Accelerator Functional Unit Developer Guide

Intel FPGA Programmable Acceleration Card N3000 Variants

Updated for Intel® Acceleration Stack for Intel® Xeon® CPU with FPGAs: **1.3.1**

Online Version
Send Feedback

UG-20248

ID: 683190
Version: 2022.07.15

Contents

1. About This Document

This document serves as a high level guide for system architects and hardware developers in developing Accelerator Functional Units (AFUs) for both:

- Intel FPGA Programmable Acceleration Card N3000
- Intel FPGA Programmable Acceleration Card N3000-N

This document is organized as follows:

1. Document Introduction and required background knowledge
2. High Level Description
3. Developing AFUs
4. Debugging AFUs

1.1. Acronym List

Acronym	Expansion	Description
Intel FPGA PAC N3000-N (referred to as N3000 for this document)	Intel FPGA Programmable Acceleration Card N3000-N	Intel FPGA Programmable Acceleration Card N3000-N is a full-duplex 100 Gbps in-system re-programmable acceleration card for multi-workload networking application acceleration.
AFU	Accelerator Functional Unit	Hardware Accelerator implemented in FPGA logic which offloads a computational operation for an application from the CPU to improve performance.
AF	Acceleration Function	Compiled Hardware Accelerator image implemented in FPGA logic that accelerates an application.
API	Application Programming Interface	A set of subroutine definitions, protocols, and tools for building software applications.
DPDK	Data Plane Development Kit	The Data Plane Development Kit consists of libraries to accelerate packet processing workloads running on many CPU architectures, including x86, POWER and ARM processors. DPDK runs mostly on Linux with a FreeBSD port available for a subset of DPDK features. The Open Source BSD License DPDK licenses DPDK.
FIU	FPGA Interface Unit	FIU is a platform interface layer that acts as a bridge between platform interfaces like PCIe* and AFU-side interfaces such as CCI-P.
OPAE	Open Programmable Acceleration Engine	The OPAE is a set of drivers, utilities, and API's for managing and accessing AFs.

2. Introduction

Before using this guide, refer to the user guide that corresponds with your card: *Intel® Acceleration Stack User Guide: Intel FPGA Programmable Acceleration Card N3000* or *Intel Acceleration Stack User Guide: Intel FPGA Programmable Acceleration Card N3000-N*. Both user guides provide an overview of the capabilities of the Intel FPGA PAC N3000 and Intel FPGA PAC N3000-N, referred to as N3000 throughout this document. Both user guides provide instructions for installation and setup of hardware and software components of the stack, including the Open Programmable Acceleration Engine (OPAE) tools used in running diagnostic tools and remotely loading FPGA images. It is essential to familiarize yourself with the concepts developed and to complete the installation and setup procedures covered in both user guides.

To perform AFU development, install the Intel Acceleration Stack for Development as described in the user guide that corresponds with your card: *Intel Acceleration Stack User Guide: Intel FPGA Programmable Acceleration Card N3000* or *Intel Acceleration Stack User Guide: Intel FPGA Programmable Acceleration Card N3000-N*.

Related Information

- Intel Acceleration Stack User Guide: Intel FPGA Programmable Acceleration Card N3000
- Intel Acceleration Stack User Guide: Intel FPGA Programmable Acceleration Card N3000-N

2.1. Base Knowledge and Skills Prerequisites

The Intel Acceleration Stack is an advanced application of FPGA technology. The platform-level complexity has been abstracted away for the AFU developer by the inclusion of all interfaces in the FPGA factory image and a standard Core Cache Interface (CCI-P) interface for host connectivity to your AFU.

This guide assumes the following FPGA logic design-related knowledge and skills:

- FPGA compilation flows including the Intel Quartus® Prime Pro Edition design flow.
- Static Timing closure, including familiarity with the Timing Analyzer tool in Intel Quartus Prime Pro Edition, applying timing constraints, Synopsys* Design Constraints (.sdc) language and Tcl scripting, and design methods to close on timing critical paths.
- RTL and coding practices to create synthesized logic.
- High level synthesis (HLS) and Platform Designer design entry tools are supported.
- RTL simulation tools.
- Signal Tap Logic Analyzer tool in the Intel Quartus Prime Pro Edition software.

2.1.1. Considerations

If you are familiar with other Intel Acceleration Products, there are similarities and differences between the Intel Programmable Acceleration Card with Intel Arria® 10 GX FPGA operation:

- Similarities:
  - CCI-P interface between user logic and Intel supplied PCIe interface
  - OPAE kernel driver and tools for diagnostics and remote debugging
  - FPGA region dedicated to OPAE management logic
- Differences:
  - Partial reconfiguration is not supported
    - The FPGA is a flat design loaded by on board flash
    - User must include encrypted blocks for board management
  - ASE simulation is not supported
  - Automated simulation and synthesis environment set up are not supported

  Note: The OPAE version for the N3000 is not compatible, with previous and current versions of OPAE supporting Intel PAC with Intel Arria 10 GX FPGA.

3. High Level Description

The N3000 provides you with a rapid design methodology for creating complex FPGA and Intel Xeon® networking applications. You are provided the following:

- An Intel certified board with Intel Arria 10 GT FPGA, PCIe interfaces, external memories, board management controller and Ethernet network interface devices.
- FPGA factory image design demonstrating all interfaces.
- Software tools for running board diagnostics with FPGA factory image, performing FPGA remote update, and reading board sensors.
- Internal FPGA Nios® II controller and firmware for Ethernet re-timer provisioning and board control functions. You must include this block in your design.
- The FPGA design flow, which supports flexible Ethernet data flow configurations supporting your developed packet processing functions.

The Intel supplied board, FPGA IP blocks and software allow you to focus on your value added functionality.

3.1. Steps for Creating Your AFU

The following steps are suggested for designing a custom FPGA application for the N3000:

1. Become familiar with the board and FPGA block diagrams, interfaces and code provided within the N3000 factory image.
2. Review the *Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*. You must follow the interface requirements and include required registers in your design for proper N3000 operation.

   In addition, the OPAE Basic Building Blocks wiki provides CCI-P tutorials and basic building blocks (BBB) for interfacing your AFU. You are strongly encouraged to review this resource. The Memory Properties Factory BBB is an essential component for transaction ordering in AFUs requiring more complex host interfacing functions.
3. Define and plan your FPGA application.
4. Copy the `Initial_Shell_AFU` files and directory structure. This directory structure is the starting point for your design.
5. Implement your FPGA application. You can use one or a combination of the following design entry methods:
   a. RTL (System Verilog/VHDL)
   b. Platform Designer
   c. HLS

   Note: Existing design blocks can be added as required.

6. Implement host software code.
7. Simulate your design at the unit level.
8. Create timing constraints files.
9. Update the Intel Quartus Prime Settings File (`afu.qsf`) to add your new blocks.
10. Compile, synthesize, place and route your new design using provided makefile.
11. Validate timing closure.
12. Validate power consumption.
13. The provided makefile compilation script includes a post-compilation script that creates a raw binary file.
14. The raw binary file is used as an input to the Intel Acceleration Stack utility PACSign. PACSign adds a required header to the raw binary file. The output file from PACSign is validated by the N3000 Intel MAX® 10 Root of Trust for storage in the N3000 flash storage.
15. Flash the binary file produced by PACSign into FPGA flash using `fpgasupdate`.
16. Use the `rsu` utility to load the new FPGA binary file from flash into the FPGA.
17. If needed, use the Signal Tap tool to diagnose and resolve issues.

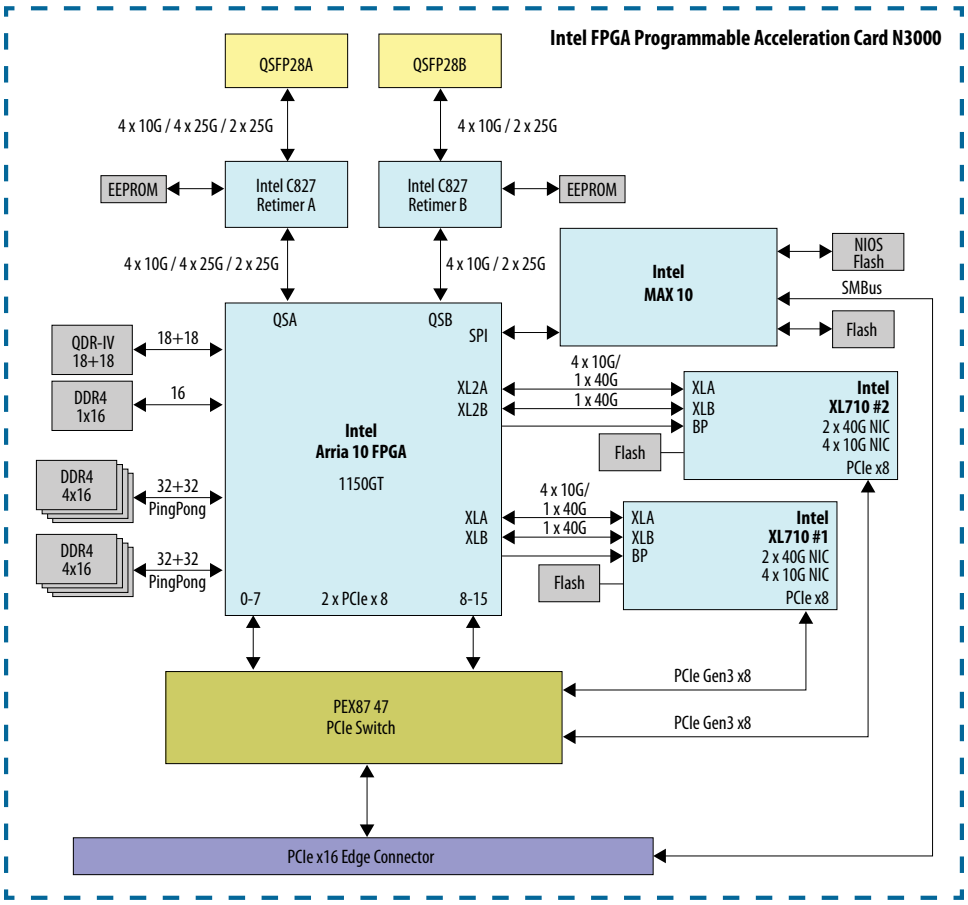
Related Information

Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual

3.2. N3000 Block Diagram

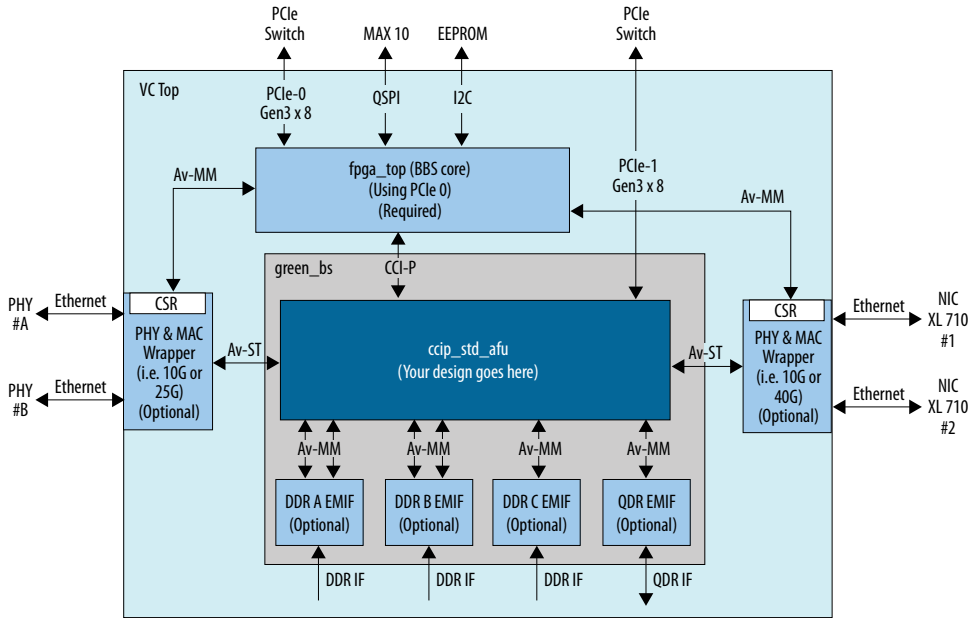
The board level N3000 block is shown below:

Figure 1. N3000 Block Diagram



As can be seen in Figure 1 on page 8, the Intel Arria 10 FPGA is central to data and control flow. Within the Intel Arria 10 FPGA, there are both data and control IP cores that are required for the board to work properly. You must include these required IP cores in your designs. Figure 2 on page 9 illustrates the Intel provided required and optional blocks as well as the `ccip_std_afu` block where your design is instantiated.

Figure 2. `ccip_std_afu` Block

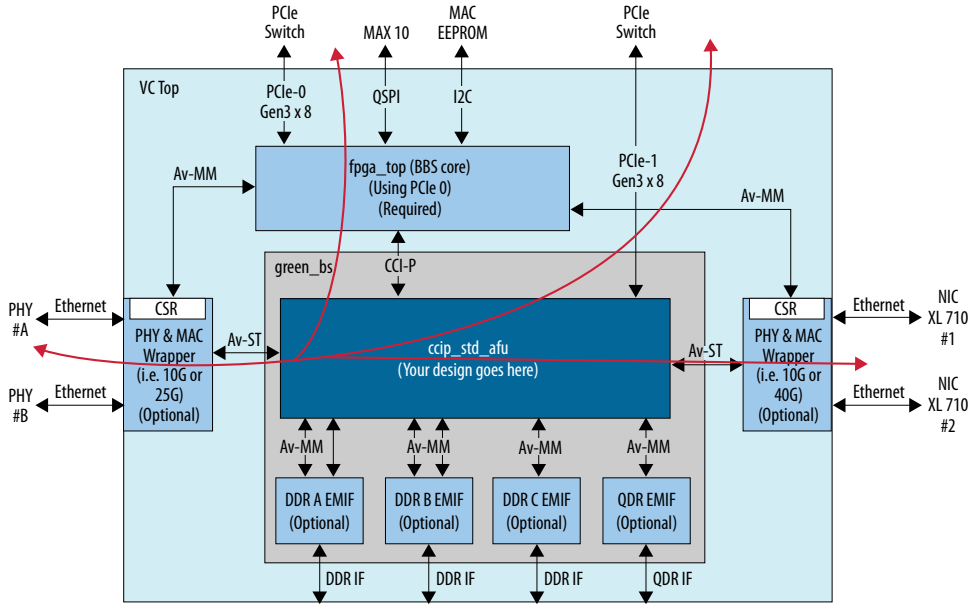


3.2.1. In-Line Data Path

The N3000 supports multiple data path options. Your application can use one or more of these data options.

Ethernet data can be processed in-line where traffic traverses: **QSFP ➤ Intel Arria 10 FPGA ➤ Intel Ethernet Controller XL710-BM2 NIC ➤ Host** and/or **QSFP ➤ Intel Arria 10 FPGA ➤ Host**. These data paths are shown below:

Figure 3. Data Path



Data can also be processed in a look-aside configuration where the data comes into the Intel Arria 10 FPGA from the host PCIe interface, the FPGA processes the data and then sends the data back to the host through the PCIe connection. Some examples of look-aside processing are compression/de-compression and encryption/decryption.

3.2.2. Supported Ethernet Network Configurations

The N3000 has three network configurations:

Network Configuration	QSFP28 A	QSFP28 B	Intel XL710 #1	Intel XL710 #2	Supported Board OPN
8 x 10GbE	4 x 10GbE	4 x 10GbE	4 x 10GbE	4 x 10GbE	BD-NFV-N3000-1
2 x 2 x 25GbE	2 x 25GbE	2 x 25GbE	2 x 40GbE	2 x 40GbE	BD-NFV-N3000-2 BD-NFV-N3000-N
4 x 25GbE	4 x 25GbE	Not Used	2 x 40GbE	2 x 40GbE	BD-NFV-N3000-2 BD-NFV-N3000-N

- 2 – QSFP ports where each QSFP supports 4 – 10 GbE lanes – this configuration is referred to as 8 X 10 G
- 2 – QSFP slots where each QSFP supports 2 – 25 GbE lanes – this configuration is referred to as 2 x 2 x 25 G
- 1 – QSFP port where 4 – 25 GbE lanes are supported. This configuration is referred to as 4 x 25 G

Note: The above network configurations are the only ones supported.

The `fpga_top` block contains a Nios II processor and firmware that configures the network settings for the Intel C827 Ethernet re-timer device. This Nios II firmware is not user editable.

The Intel Ethernet Controller XL710-BM2 network interface controller (NIC) is configured during board manufacturing to be either 10G or 40G. You cannot change the Intel Ethernet Controller XL710-BM2 NIC to switch between 10G and 40G. If your data path requires the Intel Ethernet Controller XL710-BM2 NIC, then you cannot switch between 10G and 25G network configurations. You can switch FPGA images between any of the supported network configurations.

3.2.3. Provided Files

The N3000 Acceleration Stack for Development software release provides the files for an example design. You can review this file set as a learning step for the creation of your design.

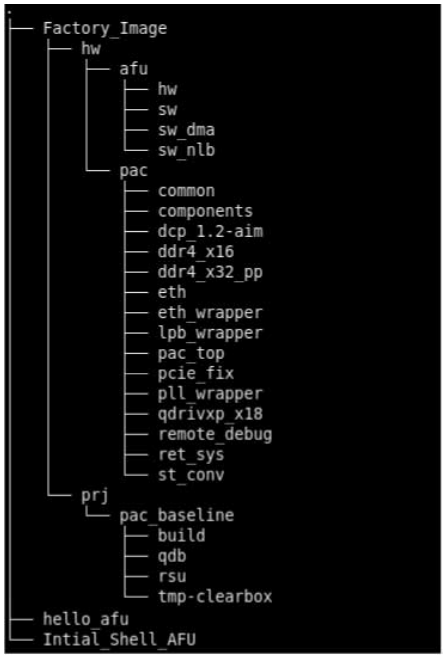
To access the files, go to your N3000 software installation directory and enter the following commands:

```
$ cd <N3000 Installation Directory>/inteldevstack/rtl/n3000_1_3_v1.5.7
$ export N3000_EXAMPLE_ROOT=$PWD
```

3.2.3.1. Directory Structure

The supplied FPGA files are a combination of clear text and encrypted files.

The directory structure of the supplied source files is shown below:



Directory Structure of the `Factory_Image` sub directory:

- `/hw/afu` – this is where the AFU factory image example is located
  - `/hw` - Sub directory with clear text RTL, `afu.qsf` and `afu.sdc`
  - `/sw` - Sub directory with example software code
- `/hw/pac` – this is where Ethernet MAC, external memory interface, and encrypted FIM is included
- `/prj/pac_baseline` – Intel Quartus Prime project files
- `/prj/pac_baseline/build` – programming files and build reports after compilation completes
- At the top of the directory tree is the Makefile used in compiling the project.
- The `hello_afu` sub directory contains a simple example AFU illustrating design points. The `Initial_Shell_AFU` contains a starting directory structure for your new AFU design.

3.2.4. Internal Interfaces

The `ccip_std_afu` module has the following interfaces:

1. Core Cache Interface (CCI-P) – This is an FPGA – Host PCIe interface required for OPAE stack operation.
2. Ethernet interface – This interface provides each Ethernet interface as individual or Multiplexed Avalon® streaming interface bus or buses.
3. Local Memory – Each external memory has an Avalon memory-mapped interface interface.
4. PCIe – Optional secondary PCIe interface can be included if needed in your AFU for additional host – FPGA data transfer capability.

3.2.4.1. Core Cache Interface (CCI-P)

The N3000 uses the CCI-P interface for compatibility with the OPAE software stack and drivers. The N3000 has the FIU capabilities of the Intel PAC with Intel Arria 10 GX FPGA as shown in the *Comparison of FIU Capabilities* section of the *Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*.

Note: You must develop a detailed understanding of the CCI-P Interface as described in the *CCI-P Interface* section of the *Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*.

The N3000 has the following signals in the CCI-P interface:

Signal	Width	Direction	Description
`pClk`	1	Input	200 MHz system clock. All CCI-P signals are synchronous to this signal.
`pClkDiv2`	1	Input	200 MHz system clock. This signal is a copy of `pClk`. Please ignore name
`pClkDiv4`	1	Input	200 MHz system clock. This signal is a copy of `pClk`. Please ignore name.
`uClk_usr`	1	Input	User clock – Default = 312.5 MHz clock. To use this clock, set USE_BBS_CLK=1 in make settings.
`uClk_usrDiv2`	1	Input	User clock – Default = 156.25 MHz clock. To use this clock, set USE_BBS_CLK=1 in make settings.
`G_CLK100`	1	Input	100 MHz global reference clock, for optional PCIe IP core or additional PLLs if needed
`t_if_ccip_Rx`	struc	Input	CCI-P data input structure defined in `ccip_if_pkg.sv`
`t_if_ccip_Tx`	struc	Output	CCI-P data output structure defined in `ccip_if_pkg.sv`
`pck_cp2af_softReset`	1	Input	Active high reset. Synchronous with `pClk` asserted for 256 clock cycles.
`pck_cp2af_pwrState`	2	Input	Present, but not used
`pck_cp2af_error`	1	Input	Present, but not used

The CCI-P clocks: `pClk`, `pClkDiv2`, `pClkDiv4`, `uClk_usr`, and `uClk_usrDiv2` do not allow you to change frequencies. If your AFU requires a different clock frequency, then instantiate a new PLL and use the `G_CLK100` as a PLL reference clock.

Related Information

Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual

3.2.4.1.1. FPGA Internal Register Access

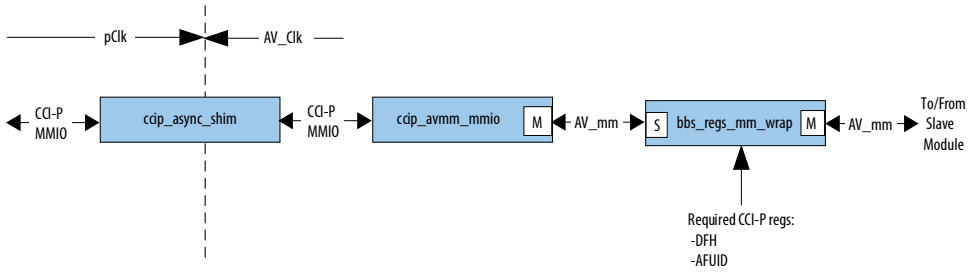
Access to internal FPGA registers with the PCIe 0 CCI-P interface uses Memory Mapped I/O (MMIO) access. You may use the following types of internal registers:

- Direct access
- Indirect access

Direct access registers consist of MMIO addressable registers. The provided example design `hello_afu.sv` illustrates direct access registers.

The CCI-P protocol MMIO address space is limited to 256 kB. The indirect access registers provide a mechanism to address larger areas by including control and response registers for extended slave addressing. AFU designers may use the provided `ccip_to_avmm` module to provide indirect access for your Avalon memory-mapped interface slave modules. The `ccip_to_avmm` module block diagram is shown below:

Figure 4. CCI-P to Avalon memory-mapped interface Block



As can be seen in this block diagram, this module consists of the following:

- `ccip_async_shim` – CCI-P to and from Avalon clock domain crossing
- `ccip_avmm_mmio` – converts MMIO to and from Avalon memory-mapped interface
- `bbs_regs_mm_wrap` – contains CCI-P required DFH and AFU ID registers and indirect command and status registers

The indirect `command` and `status` registers are defined as follows:

Table 1. Control Register

Field Name	Range	Access	Description
`cmd`	`[63:62]`	`RW`	Command for slave: 0x0 – NOP 0x1 – indirect read request 0x2 – indirect write request
`addr`	`[61:32]`	`RW`	Slave address
`Write data`	`[31:0]`	`RW`	Slave write data

For an indirect write request:

1. Write the following to the `Control` register:

- `cmd = 0x2`
- `addr`
- `Write data`

2. Poll on the `RW` valid field of the `Status` register for `RW valid = 1` to verify that the write is successful.

For an indirect read request,:

1. Write the following to the `Control` register::
   - `cmd = 0x1`
   - `addr`
2. Poll on the `RW` valid field of the `Status` register for `RW valid = 1` to verify that the `RD Data` field contains valid data.

BBS_regs_mm_wrap Access Behavior

The following figures show the `bbs_regs_mm_wrap` upstream Avalon memory-mapped interface slave to downstream Avalon memory-mapped interface slave waveforms for indirect write and read operations.

Note: Pay attention to the downstream Avalon memory-mapped interface master waveforms for proper operation with your slave module.

Indirect Write and Read Requests with Non-Blocking Access

The back pressure signal (`avmm_s_waitrequest`) is not used from the indirect access module to the CCI-P; and a write (`WR`) or read (`RD`) transaction can start at any time, but must complete in the next clock cycle.

Figure 5 on page 15 and Figure 6 on page 16 demonstrate this behavior:

Figure 5. Avalon memory-mapped interface Waveforms for Indirect Write Request with `NONBLOCKING_ACCESS_EN = 1`

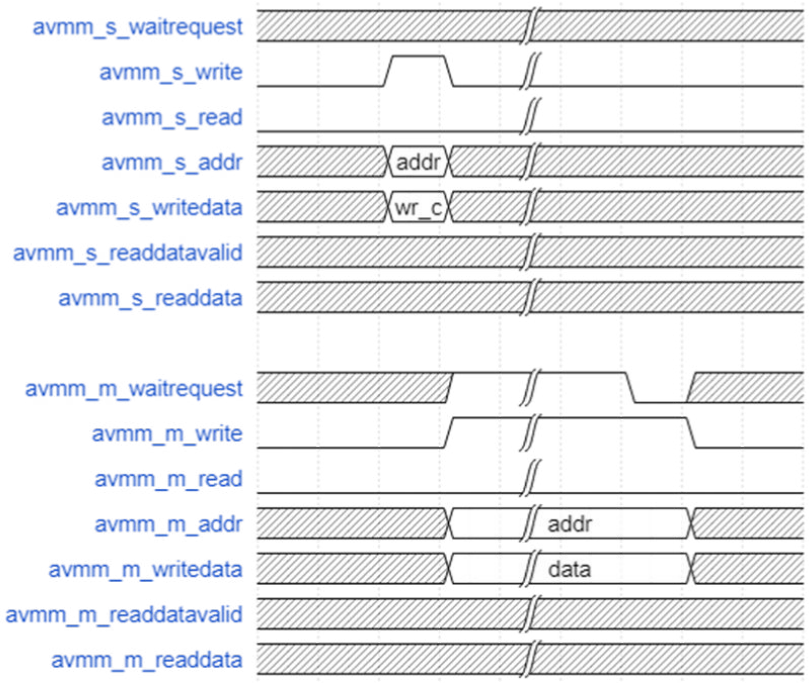
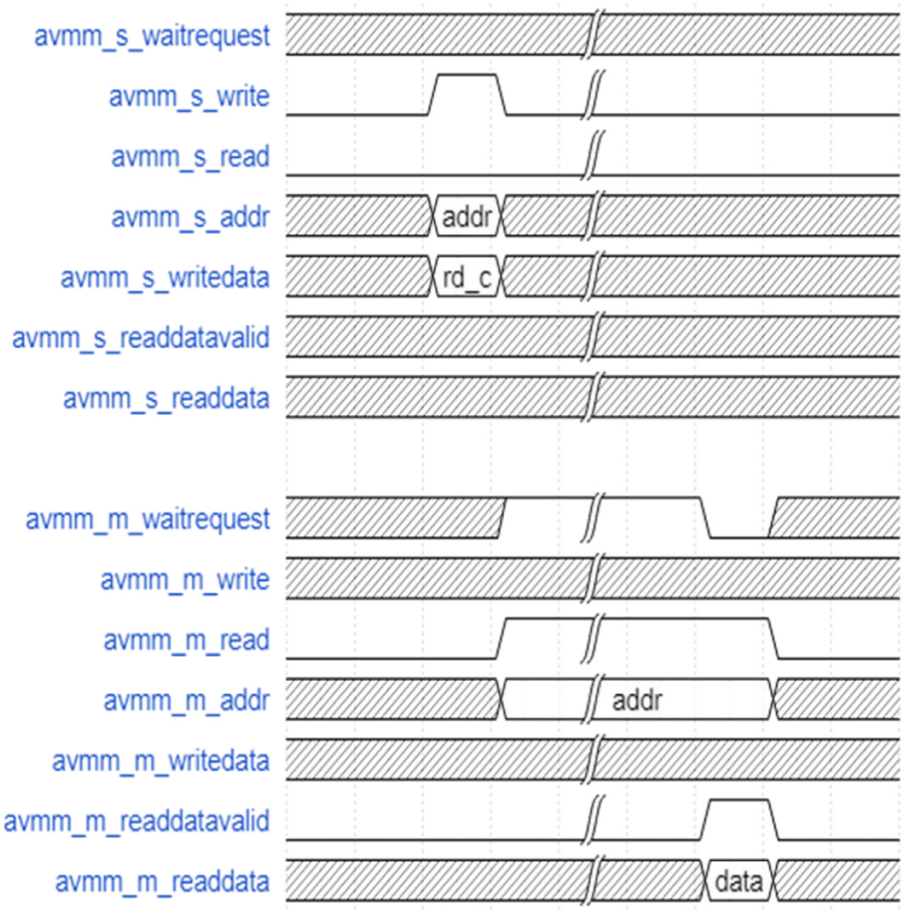


Figure 6. Avalon memory-mapped interface Waveforms of Indirect Read Request with `NONBLOCKING_ACCESS_EN = 1`



Write and Indirect Read Requests with Blocking Access

The back pressure signal (`avmm_s_waitrequest`) is always set to '1' in the NOP state; and a write (WR) or read (RD) transaction can start when `avmm_s_waitrequest = 1`, but cannot finish until `avmm_s_waitrequest != 0`.

Figure 7 on page 17 and Figure 8 on page 18 demonstrate this behavior:

Figure 7. Write Request with `NONBLOCKING_ACCESS_EN = 0`

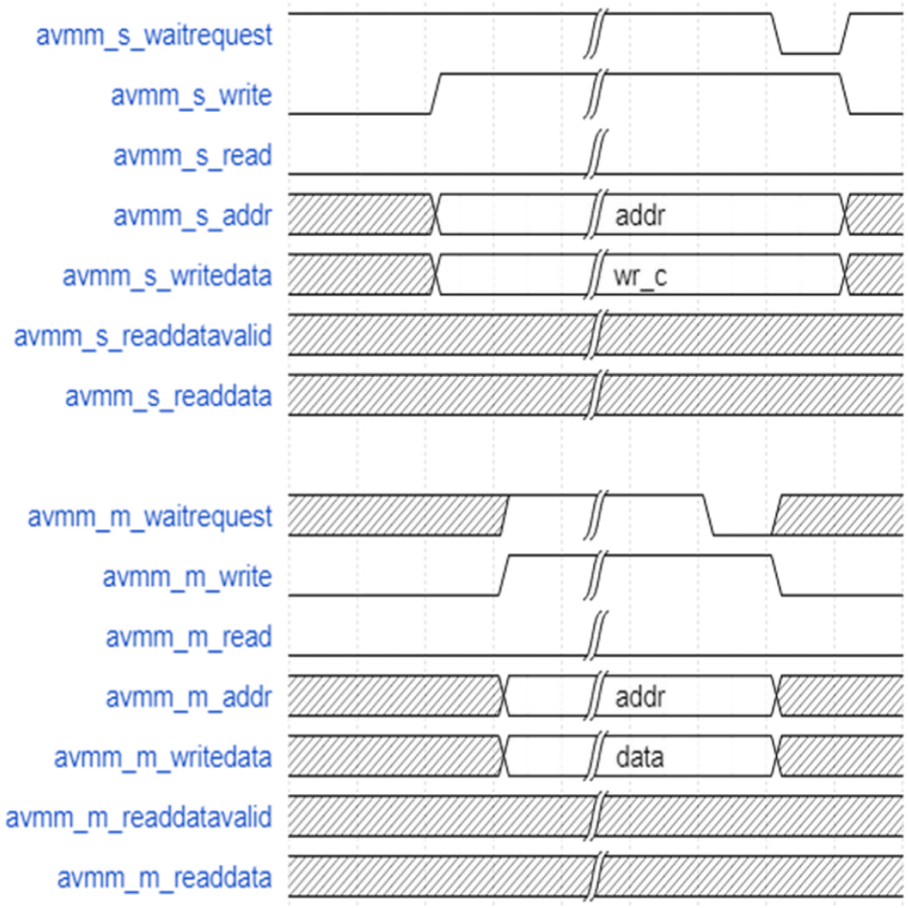
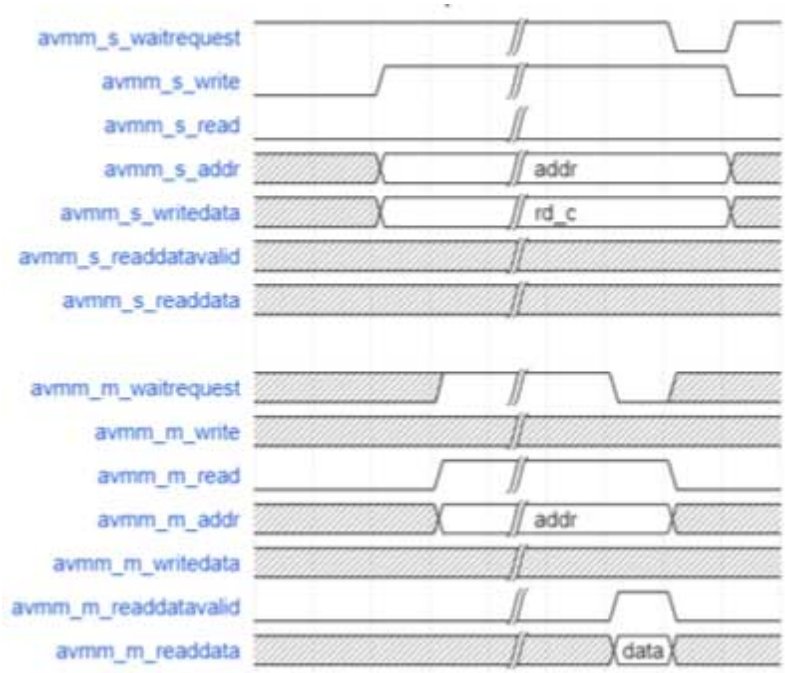


Figure 8. Waveform of Indirect Read Request with `NONBLOCKING_ACCESS_EN = 0`



Avalon memory-mapped interface Master Response to a Read Request with Non-Blocking and Blocking Access

You can read data on the bus for the following conditions:

- When `AVMM_MASTER_READDATAVALID_EN = 1` and `avmm_m_readdatavalid` are valid
- When `AVMM_MASTER_READDATAVALID_EN = 0` and `(!avmm_m_waitrequest & avmm_m_read)` are valid

Figure 9 on page 18 and Figure 10 on page 19 demonstrate this behavior:

Figure 9. Avalon memory-mapped interface Master Response to a Read Request with `NONBLOCKING_ACCESS_EN = 1`

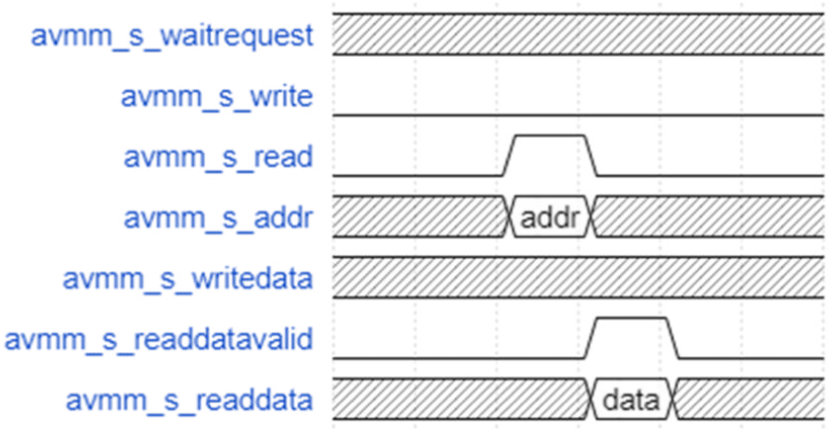


Figure 10. Avalon memory-mapped interface Master Response to a Read Request with `NONBLOCKING_ACCESS_EN = 0`

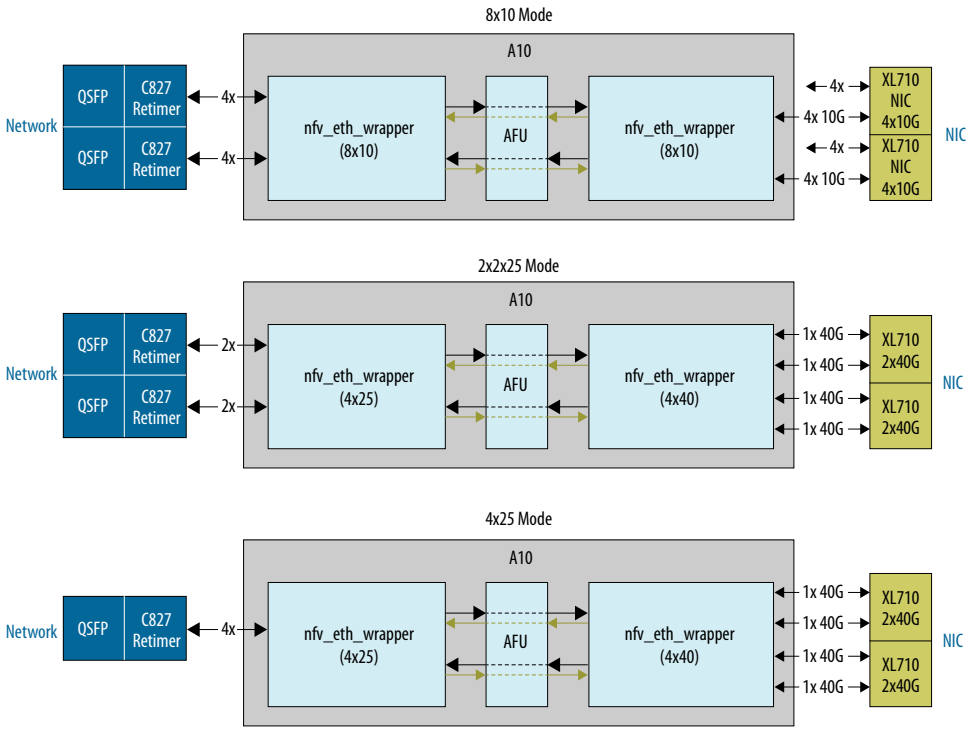
Related Information

- Ethernet MAC Wrapper Register Access on page 30
- Ethernet MAC Wrapper Register Access on page 30
- CCI-P Async Shim Basic Building Block
- CCI-P Basic Building Block Wiki

3.2.4.2. Ethernet Interface

The N3000 has Ethernet MAC IP cores to provide Ethernet receive packet delineation and transmit packet origination. The Ethernet MACs are instantiated in both the network interface and the N3000 Intel Ethernet Controller XL710-BM2 NIC interface, as shown below:

Figure 11. Instantiated Ethernet MACs



The `nfv_eth_wrapper` module is configurable by Verilog parameters for the type of Ethernet interface (10, 25 or 40 G), number of interfaces and aggregated or disaggregated style of the AFU interface. The setting of these Verilog parameters is performed by the Makefile option settings described in the *Build with make* section. The `nfv_eth_wrapper` module includes the following:

- Ethernet MAC
- PLL
- Multiplex/De-Multiplex blocks

The AFU Ethernet interface has three options with the following properties:

Aggregated:

1. One Avalon streaming interface bus aggregating all traffic from each Ethernet interface
2. Common clock
3. Each Ethernet channel is identified by Avalon streaming interface channel identifier
4. Full Ethernet MAC statistics provided

The aggregated option allows your AFU to have a common packet processing pipeline. The aggregated option uses more FPGA resources and introduces delay from a packet buffer.

Figure 12. 8x10G Multiplexor

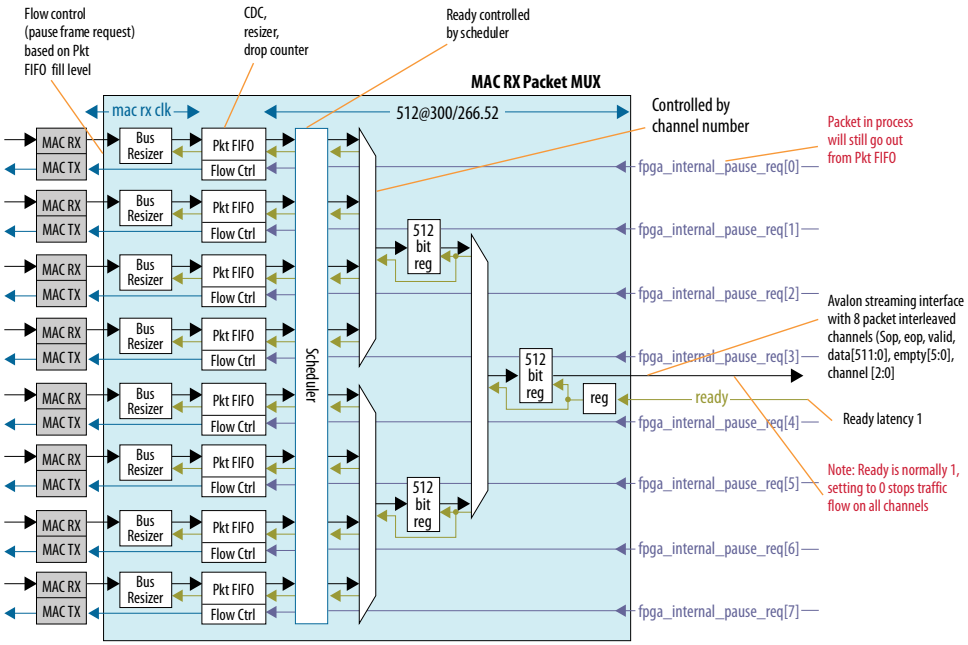
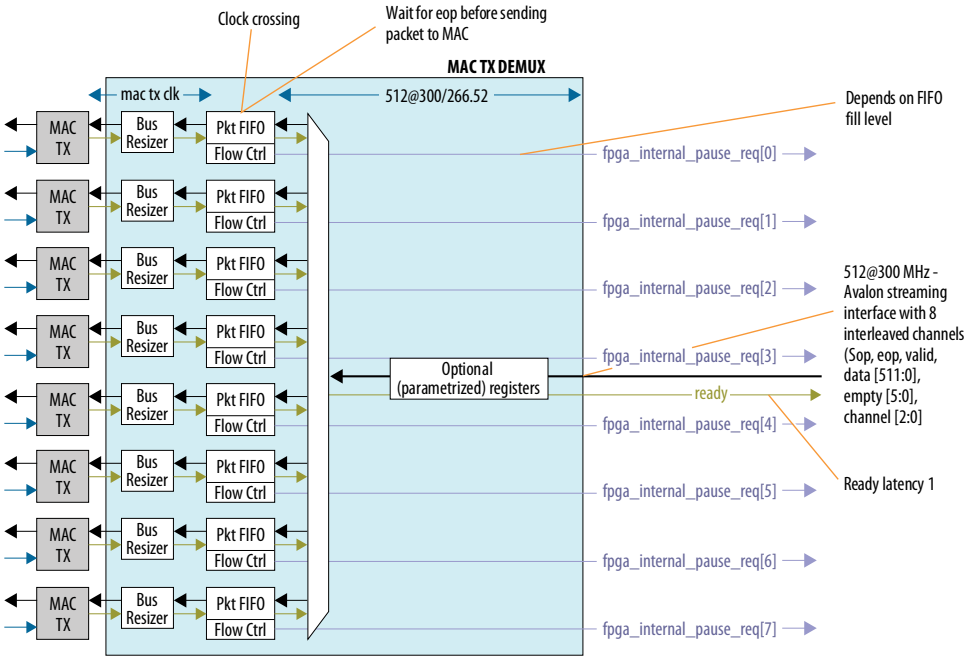


Figure 13. 8x10 De-Multiplexor



Disaggregated:

1. Each Ethernet MAC has an Avalon streaming interface bus provided as an array of Avalon streaming interface. Each channel is identified by array index.
2. Received packets with errors (CRC, length errors) are dropped from MAC.
3. Egress FIFO saturation based flow control is provided to AFU.
4. One common clock is used by AFU logic.

The disaggregated configuration reduces FPGA resources removing the multiplex/de-multiplex blocks.

Figure 14. MAC RX Packet MUX

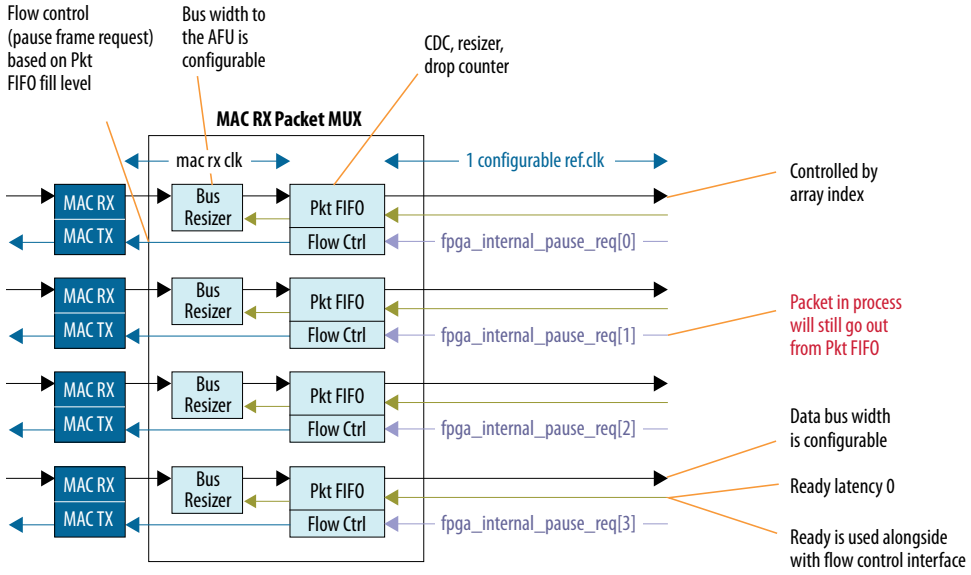
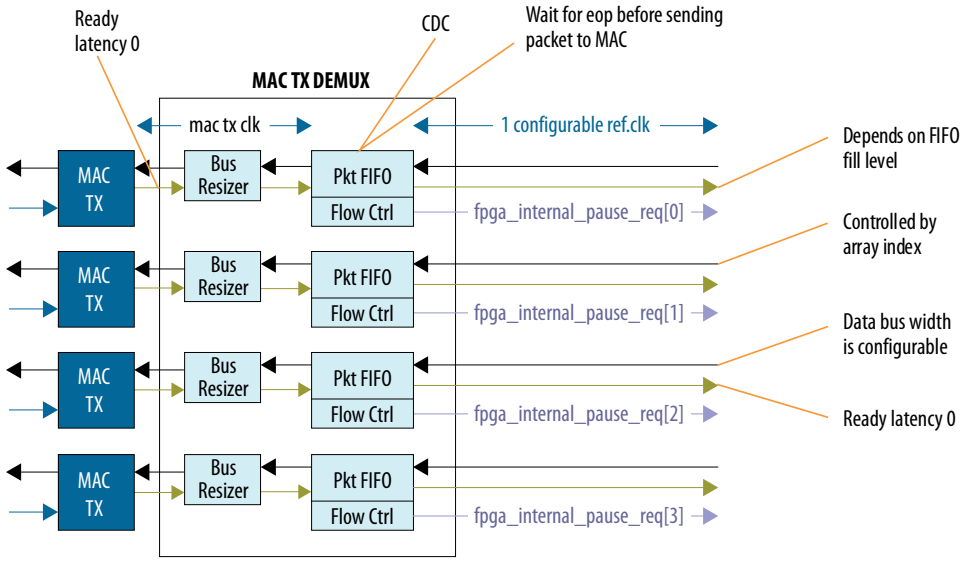


Figure 15. MAC TX DEMUX



Lightweight Mode

1. Disaggregated Avalon streaming interface interfaces from each MAC.
2. MUX function passes received traffic directly to the AFU.
3. The AFU must control the data stream by removing frames with errors and controlling the flow.
4. Each MAC interface has a separate clock.
5. No Ethernet statistics provided in Ethernet MACs.

Note: The Lightweight mode is not supported for 10G applications.

Figure 16. MAC RX Packet MUX

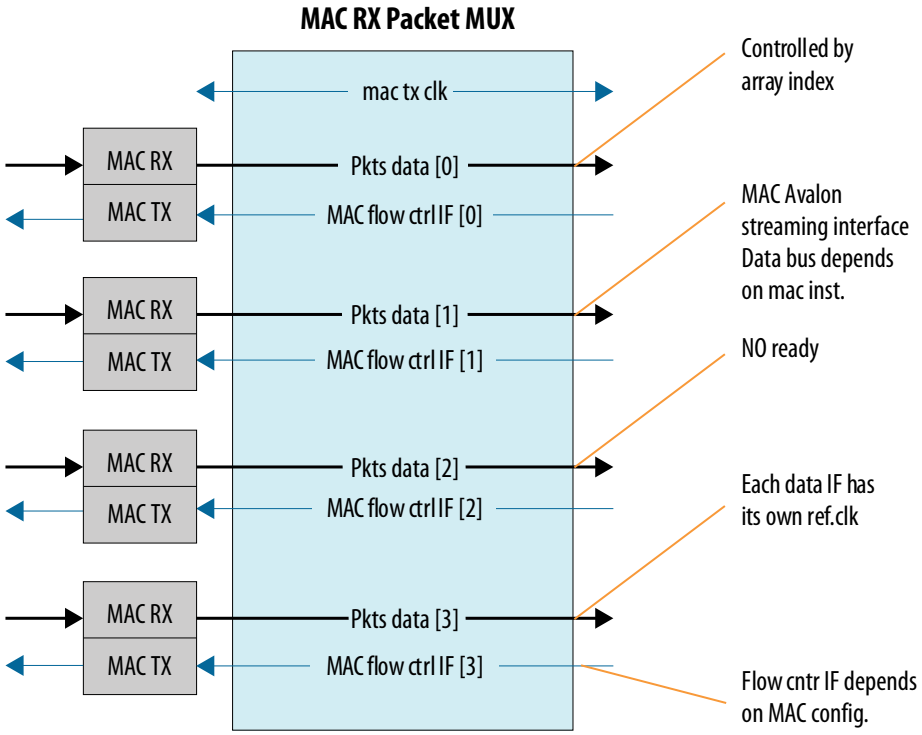
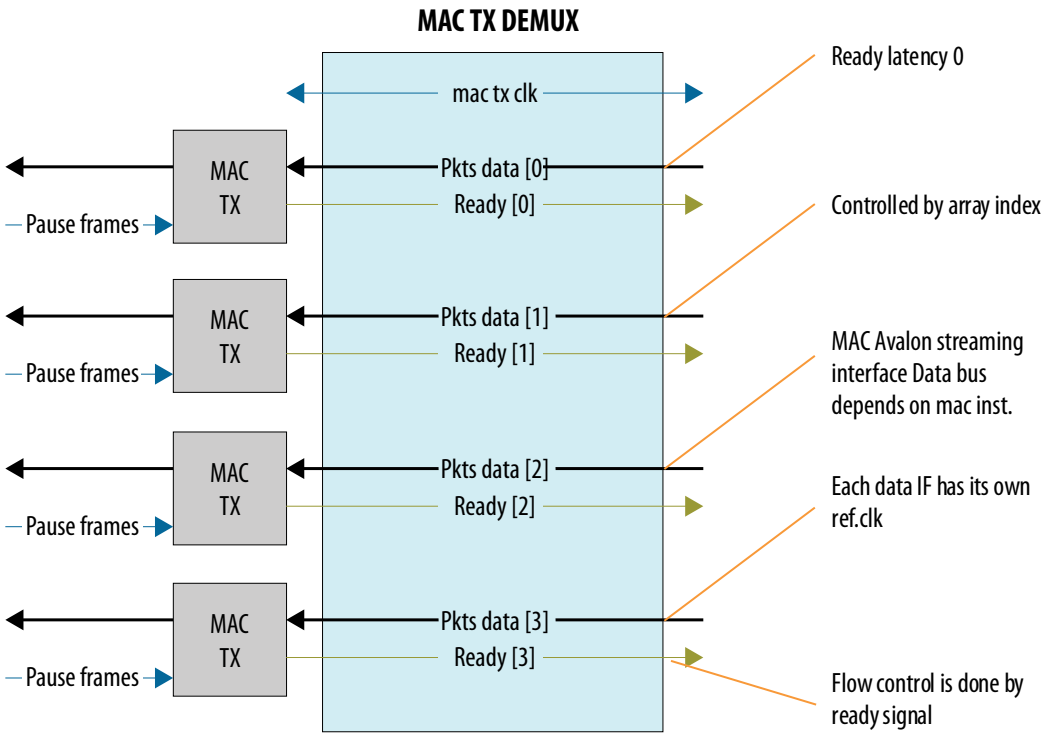


Figure 17. MAC TX DEMUX



Related Information

Build with make on page 37

3.2.4.3. Ethernet MAC

The 25 GbE MAC IP core is documented in the 25G Ethernet Intel Arria 10 FPGA IP User Guide. The N3000 configures the 25 GbE MACwith the following parameters set:

Table 2. 25G MAC IP Setting

Parameter	IP Core parameter setting
Ready Latency	0
Enable RS-FEC	Off
Enable flow control	On
Enable link fault generation	On
Enable preamble pass through	Off
Enable TX CRC pass through	Off
Enable MAC statistics counters	On Off for Light weight mode
Enable IEEE 1588	Off

The Intel C827 Re-timer performs FEC functionality, therefore the A10 Ethernet MAC does not have RS-FEC enabled.

The 10 GbE MAC IP core is documented in: Low Latency Ethernet 10G MAC Intel Arria 10 FPGA IP Design Example User Guide

The 40 GbE MAC IP core is documented in: Low Latency 40-Gbps Ethernet IP Core User Guide

The 40 and 10 GbE MAC IP core are set with the following parameters:

Table 3. 40G MAC IP Setting

Parameter	IP core parameter setting
Enable SyncE	Off
PHY reference	644.53125MHz
Use external TX MAC PLL	On
Flow control mode	Standard flow control
Average inter-packet gap	12
Enable 1588 PTP	Off
Enable link fault generation	On
Enable TX CRC insertion	On
Enable preamble pass through	Off
Enable alignment EOP on FCS word	On
Enable TX statistics	On
Enable RX statistics	On
Enable strict SFD checking	Off

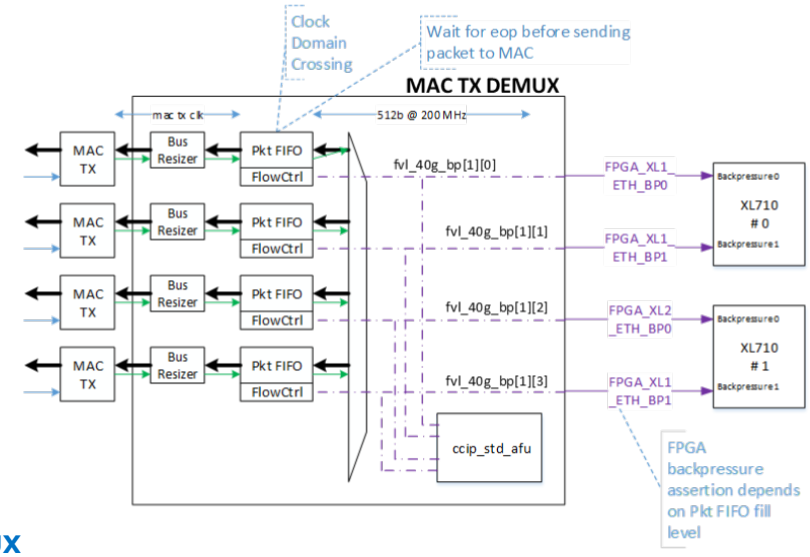
3.2.4.4. 40G – 25G Gearbox

For 25 GbE operation, the Intel Arria 10 FPGA provides a gearbox that rate adjusts between the 25 GbE network interface and the Intel Ethernet Controller XL710-BM2 NIC 40 GbE interface.

Received 25 GbE traffic is written into a per port 32 kB Intel Arria 10 FPGA FIFO. The FIFO data is read out on packet boundaries using a 40 GbE rate where an entire packet is transferred to the Intel Ethernet Controller XL710-BM2 NIC. The Intel Arria 10 FPGA extends the interframe packet gap to the Intel Ethernet Controller XL710-BM2 NIC such that the data rate is 40 Gb, however the number of packets transferred is determined by the number of packets received from the 25 GbE network port.

The Intel Ethernet Controller XL710-BM2 NIC sends Ethernet traffic to the Intel Arria 10 FPGA over a 40 GbE path. The Intel Arria 10 FPGA buffers the 40 GbE traffic in a 32 kB packet-based FIFO. If the Intel Arria 10 FPGA FIFO exceeds half fill level, then the Intel Arria 10 FPGA asserts a backpressure external pin, signaling the Intel Ethernet Controller XL710-BM2 NIC to extend the interframe packet gap. Once the Intel Arria 10 FPGA FIFO capacity drops to a quarter of capacity, then the backpressure external pin is de-asserted. This extended interframe packet gap reduces the packet rate such that the resulting data rate is 25 Gb. The backpressure signals are connected to the `ccip_std_afu` module. The Intel Ethernet Controller XL710-BM2 NIC to Intel Arria 10 FPGA data flow is shown in this figure:

Figure 18.

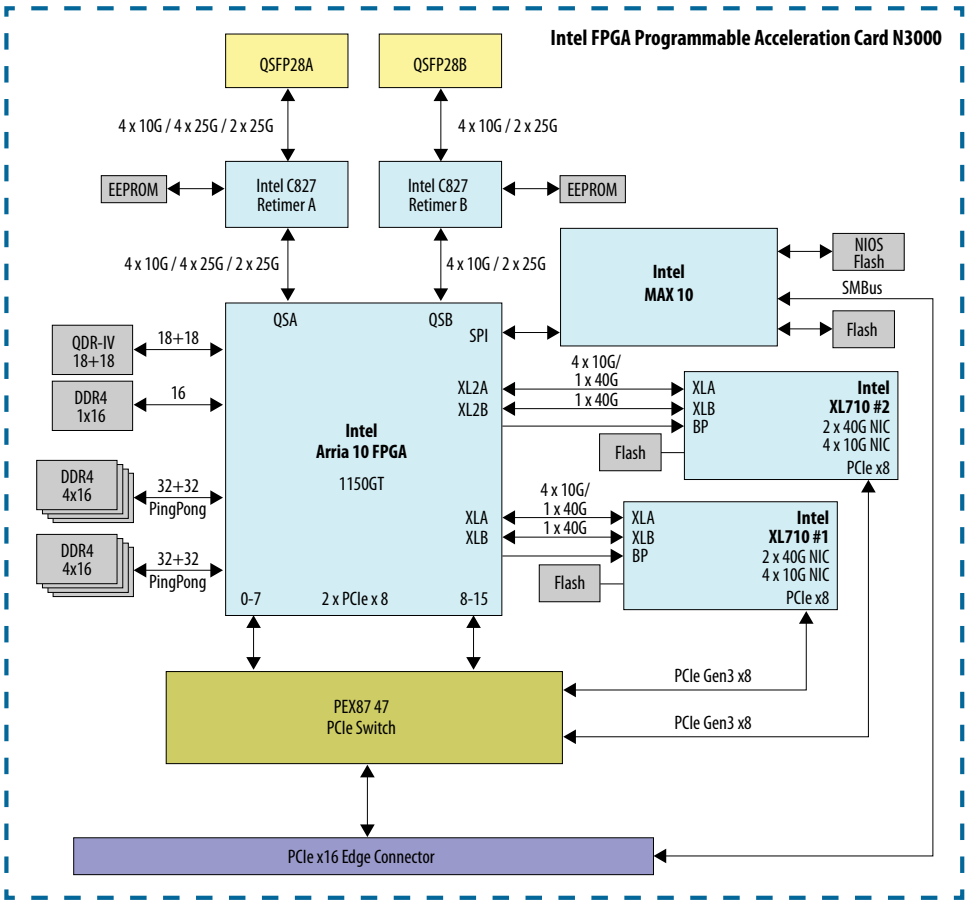


MAC TX DEMUX

3.2.4.5. External Memory Interfaces

The N3000 has the following external memory interfaces as shown in the board block diagram below:

Figure 19. External Memory Interfaces



- DDR4 – 2133 Mb/s – total 9 GB
  - — DDR4A and DDR4B - each 4 GB banks
    - 64-bit wide
    - Ping-Pong physical interface
  - — DDR4C - 1 GB bank
    - 16-bit wide
- QDR4 – 1066 MHz – 144 Mb
  - — 8M x 18

Related Information

External Memory Interfaces Intel Arria 10 FPGA IP User Guide

3.2.4.5.1. DDR4A and DDR4B

Both DDR4A and DDR4B use the Ping Pong PHY described in the *Intel Arria 10 EMIF Ping Pong PHY Description* section of the *External Memory Interfaces Intel Arria 10 FPGA IP User Guide*.

The Ping Pong PHY is physically implemented in the board design. The Ping Pong PHY design has two independent memory controllers per DDR4 interface where your interface consists of two Avalon memory-mapped interface interfaces. See DDR4A user interface below (please note, DDR4B is identical).

ccip_std_afu Direction	Width	Signal Name	Description
DDR4A_0 Interface			
input		ddr4a_avmm_0_clk	266 MHz clock sourced from EMIF
input		ddr4a_avmm_0_reset_n	Active low reset to user logic. Reset for the user clock domain. Asynchronous assertion and synchronous de-assertion
input		ddr4a_avmm_0_waitrequest	Wait-request is asserted when controller Avalon memory-mapped interface interface is busy
input	[255:0]	ddr4a_avmm_0_readdata	Read data from external memory
input		ddr4a_avmm_0_readdatavalid	Indicates readdata is valid when high
output	[6:0]	ddr4a_avmm_0_burstcount	Number of transfers in each read/write burst
output	[255:0]	ddr4a_avmm_0_writedata	AFU supplied data to written to external memory
output	[25:0]	ddr4a_avmm_0_address	Word address forAvalon memory-mapped interface interface of memory controller
output		ddr4a_avmm_0_write	Write request from AFU
output		ddr4a_avmm_0_read	Read request from AFU
output	[31:0]	ddr4a_avmm_0_byteenable	Write byte enable from AFU
DDR4A_1 Interface			
input		ddr4a_avmm_1_clk	Copy of ddr4a_avmm_0_clk
input		ddr4a_avmm_1_reset_n	Secondary active low reset to user logic. Reset for the user clock domain. Asynchronous assertion and synchronous de-assertion
input		ddr4a_avmm_1_waitrequest	Wait-request is asserted when controller Avalon memory-mapped interface interface is busy
input	[255:0]	ddr4a_avmm_1_readdata	Read data from external memory
input		ddr4a_avmm_1_readdatavalid	Indicates readdata is valid when high
output	[6:0]	ddr4a_avmm_1_burstcount	Number of transfers in each read/write burst
output	[255:0]	ddr4a_avmm_1_writedata	AFU supplied data to written to external memory
output	[25:0]	ddr4a_avmm_1_address	Word address for Avalon memory-mapped interface interface of memory controller
output		ddr4a_avmm_1_write	Write request from AFU
output		ddr4a_avmm_1_read	Read request from AFU
output	[31:0]	ddr4a_avmm_1_byteenable	Write byte enable from AFU

You can combine both of the Ping Pong Avalon memory-mapped interface interfaces from one DDR4 bank to form a 512-bit interface with an Avalon combiner. The factory image example demonstrates the use of the Avalon combiner.

The DDR4A and DDR4B interfaces are suited to large record storage, off chip deep packet queues and other storage needs.

3.2.4.5.2. DDR4C

The `ccip_std_afu` interfaces to DDR4C by an Avalon memory-mapped interface interface as defined below:

ccip_std_afu Direction	Width	Signal Name	Description
input		`ddr4c_avmm_0_clk`	266 MHz clock sourced from EMIF
input		`ddr4c_avmm_0_reset_n`	Active low reset to user logic. Reset for the user clock domain. Asynchronous assertion and synchronous de-assertion
input		`ddr4c_avmm_0_waitrequest`	Wait-request is asserted when controller Avalon memory-mapped interface interface is busy
input	[127:0]	`ddr4c_avmm_0_readdata`	Read data from external memory
input		`ddr4c_avmm_0_readdatavalid`	Indicates readdata is valid when high
output	[6:0]	`ddr4c_avmm_0_burstcount`	Number of transfers in each read/write burst
output	[127:0]	`ddr4c_avmm_0_writedata`	AFU supplied data to written to external memory
output	[25:0]	`ddr4c_avmm_0_address`	Word address for Avalon memory-mapped interface interface of memory controller
output		`ddr4c_avmm_0_write`	Write request from AFU
output		`ddr4c_avmm_0_read`	Read request from AFU
output	[15:0]	`ddr4c_avmm_0_byteenable`	Write byte enable from AFU

3.2.4.5.3. QDR4 Interface

The external QDR4 SRAM is well suited for fast table look ups and external statistics counter storage due to the fast random access capabilities of QDR4 SRAM. QDR4 SRAM transfers 4 data words per clock cycle. QDR4 SRAM also has two independent bidirectional double data rate ports that support concurrent read/write transactions on both ports.

The multiple access ports of the QDR4 SRAM results in the internal interface providing 8 – Avalon memory-mapped interface interfaces for this external memory device. The interface is shown below:

ccip_std_afu Direction	Width	Signal Name	Description
input		`qdr_avmm_clk`	266 MHz clock sourced from EMIF
input		`qdr_avmm_reset_n`	Active low reset to user logic. Reset for the user clock domain. Asynchronous assertion and synchronous de-assertion
input		`qdr_avmm_waitrequest [7:0]`	Wait-request is asserted when controller Avalon memory-mapped interface interface is busy
input	[35:0]	`qdr_avmm_readdata [7:0]`	Read data from external memory
input		`qdr_avmm_readdatavalid [7:0]`	Indicates readdata is valid when high

continued...

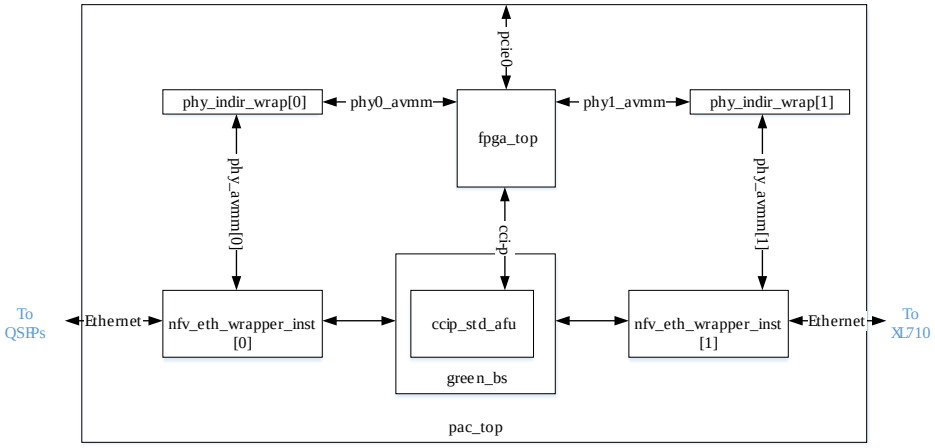
ccip_std_afu Direction	Width	Signal Name	Description
output	[2:0]	qdr_avmm_burstcount [7:0]	Number of transfers in each read/write burst
output	[35:0]	qdr_avmm_writedata [7:0]	AFU supplied data to written to external memory
output	[21:0]	qdr_avmm_address [7:0]	Word address for Avalon memory-mapped interface interface of memory controller
output		qdr_avmm_write [7:0]	Write request from AFU
output		qdr_avmm_read [7:0]	Read request from AFU

3.2.4.6. Ethernet MAC Wrapper Register Access

Host processor access to Ethernet MAC Wrapper is by the CCI-P interface using MMIO indirect access as described in the *FPGA Internal Register Access* section. The RTL modules for register access are included in the encrypted portion of N3000 design and these modules must be included in your design.

There are two Ethernet MAC Wrappers where one wrapper is connected to the network and the other is connected to the Intel Ethernet Controller XL710-BM2 NIC, as shown below:

Figure 20. Ethernet Wrapper Register Access



The Ethernet MAC Wrapper registers consist of the following:

- CCI-P required Device Feature Header (DFH) and Information registers are located inside `fpga_top`.
- Indirect access control register and status registers are located in `phy_indir_wrap`.
- Ethernet MAC, PHY and multiplex/de-multiplex control and status registers are located in `nfv_eth_wrapper`.

For an example of how software accesses the Ethernet MAC wrapper, see the python source file included with the OPAE software release installation:

```
inteldevstack/src/opae-*.*/usr/tools/extra/fpgadiag/fpgastats.py
```

The following description of registers below is provided for informational purposes. Do not change or modify this area code, but understanding how this works helps you create your AFU. When the lightweight mode is used, the Ethernet MAC registers are not included.

These registers are organized as follows:

Table 4. Ethernet MAC Registers

Register	Address Offset
ETH_GROUP_0_DFH	0x7000
ETH_GROUP_0_INFO	0x7008
ETH_GROUP_0_CTRL	0x7010
ETH_GROUP_0_STAT	0x7018
ETH_GROUP_1_DFH	0x8000
ETH_GROUP_1_INFO	0x8008
ETH_GROUP_1_CTRL	0x8010
ETH_GROUP_1_STAT	0x8018

The Information register consists of the following fields:

Table 5. Information Register Fields

FIELD NAME	RANGE	ACCESS	DEFAULT	DESCRIPTION
Reserved	[63:26]	RsvdZ	0x0	Reserved
MAC light weight mode	[25]	RO	0x0	0 - MACs are in normal mode 1 - MACs are in light weight mode
Direction	24	RO	0x0	0 – XL710 side 1 - Network side
Speed_Gbs	[23:16]	RO	0xA	Allowed: 10, 25, 40 Gbs.
NofPHYs	[15:8]	RO	0x8	Number of PHYs in group
GroupID	[7:0]	RO	0x0	Unique identifier of phy group. ETH_GROUP_0 = 0, ETH_GROUP_1 = 1

The indirect control field has one version for 10G mode and a second version for 25G and 40G mode. The 10G mode is shown below:

Table 6. 10G Indirect Control Field

FIELD NAME	RANGE	ACCESS	DEFAULT	DESCRIPTION
command	[63:62]	RW	0x0	Command: 0x0 - NOP 0x1 - RD request 0x2 - WR request
reserved	[61:54]	RO	0x0	
device select	[53:49]	RW	0x0	0x0 - Ethernet Wrapper regs select 0x2, 0x4, 0x6, 0x8, 0xA, 0xC, 0xE, 0x10 - PHY select 0x3, 0x5, 0x7, 0x9, 0xB, 0xD, 0xF, 0x11 - MAC select

continued...

PHY select				device select = 0x2, 0x4, 0x6, 0x8, 0xA, 0xC, 0xE, 0x10
add features select	[48]	RW	0x0	0x0 - phy select 0x1 - reset controller / link status select
PHY Address/reset ctrl/link status	[47:32]	RW	0x0	add features select = 0x0: PHY reconfiguration interface www.altera.com/literature/hb/arria-10/ug_arria10_xcvr_phy.pdf add features select = 0x1: ref. to add features tab.
MAC register address	[48:32]	RW	0x0	When device select = 0x3, 0x5, 0x7, 0x9, 0xB, 0xD, 0xF, 0x11 This field is for Ethernet MAC IP registers as defined in: www.altera.com/en_US/pdfs/literature/ug/ug_32b_10g_ethernet_mac.pdf.
ethernet wrapper regs address	[48:32]			When device select = 0x0 This field includes Ethernet Mux/De-Mux registers
write data	[31:0]	RW	0x0	Write data for phy registers

For 25G and 40G mode:

Table 7. 25G and 40G Indirect Control Field

FIELD NAME	RANGE	ACCESS	DEFAULT	DESCRIPTION
command	[63:62]	RW	0x0	Command: 0x0 - NOP 0x1 - RD request 0x2 - WR request
reserved	[61:54]	RO	0x0	
device select	[53:49]	RW	0x0	0x0 - ethernet wrapper regs select 0x2, 0x4, 0x6, 0x8 - PHY select 0x3, 0x5, 0x7, 0x9 - MAC select
PHY select				device select = 0x2, 0x4, 0x6, 0x8
add features select	[48]	RW	0x0	0x0 - phy select 0x1 - reset controller / link status select
PHY Address/reset ctrl/link status	[47:32]	RW	0x0	add features select = 0x0: add features select = 0x1: ref. to add features tab.
MAC select				device select = 0x3, 0x5, 0x7, 0x9
	[48:32]			
Ethernet Wrapper regs address	[48:32]			When device select = 0x0 This field includes Ethernet Mux/De-Mux registers

Related Information

- FPGA Internal Register Access on page 13
- FPGA Internal Register Access on page 13

3.3. Factory Image Description

The N3000 provides an example design that demonstrates usage of the key interfaces available to the `ccip_std_afu` module.

The block diagram of each network configuration is shown below:

Figure 21. Factory Image Block Diagram for 8x10 GbE

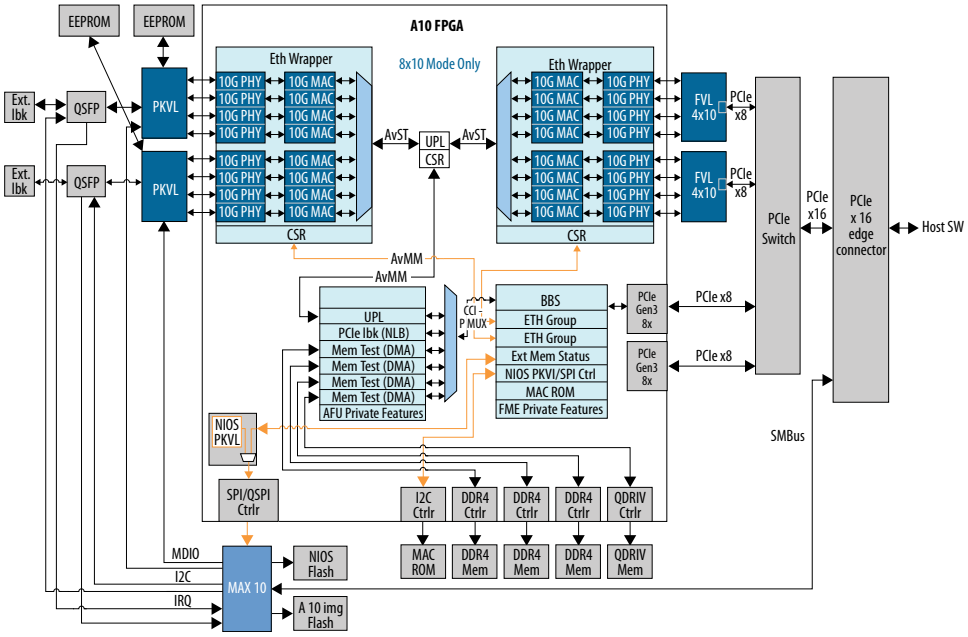


Figure 22. Factory Image Block Diagram for 2x2x25GbE

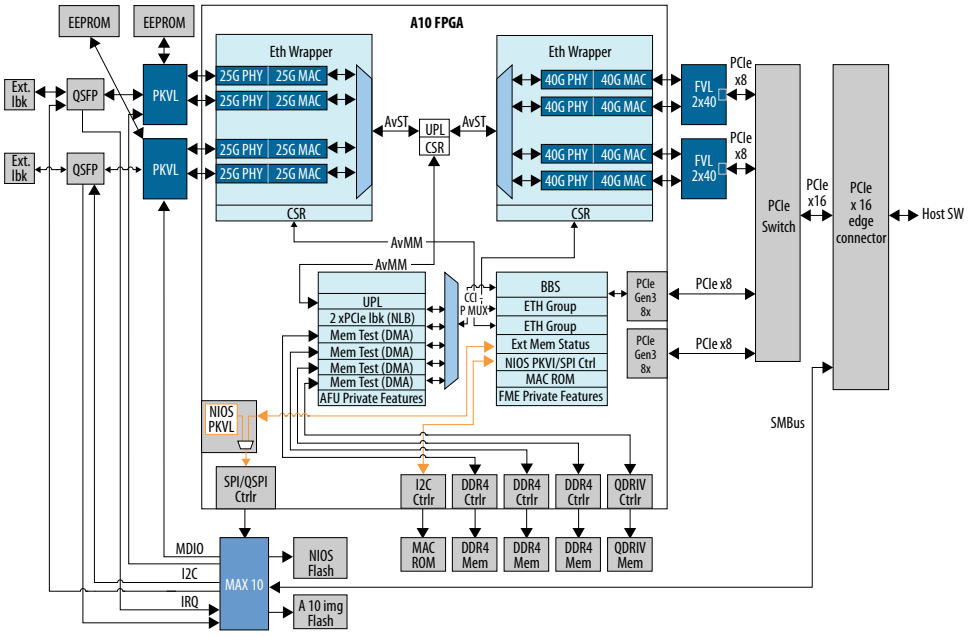
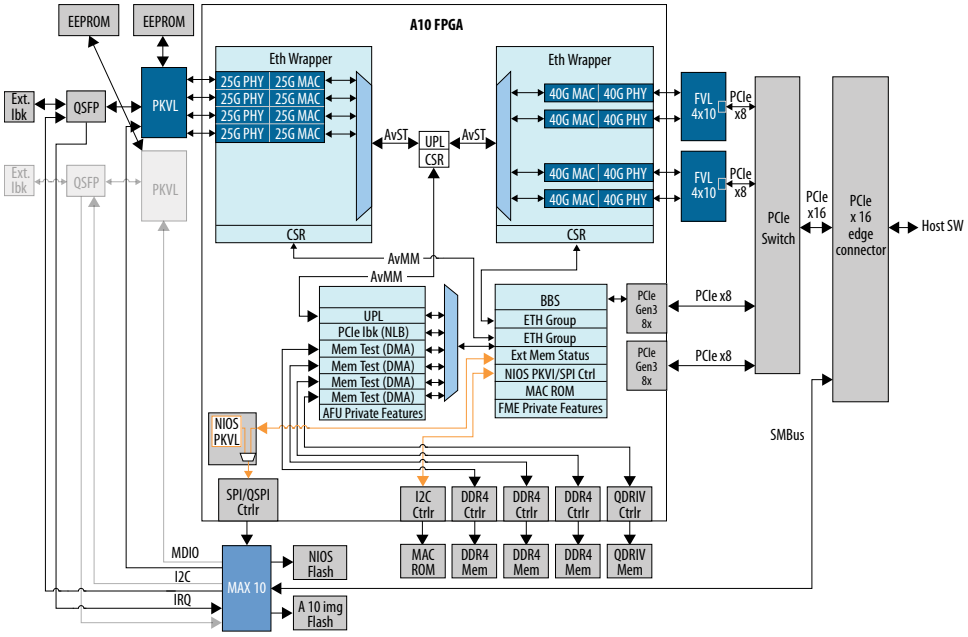


Figure 23. Factory Image Block Diagram for 4x25 GbE



The Factory Images include the following high level functions:

- Memory-to-memory DMA blocks illustrating host to and from external memory transfers. For more information about this component, refer to the *DMA Accelerator Functional Unit (AFU) User Guide: Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA*.
- Native Loopback to test memory reads and writes, bandwidth, and latency. For more information, refer to the *Native Loopback Accelerator Functional Unit (AFU) User Guide*.
- Aggregated Ethernet interface
- Required board management functions

Related Information

- DMA Accelerator Functional Unit (AFU) User Guide: Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA
- Native Loopback Accelerator Functional Unit (AFU) User Guide

4. Creating an N3000 FPGA Design

In this section, steps are provided to create your AFU. The following AFU design directories are included:

- `hello_afu` – this is a simple AFU design illustrating basic design concepts
- `Factory_Image` – this is a complex AFU design illustrating usage of Ethernet and external memories
- `Initial_Shell_AFU` – this design directory serves as the starting point for your AFU. The required project files are included.

In this section, you are referred to these design directories as a way to highlight points in the AFU creation, compilation and running of an application on the N3000. Use these points to create more complex AFU designs for your specific application.

4.1. Create New Project Directory

The `Initial_Shell_`*AFU* provides the starting structure for your *AFU* design.

Create a new project directory and copy the `Initial_Shell_`*AFU* files to the new project directory.

```
$ mkdir <Your new project directory name>
$ cd <Your new project directory name>
$ cp -R $N3000_EXAMPLE_ROOT/Initial_Shell_AFU/* .
```

Your design directory is now ready for your new design work.

4.2. Create Your AFU Design Files

As a minimum, create the following files for your AFU design:

1. `ccip_std_afu.sv` – this file is where your AFU connects to CCI-P fabric, external memory and Ethernet
2. An AFU file. You can see AFU examples in `hello_afu/hw/afu/hw/rtl/hello_afu.sv` and `Factory_Image/hw/afu/rtl/afu`
3. `afu.qsf` - this Intel Quartus Prime file adds your RTL and design files
4. `afu.sdc` – this Intel Quartus Prime file specifies your AFU timing constraints

4.2.1. ccip_std_afu.sv

The `Initial_Shell_AFU` includes `ccip_std_afu.sv` starting file where you can instantiate your AFU. Review of this file shows `import ccip_if_pkg::*` to include definitions of CCI-P interface structures.

```
import ccip_if_pkg::*;  //required for CCI-P definitions
module ccip_std_afu #( …..
;
;
```

There is a CCI-P interface register to improve timing as shown below:

```
    // ========================================================
    // Register SR <--> PR signals at interface before consuming it
    // ========================================================

    (* noprune *) logic [1:0]   pck_cp2af_pwrState_T1;
    (* noprune *) logic         pck_cp2af_error_T1;

    logic                       pck_cp2af_softReset_T1;
    t_if_ccip_Rx                pck_cp2af_sRx_T1;
    t_if_ccip_Tx                pck_af2cp_sTx_T0;

    // ========================================================
    // Register PR <--> PR signals near interface before consuming it
    // ========================================================

    ccip_interface_reg inst_green_ccip_interface_reg  (
        .pClk                           (pClk),
        .pck_cp2af_softReset_T0         (pck_cp2af_softReset),
        .pck_cp2af_pwrState_T0          (pck_cp2af_pwrState),
        .pck_cp2af_error_T0             (pck_cp2af_error),
        .pck_cp2af_sRx_T0               (pck_cp2af_sRx),
        .pck_af2cp_sTx_T0               (pck_af2cp_sTx_T0),

        .pck_cp2af_softReset_T1         (pck_cp2af_softReset_T1),
        .pck_cp2af_pwrState_T1          (pck_cp2af_pwrState_T1),
        .pck_cp2af_error_T1             (pck_cp2af_error_T1),
        .pck_cp2af_sRx_T1               (pck_cp2af_sRx_T1),
        .pck_af2cp_sTx_T1               (pck_af2cp_sTx)
    );
```

Your AFU design connects to this registered CCI-P interface.

4.2.2. AFU File

Your AFU requires a CCI-P package and a UUID for proper connectivity with host software. See example below:

```
import ccip_if_pkg::*;
module hello_afu
   (
    input  clk,    // Core clock. CCI interface is synchronous to this clock.
    input  reset,  // CCI interface ACTIVE HIGH reset.
    // CCI-P signals
    input  t_if_ccip_Rx cp2af_sRxPort,
    output t_if_ccip_Tx af2cp_sTxPort
    );
    `define AFU_ACCEL_UUID 128'h850adcc2_6ceb_4b22_9722_d43375b61c66
    // The AFU must respond with its AFU ID in response to MMIO reads of
    // the CCI-P device feature header (DFH).  The AFU ID is a unique ID
    // for a given program.  Here we generated one with the "uuidgen"
    // program and stored it in the AFU's JSON file.  ASE and synthesis
```

```
    // setup scripts automatically invoke the OPAE afu_json_mgr script
    // to extract the UUID into afu_json_info.vh.
    logic [127:0] afu_id = `AFU_ACCEL_UUID;
```

Note: For a more complicated example where multiple sub-AFUs are instantiated, refer to `Factory_Image/hw/afu/rtl/afu_dma.sv`.

The software framework and the application software use the `AFU_ID` to ensure that they are matched to the correct AFU; that is, that they are obeying the same architectural interface.

The `AFU_ID` is a 128-bit value which is generated using an UUID/GUID generator to ensure the value is unique.

For more information about UUID/GUID, refer to the "Online GUID Generator" web page.

Related Information

Online GUID Generator

4.2.3. QSF File

The `afu.qsf` is where you include your AFU RTL and any other required implementation files. The `Intial_Shell_AFU` includes an `afu.qsf` file where you can add your specific files.

4.2.4. SDC File

The `afu.sdc` is where you include your AFU timing constraints files.

4.3. Build with `make`

The process of creating an N3000 FPGA image is simplified with the provided Makefile that automates the setting of compile parameters and combining your design files with the supplied source files. The Makefile flow starts with your design input files and ends after Intel Quartus Prime synthesizes and places the output and a binary FPGA file that is ready for secure signing with PACSign.

The make flow is only supported on Linux* platforms. Your development system requires the following:

Make:

- make 3.81 (or newer)

Python:

- Python 3.6

You invoke the make flow with the following command input syntax:

- `make [target] [options] [paths] [versioning]`

Target—required, input only one:

- 2x1x25G
- 2x2x25G
- 4x25G
- 8x10G
- 2x1x25Gx2FVL
- 1x2x25G
- clean
- archive

Target archive stores whole database as a Quartus Archive (.qar) file.

Options

Option	Value	Description	Required	Default	Comment
GUI	0	run selected stage	NO	0	
	1	open Intel Quartus Prime GUI			
SEED	0 - 232-1	fitter seed	NO	1	Helpful in achieving timing closure
STAGE	compile	execute full flow (step-by-step)	YES	Not applicable	
	synthesis	execute analysis and synthesis			calls ipgenerate
	fitter	execute fitting (Fit, Place, Route)			calls ipgenerate
	fitter-timing	execute fitter and timing analysis			calls ipgenerate
	analysis-timing	execute timing analysis			requires completed fitter
	analysis-power	execute power analysis			requires completed fitter
	assembler	execute assembler			requires completed fitter
	ipgenerate	generate IPs			
	dummy	do nothing			GUI only
USE_BBS_CLK	0	do not take user clock from BBS	NO	0	
	1	take user clock from BBS			Required if CCI-P clock `uClk_usr` or `uClk_usrDiv2` is used in your design
INCLUDE_DIAGNOSTICS	0	exclude AFU diagnostics	NO	1	
	1	include AFU diagnostics			
INCLUDE_AFU_PCIE1	0	exclude AFU PCIe1	NO	1	

continued...

Option	Value	Description	Required	Default	Comment
	1	include AFU PCIe1			
INCLUDE_MEMORY	0	exclude all EMIFs	NO	1	
	1	include all EMIFs			
INCLUDE_DDR4_A	0	exclude DDR4 A	NO	INCLUDE_MEMORY	
	1	include DDR4 A			
INCLUDE_DDR4_B	0	exclude DDR4 B	NO	INCLUDE_MEMORY	
	1	include DDR4 B			
INCLUDE_DDR4_C	0	exclude DDR4 C	NO	INCLUDE_MEMORY	
	1	include DDR4 C			
INCLUDE_QDR	0	exclude QDR	NO	INCLUDE_MEMORY	
	1	include QDR			
MAC_LIGHTWEIGHT_MODE	0	disabled	NO	0	Disabled non required features in MACs for less logic consumption
	1	enabled			
DATAPATH_MODE	normal	Ethernet aggregated mode	NO	normal	
	disaggregated	disaggregated Ethernet mode			
	lightweight	lightweight Ethernet mode			
INCLUDE_SEU	0	Excludes SEU detection circuit	NO	1	
	1	Includes SEU detection circuit			
ARCHIVE_NAME	[any string]	`.qar` archive name for `.qdb` archive	NO	`snapshot.qar`	used by archive target

Usage Examples:

1. Set up your shell environment to use N3000 provided Intel Quartus Prime

```
$ source <N3000 Installation Directory>/inteldevstack/bin/init_env.sh
```

The example of the N3000 `Make` command runs the full compile process of the 2 x 2 x 25 GbE factory image using the command line (non-GUI) mode with fitter seed equal to 5:

```
$ make 2x2x25G SEED=5 STAGE=compile
```

Note: Some designs may require multiple seed passes using seeds 1 to 8 to achieve timing closure. Also, the Design Space Explorer does not work with the N3000 make design flow.

Paths

Path	Description	Default
PROJECT_FILE	main project qpf	prj/pac_baseline/chip.qpf
PAC_ROOT	N3000 sources root where main .qip is located	hw/pac
AFU_ROOT	AFU sources root where afu.qsf is located	hw/afu/hw

Versioning

Versioning	Description	Default
PAC_VER_MAJOR	SemVer Major. 0 - 15	0
PAC_VER_MINOR	SemVer Minor. 0 – 255	0
PAC_VER_PATCH	SemVer Patch. 0 – 15	0
REVISION_ID	32-bit	0
AFU_REVISION_ID	12-bit	0

Pr_Interface_ID

The OPAE tool fpgainfo lists the target configuration unique Pr_Interface_ID:

TARGET	Pr_Interface_ID
8x10G	901DD697-CA79-4B05-B843-8138CEFA2846
4x25G	F3C99413-5081-4AAD-BCED-07EB84A6D0BB
2x2x25G	A5D72A3C-C8B0-4939-912C-F715E5DC10CA

The build process combines your afu.qsf file with a top level chip.qsf that includes external memory interfaces, MACs, and the encrypted CCI-P and management blocks.

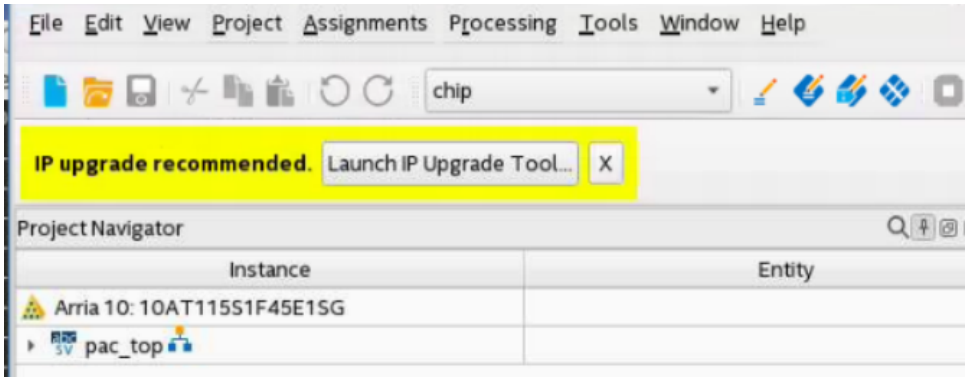
To compile the hello_afu targeting the 2x2x25 network interface, execute the following in the top directory:

```
$ cd $N3000_EXAMPLE_ROOT/hello_afu
$ make 2x2x25G GUI=1 INCLUDE_DIAGNOSTICS=0 INCLUDE_MEMORY=0 \
PAC_VER_MAJOR=3 PAC_VER_MINOR=5 PAC_VER_PATCH=6 \
REVISION_ID=12345678 INCLUDE_AFU_PCIE1=0
```

The following example steps use the Quartus GUI to illustrate the design flow. Once you are familiar with this flow you may prefer to use the non-GUI mode and additionally utilization of user created scripted or automated build flow.

This brings up the Intel Quartus Prime GUI. Click the **Start Compilation play** button.

Note: The **Launch IP Upgrade Tool** button appears. You can safely ignore this warning.



When your compile is complete, do not close Intel Quartus Prime, so that you can continue with the next steps.

Related Information

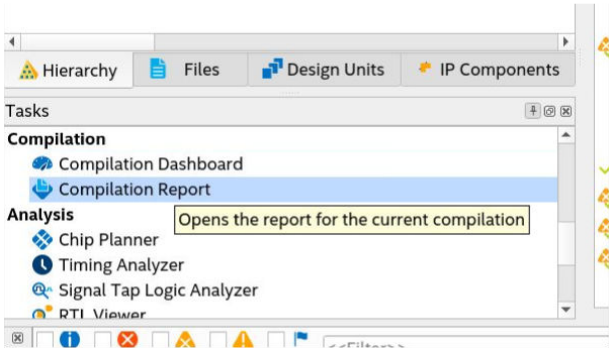
Ethernet Interface on page 19

4.4. Check Timing

Verify your compiled design meets timing and power requirements by performing the following steps:

1. Go to the "Tasks" pane and select "**Compilation Report**".

Figure 24. Compilation Report



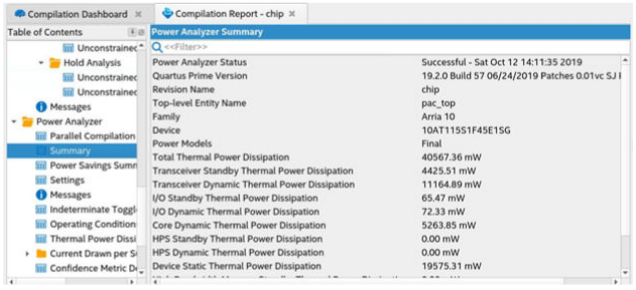
2. Under `Timing Analyzer`, verify no failing timing paths.

Figure 25. Timing Analyzer

You can safely ignore the "Unconstrained Paths" report in this release.

3. Select **Power Analyzer** and check that “Total Thermal Power Dissipation” is within the thermal characteristics of your server air flow.
4. Check the *Thermal Specifications* section of the *N3000 Data Sheet*.
   The power results shown in the power analyzer are based on the worst case FPGA junction temperature of 100° C.

Figure 26. Power Analyzer Summary

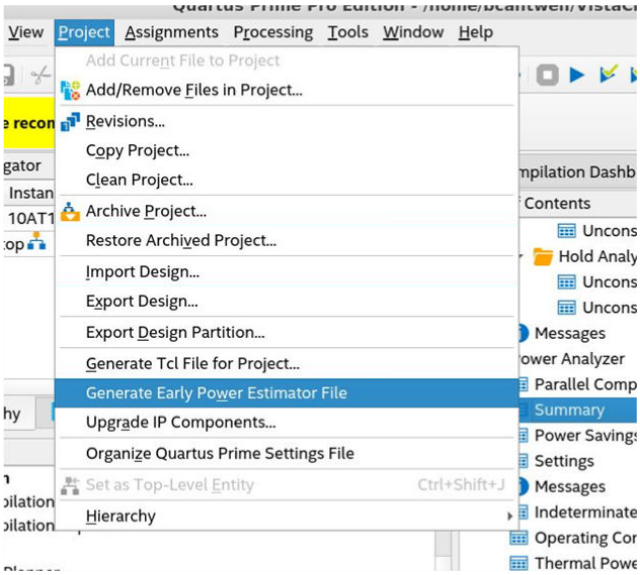


5. Select **Project ➤ Generate Early Power Estimator File** to perform additional power analysis in the Intel Arria 10 Early Power Estimator by clicking on this **download**.

   For more information, refer to the *Early Power Estimator for Intel Arria 10 FPGAs User Guide*.

   The Early Power Estimator (EPE) file can take a few minutes to generate. The default EPE file location is `/prj/pac_baseline/chip_early_pwr.csv`. This `.csv` file can be imported into the Early Power Estimator for detailed analysis of power consumption of your design.

Figure 27. Generate Early Power Estimator File



Related Information

Early Power Estimator for Intel Arria 10 FPGAs User Guide

4.5. Loading Your AFU into the Intel FPGA PAC N3000

Once your design has been compiled using the `make` process, a new directory is created with all build report files and FPGA programming files. To see these files, do the following after successfully running `make`:

Note: Must be in the same directory where `make` was invoked.

```
$ cd prj/pac_baseline/build/
$ $ ls -1
chip.asm.rpt
chip.done
chip.fit.finalize.rpt
chip.fit.place.rpt
chip.fit.plan.rpt
chip.fit.route.rpt
chip.fit.rpt
chip.fit.summary
chip.flow.rpt
chip_out.sof.rpt
chip.pin
chip.pow.rpt
chip.pow.summary
chip.sld
chip.sta.rpt
chip.sta.summary
chip.syn.rpt
chip.syn.smsg
chip.syn.summary
pac-n3000.map
pac-n3000-secure-update-raw.bin
pac-n3000.sof
```

The file `pac-n3000-secure-update-raw.bin` is a binary file formatted to be loaded into the N3000 FPGA flash. Before this file can be loaded into the flash, the prepended authentication blocks generated by PACSign must be added to the binary file prior to loading using the OPAE tool `fpgasupdate`. The following instructions guide you in creating an image file with the proper authentication blocks for an N3000 that has not had the root entry hash programmed. Typically, when developing an AFU in a lab environment, do not program the root entry hash until the AFU is production ready.

For more information, refer to the *Security User Guide for Intel FPGA Programmable Acceleration Card N3000 Variants*.

Before running PACSign, ensure you have the following environment setting:

```
export PYTHONPATH=/usr/local/lib/python3.6/site-packages/
```

1. Create the image and load using `fpgasupdate`.

   ```
   $ PACSign SR -t UPDATE -H openssl_manager  \
   -i pac-n3000-secure-update-raw.bin  -o unsigned_PAC_N3000_RSU.bin
   No root key specified.  Generate unsigned bitstream? Y = yes, N = no: y
   No CSK specified.  Generate unsigned bitstream? Y = yes, N = no: y
   ```

   By responding with 'y', you are creating an unsigned binary file that can be loaded into a N3000 board that has not had the root key hash loaded into flash.

2. Perform the `fpgasupdate` write process.

   ```
   $ sudo fpgasupdate unsigned_PAC_N3000_RSU.bin <PCIe B:D.F>
   ```

   Note: The `fpgasupdate` write process take approximately 40 minutes to complete.

3. Once `fpgasupdate` completes, perform a remote system update to load the new FPGA image, then verify the expected FPGA is loaded with OPAE tools `fpgainfo fme` and `fpgainfo port`.

   ```
   $ sudo rsu fpga <PCIe B:D.F>
   $ sudo fpgainfo fme
   $ sudo fpgainfo port
   ```

Related Information

Security User Guide for Intel FPGA Programmable Acceleration Card N3000 Variants

4.5.1. Loading Your FPGA Image with JTAG

In a development environment, you may wish to test new N3000 FPGA images without storing the image in the FPGA flash. By loading the image with JTAG, there is no 40-minute wait for the file to be loaded into flash. If the power is removed, upon power up, your new FPGA image is replaced with the image stored in the user location of FPGA flash.

You must obtain the following items:

- Intel FPGA Download Cable II (formerly the USB-Blaster II)

  Note: You need to follow installation instructions for this device.
- Samtec* SSQ-105-03-T-D Receptacle to extend the N3000 JTAG header

4.5.1.1. Preparing Your N3000 for JTAG

1. Remove the cover to the N3000 board by removing the screws and lifting off cover, as shown below:

Figure 28. N3000 with Cover Attached

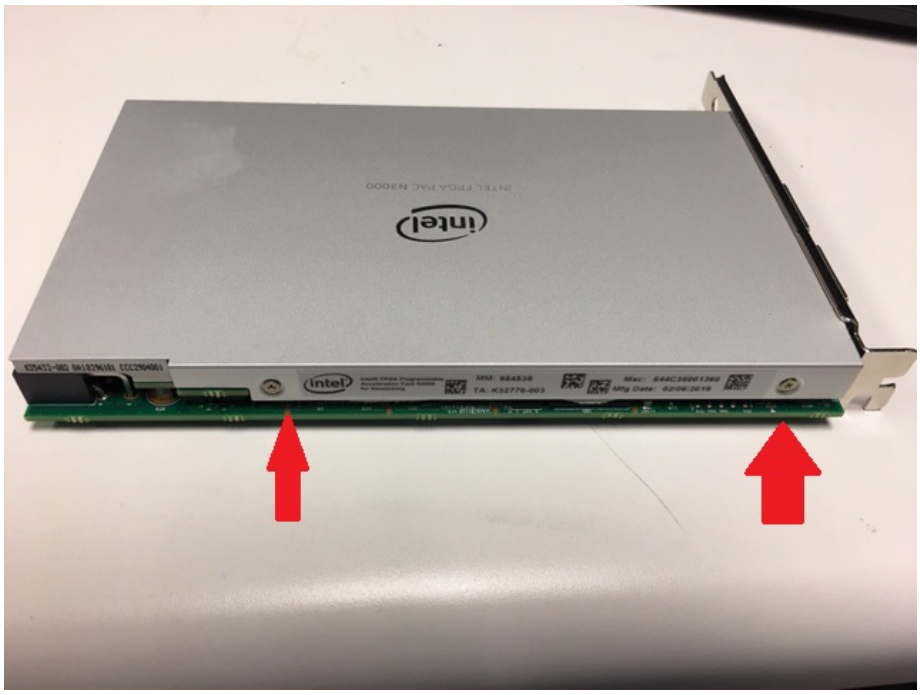
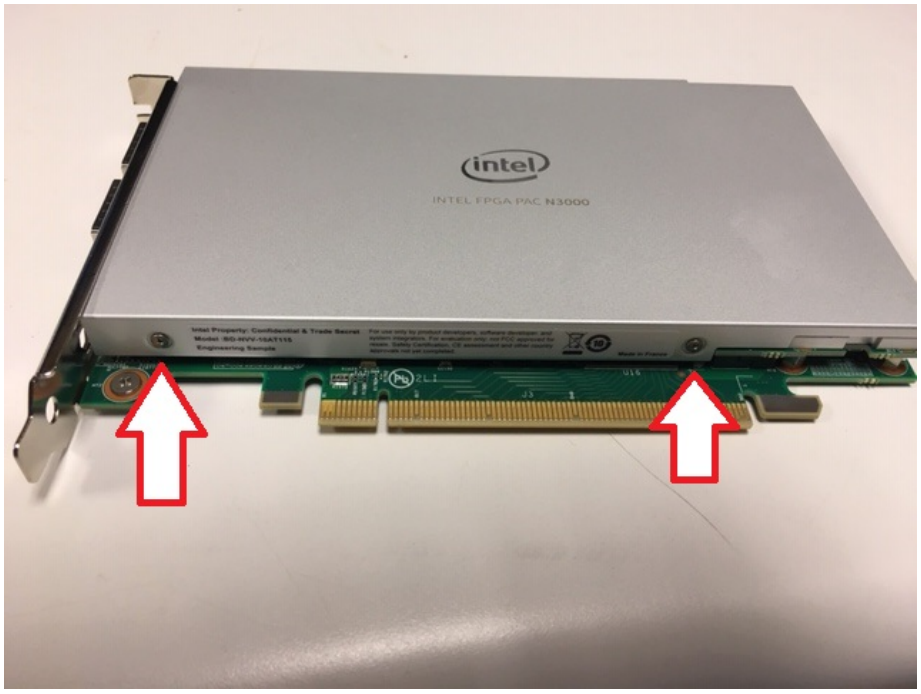


Figure 29. N3000 Board Exposed



2. Turn the N3000 board over to the backside and locate "SW2" as shown below:

Figure 30. Bottom Side of the N3000 Board

Note: The "SW2" Switch must be in the **OFF** position.

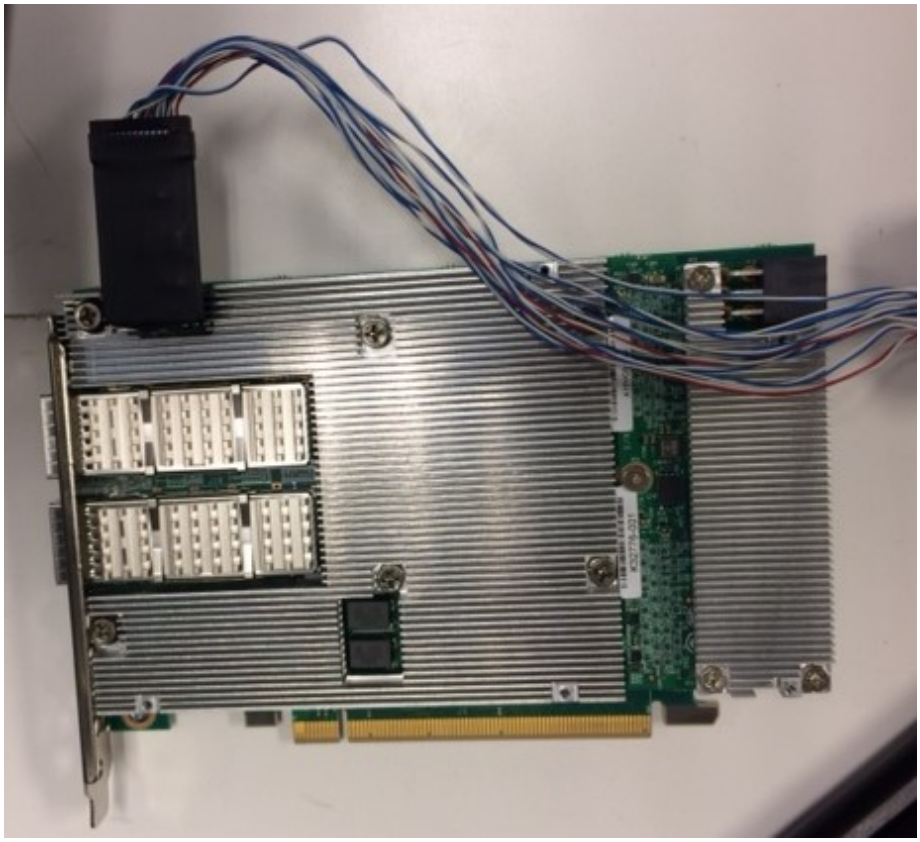
You can tell the switch is set to **OFF** by using an Ohm meter to measure resistance across the switch. When the switch is set to **OFF**, the resistance should be approximately 10-13 K Ohms. Slide the switch using a probe tool to **OFF** if required.

3. Use a 10 Position receptacle, as shown below, to extend the N3000 JTAG header for connectivity above the heatsink to the Intel FPGA Download Cable II. **Note:** Pay attention to the manufacturer and part number.

Figure 31. 10 Position Receptacle

4. Attach the Intel FPGA Download Cable II header as shown below:

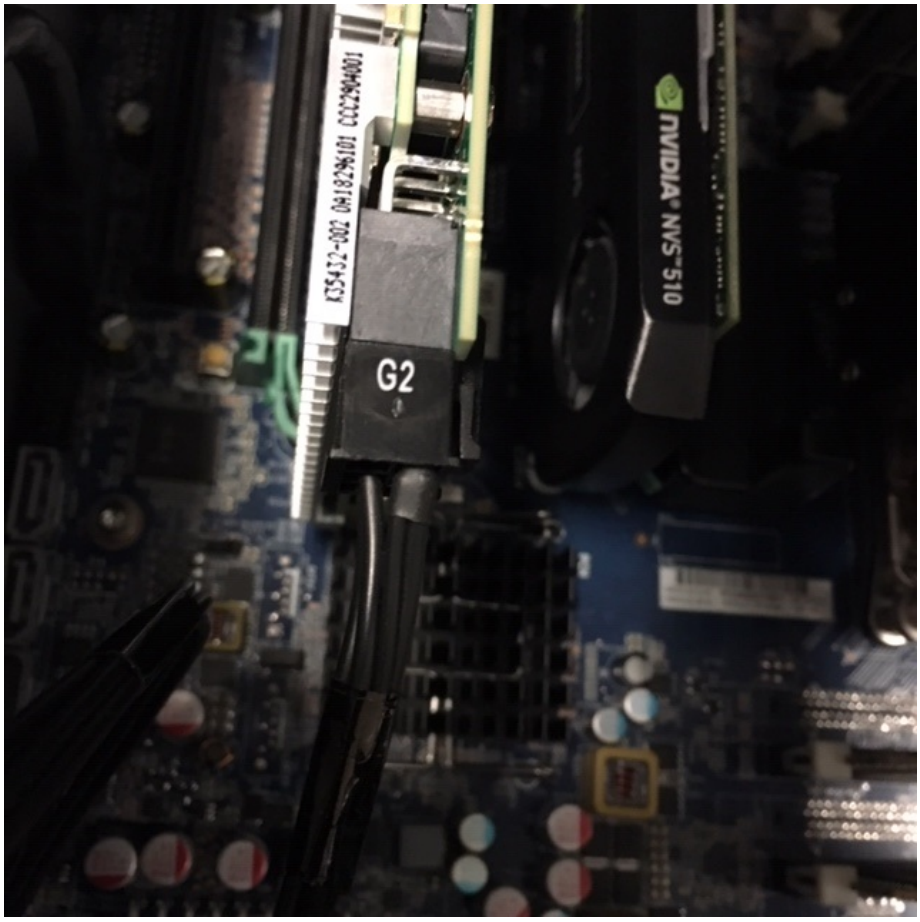
Figure 32. Intel FPGA Download Cable II Attached to the N3000



Note: Notice the Intel FPGA Download Cable II orientation as shown above. There is no keying of this connector.

5. Turn **OFF** the power to the server and insert the N3000 card into a PCIe Gen3 X 16 slot.
6. Attach the auxiliary 12 V connection to the N3000 card, as shown below:

Figure 33. Auxiliary 12 V Connection



Caution: Make absolutely certain substantial server air flow velocity is provided. The card will overheat if adequate airflow is not provided.

7. Power on server.

4.5.1.2. Disabling PCIe Automatic Error Reporting (AER)

When you program the FPGA using JTAG, the Intel Arria 10 PCIe link goes down for a moment causing a server surprise link down event. To prevent this server event, temporarily disable the PCIe AER for the N3000 PCIe slot using the following steps:

1. Find and record your N3000 PCIe `s:b:d.f` value. You will use this PCIe `s:b:d.f` value later for removing the N3000 from the PCIe bus. In this example, use this value: `0000:08:00.0`.

```
$ sudo fpgainfo fme
Board Management Controller, MAX10 NIOS FW version D.2.0.19
Board Management Controller, MAX10 Build version D.2.0.6
//****** FME ******//
Object Id                        : 0xF200000
PCIe s:b:d.f                     : 0000:08:00.0
Device Id                        : 0x0b30
Numa Node                        : 0
Ports Num                        : 01
Bitstream Id                     : 0x23000410010309
```

```
Bitstream Version            : 0.2.3
Pr Interface Id              : a5d72a3c-c8b0-4939-912c-f715e5dc10ca
Boot Page                    : user
```

2. Use the command `find_RP.sh` to get board root `PCIe s:b:d.f`.

```
$ cd <N3000 Install Directory>/N3000_supplemental_files/
$ ./find_RP.sh
0000:00:03.0                    ----- >>> This is root port, take note of
this value
0000:03:00.0
0000:04:09.0
0000:08:00.0 -> intel-fpga-dev.0
```

3. The first entry in the list is the PCIe Root port. In this example, `0000:00:03.0` is the root port. Your values may be different. The last entry is `intel-fpga-dev.0`.
4. Using the root port, find the current AER settings and record the value. Use this value when you re-enable AER.

```
$ sudo setpci -s 0000:00:03.0 ECAP_AER+0x08.L
00000000
$ sudo setpci -s 0000:00:03.0 ECAP_AER+0x14.L
00002000
```

5. Disable AER for the root port:

```
$ sudo setpci -s 0000:00:03.0 ECAP_AER+0x08.L=0xffffffff
$ sudo setpci -s 0000:00:03.0 ECAP_AER+0x14.L=0xffffffff
```

6. Using your board `PCIe s:b:d.f`, remove the N3000 from the PCIe bus. If using RHEL, you must enter the command as root:

```
# sudo echo 1 > /sys/bus/pci/devices/0000:08:00.0/remove
```

4.5.1.3. Using JTAG to Load the Intel Arria 10 *.sof file

1. Start Intel Quartus Prime Programmer

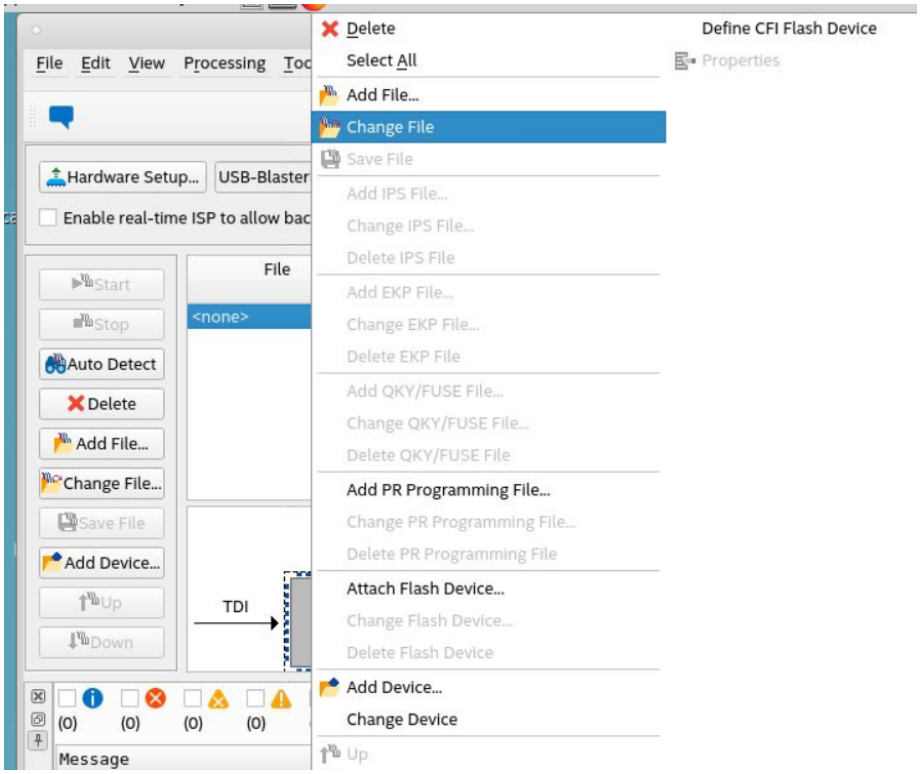
```
$ source <N3000 Installation Directory>/inteldevstack/bin/init_env.sh
$ quartus_pgmw
```

2. Select auto detect and select device 10AT115S1. If you see 10M50 as the device, then switch SW2 is not set properly – you must uninstall the card and change SW2.

Figure 34. Select Device GUI

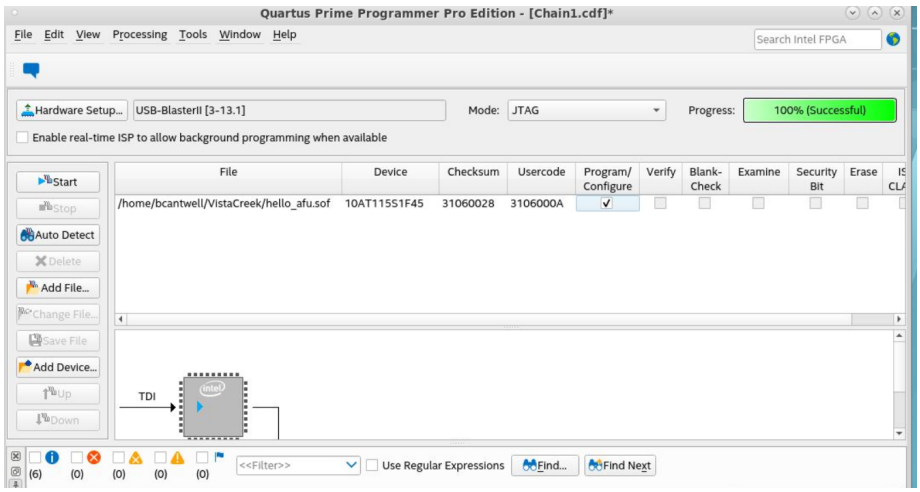
3. Right click the **File** column and select **Change File**.

Figure 35. Change File Selection



4. Navigate to the `pac-n3000.sof` file, select **Program/Configure** and press **Start**. This programs the Intel Arria 10 FPGA with the `pac-n30000.sof` file. Wait until the 100% (Successful) is shown under progress:

Figure 36. Intel Quartus Prime Programmer Pro Edition GUI



4.5.1.4. How to Rescan PCIe Bus and Re-enable PCIe AER

1. Rescan the PCIe bus to register the new FPGA.

```
# sudo echo 1 > /sys/bus/pci/rescan
```

2. Verify the new FPGA is present by checking expected bitstream ID and AFU ID using commands:

```
$ sudo fpgainfo fme
$ sudo fpgainfo port
```

3. Re-enable AER using the values read in Step 4 on page 51 of section Disabling PCIe Automatic Error Reporting (AER) on page 50 for the card under test:

```
$ sudo setpci -s 0000:00:03.0 ECAP_AER+0x08.L=0x00000000
$ sudo setpci -s 0000:00:03.0 ECAP_AER+0x14.L=0x00002000
```

You can now run your host application with the FPGA image you loaded with JTAG.

4.5.2. AFU Clocks

The `hello_afu` example uses the CCI-P `pClk` for synchronization. In this section, two examples are presented where the `hello_afu` example is modified to use a `uClk_usr` in the first example and a user instantiated PLL.

In both of these examples, the BBB_ccip_async is used to perform clock crossing for the CCI-P interface.

4.5.2.1. Hello AFU Example (`uClk_usr`)

The module `hello_afu.sv` is modified to instantiate the BBB_ccip_async module to provide a clock crossing for the CCI-P interface between `pClk` and `uClk_usr` domains. The modified code is shown below:

```
import ccip_if_pkg::*;
module hello_afu_uClk_usr
   (
    input  pClk,    // Core clock. CCI interface is synchronous to this clock.
    input  pClk_reset,  // CCI interface ACTIVE HIGH reset.

    input uClk_usr, //312.5 MHz user clock

    // CCI-P signals
    input  t_if_ccip_Rx pClk_cp2af_sRxPort,
    output t_if_ccip_Tx pClk_af2cp_sTxPort
    );

    `define AFU_ACCEL_UUID 128'h850adcc2_6ceb_4b22_9722_d43375b61c66
    // The AFU must respond with its AFU ID in response to MMIO reads of
    // the CCI-P device feature header (DFH).  The AFU ID is a unique ID
    // for a given program.  Here we generated one with the "uuidgen"
    // program and stored it in the AFU's JSON file.  ASE and synthesis
    // setup scripts automatically invoke the OPAE afu_json_mgr script
    // to extract the UUID into afu_json_info.vh.
    logic [127:0] afu_id = `AFU_ACCEL_UUID;

    logic [63:0] scratch_reg;

    //uClk_usr domain CCIP signals
    t_if_ccip_Tx af2cp_sTxPort;
    t_if_ccip_Rx cp2af_sRxPort;

    ccip_async_shim ccip_async_shim (
```

```
                    .bb_softreset    (pClk_reset),
                    .bb_clk          (pClk),
                    .bb_tx           (pClk_af2cp_sTxPort),
                    .bb_rx           (pClk_cp2af_sRxPort),
                    .afu_softreset   (reset),
                    .afu_clk         (uClk_usr),
                    .afu_tx          (af2cp_sTxPort),
                    .afu_rx          (cp2af_sRxPort)
                    );


    // The c0 header is normally used for memory read responses.
    // The header must be interpreted as an MMIO response when
    // c0 mmmioRdValid or mmioWrValid is set.  In these cases the
    // c0 header is cast into a ReqMmioHdr.
    t_ccip_c0_ReqMmioHdr mmioHdr;
    assign mmioHdr = t_ccip_c0_ReqMmioHdr'(cp2af_sRxPort.c0.hdr);

    //
    // Receive MMIO writes
    //
    always_ff @(posedge uClk_usr)
    begin
        if (reset)
        begin
            scratch_reg <= '0;
        end
        else
        begin
            // set the registers on MMIO write request
            // these are user-defined AFU registers at offset 0x40.
            if (cp2af_sRxPort.c0.mmioWrValid == 1)
            begin
                case (mmioHdr.address)
                    16'h0020: scratch_reg <= cp2af_sRxPort.c0.data[63:0];
                endcase
            end
        end
    end

    //
    // Handle MMIO reads.
    //
    always_ff @(posedge uClk_usr)
    begin
        if (reset)
        begin
            af2cp_sTxPort.c1.hdr <= '0;
            af2cp_sTxPort.c1.valid <= '0;
            af2cp_sTxPort.c0.hdr <= '0;
            af2cp_sTxPort.c0.valid <= '0;
            af2cp_sTxPort.c2.hdr <= '0;
            af2cp_sTxPort.c2.mmioRdValid <= '0;
        end
        else
        begin
            // Clear read response flag in case there was a response last cycle.
            af2cp_sTxPort.c2.mmioRdValid <= 0;

            // serve MMIO read requests
            if (cp2af_sRxPort.c0.mmioRdValid == 1'b1)
            begin
                // Copy TID, which the host needs to map the response to the
request
                af2cp_sTxPort.c2.hdr.tid <= mmioHdr.tid;

                // Post response
                af2cp_sTxPort.c2.mmioRdValid <= 1;

                case (mmioHdr.address)
                    // AFU header
```

```
                    16'h0000: af2cp_sTxPort.c2.data <= {
                        4'b0001, // Feature type = AFU
                        8'b0,    // reserved
                        4'b0,    // afu minor revision = 0
                        7'b0,    // reserved
                        1'b1,    // end of DFH list = 1
                        24'b0,   // next DFH offset = 0
                        4'b0,    // afu major revision = 0
                        12'b0    // feature ID = 0
                        };

                    // AFU_ID_L
                    16'h0002: af2cp_sTxPort.c2.data <= afu_id[63:0];

                    // AFU_ID_H
                    16'h0004: af2cp_sTxPort.c2.data <= afu_id[127:64];

                    // DFH_RSVD0 and DFH_RSVD1
                    16'h0006: af2cp_sTxPort.c2.data <= 64'h0;
                    16'h0008: af2cp_sTxPort.c2.data <= 64'h0;

                    // Scratch Register.  Return the last value written
                    // to this MMIO address.
                    16'h0020: af2cp_sTxPort.c2.data <= scratch_reg;

                    default:  af2cp_sTxPort.c2.data <= 64'h0;
                endcase
            end
        end
    end
endmodule
```

You must edit `ccip_std_afu.sv` to connect the clock to your AFU. Addition to the `ccip_std_afu.sv` is shown below:

```
hello_afu_pll afu
       (
        .pClk            (pClk),
        .pClk_reset         (pck_cp2af_softReset_T1),
          . uClk_usr      (uClk_usr),

        .pClk_cp2af_sRxPort  (pck_cp2af_sRx_T1),
        .pClk_af2cp_sTxPort  (pck_af2cp_sTx_T0)
        );
```

The file `afu.qsf` is modified to source the `ccip_async` additions as shown below:

```
# CCI-P async shim
source $AFU_SRC_ROOT/rtl/BBB_ccip_async/hw/par/ccip_async_addenda.qsf
```

The `afu.sdc` file has this additional constraint added:

```
set_false_path -from [get_clocks {sys_csr_clk_pll|outclk[0]}]\
-to [get_clocks {fpga_top|inst_fiu_top|inst_ccip_fabric_top|inst_cvl_top|\
inst_user_clk|qph_user_clk_fpll_u0|xcvr_fpll_a10_0|outclk1}]
```

Then the build process is invoked with this `make` command:

```
make 2x2x25G GUI=1 INCLUDE_DIAGNOSTICS=0 INCLUDE_MEMORY=0 \
PAC_VER_MAJOR=3 PAC_VER_MINOR=5 PAC_VER_PATCH=6 \
REVISION_ID=12345678 INCLUDE_AFU_PCIE1=0 USE_BBS_CLK=1
```

4.5.2.2. Hello AFU Example (`pll`)

A new PLL can be instantiated to provide additional clocks in your design. These steps are performed to add an Intel Arria 10 IOPLL to the `hello_afu` design:

1. Create IOPLL in using IP Catalog and set PLL to desired settings.
2. Instantiate PLL in `hello_afu` with `ccip_async_shim` to perform clock boundary crossing.
3. Edit `ccip_std_afu.sv` to connect `G_CLK100` to AFU.
4. Update the `*.qsf` and `*.sdc` files.

The updated `hello_afu` module is listed below:

```
import ccip_if_pkg::*;
module hello_afu_pll
   (
    input  pClk,    // Core clock. CCI interface is synchronous to this clock.
    input  pClk_reset,  // CCI interface ACTIVE HIGH reset.

    input G_CLK100, //100 MHz Global clock for PLL

    // CCI-P signals
    input  t_if_ccip_Rx pClk_cp2af_sRxPort,
    output t_if_ccip_Tx pClk_af2cp_sTxPort
    );

    `define AFU_ACCEL_UUID 128'h850adcc2_6ceb_4b22_9722_d43375b61c66
    // The AFU must respond with its AFU ID in response to MMIO reads of
    // the CCI-P device feature header (DFH).  The AFU ID is a unique ID
    // for a given program.  Here we generated one with the "uuidgen"
    // program and stored it in the AFU's JSON file.  ASE and synthesis
    // setup scripts automatically invoke the OPAE afu_json_mgr script
    // to extract the UUID into afu_json_info.vh.
    logic [127:0] afu_id = `AFU_ACCEL_UUID;

    logic [63:0] scratch_reg;

pll_50Mhz u0 (
        .rst      (pClk_reset),      //   input,  width = 1,   reset.reset
        .refclk   (G_CLK100),   //   input,  width = 1,  refclk.clk
        .locked   (),   //  output,  width = 1,  locked.export
        .outclk_0 (uClk_50)  //  output,  width = 1, outclk0.clk
    );


    //uClk_usr domain CCIP signals
    t_if_ccip_Tx af2cp_sTxPort;
    t_if_ccip_Rx cp2af_sRxPort;

    ccip_async_shim ccip_async_shim (
                    .bb_softreset    (pClk_reset),
                    .bb_clk          (pClk),
                    .bb_tx           (pClk_af2cp_sTxPort),
                    .bb_rx           (pClk_cp2af_sRxPort),
                    .afu_softreset   (reset),
                    .afu_clk         (uClk_50),
                    .afu_tx          (af2cp_sTxPort),
                    .afu_rx          (cp2af_sRxPort)
                    );


    // The c0 header is normally used for memory read responses.
    // The header must be interpreted as an MMIO response when
    // c0 mmmioRdValid or mmioWrValid is set.  In these cases the
    // c0 header is cast into a ReqMmioHdr.
    t_ccip_c0_ReqMmioHdr mmioHdr;
```

```
    assign mmioHdr = t_ccip_c0_ReqMmioHdr'(cp2af_sRxPort.c0.hdr);

    //
    // Receive MMIO writes
    //
    always_ff @(posedge uClk_50)
    begin
        if (reset)
        begin
            scratch_reg <= '0;
        end
        else
        begin
            // set the registers on MMIO write request
            // these are user-defined AFU registers at offset 0x40.
            if (cp2af_sRxPort.c0.mmioWrValid == 1)
            begin
                case (mmioHdr.address)
                    16'h0020: scratch_reg <= cp2af_sRxPort.c0.data[63:0];
                endcase
            end
        end
    end

    //
    // Handle MMIO reads.
    //
    always_ff @(posedge uClk_50)
    begin
        if (reset)
        begin
            af2cp_sTxPort.c1.hdr <= '0;
            af2cp_sTxPort.c1.valid <= '0;
            af2cp_sTxPort.c0.hdr <= '0;
            af2cp_sTxPort.c0.valid <= '0;
            af2cp_sTxPort.c2.hdr <= '0;
            af2cp_sTxPort.c2.mmioRdValid <= '0;
        end
        else
        begin
            // Clear read response flag in case there was a response last cycle.
            af2cp_sTxPort.c2.mmioRdValid <= 0;

            // serve MMIO read requests
            if (cp2af_sRxPort.c0.mmioRdValid == 1'b1)
            begin
                // Copy TID, which the host needs to map the response to the
request
                af2cp_sTxPort.c2.hdr.tid <= mmioHdr.tid;

                // Post response
                af2cp_sTxPort.c2.mmioRdValid <= 1;

                case (mmioHdr.address)
                    // AFU header
                    16'h0000: af2cp_sTxPort.c2.data <= {
                        4'b0001, // Feature type = AFU
                        8'b0,    // reserved
                        4'b0,    // afu minor revision = 0
                        7'b0,    // reserved
                        1'b1,    // end of DFH list = 1
                        24'b0,   // next DFH offset = 0
                        4'b0,    // afu major revision = 0
                        12'b0    // feature ID = 0
                        };

                    // AFU_ID_L
                    16'h0002: af2cp_sTxPort.c2.data <= afu_id[63:0];

                    // AFU_ID_H
                    16'h0004: af2cp_sTxPort.c2.data <= afu_id[127:64];
```

```
                    // DFH_RSVD0 and DFH_RSVD1
                    16'h0006: af2cp_sTxPort.c2.data <= 64'h0;
                    16'h0008: af2cp_sTxPort.c2.data <= 64'h0;

                    // Scratch Register.  Return the last value written
                    // to this MMIO address.
                    16'h0020: af2cp_sTxPort.c2.data <= scratch_reg;

                    default:  af2cp_sTxPort.c2.data <= 64'h0;
                endcase
            end
        end
    end
endmodule
```

The `afu.qsf` is updated as shown below:

```
# CCI-P async shim
source $AFU_SRC_ROOT/rtl/BBB_ccip_async/hw/par/ccip_async_addenda.qsf

set_global_assignment -name IP_FILE             $AFU_SRC_ROOT/rtl/pll/
pll_50Mhz.ip

set_instance_assignment -name GLOBAL_SIGNAL GLOBAL_CLOCK -to G_CLK100 -entity
pac_top
```

Compile the design with `make` as shown below:

```
$ make 2x2x25G INCLUDE_DIAGNOSTICS=0 INCLUDE_MEMORY=0 INCLUDE_AFU_PCIE1=0 GUI=1
```

Note: You may need to modify the `afu.sdc` file based on the new clock added.

4.5.3. Creating an AFU with High Level Synthesis (HLS)

This section describes how to create an AFU using HLS.

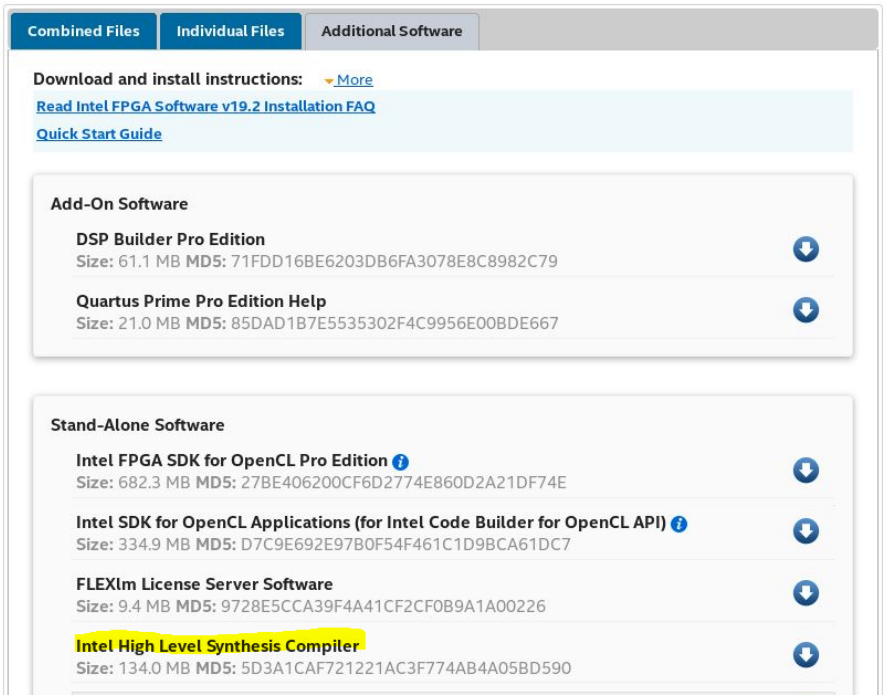
The *Intel High Level Synthesis Accelerator Functional Unit (AFU) Design Example User Guide* is adapted to the N3000 design flow to instruct the reader in performing steps to create a new AFU with the HLS design methodology.

You must obtain the Intel FPGA Programmable Acceleration Card N3000 HLS AFU Design Example code from an Intel Sales Agent.

Install HLS and set up your environment.

1. Download the HLS tool from the Intel website and install.
   a. From the Download Center for FPGAs web page, select "Additional Software".
   b. Download "Intel High Level Synthesis Compiles".

Figure 37. Additional Software Tab

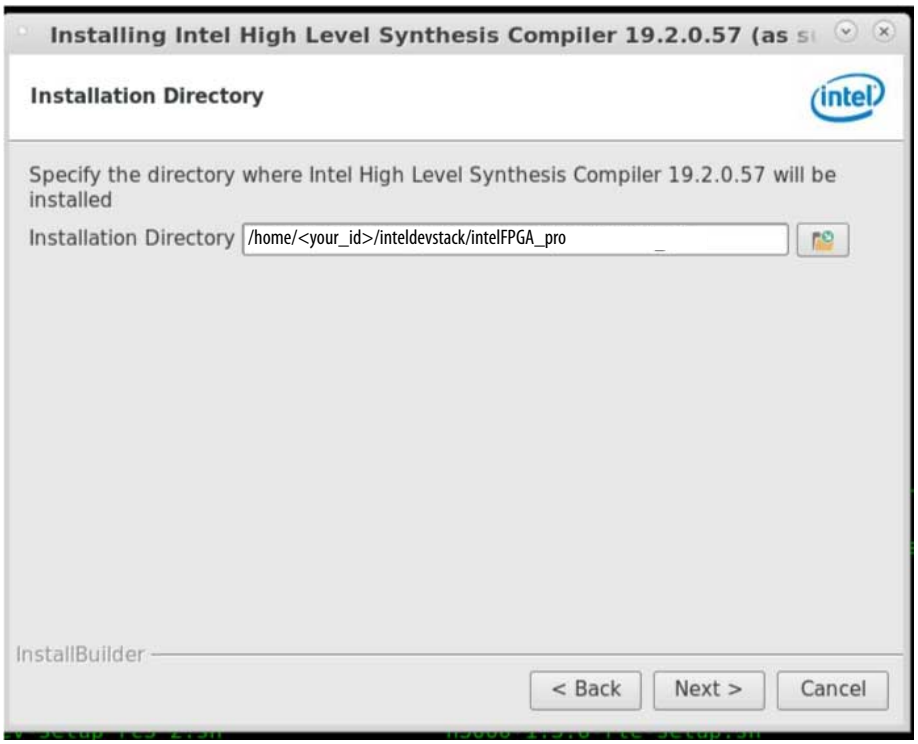


2. Set downloaded `HLSProSetup-19.2.0.57-linux.run` file as executable and run:

```
$ chmod +x HLSProSetup-19.2.0.57-linux.run
$ sudo ./HLSProSetup-19.2.0.57-linux.run
```

3. Select the N3000 Quartus Development install directory as the Installation Directory for HLS Compiler as shown below:

Figure 38. Installing Directory GUI



4. Follow instructions in Section 1.3 of the *Intel High Level Synthesis Compiler Pro Edition: Getting Started Guide*.

   For setup of the HLS Compiler, make the HLS initialization script executable:

   ```
   $ chmod +x <N3000 Install Directory>/inteldevstack/intelFPGA_pro/hls/
   init_hls.sh
   ```

   This completes the installation process.

Related Information

- Intel High Level Synthesis Accelerator Functional Unit (AFU) Design Example User Guide
- Intel High Level Synthesis Compiler Pro Edition: Getting Started Guide

4.5.3.1. Setting Up a Shell for HLS Development Work

1. Once the HLS is set up, set up the shell for a N3000 development environment and HLS.

   ```
   $ source <N3000 Install Directory>/inteldevstack/bin/init_env.sh
   $ source  <N3000 Install Directory>/inteldevstack/intelFPGA_pro/hls/
   init_hls.sh
   ```

2. You must have ModelSim installed and your PATH variable set to invoke `vsim` setup on your machine.
3. Put Platform Designer version 19.2 in your PATH, using the following command:

```
$ export PATH=$PATH: <N3000 Install Directory>/inteldevstack/intelFPGA_pro/\
qsys/bin
```

You are now ready to run the HLS example in this shell. You must use the above steps for future N3000 HLS development shell set up.

The N3000 AFU design environment differs from the Intel PAC with Intel Arria 10 GX FPGA:

- N3000 uses a flat design with a static binary file rather than a partial reconfiguration.
- N3000 does not support an ASE simulation environment.
- N3000 does not support a Platform Interface Manager design flow.

The N3000 HLS design flow is changed to accommodate these differences.

4.5.3.2. Using the Initial Shell Design as a Shell

The Intel N3000 Acceleration Stack for Development provides the `Initial_Shell_Design` as a starting point for your created designs. The `Initial_Shell_Design` is used as a shell for inclusion of the HLS AFU example.

1. Copy the provided `Initial_Shell_Design` to a new directory for your tutorial work.

```
$ mkdir hls_example
$ cd hls_example
$ cp -R $N3000_EXAMPLE_ROOT/Initial_Shell_AFU/*
```

2. Copy the Intel provided HLS AFU example `.tar` file to your `hls_example/hw directory` and `untar`

```
$ cp  <Download directory>/hls_afu_2019-04-30.tar hls_example/hw/.
$ cd hls_example/hw
$ tar xf hls_afu_2019-04-30.tar
$ cd hls_afu/hw/rtl/hls
```

3. Build and emulate the design using x86 instructions and run these commands:

```
$ make test-x86-64
i++ src/hls_afu.cpp src/test.cpp   --fp-relaxed -ghdl  -march=x86-64 -o
test-x86-64
+-------------------------------------------+
| Run ./test-x86-64 <n> to execute the test. |
| <n> is 0, 1, or 2 depending on desired     |
| test behavior:                             |
|         <n> | effect                       |
|       ------+--------------------          |
|          0  | test both (default)          |
|          1  | test ac_int only             |
|          2  | test float only              |
+-------------------------------------------+
```

```
$ ./test-x86-64
Control which component gets tested by passing an integer!
arg   | effect
------+--------------------
   0  | test both (default)
```

```
   1  | test ac_int only
   2  | test float only
test AC_INT version and FLOAT version

AC_INT COMPONENT - 81 ELEMENTS
ac_inc:
sizeof(uint512) = 64 (64)
number of 512 bit (64-byte) numbers: 6
PASS

FLOATING-POINT COMPONENT - 81 ELEMENTS
fp_inc:
PASS
OVERALL:
PASSED
```

4. Generate RTL and simulate the generated RTL with the ModelSim simulator:

```
$ make test-fpga
$ ./test-fpga
Control which component gets tested by passing an integer!
arg   | effect
------+--------------------
   0  | test both (default)
   1  | test ac_int only
   2  | test float only
test AC_INT version and FLOAT version

AC_INT COMPONENT - 81 ELEMENTS
ac_inc:
sizeof(uint512) = 64 (64)
number of 512 bit (64-byte) numbers: 6
PASS

FLOATING-POINT COMPONENT - 81 ELEMENTS
fp_inc:
PASS
OVERALL:
PASSED
```

5. Confirm that the outputs from the `test-x86-64` command and the `test-fpga` command match.

   The `test-x86-64` command runs C++ code on the processor, while the `test-fpga` command compiles the C++ source to Verilog RTL and then simulates the generated RTL using the testbench defined in the code.

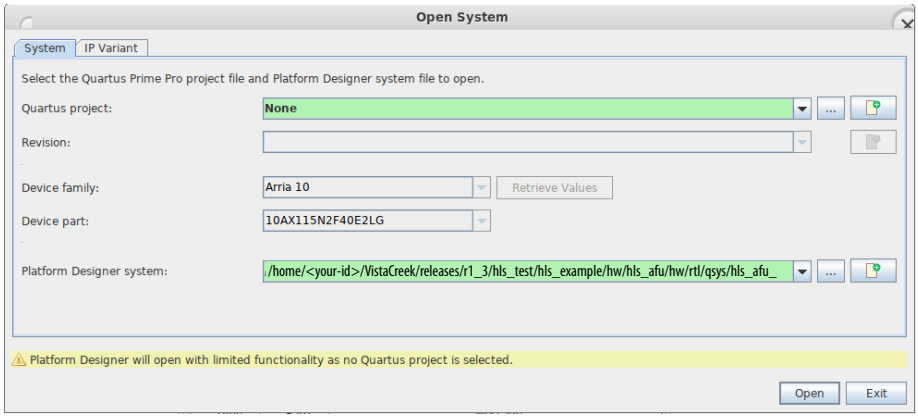
   For instructions about how to view the waveforms for this component, see the *Intel High Level Synthesis Compiler User Guide*.

6. Navigate to the `qsys` directory and open the system using Platform Designer.

```
$ cd ../qsys
$ qsys-edit hls_afu_container.qsys
```

   In the **Open System** dialog box, select **None** for the **Quartus project** dropdown.

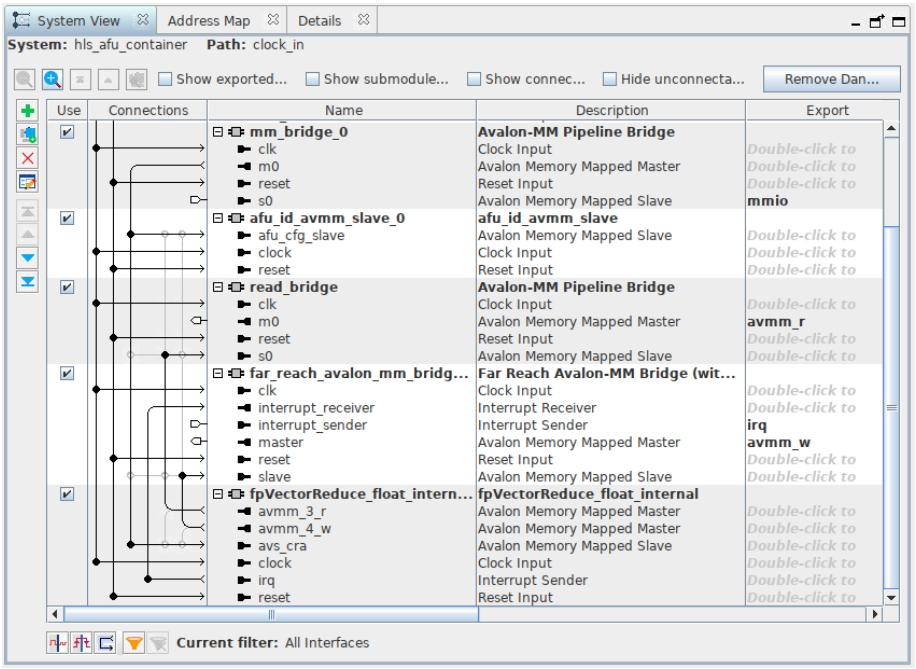
Figure 39. Open System GUI



Note: You can safely ignore the device part number for this example.

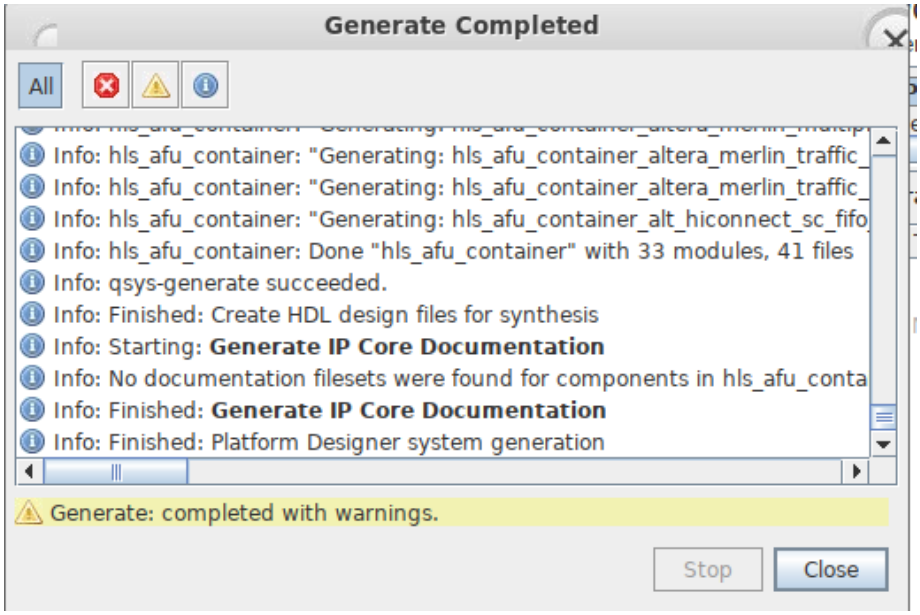
7. Click **Open**.
8. Click **Close** on the **Open System Completed** pop up.
9. To reload the system and ensure that all search paths are correct, click on **Validate System Integrity** at the bottom of the Platform Designer window.
10. After Validate System Integrity successfully completes, click **Close**. Investigate connectivity of Platform Designer components.

Figure 40. System View GUI



11. Generate HDL by clicking **Generate HDL**, then in **Generation** pop up, click **Generate** and **Save Changes**. You may safely ignore warnings. Click **Close**.

Figure 41. Generate Completed GUI



12. Exit Platform Designer and change directory to the `hw/hls_afu/hw/rtl` directory and verify contents:

```
$ cd ..
$ ls
afu.sv  BBB_cci_mpf  BBB_ccip_avmm  cci-if  ccip_interface_reg.sv
ccip_std_afu.sv  filelist.txt  hls  hls_afu.json  pcie  qsys
```

The N3000 does not support Intel AFU Simulation Environment (ASE) for co-simulation of AFU RTL and host software.

Related Information

Intel High Level Synthesis Accelerator Functional Unit (AFU) Design Example User Guide

4.5.3.3. Compiling the Design and Producing a new N3000 FPGA Bitstream

To compile the design and produce a new N3000 FPGA bitstream, perform the following steps:

1. Copy the `cci-if` and `pcie` directories to the `hw/hls_afu/hw/rtl` directory

```
$ cp -R ../../../afu/hw/rtl/cci-if .
$ cp -R ../../../afu/hw/rtl/pcie .
```

The `cci-if` and `pcie` directories were included in the `Intial_Shell_Desgin` and these directories are needed to compile the HLS AFU example.

2. The HLS Example provides an `ccip_std_afu.sv` file that is based on the N3000. You must update the `ccip_std_afu.sv` file with N3000 interfaces. The updated code is shown below:

```
//
// ***************************************************************************
// Copyright (c) 2013-2016, Intel Corporation
//
```

```
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
met:
//
// * Redistributions of source code must retain the above copyright notice,
// this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
notice,
// this list of conditions and the following disclaimer in the documentation
// and/or other materials provided with the distribution.
// * Neither the name of Intel Corporation nor the names of its contributors
// may be used to endorse or promote products derived from this software
// without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
THE
// POSSIBILITY OF SUCH DAMAGE.
//
// Module Name :    ccip_std_afu
// Project :        ccip afu top
// Description :    This module instantiates CCI-P compliant AFU

//
******************************************************************************
// Include MPF data types, including the CCI interface pacakge.

//import ccip_if_pkg::*;
`include "cci_mpf_if.vh"
import cci_mpf_csrs_pkg::*;


module ccip_std_afu #(
    parameter UPL_VERSION                       = 32'h2019_0905,
    parameter LOG2_DP_DATA_PATH1_WIDTH          = 9,
    parameter LOG2_DP_DATA_PATH0_WIDTH          = 9,
    parameter DP_CHA_WIDTH                      = 2,
    parameter NUM_AVST_IF_LINE                  = 2,
    parameter NUM_AVST_IF_FVL                   = 2,
    parameter LOG2_MAC_DATA_WIDTH               = 6,
    parameter USR_ERROR_WIDTH                   = 1,
    parameter LIGHTWEIGHT_MODE                  = 0,
    parameter TIMESTAMP_PASS                    = 0,

    parameter int TIMESTAMP_WIDTH              = 96
) (
    input  logic                                  G_CLK100,// 100MHz global
reference clock
    input  logic
pClk,                    // 400MHz - CCI-P clock domain. Primary interface
clock
    input  logic
pClkDiv2,                // not used.
    input  logic
pClkDiv4,                // not used.
    input  logic
uClk_usr,                // User clock domain. Refer to clock programming
guide
    input  logic
uClk_usrDiv2,            // User clock domain. Half the programmed
frequency
```

```
    input  logic
pck_cp2af_softReset,      // CCI-P ACTIVE HIGH Soft Reset
    input  logic [1:0]
pck_cp2af_pwrState,       // CCI-P AFU Power State
    input  logic
pck_cp2af_error,          // CCI-P Protocol Error Detected
    // Interface structures
    input  t_if_ccip_Rx                        pck_cp2af_sRx,           //
CCI-P Rx Port
    output t_if_ccip_Tx                        pck_af2cp_sTx,           //
CCI-P Tx Port

    input  logic                               pcie1_pipe_gen3_x8_ref_clk,
    input  logic                               pcie1_pcie_pins_perst_n,
    input  logic [7:0]
pcie1_pipe_gen3_x8_rx_serial,
    output logic [7:0]
pcie1_pipe_gen3_x8_tx_serial,

    input  logic                               ing_egr_clock,
    output logic                               fvl_40g_bp
[NUM_AVST_IF_LINE],
    output logic [1:0]
mac2_10g_avalon_st_pause_data[NUM_AVST_IF_LINE],
    output logic [1:0]
mac_10g_avalon_st_pause_data [NUM_AVST_IF_LINE],

    input  logic
ing_in_clk[NUM_AVST_IF_LINE],
    input  logic
ing_in_rst[NUM_AVST_IF_LINE],
    input  logic [USR_ERROR_WIDTH-1:0]
ing_in_err[NUM_AVST_IF_LINE],
    input  logic
ing_in_val[NUM_AVST_IF_LINE],
    input  logic
ing_in_sop[NUM_AVST_IF_LINE],
    input  logic
ing_in_eop[NUM_AVST_IF_LINE],
    input  logic [2**LOG2_DP_DATA_PATH0_WIDTH - 1:0]
ing_in_dat[NUM_AVST_IF_LINE],
    input  logic [(LOG2_DP_DATA_PATH0_WIDTH-3) -1:0]
ing_in_mty[NUM_AVST_IF_LINE],
    output logic
ing_in_rdy[NUM_AVST_IF_LINE],
    output logic [(8/NUM_AVST_IF_LINE)-1:0]
ing_in_fpga_internal_pause_req[NUM_AVST_IF_LINE],
    input  logic [DP_CHA_WIDTH-1:0]
ing_in_cha[NUM_AVST_IF_LINE],
    input  logic [TIMESTAMP_WIDTH-1:0]
ing_in_timestamp_96b[NUM_AVST_IF_LINE],

    input  logic
ing_out_clk[NUM_AVST_IF_LINE],
    input  logic
ing_out_rst[NUM_AVST_IF_LINE],
    output logic
ing_out_sop[NUM_AVST_IF_LINE],
    output logic
ing_out_eop[NUM_AVST_IF_LINE],
    output logic
ing_out_val[NUM_AVST_IF_LINE],
    input  logic
ing_out_rdy[NUM_AVST_IF_LINE],
    input  logic [(8/NUM_AVST_IF_LINE)-1:0]
ing_out_fpga_internal_pause_req[NUM_AVST_IF_LINE],
    output logic [(LOG2_DP_DATA_PATH1_WIDTH-3) -1:0]
ing_out_mty[NUM_AVST_IF_LINE],
    output logic [2**LOG2_DP_DATA_PATH1_WIDTH - 1:0]
ing_out_dat[NUM_AVST_IF_LINE],
    output logic [USR_ERROR_WIDTH-1:0]
```

```
ing_out_err[NUM_AVST_IF_LINE],
    output logic [DP_CHA_WIDTH-1:0]
ing_out_cha[NUM_AVST_IF_LINE],
    output logic [TIMESTAMP_WIDTH-1:0]
ing_out_timestamp_96b[NUM_AVST_IF_LINE],

    input  logic
egr_in_clk[NUM_AVST_IF_FVL],
    input  logic
egr_in_rst[NUM_AVST_IF_FVL],
    input  logic [USR_ERROR_WIDTH-1:0]
egr_in_err[NUM_AVST_IF_FVL],
    input  logic
egr_in_val[NUM_AVST_IF_FVL],
    input  logic
egr_in_sop[NUM_AVST_IF_FVL],
    input  logic
egr_in_eop[NUM_AVST_IF_FVL],
    input  logic [2**LOG2_DP_DATA_PATH1_WIDTH - 1:0]
egr_in_dat[NUM_AVST_IF_FVL],
    input  logic [DP_CHA_WIDTH-1:0]
egr_in_cha[NUM_AVST_IF_FVL],                          // NB: not used
currently
    input  logic [(LOG2_DP_DATA_PATH1_WIDTH-3) -1:0]
egr_in_mty[NUM_AVST_IF_FVL],
    output logic
egr_in_rdy[NUM_AVST_IF_FVL],
    output logic [(8/NUM_AVST_IF_LINE)-1:0]
egr_in_fpga_internal_pause_req[NUM_AVST_IF_LINE],
    input  logic [TIMESTAMP_WIDTH-1:0]
egr_in_timestamp_96b[NUM_AVST_IF_LINE],

    input  logic
egr_out_clk[NUM_AVST_IF_FVL],
    input  logic
egr_out_rst[NUM_AVST_IF_FVL],
    output logic
egr_out_sop[NUM_AVST_IF_FVL],
    output logic
egr_out_eop[NUM_AVST_IF_FVL],
    output logic
egr_out_val[NUM_AVST_IF_FVL],
    input  logic
egr_out_rdy[NUM_AVST_IF_FVL],
    input  logic [(8/NUM_AVST_IF_LINE)-1:0]
egr_out_fpga_internal_pause_req[NUM_AVST_IF_LINE],
    output logic [DP_CHA_WIDTH-1:0]
egr_out_cha[NUM_AVST_IF_FVL],
    output logic [(LOG2_DP_DATA_PATH0_WIDTH-3) -1:0]
egr_out_mty[NUM_AVST_IF_FVL],
    output logic [2**LOG2_DP_DATA_PATH0_WIDTH - 1:0]
egr_out_dat[NUM_AVST_IF_FVL],
    output logic [USR_ERROR_WIDTH-1:0]
egr_out_err[NUM_AVST_IF_FVL],
    output logic [TIMESTAMP_WIDTH-1:0]
egr_out_timestamp_96b[NUM_AVST_IF_LINE],

`ifdef INCLUDE_DDR4
    input   wire                                ddr4a_avmm_0_clk,
    input   wire                                ddr4a_avmm_0_reset_n,
    input   wire                                ddr4a_avmm_0_waitrequest,
    input   wire [255:0]                        ddr4a_avmm_0_readdata,
    input   wire                                ddr4a_avmm_0_readdatavalid,
    output  wire [6:0]                          ddr4a_avmm_0_burstcount,
    output  wire [255:0]                        ddr4a_avmm_0_writedata,
    output  wire [25:0]                         ddr4a_avmm_0_address,
    output  wire                                ddr4a_avmm_0_write,
    output  wire                                ddr4a_avmm_0_read,
    output  wire [31:0]                         ddr4a_avmm_0_byteenable,

    input   wire                                ddr4a_avmm_1_clk,
```

```
    input   wire                                    ddr4a_avmm_1_reset_n,
    input   wire                                    ddr4a_avmm_1_waitrequest,
    input   wire [255:0]                            ddr4a_avmm_1_readdata,
    input   wire                                    ddr4a_avmm_1_readdatavalid,
    output  wire [6:0]                              ddr4a_avmm_1_burstcount,
    output  wire [255:0]                            ddr4a_avmm_1_writedata,
    output  wire [25:0]                             ddr4a_avmm_1_address,
    output  wire                                    ddr4a_avmm_1_write,
    output  wire                                    ddr4a_avmm_1_read,
    output  wire [31:0]                             ddr4a_avmm_1_byteenable,

    input   wire                                    ddr4b_avmm_0_clk,
    input   wire                                    ddr4b_avmm_0_reset_n,
    input   wire                                    ddr4b_avmm_0_waitrequest,
    input   wire [255:0]                            ddr4b_avmm_0_readdata,
    input   wire                                    ddr4b_avmm_0_readdatavalid,
    output  wire [6:0]                              ddr4b_avmm_0_burstcount,
    output  wire [255:0]                            ddr4b_avmm_0_writedata,
    output  wire [25:0]                             ddr4b_avmm_0_address,
    output  wire                                    ddr4b_avmm_0_write,
    output  wire                                    ddr4b_avmm_0_read,
    output  wire [31:0]                             ddr4b_avmm_0_byteenable,

    input   wire                                    ddr4b_avmm_1_clk,
    input   wire                                    ddr4b_avmm_1_reset_n,
    input   wire                                    ddr4b_avmm_1_waitrequest,
    input   wire [255:0]                            ddr4b_avmm_1_readdata,
    input   wire                                    ddr4b_avmm_1_readdatavalid,
    output  wire [6:0]                              ddr4b_avmm_1_burstcount,
    output  wire [255:0]                            ddr4b_avmm_1_writedata,
    output  wire [25:0]                             ddr4b_avmm_1_address,
    output  wire                                    ddr4b_avmm_1_write,
    output  wire                                    ddr4b_avmm_1_read,
    output  wire [31:0]                             ddr4b_avmm_1_byteenable,

    input wire                                      ddr4c_avmm_clk,
    input wire                                      ddr4c_avmm_reset_n,
    input wire                                      ddr4c_avmm_waitrequest,
    input wire [127:0]                              ddr4c_avmm_readdata,
    input wire                                      ddr4c_avmm_readdatavalid,
    output reg [6:0]                                ddr4c_avmm_burstcount,
    output reg [127:0]                              ddr4c_avmm_writedata,
    output reg [25:0]                               ddr4c_avmm_address,
    output reg                                      ddr4c_avmm_write,
    output reg                                      ddr4c_avmm_read,
    output reg [15:0]                               ddr4c_avmm_byteenable,

    input wire                                      qdr_avmm_clk,
    input wire                                      qdr_avmm_reset_n,
    input wire                                      qdr_avmm_waitrequest [7:0],
    input wire [35:0]                               qdr_avmm_readdata [7:0],
    input wire                                      qdr_avmm_readdatavalid
[7:0],
    output reg [2:0]                                qdr_avmm_burstcount [7:0],
    output reg [35:0]                               qdr_avmm_writedata [7:0],
    output reg [21:0]                               qdr_avmm_address [7:0],
    output reg                                      qdr_avmm_write [7:0],
    output reg                                      qdr_avmm_read [7:0]
`endif
);


localparam SUB_AVMM_NUM = 2;
localparam AFU_ID_L = LIGHTWEIGHT_MODE ? 64'h9FFA_8731_438C_4BA3 :
64'hC000_C966_0D82_4272;
localparam AFU_ID_H = LIGHTWEIGHT_MODE ? 64'hADDF_01F4_E8B1_D90F :
64'h9AEF_FE5F_8457_0612;

`default_nettype none
    localparam NUM_SUB_AFUS         = 8;
    localparam NUM_PIPE_STAGES      = 2;
```

```
    localparam C_SUB_AFUS_NOF_BITS  = $clog2(NUM_SUB_AFUS);


`ifdef INCLUDE_AFU_PCIE1
    pcie1_plug pcie1_plug_inst (
        .ffs_LP32ui_vl_sync_reset_n
(pcie1_pcie_pins_perst_n),    // synchronous to AVL clock
        .ffs_LP32ui_vl_sync_pwrgood_n       (pcie1_pcie_pins_perst_n),  //
not synchronous to AVL clock

        // PCIe pins
        .pin_pcie_ref_clk_p                 (pcie1_pipe_gen3_x8_ref_clk),
        .pin_pcie_in_perst_n                (pcie1_pcie_pins_perst_n),   //
connected to HIP
        .pin_pcie_rx_p                      (pcie1_pipe_gen3_x8_rx_serial),
        .pin_pcie_tx_p                      (pcie1_pipe_gen3_x8_tx_serial)
    );
`endif

/*     // ============================================================
    // Register SR <--> PR signals at interface before consuming it
    // ============================================================

    (* noprune *) logic [1:0]   pck_cp2af_pwrState_T1;
    (* noprune *) logic         pck_cp2af_error_T1;

    logic                       pck_cp2af_softReset_T1;
    t_if_ccip_Rx                pck_cp2af_sRx_T1;
    t_if_ccip_Tx                pck_af2cp_sTx_T0;

    // ============================================================
    // Register PR <--> PR signals near interface before consuming it
    // ============================================================

    ccip_interface_reg inst_green_ccip_interface_reg  (
        .pClk                           (pClk),
        .pck_cp2af_softReset_T0         (pck_cp2af_softReset),
        .pck_cp2af_pwrState_T0          (pck_cp2af_pwrState),
        .pck_cp2af_error_T0             (pck_cp2af_error),
        .pck_cp2af_sRx_T0               (pck_cp2af_sRx),
        .pck_af2cp_sTx_T0               (pck_af2cp_sTx_T0),

        .pck_cp2af_softReset_T1         (pck_cp2af_softReset_T1),
        .pck_cp2af_pwrState_T1          (pck_cp2af_pwrState_T1),
        .pck_cp2af_error_T1             (pck_cp2af_error_T1),
        .pck_cp2af_sRx_T1               (pck_cp2af_sRx_T1),
        .pck_af2cp_sTx_T1               (pck_af2cp_sTx)
    ); */

    //split c0rx into host and mmio
    wire afu_clk;
    assign afu_clk = pClk ;
    t_if_ccip_Rx pck_cp2af_mmio_sRx;
    t_if_ccip_Rx pck_cp2af_host_sRx;
    always_comb
    begin
        pck_cp2af_mmio_sRx = pck_cp2af_sRx;
        pck_cp2af_host_sRx = pck_cp2af_sRx;
        //disable rsp valid on mmio path
        pck_cp2af_mmio_sRx.c0.rspValid = 0;
        //disable mmio valid on host path
        pck_cp2af_host_sRx.c0.mmioRdValid = 0;
        pck_cp2af_host_sRx.c0.mmioWrValid = 0;
    end

    // ========================================================================
    //
    //  Instantiate a memory properties factory (MPF) between the external
    //  interface and the AFU, adding support for virtual memory and
    //  control over memory ordering.
    //
```

```
    // ===========================================================================

    //
    // The AFU exposes the primary AFU device feature header (DFH) at MMIO
    // address 0.  MPF defines a set of its own DFHs.  The AFU must
    // build its feature chain to point to the MPF chain.  The AFU must
    // also tell the MPF module the MMIO address at which MPF should start
    // its feature chain.
    //
    //Note: with ENABLE_SEPARATE_MMIO_FIFO, MPF will not receive or forward
    //any mmio requests
    localparam MPF_DFH_MMIO_ADDR = 'h0000;
    localparam MPF_DFH_MMIO_NEXT_ADDR = 'h0000;

    //
    // MPF represents CCI as a SystemVerilog interface, derived from the
    // same basic types defined in ccip_if_pkg.  Interfaces reduce the
    // number of internal MPF module parameters, since each internal MPF
    // shim has a bus connected toward the AFU and a bus connected toward
    // the FIU.
    //

    //
    // Expose FIU as an MPF interface
    //
    cci_mpf_if fiu(.clk(afu_clk));

    // The CCI wires to MPF mapping connections have identical naming to
    // the standard AFU.  The module exports an interface named "fiu".
    ccip_wires_to_mpf
      #(
        // All inputs and outputs in PR region (AFU) must be registered!
        .REGISTER_INPUTS(1),
        .REGISTER_OUTPUTS(1)
        )
      map_ifc
       (
        .pClk(afu_clk),
        .pck_cp2af_softReset(pck_cp2af_softReset),
        .pck_cp2af_sRx(pck_cp2af_host_sRx),
        .pck_af2cp_sTx(pck_af2cp_sTx),
        .*
        );

    //
    // Instantiate MPF with the desired properties.
    //
    cci_mpf_if afu(.clk(afu_clk));

    cci_mpf
      #(
        // Should read responses be returned in the same order that
        // the reads were requested?
        .SORT_READ_RESPONSES(1),

        // Should the Mdata from write requests be returned in write
        // responses?  If the AFU is simply counting write responses
        // and isn't consuming Mdata, then setting this to 0 eliminates
        // the memory and logic inside MPF for preserving Mdata.
        .PRESERVE_WRITE_MDATA(0),

        // Enable virtual to physical translation?  When enabled, MPF
        // accepts requests with either virtual or physical addresses.
        // Virtual addresses are indicated by setting the
        // addrIsVirtual flag in the MPF extended Tx channel
        // request header.
        .ENABLE_VTP(0),

        // Enable mapping of eVC_VA to physical channels?  AFUs that both
use
        // eVC_VA and read back memory locations written by the AFU must
```

```
either
        // emit WrFence on VA or use explicit physical channels and enforce
        // write/read order.  Each method has tradeoffs.  WrFence VA is
expensive
        // and should be emitted only infrequently.  Memory requests to
eVC_VA
        // may have higher bandwidth than explicit mapping.  The MPF module
for
        // physical channel mapping is optimized for each CCI platform.
        //
        // If you set ENFORCE_WR_ORDER below you probably also want to set
        // ENABLE_VC_MAP.
        //
        // The mapVAtoPhysChannel extended header bit must be set on each
        // request to enable mapping.
        .ENABLE_VC_MAP(0),
        // When ENABLE_VC_MAP is set the mapping is either static for the
entire
        // run or dynamic, changing in response to traffic patterns.  The
mapper
        // guarantees synchronization when the mapping changes by emitting a
        // WrFence on eVC_VA and draining all reads.  Ignored when
ENABLE_VC_MAP
        // is 0.
        .ENABLE_DYNAMIC_VC_MAPPING(0),

        // Should write/write and write/read ordering within a cache
        // be enforced?  By default CCI makes no guarantees on the order
        // in which operations to the same cache line return.  Setting
        // this to 1 adds logic to filter reads and writes to ensure
        // that writes retire in order and the reads correspond to the
        // most recent write.
        //
        // ***  Even when set to 1, MPF guarantees order only within
        // ***  a given virtual channel.  There is no guarantee of
        // ***  order across virtual channels and no guarantee when
        // ***  using eVC_VA, since it spreads requests across all
        // ***  channels.  Synchronizing writes across virtual channels
        // ***  can be accomplished only by requesting a write fence on
        // ***  eVC_VA.  Syncronizing writes across virtual channels
        // ***  and then reading back the same data requires both
        // ***  requesting a write fence on eVC_VA and waiting for the
        // ***  corresponding write fence response.
        //
        .ENFORCE_WR_ORDER(0),

        // Enable partial write emulation.  CCI has no support for masked
        // writes that merge new data with existing data in a line.  MPF
        // adds byte-level masks to the write request header and emulates
        // partial writes as a read-modify-write operation.  When coupled
        // with ENFORCE_WR_ORDER, partial writes are free of races on the
        // FPGA side.  There are no guarantees of atomicity and there is
        // no protection against races with CPU-generates writes.
        .ENABLE_PARTIAL_WRITES(0),

        // Address of the MPF feature header.  See comment above.
        .DFH_MMIO_BASE_ADDR(MPF_DFH_MMIO_ADDR),
        .DFH_MMIO_NEXT_ADDR(MPF_DFH_MMIO_NEXT_ADDR)
        )
      mpf
       (
        .clk(afu_clk),
        .fiu,
        .afu,
        .c0NotEmpty(),
        .c1NotEmpty()
        );


    // ========================================================================
    //
```

```
    //  Now CCI is exposed as an MPF interface through the object named
    //  "afu".  Two primary strategies are available for connecting
    //  a design to the interface:
    //
    //    (1) Use the MPF-provided constructor functions to generate
    //        CCI request structures and pass them directly to MPF.
    //        See, for example, cci_mpf_defaultReqHdrParams() and
    //        cci_c0_genReqHdr() in cci_mpf_if_pkg.sv.
    //
    //    (1) Map "afu" back to standard CCI wires.  This is the strategy
    //        used below to map an existing AFU to MPF.
    //
    // ========================================================================

    //
    // Convert MPF interfaces back to the standard CCI structures.
    //
    t_if_ccip_Rx mpf2af_sRxPort;
    t_if_ccip_Tx af2mpf_sTxPort;

    //
    // The cci_mpf module has already registered the Rx wires heading
    // toward the AFU, so wires are acceptable.
    //
    always_comb
    begin
        mpf2af_sRxPort.c0 = afu.c0Rx;
        mpf2af_sRxPort.c1 = afu.c1Rx;

        mpf2af_sRxPort.c0TxAlmFull = afu.c0TxAlmFull;
        mpf2af_sRxPort.c1TxAlmFull = afu.c1TxAlmFull;

        afu.c0Tx = cci_mpf_cvtC0TxFromBase(af2mpf_sTxPort.c0);
        if (cci_mpf_c0TxIsReadReq(afu.c0Tx))
        begin
            // Treat all addresses as virtual.
            afu.c0Tx.hdr.ext.addrIsVirtual = 1'b0;

            // Enable eVC_VA to physical channel mapping.  This will only
            // be triggered when ENABLE_VC_MAP is set above.
            afu.c0Tx.hdr.ext.mapVAtoPhysChannel = 1'b0;

            // Enforce load/store and store/store ordering within lines.
            // This will only be triggered when ENFORCE_WR_ORDER is set.
            afu.c0Tx.hdr.ext.checkLoadStoreOrder = 1'b0;
        end

        afu.c1Tx = cci_mpf_cvtC1TxFromBase(af2mpf_sTxPort.c1);
        if (cci_mpf_c1TxIsWriteReq(afu.c1Tx))
        begin
            // Treat all addresses as virtual.
            afu.c1Tx.hdr.ext.addrIsVirtual = 1'b0;

            // Enable eVC_VA to physical channel mapping.  This will only
            // be triggered when ENABLE_VC_MAP is set above.
            afu.c1Tx.hdr.ext.mapVAtoPhysChannel = 1'b0;

            // Enforce load/store and store/store ordering within lines.
            // This will only be triggered when ENFORCE_WR_ORDER is set.
            afu.c1Tx.hdr.ext.checkLoadStoreOrder = 1'b0;
        end

        afu.c2Tx = af2mpf_sTxPort.c2;
    end

//
// ========================================================================
// ==================
// User AFU goes here
//
// ========================================================================
```

```
==================

afu afu_inst(
    .afu_clk(afu_clk),

    .reset           ( fiu.reset ) ,
    .cp2af_sRxPort       ( mpf2af_sRxPort ) ,
    .cp2af_mmio_c0rx     ( pck_cp2af_mmio_sRx.c0 ) ,
    .af2cp_sTxPort       ( af2mpf_sTxPort )
);
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//
//            USER LOGIC - ADD/REMOVE/MODIFY your logic below
//

endmodule
```

3. Copy attached `afu.qsf` file into the design.
4. Compile design using make flow. **Note:** You need to set items to remove diagnostics and external memories.

```
$ make 2x2x25G GUI=1 INCLUDE_DIAGNOSTICS=0 INCLUDE_MEMORY=0 \
PAC_VER_MAJOR=3 PAC_VER_MINOR=5 PAC_VER_PATCH=6 \
REVISION_ID=12345678 INCLUDE_AFU_PCIE1=0 AFU_ROOT=<absolute path>/hw/
hls_afu/hw
```

Note: The compile process takes approximately 1.5 hours.

4.5.3.4. Verifying Timing Constraints are Satisfied

When the compile is complete, verify timing constraints are satisfied. You can verify this in the GUI using Timing Analyzer or you can review the generated report file in `prj/pac_baseline/build/chip.sta.summary`.

1. Go to the `$ cd prj/pac_baseline/build` directory.
2. Review `chip.sta.summary` for timing constraints with negative slack
3. Create an unsigned FPGA image file for loading into flash. This instruction assumes the board root key has not been programmed.

```
$ PACSign SR -t UPDATE -H openssl_manager -i pac-n3000-secure-update-raw.bin
\
-o unsigned_PAC_N3000_RSU.bin
```

Load FPGA image:

```
$ sudo fpgasupdate unsigned_PAC_N3000_RSU.bin  <N3000 PCIe B:D.F>
>>Please note this command takes ~40 minutes to complete
$ sudo rsu fpga <N3000 PCIe B:D.F>
```

Load host application and run with FPGA:

a. Change directory to software:

```
$ cd hls_example/hw/hls_afu/sw
$ make
```

b. Set hugepage:

```
# echo 200 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

c. Run the application:

```
$ sudo ./hls_afu_host
```

Using Avalon Slave at offset 0x40. No vector size specified. Default to size 64 floats. run `./hls_afu_host <vectorsize>` to specify a vector size at runtime using test vector of size 64.

d. Running Test:

```
AFU DFH REG = 1000010000000000
AFU ID LO = 944028430b016f3d
AFU ID HI = 5fa7fd4b867c484c
AFU NEXT = 00000000
AFU RESERVED = 00000000
end of output memory before executing kernel:
    [62] - -6259853398707798016.000000 (0xdeadbeef)
    [63] - -6259853398707798016.000000 (0xdeadbeef)
    [64] - -6259853398707798016.000000 (0xdeadbeef)
    [65] - 0.000000 (0x0)
Interrupt enabled = 00000000
Interrupt enabled = 00000001
AFU Latency: 0.04500 milliseconds
Poll success. Return = 1
check output memory:
output memory OK!
sum: Expected 715.000000, calculated 715.000000
```

The FPGA writes a full 512-bit word (64 bytes) to host memory, so if the size of your test vector (in bytes) is not a multiple of 64, the FPGA will overwrite some space at the end of output memory. `fpgaPrepareBuffer()` allocates your host memory in a buffer that is a multiple of 64 bytes, so the FPGA behavior will not affect your application. You should expect to see a single `0xdeadbeef` at the end of the output memory if and only if the size of your test vector (determined by `vector_size`, and the data type) is a multiple of 64 bytes (that is, if `vector_size` is a multiple of 16).

5. Capturing Signals in AFU with Signal Tap

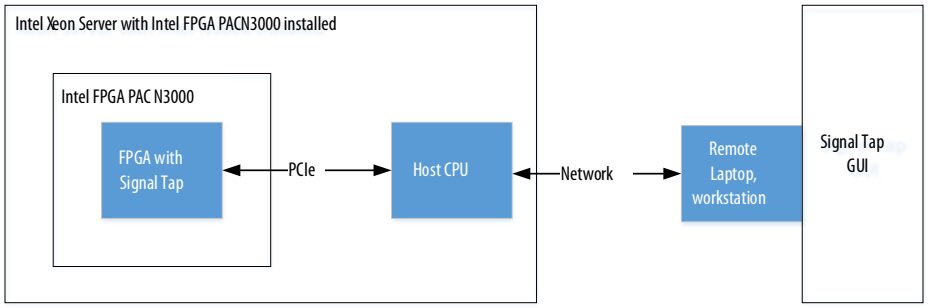
This section is a short guide on adding remote Signal Tap instances to an AFU for in system debugging. You can follow the steps in the following sections, in order of execution to create an instrumented AFU. The `hello_afu` project is used in this guide as the target AFU to be instrumented.

You need a basic understanding of Signal Tap. Please see the "Signal Tap Logic Analyzer: Introduction & Getting Started" Web Based Training for more information.

There are two ways you can run the Signal Tap GUI:

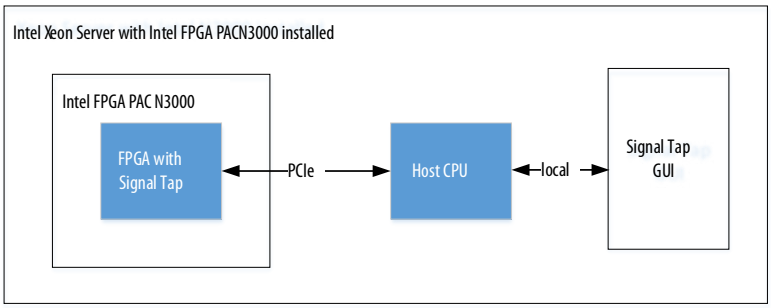
- On a remote laptop or workstation connected through the network to a server with the N3000 with a Signal Tap instrumented FPGA image.

Figure 42. Remote Debugging Scenarios



- With a Signal Tap GUI running locally on the server with the N3000.

Figure 43. Local Debugging Scenarios



Now that you have compiled the `hello_afu` design, you can proceed as follows for adding Signal Tap to the design.

5.1. Adding Signal Tap to the Design

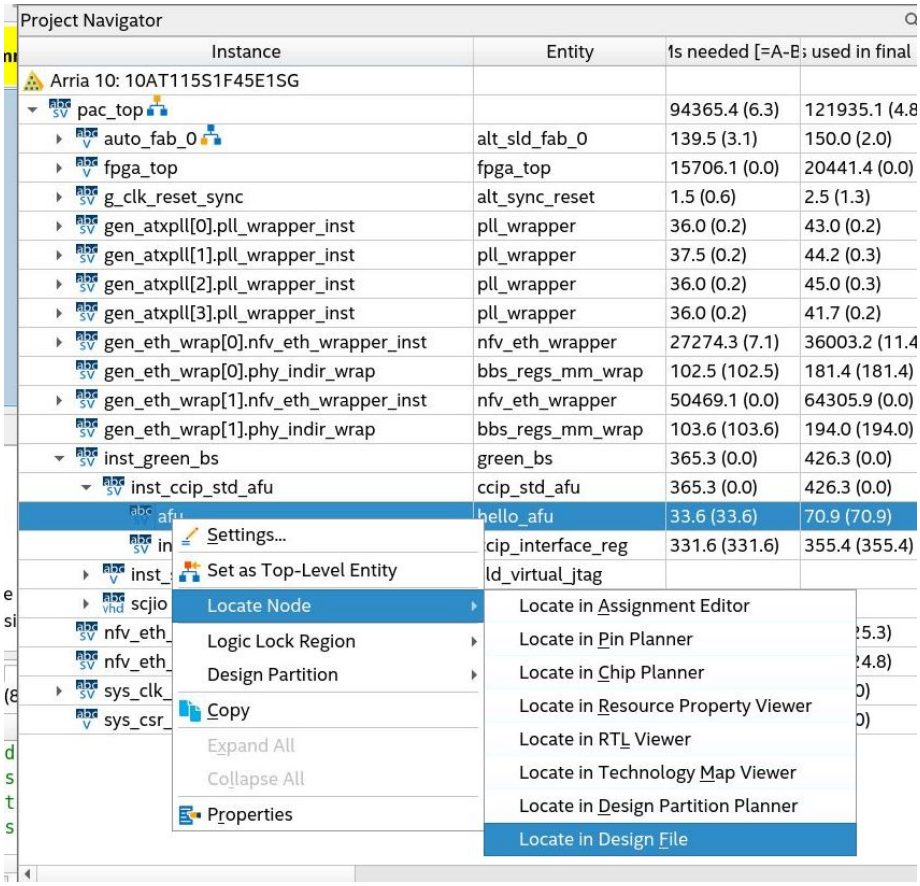
After you have compiled the `hello_afu` design, you can proceed as follows for adding Signal Tap to the design.

1. Familiarize yourself with the project hierarchy. If your Project Navigator is not already in view in your main Intel Quartus Prime window, then in the Intel Quartus Prime window select **View ➤ Project Navigator**. Detach the Project Navigator pane to expand its view. Expand the `pac_top` instance. Now expand the `inst_green_bs` instance. Next expand `inst_ccip_std_afu`. Your Project Navigator display should look as shown below:

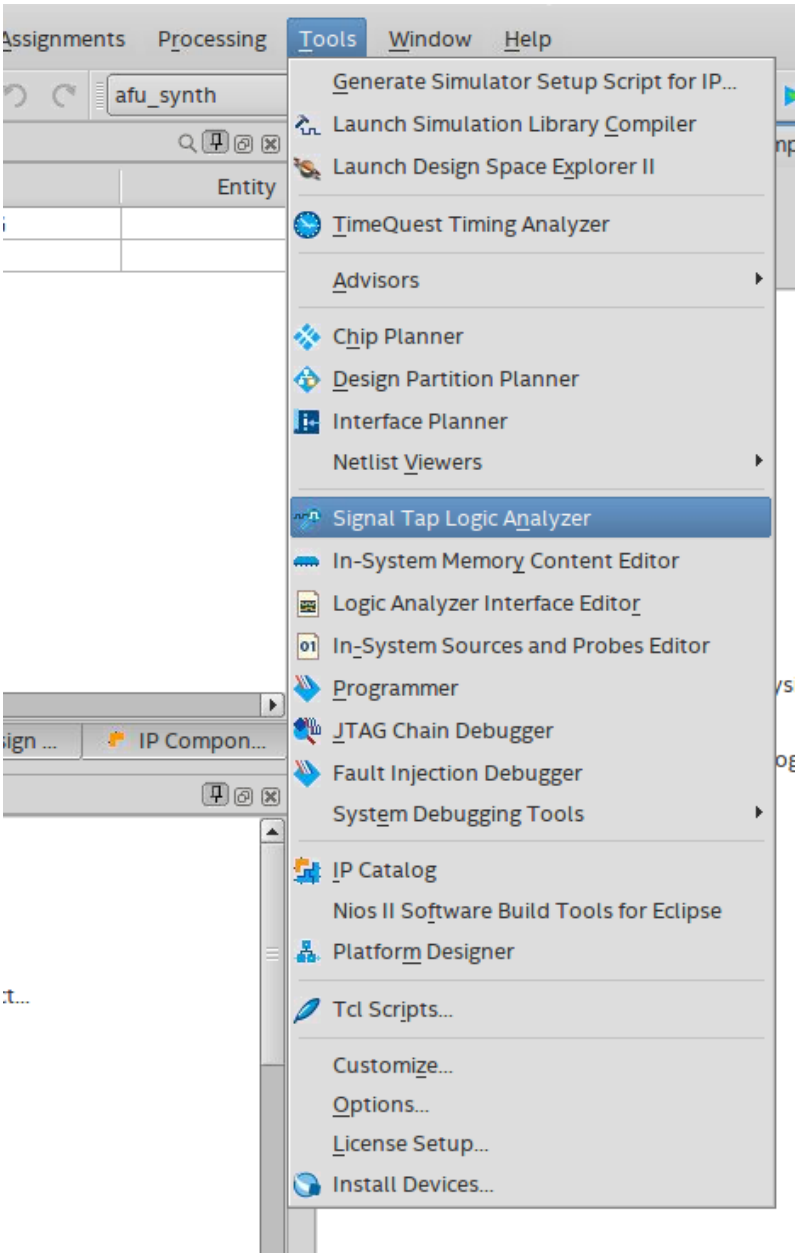
Project Navigator

Instance	Entity	1
Arria 10: 10AT115S1F45E1SG		
pac_top		9
auto_fab_0	alt_sld_fab_0	1
fpga_top	fpga_top	1
g_clk_reset_sync	alt_sync_reset	1
gen_atxpll[0].pll_wrapper_inst	pll_wrapper	3
gen_atxpll[1].pll_wrapper_inst	pll_wrapper	3
gen_atxpll[2].pll_wrapper_inst	pll_wrapper	3
gen_atxpll[3].pll_wrapper_inst	pll_wrapper	3
gen_eth_wrap[0].nfv_eth_wrapper_inst	nfv_eth_wrapper	2
gen_eth_wrap[0].phy_indir_wrap	bbs_regs_mm_wrap	1
gen_eth_wrap[1].nfv_eth_wrapper_inst	nfv_eth_wrapper	5
gen_eth_wrap[1].phy_indir_wrap	bbs_regs_mm_wrap	1
inst_green_bs	green_bs	3
inst_ccip_std_afu	ccip_std_afu	3
afu	hello_afu	3
inst_green_ccip_interface_reg	ccip_interface_reg	3
inst_sld_virtual_jtag	sld_virtual_jtag	
scjio	altera_sld_host_end...	
nfv_eth_led_0_inst	nfv_eth_led	2
nfv_eth_led_1_inst	nfv_eth_led	2
sys_clk_reset_sync	nfv_reset_sync	0
sys_csr_clk_pll	altera_pll	0

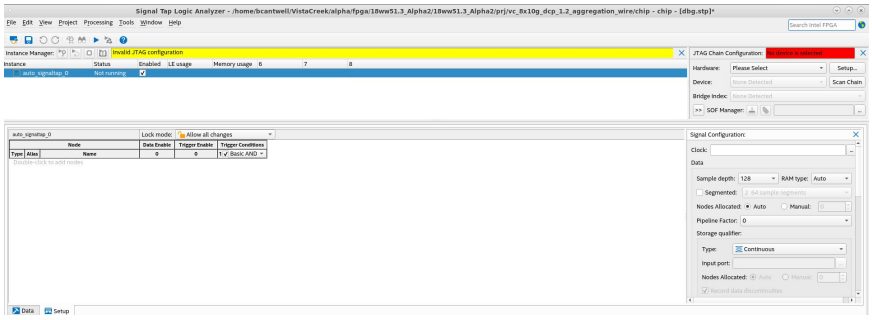
2. For this learning tutorial, the CCI-P interface and scratch register is instrumented. Select the **afu** instance in Project Navigator and right click, then select **Locate Node** and finally, **Locate in Design File**. See screen shot below:



3. This brings up the `hello_afu.sv` SystemVerilog code. Having the code available for review while defining the signals to be instrumented is highly useful.
4. Open the Signal Tap tool to create a `*.stp` file defining the signals to be instrumented. See example screen shot below on how to open the Signal Tap tool:

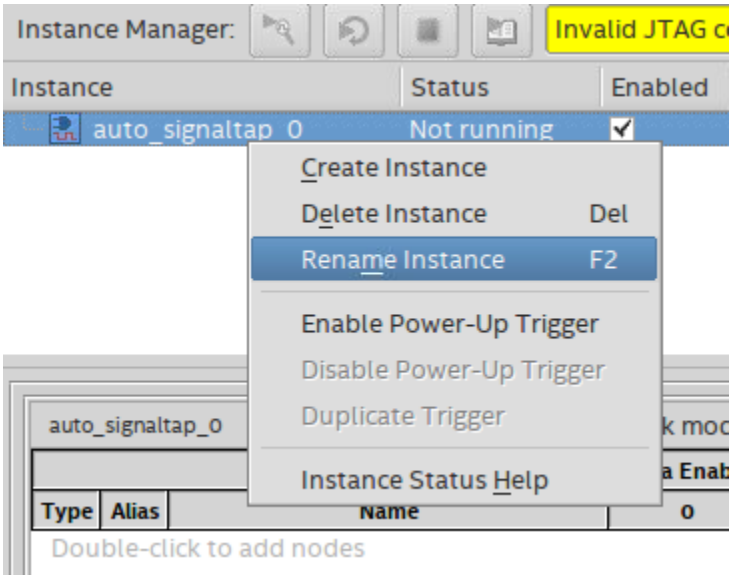


a. The Signal Tap GUI appears as shown below:

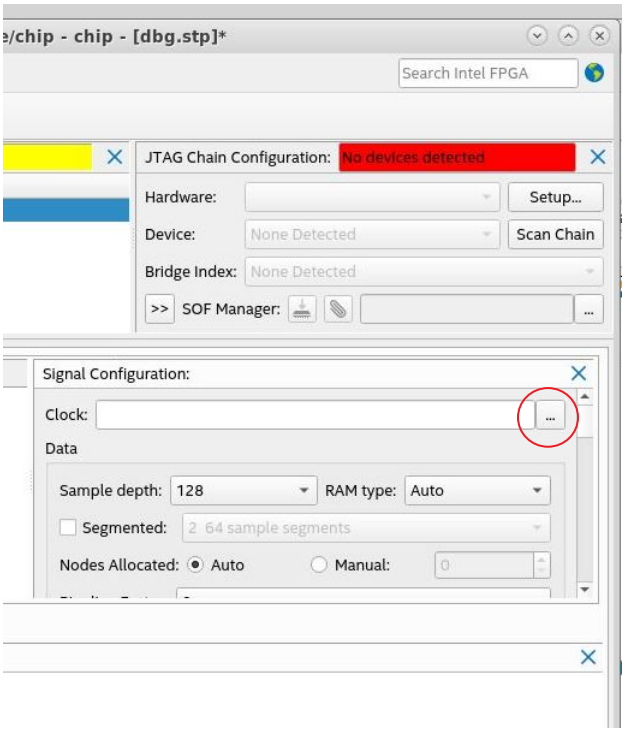


b. You can now set up a Signal Tap instance to instrument a portion of the design for observability.

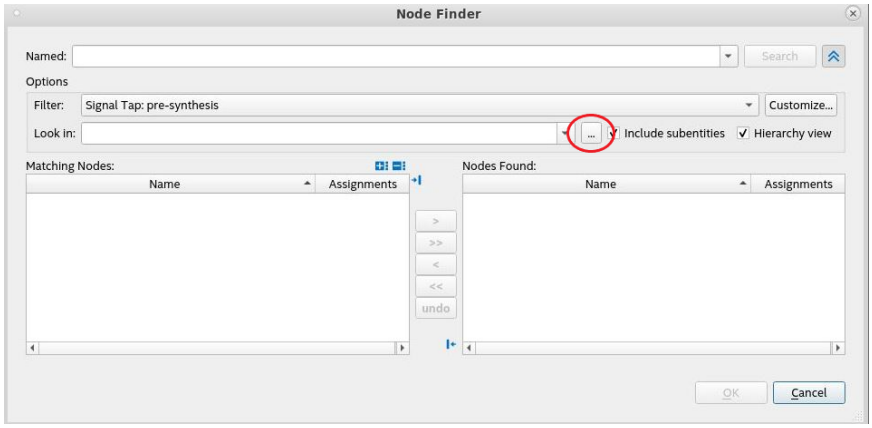
   In this example, the Signal Tap instance is renamed to `hello_afu` to indicate its use to instrument the `hello_afu` module. To rename the `auto_signaltap_0` instance, right click and select **Rename Instance**:



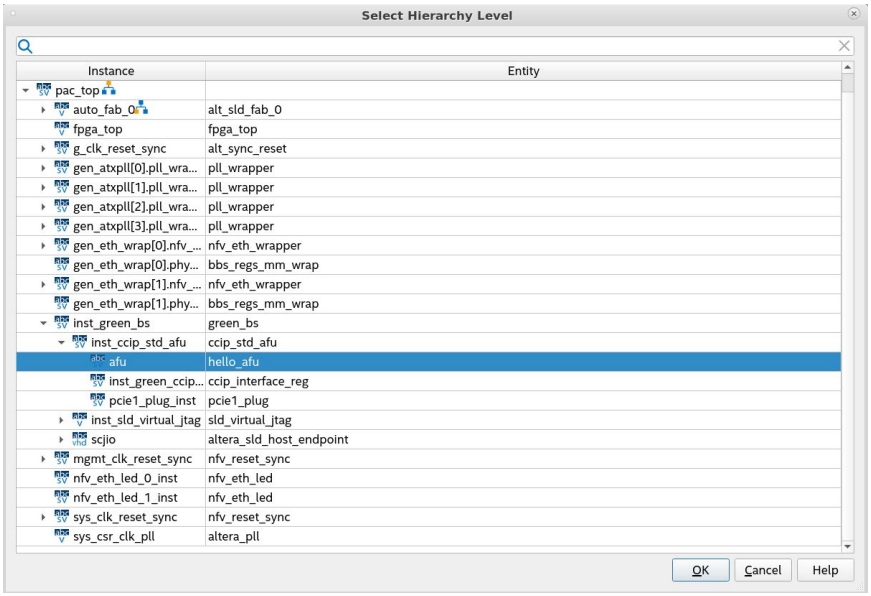
5. For the `hello_afu` Signal Tap instance define the clock used to sample the signals to be instrumented. For accurate results, the instrumented signals must be in the domain of this clock. To select the clock, select the **...** button under "Signal Configuration":



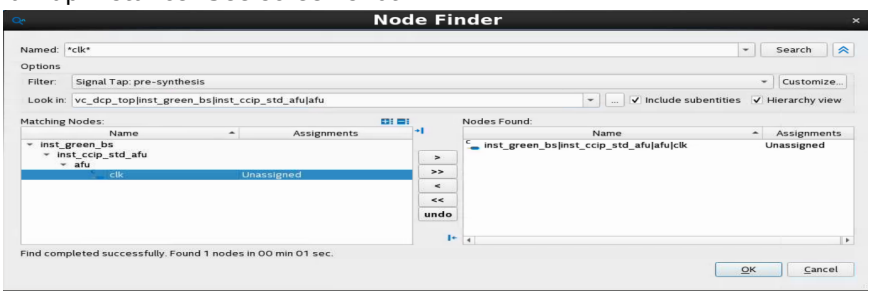
a. This brings up the Node Finder tool. Find the "Look in:" listing go to the right and click **...** to bring up the "Select Hierarchy Level" viewer. See screen shot:



b. In the Select Hierarchy Level viewer expand `pac_top`, `inst_green_bs`, `inst_ccip_std_afu` and select **afu** and click **OK**. See screen shot:

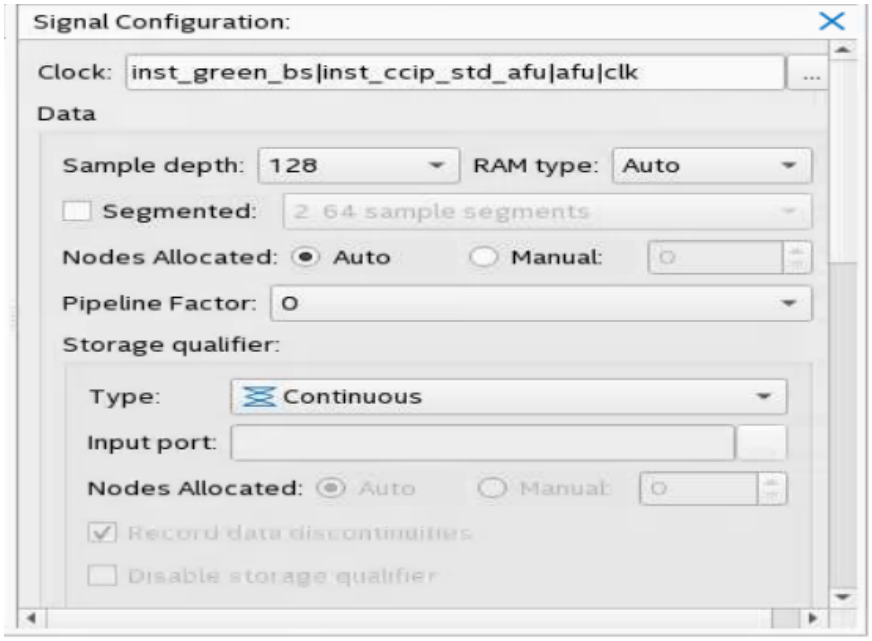


c. From reviewing the `hello_afu.sv` code, notice all signals are synchronous to signal **clk**. In the Node Finder window, type in ***clk*** in the "Named" blank and click **Search**. Expand each instance selection in the "Matching Nodes" pane. Then select **clk** and the **>** to make this the signal used for clocking the Signal Tap instance. See screen shot:

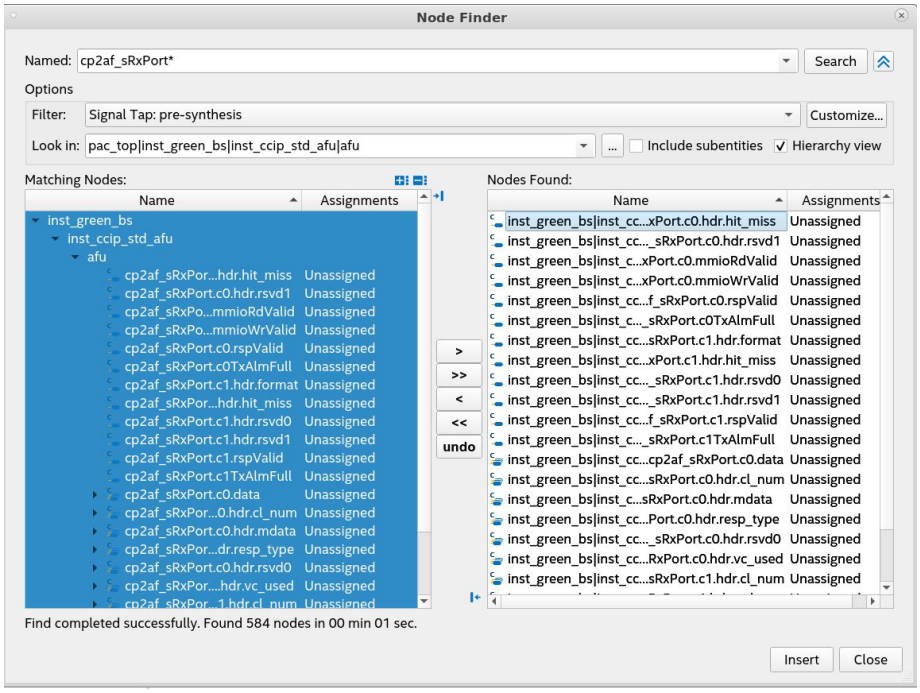


i. Press **OK**

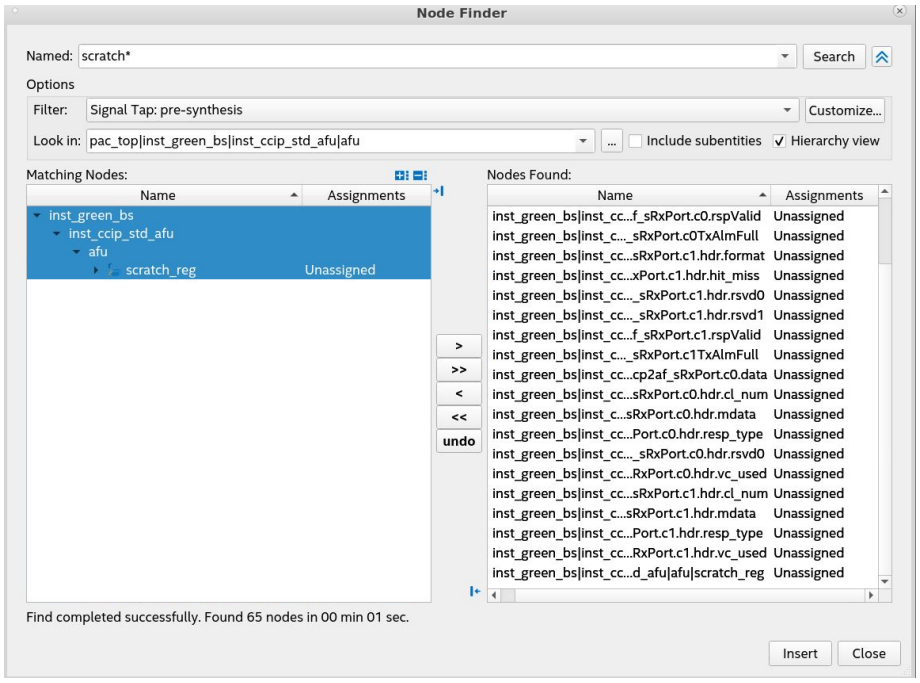
ii. The Signal Configuration dock on the far right should look as shown below:



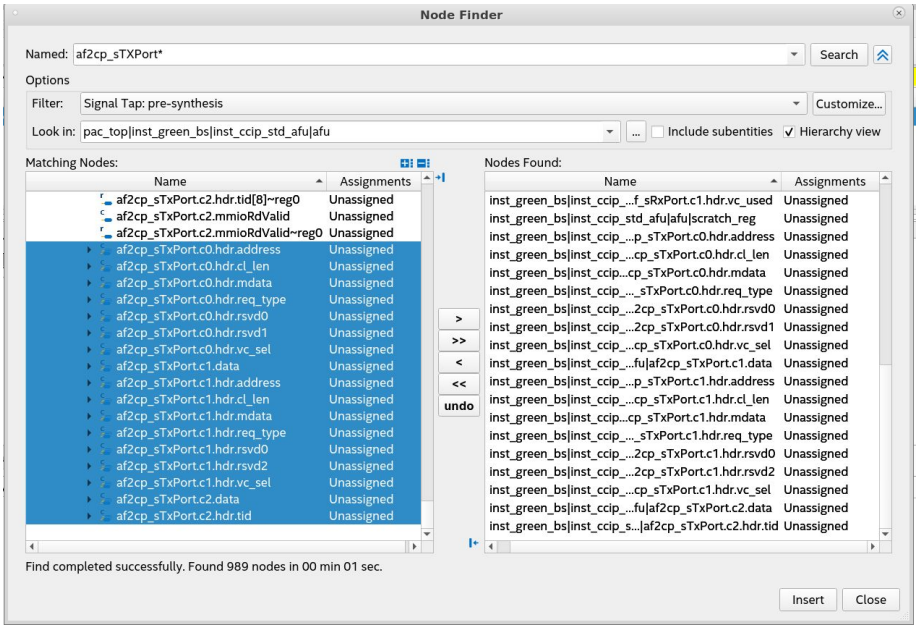
6. You can increase the depth of the samples captured by increasing **Sample depth**. For this example, the depth is increased to 1 K. Please keep in mind, increased depth means more FPGA resources used for the Signal Tap instance.
7. Select the signals to be added to the `hello_afu` Signal Tap instance by double clicking the **Double-click to add nodes** area of the "Setup" dock. This brings up the Node Finder tool. Repeat the steps above to set the "Look in:" box to narrow the search to just the `hello_afu` instance. You want to see the CCI-P interface signals and scratch pad register. Enter **cp2af_sRxPort*** in the "Named" entry field and click **Search**. Click the **>>** to select all signals to give you total visibility of the `ccip` RX bus input to the `hello_afu`. See screen shot below:



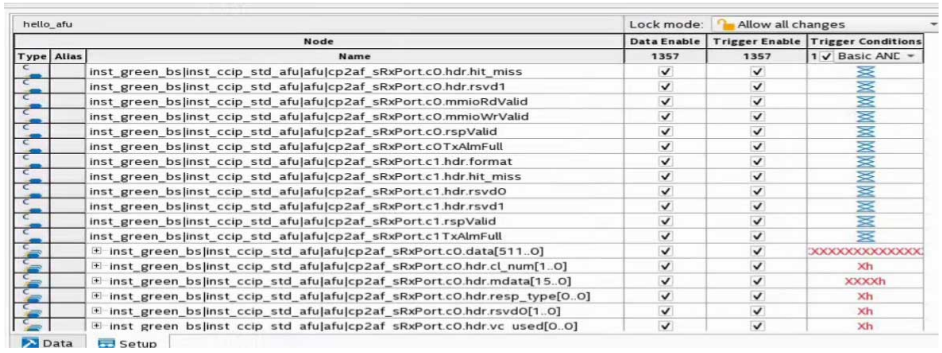
8. Now change "Named:" to **scratch*** and click **Search**. Expand Matching Nodes until the `scratch_reg` is displayed. Select **scratch_reg** and then **>>** to add the scratch register signals to the "Nodes Found" list. See screen shot:



9. Now enter **af2cp_sTxPort*** in the "Named" entry field and click **Search** to instrument the output CCI-P bus signals. Expand the instance names in "Matching Nodes". Do not select names ending in `~reg0`. Your display should be as shown below:



10. Click **Insert** and **Close**. Your display should be as shown below:



11. Select **File ➤ Save As** and save the newly created Signal Tap Logic Analyzer file as `hello_afu.stp` in the current directory.
12. When asked: "Do you want to enable Signal Tap File `hello_afu.stp` for the current project?" Click **Yes**.
13. Close the Signal Tap GUI.
14. Re-Run a full compilation to create a new FPGA implementation with `hello_afu` Signal Tap instrumentation included.
15. Once the build completes, the newly created `pac-n3000-secure-update-unsigned.bin` with Signal Tap instrumentation can be loaded into FPGA Flash for storage and use as the user image loaded into the Intel Arria 10 GT FPGA.

5.2. Loading FPGA Image

1. Change directories to the build directory where `pac-n3000-secure-update-unsigned.bin` was created by make and create an unsigned FPGA image using PACSign. Please note, this step assumes your target N3000 board has not had the root key programmed on the board.

```
$ cd prj/pac_baseline/build
```

2. Before running PACSign, ensure you have the following environment setting:

```
export PYTHONPATH=/usr/local/lib/python3.6/site-packages/
```

3. Create the image:

```
$ PACSign SR -t UPDATE -H openssl_manager  \
-i pac-n3000-secure-update-raw.bin  -o hello_afu_unsigned.bin
No root key specified.  Generate unsigned bitstream? Y = yes, N = no: y
No CSK specified.  Generate unsigned bitstream? Y = yes, N = no: y
```

   By responding with 'y', you are creating an unsigned binary file that can be loaded into a N3000 board that has not had the root key hash loaded into flash.

4. Flash your N3000 image with this new file.

   Note: This command must be done as sudo or root.

```
$ sudo fpgasupdate hello_afu_unsigned.bin 3e:00.0
```

   Note: Your PCIe b:d.f value can be different from 3e:00.0 used above.

   This command takes about 40 minutes to complete.

5. Perform a remote system update to load the new FPGA image using your PCIe B.D.f.

```
$ sudo rsu fpga 3e:00.0
```

6. Verify the `hello_afu_stp` FPGA image is loaded with the `fpgainfo port` command showing the Accelerator Id as `850adcc2-6ceb-4b22-9722-d43375b61c66`.

```
# fpgainfo port
Board Management Controller, MAX10 NIOS FW version D.2.0.19
Board Management Controller, MAX10 Build version D.2.0.6
//****** PORT ******//
Object Id : 0xED00000
PCIe s:b:d.f : 0000:3e:00.0
Device Id : 0x0b30
Numa Node : 1
Ports Num : 01
Bitstream Id : 0x21000410030509
Bitstream Version : 0.2.3
Pr Interface Id : a5d72a3c-c8b0-4939-912c-f715e5dc10ca
Accelerator Id : 850adcc2-6ceb-4b22-9722-d43375b61c66
```

5.3. Set Up Connections

1. Set up a network connection between the instrumented FPGA AFU and your Signal Tap GUI. You can set the network connection based on whether you are using a remote or local debugging configuration. See diagram below illustrating the connection steps:

Figure 44. Remote Debugging

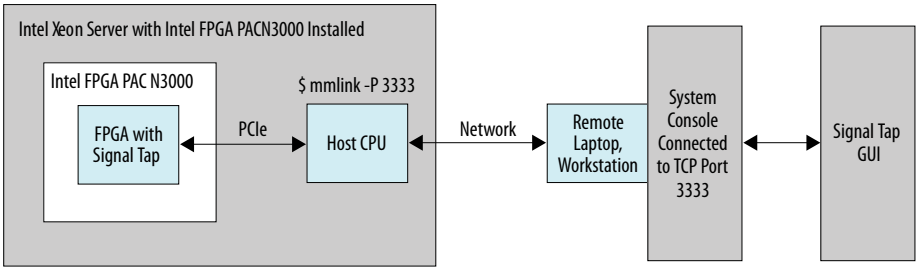
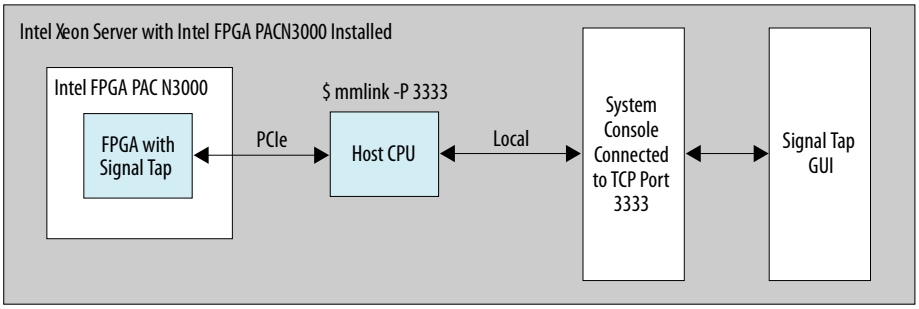


Figure 45. Local Debugging



2. Use the OPAE tool **mmlink** to enable your host system for remote Signal Tap. This tool is included with the Intel Acceleration Stack for the N3000
3. Open a TCP port to accept incoming connection requests from remote debug hosts.

```
# mmlink -P 3333 –B 0xb2 Note, -B is the bus number of the target N3000 to
connect
 ------- Command line Input START ----
 Segment : -1
 Bus : 02
 Device : -1
 Function : -1
 Socket-id : -1
 Port : 3333
 IP address : 0.0.0.0
 ------- Command line Input END ----
PORT Resource found.
Remote STP : Assert Reset
Remote STP : De-Assert Reset
Read signature value 53797343 to hw
Read version value 1 to hw
Read write fifo capacity value 32 to hw
m_listen: 4
listening on ip: 0.0.0.0; port: 3333
```

You can debug remotely from a remote machine connected to your PAC host or you can debug on the local PAC host.

— If debugging on a remote host:

a. Make sure Intel Quartus Prime Pro Edition version 19.2 is installed and the directory `$N3000_EXAMPLE_ROOT/hello_afu/hw/pac/remote_debug` is copied to the remote host.

b. Use System Console on the remote host to connect to the debug target host IP and TCP port using the following command:

```
$ cd Remote host directory with $N3000_EXAMPLE_ROOT/
hello_afu/hw/pac/\
remote_debug
$ system-console --rc_script=mmlink_setup_profiled.tcl \
remote_debug.sof <IP Address of target PAC host> 3333
```

Note: You must have the System Console executable binary added to your PATH variable. The System Console executable binary is in the Intel Quartus Prime installation directory. An example of how to update your PATH variable is the following:

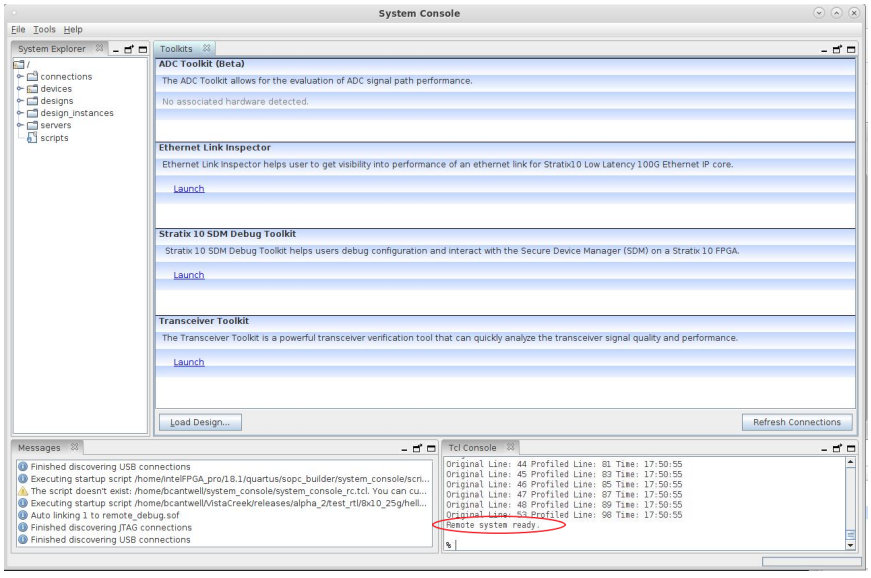
```
export PATH=$PATH:~/inteldevstack/intelFPGA_pro/quartus/
sopc_builder/bin
```

— If debugging on local host:

a. Start System Console on the local host as shown below:

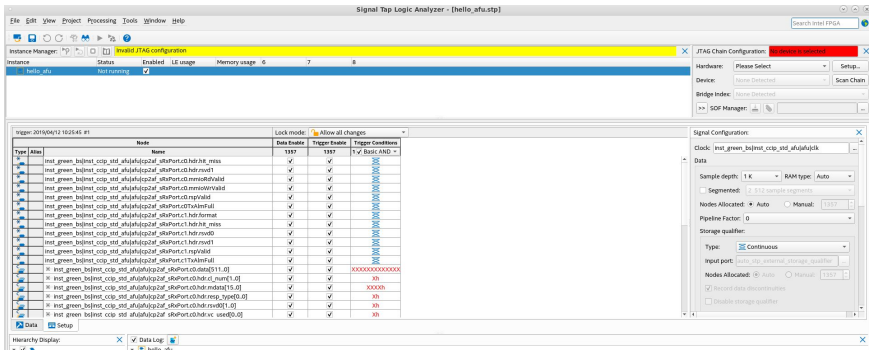
```
$ # cd $N3000_EXAMPLE_ROOT/hello_afu/hw\
/pac/remote_debug
$ system-console --rc_script=mmlink_setup_profiled.tcl \
remote_debug.sof localhost 3333
```

Whether local or remote, the Intel Quartus Prime tool System Console starts a new GUI and runs the `mmlink_setup_profiled.tcl` setup script as shown below:

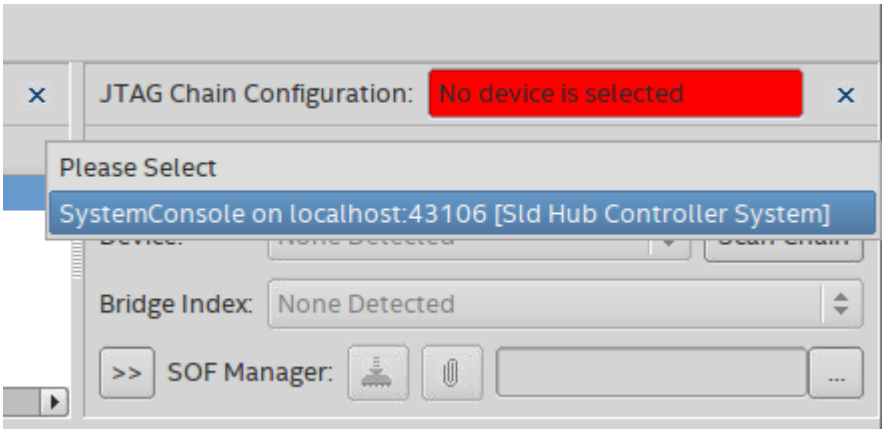


The script takes 1-2 minutes to complete. When the "Remote system ready" message is displayed Signal Tap may be connected for debugging.

b. Open Intel Quartus Prime GUI on the machine that is performing the debugging (i.e. local or remote host). In Intel Quartus Prime GUI, select **File ➤ Open** and navigate to the `hello_afu.stp` file created in previous steps, select and open this file. The Signal Tap GUI comes up as shown below:



c. In the Signal Tap GUI top right corner, "Hardware" pull down, where it says "Please Select", click on the up/down arrows to bring up the hardware selection – select the choice that has **System Console ....** as shown below:



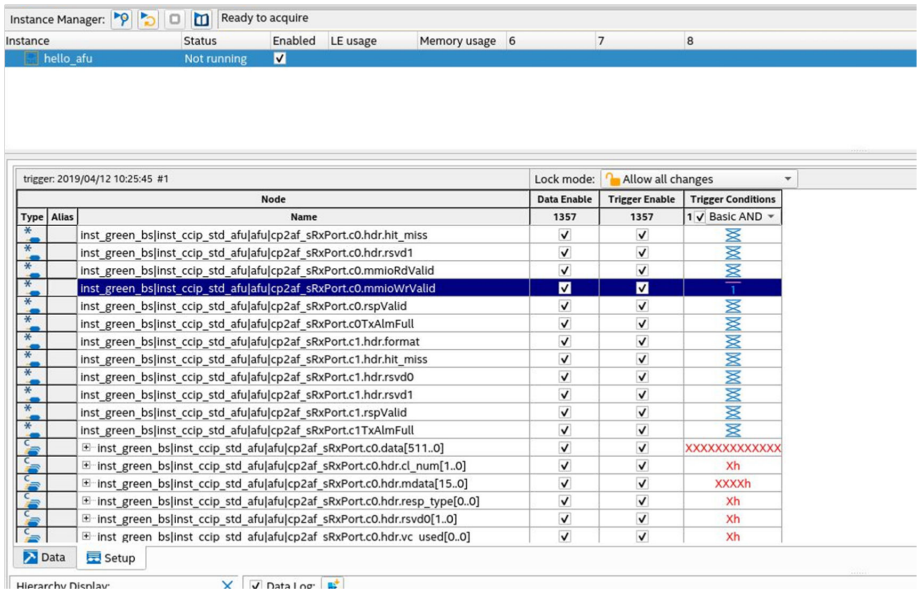
The instance manager should show "Ready to acquire".

d. Review the `hello_afu.sv` code and notice the following line:

```
81      //
82      // Receive MMIO writes
83      //
84      always_ff @(posedge clk)
85      begin
86          if (reset)
87          begin
88              scratch_reg <= '0;
89          end
90          else
91          begin
92              // set the registers on MMIO write request
93              // these are user-defined AFU registers at offset 0x40.
94              if (cp2af_sRxPort.c0.mmioWrValid == 1)
95              begin
96                  case (mmioHdr.address)
97                      16'h0020: scratch_reg <= cp2af_sRxPort.c0.data[63:0];
98                  endcase
99              end
100         end
101     end
```

e. Enable the Signal Tap instance to capture data when `cp2af_sRxPort.c0.mmioWrValid`.

4. Select the `cp2af_sRxPort.co.mmioWrValid` signal name, then right click the "Trigger Conditions" value and set this to 1 as shown below:



5. Enable the `hello_afu` Signal Tap instance by entering **F5**.
6. Modify the N3000 host application to add a shared connection to the FPGA in order to create host transactions that can cause the `hello_afu` STP interface to activate. This shared connection allows the Signal Tap and host communication to be shared through the PCIe bus.
   a. Edit `$N3000_EXAMPLE_ROOT/hello_afu/hw/afu/sw/hello_afu.c` to change the following line from:

   ```
   res = fpgaOpen(afc_token, &afc_h, 0);
   ```

   To;

   ```
   res = fpgaOpen(afc_token, &afc_h, FPGA_OPEN_SHARED);
   ```

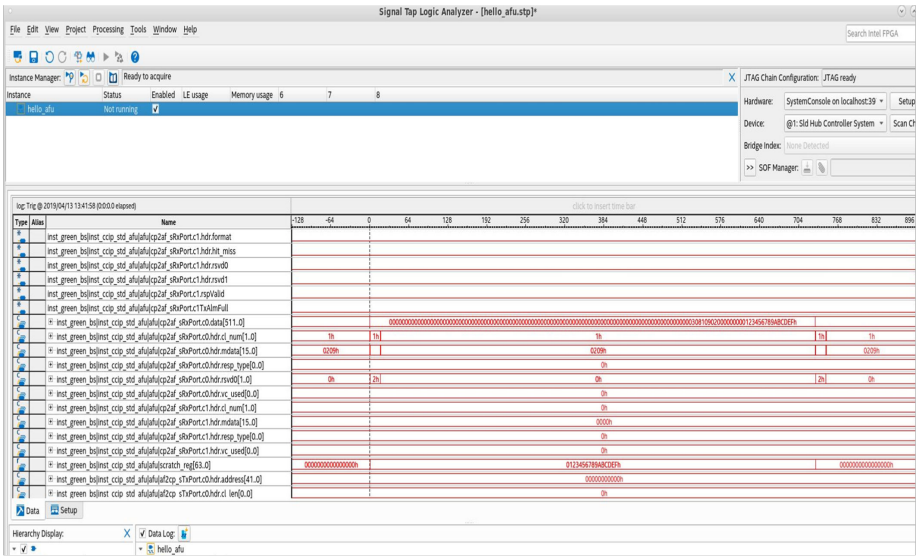
   b. Save `hello_afu.c` and build the code

   ```
   $ make
   ```

   c. Run the `hello_afu` host code as root or sudo as shown below:

   ```
   # ./hello_afu
   Running Test
   AFU DFH REG = 1000010000000000
   AFU ID LO = 9722d43375b61c66
   AFU ID HI = 850adcc26ceb4b22
   AFU NEXT = 00000000
   AFU RESERVED = 00000000
   Reading Scratch Register (Byte Offset=00000080) = 00000000
   MMIO Write to Scratch Register (Byte Offset=00000080) = 123456789abcdef
   Reading Scratch Register (Byte Offset=00000080) = 123456789abcdef
   Setting Scratch Register (Byte Offset=00000080) = 00000000
   Reading Scratch Register (Byte Offset=00000080) = 00000000
   Done Running Test
   ```

   Your Signal Tap instance captures the write transactions as shown below:



5.4. How to Exit from the Debug Session

1. Close Signal Tap instance.
2. In System Console, select **File ➤ Exit**.
3. In the debug target host shell, terminate the `mmlink` with a <Ctrl-c> key sequence.
4. If you want to debug another AFU, you must first terminate the active `mmlink` process.
5. Before loading a new AFU, be sure to terminate any OPAE host application code that has the currently loaded AFU open.

5.5. Troubleshooting Remote Debug Connections

If you get a **Failed to connect** message after invoking **System Console**:

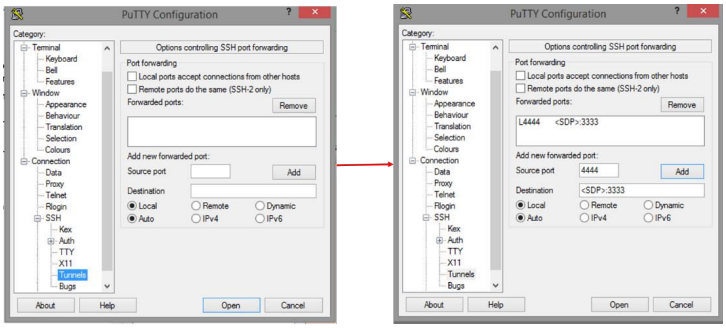
1. Check if a firewall is blocking your port. If so, unblock your port by running the following:

   ```
   firewall-cmd --add-port=3333/tcp --permanent
   ```

   Consider adding port tunneling. Do this when the debug target host is behind a firewall and your remote debug host is not.
2. On the debug target host, run **mmlink** as before. Note that **mmlink** provides an option to specify a port number. Port 3333 is the default. Refer to the following:

   ```
   $ mmlink -- port=3333
   ```
3. Setup port tunneling on the remote debug host. This example shows how to do so on a Windows remote debug host using PuTTY.
4. Use a PuTTY configuration screen as shown in the *SSH Tunneling with PuTTY* figure. For <SDP>, enter the name of the debug target host. This forwards the local port on your Windows host 4444 to port 3333 on the debug target host.

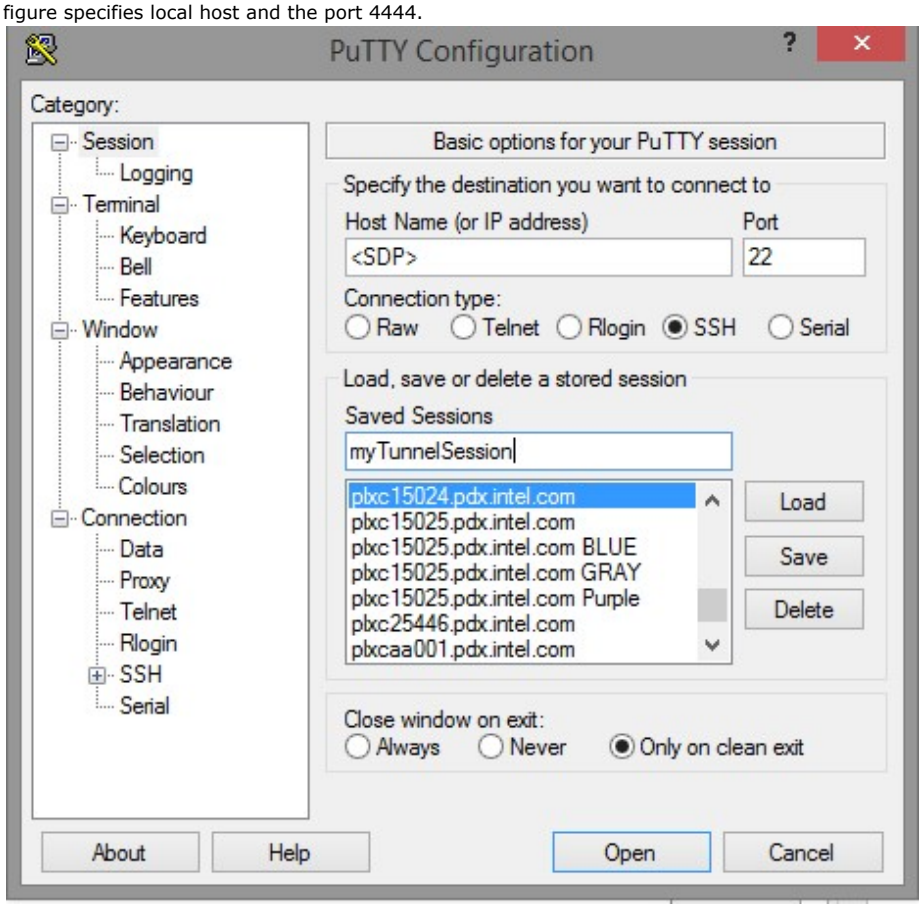
Figure 46. SSH Tunneling with PuTTY



5. Click **Session**, specify the name of the debug target host, click **Save**, and then **Open**. Login to the debug target host. This is your tunneling session.

Figure 47. Save and Open the Tunneling Session

This figure specifies local host and the port 4444.



Once the tunneling session is setup this forwarding is complete.

6. Open a Windows Command Window and issue the **system-console** command as shown in the "Save and Open the Tunneling Session" figure.
7. Run the `System Console with Port Forwarding` command:

```
$ system-console --rc_script=mmlink_setup_profiled.tcl\
remote_debug.sof localhost 4444
```

As before, the Intel Quartus Prime System Console comes up. Wait for the **Remote system ready** message on the tcl console of the System Console.

6. Document Revision History for the Accelerator Functional Unit Developer Guide: Intel FPGA Programmable Acceleration Card N3000 Variants

Document Version	Intel Acceleration Stack Version	Changes
2022.07.15	1.3.1	Updated the **Target** list in section: *Build with `make`*.
2020.09.08	1.3.1	Updated in accordance with the Intel Acceleration Stack 1.3.1 Version for Intel FPGA Programmable Acceleration Card N3000.
2020.06.15	1.3	Added enhancements for 1.3: • *Supported Ethernet Network Configurations*—Added supported Board OPN • *Ethernet Interface*—Updated the Instantiated Ethernet MACs figure to reflect that in the 4x25, only one QSFP and one Retimer are active. • Added the following sections: — *Loading Your FPGA image with JTAG* — *Prepare your N3000 for JTAG* — *Disable PCIe Automatic Error Reporting (AER)* — *Use JTAG to load A10 *.sof file* — *How to rescan PCIe bus and re-enable PCIe AER* — *AFU Clocks* — *Hello_afu – uClk_usr* — *Hello_afu – New PLL* — *Creating an AFU with High Level Synthesis (HLS)*
2019.12.06	1.1	Initial Release