Design Flow User Guide (UG106)

All Achronix Devices

Achronix
Data Acceleration

Copyrights, Trademarks and Disclaimers

Notice of Disclaimer

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

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

Table of Contents

Chapter 1 : Introduction

This user guide provides details on the Achronix design flow and toolchain. It covers transforming a user design from RTL to a bit file, explains the generation of hard IP, and demonstrates the effective mapping of these components to each other and to the device inputs and outputs. In-depth information is provided about design constraints and simulation for the input/output ring (I/O ring) IP. Additionally, the guide addresses common warnings and errors, offering insights into their potential causes and solutions. Finally, pin mapping is covered to show the relationship between the different device orientations displayed by ACE.

Chapter 2 : Overview of Interface Subsystems

The I/O ring surrounds the core of the device, and is composed of various interface subsystems that represent high-speed data interfaces along with general-purpose I/O (GPIO), clocks and resets. Since the I/O ring surrounds the core, it is the only way that signals can enter and exit the device. As such, the subsystem associated with the signal (e.g., cock I/O bank for clock signals, GPIO bank for GPIO data, etc.) must be used in order to route signals into and out of the core. The following figure provides a visual representation of the interface subsystems surrounding the core in the Speedster® AC7t1500 FPGA.

Note

Placement and number of subsystems vary for other devices in the family.

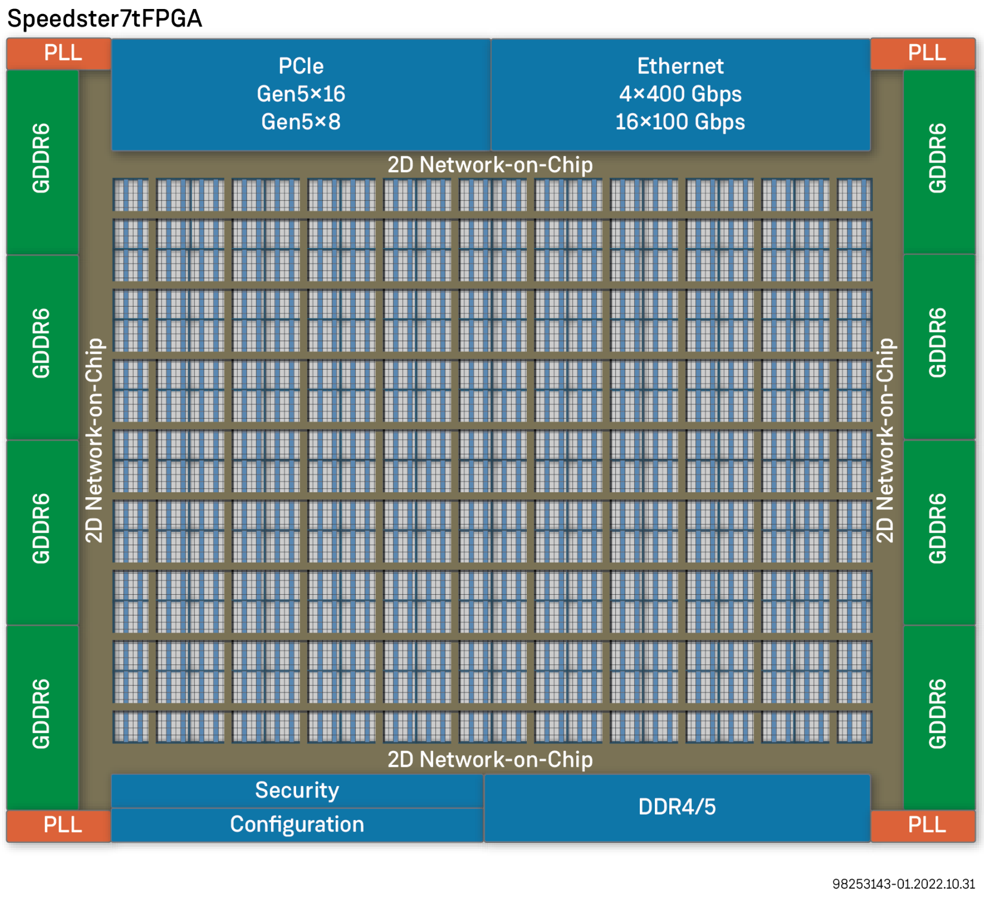


Figure 1 • Speedster AC7t1500 FPGA Top-Level Block Diagram

Note

The I/O ring does not exist in Speedcore™ eFPGAs.

Clock I/O Banks, PLLs, and Advanced PLL Subsystems

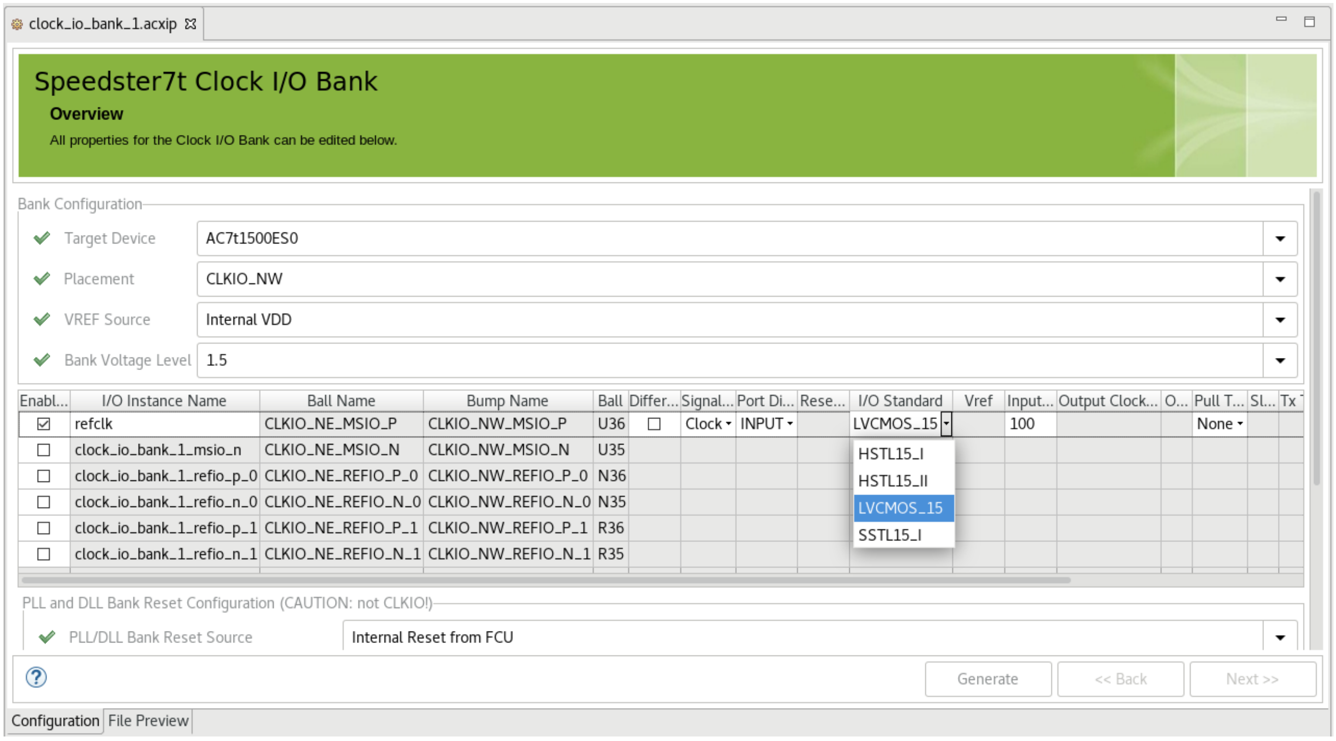
In most cases, the first step in configuring the I/O ring is to add a clock I/O bank and a PLL or an advanced PLL. Clock I/O Banks allow defining the clock I/O which can then be used as the reference clock to a PLL, an advanced PLL, or bypassed through an advanced PLL. Devices in the Speedster7t family have up to 16 PLLs and advanced PLLs that can be used for clock generation and are located in the corners of the device. Some features of these PLLs are:

- PLLs are fractional-N divide and spread-spectrum.
- PLLs can be used to drive low-skew, high-speed clocks to nearby I/O, the global clock network, and interface clocks in the FPGA core.
- Each PLL can generate up to four output clocks with a total of 32 clocks able to route on the global clock network.
- Clock signals can be phase shifted using a DLL.
- In each corner, there is one initiator DLL with eight responders available for clock phase shifting. This arrangement allows for one reference clock and up to eight synchronized clocks that can be phase shifted based on the reference clock frequency.
- I/O signals with `msio` in their name are primarily for single-ended clocks. They are marked as having both p and n sides, and can be used in either differential or single-ended mode.
- I/O signals with `refio` in their name are for differential clocks but do support single-ended clocks incoming on their p-side.

Note

Advanced PLLs are comprised of the standard PLL core but expose more options for customization (e.g., the first divider stage is customizable in the advanced PLL but not in the standard PLL). The hardware itself is shared for both subsystems. This sharing means that both a standard and an advanced PLL cannot exist at the same placement specification (for both PLLs, the placement specification designates the corner and which of the four PLLs in that corner are used). To be clear, both PLLs can co-exist in the same corner but cannot share the same placement specification.

Clocks must enter through a clock I/O Bank. Clock I/O can support multiple I/O standards including LVCMOS 15, HSTL15 I, HSTL15 II, and SSTL15 I. These standards can be set when the clock I/O is configured, as shown in the following example:


clock_io_bank_1.acxip

Speedster7t Clock I/O Bank

Overview

All properties for the Clock I/O Bank can be edited below.

Bank Configuration

Target Device	AC7t1500ES0
Placement	CLKIO_NW
VREF Source	Internal VDD
Bank Voltage Level	1.5

Enabl...	I/O Instance Name	Ball Name	Bump Name	Ball	Differ...	Signal...	Port Di...	Rese...	I/O Standard	Vref	Input...	Output Clock...	O...	Pull T...	Sl...	Tx
☑	refclk	CLKIO_NE_MSIO_P	CLKIO_NW_MSIO_P	U36	☐	Clock	INPUT		LVCMOS_15		100			None		
☐	clock_io_bank_1_msio_n	CLKIO_NE_MSIO_N	CLKIO_NW_MSIO_N	U35												
☐	clock_io_bank_1_refio_p_0	CLKIO_NE_REFIO_P_0	CLKIO_NW_REFIO_P_0	N36												
☐	clock_io_bank_1_refio_n_0	CLKIO_NE_REFIO_N_0	CLKIO_NW_REFIO_N_0	N35												
☐	clock_io_bank_1_refio_p_1	CLKIO_NE_REFIO_P_1	CLKIO_NW_REFIO_P_1	R36												
☐	clock_io_bank_1_refio_n_1	CLKIO_NE_REFIO_N_1	CLKIO_NW_REFIO_N_1	R35												

PLL and DLL Bank Reset Configuration (CAUTION: not CLKIO!)

PLL/DLL Bank Reset Source: Internal Reset from FCU

Figure 2 • Clock I/O Standards

Note

Other I/O subsystems and user RTL designs reference PLL and Advanced PLL outputs by their I/O instance name.

The reference clock for a PLL can come from:

- An incoming, external clock from a clock I/O bank:
  - The incoming signal must be in the same corner as the PLL (designated by NE, NW, SE, or SW) as designated by the Placement value (see example, above).
  - Clock I/O bank signals cannot directly route to other corners.
- Another PLL (considered PLL *cascading*). PLL cascading can occur within the same corner and adjacent corners that are populated with PLLs. Not all corners are populated on all devices. For example: a PLL in the SW corner could route to a PLL in the NW and SE corners of the device, but not diagonally across to a PLL in the NE corner (see example, below).

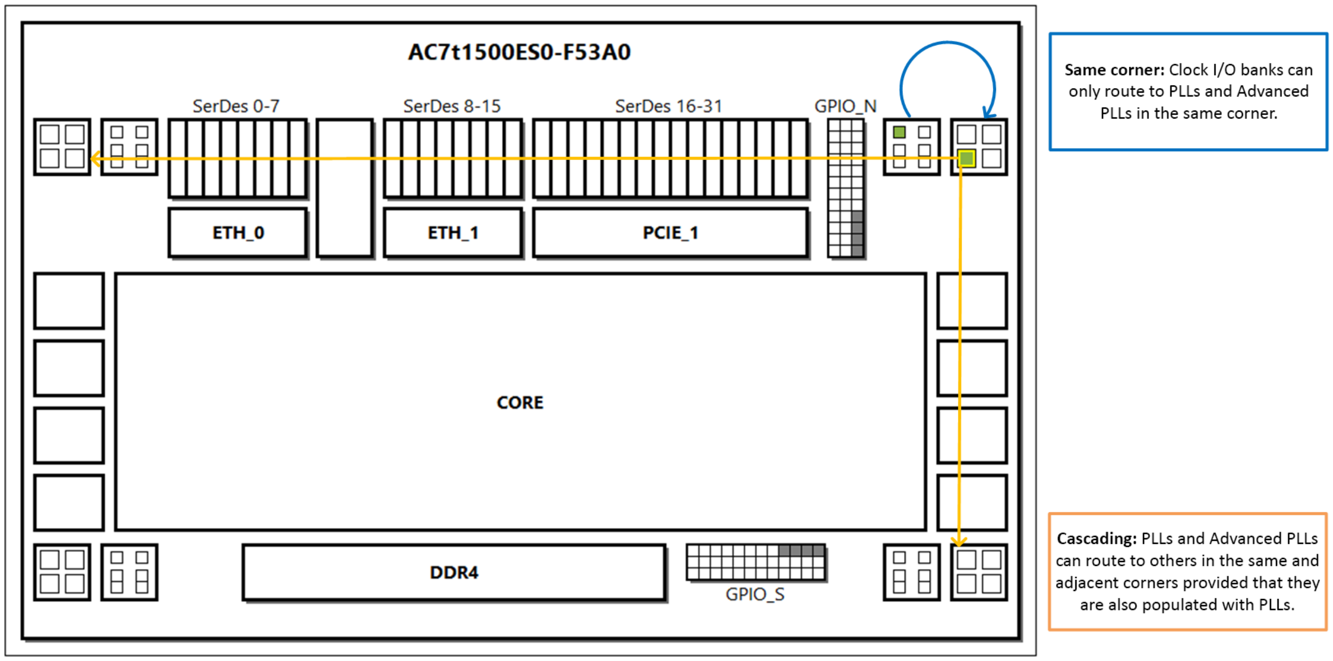

AC7t1500ES0-F53A0
SerDes 0-7
SerDes 8-15
SerDes 16-31
GPIO_N
ETH_0
ETH_1
PCIE_1
CORE
DDR4
GPIO_S
Same corner: Clock I/O banks can only route to PLLs and Advanced PLLs in the same corner.
Cascading: PLLs and Advanced PLLs can route to others in the same and adjacent corners provided that they are also populated with PLLs.

Figure 3 • Clocking Topology Options Example

Most subsystems require selecting a reference clock. There are a few options regarding where this reference clock can be sourced from, though ultimately a PLL is used in all scenarios. The reference clock can be sourced from the following:

- The synthesized output of a PLL (most common configuration).
- An Advanced PLL using the inner or outer bypass mode (specific to the advanced PLL, and cannot be used on a standard PLL):
  - Inner bypass – sends the reference clock to *all* of the advanced PLL outputs (unless any are in the outer bypass mode). The choice of reference clock is limited to a pre-determined clock I/O that varies with the advanced PLL placement value.
  - Outer bypass – the outputs of advanced PLLs are associated with a specific clock I/O. The outer bypass forwards the incoming signal from the associated clock I/O directly to the advanced PLL output, bypassing the advanced PLL's reference clock.

For more information on clocking, see the ***Speedster7t Clock and Reset Architecture User Guide*** (UG083)[1].

GPIO Bank Subsystem

Each GPIO bank has 12 pins — 8 for data, 2 for clock, and 2 for auxiliary signals. Some features of the GPIO bank include:

- All types of GPIO bank pins can be configured as differential
- Data and auxiliary pins can be registered

1 https://www.achronix.com/documentation/speedster7t-clock-and-reset-architecture-user-guide-ug083

- Option to select the clock edge used to sample input data or drive output data. This option only applies when the signal is registered.
- When signals are registered, there are three choices for the reset source: internal reset from FCU, global reset from I/O, or local reset from core
- Supports certain HSTL, HSUL, LVCMOS, and SSTL standards

Table 1 • GPIO Banks per Device

Device	Number of GPIO Banks
AC7t800	2
AC7t1500	6

Table 2 • GPIO Bank Pin Utilization

Pin Type	Purpose	Supported Directions
Data [1]	Data only.	Input Output Inout
Clock [2]	1. Clocks for the transmit and/or receive GPIO register stages. 2. Core clocking. 3. SerDes clocking associated with the GPIO bank. If the SerDes ratio value is higher than 1, the SerDes is used in conjunction with the GPIO bank. When this is the case, the GPIO bank configuration options are used to set up the SerDes.	Input Output
Auxiliary	While intended for signals that do not frequently toggle (e.g., resets), auxiliary pins may also be used for data.	Input Output Inout

Table Notes

1. The SerDes used for GPIO is slower than the raw SerDes.
2. If options 1 or 3 are used, the network-on-chip (NoC) must also be instantiated.

Note

A clock is needed when **Rx Register Mode** is checked, **Tx Register Mode** is checked, and/or the **SerDes Ratio** is higher than 1. The clock is selected by **Bank Clock Signal Name** and can come from the expected options (PLLs, advanced PLLs, advanced PLL in bypass) as well as the GPIO bank clock pins. The clock supplies the GPIO bank associated SerDes (if **SerDes Ratio** is set higher than 1) and the receive and/or transmit registers, if used. In some cases, the NoC might need to be instantiated.

The GPIO bank reset can come from three sources:

- **Internal Reset from the FCU**: – the FCU drives the reset. When entering user mode, the FCU releases the reset. This is the only option if the GPIO bank is not registered. Choose this option for simplicity in that hardware automatically performs the function. However, the disadvantage of this option is that, unlike the other two options, this option does not give the user design control over the reset.
- **Global Reset from I/O**: – a global reset track driven by a clock I/O bank input drives the reset. Choose this option to control the reset from an external source.
- **Local Reset from the Core**: – the user design drives the reset from the core fabric direct-connect interface. Choose this option to control the reset from the core. The advantage compared to global reset from I/O is that there is no additional external device required to control the reset since it is coming from the core.

Chapter 3 : Interface Subsystem and Core IP Flow

Overview

Achronix offers two types of IP:

- Core IP ("soft IP") – refers to the fabric core or, the configurable logic of the FPGA. In essence, IP located in the fabric of the device.
- Interface Subsystems ("hard IP") – part of the I/O ring surrounding the core, and includes pins, GPIO, PLLs, memory interfaces, and other hard IP implemented in the surrounding silicon. Hence the term, I/O ring.

IP and the Tool Flow

The tool flow varies depending on which type of IP is being used even though both processes begin in ACE. If using core IP, the user synthesis run must include the output products of the IP generation (e.g., HDL and constraints, if any are generated).

Note

Interface subsystems are not synthesized in Synplify Pro because ACE contains the boundary timing information between the I/O ring and the Core. However, constraints from interface subsystems are often useful as part of synthesis because they define things that otherwise would need to be specified by the user (i.e., clocks are defined as part of the PLL configuration process). In addition, signals traveling between the I/O ring and Core must be included in the top-level user RTL port list.

Typical tool flow is illustrated in the following flow chart:

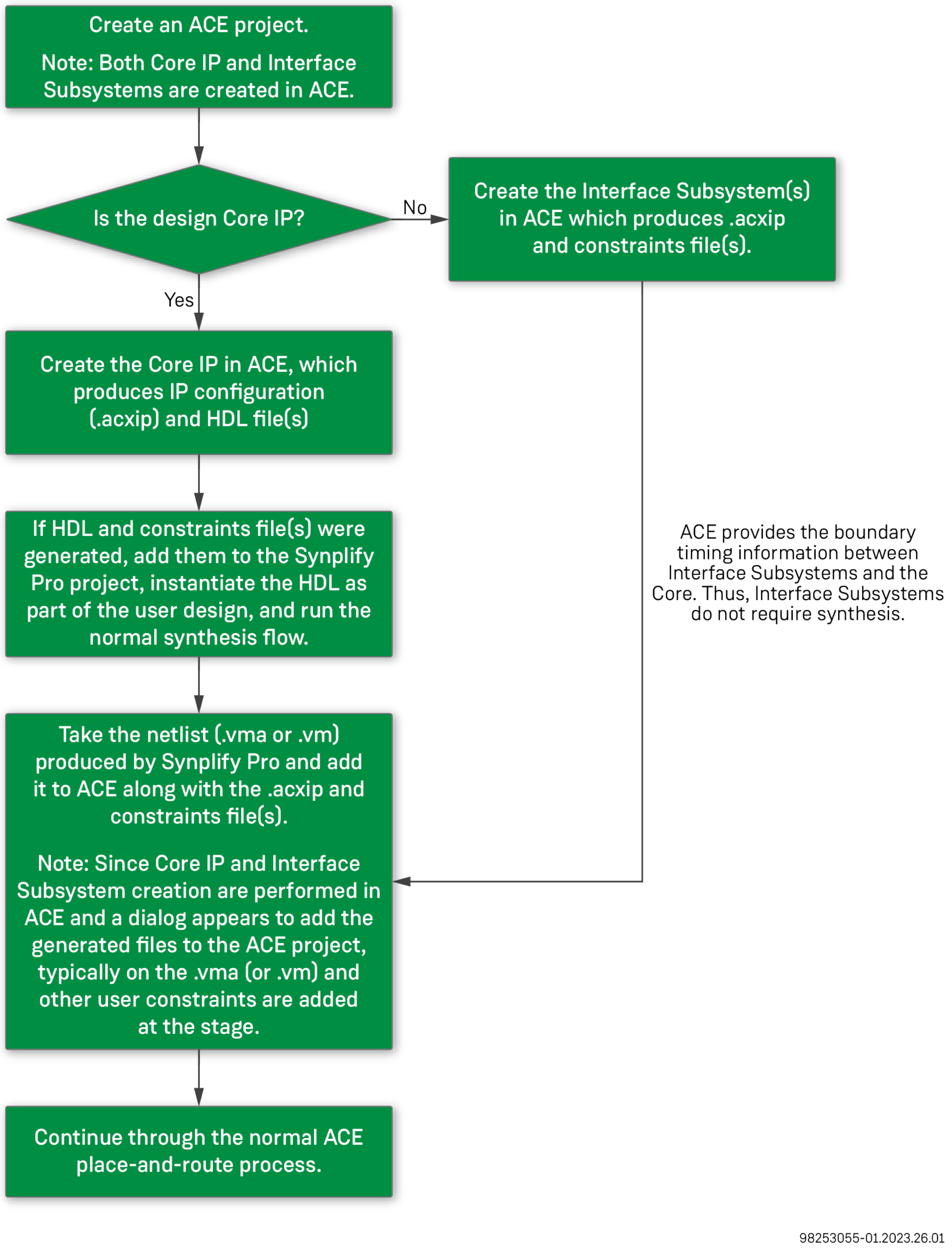
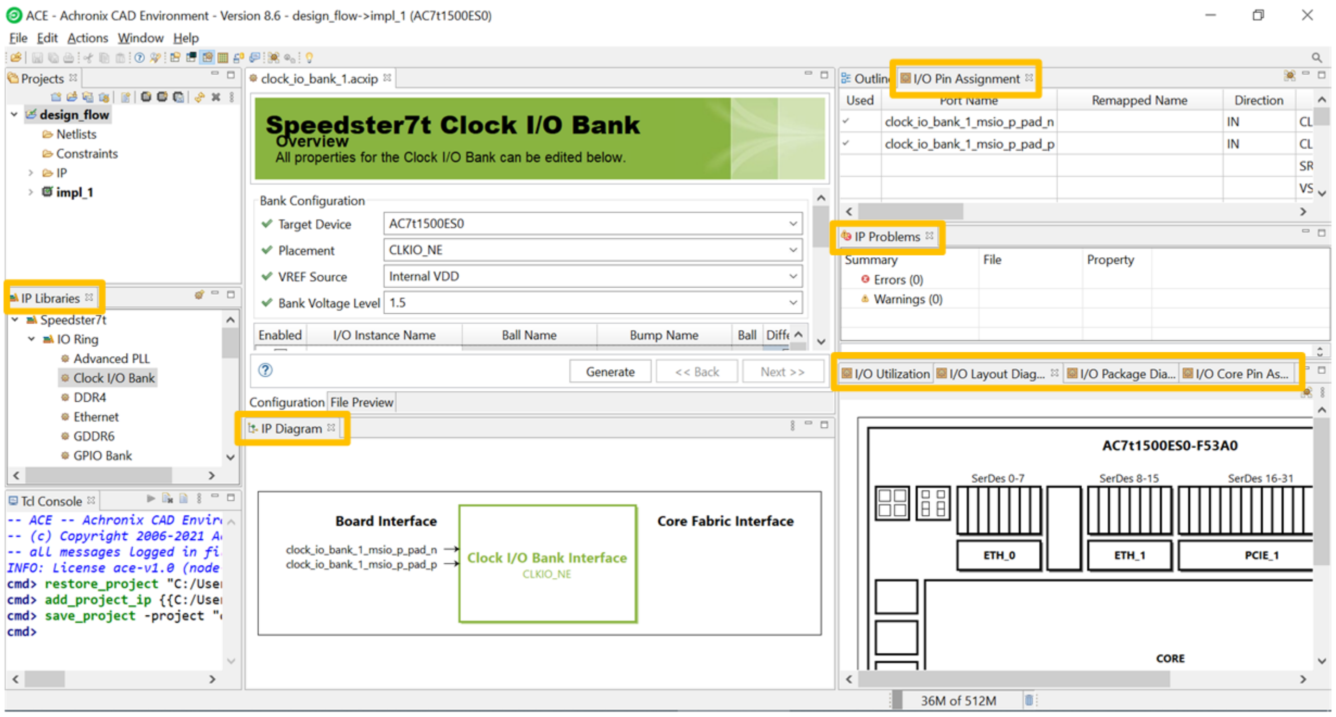

Create an ACE project.
Note: Both Core IP and Interface Subsystems are created in ACE.
Is the design Core IP?
No
Create the Interface Subsystem(s) in ACE which produces .acxip and constraints file(s).
Yes
Create the Core IP in ACE, which produces IP configuration (.acxip) and HDL file(s)
If HDL and constraints file(s) were generated, add them to the Synplify Pro project, instantiate the HDL as part of the user design, and run the normal synthesis flow.
ACE provides the boundary timing information between Interface Subsystems and the Core. Thus, Interface Subsystems do not require synthesis.
Take the netlist (.vma or .vm) produced by Synplify Pro and add it to ACE along with the .acxip and constraints file(s).
Note: Since Core IP and Interface Subsystem creation are performed in ACE and a dialog appears to add the generated files to the ACE project, typically on the .vma (or .vm) and other user constraints are added at the stage.
Continue through the normal ACE place-and-route process.
98253055-01.2023.26.01

Figure 4 • IP Flow

All Interface Subsystems and core IP can be configured in ACE using the IP Configuration perspective:



Bank Configuration

✓ Target Device	AC7t1500ES0
✓ Placement	CLKIO_NE
✓ VREF Source	Internal VDD
✓ Bank Voltage Level	1.5

Enabled	I/O Instance Name	Ball Name	Bump Name	Ball	Diffe

```
-- ACE -- Achronix CAD Envir
-- (c) Copyright 2006-2021 A
-- all messages logged in fi
INFO: License ace-v1.0 (node
cmd> restore_project "C:/Use
cmd> add_project_ip {{C:/Use
cmd> save_project -project "
cmd>
```

Used	Port Name	Remapped Name	Direction	
✓	clock_io_bank_1_msio_p_pad_n		IN	CL
✓	clock_io_bank_1_msio_p_pad_p		IN	CL
				SR
				VS

Summary	File	Property
Errors (0)		
Warnings (0)		

Figure 5 • IP Configuration Perspective

ACE GUI Views for IP Configuration:

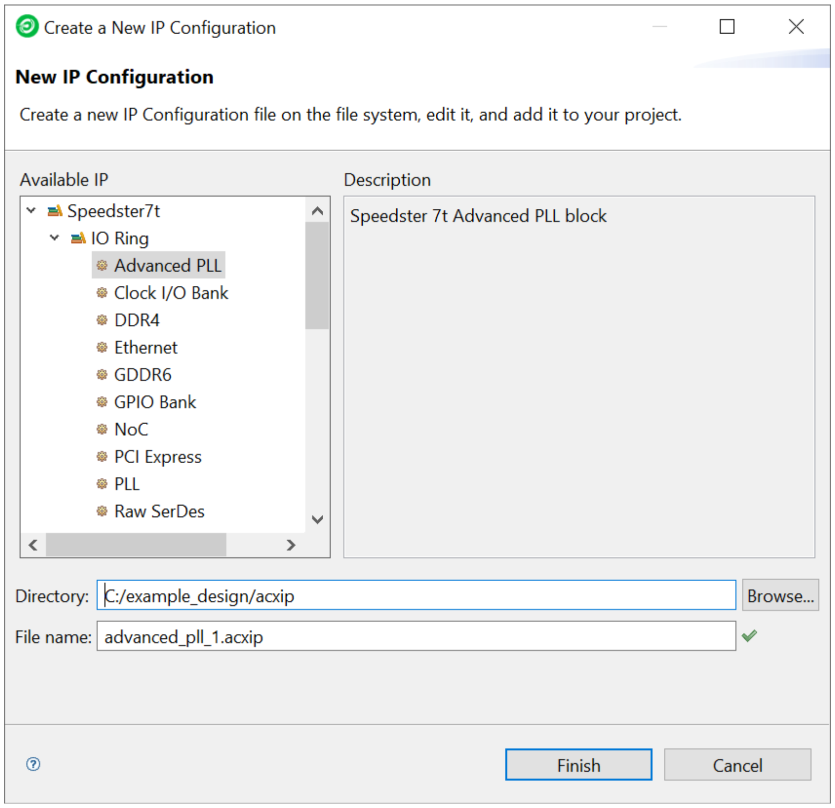
- IP Libraries – create a new instance of an interface subsystem or core IP.
- IP Diagram – shows a block diagram of the interface subsystem or core IP currently being viewed.
- I/O Designer – this view is actually a collection of views as follows:
  - I/O Utilization – shows a summary of interface subsystem utilization.
  - I/O Package Diagram – shows a diagram of color-coded package balls (hover the cursor over a ball for more detailed information).
  - I/O Pin Assignment – details pin assignment information between the I/O ring and external world, including port name (and its remapped name, if renamed), bank location, package ball, pad/macro site name (for debugging in the full-chip simulation hierarchy), and more.
  - I/O Core Pin Assignment – details pin assignment information between the I/O ring and Core, including signal name (and its remapped name, if renamed), direction, data type, Core pin name, and more.
  - I/O Layout Diagram – shows a basic block diagram of the device with I/O ring subsystems shown in green, and can be dragged-and-dropped to other locations that match the subsystem (e.g., PLLs can be dragged-and-dropped to other PLL sites).
- IP Problems – details warning and errors associated with I/O ring subsystems and core IP.

Design Flow Steps

Generating the I/O Ring Files

Using the IP Configuration Perspective in ACE, follow these steps to generate I/O ring subsystems:

1. Configure each I/O ring subsystem by double-clicking the desired subsystem in the **IP Libraries** view as shown in the image, above. Upon initial selection of an I/O ring subsystem, choose the save location for the `.acxip` file (the `.acxip` file holds the subsystem configuration information). A common practice is to create a directory called `acxip` at the same level as the ACE project, and store all of the `.acxip` files there as illustrated below.



New IP Configuration

Create a new IP Configuration file on the file system, edit it, and add it to your project.

Available IP
- Speedster7t
  - IO Ring
    - Advanced PLL
    - Clock I/O Bank
    - DDR4
    - Ethernet
    - GDDR6
    - GPIO Bank
    - NoC
    - PCI Express
    - PLL
    - Raw SerDes

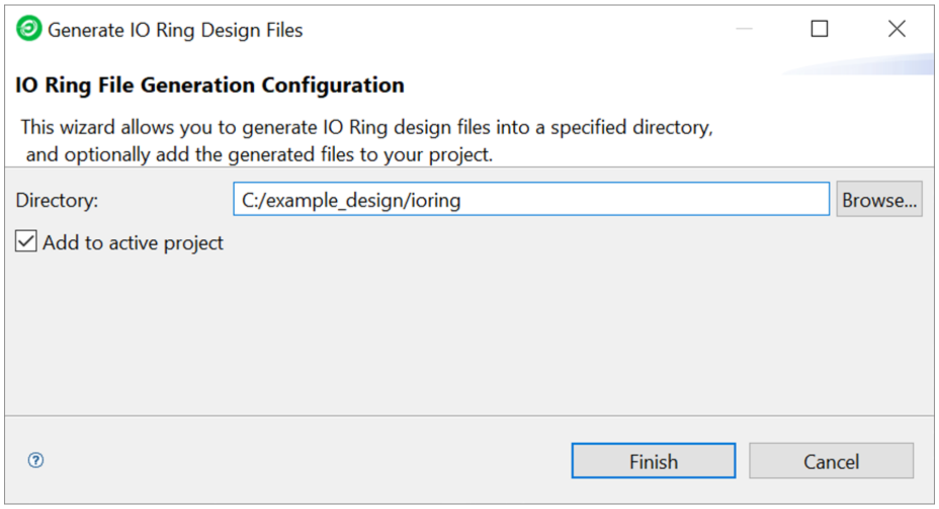
Description: Speedster 7t Advanced PLL block

Directory: C:/example_design/acxip

File name: advanced_pll_1.acxip

Figure 6 • New IP Configuration Dialog – I/O Ring Example

2. After saving all the .acxip files, click the **Generate** button in any of the I/O ring subsystem configuration windows to initiate the generation process for all configured subsystems. A common practice is to create a directory called `ioring` at the same level as the ACE project, and then store all generated I/O ring subsystem there as illustrated below.



IO Ring File Generation Configuration

This wizard allows you to generate IO Ring design files into a specified directory, and optionally add the generated files to your project.

Directory: C:/example_design/ioring

Figure 7 • Generate I/O Ring Design Files Dialog

3. While each subsystem has a `.acxip` file associated with it, other files are also generated as part of the I/O ring flow. In particular, multiple SDC files that cover different PVT points and simulation support files are created. It is recommended to select **Add to active project** to add the generated files relevant to the ACE project (as opposed to generated files that support simulation). This action adds the following files:
   - `<project>_ioring.sdc:` – constraints for clock definitions of clocks traveling from the I/O ring to the core
   - `<project>_ioring_timing_delays_<speedgrade_voltage_temp_corner>.sdc:` – constraints for timing delays on signals traveling between the I/O ring to the core
   - `<project>_ioring.pdc:` – constraints for pin placement for signals traveling between the I/O ring to the core
   - `<project>_ioring_util.xml:` – used for bitstream generation
   - `<project>_ioring_bitstream*.hex:` – the `.hex` file associated with each interface subsystem is taken and combined to produce two of these files (`<proj>_ioring_bitstream0.hex` and `<proj>_ioring_bitstream1.hex`) which are used as part of the final bitstream generation for the ACE project

   Note

   See the Working with Constraints (page 14) section for information on other constraints that should be added by the user. Also see the Simulating the the I/O Ring (page 24) section for the generated files used to support simulation.

4. At this point, the I/O ring files have been generated and added to the project. The next step is to incorporate signals traveling between the I/O ring and the core.

Note

Interface subsystems are not instantiated in RTL.

Incorporating Signals Traveling Between the I/O Ring and Core

During place-and-route, ACE connects signals going to/from the I/O ring and the core by name-matching the I/O ring signals to the top-level user RTL port list. In other words, the top-level user RTL port list should include the signal names going to/from the I/O ring and core. This typically includes the following signals:

- Clock signals going to core
- PLL lock signals
- GPIO signals
- Direct connect interface (DCI) signals of GDDR6, DDR4, PCIe, Ethernet, and Raw SerDes

Note

- To know which signals need to be in the top-level RTL port list, refer to the **I/O Core Pin Assignment** tab or the `/ioring/<project>_user_design_port_list.svh` file.
- The `<project>_user_design_port_list.svh` file is a portlist from the core perspective (i.e., PLL outputs going to the core are listed as inputs). ACE is expecting to see the ports listed in this file as part of the portlist in the top-level RTL file.

1. When the signals have been added to the port list, synthesize the RTL in Synplify Pro to generate the netlist.
2. Add the netlist to the ACE project used to create the interface subsystems and continue with place-and-route/ bitstream generation. Using the same project that generated the subsystems is convenient because the relevant I/O ring files have already been added.

Chapter 4 : Working with Constraints

Types of Constraints

Three types of constraints files are used in the Synplify/ACE flow as shown in the following table:

Table 3 • Synplify Pro and ACE Flow Constraint File Types

File Type	Used in Synplify	Used in ACE	Description
Synopsys Design Constraint (SDC)	Yes	Yes	Only used for timing constraints. While SDC files are normally capable of also constraining power and area, the ACE flow differs in this regard.
FPGA Design Constraint (FDC)	Yes	No	Used for non-timing and non-placement constraints (e.g., global or local attributes on an object, when using the define_attribute statement, and compile points).
Placement Design Constraint (PDC)	No	Yes	Used for placement constraints (region, routing, etc.) and ACE-specific commands.

Warning

The syntax of SDC constraints can vary between Synplify Pro and ACE. While the main command is typically the same, command options and field order may differ.

SDC files and a PDC file are created as part of the I/O ring generation. These files hold all constraints necessary for the I/O ring, including constraints at the boundary between the I/O ring and core. The generated SDCs constrain I/O ring timing, whereas the generated PDCs have constraints that define which signals correspond to which boundary pins in hardware. Some user-defined constraints might be necessary. For example:

- Clock relationships (e.g., false paths, multicycle paths, asynchronous relationships, and clock groups). Clocks are assumed to be related unless defined otherwise.
- Design-specific placement constraints (e.g., region locking and NAP placement)

Caution

User-created PDCs should be placed *below* the generated placement constraints.

Using Constraints

Synplify Pro

At minimum, an SDC file with basic timing constraints should be included. This file can often be used in ACE as well as Synplify Pro, though syntax may differ for some options as synthesis constraints tend to be a subset of place-and-route constraints. Certain characters (e.g., "{ }" and "[ ]") should be escaped for the constraint to be compatible with both Synplify Pro and ACE. FDC files can also be included to define non-timing and non-placement constraints. Since placement occurs in ACE, the PDC file created during I/O ring generation is not used in Synplify Pro.

See the example below for escaping "[ ]" characters allowing both Synplify Pro and ACE to use a single `set_input_delay` constraint:

Special Character Escaping Example for Constraints

```
# Example for ACE only:
set_input_delay  -clock sys_clk -min  1.0 [get_ports din[*]]

# Example for both ACE and Synplify Pro:
set_input_delay  -clock sys_clk -min  1.0 [get_ports din\[*\]]
```

Synplify Pro constraints with the form `syn_<constraint>` tend to directly affect synthesis, whereas constraints with a different syntax are typically passed to the netlist but are not applied during synthesis. There are three ways to apply constraints:

- In RTL before the object declaration (Verilog 2001 style). In this case, there must be a comma separator for multiple entries.
- In RTL after the object declaration. In this case:
  - The `synthesis` keyword must appear before the constraint.
  - Do NOT place a comma separator between multiple entries.
- In an FDC file. FDC files can be manually generated or edited through the Synplify Pro constraint editor. Normally used for project-wide attributes.

See the example below for applying constraints both before and after the object declaration. In the example, `syn_preserve` prevents `coeff_buf_pre` and `coeff_buf` from being optimized away during synthesis. `must_keep` is passed to the netlist and is used to prevent optimization of both objects during place-and-route in ACE.

Applying Constraints in RTL

```
// Buffering coefficients across the die

// Using two layers
```

```
(* must_keep=1, another_attribute=1 *) reg [2:0] coeff_buf_pre [3:0] /* synthesis
syn_preserve=1 syn_maxfan=1 */;
(* must_keep=1, another_attribute=1 *) reg [2:0] coeff_buf [3:0]     /* synthesis
syn_preserve=1 syn_maxfan=9 */;

// Using one assignment

always @(posedge clk)
begin
    coeff_buf_pre <= {4{coeff_sel}};
    coeff_buf     <= coeff_buf_pre;
end
```

See the example below for setting project-wide constraints in an FDC file. In this case, wide MUXes are enabled through `syn_acx_mux41_opt` and the maximum number of registers that can be mapped to an inferred RAM (400 in this case) through `syn_max_memsize_reg`.

Setting Constraints Through an FDC

```
# Enable wide muxes
define_global_attribute  {syn_acx_mux41_opt} {1}

# Set size of registers to infer memories
define_global_attribute  {syn_max_memsize_reg} {400}
```

Common Usage: Limiting Fanout and Preventing Optimization

Limiting fanout is particularly useful when a design is failing timing and contains nets with a high fanout. Higher fanout nets tend to be more difficult to route, and as a result are a common source of timing failures. One solution to this problem is using `syn_maxfan` along with `syn_preserve` and `must_keep`. `syn_maxfan` works on ports, nets, or register outputs by creating a duplicated tree for routing. `syn_preserve` and `must_keep` prevent optimization in Synplify and ACE, respectively.

In the example below, `syn_preserve` prevents the flops from being merged, with `must_keep` being passed in the netlist to perform the same function in ACE.

Setting Constraints to Limit Fanout

```
// Buffering coefficients across the die

// Using two layers:

(* must_keep=1 *) reg [2:0] coeff_buf_pre [3:0] /* synthesis syn_preserve=1 syn_maxfan=1
*/;
(* must_keep=1 *) reg [2:0] coeff_buf [3:0]     /* synthesis syn_preserve=1 syn_maxfan=9
*/;
```

```
// Using one assignment:

always @(posedge clk)
begin
    coeff_buf_pre <= {4{coeff_sel}};
    coeff_buf     <= coeff_buf_pre;
end
```

Note

`syn_maxfan` can be ignored under certain conditions.

For more extensive information on Synplify attributes, within Synplify Pro select **Help** → **Help Topics**. Within the **Contents** tab, select **Attribute Reference Manual**. The *Synthesis User Guide* (UG018)[2] also contains helpful information on constraints.

ACE

All ACE projects contain both SDC and PDC files as a result of I/O ring generation and/or user creation.

There are two ways to add ACE constraints:

- Added directly to an entity as part of the synthesis flow. These attributes are passed into the netlist created from synthesis.
- Added in a constraint file used in ACE. Recall that FDCs are only used in Synplify, so this would be a PDC or SDC file.

If constraints are added as part of the synthesis flow, it is good practice to check that they were applied by reviewing the netlist or by using the Synplify Technology Viewer. To do so through the Technology Viewer, select the Technology View in Synplify → Select the object of interest → Right-click and select *Properties* → Since the Technology Viewer shows the generated netlist, check that the constraint in question has the expected value.

If added through a constraint file in ACE, a good practice is to check for the existence of an object before applying the attribute. This is because remapping/renaming can change the expected name of the object.

For more information on ACE constraints, see the *ACE User Guide* (UG070)[3].

Note

As of ACE release 10.3, Achronix will no longer publish ACE GUI help in the form of a PDF user guide. The contents are accessible via the built-in ACE help system.

Common Usage: Preventing Optimization

To prevent optimization, `must_keep` can be applied to instances or nets to prevent them being optimized away. See Common Usage: Limiting Fanout and Preventing Optimization (page 16) for an example.

2 https://www.achronix.com/documentation/synthesis-user-guide-ug018
3 https://www.achronix.com/documentation/ace-user-guide-ug070

Common Usage: Limiting Fanout

Fanout can be limited either globally or by net. If a global fanout limit is desired, it can be set through the ACE GUI by first enabling by selecting **Options** → **Advanced Design Preparation** → **Fanout Control**. Then, specify the global fanout limit for all nets via **Fanout Limit**, or the limit for critical nets via **Fanout Limit for Critical Nets**. If it is instead preferred, set the global limit through the Tcl Console, either directly or as in Achronix example designs through the `ace_options.tcl` file (under `/src/constraints`), using `set_impl_option` to turn on `fanout_control` and set `fanout_limit` (or `critical_fanout_limit`) as follows:

Global Fanout Limit Example

```
# Enables fanout limit control
set_impl_option -project "<project_name>" -impl "<implementation_name>" "fanout_control"
"1"

# Sets the global fanout limit
set_impl_option -project "<project_name>" -impl "<implementation_name>" "<fanout_limit |
critical_fanout_limit>" "<number_to_limit_fanout_to>"
```

To set the fanout limit for a specific net:

Specific Net Fanout Limit Example

```
# Enables fanout limit control
set_impl_option -project "<project_name>" -impl "<implementation_name>" "fanout_control"
"1"

# Sets the fanout limit for a specific net
set_property fanout_limit <fanout_limit_number> <net>
```

Observe that while limiting fanout in this way is a constraint in that it constrains the design, it is not actually set in a constraint file. Instead, it is set as an implementation option for the project.

For more information on ACE constraints, see the *ACE User Guide* (UG070)[4].

4 https://www.achronix.com/documentation/ace-user-guide-ug070

Chapter 5 : Pin Naming/Mapping

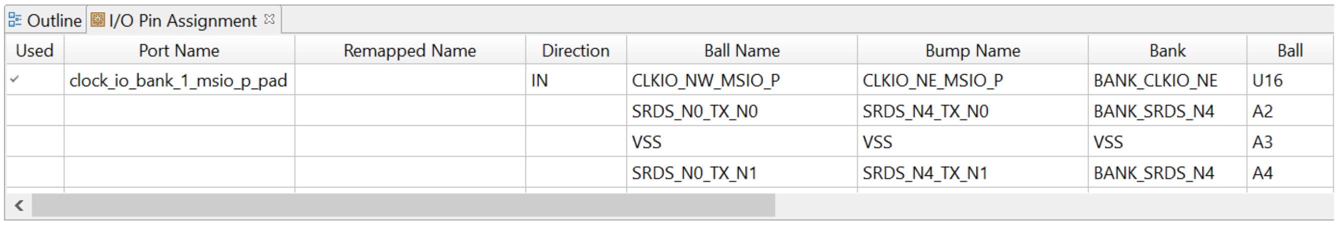
The orientation of Achronix Speedster7t FPGAs varies depending on the view in ACE. When observed in the I/O Layout Diagram view, the perspective is from that of a bare die placed flat with the substrate down. The I/O Package Diagram view is from the perspective of the die having been flipped horizontally and bonded to the package. The I/O Pin Assignment table (shown below) lists a given port name along with its signal direction, its ball name which corresponds to the I/O Package Diagram view orientation, its bump name which corresponds to the I/O Layout Diagram view orientation, and its ball, which is a unique name identifying the final connector on the package which bonds to the circuit board and remains constant regardless of the view. Interface Subsystems follow the orientation of the I/O Layout Diagram view

Note

Due to flip-chip packaging, the ball and bump names are horizontally opposite of each other, east becoming west and vice versa.

Note

Many prefer to identify a given port by its ball, which is remains consistent between circuit boards and the views in ACE.



Used	Port Name	Remapped Name	Direction	Ball Name	Bump Name	Bank	Ball
✓	clock_io_bank_1_msio_p_pad		IN	CLKIO_NW_MSIO_P	CLKIO_NE_MSIO_P	BANK_CLKIO_NE	U16
				SRDS_N0_TX_N0	SRDS_N4_TX_N0	BANK_SRDS_N4	A2
				VSS	VSS	VSS	A3
				SRDS_N0_TX_N1	SRDS_N4_TX_N1	BANK_SRDS_N4	A4

Figure 8 • ACE I/O Pin Assignment Table

The following example shows how a given clock pin appears in the I/O Layout Diagram view. The bump name in the table above corresponds to this view:

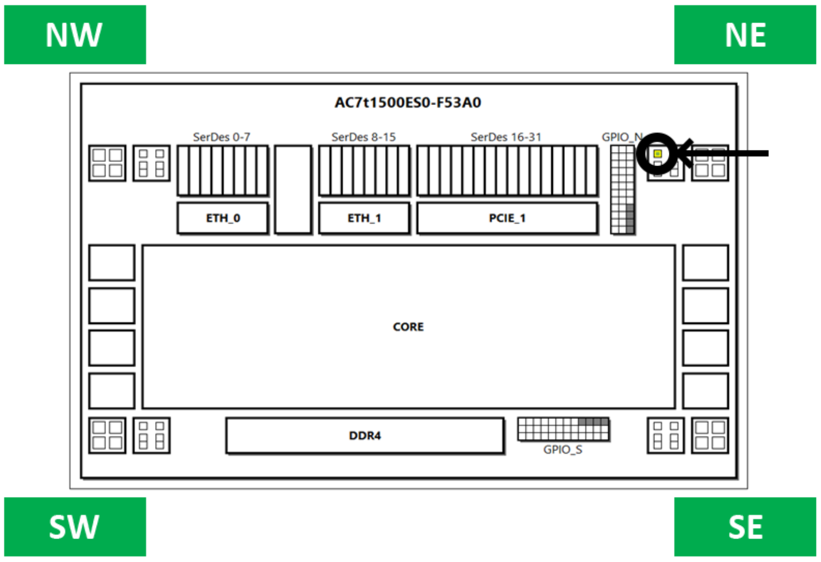


Figure 9 • I/O Layout Diagram View Example

The following example shows the same pin in the I/O Package Diagram view. The ball name in the table above corresponds to this view:

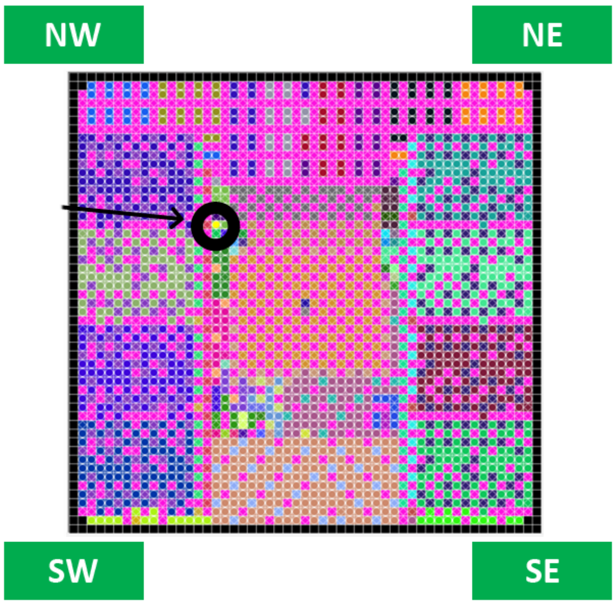
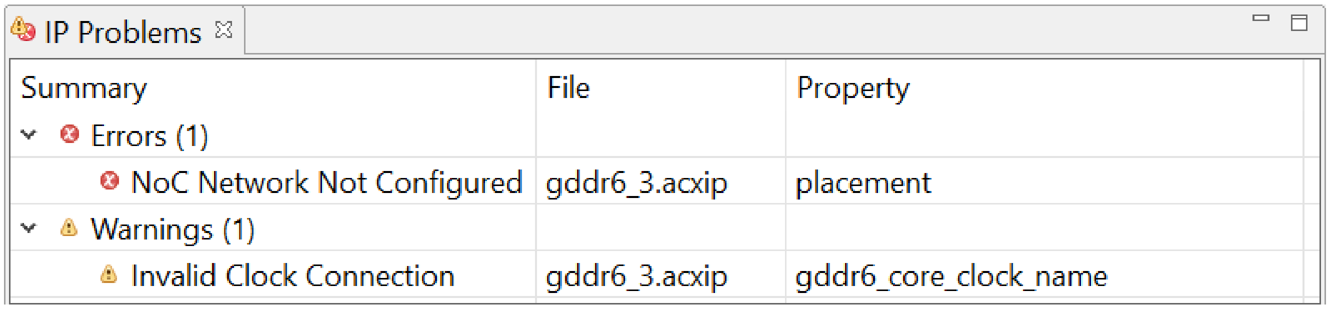


Figure 10 • I/O Package Diagram View Example

Chapter 6 : Evaluating Warnings and Errors

I/O Ring and core IP Generation

I/O ring and core IP file warnings and errors appear in the IP Problems view as shown below. All IP and interface subsystems must be warning and error free before any of their files can be generated.

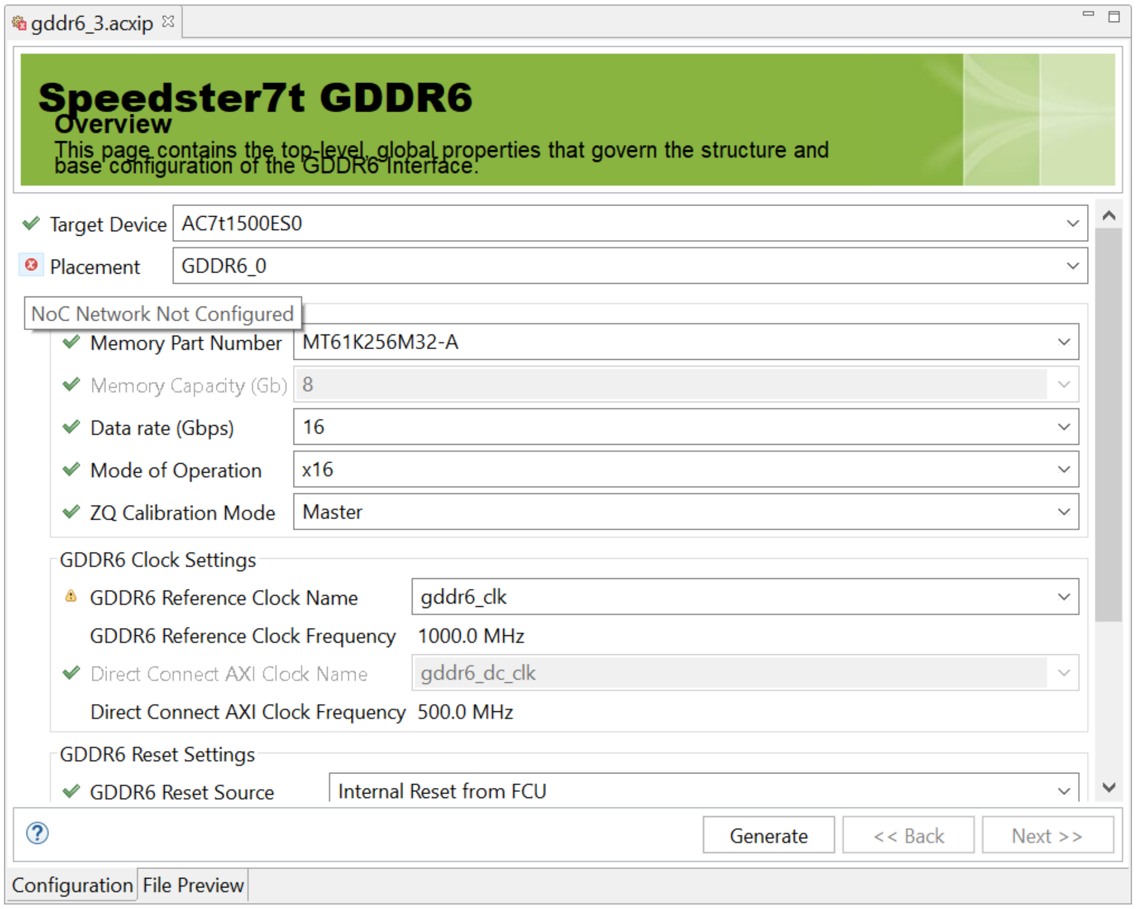


Summary	File	Property
Errors (1)		
NoC Network Not Configured	gddr6_3.acxip	placement
Warnings (1)		
Invalid Clock Connection	gddr6_3.acxip	gddr6_core_clock_name

Figure 11 • IP Problems View Example

The property item corresponds to the file named in the file column which is edited through the IP configuration view of each interface subsystem or core IP. The problem is indicated in the summary field.

Within the Configuration Overview for an interface subsystem or core IP (see the following example), an icon appears to the left of most fields indicating the validity of its value. The green checkmark (✓) indicates there are no problems with the value in the field. A warning (⚠) or error (✖) icon appears when the field value has one or more problems. A descriptive tooltip appears (as shown) with a summary of the warning or error from the IP Problems view when the cursor is placed over one of the problem indicators.



gddr6_3.acxip

Speedster7t GDDR6

Overview

This page contains the top-level, global properties that govern the structure and base configuration of the GDDR6 Interface.

Target Device	AC7t1500ES0
Placement	GDDR6_0

NoC Network Not Configured

Memory Part Number	MT61K256M32-A
Memory Capacity (Gb)	8
Data rate (Gbps)	16
Mode of Operation	x16
ZQ Calibration Mode	Master

GDDR6 Clock Settings

GDDR6 Reference Clock Name	gddr6_clk
GDDR6 Reference Clock Frequency	1000.0 MHz
Direct Connect AXI Clock Name	gddr6_dc_clk
Direct Connect AXI Clock Frequency	500.0 MHz

GDDR6 Reset Settings

GDDR6 Reset Source	Internal Reset from FCU

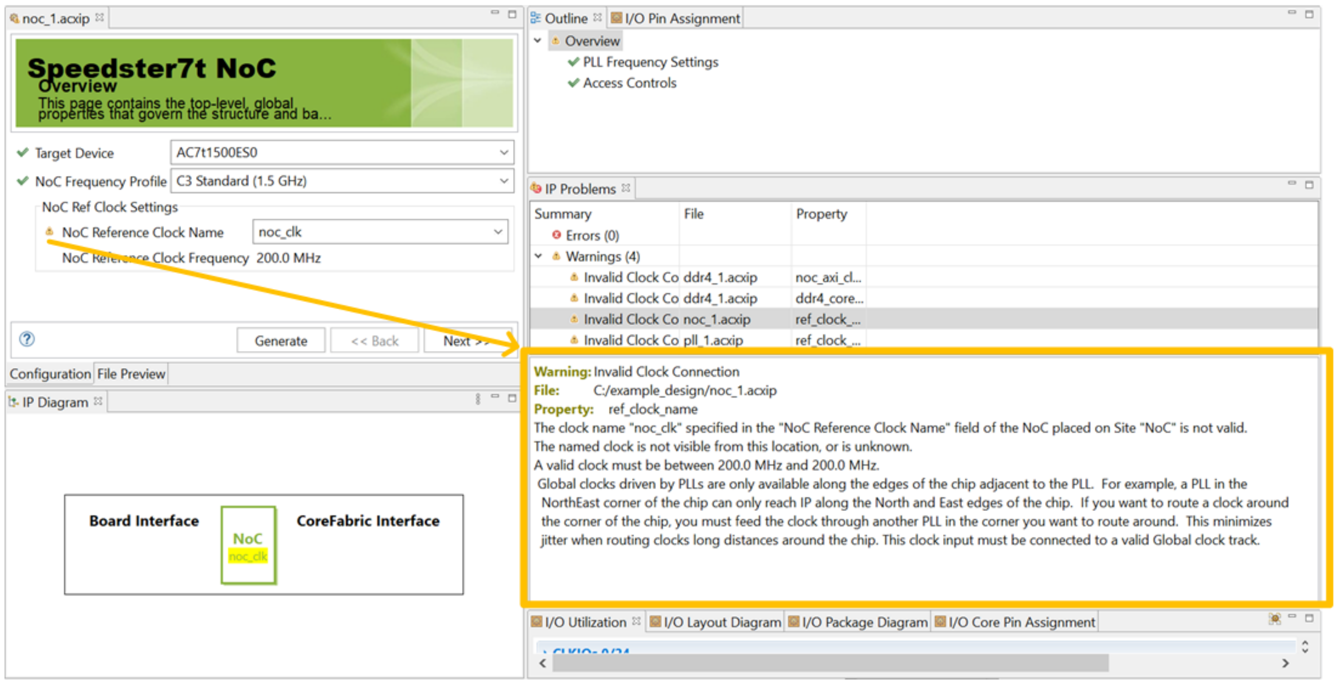
Figure 12 • Core IP Configuration Overview Example

Resolving Warnings and Errors in I/O Ring Generation

Clicking a warning or error icon in the Configuration Overview (or clicking the description of the warning or error itself in the IP Problems view) opens the box highlighted in the example below as part of the IP Problems view.

Observe that the `noc_clk` signal in the IP Diagram view is highlighted in yellow and the warning icon appears to its left in both the Configuration Overview and the IP problems view, visually indicating the issue. In this example, the problem could be one of two possibilities:

- The clock has a different name with the correct frequency. In this scenario, either select the desired clock from the **NoC Reference Clock Name** listbox or rename the clock to `noc_clk` in its PLL configuration view. Otherwise, create a new clock.
- The clock has the correct name but incorrect frequency. The highlighted box of the IP Problems view indicates that the Speedster7t NoC reference clock must be 200.0 MHz. In this scenario, correct the frequency of the clock in its PLL configuration view. Otherwise, create a new clock.



Setting	Value
Target Device	AC7t1500ES0
NoC Frequency Profile	C3 Standard (1.5 GHz)
NoC Reference Clock Name	noc_clk
NoC Reference Clock Frequency	200.0 MHz

Summary	File	Property
Errors (0)		
Warnings (4)		
Invalid Clock Co	ddr4_1.acxip	noc_axi_cl...
Invalid Clock Co	ddr4_1.acxip	ddr4_core...
Invalid Clock Co	noc_1.acxip	ref_clock_...
Invalid Clock Co	pll_1.acxip	ref_clock_...

Warning: Invalid Clock Connection
File: C:/example_design/noc_1.acxip
Property: ref_clock_name
The clock name "noc_clk" specified in the "NoC Reference Clock Name" field of the NoC placed on Site "NoC" is not valid.
The named clock is not visible from this location, or is unknown.
A valid clock must be between 200.0 MHz and 200.0 MHz.
Global clocks driven by PLLs are only available along the edges of the chip adjacent to the PLL. For example, a PLL in the NorthEast corner of the chip can only reach IP along the North and East edges of the chip. If you want to route a clock around the corner of the chip, you must feed the clock through another PLL in the corner you want to route around. This minimizes jitter when routing clocks long distances around the chip. This clock input must be connected to a valid Global clock track.

Figure 13 • IP Problems View Detail Description Box Example

Common sources of warnings and errors include:

- Having a clock with incorrect frequency
- Attempting to use SerDes lanes that are already occupied by another subsystem
- Attempting to route PLLs or Advanced PLLs to certain subsystems on the other side of the device

Chapter 7 : Simulating the I/O Ring

Simulation Support Files Created During I/O Ring Subsystem Generation

As part of I/O ring generation, several files that support simulation are created. These files, along with the user design and the Device Simulation Model (DSM), are key to simulating Achronix devices. This section describes the simulation support files and the modes of the DSM.

The I/O ring can be simulated with either of the two main DSM modes:

- Full-chip Bus Functional Model (BFM) – offers compromised cycle accuracy in I/O ring models with higher simulation speed.
- Full-chip Register Transfer Level (RTL) – offers cycle-accurate models of the I/O ring but with lower simulation speed.

A third simulation mode, Standalone, is not part of the DSM and therefore does not include I/O ring models. Instead, Standalone mode exclusively simulates the core.

Tip

A reference design is recommended as a starting point for simulating user designs and significantly reduces the time required to set up simulation.

As part of the I/O ring generation process, files are created supporting I/O ring simulation. While the `.acxip` file associated with each subsystem is not directly used in simulation, it is the basis for producing the files that support simulation of interface subsystems. These files are described in the following table.

Table 4 • Interface Subsystem Simulation Files

File	Description
`<proj>_sim_defines.f`	Includes defines needed to set global clocks and resets in the I/O ring. Must be included in the simulation file list.
`<proj>_user_design_port_bindings .svh`	Binds signal names in the RTL for signals traveling between the core and the I/O ring to core pins (i.e., a user-renamed I/O ring port `clk_in` is a named version of a hardware port from the core, e.g., `i_user_06_00_trunk_00[31]`, and must be bound so that it can be properly connected by the tool). See the Device Simulation Model (page 30) section for more details.

File	Description
`<proj>_user_design_port_list.svh`	A list of ports used to connect signals traveling to and from the core and I/O ring from the perspective of the core (i.e., PLL outputs going to the core would be listed as inputs). Must be included in the top-level RTL port list.
`<proj>_user_design_signal_list.svh`	Same as above but uses "logic" declaration. Can be included in the test bench.
`<proj>_sim_config.svh`	Calls configuration functions for any subsystems using full-chip RTL. Must be included in the test bench.
`<proj>_ioring_bitstream*_<block>.txt`	For each subsystem instance that generates a configuration file, the above `*.svh` file calls these config files when the subsystem is set to full-chip RTL by a define.

Additionally, there are specific defines used to indicate which subsystems to use in full RTL mode. These defines are specified as `<subsystem_name>_FULL` (e.g., `+define+GDDR6_2_FULL`). See the **Supported Simulation Flow Types** (page 27) table for more information.

I/O Ring to Core Connections in Hardware Versus Simulation

The `<proj>_port_list.svh` file holds the list of ports used to connect signals traveling to and from the core and I/O ring (see the previous section for a list of simulation support files created during I/O ring subsystem generation). In hardware, these connections are made between the programmable core and the periphery containing the interface subsystems. In simulation, those connections are formed with the support of the DSM. To compare hardware and simulation:

- For hardware – when generating interface subsystems, a prompt appears requesting to add some files created as part of that process to the ACE project. Among these files are `.acxip` file(s) for IP configuration. This file enables ACE to not only generate a bitstream to configure the respective IP, but also to name the applicable connections to the programmable core. These connections become top-level ports which are connected to the user design netlist during place and route.
- For simulation – a testbench instantiating and connecting to the top-level RTL file is used. Connecting these top-level ports in the user design to the appropriate named ports within the DSM, which is instantiated in the testbench, is key. To accomplish this, one of the files created during the I/O ring generation process, `<proj>_user_design_port_bindings.svh`, contains includes that associate each port by name in the top-level port list with its corresponding hardware pin on the DSM, including Direct Connect Interface (DCI) connections. This method binds each port in the user design to one on the I/O ring model in the DSM. Alternately, each DCI connection System Verilog interface allows users to interact with subsystems, including their DCI connections. See the **Device Simulation Model** (page 30) section for more information.

Note

I/O ring subsystems do *not* use RTL wrappers in either hardware or simulation. In both cases, the I/O ring communicates with the user design in the core through the top-level port list. Unlike previous simulation environments that might have required instantiating the individual RTL wrappers containing the selected IP directly into the testbench, the DSM contains models of all IP within the device. Only a single instance of a DSM is required to simulate all functions of a device.

Here are a few port binding examples:

```
`ACX_BIND_USER_DESIGN_PORT(ddr4_clk, i_user_04_00_mt_00[0])
`ACX_BIND_USER_DESIGN_PORT(ddr4_clk_alt[0], i_user_05_00_mt_00[0])
`ACX_BIND_USER_DESIGN_PORT(ddr4_clk_alt[1], i_user_07_00_mt_00[0])
`ACX_BIND_USER_DESIGN_PORT(ddr4_rstn, i_user_07_00_lut_13[18])
```

I/O Ring Modeling

Two main options exist when simulating:

- **Core Modeling Method**: Whether RTL, gate-level post-synthesis, or gate-level post-route. The method is determined by the type of file(s) used:
  - Design RTL files for an RTL simulation
  - A post-synthesis netlist for a gate-level post-synthesis simulation
  - A post-route netlist for a gate-level post-route simulation
- **I/O Ring Modeling Method**: Whether full-chip BFM or full-chip RTL as described in the following table. The default model is full-chip BFM, although defines can be used to change the model to standalone or full-chip RTL. The standalone model only simulates the core and is not a mode in the DSM.

There are three types of simulation flows supported. While simulation of the core is unaffected them, they differ in how the I/O ring subsystems are modeled:

Table 5 • Supported Simulation Flow Types

Model	Speed	Cycle accuracy	Other notes
Standalone	Fastest	Not cycle accurate	• Standalone is not part of the DSM. Instead, simplified models must be constructed by the user. • NAP-to-NAP, NAP-to-subsystem, and any subsystems (including PLLs, memory interfaces, etc.) need to be modeled in the testbench. A specific model of a NAP connected to an external memory exists, but for other NAP functions, models must be constructed by the user. For example, NAP-to-NAP and NAP-to-subsystem must be modeled because NAPs are not connected to the NoC in this mode. See Special Treatment For Simulating NAPs (page 29) for more detail. • Enabled by `+define+ACX_SIM_STANDALONE_MODE` and not instantiating the DSM during simulator compilation.
Full-Chip BFM	Medium	Cycle accurate for NoC, near cycle accurate for other subsystems	• This mode uses a model of the full chip, with cycle-accurate 2D NoC (2D NoC is RTL modeled). There are 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 simulation time compared to a full cycle-accurate simulation. • This is the default mode of the DSM. No steps are necessary to enable it.

Model	Speed	Cycle accuracy	Other notes
Full-Chip RTL	Slowest	Cycle accurate for RTL modeled subsystems, near cycle accurate for BFM modeled subsystems	• RTL models are used and, if desired, 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 is 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. • Which subsystems use full RTL models and which use BFM models can be selected using define statements (e.g., setting `+define+GDDR6_2_FULL` uses the full RTL model for GDDR6_2). • To obtain the encrypted RTL of the GDDR6, DDR4, or PCIe subsystems, a second licensed simulation package is required. Please contact Achronix Support to arrange licensing and access to this package. • I/O ring subsystems not using full-chip RTL default to full-chip BFM. In this way, full-chip BFM and full-chip RTL can be mixed. • Enabled by `+define+ACX_<HARD_IP>_FULL` during simulator compilation. Must be defined for each subsystem for which RTL modeling is desired. See the table below for more information.

The subsystems that can use the full-chip RTL model are shown in the table below, and represent the `ACX_<HARD_IP>_FULL` part of `+define+ACX_<HARD_IP>_FULL`. For more information on how to use `+define+` to enable the RTL model of an I/O ring subsystem and runtime programming scripts, see the Reference Design Simulation section at the end of any reference design document.

Table 6 • Simulation RTL defines

Module [(1)(2)]	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
GDDR6 controller 7	ACX_GDDR6_7_FULL

Module [1][2]	Define
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
All PLLs and Clock Generators [3]	ACX_CLK_NW_FULL, ACX_CLK_NE_FULL, ACX_CLK_SW_FULL, ACX_CLK_SE_FULL

Table Notes

1. Locations and names of each of the interface subsystems are visible using the ACE IP Configuration perspective, and selecting the I/O layout diagram.
2. Not all interface subsystems are available in every device. Please consult the device datasheet to confirm the precise number and naming of each subsystem.
3. All four defines, one for each corner, must be defined together. Due to shared entities, it is not possible to only define a subset of PLLs and clock generators for RTL simulation.

Special Treatment For Simulating NAPs

I/O ring subsystems communicate with the core through the 2D NoC and NAPs in the core, or through a Direct Connect Interface (DCI) directly from the subsystem to the core. NAPs are the connection point between the 2D NoC and the core for all I/O ring subsystem signals not using DCI connections.

The 2D NoC exists in both the I/O ring and above the core, communicating to the core below it through NAPs. However, there are no 2D NoC ports in the `<proj>_port_list.svh` file. To facilitate simulation, ACE establishes the NoC-to-NAP connections for simulation through NAP binding macros in the case of full-chip BFM and full-chip RTL simulations, or bind statements in the case of standalone simulation.

To summarize the difference between the hardware and simulation flows for using NAPs:

- In hardware, NAPs are enabled in the RTL using NAP macros, and ACE makes the connections between the NoC and NAPs during place-and-route.
- In simulation, the NAP macros exist in the user RTL, but an additional binding macro (in the case of full-chip BFM and full-chip RTL simulations) or bind statement (in the case of standalone simulation) for each NAP is added to the testbench. In the case of full-chip BFM or full-chip RTL, the NAP binding macro requires that the location of each NAP for a simulation must be set by the user. Both the bind statement and bind macro should be set in the testbench.

Caution!

The NoC coordinates used to bind any NAP must match the same coordinates in the ACE project `.pdc` file in order to ensure consistency between simulation and hardware.

For more information on NAPs, see the *Speedster7t 2D Network on Chip User Guide* (UG089)[5] and the DSM section below. For more information on how to run each type of simulation on the different supported simulators, see the *Simulation User Guide* (UG072)[6].

Device Simulation Model

Many designs require a simulation overlay named the device simulation model (DSM). This package combines the full RTL of the 2D 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 2D NoC and models for the interface subsystems allows developing 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, combined with BFMs of the surrounding interface subsystems. The 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 2D NoC access point (NAP). It is then necessary to bind this NAP instance to the NAP in the correct location within the 2D NoC by using the `` `ACX_BIND_NAP_RESPONDER``, `` `ACX_BIND_NAP_INITIATOR``, `` `ACX_BIND_NAP_HORIZONTAL``, `` `ACX_BIND_NAP_VERTICAL`` or `` `ACX_BIND_NAP_ETHERNET`` 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.

Selecting the Required DSM

DSM Utility Package

There is a DSM package for each device, with each DSM representing the specific features of that device. It is therefore necessary to select the correct DSM within a simulation testbench. Selection of the correct DSM is achieved by including the appropriate DSM utility package. The package then creates macros and functions to access the appropriate DSM. The utility package defines the macro `ACX_DEVICE_NAME`, which is then used to instantiate and refer to the DSM. The following DSM utility packages are available.

5 https://achronix.com/documentation/speedster7t-2d-network-chip-user-guide-ug089
6 https://www.achronix.com/documentation/simulation-user-guide-ug072

Table 7 • DSM Utility Packages

Devices	DSM Utility Package	ACX_DEVICE_NAME
AC7t1500	`ac7t1500_utils.svh`	ac7t1500
AC7t1400	`ac7t1400_utils.svh`	ac7t1400
AC7t800	`ac7t800_utils.svh`	ac7t800
AC7t700	`ac7t700_utils.svh`	ac7t700

Device-Specific Simulation Files

To allow for reusable code, the Achronix simulation flow creates a macro for each device, of the form `ACX_DEVICE_<full device name>`. The appropriate macro is present in simulation (and synthesis) when the appropriate ACE library file is included in the project. These ACE library files are located within the `<ACE_INSTALL_DIR>/libraries/device_models/<full device name>_simmodels.sv` file. The following table lists the available `simmodels.sv` files, and the device specific macro that each creates.

Table 8 • Simulation Model Files and Defines

Device	Simulation Model File	ACX_DEVICE Macro
AC7t1500	`AC7t1500_simmodels.sv`	ACX_DEVICE_AC7t1500
AC7t1400	`AC7t1400_simmodels.sv`	ACX_DEVICE_AC7t1400
AC7t800	`AC7t800_simmodels.sv`	ACX_DEVICE_AC7t800
AC7t700	`AC7t700_simmodels.sv`	ACX_DEVICE_AC7t700

Instantiate DSM Utility Package

Using the device specific macros, it is possible to create a general DSM instantiation that can be used for multiple devices. In the following example, the ACX_DEVICE_xxxx macro is used to select the appropriate DSM utility package. The macros subsequently created by the package are then used to select the appropriate DSM.

```
    // Include the appropriate DSM utility file which defines the appropriate macros
    // If an unsupported device is selected, then compilation will fail
`ifdef ACX_DEVICE_AC7t1500
    `include "ac7t1500_utils.svh"
`elsif ACX_DEVICE_AC7t800
```

```
    `include "ac7t800_utils.svh"
`endif

    // Instantiate the DSM
    // ACX_DEVICE_NAME is defined in the DSM utility file for the selected device
    // Connect the chip_ready signal
    `ACX_DEVICE_NAME `ACX_DEVICE_NAME (
        .FCU_CONFIG_USER_MODE   (chip_ready),
    );
```

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>_DSM_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 DSM is used, a task must be included within the design testbench to confirm the version compatibility. This function should be instantiated as follows:

```
    // The ACX_DEVICE_NAME macro is defined for each DSM within its appropriate utility
package
    initial begin
        // Ensure correct version of DSM is being used
        // This design requires 10.1 as a minimum
        `ACX_DEVICE_NAME.require_version(10, 1, 0, 0);
    end
```

require_version( ) Task

The require_version task has four arguments. In order:

1. Major Version – Matches the major version of the release
2. Minor Version – Matches the minor version of the release
3. Patch – Matches the patch version of the release (optional)
4. Update – Matches the update number of the release (optional)

If either patch or update is not specified, then these arguments should be set to 0. For example, for the 10.1 release, the arguments would be set as 10,1,0,0.

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 following diagram (see figure 14). This example shows the macros required for the responder 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 (page 35) and DSM Direct-Connect Interfaces (page 37) tables.

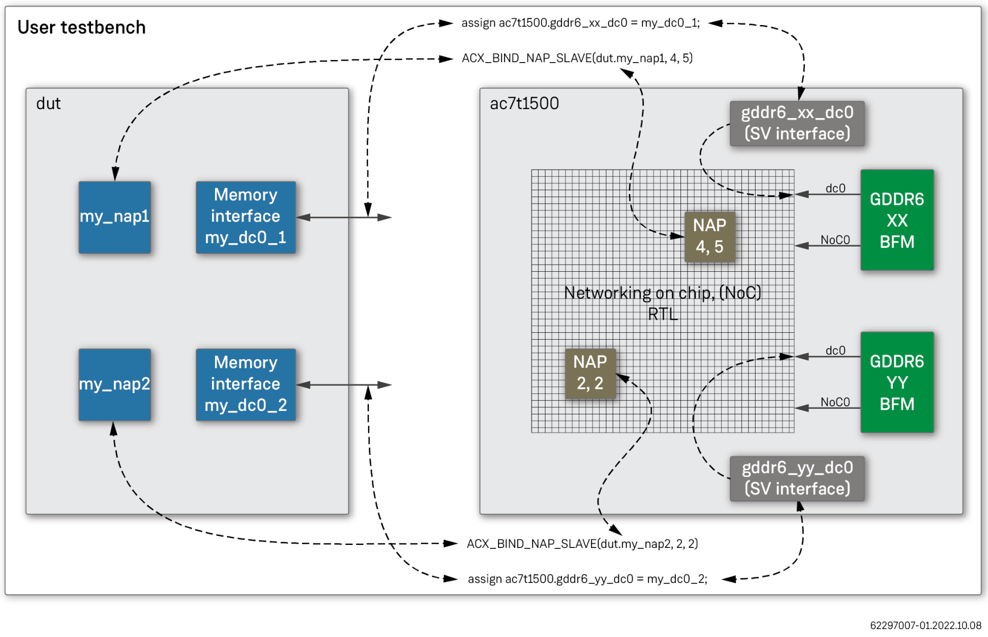

User testbench
assign ac7t1500.gddr6_xx_dc0 = my_dc0_1;
ACX_BIND_NAP_SLAVE(dut.my_nap1, 4, 5)
dut
my_nap1
Memory interface my_dc0_1
my_nap2
Memory interface my_dc0_2
ac7t1500
gddr6_xx_dc0 (SV interface)
dc0
NoC0
GDDR6 XX BFM
NAP 4, 5
Networking on chip, (NoC) RTL
NAP 2, 2
GDDR6 YY BFM
gddr6_yy_dc0 (SV interface)
ACX_BIND_NAP_SLAVE(dut.my_nap2, 2, 2)
assign ac7t1500.gddr6_yy_dc0 = my_dc0_2;
62297007-01.2022.10.08

Figure 14 • Example Simulation Structure

In the previous example, 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_RESPONDER macro:

- This macro supports inserting the coordinates of the NAP within the 2D 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 assigned to the appropriate signals within the device direct-connect SystemVerilog interface.

The Verilog code to instantiate the example, based on using the Speedster7t AC7t1500 FPGA, follows.

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

    // Set the verbosity options on the messages
    // Use the inbuilt set_verbosity() task.
```

```
initial begin
    `ACX_DEVICE_NAME.set_verbosity(2);
end

// ----------------------------------------------
// Bind NAPs
// ----------------------------------------------
// Bind my_nap1 to location 4,5
`ACX_BIND_NAP_AXI_RESPONDER(dut.my_nap1,4,5);
// Bind my_nap2 to location 2,2
`ACX_BIND_NAP_AXI_RESPONDER(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 `ACX_DEVICE_NAME.gddr6_xx_dc0.awvalid  = my_dc0_1_awvalid;
assign `ACX_DEVICE_NAME.gddr6_xx_dc0.awaddr   = my_dc0_1_awaddr;
....
// Outputs from device
assign my_dc0_1_awready = `ACX_DEVICE_NAME.gddr6_xx_dc0.awready;
....

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

// ----------------------------------------------
// Remember to connect the clock!
// ----------------------------------------------
assign my_dc0_1_clk = `ACX_DEVICE_NAME.gddr6_xx_dc0.clk;
assign my_dc0_2_clk = `ACX_DEVICE_NAME.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_RESPONDER(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 previous code sample for an example showing how to call this function.

The verbosity levels are defined in the following table.

Table 9 • Verbosity Levels

Verbosity Level	Description
0	Print no messages.
1	Print messages from initiator and responder 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 until this signal is asserted (shown in the previous example by use of a testbench `chip_ready` signal).

Bind Macros

The following bind statements are available.

Table 10 • Bind Macros

Macro	Arguments (1)	Description
`ACX_BIND_NAP_HORIZONTAL`	`user_nap_instance`, `noc_colunm`, `noc_row`	To bind a horizontal streaming NAP, instance `ACX_NAP_HORIZONTAL`.
`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_INITIATOR` (2)	`user_nap_instance`, `noc_colunm`, `noc_row`	To bind an AXI initiator NAP, instance `ACX_NAP_AXI_INITIATOR`.
`ACX_BIND_NAP_AXI_RESPONDER` (2)	`user_nap_instance`, `noc_colunm`, `noc_row`	To bind an AXI responder NAP, instance `ACX_NAP_AXI_RESPONDER`.
`ACX_BIND_NAP_ETHERNET`	`user_nap_instance`, `noc_colunm`, `noc_row`	To bind an Ethernet NAP instance, `ACX_NAP_ETHERNET`.

Table Notes

1. `user_nap_instance` is relative to the testbench, not to the top of the simulation. Normally `user_nap_instance` would be of the form `DUT.<hierarchical_path_to_nap>`.
2. For the Speedster7t AC7t800 FPGA, these macros are `ACX_BIND_NAP_AXI_INITIATOR` and `ACX_BIND_NAP_AXI_RESPONDER`.

Direct-Connect Interfaces

Within the device, the non-NAP connections between the high-speed interface subsystems (such as GDDR, DDR, Ethernet and SerDes) and the fabric are known as direct-connect interfaces (DCIs). These are comprised of:

- Additional data ports in the case of the memory interfaces (AXI)
- Dedicated data interfaces for SerDes (direct mode)
- Status and control for Ethernet

For full details of each of the subsystem 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
- Using an ACE-generated port binding file

Note

The Speedster7t AC7t800 FPGA only incorporates DCIs for the SerDes direct mode. All other data flows between interface subsystems and the fabric are made using the NAP and 2D NoC.

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 I/O ports to the appropriate locations within the DSM.

- 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.
- 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 as follows.

DSM DC Interfaces

The DSM has a SystemVerilog interface for each DCI port. The available interfaces are listed in the following table.

Table 11 • 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
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
DDR4	`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`	–	–
Serdes	`serdes_eth0_q0_dc`	North West	–	`t_ACX_SERDES_DCI`	128	–
Serdes	`serdes_eth0_q1_dc`	North West	–	`t_ACX_SERDES_DCI`	128	–

Subsystem	Interface Name	Physical Location [1]	GDDR6 Channel	SystemVerilog Interface Type	Data Width	Address Width
Serdes	`serdes_eth1_q0_dc`	North East[2]	–	`t_ACX_SERDES_DCI`	128	–
Serdes	`serdes_eth1_q1_dc`	North East[2]	–	`t_ACX_SERDES_DCI`	128	–

Table Notes

1. Physical orientation west-to-east is with regards to viewing the die in the floorplan view within ACE. The die is actually rotated about its vertical axis when packaged. Therefore, an interface shown on the floorplan, and listed in this table, 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 this table, the ACE view, and the device on board.
2. Present on the Speedster7t AC7t800 DSM.

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 DSM Direct-Connect Interfaces (page 37) table, and the element name.

The following example shows how to connect the `awready` and `awvalid` signals for a GDDR AXI interface.

```
// 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;

// Connect to the DSM GDDR_1, DC port 0.
// awready is an output from the DSM, and an input to the DUT
assign dut_awready = `ACX_DEVICE_NAME.interfaces.gddr6_1_dc0.awready;
// awvalid is an input to the DSM, and an output from the DUT
assign `ACX_DEVICE_NAME.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:

1. Create an ACE project (a netlist is not required at this stage).
2. Configure all interface subsystem IP.
3. Generate the subsystem IP files, including a file named `<design_name>_user_design_port_bindings.svh`.
4. 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.
5. Include the port binding file in the testbench.
6. 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.

Notes

- 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.

The following example shows how to connect all of the DUT ports using the port binding file.

system_files_bfm.f

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

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 capability 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 follows.

```
// 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 = `ACX_DEVICE_NAME.interfaces.gddr6_1_dc0.awready;
    assign `ACX_DEVICE_NAME.interfaces.gddr6_1_dc0.awready = dut_awvalid;
`endif

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

SRM Reset Port Binding

On AC7t1400 and AC7t700 devices, ACE automatically adds the Serdes Rate Monitor Module (SRM) to the design. The SRM reset port is automatically connected, and the user design is not required to have top-level port for it. Because of this, the IO Designer does not generate any bindings for that port for simulation, thus the SRM reset port (`o_serdes_rstn`) must be manually bound to OPIN for DSM simulations.

```
// On AC7t1400, must be bound to o_user_07_00_lut_11[7]
`ACX_BIND_USER_DESIGN_PORT(o_serdes_rstn, o_user_07_00_lut_11[7])

//On AC7t700, must be bound to o_user_06_00_lut_14[8]
`ACX_BIND_USER_DESIGN_PORT(o_serdes_rstn, o_user_06_00_lut_14[8])
```

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 I/O Designer. For simulation, the `set_clock_period` function is provided.

The following example 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
`ACX_DEVICE_NAME.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)
`ACX_DEVICE_NAME.clocks.set_clock_period("gddr6_5_dc0_clk",
GDDR6_CONTROLLER_CLOCK_PERIOD*2);
```

Note

The `set_clock_period` function is within the DSM. This model has a default timescale value of 1ps. 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 12 • Clock Frequency Interfaces

Subsystem	Interface Name	Physical Location [1]	GDDR6 Channel
GDDR6	`gddr6_0_noc0_clk` [3]	West 0 NoC	0
	`gddr6_0_noc1_clk` [3]	West 0 NoC	1
	`gddr6_1_noc0_clk` [3]	West 1 NoC	0
	`gddr6_1_noc1_clk` [3]	West 1 NoC	1
	`gddr6_2_noc0_clk` [3]	West 2 NoC	0
	`gddr6_2_noc1_clk` [3]	West 2 NoC	1
	`gddr6_3_noc0_clk`	West 3 NoC	0
	`gddr6_3_noc1_clk`	West 3 NoC	1
	`gddr6_4_noc0_clk`	East 0 NoC	0
	`gddr6_4_noc1_clk`	East 0 NoC	1
	`gddr6_5_noc0_clk`	East 1 NoC	0
	`gddr6_5_noc1_clk`	East 1 NoC	1
	`gddr6_6_noc0_clk`	East 2 NoC	0
	`gddr6_6_noc1_clk`	East 2 NoC	1
	`gddr6_7_noc0_clk`	East 3 NoC	0
	`gddr6_7_noc1_clk`	East 3 NoC	1
	`gddr6_1_dc0_clk`	West 1 DCI	0
	`gddr6_1_dc1_clk`	West 1 DCI	1
	`gddr6_2_dc0_clk`	West 2 DCI	0
	`gddr6_2_dc1_clk`	West 2 DCI	1
	`gddr6_5_dc0_clk`	East 1 DCI	0
	`gddr6_5_dc1_clk`	East 1 DCI	1
	`gddr6_6_dc0_clk`	East 2 DCI	0
	`gddr6_6_dc1_clk`	East 2 DCI	1

Subsystem	Interface Name	Physical Location [1]	GDDR6 Channel
DDR4	`ddr4_noc0_clk`	South NoC	–
	`ddr4_dci0_clk`	South DCI	–
DDR5	`ddr5_noc0_clk` [4]	South NoC	–
PCIe	`pciex16_clk` [3]	Gen5 PCIe ×16	–
	`pciex16_dc_clk`	Gen5 PCIe ×16 DCI	–
	`pciex8_clk`	Gen5 PCIe ×8	–
Ethernet	`ethernet_ref_clk` [3]	Ethernet reference clock [2]	–
	`ethernet_ff0_clk` [3]	Ethernet FIFO 0 clock [2]	–
	`ethernet_ff1_clk` [3]	Ethernet FIFO 1 clock [2]	–
Configuration	`cfg_clk`	System wide configuration clock	–

Table Notes

1. Physical orientation west-to-east is with regards to viewing the die in floorplan view within ACE. The die is actually rotated about its vertical axis when packaged. Therefore, an interface shown on the floorplan, and listed in this table, 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 this table, 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.
3. Present in the AC7t800 DSM.
4. Only present in the Speedster7t AC7t800 DSM.

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 following code snippet.

```
// -------------------------
// 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
        `ACX_DEVICE_NAME.fcu.configure( "ethernet_cfg.txt", "ethernet0" );
        `ACX_DEVICE_NAME.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 (page 35) 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 in the previous code snippet, and when the Chip Status Output 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 13 • Configuration Subsystem Names

Subsystem (4)	Interface Subsystem Name (1)	Physical Location (3)
GDDR6	`gddr6_0`	West 0
	`gddr6_1`	West 1
	`gddr6_2`	West 2
	`gddr6_3`	West 3
	`gddr6_4`	East 0
	`gddr6_5`	East 1
	`gddr6_6`	East 2
	`gddr6_7`	East 3
DDR4	`ddr4`	South

Subsystem [4]	Interface Subsystem Name [1]	Physical Location [3]
DDR5	`ddr5`	South
Ethernet	`ethernet0`	North
	`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. Refer to Configuration File Format (page 45) for details.
3. Physical orientation west-to-east is with regards to viewing the die in floorplan view within ACE. The die is actually rotated about its vertical axis when packaged. Therefore, an interface shown on the floorplan, and listed in this table, 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.
4. Not all subsystems are available in all devices. Please refer to your specific device datasheet for details of available subsystems.

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`), 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 2D 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)[7] 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.

7 https://www.achronix.com/documentation/speedster7t-2d-network-chip-user-guide-ug089

SystemVerilog Interfaces

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

Note

The following interface 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                      aclk;      // 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 initiator (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 responder (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);


   modport monitor (input    awready, bresp, bvalid, bid, wready, arready, rdata, rlast,
rresp, rvalid, rid,
                             awaddr, awlen, awid, awqos, awburst, awlock, awsize,
awvalid, awregion,  awprot, awcache,
                             bready, rready, wstrb, wvalid, wdata, wlast,
                             araddr, arlen, arid, arqos, arburst, arlock, arsize,
arvalid, arregion, arprot, arcache);
endinterface : t_ACX_AXI4
```

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 path, including the base directory, of the device files within the DSM package.

When installed in ACE integration mode, the following setting should be used (with the Speedster7t AC7t1500 FPGA as an example):

```
ACX_DEVICE_INSTALL_DIR = $ACE_INSTALL_DIR/system/data/AC7t1500
```

When installed as standalone, the following setting should be used, (with the Speedster7t AC7t1500 FPGA as an example):

```
ACX_DEVICE_INSTALL_DIR = <location of standalone package>/system/data/AC7t1500
```

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 (with the Speedster7t AC7t1500 FPGA as an example):

```
> make ACX_DEVICE_INSTALL_DIR=<location of standalone package>/system/data/
AC7t1500
```

Chapter 8 : Revision History

Version	Date	Description
1.0	07 Oct 2022	• Initial Achronix release.
1.1	18 Oct 2023	• Remove references to end-of-life devices.
1.2	11 Dec 2025	• Include SerDes rate monitor (SRM) simulation with device simulation model (DSM) binding