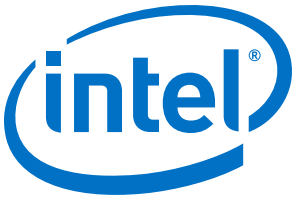


# Accelerator Functional Unit Developer's Guide for Intel® FPGA Programmable Acceleration Card

Updated for Intel® Acceleration Stack for Intel® Xeon® CPU with FPGAs: **1.2 and 2.0.1**

**UG-20169 | 2020.07.20**
Latest document on the web: **PDF** | **HTML**

**Subscribe**
**Send Feedback**

# Contents

# 1. About this Document

This document serves as a hardware developers guide for developing Accelerator Functional Units (AFUs) for the Intel Acceleration Stack for Intel Xeon® CPU with FPGAs product, hereafter referred to as the Acceleration Stack.

## 1.1. Intended Audience

The intended audience consists of FPGA RTL designers developing AFUs for the Acceleration Stack on the Intel FPGA Programmable Acceleration Card (Intel FPGA PAC) and the hardware platforms (referred to as Intel FPGA PAC throughout this document).

## 1.2. Conventions

**Table 1. Document Conventions**

| Convention | Description |
|---|---|
| **#** | Precedes a command that indicates the command is to be entered as root. |
| **$** | Indicates a command is to be entered as a user. |
| `This font` | Filenames, commands, and keywords are printed in this font. Long command lines are printed in this font. Although long command lines may wrap to the next line, the return is not part of the command; do not press enter. |
| *<variable_name>* | Indicates the placeholder text that appears between the angle brackets must be replaced with an appropriate value. Do not enter the angle brackets. |

## 1.3. Acronym List for Accelerator Functional Unit Developer's Guide

**Table 2. Acronyms**

| Acronyms | Expansion | Description |
|---|---|---|
| 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 | Accelerator Function | Compiled Hardware Accelerator image implemented in FPGA logic that accelerates an application. An AFU and associated AFs may also be referred to as GBS (Green-Bits, Green BitStream) |

***continued...***

| Acronyms | Expansion | Description |
|---|---|---|
| | | in the Acceleration Stack installation directory tree and in source code comments. |
| API | Application Programming Interface | A set of subroutine definitions, protocols, and tools for building software applications. |
| ASE | AFU Simulation Environment | Co-simulation environment that allows you to use the same host application and AF in a simulation environment. ASE is part of the Intel Acceleration Stack for FPGAs. |
| CCI-P | Core Cache Interface | CCI-P is the standard interface AFUs use to communicate with the host. |
| FIU | FPGA Interface Unit | FIU is a platform interface layer that acts as a bridge between platform interfaces like PCIe*, UPI and AFU-side interfaces such as CCI-P. |
| FIM | FPGA Interface Manager | The FPGA hardware containing the FPGA Interface Unit (FIU) and external interfaces for memory, networking, etc.<br>The FIM may also be referred to as BBS (Blue-Bits, Blue BitStream) in the Acceleration Stack installation directory tree and in source code comments.<br>The Accelerator Function (AF) interfaces with the FIM at run time. |
| NLB | Native Loopback | The NLB performs reads and writes to the CCI-P link to test connectivity and throughput. |
| OPAE | Open Programmable Acceleration Engine | The OPAE is a software framework for managing and accessing AFs. |
| PR | Partial Reconfiguration | The ability to dynamically reconfigure a portion of an FPGA while the remaining FPGA design continues to function. |
| TCP | Transmission Control Protocol | TCP is a standard Internet protocol that defines how to establish and maintain a network conversation through which application programs can exchange data. |
| PIM | Platform Interface Manager | An abstraction layer for managing top-level device ports and system-provided clock crossing. |
| HSSI | High Speed Serial Interface | Reference to the multi-gigabit serial transceiver I/O in the FIM and the corresponding interface to the AFU. |

## 1.4. Acceleration Glossary

### Table 3. Acceleration Stack for Intel Xeon CPU with FPGAs Glossary

| Term | Abbreviation | Description |
|---|---|---|
| Intel Acceleration Stack for Intel Xeon CPU with FPGAs | Acceleration Stack | A collection of software, firmware and tools that provides performance-optimized connectivity between an Intel FPGA and an Intel Xeon processor. |
| Intel FPGA Programmable Acceleration Card (Intel FPGA PAC) | Intel FPGA PAC | PCIe FPGA accelerator card. Contains an FPGA Interface Manager (FIM) that pairs with an Intel Xeon processor over the PCIe bus. |

## 1.5. Related Documentation

### Table 4. Item Description

| Item | Description |
|---|---|
| Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA | This document describes the Acceleration Stack and provides instructions for hardware and software installation and setup required for development with the stack. |
| Intel Acceleration Stack Quick Start Guide for Intel FPGA Programmable Acceleration Card D5005 | This document describes the Acceleration Stack and provides instructions for hardware and software installation and setup required for development with the stack. |
| Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual | This document describes the CCI-P protocol and requirements placed on AFUs. |
| Networking Interface for Open Programmable Acceleration Engine: Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA | This document describes the HSSI device interface offered by the Intel PAC with Intel Arria 10 GX FPGA hardware platform and the OPAE tools and driver features that support the network port feature. |
| Networking Interface for Open Programmable Acceleration Engine: Intel FPGA Programmable Acceleration Card D5005 | This document describes the HSSI device interface offered by the Intel FPGA PAC D5005 platform and the OPAE tools and driver features that support the network port feature. |
| Intel Accelerator Functional Unit Simulation Environment User Guide | This document provides instructions on how to use the Intel Accelerator Functional Unit Simulation Environment. |
| Open Programmable Acceleration Engine (OPAE) Tools Guide | This user guide documents the utilities provided in the Open Programmable Acceleration Engine (OPAE) software component of the Acceleration Stack. |

# 2. Introduction

## 2.1. Getting Started with AFU Development

Depending on which Intel FPGA PAC you are using, please refer to one of the following Quick Start Guides:

- If you are using Intel PAC with Intel Arria® 10 GX FPGA, refer to the *Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA*.
- If you are using Intel FPGA PAC D5005, refer to the *Intel Acceleration Stack Quick Start Guide for Intel FPGA Programmable Acceleration Card D5005*.

The *Quick Start Guide* provides an overview of the Acceleration Stack and provides instruction for installation and setup of hardware and software components of the stack, including the OPAE SDK used to develop AFUs and generate loadable AF images. It is essential to familiarize yourself with the concepts developed for the Acceleration Stack and to complete the installation and setup procedures covered in the *Quick Start Guide*.

This guide for AFU development builds on the concepts and environment setup established in the *Quick Start Guide*.

**Related Information**

- Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA
- Intel Acceleration Stack Quick Start Guide for Intel FPGA Programmable Acceleration Card D5005

### 2.1.1. Development Environment References

The `OPAE_PLATFORM_ROOT` environment variable points to the OPAE SDK installation as detailed in the *Quick Start Guide*.

### 2.1.2. FPGA Tools and IP Requirements

You need to download the Intel Acceleration Stack for Development to generate the Accelerator Functions (AFs).

The Intel Acceleration Stack for Development installer includes licenses for the following software and IPs required for the generation of the AFs:

- Intel Quartus® Prime Pro Edition software

  *Note:* For information on compatible version of the software for each platform, refer to the platform specific *Release Notes*.
- Intel FPGA PCI Express SR-IOV Block IP license
- Network IP license

You do not need to purchase the license separately for these IPs.

For requirements when using the ASE for AFU functional verification, refer to the *Intel Accelerator Functional Unit Simulation Environment User Guide*.

**Related Information**

- Intel Accelerator Functional Unit Simulation Environment User Guide
- Intel Acceleration Stack for Intel Xeon CPU with FPGAs Version 1.2 Release Notes
- Intel Acceleration Stack for Intel Xeon CPU with FPGAs Version 2.0 Release Notes: For the Intel FPGA Programmable Acceleration Card D5005
- Installing the Intel Acceleration Stack Development Package on the Host Machine

## 2.2. Base Knowledge and Skills Prerequisites

The Acceleration Stack is a framework and toolset to leverage FPGA technology Most of the platform-level complexity has been abstracted away for the AFU developer by the FPGA Interface Manager (FIM) in the FPGA static region. This guide assumes the following FPGA logic design-related knowledge and skills:

- Familiarity with PR compilation flows, including the Intel Quartus Prime Pro Edition PR flow, concepts of physical and logical partitioning in the FPGA, module boundary best practices, and resource restrictions.

  The hardware compilation flow automates management of the partial reconfiguration region.
- Knowledge and skills in static timing closure, including familiarity and skill 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 timing on critical paths.
- Knowledge and skills with industry standard RTL simulation tools supported by the Acceleration Stack. For more information, refer to the *Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) User Guide*.

**Related Information**

Intel Accelerator Functional Unit Simulation Environment User Guide

# 3. Getting Started with Platform Configuration

This chapter guides you through the process to generate an AF for the `hello_afu` sample AFU provided in the Acceleration Stack installation. Successful completion of the steps in this chapter quickly verifies your AFU development environment using a known-good design.

Build the `hello_afu` sample AFU by invoking the `run.sh` script from a terminal window as shown in Example 1.

Ensure that you declare the $OPAE_PLATFORM_ROOT variable before you compile the sample AFU. If it is not declared, you can source `init_env.sh` located in the Acceleration Stack directory. For more details, refer to the *Intel Acceleration Stack Quick Start Guide*.

*Note:* This step takes about 30 minutes to complete.

**Example 1. Compile `hello_afu` Sample AFU**

```
cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu
afu_synth_setup --source hw/rtl/filelist.txt build_synth
cd build_synth
run.sh
```

When the shell script completes, it indicates successful generation of the AF:

```
$OPAE_PLATFORM_ROOT/hw/samples/hello_afu/build_synth/hello_afu.gbs
```

You can optionally use the helper script `clean.sh`, to remove the following build output from the Intel Quartus Prime PR compilation invoked by `run.sh`:

- `./build/*.qdb`
- `./build/qdb`
- `./build/output_files/`
- `./build/*qarlog`
- `./build/*.qdf`

**Example 2. Clean up from the PR Compilation (Optional)**

```
cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu/build_synth
clean.sh
```

Successfully compiling the `hello_afu` sample AFU verifies that your environment is setup and ready to begin developing your own custom AFUs.

# 4. The Accelerator Functional Unit (AFU)

The AFU is a function or set of functions that can be accelerated on an OPAE hardware platform. The AFU is described in RTL and then compiled with the OPAE SDK to generate an Accelerated Function (AF) image for the target hardware platform. An AF is a compiled hardware accelerator image implemented in FPGA logic that accelerates an application. The AF image is used by OPAE to load the AFU to the PR region.

An AFU has two main communication paths between the host:

- FPGA to host transactions: The FPGA accesses host memory (256 terabyte address space) using a 512 bit data path. This data path has separate channels for read and write traffic allowing for simultaneous read and write to occur. The read and write channels support bursts of 1, 2, and 4 cache lines.
- Host to FPGA (MMIO) transactions: The host can access a 256 KB address space within the FPGA. This address space contains Device Feature Header (DFHs) and the control and status registers of the AFU hardware. DFHs are small ROMs that hold metadata about the hardware that are enumerated by the OPAE SDK.

The AFU can access host memory on a cache line basis (64 bytes) through the CCI-P interface. OPAE defines up to 256 KB of memory mapped I/O (MMIO) space for AFUs that the host can access using the OPAE driver and APIs. At the bottom of the MMIO space, the AFU must implement the following OPAE requirements:

- AFU DFH - a 64-bit header at MMIO address offset 0x0
- AFU ID - a 128-bit UUID at MMIO address offset 0x2 (CCI-P D-word address)

The following sections of the *CCI-P Reference Manual* document the CCI-P protocol and all OPAE requirements for an AFU design, including the DFH and AFU ID format:

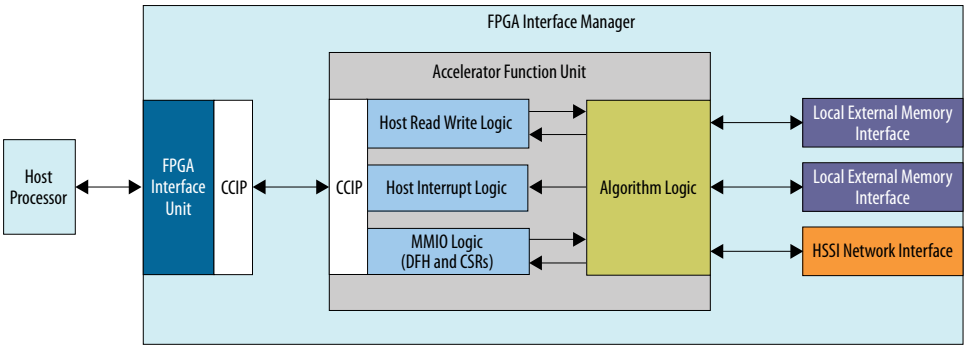
- CCI-P Interface
- AFU Requirements
- Device Feature List

### Related Information

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

## 4.1. AFU Design Components

**Figure 1. AFU High Level Block Diagram**




A typical AFU design includes the following components:

- RTL description of the algorithm or function being accelerated
- RTL description to implement the base requirements placed on AFUs by OPAE (e.g., DFH, AFU ID in MMIO space). See the *CCI-P Reference Manual* for more details on the RTL description.
- Supportive infrastructure
  - Logic to map AFU CSRs into MMIO space
  - Memory mastering logic
    - FPGA to host memory access
    - Local FPGA memory access
- Debug and Performance monitoring
  - Signal Tap with the Remote Debug feature
  - Performance monitoring and counters within the scope of the AFU

The interfaces provided by OPAE for host and local memory access are basic, slave access interfaces. The host only has access to the AFU's 256KB MMIO space. The AFU must implement a DMA to move large workload data to and from host memory. The `dma_afu` sample AFU in the OPAE platform installation provides an example for moving data between the host and local memory.

The FIM supports notification for illegal accesses made on the CCI-P interface and performance monitoring capabilities accessible by the host through the FME in the FIU. Any error handling and performance monitoring must be implemented in the AFU by developer.

The FIM provides for AFU remote debug through the FME connected to an OPAE tool that hosts the debug connection over TCP. The AFU designer must instrument the AFU with debug instances and nodes using tools such as Signal Tap. The `nlb_mode_0_stp` sample AFU in the OPAE platform installation provides an example for enabling an AFU for remote debug with Signal Tap over a TCP connection.

## 4.2. Basic Building Blocks

Intel FPGA Basic Building Blocks (BBBs) are reference designs of common functions that can be used in AFU designs to implement supportive infrastructure such as CCI-P memory access property transformations and DMA. These references are provided as-is. They are not validated by Intel. The available BBBs, including documentation, are maintained at the GitHub site.

### Related Information

Basic Building Blocks (BBB) for OPAE-managed Intel FPGAs

# 5. Developing AFUs with the OPAE SDK

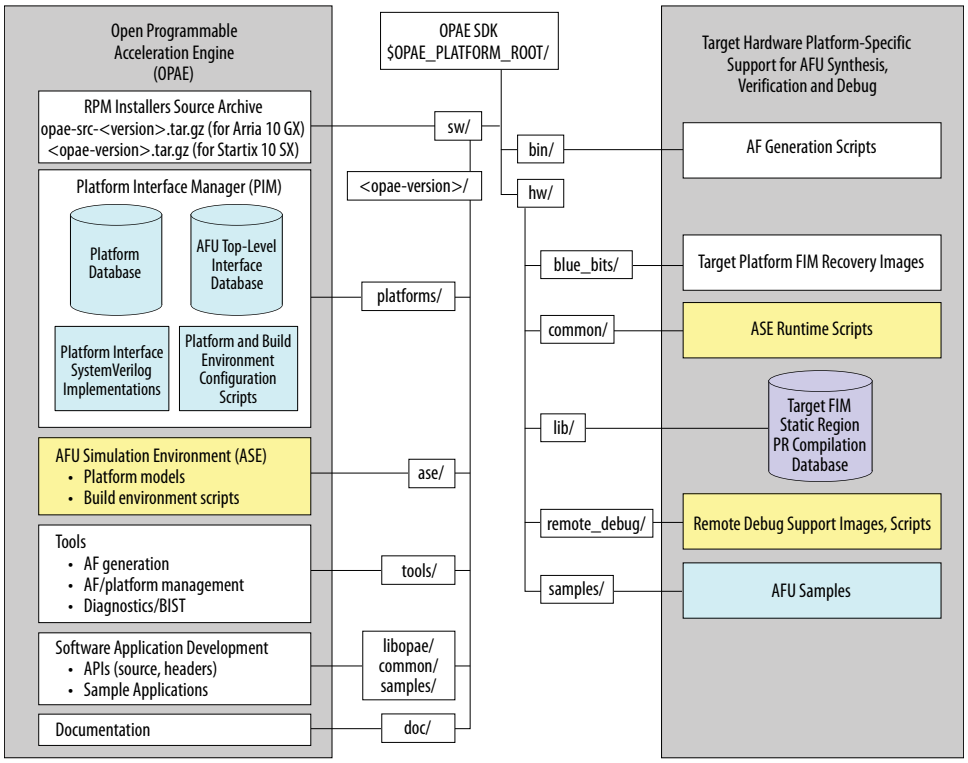
## 5.1. Overview of the OPAE SDK

The OPAE SDK is a development environment that supports synthesizing AFs targeted for a specific OPAE-compliant hardware platform from an OPAE-compliant AFU.

The OPAE SDK consists of two hardware development components:

- The database, tools, scripts and ancillary files required to target AF generation for a specific hardware platform.
- The OPAE version supported by the hardware platform used to configure a build environment for AFU simulation and compilation on the target hardware platform.

**Figure 2. Overview of the OPAE SDK**




OPAE's Platform Interface Manager (PIM) defines a non-hardware specific OPAE Platform that provides generic classes of device interfaces. The OPAE platform is an abstraction of a hardware platform for which AFUs are designed. This level of abstraction enables generating AFs from AFUs designed for the generic OPAE Platform for any OPAE-compliant hardware platform that offers the device interfaces required

by the AFU. The PIM generates a platform shim based upon device interfaces and properties requested by the AFU. The platform shim is inserted between the hardware platform's PR region boundary and the AFU and provides the top-level module interface for the AFU.

See the Overview of the OPAE Platform for AFUs on page 13 for more details on the OPAE.

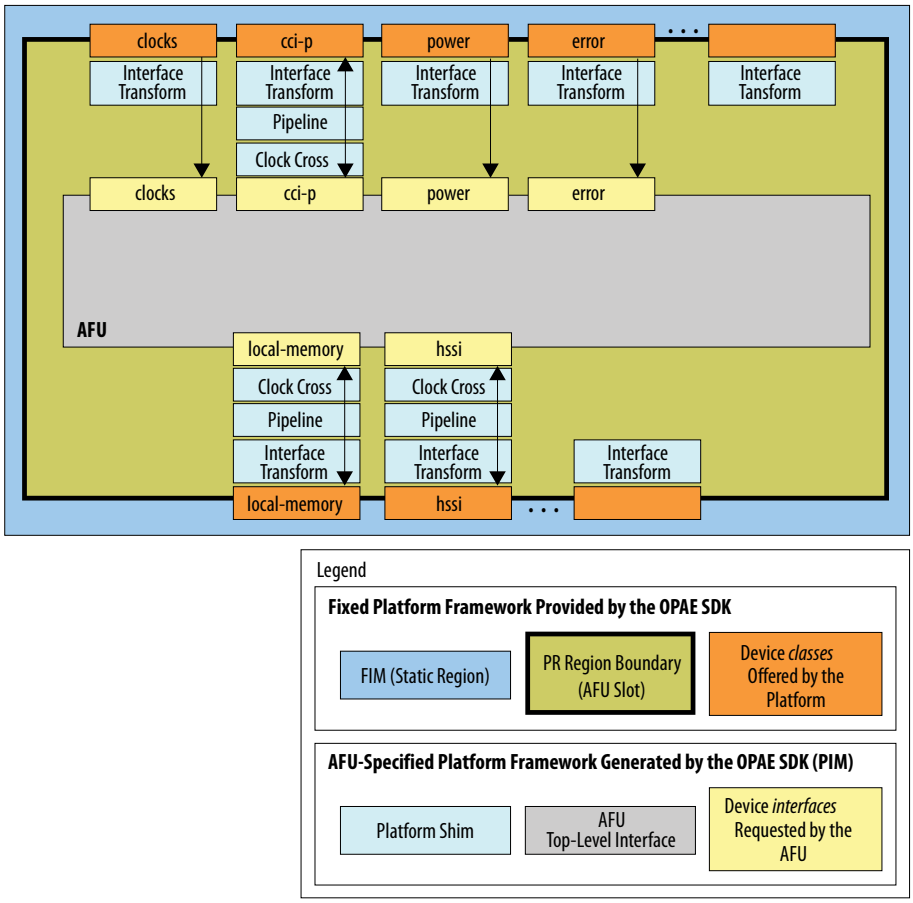
See the OPAE SDK Design Flow for AFU Development on page 17 for the process used by AFUs to request top-level interfaces and configure simulation and synthesis build environments.

## 5.2. Overview of the OPAE Platform for AFUs

The PIM defines a generic OPAE Platform for which AFU top-levels should be designed to ensure provisioning on multiple hardware platforms.

The figure below shows how the platform shim generated by the PIM enables AFU integration on a specific target hardware platform.

**Figure 3. OPAE Platform Block Diagram**



AFUs are designed to use generic top-level interfaces to a set of generic device classes such as a host device (`cci-p`), local memory, network port I/O, clocks, and power and error management. The AFU requests the device interfaces and properties it needs from the PIM using a platform configuration file specification.

### 5.2.1. Platform Device Classes

The OPAE Platform provides for AFU integration into the stack through several device classes. Each device class offers one or more port interfaces, each of which have properties of their own. AFUs request a specific device interface and properties from the PIM. The PIM implements the requested interfaces and properties in a platform shim that translates hardware platform-specific device interfaces to the OPAE Platform's generic device interfaces used by the AFU.

The Intel FPGA PAC offers the following device classes:



#### 5.2.1.1. The `clocks` Device Class

The Intel FPGA PAC platform offers the `clocks` device class with the `pClk3_usr2` interface, which consists of a list of port signals documented in the *CCI-P Reference Manual*.

#### 5.2.1.2. The `cci-p` Device Class

The Intel FPGA PAC platform offers the `cci-p` device class with the `struct` interface. The structures defined in the following package in the OPAE SDK:

```
$OPAE_PLATFORM_ROOT/sw/<opae-version>/platforms/platform_if/rtl/device_if/
ccip_if_pkg.sv
```

To use the structures defined in `ccip_if_pkg.sv` include `platform_if.vh` in your AFU source files. For example, `` `include "platform_if.vh" ``.

The CCI-P interface is used by the AFU to access host memory and to respond to MMIO requests from the host. It is composed of three command/response channels:

- Channel 0 - It is used by the AFU for host memory read requests and responses. Channel 0's response port is also used for receiving MMIO read and write requests from the host.
- Channel 1 - It is used by the AFU for host memory write requests and responses. It is also used for issuing write fences and interrupts.
- Channel 2 - It is used by the AFU for MMIO read responses back to the host.

The CCI-P interface and protocol are documented in the *CCI-P Reference Manual*.

**Related Information**

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

### 5.2.1.3. The `power` Device Class

The Intel FPGA PAC platform offers the `power` device class with a 2-bit interface. This interface drives the signal that represents the power state requests documented in the *Additional Control Signals* section of the *CCI-P Reference Manual*.

**Related Information**

Additional Control Signals

### 5.2.1.4. The `error` Device Class

The Intel FPGA PAC platform offers the `error` device class with a 1-bit interface. This interface drives the signal that represents the CCI-P protocol error documented in the *Additional Control Signals* section of the *CCI-P Reference Manual*.

**Related Information**

Additional Control Signals

### 5.2.1.5. The `hssi` Device Class

The Intel FPGA PAC platform offers the `hssi` device class with the `raw_pr` interface, which consists of a SystemVerilog interface defined in the following Verilog header in the OPAE SDK:

```
$OPAE_PLATFORM_ROOT/hw/lib/build/platform/pr_hssi_if.vh
```

The HSSI interface is used by the AFU to access the network port on the Intel FPGA PAC platforms. It is composed of the Native PHY Transceiver interface with a generic parallel interface to support multiple configurations by the HSSI PHY in the FIM.

The HSSI interface is an optional interface that AFUs can request from the Intel FPGA PAC platform. The Intel FPGA PAC platforms with HSSI interface contain sample AFUs in the directories starting with `eth_e2e_e<data_rate>` or `hssi_prbs`

**Related Information**

- Intel Arria 10 Transceiver PHY User Guide
- Intel Stratix 10 L- and H-Tile Transceiver PHY User Guide
- Intel Stratix 10 E-Tile Transceiver PHY User Guide
- Networking Interface for Open Programmable Acceleration Engine: Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA
- 10 Gbps Ethernet Accelerator Functional Unit (AFU) Design Example User Guide: Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA
- 40 Gbps Ethernet Accelerator Functional Unit (AFU) Design Example User Guide: Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA
- Networking Interface for Open Programmable Acceleration Engine User Guide: Intel FPGA Programmable Acceleration Card D5005

#### 5.2.1.6. The `local-memory` Device Class

The Intel FPGA PAC platform offers the `local-memory` device class with the following choice of interfaces:

- **`avalon_mm`** - a SystemVerilog interface defined in the following header file in the OPAE SDK:

  `$OPAE_PLATFORM_ROOT/sw/<opae-version>/platforms/platform_if/rtl/device_if/avalon_mem_if.vh`

- **`avalon_mm_legacy_wires_2bank`** - a fixed port list of signal wires specific to the Intel FPGA PAC platform. This interface is for legacy support of AFUs developed with earlier versions of the OPAE SDK. For portability to future platforms, consider porting existing AFUs designed with the legacy interface to the `avalon_mm` interface.

*Note:* The new AFU design uses the `avalon_mm` interface. Intel recommends to use `avalon_mm` interface for your new AFU designs and avoid using legacy interface.

The AFU accesses local memory on the Intel FPGA PAC through the Avalon® Memory-Mapped (Avalon-MM) slave interfaces provided by the FIM. The Intel FPGA PAC platforms typically provide one or more bank of local memory. For detailed information on bank of local memory, refer to the *FIM Data Sheet*. Each bank interface is synchronous to its own clock source provided by the interface.

The local memory interface is an optional interface that AFUs can request from the Intel FPGA PAC platform. See the following two sample AFUs for examples of using the local memory interface:

- `$OPAE_PLATFORM_ROOT/hw/samples/hello_mem_afu`
- `$OPAE_PLATFORM_ROOT/hw/samples/dma_afu`

Intel recommends using `avalom_mm` interface for all the new designs and not use the legacy interface.

### 5.2.2. The Platform Interface Manager (PIM)

The PIM contains a collection of shims. The PIM abstracts the details of the target hardware platform from the AFU to support AFU portability to multiple platforms without modifying the AFU. The PIM performs the following functions based upon the AFU's platform configuration described in its `.json` file:

- Validates that an OPAE device interface requested by the AFU is provided by the target platform.
- Properly terminates any OPAE device class offered by the platform but not requested by the AFU.
- Enables an AFU to optionally request an OPAE device interface from the target platform and adjust the build-out of its implementation based on whether the requested interface is available. For example, the AFU can optionally request local memory and build-out to use it if available from the target platform, otherwise it builds-out to function without local memory. See the `nlb_mode_0` sample AFU for an example.
- Provides register pipeline stages on requested OPAE device interfaces to aid static timing closure during AF generation.
- Provides asynchronous clock crossing from an OPAE device interface's native clock to a target clock requested by the AFU. For example, the AFU can request that all requested OPAE device interfaces be retimed to the `uClk_usr` clock source provided by the `clocks` interface. See the `hello_mem_afu` sample AFU for an example.

#### 5.2.2.1. Interface Transforms

The PIM transforms a device class offered by the platform into the specific device interface requested by the AFU. Any device classes on the platform not requested by the AFU are properly terminated to support AF generation. The transformation is typically a simple, direct connection between the platform and AFU consisting of device interface ports or structures or a bundling of the ports into an interface vector. For example, the PIM directly connects the platform's `cci-p` interface structures and `clocks`, `power` , and `error` ports to the AFU. In the case of `local-memory`, the PIM abstracts the hardware platform details from the AFU by packing the platform's interface into a SystemVerilog interface vector.

#### 5.2.2.2. Pipelining

The PIM inserts register pipeline stages on device interfaces as requested by the AFU.

#### 5.2.2.3. Clock Crossing

The PIM inserts asynchronous clock crossing on device interfaces to cross from the interface's native clock to a clock specified by the AFU. For example, the AFU can request that all device interfaces be synchronized to `uClk_usr` from the `clocks` interface.

## 5.3. OPAE SDK Design Flow for AFU Development

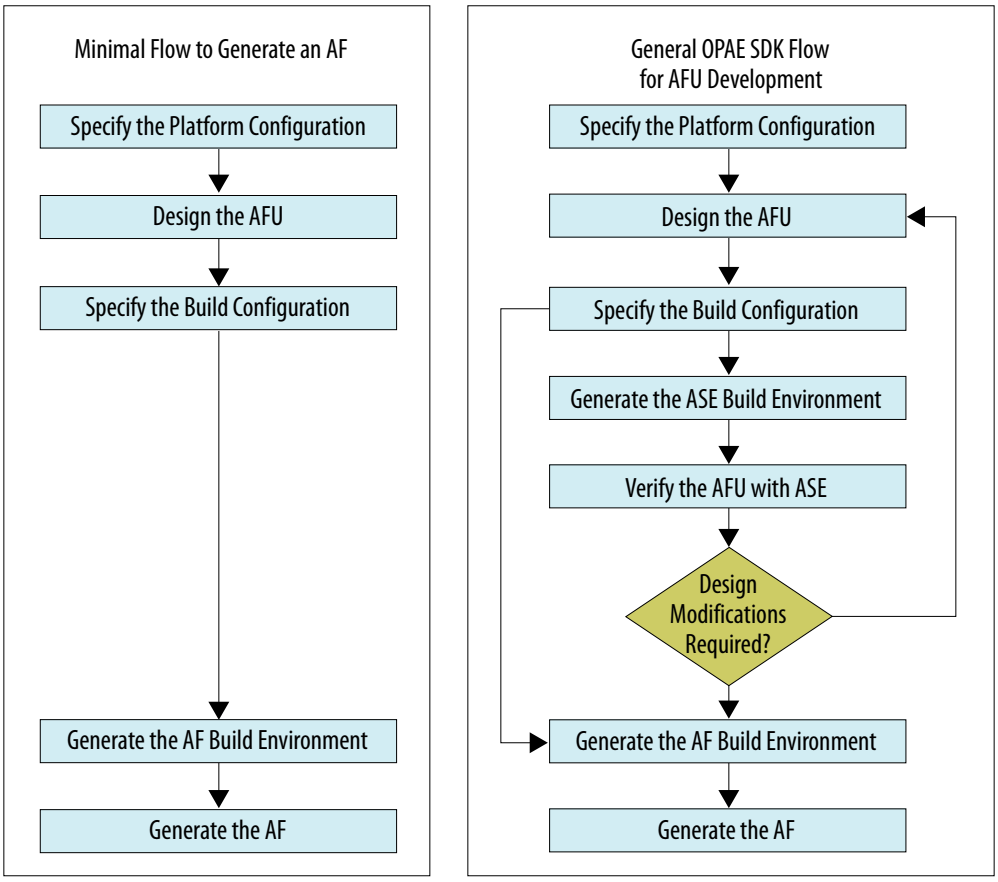
### 5.3.1. Overview of the Design Flow

This section provides a summary overview of the OPAE SDK design flow for AFU development. Refer to the Design Flow Details on page 21 for a detailed description of each step included in the flow.

The figure below shows the design flow when using the OPAE SDK to verify and synthesize AFs for a target hardware platform.

The minimal flow depicted in the figure shows the minimum flow steps to generate an AF from an AFU design, while the depiction of the general flow shows where AFU verification with ASE fits in the overall flow.

**Figure 4. OPAE SDK Design Flow for AFU Development**




- *Specify the Platform Configuration* for the AFU in a platform configuration file (`.json`) by requesting a top-level AFU interface along with any required interface properties. The top-level interface requested by the AFU defines its SystemVerilog top-level module port definition.
- *Design the AFU* within this top-level module port definition.
- With the AFU design file set established, *Specify the Build Configuration* for both AFU simulation and AF synthesis with a single build configuration file (`filelist.txt`), which lists the AFU's design source (e.g., RTL, IP, Platform Designer subsystems, constraints) along with any required macro definitions and include files.
- Using the PIM, *Generate the AF/ASE Build Environment* based upon the AFU's platform and build configuration file specifications and the target hardware platform. At this point in the flow, you can use ASE to run OPAE software applications on a simulation target instantiated from the AFU's RTL source and the hardware platform model provided by OPAE.
- Finally, *Generate the AF* using the AF generation scripts provided by the SDK.

#### 5.3.1.1. Minimal Flow Example

The following example shows the minimal flow necessary to generate an AF. It uses the `hello_afu` sample AFU included in the OPAE SDK. The `hello_afu` sample can be used as template for AFU designs that require only a host device interface and no local memory or network port I/O.

The following is a synopsis of the minimal set of OPAE SDK commands required to generate an AF from the `hello_afu` sample AFU:

```
cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu

afu_synth_setup --source hw/rtl/filelist.txt build_synth

cd build_synth

$OPAE_PLATFORM_ROOT/bin/run.sh
```

The execution of these commands generates an AF (`.gbs`) image in the `build_synth` sub-directory. The rest of this section elaborates on the minimal flow steps.

##### 5.3.1.1.1. Specify the Platform Configuration

The `hello_afu` sample specifies its platform configuration in the following `.json` file:

`$OPAE_PLATFORM_ROOT/hw/samples/hello_afu/hw/rtl/hello_afu.json`

The platform configuration file provides an example of the following:

- The AFU requests the `ccip_std_afu` top-level interface, which includes the `ccip`, `clocks`, `power` and `error` device interfaces. If the target hardware platform offers `local-memory` or `hssi` device classes, then the platform shim generated by the PIM terminates those interfaces.
- Uses top-level interface default properties (i.e., no pipelining or clock crossing).
- Specifies the AFU's UUID

##### 5.3.1.1.2. Design the AFU

The AFU's top-level interface request in its platform configuration file defines its top-level module. The `hello_afu` sample's top-level module definition is located here:

`$OPAE_PLATFORM_ROOT/hw/samples/hello_afu/hw/rtl/ccip_std_afu.sv`

The `hello_afu` sample implements the minimal requirements for an AFU specified in the *CCI-P Reference Manual* in the AFU submodule instanced by the `ccip_std_afu` top-level module and is described in the following SystemVerilog source:

`$OPAE_PLATFORM_ROOT/hw/samples/hello_afu/hw/rtl/afu.sv`

The `afu.sv` source file includes the `afu_json_info.vh` Verilog header file generated by the PIM and uses the AFU_ACCEL_UUID macro defined by `afu_json_info.vh` to set the UUID value as required by the *CCI-P Reference Manual*.

Each of the above SystemVerilog source files includes the `platform_if.vh` Verilog header file generated by the PIM, which makes available all the interface definitions used by the AFU.

#### 5.3.1.1.3. Specify the Build Configuration

The `hello_afu` sample specifies its build configuration in the following text file:

```
$OPAE_PLATFORM_ROOT/hw/samples/hello_afu/hw/rtl/filelist.txt
```

It lists all source files, including its platform configuration file (`.json`). The file references are relative to the build configuration file's directory location.

#### 5.3.1.1.4. Generate the AF Build Environment

To generate an AF build environment, open a terminal and enter the following command sequence:

```
cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu

afu_synth_setup --source hw/rtl/filelist.txt build_synth
```

The `afu_synth_setup` parses the build configuration file (`filelist.txt`) and generates a Intel Quartus Prime project in the specified directory (`build_synth`).

#### 5.3.1.1.5. Generate the AF

To generate an AF, enter the following commands:

```
cd build_synth

$OPAE_PLATFORM_ROOT/bin/run.sh
```

Completion of shell script indicates successful generation of the AF at the following location:

```
$OPAE_PLATFORM_ROOT/hw/samples/hello_afu/build_synth/hello_afu.gbs
```

### 5.3.1.2. General Flow Example

The OPAE SDK supports AFU verification with ASE which can be used at any time in the flow once you have an initial AFU design and have specified platform and build configurations. This section extends the minimal flow example by showing how to generate an ASE build environment and use ASE to run an OPAE host application against a combined RTL model of the AFU on the target hardware platform. This can be done using the `hello_afu` sample AFU.

#### 5.3.1.2.1. Generate the ASE Build Environment

Open a terminal window and enter the following commands to generate the ASE build environment:

```
cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu

afu_sim_setup --source hw/rtl/filelist.txt build_sim
```

The `afu_sim_setup` parses the `filelist.txt` file and generates a simulation project in the `build_sim` directory.

#### 5.3.1.2.2. Verify the AFU with ASE

Type the following commands to compile the AFU and platform simulation models and start the simulation server process:

```
cd build_sim

make

make sim
```

After the commands complete, ASE indicates that the server is ready for simulation. Note the instructions for setting the ASE_WORKDIR environment variable in the ASE client window.

Open a second terminal window and enter the following commands to start the ASE client process:

```
<Set ASE_WORKDIR as directed by the simulator in the server window.>

cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu/sw

make clean

make USE_ASE=1

./hello_afu
```

The OPAE host application runs on the host in the ASE client window process and the ASE server window process shows the AFU model responding to host MMIO accesses, host memory accesses initiated by the AFU, and interrupt vector information signaled by the AFU.

**Related Information**

Intel Accelerator Functional Unit Simulation Environment User Guide

### 5.3.2. Design Flow Details

This section describes each step of the OPAE SDK design flow in detail.

#### 5.3.2.1. Specify the Platform Configuration

An OPAE compliant AFU configures the OPAE Platform using a platform configuration file to specify the following to the PIM:

- Specify the AFU's UUID
- Request a top-level interface
- Extend a top-level interface with additional device interfaces
- Request pipelining on device interfaces
- Request clock crossing on device interfaces
- Specify a requested device interface as optional
- Specify AFU user clock timing

The platform configuration file uses the JSON format to specify the above tasks with key:value pairs.

#### 5.3.2.1.1. Specify the AFU's UUID

The single place to specify the AFU's UUID required by the OPAE Platform is in its platform configuration file. The PIM and OPAE runtime tools extract the AFU UUID from the platform configuration file for consumption by the AFU RTL implementation, and OPAE host applications and tools.

Specify the AFU UUID with the `afu-image:accelerator-clusters:accelerator-type-uuid` key as shown in the `json` file located at the following location:

`$OPAE_PLATFORM_ROOT/hw/samples/hello_afu/hw/rtl/hello_afu.json`

You can generate a 128-bit UUID using different ways, for example, you can use the following command:

```
uuidgen
```

#### 5.3.2.1.2. Request a Top-level Interface

The PIM defines the following two basic top-level AFU interfaces that consist of multiple device interfaces: `ccip_std_afu` and `ccip_std_afu_avalon_mm`. AFUs specify their top-level interface with the `afu-image:afu-top-interface:name` key in the platform configuration file.

The SystemVerilog interface definitions for the device interfaces listed below are documented in the following README:

`$OPAE_PLATFORM_ROOT/sw/<opae-version>/platforms/afu_top_ifc_db/README.md`

**1. `ccip_std_afu`**

This top-level interface consists of the `cci-p`, `clocks`, `power` and `error` device interfaces.

The top-level AFU module name remains `ccip_std_afu`. It includes the following device interfaces (device-class:interface):

- `cci-p:struct`
- `clocks:pClk3_usr2`
- `power:2bit`
- `error:1bit`

See the `hello_afu` sample `json` file for an example of an AFU requesting the `ccip_std_afu` top-level AFU interface at the following location:

`$OPAE_PLATFORM_ROOT/hw/samples/hello_afu/hw/rtl/hello_afu.json`

**2. `ccip_std_afu_avalon_mm`**

This top-level interface consists of the device interfaces included with the `ccip_std_afu` top-level plus a local memory interface.

The top-level AFU module name remains `ccip_std_afu`. It includes the following device interfaces (device-class:interfaces):

- All device interfaces of the `ccip_std_afu` top-level AFU module interface
- `local-memory:avalon_mm`

See the `hello_mem_afu` sample JSON file for an example of an AFU requesting the `ccip_std_afu_avalon_mm` top-level AFU interface:

`$OPAE_PLATFORM_ROOT/hw/samples/hello_mem_afu/hw/rtl/hello_mem_afu.json`

The PIM also defines a top-level AFU interface with a deprecated local memory device interface used by existing AFUs designed for earlier versions of the OPAE Platform. New AFU designs with local memory interfaces should be designed for the `ccip_std_afu_avalon_mm` top-level AFU interface.

#### 5.3.2.1.3. Extend a Top-level Interface

Additional device interfaces are accommodated by extending one of the predefined basic top-level AFU interfaces.

For example, the `eth_e2e_e10`, `eth_e2e_e40`, and `hssi_prbs` sample AFUs request an `hssi` device interface by extending the `ccip_std_afu` top-level AFU interface using the `afu-image:afu-top-interface:module-ports:[class|interface]` keys:

`$OPAE_PLATFORM_ROOT/hw/samples/eth_e2e_e10/hw/rtl/eth_e2e_e10.json`

`$OPAE_PLATFORM_ROOT/hw/samples/eth_e2e_e40/hw/rtl/eth_e2e_e40.json`

`$OPAE_PLATFORM_ROOT/hw/samples/eth_e2e_e40/hw/rtl/eth_e2e_e40.json`

#### 5.3.2.1.4. Request Device Interface Pipelining

The AFU can request the PIM to insert pipeline stages between the target hardware platform's PR region boundary and its top-level module device interfaces on the following device classes:

- `cci-p`
- `local-memory`

Use the following key:value pair on the class key you want pipeline stages inserted:

```
afu-image:afu-top-interface:module-ports:params:add-extra-timing-reg-stages:<integer-num>
```

For example, specify adding two pipeline stages on the local-memory device interfaces as follows:

```
{
    'class': 'local-memory',
    'params':
        {
```

```
            'add-extra-timing-reg-stages': 2
        }
}
```

#### 5.3.2.1.5. Request Device Interface Clock-crossing

The AFU requests the PIM to insert a clock crossing bridge to synchronize the following device class interfaces to a clock of the AFU's choosing:

- `cci-p`
- `local-memory`

Use the following key:value pair on the device class key you want synchronized to a clock chosen by the AFU:

```
afu-image:afu-top-interface:module-ports:params:clock:"<clock-name>"
```

For example, the `hello_mem_afu` sample AFU requests that the `cci-p` and `local-memory` device interfaces be synchronized to `uClk_usr` from the `clocks` interface:

`$OPAE_PLATFORM_ROOT/hw/samples/hello_mem_afu/hw/rtl/hello_mem_afu.json`

#### 5.3.2.1.6. Specify a Requested Device as Optional

By default, the PIM does not generate a platform shim for a target hardware platform that does not offer a device interface requested by the AFU. However, AFU's can specify a requested device interface as optional. For optionally requested device interfaces, the PIM generates a platform shim and build environments as long as the device interface is defined as optional by both the OPAE Platform and the target hardware platform. If the target hardware platform offers the device interface, the PIM transforms the interface with the properties requested by the AFU's platform configuration file, otherwise the PIM continues configuring the platform without any action on the unavailable device interface. In either case, the PIM defines a Verilog macro indicating whether the optionally requested interface is offered by the target hardware platform. AFU implementations must elaborate based on the macro definition.

Use the following key:value pair on the device class key you want to specify as optional (the default value is `false`):

```
afu-image:afu-top-interface:module-ports:optional:true
```

For example, the `nlb_mode_0` sample AFU optionally requests a `local-memory` interface and instantiates a memory tester module based on the related Verilog macro definition:

`$OPAE_PLATFORM_ROOT/hw/samples/nlb_mode_0/hw/rtl/nlb_mode_0.json`

The `cci-p` and `clocks` device interfaces are mandatory for AFUs.

#### 5.3.2.1.7. Specify AFU User Clock Timing

The clocks provided to the AFU by the `clocks` device interface are fixed in frequency except for the following user clocks:

- `uClk_usr`
- `uClk_usrDiv2`

The AFU specifies the frequency for `uClk_usr` in its platform configuration file using the following key:value pairs:

```
afu-image:clock-frequency-high:[<float-value>|”auto”|”auto-<float-value>”]

afu-image:clock-frequency-low:[<float-value>|”auto”|”auto-<float-value>”]
```

The above key:value pairs drive timing closure on the user clocks during AF compilation and are used to bound the frequency value configured in the PLL circuits of the target hardware platform that provides the user clocks through the `clocks` interface. The chosen frequency may vary in each compilation.

Setting the value field to a float number (e.g., `200.0` to specify 200 MHz) drives the AF generation process to close timing within the bounds set by the `low` and `high` keys and set in the AF's JSON metadata to specify the user clock PLL circuit frequency values.

Below is an example of `.json` file that sets the AFU `uClk` frequency to 300 MHz and `uClk_div2` to 200 MHz. It also operates the CCI-P interface on `uClk` instead of the default `pClk` clock domain.

```
{
   "version": 1,
   "afu-image": {
      "power": 0,
      "clock-frequency-high": "300",
      "clock-frequency-low": "200",
      "afu-top-interface":
         {
            "class": "ccip_std_afu"
            "module-ports" :
               [
                  {
                     "class": "cci-p",
                     "params":
                        {
                           "clock": "uClk_usr"
                        }
                  }
               ]
         },
      "accelerator-clusters":
         [
            {
               "name": "a_afu",
               "total-contexts": 1,
               "accelerator-type-uuid": "64e38106-4910-4488-ae90-94bfe46abfb3"
            }
         ]
   }
}
```

Below is an example of `.json` file that sets the AFU `uClk` frequency to 300 MHz or below and `uClk_div2` to 150 MHz or below. The auto setting allows the build flow to make adjustments to the frequency in the event that timing cannot be met. It also operates the CCI-P interface on `pClk` and the local SDRAM on the `uClk` clock domain instead of the default SDRAM domain.

```
{
   "version": 1,
   "afu-image": {
      "power": 0,
      "clock-frequency-high": "auto-300",
```

```
      "clock-frequency-low": "auto-150",
      "afu-top-interface":
         {
            "class": "ccip_std_afu_avalon_mm"
            "module-ports" :
               [
                  {
                     "class": "cci-p",
                     "params":
                        {
                           "clock": "pClk"
                        }
                  },
                  {
                     "class": "local-memory",
                     "params":
                        {
                           "clock": "uClk_usr"
                        }
                  }
               ]
         },
      "accelerator-clusters":
         [
            {
               "name": "b_afu",
               "total-contexts": 1,
               "accelerator-type-uuid": "233254b9-7db4-42a2-91db-1f1c53d12a76"
            }
         ]
   }
}
```

***Warning:*** AFU developers must ensure the hardware design meets timing by analyzing the static timing reports in Timing Analyzer. AFU developers must inform software developers of the maximum operating frequency (Fmax) of the user clocks so that the maximum frequency never exceeds. Exceeding the Fmax of any clock leads to indeterministic behavior of the accelerator and potentially the overall system.

*Note:* AFUs that use `uClk` or `uClk_div2` may not have accurate timing analysis results shown in the Intel Quartus Prime timing summary. This is a result of PLL tuning post compilation that occurs to determine if your AFU meets timing. You must either load the compiled design in Timing Analyzer or view the results of the `/build/output_files/timing_report` directory. Each of these methods accounts for the `uClk` and `uClk_div2` frequencies being adjusted. To determine if there are any failing paths in your design, you can inspect the file `/build/output_files/timing_report/clocks.sta.fail.summary`. If this file is empty then there are no timing failures in the design.

*Note:* There are two clock sources provided for the user clock. `Clk_1x` and `Clk_2x`. The `high` setting controls the `Clk_2x` and `low` setting controls the `Clk_1x`. There is a fixed relationship between these two clocks, except when `low` clock exceeds 300 MHz, then the `high` clock frequency matches the `low` clock frequency.

The `"auto"` setting enables the auto-timing closure mode during AF generation. The AF generation build process automatically converge on a maximum frequency of operation on the user clocks and generate AF JSON metadata to specify the auto-timing closure frequency achieved to OPAE tools.

You can combine the "auto" mode with an upper bound specification using the "auto-<float-value>" format (e.g., "auto-300" to specify auto-timing closure bounded to 300MHz).

For example, the hello_mem_afu sample AFU synchronizes all interfaces to uClk_usr and uses auto-timing closure mode:

```
$OPAE_PLATFORM_ROOT/hw/samples/hello_mem_afu/hw/rtl/
hello_mem_afu.json
```

### 5.3.2.2. Design the AFU

#### 5.3.2.2.1. Start with a Top-level Module Template

The top-level AFU interface requested in the platform configuration file defines the AFU's top-level RTL module port definition. Use the top-level module templates from OPAE or a corresponding AFU sample's top-level module as a reference for your AFU's top-level module port definition according to the top-level AFU interface requested in the platform configuration file.

OPAE top-level AFU RTL module templates are in the following location in the OPAE SDK: $OPAE_PLATFORM_ROOT/sw/<opae-version>/platforms/afu_top_ifc_db/

You can find the AFU samples at the following location in the OPAE SDK:$OPAE_PLATFORM_ROOT/hw/samples/

**Table 5. Associated Interface with Top-Level Module Template**

| Requested Top-Level Interface | Top-Level AFU RTL Module Templates |
|---|---|
| ccip_std_afu | Blank OPAE template: ccip_std_afu.sv.template<br>Sample AFU reference: hello_afu/hw/rtl/ccip_std_afu.sv |
| ccip_std_afu_avalon_mm | Blank OPAE template: ccip_std_afu_avalon_mm.sv.template<br>Sample AFU reference: hello_mem_afu/hw/rtl/ccip_std_afu.sv |

If you extend one of the above basic top-level AFU interfaces to add additional device interfaces (e.g., hssi), manually add the module ports for the added device interfaces. For example, the Ethernet sample AFUs extend the ccip_std_afu top-level AFU interface by adding an hssi device interface as shown in the following sample AFU top-level module RTL source files:

```
$OPAE_PLATFORM_ROOT/hw/samples/eth_e2e_e10/hw/rtl/ccip_std_afu.sv

$OPAE_PLATFORM_ROOT/hw/samples/eth_e2e_e40/hw/rtl/ccip_std_afu.sv

$OPAE_PLATFORM_ROOT/hw/samples/hssi_prbs/hw/rtl/ccip_std_afu.sv
```

#### 5.3.2.2.2. Including the Platform Device Interface Definitions

All RTL source in the AFU's implementation that references device interfaces defined by the OPAE Platform (e.g., cci-p, local-memory) must include the following Verilog header: `include "platform_if.vh"

The top-level AFU RTL module templates in OPAE and the sample AFUs all include `platform_if.vh`.

#### 5.3.2.2.3. Using the AFU UUID Header File

The AFU UUID should be specified in one place: the platform configuration file. The AFU implementation should extract the UUID from the following header file emitted by the PIM: `afu_json_info.vh`

The AFU should use the AFU_ACCEL_UUID macro defined within `afu_json_info.vh` to set the AFU's UUID in its implementation. For example, the `hello_afu` sample AFU includes the `afu_json_info.vh` and sets the AFU UUID using the `afu_json_info.vh` macro in the following SystemVerilog source file:

`$OPAE_PLATFORM_ROOT/hw/samples/hello_afu/hw/rtl/afu.sv`

#### 5.3.2.2.4. Clock Abstraction for the `cci-p` Device Interface

The PIM abstracts the clock and reset for the `cci-p` device interface passed to the AFU with the following Verilog macros:

- PLATFORM_PARAM_CCI_P_CLOCK
- PLATFORM_PARAM_CCI_P_RESET

The following RTL code snippet shows how to utilize the above macros to set the clock and reset signals in the AFU implementation for the `cci-p` interface:

```
`include “platform_if.vh”
logic clk;
assign clk = `PLATFORM_PARAM_CCI_P_CLOCK;
logic reset;
assign reset = `PLATFORM_PARAM_CCI_P_RESET;
```

This clock and reset abstraction enables compatibility for an AFU design's clock and reset connectivity on the `cci-p` device interface regardless of any clock-crossing requested in the platform configuration file.

The `hello_mem_afu` sample AFU provides an example for using the macro abstractions:

`$OPAE_PLATFORM_ROOT/hw/samples/hello_mem_afu/hw/rtl/ccip_std_afu.sv`

#### 5.3.2.2.5. Generating an AF Build Environment for Source Development

The OPAE SDK design flows for AFU development shown in this guide apply exactly as shown if the AFU design description is all RTL. However, if you want to design all or a portion of your AFU with Platform Designer subsystems or IP variants or want to add in-system debug components to the AFU design, it is helpful to generate an AF build environment for use in developing the AFU design description.

First, configure the build environment with a build configuration file as specified in the section *Generate the AF Build Environment*. The build configuration file is a text file that, at a minimum, consists of a single line that references the AFU's platform configuration file (`.json`). The file reference can be absolute or relative to the directory where the build configuration file resides.

Then, generate an AF build environment with the following command from an open terminal window:

```
afu_synth_setup –-source \

<path-to-build-configuration-file>/<build-configuration-filename> build_synth

cd build_synth/build

quartus dcp.qpf
```

Once the Intel Quartus Prime Pro Edition GUI opens, open the `dcp.qpf` project file and use the revisions feature to create a new revision based on the `afu_synth` revision and give it a unique name (e.g., `afu_dev`). Use the newly created revision as a workspace to develop the AFU's design description with tools such as Platform Designer or to add debug instances with tools such as Signal Tap. This method enables AFU design description development with high level, GUI-based tools in Quartus Prime Pro without corrupting the PR compilation revisions provided by the OPAE SDK for generating an AF.

### 5.3.2.3. AFU Design Guidelines

Follow these guidelines when designing a custom AFU:

#### 5.3.2.3.1. General Guidelines

- The OPAE SDK supports the following RTL language standards:
  - SystemVerilog 2005
  - VHDL 1993
- Reset and initialize all output registers to OPAE device interfaces.

**Related Information**

Intel Quartus Prime Pro Edition Handbook Volume 1 Design and Compilation

#### 5.3.2.3.2. Utilizing Clock Resources

The FIM provides several clock resources for use by AFUs. One set of clock resources is the user clock group, which includes `uClk_usr` and `uClk_usrDiv2`. Unlike `pClk` and its derivatives whose frequencies are fixed by the *CCI-P Specification*, the user clocks can be programmed for a range of frequencies supported by the AFU.

User clocks get provisioned by OPAE when an AF is loaded by the `fpgaconf` utility. When the `fpgaconf` utility loads an AF, it configures the PLL in the FIM that sources the user clocks with the frequency specified by a `key:value` pair found in the AF metadata generated by the packager utility. The desired user clock frequency `key:value` pair can be specified in a `.json` file or can be specified with a command line option (overrides entry in the `.json` file) to the packager utility. You can use the packager to generate AFs with unique metadata user clock frequency values for a single AFU PR bitstream.

The FIM reset resource, `pck_cp2af_softReset`, is not released until all clock resources are stable and locked, including the user clocks.

The AFU design must close timing on the user clocks at the maximum frequency to be supported by the AFU. Place associated clock timing constraints in a `.sdc` file and refer to the `.sdc` file in the AFU's build configuration file.

For usage information on the **Packager** utility and `.json` file metadata format, supported keyword parameters, and minimum metadata requirements, refer to the *packager* tab in the *Open Programmable Acceleration Engine (OPAE) Tools User Guide*.

**Related Information**

Open Programmable Acceleration Engine (OPAE) Tools Guide

#### 5.3.2.3.3. Interfacing with the FPGA Interface Manager (FIM)

The Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache interface (CCI-P) Reference Manual documents all the requirements for an AFU interfacing with the FIM using the CCI-P protocol. An AFU design must meet all the requirements specified in the following sections of the CCI-P reference manual:

- *CCI-P Interface*
- *AFU Requirements*
- *Device Feature List*

The above sections in the CCI-P reference manual include requirements unique to the Intel Xeon Processor with Integrated FPGA (referred to as Integrated FPGA Platform throughout this document) hardware platform, but most of the information applies to the Intel FPGA PAC. The notable differences between the two platforms are that the Intel FPGA PACs do not have a UPI channel or second PCIe link and no accelerator cache is implemented in the FIM.

The `hello_afu` example AFU included with the Acceleration Stack provides an example implementation of a simple Device Feature List that meets the requirements for an AFU as specified by the CCI-P reference manual. The `nlb_mode_0` and `dma_afu` example AFUs provide example implementations of more featured Device Feature Lists.

### 5.3.2.4. Partial Reconfiguration Design Guidelines

- You must generate the AF bitstream using the `$OPAE_PLATFORM_ROOT/bin/run.sh` script.
- Partial reconfiguration switches the PR region from one AFU to another AFU. Any software application exercising an AFU in the PR region should be terminated before initiating PR with OPAE to switch in a new AFU. This includes the remote debug feature.
- After PR, the default initial state of the registers and the contents of the MLABs and M20Ks in the PR region are indeterminate. To establish, a known initial condition for synchronous elements in the AFU, follow the guidelines below:
  - Design registers with reset logic sensitive to the FIU's `pck_cp2af_softReset` output. Do not rely on RTL initial value assignments or initial blocks.
  - Initialize MLAB and M20K contents using `.mif` files or RTL encoded values. Please refer to the *Intel Quartus Prime Pro Edition Handbook Volume 1 Design and Compilation* document for inferring or instantiating memory with initialized contents.

- The PR region must contain only core resources such as LABs, RAMs and DSPs. PLLs and Clock control blocks cannot be instantiated in the PR region.
- The placement and routing of a given AFU can vary between OPAE SDK releases and different OPAE hardware platform targets. Use seed sweeps for large resources or routing-intensive designs.
- If PR compilation results in timing violations in the FIM static region, retry PR compilation with a different fitter seed value.

**Related Information**

Intel Quartus Prime Pro Edition Handbook Volume 1 Design and Compilation

### 5.3.2.5. Specify the Build Configuration

The AFU configures the build environments generated by the PIM for simulation with ASE and AF generation with a build configuration file. The build configuration file is a text file created by the AFU designer to specify the following to the PIM:

- The AFU's platform configuration file (`.json`)
- List of simulation and synthesis source files:
  - RTL source (`.v`, `.sv`, `.vhd`)
  - Platform Designer subsystems (`.qsys`)
  - IP variants (`.ip`)
- List of additional source and constraints used during AF generation:
  - Signal Tap files (`.stp`)
  - Intel Quartus Prime Pro Edition settings (`.qsf`)
  - Timing constraints (`.sdc`)
- List of search paths at simulation or AF generation time
- List of include files at PIM build environment generation time
  - Reusable submodule (e.g., BBBs) build configuration files
- Verilog macro definitions

The build configuration file has the following format:

- Prefixes specify whether a reference is to a simulation or synthesis design file, include file or macro definition.
- File references can be absolute or relative to the directory containing the build configuration file.

For a full description of the build configuration file format and semantics, check the `rtl_src_config` command help:

```
$ rtl_src_config -h
```

See the following AFU samples located at `$OPAE_PLATFORM_ROOT/hw/samples` for examples of build configuration files:

- Simple examples:
  - `hello_afu/hw/rtl/filelist.txt`
  - `hello_mem_afu/hw/rtl/filelist.txt`
  - `hello_intr_afu/hw/rtl/filelist.txt`
- Examples with IP references and macro definitions:
  - `dma_afu/hw/rtl/filelist.txt`
  - `eth_e2e_e10/hw/rtl/filelist.txt`
  - `eth_e2e_e40/hw/rtl/filelist.txt`
- Examples with IP references, macro definitions and include references:
  - `nlb_mode_0/hw/rtl/filelist_mode_0.txt`
  - `nlb_mode_0_stp/hw/rtl/filelist_mode_0_stp`

The `afu_sim_setup` and `afu_synth_setup` use `rtl_src_config` internally to generate the appropriate fileset. Intel recommends that you use `afu_sim_setup` and `afu_synth_setup` instead of `rtl_src_config` directly.

### 5.3.2.6. Generate the ASE Build Environment

To generate a simulation build environment to verify your AFU with ASE, use the `afu_sim_setup` command:

```
afu_sim_setup --source \
<path-to-build-configuration-file>/<build-configuration-filename> <build-dir-
name>
```

For example, the following command sequence generates a simulation build environment for the `hello_afu` sample AFU in the `build_sim` directory:

```
cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu

afu_sim_setup --source hw/rtl/filelist.txt build_sim
```

As you iterate on the verification flow, you need to regenerate the simulation build environment with the `afu_sim_setup` command if either of the platform configuration or build configuration files have been modified according to design modifications. You can overwrite an existing simulation build directory by invoking the `afu_sim_setup` command with the `-f` command line option, or you can create a separate build environment by specifying a new target directory.

For a description of the full set of command line options and semantics, see the `afu_sim_setup` command help:

```
afu_sim_setup -h
```

The `afu_sim_setup` command calls the `rtl_src_config` command as part of the synthesis build environment generation process.

### 5.3.2.7. Verify the AFU with ASE

The ASE supports functional verification of AFU RTL code using host application C code developed for the OPAE API without the need for accelerator hardware. The ASE virtualizes the AFU's physical link with the host, models certain aspects of the OPAE

host memory model, and supports communication between the OPAE host application and supported RTL simulation tools used to emulate the AFU running on an actual OPAE-compliant accelerator hardware target.

ASE is useful for verifying your AFU's interoperability with the rest of the Acceleration Stack using a quick, iterative functional debug environment to minimize time spent in subsequent portions of the AFU development flow that involve more time-intensive steps (for example, PAR, timing closure). ASE also enables a more cost-efficient development environment by removing the dependency on accelerator hardware for early functional debug of AFU interoperability within the Acceleration Stack.

After using the `afu_sim_setup` to configure a simulation build environment, you are ready to start using ASE to verify your AFU. To know more about the simulations, refer to the *Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) Quick Start User Guide* and the *Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) User Guide*

**Related Information**

- Intel Accelerator Functional Unit Simulation Environment User Guide
- Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) Quick Start Guide
  For Intel PAC with Intel Arria 10 GX FPGA
- Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) Quick Start User Guide
  For Intel FPGA PAC D5005.

### 5.3.2.8. Generate the AF Build Environment

To generate a synthesis build environment to generate an AF, use the `afu_synth_setup` command as follows:

```
afu_synth_setup --source \
<path-to-build-configuration-file>/<build-configuration-filename> <build-dir \
-name>
```

For example, the following command sequence generates a synthesis build environment for the `hello_afu` sample AFU in the `build_synth` directory:

```
cd $OPAE_PLATFORM_ROOT/hw/samples/hello_afu

afu_synth_setup --source hw/rtl/filelist.txt build_synth
```

For a description of the full set of command line options and semantics, see the `afu_synth_setup` command help:

```
afu_synth_setup -h
```

The `afu_synth_setup` command calls the `rtl_src_config` command as part of the synthesis build environment generation process.

The `afu_synth_setup` uses `rtl_src_config` internally to generate the appropriate fileset. Intel recommends that you use `afu_synth_setup` instead of `rtl_src_config` directly.

You need to regenerate the build environment with the `afu_synth_setup` command if the platform configuration file has been modified. You also generally need to regenerate the build environment if the build configuration file has been modified except in the case where only the design file set has changed (source file addition, deletion, move). If only the design file set has been modified, reflect those changes in the build configuration file, and use the `rtl_src_config` command to update the `hw/afu.qsf` Quartus Prime Pro settings file in the existing build directory:

```
cd build_synth

rtl_src_config --qsf --rel build \

<reference-to-updated-build-configuration-file> >hw/afu.qsf
```

If AFU design modifications require that the synthesis build environment be regenerated, you can overwrite an existing synthesis build directory by invoking the `afu_synth_setup` command with the `-f` command line option, or you can create a separate build environment by specifying a new target directory.

Modifying existing RTL source files, Platform Designer subsystems or IP variants without changing their location as you develop the AFU does not require that the synthesis build environment be regenerated or that the Intel Quartus Prime Pro settings file be updated.

### 5.3.2.9. Generate the AF

From the synthesis build directory generated by `afu_synth_setup`, enter the following command from a terminal window to generate an AF for the target hardware platform:

```
$OPAE_PLATFORM_ROOT/bin/run.sh
```

The `run.sh` AF generation script generates the AF image with the same base filename as the AFU's platform configuration file with a `.gbs` suffix.

The `run.sh` script indicates the status of timing closure – make sure the generated AF has no hardware timing violations. Open the `dcp.qpf` Quartus project file in the Quartus Prime Pro GUI with the synthesis build project's `afu_fit` revision to view the details of the timing report and perform interactive timing analysis. The Intel Quartus Prime Pro Edition project directory is located in the `build` subdirectory of the synthesis build environment's top-level directory specified with the `afu_synth_setup` command.

# 6. AFU In-System Debug

The OPAE SDK provides a remote Signal Tap facility. Use remote Signal Tap to debug an AFU on a target hardware platform. The Signal Tap II Logic Analyzer, included in the Intel Quartus Prime Pro Edition, allows you to trigger on AFU signal events and capture traces of signals in your AFU design. The remote capability allows for control of trigger conditions and upload of captured signal traces from a networked workstation running the Signal Tap GUI.

Signal Tap is an in-system logic analyzer that you can use to debug FPGA logic. Conventional (non-remote) Signal Tap uses the physical FPGA JTAG interface and a USB cable to bridge the Intel Quartus Prime Signal Tap application running on a host system with the Signal Tap controller instances embedded in the FPGA logic. With Remote Signal Tap, you can achieve the same result without physically connecting to JTAG, which enables signal-level, in-system debug of AFUs deployed in servers where physical access is limited.

In addition to Signal Tap, the remote debug facility in OPAE supports the following in-system debug tools included with the Intel Quartus Prime Pro Edition:

- In-system Sources and Probes
- In-system Memory Content Editor
- Signal Probe
- System Console

This section describes how to generate an AF with remote Signal Tap enabled. This section then describes how to debug a user AFU using OPAE's **mmlink** utility, the System Console utility, and Intel Quartus Prime Pro Edition.

The `nlb_mode_0_stp` variation of the `nlb_mode_0` sample AFU is used to illustrate how to enable and use remote Signal Tap and can be found in the following location:

```
$OPAE_PLATFORM_ROOT/hw/samples/nlb_mode_0_stp/
```

**Related Information**

Design Debugging with the Signal Tap Logic Analyzer

## 6.1. Remote Signal Tap Setup and Use

### 6.1.1. Instrumenting the AFU Design for Signal Tap

To add Signal Tap instances and debug nodes to your AFU design, follow the procedure outlined in the Generating an AF Build Environment for Source Development on page 28 section to create a development revision. Once you have created a development revision, use the Signal Tap GUI to instrument the AFU for in-system debug as you normally would. For more information, see the related documentation for Signal Tap.

The `nlb_mode_0_stp` sample AFU has already been instrumented with Signal Tap and the `.stp` file is located in the following OPAE SDK directory: `$OPAE_PLATFORM_ROOT/hw/samples/nlb_mode_0_stp/hw/par/stp_basic.stp`.

### 6.1.2. Enable Remote Debug and Signal Tap

Signal Tap must be enabled in the AF generation flow by adding the following entries to the AFU's build configuration file:

```
+define+INCLUDE_REMOTE_STP
<path-relative-to-build-config-file>/<stp-filename>.stp
```

The `nlb_mode_0_stp` example already has the above settings added to its build configuration files:

- `$OPAE_PLATFORM_ROOT/hw/samples/nlb_mode_0_stp/hw/rtl/filelist_mode_0_stp.txt`

### 6.1.3. Generate the Remote Debug Enabled AF

After adding the above settings to the AFU's build configuration file, update the synthesis build environment and generate the remote debug enabled AF:

```
cd <path-to-synth-build-environment>

$ rtl_src_config --qsf --rel build <path-to-build-config-file>/<build-config-filename> >hw/afu.qsf

$OPAE_PLATFORM_ROOT/bin/run.sh
```

For *<path-to-synth-build-environment>*, use the directory path passed to the `afu_synth_setup` script when you created the synthesis build environment.

The `nlb_mode_0_stp` example already has a remote debug enabled AF: `$OPAE_PLATFORM_ROOT/hw/samples/nlb_mode_0_stp/bin/nlb_mode_0_stp.gbs`.

### 6.1.4. Prepare the Remote Debug Host

Copy the following files from the Acceleration Stack installation over to a convenient working directory on the remote debug host:

- The Signal Tap `.stp` file compiled with your AFU. In the case of the `nlb_mode_0_stp` example AFU, the `.stp` file is located in the Acceleration Stack installation as `$OPAE_PLATFORM_ROOT/hw/samples/nlb_mode_0_stp/hw/par/stp_basic.stp`.
- The following two files support establishing a connection on the remote debug host to the AFU Signal Tap instances on the Intel FPGA PAC. These files are part of the Acceleration Stack release – do not modify them.

  ```
  $OPAE_PLATFORM_ROOT/hw/remote_debug/mmlink_setup_profiled.tcl
  $OPAE_PLATFORM_ROOT/hw/remote_debug/remote_debug.sof
  ```

## 6.1.5. Running a Remote Debug Session

### 6.1.5.1. Connect to the AFU Target

**Follow these steps on the debug target host with the PAC installed:**

1. If not already done, load the Signal Tap-enabled AFU.

```
sudo fpgaconf $OPAE_PLATFORM_ROOT/hw/samples/nlb_mode_0_stp/bin/
nlb_mode_0_stp.gbs
```

2. Open a TCP port to accept incoming connection requests from remote debug hosts.

```
sudo mmlink -P 3333 -B <Bus number>
```

**Follow these steps on the remote debug host:**

1. Use **System Console** to connect to the debug target host's TCP port for Signal Tap debug connection on the target AFU. If the remote debug host is a Windows platform, open a command shell to run the below commands.

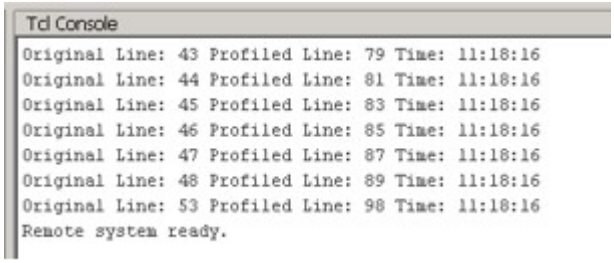
```
cd <path-to-debug-working-directory>

system-console --rc_script=mmlink_setup_profiled.tcl

remote_debug.sof <IP-address-of-debug-target-host> 3333
```

The above command assumes your PATH environment variable on the remote debug host is setup to point to the following location in the Intel Quartus Prime Pro Edition installation:

```
<installation-path>/<q-edition>/sopc_builder/bin
```

where <q-edition> is "quartus" for Intel Quartus Prime Pro Edition or Intel Quartus Prime Standard Edition. For an Intel Quartus Prime Programmer Edition installation, <q-edition> is qprogrammer.

2. After issuing the above commands, the System Console window appears. Wait for the "Remote system ready" message in the Tcl Console pane.



### 6.1.5.2. Using Signal Tap with a Remote Target Connection

Perform these steps on the remote debug host:

1. Invoke the Signal Tap GUI.
2. From **File ➤ Menu**, navigate to and open the `.stp` file you copied over from the "Prepare the Remote Debug Host" section when you were preparing the remote debug host for debugging the AFU.
3. Complete connecting to the Signal Tap controller instances in the target AFU by selecting **"System Console on ... Sld Hub Controller System"** from the Hardware drop-down option box in the **JTAG Chain Configuration** pane.
4. Wait for the **"JTAG ready"** response.

At this point, you are ready to perform in-system debug with the Signal Tap GUI in the same manner as with the conventional target connection method.

### 6.1.5.3. Stimulating the Target AFU for In-System Debug

Use host application C code software designed for the OPAE API to stimulate the AFU and verify proper operation within the Acceleration Stack. Leave the `mmlink` tool running in a separate terminal window on the debug target host while the remote debug host is connected. The `mmlink` process continuously output the status to the terminal window. Invoke OPAE host application or test software from their own terminal windows on the debug target host.

#### 6.1.5.3.1. Accessing the AFU in Shared Mode

When using OPAE application/test code running on the debug target host to stimulate the AFU for the purposes of in-system debug, both the `mmlink` tool and your host application/test code must have simultaneous access to the AFU. For this to happen, any user space code calls to the `fpgaOpen()` OPAE API function must pass the FPGA_OPEN_SHARED flag. The Acceleration Stack installation uses the FPGA_OPEN_SHARED flag with calls to `fpgaOpen()` in the source code for the `mmlink` tool and the `hello_fpga` sample application, which enables remote debug as delivered in the installation for the `nlb_mode_0_stp` example AFU stimulated by the `hello_fpga` sample application without modification.

Here is an example call to `fpgaOpen()` for shared access to the AFU:

```
fpgaOpen(afc_token, &afc_handle, FPGA_OPEN_SHARED);
```

Refer to the following sources in the Acceleration Stack installation for examples of using the FPGA_OPEN_SHARED flag:

```
$OPAE_PLATFORM_ROOT/sw/<opae-version>/tools/extra/mmlink/main.cpp
$OPAE_PLATFORM_ROOT/sw/<opae-version>/samples/hello_fpga.c
```

Any other sample applications included in the Acceleration Stack installation or host code of your own design must use the shared flag when used to stimulate the AFU during in-system remote debug where `mmlink` is required to run simultaneously.

### 6.1.5.4. Disconnect from the AFU Target

When you are finished debugging, follow these steps to gracefully end the debug connection:

First, on the remote debug host…

1. Save trace captures and exit the Signal Tap GUI.
2. From the **System Console** File menu, click exit to disconnect from the target AFU.

On the debug target host…

You can either keep the `mmlink` instance active and host debug sessions from other remote debug hosts, or you can terminate `mmlink` with the `<Ctl-C>` key sequence from its terminal window. If you choose to keep `mmlink` active, you can only debug the currently loaded AFU. If you want to debug another AFU, you must first terminate the active `mmlink` process. Before loading another AFU, make sure to terminate any OPAE host application code accessing the current AFU.

## 6.1.6. Remote Debug Guidelines

- The Signal Tap debug feature becomes non-functional when `mmlink` or **System Console** applications are closed.
- When performing PR, the AFU is non-existent and cannot be debugged. Therefore, **System Console** and `mmlink` applications should be terminated before attempting a partial reconfiguration of the AFU. Failing to do so might cause both PR and Signal Tap utilities to fail, taking the system into an unknown state. The system might have to be rebooted to restore the initial condition.
- The time to upload Signal Tap trace captures increases exponentially with sample depth. Intel recommends to use sample depths less than "2K" for a better Signal Tap user experience. Remote debug is still functional even for larger depths, but the time to upload the captured samples is significantly higher.
- **System Console** must be started after launching the `mmlink` application. If **System Console** returns an error, close the `mmlink` application, re-invoke `mmlink`, and launch **System Console** again.
- After generating an AF from an AFU with remote Signal Tap enabled, you may see cross clock timing failures between source and destination nodes in the following design hierarchy path:

  `fpga_top|inst_green_bs|auto_fab_0|alt_sld_fab_0|alt_sld_fab_0|auto_signaltap_auto_signaltap_0|sld_signaltap_inst|sld_signaltap_body|sld_signaltap_body`
  For any cross clock timing failures between source and destination nodes in the above design hierarchy path, add the following constraint applied between the nodes on each affected path to your `.sdc` timing constraint file:

```
set_false_path -from [get_registers <SOURCE_NODE_SDC_PATH>] -to
[get_registers <TO DESTINATION_NODE_SDC_PATH>]
```

The `.sdc` timing constraint file should be referenced in the build configuration file.

## 6.1.7. Troubleshooting Remote Debug Connections

If you get a **Failed to connect** message after invoking **System Console**, consider adding port tunneling. Do this when the debug target host is behind a firewall with respect to your remote debug host is not.

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.

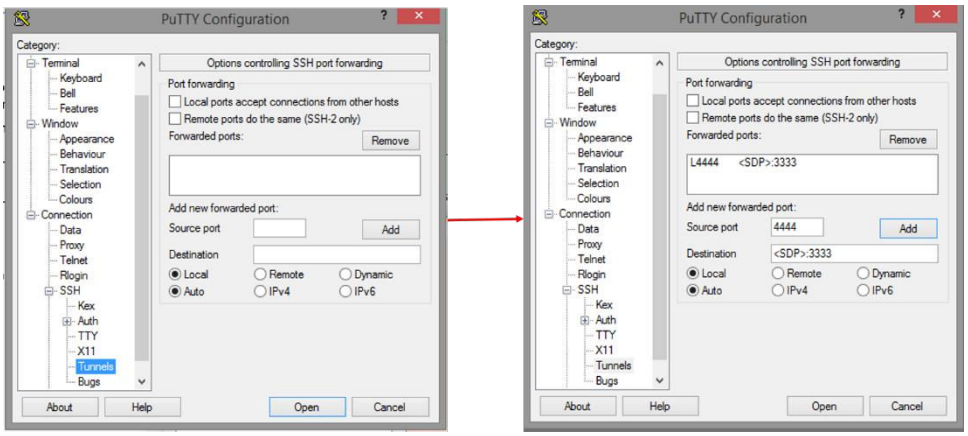
Refer to the following:

```
mmlink --port=3333
```

Setup port tunneling on the remote debug host. This example shows how to do so on a Windows remote debug host using PuTTY.

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.

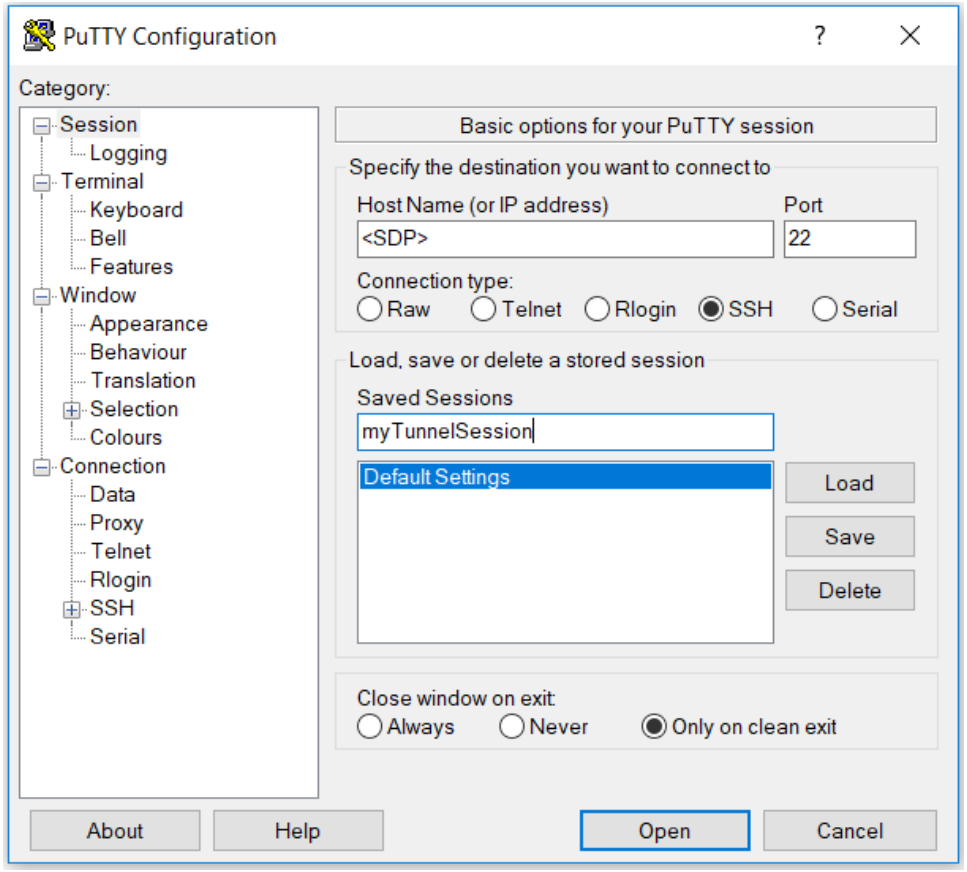
**Figure 5. SSH Tunneling with PuTTY**




Then, 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 6. 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. Open a Windows Command Window and issue the **system-console** command as shown in the "Save and Open the Tunneling Session" figure.

Run the "System Console with Port Forwarding" command:

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

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

# 7. Accelerator Functional Unit Developer's Guide for Intel FPGA Programmable Acceleration Card Archives

| Intel Acceleration Stack Version | User Guide (PDF) |
|---|---|
| 2.0 (For Intel FPGA PAC D5005) and 1.2 (For Intel FPGA PAC with Intel Arria 10 GX FPGA) | Accelerator Functional Unit (AFU) Developer's Guide for Intel FPGA Programmable Acceleration Card (Intel FPGA PAC) |
| 1.2 (For Intel FPGA PAC with Intel Arria 10 GX FPGA) | Accelerator Functional Unit (AFU) Developer's Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA |
| 1.1 (For Intel FPGA PAC with Intel Arria 10 GX FPGA) | Accelerator Functional Unit (AFU) Developer's Guide |

# 8. Document Revision History for Accelerator Functional Unit Developer's Guide for Intel FPGA Programmable Acceleration Card

| Document Version | Intel Acceleration Stack Version | Changes |
|---|---|---|
| 2020.07.20 | 2.0.1 (supported with Intel Quartus Prime Pro Edition Edition 19.2) and 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | Fixed broken link. |
| 2020.05.27 | 2.0.1 (supported with Intel Quartus Prime Pro Edition Edition 19.2) and 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | Corrected auto-programmable clock settings in section *Specify AFU User Clock Timing*. |
| 2020.03.02 | 2.0.1 (supported with Intel Quartus Prime Pro Edition Edition 19.2) and 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | Corrected a path to `main.cpp` in section *Accessing the AFU in Shared Mode*. |
| 2019.12.26 | 2.0.1 (supported with Intel Quartus Prime Pro Edition Edition 19.2) and 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | Added examples of `.json` files in section *Specify AFU User Clock Timing*. |
| 2019.11.04 | 2.0.1 (supported with Intel Quartus Prime Pro Edition Edition 19.2) and 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | • Modified the command to compile the sample AFU in section *Getting Started with Platform Configuration*.<br>• Added information about the usage of `platform_if.vh` in section *The `cci-p` Device Class*.<br>• Added new Ethernet sample AFU `hssi_prbs`.<br>• Clarified that new AFU design uses the `avalon_mm` interface in section *The `local-memory` Device Class*.<br>• Modified command to open the Intel Quartus Prime in section *Generating an AF Build Environment for Source Development*.<br>• Clarified the use of `rtl_src_config` file in section *Specify the Build Configuration*.<br>• Removed chapter *Hardware Platform OPAE Specifications* as the information is covered in *FIM Data Sheet*. |
| 2019.08.05 | 2.0 (supported with Intel Quartus Prime Pro Edition 18.1.2) and 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | • Added support for the Intel FPGA PAC D5005 platform in the current release.<br>• Updated section *FPGA Tools and IP Requirements*.<br>• Added new *Figure: High Level Block Diagram of AFU*. |

***continued...***

| Document Version | Intel Acceleration Stack Version | Changes |
|---|---|---|
| | | • Clarified the burst support information in section *The local-memory Device Class*.<br>• Updated *Figure: Save and Open the Tunneling Session*.<br>• Corrected a document title in section *Related Documentation*:<br>From *HSSI User Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA* to *Networking Interface for Open Programmable Acceleration Engine: Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA*. |
| 2019.05.06 | 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | Corrected the number of burst the local memory interface supports in section *The local-memory Device Class*. |
| 2019.04.09 | 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | Fixed broken PDF link in section *Accelerator Functional Unit (AFU) Developer's Guide for Intel FPGA Programmable Acceleration Card (Intel FPGA PAC) Archives*. |
| 2019.01.08 | 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | Minor edits. |
| 2018.12.20 | 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | Updated reference link to the Packager utility tab in *The PR Region* section. |
| 2018.12.04 | 1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1) | Maintenance release |
| 2018.08.06 | 1.1 (supported with Intel Quartus Prime Pro Edition 17.1.1) | Initial release |