# Speedster7t PCIe Reference Design Guide (RD020)

Achronix Data Acceleration

**May 18, 2021** **Reference Design**

## Introduction

The Speedster®7t PCIe reference design illustrates how to generate ingress and egress PCIe traffic, to and from the FPGA fabric as well as the memory subsystems. This design showcases the network-on-chip (NoC) connection of both the PCIe ×8 and PCIe ×16 interfaces to logic in the FPGA fabric, including all the GDDR6 channels, and the DDR4 interface. The design demonstrates how to perform memory writes and reads from a PCIe interface to the various memory subsystems on the AC7t1500, writes and reads to BRAM in the FPGA fabric, as well as writes and reads to register file space in the FPGA fabric. Additionally, the reference design demonstrates master logic in the FPGA fabric initiating transactions to both the PCIe ×8 and PCIe ×16 interfaces. Both PCIe interfaces work in parallel and can send transactions to the memory subsystems and the logic in the FPGA core. Similarly, master logic in the FPGA fabric can send transactions to either PCIe interface. The design achieves a target frequency of 500 MHz for the FPGA core logic.

### Simulation

The design can be simulated in one of two modes:

- BFM mode – The design uses a bus functional model (BFM) to model the PCIe subsystem. The simulations are not cycle accurate within the PCIe portion of the design. However, the testbench can be used for some initial performance measurements, such as testing maximum traffic throughput assuming an idealized PCIe, or latency to and from each of the targets. The PCIe BFMs include SystemVerilog tasks that can be called to initiate DMA writes and reads, register writes and reads, and tasks that respond to transactions initiated from the FPGA fabric.
- RTL mode – The simulation uses encrypted RTL of the full PCIe subsystem within the device. The simulation is hence cycle accurate and represents exactly how the device operates. This simulation requires the use of a simulation model of a PCIe root-complex in order to interface with and configure the PCIe end-point within the device. In the provided design, the Synopsys PCIe verification IP (VIP) is used (see also Obtaining and Installing Synopsys PCIe VIP (RTL Mode Only) (see page 13)). Further obtain the appropriate Device Simulation Model, (DSM), package containing the PCIe RTL from Achronix. A request for this package can be made at support@achronix.com

> **Note**
>
> In BFM simulation mode, the reference design does not support any PCIe protocols or PIPE interfaces, and there is no access to PCIe configuration registers.

By default, this PCIe reference design is configured to use BFMs for both the PCIe interfaces, and the GDDR6 and DDR4 subsystems. RTL simulations of the PCIe interfaces, or any of the memory subsystems can be enabled. Details on enabling the RTL models are included in Enabling RTL Simulation Models (see page )

> **Warning**
>
> Enabling RTL mode for either of the PCIe subsystems, or any of the memory subsystems can substantially increase simulation time.

## Core Design

The reference design demonstrates how to incorporate connections between the PCIe interfaces, the FPGA fabric, and the memory subsystems into a new design. The core design within the FPGA fabric does not change based on the simulation modes. The simulation mode only changes the interface subsystems surrounding the user design from BFMs to full RTL, it does not affect the core design. Changing the simulation mode does affect the cycle accuracy of the simulation, but not the functional behavior. This method allows developing a design quickly using the BFM modes of all the interface subsystems. If required, cycle accurate RTL simulations can be run to fully confirm design integrity.

# Description

The reference design includes a testbench for simulation, as well as scripts to implement the design in the AC7t1500 device.

As described above, the PCIe reference design has two modes of simulation, BFM and RTL. The associated testbench for each mode is different, although they are defined in a single SystemVerilog source file `tb_pcie_ref_design.sv`. The architecture of these testbenches are detailed below.

In BFM mode, the functionality of PCIe IP and the Root Complex that would be interacting with the FPGA are all modeled by the BFM. The test calls SystemVerilog tasks that are defined in the PCIe ×8 and PCIe ×16 master and slave BFMs. The structure of the testbench is shown below:

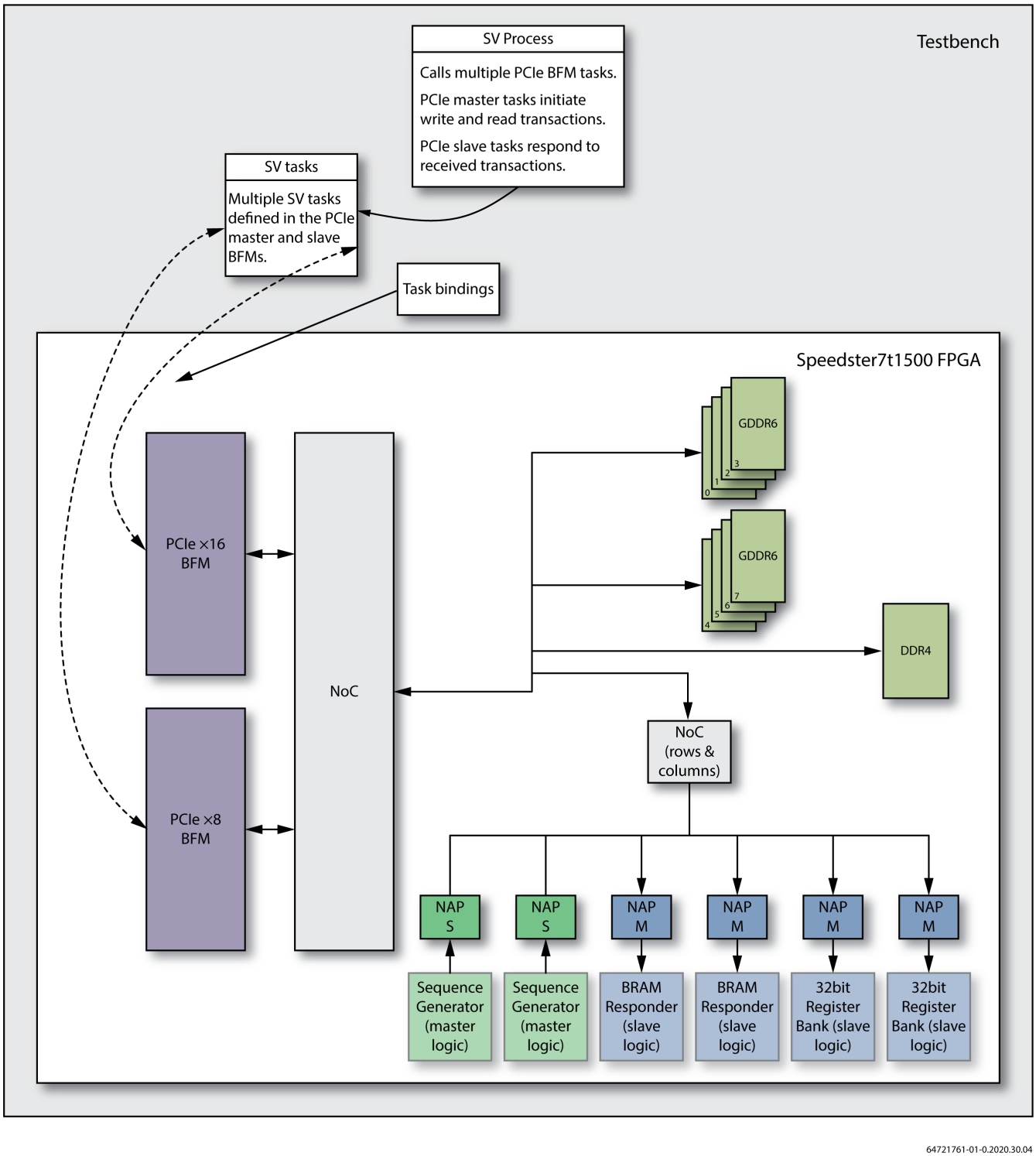


**Figure 1:** ***PCIe Reference Design BFM Mode Testbench Block Diagram***

In RTL mode, the Synopsys PCIe VIP is used to model the Root Complex (RC). Two RCs are instantiated and interact with the PCIe ×8 and PCIe ×16 subsystems in the FPGA as shown below.

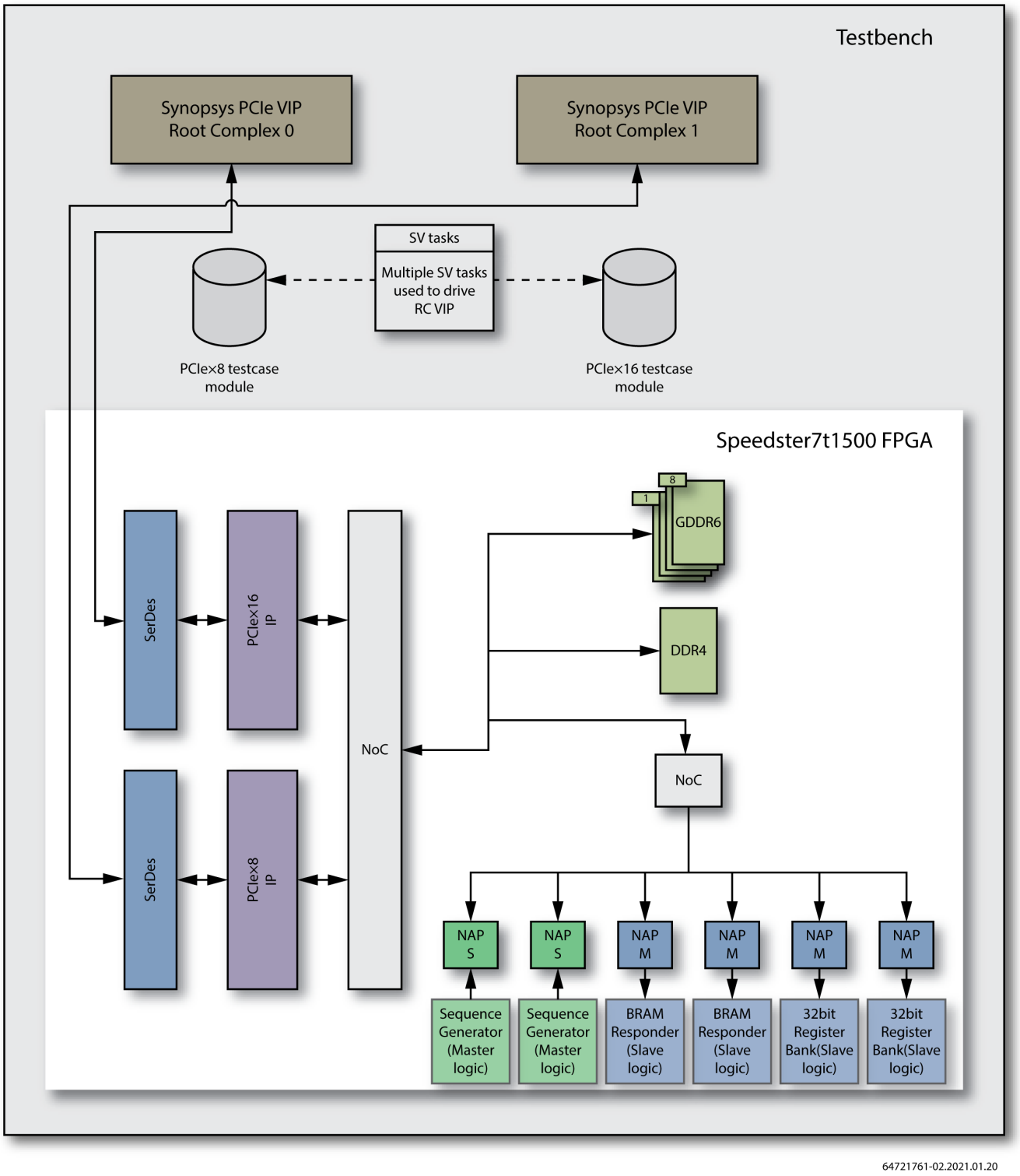


**Figure 2:** ***PCIe Reference Design RTL Mode Testbench Block Diagram***

In both modes, the design in the FPGA is the same and exercises the different connections between the two PCIe interfaces, the memory subsystems, and the FPGA fabric. All portions of the design use random generators and checkers to generate the data, and perform self-checking compares on any responses.

> **Note**
>
> The PCIe reference design is only available in the full-chip BFM or RTL simulation flow, and is not available for the standalone simulation flow.

## DMA Traffic to Memory Subsystems

The PCIe interfaces, as well as the GDDR6 channels and DDR4 interfaces are all connected to the NoC, and communicate between each other using AXI transactions. The testbench generates random data, with incrementing addresses and increasing burst lengths to write to all the memory subsystems. The testbench makes use of task calls to send and receive the transactions. The destination GDDR6 or DDR4 port is specified by using the global address space of the NoC when calling the write or read task. A compare task in the testbench checks whether the read data returned matches the expected data pattern that was written.

## DMA Traffic to BRAM in FPGA Fabric

Similar to the memory subsystem transactions, the PCIe interfaces can send transactions to a BRAM responder residing in the FPGA fabric. This reference design makes use of the NoC connection to the fabric. The BRAM responder logic connects to an AXI master NAP on a column of the NoC. The logic in the FPGA fabric receives the AXI transactions from the PCIe interfaces and writes data into the BRAMs. On a read transaction, the BRAM sends back the read data from the address specified. Specifically, the BRAM responder, `i_axi_bram_rsp1`, receives transactions from the PCIe ×8 interface while the BRAM responder, `i_axi_bram_rsp2`, receives transactions from the PCIe ×16 interface. Both BRAM responders are located at specific NAP locations which can be found in the `bind` statements in the testbench and in the `ace_placements.pdc` file for implementation. The specific NAP locations can be changed if desired. The testbench calls tasks to initiate the write and read transactions and checks whether the data returned matches the data that was written to the BRAM.

## Register Writes and Reads to FPGA Fabric

This reference design includes several different register types located in the FPGA fabric. The PCIe interfaces can send write and read transactions to these registers which perform various tasks and connect to an AXI master NAP on a column. Specifically, the `i_axi_nap_reg_set1` register set receives transactions from the PCIe ×8 interface while the `i_axi_nap_reg_set2` register set receives transactions from the PCIe ×16 interface. The register sets are located at specific NAP locations which are specified in the `bind` statements in the testbench and in the `ace_placements.pdc` file for implementation. The specific NAP locations can be changed if desired. The testbench calls appropriate tasks to initiate the transactions and compares the data returned to the expected data to determine whether they match. The different register types available are listed below:

- Eight 32-bit read/write registers.
- Eight 32-bit read only registers. Each register is set to a fixed known value, address plus bias value.
- Two 32-bit counter registers with two control registers (one up counter, one down counter, and one control register for each counter that starts, stops and clears the counter, for a total of four registers).
- Three IRQ mimic registers: Two IRQ registers (directly writable or randomly set) and one IRQ master. When a specific IRQ register bit is written to, the respective bit is cleared. The IRQ master register is set to 1 if any of the IRQ bits are set (effectively an OR of the IRQ bits).
- One clear-on-read register. Write to set bits within this 32-bit register. All 32 bits clear on a read, meaning the second read of the register returns all zeros.
- Four 64-bit read/write registers.

## Master FPGA Logic to PCIe

Master logic in the FPGA can send transactions to the PCIe slave interfaces. In this reference design, there are two blocks of master logic in the FPGA fabric. Each generates random data along with incrementing addresses and increasing burst lengths. Each block connects to an AXI slave NAP which sends transactions to the PCIe interface. One master logic block, `i_pcie16_axi_gen_chk`, sends transactions to the PCIe ×16 interface while the second block, `i_pcie8_axi_gen_chk`, sends transactions to the PCIe ×8 interface.

The master logic blocks are placed at specific NAP locations which are specified in the `bind` statements in the testbench, as well as in the `ace_placements.pdc` file for implementation. The specific NAP locations can be changed if desired.

In BFM mode, the testbench makes task calls to check for write and read requests in the corresponding PCIe slave BFMs. The `get_write_request` task processes the write request, captures the data, and saves it in an array of bytes. The testbench then calls the `issue_write_response` to indicate that the write transaction completed successfully. The checker logic in the FPGA sends read transactions to the PCIe slave. The testbench calls the tasks `get_read_request` and `issue_read_response` to send the read data back to the checker in the FPGA fabric. The checker logic uses the same pattern as the generator logic that sends the write transactions to the PCIe slave. As the read data is received in the checker, the received data is compared against the expected data.

In RTL mode, the RC VIP automatically handles the incoming write and read access from the same data generator and checker logic in the FPGA to facilitate testing of the function.

# Testbench Environment

## SystemVerilog Tasks

In BFM mode, the simulation testbench calls several high-level tasks to provide connectivity to/from the PCIe interfaces. The following tasks are available:

**Table 1: *PCIe Tasks in BFM Mode***

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| `write_dma:` | | | PCIe master task for writing to memory subsystems, or BRAMs in the FPGA fabric. Must be preceded by `set_blocking_transaction_id` task to set the AXI transaction ID. |
| byte_addr | Input | 41:0 | Starting byte address of the transaction, must be aligned on 64-byte boundary. |
| num_bytes | Input | 31:0 | Number of bytes in the transfer, maximum transfer is $2^{32}-1$. |
| data[int] | Input | 7:0 | Pointer to the first byte of write data; this is an array of bytes. |

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| resp | Output | 1:0 | The response to the transaction (AXI bresp):<br>• 2'b00 = transaction succeeded<br>• 2'b01 = task error (arguments illegal or timed out)<br>• 2'b10 = slave error (error returned by slave)<br>• 2'b11 = decode error (destination address is not valid) |
| `read_dma:` | | | PCIe master task for reading from memory subsystems, or BRAMs in the FPGA fabric. Must be preceded by `set_blocking_transaction_id` task to set the AXI transaction ID. |
| byte_addr | Input | 41:0 | Starting byte address of the transaction, must be aligned on 64-byte boundary. |
| num_bytes | Input | 31:0 | Number of bytes in the transfer, maximum transfer is $2^{32}-1$. |
| data[int] | Output | 7:0 | Pointer to the first byte of read data, this is an array of bytes. |
| resp | Output | 1:0 | The response to the transaction (AXI rresp):<br>• 2'b00 = transaction succeeded<br>• 2'b01 = task error (arguments illegal or timed out)<br>• 2'b10 = slave error (error returned by slave)<br>• 2'b11 = decode error (destination address is not valid) |
| **`write:`** | | | PCIe master task for writing to register space or other slave logic in the FPGA fabric or CSR space. Must be preceded by `set_blocking_transaction_id` task to set the AXI transaction ID. |
| byte_addr | Input | 41:0 | Starting byte address of the transaction, must be aligned on the transfer size boundary, for example a 4-byte transfer must be aligned on a 4-byte boundary. |
| data[int] | Input | 7:0 | Pointer to the first byte of write data, this is an array of bytes. |
| num_bytes | Input | 31:0 | Number of bytes in the transfer, minimum value is 4 bytes, maximum value is 512 bytes for PCIex16, 1024 for PCIex8. |
| resp | Output | 1:0 | The response to the transaction (AXI bresp):<br>• 2'b00 = transaction succeeded<br>• 2'b01 = task error (arguments illegal or timed out)<br>• 2'b10 = slave error (error returned by slave)<br>• 2'b11 = decode error (destination address is not valid) |
| `read:` | | | PCIe master task for reading from register space or other slave logic in the FPGA fabric or CSR space. Must be preceded by `set_blocking_transaction_id` task to set the AXI transaction ID. |

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| byte_addr | Input | 41:0 | Starting byte address of the transaction, must be aligned on the transfer size boundary, for example a 4-byte transfer must be aligned on a 4-byte boundary. |
| num_bytes | Input | 31:0 | Number of bytes in the transfer, minimum value is 4 bytes, maximum value is 512 bytes for PCIex16, 1024 for PCIex8. |
| data[int] | Output | 7:0 | Pointer to the first byte of read data, this is an array of bytes. |
| resp | Output | 1:0 | The response to the transaction (AXI rresp):<br>• 2'b00 = transaction succeeded<br>• 2'b01 = task error (arguments illegal or timed out)<br>• 2'b10 = slave error (error returned by slave)<br>• 2'b11 = decode error (destination address is not valid) |
| `get_write_request:` | | | PCIe slave task, waits for write request from FPGA fabric or other master and captures write address, write data, and AXI transaction ID. |
| byte_addr | Output | 41:0 | Starting byte address for the write request. |
| num_bytes | Output | 31:0 | Number of bytes in the transfer. |
| data[int] | Output | 7:0 | Pointer to the first byte of write data, this is an array of bytes. |
| id | Output | 7:0 | AXI transaction ID. |
| `issue_write_response:` | | | PCIe slave task, sends a write response to the preceding write transaction. |
| resp | Input | 1:0 | The response to the AXI write transaction (AXI bresp). |
| id | Input | 7:0 | AXI transaction ID (same ID returned from the `get_write_request` task). |
| `get_read_request:` | | | PCIe slave task, waits for read request from FPGA fabric or other master and captures the read address, length of transfer, and AXI transaction ID. |
| byte_addr | Output | 41:0 | Starting byte address for the read request received. |
| num_bytes | Output | 31:0 | Number of bytes in the transfer. |
| id | Output | 7:0 | AXI transaction ID. |
| `issue_read_response:` | | | PCIe slave task, sends read response to the preceding read request includes read data of the correct transfer length. |
| num_bytes | Input | 31:0 | Number of bytes in the read transfer, use the `num_bytes` received from `get_read_request` task. |
| data[int] | Input | 7:0 | Pointer to the first byte of read data, this is an array of bytes. |
| resp | Input | 1:0 | The response to the transaction (AXI rresp). |

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| id | Input | 7:0 | AXI transaction ID (same ID returned from the `get_read_request` task). |
| `set_blocking_transaction_id:` | | | PCIe master task, sets the AXI transaction ID for the subsequent blocking AXI transaction(s). |
| transaction_id | Input | 7:0 | Sets the AXI transaction ID to be used by the subsequent blocking transactions, can optionally use to set the transaction ID but not required. |
| `wait_cycles:` | | | Wait a number of PCIe cycles before proceeding to the next task. This task allows the testbench to keep PCIe BFM tasks aligned with the corresponding PCIe clock. |
| num_cycles | Input | 31:0 | The total number of PCIe cycles to advance the clock. |

In RTL mode, the simulation testbench calls several high-level tasks to drive the behavior of the RC VIP. The following tasks are available:

**Table 2: *PCIe Tasks in RTL Mode***

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| `MemRandWrRdComp0/1:` | | | PCIe VIP acting as RC0/1 issues a command to write a random data payload to the FPGA's memory space, reads back the data and compares. If read data mismatches write data, an error message is generated. |
| address | Input | 63:0 | Starting byte address of the transaction, must be aligned on 64-byte boundary. |
| length_in_dwords | Input | 31:0 | Number of DWORDs in the transfer. This value needs to be greater than 1. |
| tc | Input | 31:0 | Traffic class. |
| `NAPRandWrRdComp0/1:` | | | PCIe VIP acting as RC0/1 issues a command to write a random data payload to a specific NoC access point (NAP) in the FPGA, reads back the data and compares. If read data mismatches write data, an error message is generated. |
| nap_base | Input | 63:0 | Base address of the NAP. |
| col | Input | 3:0 | Column number of the NAP. |
| row | Input | 2:0 | Row number of the NAP. |
| length_in_dwords | Input | 31:0 | Number of DWORD in the transfer. This value needs to be greater than 1. |
| tc | Input | 31:0 | Traffic class. |

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| NAPRandWrRdCompDW0/1: | | | PCIe VIP acting as RC0/1 issues a command to write a single DWORD of random data payload to a specific NAP in the FPGA, reads back the data and compares. If read data mismatches write data, an error message is generated. |
| nap_base | Input | 63:0 | Base address of the NAP. |
| col | Input | 3:0 | Column number of the NAP. |
| row | Input | 2:0 | Row number of the NAP. |
| tc | Input | 31:0 | Traffic class. |
| NAPRdCompDW0/1: | | | PCIe VIP acting as RC0/1 issues a command to read a single DWORD from a specific NAP and compare it to the expected value. If read data mismatches expected data, an error message is generated. |
| nap_base | Input | 63:0 | Base address of the NAP. |
| col | Input | 3:0 | Column number of the NAP. |
| row | Input | 2:0 | Row number of the NAP. |
| tc | Input | 31:0 | Traffic class. |
| expected_rd_data | Input | 31:0 | Expected value of read data. |
| NAPWrDW0/1: | | | PCIe VIP acting as RC0/1 issues a command to write a single DWORD of data payload to a specific NAP. |
| nap_base | Input | 63:0 | Base address of the NAP. |
| col | Input | 3:0 | Column number of the NAP. |
| row | Input | 2:0 | Row number of the NAP. |
| tc | Input | 31:0 | Traffic class. |
| wr_data | Input | 31:0 | Write data. |
| NAPRdDW0/1: | | | PCIe VIP acting as RC0/1 issues a command to read a single DWORD of data from a specific NAP. |
| nap_base | Input | 63:0 | Base address of the NAP. |
| col | Input | 3:0 | Column number of the NAP. |
| row | Input | 2:0 | Row number of the NAP. |
| tc | Input | 31:0 | Traffic class. |
| rd_data | Output | 31:0 | Write data. |

| Signal Name | Direction | Width | Description |
|---|---|---|---|
| `DoInitLinkUpRcEp0/1:` | | | This task establishes a link between the RC0/1 VIP and the FPGA EP. |
| model_cfg | Input | class | This is a class that defines the characteristics of the PCIe Link to be created. Details of this class are defined in `pciesvc_device_serdes_x16_model_config.sv`. |
| `CheckTestStatus:` | | | This task checks the number of "NOTICE", "WARNING" and "ERROR" generated by the RC VIP. |

## Organization

The reference design makes use of SystemVerilog interfaces to simplify the signals for each NAP. The AXI4 ( `t_AXI4`) interface definition is included, alongside the NAP wrapper files that make use of the interface to instantiate the associated master or slave NAP. The design structure presents examples of how NAPs can be used in a design; however, if it is preferred that SystemVerilog interfaces are not used, the NAP macros can simply be instantiated directly without the wrapper or interface definition.

## Controlling the Simulation Flow

The reference design supports both BFM- and RTL-mode simulation. Selecting between these two modes and controlling options during the simulation is achieved by using switches during the make flow.

To run the simulation in BFM mode, under `sim/vcs` or `sim/questa` directory, type:

```
> make FLOW=FULLCHIP_BFM
```

or simply type:

```
> make
```

…as the make flow defaults to BFM mode when the FLOW switch is not explicitly given.

To run the simulation in RTL mode, under "`sim/vcs`" or "`sim/questa`" directory, type:

```
> make FLOW=FULLCHIP_RTL
```

The waveform dump can also be controlled from the make flow. Disabling waveform dumping speeds up the simulation. By default waveform dumping is enabled. To disable it, add the WAVE_DUMP=NO switch in the make command, for example:

```
> make FLOW=FULLCHIP_RTL WAVE_DUMP=NO
```

## Enabling RTL simulation models

When FULLCHIP_RTL flow is selected, (see above), the simulation makes use of the `system_files_rtl.f` file, within the respective simulation directory, to configure the respective interface subsystems as either BFM or RTL. This file is pre-configured to enable both PCIe subsystems as RTL, with all memory subsystems left as BFM.

When in FULLCHIP_BFM mode, (default), the simulation uses `system_files_bfm.f`, which is pre-configured to set all interface subsystems to BFM, (PCIe and memory).

To enable RTL simulation models for any of the memory subsystems, it is necessary to add the following defines to the respective `system_files_xxx.f`.

```
# Enable GDDR RTL
# Enable the desired GDDR memory controllers below
# Any undefined controllers will use their BFM model
//+define+ACX_GDDR6_0_FULL
//+define+ACX_GDDR6_1_FULL
//+define+ACX_GDDR6_2_FULL
//+define+ACX_GDDR6_3_FULL
//+define+ACX_GDDR6_4_FULL
//+define+ACX_GDDR6_5_FULL
//+define+ACX_GDDR6_6_FULL
//+define+ACX_GDDR6_7_FULL

# Enable the DDR4 memory controller RTL
//+define+ACX_DDR4_FULL

# Turn-on below defines for DDR4 Micron Model
//+define+ACX_USE_MICRON_MODEL
//+define+ACX_DDR4_3200
//+define+DDR4_X8
//+define+DDR4_2G

# Turn-on below defines for DDR4 SKHynix Model
//+define+ACX_USE_HYNIX_MODEL
//+define+DDR4_8Gx8
//+define+DDR4_3200AA
//+define+ACX_DDR4_3200
```

**Note**

When enabling RTL simulations for any of the memory subsystems, the appropriate memory device simulation model must be instantiated in the testbench. Refer to the relevant reference design of each of the memory subsystems for details of obtaining and connecting these memory device simulation models.

## Testbench Configuration Files, (RTL Mode Only)

Within the testbench, for RTL mode simulation, there are two configurations files (`/sim /Gen5_8lanes_PCIE_0_mem64bit_bitstream0.txt` and `/sim /Gen5_16lanes_PCIE_1_mem64bit_bitstream0.txt`). These are used to configure the respective PCIe cores when RTL simulation mode is used.

> **Warning**
>
> When used in simulation, the configuration specified by these files bypasses the PCIe Host-to-Endpoint handshake and programs the BAR registers directly. This action accelerates and simplifies the simulation flow. In silicon it is still necessary for the host, at start-up, to read the PCIe core capabilities, and to then configure the BAR registers based on these capabilities and the hosts requirements.
>
> Due to these amendments, these configuration files must be used as supplied. It is not possible to replace them with updated files generated directly from ACE. Subsequent releases of this reference design will not include these amendments allowing the generation of configuration files directly from ACE.

# Obtaining and Installing Synopsys PCIe VIP (RTL Mode Only)

In order to simulate the design in RTL mode, obtain a PCIe verification IP (VIP) package and license from Synopsys.

> **Note**
>
> The PCIe VIP is not required for BFM mode simulation.

## How to Obtain PCIe VIP and License

Contact Achronix Support to get instructions on how to obtain Synopsys PCIe VIP license.

## How to Install the PCIe VIP Package

1. Download the VIP package from Synopsys and unzip it to a known location, `user_synopsys_pcie_vip_dir`. This directory contains all PCIe VIP components.
2. Set an environment variable named **`DESIGNWARE_HOME`** to the location of `user_synopsys_pcie_vip_dir.`

3. Create a new directory for the VIP design files, for example, `user_design_dir`. Change directory to this location and execute:

   ```
   > $DESIGNWARE_HOME/bin/dw_vip_setup -e pcie_svt/tb_pcie_svt_verilog_basic_sys -svlog
   ```

   This script then builds a set of VIP utilities and test case components under `user_design_dir`.
4. Set an environment variable named **`DESIGN_DIR`** to the location of `user_design_dir`.
5. **Mentor QuestaSim only**: Define an environment variable **`MTI_HOME`** which points to the location of the Mentor QuestaSim installation.

The installation of the Synopsys VIP package is now complete.

When the simulation makefiles are run, a recipe titled `make_pli` compiles the Synopsys VIP for the target simulator.

> **Note**
>
> The two environment variables above ( **`DESIGNWARE_HOME`** and `DESIGN_DIR` ) are referenced by the `Makefile`, `system_files_rtl.f` and `qsim_pcie_ref_design.do` files under the `sim/vcs` and `sim/questa` simulation directories. These environment variables must be defined in the shell prior to being able to run RTL simulations.
>
> In addition, the environment variable **`MTI_HOME`** must also be defined if using Mentor QuestaSim.

# Ports

Port names starting with `i_` indicate input ports to the design; those starting with `o_` are output ports. There is a single clock in the design.

**Table 3: *PCIe Reference Design Top Ports***

| Signal Name | Width | Description |
|---|---|---|
| i_clk | 1 | Clock for all logic in the FPGA fabric, all logic runs at 500 MHz. |
| i_reset_n | 1 | Active-low reset. |
| i_start | 1 | Indicates the FPGA logic generating AXI transactions can begin. |
| pll_1_lock | 1 | PLL lock signal. |
| o_mstr_test_complete | 1 | Asserts when the master logic has completed its transactions. |
| o_mstr_test_complete_oe | 1 | Output enable for `o_mstr_test_complete`, tied to `1'b1`. |
| o_fail | 1 | Asserted when read data from PCIe slave does not match the expected data. |
| o_fail_oe | 1 | Output enable for `o_fail`, tied to `1'b1`. |

# Implementation

The reference design includes an ACE project located in `/src/ace`. The ACE project configures the PCIe subsystems using the default configurations, using the files `/axcip/pci_express_x16.acxip` and `/axcip/pci_express_x8.acxip`.

> **PCIe IP settings**
>
> Please note that the PCIe IP settings in these files are default values and do NOT directly match PCIe IP settings used in RTL mode simulation. These settings can be changed without affecting simulation.

For details of the PCIe IP settings files used in simulation please refer to TestBench Configuration Files, (RTL Mode Only) (see page )

## Clocks

The following clocks are specified within the project, using the `/acxip/pll_1.acxip` file. The PLLs are driven by `sys_clk_in`, an input to the device which is specified to be at 100 MHz.

**Table 4: *Clock Frequencies***

| Name | Frequency (MHz) | Description |
|---|---|---|
| noc_clk | 200 | Default clock for the NoC.(†) |
| pcie_clk | 1000 | PCIe subsystem reference clock.(†) |
| i_clk | 500 | Fabric core clock. |

> **Table Note**
>
> † These clock outputs connect directly from the reference PLL to their respective destinations. The clocks are not routed via the FPGA fabric, and therefore, are not available to the user logic.

## Resets

There are a number of reset sources for the design as detailed in the table below.

**Table 5: *Reset Sources***

| Source | Description |
|---|---|
| i_reset_n | External asynchronous input. Pin configuration is specified in `/acxip/gpio_bank_north.acxip`. |
| pll_1_lock | User clock PLL locked. Provides `i_clk`. |

Good design practice requires that resets are carefully synchronized and processed. This reference design uses multiple instances of the `reset_processor` module which combines multiple sources, correctly synchronizes them to the specified clock domains, and subsequently pipelines the resets to allow for fan-out control and re-timing. The output from the `reset_processor` module is a synchronous reset signal per clock domain.

The design consists of seven instances of the reset_processor for the NoC circuits, using `i_reset_n`, the PLL lock signals, and generates a reset to each NAP (`nap_rstn[6:0]`) that is synchronous to `i_clk`.

# Installing the Reference Design

## Downloading

The design is available for download on the Achronix self-service FTP site (https://secure.achronix.com), located in the folder `/public/Achronix/Reference_Designs/Speedster7t`.

## Packaging

The design is packaged in a zip file archive, using the following format:

```
<design_name>_<design_version>_<date_of_packaging>.zip
```

The archive contains all the source code, scripts to build the design, simulation script files, and optionally, GUI-based project files.

In addition the root of the archive contains release notes identifying the change history of the design.

## Operating System

### Linux

The design scripts and build flow are natively designed for Linux and have been built and tested using CentOS 7 and Ubuntu 16.04LTS. The flows use Makefiles, so are Linux shell agnostic.

### Windows

To enable the scripted simulation or implementation flows to operate under Windows 10, the user must install one of the following;

- A Linux environment installed under Windows, such as www.cygwin.com The installation should include a Tcl interpreter and the `make` executable.
- A Tcl Interpreter and a `make` executable for Windows. There are many variants available, including both no cost and licensed versions.

If the user is unable to install any of the options above, it is still possible to run some of the flows under Windows:

- Implementation – GUI projects are provided for both Synplify and ACE, enabling builds to be done using the tools directly in GUI mode.
- Simulation – A Tcl script is provided which can be executed in the QuestaSim Tcl console. This script enables simulation using QuestaSim under Windows. For further details, refer to the Simulation section.

**Note**

For correct operation of any script flow, ACE should be installed in a directory without spaces in the path name, e.g., `C:\achronix\8.3\Achronix_CAD_Environment`. Additionally the environment variable `ACE_INSTALL_DIR` must use "/" as path separators rather than "\", for example:

```
ACE_INSTALL_DIR = C:/achronix/8.3/Achronix_CAD_Environment/Achronix
```

## Tool Versions

All designs were tested with the following tools:

**Table 6: *Minimum Tool Versions***

| Software | Version |
|---|---|
| ACE | 8.3.3 |
| Device Simulation Model | 8.3.3 |
| Synplify Pro | R-2020.09X-SP1 |
| Mentor Questa | 10.7c-1 |
| Synopsys VCS | O-2018.09-SP1-1 |

## Environment Variables

### ACE_INSTALL_DIR

In order to support relocatable projects, the designs make use of an environment variable, `ACE_INSTALL_DIR`. This variable should be set to point to the ACE installation directory (where `ace.exe` or `ace` is installed). This variable is then used by both synthesis and simulation to correctly locate the ACE library files.

**Note**

For correct operation of any script flow, ACE must be installed in a directory without spaces in the path name. Examples of suitable paths are:

- Windows – `C:\achronix\8.3\Achronix_CAD_Environment`
- Linux – `/opt/achronix/ace/8.3`

Additionally when installed under Windows, the environment variable `ACE_INSTALL_DIR` must use "/" as path separators rather than "\", for example:

```
ACE_INSTALL_DIR = C:/achronix/8.3/Achronix_CAD_Environment/Achronix
```

## Directory Structure

The design has a directory structure that allows for easy navigation and separation of source and generated files. The directory structure can be easily modified to suit a users preferred layout; however, if the structure is modified, then the necessary makefiles and build scripts will need to be modified to suit. To support portability, relative paths are used, as opposed to absolute paths; with the use of environment variables to select the root directory.

**Table 7: *Design Directory Structure***

| Directory | | | Decription |
|---|---|---|---|
| <design_name> | | | Root directory. Contains release notes. |
| | /build | | Synthesis and place-and-route building |
| | /doc | | Documentation and user guide |
| | /scripts | | Scripts used for building and simulation |
| | /sim | | Simulation area |
| | | /vcs | Synopsys VCS simulation files |
| | | /questa | Mentor QuestaSim simulation files |
| | /src | | Source code |
| | | /ace | ACE GUI project |
| | | /acxip | ACE .acxip configuration files |
| | | /constraints | Placement and timing constraint files |
| | | /include | RTL include files |
| | | /ioring | ACE generated ioring files |
| | | /rtl | RTL source files |
| | | /syn | Synplify Pro GUI project |
| | | /tb | RTL testbench files |
| | | filelist.tcl | Filelist used for building and simulation |

## Language Support

The reference designs support both Verilog, SystemVerilog and VHDL RTL languages. These can be used for both building, and standalone simulation. If the full-chip BFM simulation is used, that environment requires that the top-level testbench is Verilog or SystemVerilog. However, the design under test (DUT) may be written in VHDL.

## Constraint Files

The design is supplied with a full set of constraint files, located under `/src/constraints`. These files demonstrate how various constraints and directives may be applied to the design. The constraint files, and their usage is detailed below.

**Table 8: *Constraint File Details***

| File Name | Usage |
|---|---|
| `ace_constraints.sdc` | Timing constraints used by ACE. More than one SDC file can be included in an ACE project. |
| `ace_options.sdc` | Control ACE settings, such as flow mode, speed grade, reporting of unconstrained paths. |
| `ace_placements.pdc` | Fix locations of elements within the ACE fabric, and creation of placement regions.<br>**Note**<br>Pin placements between the fabric and the I/O ring are automatically created by the I/O ring designer, and provided in the ioring PDC files. |
| `synplify_constraints.fdc` | Synplify FPGA design constraints. Set attributes such as compile points, or default memory styles. |
| `synplify_constraints.sdc` | Synplify timing constraints. Clock and timing constraints. Should match those set in `ace_constraints.sdc`. |
| `synplify_options.tcl` | Control Synplify settings, such as top module. Create synthesis specific parameters, generics and defines. |

## I/O Ring Constraint Files

In addition to the constraint files listed above, the I/O ring generates constraint files specific to the interface between the fabric core and the I/O ring containing the interface subsystems. These constraint files can be auto-generated by ACE from the respective `.acxip` files; however, to aid the build flow, pre-generated files are provided for projects that require configuration of the I/O ring interface subsystems. These files are located under `/src/ioring`. The purpose of each file is detailed below.

**Table 9: *IO Ring Constraint File Details***

| File Name | Usage |
|---|---|
| `<design_name>_ioring.sdc` | I/O timing constraints for direct connection interfaces, between the fabric and the I/O ring. |
| `<design_name>_ioring.pdc` | Placement of the fabric I/O pins, to assign them to the direct connection interfaces in the I/O ring. |
| `<design_name>_ioring_util.xml` | Used by ACE to generate a combined utilization report, combining the fabric and I/O ring resources. |

# Device Simulation Model

Many designs require a simulation overlay named the device simulation model (DSM). This package combines the full RTL of the network on chip (NoC) with bus functional models (BFMs) of the interface subsystems that surround the NoC and FPGA fabric. This combination of true RTL for the NoC and models for the interface subsystems allows users to develop their designs within a fast responsive simulation environment, while achieving cycle-accurate interfaces from the NoC, and representative cycle responses from the hard interface subsystems. This simulation environment allows a designer to iterate rapidly to develop and debug their design.

## Description

The DSM provides full RTL code for the NoC and BFM models of the surrounding interface subsystems. This structure is wrapped within a SystemVerilog module named per device, i.e., ac7t1500. Instantiate one instance of this module within the top-level testbench.

In addition, the DSM provides binding macros such that binding between elements of a design and the same elements within the device is possible. For example, the design might instantiate a NoC access point (NAP). It is then necessary to bind this NAP instance to the NAP in the correct location within the NoC by using the `ACX_BIND_NAP_SLAVE, `ACX_BIND_NAP_MASTER, `ACX_BIND_NAP_HORIZONTAL, or `ACX_BIND_NAP_VERTICAL macro, whichever is appropriate for the design.

Similarly it is necessary to bind between the ports on the design and the direct-connection interface (DCI) for the interface subsystem. Each DCI within the device is connected to a SystemVerilog interface. This interface can then be directly accessed from the top-level testbench, and signals assigned between the SystemVerilog interface and the ports on the design.

## Version Control

The DSM is version controlled. Within a release, new functions might be added and older functions might be deprecated or replaced. The release is indicated both in the package name (`ACE_<major>.<minor>.<patch>_IO_BFM_sim_<update>.zip/tgz`) and in the `readme` file placed in the root directory of the package.

To ensure that the correct version of the I/O ring simulation package is used, a task must be included within the design testbench to confirm the version compatibility. This function should be instantiated as detailed below:

**Code**

```
    // For this example the DSM instance is ac7t1500
    initial begin
        // Ensure correct version of DSM is being used
        // This design requires 8.3.3 as a minimum
        ac7t1500.require_version(8, 3, 3, 0);
    end
```

### require_version( ) Task

The require_version task has four arguments. In order

- Major Version – Matches the major version of the release
- Minor Version – Matches the minor version of the release
- Patch – Matches the patch version of the release (optional)
- Update – Matches the update number of the release (optional)

If either of patch or update is not specified, then these should be set to 0. For example, for the 8.3 release, the arguments would be set as 8,3,0,0.

> **Note**
>
> The values can be expressed either as numbers (0-9) or as strings ( "0" – "9") or as letters ("a/A", "b /B"), with the letters "a" and "b" representing alpha or beta releases. When deciding on the priority of a release, a number represents a more recent release than a letter; therefore, 8.3.alpha (defined as 8,3," a",0) precedes the full 8.3 release (designated as 8,3,0,0).

## Example Design

An example structure of a user testbench, instantiating both the DSM and the user design under test is shown in the diagram (see page 22) below. This example shows the macros required for the slave NAPs, and the DCIs for two instances of the GDDR6 subsystem. For other forms of NAPs, or for other DCI types, such as DDR, consult the Bind Macros (see page 42) and Direct-Connection Interfaces (see page 24) tables.

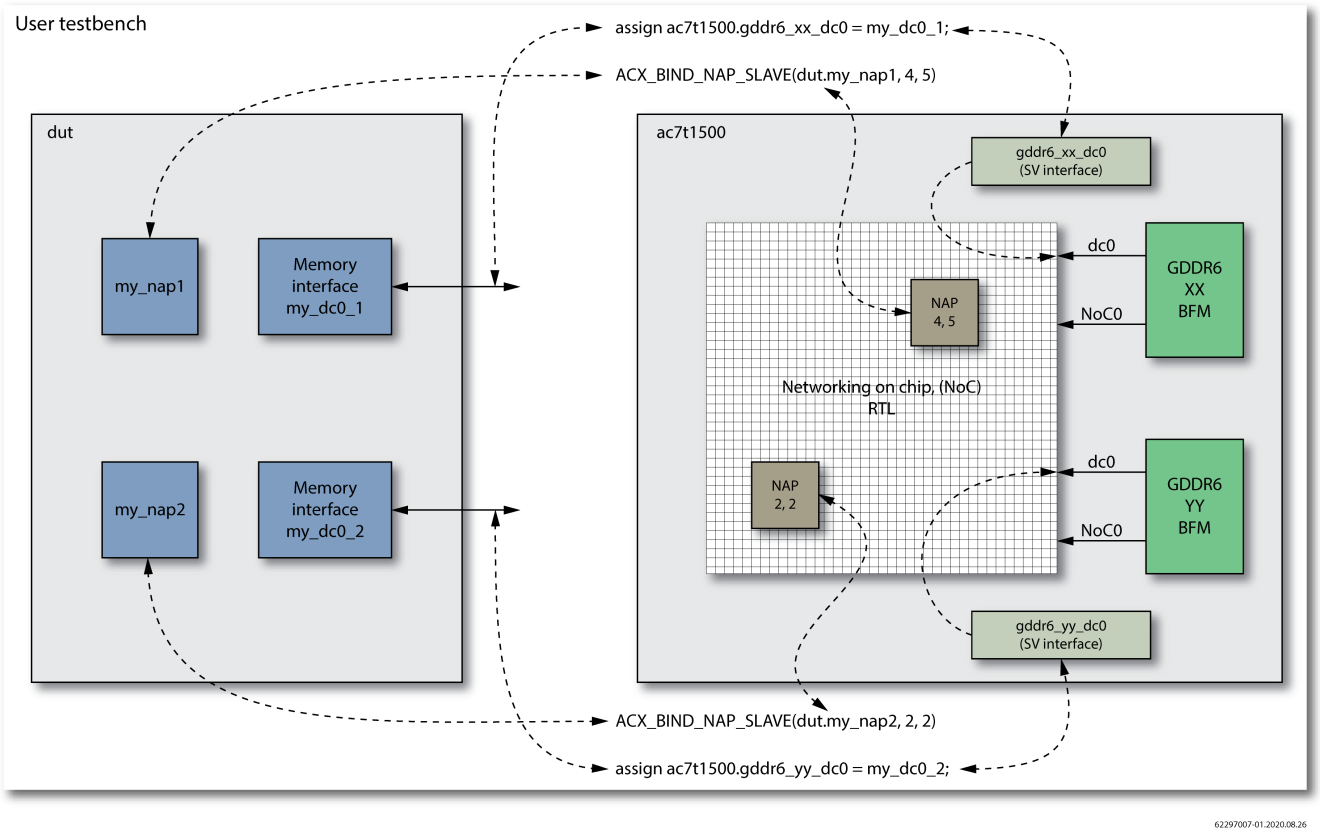


**Figure 3:** ***Example Simulation Structure***

In the example above, there are two NAPs, `my_nap1` and `my_nap2`. In addition there are two direct-connect interfaces, `my_dc0_1` and `my_dc0_2`. In the top-level, testbench bindings are made between the NAPs in the design and the NAPs within the device using the ACX_BIND_NAP_SLAVE macro. This macro supports inserting the coordinates of the NAP within the NoC in order that the simulation is aligned with physical placement of the NAP on silicon.

The DCIs are ports on the user design; these ports are then assigned to the appropriate signals within the device direct-connect SystemVerilog interface.

The Verilog code to instantiate the example is shown below. This example is based on using the ac7t1500 device.

```
// ------------------------------------------------
// Instantiate Speedster7t1500
// ------------------------------------------------
// Connect the chip ready port
// Note : All ac7t1500 ports are defined, so can be directly connected if required
ac7t1500 ac7t1500( .FCU_CONFIG_USER_MODE (chip_ready ) );

// Set the verbosity options on the messages
// Use the inbuilt set_verbosity() task.
initial begin
    ac7t1500.set_verbosity(3);
end
```

```
// ---------------------------------------------
// Bind NAPs
// ---------------------------------------------
// Bind my_nap1 to location 4,5
`ACX_BIND_NAP_AXI_SLAVE(dut.my_nap1,4,5);
// Bind my_nap2 to location 2,2
`ACX_BIND_NAP_AXI_SLAVE(dut.my_nap2,2,2);

// ---------------------------------------------
// Connect to DC interfaces
// ---------------------------------------------
// Create signals to attach to direct-connect interface
logic                          my_dc0_1_clk;
logic                          my_dc0_1_awvalid;
logic                          my_dc0_1_awaddr;
logic                          my_dc0_1_awready;
.....
logic                          my_dc0_2_clk;
logic                          my_dc0_2_awvalid;
logic                          my_dc0_2_awaddr;
logic                          my_dc0_2_awready;
.....

// Connect signals to gddr6_xx_dc0 interface within ac7t1500 device
// Inputs to device
assign ac7t1500.gddr6_xx_dc0.awvalid  = my_dc0_1_awvalid;
assign ac7t1500.gddr6_xx_dc0.awaddr   = my_dc0_1_awaddr;
....
// Outputs from device
assign my_dc0_1_awready = ac7t1500.gddr6_xx_dc0.awready;
....

// Connect signals to gddr6_xx_dc0 interface within ac7t1500 device
// Inputs to device
assign ac7t1500.gddr6_yy_dc0.awvalid  = my_dc0_2_awvalid;
assign ac7t1500.gddr6_yy_dc0.awaddr   = my_dc0_2_awaddr;
....
// Outputs from device
assign my_dc0_2_awready = ac7t1500.gddr6_yy_dc0.awready;
....

// ---------------------------------------------
// Remember to connect the clock!
// ---------------------------------------------
assign my_dc0_1_clk = ac7t1500.gddr6_xx_dc0.clk;
assign my_dc0_2_clk = ac7t1500.gddr6_yy_dc0.clk;
```

**Note**

When using bind macros, the column and row coordinates of the target NAP can be specified. To ensure consistency between simulation and silicon, add matching placement constraints to the ACE placement `.pdc` file, for example:

**In simulation**

```
`ACX_BIND_NAP_AXI_SLAVE(dut.my_nap1,4,5);
```

**In place and route**

```
set_placement -fixed {i:my_nap} {s:x_core.NOC[4][5].logic.noc.nap_s}
```

### set_verbosity( ) Task

Alongside specifying the required simulation package version and instantiating the device, the verbosity of the messages that are output from the device simulation model can be controlled. These levels are controlled by the set_verbosity task. Refer to the code sample above for an example of how to call this function.

The verbosity levels are defined in the following table.

**Table 10: *Verbosity Levels***

| Verbosity Level | Description |
|---|---|
| 0 | Print no messages |
| 1 | Print messages from master and slave interfaces only |
| 2 | Print messages from level 1 and from each NoC data transfer |
| 3 | Print messages from level 2, port bindings and NoC performance statistics |

## Chip Status Output

From initial simulation start, the device operates similarly to its silicon equivalent with an initialization period when the device is in reset. In hardware this occurs during configuration as the bitstream is loaded. After this initialization period, the device asserts the `FCU_CONFIG_USER_MODE` signal to indicate that it has entered user mode, whereby the design starts to operate.

It is suggested that the top-level testbench monitor `FCU_CONFIG_USER_MODE` and delay drive stimulus into the device unitl this signal is asserted (shown in the example above by use of a testbench `chip_ready` signal).

## Bind Macros

The following bind statements are available.

**Table 11: *Bind Macros***

| Macro | Arguments | Description |
|---|---|---|
| ACX_BIND_NAP_HORIZONTAL | user_nap_instance, noc_colunm, noc_row | To bind a horizontal streaming NAP, instance ACX_NAP_HORIZONTAL. |

| Macro | Arguments | Description |
|---|---|---|
| ACX_BIND_NAP_VERTICAL | user_nap_instance, noc_colunm, noc_row | To bind a vertical streaming NAP, instance ACX_NAP_VERTICAL. |
| ACX_BIND_NAP_AXI_MASTER | user_nap_instance, noc_colunm, noc_row | To bind an AXI master NAP, instance ACX_NAP_AXI_MASTER. |
| ACX_BIND_NAP_AXI_SLAVE | user_nap_instance, noc_colunm, noc_row | To bind an AXI slave NAP, instance ACX_NAP_AXI_SLAVE. |

# Direct-Connect Interfaces

Within the device, the non-NAP connections between the high-speed interface subsystems (such as GDDR, DDR, PCIe, Ethernet and SerDes) and the fabric are known as Direct-Connect Interfaces (DCI). These are comprised of:

- Additional data ports in the case of the memory interfaces (AXI)
- Dedicated data interfaces for PCIe (CII) and Serdes (raw mode)
- Status and control for Ethernet

For full details of each subsystem's DCI ports, refer to the appropriate interface subsystem user guide.

Connecting from the user design to the DCI ports involves one of two methods: Connecting directly using the interfaces built into the DSM, or, using an ACE generated port binding file.

## Suggested Flows

In general the direct connection to the DSM ports is used at the commencement of a project, when an ACE project might not yet have been developed. The decision can be made later in the process to use the ACE bindings file. Both methods achieve the same objective; connecting the DUT IO ports to the appropriate locations within the DSM.

- The direct connect method makes use of SystemVerilog interfaces. Therefore, it is possible to add additional features such as protocol checking and performance measurements into these interfaces.
- The ACE port binding method assists with confirming consistency of the DUT ports as presented to ACE (from both the netlist and the ACE generated IP files). This flow can be used to help debug any port naming mismatches prior to committing to place and route.

The two methods are detailed below.

## DSM DC Interfaces

The DSM has a SystemVerilog interface for each DCI port. The available interfaces are listed below.

**Table 12: *DSM Direct-Connect Interfaces***

| Subsystem | Interface Name | Physical Location[(1)] | GDDR6 Channel | SystemVerilog Interface Type | Data Width | Address Width |
|---|---|---|---|---|---|---|
| GDDR6 | gddr6_1_dc0 | West 1 | 0 | t_ACX_AXI4 | 512 | 33 |
| GDDR6 | gddr6_1_dc1 | West 1 | 1 | t_ACX_AXI4 | 512 | 33 |

| Subsystem | Interface Name | Physical Location[1] | GDDR6 Channel | SystemVerilog Interface Type | Data Width | Address Width |
|---|---|---|---|---|---|---|
| GDDR6 | gddr6_2_dc0 | West 2 | 0 | t_ACX_AXI4 | 512 | 33 |
| GDDR6 | gddr6_2_dc1 | West 2 | 1 | t_ACX_AXI4 | 512 | 33 |
| GDDR6 | gddr6_5_dc0 | East 1 | 0 | t_ACX_AXI4 | 512 | 33 |
| GDDR6 | gddr6_5_dc1 | East 1 | 1 | t_ACX_AXI4 | 512 | 33 |
| GDDR6 | gddr6_6_dc0 | East 2 | 0 | t_ACX_AXI4 | 512 | 33 |
| GDDR6 | gddr6_6_dc1 | East 2 | 1 | t_ACX_AXI4 | 512 | 33 |
| DDR | ddr4_dc0 | South | – | t_ACX_AXI4 | 512 | 40 |
| Ethernet | ethernet_0_dc | North West | - | t_ACX_ETHERNET_DCI | - | - |
| Ethernet | ethernet_1_dc | North East | - | t_ACX_ETHERNET_DCI | - | - |
| PCIe | pciex8_dc_cii | North West | - | t_ACX_CII | - | - |
| PCIe | pciex16_dc_cii | North East | - | t_ACX_CII | - | - |
| PCIe | pciex16_dc_master | North East | - | t_ACX_AXI4 | 512 | 42 |
| PCIe | pciex16_dc_slave | North East | - | t_ACX_AXI4 | 512 | 40 |
| Serdes | serdes_eth0_q0_dc | North West | - | t_ACX_SERDES_DCI | 128 | - |
| Serdes | serdes_eth0_q1_dc | North West | - | t_ACX_SERDES_DCI | 128 | - |
| Serdes | serdes_eth1_q0_dc | North East | - | t_ACX_SERDES_DCI | 128 | - |
| Serdes | serdes_eth1_q1_dc | North East | - | t_ACX_SERDES_DCI | 128 | - |

**Note**

1. Physical orientation West to East is with regards to viewing the die in floorplan view within ACE. Note that the die is rotated about its vertical axis when packaged; therefore an interface shown on the floorplan, and listed above, as being on the West is physically on the East side of the device when located on the PCB. The North to South orientation is not affected and matches with the table above, the ACE view, and the device on board.

**Note**

Not all interfaces are available in all devices. Please consult the appropriate device datasheet to understand which interfaces are present in the selected device.

## Direct Connect to DSM Interfaces

To connect to any of these interfaces, create a signal in the testbench, and connect it as a port on the DUT. Also, connect the signal to the DSM, using the DSM instance name, the interface name from the table above, and the element name.

An example of how to connect the awready and awvalid signals for a GDDR AXI interface is given below.

```
// Declare the signals in the testbench
// Note : In order to switch between port binding file and direct connect easily, the signal
//        names must match the DUT IO port names.
logic    dut_awready;
logic    dut_awvalid;


// If the DSM is instantiated with an instance name of ac7t1500, connect to GDDR_1, DC port 0.
// awready is an output from the DSM, and an input to the DUT
assign dut_awready = ac7t1500.interfaces.gddr6_1_dc0.awready;
// awvalid is an input to the DSM, and an output from the DUT
assign ac7t1500.interfaces.gddr6_1_dc0.awready = dut_awvalid;


// Instantiate the DUT
    my_design DUT (
        ......
        .dut_awready    (dut_awready),
        .dut_awvalid    (dut_awvalid),
        ......
    );
```

## Port Binding File to DSM Interfaces

To use the port binding file, configure the following in the testbench:

- Create an ACE project (a netlist is not required at this stage). Configure all interface subsystem IP. Generate the subsystem IP files, including a file named <design_name> `_user_design_port_bindings.svh`.
- Declare the signals in the testbench. The signal names must be the same as the port names on the DUT since these are the names that the port binding file uses.
- Include the port binding file in the testbench.
- Instruct the DSM to set all its DC Interfaces to be in monitor mode only. The latter is important because without this, the DSM drives the ports from the fabric to the subsystems, in addition to the DUT driving the same ports via the binding file. This situation can lead to unresolved signals and simulation failure. The DSM DC interfaces are set to monitor mode when the define `ACX_DSM_INTERFACES_TO_MONITOR_MODE` is enabled.

**Note**

In the Achronix reference design flow the generated subsystem IP files are saved to the `/src/ioring` directory rather than the default `/src/ace/ioring_design` directory.

The define `ACX_DSM_INTERFACES_TO_MONITOR_MODE` must be included in the simulation command line, so that it is present when the DSM is compiled. It cannot be included in the user testbench as this is compiled *after* the DSM. In the provided Achronix reference design flow, `ACX_DSM_INTERFACES_TO_MONITOR_MODE` is defined in the `/sim/<simulator>/system_files_bfm.f` and `/sim/<simulator>/system_files_rtl.f` files.

An example of how to connect all of the DUT ports using the port binding file is shown below.

**system_files_bfm.f**

```
# ----------------------------------------------------------------------
# Description : ac7t1500 full-chip BFM simulation filelist
# ----------------------------------------------------------------------
# Set whether the DSM DCI interfaces are set to monitor mode only
+define+ACX_DSM_INTERFACES_TO_MONITOR_MODE

# ACE libraries must be defined first as they are referenced
# by the fullchip files that follow
+incdir+$ACE_INSTALL_DIR/libraries/
$ACE_INSTALL_DIR/libraries/device_models/7t_simmodels.v

# Fullchip include filelist
# This must be placed before the user filelist as it defines
# the binding macros and utilities used by the user testbench.
$ACX_DEVICE_INSTALL_DIR/sim/ac7t1500_include_bfm.v   <---- DSM compiled here before user design
or testbench
```

**Testbench**

```
// In the testbench
// Declare ALL the DUT signals
logic dut_awready, dut_awvalid ..... ;

// Include the port binding file
`include "../../src/ioring/my_design_user_design_port_bindings.svh"

// Instantiate the DUT
    my_design DUT (
        ......
        .dut_awready    (dut_awready),
        .dut_awvalid    (dut_awvalid),
        ......
    );
```

### Dual Mode Connections to DSM Interfaces

Because there is a define required for the port binding method, this define can be used within the testbench to toggle between the two connection methods. This allows support for both flows, and switching between them simply by enabling or disabling the define. An example of a testbench which supports both methods is shown below.

```
// Declare the signals in the testbench
// Note : In order to switch between port binding file and direct connect easily, the signal
//        names must match the DUT IO port names.
logic    dut_awready;
logic    dut_awvalid;


// The options below support connect to the DSM DC ports either by using the ACE generated
// port binding file, or else using the DSM DC Interfaces.
`ifdef ACX_DSM_INTERFACES_TO_MONITOR_MODE
    `include "../../src/ioring/my_design_user_design_port_bindings.svh"
`else
    assign dut_awready = ac7t1500.interfaces.gddr6_1_dc0.awready;
    assign ac7t1500.interfaces.gddr6_1_dc0.awready = dut_awvalid;
`endif


// Instantiate the DUT
    my_design DUT (
        ......
        .dut_awready    (dut_awready),
        .dut_awvalid    (dut_awvalid),
        ......
    );
```

## Clock Frequencies

In addition to binding to the interfaces, it is possible to control the frequencies of the clocks generated by these interfaces. For design integrity, the clock frequencies set within simulation should match the desired design operating frequencies. For design implementation, the frequencies are configured within the ACE IO Designer tool. For simulation, the `set_clock_period` function is provided.

The example below shows setting the GDDR6 East 1 controller to an operating frequency of 1 GHz (suitable for 16 Gbps operation). Because the DC interface operates at half the controller frequency, it is configured for 500 MHz.

Using this method, first ensure that the simulation operates at the correct frequencies. Second, ensure that each subsystem is able to operate at a different frequency, if required.

```
// Set default GDDR6 clock frequency to 1000 ps = 1GHz
localparam GDDR6_CONTROLLER_CLOCK_PERIOD = 1000;

// Configure the NoC interface of GDDR6 E1 to 1GHz
ac7t1500.clocks.set_clock_period("gddr6_5_noc0_clk", GDDR6_CONTROLLER_CLOCK_PERIOD);

// Configure the DC interface of GDDR6 E1 to 500MHz, (double the period of the NoC interface)
ac7t1500.clocks.set_clock_period("gddr6_5_dc0_clk", GDDR6_CONTROLLER_CLOCK_PERIOD*2);
```

**Note**

The `set_clock_period` function is within the ac7t1500 model. This model has a default timescale value of 1 ps; therefore, the specified clock period is applied in picoseconds, irrespective of the timescale value of the calling module.

The following clock frequency interfaces are available.

**Table 13: *Clock Frequency Interfaces***

| Subsystem | Interface Name | Physical Location (1) | GDDR6 Channel |
|---|---|---|---|
| GDDR6 | gddr6_0_noc0_clk | West 0 NoC | 0 |
| GDDR6 | gddr6_0_noc1_clk | West 0 NoC | 1 |
| GDDR6 | gddr6_1_noc0_clk | West 1 NoC | 0 |
| GDDR6 | gddr6_1_noc1_clk | West 1 NoC | 1 |
| GDDR6 | gddr6_2_noc0_clk | West 2 NoC | 0 |
| GDDR6 | gddr6_2_noc1_clk | West 2 NoC | 1 |
| GDDR6 | gddr6_3_noc0_clk | West 3 NoC | 0 |
| GDDR6 | gddr6_3_noc1_clk | West 3 NoC | 1 |
| GDDR6 | gddr6_4_noc0_clk | East 0 NoC | 0 |
| GDDR6 | gddr6_4_noc1_clk | East 0 NoC | 1 |
| GDDR6 | gddr6_5_noc0_clk | East 1 NoC | 0 |
| GDDR6 | gddr6_5_noc1_clk | East 1 NoC | 1 |
| GDDR6 | gddr6_6_noc0_clk | East 2 NoC | 0 |
| GDDR6 | gddr6_6_noc1_clk | East 2 NoC | 1 |
| GDDR6 | gddr6_7_noc0_clk | East 3 NoC | 0 |

| Subsystem | Interface Name | Physical Location [1] | GDDR6 Channel |
|---|---|---|---|
| GDDR6 | gddr6_7_noc1_clk | East 3 NoC | 1 |
| GDDR6 | gddr6_1_dc0_clk | West 1 DCI | 0 |
| GDDR6 | gddr6_1_dc1_clk | West 1 DCI | 1 |
| GDDR6 | gddr6_2_dc0_clk | West 2 DCI | 0 |
| GDDR6 | gddr6_2_dc1_clk | West 2 DCI | 1 |
| GDDR6 | gddr6_5_dc0_clk | East 1 DCI | 0 |
| GDDR6 | gddr6_5_dc1_clk | East 1 DCI | 1 |
| GDDR6 | gddr6_6_dc0_clk | East 2 DCI | 0 |
| GDDR6 | gddr6_6_dc1_clk | East 2 DCI | 1 |
| DDR | ddr4_noc0_clk | South NoC | – |
| DDR | ddr4_dc0_clk | South DCI | – |
| PCIe | pciex16_clk | Gen5 PCIe ×16 | – |
| PCIe | pciex16_dc_clk | Gen5 PCIe ×16 DCI | – |
| PCIe | pciex8_clk | Gen5 PCIe ×8 | – |
| Ethernet | ethernet_ref_clk | Ethernet reference clock [2] | - |
| Ethernet | ethernet_ff0_clk | Ethernet FIFO 0 clock [2] | - |
| Ethernet | ethernet_ff1_clk | Ethernet FIFO 1 clock [2] | - |
| Configuration | cfg_clk | System wide configuration clock | – |

**Note**

1. Physical orientation West to East is with regards to viewing the die in floorplan view within ACE. Note that the die is rotated about its vertical axis when packaged. Therefore, an interface shown on the floorplan, and listed above, as being on the West is physically on the East side of the device when located on the PCB. The North to South orientation is not affected and matches with the table above, the ACE view, and the device on board.
2. The Ethernet clocks are common to both Ethernet subsystems. In simulation they must be set to operate from the same clock frequencies.

# Configuration

A number of the interface subsystems require configuration at power-up. In the physical device, this configuration would be performed by the bitstream pre-programming the relevant configuration registers. Within the simulation environment, there are tasks that can read configuration files and apply those files to the relevant interface subsystem. An example of applying a configuration is shown in the code snippet below.

```
// -------------------------
// Configuration
// -------------------------

// Call function within device to configure the registers
// By using fork-join, the two configurations will be run in parallel, configuring both
// Ethernet blocks.  This saves overall simulation time.
// Both blocks are configured the same, hence the use the same file
initial
begin
    fork
        ac7t1500.fcu.configure( "ethernet_cfg.txt", "ethernet0" );
        ac7t1500.fcu.configure( "ethernet_cfg.txt", "ethernet1" );
    join
end
```

## Startup Sequence

While the task `fcu.configure()` is processing the configuration (including waiting for any polling to return a valid value), the Chip Status Output (see page 24) is not asserted. This behavior mirrors that where the device only enters user mode when configuration is completed.

The simulation testbench can issue configuration processes as shown above, and when the Chip Status Output (see page 24) is asserted, the testbench knows the device is correctly configured. The testbench can then proceed to apply the necessary tests.

## fcu.configure( ) Task

The task `fcu.configure` has the following arguments:

```
fcu.configure ( <configuration filename>, <interface subsystem name> );
```

The following interface subsystem names are supported:

**Table 14: *Configuration Subsystem Names***

| Subsystem | Interface Subsystem Name (1) | Physical Location (3) |
|---|---|---|
| GDDR6 | gddr6_0 | West 0 |
| GDDR6 | gddr6_1 | West 1 |
| GDDR6 | gddr6_2 | West 2 |

| Subsystem | Interface Subsystem Name [1] | Physical Location [3] |
|---|---|---|
| GDDR6 | gddr6_3 | West 3 |
| GDDR6 | gddr6_4 | East 0 |
| GDDR6 | gddr6_5 | East 1 |
| GDDR6 | gddr6_6 | East 2 |
| GDDR6 | gddr6_7 | East 3 |
| DDR | ddr4 | South |
| Ethernet | ethernet0 | North |
| Ethernet | ethernet1 | North |
| GPIO North | gpio_n | North |
| GPIO South | gpio_s | South |
| PCIe ×8 | pcie_0 | North |
| PCIe ×16 | pcie_1 | North |
| All subsystems | full [2] | – |

**Table Notes**

1. The interface subsystem name is case insensitive.
2. When using the `full` subsystem name, the full 42-bit address is required in the configuration file. When selecting an individual subsystem, only the 28-bit address is required. See Configuration File Format (see page 33) below for details.
3. Physical orientation West to East is with regards to viewing the die in floorplan view within ACE. Note that the die is rotated about its vertical axis when packaged; therefore an interface shown on the floorplan, and listed above, as being on the West is physically on the East side of the device when located on the PCB. The North to South orientation is not affected and matches with the table above, the ACE view, and the device on board.

## Configuration File Format

The configuration file has the following format:

```
# ----------------------------------------
# Configuration file
# Supports both # and // comments
# ----------------------------------------

# A comment line
// Another comment line

# Format is <cmd> <addr> <data>

# Commands are
 "w" - write
 "r" - read
 "v" - read and verify
 "d" - Wait for the number of cycles in the data field.
       The address field is unused

# Address is either 28-bit, (7 hex characters), or 42-bit, (11 hex characters).
# 28-bits supports the configuration memory space of an single interface subsystem
# 42-bits supports the full configuration memory space

# Data is 32-bit, (8 hex characters).

# For reads, put 0x0 for the data
# For verify put the expected data value

# Examples

# Writes
w 00005c0 76543210
w 0000014 00004064

# Reads
r 00005c0 00000000
r 0000014 00000000

# Verify
v 00005c0 76543210
v 0000014 00004064


# Wait for 50 cycles
d 0000000 00000032
```

## Address Width

The address width varies according to the requirements of the file:

- When addressing an individual subsystem, only the lower 28 bits of the address field are used. The higher 14 bits are derived from the subsystem name.

- When addressing the full configuration memory space (interface subsystem name is set to `full`), then 42 bits of the address space are required. In this mode, the FCU confirms that bits [41:34] of the address field are set to 8'h20, which aligns with the NoC global memory map plus control and status register (CSR) memory area. In this mode, the one configuration file can address multiple interface subsystems. See the *Speedster7t Network on Chip User Guide* (UG089) for more details.

### Parallel Configuration

The `fcu.configure()` task is defined as a SystemVerilog automatic task allowing it to be re-entrant and run in parallel. Therefore, it is possible to program multiple interface subsystems in parallel using a `fork - join` construct. Refer to the reference design testbench for examples of this parallel programming.

## SystemVerilog Interfaces

The following SystemVerilog interfaces are defined, and are used for DCI assignments.

> **Note**
>
> The interface below is only available in the simulation environment. For code that must be synthesized, define custom SystemVerilog interfaces, or use one of the interfaces predefined within the reference designs.

```
interface t_ACX_AXI4
    #(DATA_WIDTH = 0,
      ADDR_WIDTH = 0,
      LEN_WIDTH  = 0);

   logic                       clk;       // Clock reference
   logic                       awvalid;   // AXI Interface
   logic                       awready;
   logic [ADDR_WIDTH -1:0]     awaddr;
   logic [LEN_WIDTH -1:0]      awlen;
   logic [8 -1:0]              awid;
   logic [4 -1:0]              awqos;
   logic [2 -1:0]              awburst;
   logic                       awlock;
   logic [3 -1:0]              awsize;
   logic [3 -1:0]              awregion;
   logic [3:0]                 awcache;
   logic [2:0]                 awprot;
   logic                       wvalid;
   logic                       wready;
   logic [DATA_WIDTH -1:0]     wdata;
   logic [(DATA_WIDTH/8) -1:0] wstrb;
   logic                       wlast;
   logic                       arready;
   logic [DATA_WIDTH -1:0]     rdata;
   logic                       rlast;
   logic [2 -1:0]              rresp;
   logic                       rvalid;
```

```
   logic [8 -1:0]              rid;
   logic [ADDR_WIDTH -1:0]     araddr;
   logic [LEN_WIDTH -1:0]      arlen;
   logic [8 -1:0]              arid;
   logic [4 -1:0]              arqos;
   logic [2 -1:0]              arburst;
   logic                       arlock;
   logic [3 -1:0]              arsize;
   logic                       arvalid;
   logic [3 -1:0]              arregion;
   logic [3:0]                 arcache;
   logic [2:0]                 arprot;
   logic                       aresetn;
   logic                       rready;
   logic                       bvalid;
   logic                       bready;
   logic [2 -1:0]              bresp;
   logic [8 -1:0]              bid;

   modport master (input  awready, bresp, bvalid, bid, wready, arready, rdata, rlast, rresp,
rvalid, rid,
                   output awaddr, awlen, awid, awqos, awburst, awlock, awsize, awvalid, awregion,
                          bready, wdata, wlast, rready, wstrb, wvalid,
                          araddr, arlen, arid, arqos, arburst, arlock, arsize, arvalid, arregion);

   modport slave  (output awready, bresp, bvalid, bid, wready, arready, rdata, rlast, rresp,
rvalid, rid,
                   input  awaddr, awlen, awid, awqos, awburst, awlock, awsize, awvalid, awregion,
                          bready, wdata, wlast, rready, wstrb, wvalid,
                          araddr, arlen, arid, arqos, arburst, arlock, arsize, arvalid, arregion);

endinterface : t_ACX_AXI4
```

# Installation

## Packages

### Base versus RTL

Due to licensing conditions, there are two different DSM packages available

- **Base**: This package contains the full cycle accurate simulation model of the device core (including NoC, NAPs, Ethernet and GPIO subsystems). Further, the package contains BFM simulation models of the GDDR, PCIe and DDR4 interface subsystems. These BFM models support representative timings of their respective subsystems, but deliver significantly faster simulation times. Any data transfers from these interface subsystems to the NoC use the cycle accurate NoC, further enhancing the timing accuracy.

- **RTL**: This package adds the full cycle accurate models of the PCIe, GDDR and DDR4 cores. Using these models provides full cycle accuracy of the complete systems, increasing simulation times significantly, however. This package is license controlled. To obtain this package, send a support request to support@achronix.com requesting the DSM RTL package. When licensing conditions are met, a link to download the individual watermarked version of the package is sent.

### Naming

There is a DSM package per device, available for both Linux and Windows. The base packages are named `ACE_<DSM version>_<device>_DSM_base_Sim_overlay.tgz` for Linux and `ACE_<DSM version>_<device>_base_Sim_overlay.zip` for Windows. For example:

- `ACE_8.3.3_ac7t1500_DSM_base_sim_overlay.tgz` : This is the base DSM overlay for the ac7t1500 device, targeted for Linux. The DSM, (not ACE), version is 8.3.3.

Similarly, the RTL packages are named `ACE_<DSM version>_<device>_DSM_RTL_Sim_overlay.tgz` and `ACE_<DSM version>_<device>_DSM_RTL_Sim_overlay.tgz.`

> **Note**
>
> The version number in the DSM package is the DSM version, not the ACE version. There is not necessarily a new DSM release per ACE release. Therefore, it is possible to use an older DSM release with a newer ACE release. For example, DSM 8.2.1 may be used with ACE 8.3.2.

### Download

The base packages are available on the Achronix self-service FTP site at secure.achronix.com, located in the `/Achronix/ACE/Speedster7t` folder. As each device package is independent, it is possible to either download only the device the user is targeting, or all device packages.

Due to licensing conditions, see above (see page ), RTL packages are obtained by sending a support request to support@achronix.com.

Any package is only required to be installed once — it is common for all designs targeting the selected device.

## ACE Integration

### Upgrading an Existing Installation

If a version of the DSM package was previously installed into ACE, it is recommended to first delete the existing DSM package before upgrading to ensure the integrity of the new installation. To delete an existing package, navigate to `<ACE_INSTALL_DIR>/system/data/yuma-alpha-rev0` and remove the `/sim` directory. Then return to the root of the ACE installation and proceed with the instructions for First Installation (see page 37) below.

### First Installation

The recommended installation method is to merge the contents of the package into the current ACE installation. The package contains a root directory `/system`. The contents of this folder should be merged with the selected ACE installation `/system` folder.

**Warning**

The contents of the simulation package consist of files that are not present in the base ACE installation. These files should not replace or overwrite any existing files. However, if an earlier version of the simulation package has already been downloaded, then select "overwrite" to ensure the latest version of the simulation files are written to the ACE installation.

### Standalone

In certain instances it might not be possible to modify an existing ACE installation. In these cases, it is possible to install the package separately and to simulate using files from both this simulation package and the existing simulation files within ACE.

To install as standalone, simply uncompress the package to a suitable location.

**Note**

All reference designs are configured for the simulation package to be integrated within ACE. If the standalone method is selected, the necessary environment variables in the reference design makefiles must be edited.

## Environment Variables

The locations of both ACE and the simulation package are controlled by two environment variables. For all reference designs, these two variables must be set before simulating.

### ACE_INSTALL_DIR

The environment variable `ACE_INSTALL_DIR` must be set to the directory location of the `ace`, or `ace.exe` executable. This variable is used by both simulation and synthesis to locate the correct device library files.

### ACX_DEVICE_INSTALL_DIR

The optional environment variable `ACX_DEVICE_INSTALL_DIR` is used to select the DSM files. It should be set to the base directory of the device files within the DSM package. For example, if the ac7t1500 device is selected, then the device base directory is `yuma-alpha-rev0`.

When the installed in ACE integration mode, the following setting should be used:

```
ACX_DEVICE_INSTALL_DIR = $ACE_INSTALL_DIR/system/data/yuma-alpha-rev0
```

When installed as standalone, the following setting should be used:

```
ACX_DEVICE_INSTALL_DIR = <location of standalone package>/system/data/yuma-alpha-rev0
```

**Note**

For simulation, it is only necessary to set the ACX_DEVICE_INSTALL_DIR variable if the DSM is not installed in ACE integration mode. In all the supplied designs, the simulation makefiles define ACX_DEVICE_INSTALL_DIR as shown for ACE integration mode. This definition takes precedence over any local environment variable. If using a supplied simulation makefile, override the definition of ACX_DEVICE_INSTALL_DIR in the make flow invocation as follows:

```
> make ACX_DEVICE_INSTALL_DIR=<location of standalone package>/system/data
/yuma-alpha-rev0
```

# Simulating the Reference Design

## Supported Simulators

The designs have a consistent simulation environment, providing scripts for Mentor QuestaSim and Synopsys VCS simulators.

## Location

All designs have a /sim directory located in the design root directory. Within this directory there are /vcs and /questa directories for each of the simulators.

## Simulation Flows

Where applicable the simulation supports a number of flow options, which offer a balance between speed and accuracy. Not all flow options are available with all reference designs; the relevant makefiles will list what flow options can be set.

### Standalone

Any NAP in the design uses a standalone model bound to the NAP, modelling memory behavior. This mode is the quickest simulation to run, but is the least cycle accurate. The NAP will only interact with its own memory model; therefore, this mode does not support multiple NAPs designed to access a common memory. To enable this mode in the supplied reference design, set the FLOW variable in the simulation makefile to STANDALONE

### Full-Chip BFM

This uses a model of the full chip, with cycle-accurate NoC. There are then bus functional models (BFMs) for all the hardened interfaces around the NoC. These BFMs have representative delays, allowing this mode to offer near cycle-accurate simulations. This mode does not require the interface subsystems to perform initialization and calibration steps, offering a quicker iterative time compared to a full cycle-accurate simulation.

The full-chip BFM simulations require the I/O Ring Simulation Package to have been downloaded and installed. To enable this mode in the supplied reference design, set the FLOW variable in the simulation makefile to FULLCHIP_BFM.

### Full-Chip RTL

This mode uses the full RTL of the subsystem combined. if necessary, with a cycle-accurate model of any necessary external component (such as a memory). This configuration gives a fully cycle-accurate simulation representing the final silicon operation. For most of these simulations it will be necessary to configure the relevant subsystems using the provided configuration files. As these simulations are using the full RTL of the subsystem, they run slower than the BFM equivalent simulations, while offering complete timing accuracy.

> **Note**
>
> To obtain the encrypted RTL of the GDDR/DDR or PCIe subsystems, a second licensed simulation package is required. Please contact Achronix Support to arrange licensing and access to this package.

To enable this mode in the supplied reference design, set the `FLOW` variable in the simulation makefile to `FULLCHIP_RTL`.

## Build Options

Within each simulator directory is a makefile. This makefile can be edited by the user to configure the simulation to their needs. The following variables need to be set:

- `FLOW` – To match the selected simulation flow. The options (detailed in the makefile) are `STANDALONE, FULLCHIP_BFM or FULLCHIP_RTL.`
- `TOP_LEVEL_MODULE` – Preset for the supplied design. However, if the user ports the design to their own testbench, this variable must be updated.
- `ACX_DEVICE_INSTALL_DIR` – Points to the directory (normally under ACE) where the target device files are stored, for example, `v$ACE_INSTALL_DIR/system/data/yuma-alpha-rev0`. For further information consult the I/O ring simulation installation instructions.

## Prerequisites

Before running any installation, the user must ensure the following are configured:

- `ACE_INSTALL_DIR` environment variable: This should point to the ACE installation directory, where the ACE executable is located.
- Path to the required simulator. For VCS this should also include the VCS_HOME environment variable.

## Auto File List Generation

The simulation file list is auto generated from the `../../src/filelist.tcl` file. The script to create the simulation file list is `../../scripts/create_sim_project.tcl`, and it uses a template file `../scripts/sim_template.f or ../../scripts/sim_template_bfm.f` to define the general simulation options. The create script is called by the specific simulator makefile as detailed below. The resultant file, `sim_filelist.f`, combines the template contents with the specific list of files in `../../src/filelist.tcl`.

## Files

In each `/sim/vendor` directory the following scripts are located:

- `makefile` – Makefile supporting the various simulation flows. Default target is to compile and run the simulation.
- `system_files_bfm.f` – (Full-chip BFM flow only) List of system files used by the full-chip BFM flow. Any user defines can also be added to this file.
- `system_files_rtl.f` – (Full-chip RTL flow only) List of system files used by the full-chip RTL flow. Also any defines required to specify which subsystems should be cycle-accurate RTL rather than BFM. The defines are named as <subsystem name>_FULL, i.e., GDDR6_2_FULL. Any user defines can also be added to this file.

### VCS Only

- `fullchip_bfm_vcs_waiver.cfg` – (Full-chip BFM flow only) Waiver file to remove benign warnings.
- `session.sim_output_pluson.vpd.tcl` – Session file for DVE waveform viewer.

### QuestaSim Only

- `wave.do` – Waveform file.
- `qsim_<design_name>.do` – Non-makefile GUI flow. OS independent.

## RTL Simulation Defines

In order to compile and run the Full-Chip RTL flow, the user will need to enable SystemVerilog defines for the simulation compilation. When the simulation makefile `FLOW` variable is set to `FULLCHIP_RTL`, the makefile will include the `system_files_rtl.f` file in the compilation command line.

Any required defines to toggle simulation blocks from BFM model to full RTL are enabled within the `system_files_rtl.f` file. For example, if using the GDDR6 reference design, and the user decides they need to simulate GDDR6 memory controller 1 with the full RTL, but leave all other GDDR6 controllers using the BFM model, they need to edit the `system_files_rtl.f` file as shown below:

```
# Define GDDR Data Rate
+define+GDDR6_DATA_RATE_16     # <----- For GDDR6 RTL simulation the user must
                                       enable one of the data rates.
//+define+GDDR6_DATA_RATE_14
//+define+GDDR6_DATA_RATE_12

# Turn on GDDR RTL
# Enable the desired GDDR memory controllers below
# Any undefined controllers will use their BFM model
//+define+ACX_GDDR6_0_FULL
+define+ACX_GDDR6_1_FULL        # <--- User enables this define
//+define+ACX_GDDR6_2_FULL
//+define+ACX_GDDR6_3_FULL
//+define+ACX_GDDR6_4_FULL
//+define+ACX_GDDR6_5_FULL
//+define+ACX_GDDR6_6_FULL
//+define+ACX_GDDR6_7_FULL

<-------- User does not need to change any options below here  ---->

# ACE libraries must be defined first as they are referenced
# by the fullchip files that follow
+incdir+$ACE_INSTALL_DIR/libraries/
$ACE_INSTALL_DIR/libraries/device_models/7t_simmodels.v

# Fullchip include filelist
# This must be placed before the user filelist as it defines
# the binding macros and utilities used by the user testbench.
$ACX_DEVICE_INSTALL_DIR/sim/ac7t1500_include_bfm.v
```

The table of available defines to enable each module to use full RTL rather than BFM model is listed in the table below (see page 42)

**Table 15: *Simulation RTL defines***

| Module | Define |
|---|---|
| GDDR6 controller 0 | ACX_GDDR6_0_FULL |
| GDDR6 controller 1 | ACX_GDDR6_1_FULL |
| GDDR6 controller 2 | ACX_GDDR6_2_FULL |
| GDDR6 controller 3 | ACX_GDDR6_3_FULL |
| GDDR6 controller 4 | ACX_GDDR6_4_FULL |
| GDDR6 controller 5 | ACX_GDDR6_5_FULL |
| GDDR6 controller 6 | ACX_GDDR6_6_FULL |

| Module | Define |
|---|---|
| GDDR6 controller 7 | ACX_GDDR6_7_FULL |
| DDR4 controller | ACX_DDR4_FULL |
| Ethernet subsystems (both) | ACX_ETHERNET_FULL |
| PCIe Controller 0 (×8) | ACX_PCIE_0_FULL |
| PCIe controller 1 (×16) | ACX_PCIE_1_FULL |
| GPIO North block | ACX_GPIO_N_FULL |
| GPIO South block | ACX_GPIO_S_FULL |
| All SerDes lanes | ACX_SERDES_FULL |

**Table Note**

Locations and names of each of the interface subsystems above is visible using the ACE IP Configuration perspective, and selecting the I/O layout diagram

**Warning!**

Enabling full RTL simulation for modules will increase simulation time. If several modules are enabled, the simulation time could be extended by significant amounts.

## Running the Simulation

For all simulator flows there is a `makefile` located in the simulation directory. The makefiles support the following build options

### VCS

**code**

```
$ make           : Default flow. Compile with no debug options, run in batch mode writing the
output to a VPD file.
$ make run       : Same as "make".
$ make debug     : Build with debug options (increased visibility of lower level variables).
                   Run in batch mode writing to a VPD file.
$ make open_dve  : Open the DVE waveform viewer and load the VPD file and viewing session file.
$ make clean     : Delete all generated and temporary files.
```

## QuestaSim

```
code

$ make           : Default flow. Compile with no debug options, run in batch mode writing the
output to a WLF file.
$ make run       : Default flow. Same as "make".
$ make debug     : Build with debug options (increased visibility of lower level variables).
                   Open GUI with wave.do and allow user to run interactively.
$ make open_wave : Open the GUI waveform viewer. Load the generated WLF file and viewing wave.do f
ile.
$ make clean     : Delete all generated and temporary files.
```

## QuestaSim Non-Makefile Flow

To support environments that do not natively support makefiles (such as Windows), there is an additional QuestaSim non-makefile flow using QuestaSim `.do` files. The following steps are required to use this flow:

1. Navigate to `sim/questa`
2. Launch QuestaSim GUI. Normally:

   ```
   $ vsim
   ```

   Within the QuestaSim GUI launch the script

   ```
   $ do qsim_<design_name>.do
   ```

> **Note**
>
> The QuestaSim script uses the ACE_INSTALL_DIR environment variable. For correct operation of this, (or any other), script flow, ACE should be installed in a directory without spaces in the path name, e.g., `C:\achronix\8.2.1\Achronix_CAD_Environment`. Additionally the environment variable `ACE_INSTALL_DIR` must use "/" as path separators rather than "\", for example:
>
> `ACE_INSTALL_DIR = C:/achronix/8.2.1/Achronix_CAD_Environment/Achronix`

> **Note**
>
> The `do` script is configured for the `FULLCHIP_BFM` flow. It can be modified to match the `STANDALONE` flow by selecting the appropriate options commented within the script.

## Results Verification

All the designs make use of a self-checking testbench which compares the results generated from the RTL to a verified output. The verified output can come from a number of sources, either a math package, a software model, or an RTL behavioral model. The details of the applicable verification source is given in the detail of each individual design.

# Building the Reference Design

The designs make use of a consistent build environment, using a makefile and scripts to run both Synplify Pro and ACE in batch mode.

## Prerequisites

Before running any installation, the user must ensure the following are configured:

- `ACE_INSTALL_DIR` environment variable. This variable must point to the ACE installation directory where the ACE executable is located.
- ACE should be in the environment path. The easiest method is to add `$ACE_INSTALL_DIR` to the path variable.
- Synplify Pro should be in the environment path.

## Batch Flow

In the root directory, there is a `/build` directory, within which there is a `makefile`. Before running the makefile, ensure the prerequisites above have been met. The relative paths within the makefile are intended to be run from the `/build` directory. If the makefile is moved to a new location or called outside of this directory, then the paths will require amending accordingly.

When the makefile is run (with the default options), it will create the following;

- `/build/results/syn` directory – Synplify Pro is executed in batch mode, synthesizing the design, which generates a netlist in `/results/syn/rev_1/<design_name.vm`. If synthesis is unsuccessful, the user should consult `/results/syn/rev_1/<design_name>.srr` for details of any synthesis failure. Options to the generated Synplify Pro project file are controlled by `/src/constraints /synplify_options.tcl.`
- `/build/results/ace` directory – After synthesis, ACE is run in evaluation mode (meaning that no I/O pins need be specified). If any options are required for the ACE-generated project, these are controlled by the `/src/constraints/ace_impl_options.tcl` file.

### Makefile Options

The makefile supports multiple build flow options

**code**

```
$ make            : Default flow. Synthesize and build a single implementation with ACE
$ make run      : Same as "make"
$ make syn_only    : Synthesis only
$ make pnr_only : Run ACE place and route only. This requires synthesis to have previously been
run.
$ make run_mp   : Run multiprocess. Synthesize and build multiple implementations with ACE
multiprocess.
$ make clean      : Delete all generated and temporary files
```

### Constraint Files

In addition to the constraint files listed above, additional files may be used in any build flow. All constraint files are located in `/src/constraints`:

- `ace_timing.sdc, <design_name>.sdc` – timing constraint files used by both synthesis and ACE.
- `<design_name>.fdc` – FPGA constraints used by Synplify Pro to set non-timing related directives and attributes, such as compile points.
- `<design_name>.pdc` – placement constraints used by ACE.

The full list of what files are used in the flow, and by which tool, can be determined by referring to the relevant `/src/filelist_xx.tcl` file.

## GUI Flow

The design has pre-generated GUI project files for both ACE and Synplify Pro. These files are located in `/src/ace` and `/src/syn` directories respectively. The user can open these to interactively edit or run builds.

> **Note**
>
> When using the GUI projects, any generated files will be placed beneath the GUI project file directory.
>
> - For Synplify Pro, the revision directory, rev_1 etc. will be generated in `/src/syn/rev_1.`
> - For ACE, any implementation directory will be generated as `/src/ace/impl_<name>.`
> - For ACE, if I/O Designer is used to generate new constraint files, the default directory for those files is `/src/ace/ioring_design.`

When builds are done using the batch flow, the flow writes out both the relevant project files under the `/build/results` directory. Within this directory are the generated project files for both ACE and Synplify Pro. The user can open both of these project files in GUI mode and interactively re-run or edit the builds.

# Revision History

| Version | Date | Description |
|---|---|---|
| 1.0 | 08 May 2020 | • Initial Release. |
| 2.0 | 22 Jul 2020 | • Updates to simplify PCIe master tasks.<br>• Updated signal/port names and clock description to match design changes due to ACE 8.2 I/O Designer change. |
| 3.0 | 22 Feb 2021 | • Added RTL simulation mode related content: testbench, tasks list and how to control simulation modes.<br>• Added section, Obtaining and Installing Synopsys PCIe VIP (RTL Mode Only) (see page 13). |
| 3.1 | 29 Apr 2021 | • Added how to obtain PCIe RTL simulation model<br>• Separated simulation and core design introductions<br>• Added memory model RTL details<br>• Raised ACE and DSM required versions to 8.3.3<br>• Added Info. section for PCIe acxip files |
| 3.1.1 | 18 May 2021 | • Revised to reflect changes in simulation procedure |


Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com



## Notice of Disclaimer