

# Speedster7t Ethernet Reference Designs (RD019)



August 31, 2020

Reference Design

## Introduction

The Ethernet reference designs suite demonstrates the usage and performance of the Ethernet interface subsystem built into Speedster®7t FPGAs. The number of Ethernet subsystems varies according to the Speedster7t device. These reference designs target the AC7t100 device, which supports two Ethernet subsystems.

Each Ethernet subsystem is capable of up to two channels of 400 Gbps traffic, delivering a combined bandwidth of 800 Gbps per subsystem. Each subsystem contains three MAC cores, a 400G MAC, and two quad MACs, each of which can process four channels of 100G each.

The Ethernet reference designs suite consists of two designs, showing how the user can interface and send traffic through the Ethernet subsystems. These designs highlight a number of the unique features of the Speedster7t family; in particular how the high-speed Ethernet channels can be easily managed and interfaced within the FPGA fabric by use of the network on chip (NoC) and associated network access points (NAPs). The Speedster7t NoC enables easy and rapid interfacing to the often complex design task of streaming high-bandwidth Ethernet channels into and out of the FPGA.

These designs are:

- [100G \(see page 1\)](#) – Configured as 4 × 25G. Interfaced with a single NAP.
- [400G Packet Mode \(see page 9\)](#) – Configured as 8 × 50G. Interfaced using four NAPs.

## References

For full details of the Ethernet subsystem, including configuration and usage modes, refer to the *Speedster7t Ethernet User Guide* (UG097).

For further details of the NoC, refer to *Speedster7t Network on Chip User Guide* (UG089).

For further details on the NAPs, including instantiation templates, refer to *Speedster7t IP Component Library User Guide* (UG090).

## 100G (4x25G) Design

### Operation

The 100G Ethernet reference design has two major functional data flows as described below. This design targets a single Ethernet subsystem and utilizes two channels, each capable of 100 Gbps operation, (configured as 4x25G), with one channel for each of the functional flows.

Within the reference design bundle, the design is located in the `Speedster7t_Ethernet_ref_design_RD19/4x25g` directory. All file paths listed below are relative to this directory.

## Data Injection

The first flow demonstrates data being input to the Speedster7t device from an external Ethernet file reader. This reader converts data from the industry-standard pcap file format (see [PCAP Files \(see page 6\)](#)) into a packetized data stream. The data is input to the device from the user testbench and is received within the device. The reference design includes an IP packet processor, which reverses the source and destination addresses of any IPv4 packet. The data is then output from the device to an Ethernet file writer. This writer converts the incoming packetized data stream into a pcap formatted file. The written file is then compared against a golden reference file to demonstrate that the data has been successfully transferred and processed.

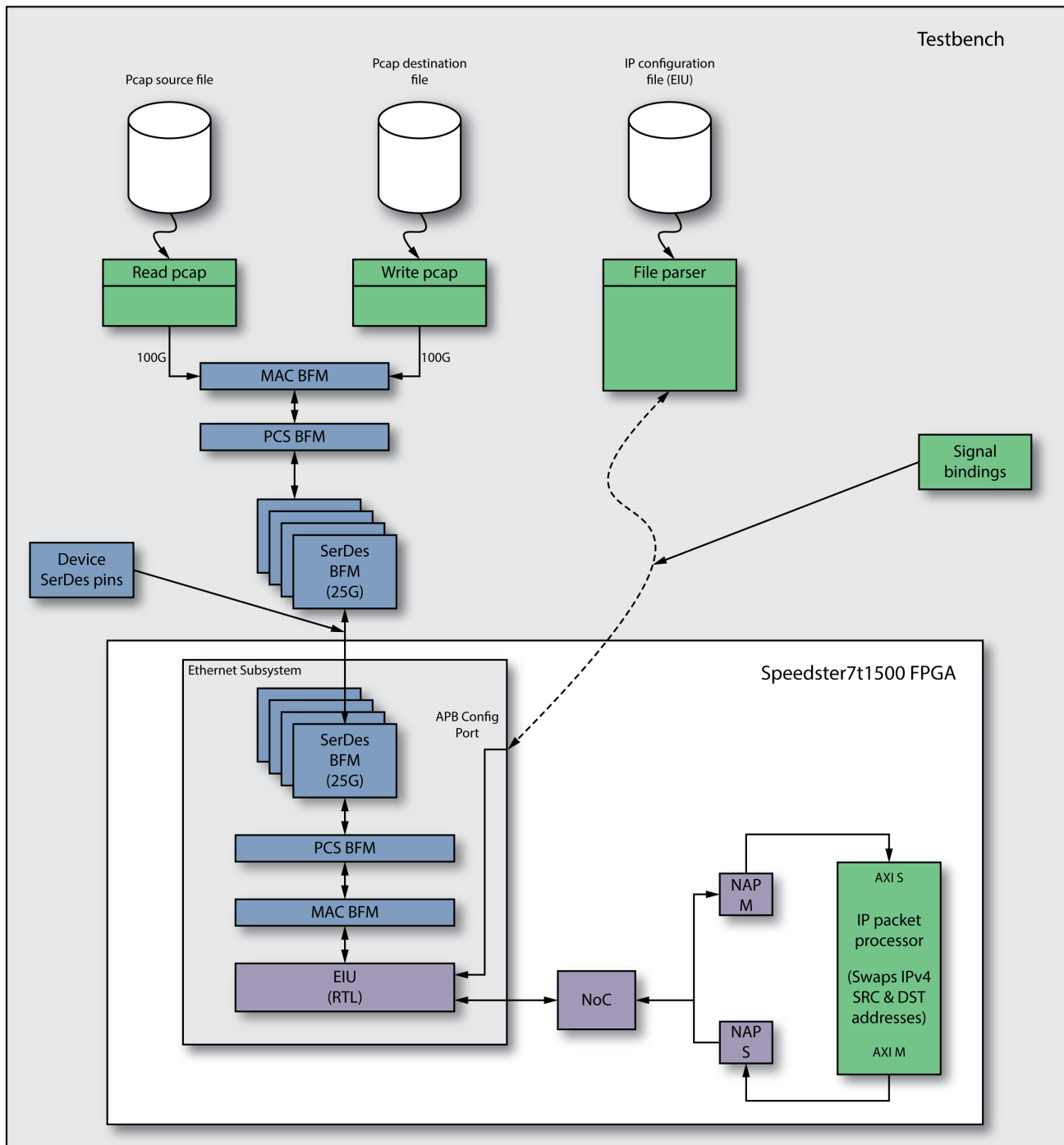


### Note

To enable repeatable file comparison, packet timestamps in the written and reference pcap files have been set to 0. Timestamps can be controlled in the `write_pcap` module by use of the `NO_TIMESTAMP` parameter. When this parameter is enabled, timestamps will be set based on simulator time. As differing simulators report differing values, it is suggested that if the user does require timestamps, that they be generated by a counter within the testbench to enable repeatable values to be obtained regardless of simulator.

The [block diagram \(see page 3\)](#) below shows this configuration of the reference design. The major elements are

- File I/O (white) – Stimulus, checker and configuration files.
- Reference design elements (green) – These comprise of both testbench utilities and the target design modules.
- Bus functional models (blue) – These BFM's mimic the behavior of some of the device functions, as well as providing those same functions within the testbench.
- Functional logic (purple) – Device logic comprising of the full cycle-accurate RTL of the device.



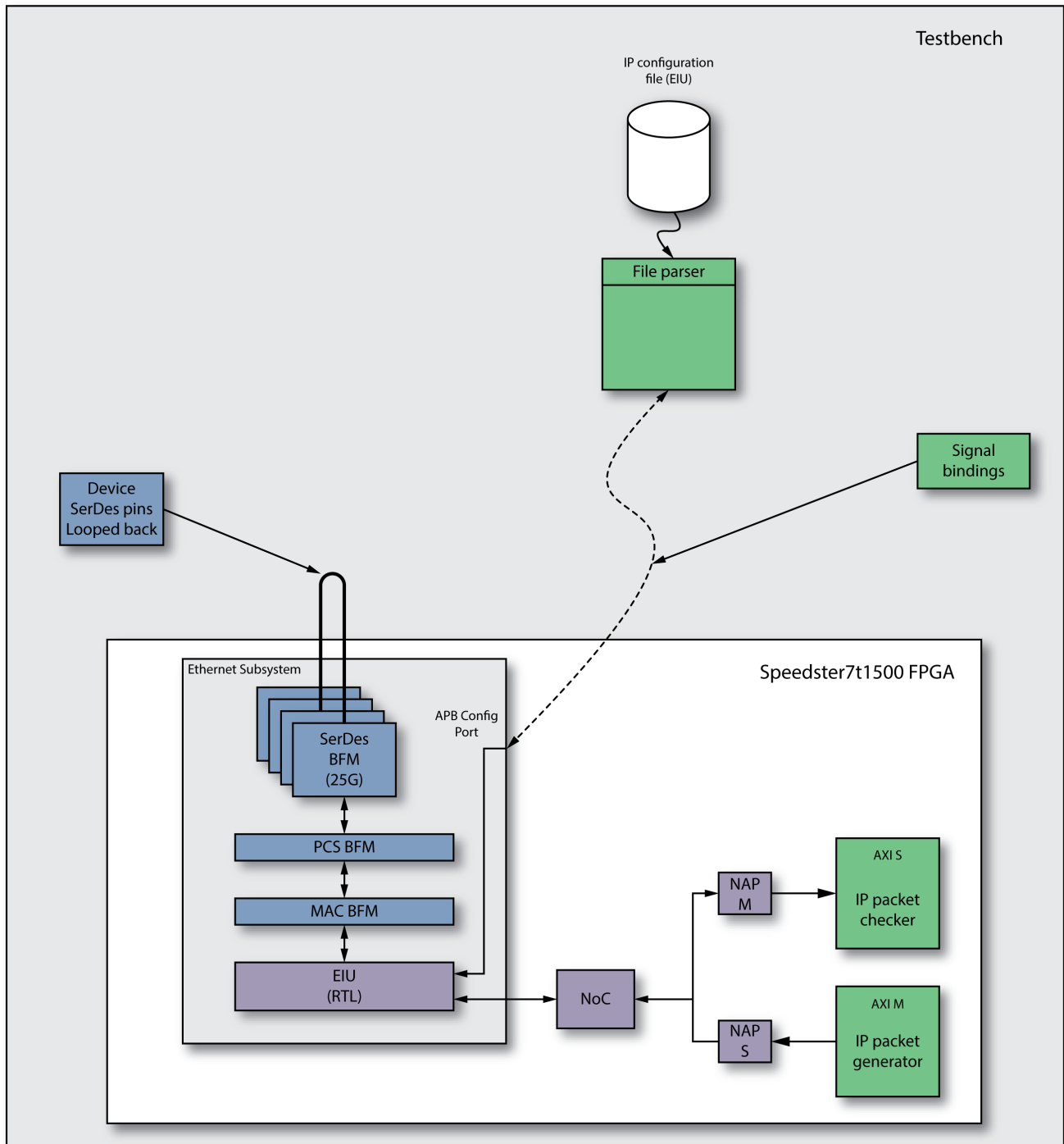
64722860-01.2020.05.22

**Figure 1: Data Injection Block Diagram**

## Data Loopback

The second flow demonstrates data being generated within the Speedster7t device, and output to the SerDes models. The SerDes pins are connected in loopback, such that the data is returned directly to the device fabric. The packet checker within the fabric validates that the transmitted data matches the received data.

The [block diagram](#) (see page 4) block diagram below shows this configuration of the reference design.



64722860-02.2020.05.22

**Figure 2: Data Loopback Block Diagram**

## Channel Configuration

The Ethernet reference design is configured as a single Ethernet subsystem, with each subsystem configured as two channels, both operating at 100 Gbps. Each 100 Gbps channel is configured as  $4 \times 25$  Gbps; therefore, each SerDes model is configured to operate at 25 Gbps, with a data width of 64 bits. The four SerDes lanes are combined to create a 256-bit data stream, which is passed from the input through to the NAP interface inside the device.

Each Ethernet subsystem has 8 SerDes lanes, so the two functional flows demonstrate simultaneous usage of all SerDes lanes.

## Ethernet Subsystem

Although this reference design only uses a single Ethernet subsystem (subsystem 0 by default), the target device, ac7t1500, contains two Ethernet subsystems. The testbench and configuration files support both subsystems, with selection of the desired subsystem controlled by a testbench define. See [Defines \(see page 5\)](#) for details as to how to switch between either of the available subsystems.

## Testbench Environment


### Parameters

The simulation testbench can be configured with a number of parameters.

**Table 1: Testbench Parameters**

Name	Description
PCAP_INPUT_FILE	Data injection test stimulus filename.
PCAP_OUTPUT_FILE	Data injection test result filename.
PCAP_REFERENCE_FILE	Data injection test reference filename.
NUM_LOOPBACK_PKTS	Data loopback test number of packets. 0 = run continuously.
CFG_FILENAME	EIU configuration filename.

### Defines

Name	Description
ACX_USE_ETHERNET1	<p>Use the simulation model Ethernet 1 subsystem. When enabled this affects the location of the NAPs (to correspond to the appropriate Ethernet subsystem), the destination of the configuration, and the selection of SerDes pins for data injection). By default this is undefined, selecting the Ethernet 0 subsystem:</p> <div>  This define only affects the testbench selection of Ethernet subsystem. The reference design itself is unchanged, regardless of the selected Ethernet subsystem. </div>

Name	Description
ACX_DUMP_SIM_SIGNALS	Enable writing of simulation waveforms to a file. If disabled, results in faster simulation; however no waveforms will be captured.

## PCAP Files

Pcap files are used by the testbench as they are an industry-standard format and can be read and generated by multiple readily available tools. These tools enable the packet contents to be examined and compared, including checking for correct packet formatting and checksum calculation. Many tools include a capture function which enables the user to capture packets from their Ethernet interface. These sequences can then be used as stimulus to the device. In addition tools are available which enable the user to generate their own pcap files to test particular stimulus sequences.

## Models

The Ethernet reference design uses a number of models to mimic functionality within the Speedster7t device and significantly accelerate simulation speeds. In addition these models are required by the testbench in order to provide encoding of any transmitted data and decoding of any received data.

- SerDes – All the SerDes interfaces are replaced by a SerDes bus functional model (BFM). This model serializes the data to mimic the operation of the true SerDes. The model includes link initialization using magic packets.



### Note

This SerDes model does not perform any of the clock recovery or sophisticated link initialization found in the actual Speedster7t SerDes.

- PCS – A 64/66b BFM of a PCS is used. The scrambling and descrambling of the data stream match the 64/66b specifications. The PCS model interfaces between the data transmitted by the SerDes model and converts this data to a packet format, with sop, eop, mod, valid and data signals. The PCS model is placed between the SerDes and any IP processing block.
- PCS to IP – This model converts the PCS packet data into an IPv4 or IPv6 data stream, inserting and removing preamble and FCS as required.



### Notes

The models are not cycle accurate but rather cycle approximate. However, as all Ethernet traffic is transmitted and received through the cycle-accurate Ethernet interface unit (EIU), the user interface at the NAP is cycle accurate.

The models are provided for simulation purposes. They are not required or intended to be used for synthesis and implementation. When the reference design is synthesized and implemented, the hard subsystem within the device will be selected. Therefore, users can be confident the performance and resources measured during implementation are correct and are not affected by using models during simulation.

## Configuration

The reference design uses the Ethernet interface unit (EIU), which is an integral part of the Ethernet subsystem. The EIU requires configuration in order to correctly route from the required MAC to the NoC, and hence to the selected NAPs.

The configuration is performed by reading of a configuration file within the testbench; the results of this configuration file are then directly applied to the configuration port of the EIU.



#### Note

The user does not need to replicate the configuration processing block in their implementation. When the Ethernet subsystem is configured within ACE; ACE will include the configuration values within the final generated bitstream. Therefore, the EIU will be correctly configured once the device is powered up and has entered user mode.

## Implementation

The reference design includes an ACE project located in `/src/ace`. The ACE project included all the required IP configuration files, (`.acxip`), to correctly configure the clocks, PLLs, all IO and the Ethernet subsystems.

### Clocks

The following clocks are specified within the project, using the `/acxip/pll_<name>.acxip` files. The PLLs are driven by the 100MHz `sys_clk` input.

**Table 2: Clock Frequencies**

Name	Frequency (MHz)	Description
<code>noc_clk</code>	200	Default clock for the NoC. <sup>(†)</sup>
<code>eth_ref_clk</code>	900	Ethernet subsystem reference clock. <sup>(†)</sup>
<code>eth_ff0_clk</code>	600	Ethernet subsystem 0 FIFO clock. <sup>(†)</sup>
<code>eth_ff1_clk</code>	600	Ethernet subsystem 1 FIFO clock. <sup>(†)</sup>
<code>i_eth_clk</code>	507	User design Ethernet clock. Must be set to 507 MHz exactly.



#### Table Note

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

### Resets

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

**Table 3: Reset Sources**

Source	Description
<code>i_reset_n</code>	External asynchronous input. Pin configuration is specified in <code>/acxip/gpio_n0.acxip</code> .
<code>pll_eth_sys_lock</code>	Ethernet system PLL locked. Provided <code>eth_ref_clk</code> , <code>eth_ff0_clk</code> and <code>eth_ff1_clk</code> .

Source	Description
pll_noc_lock	NoC PLL locked. Provides noc_clk.
pll_usr_lock	User clock PLL locked. Provides i_eth_clk.

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

## Inputs

The following signals are input to the design.

**Table 4: Input Signals**

Source	Description
i_start	Asserted to start testing. Synchronous to i_eth_clk. The rising edge is used to initiate the packet generator. Pin configuration is specified in <code>/acxip/gpio_n0.acxip</code> .
ethernet_1_quad0_<function>	Flow control signals for Ethernet 1, Quad0 MAC. (†)
ethernet_1_m0_<function>	Status signals from Ethernet 1, Quad0 MAC. (†)
ethernet_1_quad1_<function>	Flow control signals for Ethernet 1, Quad1 MAC. (†)
ethernet_1_m1_<function>	Status signals from Ethernet 1, Quad1 MAC. (†)



### Table Note

† These signals are not used in the design as the Ethernet MAC is represented by a BFM. However, these signals must be included in the design as they are automatically included and defined by ACE when the Ethernet subsystem is configured within ACE I/O Designer.

## Outputs

There are a number of status outputs from the design.

**Table 5: Output Signals**

Source	Description
o_pkt_num0 through o_pkt_num31	Number of loopback packets received. Increases until it reaches NUM_LOOPBACK_PKTS. (1)
o_pkt_num0_oen through o_pkt_num31_oen	Active-high output enable for o_pkt_numX. Must be asserted to 1'b1 to enable output. (1)



Source	Description
o_checksum_error	Asserted if any of the loopback packets has a checksum error.
o_checksum_error_oen	Active-high output enable for o_checksum_error . Must be asserted to 1'b1 to enable output
o_pkt_size_error	Asserted if the length of any of the loopback packets does not match the IPv4 length field.
o_pkt_size_error_oen	Active-high output enable for o_pkt_size_error . Must be asserted to 1'b1 to enable output
o_payload_error	Asserted if any of the loopback packets has a payload error.
o_payload_error_oen	Active-high output enable for o_payload_error . Must be asserted to 1'b1 to enable output
ethernet_1_quad0_<function>	Flow control signals for Ethernet 1, Quad0 MAC. Not used in design. <sup>(2)</sup>
ethernet_1_quad1_<function>	Flow control signals for Ethernet 1, Quad0 MAC. Not used in design. <sup>(2)</sup>

**Table Note**

- Each of these outputs form a 32-bit bus. For this release, to match the compatible ACE version, the signals for the bus need to be individually named. In future releases each bus will be represented as a single entity.
- Outputs to the built in subsystems do not require associated \_oen signals.

## 400G Packet Mode

### Operation

The 400G packet mode Ethernet reference design demonstrates successful loopback of four 100G Ethernet packets streams. Each packet stream is generated within the FPGA fabric, then transmitted to the Ethernet subsystem, and out of the FPGA via the SerDes pins. The signals are then looped back from the SerDes, into the Ethernet subsystem, and received back in the FPGA fabric. The design targets a single Ethernet subsystem, using a single 400G Ethernet MAC, interfaced with eight SerDes lanes, each lane operating at 50Gbps.

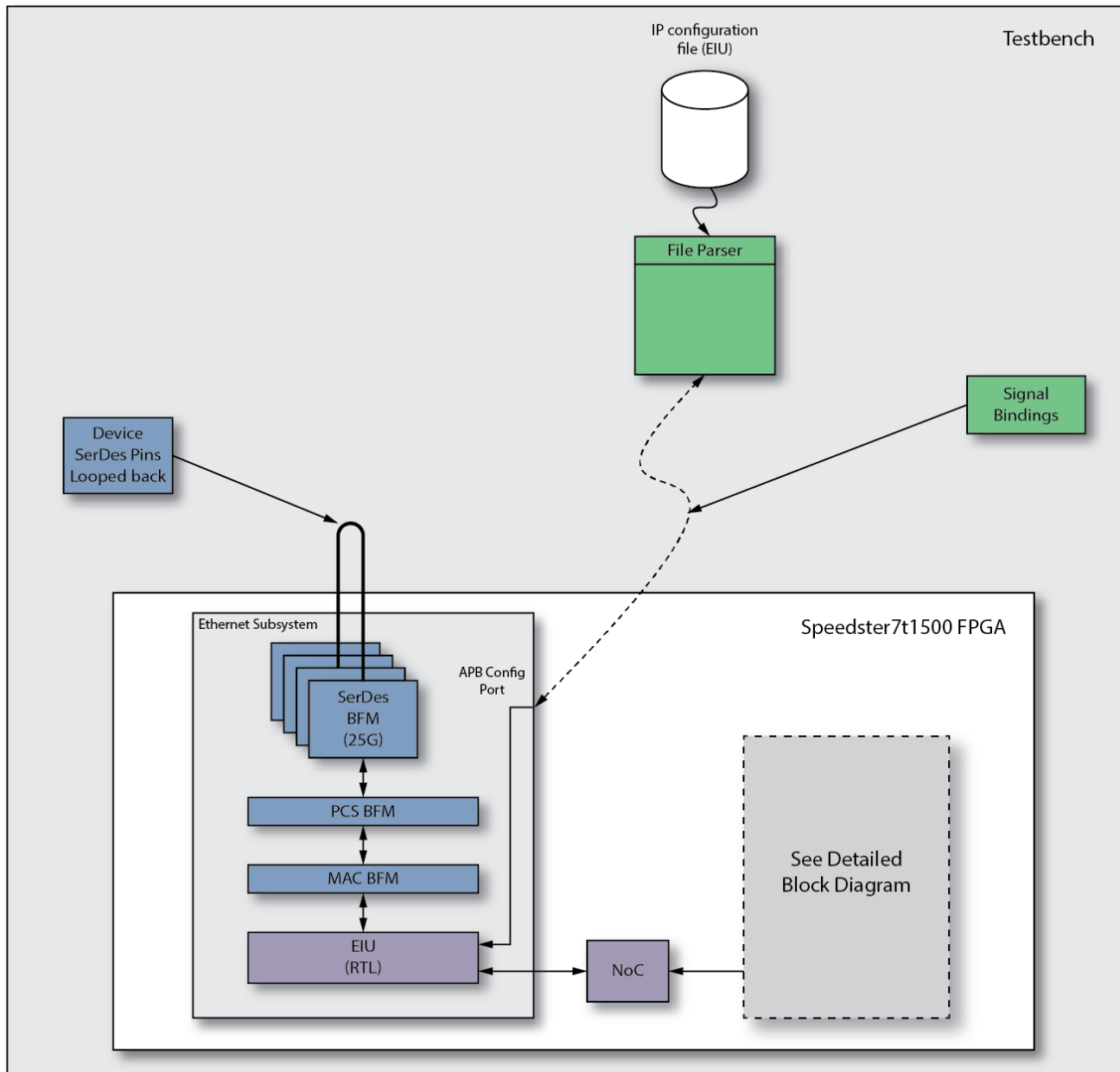
Within the reference design bundle, the design is located in the Speedster7t\_Ethernet\_ref\_design\_RD19 /8x50g\_pkt\_mode directory. All file paths listed below are relative to this directory.

### Structure

The overall structure of the testbench and target design is shown below. Data is generated within the Speedster7t device and output to the SerDes models. The SerDes pins are connected in loopback, such that the data is returned directly to the device fabric. The packet checkers within the fabric validates that the transmitted data matches the received data.

The [block diagram \(see page 10\)](#) below shows this overview of the reference design. The major elements are

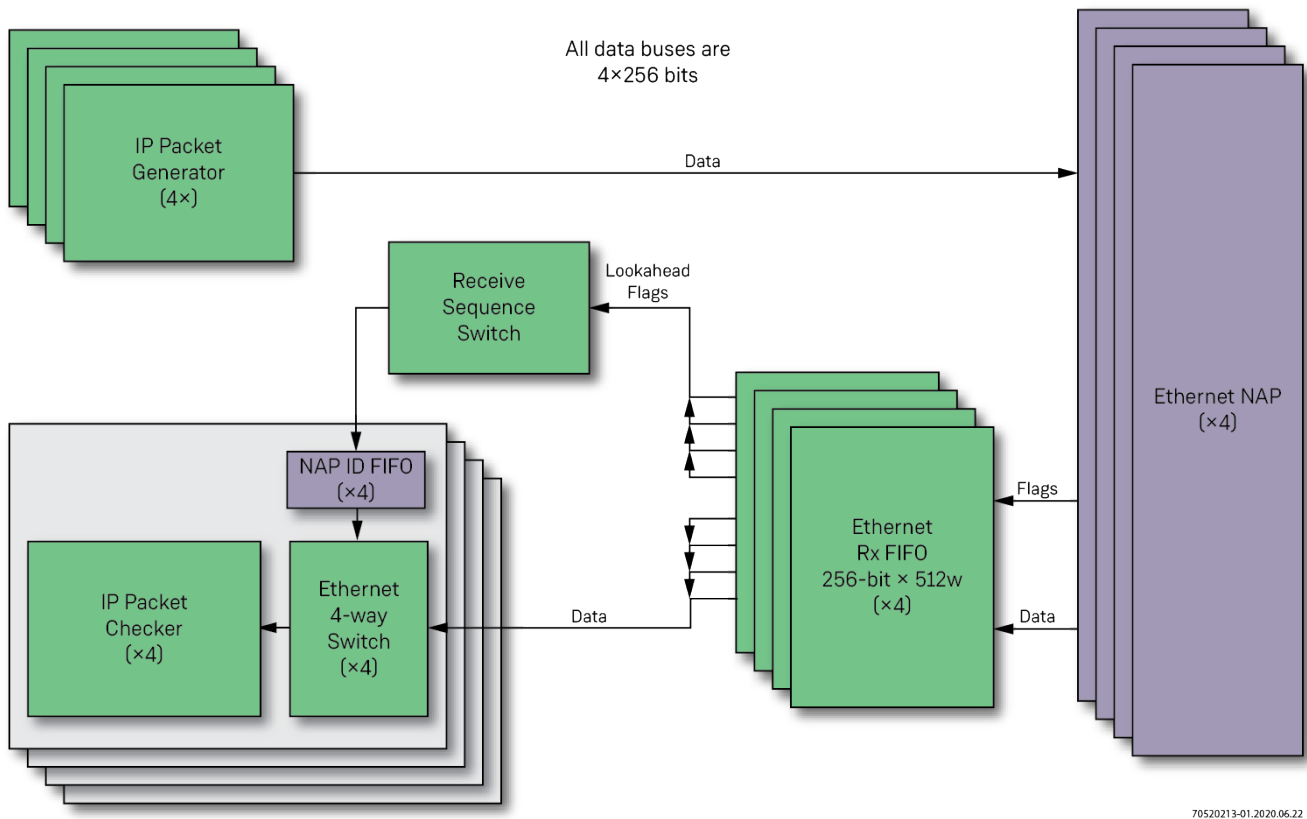
- File I/O (white) – Stimulus, checker and configuration files.
- Reference design elements (green) – These comprise of both testbench utilities and the target design modules.
- Bus functional models (blue) – These BFM's mimic the behavior of some of the device functions, as well as providing those same functions within the testbench.
- Functional logic (purple) – Device logic comprising of the full cycle-accurate RTL of the device.



64722860-02.2020.06.22

**Figure 3: Testbench and Design Overview**

The component detail of the packet mode reference design is shown in the [block diagram \(see page 11\)](#) below



**Figure 4: Packet Mode Detailed Block Diagram**

## Channel Configuration

This Ethernet reference design is configured as a single Ethernet subsystem, with the subsystem configured as a single channel operating at 400 Gbps. The channel is configured as  $8 \times 50$  Gbps; therefore within the Ethernet subsystem, each SerDes model is configured to operate at 50 Gbps, with an individual data width of 64 bits. The eight SerDes lanes are combined to create a 512-bit data stream. As the SerDes operate at double the frequency of the MAC, the SerDes data paths are multiplexed from the 512-bit double data rate to a 1024-bit single data rate bus which interfaces to the MAC BFM. The MAC interfaces directly to the EIU which receives and transmits the packets to the NoC, and hence to the four user NAPs.

Each Ethernet subsystem has 8 SerDes lanes, so this configuration demonstrates usage of all SerDes lanes within a single Ethernet subsystem.

## Ethernet Subsystem

Although this reference design only uses a single Ethernet subsystem (subsystem 0 by default), the target device, ac7t1500, contains two Ethernet subsystems. The testbench and configuration files support both subsystems, with selection of the desired subsystem controlled by a testbench define. See [Defines \(see page 15\)](#) for details as to how to switch between either of the available subsystems.

## Data Flow

The key component of 400G Ethernet design is to understand and select the most appropriate mode for packet transmission and reception. It is impractical to transmit 400 Gbps data through a single NAP (which with a 256-bit interface, would require an operating frequency approaching 2 GHz). Therefore, the NoC supports spreading the bandwidth across four NAPs, each of which operating at a fixed frequency of 507 MHz. The NoC then further supports two operating modes for these four NAPs:

- **Quad segmented mode (QSI)** (complimentary design). The four NAPs are combined to form a single 1024-bit bus. The packet is rotated around the four NAPs such that a new packet will start on the lane after the previous packet ended. For example, if the NAPs are identified as NAP-1 to NAP-4, then if the end-of-packet (EoP) word is output on NAP-2, and the next packet will assert its start-of-packet (SoP) word on NAP-3.
- **Packet mode** (this design). The four NAPs each process a whole packet, providing four 100G streams to the fabric. Each NAP operates through its 256-bit data bus interface. The NoC transmits the next available packet to whichever NAP is not currently transmitting a packet.

## Mode Considerations

The two schemes each have advantages and disadvantages:

- For QSI, packet ordering is guaranteed as the NAPs present a single 1024-bit bus for both transmit and receive. However, the width of the bus can create routing challenges, or difficulties with interfacing the data to other interface subsystems, such as memory, using the NAPs due to the NAP's native data width of 256-bits. In addition the requirement to barrel shift the packet around the bus for different packet start lanes can add complexity to a design.
- Packet mode has the advantage of having whole packets contained within a single 256-bit bus, which can then be easily connected to other NAPs, and hence to other interface subsystems or other parts of the design. However in packet mode the four streams are separate, with no guaranteed packet ordering (the Ethernet standard does not guarantee packet ordering due to the architecture of multiple routes through a network). However, if point-to-point communication is being used, and packet ordering is required, then in packet mode, receive packet ordering will need to be implemented. For transmission, the Ethernet subsystem will forward a packet as it is delivered from the NoC. This ordering is dependent upon not only the order packets are input to their respective NAPs, but also the buffering within the NAP and then the EIU. As a result, it is not possible to implement transmit packet ordering while in packet mode.

For full details on the 400G modes (which also apply to 200G operation), the two schemes are explained in depth in *Speedster7t Ethernet User Guide* (UG097).

## Receive Packet Re-order

As described above, in packet mode, any packet is received by the next available NAP. Therefore, the order of packets as received by the Ethernet subsystem can not be established by simply taking the output from each NAP in order. Again as discussed above, the Ethernet standard does not prescribe packet ordering, as the multiple routes that packets can take through a network can result in received packets being out of order. Higher level Ethernet protocols such as TCP/IP carry header information which enables a receiver to reassemble the original packet order.

Notwithstanding the above, in packet mode, it is possible to reassemble the packet order as received by the Speedster7t FPGA. Each received Ethernet packet from a NAP is accompanied by a set of flags. These flags include both a 30-bit timestamp and a 5-bit receive sequence count (full details of the flags are in *Speedster7t IP Component Library User Guide* (UG086)). Using the receive sequence count, packets can be re-order to match their arrival at the device.

## Design Components

### IP Packet Generator

The reference design has a unique requirement for received packet routing as is it required to not only re-order the received packets, but also to then direct the packets to the appropriate IP checker. Hence, the receive sequence switch requires knowledge of which IP generator originated the packet. To enable this, each IP packet generator inserts a stream identifier. In addition, to aid packet traceability and debug during simulation, each stream has its own packet count. These identifiers and counts are inserted into the MAC header as follows

- MAC source address[47:40] = Stream Identifier. Range 0x10 to 0x14
- MAC source address[39:32] = Stream packet count. 8-bit rotating value



#### Note

The above identifiers are only needed for the purposes of the reference design. The stream identifier is used to correctly route packets to the appropriate IP checker. The stream packet count is purely for convenience to assist with simulation debug.

Both of the above identifiers would be replaced in a production system with the correct MAC source address. If stream identifiers were then required to ensure that streams were directed to the appropriate processing or storage elements, higher level protocols would be used and decoded upon reception.

### Transmit Rate Control

In order to support packets of variable length, including those that may only use a few bytes of any transmitted word, the user logic and NAPs operate at 507 MHz. This clock frequency then creates a theoretical maximum data rate of 530 Gbps (507MHz × 1024 bits). This 13/10 ratio is faster than the maximum rate of 400 Gbps.

In order to ensure that the overall data rate does not exceed 400 Gbps, the [IP Packet Generator \(see page 13\)](#) is controlled by a transmit rate control module. This module monitors the 13/10 ratio between the 507 MHz clock and the valid words written, and where necessary, directs the [IP Packet Generator \(see page 13\)](#) to hold generation of the next packet, thereby ensuring the overall data rate does not exceed the maximum permissible.

### Ethernet RX FIFO (256 × 512)

A requirement of 400G operation is that none of the four NAPs can stall packet reception. If stalling were to occur, then it is possible that data will be prevented from being received by other NAPs in the same column, leading to data loss and corruption.

As the received packets need to be re-ordered, this necessitates holding packets until they are ready to be played out. Therefore, each NAP has an associated Ethernet receive FIFO to buffer the received packets. For the main data path, the FIFO is configured as 256 bits wide by 512 entries deep, and is composed of four BRAM72K\_FIFO. This space is sufficient to store a whole jumbo (9 KB) packet. In addition to the main data path, this FIFO supports a side channel which stores the received sequence count and stream identifier. These values are presented in a lookahead output for the [Receive Sequence Switch \(see page 14\)](#).



#### Note

The stream identifier output is based upon the same byte location as the IP generator. For a production system, if stream identifiers are used in higher protocols, then the Ethernet FIFO will need modifying to parse the protocols and extract the appropriate stream identifiers.

## Receive Sequence Switch

As stated, for the purposes of the reference design, it is necessary to not only re-order the received packets (based on packet sequence ID), but also to route the packets in order to the appropriate IP checker, based on the stream identifiers.

The receive sequence switch operates its own receive sequence count (`current_seq_id`). It compares that count against those available from the four Ethernet receive FIFO lookahead channels (`rx_seq_id[3:0]`). If the receive sequence count matches, the receive sequence switch extracts the associated stream identifier, `rx_stream_id` (from the same lookahead channel). The identifier informs the switch which IP checker the packet is intended for. The receive sequence switch then writes the number of the relevant Ethernet receive FIFO (`seq_id_valid_match`) into a small, eight-entry NAP ID FIFO (`nap_id_fifo`) that is associated with each IP checker. The result is that each IP checker has a sequence of identifiers that indicate which Ethernet receive FIFO contains its next packet.

Having successfully identified the destination of a packet, the receive sequence switch increments the its receive sequence count and the process continues.

## Ethernet Receive Four-way Switch

The final part of the receive process is the four-way switch associated with each IP checker. The four-way switch receives the output from the NAP ID FIFO and determines if any other checker is currently receiving packets from the selected Ethernet receive FIFO (`active_ch_sum`). If so, the switch waits until that packet is completed.

Once the input channel is available, the switch will then select the desired Ethernet receive FIFO and direct its output to the IP checker. At the same time it indicates that the channel is now in use (`active_ch`). In addition the switch indicates to the NAP ID FIFO that it has started to download the packet (`frame_start`), allowing the NAP ID FIFO to perform a lookahead and pre-select the next required input channel.



### Note

The above design components are provided "as is" for the purposes of demonstrating the reference design. It is recognized that some elements, modified or otherwise, may be used within production designs. It is the user's responsibility to determine that these elements meet their needs and to fully verify their operation within their own systems.

## Throughput Monitors

To measure throughput, a set of Ethernet stream monitors are instantiated within the design. These monitors perform the following functions:

- Check for legal sequence. The monitors will issue an error if an EoP is seen outside of a valid frame, or a SoP within a valid frame.
- Count numbers of packets, and number of bytes.
- Display average throughput in Gbps.

The throughput monitors can either be enabled with external signals, or they can auto-start on the first valid packet and finish measurements when a certain number of packets is counted.

## Throughput

For this design, because packet re-ordering is implemented solely for the purposes of checking received packets in sequence, there is a degree of throughput loss. The cause is that when packets for the same stream arrive at two different NAPs, the second packet has to be held for the first to be processed.

For this reference design, with mixed packet sizes, a throughput of ~330 Gbps is achieved. To achieve higher throughput in a user design that implements packet re-ordering, the user needs to implement greater buffering, or increase the frequency of the downstream processing.

In general terms it is recommended that packet mode is used in designs where packet re-ordering is not required, where each received packet can be processed independently. In this scenario, full 400 Gbps throughput should be achieved.

## Testbench Environment


### Parameters

The simulation testbench can be configured with a number of parameters.

**Table 6: Testbench Parameters**

Name	Description
NUM_LOOPBACK_PKTS	Data loopback test number of packets. This value is per 100G stream, therefore 4x this number of packets will be sent and received in total. 0 = run continuously.
CFG_FILENAME	EIU configuration filename.

### Defines

Name	Description
ACX_USE_ETHERNET1	Use the simulation model Ethernet 1 subsystem. When enabled this affects the location of the NAPs (to correspond to the appropriate Ethernet subsystem), the destination of the configuration, and the selection of SerDes pins for data injection). By default this is undefined, selecting the Ethernet 0 subsystem:  <div>  This define only affects the testbench selection of Ethernet subsystem. The reference design itself is unchanged, regardless of the selected Ethernet subsystem. </div>
ACX_DUMP_SIM_SIGNALS	Enable writing of simulation waveforms to a file. If disabled, results in faster simulation; however, no waveforms will be captured.

### Models

The Ethernet reference design uses a number of models to mimic functionality within the Speedster7t device and significantly accelerate simulation speeds. In addition these models are required by the testbench in order to provide encoding of any transmitted data and decoding of any received data.

- SerDes – All the SerDes interfaces are replaced by a SerDes bus functional model (BFM). This model serializes the data to mimic the operation of the true SerDes. The model includes link initialization using magic packets.

**Note**

This SerDes model does not perform any of the clock recovery or sophisticated link initialization found in the actual Speedster7t SerDes.

- PCS – A 64/66b BFM of a PCS is used. The scrambling and descrambling of the data stream match the 64/66b specifications. The PCS model interfaces between the data transmitted by the SerDes model and converts this data to a packet format, with sop, eop, mod, valid and data signals. The PCS model is placed between the SerDes and any IP processing block.
- PCS to IP – This model converts the PCS packet data into an IPv4 or IPv6 data stream, inserting and removing preamble and FCS as required.

**Notes**

The models are not cycle accurate but rather cycle approximate. However, as all Ethernet traffic is transmitted and received through the cycle-accurate Ethernet interface unit (EIU), the user interface at the NAP is cycle accurate.

The models are provided for simulation purposes. They are not required or intended to be used for synthesis and implementation. When the reference design is synthesized and implemented, the hard subsystem within the device will be selected. Therefore, users can be confident the performance and resources measured during implementation are correct and are not affected by using models during simulation.

## Configuration

The reference design uses the Ethernet interface unit (EIU), which is an integral part of the Ethernet subsystem. The EIU requires configuration in order to correctly route from the required MAC to the NoC, and hence to the selected NAPs.

The configuration is performed by reading of a configuration file within the testbench; the results of this configuration file are then directly applied to the configuration port of the EIU.

**Note**

The user does not need to replicate the configuration processing block in their implementation. When the Ethernet subsystem is configured within ACE; ACE will include the configuration values within the final generated bitstream. Therefore, the EIU will be correctly configured once the device is powered up and has entered user mode.

## Implementation

The reference design includes an ACE project located in `/src/ace`. The ACE project included all the required IP configuration files, (.acxip), to correctly configure the clocks, PLLs, all IO and the Ethernet subsystems.




## Clocks

The following clocks are specified within the project, using the `/acxip/pll_<name>.acxip` files. The PLLs are driven by the 100MHz `sys_clk` input.

**Table 7: Clock Frequencies**

Name	Frequency (MHz)	Description
<code>noc_clk</code>	200	Default clock for the NoC. <sup>(†)</sup>
<code>eth_ref_clk</code>	900	Ethernet subsystem reference clock. <sup>(†)</sup>
<code>eth_ff_clk</code>	800	Ethernet subsystem FIFO clocks. <sup>(†)</sup>
<code>i_eth_clk</code>	507	User design Ethernet clock. Must be set to 507MHz exactly.


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

## Resets

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

**Table 8: Reset Sources**

Source	Description
<code>i_reset_n</code>	External asynchronous input. Pin configuration is specified in <code>/acxip/gpio_n0.acxip</code> .
<code>pll_noc_lock</code>	NoC PLL locked. Provides <code>noc_clk</code> .
<code>pll_eth_ref_lock</code>	Ethernet reference PLL locked. Provides <code>eth_ref_clk</code> .
<code>pll_eth_ff_lock</code>	Ethernet FIFO PLL locked. Provides <code>eth_ff_clk</code> .
<code>pll_usr_lock</code>	User clock PLL locked. Provides <code>i_eth_clk</code> .

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

## Inputs

The following signals are input to the design.

**Table 9: Input Signals**

Source	Description
i_start	Asserted to start test. Synchronous to i_eth_clk. The rising edge is used to initiate the packet generator. Pin configuration is specified in /acxip/gpio_n0.acxip.
ethernet_1_m0_<function>	Flow control and status signals from Ethernet 1, 400G MAC 0. <sup>(†)</sup>
ethernet_1_m0_rx /tx_bufferX_at_threshold	Buffer level status signals for Ethernet 1, 400G MAC 0 transmit and receive buffers. <sup>(†)</sup>
ethernet_1_m1_<function>	Flow control and status signals from Ethernet 1, 400G MAC 1. <sup>(†)</sup>
ethernet_1_m1_rx /tx_bufferX_at_threshold	Buffer level status signals for Ethernet 1, 400G MAC 1 transmit and receive buffers. <sup>(†)</sup>

**Table Note**

† These signals are not used in the design as the Ethernet MAC is represented by a BFM. However, the signals must be included in the design as they are automatically included and defined by ACE when the Ethernet subsystem is configured within ACE I/O Designer.

## Outputs

There are a number of status outputs from the design

**Table 10: Output Signals**

Source	Description
o_pkt_num0 through o_pkt_num31	Number of loopback packets received. Increases until it reaches NUM_LOOPBACK_PKTS. <sup>(1)</sup>
o_pkt_num0_oen through o_pkt_num31_oen	Active-high output enable for o_pkt_numX. Must be asserted to 1'b1 to enable output. <sup>(1)</sup>
o_checksum_error	Asserted if any of the loopback packets has a checksum error.
o_checksum_error_oen	Active-high output enable for o_checksum_error. Must be asserted to 1'b1 to enable output
o_pkt_size_error	Asserted if the length of any of the loopback packets does not match the IPv4 length field.
o_pkt_size_error_oen	Active-high output enable for o_pkt_size_error. Must be asserted to 1'b1 to enable output.
o_payload_error	Asserted if any of the loopback packets has a payload error.

Source	Description
o_payload_error_oen	Active-high output enable for o_payload_error. Must be asserted to 1'b1 to enable output
ethernet_1_m0_<function>	Flow control signals for Ethernet 1, 400G MAC 0. Not used in design. <sup>(2)</sup>
ethernet_1_m1_<function>	Flow control signals for Ethernet 1, 400G MAC 1. Not used in design. <sup>(2)</sup>

**Table Note**

- Each of these outputs form a 32-bit bus. For this release, to match the compatible ACE version, the signals for the bus need to be individually named. In future releases each bus will be represented as a single entity.
- Outputs to the built in subsystems do not require associated \_oen signals.

## Instructions

### Installing the Reference Design

#### Downloading

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

#### Packaging

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

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

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

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

#### Operating System

##### Linux

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

##### Windows

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

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

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

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



#### Note

For correct operation of any script flow, ACE should be installed in a directory without spaces in the path name, e.g., `C:\achronix\8.1.1\Achronix_CAD_Environment`. Additionally the environment variable `ACE_INSTALL_DIR` must use "/" as path separators rather than "\", for example:

```
ACE_INSTALL_DIR = C:/achronix/8.1.1/Achronix_CAD_Environment/Achronix
```

## Tool Versions

All designs were tested with the following tools:

**Table 11: Minimum Tool Versions**

Software	Version
ACE	8.2
ACE IO BFM sim package	8.2.update1
Synplify Pro	Q-2020.03X
Modelsim Questa	10.7c-1
Synopsys VCS	O-2018.09-SP1-1

## Environment Variables

### ACE\_INSTALL\_DIR

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



#### Note

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

- Windows – `C:\achronix\8.2\Achronix_CAD_Environment`
- Linux – `/opt/achronix/ace/8.2`

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

```
ACE_INSTALL_DIR = C:/achronix/8.2/Achronix_CAD_Environment/Achronix
```

## Directory Structure

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

**Table 12: Design Directory Structure**

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

## Language Support

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

## Constraint Files

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

**Table 13: Constraint File Details**

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

## I/O Ring Constraint Files

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

**Table 14: 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_util.xml	Used by ACE to generate a combined utilization report, combining the fabric and I/O ring resources.

## I/O Ring Simulation Package

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

### Description

The I/O ring simulation structure provides full RTL code for the NoC and BFM models of the surrounding interface subsystems. This structure is wrapped within a SystemVerilog module named per device, i.e., ac7t1500. The user needs to instantiate one instance of this module within their top-level testbench.

In addition, the simulation package provides binding macros such that the user can bind between elements of their design and the same elements within the device. For example, the design may instantiate a NoC access point (NAP). It is then necessary to bind this NAP instance to the NAP in the correct location within the NoC by using the `ACX\_BIND\_NAP\_SLAVE`, `ACX\_BIND\_NAP\_MASTER`, `ACX\_BIND\_NAP\_HORIZONTAL`, or `ACX\_BIND\_NAP\_VERTICAL` macro, whichever is appropriate for the design.

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

### Version Control

The I/O ring simulation package is version controlled. Within a release, new functions may be added and older functions may be deprecated or replaced. The release is indicated both in the package name (ACE\_<major>.<minor>.<patch>\_IO\_BFM\_sim\_<update>.zip/tgz) and in the readme file placed in the root directory of the package.

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

#### Code

```
// For this example the FPGA instance is ac7t1500
initial begin
```

```
// Ensure correct version of sim package is being used
// This design requires 8.1.2.update2 as a minimum
ac7t1500.require_version(8, 1, 2, 2);
end
```

## require\_version() Task

The `require_version` task has four arguments. In order

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

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



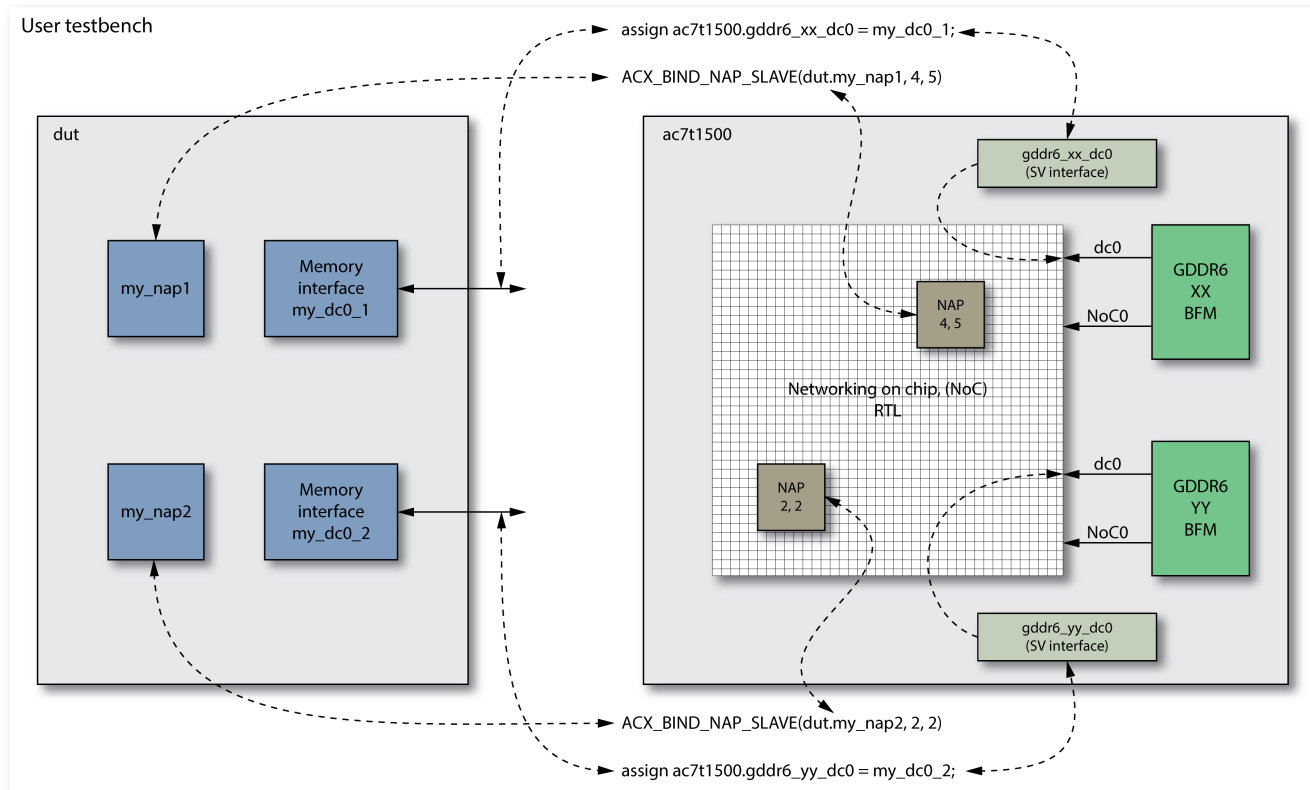
### Note

The values can be expressed either as numbers (0-9) or as strings ( "0" – "9" ) or as letters ( "a/A", "b/B" ), with the letters "a" and "b" represent 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) will precede the full 8.3 release (designated as 8,3,0,0).

## Example Design

An example structure of the I/O ring simulation, combined within a testbench, and with a design under test is shown in the [diagram \(see page 25\)](#) below. This example shows the macros required for the slave NAPs, and the DCIs for two instances of the GDDR6 subsystem. For other forms of NAPs, or for other DCI types, such as DDR, consult the [Bind Macros \(see page 27\)](#) and [Direct-Connection Interfaces \(see page 28\)](#) tables.





62297007-01.2020.08.26

**Figure 5: Example I/O Ring Simulation Structure**

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

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

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

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

// Set the verbosity options on the messages
initial begin
    ac7t1500.verbosity = 3;
end

// -----
// Bind NAPs
```

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

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

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

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

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



#### Note

When using bind macros, the user is able to specify the column and row coordinates of the target NAP. To ensure consistency between simulation and silicon, the user should add matching placement constraints to the ACE placement .pdc file, for example:

#### In simulation

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

#### In place and route

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

## 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 only starts to drive stimulus into the device once this signal is asserted (shown in the example above by use of a testbench `chip_ready` signal).

## Bind Macros

The following bind statements are available.

**Table 15: Bind Macros**

Macro	Arguments	Description
ACX_BIND_NAP_HORIZONTAL	user_nap_instance, noc_column, noc_row	To bind a horizontal streaming NAP, instance ACX_NAP_HORIZONTAL.
ACX_BIND_NAP_VERTICAL	user_nap_instance, noc_column, noc_row	To bind a vertical streaming NAP, instance ACX_NAP_VERTICAL.
ACX_BIND_NAP_AXI_MASTER	user_nap_instance, noc_column, noc_row	To bind an AXI master NAP, instance ACX_NAP_AXI_MASTER.
ACX_BIND_NAP_AXI_SLAVE	user_nap_instance, noc_column, noc_row	To bind an AXI slave NAP, instance ACX_NAP_AXI_SLAVE.

## Direct-Connect Interfaces

The following direct-connect interfaces are available.

**Table 16: Direct-Connect Interfaces**

Subsystem	Interface Name	Physical Location	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

Subsystem	Interface Name	Physical Location	GDDR6 Channel	SystemVerilog Interface Type	Data Width	Address Width
GDDR6	gddr6_5_dc0	East 1	0	t_ACX_AXI4	512	33
GDDR6	gddr6_5_dc1	East 1	1	t_ACX_AXI4	512	33
GDDR6	gddr6_6_dc0	East 2	0	t_ACX_AXI4	512	33
GDDR6	gddr6_6_dc1	East 2	1	t_ACX_AXI4	512	33
DDR	ddr4_dc0	South	NA	t_ACX_AXI4	512	40

**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.

## Clock Frequencies

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

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

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

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

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

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

**Note**

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

The following clock frequency interfaces are available

**Table 17: Clock Frequency Interfaces**

Subsystem	Interface Name	Physical Location	GDDR6 Channel
GDDR6	gddr6_0_noc0_clk	West 0 NoC	0
GDDR6	gddr6_0_noc1_clk	West 0 NoC	1
GDDR6	gddr6_1_noc0_clk	West 1 NoC	0
GDDR6	gddr6_1_noc1_clk	West 1 NoC	1
GDDR6	gddr6_2_noc0_clk	West 2 NoC	0
GDDR6	gddr6_2_noc1_clk	West 2 NoC	1
GDDR6	gddr6_3_noc0_clk	West 3 NoC	0
GDDR6	gddr6_3_noc1_clk	West 3 NoC	1
GDDR6	gddr6_4_noc0_clk	East 0 NoC	0
GDDR6	gddr6_4_noc1_clk	East 0 NoC	1
GDDR6	gddr6_5_noc0_clk	East 1 NoC	0
GDDR6	gddr6_5_noc1_clk	East 1 NoC	1
GDDR6	gddr6_6_noc0_clk	East 2 NoC	0
GDDR6	gddr6_6_noc1_clk	East 2 NoC	1
GDDR6	gddr6_7_noc0_clk	East 3 NoC	0
GDDR6	gddr6_7_noc1_clk	East 3 NoC	1
GDDR6	gddr6_1_dc0_clk	West 1 DCI	0
GDDR6	gddr6_1_dc1_clk	West 1 DCI	0
GDDR6	gddr6_2_dc0_clk	West 2 DCI	0
GDDR6	gddr6_2_dc1_clk	West 2 DCI	1
GDDR6	gddr6_5_dc0_clk	East 1 DCI	0
GDDR6	gddr6_5_dc1_clk	East 1 DCI	1
GDDR6	gddr6_6_dc0_clk	East 2 DCI	0
GDDR6	gddr6_6_dc1_clk	East 2 DCI	1
DDR	ddr4_noc0_clk	South NoC	NA

Subsystem	Interface Name	Physical Location	GDDR6 Channel
DDR	ddr4_dc0_clk	South DCI	NA
PCIe	pciex16_clk	Gen5 PCIe x16	NA
Configuration	cfg_clk	System wide configuration clock	NA

## Configuration

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

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

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

## Startup Sequence

While the task `fcu.configure()` is processing the configuration (including waiting for any polling to return a valid value), the [Chip Status Output \(see page 27\)](#) is not asserted. This behavior mirrors that of the device where the device will only enter user mode once configuration is completed.

The simulation testbench can issue configuration processes as shown above, and once the [Chip Status Output \(see page 27\)](#) is asserted, the testbench will know 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 18: Configuration Subsystem Names**

Subsystem	Interface Name	Physical Location
GDDR6	gddr6_0	West 0
GDDR6	gddr6_1	West 1
GDDR6	gddr6_2	West 2
GDDR6	gddr6_3	West 3
GDDR6	gddr6_4	East 0
GDDR6	gddr6_5	East 1
GDDR6	gddr6_6	East 2
GDDR6	gddr6_7	East 3
DDR	ddr4	South
Ethernet	ethernet0	North
Ethernet	ethernet1	North

**Table Note**

Configuration subsystem interface names are case sensitive

**Configuration File Format**

The configuration file has the following format:

```

# -----
# Config 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
"p" - poll until bit set

```

```

# Address is 28-bit, (7 hex characters). This supports the configuration
# memory space of an interface subsystem

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

# For reads, put 0x0 for the data
# For verify put the expected data value
# For poll, set relevant bits to 1.
# Poll will complete when all bits set to 1 are asserted

# Examples

# Writes
w 00005c0 76543210
w 0000014 00004064

# Reads
r 00005c0 00000000
r 0000014 00000000

# Verify
v 00005c0 76543210
v 0000014 00004064

# Poll
# Waiting for bits [17:16] and [0] to be asserted
p 0123456 00030001

```

## SystemVerilog Interfaces

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



### Note

The interface below is only available in the simulation environment. For code that must be synthesized, users need to define their own SystemVerilog interfaces, or use one of the interfaces predefined within the reference designs.

```

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

    logic                clk;           // Clock reference
    logic                awvalid;       // AXI Interface
    logic                awready;
    logic [ADDR_WIDTH-1:0] awaddr;
    logic [LEN_WIDTH-1:0]  awlen;
    logic [8-1:0]         awid;
    logic [4-1:0]         awqos;
    logic [2-1:0]         awburst;

```



```

logic                                awlock;
logic [3 -1:0]                      awsize;
logic [3 -1:0]                      awregion;
logic [3:0]                          awcache;
logic [2:0]                          awprot;
logic                                wvalid;
logic                                wready;
logic [DATA_WIDTH -1:0]             wdata;
logic [(DATA_WIDTH/8) -1:0]         wstrb;
logic                                wlast;
logic                                aready;
logic [DATA_WIDTH -1:0]             rdata;
logic                                rlast;
logic [2 -1:0]                      rresp;
logic                                rvalid;
logic [8 -1:0]                      rid;
logic [ADDR_WIDTH -1:0]             araddr;
logic [LEN_WIDTH -1:0]             arlen;
logic [8 -1:0]                      arid;
logic [4 -1:0]                      arqos;
logic [2 -1:0]                      arburst;
logic                                arlock;
logic [3 -1:0]                      arsize;
logic                                arvalid;
logic [3 -1:0]                      arregion;
logic [3:0]                          arcache;
logic [2:0]                          arprot;
logic                                aresetn;
logic                                rready;
logic                                bvalid;
logic                                bready;
logic [2 -1:0]                      bresp;
logic [8 -1:0]                      bid;

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

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

endinterface : t_ACX_AXI4

```

## Installation

There is a simulation package per device, available for both Linux and Windows. The packages are named `<device>_IO_BFM_Sim_<version>.tgz` for Linux and `<device>_IO_BFM_Sim_<version>.zip` for Windows. The packages are available on the Achronix self-service FTP site at [secure.achronix.com](https://secure.achronix.com), located in the `/Achronix/ACE/Speedster7t` folder. As each device package is independent, it is possible to either download only the device the user is targeting, or all device packages.

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

## ACE Integration

### Upgrading an Existing Installation

If a version of the simulation package was previously installed into ACE, it is recommended to first delete the existing simulation package before upgrading to ensure the integrity of the new installation. To delete an existing package, navigate to `<ACE_INSTALL_DIR/system/data/yuma-alpha-rev0` and remove the `/sim` directory. Then return to the root of the ACE installation and proceed with the instructions for [First Installation](#) (see page 34) below.

### First Installation

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



#### Warning!

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

## Standalone

In certain instances it may not be possible for a user to modify their existing ACE installation. In these cases it is possible to install the package separately and to simulate using files from both this simulation package and the existing simulation files within ACE.

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



#### Note

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

## 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 environment variable ACX\_DEVICE\_INSTALL\_DIR is used to select the device I/O ring simulation files. It should be set to the base directory of the device files within the I/O ring simulation package. For example if the ac7t1500 device is selected, then the device base directory is yuma-alpha-rev0.

When the installation is done as ACE integration mode, then the following setting should be used:

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

When the installation is done as standalone, the the following setting should be used:

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

## Simulating the Reference Design

### Supported Simulators

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

### Location

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

### Simulation Flows

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

### Standalone

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

### Full-Chip BFM

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

The full-chip BFM simulations require the I/O Ring Simulation Package to have been downloaded and installed.

## Full-Chip RTL

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



### Note

To obtain the encrypted RTL of the various subsystems, a second licensed simulation package is required. Please contact Achronix Support to arrange licensing and access to this package.

## Build Options

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

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

## Prerequisites

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

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

## Auto File List Generation

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

## Files

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

- `makefile` – Makefile supporting the various simulation flows. Default target is to compile and run the simulation.
- `system_files_bfm.f` – (Full-chip BFM flow only) List of system files used by the full-chip BFM flow. Any user defines can also be added to this file.

- `system_files_rtl.f` – (Full-chip RTL flow only) List of system files used by the full-chip RTL flow. Also any defines required to specify which subsystems should be cycle-accurate RTL rather than BFM. The defines are named as `<subsystem name>_FULL`, i.e., `GDDR6_W2_FULL`. Any user defines can also be added to this file.

## VCS Only

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

## QuestaSim Only

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

## Running the Simulation

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

## VCS

### code

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

## QuestaSim

### code

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

## QuestaSim Non-Makefile Flow

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

1. Navigate to `sim/questa`

## 2. Launch QuestaSim GUI. Normally:

```
$ vsim
```

Within the QuestaSim GUI launch the script

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



### Note

The QuestaSim script uses the `ACE_INSTALL_DIR` environment variable. For correct operation of this, (or any other), script flow, ACE should be installed in a directory without spaces in the path name, e.g., `C:\achronix\8.2\Achronix_CAD_Environment`. Additionally the environment variable `ACE_INSTALL_DIR` must use "/" as path separators rather than "\", for example:

```
ACE_INSTALL_DIR = C:/achronix/8.2/Achronix_CAD_Environment/Achronix
```



### Note

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

## Results Verification

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

## Building the Reference Design

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

## Prerequisites

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

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

## Batch Flow

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

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

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

## Makefile Options

The makefile supports multiple build flow options

### code

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

## Constraint Files

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

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

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

## GUI Flow

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



### Note

When using the GUI projects, any generated files will be placed beneath the GUI project file directory.

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

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



## Revision History

Version	Date	Description
1.0	20 Apr 2020	<ul style="list-style-type: none"><li>Initial Release.</li></ul>
1.1	28 May 2020	<ul style="list-style-type: none"><li>Added note regarding only single Ethernet subsystem available.</li><li>Added description on how clock periods are defined in testbench.</li><li>Added description on how testbench performs configuration.</li><li>Corrected launch of QuestaSim GUI command.</li><li>Updated example instantiation of AC7t1500.</li></ul>
2.0	22 Jun 2020	<ul style="list-style-type: none"><li>Added new sections:<ul style="list-style-type: none"><li>400G Packet Mode (see page 9)</li><li>400G Quad Segmented Mode</li></ul></li></ul>
3.0	29 Jul 2020	<ul style="list-style-type: none"><li>Updated signal I/O to match GPIO generator.</li><li>Add support for both Ethernet subsystems to all designs.</li><li>Required ACE version updated to 8.2.</li></ul>
3.1	31 Aug 2020	<ul style="list-style-type: none"><li>Removed 400G Quad Segmented Mode design</li></ul>



Achronix Semiconductor Corporation

2903 Bunker Hill Lane  
Santa Clara, CA 95054  
USA

Website: [www.achronix.com](http://www.achronix.com)  
E-mail : [info@achronix.com](mailto:info@achronix.com)

---

Copyright © 2020 Achronix Semiconductor Corporation. All rights reserved. Achronix, Speedcore, Speedster, and ACE are trademarks of Achronix Semiconductor Corporation in the U.S. and/or other countries All other trademarks are the property of their respective owners. All specifications subject to change without notice.

## Notice of Disclaimer

The information given in this document is believed to be accurate and reliable. However, Achronix Semiconductor Corporation does not give any representations or warranties as to the completeness or accuracy of such information and shall have no liability for the use of the information contained herein. Achronix Semiconductor Corporation reserves the right to make changes to this document and the information contained herein at any time and without notice. All Achronix trademarks, registered trademarks, disclaimers and patents are listed at <http://www.achronix.com/legal>.