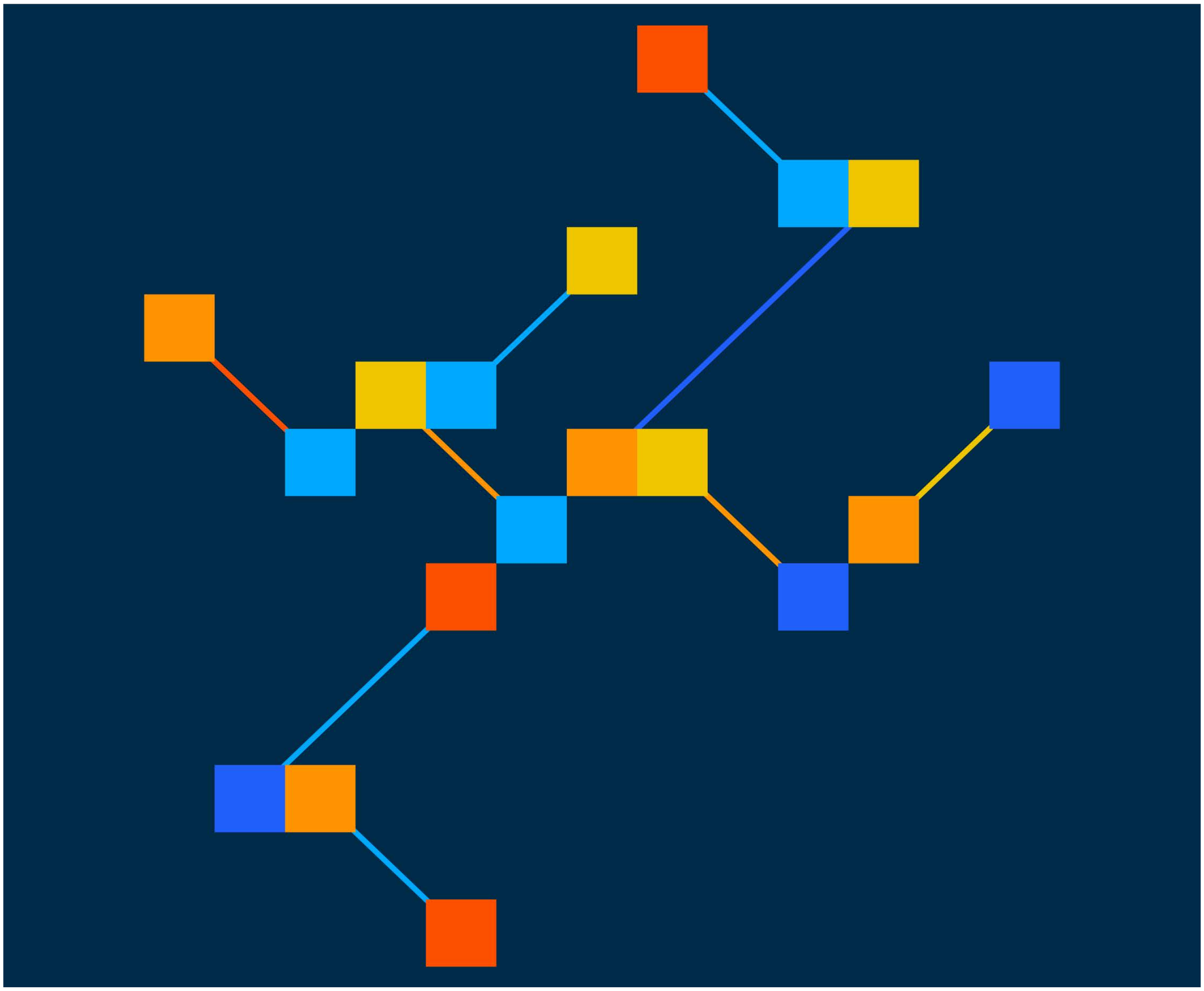
ARTERIS IP

System Integration Guide
Ncore 3.6



arteris.com

Doc Name	Revision	Date	Description
100012_IG_3.6.4	1.0_A.1	Sep 2024	Release Update
100012_IG_3.6.3	1.0_A.1	Jun 2024	Release Update
100012_IG_3.6.2	1.0_A.1	Mar 2024	Release Update
100012_IG_3.6.1	1.0_A.1	Jan 2023	Release Update
100012_IG_3.6	1.0_A.1	Nov 2023	3.6 Initial Release

Contents

1 Introduction

Ncore 3 is the third version of the Arteris IP configurable cache coherent interconnect product that enables the integration of heterogeneous coherent agents and non-coherent agents into complex SoCs. Ncore 3 supports the AMBA® CHI, ACE, and AXI protocols. Ncore also allows cache coherent agents using these different protocols to be assembled together and kept coherent with each other.

The modular and distributed architecture of Ncore 3 results in higher performance, lower power consumption, and a smaller die area.

This guide describes the necessary information and considerations when integrating an Ncore 3 interconnect into a host system using a standard ASIC design flow. This includes the RTL level with verification, timing, logic synthesis, and SRAM integration; use of resiliency optional features; use of IP-XACT for documentation; and automatic assembly with other IP.

Note: The software integration considerations, including use of registers for configuration, are covered in the Ncore 3 Reference Manual.

2

Interface Support

Ncore, an interconnect IP, interfaces to other IP logic and customer design logic.

2.1 APB4 Interface Support

Ncore supports APB4 protocol and the list of signals with their associated functionality is reported in the table below.

TABLE 1. **APB4 support in Ncore**

APB4 Fields	Default	Comment
PCLK	N/A	Functionality fully supported
PRESETn	1'b1	Functionality fully supported
PADDR	N/A	Functionality fully supported
PPROT[2:0]	3'b000 Bits [0] & [2] are a don't care for Ncore3.6	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access. [0] : 0 = Normal, 1 = Privileged [1] : 0 = Secure, 1 = Non-Secure [2] : 0 = Data Access, 1 = Instruction Access
PSELx	1'b0	Functionality fully supported
PENABLE	1'b0	Functionality fully supported
PWRITE	1'b0	Functionality fully supported
PWDATA	N/A	Functionality fully supported
PSTRB[n]	Assert all HIGH for write transaction. Assert all LOW for read transaction. Sparse data transfers are not allowed on Ncore3.6.	Write strobes. This signal indicates which byte lanes to update during a write transfer. There is one write strobe for each eight bits of the write data bus. Therefore, PSTRB[n] corresponds to PWDATA[(8n + 7):(8n)]. Write strobes must not be active during a read transfer.
PREADY	1'b0	Functionality fully supported
PRDATA	N/A	Functionality fully supported
PSLVERR	1'b0	Functionality fully supported

2.2 Interface Timing

From a timing perspective, these interfaces are either synchronously timed, asynchronously timed, or both.

Synchronous interfaces are the mainstream means to connect to Ncore. For details on integrating IP and custom logic to Ncore via synchronous interfaces, refer to “Integrating Synchronous Interfaces”.

Ncore's asynchronously timed interfaces and signals are used for interrupt and power control functions. For details in integrating these interfaces into a host design, refer to “Integrating Asynchronous Interfaces”.

2.3 Integrating Synchronous Interfaces

Each synchronous interface has a reference clock, a set of signals that are driven into Ncore, and a set of signals that are driven from Ncore.

The reference clock name for a given set of signals is in the IPXACT file, generated as part of the RTL export, in the Ncore project.

Ncore does not require the reference clock to have a specific duty cycle as it is designed using positive edge logic. The duty cycle is limited by the implementation technology.

Signals inside Ncore should be valid for at least 20% of the clock period. That means that the signals driven from Ncore will be valid for a maximum of 80% of the reference clock cycle period (this includes the clock uncertainty). Signals that are driven to Ncore will be valid for a maximum of 80% of the reference clock cycle period (this includes the clock uncertainty).

2.4 Integrating Asynchronous Interfaces

All asynchronous signals that are driven into a Ncore interconnect will be internally synchronized to the sampling clock.

All asynchronous signals that are driven from a Ncore interconnect must be synchronized, by the user, to the relevant sampling clock(s).

3

Select Maestro Flow Considerations

The following Maestro flow and configuration items should be reviewed and considered for applicability.

3.1 Embedded Memories

Ncore enables the user to, optionally, implement specific data structures as embedded memories.

For example, the user could specify tag-based snoop filter data structures to be implemented as compiled SRAMs, while other data structures, for example staging data buffers, are implemented as cell-based flops.

3.1.1 Memory Wrapper

Ncore creates a "wrapper" for each logical data structure, which is an empty RTL module with control, address, and data signals required to implement a functional interface that is driven from Ncore to the embedded memory wrapper. The user is responsible for implementing the wrapper module by instantiating one or more compiled memory arrays and specifying any other external logic required to fully implement the memory side of the Ncore control interface. The user can add or remove signals required for BIST/DFT or other functionalities. These signals are also driven from Ncore to the embedded memory wrapper.

The user-supplied functionality for the wrapper module **must** fully implement the specification that is generated in the embedded memories section of the reports summary. This logical embedded memory specification describes control sequencing, embedded memory depth requirements, and embedded memory width requirements.

Ncore creates both single and double ported memory wrappers. The list of signals for a single ported memory wrapper is as follows:

D:	Write Data
ADR:	Address
ME:	Memory Enable. Asserted high for each read or write
WE:	Write Enable. Asserted high for writes
WEM:	Write Enable Mask. WEM is as wide as the data and has negative polarity. When it is 0, then the associated bit should be written.
Q:	Read Data

The timing of the embedded memories is shown in Figure 1, which shows read timing and Figure 2, which shows write timing.

Figure 1. Read Timing

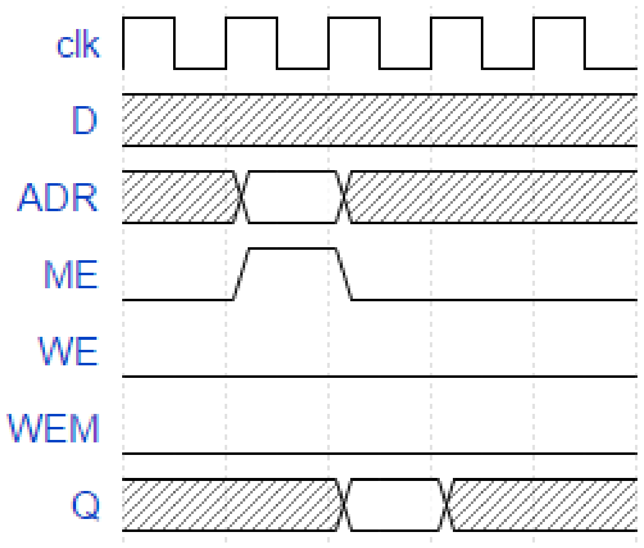
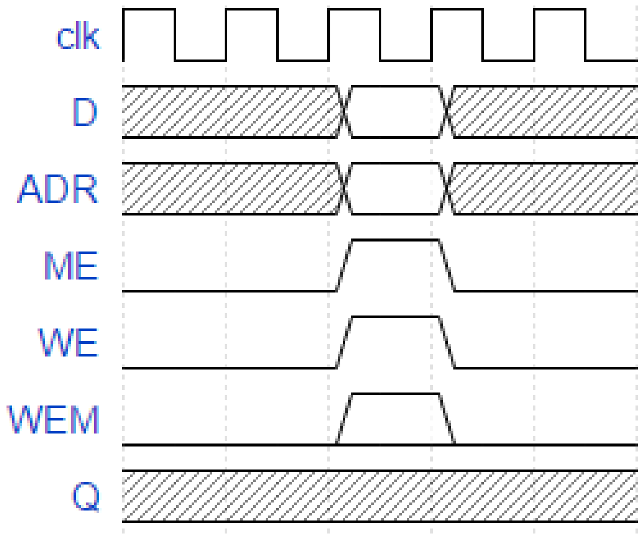


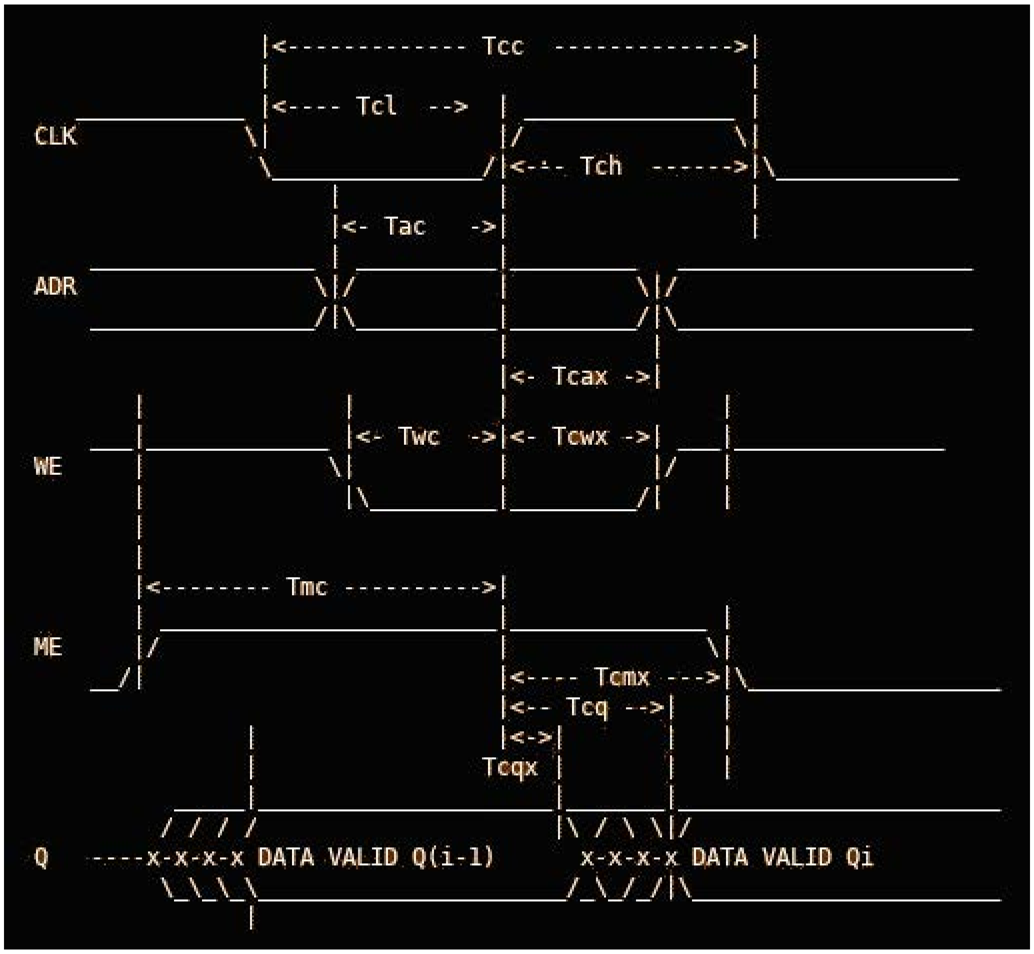
Figure 2. Write Timing



3.1.2 DCE Memory Read Operation Assumption

Ncore Distributed Coherence Engine embeds memories for the directory structures to track the location of data. The DCE unit has an implicit assumption, for pipeline stall/replay logic, that the Memory Output holds after a read operation completes that is aligned to the memory behavior as shown in the following figure.

Figure 3. Memory Read Cycle Timing Waveforms

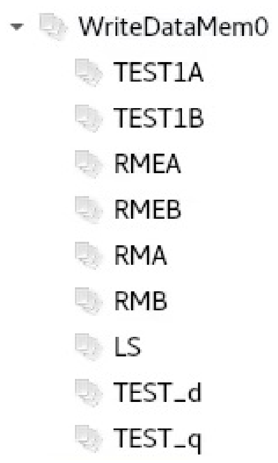


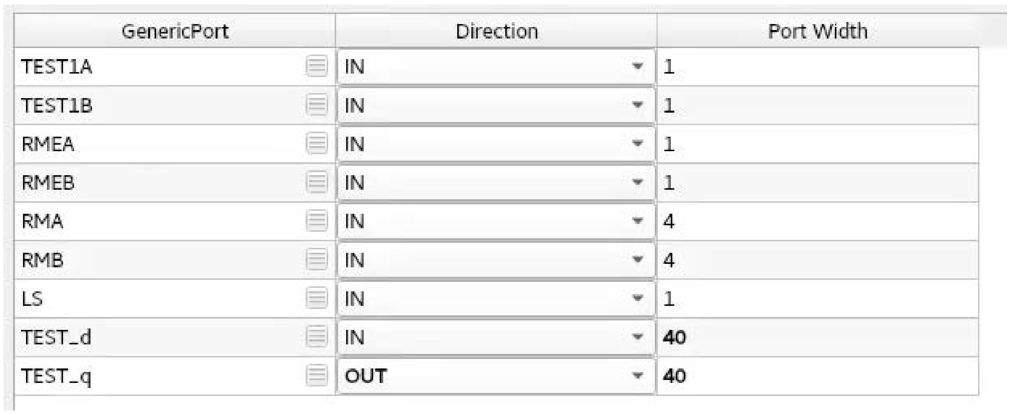
3.2 SRAM Test Signals

Maestro provides the ability to define and create test signals for memories of type SRAM. They are created during the Topology Structure phase of Maestro. A test signal is made up of three parameters that must be provided during creation:

- Name
- Direction
- Bit width

Example





GenericPort	Direction	Port Width
TEST1A	IN	1
TEST1B	IN	1
RMEA	IN	1
RMEB	IN	1
RMA	IN	4
RMB	IN	4
LS	IN	1
TEST_d	IN	40
TEST_q	OUT	40

- When RTL is generated, the test signals will be created at the top level of the design and will be routed down the hierarchy to the associated memory wrapper. This provides a method to access the SRAM signals outside the IP, where they can be used for MBIST.
- All test signals are grouped into a single bus by direction. The bus (in, out) is then routed through the hierarchy into each memory wrapper. Inside the memory wrapper, the individual signals are re-created from the bits of the bus allowing for the connection of the test signals to the individual memory slices as needed.

```
module dmi_wrdatamem_em_mem_external_a (
    input clk,
    input cg_test_en,
    input [372:0] int_data_in,
    output [372:0] int_data_out,
    input [2:0] int_address,
    input int_write_en,
    input int_chip_en,
    input [52:0] in,
    output [39:0] out);

    assign { TEST1A, TEST1B, RMEA, RMEB, RMA, RMB, LS, TEST_d} = in;
    assign  out = { TEST_q};

    dmi_wrdatamem_external_mem_a external_mem_inst (
    .CLK (dmi_wrdatamem__CLK),
    .D (dmi_wrdatamem__D),
    .ADR (dmi_wrdatamem__ADR),
    .WE (dmi_wrdatamem__WE),
    .ME (dmi_wrdatamem__ME),
    .Q (dmi_wrdatamem__Q),
    . TEST1A ( TEST1A),
    . TEST1B ( TEST1B),
    . RMEA ( RMEA),
    . RMEB ( RMEB),
    . RMA ( RMA),
    . RMB ( RMB),
    . LS ( LS),
    . TEST_d ( TEST_d),
    . TEST_q ( TEST_q));
module dmi_wrdatamem_external_mem_a
# (
   parameter DATA_WIDTH = 373
  ,parameter DATA_DEPTH = 8
  ,parameter ADDR_WIDTH = 3
) (
    input CLK,
    input [DATA_WIDTH-1:0] D,
    input [ADDR_WIDTH-1:0] ADR,
    input WE,
    input ME,
    output reg [DATA_WIDTH-1:0] Q,
    input  TEST1A,
    input  TEST1B,
    input  RMEA,
    input  RMEB,
    input [3:0] RMA,
    input [3:0] RMB,
    input  LS,
    input [39:0] TEST_d,
    output[39:0] TEST_q);
`ifndef ARTERIS_BACKEND
          assign TEST_q = 'h0;
    reg [DATA_WIDTH-1:0] Q_int;
```

```
        reg [DATA_WIDTH-1:0] mem_core [DATA_DEPTH-1:0];
        always @(posedge CLK) begin
            if (ME & ~WE) begin
                Q_int <= mem_core[ADR];
            end
        end
        // mem_core flops
        always @(posedge CLK) begin
            if (ME & WE) begin
                mem_core[ADR] <= D;
            end
        end
        // mem output
        always @(*) begin
            Q = Q_int;
        end
    ////////////////////////////////////////////////////////////////////////////////
    `endif
    endmodule
```

3.3 Replication

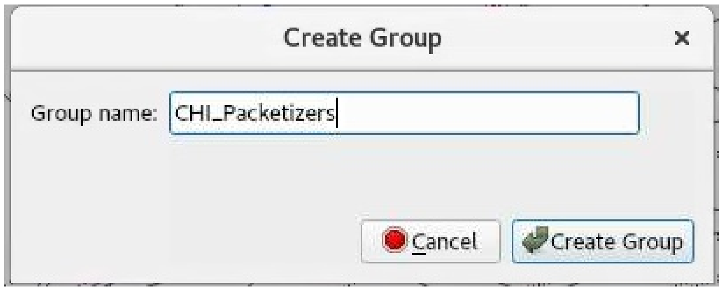
Ncore units that have identical parameters will have an identical Verilog definition. To reduce design time, the Ncore unit can be physically implemented once and then saved as a physical block. This physical block can then be replicated in the design multiple times for reuse.

Every Ncore unit has a routing table which it uses to determine where it needs to send messages. These routing tables are unique for every unit, even if those units have identical parameters. The routing tables are stored in the packetizers associated with each unit.

To replicate identical Ncore units, you need to separate the Ncore unit base logic from its packetizers and depacketizers. In the GUI, Groups is used to accomplish this task. The grouping is done during the Topology Structure phase, prior to mapping.

3.3.1 Replication Steps of Ncore Units

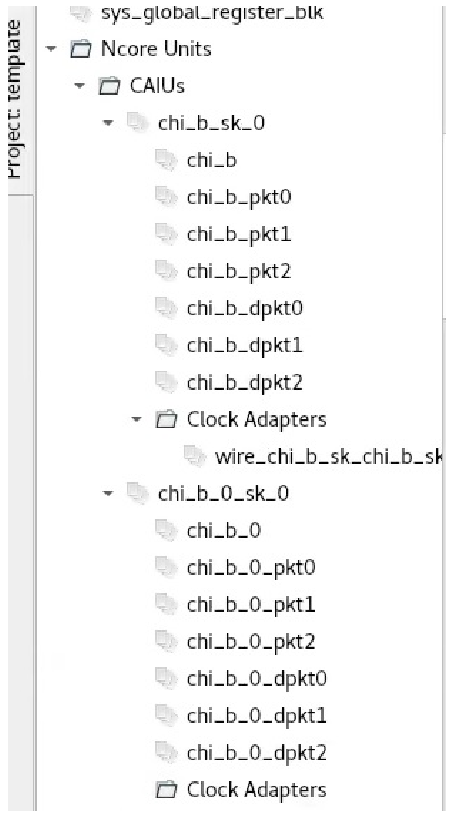
1. Start in the Groups pane, right-click, and create a new group. The group name will be the module name of the newly formed Verilog.



Create Group

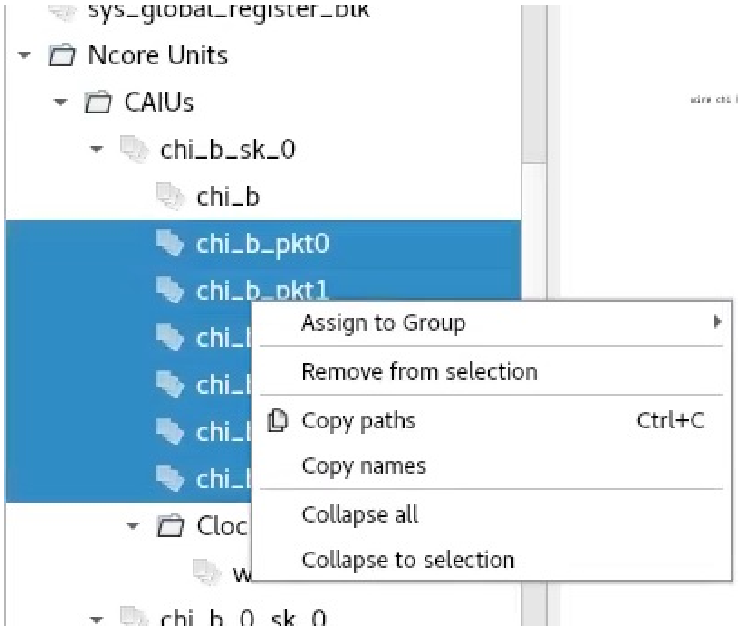
Group name: CHI_Packetizers

2. In the Project Tree, display the Ncore Units. Here, under CAIUs, you can see `chi-b_sk_0`. This is an NIU, a container for the CHI unit and its packetizer/ depacketizer, as seen in the following.

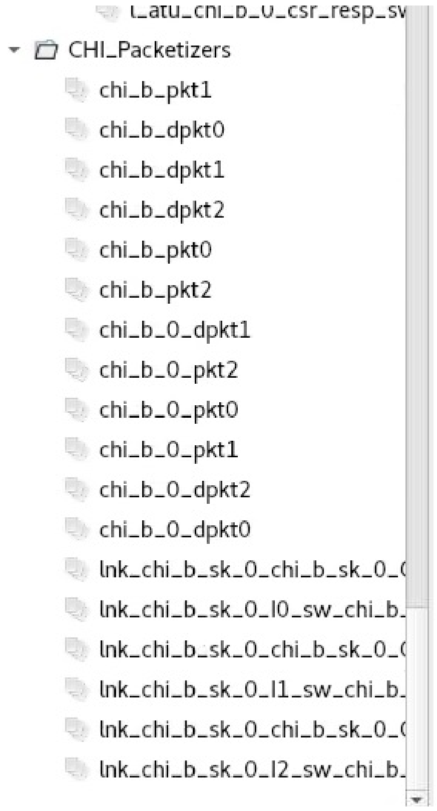


```
sys_global_register_blk
Ncore Units
  CAIUs
    chi_b_sk_0
      chi_b
      chi_b_pkt0
      chi_b_pkt1
      chi_b_pkt2
      chi_b_dpkt0
      chi_b_dpkt1
      chi_b_dpkt2
      Clock Adapters
        wire_chi_b_sk_chi_b_sk
    chi_b_0_sk_0
      chi_b_0
      chi_b_0_pkt0
      chi_b_0_pkt1
      chi_b_0_pkt2
      chi_b_0_dpkt0
      chi_b_0_dpkt1
      chi_b_0_dpkt2
      Clock Adapters
```

3. Move the packetizers/depacketizers into the newly created group. This can be done several ways, but it is easy to multi-select the objects, right-click, and choose “Assign to Group”.



4. These are now moved in the Groups pane to the newly created group CHI_Packetizers..


t_atu_chi_b_0_csr_resp_sv
CHI_Packetizers
chi_b_pkt1
chi_b_dpkt0
chi_b_dpkt1
chi_b_dpkt2
chi_b_pkt0
chi_b_pkt2
chi_b_0_dpkt1
chi_b_0_pkt2
chi_b_0_pkt0
chi_b_0_pkt1
chi_b_0_dpkt2
chi_b_0_dpkt0
lnk_chi_b_sk_0_chi_b_sk_0_
lnk_chi_b_sk_0_l0_sw_chi_b_
lnk_chi_b_sk_0_chi_b_sk_0_
lnk_chi_b_sk_0_l1_sw_chi_b_
lnk_chi_b_sk_0_chi_b_sk_0_
lnk_chi_b_sk_0_l2_sw_chi_b_

5. Complete the flow as normal, and upon collateral generation, the two CHI units will share the same RTL and can be implemented as a replicated block.

3.4 RTL Prefixing

When multiple Ncore designs exist in the same SoC it is possible that there can be RTL module collisions (i.e., multiple modules named the same thing) at the SoC level. To avoid this problem, the ncores should be RTL prefixed to ensure each ncore is uniquely named.

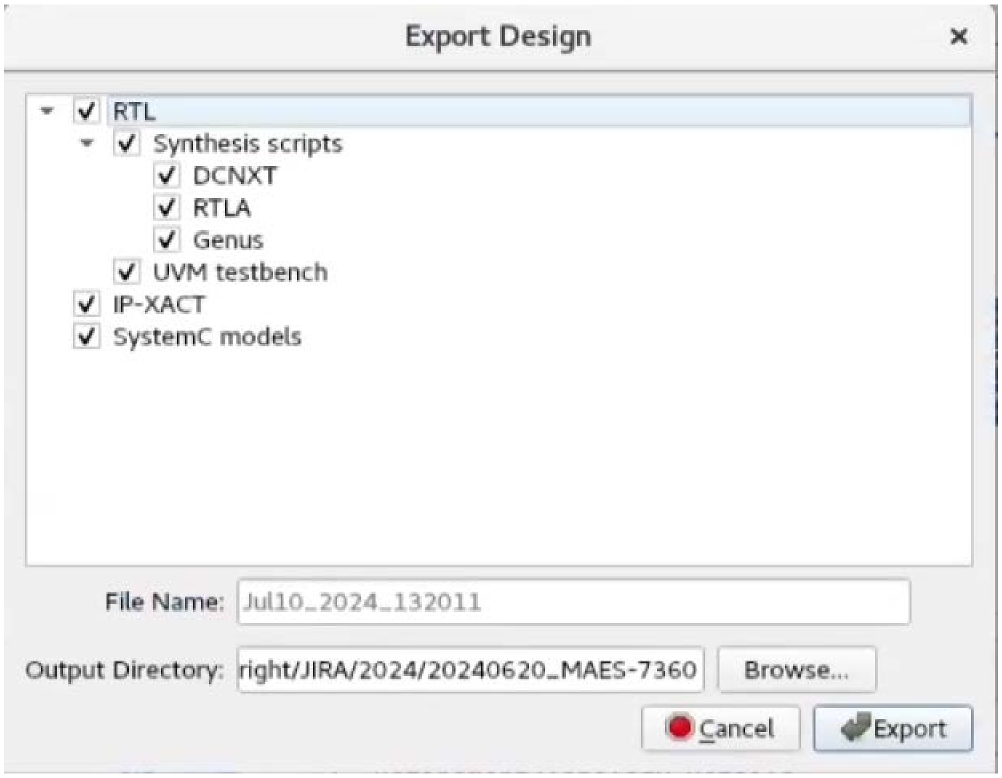
The RTL prefix feature is available through the parameter `useRtlPrefix` found in the solution (e.g., `project/default_chip/default_system/default_subsystem/default_solution`).

4 Generating Maestro Outputs

The output tarball (e.g., `output.tgz`) from Maestro can be generated via the GUI (see Figure 4 on page 20) or the TCL command line (i.e., `gen_collateral`). When the tarball is extracted a directory of Maestro outputs is presented as shown.

```
 output
├── doc
│   ├── csv
│   └── pdf
├── IPXACT
├── models
│   ├── standalone
│   │   └── TLM
│   └── synopsysPA
│       ├── TLM
│       └── verilated
├── rtl
│   ├── design
│   ├── models
│   │   ├── cells
│   │   └── memories
│   ├── placeholders
│   └── reports
├── setup
├── simulation
├── synthesis
│   ├── DCNXT
│   ├── Genus
│   ├── RTLA
│   └── upf
└── tb
    └── xsim
    └── vcs
```

Figure 4. Design Export



Export Design

- [✓] RTL
  - [✓] Synthesis scripts
    - [✓] DCNXT
    - [✓] RTLA
    - [✓] Genus
  - [✓] UVM testbench
- [✓] IP-XACT
- [✓] SystemC models

File Name: Jul10_2024_132011

Output Directory: right/JIRA/2024/20240620_MAES-7360

4.1 The directory output/doc and Documentation

The directory `output/doc/pdf` contains PDF documents covering CSR registers descriptions and embedded memories. Based on the NoC configuration the directory may also contain Functional Safety Controller fault bit mapping and other documents. The directory `output/doc/csv` contains CSR description per unit but in CSV format (e.g., machine readable).

4.2 The directory output/IPXACT and IP-XACT

The IP-XACT project information is created in the directory `output/IPXACT`. The files `<project_name>_2009.xml` and `<project_name>_2014.xml` are populated with several sections containing information about the Ncore3 IP, and are emitted based on the configuration.

4.2.1 Bus Interfaces

This section contains a list of all Ncore sockets exposed outside the top level of the formats:

```
<ipxact:busInterfaces>[<ipxact:busInterface>]#
<spirit:busInterfaces>[<spirit:busInterface>]#
```

This section contains the various sockets. Each socket `<ipxact:busInterface>` or `<spirit:busInterface>` describes the mapping of physical pins to their corresponding logical roles.

4.2.2 Memory Maps

A memory map is defined for each CSR slave interface of a component. The Memory Maps are of the following formats:

```
<ipxact:memoryMaps>[<ipxact:memoryMap>]#
<spirit:memoryMaps>[<spirit:memoryMap>]#
```

4.3 The directory output/models

The directory `output/models` contains content which is part of the verification platform. See “RTL Simulation Flows” on page 23 for more details.

4.4 The directory output/rtl and the Design Verilog

The directory `output/rtl` and the sub-directories contain the Verilog RTL modules which make up the design. RTL manifest files “`<>.flist`” also exist in the `output/rtl/design` sub-tree. The `top.flist` exists as the manifest for the complete design. Unit level manifest files (e.g., “`<unit>.flist`”) also exist for exploration and debug purposes.

```
output
├── rtl
│    ├── design
│    ├── models
│    │    ├── cells
│    │    └── memories
│    ├── placeholders
             │    └── reports
```

The directory `output/rtl/design` contains Verilog RTL which should be considered golden and consumed as is without modification. Prior to release 3.7 there is one caveat with respect to `cg_prim.v` that is described below.

The description of the design can contain relatively fine grain primitives for which precise mapping from RTL to technology specific gate(s) is desired. An example of this is cg_prim.v which can be instantiated in the design as an architectural clock-gater. EDA tools may map to a discrete combinational gate and a latch (undesired) or to a monolithic ICG cells (desired). The directory `output/rtl/models/cells` contains those Verilog RTL files for which a precise mapping may be desired. The exact collection of files varies based on design configuration and release.

The Verilog RTL description is suitable for verification, but the module should likely get mapped to technology specific cell(s) prior to executing an implementation flow.

The directory `output/rtl/models/memories` contains those Verilog RTL files that represent functional memories. The exact collection of files varies based on design configuration and release. The Verilog RTL description is suitable for verification but needs to get mapped to technology specific SRAM(s) prior to executing an implementation flow. One functional memory (e.g., 512x16) may be implemented via several smaller pieces (e.g., 4 256x8 SRAM, read mux, etc). See “Select Maestro Flow Considerations” on page 9for more details

The directory `output/rtl/placeholders` contains RTL files that can be modified by the end-user. Typically, modifications are for purposes of native interface protection.

The directory `output/reports` is currently empty.

4.5 The directory output/setup

The directory `output/setup` is currently empty.

4.6 The directory output/simulation

The directory `output/simulation` is currently empty.

4.7 The directory output/synthesis

The directory `output/synthesis` contains some example EDA synthesis solutions. See “Implementation Flows” on page 29 for more details.

4.8 The directory output/tb

The directory `output/tb` contains content which is part of the verification platform. See “RTL Simulation Flows” on page 23 for more details.

5

RTL Simulation Flows

With every RTL export, an example testbench is generated for simulation in either vcs or xsim. Cadence xsim support in v3.6.4 is preliminary. This section assumes you have your vendor licenses for simulation and VIP set up properly and that you can successfully run a non-Ncore simulation. Below is a description of what gets exported.

5.1 Example Testbench Files

The example testbench files are generated into the `<output_dir>/tb/<SIM_TOOL_dir>` directory (`<SIM_TOOL_dir>` can either be xsim or vcs), which contains the following directories and files:

`<output_dir>/tb/<SIM_TOOL_dir>`

`common/`	Directory that contains common files between simulators. (This will be moved up a level to tb/ in a later release)
`env/`	Environments in the testbench.
`seq/`	Base sequences.
`tests/`	Example tests.
`Utils/`	Utility support (e.g., assertions).
`vseq/`	Virtual sequences.
`Makefile`	A makefile that supports compiling the testbench environment, and running the tests, either independently, or all at once.
`*.sv, *.svh, *.svi`	Verilog and SystemVerilog implementation of various components of the example testbench, as well as VIP defines.

Note that the structure and contents of the `tb/` directory tree may change in future releases.

5.2 Example Testbench Set-up

Synopsys (vcs)

To run the example testbench:

1. Set `$VCS_HOME` to point to the VCS installation (vcs will be in the `bin/` directory of this path).
2. Set `$DESIGNWARE_HOME` to point to the Synopsys `VIP` directory.
3. Set `$SNPS_AMBA_VIP` to point to the Synopsys `CHI/ACE VIP` project directory (can be generated in local workspace using the command given by make if `$SNPS_AMBA_VIP` is not set).
4. Set `$PROJ HOME` to the absolute path of Maestro output (e.g., `/path/to/exe/output`).
5. Set `$VERDI_HOME` to point to the Synopsys Verdi installation.

Cadence (xsim)

To run the example testbench using Cadence VIP (assuming a 64-bit installation on x86 linux, xcelium version 24.03-s001, and VIP version 11.30.096):

1. Set `$VCS_HOME` to point to the VCS installation (will use Synopsys `$VCS_HOME/bin/ralgen` to generate `ncore_system_register_map`. Contact your AE if you do not have access to Synopsys tools. This will be fixed in a future release.).
2. Set `$PROJ_HOME` to the absolute path of maestro output (e.g., `/path/to/exe/output`).
3. Set `$CDS_INST_DIR` to point to the Cadence xcelium install (`xrun` should be under `bin` in this path.
4. Set `$CDN_VIP_ROOT` to point to the Cadence VIP installation.
5. Set `$CDN_VIP_LIB_PATH` to point to the Cadence VIP library installation.

5.3 Using the Example Testbench

The example testbench environment includes a set of tests to test the coherent subsystem. The environment is based on the UVM environment and leverages the commercial VIPs. Each one of the interfaces' name is based on the name assigned to the structure in the Transport Definition perspective. The example testbench is constructed using a standard makefile. The make targets are described as follows:

`make build`	Builds and compiles the testbench for the coherent subsystem portion of the design.
`make <test_name>`	Runs a single test for the coherent subsystem portion of the design.
`make clean`	Removes the results of previous calls of make.
`make all`	Builds the testbench and runs `ncore_sys_test`, `ncore_ral_reset_value_test`, and `ncore_connectivity_test`.

Run `make` from `<output_dir>/tb/<SIM_TOOL_dir>`.

The build outputs are in the `<output_dir>/tb/<SIM_TOOL_dir>/build` directory.

Test outputs are in the `<output_dir>/tb/<SIM_TOOL_dir>/run/<test_name>` directory.

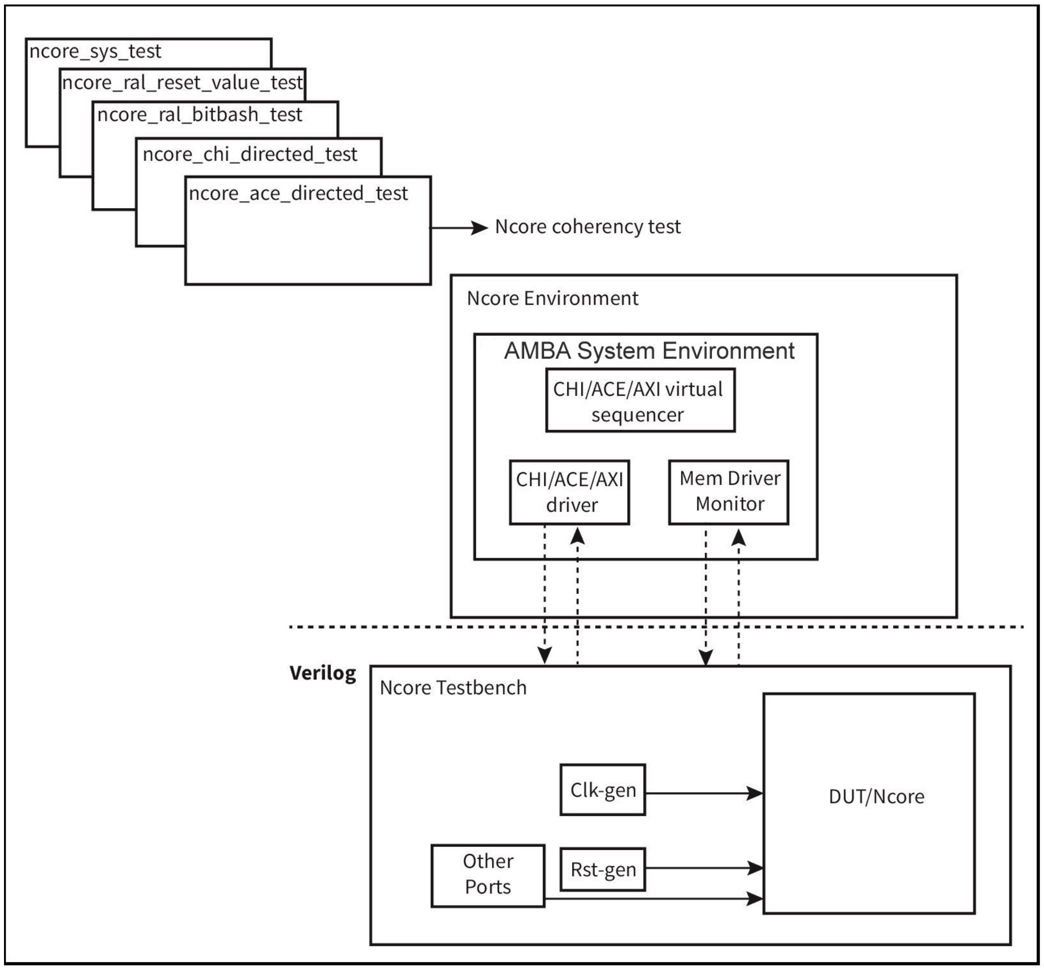
5.4 Model Structure

The UVM testbench for coherency consists of two parts: a Verilog part and a SystemVerilog/UVM part.

In the Verilog part, there is a top level testbench which instantiates the top level Ncore module, which is the DUT. On the master side, the testbench has interfaces which connect Verification IP (VIP) to all the agent CAIUs as well as to the NCAIUs. On the slave side, it connects VIP slave memory models. It also connects VIP slave memory models to any non-coherent interconnect target interface that receives traffic from the DII.

In the SystemVerilog with UVM part, the user will generate traffic using commercial VIPs.

Figure 5. Ncore Testbench Block Diagram



5.5 Individual Test Descriptions

Example testbench include the following tests:

`ncore_sys_test`	This test runs a boot-up sequence that configures registers required to initialize the DUT.
`ncore_ral_reset_value_test`	This test checks all registers to make sure that the value read is the expected reset value as described in the RAL class of the register.
`ncore_ral_bitbash_test`	This test evaluates each bit of every Ncore register to ensure that the resulting value matches the mirrored value as described in the RAL class of the register.
`ncore_chi_directed_test`	This test issues CHI traffic through VIP CHI BFM. Users can send different types of coherent transactions in svt_chi_transaction and are allowed to modify start address, address offset, address increment, master ID, cache value, and sequence length.
`ncore_ace_directed_test`	This test issues AXI/ACE traffic through VIP ACE BFM. Users can send different types of coherent transactions in svt_axi_transaction and are allowed to modify start address, address offset, address increment, master ID, cache value, and sequence length.
`ncore_connectivity_test`	This test issues 1 transaction from all initiators to all memory regions in a sequence order. The main goal is to check that all initiators communicate with all slaves.
`ncore_bandwidth_test`	The goal of this test is to measure bandwidth and latency for every master. The tests issues a 1K coherent and non-coherent read/writer transactions from each master sequentially.
`ncore_bandwidth_test_multi`	The goal of this test is to measure bandwidth and latency for every master. The tests issues a 1K coherent and non-coherent read/writer transactions from each master in parallel.
`ncore_snoop_test`	This test runs snoop transactions between different masters.
`ncore_cache_access_test`	This test initiates WriteUnique to slave address allocating bit high then ReadOnce to the same address to access SMC and proxy cache.
`ncore_fsc_ralgen_err_intr_test`	This test checks for software triggered error interrupt. It inserts error logging alias registers to check if any correct interrupt register value gets logged and if interrupt pin is toggled.

5.6 Test Output

In the `<output_dir>/tb/<SIM_TOOL_dir>/run/<test_name>` folder, each test output directory contains a `vcs.log/xrun.log` for Synopsys/Cadence and `test.fsdb/waves.shm` for Synopsys/Cadence that includes waveforms of all signals in the simulation.

6 Implementation Flows

With each RTL export, Ncore can generate example implementation scripts. These scripts span multiple EDA vendors and tools and currently include Synopsys Design Compiler NXT, Synopsys RTL Architect, and Cadence Genus. These scripts are primarily intended to demonstrate Arteris outputs (e.g., RTL, SDC) into the various EDA solutions. The solutions are simple flat block level synthesis solutions. These scripts are technology independent and can be used for purposes of early RTL to gates mapping for coarse area and timing estimates.

The end-user is ultimately responsible for a production quality implementation solution. This would include but is not limited to topics such as DFT and BIST insertion, design partitioning, floor planning, PNR, etc.

6.1 Global Synthesis Options

This section describes the global synthesis options used in all flows:

`rtlWrapperDir`	RTL directory
`maxTransition`	Maximum transition
`outputLoad`	Output load
`clockUncertainty`	Clock uncertainty

- Clock Uncertainty: Specifies the default clock uncertainty in the synthesis scripts.
- RTL Wrapper directory: Specifies the location of memory wrappers.
- Max Transition: Default transition delay on functional input ports.
- Output Load: Default capacitive load on functional output ports.

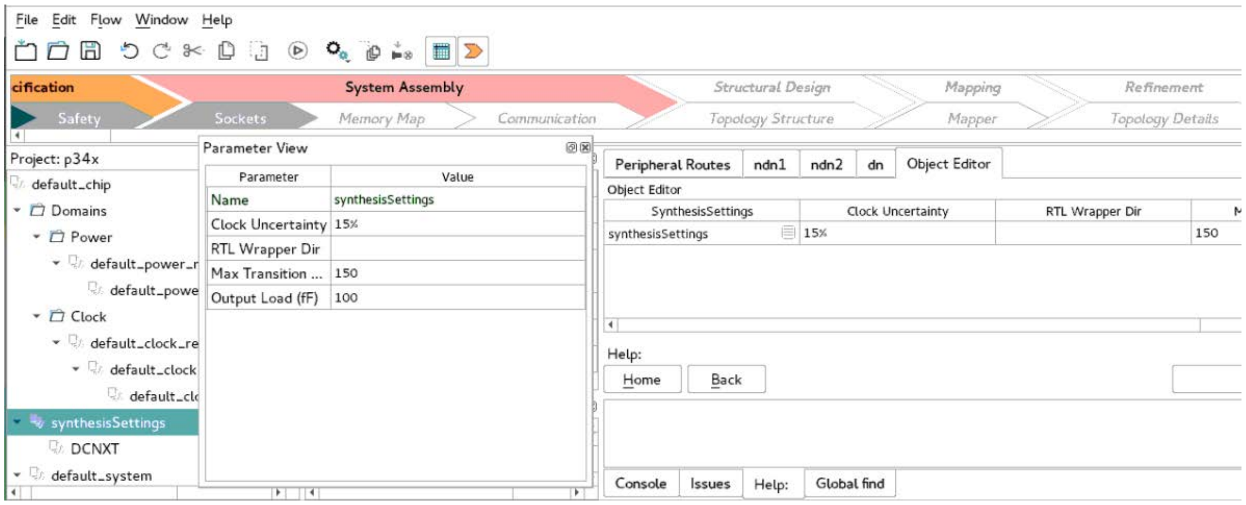
6.1.1 Synopsys DCNXT

Inside the `scripts` directory is a makefile which is used to launch the synthesis job. If synthesis is selected to be “Bottom Up”, then there will be a subdirectory for each Ncore unit along with a subdirectory named `gen_wrapper` which is the top level of the design. In the Bottom up synthesis flow, each Ncore unit is synthesized first and then the blocks are read into the top level as it is synthesized. If the Bottom Up synthesis is not selected then the design is synthesized flat and the only subdirectory present is the `gen_wrapper`.

Prior to running the synthesis flow, you must configure the `dc_setup.tcl` file with appropriate library information. This includes items like library pointers, tech file, TLU+ files, min/max layer, etc.

The synthesis settings bar in Maestro allows you to customize the the various settings which impact script generation (e.g., SDC) and/or the synthesis flow.

Figure 6. Synthesis Settings Bar



Parameter	Value
Name	synthesisSettings
Clock Uncertainty	15%
RTL Wrapper Dir	
Max Transition ...	150
Output Load (fF)	100

Object Editor

SynthesisSettings	Clock Uncertainty	RTL Wrapper Dir	M
synthesisSettings	15%		150

The description of the synthesis is listed below:

1. Compile and Link Only: Generates a synthesis script which reads in all the RTL and other specified files and stops after doing an initial link. It is used to make sure all the required files are specified and there are no unresolved references in the design.
2. Technology: Custom generates a technology template file which contains the variables which need to be filled in with the user's technology library information before running synthesis.
3. Hard Macros: Specifies the location and names of the hard macros in the design.
4. Bottom Up: If selected exports synthesis scripts for a hierarchical synthesis run. Ncore units are synthesized individually first then compiled into the top level If Bottom Up is not selected, the design is written out flat into one directory and all the logic is synthesized at once.
5. Incremental: If Incremental and Bottom Up are both selected, then the block will be incrementally re-synthesized during the top level synthesis.
6. ULVT: This sets the percentage limit on the amount of ULVT cells allowed for use during synthesis.
7. Compile command: This is the DC compile command used in scripts along with the base options.

6.1.2 Synthesis Script Files

The synthesis scripts are generated into the `<output_dir>/synthesis/DCNXT/scripts` directory. At the top level of the directory is a makefile and one or more subdirectories that contains the design. Each of the subdirectories contains the following files:

`dc.tcl`	Top level synthesis Tcl script.
`dc_config.json`	Synthesis configuration JSON file.
`dc_setup.tcl`	Synthesis template file needing to be filled in with customer technology libraries and parasitic information
`<design>.func.sdc`	Functional mode synthesis timing constraints SDC file.
`<design>.cdc.sdc`	SDC mode synthesis timing constraints SDC file.
`<design>.read_verilog.tcl`	Synthesis Verilog read-in Tcl file.
`dc_procs.tcl`	Synthesis utilities Tcl file.
`json_tcl.tcl`	JSON parser Tcl file.
`json_read.tcl`	JSON read utilities Tcl file.

6.1.3 Running Synthesis

A makefile is created in the `<output_dir>/DCNXT/scripts` directory. Typing `make` will launch the synthesis of Ncore. If the synthesis is hierarchical, the makefile will synthesize each of the Ncore modules first before synthesizing the top level.

6.1.4 Synthesis Options

This section describes the DCNXT options in the `dc_config.json` file.

`topDesign`	Top design name
`hardMacroDbs`	Hard macro db files
`checkOnly`	Pre-compile check only
`topLevelCompileOnly`	Perform top-level compile by reading in all sub modules located in the `scripts/outputs` directory
`ungroupStartLevel`	Ungroup start level
`compileCommand`	Compile command that's used in the synthesis
`incrementalOpt`	Incremental optimization which is the second compile in the synthesis run
`ulvtPercentage`	Percentage of ulvt cell to be used in the synthesis run to improve timing
`maxPaths`	Maximum number of paths in the timing report

The `dc_setup.tcl` file is a template file which requires process and technology information. The required information is dependent on whether a wireload model or physical based synthesis is being performed.

Map (1st run) Compile Flow/Options

The first compile maintains the design hierarchy, but may prevent cross-module area and timing optimization. This section describes the options that are set by the script.

`set_app_var hdlin_preserve_sequential true`	Preserves unloaded sequential cells when reading in the design.

6.1.5 Synthesis Reports

The synthesis run writes all reports to the `reports/` folder or the `hierarchical_reports/` folder, depending on the type of synthesis run.

`<design>.check_design.precompile.rpt`	Check design report file for pre-compile design
`<design>.check_timing.precompile.rpt`	Check timing report file for pre-compile design
`<design>.hierarchy.precompile.rpt`	Hierarchy report file for pre-compile design
`<design>.hierarchy_reference.precompilerpt`	Hierarchy reference report file for pre-compile design
`<design>.area.rpt`	Area report
`<design>.check_timing.rpt`	Check timing report
`<design>.clock_gating.rpt`	Clock gating report for gated registers
`<design>.clock_gating_ungated.rpt`	Clock gating report for ungated registers
`<design>.constraints.rpt`	Constraints report
`<design>.design.rpt`	Design report
`<design>.dont_touch.rpt`	Don't touch report
`<design>.hierarchy.rpt`	Hierarchy report
`<design>.input_ports.rpt`	Input ports report
`<design>.link.rpt`	Link report
`<design>.net`	Net report
`<design>.output_ports.rpt`	Output ports report
`<design>.physical_constraints.rpt`	Physical constraints report
`<design>.power`	Power report
`<design>.qor.rpt`	Quality of result report
`<design>.reference.rpt`	Reference report
`<design>.timing_max.<clock>.rpt`	Timing max corner for <clock> report
`<design>.timing_max.feedthru.rpt`	Timing max corner for feed-through report

`<design>.timing_max.flop2-flop.rpt`	Timing max corner for flop-to-flop report
`<design>.timing_max.flop2io.rpt`	Timing max corner for flop-to-io report
`<design>.timing_max.io2flop.rpt`	Timing max corner for io-to-flop report
`<design>.timing.rpt`	Timing report
`<design>.track.rpt`	Track report
`<design>.vt.rpt`	Voltage threshold report

6.2 Synopsys RTLA

6.2.1 Scripts

Inside the `scripts` directory are directories for each unit in the design. There will be a subdirectory for each Ncore unit along with a subdirectory named `gen_wrapper` which is the top level of the design.

Prior to running the synthesis flow, you must configure the `flow_setup.tcl` file. This includes items like library pointers, tech file, TLU+ files, min/max layer, etc.

6.2.2 Flow Steps

- `flow_setup.tcl` – Edit to include required technology and design specific information.
- `block_setup.tcl` – Used to override any flow_setup information for this design.
- `initDesign.tcl` – Reads the RTL and links the design.
- `rtlCond.tcl` – elaborate and prepare the design for virtual optimization.
- `rtlOpt.tcl` – completes virtual optimization such that the design is ready for analysis.
- `metrics.tcl` – write out a defined set of reports.
- `viewCopy.tcl` –open a design view and copy it to a different design name.
- `viewInPlace.tcl` – open a design view for analysis.

6.3 Cadence Genus

6.3.1 Scripts

- `flow_setup.tcl` – Edit to include required information
- `block_setup.tcl` – Used to override any flow_setup information for this design
- `genus.tcl` – Defines flow from reading RTL to generating reports

6.3.2 Timing Scenarios

The synthesis flows are set up to use two timing scenarios (aka views).

- First scenario — represents the functional mode of operation and is `<design>.func.sdc`.
- Second scenario — is a special check associated with Clock Domain Crossing (CDC) circuitry and is `<design>.cdc.sdc`. The CDC constraints (i.e., `set_max_delay`) help ensure proper functionality. For blocks without CDC circuitry, the `<design>.cdc.sdc` will be essentially empty.

7 Implementation Considerations

The following sections provide guidance and comments on frequently encountered implementation topics.

7.1 Timing Constraints

Implementation timing requirements and characteristics for a block are provided via the usage of two scenarios (aka views). The first scenario (e.g., `gen_wrapper.func.sdc`) represents the functional mode timing. Given the ncore is highly single cycle synchronous, these constraints are straightforward.

The second scenario (e.g., `gen_wrapper.cdc.sdc`) represents timing exceptions involving clock domain crossings (i.e., CDC). An attempt has been made to explicitly constrain all expected exceptions paths. If an asynchronous path is encountered that cannot be identified in `<>.cdc.sdc`, contact Arteris support.

Based on the configuration of the ncore, there are multiple different types of exceptions that could be present in the `<>.cdc.sdc`. The exception types are written out in groups in the SDC; each group is preceded by basic comments for end-user benefit as well as group-specific tuning variable(s). The exceptions should be a subset of the following (applicable tuning variable(s)).

- Write pointer, of asynchronous clock adapter (CDC_Scaler)
- Read pointer of asynchronous clock adapter (CDC_Scaler)
- DataPath of asynchronous clock adapter (CDC_Scaler, CDC_DataPathRatio)
- Functional Safety Controller (FSDC_CDC_Delay)
- Externally Driven Control (EDC_CDC_Delay)
- Trigger Control (TRIG_CDC_Delay)
- Event Control (EVT_CDC_Delay)
- Interrupt Signals (IRQ_CDC_Delay)
- PMA handshake (PWR_CDC_Delay)

One form of exception timing includes explicit set_max_delay constraints on gray code pointers. These pointers must meet these constraints to ensure proper operation. Other exception timing includes explicit set_max_delay constraints on various asynchronous control signals. Often timing on these signals can be relaxed, and by default the delay is often a function of the slowest clock in the system. These can easily be relaxed or tightened by the end-user by the modification of a variable in the SDC.

The previously mentioned internal asynchronous paths are architected to be asynchronous. It may turn out that the source and destination clocks are the same. That being the case these paths will get timed in the func scenario as synchronous. If these turn out to be problematic the end-user is free to add a false path or multi-cycle path in the func scenario address. It is expected this will be addressed in a future release with an improvement to `<>.func.sdc`.

Some of the fore mentioned asynchronous paths travel through an output (/input) ports of the ncore. It may turn out that when the ncore is instantiated in the context of an SoC, the source and destination FFs are clocked by the same functional clock. If the end-user wishes to treat these paths as multi- cycle paths, they are free to modify the constraints to do so.

The top level (e.g., gen_wrapper) is the most complete and robust SDC and should be used for budgeting and partitioning. There are `<unit>.func.sdc` and `<unit>.cdc.sdc` for all units (e.g., dce, ioaiu_top, ndn1, gen_wrapper). The SDC associated with the other units may be slightly less accurate and complete due to nuances of RTL budgeting, etc.

7.2 CDC Considerations

Synopsys VC Static with CDC has been done. The following checks: “setup”, “integ”, “sync”, “struct” (minus CDC_COHERENCY_*) have been performed. Due to architectural divergence and convergence across asynchronous boundaries it makes no sense to perform CDC_COHRENCY_* checks on ncore. The results are considered clean. Contact Arteris support if you encounter issues.

7.3 RDC Considerations

Synopsys VC Static with RDC has been done. The following checks: “setup”, “integ”, “corruption”, “conv”, “glitch” have been performed. Contact Arteris support if you encounter issues.

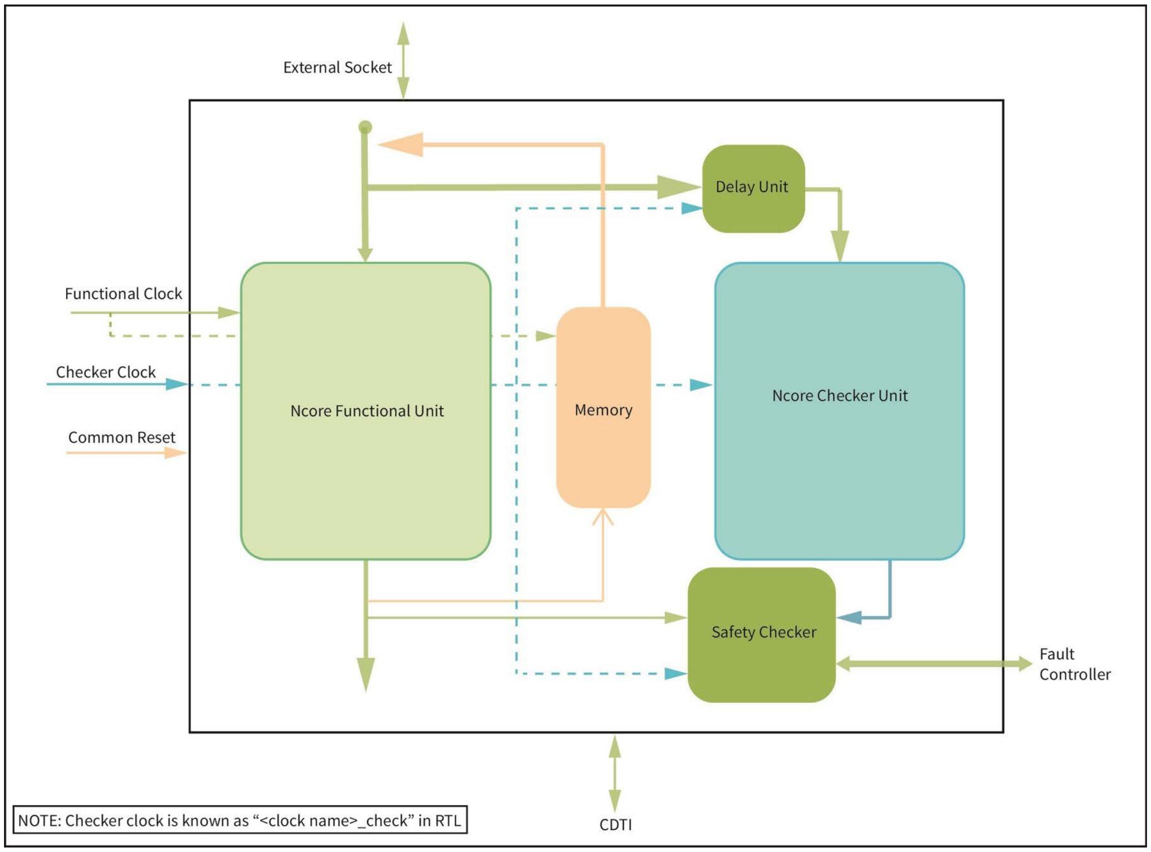
7.4 Physical Implementation Considerations

7.4.1 Resiliency Integration

Ncore designs can be specified as resilient. Ncore supports several different features to support resiliency. Duplication is one of the features for resilient designs in Ncore; when enabled, the duplication units are driven by a checker clock as illustrated in Figure 7.

1. Both clock ports must be driven by the same synchronous clock source.
2. Both clock trees should be balanced with each other.

Figure 7. Modules with separate clock ports



7.4.2 Integrating External Interface Protection

External Interface Protection is an optional resiliency feature. The clock and the reset connected to this module is the same as the ones specified in the component where the protection modules are instantiated. The protection modules are also referred to as placeholders and are configurable for additional signals.

The connections between the protection module and the resilient Ncore component are synchronous. The synchronous timing constrains specified in section 2.3 apply to them. For CHI-E and CHI-B an interface protection logic is available and can be selected with the GUI.