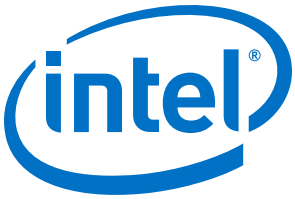


Intel High Level Synthesis Accelerator Functional Unit Design Example User Guide

UG-20192 | 2019.03.12
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 with the Intel HLS compiler. This design example transfers data between a host program and a simple AFU generated with the HLS compiler. To use this design, you should be familiar with the fundamentals of both the HLS compiler and the Acceleration Stack.

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. Acronyms in the HLS AFU Design Example User Guide

Acronym	Definition	Description
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	The FIM bitstream is 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.
HLS	High-level synthesis	A compiler that translates C++ source code into RTL for use in FPGA designs
MPF	Memory properties factory	The MPF is a BBB that AFUs can use to provide CCI-P traffic shaping operations for transactions with the FIU.

continued...

Acronym	Definition	Description
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 a 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 HFS AFU Design Example

You need various commands to build and test the HLS AFU design. If you want to quickly verify your installation, you can follow the abbreviated instructions in the `README.txt` file.

The design example shows how you add an HLS component to an Intel Acceleration Stack AFU. The HLS AFU design example targets the Intel Programmable Acceleration Card (PAC) 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 the user's 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.

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 version 1.1 or later (and associated prerequisites)
- Intel Quartus® Prime Pro Edition v17.1.1 for the Acceleration Stack software
- Intel Quartus Prime Pro Edition v18.0 or later to compile your HLS code

Note: Intel only supports HLS v18.0 for RHEL 6.x (or a community equivalent). Refer to *Troubleshooting* for a workaround to run HLS 18.0 on RHEL 7.x (or a community equivalent).

Related Information

- Troubleshooting HLS AFU Designs on page 34
- Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA

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 is Intel Quartus Prime Pro Edition v18.0 or later.

Note: The HLS code may fail to compile if you are using HLS v18.0 in an environment that does not have the correct version of the GCC libraries. If you are using RHEL 7.x, you must install the gcc 4.4.7 compatibility libraries:

```
# sudo yum install compat-gcc-44 compat-gcc-44-c++
```

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 Intel Quartus Prime Pro Edition installation directory.

   For example:

   ```
   $ cd /home/<username>/intelFPGA_pro/18.0/hls
   ```

   Note: If you are using v18.0 on RHEL 7.x, you must override the CPLUS_INCLUDE_PATH environment variable to have i++ prioritize gcc 4.4.7.

   ```
   $ export CPLUS_INCLUDE_PATH=/usr/lib/gcc/x86_64-redhat-linux/4.4.7:/usr/
   include/c++/4.4.7:/usr/include/c++/4.4.7/x86_64-redhat-linux
   ```

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. Run the i++ command from this terminal session (the HLS window).

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

4. To build and emulate the design using x86 instructions, navigate to that directory using the HLS window, and run these commands:

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

   You see 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
```

Note: If you are using HLS v18.1 to generate your accelerator, you should use the `--save-temps` flag when you invoke i++. The included makefile detects HLS v18.1 and uses this flag automatically.

2.3. Generating a Platform Designer Container for the HLS Component

Verify that all sources are linked correctly. Ensure the Acceleration Stack environment is Intel Quartus Prime Pro Edition v17.1.1.

1. If you generated your AFU using HLS v18.1, you need to prepare the `.ip` file for Platform Designer v17.1.1:
   a. Navigate to `hls_afu/hw/rtl/hls/test-fpga.prj/components/<component>` (in this example it is `hw/rtl/hls/test-fpga.prj/components/fpVectorReduce_float)/`, and run this script:

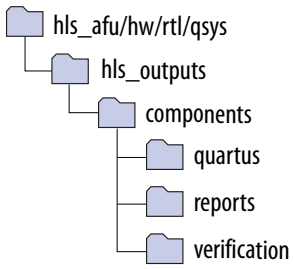
   ```
   $ qsys-script --script=<component>.tcl --quartus-project=none
   ```

   b. For the **fpVectorReduce_float** component, run this command:

   ```
   $ qsys-script --script=fpVectorReduce_float.tcl --quartus-project=none
   ```

   A new `fpVectorReduce_float.ip` file is generated that is compatible with Platform Designer v17.1.1.
2. Copy the `components`, `quartus`, `reports`, and `verification` folders from the `test-fpga.prj` folder to the `hls_outputs` folder in the Platform Designer project.

Figure 1. Correct Directory Structure



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

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

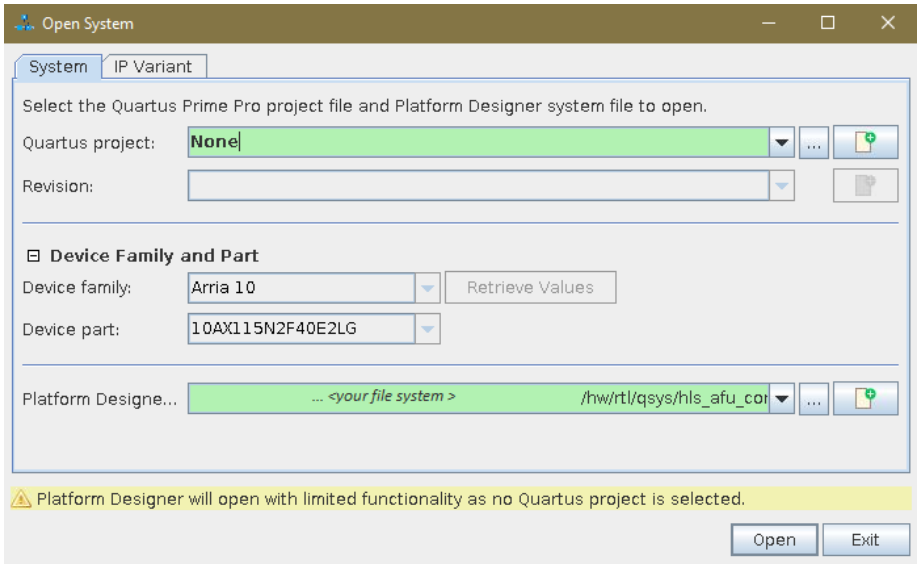
4. In the **Open System** dialog, select **None** for the **Quartus project** dropdown.

   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.

5. Click **Open**.

Figure 2. Open System Dialog box



6. To reload the system and ensure that all search paths are correct, click **Sync System Info** at the bottom of the **Open System** window.

   If the reload fails, make sure you delete any extra files in the `hls_outputs` folder.

7. Click **Finish**.
8. Click **No** to not generate the HDL.

2.4. Generating the ASE Testbench

If you want to run your design in hardware, skip this task and go to *Compiling and test the AF Bitstream*. Otherwise, if you want to simulate your design, continue. Refer to `setup_ase.sh` to automate these steps.

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 Libraries` section (Line 98):

```
# 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
```

4. Run the following commands:

```
$ make
$ make sim
```

When you see this message you can build and run the host:

```
#   [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...
```

2.5. Running the ASE Testbench

1. Open a new terminal window, the **host** window, to compile the host application. The previous terminal window is the **ASE** window.
2. Run these commands in the **host** window:
   a. Navigate to `hls_afu/sw`.
   b. Export the *ASE_WORKDIR* environment variable using the export command from the **ASE** window:

```
$ export ASE_WORKDIR=<path to work folder>
```

c. In the sw directory, run:

```
$ make USE_ASE=1
```

3. Run the host executable in the host window using the command:

```
$ ./hls_afu_host
```

When the executable runs successfully, the host application executes in the host window and displays a `Test Passed` message. All CCI-P transactions execute in the ASE window. You can view the ASE simulation waveforms if you want to debug your AFU. For more details, refer to the ASE documentation.

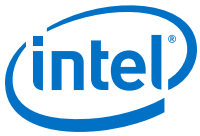


Figure 3. Host window (end of output)

```
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 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...
#
#   Transaction count     |       VA      VL0      VH0      VH1 |    MCL-1
MCL-2    MCL-4
#
================================================================================
==========
#   MMIOWrReq           6 |
#   MMIORdReq          11 |
#   MMIORdRsp          11 |
#   IntrReq             1 |
#   IntrResp            1 |
#   RdReq               4 |        0        0        1        0 |
0        0        1
#   RdResp              4 |        0        0        4        0 |
#   WrReq               4 |        0        0        4        0 |
0        0        1
#   WrResp              4 |        0        0        1        0 |
0        0        1
#   WrFence             0 |        0        0        0        0 |
#   WrFenRsp            0 |        0        0        0        0 |
#
# ** 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
```

2.6. Compiling the AF Bitstream

This task takes significantly longer than simulating the AFU because it runs Intel Quartus Prime and generates an FPGA bitstream. Refer to `setup_gbs.sh` to automate steps 1 and 2.

1. Configure your system to use appropriately sized hugepages:

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

2. 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
```

The output of a successful completion AF creation process is:

```
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

`run.sh` may fail if you compiled your HLS component using Intel Quartus Prime v18.1. If you run it again, it should work.

1. Load the AF into the FPGA:

```
# sudo fpgaconf hls_afu.gbs
```

2. Navigate to the `hls_afu/sw` directory.
3. Build and run the host application (do not use 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
```

3. Changing Components in the HLS AFU Design Example

You can substitute a different HLS accelerator into this AFU to create your own HLS accelerators. This procedure replaces the `fpVectorReduce_float` component with the `fpVectorReduce_ac_int` component.

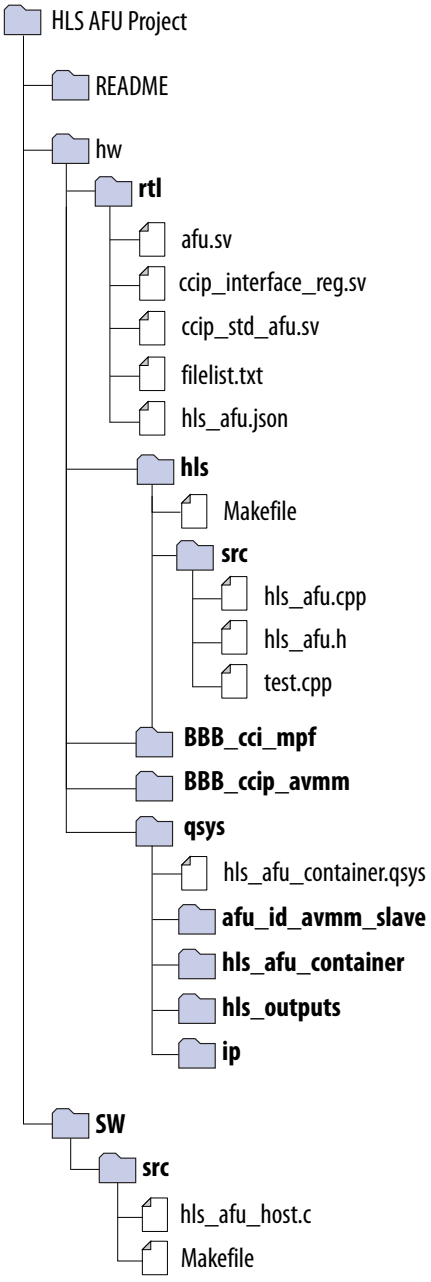
BEFORE YOU BEGIN

These steps are not necessary for this task, as Intel provide a replacement HLS component.

1. Create an HLS component that meets the HLS AFU Avalon-MM I/O Slave and HLS AFU Avalon-MM master interface requirements. Refer to HLS AFU Avalon-MM I/O Slave Interface on page 27 and HLS AFU Avalon-MM Master Interfaces on page 27
2. Generate RTL with the i++ command.
3. Create a new UUID for your new AFU. Refer to the *Accelerator Functional Unit (AFU) Developer's Guide*.

Figure 5. Directory Structure and Files

This figure shows only the files you may edit.



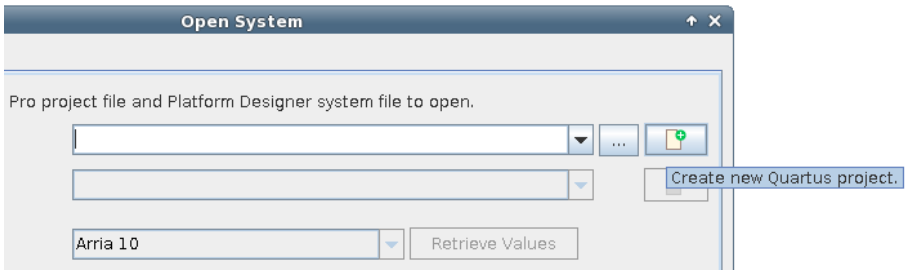
STEPS

1. Navigate to the `qsys` folder and open the system with Platform Designer v17.1.1.

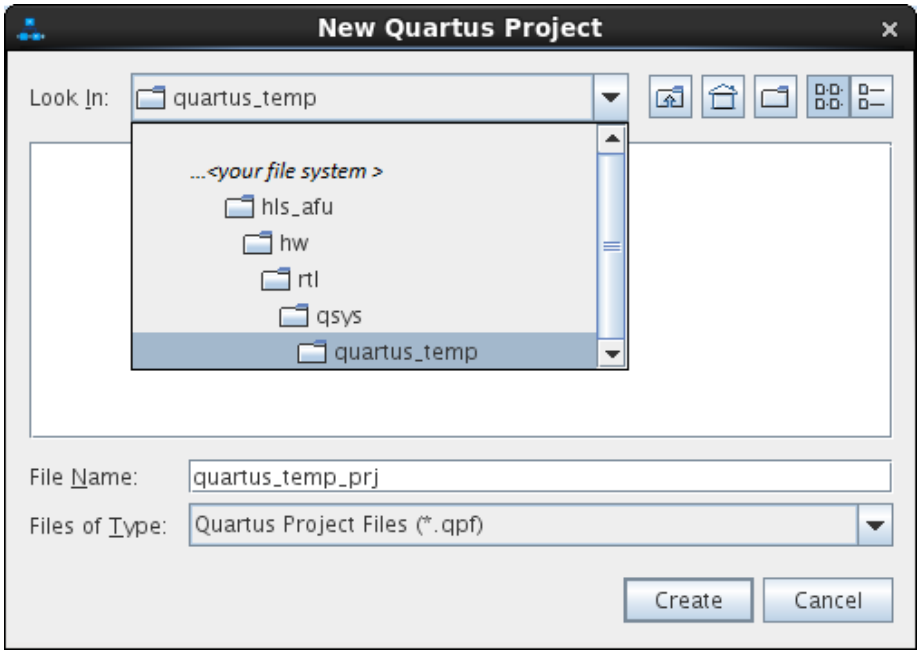
   ```
   $ qsys-edit hls_afu_container.qsys
   ```

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

Figure 6. New Quartus Project

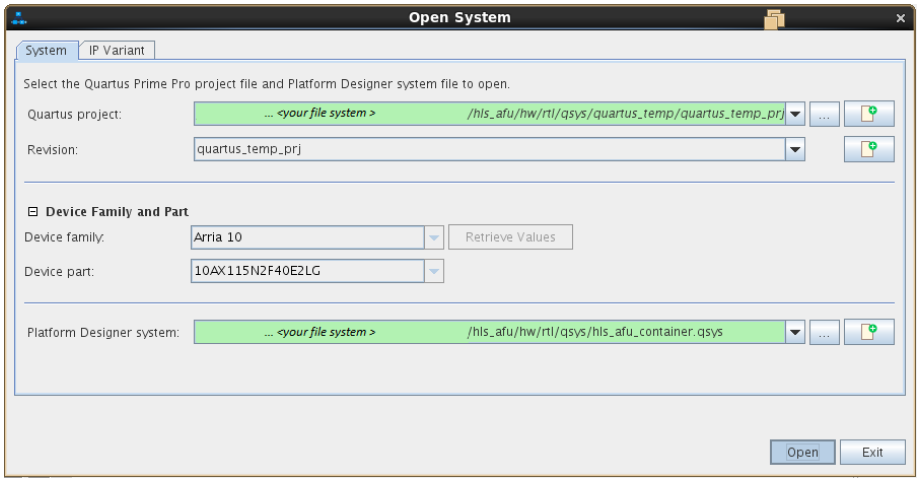


3. Create a `quartus_temp_prj` project in a folder called `quartus_temp`.



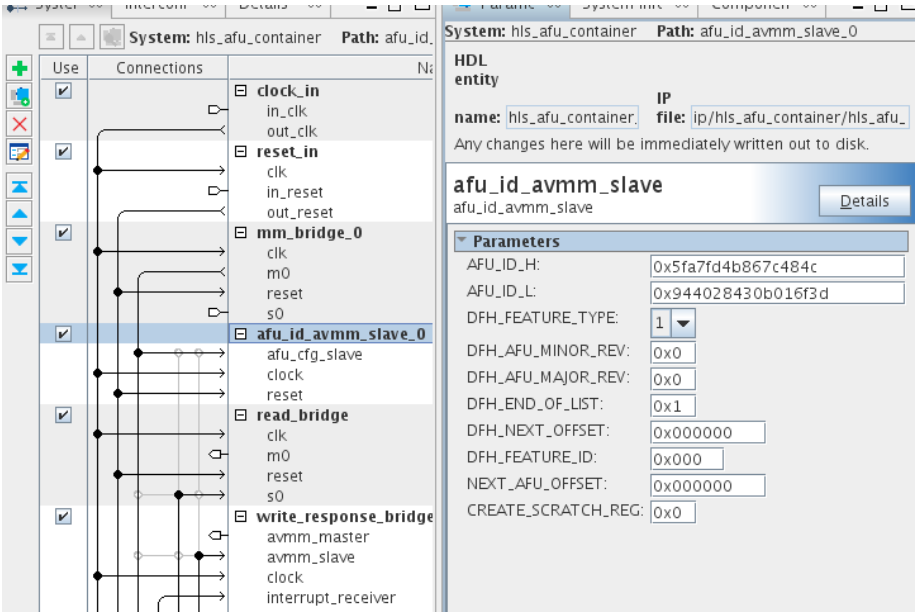
4. Click **Open** on the **Open System** window.

Figure 7. Open System



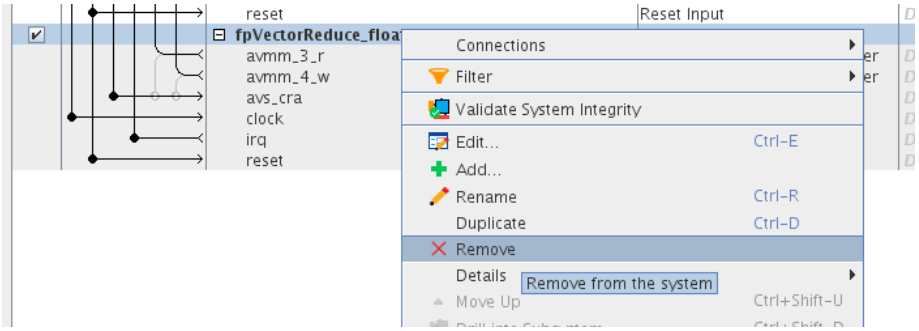
5. If you created a new UUID for your component, enter it in the **afu_id_avmm_slave** component.

Figure 8. UUID



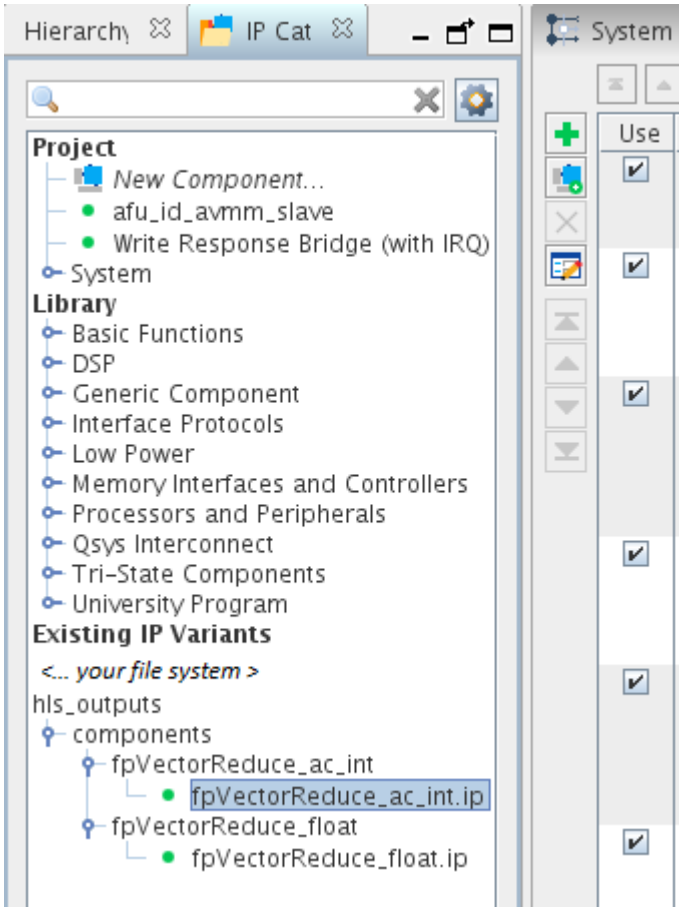
6. Remove the **fpVectorReduce_float** component by right-clicking it and clicking **Remove**.

Figure 9. Remove Component



7. In the **Delete IP Variant** dialog, click **No** so that Platform Designer does not delete the **fpVectorReduce_float** IP file.
8. Add the **fpVectorReduce_ac_int** component by double clicking it in **IP Catalog**.

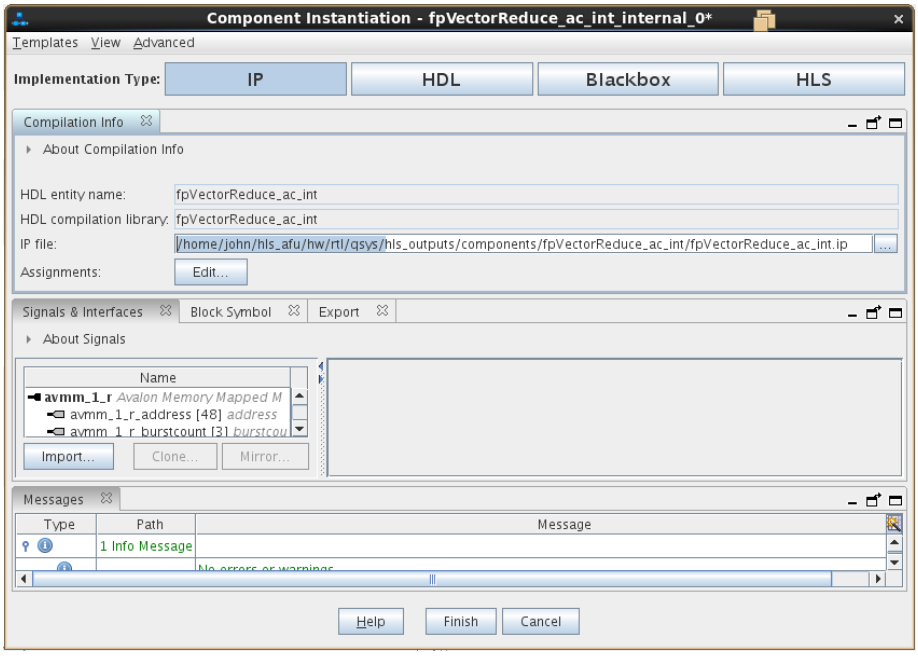
Figure 10. Add Component



9. In the **Component Instantiation** window, delete all the text preceding the `hls_outputs` folder (including the leading '/').

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

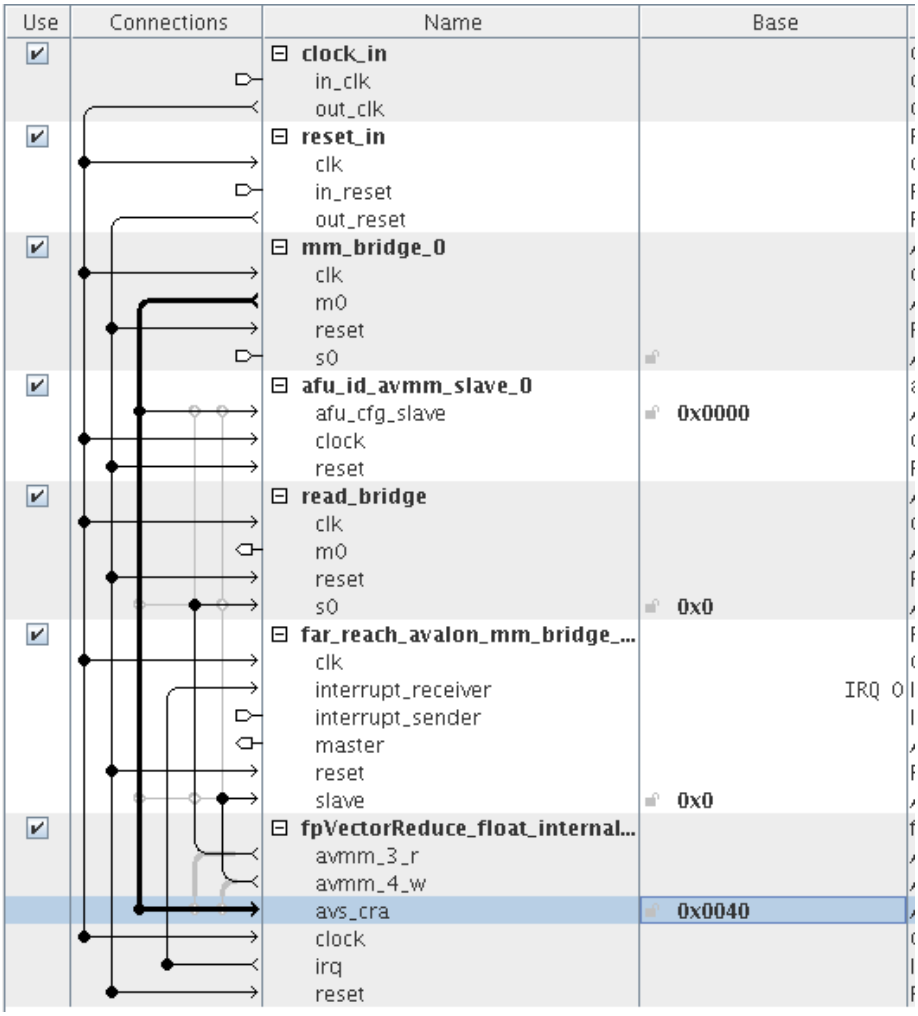
Figure 11. Relative Path



10. Connect up your HLS component.

    Make sure that you set the base address of your component's Avalon-MM slave to `0x0040`, so 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



11. Click **Sync System Info** to update the Platform Designer system. A message appears at the bottom of the Platform Designer window indicating success.
12. Configure your AFU by modifying the Acceleration Stack configuration files `filelist.txt` and `hls_afu.json`.

    Refer to *Acceleration Stack Configuration Files filelist.txt and hls_afu.json*, for examples of these files.
13. Modify the host application to include the Avalon-MM slave register map that is produced by HLS.
    a. Copy the `test-fpga.prj/components/fpVectorReduce_ac_int/fpVectorReduce_ac_int_csr.h` header file to your host application project.
    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.

WHAT TO DO NEXT

1. Generate the ASE testbench. Refer to *Generating the ASE Testbench.*
2. Compile the AF bitstream. Refer to *Compiling the AF Bitstream*.

Related Information

- HLS AFU Avalon-MM I/O Slave Interface on page 27
- HLS AFU Avalon-MM Master Interfaces on page 27
- Compiling and Simulating the HLS Component with the i++ Command on page 6
- Generating a Platform Designer Container for the HLS Component on page 7
- Generating the ASE Testbench on page 9
- Compiling the AF Bitstream on page 13
- Accelerator Functional Unit (AFU) Developer's Guide

3.1. Acceleration Stack Configuration Files filelist.txt and hls_afu.json

The examples refer to the HLS AFU design example.

filelist.txt

The `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+qsys/hls_outputs/components/fpVectorReduce_float
+incdir+qsys/hls_outputs/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.json

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

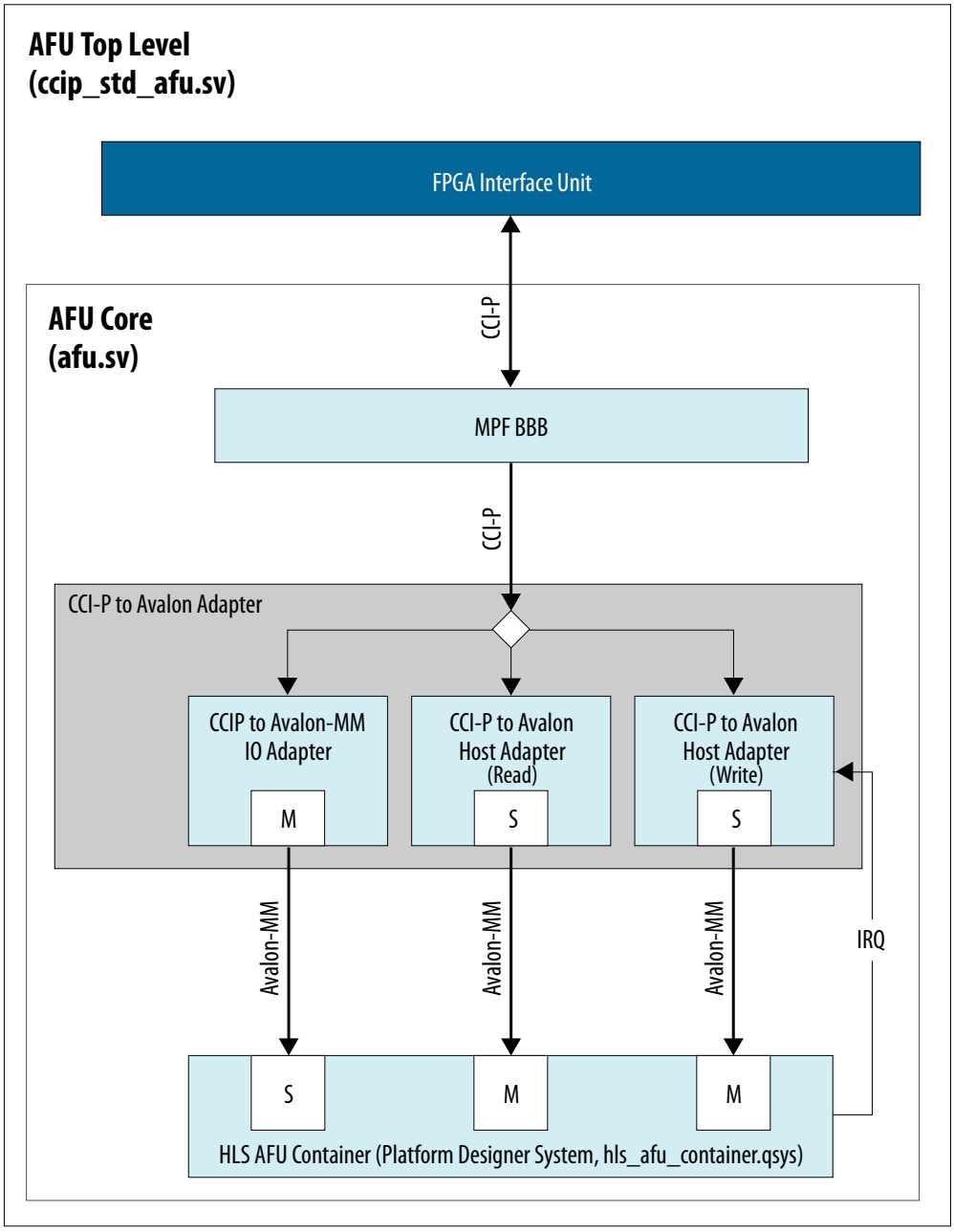
The HLS AFU design comprises an AFU (contained in the hw folder) that runs on an Intel PAC, and a host application that runs on an Intel Xeon CPU.

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 it needs some RTL to initialize and connect the required hardware components.

This design is based on the DMA AFU design, except that it lacks Avalon-MM master interfaces for communicating with the DRAM banks on the Intel 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 are necessary for connecting an HLS component with an 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



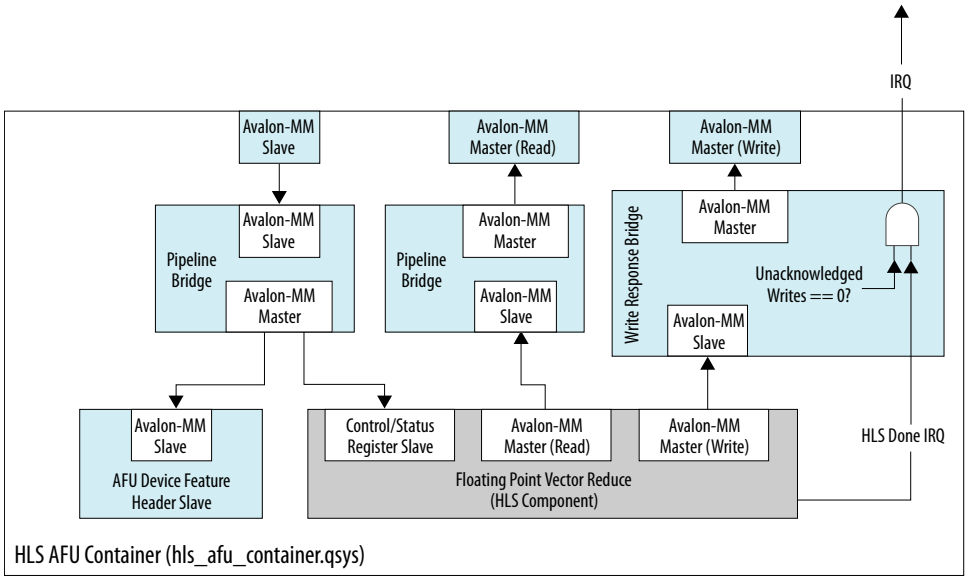
4.1.1. HLS AFU Container

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 (Pipeline bridge)	Allows the slave interfaces of the HLS component and the AFU ID slave to share the same slave 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*)
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 (Pipeline bridge)	Pipelines the HLS component's read only Avalon-MM master interface
write_response_bridge_irq_0 (Write response bridge)	The HLS 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.

Figure 15. Simplified HLS Component

This code contains the basic source code that performs the reduction. It reads a single 32-bit floating-point value each cycle and accumulates the total.

```
1.    component
2.    float floatingPointVectorReduce_basic(float *masterRead,
3.                                          float *masterWrite,
4.                                          int size)
5.    {
6.        float sum = 0.0f;
7.        for (int idx = 0; idx < size; idx++)
8.       {
9.           float readVal = masterRead[idx];
10.          sum += readVal;
11.
12.          masterWrite[idx] = readVal + 1.0f;
13.      }
14.
15.        return sum;
16.   }
```

While this is valid HLS source code, it is insufficient for an AFU design. The Intel Acceleration Stack has specific requirements that dictate how AFUs may access host memory and how the host system sees them. Fortunately, HLS is flexible enough that you can reconfigure your component to meet these constraints. The requirements are:

1. The controls and parameters should be exposed through an Avalon-MM I/O slave interface, not the conduit interfaces that HLS uses by default.
2. AFUs may have two Avalon-MM master interfaces for accessing system 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 moves the `start`, `busy`, `stall`, and `done` control signals into the component's control and status register (CSR). You can also apply the `hls_avalon_slave_register_argument` attribute to each of the component's parameters to move them into the component's CSR. The component's signature is:

```
1. hls_avalon_slave_component
2. component
3. float fpVectorReduce_basic(
4.    hls_avalon_slave_register_argument float *masterRead,
5.    hls_avalon_slave_register_argument float *masterWrite,
6.    hls_avalon_slave_register_argument uint64 size)
```

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

4.1.1.2. HLS AFU Avalon-MM Master Interfaces

AFUs may 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 component's signature and the algorithm.

As the smallest unit of data that an AFU can access in host memory is 64 bytes (512 bits), you need to configure the Avalon-MM master interfaces using HLS `ihc::mm_master` objects. For details about the parameters, refer to the *Intel High-Level Synthesis Compiler Reference Manual*. Then, you need to modify the code Figure 19 on page 30 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's 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 garbage data.

For best performance, you should not attempt read-modify-write operations. Because the design has two separate Avalon-MM master interfaces, the 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:

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

`float` Accesses

The component also allows the HLS compiler to handle the slicing operations for you. The signature is identical to the signature in Figure 19 on page 30, except that the `mm_master` type is float instead of `ac_int<512, false>`.

Figure 16. `float`-based body

```
1.    #pragma unroll 16
2.    for (int itr = 0; itr < 16; itr++)
3.    {
4.        int idx = itr + (loop_idx * 16);
5.        if (idx < size)
6.        {
7.            float readVal = masterRead[idx];
8.            readSum += readVal;
9.            masterWrite[idx] = readVal + 1.0f;
10.       }
11.   }
12.   sum += readSum;
```

Access the `mm_master` inside the unrolled loop body 32 bits at a time, instead of outside the loop body 512 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 18 on page 29.

Figure 17. HLS Report showing float-based component.

Observe the coalesced Avalon-MM master interfaces.

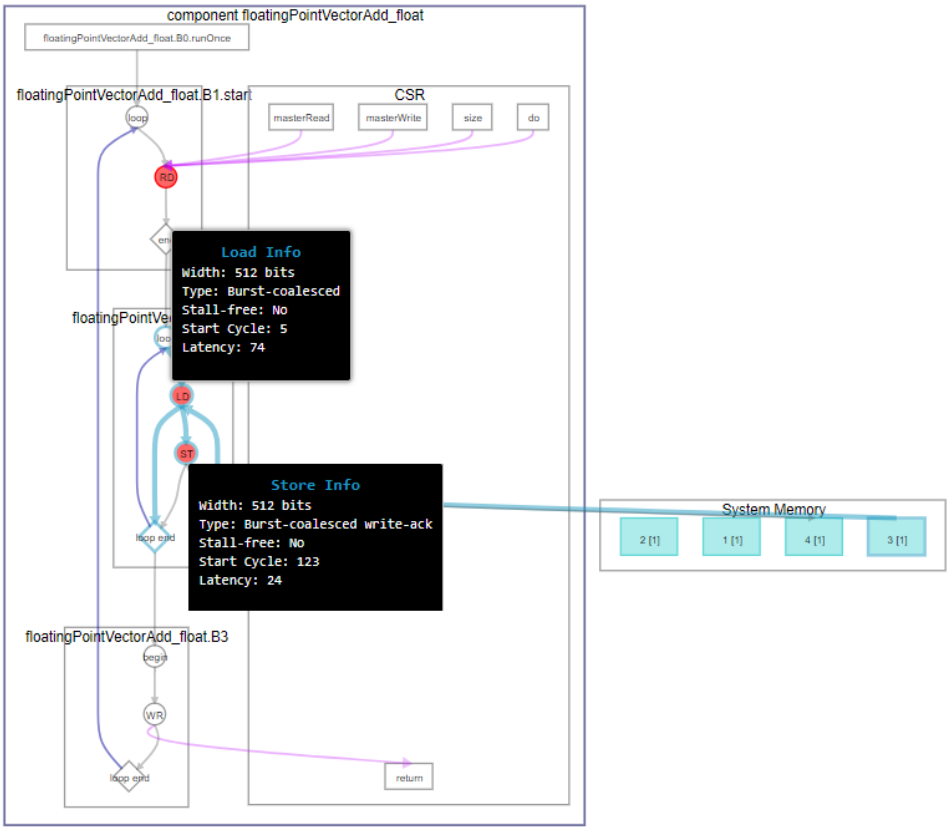
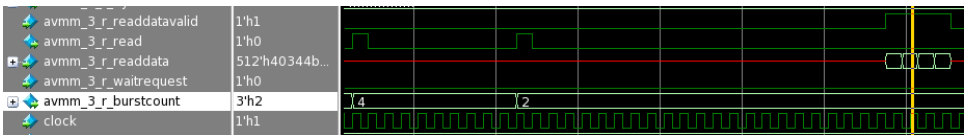


Figure 18. ModelSim waveform showing host memory accesses of `float`-based component



`ac_int` Accesses

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

Figure 19. Signature for `ac_int`-based component

```
1. typedef ac_int<512, false> uint512; // 512-bit unsigned integer
2.    typedef ihc::mm_master<uint512, ihc::dwidth<512>,
3.                           ihc::awidth<48>, ihc::latency<0>,
4.                           ihc::aspace<1>, ihc::readwrite_mode<readonly>,
5.                           ihc::waitrequest<true>, ihc::align<64>,
6.                           ihc::maxburst<4> > MasterReadAcInt;
7.
8.    typedef ihc::mm_master<uint512, ihc::dwidth<512>,
9.                           ihc::awidth<48>, ihc::latency<0>,
10.                          ihc::aspace<2>, ihc::readwrite_mode<writeonly>,
11.                          ihc::waitrequest<true>, ihc::align<64>,
12.                          ihc::maxburst<4> > MasterWriteAcInt;
13.
14.   component
15.   hls_avalon_slave_component
16.   float fpVectorReduce_ac_int(
17.      hls_avalon_slave_register_argument MasterReadAcInt &masterRead,
18.      hls_avalon_slave_register_argument MasterWriteAcInt &masterWrite,
19.      hls_avalon_slave_register_argument uint64_t size);
```

These parameter settings are specific for the Intel Acceleration Stack:

- 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). Readonly and writeonly (one Avalon-MM master of each) (`readwrite_mode` parameter)

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 20. ac_int-based body

```
1.  uint512 readVal = masterRead[loop_idx];
2.    uint512 writeVal = 0;
3.
4.    #pragma unroll 16 // do each loop iteration concurrently.
5.                      // Use 16 iterations because there are 16
6.                      // 32-bit slices in each 512-bit word.
7.    for (int itr = 0; itr < 16; itr++)
8.    {
9.        int idx = itr + (loop_idx * 16);
10.      if (idx < size)
11.      {
12.            // grab a 32-bit piece of the 512-bit value that we read
13.            uint32 readVal_32 = readVal.slc<32>(itr * 32);
14.
15.            // use explicit type casting to process the bits pointed
16.            // to by &readVal_32 as a float.
17.            void *readVal_32_ptr = &readVal_32;
18.            float readVal_f;
19.            float *readVal_f_ptr = &readVal_f;
20.            *readVal_f_ptr = *((float *) readVal_32_ptr);
21.            readSum += readVal_f;
22.
23.            // increment and output
24.            float writeVal_f = readVal_f + 1.0f;
25.
26.            // use explicit type casting to process the bits pointed
27.            // to by &writeVal_f as a uint32.
28.            float *writeVal_f_ptr = &writeVal_f;
29.            uint32 *writeVal_32_ptr = (uint32 *) writeVal_f_ptr;
30.            uint32 writeVal32 = *writeVal_32_ptr;
31.
32.            unsigned int bit_offset = itr * 32;
33.            writeVal.set_slc(bit_offset, writeVal32);
34.      }
35.   }
36.   masterWrite[loop_idx] = writeVal;
```

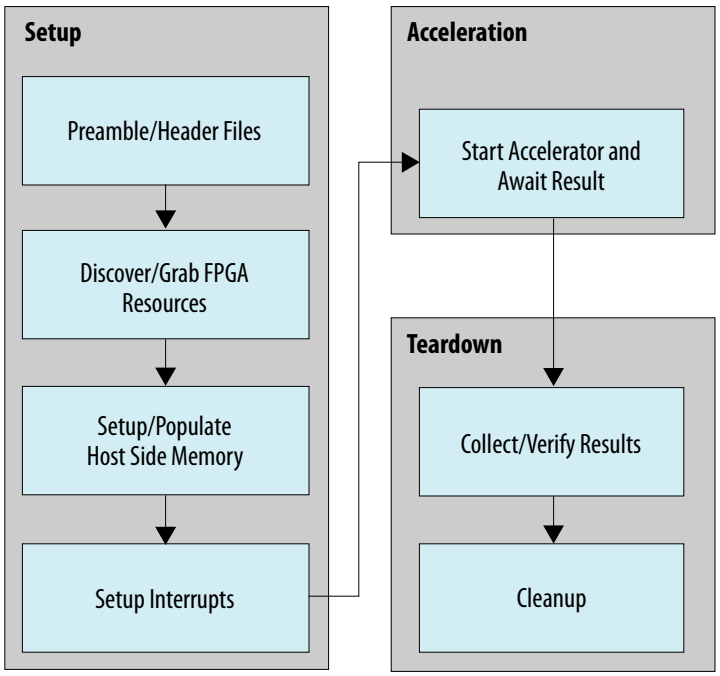
Related Information

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 21. Flow summary of a typical OPAE application



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_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	0x0000 to 0x003F
HLS component	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 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 Results

The host checks that the interrupt returned correctly (or didn't 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. HLS Design Fails to Compile

The HLS code may fail to compile if you are using HLS v18.0 in an environment that does not have the correct version of the GCC libraries. If you are using RHEL 7.x with HLS v18.0, you must install the gcc 4.4.7 compatibility libraries:

```
# sudo yum install compat-gcc-44 compat-gcc-44-c++
```

Then, you must override the *CPLUS_INCLUDE_PATH* environment variable to have i++ prioritize gcc 4.4.7.

```
$ export CPLUS_INCLUDE_PATH=/usr/lib/gcc/x86_64-redhat-linux/4.4.7:/usr/
include/c++/4.4.7:/usr/include/c++/4.4.7/x86_64-redhat-linux
```

5.2. Platform Designer Opens with an Error

If you open the Platform Designer system, you may see this 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. You can also avoid this error 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).

5.3. 'design unit not found' Errors During Make Sim

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
```

5.4. 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).

5.5. Compilation Errors During ASE Testbench Generation

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

Related Information

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

5.6. Incorrect Output During Simulation

ASE saves all waveforms from your HLS AFU design, so you can use these to debug your design.Refer to section 1.4.1.1 of the *Intel Acceleration Stack Quick Start Guide for Intel ® Programmable Acceleration Card with Intel ® Arria ® 10 GX FPGA*, for more information

Related Information

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

5.7. 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

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

Document Version	Changes
2019.03.12	Corrected *Correct Directory Structure* figure.
2019.01.31	Corrected code: `$ $OPAE_PLATFORM_ROOT/bin/run.sh`
2018.11.30	Initial release.