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-20127 | 2020.03.06

Latest document on the web: PDF | HTML

Contents

Send Feedback

1. About this Document

This document describes the OpenCL* implementation for the Intel® Programmable Acceleration Card with Intel Arria® 10 GX FPGA.

1.1. Intended Audience

The intended audience comprises hardware or software developers that require an Accelerator Function (AF) to buffer data locally in memory connected to the Intel FPGA device.

1.2. Conventions

Table 1. Document Conventions

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

1.3. 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.
DFH	Device Feature Header	Creates a linked list of feature headers to provide an extensible way of adding features.

continued...

Acronyms	Expansion	Description
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.

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.

2. DMA AFU Description

2.1. Introduction

The Direct Memory Access (DMA) AFU example shows how to manage memory transfers between the host processor and the FPGA. You can integrate the DMA AFU into your design to move data between the host memory and the FPGA local memory.

The DMA AFU comprises the following submodules:

- MMIO Decoder Logic
- Memory Properties Factory (MPF) Basic Building Block (BBB)
- Core Cache Interface (CCI-P) to the Avalon® Memory-Mapped (Avalon-MM) Adapter
- DMA Test System which contains the DMA BBB

These submodules are described in more detail in the *DMA AFU Hardware Components* topic below.

Related Information

- The DMA AFU Hardware Components on page 6
- Avalon Interface Specifications
  For more information about the Avalon-MM protocol, including timing diagrams for read and write transactions.

2.2. The DMA AFU Software Package

The Intel Acceleration Stack for Intel Xeon CPU with FPGAs package file (`*.tar.gz`), includes the DMA AFU example. This example provides a user space driver. The host application uses this driver such that the DMA moves data between host and FPGA memory. The hardware binaries, sources, and the user space driver are available in the following directory: *$OPAE_PLATFORM_ROOT*`/hw/samples/dma_afu` .

Before experimenting with the DMA AFU, you must install the Open Programmable Acceleration Engine (OPAE) software package. Refer to *Installing the OPAE Software Package* in the *Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA* for installation instructions. This *Quick Start Guide* also includes basic information about the Open Programmable Acceleration Engine (OPAE) and configuring an AFU.

After installing the Open Programmable Acceleration Engine (OPAE) software package, a sample host application and the DMA AFU user space driver are available in the following directory: *$OPAE_PLATFORM_ROOT*`/hw/samples/dma_afu/sw` .

A sample application, `fpga_dma_test` implements the DMA AFU user space driver.

Related Information

- Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA
- Installing the OPAE Software Package

2.3. The DMA AFU Hardware Components

The DMA AFU interfaces with the FPGA Interface Unit (FIU) and two banks of local DDR4-SDRAM. The total memory addressable on the device is 8 gigabytes (8 GB). The memory comprises two, 4 GB banks.

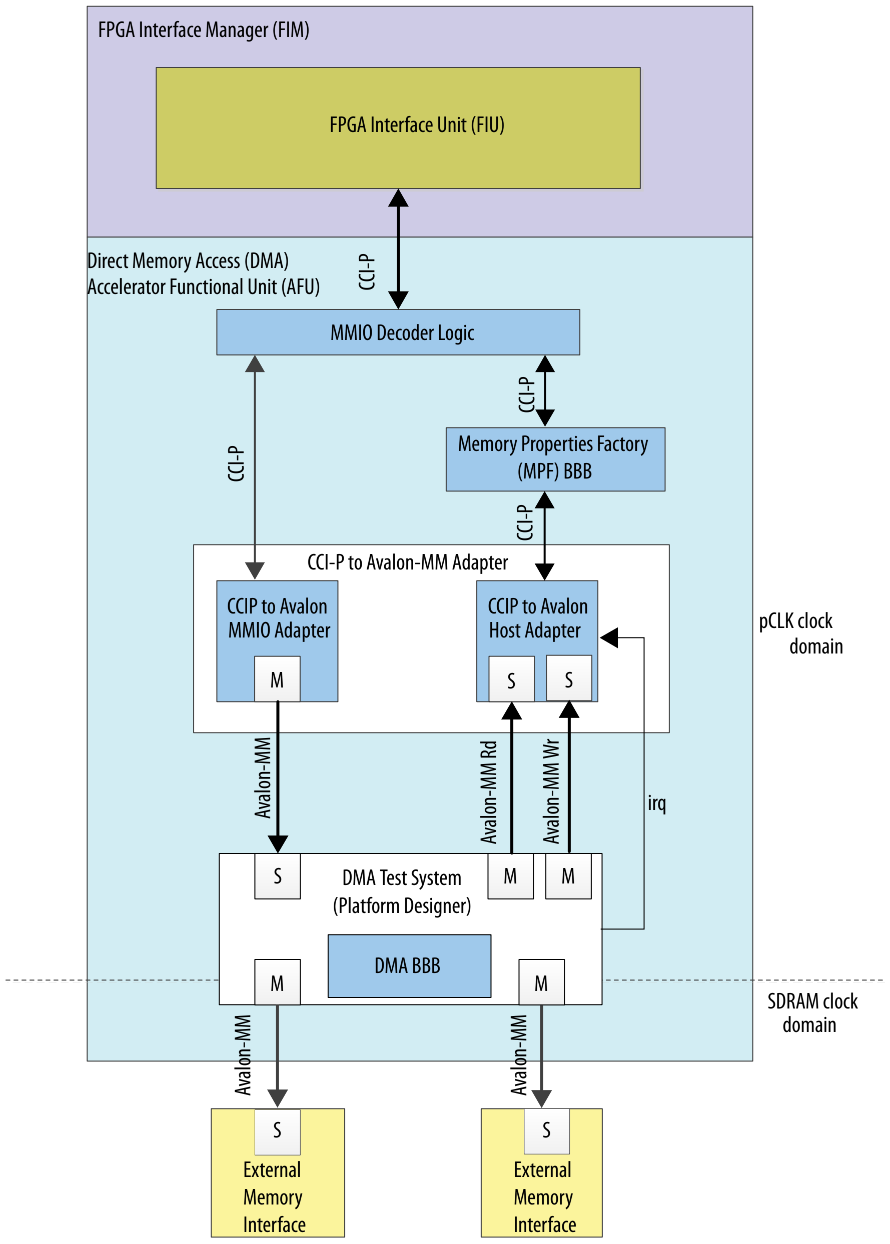
Note: The currently available hardware dictates this memory configuration. Future hardware may support different memory configurations.

You can use the DMA AFU to copy data between the following source and destination locations:

- The host to device FPGA memory
- Device FPGA memory to the host

A Platform Designer system, `$OPAE_PLATFORM_ROOT/hw/samples/dma_afu/hw/rtl/qsys/<`*`device`*`>/dma_test_system.qsys` implements most of the DMA AFU. The DMA BBB subsystem is located at `<installation path>/hw/samples/dma_afu/hw/rtl/qsys/<`*`device`*`>/msgdma_bbb.qsys`

Figure 1. DMA AFU Hardware Block Diagram


FPGA Interface Manager (FIM)
FPGA Interface Unit (FIU)
Direct Memory Access (DMA) Accelerator Functional Unit (AFU)
CCI-P
MMIO Decoder Logic
CCI-P
Memory Properties Factory (MPF) BBB
CCI-P
CCI-P
CCI-P to Avalon-MM Adapter
CCIP to Avalon MMIO Adapter
M
CCIP to Avalon Host Adapter
S
S
pCLK clock domain
Avalon-MM
Avalon-MM Rd
Avalon-MM Wr
irq
S
DMA Test System (Platform Designer)
M
M
DMA BBB
M
M
SDRAM clock domain
Avalon-MM
Avalon-MM
S
External Memory Interface
S
External Memory Interface

The DMA AFU includes the following internal modules to interface with the FPGA Interface Unit (FIU):

- Memory-Mapped IO (MMIO) Decoder 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.
- Memory Properties Factory (MPF): This module ensures that read responses from the DMA return in the order that they were issued. The Avalon-MM protocol requires read responses to return in the correct order.
- CCI-P to Avalon-MM Adapter: This module translates between CCI-P and Avalon-MM transactions, as follows:
  - CCI-P to Avalon-MMIO Adapter: This path translates CCI-P MMIO transactions into Avalon-MM transactions.

    Note: MMIO accesses do not support backpressure. As a result, the CCI-P to Avalon-MM Adapter does not support the `waitrequest` signal. Intel recommends that you add an Avalon-MM Clock Crossing Bridge, available in the IP Catalog, between the CCI-P to Avalon MMIO Adapter master port and the DMA Test System Avalon-MM slave port. Intel recommends that you set the clock crossing command depth to 64 entries deep and disable burst support.
  - Avalon-MM to CCI-P: These paths create separate read-only and write-only paths for the DMA to access host memory.

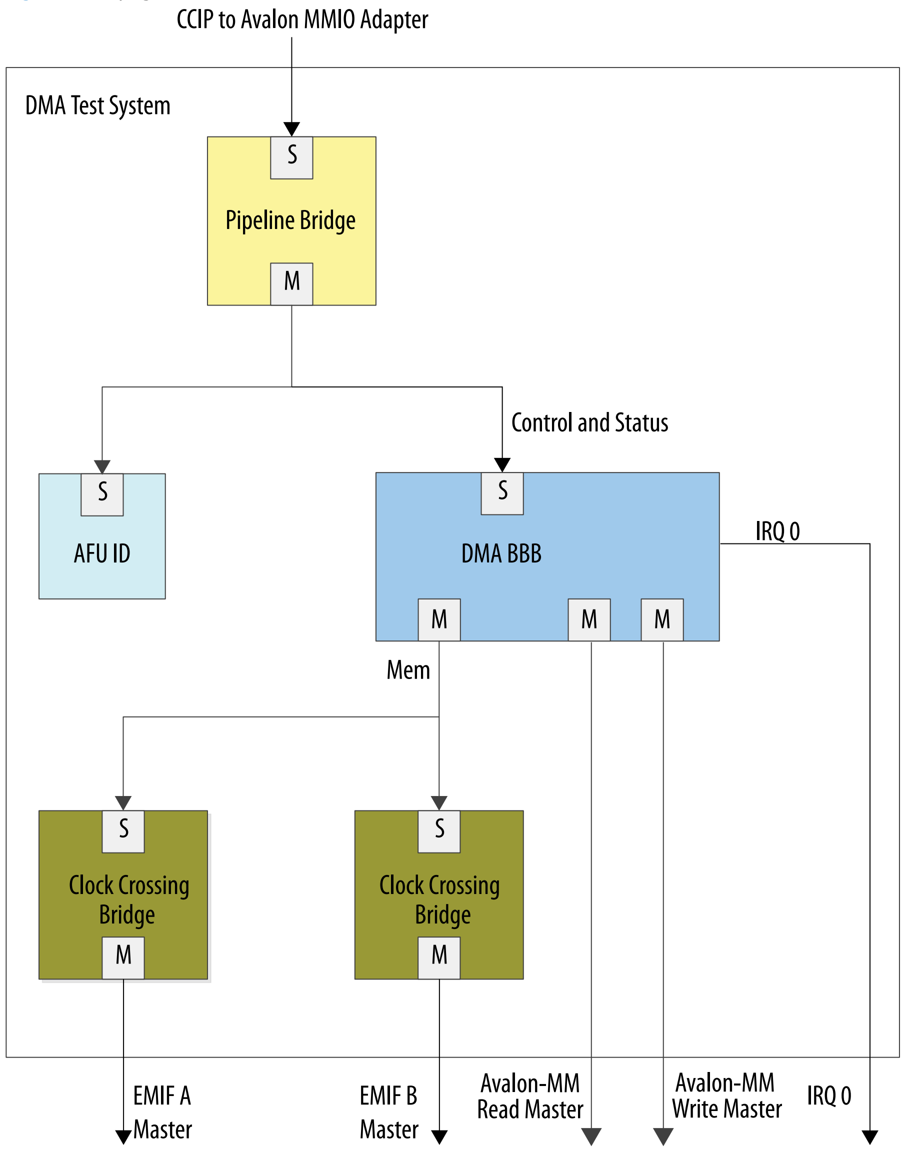
  The Avalon-MM write slave of the CCI-P to Avalon Adapter includes an extra, high-order bit to implement write fences. When the high-order bit is set to 1'b1, the CCI-P adapter first issues a write fence. Then, the CCI-P bridge writes data to the host physical address space with the high-order bit is set to 1'b0. This operation allows the DMA to synchronize writes to host memory. The DMA BBB is not capable of receiving write responses so that the write fence is used to synchronize the write data with the host.
- DMA Test System: This module serves as a wrapper around the DMA BBB to expose the DMA masters and interrupt interfaces to the rest of the logic in the AFU. It provides the interface between the DMA BBB and the CCI-P to Avalon Adapter. It also provides the interface between the DMA BBB and the local FPGA SDRAM banks.

2.3.1. DMA Test System

The DMA test system tests the DMA AFU.

Figure 2. DMA Test System Block Diagram

This block diagram shows the internals of the DMA test system. The DMA test system is shown as a monolithic block in Figure 1 on page 7.



The DMA test system includes the following internal modules:

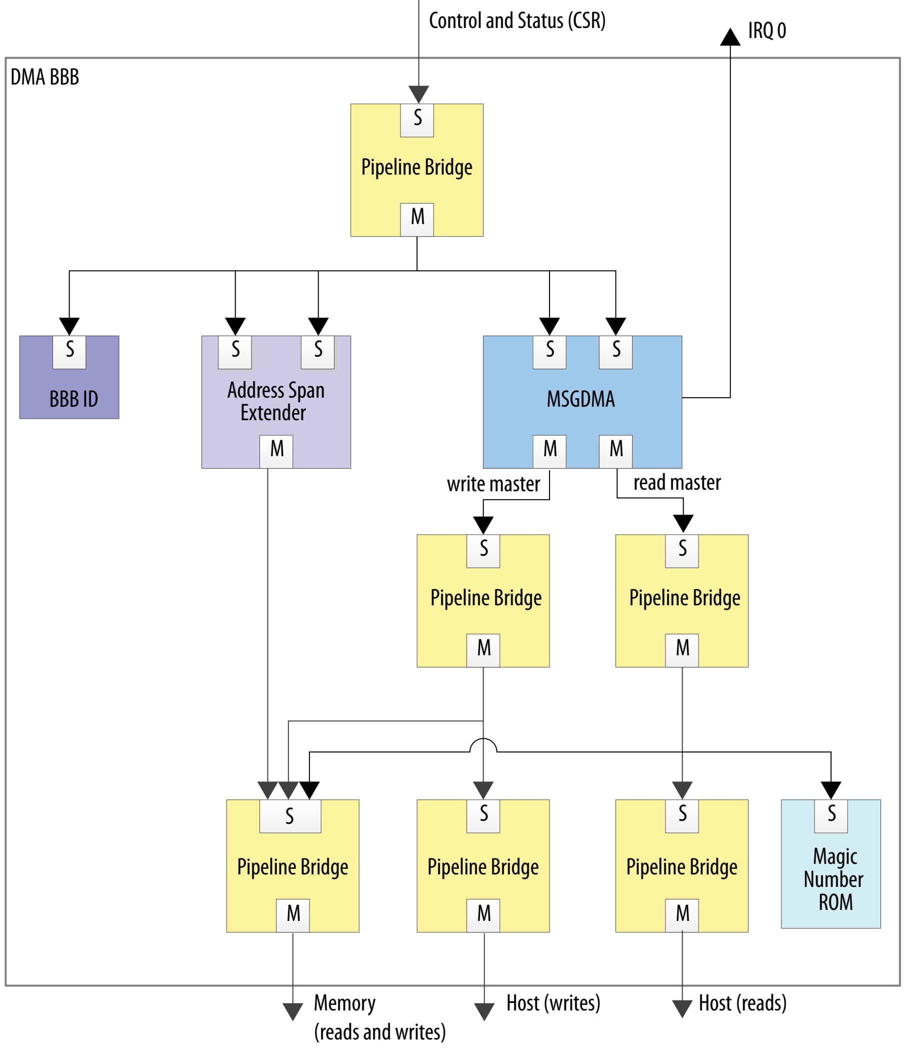
- AFU ID: This component stores the 64-bit `Device Feature Header (DFH)` and also includes the universally unique identifier (UUID). The `AFU_ID_L` register stores the lower 32 bits of the AFU ID. The `AFU_ID_H` register stores the upper 32 bits of the AFU ID. A software driver scans the DMA Test System, finds the AFU ID, and identifies the DMA BBB.
- DMA Basic Building Block (BBB): This component moves data between the host and local device memory spaces. DMA BBB interrupt connects to the `IRQ 0` signal. The `IRQ 0` signal is an input to the CCI-P to Avalon Adapter. The CCI-P to Avalon Adapter forwards the interrupt to the host.
- Pipeline Bridge: The Pipeline Bridge inserts pipeline stages between memory mapped IP cores. By default, Platform Designer optimizes for low latency. Consequently, the Pipeline Bridge improve the system maximum operating frequency (F_{MAX}) at the expense of additional latency.
- Clock Crossing Bridge: The Clock Crossing Bridge isolates Avalon-MM masters and slaves that are in different clock domains. Because the Clock Crossing Bridge includes clock-crossing logic, it adds FIFOs that have a greater latency than the standard Pipeline Bridge. The Clock Crossing Bridge ensures that the memory transactions from the DMA BBB safely cross to the local SDRAM clock domain.

2.3.2. DMA BBB

The DMA BBB subsystem transfers data from source to destination addresses using memory-mapped transactions. The DMA AFU accesses control and status registers in the DMA BBB subsystem. The DMA BBB comprises five IP cores available in the Platform Designer IP Catalog as shown in the following figure.

Figure 3. DMA BBB Platform Designer Block Diagram

This block diagram excludes some internal Pipeline Bridge IP cores.



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

- Modular Scatter-Gather DMA (MSGDMA): This IP core performs memory mapped transfers between source and destination addresses. The MSGDMA transfers 64 bytes per clock cycle. The data must be aligned to 64-byte boundaries. The transfer length must be a multiple of 64 bytes. The MSGDMA supports 50-bit addressing and can transfer up to 16,777,152 bytes per descriptor. In this implementation, the driver limits the transfer size to 1,047,552 bytes per descriptor.
- Address Span Extender: This IP core implements memory transfers that are not aligned on a 64-byte boundary. The host uses it to perform MMIO accesses to FPGA device memory that are not aligned on a 64-byte boundary. The Address Span Extender accesses a 4 kilobyte (4 KB) window into the local device memory. The control port sets the base address of the (4 KB) window. The base address must be aligned to a 4 KB boundary so that the window is aligned to the window size. For example, to access FPGA memory address 0xF340, set the window address to 0xF000 and then access offset 0x0340 within the address span extender data window.
- BBB ID: This component stores the 64-bit `Device Feature Header(DFH)` and the UUID. The BBB_ID_L register stores the lower 32 bits of the BBB ID. The BBB_ID_H register stores the upper 32 bits of the BBB ID. A software driver scans the BBB ID to identify the functionality of this DMA subsystem.
- Magic Number ROM: This IP core contains a single, read-only 64-byte value. The DMA uses this value to create a write fence in host memory. This ROM is only visible to the MSGDMA. The host cannot access it.
- Pipeline Bridge: The Pipeline Bridge inserts pipeline stages to improve the system F_{MAX} at the expense of latency.

3. Register Map and Address Spaces

The DMA AFU supports two memory views: The DMA view and the host view.

The DMA view supports a 50-bit address space. The lower half of the DMA view maps to the FPGA device memory and the magic number ROM inside the DMA BBB. The upper half of the DMA view maps to the host memory which is split into a direct access and a write fence region. The direct access and memory write fence regions overlaps into the same 48-bit host physical address space.

The host view includes all the registers accessible through MMIO accesses such as the DFH tables, and the control/status registers of the various IP cores used inside the DMA AFU. The host view includes an indirect mapping mechanism to the device memory so that the host can access the FPGA device memory using a 4KB windowing mechanism built into the DMA BBB.

The MMIO registers in the DMA BBB and AFU support 32- and 64- bit access. The DMA AFU does not support 512-bit MMIO accesses. Accesses to the MSGDMA registers inside the DMA BBB must be 32 bits.

3.1. DMA AFU Register Map

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

Table 4. DMA AFU Memory Map

Byte Address Offsets	Name	Span in Bytes	Description
0x0_0000	`AFU DFH`	8	Refer to Table 5 on page 13 for the bit fields.
0x0_0008	`AFU ID_L`	8	Set to 0x9081F88B8F655CAA for the DMA AFU.
0x0_0010	`AFU ID_H`	8	Set to 0x331DB30C988541EA for the DMA AFU.
0x0_2000	`MPF DFH`	240	Specifies IDs, feature list, and `control` and `status` registers. The MPF decodes this information. This information is not available inside the DMA Platform Designer system.
0x2_0000	`DMA BBB`	8192	The DMA BBB memory map. Refer to Table 6 on page 14 for the register offsets.

Table 5. DMA AFU DFH Encoding

Bit Field	Description
[11:0]	Feature ID. Set to 0.
[15:12]	AFU major revision number. Set to 0.

continued...

Bit Field	Description
[39:16]	Next DFH byte offset/DFH region size. Set to 8192.
[40]	End of DFH list. When set, the DFH is at the end of the list. The default value is 0.
[47:41]	Reserved.
[51:48]	AFU minor revision number. Set to 0.
[59:52]	Reserved.
[63:60]	Feature type. Set to 1 (AFU).

Table 6. DMA BBB Memory Map

Add the byte addresses below to the BBB DMA base address, 0x2_0000.

Byte Address Offsets	Name	Span in Bytes	Description
0x0000	`BBB DFH`	8	Refer to Table 6 on page 14 for the bit fields.
0x0008	`BBB ID_L`	8	Set to 0xA9149A35BACE01EA for the DMA BBB.
0x0010	`BBB ID_H`	8	Set to 0xEF82DEF7F6EC40FC for the DMA BBB.
0x0040	`MSGDMA CSR`	32	Controls the DMA.
0x0060	`MSGDMA Descriptor`	32	Receives DMA descriptors.
0x0200	`Address Span Extender Control`	8	Moves the address window that the data port accesses.
0x1000	`Address Span Extender Data`	4096	Maps a 4 KB window to a local device memory.

Table 7. DMA BBB DFH Encoding

Bit Field	Description
[11:0]	Feature ID. Set to 0
[15:12]	AFU major revision number. Set to 0.
[39:16]	Next DFH byte offset / DFH region size. Set to 8192.
[40]	End of DFH list. When set, the DFH is at the end of the list. The default value is 0.
[47:41]	Reserved.
[51:48]	AFU minor revision number. Set to 0
[59:52]	Reserved.
[63:60]	Feature type. Set to 2 (BBB).

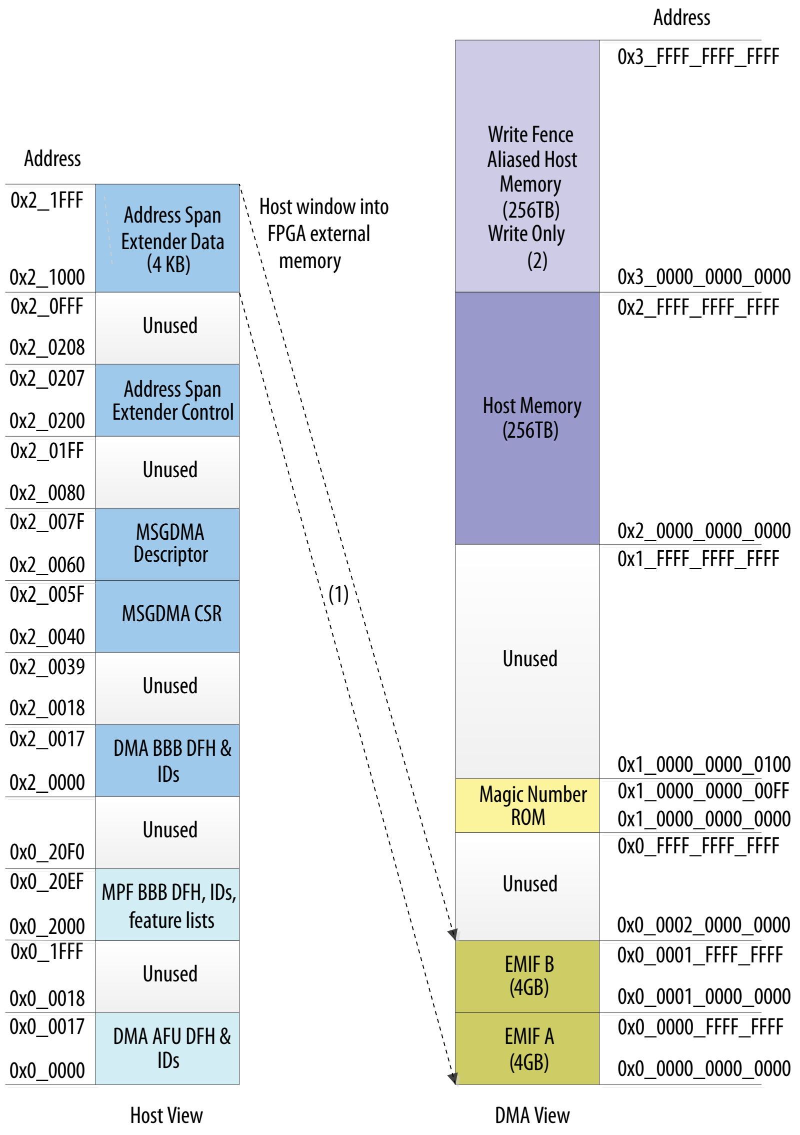
3.1.1. DMA AFU Address Space

The host can access registers listed in the Table 4 on page 13 and the Table 6 on page 14. Host accesses to FPGA local memory must use the Address Span Extender IP core included in the DMA BBB subsystem.

The MSGDMA in the DMA BBB subsystem has access to the full 50-bit address space. The lower have of this address space includes the local memories and the Magic Number ROM. The upper half of this address space includes host memory.

The following figure shows the host and MSGDMA views of memory.

Figure 4. The DMA AFU and Host Views of Memory



(1) The Address Span Extender can only access the EMIF A and EMIF B address spaces.
(2) The write fence aliased host memory, addresses 0x3_0000_0000_0000-0x3_FFFF_FFFF_FFFF, aliases to the host memory spanning 0x2_0000_0000_0000-0x2_FFFF_FFFF_FFFF. The write fence aliased host memory span is write only. Writes to the write fence aliased host memory cause a write fence to be issued followed by the write data accompanying it. This address space should only be written to infrequently to send write fences to synchronize with the host. Reads to this address space are undefined.

4. Software Programming Model

The DMA AFU includes a software driver that you can use in your own host application. The `fpga_dma.c` and `fgpa_dma.h` files located in the `<installation_path>/hw/samples/dma_afu/sw` directory implement the software driver.
This driver supports the following functions:

API	Description
`fpgaDMAOpen()`	Opens a handle to the DMA BBB.
`fpgaDMATransferSync()`	Transfers data from a source location to a destination location. The source and destination can be located in host or device memory.
`fpgaDMATransferClose()`	Closes DMA BBB handle previously allocated.

4.1. Software APIs

4.1.1. fpgaDmaOpen()

`fpgaDmaOpen()` scans the device feature chain to locate the DMA BBB and then creates a handle for the DMA BBB.

Prototype	fpga_result fpgaDmaOpen(fpga_handle fpga, fpga_dma_handle *dma)	
Arguments	`fpga`	Input containing fpga object handle from `fpgaOpen()`.
	`dma`	Output containing handle to the DMA BBB.
Returns	FPGA_OK on success; otherwise an error return code.	

4.1.2. fpgaDmaTransferSync()

`fpgaDmaTransferSync()` accepts a handle to an opened DMA and performs a transfer of data from the source address to the destination address. This function is a blocking call and returns when the DMA transfer completes.

Prototype	fpga_result fpgaDmaTransferSync(fpga_dma_handle dma, uint64_t dst, uint64_t src, size_t count, fpga_dma_transfer_t type)	
Arguments	`dma`	dmaInput specifying DMA handle obtained from `fpgaDmaOpen()`.
	`dst`	Input specifying the destination byte address of the transfer. To maximize performance, make `dst` a multiple of 64 bytes.
	`src`	Input specifying the source byte address of the transfer. To maximize performance, make `src` a multiple of 64 bytes.

continued...

Prototype	fpga_result fpgaDmaTransferSync(fpga_dma_handle dma, uint64_t dst, uint64_t src, size_t count, fpga_dma_transfer_t type)	
	`count`	Input specifying the length of the transfer in bytes. To maximize performance, make `count` a multiple of 64 bytes.
	`type`	Input specifying the type of transfer. `type` has the following valid values: `HOST_TO_FPGA_MM`, `FPGA_TO_HOST_MM`, or `FPGA_TO_FPGA_MM`.
Returns	FPGA_OK on success; otherwise error return code.	

4.1.3. fpgaDmaClose()

`fpgaDmaClose()` closes the previously allocated DMA BBB handle.

Prototype	fpga_result fpgaDmaClose(fpga_Dma_handle dma)	
Parameters	`dma`	Input containing DMA handle obtained from `fpgaDmaOpen()`.
Returns	FPGA_OK on success; otherwise error return code.	

5. Running DMA 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 27 section for details on how to set the hugepages on both Red Hat Enterprise Linux (RHEL) and Ubuntu systems.

Complete the following steps to download the DMA AF bitstream and build and run the example software:

1. Change to the DMA application and driver directory:

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

2. Build the driver and application:

   ```
   $ make
   ```

3. Download the DMA AFU bitstream:

   ```
   sudo fpgasupdate $OPAE_PLATFORM_ROOT/hw/samples/dma_afu \
   /bin/dma_afu_unsigned.gbs
   ```

4. Execute the host application:

   ```
   ./fpga_dma_test 0
   ```

   The DMA software takes a few minute to populate test buffers and verify the results. The software prints the following messages during a successful run:

   ```
   Running test in HW mode
   Buffer Verification Success!
   Buffer Verification Success!
   Running DDR sweep test
   Buffer pointer = 0x7f6edd1fc000, size = 0x100000000 (0x7f6edd1fc000 through
   0x7f6fdd1fc000)
   Allocated test buffer
   Fill test buffer
   DDR Sweep Host to FPGA
   Measured bandwidth = 6660.504131 Megabytes/sec
   Clear buffer
   DDR Sweep FPGA to Host
   Measured bandwidth = 6840.013692 Megabytes/sec
   Verifying buffer..
   Buffer Verification Success!
   ```

   Note: The test application prints bandwidth results for different transfer sizes, source and destination addresses.

Related Information

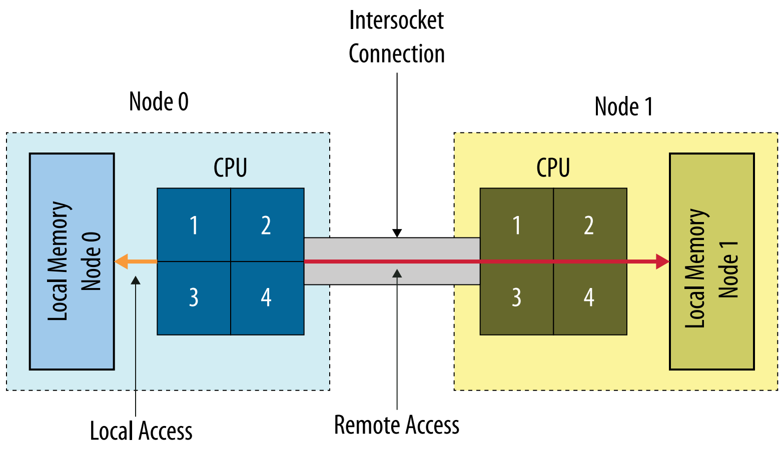
Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA

5.1. Optimization for Improved DMA Performance

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

A typical NUMA configuration is shown in the diagram below. The local access 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 5. 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 DMA AFU sample directory:

    ```
    cd $OPAE_PLATFORM_ROOT/hw/samples/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/dma_afu/build_synth/dma_afu.gbs`.

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

    ```
    PACSign PR -t UPDATE -H openssl_manager -i dma_afu.gbs -o
    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 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 Simulation Environment 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 DMA AFU:

1. Change to the DMA AFU sample directory:

```
cd $OPAE_PLATFORM_ROOT/hw/samples/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=/Tools/ias/inteldevstack/
a10_gx_pac_ias_1_2_1_pv/hw/samples/dma_afu/build_ase_dir/work
[SIM]  tcsh/csh | setenv ASE_WORKDIR /Tools/ias/inteldevstack/
a10_gx_pac_ias_1_2_1_pv/hw/samples/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 DMA AFU software in the simulation environment:

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

```
cd $OPAE_PLATFORM_ROOT/hw/samples/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=/Tools/ias/inteldevstack/
a10_gx_pac_ias_1_2_1_pv/hw/samples/dma_afu/build_ase_dir/work
[SIM]  tcsh/csh | setenv ASE_WORKDIR /Tools/ias/inteldevstack/
a10_gx_pac_ias_1_2_1_pv/hw/samples/dma_afu/build_ase_dir/work
```

4. Compile the software:

```
make USE_ASE=1
```

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

```
./fpga_dma_test
```

Related Information

Intel Accelerator Functional Unit Simulation Environment Quick Start User Guide

8. DMA Accelerator Functional Unit User Guide Archives

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

9. Document Revision History for the 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 following new sections: — *Compiling the Accelerator Function (AF)* — *Simulating the AFU Example* — *Enabling Hugepages* • Modified the path of the Platform Designer system that implements the DMA AFU and the DMA BBB block in section *The DMA AFU Hardware Components*. • Modified steps to run the DMA AFU in section *Running DMA AFU Example*.
2018.12.04	1.2 (supported with Intel Quartus Prime Pro Edition 17.1.1)	• Added new section *Optimization for Improved DMA Performance.* • Added new path for a Platform Designer system in *The DMA AFU Hardware Components* section. • Modified command in *Running DMA AFU Example* section.
2018.08.15	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)	• Corrected the broken link in *DMA Test System*. • Minor edits in *Running DMA AFU Example*.
2018.08.06	1.1 (supported with Intel Quartus Prime Pro Edition 17.1.1) and 1.0 (supported with	• Modified *DMA AFU Block Diagram* to show that MMIO Decoder Logic is used in place of Asynchronous Shim BBB for CCI-P transactions. • Modified the environment variable from `$DCP_LOC` to `$OPAE_PLATFORM_ROOT` in the *Running DMA AFU Example* section.

continued...

Document Version	Intel Acceleration Stack Version	Changes
	Intel Quartus Prime Pro Edition 17.0.0)	
2018.04.11	1.0 (supported with Intel Quartus Prime Pro Edition 17.0.0)	• Updated Figure: *DMA AFU Hardware Block Diagram* to include the DMA BBB. • Updated information in — The DMA AFU Hardware Components — Register Map and Address Spaces — Running DMA AFU Example • Corrected the title of the reference document from the *Altera Acceleration Stack for Intel Xeon CPU with FPGAs Getting Started Guide* to *Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA* in *The DMA AFU Software Package* section. • Editorial modifications.
2017.12.22	1.0 Beta (supported with Intel Quartus Prime Pro Edition 17.0.0)	Initial release.

A. Enabling Hugepages

This section covers information about how to enable 2 MB 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 2 MB Hugepages

If you have already boot your system into the operating system, you can enable twenty 2 MB hugepages by running the following command:

```
sudo sh -c "echo 20 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages"
```

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

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

Persistently Enabling 2 MB Hugepages

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

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

To revert your system back to its default boot settings, follow the instructions from section *Persistently Enabling 2 MB 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.