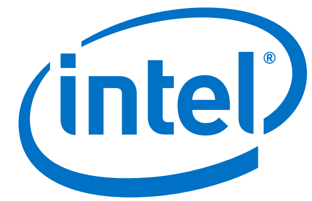


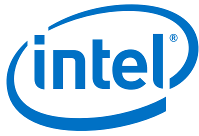
Intel Acceleration Stack for Intel® Xeon® CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual

Subscribe

Send Feedback

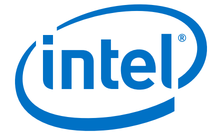
MNL-1092 | 2019.11.04

Latest document on the web: PDF | HTML



Contents

1. Acceleration Stack for Intel® Xeon® CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual.......... 3
1.1. About this Document..........3
1.1.1. Intended Audience..........3
1.1.2. Conventions..........3
1.1.3. Related Documentation.......... 4
1.1.4. Acronym List for Acceleration Stack for Intel Xeon® CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual.......... 5
1.1.5. Acceleration Glossary.......... 7
1.2. Introduction..........7
1.2.1. FPGA Interface Manager (FIM)..........9
1.2.2. Intel FPGA Interface Unit (FIU).......... 10
1.2.3. Memory and Cache Hierarchy.......... 13
1.3. CCI-P Interface..........15
1.3.1. Signaling Information.......... 16
1.3.2. Read and Write to Main Memory..........16
1.3.3. Interrupts.......... 18
1.3.4. UMsg.......... 18
1.3.5. MMIO Accesses to I/O Memory.......... 19
1.3.6. CCI-P Tx Signals..........21
1.3.7. Tx Header Format.......... 22
1.3.8. CCI-P Rx Signals..........26
1.3.9. Multi-Cache Line Memory Requests..........30
1.3.10. Byte Enable Memory Request (Intel FPGA PAC D5005).......... 31
1.3.11. Additional Control Signals..........35
1.3.12. Protocol Flow..........36
1.3.13. Ordering Rules..........38
1.3.14. Timing Diagram.......... 44
1.3.15. CCI-P Guidance..........46
1.4. AFU Requirements.......... 46
1.4.1. Mandatory AFU CSR Definitions..........47
1.4.2. AFU Discovery Flow..........49
1.4.3. AFU_ID.......... 49
1.5. Intel FPGA Basic Building Blocks.......... 50
1.6. Device Feature List..........50
1.7. Document Revision History for Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual..........55



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

1.1. About this Document

1.1.1. Intended Audience

The intended audience for this document is system engineers, platform architects, hardware, and software developers.

You must design the hardware AFU to be compliant with the CCI-P specification.

1.1.2. 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.

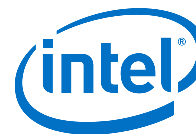
Intel Corporation. All rights reserved. Agilex, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

1.1.3. Related Documentation

Table 2. Related Documentation

Document	Description
Intel® Software Developers Manual	This document contains all three volumes of the *Intel 64 and IA-32 Architecture Software Development Manual:* Basic Architecture, Order Number 253665; Instruction Set Reference A-Z, Order Number 325383; System Programming Guide, Order Number 325384. Refer to all three volumes when evaluating your design needs.
Intel Virtualization Technology for Directed I/O Architecture Specification	This document describes the Intel Virtualization Technology for Directed I/O (Intel VT for Directed I/O); specifically, it describes the components supporting I/O virtualization as it applies to platforms that use Intel processors and core logic chipsets complying with Intel platform specifications.



1.1.4. Acronym List for Acceleration Stack for Intel Xeon® CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual

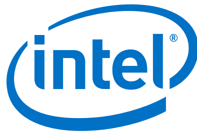
Table 3. Acronyms

The "A/B" column indicates whether the term applies to:

- **A:** Intel Xeon® Scalable Platform with Integrated FPGA, referred to as Integrated FPGA Platform throughout this document.
- **B:** Intel FPGA Programmable Acceleration Card (Intel FPGA PAC), referred to as Intel FPGA PAC throughout this document.
- **A, B:** Both packages

Acronyms	Expansion	A/B	Description
AF	Accelerator Function	A, B	Compiled Hardware Accelerator image implemented in FPGA logic that accelerates an application.
AFU	Accelerator Functional Unit	A, B	Hardware accelerator implemented in FPGA logic which offloads a computational operation for an application from the CPU to improve performance.
BBBs	Intel FPGA Basic Building Blocks	A, B	Intel FPGA Basic Building Blocks are defined as components that can be interfaced with the CCI-P bridge. For more information, refer to the Basic Building Blocks (BBB) for OPAE-managed Intel FPGAs web page.
CA	Caching Agent	A	A caching agent (CA) makes read and write requests to the coherent memory in the system. It is also responsible for servicing snoops generated by other Intel UltraPath Interconnect (Intel UPI) agents in the system.
CCI-P	Core Cache Interface	A, B	CCI-P is the standard interface AFUs use to communicate with the host.
CL	Cache Line	A, B	64-byte cache line
DFL	Device Feature Lists	A, B	DFL defines a structure for grouping like functionality and enumerating them.
FIM	FPGA Interface Manager	A, B	The FPGA hardware containing the FPGA Interface Unit (FIU) and external interfaces for memory, networking, etc. The Accelerator Function (AF) interfaces with the FIM at run time.
FIU	FPGA Interface Unit	A, B	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.
KiB	1024 bytes	A, B	The term KiB is for 1024 bytes and KB for 1000 bytes. When referring to memory, KB is often used and KiB is implied. When referring to clock frequency, kHz is used, and here K is 1000.
Mdata	Metadata	A, B	This is a user-defined field, which is relayed from Tx header to the Rx header. It may be used to tag requests with transaction ID or channel ID.
RdLine_I	Read Line Invalid	A, B	Memory Read Request, with FPGA cache hint set to invalid. The line is not cached in the FPGA, but may cause FPGA cache pollution.

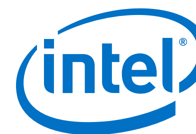
continued...



Acronyms	Expansion	A/B	Description
			Note: The cache tag tracks the request status for all outstanding requests on Intel Ultra Path Interconnect (Intel UPI). Therefore, even though RdLine_I is marked invalid upon completion, it consumes the cache tag temporarily to track the request status over UPI. This action may result in the eviction of a cache line, resulting in cache pollution. The advantage of using RdLine_I is that it is not tracked by CPU directory; thus it prevents snooping from CPU. *Note:* Cache functionality only applies to Intel Xeon Processor with Integrated FPGA.
RdLine-S	Read Line Shared	A	Memory read request with FPGA cache hint set to shared. An attempt is made to keep it in the FPGA cache in a shared state.
Rx	Receive	A, B	Receive or input from an AFU's perspective
SMBUS	System Management Bus	A	The System Management Bus (SMBUS) interface performs out-of-band temperature monitoring, configuration during the bootstrap process, and platform debug purposes.
Tx	Transmit	A, B	Transmit or output from an AFU's perspective
Upstream	Direction up to CPU	A, B	Logical direction towards CPU. Example: upstream port means port going to CPU.
UMsg	Unordered Message from CPU to AFU	A	An unordered notification with a 64-byte payload
UMsgH	Unordered Message Hint from CPU to AFU	A	This message is a hint to a subsequent UMsg. No data payload.
Intel UPI	Intel Ultra Path Interconnect	A	Intel's proprietary coherent interconnect protocol between Intel cores or other IP.
WrLine_I	Write Line Invalid	A, B	Memory Write Request, with FPGA cache hint set to Invalid. The FIU writes the data with no intention of keeping the data in FPGA cache.
WrLine_M	Write Line Modified	A	Memory Write Request, with the FPGA cache hint set to Modified. The FIU writes the data and leaves it in the FPGA cache in a modified state.
WrPush_I	Write Push Invalid	A	Memory Write Request, with the FPGA cache hint set to Invalid. The FIU writes the data into the processor’s Last Level Cache (LLC) with no intention of keeping the data in the FPGA cache. The LLC it writes to is always the LLC associated with the processor where the DRAM address is homed.

Related Information

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



1.1.5. Acceleration Glossary

Table 4. 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* accelerator card with an Intel FPGA PAC. Contains a FPGA Interface Manager (FIM) that connects to an Intel Xeon processor over PCIe bus.
Intel Xeon Scalable Platform with Integrated FPGA	Integrated FPGA Platform	A platform with the Intel Xeon and FPGA in a single package and sharing a coherent view of memory using the Intel Ultra Path Interconnect (UPI).

1.2. Introduction

CCI-P is a host interface bus for an Accelerator Functional Unit (AFU) with separate header and data wires. It is intended for connecting an AFU to an FPGA Interface Unit (FIU) within an FPGA. This document defines the CCI-P protocol and signaling interface. It includes definitions for request types, header formats, timing diagrams and memory model.

In addition to the CCI-P signaling and protocol, this document also describes:

1. Mandatory AFU registers required to design a CCI-P compliant AFU.
2. Device Feature Lists (DFLs)—A standard for register organization that promotes modular design and easy enumeration of AFU features from the software.
3. Intel FPGA Basic Building Blocks (BBBs)—An architecture of defining reusable FPGA libraries that may consists of hardware and software modules.

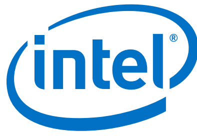
The CCI-P offers an abstraction layer that can be implemented on top of a variety of platform interfaces like PCIe and UPI, thereby enabling interoperability of CCI-P compliant AFU across platforms.

The table below summarizes the features unique to the CCI-P interface for the AFU.

Table 5. CCI-P Features

Feature	Description
MMIO Request—CPU read/write to AFU I/O Memory	• MMIO Read payload—4B, 8B • MMIO Write Payload—4B, 8B, 64B — MMIO writes could be combined by the x86 write combining buffer. — 64B MMIO writes require a CPU with capability of generation 64B Writes. — CPU for Integrated FPGA Platform can use AVX512 to generate 64B MMIO Write.
Memory Request	AFU read or write to memory.

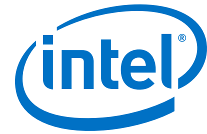
continued...



Feature	Description
	• Addressing Mode—Physical Addressing Mode • Addressing Width (CL aligned addresses)—42 bits (CL address) • Data Lengths—64 bytes (1 CL), 128 bytes (2 CLs), 256 bytes (4 CLs) • Byte Addressing—Not supported
FPGA Caching Hint (Integrated FPGA Platform only)	The AFU can ask the FIU to cache the CL in a specific state. For requests directed to VL0, FIU attempts to cache the data in the requested state, given as a hint. Except for WrPush_I, cache hint requests on VH0 and VH1 are ignored. *Note:* The caching hint is only a hint and provides no guarantee of final cache state. Ignoring a cache hint impacts performance but does not impact functionality. • <request>_I—No intention to cache • <request>_S—Desire to cache in shared (S) state • <request>_M—Desire to cache in modified (M) state
Virtual Channels (VC)	Physical links are presented to the AFU as virtual channels. The AFU can select the virtual channel for each memory request. • VL0—Low latency virtual channel (Mapped to UPI) (only for Integrated FPGA Platform). • VH0—High latency virtual channel. (Mapped to PCIe0). This virtual channel is tuned to handle large data transfers. • VH1—High latency virtual channel. (Mapped to PCIe1). This virtual channel is tuned to handle large data transfers (only for Integrated FPGA Platform). • Virtual Auto (VA)—FIU implements a policy optimized to achieve maximum cumulative bandwidth across all available physical links. — Latency—Expect to see high variance — Bandwidth—Expect to see high steady state bandwidth
UMsg (Integrated FPGA Platform only)	Unordered notification directed from CPU to AFU • UMsgs data payload—64B • Number of UMsg supported—8 per AFU
Response Ordering	Out of order responses
Upstream Requests	Yes

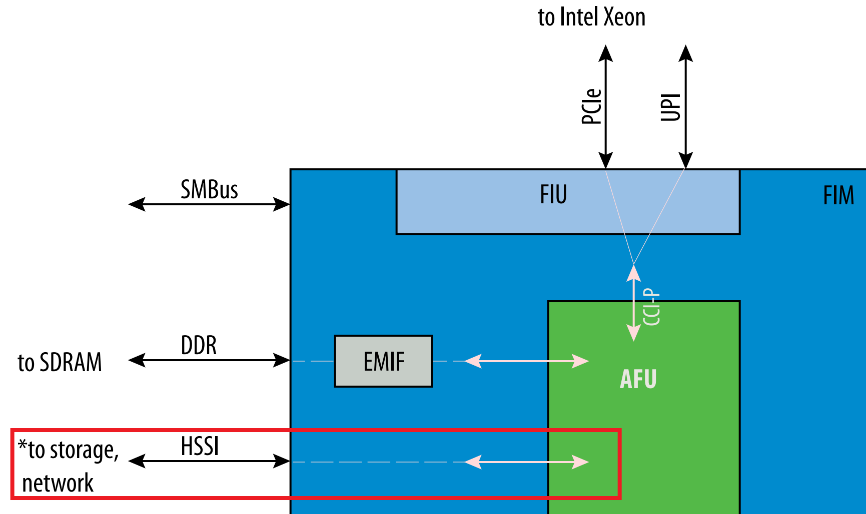
Related Information

Intel FPGA Basic Building Blocks on page 50



1.2.1. FPGA Interface Manager (FIM)

Figure 1. Overview of FPGA Blocks



The Intel FPGA Accelerator package consists of an FIM and an AFU. The FIM consists of the following:

- FIU
- EMIF for interfacing to external memory
- HSSI for external transceiver interfacing

For more information, refer to your platform for specific details about the FIM implementation.

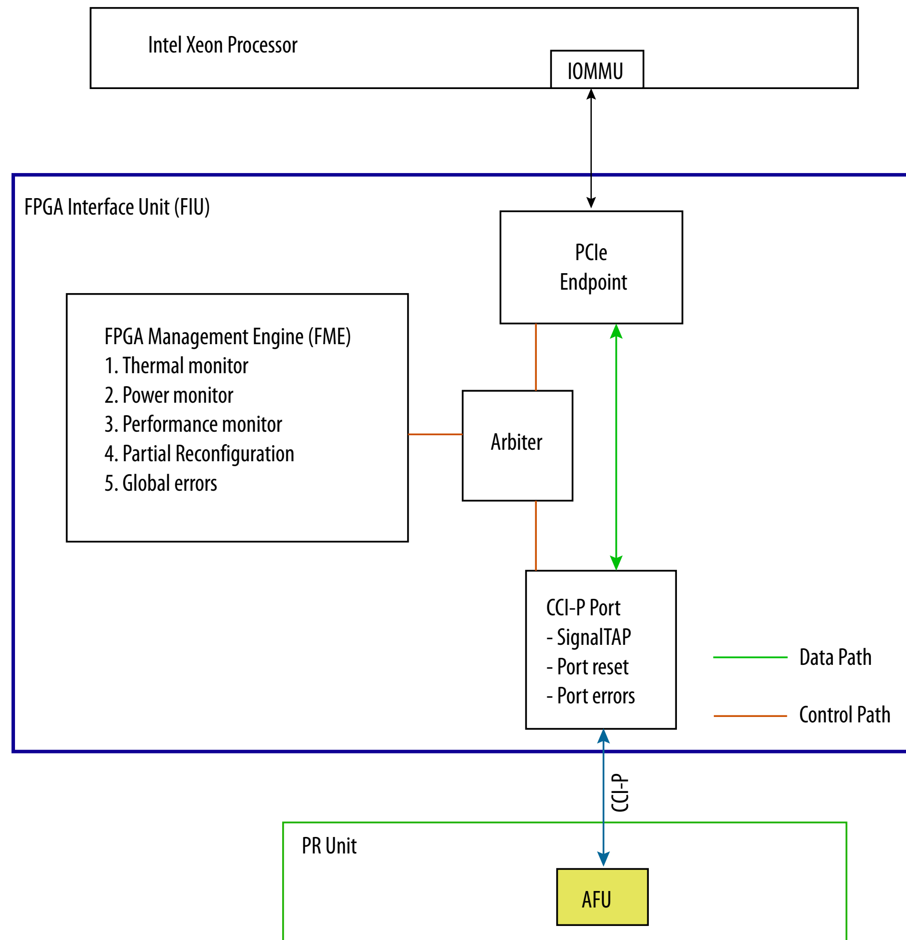
The FIU acts like a bridge between the AFU and the platform. Figure 1 on page 9 shows the FIU connection between the PCIe, SMBus for manageability, and the full UPI stack to the host. In addition, the FIM also owns all hard IPs on FPGA (for example PLLs), partial reconfiguration (PR) engine, JTAG atom, IOs, and temperature sensors. The FIM is configured first at boot up and persists until the platform power cycles, whereas the AFU can be dynamically reconfigured. Intel partial reconfiguration technology enables the dynamic reconfiguration capability, where the AFU is defined as a partial reconfiguration region and the FIM is defined as a static region. The interfaces between AFU and FIU provides hot plug capability to pause the traffic, and to re-enumerate the AFU after partial reconfiguration.

The FIM may present one or more interfaces to the AFU, depending on the platform capabilities. This document focuses on CCI-P, an interface for the AFU to communicate with the Intel Xeon processor. The CCI-P provides address space isolation and protection using Intel Virtual Technology for Directed I/O (Intel VT-d). The CCI-P has the same protections that are defined for a PCIe function. An AFU is a single function device from a PCIe enumeration and VT-d point of view.
The FIU may also implement manageability functions like error monitoring and reporting, power and temperature monitoring, configuration bootstrap, bitstream security flows, and remote debug to ease the deployment and management of FPGAs in a data center environment. In some implementations, FIU may also have an out-of-band communication interface to the board management controller (BMC).

1.2.2. Intel FPGA Interface Unit (FIU)

1.2.2.1. FIU for Intel FPGA PAC

Figure 2. FIU for Intel FPGA PAC Block Diagram



The Intel FPGA PAC connects to the Intel Xeon Processor over a PCIe physical link. The Intel FPGA PAC FIU block diagram in Figure 2 on page 10, shows only the blocks that map CCI-P to the PCIe link. It does not show FIM blocks for board-local memory going to the AFU. The FIU for Intel FPGA PAC has a simple function to map one physical link to CCI-P.

The Intel FPGA PAC FIU maps the CCI-P virtual channels VH0 and VA to the PCIe link. The virtual auto channel (VA) maps requests across all available channels on any platform optimally in order to achieve maximum bandwidth.

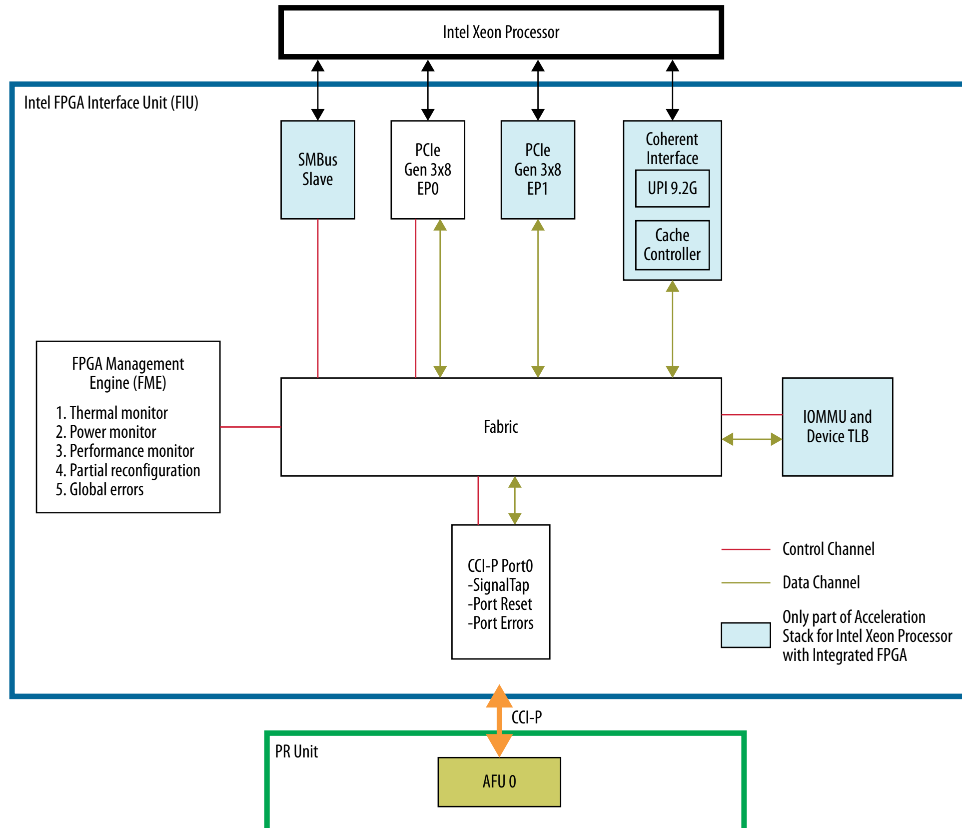
For more information about virtual channels, refer to the "CCI-P Features Summary" section. The downstream PCIe control path is address mapped to the FPGA management engine (FME), CCI-P port and AFU.

Send Feedback

The FME provides capabilities for error, performance, power, thermal monitoring and partial reconfiguration of the AFU. The CCI-P port module implements the per port reset, quiesce, error monitoring and remote debug using Signal Tap over network.

1.2.2.2. FIU for Intel Integrated FPGA Platform

Figure 3. FIU for Intel Integrated FPGA Platform Block Diagram



The Integrated FPGA Platform has three links connecting the FPGA to the processor: one Intel UPI coherent link and two PCIe Gen3x8 links. It is the function of the FIU to map these three links to the CCI-P interface, such that the AFU sees a single logical communication interface to the host processor with bandwidth equal to the aggregate bandwidth across the three links. Figure 1 on page 9 shows only the FIU logic associated with mapping UPI and PCIe links to CCI-P.

FIU implements the following functionality for providing the CCI-P mapping:

- Single logical upstream link: CCI-P maps the three physical links to four virtual channels. PCIe0 to VH0, PCIe1 to VH1, UPI to VL0 and all physical links to VA. An AFU using VA is agnostic of the physical links and it interfaces with a single logical link that can utilize the total upstream bandwidth available to the FPGA. VA implements a weighted de-multiplexer to route the requests to all of the physical links. To design a platform agnostic AFU, the VA virtual channel is the preferred choice.
- Single point of control: FIU registers a single control interface with the system software stack. All driver interactions to the FIU are directed to PCIe-0. The AFU is discovered and enumerated over PCIe-0.
- Single identity for VT-d provides a unified address space: All upstream requests use a single function number for address translation. For this reason, the Intel Xeon Scalable Platform with Integrated FPGA disables the IOMMU at PCIe-0 and PCIe-1 root ports and instead instantiates an IOMMU in FIU. This IOMMU is used for translating requests going upstream through all three physical links.

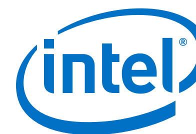
Similar to Intel FPGA PAC, the Integrated FPGA Platform also implements a full set of services provided by the FME and CCI-P ports to deploy and manage the FPGA.

1.2.2.3. Comparison of FIU Capabilities

The following table, provides a comparison of capabilities supported on the Intel FPGA PAC versus the Integrated FPGA Platform.

Table 6. Comparison of FIU Capabilities

FIU Capability	Supported on Intel FPGA PAC	Supported on Integrated FPGA Platform
Unified address space	Yes	
Intel VT-d for AFU	Yes	
Partial Reconfiguration	Yes	
Remote Debug	Yes	
FPGA Cache Size	N/A	128 KiB direct mapped
CCI-P		
Memory Mapped I/O (MMIO) read and write	Yes	
AFU interrupts to CPU	Yes	No
UMsg from CPU to AFU	No	Yes
CCI-P memory requests		
Data Transfer Size	64 bytes (1 CL), 128 bytes (2 CL), 256 bytes (4 CL)	
Addressing Mode	Physical Addressing Mode	
Addressing Width (CL aligned addresses)	42 bits	
Caching Hints	No	Yes
Virtual Channels	VA, VH0	VA, VH0, VH1, VL0

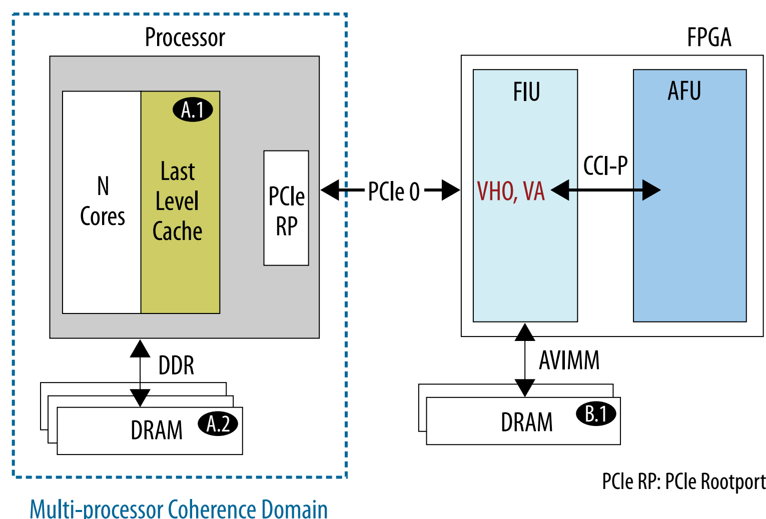


1.2.3. Memory and Cache Hierarchy

The CCI-P protocol provides a cache hint mechanism. Advanced AFU developers can use this mechanism to tune for performance. This section describes the memory and cache hierarchy for both the Intel FPGA PAC and Integrated FPGA Platform. The CCI-P provided control mechanisms are discussed in the "Intel FPGA PAC" and "Integrated FPGA Platform" sections, below.

Intel FPGA PAC

Figure 4. Intel FPGA PAC Memory Hierarchy



The above figure shows an Intel FPGA PAC memory and cache hierarchy in a single processor Intel Xeon platform. Intel FPGA PAC has two memory nodes:

- Processor **Synchronous Dynamic Random Access Memory (SDRAM)**, referred to as host memory
- FPGA attached **SDRAM**, referred to as local memory

AFU decides if the request must be routed to local memory or CPU memory.

Local Memory **(B.1)** is in a separate address space from host memory **(A.2)**. AFU requests, targeted to local memory, are always serviced directly by the **SDRAM** (denoted **(B.1)** in Figure 4 on page 13).

Note: There is no cache along the local memory access path.

AFU requests targeted to CPU memory over PCIe, can be serviced by the Processor-side, as shown in Figure 4 on page 13.

For the **Last Level Cache** (denoted **(A.1)**):

- A read request received has a lower latency than reading from the **SDRAM** (denoted **(A.2)**).
- A write request hint can be used to instruct the **Last Level Cache** how to treat the data written (for example: cacheable, non-cacheable, and locality).

If a request misses the **Last Level Cache**, it can be serviced by the **SDRAM**.

For more information, refer to the `WrPush_I` request in the CCI-P protocol definition.

Integrated FPGA Platform

Figure 5. Integrated FPGA Platform Memory Hierarchy

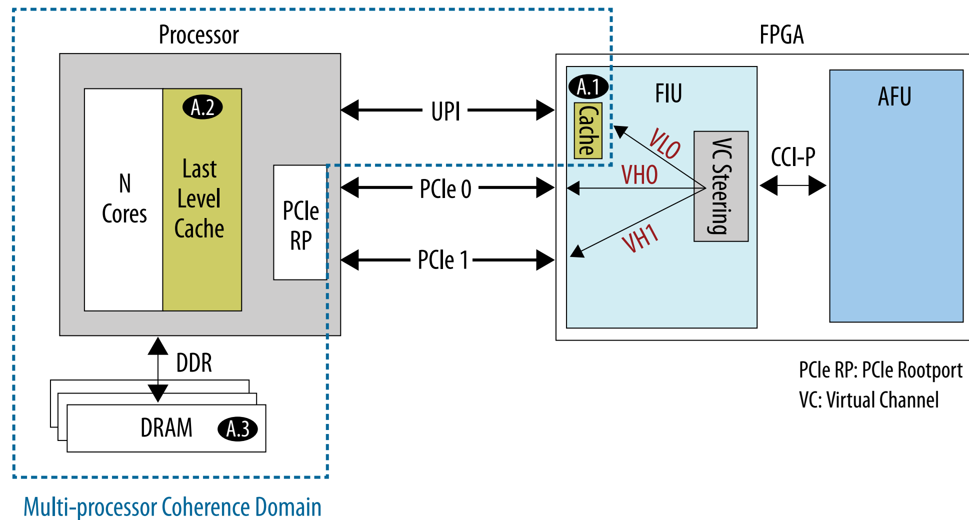
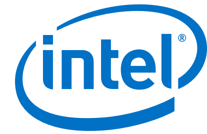


Figure 5 on page 14 shows the three level cache and memory hierarchy seen by an AFU in an Integrated FPGA Platform with one Intel Xeon processor. A single processor Integrated FPGA Platform has only one memory node, the Processor-side: **SDRAM** (denoted **(A.3)**). The Intel UPI coherent link extends the Intel Xeon processor's coherency domain to the FPGA as shown by the green dotted line in Figure 5 on page 14. A UPI caching agent keeps the FPGA cache in FIU, coherent with the rest of the CPU memory. An upstream AFU request targeted to CPU memory can be serviced by:

- FPGA Cache **(A.1)**—Intel UPI coherent link extends the Intel Xeon processor's coherency domain to the FPGA cache. Requests hitting in FPGA cache has the lowest latency and highest bandwidth. AFU requests that use VL0 virtual channel and VA requests that are selected to use UPI path, look up the FPGA cache first, and only upon a miss are sent off the chip to the processor.
- Processor-side cache **(A.2)**—A read request that hits the processor-side cache has higher latency than FPGA cache, but lower latency than reading from Processor **SDRAM**. A write request hint can be used to direct the write to processor-side cache. For more information, refer to `WrPush_I` request in CCI-P protocol definition.
- Processor **SDRAM (A.3)**—A request that misses the processor-side cache is serviced by the **SDRAM**.

The data access latencies increase from **(A.1)** to **(A.3)**.

Note: Most AFUs achieve maximum memory bandwidth by choosing the VA virtual channel, rather than explicitly selecting using VL0, VH0 and VH1. The VC steering logic implemented in the FIU has been tuned for the platform, it takes into account the physical link latency and efficiency characteristics, physical link utilization and traffic distribution to provide maximum bandwidth.



One limitation of the VC steering logic is that it does not factor the cache locality in the steering decision. The VC steering decision is made before the cache lookup. This means a request can get steered to VH0 or VH1 even though the cache line is in the FPGA cache. Such a request may incur an additional latency penalty, because the processor may have to snoop the FPGA cache in order to complete the request. If the AFU knows about the locality of accesses, then it may be beneficial to use VL0 virtual channel to exploit the cache locality.

Related Information

Avalon® Interface Specifications

1.3. CCI-P Interface

CCI-P implements to memory address spaces:

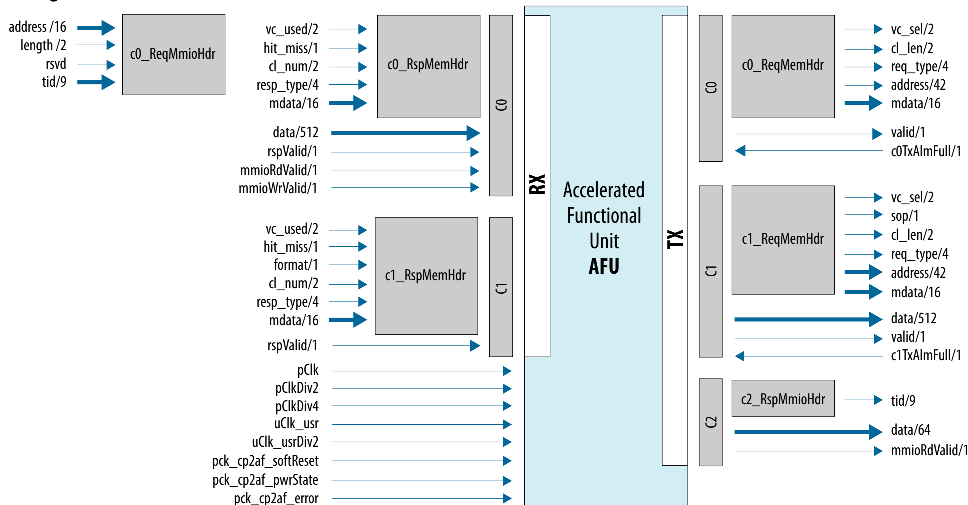
- Main memory
- Memory Mapped I/O (MMIO)

Table 7. CCI-P Memory Access Types

Memory Type	Description
Main Memory	Main memory is the memory attached to the processor and exposed to the operating system. Requests from the AFU to main memory are called upstream requests. Subsequent to this section, main memory is just referred to as memory.
Memory Mapped I/O	I/O memory is implemented as CCI-P requests from the host to the AFU. MMIO is typically used as AFU control registers. How this memory is implemented and organized is up to the AFU developer. The AFU may choose logic, M20Ks or MLABs. The CCI-P interface defines a request format to access I/O memory using memory mapped I/O (MMIO) requests. Requests from the processor to I/O memory are called downstream requests. The AFU's MMIO address space is 256 kB in size.

Figure 6. CCI-P Signals

This figure shows all CCI-P signals grouped into three Tx channels, two Rx channels and some additional control signals.



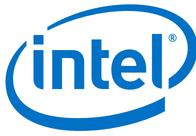


Table 8. CCI-P signals

Signal Type	Description
Tx/Rx	The flow direction is from the AFU point of view. Tx flows from AFU to FIU. Rx flows from FIU to AFU.
Channels	Grouping of signals that together completely defines the request or response.

1.3.1. Signaling Information

- All CCI-P signals must be synchronous to `pClk`.
- Intel recommends using the CCI-P structures defined inside `ccip_if_pkg.sv` file. This file can be found in the RTL package.
- All AFU input and output signals must be registered.
- AFU output bits marked as `RSVD` are reserved and must be driven to 0.
- AFU output bits marked as `RSVD-DNC`, are don’t care bits. The AFU can drive either 0 or 1.
- AFU input bits marked as RSVD must be treated as don’t care (X) by the AFU.
- All signals are active high, unless explicitly mentioned. Active low signals use a suffix `_n`.

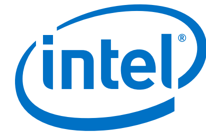
The figure below shows the port map for the `ccip_std_afu` module. The AFU must be instantiated under here. The subsequent sections explain the interface signals.

Figure 7. ccip_std_afu Port Map

```
$ module ccip_std_afu(
  // CCI-P Clocks and Resets
  input          logic          pClk,          //          CCI-P clock domain. Primary
                                               // interface clock
  input          logic          pClkDiv2,      //          CCI-P clock domain
  input          logic          pClkDiv4,      //          CCI-P clock domain
  input          logic          uClk_usr,      // User clock domain
  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
);
```

1.3.2. Read and Write to Main Memory

CCI-P defines upstream memory read and write requests for accessing the processor main memory using physical addresses. In a non-virtualized system, the AFU is expected to drive a host physical address. When in a virtualized system, the AFU is expected to drive a guest physical address. The addressing mode is transparent to the AFU hardware developer. The software application developer must ensure that software provides a physical address to the AFU.



CCI-P specification defines a weak memory consistency model for upstream memory requests.

For more information, refer to the "Ordering Rules" section.

Related Information

Ordering Rules on page 38

1.3.2.1. Reading from Main Memory

The AFU sends a memory read request over CCI-P `Channel 0 (C0)`, using `pck_af2cp_sTx.c0`; and receives the response over `C0`, using `pck_cp2af_sRx.c0`.

The `c0_ReqMemHdr` structure provides a convenient mapping from a flat bit-vector to read request fields. The AFU asserts the `pck_af2cp_sTx.c0.valid` signal and drives the memory read request on `hdr`. The `req_type` specifies the cache hint: (i) `RDLINE_I` to specify no caching and (ii) `RDLINE_S` for caching in shared state. The `mdata` field is a user-defined request ID that is returned unmodified with the response.

The `c0_RspMemHdr` structure provides a convenient mapping from flat bit-vector to response fields. The FIU asserts the `pck_cp2af_sRx.c0.resp_valid` signal and drives the read response and data on `hdr` and `data`, respectively. The `resp_type` is decoded to identify the response type: Memory Read or UMsg. Since the read response order is not guaranteed, you must define the `mdata` field to return the same value that was transmitted with the request.

For example, the AFUs may use `mdata` for routing the request and response internal to the AFU; or carrying information on the next triggered action.

1.3.2.2. Writing to Main Memory

The AFU sends memory write requests over CCI-P `Channel 1 (C1)`, using `pck_af2cp_sTx.c1`; and receives write completion acknowledgement responses over C1, using `pck_cp2af_sRx.c1`.

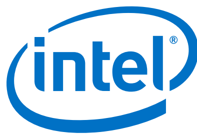
The `c1_ReqMemHdr` structure provides a convenient mapping from flat bit-vector to write request fields. The AFU asserts `pck_af2cp_sTx.c1.valid` signal and drives the memory write request and data on `hdr` and `data`, respectively. The `req_type` signal specifies the request type and caching hint:

- `WrLine_I` to specify no FPGA caching intent
- `WrLine_M` to specify intent to leave FPGA cache in M state
- `WrPush_I` for intent to cache in processor-side cache

The `c1_ReqMemHdr` structure also provides a mode field **pck_af2cp_sTx.c1.hdr.mode** that specifies which type of memory write request to issue. The two memory write request modes are as follows:

- `eMOD_CL` to specify a single or multi cache-aligned write
- `eMOD_BYTE` to specify a byte enable write

  Note: This memory request mode is not available for Intel PAC with Intel Arria® 10 GX FPGA.



The `c1_RspMemHdr` structure provides a convenient mapping from flat bit-vector to response fields. FIU asserts `pck_cp2af_sRx.c1.resp_valid` signal and drives the read response on `hdr`. The `resp_type` field is decoded to decode the response type: Memory write, Write Fence or Interrupt.

A `WrFence` is used to make the memory write requests globally visible. `WrFence` request follows the same flow as memory write requests, except that it does not accept a data payload and address.

For more information, refer to the Write Request header format in the *Tx Header Format*.

Related Information

Tx Header Format on page 22

1.3.3. Interrupts

Interrupts are not supported in the Integrated FPGA Platform.

The AFU sends an interrupt over Tx channel C1, using an interrupt ID, and receives the response over Rx channel C1.

An AFU should only have one interrupt ID issued at any given time. If the AFU does not wait for the response to return for the interrupt ID issued and issues another interrupt with the same ID, the host may not observe the arrival of the second interrupt. It is recommended for an interrupt request be serviced by software before the AFU issues an interrupt using the same interrupt ID.

1.3.4. UMsg

Attention: UMsg is only supported in the Integrated FPGA Platform.

UMsg provides the same functionality as a spin loop from the AFU, without burning the CCI-P read bandwidth. Think of it as a spin loop optimization, where a monitoring agent inside the FPGA cache controller is monitoring snoops to cache lines allocated by the driver. When it sees a snoop to the cache line, it reads the data back and sends an UMsg to the AFU.

UMsg flow makes use of the cache coherency protocol to implement a high speed unordered messaging path from CPU to AFU. This process consists of two stages as shown in Figure 8 on page 19.

The first stage is initialization, this is where SW pins the UMsg Address Space (UMAS) and shares the UMAS start address with the FPGA cache controller. Once this is done, the FPGA cache controller reads each cache line in the UMAS and puts it as shared state in the FPGA cache.

The second stage is actual usage, where the CPU writes to the UMAS. A CPU write to UMAS generates a snoop to FPGA cache. The FPGA responds to the snoop and marks the line as invalid. The CPU write request completes, and the data become globally visible. A snoop in UMAS address range, triggers the Monitoring Agent (MA), which in turn sends out a read request to CPU for the Cache Line (CL) and optionally sends out an UMsg with Hint (UMsgH) to the AFU. When the read request completes, an UMsg with 64B data is sent to the AFU.

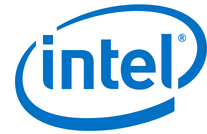
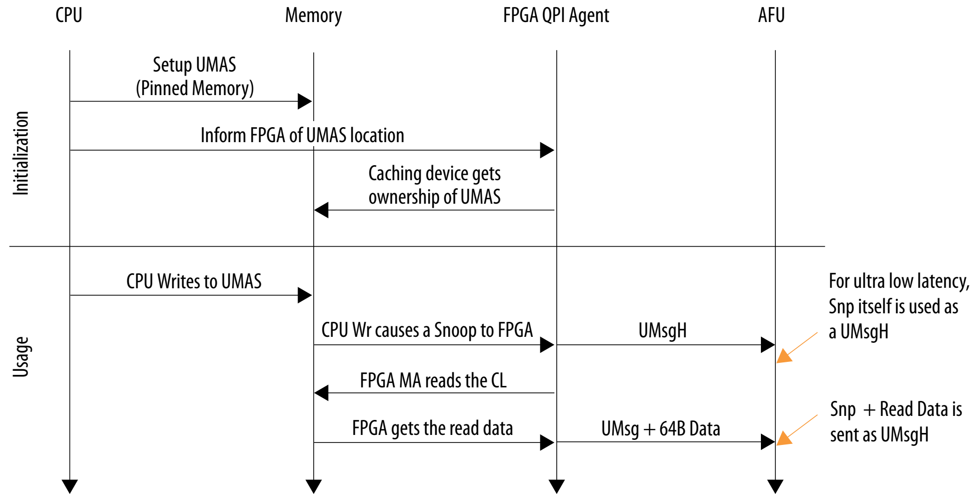


Figure 8. UMsg Initialization and Usage Flow



Functionally, UMsg is equivalent to a spin loop or a monitor and mwait instruction on an Intel Xeon processor.

Key characteristics of UMsgs:

- Just as spin loops to different addresses in a multi-threaded application have no relative ordering guarantee, UMsgs to different addresses have no ordering guarantee between them.
- Every CPU write to a UMAS CL, may not result in a corresponding UMsg. The AFU may miss an intermediate change in the value of a CL, but it is guaranteed to see the newest data in the CL. Again it helps to think of this like a spin loop: if the producer thread updates the flag CL multiple times, it is possible that polling thread misses an intermediate change in value, but it is guaranteed to see the newest value.

Below is an example usage. Software updates to a descriptor queue pointer that may be mapped to an UMsg. The pointer is always expected to increment. The UMsg guarantees that the AFU sees the final value of the pointer, but it may miss intermediate updates to the pointer, which is acceptable.

1. The UMsg uses the FPGA cache, as a result it can cause cache pollution, a situation in which a program unnecessarily loads data into the cache and causes other needed data to be evicted, thus degrading performance.
2. Because the CPU may exhibit false snooping, UMsgH should be treated as a hint. That is, you can start a speculative execution or pre-fetch based on UMsgH, but you should wait for UMsg before committing the results.
3. The UMsg provides the same latency as an AFU read polling using RdLine_S, but it saves CCI-P channel bandwidth which can be used for read traffic.

1.3.5. MMIO Accesses to I/O Memory

The CCI-P defines MMIO read and write requests for accessing the AFU register file. MMIO requests are routed from the CPU to the AFU over a single PCIe channel.

MMIO Reads

The AFU receives an MMIO read request over `pck_cp2af_sRx.c0`. The CCI-P asserts `mmioRdValid` and drives the MMIO read request on `hdr`. The `c0_ReqMmioHdr` structure provides a convenient mapping from a flat bit-vector to MMIO request fields – `{address, length, tid}`.

The AFU drives an MMIO read response over `pck_af2cp_sTx.c2`. The AFU asserts `mmioRdValid` and drives the response header and data on `hdr` and `data`, respectively. The AFU is expected to return the request `tid` with the corresponding response it used to associate the response with request.

The following list describes key attributes of a CCI-P MMIO read request:

- Data lengths supported are 4 bytes and 8 bytes
- Response length must match the request length. For example, it is illegal to return two 4-byte responses to an 8-byte MMIO read request
- Maximum number of outstanding MMIO read requests is limited to 64
- MMIO reads to undefined AFU registers should still return a response

MMIO Writes

The AFU receives an MMIO write request over `pck_cp2af_sRx.c0`. The CCI-P asserts `mmioWrValid` and drives the MMIO write request header and data on `hdr` and `data`, respectively. The `c0_ReqMmioHdr` structure provides a convenient mapping from a flat bit-vector to MMIO request fields – `{address, length, tid}`. The MMIO write request is posted and no response is expected from the AFU.

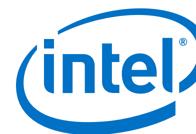
The data lengths supported are 4 bytes, 8 bytes, and 64 bytes.

Note: Not supported on all Intel Xeon platforms.

Implementation Note for all MMIO Accesses

The following is a list of important considerations when designing an AFU MMIO register file:

- It is mandatory for the AFU to support 8-byte accesses in order to implement the DFH.
- Support for 4-byte MMIO accesses is optional. The AFU developer can coordinate with the software application developer to avoid 4-byte accesses.
- The AFU can accept MMIO requests as they arrive, consecutively, without any delays.
- Unaligned MMIO accesses results in an error. The software application developer must ensure that the MMIO address is aligned to the request length. For example: 8-byte MMIO request byte address should be a multiple of 8, which means byte address[2:0] should be 0.



1.3.6. CCI-P Tx Signals

Figure 9. Tx Interface Structure Inside `ccip_if_pkg.sv`

```
$ typedef struct packed {
    t_if_ccip_c0_Tx      c0;
    t_if_ccip_c1_Tx      c1;
    t_if_ccip_c2_Tx      c2;
} t_if_ccip_Tx;
```

There are three Tx channels:

The C0 and C1 Tx channels are used for memory requests. Both C0 and C1 Tx channels have independent flow control. The C0 Tx channel is used for memory read requests; the C1 Tx channel is used for memory write requests.

The C2 Tx channel is used to return MMIO Read response to the FIU. The CCI-P port guarantees to accept responses on C2; therefore, it has no flow control.

Figure 10. Tx Channel Structure Inside `ccip_if_pkg.sv`

```
// Channel 0 : Memory Reads
typedef struct packed {
    t_ccip_c0_ReqMemHdr  hdr;            // Request Header
    logic                valid;          // Request Valid
} t_if_ccip_c0_Tx;
// corresponding AlmostFull inside t_if_ccip_Rx.c0TxAlmFull

// Channel 1 : Memory Writes
typedef struct packed {
    t_ccip_c1_ReqMemHdr  hdr;            // Request Header
    t_ccip_clData        data;           // Request Data
    logic                valid;          // Request Wr Valid
} t_if_ccip_c1_Tx;
// corresponding AlmostFull inside t_if_ccip_Rx.c1TxAlmFull

// Channel 2 : MMIO Read response
typedef struct packed {
    t_ccip_c2_RspMmioHdr    hdr;         // Response Header
    logic                   mmioRdValid; // Response Read Valid
    t_ccip_mmioData         data;        // Response Data
} t_if_ccip_c2_Tx;
```

Each Tx channel has a valid signal to qualify the corresponding header and data signals within the structure.

The following tables describe the signals that make up the CCI-P Tx interface.

Table 9. Tx Channel Description for Channel 0

Signal	Width (bits)	Direction	Description
`pck_af2cp_sTx.c0.hdr`	74	Output	Channel 0 request header. Refer to Table 18 on page 26.
`pck_af2cp_sTx.c0.valid`	1	Output	When set to 1, it indicates channel 0 request header is valid.
`pck_cp2af_sRx.c0TxAlmFull`	1	Input	When set to 1, Tx Channel 0 is almost full. After this signal is set, AFU is allowed to send a maximum of 8 requests. When set to 0, AFU can start sending requests immediately.

Table 10. Tx Channel Description for Channel 1

Signal	Width	Direction	Description
pck_af2cp_sTx.c1.hdr	80	Output	Channel 1 request header. Refer to Table 12 on page 22.
pck_af2cp_sTx.c1.data	512	Output	Channel 1 data
pck_af2cp_sTx.c1.valid	1	Output	When set to 1, it indicates channel 1 request header and data is valid.
pck_cp2af_sRx.c1TxAlmFull	1	Input	When set to 1, Tx Channel 1 is almost full. After this signal is set, AFU is allowed to send a maximum of 8 requests or data. When set to 0, AFU can start sending requests immediately.

Table 11. Tx Channel Description for Channel 2

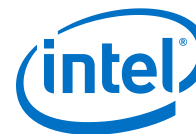
Signal	Width (bits)	Direction	Description
pck_af2cp_sTx.c2.hdr	9	Output	Channel 2 response header. Refer to Table 12 on page 22.
pck_af2cp_sTx.c2.mmioRdValid	1	Output	When set to 1, indicates Channel 2 response header and data is valid.
pck_af2cp_sTx.c2.data	64	Output	Channel 2 data. MMIO Read Data that AFU returns to FIU. For 4 bytes reads, data must be driven on bits [31:0]. For 8 bytes reads, AFU must drive one 8 bytes data response. Response cannot be split into two 4 bytes responses.

1.3.7. Tx Header Format

Table 12. Tx Header Field Definitions

Field	Description
mode	Memory Access Mode • eMOD_CL (1'b0)—cache aligned write. Enables 1, 2, or 4 cache line writes to host memory, specified by cl_len. • eMOD_BYTE(1'b1)—byte aligned write. Write a contiguous subset of a line to host memory, as specified by the byte_start and byte_len fields. **Note:** This memory request mode is not available for Intel PAC with Intel Arria 10 GX FPGA. *Note:* When set to eMOD_BYTE, the cache length (cl_len) must be set to 0, indicating a single cache line write. When set to eMOD_CL, byte_len and byte_start must be set to 0. *Note:* You cannot change modes in the middle of a multi cache line write. *Note:* This field is RSVD0 for Intel FPGA PAC N3000 and Intel PAC with Intel Arria 10 GX FPGA
byte_start	Byte Start Index for Byte Access Mode • Indicates index of first byte in the 512-bit TX Data bus to write to host memory. • When mode = eMOD_CL, byte_start must be set to 0. • When mode = eMOD_BYTE, byte_start is set in byte_enable mode and the legal range is 0 - 63. *Note:* This field is RSVD0 for Intel FPGA PAC N3000 and Intel PAC with Intel Arria 10 GX FPGA

continued...

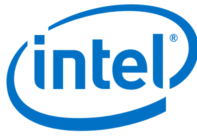


Field	Description
byte_len	Byte Length for Byte Access Mode (mode = eMOD_BYTE) • Indicates how many bytes to write to host memory. • byte_len—specifies the number of bytes to the left (most significant) of the byte_start index to include a memory request in Byte Access Mode. • When mode = eMOD_CL, byte_len must be set to 0. • When mode = eMOD_BYTE, byte_len is set in byte enable mode and the legal range is 1 - 63. *Note:* This field is RSVD0 for Intel FPGA PAC N3000 and Intel PAC with Intel Arria 10 GX FPGA
mdata	Metadata: user defined request ID that is returned unmodified from request to response header. For multi-CL writes on C1 Tx, mdata is only valid for the header when sop=1.
tid	Transaction ID: AFU must return the tid MMIO Read request to response header. It is used to match the response against the request.
vc_sel	Virtual Channel selected • 2’h0 – VA • 2’h1 – VL0 • 2’h2 – VH0 • 2’h3 – VH1 All CLs that form a multi-CL write request are routed over the same virtual channel.
req_type	Request types listed in Table 13 on page 23.
sop	Start of Packet for multi-CL memory write • 1’h1 – marks the first header. Must write in increasing address order. • 1’h0 – subsequent headers
cl_len	Length for memory requests • 2’h0 – 64 bytes (1 CL) • 2’h1 – 128 bytes (2 CLs) • 2’h3 – 256 bytes (4 CLs) *Note:* When mode = eMOD_BYTE, cl_len must be 2’h0.
address	64-byte aligned Physical Address, that is, byte_address>>6 The address must be naturally aligned with regards to the cl_len field. For example for cl_len=2’b01, address[0] must be 1'b0, similarity for cl_len=2'b11, address[1:0] must be 2'b00.

Table 13. AFU Tx Request Encodings and Channels Mapping

Request Type	Encoding	Data Payload	Description	Header Format
t_if_ccip_c0_tx: enum t_ccip_c0_req				
eREQ_RDLINE_I	4’h0	No	Memory read request with no intention to cache.	C0 Memory Request Header. Refer to Table 14 on page 24.
eREQ_RDLINE_S	4’h1	No	Memory read request with caching hint set to Shared.	
t_if_ccip_c1_tx: enum t_ccip_c1_req				
eREQ_WRLINE_I	4’h0	Yes	Memory write request with no intention of keeping the data in FPGA cache. Does not keep the cache line in the FPGA cache and does not provide guidance on the CPU-side caching.	C1 Memory Request Hdr. Refer to Table 15 on page 25.

continued...



Request Type	Encoding	Data Payload	Description	Header Format
			Note: The CPU is responsible for the CPU-side caching.	
eREQ_WRLINE_M	4'h1	Yes	Memory write request with caching hint set to Modified.	
eREQ_WRPUSH_I	4'h2	Yes	Memory Write Request, with caching hint set to Invalid. FIU writes the data into the processor's last level cache (LLC) with no intention of keeping the data in FPGA cache. The LLC it writes to is always the LLC associated with the processor where the **SDRAM** address is homed. Does not keep the cache line in the FPGA cache, but pushes the line into the CPU LLC.	
eREQ_WRFENCE	4'h4	No	Memory write fence request.	Fence Hdr. Refer to Table 16 on page 25.
eREQ_INTR	4'h6	No	Interrupt	Interrupt Hdr. Refer to Table 17 on page 25
t_if_ccip_c2_tx – does not have a request type field				
MMIO Rd	NA	Yes	MMIO read response	MMIO Rd Response Hdr Refer to Table 18 on page 26.

All unused encodings are considered RSVD0.

Table 14. C0 Read Memory Request Header Format Structure; t_ccip_c0_ReqMemHdr

Bit	Number of Bits	Field
[73:72]	2	vc_sel
[71:70]	2	RSVD
[69:68]	2	cl_len
[67:64]	4	req_type
[63:58]	6	RSVD
[57:16]	42	address
[15:0]	16	mdata

To determine if byte enable is available on your platform, you must use Verilog to verify CCIP_ENCODING_HAS_BYTE_WR is defined and the parameter ccip_cfg_pkg::BYTE_EN_SUPPORTED is non-zero. These two conditions must be true in order for byte enable to be available.

Note: When CCIP_ENCODING_HAS_BYTE_WR is defined, byte_start and byte_len are available. This does not mean that byte enable is available on your platform.

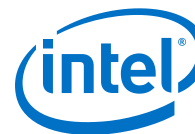


Table 15. C1 Write Memory Request Header Format Structure: t_ccip_c1_ReqMemHdr

Bit	Number of Bits	Field SOP=1	Field SOP=0
[79:74]	6	byte_len (must be 0 when mode=eMOD_CL) *Note:* This field is RSVD0 for Intel FPGA PAC N3000 and Intel PAC with Intel Arria 10 GX FPGA	byte_len (must be 0 when sop=0) *Note:* This field is RSVD0 for Intel FPGA PAC N3000 and Intel PAC with Intel Arria 10 GX FPGA
[73:72]	2	vc_sel	RSVD-DNC
[71]	1	sop=1	sop=0
[70]	1	mode *Note:* This field is RSVD0 for Intel FPGA PAC N3000 and Intel PAC with Intel Arria 10 GX FPGA	mode (must be eMOD_CL when sop=0) *Note:* This field is RSVD0 for Intel FPGA PAC N3000 and Intel PAC with Intel Arria 10 GX FPGA
[69:68]	2	cl_len	RSVD-DNC
[67:64]	4	req_type	req_type
[63:58]	6	byte_start (must be 0 when mode=eMOD_CL) *Note:* This field is RSVD0 for Intel FPGA PAC N3000 and Intel PAC with Intel Arria 10 GX FPGA	byte_start (must be 0 when sop=0) *Note:* This field is RSVD0 for Intel FPGA PAC N3000 and Intel PAC with Intel Arria 10 GX FPGA
[57:18]	40	address[41:0]	RSVD-DNC
[17:16]	2		address[1:0]
[15:0]	16	mdata	RSVD-DNC

Table 16. C1 Fence Header Format Structure: t_ccip_c1_ReqFenceHdr

Bit	Number of Bits	Field
[79:74]	6	RSVD
[73:72]	2	vc_sel
[71:68]	4	RSVD
[67:64]	4	req_type
[63:16]	48	RSVD
[15:0]	16	mdata

Table 17. C1 Interrupt Header Format Structure: t_ccip_c1_ReqIntrHdr (Intel FPGA PAC only)

Bit	Number of Bits	Field
[79:74]	6	RSVD
[73:72]	2	vc_sel
[71:68]	4	RSVD

continued...

Bit	Number of Bits	Field
[67:64]	4	req_type
[63:12]	62	RSVD
[1:0]	2	id

Table 18. C2 MMIO Response Header Format

Bit	Number of Bits	Field
[8:0]	9	tid

1.3.8. CCI-P Rx Signals

Figure 11. Rx Interface Structure Inside ccip_if_pkg.sv

```
typedef struct packed {
    logic                c0TxAlmFull;    //  C0 Request Channel Almost Full
    logic                c1TxAlmFull;    //  C1 Request Channel Almost Full

    t_if_ccip_c0_Rx      c0;
    t_if_ccip_c1_Rx      c1;
} t_if_ccip_Rx;
```

There are two Rx channels:

- Channel 0 interleaves memory responses, MMIO requests and UMsgs.
- Channel 1 returns responses for AFU requests initiated on Tx Channel 1.

The c0TxAlmFull and c1TxAlmFull signals are inputs to the AFU. Although they are declared with the Rx signals structure, they logically belong to the Tx interface and so were described in the previous section.

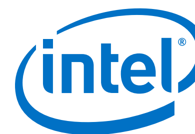
Rx Channels have no flow control. The AFU must accept responses for memory requests it generated. The AFU must pre-allocate buffers before generating a memory request. The AFU must also accept MMIO requests.

Rx Channel 0 has separate valid signals for memory responses and MMIO requests. Only one of those valid signals can be set in a cycle. The MMIO request has a separate valid signal for MMIO Read and MMIO Write. When either mmioRdValid or mmioWrValid is set, the message is an MMIO request and should be processed by casting t_if_ccip_c0_Rx.hdr to t_ccip_c0_ReqMmioHdr.

Table 19. Rx Channel Signal Description for Channel 0

Signal	Width (bits)	Direction	Description
pck_cp2af_sRx.c0.hdr	28	Input	Channel 0 response header or MMIO request header. Refer to Table 21 on page 27.
pck_cp2af_sRx.c0.data	512	Input	Channel 0 Data bus Memory Read Response and UMsg: • Returns 64 bytes data MMIO Write Request: • For 4 bytes write, data driven on bits [31:0] • For 8 bytes write, data driven on bits [63:0]

continued...



Signal	Width (bits)	Direction	Description
pck_cp2af_sRx.c0.rspValid	1	Input	When set to 1, it indicates header and data on Channel 0 are valid. The header must be interpreted as a memory response, decode resp_type field.
pck_cp2af_sRx.c0.mmioRdValid	1	Input	When set to 1, it indicates a MMIO Read request Channel 0.
pck_cp2af_sRx.c0.mmioWrValid	1	Input	When set to 1, it indicates a MMIO Write request on Channel 0.

Table 20. Rx Channel Signal Description for Channel 1

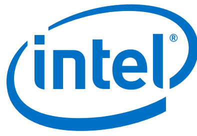
Signal	Width (bits)	Direction	Description
pck_cp2af_sRx.c1.hdr	28	Input	Channel 1 response header. Refer to Table 21 on page 27
pck_cp2af_sRx.c1.rspValid	1	Input	When set to 1, it indicates header on channel 1 is a valid response.

1.3.8.1. Rx Header and Rx Data Format

Table 21. Rx Header Field Definitions

Field	Description
mdata	Metadata: User defined request ID, returned unmodified from memory request to response header. For multi-CL memory response, the same mdata is returned for each CL.
vc_used	Virtual channel used: when using VA, this field identifies the virtual channel selected for the request by FIU. For other VCs it returns the request VC.
format	When using multi-CL memory write requests, FIU may return a single response for the entire payload or a response per CL in the payload. • 1’b0: Unpacked write response – Returns a response per CL. Look up the cl_num field to identify the cache line. • 1’b1: Packed write response – Returns a single response for the entire payload. The cl_num field gives the payload size that is: 1 CL, 2 CLs, or 4 CLs. *Note:* Write responses from the Memory Properties Factory (MPF) Intel FPGA Basic Building Blocks is always packed when sent to the AFU
cl_num	**Format=0:** For a response with >1 CL data payload, this field identifies the cl_num. 2’h0 – First CL. Lowest Address 2’h1 – Second CL 2'h2 – Third CL 2’h3 – Fourth CL. Highest Address *Note:* Responses may be returned out of order.
	Format=1: This field identifies the data payload size. 2’h0 – 1 CL or 64 bytes 2’h1 – 2 CL or 128 bytes 2’h3 – 4 CL or 256 bytes
hit_miss	Cache Hit/Miss status. AFU can use this to generate fine grained hit/miss statistics for various modules. 1’b0 – Cache Miss 1’b1 – Cache Hit

continued...



Field	Description
MMIO Length	Length for MMIO requests: 2’h0 – 4 bytes 2’h1 – 8 bytes 2'h2 - 64 bytes (for MMIO Writes only)
MMIO Address	Double word (DWORD) aligned MMIO address offset, that is, byte address>>2.
UMsg ID	Identifies the CL corresponding to the UMsg
UMsg Type	Two type of UMsg are supported: 1’b1 – UMsgH (Hint) without data 1’b0 – UMsg with Data

Table 22. AFU Rx Response Encodings and Channels Mapping

Request Type	Encoding	Data Payload	Hdr Format
t_if_ccip_c0_Rx: enum t_ccip_c0_rsp			
eRSP_RDLINE	4’h0	Yes	Memory Response Header. Refer to Table 23 on page 28. Qualified with c0.rspValid
MMIO Read	NA	No	MMIO Request Header. Refer to Table 24 on page 29
MMIO Write	NA	Yes	NA
eRSP_UMSG	4’h4	Yes/No	UMsg Response Header. Refer to Table 26 on page 29. Qualified with c0.rspValid
t_if_ccip_c1_Rx: enum t_ccip_c1_rsp			
eRSP_WRLINE	4’h0	No	Memory Response Header. Refer to Table 25 on page 29. Qualified with c1.rspValid
eRSP_WRFENCE	4'h4	No	Wr Fence Response Header. Refer to Table 27 on page 29.
eRSP_INTR	4'h6	No	Interrupt Response Header. Refer to Table 28 on page 29

Table 23. C0 Memory Read Response Header Format Structure: t_ccip_c0_RspMemHdr

Bit	Number of Bits	Field
[27:26]	2	vc_used
[25]	1	RSVD
[24]	1	hit_miss
[23:22]	2	RSVD
[21:20]	2	cl_num
[19:16]	4	resp_type
[15:0]	16	mdata

Send Feedback

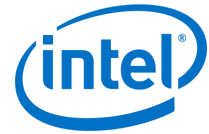


Table 24. MMIO Request Header Format

Bit	Number of Bits	Field
[27:12]	16	`address`
[11:10]	2	`length`
[9]	1	RSVD
[8:0]	9	`TID`

Table 25. C1 Memory Write Response Header Format Structure: `t_ccip_c1_RspMemHdr`

Bit	Number of Bits	Field
[27:26]	2	`vc_used`
[25]	1	RSVD
[24]	1	`hit_miss`
[23]	1	`format`
[22]	1	RSVD
[21:20]	2	`cl_num`
[19:16]	4	`resp_type`
[15:0]	16	`mdata`

Table 26. UMsg Header Format (Integrated FPGA Platform only)

Bit	Number of Bits	Field
[27:20]	8	RSVD
[19:16]	4	`resp_type`
[15]	1	UMsg Type
[14:3]	12	RSVD
[2:0]	3	UMsg ID

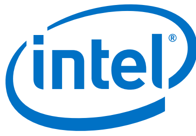
Table 27. WrFence Header Format Structure: `t_ccip_c1_RspFenceHdr`

Bit	Number of Bits	Field
[27:20]	8	RSVD
[19:16]	4	`resp_type`
[15:0]	16	`mdata`

Table 28. Interrupt Header Format Structure: `t_ccip_c1_RspIntrHdr` (Intel FPGA PAC only)

Bit	Number of Bits	Field
[27:26]	2	`vc_used`
[25:20]	6	RSVD

continued...



Bit	Number of Bits	Field
[19:16]	4	resp_type
[15:2]	14	RSVD
[1:0]	2	id

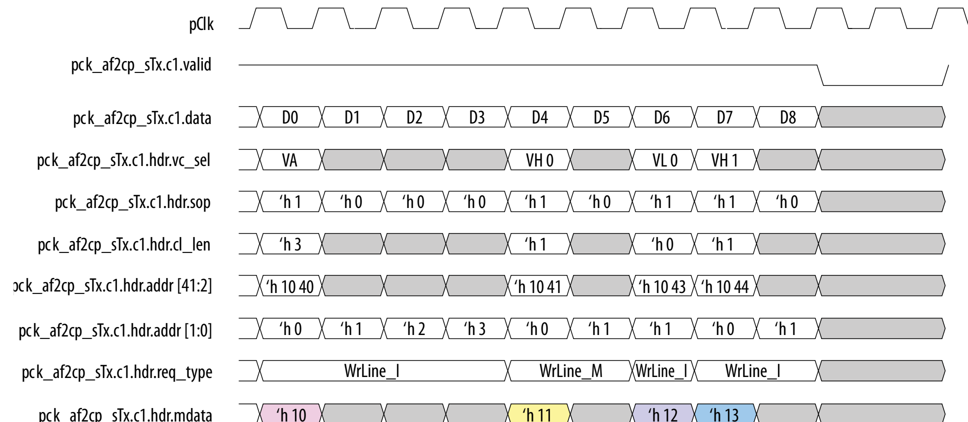
1.3.9. Multi-Cache Line Memory Requests

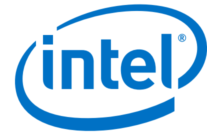
To achieve highest link efficiency, pack the memory requests into large transfer sizes. Use the multi-CL requests for this. Listed below are the characteristics of multi-CL memory requests:

- Highest memory bandwidth is achieved when using a data payload of 4 CLs.
- Memory write request should always begin with the lowest address first. SOP=1 in the c1_ReqMemHdr marks the first CL. All subsequent headers in the multi-CL request must drive incremental values in Address[1:0]and Address[41:2] is treated as don't care.
- An N CL memory write request takes N cycles on Channel 1. It is legal to have idle cycles in the middle of a multi-CL request, but one request cannot be interleaved with another request. It is illegal to start a new request without completing the entire data payload for a multi-CL write request.
- FIU guarantees to complete the multi-CL VA requests on a single VC.
- The memory request address must be naturally aligned. A 2CL request should start on a 2-CL boundary and its CL address must be divisible by 2, that is address[0] = 1'b0. A 4CL request should be aligned on a 4-CL boundary and its CL address must be divisible by 4, that is address[1:0] = 2'b00.
- A multi-CL burst must complete by transmitting all words before issuing any other request. This means that the following special memory write requests cannot be interleaved within a single multi-CL burst:
  - — Write Fences
  - — Interrupts
  - — Byte enable writes

The figure below is an example of a multi-CL Memory Write Request.

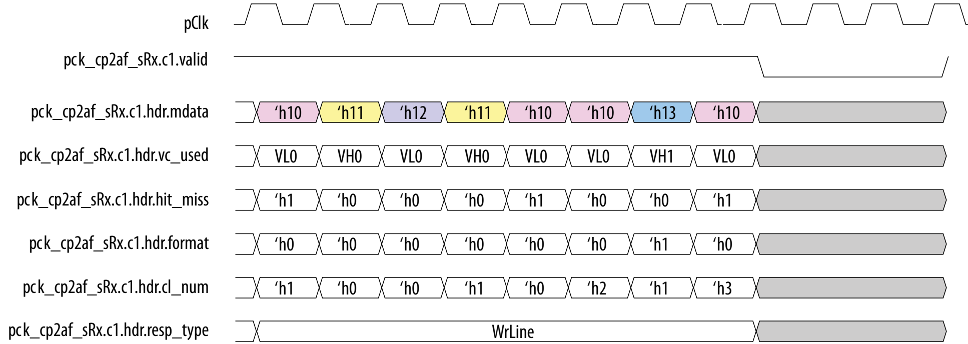
Figure 12. Multi-CL Memory Request





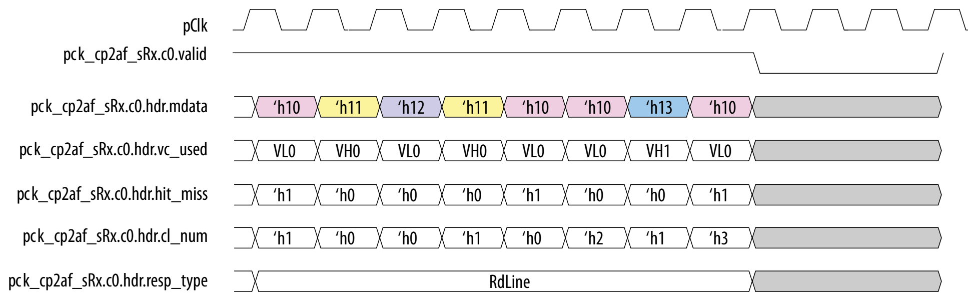
The figure below is an example for a Memory Write Response Cycles. For unpacked response, the individual CLs could return out of order.

Figure 13. Multi-CL Memory Write Responses



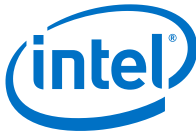
Below is an example of a Memory Read Response Cycle. The read response can be reordered within itself; that is, there is no guaranteed ordering between individual CLs of a multi-CL Read. All CLs within a multi-CL response have the same mdata and same vc_used. Individual CLs of a multi-CL Read are identified using the cl_num field.

Figure 14. Multi-CL Memory Read Responses



1.3.10. Byte Enable Memory Request (Intel FPGA PAC D5005)

To achieve fine control of write data so that only specific bytes of data are written to host memory, use the byte enable mode. Listed below are the characteristics of byte enable memory requests:



- Byte enable memory requests use a byte-invariant little endianness scheme. This means that:
  - For any single or multi-byte element in a data structure, the element uses the same contiguous bytes of memory, specified by the `byte_start` and `byte_len` header fields.
  - Write data is positioned in the data field at the offset where it is stored within the cache line. The first data bit written is bit `byte_start*8`.
- `byte_start` specifies the byte index, where the least significant byte is 0 (`pck_af2cp_sTx.c1.data[7:0]`), and the most significant byte is 63 (`pck_af2cp_sTx.c1.data[511:504]`)
- `byte_len` specifies the number of bytes to be included in a byte-enabled memory write transaction. The byte length extends the write data towards the most significant byte.
- Byte enable memory requests must operate with cache length (`cl_len`) set to 0 (a 1 CL memory write request).
- The length cannot extend past byte 63 of `pck_af2cp_sTx.c1.data`. The maximum allowable byte length can be represented by the following equations:
  - If `byte_start` is 0:
    - `MAX_BYTE_LEN` = 63
  - If `byte_start` is not 0:
    - `MAX_BYTE_LEN` = 64 – `byte_start`

The following table is an example of how the channel 1 request header (`pck_af2cp_sTx.c1.hdr`) indexes bytes in byte enable mode. In this example, the AFU designer is writing bytes [20:4] (`pck_af2cp_sTx.c1.data[167:32]`) of the data word 0xAAAABBBBCCCCDDDDE to bytes [20:4] of the host memory at address 0xFFF00.

	Bit Index	Width (bits)	Field	Value
Header	79:74	6	`byte_len`	0x11
	73:72	2	`vc_sel`	`eVC_VA` (0x0)
	71	1	`sop=1`	1
	70	1	(CL/byte) 0:CL, 1:byte	1
	69:68	2	`cl_len`	0
	67:64	4	`req_type`	`eREQ_WRLINE_I` (0x0)
	63:58	6	`byte_start`	0x4
	57:18	40	`address`	0xFFF00
	17:16	2	—	0x0
	15:0	16	`mdata`	0x0
Data	—	512	Data	0xAAAABBBBCCCCDDDDE

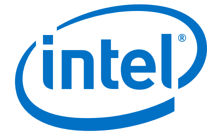
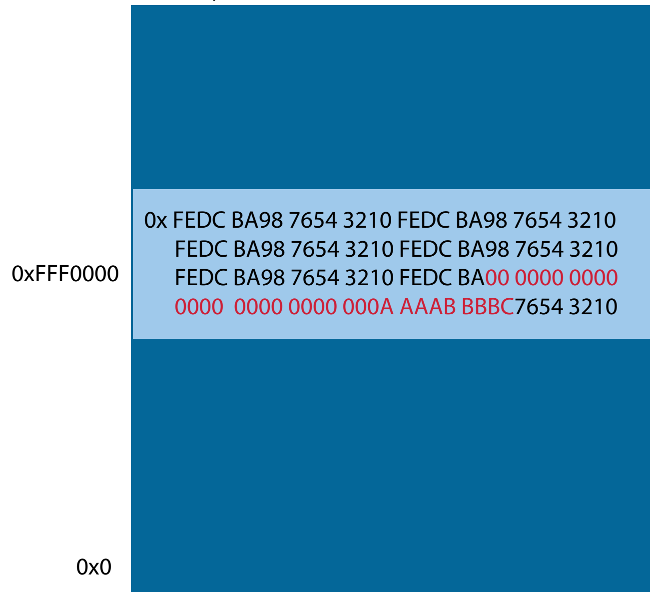


Figure 15. Host memory at address 0xFFF00.

This figure shows the value in host memory before and after a write.

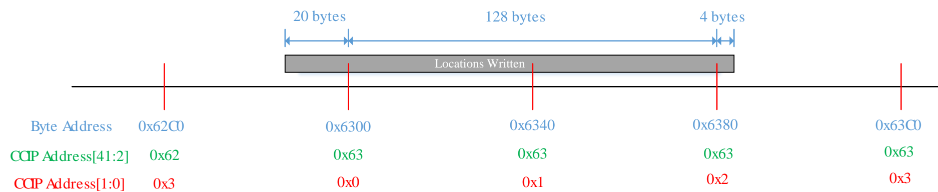


1.3.10.1. Mixing Byte Enable and Full Cache Line Accesses

In some applications, it is necessary for an AFU to access buffers that either start unaligned to 64-byte boundaries or end before the next 64-byte boundary in host memory. An AFU can use a mix of byte enable transactions and full cache line accesses to perform buffer writes that start or end on any boundary. For such a transfer the AFU must not mix byte enable bursts (`mode=eMOD_BYTE`) with full cache line bursts (`mode=eMOD_CL`).

In the following example, the AFU writes 152 bytes of an incrementing pattern starting at byte address 0x62EC. Since the first and last byte address being accessed do not line up to 64-byte boundaries, the transfer is broken up into three sections with the start and end sections utilizing byte enables to update a subset of bytes within a 64-byte aligned region of memory. The first section writes 20 bytes to memory, then 128 bytes are written using a full cache line burst, followed by a final write of 4 bytes.

Figure 16. Memory Accessed and Corresponding CCIP Address

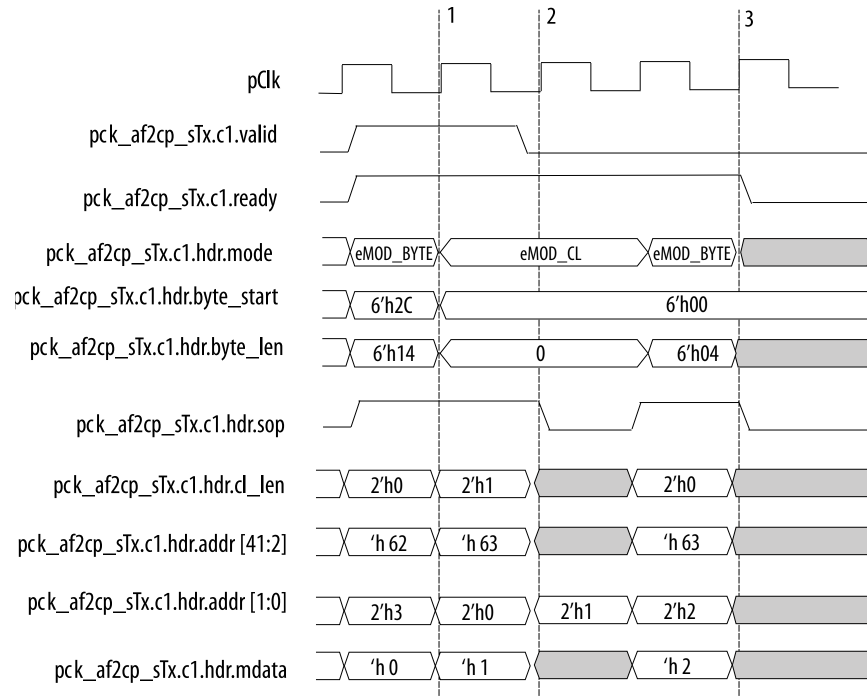


Since the first access does not start on a 64-byte boundary, the mode is set to `eMOD_BYTE`. The `byte_start` field is 0x2C, the `byte_len` field is 0x14, and the CCIP address bits 41:2 set to 0x62 and CCIP address bits 1:0 set to 0x3.

Since the second access is aligned to a 2CL boundary the next 128 bytes can be posted as a two beat burst with mode set to `eMOD_CL`. This access cannot be combined with beats that set mode to `eMOD_BYTE` because the two modes cannot be interleaved in the same burst.

The third access starts on a 64-byte boundary but only accesses four bytes of memory so the mode is set to `eMOD_BYTE`. The `byte_start` field is 0x0, the `byte_len` field is 0x4, and the CCIP address bits 41:2 set to 0x63 and bits 1:0 set to 0x2.

Figure 17. Mixed Byte Enable and Full Cache Line Access Timing Diagram



Notes:

1. 20-byte access posted at starting byte address 0x62EC using byte enables (*mode = eMOD_BYTE*)
2. 128-byte access posted at starting byte address 0x6300 as a 2CL burst (*mode = eMOD_CL*)
3. 4-byte access posted at starting byte address 0x6380 using byte enables (*mode = eMOD_BYTE*)

Send Feedback

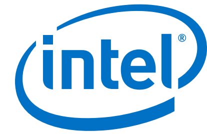
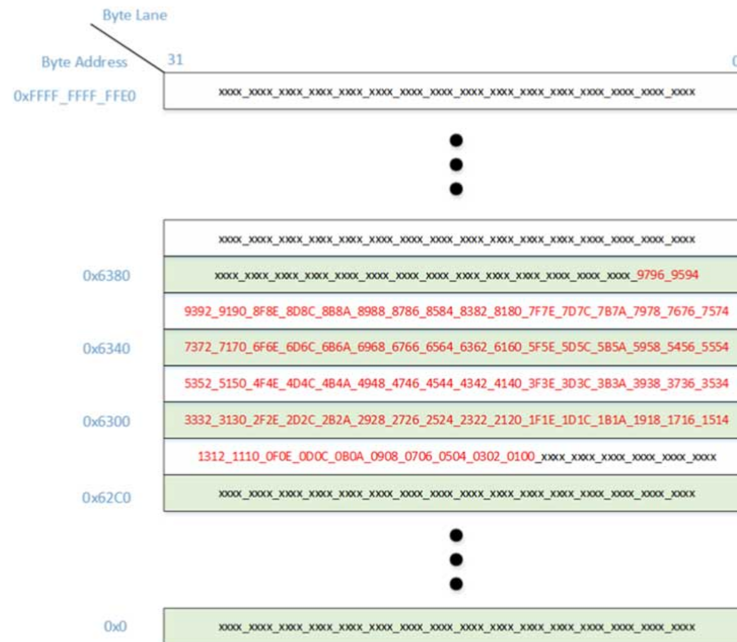


Figure 18. Host Memory



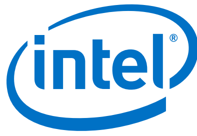
1.3.11. Additional Control Signals

Unless otherwise mentioned, all signals are active high.

Table 29. Clock and Reset

Signal	Width (bits)	Direction	Description
pck_cp2af_softReset	1	Input	Synchronous ACTIVE HIGH soft reset. When set to 1, AFU must reset all logic. The minimum reset pulse width is 256 pClk cycles. All outstanding CCI-P requests are flushed before de-asserting soft reset. A soft reset does not reset the FIU.
pClk	1	Input	Primary interface clock. All CCI-P interface signals are synchronous to this clock.
pClkDiv2	1	Input	Synchronous and in phase with pClk. 0.5x, the pClk clock frequency.
pClkDiv4	1	Input	Synchronous and in phase with pClk. 0.25x, the pClk clock frequency.
uClk_usr	1	Input	The user-defined clock is not synchronous with the pClk. AFU must synchronize the signals to pClk domain before driving the CCI-P interface. The AFU load utility programs the user-defined clock frequency before de-asserting pck_cp2af_softReset.
uClk_usrDiv2	1	Input	Synchronous with uClk_usr and 0.5x the frequency.

continued...



Signal	Width (bits)	Direction	Description
			Note: You can set the frequency to a value that is not synchronous with the `uClk_usr`.
`pck_cp2af_pwrState`	2	Input	Indicates the current AFU power state request. In response to this, the AFU must attempt to reduce its power consumption. If sufficient power reduction is not achieved, the AFU may be Reset. 2’h0 – AP0 - Normal operation mode 2’h1 – AP1 - Request for 50% power reduction 2’h2 – Reserved 2’h3 – AP2 - Request for 90% power reduction When `pck_cp2af_pwrState` is set to AP1, the FIU starts throttling the memory request path to achieve 50% throughput reduction. The AFU is also expected to reduce its power utilization to 50%, by throttling back accesses to FPGA internal memory resources and its compute engines. Similarly upon transition to AP2, the FIU throttles the memory request paths to achieve 90% throughput reduction over normal state, and AFU in turn is expected to reduce its power utilization to 90%.
`pck_cp2af_error`	1	Input	CCI-P protocol error has been detected and logged in the `PORT Error` register. This register is visible to the AFU. It can be used as a signal tap trigger condition. When such an error is detected, the CCI-P interface stops accepting new requests and sets `AlmFull` to 1. In the event of a CCI-P protocol error, you should not expect any outstanding transactions to complete even though the AFU is still active (not held in reset).

Related Information

Accelerator Functional Unit (AFU) Developer’s Guide for Intel FPGA Programmable Acceleration Card

1.3.12. Protocol Flow

1.3.12.1. Upstream Requests

Table 30. Protocol Flow for Upstream Request from AFU to FIU

The Tx Data column identifies whether the request expects a Tx Data payload. The Rx Data column identifies whether the response returns an Rx Data payload.

Type	Tx Request	Tx Data	Rx Response	Rx Data
Memory Write	WrLine_I	Yes	WrLine	No
	WrLine_M			
	WrPush_I			
Memory Read	RdLine_I	No	RdLine	Yes
	RdLine_S			
Special Messages	WrFence	No	WrFence	No
	Interrupt	No	Interrupt	No

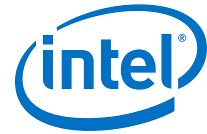


Table 31. Protocol Flow for Upstream Request from AFU to FIU

- WrLine_I: Requires special handling, because it must first write to the CL and then evict it from the cache. The eviction forms Phase 2 of the request.
- RdLine_I: Recommended as the default read type.
- RdLine_S: Use sparingly only for cases where you have identified highly referenced CLs.
- RdCode: Updates the CPU directory and lets the FPGA cache the line in Shared state. RdCur does NOT update the CPU directory, FPGA does not cache this line. A future access to this line from CPU, does not snoop the FPGA.

CCI-P Request	FPGA Cache		UPI Cycle	Next State	CCI-P Response	UPI Cycle	Next State	CCI-P Response	UPI Cycle	Next State
	Hit/Miss	State	Phase 1			Phase 2			Phase 3	
WrLine_I	Hit	M	None	M	WrLine	WbMtoI	I			
	Hit	S	InvItoE							
	Miss	I								
WrLine_M	Hit	M	None	M	WrLine	NA				
	Hit	S	InvtoE							
	Miss	I								
WrLine_I	Miss	M	WbMotI	I		InvItoE	M	WrLine	WbMotI	I
WrLine_M										
WrPush_I									WbPushMotI	I
WrLine_I	Miss	S	EvctCln	I		InvItoE	M	WrLine	WbMotI	I
WrLine_M										
WrPush_I									WbPushMotI	I
WrPush_I	Hit	M	None	M	WrLine	WbPush MotI	I			
		S,I	InvItoE							
RdLine_S	Hit	S,M	None	No Change	RdLine	N.A				
	Miss	I	RdCode	S	RdLine					
RdLine_I	Hit	S,M	None	No Change	RdLine	NA				
	Miss	I	RdCur	I	RdLine					
RdLine_I	Miss	M	WbMtoI	I		RdCur	I	RdLine		
RdLine_S						RdCode	S			
RdLine_I		S	EvctCln			RdCur	I			
RdLine_S						RdCode	S			

1.3.12.2. Downstream Requests

Table 32. Protocol Flow for Downstream Requests from CPU to AFU

Rx Request	Rx Data	Tx Response	Tx Data
MMIO Read	No	MMIO Read Data	Yes
MMIO Write	Yes	None	NA
UMsg	Yes	None	NA
UMsgH	No	None	NA

1.3.13. Ordering Rules

1.3.13.1. Memory Requests

The CCI-P memory consistency model is different from the PCIe consistency model. The CCI-P implements a "relaxed" memory consistency model.

It relaxes ordering requirements for requests to:

- Same address
- Different addresses

Table 33 on page 38 defines the ordering relationship between two memory requests on CCI-P. The same rules apply for requests to the same address or different addresses. The table entries are defined as follows:

- Yes: Requests from first column may pass request from first row.
- No: Requests from first column cannot pass request from first row.

Table 33. Ordering Rules for Upstream Requests from AFU

Row Bypass Column?	Read	Write	WrFence	Interrupt
Read	Yes	Yes	Yes	Yes
Write	Yes	Yes	No	Yes
WrFence	Yes	No	No	No
Interrupt	Yes	Yes	No	Yes

You can interpret the table:

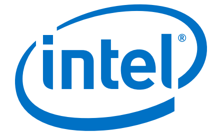
- All operations, except reads, are ordered with regards to WrFences.
- All other operations are unordered.

Intra-VC Write Observability

Upon receiving a memory write response, the write has reached a local observability point.

Note: VA is not a physical channel and there are no such guarantees for requests to VA.

- All future reads from AFU to the same physical channel receive the new data.
- All future writes on the same physical channel replace the data.



Inter-VC Write Observability

A memory write response does NOT mean the data are globally observable across channels. A subsequent read on a different channel may return old data and a subsequent write on a different channel may retire ahead of the original write. WrFence to VA invokes a protocol that is guaranteed to synchronize across VCs. A WrFence VA performs a broadcast operation across all channels.

- All writes preceding a write fence are pushed to a global observability point.
- Upon receiving a WrFence response, all future reads from an AFU receive the latest copy of data written, previously, to the write fence being issued.

1.3.13.1.1. Memory Write Fence

CCI-P WrFence Request

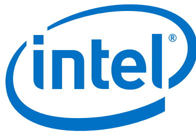
CCI-P defines a `WrFence` request type, this can be used for all VCs, including VA. The FIU implementation of `WrFence` stalls the C1 channel and hence block all write streams sharing the CCI-P write path. Furthermore, a `WrFence` request guarantees global observability, which means for PCIe paths, FIU generates a Zero Length Read (ZLR) to push out the writes. Given this, `WrFence` requests could incur long stalls on the C1 Channel. To avoid this from happening, restrict its use to synchronization points in your AFU's data flow.

- WrFence guarantees that all interrupts or writes preceding the fence are committed to memory before any writes following the Write Fence are processed.
- A WrFence is not re-ordered with other memory writes, interrupts, or WrFence requests.
- WrFence provides no ordering assurances with respect to Read requests.
- A WrFence does NOT block reads that follow it. In other words, memory reads can bypass a WrFence. This rule is described in the "Memory Requests" section.
- WrFence request has a vc_sel field. This allows determination of which virtual channels the WrFence is applied to. For example, if moving the data block using VL0, only serialize with respect to other write requests on VL0. That is, you must use WrFence with VL0. Similarly, if using memory writes with VA, then use WrFence with VA.
- A WrFence request returns a response. The response is delivered to the AFU over RX C1 and identified by the resp_type field. Since reads can bypass a WrFence, to ensure the latest data is read in a write followed by read (RaW hazard), issue a WrFence and then wait for the WrFence response before issuing the read to the same location.

Write Response Counting

AFU implements the memory write barrier, it can do this by waiting for all outstanding writes to complete before sending the next write after the barrier. The logic to track the outstanding writes can be a simple counter that increments on request and decrements on response, hence the name "write response counting". Write responses only guarantee local observability. This technique only works for implementing a memory barrier on a write stream targeted to a single VC (for example: VL0, VH0, VH1). This technique should not be used if a write stream uses VA or a mix of VCs.

Note: In cases such as these, you should implement a write fence instead.



One of the key advantages of this technique is that AFU can implement fine grained barriers. For example, if AFU has two independent write streams, it can implement a write response tracker per stream. If write stream 1 needs a memory barrier, it would only stall the writes from stream 1 while continuing to send writes from stream 2. The `Mdata` field can be used to encode the stream id. Such a fine grained memory barrier may:

- Minimize the latency cost of the barrier because it would only wait on specific outstanding writes to complete, instead of all of them.
- Improve link utilization because unrelated write streams can continue to make forward progress.

Related Information

Memory Requests on page 38

1.3.13.1.2. Memory Consistency Explained

CCI-P can re-order requests to the same and different addresses. It does not implement logic to identify data hazards for requests to same address.

Two Writes to the Same VC

Memory may see two writes to the same VC in a different order from their execution, unless the second write request was generated after the first write response was received. This is commonly known as a write after write (WaW) hazard.

The table below shows two writes to the same VC when the second write is executed after the first write is received.

Table 34. Two Writes to Same VC, Only One Outstanding

AFU	Processor

`VH1: Write1 Addr=X, Data=A`
`Resp 1`
`VH1: Write2 Addr=X, Data=B`
`Resp 2`
	—

—	`Read1 Addr=X, Data = A` `Read2 Addr=X, Data = B`

AFU writes to address X twice on same VC, but it only sends the second write after the first write is received. This ensures that the first write was sent out on the link, before the next one goes out. The CCI-P guarantees that these writes are seen by the Processor in the order that they were issued. The processor sees Data A, followed by Data B when reading from address X multiple times.

Use a WrFence instead to enforce ordering between writes to same VC. Note that WrFence has stronger semantics, it stalls processing all writes after the fence until all previous writes have completed.

Two Writes to Different VCs

The table below shows two writes to different VCs may be committed to memory in a different order than they were issued.

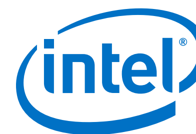


Table 35. Write Out of Order Commit

AFU	Processor
VH1: Write1 X, Data=A VL0: Write2 X, Data=B	—
—	Read1 X, Data = B Read2 X, Data = A

AFU writes to X twice, Data=A over VH1 and Data=B over VL0. The processor polls on address X and may see updates to X in reverse order; that is, the CPU may see Data=B, followed by Data=A. In summary, the write order seen by the processor may be different from the order in which AFU completed the writes. Writes to separate channels have no ordering rules and as a result you should broadcast a write fence to VA to synchronize across them.

The table below shows the use of WrFence to enforce write ordering.

Table 36. Use WrFence to Enforce Write Ordering

AFU	Processor
VH1: Write1 Addr=X, Data=A VA: WrFence VL0: Write2 Addr=X, Data=B	—
—	Read1 Addr=X, Data = A Read2 Addr=X, Data = B

This time the AFU adds a VA WrFence between the two writes. The WrFence ensures that the writes become visible to the processor before the WrFence followed by the writes after the WrFence. Hence, the processor sees Data=A and then Data=B. The WrFence was issued to VA, because the writes to be serialized were sent on different VCs.

Two Reads from Different VCs

Issuing reads to different VCs may complete out of order; the last read response may return old data.

The table below shows how reads from the same address over different VCs may result in re-ordering.

Table 37. Read Re-Ordering to Same Address, Different VCs

Processor	AFU	
Store addr=X, Data=A Store addr=X, Data=B	Request	Response
	VH1: Read1 Addr=X	—
	VL0: Read2 Addr=X	—
	—	VL0: Resp2 Addr=X, Data=B
	—	VH1: Resp1 Addr=X, Data=A

Processor writes X=1 and then X=2. The AFU reads address X twice over different VCs. Read1 was sent on VH1 and Read2 on VL0. The FIU may re-order the responses and return data out of order. AFU may see X=2, followed by X=1. This is different from the processor write order.

Two Reads from the Same VC

Reads to the same VC may complete out of order; the last read response always returns the most recent data. The last read response may correspond to an older read request as shown in the following table.

Note: VA reads behave like two reads from different VCs.

The following table shows how reads from the same address over the same VC may result in re-ordering. However, the AFU sees updates in the same order in which they were written.

Table 38. Read Re-Ordering to Same Address, Same VC

Processor	AFU	
Store Addr=X, Data=A Store Addr=X, Data=B	Request	Response
	VL0: Read1 Addr=X	—
	VL0: Read2 Addr=X	—
	—	VL0: Resp2 Addr=X, Data=A
	—	VL0: Resp1 Addr=X, Data=B

Processor writes X=1 and then X=2. The AFU reads address X twice over the same VC. Both Read1 and Read2 are sent to VL0. The FIU may still re-order the read responses, but the CCI-P standard guarantees to return the newest data last; that is, the AFU sees updates to address X in the order in which processor writes to it.

When using VA, FIU may return data out of order, because VA requests may be directed to VL0, VH0 or VH1.

Read-After-Write from Same VC

CCI-P standard does not order read and write requests to even the same address. The AFU must explicitly resolve such dependencies.

Read-After-Write from Different VCs

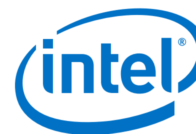
The AFU cannot resolve a read-after-write dependency when different VCs are used.

Write-after-Read to Same or Different VCs

CCI-P does not order write after read requests even when they are to the same address. The AFU must explicitly resolve such dependencies. The AFU must send the write request only after read response is received.

Transaction Ordering Example Scenarios

Transactions to the Same Address—More than one outstanding read/write requests to an address results in non-deterministic behavior.



- **Example 1:** Two writes to same address X can be completed out of order. The final value at address X is non-deterministic. To enforce ordering add a WrFence between the write requests. Or, wait for the response from the first write to return before issuing the second write if the same virtual channel is accessed.
- **Example 2:** Two reads from same address X, may be completed out of order. This is not a data hazard, but an AFU developer should make no ordering assumptions. The second read response received contains the latest data stored at address X assuming both reads are issued to the same virtual channel.
- **Example 3:** Write to address X, followed by read from address X. It is non-deterministic; that is, the read returns the new data (data after the write) or the old data (data before the write) at address X. To ensure the latest data is read wait for the write response to return before issuing the read to address X using the same virtual channel.
- **Example 4:** Read followed by write to address X. It is non-deterministic; that is, the read returns the new data (data after the write) or the old data (data before the write) at address X.

  Use the read responses to resolve read dependencies.

Transactions to Different Addresses—Read/write requests to different addresses may be completed out of order.

- **Example 1:** AFU writes the data to address Z and then wants to notify the SW thread by updating a value of flag at address X.

  To implement this, the AFU must use a write fence between write to Z and write to X. The write fence ensures that Z is globally visible before write to X is processed.
- **Example 2:** AFU reads data starting from address Z and then wants to notify a software thread by updating the value of flag at address X.

  To implement this, the AFU must perform the read from Z, wait for all the read responses and then perform the write to X.

1.3.13.2. MMIO Requests

The FIU maps the AFU's MMIO address space to a 64-bit prefetchable PCIe BAR. The AFU's MMIO mapped registers does not have read side-effects; and writes to these registers are able to tolerate write-merging.

For more information about prefetchable BAR, refer to the *PCIe Specification*.

MMIO requests targeted to the AFU, are sent to the AFU in the same order they were received from the PCIe link. Similarly, MMIO read responses are returned to the PCIe link in the same order that the AFU sends it to the CCI-P interface. In other words, the FIU does not re-order MMIO requests or responses targeted to the AFU.

The IA processor can map the PCIe BAR as either a UC or WC memory type. Table 39 on page 44 summarizes the IA's ordering rules for UC and WC typed BAR.

For more information about uncacheable (UC) and write combining (WC) ordering rules, refer to the *Intel Software Developers Manual*.

Table 39. MMIO Ordering Rules

Request	Memory Attribute	Payload Size	Memory Ordering	Comments
MMIO Write	UC	4 bytes, 8 bytes, or 64 bytes	Strongly ordered	Common case-software behavior
	WC	4 bytes, 8 bytes, or 64 bytes (requires Intel Advanced Vector Extensions 512 (Intel AVX-512))	Weakly ordered	Special case
MMIO Read	UC	4 bytes or 8 bytes	Strongly ordered	Common case-software behavior
	WC	4 bytes or 8 bytes	Weakly ordered	Special case-streaming read (MOVNTDQA) can cause wider reads. NOT supported

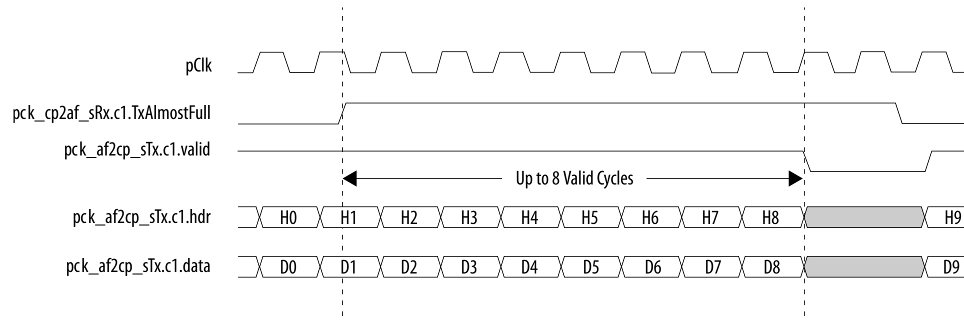
Related Information

Intel Software Developers Manual

1.3.14. Timing Diagram

This section provides the timing diagrams for CCI-P interface signals.

Figure 19. Tx Channel 0 and 1 Almost Full Threshold



Note: The TX channel 0 and 1, almost full threshold signals, assert when there is room for only eight more transactions to be accepted. TX channels 0 and 1 must deassert the valid signals up to eight cycles after almost full asserts.

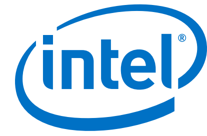
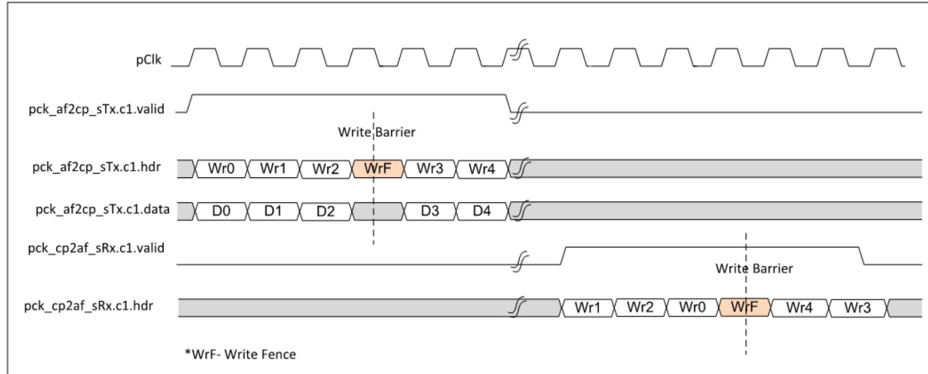


Figure 20. Write Fence Behavior



The WrFence is inserted between WrLine requests. A WrFence response returns from Rx channel 1.

Note: In Figure 20 on page 45, all of the writes generated before the WrFence are responded to (completed) before any of the writes that arrive after the WrFence are completed.

The WrFence only fences previous writes issued to the VC selected. Chose VA to fence writes across all VCs.

Figure 21. C0 Rx Channel Interleaved between MMIO Requests and Memory Responses

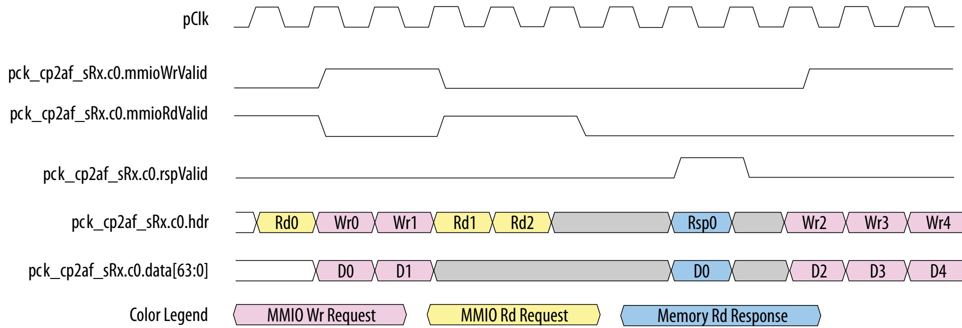
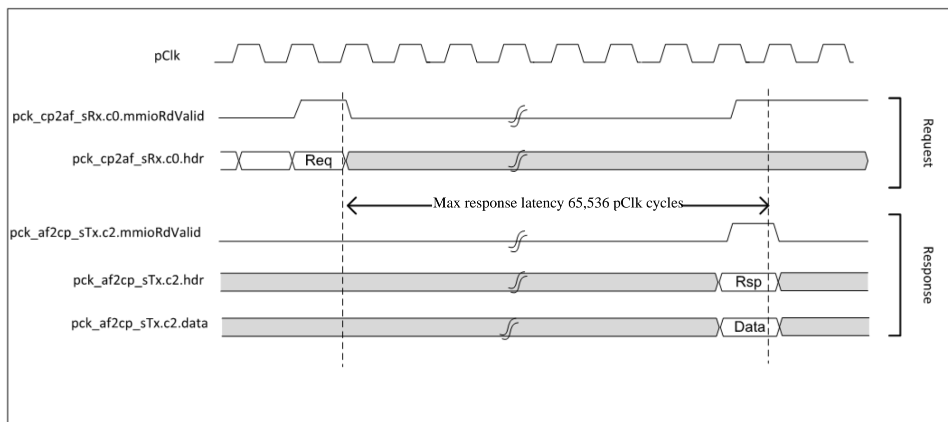
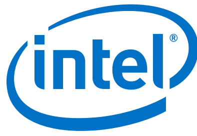


Figure 22. MMIO Read Response Timeout





Note: The AFU responds to MMIO read transactions in a Max response time of 65,536 pClks cycles.

1.3.15. CCI-P Guidance

This section suggests techniques and settings that are useful when beginning to use the Integrated FPGA Platform or Intel FPGA PAC with Intel FPGA IP system.

The CCI-P interface provides several advanced features for fine grained control of FPGA caching states and virtual channels. When used correctly, optimal performance through the interface can be obtained; if used incorrectly, you may see significant degradation in performance.

The table below lists some suggested parameters for request fields.

Table 40. Recommended Choices for Memory Requests

Field	Recommended Option	
`vc_sel`	For producer-consumer type flows	VA
	For latency sensitive flows	VL0
	For data dependent flow	Use any one of the VCs, except VA; or use MPF's VC map
`cl_len`	For maximum bandwidth	4 CLs (256 bytes)
`req_type`	Memory reads	`RdLine_I`
	Memory writes	`WrLine_M`

Use the following guidance, when setting the size of the request buffers in the AFU:

- Intel FPGA PAC
  - 64 outstanding requests on VH0
  - VA and VH0 can share the same 64 outstanding request buffers
- Integrated FPGA Platform
  - VH0 and VH1 can each have 64 outstanding requests.
  - VL0 requires at least 128 transactions in flight to reach full bandwidth, and no more than 256 outstanding requests are required to cover the long latency tail.
  - For VA, maximum performance can be achieved with a minimum of 256 transactions and a maximum of 384 transactions. Consider sharing VA buffers with other VCs, to save design area.

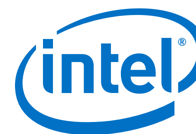
1.4. AFU Requirements

This section defines the AFU initialization flow upon power on and the mandatory AFU control and status registers (CSRs).

For more information about AFU CSRs, refer to the "Device Feature List" section.

Related Information

Device Feature List on page 50



1.4.1. Mandatory AFU CSR Definitions

The following requirements are defined for software access to AFU CSRs.

1. Software is expected to access 64-bit CSRs as aligned quad words (8 bytes). To modify a field (for example, bit or byte) in a 64-bit register, the entire quad word is read, the appropriate field(s) are modified, and the entire quad word is written back (read-modify-write operation).
2. Similarly for an AFU supporting 32-bit CSRs, software is expected to access them as aligned double words (4 bytes).

Each CCI-P-compliant AFU is required to implement the four (not including `DEV_FEATURE_HDR (DFH)` at 0x0000) mandatory registers defined in the table below. If you do not implement these registers or if you implement them incorrectly, AFU discovery could fail or some other unexpected behavior may occur.

Table 41. Register Attribute Definition

Attribute	Expansion	Description
RO	Read Only	The bit is set by hardware only. Software can only read this bit. Writes do not have any effect.
Rsvd	Reserved	Reserved for future definition. AFU must set them to 0s. Software must ignore these fields.

Table 42 on page 47 shows both byte and DWORD offsets for the mandatory AFU CSRs. The base address is set by the platform and need not be specified by the AFU.

Table 42. Mandatory AFU CSRs

Name	DWORD Address Offset (CCI-P)	Byte Address Offset (Software)
DEV_FEATURE_HDR (DFH) For bit descriptions, refer to Table 43 on page 48. *Note:* AFU CSRs are 64 bits.	0x0000	0x0000
AFU_ID_L Lower 64 bits of the AFU_ID GUID.	0x0002	0x0008
AFU_ID_H Upper 64 bits of the AFU_ID GUID.	0x0004	0x0010
DFH_RSVD0	0x0006	0x0018
DFH_RSVD1	0x0008	0x0020

Figure 23 on page 48 shows how the AFU might set the mandatory AFU CSRs. You must define your own AFU ID. Note that the AFU uses DWORD addresses. Figure 24 on page 48 shows how software program might read the AFU ID.

Figure 23. Set the Mandatory AFU Registers in the AFU

```
t_ccip_c0_ReqMmioHdr mmioHdr;
:
:
case (mmioHdr.address)
// AFU header
       16'h0000 : af2cp_sTxPort.c2.data <= {                // DFH
          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 version   = 0
          12'b0     // Feature ID          = 0
       };
     16'h0002 : af2cp_sTxPort.c2.data <= 64'ha12e_bb32_8f7d_d35c;   // AFU_ID_L
                                                                   // (arbitrary example)
     16'h0004 : af2cp_sTxPort.c2.data <= 64'ha455_783a_3e90_43b9;   // AFU_ID_H
                                                                   // (arbitrary example)
     16'h0006 : af2cp_sTxPort.c2.data <= 64'h0;                     // Reserved
     16'h0008 : af2cp_sTxPort.c2.data <= 64'h0;                     // Reserved
```

The software and the AFU RTL must reference the same AFU ID.

Figure 24. Software Reads the AFU ID

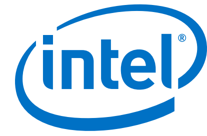
```
   btUnsigned32bitInt AFUID_H, AFUID_L;
:
:
   IALIMMIO      *m_pALIMMIOService;        //< Pointer to MMIO Service:
:
:
// the AFUID to be passed to the Resource Manager. It will be used to locate the appropriate device.
     ConfigRecord.Add(keyRegAFU_ID,"A455783A-3E90-43B9-A12E-BB328F7DD35C");
:
     m_pALIMMIOService->mmioRead32(0x0008, &AFUID_L);
     printf("Read AFUID_L= 0x%08x\n", AFUID_L);

     m_pALIMMIOService->mmioRead32(0x0010, &AFUID_H);
     printf("Read AFUID_H= 0x%08x\n", AFUID_H);
```

Table 43. Feature Header CSR Definition

Address Offset = 0x0

Bit	Attribute	Default	Description
63:60	RO	0x1	Type: AFU
59:52	Rsvd	0x0	Reserved
51:48	RO	0x0	AFU Minor version number User defined value
47:41	Rsvd	0x0	Reserved
40	RO	N/A	End of List 1'b0: There is another feature header beyond this (see "Next DFH Byte Offset") 1'b1: This is the last feature header for this AFU
39:16	RO	0x0	Byte offset to the Next Device Feature Header; that is, offset from the current address. For an example of DFH byte offset, refer to Table 45 on page 53.
15:12	RO	0x0	AFU Major version number

continued...

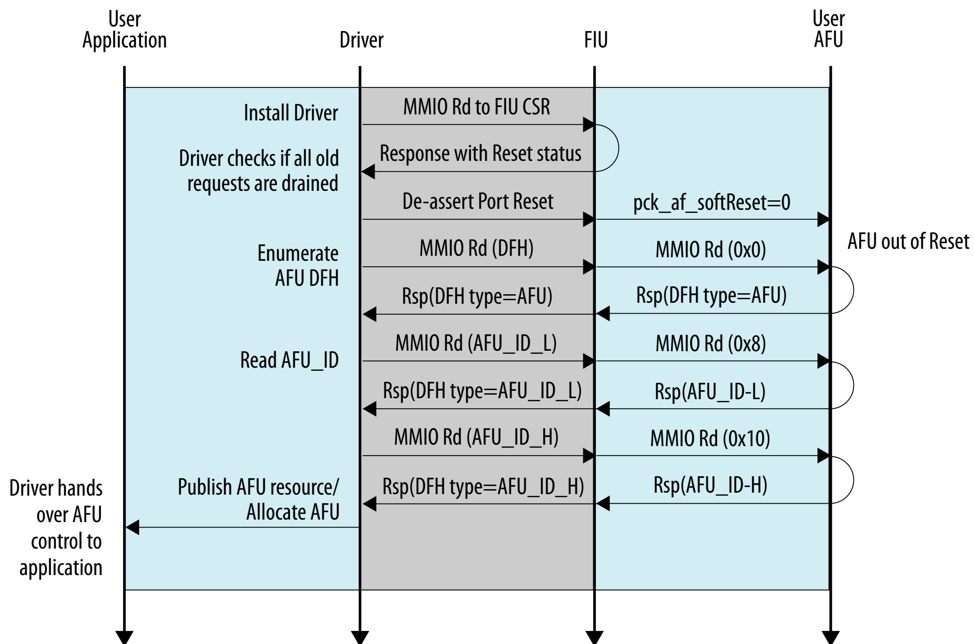


Bit	Attribute	Default	Description
			User defined value
11:0	RO	N/A	CCI-P version number Use the CCIP_VERSION_NUMBER parameter from `ccip_if_pkg.sv`

1.4.2. AFU Discovery Flow

A CCI-P compliant AFU must implement the mandatory AFU CSRs. The following figure shows initial transactions immediately after `pck_cp2af_softReset` is de-asserted. The AFU has to accept the MMIO Read cycles immediately after soft reset is de-asserted.

Figure 25. AFU Discovery Flow

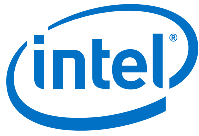


1.4.3. AFU_ID

The purpose of an AFU_ID is to precisely identify the architectural interface of an AFU. This interface is the contract that the AFU makes with the software.

Multiple instantiations of an AFU can have the same AFU_ID value, but if the architectural interface of the AFU changes, then it needs a new AFU_ID.

The architectural interface of an AFU comprises the syntax and semantics of the AFU design, consisting of the AFU's functionality, its CSR definitions, the protocol expected by the AFU when manipulating its CSRs, and all implicit or explicit assumptions or guarantees about its buffers.



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, and can be 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

1.5. Intel FPGA Basic Building Blocks

Intel FPGA Basic Building Blocks (BBBs) are Intel-provided IPs that users can instantiate in their AFU. There are two types of BBBs: software-visible (exposes a register interface and requires software interaction) and software-invisible (does not require software interaction). In both cases, if you are the AFU developer, it is your responsibility to integrate the hardware and software into your AFU.

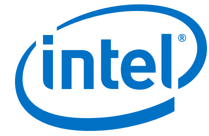
For more information about BBBs, refer to the "Intel FPGA Basic Building Blocks (BBB)" web page.

Related Information

Intel FPGA Basic Building Blocks (BBB)

1.6. Device Feature List

This section defines a Device Feature List (DFL) structure that creates a linked list of features within MMIO space, thus providing an extensible way of adding and enumerating features. A feature region (sometimes referred to as a "feature") is a group of related CSRs. For example, two different features of a DMA engine can be queue management and QoS functions. You can group queue management and QoS functions into two different feature regions. A Device Feature Header (DFH) register marks the start of the feature region.

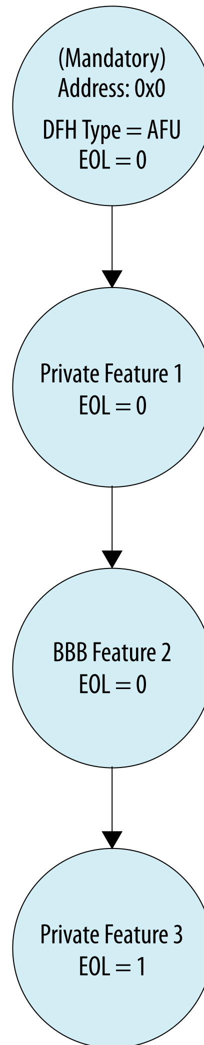


The software can walk through the DFL to enumerate the following:

- AFU
  - An AFU is compliant to the CCI-P interface and connected directly to the CCI-P Port. Must implement mandatory AFU registers, including AFU ID.
  - The AFU DFH must be located at MMIO address 0x0.
- Private features
  - These are a linked list of features within the AFU, which provides a way of organizing functions within an AFU. It is the AFU developer's responsibility to enumerate and manage them.
  - They are not required to implement a ID.
- Intel FPGA Basic Building Blocks
  - Are special features within the AFU, which are meant to be reusable building blocks (design once, reuse many times). Software visible Intel FPGA Basic Building Blocks typically come with a corresponding software service to enumerate and configure the Intel FPGA Basic Building Blocks, and possibly provide a higher-level software interface to the Intel FPGA Basic Building Blocks.
  - Do not have strong hardware interface requirements like an AFU, but must have well defined architectural semantics from a software point of view.
  - Must implement the mandatory DFH registers when visible.
  - Must implement a GUID only for software-visible Intel FPGA Basic Building Blocks.

The following figure shows an example of an AFU's feature hierarchy made up of BBBs and private features.

Figure 26. Example Feature Hierarchy

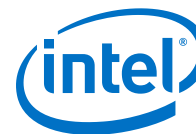


A Device Feature Header (DFH) register (shown in below) marks the start of the feature region.

Table 44. Device Feature Header CSR

Device Feature Header			
Bit	**Description**		
63:60	Feature Type		
	4'h1 – AFU	4'h2 – BBB	4'h3 – Private Features
59:52	Reserved		
51:48	AFU Minor version number User defined value	Reserved	
47:41	Reserved		
40	End of List		

continued...



Device Feature Header		
Bit	**Description**	
	1'b0 There is another feature header beyond this (see "Next DFH Byte Offset") 1'b1 This is the last feature header for this AFU	
39:16	Next DFH Byte offset Next DFH Address = Current DFH Address + Next DFH Byte offset Also used as indication for the maximum size of MMIO region occupied by this feature. For last feature, this offset points to the beginning of the unallocated MMIO region, if any (or beyond the end of the MMIO space). Refer to the example in Table 45 on page 53.	
15:12	AFU Major VersionNumber User defined	Feature Revision Number User defined
11:0	CCI-P Version Number Use the CCIP_VERSION_NUMBER parameter from `ccip_if_pkg.sv`	Feature ID Contains user defined ID to identify features within an AFU

Table 45. Next DFH Byte Offset Example

Feature	DFH Address	EOL	Next DFH Byte Offset
0	0x0	0x0	0x100
1	0x100	0x0	0x180
2-Last Feature	0x280	0x1	0x80
Unallocated MMIO space, no DFH	0x300	N/A	N/A

A DFH with the type set to BBB must be followed by the mandatory BBB registers listed below.

Table 46. Mandatory BBB DFH Register Map

Byte Address offset within DFH	Register Name
0x0000	DFH Type=BBB
0x0008	BBB_ID_L
0x0010	BBB_ID_H

Table 47. BBB_ID_L CSR Definition

Register Name		BB_ID_L
Bit	**Attribute**	**Description**
63:0	RO	Lower 64-bits of the BBB_ID ID

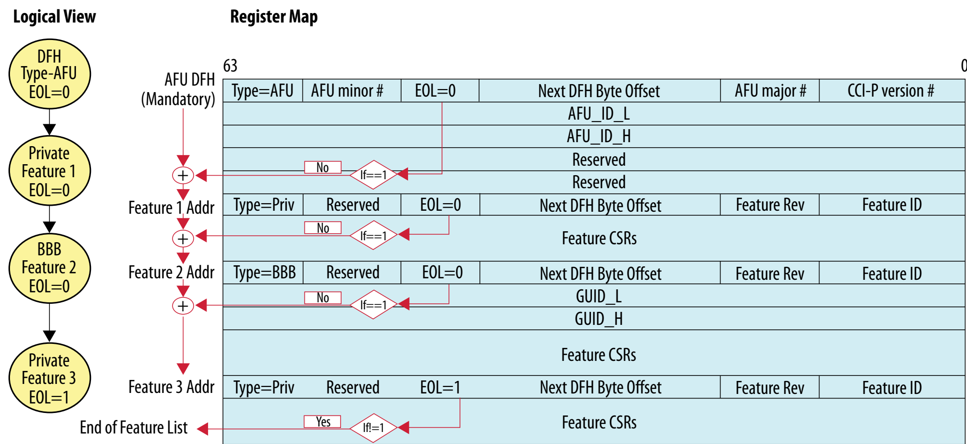
Table 48. BBB_ID_H CSR Definition

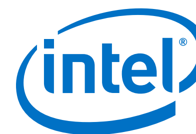
Register Name		BB_ID_H
Bit	**Attribute**	**Description**
63:0	RO	Upper 64-bits of the BBB_ID ID

The BBB_ID is an GUID, similar in concept to an AFU_ID. It is defined so that each BBB has a unique identifier. This allows the software to identify the software service associated with the BBB hardware.

The figure below shows how a logical feature hierarchy (shown on left-hand side) can be expressed using DFH registers defined in this section.

Figure 27. Device Feature Conceptual View





1.7. Document Revision History for Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual

Document Version	Intel Acceleration Stack Version	Changes
2019.11.04	2.0.1 (supported with Intel Quartus® Prime Pro Edition 19.2)and 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 the feature CCI-P Byte Enable.
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)	• *Acronym List for Acceleration Stack for CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*: Added Intel FPGA Programmable Acceleration Card (Intel FPGA PAC) to the RdLine_I acronym. • *Memory and Cache Hierarchy*: Updated the Intel FPGA PAC Memory Hierarchy figure. • *CCI-P Interface*: Updated the CCI-P Signals figure. • *MMIO Requests*: Changed 64-byte to 64-bit in this sentence: "The FIU maps the AFU's MMIO address space to a **64-bit** prefetchable PCIe BAR."
2018.12.04	1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1)	Added the "Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual Archives" section that contains the archived versions of this document.
2018.08.06	1.1 (supported with Intel Quartus Prime Pro Edition 17.1.1) and 1.0 (supported with Intel Quartus Prime Pro Edition 17.0.0)	Removed detailed clock frequencies from Table 6 in the "Comparison of FIU Capabilities" section; and removed the "Clock Frequency" section from the document. *Note:* Removed clock frequencies from document because there are multiple platforms discussed in this document.
2018.04.11	1.0 (supported with Intel Quartus Prime Pro Edition 17.0)	• Document restructured to explicitly define the differences between the Intel FPGA PAC and the Integrated FPGA Platform. • Added IRQ ordering with respect to writes, reads, and write fences added to the "Memory Requests" section.