# Speedster7t 2D NoC Reference Design Guide (RD022)

Achronix Data Acceleration

**January 28, 2022** **Reference Design**

## Introduction

The Speedster®7t 2D NoC reference design suite introduces the Achronix two-dimensional network on chip (2D NoC). Specifically, the designs demonstrate how to instantiate the different types of 2D NoC access points (NAPs) in the FPGA fabric, and how they communicate between each other. The designs also show how different modules in the fabric can reside on opposite sides of the device and communicate easily with each other using the 2D NoC while achieving an overall 500 MHz target frequency.

There are two designs within this reference design suite. The Speedster7t AC7t1500 is focused on the different forms of NAP-to-NAP communication that are available within the FPGA fabric and are intended as an example showing how to incorporate similar connections in a user design.

The 2D NoC with Cryptographic Engine design incorporates the integrated fixed soft IP Cryptographic engine core. The horizontal NAP-to-NAP communication passes through this core, forming both an encryption and a decryption path. This design demonstrates use of the Cryptographic engine alongside both NAP-to-NAP vertical and horizontal communication. This reference design is intended for users requiring the Cryptographic engine for encryption or decryption purposes and is targeted to the AC7t1550 device.

> **Note**
>
> For clarity, the Cryptographic engine core is not required for use with either the NoC, or NAPs specifically. It is demonstrated within the context of these reference designs to illustrate how it may be closely integrated with other functions available within the device. However, both NAPs and Cryptographic engine core can be used independently of each other.

Both reference designs include the following for both the Speedster7t AC7t1500 and AC7t1550 devices:

- A testbench for simulation
- ACE and Synplify projects
- Batch scripts to implement the design

Full details of both designs are available as shown:

- Speedster7t AC7t1500 (see page 2) – features horizontal, vertical and AXI NAP-to-NAP communication. This design is targeted at the VectorPath® S7t-VG6 accelerator card, and is provided with a bitstream and scripts to run the design on the card.
- 2D NoC with Cryptographic Engine (see page 19) – features horizontal NAP-to-NAP passing encrypted data combined with vertical and AXI NAP-to-NAP communication.

# AC7t1500 Design

## Description

The AC7t1500 2D NoC reference design demonstrates two types of transactions on the 2D NoC:

1. Data streaming transactions.
2. AXI transactions.

Both portions of the design use a random data generator and checker for the data portion of the transactions.

> **Note**
>
> The AC7t1500 2D NoC reference design is only available in the full-chip, bus-functional model (BFM) simulation flow, and is not available for the standalone simulation flow.

### Data Streaming Transactions

Data streaming NAPs are used for fabric-to-fabric communication and act like a FIFO to push data across the device. Within the FPGA fabric, data streaming provides the lowest latency option to drive large amounts of data long distances across the 2D NoC. This portion of the reference design uses two horizontal NAPs along a single row of the 2D NoC, and two vertical NAPs along a single column of the 2D NoC to send data from one point to the other. For further details on data streaming transactions, refer to the *Speedster7t Network on Chip User Guide* (UG089). The figure below shows the location of the NAPs used and how the data moves.

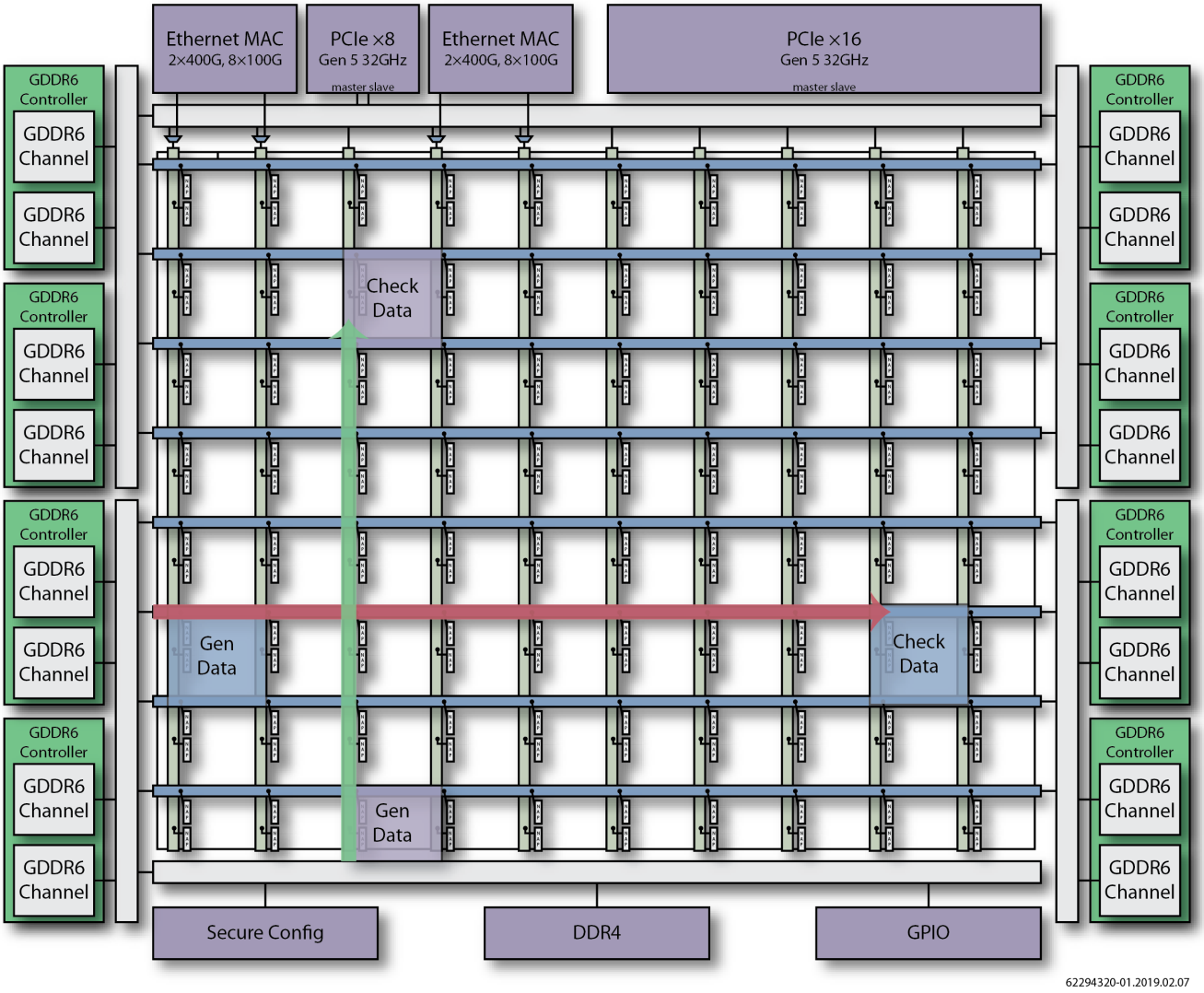


**Figure 1:** ***Data Streaming NAPs***

## Data Streaming Design Details

The data streaming portion of the design uses a data generator module that creates 288-bit data for the 2D NoC row and 293-bit data for the 2D NoC column, and pushes that data to a NAP using a simple hand-shake protocol for data streaming transactions. The checker on the opposite end of the device expects correspondingly sized data using the same pattern and checks against the data it receives from a NAP. Here, the checker handles the simple hand-shake protocol of reading a data streaming transaction.

One set of generator/checker modules is connected to horizontal NAPs, while the other set is connected to vertical NAPs. The horizontal data generator is connected to a NAP located at row 3, column 1, which then pushes data onto the 2D NoC and sends the transaction along the 2D NoC row to the NAP located on row 3, column 9. The vertical data generator is connected to a NAP located at row 1, column 3, which then pushes data on the 2D NoC and sends its transaction along the 2D NoC column to the NAP located on row 7, column 3.

The reference design sets the location of the NAPs using `bind` statements in the testbench for simulation. For implementation, the NAP placements are set in the `ace_placements.pdc` file. To change the location of the NAPs, the only requirement is that the horizontal NAPs must be located on the same row, and the vertical NAPs must be located on the same column. It is then necessary to change the location for the NAPs in the `bind` statements of the testbench for simulation, and similarly change the location of the NAPs in the `ace_placements.pdc` file for implementation.

## AXI Transactions

This portion of the reference design uses master logic connected to an AXI slave NAP that sends writes and reads to slave logic connected to an AXI master NAP on the opposite corner of the device. In this design, the master logic in the FPGA fabric generates random data along with incrementing addresses and increasing burst lengths to write to a BRAM located in the slave logic on the other side of the device. On a read, the BRAM returns the data that was previously written. Using the 2D NoC, the design is able to communicate over long distance and still maintain a target frequency of 500 MHz.

For further details on AXI transactions on the NoC, refer to the *Speedster7t Network on Chip User Guide* (UG089) . The figure below shows the location of the NAPs used and how the data moves between the master and slave logic.

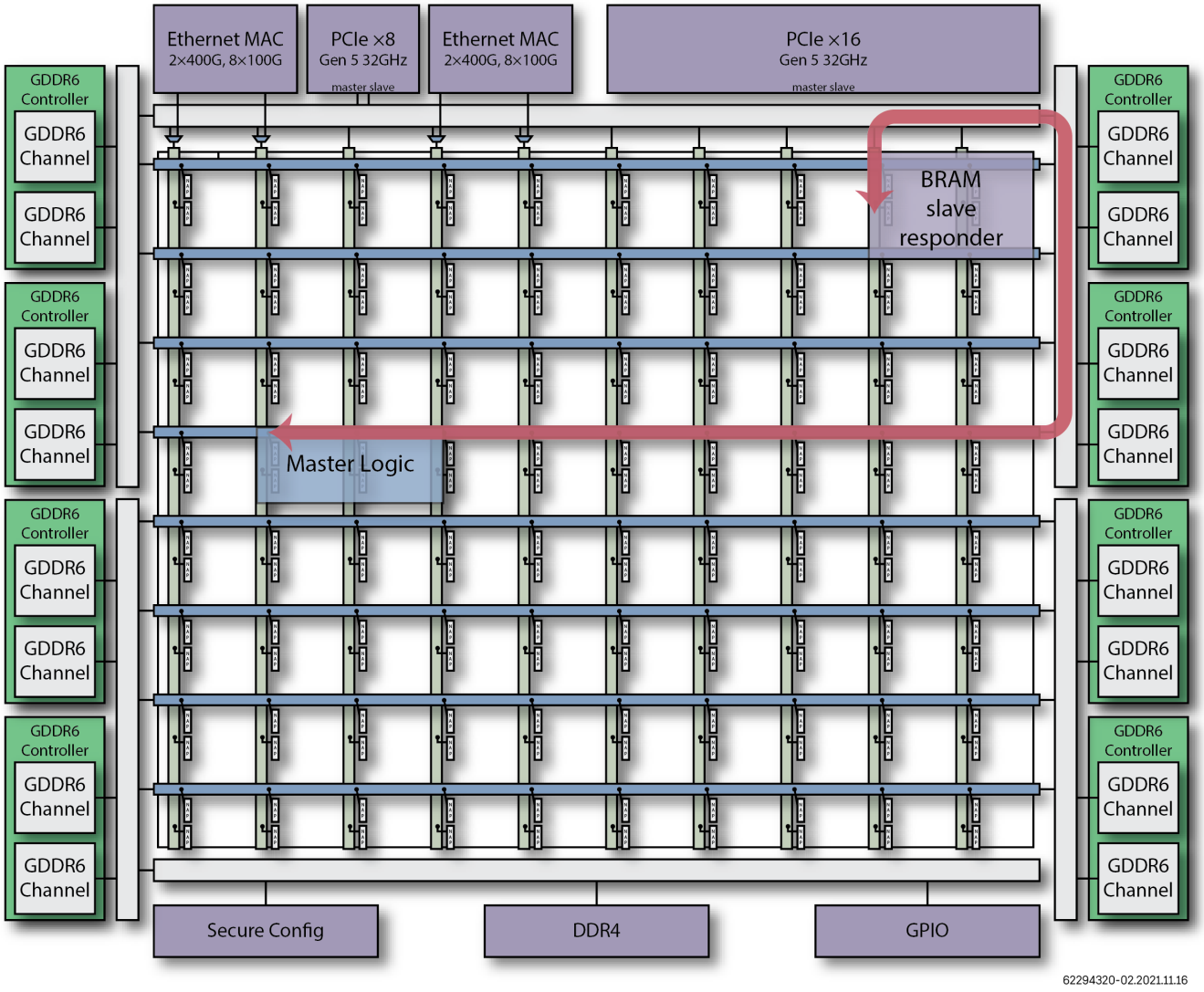


**Figure 2:** ***AXI Master and Slave NAPs***

## AXI Design Details

The AXI transaction portion of the design uses a data generator module that creates a pattern of 256-bit data along with incrementing address and burst lengths and sends the AXI write transaction to an AXI slave NAP located at row 5, column 2. The AXI transaction travels on the 2D NoC to the destination AXI master NAP, located at row 8, column 9 that connects to a slave BRAM responder. When the slave module writes the data for the transaction in the BRAM, the slave logic sends the AXI write response back across the 2D NoC to the initiating NAP.

The data checker module is also connected to the AXI slave NAP located at row 5, column 2. This module expects the same pattern of 256-bit data. A small FIFO is used between the generator and checker to pass along the address and burst length for the associated read transaction. The checker then sends out the read AXI command to the NAP which travels on the 2D NoC to the AXI master NAP located at the BRAM responder logic. The slave logic accepts the read transaction, reads the address from the BRAM, and returns the read data using an AXI read response. The read response is received at the checker AXI slave NAP, and compared against the expected data.

The design sets the location of the NAPs using `bind` statements in the testbench for simulation. For implementation, the NAP placements are set in the `ace_placements.pdc` file. To change the location of the NAPs, it is only necessary to change the location for the NAPs in the `bind` statements of the testbench for simulation, and similarly change the location of the NAPs in the `ace_placements.pdc` file for implementation.

### Organization

The reference design makes use of System Verilog interfaces to simplify the signals for each NAP. There are both AXI (`t_AXI4`) and data streaming (`t_DATA_STREAM`) interfaces included, as well as NAP wrapper files that instantiate the associated NAP. The NAP wrapper files also make use of the associated AXI or data streaming interface definitions. This structure presents an example of how NAPs can be used in a design. If System Verilog interfaces are not preferred, simply instantiate the NAP macros directly without the wrapper or interface definition.

## Ports and Parameters

### Parameters

**Table 1:** ***AC7t1500 2D NoC Reference Design Parameters***

| Parameter Name | Default | Description |
|---|---|---|
| LINEAR_PKTS_DS | 0 | Enable all data streaming generators, (horizontal and vertical), to send linear packets:<br>• 0 – all data streaming generators send randomized packets<br>• 1 – all data streaming generators send linear packets, each packet being an incrementing count from the previous |
| LINEAR_PKTS_AXI | 0 | Enable AXI packet generator to send linear packets:<br>• 0 – all AXI packets are randomized<br>• 1 – all AXI packets are an incrementing count from the previous |
| LINEAR_ADDR_AXI | 1 | Enable AXI packet generator to send to linear addresses:<br>• 0 – all AXI packets addresses are randomized<br>• 1 – all AXI packets addresses are an incrementing count from the previous |

### Ports

Port names starting with `i_` indicate input ports while those starting with `o_` are output ports.

**Table 2:** ***AC7t1500 2D NoC Reference Design Ports***

| Signal Name | Width | Direction | Description |
|---|---|---|---|
| i_send_clk | 1 | In | Clock for logic sending the data for vertical and horizontal NAPs. Connected to the logic of the AXI slave NAP that initiates reads and writes. |
| i_chk_clk | 1 | In | Clock for all logic checking the data for vertical and horizontal NAPs. Connected to the AXI master NAP that responds to reads and writes. |
| i_reg_clk | 1 | In | Clock for driving the register control block. |
| pll_send_clk_lock | 1 | In | PLL lock signal for i_send_clk, i_reg_clk and i_cc_clk. |
| pll_chk_clk_lock | 1 | In | PLL lock signal for i_chk_clk. |
| led_l0 | 1 | Out | VectorPath LED 0. Asserts at start of sending/checking transactions on AXI slave NAP. |
| led_l0_oe | 1 | Out | Output enable for led_l0; active high. |
| led_l1 | 1 | Out | VectorPath LED 1. Asserts at start of sending/checking transactions on horizontal/vertical NAP. |
| led_l1_oe | 1 | Out | Output enable for led_l1; active high. |
| led_l2 | 1 | Out | VectorPath LED 2. Asserted when the read data received on the AXI slave NAP does not match the expected data. |
| led_l2_oe | 1 | Out | Output enable for led_l2; active high. |
| led_l3 | 1 | Out | VectorPath LED 3. Asserted when the data received on the vertical NAP does not match the expected data. |
| led_l3_oe | 1 | Out | Output enable for led_l3; active high. |
| led_l6 | 1 | Out | VectorPath LED 6. Asserted when the data received on the horizontal NAP does not match the expected data. |
| led_l6_oe | 1 | Out | Output enable for led_l6; active high. |
| led_l7 | 1 | Out | VectorPath LED 7. Asserted when the total requested number of AXI transactions have been completed. |
| led_l7_oe | 1 | Out | Output enable for led_l7. Active high. |

## Implementation

The reference design includes an ACE GUI project located in the directory /ac7t1500/src/ace, and a Synplify Pro synthesis GUI project located in /ac7t1500/src/syn. The ACE project configures the 2D NoC and all clocks using IP configuration files located in /ac7t1500/src/acxip.

## 2D NoC

The 2D NoC is configured with a reference clock, (`noc_clk`), using the configuration file `/ac7t1500/src/axcip/2d_noc.acxip`. The 2D NoC core frequency is set to 1.5GHz. This core clock frequency may be adjusted to balance between required throughput and power consumption.

## Clocks

The following clocks are specified within the project using the `/acxip/pll_send_clk.acxip` and `/acxip/pll_chk_clk.acxip` files. The PLLs are respectively driven by the `fpga_fab_clk_7` and `fpga_fab_clk_6` reference clocks which are inputs on the VectorPath card.

There are three clocks in the design. Two clocks demonstrate that logic connected to different NAPs do not need to operate on the same clock domain even when communicating with each other. Although the NAP receiving clock is fractionally slower than the NAP transmitting clock, the small difference, and the buffering built into the 2D NoC ensure data integrity in the transfer. It is anticipated that in a user design the two NAPs would normally reside on the same clock.

The third clock is the register control block clock, which is asynchronous to all the data streaming and AXI blocks.

**Table 3: *Clock Frequencies***

| Name | Frequency (MHz) | Description |
|---|---|---|
| **pll_send_clk** | | |
| `fpga_fab_clk_7` | 100 | External reference input clock. |
| `noc_clk` | 200 | Default clock for the 2D NoC. (1) |
| `i_send_clk` | 500 | Fabric clock to sending logic. |
| `i_reg_clk` | 200 | Register control block clock. |
| **pll_chk_clk** | | |
| `fpga_fab_clk_6` | 50 | External reference input clock. |
| `i_chk_clk` | 499 | Fabric clock to checking logic. |

**Table Notes**

1. Clock output connects directly from the reference PLL to respective destinations. The clock is not routed via the FPGA fabric and, therefore, is not available to the user logic.

## Resets

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

**Table 4: *Reset Sources***

| Source | Description |
|---|---|
| `pll_send_clk_lock` | User clock PLL locked. Provides `i_send_clk`, `noc_clk` and `i_reg_clk`. |
| `pll_chk_clk_lock` | User clock PLL locked. Provides `i_chk_clk`. |
| `reg_rstn` | Internally generated synchronous reset for the register control block. |

## Control and Status

The register control block creates a set of control and status registers within the user design. It uses a ACX_NAP_AXI_MASTER to read and write the registers. The NAP can be accessed via either the PCIe subsystem, or via the device JTAG port. The registers allow the user design to control functionality, and to measure results and performance, both in simulation and at runtime. Further, the block contains four fixed registers, Major, Minor, Patch and Revision Control, which can be used for versioning and traceability across multiple builds.

Within the reference design, the register control block is used to start and stop the tests and to ensure that all tests have passed. In addition, where applicable, traffic monitors are read to measure the throughput and latency of targeted data paths, particularly those from the hardened interface subsystems.

### Header Files

The `reg_control_block.sv` file requires two header files, which are included in the `/src/include` directory:

**Table 5: *Register Control Block Include Files***

| Filename | Description |
|---|---|
| `reg_control_defines.svh` | Defines the type `t_ACX_USER_REGS` as a 32-bit logic vector. |
| `version_defines.svh` | Defines the values of the four fixed-version registers. These values can be used for versioning of the user design. They may be changed on each build to reflect different releases. |

**Table Note**

- The registers are set to a width of 32-bits, with addressing of registers on a 4-byte boundary. If registers wider than 32-bits are required, use two registers in parallel or modify the `reg_control_block.sv` file to accommodate the new width and greater address span between registers.

## Parameters

**Table 6:** ***Register Control Block Parameters***

| Parameter | Supported Values | Default Value | Description |
|---|---|---|---|
| NUM_USER_REGS | 2–1024 | 2 | Number of read and write registers. |
| IN_REGS_PIPE | 0–4 | 0 | Pipeline added to input registers. |
| OUT_REGS_PIPE | 0–4 | 0 | Pipeline added to output registers. |

**Table Note**

- There is no actual limit to the maximum values shown for all supported values. Higher values may be used, however considerable fabric resources might be consumed.

## Ports

**Table 7: *Register Control Block Ports***

| Name | Direction | Description |
|---|---|---|
| i_clk | Input | Clock. |
| i_reset_n | Input | Negative synchronous reset. |
| i_user_regs_in | Input | Array of NUM_USER_REGS of type t_ACX_USER_REGS. |
| o_user_regs_out | Output | Array of NUM_USER_REGS of type t_ACX_USER_REG. |

**Table Note**

- The input and output registers are provided separately as Read-Only and Write-Only registers. In order to create a Read-Write register, assign the respective i_user_regs_in to the corresponding o_user_regs_out.

## Instantiation

An example instantiation of the register control block is shown below.

```
// Include definitions of register types
`include "reg_control_defines.svh"


// Define number of registers
localparam      NUM_USER_REGS = (REGS_PER_ETH_CH*MAX_ETH_CHANNELS)+9;


// Define the registers
t_ACX_USER_REG  user_regs_write [NUM_USER_REGS -1:0];
t_ACX_USER_REG  user_regs_read  [NUM_USER_REGS -1:0];

// Instantiate the register control block
reg_control_block  #(
    .NUM_USER_REGS      (NUM_USER_REGS),    // Number of user registers
    .IN_REGS_PIPE       (2),
    .OUT_REGS_PIPE      (1)
) i_reg_control_block (
    .i_clk              (i_reg_clk),
    .i_reset_n          (reg_rstn),
    .i_user_regs_in     (user_regs_read),
    .o_user_regs_out    (user_regs_write)
);

//------------------------------------------------------------------------
// Make top register a scratch register, looping back on itself
//------------------------------------------------------------------------
assign user_regs_read[NUM_USER_REGS-1] = user_regs_write[NUM_USER_REGS-1];
```

## Addressing

Within the SystemVerilog code, each register occupies a single index. Consecutive registers can be addressed by incrementing the index by 1. However, when addressing each register from the NAP, as each register is 4-bytes wide, consecutive registers are at 4-byte boundary addresses and are 4-bytes apart. An example of register address definition is shown below.

```
// Register definition in SystemVerilog
localparam CONTROL_REG_ADDR          = 0;
localparam STATUS_REG_ADDR           = 1;
localparam NUM_PKTS_TX_REG_ADDR      = 2;
localparam SCRATCH_REG_ADDR          = NUM_USER_REGS-1;

# Same registers defined in demo Tcl script, (formatted to hex)
set CONTROL_REG_ADDR          [format %X [expr {0 * 4}]]
set STATUS_REG_ADDR           [format %X [expr {1 * 4}]]
set NUM_PKTS_TX_REG_ADDR      [format %X [expr {2 * 4}]]
set SCRATCH_REG_ADDR          [format %X [expr {($NUM_USER_REGS-1) * 4}]]
```

The the register control block does not support partial writes to a register, it requires all 32-bits to be written in each transaction.

## Demo Tcl Script

As stated, the register control block can be used in both simulation and for runtime. In both cases, the sequence to drive the register control block is defined by the demo Tcl script located in `/demo/scripts/<design_name>_demo.tcl`. Use of the same source script ensures the sequence of operations in simulation is consistent with that at runtime on hardware. For details of the demo Tcl script format, refer to Runtime Programming Scripts.

## Access Functions

The functions to read and write from the ACX_NAP_AXI_MASTER are provided within the ACE runtime register library and are documented in Runtime Programming Scripts. The three specific functions are detailed below.

**Table 8: *Register Control Block Access Funtions***

| Function | Arguments (1) | Description |
|---|---|---|
| `<device_namespace>::nap_axi_write` | NAP_SPACE row col address value | Write a value to a register. |
| `<device_namespace>::nap_axi_read` | NAP_SPACE row col address | Read a value from a register. Returns a hex value in the range `32'h0`–`32'hffff_ffff`. |
| `<device_namespace>::nap_axi_verify` | NAP_SPACE row col address exp_value | Verify a value from a register. If the verify value is incorrect:<br>• In simulation – the FCU BFM asserts an error flag, which can be used in the testbench to indicate test failure.<br>• In hardware – the verify function returns an error code (–1) which can be used to issue a fail message. |

**Table Notes**

1. Argument values:
   row – decimal value, range 1–8. Must match the values in the testbench `ACX_BIND` statement and `/src/constraints/ace_placements.pdc`.
   col – decimal value range 1–10. Must match the values in the testbench `ACX_BIND` statement and `/src/constraints/ace_placements.pdc`.
   address – hex value, range 0–(4 × (NUM_USER_REGS – 1)). This is the register address, not NAP address.
   value – hex value, range `32'h0`–`32'hffff_ffff`. May have `0x` prefix.
   exp_value – hex value, range `32'h0`–`32'hffff_ffff`. Must not have `0x` prefix.

## Simulation

The simulation `Makefile` processes the demo Tcl script with ACE to create a Simulation Command File (.txt). This file is written to the `/sim` directory. The simulation command file has the same format as the "Configuration File Format" shown in Device Simulation Model.

The testbench reads this file after the DSM has entered user mode, and applies the sequence via the DSM FCU to the ACX_NAP_AXI_MASTER, and hence to the register array. This flow is shown in the diagram below.

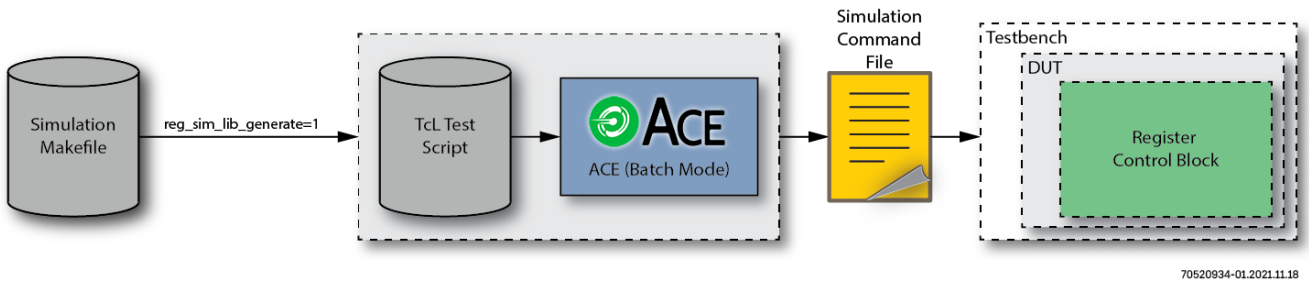


**Figure 3: *Register Control Block Simulation Flow***

For further details on the simulation flow, please refer to Simulating the Reference Design.

## Runtime JTAG

In runtime, the demo Tcl script is read in the ACE Tcl console window, and is applied directly to the JTAG port of the connected device. In runtime both the Tcl command `puts` and the ACE built-in command `message` can be used to provide status as the script is run.

The runtime flow is shown in the diagram below.

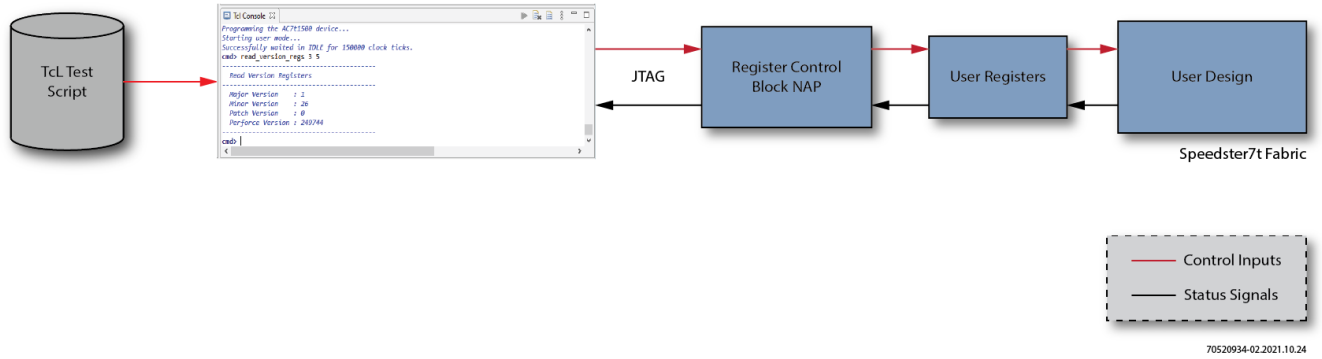


**Figure 4:** ***Register Control Block Runtime Flow***

## Runtime PCIe

The ACX_NAP_AXI_MASTER used by the register control block is accessible from any of the programming methods into the device. Therefore the register control block can be accessed and controlled via any of the devices PCIe ports. For details of PCIe drivers please contact Achronix support.

## Register Definitions

The reference design has four user registers `CONTROL_REG`, `STATUS_REG`, `NUM_TRANSACTIONS_REG` and `SCRATCH_REG`, which are described below. The `register_control_block` is located in col = 5, row = 5. This location may be changed to suit the user design.

The placement of register control ACX_NAP_AXI_MASTER is specified in both the `/src/tb /tb_2d_noc_ref_design.sv` and `/src/constraints/ace_placements.pdc` files.

**Table 9:** ***CONTROL_REG (Offset : 0x00_0000)***

| Name | Bits | Access | Default | Description |
|---|---|---|---|---|
| `start_axi` | 0 | RW | `1'b0` | Assert to start sending transactions on the AXI Slave NAP. |
| `start_h_send` | 1 | RW | `1'b0` | Start sending transactions on the horizontal NAP. |
| `start_v_send` | 2 | RW | `1'b0` | Start sending transactions on the vertical NAP. |
| `start_h_chk` | 3 | RW | `1'b0` | Start checking transactions on the horizontal NAP. |
| `start_v_chk` | 4 | RW | `1'b0` | Start checking transactions on the vertical NAP. |
| `nap_chk_rstn` | 5 | RW | `1'b0` | Active low reset input for block checking transactions. |
| `nap_send_rstn` | 6 | RW | `1'b0` | Active low reset input for block generating transactions. |

**Table 10: *STATUS_REG (Offset : 0x00_0004)***

| Name | Bits | Access | Default | Description |
|---|---|---|---|---|
| pll_chk_clk_lock_reg | 0 | RO | 1'b0 | PLL lock signal for i_chk_clk. |
| pll_send_clk_lock_reg | 1 | RO | 1'b0 | PLL lock signal for i_send_clk. |
| xact_done | 2 | RO | 1'b0 | Asserted when the total requested number of AXI transactions have been completed. |
| fail_row | 3 | RO | 1'b0 | Asserted when data received on the horizontal NAP does not match the expected data. |
| fail_col | 4 | RO | 1'b0 | Asserted when data received on the vertical NAP does not match the expected data. |
| fail_axi | 5 | RO | 1'b0 | Output enable for o_fail_col, tied to 1'b1. |

**Table 11: *NUM_TRANSACTIONS_REG (Offset : 0x00_0008)***

| Name | Bits | Access | Default | Description |
|---|---|---|---|---|
| num_transactions | [12:0] | RW | 13'b0 | The number of transactions to be sent by the AXI packet generator. This value is used to control the overall test and to indicate xact_done when all transactions are complete.<br>If the value is set to 13'h0, all tests run continuously and xact_done is not asserted. |
| Reserved | [31:13] | RW | 19'b0 | Unused. Must be set to 19'b0. |

**Table 12: *SCRATCH_REG (Offset : 0x00_000C)***

| Name | Bits | Access | Default | Description |
|---|---|---|---|---|
| scratch | [31:0] | RW | 32'b0 | Scratch Register to test read and write access. |

## Script and Command File

For this design, the demo Tcl script file is /demo/scripts/ac7t1500_2D_NoC_demo.tcl , and the generated simulation command file is /sim/ac7t1500_2D_NoC_sim.txt.

# Design Considerations

Within the design there are a number of considerations that must be taken into account.

## VectorPath Card

This design is compatible with the Achronix VectorPath® S7t-VG6 accelerator card, revision 1 onwards. The provided bitstream, in `/demo/bitstream`, can be downloaded to the VectorPath card, and the test executed using the script in `/demo/scripts`. In addition, the design can be rebuilt to run on the card.

The design includes the IP configuration files, (`.acxip`), in `/src/acxip`, that specify the pinout of the VectorPath card, including clock and GPIO signals. Included files are shown in the table below.

**Table 13: *VectorPath IP Configuration Files***

| File | Ports Connected | Direction | Description |
|---|---|---|---|
| vp_clkio_ne.acxip | ext_pps | Input | |
| | fpga_rst_l | Input | Driven from BMC controller |
| | pcie_perst_l | Input | |
| | fpga_fab_clk_2 | Input | 400MHz differential |
| vp_clkio_nw.acxip | fab_refclk_0 | Output | 100MHz |
| | fpga_fab_clk_3 | Input | 800MHz differential |
| | fpga_fab_clk_4 | Input | 10MHz differential |
| vp_clkio_se.acxip | fpga_fab_clk_5 | Input | 500MHz differential |
| | fpga_fab_clk_6 | Input | 50MHz differential |
| vp_clkio_sw.acxip | fpga_fab_clk_7 | Input | 100MHz differential |
| | fpga_fab_clk_1 | Input | 200MHz differential |
| vp_gpio_n_b0.acxip | exp_gpio_fpga[7:0] | Inout | General purpose IO |
| | ext_gpio_oe_l | Output | GPIO output enable |
| | led_oe_l | Output | Enable LED outputs |
| vp_gpio_n_b1.acxip | ext_gpio_dir[7:0] | Output | GPIO signal direction |
| | led_l[5:4] | Output | Board LEDs. Active low |
| vp_gpio_n_b2.acxip | led_l[7:6, 3:0] | Output | Board LEDs. Active low |

**Table 14: *VectorPath IP Configuration Files (cont.)***

| File | Ports Connected | Direction | Description |
|---|---|---|---|
| vp_gpio_s_b0.acxip | fpga_i2c_mux_gnt | Input | I2C interface |
| | fpga_i2c_req_l | Output | I2C interface. Active low |
| | fpga_avr_txd | Output | AVR interface |
| | fpga_avr_rxd | Input | AVR interface |
| | fpga_ftdi_txd | Output | FTDI interface |
| | fpga_ftdi_rxd | Input | FTDI interface |
| | irq_to_fpga | Input | Interrupt |
| | irq_to_avr | Output | Interrupt |
| | test[1] | Output | Test interface |
| | qsfp_int_fpga_l | Input | QSFP interface. Active low |
| vp_gpio_s_b1.acxip | mcio_vio[3:0] | Inout | MCIO interface |
| | mcio_dir[3:0] | Output | MCIO interface |
| | mcio_dir_45 | Output | MCIO interface |
| | test[2] | Output | Test interface |
| vp_gpio_s_b2.acxip | mcio_scl | Inout | MCIO interface |
| | mcio_sda | Inout | MCIO interface |
| | mcio_oe2_l | Output | MCIO interface |
| | mcio_oe_45_1 | Output | MCIO interface |
| | fpga_sys_scl | Inout | I2C interface |
| | fpga_sys_sda | Inout | I2C interface |
| vp_pll_nw_2.acxip | pll_nw_2_ref0_312p5_clk | Output | 312.5MHz clock |
| vp_pll_sw_2.acxip | pll_sw_2_ref0_312p5_clk | Output | 312.5MHz clock |

**Table Note**

- If porting a user design to the VectorPath card, it is recommended to use these files to define the interface to the card.

# 2D NoC with Cryptographic Engine

## Introduction

The design for the Speedster7t AC7t1550 device shares many features with the AC7t1500 design. However, the reference design also includes the AC7t1550 cryptographic engine performing data encryption and decryption, and passing the encrypted traffic through the 2D NoC.

Although the cryptographic engine core is implemented in the fabric, it is considered a hard IP and cannot therefore be modified.

## References

For full details on the cryptographic engine, please refer to *Speedster7t Cryptographic Engine User Guide* (UG104).

For full details on the NoC operation please refer to *Speedster7t Network on Chip User Guide* (UG089).

For full details on the various NAP types, please refer to *Speedster7t Component Library User Guide* (UG086).

## Description

The 2D NoC with cryptographic engine design presents two types of transactions on the 2D NoC. One demonstrates data streaming transactions and the other demonstrates encrypted data being transferred from the cryptographic engine.

The transactions are divided into data streaming transactions taking place on the horizontal and vertical NAPs, and AXI transactions occurring across the 2D NoC.

### Cleartext Data Streaming

The vertical NAP data streaming transactions and the AXI transactions are the same as described in the AC7t1500 Design (see page 2).

### Encrypted Data Streaming

The portion of the reference design targeting the cryptographic engine utilizes data streaming transactions on the horizontal NAPs. The figure below shows the location of the cryptographic engine, NAPs and the data flow.

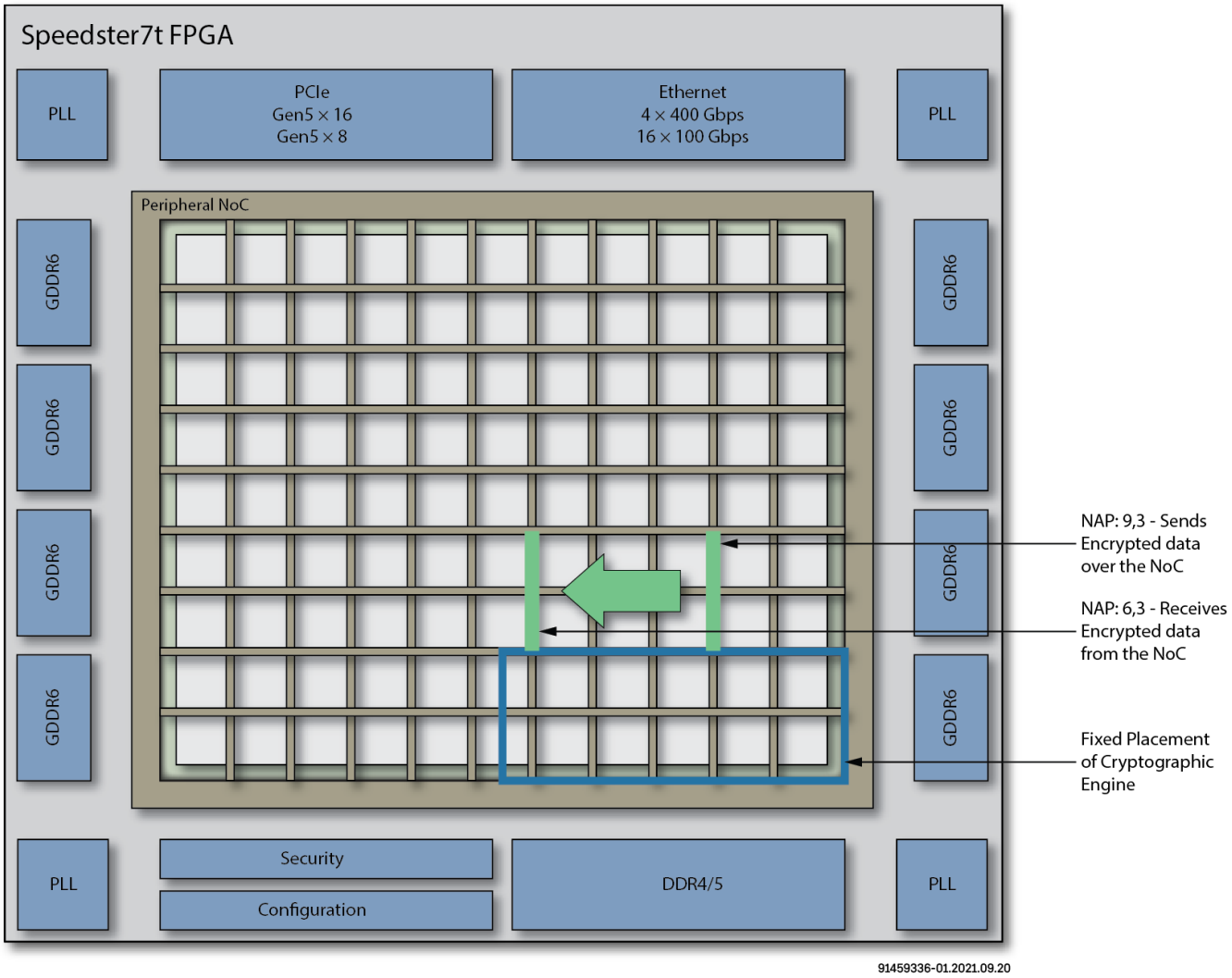


**Figure 5:** ***Encrypted Data Streaming Data Flow***

The encrypted data flow through the 2D NoC consists of the following design flow.

A data streaming generator module creates cleartext data and sends the data to the cryptographic engine. During encryption operation, the cryptographic engine encrypts the 128-bit cleartext data and sends it to the `i_nap_row_in` NAP.

This NAP sends the encrypted data, via the NoC, to the `i_nap_row_out` NAP which is located near to the data streaming generator. The data from the NAP is stored in a FIFO to ensure that there are no stalls on the 2D NoC. The control logic detects that the FIFO has a packet of data. This pauses the data generator, switches the cryptographic engine to decryption mode, and plays the encrypted data from the FIFO into the cryptographic engine, which decrypts the data and plays it to the data streaming checker where it is compared to the original data from the data generator.

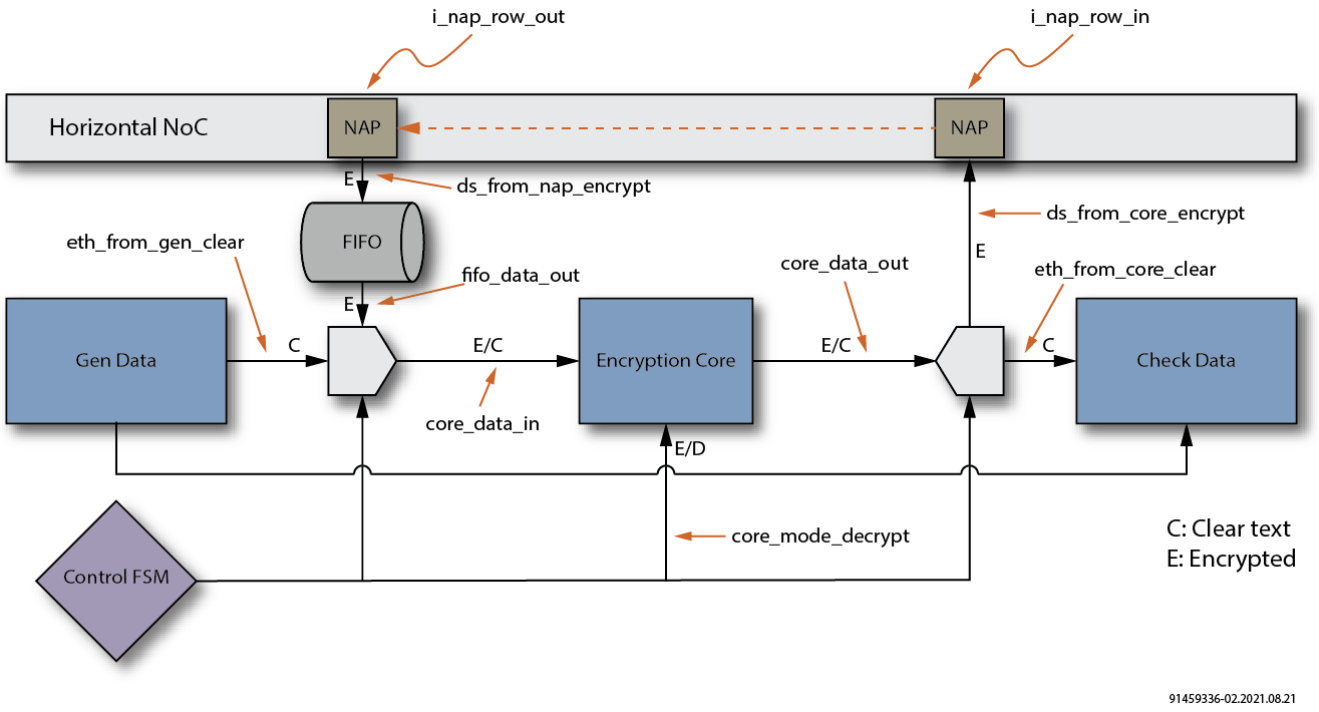


**Figure 6:** ***Cryptographic Engine Block Diagram***

## Last Word

Though the cryptographic engine's data input interface is 128 bits wide, the data can be input with byte resolution. This is achieved with the `ibyte` and `last_w` inputs. The `last_w` signal indicates the last word being input. The `ibyte` signal is ignored until `last_w` is asserted, indicating the number of valid bytes in the last word (counting from the MSB) minus 1.

So `ibyte` = "0000" means that only the first byte in the incoming word is valid and `ibyte`="1111" means that all are valid.

> **Note**
>
> This scheme is unique to the cryptographic engine and should not be confused with other schemes used throughout other reference designs where a mod value of 0 equates to all bytes being active in the last word.

## NAP positions

The NAP on the output of the cryptographic engine core is located at column 9, row 3, which then pushes the data onto the 2D NoC and sends the transaction along the 2D NoC to the NAP located on column 6, row 3.

The reference design sets the location of the NAPs using `ACX_BIND` macros in the testbench for simulation. For implementation, the NAP placements are set in the `ace_placements.pdc` file. To change the location of the NAPs, the requirements are that the horizontal NAPs must be located on the same row, and the vertical NAPs must be located on the same column. NAPs should not be placed into the South East quadrant of the fabric as this is where the cryptographic engine is located. In addition, the transmitting NAP (`i_nap_row_in`) must set the column address of the receiving NAP (`i_nap_row_out`).

The following code shows where these locations and addresses are set in the design source, testbench and constraints:

- In file `noc_2d_ref_design_top.sv`:

```
// Set the address from this NAP to the destination nap on the same row, (column 6)
// This column must match the same assignments in both the testbench and the ace_placements.
pdc
assign ds_from_core_encrypt.addr = 4'h6;
```

- In file `tb_2d_noc_ref_design.sv`:

```
// ----------------------------------------
// horizontal NAP at col=9, row=3
`ACX_BIND_NAP_HORIZONTAL(DUT.i_nap_row_in.i_nap_horizontal,9,3);
// horizontal NAP at col=6, row=3
`ACX_BIND_NAP_HORIZONTAL(DUT.i_nap_row_out.i_nap_horizontal,6,3);
```

- In file `ace_placements.pdc`:

```
# col=9 row=3
set_placement -fixed {i:i_nap_row_in.i_nap_horizontal}  {s:x_core.NOC[9][3].logic.noc.nap_s}
# col=6 row=3
set_placement -fixed {i:i_nap_row_out.i_nap_horizontal}  {s:x_core.NOC[6][3].logic.noc.
nap_s}
```

## NAP Interfaces

The reference design makes use of System Verilog interfaces to simplify the signals for each NAP. There are data streaming interfaces (`t_DATA_STREAM`), as well as NAP wrapper files that instantiate the associated NAP. The NAP wrapper files also make use of the associated AXI or data streaming interface definitions. This structure presents an example of how NAPs can be used in a design. If System Verilog interfaces are not required, simply instantiate the NAP macros directly without the wrapper or interface definition.

## Test Sequence

The test is controlled by the register control block as described in Control and Status. Using the demo Tcl script (in both simulation and hardware) the register control block sets the desired number of transactions on the AXI transfers, then initiates transfers on the horizontal (cryptographic engine data), vertical (data streaming) and AXI transactions. When the AXI has completed the required number of transfers, then test is complete.

# Ports and Parameters

## Parameters

**Table 15: *2D NoC with Cryptographic Engine Parameters***

| Parameter Name | Default | Description |
|---|---|---|
| LINEAR_PKTS_DS | 0 | Enable all data streaming generators (horizontal and vertical) to send linear packets:<br>• 0 – all data streaming generators send randomized packets.<br>• 1 – all data streaming generators send linear packets, each packet being an incrementing count from the previous. |
| LINEAR_PKTS_AXI | 0 | Enable AXI packet generator to send linear packets:<br>• 0 – all AXI packets are randomized.<br>• 1 – all AXI packets are an incrementing count from the previous. |
| LINEAR_ADDR_AXI | 1 | Enable AXI packet generator to send to linear addresses:<br>• 0 – all AXI packets addresses are randomized.<br>• 1 – all AXI packets addresses are an incrementing count from the previous. |

## Ports

Port names starting with i_ indicate input ports while those starting with o_ are output ports.

**Table 16: *2D NoC with Cryptographic Engine Ports***

| Signal Name | Width | Direction | Description |
|---|---|---|---|
| i_send_clk | 1 | In | Clock for logic sending the data for vertical and horizontal NAPs. Clock is connected to the logic of the AXI slave NAP initiating reads and writes. |
| i_chk_clk | 1 | In | Clock for all logic checking the data for vertical and horizontal NAPs. Clock is connected to the AXI master NAP responding to reads and writes. |
| i_cc_clk | 1 | In | Clock for driving the cryptographic engine. |
| i_reg_clk | 1 | In | Clock for register control block. |
| pll_send_clk_lock | 1 | In | PLL lock signal for i_send_clk and i_cc_clk. |
| pll_chk_clk_lock | 1 | In | PLL lock signal for i_chk_clk. |
| led_l[0] | 1 | Out | VectorPath LED 0. Asserts at start of sending/checking transactions on the AXI slave NAP. |

| Signal Name | Width | Direction | Description |
|---|---|---|---|
| led_l_oe[0] | 1 | Out | Output enable for led_l[0]. Active high. |
| led_l[1] | 1 | Out | VectorPath LED 1. Asserts at start of sending/checking transactions on horizontal/vertical NAP. |
| led_l_oe[1] | 1 | Out | Output enable for led_l[1]. Active high. |
| led_l[2] | 1 | Out | VectorPath LED 2. Asserted when the read data received on the AXI slave NAP does not match the expected data. |
| led_l_oe[2] | 1 | Out | Output enable for led_l[2]. Active high. |
| led_l[3] | 1 | Out | VectorPath LED 3. Asserted when the data received on the vertical NAP does not match the expected data. |
| led_l_oe[3] | 1 | Out | Output enable for led_l[3]. Active high. |
| led_l[6] | 1 | Out | VectorPath LED 6. Asserted when the data received on the horizontal NAP does not match the expected data. |
| led_l_oe[6] | 1 | Out | Output enable for led_l[6]. Active high. |
| led_l[7] | 1 | Out | VectorPath LED 7. Asserted when the total requested number of AXI transactions have been completed. |
| led_l_oe[7] | 1 | Out | Output enable for led_l[7]. Active high. |

**Note**

Although the ports used for the design match the VectorPath® S7t-VG6 accelerator card, the accelerator card uses the AC7t1500 device, not the AC7t1550 device. Therefore it is not possible to run the design on the VectorPath card.

# Implementation

The reference design includes an ACE GUI project located in the directory `/ac7t1550/src/ace`, and a Synplify Pro synthesis GUI project located in `/ac7t1550/src/syn`. The ACE project configures the 2D NoC and all clocks using IP configuration files located in `/ac7t1550/src/acxip`.

## 2D NoC

The 2D NoC is configured with a reference clock (`noc_clk`) using the configuration file `/ac7t1550/src/axcip /2d_noc.acxip`. The 2D NoC core frequency is set to 1.5GHz. This core clock frequency may be adjusted to balance between required throughput and power consumption.

## Clocks

The following clocks are specified within the project using the `/acxip/pll_send_clk.acxip` and `/acxip /pll_chk_clk.acxip` files. The PLLs are respectively driven by the `fpga_fab_clk_7` and `fpga_fab_clk_6` reference clocks which are inputs on the VectorPath card.

In total there are four clock signals in the design. Two of the clocks, `i_send_clk` and `i_chk_clk`, demonstrate that logic connected to different NAPs does not need to operate in the same clock domain. Although the NAP receiving clock is fractionally slower than the NAP transmitting clock, the small difference, and the buffering built into the 2D NoC ensure data integrity in the transfer. It is anticipated that, in a user design, the two NAPs would normally reside on the same clock.

The clock signal `i_cc_clk` drives the cryptographic engine and the clock `i_reg_clk` drives the register control block.

**Table 17: *Design Clock Sources and Frequencies***

| Name | Frequency (MHz) | Description |
|---|---|---|
| **pll_send_clk** | | |
| `fpga_fab_clk_7` | 100 | External reference input clock. |
| `noc_clk` | 200 | Default clock for the 2D NoC.(1) |
| `i_send_clk` | 500 | Fabric clock to sending logic. |
| `i_cc_clk` | 200 | Fabric clock to cryptographic engine.(2) |
| `i_reg_clk` | 200 | Register control block clock.(2) |
| **pll_chk_clk** | | |
| `fpga_fab_clk_6` | 50 | External reference input clock. |
| `i_chk_clk` | 499 | Fabric clock to checking logic. |

**Table Notes**

1. Clock output connects directly from the reference PLL to the respective destinations. The clock is not routed via the FPGA fabric so is not available to the user logic.
2. These clocks share a common frequency, however, their usage is independent of one another. Within this design they are treated as asynchronous clocks. These clocks may be combined if required.

## Resets

There are three reset sources for the design as detailed in the table below.

**Table 18: *Reset Sources***

| Source | Description |
|---|---|
| `i_reset_n` | External asynchronous input. Pin configuration is specified in `/acxip /gpio_bank_north.acxip` |
| `pll_send_clk_lock` | PLL locked user clock. Provides `i_send_clk`, `i_reg_clk` and `i_cc_clk`. |
| `pll_chk_clk_lock` | PLL locked user clock. Provides `i_chk_clk`. |

## Control and Status

The register control block creates a set of control and status registers within the user design. It uses a ACX_NAP_AXI_MASTER to read and write the registers. The NAP can be accessed via either the PCIe subsystem, or via the device JTAG port. The registers allow the user design to control functionality, and to measure results and performance, both in simulation and at runtime. Further, the block contains four fixed registers, Major, Minor, Patch and Revision Control, which can be used for versioning and traceability across multiple builds.

Within the reference design, the register control block is used to start and stop the tests and to ensure that all tests have passed. In addition, where applicable, traffic monitors are read to measure the throughput and latency of targeted data paths, particularly those from the hardened interface subsystems.

### Header Files

The `reg_control_block.sv` file requires two header files, which are included in the `/src/include` directory:

**Table 19:** ***Register Control Block Include Files***

| Filename | Description |
|---|---|
| `reg_control_defines.svh` | Defines the type `t_ACX_USER_REGS` as a 32-bit logic vector. |
| `version_defines.svh` | Defines the values of the four fixed-version registers. These values can be used for versioning of the user design. They may be changed on each build to reflect different releases. |

**Table Note**

- The registers are set to a width of 32-bits, with addressing of registers on a 4-byte boundary. If registers wider than 32-bits are required, use two registers in parallel or modify the `reg_control_block.sv` file to accommodate the new width and greater address span between registers.

## Parameters

**Table 20:** ***Register Control Block Parameters***

| Parameter | Supported Values | Default Value | Description |
|---|---|---|---|
| NUM_USER_REGS | 2–1024 | 2 | Number of read and write registers. |
| IN_REGS_PIPE | 0–4 | 0 | Pipeline added to input registers. |
| OUT_REGS_PIPE | 0–4 | 0 | Pipeline added to output registers. |

**Table Note**

- There is no actual limit to the maximum values shown for all supported values. Higher values may be used, however considerable fabric resources might be consumed.

## Ports

**Table 21: *Register Control Block Ports***

| Name | Direction | Description |
|---|---|---|
| i_clk | Input | Clock. |
| i_reset_n | Input | Negative synchronous reset. |
| i_user_regs_in | Input | Array of NUM_USER_REGS of type t_ACX_USER_REGS. |
| o_user_regs_out | Output | Array of NUM_USER_REGS of type t_ACX_USER_REG. |

**Table Note**

- The input and output registers are provided separately as Read-Only and Write-Only registers. In order to create a Read-Write register, assign the respective i_user_regs_in to the corresponding o_user_regs_out.

## Instantiation

An example instantiation of the register control block is shown below.

```
// Include definitions of register types
`include "reg_control_defines.svh"


// Define number of registers
localparam      NUM_USER_REGS = (REGS_PER_ETH_CH*MAX_ETH_CHANNELS)+9;


// Define the registers
t_ACX_USER_REG  user_regs_write [NUM_USER_REGS -1:0];
t_ACX_USER_REG  user_regs_read  [NUM_USER_REGS -1:0];

// Instantiate the register control block
reg_control_block  #(
    .NUM_USER_REGS      (NUM_USER_REGS),    // Number of user registers
    .IN_REGS_PIPE       (2),
    .OUT_REGS_PIPE      (1)
) i_reg_control_block (
    .i_clk              (i_reg_clk),
    .i_reset_n          (reg_rstn),
    .i_user_regs_in     (user_regs_read),
    .o_user_regs_out    (user_regs_write)
);

//------------------------------------------------------------------------
// Make top register a scratch register, looping back on itself
//------------------------------------------------------------------------
assign user_regs_read[NUM_USER_REGS-1] = user_regs_write[NUM_USER_REGS-1];
```

## Addressing

Within the SystemVerilog code, each register occupies a single index. Consecutive registers can be addressed by incrementing the index by 1. However, when addressing each register from the NAP, as each register is 4-bytes wide, consecutive registers are at 4-byte boundary addresses and are 4-bytes apart. An example of register address definition is shown below.

```
// Register definition in SystemVerilog
localparam CONTROL_REG_ADDR          = 0;
localparam STATUS_REG_ADDR           = 1;
localparam NUM_PKTS_TX_REG_ADDR      = 2;
localparam SCRATCH_REG_ADDR          = NUM_USER_REGS-1;

# Same registers defined in demo Tcl script, (formatted to hex)
set CONTROL_REG_ADDR          [format %X [expr {0 * 4}]]
set STATUS_REG_ADDR           [format %X [expr {1 * 4}]]
set NUM_PKTS_TX_REG_ADDR      [format %X [expr {2 * 4}]]
set SCRATCH_REG_ADDR          [format %X [expr {($NUM_USER_REGS-1) * 4}]]
```

The the register control block does not support partial writes to a register, it requires all 32-bits to be written in each transaction.

## Demo Tcl Script

As stated, the register control block can be used in both simulation and for runtime. In both cases, the sequence to drive the register control block is defined by the demo Tcl script located in `/demo/scripts /<design_name>_demo.tcl`. Use of the same source script ensures the sequence of operations in simulation is consistent with that at runtime on hardware. For details of the demo Tcl script format, refer to Runtime Programming Scripts.

## Access Functions

The functions to read and write from the ACX_NAP_AXI_MASTER are provided within the ACE runtime register library and are documented in Runtime Programming Scripts. The three specific functions are detailed below.

**Table 22: *Register Control Block Access Funtions***

| Function | Arguments (1) | Description |
|---|---|---|
| `<device_namespace>::nap_axi_write` | NAP_SPACE row col address value | Write a value to a register. |
| `<device_namespace>::nap_axi_read` | NAP_SPACE row col address | Read a value from a register. Returns a hex value in the range `32'h0`–`32'hffff_ffff`. |
| `<device_namespace>::nap_axi_verify` | NAP_SPACE row col address exp_value | Verify a value from a register. If the verify value is incorrect:<br>• In simulation – the FCU BFM asserts an error flag, which can be used in the testbench to indicate test failure.<br>• In hardware – the verify function returns an error code (–1) which can be used to issue a fail message. |

**Table Notes**

1. Argument values:
   row – decimal value, range 1–8. Must match the values in the testbench `ACX_BIND` statement and `/src/constraints/ace_placements.pdc`.
   col – decimal value range 1–10. Must match the values in the testbench `ACX_BIND` statement and `/src/constraints/ace_placements.pdc`.
   address – hex value, range 0–(4 × (NUM_USER_REGS – 1)). This is the register address, not NAP address.
   value – hex value, range `32'h0`–`32'hffff_ffff`. May have `0x` prefix.
   exp_value – hex value, range `32'h0`–`32'hffff_ffff`. Must not have `0x` prefix.

## Simulation

The simulation `Makefile` processes the demo Tcl script with ACE to create a Simulation Command File (.txt). This file is written to the `/sim` directory. The simulation command file has the same format as the "Configuration File Format" shown in Device Simulation Model.

The testbench reads this file after the DSM has entered user mode, and applies the sequence via the DSM FCU to the ACX_NAP_AXI_MASTER, and hence to the register array. This flow is shown in the diagram below.

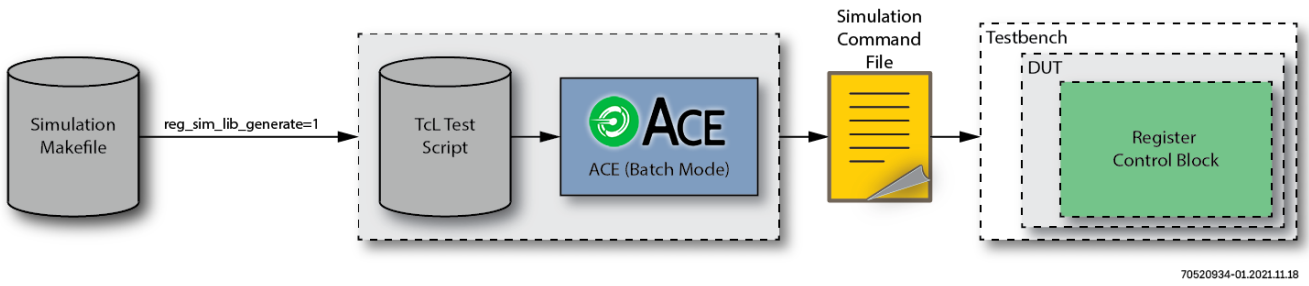


**Figure 7: *Register Control Block Simulation Flow***

For further details on the simulation flow, please refer to Simulating the Reference Design.

## Runtime JTAG

In runtime, the demo Tcl script is read in the ACE Tcl console window, and is applied directly to the JTAG port of the connected device. In runtime both the Tcl command `puts` and the ACE built-in command `message` can be used to provide status as the script is run.

The runtime flow is shown in the diagram below.

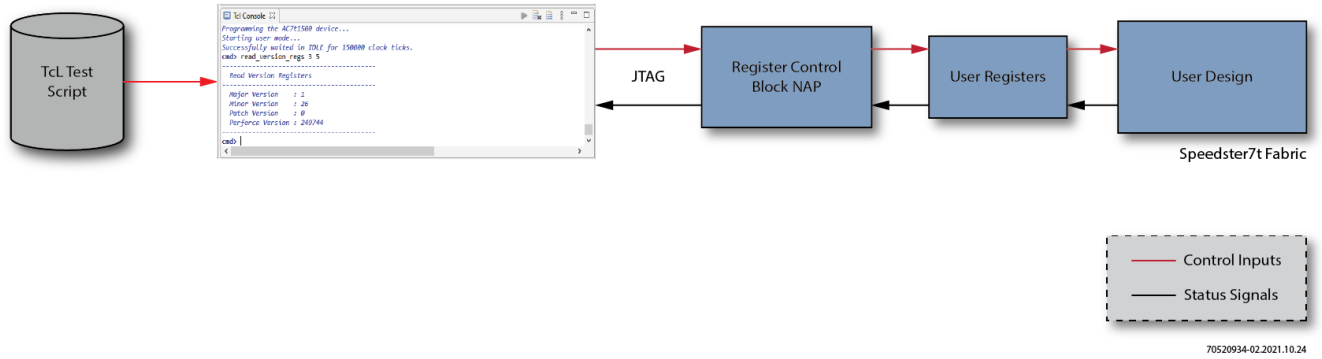


**Figure 8:** ***Register Control Block Runtime Flow***

## Runtime PCIe

The ACX_NAP_AXI_MASTER used by the register control block is accessible from any of the programming methods into the device. Therefore the register control block can be accessed and controlled via any of the devices PCIe ports. For details of PCIe drivers please contact Achronix support.

## Register Definitions

The reference design has four user registers, `CONTROL_REG`, `STATUS_REG`, `NUM_TRANSACTIONS_REG` and `SCRATCH_REG`, which are described below. The `register_control_block` is located in col = 3, row = 3. This location may be changed to suit the user design.

The placement of register control ACX_NAP_AXI_MASTER is specified in both the `/src/tb /tb_2d_noc_ref_design.sv` and `/src/constraints/ace_placements.pdc` files.

**Table 23:** ***CONTROL_REG (Offset : 0x00_0000)***

| Name | Bits | Access | Default | Description |
|---|---|---|---|---|
| `start_axi` | 0 | RW | `1'b0` | Assert to start sending transactions on the AXI Slave NAP. |
| `start_h_send` | 1 | RW | `1'b0` | Start sending transactions on the horizontal NAP. |
| `start_v_send` | 2 | RW | `1'b0` | Start sending transactions on the vertical NAP. |
| `start_h_chk` | 3 | RW | `1'b0` | Start checking transactions on the horizontal NAP. |
| `start_v_chk` | 4 | RW | `1'b0` | Start checking transactions on vertical NAP. |
| `nap_chk_rstn` | 5 | RW | `1'b0` | Active low reset input for block checking transactions. |
| `nap_send_rstn` | 6 | RW | `1'b0` | Active low reset input for block generating transactions. |

**Table 24: *STATUS_REG (Offset : 0x00_0004)***

| Name | Bits | Access | Default | Description |
|---|---|---|---|---|
| pll_chk_clk_lock_reg | 0 | RO | 1'b0 | PLL lock signal for i_chk_clk. |
| pll_send_clk_lock_reg | 1 | RO | 1'b0 | PLL lock signal for i_send_clk. |
| xact_done | 2 | RO | 1'b0 | Asserted when the total requested number of AXI transactions have been completed. |
| fail_row | 3 | RO | 1'b0 | Asserted when data received on the horizontal NAP does not match the expected data. |
| fail_col | 4 | RO | 1'b0 | Asserted when data received on the vertical NAP does not match the expected data. |
| fail_axi | 5 | RO | 1'b0 | Output enable for o_fail_col, tied to 1'b1. |

**Table 25: *NUM_TRANSACTIONS_REG (Offset : 0x00_0008)***

| Name | Bits | Access | Default | Description |
|---|---|---|---|---|
| num_transactions | [12:0] | RW | 13'b0 | The number of transactions that are sent by the AXI packet generator. This value is used to control the overall test and indicate xact_done when all transactions are complete.<br>If the value is set to 13'h0, all tests run continuously and xact_done is not asserted. |
| Reserved | [31:13] | RW | 19'b0 | Unused. Must be set to 19'b0. |

**Table 26: *SCRATCH_REG (Offset : 0x00_000C)***

| Name | Bits | Access | Default | Description |
|---|---|---|---|---|
| scratch | [31:0] | RW | 32'b0 | Scratch Register to test read and write access. |

## Script and Command File

For this design, the demo Tcl script is /ac7t1550/demo/scripts/ac7t1550_2D_NoC_demo.tcl, and the generated simulation command file is /ac7t1550/sim/ac7t1550_2D_NoC_sim.txt.

## Design Considerations

Within the design there are a number of considerations that must be taken into account.

**Note**

The design supplied is compatible with the VectorPath® S7t-VG6 accelerator card layout and pinout. However the VectorPath card is supplied with an AC7t1500ES0 device, rather than the AC7t1550ES0 device which is needed for the cryptographic engine. Therefore the design supplied cannot be downloaded or run on the VectorPath card.

The details below are provided purely to describe the interface and clock signals that are input to the design. All references to downloading bitstreams and executing tests should be ignored.

## VectorPath Card

This design is compatible with the Achronix VectorPath® S7t-VG6 accelerator card, revision 1 onwards. The provided bitstream, in `/demo/bitstream`, can be downloaded to the VectorPath card, and the test executed using the script in `/demo/scripts`. In addition, the design can be rebuilt to run on the card.

The design includes the IP configuration files, (`.acxip`), in `/src/acxip`, that specify the pinout of the VectorPath card, including clock and GPIO signals. Included files are shown in the table below.

**Table 27: *VectorPath IP Configuration Files***

| File | Ports Connected | Direction | Description |
|---|---|---|---|
| vp_clkio_ne.acxip | ext_pps | Input | |
| | fpga_rst_l | Input | Driven from BMC controller |
| | pcie_perst_l | Input | |
| | fpga_fab_clk_2 | Input | 400MHz differential |
| vp_clkio_nw.acxip | fab_refclk_0 | Output | 100MHz |
| | fpga_fab_clk_3 | Input | 800MHz differential |
| | fpga_fab_clk_4 | Input | 10MHz differential |
| vp_clkio_se.acxip | fpga_fab_clk_5 | Input | 500MHz differential |
| | fpga_fab_clk_6 | Input | 50MHz differential |
| vp_clkio_sw.acxip | fpga_fab_clk_7 | Input | 100MHz differential |
| | fpga_fab_clk_1 | Input | 200MHz differential |
| vp_gpio_n_b0.acxip | exp_gpio_fpga[7:0] | Inout | General purpose IO |
| | ext_gpio_oe_l | Output | GPIO output enable |
| | led_oe_l | Output | Enable LED outputs |
| vp_gpio_n_b1.acxip | ext_gpio_dir[7:0] | Output | GPIO signal direction |
| | led_l[5:4] | Output | Board LEDs. Active low |
| vp_gpio_n_b2.acxip | led_l[7:6, 3:0] | Output | Board LEDs. Active low |

**Table 28:** ***VectorPath IP Configuration Files (cont.)***

| File | Ports Connected | Direction | Description |
|---|---|---|---|
| vp_gpio_s_b0.acxip | fpga_i2c_mux_gnt | Input | I2C interface |
| | fpga_i2c_req_l | Output | I2C interface. Active low |
| | fpga_avr_txd | Output | AVR interface |
| | fpga_avr_rxd | Input | AVR interface |
| | fpga_ftdi_txd | Output | FTDI interface |
| | fpga_ftdi_rxd | Input | FTDI interface |
| | irq_to_fpga | Input | Interrupt |
| | irq_to_avr | Output | Interrupt |
| | test[1] | Output | Test interface |
| | qsfp_int_fpga_l | Input | QSFP interface. Active low |
| vp_gpio_s_b1.acxip | mcio_vio[3:0] | Inout | MCIO interface |
| | mcio_dir[3:0] | Output | MCIO interface |
| | mcio_dir_45 | Output | MCIO interface |
| | test[2] | Output | Test interface |
| vp_gpio_s_b2.acxip | mcio_scl | Inout | MCIO interface |
| | mcio_sda | Inout | MCIO interface |
| | mcio_oe2_l | Output | MCIO interface |
| | mcio_oe_45_1 | Output | MCIO interface |
| | fpga_sys_scl | Inout | I2C interface |
| | fpga_sys_sda | Inout | I2C interface |
| vp_pll_nw_2.acxip | pll_nw_2_ref0_312p5_clk | Output | 312.5MHz clock |
| vp_pll_sw_2.acxip | pll_sw_2_ref0_312p5_clk | Output | 312.5MHz clock |

**Table Note**

- If porting a user design to the VectorPath card, it is recommended to use these files to define the interface to the card.

# Instructions

## Installing the Reference Design

### Downloading

The design can be downloaded via the Achronix website. The knowledge base article How do I Download Reference Designs for Speedster7t and Speedcore Devices includes the download link for all reference and demo designs.

### 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

Scripted simulation or implementation flows under Windows 10 require installing 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 any of the options above cannot be installed, it is still possible to run some of the flows under Windows:

- Implementation – GUI projects are provided for both Synplify and ACE allowing builds 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\8.6\Achronix_CAD_Environment`. Additionally, the environment variable `ACE_INSTALL_DIR` must use "/" as path separators rather than "\". For example:
> `ACE_INSTALL_DIR = C:/achronix/8.6/Achronix_CAD_Environment/Achronix`

## Tool Versions

All designs were tested with the following tools:

**Table 29: *Minimum Tool Versions***

| Software | Version |
|---|---|
| ACE | 8.6.1 |
| Device Simulation Model | 8.6.1 |
| Synplify Pro | R-2021.03X |
| 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\8.6\Achronix_CAD_Environment`
- Linux – `/opt/achronix/ace/8.6`

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/8.6/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 the preferred layout. However, if the structure is modified, the necessary makefiles and build scripts must be modified to suit. To support portability, use relative paths as opposed to absolute paths with environment variables to select the root directory.

**Note**

Not every directory exists in every design. Some designs do not require certain files.

**Table 30: *Design Directory Structure***

| Directory | | | Decription |
|---|---|---|---|
| <design_name> | | | Root directory. Contains release notes. |
| | /build | | Synthesis and place-and-route building. |
| | /demo | | ACE Hardware Demo directory. |
| | | /bitstream | Bitstream to download to target board. |
| | | /gui | GUI XML file to create the ACE HW demo GUI layout. |
| | | /scripts | Scripts to run on the target board. |
| | /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 Verilog, SystemVerilog and VHDL RTL languages. These languages can be used for both building and standalone simulation. If using the full-chip BFM simulation, that environment requires Verilog or SystemVerilog for the top-level testbench. 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 are detailed below.

**Table 31: *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` | Controls ACE settings such as flow mode, speed grade and reporting of unconstrained paths. |
| `ace_placements.pdc`(1) | Fixes locations of elements within the ACE fabric and creation of placement regions. |
| `synplify_constraints.fdc` | Synplify FPGA design constraints. Sets 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` | Controls Synplify settings such as top module. Creates synthesis specific parameters, generics and defines. |

**Table Notes**

1. Pin placements between the fabric and the I/O ring are automatically created by the I/O ring designer, and provided in the `<design_name>_ioring.pdc` files.

### 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 32: *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 including 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 Register Transfer Level (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 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 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 ``, `` `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.

**Table 33: *DSM Utility Packages***

| Device | DSM Utility Package | ACX_DEVICE_NAME |
|---|---|---|
| AC7t1500ES0 | ac7t1500_utils.svh | ac7t1500 |
| AC7t1550ES0 | ac7t1550_utils.svh | ac7t1550 |

## 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 table below lists the available `simmodels.sv` files, and the device specific macro that each creates.

**Table 34: *Simulation Model Files and Defines***

| Device | Simulation Model File | ACX_DEVICE Macro |
|---|---|---|
| AC7t1500ES0 | AC7t1500ES0_simmodels.v | ACX_DEVICE_AC7t1500ES0 |
| AC7t1550ES0 | AC7t1550_simmodels.v | ACX_DEVICE_AC7t1550ES0 |

## 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 example below, 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_AC7t1500ES0
    `include "ac7t1500_utils.svh"
`elsif ACX_DEVICE_AC7t1550ES0
    `include "ac7t1550_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 detailed below:

```
// 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 8.5.0.0 as a minimum
    `ACX_DEVICE_NAME.require_version(8, 5, 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 8.5 release, the arguments would be set as 8,5,0,0.

> **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 45) 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 48) and DSM Direct-Connect Interfaces (see page 49) tables.

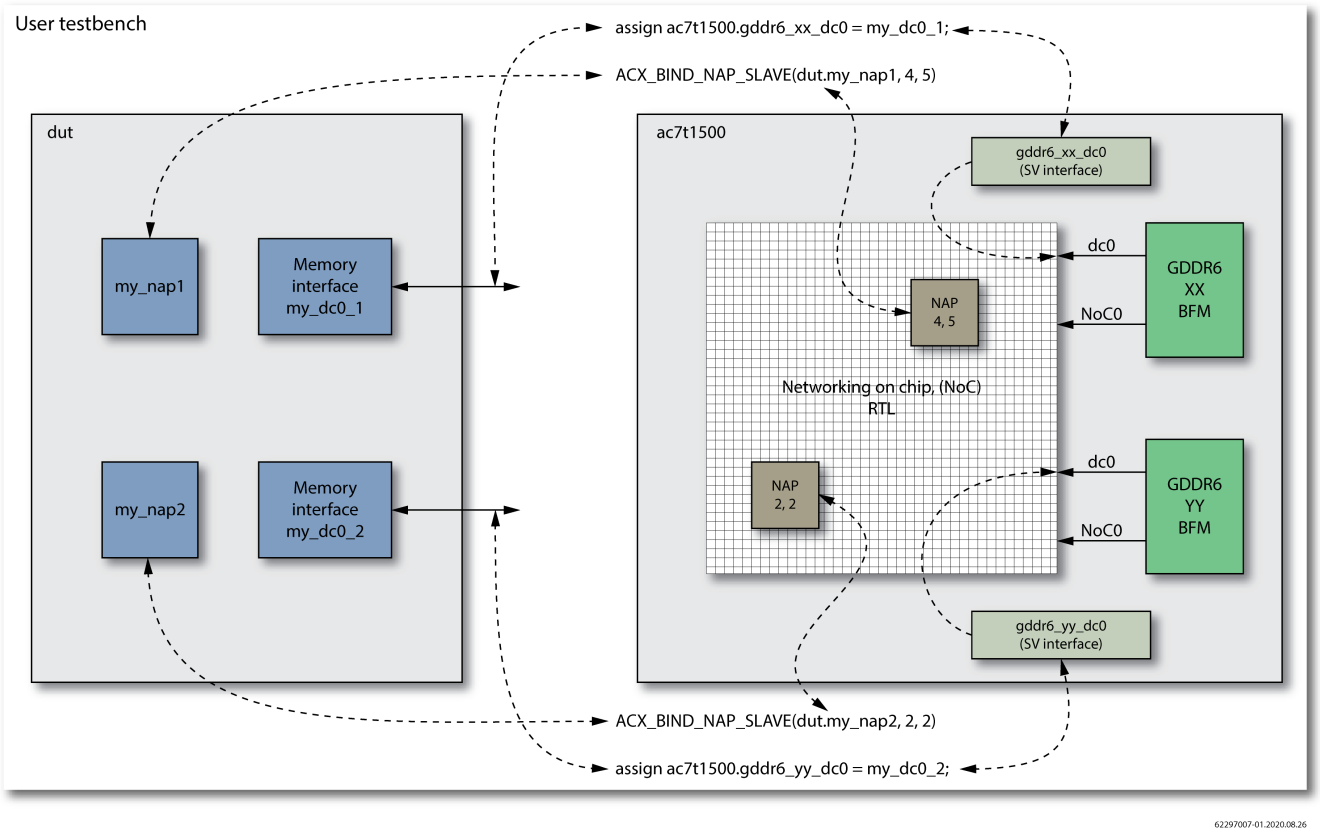


**Figure 9:** ***Example Simulation Structure***

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 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_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 `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_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 35: *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 until 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 36: *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. |
| 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. |
| ACX_BIND_NAP_ETHERNET | user_nap_instance, noc_colunm, noc_row | To bind an Ethernet NAP instance, ACX_NAP_ETHERNET. |

**Table Note**

- `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>`.

## 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
- 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.

- 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 below.

## DSM DC Interfaces

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

**Table 37: *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 |
| 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 |

| Subsystem | Interface Name | Physical Location[1] | GDDR6 Channel | SystemVerilog Interface Type | Data Width | Address Width |
|---|---|---|---|---|---|---|
| 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 | – |

**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 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;


// 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.

**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 : 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 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 = `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),
        ......
    );
```

## 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
`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 38: *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 | – |

**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 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
        `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 (see page 47) 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 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 39: *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 57) below 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 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. However, simulation times are increased significantly. 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.

> **Note**
>
> The Base DSM for each device is included within ACE from version 8.5 onwards. The installation details that follow are only required for installing the RTL package, or when using older versions of ACE prior to version 8.5

### 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.zip.`

> **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 8.3.3 may be used with ACE 8.5.

### Download

The base DSM packages are included within all ACE releases from 8.5 onwards. Ensure when installing ACE that the relevant DSM archive is expanded and installed into the locations listed below.

Due to licensing conditions (above (see page 61)), RTL packages are obtained by sending a support request to support@achronix.com.

Any package is only required to be installed once, the package 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:

1. Navigate to `<ACE_INSTALL_DIR>/system/data/AC7t1500ES0`, (or the relevant device directory).
2. Remove the `/sim` directory.
3. Return to the root of the ACE installation.
4. Proceed with the instructions for First Installation (see page 62) 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 for ACE versions prior to 8.5. 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 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 AC7t1500ES0 as an example):

```
ACX_DEVICE_INSTALL_DIR = $ACE_INSTALL_DIR/system/data/AC7t1500ES0
```

When installed as standalone, the following setting should be used, (with AC7t1500ES0 as an example):

```
ACX_DEVICE_INSTALL_DIR = <location of standalone package>/system/data/AC7t1500ES0
```

**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 AC7t1500ES0 as an example):

```
> make ACX_DEVICE_INSTALL_DIR=<location of standalone package>/system/data
/AC7t1500ES0
```

## 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 list the flow options that 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 only interacts 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 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 Register Transfer Level (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 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.

> **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 to configure the simulation to suit the user design. 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 design is ported to the user 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, `$ACE_INSTALL_DIR/system/data/AC7t1500ES0`. For further information consult the I/O ring simulation installation instructions.

## Prerequisites

Before running any installation, 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 contains 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

To compile and run the Full-Chip RTL flow, it is necessary to enable SystemVerilog defines for the simulation compilation. Setting the simulation makefile `FLOW` variable to `FULLCHIP_RTL` instructs the makefile to 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 it is desired to simulate GDDR6 memory controller 1 with the full RTL, but leave all other GDDR6 controllers using the BFM model, the `system_files_rtl.f` file must be edited 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
```

The available defines to enable full RTL for each module, rather than the BFM, are listed in the table below.

**Table 40: *Simulation RTL defines***

| Module [1] | 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 |
| DDR4 controller | ACX_DDR4_FULL |

| Module [1] | Define |
|---|---|
| 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 [2] | 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. 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.

**Warning!**

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

## Runtime Programming Scripts

Many of the reference designs make use of a control module within the user design (`/src/reg_control_block.sv`). This module creates a set of user registers within the reference design which are used for controlling and monitoring the test during runtime (when the chip is in user mode). The programming of these registers is distinct from the configuration phase of the device, when the static configuration is loaded into the various interface subsystems CSR (Control and Status Registers) memory space. These user registers give the design flexibility in configuration, can be used for version control and to determine design capabilities, and can control and monitor sequences and tests.

### Simulation Command Files

The `reg_control_block` contains a ACX_NAP_AXI_MASTER, which controls the generated user registers. Programming of these registers can then be performed using a simulation command file. In simulation, this command file is read by the BFM model of the FPGA Configuration Unit (FCU). The commands in the simulation command file are transferred to the ACX_NAP_AXI_MASTER, which in turn performs the desired operation on the user registers in the reference design. By convention, simulation command files are text files with a `.txt` extension. They have the same format as the "Configuration File Format" shown in Device Simulation Model.

## Runtime Programming Scripts

Simulation command files are generated from a runtime programming script. Runtime programming scripts are written in Tcl, and can be executed in the ACE Tcl console directly on the hardware. By using the same runtime programming script to generate the simulation command file, the same sequence of operations can be tested in both simulation and hardware.

There are a number of key sections to a runtime programming script. These are detailed below.

```
# Include any utility files that may have common functions
source AC7t1500ES0_common_utils.tcl

# Define simulation command filename name and location.
# Path is relative to this script location
set OUTPUT_FILENAME "../../sim/<simulation command filename>.txt"

# Process any input arguments
# IMPORTANT : This must be included for the reg_lib_sim_generate option
ac7t1500::process_args $argc $argv

# Open the simulation command file.
# File will only be created if not running under ACE
# Pass script name, ($argv0), for header
ac7t1500::open_command_file $OUTPUT_FILENAME $argv0

# When running under ACE, ensure jtag port is open
# When generating a simulation command file, this call is ignored.
ac7t1500::open_jtag

# ------------------------------------------------------------------
#                 Test code starts here
# ------------------------------------------------------------------

# ------------------------------------------------------------------
#                 Test code ends here
# ------------------------------------------------------------------

# Close command file
# File will only exist when not running under ACE
ac7t1500::close_command_file $OUTPUT_FILENAME
```

**Note**

The register library Tcl functions make use of a namespace, based on the device, in order to use common function names across multiple devices. The format of a command is `<device namespace>::function()`

The example above is for the Speedster7t AC7t1500 device. When using a different device, change the namespace of the command to the respective desired device

## Simulation Command File Generation

The runtime programming script and the simulation command file both make use of the device register library built into ACE. Full details on this register library can be found in Runtime Programming of Speedster FPGAs. To generate the simulation command file, ACE is run in batch mode using the runtime programming script, as shown below.

```
# Call to ACE in batch mode to create a simulation command file
ace -batch -script_file <path to runtime programming script> -script_args "-reg_lib_sim_generate
1"
```

For the reference designs that make use of a simulation command file, the generation command is included in the `/sim/<simulator>/Makefile`. The simulation command file is generated in the location specified in the runtime programming script. In the reference designs, the simulation command file is written to the `/sim` directory so it is available for all simulators.

## reg_lib_sim_generate

It is necessary for ACE to distinguish between a runtime programming script that is being executed in hardware (and so might need to open jtag ports, or poll for register responses) and a runtime programming script that is being used to create a simulation command file. In order to separate these two modes, an embedded variable in the device register library is used: `reg_lib_sim_generate`.

When ACE is running against hardware, `reg_lib_sim_generate` is not set, and the Tcl script commands are executed directly in the ACE Tcl console.

When ACE is executed in batch mode, using the command above, `reg_lib_sim_generate` is set. ACE therefore translates the runtime programming script Tcl commands into equivalent simulation commands and writes these to the simulation command file.

If it is necessary for commands to vary between hardware and simulation (for example, in hardware it is possible to run extended tests with many millions of transactions which would be infeasible to simulate) then the `reg_lib_sim_generate` variable can be used within the runtime programming script to select separate hardware and simulation control flows. The state of the variable can be read using the Tcl function `<device namespace>::get_reg_lib_sim_generate()`. An example of setting different test lengths using this function is shown below.

```
# Use function get_reg_lib_sim_generate to determine if this script is being run under hardware
or simulation
if { ![ac7t1500::get_reg_lib_sim_generate] } {
    # Hardware test - test 10M packets
    set num_pkts 10000000
} else {
    # Simulation test - test 1000 packets
    set num_pkts 1000
}
```

## Testbench Sequence

When using the simulation command file, it is important to ensure that it is applied in the right sequence to correspond to when it would be executed in hardware.

The runtime programming script is executed when the device is in user mode. This follows the configuration mode when the bitstream is loaded into the interface subsystems and the fabric. The mode is indicated by the FCU_CONFIG_USER_MODE pin on a Speedster7t device. Execution of the simulation command file should only occur when this signal has been asserted. For full details of the configuration pins and programming modes, see the *Speedster7t Configuration User Guide* (UG094).

An example sequence, showing the connections to the DSM model, the device configuration, and the runtime programming sequence is shown below.

```
// Create signal to indicate device configuration state
// When set, device is in user mode
logic chip_ready;


// Instantiate the DSM model, (device dependant)
`ACX_DEVICE_NAME `ACX_DEVICE_NAME (
    .FCU_CONFIG_USER_MODE (chip_ready)
);


// Programming sequence
initial
begin
    // Device configuration phase.  Use ACE generated configurations
    // These configurations perform the same function as loading the bitstream
    `include "../../src/ioring/<design name>_sim_config.svh"


    // Wait for device to enter user mode
    while ( chip_ready !== 1'b1 )
        @(posedge clk);

    // Device now in user mode
    // Execute simulation command file to perform runtime programming
    `ACX_DEVICE_NAME.fcu.configure( "../<simulation command file>.txt", "full");


    // When simulation command file completes, test is done
    $finish
end
```

## Running the Simulation

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

### VCS

```
$ 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

```
$ 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
file.
$ 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
   ```

3. 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, Install ACE in a directory without spaces in the path name. Example:
  C:\achronix\8.5\Achronix_CAD_Environment
- Additionally, the environment variable, ACE_INSTALL_DIR, must use "/" as path separators rather than "\". Example:
  ACE_INSTALL_DIR = C:/achronix/8.5/Achronix_CAD_Environment/Achronix
- The do script is configured for the FULLCHIP_BFM flow. It can be modified to match the STANDALONE or FULLCHIP_RTL flow by selecting the appropriate options commented within the script.

### Results Verification

All 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 are provided in the details of each individual design.

## Changing Devices In Simulation

All reference designs are pre-configured for a single device, usually the Speedster7t AC7t1500ES0. However, It might be necessary to change the selected device in simulation to match the device being implemented.

### Verilog Macro Defines

To aid changing between devices, ACE libraries include device-specific simulation and synthesis files. For simulation, these files are named <ACE_INSTALL_DIR>/libraries/device_models/<DEVICE>_simmodels.v . These files include device-specific defines that include or exclude required portions of the code.

These created defines are named ACX_DEVICE_<full device name>. The code example below shows how the defines are used in a testbench 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_AC7t1500ES0
    `include "ac7t1500_utils.svh"
`elsif ACX_DEVICE_AC7t1550
    `include "ac7t1550_utils.svh"
`endif
```

In the above example, the selected DSM utility file creates another define, ACX_DEVICE_NAME, which matches the full device name. This define is used throughout the testbench to refer to the DSM as shown in the code example below.

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

## Changes Required

The following steps illustrate how to change the simulated device:

1. In `/sim/<simulator>/Makefile`, modify the `DEVICE` variable:

   ```
   # Define the target device
   # Two devices currently supported; AC7t1500ES0 and AC7t1550
   DEVICE     := AC7t1500ES0
   ```

2. Alternately, in any call to the `Makefile`, the default device can be overridden on the command line invocation:

   ```
   $> make DEVICE=AC7t1550
   ```

3. For Questa GUI mode, modify the `DEVICE` variable in `/sim/questa/qsim_<design>_top_ref_design.do`:

   ```
   # ------------------------------------------------------------------
   # Define the target device
   # ------------------------------------------------------------------
   # 2 devices currently supported; AC7t1500ES0 and AC7t1550
   set DEVICE               "AC7t1500ES0"
   ```

> **Note**
>
> The defines and device names are case sensitive. The correct capitalization must be applied when overriding or specifying a device name.

# 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, ensure the following are configured:

- `ACE_INSTALL_DIR` environment variable. This variable must point to the ACE installation directory containing the ACE executable.
- ACE should be in the environment path. Add `$ACE_INSTALL_DIR` to the path variable.
- Synplify Pro should be in the environment path.

## I/O Ring Files

For the Speedster7t family of devices, the hardened subsystems (GDDR, Ethernet, etc.) have individual configuration files specifying the configuration of each of the subsystems. These files are located in the `/src/acxip` directory and are included within the ACE project, either directly in the `/src/ace` GUI project file, or for the batch flow, under the `multi_acxip_files` section of the `/src/filelist.tcl` file.

In order for ACE to compile and place and route a design, it is necessary to use these source `acxip` configuration files to generate I/O ring constraint files. These I/O ring files specify the timing and placement constraints between the hardened subsystems (which form an I/O ring around the programmable fabric) and the fabric logic itself. Further, the I/O ring files include the bitstream programming information for the hardened subsystems.

For all Speedster7t reference designs, the pre-generated I/O ring files are included in the `/src/ioring` directory. These files are pre-generated using the specified ACE version which can be found in the Release Notes for the design.

It might be necessary to regenerate these I/O ring files if using an updated version of ACE. Follow the steps below to update the files.

> **Note**
>
> Before updating the /src/ioring files, ensure that the files are writable. In addition, the `/src/acxip` files also must be writable as newer versions of ACE might update them. As delivered, the files are set to be read-only.

### I/O Ring Batch Flow

1. Use either the `$ make` default flow (no arguments) or the `$ make ioring_only` flow to re-generate the I/O ring files. The `ioring_only` option only generates the I/O ring files, it does not run synthesis or ACE place and route.
2. The directory that the I/O ring files are written to is set with the `generate_ioring_path` variable within the `/src/filelist.tcl` file. The path is relative to the `/src/ace` project directory. By default, this variable is set to `../ioring`, which results in the `/src/ioring` directory being specified.

### I/O Ring GUI Flow

1. In the ACE GUI, select the **IP Configuration** Perspective.

2. Open any of the project hardened subsystem configuration files located under the **Project** → **IP** selection.
3. Ensure that, in the **IP Problems** window, there are no Errors or Warnings. If there are, resolve these first.

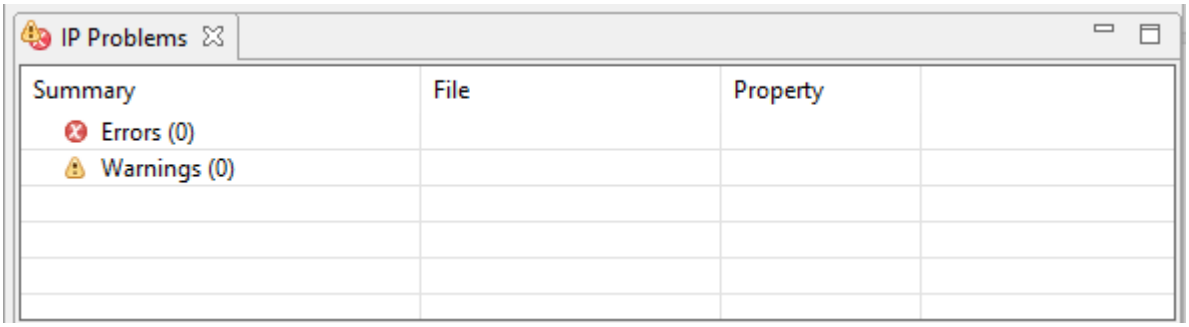


**Figure 10:** ***IP Problems Window***

4. On the opened IP configuration file, select **Generate**.
5. ACE prompts for the directory in which to save the I/O ring files. Select `/src/ioring`. Do not use the ACE default of `/src/ace/ioring_design`.

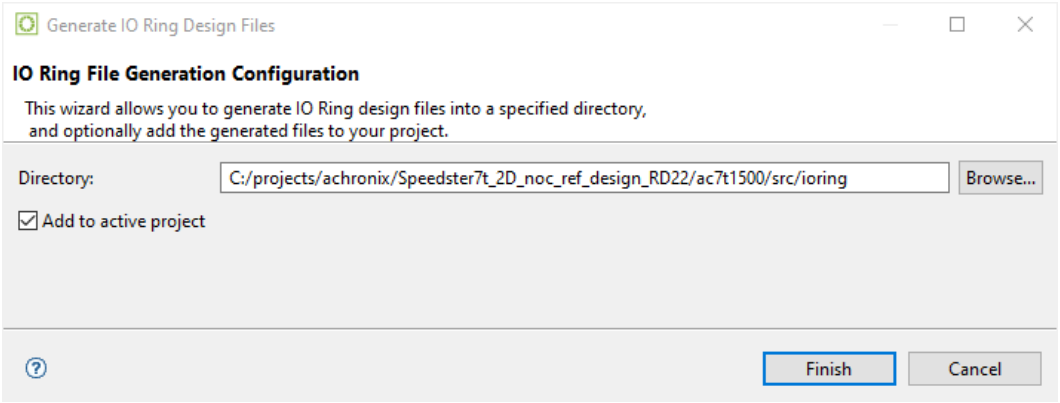


**Figure 11:** ***ACE Select IP Generated Files Directory***

6. Issue the command `generate_ioring_design_files` in the Tcl console window.

When the I/O ring files have been re-generated, the `/src/ioring` directory can be checked to ensure that the new files are present, and that the versions of these files match those of the ACE version used to generate them.

## Changing Speed Grades

ACE generates I/O ring `.sdc` delay files named according to the speed grade selected within the ACE project ( **Projects Perspective** → **Options** → **Speed Grade**). If the speed grade is changed, then ACE generates new files named accordingly. These new files are added to the ACE project in both batch and GUI mode. It is the user's responsibility to remove previous speed grade files from the project.

> **Warning!**
>
> When using the batch flow and changing speed grade, if I/O ring generation is selected using the default flow, then for that run the ACE project has the new speed grade constraint files. However, on any subsequent run, the ACE project is built on the fly using the files specified in `/src/filelist.tcl` . Therefore, if a new speed grade is selected, the `filelist.tcl` file must be updated to include the new files appropriate to the chosen speed grade.

## Batch Flow

In the root directory, there is a `/build` directory containing `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, the paths must be amended accordingly.

When the makefile is run (with the default options), the following are created:

- `/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, 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). Any options required for the ACE-generated project are controlled by the `/src /constraints/ace_impl_options.tcl` file.

### Makefile Options

The makefile supports multiple build flow options:

```
$ make               : Default flow. Regenerate all files in /src/ioring. 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 ioring_only : Regenerate all files in /src/ioring.
$ 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 the files 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 the `/src/ace` and `/src/syn` directories respectively. Open these files to interactively edit or run builds.

> **Note**
>
> When using the GUI projects, any generated files are placed in the GUI project file directory.
>
> - For Synplify Pro, the revision directory, rev_1, etc., is generated in `/src/syn/rev_1`.
> - For ACE, any implementation directory is generated as `/src/ace/impl_<name>`.
> - For ACE, when I/O Designer is used to generate new IP configuration files, the reference design target directory for those files is `/src/ioring`.
>
> ACE defaults to `/src/ace/ioring_design`. When saving these files it is necessary to explicitly specify the `/src/ioring` path.

When running builds using the batch flow, both the relevant project files are written under the `/build/results` directory. Within this directory are the generated project files for both ACE and Synplify Pro. Both of these project files can be opened in GUI mode and interactively re-run or the builds edited.

## Changing Devices When Building

All reference designs are pre-configured for a single device; usually the AC7t1500ES0. However, it might be necessary to change the selected device in synthesis and place and route to match the device being implemented.

### Verilog Macro Defines

To aid changing between devices, ACE libraries include device-specific simulation and synthesis files. For synthesis, these files are named `<ACE_INSTALL_DIR>/libraries/device_models/<DEVICE>_synplify.v`. These files include device-specific defines that include or exclude required portions of the code.

These created defines are named `ACX_DEVICE_<full device name>`. The code example below shows how the defines are used in a design to optionally include device-specific modules:

```
    // ------------------------------------------------------------------
    // Support for the AC7t1550 device
    // ------------------------------------------------------------------
    // If this design is intended to be targeted to the ac7t1550 device,
    // (which includes the hard-ip cyptographic core), then it is necessary to
    // instantiate the core in the code, even if unused.
    // ------------------------------------------------------------------
`ifdef ACX_DEVICE_AC7t1550
    ACX_AESX_GCM_K_BYPASS x_ACX_AESX_GCM_K_BYPASS ();
`endif
```

## GUI Flow

When first changing the device, it is necessary to run the GUI flow to target the IP definition files, `/acxip`, to the new device, and then create new IP generation files, `/src/ioring`, properly configured to the new device.

### *Synthesis*

The first step is to create a netlist targeted to the new device using the Synplify Pro GUI project. Within `/src/syn/<design_name>.prj`, make the following changes:

```
#project files
add_file -verilog "$ACE_INSTALL_DIR/libraries/device_models/AC7t1550_synplify.v"  <- Update
library file to new device


#device options
set_option -part AC7t1550 <- Update selected part
```

> **Note**
>
> It is suggested to edit the Synplify Pro project file with a text editor, rather than from within the tool. The tool does not recognise the environment variable, `ACE_INSTALL_DIR` that is used to create relative paths to the ACE library files. If the project is edited within the tool, the relative paths are replaced with absolute paths, which reduces project portability.

When the above changes are made, synthesis can be performed and a netlist generated.

### *ACE*

When a netlist for the new device has been created, the ACE GUI project must be updated with the following steps:

1. From the **Projects Perspective, Options** tab, select the new **Target Device**.

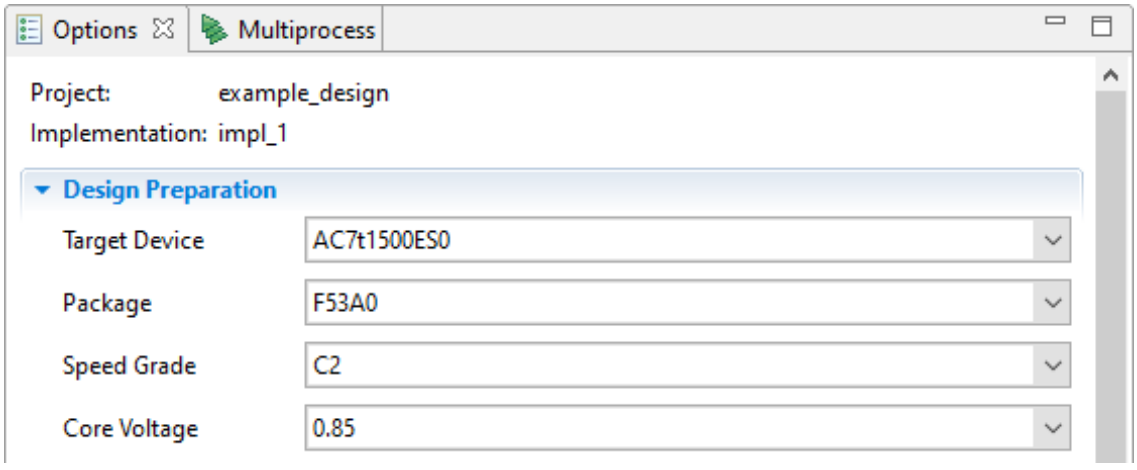


**Figure 12:** ***ACE Select Target Device***

2. Open each of the IP definition files from the **Project Perspective, Projects** tab, **IP** selection.

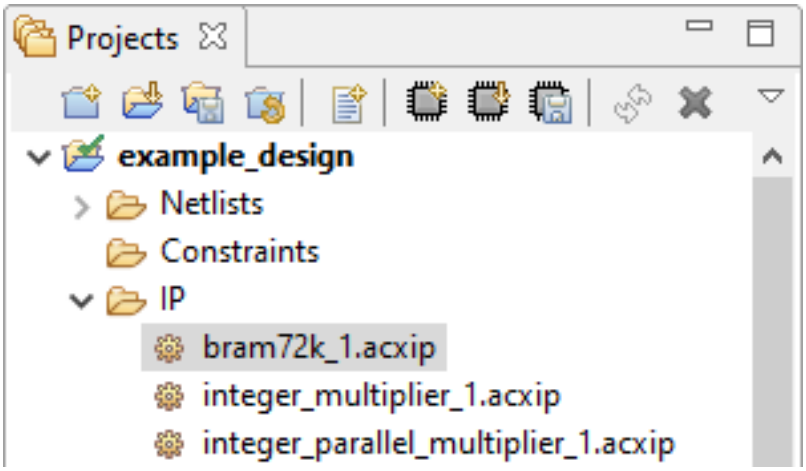


**Figure 13:** ***ACE Select IP***

3. Within each IP definition file, select the new **device**.

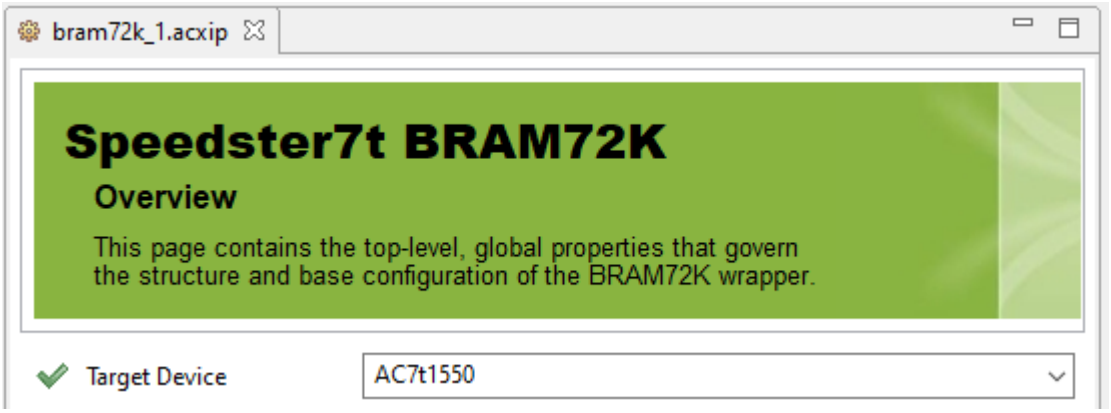


**Figure 14:** ***Select IP Device***

4. **Save** all of the IP definition files, (either **Ctrl** → **S** on each file, or else the I/O ring generation process saves all `acxip` files).
5. Run I/O Ring generation, see I/O Ring Files (see page 74) above.

When the above changes have been made, the design can be placed and routed normally.

## Batch Flow

It is recommended that, before running the batch build flow with a new device, the GUI flow is used to update the ACE generated IP files as detailed above. When these files are present, do the following to change the device:

1. In `/build/Makefile`, modify the `DEVICE` variable:

```
# ------------------------------
# If targeting Speedster7t AC7t1500 device, enable the lines below
# If using a Speedcore, disable this section
# ------------------------------
# Currently supported devices, AC7t1500ES0 and AC7t1550
DEVICE     := "AC7t1500ES0"
```

2. Alternately, in any call to the `Makefile`, the default device can be overridden on the command line invocation:

```
$> make DEVICE=AC7t1550
```

# Revision History

| Version | Date | Description |
|---|---|---|
| 1.0 | 24 Feb 2020 | • Initial Release. |
| 2.0 | 17 Apr 2020 | • Updates related to ACE 8.1.2. |
| 2.1 | 28 May 2020 | • Added description on how clock periods are defined in testbench.<br>• Added description on how testbench performs configuration.<br>• Corrected launch of QuestaSim GUI command.<br>• Updated example instantiation of AC7t1500. |
| 2.2 | 22 Jul 2020 | • Updated signal/port names and clock descriptions to match design changes due to ACE 8.2 I/O Designer change. |
| 2.3 | 27 Jan 2021 | • Updated output enable signals to be *_oe for compatibility with ACE 8.3. |
| 3.0 | 01 Jun 2021 | • Update ACE and DSM requirements to 8.3.3<br>• DSM details added. |
| 4.0 | 21 Sep 2021 | • Rename to 2D NoC<br>• Split into two designs, AC7t1500 and AC7t1550<br>• Describe Cryptographic engine core functionality and usage |
| 4.1 | 18 Nov 2021 | • Both designs use register control block to control tests<br>• Add parameters to configure randomized or linear traffic |
| 4.2 | 28 Jan 2022 | • Update LED port designators (see page 23) to match design |


Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

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



## Notice of Disclaimer