectory.

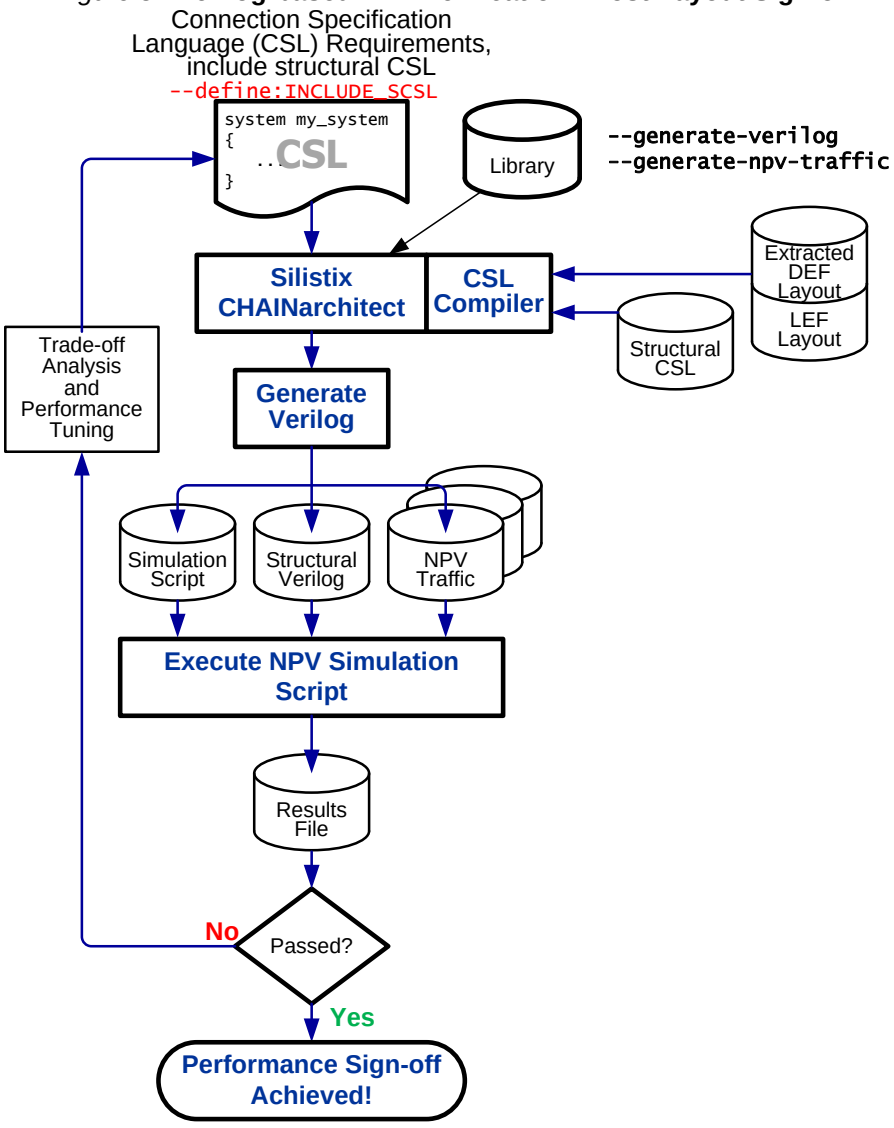
Use the DEF file during logic synthesis and placement and routing. The ultimate result is a post-layout DEF file. CHAINworks provides a script for your EDA flow to extract DEF files from your laid out design that can be read directly into CSL Compiler for the generation of timing annotated networks for NPV performance closure and signoff.

Continue to "Post Layout Phase" on page 16.

Post Layout Phase

Although FPE generally provides good estimates of wire lengths, the actual location of some network components likely changed during final layout. To complete the performance sign-off process, extract actual placement and wire delay information from the final design and feed it back through the system, as shown in Figure 8. This back-annotation step produces highly accurate timing information to guarantee final performance closure.

Figure 8: **Verilog-based NPV Verification: Post-Layout Sign-off**



Run Simulation Script on Each NPV File

After generating the NPV files, execute the simulation script for each NPV file. See "Executing an NPV Test" on page 20 for more information.

Examine the summary at the end of the simulation log file for each NPV. Verify that the mode meets the current requirements. See "Example Report File Output" on page 23 for more information.

If all NPV simulations pass, then the design achieved performance closure. Proceed to silicon fabrication.

What If NPV Results Do Not Pass?

If the application passes all NPV validation steps but fails to meet the original design requirement during the Post-Layout Phase, there are two options.

1. **Modify the bandwidth or latency requirement(s) in the CSL source file.**

   Examine the cases of negative slack described in the report file. Determine whether the design can tolerate slackening the bandwidth or latency requirements described in the CSL source file. The reported negative slack may be negligible or the original requirement might be over-specified or unrealistic.

2. **Modify structural CSL, re-generate Verilog, re-synthesize, re-layout, and iterate again**

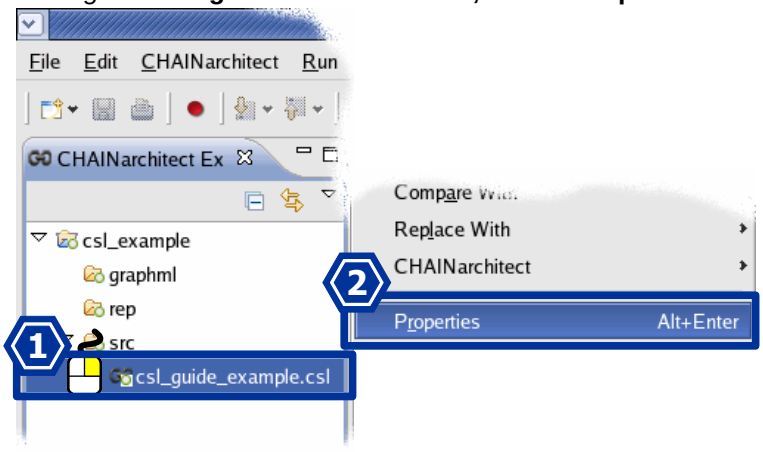
   The other option is to modify the structural CSL, which is more complex. This also requires that the design be re-processed back through enire the design flow, which is a time consuming process.

Generating NPV Files in CHAINarchitect

To generate NPV files, include the `--generate-npv-traffic` CSL Compiler option for Generate Verilog, as described below.

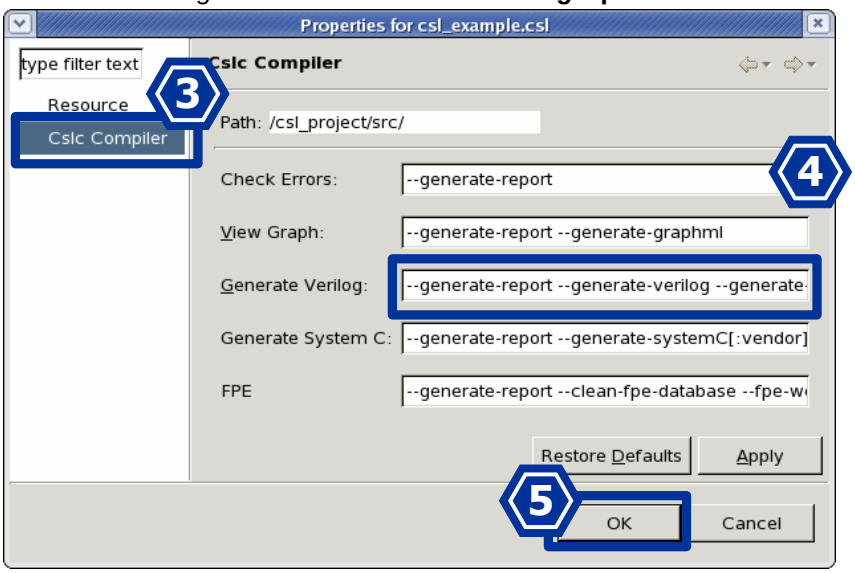
The Silistix CSL Compiler underlies all CHAINarchitect operations. To set specific options for the CSL Compiler software, follow the steps outlined after Figure 9. See "NPV-Related CSL Compiler Options" on page 21 for a list of all NPV options.

Figure 9: **Right-click on CSL File, Select Properties**



1. As shown in Figure 9, expand the project tree to reveal the CSL file. Right-click on the CSL file name.
2. Select **Properties** from the resulting pop-up menu.

Figure 10: **Set Generate Verilog Options**



3. As shown in Figure 10, click **Cslc Compiler** to reveal the available option settings.
4. Modify the **Generate Verilog** options using one of the following two option settings. Be sure to include the `--generate-npv-traffic option`.
   - Before First Placement Estimator (FPE)
   - After First Placement Estimator (FPE)
5. When finished, click **OK** to save the new option settings.

Before First Placement Estimator (FPE)

Use these options without generating a First Placement Estimation.

```
--generate-verilog --generate-npv-traffic
```

After First Placement Estimator (FPE)

Use these options to generate a First Placement Estimation. The output design includes any pipelatch components required to guarantee bandwidth across the chip.

```
--clean-fpe-database [other FPE options] --perform-fpe --generate-verilog \
--generate-npv-traffic
```

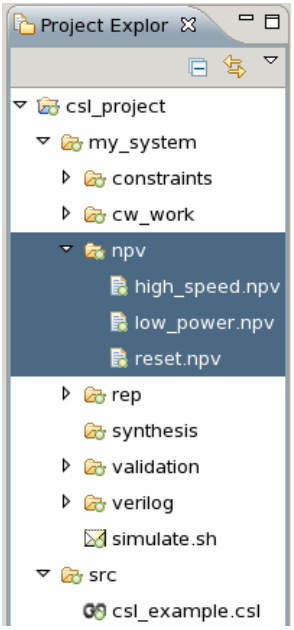
Files Created

After generating Verilog, CHAINarchitect or the CSL Compiler generates a Simulation Shell Script and one or more NPV Files. The resulting files appear in the CHAINarchitect Project Explorer, as shown in Figure 11.

Simulation Shell Script

The simulation shell script, `simulate.sh`, appears under the **verilog** project sub-directory.

Figure 11: **NPV File Directory**



NPV Files

CHAINarchitect or the CSL Compiler creates one NPV file for each `mode` statement in the CSL source file. If no `mode` statements are used, then only one file is generated, called `default.npv`. The NPV files appear under the **npv** project sub-directory.

Simulator Setup

The Simulation Shell Script requires a supported Supported Simulator and a $SILISTIX_HOME Environment Variable.

Supported Simulators

The Silistix NPV simulation script supports the following simulators. Add the appropriate option settings shown in Table 1 to the simulator-specific script command line.

- Mentor Graphics® ModelSim® or Questa® (vsim)
- Synopsys® VCS®
- Cadence® NC-Verilog®
- SystemC Simulator

`$SILISTIX_HOME` Environment Variable

The simulator shell script requires that the `$SILISTIX_HOME` variable be set, which is also required to invoke the CHAINarchitect or CSL Compiler software.

Executing an NPV Test

Run the Simulation Shell Script from a console window. If more than one mode is specified in the CSL file, run the shell script for each `*.npv` file.

Run the script from the within the CSL system director. By default, the script output appears on the standard output. Generally, redirect the log file output to a specified output file. An example command line appears below for the default mode.

```
cd my_system

./simulate.sh --batch --vectors npv/default.npv --logfile default.log
```

The simulator script supports the options listed in Table 1.

Table 1: **Simulation Script Options**

Option	Description	Alias
`--structural-hardmacros`	Use structural hardmacro models in simulations	`-struct --structural`
`--behavioral-hardmacros`	Use behavioral hardmacro models in simulations	`-behav --behavioral`
`--logfile <filename>`	Output simulator transcript to filename	`-l`
`--batch`	Run simulator on the command line and exit on completion	`-c`
`--gui`	Run the simulator with graphical user interface	`-gui`
`--verbose`	Run the simulator in verbose mode	
`--quiet`	Run the simulator in quiet mode	
`--vectors <filename>`	Use the vectors from filename for the simulation (multiple -vectors <filename> can be used)	
`--all-vectors`	Use all vectors found for the current project	
`--ncverilog`	Use CadenceNC-Verilog simulator	`-ncv --ncv`
`--vsim`	Use Mentor Graphics ModelSim or Questa simulator	`-vsim --modelsim --questasim`
`--vcs`	Use Synopsys VCS simulator	`-vcs`
`--work`	Set ModelSim work directories	
`--systemc`	Use SystemC simulator	
`--post_synthesis`	Use post synthesis netlists and back-annotated SDF	`-sdf --backannotate`
`--compile-arguments`	Pass arguments to the compile phase of the simulation. e.g. --cargs "<option1> <option2>..." typically things like +define+	`-cargs --cargs`
`--run-arguments`	Pass commands to simulation when it is run. e.g. --rargs "<option1> <option2>..." typically things like -gui	`-rargs --rargs`
`--run-commands`	Pass commands to simulation once it is loaded. e.g. --rcom "<option1> <option2>..." typically tcl commands such as run -a	`-rcom`
`--help`	Display this help text	`-h -help`

NPV-Related CSL Compiler Options

Table 2 lists the NPV-related options for CSL Compiler.

Table 2: **CSL Compiler Options for NPV**

Option	Description	Alias
`--define:INCLUDE_SCSL`	Include the referenced structure CSL file	
`--generate-npv-traffic[:<dir>]`	Generate NPV traffic files.	`-npv`
`--npv-threshold:<tolerance>`	NPV bandwidth tolerance.	`-npvt`
`--npv-iterations:<n>`	NPV minimum traffic iteration count for initiators. Default is 10.	`-npvi`
`--fast-npv-traffic`	Generate NPV traffic for faster (less accurate) simulation	
`--distribute-npv-traffic`	Distribute NPV traffic	
`--randomize-npv-traffic`	Randomly stagger NPV traffic	
`--perform-fpe[:<dbpath>]`	Perform First Placement Estimation (FPE)	`-fpe`
`--clean-fpe-database`	Clean FPE Database	`-cfpe`
`--fpe-center-gateways`	Center Network Gateways within Soft IP Block	`-fpcgw`
`--fpe-weights:w1_w2_w3_w4`	Set FPE Weights for the various inter-block connections	`-wt:`
`--fpe-sub-block-size:<size>`	Sets the FPE Sub-block Size, in µm. Default is 30um.	`-sbs:`
`--fpe-wrap-edges`	Wrap Soft IP Block Edges	`-wse`
`--no-pipelatch-insertion`	No Pipelatch Insertion	`-nopl`
`--generate-scsl[:filename]`	Generate Structural CSL File	`-gf`
`--generate-verilog`	Generate DEF File	`-gv`

Example NPV Stimulus File

Figure 12 shows an example NPV stimulus file generated automatically by CHAINarchitect or CSL Compiler. A separate NPV file is generated for every `mode` statement in the CSL source file, plus a `default.npv` for any connections that are not declared within a `mode` statement.

Figure 12: Example NPV Stimulus File

```
#
# CHAINworks system-level test pattern file
# Cslc version Aug 22 2008:15:12:49
# Run on Tue Sep 23 17:16:34 2008
# command line = "src/silistix_training_demo_master.csl -or -gv -bm -npv -dq "
# system: silistix_training_demo mode: default
#
# Initiator IDs
#   0 cpu_block/cpu/i_port
#   1 mpeg_block/mpeg/i_port
#   2 dma_block/dma/i_port
# Target IDs
#   0 host_interface/eth/t_port
#   1 dram_block/dram/t_port
#
@initiator cpu_i_port 0 AXI 32 32
@initiator mpeg_i_port 1 AXI 32 32
@initiator dma_i_port 2 AXI 32 32

@stimuli

#
# Setup Instrumentation
#
# dma_block/dma/i_port <= host_interface/eth/t_port
%2 check_read_bandwidth 0 104857600
# dma_block/dma/i_port => host_interface/eth/t_port
%2 check_write_bandwidth 0 104857600
# cpu_block/cpu/i_port <= dram_block/dram/t_port
%0 check_roundtrip_read_latency 1 140e-9
%0 check_read_bandwidth 1 209715200
# dma_block/dma/i_port <= dram_block/dram/t_port
%2 check_read_bandwidth 1 209715200
# cpu_block/cpu/i_port => dram_block/dram/t_port
%0 check_write_bandwidth 1 209715200
# mpeg_block/mpeg/i_port => dram_block/dram/t_port
%1 check_write_bandwidth 1 209715200
# dma_block/dma/i_port => dram_block/dram/t_port
%2 check_write_bandwidth 1 209715200
# mpeg_block/mpeg/i_port <= dram_block/dram/t_port
%1 check_read_bandwidth 1 838860800
#
# Initiator: cpu_block/cpu/i_port
#
%0 start_bandwidth_window
%0 repeat_block 39
  %0 read 1 INCR 8 0 0x100 32 0x8172de33 60e-9 0 0 # target: dram_block/dram/t_port
  %0 read 1 INCR 8 1 0x104 32 0xed860653 0e-9 0 0 # target: dram_block/dram/t_port
  %0 read 1 INCR 8 2 0x108 32 0xb3262fd1 0e-9 0 0 # target: dram_block/dram/t_port
  %0 read 1 INCR 8 3 0x10c 32 0x59b7f78f 0e-9 0 0 # target: dram_block/dram/t_port
  %0 read 1 INCR 8 4 0x110 32 0x26dc250a 0e-9 0 0 # target: dram_block/dram/t_port
  %0 read 1 INCR 8 5 0x114 32 0x98f2328a 0e-9 0 0 # target: dram_block/dram/t_port
  %0 read 1 INCR 8 6 0x118 32 0x4c5b962e 0e-9 0 0 # target: dram_block/dram/t_port
  %0 read 1 INCR 8 7 0x11c 32 0x33155db4 0e-9 0 0 # target: dram_block/dram/t_port
  %0 write 1 INCR 8 0 0x100 32 0x873ce775 0e-9 0 0 # target: dram_block/dram/t_port
  %0 write 1 INCR 8 1 0x104 32 0xc2edc876 0e-9 0 0 # target: dram_block/dram/t_port
  %0 write 1 INCR 8 2 0x108 32 0x13f7476d 0e-9 0 0 # target: dram_block/dram/t_port
  %0 write 1 INCR 8 3 0x10c 32 0xae3efcd4 0e-9 0 0 # target: dram_block/dram/t_port
  %0 write 1 INCR 8 4 0x110 32 0x1b21deb3 0e-9 0 0 # target: dram_block/dram/t_port
  %0 write 1 INCR 8 5 0x114 32 0x14103d60 0e-9 0 0 # target: dram_block/dram/t_port
  %0 write 1 INCR 8 6 0x118 32 0x6cd38e9f 0e-9 0 0 # target: dram_block/dram/t_port
  %0 write 1 INCR 8 7 0x11c 32 0xe9eb5470 60e-9 0 0 # target: dram_block/dram/t_port
```

```
  %0 wait_until 1221e-9
%0 end_block
#

#
# Report Results
#
%0 report_roundtrip_read_latency 1
%0 report_read_bandwidth 1
%0 report_write_bandwidth 1
%1 report_write_bandwidth 1
%1 report_read_bandwidth 1
%2 report_read_bandwidth 0
%2 report_write_bandwidth 0
%2 report_read_bandwidth 1
%2 report_write_bandwidth 1
```

Example Report File Output

Executing an NPV file creates a large report file. The end of the report file includes a Summary of results of the NPV testing. The details of the specific test patterns are listed in detail in the Trace Report.

Summary

Figure 13 shows an example summary section from an output file generated by the simulation script. The output shown here is formatted to fit the page and is slightly different than the actual format as it appears in the file.

The summary report lists the required and achieved latency or bandwidth values from every initiator to every connected target. Results are summarized separately for every initiator and then, for each initiator, further summarized by each connected target. For connections with specific requirements, the report file also shows the available slack in the specification.

- A **positive slack** indicates that the connection meets the requirement.
- A **negative slack** indicates that the connection misses the requirement. Further analysis and possible system modifications are required for any connection with negative slack.

Figure 13: **Example Report File Summary**

```
#================================================================================
#
#                                NPV Summary Report
#
# Results cpu_i_port
#  Connection to: dram_t_port:  fwd transactions= 600; rsp transactions=400
#  Read latency cpu_i_port => dram_t_port:
       (unconstrained, achieved min 145.00ns, max 190.00ns, average 149.00ns)
#  Write latency cpu_i_port => dram_t_port:
       (unconstrained, achieved min 87.00ns, max 112.00ns, average 90.00ns
#  Read bandwidth cpu_i_port <= dram_t_port:
       (required  10.0MBps, achieved   10.5MBps, slack 0.53471MBps)
#  Write bandwidth cpu_i_port => dram_t_port:
       (required  10.0MBps, achieved   8.9MBps, slack -1.13583MBps)
```

Trace Report

The trace report provides a detailed accounting of each action and transaction that occurs on the network.

Example Output

Figure 14 provides an example snippet from the trace report. All actions listed in chronological order, are time stamped, with all relevant transfer details (Transaction Type, address, data, size, ID), and where possible, referenced back to a line number in the NPV stimulus file.

Figure 14: **Example Trace Report Snippet**

```
                Initiator          Target
#   131.00  cpu_i_port wc---> dram_t_port (line=82  ):  awaddr=0x00000100
              awsize=0x2 awid=0 (initiator 0)
                         Transaction type
#   142.70   cpu_i_port --->d  dram_t_port            :  wdata=0x5e100ba2 wstrb=0b1111
              wlast=0 wid= 0 (initiator 0's wid=0)

#   146.25 mpeg_i_port  d---> dram_t_port (line=802 ):  wdata=0x16bbb6cf wid=0
              (initiator 1)

          . . .
  Time stamp
#  277.87   cpu_i_port <---wr dram_t_port            :  bresp=0 bid=0
```

Transaction Type

Each transaction in the NPV trace file includes a transaction code, shown in Table 3, indicating the type of transaction and the direction of data flow.

Table 3: **NVP Trace Transaction Type Codes**

Code	Description
Commands	
`rc--->`	Initiator sends a read command
`wc--->`	Initiator sends a write command
`--->rc`	Target accepts read command
`--->wc`	Target accepts write command
Write Data	
`d--->`	Initiator sends write data
`--->d`	Target receives write data
Responses	
`<---rr`	Target sends read response
`<---ww`	Target sends write response
`rr<---`	Initiator receives read response
`wr<---`	Initiator receives write response

Basics of Performance Closure

The NPV test environment helps achieve performance closure across all modes of operation within a system. The environment generates representative worst case traffic and ensures that the transactions meet the bandwidth and latency requirements of the system under actual operating conditions.

While SystemC modeling and other high-level mechanisms for modeling system level performance are useful, NPV stresses the performance corner cases by generating worst case traffic and proving that the requirements are met.

Three conditions determine whether an on-chip network meets the bandwidth and latency requirements specified by in a CSL source file.

1. Data Production,
2. Data Consumption, and
3. Data Transmission.

In other words, the NPV performance closure process validates that the system can produce, consume, and tramsit data at the required rates to guarantee full system operation?

Data Production

Data production occurs at both the initiator and target end of the network. Initiators on the Silistix network produce transactions that carry data during write operations or that issue commands and addresses to a target to which the target must respond. A target produces data during read operations or by responding to a request. In either case, the rate at which an initiator or target produces daa depends upon the initiator's operating frequency, and the data width of the transfer. For burst write operations, another aspect of data production is the number of beats (cycles of data width bits) necessary to transmit the entire burst.

Similarly, a target produces data during a read operation or during some response operations back to the initiator.

NPV uses the `read_response` and `write_response` statement declared in the CSL source file to determine how quickly an initiator or target can produce data once a transaction is received.

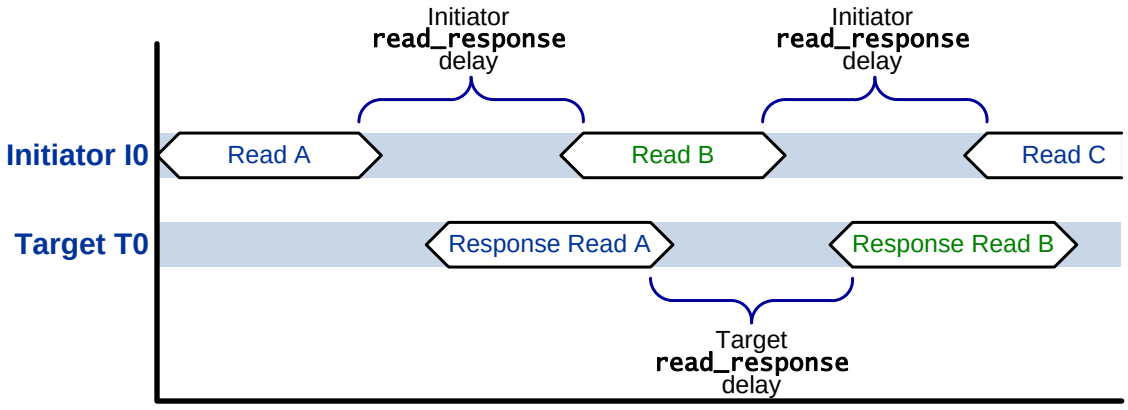
Data Consumption

Data consumption, like data production, happens at both the initiator and target end of the network. During a read operation, the initiator consumes the data produced by the target. During a write operation, the target consumes the data produced by the initiator. Similar to data production, data consumption is dictated by the initiator's or target's operating frequency, its data width, and burst length.

NPV uses the `read_response` and `write_response` CSL attributes to determine how quickly an initiator or target should respond after the data of a transaction is consumed. Figure 15 depicts

how the `read_response` time of both the initiator and target are reflected in how data is produced and consumed.

Figure 15: **Read and Write Response Delays**



When a burst transfer occurs, the response times are applied after the last beat of the burst.

Note that the `read_response` and `write_response` times of an initiator or target must be expressed as a multiple of the period (1/frequency) of the endpoint in nanoseconds.

Data Transmission

Data transmission is the most difficult portion of traffic generation as it relates directly to the amount of traffic being transmitted ***simultaneously*** throughout the network during a given mode of operation. CSL Compiler uses a static model for provisioning networks and to estimate the probability of network contention. The primary goal of the NPV validation process is to demonstrate that the network correctly handles the worst case contention while maintaining the specified bandwidth and latency requirements.

Contention occurs on the Silistix network under the following conditions

- A Silistix merge or switch network component is currently processing a transaction when another transaction arrives.
- More transactions arrive at an initiator or target than it has storage to accommodate, which causes “back pressure” in the system.

Complex SoC designs sometimes handle these contention issues using expensive quality of service schemes. By comparison, CHAINworks statically generates a properly-provisioned network that handles all combinations of traffic. The NPV process then proves that the resulting network meets or exceeds the bandwidth and latency requirements specified by the user. The end result is a fast, small network proven in advance to meet the specified system communication requirements.

CSL Compiler Traffic Generation

The CSL Compiler generates network traffic based on operating modes declared in the CSL file and the connections defined within each mode. Each `mode` statement represents a distinct, exclusive operating mode. This model further assumes that all transactions between initiators and targets happen exclusively within the mode; there is no interaction between initiators and targets operating in another mode. Therefore, CSL Compiler simply generates traffic data that meets the requirements specified for each initiator/target pair within a given operating mode.

A fundamental concept of NPV and the CSL Compiler static provisioning model is that each mode operates independently, for an unspecified period of time. Assume for example that a system has three exclusive operating modes: reset, low power mode, and high-speed data streaming. While initiators or targets operate in one particular mode, it is unlikely that they simultaneously operate in another mode. The duration of a mode for NPV validation has no relationship to the amount of time that the real application will operate in that mode.

CSL Compiler generates NPV stimulus for a mode using its own calculated time window. CSL Compiler generates sufficient network traffic to show that the CSL Compiler static provisioning and contention models meet the specified bandwidth and latency requirements described in the CSL source file.

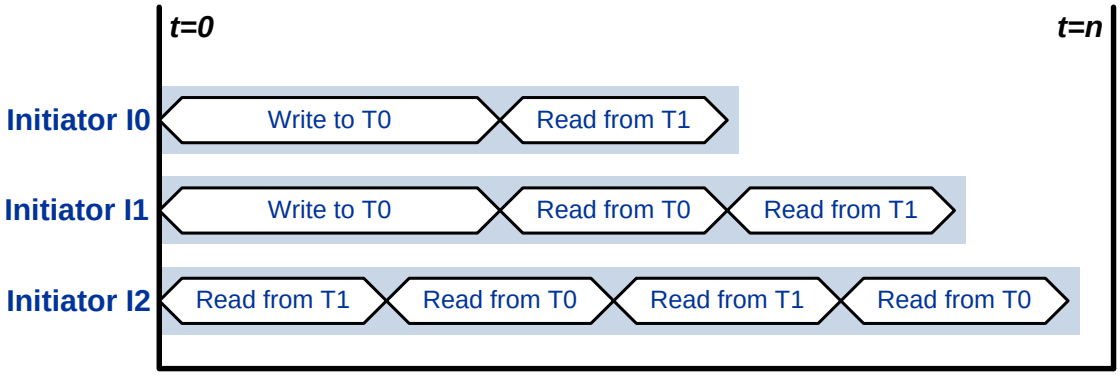
Structure of a Mode

Within NPV, a mode is represented by a single NPV traffic file with the extension `.npv`. All CSL connections that are not defined within a mode are validated with the `default.npv` file. Consequently, every CSL file results in at least one NPV traffic file. Any user specified modes result in a separated NPV traffic file with the mode's name and the `.npv` extension.

Conceptually, traffic patterns within a mode are separated into initiator blocks, as shown in Figure 12. Each initiator block describes a series of read and write transaction requests from the initiator to a target. Each transaction results in a burst of traffic. The length of the burst is specified in the CSL file.

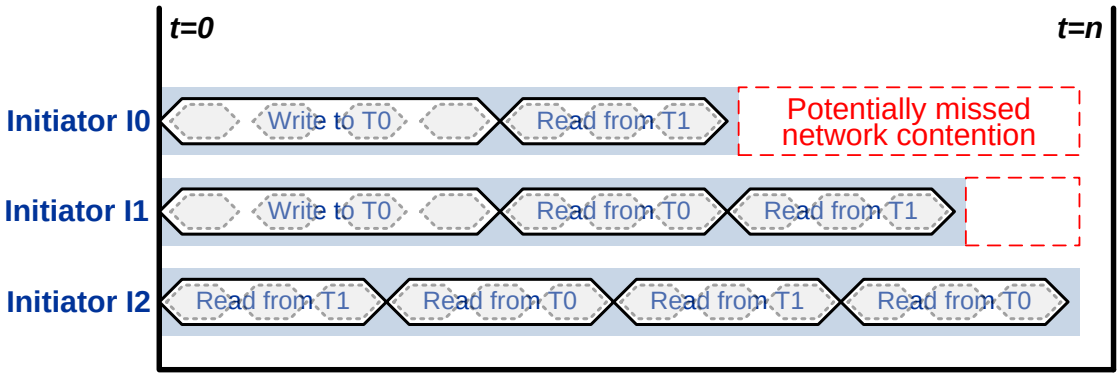
In Figure 16, initiators are named I0, I1, I2, etc., and targets are named T0, T1, T2, etc. Each of the horizontal rectangles represents an initiator block and the transactions that it generates during the mode. Both write and read transactions are interleaved proportionally based on their percentage of the total traffic they represent for the initiator with the mode.

Figure 16: **Three Initiator Blocks within a Mode**



In reality, however, each read and write transaction consists of one or more beats of data. A beat of data is defined by the data width of the initiator. Figure 17 extends Figure 16 to depict multiple beat bursts.

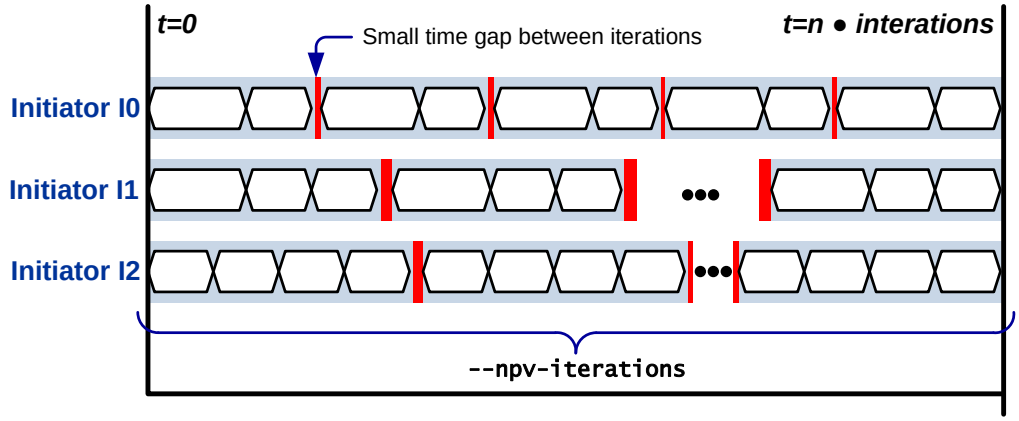
Figure 17: **Three Initiator Blocks with Multiple Beat Bursts within a Mode**



From Figure 17, observe that all three initiators I0, I1 and I2, have different execution times. If all three initiators started generating traffic simultaneously, then the simulation might miss potential network contention toward the end of the simulation. In this example, I0 finishes first. Consequently, I1 will not have to contend with I0 during roughly the last third of its transactions. Similarly, I2 will not have to compete with I0 for the last beat of its third transaction or any of its fourth transaction. I2 will also not have to compete with I1 during the last beat of its final transaction.

To remedy these potentially inaccurate simulations of worst-cast traffic, CSL Compiler repeats each initiator block a proportionate number of times so that all initiators complete their transactions at approximately the same time, as shown in Figure 18. This approach ensures that any potential network contention between initiators continues throughout the full simulation.

Figure 18: **Proportional Iterations to Sustain Contention**



Iteration Count

CSL Compiler generates just enough traffic to determine whether the design meets the specified CSL bandwidth and latency requirements, resulting in fast simulation times. By default, NPV generates ten iterations. However, it is possible that potential contention conditions within a mode might be missed due to small time gaps between iterations, as highlighted in Figure 18. To address this potential issue, specify a larger iteration count (> 10), indicating number of times to repeat the traffic pattern for an initiator. Set the iteration count using CSL Compiler's `--npv-iterations` command line option. As might be expected, a larger number of iterations increases simulation execution time. However, the larger number of iterations may potentially identify corner cases where a network should be provisioned differently.

Importance of CSL Accuracy

The fundamental goal of NPV is to generate worst case data traffic patterns that stress the network in order to guarantee that the design meets the specified bandwidth and latency requirements.

Modes

Virtually all SoCs have inherent modes of operation. For example, many SoCs have a reset mode, a low power mode, and a high performance mode. Some designs may have dozens of operating modes. Within a CSL file, use a separate `mode` statement to specify the bandwidth and latency requirements for each exclusive operating mode. CSL Compiler leverages this information to create a properly-provisioned network and to generate the NPV traffic files to test the network.

Failing to realistically describe modes and their communication requirements could result in a network that passes all tests but fails to perform satisfactorily with actual software running on the final silicon.

Connection Requirements

Set realistic latency and bandwidth requirements for the connections defined within a `mode` statement. Overly aggressive latency or bandwidth requirements may skew the network, resulting in increased latency and lower bandwidth elsewhere on the network, and likely making the resulting network far larger than necessary. CSL Compiler will attempt meet these requirements and to generate NPV traffic files to test them. However, network performance failures are likely to result and will need to be investigated and resolved.

Data Width

The data width of a CSL endpoint describes how many bits of data are transmitted in a single cycle at the frequency of the endpoint, also called a beat. The wider the data width, the more data bits are transmitted in a shorter period of time. However, the data widths of different endpoints need not be the same, although there is additional logic overhead when a wide endpoint communicates with a narrower endpoint. Arbitrarily widening a data width may not provide the expected results.

When CSL Compiler encounters mismatched data widths between an initiator and a target, it generates NPV traffic with the beat width equal to the smaller data width of the two endpoints.

Burst Length and Addressing

Burst length is a critical component of bandwidth; a longer burst has less overhead per transaction in terms of total bits transmitted and time is involved. While a longer burst length improves bandwidth, it also has a large impact on the storage necessary to hold them.

When CSL Compiler generates NPV traffic, it transmits random data value. The NPV test harness checks that these random values were correctly received by the target during a write operation or by the initiator during a read operation. Each beat of a burst uses an incremented address from the previous beat, cycling through and wrapping around within the address map locations of the target.

Outstanding Transactions

In CSL, the `outstanding` statement describes the maximum number of outstanding transactions that can be in flight over the network at any point in time. Presently, only AXI and OCP protocols support multiple outstanding. Increasing the number of outstanding transactions of an initiator can dramatically improve network bandwidth capabilities, but at the cost of a significant amount of storage within the network.

The CSL Compiler software uses the value of the `outstanding` statement to properly provision the network and to estimate bandwidth performance. The `outstanding` statement does not affect NPV traffic. Instead, the NPV test harness communicates with the adapter hardware that tracks the number of outstanding transactions that the adapter can support. The adapter signals the NPV test harness when it cannot yet accept another transaction.

Utilization Threshold

The CSL `utilization_threshold` statement provides a useful way to over provision a network for those cases when bandwidth requirements can only be roughly approximated or are subject to change. Setting the `utilization_threshold` to 80% provides an additional 20% overhead to handle any uncertainties.

Role of the First Placement Estimator (FPE)

The First Placement Estimator (FPE) tool uses pre-placement information to provide NPV with more realistic wire delays. Wire delays have a significant impact on the actual network latency and bandwidth. Running NPV validation on a network ***after*** performing FPE is an important step to guaranteed performance closure.

Silistix Glossary

Term	Description
FPE	First Placement Estimator
NoC	Network on Chip
NPV	NoC Performance Validation
RTL	Register Transfer Level
SoC	System on Chip
Tcl	Tool Command Language

Revision History

Revision	Date	Description/Revisions
1.0.2	20-JAN-2009	Updated CHAINworks version to 2.2.
1.0.1	12-DEC-2008	Minor corrections and updates.
1.0	31-OCT-2008	Initial release.

Feedback

Feedback on this Silistix document and all Silistix products is highly encouraged. If you have a comment, correction, or suggestion to improve this document, please send us an E-mail. Please include complete details including page numbers, section titles, or figure or table numbers where appropriate. Thank you in advance for helping us to improve our products and services.

feedback@silistix.com