Streaming DMA Accelerator Functional Unit User Guide

Intel® Programmable Acceleration Card with Intel® Arria® 10 GX FPGA

Updated for Intel® Acceleration Stack for Intel® Xeon® CPU with FPGAs: **1.2.1**

Subscribe
Send Feedback

UG-20171 | 2020.03.06
Latest document on the web: PDF | HTML

Contents

1. About this Document

This document describes the streaming direct memory access (DMA) Accelerator Functional Unit (AFU) implementation and how to build the design to run on hardware or in simulation.

1.1. Intended Audience

This document is intended for hardware or software developer who requires an Accelerator Function (AF) that accesses the data buffered in memory and provides it to an accelerator as a serial stream of data. Intel recommends you gain familiarity with Platform Designer before using this design example.

Related Information

Intel Quartus Prime Pro Edition User Guide: Platform Designer User Guide

1.2. Conventions

Table 1. Document Conventions

Convention	Description
#	If this symbol precedes a command, enter the command as a root.
$	If this symbol precedes a command, enter the command as a user.
`This font`	Indicates file names, commands, and keywords. The font also indicates long command lines. For long command lines, press Enter only if the next line starts a new command, where the # or $ character denotes the start of the next command.
<variable_name>	Indicates placeholder text that you must replace with appropriate values. Do not include the angle brackets.

1.3. Acronyms

Table 2. Acronyms

Acronyms	Expansion	Description
AF	Accelerator Function	Compiled Hardware Accelerator image implemented in FPGA logic that accelerates an application.
AFU	Accelerator Functional Unit	Hardware Accelerator implemented in FPGA logic which offloads a computational operation for an application from the CPU to improve performance.
API	Application Programming Interface	A set of subroutine definitions, protocols, and tools for building software applications.
CCI-P	Core Cache Interface	CCI-P is the standard interface AFUs use to communicate with the host.

continued...

Acronyms	Expansion	Description
DFH	Device Feature Header	Creates a linked list of feature headers to provide an extensible way of adding features.
FIM	FPGA Interface Manager	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	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.
MPF	Memory Properties Factory	The MPF is a Basic Building Block (BBB) that AFUs can use to provide CCI-P traffic shaping operations for transactions with the FIU.
BBBs	Intel® Basic Building Block	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.

1.4. Acceleration Glossary

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

Term	Abbreviation	Description
Intel Acceleration Stack for Intel Xeon® CPU with FPGAs	Acceleration Stack	A collection of software, firmware and tools that provides performance-optimized connectivity between an Intel FPGA and an Intel Xeon processor.
Intel Programmable Acceleration Card with Intel Arria® 10 GX FPGA	Intel PAC with Intel Arria 10 GX FPGA	PCIe accelerator card with an Intel Arria 10. Programmable Acceleration Card is abbreviated PAC. Contains an FPGA Interface Manager (FIM) that pairs with an Intel Xeon processor over PCIe bus.
OPAE_PLATFORM_ROOT		A Linux shell environment variable set up during the process of installing the OPAE SDK delivered with the Acceleration Stack.

Send Feedback

2. Streaming DMA AFU Description

The streaming DMA AFU design example shows how to transfer data between the memory and Avalon®-ST sources and sinks. Most commonly, a streaming DMA is utilized to transfer data from host memory into a hardware accelerator and stream the results back to host memory without using the local FPGA memory as a temporary buffer. These streams typically operate in parallel mode and reduce the latency of a hardware accelerator by removing the additional memory copy operations.

The streaming DMA AFU comprises of the following sub-modules:

- Memory Properties Factory (MPF) Basic Building Block (BBB)
- Core Cache Interface (CCI-P) to Avalon-MM Adapter
- Streaming DMA Test System, which includes:
  - Memory-to-Stream (M2S) DMA BBB
  - Steam-to-Memory (S2M) DMA BBB
  - Streaming Pattern Checker and Generator

The streaming DMA AFU design example includes a user space driver as well as a host application that performs data transfer between host memory and the FPGA pattern checker and generator. You can use this design example as a starting point to implement streaming data transfers in your own AFU design by replacing the pattern checker and generator with your hardware accelerator and modifying the host application accordingly.

Both M2S and S2M DMA BBBs support packetized data, therefore the streaming data includes the start-of-packet (SOP), end-of-packet (EOP), and empty signals. You can use this packet support to transfer a hardware driven payload size. For example, a compression accelerator typically receives a known payload size; and the compression results have an unknown length until the accelerator completes this task. The compression accelerator simply issues a packet to the S2M DMA BBB and the driver provides the host application metadata that describes how much data was transferred.

Related Information

Avalon Interface Specifications

2.1. Hardware Subsystems

The Streaming DMA AFU accesses the host memory through the FPGA Interface Unit (FIU) and the local SDRAM directly. In practice the streaming DMAs typically only need to be connect to the FIU to access host memory. The streaming DMAs can access up to 256 TB of local memory.

You can use the streaming DMA AFU to perform the following data transfer:

- Host memory to FPGA stream
- FPGA stream to host memory
- Local FPGA memory to FPGA stream(1)
- FPGA stream to local FPGA memory(1)

The Platform Designer system implements most part of the streaming DMA AFU, M2S and S2M DMA BBBs.

The Streaming DMA AFU implemented in the Platform Designer system can be found in the following location:

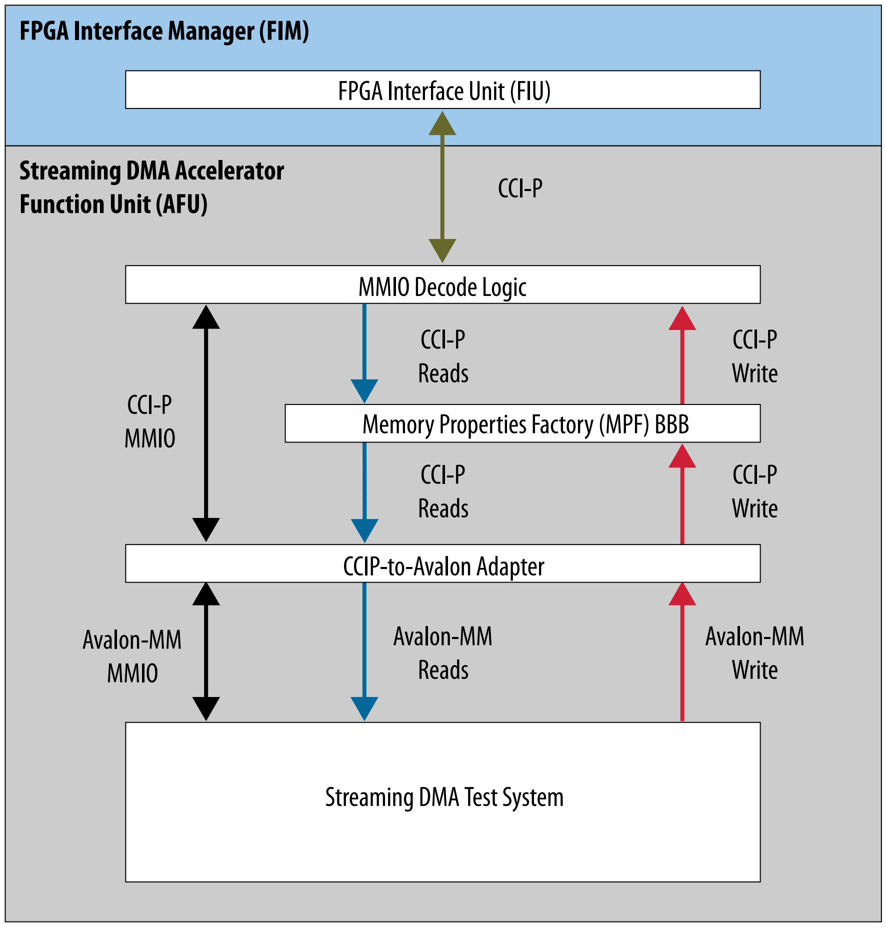
```
$OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu/hw/rtl/<device>/
```

You can find the two DMA BBBs in the following location:

- S2M DMA BBB:
  ```
  $OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu/hw/rtl/
  stream_to_memory_dma_bbb
  ```
- M2S DMA BBB:
  ```
  $OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu/hw/rtl/
  memory_to_stream_dma_bbb
  ```

(1) Supported in a future release of the Intel Acceleration Stack.

Figure 1. High Level System Diagram


FPGA Interface Manager (FIM)
FPGA Interface Unit (FIU)
CCI-P
Streaming DMA Accelerator Function Unit (AFU)
MMIO Decode Logic
CCI-P MMIO
CCI-P Reads
CCI-P Write
Memory Properties Factory (MPF) BBB
CCI-P Reads
CCI-P Write
CCIP-to-Avalon Adapter
Avalon-MM MMIO
Avalon-MM Reads
Avalon-MM Write
Streaming DMA Test System

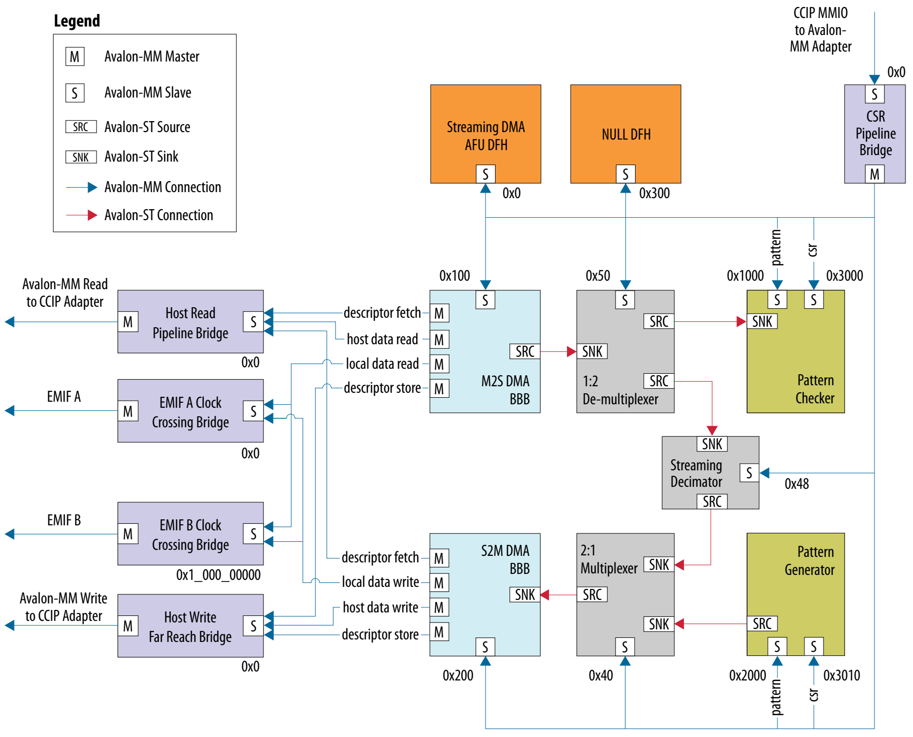
The streaming DMA AFU includes the following modules that connect to the FIU:

- Memory-Mapped IO (MMIO) Decode Logic—detects MMIO read and write transactions and separates them from the CCI-P RX channel 0 that they arrive from. This ensures that MMIO traffic never reaches the MPF BBB and is serviced by an independent MMIO command channel.
- MPF BBB—ensures that reads issued by the M2S DMA BBB are returned in the order that they were issued. The streaming DMA BBBs use the Avalon-MM protocol which requires the read data to return in-order.
- CCI-P to Avalon-MM Adapter—translates MMIO accesses to Avalon-MM read and write transactions. This module also receives Avalon-MM read and write transactions from the streaming DMA BBBs and converts them to CCI-P transactions that are issued to the host.
- Streaming DMA Test System—a wrapper around the two streaming DMA BBBs and includes pattern checker and generator components. This module exposes Avalon-MM master and slave interfaces that connect to the CCI-P to Avalon-MM adapter.

2.2. Streaming DMA Test System

The streaming DMA test system is a Platform Designer system that connects the streaming DMA BBBs to other IP in the system.

Figure 2. Streaming DMA Test System Block Diagram



The streaming DMA test system includes the following modules:

- AFU DFH—stores the 64-bit device feature header (DFH) for the streaming DMA AFU. The host software enumerates the DFH list (scans) that is searching for the AFU. The DMA driver enumerates the DFH list that is searching for DMA BBBs. The AFU DFH is setup to point to the next DFH at offset 0x100.
- M2S DMA BBB—reads buffers from memory and provides the data as a serial stream to the Avalon-ST source port. In this design example, the streaming data is sent to the pattern checker.
- S2M DMA BBB—accepts a serial stream of data from its Avalon-ST port and writes the data to buffers in memory. In this design example, the streaming data is sent from the pattern generator.
- Pattern Checker and Generator—these modules are programmed by the host with an incrementing pattern. The supplied host software configures each component with a pattern that increments by one for every increasing byte.
- Clock Crossing Bridge—this module has been added between the streaming DMAs and the local FPGA external memory to operate the streaming DMA AFU in the `pClk` clock domain.

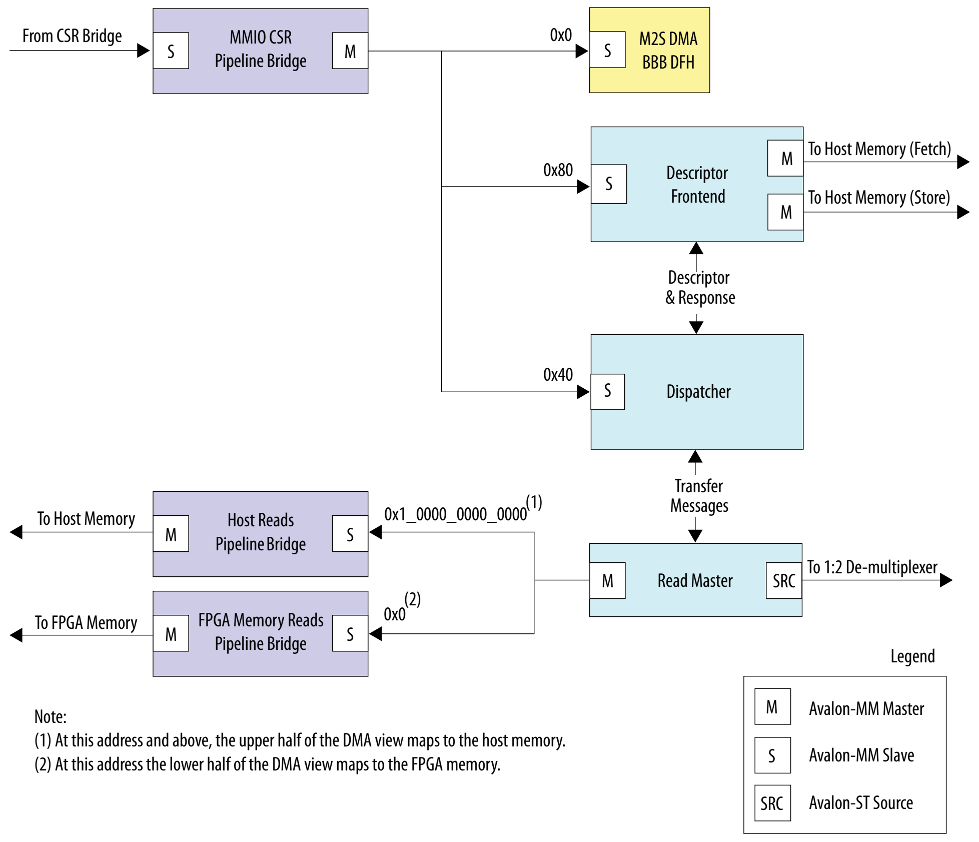
- Pipeline Bridge—this module has been added between the M2S DMA BBB and host read interface of the CCI-P to Avalon-MM adapter to improve the maximum operating frequency (Fmax) of the streaming DMA AFU.
- Far Reach Avalon-MM Bridge—this module has been added between the S2M DMA BBB and host write interface of the CCI-P to Avalon-MM adapter to improve the maximum operating frequency (Fmax). It also sends write responses from the CCI-P interface to the S2M DMA.
- Null DFH—A DFH with its last DFH field set to terminate the DFH list. This module helps you to add more DMA channels to the design and have a module to terminate the DFH list.
- Streaming Decimator—performs loopback testing that programmatically filters out streaming data. This block emulates a hardware accelerator that performs reduction operations (compression for example). It can also be configured for pass-through operation.
- Streaming Multiplexer/De-multiplexer—2:1 and 1:2 multiplexer and de-multiplexer that route the streaming data either to the pattern checker and generator or perform loopback testing between the M2S and S2M DMAs.

2.3. Memory-to-Stream DMA BBB

The Memory-to-Stream (M2S) DMA BBB reads data from a buffer stored in memory and converts it into an Avalon-ST source stream. The buffer must be aligned to 64 bytes. The M2S DMA BBB is configured to handle up to a 1 gigabyte (GB) transfer size, which requires a buffer to be allocated with a 1 GB hugepage to ensure it resides in continuous physical memory. The M2S DMA BBB can also transfer payloads up to 4 KB and 2 MB of size depending on the page size used when allocating the pinned memory.

The M2S DMA BBB streaming interface supports packet generation by exposing the start-of-packet (SOP), end-of-packet (EOP), and empty signals. Your host application can optionally instruct the streaming DMA driver to generate packetized data. If you enable the packetized data, then the empty signal conveys the number of bytes at the end of a transfer that are invalid when the EOP signal is asserted. For example, a DMA transfer of 4100 bytes contains 65 beats of streaming data with SOP asserted during the first beat and EOP asserted during the last beat. The empty signal is set to 60 (0x3C) on the last streaming beat. Since the first 64 beats transfer 64 bytes each, the last beat only contains four valid bytes (first four bytes out of 64 are valid).

Figure 3. M2S DMA BBB Platform Designer System



The components in the M2S DMA BBB Platform Designer system implement the following functions:

- M2S DMA BBB DFH—stores the 64-bit device feature header (DFH) for the M2S DMA BBB. The host driver scans the hardware that is searching for the DMA BBBs. The M2S DMA BBB DFH is setup to point to the next DFH at offset 0x100.
- Dispatcher—buffers descriptors before issuing read transfer commands to the read master.
- Read Master—accepts commands from the dispatcher and reads from memory and converts the data to an Avalon-ST stream. The data leaving the streaming port can be accompanied by streaming sideband signaling for SOP, EOP, and empty signals. If you require the stream to support non-multiples of 64 bytes, then you must request the driver to send packetized data. Therefore, if the last beat is not 64 bytes in size, then the empty signal informs your downstream hardware about the invalid bytes. Only the last beat can contain invalid bytes, all other beats must be 64 bytes in size which is defined by the Avalon-ST specification.
- Pipeline Bridge—To improve the maximum operating frequency (Fmax) of the M2S DMA BBB, the following pipeline bridge components have been added:
  - MMIO CSR Pipeline Bridge: Connects to all the Avalon slaves inside the DMA BBB (Descriptor Frontend, Dispatcher, DMA BBB DFH) and span an address range of 0x100.
  - Host Reads Pipeline Bridge: Reads data from host memory. Added between the Read Master and host memory.
  - FPGA Memory Reads Pipeline Bridge: Reads data from FPGA memory. Added between the Read Master and FPGA memory.

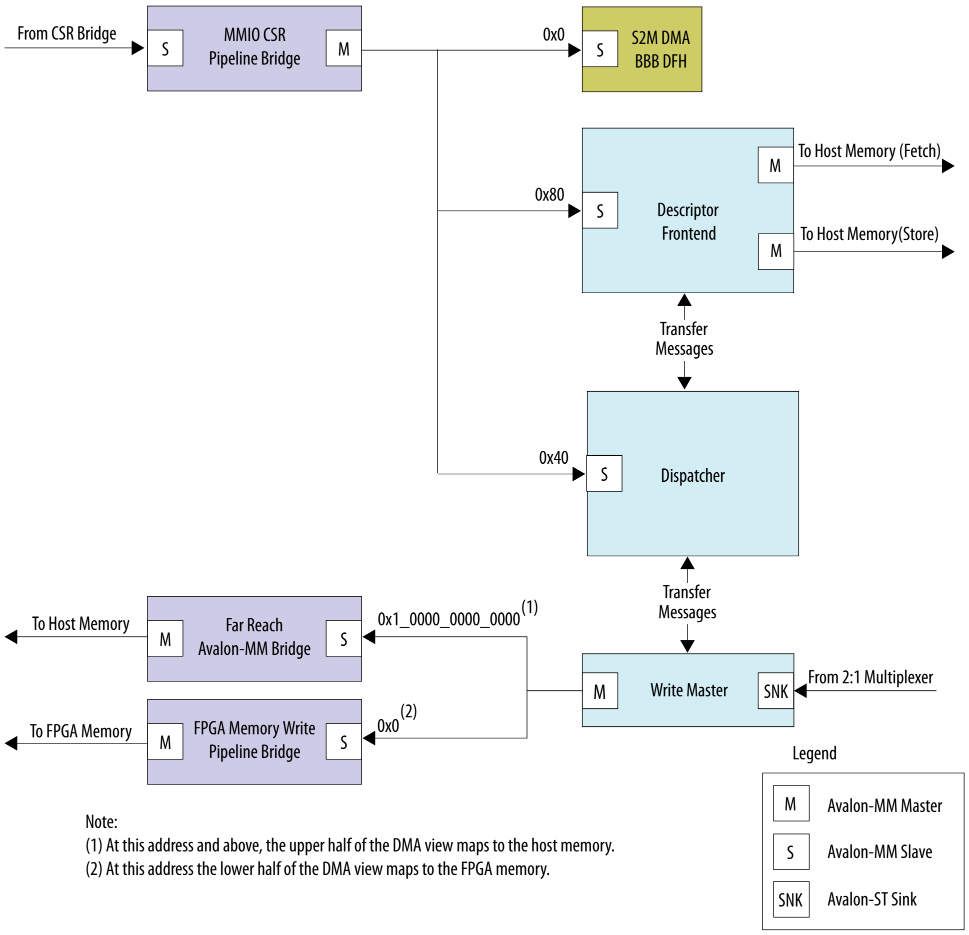
  If your design does not require the M2S DMA BBB to connect to local FPGA memory, then export the pipeline bridge master interface and ground all of its master inputs.
- Descriptor Frontend—fetches transfer descriptors from the host memory and overwrites them with the status information after the transfer completes.

2.4. Stream-to-Memory DMA BBB

The Steam-to-Memory (S2M) DMA BBB accepts Avalon-ST data and transfers it to a buffer in memory. The buffer must be aligned to 64-bytes. The S2M DMA BBB is configured to handle up to a 1 GB transfer size, which requires a buffer to be allocated with a 1 GB hugepage to ensure it resides in continuous physical memory.

The S2M DMA BBB streaming interface supports receiving packetized data by exposing the SOP, EOP, and empty signals. Your host application instructs the streaming DMA driver to use the packet signaling when it requests a streaming transfer. By using the packetized data, the hardware accelerator that provides the data can determine when transfer complete. For example, if a data compression engine is connected to the S2M DMA BBB, the host application does not know how much data might stream until the compression operation is complete. Instead of dividing this data into frames, your hardware accelerator simply notifies the start and end of the payload via asserting SOP and EOP respectively. The DMA transfers the entire payload to memory using one or more DMA transfers and DMA driver instructs the host application of the payload length of each transfer upon completion.

Figure 4. S2M DMA BBB Platform Designer System



Note:
(1) At this address and above, the upper half of the DMA view maps to the host memory.
(2) At this address the lower half of the DMA view maps to the FPGA memory.

The components in the S2M DMA BBB Platform Designer system implement the following functions:

- S2M DMA BBB DFH—stores the 64-bit device feature header (DFH) for the S2M DMA BBB. The host driver scans the hardware that is searching for the DMA BBBs. The S2M DMA DMA BBB DFH points to the next DFH at offset 0x100.
- Dispatcher—buffers descriptors before issuing read transfer commands to the read master.
- Write Master—accepts commands from the dispatcher and writes the data accepted by the Avalon-ST sink interface to memory. The data arriving at the streaming port can be accompanied by streaming sideband signaling for SOP, EOP, and empty signals.

- Pipeline Bridge— To improve the maximum operating frequency (Fmax) of the S2M DMA BBB, the following pipeline bridge components have been added:
  - MMIO CSR Pipeline Bridge: Connects to all the Avalon slaves inside the DMA BBB (Descriptor Frontend, Dispatcher, DMA BBB DFH) and span an address range of 0x100.
  - FPGA Memory Write Pipeline Bridge: Writes data to FPGA memory. Added between the Write Master and FPGA memory.

  If your design does not require the S2M DMA BBB to connect to local FPGA memory, then export that pipeline bridge master interface and ground all of its inputs.
- Far Reach Avalon-MM Bridge—this component has been added between the Write Master and host write interface of the CCI-P to Avalon-MM adapter to improve the maximum operating frequency (Fmax) of the S2M DMA BBB. It also forwards write responses to the write master.
- Descriptor Frontend—fetches transfer descriptors from the host memory and overwrites them with the status information after the transfer completes.

3. Memory Map and Address Spaces

The streaming DMA AFU has three memory views:

- DMA view
- Host view
- DMA Descriptor view

The DMA view supports a 49-bit address space. The lower half of the DMA view maps to the local FPGA memory. Only the streaming DMA BBBs have connectivity to the local FPGA memory, the host cannot access the local FPGA memory. The upper half of the DMA view maps to host memory.

The host view includes all the registers accessible through MMIO accesses such as DFH table, and control/status registers of the various components that are used inside the streaming DMA AFU.

The DMA Descriptor view is a 48-bit address space that maps to the host memory. Because the DMA Fetch engine only accesses the host memory, it sees host memory at address 0x00 unlike the DMA view.

The MMIO registers in both streaming DMA BBBs and the streaming DMA AFU support 32- and 64-bit access. The streaming DMA AFU does not support 512-bit MMIO accesses. The dispatcher registers inside each streaming DMA BBB must be accessed using 32-bit accesses.

3.1. Streaming DMA AFU Memory Map

The streaming DMA register map provides the absolute addresses of all the locations within the unit. These registers are in the host view because only the host can access them.

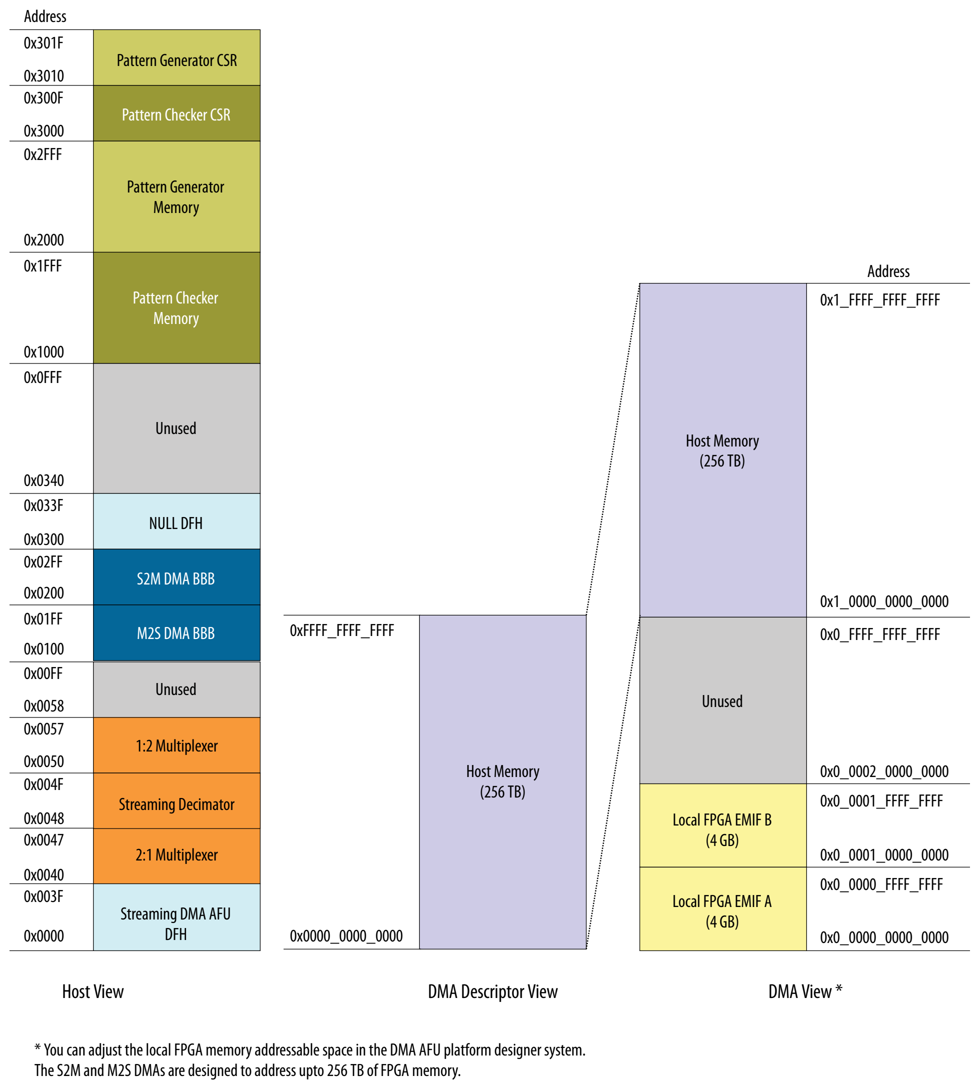
Table 4. Streaming DMA AFU Memory Map

Byte Address	Register Name	Span in Bytes	Description
0x0000	Streaming DMA AFU DFH	0x40	Device feature header for the streaming DMA AFU. This DFH points to 0x100 as the next DFH offset.
0x0040	2:1 Multiplexer	0x8	Routes the streaming data from either the pattern generator or the Decimator to the S2M BBB.
0x0048	Streaming Decimator	0x8	Performs loopback testing that programmatically filters out streaming data.

continued...

Byte Address	Register Name	Span in Bytes	Description
0x0050	1:2 De-multiplexer	0x8	Routes the streaming data to pattern checker and generator or perform loopback testing between the M2S and S2M DMAs.
0x0100	M2S DMA BBB	0x100	Memory-to-stream DMA BBB. The M2S DMA BBB points to 0x100 as the next DFH offset.
0x0200	S2M DMA BBB	0x100	Stream-to-memory DMA BBB. The S2M DMA BBB DFH points to 0x100 as the next DFH offset.
0x0300	NULL DFH	0x40	Null device feature header terminating the DFH linked list.
0x1000	Pattern Checker Memory Slave	0x1000	Pattern checker memory populated by the host application.
0x2000	Pattern Generator Memory Slave	0x1000	Pattern generator memory populated by the host application
0x3000	Pattern Checker CSR Slave	0x10	Pattern checker control and status registers
0x3010	Pattern Generator CSR Slave	0x10	Pattern generator control and status registers.

Figure 5. Streaming DMA AFU Memory Views



3.2. Memory-to-Stream DMA BBB Memory Map

The M2S DMA BBB memory map provides the address offsets of all the locations within the BBB. The following streaming DMA AFU registers reside at offset 0x100 in the MMIO address space.

Send Feedback

Table 5. Memory-to-Stream DMA BBB Memory Map

Byte Address Offsets	Slave Name	Span in Bytes	Description
0x00	M2S DMA BBB DFH	0x40	Device feature header for the M2S DMA BBB. This DFH points to 0x100 as the next DFH offset.
0x40	M2S DMA Dispatcher CSR	0x20	Control port for the mSGDMA within the memory-to-stream DMA BBB. The driver accesses this location to control the DMA or query its status.
0x60	M2S DMA Descriptor	0x20	Descriptor port for the mSGDMA within the memory-to-stream DMA BBB. The driver writes descriptors to this location.

3.3. Stream-to-Memory DMA BBB Memory Map

The S2M DMA BBB memory map provides the address offsets of all the locations within the BBB. The following streaming DMA AFU registers reside at offset 0x200 in the MMIO address space.

Table 6. Stream-to-Memory DMA BBB Memory Map

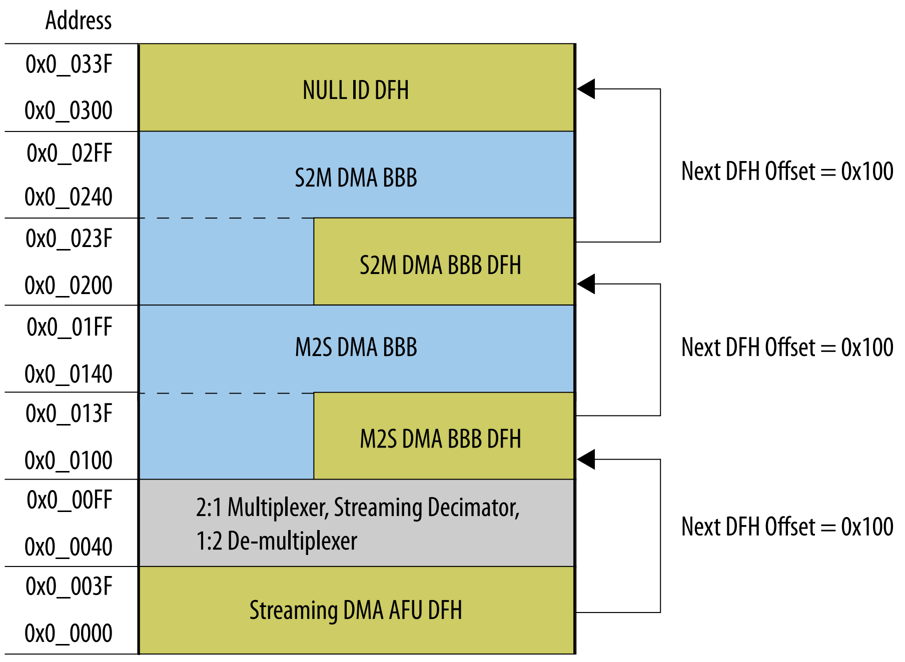
Byte Address Offsets	Slave Name	Span in Bytes	Description
0x00	S2M DMA BBB DFH	0x40	Device feature header for the S2M DMA BBB. This DFH points to 0x100 as the next DFH offset.
0x40	S2M DMA Dispatcher CSR	0x20	Control port for the mSGDMA within the stream-to-memory DMA BBB. The driver accesses this location to control the DMA or query its status.
0x80	S2M DMA Response	0x8	Response port for the mSGDMA within the stream-to-memory DMA BBB. The driver reads this port to determine how much data was streamed to the memory.
0x80	S2M DMA Descriptor Frontend CSR	0X40	Control port for the S2M DMA descriptor frontend. The driver accesses this location to control the descriptor frontend or query its status.

3.4. Device Feature Header Linked-list

The streaming DMA AFU design example contains four device feature headers (DFH) that form a linked list. This linked list allows the sample application to identify the streaming DMA AFU as well as the driver to identify each of the streaming DMA BBBs.

A NULL DFH is included at the end of the list. The inclusion of the null DFH at the end of the linked list allows you to add more streaming DMA BBBs to your design. You simply need to move the NULL DFH to an address after the other BBBs. Each streaming DMA BBB expects the next DFH to be located 0x100 bytes from the base address of the BBB. The following figure depicts the linked-list for the streaming DMA AFU design example.

Figure 6. Streaming DMA AFU Device Feature Header (DFH) Chaining



If you want two M2S and two S2M DMA BBBs in your design, then you can use the following address map to implement four streaming channels. The four streaming DMA BBBs can reside anywhere in the address map if they are packed together in the MMIO address space every 0x100 bytes. The DFH that follows the streaming DMA BBB must be located at offset 0x100 from the previous streaming DMA BBB channel and it can be the NULL DFH or other DFHs.

Table 7. Four-channel Streaming DMA AFU Example Configuration

Byte Address	Register Name	Span in Bytes	Description
0x000	Streaming DMA AFU DFH	0x40	Your AFU DFU. This DFH points to 0x100 as the next DFH offset.
0x100	M2S DMA BBB #1	0x100	First memory-to-stream DMA BBB. Next DFH set to 0x100.
0x200	M2S DMA BBB #2	0x100	Second memory-to-stream DMA BBB. Next DFH set to 0x100.
0x300	S2M DMA BBB #1	0x100	First stream-to-memory DMA BBB. Next DFH set to 0x100.
0x400	S2M DMA BBB #2	0x100	Second stream-to-memory DMA BBB. Next DFH set to 0x100.
0x500	NULL DFH	0x40	Null DFH at the end of the linked list.

4. Software Programming Model

The streaming DMA AFU includes a software driver that you can use in your own host application. The `fpga_dma_st.c` and `fgpa_dma.h` files located at the following location implement the software driver:

```
$OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu/sw
```

This driver supports the following functions:

API	Description
`fpgaCountDMAChannels`	Scans the device feature chain for DMA BBBs and count all available channels.
`fpgaDMAOpen`	Opens a handle to the DMA channel.
`fpgaDMAClose`	Closes a handle to the DMA channel.
`fpgaGetDMAChannelType`	Query DMA channel type. Possible type of query channel is TX streaming (TX_ST) and RX streaming (RX_ST).
`fpgaDMATransferInit`	Initializes an object that represents the DMA transfer.
`fpgaDMATransferReset`	Resets the DMA transfer attribute object to default values.
`fpgaDMATransferDestroy`	Destroys the DMA transfer attribute object.
`fpgaDMATransferSetSrc`	Sets the source address of the transfer. This address must be 64 byte aligned.
`fpgaDMATransferSetDst`	Sets the destination address of the transfer. This address must be 64 byte aligned.
`fpgaDMATransferSetLen`	Sets the transfer lengths in bytes. For non-packet transfers, you must set the transfer length to a multiple of 64 bytes. For packet transfers, this is not a requirement.
`fpgaDMATransferSetTransferType`	Sets the transfer type. Legal values are: • HOST_MM_TO_FPGA_ST = TX (host to AFU streaming) • FPGA_ST_TO_HOST_MM = RX (AFU streaming to host)
`fpgaDMATransferSetTxControl`	Sets TX control. This allows the driver to optionally generate in-band SOP and EOP in the data stream sent from the TX DMA. TX control is only valid for HOST_MM_TO_FPGA_ST transfer. Valid values are: • TX_NO_PACKET (No SOP, EOP, or empty value generated) • GENERATE_SOP_AND_EOP • GENERATE_SOP • GENERATE_EOP

continued...

API	Description
`fpgaDMATransferSetRxControl`	Sets RX control. This allows the driver to handle an unknown amount of receive data from the FPGA, When END_ON_EOP is set, the RX DMA ends the transfer when EOP arrives in the receive stream or when `rx_count` bytes have been received (whichever occurs first). RX control is only valid for FPGA_ST_TO_HOST_MM transfer. Valid values are: • RX_NO_PACKET (deterministic length transfer) • END_ON_EOP
`fpgaDMATransferSetTransferCallback`	Registers callback for notification on asynchronous transfer completion. If you specify a callback, `fpgaDMATransfer` returns immediately (asynchronous transfer). If you do not specify a callback, `fpgaDMATransfer` returns after the transfer is complete (synchronous/blocking transfer).
`fpgaDMATransferGetBytesTransferred`	Returns the number of bytes transferred by an RX transfer request. The application uses this data when receiving packetized data (`rx_control` set to END_ON_EOP when transfer request was issued).
`fpgaDMATransferCheckEopArrived`	Retrieves EOP status Legal vales are: • 0: EOP did not arrive • 1: EOP arrived
`fpgaDMATransferSetLast`	Indicates the last transfer so the DMA can start processing the prefetched transfers. The default value is 64 transfers in the pipeline before the DMA starts to work on the transfers.
`fpgaDMATransfer`	Performs a DMA transfer.

For more information about the API, input, and output arguments, refer to the header file located at `$OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu/sw/fpga_dma.h`

To know more about software driver use model, refer to the README file located at `$OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu/README.md`

5. Running the AFU Example

Before you begin:

- Intel recommends you refer to the Quick Start Guide for your Intel PAC with Intel Arria 10 GX FPGA to be familiar with running similar examples. Before you proceed through the following steps, verify that the OPAE_PLATFORM_ROOT environment variable is set to the OPAE SDK installation directory.
- The sample application requires two 1 GB hugepages. Refer to the Enabling Hugepages on page 29 section for details on how to set the hugepages on both Red Hat Enterprise Linux (RHEL) and Ubuntu systems.

Perform the following steps to download the Streaming DMA Accelerator Function (AF) bitstream, to build the application and driver, and to run the design example:

1. Change to the Streaming DMA application and driver directory:

   ```
   cd $OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu/sw
   ```

2. Build the driver and application:

   ```
   make
   ```

3. Download the streaming DMA AFU bitstream:

   ```
   sudo fpgasupdate ../bin/streaming_dma_afu_unsigned.gbs
   ```

4. Execute the host application to transfer 100 MB in 1 MB portions from host memory to the FPGA pattern checker:

   ```
   ./fpga_dma_st_test -l off -s 104857600 -p 1048576 -r mtos -t fixed
   ```

5. Execute the host application to transfer 100 MB in 1 MB portions from the FPGA pattern generator to host memory:

   ```
   ./fpga_dma_st_test -l off -s 104857600 -p 1048576 -r stom -t fixed
   ```

6. Execute the host application to transfer 100 MB in 1 MB portions from host memory back to host memory in loopback mode:

   ```
   ./fpga_dma_st_test -l on -s 104857600 -p 1048576 -t fixed -f 0
   ```

Related Information

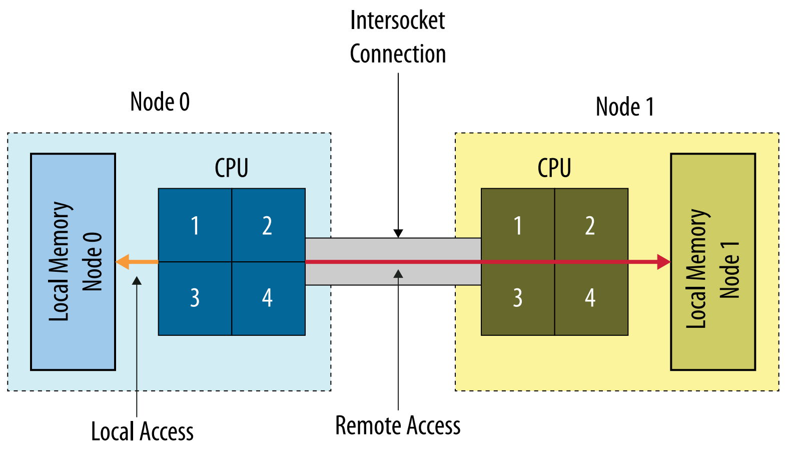
Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA

5.1. Optimization for Improved DMA Performance

Implementation of NUMA (non-uniform memory access) optimization in `fpga_dma_st_test.c` allows the processor to access its own local memory. This implementation is faster than accessing non-local memory (memory local to another processor).

A typical NUMA configuration is shown in the diagram below. The local access arrow represents access from a core to memory local to the same core. The remote access illustrates the path taken when a core on Node 0 accesses memory that resides in memory local to Node 1.

Figure 7. Typical NUMA Configuration



Use the following code to implement NUMA optimization in your test application:

```
// Set up proper affinity if requested
    if (cpu_affinity || memory_affinity) {
        unsigned dom = 0, bus = 0, dev = 0, func = 0;
        fpga_properties props;
        int retval;
        #if(FPGA_DMA_DEBUG)
                char str[4096];
        #endif
        res = fpgaGetProperties(afc_token, &props);
        ON_ERR_GOTO(res, out_destroy_tok, "fpgaGetProperties");
        res = fpgaPropertiesGetBus(props, (uint8_t *) & bus);
        ON_ERR_GOTO(res, out_destroy_tok, "fpgaPropertiesGetBus");
        res = fpgaPropertiesGetDevice(props, (uint8_t *) & dev);
        ON_ERR_GOTO(res, out_destroy_tok, "fpgaPropertiesGetDevice");
        res = fpgaPropertiesGetFunction(props, (uint8_t *) & func);
        ON_ERR_GOTO(res, out_destroy_tok, "fpgaPropertiesGetFunction");

        // Find the device from the topology
        hwloc_topology_t topology;
        hwloc_topology_init(&topology);
        hwloc_topology_set_flags(topology,
                    HWLOC_TOPOLOGY_FLAG_IO_DEVICES);
        hwloc_topology_load(topology);
        hwloc_obj_t obj = hwloc_get_pcidev_by_busid(topology, dom, bus, dev,
func);
        hwloc_obj_t obj2 = hwloc_get_non_io_ancestor_obj(topology, obj);
        #if (FPGA_DMA_DEBUG)
            hwloc_obj_type_snprintf(str, 4096, obj2, 1);
            printf("%s\n", str);
            hwloc_obj_attr_snprintf(str, 4096, obj2, " :: ", 1);
            printf("%s\n", str);
            hwloc_bitmap_taskset_snprintf(str, 4096, obj2->cpuset);
            printf("CPUSET is %s\n", str);
            hwloc_bitmap_taskset_snprintf(str, 4096, obj2->nodeset);
            printf("NODESET is %s\n", str);
        #endif
        if (memory_affinity) {
            #if HWLOC_API_VERSION > 0x00020000
                retval = hwloc_set_membind(topology, obj2->nodeset,
                                HWLOC_MEMBIND_THREAD, HWLOC_MEMBIND_MIGRATE |
HWLOC_MEMBIND_BYNODESET);
            #else
                retval =
                hwloc_set_membind_nodeset(topology, obj2->nodeset,
                                HWLOC_MEMBIND_THREAD,
                                HWLOC_MEMBIND_MIGRATE);
            #endif
            ON_ERR_GOTO(retval, out_destroy_tok, "hwloc_set_membind");
        }
        if (cpu_affinity) {
            retval = hwloc_set_cpubind(topology, obj2->cpuset,
HWLOC_CPUBIND_STRICT);
            ON_ERR_GOTO(retval, out_destroy_tok, "hwloc_set_cpubind");
        }
    }
```

6. Compiling the Accelerator Function (AF)

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

1. Change to the streaming DMA AFU sample directory:

   ```
   cd $OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu
   ```

2. Generate the design build directory:

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

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

   ```
   cd build_synth

   run.sh
   ```

   The `run.sh` AF generation script creates the AF image with the same base filename as the AFU's platform configuration file with a `.gbs` suffix at the location: `$OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu/build_synth/streaming_dma_afu.gbs`.

4. Create an unsigned copy of the generated `.gbs` file:

   ```
   PACSign PR -t UPDATE -H openssl_manager -i streaming_dma_afu.gbs -o
   streaming_dma_afu_unsigned.gbs
   ```

   Note: If your Intel PAC already implements bitstream authentication, review the steps outlined in the *Security User Guide: Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA* to sign the Streaming DMA bitstream.

Related Information

Security User Guide: Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA

7. Simulating the AFU Example

Intel recommends you refer to the *Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) Quick Start Guide* for your Intel PAC to be familiar with simulating similar examples and to setup your environment. Before you proceed through the following steps, verify that the `OPAE_PLATFORM_ROOT` environment variable is set to the OPAE SDK installation directory.

Complete the following steps to setup the hardware simulator for the streaming DMA AFU:

1. Change to the streaming DMA AFU sample directory:

   ```
   cd $OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu
   ```

2. Create an ASE environment in a new directory and configure it for simulating an AFU:

   ```
   afu_sim_setup --source=./hw/rtl/filelist.txt build_ase_dir
   ```

3. Change to the ASE build directory:

   ```
   cd build_ase_dir
   ```

4. Build the driver and application:

   ```
   make
   ```

5. Make simulation:

   ```
   make sim
   ```

Sample output from the hardware simulator:

```
[SIM]  ** ATTENTION : BEFORE running the software application **
[SIM]  Set env(ASE_WORKDIR) in terminal where application will run (copy-and-
paste) =>
[SIM]  $SHELL   | Run:
[SIM]  ---------+---------------------------------------------------
[SIM]  bash/zsh | export ASE_WORKDIR=/mnt/Tools/ias/hw/samples/
streaming_dma_afu/build_ase_dir/work
[SIM]  tcsh/csh | setenv ASE_WORKDIR /mnt/Tools/ias/hw/samples/
streaming_dma_afu/build_ase_dir/work
[SIM]  For any other $SHELL, consult your Linux administrator
[SIM]
[SIM]  Ready for simulation...
[SIM]  Press CTRL-C to close simulator...
```

Complete the following steps to compile and execute the streaming DMA AFU software in the simulation environment:

1. Open a new terminal window.
2. Change directory to:

```
cd $OPAE_PLATFORM_ROOT/hw/samples/streaming_dma_afu/sw
```

3. **Copy** the environment setup string (choose string appropriate for your shell) from the steps above in the hardware simulation to the terminal window. See the following lines in the sample output from the hardware simulator.

```
[SIM]  bash/zsh | export ASE_WORKDIR=/mnt/Tools/ias/hw/samples/
streaming_dma_afu/build_ase_dir/work
[SIM]  tcsh/csh | setenv ASE_WORKDIR /mnt/Tools/ias/hw/samples/
streaming_dma_afu/build_ase_dir/work
```

4. Compile the software:

```
make USE_ASE=1
```

5. Run one of the following commands:

   - Execute the host application to transfer 4 KB in 1 KB portions from the host memory to the FPGA pattern checker:

   ```
   ./fpga_dma_st_test -l off -s 4096 -p 1024 -r mtos -t fixed
   ```

   - Execute the host application to transfer 4 KB in 1 KB portions from the FPGA pattern checker to the host memory:

   ```
   ./fpga_dma_st_test -l off -s 4096 -p 1024 -r stom -t fixed
   ```

   - Execute the host application to transfer 4 KB in 1 KB portions from the host memory back to host memory in the loopback mode:

   ```
   ./fpga_dma_st_test -l on -s 4096 -p 1024 -t fixed
   ```

   Note: To run any of these commands in this step, run `make sim` in other terminal window first.

Related Information

Intel FPGA Acceleration Hub: Knowledge Center
Provides more information about the related resources, collateral, and training.

8. Streaming DMA AFU User Guide Archives

Intel Acceleration Stack Version	User Guide (PDF)
1.2	Streaming DMA Accelerator Functional Unit (AFU) User Guide
1.1	Streaming DMA Accelerator Functional Unit (AFU) User Guide

9. Document Revision History for Streaming DMA Accelerator Functional Unit User Guide

Document Version	Intel Acceleration Stack Version	Changes
2020.03.06	1.2.1 (supported with Intel Quartus® Prime Pro Edition Edition 19.2)	• Added new appendix section *Enabling Hugepages*. • Updated the *Figure: Streaming DMA Test System Block Diagram*. • Updated the *Figure: M2S DMA BBB Platform Designer System*. • Updated the *Figure: S2M DMA BBB Platform Designer System*. • Updated the following in section *Memory Map and Address Spaces*: — Added DMA descriptor view information — Updated *Figure: Streaming DMA AFU Memory Views* • Added APIs and description table in section *Software Programming Model*. • Updated a command to generate the design build directory in section *Compiling the Accelerator Function (AF)*. • Added a note in section *Simulating the AFU Example* to clarify the successful execution of the host application transfer.
2018.12.04	1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1)	• Updated the *Running the AFU Example* and *Simulating the AFU Example* steps. • Replaced the Write Response Bridge with the Far Reach Avalon-MM Bridge. • Added a new section *NUMA Optimization*. • Added a new chapter: *Streaming DMA AFU User Guide Archives*.
2018.08.06	1.1 (supported with Intel Quartus Prime Pro Edition 17.1.1)	Initial release.

A. Enabling Hugepages

This section covers information about how to enable 1 GB hugepages temporarily and persistently. Intel recommends you to begin by temporarily enabling huge pages and later if you want to avoid revisiting the steps to enable hugepages persistently.

Temporarily Enabling 1 GB Hugepages

If you have already boot your system into the operating system, you can enable two 1 GB hugepages by running the following command:

```
sudo sh -c "echo 2 > /sys/kernel/mm/hugepages/hugepages-1048576kB/nr_hugepages"
```

You can also enable two 1 GB hugepages and twenty 2 MB hugepages at boot time by passing the following boot time arguments to GRUB:

```
default_hugepagesz=2MB hugepagesz=1G hugepages=2 hugepagesz=2M hugepages=20
```

By enabling 2 MB hugepages as the default hugepage size, when software allocates buffers between 4 KB and 2 MB the memory allocation system will default to 2 MB hugepages instead of 1 GB hugepages. Some of the other acceleration stack examples require up to twenty 2 MB hugepages so setting both page sizes as shown above will allow those designs to continue operating correctly using the same settings.

Persistently Enabling 1 GB Hugepages

This section explains how to enable two 1 GB hugepages and twenty 2 MB hug pages persistently so that the setting remains persistent across reboots and power cycles.

- **In Ubuntu**: Perform the following steps to enable two 1 GB hugepages and twenty 2 MB hugepages on a Ubuntu system:
  1. Edit `/etc/default/grub` and add the following text to the end of the file:
     ```
     GRUB_CMDLINE_LINUX_DEFAULT="${GRUB_CMDLINE_LINUX_DEFAULT}
     default_hugepagesz=2MB hugepagesz=1G hugepages=2 hugepagesz=2M
     hugepages=20"
     ```
  2. Save the GRUB file.
  3. Update GRUB by committing the updated settings:
     ```
     sudo update-grub
     ```
  4. Reboot the system.
- **In RHEL**: Perform the following steps to enable two 1 GB hugepages and twenty 2 MB hugepages on a RHEL system:
  1. Edit `/etc/default/grub` and add the following text to the end of the file:
     ```
     GRUB_CMDLINE_LINUX="${GRUB_CMDLINE_LINUX} default_hugepagesz=2MB
     hugepagesz=1G hugepages=2 hugepagesz=2M hugepages=20"
     ```
  2. Save the GRUB file.
  3. Update GRUB by committing the updated settings:
     ```
     sudo grub2-mkconfig -o /boot/efi/EFI/redhat/grub.cfg
     ```
  4. Reboot the system.

Persistently Disabling 1 GB Hugepages

To revert your system back to its default boot settings, follow the instructions from section *Persistently Enabling 1 GB Hugepages* and remove the text that you added in step 1, and then follow steps 2 to 4. If you need to periodically enable and disable hugepages, instead of removing test from step 1 simply comment out the line by post-pending "#" to comment the line out.