Intel® High Level Synthesis Accelerator Functional Unit Design Example User Guide

Updated for Intel® Quartus® Prime Design Suite: **19.1**

Subscribe
Send Feedback

UG-20246 | 2019.07.19
Latest document on the web: PDF | HTML

Contents

1. About the HLS AFU Design Example

The Intel High Level Synthesis (HLS) Accelerator Functional Unit (AFU) design example shows how to create AFUs for the Intel® Acceleration Stack for Intel Xeon® CPU with FPGAs with with the Intel HLS Compiler

Before continuing, you should be familiar with the fundamentals of both the Intel HLS Compiler and the Acceleration Stack.

This design example transfers data between a host program and a simple AFU generated with the Intel HLS Compiler. The AFU is a vector reduction design named `hls_afu`, and the design example uses `ac_int` and `float` datatypes.

To obtain the HLS AFU Design Example code, contact an Intel Sales agent.

You can use this code as a model to create your own HLS AFUs if your AFUs use the same interfaces as the example design. Also, you might be able to convert your HLS application into an AFU by adding the required interfaces to the hardware design.

Related Information

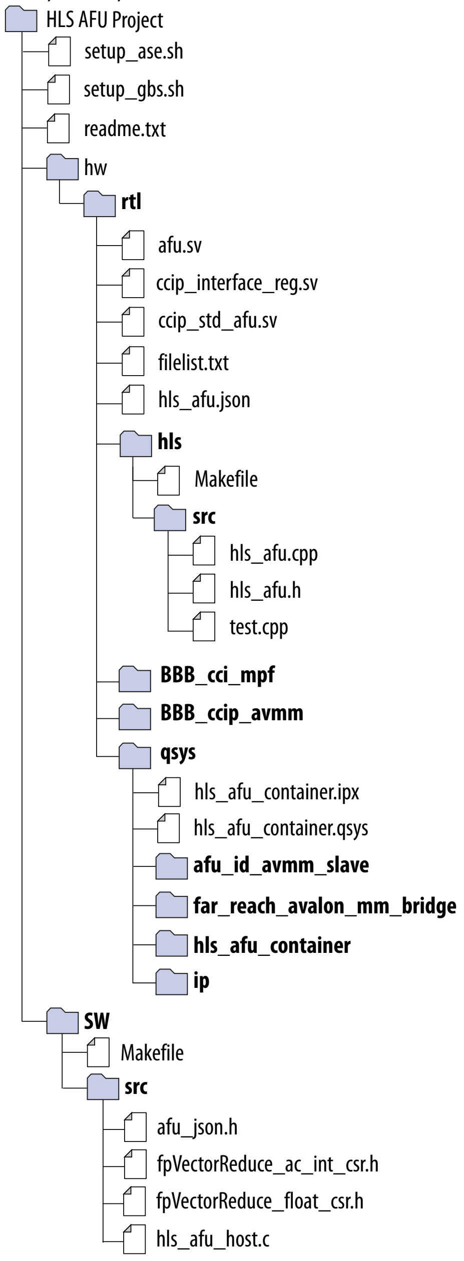
- Intel Acceleration Stack Knowledge Center
- Intel High Level Synthesis Compiler Getting Started Guide
- Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA

1.1. HLS AFU Design Example Files

The following figure shows the files in the HLS AFU design example that you might need to edit or otherwise work with either to run the design example or to replace the HLS component in the design example with your own component.

Figure 1. Directory Structure and Files

This figure shows only the files that you may edit.



1.2. Acronyms in the HLS AFU Design Example User Guide

Acronym	Term	Definition
AF	Accelerator function	Compiled hardware accelerator image implemented in FPGA logic that accelerates an application. This image is unique for a specific FPGA board.
AFU	Accelerator functional unit	Hardware accelerator implemented in FPGA logic that offloads a computational operation for an application from a processor to improve performance. This uncompiled code is platform-agnostic.
API	Application programming interface	An API is a set of conventions defined by a programmer for accessing re-usable code, such as in a library.
ASE	AFU simulation environment	Cosimulation environment that allows you to use the same host application and AF in a simulation environment. ASE is part of the Intel acceleration stack for FPGAs.
BBB	Basic building block	Basic building blocks (BBB) for Intel FPGAs is a suite of application building blocks and shims for transforming the CCI-P.
CCI-P	Core cache interface	CCI-P is the standard interface that AFUs use to communicate with the host system and Xeon processor.
DFH	Device feature header	Creates a linked list of feature headers to provide an extensible way of adding features.
FIM Bitstream	FPGA interface manager bitstream	An unchanging region in the FPGA that enables AFs to be swapped in and out. The FIM bitstream contains interfacing logic that allows the AF to communicate with the host and onboard peripherals.
FIU	FPGA interface unit	A platform interface layer that bridges platform interfaces like PCIe* and UPI with AFU-side interfaces like CCI-P.
HLS	High-level synthesis	A compiler that translates C++ source code into RTL for use in FPGA designs
MPF	Memory properties factory	A BBB that AFUs can use to provide CCI-P traffic shaping operations for transactions with the FIU.
OPAE	Open programmable acceleration engine	The OPAE is a software framework for managing and accessing AFs. For more details, refer to the *Open Programmable Acceleration Engine C API Programming Guide*).
Intel PAC	Intel Programmable Acceleration Card	The PCIe accelerator card with an Intel Arria 10 or Stratix 10 FPGA contains a FIM that connects to an Intel Xeon processor over PCIe bus.
RTL	Register transfer level	Logic-level representation of hardware to implement in an FPGA. You can write this logic using an HDL such as Verilog HDL and, generate it using a tool like the Intel HLS compiler.
UUID	Universally unique identifier	A 128-bit number that identifies information in computer systems.

Related Information

Open Programmable Acceleration Engine C API Programming Guide

2. Building the HLS AFU Design Example

You need to run various commands to build and test the HLS AFU design. If you want to quickly verify your set-up, you can follow the abbreviated instructions in the example design `/hls_afu/README.txt` file.

The design example shows how you add an Intel HLS Compiler component to an Acceleration Stack AFU. The HLS AFU design example targets the Intel Programmable Acceleration Card with Intel Arria® 10 GX FPGA. It includes:

- An AFU that runs on an FPGA. You can compile the platform-agnostic AFU into a platform-specific accelerator function (AF) that executes on the FPGA. This function interacts with memory on the host computer (host memory) through the core cache interface (CCI-P).
- Host software that runs on an Intel Xeon CPU. This software provides your application logic and is responsible for allocating memory to be shared with the AF, sending data to the FPGA, and collecting the results when the AF finishes executing.

Building the design examples includes the following steps:

1. Compile and simulate the HLS component with the Intel HLS Compiler to verify functional correctness.
2. Add the HLS component to an AFU by creating a Platform Designer container.
3. Cosimulate the design with the Intel AFU Simulation Environment (ASE) to confirm the AFU functionality after adding the HLS component:
   a. Generate the ASE testbench.
   b. Run the ASE testbench.
4. Compile the AF bitstream.
5. Load the AF bitstream on hardware and run the host application.

2.1. HLS AFU Design Example Software Requirements

Ensure that you configure your system for building Intel Acceleration Stack designs as described in the *Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA*

Before you use the design example, install the following software:

- Intel Acceleration Stack for Development Version 1.2 or later (and associated prerequisites)

  The Acceleration Stack includes Intel Quartus Prime Pro Edition Version 17.1.1
- Intel High Level Synthesis (HLS) Compiler Version 19.1 or later to compile your HLS code

  Get the Intel High Level Synthesis Compiler from the **Additional Software** tab of the Intel Quartus Prime Pro Edition Version 19.1 download page at the **Download Center for FPGAs**.

  For instructions about how to integrate Intel HLS Compiler Pro Edition Version 19.1 into your Intel Quartus Prime Pro Edition Version 17.1.1 installation, review the *Intel High Level Synthesis Compiler Getting Started Guide*.
- (Optional) 64-bit Mentor ModelSim* SE* Simulator (Version 10.5c) or 64-bit Mentor Questa* Advanced Simulator (Version 10.5c)

  These simulators are required if you want to simulate the AFU.

  If you want to simulate only the HLS component, you can use the 32-bit ModelSim - Intel FPGA Edition software.

Related Information

- Troubleshooting HLS AFU Designs on page 36
- Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA
- Intel High Level Synthesis Compiler Getting Started Guide
- Download Center for FPGAs - Quartus Prime Pro Edition V19.1

2.2. Compiling and Simulating the HLS Component with the i++ Command

As with other HLS design examples, you can compile this example design using the included makefile. This makefile uses similar conventions to the HLS design examples. Ensure your development environment includes the Intel HLS Compiler Pro Edition Version 19.1 or later.

Note: The HLS code might not compile if you are using the Intel HLS Compiler Pro Edition Version 19.1 in an environment that does not have the correct version of the GCC libraries. To ensure that you have the correct libraries, review the instructions in the *Intel High Level Synthesis Compiler Getting Started Guide*.

1. Initialize your current session so that you can run the Intel HLS Compiler. In your terminal session, change directories to the `hls` directory in your Acceleration Stack installation directory.

   For example:

   ```
   $ cd /home/<username>/inteldevstack/intelFPGA_pro/hls
   ```
2. Run the following command from the `hls` directory to set the environment variables for the i++ command in the current terminal session:

   ```
   $ source init_hls.sh
   ```

The environment initialization script shows the environment variables that it sets.

3. Navigate to the HLS soruce code.

   The HLS source code is in `<design location>/hls_afu/hw/rtl/hls/`.

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

```
$ make test-x86-64
$ ./test-x86-64
```

   The `test-x86-64` command gives you the following output:

```
i++ src/hls_afu.cpp src/test.cpp   --fp-relaxed -ghdl  -march=x86-64 -o
test-x86-64
+---------------------------------------------+
| Run ./test-x86-64 <n> to execute the test. |
| <n> is 0, 1, or 2 depending on desired     |
| test behavior:                             |
|         <n> | effect                       |
|       ------+--------------------          |
|          0  | test both (default)          |
|          1  | test ac_int only             |
|          2  | test float only              |
+---------------------------------------------+
Control which component gets tested by passing an integer!
arg   | effect
------+--------------------
   0  | test both (default)
   1  | test ac_int only
   2  | test float only
test AC_INT version and FLOAT version

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

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

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

```
$ make test-fpga
$ ./test-fpga
```

   The `test-fpga` command gives you the following output:

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

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

FLOATING-POINT COMPONENT - 81 ELEMENTS
fp_inc:
```

```
PASS
OVERALL:
PASSED
```

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

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

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

Related Information

- Intel High Level Synthesis Compiler User Guide
- Intel High Level Synthesis Compiler Getting Started Guide

2.3. Generating a Platform Designer Container for the HLS Component

Use Platform Designer to integrate the HLS component into an AFU with the predesigned hardware interfaces available in the Acceleration Stack, and verify that all sources are linked correctly.

1. You might need to set the environment variable values required by the Acceleration Stack. To set the variables, run the following command:

```
$ source /home/<username>/inteldevstack/init_env.sh
```

2. Navigate to the `qsys` folder and open the system using Platform Designer.

```
$ qsys-edit hls_afu_container.qsys
```

   You can use a `.ipx` file to point to your IP files. In the design example, the `hls_afu_container.ipx` file points to where the HLS compiler-generated RTL is expected to be.

   If you have other files in other locations that you need to include in your Platform Designer system, update the `filelist.txt` file with the paths to those files.

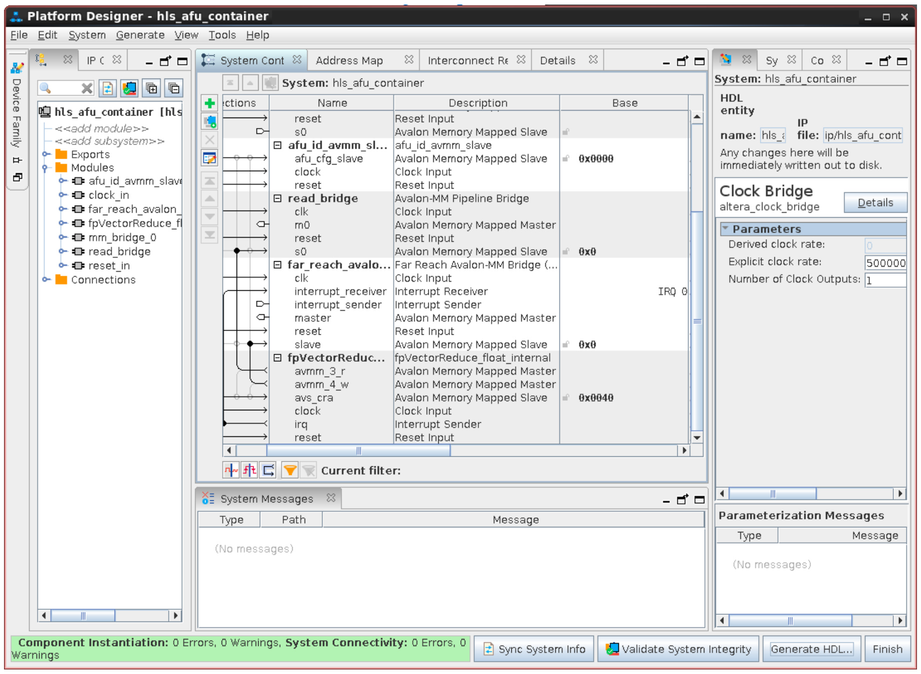
3. In the **Open System** dialog box, select **None** for the **Quartus project** dropdown.

   Ensure the **Device part** is **10AX115N2F40E2LG**, which matches the FPGA on the Intel PAC with Intel Arria 10 GX device.

   If you want to modify the Platform Designer system, associate it to a temporary Intel Quartus Prime Pro Edition project.

4. Click **Open**.

Figure 2. Open System Dialog box



Platform Designer - hls_afu_container

File Edit System Generate View Tools Help

hls_afu_container [hls
<<add module>>
<<add subsystem>>
Exports
Modules
afu_id_avmm_slave
clock_in
far_reach_avalon_
fpVectorReduce_fl
mm_bridge_0
read_bridge
reset_in
Connections

System: hls_afu_container

Name	Description	Base
reset	Reset Input	
s0	Avalon Memory Mapped Slave	
afu_id_avmm_sl...	afu_id_avmm_slave	
afu_cfg_slave	Avalon Memory Mapped Slave	0x0000
clock	Clock Input	
reset	Reset Input	
read_bridge	Avalon-MM Pipeline Bridge	
clk	Clock Input	
m0	Avalon Memory Mapped Master	
reset	Reset Input	
s0	Avalon Memory Mapped Slave	0x0
far_reach_avalo...	Far Reach Avalon-MM Bridge (...	
clk	Clock Input	
interrupt_receiver	Interrupt Receiver	IRQ 0
interrupt_sender	Interrupt Sender	
master	Avalon Memory Mapped Master	
reset	Reset Input	
slave	Avalon Memory Mapped Slave	0x0
fpVectorReduc...	fpVectorReduce_float_internal	
avmm_3_r	Avalon Memory Mapped Master	
avmm_4_w	Avalon Memory Mapped Master	
avs_cra	Avalon Memory Mapped Slave	0x0040
clock	Clock Input	
irq	Interrupt Sender	
reset	Reset Input	

Current filter:

System Messages

Type	Path	Message
(No messages)		

System: hls_afu_container
HDL entity name: hls_ IP file: ip/hls_afu_cont
Any changes here will be immediately written out to disk.

Clock Bridge
altera_clock_bridge

Parameters
Derived clock rate: 0
Explicit clock rate: 500000
Number of Clock Outputs: 1

Parameterization Messages

Type	Message
(No messages)	

Component Instantiation: 0 Errors, 0 Warnings, System Connectivity: 0 Errors, 0 Warnings

5. To reload the system and ensure that all search paths are correct, click **Validate System Integrity** at the bottom of the **Platform Designer** window.
6. After **Validate System Integrity** successfully completes, click **Close**.
7. Exit Platform Designer.
8. Save the Platform Designer system.
9. Click **Yes** to generate the HDL. Ignore any warnings.

2.4. Generating the ASE Testbench

After integrating the HLS component into an AFU, you might want to cosimulate the AFU in the Intel AFU Simulation Environment (ASE). Use this cosimulation to quickly confirm the functionality of your HLS component within the AFU.

If you want to skip co-simulating the AFU in the ASE and run the design on hardware, go to Compiling the AF Bitstream on page 14.

Before you co-simulate your AFU in the ASE, ensure that your ASE is configured according to the instructions in the *Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) Quick Start User Guide*.

You must have a valid license for one of the supported simulators to use the ASE.

Ensure that your simulation software environment variables are set correctly:

- For Mentor ModelSim SE or Mentor Questa Advanced Simulator:

```
$ export MTI_HOME=<path to ModelSim/Questa installation directory>
$ export PATH=$MTI_HOME/linux_x86_64/:$MTI_HOME/bin/:$PATH
```

Refer to `hls_afu/setup_ase.sh` to automate these steps.

To generate the ASE testbench:

1. Navigate to the root of your project (the `hls_afu` directory) and run this command:

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

2. Navigate to `build_ase_dir` directory.
3. Open `Makefile` and add the `twentynm` libraries to the gate level library as follows:

```
# Gate level libraries to add to simulation
GLS_VERILOG_OPT = $(QUARTUS_HOME)/eda/sim_lib/altera_primitives.v
GLS_VERILOG_OPT+= $(QUARTUS_HOME)/eda/sim_lib/220model.v
GLS_VERILOG_OPT+= $(QUARTUS_HOME)/eda/sim_lib/sgate.v
GLS_VERILOG_OPT+= $(QUARTUS_HOME)/eda/sim_lib/altera_mf.v
GLS_VERILOG_OPT+= $(QUARTUS_HOME)/eda/sim_lib/altera_lnsim.sv
GLS_VERILOG_OPT+= $(QUARTUS_HOME)/eda/sim_lib/twentynm_atoms.v
GLS_VERILOG_OPT+= $(QUARTUS_HOME)/eda/sim_lib/mentor/twentynm_atoms_ncrypt.v
```

   The `twentynm` libraries are highlighted in bold text.

4. Run the following commands:

```
$ make
$ make sim
```

   When you see the following message, you can build and run the host.

   Take note of the `export` command in your `make sim` command output. You need this command to set your environment to run the host program. The `export` command is highlighted in the following output example:

```
#   [SIM]  ASE lock file .ase_ready.pid written in work directory
#   [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=/home/john/hls_afu/build_ase_dir/
work
#   [SIM]  tcsh/csh | setenv ASE_WORKDIR /home/john/hls_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...
```

   Leave this terminal window open. This terminal window is referred to in later steps as the ASE terminal window.

Related Information

Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) Quick Start User Guide

2.5. Running the ASE Testbench

Before you run the testbench, ensure that you have the `export` command from the output of the `make sim` command when you generated the ASE testbench. Ensure that the terminal window where you ran the `make sim` command is still open.

The terminal window where you ran the `make sim` command is referred to as the ASE terminal window.

To run the ASE testbench:

1. Open a new terminal window to compile the host application. This window is referred to as the host terminal window.

   Keep the ASE terminal window open so that you can refer to the output in that window.

   Perform the remaining steps in the host terminal window.
2. Navigate to `hls_afu/sw` directory.
3. Export the *ASE_WORKDIR* environment variable using the `export` command from the output of the `make sim` command in the ASE terminal window:

   ```
   export ASE_WORKDIR=<path to work folder>
   ```
4. Run the following command:

   ```
   make USE_ASE=1
   ```
5. Run the host executable with the following command:

   ```
   $ ./hls_afu_host
   ```

When the executable runs successfully, the host application executes in the host terminal window and displays a `Test` PASSED message. All CCI-P transactions execute in the ASE terminal window. You can view the ASE simulation waveforms if you want to debug your AFU. For more details, see the *Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) Quick Start User Guide*.

Figure 3. Host Terminal Window (End of Output)

```
Interrupt enabled = 00000001
  [APP]  MMIO Write     : tid = 0x00a, offset = 0x68, data = 0xc3ca00000
  [APP]  MMIO Write     : tid = 0x00b, offset = 0x70, data = 0xb04400000
  [APP]  MMIO Write     : tid = 0x00c, offset = 0x78, data = 0x40
  [APP]  MMIO Write     : tid = 0x00d, offset = 0x48, data = 0x1
AFU Latency: 4459.48300 milliseconds
Poll success. Return = 1
check output memory:
output memory OK!
  [APP]  MMIO Read      : tid = 0x00e, offset = 0x58
  [APP]  MMIO Read Resp : tid = 0x00e, data = 3
  [APP]  MMIO Write     : tid = 0x00f, offset = 0x58, data = 0x3
  [APP]  MMIO Read      : tid = 0x010, offset = 0x60
  [APP]  MMIO Read Resp : tid = 0x010, data = 4432c000
sum: Expected 715.000000, calculated 715.000000.

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

end of output memory after executing kernel:
    [62] - 22.333334 (0x41b2aaab)
    [63] - 22.666666 (0x41b55555)
    [64] - -6259853398707798016.000000 (0xdeadbeef)
    [65] - 0.000000 (0x0)
Vector size is 64 (256 bytes), so expect memory output at [64] = 0xdeadbeef
Finished Running Test.
  [APP]  Deallocate request index = 3 ...
  [APP]  Deallocating memory /buf1.369227379399493 ...
  [APP]  SUCCESS
  [APP]  Deallocate request index = 2 ...
  [APP]  Deallocating memory /buf0.369227379399493 ...
  [APP]  SUCCESS
  [APP]  Deinitializing simulation session
  [APP]  Closing Watcher threads
  [APP]  Deallocating UMAS
  [APP]  Deallocating memory /umas.369227379399493 ...
  [APP]  SUCCESS
  [APP]  Deallocating MMIO map
  [APP]  Deallocating memory /mmio.369227379399493 ...
  [APP]  SUCCESS
  [APP]  Deallocate all buffers ...
  [APP]      Took 6,302,858,736 nsec
  [APP]  Session ended
Test PASSED
```

Figure 4. ASE Terminal Window (End of Trace)

```
#   [SIM]  Ready for simulation...
#   [SIM]  Press CTRL-C to close simulator...
#   [SIM]  Session requested by PID = 153501
#   [SIM]  Session ID => 356353904181555
#   [SIM]  Event socket server started
#   [SIM]  SIM-C : Creating Socket Server@/tmp/
ase_event_server_356353904181555...
#   [SIM]  SIM-C : Started listening on server /tmp/
ase_event_server_356353904181555
#   [SIM]  0    ADDED       /mmio.356353904181555
#   [SIM]  1    ADDED       /umas.356353904181555
#   [SIM]  2    ADDED       /buf0.356353904181555
#   [SIM]  3    ADDED       /buf1.356353904181555
#   [SIM]  SIM-C : AFU Interrupt event 0
#   [SIM]  Request to deallocate "/buf1.356353904181555" ...
#   [SIM]  3    REMOVED     /buf1.356353904181555
#   [SIM]  Request to deallocate "/buf0.356353904181555" ...
#   [SIM]  2    REMOVED     /buf0.356353904181555
#   [SIM]  Request to deallocate "/umas.356353904181555" ...
#   [SIM]  1    REMOVED     /umas.356353904181555
#   [SIM]  Request to deallocate "/mmio.356353904181555" ...
#   [SIM]  0    REMOVED     /mmio.356353904181555
#   [SIM]  ASE recognized a SW simkill (see ase.cfg)... Simulator will EXIT
#   [SIM]  SIM-C : Exiting event socket server@/tmp/
ase_event_server_356353904181555...
#   [SIM]  Closing message queue and unlinking...
#   [SIM]  Unlinking Shared memory regions....
#   [SIM]  Session code file removed
#   [SIM]  Removing message queues and buffer handles ...
#   [SIM]  Cleaning session files...
#   [SIM]  Simulation generated log files
#   [SIM]          Transactions file       | $ASE_WORKDIR/ccip_transactions.tsv
#   [SIM]          Workspaces info         | $ASE_WORKDIR/workspace_info.log
#   [SIM]          ASE seed                | $ASE_WORKDIR/ase_seed.txt
#   [SIM]
#   [SIM]  Tests run     => 1
#   [SIM]
#   [SIM]  Sending kill command...
#   [SIM]  Simulation kill command received...
#

...


#
# ** Note: $finish    : /nfs/tor/disks/swuser_work_whitepau/OPAE_Samples/
hls_afu_beta3/hls_afu/hls_afu/build_ase_dir/rtl/ccip_emulator.sv(2654)
#    Time: 833760 ns  Iteration: 2  Instance: /ase_top/ccip_emulator
# End time: 17:21:11 on Oct 31,2018, Elapsed time: 0:18:04
# Errors: 3, Warnings: 4680
```

You can safely ignore the errors listed during this run. The errors result mainly from write and read responses happening simultaneously during cosimulation.

Related Information

Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) Quick Start User Guide

2.6. Compiling the AF Bitstream

Compiling a bitstream takes significantly longer than simulating the AFU. The AFU is compiled with Intel Quartus Prime to generate an FPGA bitstream. is generated.

Before you compile the AF bitstream, ensure that you have complete the steps in Generating a Platform Designer Container for the HLS Component on page 9.

Refer to `hls_afu/setup_gbs.sh` to automate these steps.

To compile the AF bitstream:

1. Generate the AF build environment and create the AF (.gbs) image.

   This process takes approximately 30 minutes.

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

When the AF is created successfully, you get the following message:

```
Wrote hls_afu.gbs

===========================================================================
 PR AFU compilation complete
 AFU gbs file is 'hls_afu.gbs'
===========================================================================
```

2.7. Loading AF Bitstream and Running the Host Application

To run the bitstream, ensure that your host system contains an Intel FPGA PAC and that you have Acceleration Stack (including OPAE) installed and configured. For details, see *Intel Acceleration Stack Quick Start Guide for Intel PAC with Intel Arria 10 GX FPGA.*

1. Start a terminal session and navigate to the root of the project (the `hls_afu` directory).
2. Configure your system to use appropriately sized `hugepages`:

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

3. Load the AF into the FPGA:

   ```
   $ sudo fpgaconf hls_afu.gbs
   ```

4. Navigate to the `hls_afu/sw` directory.
5. Build and run the host application (do not specify USE_ASE=1).

   ```
   $ make
   $ sudo ./hls_afu_host
   ```

The expected output is:

```
Using Avalon Slave at offset 0x40
No vector size specified. Default to size 64 floats! run ./hls_afu_host
<vectorsize> to specify a vector size at runtime.
Using test vector of size 64.
Running Test
AFU DFH REG = 1000010000000000
AFU ID LO = 944028430b016f3d
AFU ID HI = 5fa7fd4b867c484c
AFU NEXT = 00000000
```

```
AFU RESERVED = 00000000
end of output memory before executing kernel:
    [62] - -6259853398707798016.000000 (0xdeadbeef)
    [63] - -6259853398707798016.000000 (0xdeadbeef)
    [64] - -6259853398707798016.000000 (0xdeadbeef)
    [65] - 0.000000 (0x0)
Interrupt enabled = 00000000
Interrupt enabled = 00000001
AFU Latency: 0.01600 milliseconds
Poll success. Return = 1
check output memory:
output memory OK!
sum: Expected 715.000000, calculated 715.000000.

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

end of output memory after executing kernel:
    [62] - 22.333334 (0x41b2aaab)
    [63] - 22.666666 (0x41b55555)
    [64] - -6259853398707798016.000000 (0xdeadbeef)
    [65] - 0.000000 (0x0)
Vector size is 64 (256 bytes), so expect memory output at [64] = 0xdeadbeef
Finished Running Test.
Test PASSED
```

Related Information

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

3. Customizing the HLS AFU

You can substitute a different HLS component into the AFU in the HLS AFU design example to customize the accelerator.

To integrate an HLS component into an AFU, the AFU must provide the interfaces that your HLS component needs. You might need to update your host code as well.

At a high-level, to create an AFU from an HLS component, you must generate RTL for your component with the i++ command and then create a new UUID for the new AFU.

For details about how to create a UUID, see the Accelerator Functional Unit (AFU) Developer's Guide.

The steps that follow provide an example of how to replace the HLS component in the design example by replacing the `fpVectorReduce_float` component with the `fpVectorReduce_ac_int` component.

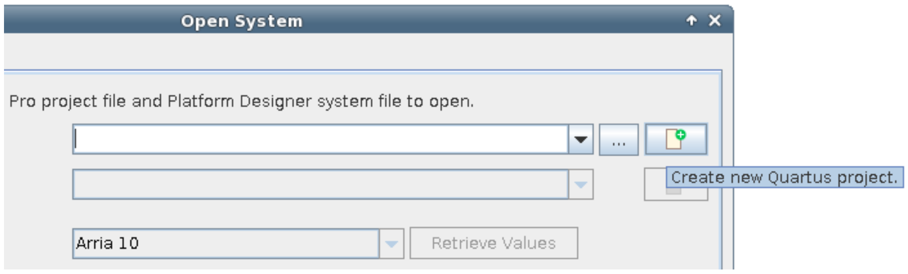
To replace the HLS component in the design example:

1. Navigate to the `qsys` folder and create a new folder there called `quartus_temp`.
2. Open the system with Platform Designer (from Intel Quartus Prime Pro Edition Version 17.1.1, provided with the Acceleration Stack) by running the following command from the `qsys` folder:

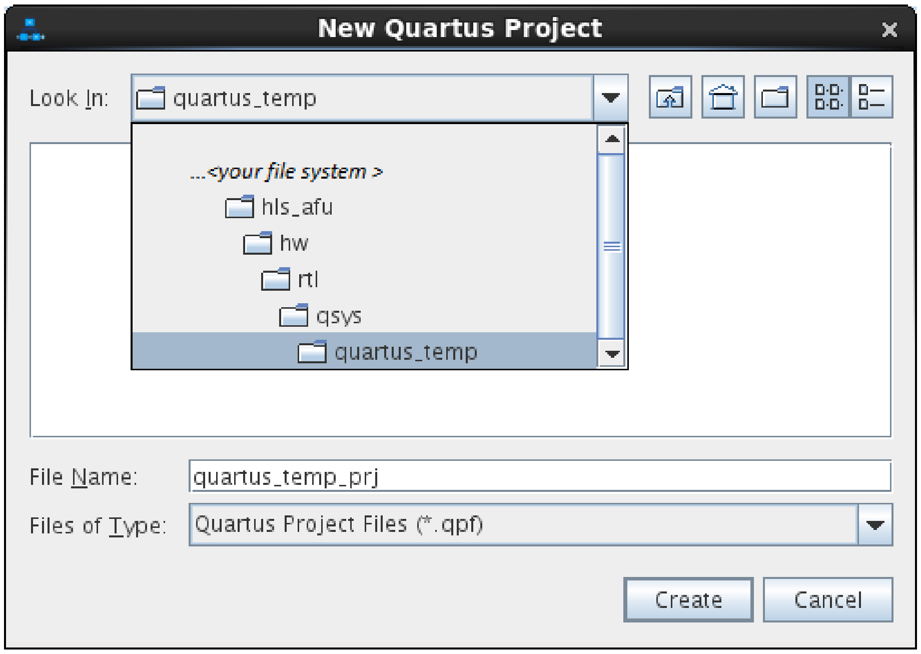
```
$ qsys-edit --search-path=../hls/test-fpga.prj/components/
fpVectorReduce_float,\
../hls/test-fpga.prj/components/fpVectorReduce_ac_int,$
hls_afu_container.qsys
```

3. To edit `hls_afu_container.qsys`, create an Intel Quartus Prime project by clicking **Create New Quartus Project**.

Figure 5. New Quartus Project


Open System
Pro project file and Platform Designer system file to open.
Create new Quartus project.
Arria 10

4. Create a `quartus_temp_prj` project in the `quartus_temp` folder.



New Quartus Project

Look In: quartus_temp

...<your file system >
hls_afu
hw
rtl
qsys
quartus_temp

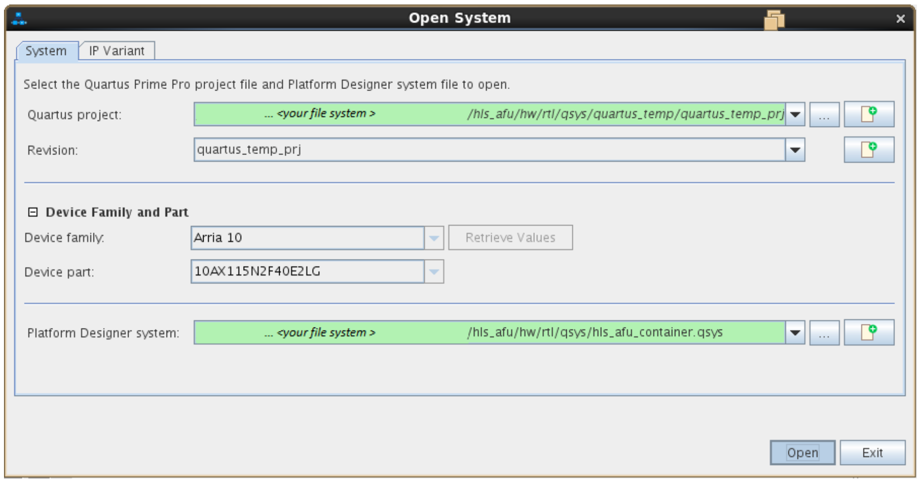
File Name: quartus_temp_prj

Files of Type: Quartus Project Files (*.qpf)

Create | Cancel

5. Click **Open** on the **Open System** window.

Figure 6. Open System



Open System

System | IP Variant

Select the Quartus Prime Pro project file and Platform Designer system file to open.

Quartus project: ... <your file system > /hls_afu/hw/rtl/qsys/quartus_temp/quartus_temp_prj

Revision: quartus_temp_prj

Device Family and Part

Device family: Arria 10 | Retrieve Values

Device part: 10AX115N2F40E2LG

Platform Designer system: ... <your file system > /hls_afu/hw/rtl/qsys/hls_afu_container.qsys

Open | Exit

6. Optional: If you have a custom HLS component that has a new UUID for your component, enter the UUID it in the **afu_id_avmm_slave** component parameters.

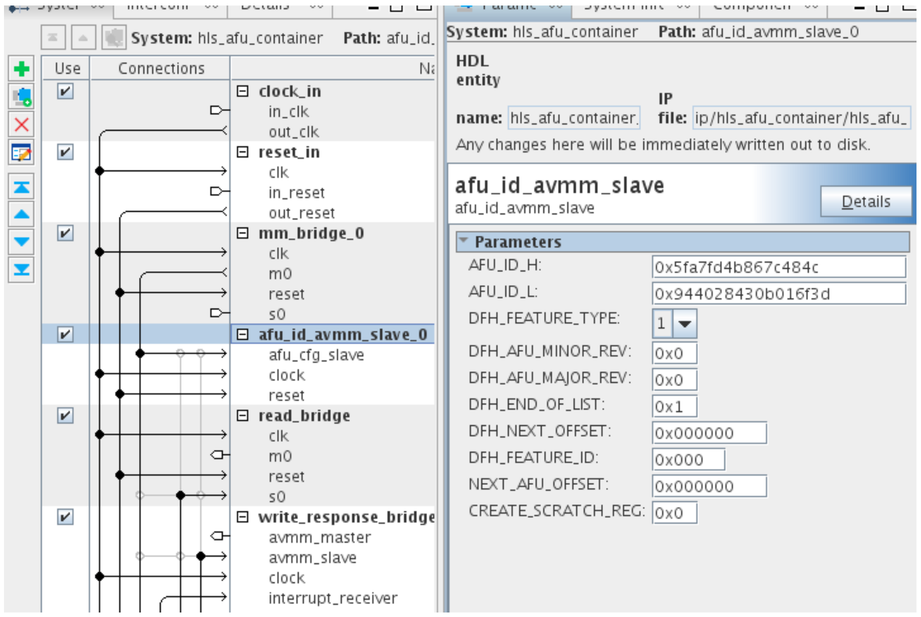
   Enter the UUID as follows:

   a. In the **AFU_ID_H** field, enter the upper 64 bits of the UUID.

   b. In the **AFU_ID_L** field, enter the lower 64 bits of the UUID.

   The host program must also be updated with the new UUID information. For details, see Host Application Description on page 33.

Figure 7. UUID



System: hls_afu_container Path: afu_id_avmm_slave_0

HDL entity name: hls_afu_container

IP file: ip/hls_afu_container/hls_afu_

Any changes here will be immediately written out to disk.

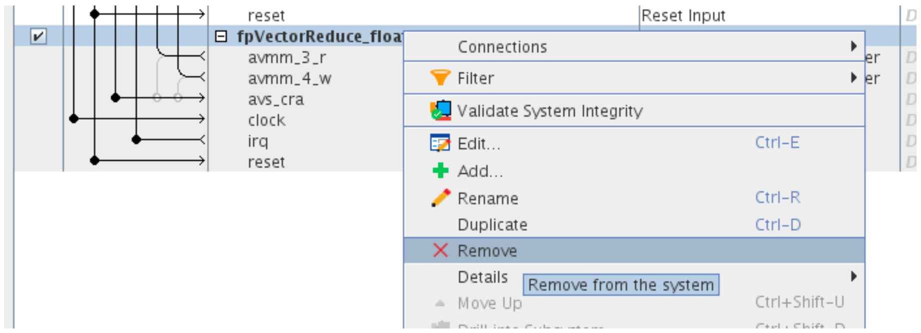
afu_id_avmm_slave
afu_id_avmm_slave

Parameters

AFU_ID_H:	0x5fa7fd4b867c484c
AFU_ID_L:	0x944028430b016f3d
DFH_FEATURE_TYPE:	1
DFH_AFU_MINOR_REV:	0x0
DFH_AFU_MAJOR_REV:	0x0
DFH_END_OF_LIST:	0x1
DFH_NEXT_OFFSET:	0x000000
DFH_FEATURE_ID:	0x000
NEXT_AFU_OFFSET:	0x000000
CREATE_SCRATCH_REG:	0x0

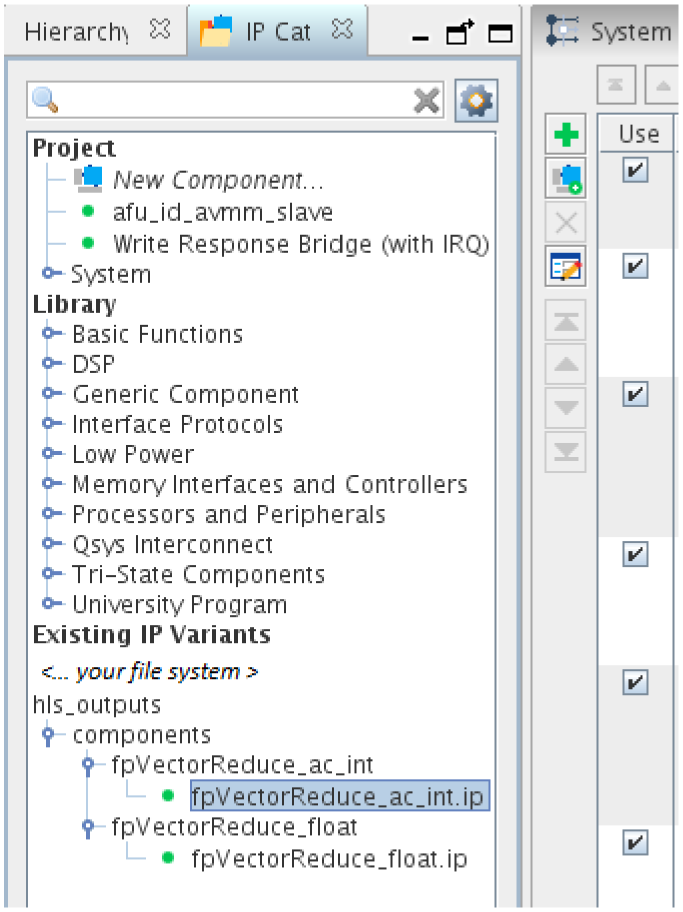
7. Remove the **fpVectorReduce_float** component by right-clicking it and clicking **Remove**.

Figure 8. Remove Component



8. In the **Delete IP Variant** dialog box, click **No** so that Platform Designer does not delete the **fpVectorReduce_float** IP file.
9. Search for the **fpVectorReduce_ac_int** component in **IP Catalog**. Double-click the component to add it to the system.

Figure 9. Add Component

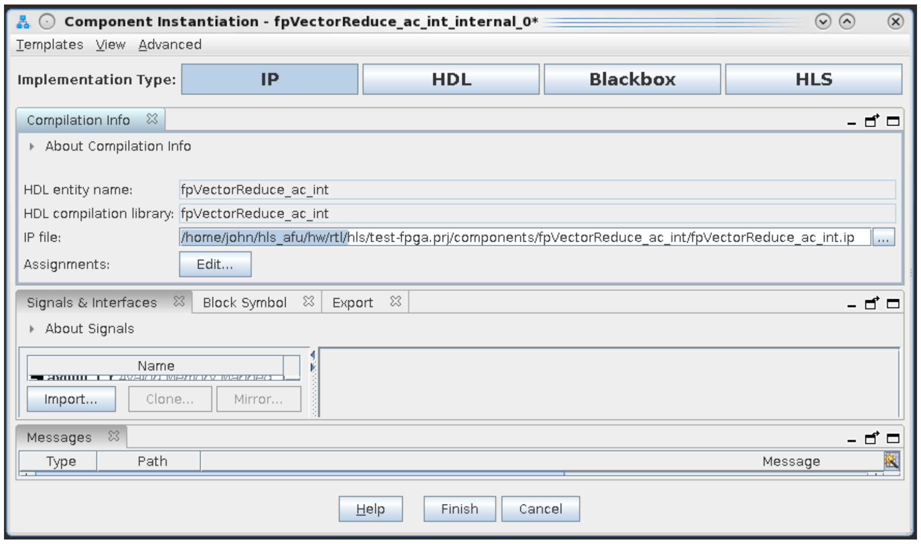


10. In the **Component Instantiation** window, convert the absolute path for the IP file information to a relative path.

    To convert the path to a relative path, edit the location of the IP file information to delete all the text preceding the `hls` folder (including the leading "/" character) and replace it with "`..`" (two periods).

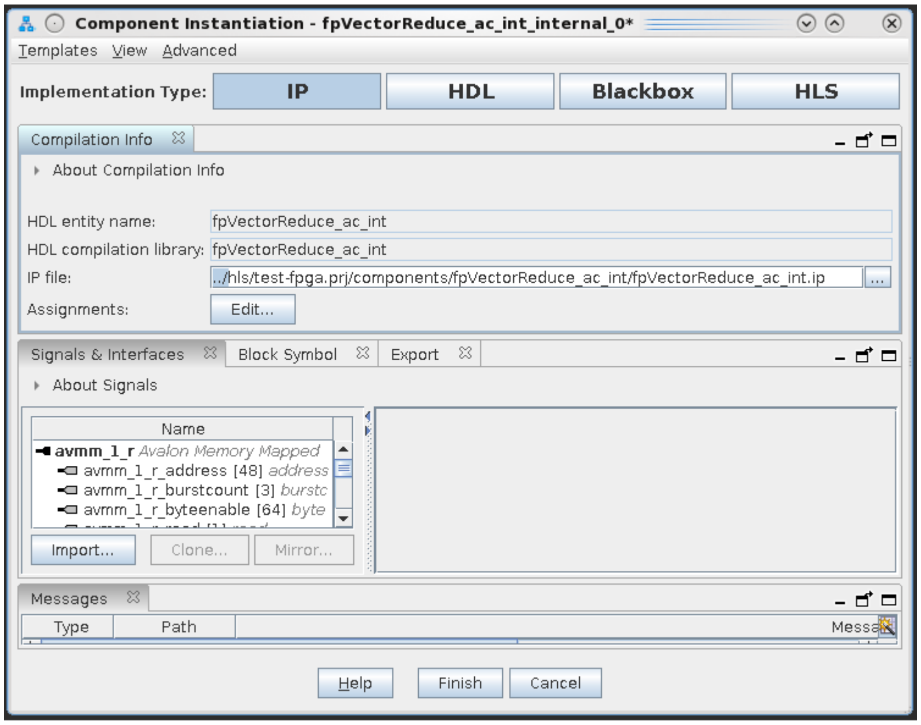
    Having a relative path here gives the `.qsys` file a relative path to your `.ip` file, so you can move your AFU project safely to a different file system.

Figure 10. Absolute Path Before Deleting Text


Component Instantiation - fpVectorReduce_ac_int_internal_0*
Templates View Advanced
Implementation Type: IP HDL Blackbox HLS
Compilation Info
About Compilation Info
Signals & Interfaces Block Symbol Export
About Signals
Name
Import... Clone... Mirror...
Messages
Type Path Message
Help Finish Cancel

HDL entity name: fpVectorReduce_ac_int
HDL compilation library: fpVectorReduce_ac_int
IP file: /home/john/hls_afu/hw/rtl/hls/test-fpga.prj/components/fpVectorReduce_ac_int/fpVectorReduce_ac_int.ip
Assignments: Edit...

Figure 11. Relative Path After Deleting Text

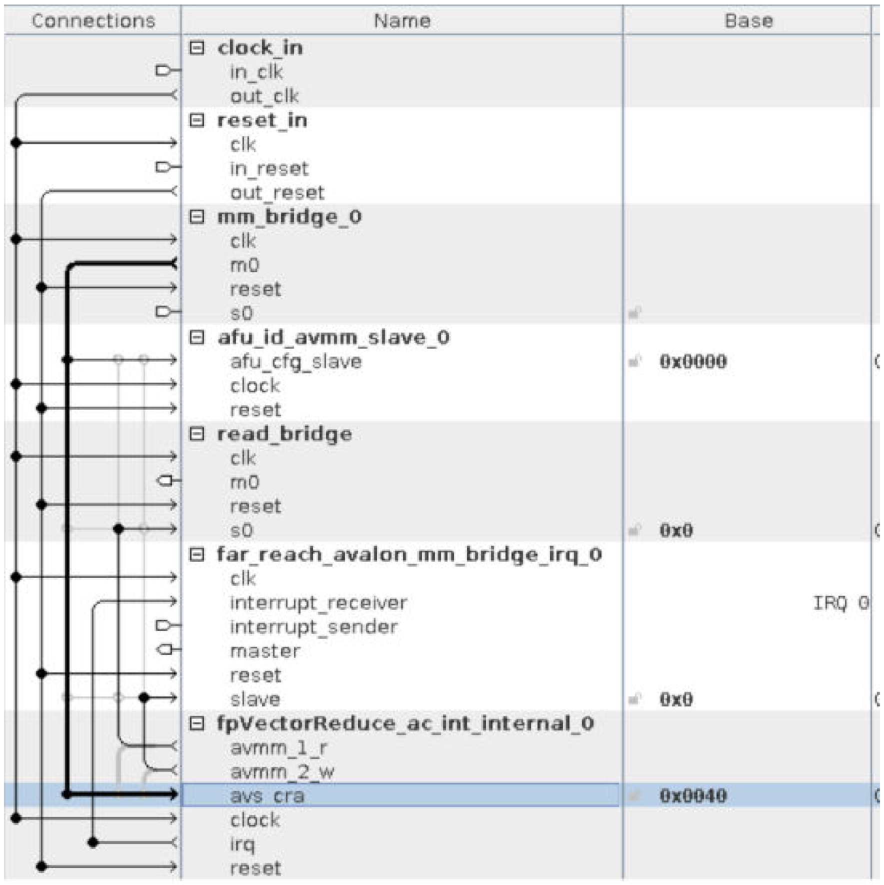

Component Instantiation - fpVectorReduce_ac_int_internal_0*
Templates View Advanced
Implementation Type: IP HDL Blackbox HLS
Compilation Info
About Compilation Info
Signals & Interfaces Block Symbol Export
About Signals
Name
avmm_1_r Avalon Memory Mapped
avmm_1_r_address [48] address
avmm_1_r_burstcount [3] burstc
avmm_1_r_byteenable [64] byte
Import... Clone... Mirror...
Messages
Type Path Messa
Help Finish Cancel

HDL entity name: fpVectorReduce_ac_int
HDL compilation library: fpVectorReduce_ac_int
IP file: ../hls/test-fpga.prj/components/fpVectorReduce_ac_int/fpVectorReduce_ac_int.ip
Assignments: Edit...

11. Connect the new HLS component. In this case, the new component is the `fpVectorReduce_ac_int` component.

    Make sure that you set the base address of your component Avalon-MM slave to `0x0040`, so that the HLS component and AFU ID slave can share the same memory space, and the host application can access both.

Figure 12. Set Base Address



Connections	Name	Base
	⊟ **clock_in**	
	in_clk	
	out_clk	
	⊟ **reset_in**	
	clk	
	in_reset	
	out_reset	
	⊟ **mm_bridge_0**	
	clk	
	m0	
	reset	
	s0	
	⊟ **afu_id_avmm_slave_0**	
	afu_cfg_slave	0x0000
	clock	
	reset	
	⊟ **read_bridge**	
	clk	
	m0	
	reset	
	s0	0x0
	⊟ **far_reach_avalon_mm_bridge_irq_0**	
	clk	
	interrupt_receiver	IRQ 0
	interrupt_sender	
	master	
	reset	
	slave	0x0
	⊟ **fpVectorReduce_ac_int_internal_0**	
	avmm_1_r	
	avmm_2_w	
	avs_cra	0x0040
	clock	
	irq	
	reset	

12. Click **Validate System Integrity** to update the Platform Designer system. A message appears at the bottom of the Platform Designer window indicating success.

13. Save the system and exit Platform Designer.

    Click **Yes** if you are asked to generate the changes.

14. Configure your AFU by modifying the Acceleration Stack configuration files `hls_afu/filelist.txt` and `hls_afu/hls_afu.json`.

    For examples of these files, see Acceleration Stack Configuration Files on page 23.

15. Modify the host application to include the Avalon-MM slave register map that is produced by HLS.

    a. Copy the `hls_afu/hw/rtl/hls/test-fpga.prj/components/fpVectorReduce_ac_int/fpVectorReduce_ac_int_csr.h` header file to the host application project in `hls_afu/sw/src`.

    b. In `hls_afu_host.c`, include `fpVectorReduce_ac_int_csr.h` instead of `fpVectorReduce_float_csr.h`. Also replace the references to `fpVectorReduce_float` registers with references to `fpVectorReduce_ac_int` registers.

After you have customized the HLS AFU:

1. Generate the ASE testbench.
2. Compile the AF bitstream.

Related Information

- HLS AFU Avalon-MM I/O Slave Interface on page 28
- HLS AFU Avalon-MM Master Interfaces on page 28
- Compiling and Simulating the HLS Component with the i++ Command on page 7
- Generating a Platform Designer Container for the HLS Component on page 9
- Generating the ASE Testbench on page 10
- Compiling the AF Bitstream on page 14
- Accelerator Functional Unit (AFU) Developer's Guide

3.1. Acceleration Stack Configuration Files

The Acceleration Stack needs two configuration files to build and run the AFU: a build configuration file (`filelist.txt`) and a platform configuration (`.json`) file.

The build configuration file lists the AFU design source (for example, RTL, IP, Platform Designer subsystems, and constraints) along with any required macro definitions and `include` files.

The platform configuration file specifies information needed by the OPAE platform interface manager (PIM) to generate a platform shim that provided the top-level module interface for the AFU.

For more details about these files, see Accelerator Functional Unit (AFU) Developer’s Guide

The example information in these descriptions refer to the HLS AFU design example. If you are integrating a custom HLS component, your information might be different.

Build Configuration (`filelist.txt`) File

The `hls_afu/hw/filelist.txt` file must contain paths to:

- All the top-level source files. For example, `afu.sv, ccip_interface_reg.sv`, and `ccip_std_afu.sv`
- CCI-P/Avalon adapter. For example, the lines:

```
QI:BBB_ccip_avmm/hw/par/ccip_avmm_addenda.qsf
SI:BBB_ccip_avmm/hw/sim/ccip_avmm_sim_addenda.txt
```

- MPF BBB. For example, the lines:

```
+define+MPF_PLATFORM_DCP_PCIE=1
QI:BBB_cci_mpf/hw/par/qsf_cci_mpf_PAR_files.qsf
SI:BBB_cci_mpf/hw/sim/cci_mpf_sim_addenda.txt
```

- The Platform Designer system, `hls_afu_container.qsys`.
- All IP parameterizations that the Platform Designer system uses. For example, all the `.ip` files listed in `qsys/ip/hls_afu_container` (e.g. `hls_afu_container_mm_bridge_0.ip`), and any `.ip` files that the HLS Compiler produces (e.g. `fpVectorReduce_float.ip`).
- All directories that contain components that are not in the `qsys/ip/hls_afu_container folder`. For example, the lines:

```
+incdir+hls/test-fpga.prj/components/fpVectorReduce_float
+incdir+hls/test-fpga.prj/components/fpVectorReduce_float/ip
```

You do not need to explicitly link your Platform Designer system RTL sources or the RTL produced by HLS (other than the IP files you use). If your design does not instantiate any IP files that filelist.txt refers to in your design, Intel Quartus Prime Pro Edition fails when you compile the AF bitstream. For this design, you may comment out the lines that contain references to `fpVectorReduce_float`, and uncomment the lines that contain references to `fpVectorReduce_ac_int`.

HLS AFU Platform Configuration (`hls_afu.json`) file

The `hls_afu.json` file must contain the accelerator UUID, the name of the AFU, and a top interface to define which PAC resources the AFU needs.

For more details about specifying the AFU platform configuration, refer to the Accelerator Functional Unit (AFU) Developer’s Guide. If you add additional interfaces, you may also need to modify `afu.sv` and `ccip_std_afu.sv`.

Related Information

Accelerator Functional Unit (AFU) Developer’s Guide

4. HLS AFU Design Example Description

The HLS AFU design comprises an AFU (contained in the hw folder) that runs on an Intel FPGA Programmable Acceleration Card (Intel FPGA PAC), and a host application that runs on an Intel Xeon CPU.

4.1. AFU Description

Like other AFU designs, the HLS AFU design example defines the top-level functionality of the AFU in `ccip_std_afu.sv`. The design example implements all the compute functionality of the AFU in the HLS component. However, the design example needs some RTL to initialize and connect the required hardware components.

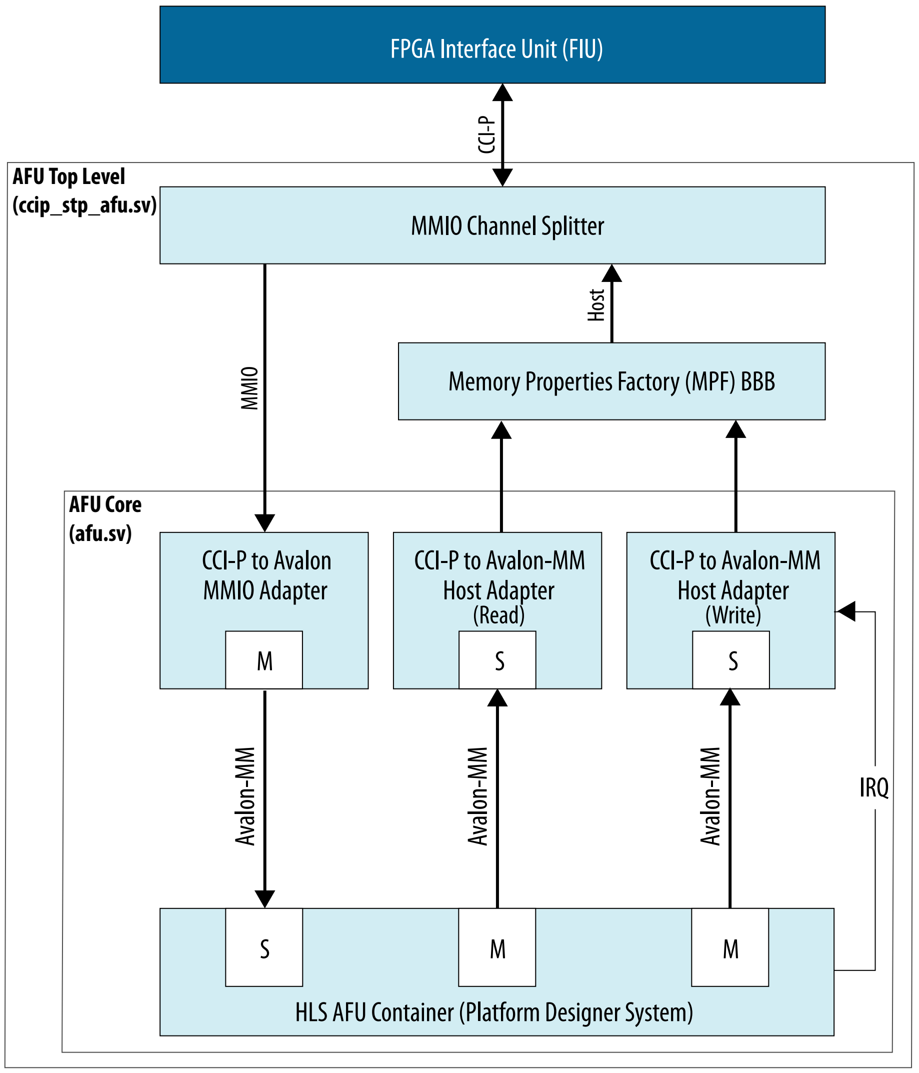
This design is based on the DMA AFU design, except that this design lacks Avalon-MM master interfaces for communicating with the DRAM banks on the Intel FPGA PAC.

The most important parts of this design are the CCI-P to Avalon-MM adapter component and MPF BBB. These components buffer CCI-P transactions and translate them to Avalon-MM transactions, and vice-versa.

The MPF BBB and CCI-P to Avalon-MM adapter components included with this design support more of the Avalon specification than the adapter included with the DMA AFU design. For more details about the DMA AFU design, refer to the DMA Accelerator Functional Unit (AFU) User Guide.

These two components connect the HLS component with the Acceleration Stack host, because the Acceleration Stack infrastructure only exposes a CCI-P, not an Avalon-MM interface.

Figure 13. Overview of HLS AFU hardware


FPGA Interface Unit (FIU)
CCI-P
AFU Top Level (ccip_stp_afu.sv)
MMIO Channel Splitter
Host
MMIO
Memory Properties Factory (MPF) BBB
AFU Core (afu.sv)
CCI-P to Avalon MMIO Adapter
M
CCI-P to Avalon-MM Host Adapter (Read)
S
CCI-P to Avalon-MM Host Adapter (Write)
S
Avalon-MM
Avalon-MM
Avalon-MM
IRQ
S
M
M
HLS AFU Container (Platform Designer System)

4.1.1. HLS AFU Container

The HLS AFU container is a Platform Designer system that contains the component produced by the HLS compiler and some supporting components.

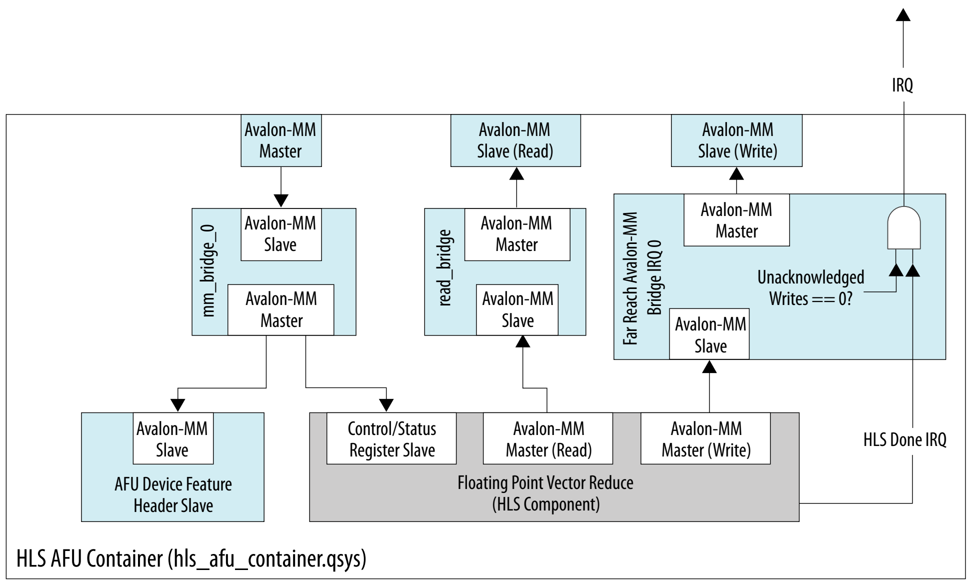
Table 1. HLS AFU Container Components

Component	Description
mm_bridge_0 (Avalon-MM pipeline bridge)	Allows the slave interfaces of the HLS component and the AFU ID slave to share the same address space as the master interface.
afu_id_avmm_slave_0 (AFU ID Avalon-MM slave)	Stores the AFU's DFH, which includes its UUID. Expose the DFH to the host (refer to the *Accelerator Functional Unit (AFU) Developer's Guide*)

continued...

Component	Description
floatingPointVectorReduce_float	The HLS component. This component consumes a vector of floating-point numbers and reduces it by adding all the elements together.
read_bridge (Avalon-MM pipeline bridge)	Pipelines the HLS component read-only Avalon-MM master interface
far_reach_avalon_mm_bridge_irq_0 (Far reach Avalon-MM bridge with IRQ gate)	The HLS component Avalon-MM master interface does not support write-acknowledgements from host memory. This custom component prevents the HLS component's interrupt signal until the design acknowledges all outstanding host memory writes. The component ensures that the design does not interrupt the host until you write all results to host memory. If you do not use this component, the HLS done interrupt might reach the host application before the design commits all writes to host memory.

Figure 14. HLS AFU Container block diagram



The HLS component performs a simple vector reduction. It also copies the input vector back into host memory, incrementing each vector value by 1.0f.

The following code is a simple example of an HLS component that performs vector reduction. It reads a single 32-bit floating-point value each cycle and accumulates the total.

Figure 15. Simplified HLS Component

```
component
float floatingPointVectorReduce_basic(float *masterRead,
                                      float *masterWrite,
                                      int size)
{
  float sum = 0.0f;
  for (int idx = 0; idx < size; idx++)
  {
    float readVal = masterRead[idx];
    sum += readVal;

    masterWrite[idx] = readVal + 1.0f;
  }
  return sum;
}
```

While this example is valid HLS source code, it is insufficient for an AFU design.

The Intel Acceleration Stack has specific requirements that dictate how AFUs can access host memory and how the host system sees them. Fortunately, the Intel HLS Compiler is flexible enough that you can reconfigure your component to meet the following constraints required by the Intel Acceleration Stack:

- The controls and parameters should be exposed through an Avalon-MM I/O slave interface, not the conduit interfaces that the Intel HLS Compiler uses by default.
- AFUs may have two Avalon-MM master interfaces for accessing host memory. Configure one Avalon-MM master as read-only; the other as write-only.
  - Both Avalon-MM master interfaces must be 512 bits wide.
  - Both Avalon-MM master interfaces must use 48-bit addresses.

4.1.1.1. HLS AFU Avalon-MM I/O Slave Interface

The HLS `hls_avalon_slave_component` attribute in the `hls_afu/hw/rtl/src/hls_afu.h` header file moves the `start`, `busy`, `stall`, and done control signals into the component control and status register (CSR). You can also apply the `hls_avalon_slave_register_argument` attribute to each of the component parameters to move them into the component CSR.

A generic component function signature is as follows:

```
hls_avalon_slave_component
component
float fpVectorReduce_basic(
    hls_avalon_slave_register_argument float *masterRead,
    hls_avalon_slave_register_argument float *masterWrite,
    hls_avalon_slave_register_argument uint64 size)
```

When you access the CSR of your HLS component, use 64-bit reads and writes.

4.1.1.2. HLS AFU Avalon-MM Master Interfaces

An AFU can have two Avalon-MM master interfaces for accessing system memory. You must configure one Avalon-MM master as read-only; the other as write-only.

This requirement requires modifications to both the HLS component signature and the algorithm.

Because the smallest unit of data that an AFU can access in host memory is 64 bytes (512 bits), configure the Avalon-MM master interfaces by using HLS `ihc::mm_master` objects. For details about the parameters, refer to the Intel High Level Synthesis Compiler Reference Manual.

Then, modify the code found in the Simplified HLS Component to take advantage of the bandwidth afforded by the mandatory 512-bit data bus and access 16 32-bit values concurrently.

This access-size constraint means that if your vector length is not a multiple of 16 (equivalently, a multiple of 64 bytes), the host memory locations between the end of your vector and the next multiple of 16 fills with meaningless data.

For best performance, do not attempt read-modify-write operations. Because the design has two separate Avalon-MM master interfaces, the Intel HLS Compiler assumes separate address spaces, and assumes no dependencies exist between the two Avalon-MM master interfaces.

The HLS AFU design example demonstrates how to connect with the 512-bit Avalon-MM master interfaces. You can do the following things:

- Let the Intel HLS Compiler abstract away that detail and assume the accesses are floats.
- Assume host-memory accesses are 512-bit unsigned integers.

`float` Accesses

You can define the `ihc::mm_master` using an unsigned 512-bit `float` as the underlying data type.

Figure 16. Signature for `float`-based component

```
typedef ihc::mm_master<float, ihc::dwidth<512>, ihc::awidth<48>,
                       ihc::latency<0>, ihc::aspace<1>,
                       ihc::readwrite_mode<readonly>, ihc::waitrequest<true>,
                       ihc::align<64>, ihc::maxburst<4>> MasterReadFloat;
typedef ihc::mm_master<float, ihc::dwidth<512>, ihc::awidth<48>,
                       ihc::latency<0>, ihc::aspace<2>,
                       ihc::readwrite_mode<writeonly>,
                       ihc::waitrequest<true>, ihc::align<64>,
                       ihc::maxburst<4>> MasterWriteFloat;

component
hls_avalon_slave_component
float fpVectorReduce_float(hls_avalon_slave_register_argument MasterReadFloat
&masterRead,
                           hls_avalon_slave_register_argument MasterWriteFloat
&masterWrite,
                           hls_avalon_slave_register_argument uint64_t size);
```

Use the following parameter settings to enable an HLS component to communicate with host memory from the AFU:

- 48-bit wide address (`awidth` parameter)
- DRAM: requires variable latency and wait-request signal (`latency` and `waitrequest` attributes)
- 64 concurrent bytes can be read at once (`align` parameter)
- Maximal burst size is 4 512-bit reads (`maxburst` parameter)
- Separate physical Avalon-MM master ports (`aspace` parameter). Read-only and write-only (one Avalon-MM master of each) (`readwrite_mode` parameter)

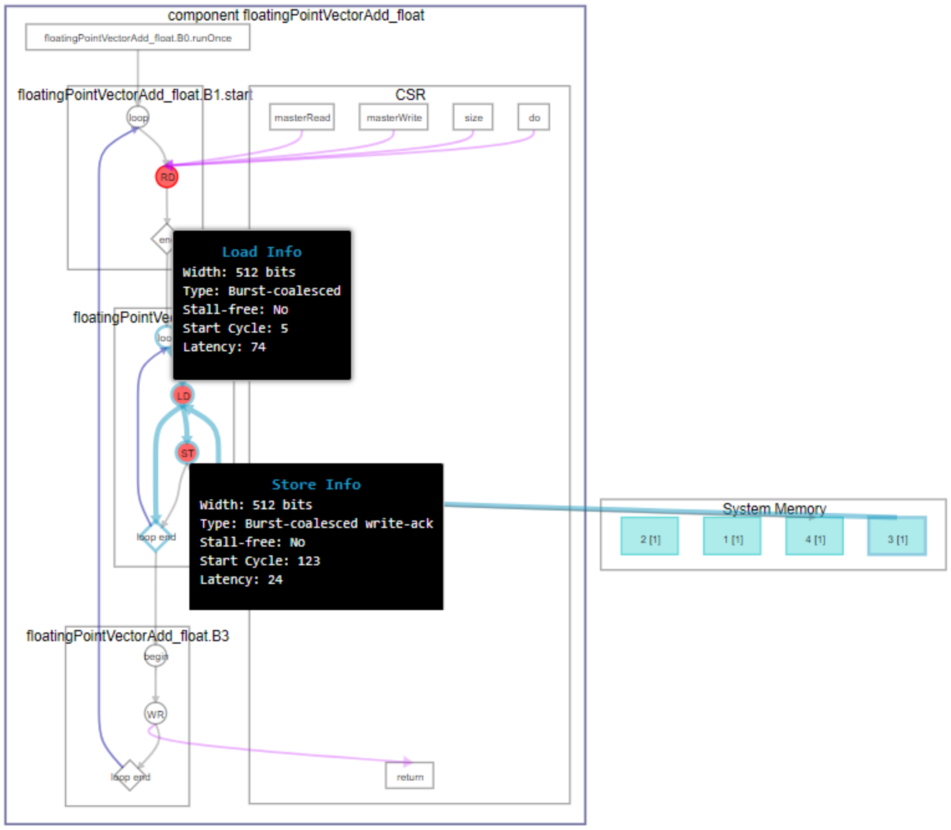
Figure 17. `float`-based body

```
#pragma unroll 16
for (int itr = 0; itr < 16; itr++)
{
  int idx = itr + (loop_idx * 16);
  if (idx < size)
  {
    float readVal = masterRead[idx];
    readSum += readVal;
     masterWrite[idx] = readVal + 1.0f;
  }
}
sum += readSum;
```

Access the `mm_master` inside the unrolled loop body 32 bits at a time. To verify that the compiler infers everything properly, look at the `Component Viewer` section of the generated HLS `report.html` report to verify that you have 512-bit burst-coalesced LSUs, and make sure that they are aligned. If you want to be certain that your loads occur 512 bits at a time, look at the simulation waveforms Figure 19 on page 31.

Figure 18. HLS Report showing float-based component.

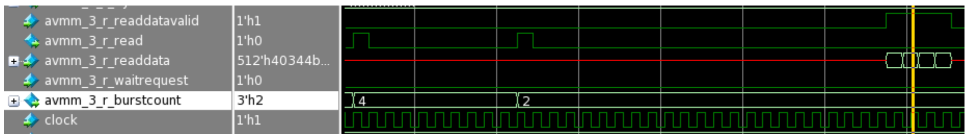
Observe the coalesced Avalon-MM master interfaces.


component floatingPointVectorAdd_float
floatingPointVectorAdd_float.B0.runOnce
floatingPointVectorAdd_float.B1.start
CSR
masterRead
masterWrite
size
do
loop
RD
floatingPointVe
LD
ST
loop end
System Memory
2 [1]
1 [1]
4 [1]
3 [1]
floatingPointVectorAdd_float.B3
begin
WR
loop end
return

```
Load Info
Width: 512 bits
Type: Burst-coalesced
Stall-free: No
Start Cycle: 5
Latency: 74
```

```
Store Info
Width: 512 bits
Type: Burst-coalesced write-ack
Stall-free: No
Start Cycle: 123
Latency: 24
```

Figure 19. ModelSim waveform showing host memory accesses of `float`-based component



```
avmm_3_r_readdatavalid   1'h1
avmm_3_r_read            1'h0
avmm_3_r_readdata        512'h40344b...
avmm_3_r_waitrequest     1'h0
avmm_3_r_burstcount      3'h2
clock                    1'h1
```

ac_int Accesses

You can define the `ihc::mm_master` using an unsigned 512-bit `ac_int` as the underlying data type.

This signature is the same as the Figure 16 on page 29, except that that `mm_master` type is `ac_int` instead of `float`.

Figure 20. Signature for `ac_int`-based component

```
typedef ac_int<512, false> uint512; // 512-bit unsigned integer
typedef ihc::mm_master<uint512, ihc::dwidth<512>,
                       ihc::awidth<48>, ihc::latency<0>,
                       ihc::aspace<1>, ihc::readwrite_mode<readonly>,
                       ihc::waitrequest<true>, ihc::align<64>,
                       ihc::maxburst<4> > MasterReadAcInt;

typedef ihc::mm_master<uint512, ihc::dwidth<512>,
                       ihc::awidth<48>, ihc::latency<0>,
                       ihc::aspace<2>, ihc::readwrite_mode<writeonly>,
                       ihc::waitrequest<true>, ihc::align<64>,
                       ihc::maxburst<4> > MasterWriteAcInt;

component
hls_avalon_slave_component
float fpVectorReduce_ac_int(
   hls_avalon_slave_register_argument MasterReadAcInt &masterRead,
   hls_avalon_slave_register_argument MasterWriteAcInt &masterWrite,
   hls_avalon_slave_register_argument uint64_t size);
```

This method is more verbose, but it guarantees that all Avalon-MM master accesses coalesce to 512-bits wide. You can access the 32-bit parts of the 512-bit wide read result using the `slc` and `set_slc` functions provided by `ac_int` (refer to the *ac_int Reference Manual, Mentor Graphics Corporation* for more information on these functions). This component explicitly performs 512-bit reads and writes (line 1 and line 36).

Figure 21. ac_int-based body

As found in `hls_afu.cpp`

```
uint512 readVal = masterRead[loop_idx];
uint512 writeVal = 0;

#pragma unroll 16 // do each loop iteration concurrently.
                   // Use 16 iterations because there are 16
                   // 32-bit slices in each 512-bit word.
for (int itr = 0; itr < 16; itr++)
{
  int idx = itr + (loop_idx * 16);
  if (idx < size)
  {
    // grab a 32-bit piece of the 512-bit value that we read
    uint32 readVal_32 = readVal.slc<32>(itr * 32);

    // use explicit type casting to process the bits pointed
    // to by &readVal_32 as a float.
    void *readVal_32_ptr = &readVal_32;
    float readVal_f;
    float *readVal_f_ptr = &readVal_f;
    *readVal_f_ptr = *((float *) readVal_32_ptr);
    readSum += readVal_f;

    // increment and output
    float writeVal_f = readVal_f + 1.0f;

    // use explicit type casting to process the bits pointed
    // to by &writeVal_f as a uint32.
    float *writeVal_f_ptr = &writeVal_f;
    uint32 *writeVal_32_ptr = (uint32 *) writeVal_f_ptr;
    uint32 writeVal32 = *writeVal_32_ptr;

    unsigned int bit_offset = itr * 32;
    writeVal.set_slc(bit_offset, writeVal32);
  }
}
masterWrite[loop_idx] = writeVal;
```

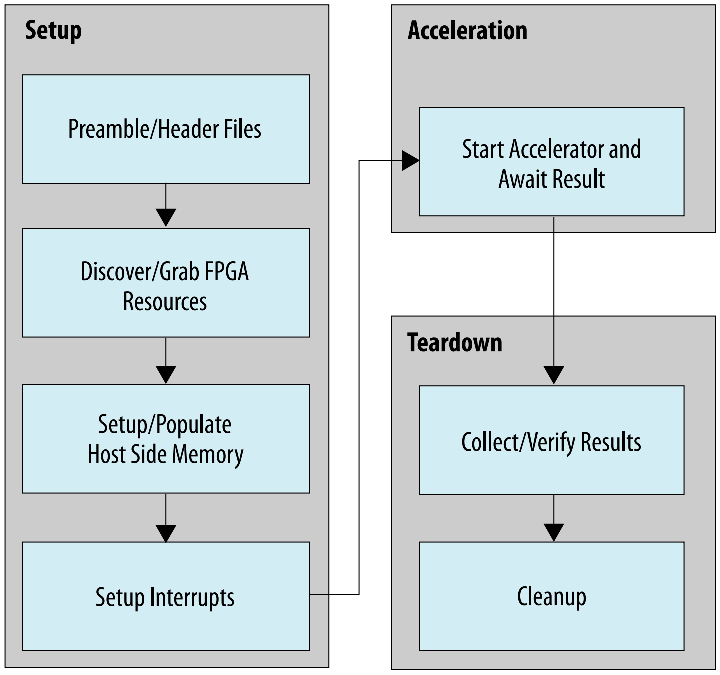
Related Information

Intel High Level Synthesis (HLS) Compiler Reference Manual

4.2. Host Application Description

The host application uses the OPAE software API to communicate with the accelerator that runs inside your Intel PAC. OPAE is an Intel API that allows host applications to access the functionality of accelerators such as FPGA cards. You can learn more about the general usage of the OPAE software API in the *Open Programmable Acceleration Engine (OPAE) C API Programming Guide*.

Figure 22. Flow summary of a typical OPAE application


Setup
Preamble/Header Files
Discover/Grab FPGA Resources
Setup/Populate Host Side Memory
Setup Interrupts
Acceleration
Start Accelerator and Await Result
Teardown
Collect/Verify Results
Cleanup

This host application is simplified compared with a production design host application. All API calls occur in the main source file. In a production design, you are more likely to make these API calls in libraries, similarly to the DMA AFU design. For clarity, all the host code is in a single source file, `hls_afu/sw/src/hls_afu_host.c`. The headings in this section match the headings in `hls_afu_host.c`.

Table 2. AFU Avalon MM Slave memory map

Slave Name	Address Range
Device feature header slave (`afu_id_avmm_slave_0`)	0x0000 to 0x003F
HLS component (`fpVectorReduce_ac_int_internal_0` or `fpVectorReduce_float_internal_0`)	0x0040 to 0x007F

Preamble/Header Files

The first section of the host code includes necessary libraries, and defines several constant address offsets. The design derives the CSR constants for the HLS component from the constants in `fpVectorReduce_float_csr.h`, which the HLS compiler emits. Because the HLS component's slave interface shares a memory space with the AFU ID MM slave, a base offset ensures that each register in the AFU ID MM slave and the HLS component has a unique address.

Discover/Grab FPGA Resources

This block of code is boilerplate. The design queries the FPGA hardware for available accelerators, and if the design finds the accelerator required by the host application, the host application attempts to control the FPGA device. In this design, the host also exercises the AF registers that the Acceleration Stack requires. The HLS component does not implement these registers, which are in the AFU device feature header Avalon-MM slave.

Setup and Populate Host-Side Memory

This block of code configures a contiguous host-side memory buffer that the AF can access. When you run an Acceleration Stack host, make sure that you configure it to use 2 MB hugepages using this command (you do not need this command if you are running using ASE):

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

This command allows the host application to allocate 2MB pinned continuous buffers in its memory.

The `fpgaPrepareBuffer()` function allocates the host-side buffer that the design shares with the AF. This function allocates a block of memory starting at a user-specified address. Additionally, it guarantees that the memory block is 64-byte (512 bits) aligned, which makes the AFU accesses efficient. `fpgaGetIOAddress()` gets a pointer that the AF can use to access the same memory space as the host. The host can populate the block of RAM as it does for any other array.

Setup Interrupts

This design uses an interrupt framework to allow the AF to report to the host when it finishes processing. The HLS component generates the required interrupt in this design, so the host needs to write into the HLS component's slave memory space to enable the interrupt. First, the host checks if the interrupt is already enabled, by reading the CSR_INTERRUPT_ENABLE register. Refer to the Hello Interrupt AFU example included with Intel Acceleration Stack for more details about interrupts.

Start AF and Wait for Result

To start the AF, the host writes input variables into the HLS component's slave space (input data starting address, output data starting address, data size). Then it writes a 1 into the START bit in the HLS component's slave space. Using the poll API, the host waits for the AF to finish.

Check to Make Sure that the Calculation Was Correct

The host checks that the interrupt returned correctly (or did not time out) and verifies that the output memory contains the expected values. This design also prints out some debug data at the end of the memory space, to illustrate that AFs can only perform 512-bit reads and writes. If you pass a vector whose length is not a multiple of 512 bits, (64 bytes), the design overwrites some data in the output vector memory space.

Cleanup

Finally, the host application disposes of the resources that it allocated during its execution.

Related Information

Open Programmable Acceleration Engine (OPAE) C API Programming Guide

5. Troubleshooting HLS AFU Designs

5.1. Platform Designer Opens with an Error

If you open the Platform Designer system, you might see the following error when you try to generate your system: `Error: Failed to retrieve source files from Quartus project, manually re-run the commands included in …/quartus_sh_tcl_file_for_qsyspro.tcl in Quartus tcl shell.`

You can ignore this error if you delete the files generated by Platform Designer, as they regenerate when you generate the ASE testbench and when you compile the AF bitstream.

The `qsys-edit` command might not work if your $PATH environment variable does not include the path to Platform Designer.

You can also avoid these errors by opening the temporary Intel Quartus Prime Pro Edition project you create in *Changing Components in the HLD AFU Design Example* using Intel Quartus Prime Pro Edition 17.1.1, and then opening Platform Designer using the Intel Quartus Prime Pro Edition GUI (instead of using `qsys-edit` from the command line).

5.2. The `design unit was not found` Error When Running the `make sim` Command

These types of errors commonly occur with your HLS AFU design when you incorrectly specify your `filelist.txt`. Ensure that the IPs and folders listed in `filelist.txt` match the structure of your project. Pay attention to the names of the IP files, as sometimes Platform Designer renames the IPs it uses. These errors may also occur if you added the HLS-generated IPs to the Intel Quartus Prime project-level search path instead of the global search path.

```
# Loading work.hls_afu_container_afu_id_Avalon-MM_slave_0
# Loading work.afu_id_Avalon-MM_slave
# Loading work.hls_afu_container_clock_in
# ** Error: (vsim-3033) /home/john/hls_afu/hls_afu/build_ase_dir/qsys_sim/
qsys_0/hls_afu_container/synth/hls_afu_container.v(136): Instantiation of
'fpVectorReduce_ac_int' failed. The design unit was not found.
#    Time: 0 ns  Iteration: 0  Instance: /ase_top/platform_shim_ccip_std_afu/
ccip_std_afu/afu_inst/u0 File: /home/john/hls_afu/hls_afu/build_ase_dir/
qsys_sim/qsys_0/hls_afu_container/synth/hls_afu_container.v
#         Searched libraries:
#             /home/john/hls_afu/hls_afu/build_ase_dir/work/work
# Loading work.hls_afu_container_mm_bridge_0
# Loading work.altera_Avalon_mm_bridge
# Loading work.hls_afu_container_mm_bridge_1
# Loading work.hls_afu_container_reset_in
```

5.3. Verilog HDL Compilation Errors

The **hls_afu_container** Platform Designer system is instantiated in `afu.sv`. Make sure that the instantiation matches the interface defined in `hls_afu/hw/rtl/qsys/hls_afu_container/hls_afu_container_inst.v` (which appears after generating the **hls_afu_container** system in Platform Designer).

5.4. Compilation Errors During ASE Testbench Generation

Make sure that you have correctly set up your system environment variables. Refer to *Setting Up the Environment* in the *Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) Quick Start User Guide*.

Running the `make` utility from the `build_ase_dir` directory causes an error if the environment variables for your simulation software are not set. Ensure that you set the following environment variables, depending on your simulation software:

- For Mentor ModelSim SE or Mentor Questa Advanced Simulator:

```
$ export MTI_HOME=<path to ModelSim/Questa installation directory>
$ export PATH=$MTI_HOME/linux_x86_64/:$MTI_HOME/bin/:$PATH
```

Even if you have the correct path information to your simulation software set in your PATH environment variables, the Acceleration Stack requires the simulator-specific environment variables to be set.

If the environment variables are not set correctly, running the `make` command, generates an error message as follows:

```
In file included from /home/john/hls_afu/hls_afu/build_ase_dir/sw/
linked_list_ops.c:36:0:
/home/john/hls_afu/hls_afu/build_ase_dir/sw/ase_common.h:66:19: fatal error:
svdpi.h: No such file or directory
compilation terminated.
In file included from /home/john/hls_afu/hls_afu/build_ase_dir/sw/
randomness_control.c:28:0:
/home/john/hls_afu/hls_afu/build_ase_dir/sw/ase_common.h:66:19: fatal error:
svdpi.h: No such file or directory
compilation terminated.
make[1]: *** [sw_build] Error 1
make[1]: Leaving directory `/home/john/hls_afu/hls_afu/build_ase_dir'
make: *** [build] Error 2
```

If the environment variables are not set correctly, running the `make sim` command, generates an error message as follows:

```
# ** Error: (vsim-3763) SystemVerilog DPI cannot access file './ase_libs.so'
# No such file or directory. (errno = ENOENT)
# Use the -help option for complete vsim usage.
# vsim -c -l run.log -dpioutoftheblue 1 -novopt -sv_lib ase_libs -do "/home/
john/hls_afu/hls_afu/vsim_run.tcl" \
-sv_seed 1234 "+CONFIG=/home/john/hls_afu/hls_afu/ase.cfg" "+SCRIPT=/home/john/
hls_afu/hls_afu/ase_regress.sh" ase_top
# Error loading design
Error loading design
# Errors: 1, Warnings: 0
/bin/sh: line 0: cd: OLDPWD not set
make: *** [sim] Error 1
```

Related Information

Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) Quick Start User Guide

5.5. Incorrect Output During Simulation

ASE saves all waveforms from your HLS AFU design, so you can use these to debug your design.

For details, see section 1.4.1.1 of the *Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA*.

Related Information

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

5.6. AF Bitstream Compilation Fails

Make sure that you don't have any unused IPs defined in `filelist.txt`. Delete the Platform Designer-generated files (`hls_afu_container` directory) from the `hls_afu/hw/rtl/qsys` directory

5.7. Verilog Files Not Found Errors

When you run the `setup_ase.sh` or `setup_gpb.sh` scripts, you might sometimes get error messages indicating that Verilog files were not found.

When you get a Verilog files not found error from running these scripts, open the `hls_afu_container.qys` file in Platform Designer and manually generate the system. You can also try to generate the `hls_afu_container.qys` system from a command line.

6. Document Revision History for the HLS AFU Design Example User Guide

Document Version	Changes
2019.07.19	• Corrected error in step 6 in Generating a Platform Designer Container for the HLS Component on page 9: The step now says "After **Validate System Integrity** successfully completes, click **Close**.". Previously this step referred to "**Sync System Infos**". • Revised step 10 in Customizing the HLS AFU on page 17 and added new images to show UI before and after effect of the step. • Changed document part number to UG-20246. Previously, this document was part number UG-20192.
2019.05.10	• Updates for Intel Acceleration Stack for Intel Xeon CPU with FPGAs Version 1.2 and Intel HLS Compiler Pro Edition Version 19.1.
2019.03.12	• Corrected *Correct Directory Structure* figure.
2019.01.31	• Corrected code: `$ $OPAE_PLATFORM_ROOT/bin/run.sh`
2018.11.30	• Initial release.