
# HDK for AWS EC2 F1

Revision 0.8

# 1 Revision History

| Revision | Data | Content |
|---|---|---|
| 0.1 | July 8th 2016 | • Initial release for partners based on VU190 and without HMC/InterFPGA links |
| 0.2 | August 24th 2016 | • Added init script that must be run before simulation (see section 6.1)<br>• PCIe BFM restriction removed<br>• Added VU190 Board example bitfile/tools |
| 0.3 | September 7th 2016 | • Change to Vivado 2016.2 |
| 0.4 | September 16th 2016 | • Fixed location of simulation filelists |
| 0.5 | 9/30/2016 | • Updated cl_simple diagram<br>• Removed limitation of no pcie status |
| 0.6 | 10/14/2016 | • Updates to simulation section |
| 0.7 | 10/18/2016 | • Added pipeline blocks |
| 0.8 | 10/31/2016 | • Build support for VU9P<br>• More implementation notes<br>• Change to Vivado 2016.3 |

# 2 F1 HDK

The HDK Delivery includes the following components:

- RTL
  - Top level RTL
  - HL RTL
  - Sample CL RTL
- Vivado (**version 2016.3 or later only**)
  - TCL based project files
  - DDR/PCIe IP
  - Constraints

The HDK delivery enables both simulation and implementation of the complete HL/CL design.

For release notes and restrictions of this release, please refer to "Release notes" section.

# 3 Directory Structure

The HDK is delivered as a tar ball. The tar ball should be extracted with the command:

```
tar –xvf <tar_file>.tgz
```

There are two main directories at the top level of the HDK package:

- tb – All the verification components (tb stands for testbench)
- top_cl – All of the design components including RTL models of the HL and CL

## 3.1 tb

The tb directory consists of the following sub-directories

- tb/sv – Testbench System Verilog files
- tb/tests – Tests
- tb/scripts – Scripts and filelists to run tests with VCS simulator
- tb/vivado - Scripts and filelists to run tests with Vivado simulator

## 3.2 top_cl (Design)

The design is organized as a Vivado GUI project

- top_cl.srcs/constrs_1/ - Constraints
- top_cl.srcs/sources_1/imports/rtl – All the RTL files, excluding any IP
- top_cl.srcs/sources_1/ip/ - All the Xilinx IP

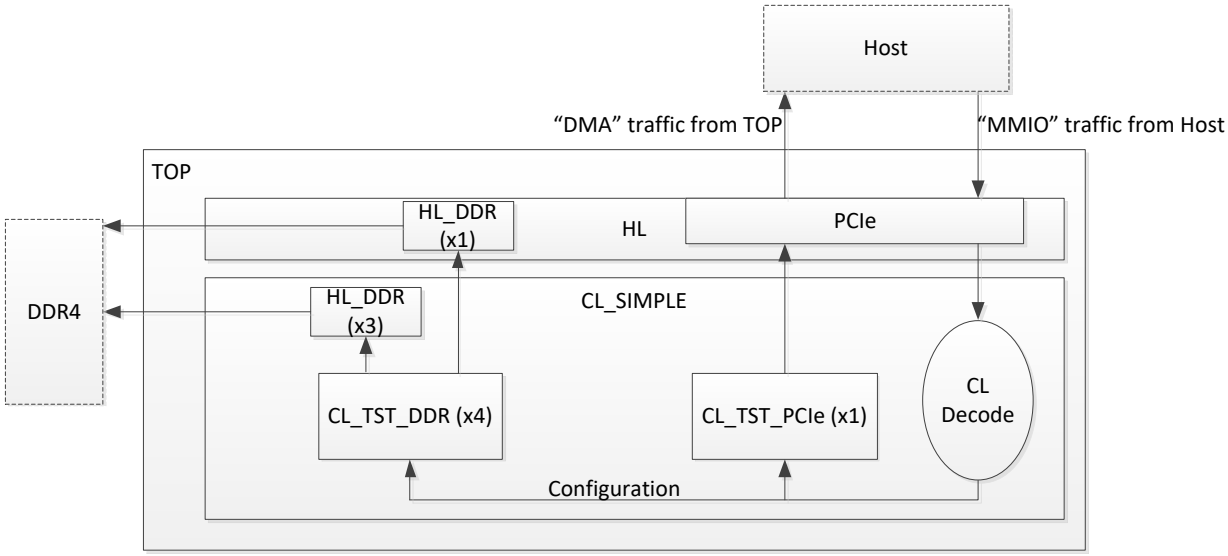
## 3.3 models

The models directory contains simulation models of the external components: PCIe Root Complex model, and DDR4 DIMMs.

# 4 CL Simple Design

The CL Design consists of traffic generators connected to each interface (PCIe/DDR). There is an independent traffic generator per interface (CL_TST). The traffic generator is programmed through MMIO from the Host (through the User PF). Each traffic generator has 256B of address space that is used to configure the traffic generation and read back status. The traffic generators can generate writes and reads, and a read verify function is also implemented. Refer to the RTL file "top_cl/top_cl.srcs/sources_1/imports/rtl/cl/cl_tst.sv" for details on programming the traffic generator.

Below is the block diagram of the design. Note the arrow directions indicate a Master/Slave relationship (direction of arrow is from Master to Slave). All interfaces include both read and write access.




All of the design files are in top_cl.srcs/imports/rtl. All of the RTL except for top and the CL files are encrypted. The top level module is at:

top_cl.srcs/imports/rtl/top/top.sv

The CL files are at:

top_cl.srcs/imports/rtl/cl/

cl_simple.sv – Top level of the CL. Note the module name and file name do not match (module name is CL).

cl_tst.sv – Traffic generator.

## 4.1 Creating a new CL

To create a new CL, create a new file with the same interface ports as cl_simple.sv. The filename can be anything, but the module name must be "cl". This is required for partial reconfiguration.

Note the CL must instantiate hl_ddr (DDR4 controllers).

# 5 Vivado Implementation

The HDK supports implementation on two devices:

- VU190 – Pre-production VU190 cards. These cards have been deprecated. New designs should not use the VU190 flow. The VU190 flow does not support partial reconfiguration, and there are no VU190 cards that can be used in the AWS cloud.
- VU9P – Production cards. These are the cards that are available in the AWS cloud. All designs should use the VU9P implementation flow.

## 5.1 VU190

The HDK supports running Vivado in batch mode. There is a build script "**top_cl/run_build_top_cl.scr**". This calls Vivado in batch mode and runs the TCL script "**top_cl/build_top_cl.tcl**". The script performs all steps from reading files through bitstream generation. Each step is blocked in the TCL script and can be modified as needed including adding constraints, adding/removing reports, changing optimization parameters, etc….

If using build_top_cl.tcl to build a different CL design:

- the variable "$origin_dir" is used to set the root directory to reference all files. This variable can be overridden on the command line by the "--origin_dir <path>" option, or set in the TCL script.
- The "Verilog Global Defines" section should be changed
- The "Include directories" section should be changed to reference the new CL files
- The "Add constraints" section may need to be added to (to add CL specific constraints)

The IP and existing constraints must not be modified (top.xdc/ddr.xdc). Any changes will be ignored during final implementation.

## 5.2 VU9P

The VU9P implementation flow supports the production partial reconfiguration flow. Note a "design checkpoint" (DCP) is a Xilinx file that includes a fully or partially implemented design. The design checkpoint is used to save the state of an implementation, and can later be read in by the Xilinx tools to load the implementation for use, analysis, or to continue with implementation. The DCP can be generated at any point in the flow (synthesis, placement, routing).

Here is an outline of the partial reconfiguration flow:

- AWS supplies the Shell design checkpoint. This is a fully implemented shell design (routed design).
- Developer synthesizes Developer's CL design

- Developer implements the entire design using Developer's CL synthesis checkpoint, and AWS provided Shell design. This produces a complete design checkpoint (CL + SH).
- Developer provides AWS the complete design checkpoint for ingestion

Reference scripts are provided for the flow in the **developer/**. The scripts provided implement the CL_SIMPLE design. The scripts can be modified by the Developer to add Developer specific files and constraints. Refer to **developer/README** for more information.

## 5.3 Implementation notes

High speed logic implementations in the large Ultrascale+ FPGAs require careful attention to placement of logic and RAMs. Modest levels of logic may fail to meet timing due to the distance traversed by a path. The implementation script adds the following to help with timing:

- extra registers are added to span distances between logic sub blocks (pipelining registers)
- synthesis is run with the "-keep_equivalent_registers" option, to prevent Synthesis from optimizing away pipelining registers
- "-directive Explore" is used to do run multiple passes of the optimization
- phys_opt_design is run after placement and routing. This is an optional step that enables physical optimizations such as replication of high fanout nets.

The VU9P device employs Xilinx Stacked Silicon technology with 3 Super Logic Regions (SLRs). Crossing SLRs adds delay to paths, and care should be taken to not have critical paths span SLRs. This can generally be achieved with good design partitioning, registering interfaces between blocks, and adequate signal pipelining between blocks. With good design partitioning the Xilinx tools generally do a good job of containing related logic within a single SLR region. If critical paths span SLRs, the design partitioning should be analyzed.

### 5.3.1 Signal Pipelining

All signals between the SH and CL should be registered at the boundary. AXI-4 interfaces can be pipelined using the provided library component (see AXI-4 Pipelining). Other signals and interfaces can be pipelined using the lib_pipe.sv library component. Generally 4 levels of pipeline is sufficient to pipeline between the SH and the CL logic (refer to the CL_SIMPLE design for examples).

### 5.3.2 AXI-4 Pipelining

There is an AXI-4 Pipelining block (axi4_flop_fifo.sv) that should be used to register the interfaces between the CL and SH. This can also be used inside the CL design to add pipeline stages between user logic and the instantiated interface blocks. Refer to the cl_simple.sv design for details on instantiating the axi4_flop_fifo block. The parameters are:

- IN_FIFO – Set to '1' if CL is slave, otherwise set to '0'
- ADDR_WIDTH – AXI-4 Address width
- DATA_WIDTH – AXI-4 Data width
- ID_WIDTH – AXI-4 ID width (must be at least 1, even if not used)
- A_USER_WIDTH – AXI-4 Address width (must be at least 1, even if not used)
- FIFO_DEPTH – Depth of the FIFO. Recommended value is '3'.

### 5.3.3 Reset

Asynchronous reset propagation to flip flops takes up routing resources in the FPGA and may not be necessary. Generally data path and pipelining flip flops do not need resets, and control path (i.e. state machines) do need resets. FPGAs support initializing flip flops to a particular state when a design is loaded. This is also used to initialize flip flops for simulation. When declaring a flip flop, the initial value can be specified:

```
logic[7:0] my_flop = 8'h00;


always @(posedge clk)
        my_flop <= .....;
```

The reset from the SH should be "flopped' using a reset synchronizer before being used in the CL (refer to CL_SIMPLE). Also judicious use of additional reset synchronizers in sub-blocks of the CL design will help with timing.

# 6 Simulation

The HDK includes a testbench and simple interface tests for each of the interfaces using the CL_TST traffic generator. The testbench is setup to run with the the Vivado simulator or the Synopsys VCS simulator. For other simulators you must change/create scripts to run a simulation.

Note there are two filelists used for simulation (tb/scripts/):

- ddr_files.f – includes all of the DDR model files
- filelist.f – includes everything else, including Xilinx models, PCIe RC model, and all the design files (including Xilinx IP).

## 6.1 Setup

You must run an init script before running any simulations:

- cd top_cl
- ./init.sh

This will create some of the models required for simulation. Note Vivado must be installed an in your $PATH before you run the "init.sh" script.

## 6.2 Mentor Questa Simulation

To run a Questa simulation:

- cd to tb/scripts (note the next 3 steps are the same as the Vivado simulation)
- Open the file Makefile.inc and change the line "export XILINX_VIVADO=…" to point to your Vivado installation directory. This is where the simulation will pull Xilinx simulation models from.

- Optionally change the line “export RUN_DIR = …” to point to where you would like to put the simulation outputs including the log and waves dump.
- Create the Questa library.

  make create_libs_questa
- Run the test: **make <testname_abr>** where testname_abr is the testname without the _test suffix. For example to run the test “cl_pcie_test” :

  make cl_pcie

  Refer to the “Tests” section for a description of the tests.

## 6.3 Other Simulators

To run with another simulator you must generate your own scripts using the filelists provided. Note the filelists use the following environment variables to find the files:

$XILINX_VIVADO – Location of a Vivado installation. Used to get Xilinx simulation models

$DESIGN_DIR – Location of the Design Directory (top_cl/ directory)

$IP_DESIGN_DIR – Location of the AWS design files (top_cl/top_cl.srcs/sources_1/imports/rtl)

$MODELS_DIR – Location of the simulation models directory (models/)

$SCRIPTS_DIR – Location of where the .f files are (ddr_files.f in particular)

The files are encrypted with IEEE P1735 encryption. Tools from the following vendors are supported:

- Xilinx
- Synopsys
- Mentor

Please contact AWS if you require a vendor not listed above.

## 6.4 Tests

The test is selected with a simulator plusarg “+TESTNAME=<testname>”. All testing does some writes and some reads, and the reads verify data returned matches what was written. This is all done by the CL_TST traffic generator block.

There are several tests, here is a breakdown of the categories:

- Each test can be run per interface (pcie, ddr0, ddr1, ddr2, ddr3).
- Each interface can run one of two flavors of test:
  - single – A single write/read
  - iteration – 100 iterations of write/read with incrementing addresses
- Each test can be runs with “incrementing data” (increments per 32-bits)

This yields the following test names (all the permutations):

- cl_pcie_test – Single PCIe, incrementing data
- cl_pcie_iter_test – 100 iterations on PCIe, incrementing data

That group of tests is repeated for ddr0, ddr1, ddr2, ddr3 (substitute pcie for ddrX in the testname).

### 6.5 PCIe Model

The PCIe model is a modified Root Complex model from Xilinx. The simulations use PCIe PIPE-to-PIPE connections (bypass SerDes), to speed up the simulation runtimes.

## 7 VU190 Board Files

There is an example bitfile and tools to verify the VU190 board interfaces are functional. The bitfile directory is:

top_cl/example_bitfile/

The directory contains an example bitfile (build_top_cl.bit.gz) that you can load on the board using the Xilinx JTAG programmer. There are tools (hdk_tools.tar.gz) that can be installed on the server that the card is plugged into to verify the interfaces. The tools are only supported on Linux. Please refer to the README.txt file in the hdk_tools package for information on how to validate the interfaces.

## 8 Release Notes

The current release has the following limitations:

- No HMC – The HMC interfaces are not included in this release
- No inter-FPGA Serial – The inter-FPGA interfaces are not included in this release