Speedster7t GDDR6 Reference Design Guide

V6.0 – May 28, 2025

Reference Design

Introduction

The GDDR6 reference design demonstrates the ability to perform read and write transactions to all GDDR6 subsystems across the Speedster®7t family of FPGAs. This guide details taking the design through synthesis and the ACE tool flow for place and route across multiple devices.

The Speedster7t AC7t1400 and AC7t1500 FPGAs each have 8 GDDR6 subsystems and 16 channels while the Speedster7t AC7t800 has 3 GDDR6 subsystems and 6 channels (see the ***Speedster7t GDDR6 User Guide*** **(UG091)** for more details).

Note

For this release of the GDDR6 reference design, the supported design files only target the AC7t1500 device on the VectorPath 815 accelerator card. Therefore, this document will only cover the AC7t1500 variation of the design.

On the Speedster7t AC7t1400 and AC7t1500 FPGAs, transactions to the GDDR6 subsystems can be performed using either the 2D network-on-chip (2D NoC) interface or the direct-connect (DC) interface. The design can be used to test both channels of each GDDR6 subsystem and can be modified to expand the design to test all 16 channels of the eight total GDDR6 subsystems.

This design demonstrates interfacing between the NAPs or DC interface ports from the fabric to the GDDR6 memories via simulation.

The bitstream generated from this design and the included runtime scripts are compatible with the VectorPath 815 accelerator card.

Description

The design consists of the following logic blocks and interfaces:

Achronix Device Manager

The GDDR6 reference design makes use of the `ACX_DEVICE_MANAGER` soft IP which trains the GDDR6 subsystems for the AC7t1400 and AC7t1500 FPGAs. When the phase locked loops (PLLs) are locked, a reset is released which initiates the training process. The two least-significant bits of the 32-bit `o_status` output for this module are set high when the training process is complete. These status bits are used by the runtime script to ensure GDDR6 training is complete before starting transactions to GDDR6 memory. Further information about this soft IP can be found in the ***Speedster7t Soft IP User Guide*** **(UG103)**.

NAP Responder Module

NAP responders are used to transfer data between the GDDR6 memory and the AXI memory channel logic in the fabric. The design makes use of all eight GDDR6 subsystems; however not all channels for each GDDR6 subsystem are used. The design uses the 2D NoC interface for channel 1 of all of the subsystems. Each GDDR6 interface has a corresponding AXI memory channel logic block which serves as a data generator and checker in the design. Each of the NAPs can be bound to any arbitrary location in simulation and during placement in ACE. This flexibility enables testing scenarios where the GDDR6 subsystems can handle transactions from any NAP located in the fabric.

NAP Locations

For this VectorPath 815 reference design, the location of the NAPs were chosen to be adjacent to the target GDDR6 subsystem on the east or west side of the FPGA depending on which side was closest to the target GDDR6 subsystem. The locality was selected to reduce latency between the NAP and the GDDR6 controller. The 8 NAPs used by the design are on rows 3, 4, 5, and 6 of columns 3 and 8. The locations are specified in both the `/src/tb/tb_gddr_ref_design.sv` and the `/src/constraints/ace_placements.pdc` files. By aligning the placement between simulation and the implemented build, the most accurate correlation between the two flows is ensured.

Direct Connect (DC) Interface

The direct connect (DC) interface is a signal interface that provides fabric connectivity directly to a subsystem. The DC interface is used for channel 0 of subsystems 1, 2, 5, and 6. This design connects the GDDR6 subsystem DC interface signals to an AXI memory channel to directly perform memory accesses to the GDDR6 memory. By using the GDDR6 DC interface, a design can access the GDDR6 memory without using a NAP for transactions over the NoC. This can be advantageous for reducing traffic on the NoC and conserving NAP resources.

DC Interface Logic Locations

The AXI memory channel logic interfacing with the GDDR6 DC interface can be placed at any location within the FPGA fabric and is not constrained within the reference design. However, for better timing closure between the DC interface and the user design, the logic is naturally placed near the DC interface ports on either the east or west side of the FPGA fabric depending on the target GDDR6 subsystem.

AXI Memory Channel

The design instantiates a total of twelve instances of the AXI memory channel. Each memory channel generates AXI packets to write data to the GDDR6 subsystems and reads back the data to verify that each transaction was processed correctly. Each GDDR6 interface in this design makes use of a memory channel. Therefore, a total of eight channels are connected to NAPs and a total of four channels are connected to DC interface pins.

The AXI Memory Channel integrates an AXI packet generator, an LRAM FIFO, an AXI packet checker, and an AXI performance monitor. Together, these components act as a memory test channel. Through this module, packets are generated and sent through an AXI interface to be processed by a memory interface. These same packets are sent back through the same AXI interface in order to be compared with the sent data to verify functionality of the memory interface. An array of registers is ported to the testing module to configure and monitor the channel. The array of registers connects directly to a `register_control_block`.

Parameters

Table 1 • AXI Memory Channel Parameters

Parameter Name	Default Value	Description
`AXI_DATA_WIDTH`	256	AXI interface data width.
`AXI_ADDR_WIDTH`	42	AXI interface address width.
`LINEAR_PKTS`	0	Set to 1 to make packets have linear counts.
`LINEAR_ADDR`	0	Set to 1 to have linear addresses.
`MAX_BURST_LEN`	0	Maximum number of AXI packets in a burst.
`TGT_ADDR_WIDTH`	0	Target address width.
`TGT_ADDR_PAD_WIDTH`	0	Target address padding.
`TGT_ADDR_ID`	0	Target address ID.
`RAND_DATA_INIT`	0	Random value for data starting point.
`NO_AR_LIMIT`	0	Minimum gap to prevent GDDR/DDR pipeline reordering.
`NUM_REGS`	8	Number of read and write registers used in the channel.
`CHANNEL_CLK_FREQ`	400	Frequency of channel clock, also used to calibrate performance monitor.
`OUTPUT_PIPELINE_LENGTH`	4	Length of output pipeline.

Ports

Table 2 • AXI Memory Channel Ports

Name	Width	Direction	Description
`i_ch_clk`	1	Input	Channel clock.

Name	Width	Direction	Description
`i_reg_clk`	1	Input	Register clock. Must match the `register_control_block` clock.
`i_ch_reset_n`	1	Input	Negative reset on channel clock.
`i_regs_write`	32x `NUM_REGS`	Input	Array of registers to configure channels.
`axi_if`	–	Input/Output	AXI4 interface.
`o_regs_read`	32x `NUM_REGS`	Output	Array of registers to monitor channel.
`o_running`	1	Output	Indicates test is running.
`o_done`	1	Output	Indicates test is complete.
`o_fail`	1	Output	Indicates test has failed.

Registers

Each memory channel uses 11 registers for configuration and monitoring. Only registers 0 and 2 can be written from the register control block, all other registers are read-only. The following table lists the registers and their respective functions.

Table 3 • CONTROL_REG (Offset : 0x00_0000)

Name (†)	Bits	Access	Default	Description
`gen_rstn_pre`	0	RW	`1'b0`	Packet generator reset.
`chk_rstn_pre`	1	RW	`1'b0`	Packet checker reset.
`fifo_rstn_pre`	2	RW	`1'b0`	FIFO reset.
`mon_rstn_pre`	3	RW	`1'b0`	Performance monitor reset.
`fifo_flush_pre`	4	RW	`1'b0`	Flushes the FIFO.
`max_bursts_pre`	5	RW	`1'b0`	Determines whether the AXI packet generator creates maximum length packets (`1'b1`) or variable length packets (`1'b0`).
`gen_start_pre`	6	RW	`1'b0`	Start packet generator.

Name [(†)]	Bits	Access	Default	Description
	[31:7]	RO	`25'b0`	Reserved.

Table Notes

† The signals above are synchronized to the channel clock after being read in via the register control block clock.

Table 4 • STATUS_REG (Offset: 0x00_0004)

Name	Bits	Access	Default	Description
`test_running`	0	RO	`1'b0`	Asserted when transactions are still running.
`test_done`	1	RO	`1'b0`	Asserted when transactions are complete.
`gen_running`	2	RO	`1'b0`	Asserted when transactions are still running.
`gen_done`	3	RO	`1'b0`	Asserted when transactions are complete.
`pkt_error`	4	RO	`1'b0`	Asserted when a packet mismatch occurs.
`wr_error`	5	RO	`1'b0`	Asserted when a writing error occurs.
	[7:6]	RO	`2'b0`	Reserved.
`outstanding_compares`	[15:8]	RO	`8'b0`	8-bit number comparing transactions read against transactions compared.
	[31:16]	RO	`16'b0`	Reserved.

Table 5 • NUM_TRANSFERS_REG (Offset: 0x00_0008)

Name	Bits	Access	Default	Description
`num_transfers_reg`	[31:0]	RW	`32'b0`	Sets number of transactions to run during test. Performs continuous transactions if set to 0.

Table 6 • TEST_GEN_COUNT (Offset: 0x00_000C)

Name	Bits	Access	Default	Description
`test_gen_count`	[31:0]	RO	`32'b0`	Stores number of remaining transactions to run in test.

Table 7 • BW_RESULTS_RD (Offset: 0x00_0010)

Name	Bits	Access	Default	Description
`rd_bw_min_norm`	[9:0]	RO	`10'b0`	Minimum read bandwidth.
`rd_bw_max_norm`	[19:10]	RO	`10'b0`	Maximum read bandwidth.
`rd_bw_norm`	[29:20]	RO	`10'b0`	Average read bandwidth.
	[31:30]	RO	`2'b0`	Reserved.

Table 8 • BW_RESULTS_WR (Offset: 0x00_0014)

Name	Bits	Access	Default	Description
`wr_bw_min_norm`	[9:0]	RO	`10'b0`	Minimum write bandwidth.
`wr_bw_max_norm`	[19:10]	RO	`10'b0`	Maximum write bandwidth.
`wr_bw_norm`	[29:20]	RO	`10'b0`	Average write bandwidth.
	[31:30]	RO	`2'b0`	Reserved.

Table 9 • LATENCY_RESULTS_CURRENT (Offset: 0x00_0018)

Name	Bits	Access	Default	Description
`latency_counter_write`	[11:0]	RO	`12'b0`	Latest write latency.
	[15:12]	RO	`4'b0`	Reserved.
`latency_counter_read`	[27:16]	RO	`12'b0`	Latest read latency.
	[31:28]	RO	`4'b0`	Reserved.

Table 10 • LATENCY_RESULTS_AVG (Offset: 0x00_001C)

Name	Bits	Access	Default	Description
`wr_lat_avg`	[11:0]	RO	`12'b0`	Average write latency.
	[15:12]	RO	`4'b0`	Reserved.

Name	Bits	Access	Default	Description
`rd_lat_avg`	[27:16]	RO	`12'b0`	Average read latency.
	[31:28]	RO	`4'b0`	Reserved.

Table 11 • LATENCY_RESULTS_MAX(Offset: 0x00_0020)

Name	Bits	Access	Default	Description
`wr_lat_max`	[11:0]	RO	`12'b0`	Maximum write latency.
	[15:12]	RO	`4'b0`	Reserved.
`rd_lat_max`	[27:16]	RO	`12'b0`	Maximum read latency.
	[31:28]	RO	`4'b0`	Reserved.

Table 12 • LATENCY_RESULTS_MIN(Offset: 0x00_0024)

Name	Bits	Access	Default	Description
`wr_lat_min`	[11:0]	RO	`12'b0`	Minimum write latency.
	[15:12]	RO	`4'b0`	Reserved.
`rd_lat_min`	[27:16]	RO	`12'b0`	Minimum read latency.
	[31:28]	RO	`4'b0`	Reserved.

Table 13 • CLOCK_FREQUENCY_DATA_WIDTH(Offset: 0x00_0028)

Name	Bits	Access	Default	Description
`data_width`	[15:0]	RO	`16'b0`	AXI data bus width.
`clock_frequency`	[29:16]	RO	`14'b0`	Clock frequency in MHz.
	[31:30]	RO	`2'b0`	Reserved.

AXI Packet Generator

The AXI data generator module generates AXI4 transactions of varying burst lengths and memory addresses. The generated data and address can be either sequential or random packets targeting sequential or random addresses using the AXI4 interface.

The generator can be configured to accommodate the memory subsystems as either NAP or DCI connections. The generator output can be stored in a FIFO for later checking using an AXI packet checker.

Parameters

Table 14 • AXI Packet Generator Parameters

Parameter Name	Default Value	Description
`LINEAR_PKTS`	0	Set to 1 for packets with linear counts.
`LINEAR_ADDR`	0	Set to 1 for linear addresses.
`TGT_ADDR_WIDTH`	0	Target address width.
`TGT_ADDR_PAD_WIDTH`	0	Target address padding.
`TGT_ADDR_ID`	0	Target address ID.
`TGT_DATA_WIDTH`	0	Target data width.
`MAX_BURST_LEN`	0	Maximum number of AXI packets in a burst.
`AXI_ADDR_WIDTH`	0	Width of axi_if address field.
`RAND_DATA_INIT`	0	Random value for data starting point.

Ports

Table 15 • AXI Packet Generator Ports

Name	Width	Direction	Description
`i_clk`	1	Input	Input clock.
`i_reset_n`	1	Input	Negative synchronous reset.
`i_start`	1	Input	Start sequence when applied.
`i_enable`	1	Input	Generate new packet when set.
`i_max_burst`	1	Input	Generate bursts of size `MAX_BURST_LEN` when set, otherwise increment bursts from 0 to `MAX_BURST_LEN` .

Name	Width	Direction	Description
`axi_if`	–	Input/Output	AXI4 interface.
`o_wr_error`	1	Output	Set when a writing error occurs.
`o_addr_written`	`TGT_ADDR_WIDTH`	Output	AXI write address output.
`o_len_written`	8	Output	Applied to data written into FIFO for checking algorithm.
`o_written_valid`	1	Output	Set high when data is successfully written from generator.

AXI Packet Checker

The AXI packet checker fetches packet addresses and lengths (usually stored in a FIFO) then performs a read from the specified address comparing the received data against its own internal random data generator which has the same sequence as the random data generator in the accompanying AXI packet generator. If the read data matches the expected data, the checker indicates the memory transaction was successful. If there is a mismatch between the read data and expected data, the `o_pkt_error` is set high, indicating a failed transaction has occurred.

Parameters

Table 16 • AXI Packet Checker Parameters

Parameter Name	Default Value	Description
`LINEAR_PKTS`	0	Set to 1 for packets with linear counts.
`TGT_ADDR_WIDTH`	0	Target address width.
`TGT_ADDR_PAD_WIDTH`	0	Target address padding.
`TGT_ADDR_ID`	0	Target address ID.
`TGT_DATA_WIDTH`	0	Target data width.
`AXI_ADDR_WIDTH`	0	Width of `axi_if` address field.
`RAND_DATA_INIT`	0	Random value for data starting point.
`NO_AR_LIMIT`	0	Gap to prevent GDDR/DDR pipeline reordering.

Ports

Table 17 • AXI Packet Checker Ports

Name	Width	Direction	Description
`i_clk`	1	Input	Input clock.
`i_reset_n`	1	Input	Negative synchronous reset.
`i_xact_avail`	1	Input	Asserted when a transaction is available to read.
`i_xact_addr`	`TGT_ADDR_WIDTH`	Input	Packet address received from FIFO.
`i_xact_len`	8	Input	Packet length received from FIFO.
`axi_if`	–	Input/Output	AXI interface.
`o_xact_read`	1	Output	Asserted when data is ready to be read.
`o_pkt_compared`	1	Output	Asserted at the end of each packet comparison.
`o_pkt_error`	1	Output	Asserted if a packet mismatch occurs.

AXI Performance Monitor

The AXI performance monitors are designed to measure the read and write data rates, along with latency, through the respective AXI interfaces. The monitor computes a moving average of the read and write bandwidth on an AXI data word level over a definable sample window.

In addition, the monitor calculates the average read and write latencies over a definable number of transactions. Further, the monitor measures the minimum and maximum values of both bandwidth and latency.

The monitor can be directly connected to the register control block.

Operating Modes

The monitor can be operated in various modes:

- Manual start and stop – to start and stop performance measurement, the `i_start` and `i_stop` signal need to be asserted. The result reflects the variations in bandwidth (BW) at the end of the measurement.
- Auto Start – measurement starts after first AXI read data is received, stop is still manually asserted.
- Auto Stop – measurement is manually started, but stops after a predefined number of clock cycles.
- Automated – measurement starts after first AXI read data is received and stops after the defined number of clock cycles.

Parameters

Table 18 • AXI Performance Monitor Parameters

Parameter Name	Default Value	Description
`BW_WINDOW_SIZE`	2048	Size of the moving window for the bandwidth calculation. Maximum of 2048 used as default. Power of 2 number <= 2048.
`LAT_AVERAGE_SIZE_EXP`	5	Quantity of the transactions used to calculate the average latency. The number of samples averaged is `(2 ^ LAT_AVERAGE_SIZE_EXP)`.
`STOP_COUNT`	0	If non-zero, stop measuring on this number of cycles. Ignore `i_stop`. Within the range of [0–2^15 – 1]
`AUTO_START`	0	If enabled, start counting when first read word is received. Ignore `i_start`.
`CLOCK_FREQ`	500	Operating clock frequency of the AXI interface in MHz. This value gets propagated to the register interface. Eases the computation of latency and bandwidth.
`DATA_WIDTH`	32	Data width of the AXI interface in bytes. This value gets propagated to the register interface. Eases the computation of bandwidth.

Ports

Table 19 • AXI Performance Monitor Ports

Signal Name	Width	Direction	Description
`i_clk`	1	Input	Clock input. Must be assigned to `axi_if` master clock.
`i_reset_n`	1	Input	Negative synchronous reset.
`i_start`	1	Input	Assert to start the performance measurement.
`i_stop`	1	Input	Assert to stop the performance measurement.
`i_counter_reset`	1	Input	Assert to reset the performance counters.
`axi_if`	–	Input	Direct connection to AXI interface signals in monitor mode.

Signal Name	Width	Direction	Description
`o_bw_results_rd`	32	Output	A combined 32-bit word that contains average, max and min read bandwidth: [31:30] – reserved. Returns `2'b0`. [29:20] – average read bandwidth. (1) [19:10] – maximum read bandwidth. (1) [9:0] – minimum read bandwidth. (1)
`o_bw_results_wr`	32	Output	A combined 32-bit word that contains average, max and min write bandwidth: [31:30] – reserved. Returns `2'b0`. [29:20] – average write bandwidth. (1) [19:10] – maximum write bandwidth. (1) [9:0] – minimum write bandwidth. (1)
`o_latency_results_current`	32	Output	A combined 32 bit word measuring the read and write latency of the latest transactions: [31:28] – reserved. Returns `4'b0`. [27:16] – latest read latency. [15:12] – reserved. Returns `4'b0`. [11:0] – latest write latency.
`o_latency_results_avg`	32	Output	A combined 32-bit word measuring average read and write latency: [31:28] – reserved, returns `4'b0`. [27:16] – average read latency. [15:12] – reserved, returns `4'b0`. [11:0] – average write latency.
`o_latency_results_max`	32	Output	A combined 32-bit word measuring maximum read and write latency: [31:28] – reserved, returns `4'b0`. [27:16] – maximum read latency. [15:12] – reserved, returns `4'b0`. [11:0] – maximum write latency.
`o_latency_results_min`	32	Output	A combined 32-bit word measuring minimum read and write latency: [31:28] – reserved, returns `4'b0`. [27:16] – minimum read latency. [15:12] – reserved, returns `4'b0`. [11:0] – minimum write latency.

Signal Name	Width	Direction	Description
`o_clock_freq_data_width`	32	Output	A combined 32-bit word that reports the clock frequency and the data width of the AXI port: [31:30] – `2'b00`. Indicates AXI performance monitor. [29:16] – clock frequency in MHz. Equivalent to `CLOCK_FREQ`. [15:0] – AXI data bus width. Equivalent to `DATA_WIDTH`. This register is intended to allow software to automatically calculate the bandwidth and latency values without requiring access to the RTL source.

Table Notes

1. Values are normalized to 10-bit values (1024).

Calculations

To calculate the bandwidth in Mbytes per second, the following formula can be used:

- Bandwidth (bytes per second) = `(o_bw_results_XX /1024) × (CLOCK_FREQ × AXI_DATA_WIDTH)`

To calculate the latency in ns the following formula can be used:

- Latency (ns) = `o_latency_results_YY × 1000/CLOCK_FREQ`

Clock Domain Considerations

All inputs and outputs are synchronous to the clock input, `i_clk`, which must be connected to the AXI interface clock that drives `axi_if`.

It is supposed that the AXI performance monitor is directly connected to the register control block. In these instances, it is necessary to consider the multiple clock domains between the monitor and the register block. The following methods are suggested to ensure data integrity when reading the result registers:

1. Connect the register control block to the same clock as `i_clk`. In many designs this might not be desirable if the register control block controls many functions of the design, or for timing closure reasons.
2. Start and stop the measurements using `i_start` and `i_stop`. Read the results after `i_stop` has been issued, when the output result registers are stable. This is the method used in the provided reference designs.
3. Insert clock domain crossing synchronizers (`ACX_SYNCHRONIZER`) on all AXI performance monitor register outputs (not required on `o_clock_freq_data_width` as this is a constant value). This results in an additional `6 registers × 32-bits × 2 flops per synchronizer = 384` registers.

Instantiation

The provided design consists of three separate modules. The AXI performance monitor combines two sub-modules, with the bandwidth measurement being performed by `axi_bw_monitor`, and the latency measurement by `axi_lat_monitor`. These two modules are then combined within `axi_performance_monitor` to create the combined bandwidth and latency monitor as described above. It is expected that the the full `axi_performance_monitor` would normally be instantiated, however if the user design requires it, it is possible to instantiate both of the sub-modules separately. Instantiation templates for all three instances are shown in the following example.

```
    // Instantiate AXI performance monitor
    axi_performance_monitor #(
         .BW_WINDOW_SIZE                      (1024),
         .LAT_AVERAGE_SIZE_EXP                (5)
         .STOP_COUNT                          (STOP_COUNT),
         .AUTO_START                          (AUTO_START),
         .CLOCK_FREQ                          (CLOCK_FREQ),
         .DATA_WIDTH                          (AXI_DATA_WIDTH)
    ) i_axi_performance_monitor_nap (
         // Inputs
         .i_clk                               (clk),
         .i_reset_n                           (reset_n),
         .i_start                             (start),
         .i_stop                              (stop),
         .i_counter_reset                     (counter_reset),

         // Interfaces
         .axi_if                              (axi_if),

         // Outputs
         .o_bw_results_rd                     (bw_results_rd),
         .o_bw_results_wr                     (bw_results_wr),
         .o_latency_results_current           (latency_results_current),
         .o_latency_results_avg               (latency_results_avg),
         .o_latency_results_max               (latency_results_max),
         .o_latency_results_min               (latency_results_min),
         .o_clock_freq_data_width             (clock_freq_data_width)
    );

    // Instantiate submodule AXI bandwidth monitor
    axi_bw_monitor #(
        .BW_WINDOW_SIZE                    (BW_WINDOW_SIZE),
        .STOP_COUNT                        (STOP_COUNT),
        .AUTO_START                        (AUTO_START),
        .CLOCK_FREQ                    (CLOCK_FREQ),
        .DATA_WIDTH                    (DATA_WIDTH)
    ) i_axi_bw_monitor_nap (
        // Inputs
        .i_clk                           (i_clk),
        .i_reset_n                       (i_reset_n),
        .i_start                         (i_start),
        .i_stop                          (i_stop),
        .i_counter_reset                 (i_counter_reset),

        // Interfaces
        .axi_if                          (axi_if),

        // Outputs
        .o_bw_results_rd                 (bw_results_rd),
        .o_bw_results_wr                 (bw_results_wr),
        .o_clock_freq_data_width         (clock_freq_data_width)
    );

    // Instantiate submodule AXI latency monitor
    axi_latency_monitor #(
        .LAT_AVERAGE_SIZE_EXP            (LAT_AVERAGE_SIZE_EXP),
        .STOP_COUNT                      (STOP_COUNT),
        .AUTO_START                      (AUTO_START),
        .CLOCK_FREQ                      (CLOCK_FREQ),
        .DATA_WIDTH                      (DATA_WIDTH)
    ) i_axi_latency_monitor_nap (
        // Inputs
        .i_clk                           (i_clk),
        .i_reset_n                       (i_reset_n),
        .i_start                         (i_start),
        .i_stop                          (i_stop),
        .i_counter_reset                 (i_counter_reset),

        // Interfaces
        .axi_if                          (axi_if),

        // Outputs
        .o_latency_results_current       (latency_results_current),
        .o_latency_results_avg           (latency_results_avg),
        .o_latency_results_max           (latency_results_max),
        .o_latency_results_min           (latency_results_min),
        .o_clock_freq_data_width         (clock_freq_data_width)
    );
```

Register Control Block

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 to control functionality, read status results, and measure performance both in simulation and at runtime. 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 20 • 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 Notes

† 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 21 • 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 Notes

† 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 22 • 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`.

Name	Direction (†)	Description

Table Notes

† 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

Instantiation of the register control block is shown in the following example.

```
// 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. Register address definition is shown in the following example.

```
// 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 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

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 in the following table.

Table 23 • Register Control Block Access Funtions

Function	Arguments (†)	Description
`<device_namespace>::nap_axi_write`	`NAP_SPACE` row col address value	Write a value to a register.

Function	Arguments [†]	Description
`<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: • In simulation – the FCU BFM asserts an error flag, which can be used in the testbench to indicate test failure. • In hardware – the verify function returns an error code (-1) which can be used to issue a fail message.

Table Notes

† 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 to the register array. This flow is shown in the following diagram.

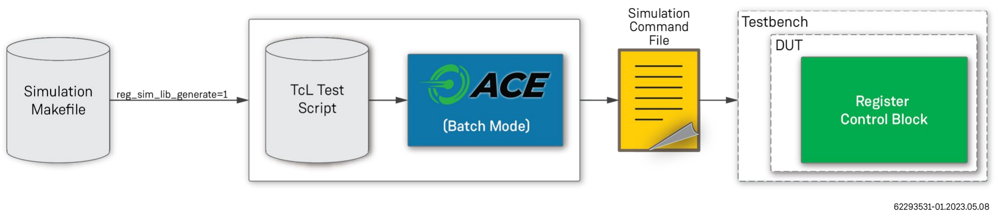
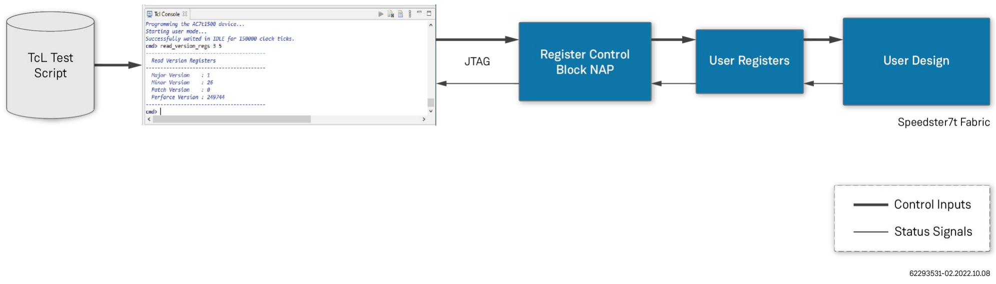


Figure 1 • Register Control Block Simulation Flow

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 following diagram.



```
Programming the AC7t1500 device...
Starting user mode...
Successfully waited in IDLE for 150000 clock ticks.
cmd> read_version_regs 3 5
--------------------------------------
  Read Version Registers
--------------------------------------
  Major Version    : 1
  Minor Version    : 26
  Patch Version    : 0
  Perforce Version : 249744
--------------------------------------
cmd>
```

Figure 2 • 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.

Top-Level Register Map

Table 24 • Speedster7t AC7t1400 and AC7t1500 Register Addresses

Target	Address	Bits	Access	Description
`GDDR_CONTROL_REG`	0x0000	[31:0]	RW	Currently not in use. Kept for future control such as resets.
`GDDR_STATUS_REG_NOC`	0x0004	[31:0]	RO	Status register for all NAP channels.
`GDDR_STATUS_REG_DCI`	0x0008	[31:0]	RO	Status register for all DC interface channels.
`TRAIN_CONTROL_REG`	0x000C	[31:0]	RO	Status register for GDDR6 training. Connected to `ACX_DEVICE_MANAGER` status output.
`NAP_REGS_BASE` (`GDDR6_0`)	0x0010	–	–	Refer to AXI Memory Channel and corresponding offsets.
`NAP_REGS_BASE` (`GDDR6_1`)	0x003C	–	–	

Target	Address	Bits	Access	Description
NAP_REGS_BASE(GDDR6_2)	0x0068	–	–	
NAP_REGS_BASE(GDDR6_3)	0x0094	–	–	
NAP_REGS_BASE(GDDR6_4)	0x00C0	–	–	
NAP_REGS_BASE(GDDR6_5)	0x00EC	–	–	
NAP_REGS_BASE(GDDR6_6)	0x0118	–	–	
NAP_REGS_BASE(GDDR6_7)	0x0144	–	–	
DCI_REGS_BASE(GDDR6_1)	0x0170	–	–	
DCI_REGS_BASE(GDDR6_2)	0x019C	–	–	
DCI_REGS_BASE(GDDR6_3)	0x01C8	–	–	
DCI_REGS_BASE(GDDR6_4)	0x01F4	–	–	
SCRATCH_REG	0x0220	[31:0]	RW	Scratch Register.

Table 25 • GDDR_STATUS_REG_NOC

Name	Bits	Access	Default	Description
pll_lock	0	RO	1'b0	Set if the PLLs of i_nap_clk are locked.
-	[3:1]	RO	3'b0	Reserved.
nap_fail	[11:4]	RO	8'b0	Indicates which NAP channels have failed transactions with bit 4 corresponding to GDDR6_0, bit 5 corresponding to GDDR6_1, etc.
nap_done	[19:12]	RO	8'b0	Indicates which NAP channels have completed all of their respective transactions as specified in NUM_TRANSFERS_REG with bit 12 corresponding to GDDR6_0, bit 13 corresponding to GDDR6_1, etc.
nap_running	[27:20]	RO	8'b0	Indicates which NAP channels are still running with bit 20 corresponding to GDDR6_0, bit 21 corresponding to GDDR6_1, etc.

Name	Bits	Access	Default	Description
-	[31:28]	RO	`4'b0`	Reserved.

Table 26 • GDDR_STATUS_REG_DCI

Name	Bits	Access	Default	Description
`pll_gddr_NW_dci_lock`	0	RO	`1'b0`	Set if the PLL for the west GDDR6 DC interface channels is locked.
`pll_gddr_NE_dci_lock`	1	RO	`1'b0`	Set if the PLL for the east GDDR6 DC interface channels is locked.
-	[3:2]	RO	`2'b0`	Reserved.
`dci_fail`	[7:4]	RO	`4'b0`	Indicates which DC interface channels have failed transactions with bit 4 corresponding to `GDDR6_1`, bit 5 corresponding to `GDDR6_2`, bit 6 corresponding to `GDDR6_5`, and bit 7 corresponding to `GDDR6_6`.
`dci_done`	[11:8]	RO	`4'b0`	Indicates which DC interface channels have completed all of their respective transactions as specified in `NUM_TRANSFERS_REG` with bit 8 corresponding to `GDDR6_1`, bit 9 corresponding to `GDDR6_2`, bit 10 corresponding to `GDDR6_5`, and bit 11 corresponding to `GDDR6_6`.
`dci_running`	[15:12]	RO	`4'b0`	Indicates which DC interface channels are still running with bit 12 corresponding to `GDDR6_1`, bit 13 corresponding to `GDDR6_2`, bit 14 corresponding to `GDDR6_5`, and bit 15 corresponding to `GDDR6_6`.
-	[31:16]	RO	`16'b0`	Reserved.

Table 27 • TRAIN_CONTROL_REG

Name	Bits	Access	Default	Description
`ip_status[0]`	0	RO	`1'b0`	Set when startup is complete and successful.
`ip_status[1]`	1	RO	`1'b0`	Set when startup is complete.

Name	Bits	Access	Default	Description
-	[31:2]	RO	`30'b0`	Reserved.

Test Structure Using the 2D NoC-GDDR6 Interface

The following diagram shows the AXI memory channel modules interaction with the GDDR6 subsystems via the 2D NoC on the Speedster7t AC7t1400 and AC7t1500 FPGAs. The number of 2D NoC or DC interface-enabled GDDR6 subsystems can be configured and the target controller ID for each can be set in the top-level module.

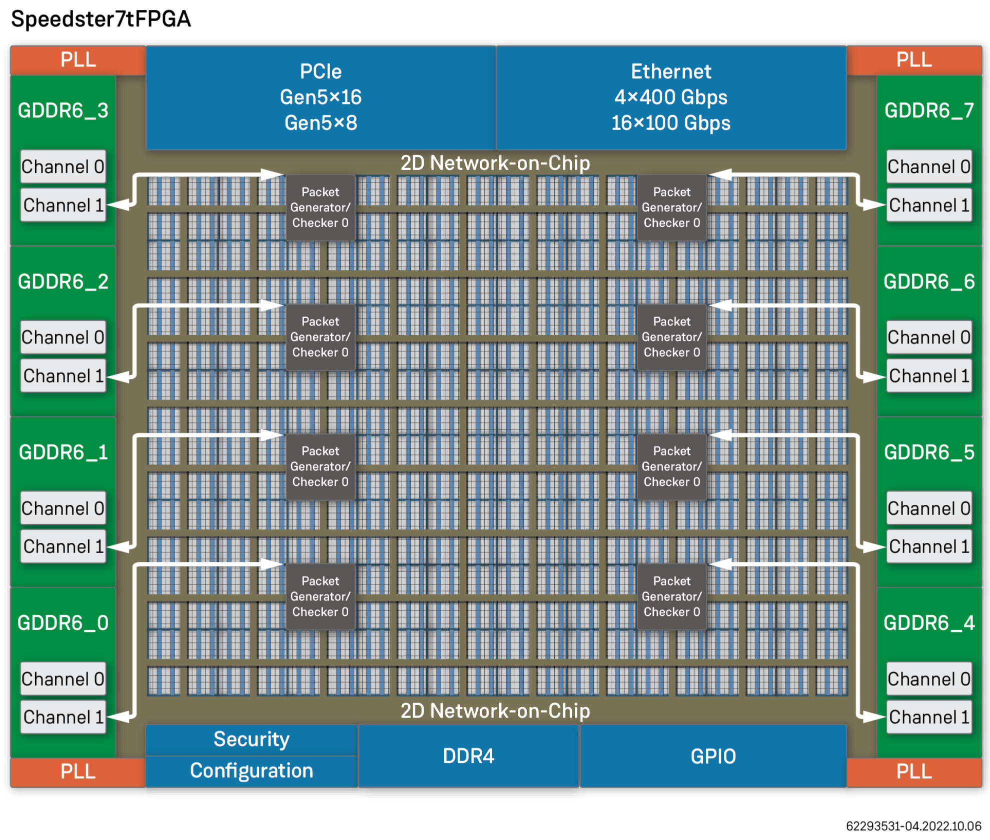


Figure 3 • Speedster7t AC7t1400 and AC7t1500 GDDR6 Generator and Checker Logic Via 2D NoC

Addressing

Each packet generator/checker logic block is bound to a NAP to interface with an associated target GDDR6 subsystem. The values in the table below are used in the controller ID field of the NoC address scheme for GDDR6 to indicate the target GDDR6 endpoint for each NAP to send and receive memory transactions. More details on the NAP address scheme can be found in the *Speedster7t 2D Network on Chip User Guide* (UG089).

Table 28 • GDDR6 Subsystem Control IDs on Speedster7t AC7t1400 and AC7t1500

GDDR6 Controller	Control ID [36:33]
`GDDR6_0`	Channel 0 – `4'b1100`
	Channel 1 – `4'b1101`
`GDDR6_1`	Channel 0 – `4'b0100`
	Channel 1 – `4'b0101`
`GDDR6_2`	Channel 0 – `4'b0000`
	Channel 1 – `4'b0001`
`GDDR6_3`	Channel 0 – `4'b1000`
	Channel 1 – `4'b1001`
`GDDR6_4`	Channel 0 – `4'b1111`
	Channel 1 – `4'b1110`
`GDDR6_5`	Channel 0 – `4'b0111`
	Channel 1 – `4'b0110`
`GDDR6_6`	Channel 0 – `4'b0011`
	Channel 1 – `4'b0010`
`GDDR6_7`	Channel 0 – `4'b1011`

GDDR6 Controller	Control ID [36:33]
	Channel 1 - `4'b1010`

To construct the GDDR6 memory addresses according to the NoC address map, these Control ID values are defined as 9-bit values in the `GDDR6_ID_NOC_CH1` local parameter in the top-level RTL file.

```
// GDDR6 target address ID. Pages are defined in 2D NoC User Guide, Address Mapping
// Defined as 9 bit field. 9th bit(LSB) controls channel selection. All 2D NoC
interfaces are set to channel 1
localparam [71:0] GDDR6_ID_NOC_CH1 = {9'd10, 9'd2, 9'd6, 9'd14, 9'd9, 9'd1, 9'd5,
9'd13};
```

Note

The GDDR6 controllers on the east side use even addresses for channel 1, whereas the west side uses odd addresses. Therefore, four `GDDR6_ID_NOC_CH1` values are even, and four are odd.

Test Structure Using the DC Interface

The DC Interface is only applicable to Speedster7t AC7t1400 and AC7t1500 FPGAs

The following diagram shows the AXI memory channel modules interaction with the GDDR6 subsystems via the DC interface. There are four GDDR6 subsystems, namely GDDR6 1, 2, 5 and 6, that support the direct-to-fabric connect interface as shown in the figure. The desired subsystem can be specified as shown in the table GDDR6 Subsystem Control ID. In this reference design, all DC interfaces are connected to channel 0 of the GDDR6 subsystems. For further details on the DC interface connection, refer to the *Speedster7t GDDR6 User Guide* (UG091).

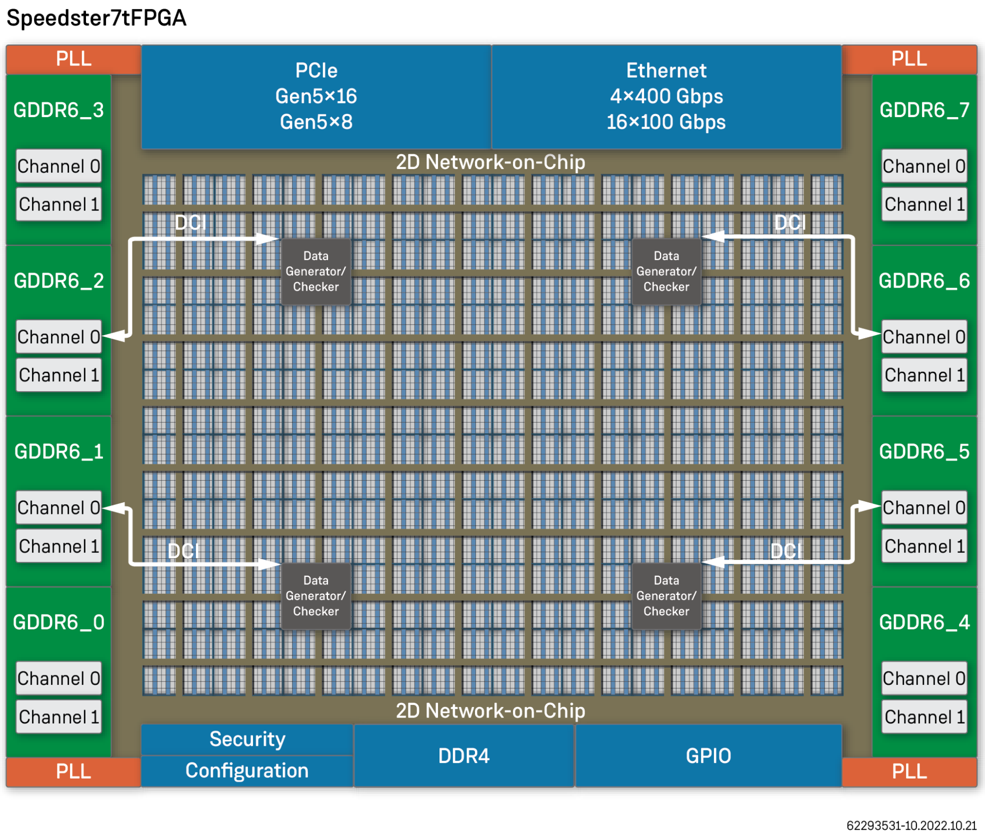


Figure 4 • Speedster7t AC7t1400 and AC7t1500 Generator/Checker Logic Via DC Interface

Simulation

This design supports simulation in one of two modes: bus-functional model (BFM) mode or standalone mode. Full-chip RTL simulation is not available and standalone mode simulation is not supported for Riviera .

- In BFM mode, the testbench incorporates BFMs for the GDDR6 subsystem and memories.
- In standalone mode, only the fabric RTL is simulated along with behavioral modules of the 2D NoC. DC Interface channels are not simulated.

Ports and Parameters

Top-Level Parameters

The GDDR6 reference design top-level RTL module has the following parameters:

Table 29 • Speedster7t AC7t1400 and AC7t1500 GDDR6 Reference Design Top-Level RTL Parameters

Parameter Name	Default Value	Description
`GDDR6_NOC_CONFIG`	`8'b11111111`	Signifies which of the 8 GDDR6 subsystems are enabled for 2D NoC interface testing (1 = enabled, 0 = disabled): bit[0] – `GDDR6_0` bit[1] – `GDDR6_1` … bit[7] – `GDDR6_7`
`GDDR6_DCI_CONFIG`	`4'b1111`	Signifies which of the 4 GDDR6 subsystems are enabled for DC interface testing (1 = enabled, 0 = disabled): bit[0] – `GDDR6_1` bit[1] – `GDDR6_2` bit[2] – `GDDR6_5` bit[7] – `GDDR6_6`

Memory Channel Parameters

The NAP and DC interface channels can support up to 16-beat maximum and 64-beat maximum bursts, respectively. Therefore, when instantiating the memory channel, the `MAX_BURST_LEN` parameter must be declared accordingly.

Top-Level Ports

The GDDR6 reference design top-level RTL module has the following I/O ports:

Table 30 • Speedster7t AC7t1400 and AC7t1500 GDDR6 Reference Design Top-Level RTL Ports

Signal Name	Width	Description
`i_nap_clk`	1	Clock driving the logic interfacing with the NAP running at 500 MHz.
`i_reg_clk`	1	Clock driving the logic interfacing with the register control block running at 200 MHz.
`i_adm_clk`	1	Clock driving the `ACX_DEVICE_MANAGER` soft IP running at 100 MHz.

Signal Name	Width	Description
`led_l`	8	LED output bus to determine status of training and test which are tied to GPIO LED outputs on the VectorPath S7t-VG6 and 815 cards (only applicable to AC7t1500).
`gddr6_*_dc0_clk`	1	Clock driving the logic interfacing with the DC interface.
`gddr6_*_dc0_*`	–	DC interface AXI signals.
`pll_lock`	1	Lock signal for clock `i_nap_clk`.
`pll_reg_adm_lock`	1	Lock signal for clocks `i_reg_clk` and `i_adm_clk.`
`pll_gddr_NE_nap_lock`	1	Lock signal for PLLs driving eastern GDDR6 NAP channels
`pll_gddr_NE_dci_lock`	1	PLL lock signal for DC interface channels on GDDR subsystems 5 and 6.
`pll_gddr_NW_nap_lock`	1	Lock signal for PLLs driving western GDDR6 NAP channels
`pll_gddr_NW_dci_lock`	1	PLL lock signal for DC interface channels on GDDR subsystems 1 and 2.

Design Considerations

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

Clocking

The source `acxip` files for the PLLs are located in the `/src/acxip` directory. The clocks for the PLLs needed to drive the design are detailed in the following tables.

Table 31 • Reference Design Clocks on Speedster7t AC7t1400 and AC7t1500

Clock	Direction	Frequency (MHz)	Description
pll			
`fpga_fab_clk_7`	Input	100	External reference input clock for the PLL.
`i_nap_clk`	Output	500	Clock for NAP memory channels.
pll_reg_adm			
`fpga_fab_clk_7`	Input	100	External reference input clock for `pll_reg_adm`.

Clock	Direction	Frequency (MHz)	Description
`noc_clk` (†)	Output	200	2D NOC Reference clock.
`i_reg_clk`	Output	200	Clock for register control block logic.
`i_adm_clk`	Output	100	Clock for `ACX_DEVICE_MANAGER` soft IP.
pll_gddr_NE_dci			
`fpga_fab_clk_2`	Input	400	External reference input clock for `pll_gddr_NE_dci`.
`gddr6_5/6_dci_clk` (1)	Output	275	Reference clock to the corresponding GDDR6 direct-connect interface. This clock is the source of the GDDR6 DC interface `gddr6_*_dc0_clk` input to the design.
pll_gddr_NE_nap			
`fpga_fab_clk_2`	Input	400	External reference input clock for `pll_gddr_NE_nap`.
`gddr_ctlr_clk_NE` (1)	Output	875 (2), 1000 (3)	Reference clock to GDDR6 controller subsystems on the east side of the FPGA.
pll_gddr_NW_dci			
`fpga_fab_clk_4`	Input	10	External reference input clock for `pll_gddr_NW_dci`.
`gddr6_1/2_dci_clk` (1)	Output	275	Reference clock to the corresponding GDDR6 direct-connect interface. This clock is the source of the GDDR6 DC interface `gddr6_*_dc0_clk` input to the design.
pll_gddr_NW_nap			
`fpga_fab_clk_4`	Input	10	External reference input clock for `pll_gddr_NW_nap`.
`gddr_ctlr_clk_NW` (1)	Output	875 (2), 1000 (3)	Reference clock to GDDR6 controller subsystems on the West side of the FPGA.

Clock	Direction	Frequency (MHz)	Description

Table Notes

1. These clock outputs connect directly from the reference PLL to their respective destinations. The clocks are not routed via the FPGA fabric so are not available to the user logic.
2. The VectorPath S7t-VG6 GDDR6 controller clock targets 875 MHz
3. The VectorPath 815 GDDR controller clock targets 1000 MHz.

Clock Domains

The design consists of two parallel traffic generator and checker blocks, one for each NAP interface and one for each DC interface. The clock domains are listed in the following table.

Table 32 • Clock Domains on Speedster7t AC7t1400 and AC7t1500

Source (†)	Description
`i_nap_clk`	NAP interface clock source.
`gddr6_*_dc0_clk`	Corresponding DC interface clock domain source.
`i_reg_clk`	Register control block clock domain source.
`i_adm_clk`	Achronix Device Manager clock domain source.

table note

† The frequencies of these clocks are set by their respective PLLs. For simulation, the clock frequencies are defined by parameters in the top-level testbench file, `/src/tb/tb_gddr_ref_design.sv`. These clocks are
asynchronous to one another. Therefore, clock domain crossing circuits are employed for signals interfacing between these two domains. In addition, they are defined as separate clock groups in `/src/constraints/ace_constraints.sdc and /src/constraints/synplify_constraints.sdc`.

Note

For consistency, maintain the same frequency/period values in both testbench and synthesis constraint files.

Resets

The design makes use of the PLL lock signals for reset handling.

Good design practice requires that resets are carefully synchronized and processed. The reference design uses multiple instances of the `reset_processor_v2` module which combines multiple sources, correctly synchronizes them to the specified clock domains and subsequently pipelines the resets to allow for fan-out control and re-timing. The output from the `reset_processor_v2` module is a synchronous reset signal per clock domain.

The design consists of five instances of the reset_processor:

- One instance is for the 2D NoC circuits, which uses the PLL lock signals as asynchronous resets and generates a reset, `reg_rstn`, that is synchronous to `i_reg_clk`. This output also gets synchronized to the NAP clock domain.
- Instances are instantiated for each DC interface, resulting in four instances in total. Each instance also uses the PLL lock signals as asynchronous resets, along with their respective DC interface, `dci_input_rstn`, producing a local signal, `dci_output_rstn`.

GDDR6 Channel Assignments for 2D NoC and DC Interfaces

The reference design is configured to use channel 1 for all of the 2D NoC interfaces and channel 0 for all of the DC interfaces, so that the various AXI memory channels do not encounter conflict due to writing different values to the same memory space. However, it may be desirable in a user design for both of these interfaces to access the same address space in the memory, i.e., the 2D NoC writing to a memory location, and the DC interface reading from the same address or vice versa.

For the DC interfaces, only the `gddr6_*_dc0` signals are driven.

BMC Interface Module

VectorPath S7t-VG6 cards revision 3 or greater with BMC firmware 1.4.0 or greater require an instantiation of the `BW_BMC_IF` module within the user-generated fabric design. VectorPath 815 cards also require instantiation of this module. This fabric module allows continuous monitoring of the temperature through the Bittware SDK while the VectorPath S7t-VG6/815 card's JTAG connection is open.

The `BW_BMC_IF` also optionally instantiates the I^{2}C Initiator module which controls the communication between the Speedster7t FPGA and QSFP/QSFP-DD ports on the VectorPath S7t-VG6/815 cards.

The `BW_BMC_IF` module instantiates a NAP_AXI_SLAVE/MASTER pair at column 9, row 2 of the fabric as the default location. Therefore, it is important for VectorPath S7t-VG6/815 card fabric designs to not place any other components in that region. However, if the design is not intended to provide the host access to the `BW_BMC_IF` module over the USB cable, then these NAPs can be placed anywhere within the fabric. The `BW_BMC_IF` module also requires another NAP_AXI_MASTER to be available in the fabric if the I^2C initiator module is instantiated.

This module is placed in the `/src/include` directory within the design. The following line of code must be placed in the top-level RTL file:

```
// Include the BittWare BMC and I2C interface
`include "BW_BMC_IF.svp"
```

Parameters

Table 33 • BW_BMC_IF Module Parameters

Parameter Name	Default Value	Description
`BMC_NAP_ROW`	2	Row for BMC NAP placement.
`BMC_NAP_COLUMN`	9	Column for BMC NAP placement.
`INCLUDE_I2C`	0	Set to include I^2C module.
`I2C_NAP_ROW`	f	Row for I^2C initiator NAP.
`I2C_NAP_COLUMN`	f	Column for I^2C initiator NAP.
`I2C_CLK_WIDTH`	1000	Period of I^2C interface clock input, `i2c_clk_in`, in `ACLK` cycles.

Ports

Table 34 • BW_BMC_IF Ports

Name	Width	Direction	Description
`i_clk`	1	Input	100 MHz clock.

Name	Width	Direction	Description
`i_rstn`	1	Input	Negative sense reset.
`i_timestamp`	32	Input	Optionally add timestamp to build.
`i_fw_version`	32	Input	Optionally add a version register to the build.
`i_adm_status`	32	Input	Status from the Achronix Device Manager.
`i_fpga_avr_rxd`	1	Input	Input from the BMC.
`o_fpga_avr_txd`	1	Output	Output to the BMC.
`o_fpga_avr_txd_oe`	1	Output	Output enable for `o_fpga_avr_txd`.
`i_fpga_sys_scl_in` (2)	1	Input	I^2C interface clock input.
`o_fpga_sys_scl_out` (2)	1	Output	I^2C interface clock output.
`o_fpga_sys_scl_oe` (2)	1	Output	I^2C interface clock output enable.
`i_fpga_sys_sda_in` (2)	1	Input	I^2C interface data input.
`o_fpga_sys_sda_out` (2)	1	Input	I^2C interface data output.
`i_fpga_sys_sda_oe` (2)	1	Input	I^2C interface data output enable.
`o_fpga_i2c_req_l` (1)(2)	1	Output	I^2C interface request signal from the Speedster7t FPGA.
`o_fpga_i2c_req_l_oe` (2)	1	Output	I^2C interface request enable.
`i_fpga_i2c_mux_gnt` (1)(2)	1	Input	I^2C interface grant signal from the BMC.

Table Notes

1. `o_fpga_i2c_req_l` and `i_fpga_i2c_mux_gnt` are discrete signals that control a shared I^{2}C bus between the FPGA and the BMC. The `o_fpga_i2c_req_l` signal is used by the FPGA to request access to the I^{2}C bus. When `o_fpga_i2c_req_l` is asserted, the BMC grants access to the bus by asserting the `i_fpga_i2c_mux_gnt` signal. When the FPGA is finished with the bus, it must deassert the `o_fpga_i2c_req_l` signal.
2. These pins are not instantiated when the parameter `INCLUDE_I2C` is not set to 1.

Number of GDDR6 Subsystems

As shown in the table, **GDDR6 Reference Design Top-Level RTL Parameters**, parameters can be used to configure the number of GDDR6 subsystems used in this design. The default configuration is to enable all GDDR6 subsystems.

If the design is configured to use only some of the GDDR6 subsystems, the default placement constraint file, `/src/constraints/ace_placements.pdc`, must also be edited to only place the GDDR6 subsystems used.

Continuous and Non-Continuous Transactions

To run the long-term stability test, make use of the script stored in the `ac7t1500/demo/scripts/gddr6_ref_design_long_term.tcl` file. This script continuously performs transactions on the memory channels for 24 hours. Every hour, this script displays status messages on each memory channel, indicating if any of them have failed transactions. This test should only be run on silicon.

To perform the non-continuous test, make use of the script stored in the `ac7t1500/demo/scripts/gddr6_ref_design.tcl` file. This script performs 16 million transactions on each of the memory channels. During simulation, this script is also in use. However, only 300 transactions are performed on each of the memory channels instead.

GDDR6 ACXIP Configurations

VectorPath 815 Accelerator Card

The GDDR6 banks are configured to interface with MT61K512M32-E Micron devices when targeting the Speedster7t AC7t1500 (production silicon) FPGA on the VectorPath 815 card. The design utilizes two PLL banks to configure the GDDR6 controller clocks at 1000 MHz and DC interface clocks at 275 MHz. The memory devices are configured to operate at a data rate of 16 Gbps in ×8 clamshell mode.

VectorPath 815 CLKIO and GPIO ACXIP Configurations

This design is compatible with the Achronix VectorPath 815 accelerator card. The provided bitstream in `/ac7t1500_vp815/demo/bitstream` can be downloaded to the VectorPath 815 card, and the test executed using the script in `/ac7t1500_vp815/demo/scripts`. In addition, the design can be rebuilt to run on the card.

The design includes the IP configuration (.acxip) files in `/ac7t1500_vp815/src/acxip`, which specify the pinout of the VectorPath 815 card, including clock and GPIO signals. Included files for CLKIO and GPIO are shown in the following table.

Table 35 • VectorPath 815 CLKIO and GPIO IP Configuration Files

File (†)	Ports Connected	Direction	Description
`vp815_clkio_ne.acxip`	`pcie_perst_l`	Input	PCIe PERST
	`fpga_fab_clk_2`	Input	400 MHz differential
`vp815_clkio_nw.acxip`	`fpga_fab_clk_3`	Input	
	`fpga_fab_clk_4`	Input	10 MHz differential
`vp815_clkio_se.acxip`	`fpga_fab_clk_5`	Input	400 MHz differential
	`fpga_fab_clk_6`	Input	50 MHz differential
`vp815_clkio_sw.acxip`	`fpga_fab_clk_7`	Input	100 MHz differential
	`fpga_fab_clk_1`	Input	200 MHz differential
`vp815_gpio_n_b0.acxip`	`exp_gpio_fpga[7:0]`	Output	General-purpose I/O
	`ext_gpio_oe_l`	Output	GPIO output enable
	`led_oe_l`	Output	Enable LED outputs
`vp815_gpio_n_b1.acxip`	`ext_gpio_dir[7:0]`	Output	GPIO signal direction
	`led_l[5:4]`	Output	Board LEDs. active low
`vp815_gpio_n_b2.acxip`	`led_l[7:6, 3:0]`	Output	
`vp815_gpio_s_b0.acxip`	`fpga_i2c_mux_gnt`	Input	I^2C interface

File (†)	Ports Connected	Direction	Description
	`fpga_i2c_req_l`	Output	
	`fpga_avr_txd`	Output	AVR interface
	`fpga_avr_rxd`	Input	
	`fpga_ftdi_txd`	Output	FTDI interface
	`fpga_ftdi_rxd`	Input	
	`fpga_rst_l`	Input	Driven from BMC controller
	`irq_to_avr`	Output	Interrupt
	`recov_clk_0`	Output	Connect to inputs on clock chip on VP815
	`qsfp_int_fpga_l`	Input	QSFP interface. Active low
`vp815_gpio_s_b1.acxip`	`u1pps_in`	Input	Provide PPS connectivity to the front panel
	`u1pps_1`	Inout	
	`u1pps_2`	Inout	
	`u1pps_1_dir`	Output	
	`u1pps_2_dir`	Output	
	`u1pps_en_l`	Output	
	`freq_dec`	Output	Tunes PLL clock frequency on VP815 clock device
	`freq_inc`	Output	
	`clk_gpio0`	Output	Connect to inputs on clock chip on VP815
	`clk_gpio1`	Output	
`vp815_gpio_s_b2.acxip`	`recov_clk_1`	Output	
	`fpga_sys_scl`	Inout	I^{2}C interface
	`fpga_sys_sda`	Inout	

File [†]	Ports Connected	Direction	Description

Table Notes

† If porting a user design to the VectorPath 815 card, it is recommended to use these files to define the interface to the card.

Simulation Configuration

Simulation Modes

This design supports the full-chip bus functional model (BFM) simulation mode using the VCS, Questa, or Riviera simulation tools. The standalone mode of simulation is supported using the VCS or Questa simulation tools. Check the simulation makefile for which modes are supported. The default simulation flow used by the makefile is the full-chip BFM mode. To run the simulation in standalone mode, either append `FLOW=STANDALONE` to the simulation `make` command:

```
> make (other options) FLOW=STANDALONE
```

...or edit the flow variable in the makefile to ensure that all future simulation runs use RTL mode:

```
FLOW := STANDALONE
```

GDDR6 BFM (BFM Mode Only)

This module is the bus functional model (BFM) for the entire GDDR6 subsystem that mimics the GDDR6 controller, PHY and memory setup. The BFM delay models are also designed to account for delays in each of these corresponding blocks. Each of the eight instantiated GDDR6 BFMs for the 2D NoC use a unique NAP target address that indicates the NAP/2D NoC structure where each of the data packets need to be routed. The target address (indicated at the top-level design file) also has a designated bit to indicate which of the two channels in the GDDR6 subsystem is being used for the transactions. The GDDR6 BFM is configured to run at a data rate of 16 Gbps in this design.

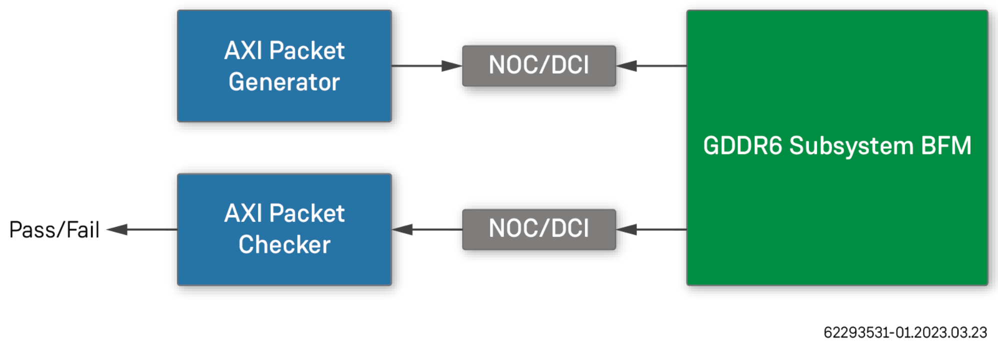


Figure 5 • Single GDDR6 Subsystem 2D NoC/DC Interface Test Structure Block Diagram

GDDR6 Data Rate

The simulation GDDR6 data rate can also be defined by enabling the corresponding lines in the following file:

- BFM mode – `/sim/<simulator>/system_files_bfm.f`

> **Note**
>
> FULLCHIP RTL mode is not supported in this release.

Installing the Design

Downloading

Designs can be downloaded via the Achronix Support website (an Achronix support account is required). There are two knowledge base articles which in total contain the demonstration, reference and tutorial design downloads:

- How do I Download Demonstration and Reference Designs? contains designs for reference and designs ported to the VectorPath card.
- How do I Download Tutorials? for designs focused on the tool flow, using ACE and Synplify Pro.

> **Note**
>
> Login is required to access the above articles. To gain access, see How Do I Register for an Achronix Support Account?.

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 the following:

- All source code
- Scripts to build the design
- Simulation script files
- 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 build scripts and flows are natively designed for Linux and have been built and tested using Ubuntu 22.04LTS. The batch build flow uses a makefile and has been tested with Bash.

Windows

Scripted simulation or implementation flows under Windows 10 require installing one of the following:

- A Linux environment under Windows, such as **www.cygwin.com**. The installation should include a Tcl interpreter and the `make` executable. When using Cygwin, which has a number of differences from native Linux, please refer to the Knowledge Base article, **How Do I Set Up My Cygwin Environment to Run the Achronix Tool Suite?** available at support.achronix.com.
- A Tcl interpreter and a `make` executable for Windows, if not included in cygwin. 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 must be installed in a directory without spaces in the path name. Examples of suitable paths are:

- Windows – `C:\Achronix\ACE\x.y.z\Achronix_CAD_Environment`
- Linux (single-user) – `/home/<user_id>/ACE/Achronix-linux`
- Linux (shared) – `/opt/ACE/Achronix-linux`

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/ACE/10.3.1/Achronix_CAD_Environment/Achronix
```

Tool Versions

All designs are compatible with the following tools:

Table 36 • Minimum Tool Versions

Software	Version (†)
ACE	10.3.1
Device Simulation Model	10.2
Synplify Pro	V-2023.09X
Mentor Questa	10.7c-1
Synopsys VCS	P-2019.06
Aldec Riviera Pro	2021.10-x64

Table Notes

† Tool versions are the minimum required. It is anticipated that more recent versions should also be compatible with this design.

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 (i.e., where the ace executable is installed). This variable is used by both synthesis and simulation to correctly locate the ACE library files. See Operating System for additional details.

SYNPLIFY_HOME

From versions 10.0 and above, ACE supports integrated synthesis; using Synplify Pro within the ACE design environment. To support this, it is necessary to define the location of Synplify Pro. This is defined using the `SYNPLIFY_HOME` environment variable. The `SYNPLFY_HOME` variable points to the installation directory of Synplify Pro. Within the `SYNPLIFY_HOME` directory should be the `/bin` directory that contains the Synplify Pro executable.

Note

The `SYNPLIFY_HOME` variable must be set in the environment before ACE is started

```
# Example of checking and then setting SYNPLIFY_HOME within Linux Bash shell.
# Please check your particular shell for how to set environment variables

bash-4.2$ echo $SYNPLIFY_HOME                                                    #
Check if variable is set.
                                                                                 # A
blank response indicates that the variable is not set
bash-4.2$ which synplify_pro
/opt/synopsys/SynplifyPro/V-2023.09X/bin/synplify_pro                            #
Find path to Synplify Pro installation
bash-4.2$ export SYNPLIFY_HOME=/opt/synopsys/SynplifyPro/V-2023.09X              # Set
the variable to the installation directory
bash-4.2$ echo $SYNPLIFY_HOME                                                    #
Check variable is set correctly
/opt/synopsys/SynplifyPro/V-2023.09X
```

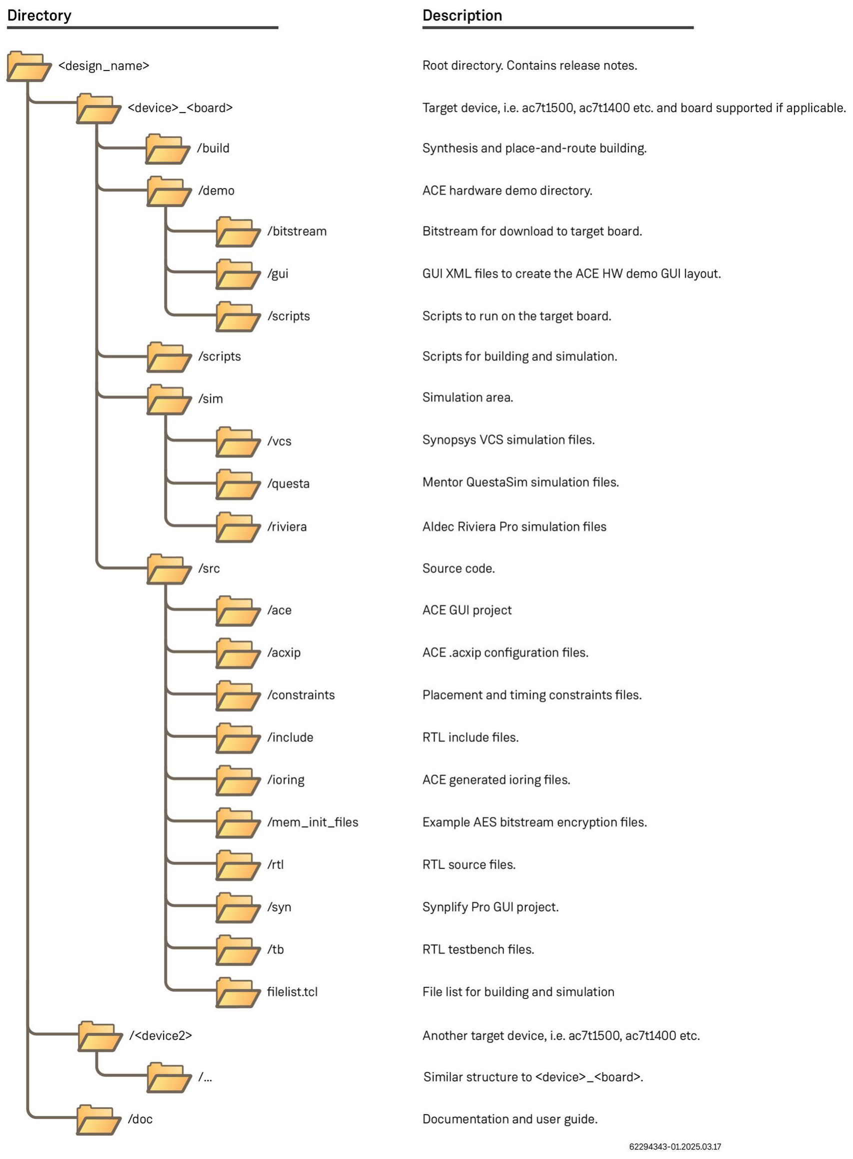
Directory Structure

The design has a directory structure that allows for easy navigation and separation of source and generated files. There is a top-level directory for each supported device. 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.

If it is desired to change the device selected within a design, then please refer to the following Knowledge Base article : How do I Convert an Achronix Speedster Customer Design to a New Device?

Note

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



Directory	Description
<design_name>	Root directory. Contains release notes.
<device>_<board>	Target device, i.e. ac7t1500, ac7t1400 etc. and board supported if applicable.
/build	Synthesis and place-and-route building.
/demo	ACE hardware demo directory.
/bitstream	Bitstream for download to target board.
/gui	GUI XML files to create the ACE HW demo GUI layout.
/scripts	Scripts to run on the target board.
/scripts	Scripts for building and simulation.
/sim	Simulation area.
/vcs	Synopsys VCS simulation files.
/questa	Mentor QuestaSim simulation files.
/riviera	Aldec Riviera Pro simulation files
/src	Source code.
/ace	ACE GUI project
/acxip	ACE .acxip configuration files.
/constraints	Placement and timing constraints files.
/include	RTL include files.
/ioring	ACE generated ioring files.
/mem_init_files	Example AES bitstream encryption files.
/rtl	RTL source files.
/syn	Synplify Pro GUI project.
/tb	RTL testbench files.
filelist.tcl	File list for building and simulation
/<device2>	Another target device, i.e. ac7t1500, ac7t1400 etc.
/...	Similar structure to <device>_<board>.
/doc	Documentation and user guide.

Figure 6 • Design Directory Structure

The <device> sub-directories with a suffix such as "s7tvg6" indicate that the design files in the sub-directory are targeted to a specific board. If the <device> sub-directory does not have a suffix appended to the sub-directory name, the source files residing in it may not target any specific board.

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 in the following table.

Table 37 • 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.tcl`	Controls ACE settings such as implementation options, flow mode, speed grade and reporting of unconstrained paths. Also used to define Synplify Pro options for ACE-driven integrated synthesis flow.
`ace_placements.pdc` (†)	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 options for standalone synthesis flow, such as top module. Creates synthesis specific parameters, generics, and defines.

File Name	Usage

Table Notes

† 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 ACE I/O Designer Toolkit generates constraint files specific to the interface between the fabric core and the I/O ring containing the interface subsystems. These constraint files can optionally be re-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`. An explanation of some of the more useful files is below.

Table 38 • I/O 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_delays_<speed_grade>_<temperature>_<fast/slow>.sdc`	Timing constraints from the I/O ring peripheral signals to the core fabric
`<design_name>_ioring_util.xml`	Used by ACE to generate a combined utilization report including the fabric and I/O ring resources.
`<design_name>_ioring_power.xml`	Used by ACE to generate an estimated power report for the design.

Building the 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 building the design, 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 `$ACE_INSTALL_DIR` to the path variable.
- Synplify Pro should be in the environment path.
- If using ACE integrated synthesis (ACE 10.0 onward) the environment variable `$SYNPLIFY_HOME` must point to the Synplify Pro installation directory (the level which contains the `/bin` directory) or the environment variable `$ACX_SYNPLIFY_TOOL_PATH` must be set to point to the synplify_pro executable.

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. Use the following steps 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.

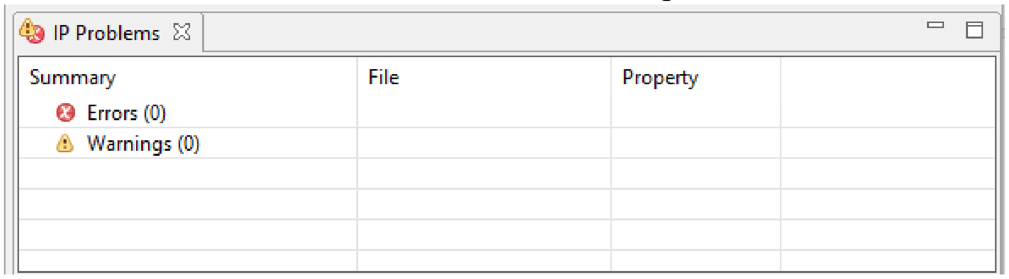
I/O Ring Batch Flow

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. Ensure that, in the IP Problems window, there are no errors or warnings. If there are, then resolve those first.



Summary	File	Property	
Errors (0)			
Warnings (0)			

Figure 7 • IP Problems Window

3. To prompt I/O ring design file generation, use one of the GUI methods below:
   - Open any of the project hardened subsystem configuration files located under the **Project** → **IP** selection. In the open IP configuration file, select **Generate**.
   - Click the **Generate I/O ring Design Files** button in the toolbar.

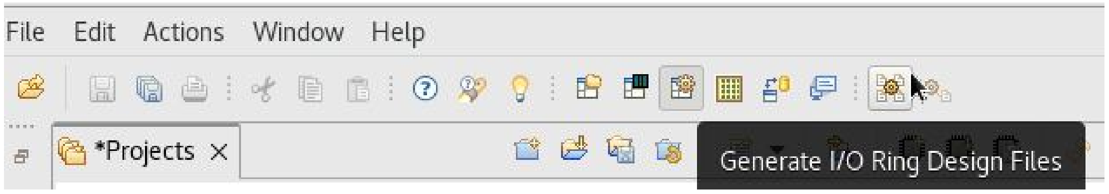
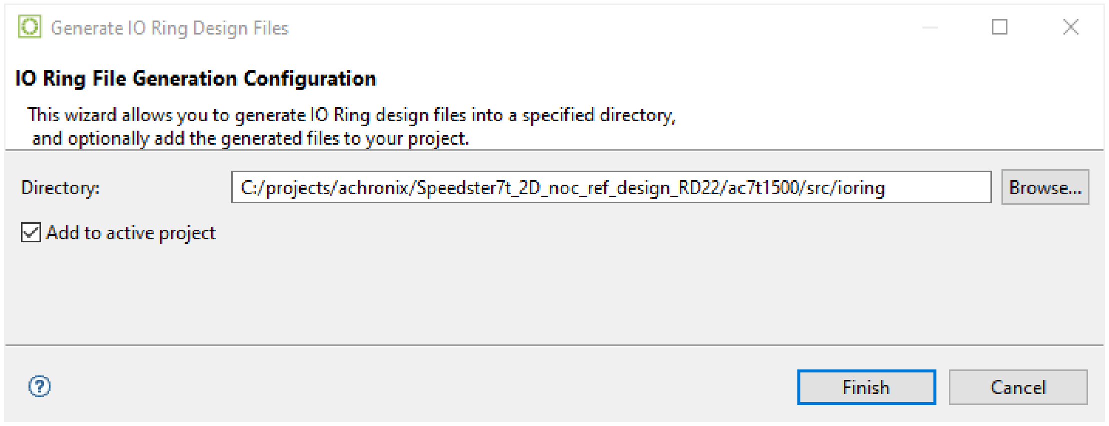


Figure 8 • Toolbar Button toGenerates I/O Ring Design Files

4. 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`.


Generate IO Ring Design Files

IO Ring File Generation Configuration

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

Directory: C:/projects/achronix/Speedster7t_2D_noc_ref_design_RD22/ac7t1500/src/ioring

☑ Add to active project

Figure 9 • ACE Select IP Generated Files Directory

5. Click **Finish** in the IO Ring File Generation Configuration dialog box to generate the I/O ring files.

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, ACE generates new files named accordingly. These new files are added to the ACE project in both batch and GUI mode (be sure 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, the ACE project has the new speed grade constraint files for that run. 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.

The batch flow can be used with either ACE-driven integrated synthesis or standalone synthesis in Synplify Pro.

Note

The default flow uses ACE-driven integrated synthesis which requires ACE 10.0 or later.

ACE-Driven Integrated Synthesis (default)

In this flow, the following folders are created

- `/build/results/ace/impl_1/syn` – contains a generated Synplify Pro project and the generated netlist.
- `/build/results/ace/impl_1/pnr` – contains the placed and routed design.
  - Within this directory, the `/output` directory contains the bitstream and any other files needed for download and runtime.
- Custom Synplify Pro options and ACE options can be set in `/src/constraints/ace_options.tcl`.

Note

ACE 10.0 or later is required to run ACE-driven integrated synthesis flow.

Standalone Synthesis in Synplify Pro

The standalone synthesis makefile target is "make run_ss". For ACE versions prior to 10.0, or if using the standalone synthesis flow with ACE 10.0 and later, the following folders are created:

- `/build/results/syn` directory – Synplify Pro is executed in batch mode, synthesizing the design, which generates a netlist in `/build/results/syn/rev_1/<design_name>.vm`. If synthesis is unsuccessful, consult `/build/results/syn/rev_1/<design_name>.srr` for details of any synthesis failure.
  - Custom Synplify Pro options can be set in `/src/constraints/synplify_options.tcl`.
- `/build/results/ace` (pre 10.0) directory or `/build/results/ace/pnr` (10.0 onwards) directory – contains the placed and routed design.
  - Within this directory, the `/output` directory contains the bitstream and any other files needed for download and runtime.
  - ACE options can be set in `/src/constraints/ace_options.tcl`

Change Default Flow

If using ACE 10.0 or later and standalone synthesis flow is desired as the default, then a modification is needed to the `REV_DIR` variable of the makefile. Change the `REV_DIR` variable to a value other than "ace" to force the standalone synthesis flow.

```
# Set $REV_DIR = "ace" for integrated ACE synthesis by default, ACE 10.0 onwards
# Set $REV_DIR to another value, i.e. rev_1, to use separate standalone synthesis by
default
# To override the default setting of REV_DIR, use the run_ss receipe
REV_DIR  := ace
```

Alternatively, call the make flow with the `run_ss` recipe.

```
$> make run_ss
```

Makefile Options

The makefile supports multiple build flow options:

Command	Usage
$ make	Default flow. Regenerate all files in /src/ioring. Synthesize and build a single implementation with ACE.(†)
$ make run	Same as "make, except does not regenerate the ioring files.(†)
$ make syn_only	Synthesis only, using standalone Synplify Pro.
$ 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 run_ss	Same as "make run" except uses standalone Synplify Pro.
$ make ioring_only	Regenerate all files in /src/ioring.
$ make clean	Delete all generated and temporary files.

Note

† By default uses ACE-driven integrated synthesis (ACE 10.0 or later)

Constraint Files

In addition to any I/O ring constraint files, additional constraint files are used in any build flow. Please refer to the Installing the Design → Constraint Files section for a full description of all available constraint files.

Note

At this time, Achronix recommends separate timing constraint (`.sdc`) files for Synplify Pro and ACE. Specifically, Achronix does not recommend using the generated `.scf` file from Synplify Pro to constrain the timing within 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.tcl` file.

Note

Some designs use multiple `filelist.tcl` files with varying prefixes or suffixes to the "filelist" filename.

GUI Flow

The design has a pre-generated GUI project file for ACE. This file is located in the `/src/ace` directory. Open this file to interactively edit or run builds.

Note

When using the GUI projects, any generated files are placed in the GUI project file directory.

- If generating an entirely separate Synplify Pro project, the netlist generated from that project needs to be imported accordingly into the ACE project.
- For ACE, any implementation directory is generated as `/src/ace/impl_<name>` . While using integrated synthesis within the implementation directory there are `/pnr` and `/syn` directories containing the place and route, and integrated synthesis results respectively.
- 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.

Timing Closure

All of the Achronix customer designs are set to meet timing using the tool versions that they were released against. Please refer to any `README.txt` or `Release_Notes.txt` files in the root of the project for details of the tool versions. Timing is met in both batch and GUI flows on Linux systems.

Caution!

Timing closure for the design is not guaranteed when compiling on Windows systems.

For designs compatible with ACE 10.0 or later, timing is met when ACE integrated synthesis is used. Standalone synthesis may have differing synthesis options. Therefore, timing closure on builds using standalone synthesis will not be guaranteed for those designs.

If newer tools are used than those the design was released against, it is generally expected that timing is met. In the event timing is not met, it is suggested the to run ACE multiprocess in order to find a new set of implementation options that close timing. It is suggested to first reset all implementation options to default values before running ACE multiprocess.

- In GUI mode, when opening the project with a new version of ACE, select **Reset all options**
- In batch mode, remove any options from the `ace_options.tcl` file listed under the section marked as extracted from multiprocess.

Device Setup

All reference designs are pre-configured for each device in this way, <DESIGN>/<DEVICE>; 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 in 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 AC7t1400 device
    // ---------------------------------------------------------------------
    // If this design is intended to be targeted to the AC7t1400 device,
    // then it is necessary to instantiate the SRM, (Serial Rate Monitor).
    // This is required in all AC7t1400 designs, as shown below
    // ---------------------------------------------------------------------
    // The define ACX_DEVICE_AC7t1400 is set as follows :
```

```
    //      In simulation by $ACE_INSTALL_DIR/libraries/device_models/
AC7t1400_simmodels.v
    //      In synthesis by  $ACE_INSTALL_DIR/libraries/device_models/
AC7t1400_synplify.v
    //
    //      For this design the above files are selected as follows
    //      In simulation, in the appropriate /sim/<simulator>/Makefile
    //      In batch build flow, the selection is done in /scripts/
create_syn_project.tcl based on the selected device
    // ---------------------------------------------------------------------
    `ifdef ACX_DEVICE_AC7t1400

        (* must_keep *) ACX_SRM x_ACX_SRM () /* synthesis syn_noprune=1 */;
    `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

If standalone synthesis is being used, then the first step is to create a netlist targeted to the new device using the Synplify Pro GUI project. Within `.prj` file, make the following changes:

```
#project files
add_file -verilog "$ACE_INSTALL_DIR/libraries/device_models/AC7t1400_synplify.v"  # <-
Update library file to new device

#device options
set_option -part AC7t1400 # <- Update selected part
```

Note

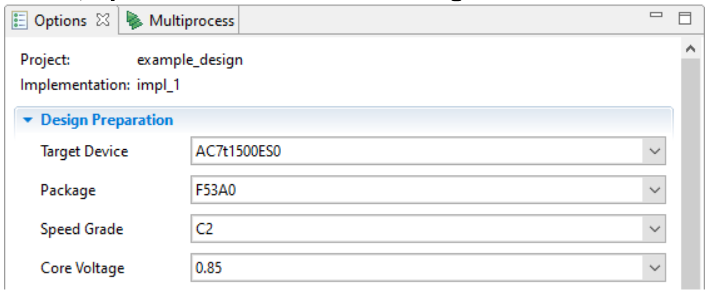
It is suggested to edit the Synplify Pro project file with a text editor, rather than from within the tool. The tool does not recognise the environment variable, `ACE_INSTALL_DIR` that is used to create relative paths to the ACE library files. If the project is edited within the tool, the relative paths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

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

ACE

The ACE GUI project must be updated with the following steps:

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

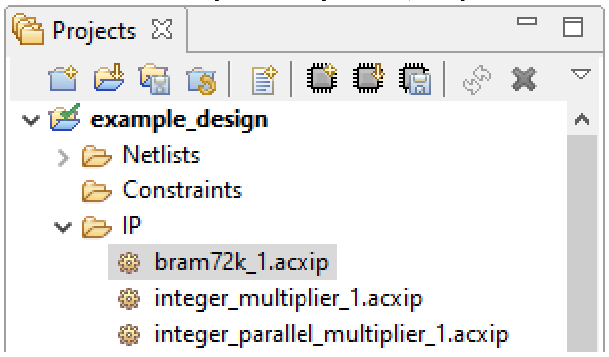


Project: example_design
Implementation: impl_1

Target Device	AC7t1500ES0
Package	F53A0
Speed Grade	C2
Core Voltage	0.85

Figure 10 • ACE Select Target Device

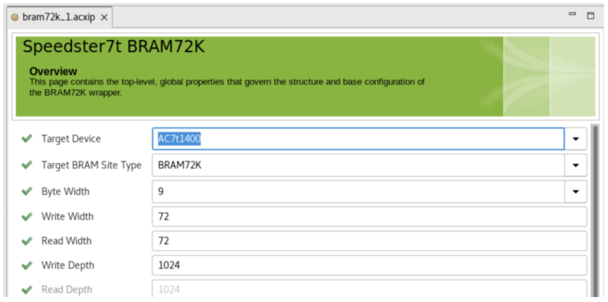
2. Open each of the IP definition files from the **Project Perspective, Projects** tab, **IP** selection.



```
example_design
  Netlists
  Constraints
  IP
    bram72k_1.acxip
    integer_multiplier_1.acxip
    integer_parallel_multiplier_1.acxip
```

Figure 11 • ACE Select IP

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



Speedster7t BRAM72K

Overview
This page contains the top-level, global properties that govern the structure and base configuration of the BRAM72K wrapper.

Target Device	AC7t1400
Target BRAM Site Type	BRAM72K
Byte Width	9
Write Width	72
Read Width	72
Write Depth	1024
Read Depth	1024

Figure 12 • 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, in I/O Ring Files.

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 updated, 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, AC7t1500, AC7t1400, AC7t800
DEVICE     := "AC7t1500"
```

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

```
$> make DEVICE=AC7t1500
```

Running the Design

This reference design can be targeted to silicon. Pre-compiled bitstreams are included. Runtime scripts are provided to enable running the design. If the design is rebuilt, the same methods and scripts can be used to confirm operation of the new design on the card(s) supported by the design.

References

For full details on how to program bitstreams, run the Tcl script flow and reference the available Tcl commands, refer to ***Runtime Programming of Speedster FPGAs*** (AN025).

Prerequisites

The card hosting the supported device must be powered up and connected via USB to the host computer. It is recommended that the host computer should have the same ACE version that was used to build the design. For the specific ACE version used, consult the release notes or the README files within the design.

Note

ACE versions 9.2 or greater supports the bitstream processes detailed in the section below.

Warning

If ACE is installed on Linux, it is often necessary to grant permission to ACE to access the USB ports. This process can vary according to the Linux distribution. For full details on installing ACE, refer to the *ACE Installation and Licensing Guide* (UG002).

Note

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

Files and Directories

The runtime bitstreams and scripts are distributed in the following directories:

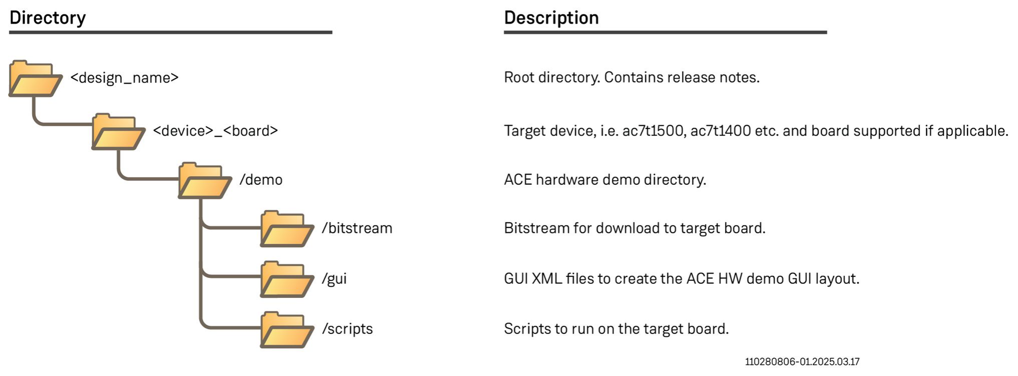

Directory
Description
<design_name>
Root directory. Contains release notes.
<device>_<board>
Target device, i.e. ac7t1500, ac7t1400 etc. and board supported if applicable.
/demo
ACE hardware demo directory.
/bitstream
Bitstream for download to target board.
/gui
GUI XML files to create the ACE HW demo GUI layout.
/scripts
Scripts to run on the target board.
110280806-01.2025.03.17

Figure 13 • Runtime File Directory Structure

Runtime

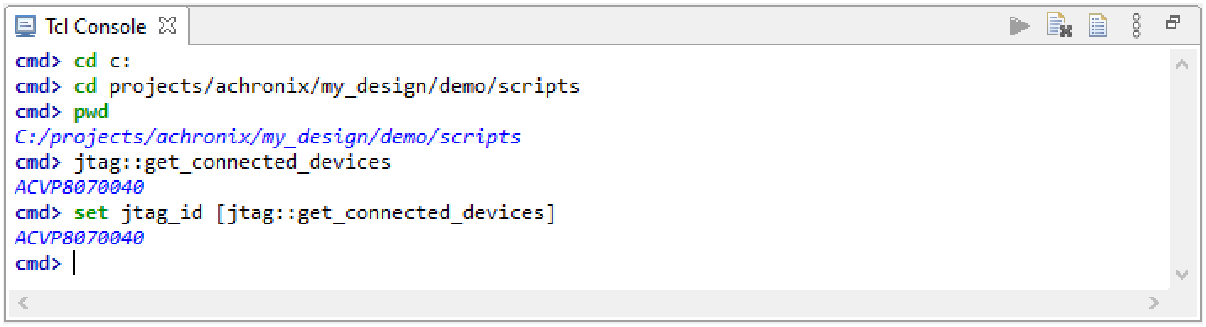
Program the hex bitstream and execute the runtime script to demonstrate the design on the FPGA.

Programming the Bitstream

Download the reference design, then follow these steps:

1. Open ACE.
2. Within the Tcl console window, navigate to the `<design_name>/<device>_<board>/demo/scripts` directory.
3. Obtain the JTAG ID of the connected card. The ID should be in the form `ACVPxxxxxx`.
4. Assign this value to the Tcl variable `jtag_id`.

   The sequence for these steps is illustrated in the following example.

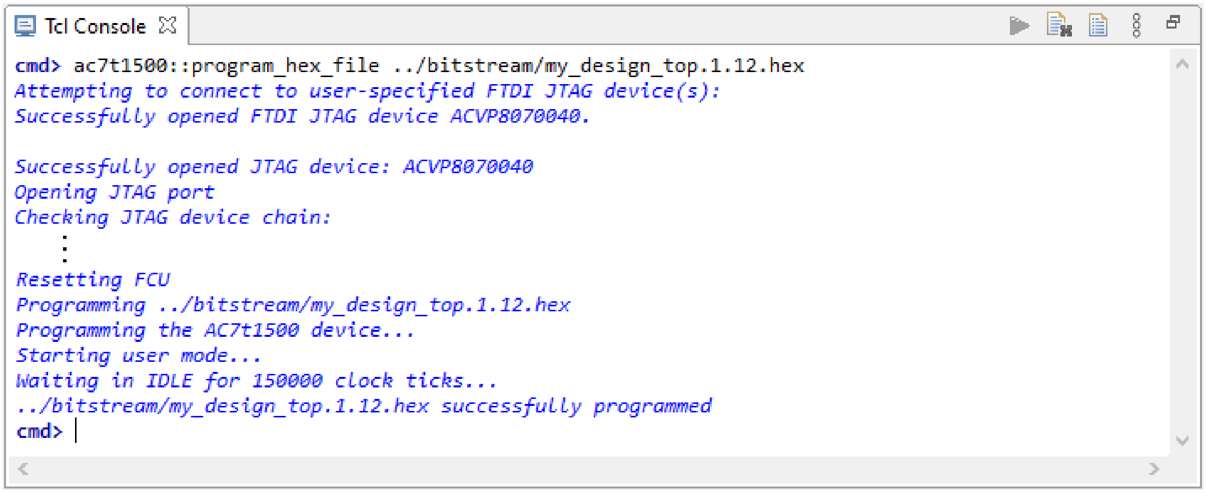


```
cmd> cd c:
cmd> cd projects/achronix/my_design/demo/scripts
cmd> pwd
C:/projects/achronix/my_design/demo/scripts
cmd> jtag::get_connected_devices
ACVP8070040
cmd> set jtag_id [jtag::get_connected_devices]
ACVP8070040
cmd>
```

Figure 14 • Setting the jtag_id Variable

5. Program the bitstream, ensuring the correct device version, `program_hex_file`, is used.

   It is recommended to use relative paths from the current directory. The `program_hex_file` opens the JTAG port if it has not yet been opened.



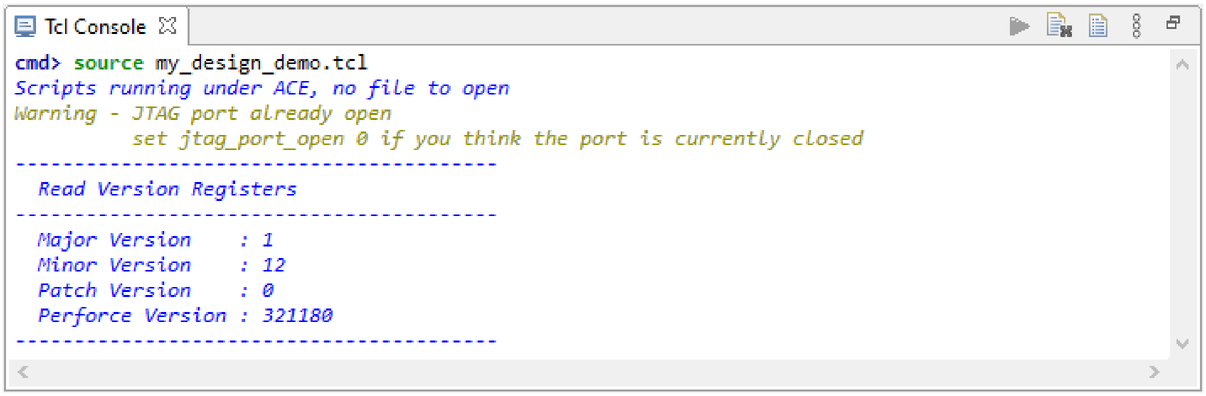
```
cmd> ac7t1500::program_hex_file ../bitstream/my_design_top.1.12.hex
Attempting to connect to user-specified FTDI JTAG device(s):
Successfully opened FTDI JTAG device ACVP8070040.

Successfully opened JTAG device: ACVP8070040
Opening JTAG port
Checking JTAG device chain:
    ⋮
Resetting FCU
Programming ../bitstream/my_design_top.1.12.hex
Programming the AC7t1500 device...
Starting user mode...
Waiting in IDLE for 150000 clock ticks...
../bitstream/my_design_top.1.12.hex successfully programmed
cmd>
```

Figure 15 • Programming the Bitstream

Some designs may contain scripts within the `demo/scripts` directory, performing all the steps listed above. For more information, consult the release notes/readme stored in each design.

Running the Design

1. Source the runtime script to execute it. This publishes status and results to the console.



```
cmd> source my_design_demo.tcl
Scripts running under ACE, no file to open
Warning - JTAG port already open
          set jtag_port_open 0 if you think the port is currently closed
-----------------------------------------
  Read Version Registers
-----------------------------------------
  Major Version    : 1
  Minor Version    : 12
  Patch Version    : 0
  Perforce Version : 321180
-----------------------------------------
```

Figure 16 • Executing the Runtime Script

Note

- The warning that the JTAG port is already open is expected. The JTAG port was opened in the previous programming step. The runtime scripts include a command to open the JTAG port in the event that it was closed since the device was programmed.
- To aid revision control of builds, the Achronix supplied tool flow creates bitstreams with the major and minor version numbers included into the bitstream name, (`my_design_top.1.12.hex`). The major and minor version numbers are defined within the `/src/include/version_defines.svh` file; in addition these values populate the version registers within the register control block. Thus, it is possible to correlate between the name of a bitstream and the internal version used to indicate which build is being used.

Monitoring the Board Status

If applicable, observe the four multicolor LEDs on the card and ensure they match the design documented behavior. The color displayed on a specific LED is determined by the setting of the pair of bits for each LED in the 8-bit `led_l` bus as illustrated in the following tables.

Table 39 • VectorPath S7t-VG6 Card Multicolor LEDs

LED(†)	Signal	FPGA Pin	Color	Behavior	FPGA Pin Name	Vertical Location
D3	`led_l[0]`	AP17	Orange	Active Low	GPIO_N0_BYTE2_BIT_0	Bottom (nearest the board).
	`led_l[4]`	AN16	Green	Active Low	GPIO_N0_BYTE1_BIT_10	
D4	`led_l[1]`	AR17	Orange	Active Low	GPIO_N0_BYTE2_BIT_1	Second position.

LED[†]	Signal	FPGA Pin	Color	Behavior	FPGA Pin Name	Vertical Location
	`led_l[5]`	AR16	Green	Active Low	GPIO_N0_BYTE1_BIT_11	
D5	`led_l[2]`	AT17	Orange	Active Low	GPIO_N0_BYTE2_BIT_2	Third position.
	`led_l[6]`	AR19	Green	Active Low	GPIO_N0_BYTE2_BIT_6	
D6	`led_l[3]`	AV18	Orange	Active Low	GPIO_N0_BYTE2_BIT_3	Top position.
	`led_l[7]`	AT20	Green	Active Low	GPIO_N0_BYTE2_BIT_7	

Table Notes

† An LED turns yellow when both `led_l` bits for that particular LED are set low.

Table 40 • VectorPath 815 Card Multicolor LEDs

LED[†]	Signal	FPGA Pin	Color	Behavior	FPGA Pin Name	Vertical Location
LED1	`led_l[0]`	AP17	Green	Active Low	GPIO_N0_BYTE2_BIT_0	Bottom (nearest the board).
	`led_l[4]`	AN16	Orange	Active Low	GPIO_N0_BYTE1_BIT_10	
LED2	`led_l[1]`	AR17	Green	Active Low	GPIO_N0_BYTE2_BIT_1	Second position.
	`led_l[5]`	AR16	Orange	Active Low	GPIO_N0_BYTE1_BIT_11	
LED3	`led_l[2]`	AT17	Green	Active Low	GPIO_N0_BYTE2_BIT_2	Third position.
	`led_l[6]`	AR19	Orange	Active Low	GPIO_N0_BYTE2_BIT_6	
LED4	`led_l[3]`	AV18	Green	Active Low	GPIO_N0_BYTE2_BIT_3	Top position.
	`led_l[7]`	AT20	Orange	Active Low	GPIO_N0_BYTE2_BIT_7	

LED[†]	Signal	FPGA Pin	Color	Behavior	FPGA Pin Name	Vertical Location

Table Notes

† An LED turns yellow when both `led_l` bits for that particular LED are set low.

Simulating the Design

Supported Simulators

Reference designs are provided with a simulation environment. Scripts for Mentor QuestaSim, Synopsys VCS and Aldec Riviera simulators are included.

Location

All designs have a `/sim` directory located in the design root directory. Within this directory there are `/vcs` , `/questa` and `/riviera` 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 for 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.

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/AC7t1500`. 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 in the following example. 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 and Riviera 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 follows:

```
# 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 following table.

Table 41 • Simulation RTL defines

Module [1][2]	Define
GDDR6 controller 0	ACX_GDDR6_0_FULL
GDDR6 controller 1	ACX_GDDR6_1_FULL
GDDR6 controller 2	ACX_GDDR6_2_FULL
GDDR6 controller 3	ACX_GDDR6_3_FULL
GDDR6 controller 4	ACX_GDDR6_4_FULL
GDDR6 controller 5	ACX_GDDR6_5_FULL
GDDR6 controller 6	ACX_GDDR6_6_FULL
GDDR6 controller 7	ACX_GDDR6_7_FULL
DDR4 controller	ACX_DDR4_FULL
Ethernet subsystems (both)	ACX_ETHERNET_FULL
PCIe Controller 0 (×8)	ACX_PCIE_0_FULL
PCIe controller 1 (×16)	ACX_PCIE_1_FULL
GPIO North block	ACX_GPIO_N_FULL
GPIO South block	ACX_GPIO_S_FULL
All SerDes lanes	ACX_SERDES_FULL
All PLLs and Clock Generators [3]	ACX_CLK_NW_FULL, ACX_CLK_NE_FULL, ACX_CLK_SW_FULL, ACX_CLK_SE_FULL

Module [1][2]	Define

Table Notes

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

Warning!

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

GDDR6 BFM Memory Size

As previously described, the GDDR6 is supported with both BFM and RTL models for simulation. To set the supported memory size in RTL, the appropriate GDDR6 model must be instantiated in the testbench. Alternatively, when using the GDDR6 BFM model it is possible to switch the model between supporting a 16Gb and an 8Gb sized device.

By default, the GDDR6 BFM is configured to support a 16Gb device (two 8Gb channels). To set the GDDR6 BFM to support only 8Gb per device (two 4Gb channels), set the following define for all GDDR6 controller modules:

```
ACX_GDDR6_BFM_8Gb
```

It is recommended that the define is set in the `/sim/<simulator>/system_files_bfm.f` file.

Note

The define `ACX_GDDR6_BFM_8Gb` applies to all GDDR6 BFMs within the DSM. It is not possible to set different GDDR6 BFMs to different memory sizes. This applies to simulation only. In hardware, GDDR6 controllers may be individually configured to match the size of the respective connected GDDR6 devices.

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 as follows:

```
# Include any utility files that may have common functions
source AC7t1500_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 previous example is for the Speedster7t AC7t1500 FPGA. 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*** (AN025). To generate the simulation command file, ACE is run in batch mode using the runtime programming script, as follows:

```
# 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 -device $(BASE_NAME)"
```

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. An example simulation command file can be found in the section "Configuration File Format".

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 `ace` command from the previous example, `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()`, as in the following example.

```
# 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 follows.

```
// 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 and Riviera

```
$ 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 and Riviera Non-Makefile Flow

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

Note

The following example is shown for QuestaSim. The same steps can be used for Riviera, when run in the `/riviera` directory

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

```
$ vsim
```

3. Within the QuestaSim GUI, launch the script:

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

Note

- The QuestaSim and Riviera scripts make use of 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. Additionally, the environment variable, `ACE_INSTALL_DIR`, must use "/" as path separators rather than "\". For example:

  `ACE_INSTALL_DIR = C:/achronix/10.0/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 each device; 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.sv`. 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 following code example 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_AC7t1500
    `include "ac7t1500_utils.svh"
`elsif ACX_DEVICE_AC7t800
    `include "ac7t800_utils.svh"
`endif
```

In the previous 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 following code example.

```
// 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
# Devices currently supported; AC7t1500, AC7t1400 and AC7t800ES0
DEVICE     := AC7t1500
```

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

```
$> make DEVICE=AC7t800
```

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

```
# ----------------------------------------------------------------------
# Define the target device
# ----------------------------------------------------------------------
# Devices currently supported; AC7t1500, AC7t1400 and AC7t800ES0
DEVICE     := "AC7t1500"
```

Note

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

Device Simulation Model

Many designs require a simulation overlay named the device simulation model (DSM). This package combines the full RTL of the 2D network on chip (NoC) with bus functional models (BFMs) of the interface subsystems that surround the NoC and FPGA fabric. This combination of true RTL for the 2D NoC and models for the interface subsystems allows developing designs within a fast responsive simulation environment, while achieving cycle-accurate interfaces from the NoC, and representative cycle responses from the hard interface subsystems. This simulation environment allows a designer to iterate rapidly to develop and debug their design.

Description

The DSM provides full RTL code for the NoC, combined with BFMs of the surrounding interface subsystems. The structure is wrapped within a SystemVerilog module named per device (i.e., `ac7t1500`). Instantiate one instance of this module within the top-level testbench.

In addition, the DSM provides binding macros such that binding between elements of a design and the same elements within the device is possible. For example, the design might instantiate a 2D NoC access point (NAP). It is then necessary to bind this NAP instance to the NAP in the correct location within the 2D NoC by using the `` `ACX_BIND_NAP_RESPONDER ``, `` `ACX_BIND_NAP_INITIATOR ``, `` `ACX_BIND_NAP_HORIZONTAL ``, `` `ACX_BIND_NAP_VERTICAL `` or `` `ACX_BIND_NAP_ETHERNET `` macro, whichever is appropriate for the design.

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

Selecting the Required DSM

DSM Utility Package

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

Table 42 • DSM Utility Packages

Devices	DSM Utility Package	ACX_DEVICE_NAME
AC7t1500, AC7t1500ES0	`ac7t1500_utils.svh`	ac7t1500
AC7t1400, AC7t1400ES0	`ac7t1400_utils.svh`	ac7t1400
AC7t800, AC7t800ES0	`ac7t800_utils.svh`	ac7t800

Device-Specific Simulation Files

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

Table 43 • Simulation Model Files and Defines

Device	Simulation Model File	ACX_DEVICE Macro
AC7t1500	`AC7t1500_simmodels.sv`	ACX_DEVICE_AC7t1500
AC7t1500ES0	`AC7t1500ES0_simmodels.sv`	ACX_DEVICE_AC7t1500ES0

Device	Simulation Model File	ACX_DEVICE Macro
AC7t1400	`AC7t1400_simmodels.sv`	ACX_DEVICE_AC7t1400
AC7t1400ES0	`AC7t1400ES0_simmodels.sv`	ACX_DEVICE_AC7t1400ES0
AC7t800	`AC7t800_simmodels.sv`	ACX_DEVICE_AC7t800
AC7t800ES0	`AC7t800ES0_simmodels.sv`	ACX_DEVICE_AC7t800ES0

Instantiate DSM Utility Package

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

```
    // Include the appropriate DSM utility file which defines the appropriate macros
    // If an unsupported device is selected, then compilation will fail
`ifdef ACX_DEVICE_AC7t1500ES0
    `include "ac7t1500_utils.svh"
`elsif ACX_DEVICE_AC7t1500
    `include "ac7t1500_utils.svh"
`elsif ACX_DEVICE_AC7t800ES0
    `include "ac7t800_utils.svh"
`endif

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

Version Control

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

To ensure that the correct version of the DSM is used, a task must be included within the design testbench to confirm the version compatibility. This function should be instantiated as follows:

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

require_version( ) Task

The require_version task has four arguments. In order:

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

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

Note

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

Example Design

An example structure of a user testbench, instantiating both the DSM and the user design under test is shown in the following diagram. This example shows the macros required for the responder NAPs, and the DCIs for two instances of the GDDR6 subsystem. For other forms of NAPs, or for other DCI types, such as DDR, consult the Bind Macros and DSM Direct-Connect Interfaces tables.

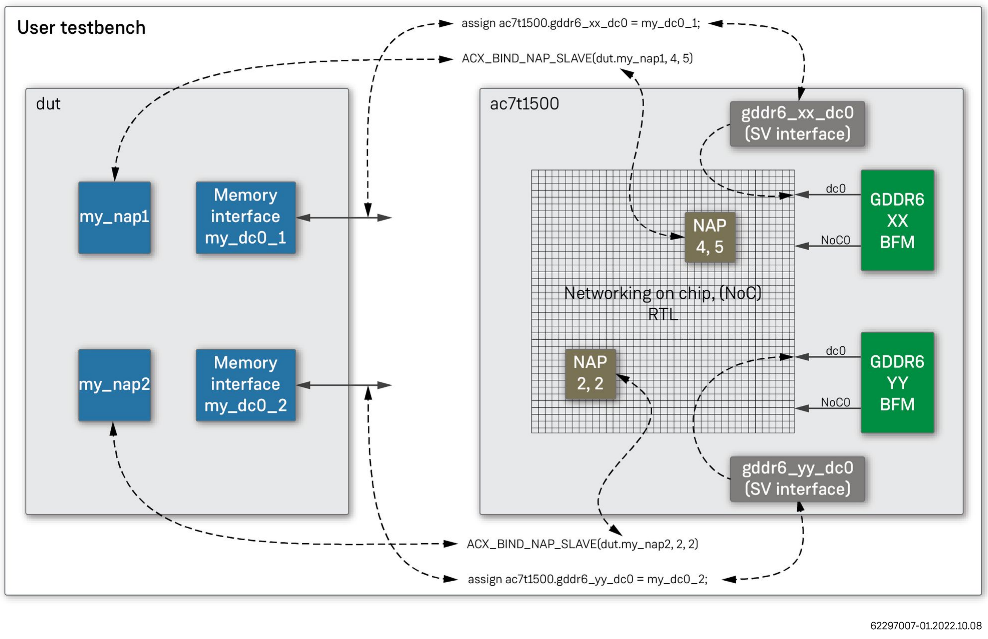

User testbench
dut
my_nap1
Memory interface my_dc0_1
my_nap2
Memory interface my_dc0_2
ac7t1500
gddr6_xx_dc0 (SV interface)
gddr6_yy_dc0 (SV interface)
NAP 4, 5
NAP 2, 2
Networking on chip, (NoC) RTL
GDDR6 XX BFM
GDDR6 YY BFM
dc0
NoC0
assign ac7t1500.gddr6_xx_dc0 = my_dc0_1;
ACX_BIND_NAP_SLAVE(dut.my_nap1,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

Figure 17 • Example Simulation Structure

In the previous example, there are two NAPs, `my_nap1` and `my_nap2`. In addition, there are two direct-connect interfaces, `my_dc0_1` and `my_dc0_2`. In the top-level, testbench bindings are made between the NAPs in the design and the NAPs within the device using the ACX_BIND_NAP_RESPONDER macro:

- This macro supports inserting the coordinates of the NAP within the 2D NoC in order that the simulation is aligned with physical placement of the NAP on silicon.
- The DCIs are ports on the user design. These ports are assigned to the appropriate signals within the device direct-connect SystemVerilog interface.

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

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

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

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

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

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

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

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

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

Note

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

In simulation

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

In place and route

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

set_verbosity( ) Task

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

The verbosity levels are defined in the following table.

Table 44 • Verbosity Levels

Verbosity Level	Description
0	Print no messages.
1	Print messages from initiator and responder interfaces only.
2	Print messages from level 1 and from each NoC data transfer.
3	Print messages from level 2, port bindings and NoC performance statistics.

Chip Status Output

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

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

Bind Macros

The following bind statements are available.

Table 45 • Bind Macros

Macro	Arguments [1]	Description
`ACX_BIND_NAP_HORIZONTAL`	`user_nap_instance`, `noc_colunm`, `noc_row`	To bind a horizontal streaming NAP, instance `ACX_NAP_HORIZONTAL`.
`ACX_BIND_NAP_VERTICAL`	`user_nap_instance`, `noc_colunm`, `noc_row`	To bind a vertical streaming NAP, instance `ACX_NAP_VERTICAL`.
`ACX_BIND_NAP_AXI_INITIATOR` [2]	`user_nap_instance`, `noc_colunm`, `noc_row`	To bind an AXI initiator NAP, instance `ACX_NAP_AXI_INITIATOR`.
`ACX_BIND_NAP_AXI_RESPONDER` [2]	`user_nap_instance`, `noc_colunm`, `noc_row`	To bind an AXI responder NAP, instance `ACX_NAP_AXI_RESPONDER`.
`ACX_BIND_NAP_ETHERNET`	`user_nap_instance`, `noc_colunm`, `noc_row`	To bind an Ethernet NAP instance, `ACX_NAP_ETHERNET`.

Table Notes

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

Direct-Connect Interfaces

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

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

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

Connecting from the user design to the DCI ports involves one of two methods:

- Connecting directly using the interfaces built into the DSM
- Using an ACE-generated port binding file

Note

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

Suggested Flows

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

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

The two methods are detailed as follows.

DSM DC Interfaces

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

Table 46 • 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

Subsystem	Interface Name	Physical Location [1]	GDDR6 Channel	SystemVerilog Interface Type	Data Width	Address Width
GDDR6	`gddr6_6_dc0`	East 2	0	`t_ACX_AXI4`	512	33
GDDR6	`gddr6_6_dc1`	East 2	1	`t_ACX_AXI4`	512	33
DDR4	`ddr4_dc0`	South	–	`t_ACX_AXI4`	512	40
Ethernet	`ethernet_0_dc`	North West	–	`t_ACX_ETHERNET_DCI`	–	–
Ethernet	`ethernet_1_dc`	North East	–	`t_ACX_ETHERNET_DCI`	–	–
Serdes	`serdes_eth0_q0_dc`	North West	–	`t_ACX_SERDES_DCI`	128	–
Serdes	`serdes_eth0_q1_dc`	North West	–	`t_ACX_SERDES_DCI`	128	–
Serdes	`serdes_eth1_q0_dc`	North East[2]	–	`t_ACX_SERDES_DCI`	128	–
Serdes	`serdes_eth1_q1_dc`	North East[2]	–	`t_ACX_SERDES_DCI`	128	–

Table Notes

1. Physical orientation west-to-east is with regards to viewing the die in the floorplan view within ACE. The die is actually rotated about its vertical axis when packaged. Therefore, an interface shown on the floorplan, and listed in this table, as being on the west is physically on the east side of the device when located on the PCB. The north-to-south orientation is not affected and matches with this table, the ACE view, and the device on board.
2. Present on the Speedster7t AC7t800 DSM.

Note

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

Direct Connect to DSM Interfaces

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

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

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

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

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

Port Binding File to DSM Interfaces

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

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

Notes

- In the Achronix reference design flow the generated subsystem IP files are saved to the `/src/ioring` directory rather than the default `/src/ace/ioring_design` directory.
- The define `ACX_DSM_INTERFACES_TO_MONITOR_MODE` must be included in the simulation command line, so that it is present when the DSM is compiled. It cannot be included in the user testbench as this is compiled *after* the DSM.
- In the provided Achronix reference design flow, `ACX_DSM_INTERFACES_TO_MONITOR_MODE` is defined in the `/sim/<simulator>/system_files_bfm.f` and `/sim/<simulator>/system_files_rtl.f` files.

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

system_files_bfm.f

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

Testbench

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

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

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

Dual-Mode Connections to DSM Interfaces

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

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

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

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

Clock Frequencies

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

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

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

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

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

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

Note

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

The following clock frequency interfaces are available.

Table 47 • Clock Frequency Interfaces

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

Subsystem	Interface Name	Physical Location [1]	GDDR6 Channel
	`gddr6_6_noc1_clk`	East 2 NoC	1
	`gddr6_7_noc0_clk`	East 3 NoC	0
	`gddr6_7_noc1_clk`	East 3 NoC	1
	`gddr6_1_dc0_clk`	West 1 DCI	0
	`gddr6_1_dc1_clk`	West 1 DCI	1
	`gddr6_2_dc0_clk`	West 2 DCI	0
	`gddr6_2_dc1_clk`	West 2 DCI	1
	`gddr6_5_dc0_clk`	East 1 DCI	0
	`gddr6_5_dc1_clk`	East 1 DCI	1
	`gddr6_6_dc0_clk`	East 2 DCI	0
	`gddr6_6_dc1_clk`	East 2 DCI	1
DDR4	`ddr4_noc0_clk`	South NoC	–
	`ddr4_dci0_clk`	South DCI	–
DDR5	`ddr5_noc0_clk` [4]	South NoC	–
PCIe	`pciex16_clk` [3]	Gen5 PCIe ×16	–
	`pciex16_dc_clk`	Gen5 PCIe ×16 DCI	–
	`pciex8_clk`	Gen5 PCIe ×8	–
Ethernet	`ethernet_ref_clk` [3]	Ethernet reference clock [2]	–
	`ethernet_ff0_clk` [3]	Ethernet FIFO 0 clock [2]	–
	`ethernet_ff1_clk` [3]	Ethernet FIFO 1 clock [2]	–
Configuration	`cfg_clk`	System wide configuration clock	–

Subsystem	Interface Name	Physical Location [(1)]	GDDR6 Channel

Table Notes

1. Physical orientation west-to-east is with regards to viewing the die in floorplan view within ACE. The die is actually rotated about its vertical axis when packaged. Therefore, an interface shown on the floorplan, and listed in this table, as being on the west is physically on the east side of the device when located on the PCB. The north-to-south orientation is not affected and matches with this table, the ACE view, and the device on board.
2. The Ethernet clocks are common to both Ethernet subsystems. In simulation they must be set to operate from the same clock frequencies.
3. Present in the AC7t800 DSM.
4. Only present in the Speedster7t AC7t800 DSM.

Configuration

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

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

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

Startup Sequence

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

The simulation testbench can issue configuration processes as shown in the previous code snippet, and when the Chip Status Output is asserted, the testbench knows the device is correctly configured. The testbench can then proceed to apply the necessary tests.

fcu.configure() Task

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

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

The following interface subsystem names are supported:

Table 48 • Configuration Subsystem Names

Subsystem [4]	Interface Subsystem Name [1]	Physical Location [3]
GDDR6	`gddr6_0`	West 0
	`gddr6_1`	West 1
	`gddr6_2`	West 2
	`gddr6_3`	West 3
	`gddr6_4`	East 0
	`gddr6_5`	East 1
	`gddr6_6`	East 2
	`gddr6_7`	East 3
DDR4	`ddr4`	South
DDR5	`ddr5`	South
Ethernet	`ethernet0`	North
	`ethernet1`	North
GPIO North	`gpio_n`	North
GPIO South	`gpio_s`	South
PCIe ×8	`pcie_0`	North
PCIe ×16	`pcie_1`	North
All subsystems	`full` [2]	–

Subsystem [4]	Interface Subsystem Name [1]	Physical Location [3]

Table Notes

1. The interface subsystem name is case insensitive.
2. When using the `full` subsystem name, the full 42-bit address is required in the configuration file. When selecting an individual subsystem, only the 28-bit address is required. Refer to Configuration File Format for details.
3. Physical orientation west-to-east is with regards to viewing the die in floorplan view within ACE. The die is actually rotated about its vertical axis when packaged. Therefore, an interface shown on the floorplan, and listed in this table, as being on the west is physically on the east side of the device when located on the PCB. The north-to-south orientation is not affected.
4. Not all subsystems are available in all devices. Please refer to your specific device datasheet for details of available subsystems.

Configuration File Format

The configuration file has the following format:

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

# A comment line
// Another comment line

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

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

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

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

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

# Examples

# Writes
```

```
w 00005c0 76543210
w 0000014 00004064

# Reads
r 00005c0 00000000
r 0000014 00000000

# Verify
v 00005c0 76543210
v 0000014 00004064


# Wait for 50 cycles
d 0000000 00000032
```

Address Width

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

- When addressing an individual subsystem, only the lower 28 bits of the address field are used. The higher 14 bits are derived from the subsystem name.
- When addressing the full configuration memory space (interface subsystem name is set to `full`), 42 bits of the address space are required. In this mode, the FCU confirms that bits [41:34] of the address field are set to `8'h20`, which aligns with the 2D NoC global memory map plus control and status register (CSR) memory area. In this mode, the one configuration file can address multiple interface subsystems. See the ***Speedster7t Network on Chip User Guide* (UG089)** 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 following interface is only available in the simulation environment. For code that must be synthesized, define custom SystemVerilog interfaces, or use one of the interfaces predefined within the reference designs.

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

   logic                       aclk;      // Clock reference
   logic                       awvalid;   // AXI Interface
   logic                       awready;
   logic [ADDR_WIDTH -1:0]     awaddr;
   logic [LEN_WIDTH -1:0]      awlen;
   logic [8 -1:0]              awid;
   logic [4 -1:0]              awqos;
   logic [2 -1:0]              awburst;
   logic                       awlock;
   logic [3 -1:0]              awsize;
   logic [3 -1:0]              awregion;
   logic [3:0]                 awcache;
   logic [2:0]                 awprot;
   logic                       wvalid;
   logic                       wready;
   logic [DATA_WIDTH -1:0]     wdata;
   logic [(DATA_WIDTH/8) -1:0] wstrb;
   logic                       wlast;
   logic                       arready;
   logic [DATA_WIDTH -1:0]     rdata;
   logic                       rlast;
   logic [2 -1:0]              rresp;
   logic                       rvalid;
   logic [8 -1:0]              rid;
   logic [ADDR_WIDTH -1:0]     araddr;
   logic [LEN_WIDTH -1:0]      arlen;
   logic [8 -1:0]              arid;
   logic [4 -1:0]              arqos;
   logic [2 -1:0]              arburst;
   logic                       arlock;
   logic [3 -1:0]              arsize;
   logic                       arvalid;
   logic [3 -1:0]              arregion;
   logic [3:0]                 arcache;
   logic [2:0]                 arprot;
   logic                       aresetn;
   logic                       rready;
   logic                       bvalid;
   logic                       bready;
   logic [2 -1:0]              bresp;
   logic [8 -1:0]              bid;

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

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


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

Environment Variables

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

ACE_INSTALL_DIR

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

ACX_DEVICE_INSTALL_DIR

The optional environment variable `ACX_DEVICE_INSTALL_DIR` is used to select the DSM files. It should be set to the path, including the base directory, of the device files within the DSM package.

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

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

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

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

Note

For simulation, it is only necessary to set the `ACX_DEVICE_INSTALL_DIR` variable if the DSM is not installed in ACE integration mode. In all the supplied designs, the simulation makefiles define `ACX_DEVICE_INSTALL_DIR` as shown for ACE integration mode. This definition takes precedence over any local environment variable. If using a supplied simulation makefile, override the definition of `ACX_DEVICE_INSTALL_DIR` in the make flow invocation as follows (with the Speedster7t AC7t1500 FPGA as an example):

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

Revision History

Version	Date	Description
1.0	24 Feb 2020	• Initial Release.
1.1	03 Mar 2020	• Added Design Considerations section. • Changes to the block diagram to point to the correct channel. • Other content updates.
1.2	06 Mar 2020	• Updated notes on autogenerated SDC based on ACE 8.1.1 fix.
1.3	21 May 2020	• Added description on how clock periods are defined in testbench. • Corrected launch of QuestaSim GUI command. • Updated example instantiation of Speedster7t AC7t1500.

Version	Date	Description
2.0	22 Jul 2020	• Added description of RTL mode simulation, reformatted BFM simulation description. • Updated signal/port names and clock description to match design changes due to ACE 8.2 I/O Designer change. • Removed warning for autogenerated constraint files because issues have been fixed. • Removed table for testbench parameters because they are now being controlled by defines in `system_files_*.f`.
2.1	23 Jul 2020	• Corrected the polarity description of *_oen signals listed in **Table: GDDR6 Reference Design Top-Level RTL Ports**.
2.2	22 Sep 2020	• Added description of GDDR6 Register Checker logic block. • Updated diagrams to reflect accurate placement and updated IP block names and signal names. • Added description to detail RTL simulation mode related files and scripts.
2.3	28 Sep 2020	• Added details on why two separate PLLs are needed for GDDR6 subsystems. • Updated minimum ACE version needed to 8.2.1.
3.0	01 Jun 2021	• Raise ACE and DSM requirements to 8.3.3 • Additional DSM details added.
4.0	21 Nov 2022	• Added reg control block along with script and used registers to start, stop and monitor tests. • Ported design to VectorPath card with necessary `.acxip` files. • Changed all GDDR6 to 12 Gbps for this release • Removed GDDR6 training block and replaced with ACX_DEVICE_MANAGER • Implemented AXI memory channel that instantiates axi performance monitor, axi packet generator/checker, and FIFO. • Updated to ACE 8.8.2
4.1	25 Aug 2023	• Changed design so that GDDR6 now runs at 14 Gbps. • Added runtime instructions. • Changed order of building and simulation flows for clarity.

Version	Date	Description
5.0	02 May 2024	• Add BW_BMC_IF. • Added support for Speedster7t AC7t800 and AC7t1450 FPGAs. • Implemented with the unified ACE flow. • Added long term stability test.
5.1	03 Sep 2024	• Replace all AC7t1450 references with AC7t1400.
6.0	29 May 2025	• Release with VectorPath 815 accelerator card support only • Exclude files for VectorPath S7t-VG6 accelerator card, Speedster7t AC7t800 and Speedster7t AC7t1400

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

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

Notice of Disclaimer